CS224W: Machine Learning with Graphs **Fall 2019**

Homework 2

Due 11:59pm PT Thursday October 24 2019

This problem set should be completed individually.

General Instructions

These questions require thought, but do not require long answers. Please be as concise as possible. You are allowed to take a maximum of 1 late period (see the information sheet at the end of this document for the definition of a late period).

Submission instructions: You should submit your answers and code via Gradescope. There will be a separate submission assignment for written and code.

Submitting answers: Prepare answers to your homework in a single PDF file and submit it via Gradescope to the HW2 (Written) assignment. Make sure that the answer to each sub-question is on a *separate, single page*. The number of the question should be at the top of each page. Please use the submission template files included in the bundle to prepare your submission. Failure to use the submission template file will result in a reduction of 2 points from your homework score.

Information sheet: Fill out the information sheet located at the end of the submission template file, and sign it in order to acknowledge the Honor Code (if typesetting the homework, you may type your name instead of signing). This should be the last page of your submission. Failure to fill out the information sheet will result in a reduction of 2 points from your homework score.

Submitting code: Upload a zipfile containing all your on Gradescope to the HW2 (Code) assignment. Failure to submit your code will result in reduction of all points for that question from your homework score.

Homework survey: After submitting your homework, please fill out the Homework 2 Feedback Form. Respondents will be awarded extra credit.

Questions

1 Node Classification [25 points]

1.1 Relational Classification [10 points]

As we discussed in class, we can use relational classification to predict node labels. Consider the graph G as shown in Figure 1. We would like to classify nodes into 2 classes "+" and "-". Labels for node 3, 5, 8 and 10 are given (red for "+", blue for "-"). Recall that using a probabilistic relational classifier to predict label Y_i for node i is defined as:

$$P(Y_i = c) = \frac{1}{|N_i|} \sum_{(i,j)\in E} W(i,j)P(Y_j = c)$$

where $|N_i|$ is the number of neighbors of node i. Assume all the edges have edge weight $W(i,j) = 1$ in this graph. For labeled nodes, initialize with the ground-truth Y labels, i.e., $P(Y_3 = +) = P(Y_5 = +) = 1.0, P(Y_8 = +) = P(Y_{10} = +) = 0$. For unlabeled nodes, use unbiased initialization $P(Y_i = +) = 0.5$. Update nodes by node ID in ascending order (i.e., update node 1 first, then node 2, etc.)

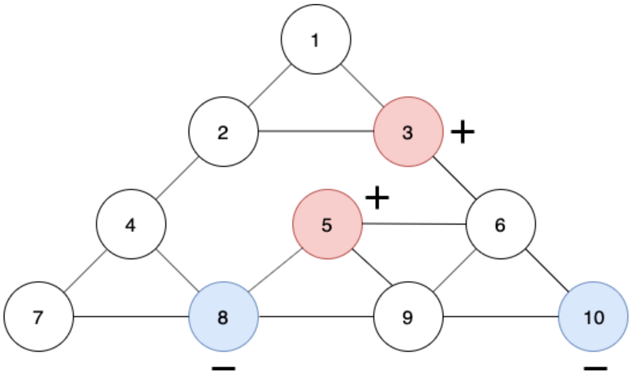


Figure 1: Graph G for Question 1.1

(i) After the second iteration, give $P(Y_i = +)$ for $i = 2, 4, 6$. [6 points]

(ii) If we use 0.5 as the probability threshold, i.e., consider a node i belonging to class "+" if $P(Y_i = +) > 0.5$, which node will belong to class "+" in the end? Which will belong to class "-"? [4 points]

1.2 Belief Propagation [15 points]

In this problem, we will be using Belief Propagation(BP) on Conditional Random Field to solve inference problems.

Conditional Random Fields (CRF) are an important special case of Markov Random Fields to model conditional probability distribution. They define a probability distribution p over variables

$x_1, x_2, ..., x_n$ and observations $y_1, y_2, ..., y_m$, which in turn elicit an undirected graph G. The distribution has the form:

$$p(x_1, ..., x_n | y_1, y_2, ..., y_m) = \frac{1}{Z} \prod_{c \in C} \phi_c(x_c, y_c)$$

where C denotes the set of cliques (i.e., fully connected subgraphs) of G, each factor ϕ_c is a nonegative function over the variables in a clique, and Z denotes the normalizing constant that ensures the distribution p sums to 1.

For the example (Figure 2) below, the filled-in circles represent the observed nodes y_i and the empty circles represent the variables x_i. A clique in this graph is just a pair of two connected nodes, and then we have $p(x_1, ..., x_n | y_1, y_2, ..., y_n) = \frac{1}{Z} \prod_{i=2}^{n} \phi_{i-1,i}(x_{i-1}, x_i) \prod_{j=1}^{n} \phi_j(x_j, y_j)$, where $\phi_j(x_j, y_j)$ describes some statistical dependency between x_j and y_j at each position j, and $\phi_{i-1,i}(x_{i-1}, x_i)$ describes compatibility between the nearby variables x_{i-1} and x_i.

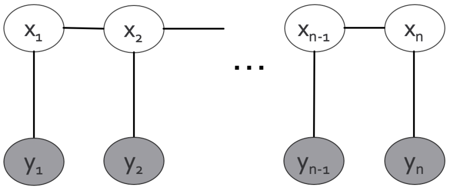


Figure 2: Example of Conditional Random Fields.

For consistency with the lecture slides, we will use $\phi_j(x_j) = \phi_j(x_j, y_j)$ since we have already observed nodes y_j, and use $\psi_{ik}(x_i, x_k) = \phi_{i-1,i}(x_{i-1}, x_i)$ with i, k adjacent. Then the above example can be written as $p(x_1, ..., x_n | y_1, y_2, ..., y_n) = \frac{1}{Z} \prod_{i,k \text{ adjacent}} \psi_{ik}(x_i, x_k) \prod_{j=1}^{n} \phi_j(x_j)$. Note that $\psi_{ik}(x_i, x_k) = \psi_{ki}(x_i, x_k) = \psi_{ki}(x_k, x_i) = \psi_{ik}(x_k, x_i)$.

We define the following for BP as from class:

Message $m_{ij}(x_j)$: can be intuitively understood as a message from hidden node x_i to hidden node x_j about what state node i thinks node j should be in.

$$m_{ij}(x_j) = \sum_{x_i} \phi_i(x_i) \psi_{ij}(x_i, x_j) \prod_{k \in N_i \setminus j} m_{ki}(x_i),$$

where $N_i \setminus j$ denotes neighbors of node i except node j.

Belief $b_i(x_i)$: $b_i(x_i) = \frac{1}{Z} \phi_i(x_i) \prod_{j \in N_i} m_{ji}(x_i)$, where Z denotes the normalizing constant that ensures elements in $b_i(x_i)$ sum to 1.

(i) Consider the network with four hidden nodes shown in Figure 3. Compute the belief at node 1, $b_1(x_1)$, using the belief propagation rules, and write the result in terms of ϕ's and ψ's. [5 points]

(ii) Prove that the belief at node 1 calculated above, $b_1(x_1)$, is the same as the marginal probability of x_1 conditioned on the observations, $p(x_1 | y_1, y_2, y_3, y_4)$. [2 points]

(iii) **Coding.** Let's work with a graph without cycles as shown in Figure 4. Assume x and y only have two states (0 and 1) and the graphical model has 5 hidden variables, and two variables observed with $y_2 = 0, y_4 = 1$. The compatibility matrices are given in the arrays below. For

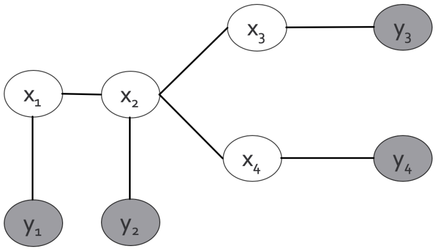


Figure 3: For problem (i)

example, in $\psi_{12}(x_1, x_2)$, row indices are states 0, 1 for x_1, and column indices are states 0, 1 for x_2. Remember that in this case there won't be $\phi_1(x_1), \phi_3(x_3), \phi_5(x_5)$ since there is no observation for them.

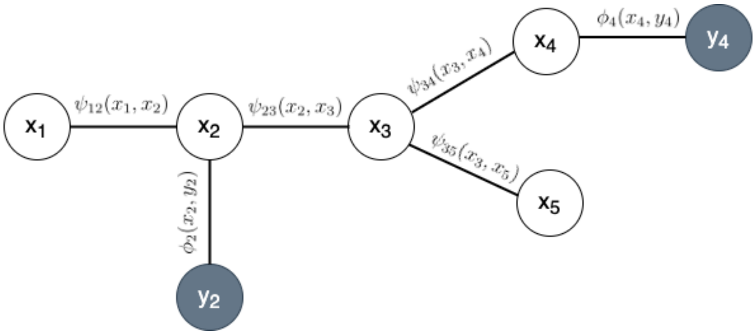


Figure 4: For problem (iii)

$\psi_{12}(x_1, x_2) = \psi_{34}(x_3, x_4) = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix}$, $\psi_{23}(x_2, x_3) = \psi_{35}(x_3, x_5) = \begin{bmatrix} 0.1 & 1 \\ 1 & 0.1 \end{bmatrix}$, $\phi_2(x_2, y_2) = \phi_4(x_4, y_4) = \begin{bmatrix} 1 & 0.1 \\ 0.1 & 1 \end{bmatrix}$.

Write Python code to calculate $b_1(x_1), b_2(x_2), b_3(x_3), b_4(x_4), b_5(x_5)$ by using a library like `numpy`. (*You don't need to write a general BP algorithm—this coding assignment should be simply based on linear algebra, e.g., matrix multiplication, to help you calculate the belief values. It should not take you more than a few lines of code with* `numpy`.) Is the prediction what you expected? [8 points]

What to Submit

1.1:
- Probabilities of node 2, 4 and 6 belonging to class "+".
- Predicted node labels in class "+" and class "-".

1.2:
- Belief at node 1.
- Proof of $b_1(x_1)$ is the same as $p(x_1|y_1, y_2, y_3, y_4)$.
- Five belief values and a brief argument about whether the beliefs are expected (a few lines). Put the code in `q1.py` in the zip file.

2 Node Embeddings with TransE [25 points]

While many real world systems are effectively modeled as graphs, graphs can be a cumbersome format for certain downstream applications, such as machine learning models. It is often useful to represent each node of a graph as a vector in a continuous low dimensional space. The goal is to preserve information about the structure of the graph in the vectors assigned to each node. For instance, the spectral embedding in Homework 1 preserved structure in the sense that nodes connected by an edge were usually close together in the (one-dimensional) embedding x.

Multi-relational graphs are graphs with multiple types of edges. They are incredibly useful for representing structured information, as in knowledge graphs. There may be one node representing "Washington, DC" and another representing "United States", and an edge between them with the type "Is capital of". In order to create an embedding for this type of graph, we need to capture information about not just which edges exist, but what the types of those edges are. In this problem, we will explore a particular algorithm designed to learn node embeddings for multi-relational graphs.

The algorithm we will look at is TransE.[1] We will first introduce some notation used in the paper describing this algorithm. We'll let a multi-relational graph $G = (E, S, L)$ consist of the set of *entities* E (i.e., nodes), a set of edges S, and a set of possible relationships L. The set S consists of triples (h, ℓ, t), where $h \in E$ is the *head* or source-node, $\ell \in L$ is the relationship, and $t \in E$ is the *tail* or destination-node. As a node embedding, TransE tries to learn embeddings of each entity $e \in E$ into $\mathbb{R}^k$ (k-dimensional vectors), which we will notate by $\mathbf{e}$. The main innovation of TransE is that each relationship ℓ is also embedded as a vector $\boldsymbol{\ell} \in \mathbb{R}^k$, such that the difference between the embeddings of entities linked via the relationship ℓ is approximately $\boldsymbol{\ell}$. That is, if $(h, \ell, t) \in S$, TransE tries to ensure that $\mathbf{h} + \boldsymbol{\ell} \approx \mathbf{t}$. Simultanesouly, TransE tries to make sure that $\mathbf{h} + \boldsymbol{\ell} \not\approx \mathbf{t}$ if the edge (h, ℓ, t) does not exist.

Note on notation: we will use unbolded letters e, ℓ, etc. to denote the entities and relationships in the graph, and bold letters $\mathbf{e}, \boldsymbol{\ell}$, etc., to denote their corresponding embeddings.

TransE accomplishes this by minimizing the following loss:

$$\mathcal{L} = \sum_{(h,\ell,t)\in S} \Big(\sum_{(h',\ell,t')\in S'_{(h,\ell,t)}} [\gamma + d(\mathbf{h} + \boldsymbol{\ell}, \mathbf{t}) - d(\mathbf{h'} + \boldsymbol{\ell}, \mathbf{t'})]_+ \Big). \tag{1}$$

Here (h', ℓ, t') are "corrupted" triplets, chosen from the set $S'_{(h,\ell,t)}$ of corruptions of (h, ℓ, t), which are all triples where either h or t (but not both) is replaced by a random entity.

$$S'_{(h,\ell,t)} = \{(h', \ell, t) | h' \in E\} \cup \{(h, \ell, t') | t' \in E\}$$

Additionally, $\gamma > 0$ is a fixed scalar called the *margin*, the function $d(\cdot, \cdot)$ is the Euclidean distance, and $[\cdot]_+$ is the positive part function (defined as $\max(0, \cdot)$). Finally, TransE restricts all the entity embeddings to have length 1: $||\mathbf{e}||_2 = 1$ for every $e \in E$.

For reference, here is the TransE algorithm, as described in the original paper on page 3:

[1] See the 2013 NeurIPS paper by Bordes et al: https://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf

```
input Training set $S = \{(h, \ell, t)\}$, entities and rel. sets $E$ and $L$, margin $\gamma$, embeddings dim. $k$.
 1: initialize $\boldsymbol{\ell} \leftarrow$ uniform$(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each $\ell \in L$
 2:            $\boldsymbol{\ell} \leftarrow \boldsymbol{\ell} / \|\boldsymbol{\ell}\|$ for each $\ell \in L$
 3:            $\mathbf{e} \leftarrow$ uniform$(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each entity $e \in E$
 4: loop
 5:   $\mathbf{e} \leftarrow \mathbf{e} / \|\mathbf{e}\|$ for each entity $e \in E$
 6:   $S_{batch} \leftarrow$sample$(S, b)$ // sample a minibatch of size $b$
 7:   $T_{batch} \leftarrow \emptyset$ // initialize the set of pairs of triplets
 8:   for $(h, \ell, t) \in S_{batch}$ do
 9:     $(h', \ell, t') \leftarrow$sample$(S'_{(h,\ell,t)})$ // sample a corrupted triplet
10:     $T_{batch} \leftarrow T_{batch} \cup \{((h, \ell, t), (h', \ell, t'))\}$
11:   end for
12:   Update embeddings w.r.t. $\sum_{((h,\ell,t),(h',\ell,t')) \in T_{batch}} \nabla[\gamma + d(\boldsymbol{h} + \boldsymbol{\ell}, \boldsymbol{t}) - d(\boldsymbol{h'} + \boldsymbol{\ell}, \boldsymbol{t'})]_+$
13: end loop
```

2.1 Warmup: Why the Comparative Loss? [3 points]

Say we were intent on using a simpler loss function. Our objective function (1) includes a term maximizing the distance between $\mathbf{h}' + \boldsymbol{\ell}$ and $\mathbf{t}'$. If we instead simplified the objective, and just tried to minimize

$$\mathcal{L}_{\text{simple}} = \sum_{(h,\ell,t) \in S} d(\mathbf{h} + \boldsymbol{\ell}, \mathbf{t}), \tag{2}$$

we would obtain a useless embedding. Give an example of a simple graph and corresponding embeddings which will minimize the new objective function (2) all the way to zero, but still give a completely useless embedding. (Note: your graph should be non-trivial, i.e., it should include at least two nodes and at least one edge. Assume the embeddings are in 2 dimensions, i.e., $k = 2$.)

2.2 The Purpose of the Margin [5 points]

We are interested in understanding what the margin term γ accomplishes. If we removed the margin term γ from our loss, and instead optimized

$$\mathcal{L}_{\text{no margin}} = \sum_{(h,\ell,t) \in S} \sum_{(h',\ell,t') \in S'_{(h,\ell,t)}} [d(\mathbf{h} + \boldsymbol{\ell}, \mathbf{t}) - d(\mathbf{h}' + \boldsymbol{\ell}, \mathbf{t}')]_+, \tag{3}$$

it turns out that we would again obtain a useless embedding. Give an example of a simple graph and corresponding embeddings which will minimize the new objective function (3) all the way to zero, but still give a completely useless embedding. By useless, we mean that in your example, you cannot tell just from the embeddings whether two nodes are linked by a particular relation (Note: your graph should be non-trivial, i.e., it should include at least two nodes and at least one edge. Assume the embeddings are in 2 dimensions, i.e., $k = 2$.)

2.3 Why are Entity Embeddings Normalized? [7 points]

Recall that TransE normalizes every entity embedding to have unit length (see line 5 of the algorithm). The quality of our embeddings would be much worse if we did not have this step. To understand why, imagine running the algorithm with line 5 omitted.

What could the algorithm do to trivially minimize the loss in this case? What would the embeddings it generates look like?

2.4 Where TransE fails [10 points]

Give an example of a simple graph for which no perfect embedding exists, i.e., no embedding perfectly satisfies $\mathbf{u} + \boldsymbol{\ell} = \mathbf{v}$ for all $(u, \ell, v) \in S$ and $\mathbf{u} + \boldsymbol{\ell} \neq \mathbf{v}$ for $(u, \ell, v) \notin S$, for any choice of entity embeddings ($\mathbf{e}$ for $e \in E$) and relationship embeddings ($\boldsymbol{\ell}$ for $\ell \in L$). Explain why this graph has no perfect embedding in this system, and what that means about the expressiveness of TransE embeddings. Assume the embeddings are in 2 dimensions ($k = 2$).

What to Submit

2.1:
- Graph and corresponding useless embeddings which minimize (2) to 0.

2.2:
- Graph and corresponding useless embeddings which minimize (3) to 0.

2.3:
- Discussion of behavior of algorithm and resulting embeddings without normalization step.

2.4:
- Graph for which no perfect embedding exists, with justification.
- Discussion on TransE embedding expressiveness.

3 GNN Expressiveness [25 points]

Graph Neural Networks (GNNs) are a class of neural network architectures used for deep learning on graph-structured data. Broadly, GNNs aim to generate high-quality embeddings of nodes by iteratively aggregating feature information from local graph neighborhoods using neural networks; embeddings can then be used for recommendations, classification, link prediction or other downstream tasks. Two important types of GNNs are GCNs (graph convolutional networks) and GraphSAGE (graph sampling and aggregation).

Let $G = (V, E)$ denote a graph with node feature vectors X_u for $u \in V$. To generate the embedding for a node u, we use the neighborhood of the node as the computation graph. At every layer l, for each pair of nodes $u \in V$ and its neighbor $v \in V$, we compute a message function via neural networks, and apply a convolutional operation that aggregates the messages from the node's local graph neighborhood (Figure 5), and updates the node's representation at next layer. By repeating this process through K GNN layers, we capture feature and structural information from the local K-hop neighborhood. For each of the message computation, aggregation and update functions, the learnable parameters are shared across all nodes in the same layer.

We initialize the feature vector of each node X_u based on the data we have available. If we already have outside information about the nodes, we can embed that as a feature vector. Otherwise we can use constant feature (vector of 1) or the degree of the node as the feature vector.

These are the key steps in each layer of a GNN:

- **Message computation:** We use a neural network to learn a message function between nodes. For each pair of nodes u and its neighbor v, the neural network message function can be expressed as $M(h_u^k, h_v^k, e_{u,v})$. In GCN and GraphSAGE, this can simply be $\sigma(Wh_v + b)$, where W and b are the weights are bias of a neural network linear layer. Here h_u^k refers to the hidden representation of node u at layer k, and $e_{u,v}$ denotes available information about the edge (u, v), like the edge weight or other features. For GCN and GraphSAGE, the neighbors of u are simply defined as nodes that are connected to u. However, many other variants of GNNs have different definitions of neighborhood.

- **Aggregation:** At each layer, we apply a function to aggregate information from all of the neighbors of each node. The aggregation function is usually permutation invariant, to reflect the fact that nodes' neighbors have no canonical ordering. In a GCN, the aggregation is done by a weighted sum, where the weight for aggregating from v to u corresponds to the (u, v) entry of the normalized adjacency matrix $D^{-1/2}AD^{-1/2}$.

- **Update:** We update the representation of a node based on the aggregated representation of the neighborhood. For example, in GCNs, a multi-layer perceptron (MLP) is used; GraphSAGE combines a skip layer with the MLP.

- **Pooling:** The representation of an entire graph can be obtained by adding a pooling layer at the end. The simplest pooling methods are just taking the mean, max or sum of all of the individual node representations.[2] This is usually done for the purposes of graph classification[3].

[2] More complex pooling such as DiffPool and Graph CNN (Defferrard *et al.*) can be done at intermediate layers

[3] However, note that some architectures such as graph networks (Battaglia *et al.*) use pooling machanism for node-level tasks as well.

We can formulate the Message computation, Aggregation and Update steps for a GCN as a layer-wise propagation rule given by:

$$h^{k+1} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}h^kW^k\right) \tag{4}$$

where h^k represents the matrix of activations in the k-th layer, $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the normalized adjacency of graph G, W^k is a layer-specific learnable matrix, and σ is a non-linearity function. Dropout and other forms of regularization can also be used.

We provide the pseudo-code for GraphSAGE embedding generation below. This will also be relevant to Question 4.

Algorithm 1: Pseudo-code for forward propagation in GraphSAGE

```
Input  : Graph $G(V, E)$; input features $\{x_v, \forall v \in V\}$; depth $K$; non-linearity $\sigma$;
         weight matrices $\{W^k, \forall k \in [1, K]\}$; neighborhood function $\mathcal{N} : v \to 2^V$;
         aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$
Output: Vector representations $z_v$ for all $v \in V$
$h_v^0 \leftarrow x_v, \forall v \in V$ ;
for $k = 1...K$ do
    for $v \in V$ do
        $h^k_{\mathcal{N}(v)} \leftarrow \text{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ // aggregation
        $h_v^k \leftarrow \sigma\left(W^k \cdot \text{CONCAT}(h_v^{k-1}, h^k_{\mathcal{N}(v)})\right)$ // MLP with skip connection
    $h_v^k \leftarrow h_v^k / \|h_v^k\|_2, \forall v \in V$ // update step
$z_v \leftarrow h_v^K, \forall v \in V$
```

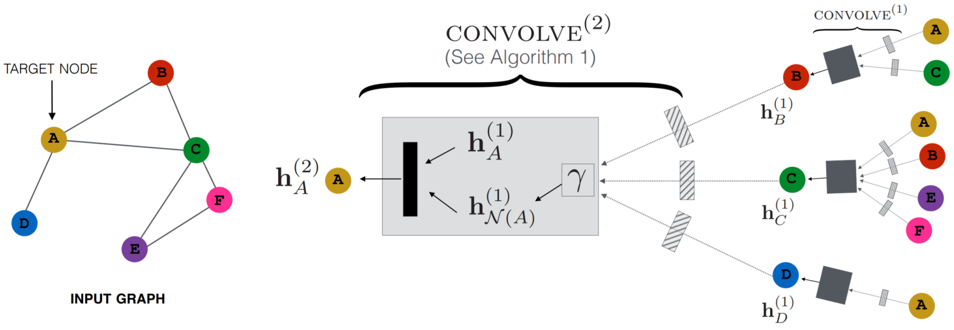
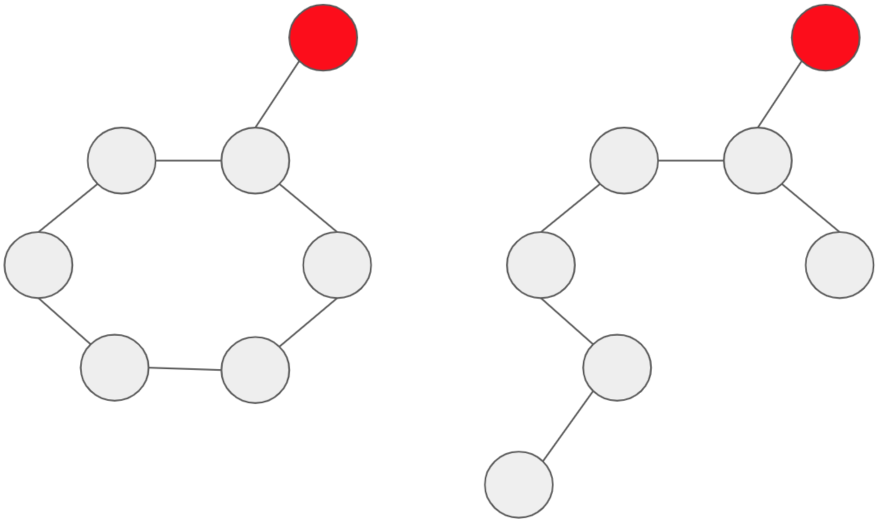


Figure 5: GNN architecture

In this question, we investigate the effect of the number of message passing layers on the expressive power of Graph Convolutional Networks. In neural networks, expressiveness refers to the set of functions (usually the loss function for classification or regression tasks) a neural network is able to compute, which depends on the structural properties of a neural network architecture.

3.1 Effect of Depth on Expressiveness [7 Points]

(i) Consider the following 2 graphs, where all nodes have 1-dimensional feature [1]. We use a simplified version of GNN, with no nonlinearity and linear layers, and sum aggregation. We run the GNN to compute node embeddings for the 2 red nodes respectively. Note that the 2 red nodes have different 4-hop neighborhood structure. How many layers of message passing are needed so that these 2 nodes can be distinguished (i.e., have different GNN embeddings)?



(ii) Consider training a GNN on a node classification task, where nodes are classified to be positive if they are part of an induced cyclic subgraph of length 10, and negative otherwise. Give an example of a graph and the node which should be classified as True. Show that no GNN with fewer than 5 layers can perfectly perform this classification task.

3.2 Relation to Random Walk [7 Points]

(i) Let's explore similarity between message passing and random walks. Let $h_i^{(l)}$ be the embedding of node i in layer l. Suppose that we are using a mean aggregator for message passing, and omit the learned linear transformation and non-linearity: $h_i^{(l+1)} = \frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} h_j^{(l)}$. If we start at a node u, and take a uniform random walk for 1 step, the expectation of the embedding of the node we end on is exactly the embedding of u in the next layer. What is the transition matrix of the random walk? Describe the transition matrix using the adjacency matrix A, and degree matrix D, a diagonal matrix where $D_{i,i}$ is the degree of node i.

(ii) Suppose that we add skip connection in aggregation: $h_i^{(l+1)} = \frac{1}{2}h_i^{(l)} + \frac{1}{2}\frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} h_j^{(l)}$. What is the corresponding transition matrix?

3.3 Over-Smoothing Effect [5 Points]

(i) In Question 3.1 we see that increasing depth could give more expressive power. On the other hand, however, very large depth also gives rise to the undesirable effect of oversmoothing. Assume we are still using the aggregation function from part 3.2 (i): $h_i^{(l+1)} = \frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} h_j^{(l)}$. Show that the node embedding $h_i^{(l)}$ will converge as $l \to \infty$. Here we assume that the graph is connected and has no bipartite components. In practice, learnable weights, non-linearity and other architecture choices may alleviate the over-smoothing effect.

3.4 Learning BFS with GNN [6 Points]

(i) Next we investigate the expressive power of GNN for learning simple graph algorithms. Consider breadth-first search (BFS), where at every step, nodes that are connected to visited nodes become visited. Suppose that we use GNN to learn to execute the BFS algorithm. Suppose that the embeddings are 1-dimensional. Initially, all nodes have input feature 0, except a source node which has input feature 1. At every step, nodes reached by BFS has embedding 1, and nodes not reached by BFS has embedding 0. Write the update rule at every step of BFS execution.

(ii) Describe a message function and an aggregation function for the GNN such that it learns the task perfectly.

What to Submit

3.1:
- Reasoning of why a proposed number of message passing step is enough to distinguish the neighborhoods.
- An example that should be classified to True.
- Reasoning of why GNNs with fewer than 5 layers do not have enough expressive power.

3.2:
- Correct transition matrix, in terms of D and A.
- Correct transition matrix for the case with skip layer.

3.3:
- Proof that embedding converges, using Markov chain.

3.4:
- Update rule: the function that determines whether a node is reachable from source at step t, given its previous reachability and its neighbors' previous reachability.
- aggregation method (anything containing sum+clamp; max; min) all acceptable.

4 GNN training [25 Points]

Attention mechanisms have become the state-of-the-art in many sequence-based tasks such as machine translation and learning sentence representations. One of the major benefits of attention-based mechanisms is their ability to focus on the most relevant parts of the input to make decisions. In this problem, we will see how attention mechanisms can be used to perform node classification of graph-structured data through the usage of Graph Attention Networks (GATs).

The building block of the Graph Attention Network is the graph attention layer, which is a variant of the aggregation function (see step 2 in Question 3). Let N be the number of nodes and F be the dimension of the feature vector for each node. The input to each graph attentional layer is a set of node features: $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, \dots, \overrightarrow{h_N}\}$, $\overrightarrow{h_i} \in R^F$. The output of each graph attentional layer is a new set of node features, which may have a new dimension F': $\mathbf{h}' = \{\overrightarrow{h'_1}, \overrightarrow{h'_2}, \dots, \overrightarrow{h'_N}\}$, with $\overrightarrow{h'_i} \in \mathbb{R}^{F'}$.

We will now describe this transformation of the input features into higher-level features performed by each graph attention layer. First, a shared linear transformation parametrized by the weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is applied to every node. Next, we perform self-attention on the nodes. We use a shared attentional mechanism:

$$a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}. \tag{5}$$

This mechanism computes the attention coefficients that capture the importance of node j's features to node i:

$$e_{ij} = a(\mathbf{W}\overrightarrow{h_i}, \mathbf{W}\overrightarrow{h_j}) \tag{6}$$

The most general formulation of self-attention allows every node to attend to all other nodes which drops all structural information. To utilize graph structure in the attention mechanisms, we can use masked attention. In masked attention, we only compute e_{ij} for nodes $j \in \mathcal{N}_i$ where $\mathcal{N}_i$ is some neighborhood of node i in the graph.

To easily compare coefficients across different nodes, we normalize the coefficients across j using a softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

For this problem, our attention mechanism a will be a single-layer feedforward neural network parametrized by a weight vector $\overrightarrow{a} \in \mathbb{R}^{2F'}$, followed by a LeakyReLU nonlinearity (with negative input slope 0.2). Let $\cdot^T$ represent transposition and $||$ represent concatenation. The coefficients computed by our attention mechanism may be expressed as:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{a}^T[\mathbf{W}\overrightarrow{h_i}||\mathbf{W}\overrightarrow{h_j}]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\overrightarrow{a}^T[\mathbf{W}\overrightarrow{h_i}||\mathbf{W}\overrightarrow{h_k}]\right)\right)} \tag{7}$$

Now, we use the normalized attention coefficients to compute a linear combination of the features corresponding to them. These aggregated features will serve as the final output features for every node.

$$h_i' = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\overrightarrow{h_j}. \tag{8}$$

To stabilize the learning process of self-attention, we use multi-head attention. To do this we use K independent attention mechanisms, or "heads" compute output features as in Equation 8. Then, we concatenate these output feature representations:

$$\overrightarrow{h_i}' = ||_{k=1}^{K}\Big(\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{(k)}\mathbf{W}^{(k)}\overrightarrow{h_j}\Big) \tag{9}$$

where $||$ is concentation, $\alpha_{ij}^{(k)}$ are the normalized attention coefficients computed by the k-th attention mechanism (a^k), and $\mathbf{W}^{(k)}$ is the corresponding input linear transformation's weight matrix. Note that for this setting, $\mathbf{h}' \in \mathbb{R}^{KF'}$.

In this problem, we will be implementing a general Graph Neural Network Stack, as well as GraphSAGE and GATs. We will use our implementations to complete graph and node classification on two benchmarks:

- CORA is a standard citation network benchmark dataset. In this dataset, nodes correspond to documents and edges correspond to undirected citations. Each node has a class label. The node features are elements of a bag-or-words representation of a document. For the Cora dataset, there are 2708 nodes, 5429 edges, 7 prediction classes for nodes, and 1433 features per node. We will be performing node classification on CORA.
- ENZYMES is a relatively small graph classification benchmark, containing graphs representing the protein tertiary structures from the BRENDA enzyme database. The task is to correctly assign each enzyme to one of the 6 EC top-level classes. It has 600 graphs, classified into 6 classes. The average number of nodes 32.63; average number of edges 62.14. Every node has attribute with 18 dimensions. We will be performing graph classification on ENZYMES.

For the questions below, you will need to modify the files `train.py` and `models.py`. We've left `TODO` blocks in `model.py` indicating what you need to modify. To implement GraphSage and GAT, we will be extending the `MessagePassing` base class of PyTorch geometric. You may find the `MessagePassing` documentation found here to be useful. In this documentation, you will find an example implementation of GCNs by extending the `MessagePassing` base class. We will be doing a similar extension for the implementations of `GraphSage` and `GAT`.

Before starting the implementation, we recommend reading over `train.py` and `models.py` and understanding the structure of the code. In `train.py`, take a look at the training and testing loops and what we do differently for node and graph classification.

4.1 Examine Dataset [1 Point]

(i) *Warm-up.* First let's examine the data. To do this, you will need use `train.py`. How many nodes are there in the test set of CORA? How many graphs are there in the test set of ENZYMES?

4.2 GNN Implementation [15 Points]

(i) **Coding.** Now we will implement the GNN Stack. Complete the `forward` method for `GNNStack` in `models.py`. For the purpose of this exercise, each layer of the GNN stack should contain a CONV → RELU → DROPOUT structure, where the CONV layer is specified by the model type. This forward call will be used for both the node classification task and graph classification task later.

(ii) **Coding.** Complete the `GraphSage` implementation in `models.py` by filling in the `__init__`, `forward`, and `message` methods. Use Algorithm 1 given in Question 3 for reference. For the AGGREGATE function, we will be using a dense layer followed by a RELU non-linearity, and a mean aggregator.

(iii) **Coding.** Complete the `GAT` implementation by filling in the `__init__`, `forward`, and `message` methods. In `__init__` we will need to define the layers we need for the attention mechanism and for aggregating the final features. In `forward`, we will apply a linear transformation to the node feature matrix before starting message propagation. For `message`, we will implement GAT message passing following the equation 7.

4.3 Training [9 Points]

(i) **Coding.** Run GNN training for node classification task on the CORA dataset, and graph classification task on the ENZYMES dataset. Do this for GCN, GraphSage, and GAT. You will be using the Torch Geometric implementation of GCN, and your implementations of GraphSage and GAT. You can specify the model type to use in training by the flag –model_type of `train.py`. Write code to plot the validation accuracy over number of epochs. For each dataset, please plot the validation accuracy vs epochs for each of the models.

(ii) Describe the performance differences between GCN, GraphSage, and GAT on both tasks. Which model performed best for each task?

What to Submit

4.1:
- Number of nodes in CORA.
- Number of graphs in ENZYMES.

4.2:
- Submit `models.py` and `train.py` in the code zip file to HW2 (Code) on Gradescope.

4.3:
- Two plots, one for each dataset. Each plot will be of validation accuracy vs epoch for each of the three models.
- Description of the performance differences between the three models. For each task, which model performed the best?