GraphSAGE

Inductive Representation Learning on Large Graphs

William L. Hamilton, Rex Ying and Jure Leskovec

NIPS 2017

Paper Review

Jihwan Oh

2023.02.07

Contents

Introduction

1. Limitation of previous tasks

Low-dimensional vector embeddings is useful for graph prediction and analysis.

However, it has focused on embedding nodes from a **single fixed graph (Transductive)**

Real-world applications require embeddings to be quickly generated for unseen nodes,
or entirely new (sub)graphs

Therefore, we should consider the **generalization** to unseen node. **(Inductive)**

Introduction

2. Transductive vs Inductive Learning

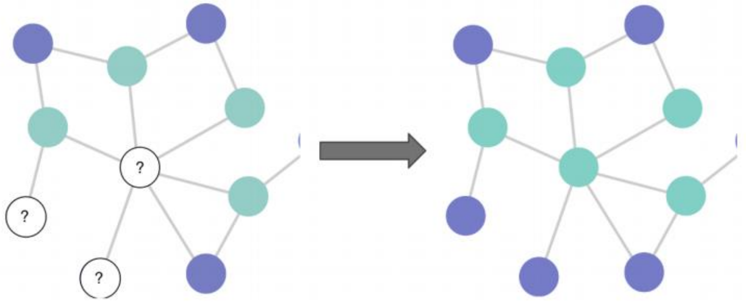
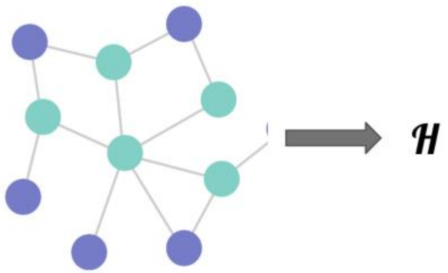
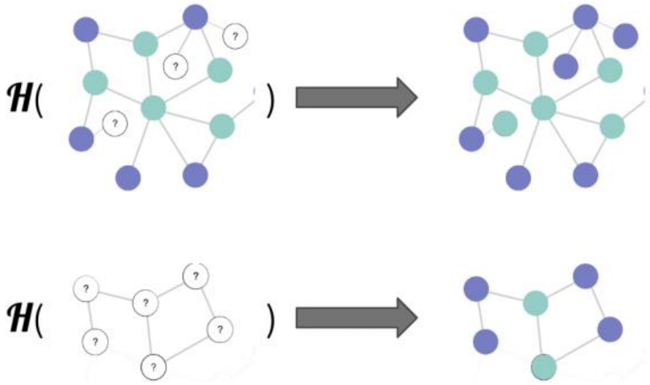


Figure 1. Node classification in transductive setting. At training time, the learning algorithm has access to all the nodes and edges including nodes for which labels are to be predicted.

Transductive Learning



(a) A model $\mathcal{H}$ is learned over some graph



(b) The model is then by applied to new nodes and edges

Figure 2. Node classification in inductive setting. Once learned, the model can be applied to new unseen nodes (outlined in red). There may or may not exist edges between such new nodes and the nodes used for training.

Inductive Learning

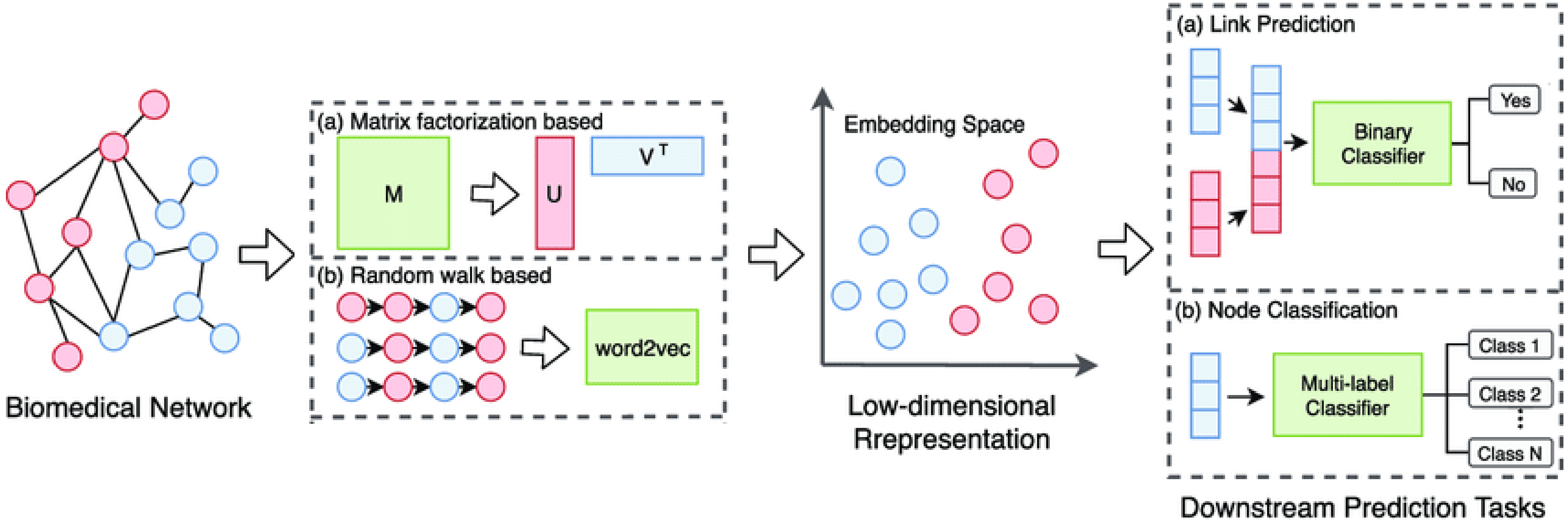
Related Works

1. Factorization-based embedding approaches

Low-dimensional embeddings using random walk statistics and matrix factorization-based learning objectives

- Directly train node embeddings for individual nodes (Transductive)

- Require expensive additional training to make predictions on new nodes

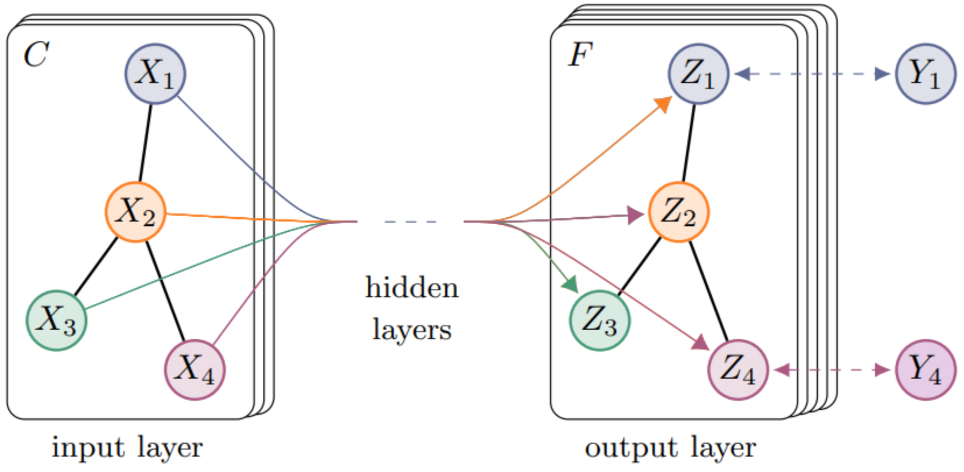


Related Works

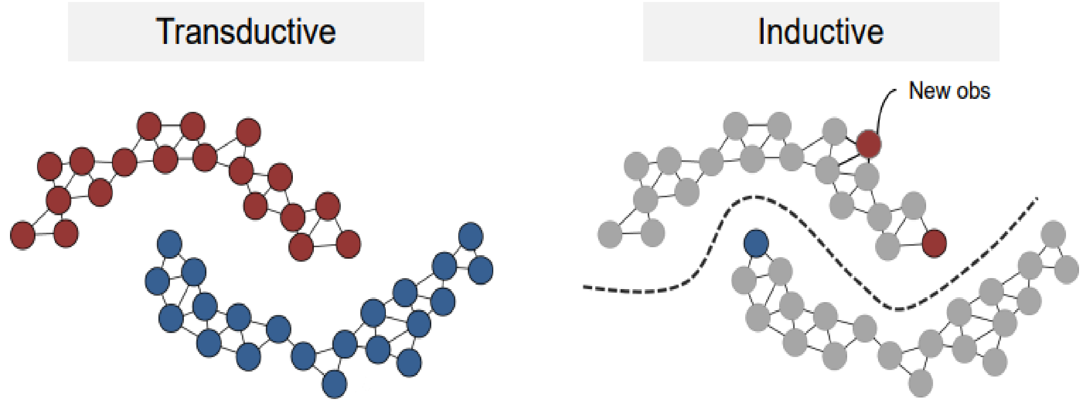
2. Graph convolutional networks (GCN)

Graph convolutional networks (GCNs) have only been applied in the transductive setting with fixed graphs

GraphSAGE can be viewed as an extension of the GCN framework to the inductive setting



(a) Graph Convolutional Network



GraphSAGE

1. Proposed model: GraphSAGE

GraphSAGE: SAmple and aggreGatE

- Generalized embedding (Inductive)
- Using **aggregate function** from neighbor nodes
- Both applied to supervised and unsupervised learning
- Both learn distribution and topological structure in neighbor nodes.
- Low computational cost
- Better performance than previous tasks (DeepWalk, GCN, etc.)

GraphSAGE

2. Embedding generation algorithm

Assume: Model already trained = Parameter fixed

Parameter = A**GGREGATE**$_{\mathbf{K}}$: aggregator function, $\mathbf{W^k}$: weight matrix

---

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 $\mathcal{N} : v \to 2^{\mathcal{V}}$

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

$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

GraphSAGE

2. Embedding generation 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 $v \in \mathcal{V}$

```
h_v^0 ← x_v, ∀v ∈ V ;
for k = 1...K do
    for v ∈ V do
        h_N(v)^k ← AGGREGATE_k({h_u^(k-1), ∀u ∈ N(v)});
        h_v^k ← σ(W^k · CONCAT(h_v^(k-1), h_N(v)^k))
    end
    h_v^k ← h_v^k / ||h_v^k||_2, ∀v ∈ V
end
z_v ← h_v^K, ∀v ∈ 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
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

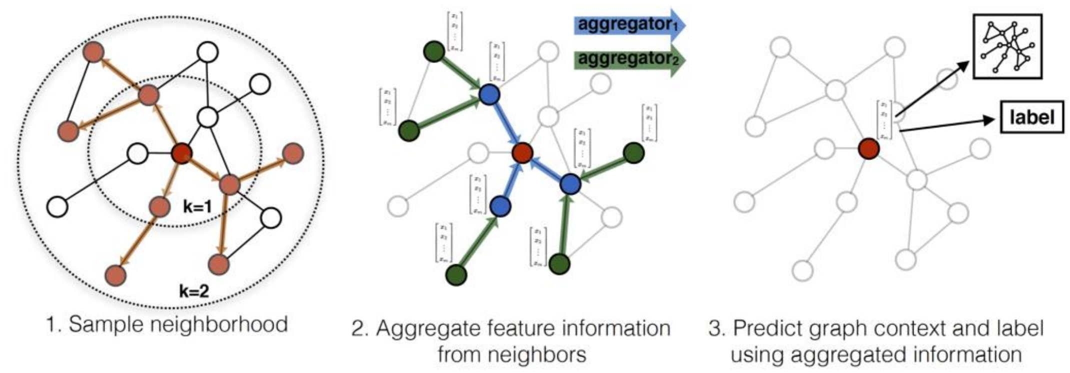


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

GraphSAGE

2. Embedding generation algorithm (mini batch)

Algorithm 2: GraphSAGE minibatch forward propagation algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$;
input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
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 sampling functions, $\mathcal{N}_k : v \to 2^{\mathcal{V}}, \forall k \in \{1, ..., K\}$

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

```
B^K ← B;
for k = K...1 do
    B^{k-1} ← B^k ;
    for u ∈ B^k do
        B^{k-1} ← B^{k-1} ∪ N_k(u);
    end
end
h_u^0 ← x_v, ∀v ∈ B^0 ;
for k = 1...K do
    for u ∈ B^k do
        h_{N(u)}^k ← AGGREGATE_k({h_{u'}^{k-1}, ∀u' ∈ N_k(u)});
        h_u^k ← σ(W^k · CONCAT(h_u^{k-1}, h_{N(u)}^k));
        h_u^k ← h_u^k / ||h_u^k||_2;
    end
end
z_u ← h_u^K, ∀u ∈ B
```

Make each mini batch $\boldsymbol{B^k}$, k $\in \{1, \ldots, K\}$

Identical to previous psudocode

GraphSAGE

3. Neighborhood Definition

Use a fixed-size set of neighbors in order to keep computational cost.

- Without this sampling, $O(V)$ $\rightarrow$ High computational cost

- With this sampling, $O(\prod_{i=1}^{k} S_i)$, S_i is the size of neighborhood set for i $\in \{1, \ldots, K\}$

- ***K*=2, $\boldsymbol{S_1 \times S_2 \leq 500}$** is working well.

- Low computational cost

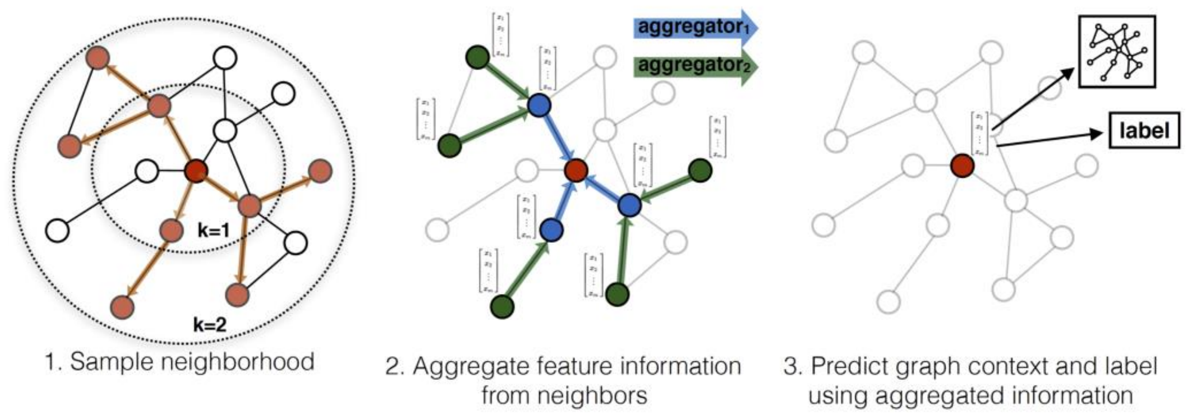


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

GraphSAGE

4. Aggregator Architectures

In Graph, nodes and its neighbors have no ordering.

→ Aggregator should be symmetric. (Permutation invariant)

Symmetric, Trainable, High representational capacity

- Mean Aggregator
- LSTM Aggregator
- Pooling Aggregator

GraphSAGE

4. Aggregator Architectures

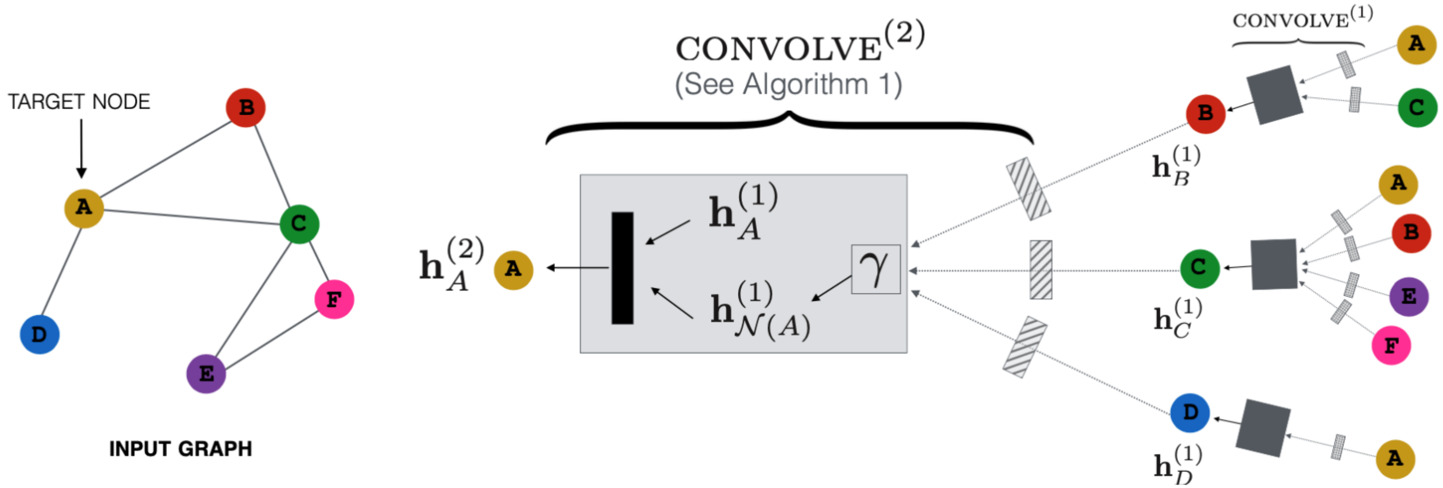
In Graph, nodes and its neighbors have no ordering.

→ Aggregator should be symmetric. (Permutation invariant)

Symmetric, Trainable, High representational capacity

- Mean Aggregator
- LSTM Aggregator
- Pooling Aggregator

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



GraphSAGE

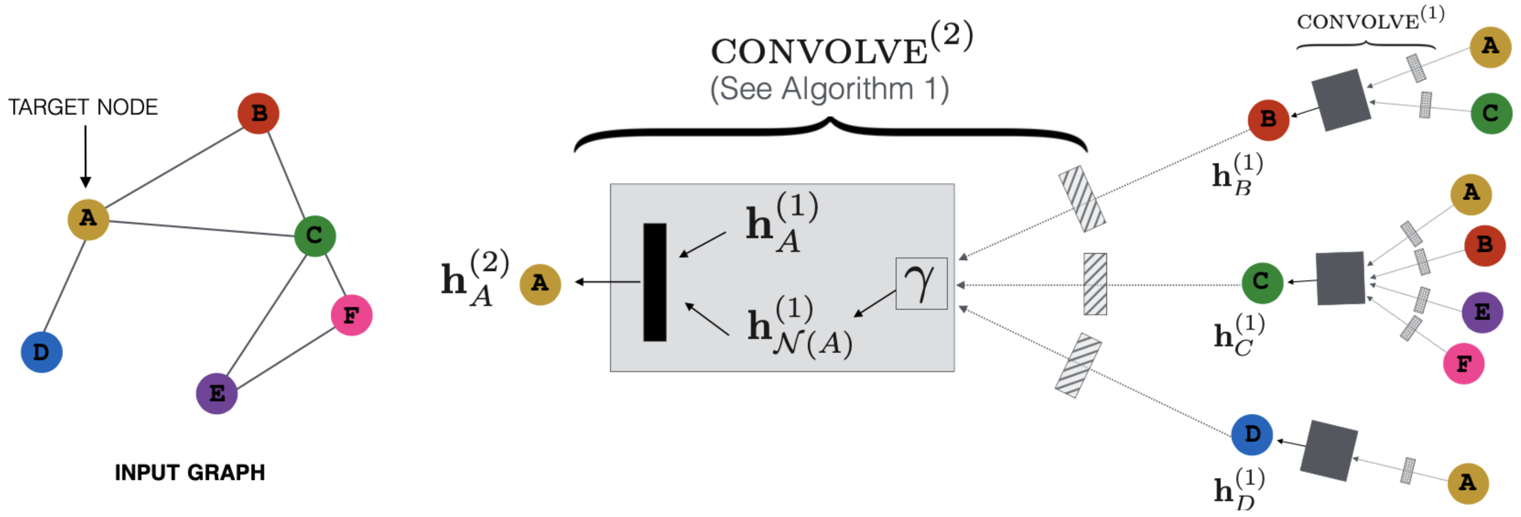
4. Aggregator Architectures

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
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
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

(1) **Mean** aggregator

- Similar to GCN, but it use Concatenating instead of Adding.

$$\text{AGGREGATE}_\text{K} : \ \mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$$



GraphSAGE

4. Aggregator Architectures

(2) **LSTM** aggregator

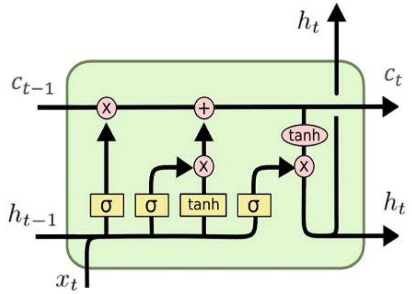
LSTM: Long Short-Term Memory

- Variant of RNN (Better performance than original RNN)

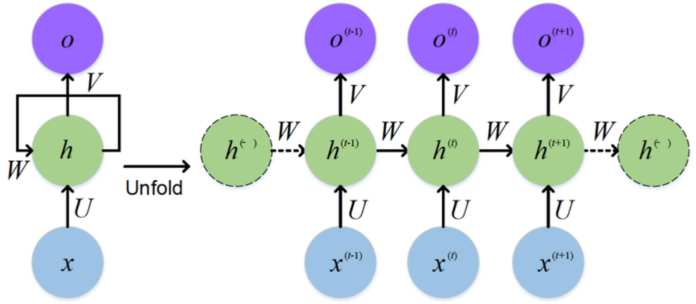
- Advantage of **larger expressive capability**

- It is not permutation invariant since their inputs are sequential.

→ Applying the LSTMs to a **random permutation** of the node’s neighbors (**Permutation invariant**)



LSTM
(Long-Short Term Memory)

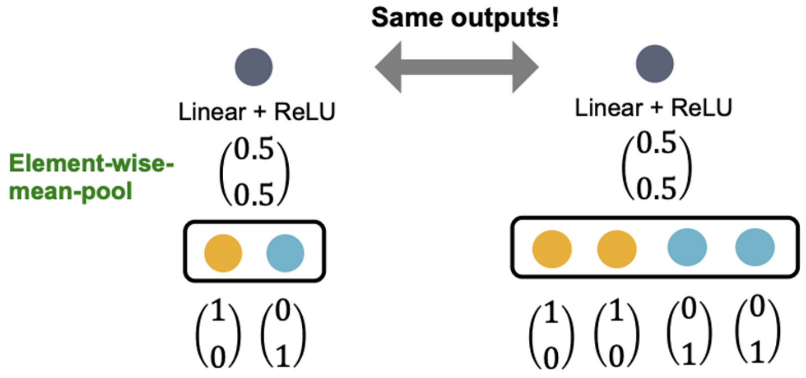


4. Aggregator Architectures

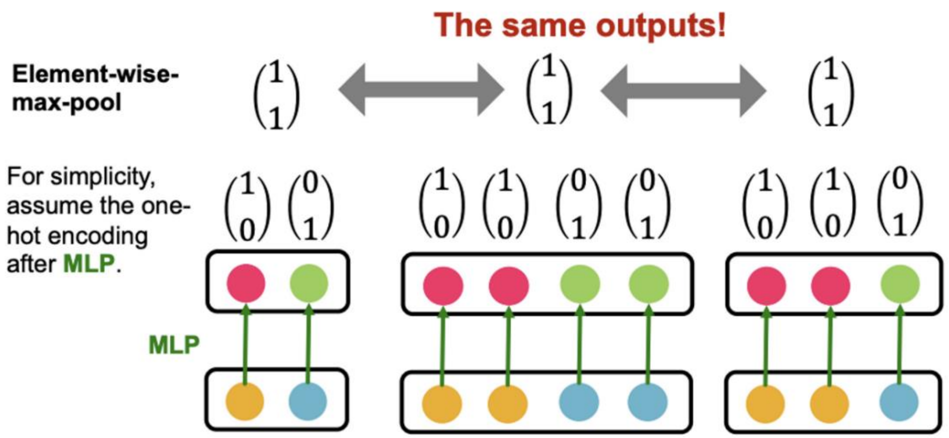
(3) **Pooling** aggregator

- Each neighbor's vector is independently fed through MLP, then do element-wise max pooling.

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\})$$



GCN : Mean pooling



GraphSAGE : Max pooling

GraphSAGE

5. Learning the parameters of GraphSAGE

Unsupervised Learning : Graph-based loss

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right)$$

- v : Node that co-occur near u on fixed-length random walk
- z_u : Output representation by algorithm
- P_n : Negative sampling distribution
- v_n : Negative samples
- Q : Number of negative samples

Supervised Learning : Cross-Entropy loss

GraphSAGE

5. Learning the parameters of GraphSAGE

Unsupervised Learning : Graph-based loss

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right)$$

- v : Node that co-occur near u on fixed-length random walk
- z_u : Output representation by algorithm
- P_n : Negative sampling distribution
- v_n : Negative samples
- Q : Number of negative samples

Supervised Learning : Cross-Entropy loss

Train **Aggregator function** and **Weight matrix** with SGD

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

Experiment

1. Models and Loss function

Models

- **GraphSAGE : GCN, Mean, LSTM, Max polling** (K=2, $S_1 \times S_2 \leq 500$)
- Random classifier
- Logistic Regression feature-based classifier (Ignore graph structure)
- DeepWalk
- DeepWalk + Raw features (Logistic Regression)

Loss function

- Supervised learning: Cross-entropy loss
- Unsupervised learning: Graph-based loss

Experiment

2. Dataset

Citation Data

- Predicting paper subject categories on a large citation dataset
- Train : Test = 8:2 (approximate)

Reddit Data

- Predict which community different Reddit posts belong to
- Train : Test = 8:2 (approximate)

PPI (Protein-Protein Interaction) Data

- Predict protein-protein interactions
- For multi-graph generalization

Experiment

3. Results

1. GraphSAGE performed better than benchmark tasks
2. Performance : LSTM, Max pooling > mean > GCN

*Environment: Non-linear activation function: **ReLU**, Optimizer: **Adam**

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

Experiment

3. Results

A: GraphSAGE is faster than benchmark tasks when training data
B: Neighborhood sample size and accuracy (K=2, $S_1 = S_2$)

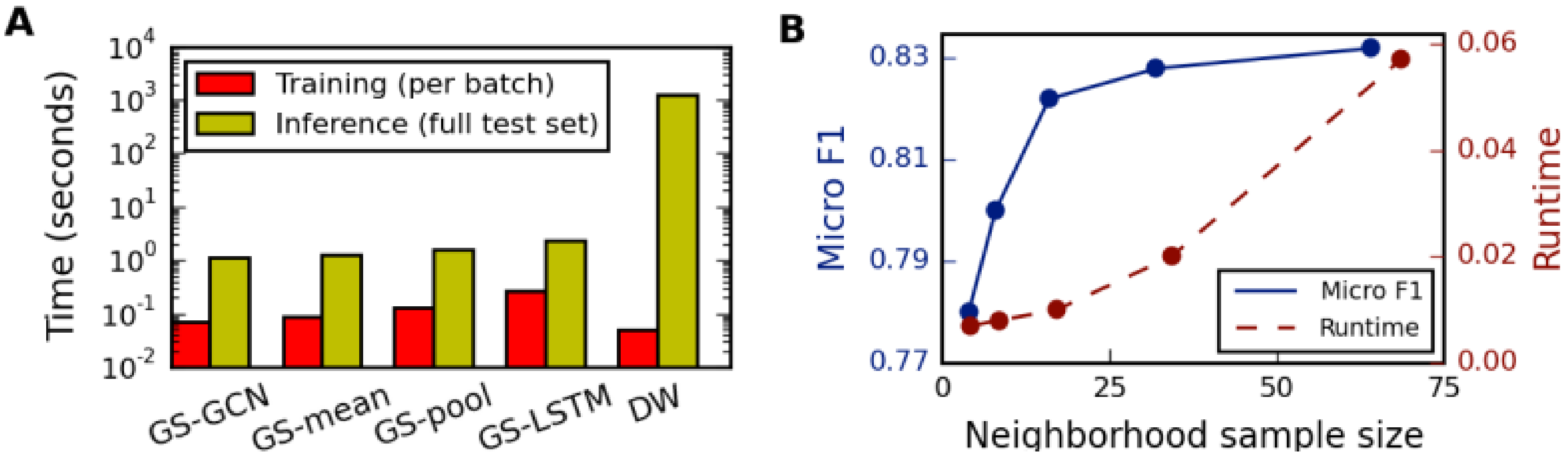


Figure 2: **A**: Timing experiments on Reddit data, with training batches of size 512 and inference on the full test set (79,534 nodes). **B**: Model performance with respect to the size of the sampled neighborhood, where the "neighborhood sample size" refers to the number of neighbors sampled at each depth for $K = 2$ with $S_1 = S_2$ (on the citation data using GraphSAGE-mean).

Conclusion

GraphSAGE

1. GraphSAGE is efficient algorithm for generating embeddings from unseen nodes.
   (Inductive Learning)

2. Effectively trade off performance and runtime in large graphs.

3. A number of extensions and potential improvements are possible,
   such as extending GraphSAGE to incorporate directed or multi-modal graphs.

Implement

Environment Setting

```
import torch
import dgl
from dgl.data import CoraGraphDataset #Data 1
from dgl.data import RedditDataset #Data 2
#from dgl.data import PPIDataset #Data 3 (Not working)

from dgl.nn import SAGEConv

import matplotlib.pyplot as plt
import numpy as np

import networkx as nx
from torch.nn.parameter import Parameter
from torch.nn.modules.module import Module

import scipy
import scipy.sparse as sp
import torch.nn as nn
import torch.nn.functional as F
```

→ **Deep graph library**

→ **Data Loading**

→ **GraphSAGE layer**

$$h_{\mathcal{N}(i)}^{(l+1)} = \text{aggregate}\left(\{h_j^l, \forall j \in \mathcal{N}(i)\}\right)$$

$$h_i^{(l+1)} = \sigma\left(W \cdot \text{concat}(h_i^l, h_{\mathcal{N}(i)}^{l+1})\right)$$

$$h_i^{(l+1)} = \text{norm}(h_i^{(l+1)})$$

Environment Setting

```
[6]  1 torch.__version__

'1.13.1+cu116'
```

→ **Check pytorch version**

```
[1]  1 pip install --pre dgl-cu116 -f https://data.dgl.ai/wheels-test/repo.html

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Looking in links: https://data.dgl.ai/wheels-test/repo.html
Collecting dgl-cu116
  Downloading https://data.dgl.ai/wheels-test/dgl_cu116-1.0a230116-cp38-cp38-manylinux1_x86_64.whl (265.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 265.6/265.6 MB 4.1 MB/s eta 0:00:00
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.8/dist-packages (from dgl-cu116) (2.25.1)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.8/dist-packages (from dgl-cu116) (1.21.6)
Collecting psutil>=5.8.0
  Downloading psutil-5.9.4-cp36-abi3-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (280 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 280.2/280.2 KB 6.0 MB/s eta 0:00:00
```

→ **Download dgl cuda**

Implement

Dataset

```
[6]  1 dataset1 = CoraGraphDataset(verbose=True)

Downloading /root/.dgl/cora_v2.zip from https://data.dgl.ai/dataset/cora_v2.zip...
Extracting file to /root/.dgl/cora_v2
Finished data loading and preprocessing.
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done saving data into cached files.
```

Cora Dataset

```
[7]  1 dataset2 = RedditDataset(self_loop=False, verbose=True)

Downloading /root/.dgl/reddit.zip from https://data.dgl.ai/dataset/reddit.zip...
Extracting file to /root/.dgl/reddit
Finished data loading.
  NumNodes: 232965
  NumEdges: 114615892
  NumFeats: 602
  NumClasses: 41
  NumTrainingSamples: 153431
  NumValidationSamples: 23831
  NumTestSamples: 55703
Done saving data into cached files.
```

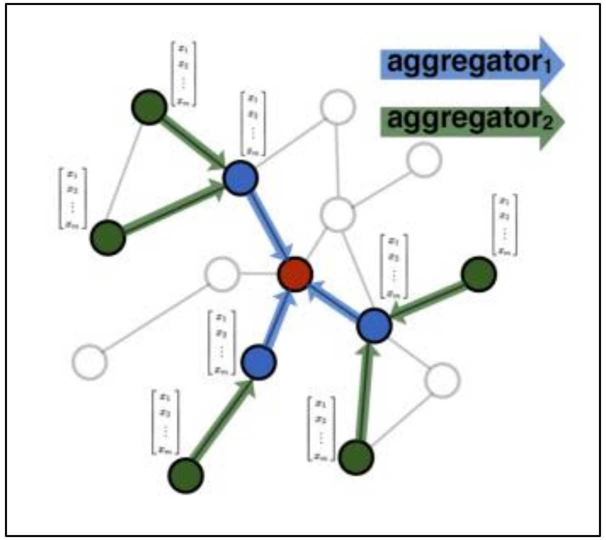
Reddit Dataset

Implement

Model : GraphSAGE

```python
class SAGE(nn.Module):
    def __init__(self, in_size, hid_size, out_size):
        super().__init__()
        self.layers = nn.ModuleList()
        # two-layer GraphSAGE-gcn,mean,pool,lstm
        self.layers.append(SAGEConv(in_size, hid_size, "gcn"))
        self.layers.append(SAGEConv(hid_size, out_size, "gcn"))
        self.dropout = nn.Dropout(0.5)

    def forward(self, graph, x):
        h = x
        for l, layer in enumerate(self.layers):
            h = layer(graph, h)
            if l != len(self.layers) - 1:
                h = F.relu(h)
                h = self.dropout(h)
        return h
```

Two layers(K=2)



Implement

Evaluate function : F1-micro

```python
# f1-micro
def evaluate(g, features, labels, mask, model):
    model.eval()
    with torch.no_grad():
        y_actual = labels
        y_pred = model(g, features)
        y_pred = y_pred[mask]
        y_actual = y_actual[mask]
        _, indices = torch.max(y_pred, dim=1)
        correct = torch.sum(indices == y_actual)
        incorrect = torch.sum(indices != y_actual)
        #f1-micro = TP/(TP*0.5(FP+FN))
        return correct.item() / (correct.item() + 0.5 * incorrect.item())
```

Implement

Train
(Full batch)

```
def train(g, features, labels, masks, model):
    # define train/val samples, loss function and optimizer
    train_mask, val_mask = masks
    loss_fcn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=5e-4)

    y_actual = labels
    accuracy_list = []
    loss_list = []
    # training loop
    for epoch in range(200):
        model.train()
        y_pred = model(g, features)
        loss = loss_fcn(y_pred[train_mask], y_actual[train_mask])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        acc = evaluate(g, features, y_actual, val_mask, model)

        accuracy_list.append(acc)
        loss_list.append(loss.item())

        print(
            "Epoch {:05d} | Loss {:.4f} | Accuracy {:.4f} ".format(
                epoch, loss.item(), acc
            )
        )
    return accuracy_list, loss_list
```

GraphSAGE

Embedding generation algorithm (mini batch)

Algorithm 2: GraphSAGE minibatch forward propagation algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$;
input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
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 sampling functions, $\mathcal{N}_k : v \to 2^{\mathcal{V}}, \forall k \in \{1, ..., K\}$

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

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

Make each mini batch $\boldsymbol{B^k}$, k $\in \{1, \ldots, K\}$

Identical to previous psudocode

Implement

Define Neighborhood Sampler

```python
from dgl.dataloading import DataLoader, NeighborSampler

sampler = NeighborSampler([5,8])
dataloader1 = DataLoader(
        dataset1[0], torch.arange(dataset1[0].num_nodes()).to(dataset1[0].device), sampler, device=device,
        batch_size=256, shuffle=False, drop_last=False,
        num_workers=0)
```

→ **Neighborhood sampler (S_1, S_2)**

Original Graph

Node index

Sampler type

[13]

```python
for it, (input_nodes, output_nodes, blocks) in enumerate(dataloader1):
    print(blocks)
```

```
[Block(num_src_nodes=1779, num_dst_nodes=960, num_edges=4261), Block(num_src_nodes=960, num_dst_nodes=256, num_edges=1053)]
[Block(num_src_nodes=1814, num_dst_nodes=978, num_edges=4349), Block(num_src_nodes=978, num_dst_nodes=256, num_edges=1059)]
[Block(num_src_nodes=1801, num_dst_nodes=935, num_edges=4195), Block(num_src_nodes=935, num_dst_nodes=256, num_edges=996)]
[Block(num_src_nodes=1783, num_dst_nodes=882, num_edges=3922), Block(num_src_nodes=882, num_dst_nodes=256, num_edges=882)]
[Block(num_src_nodes=1758, num_dst_nodes=904, num_edges=4042), Block(num_src_nodes=904, num_dst_nodes=256, num_edges=986)]
[Block(num_src_nodes=1802, num_dst_nodes=951, num_edges=4164), Block(num_src_nodes=951, num_dst_nodes=256, num_edges=1041)]
[Block(num_src_nodes=1747, num_dst_nodes=933, num_edges=4218), Block(num_src_nodes=933, num_dst_nodes=256, num_edges=1182)]
[Block(num_src_nodes=1544, num_dst_nodes=841, num_edges=3753), Block(num_src_nodes=841, num_dst_nodes=256, num_edges=1209)]
[Block(num_src_nodes=1494, num_dst_nodes=797, num_edges=3616), Block(num_src_nodes=797, num_dst_nodes=256, num_edges=996)]
[Block(num_src_nodes=1419, num_dst_nodes=756, num_edges=3065), Block(num_src_nodes=756, num_dst_nodes=256, num_edges=783)]
[Block(num_src_nodes=632, num_dst_nodes=355, num_edges=1061), Block(num_src_nodes=355, num_dst_nodes=148, num_edges=301)]
```

Implement

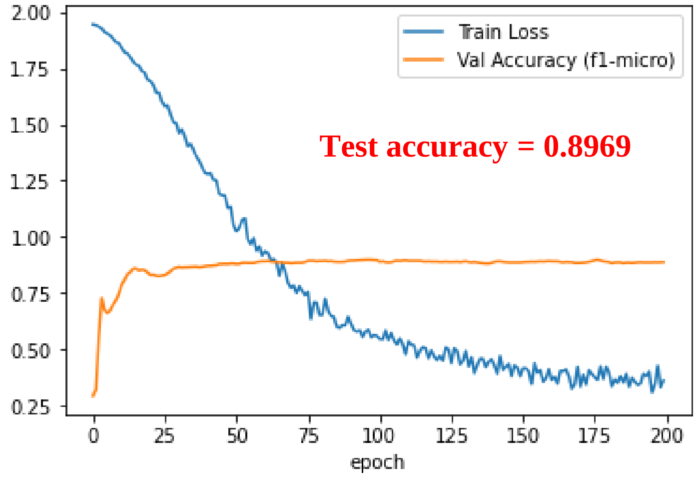
Train (mini batch)

```python
def train(model, dataloader):
    # define train/val samples, loss function and optimizer
    loss_fcn = nn.CrossEntropyLoss()

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=5e-4)

    train_loss_list = []
    val_acc_list = []

    # training loop
    for epoch in range(200):
        model.train()
        val_acc = 0
        train_loss = 0

        for input_nodes, output_nodes, batch_graphs in dataloader:

            features = batch_graphs[0].srcdata['feat']
            labels = batch_graphs[1].dstdata['label']
            train_mask, val_mask = batch_graphs[1].dstdata["train_mask"], batch_graphs[1].dstdata["val_mask"]
            #in_size = features.shape[1] = 1433 # number of features
            #out_size = dataset1.num_classes  = 7 # number of classes

            logits = model(batch_graphs, features)
            loss = loss_fcn(logits[train_mask], labels[train_mask])
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

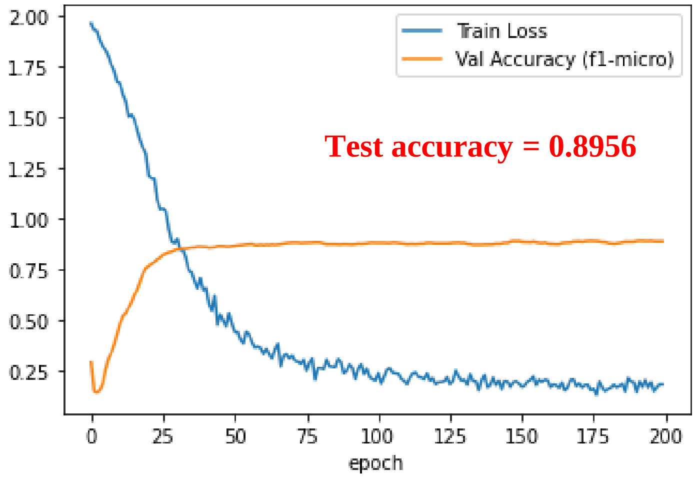
→ **Batch training**

Implement

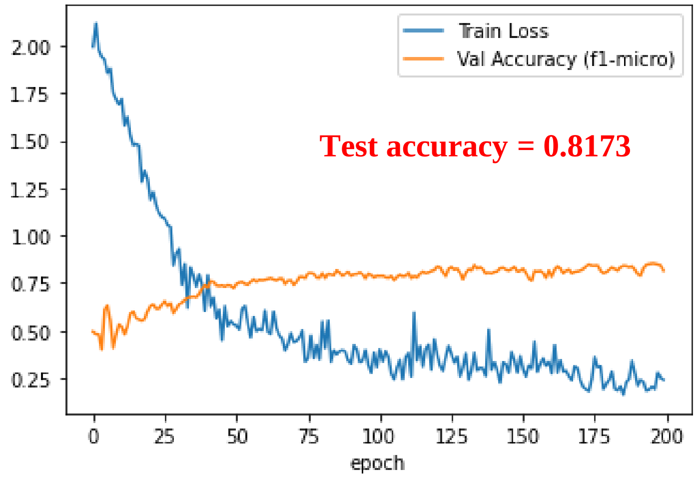
Cora Citation Dataset Result (full batch)



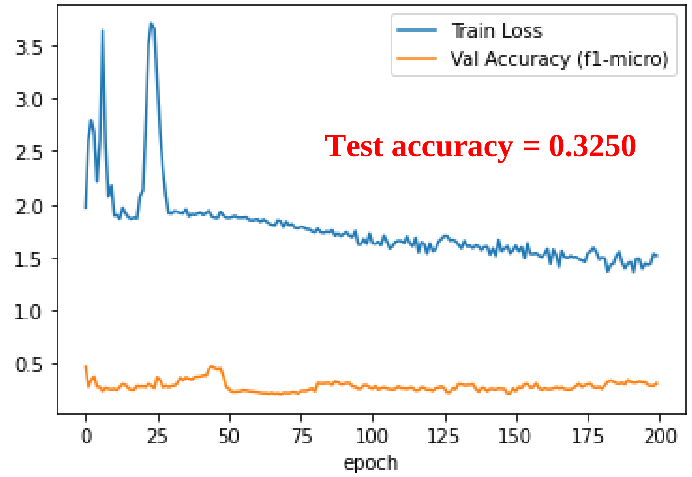
GraphSAGE-GCN



GraphSAGE-mean



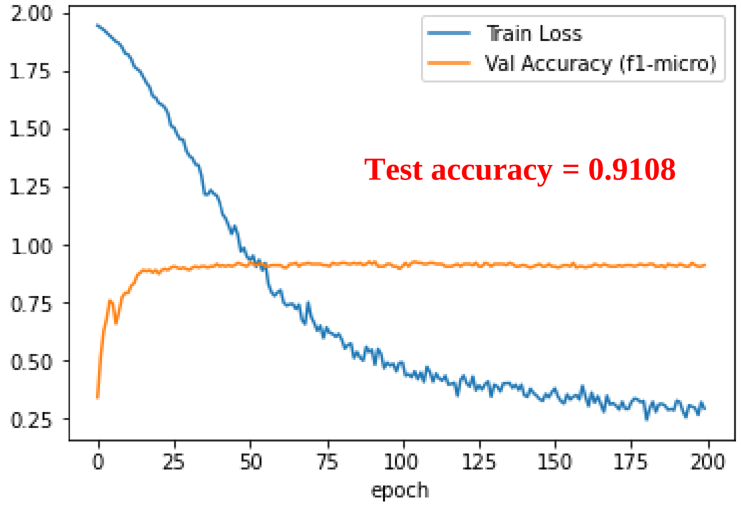
GraphSAGE-max pooling



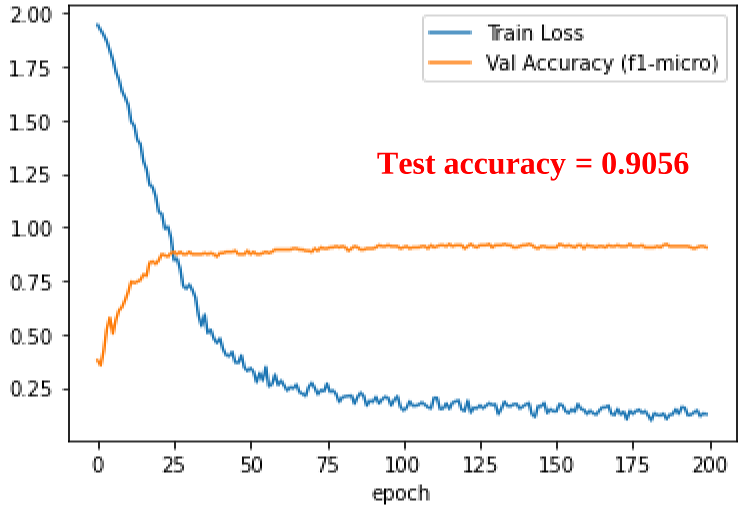
GraphSAGE-LSTM

Implement

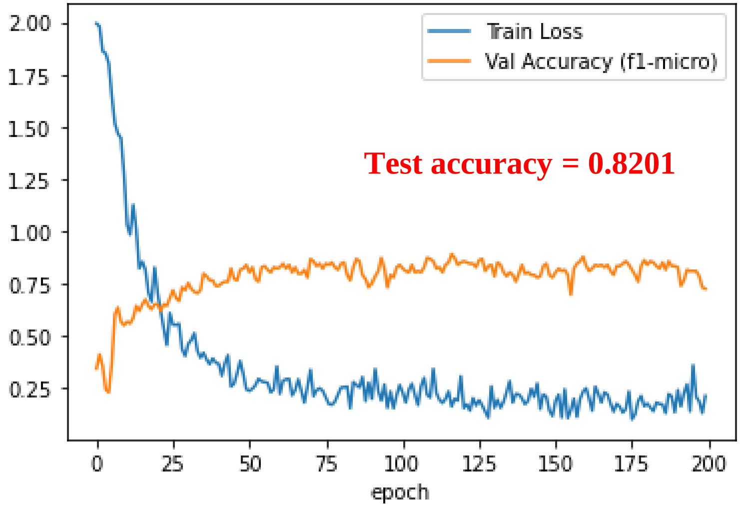
Cora Citation Dataset Result (mini batch)



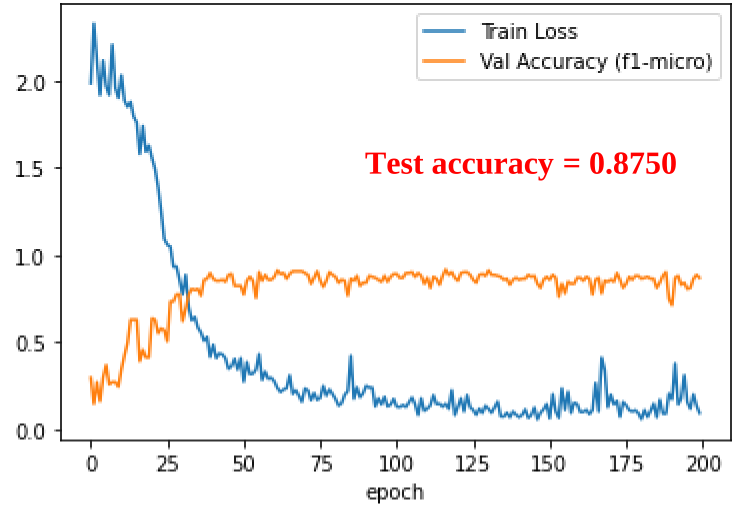
GraphSAGE-GCN



GraphSAGE-mean



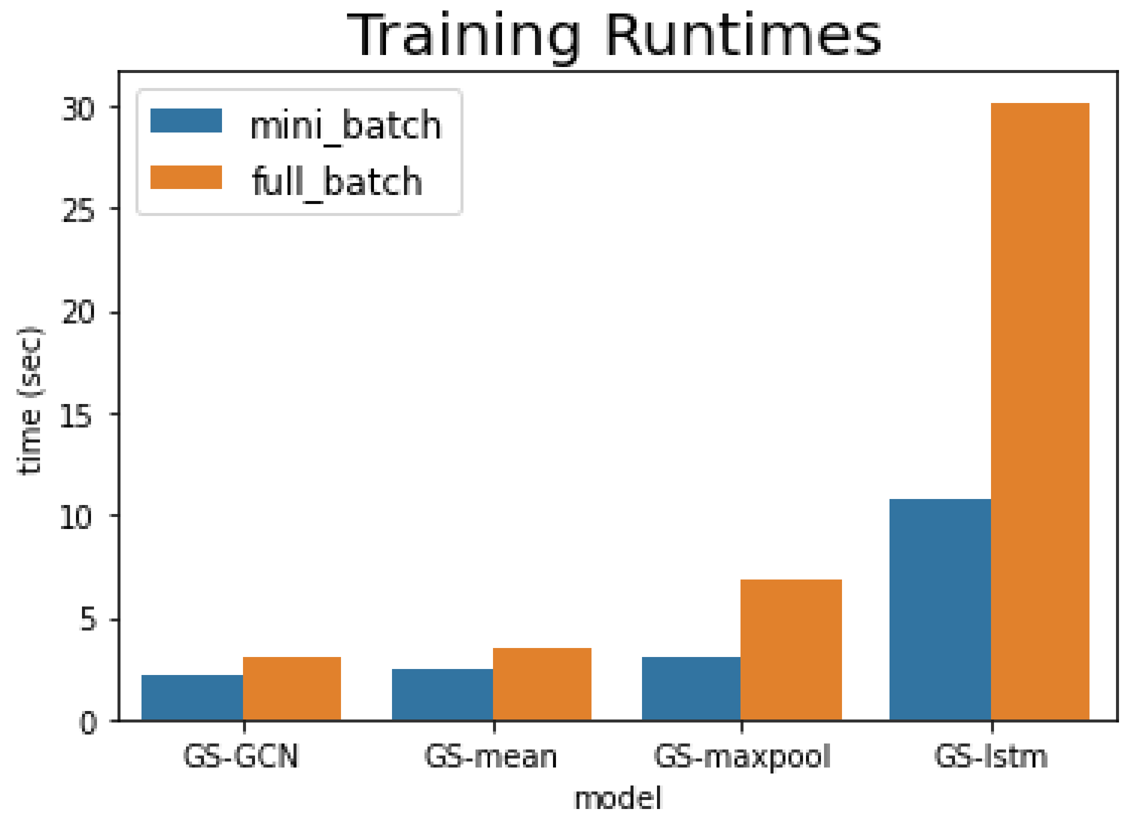
GraphSAGE-max pooling



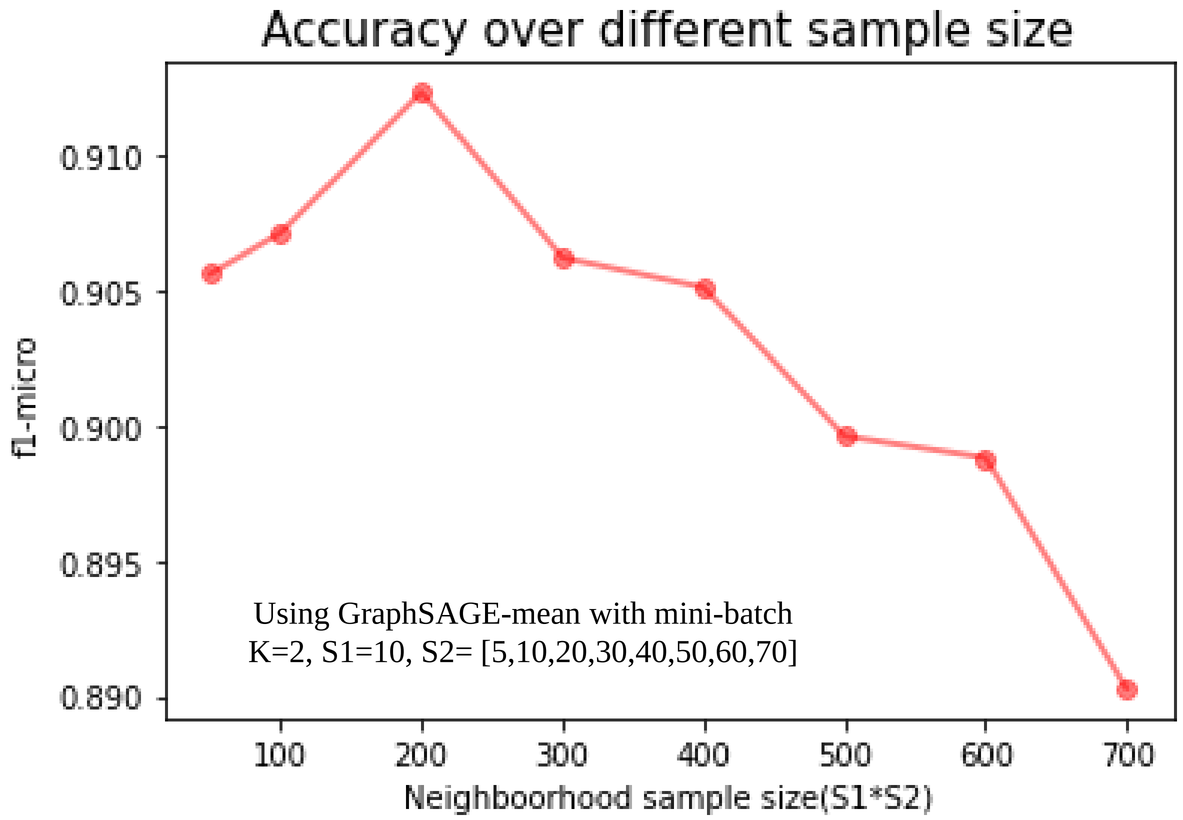
GraphSAGE-LSTM

Implement

Runtime Result
Cora Citation Dataset



Accuracy over
Neighborhood Sample size



Thank you