Proceedings of TerraBytes at the International Conference on Machine Learning 2025 1–14

Optimizing Cloud-to-GPU Throughput for Deep Learning With Earth Observation Data

Akram Zaytar* AKRAMZAYTAR@MICROSOFT.COM
Caleb Robinson DAVROB@MICROSOFT.COM
Girmaw Abebe Tadesse GTADESSE@MICROSOFT.COM
Gilles Quentin Hacheme GHACHEME@MICROSOFT.COM
Tammy Glazer TAMMY.GLAZER@MICROSOFT.COM
Anthony Ortiz ANTHONY.ORTIZ@MICROSOFT.COM
Rahul Dodhia RAHUL.DODHIA@MICROSOFT.COM
Juan M. Lavista Ferres JLAVISTA@MICROSOFT.COM
Microsoft AI for Good Research Lab

Abstract

Training deep learning models on petabyte-scale Earth Observation (EO) data requires separating compute resources from data storage. However, standard PyTorch data loaders cannot keep modern GPUs utilized when streaming `GeoTIFF` files directly from cloud storage. In this work, we benchmark `GeoTIFF` loading throughput from both cloud object storage and local SSD, systematically testing different loader configurations and data parameters. We focus on tile-aligned reads and worker thread pools, using Bayesian optimization to find optimal settings for each storage type. Our optimized configurations increase remote data loading throughput by 20× and local throughput by 4× compared to default settings. On three public EO benchmarks, models trained with optimized remote loading achieve the same accuracy as local training within identical time budgets. We improve validation IoU by 6–15% and maintain 85–95% GPU utilization versus 0–30% with standard configurations. Code is publicly available[1].

* Corresponding author.

1. https://github.com/microsoft/pytorch-cloud-geotiff-optimization

Keywords: Geospatial Deep Learning, Cloud-Native Training, Data Loader Optimization

1. Introduction

EO datasets have reached petabyte scale volumes (Wilkinson et al., 2024), yet training deep learning models on this data faces a fundamental bottleneck. When streaming data directly from cloud storage, multiple factors introduce latency that leaves GPUs severely underutilized. The alternative of downloading entire datasets locally is impractical at this scale, as it demands significant storage capacity and repeated downloads for each experiment.

This bottleneck stems from both data loading mechanisms and format characteristics. The `PyTorch DataLoader` (Paszke, 2019) was originally designed for local training scenarios with minimal access latency. Although it offers configuration options like worker processes, batch sizes, and prefetching to improve throughput, these assume fast local storage. At the same time, data format choices significantly impact remote access performance. Factors such as tiling block size, compression type and level,

and image dimensions all affect how efficiently patches can be extracted from remote files. When data resides in cloud storage, the combination of suboptimal loader configurations and format characteristics can consume over half of each training epoch (Mohan et al., 2020b), leaving GPUs idle while waiting for the next batch.

The machine learning community has developed several approaches to address remote data loading challenges. These solutions generally follow three patterns: custom file formats for efficient streaming, asynchronous loading mechanisms, and shared caching systems. FFCV (Leclerc et al., 2023) combines an efficient container format with asynchronous transfers to improve I/O throughput and GPU utilization. WebDataset (Aizman et al., 2019) shards samples into tar archives to support efficient sequential reads from cloud/object storage and has been used to scale large vision workloads. GPU-accelerated preprocessing/data-pipeline frameworks (e.g., NVIDIA DALI; see (Zolnouri et al., 2020)) aim to reduce host-side decoding and augmentation overhead by overlapping data preparation with training. Concurrent/cloud-aware loaders (Svogor et al., 2022; Mohan et al., 2020a) overlap network I/O across workers and coordinate caches to avoid redundant reads in distributed settings. These advances underscore the impact of optimized data pipelines, but they largely target natural image and video corpora.

While general approaches address key remote loading challenges, raster data present unique characteristics that require specialized solutions. Raster imagery is often stored in large compressed tiles (e.g., $\geq$ 10,000 $\times$ 10,000 pixels) from which training patches must be extracted via windowed reads or offline preprocessing. EO workflows increasingly adopt cloud-native formats such as Cloud Optimized `GeoTIFF`s (COG) (cog, 2019) and `Zarr` (Consortium et al., 2023), both of which expose internal tiling/chunking schemes that enable efficient range requests. In these formats, imagery is partitioned into fixed $k \times k$ pixel tiles or chunks (Figure 1). An aligned window read requires loading and decompressing exactly one tile; a mis-aligned read forces multiple tiles to be fetched and decompressed, inflating I/O and CPU costs. `GeoTIFF` supports multiple compression methods (e.g., `DEFLATE`, `LERC-ZSTD`, `LZW`) that trade transfer size against decode time. `Zarr` arrays integrate with the *Xarray*/`Dask` ecosystem, enabling concurrent chunk loading and direct feeding into `PyTorch DataLoader`s via tooling such as *xbatcher* (Jones et al., 2023).

However, systematic optimization for `GeoTIFF` workflows remains unexplored, despite most EO archives storing data in this format. While streaming optimization for `Zarr` has been explored (Jones et al., 2023), `GeoTIFF` poses unique challenges due to significant variability in internal structure, compression methods, and tiling schemes across different archives. Classical approaches are ineffective for `GeoTIFF` workflows. Caching systems fail because geospatial sampling exhibits negligible data reuse– a consequence of vast spatio-temporal domains and diverse sampling strategies (random or spatially distributed) commonly used in EO model training. Custom file formats are also impractical due to conversion limitations when using COG, though asynchronous loading remains applicable.

This paper addresses the gap through systematic optimization of both `PyTorch DataLoader` configurations and `GeoTIFF` format characteristics specifically for remote streaming. Our contributions include:

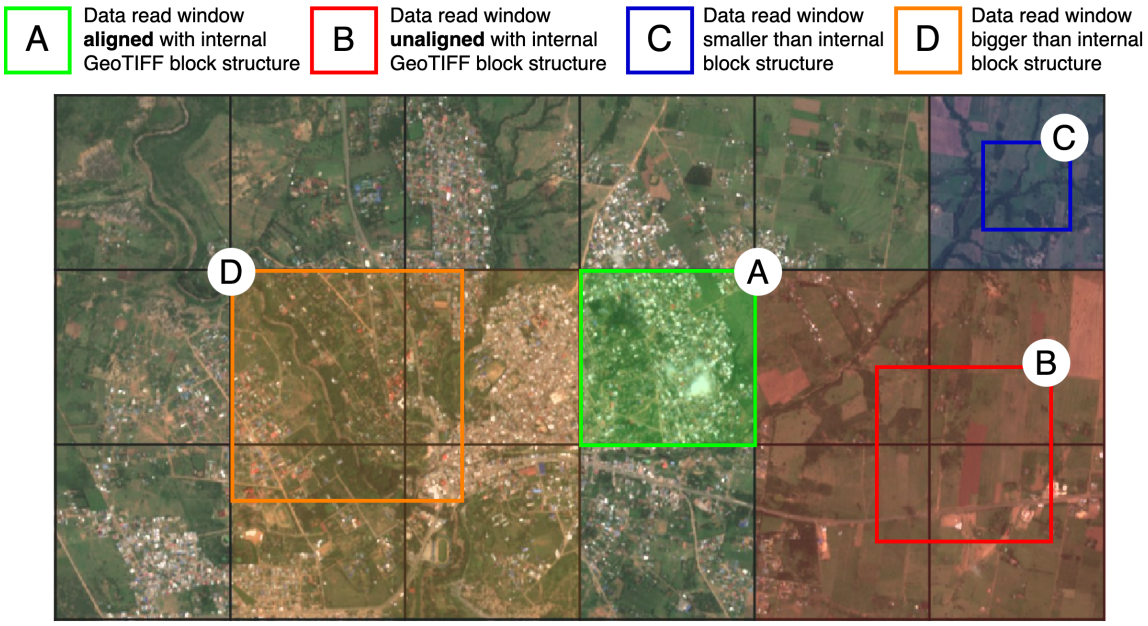

Figure 1: Sentinel-2 image with internal `GeoTIFF` block structure overlaid as black grid squares. Colored rectangles show four windowed read patterns and their data loading behavior (colored backgrounds). (**A**) shows an aligned read that matches block boundaries, efficiently loading only the requested data from a single block. (**B**) demonstrates a random window read of block size that intersects four blocks, requiring 4× more data transfer than needed. (**C**) represents sub-block reads that still force entire block loading, creating overhead for small requests. (**D**) shows large reads spanning multiple blocks, necessitating loading of all intersected blocks.

- A benchmarking framework using Bayesian optimization to identify optimal configurations for `GeoTIFF` loading, focusing on tile-aligned reads and worker thread pools.
- An optimized configuration achieving 20× higher remote throughput over baseline settings and 4× improvement for local reads.
- Empirical validation through EO segmentation benchmarks showing optimized remote loading matches local-disk training accuracy within a fixed time budget.

2. Experimental setup

2.1. Data preparation

Our goal is to simulate the real-world scenario of streaming raster patches from COGs hosted in Azure Cloud Storage. Since replicating an entire online archive for different compression types and levels is impractical, we address this by deliberately disabling caching to simulate fresh patch fetching and selecting a subset of images for our transforms and experimentation. We use Sentinel-2 (Drusch et al., 2012) multispectral satellite imagery, which provides global coverage at 10 meter resolution with a 5 day revisit

time. This imagery is widely used in environmental monitoring (Gorroño et al., 2023), agriculture (Segarra et al., 2020), and land cover classification tasks (Phiri et al., 2020), making it representative of typical EO deep learning workflows.

We download data through Microsoft's Planetary Computer SpatioTemporal Asset Catalogs API (Source et al., 2022), focusing on a 168 km^2 region centered on Nairobi, Kenya. We selected imagery from 2024 with cloud cover below 5%, resulting in 10 scenes for our benchmark dataset. From each scene, we extract four spectral bands ("B02", "B03", "B04", and "B08"), representing blue, green, red, and near-infrared, respectively.

To systematically evaluate how compression affects data loading throughput, we process each scene into 6 compression variants: uncompressed (`None`), `LZW` with predictor 2, `DEFLATE` at three different levels (i.e., `DEFLATE_1`, `DEFLATE_6`, and `DEFLATE_9`), and `LERC-ZSTD`. These methods represent different compression philosophies: `LZW` and `DEFLATE` provide varying speed-compression tradeoffs (where higher `DEFLATE` levels achieve better compression at the cost of slower encoding), while `LERC-ZSTD` offers an alternative approach optimized for numerical data. This allowed us to analyze the tradeoffs between file size and processing speed across various compression methods.

We store all variants as COGs with consistent internal tiling structure (512×512 pixel blocks). The resulting dataset consists of 60 GeoTIFF files (10 scenes $\times$ 6 compression variants). We upload all dataset variants to Azure Blob Storage while maintaining an identical local copy for comparative benchmarking. This setup enables direct comparison between cloud and local storage performance across different compression configurations.

2.2. Compute environment

We conduct all experiments on an Azure `Standard_NC96ads_A100_v4` instance, featuring 96 AMD EPYC 7V13 vCPUs, 866 GB RAM, and an `NVIDIA A100 80GB` GPU in the West US 3 region. For local storage benchmarks, we use Azure temporary storage with a single `250 GB` disk formatted with the `ext4` filesystem. For remote storage experiments, we use Azure Blob Storage with `Standard_LRS` in the `Hot` tier, located in West US 2, resulting in approximately `164 ms` cross-region latency.

3. Methods

We aim to maximize the data loading throughput delivered to the training loop under different hyperparameter configurations. Throughput is measured in megabytes per second (MB/s) and represents how much image data the pipeline can process per unit time. To calculate throughput, we time the complete data loading process. We count the total number of processed pixels and convert it to bytes. The throughput is then computed as the total data volume processed divided by the elapsed wall-clock time. This measurement captures the end-to-end performance of the data loading pipeline.

3.1. Internal tile sampling

We introduce a binary hyperparameter `blocked` $\in \{True, False\}$ that enforces read alignment in the `DataLoader`. When `blocked` = `True`, the sampler chooses a random $k \times k$ tile, then–if the window size is $p \times p$ and $p \leq k$–we jitter the window to a random position that stays wholly inside that tile (Figure 1 `(C)`); otherwise the window starts at the tile origin. Any window that fits in one tile is thus served by a single block read, cutting I/O by up to $4\times$.

3.2. Worker thread pools

We introduce an intra-worker thread pool of width `num_threads` $\in \{1, 2, \ldots, 32\}$. Each worker issues up to `num_threads` concurrent range requests, hiding the ≈164 ms round-trip latency of Azure Blob Storage behind computation already in flight. This approach aims to increase throughput without spawning more workers.

3.3. Bayesian search

The search space is in the order of 10^4 configurations, making exhaustive search impractical. We employ Optuna (Akiba et al., 2019) with the Tree-structured Parzen Estimator (Bergstra et al., 2011), which builds probability models of good and bad configurations to select candidates with high expected improvement. We explore: compression, patch size, number of workers, thread pool size, block alignment and batches preloaded per worker (prefetch factor) as shown in Table 1. Each trial measures throughput (MB/s) over 5 epochs, with each epoch streaming 1024 patches. We conduct 100 trials without early stopping criteria, monitoring both throughput and average GPU utilization.

Table 1: Experimental configurations explored in the data loading pipeline.

Parameter	Values
Compression	`None`, `deflate`{1,6,9}, `lzw`, `lerc-zstd`
`patch_size`	{128, 256, 512, 1024}
`num_workers` (w)	{1, 2, 4, 8, 16, 32, 64}
`num_threads`	{1, 2, 4, 8, 16}
`blocked`	{`True`, `False`}
`prefetch_factor`	{1, 2, 4, 8, 16}

3.4. Grid search

While Bayesian optimization quickly finds promising regions in the hyperparameter space, it may miss important interactions between specific hyperparameter pairs. To address this, we implement $2D$ grid search that explores the cross-product of values for selected hyperparameter pairs. For each combination, we run 5 data loading simulations and calculate the mean and standard deviation of throughput.

We focus our grid search on hyperparameter pairs with the highest feature importance scores derived from our Bayesian optimization results, maintaining consistent experimental conditions and setting optimal values for the fixed hyperparameters. This approach creates detailed visualizations of the performance landscape that reveal how different hyperparameter combinations influence data loading performance.

4. Results

Our Bayesian optimization search revealed distinct optimal configurations for local versus remote storage scenarios, as shown in Table 2. Local storage achieved peak throughput of 1285 ± 29 MB/s with uncompressed imagery and block-aligned patch size reads, while remote storage reached 849 ± 51 MB/s using 64 workers and `LERC-ZSTD` compression. This significant difference in optimal configurations highlights the need for storage-specific optimization approaches. Feature importance in Table 3 tells the same story, showing different optimization priorities between storage types. For local storage, compression type (35.24%) and `num_threads` (32.46%) dominate performance factors. In contrast, remote storage performance is primarily determined by `num_workers` (28.08%) and `num_threads` (26.06%), with compression type contributing only 8.59% to performance variation.

Table 2: Bayesian optimization search: throughput improvement over baseline configurations. **Optimization achieved dramatic speedups** of 4.1× for local storage (1285 vs 313 MB/s) and 20.5× for remote storage (849 vs 41 MB/s), with remote storage benefiting more from worker scaling and local storage from eliminating compression overhead.

Storage	Compression	patch_size	blocked	num_workers	Throughput	Speedup
Local	DEFLATE_6	256	False	4	313±12 MB/s	-
	None	512	True	4	1285±29 MB/s	4.1x
Remote	DEFLATE_6	256	False	4	41±1 MB/s	-
	LERC-ZSTD	1024	True	64	849±51 MB/s	20.5x

Compression strategies showed context-dependent effectiveness. As Table 4 demonstrates, uncompressed data consistently outperformed all compression methods for local storage, with throughput advantages of 1.3-1.6× (974 MB/s for LERC-ZSTD vs. 1286 MB/s for uncompressed at optimal worker counts). For remote storage, LERC-ZSTD compression balanced network transfer efficiency with reasonable decompression overhead, making it the optimal choice despite compression being less important overall.

Tables 5 and 6 consistently show maximum throughput at 1024-pixel patches across both local and remote storage. Notably, this value represents our hyperparameter search's upper bound, suggesting potential for further optimization (although limited by GPU memory). The 1024-pixel patches align efficiently with model processing by fetching 2×2 tiles of 512 pixels each, enabling complete utilization without wasted computation.

Worker-thread interactions showed counter-intuitive results in Table 7. Multi-worker setups with minimal threading achieved higher throughput. Two key limitations likely account for these results: cloud provider rate-limiting when too many worker-thread concurrent requests are made, and our thread pool implementation requiring all requests to complete before returning data, causing a single slow request to degrade overall performance.

For remote storage, block-aligned sampling substantially outperformed random access. Table 6 shows this advantage growing dramatically with patch size–from 45% improvement at 128 pixels (32 MB/s vs. 22 MB/s) to 79% at 1024 pixels (810 MB/s vs. 453 MB/s). This confirms that respecting tile boundaries significantly reduces unnecessary data transfers, with benefits compounding at larger scales. Remote storage loading also benefited from prefetching, converging on factor of 8 to mask the higher network latency. This difference reflects the fundamental need to hide connection latency when streaming from cloud storage.

Based on our findings, we recommend the following practices for training on cloud raster imagery: **match the patch size to the underlying tile structure** of your `GeoTIFF` files (typically 256 or 512 pixels), **use block-aligned sampling** to prevent inefficient partial tile reads, **prioritize high worker counts** (32–64) over threads, **consider `LERC-ZSTD` compression** to bal-

Table 3: Feature-importance rankings for the hyperparameters. Importance scores calculated using functional ANOVA with a random forest surrogate model to decompose variance in the objective function attributable to each hyperparameter. Parameter importance varies significantly between storage types: local performance depends primarily on compression (35%) while remote performance is dominated by worker count (28%), reflecting different bottlenecks in each scenario.

	Local		Remote	
Rank	Parameter	Importance (%)	Parameter	Importance (%)
1	compression	35.24	num_workers	28.08
2	num_threads	32.46	num_threads	26.06
3	patch_size	25.63	patch_size	17.95
4	num_workers	4.22	Blocked	11.11
5	prefetch_factor	1.97	compression	8.59

ance transfer and decompression, and **implement aggressive pre-fetching** (factor of 8).

5. Impact of optimized loading on training

We evaluate loading configurations on model training using three standard semantic segmentation benchmarks. The ISPRS Vaihingen dataset (International Society for Photogrammetry and Remote Sensing (ISPRS), 2014) contains 33 high-resolution (9 cm) aerial scenes of urban areas in Vaihingen, Germany, with three spectral bands (near-infrared, red, green) and digital surface models. The Potsdam dataset (International Society for Photogrammetry and Remote Sensing (ISPRS), 2014) provides 38 tiles of 6000 × 6000 pixels at 5 cm resolution with four spectral bands (RGB plus near-infrared). Both datasets feature six manually-labeled land cover classes (impervious surfaces, buildings, low vegetation, trees, cars, and clutter/background). The IEEE GRSS DFC-22 dataset (Hänsch et al., 2022) offers RGB imagery from 19 French urban areas at 50 cm resolution with approximately 2000 × 2000 pixel images, featuring 14 land cover classes for more complex segmentation tasks.

To systematically evaluate the impact of storage format and loader configuration on training performance, we create three dataset versions for each benchmark. The `default` version converts original `GeoTIFF` files to COGs using `DEFLATE` compression (`zlevel` 6) with 512 × 512 tiles. The `local-optimal` version uses uncompressed data, while `remote-optimal` applies `LERC-ZSTD` compression based on our Bayesian optimization results for cloud storage. We upload `default` and `remote-optimal` version copies to Azure Blob Storage to test remote training scenarios. Each dataset version uses storage-specific loader configurations: `default` employs standard PyTorch settings (4 workers, 256-pixel patches), `local-optimal` uses 4 workers with 512-pixel patches, and `remote-optimal` uses 64 workers with 8× prefetch factor and block-aligned reads.

We train a *UNet* (Ronneberger et al., 2015) segmentation model with a *ResNet-*

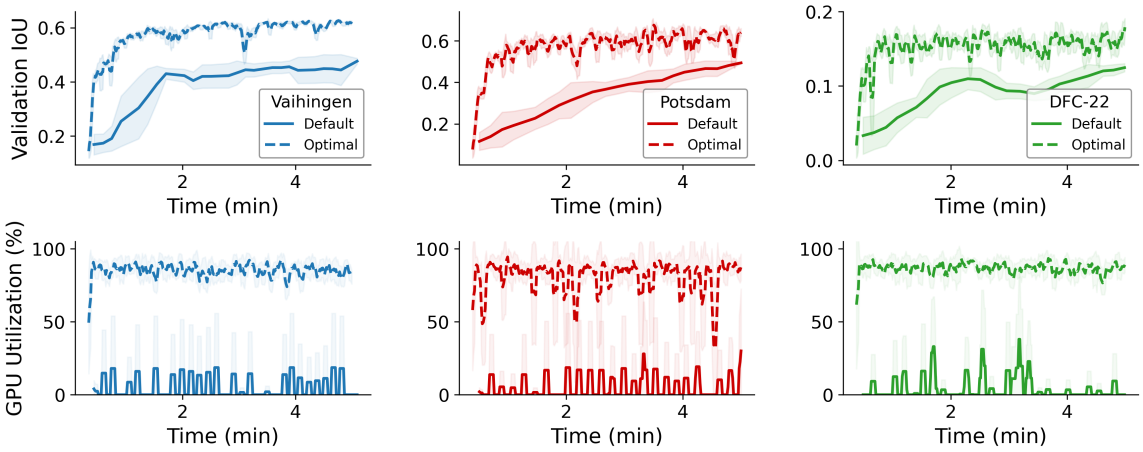

Figure 2: **Training on Azure Cloud Storage**. Top row: Validation IoU over time shows optimal configurations (dashed lines) consistently outperforming default ones (solid lines) across Vaihingen (blue, 0.6 vs 0.45), Potsdam (red, 0.6 vs 0.5), and DFC-22 (green, 0.18 vs 0.12) datasets. Bottom row: GPU utilization demonstrates sustained high usage (85–95%) for optimal configurations versus intermittent, low utilization ($\leq$ 30%) for default settings.

18 (He et al., 2016) encoder using *AdamW* (Loshchilov and Hutter, 2017) optimization with a learning rate of 10^{-3} and standard data augmentation techniques (flips, rotations). We employ a fixed 5-minute training time budget for each configuration, as "speed-runs" are good for measuring how data loading efficiency impacts training dynamics (Coleman et al., 2017; Mattson et al., 2020) before performance curves saturate. We measure validation IoU and GPU utilization throughout the process. GPU utilization was sampled at 1-second intervals using NVIDIA Management Library in a separate thread running concurrently with training. To ensure fair comparison, we use identical model architecture, hyperparameters, and validation data across all configurations, isolating the impact of data loading strategies.

The results visualized in Figure 2 and Figure 3 show that data-loading configuration has the greatest impact when training directly from cloud storage. In the remote case (Figure 2), the `remote-optimal` configuration sustains high GPU utilization and reaches validation IoU comparable to local training across all three datasets, whereas the `default` remote configuration exhibits highly erratic utilization that can drop to 0% and yields substantially lower final IoU: Vaihingen 0.6 vs 0.45, Potsdam 0.6 vs 0.5, and DFC-22 0.18 vs 0.12. Because the model, hyperparameters, and training budget are held constant, these accuracy gaps are attributable to data loading configuration. By contrast, when training from local storage (Figure 3), default and optimal runs converge to similar validation IoU across all datasets (with only a marginal gain on Vaihingen); the main difference is GPU utilization, where the optimal configuration maintains much higher usage (80–100% vs

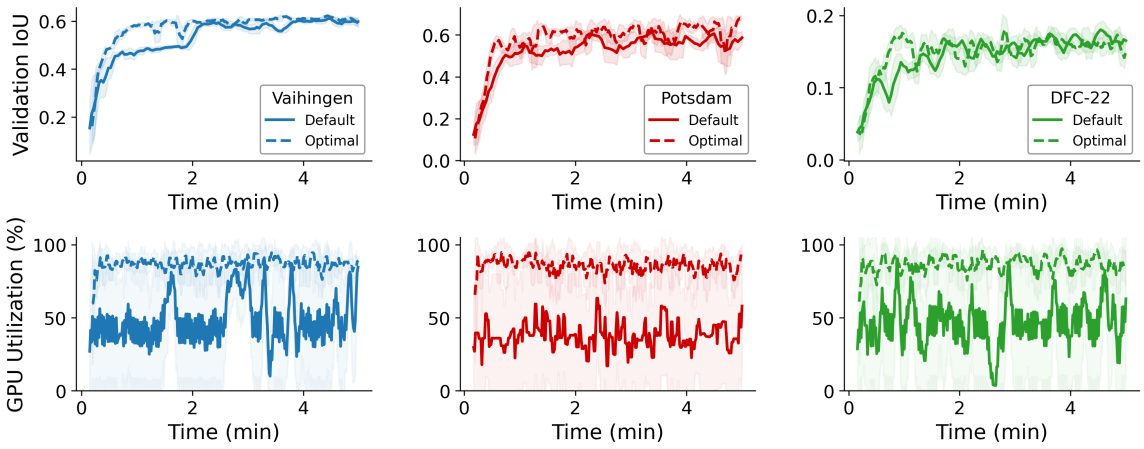

Figure 3: **Training on SSD.** Top row: Validation IoU over time shows optimal configurations (dashed lines) achieving marginal improvements over default configurations (solid lines) for Vaihingen (blue, 0.6 vs 0.5), while reaching similar final performance for Potsdam (red, 0.6) and DFC-22 (green, 0.17) datasets. Bottom row: GPU utilization reveals the key advantage of optimal configurations, which maintain consistent high utilization (90%) across all datasets compared to default configurations' variable utilization.

30–60%), indicating faster time-to-accuracy even when end-of-run accuracy is similar.

These results highlight the critical importance of data format and loading configurations in EO deep learning. Properly aligned reads, optimal worker counts, and appropriate compression formats directly translate to higher GPU utilization and better performance. Most importantly, our experiments demonstrate that with optimized configurations, models can train directly on cloud-stored data without performance penalty, eliminating the need for costly local storage.

6. Limitations

Our approach has several practical limitations. Public archives like Microsoft Planetary Computer use fixed data formats that cannot be modified, restricting our optimizations to data loading rather than storage format improvements. While we could transform and re-store data on-the-fly, this would add infrastructure complexity. Some optimization choices may negatively impact deep learning workflows. For example, tile-aligned reads prevent random cropping during training, potentially reducing model performance. Our investigation also focuses exclusively on lossless compression methods within standard `PyTorch DataLoader` configurations. We do not explore lossy compression tradeoffs, which could offer different performance characteristics depending on imagery spatial resolution and downstream tasks. Additionally, we do not account for GDAL environment variables that can affect read performance. Finally, our evaluation assumes remote data access across different data centers (Azure West US 2 and 3). Co-locating data and compute would reduce latency and may change our optimization recommendations.

7. Future work

As cloud-hosted EO datasets continue to grow in scale, traditional epoch-based training becomes increasingly impractical. Hence, we propose replacing epochs with infinite samplers that continuously fetch data patches using asynchronous, non-blocking frameworks. This eliminates artificial epoch boundaries and samples according to defined probability distributions. These training systems must handle network failures gracefully through retry mechanisms, placeholder tensors, and data replication to maintain batch sizes when network issues occur. Beyond training, we need systematic studies comparing streaming efficiency across data formats beyond COG, including `Zarr`, covering chunking strategies, compression methods, and access patterns for each format. In parallel, data providers can make EO archives more "training-friendly" by publishing cloud-native rasters with tiles aligned to common ML patch sizes (e.g., 512×512), exposing efficient concurrent byte-range access, using lossless yet high-throughput compression, and supplying machine-readable metadata (e.g., STAC) that surfaces tile layout and nodata masks so loaders can issue block-aligned reads. Finally, the EO community should establish standardized performance benchmarks similar to *DAWNBench* (Coleman et al., 2017) and *MLPerf* (Mattson et al., 2020) to drive discovery of efficient training practices for cloud imagery and enable fair comparisons across methods.

Impact statement

Our work addresses the efficiency bottlenecks in EO deep learning by optimizing Cloud-to-GPU data streaming. Our approach removes a key barrier to experimentation, enabling faster iteration in geospatial machine learning. As EO data volumes continue growing and geospatial foundation models emerge, efficient data streaming becomes critical for scalable model development. Our optimizations reduce computational waste, lower development costs, and enable more efficient resource utilization across the research community. The primary societal benefit is faster insight generation for environmental monitoring and decision making. Researchers can now train models directly on expanding satellite archives, facilitating timely incorporation of new observations for dynamic systems like vegetation change, urban development, and natural disasters. This improved accessibility has the potential to accelerate progress on pressing environmental challenges.

References

Ogc geotiff standard, version 1.1 (ogc doc no. 19-008r4). Implementation Standard 19-008r4, Open Geospatial Consortium, Arlington, VA, USA, September 2019.

Alex Aizman, Gavin Maltby, and Thomas Breuel. High performance i/o for large scale deep learning. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5965–5967. IEEE, 2019.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris

Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

Open Geospatial Consortium et al. Zarr storage specification 2.0 community standard, 2023.

Matthias Drusch, Umberto Del Bello, Sébastien Carlier, Olivier Colin, Veronica Fernandez, Ferran Gascon, Bianca Hoersch, Claudia Isola, Paolo Laberinti, Philippe Martimort, et al. Sentinel-2: Esa's optical high-resolution mission for gmes operational services. *Remote sensing of Environment*, 120:25–36, 2012.

Javier Gorroño, Daniel J. Varon, Itziar Irakulis-Loitxate, and Luis Guanter. Understanding the potential of sentinel-2 for monitoring methane point emissions. *Atmospheric Measurement Techniques*, 16(1):89–102, 2023. doi: 10.5194/amt-16-89-2023.

Ronny Hänsch, Claudio Persello, Gemine Vivone, Javiera Castillo Navarro, Alexandre Boulch, Sebastien Lefevre, and Bertrand Saux. The 2022 ieee grss data fusion contest: Semisupervised learning [technical committees]. *IEEE Geoscience and Remote Sensing Magazine*, 10(1):334–337, 2022.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

International Society for Photogrammetry and Remote Sensing (ISPRS). Isprs 2d semantic labeling benchmark – vaihingen and potsdam, 2014. URL https://www.isprs.org/education/benchmarks/UrbanSemLab/semantic-labeling.aspx. Dataset.

Max Jones, Joseph J Hamman, and Wei Ji Leong. Xbatcher-a python package that simplifies feeding xarray data objects to machine learning libraries. In *103rd AMS Annual Meeting*. AMS, 2023.

Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Madry. Ffcv: Accelerating training by removing data bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12011–12020, 2023.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2: 336–349, 2020.

Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *arXiv preprint arXiv:2007.06775*, 2020a.

Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *arXiv preprint arXiv:2007.06775*, 2020b.

A Paszke. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

Darius Phiri, Matamyo Simwanda, Serajis Salekin, Vincent R. Nyirenda, Yuji Murayama, and Manjula Ranagalage. Sentinel-2 data for land cover/use mapping: A review. *Remote Sensing*, 12(14): 2291, 2020. doi: 10.3390/rs12142291.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

Joel Segarra, Maria Luisa Buchaillot, Jose Luis Araus, and Shawn C. Kefauver. Remote sensing for precision agriculture: Sentinel-2 improved features and applications. *Agronomy*, 10(5):641, 2020. doi: 10.3390/agronomy10050641.

Microsoft Open Source, Matt McFarland, Rob Emanuele, Dan Morris, and Tom Augspurger. microsoft/planetarycomputer: October 2022, October 2022. URL https://doi.org/10.5281/zenodo.7261897.

Ivan Svogor, Christian Eichenberger, Markus Spanring, Moritz Neun, and Michael Kopp. Profiling and improving the pytorch dataloader for high-latency storage: A technical report. *arXiv preprint arXiv:2211.04908*, 2022.

R Wilkinson, MM Mleczko, RJW Brewin, KJ Gaston, M Mueller, JD Shutler, X Yan, and K Anderson. Environmental impacts of earth observation data in the constellation and cloud computing era. *Science of The Total Environment*, 909:168584, 2024.

Mahdi Zolnouri, Xinlin Li, and Vahid Partovi Nia. Importance of data loading pipeline in training deep neural networks. *arXiv preprint arXiv:2005.02130*, 2020.

Appendix A. Grid search results

Table 4: **Local Throughput**: Num Workers vs. Compression. **Uncompressed data consistently outperforms all compression methods** by 1.3-1.6×, achieving peak performance of 1286 MB/s with 8 workers, demonstrating that compression overhead outweighs benefits for local storage access. Values: MB/s ± std. (underlined: column best; bold underlined: overall best)

	Compression					
num_workers	DEFLATE_1	DEFLATE_6	DEFLATE_9	LERC-ZSTD	LZW	None
1	268 (11)	268 (15)	270 (03)	388 (08)	295 (07)	610 (05)
2	491 (17)	496 (02)	475 (04)	685 (13)	525 (21)	981 (17)
4	764 (42)	748 (19)	746 (14)	974 (14)	806 (67)	1244 (48)
8	786 (18)	761 (12)	747 (16)	867 (61)	796 (40)	**1286** (22)
16	782 (22)	753 (16)	720 (11)	856 (53)	818 (45)	1242 (26)
32	765 (45)	750 (20)	730 (22)	876 (73)	773 (50)	1200 (33)
64	705 (71)	719 (29)	711 (14)	857 (101)	690 (61)	1184 (14)

Table 5: **Local Throughput**: Num. Workers vs. Patch Size. **Larger patch sizes consistently yield higher throughput**, with 1024-pixel patches achieving optimal performance (1298 MB/s at 8 workers) by efficiently utilizing GPU memory through 2×2 tiles of 512 pixels each. Values: MB/s ± std. (underlined: column best; bold underlined: overall best)

	patch_size			
num_workers	128	256	512	1024
1	127 (06)	297 (07)	592 (12)	674 (31)
2	229 (05)	510 (15)	946 (09)	1058 (20)
4	340 (18)	701 (31)	1147 (40)	1269 (40)
8	323 (17)	738 (40)	1124 (33)	**1298** (26)
16	310 (08)	719 (16)	1148 (08)	1297 (37)
32	279 (11)	682 (10)	1105 (24)	1273 (09)
64	211 (10)	617 (10)	1055 (21)	1217 (50)

Table 6: **Remote Throughput**: Sampler vs. Patch Size. **Block-aligned sampling dramatically outperforms random access**, with advantages growing from 45% at 128 pixels to 79% at 1024 pixels (810 vs 453 MB/s), confirming that respecting tile boundaries minimizes unnecessary data transfers. Values: MB/s ± std. (underlined: column best; bold underlined: overall best)

	patch_size			
Blocked	128	256	512	1024
False	22 (8)	83 (20)	254 (79)	453 (81)
True	32 (1)	117 (9)	401 (4)	**810** (57)

Table 7: **Remote Throughput**: Workers vs. Threads. Counter-intuitively, fewer threads with more workers achieve higher throughput (817 MB/s with 16 workers, 1 thread), likely due to cloud provider rate-limiting and thread pool synchronization bottlenecks that cause slow requests to degrade overall performance. Values: MB/s ± std. (underlined: column best; bold underlined: overall best)

	Number of Threads			
num_workers	1	2	8	32
1	112 (03)	187 (16)	359 (08)	362 (19)
2	213 (10)	340 (09)	486 (38)	402 (20)
4	412 (12)	560 (17)	497 (38)	356 (41)
8	600 (52)	729 (58)	511 (11)	372 (37)
16	**817** (61)	669 (118)	487 (30)	375 (11)
32	773 (113)	666 (119)	496 (20)	371 (27)
64	761 (180)	686 (85)	474 (60)	381 (19)