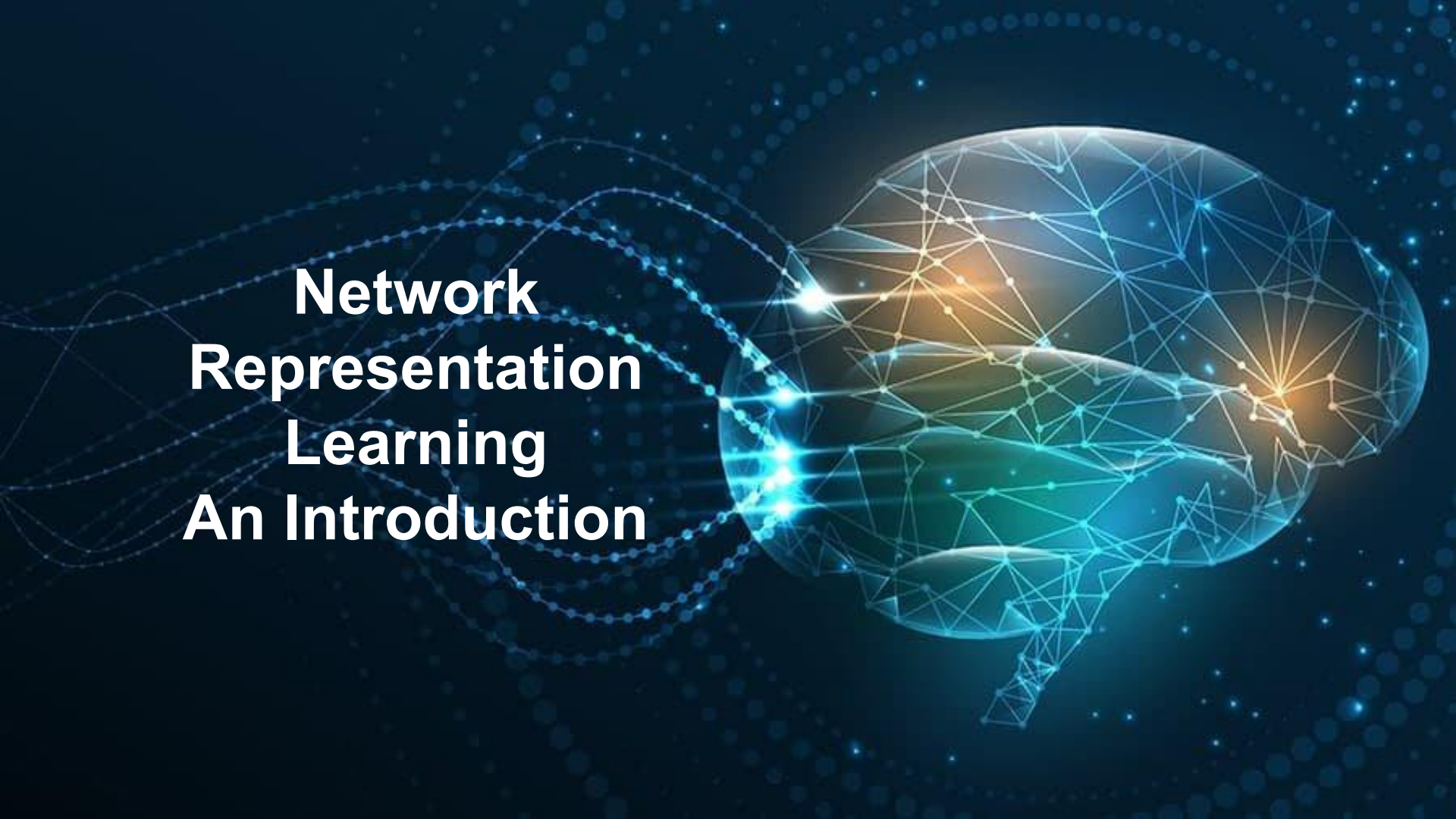
Network Representation Learning An Introduction



Network Representation Learning

Network Representation Learning (NRL) aims to represent the nodes and edges of complex graph structured data into a low dimensional latent space while preserving the network topology, vertex attributes, edge attributes along with some other side information such as vertex labels.

Use of NRL based representations

Representations derived by NRL are used to perform various downstream tasks as follow:

- Clustering
- Link Prediction
- Recommendation Systems
- Vertex Classification

Different NRL Paradigms

To learn the complex graph structure properties, different network paradigms have been proposed.

- **Matrix Factorization** based methods
  e.g. MNMF (Modularity maximized NMF), HOPE
- **Random Walk** based methods
  e.g. DeepWalk, Node2Ve
- **Graph Neural Network** based methods
  e.g. GCN, GraphSage
- **Transformer** based methods
  e.g. Graph Transformers, Graph-Bert

DeepWalk

A Random Walk Based Method

Ref: https://arxiv.org/pdf/1403.6652.pdf

Introduction

Natural Language Processing has shown significant results by using sentences or continuous sequences of words to train the learning models, e.g. **Skip-gram** or **Continuous Bag of Words** Model. DeepWalk proposes to adapt the methodology in graph context by generating random sequences of nodes by traversing over the edges. This sequences are used to learn the social representations of vertices. Social Representations are the latent features that capture the neighbourhood similarity and community relationship.

DeepWalk

- DeepWalk learns the generalized vertex label independent representations of graph nodes, which can be used different downstream tasks.
- DeepWalk performs quite well when the graph is sparse which can act as a barrier for performance for others.
- Strong performance with DeepWalk's representations can be achieved by using very simple classifiers.
- DeepWalk provides another benefit of being an online algorithm and trivially parallelizable.

Algorithm 1 DEEPWALK(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t

Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for each $v_i \in \mathcal{O}$ **do**
    $\mathcal{W}_{v_i} = RandomWalk(G, v_i, t)$
    SkipGram$(\Phi, \mathcal{W}_{v_i}, w)$
  end for
end for

Algorithm Part 1:

According to deepwalk algorithm, for each vertex in graph ***γ*** walks are generated each with length **t**. In this generated sequence, window size of **w** is considered on both sides of the vertex under consideration. Dimension **d** denotes the dimension of the learned representations.

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

1: for each $v_j \in \mathcal{W}_{v_i}$ do
2: for each $u_k \in \mathcal{W}_{v_i}[j-w : j+w]$ do
3: $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
4: $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
5: end for
6: end for

Algorithm Part 2:

On the generated random sequences, skip-gram algorithm is applied which updates the the vertex representations according to the loss.

Working of DeepWalk

- First for any vertex, γ random walks are generated of length t.
- Walks starting from a vertex are generated by choosing next vertex uniformly randomly from all neighbours of the current vertex.
- All this walks are used as context to learn the representations using Skipgram model. Skipgram is a language model that maximizes the co-occurrence probability of words in the context windows of size w on both sides of the word under consideration given vertex representations.
  Loss function used to maximize the probability of context words:

$$\underset{\Phi}{\text{minimize}} \quad -\log \Pr\left(\{v_{i-w}, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{i+w}\} \mid \Phi(v_i)\right)$$

Working of DeepWalk Continued ...

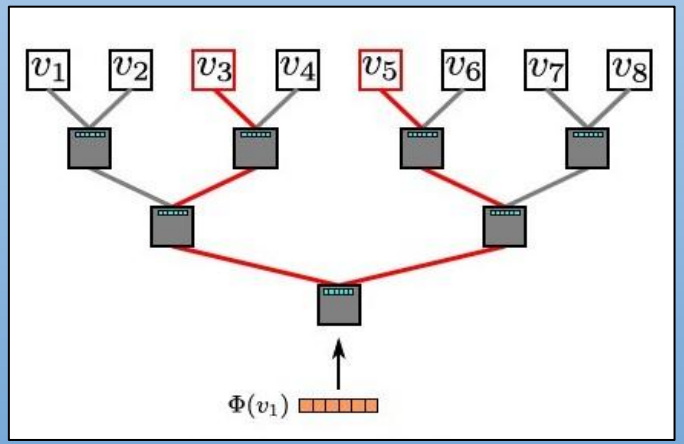
- To solve the loss function optimization problem, one standard assumption is that event of different words occurring in the context window of a vertex are independent. This simplifies the probability to:

$$-\log \prod_{j=0, j\neq m}^{2m} P(w_{c-m+j} | w_c)$$

- Now to calculate $Pr(u_k \mid \phi(v))$, using classifiers like linear classifiers would take $|V|$ operations which will be computationally expensive. To solve this issue, hierarchical softmax is used to calculate the probability.

Working of DeepWalk Continued ...

- **Hierarchical Softmax**: In this, all graph nodes are assumed as leaves of a balanced binary tree and all other nodes of the tree act as binary classifiers. Calculating the probability of a vertex u occurring given another vertex v's representation is considered as maximising the probability of path from root to the leaf u,. This turns into adjacent equation. This b's are binary classifiers. This reduces the complexity of $Pr(u_k \mid \phi(v))$ from O(|V|) to O(|log(V)|). Now using SGD, parameters of classifiers and vertex representations are updated.



$$\Pr(u_k \mid \Phi(v_j)) = \prod_{l=1}^{\lceil \log |V| \rceil} \Pr(b_l \mid \Phi(v_j))$$

Performance

- DeepWalk being an online algorithm, generates embeddings which can be updated in case of graph modification.
- DeepWalk is highly parallelizable and computationally fast.
- DeepWalk outperforms most paradigms in use until deepwalk was proposed on various downstream tasks.
- DeepWalk performance is much better than paradigms like EdgeCluster, Modularity, and wvRN when data is sparse.

Node2Vec

A Random Walk Based Method

Ref: https://arxiv.org/pdf/1607.00653.pdf

Introduction

Node2Vec tries to optimize an objective that seeks to preserve the neighbourhood structure using stochastic gradient descent. Node2Vec like DeepWalk, uses the randomly sampled node sequences to learn the representation. But it differs from DeepWalk in the way those sequences are sampled - Node2Vec uses a 2nd order random walks which are sampled using flexible sampling strategy which takes into account both community similarity as well as structural similarity.

Node2Vec

- Semi-supervised algorithm for scalable feature learning of networks.
- Biased random walks are used to explore diverse neighbourhood of a vertex.
- Using the tunable parameters ***p*** and ***q*** for the sampling can model the full spectrum of equivalences observed in networks.
- Feature representations learnt by nodes in Node2Vec can be extended to edges by using different binary operators on vertex representations.
- Major phases of Node2Vec are trivially parallelizable.

Tunable Sampling Strategy

- Prediction task often shuttle between 2 parts: homophily and structural equivalence.
- Structural equivalence deals with close embeddings of nodes with structurally equivalent positions in network. BFS look out for immediate neighbourhood of the node, which can be used to model the structural equivalence.
- Homophily deals with the close embedding of nodes in closely connected clusters, communities. DFS can be used go a little further away from the node and reach out to different connected nodes in a cluster.

So for a good sampling algorithm, it should have a mixture of both BFS and DFS.

Tunable Sampling Strategy Continued ...

Node2Vec uses a generalized sampling strategy with tunable parameters p and q which can model complete spectrum of equivalences in networks.

Consider a random walk which just traversed edge (t,v) and now on node v. To choose the next vertex, the probabilities are decided according to following:

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}$$

where π_{vx} is the unnormalized transition probability between nodes v and x, and Z is the normalizing constant.

Tunable Sampling Strategy Continued ...

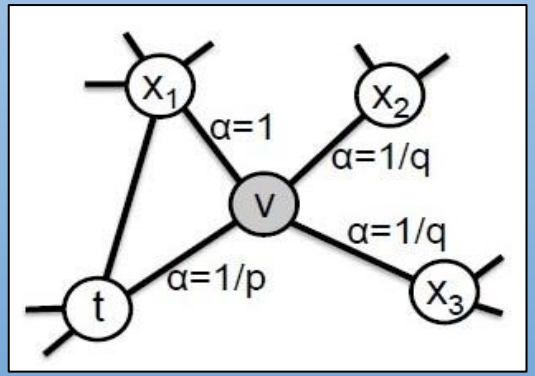
Now how that π function is constructed let's see:

$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

where the d value denotes the the distance between nodes t and x.

Return parameter *p*: It controls the likelihood of immediately revisiting a node.
In-out parameter *q*: if q > 1, the walk is biased for nodes close to node t giving local neighbourhood view. If q < 1, the walk explores further vertices going away From t.



Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
π = PreprocessModifiedWeights(G, p, q)
$G' = (V, E, \pi)$
Initialize $walks$ to Empty
for $iter = 1$ **to** r **do**
  for all nodes $u \in V$ **do**
    $walk$ = node2vecWalk(G', u, l)
    Append $walk$ to $walks$
f = StochasticGradientDescent(k, d, $walks$)
return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
Inititalize $walk$ to $[u]$
for $walk\_iter = 1$ **to** l **do**
  $curr = walk[-1]$
  V_{curr} = GetNeighbors($curr$, G')
  s = AliasSample(V_{curr}, π)
  Append s to $walk$
return $walk$

Node2Vec Algorithm

This is node2vec algorithm. First this algorithm calculates the **π**, 2nd order transition probability. Then using the matrix, random walks are generated by node2vecWalk. The embedding function ***f*** is learnt by optimizing using stochastic gradient descent.

Working of Node2Vec

It tries to optimise the loss function as shown:

$$\max_f \sum_{u \in V} \log Pr(N_S(u)|f(u)).$$

where $N_s(u)$ denotes the context neighbourhood of the vertex u.

This is simplified using 2 assumptions:

1. Conditional Independence: Likelihood of observing 2 different nodes in the same context is independent of each other making it:

$$Pr(N_S(u)|f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i|f(u)).$$

2. Symmetry in feature space: 2 nodes in neighbourhood of each other have symmetric effects over each other:

$$Pr(n_i|f(u)) = \frac{\exp(f(n_i) \cdot f(u))}{\sum_{v \in V} \exp(f(v) \cdot f(u))}.$$

Working of Node2Vec Continued ...

So it finally reduces to optimizing the function: where Z_u is the softmax denominator.

$$\max_f \sum_{u \in V} \left[-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right]$$

This function is optimized using stochastic gradient descent to learn f.

The parameters ***p*** and ***q*** are learnt in semi supervised manner using very small portion of labelled data available in the network.

Performance

- It outperforms state-of-the-art methods by up to 26.7% on multi-label classification and up to 12.6% on link prediction.
- Can learn the edge representation too using binary operators on the node representations.
- Can perform well even when very less labelled data is provided.
- Scalable for huge graphs.

GCN

A Neural Network Based Method

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

Introduction

- Graph Neural Networks take graph data as input to the neural network model and by computing through several hidden layers gives the node embeddings.
- GCN averages out the neighbourhood node properties of the node under consideration using stacked convolutional layers.
- It takes input the adjacency matrix and the node attributes.
- Learning in a semi supervised way, it trains itself on the labelled data available.
- Convolutional layers by their construction are trivially parallelizable.

Working of GCN

- GCN tries to optimize the following loss function:

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{\text{reg}}, \quad \text{with} \quad \mathcal{L}_{\text{reg}} = \sum_{i,j} A_{ij} \| f(X_i) - f(X_j) \|^2 = f(X)^\top \Delta f(X)$$

where L_0 is the supervised loss w.r.t to the labelled nodes, f(.) a neural network like differentiable function and Δ being the unnormalized graph laplacian of the input graph G.

$$\mathcal{L} = - \sum_{l \in \mathcal{Y}_L} \sum_{f=1}^{F} Y_{lf} \ln Z_{lf}$$

L_0 loss is calculated by the cross entropy loss over all labels for the vertices which are labeled.

Working of GCN Continued ...

- A and X are fed into the neural network where A stands for adjacency/weight matrix and X stands for node attributes/properties matrix.
- Now as the input propagates through the neural network, the next hidden state is calculated in the fashion:

$$\mathbf{X}^{p+1} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}^{p}\mathbf{\Theta}^{p}\right),$$

where $(p+1)^{th}$ hidden layer is calculated as above. A is the adjacency matrix, A^{hat} is A + I, D^{hat} is degree matrix of A^{hat}, X^{i} is the hidden i^{th} layer, $\ominus$ is the weight matrix for that corresponding layer and σ is the non linearity.

This propagation rule is motivated via a first-order approximation of localized spectral filters on graphs.

Limitations

Though GCN performs good on many datasets giving new best results, it suffers from following limitations:

- For the computation, graph data has to be stored in memory which can acts as a barrier.
- Original GCN architecture naturally does not support directed edges.
- How much effect the self loop will have on the k neighbourhood depend on dataset.
- Transduction of GCN interferes with the generalization, making the learning of representations of the unseen nodes in the same graph and the nodes in an entirely different graph more difficult.

GraphSAGE

A Neural Network Based Method

Ref: https://arxiv.org/pdf/1706.02216.pdf

Introduction

- GraphSAGE is a spatial propagation-based graph convolutional network.
- GraphSAGE aggregates the neighbourhood node features of the node under consideration by using different aggregation functions.
- It can be trained in a completely unsupervised as well as completely supervised way.
- Provides inductive way to learn the features and perform very well on unknown data and structural groups.
- Even while it uses feature information to learn, it can learn structural similarity.

Working of GraphSAGE

- In GraphSAGE, one main difference is that neighbourhood node features are aggregated using aggregation operators instead of having static weight matrices for the purpose. There are different choices available for this.

  In the original GraphSAGE paper, they used a fixed size set of nodes from the neighbourhood of a node and randomly shuffled them before passing to the aggregation function.

- It learns k different aggregation functions for k layers of the neural network.

  One main thing about aggregation function should be that they should be Independent of the order in which nodes are given as neighbourhood is not numbered.

Algorithm

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Aggregation Function

3 aggregation functions are mentioned in GraphSAGE:

1. **Mean aggregator:** It is different from others in way it joins nodes previous layer embedding to the neighbourhood embeddings for mean.
2. **LSTM aggregator:** It has one problem as LSTM processes input in sequential way so indirectly some sequence structure comes in play. To overcome this, we randomly shuffle the nodes.
3. **Pool aggregator:** It applies non linearity after a fully connected layer and gives aggregator output as the pooling performed over all those results.

Attention

It is all you need

Ref: https://arxiv.org/pdf/1706.03762.pdf

Introduction

- Attention is mechanism which enables to generalize relations between input and output at global level irrespective of sequence length.
- Transformer, a new architecture proposed based on attention has outperformed all the existing SOTA architectures.
- Calculating attention is parallelizable, so giving edge over previous sequential processing of the sequences by RNNs.

Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

- Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.
- The compatibility function between key and the query can be calculated in ways like dot product attention and single layer feed forward network.

Calculating Attention

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Here Q is query matrix, K is key matrix and V is values matrix. Dimension of query and key vectors is d_k and dimension of value vector is d_v. Attention functions used is either dot product attention or additive attention. In dot product attention, dot product of query and key is scaled by sqrt(d_k) while in additive attention, they are passed through a feed forward network of single hidden layer.

While for small values of d_k, additive attention has proved to provide better results, but for larger values of d_k, dot product attention is preferred.

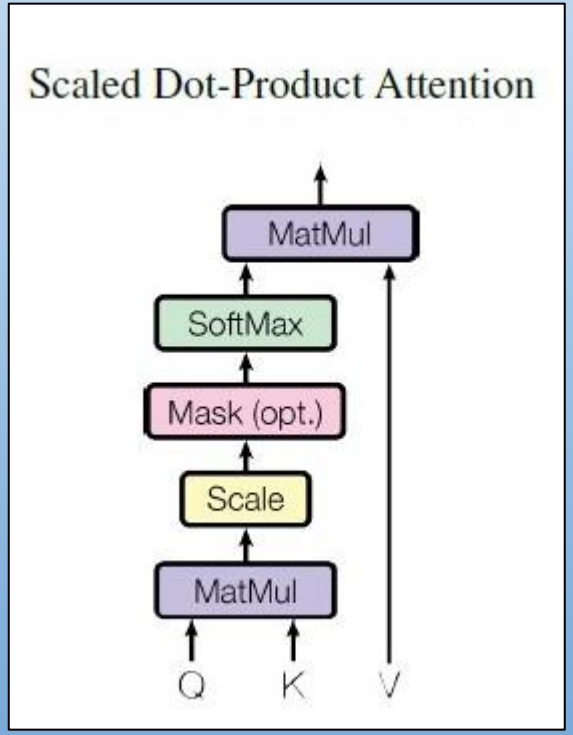
Multi Head Attention

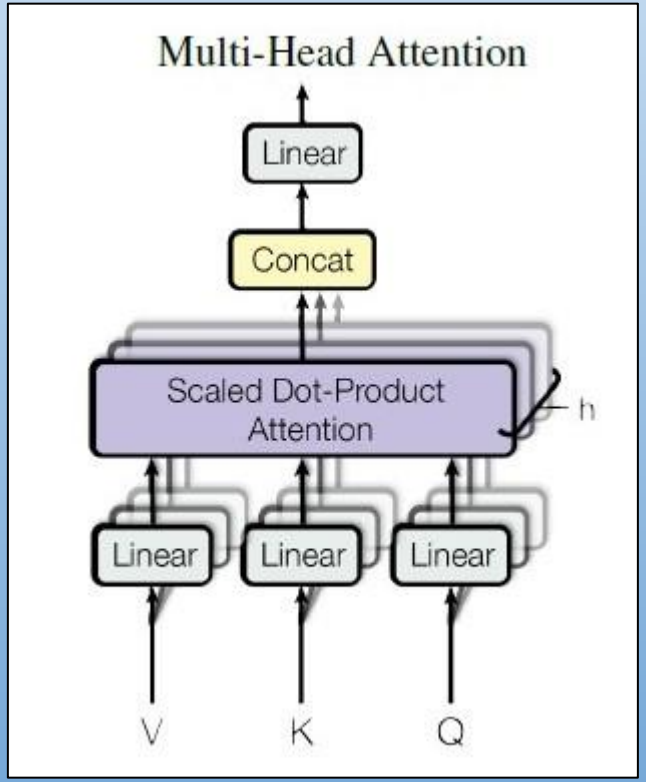
Instead of performing a single attention function with d_{model} dimensional keys, values and queries, it is better to linearly project the queries, keys and values h times with different, learned linear projections to dk, dk and dv dimensions, respectively. On each of these projected versions of queries, keys and values, perform the attention function in parallel, yielding d_v dimensional output values. These are concatenated and once again projected, resulting in the final values.

In this multihead setting, $d_k = d_v = d_{model} / h$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

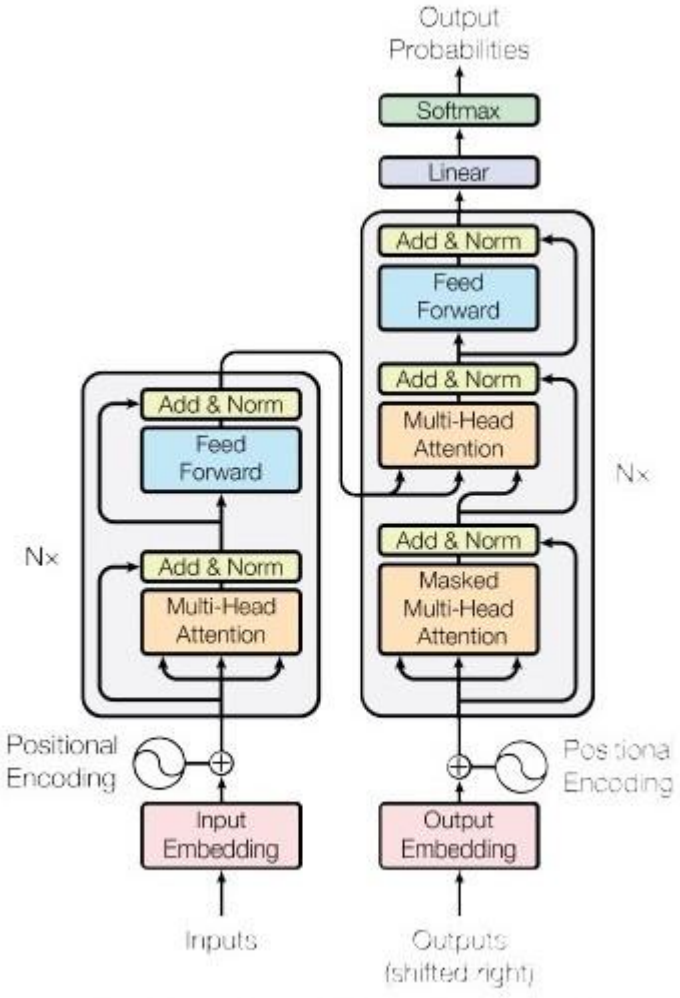
Attention Calculation (Diag)





Transformer Architecture

There are N stacked units of each at encoder as well as decoder part.



Positional Encoding

When attention is calculated, there is no sequential order is considered. So to enforce the sequence information, positional encoding is added to the embeddings fed into model.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Graph Transformer

Transformer Based Method

Ref: https://arxiv.org/pdf/2012.09699.pdf

Introduction

- Graph Transformer uses transformer based architecture on graph data along with positional encoding to generate inductive model.
- It calculates attention over neighbourhood of the node under consideration rather than all nodes like sentence. It uses graph sparsity efficiently for attention calculation.
- Due to its inductive nature, generalizes well to unknown nodes and datasets.

Working of Graph Transformer

- We first prepare node input features and node embeddings to be passed to the Graph Transformer.
  For a graph G with node features αi ∈ $R^{dn *1}$ for each node i and edge features eij ∈ $R^{de *1}$ for each edge between node i and node j.

$$\hat{h}_i^0 = A^0\alpha_i + a^0 \;;\; e_{ij}^0 = B^0\beta_{ij} + b^0,$$
$$A^0 \in \mathbb{R}^{d\times d_n},\ B^0 \in \mathbb{R}^{d\times d_e} \text{ and } a^0, b^0 \in \mathbb{R}^d$$

$$\lambda_i^0 = C^0\lambda_i + c^0 \;;\; h_i^0 = \hat{h}_i^0 + \lambda_i^0,$$

  Here λi is the positional encodings.

Working of Graph Transformer Continued ...

- Positional encodings are calculated from the factorization of the graph laplacian.

$$\Delta = \mathrm{I} - D^{-1/2} A D^{-1/2} = U^T \Lambda U,$$

  U is the matrix of eigenvectors and Λ is diag matrix of eigenvalues. We use the k smallest non-trivial eigenvectors of a node as its positional encoding and denote by λ_i for node i.
- These encodings are fed into multiple layers with attention.

Working of Graph Transformer Continued ...

$$\hat{h}_i^{\ell+1} = O_h^{\ell} \Big\Vert_{k=1}^{H} \Big(\sum_{j \in \mathcal{N}_i} w_{ij}^{k,\ell} V^{k,\ell} h_j^{\ell} \Big),$$

$$\text{where, } w_{ij}^{k,\ell} = \text{softmax}_j \Big(\frac{Q^{k,\ell} h_i^{\ell} \cdot K^{k,\ell} h_j^{\ell}}{\sqrt{d_k}} \Big)$$

$$\hat{\hat{h}}_i^{\ell+1} = \text{Norm}\Big(h_i^{\ell} + \hat{h}_i^{\ell+1} \Big),$$

$$\hat{\hat{\hat{h}}}_i^{\ell+1} = W_2^{\ell} \text{ReLU}(W_1^{\ell} \hat{\hat{h}}_i^{\ell+1}),$$

$$h_i^{\ell+1} = \text{Norm}\Big(\hat{\hat{h}}_i^{\ell+1} + \hat{\hat{\hat{h}}}_i^{\ell+1} \Big),$$

This equations are transformer equations used to calculate values across layers in network. The representation at last layer are passed through MLP layer for task dependent learning and loss function.

Working of Graph Transformer Continued ...

It can be extended to include the edge information too in following way:

$$\hat{h}_i^{\ell+1} = O_h^{\ell} \Big\Vert_{k=1}^{H} \Big(\sum_{j \in \mathcal{N}_i} w_{ij}^{k,\ell} V^{k,\ell} h_j^{\ell} \Big),$$

$$\hat{e}_{ij}^{\ell+1} = O_e^{\ell} \Big\Vert_{k=1}^{H} \Big(\hat{w}_{ij}^{k,\ell} \Big), \quad \text{where,}$$

$$w_{ij}^{k,\ell} = \text{softmax}_j(\hat{w}_{ij}^{k,\ell}),$$

$$\hat{w}_{ij}^{k,\ell} = \Big(\frac{Q^{k,\ell} h_i^{\ell} \cdot K^{k,\ell} h_j^{\ell}}{\sqrt{d_k}} \Big) \cdot E^{k,\ell} e_{ij}^{\ell},$$

$$\hat{\hat{h}}_i^{\ell+1} = \text{Norm}\Big(h_i^{\ell} + \hat{h}_i^{\ell+1} \Big),$$

$$\hat{\hat{\hat{h}}}_i^{\ell+1} = W_{h,2}^{\ell} \text{ReLU}(W_{h,1}^{\ell} \hat{\hat{h}}_i^{\ell+1}),$$

$$h_i^{\ell+1} = \text{Norm}\Big(\hat{\hat{h}}_i^{\ell+1} + \hat{\hat{\hat{h}}}_i^{\ell+1} \Big),$$

$$\hat{\hat{e}}_{ij}^{\ell+1} = \text{Norm}\Big(e_{ij}^{\ell} + \hat{e}_{ij}^{\ell+1} \Big),$$

$$\hat{\hat{\hat{e}}}_{ij}^{\ell+1} = W_{e,2}^{\ell} \text{ReLU}(W_{e,1}^{\ell} \hat{\hat{e}}_{ij}^{\ell+1}),$$

$$e_{ij}^{\ell+1} = \text{Norm}\Big(\hat{\hat{e}}_{ij}^{\ell+1} + \hat{\hat{\hat{e}}}_{ij}^{\ell+1} \Big),$$

Graph BERT

Transformer Based Method

Ref: https://arxiv.org/pdf/2001.05140.pdf

Introduction

- Graph Transformer uses transformer based architecture on graph data along with positional encoding to generate inductive model.
- It calculates attention over neighbourhood of the node under consideration rather than all nodes like sentence. It uses graph sparsity efficiently for attention calculation.
- Due to its inductive nature, generalizes well to unknown nodes and datasets.

Improvements

- Training is independent of the links in the graph, making it able to learn more generalized representations.
- Training is done in unsupervised way.
- This model can be transferred with some necessary fine tuning for some specific downstream task which is not possible in most models of GNN
- Learning is done on sampled subgraphs instead of the complete graph.

Some Terminologies

- **Intimacy score:** matrix for graph sub sampling

  $A` = AD^{-1}$ and α is in range [0,1].

$$S = \alpha \cdot \left(I - (1-\alpha) \cdot \bar{A}\right)^{-1}$$

- **Node Context:** given G and intimacy matrix S, node context is defined as:

  $T(v_i) = \{v_j \mid v_j \in V \setminus \{v_i\} \wedge S(i,j) \geq \Theta_i\}$

- **Sampled Subgraph of v_i:** $g_i = T(v_i) \cup \{v_i\}$

  The value of Θ_i is decided so that size of the $T(V_i)$ is k, a constant for model.

Precomputed Embeddings

4 types of embeddings are fed into model for training:

1. Raw Feature Vector Embedding
2. Weisfeiler-Lehman Absolute Role Embedding
3. Intimacy based Relative Positional Embedding
4. Hop based Relative Distance Embedding

Raw Feature Vector Embedding

For each node vj ∈ Vi in the subgraph gi, we can embed its raw feature vector into a shared feature space (of the same dimension dh) with its raw feature vector xj by following way:

$$\mathbf{e}_j^{(x)} = \text{Embed}\left(\mathbf{x}_j\right) \in \mathbb{R}^{d_h \times 1}.$$

where Embed(·) is either a CNN or LSTM/BERT or a fully connected layer network.

Weisfeiler-Lehman Absolute Role Embedding

Using Weisfeiler-Lehman (WL) algorithm nodes are labelled according to their structural roles in the graph data, where the nodes with the identical roles will be labeled with the same code.

$$\mathbf{e}_j^{(r)} = \text{Position-Embed}\left(\text{WL}(v_j)\right)$$
$$= \left[sin\left(\frac{\text{WL}(v_j)}{10000^{\frac{2l}{d_h}}}\right), cos\left(\frac{\text{WL}(v_j)}{10000^{\frac{2l+1}{d_h}}}\right) \right]_{l=0}^{\lfloor \frac{d_h}{2} \rfloor},$$

This embeddings are absolute for a node irrespective of in which subgraph it is.

Intimacy based Relative Positional Embedding

Weisfeiler-Lehman embedding capture the global structure. Intimacy based embedding tries to capture local neighbourhood structure. For this, all nodes in the sampled graph are ordered in the decreasing order of their intimacy score with the node under consideration. Based on that serialized list, position of v_j is defined as $P(v_j)$. Then

$$\mathbf{e}_j^{(p)} = \text{Position-Embed}\left(\mathrm{P}(v_j)\right) \in \mathbb{R}^{d_h \times 1},$$

Hop based Relative Distance Embedding

Formally, for node $v_j \in V_i$ in the subgraph g_i, we can denote its relative distance in hops to v_i in the original input graph as $H(v_j ; v_i)$. Then

$$e_j^{(d)} = \text{Position-Embed}\left(H(v_j; v_i)\right) \in \mathbb{R}^{d_h \times 1}.$$

Graph BERT Working

- Initial h_j^0 is defined for node v_j in subgraph $h_j^{(0)} = \text{Aggregate}(e_j^{(x)}, e_j^{(r)}, e_j^{(p)}, e_j^{(d)})$.

  Initial input vectors of all nodes $H^{(0)}$ in subgraph g_i is defined by contcataning all the h vectors corresponding to nodes in subgraph g_i.

- This H value is given to model and further layers are calculated as shown aside:

$$H^{(l)} = \text{G-Transformer}\left(H^{(l-1)}\right)$$
$$= \text{softmax}\left(\frac{QK^\intercal}{\sqrt{d_h}}\right)V + \text{G-Res}\left(H^{(l-1)}, X_i\right),$$

where

$$\begin{cases} Q &= H^{(l-1)}W_Q^{(l)}, \\ K &= H^{(l-1)}W_K^{(l)}, \\ V &= H^{(l-1)}W_V^{(l)}. \end{cases}$$

Graph BERT Working Continued ...

- Different from conventional residual learning, Graph BERT add the residual terms computed for the target node v_i to the hidden state vectors of all the nodes in the subgraph at each layer.
- At the last layer, fusion function is applied. It average the representations of all the nodes in input list, which defines the final state of the target v_i.

$$\mathbf{z}_i = \text{Fusion}\left(\mathbf{H}^{(D)}\right)$$

Questions?

Thank You !!