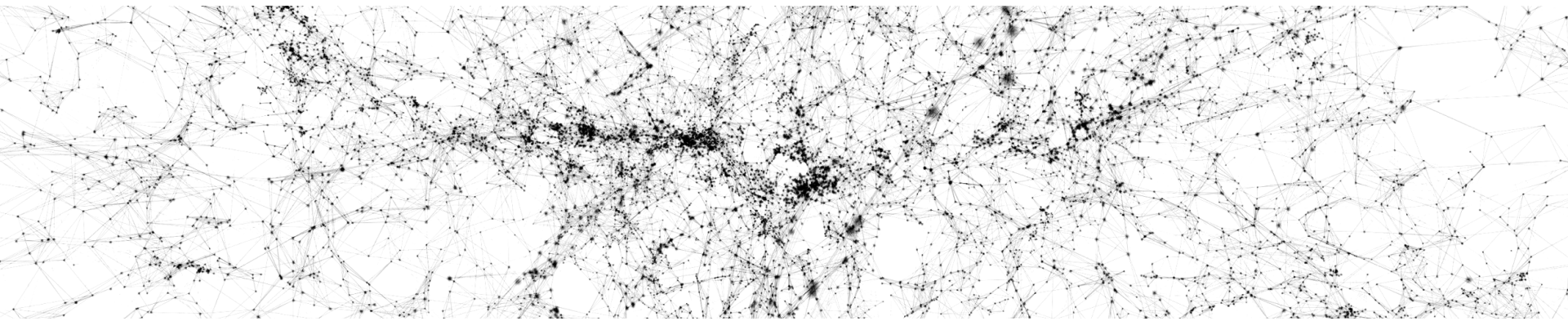
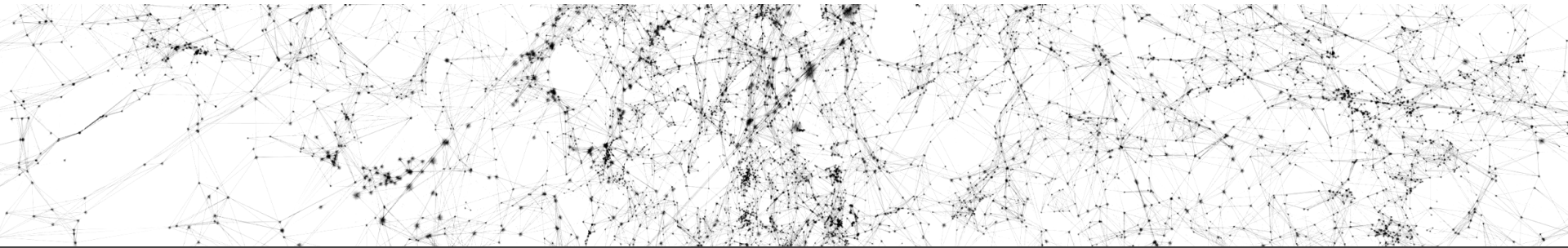


Graph Neural Networks

Analysis of complex interconnected data

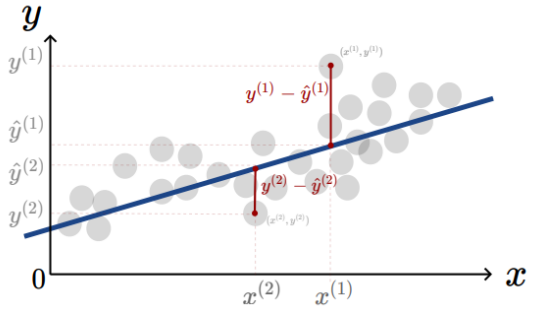


Neural Networks - Short Intro - Linear Model

- Linear regression: $f(x) = w^\top x = \sum_d w_d x_d$

$$x = [1, x_1, \ldots, x_D]^\top$$

$$w = [w_0, w_1, \ldots, w_D]^\top$$



Model: linear combination of features and weights
Learning: find the weights that minimize a cost function
Cost: sum of losses per individual point

$$J(w) = \frac{1}{2} \sum_{n=1}^{N} \left(y^{(n)} - w^\top x^{(n)} \right)^2$$

$$w^* = \arg\min_w J(w)$$

$$X = \begin{bmatrix} x^{(1)\top} \\ x^{(2)\top} \\ \vdots \\ x^{(N)\top} \end{bmatrix} = \begin{bmatrix} x_1^{(1)}, & x_2^{(1)}, & \cdots, & x_D^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)}, & x_2^{(N)}, & \cdots, & x_D^{(N)} \end{bmatrix} \in \mathbb{R}^{N \times D}$$

one instance

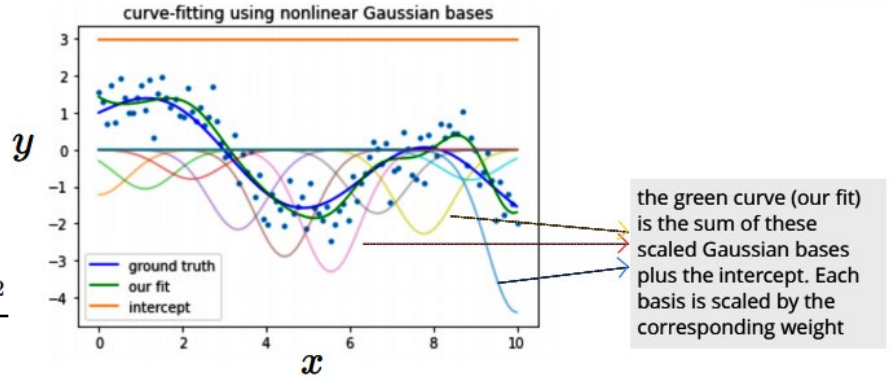
one feature

Neural Networks - Short Intro - Beyond Linear

- Linear regression: assume input of one dimension $f(x; w) = wx$
- More expressive: use nonlinear bases: $f(x; w) = \sum_m w_m \phi_m(x)$
  - Transform the input with nonlinearities then apply linear model

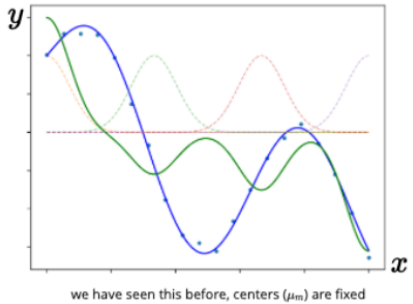
Example: blue dots are the datapoints, generated from blue ground truth curve with some random noise. We can get a very close nonlinear fit, green line, with this linear regression model when using 10 nonlinear Gaussian bases, equally spaced

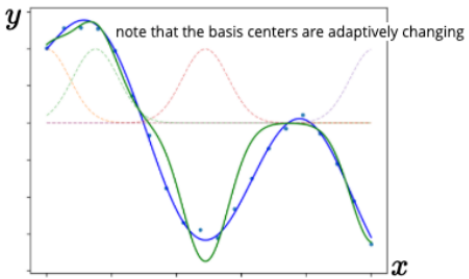
$$\phi_k(x) = e^{-\frac{(x-\mu_k)^2}{s^2}}$$



Neural Networks - Short Intro - Adaptive Bases

- Linear regression: assume input of one dimension $f(x; w) = wx$
- More expressive: use nonlinear bases: $f(x; w) = \sum_m w_m \phi_m(x)$
  - Transform the input with fixed nonlinearities then apply linear model
- Neural networks use **adaptive** nonlinear bases $f(x; w, \mu) = \sum_m w_m \phi_m(x; \mu_m)$
  - Transform the input with learnable nonlinearities then apply linear model

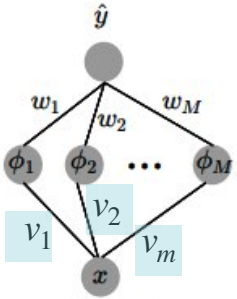


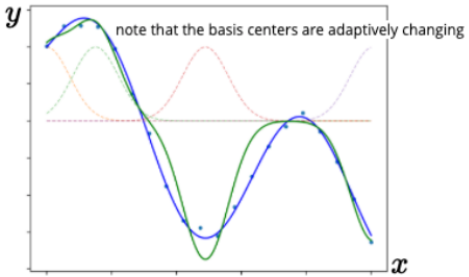


Neural Networks - Short Intro - Adaptive Bases

- Linear regression: assume input of one dimension $f(x; w) = wx$
- More expressive: use nonlinear bases: $f(x; w) = \sum_m w_m \phi_m(x)$
  - Transform the input with fixed nonlinearities then apply linear model
- Neural networks use **adaptive** nonlinear bases $f(x; w, \mu) = \sum_m w_m \phi_m(x; \mu_m)$
  - Transform the input with learnable nonlinearities then apply linear model

We can represent the resulting model with the data transformation graphically as:





Neural Networks - Short Intro - MLP

- Neural networks use **adaptive** nonlinear bases
  - Learning the (weights of) nonlinear bases

$$\hat{y}_c = g\left(\sum_m W_{c,m} h\left(\sum_d V_{m,d} x_d\right)\right)$$

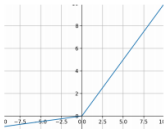
g: nonlinearity, activation function: we have different choices

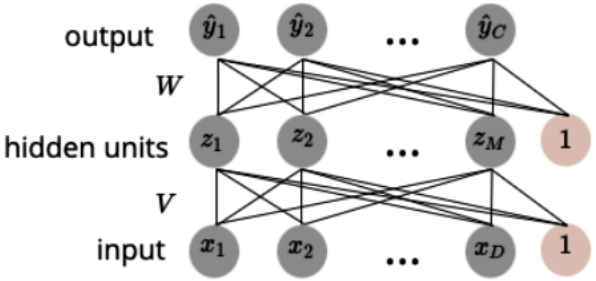
more compressed form

$$\hat{y} = f(x; W, V) = g\big(Wh(Vx)\big)$$

  - The most common non-linearity

leaky ReLU $h(x) = \max(0, x) + \gamma \min(0, x)$





$x \in \mathbb{R}^{D\times 1}$
$V \in \mathbb{R}^{M\times D}$
$Z = h(Vx) \in \mathbb{R}^{M\times 1}$
$W \in \mathbb{R}^{C\times M}$
$y \in \mathbb{R}^{C\times 1}$

$$\min_{W,V} \sum_n L(y^{(n)}, f(x^{(n)}; W, V))$$

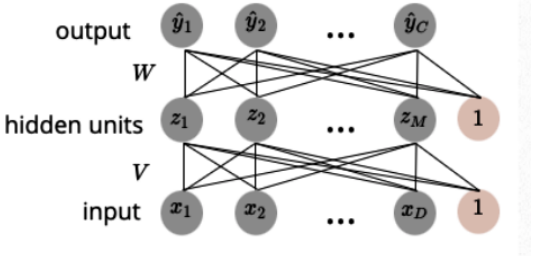
We learn V and W with backpropagation to minimize a loss function

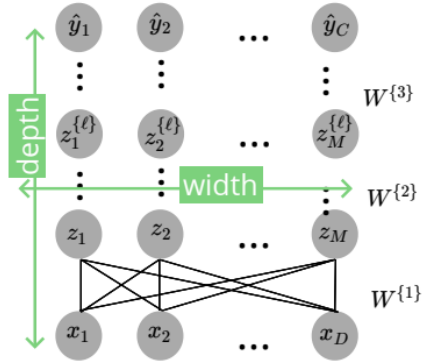
```
while Condition:
    dw, dv = gradient(x, y, w, v)
    w = w - lr*dw
    v = v - lr*dv
```

But what is a neural network?

Neural Networks - Short Intro - Deep Networks

- Neural networks use **adaptive** nonlinear bases
  - Example: 2 layers MLP
- **Deep** networks: neural networks with many layers
  - Stack/compose layers of adaptive nonlinear bases
  - # layers is called the depth of the network
  - each layer can be fully connected (dense) or sparse
  - empirically, increasing the depth is often more effective than increasing the width (#parameters per layer) Not the case for GNNs
  - parameters may be shared across units (e.g., in conv-nets)





$$z^{\{l\}} = h\left(W^{\{l\}} z^{\{l-1\}}\right)$$

output of one layer is input to the next

Deep Networks for Graphs?

Can we feed an adjacency matrix to this?

Not the best choice: Order $\mathcal{O}(n)$ parameters, also Sensitive to order of nodes

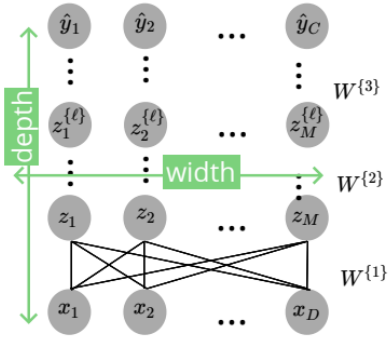
What if it is for Graph Classification, flatten the matrix into a vector?

$$z = MLP(A[1] \oplus A[2] \oplus \ldots A[n])$$

concatenation

What is the size of D? Input dimension? n^2

Still same issues, depends on the arbitrary ordering of nodes that we used in the adjacency matrix



Permutation Invariance & Equivariance

function f that takes an adjacency matrix A as input should be:

- Permutation Invariance

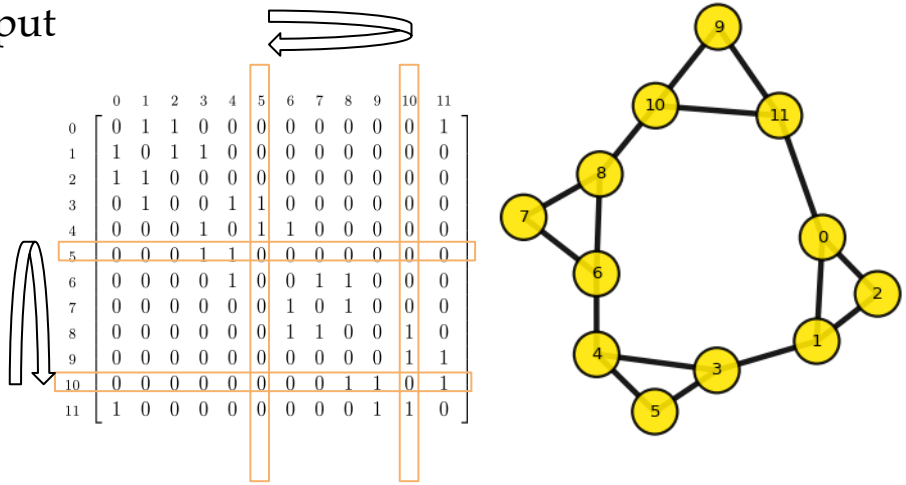
$$f(PAP^\top) = f(A)$$

or

- Permutation Equivariance

$$f(PAP^\top) = Pf(A)$$

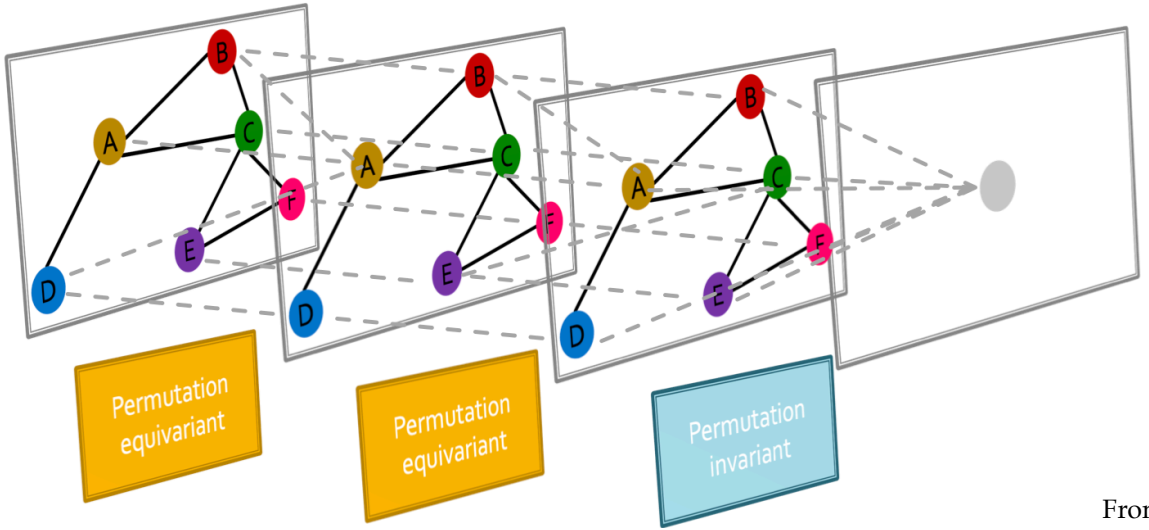
where P is a permutation matrix that reorders nodes



Since changing order of nodes in the adjacency does not change the graph

Graph Neural Networks

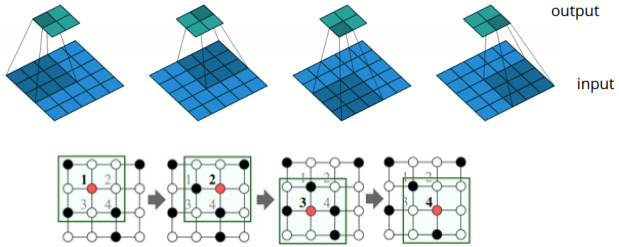
- GNNs consist of multiple permutation equivariant / invariant functions

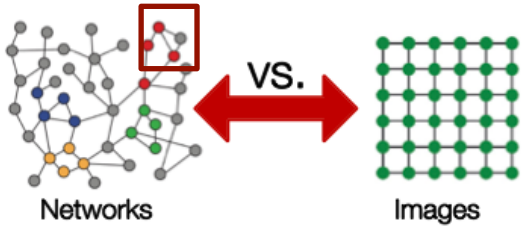


From https://web.stanford.edu/class/cs224w/slides/06-GNN1.pdf

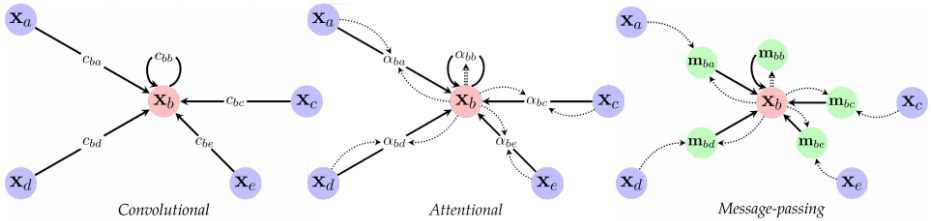
Graph Neural Networks

Use the local neighbourhood similar to convolution on images: use local neighbourhood similar to CNN for images





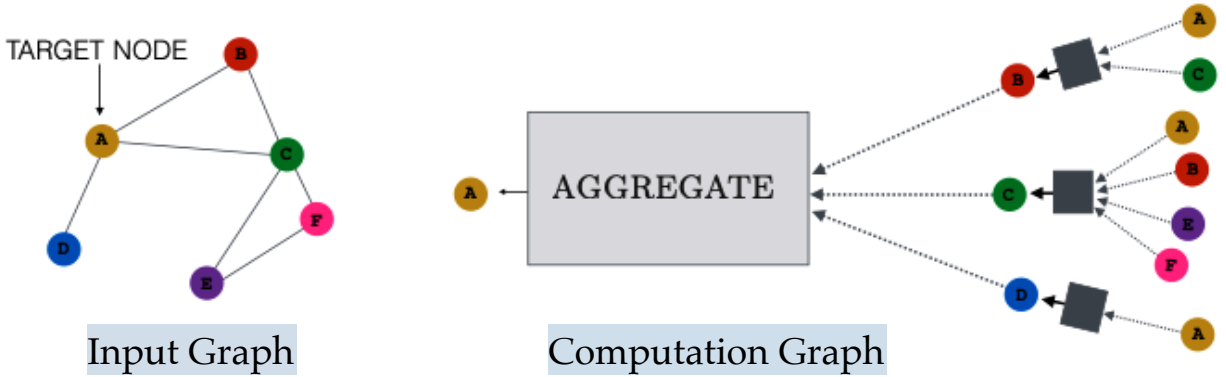
GNN model uses a message passing framework



See https://petar-v.com/talks/GNN-Wednesday.pdf

Graph Neural Networks: Neural Message Passing

vector messages are exchanged between nodes and updated using neural networks



Input Graph

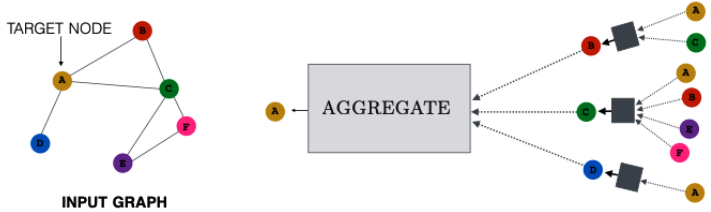
Computation Graph

a tree structure by unfolding the neighbourhood around the target node

Graph Neural Networks: Neural Message Passing

vector messages are exchanged between nodes and updated using neural networks

a *hidden embedding* $\mathbf{h}_u^{(k)}$



$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right)$$
$$= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right),$$

UPDATE and AGGREGATE are arbitrary differentiable functions

AGGREGATE function takes a set as input, GNNs defined in this way are permutation equivariant by design.

initial embeddings at k = 0 are set to the input features for all the nodes, if no feature can be identity: a one-hot indicator feature, which uniquely identifies that node

Basic GNN

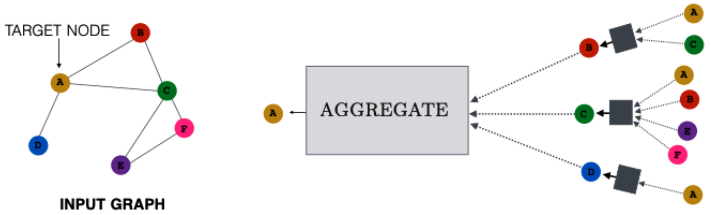
vector messages are exchanged between nodes and updated using neural networks

a *hidden embedding* $\mathbf{h}_u^{(k)}$

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right)$$
$$= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right),$$

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v,$$
$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right)$$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$



$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)$$

trainable parameter matrices

In matrix form:

$$\mathbf{H}^{(t)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}_{\text{neigh}}^{(k)} + \mathbf{H}^{(k-1)}\mathbf{W}_{\text{self}}^{(k)}\right)$$

Intuition

If we have:

$$A_{ij} = \begin{cases} 1, & i \text{ links to } j \\ 0, & \text{otherwise} \end{cases} \qquad X_{ik} = \begin{cases} 1, & i \text{ has } k \\ 0, & \text{otherwise} \end{cases}$$

Then simple matrix multiplication of A and X, AX, gives us the number of neighbors of a particular attribute/type for each node, i.e.

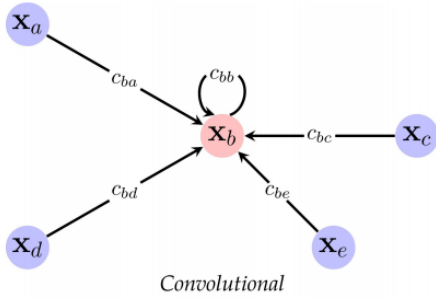
- k^{th} column of AX shows the number of type k neighbors for all nodes,
  - e.g., number of 'male' friends each person has.
- i^{th} row of AX shows the number of neighbors node i for all types,
  - e.g., number of friends 'smith' has of each type, say male and female

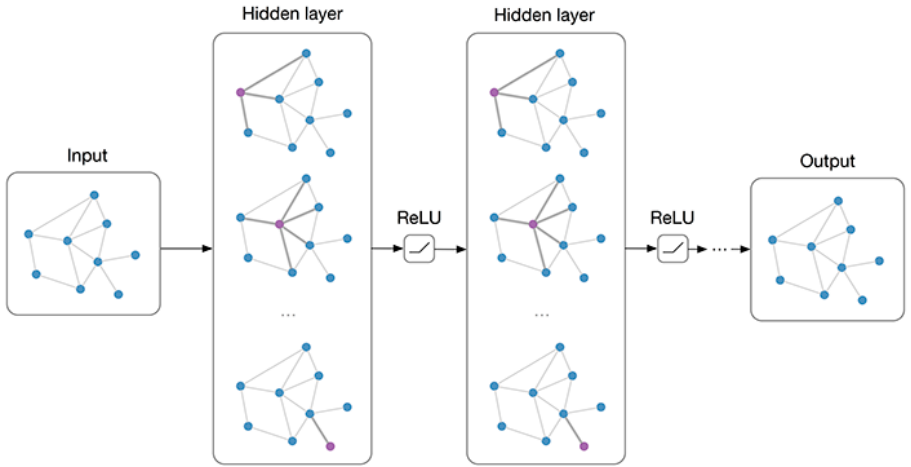
For graph neural networks, after the first iteration (k = 1), every node embedding contains a information from its 1-hop neighbourhood

$$\mathbf{H}^{(t)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}_{\text{neigh}} + \mathbf{H}^{(k-1)}\mathbf{W}^{(k)}_{\text{self}}\right)$$

Convolutional GNN

GCN (Kipf & Welling, ICLR'17)



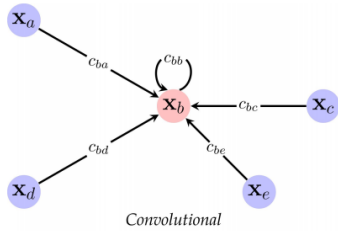


Multi-layer Graph Convolutional Network (GCN) with first-order filters.

$$H^{l+1} = \phi(AH^lW^l)$$

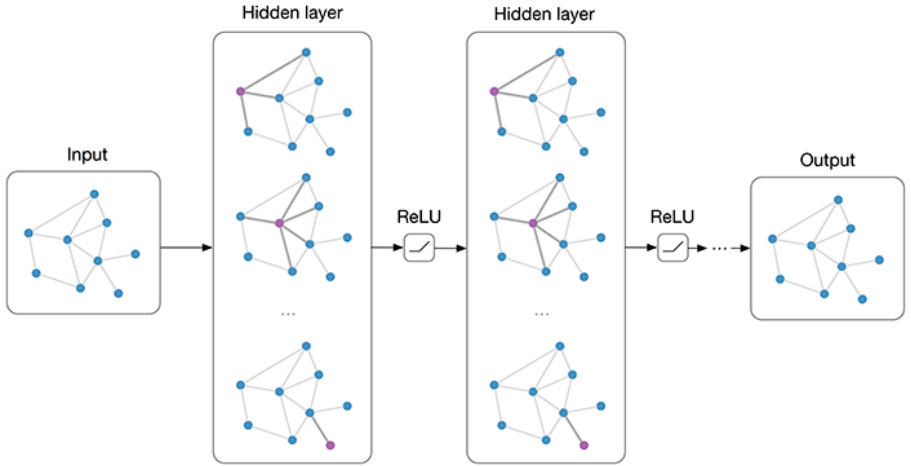
From https://petar-v.com/talks/GNN-Wednesday.pdf
https://www.youtube.com/watch?v=uF53xsT7mjc

Convolutional GNN



GCN (Kipf & Welling, ICLR'17): has self-loop and symmetric normalization

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$



Multi-layer Graph Convolutional Network (GCN) with first-order filters.

$$H^{l+1} = \phi(AH^lW^l)$$

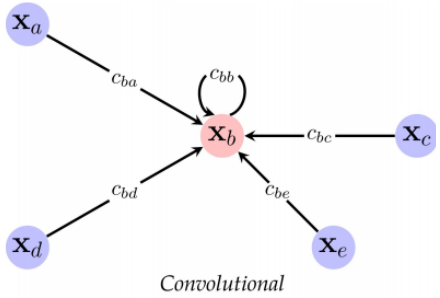
$$H^{l+1} = \phi(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^lW^l)$$

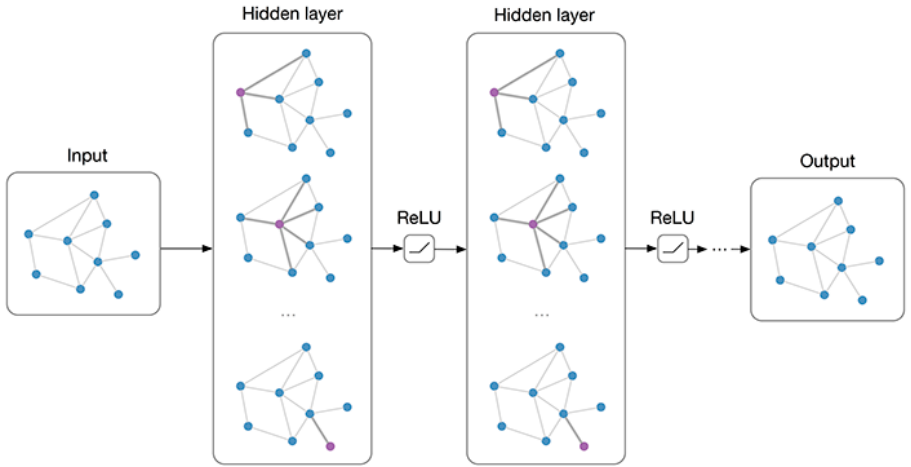
$$\hat{A} = A + I$$

From https://petar-v.com/talks/GNN-Wednesday.pdf
https://www.youtube.com/watch?v=uF53xsT7mjc

Convolutional GNN

GCN (Kipf & Welling, ICLR'17)





Multi-layer Graph Convolutional Network (GCN) with first-order filters.

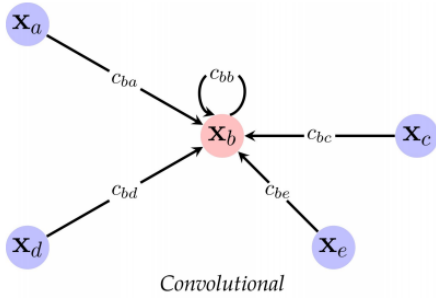
$$H^{l+1} = \phi(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^l W^l)$$

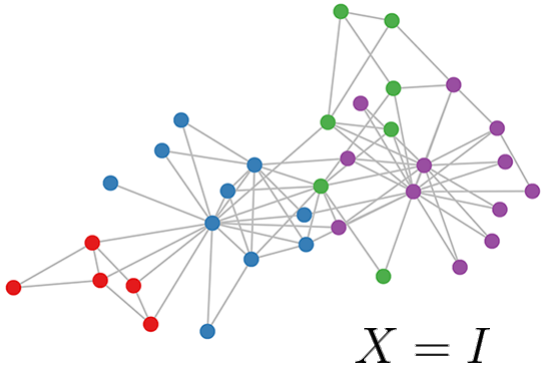
$$h_i^{l+1} = \phi(\sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} h_j^l W^l)$$

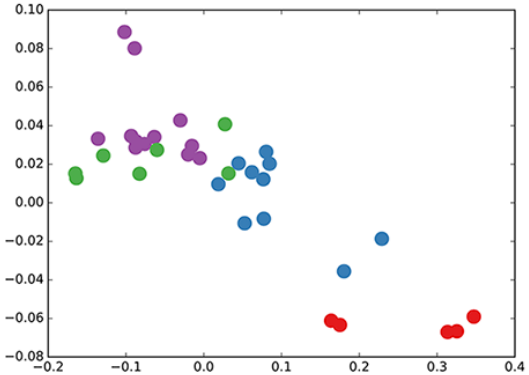
From https://petar-v.com/talks/GNN-Wednesday.pdf
https://www.youtube.com/watch?v=uF53xsT7mjc

Convolutional GNN

GCN (Kipf & Welling, ICLR'17)





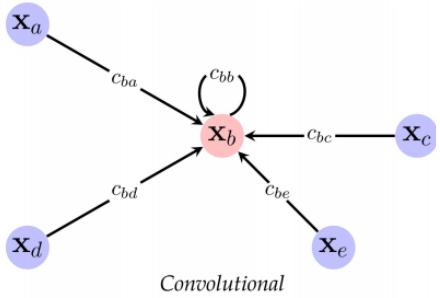


3-layer with random weights (untrained!)

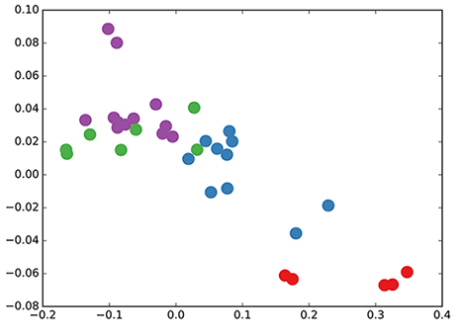
From https://petar-v.com/talks/GNN-Wednesday.pdf
https://www.youtube.com/watch?v=uF53xsT7mjc

Convolutional GNN

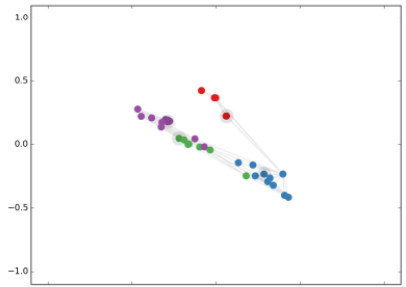
GCN (Kipf & Welling, ICLR'17)



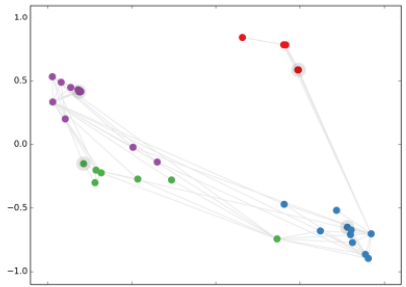
Convolutional



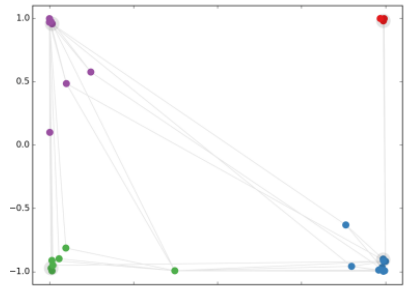
$X = I$



(a) Iteration 25



(b) Iteration 50



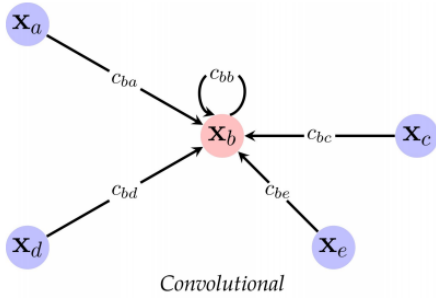
(f) Iteration 300

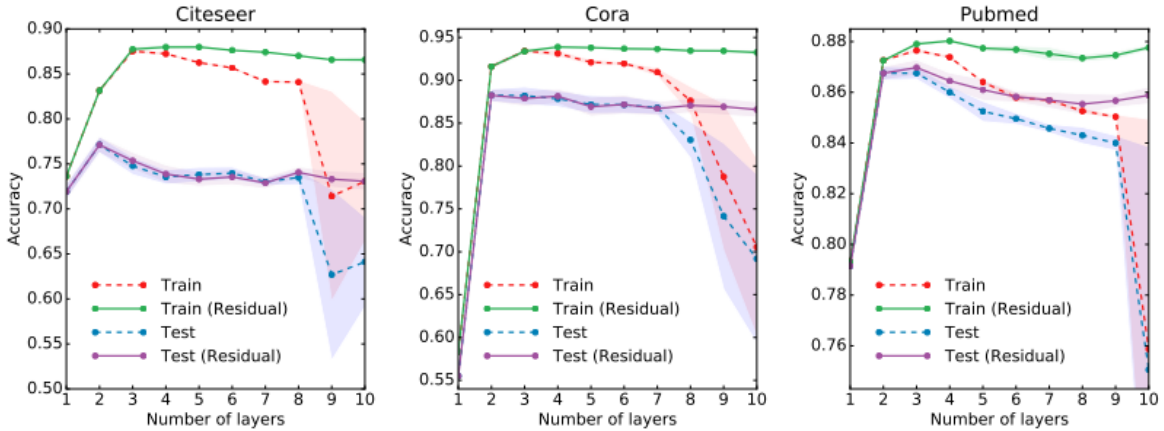
Iteration 0 $\Rightarrow$ Gets better as we learn the weights

From https://arxiv.org/pdf/1609.02907.pdf

Convolutional GNN

GCN (Kipf & Welling, ICLR'17)





More layers do not help due to over smoothing

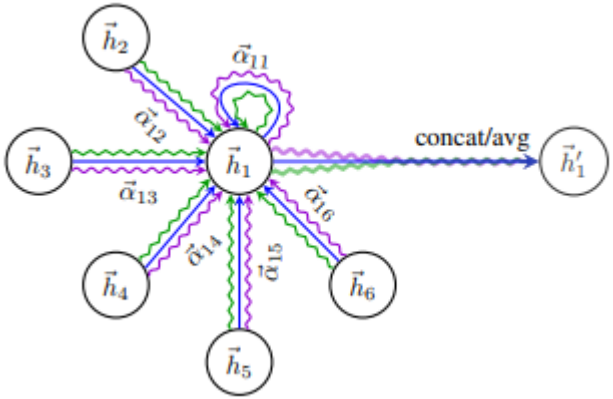
From https://arxiv.org/pdf/1609.02907.pdf

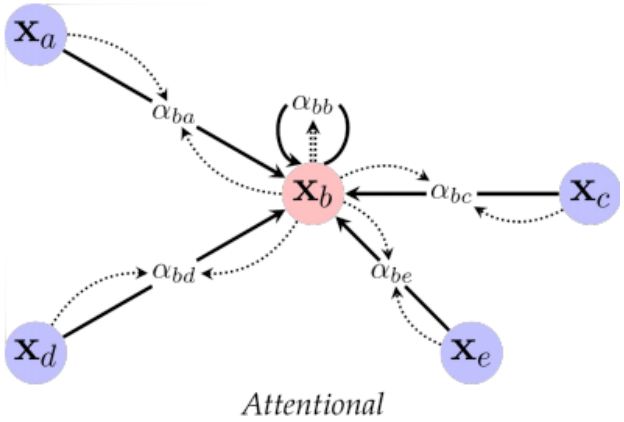
Attentional GNN

GAT (Veličković et al., ICLR'18)

compute scalar value in each edge

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v;$$



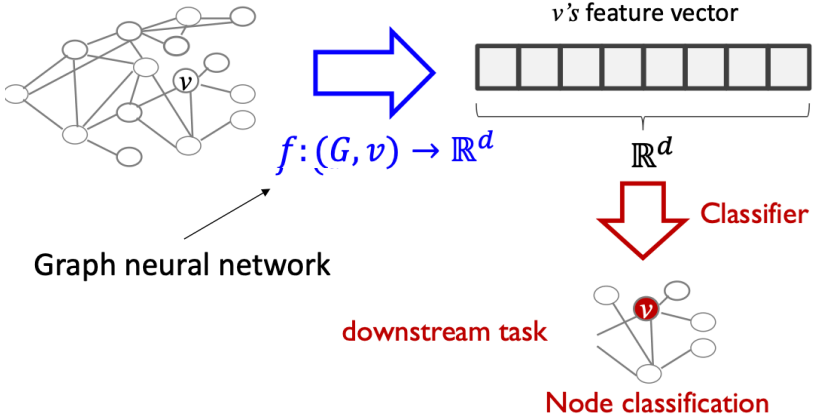


Attentional

Transductive			
Method	**Cora**	**Citeseer**	**Pubmed**
MLP	55.1%	46.5%	71.4%
ManiReg (Belkin et al., 2006)	59.5%	60.1%	70.7%
SemiEmb (Weston et al., 2012)	59.0%	59.6%	71.7%
LP (Zhu et al., 2003)	68.0%	45.3%	63.0%
DeepWalk (Perozzi et al., 2014)	67.2%	43.2%	65.3%
ICA (Lu & Getoor, 2003)	75.1%	69.1%	73.9%
Planetoid (Yang et al., 2016)	75.7%	64.7%	77.2%
Chebyshev (Defferrard et al., 2016)	81.2%	69.8%	74.4%
GCN (Kipf & Welling, 2017)	81.5%	70.3%	**79.0%**
MoNet (Monti et al., 2016)	81.7 ± 0.5%	—	78.8 ± 0.3%
GCN-64*	81.4 ± 0.5%	70.9 ± 0.5%	**79.0** ± 0.3%
GAT (ours)	**83.0** ± 0.7%	**72.5** ± 0.7%	**79.0** ± 0.3%

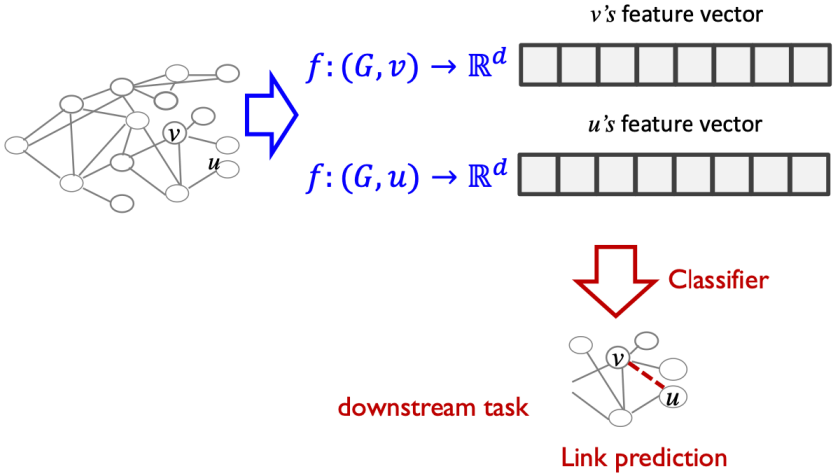
From https://petar-v.com/talks/GNN-Wednesday.pdf

Goal: Learn $f(G, v)$ for accurate node classification

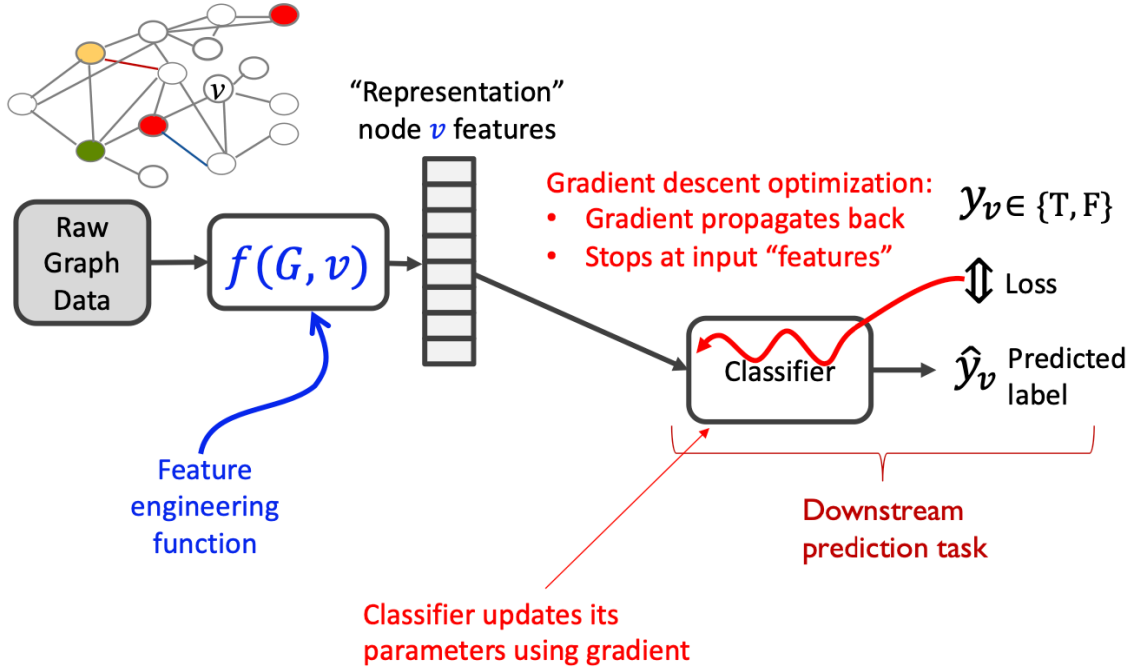


Overview: Link Prediction

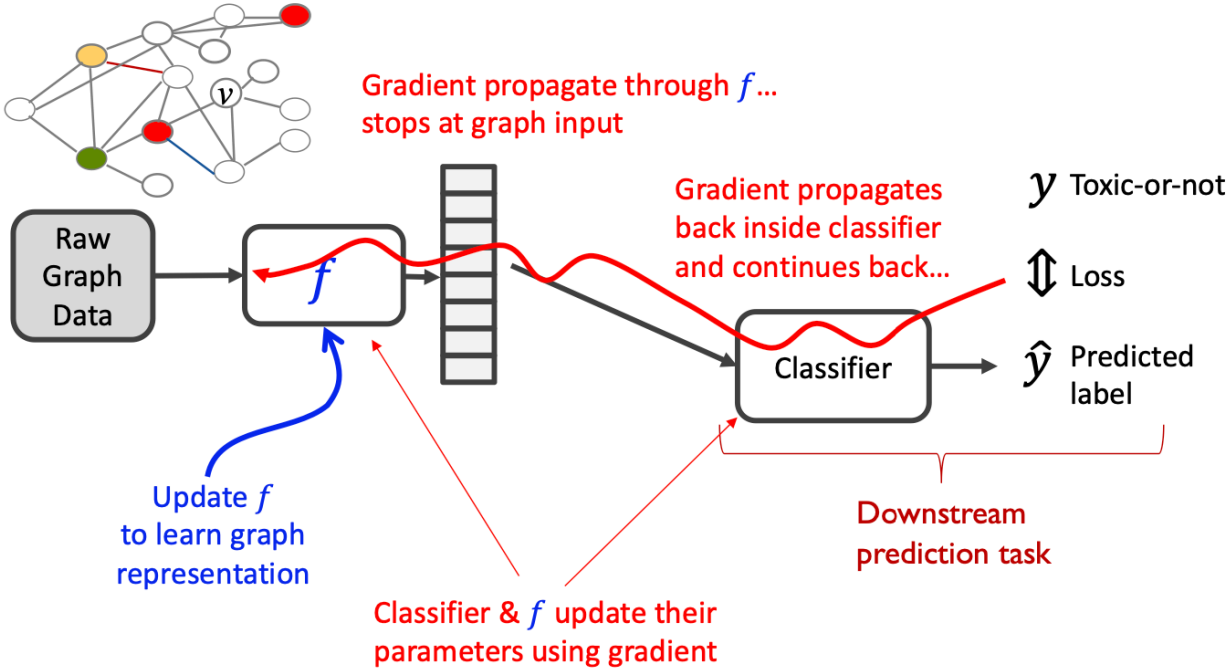
Goal: Learn $f(G, v)$ and $f(G, u)$ for accurate link prediction



Classical Supervised Learning Pipeline

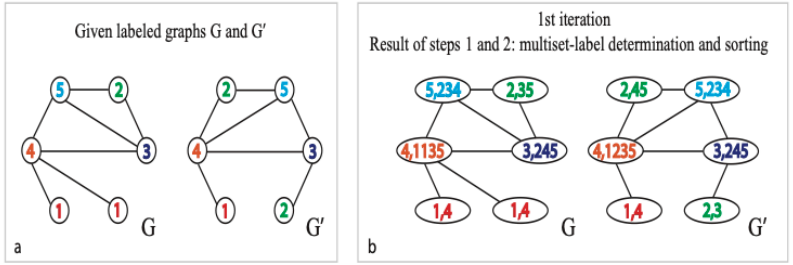


End-to-end Learning for Node Classification



Weisfeiler-Lehman (**WL-1**) Algorithm

- Weisfeiler-Lehman Algorithm (WL-1)
  - a.k.a. Color Refinement Algorithm
- Recursive algorithm to determine if two graphs are isomorphic
  - Works by recursively coloring nodes
  - Valid isomorphism test for most graphs (Babai and Kucera, 1979)
  - Cai et al., 1992 shows counter-examples not distinguished by it
  - Earliest works in graph kernels use these colors as features (Shervashidze et al., 2011)



Shervashidze et al. 2011

Initialize:

$\boldsymbol{h}_v$ = initial color of node v

$k = 0$

function WL-1-coloring(G):

while vertex attributes change **do**:

$k \leftarrow k + 1$

for all vertices $v \in G$ **do**

$$\boldsymbol{h}_{k,v} \leftarrow \text{hash}\left(\boldsymbol{h}_{k-1,v}, \left\{\boldsymbol{h}_{k-1,u} : \forall u \in \text{Neighbors}(v)\right\}\right)$$

Return $\{\boldsymbol{h}_{k,v} : \forall v \in G\}$

node v's color

neighbors of node v

Resources: Libraries and Datasets

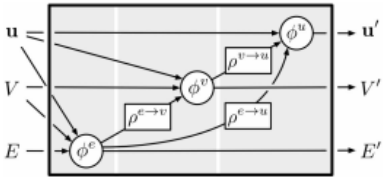
Ogb.stanford.edu

github.com/rusty1s/pytorch_geometric

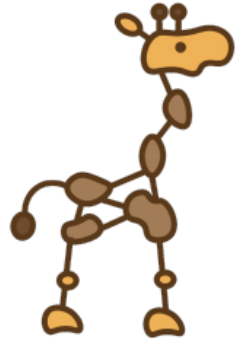
graphneural.network

Graphlearning.io

dgl.ai



github.com/deepmind/graph_nets



github.com/deepmind/jraph

https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html

github.com/graphdeeplearning/benchmarking-gnns

slides based on https://petar-v.com/talks/GNN-Wednesday.pdf