DGL 3 — How to train a GNN?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

https://github.com/udlbook/

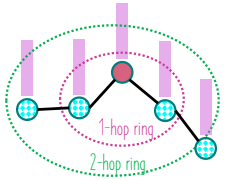
Overview: GCN training and testing (inductive)

Reminder

Learn feature embeddings (DGL-2)

A generalized message propagation rule to update the embeddings in a GCN layer is based on two steps:
(1) aggregate information from "myself" (a given node) and "my neighbors" (its neighbors), and
(2) apply a linear transformation.

$$H_{k+1} = \underbrace{f(H_k, A)}_{\text{generalized propagation rule}}$$



The node's neighborhood defines a computation graph over which we aggregate and transform embeddings.

a) Node embedding update

$$\mathbf{agg}[n,k] = \sum_{m \in \mathbf{ne}[n]} \mathbf{h}_k^{(m)},$$

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \mathbf{agg}[n,k]\right]$$

self-encoding — neighborhood encoding

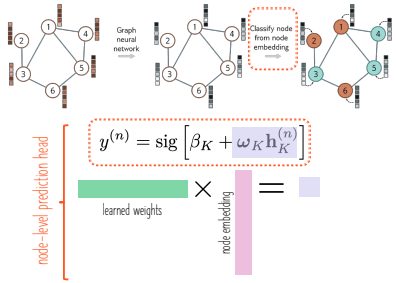
b) Update rule for all node embeddings

$$\begin{aligned}\mathbf{H}_{k+1} &= \mathbf{a}\left[\beta_k \mathbf{1}^T + \Omega_k \mathbf{H}_k + \Omega_k \mathbf{H}_k \mathbf{A}\right] \\ &= \mathbf{a}\left[\beta_k \mathbf{1}^T + \mathbf{\Omega}_k \mathbf{H}_k (\mathbf{A}+\mathbf{I})\right],\end{aligned}$$

Our ultimate goal: Use the learned embeddings in the final layer for downstream learning tasks (e.g., regression/classification).

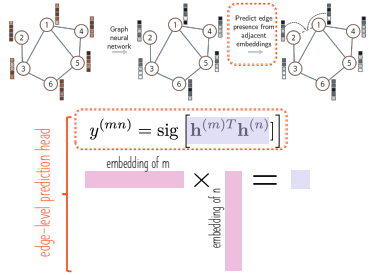
Graph learning tasks (DGL-1)

node classification



What are the parameters to learn?

edge/link prediction



How to learn and optimize these parameters?

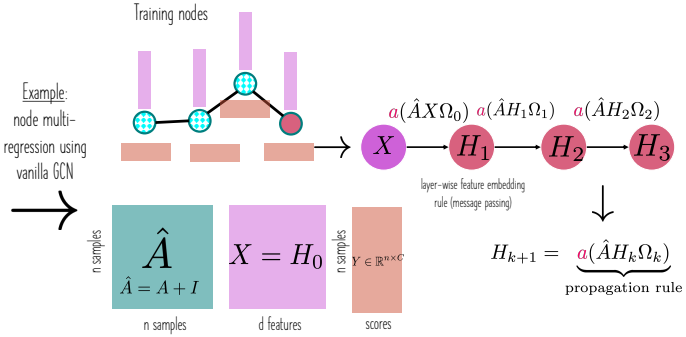
GCN training and its powerful inductive capability at testing

Note: we will examine supervised node- and graph-based classification/regression. In a similar way, we can adapt such formalization and steps to other learning tasks.

Design your GCN layer and optimize your loss functiob

1) Define a neighborhood aggregation function.
2) Define the building blocks (operations) of a GCN layer (e.g., linear transformation, nonlinear activation function).
3) Define your loss function to optimize.
Example: use binary or multiclass cross-entropy (CE) for classification and the least squares (or MSE) for regression. For classification, the output is passed through a sigmoid or softmax function then CE is applied.
4) Define the prediction head (node, edge, or graph), which may include additional hyperparameters (e.g., β and $\boldsymbol{\omega}$ encoding the classifier bias and weights to learn).
5) Optimize the loss over embeddings and additional hyperparameters (e.g., classifier weights and biases) during training on the training samples using stochastic gradient descent (SGD).
- Compute the loss during the forward pass.
- Compute the gradient of the loss using the chain rule during the backward pass.
- Update the network (model) parameters.

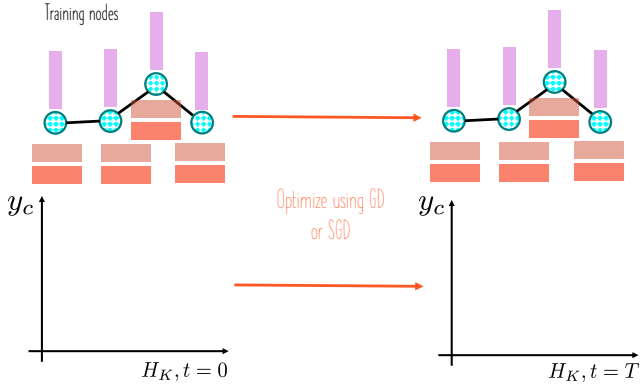
Example: node multi-regression using vanilla GCN



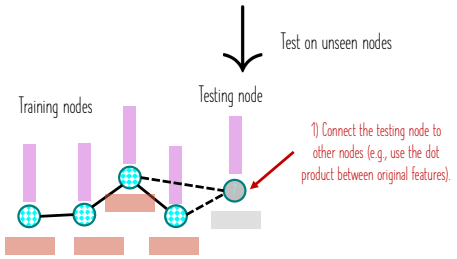
Loss optimization through gradient descent (let us imagine!)

$$\Theta_t = \Theta_{t-1} - \eta \nabla \mathcal{L}(\Theta_{t-1})$$

optimizer iterations



Reminder (DGL-2): Through this optimization, we aim to find' an ultimate low-dimensional representation space (a much simpler manifold) where linear models can work well on the final embeddings.

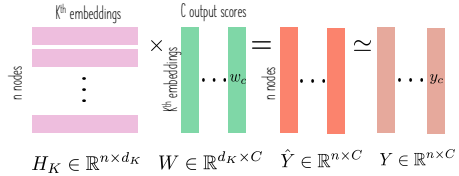


(Reminder) Non-linear multi-regression models

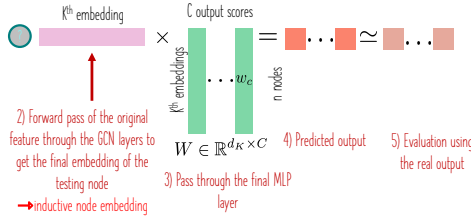
$$H_K = a(\hat{A}H_{K-1}\Omega_{K-1})$$

$$\mathcal{L}(\Theta) = ||H_K W - Y||_2^2 = \frac{1}{C}\sum_{c=1}^{C}(H_K w_c - y_c)^2$$

$\Theta =$ all network parameters $=$



Forward pass through the GNN layers and prediction head



BASIRA https://basira-lab.com/

YouTube Search "BASIRA Lab"

https://github.com/basiralab/DGL

DGL 3
How to train & design GNNs ?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

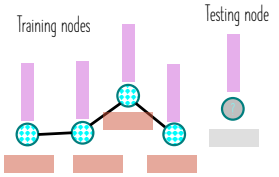

https://github.com/udlbook/

Overview: GCN training and testing (inductive)

GCN training and its powerful inductive capability at testing

Inductive capability of GCNs

What happens if we don't connect the node? Can we still predict its output scores?



☺ Yes! The node is self-connected, so the encoding will only comprise the "self-encoding" component and the "neighborhood encoding" is zero.

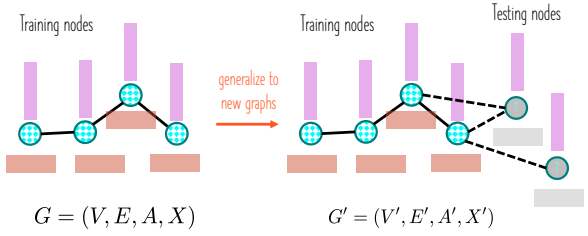
node embedding update

$$\mathbf{agg}[n,k] = \sum_{m \in \mathbf{ne}[n]} \mathbf{h}_k^{(m)},$$

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \mathbf{agg}[n,k]\right]$$

self-encoding | neighborhood encoding

Can GCN generalize to many new nodes as they arrive?



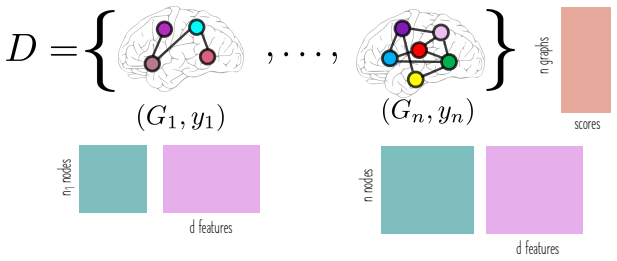
$G = (V, E, A, X)$ $\quad G' = (V', E', A', X')$

☺ One of the most compelling aspects of GNN models lies in their inductive capability where we can easily generalize to new graphs with varying sizes.
☺ This power stems from the feature aggregation and parameter sharing at the node level.

What happens if we don't share the parameters across all nodes?
→ We could learn a model with separate parameters associated with each node. However, the network must relearn the meaning of the connections in the graph at each position, and training would require many graphs with the same topology. Instead, we build a model that uses the same parameters at every node, reducing the number of parameters dramatically, and sharing what the network has learned at each node across the entire graph.

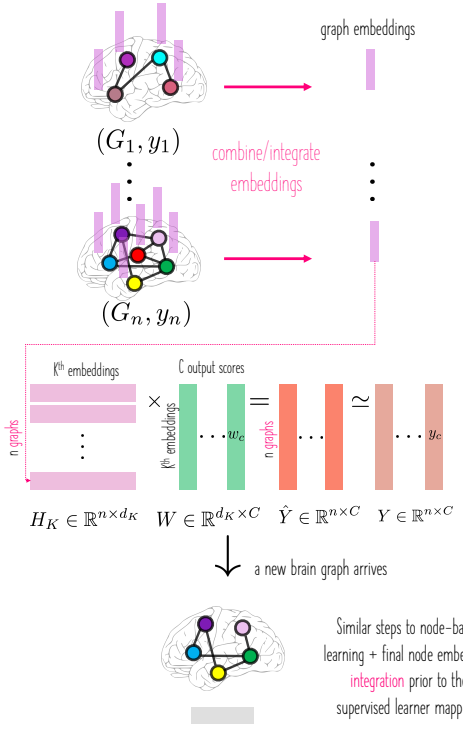
Can we classify graphs with different sizes?

Using ML or DL can we multi-regress this dataset of multiresolution brain graphs?



How can we extend the node-level regression to a graph-level regression?

Hint: leverage the power of integration.
→ integrate node embeddings across the whole graph and use the integrated embedding to solve the same optimization problem for node-based learning.



Similar steps to node-based learning + final node embedding integration prior to the supervised learner mapping.

Graph-level prediction head: How to combine the node embeddings across the entire graph?

Global mean pooling

$$h_K^G = Mean\{h_K^{(i)} \in \mathbb{R}^{d_K}, \forall i \in V_G\}$$

Global max pooling

$$h_K^G = Max\{h_K^{(i)} \in \mathbb{R}^{d_K}, \forall i \in V_G\}$$

Global sum pooling

$$h_K^G = Sum\{h_K^{(i)} \in \mathbb{R}^{d_K}, \forall i \in V_G\}$$

☹ Global pooling over a (large) graph will lose information and cannot differentiate between different graph structures.
Example: Let us find a case where two graphs G_1 and G_2 have different embedding distributions but their sum produces the same number (e.g., zero).
G_1 = {-0.5, 1, -0.6, 0,1} and G_2 = {18, 0, -5, -13}
→ We cannot differentiate between G_1 and G_2 using global sum pooling.

Hierarchical pooling (DiffPool 2018)

☺ This allows for a better graph differentiation. Think about a toy example that shows this..

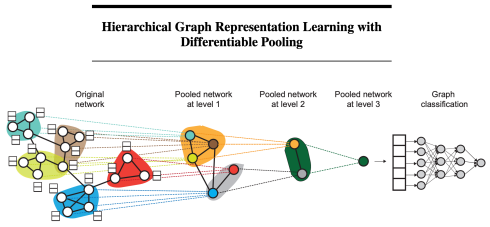


Figure 1: High-level illustration of our proposed method DIFFPOOL. At each hierarchical layer, we run a GNN model to obtain embeddings of nodes. We then use these learned embeddings to cluster nodes together and run another GNN layer on this coarsened graph. This whole process is repeated for L layers and we use the final output representation to classify the graph.

Ying, Zhitao, et al. "Hierarchical graph representation learning with differentiable pooling." Advances in neural information processing systems 31 (2018). https://proceedings.neurips.cc/paper_files/paper/2018/file/e77dbaf6759253c7c6d0efc5690369c7-Paper.pdf

DGL 3
How to train & design GNNs?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.


https://github.com/udlbook/

GNN layer operations (building blocks)

1) The operation order can change depending on the rationale and target task and a few operations can be **merged** or **removed**.

2) How and when to perform an operation? And why?
Example: how and when to "aggregate" and "transform"? We can first transform then aggregate or the other way around.

GNN layer operations

Transform
Classical in ML
Represent it better

MSG passing
The mixed actions of (AGG + Transform)
$H_{k+1} = \underbrace{f(H_k, A)}_{\text{generalized propagation rule}}$

AGG
Graph-based aggregation (the GNN hallmark)
Integrate information from self & neighbors

BN
Stabilize the optimization & control embedding variance between layers

Prediction head
Define the target learning task & its params

Non-linearity
Capture complexity & identify important patterns

Dropout
Drop nodes to regularize & increase expressiveness

BASIRA
https://basira-lab.com/

YouTube
Search "BASIRA Lab"

https://github.com/basiralab/DGL

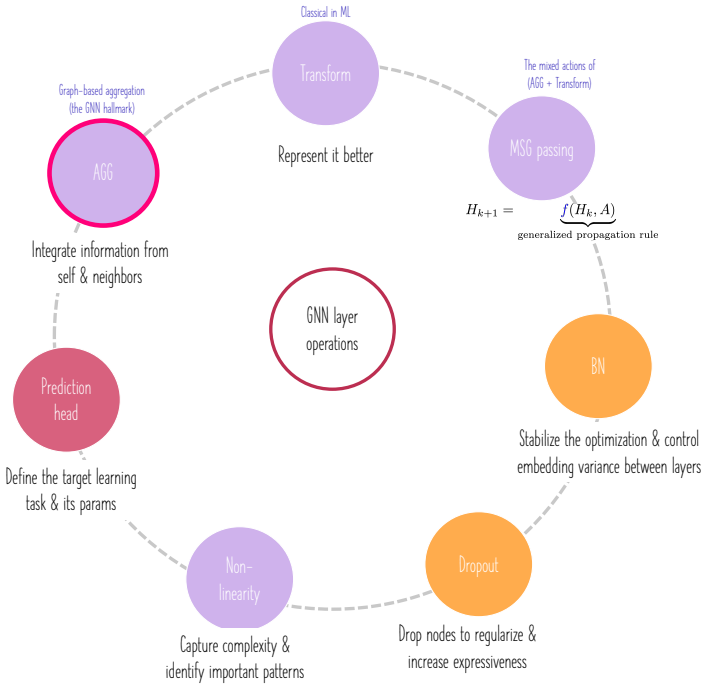
DGL 3

How to train & design GNNs ?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

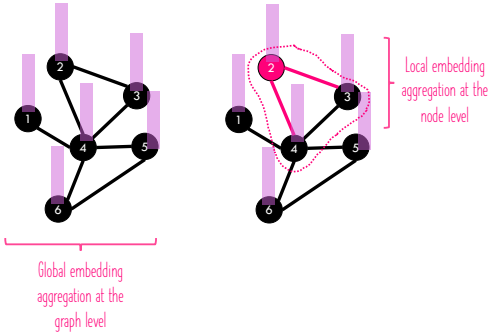
https://github.com/udlbook/

GNN layer operations (building blocks)



More local and global aggregation methods (UDL-13)

How to aggregate information locally (at the node level) or globally (at the graph level)?



Mean aggregation

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot \mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k \cdot \mathbf{agg}[n,k]\right]$$

summing the embeddings of neighbors → averaging the embeddings of neighbors

$$\mathbf{agg}[n] = \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_m \longrightarrow \mathbf{agg}[n] = \frac{1}{|\mathrm{ne}[n]|} \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_m$$

↓

The new GCN layer can be written using the degree matrix as:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{A}\mathbf{D}^{-1} + \mathbf{I})\right]$$

Kipf normalization

The logic behind this: information coming from nodes with a very large number of neighbors should be downweighted since there are many connections and they provide less unique information

$$\mathbf{agg}[n] = \sum_{m \in \mathrm{ne}[n]} \frac{\mathbf{h}_m}{\sqrt{|\mathrm{ne}[n]||\mathrm{ne}[m]|}},$$

↓

The new GCN layer can be written as:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} + \mathbf{I})\right]$$

Max pooling

Max-pooling can also be applied at the node level to update the node embedding.
The operator max[•] returns the element-wise maximum of the embeddings that are neighbors to node n.

$$\mathbf{agg}[n] = \max_{m \in \mathrm{ne}[n]} [\mathbf{h}_m]$$

Aggregation by attention

Reminder: Previous aggregation methods consider the neighbors equally or in a way that depends on the graph topology → the aggregation depends on the node embeddings and local topology.

Rule: first transform, then pay attention.

1) Linearly transform the embeddings of a current node as follows:

$$\mathbf{H}'_k = \boldsymbol{\beta}_k \mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{\mathbf{k}} \mathbf{H}.$$

Transformed embeddings

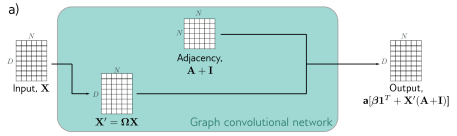


Figure 13.12 Comparison of graph convolutional network, dot product attention, and graph attention network. In each case, the mechanism maps N embeddings of size D stored in a $D \times N$ matrix $\mathbf{X}$ to an output of the same size. The graph convolutional network applies a linear transformation $\mathbf{X}' = \boldsymbol{\Omega}\mathbf{X}$ to the data matrix. It then computes a weighted sum of the transformed data, where the weighting is based on the adjacency matrix. A bias $\boldsymbol{\beta}$ is added and the result is passed through an activation function. b) The outputs of the self-attention

2) Define a node-to-node similarity matrix that will act as an attention matrix in the embedding aggregation step:

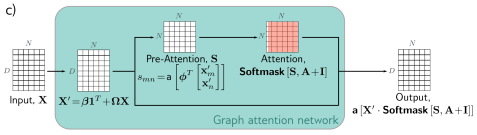
$$s_{mn} = \mathbf{a}\left[\boldsymbol{\phi}_k^T \begin{bmatrix} \mathbf{h}'_m \\ \mathbf{h}'_n \end{bmatrix}\right] \longrightarrow \mathbf{S}$$

(nodes × nodes)

As in dot-product self-attention, the attention weights that contribute to each output embedding are normalized to be positive and sum to one using the softmax operation. These weights are applied to the transformed embeddings:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\mathbf{H}'_k \cdot \mathbf{Softmask}[\mathbf{S}, \mathbf{A} + \mathbf{I}]\right]$$

The function Softmask[•,•] applies the softmax operation separately to each column of its first argument, setting values where the second argument is zero to negative infinity. This has the effect of ensuring that the attention to non-neighboring nodes is zero.
The attentions are masked so that each node only attends to itself and its neighbors.



The graph attention network combines both of these mechanisms; the weights are both computed from the data (i.e., embeddings) and based on the adjacency matrix (graph topology).

BASIRA https://basira-lab.com/

YouTube Search "BASIRA Lab"

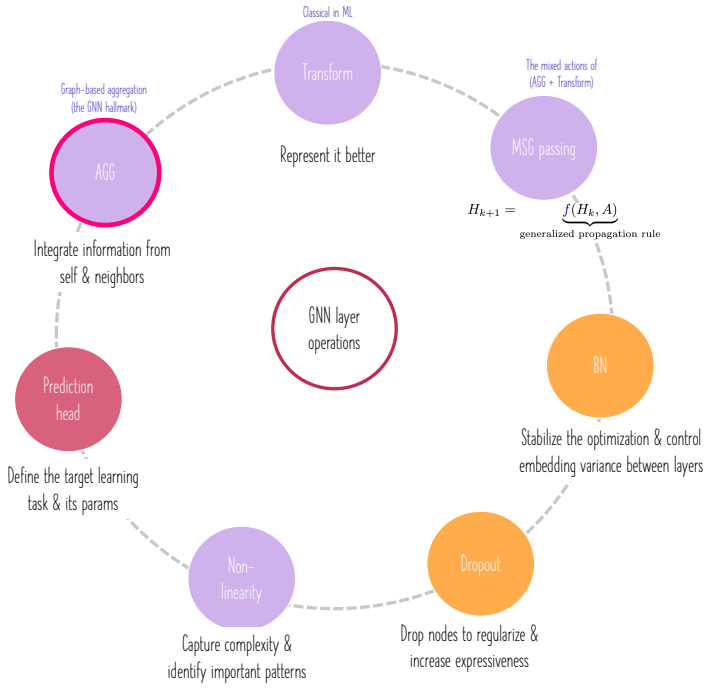
https://github.com/basiralab/DGL

DGL 3

How to train & design GNNs ?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

https://github.com/udlbook/



Recall box: summarize what you remember from this lecture below

Extra resources and links:

- Graph convolutional neural networks https://mbernste.github.io/posts/gcn
- A Gentle Introduction to Graph Neural Networks https://distill.pub/2021/gnn-intro
- Graph Convolutional Networks https://tkipf.github.io/graph-convolutional-networks/

BASIRA

https://basira-lab.com/

YouTube Search "BASIRA Lab"

https://github.com/basiralab

YOU CAN TAKE NOTES
ON THE FOLLOWING SLIDES

Let us reflect and self-quiz!

DGL 3
How to train a design GNNs?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.


https://github.com/udlbook/

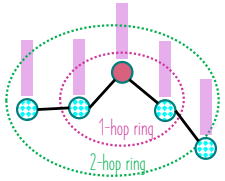
Overview: GCN training and testing (inductive)

Reminder

Learn feature embeddings (DGL-2)

A generalized message propagation rule to update the embeddings in a GCN layer is based on two steps:
(1) aggregate information from "myself" (a given node) and "my neighbors" (its neighbors), and
(2) apply a linear transformation.

$$H_{k+1} = \underbrace{f(H_k, A)}_{\text{generalized propagation rule}}$$



The node's neighborhood defines a computation graph over which we aggregate and transform embeddings.

a) Node embedding update

$$\mathbf{agg}[n,k] = \sum_{m \in \mathbf{ne}[n]} \mathbf{h}_k^{(m)},$$

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \underbrace{\Omega_k \cdot \mathbf{h}_k^{(n)}}_{\text{self-encoding}} + \underbrace{\Omega_k \cdot \mathbf{agg}[n,k]}_{\text{neighborhood encoding}}\right]$$

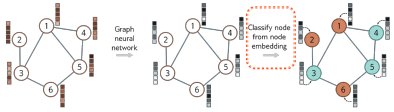
b) Update rule for all node embeddings

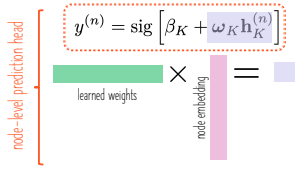
$$\begin{aligned}\mathbf{H}_{k+1} &= \mathbf{a}\left[\beta_k \mathbf{1}^T + \Omega_k \mathbf{H}_k + \Omega_k \mathbf{H}_k \mathbf{A}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{A} + \mathbf{I})\right],\end{aligned}$$

Our ultimate goal: Use the learned embeddings in the final layer for downstream learning tasks (e.g., regression/classification).

Graph learning tasks (DGL-1)

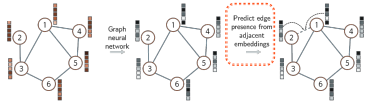
node classification

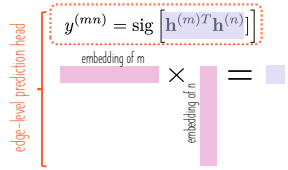




What are the parameters to learn?

edge/link prediction





How to learn and optimize these parameters?

Personal reflections and notes

DGL 3
How to train & design GNNs?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

https://github.com/udlbook/

Overview: GCN training and testing (inductive)

Personal reflections and notes

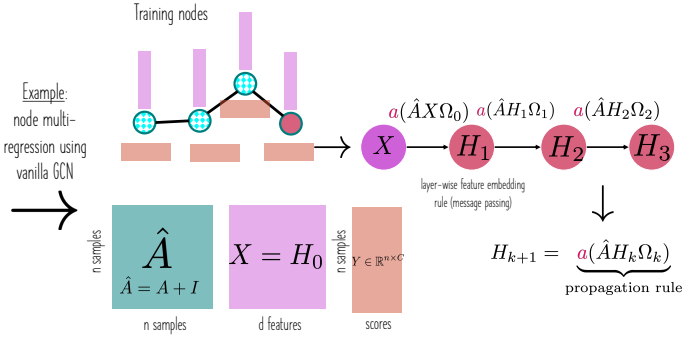
GCN training and its powerful inductive capability at testing

Note: we will examine supervised node- and graph-based classification/regression. In a similar way, we can adapt such formalization and steps to other learning tasks.

Design your GCN layer and optimize your loss functiob

1) Define a neighborhood aggregation function.
2) Define the building blocks (operations) of a GCN layer (e.g., linear transformation, nonlinear activation function).
3) Define your loss function to optimize.
   Example: use binary or multiclass cross-entropy (CE) for classification and the least squares (or MSE) for regression. For classification, the output is passed through a sigmoid or softmax function then CE is applied.
4) Define the prediction head (node, edge, or graph), which may include additional hyperparameters (e.g., $\boldsymbol{\beta}$ and $\boldsymbol{\omega}$ encoding the classifier bias and weights to learn).
5) Optimize the loss over embeddings and additional hyperparameters (e.g., classifier weights and biases) during training on the training samples using stochastic gradient descent (SGD).
   - Compute the loss during the forward pass.
   - Compute the gradient of the loss using the chain rule during the backward pass.
   - Update the network (model) parameters.

Example: node multi-regression using vanilla GCN

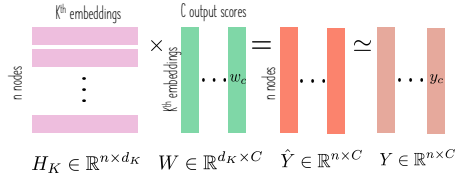


(Reminder) Non-linear multi-regression models

$$H_K = a(\hat{A}H_{K-1}\Omega_{K-1})$$

$$\mathcal{L}(\Theta) = ||H_K W - Y||_2^2 = \frac{1}{C}\sum_{c=1}^{C}(H_K w_c - y_c)^2$$

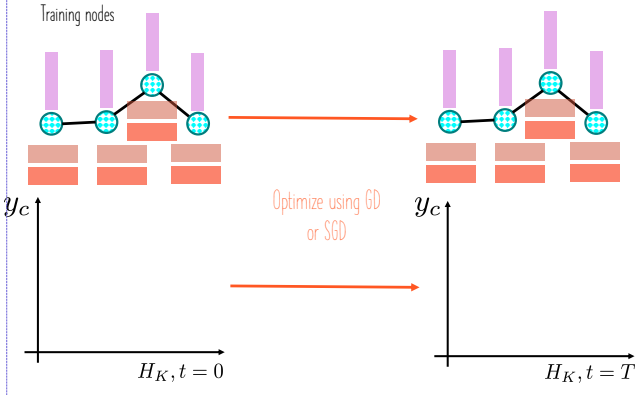
$\Theta =$ all network parameters $=$



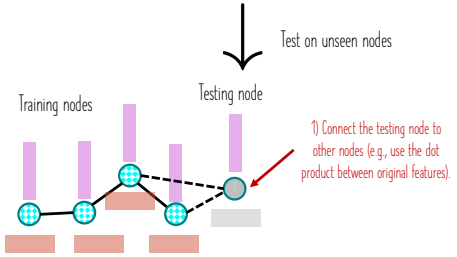
Loss optimization through gradient descent (let us imagine!)

$$\Theta_t = \Theta_{t-1} - \eta\nabla\mathcal{L}(\Theta_{t-1})$$

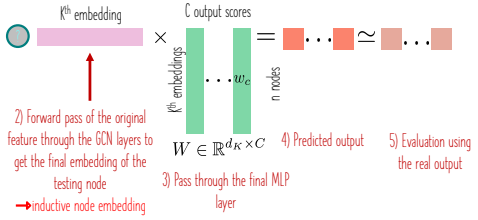
optimizer iterations



Reminder (DGL-2): Through this optimization, we aim to find' an ultimate low-dimensional representation space (a much simpler manifold) where linear models can work well on the final embeddings.



Forward pass through the GNN layers and prediction head



BASIRA
https://basira-lab.com/

YouTube Search "BASIRA Lab"

https://github.com/basiralab/DGL

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

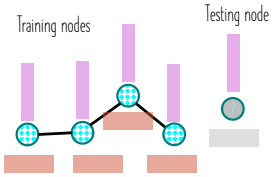
https://github.com/udlbook/

Overview: GCN training and testing (inductive)

GCN training and its powerful inductive capability at testing

Inductive capability of GCNs

What happens if we don't connect the node? Can we still predict its output scores?



☺ Yes! The node is self-connected, so the encoding will only comprise the "self-encoding" component and the "neighborhood encoding" is zero.

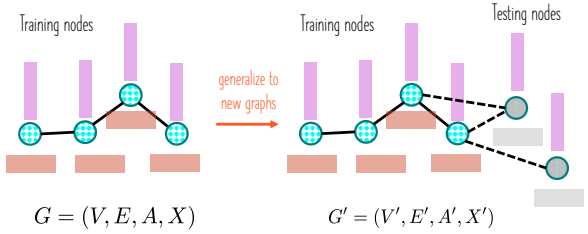
node embedding update

$$\mathbf{agg}[n,k] = \sum_{m \in \mathbf{ne}[n]} \mathbf{h}_k^{(m)},$$

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \mathbf{agg}[n,k]\right]$$

self-encoding | neighborhood encoding

Can GCN generalize to many new nodes as they arrive?



$G = (V, E, A, X)$ $\quad$ $G' = (V', E', A', X')$

☺ One of the most compelling aspects of GNN models lies in their inductive capability where we can easily generalize to new graphs with varying sizes.
☺ This power stems from the feature aggregation and parameter sharing at the node level.

What happens if we don't share the parameters across all nodes?
→ We could learn a model with separate parameters associated with each node. However, the network must relearn the meaning of the connections in the graph at each position, and training would require many graphs with the same topology. Instead, we build a model that uses the same parameters at every node, reducing the number of parameters dramatically, and sharing what the network has learned at each node across the entire graph.

Personal reflections and notes

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

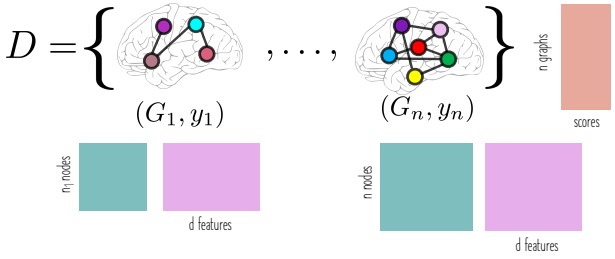
https://github.com/udlbook/

Overview: GCN training and testing (inductive)

GCN training and its powerful inductive capability at testing

Can we classify graphs with different sizes?

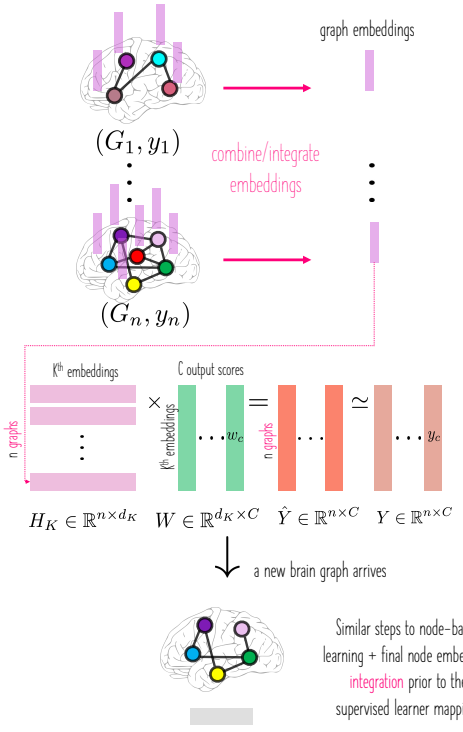
Using ML or DL can we multi-regress this dataset of multiresolution brain graphs?



How can we extend the node-level regression to a graph-level regression?

Hint: leverage the power of integration.

→ integrate node embeddings across the whole graph and use the integrated embedding to solve the same optimization problem for node-based learning.



Similar steps to node-based learning + final node embedding integration prior to the supervised learner mapping.

Personal reflections and notes

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

https://github.com/udlbook/

Overview: GCN training and testing (inductive)

GCN training and its powerful inductive capability at testing

Graph-level prediction head: How to combine the node embeddings across the entire graph?

Global mean pooling

$$h_G = Mean\{h_K^{(i)} \in \mathbb{R}^{d_K}, \forall i \in V_G\}$$

Global max pooling

$$h_G = Max\{h_K^{(i)} \in \mathbb{R}^{d_K}, \forall i \in V_G\}$$

Global sum pooling

$$h_G = Sum\{h_K^{(i)} \in \mathbb{R}^{d_K}, \forall i \in V_G\}$$

☹ Global pooling over a (large) graph will lose information and cannot differentiate between different graph structures.

Example: Let us find a case where two graphs G_1 and G_2 have different embedding distributions but their sum produces the same number (e.g., zero).

G_1 = [-0.5, 1, -0.6, 0,1] and G_2 = [18, 0, -5, -13]

→ We cannot differentiate between G_1 and G_2 using global sum pooling.

Hierarchical pooling (DiffPool 2018)

☺ This allows for a better graph differentiation. Think about a toy example that shows this..

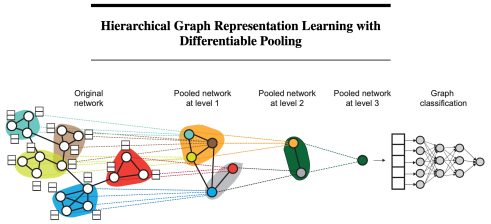


Figure 1: High-level illustration of our proposed method DiffPool. At each hierarchical layer, we run a GNN model to obtain embeddings of nodes. We then use these learned embeddings to cluster nodes together and run another GNN layer on this coarsened graph. This whole process is repeated for L layers and we use the final output representation to classify the graph.

Ying, Zhitao, et al. "Hierarchical graph representation learning with differentiable pooling." Advances in neural information processing systems 31 (2018). https://proceedings.neurips.cc/paper_files/paper/2018/file/e77dbaf6759253c7c6d0efc5690369c7-Paper.pdf

Personal reflections and notes

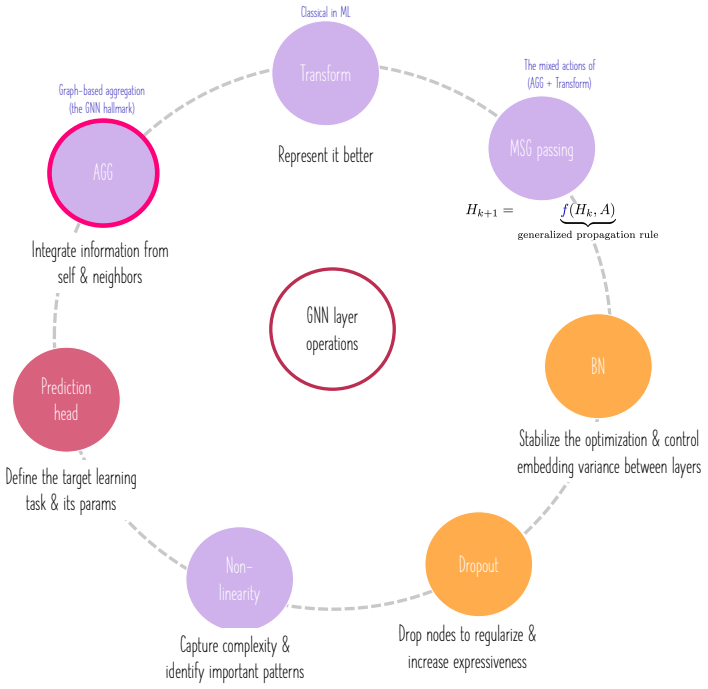
DGL 3

How to train & design GNNs?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

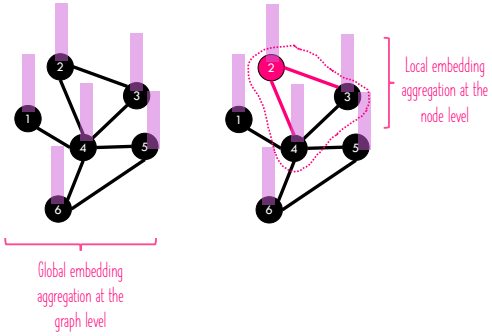
https://github.com/udlbook/

GNN layer operations (building blocks)



More local and global aggregation methods (UDL-13)

How to aggregate information locally (at the node level) or globally (at the graph level)?



Mean aggregation

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot \mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k \cdot \mathbf{agg}[n,k]\right]$$

summing the embeddings of neighbors

$$\mathbf{agg}[n] = \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_m$$

averaging the embeddings of neighbors

$$\longrightarrow \quad \mathbf{agg}[n] = \frac{1}{|\mathrm{ne}[n]|} \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_m$$

↓

The new GCN layer can be written using the degree matrix as:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{A}\mathbf{D}^{-1} + \mathbf{I})\right]$$

Kipf normalization

The logic behind this: information coming from nodes with a very large number of neighbors should be downweighted since there are many connections and they provide less unique information

$$\mathbf{agg}[n] = \sum_{m \in \mathrm{ne}[n]} \frac{\mathbf{h}_m}{\sqrt{|\mathrm{ne}[n]||\mathrm{ne}[m]|}},$$

↓

The new GCN layer can be written as:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} + \mathbf{I})\right]$$

Max pooling

Max-pooling can also be applied at the node level to update the node embedding.
The operator max[•] returns the element-wise maximum of the embeddings that are neighbors to node n.

$$\mathbf{agg}[n] = \max_{m \in \mathrm{ne}[n]} [\mathbf{h}_m]$$

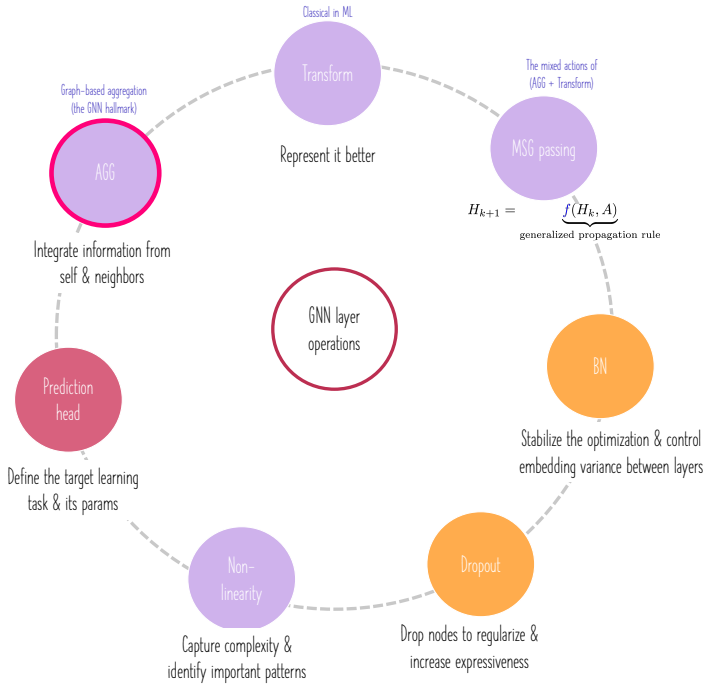
Personal reflections and notes

DGL 3
How to train & design GNNs ?

Special thanks to Simon Prince, Alex Fornito, Andrew Zalesky, Edward Bullmore, Jure Leskovec and all those who shared their passion about graphs and deep learning.

https://github.com/udlbook/

GNN layer operations (building blocks)



More local and global aggregation methods (UDL-13)

Personal reflections and notes

Aggregation by attention

Reminder: Previous aggregation methods consider the neighbors equally or in a way that depends on the graph topology → the aggregation depends on the node embeddings and local topology.

Rule: first transform, then pay attention.

1) Linearly transform the embeddings of a current node as follows:

$$\mathbf{H}'_k = \boldsymbol{\beta}_k \mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{\mathbf{k}} \mathbf{H}.$$

Transformed embeddings

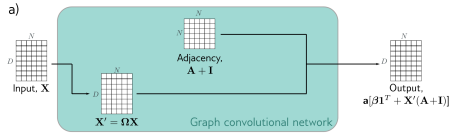


Figure 13.12 Comparison of graph convolutional network, dot product attention, and graph attention network. In each case, the mechanism maps N embeddings of size D stored in a $D \times N$ matrix $\mathbf{X}$ to an output of the same size. The graph convolutional network applies a linear transformation $\mathbf{X}' = \boldsymbol{\Omega}\mathbf{X}$ to the data matrix. It then computes a weighted sum of the transformed data, where the weighting is based on the adjacency matrix. A bias $\boldsymbol{\beta}$ is added and the result is passed through an activation function. b) The outputs of the self-attention

2) Define a node-to-node similarity matrix that will act as an attention matrix in the embedding aggregation step:

$$s_{mn} = \mathrm{a}\left[\boldsymbol{\phi}_k^T \begin{bmatrix} \mathbf{h}'_m \\ \mathbf{h}'_n \end{bmatrix}\right] \longrightarrow \mathbf{S}$$

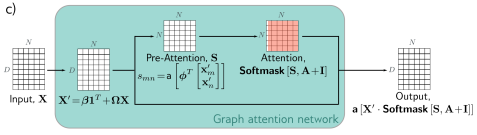
(nodes × nodes)

As in dot-product self-attention, the attention weights that contribute to each output embedding are normalized to be positive and sum to one using the softmax operation. These weights are applied to the transformed embeddings:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\mathbf{H}'_k \cdot \mathbf{Softmask}[\mathbf{S}, \mathbf{A}+\mathbf{I}]\right]$$

The function Softmask[•,•] applies the softmax operation separately to each column of its first argument, setting values where the second argument is zero to negative infinity. This has the effect of ensuring that the attention to non-neighboring nodes is zero.

The attentions are masked so that each node only attends to itself and its neighbors.



The graph attention network combines both of these mechanisms; the weights are both computed from the data (i.e., embeddings) and based on the adjacency matrix (graph topology).