Technical Report 1: Analysis of a Computational Bottleneck in a Scalable, Learnable Graph Operator

Mohana Rangan Desigan

August 16, 2025

Abstract

This report documents the current status and primary technical challenge of the Jörmungandr-Semantica research program. The program's goal is to develop a learnable, anisotropic graph diffusion operator for unsupervised representation learning. We chronicle the systematic failure of several hand-crafted heuristics (based on curvature and community structure), which led to the development of a scalable, differentiable learning framework built in PyTorch. This framework replaces the intractable $O(N^3)$ differentiable eigendecomposition with a scalable $O(m|E|D)$ Chebyshev polynomial approximation. While this new architecture is theoretically sound and has resolved all prior bugs, its practical implementation on CPU has revealed a significant performance bottleneck. A single training epoch on the 11,314-node 20 Newsgroups graph takes several minutes, making iterative research infeasible. This report formalizes the architecture, pinpoints the source of the bottleneck, and frames the immediate research objective: to overcome this performance barrier via GPU acceleration.

1 Architectural Overview

The current system is a modular pipeline designed to learn an optimal anisotropic diffusion operator.

1.1 Core Components

1. **Graph Construction:** A k-NN graph $G = (V, E, W)$ is built from input embeddings using Faiss.
2. **Geometric Signal:** Forman-Ricci curvature, κ_{ij}, is computed for each edge.
3. **Learnable Operator:** A PyTorch `nn.Module`, the `DifferentiableChebyshevOperator`.
4. **Training Loop:** Optimizes the operator's parameters θ by minimizing a triplet margin loss.

1.2 The Scalable, Differentiable Forward Pass

The core of the system computes the action of the wavelet transform $X' = e^{-tL_{\text{aniso},\theta}}X$ via a Chebyshev polynomial approximation. The key computational step is a recurrence loop:

```
# Inside the Chebyshev recurrence loop:
Tk_X = 2 * torch.sparse.mm(L_rescaled, T1_X) - T0_X
```

2 The Performance Bottleneck: Diagnosis

2.1 Problem Statement

Despite being theoretically scalable, the practical performance on CPU is prohibitive. A single training epoch on the 20 Newsgroups dataset ($N \approx 11k, |E| \approx 118k, D = 384$) takes several minutes.

2.2 Analysis

The system appears to hang before the first epoch's progress is logged. The dominant computation is the Chebyshev recurrence, which involves $m = 30$ successive `torch.sparse.mm` operations. The bottleneck is attributed to a combination of factors:

- **Computational Cost:** Each epoch requires $\approx 5.4 \times 10^9$ floating point operations.
- **Memory Traffic:** Intermediate products are large, dense matrices ($(11314, 384) \approx 17.4$ MB each), leading to significant latency when repeatedly accessed in CPU RAM.
- **Framework Overhead:** PyTorch's CPU backend for sparse operations and its associated autograd machinery are less optimized than their dense or GPU (CUDA) counterparts.

3 Current Status and Path Forward

We are at a critical juncture where the project is a theoretical success but a practical bottleneck.

- **What Works:** The pipeline is architecturally sound. All previously identified bugs have been resolved.
- **What is Broken:** The performance on CPU is too slow for effective research.

The immediate research objective is to **overcome this computational bottleneck**.

1. **Primary Path: GPU Acceleration (Kaggle/Colab).** The most direct solution is to execute the existing PyTorch code on a platform with an NVIDIA GPU and a CUDA environment. The `torch.sparse.mm` operation is highly optimized for CUDA. This is expected to yield a 10-100x speedup. This will be the focus of the next experimental sprint.
2. **Secondary Path (Investigation): MPS Backend on Apple Silicon.** Our attempt to use the `mps` backend failed due to incomplete support for sparse tensors in PyTorch. We will continue to monitor the state of PyTorch MPS support.
3. **Tertiary Path (Investigation): JAX Re-implementation.** A mature JAX implementation remains a viable alternative, as its JIT compiler (`@jax.jit`) is exceptionally effective at optimizing sequential operations like the Chebyshev recurrence on both CPU and GPU.

This report concludes with the system in a "ready for acceleration" state. The next step is to execute the capstone experiment on a GPU-enabled platform to finally validate the performance of the learned anisotropic operator.