FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling

Credit: Jie Chen, Tengfei Ma, Cao Xiao

IBM Research

Presenter: Ryan McCampbell

https://qdata.github.io/deep2Read

Outline

Outline

Introduction

- Real world data often comes naturally as graphs
  - Social networks
  - Gene expression
  - Knowledge graphs
- Graph-based learning tasks
  - Node classification
  - Link prediction
- Recent work: extending popular network architectures to graphs
  - RNNs
  - CNNs

Graph Convolutions

- Node feature representation

$$H^{(l+1)} = \sigma(AH^{(l)}W^{(l)})$$

- A: adjacency matrix
- W: parameter matrix
- σ: nonlinearity

Graph Convolutions

- Transductive: embeddings computed for both train and test data simultaneously
  - Test data might not be available
- Bigger problem: recursive expansion of neighborhoods across layers
  - Particularly bad for dense graphs
  - Makes minibatch training non-scalable

FastGCN

- Idea: Idea: interpret vertices as iid samples of a probability distribution
- Treat loss and convolution layers as integral transforms of embedding functions
- Evaluate integrals through Monte Carlo approximation to get sample gradient

Outline

Related Work

- Spectral graph theory: whole graph feature representation
- Graph vertex embeddings
  - Matrix factorization based
  - Random walk based
  - Graph convolution (GCN)
- GraphSAGE
  - Learns node representations through aggregation of neighborhood information Use sampling to restrict neighborhood size
  - Distinction: This paper samples vertices rather than neighbors

Outline

Integral Transform

- For graphs: can't use independence to compute sample gradients
- Assume possibly infinite graph G' s.t. G is subgraph of G' and its vertices are iid samples of V' with probability distribution P

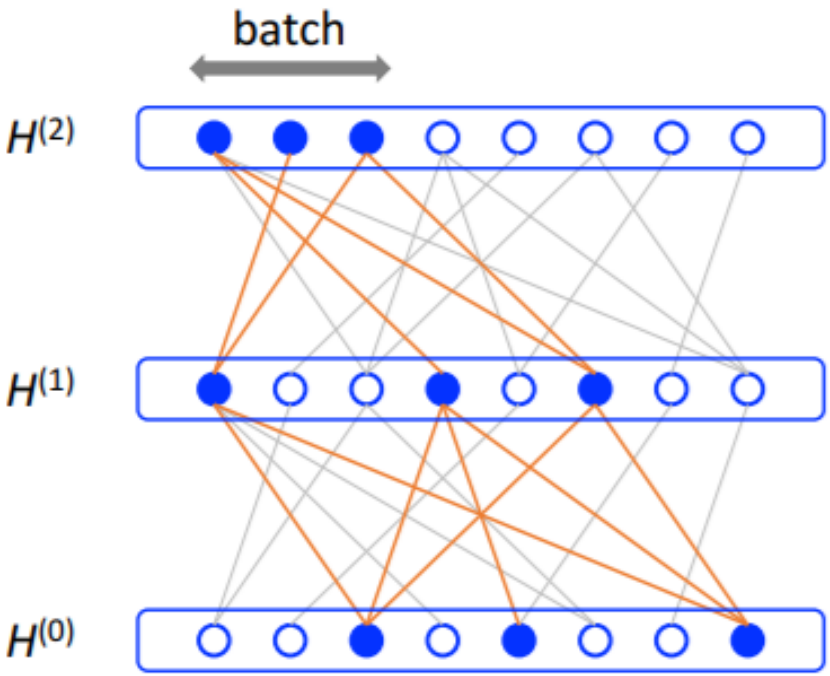
$$\tilde{H}^{(l+1)} = AH^{(l)}W^{(l)},\ H = \sigma(\tilde{H}),\ L = \frac{1}{n}\sum_{i=1}^{n} g(H^{(M)}(i,:))$$

$$\downarrow$$

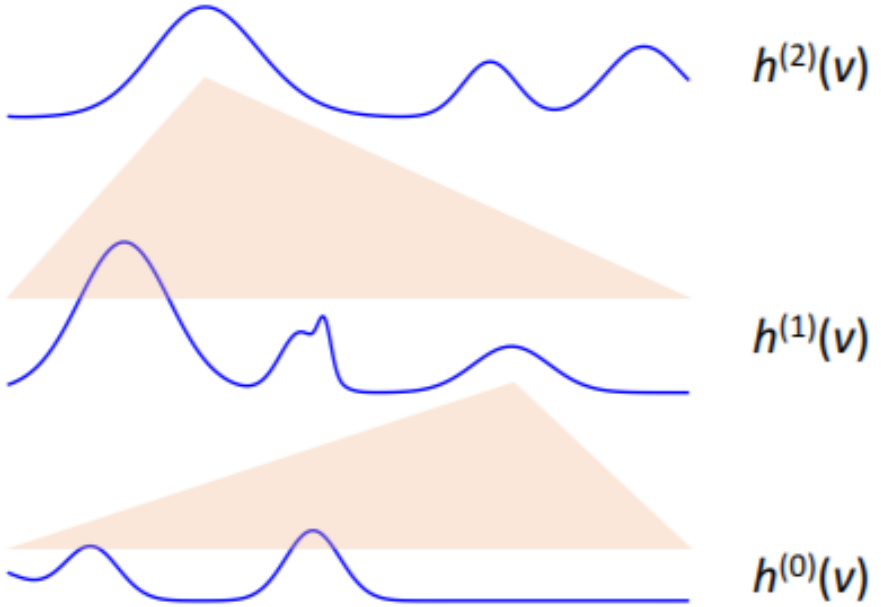
$$\tilde{h}^{(l+1)}(v) = \int A(v,u)h^{(l)}(u)W^{(l)}dP(u),\ h = \sigma(\tilde{h}),$$

$$L = E_{v\sim P}[g(h^{(M)}(v)] = \int g(h^{(M)}(v))dP(v)$$

Integral Transform



Graph convolution view



Integral transform view

Monte Carlo Estimation

- For each layer l, use t_l iid samples $u^{(l)}_{1\ldots t_l} \sim P$ to approximate the integral:

$$\tilde{h}^{(l+1)}_{t_{l+1}}(v) = \frac{1}{t_l}\sum_{j=1}^{t_l} A(v, u^{(l)}_j) h^{(l)}_{t_l}(u^{(l)}_j) W^{(l)}$$

$$L = \frac{1}{t_M}\sum_{i=1}^{t_M} g(h^{(M)}_{t_M}(u^{(M)}_i))$$

In practice

- Use bootstrapping: for each batch, sample uniformly with replacement each layer to obtain $u^{(l)}_{1\dots t_l}$

$$L = \frac{1}{t_M} \sum_{i=1}^{t_M} g(H^{(M)}(u_i^{(M)}, :))$$

$$H^{(l+1)}(v, :) = \sigma\left(\frac{n}{t_l} \sum_{j=1}^{t_l} A(v, u_j^{(l)}) H^{(l)}(u_j^{(l)}, :) W^{(l)} \right)$$

Algorithm

Algorithm 1 FastGCN batched training (one epoch)

for each batch **do**
  For each layer l, sample uniformly t_l vertices $u_1^{(l)}, \ldots, u_{t_l}^{(l)}$
  for each layer l **do** ▷ Compute batch gradient ∇L_{batch}
    If v is sampled in the next layer,

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

  end for
  $W \leftarrow W - \eta \nabla L_{\text{batch}}$ ▷ SGD step
end for

Variance Reduction

- Use importance sampling
- Let $Q(U)$ be new probability distribution

$$dQ(u) = \frac{b(u)^2 dP(u)}{\int b(u)^2 dP(u)}, \; b(u) = \left[\int A(v,u)^2 dP(v) \right]^{\frac{1}{2}}$$

- Discretized:

$$q(u) = \frac{\|A(:,u)\|^2}{\sum_{u' \in V} \|A(:,u')\|^2}$$

Algorithm

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

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

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

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

Inference

- Test data separated from training data
- Either compute test embeddings using full GCN architecture
- Or approximate them through sampling as for parameter learning
- Paper uses full architecture for inference

Comparison with GraphSAGE

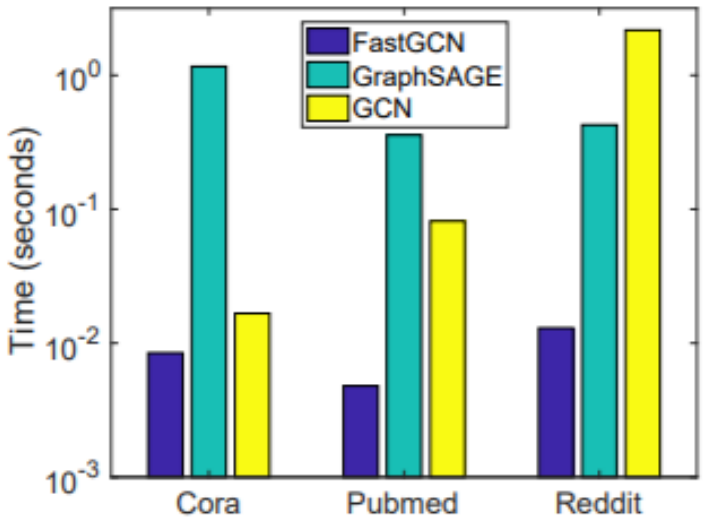
- GCN and GraphSAGE both have bottleneck caused by recursive neighborhood expansion
- GraphSAGE restricts neighborhood size for each layer: in worst case, $O(\prod t_l)$
- FastGCN samples vertices rather than neighbors in each layer: $O(\sum t_l)$

Outline

Results

Table 1: Dataset Statistics

Dataset	Nodes	Edges	Classes	Features	Training/Validation/Test
Cora	2,708	5,429	7	1,433	1,208/500/1,000
Pubmed	19,717	44,338	3	500	18,217/500/1,000
Reddit	232,965	11,606,919	41	602	152,410/23,699/55,334



Micro F1 Score

	Cora	Pubmed	Reddit
FastGCN	0.850	0.880	0.937
GraphSAGE-GCN	0.829	0.849	0.923
GraphSAGE-mean	0.822	0.888	0.946
GCN (batched)	0.851	0.867	0.930
GCN (original)	0.865	0.875	NA

Figure: Per-batch training time and prediction accuracy

Results

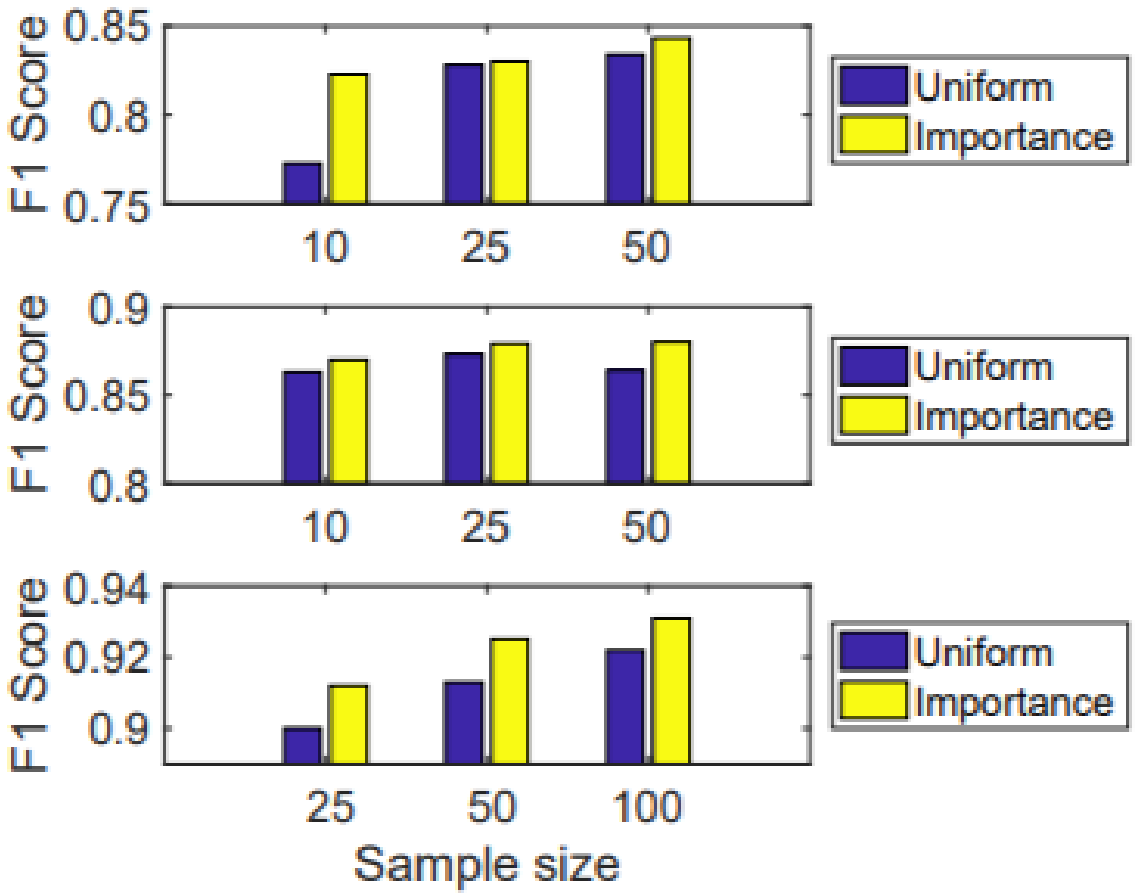


Figure: Comparison between uniform and importance sampling

Outline

Conclusions

- This largely solves the problem of dimensionality in large graphs without sacrificing accuracy
- It Generalizes to inductive learning on graphs with continuously changing nodes
- It shows that the receptive field does not have to increase for more distant nodes to provide reasonable accuracy