Inductive Representation Learning on Large Graph

2024.01.31 이수찬

ISysE
KAIST
Industrial & Systems Engineering

ABSTRACT (SUMMARY)

- Low-dimensional embeddings of nodes in large graphs have proven to be extremely useful in a variety of prediction tasks, from content recommendation to protein function identification.
- However, existing approaches struggle to predict sparse data or unseen nodes.
- GraphSAGE solves these problems by learning the topological structure around a node and the feature distribution of other nodes in the neighborhood.

1. INTRODUCTION

- Low-dimensional vector embeddings of nodes in large graphs have proven to be very useful as feature inputs for a variety of prediction and graph analysis tasks.
- However, previous work has focused on embedding nodes in one fixed graph, while many real-world applications require rapidly generating embeddings for unseen nodes or entirely new (sub)graphs. (transductive)
- Machine learning systems operate on evolving graphs and constantly encounter unseen nodes.
- An inductive approach to generating node embeddings also facilitates generalization across graphs with the same shape of features.
- The inductive node embedding problem requires the already optimized node embedding to "align" the newly observed subgraph with the structure and characteristics of the surrounding nodes in order to generalize to unseen nodes.

1. INTRODUCTION

- Most existing approaches to generating node embeddings are inherently transductive.
- The majority of these approaches directly optimize the embeddings for each node using matrix-factorization-based obj ectives, and do not naturally generalize to unseen data, since they make predictions on nodes in a single, fixed graph.
- There are also recent approaches to learning over graph structures using convolution operators that offer promise as a n embedding methodology.

- By incorporating node features in the learning algorithm, we simultaneously learn the topological structure of each no de's neighborhood as well as the distribution of node features in the neighborhood.
- Instead of training a distinct embedding vector for each node, we train a set of aggregator functions that learn to agg regate feature information from a node's local neighborhood.

1. INTRODUCTION

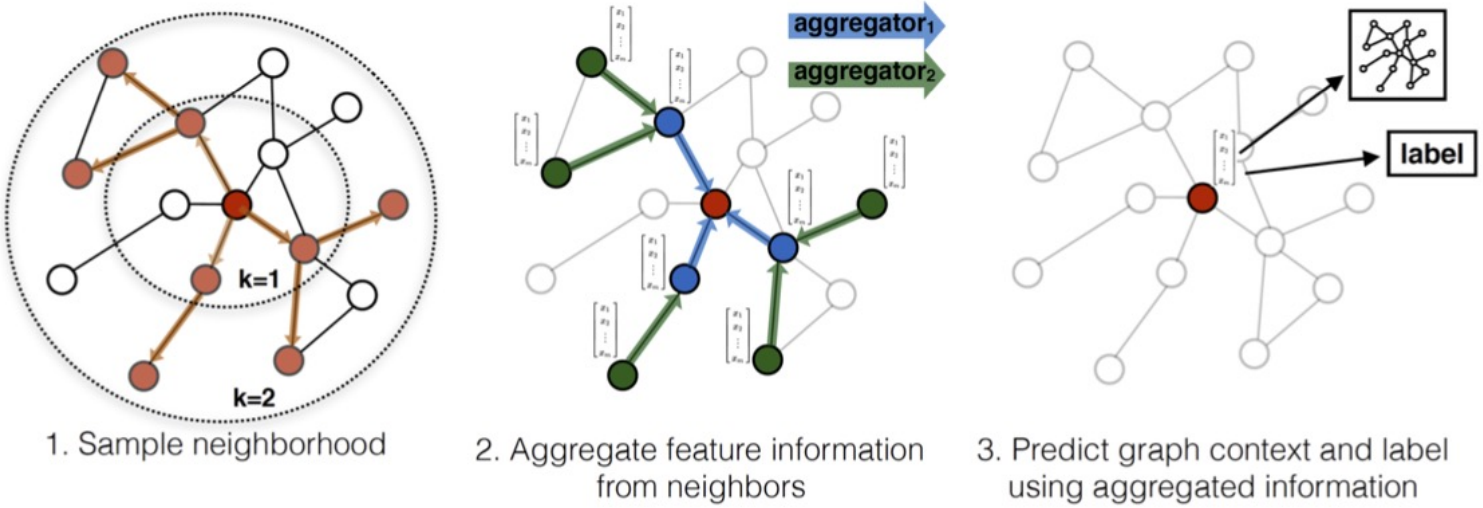


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

2. RELATED WORK

- **Factorization-based embedding approaches**
  - There are a number of recent node embedding approaches that learn low-dimensional embeddings using random walk statistics and matrix factorization-based learning objectives.
  - These methods also bear close relationships to more classic approaches to spectral clustering, multi-dimensional scaling, as well as the PageRank algorithm.
  - The objective function is invariant to orthogonal transformations of the embeddings, which means that the embedding space does not naturally generalize between graphs and can drift during retraining.
  - One notable exception to this trend is the Planetoid-I algorithm introduced by Yang et al. , which is an inductive, embedding- based approach to semi-supervised learning.
    - However, Planetoid-I does not use any graph structural information during inference.
    - Instead, it uses the graph structure as a form of regularization during training.

2. RELATED WORK

- **Supervised learning over graphs**
  - There is a rich literature on supervised learning over graph-structured data including graph kernel method.
  - There are also a number of recent neural network approaches to supervised learning over graph structures.
  - The focus of this work is generating useful representations for individual nodes, not entire graphs.
- **Graph convolutional networks**
  - In recent years, several convolutional neural network architectures for learning over graphs have been proposed.
  - The majority of these methods do not scale to large graphs or are designed for whole-graph classification.
  - The original GCN algorithm is designed for semi-supervised learning in a transductive setting, and the exact algorithm requires that the full graph Laplacian is known during training.
  - A simple variant of paper's algorithm can be viewed as an extension of the GCN framework to the inductive setting.

3. PROPOSED METHOD: GRAPH SAGE

3.1 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 $v \in \mathcal{V}$

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

3. PROPOSED METHOD: GRAPH SAGE

3.1 EMBEDDING GENERATION ALGORITHM

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, $\mathcal{N}_k : v \rightarrow 2^{\mathcal{V}}, \forall k \in \{1, ..., K\}$

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

$\mathcal{B}^K \leftarrow \mathcal{B}$;
for $k = K...1$ **do**
  $B^{k-1} \leftarrow \mathcal{B}^k$;
  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
$\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$

3. PROPOSED METHOD: GRAPH SAGE

3.2 LEARNING THE PARAMETERS OF GRAPHSAGE

- In order to learn useful, predictive representations in a fully unsupervised setting, this paper applies a graph-based loss function to the output representations, and tune the weight matrices, and parameters of the aggregator functions via SGD.
- The graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the repre sentations of disparate nodes are highly distinct:

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

- In cases where representations are to be used only on a specific downstream task, the unsupervised loss (above equati on) can simply be replaced, or augmented, by a task-specific objective (e.g., cross-entropy loss).

3. PROPOSED METHOD: GRAPH SAGE

3.3 AGGREGATOR ARCHITECTURES

- Unlike machine learning over N-D lattices, a node's neighbors have no natural ordering; thus, the aggregator functions in Algorithm 1 must operate over an unordered set of vectors.
- Ideally, an aggregator function would be symmetric (i.e., invariant to permutations of its inputs) while still being trainable and maintaining high representational capacity.
- The symmetry property of the aggregation function ensures that this neural network model can be trained and applied to arbitrarily ordered node neighborhood feature sets.

3. PROPOSED METHOD: GRAPH SAGE

3.3 AGGREGATOR ARCHITECTURES

- **Mean aggregator**
  - Mean aggregator simply take the elementwise mean of the vectors.
  - The mean aggregator is nearly equivalent to the convolutional propagation rule in the transductive GCN framework.

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

  - We call this modified mean-based aggregator convolutional since it is a rough, linear approximation of a localized spectral convolution.
  - An important distinction between this convolutional aggregator and other proposed aggregators is that it does not perform the concatenation operation in line 5 of Algorithm 1.
    - The convolutional aggregator does concatenate the node's previous layer representation with the aggregated neighborhood vector

3. PROPOSED METHOD: GRAPH SAGE

3.3 AGGREGATOR ARCHITECTURES

- **LSTM aggregator**
  - LSTMs have the advantage of larger expressive capability.
  - However, it is important to note that LSTMs are not inherently symmetric, since they process their inputs in a sequential manner.
  - We adapt LSTMs to operate on an unordered set by simply applying the LSTMs to a random permutation of the node's neighbors.

3. PROPOSED METHOD: GRAPH SAGE

3.3 AGGREGATOR ARCHITECTURES

- **Pooling aggregator**
  - Pooling aggregator is both symmetric and trainable.
  - In this pooling approach, each neighbor's vector is independently fed through a fully-connected neural network; following this transformation, an elementwise max-pooling operation is applied to aggregate information across the neighbor set:

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

  - In principle, the function applied before the max pooling can be an arbitrarily deep multi-layer perceptron, but we f ocus on simple single-layer architectures in this work.
  - No significant difference between max- and mean-pooling in developments test and thus focused on max-pooling for the rest of experiments.

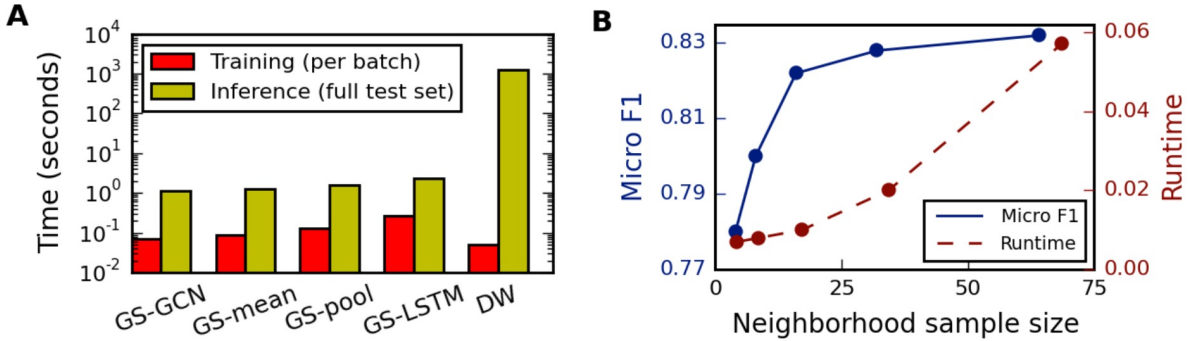
4. EXPERIMENTS

- Use 4 baselines: a random classifer, a logistic regression feature-based classifier (that ignores graph structure), the Dee pWalk algorithm as a representative factorization-based approach, and a concatenation of the raw features and Deep Walk embeddings.
- Use cross-entropy loss and ReLU as activation function.
- Set K = 2 with neighborhood sample sizes S1 = 25 and S2 = 10

4. EXPERIMENTS

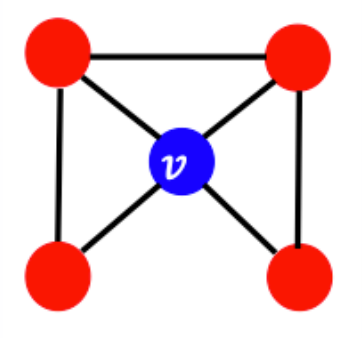
Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervise and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

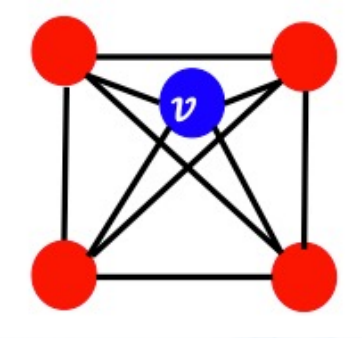
	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	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%



5. THEORETICAL ANALYSIS

- Provide insight into how GrpahSAGE can learn about graph structure, even though it is inherently based on features.
- Examine the ability to predict the clustering coefficient, which measures how clustered the nodes on a graph are.
- Clustering coefficient: The clustering coefficient represents the ratio of triangles (pairs of connected nodes) between a node's 1-hop neighbors (directly connected nodes).





5. THEORETICAL ANALYSIS

Theorem 1. *Let $\mathbf{x}_v \in U, \forall v \in \mathcal{V}$ denote the feature inputs for Algorithm 1 on graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where U is any compact subset of $\mathbb{R}^d$. Suppose that there exists a fixed positive constant $C \in \mathbb{R}^+$ such that $\|\mathbf{x}_v - \mathbf{x}_{v'}\|_2 > C$ for all pairs of nodes. Then we have that $\forall \epsilon > 0$ there exists a parameter setting $\mathbf{\Theta}^*$ for Algorithm 1 such that after $K = 4$ iterations*

$$|z_v - c_v| < \epsilon, \forall v \in \mathcal{V},$$

where $z_v \in \mathbb{R}$ are final output values generated by Algorithm 1 and c_v are node clustering coefficients.

- This theorem claims that GraphSAGE can approximate clustering coefficients to arbitrary degrees of accuracy.
- This is true when each node's feature input is high-dimensional and there is a certain minimum distance between every pair of nodes.
- The theorem proves that Algorithm 1 produces a final output value after K=4 iterations, and that this value is close enough for each node's clustering coefficient.
- Computing the clustering coefficient is similar to the pooling approach, so this proof is based on GraphSAGE with a pooling aggregator.
  - Prove the high performance of GraphSAGE with pooling aggregator

6. CONCLUSION

- Theoretical analysis about GraphSAGE provides insight into how this approach can learn about local graph structures.
- A number of extensions and potential improvements are possible, such as extending GraphSAGE to incorporate directed or multi-modal graphs.
- A particularly interesting direction for future work is exploring non-uniform neighborhood sampling functions, and perhaps even learning these functions as part of the GraphSAGE optimization.

7.IMPLEMENTATION

```python
# load dataset
dataset = Planetoid(root='/tmp/Cora', name='Cora')
data = dataset[0]
```

```python
# neighbor sampling torch lib
train_loader = NeighborLoader(
    data,
    num_neighbors=[25, 10],
    batch_size = 40,
    input_nodes = data.train_mask,
)
```

7.IMPLEMENTATION

```python
from torch_geometric.datasets import PPI


# load dataset
train_dataset = PPI(root=".", split='train')
val_dataset = PPI(root=".", split='val')
test_dataset = PPI(root=".", split='test')

from torch_geometric.data import Batch
from torch_geometric.loader import DataLoader, NeighborLoader

# apply neighbor sampling to training set
train_data = Batch.from_data_list(train_dataset)
loader = NeighborLoader(train_data, batch_size=2048, shuffle=True, num_neighbors=[25, 10], num_workers=2, persistent_workers=True)

# data loaders
train_loader = DataLoader(train_dataset, batch_size=2)
val_loader = DataLoader(val_dataset, batch_size=2)
test_loader = DataLoader(test_dataset, batch_size=2)
```

7.IMPLEMENTATION

```python
torch.manual_seed(-1) # initializing seed for reproducibility

class GraphSAGE(torch.nn.Module):
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.sage1 = SAGEConv(in_channels=dim_in, out_channels=dim_h, concat=False)
        self.sage2 = SAGEConv(in_channels=dim_h, out_channels=dim_out, concat=False)

    def forward(self, x, edge_index):
        h = self.sage1(x, edge_index)
        h = torch.relu(h)
        h = F.dropout(h, p=0.5, training=self.training)
        h = self.sage2(h, edge_index)
        return F.log_softmax(h, dim=1)
```

7.IMPLEMENTATION

```python
def fit(self, data, epochs):
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(self.parameters(), lr=0.01)

    self.train()
    for epoch in range(epochs+1):
        total_loss = 0
        total_f1 = 0
        val_loss = 0
        val_f1 = 0

        for batch in train_loader:
            optimizer.zero_grad()
            out = self(batch.x, batch.edge_index)
            loss = criterion(out[batch.train_mask], batch.y[batch.train_mask])
            total_loss += loss.item()

            preds = out[batch.train_mask].argmax(dim=1)
            labels = batch.y[batch.train_mask]
            total_f1 += f1_score(labels.cpu(), preds.cpu(), average='weighted')

            loss.backward()
            optimizer.step()

            val_out = self(batch.x, batch.edge_index)
            val_loss += criterion(val_out[batch.val_mask], batch.y[batch.val_mask])
            val_preds = val_out[batch.val_mask].argmax(dim=1)
            val_labels = batch.y[batch.val_mask]
            val_f1 += f1_score(val_labels.cpu(), val_preds.cpu(), average='weighted')

        if epoch % 5 == 0:
            print(f'Epoch {epoch:>3} | Train Loss: {total_loss/len(train_loader):.4f}  | Val Loss: {val_loss/len(train_loader):.4f} | Val F1: {val_f1/len(train_loader):.4f}')

```

7.IMPLEMENTATION

```python
@torch.no_grad()
def test(self, data):
    # data = data.sort(sort_by_row=False) # sorting for LSTM
    self.eval()
    out = self(data.x, data.edge_index)
    test_preds = out.argmax(dim=1)[data.test_mask]
    test_labels = data.y[data.test_mask]
    test_f1 = f1_score(test_labels.cpu(), test_preds.cpu(), average='weighted')
    return test_f1

```

```python
# GraphSAGE model
graphsage = GraphSAGE(dataset.num_features, 64, dataset.num_classes)

print(graphsage)

# Train
graphsage.fit(data, 30)

# Test
acc = graphsage.test(data)
print(f'GraphSAGE test F1 score: {acc:.4f}')
```

7.IMPLEMENTATION

	Sup. F1 score with Cora	Time with Cora	Sup. F1 score with PPI	Time with PPI
GraphSAGE-GCN	**0.7915**	1.2s	0.8280	51.7s
GraphSAGE-mean	**0.7915**	1.2s	0.8272	50.1s
GraphSAGE-LSTM	0.4396	**7m 49.0s**	-	-
GraphSAGE-pool	0.7307	1.5s	0.7745	2m 17.9s

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%

7.IMPLEMENTATION

- 잘된 점
  - document dataset, multi graph dataset에 대한 4가지 aggregation으로 GraphSAGE 성능 확인
- 보완할 점
  - GraphSAGE의 LSTM aggregation에 대한 성능
  - Unsupervised manner에 대한 GraphSAGE의 성능 확인