Performance Analysis of MedNet for Temporal Medical Imaging

A Deep Dive into Memory and Compute Scaling

Tymo van Rijn

May 2026

Abstract

This analysis investigates the performance characteristics of MedNet, Idiap's in-house medical imaging framework, when scaling from single-frame to multi-frame temporal data. Using the angioreport dataset as a case study, we explore why processing 12+ sequential frames leads to out-of-memory (OOM) errors on modern GPUs. Through algorithmic complexity analysis and concrete memory calculations, we identify the bottlenecks and compare MedNet's approach with alternatives like MONAI. The goal: provide actionable insights for the MedAI team to optimize MedNet for temporal workloads.

1 Introduction: The Problem We're Solving

Picture this: you have a medical imaging pipeline that works great on single frames. Your Vision Transformer (ViT) classifies angiography images with solid accuracy. But medicine doesn't happen in snapshots—blood flows, contrast agents travel, and pathology reveals itself over time. You need *temporal* information.

So you think: "Let's just use 12 frames instead of 1, add a GRU to capture the temporal patterns, and we're done." You hit run, and...

```
CUDA out of memory.  Tried to allocate 2.4 GB...
```

Welcome to the world of GPU memory management. This analysis will take you through exactly *why* this happens, *where* the memory goes, and *what* can be done about it.

1.1 The Setup

We're working with:

- **Model**: ViT-Small (patch size 16, image size 224×224) with RetFound Green pretrained weights
- **Architecture**: ViT extracts features per frame → GRU processes the sequence → classification head
- **Data**: Angioreport dataset, 12 frames per examination, variable time intervals
- **Hardware**: RTX 3090 (24 GB VRAM), SLURM cluster with Tesla GPUs
- **Observation**: Batch size 16 with 20+ frames causes OOM, even with gradient checkpointing

2 Sub-Questions

To systematically analyze this problem, we need to answer the following questions:

1. **Where does GPU memory actually go during training?**
   What are the components that consume VRAM, and how big is each piece?
2. **How does memory scale when we add more frames?**
   Is it linear? Quadratic? What's the Big-O complexity?
3. **Why does OOM happen at batch=16, frames=20?**
   Can we calculate the exact memory requirement and compare it to the 24 GB limit?
4. **What is the compute cost of temporal processing?**
   How much slower is training with multiple frames?
5. **How do alternatives (like MONAI) handle this?**
   Are there smarter approaches we're missing?
6. **What are MedNet's current limitations for temporal data?**
   Where would changes need to happen in the codebase?

3 Memory Anatomy: Where Does It All Go?

Before we can optimize anything, we need to understand what's eating our precious VRAM. During neural network training, GPU memory is consumed by four main components:

3.1 Component 1: Model Parameters

The actual weights of the network. For ViT-Small:

$$\text{Parameters} \approx 22 \text{ million} \tag{1}$$

$$\text{Memory (FP32)} = 22 \times 10^6 \times 4 \text{ bytes} = \mathbf{88} \text{ MB} \tag{2}$$

That's surprisingly small! The ViT-Small is quite efficient. For comparison, ViT-Large would be $\sim$1.2 GB.

3.2 Component 2: Optimizer States

Adam (and AdamW) stores two additional values per parameter: the momentum (first moment) and variance (second moment). This triples our parameter memory:

$$\text{Optimizer states} = 2 \times \text{Parameters} \tag{3}$$

$$= 2 \times 88 \text{ MB} = \mathbf{176} \text{ MB} \tag{4}$$

3.3 Component 3: Gradients

During backward pass, we store gradients for each parameter:

$$\text{Gradients} = 1 \times \text{Parameters} = \mathbf{88} \text{ MB} \tag{5}$$

3.4 Component 4: Activations (The Silent Killer)

Here's where things get interesting. During forward pass, we need to save intermediate activations to compute gradients during backward pass. This is called the "activation memory" and it scales with:

- Batch size (B)
- Number of frames (T)
- Model architecture (layers, hidden dimensions, attention heads)

For a single image through ViT-Small, let's trace the memory:
Step 1: Patch Embedding

$$\text{Patches} = \frac{224}{16} \times \frac{224}{16} = 14 \times 14 = 196 \tag{6}$$

$$\text{Sequence length} = 196 + 1 \text{ (CLS token)} = 197 \tag{7}$$

$$\text{Hidden dimension} = 384 \tag{8}$$

Step 2: Per-Layer Activations
Each of the 12 transformer layers stores:

- Query, Key, Value projections: $3 \times 197 \times 384 = 227,016$ floats
- Attention weights: 6 heads $\times$ $197 \times 197 = 232,854$ floats
- MLP intermediate: $197 \times 1536 = 302,592$ floats ($4\times$ expansion)
- Layer outputs and residuals: $\sim 197 \times 384 = 75,648$ floats

Per layer, roughly: $\sim 840,000$ floats $\times 4$ bytes $\approx$ **3.4 MB per layer**.
With 12 layers: $12 \times 3.4 \approx$ **40 MB per image**.

3.5 The Scaling Formula

Now we can write the total memory formula:

$$\boxed{M_{\text{total}} = M_{\text{params}} + M_{\text{optimizer}} + M_{\text{gradients}} + B \times T \times M_{\text{activations}}} \tag{9}$$

Where:

- $M_{\text{params}} = 88$ MB (fixed)
- $M_{\text{optimizer}} = 176$ MB (fixed)
- $M_{\text{gradients}} = 88$ MB (fixed)
- $M_{\text{activations}} \approx 40$ MB per image
- B = batch size
- T = frames per sample

The activation memory grows as $O(B \times T)$—linear in both batch size and frame count.

4 The OOM Calculation: Why Batch=16, Frames=20 Fails

Let's plug in the numbers for the failing case:

$$M_{\text{fixed}} = 88 + 176 + 88 = 352 \text{ MB} \tag{10}$$

$$M_{\text{activations}} = 16 \times 20 \times 40 \text{ MB} = 12,800 \text{ MB} = \mathbf{12.5} \text{ GB} \tag{11}$$

But wait, we're not done. We haven't counted:

- **GRU memory**: Processing 20 timesteps with hidden size ∼384 adds another ∼200 MB per batch
- **CUDA context**: PyTorch and CUDA reserve ∼1-2 GB for internal operations
- **Memory fragmentation**: Real allocations are rarely perfectly packed, add ∼20% overhead
- **Peak memory spikes**: During backward pass, temporary tensors can spike memory by 1.5-2×

Realistic estimate:

$$M_{\text{realistic}} \approx (352 + 12,800 + 200) \times 1.2 + 1,500 \tag{12}$$

$$\approx 16,022 + 1,500 = \mathbf{17.5} \text{ GB} \tag{13}$$

This should fit in 24 GB... so why OOM?

4.1 The Hidden Culprit: Peak Memory During Backward

The calculation above is for *steady-state* memory. During the backward pass, PyTorch:

1. Keeps all forward activations in memory
2. Creates intermediate gradient tensors
3. Sometimes materializes the full attention matrix for gradient computation

For self-attention, the gradient of the attention operation can temporarily require storing the full batch × heads × seq × seq tensor:

$$M_{\text{attn\_grad}} = B \times T \times \text{heads} \times \text{seq}^2 \times 4 \text{ bytes} \tag{14}$$

$$= 16 \times 20 \times 6 \times 197^2 \times 4 \tag{15}$$

$$= 16 \times 20 \times 6 \times 38,809 \times 4 \tag{16}$$

$$\approx 2.98 \text{ GB (peak spike!)} \tag{17}$$

Add this spike to our base estimate, and we hit **∼20.5 GB**—dangerously close to the limit. Any additional overhead pushes us over.

4.2 What About Gradient Checkpointing?

Gradient checkpointing trades compute for memory by not storing all activations—instead, it recomputes them during backward pass. Typically this reduces activation memory by 3-5×.

With checkpointing:

$$M_{\text{activations}} \approx \frac{12,800}{4} = 3,200 \text{ MB} \tag{18}$$

But checkpointing doesn't help with the attention gradient spike, and it doesn't reduce optimizer states or CUDA overhead. If you're still hitting OOM with checkpointing, the spike is likely the culprit.

5 Compute Complexity: How Much Slower?

Memory isn't the only concern—training time matters too. Let's analyze the computational complexity.

5.1 ViT Forward Pass

The dominant operation in ViT is self-attention, with complexity:

$$O(\text{seq}^2 \times d) = O(197^2 \times 384) \approx O(14.9 \text{ million ops per layer}) \tag{19}$$

For 12 layers and $B \times T$ images:

$$\text{Total ops} = O(B \times T \times 12 \times 197^2 \times 384) \tag{20}$$

This is **linear in** T—good news! Doubling frames doubles compute time (roughly).

5.2 GRU Sequential Bottleneck

The GRU introduces a sequential dependency that can't be parallelized across timesteps:

$$\text{GRU time} = O(T \times d_{\text{hidden}}^2) \tag{21}$$

For $T = 12$ and $d = 384$: this is small compared to ViT, but it serializes the computation.

5.3 Real-World Estimate

If single-frame training takes time t_1, then T-frame training takes approximately:

$$t_T \approx T \times t_1 + \text{GRU overhead} \tag{22}$$

For 12 frames: expect ∼12-15× slower training per sample (partially offset by fewer samples needed per epoch if each sample is more informative).

6 Comparison: How Do Others Handle This?

6.1 MONAI's Approach

MONAI (Medical Open Network for AI) is the most popular alternative framework. For temporal/volumetric data, MONAI offers:

1. **Sliding window inference**: Process chunks of frames, aggregate results
2. **Patch-based training**: Don't process full volumes at once

3. **Built-in 3D models**: Networks designed for volumetric data (3D UNet, etc.)

4. **CacheDataset**: Smart caching with transforms applied lazily

Key difference: MONAI expects volumetric data as a first-class citizen. The data loaders, transforms, and models are designed around 3D/4D tensors from the start.

6.2 Other Strategies in the Wild

Strategy	Memory Impact	Trade-off
Frame sampling	Reduce T	Lose temporal resolution
Feature caching	Store ViT outputs, train GRU separately	Two-stage training
Mixed precision (FP16)	$\sim$50% reduction	Minor accuracy impact
Gradient accumulation	Virtual larger batch	Same memory, slower
Model parallelism	Split across GPUs	Communication overhead

Table 1: Common strategies for handling memory constraints

6.3 The Most Practical Solution

For our future setup (ViT + GRU, 12 frames), the most effective approaches are:

1. **Feature caching**: Run ViT inference once per frame, cache the 384-dim features. Train only the GRU on cached features. This reduces activation memory from 12.5 GB to $\sim$100 MB.

2. **Mixed precision**: Use `torch.cuda.amp` for automatic FP16. Halves activation memory with minimal accuracy loss.

3. **Frame sampling during training**: Use 6-8 frames during training (randomly sampled), full 12 during inference.

7 MedNet's Current Limitations

Looking at MedNet's codebase, several architectural decisions make temporal data challenging:

7.1 Single-Sample Assumption

The `datamodule.py` loads samples as dictionaries with `batch["image"]` containing a single tensor. There's no built-in concept of "a sample is a sequence of frames."

7.2 No Temporal Model Support

The classification models in `mednet.models.classify` expect input shape `(B, C, H, W)`. Temporal models need `(B, T, C, H, W)` or similar.

7.3 Missing Memory Optimization Tools

MedNet doesn't expose:

- Gradient checkpointing configuration in training configs
- Mixed precision training flags
- Feature caching utilities

These would need to be added or configured manually through Lightning's trainer.

8 Summary of Findings

1. **Memory scales as** $O(B \times T)$ for activation memory, which dominates GPU usage.
2. **The OOM at batch=16, frames=20** is caused by activation memory (∼12.5 GB) plus peak spikes during backward pass (∼3 GB), plus system overhead (∼2 GB), totaling ∼20+ GB with fragmentation.
3. **Gradient checkpointing helps but doesn't solve it**—the attention gradient spike remains.
4. **Feature caching is the most effective solution**—pre-extract ViT features, train only the temporal model (GRU) on cached features.
5. **MedNet needs architectural changes** to natively support temporal data: new data loaders, temporal model wrappers, and exposed memory optimization flags.
6. **MONAI handles this better by design**—but switching frameworks has its own costs. Extending MedNet may be more practical for Idiap's workflow.

What's Next?

This analysis provides the "why" behind the memory problems. The advice document will translate these findings into concrete recommendations for the MedAI team—what to change, in what order, and what trade-offs to expect.