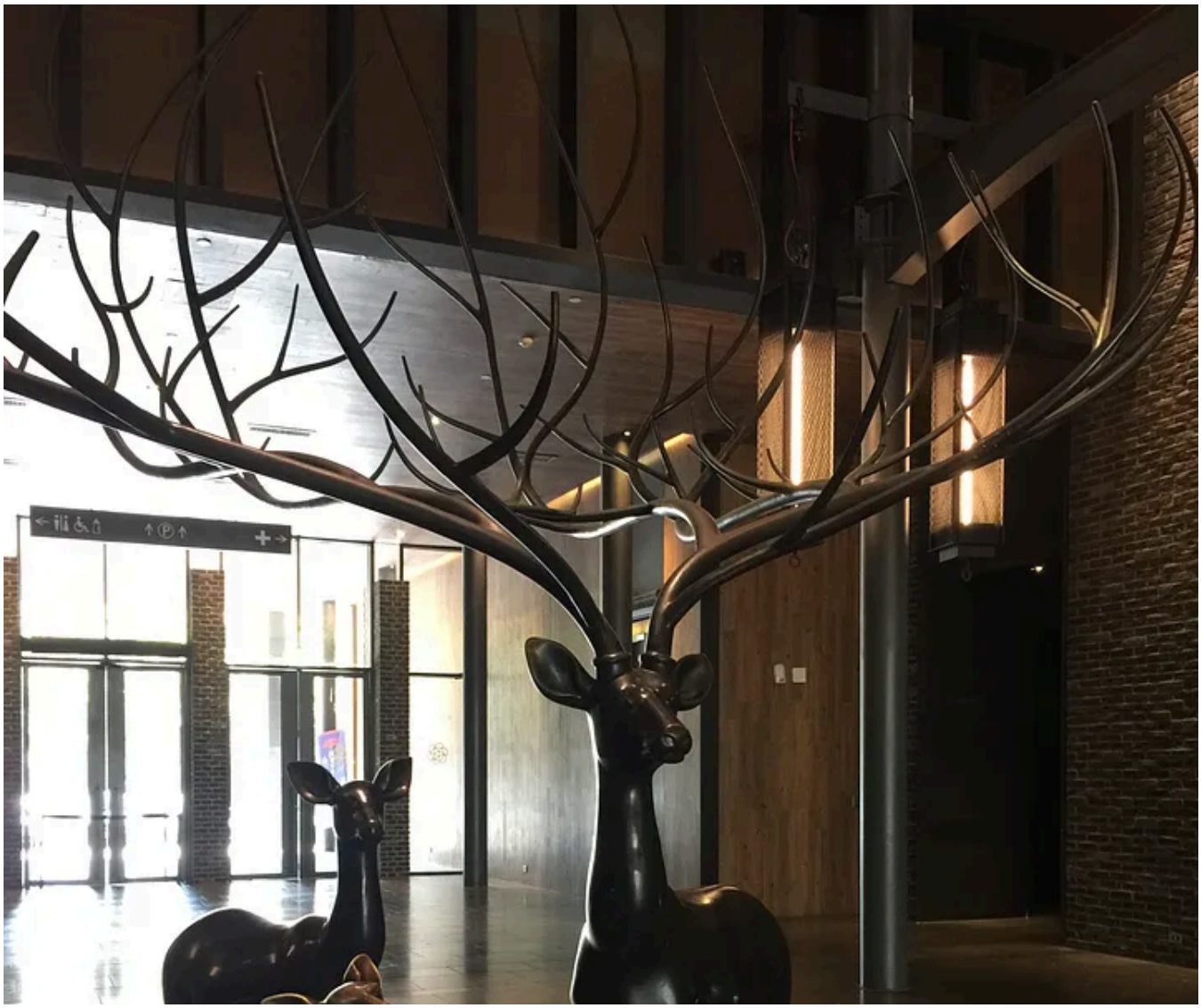
Graph Convolutional Networks (GCN) & Pooling

Jonathan Hui · Follow

21 min read · Feb 24, 2021

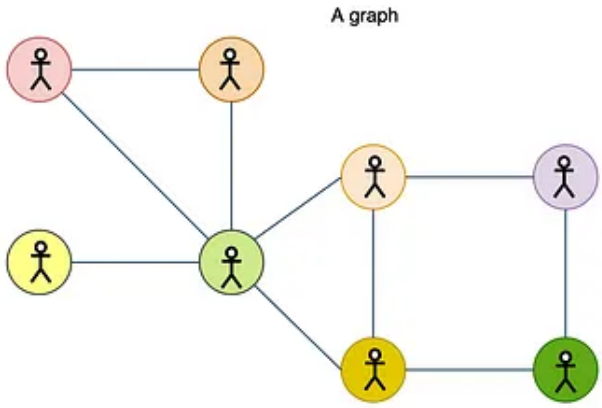
Listen Share More



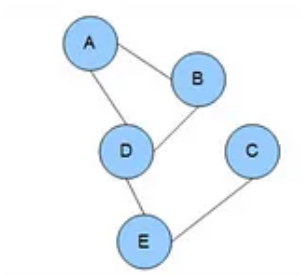
> *You know, who you choose to be around you, let's you know who you are. — The Fast and the Furious: Tokyo Drift.*

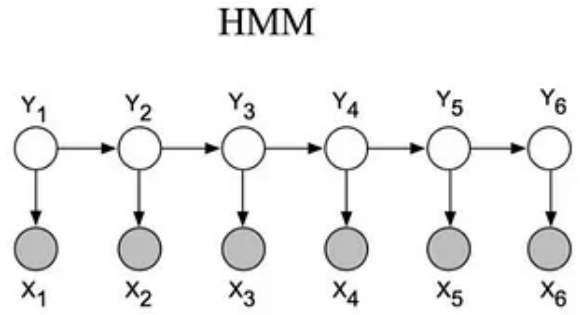
What is the challenge?

In social networks, friend connections can be realized by a social graph.

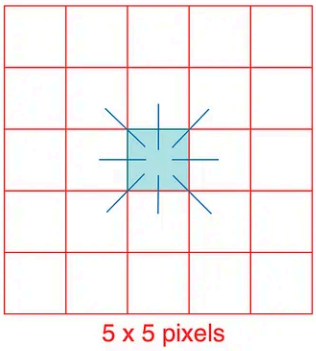


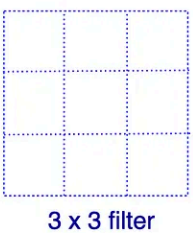
In speech recognition, the phoneme Y_i and the acoustic model x_i form an HMM (a graph for speech recognition).

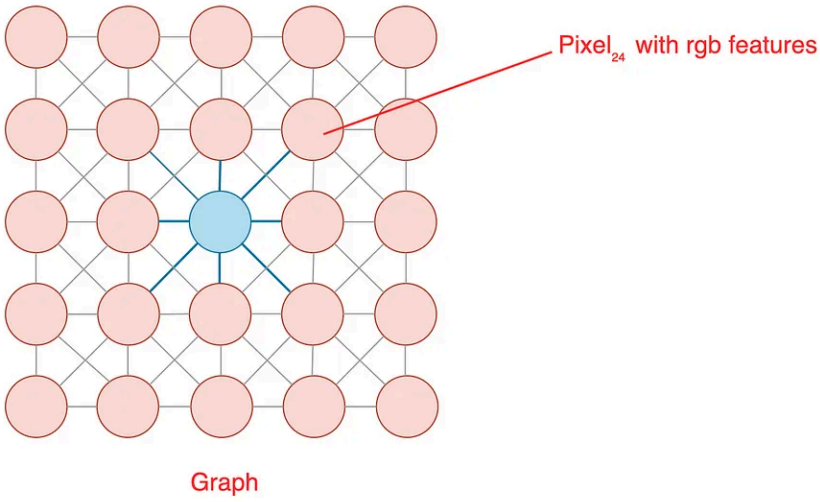




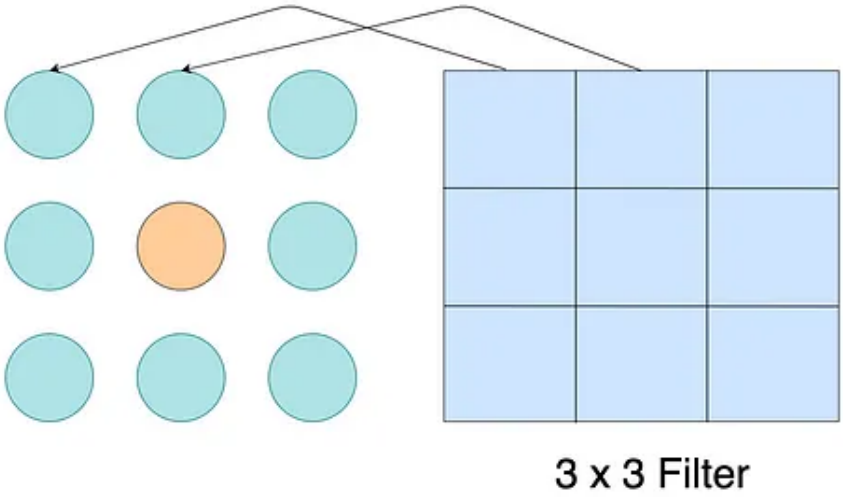
Even on CNN, an input image can be modeled as a graph. For example, the right diagram below is the graph for a 5 × 5 image. Each node represents a pixel and for the case of a 3 × 3 filter, every node is connected to its eight immediate neighbors.



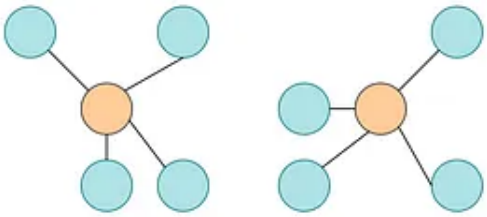




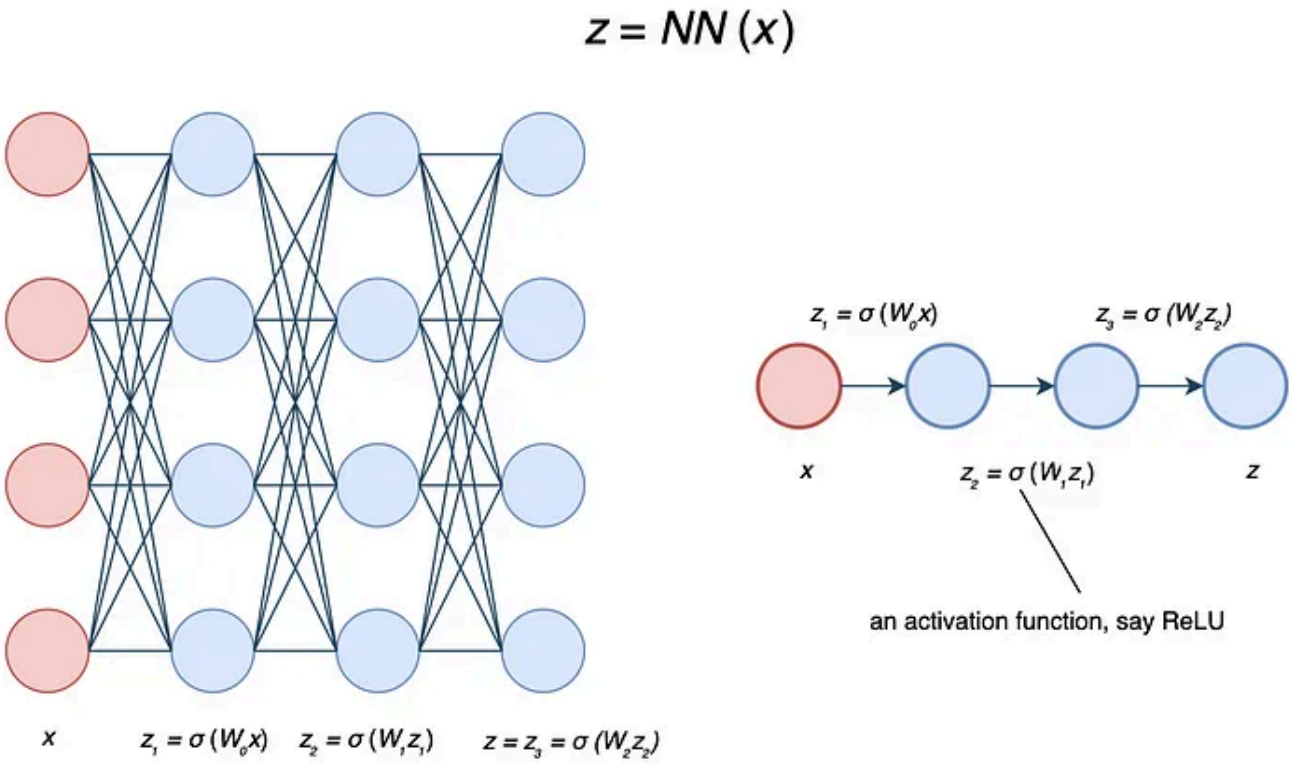
Even though it is overkill to represent an image this way, a large percentage of machine learning (ML) problems will be much natural and effective to be modeled by a graph. In particular, when the relationships between neighboring nodes are irregular and high dimensional, we need to define them explicitly in order to solve them efficiently. In CNN, we work in a Euclidean space. How weights are associated with the input features (pixels) is well defined.



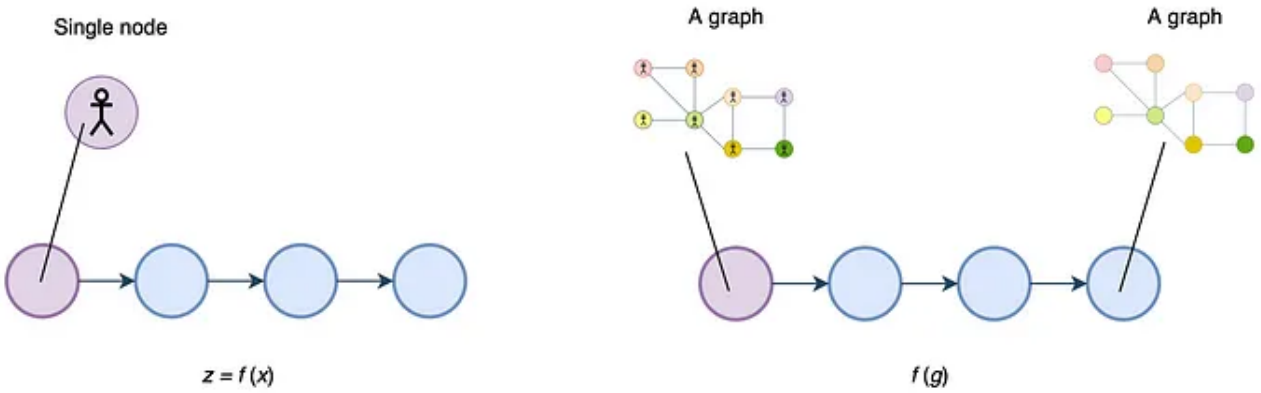
But this is not the case for a graph. For example, the graphs below are the same even though it looks different spatially.



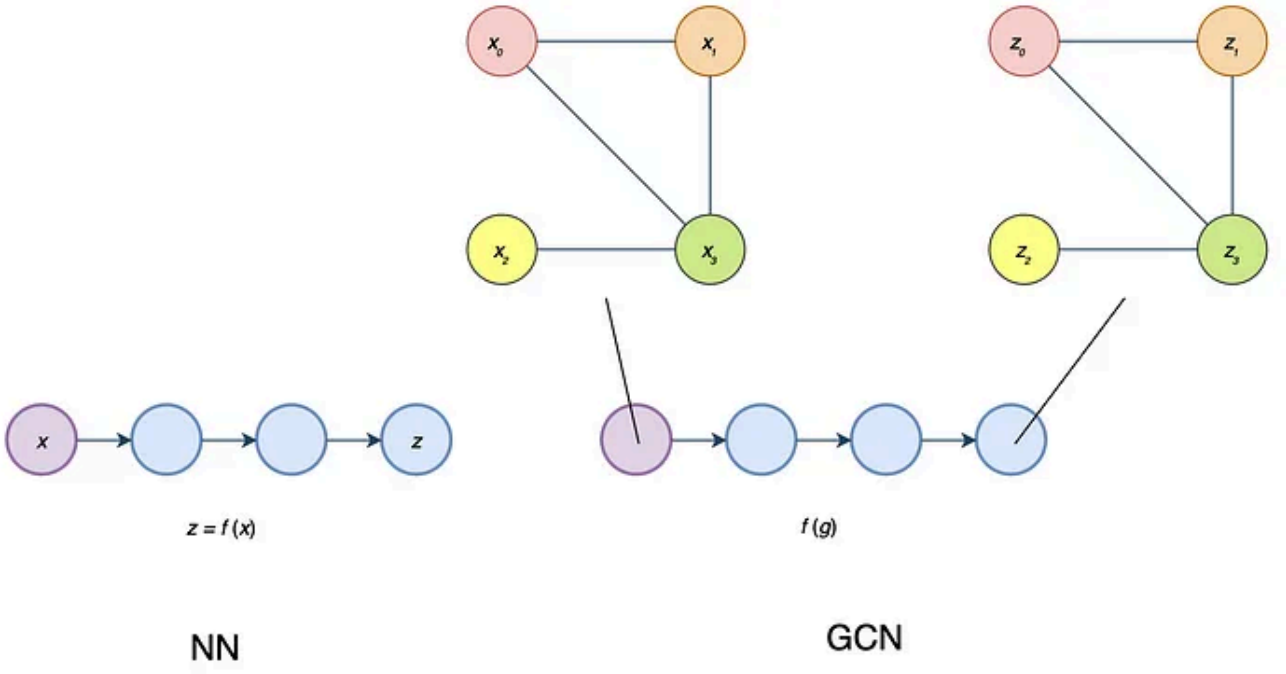
In general, neural networks (NNs) takes an input x to predict z.



This leads us to the challenge of how a NN can process a graph directly.



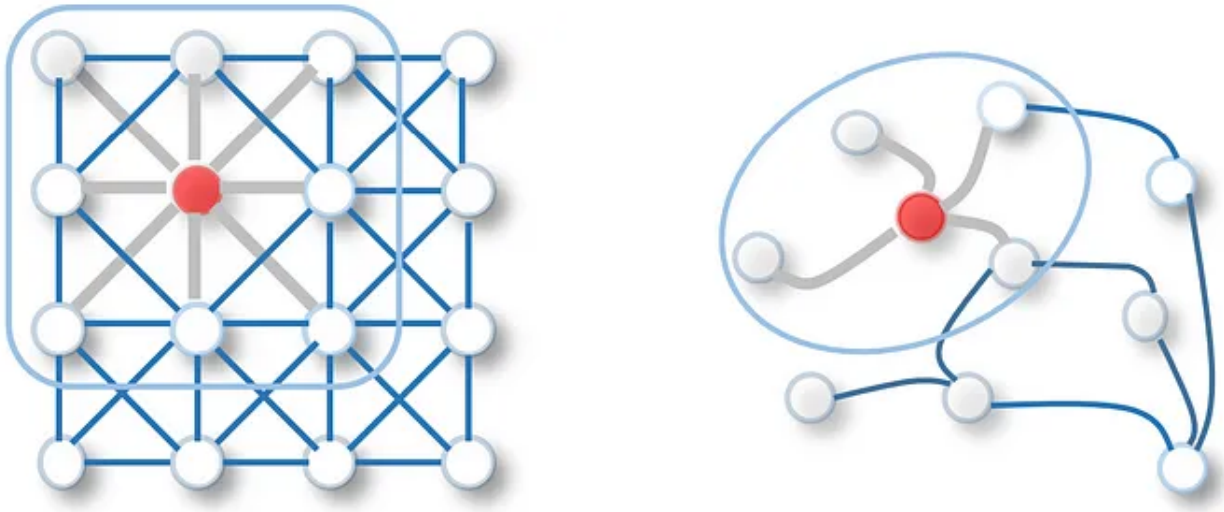
In GCN (Graph Convolutional Network), the input to the NN will be a graph. Also, instead of inferring a single z, it infers the value z_i for each node i in the graph. And to make predictions for Z_i, GCN utilizes both X_i and its neighboring nodes in the calculation.



Let's detail it a little more.

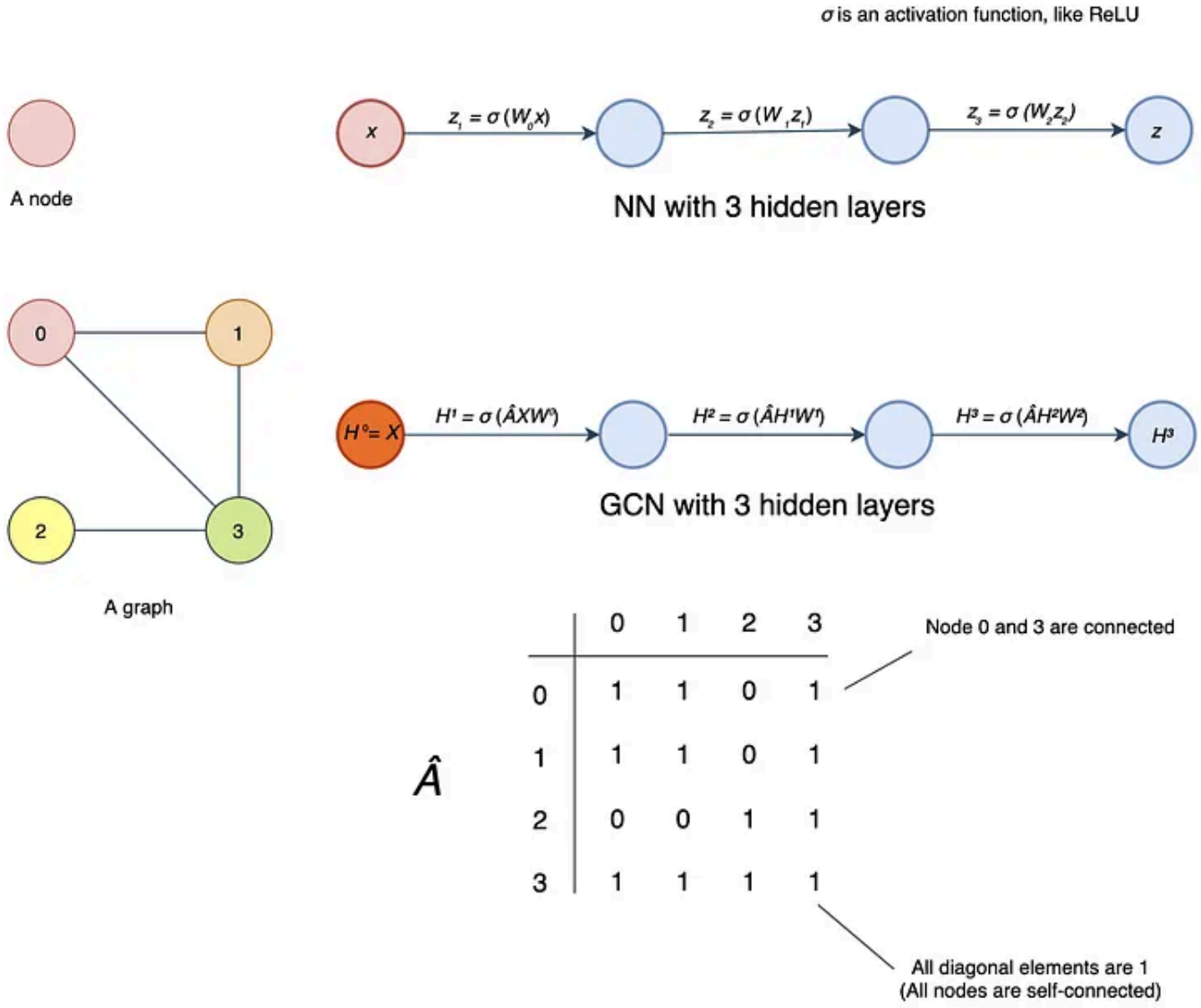
Graph Convolutional Networks (GCN)

The general idea of GCN is to apply convolution over a graph. Instead of having a 2-D array as input, GCN takes a graph as an input.



Source

The first diagram (the first row) below is the NN as we know and the second diagram is the GCN with a graph containing four nodes as the input.



In the first NN, it contains multiple dense layers (fully connected layers). x is the input for the first layer and z_i is the output of layer i. For each layer, we multiple z (or x for the first layer) with the weight matrix W and pass the output to an activation function σ, say ReLU. GCN is very similar, but the input to σ is $\hat{A}H^iW^i$ instead of W_iz_i. i.e. $\sigma(W_iz_i)$ v.s. $\sigma(\hat{A}H^iW^i)$ where z_i *and* H^i are the output vectors from the last hidden layer for NN and GCN respectively. But please note that W^i and W_i are different and have different dimensions. And for the first layer in GCN, X contains an array of nodes instead of a single node x. X will be encoded as a matrix with each row contains the features of a node.

So what is $\hat{A}$? GCN introduces an **adjacency matrix** A. The element A_{ij} in A equals 1 if node i and j are connected. Otherwise, it will be zero. So $\hat{A}$ indicates the neighbors of a node. But we will make one more adjustment to indicate all nodes are self-connected. This indicates the output of a node in a hidden layer depends on itself and its neighbors. So, we convert all diagonal elements of A to 1 to form $\hat{A}$. Mathematically, $\hat{A}$ equals A + I.

$\hat{A}$

	0	1	2	3
0	1	1	0	1
1	1	1	0	1
2	0	0	1	1
3	1	1	1	1

Node 0 and 3 are connected

All diagonal elements are 1
(All nodes are self-connected)

That comes to the output of the hidden layer to be $\sigma(\hat{A}H^iW^i)$. If we ignore W for a second, for each node in a hidden layer, $\hat{A}H^i$ sums up features on each node with its neighbors.

However, we may face the diminishing or exploding problem in a NN if we don't have certain control over the range of the hidden layer output. In specific, GCN wants $\hat{A}$ to be normalized to maintain the scale of the output feature vectors. One possibility is to multiple $\hat{A}$ with $\hat{D}^{-1}$ where $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$ in measuring the degree of each node. At a high level, instead of summing up itself with its neighbor, multiplying the sum with the inverse $\hat{D}^{-1}$ sort of averages them. Specifically, $\hat{D}$ is a diagonal matrix with each diagonal element $\hat{D}_{ii}$ counts the number of edges for the corresponding node i. And the output for each hidden layer becomes $\sigma(\hat{D}^{-1}\hat{A}H^iW^i)$, instead of $\sigma(\hat{A}H^iW^i)$.

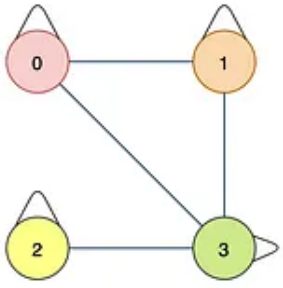
Let's calculate $\hat{D}$ in our example. For an undirected graph, the degree of a node is counted as the number of times an edge terminates at that node. So a self-loop will count twice. In our example, node 0 has 2 edges connecting to its neighbors plus a self-loop. Its degree equals 4 (i.e. 2 + 2). For node 3, its degree equals 5 (3 + 2).

$\hat{A}$

	0	1	2	3
0	1	1	0	1
1	1	1	0	1
2	0	0	1	1
3	1	1	1	1

$\hat{D}$

	0	1	2	3
0	4	0	0	0
1	0	4	0	0
2	0	0	3	0
3	0	0	0	5

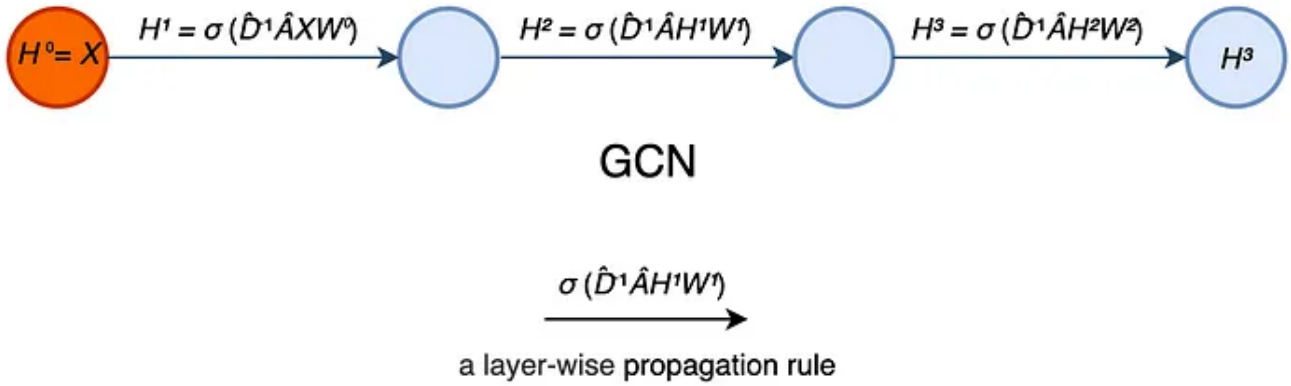


And $\hat{D}^{-1}$ equals

$\hat{D}^{-1}$

	0	1	2	3
0	1/4	0	0	0
1	0	1/4	0	0
2	0	0	1/3	0
3	0	0	0	1/5

The diagram below summarizes the model discussed so far. In this example, it has 3 hidden layers and for each hidden layer, it computes its output as $\sigma(\hat{D}^{-1}\hat{A}H^{i}W^{i})$. The equation used to compute a hidden layer output from the last layer output is called the **propagation rule**.



Besides using $\sigma(\hat{D}^{-1}\hat{A}H^{i}W^{i})$, there are other choices. Indeed, the propagation rule can be generalized as:

$$H^{(l+1)} = f(H^{(l)}, A)$$

Different choices of f result in different variants of the models. As a preview, the GCN paper applies the propagation rule below

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

where

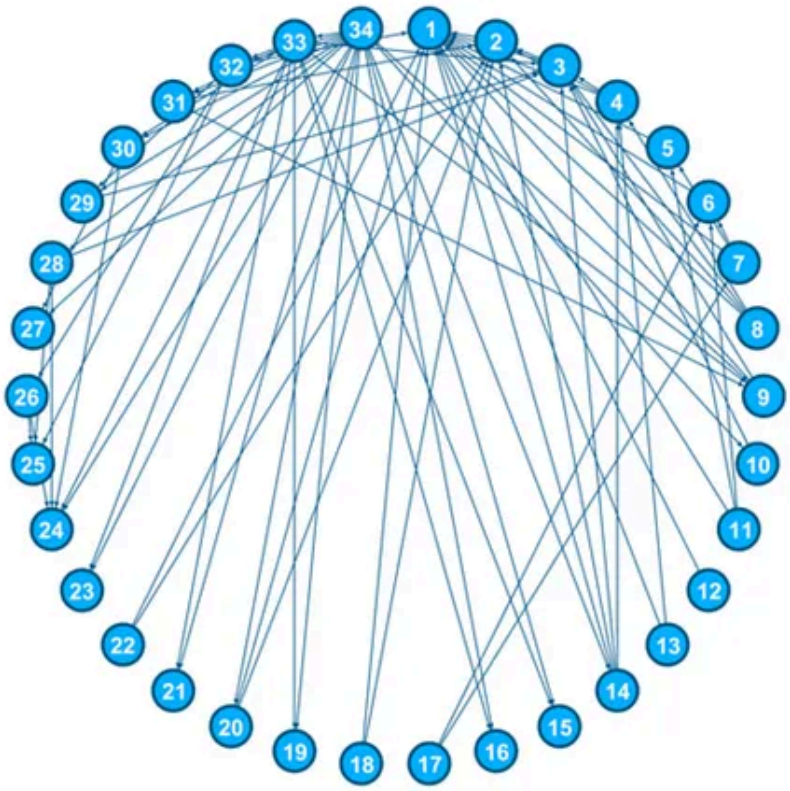
$$\tilde{A} = A + I_N$$

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$

$\hat{A}$ and $\hat{D}^{-1}$ are calculated in the same way as before. But let's defer the discussion later and work on an example instead. Let's demonstrate it with Zachary's karate club network problem.

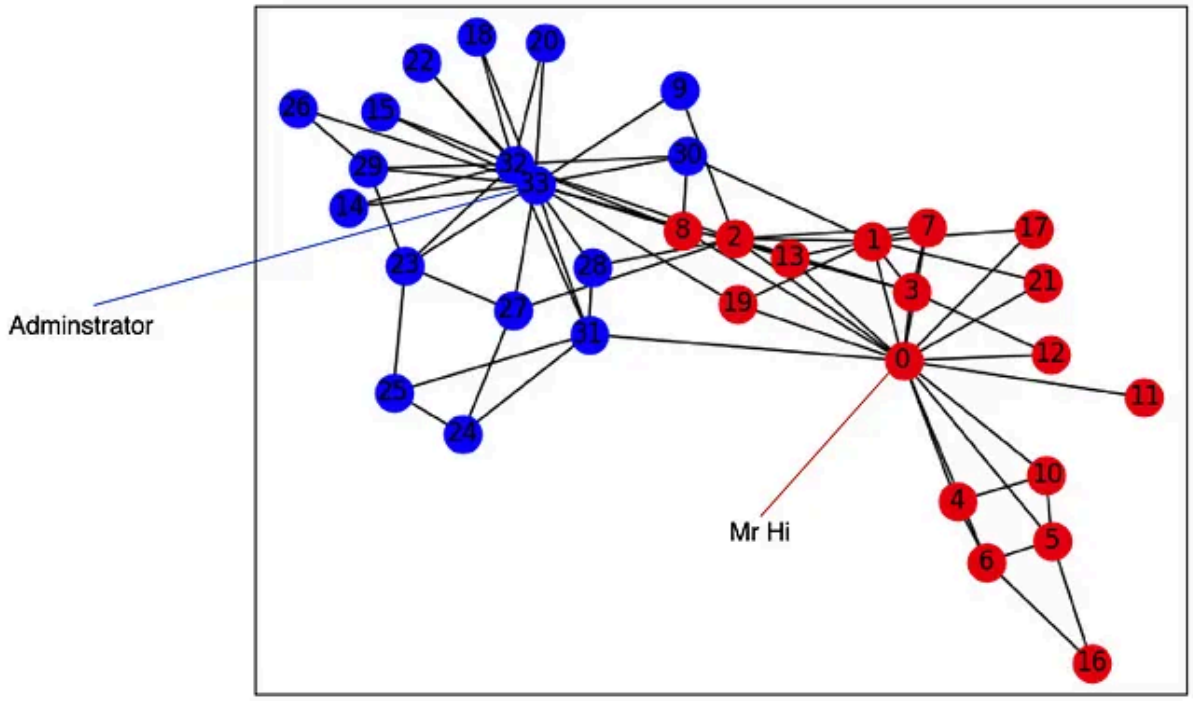
Zachary's karate club

There is a karate club that has two major stakeholders: the instructor (Mr. Hi) and the administrator. Unfortunately, the dispute between them causes it to split into 2 clubs. The original members will need to choose a side and pick which one to join. Their decisions will be based on how well they are connected with Mr. Hi or the administrator. This also includes how well they are connected to members that are associated with them. The diagram below is the social graph in representation connections between members.



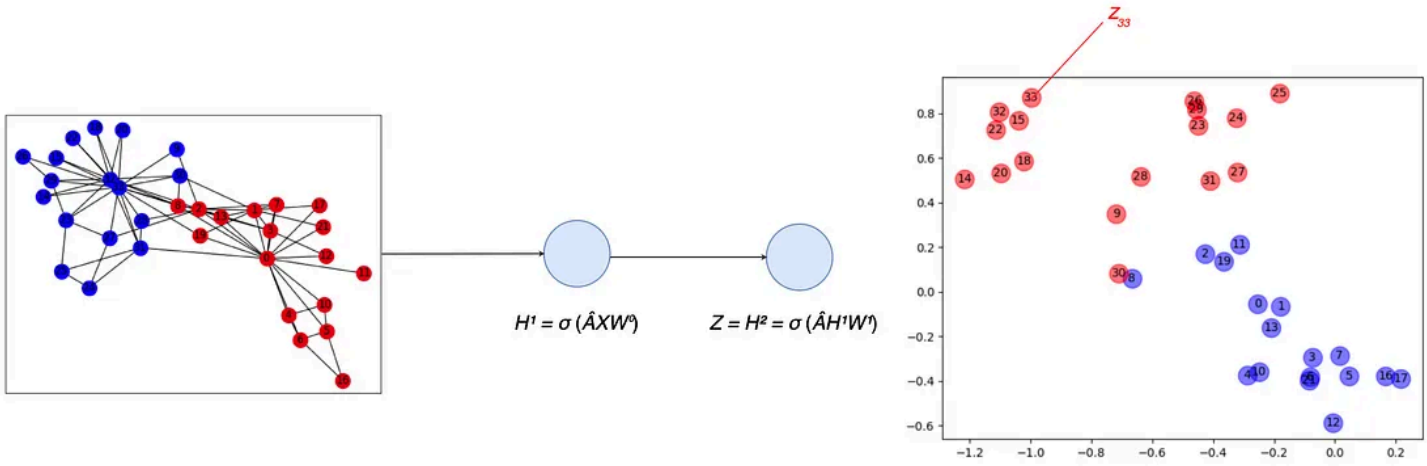
Source: Wikipedia (The member # start from 1)

Let's organize the graph a little bit more with blue and red being the ground truth of which club a member will join (Mr. Hi's club is marked with red color below).

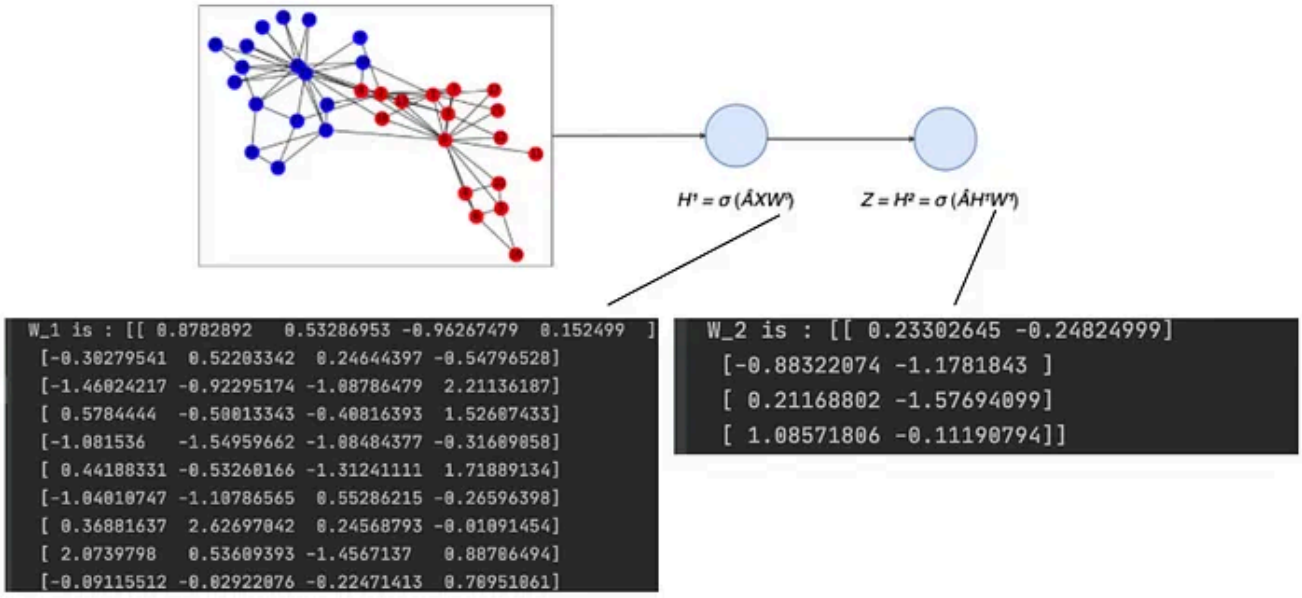


The member # start from 0

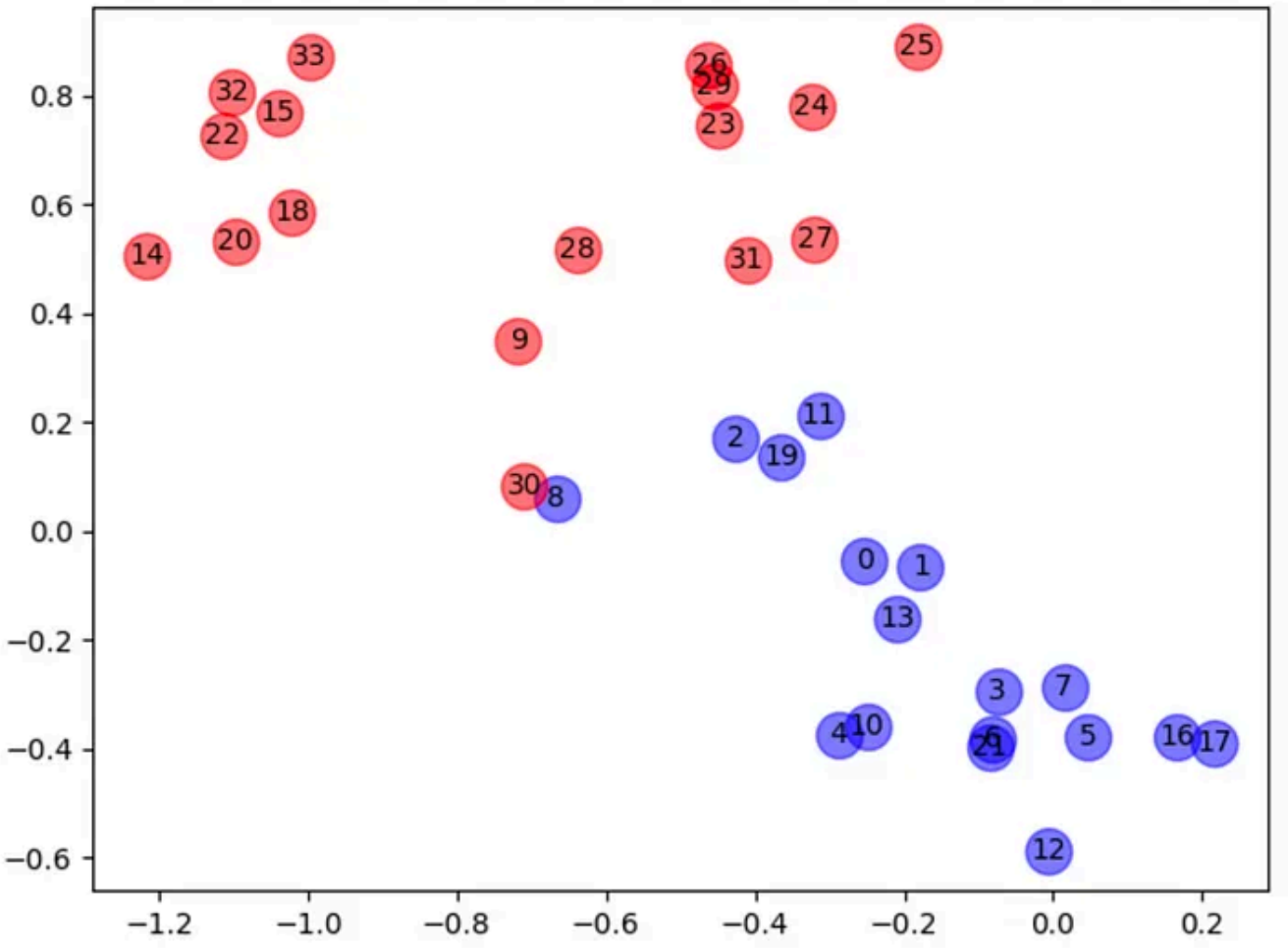
To find the club membership, we apply two hidden layers below with $H^{i+1} = \sigma(\hat{D}^{-1}\hat{A}H^{i}W^{i})$. The input will be the social graph above (without the club labels) and the last hidden layer will be the output.



The last hidden layer outputs 2 scalar latent features per node and we use these values to represent a node. The nice part is, in this example, we can compute the latent representation based on the social graph alone even without training. We run the model for one iteration with W^0 and W^1 initialized randomly with a normal distribution. But we are not going to perform gradient descent to update W. For X, we simply use an identity matrix as there is no node feature (as we have no prior knowledge of these features).



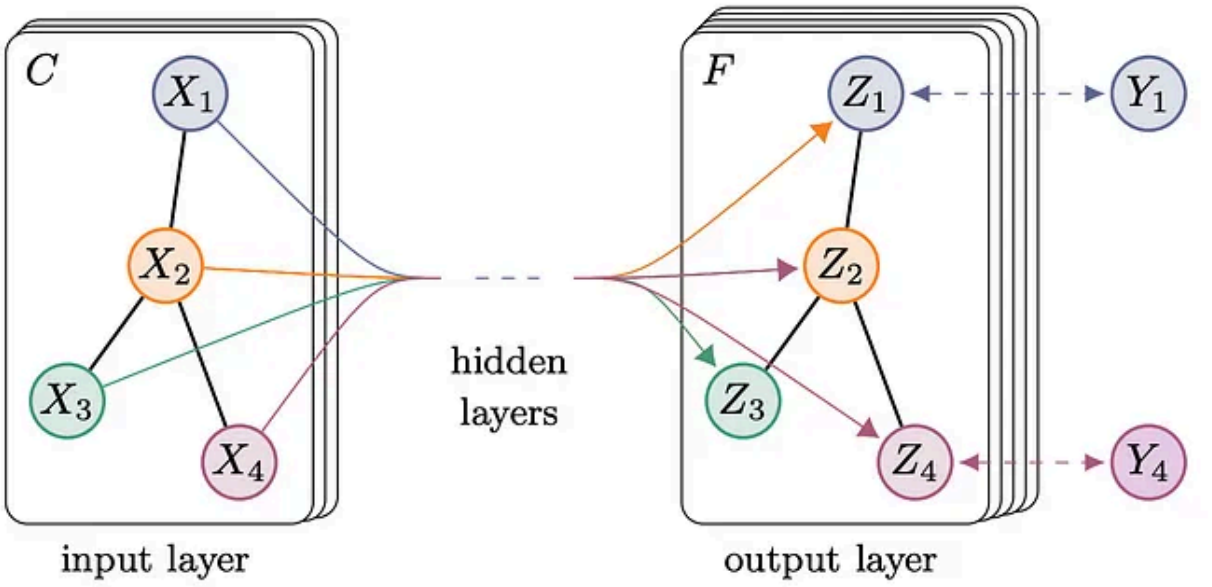
In the end, we plot out each node using the output latent features as the x and y coordinates.



This model is using $\sigma(\hat{D}^{-1}\hat{A}H^{i}W^{i})$ as the propagation rule

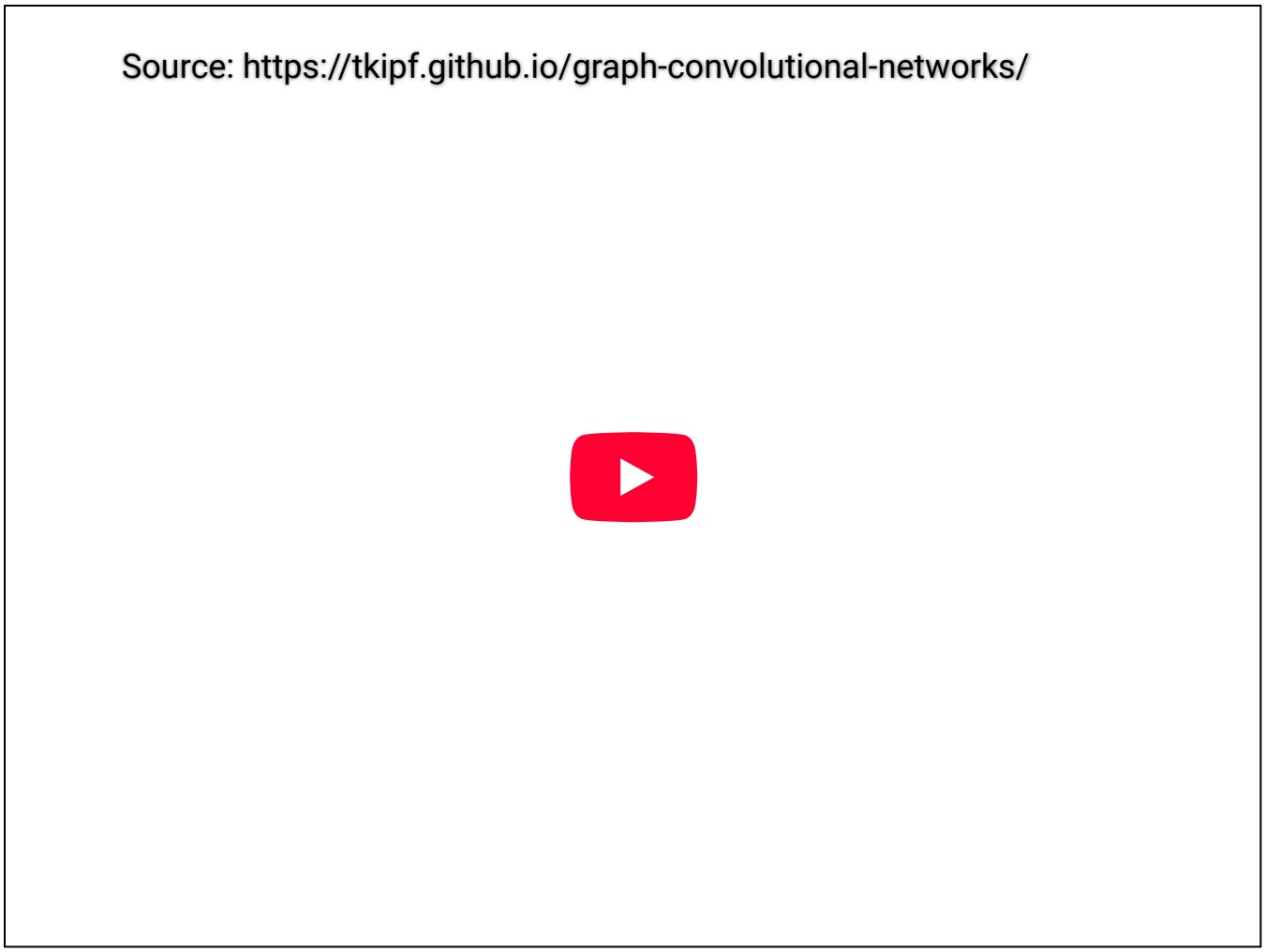
As shown, when we color each node with its ground truth (blue for Mr. Hi club and red for the administrator), the latent features generated are highly correlated with which club a person belongs to. In short, we just apply clustering on the input social graph. Intuitively, our example produces latent features similar to their neighbors as the hidden layer is averaging the latent features with its neighbors. The algorithm manages to cluster neighbors together by generating latent factors related to their neighbors.

Let's consider a semi-supervised problem for classification or clustering in which only one data point is labeled for each class or cluster. Once the latent features for each node are calculated, we can compute the distance between non-labeled data and the labeled data. Then, we find the nearest neighbor to a known labeled data to classify or cluster the unlabeled data.



Source

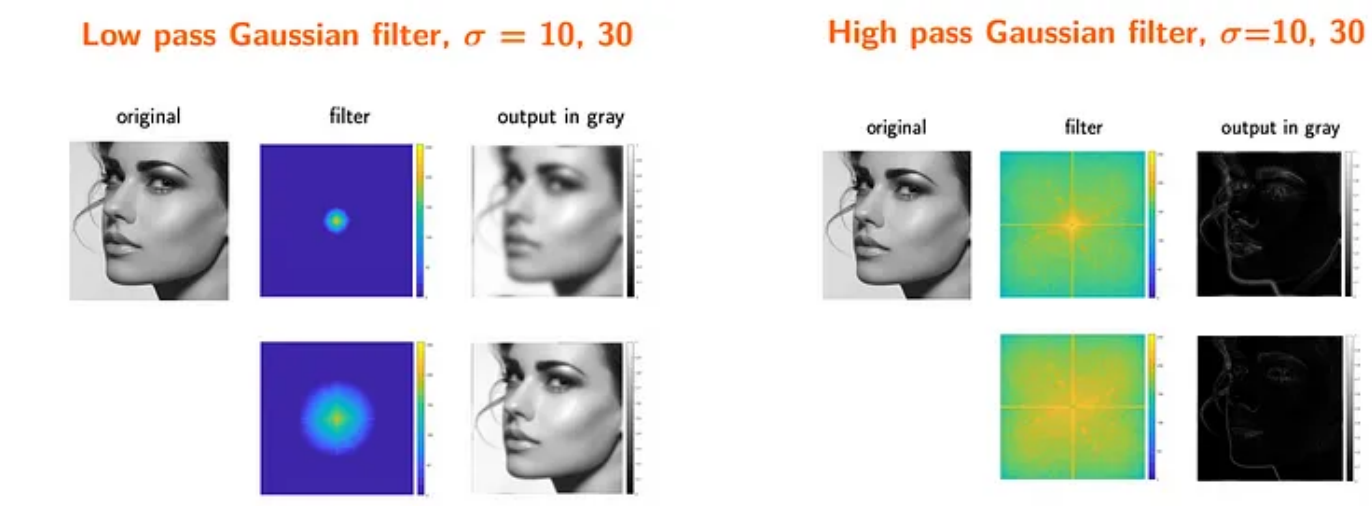
All propagation rules discussed so far are differentiable. For semi-supervised or unsupervised problems, we can use backpropagation to train the weights using the labeled data if needed. (For our last example, even without the extra training, the result is good.) As a demonstration, the GCN model below is trained for 300 iterations. In this model, only a single label per class is provided. As the training progress, we see how nodes belonging to the same ground truth classes start clustering together.



Source

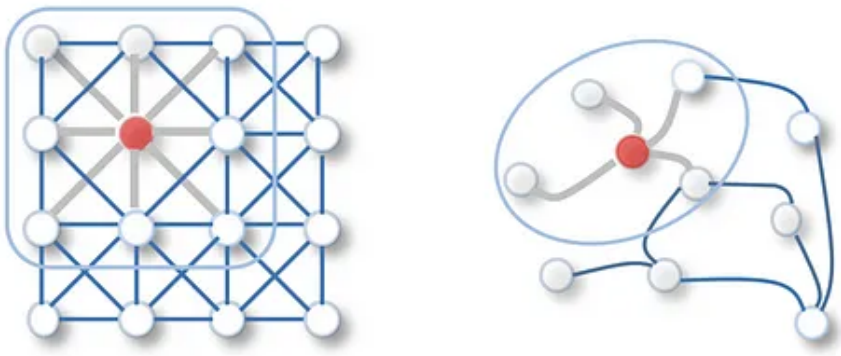
Spectral Graph Convolution

Engineering problems can be solved in the spatial domain or the spectral domain (a.k.a. frequency domain). For example, in signal processing, we apply a Fourier transform to convert an audio input from the temporal domain to the frequency domain and apply a low-pass filter to remove the high-frequency noise. In convolution, these filters can be defined with a spatial or a spectral approach. Which domain to use for the specific step depends on how easy to design the filter and how easy to perform the operation.



Source

In deep learning, researchers also approach the convolution from either the spatial or spectral-domain. For example, in the previous section, we apply a spatial graph convolution in the form of $\sigma(\hat{D}^{-1}\hat{A}H^{i}W^{i})$.



Source

But GCN is actually a spectral graph convolution. It is a localized first-order approximation of spectral graph convolutions with the propagation rule below.

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

where

$$\tilde{A} = A + I_N$$

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$

The journey leading to this new propagation rule can be very long. But it is not very important to understand it thoroughly to use it. In particular, the new rule is quite similar to the one before. We just add more mathematical justification. Feel free to browse through the next two sections quickly.

Spectral Graph Convolution

However, it will need some mathematical concepts, equations, and a few research papers to explain a spectral graph convolution. And because of its complexity, we will show the high-level steps only.

Convolutional filters are translation-invariant to allow weight sharing (the same filter slides over an input). When working in the spatial domain, it recognizes identical features independently of their spatial locations. A graph does not have a clear spatial concept or a mathematical definition of spatial translation. This leads to questions on what is the mathematical foundations that spatial graph filters are based on.

On the other hand, the spectral graph convolution is based on spectral graph theory. It provides a mathematical framework to design operators (filters) with the translation-invariant property. This does not imply the spatial graph filters are non-scientific. But sometimes, they may fall back to empirical results for justifications.

At a high level, a spectral graph convolution is defined in the Fourier domain by applying the filter gθ on the input signal x.

$$\mathbf{g}_\theta \star \mathbf{x}$$

$\mathbf{g}_\theta$ =diag(θ) parameterized by $\theta \in \mathbb{R}^N$

A diagonal matrix

Previously, we model a graph with the adjacency matrix A and normalize it with the diagonal node degree matrix D. Under the spectral graph theory, an undirected graph is represented by the normalized **graph Laplacian** matrix L which is defined as

$$\text{normalized graph Laplacian } L = I_N - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} = U\Lambda U^\top$$

L is a real symmetrical positive semidefinite matrix. But for an $N \times N$ matrix, if it is positive semidefinite, it has N orthogonal eigenvectors. (We will skip the proof here.) In short, these N independent vectors form a basis that can diagonalize a matrix, i.e., L can be factored as $L = U\Lambda U^T$ like the one below where U contains the eigenvectors of L and Λ is a diagonal matrix containing the corresponding eigenvalues. These eigenvectors are also known as the graph Fourier modes, and the corresponded eigenvalues are the frequencies of the graph.

$$S = Q\Lambda Q^T$$

$$S = \begin{pmatrix} \uparrow & \uparrow & & \uparrow \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \\ \downarrow & \downarrow & & \downarrow \end{pmatrix} \begin{pmatrix} \lambda_1 & & & 0 \\ & \lambda_2 & & \\ & & \ddots & \\ 0 & & & \lambda_n \end{pmatrix} \begin{pmatrix} \longleftarrow \mathbf{v}_1^T \longrightarrow \\ \longleftarrow \mathbf{v}_2^T \longrightarrow \\ \vdots \\ \longleftarrow \mathbf{v}_n^T \longrightarrow \end{pmatrix}$$

$$Q \qquad \Lambda \qquad Q^T$$

The Laplacian is diagonalized by the Fourier basis U and the spectral convolution will be defined as a linear operator with U. In specific, the graph Fourier transform on x is defined as

The graph Fourier transform to x and its inverse is defined as:

$$\mathscr{F}(\mathbf{x}) = \mathbf{U}^T\mathbf{x} \qquad \mathscr{F}^{-1}(\hat{\mathbf{x}}) = \mathbf{U}\hat{\mathbf{x}}$$

Here, the graph convolution operator is defined in the Fourier domain. It is the multiplication of the graph x with a filter in the Fourier space. i.e.

Fourier space

element-wise product on a matrix.

$$\begin{aligned} \mathbf{x} *_G \mathbf{g} &= \mathscr{F}^{-1}(\mathscr{F}(\mathbf{x}) \odot \mathscr{F}(\mathbf{g})) \\ &= \mathbf{U}(\mathbf{U}^T\mathbf{x} \odot \mathbf{U}^T\mathbf{g}), \end{aligned}$$

$x *G g$ means performing spectral convolution on x with filter g.

If we denote a filter as

$$\mathbf{g}_\theta = diag(\mathbf{U}^T\mathbf{g}) \qquad \mathbf{g} \in \mathbf{R}^n$$

Then the graph convolution can be simplified as:

$$\mathbf{x} *_G \mathbf{g}_\theta = \mathbf{U}\mathbf{g}_\theta\mathbf{U}^T\mathbf{x}.$$

Spectral Convolutional Neural Network assumes the filter gθ is a diagonal matrix filled with learnable parameters Θ^k_{ij}. Θ^k_{ij} contains parameters for layer *k* that maps the *i*th input feature to the *j*th output feature. For layer *k*, the output of a hidden layer output is

f_{k-1} and f_k are the number of input channels and output channels for layer k respectively.

filter $\mathbf{g}_\theta = \mathbf{\Theta}^{(k)}_{i,j}$

for input channel *i* to output channel *j*

$$\mathbf{H}^{(k)}_{:,j} = \sigma\Big(\sum_{i=1}^{f_{k-1}} \mathbf{U}\mathbf{\Theta}^{(k)}_{i,j}\mathbf{U}^T\mathbf{H}^{(k-1)}_{:,i}\Big) \quad (j = 1, 2, \cdots, f_k)$$

for output channel *j* in layer *k*

This is the mathematical framework for a spectral graph convolution.

The shortfall of Spectral Graph Convolution

While the spectral graph theory has operators with well-defined translation properties, computing the eigenvectors of *L* is computation expensive. We want to avoid that. Another shortfall is that these filters are not localized in general. We may not visit the *k*th closest neighbors for each node only. Its complexity grows with the size of the graph and not scalable. Indeed, to compute the output of a node efficiently, we should just hop over a limited number of neighbors, say at most *K* hops from the central node. This leads to researches of applying approximation to the spectral graph convolution, like those in ChebNet and GCN, such that filters are localized. And because of this issue, spatial graph convolution also receives more attention in recent years.

ChebNet

So how can we find a localized filter in the spectral graph convolution? Chebyshev Spectral CNN (ChebNet) approximates the filter gθ using a truncated expansion in

terms of Chebyshev polynomials $Tk(x)$ up to Kth order.

Chebyshev coefficients — max. eigenvalues

$$\mathbf{g}_\theta \approx \sum_{i=0}^{K} \theta_i T_i(\tilde{\mathbf{\Lambda}}), \text{ where } \tilde{\mathbf{\Lambda}} = 2\mathbf{\Lambda}/\lambda_{max} - \mathbf{I_n}$$

lie in [-1. 1]

$$T_i(\mathbf{x}) = 2\mathbf{x}T_{i-1}(\mathbf{x}) - T_{i-2}(\mathbf{x}) \text{ with } T_0(\mathbf{x}) = 1 \text{ and } T_1(\mathbf{x}) = \mathbf{x}.$$

The convolution of a graph x with the defined filter gθ becomes:

$$\mathbf{x} *_G \mathbf{g}_\theta \approx \mathbf{U}(\sum_{i=0}^{K} \theta_i T_i(\tilde{\mathbf{\Lambda}}))\mathbf{U}^T\mathbf{x},$$

This is K-localized as it is a Kth-order polynomial in the Laplacian. It depends only on nodes that are at maximum K steps away from the central node (details). It means it is localized in space and scales well regardless of the graph size. Even ChebNet and GCN start as a Spectral Graph Convolution, they apply approximation such that their filters are localized.

As $T_i(\hat{L})$ can be written as:

$$\text{As } T_i(\tilde{\mathbf{L}}) = \mathbf{U}T_i(\tilde{\mathbf{\Lambda}})\mathbf{U}^T,$$
$$\text{where } \tilde{\mathbf{L}} = 2\mathbf{L}/\lambda_{max} - \mathbf{I_n}$$

The convolution becomes

$$\mathbf{x} *_G \mathbf{g}_\theta \approx \sum_{i=0}^{K} \theta_i T_i(\tilde{\mathbf{L}})\mathbf{x},$$

Hence, we don't need to calculate the eigenvectors and it is localized.

GCN

GCN introduces a first-order approximation of ChebNet by assuming $K = 1$ and $\lambda max = 2$. The equation becomes

$$\mathbf{x} *_G \mathbf{g}_\theta \approx \theta_0 \mathbf{x} - \theta_1 \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} \mathbf{x}$$

To reduce the number of free parameters and to avoid over-fitting, GCN assumes $\theta = \theta_0 = -\theta_1$, and the equation becomes

$$\mathbf{x} *_G \mathbf{g}_\theta \approx \theta(\mathbf{I_n} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}})\mathbf{x}$$

But the L.H.S. term below has eigenvalues in the range [0, 2], to avoid exploding/vanishing problems, a renormalization trick is applied that results in the R.H.S. term below.

$$\underbrace{I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}}_{\text{has eigenvalues in the range [0, 2]}} \rightarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$$

$$\tilde{A} = A + I_N \text{ and } \tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$

Finally, the propagation rule for GCN is

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

where $\Theta \in \mathbb{R}^{C \times F}$ is now a matrix of filter parameters and $Z \in \mathbb{R}^{N \times F}$ is the convolved signal matrix.

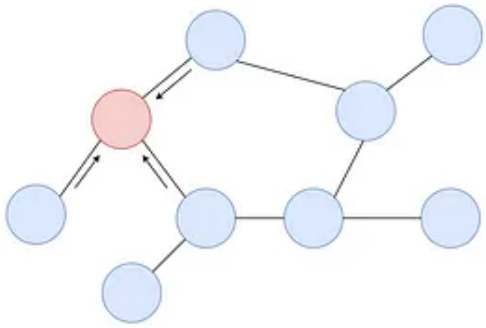
$X \in \mathbb{R}^{N \times C}$ with C input channels — Number of output channels

(Equation credits in this section: Source 1 and 2.)

Next, we will move onto Spatial Graph convolution.

Spatial Graph Convolution

Different spatial graph convolutions depend on different aggregators to gather information from each node's neighbors. Conceptually, we can also view it as message passing.



Without the approximations in Spectral Graph Convolutions, Spatial Graph Convolutions are usually more scalable since their filters are localized. The major challenge is defining a local invariance of CNNs that work with central nodes that have different numbers of neighbors. In the next few sections, we will overview different Spatial Graph convolution methods quickly. We may present equations that it may take sometimes for you to connect the dots. But not to lengthen the article further, please refer to individual papers for details.

Message Passing Neural Network (MPNN)

MPNN outlines a general message passing framework for Spatial Graph Convolution. It passes information (message) from one node to another along edges and repeats it *K*-step to let information propagate through the graph. The equation below is the hidden feature representation for the *k*th layer for node *v*. It depends on the hidden feature of *v* and its neighbors in the previous layer as well as the edge features with its neighbors. Different choices of *U* and *M* functions will lead to different variants of the model.

neighbors of v — the edge feature between u and v

$$\mathbf{h}_v^{(k)} = U_k(\mathbf{h}_v^{(k-1)}, \sum_{u \in N(v)} M_k(\mathbf{h}_v^{(k-1)}, \mathbf{h}_u^{(k-1)}, \mathbf{x}_{vu}^e))$$

where $\mathbf{h}_v^{(0)} = \mathbf{x}_v$, $U_k(\cdot)$ and $M_k(\cdot)$ are functions with learnable parameters.

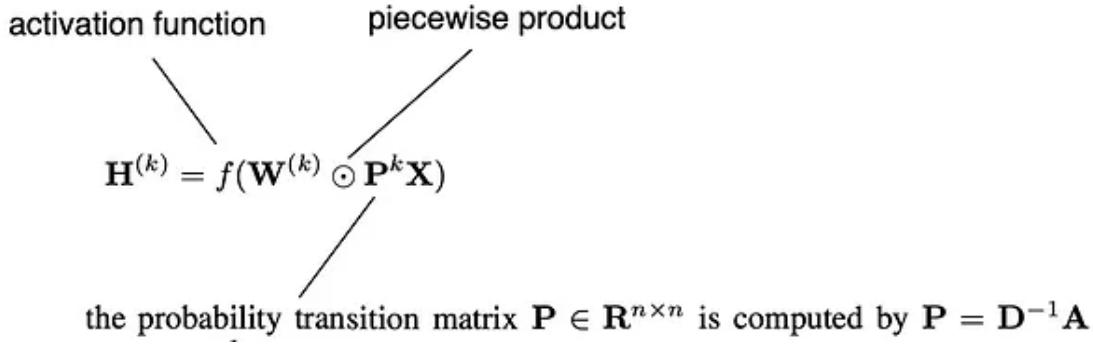
The latent representation of a node in the last hidden layer can be passed to an output layer to perform node-level predictions. Or it can pass to a readout function *R* with learnable parameters to perform graph-level predictions.

$$\mathbf{h}_G = R(\mathbf{h}_v^{(K)} | v \in G)$$

For example, in drug discovery, a graph represents a chemical compound with atoms as nodes and chemical bonds as edges. We may want to classify whether this compound can hinder the growth of cancer cells or it is carcinogenic. Therefore, we can perform a readout with the equation above in making a graph-level prediction.

Diffusion Convolutional Neural Network (DCNN)

DCNN treats graph convolutions as a diffusion process. It transfers information from one node to one of its neighbors with a probability transition matrix *P* that equals $D^{-1}A$. This process will repeat *K* times so that information distribution will reach an equilibrium. And the hidden representation H^k at the *k*th step will be calculated from P^k, but not on the last hidden representation.



DCNN concatenates $H^{(1)}$, $H^{(2)}$, $\cdots$, $H^{(K)}$ together to form the final model outputs.

Graph Isomorphism Network (GIN)

However, GIN claims that the graph embeddings in MPNN methods fail to distinguish different graph structures. A good scheme should distinguish different graph structures and map them to different latent features in the embedding space. To fix that, GIN introduces a learnable parameter ε^k to adjust the weight of the central node.

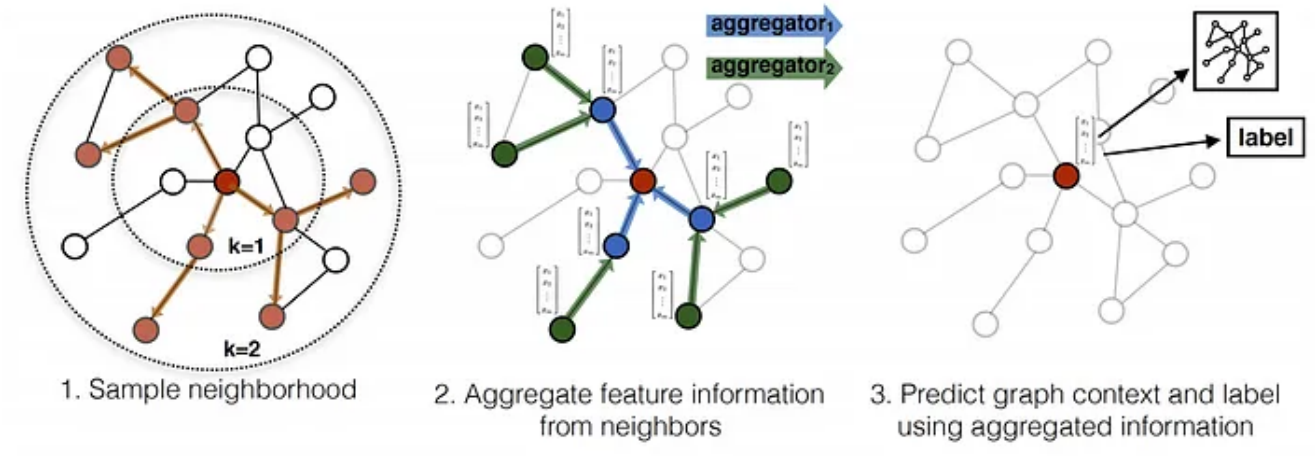
multi-layer perceptron

$$\mathbf{h}_v^{(k)} = MLP((1+\epsilon^{(k)})\mathbf{h}_v^{(k-1)} + \sum_{u \in N(v)} \mathbf{h}_u^{(k-1)})$$

learnable parameters

GraphSage

Social influencers have many connections. For a graph with nodes that have many edges, convolutions may not scale well. GraphSage uses sampling to obtain a fixed number of neighbors for each node for message passing instead of all neighboring nodes.



Source

Here is the formularization:

like mean, sum or max function

$$\mathbf{h}_v^{(k)} = \sigma(\mathbf{W}^{(k)} \cdot f_k(\mathbf{h}_v^{(k-1)}, \{\mathbf{h}_u^{(k-1)}, \forall u \in S_{\mathcal{N}(v)}\}))$$

where $\mathbf{h}_v^{(0)} = \mathbf{x}_v$, $f_k(\cdot)$ is an aggregation function, $S_{\mathcal{N}(v)}$ is a random sample of the node v's neighbors.

Alternatively, we can compute an aggregated value for the neighbors first. For example, we can use an LSTM aggregator to compute a representation of the

neighbors. Then we concatenate the values with the central node and apply a dense layer. Otherwise, we can apply W to all neighbors followed by an activation function and a max pool.

$$\mathbf{h}_{\mathcal{N}_v}^t = \text{AGGREGATE}_t\left(\{\mathbf{h}_u^{t-1}, \forall u \in \mathcal{N}_v\}\right)$$
$$\mathbf{h}_v^t = \sigma\left(\mathbf{W}^t \cdot [\mathbf{h}_v^{t-1} \| \mathbf{h}_{\mathcal{N}_v}^t]\right)$$

$$\mathbf{h}_{\mathcal{N}_v}^t = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_u^{t-1} + \mathbf{b}\right), \forall u \in \mathcal{N}_v\})$$

Source

Sampling

FastGCN refines the sampling algorithm further. Instead of sampling neighbors for each node, FastGCN uses importance sampling to reduce variance. It samples nodes (u) from a distribution q that reflects how well it is connected to other nodes. Then, it applies importance sampling to estimate the loss gradient of node v from u.

Algorithm 2 FastGCN batched training (one epoch), improved version

1: For each vertex u, compute sampling probability $q(u) \propto \|\hat{A}(:,u)\|^2$
2: **for** each batch **do**
3: For each layer l, sample t_l vertices $u_1^{(l)}, \ldots, u_{t_l}^{(l)}$ according to distribution q
4: **for** each layer l **do** ▷ Compute batch gradient ∇L_{batch}
5: If v is sampled in the next layer,

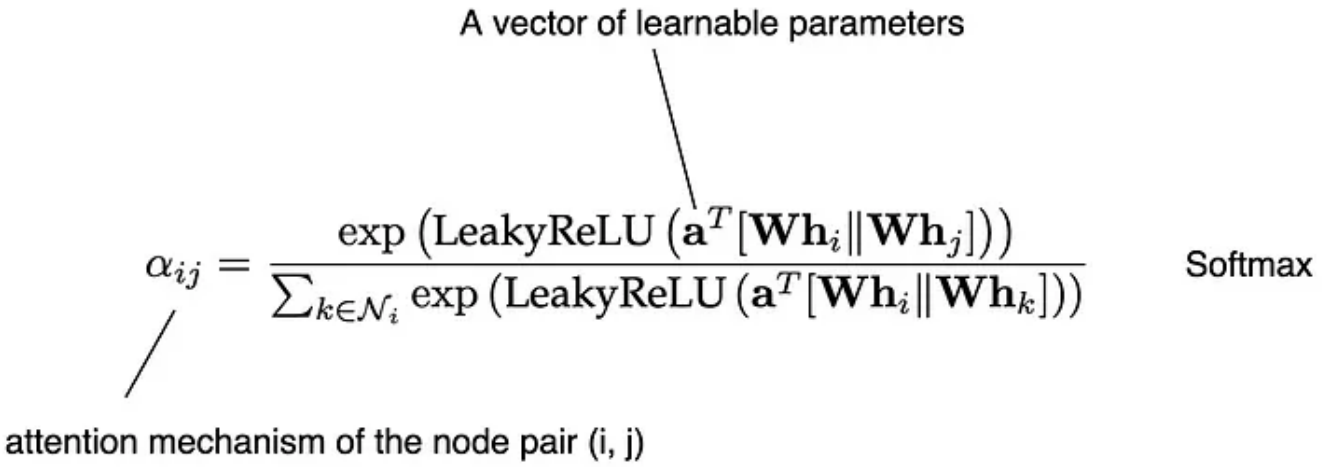
$$\nabla \tilde{H}^{(l+1)}(v,:) \leftarrow \frac{1}{t_l}\sum_{j=1}^{t_l}\frac{\hat{A}(v,u_j^{(l)})}{q(u_j^{(l)})}\nabla\left\{H^{(l)}(u_j^{(l)},:)W^{(l)}\right\}$$

6: **end for**
7: $W \leftarrow W - \eta\nabla L_{\text{batch}}$ ▷ SGD step
8: **end for**

Source

Graph Attention Network (GAT)

GAT uses attention to learn the relative weights between two connected nodes in message passing. First, GAT calculates the attention for the node pair i and j. The resulting vectors Wh_i (h_i is the hidden representation of node i) and Wh_j are concatenated (|| operator) and multiply with learnable vector a. Their attention, measuring the connective strength between them, is then computed with a softmax function.



The hidden representation for node i is then weighted by the computed attention α with the final result as

$$\mathbf{h}_i' = \sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}\mathbf{W}\mathbf{h}_j\right)$$

Mixture Model Network (MoNet)

MoNet uses node pseudo-coordinates to determine the relative position between a node and its neighbor. Consider node *y* is a neighbor of node *x*. First, a *d*-dimensional vector of pseudo-coordinates *u*(*x, y*) is calculated as below. The MoNet paper uses the degrees of the nodes (number of connection) as the pseudo-coordinates and then transforms them further with a fully-connected network.

$$\mathbf{u}(x,y) = \left(\frac{1}{\sqrt{\deg(x)}}, \frac{1}{\sqrt{\deg(y)}}\right)^\top$$

$$\tilde{\mathbf{u}}(x,y) = \tanh(\mathbf{A}\mathbf{u}(x,y) + \mathbf{b})$$

pseudo-coordinates

learned parameters

Modified from source

Once the relative position between two nodes is calculated, a weight function maps the relative position to the relative weight between these two nodes. This weighting function **w** is parametrized by learnable parameters Θ and composed of J components:

$$\mathbf{w}_{\Theta}(\mathbf{u}) = (w_1(\mathbf{u}), \ldots, w_J(\mathbf{u}))$$

In MoNet, the weight function is defined as:

$$w_j(\mathbf{u}) = \exp(-\tfrac{1}{2}(\mathbf{u}-\boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1}(\mathbf{u}-\boldsymbol{\mu}_j))$$

jth component

learnable parameters

(covariance matrix & mean vector of a Gaussian)

where μ_j and Σ_j are trainable parameters for a Gaussian.

Let's summarize it. The coordinate system (x, y) for an image is defined in Euclidean space with the output pixel value being $f(x, y)$. With MoNet, the weight function above allows us to expand the concept to non-Euclidean space and apply a shared filter that works across different locations. Below is the detailed equation for applying the convolution filter g on f.

$$(f \star g)(x) = \sum_{j=1}^{J} g_j \, D_j(x) f.$$

$$D_j(x) f = \sum_{y \in \mathcal{N}(x)} w_j(\mathbf{u}(x,y)) f(y), \quad j = 1, \ldots, J,$$

$$D_j(x) f_l = \sum_{y \in \mathcal{N}(x)} e^{-\frac{1}{2}(\tilde{\mathbf{u}}(x,y)-\boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1}(\tilde{\mathbf{u}}(x,y)-\boldsymbol{\mu}_j)} f_l(y)$$

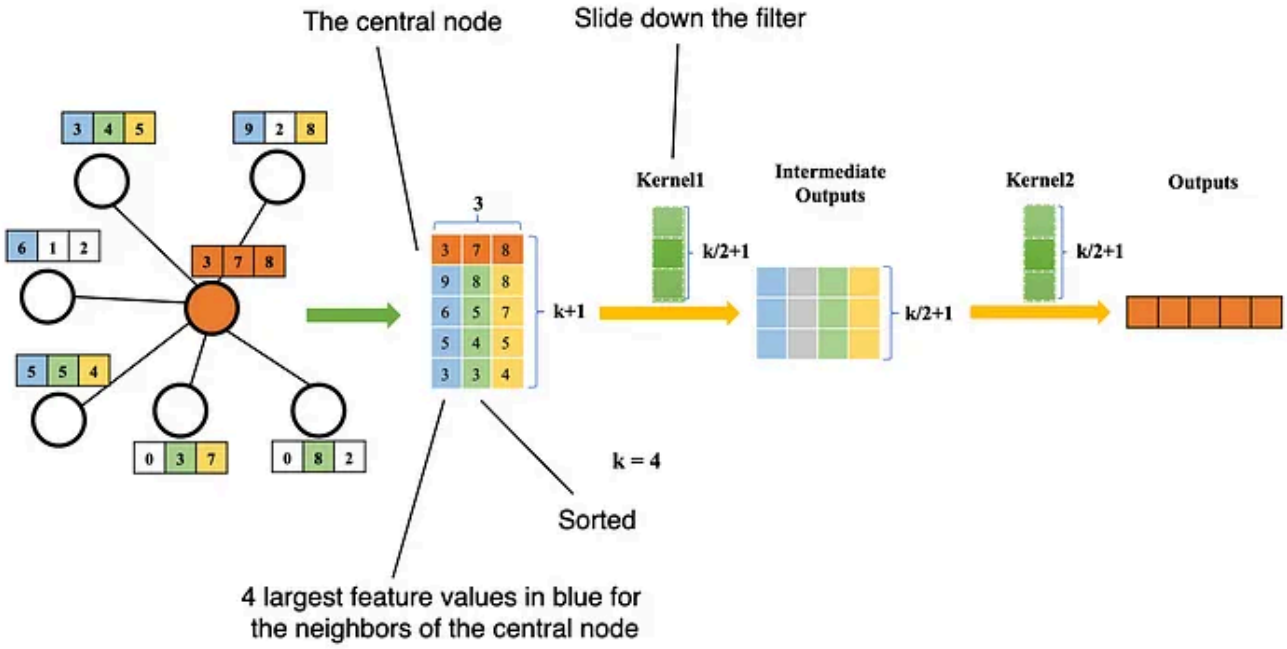
MoNet can be generalized with different choices of u and weight functions. The diagram below shows how different GNN can be viewed as MoNet using the specific u and w.

Method	Pseudo-coordinates	$\mathbf{u}(x,y)$	Weight function $w_j(\mathbf{u}), j = 1, \ldots, J$
CNN [28]	Local Euclidean	$\mathbf{x}(x,y) = \mathbf{x}(y) - \mathbf{x}(x)$	$\delta(\mathbf{u} - \bar{\mathbf{u}}_j)$
GCNN [32]	Local polar geodesic	$\rho(x,y), \theta(x,y)$	$\exp(-\frac{1}{2}(\mathbf{u}-\bar{\mathbf{u}}_j)^\top \begin{pmatrix} \sigma_\rho^2 & \\ & \bar{\sigma}_\theta^2 \end{pmatrix}^{-1} (\mathbf{u}-\bar{\mathbf{u}}_j))$
ACNN [7]	Local polar geodesic	$\rho(x,y), \theta(x,y)$	$\exp(-\frac{1}{2}\mathbf{u}^\top \mathbf{R}_{\bar{\theta}_j} \begin{pmatrix} \bar{\alpha} & \\ & 1 \end{pmatrix} \mathbf{R}_{\bar{\theta}_j}^\top \mathbf{u})$
GCN [26]	Vertex degree	$\deg(x), \deg(y)$	$\left(1 - \lvert 1 - \frac{1}{\sqrt{u_1}} \rvert\right)\left(1 - \lvert 1 - \frac{1}{\sqrt{u_2}} \rvert\right)$
DCNN [2]	Transition probability in r hops	$p^0(x,y), \ldots, p^{r-1}(x,y)$	$\mathrm{id}(u_j)$

Source

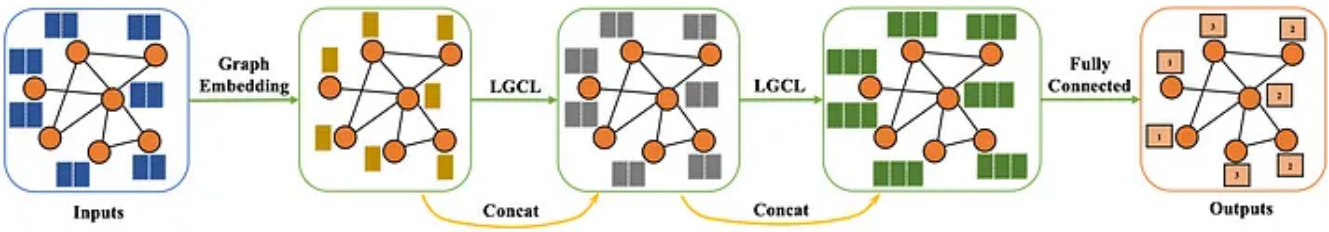
Large-Scale Learnable Graph Convolutional Networks (LGCN)

In each LGCL layer, it selects and sorts the largest *k* values for each input feature channel for the neighbors of the central node. With the features of the central node, it forms a (k+1) × 3 matrix. Then a CNN filter is applied. In the diagram below, two filters are applied to create a new node representation with 5 output channels.



Modified from source

LGCN applies multiple layers of LGCL as in the diagram below.



Source

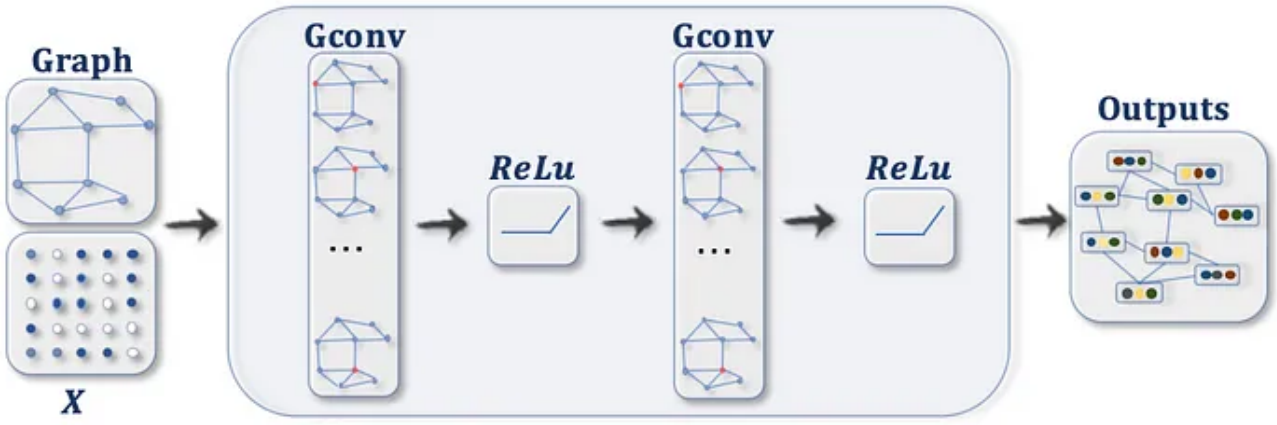
The input nodes are transformed into low-dimensional representations using a graph embedding layer. In some cases, it can simply use a linear transformation like the one below.

$$X_1 = X_0 W_0,$$

Then it follows with two LGCL layers with skip concatenation connections to produce the output, i.e. the inputs and outputs of LGCL are concatenated together. Finally, a fully-connected layer is used to classify each node.

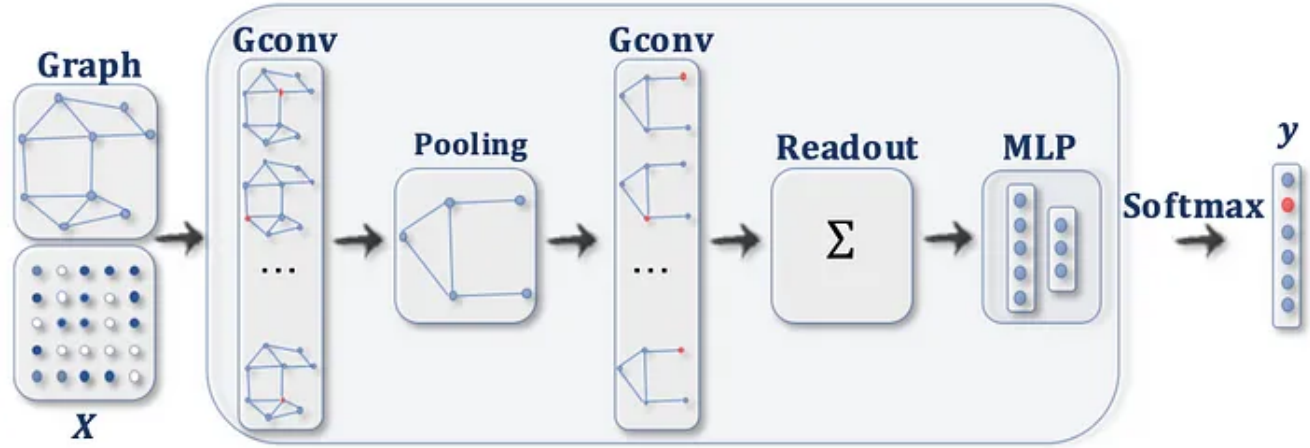
Pooling

In the diagram below, we apply layers of convolutions and ReLU to generate the latent representation for each node.



Source

But in other applications, we are interested in generating a representation of the graph. For example, consider a sample (G, y) where G is a graph, and y is its class. As a classification problem, we read G and predict y — the representation of the graph. In this context, pooling can be applied to generate a coarser graph which can be further reduced with a readout operation. This is similar to the conventional CNN where we apply max pool to reduce the spatial resolution.



Source

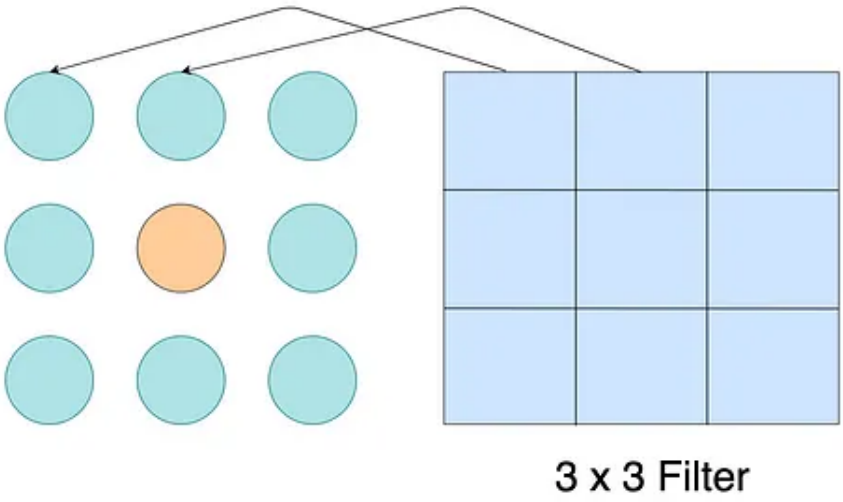
In general, the pooling and the readout are similar and mathematically, can be generalized as:

$$\mathbf{h}_G = mean/max/sum(\mathbf{h}_1^{(K)}, \mathbf{h}_2^{(K)}, ..., \mathbf{h}_n^{(K)})$$

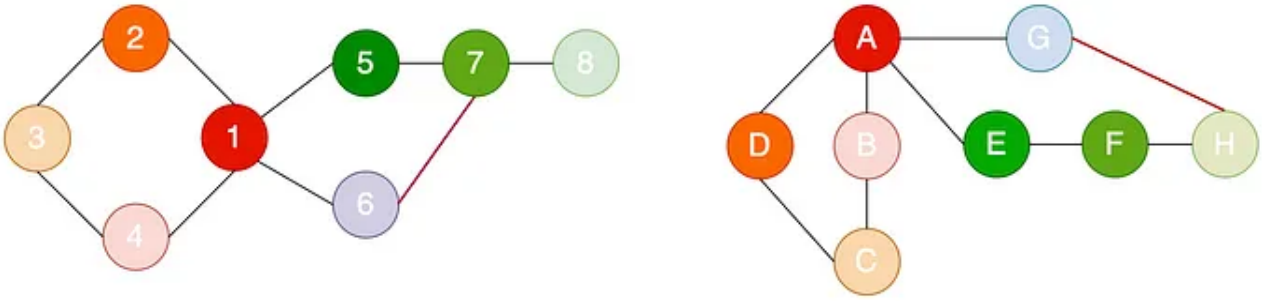
Source

DGCNN

If there is a consistent way to map nodes in a graph to a Euclidean space, there will be no ambiguity on which weight should be associated with a specific node when a filter is applied. So for a spatial graph convolution, the challenge is how to assign weights to nodes consistently.

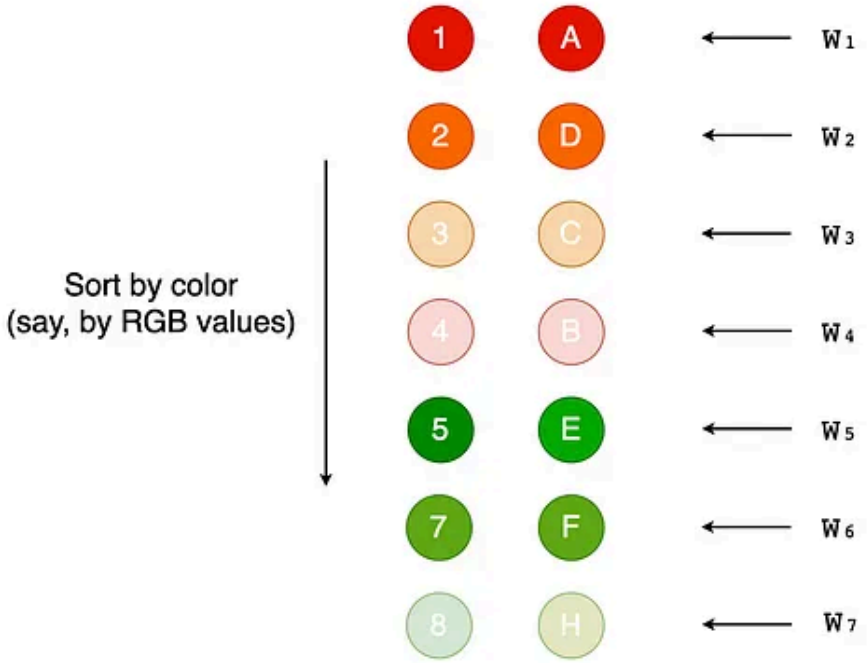


The two graphs below are similar in topology with the exception of the two red edges below. We color nodes in both graphs with similar colors if they have similar network topology. For example, both nodes D and 2 are colored as orange.

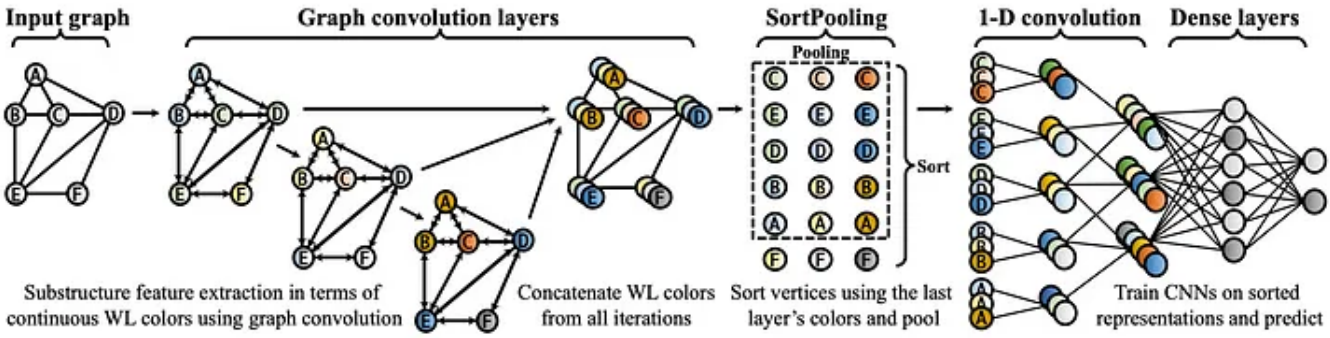


And when we apply a convolution filter, both graphs should generate similar latent features for the corresponding nodes. One way to do it is to sort nodes by the

assigned color (say, by RGB values) and assign the corresponding weights accordingly.

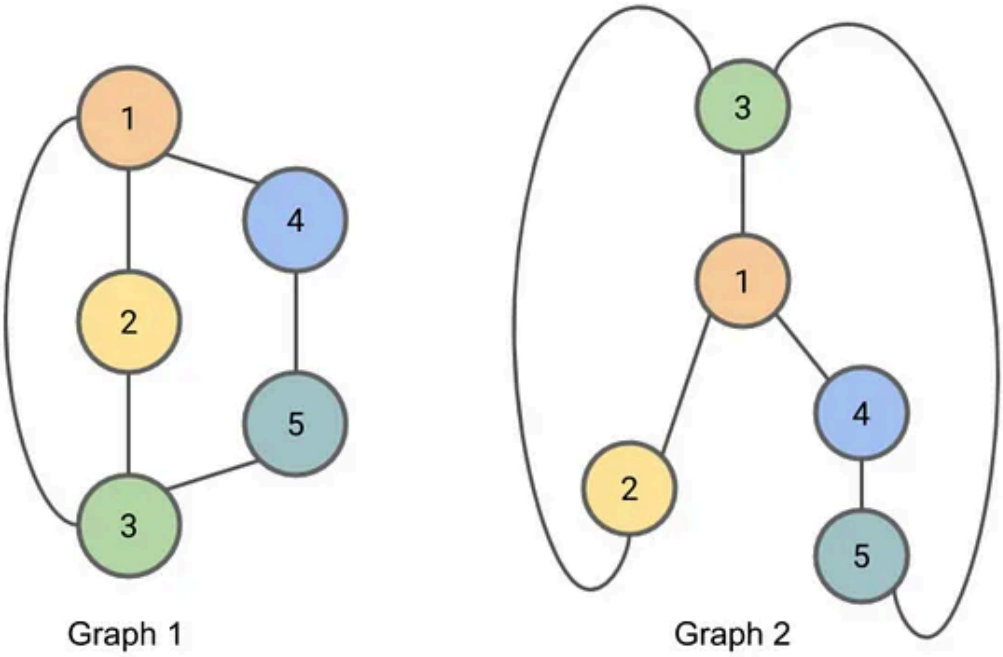


DGCNN addresses the same problem on how to sequentially read a graph and associate weights in a consistent order. So it introduces the SortPooling layer which sorts graph nodes by color so that neural networks can be trained on these ordered nodes.



Source

By presenting graphs in a consistent way according to their topology, the 1-D convolution and dense layers will be much easier to train. For example, the graphs below are isomorphic and will be presented in the same way to the NN after the SortPooling layer.



Source

So how can we color nodes in DGCNN? It uses the Weisfeiler-Lehman algorithm.

For all nodes $v_i \in \mathcal{G}$:

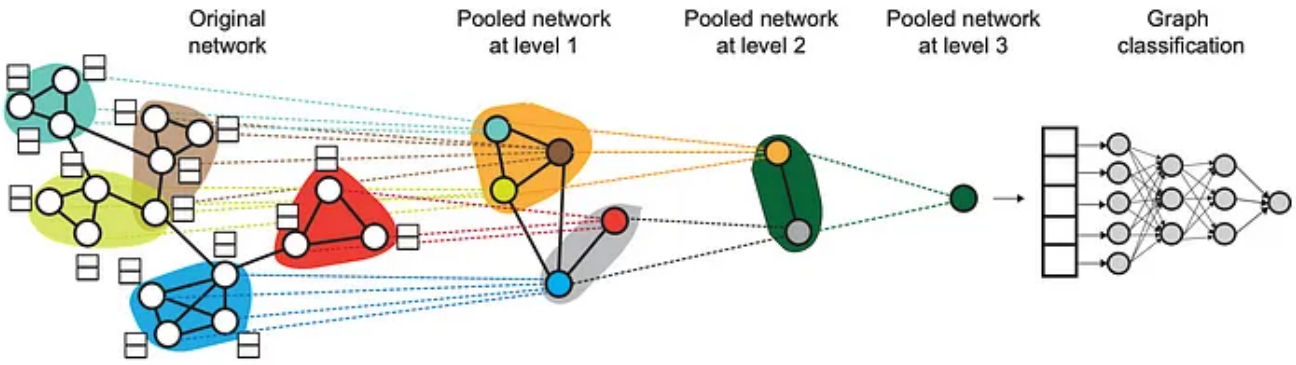
- Get features[4] $\{h_{v_j}\}$ of neighboring nodes $\{v_j\}$
- Update node feature $h_{v_i} \leftarrow \text{hash}\left(\sum_j h_{v_j}\right)$, where hash$(\cdot)$ is (ideally) an injective hash function

Repeat for k steps or until convergence.

Source

The concept is very similar to our first clustering example for Zachary's karate club. But we will not get into the details here. Please refer to the DGCNN paper. Basically, it learns the network embedding (signature) of the nodes based on the network topology. It concatenates a node's color with its immediate neighbors and then sorts it lexicographically to assign new colors. Nodes with the same signature will be assigned with the same color. The process is repeated until the colors converge or after h iterations. In the end, nodes with the same color share the same structural role within the graph. Then the colors in the last convolution layer will be used to sort the nodes. Therefore, a consistent ordering is imposed for graph nodes. After sorting, DGCNN truncates or extends the array of nodes. This supports graphs with different numbers of nodes.

DiffPool



Source

DiffPool is based on a hierarchical concept in generating a coarser graph. It learns a soft cluster assignment matrix S^l with element i, j contains the probability for the node *i* in layer l to be clustered and assigned to node *j* in the next hierarchical layer in a coarser graph.

the number of nodes in layer l+1

$$S^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}.$$

the number of nodes in layer l

Then, this soft assignment can be used to calculate the new node embedding and adjacency matrix for the next hierarchical layer.

the input node embedding matrix for layer l

a new matrix of node embeddings

$$X^{(l+1)} = S^{(l)^T} Z^{(l)} \in \mathbb{R}^{n_{l+1} \times d},$$
$$A^{(l+1)} = S^{(l)^T} A^{(l)} S^{(l)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}}.$$

adjacency matrix for layer l

where

$$Z^{(l)} = \text{GNN}_{l,\text{embed}}(A^{(l)}, X^{(l)}).$$

Source for the equations

With S^l generated by

$$S^{(l)} = \text{softmax}\left(\text{GNN}_{l,\text{pool}}(A^{(l)}, X^{(l)})\right)$$

This model is differentiable and the objective is to learn both GNN models that generate Z and S.

Credits and References

Graph Convolutional Networks

Semi-Supervised Classification with Graph Convolutional Networks

Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

Math Behind GCN

Graph Convolutional Networks on GitHub

A Comprehensive Survey on Graph Neural Networks

Graph Neural Networks: A Review of Methods and Applications

An End-to-End Deep Learning Architecture for Graph Classification

Machine Learning with Graphs

Benchmarking Graph Neural Networks

Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering

Spectral Networks and Deep Locally Connected Networks on Graphs

Machine Learning | Artificial Intelligence | Data Science | Deep Learning

Follow

Written by Jonathan Hui

41K Followers · 39 Following

Deep Learning

Responses (6)

Shubhankar Samanta

What are your thoughts?

Keechu
Dec 19, 2024

Hi Jonathan,

This topic is incredibly interesting, and I would like to explore it further. I plan to include the 10th figure, which describes the GCN layers, in my research. I would like to seek your approval to obtain the copyright for the figure. Would that be alright?

Kind regards,

Jasmin

1 reply Reply

Wang yiyi
Nov 17, 2024

Thank you sir for this great resource. I have one question could we get the nodes embedding without training? GCN

Reply

Aaronjohnspalding
Nov 8, 2022

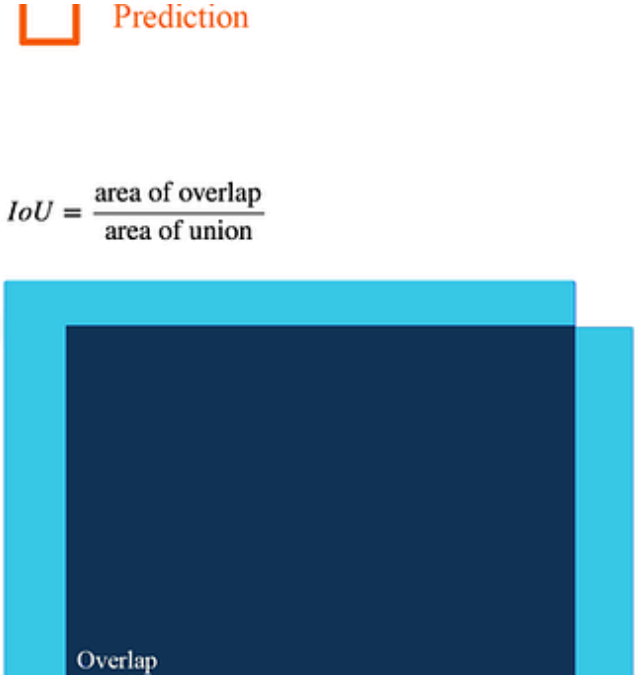
Thank you sir for this great resource. I have a question about " For X, we simply use an identity matrix as there is no node feature " In this sense, the input X is simply a 34×34 identity matrix without any

meaniningful information, why would it... more

1 reply Reply

See all responses

More from Jonathan Hui



Jonathan Hui

mAP (mean Average Precision) for Object Detection

AP (Average precision) is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, SSD, etc. Average precision...

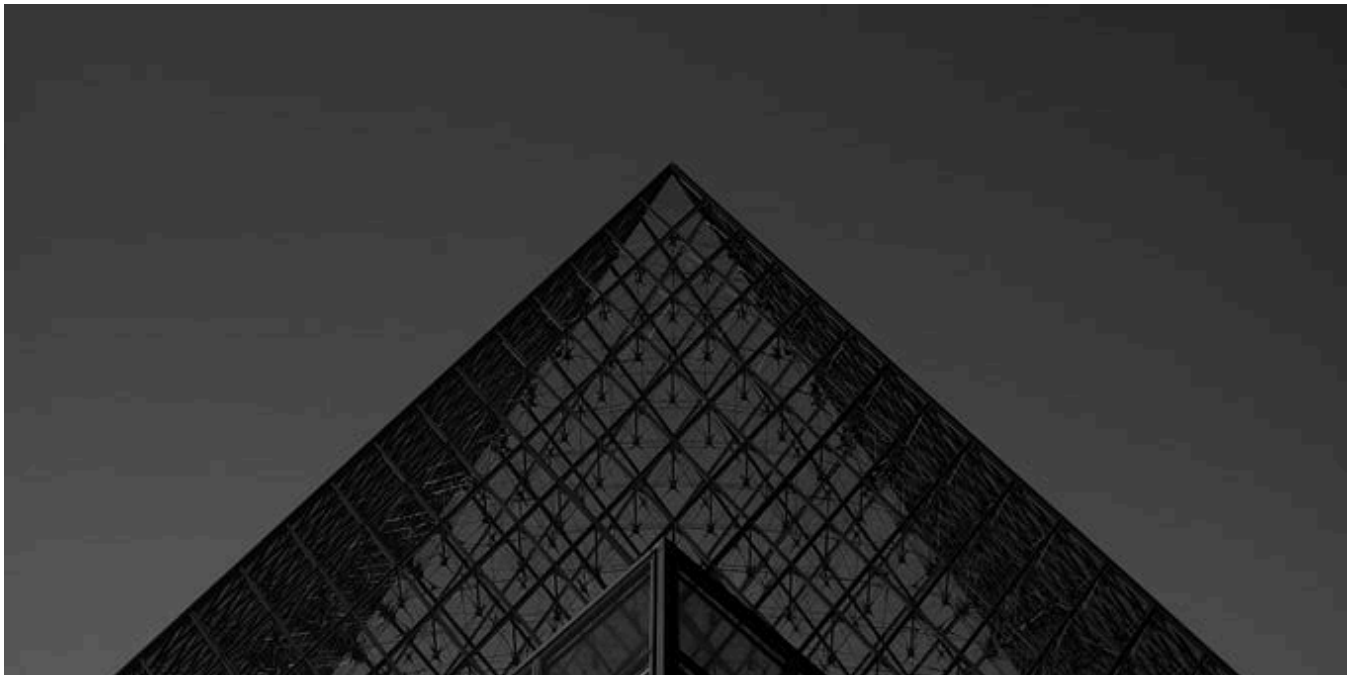
Mar 7, 2018 7.6K 54

Jonathan Hui

Machine Learning—Singular Value Decomposition (SVD) & Principal Component Analysis (PCA)

In machine learning (ML), one of the most important linear algebra concepts is the singular value decomposition (SVD). With all the raw...

Mar 7, 2019 4K 35



Jonathan Hui

Understanding Feature Pyramid Networks for object detection (FPN)

Detecting objects in different scales is challenging in particular for small objects. We can use a pyramid of the same image at different...

Mar 27, 2018 4.2K 30

Jonathan Hui

RL — Proximal Policy Optimization (PPO) Explained

A quote from OpenAI on PPO:

Sep 17, 2018 1.8K 10

See all from Jonathan Hui

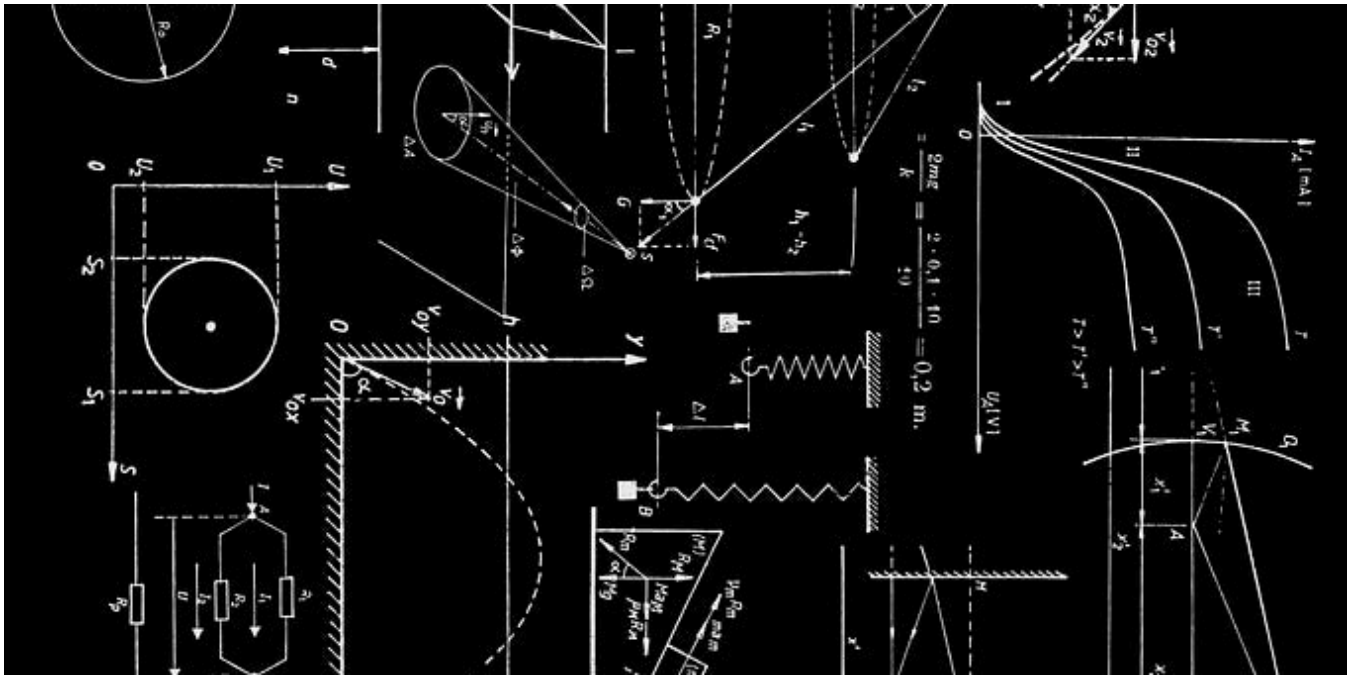
Recommended from Medium

In Level Up Coding by Sahib Dhanjal

How To Train Your PyTorch Models With Less Memory

Strategies I regularly use to reduce GPU memory consumption by almost 20x

Feb 24 428 3



Anna Alexandra Grigoryan

Understanding Physics-Informed Neural Networks (PINNs)—Part 1

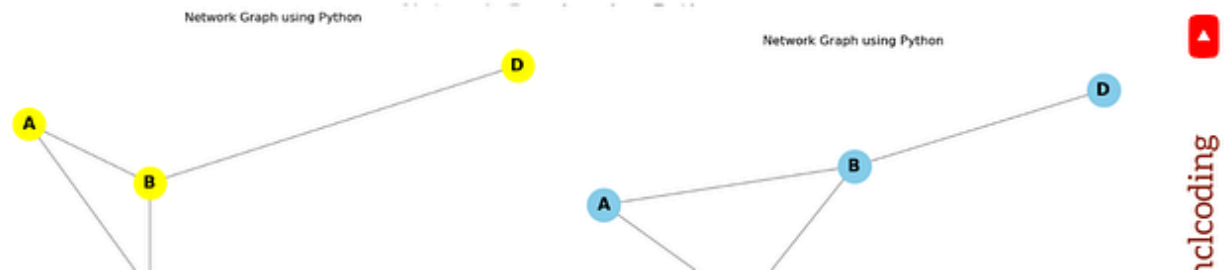
Physics-Informed Neural Networks (PINNs) represent a unique approach to solving problems governed by Partial Differential Equations (PDEs)...

Dec 25, 2024 6

```
edges = [("A", "B"), ("A", "C"), ("B", "C"), ("B", "D"), ("C", "E")]
G.add_edges_from(edges)

plt.figure(figsize=(8, 6))
nx.draw(G, with_labels=True, node_color="yellow", node_size=1000,
        edge_color="gray", font_size=16, font_weight="bold")
plt.title("Network Graph using Python")
plt.show()

#source code --> clcoding.com
```



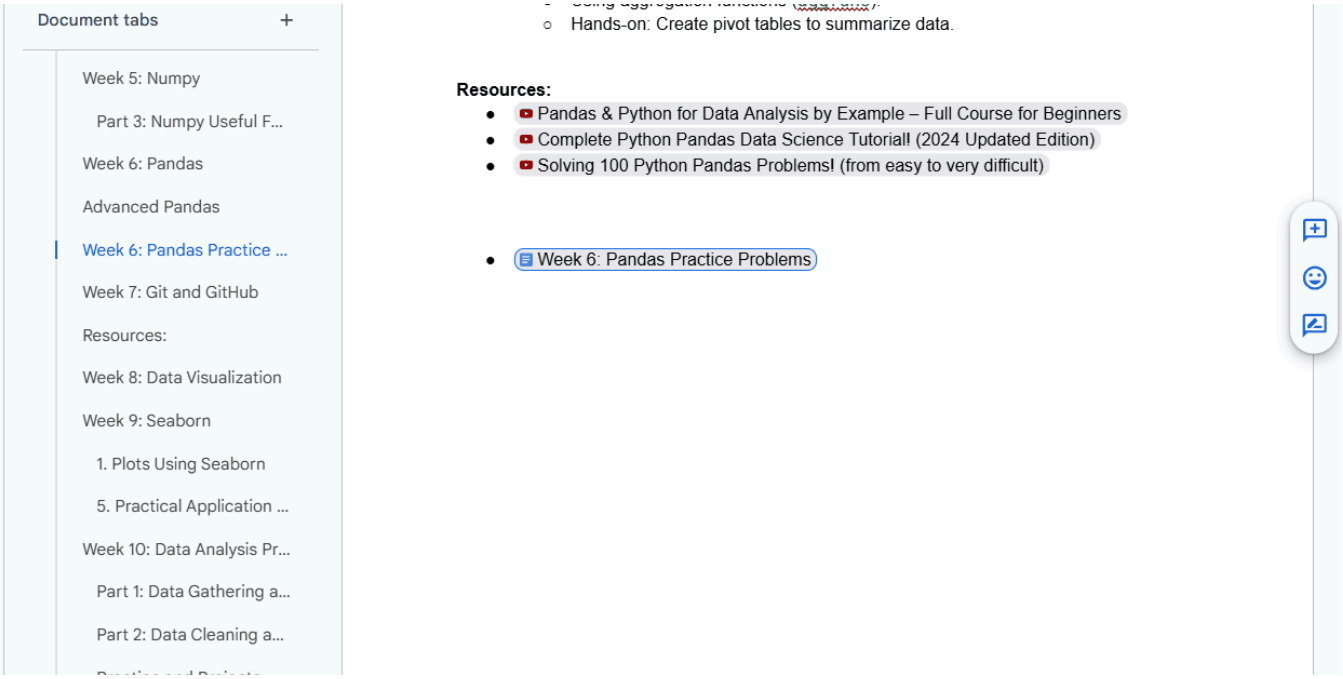
/c /Pythoncoding clcoding

Python Coding

Network Graph using Python

This code snippet demonstrates how to create and visualize a simple network graph using the networkx and matplotlib libraries in Python.

Nov 10, 2024 5 1

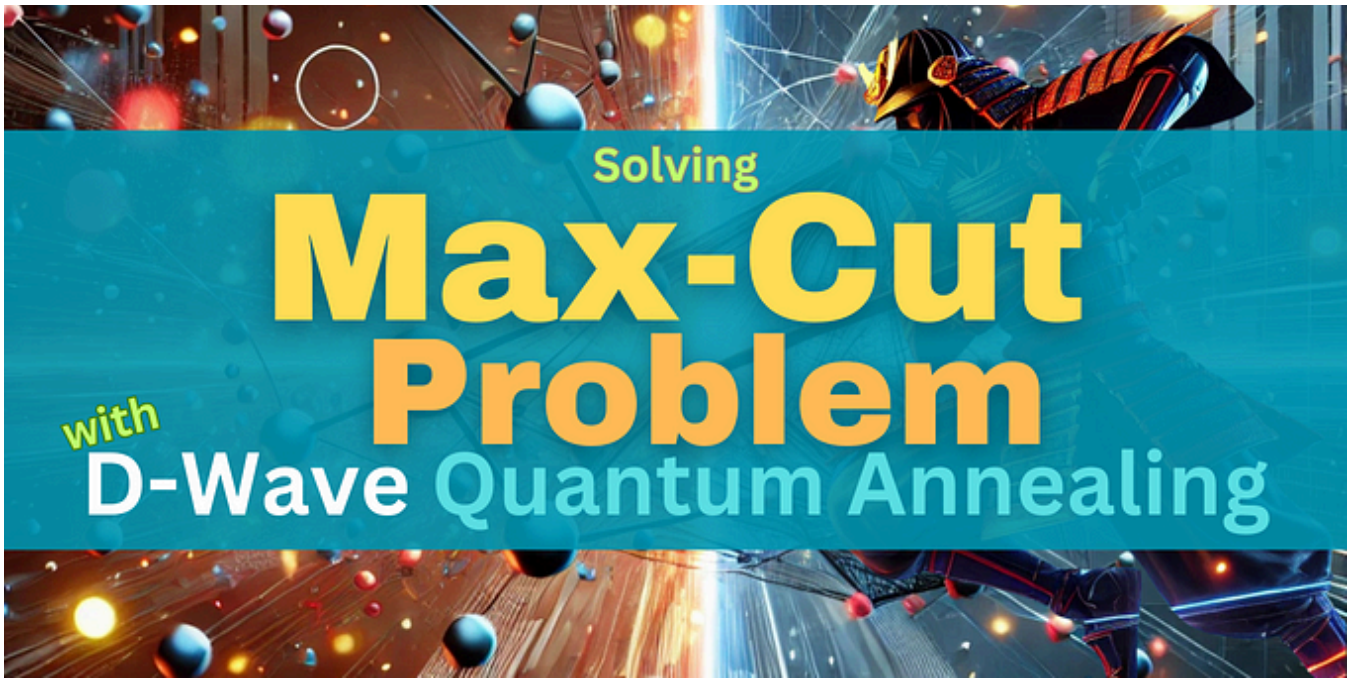


In Data Scientist's Diary by Amit Yadav

Graph-Based Anomaly Detection Techniques

Anomalies—those pesky, out-of-place data points—can be goldmines for insights or red flags in critical systems. Whether you're...

Oct 21, 2024 61



Naoki

Solving Max-Cut Problems with D-Wave Quantum Annealing

Split the Network for Maximum Gain!

Nov 24, 2024 4

Austin Starks

You are an absolute moron for believing in the hype of “AI Agents”.

All of my articles are 100% free to read. Non-members can read for free by clicking my friend link!

Jan 11 7.6K 333

See more recommendations