ComTech: Computer, Mathematics and Engineering Applications, 17(1), June 2026, 63−72 P-ISSN: 2087-1244
DOI: 10.21512/comtech.v17i1.14648 E-ISSN: 2476-907X

Comparative Performance Analysis of Object-Oriented Programming and Data-Oriented Programming in TensorFlow

Mangapul Siahaan[1]*; Jefriyanto Chandra[2]; Muhamad Dody Firmansyah[3]

[1-3]Prodi Sistem Informasi, Fakultas Ilmu Komputer, Universitas Internasional Batam, Batam, Indonesia, 29442

[1]mangapul.siahaan@uib.ac.id; [2]2231067.jefriyanto@uib.edu; [3]dody.firmansyah@uib.edu

Received: 4th November 2025/ **Revised:** 20th January 2026/ **Accepted:** 27th January 2026

How to Cite: Siahaan, M., Chandra, J., & Firmansyah, M. D. (2026). Comparative performance analysis of object-oriented programming and data-oriented programming in TensorFlow. *ComTech: Computer, Mathematics and Engineering Applications, 17*(1), 63−72. https://doi.org/10.21512/comtech.v17i1.14648

Abstract **-** The rapid advancement of deep learning significantly increases computational demands, making performance optimization essential for model scalability and deployment. While numerous studies optimize neural network architectures, the effect of different programming paradigms on computational efficiency remains insufficiently explored. This study aims to compare Object-Oriented Programming (OOP) and Data-Oriented Programming (DOP) paradigms in TensorFlow-based deep learning workflows, focusing on their performance across four processing phases: build, compile, train, and evaluate, under a controlled experimental environment with repeated iterations and systematic measurements. Both paradigms are implemented using identical Convolutional Neural Network (CNN) architectures trained on the CIFAR-100 image dataset over thirty controlled experimental iterations. A custom profiler integrating the Python System and Process Utilities (psutil) and NVIDIA Management Library (pynvml) monitors real-time system performance, capturing CPU and GPU utilization as well as memory usage. The results reveal that DOP achieves better resource efficiency with lower memory usage (549.98 MB versus 676.25 MB), higher GPU utilization (64.68% versus 61.08%), and faster evaluation execution (1.50 seconds versus 2.59 seconds), while also attaining higher model accuracy (32.38% versus 28.08%). In contrast, OOP benefits from TensorFlow's Sequential API optimizations, resulting in faster training times but greater CPU and memory consumption. These findings highlight that DOP provides superior runtime efficiency and offers practical benefits for performance-critical deep learning applications.

Keywords**:** Object-Oriented Programming, Data-Oriented Programming, TensorFlow, computational efficiency

I. INTRODUCTION

In recent years, the rapid advancement of artificial intelligence and deep learning leads to a sharp increase in computational demands (Sevilla et al., 2022). Consequently, performance bottlenecks become more prominent and are influenced not only by neural network architecture but also by the programming paradigm and the quality of the data pipeline. The data pipeline is a core component of the deep learning workflow and is responsible for preparing, batching, and delivering data to the model.

Even with identical neural architectures, differences in programming paradigms produce measurable variations in execution time, memory usage, and overall throughput (Murray et al., 2021). These factors are crucial for achieving scalable and resource-efficient implementations, underscoring the need to examine not only how models are structured but also how the underlying code is organized. As computational demands continue to grow, the choice of programming paradigm becomes an essential component of system performance and influences how effectively a deep learning workflow utilizes available hardware resources.

As model complexity continues to rise, understanding the influence of the programming paradigm on performance becomes increasingly critical for efficient resource utilization. Object-Oriented Programming (OOP) is a widely used paradigm in many machine learning frameworks and is valued for its modular structure and readability. By organizing code into classes and reusable components, OOP facilitates easier maintenance and scalability in complex systems. However, these advantages come at the cost of additional abstraction layers, such as class hierarchies and inheritance structures. These abstractions may introduce computational overhead

*Corresponding Author

and reduce overall performance efficiency in deep learning workflows.

In contrast, Data-Oriented Programming (DOP) emphasizes direct data processing with minimal encapsulation. This approach prioritizes efficient data handling, reduces object-related overhead, and optimizes data flow (Mironov et al., 2021; Wingqvist et al., 2022). While the conceptual contrast between OOP and DOP provides a theoretical foundation, practical evidence from existing research further illustrates the importance of data handling and processing efficiency in deep learning pipelines. This principle is also reflected in other computational domains, such as text mining for identifying student learning problems (Christian, 2022) and algorithm comparison studies for hate speech classification (Suwarno & Kusnadi, 2021), where design choices strongly influence performance and reinforce the broader relevance of how workflow structure affects computational efficiency.

Several studies highlight the importance of pipeline efficiency. For instance, the TensorFlow tf.data framework introduces parallel mapping, caching, and prefetching, which significantly reduce GPU idle time and improve training throughput (Murray et al., 2021). Further enhancements through the tf.data service decouple data preprocessing from training, resulting in up to a 31.7 times increase in throughput and reduced training costs (Audibert et al., 2023). A context-aware data-loading framework, Cedar, is also developed to optimize runtime efficiency and achieves a 10.65 times improvement in data handling performance (Zhao et al., 2025).

In addition to improving throughput, these studies collectively show that even small adjustments to data handling significantly influence overall performance. This suggests that performance is not determined solely by hardware but also by how efficiently the software structure interacts with the data pipeline. Therefore, the choice of programming paradigm becomes increasingly relevant as models and datasets continue to grow.

In addition to these improvements, several researchers examine the negative consequences of inefficient pipeline management and reveal how suboptimal data flow significantly degrades overall training throughput. Studies report that non-optimized pipelines reduce throughput by up to 13 times (Isenko et al., 2022), while data ingestion and validation processes consume more computation time than model training itself (Xin et al., 2021). Automation approaches are also proposed to address these inefficiencies, such as AutoML frameworks that optimize hyperparameters and network structures to improve overall pipeline performance (Filippou et al., 2023). Similarly, optimized real-time streaming pipelines process data up to 30–90 times faster than conventional Python-based implementations (Im et al., 2024).

While these studies focus primarily on architectural and algorithmic improvements, few investigate how the implemented programming paradigm itself affects performance. Evidence from other fields suggests that DOP outperforms OOP by reducing cache misses and improving scalability. In game development, studies report a 13.25 times performance gain for DOP (Wingqvist et al., 2022), while other research finds 33% faster computation and improved multicore scalability in a trading strategy backtester (Mironov et al., 2021). It is also shown that separating code from data simplifies system design and reduces complexity (Sharvit, 2022). The TensorFlow documentation highlights the importance of asynchronous data delivery through prefetching and parallel mapping to maximize throughput (TensorFlow Team, 2024).

Research on data quality also shows that high-quality data has a significant impact on pipeline performance. Compliance with strong data quality standards significantly enhances model performance (Rangineni, 2023). Similarly, adopting data-centric and flow-based programming architectures improves efficiency and maintainability in machine learning systems (Paleyes et al., 2022).

These findings collectively underscore that the efficiency of data processing frameworks directly affects model scalability and reliability. Efficient pipelines not only reduce computational overhead but also ensure that hardware resources are utilized more effectively during training and inference. However, despite extensive work on optimization techniques, the effect of the programming paradigm itself remains largely unexplored in deep learning workflows.

Considering the potential advantages of DOP and the prevalence of OOP in machine learning frameworks, it becomes necessary to examine how these paradigms affect the performance of deep learning workflows. This study aims to systematically compare OOP and DOP implementations in TensorFlow and evaluate their impact on execution time, memory usage, CPU utilization, and GPU utilization. By focusing on identical architectures and datasets, this study isolates the performance impact of the programming paradigm itself rather than differences in algorithmic complexity. The objective is not only to determine which paradigm is faster but also to observe how each paradigm influences hardware utilization, memory behavior, and training consistency. Understanding these relationships provides valuable insight for developers who optimize TensorFlow pipelines for efficiency without sacrificing maintainability.

II. METHODS

This study uses a quantitative experimental approach to systematically evaluate the performance of OOP and DOP paradigms in TensorFlow-based deep learning workflows. The experiment measures several performance metrics, including CPU and GPU utilization, execution time, and memory usage. The research follows a structured performance evaluation framework, adapted from existing guidelines, which emphasizes consistent evaluation metrics and

statistically valid comparisons (Rainio et al., 2024). The experimental setup is also designed to ensure systematic configuration, reproducibility, and control over external factors that could influence the results (Arboretti et al., 2022).

This study is conducted in multiple stages, consisting of environment configuration, dataset preparation, paradigm implementation, experiment execution, data collection and statistical analysis, and result visualization and interpretation. Each phase is designed to ensure that the comparison between paradigms remains systematic, reproducible, and free from external interference. The overall research flow is illustrated in Figure 1, which shows the experimental workflow from environment configuration to result visualization and interpretation.

The research workflow consisted of six sequential phases as shown in Figure 1. The first phase, environment configuration, involved setting up the computational environment and ensuring reproducibility across all experimental runs. This phase included installing Python 3.9.13, TensorFlow 2.10.0, CUDA 11.2.0, cuDNN 8.1.0, and Visual Studio Code as the development environment. The hardware used for this experiment is shown in Table 1.

The second phase, dataset preparation, focuses on processing the CIFAR-100 dataset, which consists of 50,000 training images and 10,000 test images at 32×32 pixels, divided into 100 classes and 20 superclasses. This choice aligns with prior studies that use CIFAR-100 as a standard benchmark for deep learning performance assessment (Meir et al., 2024; Rafidison et al., 2023). All images are normalized and prepared using TensorFlow's tf.data API, which implements techniques such as caching, prefetching, and parallel mapping to improve throughput. CIFAR-100 is selected for its moderate complexity, enabling performance differences to be observed without reaching computational limits. Its balanced distribution across 100 classes ensures a consistent GPU workload across training iterations and helps isolate the effect of paradigm implementation rather than dataset bias. An overview of the CIFAR-100 dataset used in this study is shown in Figure 2.

The third phase, paradigm implementation, involves developing two TensorFlow workflows with identical Convolutional Neural Network (CNN) architectures but using different programming paradigms. The CNN structure consists of three convolutional layers and two dense layers. The architecture is deliberately kept minimal to emphasize the influence of the programming paradigm rather than architectural complexity. By maintaining the same layer arrangement, activation functions, and optimizer parameters across both implementations, any performance difference is attributed to how each

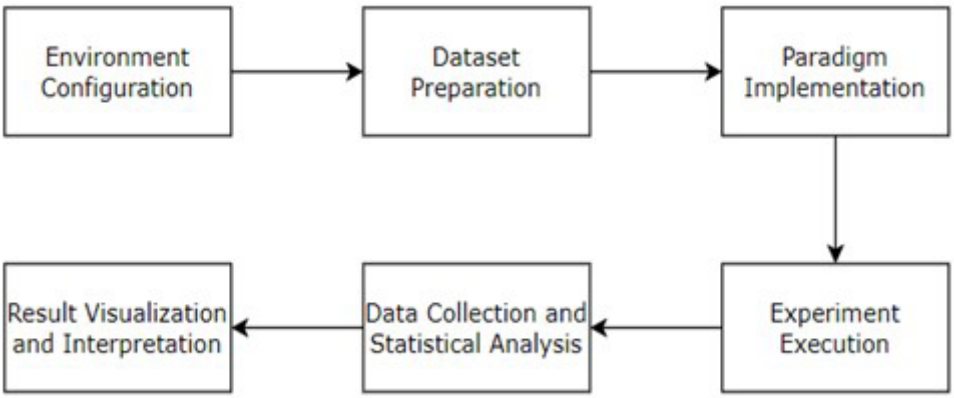

Figure 1 Research Flow

Table 1 Hardware and Software Specifications

No.	Component	Specifications
1	CPU	Ryzen 7 4800H
2	RAM	16GB (2x8GB) DDR4 3200Mhz
3	Storage	512 GB M.2 NVMe
4	GPU	NVIDIA GeForce RTX 3050 4GB
5	OS	Windows 11
6	Framework	TensorFlow 2.10.0, CUDA 11.2.0, cuDnN 8.1.0
7	Python	Version 3.9.13

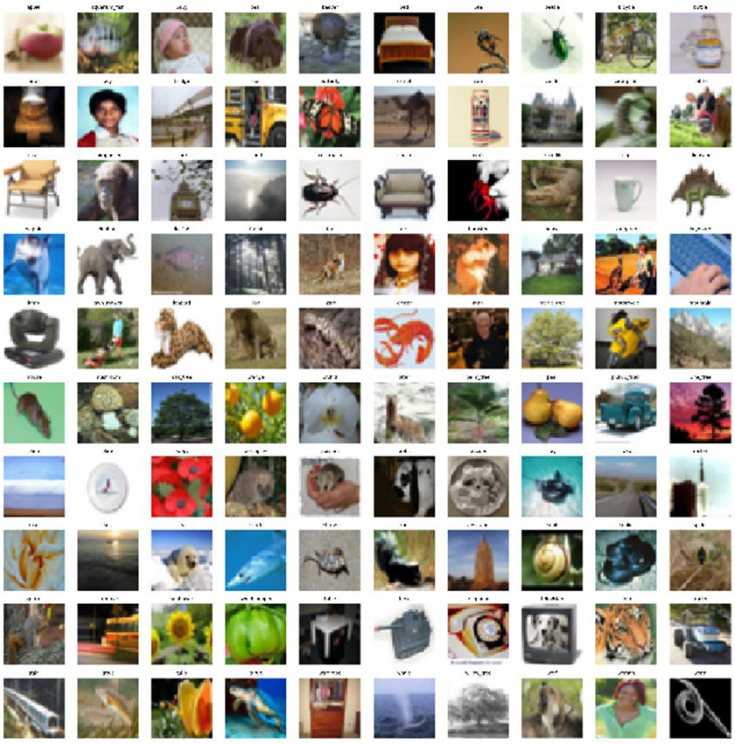

Figure 2 CIFAR-100 Dataset

```
class CNNModel(BaseModel):
    def build_model(self) -> tf.keras.Model:
        self.conv_layers = [
            tf.keras.layers.Conv2D(32, (3, 3)),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.Activation('relu'),
            # ...
        ]

        self.dense_layers = [
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(128),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.Activation('relu'),
            # ...
            tf.keras.layers.Dense(self.num_classes, activation='softmax')
        ]

        self.model = tf.keras.Sequential(self.conv_layers + self.dense_layers)
        return self.model
```

Figure 3 OOP Implementation

paradigm manages execution flow, data encapsulation, and callback handling within TensorFlow.

The OOP implementation applies encapsulation and inheritance through classes such as BaseModel, CNNModel, and ModelTrainer, and integrates callbacks such as ModelCheckpoint and EarlyStopping to reflect common object-oriented design practices (Lott & Phillips, 2021). This structure organizes the workflow into modular components, allowing each class to manage a specific function within the training pipeline. Such modularity enhances code readability and maintainability, but it also introduces additional abstraction layers that may lead to execution overhead.

The OOP implementation, as shown in Figure 3, encapsulates model construction and training logic into reusable classes. This modular structure promotes maintainability but introduces additional abstraction layers such as callback objects and inherited class initialization. These layers increase the number of function calls, which, while effective for readability and extensibility, add minor latency and memory overhead during object creation and callback registration. The profiler data confirms that this initialization process slightly elevates memory usage during the build and

```
def create_model_architecture(input_shape: Tuple[int, ...], num_classes: int) -> tf.keras.Model:
    inputs = tf.keras.Input(shape=input_shape)
    x = tf.keras.layers.Conv2D(32, (3, 3))(inputs)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation('relu')(x)
    # ...
    x = tf.keras.layers.Dense(128)(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation('relu')(x)
    # ...
    outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)
    return tf.keras.Model(inputs=inputs, outputs=outputs)
```

Figure 4 DOP Implementation

compile phases.

In contrast, the DOP implementation adopts a functional approach, eliminates abstraction layers, and structures operations through sequential functions (create_model(), train_model(), and evaluate_model()), thereby minimizing overhead and enhancing data locality while maintaining identical model behavior. This design allows each operation to be executed directly without the need for class instantiation or inherited methods. By streamlining execution, the DOP paradigm enables TensorFlow to compute more efficiently and improves overall resource utilization.

Figure 4 illustrates the DOP workflow, which prioritizes direct data and computation flow through sequential functions. By eliminating class-based encapsulation, DOP minimizes runtime context switching and reduces memory fragmentation. The resulting pipeline shows reduced latency in the data path between preprocessing and GPU execution. These structural efficiencies explain the lower mean memory consumption and higher GPU utilization observed in subsequent experimental results. The simplicity of the DOP structure also facilitates better optimization by TensorFlow's internal execution graph.

The fourth phase, experiment execution, consists of running each paradigm under a controlled environment for 30 iterations to ensure statistical validity. Each iteration includes four stages—build, compile, train, and evaluate—which are monitored by a custom profiler class that integrates psutil and pynvml to record real-time performance metrics (M et al., 2024). The profiler uses a start–stop mechanism to capture baseline and post-execution data, ensuring consistency across runs and accurate resource usage records. TensorFlow sessions are cleared using the tf.keras.backend.clear_session() function, and garbage collection is performed using the gc.collect() function to prevent memory leaks.

To maintain consistency, all experiments are run under identical system conditions, and no other background processes are allowed to occupy GPU resources. This setup helps reduce uncontrolled variability across iterations and ensures that each measurement reflects the behavior of the programming paradigm rather than external system interference. As a result, performance noise from background processes is minimized, and the results provide more reliable data for comparing both paradigms.

The fifth phase, data collection and statistical analysis, compiles all recorded metrics into structured Pandas DataFrames for processing. The collected data include execution time (s), CPU usage (%), GPU utilization (%), memory usage (MB), and model accuracy (%). Statistical analysis employs Welch's t-test to determine whether performance differences between OOP and DOP are statistically significant, as computed using Equation (1).

$$t = \frac{\dot{X}_1 - \dot{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \quad (1)$$

Equation (1) shows the t-test formula, where $\dot{X}_1$ and $\dot{X}_2$ represent the mean values, s_1^2 and s_2^2 denote the standard deviations, and n_1 and n_2 indicate the number of iterations for each paradigm. This formulation is appropriate when the two samples may have unequal variances, making it suitable for evaluating performance metrics that fluctuate across experimental runs. By applying this statistical approach, the study ensures that observed differences between OOP and DOP are not due to random variation but reflect real distinctions in computational behavior. This study follows the approach suggested by prior research, which evaluates both accuracy and resource efficiency to emphasize the relationship between computational cost and model performance (Cueto-Mendoza & Kelleher, 2024).

The final phase, result visualization and interpretation, presents the findings using Matplotlib and Seaborn to visualize performance metrics. These tools are used to create clear and structured plots that effectively summarize trends across all iterations. Graphs illustrating CPU and GPU utilization, memory consumption, execution time, and accuracy provide clearer interpretation of performance differences

between the two paradigms. By visualizing these metrics side by side, the study highlights patterns that may not be immediately apparent from numerical tables alone and supports a more comprehensive understanding of how each programming paradigm behaves across different stages of the deep learning workflow.

III. RESULTS AND DISCUSSIONS

The experiment comparing OOP and DOP is conducted on the CIFAR-100 dataset across 30 controlled iterations. Each paradigm is executed through four phases—build, compile, train, and evaluate—with each phase measuring specific performance metrics. The descriptive statistics for all recorded metrics are presented in Table 2.

The values summarized in Table 2 provide an overview of performance consistency across phases. These descriptive statistics highlight differences in execution time, memory consumption, and hardware utilization, revealing where each paradigm demonstrates strengths or limitations. To better understand these patterns, each stage of the deep learning workflow is discussed in detail below.

During the build phase, DOP shows significantly higher efficiency compared to OOP. Its average execution time is shorter, and it consumes less memory, while CPU usage remains minimal. This efficiency results from DOP's functional structure, which directly constructs computational graphs without class-based abstractions and minimizes initialization overhead. In contrast, OOP's encapsulated model creation through class hierarchies consumes additional memory and computational resources during this phase.

The initialization overhead in OOP primarily manifests as object instantiation costs. Each class constructor allocates memory for instance variables, establishes inheritance hierarchies, and registers callback functions with TensorFlow's computation graph. These operations, while individually minimal, accumulate across multiple object creations. DOP avoids this overhead by maintaining a flat execution structure in which functions operate directly on data without intermediate object layers.

In the compile phase, both paradigms exhibit nearly identical execution times, suggesting that compilation overhead is primarily determined by the TensorFlow framework rather than the programming paradigm. However, DOP consistently shows lower memory usage, indicating greater efficiency. OOP's slightly higher resource consumption is associated with callback mechanisms such as ModelCheckpoint and TensorBoard, which add additional layers of abstraction during compilation.

During training, OOP displays a slightly faster average execution time of 48.05 seconds compared to 51.57 seconds for DOP. This result aligns with TensorFlow's optimization of the Sequential API used in the OOP design, which is tuned for linear model structures. However, DOP demonstrates superior efficiency in memory management and GPU utilization. Its average memory usage is 549.98 MB, lower than OOP's 676.52 MB, and GPU utilization is higher (64.68% versus 61.08%), indicating greater consistency in hardware utilization. These results reveal a trade-off between execution speed and resource efficiency, where OOP benefits from TensorFlow's internal optimizations while DOP achieves more efficient hardware utilization.

While training results reveal a balance between TensorFlow's built-in optimizations and DOP's resource efficiency, evaluation performance highlights how each paradigm behaves when the workload transitions from learning to inference. This phase changes the computational demands from iterative runs to streamlined execution. As a result, the evaluation phase provides a more direct measure of how effectively each programming paradigm manages resources.

The evaluation phase revealed the largest performance gap between the paradigms. DOP completed the evaluation in 1.50 seconds, nearly 1.73 times faster than OOP's 2.59 seconds. It also used

Table 2 Summary of Descriptive Statistics

Paradigm	Phase	Execution Time (mean ± std)	Memory Usage (mean ± std)	CPU Usage (mean ± std)	GPU Utilization (mean ± std)
OOP	Build	0.2966 ± 0.1202 s	15.9495 ± 87.2268 MB	3.1467 ± 6.9093 %	0.0 ± 0.0 %
OOP	Compile	0.270 ± 0.0069 s	0.0367 ± 0.0883 MB	0.4167 ± 2.2822 %	0.0 ± 0.0 %
OOP	Train	48.0538 ± 8.5351 s	676.2503 ± 399.022 MB	1.25 ± 5.0322 %	61.0809 ± 1.19 %
OOP	Evaluate	2.59 ± 0.0473 s	238.8431 ± 2.2601 MB	0.8933 ± 3.4079 %	21.31 ± 1.5575 %
DOP	Build	0.2712 ± 0.0062 s	0.022 ± 0.0044 MB	0.4167 ± 2.2822 %	0.0 ± 0.0 %
DOP	Compile	0.269 ± 0.0063 s	0.021 ± 0.0019 MB	0.8933 ± 3.4079 %	0.0 ± 0.0 %
DOP	Train	51.5738 ± 0.6188 s	549.9815 ± 236.025 MB	0.8333 ± 4.5644 %	64.6788 ± 0.9697 %
DOP	Evaluate	1.4977 ± 0.103 s	117.7844 ± 1.2277 MB	0.0 ± 0.0 %	16.82 ± 2.7037 %

significantly less memory with an average usage of 117.78 MB compared to OOP's 238.84 MB. GPU utilization during this phase was also lower in DOP (16.82%), while OOP utilized 21.31%, indicating that DOP has lower overhead and improved efficiency. These results confirm that DOP is better suited to resource-constrained environments. In contrast, OOP offers advantages in scenarios where speed is prioritized, and TensorFlow's built-in optimizations can be fully leveraged.

To provide a clearer visualization of the descriptive statistics, a comparative plot is shown in Figure 5. This figure presents a consolidated view of the performance metrics for both OOP and DOP and enables a systematic comparison of execution time, memory usage, CPU activity, and GPU utilization. The graphical representation facilitates the identification of performance trends that may not be immediately evident from tabular data alone and supports a more comprehensive interpretation of the experimental results.

Figure 5 compares the performance of each metric across the four phases for both OOP and DOP. The training phase stands out as the most resource-intensive stage, with high values for execution time, memory usage, and GPU utilization. While OOP shows a slightly lower average training time, DOP demonstrates more consistent GPU utilization and notably lower memory consumption. These results indicate that DOP uses computational resources more efficiently.

The evaluation phase illustrates the most noticeable performance gap between the paradigms. DOP completes the evaluation faster and consumes fewer resources than OOP. This improved efficiency results from DOP's functional approach and streamlined data processing, which reduce computational overhead. In contrast, OOP's use of class hierarchies and callback mechanisms leads to higher memory usage and slower execution.

During the training phase, DOP achieves higher GPU utilization, averaging nearly 65%, compared to approximately 61% for OOP. However, during the evaluation phase, OOP shows slightly higher GPU utilization but with longer execution times and higher memory consumption. These results indicate that DOP leverages GPU acceleration more effectively, while OOP's abstraction layers may limit overall hardware efficiency.

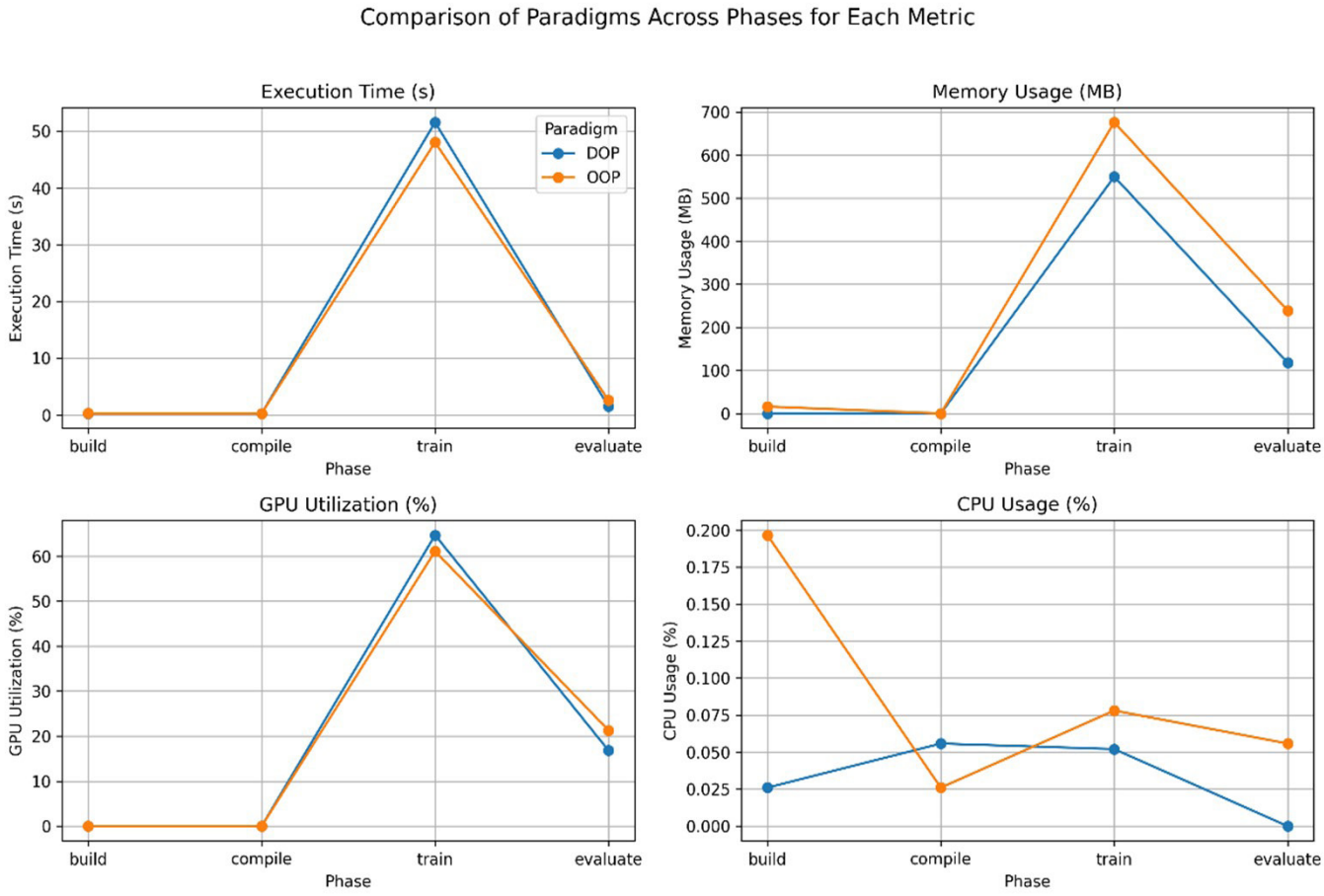

Figure 5 Comparison of Performance in each metric

This test is selected due to its suitability for samples with unequal variances and potentially different sample sizes, making it appropriate for comparing performance measurements across experimental iterations. The statistical outcomes, including t-statistics and p-values for each metric and phase, are presented in Table 3 and provide a basis for assessing the significance and consistency of the differences between the two programming paradigms. This approach helps ensure that the comparison between paradigms is fair and statistically valid. The results support the comparative analysis presented in this study.

The statistical analysis shows that although some differences are not statistically significant, notable differences emerge in CPU usage during the build phase, execution time during training, and several evaluation metrics. These findings indicate that certain performance variations are inherent to the structural design of each programming paradigm. DOP shows statistically significant improvements during the evaluation phase, demonstrating clearer performance gains as the workflow transitions from training to evaluation.

Model accuracy is also compared to ensure that computational efficiency does not negatively impact predictive performance. As shown in Table 4, DOP achieves a slightly higher accuracy of 32.38% compared to OOP's 28.08%. The smaller standard deviation observed in DOP indicates more stable model convergence, suggesting that its streamlined data processing improves learning consistency. Although OOP's EarlyStopping callback shortens training time, it may prevent full convergence, which explains the slightly lower accuracy.

This result reflects a common trade-off between automation and control. OOP's reliance on callback-driven stopping criteria leads to early termination when short-term validation loss stabilizes, whereas DOP's manual training loop runs to completion across all epochs. The latter approach provides steadier gradient updates and contributes to marginally higher accuracy and lower standard deviation across iterations. Lower variance in accuracy metrics indicates more consistent and predictable model behavior across training runs, which is especially valuable during hyperparameter tuning and model selection. This consistency is particularly important in scenarios that require strict performance guarantees, such as safety-critical applications or regulatory compliance.

Overall, these findings show a clear trade-off between paradigms. OOP achieves faster build and training times due to TensorFlow's optimizations but consumes more memory and CPU resources because of class-based abstraction layers. In contrast, DOP provides more efficient memory usage, higher GPU

Table 3 Welch's T-Test Results

Phase	Metric	OOP Mean	DOP Mean	t-Statistic	p-Value	Significance
Build	Execution Time (s)	0.297	0.271	1.155	0.257	Not Significant
Build	Memory Usage (MB)	15.949	0.022	1.000	0.326	Not Significant
Build	CPU Usage (%)	0.002	0.000	2.055	0.047	Significant
Build	GPU Utilization (%)	0.000	0.000	NaN	NaN	Not Significant
Compile	Execution Time (s)	0.270	0.269	0.611	0.543	Not Significant
Compile	Memory Usage (MB)	0.037	0.021	0.978	0.336	Not Significant
Compile	CPU Usage (%)	0.000	0.001	-0.637	0.527	Not Significant
Compile	GPU Utilization (%)	0.000	0.000	NaN	NaN	Not Significant
Train	Execution Time (s)	48.054	51.574	-2.253	0.032	Significant
Train	Memory Usage (MB)	676.250	549.982	1.492	0.142	Not Significant
Train	CPU Usage (%)	0.001	0.001	0.336	0.738	Not Significant
Train	GPU Utilization (%)	61.081	64.679	-12.838	< 0.001	Significant
Evaluate	Execution Time (s)	2.590	1.498	52.774	< 0.001	Significant
Evaluate	Memory Usage (MB)	238.843	117.784	257.795	< 0.001	Significant
Evaluate	CPU Usage (%)	0.001	0.000	1.436	0.162	Not Significant
Evaluate	GPU Utilization (%)	21.310	16.820	7.882	< 0.001	Significant

Table 4 Model Accuracy and Performance Comparison

Paradigm	Accuracy Mean	Accuracy Std	Training Time (s)	Memory Usage (MB)
OOP	28.08%	4.94%	48.0538	676.2503
DOP	32.38%	2.08%	51.5738	549.9815

utilization, and faster evaluation while maintaining higher accuracy and stability. These results indicate that while OOP offers modularity and readability, DOP delivers superior computational efficiency.

IV. CONCLUSIONS

This study demonstrates that programming paradigms have a measurable impact on the performance of deep learning workflows implemented in TensorFlow. The comparison between OOP and DOP shows that each paradigm offers distinct trade-offs in terms of computational efficiency, resource utilization, and model performance. DOP consistently achieves lower memory usage, higher GPU utilization, and faster evaluation time, indicating clear advantages in resource efficiency and inference performance. In contrast, OOP benefits from TensorFlow's built-in optimizations and achieves shorter training times, but at the cost of higher memory and CPU consumption. Furthermore, DOP produces models with higher and more stable accuracy, suggesting that its simplified data flow promotes more consistent learning outcomes.

Overall, DOP provides a more efficient and stable approach for performance-oriented deep learning implementations, whereas OOP remains advantageous for modular development and faster training cycles. The streamlined data flow in DOP enables more efficient resource utilization, particularly during training and evaluation. In contrast, OOP benefits from TensorFlow's built-in optimizations, which reduce training time but introduce additional abstraction overhead. These findings highlight the importance of selecting an appropriate programming paradigm to balance performance, efficiency, and maintainability in deep learning system design.

This study uses a single CNN architecture and the CIFAR-100 dataset to ensure controlled comparisons. While this experimental design isolates the impact of the programming paradigm, it may limit the generalizability of the findings to more complex architectures. Models such as ResNet or Transformer-based architectures may exhibit different performance characteristics due to differences in structure and computational requirements.

Future research can extend this comparison to other paradigms, such as functional or procedural programming, to provide a broader perspective on software design choices. Studies may also explore the effects of distributed training frameworks such as tf.distribute or PyTorch Distributed Data Parallel (DDP). In addition, evaluating performance under varying batch sizes, optimizers, and dataset complexities would help identify how different configurations influence computational efficiency. Expanding this evaluation would provide deeper insights into how software design principles affect deep learning performance at scale.

AUTHOR CONTRIBUTIONS

Conceived and designed the analysis; Collected the data; Contributed data or analysis tools; Performed the analysis; Wrote the paper, M. S., J. C. and M. D. F.

DATA AVAILABILITY

The authors confirm that the data supporting the research findings are available in the article and its supplementary materials.

REFERENCES

Arboretti, R., Ceccato, R., Pegoraro, L., & Salmaso, L. (2022). Design of experiments and machine learning for product innovation: A systematic literature review. In *Quality and Reliability Engineering International* (Vol. 38, Issue 2, pp. 1131–1156). John Wiley and Sons Ltd. https://doi.org/10.1002/qre.3025

Audibert, A., Chen, Y., Graur, D., Klimovic, A., Šimša, J., & Thekkath, C. A. (2023). tf.data service: A case for disaggregating ML input data processing. *SoCC 2023 - Proceedings of the 2023 ACM Symposium on Cloud Computing*, 358–375. https://doi.org/10.1145/3620678.3624666

Cueto-Mendoza, E., & Kelleher, J. (2024). A framework for measuring the training efficiency of a neural architecture. *Artificial Intelligence Review*, *57*(12), 349. https://doi.org/10.1007/s10462-024-10943-8

Filippou, K., Aifantis, G., Papakostas, G. A., & Tsekouras, G. E. (2023). Structure learning and hyperparameter optimization using an Automated Machine Learning (AutoML) pipeline. *Information (Switzerland)*, *14*(4), 232. https://doi.org/10.3390/info14040232

Im, J., Lee, J., Lee, S., & Kwon, H. Y. (2024). Data pipeline for real-time energy consumption data management and prediction. *Frontiers in Big Data*, *7,* 1308236. https://doi.org/10.3389/fdata.2024.1308236

Isenko, A., Mayer, R., Jedele, J., & Jacobsen, H. A. (2022). Where is my training bottleneck? Hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data* (pp. 1825-1839). https://doi.org/10.1145/3514221.3517848

Liang, S. (2021). Comparative analysis of SVM, XGBoost and Neural Network on hate speech classification. *Jurnal RESTI*, *5*(5), 896–903. https://doi.org/10.29207/resti.v5i5.3506

Lott, S. F., & Phillips, Dusty. (2021). *Python object-oriented programming: Build robust and maintainable object-oriented Python applications and libraries*. Packt Publishing.

M, P., J, A. B., & E.S, S. (2024). Real-time web server monitoring system using Python. *Journal of Artificial Intelligence and Capsule Networks*, *6*(3), 332–339. https://doi.org/10.36548/jaicn.2024.3.006

Meir, Y., Tzach, Y., Hodassman, S., Tevet, O., & Kanter, I. (2024). Towards a universal mechanism for successful deep learning. *Scientific Reports*, *14*(1). https://doi.org/10.1038/s41598-024-56609-x

Mironov, T., Motaylenko, L., Andreev, D., Antonov, I., & Aristov, M. (2021). Comparison of object-oriented programming and data-oriented design for implementing trading strategies backtester. *Vide. Tehnologija. Resursi - Environment, Technology, Resources*, *2*, 124–130. https://doi.org/10.17770/etr2021vol2.6629

Murray, D. G., Šimša, J., Klimovic, A., & Indyk, I. (2021). tf.Data: A machine learning data processing framework. *Proceedings of the VLDB Endowment*, *14*(12), 2945–2958. https://doi.org/10.14778/3476311.3476374

Paleyes, A., Cabrera, C., & Lawrence, N. D. (2022). An empirical evaluation of flow based programming in the machine learning deployment context. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI* (pp. 54-64). https://doi.org/10.1145/3522664.3528601

Rafidison, M. A., Ramafiarisona, H. M., Randriamitantsoa, P. A., Rafanantenana, S. H. J., Toky, F. M. R., Rakotondrazaka, L. P., & Rakotomihamina, A. H. (2023). Image classification based on light convolutional neural network using pulse couple neural network. *Computational Intelligence and Neuroscience*, *2023*(1), 7371907. https://doi.org/10.1155/2023/7371907

Rainio, O., Teuho, J., & Klén, R. (2024). Evaluation metrics and statistical tests for machine learning. *Scientific Reports*, *14*(1), 6086. https://doi.org/10.1038/s41598-024-56706-x

Rangineni, S. (2023). An analysis of data quality requirements for machine learning development pipelines frameworks. *International Journal of Computer Trends and Technology*, *71*(8), 16–27. https://doi.org/10.14445/22312803/ijctt-v71i8p103

Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., & Villalobos, P. (2022). *Compute trends across three eras of machine learning.* In *2022 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.* https://doi.org/10.1109/IJCNN55064.2022.9891914

Sharvit, Y. (2022). *Data-oriented programming: Reduce software complexity*. Manning.

TensorFlow Team. (2024, August 15). *Better performance with the tf.data API*. https://www.tensorflow.org/guide/data_performance

Wingqvist, D., Wickstrom, F., & Memeti, S. (2022). Evaluating the performance of object-oriented and data-oriented design with multi-threading in game development. *2022 IEEE Games, Entertainment, Media Conference, GEM 2022*. https://doi.org/10.1109/GEM56474.2022.10017610

Xin, D., Miao, H., Parameswaran, A., & Polyzotis, N. (2021). Production machine learning pipelines: Empirical analysis and optimization opportunities. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2639–2652. https://doi.org/10.1145/3448016.3457566

Yefta, C. (2022). Analysis study to detect student learning problems in online learning using text mining. *Journal of Theoretical and Applied Information Technology*, *31*(6).

Zhao, M., Adamiak, E., & Kozyrakis, C. (2025). cedar: Optimized and unified machine learning input data pipelines. *Proceedings of the VLDB Endowment*, *18*(2), 488–502. https://doi.org/10.14778/3705829.3705861