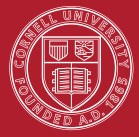


Lecture 21 - Chebyshev Graph Neural Networks

ECE 5260 – ORIE 5735

Anna Scaglione

Cornell Tech

April 21, 2026

Content

Cornell University

Outline

Cornell University

From graph structure to graph spectral priors

- A graph signal is a vector $\boldsymbol{x} \in \mathbb{R}^N$ whose entries live on the nodes of a graph.
- A graph shift operator $\mathbf{S}$ (e.g., adjacency or Laplacian) encodes local structure.
- A graph filter is a polynomial in $\mathbf{S}$ $\quad \rightarrow \mathcal{H}(\mathbf{S}) = \sum_{k=0}^{K} h_k \mathbf{S}^k$.
- If $\mathbf{S} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{-1}$, then $\mathbf{U}$ defines the graph Fourier basis $\rightarrow \tilde{\boldsymbol{x}} = \mathbf{U}^{-1}\boldsymbol{x}$.
- For Laplacian-based GSOs, low eigenvalues correspond to smooth graph modes, high eigenvalues to oscillatory modes.

Frequent prior

Frequent modeling idea

Many graph signals of interest are approximately *low-pass*: most of their energy is concentrated in a few low-frequency graph Fourier modes.

- Social opinion dynamics
- Financial Networks
- Many infrastructure networks (power, water, gas)
- Epidemic

What shallow graph methods assume

- In the previous lecture, we saw that many shallow methods rely on the same spectral prior:

$$\boldsymbol{x} \approx \mathbf{U}_{\mathcal{K}} \boldsymbol{\alpha},$$

  where $\mathbf{U}_{\mathcal{K}}$ contains a small number of low-frequency eigenvectors.

- This viewpoint explains why graph data often look approximately low-rank:

$$\widehat{\mathbf{C}}_x \approx \mathbf{U}_{\mathcal{K}} \, \boldsymbol{\Sigma}_{\mathcal{K}} \, \mathbf{U}_{\mathcal{K}}^{\top}.$$

- It also motivates several unsupervised or weakly supervised tasks:
  - interpolation from partial node observations,
  - sampling and recovery,
  - graph topology learning from observed signals,
  - community detection,
  - anomaly detection.

What shallow graph methods do

Interpretation

Shallow methods do not learn the representation from scratch: they impose a structural prior derived from the graph spectrum.

Why move beyond shallow methods?

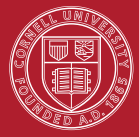
- Shallow methods are powerful when the prior is known in advance:
  - smoothness,
  - low rank / bandlimitedness,
  - diffusion structure,
  - sparsity of anomalies.
- But they are limited when the input–output relation is task-dependent, nonlinear, or requires multiple learned features.
- The next step is to keep the same locality bias induced by the graph, but *learn* the relevant graph filters from data.

Bridge to GNNs

A GNN can be viewed as a composition of *learned local graph filters* and pointwise nonlinearities:

$$\text{graph prior} \quad \Longrightarrow \quad \text{learned graph representation.}$$

Outline



Cornell University

Empirical Risk Minimization

Cornell University

In Empirical Risk Minimization, we are given

- A training set $\mathcal{T}$ containing observation pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{T}$.
- A loss function $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to evaluate the similarity between $\boldsymbol{y}$ and an estimate $\hat{\boldsymbol{y}}$.
- A function class $\mathcal{C}$

Learning as an optimization problem:

Learning means finding function $\Phi^* \in \mathcal{C}$ that minimizes loss $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ averaged over training set

$$\phi^* = \underset{\phi \in \mathcal{C}}{\operatorname{argmin}} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}))$$

We use $\Phi^*(\boldsymbol{x})$ to estimate outputs $\boldsymbol{y} = \Phi(\boldsymbol{x})$ when inputs $\boldsymbol{x}$ are observed but outputs $\boldsymbol{y}$ are unknown

Empirical Risk Minimization

In Empirical Risk Minimization, we are given

- A training set $\mathcal{T}$ containing observation pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{T}$.
- A loss function $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to evaluate the similarity between $\boldsymbol{y}$ and an estimate $\hat{\boldsymbol{y}}$.
- A function class $\mathcal{C}$

Learning as an optimization problem:

Learning means finding function $\boldsymbol{\Phi}^* \in \mathcal{C}$ that minimizes loss $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ averaged over training set

$$\phi^* = \underset{\phi \in \mathcal{C}}{\operatorname{argmin}} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}))$$

We use $\Phi^*(\boldsymbol{x})$ to estimate outputs $\boldsymbol{y} = \Phi(\boldsymbol{x})$ when inputs $\boldsymbol{x}$ are observed but outputs $\boldsymbol{y}$ are unknown

Learning with a Graph Convolutional Filter

- Input / output signals $\boldsymbol{x}$ / $\boldsymbol{y}$ are graph signals supported on a common graph with shift operator $\mathbf{S}$

- For graph filters of order K supported on $\mathbf{S}$:

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

- Learn ERM solution restricted to graph filter class

$$\boldsymbol{h}^* = \underset{\boldsymbol{h}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}))$$

The optimization is over filter coefficients $\boldsymbol{h}$ with the graph shift operator $\mathbf{S}$ given.

Learning with a Graph Convolutional Filter

- Input / output signals $\boldsymbol{x}$ / $\boldsymbol{y}$ are graph signals supported on a common graph with shift operator $\mathbf{S}$
- For graph filters of order K supported on $\mathbf{S}$:

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

- Learn ERM solution restricted to graph filter class

$$\boldsymbol{h}^* = \underset{\boldsymbol{h}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}))$$

The optimization is over filter coefficients $\boldsymbol{h}$ with the graph shift operator $\mathbf{S}$ given.

Graph Neural Networks (GNNs)

- A point-wise nonlinearity is a nonlinear function applied component-wise.

$$\sigma[\boldsymbol{x}] = \sigma \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_n) \end{bmatrix} \tag{1}$$

- A pointwise nonlinearity is the simplest nonlinear function we can apply to a vector. Typical choices are:
  - **ReLU**: $\sigma(x) = \max(0, x)$.
  - **Hyperbolic tangent**: $\sigma(x) = \left(e^{2x} - 1\right) / \left(e^{2x} + 1\right)$.
  - **Absolute value**: $\sigma(x) = |x|$.

Graph Neural Networks (GNNs)

- A point-wise nonlinearity is a nonlinear function applied component-wise.

$$\sigma[\boldsymbol{x}] = \sigma \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_n) \end{bmatrix} \tag{1}$$

- A pointwise nonlinearity is the simplest nonlinear function we can apply to a vector. Typical choices are:
  - **ReLU**: $\sigma(x) = \max(0, x)$.
  - **Hyperbolic tangent**: $\sigma(x) = \left(e^{2x} - 1\right) / \left(e^{2x} + 1\right)$.
  - **Absolute value**: $\sigma(x) = |x|$.

Graph Neural Networks (GNNs)

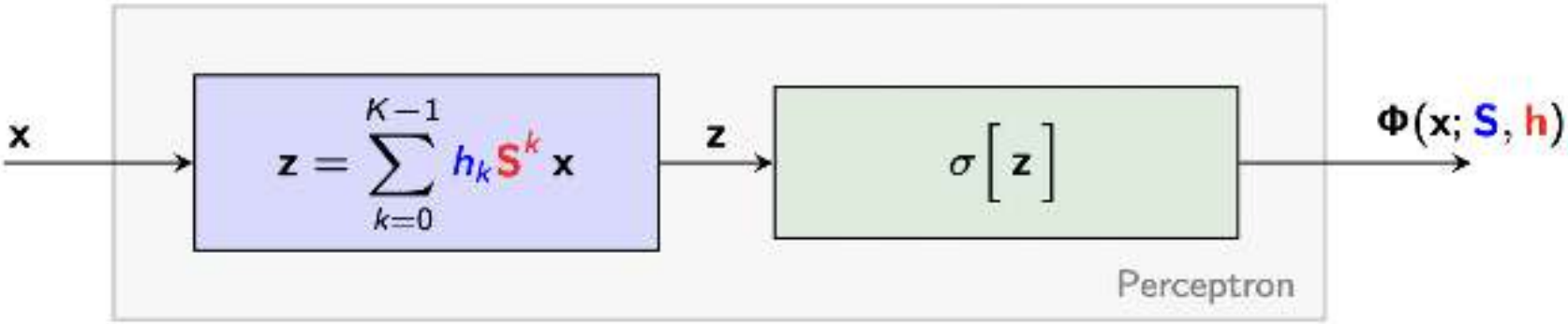
- A point-wise nonlinearity is a nonlinear function applied component-wise.

$$\sigma[\boldsymbol{x}] = \sigma \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_n) \end{bmatrix} \tag{1}$$

- A pointwise nonlinearity is the simplest nonlinear function we can apply to a vector. Typical choices are:
  - **ReLU**: $\sigma(x) = \max(0, x)$.
  - **Hyperbolic tangent**: $\sigma(x) = \left(e^{2x} - 1\right) / \left(e^{2x} + 1\right)$.
  - **Absolute value**: $\sigma(x) = |x|$.

Graph Neural Networks (GNNs)

- Graph filters have limited expressive power because they can only learn linear maps
- A first approach to nonlinear maps is the graph perceptron



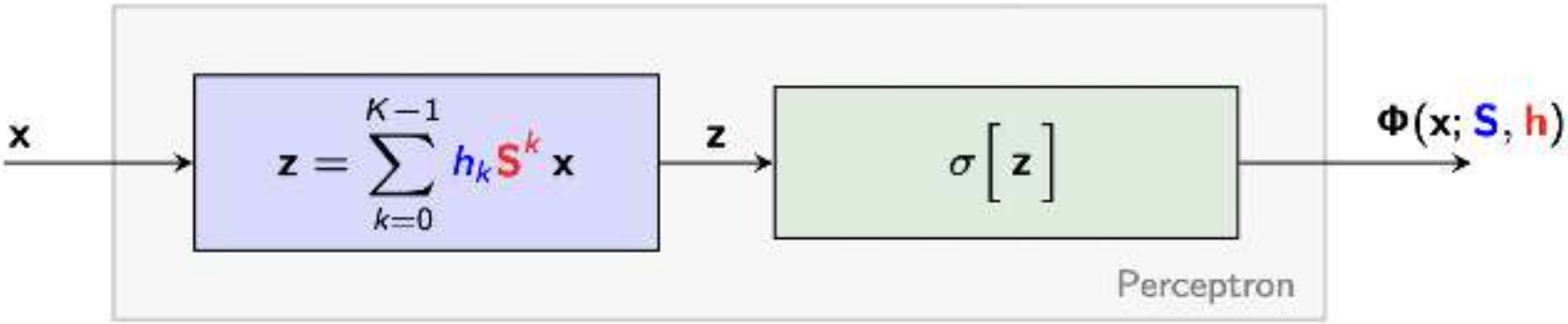
- The optimal regressor restricted to perceptron class is:

$$\boldsymbol{h}^* = \underset{\boldsymbol{h}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}))$$

- Perceptron allows learning of nonlinear maps that are more expressive.

Graph Neural Networks (GNNs)

- Graph filters have limited expressive power because they can only learn linear maps
- A first approach to nonlinear maps is the graph perceptron



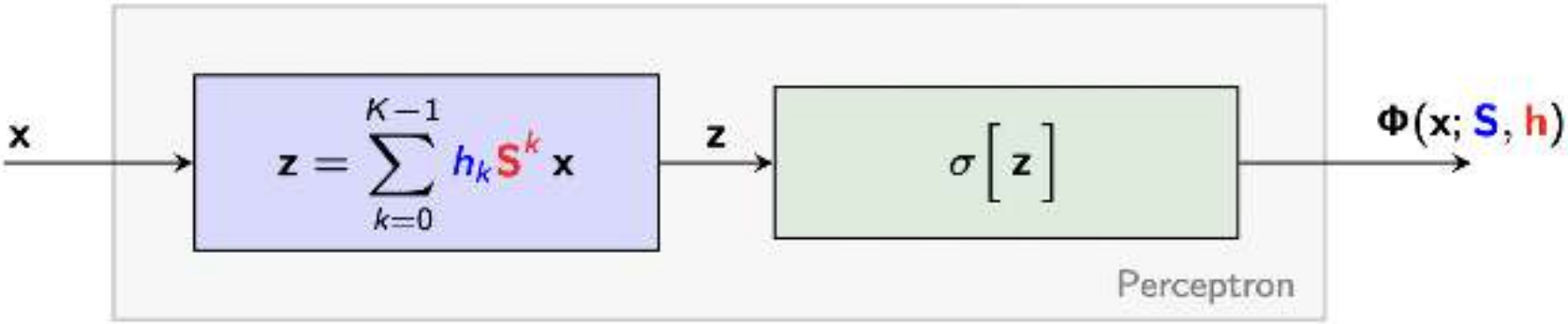
- The optimal regressor restricted to perceptron class is:

$$\boldsymbol{h}^* = \underset{\boldsymbol{h}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}))$$

- Perceptron allows learning of nonlinear maps that are more expressive.

Graph Neural Networks (GNNs)

- Graph filters have limited expressive power because they can only learn linear maps
- A first approach to nonlinear maps is the graph perceptron



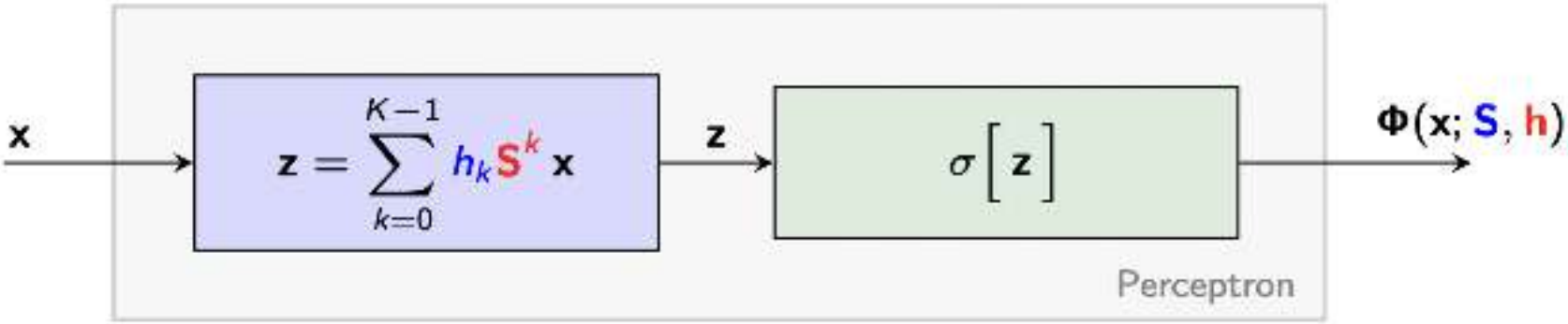
- The optimal regressor restricted to perceptron class is:

$$\boldsymbol{h}^* = \underset{\boldsymbol{h}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}))$$

- Perceptron allows learning of nonlinear maps that are more expressive.

Graph Neural Networks (GNNs)

- Graph filters have limited expressive power because they can only learn linear maps
- A first approach to nonlinear maps is the graph perceptron



- The optimal regressor restricted to perceptron class is:

$$\boldsymbol{h}^* = \underset{\boldsymbol{h}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathcal{L}(\boldsymbol{y}, \Phi(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}))$$

- Perceptron allows learning of nonlinear maps that are more expressive.

Graph Neural Networks (GNNs)

- Layer ℓ processes its input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\boldsymbol{h}_\ell = [h_{\ell 0}, \ldots, h_{\ell,K-1}]$ to produce output $\boldsymbol{x}_\ell$.

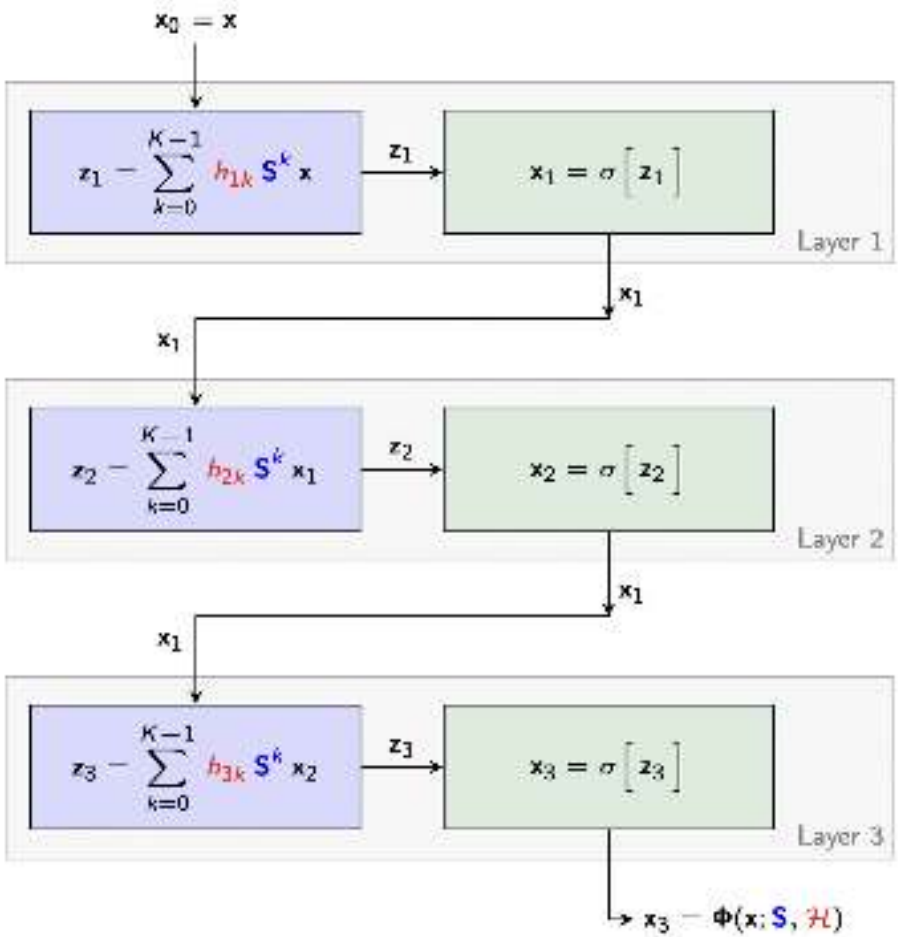
$$\boldsymbol{x}_\ell = \sigma\left[\boldsymbol{z}_\ell\right] = \sigma\left[\sum_{k=0}^{K-1} h_{\ell k} \mathbf{S}^k \boldsymbol{x}_{\ell-1}\right]$$

- If it has L layers, the GCN output

$$\Rightarrow \boldsymbol{x}_L = \overbrace{\Phi\left(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_L\right)}^{=\Phi(\boldsymbol{x};\mathbf{S},\mathcal{H})}$$

- The trainable parameter is filter tensor $\mathcal{H} = [h_1, \ldots, h_L]$. **The GSO is prior information.**

Example: $\mathcal{H} = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$



Graph Neural Networks (GNNs)

- Layer ℓ processes its input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\boldsymbol{h}_\ell = [h_{\ell 0}, \ldots, h_{\ell,K-1}]$ to produce output $\boldsymbol{x}_\ell$.

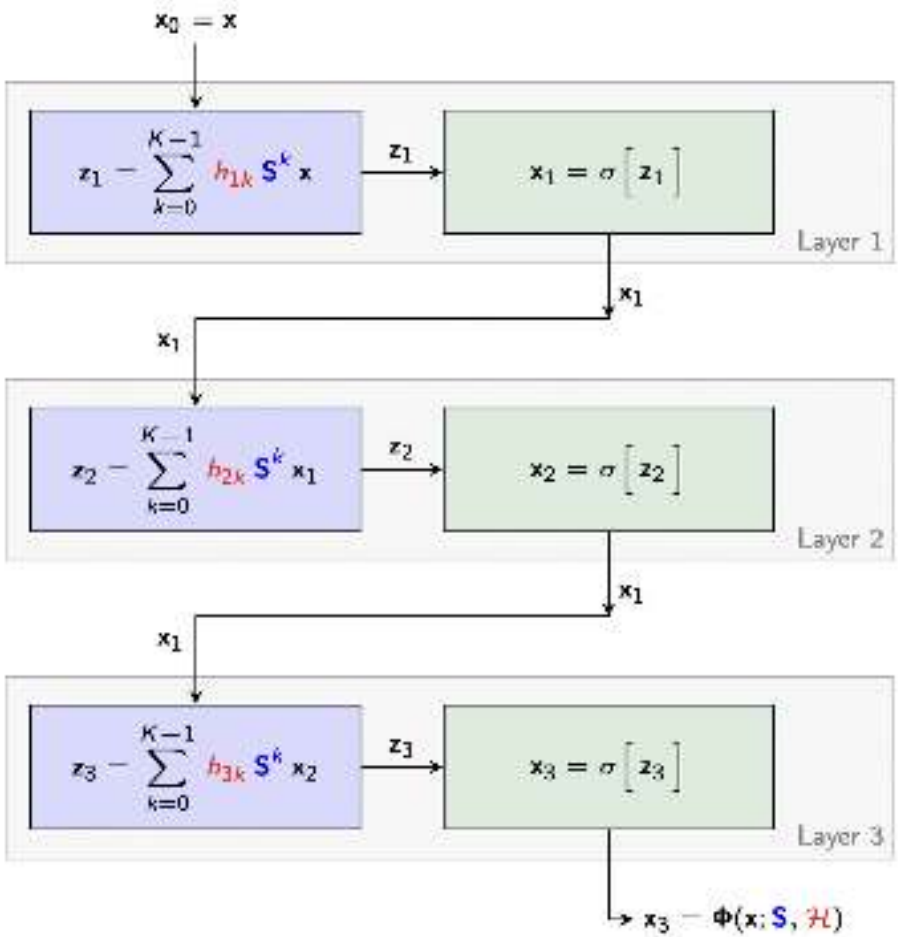
$$\boldsymbol{x}_\ell = \sigma\left[\boldsymbol{z}_\ell\right] = \sigma\left[\sum_{k=0}^{K-1} h_{\ell k} \mathbf{S}^k \boldsymbol{x}_{\ell-1}\right]$$

- If it has L layers, the GCN output

$$\Rightarrow \boldsymbol{x}_L = \overbrace{\Phi\left(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_L\right)}^{=\Phi(\boldsymbol{x};\mathbf{S},\mathcal{H})}$$

- The trainable parameter is filter tensor $\mathcal{H} = [h_1, \ldots, h_L]$. **The GSO is prior information.**

Example: $\mathcal{H} = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$



Graph Neural Networks (GNNs)

- Layer ℓ processes its input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\boldsymbol{h}_\ell = [h_{\ell 0}, \ldots, h_{\ell,K-1}]$ to produce output $\boldsymbol{x}_\ell$.

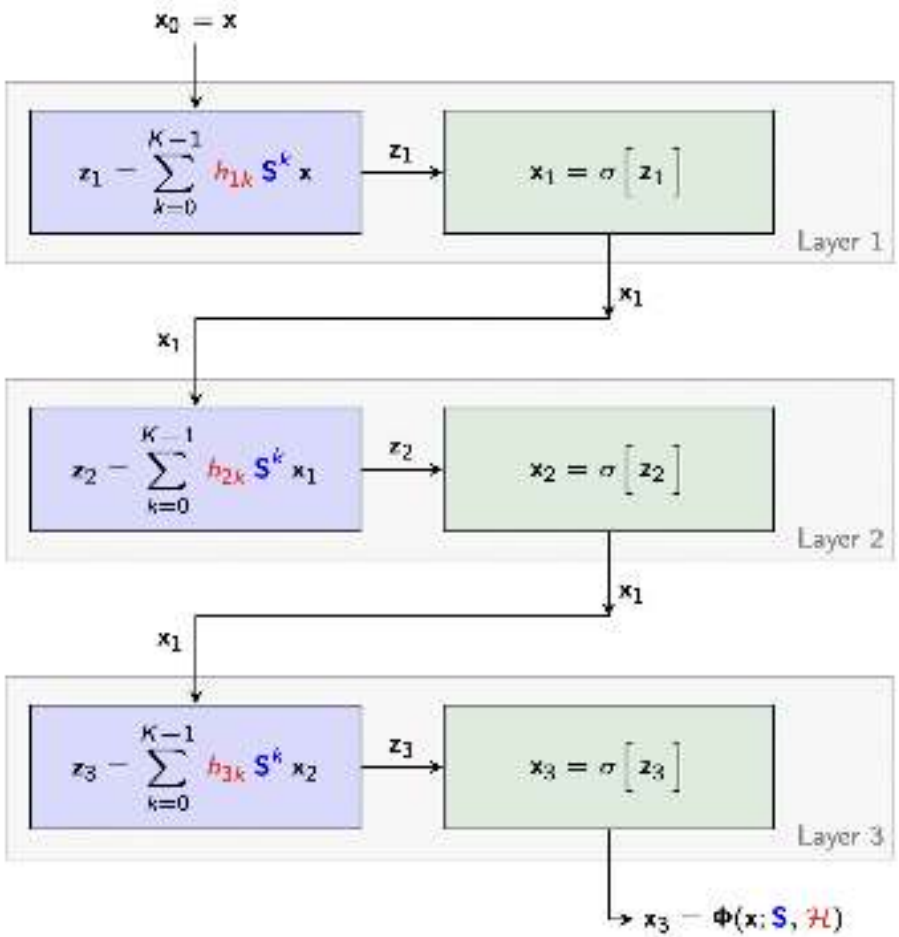
$$\boldsymbol{x}_\ell = \sigma\left[\boldsymbol{z}_\ell\right] = \sigma\left[\sum_{k=0}^{K-1} h_{\ell k} \mathbf{S}^k \boldsymbol{x}_{\ell-1}\right]$$

- If it has L layers, the GCN output

$$\Rightarrow \boldsymbol{x}_L = \overbrace{\Phi\left(\boldsymbol{x}; \mathbf{S}, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_L\right)}^{=\Phi(\boldsymbol{x};\mathbf{S},\mathcal{H})}$$

- The trainable parameter is filter tensor $\mathcal{H} = [\boldsymbol{h}_1, \ldots, \boldsymbol{h}_L]$. **The GSO is prior information.**

Example: $\mathcal{H} = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$



GCN vs Fully Connected NN

- Since GCNs are special cases of fully connected NNs, the latter architecture can attain a smaller cost

$$\min_{\mathcal{H}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \ell(\Phi(\boldsymbol{x};\mathcal{H}),\boldsymbol{y}) \leq \min_{\mathcal{H}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \ell(\Phi(\boldsymbol{x};\mathbf{S},\mathcal{H}),\boldsymbol{y})$$

- The fully connected NN does better. But this holds for the training set

- In practice, GNNs do better because it generalizes better to unseen signals, just like CNN do, as it exploits internal symmetries of graph signals codified in the graph shift operator $\mathbf{S}$

GCN vs Fully Connected NN

- Since GCNs are special cases of fully connected NNs, the latter architecture can attain a smaller cost

$$\min_{\mathcal{H}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \ell(\Phi(\boldsymbol{x};\mathcal{H}),\boldsymbol{y}) \leq \min_{\mathcal{H}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \ell(\Phi(\boldsymbol{x};\mathbf{S},\mathcal{H}),\boldsymbol{y})$$

- The fully connected NN does better. But this holds for the training set

- In practice, GNNs do better because it generalizes better to unseen signals, just like CNN do, as it exploits internal symmetries of graph signals codified in the graph shift operator $\mathbf{S}$

GCN vs Fully Connected NN

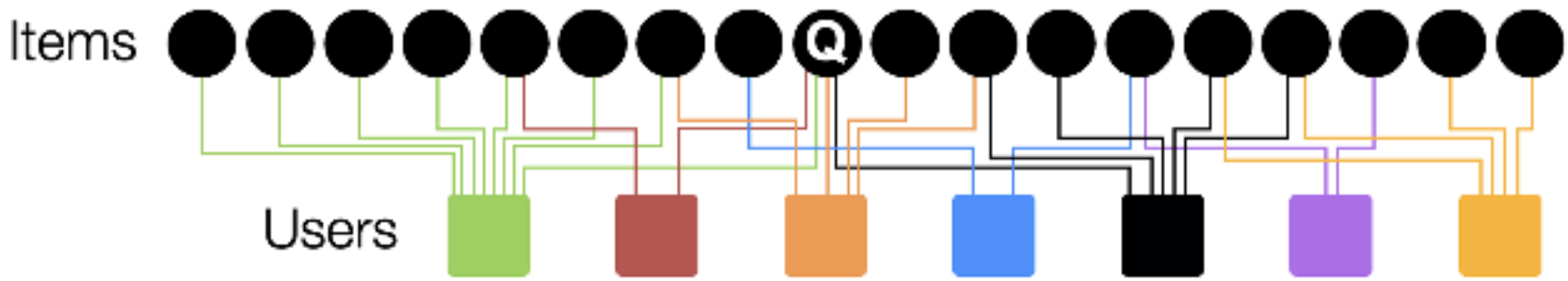
- Since GCNs are special cases of fully connected NNs, the latter architecture can attain a smaller cost

$$\min_{\mathcal{H}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \ell(\Phi(\boldsymbol{x};\mathcal{H}),\boldsymbol{y}) \leq \min_{\mathcal{H}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \ell(\Phi(\boldsymbol{x};\mathbf{S},\mathcal{H}),\boldsymbol{y})$$

- The fully connected NN does better. But this holds for the training set

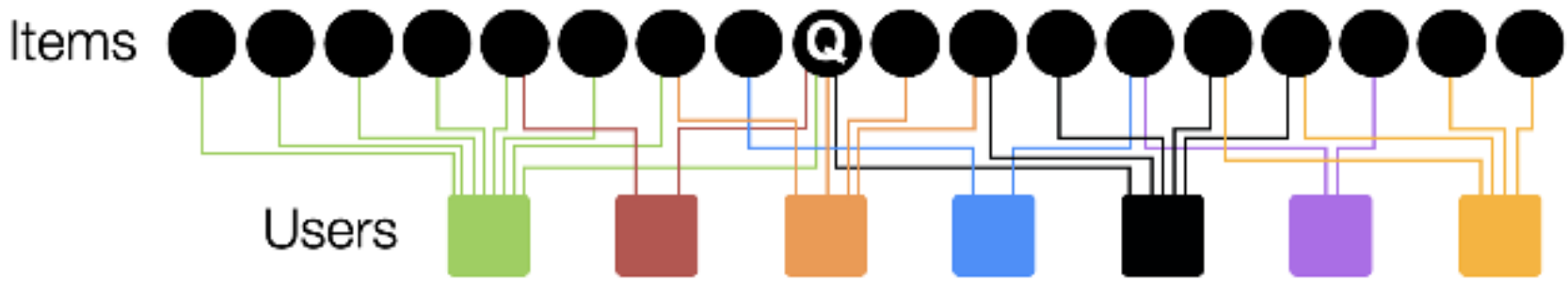
- In practice, GNNs do better because it generalizes better to unseen signals, just like CNN do, as it exploits internal symmetries of graph signals codified in the graph shift operator $\mathbf{S}$

Example: Recommendation Systems



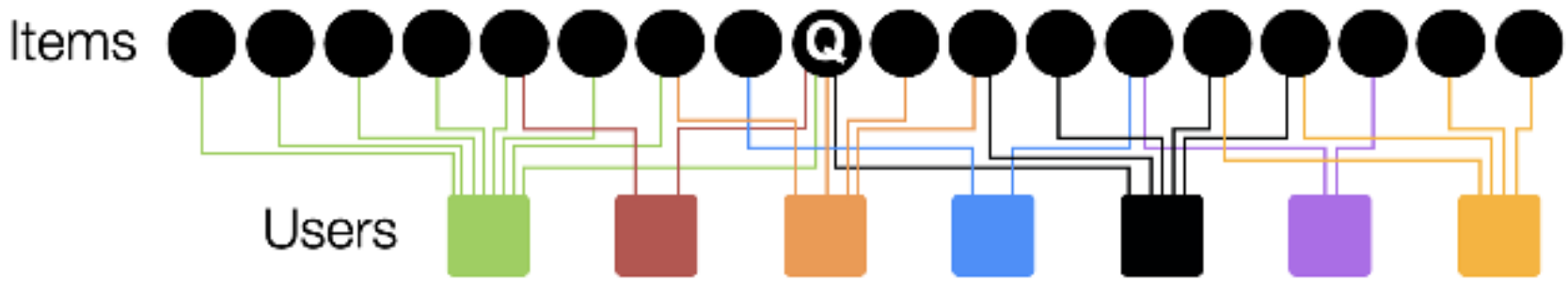
- We will talk about it more next week, but this provides a good example while GNN can do better than NN.
- Above we have the bipartite graph of users and products for a recommendation system.
- The users provided ratings for some of the items they selected.
- We can project this graph onto a user graph whose nodes $\mathcal{V}$ are the users and two users (v, u) are connected by an edge if they buy the same items more times than a given threshold.

Example: Recommendation Systems



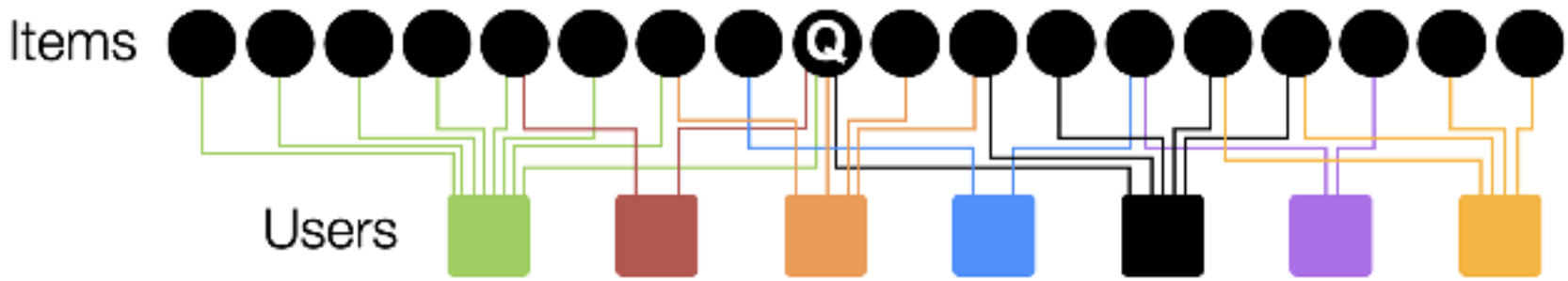
- We will talk about it more next week, but this provides a good example while GNN can do better than NN.
- Above we have the bipartite graph of users and products for a recommendation system.
- The users provided ratings for some of the items they selected.
- We can project this graph onto a user graph whose nodes $\mathcal{V}$ are the users and two users (v, u) are connected by an edge if they buy the same items more times than a given threshold.

Example: Recommendation Systems



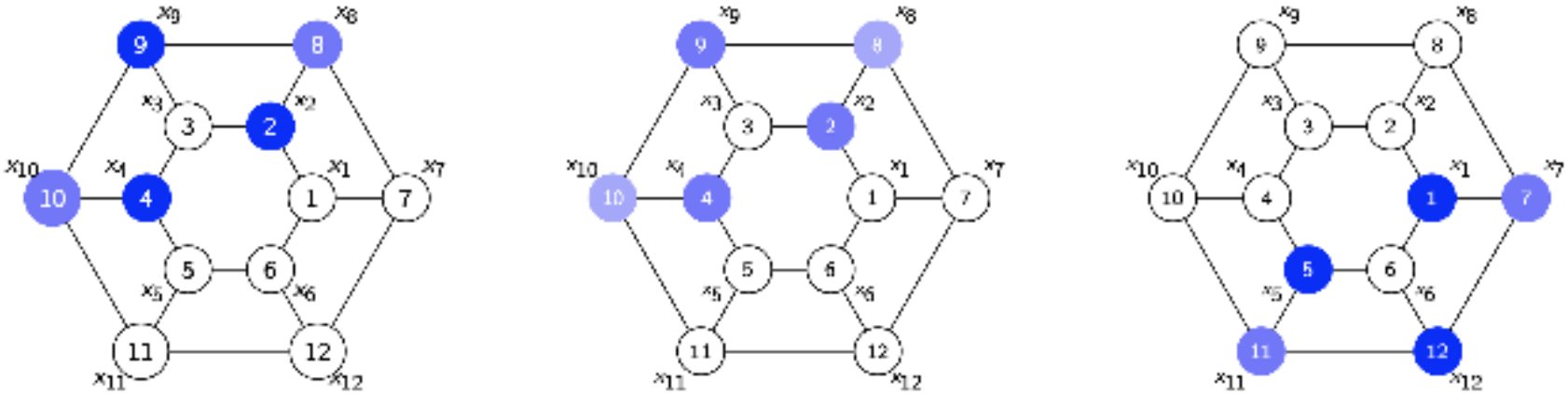
- We will talk about it more next week, but this provides a good example while GNN can do better than NN.
- Above we have the bipartite graph of users and products for a recommendation system.
- The users provided ratings for some of the items they selected.
- We can project this graph onto a user graph whose nodes $\mathcal{V}$ are the users and two users (v, u) are connected by an edge if they buy the same items more times than a given threshold.

Example: Recommendation Systems



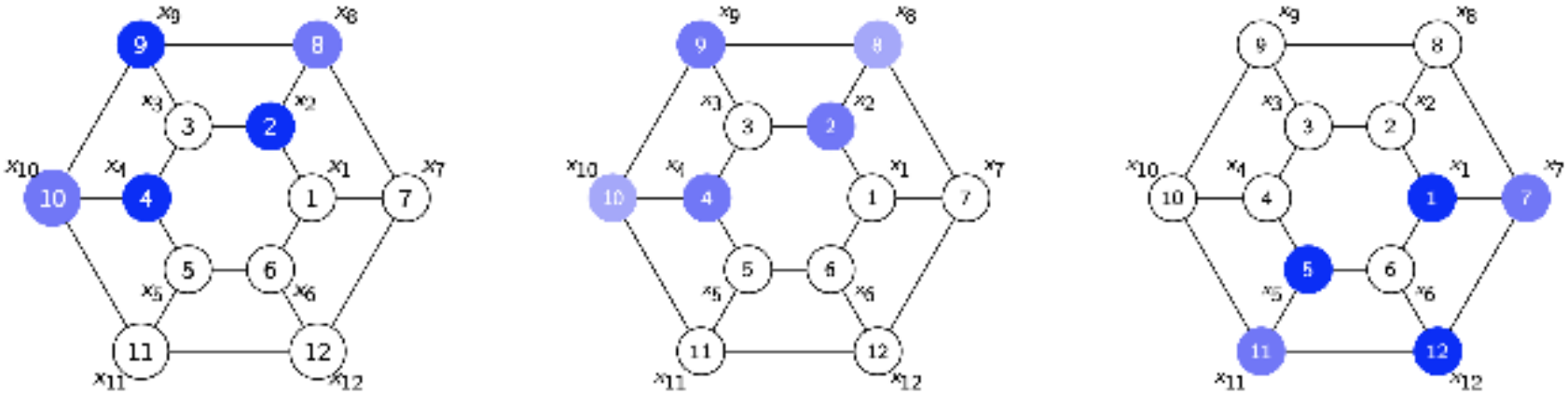
- We will talk about it more next week, but this provides a good example while GNN can do better than NN.
- Above we have the bipartite graph of users and products for a recommendation system.
- The users provided ratings for some of the items they selected.
- We can project this graph onto a user graph whose nodes $\mathcal{V}$ are the users and two users (v, u) are connected by an edge if they buy the same items more times than a given threshold.

Generalization with a Neural Network



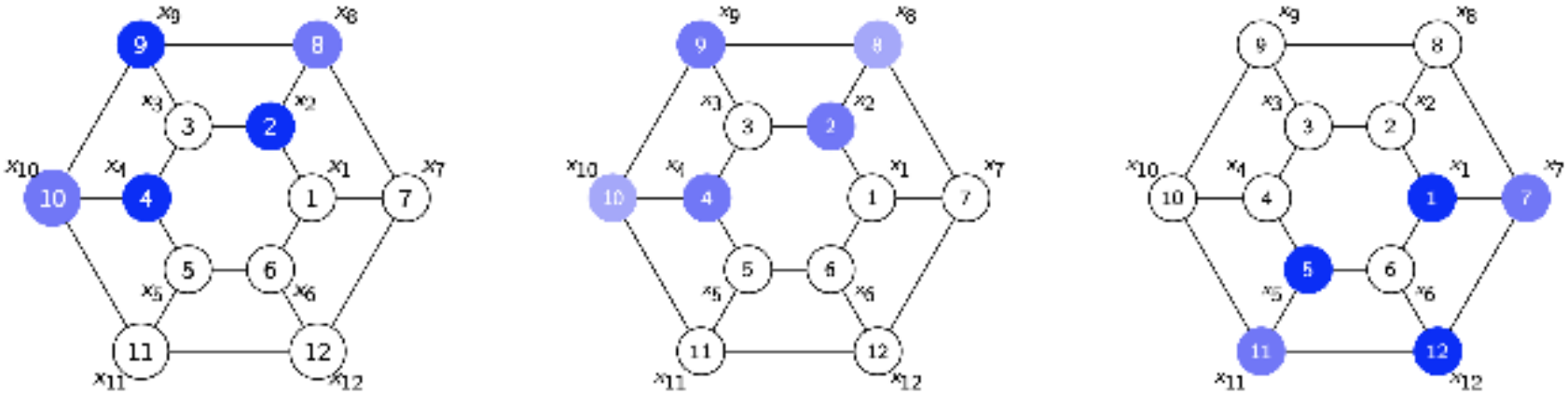
- Let us assume that the rating of an item is the optimum probability of recommending it, since the user will like it.
- In the user graph represented in the figure above on the left, the ratings of a set of item for the 5 users $\{x_2, x_4, x_8, x_9, x_{10}\}$ are known (the depth of the color represent the values of the ratings of one such item).
- We want to interpolate the ratings for new items that the users have not rated.

Generalization with a Neural Network



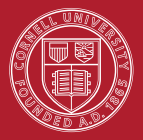
- Let us assume that the rating of an item is the optimum probability of recommending it, since the user will like it.
- In the user graph represented in the figure above on the left, the ratings of a set of item for the 5 users $\{x_2, x_4, x_8, x_9, x_{10}\}$ are known (the depth of the color represent the values of the ratings of one such item).
- We want to interpolate the ratings for new items that the users have not rated.

Generalization with a Neural Network

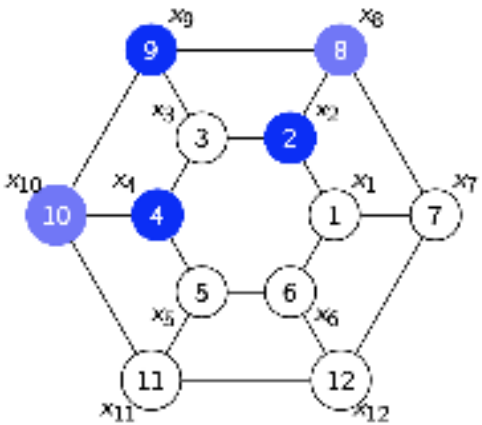


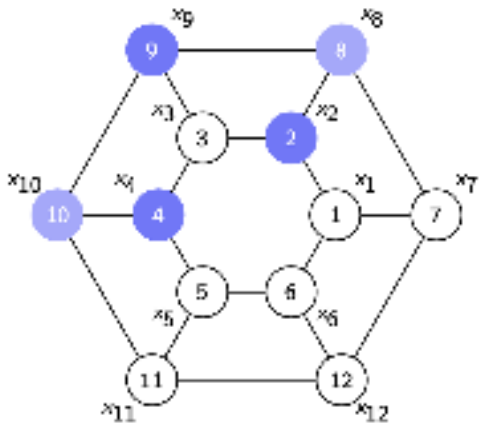
- Let us assume that the rating of an item is the optimum probability of recommending it, since the user will like it.
- In the user graph represented in the figure above on the left, the ratings of a set of item for the 5 users $\{x_2, x_4, x_8, x_9, x_{10}\}$ are known (the depth of the color represent the values of the ratings of one such item).
- We want to interpolate the ratings for new items that the users have not rated.

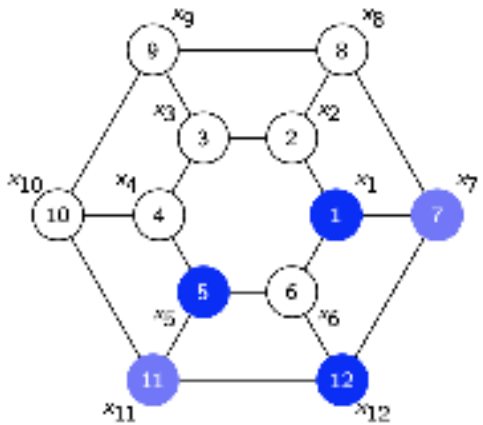
Generalization with a Neural Network



- In this example a conventional NN can infer ratings only for users that have provided some ratings
- Users that have not rated any of the subset of items considered, cannot be assigned a rating for them with NN, as shown in the middle-figure.
- The GNN, instead, can interpolate graph signals and learn how to best do it using the training data. In the right figure we show the results on a toy example where we learn the rating of an item for nodes $\{x_1, x_5, x_7, x_{11}, x_{12}\}$ from the rating of the training set for users $\{x_2, x_4, x_8, x_9, x_{10}\}$

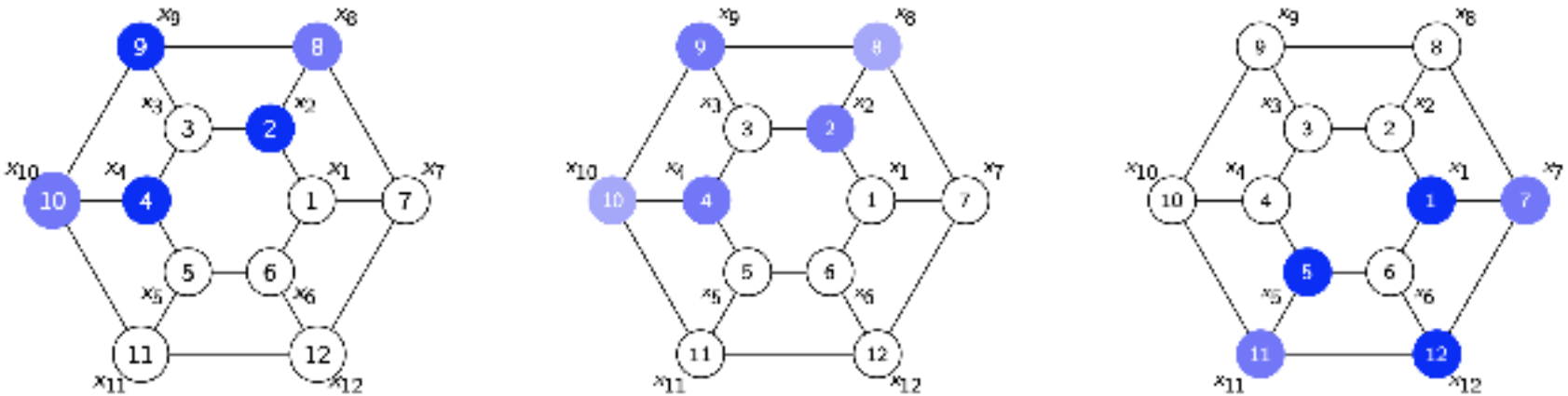






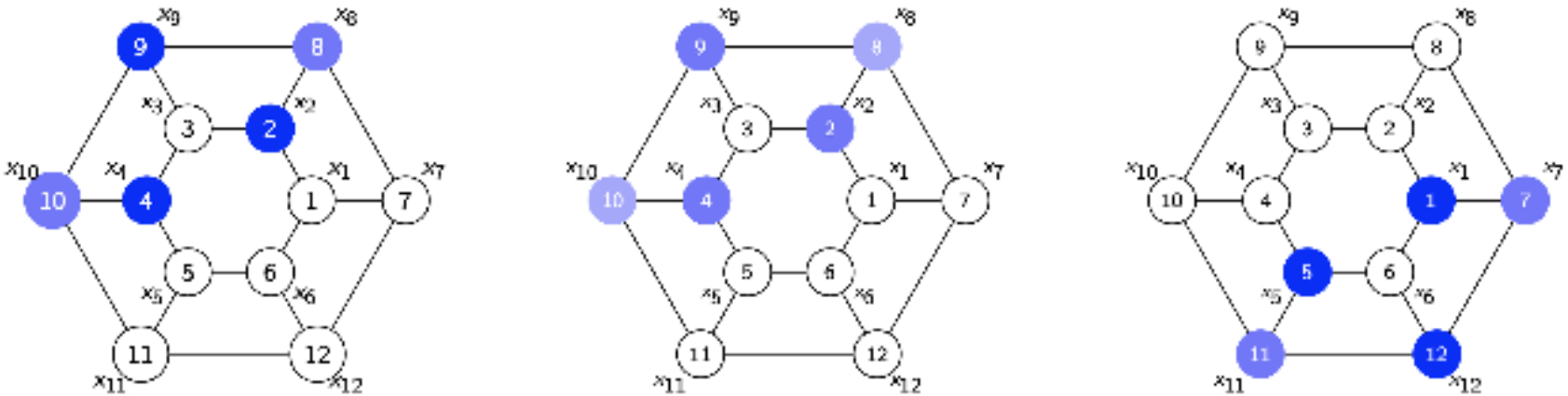
Generalization with a Neural Network

- In this example a conventional NN can infer ratings only for users that have provided some ratings
- Users that have not rated any of the subset of items considered, cannot be assigned a rating for them with NN, as shown in the middle-figure.
- The GNN, instead, can interpolate graph signals and learn how to best do it using the training data. In the right figure we show the results on a toy example where we learn the rating of an item for nodes $\{x_1, x_5, x_7, x_{11}, x_{12}\}$ from the rating of the training set for users $\{x_2, x_4, x_8, x_9, x_{10}\}$

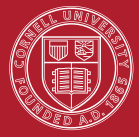


Generalization with a Neural Network

- In this example a conventional NN can infer ratings only for users that have provided some ratings
- Users that have not rated any of the subset of items considered, cannot be assigned a rating for them with NN, as shown in the middle-figure.
- The GNN, instead, can interpolate graph signals and learn how to best do it using the training data. In the right figure we show the results on a toy example where we learn the rating of an item for nodes $\{x_1, x_5, x_7, x_{11}, x_{12}\}$ from the rating of the training set for users $\{x_2, x_4, x_8, x_9, x_{10}\}$



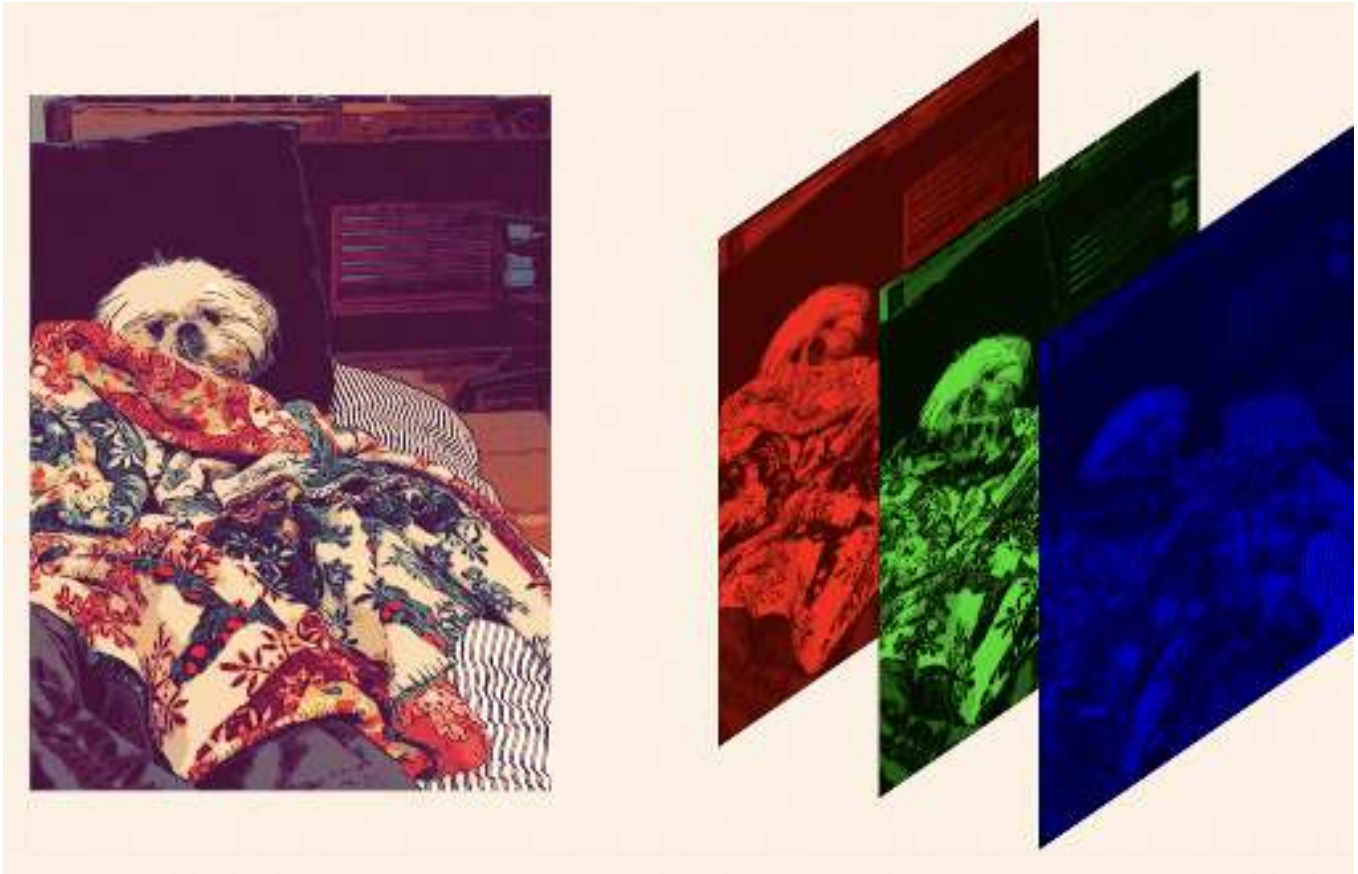
Outline



Cornell University

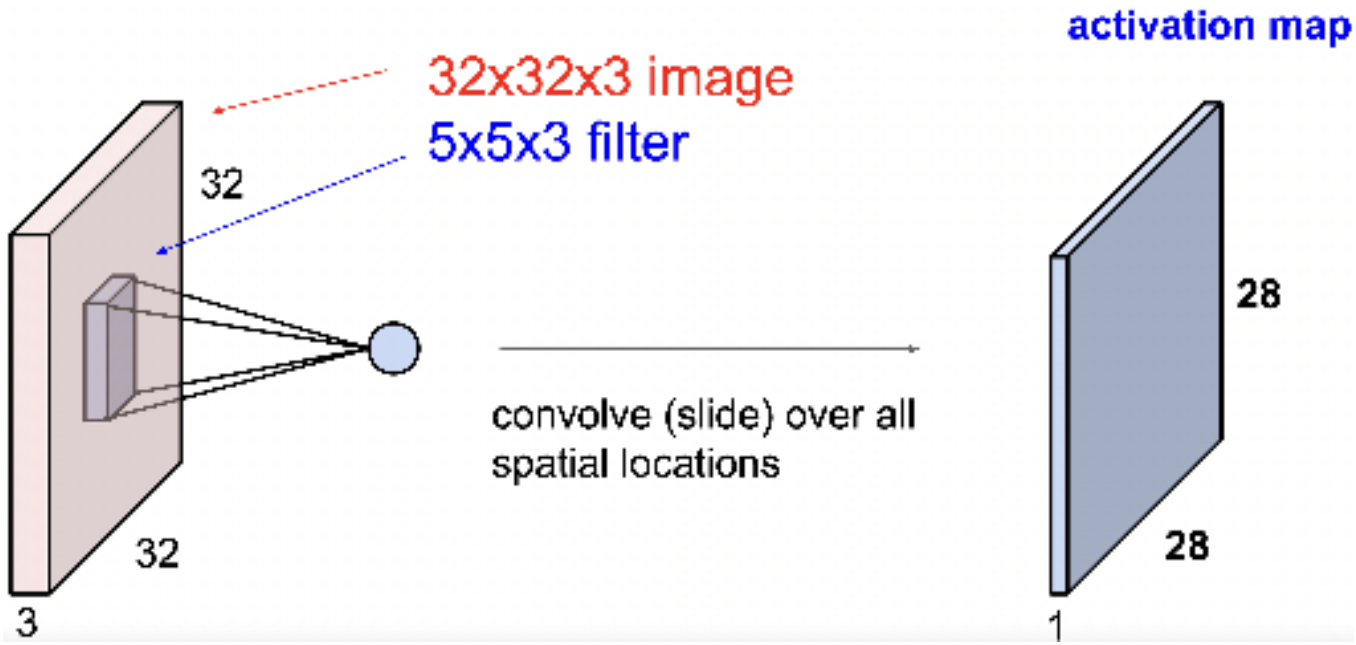
CNN for Computer Vision

- Colorful images have three channels: red channel, green channel and blue channel.
- CNN takes the RGB color images as multiple features for its 2-d convolutions.



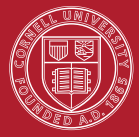
CNN for Computer Vison

- A closer look at spatial dimensions:



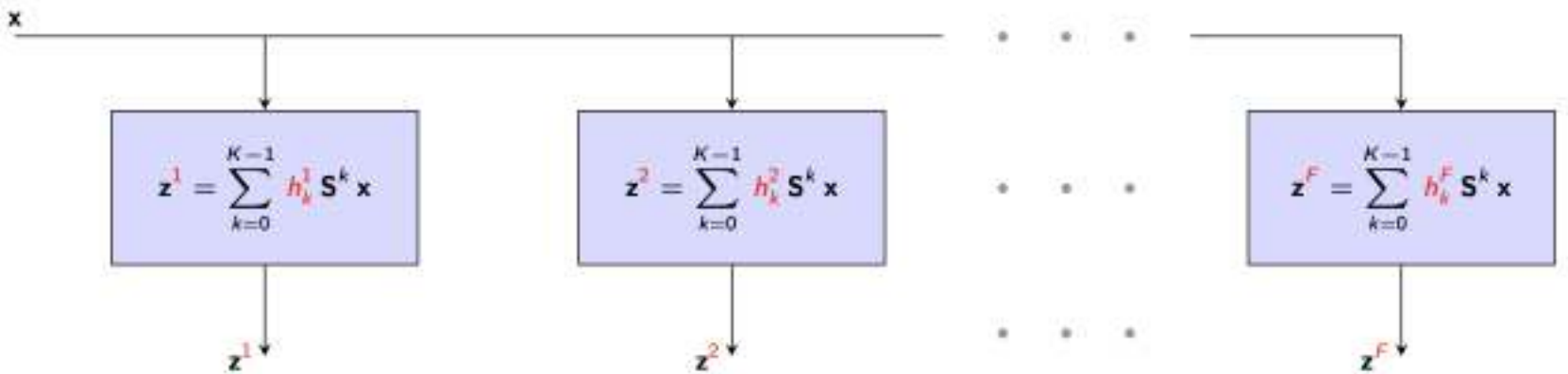
How can one generalize multiple-feature image signals to graph signals?

Graph Filter Banks



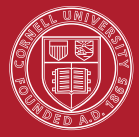
Cornell University

- Filters isolate features. When we are interested in multiple features, we use Banks of filters
- A graph filter bank is a collection of filters. Use F to denote total number of filters in the bank
- Filter f in the bank uses coefficients $\boldsymbol{h}^f = \left[h_1^f; \ldots; h_{K-1}^f\right] \Rightarrow$ Output $\boldsymbol{z}^f$ is a graph signal:

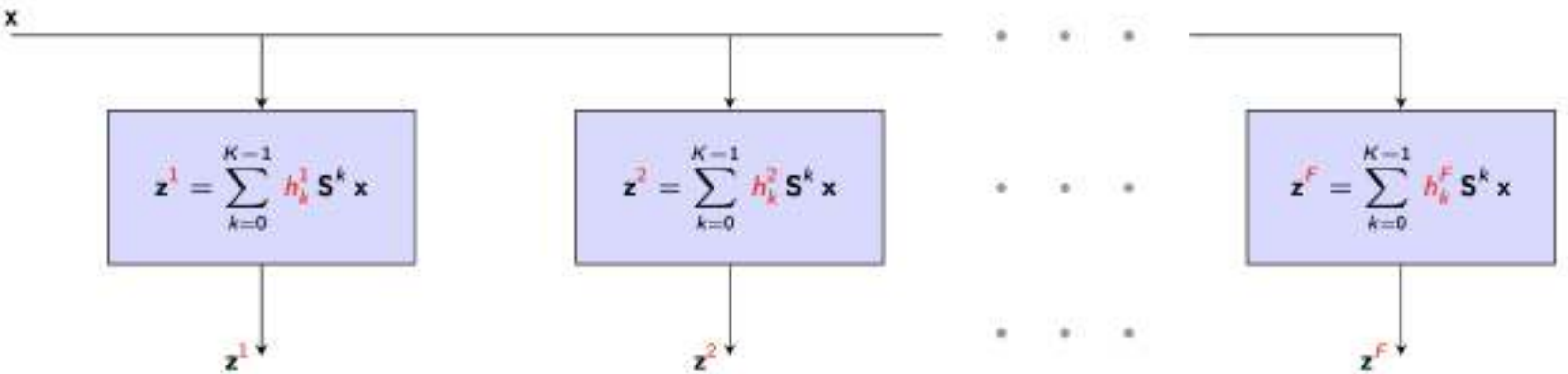


- Filter bank output is a collection of F graph signals $\Rightarrow$ Matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.

Graph Filter Banks

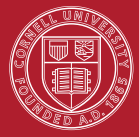


- Filters isolate features. When we are interested in multiple features, we use Banks of filters
- A graph filter bank is a collection of filters. Use F to denote total number of filters in the bank
- Filter f in the bank uses coefficients $\boldsymbol{h}^f = \left[h_1^f; \ldots; h_{K-1}^f\right] \Rightarrow$ Output $\boldsymbol{z}^f$ is a graph signal:

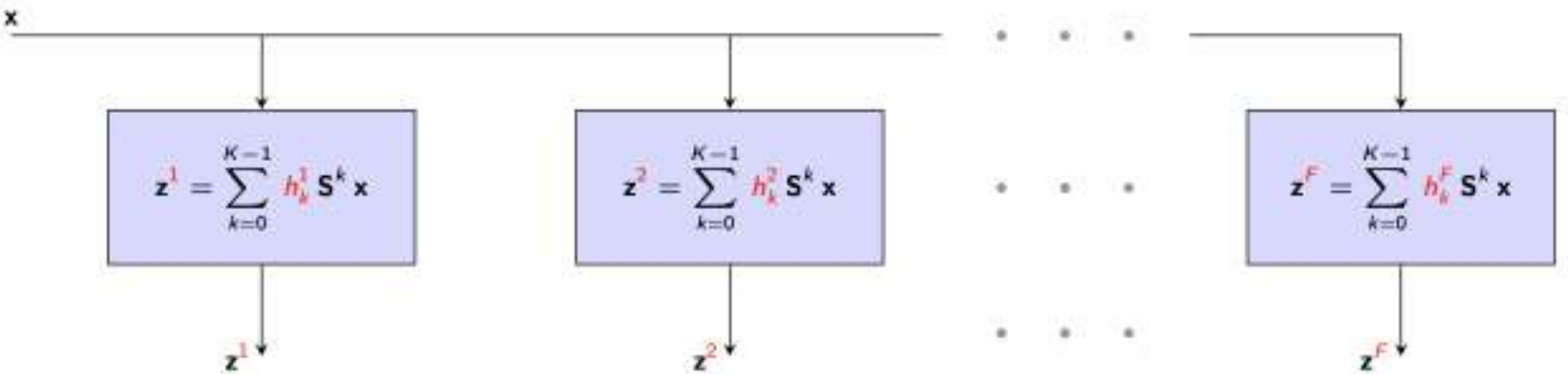


- Filter bank output is a collection of F graph signals $\Rightarrow$ Matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.

Graph Filter Banks

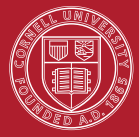


- Filters isolate features. When we are interested in multiple features, we use Banks of filters
- A graph filter bank is a collection of filters. Use F to denote total number of filters in the bank
- Filter f in the bank uses coefficients $\boldsymbol{h}^f = \left[h_1^f; \ldots; h_{K-1}^f\right] \Rightarrow$ Output $\boldsymbol{z}^f$ is a graph signal:



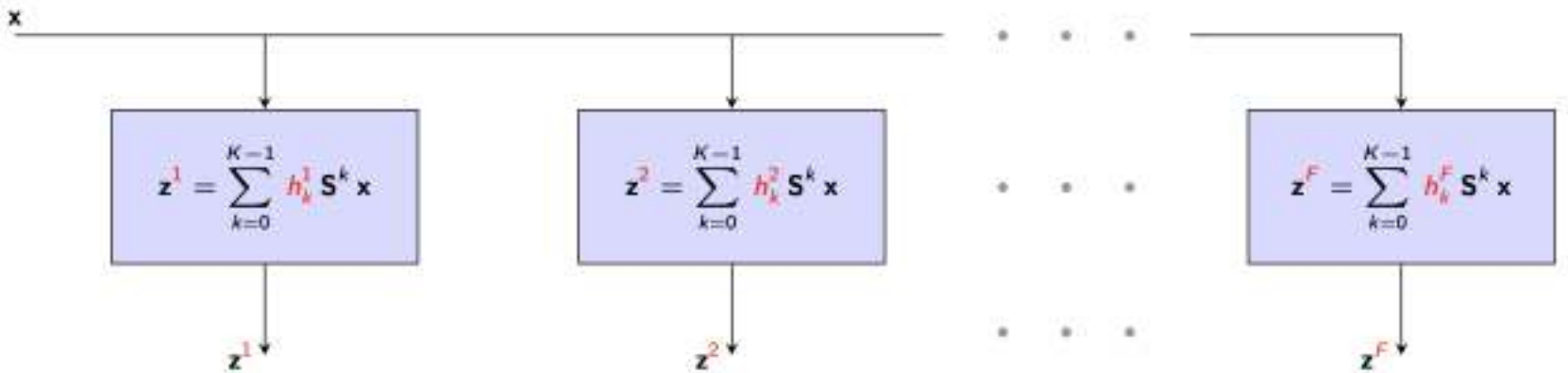
- Filter bank output is a collection of F graph signals $\Rightarrow$ Matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.

Graph Filter Banks



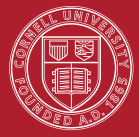
Cornell University

- Filters isolate features. When we are interested in multiple features, we use Banks of filters
- A graph filter bank is a collection of filters. Use F to denote total number of filters in the bank
- Filter f in the bank uses coefficients $\boldsymbol{h}^f = \left[h_1^f; \ldots; h_{K-1}^f\right] \Rightarrow$ Output $\boldsymbol{z}^f$ is a graph signal:

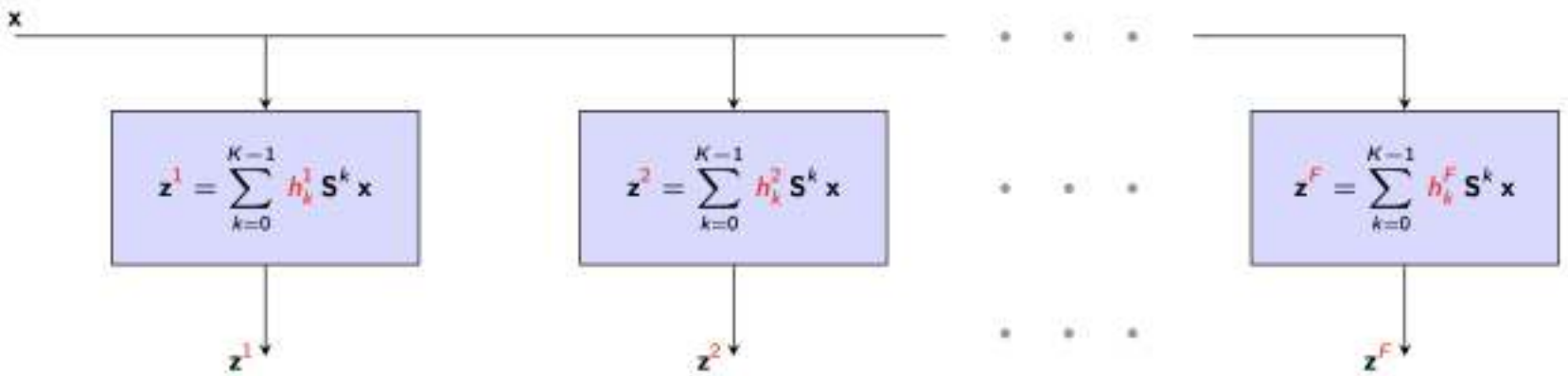


- Filter bank output is a collection of F graph signals $\Rightarrow$ Matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.

Multiple Feature GNNs

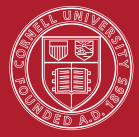


- We leverage filter banks to create GNNs that process multiple features per layer.
- Filter banks output a collection of multiple graph signals $\Rightarrow$ A matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.
- The F graph signals $\boldsymbol{z}^f$ represent F features per node.

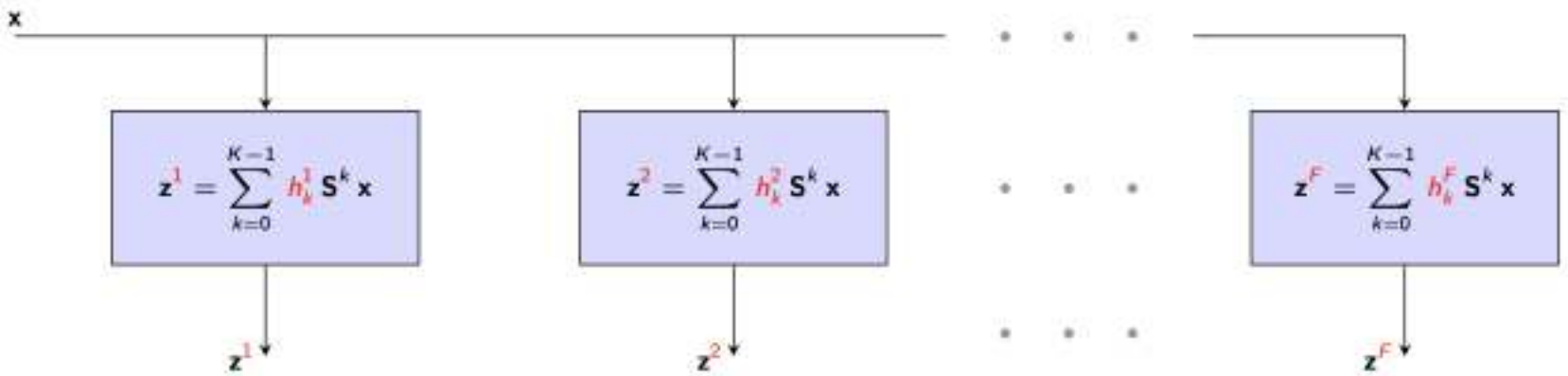


- We would now like to process multiple feature graph signals. Process each feature with a filter-bank.

Multiple Feature GNNs



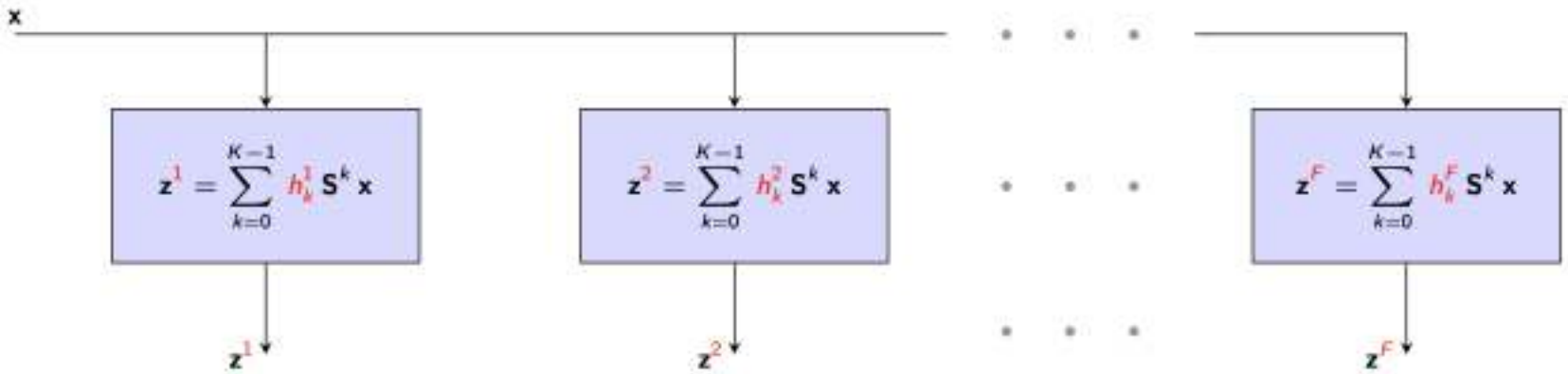
- We leverage filter banks to create GNNs that process multiple features per layer.
- Filter banks output a collection of multiple graph signals $\Rightarrow$ A matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.
- The F graph signals z^f represent F features per node.



- We would now like to process multiple feature graph signals. Process each feature with a filter-bank.

Multiple Feature GNNs

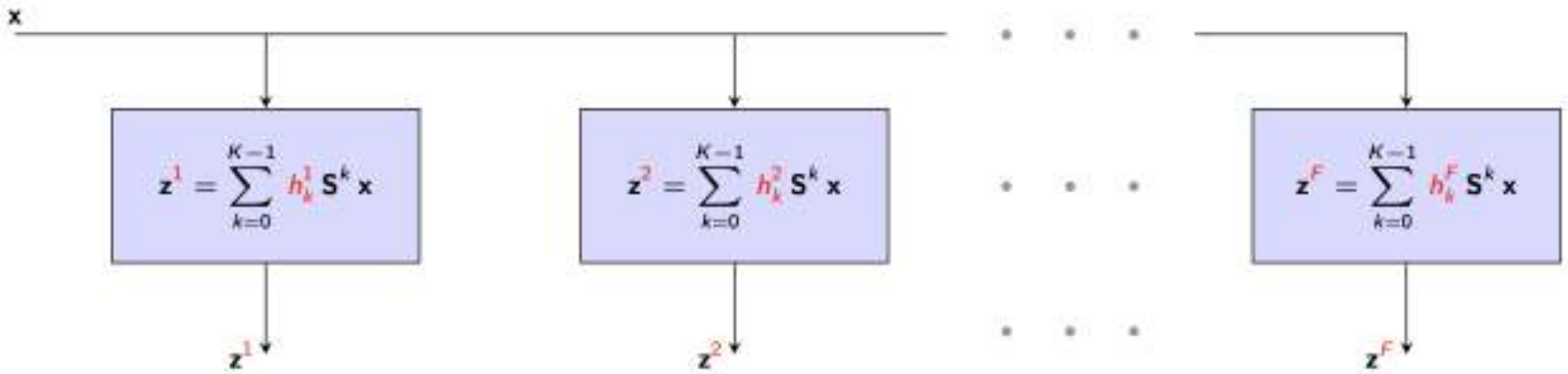
- We leverage filter banks to create GNNs that process multiple features per layer.
- Filter banks output a collection of multiple graph signals $\Rightarrow$ A matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.
- The F graph signals $\boldsymbol{z}^f$ represent F features per node.



- We would now like to process multiple feature graph signals. Process each feature with a filter-bank.

Multiple Feature GNNs

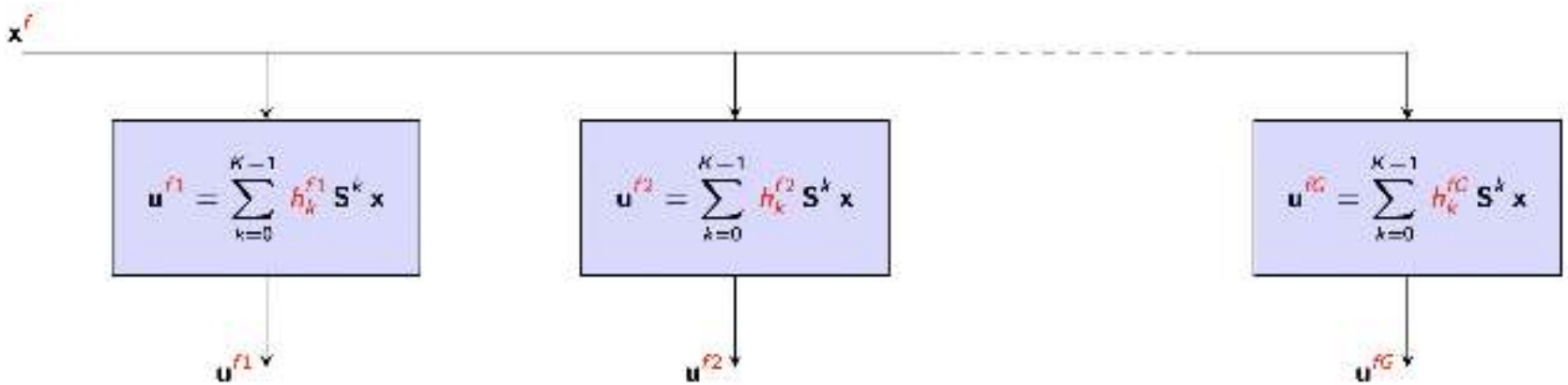
- We leverage filter banks to create GNNs that process multiple features per layer.
- Filter banks output a collection of multiple graph signals $\Rightarrow$ A matrix graph signal $\mathbf{Z} = [\boldsymbol{z}^1, \cdots, \boldsymbol{z}^F]$.
- The F graph signals $\boldsymbol{z}^f$ represent F features per node.



- We would now like to process multiple feature graph signals. Process each feature with a filter-bank.

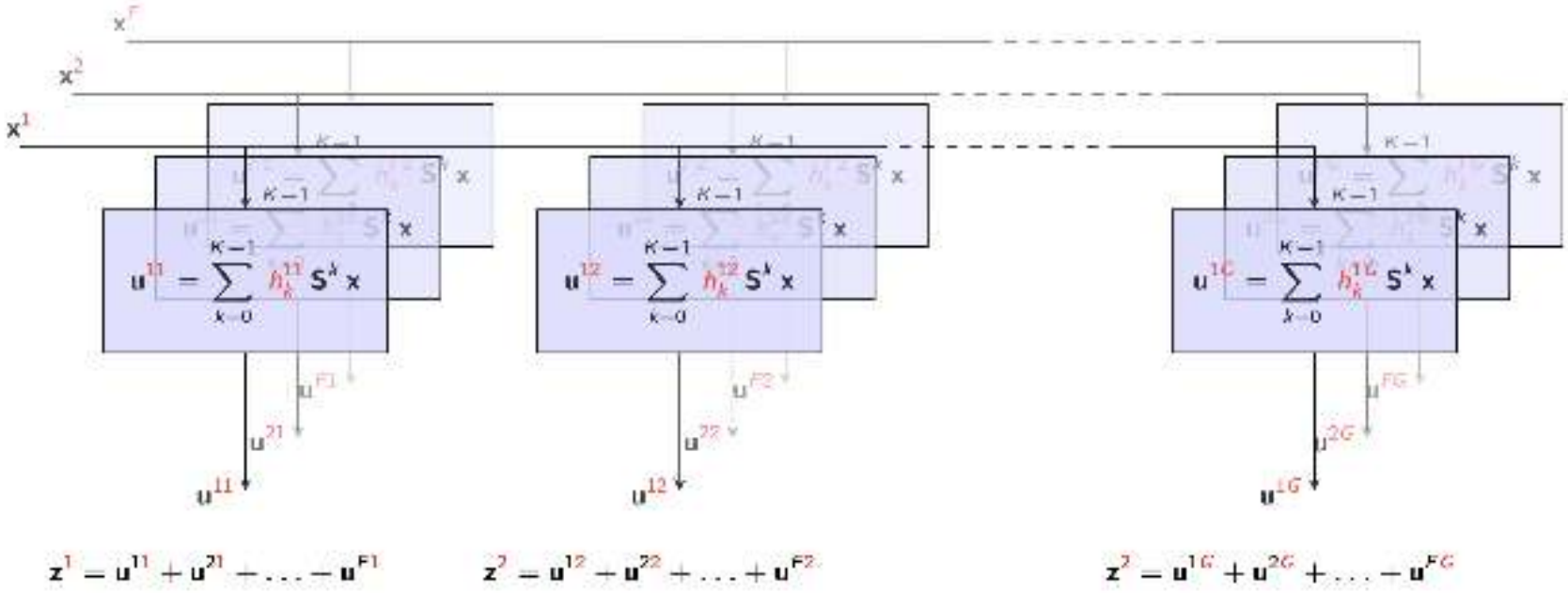
MIMO Graph Filters

- Each of the F features $\boldsymbol{x}^f$ is processed with G filters with coefficients $h_k^{fg} \Rightarrow \boldsymbol{u}^{fg} = \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \boldsymbol{x}^f$
- Therefore, we have



MIMO Graph Filters

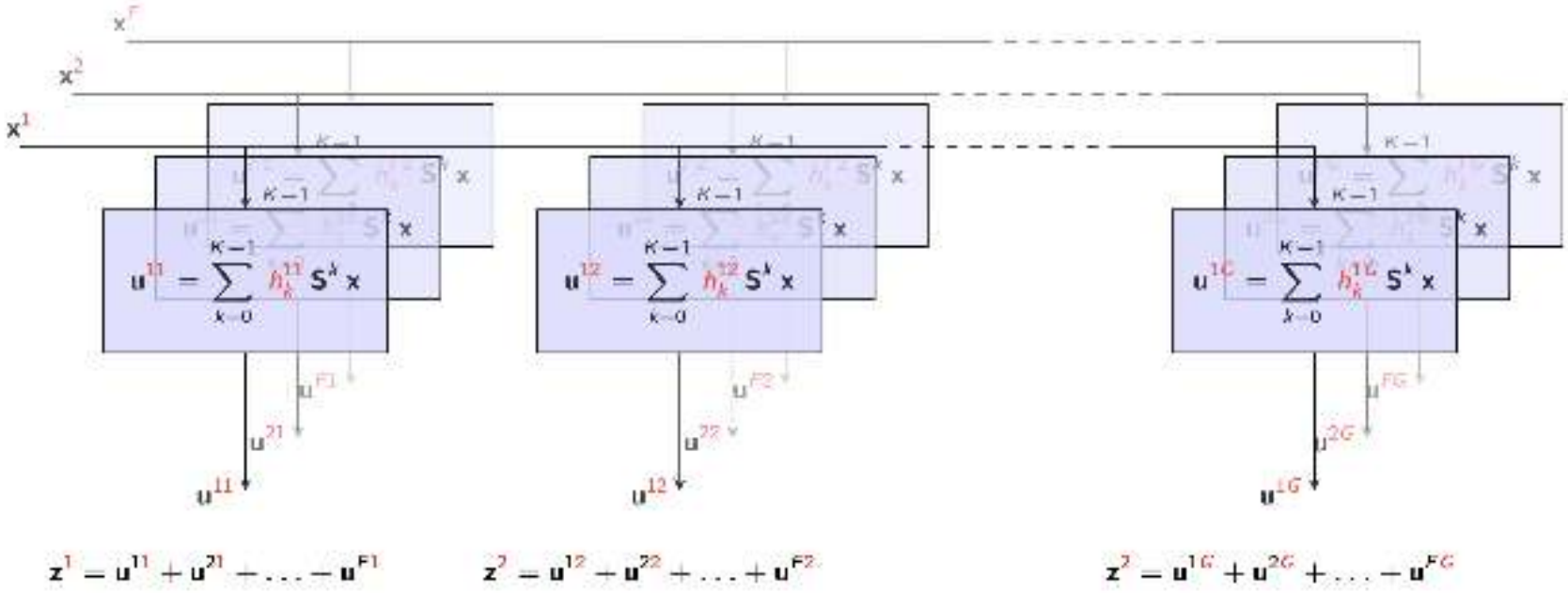
- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- Reduce to G outputs with sum over input features for given $g \Rightarrow \boldsymbol{z}^g = \sum_{f=1}^{F} \boldsymbol{u}^{fg} = \sum_{f=1}^{F} \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \boldsymbol{x}^f$



- Easier with matrices $\Rightarrow G \times F$ coefficient matrix $\mathbf{H}_k$ with entries $[\mathbf{H}_k]_{fg} = h_k^{fg}$: $\mathbf{Z} = \sum_{k=0}^{k-1} \mathbf{S}^k \times \mathbf{X} \times \mathbf{H}_k$

MIMO Graph Filters

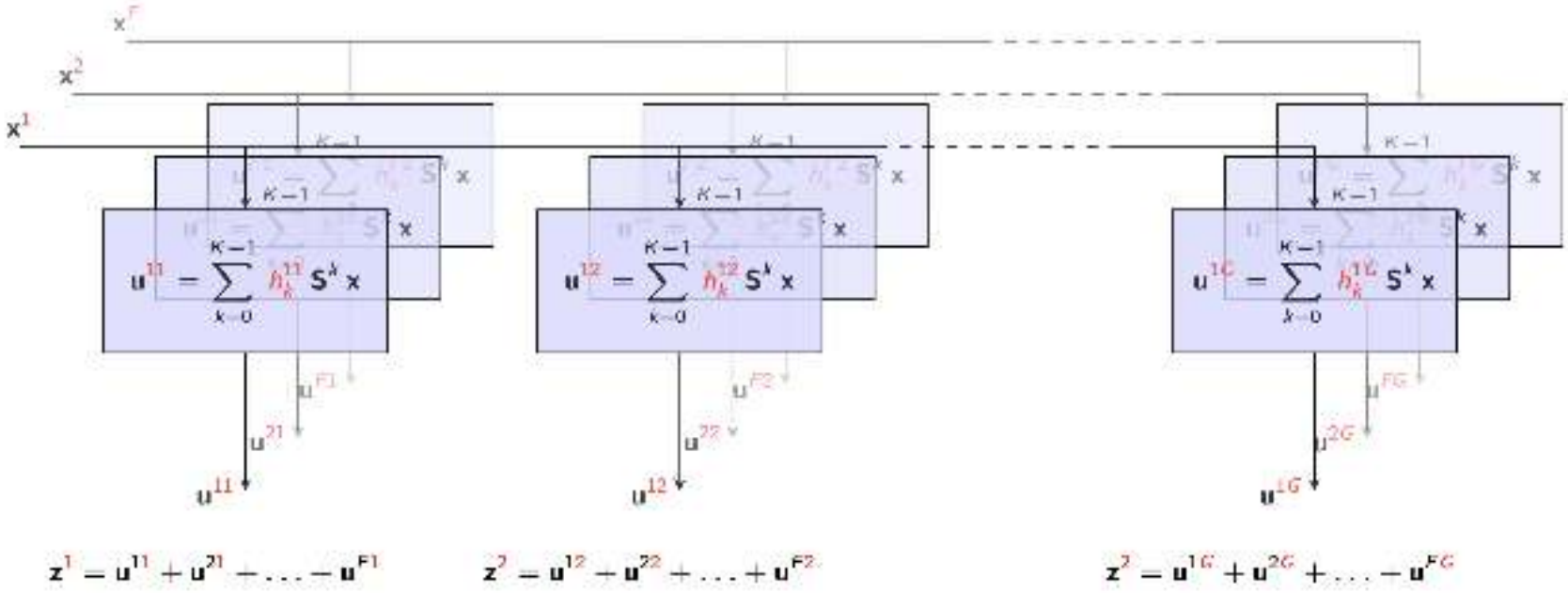
- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- Reduce to G outputs with sum over input features for given $g \Rightarrow \boldsymbol{z}^g = \sum_{f=1}^{F} \boldsymbol{u}^{fg} = \sum_{f=1}^{F} \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \boldsymbol{x}^f$



- Easier with matrices $\Rightarrow G \times F$ coefficient matrix $\mathbf{H}_k$ with entries $[\mathbf{H}_k]_{fg} = h_k^{fg}$: $\mathbf{Z} = \sum_{k=0}^{k-1} \mathbf{S}^k \times \mathbf{X} \times \mathbf{H}_k$

MIMO Graph Filters

- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- Reduce to G outputs with sum over input features for given $g \Rightarrow \boldsymbol{z}^g = \sum_{f=1}^{F} \boldsymbol{u}^{fg} = \sum_{f=1}^{F} \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \boldsymbol{x}^f$



- Easier with matrices $\Rightarrow G \times F$ coefficient matrix $\mathbf{H}_k$ with entries $[\mathbf{H}_k]_{fg} = h_k^{fg}$: $\mathbf{Z} = \sum_{k=0}^{k-1} \mathbf{S}^k \times \mathbf{X} \times \mathbf{H}_k$

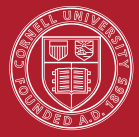
MIMO GNN / Multiple Feature GNN

- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- MIMO GNN stacks MIMO perceptrons $\Rightarrow$ Compose of MIMO filters with pointwise nonlinearities
- Layer ℓ processes input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\mathbf{H}_{\ell} = [\mathbf{H}_{\ell 0}, \ldots, \mathbf{H}_{\ell,K-1}]$ to produce output $\mathbf{X}_{\ell}$

$$\mathbf{X}_{\ell} = \sigma\left[\mathbf{Z}_{\ell}\right] = \sigma\left[\sum_{k=0}^{K-1} \mathbf{S}^{k} \mathbf{X}_{\ell-1} \mathbf{H}_{\ell k}\right]$$

- Denoting the Layer 1 input as $\mathbf{X}_0 = \mathbf{X}$, this provides a recursive definition of a MIMO GNN
- The filter tensor $\mathcal{H} = [\mathbf{H}_1, \ldots, \mathbf{H}_L]$ is the trainable parameter. The graph shift is prior information.

MIMO GNN / Multiple Feature GNN



- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- MIMO GNN stacks MIMO perceptrons $\Rightarrow$ Compose of MIMO filters with pointwise nonlinearities
- Layer ℓ processes input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\mathbf{H}_{\ell} = [\mathbf{H}_{\ell 0}, \ldots, \mathbf{H}_{\ell,K-1}]$ to produce output $\mathbf{X}_{\ell}$

$$\mathbf{X}_{\ell} = \sigma\left[\mathbf{Z}_{\ell}\right] = \sigma\left[\sum_{k=0}^{K-1} \mathbf{S}^{k} \mathbf{X}_{\ell-1} \mathbf{H}_{\ell k}\right]$$

- Denoting the Layer 1 input as $\mathbf{X}_0 = \mathbf{X}$, this provides a recursive definition of a MIMO GNN
- The filter tensor $\mathcal{H} = [\mathbf{H}_1, \ldots, \mathbf{H}_L]$ is the trainable parameter. The graph shift is prior information.

MIMO GNN / Multiple Feature GNN

- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- MIMO GNN stacks MIMO perceptrons $\Rightarrow$ Compose of MIMO filters with pointwise nonlinearities
- Layer ℓ processes input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\mathbf{H}_{\ell} = [\mathbf{H}_{\ell 0}, \ldots, \mathbf{H}_{\ell,K-1}]$ to produce output $\mathbf{X}_{\ell}$

$$\mathbf{X}_{\ell} = \sigma\left[\mathbf{Z}_{\ell}\right] = \sigma\left[\sum_{k=0}^{K-1} \mathbf{S}^{k} \mathbf{X}_{\ell-1} \mathbf{H}_{\ell k}\right]$$

- Denoting the Layer 1 input as $\mathbf{X}_0 = \mathbf{X}$, this provides a recursive definition of a MIMO GNN
- The filter tensor $\mathcal{H} = [\mathbf{H}_1, \ldots, \mathbf{H}_L]$ is the trainable parameter. The graph shift is prior information.

MIMO GNN / Multiple Feature GNN

- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- MIMO GNN stacks MIMO perceptrons $\Rightarrow$ Compose of MIMO filters with pointwise nonlinearities
- Layer ℓ processes input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\mathbf{H}_{\ell} = [\mathbf{H}_{\ell 0}, \ldots, \mathbf{H}_{\ell,K-1}]$ to produce output $\mathbf{X}_{\ell}$

$$\mathbf{X}_{\ell} = \sigma\left[\mathbf{Z}_{\ell}\right] = \sigma\left[\sum_{k=0}^{K-1} \mathbf{S}^{k} \mathbf{X}_{\ell-1} \mathbf{H}_{\ell k}\right]$$

- Denoting the Layer 1 input as $\mathbf{X}_0 = \mathbf{X}$, this provides a recursive definition of a MIMO GNN
- The filter tensor $\mathcal{H} = [\mathbf{H}_1, \ldots, \mathbf{H}_L]$ is the trainable parameter. The graph shift is prior information.

MIMO GNN / Multiple Feature GNN

- This Multiple-Input-Multiple-Output Graph Filter generates an output with $F \times G$ features
- MIMO GNN stacks MIMO perceptrons $\Rightarrow$ Compose of MIMO filters with pointwise nonlinearities
- Layer ℓ processes input signal $\boldsymbol{x}_{\ell-1}$ with perceptron $\mathbf{H}_{\ell} = [\mathbf{H}_{\ell 0}, \ldots, \mathbf{H}_{\ell,K-1}]$ to produce output $\mathbf{X}_{\ell}$

$$\mathbf{X}_{\ell} = \sigma\left[\mathbf{Z}_{\ell}\right] = \sigma\left[\sum_{k=0}^{K-1} \mathbf{S}^{k} \mathbf{X}_{\ell-1} \mathbf{H}_{\ell k}\right]$$

- Denoting the Layer 1 input as $\mathbf{X}_0 = \mathbf{X}$, this provides a recursive definition of a MIMO GNN
- The filter tensor $\mathcal{H} = [\mathbf{H}_1, \ldots, \mathbf{H}_L]$ is the trainable parameter. The graph shift is prior information.

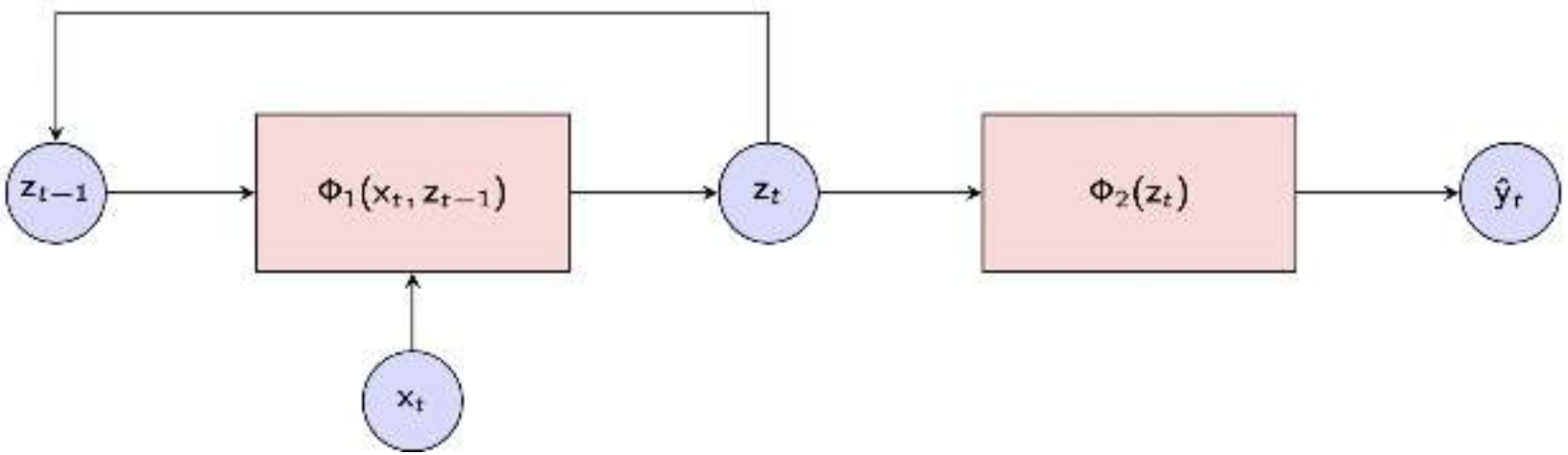
Outline

Cornell University

Recurrent Neural Networks

Cornell University

- A recurrent neural network is made up of two separate learning parametrizations
  $\Rightarrow \Phi_1(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}) \Rightarrow$ From observed state $\boldsymbol{x}_t \Rightarrow$ and hidden state $\boldsymbol{z}_{t-1} \Rightarrow$ to hidden state update $\boldsymbol{z}_t$
  $\Rightarrow \Phi_2(\boldsymbol{z}_t) \Rightarrow$ From updated hidden state $\boldsymbol{z}_t \Rightarrow$ to output estimate $\hat{y}_t$

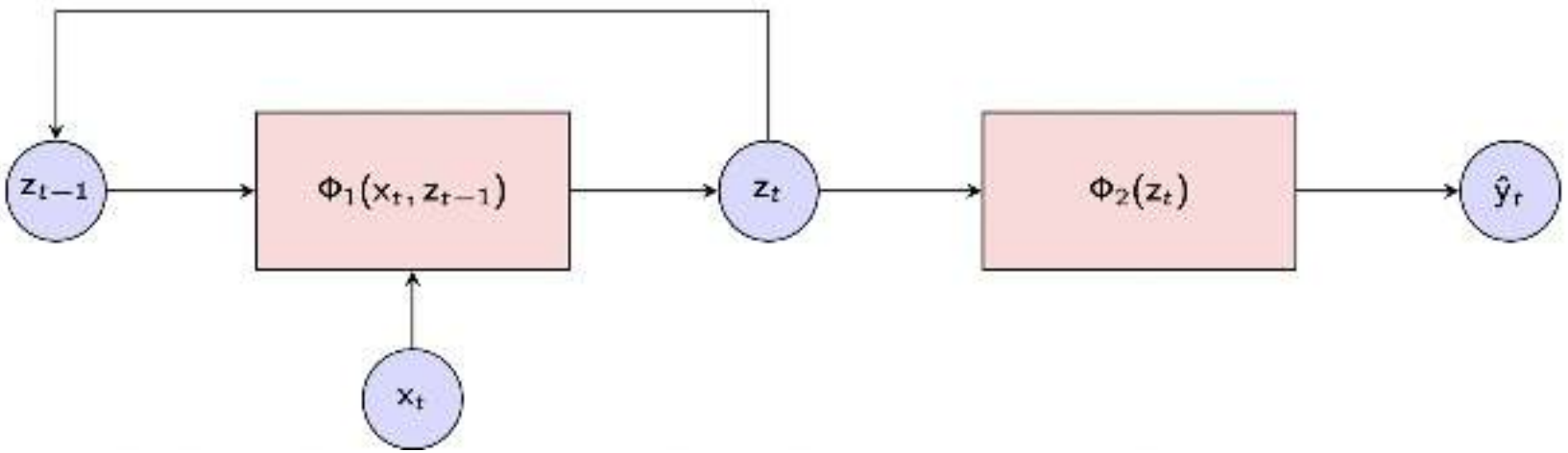


- It is a recurrent neural network because hidden states are fed-back as inputs for the next time step

Recurrent Neural Networks

Cornell University

- A recurrent neural network is made up of two separate learning parametrizations
  $\Rightarrow \Phi_1(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}) \Rightarrow$ From observed state $\boldsymbol{x}_t \Rightarrow$ and hidden state $\boldsymbol{z}_{t-1} \Rightarrow$ to hidden state update $\boldsymbol{z}_t$
  $\Rightarrow \Phi_2(\boldsymbol{z}_t) \Rightarrow$ From updated hidden state $\boldsymbol{z}_t \Rightarrow$ to output estimate $\hat{y}_t$

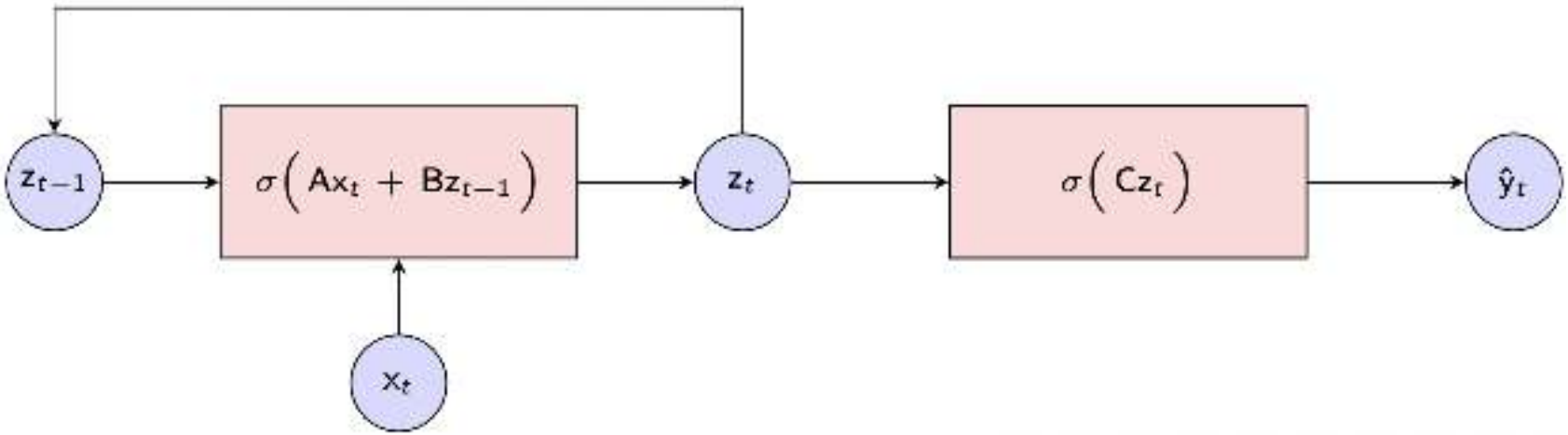


- It is a recurrent neural network because hidden states are fed-back as inputs for the next time step

Recurrent Neural Networks

Cornell University

- Use a perceptron that updates the hidden state $\boldsymbol{x}_t$
  $\Rightarrow \Phi_1(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}) = \sigma(\mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{z}_{t-1})$
- Number of learnable parameters: Entries of $\mathbf{A}$ and $\mathbf{B}$ $\Rightarrow$ Does not depend on the time index t

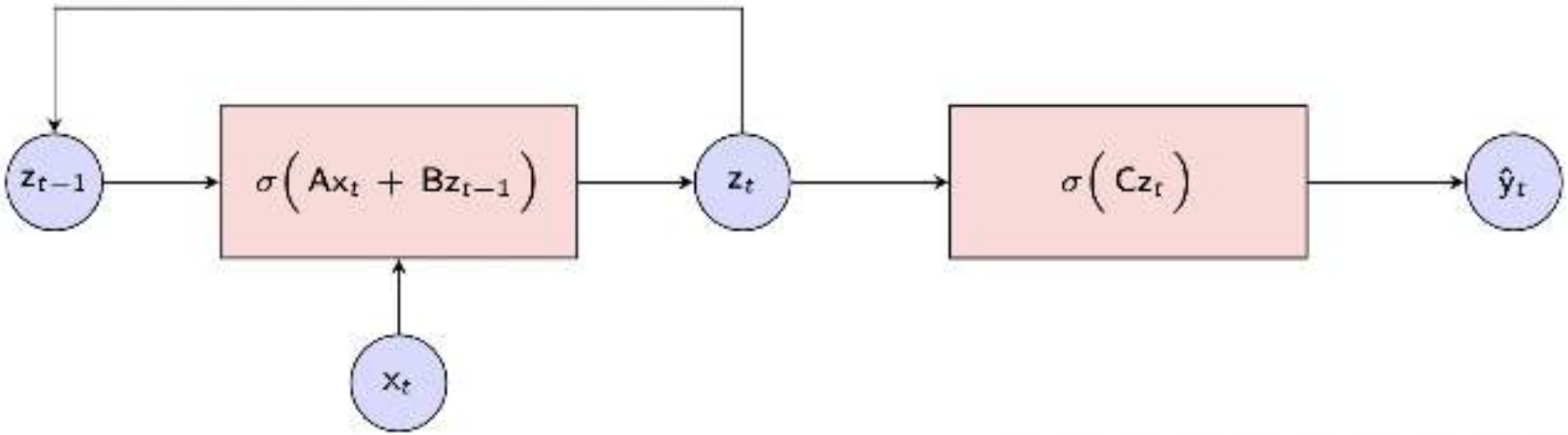


- Use another perceptron that predicts the output $\boldsymbol{y}_t$
  $\Rightarrow \Phi_1(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}) = \sigma(C\boldsymbol{x}_t)$

Recurrent Neural Networks

Cornell University

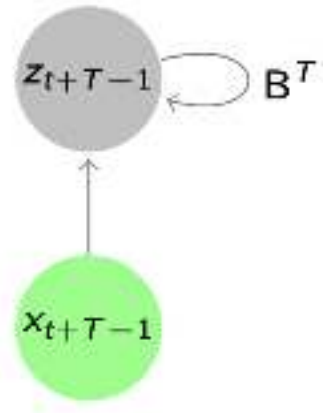
- Use a perceptron that updates the hidden state $\boldsymbol{x}_t$
  $\Rightarrow \Phi_1\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}\right) = \sigma\left(\mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{z}_{t-1}\right)$
- Number of learnable parameters: Entries of $\mathbf{A}$ and $\mathbf{B}$ $\Rightarrow$ Does not depend on the time index t



- Use another perceptron that predicts the output $\boldsymbol{y}_t$
  $\Rightarrow \Phi_1\left(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}\right) = \sigma\left(C\boldsymbol{x}_t\right)$

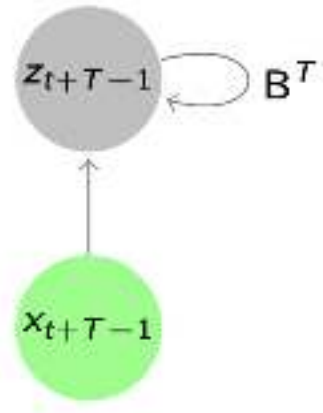
Time Gating

- Next we discuss
  - the problem of vanishing/exploding gradients in recurrent neural networks
  - the definition of gating mechanisms in the form of long short-term memory for gated recurrent units
- In some tasks, the RNN may have to learn how to model long term dependencies of length $\mathcal{T}$.
- This poses a challenge $\Rightarrow$ the Jacobian $\partial \boldsymbol{z}_T / \partial \mathbf{B}$ will depend on a chain of multiplications by $\mathbf{B}$
- If the eigenvalues of $\mathbf{B} \ll 1$, the gradients tend to vanish, leading to exponentially smaller weights $\mathbf{B}$
- If eigenvalues of $\mathbf{B} \gg 1$, instead, the gradients tend to explode, leading to exponentially larger weights $\mathbf{B}$



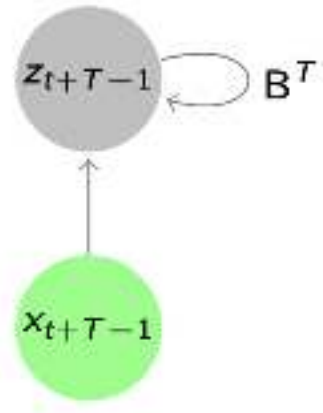
Time Gating

- Next we discuss
  - the problem of vanishing/exploding gradients in recurrent neural networks
  - the definition of gating mechanisms in the form of long short-term memory for gated recurrent units
- In some tasks, the RNN may have to learn how to model long term dependencies of length $\mathcal{T}$.
- This poses a challenge $\Rightarrow$ the Jacobian $\partial \boldsymbol{z}_T / \partial \mathbf{B}$ will depend on a chain of multiplications by $\mathbf{B}$
- If the eigenvalues of $\mathbf{B} \ll 1$, the gradients tend to vanish, leading to exponentially smaller weights $\mathbf{B}$
- If eigenvalues of $\mathbf{B} \gg 1$, instead, the gradients tend to explode, leading to exponentially larger weights $\mathbf{B}$



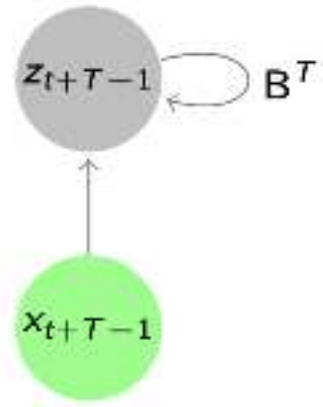
Time Gating

- Next we discuss
  - the problem of vanishing/exploding gradients in recurrent neural networks
  - the definition of gating mechanisms in the form of long short-term memory for gated recurrent units
- In some tasks, the RNN may have to learn how to model long term dependencies of length $\mathcal{T}$.
- This poses a challenge $\Rightarrow$ the Jacobian $\partial \boldsymbol{z}_T / \partial \mathbf{B}$ will depend on a chain of multiplications by $\mathbf{B}$
- If the eigenvalues of $\mathbf{B} \ll 1$, the gradients tend to vanish, leading to exponentially smaller weights $\mathbf{B}$
- If eigenvalues of $\mathbf{B} \gg 1$, instead, the gradients tend to explode, leading to exponentially larger weights $\mathbf{B}$



Time Gating

- Next we discuss
  - the problem of vanishing/exploding gradients in recurrent neural networks
  - the definition of gating mechanisms in the form of long short-term memory for gated recurrent units
- In some tasks, the RNN may have to learn how to model long term dependencies of length $\mathcal{T}$.
- This poses a challenge $\Rightarrow$ the Jacobian $\partial \boldsymbol{z}_T / \partial \mathbf{B}$ will depend on a chain of multiplications by $\mathbf{B}$
- If the eigenvalues of $\mathbf{B} \ll 1$, the gradients tend to vanish, leading to exponentially smaller weights $\mathbf{B}$
- If eigenvalues of $\mathbf{B} \gg 1$, instead, the gradients tend to explode, leading to exponentially larger weights $\mathbf{B}$



Long Term Dependencies

- The most popular gated RNN architecture is the Long Short-Term Memory (LSTM) cell, where we add three types parameters (f_t, g_t, q_t) in addition to those in $\mathcal{H}$
- These parameters control three types of gates:
  1. a **forget gate** $f_t \in [0, 1]$,
  2. an **input gate** $g_t \in [0, 1]$
  3. a **cell output gate** $q_t \in [0, 1]$
- Let $\boldsymbol{x}_t$ be the input, $\boldsymbol{z}_t$ the state, and define the internal memory $\boldsymbol{u}_t$ of the LSTM cell
- The memory $\boldsymbol{u}_t$ updated by applying the forget gate to $\boldsymbol{u}_{t-1}$ and the input gate to the state update

$$\boldsymbol{u}_t = f_t \boldsymbol{u}_{t-1} + g_t \sigma \left(\mathbf{A} \boldsymbol{x}_t + \mathbf{B} \boldsymbol{z}_{t-1} \right) \tag{2}$$

- State $\boldsymbol{z}_t$ updated by applying the cell output gate to the internal cell memory $\boldsymbol{u}_t$

$$\boldsymbol{z}_t = q_t \sigma(\boldsymbol{u}_t) \tag{3}$$

Long Term Dependencies

- The most popular gated RNN architecture is the Long Short-Term Memory (LSTM) cell, where we add three types parameters (f_t, g_t, q_t) in addition to those in $\mathcal{H}$
- These parameters control three types of gates:
  1. a **forget gate** $f_t \in [0, 1]$,
  2. an **input gate** $g_t \in [0, 1]$
  3. a **cell output gate** $q_t \in [0, 1]$
- Let $\boldsymbol{x}_t$ be the input, $\boldsymbol{z}_t$ the state, and define the internal memory $\boldsymbol{u}_t$ of the LSTM cell
- The memory $\boldsymbol{u}_t$ updated by applying the forget gate to $\boldsymbol{u}_{t-1}$ and the input gate to the state update

$$\boldsymbol{u}_t = f_t \boldsymbol{u}_{t-1} + g_t \sigma \left(\mathbf{A} \boldsymbol{x}_t + \mathbf{B} \boldsymbol{z}_{t-1} \right) \tag{2}$$

- State $\boldsymbol{z}_t$ updated by applying the cell output gate to the internal cell memory $\boldsymbol{u}_t$

$$\boldsymbol{z}_t = q_t \sigma(\boldsymbol{u}_t) \tag{3}$$

Long Term Dependencies

- The most popular gated RNN architecture is the Long Short-Term Memory (LSTM) cell, where we add three types parameters (f_t, g_t, q_t) in addition to those in $\mathcal{H}$
- These parameters control three types of gates:
  1. a **forget gate** $f_t \in [0, 1]$,
  2. an **input gate** $g_t \in [0, 1]$
  3. a **cell output gate** $q_t \in [0, 1]$
- Let $\boldsymbol{x}_t$ be the input, $\boldsymbol{z}_t$ the state, and define the internal memory $\boldsymbol{u}_t$ of the LSTM cell
- The memory $\boldsymbol{u}_t$ updated by applying the forget gate to $\boldsymbol{u}_{t-1}$ and the input gate to the state update

$$\boldsymbol{u}_t = f_t \boldsymbol{u}_{t-1} + g_t \sigma\left(\mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{z}_{t-1}\right) \tag{2}$$

- State $\boldsymbol{z}_t$ updated by applying the cell output gate to the internal cell memory $\boldsymbol{u}_t$

$$\boldsymbol{z}_t = q_t \sigma(\boldsymbol{u}_t) \tag{3}$$

Long Term Dependencies

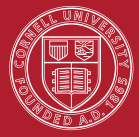
- The most popular gated RNN architecture is the Long Short-Term Memory (LSTM) cell, where we add three types parameters (f_t, g_t, q_t) in addition to those in $\mathcal{H}$
- These parameters control three types of gates:
  1. a **forget gate** $f_t \in [0, 1]$,
  2. an **input gate** $g_t \in [0, 1]$
  3. a **cell output gate** $q_t \in [0, 1]$
- Let $\boldsymbol{x}_t$ be the input, $\boldsymbol{z}_t$ the state, and define the internal memory $\boldsymbol{u}_t$ of the LSTM cell
- The memory $\boldsymbol{u}_t$ updated by applying the forget gate to $\boldsymbol{u}_{t-1}$ and the input gate to the state update

$$\boldsymbol{u}_t = f_t \boldsymbol{u}_{t-1} + g_t \sigma \left(\mathbf{A} \boldsymbol{x}_t + \mathbf{B} \boldsymbol{z}_{t-1} \right) \tag{2}$$

- State $\boldsymbol{z}_t$ updated by applying the cell output gate to the internal cell memory $\boldsymbol{u}_t$

$$\boldsymbol{z}_t = q_t \sigma(\boldsymbol{u}_t) \tag{3}$$

Long Term Dependencies



- The most popular gated RNN architecture is the Long Short-Term Memory (LSTM) cell, where we add three types parameters (f_t, g_t, q_t) in addition to those in $\mathcal{H}$
- These parameters control three types of gates:
  1. a **forget gate** $f_t \in [0, 1]$,
  2. an **input gate** $g_t \in [0, 1]$
  3. a **cell output gate** $q_t \in [0, 1]$
- Let $\boldsymbol{x}_t$ be the input, $\boldsymbol{z}_t$ the state, and define the internal memory $\boldsymbol{u}_t$ of the LSTM cell
- The memory $\boldsymbol{u}_t$ updated by applying the forget gate to $\boldsymbol{u}_{t-1}$ and the input gate to the state update

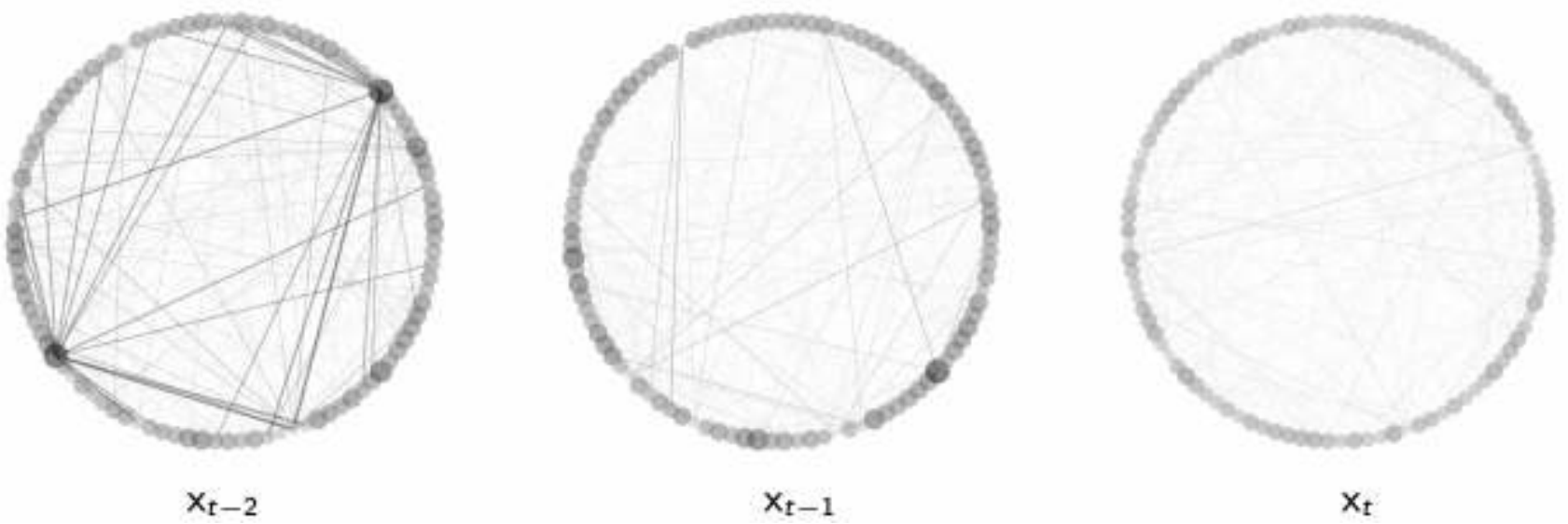
$$\boldsymbol{u}_t = f_t \boldsymbol{u}_{t-1} + g_t \sigma \left(\mathbf{A} \boldsymbol{x}_t + \mathbf{B} \boldsymbol{z}_{t-1} \right) \tag{2}$$

- State $\boldsymbol{z}_t$ updated by applying the cell output gate to the internal cell memory $\boldsymbol{u}_t$

$$\boldsymbol{z}_t = q_t \sigma(\boldsymbol{u}_t) \tag{3}$$

Graph Recurrent Neural Networks

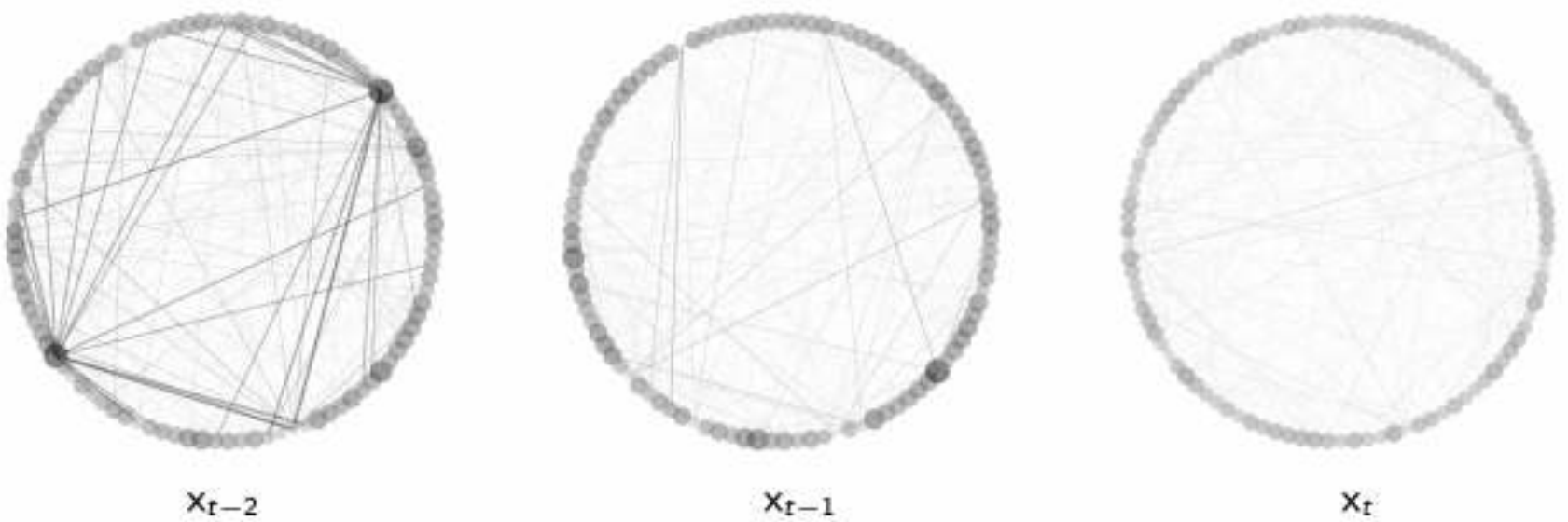
- Consider a time varying process $\boldsymbol{x}_t$ in which each of the signals is supported on shift operator $\mathbf{S}$.



- A graph recurrent neural network (GRNN) combines $\Rightarrow$ A GNN because $\boldsymbol{x}_t$ is supported on a graph. $\Rightarrow$ An RNN because $\boldsymbol{x}_t$ is a sequence.
- An RNN has a hidden state $\boldsymbol{z}_t$ updated with the perceptron $\Rightarrow$ $\boldsymbol{z}_t = \sigma\left(\mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{z}_{t-1}\right)$

Graph Recurrent Neural Networks

- Consider a time varying process $\boldsymbol{x}_t$ in which each of the signals is supported on shift operator $\mathbf{S}$.

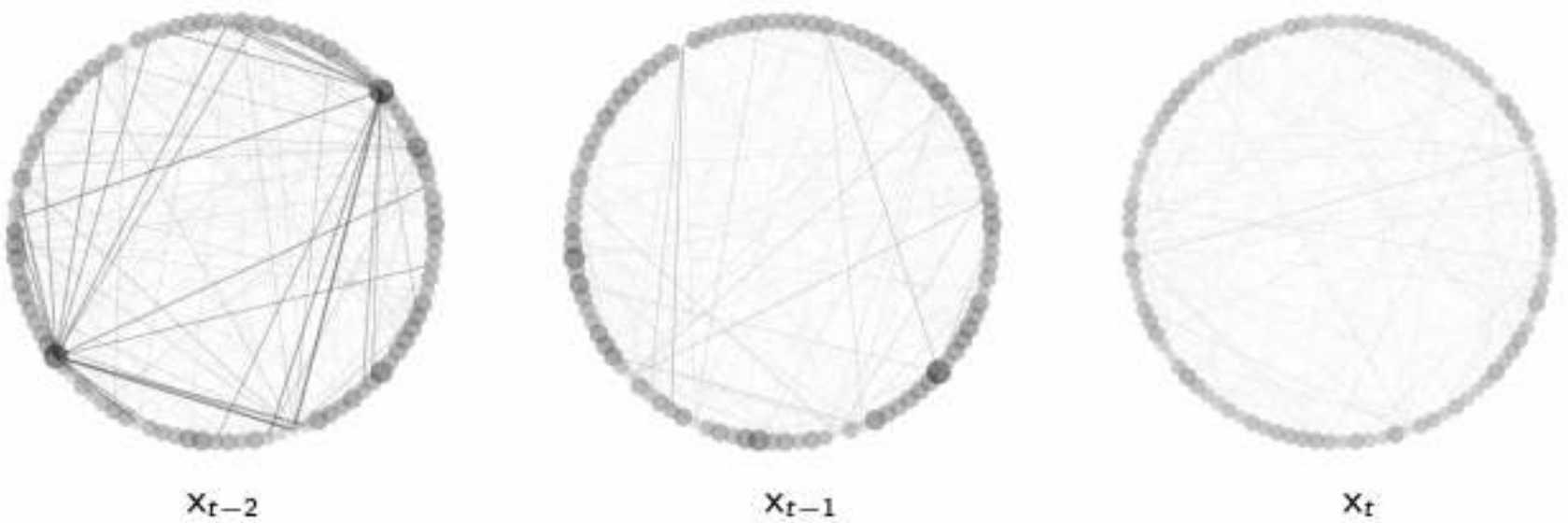


- A graph recurrent neural network (GRNN) combines
  $\Rightarrow$ A GNN because $\boldsymbol{x}_t$ is supported on a graph. $\Rightarrow$ An RNN because $\boldsymbol{x}_t$ is a sequence.
- An RNN has a hidden state $\boldsymbol{z}_t$ updated with the perceptron $\Rightarrow$ $\boldsymbol{z}_t = \sigma\left(\mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{z}_{t-1}\right)$

Graph Recurrent Neural Networks

Cornell University

- Consider a time varying process $\boldsymbol{x}_t$ in which each of the signals is supported on shift operator $\mathbf{S}$.



- A graph recurrent neural network (GRNN) combines
  $\Rightarrow$ A GNN because $\boldsymbol{x}_t$ is supported on a graph. $\Rightarrow$ An RNN because $\boldsymbol{x}_t$ is a sequence.
- An RNN has a hidden state $\boldsymbol{z}_t$ updated with the perceptron $\Rightarrow$
  $\boldsymbol{z}_t = \sigma\left(\mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{z}_{t-1}\right)$

Graph Recurrent Neural Networks

Cornell University

- Hidden and observed states propagate through graph filters to update the hidden state. The state update is

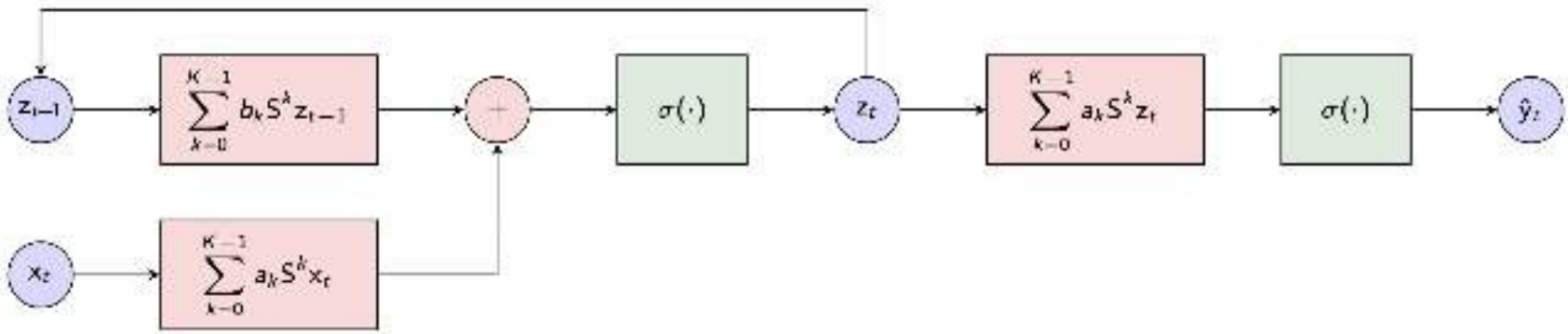
$$\Rightarrow \boldsymbol{z}_t = \sigma\left[\mathcal{A}(\mathbf{S})\boldsymbol{x}_t + \mathcal{B}(\mathbf{S})\boldsymbol{z}_{t-1}\right] = \sigma\left[\sum_{k=0}^{k-1} a_k\ \mathbf{S}^k\boldsymbol{x}_t + \sum_{k=0}^{k-1} b_k\ \mathbf{S}^k\boldsymbol{z}_{t-1}\right].$$

where

$$\mathcal{A}_{\mathbf{S}} = \mathcal{A}(\mathbf{S}) = \sum_{k=0}^{K-1} a_k\mathbf{S}^k \quad \mathcal{B}_{\mathbf{S}} = \mathcal{B}(\mathbf{S}) = \sum_{k=0}^{K-1} b_k\mathbf{S}^k.$$

- The prediction of the output $\hat{\boldsymbol{y}}_t$ is given by

$$\hat{\boldsymbol{y}}_t = \sigma\left[\mathbf{C}(\mathbf{S})\boldsymbol{z}_t\right] = \sigma\left[\sum_{k=0}^{K-1} c_k\ \mathbf{S}^k\boldsymbol{z}_t\right].$$



Graph Recurrent Neural Networks

Cornell University

- Hidden and observed states propagate through graph filters to update the hidden state. The state update is

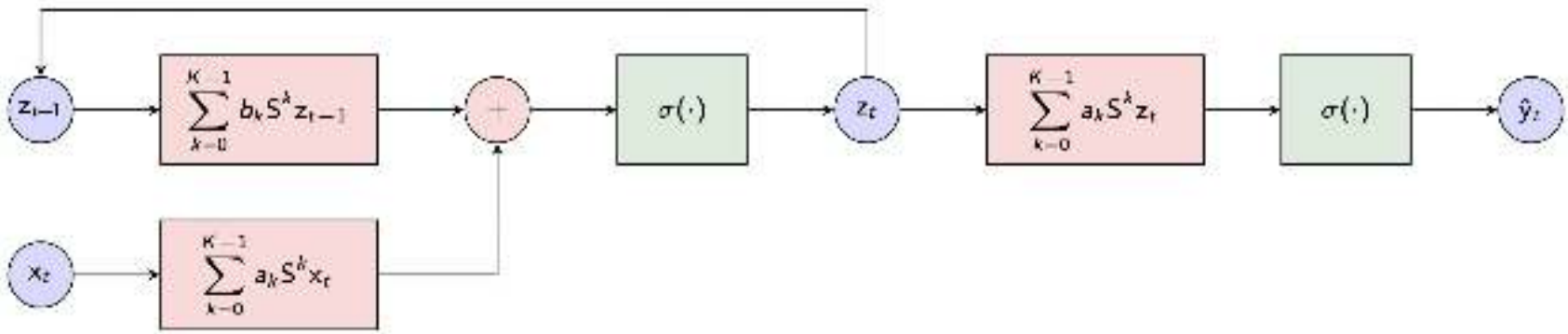
$$\Rightarrow \boldsymbol{z}_t = \sigma\left[\mathcal{A}(\mathbf{S})\boldsymbol{x}_t + \mathcal{B}(\mathbf{S})\boldsymbol{z}_{t-1}\right] = \sigma\left[\sum_{k=0}^{k-1} a_k\ \mathbf{S}^k\boldsymbol{x}_t + \sum_{k=0}^{k-1} b_k\ \mathbf{S}^k\boldsymbol{z}_{t-1}\right].$$

where

$$\mathcal{A}_{\mathbf{S}} = \mathcal{A}(\mathbf{S}) = \sum_{k=0}^{K-1} a_k\mathbf{S}^k \quad \mathcal{B}_{\mathbf{S}} = \mathcal{B}(\mathbf{S}) = \sum_{k=0}^{K-1} b_k\mathbf{S}^k.$$

- The prediction of the output $\hat{\boldsymbol{y}}_t$ is given by

$$\hat{\boldsymbol{y}}_t = \sigma\left[\mathbf{C}(\mathbf{S})\boldsymbol{z}_t\right] = \sigma\left[\sum_{k=0}^{K-1} c_k\ \mathbf{S}^k\boldsymbol{z}_t\right].$$



Graph Recurrent Neural Networks

- Hidden and observed states propagate through graph filters to update the hidden state. The state update is

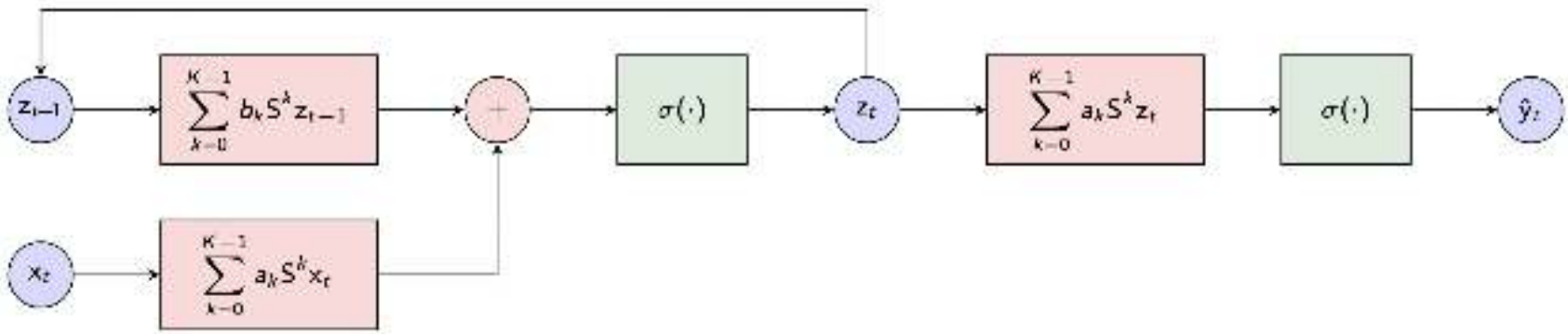
$$\Rightarrow \boldsymbol{z}_t = \sigma\left[\mathcal{A}(\mathbf{S})\boldsymbol{x}_t + \mathcal{B}(\mathbf{S})\boldsymbol{z}_{t-1}\right] = \sigma\left[\sum_{k=0}^{k-1} a_k\ \mathbf{S}^k\boldsymbol{x}_t + \sum_{k=0}^{k-1} b_k\ \mathbf{S}^k\boldsymbol{z}_{t-1}\right].$$

where

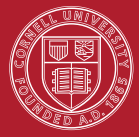
$$\mathcal{A}_{\mathbf{S}} = \mathcal{A}(\mathbf{S}) = \sum_{k=0}^{K-1} a_k\mathbf{S}^k \quad \mathcal{B}_{\mathbf{S}} = \mathcal{B}(\mathbf{S}) = \sum_{k=0}^{K-1} b_k\mathbf{S}^k.$$

- The prediction of the output $\hat{\boldsymbol{y}}_t$ is given by

$$\hat{\boldsymbol{y}}_t = \sigma\left[\mathbf{C}(\mathbf{S})\boldsymbol{z}_t\right] = \sigma\left[\sum_{k=0}^{K-1} c_k\ \mathbf{S}^k\boldsymbol{z}_t\right].$$



Outline



Cornell University

Gating in GRNNs

- We discuss long range graph dependencies and introduce node and edge gating
- Like RNNs, GRNNs may also experience the problem of vanishing/exploding gradients
- It happens when eigenvalues of $\mathbf{B}(\mathbf{S})$ are much smaller/larger than 1

$$\mathbf{Z}_t = \sigma\left(\hat{\mathcal{Q}}\left\{\mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right)\right\} + \check{\mathcal{Q}}\left\{\mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_{t-1}\right)\right\}\right) \tag{4}$$

where $\mathbf{X}_t \triangleq [\boldsymbol{x}_t^1, \cdots, \boldsymbol{x}_t^F]$.

- Input gate operator $\hat{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the input $\mathbf{X}_t$ at time t
- Forget gate operator $\check{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the state $\mathbf{Z}_t$ at time t

Gating in GRNNs

- We discuss long range graph dependencies and introduce node and edge gating
- Like RNNs, GRNNs may also experience the problem of vanishing/exploding gradients
- It happens when eigenvalues of $\mathbf{B}(\mathbf{S})$ are much smaller/larger than 1

$$\mathbf{Z}_t = \sigma\left(\hat{\mathcal{Q}}\left\{\mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right)\right\} + \check{\mathcal{Q}}\left\{\mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_{t-1}\right)\right\}\right) \tag{4}$$

where $\mathbf{X}_t \triangleq [\boldsymbol{x}_t^1, \cdots, \boldsymbol{x}_t^F]$.

- Input gate operator $\hat{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the input $\mathbf{X}_t$ at time t
- Forget gate operator $\check{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the state $\mathbf{Z}_t$ at time t

Gating in GRNNs

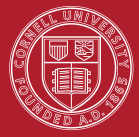
- We discuss long range graph dependencies and introduce node and edge gating
- Like RNNs, GRNNs may also experience the problem of vanishing/exploding gradients
- It happens when eigenvalues of $\mathbf{B}(\mathbf{S})$ are much smaller/larger than 1

$$\mathbf{Z}_t = \sigma\left(\hat{\mathcal{Q}}\left\{\mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right)\right\} + \check{\mathcal{Q}}\left\{\mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_{t-1}\right)\right\}\right) \tag{4}$$

where $\mathbf{X}_t \triangleq [\boldsymbol{x}_t^1, \cdots, \boldsymbol{x}_t^F]$.

- Input gate operator $\hat{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the input $\mathbf{X}_t$ at time t
- Forget gate operator $\check{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the state $\mathbf{Z}_t$ at time t

Gating in GRNNs



- We discuss long range graph dependencies and introduce node and edge gating
- Like RNNs, GRNNs may also experience the problem of vanishing/exploding gradients
- It happens when eigenvalues of $\mathbf{B}(\mathbf{S})$ are much smaller/larger than 1

$$\mathbf{Z}_t = \sigma\left(\hat{\mathcal{Q}}\left\{\mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right)\right\} + \check{\mathcal{Q}}\left\{\mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_{t-1}\right)\right\}\right) \tag{4}$$

where $\mathbf{X}_t \triangleq [\boldsymbol{x}_t^1, \cdots, \boldsymbol{x}_t^F]$.

- Input gate operator $\hat{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the input $\mathbf{X}_t$ at time t
- Forget gate operator $\check{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the state $\mathbf{Z}_t$ at time t

Gating in GRNNs

- We discuss long range graph dependencies and introduce node and edge gating
- Like RNNs, GRNNs may also experience the problem of vanishing/exploding gradients
- It happens when eigenvalues of $\mathbf{B}(\mathbf{S})$ are much smaller/larger than 1

$$\mathbf{Z}_t = \sigma\left(\hat{\mathcal{Q}}\left\{\mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right)\right\} + \check{\mathcal{Q}}\left\{\mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_{t-1}\right)\right\}\right) \tag{4}$$

where $\mathbf{X}_t \triangleq [\boldsymbol{x}_t^1, \cdots, \boldsymbol{x}_t^F]$.

- Input gate operator $\hat{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the input $\mathbf{X}_t$ at time t
- Forget gate operator $\check{\mathcal{Q}} : \mathbb{R}^{N\times H} \to \mathbb{R}^{N\times H} \Rightarrow$ controls the importance of the state $\mathbf{Z}_t$ at time t

Gating in GRNNs

- First type of gating for GRNNs is time gating $\Rightarrow$ simple extension of input and forget gates of RNNs
- In the Time-Gated GRNN, the input and forget gate operators are expressed as

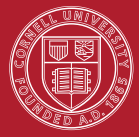
$$\hat{\mathcal{Q}}\left\{\mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right)\right\} = \hat{q}_t \mathcal{A}_{\mathbf{S}}\left(\mathbf{X}_t\right), \quad \check{\mathcal{Q}}\left\{\mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_t\right)\right\} = \check{q}_t \mathcal{B}_{\mathbf{S}}\left(\mathbf{Z}_t\right) \tag{5}$$

- Time gating multiplies the input and the state by scalar gates $\hat{q}_t \in [0,1]$ and $\check{q}_t \in [0,1]$.
- A single scalar gate is applied to the whole graph signal $\Rightarrow$ same gate value for all nodes

Gating in GRNNs

- Even if the eigenvalues of $\mathcal{B}(\mathbf{S}) \approx 1$, spatial imbalances can cause gradients to vanish in space
- Some nodes/paths might get assigned more importance than others in long range exchanges
- **Example**: graphs with community structure, where some nodes are highly connected within clusters
  - Gradients of $\mathbf{Z}_t$ depend on successive products of $\mathcal{B}(\mathbf{S}) \Rightarrow$ successive products of $\mathbf{S}$.
  - For large K, the matrix entries in $\mathbf{S}^K$ with highly connected nodes will get densely populated
  - Overshadows community structure $\Rightarrow$ can't encode long processes that are local on the graph

Gating in GRNNs



- Even if the eigenvalues of $\mathcal{B}(\mathbf{S}) \approx 1$, spatial imbalances can cause gradients to vanish in space
- Some nodes/paths might get assigned more importance than others in long range exchanges
- **Example**: graphs with community structure, where some nodes are highly connected within clusters
  - Gradients of $\mathbf{Z}_t$ depend on successive products of $\mathcal{B}(\mathbf{S}) \Rightarrow$ successive products of $\mathbf{S}$.
  - For large K, the matrix entries in $\mathbf{S}^K$ with highly connected nodes will get densely populated
  - Overshadows community structure $\Rightarrow$ can't encode long processes that are local on the graph

Spatial Gating

- Node and edge structure of the graph allows for other forms of gating $\Rightarrow$ spatial gating
- Node gating $\Rightarrow$ one input and one forget gate for each node of the graph

- Spatial gating strategies help encode long range spatial dependencies in graph processes

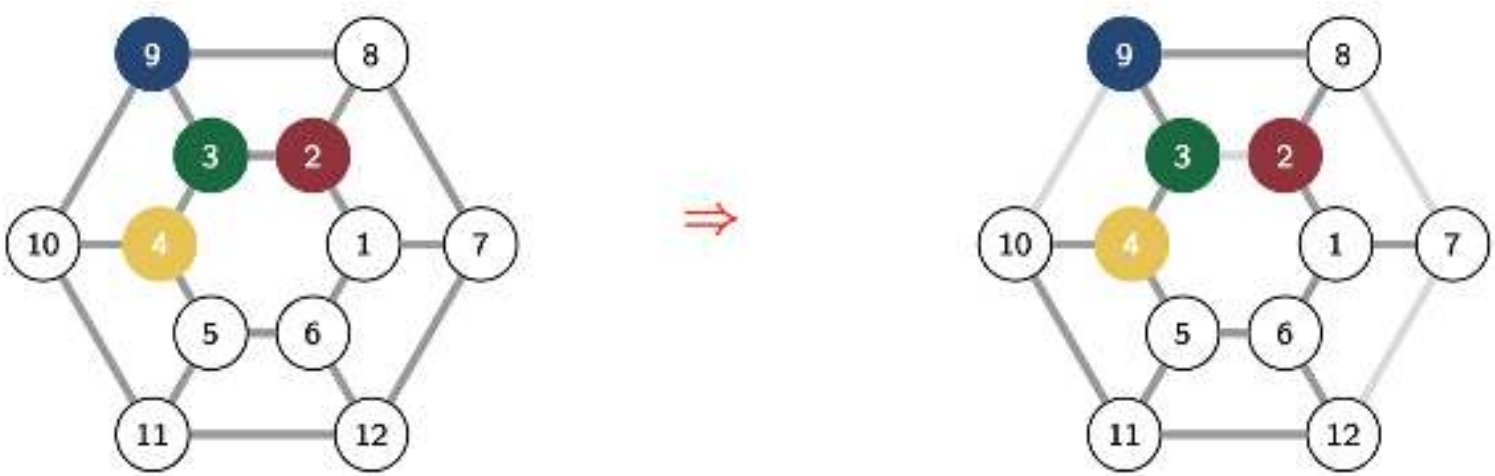
Spatial Gating

- Node and edge structure of the graph allows for other forms of gating $\Rightarrow$ spatial gating
- Node gating $\Rightarrow$ one input and one forget gate for each node of the graph

- Spatial gating strategies help encode long range spatial dependencies in graph processes

Spatial Gating

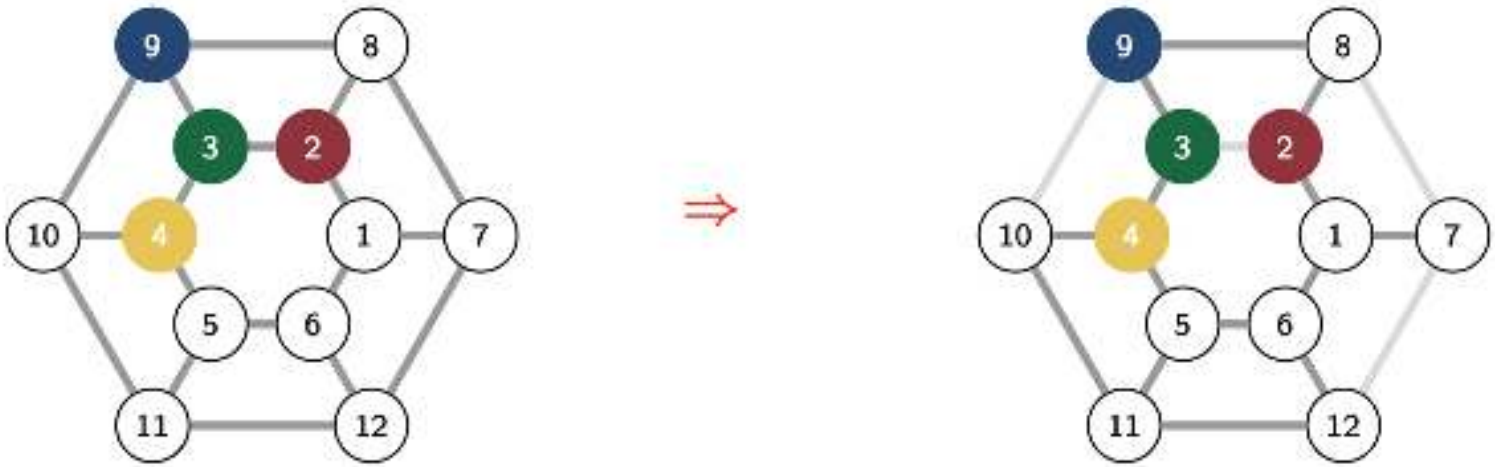
- Node and edge structure of the graph allows for other forms of gating $\Rightarrow$ spatial gating
- Edge gating $\Rightarrow$ one input and one forget gate for each edge of the graph



- Spatial gating strategies help encode long range spatial dependencies in graph processes

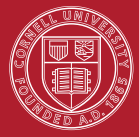
Spatial Gating

- Node and edge structure of the graph allows for other forms of gating $\Rightarrow$ spatial gating
- Edge gating $\Rightarrow$ one input and one forget gate for each edge of the graph



- Spatial gating strategies help encode long range spatial dependencies in graph processes

Conclusion



- We introduce Chebyshev spectral graph convolutional neural networks.
- We extend it to graph filter banks and multiple GNNs.
- We also introduce the Graph Recurrent Neural Networks, and spatial gating.
- We will introduce the application of Graph neural networks and the industry that is using these algorithms next week.

$$p\Big(\tilde{G}_i \mid X_i, Y_i, A_i, \theta_{1:K}\Big) = \prod_{k=1}^{K} \mathcal{N}\Big(\tilde{g}_{k,i};, h_k(X_i, Y_i, a_{k,i}; \theta_k),, \sigma^2 I_p\Big),$$