Time and Space Complexity of Graph Convolutional Networks

Derrick Blakely, Jack Lanchantin, Yanjun Qi
University of Virginia
{dcb7xz, jjl5sw, yq2h}@virginia.edu

Abstract

We analyze the time and space complexity of graph convolutional network (GCN) layers. This is done for (1) forward computation and (2) back-propagation with gradient descent and stochastic gradient descent. We draw connections to software implementations, particularly PyTorch Geometric.

1 Background

1.1 The GCN Layer

We introduce the semi-supervised graph convolutional network (GCN) model from [4]. Suppose we are given a graph $G = (V, E, A)$, with $N = |V|$ nodes, $|E|$ edges, and a sparse adjacency matrix A of size $N \times N$. Each node corresponds to an F-dimensional embedding, so we have an $N \times F$ embedding matrix X. On average, each node has a degree of d.

To show the computation of a GCN, first we define $\hat{A} = A + I$, the adjacency matrix with self-loops. $\hat{D}$ is the diagonal degree matrix of $\hat{A}$. A' is the reduced adjacency matrix with self-loops added to every node:

$$\mathbf{A}' = \hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}} \tag{1}$$

The computation of the lth layer of a GCN network is:

$$\mathbf{X}^{l+1} = \sigma(\mathbf{A}'\mathbf{X}^{l}\mathbf{W}^{l}) \tag{2}$$

Where $\sigma(\cdot)$ is a non-linear activation function (typically ReLU) and $\mathbf{W}^l$ is a feature transformation matrix $\in \mathbb{R}^{F_l \times F_{l+1}}$. In other words, it maps node features of size F_l to features of size F_{l+1}. For simplicity, we assume the node features at every layer are size-F. As such, $\mathbf{W}^l$ is an $F \times F$ matrix.

It is common to decompose a GCN layer's computation into two parts: (1) computation of $\mathbf{Z}^l$, the transformation of the input features; (2) computation of $\mathbf{X}^{l+1}$, the final outputted embeddings of the layer:

$$\mathbf{Z}^{l} = \mathbf{X}^{l}\mathbf{W}^{l} \tag{3}$$

$$\mathbf{X}^{l+1} = \sigma(\mathbf{A}'\mathbf{Z}^{l}) \tag{4}$$

1.2 Implementation and Semantics

The layers of many GNNs, including GCNs, are often given Gather-Apply-Scatter (GAS) or Message-Passing semantics. Just a few examples: PyTorch Geometric [2], NGra [5], Cavs [8], and DGL [1] all do this. These semantics provide intuition and influence software implementations.

First, the $\mathbf{Z}^l = \mathbf{X}^l\mathbf{W}^l$ multiplication transforms input features to output features. In subsequent steps, these are to be used by normalization and neighborhood aggregation. Computing $\mathbf{Z}^l$ simply requires

a dense matrix multiplication. For example, PyTorch Geometric runs $Z = torch.matmul(X, W)$ as the first line of code in a GCN layer.

Next, we must compute $\mathbf{A}'$. PyTorch Geometric does not directly compute this. Rather, they use a coordinate (COO) form edge index matrix of size $2 \times |E|$ and store the normalization coefficients in a separate matrix called "norm." These steps can be precomputed and reused by all layers and each subsequent epoch.

The computation of $\mathbf{A}'\mathbf{Z}^l$ is what involves the so-called "message passing" and aggregation. This is computed as a sparse operation decomposed into three phases. Conveniently, the three phases lend themselves to a GAS or Message-Passing interpretation:

1. Create messages: in the GCN case, we take each $z_i \in \mathbf{Z}^l$ and normalize. Each normalized z_i is described as a "message" that must be passed from a node to each of its neighbors.
2. Scatter: this phase performs a sparse multiplication of the adjacency matrix with $\mathbf{Z}^l$. Because A is not explicitly used, the scatter stage loops over $\mathbf{Z}^l$ and E to accumulate each z_i into its neighbors. It could be achieved with a literal $\mathbf{A}'\mathbf{Z}^l$ multiplication, but this would be inefficient as discussed below. The output of scatter is almost the final embedding output for the layer.
3. Update (optional): provides an additional update to each node embedding. For example, an additive bias.

The output of the above stages is $\mathbf{A}'\mathbf{X}^l\mathbf{W}^l$.

In summary, $\mathbf{Z}^l = \mathbf{X}^l\mathbf{W}^l$ is implemented as an explicit matrix multiplication. $\mathbf{A}'\mathbf{Z}^l$ is an implicit operation with message-passing semantics, carried out in three stages. The final step of a GCN layer is simply to apply the nonlinearity, yielding the final embeddings as $\mathbf{X}^{l+1} = \sigma(\mathbf{A}'\mathbf{X}^l\mathbf{W}^l)$.

2 Forward Pass

We analyze the complexity of the forward step by decomposing Equation 2 into three high-level operations:

1. $\mathbf{Z}^l = \mathbf{X}^l\mathbf{W}^l$: feature transformation
2. $\mathbf{X}^{l+1} = \mathbf{A}'\mathbf{Z}^l$: neighborhood aggregation
3. $\sigma(\cdot)$: activation

Part 1 is a dense matrix multiplication between matrices of size $N \times F_l$ and $F_l \times F_{l+1}$. We assume for all l, $F_l = F_{l+1} = F$. Therefore, this is $O(NF^2)$.

Naively, part 2 is a multiplication between matrices of size $N \times N$ and $N \times F$, yielding $O(N^2F)$ time complexity. In practice, we compute this using a sparse operator, such as the PyTorch [6] scatter function. For each row (i, j) of the edge matrix E, we must compute $x_j^{l+1} += z_i^l$, which are each F-dimensional vectors. This yields a total cost of $O(|E|F)$. An alternative view is that each node has d neighbors on average. Neighborhood aggregation for each node therefore requires $O(dF)$ work, with a total of $O(NdF) = O(|E|F)$.

Part 3 is simply an element-wise function, so its cost is $O(N)$.

Over L layers, this results in $O(LNF^2 + LNdF + LN) = O(LNF^2 + LNdF) = O(LNF^2 + L|E|F)$.

3 Backward Pass

3.1 Full Gradient Descent

We learn the L $\mathbf{W}^l$ and $\mathbf{X}^l$ matrices in equation 2 with gradient descent by minimizing:

$$\mathcal{L} = \frac{1}{|Y|} \sum_{i \in [|Y|]} loss(y_i, \hat{y}_i) \tag{5}$$

where $\hat{y}_i$ is the ith row of $\hat{\mathbf{Y}} = \sigma(\mathbf{A}'\mathbf{X}^L\mathbf{W}^L)$, y_i is the ith true label from Y. Full gradient descent is used in the original GCN paper [4]. For each training iteration, we must compute $\nabla_W \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial W^1}, ..., \frac{\partial \mathcal{L}}{\partial \mathbf{W}^L}]$ and $\nabla_X \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial X^1}, ..., \frac{\partial \mathcal{L}}{\partial \mathbf{X}^L}]$, which are upper bounded by the computation of $\frac{\partial \mathcal{L}}{\partial W^1}$ and $\frac{\partial \mathcal{L}}{\partial X^1}$, respectively. Applying the chain rule, we are interested in computing:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^L}\Big)\Big(\frac{\partial \mathbf{Z}^L}{\partial \mathbf{X}^L}\Big)...\Big(\frac{\partial \mathbf{Z}^l}{\partial \mathbf{X}^l}\Big)\Big(\frac{\partial \mathbf{X}^l}{\partial \mathbf{Z}^{l-1}}\Big)...\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}^1} = \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^L}\Big)\Big(\frac{\partial \mathbf{Z}^L}{\partial \mathbf{X}^L}\Big)...\Big(\frac{\partial \mathbf{Z}^l}{\partial \mathbf{X}^l}\Big)\Big(\frac{\partial \mathbf{X}^l}{\partial \mathbf{Z}^{l-1}}\Big)...\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{7}$$

Evaluation of these equations requires evaluating a number of Jacobians. Full evaluation of any of these Jacobians is prohitively expensive in both time and memory. However, using reverse-mode automatic differentiation, as in PyTorch [7], evaluation of a function's Jacobian requires only a constant factor more computation than forward evaluation of the function. Let f be a matrix-valued function using elementary operators, $T(f, \nabla f)$ be the computation time for evaluating f and ∇f, and $T(f)$ be the computation time for evaluating f. As shown in [3], their ratio is bound by a constant factor C using reverse-mode automatic differentiation:

$$\frac{T(f, \nabla f)}{T(f)} \leq C \tag{8}$$

From this we may conclude that the backwards pass is also $O(LNF^2 + L|E|F)$. However, we show in practice how this can be achieved for the composition of matrix operations from equation 2. We start with the case of a 1-layer GCN and build up to a generalized expression.

3.2 1-Layer GCN: Differentiating with respect to W

We look at the case of performing backpropagation for a 1-layer GCN. For simplicity, we ignore the activation $\sigma(\cdot)$, defining $\hat{\mathbf{Y}} = \mathbf{A}'\mathbf{Z}^1$ and $\mathbf{Z}^1 = \mathbf{X}^\mathbf{1}\mathbf{W}^\mathbf{1}$. Therefore, we seek to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{9}$$

We assume we are given $\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}$, so we must next compute $\frac{\partial \hat{\mathbf{Y}}}{\partial Z^L}$. If $\hat{\mathbf{Y}}$ and Z^L have dimensions $N \times F$, then this partial is an $(N \times F) \times (N \times F)$ Jacobian. Full evaluation of this Jacobian is $\Omega(N^2F^2)$ in time and space. Just imagine how expensive this would be if we let $N = F = 1000$, reasonable sizes for a hidden GCN layer. We would need to compute 10^{12} entries and if each one is a 32-bit float, we would need 4TB of memory.

Fortunately, we do not need to fully evaluate the Jacobian of $\hat{\mathbf{Y}}$ at $\mathbf{Z}^1$. Many of the entries will simply be 0, thus making no contribution to any parameter updates. And in the case of most neural network layers, the non-zero entries can computed efficiently. Rather, we can focus on what really needs to be computed to obtain $\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial Z^L}\Big)$. In fact, we can simply compute:

$$\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial Z^L}\Big) = \Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big) \tag{10}$$

Because $(\mathbf{A}')^\top$ is $N \times N$ and $\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)$ is $N \times F$, this requires time $O(N^2F)$. But as noted previously, A' is a sparse adjacency matrix, so we can compute this in time $O(|E|F)$. In addition to being much more efficient, this formulation also has a nice interpretation. Recall that $A'Z$ performs neighborhood aggregation. $(A')^\top$ reverses the direction of all of the edges, so $(A')^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)$ can be seen as dispersing loss values back to the nodes' neighbors.

For clarity, we let $\mathbf{D} = (\mathbf{A}')^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)$. Therefore, continuing with backpropagation, we have:

$$\frac{\partial \mathcal{L}}{\partial W^1} = \mathbf{D}\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{11}$$

Here, we encounter the same problem as before: $\frac{\partial \mathbf{Z}^L}{\partial \mathbf{W}^L}$ is a Jacobian of size $(N \times F) \times (F \times F)$, which is prohibitively large. However, we can use essentially the same trick as before to avoid full evaluation. Noting that $\mathbf{Z}^1 = \mathbf{X}^1\mathbf{W}^1$, we have:

$$\frac{\partial \mathcal{L}}{\partial W^1} = \mathbf{X}^\top \mathbf{D} \tag{12}$$

$$= \mathbf{X}^\top (\mathbf{A}')^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big) \tag{13}$$

$\mathbf{X}^\top$ is $(F \times N)$ and $\mathbf{D}$ is $(N \times F)$, so the output is $(F \times F)$. This is as expected considering we need a total of F^2 $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{i,j}}$ values to perform the weight updates. This multiplication is $O(NF^2)$. The total cost of computing $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1}$ is $O(NF^2 + |E|F)$.

3.3 1-Layer GCN: Differentiating with respect to X

In this case, we seek to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}^1} = \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{14}$$

As before:

$$\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^1}\Big) = \Big(\mathbf{A}'\Big)^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big) = \mathbf{D} \tag{15}$$

Which requires $O(|E|F)$ time. Next we compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}^1} = \mathbf{D}\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{16}$$

In this case, we can avoid explicit evaluation of the Jacobian of $\mathbf{Z}^1$ at $\mathbf{X}^1$ by observing:

$$\mathbf{D}\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) = \mathbf{D}\mathbf{W}^\top = \Big(\mathbf{A}'\Big)^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\mathbf{W}^\top \tag{17}$$

This requires an $(N \times F)$ with $(F \times F)$ matrix multiplication, done in $O(NF^2)$ time. The total cost of computing $\frac{\mathcal{L}}{\mathbf{X}^1}$ is $O(NF^2 + |E|F)$.

3.4 2-Layer GCN

Now let us examine the case of a 2-layer GCN, again with activations omitted, where we seek to compute the partial derivative of the loss with respect to the first layer node embeddings $\mathbf{X}^1$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}^1} = \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^2}\Big)\Big(\frac{\partial \mathbf{Z}^2}{\partial \mathbf{X}^2}\Big)\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{18}$$

Applying the two techniques described in the 1-layer GCN for avoiding full Jacobian evaluation, we can compute this as follows:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}^1} = \Big[\Big(\mathbf{A}'\Big)^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \mathbf{Z}^2}{\partial \mathbf{X}^2}\Big)\Big]\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{19}$$

$$= \Big[\Big(\mathbf{A}'\Big)^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top\Big]\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{20}$$

$$= \Big(\mathbf{A}'\Big)^\top \Big[\Big(\mathbf{A}'\Big)^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top\Big]\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{X}^1}\Big) \tag{21}$$

$$= \Big(\mathbf{A}'\Big)^\top \Big[\Big(\mathbf{A}'\Big)^\top \Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top\Big]\Big(\mathbf{W}^1\Big)^\top \tag{22}$$

The computation is almost identical with respect to the first layer weight matrix $\mathbf{W^1}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \Big[\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \hat{\mathbf{Y}}}{\partial \mathbf{Z}^2}\Big)\Big(\frac{\partial \mathbf{Z}^2}{\partial \mathbf{X}^2}\Big)\Big]\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{23}$$

$$= \Big[\Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\frac{\partial \mathbf{Z}^2}{\partial \mathbf{X}^2}\Big)\Big]\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{24}$$

$$= \Big[\Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top\Big]\Big(\frac{\partial \mathbf{X}^2}{\partial \mathbf{Z}^1}\Big)\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{25}$$

$$= \Big(\mathbf{A}'\Big)^\top\Big[\Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top\Big]\Big(\frac{\partial \mathbf{Z}^1}{\partial \mathbf{W}^1}\Big) \tag{26}$$

$$= \Big(\mathbf{X}^1\Big)^\top\Big(\mathbf{A}'\Big)^\top\Big[\Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top\Big] \tag{27}$$

3.5 L-Layer GCN

We note that the term $\Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}\Big)\Big(\mathbf{W}^2\Big)^\top$ above is equivalent to $\Big(\frac{\partial \mathcal{L}}{\partial \mathbf{X}^2}\Big)$, which we can compute and store at each layer. Thus, the general recursive form of the partial derivatives of the loss with respect to the node embedding and weight matrix at each layer are:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}^{l-1}} = \Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \mathbf{X}^{l}}\Big)\Big(\mathbf{W}^{l-1}\Big)^\top \tag{28}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{l-1}} = \Big(\mathbf{X}^{l-1}\Big)^\top\Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \mathbf{X}^{l}}\Big) \tag{29}$$

where the initial condition is $\frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\hat{L+1}}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}}$, and we can compute the partials for layer L down to 1.

3.6 Complexity

At each layer, the three operations we need to compute are:

$$\mathbf{D} = \Big(\mathbf{A}'\Big)^\top\Big(\frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\hat{\mathbf{l+1}}}}\Big), \tag{30}$$

$$\Big(\mathbf{X}^{l}\Big)^\top\Big(\mathbf{D}\Big), \tag{31}$$

$$\Big(\mathbf{D}\Big)\Big(\mathbf{W}^{l}\Big)^\top. \tag{32}$$

Since $\frac{\partial L}{\partial \hat{\mathbf{Y}}}$ is of size $(N \times F)$, $\mathbf{A}'$ is $(N \times N)$, $\mathbf{X}^l$ is $(N \times F)$, and $\mathbf{W}^l$ is $(F \times F)$, these operations result in the following time complexity:

$$O(NF^2 + N^2F) \tag{33}$$

Again noting that each multiplication with A' is a sparse multiplication, we have:

$$O(L|E|F^2 + LN^2F) \tag{34}$$

3.7 Stochastic Gradient Descent

GNN convergence is improved with SGD, with a bath $\mathcal{B} \subseteq [N]$; $\mathcal{B}$ is a batch of node indexes. Let $b = |\mathcal{B}|$ So we have:

$$\mathcal{L} = \frac{1}{b}\sum_{i \in \mathcal{B}} \nabla loss(y_i, \hat{y_i}) \tag{35}$$

3.8 Summary

	Dense	Sparse
Forward Time	$LN^2F + LNF^2$	$LEF + LNF^2$
Forward Space	$N^2 + LF^2 + LNF$	$E + LF^2 + LNF$
Backward Time	$LN^2F + LNF^2$	$LEF + LNF^2$
Backward Space	$N^2 + LF^2 + LNF$	$E + LF^2 + LNF$

4 Self Attention

In the transformer network and graph attention networks, instead of using the original adjacency matrix $\boldsymbol{A}'$, we instead use a parameterized matrix $\boldsymbol{\alpha}$, which indicates edge importance for each update.

$$\mathbf{X}^{l+1} = \sigma(\boldsymbol{\alpha}\mathbf{X}^l\mathbf{W}_V^l) \tag{36}$$

4.1 Forward Pass

Specifically, the update steps for self attention are as follows:

$$\begin{aligned}
\mathbf{Q}^l &= \mathbf{X}^l\mathbf{W}_Q^l \\
\mathbf{K}^l &= \mathbf{X}^l\mathbf{W}_K^l \\
\mathbf{V}^l &= \mathbf{X}^l\mathbf{W}_V^l \\
\boldsymbol{\alpha}^{l'} &= \mathbf{Q}^l{\mathbf{K}^l}^\top \\
\boldsymbol{\alpha}^l &= \text{softmax}(\boldsymbol{\alpha}^{l'}) \\
\mathbf{X}^{l+1} &= \boldsymbol{\alpha}^l\mathbf{V}^l
\end{aligned}$$

where each $\mathbf{W}^l$ is an $F \times F$ matrix, $\mathbf{X}^l$ is $N \times F$, and $\boldsymbol{\alpha}^l$ is $N \times N$.

At each layer l, we need to compute $\mathbf{X}^l\mathbf{W}_Q^l$, $\mathbf{X}^l\mathbf{W}_K^l$, and $\mathbf{X}^l\mathbf{W}_V^l$, which each take O($NF^2$) time and O($NF + F^2$) space. Unlike in the GCN case, we also now need to compute $\mathbf{Q}^l{\mathbf{K}^l}^\top$ at each layer in order to get $\boldsymbol{\alpha}^{l'}$. This operation takes O(N^2F) time and O($N^2 + NF$) space. Finally, computing $\boldsymbol{\alpha}^l\mathbf{V}^l$ takes O(N^2F) time and O($N^2 + NF + F^2$) space. These computations result in a per layer time complexity of O($N^2F + NF^2$) and space complexity of O($N^2 + NF + F^2$). All of these operations are computed at each layer, leading to final time complexity of O($LN^2F + LNF^2$) and space complexity of O($LN^2 + LNF + LF^2$).

4.2 Backward Pass

4.3 Summary

	Dense	Sparse
Forward Time	$LN^2F + LNF^2$	$LEF + LNF^2$
Forward Space	$LN^2 + LF^2 + LNF$	$LE + LF^2 + LNF$
Backward Time	$LN^2F + LNF^2$	
Backward Space	$LN^2 + LF^2 + LNF$	

References

[1] Deep graph library (dgl), 2019. Accessed: 2019-06-06.

[2] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[3] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[4] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[5] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Towards efficient large-scale graph neural network computing. *arXiv preprint arXiv:1810.08403*, 2018.

[6] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch. *Computer software. Vers. 0.3*, 1, 2017.

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[8] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*, pages 937–950, 2018.