FASTGCN: FAST LEARNING WITH GRAPH CONVOULTIONAL NETWORKS VIA IMPORTANCE SAMPLING

Jie Chen, Tengfei Ma, Cao Xiao

ICLR 2018

김준하

CONTENTS

Motivation

- Computational scalability

Computational Scalability

- Recursive expansion of neighborhoods across layers
  - Incurring expensive computations in **batched training**
  - Even for a single vertex, quickly fill up a large portion of the graph

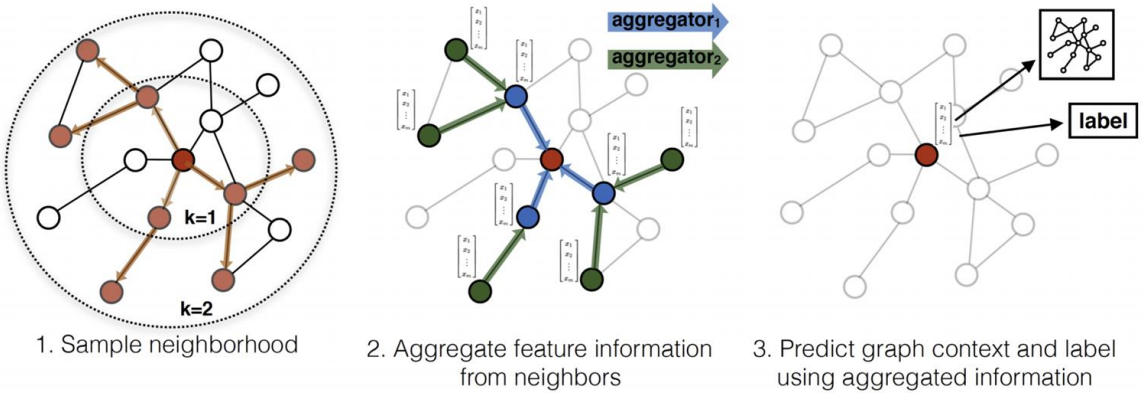


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

Computational Scalability

- Propose new interpretation of graph convolution
  - Yield a **controllable cost for per-batch computation**

Method

- New interpretation of graph convolution
  a. As **integral transforms of embedding function** under probability measures
  b. Approximation of the integral transforms by **Monte Carlo manner**
  c. **Reduce computational cost** by controlling number of samples while the **performance is still comparable**

Graph convolution as integral transform of function

- Assumption
  a. Graph vertices are i.i.d samples of some probability distribution *P*
  b. GCN operation: each layer defines an embedding function of vertices

$$\tilde{H}^{(l+1)} = \hat{A}H^{(l)}W^{(l)}, \quad H^{(l+1)} = \sigma(\tilde{H}^{(l+1)}), \quad l = 0, \ldots, M-1, \quad L = \frac{1}{n}\sum_{i=1}^{n} g(H^{(M)}(i,:)). \tag{1}$$

Graph convolution as integral transform of function

- Assumption
  a. Graph vertices are i.i.d samples of some probability distribution *P*
  b. GCN operation: each layer defines an embedding function of vertices
  c. Integral transform of the embedding function

$$\tilde{H}^{(l+1)} = \hat{A}H^{(l)}W^{(l)}, \quad H^{(l+1)} = \sigma(\tilde{H}^{(l+1)}), \quad l = 0, \ldots, M-1, \quad L = \frac{1}{n}\sum_{i=1}^{n} g(H^{(M)}(i,:)). \tag{1}$$

For the functional generalization, we write

$$\tilde{h}^{(l+1)}(v) = \int \hat{A}(v,u)h^{(l)}(u)W^{(l)}\, dP(u), \quad h^{(l+1)}(v) = \sigma(\tilde{h}^{(l+1)}(v)), \quad l = 0, \ldots, M-1, \tag{2}$$

$$L = \mathrm{E}_{v\sim P}[g(h^{(M)}(v))] = \int g(h^{(M)}(v))\, dP(v). \tag{3}$$

Approximation by Monte Carlo manner

- Monte Carlo method
  - A subset of computational algorithms that use the process of **repeated random sampling** to make numerical estimations of unknown parameters
  - Writing GCN in the functional form allows for evaluating the integrals in the Monte Carlo manner, which leads to a **batched training algorithm**

Approximation by Monte Carlo manner

- Monte Carlo method (GCN, integral, Apprx by MC)

$$\tilde{H}^{(l+1)} = \hat{A}H^{(l)}W^{(l)}, \quad H^{(l+1)} = \sigma(\tilde{H}^{(l+1)}), \quad l = 0, \ldots, M-1,$$

For the functional generalization, we write

$$\tilde{h}^{(l+1)}(v) = \int \hat{A}(v,u)h^{(l)}(u)W^{(l)}\, dP(u), \quad h^{(l+1)}(v) = \sigma(\tilde{h}^{(l+1)}(v)), \quad l = 0, \ldots, M-1,$$

$$\tilde{h}^{(l+1)}_{t_{l+1}}(v) := \frac{1}{t_l}\sum_{j=1}^{t_l} \hat{A}(v, u_j^{(l)})h^{(l)}_{t_l}(u_j^{(l)})W^{(l)}, \quad h^{(l+1)}_{t_{l+1}}(v) := \sigma(\tilde{h}^{(l+1)}_{t_{l+1}}(v)), \quad l = 0, \ldots, M-1,$$

Approximation by Monte Carlo manner

- Monte Carlo method (GCN, integral, Apprx by MC)

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

$$L = \mathrm{E}_{v\sim P}[g(h^{(M)}(v))] = \int g(h^{(M)}(v))\, dP(v).$$

$$L_{t_0,t_1,\ldots,t_M} := \frac{1}{t_M}\sum_{i=1}^{t_M} g(h_{t_M}^{(M)}(u_i^{(M)})).$$

Algorithm

- Assume sample distribution as uniform
  - Original distribution is unknown

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,

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

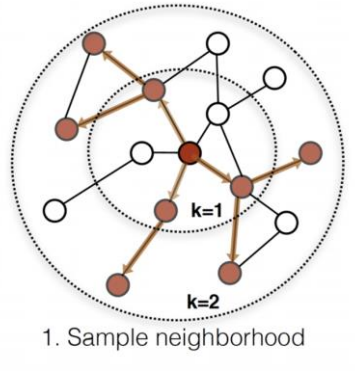
  end for
  $W \leftarrow W - \eta \nabla L_{\text{batch}}$ ▷ SGD step
end for

Comparison with GraphSAGE

- GraphSAGE (Hamilton et al., 2017)
  - Architecture for generating vertex embeddings through aggregating neighborhood information
  - Memory bottleneck with GCN, caused by recursive neighborhood expansion
  - Also restrict neighborhood size to reduce computational cost

Comparison with GraphSAGE

- GraphSAGE (Hamilton et al., 2017)
  - Samples neighbors
  - t_L neighbors for each vertex in L-th layer
    - Total cost of expanded neighborhood: $O(\Pi_L t_L)$: product of t_L's
- FastGCN
  - Sample vertices
  - t_L vertices for L-th layer
    - Total cost of expanded neighborhood: $O(\Sigma_L t_L)$: sum of t_L's



Variance Reduction by Importance Sampling

- Importance sampling
  - Certain values of the input random variables have **more impact** on the parameter being estimated than others.
  - Choose a distribution which **"encourages" the important values**

Variance Reduction by Importance Sampling

$$Y_i = \frac{f(X_i)}{pdf(X_i)}$$

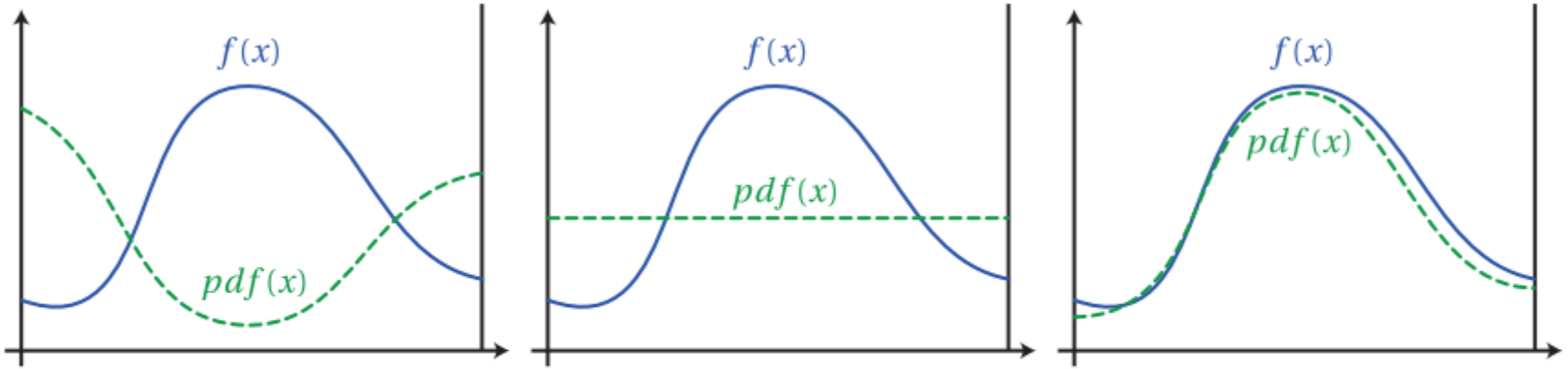


Figure A.2: Comparison of three probability density functions. The PDF on the right provides variance reduction over the uniform PDF in the center. However, using the PDF on the left would significantly increase variance over simple uniform sampling.

Variance Reduction by Importance Sampling

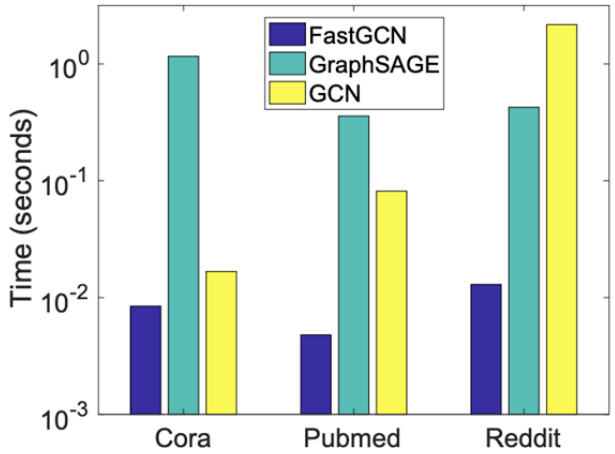
Algorithm 2 FastGCN batched training (one epoch), improved version

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

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

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

Results



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 3: Per-batch training time in seconds (left) and prediction accuracy (right). For timing, GraphSAGE refers to GraphSAGE-GCN in Hamilton et al. (2017). The timings of using other aggregators, such as GraphSAGE-mean, are similar. GCN refers to using batched learning, as opposed to the original version that is nonbatched; for more details of the implementation, see the appendix. The nonbatched version of GCN runs out of memory on the large graph Reddit. The sample sizes for FastGCN are 400, 100, and 400, respectively for the three data sets.

Results

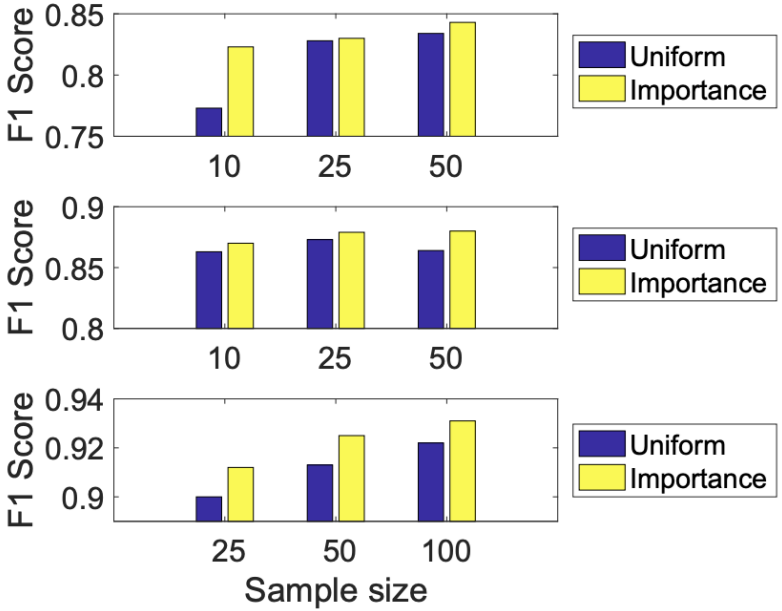


Figure 2: Prediction accuracy: uniform versus importance sampling. The three data sets from top to bottom are ordered the same as Table 1.

Conclusion

- Propose **sampling scheme** in reformulation of loss and gradient, well justified through an **alternative view of graph convolutions** in form of integral transforms of embedding function, which yield **controllable cost for per-batch computation**

- An additional investigation whether and **how variance reduction may improve the estimator by importance sampling**

Code Analysis

- The code is tensorflow based implementation
  - link: https://github.com/matenure/FastGCN
- Main training loop
  - Importance sampling
- Main model implementation
  - Sparse matrix operation
- Results
- Execution guide

Main training loop

- rank: # of samples
- support: normalized adj matrix
- q1: sampled indices
- p0: degree

```
for batch in iterate_minibatches_listinputs([normADJ_train, y_train], batchsize=1024, shuffle=True):
    [normADJ_batch, y_train_batch] = batch

    if rank1 is None:
        support1 = sparse_to_tuple(normADJ_batch)
        features_inputs = train_features
    else:
        distr = np.nonzero(np.sum(normADJ_batch, axis=0))[1]
        if rank1 > len(distr):
            q1 = distr
        else:
            q1 = np.random.choice(distr, rank1, replace=False, p=p0[distr]/sum(p0[distr]))  # top layer


        support1 = sparse_to_tuple(normADJ_batch[:, q1].dot(sp.diags(1.0 / (p0[q1] * rank1))))
        if len(support1[1])==0:
            continue
        features_inputs = train_features[q1, :]  # selected nodes for approximation
    # Construct feed dictionary
    feed_dict = construct_feeddict_forMixlayers(features_inputs, support1, y_train_batch,
                                                placeholders)
    feed_dict.update({placeholders['dropout']: FLAGS.dropout})
```

Main training loop

If rank is not given or larger than total number of nodes

→ Use all samples

```
for batch in iterate_minibatches_listinputs([normADJ_train, y_train], batchsize=1024, shuffle=True):
    [normADJ_batch, y_train_batch] = batch

    if rank1 is None:
        support1 = sparse_to_tuple(normADJ_batch)
        features_inputs = train_features
    else:
        distr = np.nonzero(np.sum(normADJ_batch, axis=0))[1]
        if rank1 > len(distr):
            q1 = distr
        else:
            q1 = np.random.choice(distr, rank1, replace=False, p=p0[distr]/sum(p0[distr]))  # top layer


        support1 = sparse_to_tuple(normADJ_batch[:, q1].dot(sp.diags(1.0 / (p0[q1] * rank1))))
        if len(support1[1])==0:
            continue
        features_inputs = train_features[q1, :]  # selected nodes for approximation
    # Construct feed dictionary
    feed_dict = construct_feeddict_forMixlayers(features_inputs, support1, y_train_batch,
                                                placeholders)
    feed_dict.update({placeholders['dropout']: FLAGS.dropout})
```

Main training loop

If not, sample node indices according to distribution proportional to degree of the nodes.

: **Importance sampling**

And then use subset of matrix and input features corresponding to the sampled indices

```
for batch in iterate_minibatches_listinputs([normADJ_train, y_train], batchsize=1024, shuffle=True):
    [normADJ_batch, y_train_batch] = batch

    if rank1 is None:
        support1 = sparse_to_tuple(normADJ_batch)
        features_inputs = train_features
    else:
        distr = np.nonzero(np.sum(normADJ_batch, axis=0))[1]
        if rank1 > len(distr):
            q1 = distr
        else:
            q1 = np.random.choice(distr, rank1, replace=False, p=p0[distr]/sum(p0[distr]))  # top layer


        support1 = sparse_to_tuple(normADJ_batch[:, q1].dot(sp.diags(1.0 / (p0[q1] * rank1))))
        if len(support1[1])==0:
            continue
        features_inputs = train_features[q1, :]  # selected nodes for approximation
    # Construct feed dictionary
    feed_dict = construct_feeddict_forMixlayers(features_inputs, support1, y_train_batch,
                                                placeholders)
    feed_dict.update({placeholders['dropout']: FLAGS.dropout})
```

Main training loop

Finally, with selected adjacency matrix (normalized) and input features, train the model

```
# Construct feed dictionary
feed_dict = construct_feeddict_forMixlayers(features_inputs, support1, y_train_batch,
                                placeholders)
feed_dict.update({placeholders['dropout']: FLAGS.dropout})
# Training step
outs = sess.run([model.opt_op, model.loss, model.accuracy], feed_dict=feed_dict)
```

Model implementation

- Get preprocessed input features as AX, not just input X
- Use Adam optimizer
- Final output dimension is same as number of class of label

```
class GCN_APPRO_Mix(Model): #mixture of dense and gcn
    def __init__(self, placeholders, input_dim, **kwargs):
        super(GCN_APPRO_Mix, self).__init__(**kwargs)
        self.inputs = placeholders['AXfeatures']# A*X for the bottom layer, not original feature X
        self.input_dim = input_dim
        # self.input_dim = self.inputs.get_shape().as_list()[1]  # To be supported in future Tensorflow versions
        self.output_dim = placeholders['labels'].get_shape().as_list()[1]
        self.placeholders = placeholders
        self.support = placeholders['support']

        self.optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate)

        self.build()
```

Model implementation

- Model consists of a FC layer with relu followed by GCN

```
def _build(self):
    self.layers.append(Dense(input_dim=self.input_dim,
                             output_dim=FLAGS.hidden1,
                             placeholders=self.placeholders,
                             act=tf.nn.relu,
                             dropout=True,
                             sparse_inputs=False,
                             logging=self.logging))

    self.layers.append(GraphConvolution(input_dim=FLAGS.hidden1,
                                        output_dim=self.output_dim,
                                        placeholders=self.placeholders,
                                        support=self.support,
                                        act=lambda x: x,
                                        dropout=True,
                                        logging=self.logging))
```

Model implementation

- For GCN module, use **sparse matrix multiplication** implemented in tensorflow

```
def dot(x, y, sparse=False):
    """Wrapper for tf.matmul (sparse vs dense)."""
    if sparse:
        res = tf.sparse_tensor_dense_matmul(x, y)
    else:
        res = tf.matmul(x, y)
    return res
```

Model implementation

- Output probability by using **softmax** function
- **Cross entropy loss** for multiple-classification loss

```
def predict(self):
    return tf.nn.softmax(self.outputs)

def _loss(self):
    # Weight decay loss
    for var in self.layers[0].vars.values():
        self.loss += FLAGS.weight_decay * tf.nn.l2_loss(var)

    # Cross entropy error
    self.loss += softmax_cross_entropy(self.outputs, self.placeholders['labels'])
```

Results

- Overall score is similar to the one reported
- Cora
  - The result for (uni, 50) is far lower than reported
    - About 83(%)
- Pubmed
  - Uniform > Importance
- The difference might caused by
  - Different version of package
  - GPU model

Dataset	Sampling method	# of samples	Accuracy	Training time (per epoch)
Cora	Uniform	10	77.60	0.0183s
		50	68.30	0.0203s
	Importance	10	78.30	0.0220s
		50	84.70	0.0209s
Pubmed	Uniform	10	85.20	0.2977s
		50	86.60	0.3336
	Importance	10	82.90	0.2266s
		50	86.10	0.2164s

Execution guide

1. docker pull tensorflow/tensorflow:1.4.1-devel-gpu-py3
2. Run the image and attach the created container
3. apt update && apt upgrade -y && apt install git -y
4. git clone https://github.com/matenure/FastGCN.git
    a. Dataset in data/ directory (no need to download)
5. pip install networkx==1.11
6. Run code
    a. python3 pubmed_Mix_sampleA.py (for importance sampling)
    b. python3 pubmed_Mix_uniform.py (for uniform sampling)
    c. Change argument of main() function to change # of samples