Seminar Series on Graph Neural Networks 03

A graph signal processing viewpoint of graph neural networks

Yong-Min Shin
School of Mathematics and Computing (Computational Science and Engineering)
Yonsei University

2025.04.14

Before going in….

Towards application of graph neural networks

Towards efficient graph learning

Explainable graph neural networks

Fundamental topics on graph neural networks

On the representational power of graph neural networks

A graph signal processing viewpoint of graph neural networks

From label propagation to graph neural networks

On the problem of oversmoothing and oversquashing

Introduction to graph mining and graph neural networks
(Basic overview to kick things off)

* Presentation slides are available at:
(jordan7186.github.io/presentations/)

Objectives

1. Preliminary: Singal processing (3blue1brown)
2. Understanding of **graph signals & graph Fourier transform**
3. Understanding the **formulation of ChebNet**
4. **Re-reading GCN, understanding in the original author's way**

Preliminary: Signal processing

Signal processing & filtering

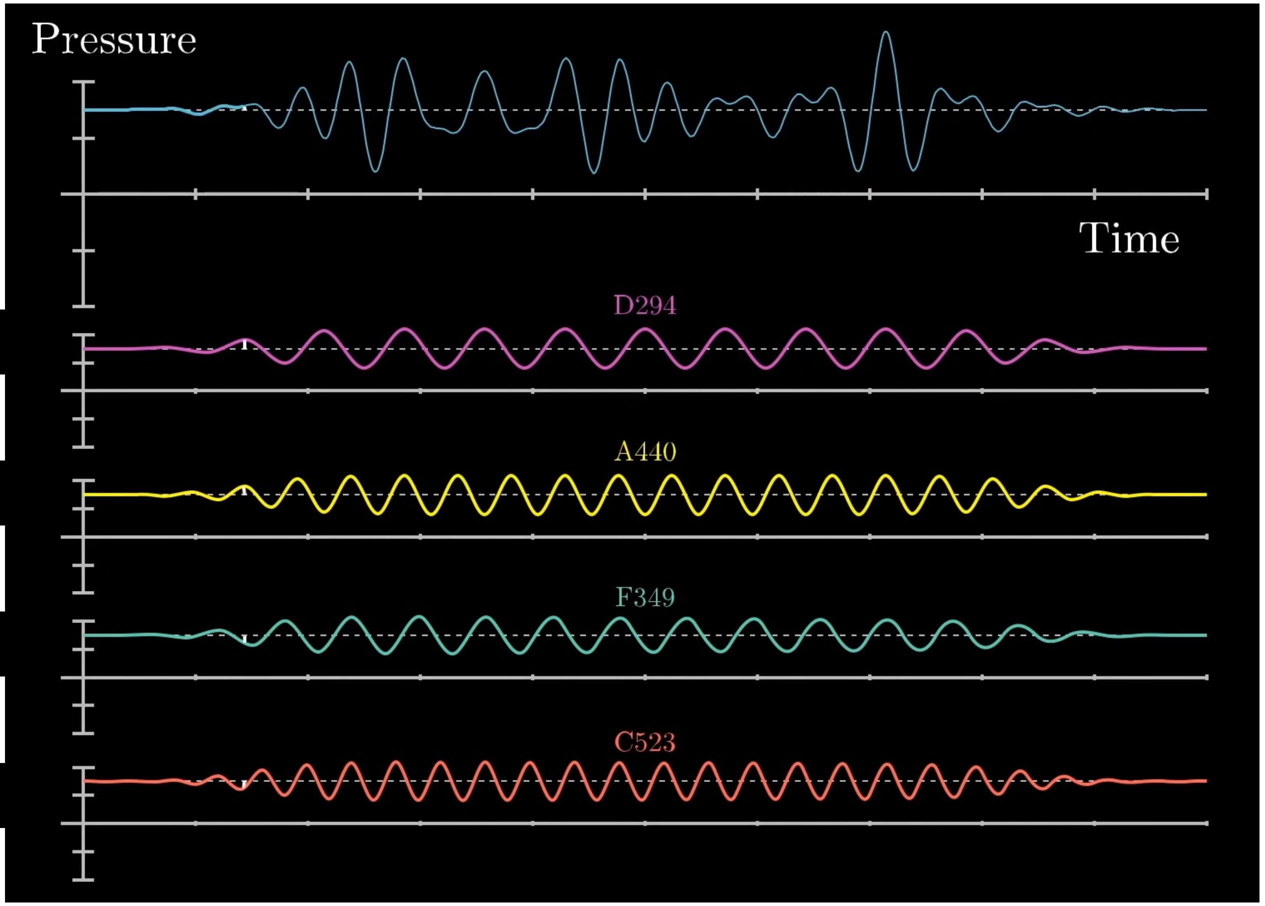
Observed signal

Signal 1

Signal 2

Signal 3

Signal 4



Fourier transforms can be used to analyze signals

Observed signal with multiple frequencies

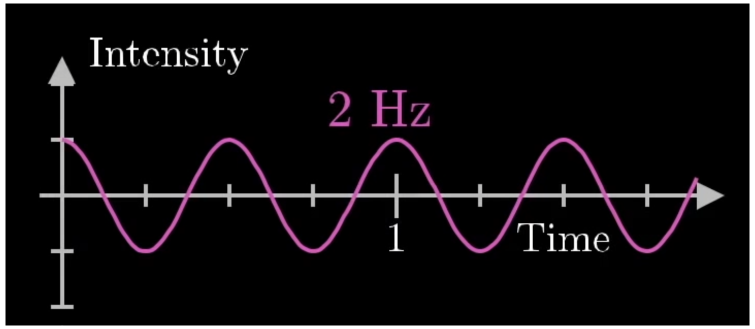
⬍

Individual signals with single frequencies

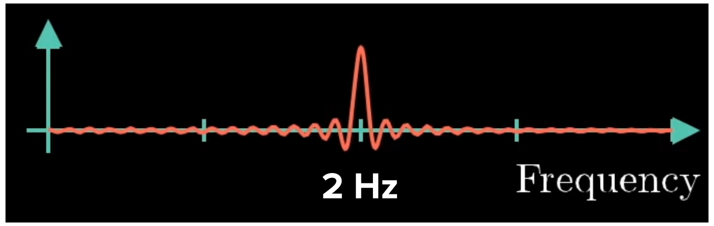
Signal processing & filtering

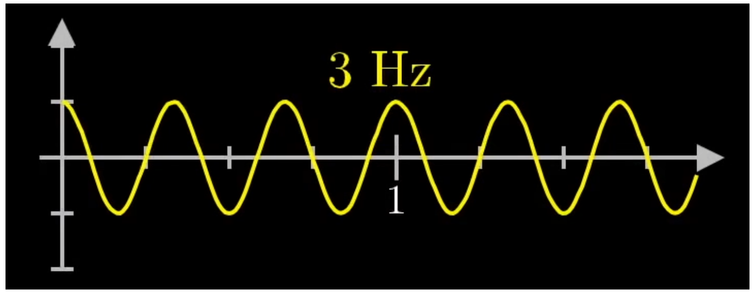
Time domain

Frequency domain

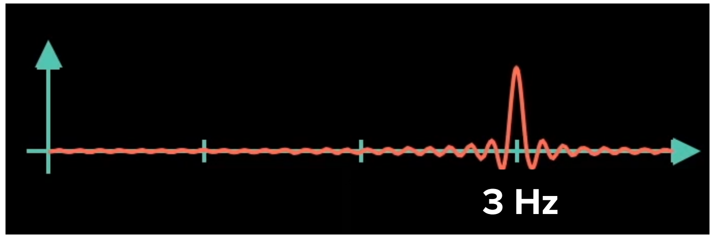


Fourier transform $\longrightarrow$





Fourier transform $\longrightarrow$



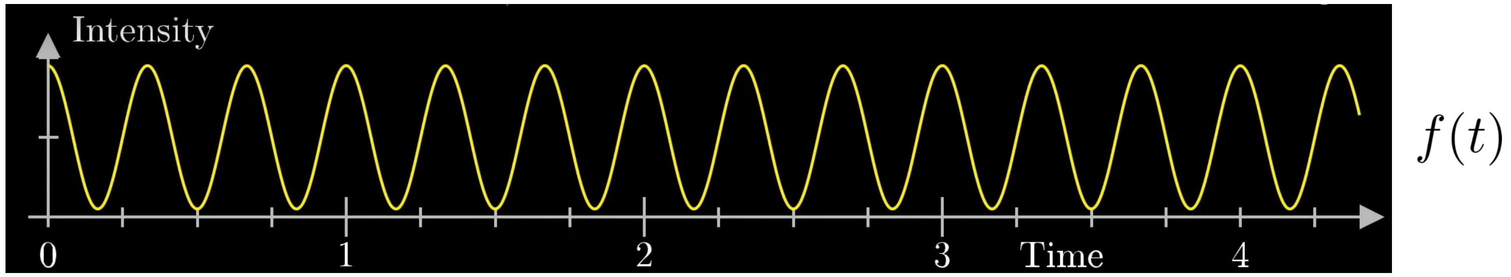
$f(t)$

$\hat{f}(\zeta)$

$$\hat{f}(\zeta) = \int_{\mathbb{R}} f(t) e^{-2\pi i \zeta t} dt$$

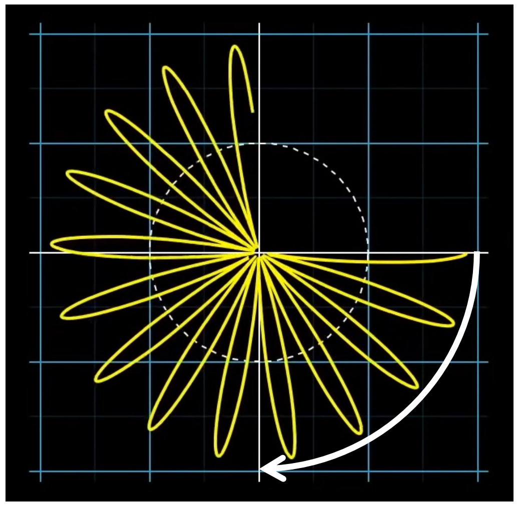
Signal processing & filtering

3 beats / sec

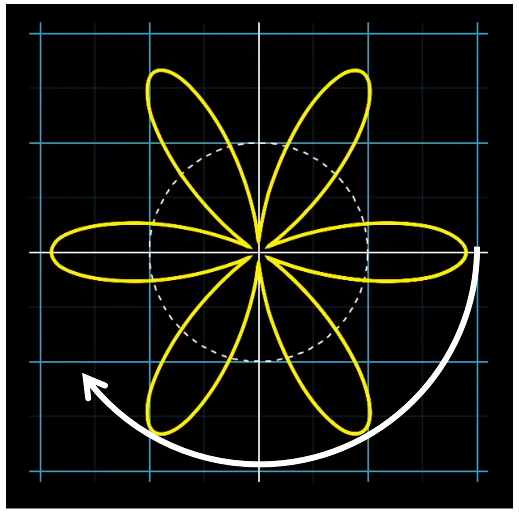


Imagine winding the wave in a circle in a 2D plane in different frequencies

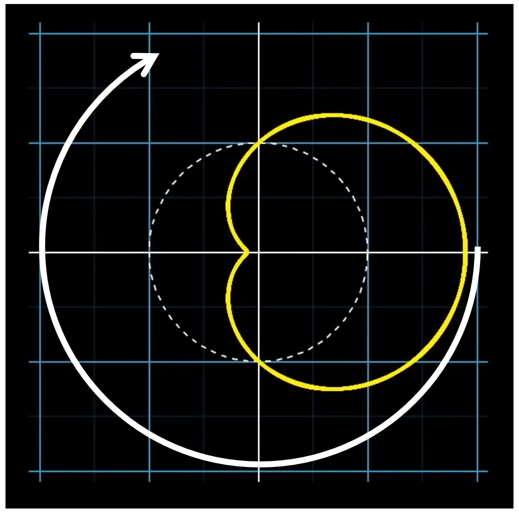
0.20 cycles / sec



0.5 cycles / sec



3 cycles / sec



Observation

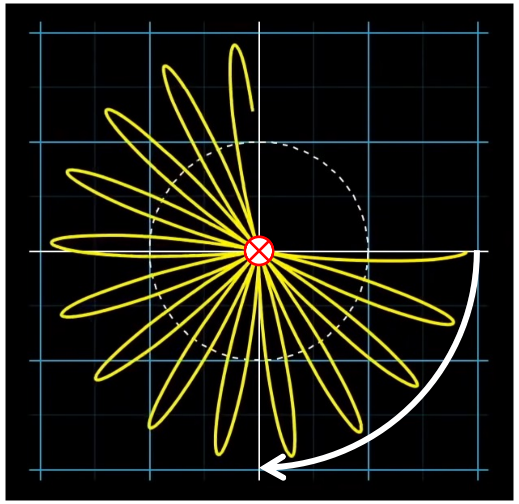
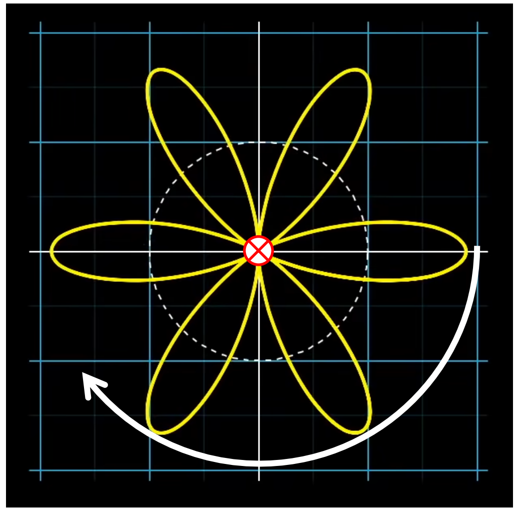
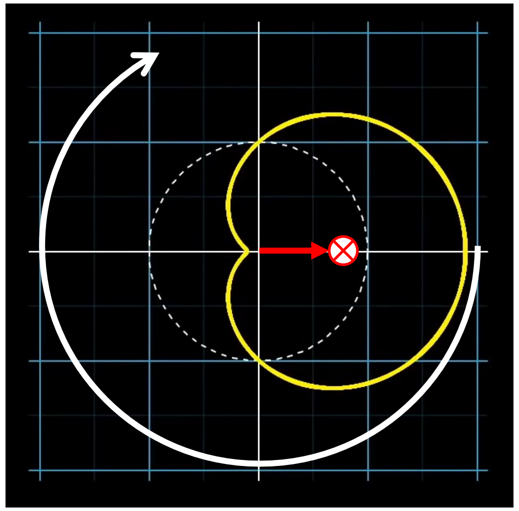
Something unique happens when the winding frequency *exactly* matches the signal frequency

How to measure this?

Signal processing & filtering

⊗ : **Center of mass**

Measurement of the distance between the origin and **center of mass**

Imagine **winding the wave in a circle** in a 2D plane in different **frequencies**

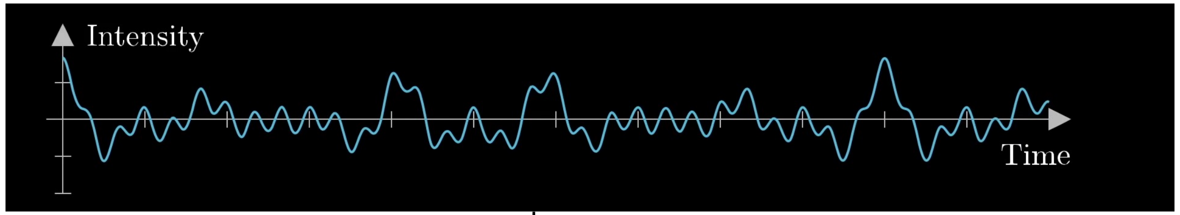
$$\boxed{f(t)}e^{-2\pi i\zeta t}$$

Measurement of the distance between the origin and **center of mass**

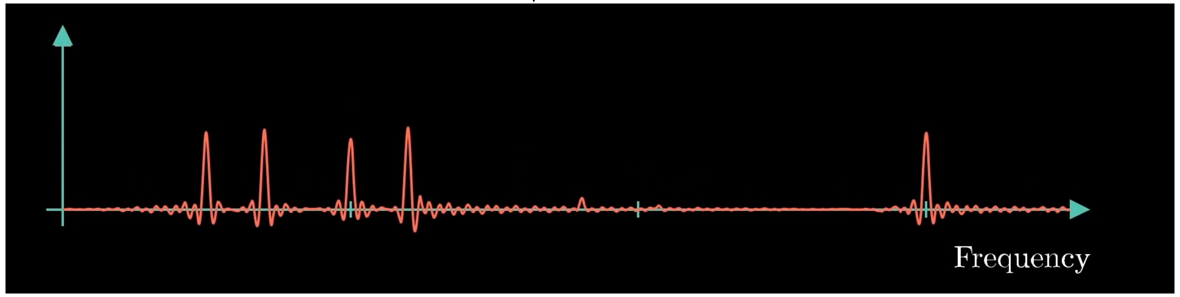
$$\boxed{\frac{1}{Z}\int_{\mathbb{R}}} f(t)e^{-2\pi i\zeta t}\boxed{dt}$$

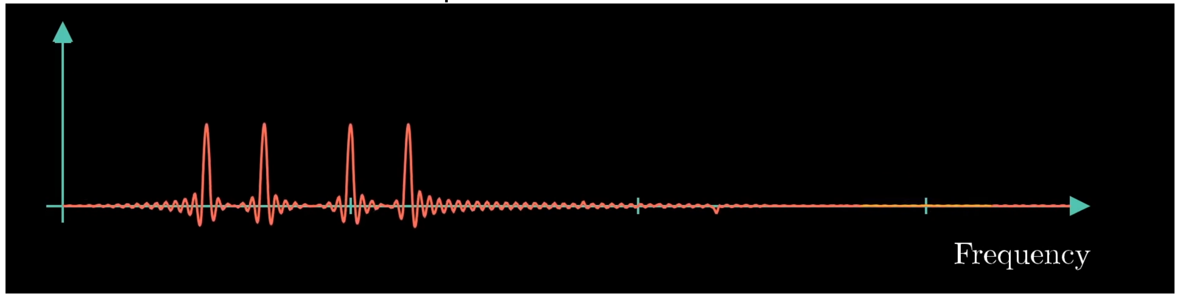
$$\hat{f}(\zeta) = \int_{\mathbb{R}} f(t)e^{-2\pi i\zeta t}dt$$

Low-pass filtering

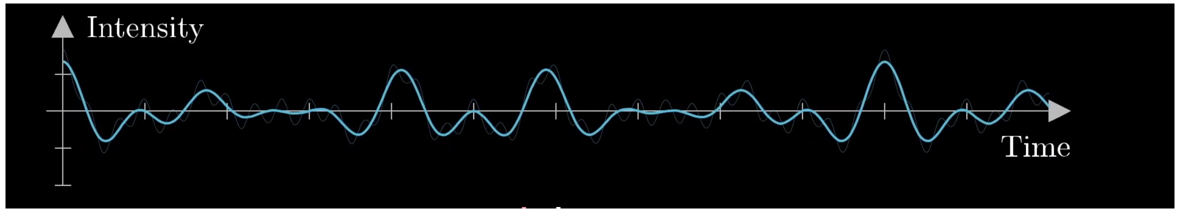


1. Fourier transform



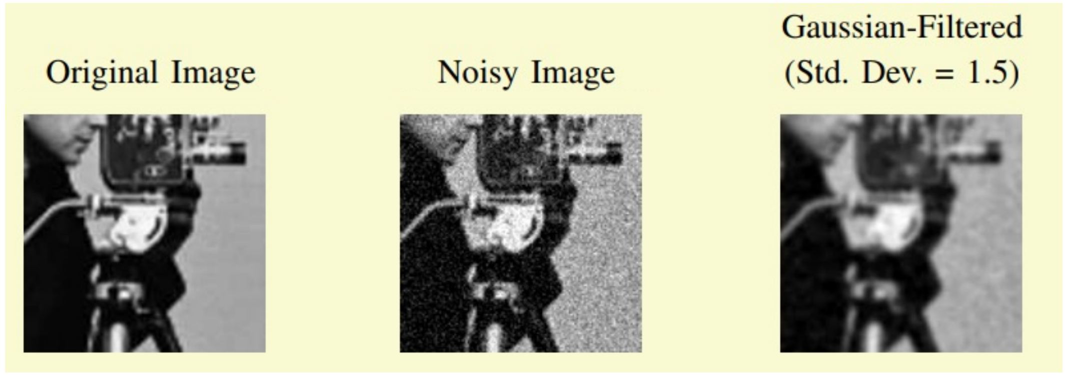


3. Inverse Fourier transform



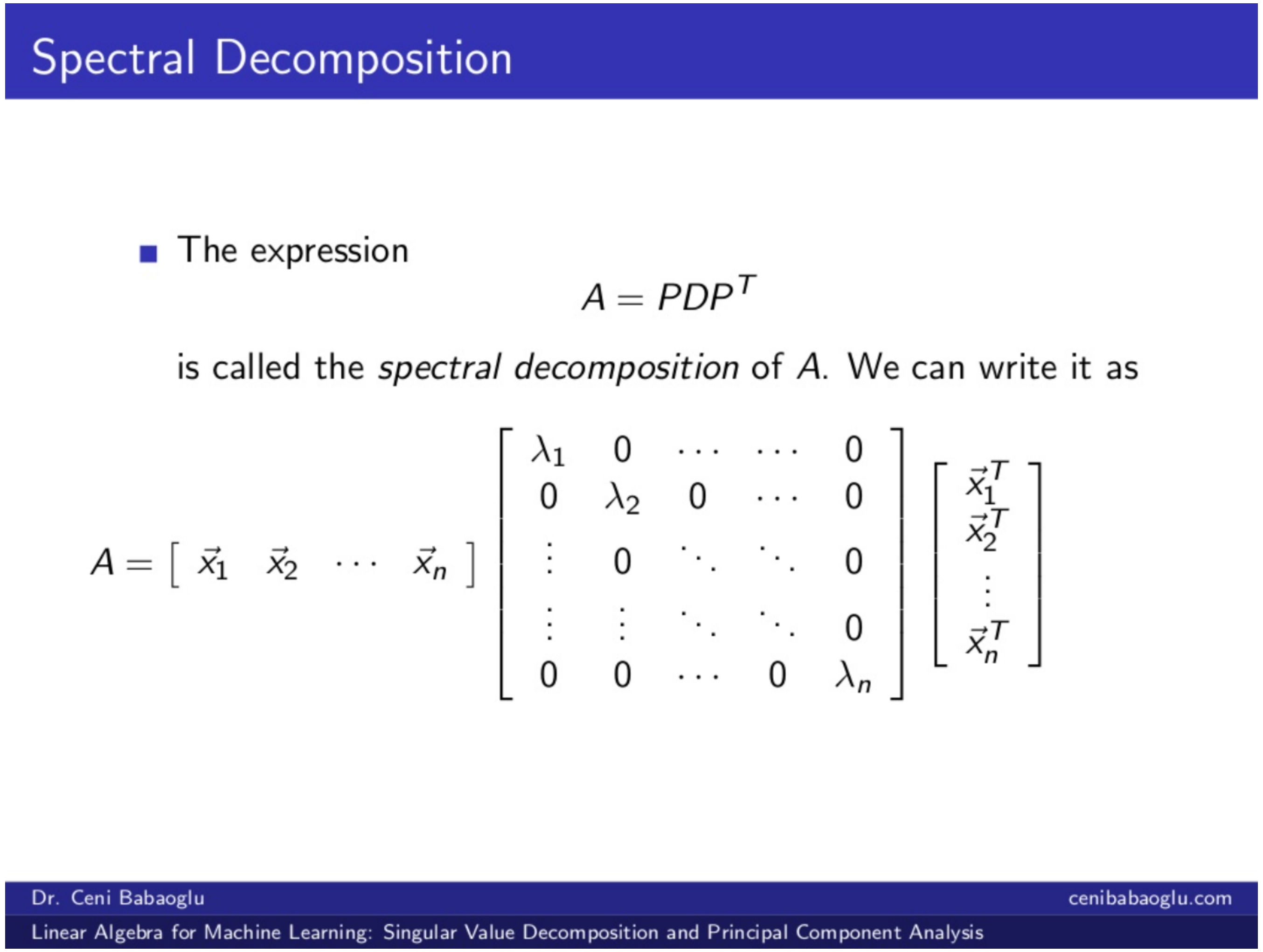
2. Reduce high-frequency noise

**David I Shuman et al., The Emerging Field of Signal Processing on Graphs*



3Blue1Brown, https://www.youtube.com/watch?v=spUNpyF58BY

Additional linear algebra: Spectral decomposition



Spectral Decomposition

- The expression

$$A = PDP^T$$

is called the *spectral decomposition* of A. We can write it as

$$A = \begin{bmatrix} \vec{x}_1 & \vec{x}_2 & \cdots & \vec{x}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \lambda_n \end{bmatrix} \begin{bmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \vdots \\ \vec{x}_n^T \end{bmatrix}$$

Dr. Ceni Babaoglu cenibabaoglu.com

Linear Algebra for Machine Learning: Singular Value Decomposition and Principal Component Analysis

- When decomposing the (symmetric) adjacency matrix, the eigenvalues are real.
- The eigenvalues are usually ordered.

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$$

We can also rearrange in an ascending order, just swap the corresponding eigenvectors accordingly.

Slide from: https://www.slideshare.net/CeniBabaogluPhDinMat/5-linear-algebra-for-machine-learning-singular-value-decomposition-and-principal-component-analysis-147828329

Understanding of graph signals & graph Fourier transforms

Shumann et al., The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains, IEEE Signal Process. Mag. 30(3): 83-98 (2013)

How should we think of signals on graphs?

We will consider a simple 1D node features, which can be easily extended to muti-dimensional case

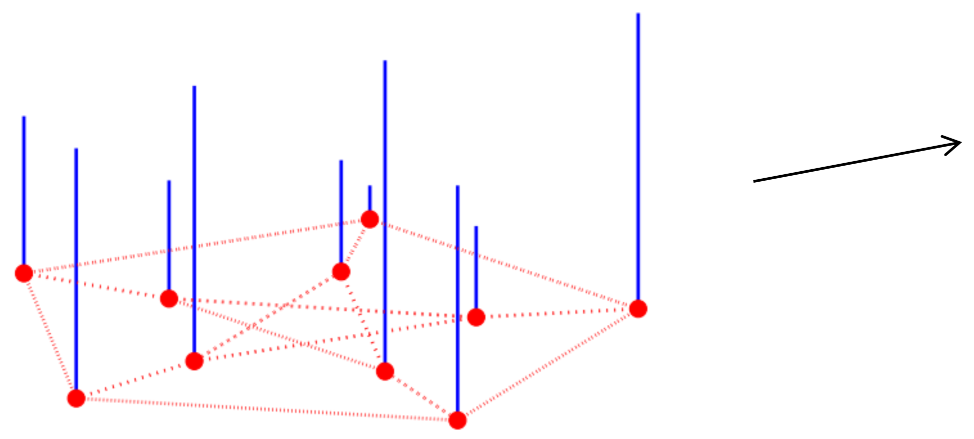
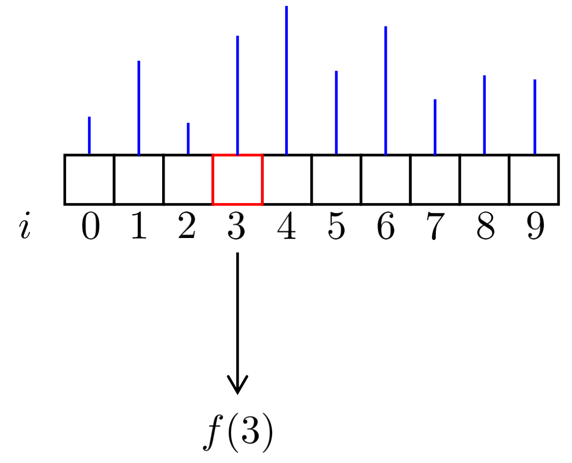
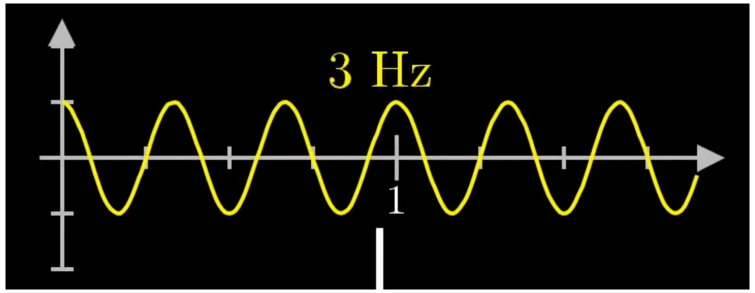


Fig. 1. A random positive graph signal on the vertices of the Petersen graph. The height of each blue bar represents the signal value at the vertex where the bar originates.

We can imagine assigning a single value to each vertex
Think of 1-dimensional feature matrix as a function





Generalization of Fourier transforms to graphs

A GRAPH FOURIER TRANSFORM AND NOTION OF FREQUENCY

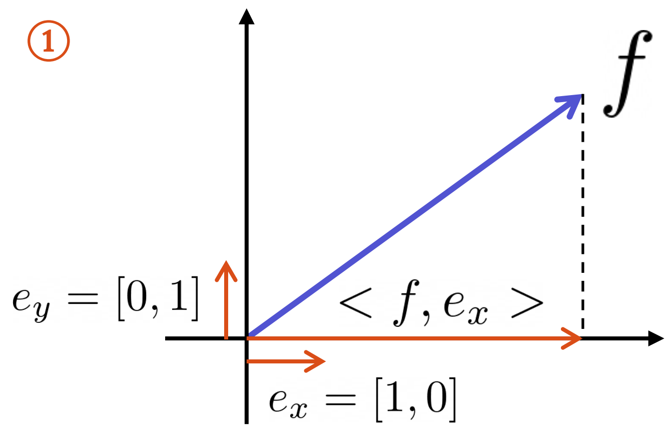
The classical Fourier transform

$$\hat{f}(\xi) := \langle f, e^{2\pi i \xi t} \rangle = \int_{\mathbb{R}} f(t) e^{-2\pi i \xi t} dt$$

is the ① expansion of a function f in terms of the ② complex exponentials, which are the eigenfunctions of the one-dimensional (1-D) Laplace operator ③

$$-\Delta (e^{2\pi i \xi t}) = -\frac{\partial^2}{\partial t^2} e^{2\pi i \xi t} = (2\pi\xi)^2 e^{2\pi i \xi t}. \quad (2)$$

①



②

Fourier transform = Inner product with some function
Some function = complex expoenetials?

③

Some function = <u>Eigenfunction</u> of the Laplace operator

$$\Delta = \frac{\partial^2}{\partial t^2} \quad \Big| \quad \Delta \boxed{e^{2\pi i \xi t}} = -(2\pi\xi)^2 \boxed{e^{2\pi i \xi t}}$$

Eigenfunction

Roughly, the Laplace operator measures the **local difference between the function and average**.

Generalization of Fourier transforms to graphs

Conclusion of the previous slide:

Fourier transform is the inner product between <u>the target function</u> and the <u>eigenfunction of the Laplace operator</u>.

Graph signal!
(# of nodes = N)

Analogously, we can define the graph Fourier transform $\hat{\mathrm{f}}$ of any function $\boxed{\mathrm{f} \in \mathbb{R}^N}$ on the vertices of $\mathcal{G}$ as the expansion of f in terms of the eigenvectors of the graph Laplacian:

$$\underline{\hat{f}(\lambda_\ell) := \langle \mathrm{f}, \mathbf{u}_\ell \rangle} = \underline{\sum_{i=1}^{N} f(i) u_\ell^*(i)}. \qquad (3)$$

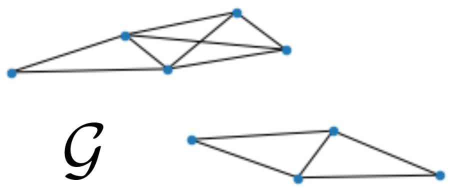
This part is the same from before

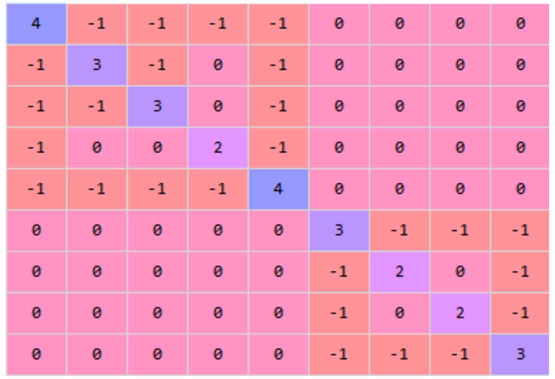
This part is different because now its discrete

Graph Laplacian

Laplace operator measures the **local difference between the function and average**.

$$\mathcal{L} = \mathbf{D} - \mathbf{A}$$





4	-1	-1	-1	-1	0	0	0	0
-1	3	-1	0	-1	0	0	0	0
-1	-1	3	0	-1	0	0	0	0
-1	0	0	2	-1	0	0	0	0
-1	-1	-1	-1	4	0	0	0	0
0	0	0	0	0	3	-1	-1	-1
0	0	0	0	0	-1	2	0	-1
0	0	0	0	0	-1	0	2	-1
0	0	0	0	0	-1	-1	-1	3

And we can just get the eigenvectors of the graph Laplacian via spectral decomposition.

Complementary note 1: On the physical intuition of the graph Laplacian

Why is Laplacian related to measuring the difference between the function and average?

Need some further generalizations of mathematical concepts… "Discrete calculus"

Definition 1. *Edge derivative* at *e=(i, j)*

$$\left.\frac{\partial f}{\partial e}\right|_i := f(j) - f(i)$$

$$\Downarrow$$

Definition 2. *Graph gradient* at vertex *i*

$$\nabla_i f := \left[\left\{ \left.\frac{\partial f}{\partial e}\right|_i \right\}_{\text{for some edges from } i} \right]$$

$$\Downarrow$$

Definition 3. *Local variation* at vertex *i*

$$||\nabla_i f||_2 := \left[\sum_{\text{for some edges from } i} \left(\left.\frac{\partial f}{\partial e}\right|_i \right)^2 \right]^{1/2}$$

$$= \left[\sum_{j \in \mathcal{N}_i} (f(j) - f(i))^2 \right]^{1/2}$$

We can calculate the *total variation of the whole graph* as *the sum of local variation (squared) for all nodes in the graph:*

$$\frac{1}{2} \sum_{i \in V} \sum_{j \in \mathcal{N}_i} (f(j) - f(i))^2$$

which is

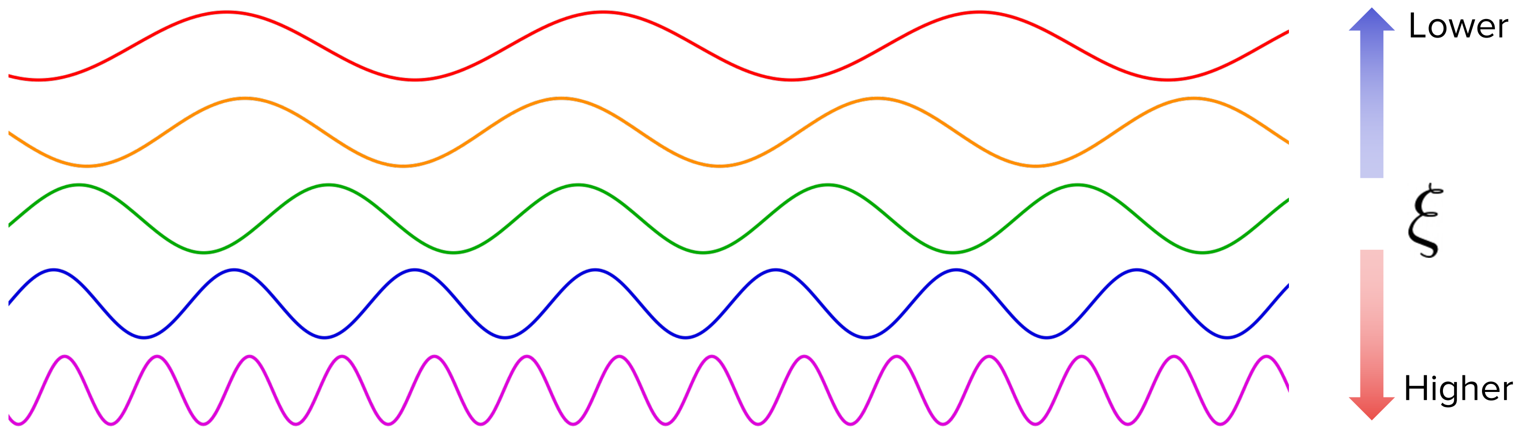
$$f^T \mathcal{L} f$$

….So we can at least understand why the graph Laplacian is useful to capture the patterns of the graph signal. It measures how much the signal differs locally, eventually containing all information on variation of signals.

Complementary note 2: Analogy of eigenfunctions across frequencies

Used to capture macro-patterns

Eigenfunction represents slow-oscillating patterns (**low frequency**)



Eigenfunction represents fast-oscillating patterns (**high frequency**)

Used to capture micro-patterns (details / noise)

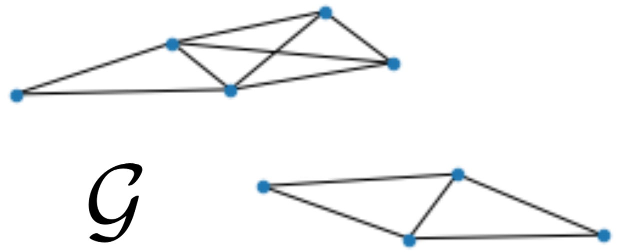
$$e^{2\pi i \boxed{\xi} t}$$

The eigenfunction of the Laplacian

Is this also analogous to graph Fourier basis?

Complementary note 2: Analogy of eigenfunctions across frequencies

Concrete example 1



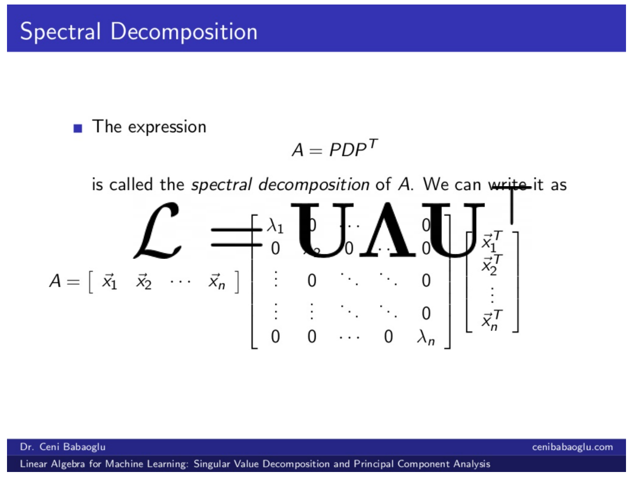
$\mathcal{G}$

$A_1 \in \mathbb{R}^{9\times 9}$

0	1	1	1	1	0	0	0	0
1	0	1	0	1	0	0	0	0
1	1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0
1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	1	1	0

$\mathcal{L}_1 \in \mathbb{R}^{9\times 9}$

4	-1	-1	-1	-1	0	0	0	0
-1	3	-1	0	-1	0	0	0	0
-1	-1	3	0	-1	0	0	0	0
-1	0	0	2	-1	0	0	0	0
-1	-1	-1	-1	4	0	0	0	0
0	0	0	0	0	3	-1	-1	-1
0	0	0	0	0	-1	2	0	-1
0	0	0	0	0	-1	0	2	-1
0	0	0	0	0	-1	-1	-1	3



Spectral Decomposition

- The expression

$$A = PDP^T$$

is called the *spectral decomposition* of A. We can write it as

$$\mathcal{L} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top$$

$$A = \begin{bmatrix} \vec{x}_1 & \vec{x}_2 & \cdots & \vec{x}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \lambda_n \end{bmatrix} \begin{bmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \vdots \\ \vec{x}_n^T \end{bmatrix}$$

Dr. Ceni Babaoglu cenibabaoglu.com
Linear Algebra for Machine Learning: Singular Value Decomposition and Principal Component Analysis

$\mathbf{U}$

							$\mathbf{u}_1$	$\mathbf{u}_0$
-2.32e-17	-0.894	-2.76e-18	0	0	0	-1.35e-16	0	0.447
-0.289	0.224	0.707	0	0	0	0.408	0	0.447
-0.289	0.224	-0.707	0	0	0	0.408	0	0.447
-0.289	0.224	-5.67e-16	0	0	0	-0.816	0	0.447
0.866	0.224	2.8e-16	0	0	0	-4.25e-16	0	0.447
0	0	0	-0.866	1.02e-16	4.96e-17	0	0.5	0
0	0	0	0.289	0.408	-0.707	0	0.5	0
0	0	0	0.289	0.408	0.707	0	0.5	0
0	0	0	0.289	-0.816	-1.6e-17	0	0.5	0

Matrix of eigenvectors

Corresponding eigenvalues

$\boldsymbol{\Lambda}$

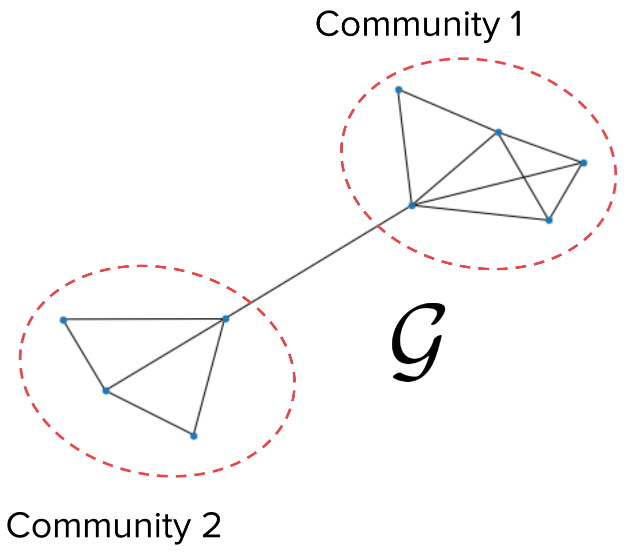
25
25
16
16
16
4
4
0
0

Lowest two frequencies *(eigenvalues)*

Notice that the trivial eigenpair reveal the most macro pattern in a graph: Number of connected compoents ("blobs")

Complementary note 2: Analogy of eigenfunctions across frequencies

Concrete example 2



1) Notice that there is only one trivial eigenpair since the graph is one giant connected component

							$\mathbf{u}_1$	$\mathbf{u}_0$
-0.153	0.866	-0.125	-1.16e-16	-1.06e-16	-4.25e-17	-9.27e-17	0.316	0.333
-0.153	-0.289	-0.125	0.599	0.376	0.116	0.391	0.316	0.333
-0.153	-0.289	-0.125	-0.599	-0.376	0.116	0.391	0.316	0.333
-0.153	-0.289	-0.125	-2.79e-16	3.94e-16	-0.233	-0.783	0.316	0.333
0.817	-3.39e-15	0.421	8.45e-16	5.95e-16	-8.86e-18	-9.88e-17	0.212	0.333
-0.465	-4.44e-15	0.775	1.33e-15	1.05e-15	0	5.55e-17	-0.27	0.333
0.0873	1.35e-15	-0.231	0.217	-0.346	0.678	-0.201	-0.401	0.333
0.0873	1.37e-15	-0.231	0.217	-0.346	-0.678	0.201	-0.401	0.333
0.0873	1.34e-15	-0.231	-0.434	0.692	3.56e-16	5.62e-16	-0.401	0.333

$\mathbf{U}$

40
25
18.9
16
16
4
4
0.107
0

$\mathbf{\Lambda}$

Lowest two frequencies *(eigenvalues)*

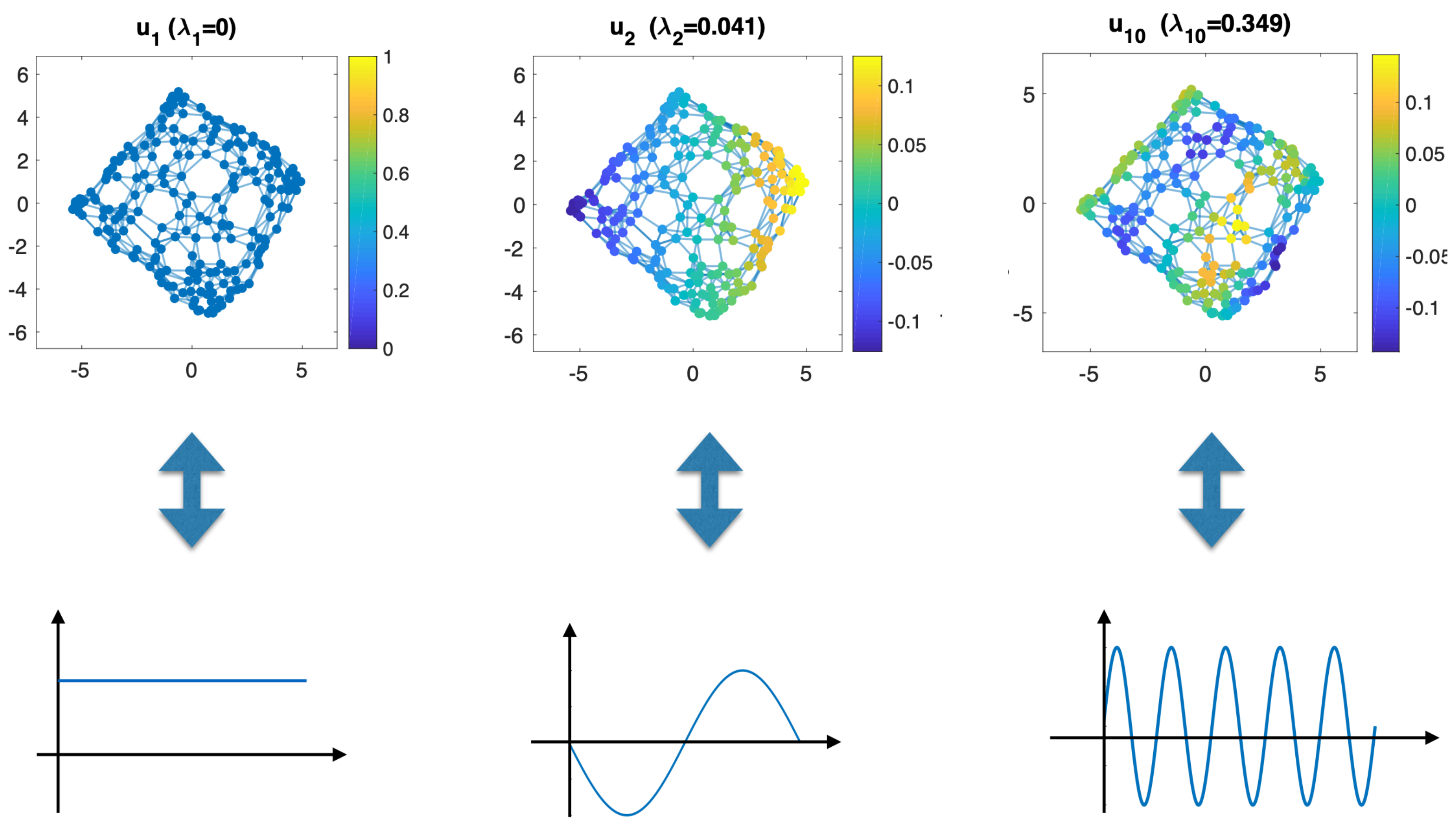
2) Still, also notice that the first non-trivial eigenvectors returns a 'soft' community assignment, which is the the next macro-pattern.
(HIGHLY incourage to read spectral clustering [1])

[1] Luxburg, A Tutorial on Spectral Clustering

Complementary note 2: Analogy of eigenfunctions across frequencies

Some further examples... (external slide, see below for reference)

Fourier Bases on Graphs



Fourier Bases on Graphs

- The "complex exponentials" in the graph domain are the eigenvectors of L: [Shuman, 2013]

$$Lu_k = \lambda_k u_k \quad \text{for} \quad k = 1, \ldots, N$$

- Compare:

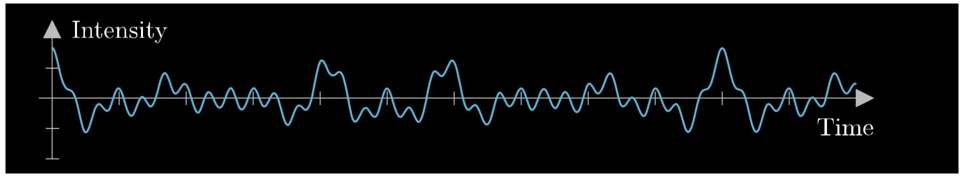
$$-\Delta(e^{j\Omega t}) = \underbrace{\boxed{\Omega^2}}_{\text{frequency}} e^{j\Omega t} \longleftrightarrow Lu_k = \underbrace{\boxed{\lambda_k}}_{\text{frequency}} u_k$$

- As λ_k increases, u_k varies more rapidly on the graph.

Slides from: Elif Vural, "Spectral Graph Theory and Graph Signal Processing", 2021, https://indico.truba.gov.tr/event/56/contributions/456/attachments/118/280/CizgeOkulu_ElifVural.pdf

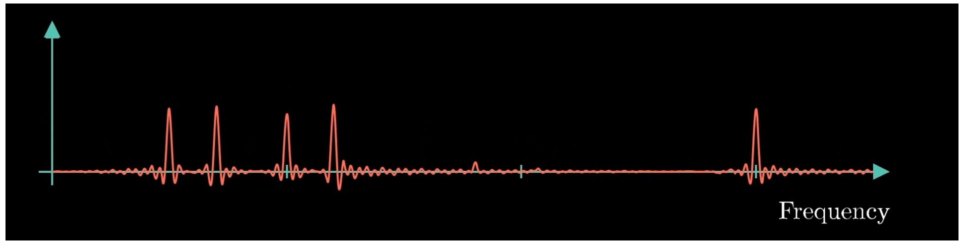
…Coming back, completing the full graph Fourier transformation

Original domain (time)



$f(i)$

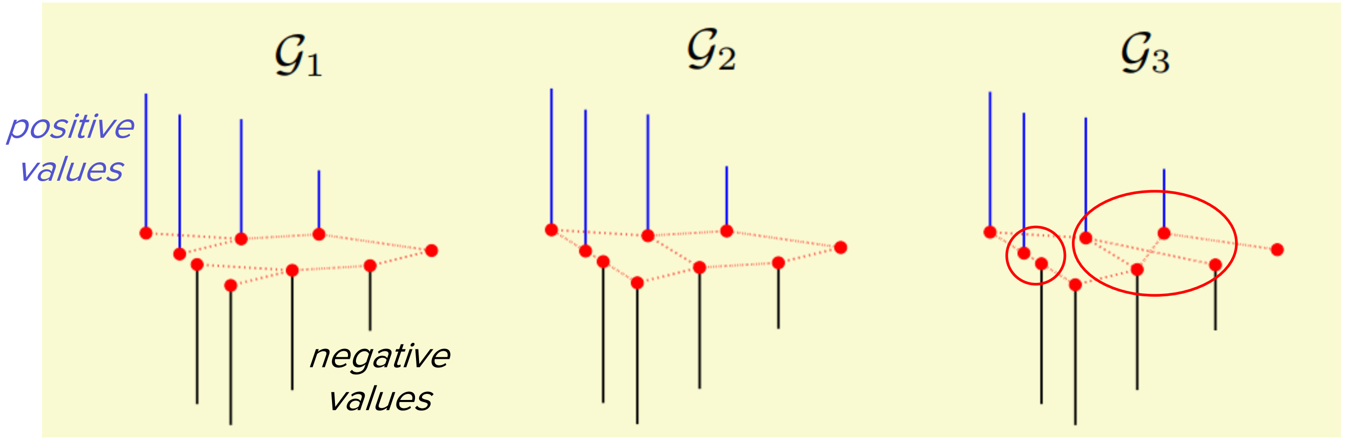
Fourier transform



$\hat{f}(\lambda_l)$

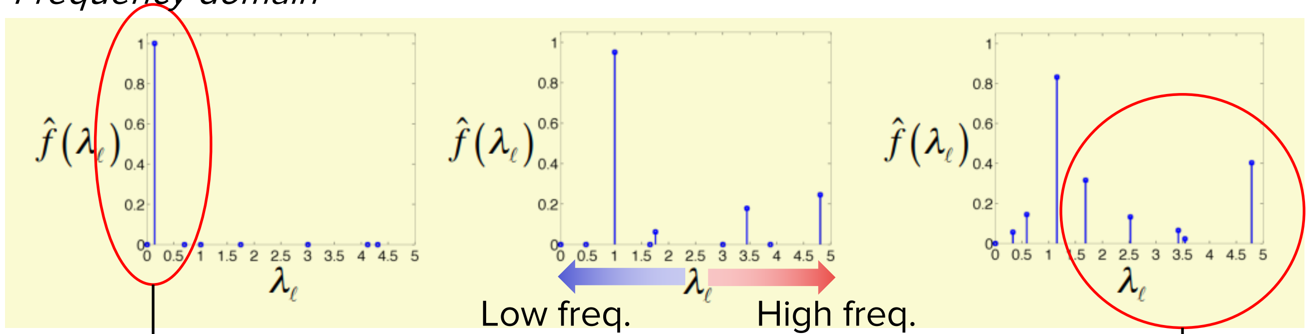
Frequency domain

Original domain (vertex)



Fourier transform

Frequency domain



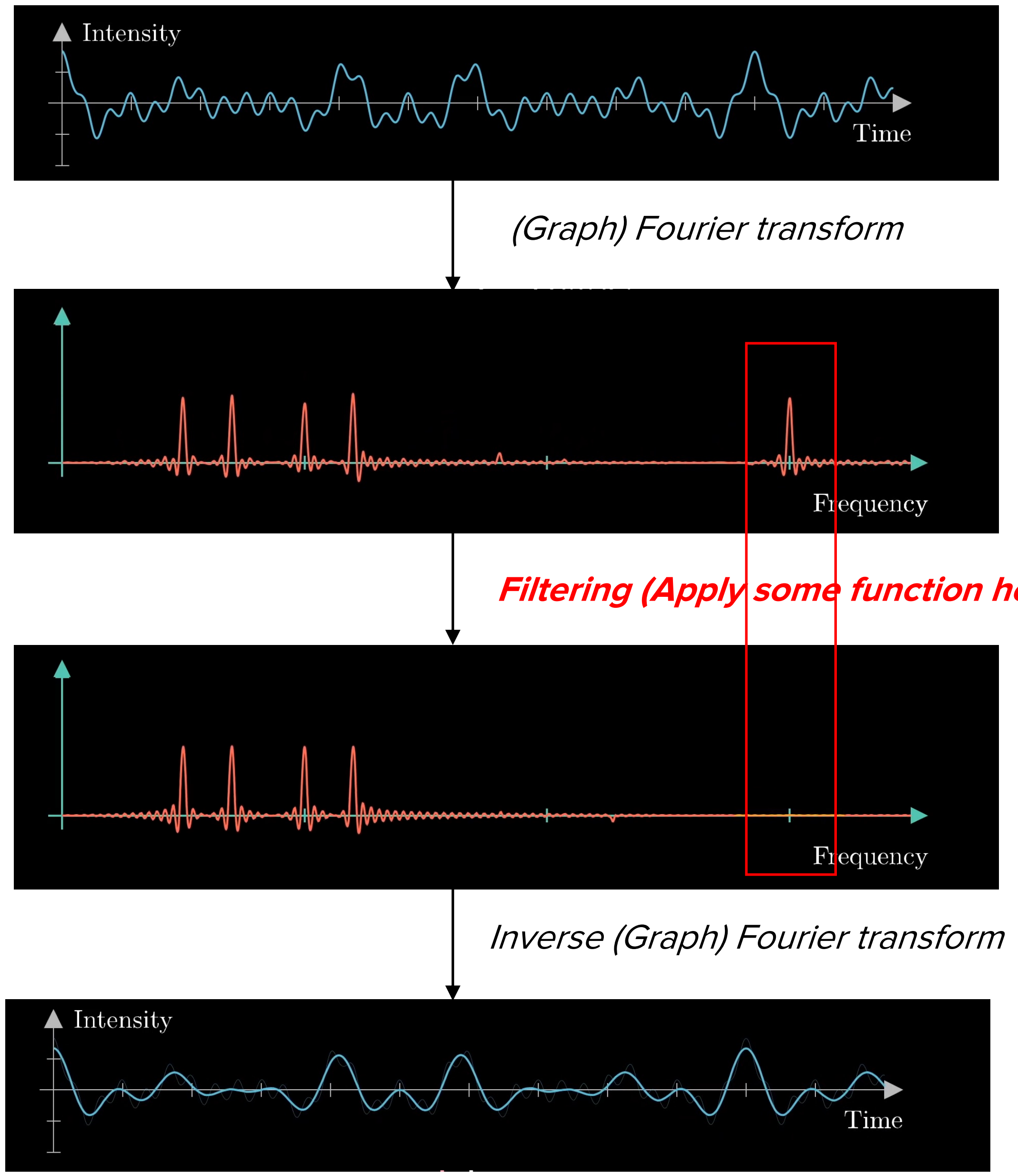
Note that the smoothness of the first graph is reflected in the **high signal of low eigenvalue/frequencies**

*** Low frequency = smooth, little variance**

In the third graph, **high signals of eigenvalue/frequencies start to appear**

*** High frequency = rapid oscillation, high variance**

…Coming back, completing the full graph Fourier transformation



(Graph) Fourier transform

Filtering (Apply some function here)

Inverse (Graph) Fourier transform

$f(i)$

$\hat{f}(\lambda_l)$

$$\hat{f}_{out}(\lambda_l) = \hat{f}(\lambda_l)\boxed{h(\lambda_l)}$$

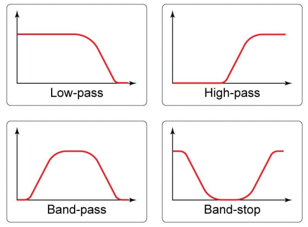
Send f in vertex space to frequency space

$\mathcal{L} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top$ $\quad$ $\mathcal{L} = \mathbf{D} - \mathbf{A}$

$$\hat{f}(\lambda_l) =< f, u_i >= \mathbf{U}^T f$$

Apply a filtering function h to eigenvalues

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_0 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \lambda_{N-1} \end{bmatrix} \Rightarrow h(\boldsymbol{\Lambda}) = \begin{bmatrix} h(\lambda_0) & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & h(\lambda_{N-1}) \end{bmatrix}$$



Send back to vertex space

$$f_{new} = \mathbf{U}h(\boldsymbol{\Lambda})\mathbf{U}^T f$$

Image: https://www.allaboutcircuits.com/technical-articles/low-pass-filter-tutorial-basics-passive-RC-filter/

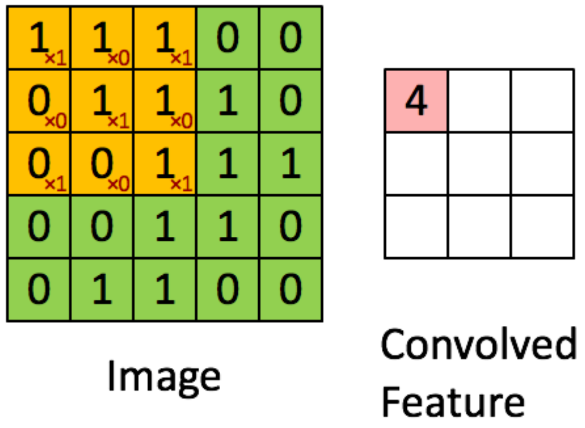
Understanding the **formulation of ChebNet & GCN**

Application of convolution (filters) to graphs

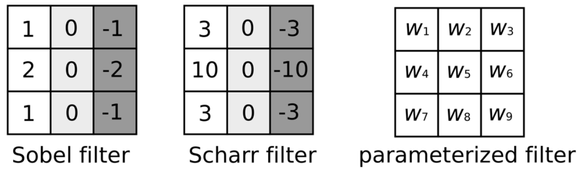
Defferrard et al., Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

1. **Spectral formulation**: Extend the formulation of graph signal processing
2. **Strictly localized filters**: Design a local filter localized in K hop from the central vertex

CNNs were successful because of local parameterized filters



✓ **Parameterized (Learnable) filter**



✓ **Convolution theorem**: Multiplication in the spectral space = Convolution in the original space

✓ **Localized** = the kernel is smaller than the original space
(for example, in the left figure, the kernel is 3 by 3, which is smaller than the image 5 by 5)

3. **Low computational complexity**: Expensive eigenvalue decomposition (spectral decomposition) is not needed

Image source: Left - https://techblog-history-younghunjo1.tistory.com/125 Right - https://anhreynolds.com/blogs/cnn.html

Localizing graph filters

Defferrard et al., Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

Graph in spectral domain

$$\hat{f}(\lambda_l)$$

Filtered by *h*

$$\hat{f}_{out}(\lambda_l) = \hat{f}(\lambda_l)\boxed{h(\lambda_l)}$$

(Convolution theorem:
Convolution in the original domain
= Product in the Fourier domain)

Apply a filtering function *h* to eigenvalues

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_0 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \lambda_{N-1} \end{bmatrix} \Rightarrow h(\mathbf{\Lambda}) = \begin{bmatrix} h(\lambda_0) & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & h(\lambda_{N-1}) \end{bmatrix} \Rightarrow h_{\boxed{\theta}}(\mathbf{\Lambda}) = \begin{bmatrix} \theta_0 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \theta_{N-1} \end{bmatrix}$$

Now we parameterize by *θ*
(learnable!)

$$\theta = [\theta_0, \cdots, \theta_{N-1}] \in \mathbb{R}^N$$

Notice that the parameters have the same size as the input *N*

Localizing graph filters

Defferrard et al., Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

$$h_\theta(\mathbf{\Lambda}) = \begin{bmatrix} \theta_0 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \theta_{N-1} \end{bmatrix}$$

$$\downarrow$$

$$\theta = [\theta_0, \cdots, \theta_{N-1}] \in \mathbb{R}^N$$

Global filter

Alternative formulation $\longrightarrow$

Polynomial filter

$$h_\theta(\mathbf{\Lambda}) = \underbrace{\theta_0\mathbf{I} + \theta_1\mathbf{\Lambda} + \theta_2\mathbf{\Lambda}^2 + \cdots \theta_{K-1}\mathbf{\Lambda}^{K-1}}_{\text{Polynomial filter}}$$

$$= \sum_{k=0}^{K-1} \theta_k \boxed{\mathbf{\Lambda}^k}$$

$$\downarrow$$

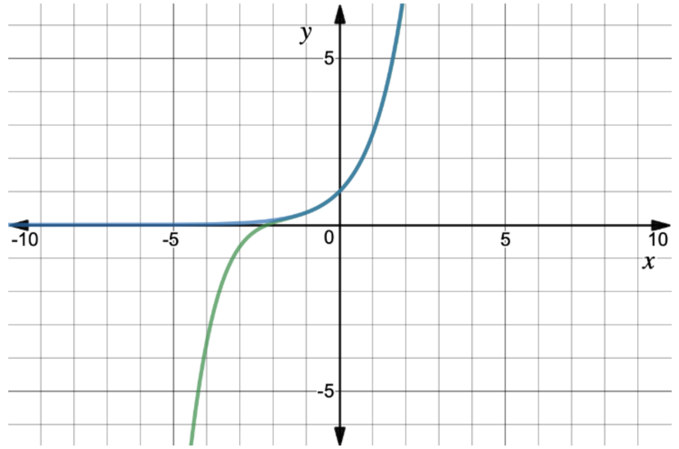
$$\theta = [\theta_0, \cdots, \theta_{K-1}] \in \mathbb{R}^K$$

Local filter ($K < N$)

$$\updownarrow$$

To calculate $\mathbf{\Lambda}^k$, we calculate up to $\mathcal{L}^k$.

We have the curve $f(x) = e^x$ in blue, and a Polynomial Approximation with equation $g(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5$ in green.



$$h_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{K-1} \theta_k \mathbf{\Lambda}^k$$

Local filter ($K < N$)

$$\mathbf{U}h_\theta(\mathbf{\Lambda})\mathbf{U}^\top x = \mathbf{U}\left(\sum_{k=0}^{K-1} \theta_k \mathbf{\Lambda}^k\right)\mathbf{U}^\top x$$

Now think of our original graph convolution, which becomes…

$$= \left(\sum_{k=0}^{K-1} \theta_k \mathcal{L}^k\right) x$$

…this!

$$\mathbf{U}\mathbf{U}^\top = \mathbf{I}$$

Polynomial approximation (right): https://www.expii.com/t/what-is-a-polynomial-approximation-317

Efficient implementation of local graph filters

Defferrard et al., Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

$$h_\theta(\mathbf{\Lambda}) = \overbrace{\theta_0\mathbf{I} + \theta_1\mathbf{\Lambda} + \theta_2\mathbf{\Lambda}^2 + \cdots \theta_{K-1}\mathbf{\Lambda}^{K-1}}^{\text{Polynomial filter}}$$

$$= \sum_{k=0}^{K-1} \theta_k \boxed{\mathbf{\Lambda}^k}$$

$$\downarrow$$

$$\theta = [\theta_0, \cdots, \theta_{K-1}] \in \mathbb{R}^K$$

Local filter ($K < N$)

$$\updownarrow$$

To calculate $\mathbf{\Lambda}^k$, we calculate up to $\mathcal{L}^k$.

But calculating $\mathcal{L}^k$ still seems heavy…

Idea

1. We can represent polynomials with Chebyshev expansion
2. Chebyshev expansion can be *efficiently* calculated via recursive relation
3. Also, they are superior than polynomial basis since its orthogonal

$$h_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{K-1} \theta_k \boxed{\mathbf{\Lambda}^k}$$

$\Downarrow$ ***Replace to…***

$$h_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{K-1} \theta_k \boxed{T_k(\mathbf{\Lambda})}$$

Chebyshev polynomials

where…

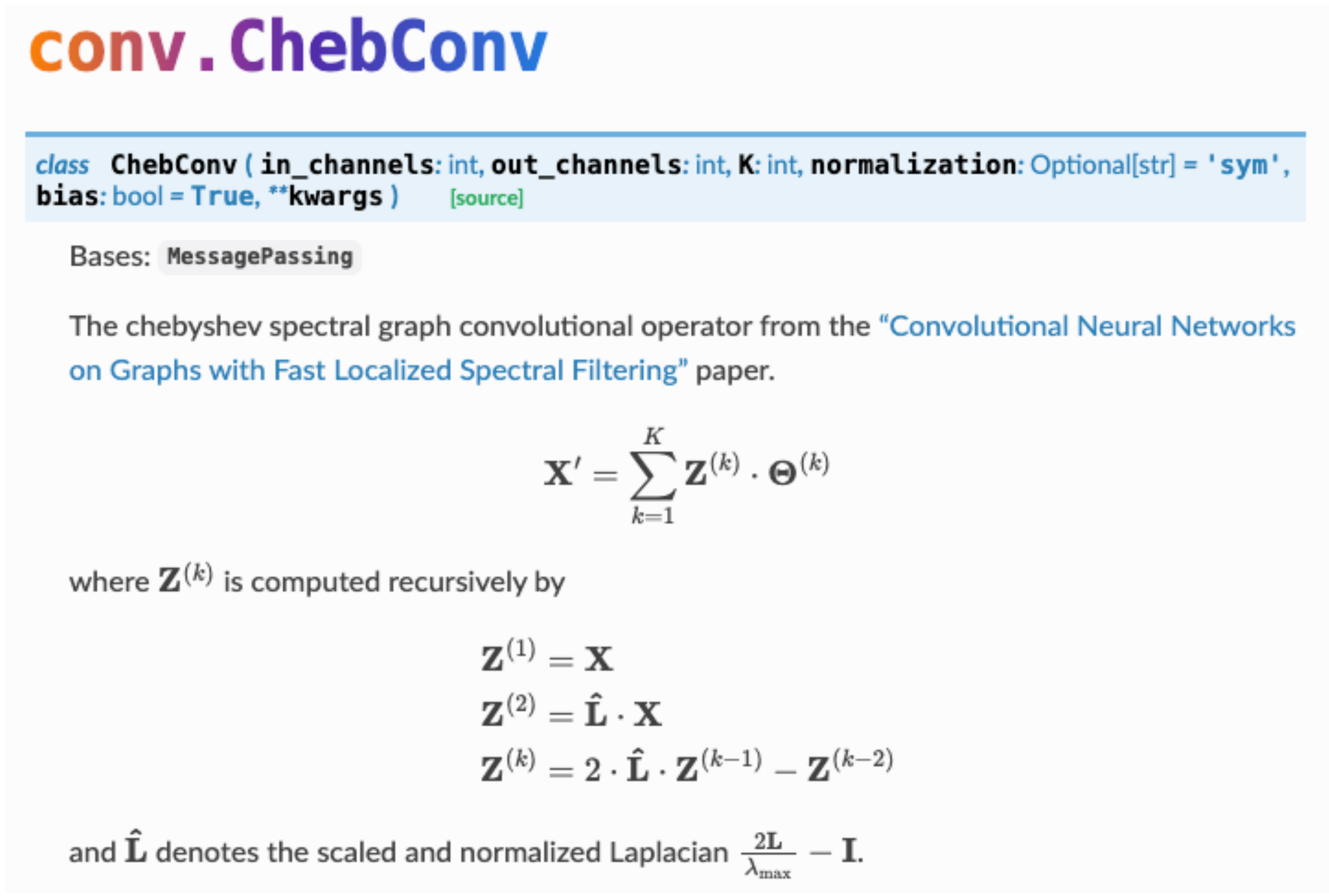
Efficient!

$$\begin{cases} T_0 = 1 \\ T_1 = x \\ T_k = 2xT_{k-1} - T_{k-2} \end{cases}$$

$$\mathcal{O}(N^2) \rightarrow \mathcal{O}(K|\mathcal{E}|)$$

Efficient implementation of local graph filters

Defferrard et al., Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering



conv.ChebConv

class **ChebConv** (**in_channels**: int, **out_channels**: int, **K**: int, **normalization**: Optional[str] = 'sym', **bias**: bool = True, **kwargs) [source]

Bases: `MessagePassing`

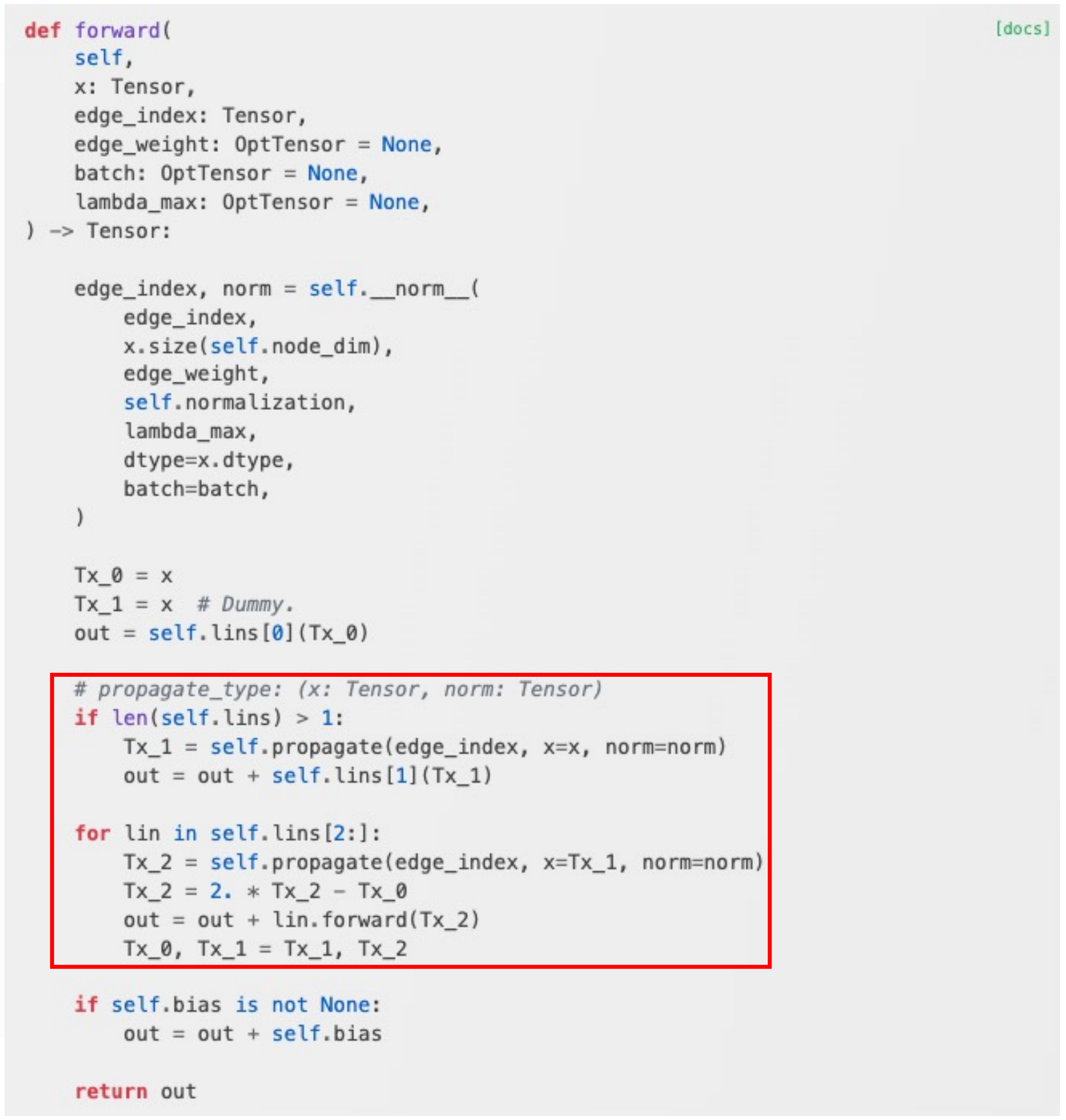
The chebyshev spectral graph convolutional operator from the "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering" paper.

$$\mathbf{X}' = \sum_{k=1}^{K} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

where $\mathbf{Z}^{(k)}$ is computed recursively by

$$\mathbf{Z}^{(1)} = \mathbf{X}$$
$$\mathbf{Z}^{(2)} = \mathbf{\hat{L}} \cdot \mathbf{X}$$
$$\mathbf{Z}^{(k)} = 2 \cdot \mathbf{\hat{L}} \cdot \mathbf{Z}^{(k-1)} - \mathbf{Z}^{(k-2)}$$

and $\mathbf{\hat{L}}$ denotes the scaled and normalized Laplacian $\frac{2\mathbf{L}}{\lambda_{\max}} - \mathbf{I}$.

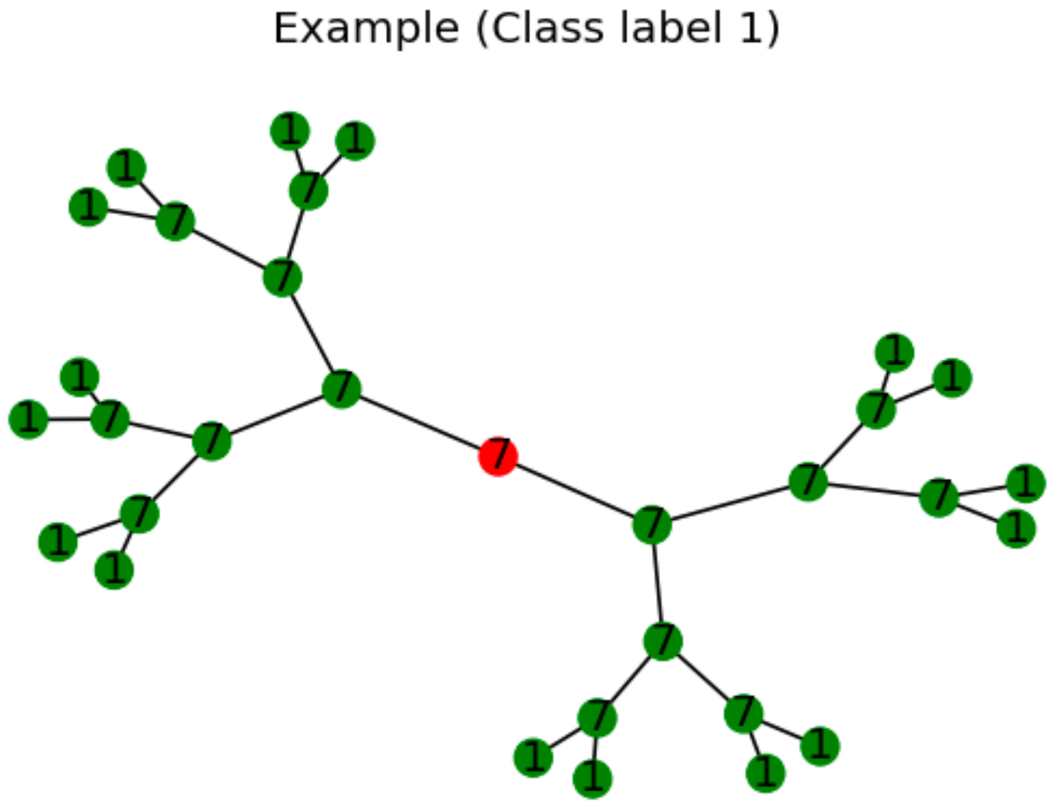


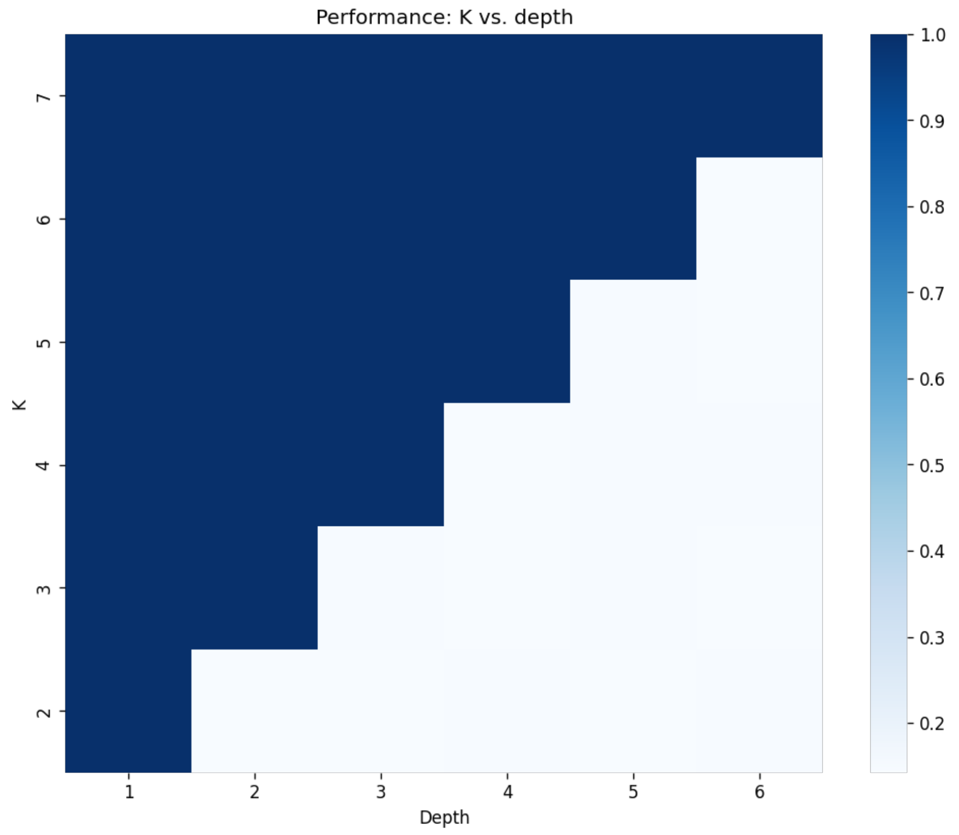
```
def forward(                                                        [docs]
    self,
    x: Tensor,
    edge_index: Tensor,
    edge_weight: OptTensor = None,
    batch: OptTensor = None,
    lambda_max: OptTensor = None,
) -> Tensor:

    edge_index, norm = self.__norm__(
        edge_index,
        x.size(self.node_dim),
        edge_weight,
        self.normalization,
        lambda_max,
        dtype=x.dtype,
        batch=batch,
    )

    Tx_0 = x
    Tx_1 = x  # Dummy.
    out = self.lins[0](Tx_0)

    # propagate_type: (x: Tensor, norm: Tensor)
    if len(self.lins) > 1:
        Tx_1 = self.propagate(edge_index, x=x, norm=norm)
        out = out + self.lins[1](Tx_1)

    for lin in self.lins[2:]:
        Tx_2 = self.propagate(edge_index, x=Tx_1, norm=norm)
        Tx_2 = 2. * Tx_2 - Tx_0
        out = out + lin.forward(Tx_2)
        Tx_0, Tx_1 = Tx_1, Tx_2

    if self.bias is not None:
        out = out + self.bias

    return out
```

A simple example to showcase the locality

Application of ChebNet to the NEIGHBORSMATCH (Alon & Yahav, 2021) problem

A modified NEIGHRBORSMATCH probem





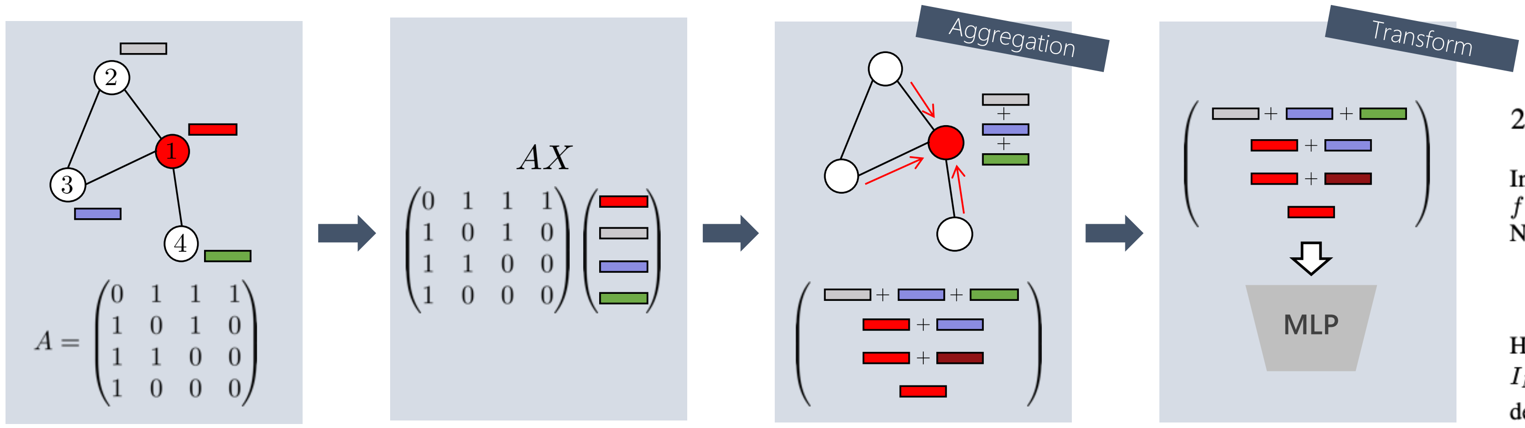
- Target: Root node (red)'s label
- The leaf node's node feature (1 in the example) is the root node's label
- The rest of the node's feature are completely irrelevant
- The model MUST be able to aggregate at least 4-hop local neighbor's information.

- Use a single ChebConv layer
- The model's locality (K) should at least match the minimum depth required by the NEIGHBORSMATCH problem.
- Since the problem is very easy to solve (as long as the information is properly gathered), the performance is 100% or near 1/(# of classes)%.

Alon & Yahav, On the bottleneck of graph neural networks and its practical implications, ICLR 2021
For more reading on this simple experiment, go to: https://jordan7186.github.io/blog/2022/ChebConv/

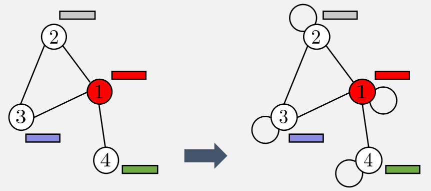
Revisiting GCNs, but in the original author’s way

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks


2
1
3
4
$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$
AX
$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$
Aggregation
Transform
MLP

Resolution to problem 1

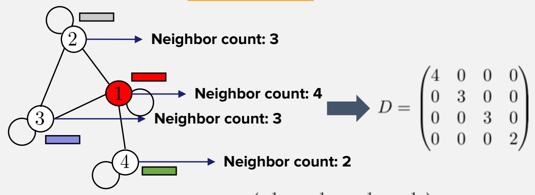
Add self-loops to each node



$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \hat{A} = A + I = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

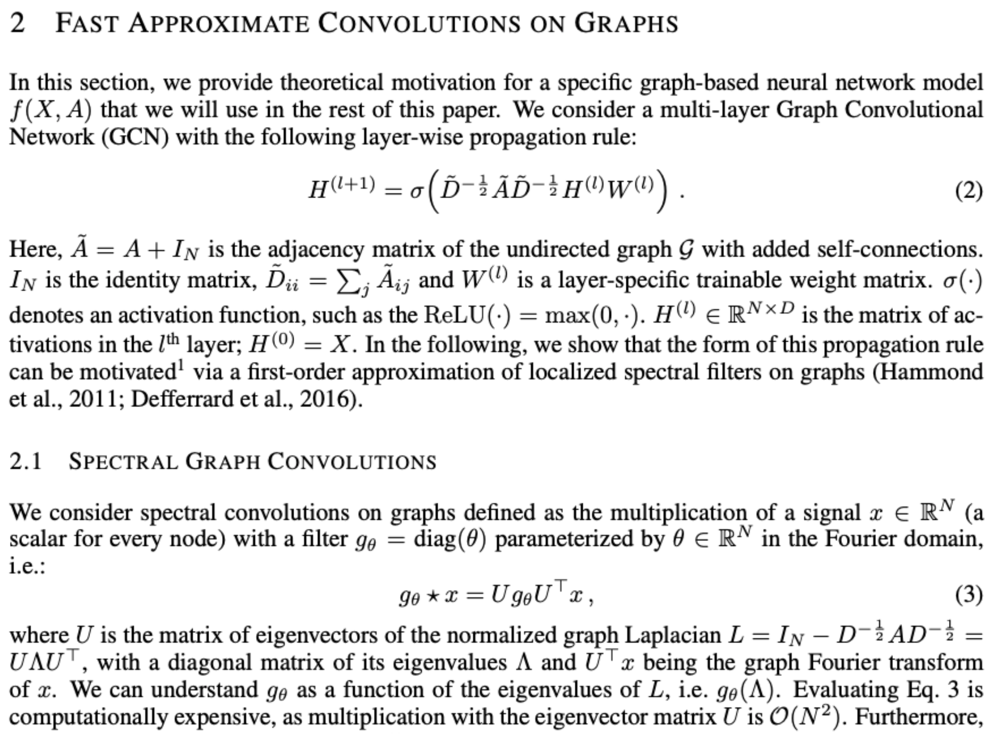
Resolution to problem 2

Normalization of $\hat{A}$:



$$D = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

$$\tilde{A} = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} = \begin{pmatrix} \frac{1}{4} & \frac{1}{\sqrt{12}} & \frac{1}{\sqrt{12}} & \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{12}} & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{\sqrt{12}} & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{\sqrt{8}} & 0 & 0 & \frac{1}{2} \end{pmatrix}$$



2 FAST APPROXIMATE CONVOLUTIONS ON GRAPHS

In this section, we provide theoretical motivation for a specific graph-based neural network model $f(X, A)$ that we will use in the rest of this paper. We consider a multi-layer Graph Convolutional Network (GCN) with the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) . \qquad (2)$$

Here, $\tilde{A} = A + I_N$ is the adjacency matrix of the undirected graph $\mathcal{G}$ with added self-connections. I_N is the identity matrix, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and $W^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes an activation function, such as the $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$. $H^{(l)} \in \mathbb{R}^{N \times D}$ is the matrix of activations in the l^{th} layer; $H^{(0)} = X$. In the following, we show that the form of this propagation rule can be motivated[1] via a first-order approximation of localized spectral filters on graphs (Hammond et al., 2011; Defferrard et al., 2016).

2.1 SPECTRAL GRAPH CONVOLUTIONS

We consider spectral convolutions on graphs defined as the multiplication of a signal $x \in \mathbb{R}^N$ (a scalar for every node) with a filter $g_\theta = \mathrm{diag}(\theta)$ parameterized by $\theta \in \mathbb{R}^N$ in the Fourier domain, i.e.:

$$g_\theta \star x = U g_\theta U^\top x \, , \qquad (3)$$

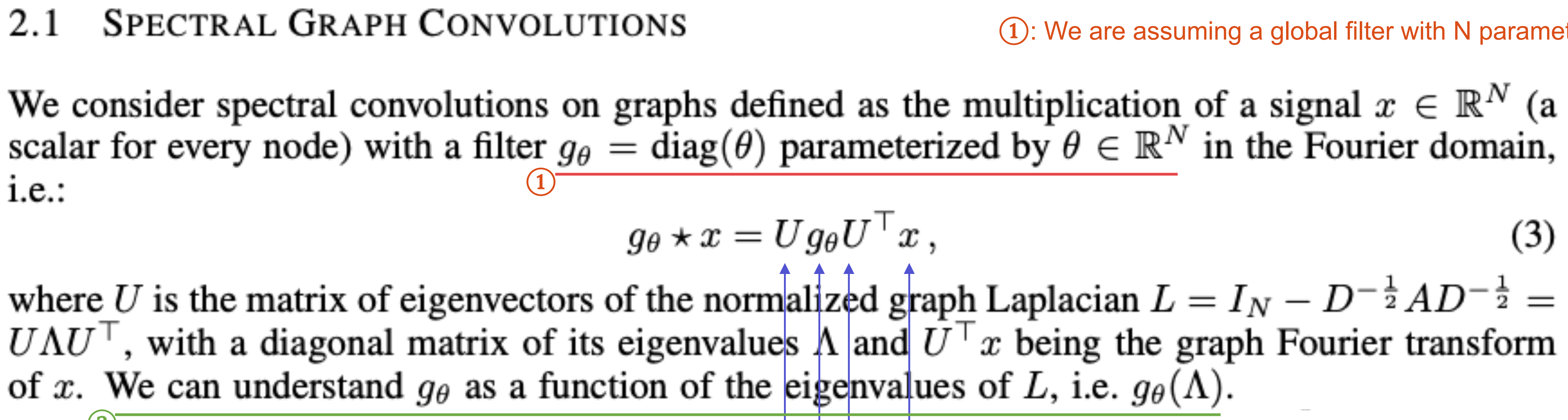
where U is the matrix of eigenvectors of the normalized graph Laplacian $L = I_N - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} = U\Lambda U^\top$, with a diagonal matrix of its eigenvalues Λ and $U^\top x$ being the graph Fourier transform of x. We can understand g_θ as a function of the eigenvalues of L, i.e. $g_\theta(\Lambda)$. Evaluating Eq. 3 is computationally expensive, as multiplication with the eigenvector matrix U is $\mathcal{O}(N^2)$. Furthermore,

We are now ready to follow the original author’s motivation for GCN.
But keep in mind that all of the previous explanations are still intact. **It is still the same model.**

Revisiting GCNs, but in the original author’s way

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks

We now understand the first part of Section 2.1.



2.1 Spectral Graph Convolutions

We consider spectral convolutions on graphs defined as the multiplication of a signal $x \in \mathbb{R}^N$ (a scalar for every node) with a filter $\underbrace{g_\theta = \mathrm{diag}(\theta)}_{①}$ parameterized by $\theta \in \mathbb{R}^N$ in the Fourier domain, i.e.:

$$g_\theta \star x = U g_\theta U^\top x \,, \tag{3}$$

where U is the matrix of eigenvectors of the normalized graph Laplacian $L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} = U \Lambda U^\top$, with a diagonal matrix of its eigenvalues Λ and $U^\top x$ being the graph Fourier transform of x. ② We can understand g_θ as a function of the eigenvalues of L, i.e. $g_\theta(\Lambda)$.

①: We are assuming a global filter with N parameters.

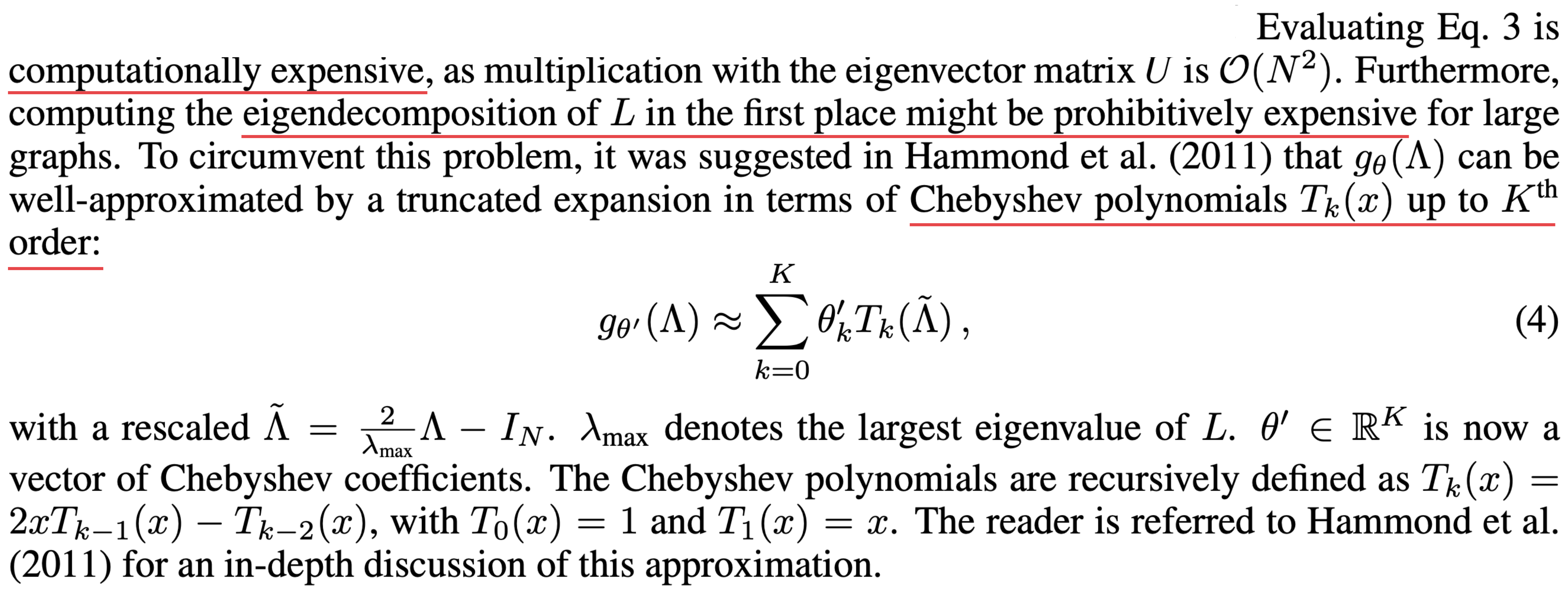
②: Use the original eigenvalues as a basis of some approximation, then we can learn the coefficients.

1. Initial graph signal on every node
2. Send the signal to the frequency domain (Fourier)
3. Apply the (learnable) filter function (Equalizer!)
4. Send the edited signal back to the original domain (inverse Fourier)

Revisiting GCNs, but in the original author's way

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks

The next part is now also familiar to us…



Evaluating Eq. 3 is computationally expensive, as multiplication with the eigenvector matrix U is $\mathcal{O}(N^2)$. Furthermore, computing the eigendecomposition of L in the first place might be prohibitively expensive for large graphs. To circumvent this problem, it was suggested in Hammond et al. (2011) that $g_\theta(\Lambda)$ can be well-approximated by a truncated expansion in terms of Chebyshev polynomials $T_k(x)$ up to K^{th} order:

$$g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{\Lambda}) \,, \tag{4}$$

with a rescaled $\tilde{\Lambda} = \frac{2}{\lambda_{\max}}\Lambda - I_N$. $\lambda_{\max}$ denotes the largest eigenvalue of L. $\theta' \in \mathbb{R}^K$ is now a vector of Chebyshev coefficients. The Chebyshev polynomials are recursively defined as $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$, with $T_0(x) = 1$ and $T_1(x) = x$. The reader is referred to Hammond et al. (2011) for an in-depth discussion of this approximation.

As we have seen, this idea has been extensively employed by ChebConv

Revisiting GCNs, but in the original author's way

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks

The next part is now also familiar to us…

Going back to our definition of a convolution of a signal x with a filter $g_{\theta'}$, we now have:

$$g_{\theta'} \star x \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) x \,, \tag{5}$$

with $\tilde{L} = \frac{2}{\lambda_{\max}} L - I_N$; as can easily be verified by noticing that $(U\Lambda U^\top)^k = U\Lambda^k U^\top$. Note that this expression is now K-localized since it is a K^{th}-order polynomial in the Laplacian, i.e. it depends only on nodes that are at maximum K steps away from the central node (K^{th}-order neighborhood). The complexity of evaluating Eq. 5 is $\mathcal{O}(|\mathcal{E}|)$, i.e. linear in the number of edges. Defferrard et al. (2016) use this K-localized convolution to define a convolutional neural network on graphs.

In other words, each node requires up to K-hop local neighborhood information to capture up to K-th complex patterns.

Revisiting GCNs, but in the original author's way

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks

In Section 2.2, the authors start to introduce the 'deep learning' style motivations

2.2 LAYER-WISE LINEAR MODEL

Let's consider the extremely simplified case as a single layer, and let the designer choose how much layer to stack.

A neural network model based on graph convolutions can therefore be built by stacking multiple convolutional layers of the form of Eq. 5, each layer followed by a point-wise non-linearity. Now, imagine we limited the layer-wise convolution operation to $K = 1$ (see Eq. 5), i.e. a function that is linear w.r.t. L and therefore a linear function on the graph Laplacian spectrum.

$$h_\theta(\mathbf{\Lambda}) = \overbrace{\boxed{\theta_0 \mathbf{I} + \theta_1 \mathbf{\Lambda}} + \cancel{\theta_2 \mathbf{\Lambda}^2 + \cdots \theta_{K-1}\mathbf{\Lambda}^{K-1}}}^{\text{Polynomial filter}}$$

⋮

In this linear formulation of a GCN we further approximate $\lambda_{\max} \approx 2$, as we can expect that neural network parameters will adapt to this change in scale during training. Under these approximations Eq. 5 simplifies to:

$$g_{\theta'} \star x \approx \boxed{\theta'_0 x + \theta'_1 \left(L - I_N\right) x} = \theta'_0 x - \theta'_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x \,, \tag{6}$$

with two free parameters θ'_0 and θ'_1. The filter parameters can be shared over the whole graph. Successive application of filters of this form then effectively convolve the k^{th}-order neighborhood of a node, where k is the number of successive filtering operations or convolutional layers in the neural network model.

Revisiting GCNs, but in the original author’s way

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks

…which leads us to the final (and familiar) GCN layer.

Do we even need to differentiate θ'_0 and θ'_1? Let’s combine them to a single parameter.

In practice, it can be beneficial to constrain the number of parameters further to address overfitting and to minimize the number of operations (such as matrix multiplications) per layer. This leaves us with the following expression:

$$g_\theta \star x \approx \theta \left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \right) x \,, \tag{7}$$

with a single parameter $\theta = \theta'_0 = -\theta'_1$. Note that $I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ now has eigenvalues in the range $[0, 2]$. Repeated application of this operator can therefore lead to numerical instabilities and exploding/vanishing gradients when used in a deep neural network model. To alleviate this problem, we introduce the following *renormalization trick*: $I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \rightarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, with $\tilde{A} = A + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.

We can generalize this definition to a signal $X \in \mathbb{R}^{N \times C}$ with C input channels (i.e. a C-dimensional feature vector for every node) and F filters or feature maps as follows:

$$\boxed{Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta}\,, \tag{8}$$

GCNs: Let’s be more efficient then ChebConv

Kipf & Welling, Semi-supervised Classification with Graph Convolutional Networks

1. Always set $K = 2$ (Up to linear term)

$$h_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{1} \theta_k T_k(\mathbf{\Lambda}) = \theta_0 + \theta_1 \mathbf{\Lambda}$$

2. Instead, **stack multiple layers**

3. Use normalized Laplacians

$$\mathbf{L} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$$

Now the convolution (per layer) is modeled as:

$$\theta(\underbrace{\underbrace{I}_{\text{Eigenvalue: } [0,1]} + \underbrace{\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}}_{\text{Eigenvalue: } [0,1]}}_{\text{Eigenvalue: } [0,2]})$$

Eigenvalue: [0,2]
- Stacking multiple layers may cause explosion of eigenvalue (Numerical instability)
- New normalization is needed to keep in [0,1]

$$\Big\downarrow \ \textit{renormalization trick}$$

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$$

$$I + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} \rightarrow \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$$

Final convolution layer

$$\sigma(\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}\Theta)$$

Takeaways

1. Graph Fourier Transform: Start with the generalized concept of Fourier transform, everything else is the same.

2. ChebNet / ChebConv: A learnable, localized filter for graphs

3. GCN: Push the simplification of graph filters to the extreme, compensate by stacking multiple layers.

*If you are interested in the actual efficiency between ChebNet vs. GCN, check out
https://jordan7186.github.io/blog/2022/Efficiency_Comparison/

*Highly recommended reading (for follow-up work and great summary):
Wang & Zhang, How powerful are spectral graph neural networks, ICML 2022

Thank you!

Please feel free to ask any questions :)

jordan7186.github.io