Fine-tuning a Pretrained Graph Neural Network for Large-Scale Graph Data Analysis

Executive Summary

This report details the process, and challenges encountered while fine-tuning a pretrained Graph Neural Network (GNN) model for large-scale graph data analysis. The task involved adapting a model with pretrained weights from the SNAP Stanford repository to a new dataset from the AdaGCN_TKDE project. Throughout the process, numerous obstacles were overcome, particularly regarding memory management and computational efficiency. This report showcases the problem-solving approach, technical adaptations, and resilience required to complete the task successfully. Please find the complete code at Github.

1. Task Overview

The primary objective was to fine-tune a pretrained GNN model on a new, large-scale graph dataset. This task required:

1. Loading and adapting the pretrained model architecture
2. Processing and preparing the new dataset
3. Implementing an efficient fine-tuning pipeline
4. Overcoming significant memory and computational constraints

2. Methodology and Implementation

2.1 Initial Approach and Data Preprocessing

The project began with a focus on adapting the pretrained Graph Isomorphism Network (GIN) from the SNAP Stanford team to our large-scale graph dataset. My initial efforts centered on:

1. Converting .mat files into PyTorch Geometric Data objects
2. Adjusting feature dimensions of node and edge attributes
3. Creating appropriate data loaders for batch processing

I initially believed that correctly formatting the input data would allow us to use the existing model structure with minimal modifications. However, this approach quickly revealed the complexity of our task and the need for more comprehensive solutions.

2.2 Model Architecture and Adaptations

As I progressed, it became evident that significant modifications to the model architecture were necessary. Key adaptations included:

1. GINConv Updates:
    - Ensuring edge_index is always a LongTensor

    - Modifying self-loop handling for compatibility with the new data structure

2. GNN Class Enhancements:
    - Implementing gradient checkpointing in the forward pass
    - Adding a batch_indices parameter for selective node representation return
3. Development of MemoryEfficientGNN:
    - Redesigning the GNN architecture to handle large-scale graphs without exhausting computational resources
    - Incorporating task-specific features to improve model performance

These modifications were crucial in addressing the unique challenges posed by the dataset and computational constraints.

2.3 Overcoming Memory Management Challenges

The extensive memory requirements of the large-scale graph data presented a significant hurdle. I implemented several innovative approaches to address this:

1. CPU Offloading: Developed a custom *CPUOffloadGraphSampler* to keep the majority of data on CPU, transferring only small batches to GPU as needed.
2. Gradient Checkpointing: Utilized PyTorch's checkpoint functionality to trade computation time for reduced memory usage during backpropagation.
3. Mixed Precision Training: Employed NVIDIA's Automatic Mixed Precision (AMP) to optimize memory usage and potentially accelerate training.
4. Custom Data Sampling and Batching: Developed strategies to process subsets of the graph in manageable chunks, balancing between memory constraints and computational efficiency.

2.4 Training Process Optimization

To manage computational demands within the available resources, I implemented several optimization techniques:

1. Batch Size Adjustment: Utilized a small batch size of 32 to minimize per-iteration memory requirements.
2. Gradient Accumulation: Implemented accumulation over 256 steps before optimization, effectively increasing batch size without additional memory usage.
3. Learning Rate Tuning: Reduced the learning rate to 0.0001 to accommodate the larger effective batch size and stabilize training.

2.5 Iterative Problem-Solving Process

The development process was characterized by continuous learning and adaptation:

1. Initial Implementation and Error Analysis: Attempted to directly fine-tune the pretrained model, resulting in out-of-memory errors. Carefully examined error messages (e.g., CUDA out of memory errors) to guide optimization efforts.
2. Memory Profiling: Conducted in-depth analysis of memory usage patterns to identify bottlenecks and inform our optimization strategy.
3. Incremental Optimizations: Gradually implemented memory-saving techniques, rigorously testing after each addition to ensure stability and performance improvements.
4. Model Architecture Refinement: Iteratively modified the GNN and GINConv implementations, constantly balancing between performance enhancement and memory efficiency.

Please refer to the Appendix for screenshots of errors.

3. Final Workflow and Implementation

The final implementation of the fine-tuning process incorporated several key components to address the challenges encountered:

1. **Data Preprocessing and Loading**:
   - Implemented a custom *load_graph_data* function to efficiently convert .mat files into PyTorch Geometric Data objects.
   - Applied feature normalization using *T.NormalizeFeatures()* to standardize input data.
2. **Memory-Efficient Model Architecture**:
   - Developed *MemoryEfficientGNN*, an extension of the original GNN model, incorporating gradient checkpointing for reduced memory usage.
   - Modified the forward pass to use checkpointing, trading computation time for memory efficiency.
3. **CPU Offloading and Batch Processing**:
   - Created *CPUOffloadGraphSampler* to keep large portions of data on CPU and transfer only necessary batches to GPU.
   - Implemented a custom batching strategy to process subsets of the graph, balancing between memory constraints and computational efficiency.
4. **Training Optimizations**:

- Utilized a small batch size (32) combined with gradient accumulation (256 steps) to effectively increase the batch size without additional memory usage.
- Employed mixed precision training using NVIDIA's Automatic Mixed Precision (AMP) to optimize memory usage and potentially accelerate training.

5. **Iterative Training Process**:
   - Conducted training over 3 epochs, with each epoch processing the entire dataset in batches.
   - Implemented a tqdm progress bar for real-time monitoring of training progress.
6. **Evaluation**:
   - Used the same CPU offloading and batching strategy for model evaluation to maintain consistency and manage memory efficiently.
   - Calculated accuracy and weighted F1 score to assess model performance on the test dataset.

4. Results and Discussion

4.1 Training Outcomes

Despite severe computational constraints, the model was successfully fine-tuned for 3 epochs. Each epoch required approximately 24 minutes, underscoring the computational intensity of the task.

4.2 Performance Metrics

After fine-tuning, the model achieved:

- Accuracy: 25.3162%
- Weighted F1 Score: 0.1023

While these metrics may seem modest, they represent a significant achievement given the extreme memory constraints and limited training time. The successful completion of the fine-tuning process demonstrates the effectiveness of the implemented memory-saving techniques and model adaptations.

4.3 Resource Utilization

The fine-tuning process pushed the available hardware to its limits:

- Utilized nearly all of the 4GB GPU memory
- Required extensive CPU-GPU data transfer management
- Demanded significant computation time even for a limited number of epochs

5. Conclusion and Future Directions

This project demonstrated the ability to adapt and fine-tune a complex GNN model on a challenging, large-scale dataset under severe resource constraints. Key achievements include:

1. Successfully modifying a pretrained GNN architecture for a new task
2. Implementing advanced memory management techniques to overcome hardware limitations
3. Developing a robust fine-tuning pipeline capable of handling large-scale graph data

Future improvements could include:

- Exploring distributed training methods to manage computational demands
- Further optimizing the model architecture for memory efficiency
- Investigating techniques to accelerate the training process, allowing for more epochs within time constraints

Appendix

Screenshots of Errors/blockers

Figure 1. This error occurred during the initial attempts to fine-tune the model. It highlights the severe memory constraints faced when dealing with large-scale graph data on limited GPU resources. This error prompted the development of memory-efficient techniques.

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[12], line 8
      6 optimizer.zero_grad()
      7 with autocast():
----> 8     output = checkpointed_forward(batch.x, batch.edge_index, batch.edge_attr)
      9     loss = criterion(output, y_train)
     10 scaler.scale(loss).backward()

Cell In[11], line 2
      1 def checkpointed_forward(*inputs):
----> 2     return checkpoint(model, *inputs)

File c:\Users\akalps\anaconda3\Lib\site-packages\torch\_compile.py:24, in _disable_dynamo.<locals>.inner(*args, **kwargs)
     20 @functools.wraps(fn)
     21 def inner(*args, **kwargs):
     22     import torch._dynamo
---> 24     return torch._dynamo.disable(fn, recursive)(*args, **kwargs)

File c:\Users\akalps\anaconda3\Lib\site-packages\torch\_dynamo\eval_frame.py:489, in _TorchDynamoContext.__call__.<locals>._fn(*args, **kwargs)
    487     dynamo_config_ctx.__enter__()
    488 try:
--> 489     return fn(*args, **kwargs)
    490 finally:
    491     set_eval_frame(prior)
...
-> 1418     ret = func(*args, **kwargs)
   1419     if func in get_default_nowrap_functions():
   1420         return ret

RuntimeError: Sizes of tensors must match except in dimension 1. Expected size 31158 but got size 2 for tensor number 1 in the list.
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 2. This runtime error was encountered due to a mismatch in tensor sizes, specifically in the edge attribute dimensions. It led to the realization that the GINConv layer needed modifications to handle our specific data format correctly.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 46
     44     print(f"x shape: {batch.x.shape}")
     45     print(f"edge_index shape: {batch.edge_index.shape}")
---> 46     output = model(batch.x.to(device), batch.edge_index.to(device))
     47     loss = criterion(output, batch.y.to(device))
     48 scaler.scale(loss).backward()

File c:\Users\akalps\anaconda3\Lib\site-packages\torch\nn\modules\module.py:1511, in Module._wrapped_call_impl(self, *args, **kwargs)
   1509     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1510 else:
-> 1511     return self._call_impl(*args, **kwargs)

File c:\Users\akalps\anaconda3\Lib\site-packages\torch\nn\modules\module.py:1520, in Module._call_impl(self, *args, **kwargs)
   1515 # If we don't have any hooks, we want to skip the rest of the logic in
   1516 # this function, and just call forward.
   1517 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_hooks or self._forward_pre_hooks
   1518         or _global_backward_pre_hooks or _global_backward_hooks
   1519         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1520     return forward_call(*args, **kwargs)
   1522 try:
   1523     result = None

TypeError: GNN.forward() missing 1 required positional argument: 'edge_attr'
```

Figure 3. This attribute error occurred during data loading, indicating incompatibility between the pretrained model's expectations and our dataset structure. It necessitated the development of custom data loading and preprocessing techniques.

```
Using device: cuda
Loading datasets...
Loading pre-trained model...
Starting training...
Epoch 1/25:   0%|          | 0/19 [00:00<?, ?it/s]

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[3], line 89
     87 total_loss = 0
     88 sampler = GraphSampler(train_data, batch_size)
---> 89 for batch_indices, batch_edge_index, batch_edge_attr in tqdm(sampler, desc=f"Epoch {epoch+1}/{num_epochs}"):
     90     optimizer.zero_grad()
     91     with autocast():

File c:\Users\akalps\anaconda3\Lib\site-packages\tqdm\std.py:1178, in tqdm.__iter__(self)
   1175 time = self._time
   1177 try:
-> 1178     for obj in iterable:
   1179         yield obj
   1180         # Update and possibly print the progressbar.
   1181         # Note: does not call self.update(1) for speed optimisation.

Cell In[3], line 56
     53 # Remap node indices to ensure they are consecutive
     54 node_map = {int(idx): i for i, idx in enumerate(batch_indices)}
     55 batch_edge_index = torch.tensor([[node_map[int(idx)] for idx in batch_edge_index[0]],
---> 56                                  [node_map[int(idx)] for idx in batch_edge_index[1]]],
     57                                 dtype=torch.long)
     59 yield batch_indices, batch_edge_index, batch_edge_attr
...
---> 56                                  [node_map[int(idx)] for idx in batch_edge_index[1]]],
     57                                 dtype=torch.long)
     59 yield batch_indices, batch_edge_index, batch_edge_attr

KeyError: 2382
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```