Evaluating Accuracy–Performance Trade-Offs in Deep Learning Frameworks for Diabetic Retinopathy Detection

Bruno Morales(✉), Cristiano Künas, Rodrigo Machado, Pedro Rigon, and Philippe O. A. Navaux

Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
{bmmorales,cakunas,rcmachado,phcrigon,navaux}@inf.ufrgs.br
https://www.inf.ufrgs.br/site/

Abstract. Deep learning has become one of the most important tools in medical image processing, requiring models that meet response-time constraints, accuracy requirements, and memory limitations imposed by hardware limitations. Given our current scenario, this demand is even more important in hospitals and institutions with limited computing resources, where access to high-performance GPUs is not always available for daily use. Our results show that optimization can significantly improve computational efficiency, with throughput gains of up to 288.8% and reductions in training time of up to 70.3%. However, the effect on Area Under the Curve (AUC) depends on the architecture used. Our research supports the selection and optimization of healthcare workloads running on GPUs, aiming to deliver more efficient, accessible, and clinically appropriate solutions.

Keywords: Deep Learning · High Performance Computing · Healthcare

1 Introduction

Artificial intelligence (AI) and deep learning (DL) have advanced very rapidly in a short period of time, enabling increasingly robust and optimized solutions for complex pattern recognition tasks across multiple domains [1]. Especially in healthcare, these advancements have been very significant, as deep learning models based on image classifications continue to improve the accuracy, speed, and scalability of diagnostic support systems. In ophthalmology, automated fundus image analysis has become particularly relevant, with deep learning now playing a central role in diabetic retinopathy (DR) screening pipelines [2].

The significance of such improvements extends beyond technology to healthcare delivery. A shortage of specialists, long waiting lists, and overcrowding are common in public healthcare organizations. With all these constraints in

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026, LNCS 16767, pp. 303–320, 2027.
https://doi.org/10.1007/978-3-032-30491-9_20

mind, an automated tool that screens patients to determine whether they need a specialist referral could prove highly effective in improving early diagnosis and reducing the number of patients referred to specialist care.

However, developing deployable solutions based on deep learning algorithms entails more than simply building a powerful prediction model. Modern neural networks require substantial amounts of both RAM (host memory) and VRAM (GPU memory). Both components are expensive and rare. Therefore, optimizing computational power usage is crucial when developing deployable neural network-based prediction models that can be trained on a single GPU.

Figure 1 illustrates representative fundus images without diabetic retinopathy and with diabetic retinopathy, highlighting the visual contrast addressed by the binary classification task considered in this study.

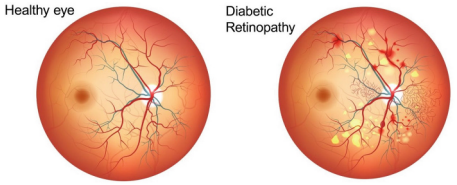

Fig. 1. Illustrative representation of diabetic retinopathy.

Motivated by this current scenario in the evolution of technology, we investigated the automated classification of diabetic retinopathy from fundus images, focusing on the practical trade-offs between predictive performance and training efficiency. We analyze six variants of the same image classification pipeline based on the pre trained InceptionV3 convolutional neural network (CNN) [3]. InceptionV3 combines multi-scale feature extraction with computational efficiency, making it a strong backbone for transfer learning, which has become a standard strategy in DR grading tasks [4]. The model is initialized with weights pre-trained on large-scale datasets such as ImageNet [5], allowing it to adapt effectively to downstream medical imaging tasks.

The evaluated variants include three baseline implementations developed in TensorFlow, PyTorch, and MONAI, as well as three optimized counterparts designed to improve performance under the selected evaluation criteria. Cross-framework comparisons of this kind have already attracted attention in medical imaging research [6]. In the optimized configurations, we explore reduced-precision execution through FP16 mixed precision training and fine-tuning, in which early network layers remain fixed while higher level layers are updated. These strategies are intended to lower computational cost, and improve convergence stability when adapting a pre trained model to a smaller dataset.

The trade-offs will be evaluated by comparing the six pipeline variations based on AUC scores, training time, and throughput. This is done for both ARM and x86 platforms, allowing us to study how efficiency and accuracy are

affected by platform architecture, deep learning framework, and floating-point precision in the context of real-world AI use in DR classification. We want to demonstrate how choices made during the design of a training workflow can affect both AI efficiency and usability intended to support specialists in diabetic retinopathy screening.

The remainder of this paper is organized as follows. Section 2 presents the background on PyTorch, TensorFlow/Keras, and MONAI. Section 3 reviews related work on diabetic retinopathy detection and on efficiency-oriented optimization strategies for deep learning pipelines. Section 4 describes the dataset, hardware platforms, model architecture, training procedure, and optimization techniques adopted in the experiments. Section 5 presents and discusses the results in terms of AUC, throughput, training time and memory. Finally, Sect. 6 concludes the paper and outlines directions for future work.

2 Background

We will analyze frameworks such as PyTorch, TensorFlow/Keras, and MONAI, as they are relevant to different aspects of our research. PyTorch is an advanced deep learning platform that supports dynamic computational graphs and emphasizes Python. In turn, TensorFlow/Keras will be an excellent benchmark due to its mature ecosystem of solutions with scalable graph-based execution. Furthermore, MONAI has a particular focus on medical imaging solutions. Thus, using these frameworks, it will be possible to evaluate both general and healthcare-specific deep learning solutions to obtain a comparison across multiple isolated scenarios.

2.1 Pytorch

PyTorch by Meta AI (2016) facilitates neural network programming using idiomatic Python and efficient use of hardware accelerators [7]. The dynamic execution paradigm of PyTorch enables quick debugging and inspection of tensors, their shapes, values, and gradients. In distributed training, PyTorch leverages distributed tooling such as DistributedDataParallel (DDP) for gradient synchronization [8]. The interplay between autograd and continual optimizations, in terms of runtime and accelerator support, makes PyTorch effective for distributed and interactive training.

2.2 TensorFlow and Keras

TensorFlow (Google Brain, 2015) [9] supports scalable execution via computation graphs and enables efficient distributed training [10]. TensorFlow is available in both graph mode and eager execution mode. Further, TensorFlow can compile and optimize intensive computations using XLA [11].

2.3 MONAI

Medical Open Network for AI (MONAI) is an open-source framework on top of PyTorch for developing applications in medical imaging [12]. It provides a wide range of domain-specific tools, such as transforms, architectures, and losses, specifically tuned for volumetric data and inference.

3 Related Work

Research on diabetic retinopathy (DR) has changed considerably over the last few years. Earlier studies often relied on handcrafted lesion descriptors and multi-stage pipelines. In contrast, more recent work has increasingly adopted end-to-end deep learning models for classification and severity grading from fundus images. Reviews of the field consistently document this shift, particularly in studies trained and evaluated on public datasets such as EyePACS, APTOS, Messidor, IDRiD, and FGADR [13,14]. Beyond reporting accuracy on a single dataset, some studies have also highlighted transferability across settings. In this context, Zhou *et al.* [15] introduced FGADR as a benchmark for lesion segmentation, grading, and transfer learning, thereby broadening the discussion on model generalization.

Among the strategies used in DR screening, transfer learning remains one of the most practical. This is primarily due to retinal datasets, which are generally small, class unbalanced, and collected under heterogeneous imaging conditions. Prior work has shown that pretrained convolutional backbones can achieve competitive results for DR diagnosis and grading [16]. Related transfer-learning strategies have also been successfully applied in smartphone based retinal imaging settings [17]. More recent studies have explored alternatives that go beyond conventional CNN pretraining. Alam *et al.* [18] reported gains from contrastive pretraining, particularly in terms of representation quality and cross-institution transfer. Goh *et al.* [19] showed that Vision Transformers are strong candidates for referable DR detection. There is also evidence from prospective screening settings that model performance should be evaluated alongside robustness and deployment conditions, rather than solely through aggregate test metrics [20].

Another relevant body of work concerns efficiency during training and inference. In this literature, the central question is not only whether a model is accurate, but also how efficiently it can be trained and executed. Mixed precision computation is one of the best known examples, and prior studies have shown that it can improve the runtime efficiency of deep neural networks [21]. However, these benefits are not always automatic. He *et al.* [22] showed that precision choices should account for casting overhead, since lower precision does not necessarily yield better end-to-end performance. Similar observations appear in work on tensor layout, where choices such as NCHW and NHWC can substantially affect throughput [23]. Systems-oriented studies have likewise shown that the input pipeline itself can become a bottleneck, limiting GPU utilization when data loading and preprocessing are not carefully optimized [24,25].

Although both lines of research are well established, they have been less frequently examined together in the DR literature. Ophthalmic studies typically emphasize predictive metrics such as AUC, sensitivity, specificity, and interpretability, whereas systems papers usually evaluate performance on more generic vision workloads. Consequently, evidence remains limited on how framework-level and optimization choices behave under the same retinal classification task. This issue is also relevant for healthcare-oriented ecosystems such as MONAI, a PyTorch-based framework designed for medical imaging [12]. However, comparable evidence across TensorFlow, native PyTorch, and MONAI-based pipelines under equivalent experimental conditions remains scarce. Motivated by this gap, our study compares accuracy, throughput, training time, for the same DR classification pipeline implemented in TensorFlow, PyTorch, and MONAI, under both baseline and optimized settings.

4 Methodology

This section describes the methodology adopted in our experiments, including the dataset, architecture, training procedure, and optimization strategies. TensorFlow was used as the baseline framework, since it corresponds to the original implementation. To make the experiments comparable across frameworks, the TFRecord (TfRec) files were converted to IDX format. In addition, all images were prepared at a resolution of 299×299 pixels, which is the expected input size for InceptionV3.

4.1 Dataset

For the experiments reported in this article, we used 15,816 fundus images from a Brazilian diabetic retinopathy dataset, with labels binarized into a two-class setting (0/1). The dataset is imbalanced, with 11,625 samples in class 0 and 4,191 in class 1. Approximately 70/30 train-test split was adopted, yielding 11,000 training images and 4,816 test images. Details on data sources, acquisition, inclusion criteria, and labeling protocol are provided in [26].

4.2 Hardware Platforms

The machines used in the experiment were 1xNVIDIA L40S with 18,176 CUDA cores and 48 GB of global memory, and 1xNVIDIA GeForce RTX 4090 with 16,384 CUDA cores and 24 GB of global memory available in the LPPD infrastructure[1] at INF/UFRGS. Hydra node at the Lyon site of Grid'5000[2] [27], equipped with an 1xNVIDIA GH200 with 16,896 CUDA cores and 96 GB of global memory.

[1] http://gppd-hpc.inf.ufrgs.br.
[2] https://www.grid5000.fr.

4.3 Training and Optimization

Table 1 summarizes the software stack and container versions used for the TensorFlow, PyTorch, and MONAI experiments.

Table 1. Version comparison across TensorFlow, PyTorch, and MONAI.

Component	TensorFlow	PyTorch	MONAI
TensorFlow	2.17.0	—	—
PyTorch	—	2.5.0	2.5.0
MONAI	—	2.5.0	1.3.2
CUDA	12.6.3	12.6.3	12.6.3
cuDNN	9.6.0.74	9.6.0.74	9.6.0.74
DALI	1.44	1.44	1.44
Python	3.12	3.12	3.12

To reduce variations in the training process, each configuration was executed 10 times. We refer to the unoptimized runs as Base and to the optimized runs as Opt. For each framework and GPU, the corresponding Base and Opt runs used the same random seeds, so paired tests were applied to the run-wise AUC differences. Each run used 200 epochs and a batch size of 96 [29], chosen as a compromise between stochastic gradient stability and computational efficiency. Figure 2 illustrates the experimental workflow chosen to compare the six training pipelines. All implementations are built on the same InceptionV3 architecture, which provides a common architectural foundation for the analysis. From there, the workflow is divided into reference and optimized versions. The reference branch corresponds to the original training configuration in each framework without any optimization, while the optimized branch includes the strategies implemented to improve performance. Despite these differences, all variants use the same training settings: 10 runs, 200 epochs, and a batch size of 96. This organization helps clarify which elements remain fixed between experiments and which are modified in the optimized configurations.

We report the mean and standard deviation of all metrics across these repetitions. Each variant uses the InceptionV3 architecture as its convolutional backbone, initialized with pre-trained weights on ImageNet. We separated our six approaches into clean, unoptimized versions (BASE) and optimized versions (OPT). The base versions follow a straightforward training procedure using each framework's standard data pipeline and, in most cases, the default precision and scheduling settings. The three optimized variants TF-Opt, PT-Opt, and MN-Opt—incorporate a shared set of performance and convergence techniques, while also making architecture-specific choices such as tensor memory layout and compiler backend.

During training, we also investigated how the memory organization behavior of the input tensors affects model execution. We evaluated two tensor layouts

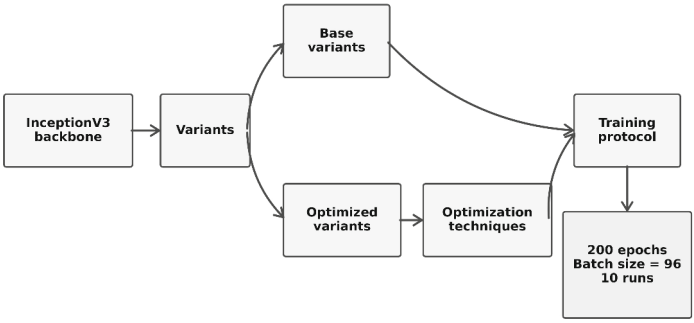

Fig. 2. Workflow of the steps.

supported by the frameworks in isolation, since the order of dimensions influences how data is stored and accessed during convolution operations. TensorFlow uses the NHWC tensor layout, which denotes the dimension order [batch, height, width, channels], whereas PyTorch was evaluated with the NCHW tensor layout, which denotes the order [batch, channels, height, width] [28]. Although both layouts represent the same input data, the different order of dimensions can affect memory access patterns, convolution performance, and overall training scalability.

To assess the practical impact of these layouts, we present the AUC and training time for each configuration discussed, allowing a direct comparison of predictive performance and computational efficiency.

We also addressed the use of FP16 in optimized versions, seeking to reduce the memory consumption of the computing cores [21]. The benefits of mixed precision often depend on adjustments that take into account the costs. For the MONAI variant, we used specialized mixed-precision toolkits for the optimization phase, avoiding loss of precision.

Four optimization techniques are consistently applied in TF-Opt, PT-Opt, and MN-Opt. Each of these optimizations addressed a different aspect of the training process and together, they formed the optimization base upon which specific structure improvements are added.

4.4 GPU-Accelerated Data Loading via NVIDIA DALI

In the basic variants where we have no optimizations, image decoding and preprocessing are managed by each framework's standard CPU based data pipeline. When the CPU cannot prepare batches as quickly as the GPU can consume them, the GPU becomes idle, causing a problem known as pipeline depletion, and training performance suffers the consequences. All three optimized variants replace the CPU side pipeline with NVIDIA DALI, which performs data decoding, resizing, and augmentation directly on the GPU. By moving these operations to the accelerator and overlapping data preparation with model computation, DALI reduces the transfer bottleneck between the host and the device and allows

the GPU to maintain higher utilization throughout training. The practical result is a significant increase in training performance, especially on GPUs with high computational capacity that would otherwise be underutilized while waiting for data, negatively impacting the total training time.

4.5 Mixed-Precision Training (FP16)

All optimized variants use AMP-based FP16 mixed-precision training [21], so that most forward and backward computations are performed in 16-bit floating-point. This helps reduce memory usage and enables matrix multiplications to leverage the GPU's Tensor Cores, which are optimized for reduced-precision arithmetic and can deliver higher throughput than standard FP32 units. In cost-aware optimization settings [22], these memory savings may also make it easier to train on GPUs with limited VRAM or to use larger effective batch sizes on the same hardware.

4.6 Cosine Annealing Learning Rate Schedule with Warmup

Rather than maintaining a constant learning rate or using step-wise decay, each optimized variant uses a cosine annealing schedule with a linear warmup phase. During warmup, the learning rate increases from a near-zero value to its peak, preventing the large, potentially destabilizing gradients that commonly arise in the first few epochs particularly when adapting a pre-trained backbone to a new classification task. After warmup, the cosine schedule smoothly reduces the learning rate toward a specified minimum, encouraging the optimizer to settle into flatter regions of the loss landscape. The specific number of warmup epochs varies slightly across frameworks. Still, the underlying strategy remains identical and provides a more stable, effective training trajectory than a flat learning rate.

4.7 Exponential Moving Average (EMA) of Model Weights

Each optimized variant maintains an exponential moving average (EMA) of the model parameters with a decay factor of 0.999. Rather than evaluating the model at its current parameter values, which may fluctuate due to stochastic gradient noise, EMA produces a smoothed version of the weights that reflects the running average of the optimization trajectory. This smoothed model is used at evaluation time and tends to generalize better than the instantaneous parameters, particularly in the later stages of training when the loss landscape becomes flatter. The optimizer oscillates around a local minimum.

4.8 Ablation Protocol

To isolate the contribution of each optimization used in the optimized pipeline, we performed a component-wise ablation study. Starting from the baseline configuration, we enabled one optimization at a time DALI AMP/FP16, cosine

learning-rate scheduling, and EMA—and compared these variants against the full optimized configuration. All other factors, including model architecture, dataset split, batch size, number of epochs, and evaluation protocol, were kept fixed. This design allows the effect of each component to be assessed relative to the same baseline, while also showing the overall behavior of the complete optimized stack.

4.9 Statistical Analysis

To assess whether optimization significantly changed predictive performance, we compared the run-level AUC values of the Base and Opt variants within each framework and GPU. Because Base and Opt runs were matched by run index under the same seed schedule, paired tests were applied to the run-wise AUC differences. Normality of these paired differences was evaluated using the Shapiro–Wilk test [30]. When normality was not rejected, a paired t-test was used; otherwise, a Wilcoxon signed-rank test [31] was applied. A significance level of $\alpha = 0.05$ was adopted for all comparisons.

5 Results

This section presents the quantitative evaluation of six training pipelines: TensorFlow Base (TF-Base), TensorFlow Optimized (TF-Opt), PyTorch Base (PT-Base), PyTorch Optimized (PT-Opt), MONAI Base (MN-Base), and MONAI Optimized (MN-Opt). Each experiment was executed on a single GPU. This design choice has an important implication: the inter-GPU communication fabric (e.g., NVLink, InfiniBand, or PCIe switches) remains entirely inactive. Therefore, any performance differences observed across the three platforms can be attributed exclusively to on-chip resources such as memory bandwidth, Tensor Core availability, and sustained clock frequency rather than to interconnect or topology effects.

5.1 Model Quality: AUC

AUC was used as the main quality metric because it shows how well the model separates the two classes. Since it does not depend on a single decision threshold, it is useful for comparing the effect of different pipelines on classification performance. Figure 3 summarizes the mean AUC obtained by each framework variant on each GPU platform.

The figure shows that changing the GPU had little effect on AUC as long as the training pipeline remained the same. Differences in discriminative performance were driven primarily by the software configuration rather than by the hardware itself.

The statistical analysis of the Base–Opt AUC differences confirmed that the effect of optimization was framework-dependent across all three GPUs. On the GH200, PyTorch showed a significant increase in AUC (0.9186 → 0.9855,

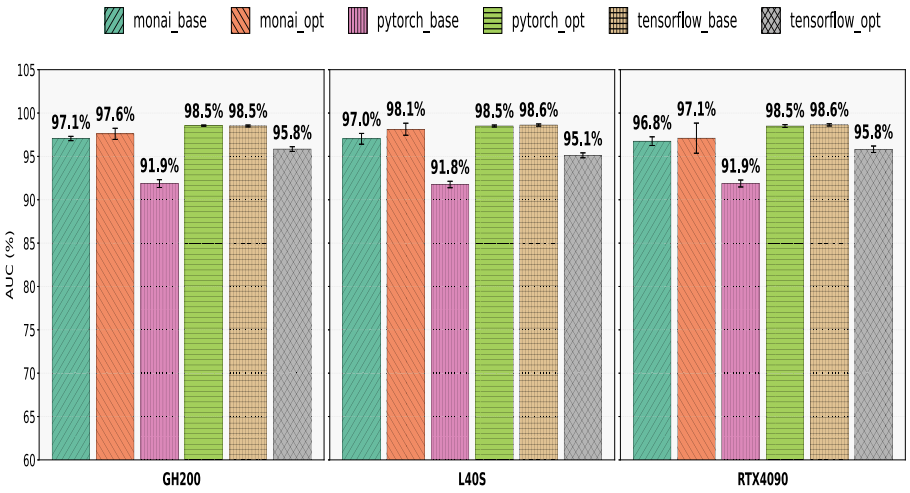

Fig. 3. Mean AUC per framework variant on each GPU platform.

$p = 5.32 \times 10^{-12}$), whereas TensorFlow showed a significant decrease (0.9851 → 0.9585, $p = 1.50 \times 10^{-10}$). In contrast, the MONAI difference on this GPU did not reach statistical significance (0.9706 → 0.9760, $p = 0.0588$).

On the L40S, all three frameworks showed significant Base–Opt differences: PyTorch ($p = 2.96 \times 10^{-12}$), TensorFlow ($p = 1.73 \times 10^{-10}$), and MONAI ($p = 0.00199$). On the RTX 4090, PyTorch ($p = 1.01 \times 10^{-12}$) and TensorFlow ($p = 1.04 \times 10^{-8}$) also remained significant, whereas MONAI again showed no significant difference (Wilcoxon, $p = 0.359$). Overall, these results indicate that optimization consistently altered AUC in PyTorch and TensorFlow, but produced smaller and less consistent effects in MONAI.

For TensorFlow, the baseline setting produced the best overall AUC values, with TF-Base remaining close to 98.5%–98.6% across all three GPUs. After optimization, performance declined on all platforms, reaching 95.8% on the GH200, 95.1% on the L40S, and 95.8% on the RTX 4090. Although the optimized version improved computational efficiency, this came at a clear cost in predictive performance.

PyTorch showed the opposite pattern. Its baseline results were the weakest in the study, with PT-Base ranging from 91.8% to 91.9%, but optimization substantially improved performance. PT-Opt reached about 98.5% across all three GPUs, representing the largest AUC gain observed in this study. This improvement was reproduced consistently across different hardware platforms.

MONAI occupied an intermediate position between these two cases. The baseline pipeline produced AUC values ranging from 96.8% to 97.1%, and optimization yielded smaller gains than in PyTorch, although still positive in most settings. MN-Opt reached 97.6% on the GH200 and 98.1% on the L40S, while the RTX 4090 remained at 97.1%. Overall, these results suggest that optimization

did not affect all frameworks equally: it reduced AUC in TensorFlow, strongly improved it in PyTorch, and produced more limited gains in MONAI.

5.2 Throughput

The throughput was included to show how many images each pipeline can process per second. This is important because it reflects how efficiently each framework uses the available GPU resources during training and helps in the final analysis of how each approach utilized the available resources in each process. Figure 4 summarizes the mean training throughput achieved by each framework variant on the GH200, L40S, and RTX 4090.

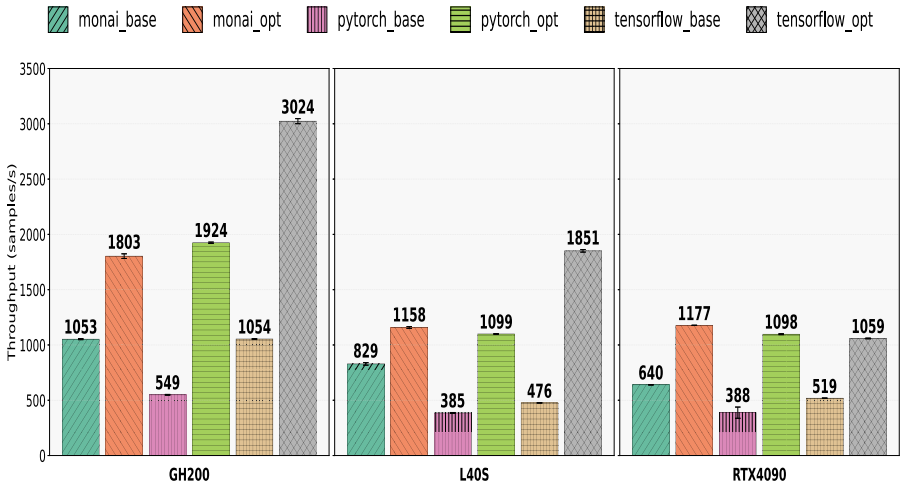

Fig. 4. Average training throughput (img/s) per framework variant on each selected GPU.

The figure shows that across all three frameworks, the optimized pipelines consistently achieved higher throughput than their respective baselines. However, because each optimized variant combines multiple modifications introduced in Sect. 4, these gains should be interpreted as the result of the full optimized pipeline rather than as the isolated effect of any single optimization.

TensorFlow delivered the highest absolute throughput values in this study. On the GH200, TF-Opt increased throughput from 1,054 to 3,024 img/s, yielding a 186.9% gain. On the L40S, throughput increased from 476 to 1,851 img/s, a 288.8% gain. On the RTX 4090, throughput increased from 519 to 1,059 img/s, corresponding to a 104.1% increase. Under the optimized TensorFlow configuration, the GH200 achieved 1.63× the throughput of the L40S and 2.86× that of the RTX 4090. These results indicate that the optimized TensorFlow pipeline

benefited more strongly from the GH200 and L40S platforms than from the RTX 4090.

PyTorch also improved throughput on all three platforms. On the GH200, PT-Opt increased throughput from 549 to 1,924 img/s, yielding a 250.4% gain. On the L40S, throughput increased from 385 to 1,099 img/s, a 185.3% gain. On the RTX 4090, throughput increased from 388 to 1,098 img/s, a 183.1% increase. Notably, the optimized PyTorch results on the L40S and RTX 4090 were nearly identical, differing by only about 1 img/s. In this case, the optimized PyTorch pipeline produced very similar throughput on these two GPUs despite their different hardware classes.

MONAI-Opt also improved throughput on all platforms. On the GH200, throughput increased from 1,053 to 1,803 img/s, a 71.3% gain. On the L40S GPU, throughput increased from 829 to 1,158 img/s, a 39.8% gain. On the RTX 4090, throughput increased from 640 to approximately 1,178 img/s, corresponding to an 84.1% increase. Although these relative gains were smaller than those observed for TensorFlow and PyTorch on some GPUs, MONAI-Opt remained competitive and achieved the highest optimized throughput on the RTX 4090 among the three frameworks.

In the final analysis, these results show that the interaction between the software stack and the hardware platform strongly influenced throughput. TF-Opt achieved the highest throughput on the GH200 at 3,024 img/s and on the L40S at 1,851 img/s. On the RTX 4090, however, the three optimized pipelines were much closer, ranging from approximately 1,059 to 1,178 img/s, with MONAI-Opt holding a slight lead. Finally, the throughput results suggest that optimization substantially improved training efficiency, but that the magnitude of this improvement depended on both the framework implementation and the GPU used.

5.3 Train Time

Training time was considered because it shows the total wall-clock time needed to complete training. This metric is useful for evaluating the practical cost of each pipeline. Figure 5 reports the mean wall-clock training time, in seconds, for the full 200-epoch run.

PT-Base is the slowest configuration on every platform, exceeding 5 000 s even on the GH200. The DALI + AMP + `torch.compile` stack of PT-Opt cuts training time by −70.3 % on the GH200 (5 007 → 1 488 s), −61.5 % on the L40S (6 836 → 2 629 s), and −60.6 % on the RTX 4090 (7 032 → 2 771 s). The consistent 60–70 % reduction across all three platforms indicates that PT-Base's bottleneck resides predominantly in the Python DataLoader and CPU-side TFRecord decoding: once DALI moves decoding to the GPU and `torch.compile` eliminates per-step Python dispatch overhead, and the GPU is kept saturated regardless of the underlying architecture. The additional 10 % gain observed on the GH200 (70 % vs. 60 %) is attributable to higher memory bandwidth allowing DALI's prefetch buffer to drain and refill faster.

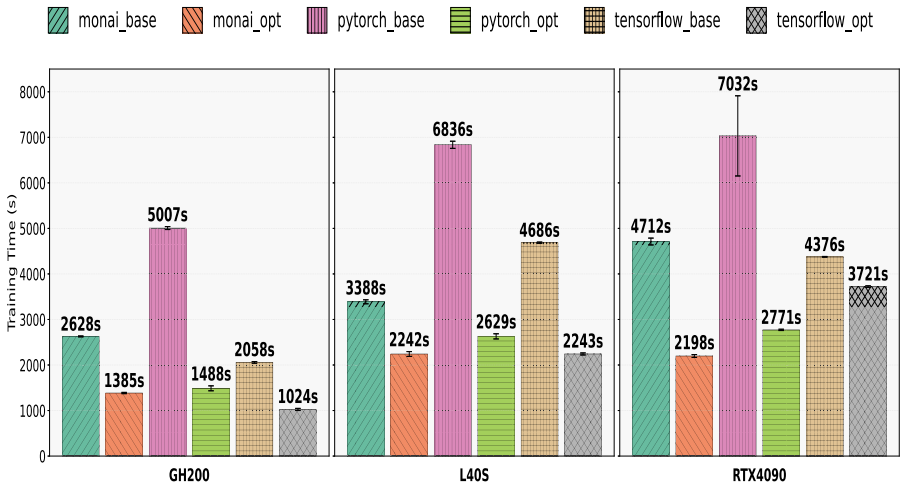

Fig. 5. Mean training time (seconds) per framework.

TF-Opt reduces training time by −50.2 % on the GH200 (2 058 → 1 024 s) and −52.1 % on the L40S (4 686 → 2 243 s), but only −15.0 % on the RTX 4090 (4 376 → 3 721 s). This asymmetry mirrors the throughput divergence discussed in Sect. 5.2: the XLA-compiled DALI pipeline does not extract comparable latency improvements from the consumer GPU architecture. While the L40S benefits from XLA kernel fusion with its larger shared-memory banks, RTX 4090's TDP management and scheduler constraints limit sustained utilization, preventing the same wall-time compression. Consequently, TF-Opt on the RTX 4090 still requires over one hour of training, whereas PT-Opt and MN-Opt finish in under 46 min on the same hardware.

In the MONAI-Opt approach, training time is reduced by −47.3 % on the GH200 (2 628 → 1 385 s), −33.8 % on the L40S (3 388 → 2 242 s), and −53.4 % on the RTX 4090 (4 712 → 2 198 s). The more moderate gains relative to PT-Opt and TF-Opt reflects MONAI modular design, where DALI integration is optional, and not all transforms are offloaded to the GPU by default. It is noteworthy that MN-Opt achieves the *fastest absolute time* on the RTX 4090 (2 198 s), surpassing even PT-Opt (2 771 s), because the cosine scheduler and warmup allow convergence with fewer wasted gradient steps, and the `channels_last` layout interacts favorably with the Ada Lovelace memory subsystem.

5.4 Memory

Table 2 reports the mean peak VRAM usage for the Base and Opt configurations across frameworks and GPUs.

Table 2. Mean peak VRAM usage (MB) for Base and Opt configurations by GPU and framework. Values are averaged across the 10 repetitions.

Framework	GPU	Base	Opt
TensorFlow	GH200	12215.5	1667.5
	L40S	12358.2	4370.5
	RTX 4090	10859.0	5096.8
PyTorch	GH200	18967.2	8364.8
	L40S	18911.0	18646.0
	RTX 4090	18726.0	18600.4
MONAI	GH200	7282.0	6772.0
	L40S	7400.0	6814.0
	RTX 4090	7400.0	6814.0

The decimal values in Table 2 result from averaging the peak VRAM measured across the 10 repetitions of each configuration. MONAI was the most stable and memory-efficient implementation, remaining in a narrow range of approximately 6.8–7.4 GB of VRAM across all GPUs, with a small but consistent reduction in the optimized version. In contrast, PyTorch-Base showed the highest memory footprint, reaching about 18.7–19.0 GB. PyTorch-Opt substantially reduced peak VRAM on the GH200, but only marginal changes were observed on the L40S and RTX 4090. TensorFlow showed the largest memory reduction after optimization: TensorFlow-Opt decreased peak VRAM from about 10.9–12.4 GB to 1.7–5.1 GB, a pattern consistent with the combined effect of mixed precision and backend-level optimizations. Overall, these results indicate that the impact of the optimized pipeline on memory usage was not uniform across frameworks or GPUs, but it was particularly pronounced in TensorFlow and, to a lesser extent, in MONAI.

5.5 Ablation Study of the Optimization Stack

To better understand the contribution of each optimization used in the final pipeline, we performed a component-wise ablation study. Starting from the baseline configuration, we enabled one optimization at a time (DALI, AMP/FP16, cosine learning-rate scheduling, and EMA), and also evaluated the full optimized configuration. Table 3 summarizes the training time, mean throughput, and final validation AUC for all configurations.

As shown in Table 3, AMP/FP16 was the main source of computational gains in this setup. When enabled alone, it reduced total training time from 3865.5 s to 2253.6 s and increased mean training throughput from 671.7 to 1327.6 img/s, while causing only a small drop in final validation AUC from 0.9858 to 0.9824. In contrast, DALI alone did not improve training throughput in this configuration and slightly reduced AUC, suggesting that the baseline was not primarily limited

Table 3. Training time, mean throughput, and final validation AUC for each experimental configuration.

Run	Time (s)	Mean Thr. (img/s)	Final AUC
baseline	3865.5	671.67	0.9858
dali_only	3895.0	659.48	0.9805
fp16_only	2253.6	1327.58	0.9824
cosine_only	3870.5	669.77	0.9851
ema_only	3869.6	668.70	0.9790
full_opt	2526.6	1090.02	0.9854

by the input pipeline. The cosine schedule had a nearly neutral effect on both accuracy and efficiency, indicating that its contribution is better interpreted as training stabilization than as a direct performance booster. EMA alone reduced final validation AUC to 0.9790 without improving training time or throughput, making it the weakest individual component in this ablation.

When all optimizations were combined, the final configuration preserved baseline-level predictive performance, with final validation AUC remaining practically unchanged (0.9858 vs. 0.9854), while reducing training time to 2526.6 s and increasing mean training throughput to 1090.0 img/s. These results indicate that the best overall trade-off was achieved by the combined optimized pipeline, with AMP accounting for most of the speedup and the remaining techniques contributing mainly to the final balance between efficiency and predictive performance.

5.6 Summary of Base-to-Optimized Gains

This section summarizes the overall effect of moving from the Base configuration to the Opt configuration for each framework and GPU. Table 4 reports the absolute AUC variation and the relative changes in throughput, training time, and peak memory usage. Positive values indicate an increase after optimization, whereas negative values indicate a reduction.

Overall, the results show that optimization improved computational efficiency in all frameworks, but with different effects on predictive performance. TensorFlow achieved the largest throughput gains, although with a reduction in AUC. PyTorch showed the strongest balance between AUC improvement and training-time reduction, while MONAI presented smaller but more stable changes across the evaluated platforms.

Table 4. Base-to-Opt changes in AUC, throughput, training time, and memory.

Framework	GPU	AUC	Thr.	Time	Memory
TensorFlow	GH200	−0.027	+186.9%	−50.2%	−86%
	L40S	−0.035	+288.8%	−52.1%	−65%
	RTX 4090	−0.028	+104.1%	−15.0%	−53%
PyTorch	GH200	+0.067	+250.4%	−70.3%	−56%
	L40S	+0.068	+185.3%	−61.5%	−1%
	RTX 4090	+0.066	+183.1%	−60.6%	−1%
MONAI	GH200	+0.005	+71.3%	−47.3%	−7%
	L40S	+0.010	+39.8%	−33.8%	−8%
	RTX 4090	+0.003	+84.1%	−53.4%	−8%

6 Conclusion

Finally, our study compared six training pipelines on a GPU for diabetic retinopathy detection using TensorFlow, PyTorch, and MONAI on three NVIDIA GPU platforms. Overall, AUC varied less between GPUs than between software pipelines, while throughput and training time were strongly influenced by both framework and hardware. TF-Opt achieved the highest throughput on the GH200 and L40S GPUs, while PT-Opt showed the best overall balance, reaching 98.5% AUC on all three GPUs, along with considerable efficiency gains.

All our results reflect the behavior of fully optimized pipelines, and not isolated effects of the structure, since each optimized version combined multiple changes, including GPU-based data loading, mixed precision, compiler optimizations, tensor layout adjustments, and modified training schedules. In this configuration, throughput increased by up to 288.8% and training time decreased by up to 70.3%. The findings suggest that pipeline optimization can make training workflows for diabetic retinopathy more practical in resource-constrained environments. Future work should evaluate larger datasets, multi-GPU environments, and other medical imaging tasks.

Acknowledgments. This research was conducted with support from Petrobras (concessão n.o 2020 /00182-5), CNPq and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Some experiments in this work used the PCAD infrastructure (http://gppd-hpc.inf.ufrgs.br), at INF/UFRGS.

References

1. Alyoubi, W.L., Shalash, W.M., Abulkhair, M.F.: Diabetic retinopathy detection through deep learning techniques: a review. Inform. Med. Unlocked **20**, 100377 (2020)
2. Tsiknakis, N., et al.: Deep learning for diabetic retinopathy detection and classification based on fundus images: a review. Comput. Biol. Med. **135**, 104599 (2021)
3. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2818–2826 (2016)
4. Mutawa, A.M., Alnajdi, S., Sruthi, S.: Transfer learning for diabetic retinopathy detection: a study of dataset combination and model performance. Appl. Sci. **13**(9), 5685 (2023)
5. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L.: ImageNet: a large-scale hierarchical image database. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 248–255 (2009)
6. Bećirović, M., Kurtović, A., Smajlović, N., Kapo, M., Akagić, A.: Performance comparison of medical image classification systems using TensorFlow Keras, PyTorch, and JAX. arXiv arXiv:2507.14587 (2025)
7. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, vol. 32, pp. 8024–8035 (2019)
8. Li, S., et al.: PyTorch distributed: experiences on accelerating data parallel training. Proc. VLDB Endow. **13**(12), 3005–3018 (2020)
9. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16, pp. 265–283 (2016)
10. Nguyen, Q.-H., Doan, H., Thoai, N.: Performance evaluation of distributed training in Tensorflow 2. In: 2020 International Conference on Advanced Computing and Applications (ACOMP), pp. 155–159 (2020)
11. OpenXLA Project: XLA architecture. https://openxla.org/xla/architecture. Accessed 20 Mar 2026
12. Cardoso, M.J., Li, W., Brown, R., et al.: MONAI: an open-source framework for deep learning in healthcare. arXiv arXiv:2211.02701 (2022)
13. Sebastian, A., Elharrouss, O., Al-Maadeed, S., Almaadeed, N.: A survey on deep-learning-based diabetic retinopathy classification. Diagnostics **13**(3), 345 (2023)
14. Bhulakshmi, D., Rajput, D.S.: A systematic review on diabetic retinopathy detection and classification based on deep learning techniques using fundus images. PeerJ Comput. Sci. **10**, e1947 (2024)
15. Zhou, Y., Wang, B., Huang, L., Cui, S., Shao, L.: A benchmark for studying diabetic retinopathy: segmentation, grading, and transferability. IEEE Trans. Med. Imaging **40**(3), 818–828 (2021)
16. Martínez-Murcia, F.J., Ortiz, A., Ramírez, J., Górriz, J.M., Cruz, R.: Deep residual transfer learning for automatic diagnosis and grading of diabetic retinopathy. Neurocomputing **452**, 424–434 (2021)
17. Hacisoftaoglu, R.E., Karakaya, M., Sallam, A.B.: Deep learning frameworks for diabetic retinopathy detection with smartphone-based retinal imaging systems. Pattern Recogn. Lett. **135**, 409–417 (2020)
18. Alam, M.N., et al.: Contrastive learning-based pretraining improves representation and transferability of diabetic retinopathy classification models. Sci. Rep. **13**(1), 6047 (2023)

19. Goh, J.H.L., et al.: Comparative analysis of vision transformers and conventional convolutional neural networks in detecting referable diabetic retinopathy. Ophthalmol. Sci. **4**(6), 100552 (2024)
20. Gulshan, V., et al.: Performance of a deep-learning algorithm vs manual grading for detecting diabetic retinopathy in India. JAMA Ophthalmol. **137**(9), 987–993 (2019)
21. Dörrich, M., Fan, M., Kist, A.M.: Impact of mixed precision techniques on training and inference efficiency of deep neural networks. IEEE Access **11**, 57627–57634 (2023)
22. He, X., Sun, J., Chen, H., Li, D.: Campo: cost-aware performance optimization for mixed-precision neural network training. In: 2022 USENIX Annual Technical Conference, USENIX ATC '22, pp. 505–518 (2022)
23. Li, B., et al.: Optimizing data layout for training deep neural networks. In: Companion Proceedings of the Web Conference 2022, pp. 548–554 (2022)
24. Um, T., Oh, B., Seo, B., Kweun, M., Kim, G., Lee, W.-Y.: FastFlow: accelerating deep learning model training with smart offloading of input data pipeline. Proc. VLDB Endow. **16**(5), 1086–1099 (2023)
25. Leclerc, G., Ilyas, A., Engstrom, L., Park, S.M., Salman, H., Mądry, A.: FFCV: accelerating training by removing data bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 12011–12020 (2023)
26. dos Reis, M.A., Künas, C.A., et al.: Advancing healthcare with artificial intelligence: diagnostic accuracy of machine learning algorithm in diagnosis of diabetic retinopathy in the Brazilian population. Diabetol. Metab. Syndr. **16**, 209 (2024)
27. Balouek, D., et al.: Adding virtualization capabilities to the Grid'5000 testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) CLOSER 2012. CCIS, vol. 367, pp. 3–20. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-04519-1_1
28. NVIDIA: Core Concepts. In: NVIDIA cuDNN Frontend Documentation. https://docs.nvidia.com/deeplearning/cudnn/frontend/latest/developer/core-concepts.html. Accessed 20 Mar 2026
29. Chan, H.-L., et al.: Deep neural network for the detections of fall and physical activities using foot pressures and inertial sensing. Sensors **23**(1), 495 (2023)
30. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). Biometrika **52**(3/4), 591–611 (1965)
31. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics Bull. **1**(6), 80–83 (1945)