BackProp GCN

End-to-End Training Through Differentiable Ranking

Ian Bezerra

2025

1 The Problem

Normal GCN training uses frozen features! Normally from a vision model like ViT or Dino...

Here I will put some code snippets of the idea of making the whole process fit into one backward call and be able to run an optimizer like AdamW through the whole GCN into the Extractor!

1.1 How We Normally Do Things

```
# Extract features...
features = dinov2(images)  # No gradients!
features = features.detach()

# Compute rankings...
rankings = compute_rankings(features)

# Train GCN on FIXED graph!!
for epoch in range(100):
    loss = gcn(features, rankings)
    loss.backward()  # Only updates GCN, NOT DINOv2!
```

The key point here is that the Extractor doesn't learn anything. The features might not be optimal for the GCN task!

2 The Idea: Differentiable Ranking

2.1 Step 1: Compute Similarity (Differentiable)

```
# Normalize features
features_norm = F.normalize(features, p=2, dim=1)

# Cosine similarity matrix
similarity = features_norm @ features_norm.T
```

2.2 Step 2: Apply Soft Top-K (Differentiable)

```
# Apply temperature and softmax
soft_weights = F.softmax(similarity / temperature, dim=1)

# Get top-k indices
_, rankings = torch.topk(similarity, k=10)
```

```

# Extract soft weights for top-k neighbors
top_k_weights = soft_weights.gather(1, rankings)
```

We use `softmax` instead of hard `topk`.

2.3 Step 3: Build Graph

```
# Create edges from rankings
for i in range(N):
    for j in range(k):
        neighbor = rankings[i, j]
        edges.append([i, neighbor])
```

The edge structure depends on rankings, which depend on features!

2.4 Step 4: GCN Forward

```
# Features flow through GCN
x = features  # [N, D] - has gradients!
x = gcn_layer1(x, edges)
x = relu(x)
x = gcn_layer2(x, edges)
output = log_softmax(x)
```

Everything is differentiable.

2.5 Step 5: Backward Pass

```
loss = nll_loss(output[train_mask], labels[train_mask])
loss.backward()

# Gradients flow backward through:
# loss -> output -> gcn_layer2 -> gcn_layer1 -> features
#      -> soft_weights -> similarity -> features_norm
#      -> features -> dinov2!
```

3 End-to-End Training Loop

```
# Optimizer includes BOTH GCN and DINOv2
optimizer = Adam(gcn.parameters() + dinov2.parameters())

for epoch in range(200):
    # 1. Extract features (with gradients!)
    features = dinov2(images)  # Gradients enabled!

    # 2. Compute differentiable rankings
    similarity = features_norm @ features_norm.T
    soft_weights = softmax(similarity / temperature)
    rankings, top_k_weights = topk(similarity, soft_weights, k=10)

    # 3. Build graph from rankings
    edges = build_edges(rankings)

    # 4. Forward through GCN
```

```
    output = gcn(features, edges)

    # 5. Compute loss
    loss = nll_loss(output[train_mask], labels)

    # 6. Backward (updates BOTH GCN and DINOv2!)
    loss.backward()

    # 7. Update parameters
    optimizer.step()  # Updates both GCN AND DINOv2!
```

4 Math: Differentiable Top-K Selection

4.1 The Problem with Hard Top-K

Traditional ranking uses discrete operations:

```
indices = argsort(scores)[:k]
```

This has **zero gradient everywhere**:

$$\frac{\partial \text{topk}(x)}{\partial x} = 0 \tag{1}$$

No gradients means no backpropagation!

4.2 Our Solution: Soft Top-K with Softmax

Instead, we use a **soft approximation** via temperature-scaled softmax:

4.2.1 Step 1: Compute Similarity Matrix

For features $F \in \mathbb{R}^{N \times D}$:

$$S_{ij} = \frac{F_i \cdot F_j}{\|F_i\| \|F_j\|} \tag{2}$$

This is the **cosine similarity** between feature vectors.

4.2.2 Step 2: Apply Temperature-Scaled Softmax

For each node i, compute soft weights over all nodes:

$$w_{ij} = \frac{\exp(S_{ij}/\tau)}{\sum_{l=1}^{N} \exp(S_{il}/\tau)} \tag{3}$$

Where:

- τ is the **temperature** parameter
- Lower $\tau \to$ sharper distribution (closer to hard selection)
- Higher $\tau \to$ smoother distribution

4.2.3 Step 3: Select Top-K Neighbors

Get indices of top-k similarities:

$$\mathcal{N}_k(i) = \text{topk}(S_i, k) \tag{4}$$

Extract corresponding soft weights:

$$\tilde{w}_{ij} = w_{ij}, \quad \forall j \in \mathcal{N}_k(i) \tag{5}$$

These weights $\tilde{w}_{ij}$ are **differentiable**!

4.3 Why This Works: The Gradient

The softmax function has well-defined gradients:

$$\frac{\partial w_{ij}}{\partial S_{il}} = \frac{1}{\tau} w_{ij}(\delta_{jl} - w_{il}) \tag{6}$$

Where δ_{jl} is the Kronecker delta.
This means:

- **Non-zero gradients** flow backward through the softmax
- Gradients propagate from $w_{ij} \to S_{ij} \to F_i \to$ DINOv2
- The entire pipeline remains differentiable!

4.4 Temperature Parameter Effect

The temperature τ controls the approximation quality:

$$\lim_{\tau \to 0} w_{ij} = \begin{cases} 1 & \text{if } j = \arg\max_l S_{il} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

In practice:

- $\tau = 0.1$: Sharp, almost discrete (gradients can vanish)
- $\tau = 1.0$: Balanced (recommended)
- $\tau = 10.0$: Very smooth (weak selection)

4.5 Complete Forward Pass Formula

For node i with features f_i:

$$S_i = f_i \cdot F^T / (\|f_i\| \|F\|) \quad \text{(similarity)} \tag{8}$$

$$w_i = \text{softmax}(S_i / \tau) \quad \text{(soft weights)} \tag{9}$$

$$\mathcal{N}_k(i) = \text{topk}(S_i, k) \quad \text{(neighbor indices)} \tag{10}$$

$$\tilde{w}_i = w_i[\mathcal{N}_k(i)] \quad \text{(top-k weights)} \tag{11}$$

All operations are differentiable with respect to f_i!

4.6 Backward Pass (Chain Rule)

Given loss $\mathcal{L}$, gradients flow:

$$\frac{\partial \mathcal{L}}{\partial f_i} = \sum_{j \in \mathcal{N}_k(i)} \frac{\partial \mathcal{L}}{\partial \tilde{w}_{ij}} \frac{\partial \tilde{w}_{ij}}{\partial w_{ij}} \frac{\partial w_{ij}}{\partial S_{ij}} \frac{\partial S_{ij}}{\partial f_i} \tag{12}$$

Every term is **non-zero and computable!**