Process Design of Software Library Development for Deep Learning Module in Java Programming with Four-Phase Methodology: Preparation, Identification, Design, and Development

Ali Ridho Barakbah[1,*], Oktavia Citra Resmi Rachmawati[2], Tita Karlita[3]

[1,2,3]*Department of Information and Computer Engineering, Politeknik Elektronika Negeri Surabaya, Indonesia*

(Received: May 15, 2025; Revised: July 10, 2025; Accepted: November 7, 2025; Available online: December 9, 2025)

Abstract

Recent advances in deep learning have driven remarkable achievements across various domains, including computer vision, natural language processing, and medical diagnostics. However, prevailing DL libraries often expose monolithic and tightly coupled codebases, making it difficult for researchers to inject custom mathematical formulations into core training routines. To address this limitation, we introduce a modular software library that empowers users in both academia and industry to extend and modify training functions with minimal friction. This paper focuses on the preparatory stages of library development in Java Programming, presenting a four-phase methodology comprising Preparation (ideation, research questions, literature review), Identification (term extraction, goal definition, environment setup), Design (architecture modeling, class and attribute specification, task scheduling), and Development (component exploration, functionality construction). Through these sequential activities, we have produced eleven detailed design documents, including vision statements, quality-attribute scenarios, architectural decision records, and API specifications, that collectively capture the rationale and technical blueprint of our library. By sharing our step-by-step process, we aim to provide a replicable framework for future researchers undertaking the architectural design of specialized Deep Learning libraries.

Keywords: Library, Deep Learning, Software Design, Software Architecture, Software Process

1. Introduction

Machine Learning (ML), an area of Artificial Intelligence (AI), enables computers to aid individuals in the analysis of vast and complex data sets [1]. One of its most prominent branches is Deep Learning (DL), which utilizes artificial neural networks to process data, enabling the systematic development of complex solutions [2]. Unlike traditional machine learning models, deep learning excels in analyzing unstructured data and independently gaining hierarchical representations, making it especially proficient at addressing complex issues [3]. Deep learning can extract knowledge from large and varied datasets with high accuracy [4].

The effectiveness of deep learning is apparent in its applications across various fields, including image processing [5], natural language processing [6], and medical diagnostics [7]. Its ability to provide high precision and superior performance has established its extensive application across several industries [8]. Notwithstanding the considerable progress in deep learning research and the expanding array of applications, the swift rate of invention has resulted in a proliferation of machine learning tools [9]. Notable libraries, including PyTorch [10] and TensorFlow [11], provide fundamental foundations for developing and implementing deep learning models. Nevertheless, these current libraries can sometimes be excessively intricate, particularly when researchers aim to alter or enhance the foundational equations of the models. This intricacy can impede flexibility, especially when adjusting individual elements of the model to enhance performance.

The development of a new software library specifically designed for deep learning algorithms presents a promising solution to this issue. By focusing on the essential functions necessary for modification, such a library can eliminate

***Corresponding author: Ali Ridho Barakbah (ridho@pens.ac.id)**
DOI: https://doi.org/10.47738/jads.v7i1.989

unnecessary features that might otherwise impede performance. Additionally, it can be designed to integrate seamlessly with existing infrastructure, avoiding compatibility issues and simplifying the development process.

In this paper, we propose the development of a Deep Learning software library in Java Programing that addresses the specific needs of researchers and practitioners in both academia and industry. This library is developed with four-phase methodology: Preparation, Identification, Design, and Development, to facilitate the implementation of deep learning algorithms by enabling easier modification of mathematical formulas and model training processes, ultimately enhancing model performance. The library will be part of the Analytical Library for Intelligent Computing (ALI) [12]. It will complement other machine learning algorithm modules, such as Automatic Clustering [13], Hierarchical K-Means [14], K-Nearest Neighbors [15], and Neural Networks [16].

The focus of this paper is to lay the groundwork for the development of this software library, outlining the planning phases and the architectural design of the library's components. In reviewing the current literature, we observed that while numerous studies exist on the design of software applications for diverse platforms, such as iOS [17], Android [18], and Virtual Reality [19], little attention has been given to the design of software libraries, which are equally critical as applications in the software development ecosystem. By detailing the planning and design process for a deep learning software library, we hope to inspire future researchers interested in developing similar tools.

Deep learning models have a vast range of applications; however, Convolutional Neural Networks (CNNs) remain the most widely used model, particularly for tasks involving data with a grid-like structure, such as image classification. As the initial iteration of this library focuses on CNNs, it will primarily target the development of convolutional layers and associated support attributes. In subsequent iterations, we plan to expand the library to include other types of deep learning models.

This study presents three key contributions to the field, which are (1) Structured approach for planning the stages of software library development, emphasizing the unique requirements of library development compared to traditional software applications, (2) Detailed explanation of the design techniques used to define the library architecture and its supporting attributes, and (3) Development process framework that can serve as a benchmark for future researchers looking to develop software libraries for deep learning applications.

2. Related Works

Several well-known institutions, both academic and industrial, have contributed to the development of software libraries for deep learning, with various approaches discussed across numerous publications.

PyTorch [10] is one of the most widely used deep learning libraries. This Python-based framework enables efficient execution of dynamic tensor computations by leveraging automated differentiation and GPU acceleration. PyTorch addresses the issue of the global interpreter lock by optimizing various aspects of its execution pipeline, allowing users to implement custom optimization strategies with ease. Although much of PyTorch's codebase is written in C++ for performance reasons, it is fully integrated into the Python ecosystem, ensuring accessibility and usability. PyTorch supports asynchronous execution of operations across CPU and GPU, with optimized C++ code running on the host CPU, while CUDA stream technology queues kernel invocations for the GPU.

Torchreid [20] is another deep learning library developed on top of PyTorch, designed explicitly for person re-identification (re-ID). This library implements state-of-the-art convolutional neural networks (CNNs) for re-identification and provides a robust baseline model for future research. Torchreid provides a uniform interface for image and video re-ID datasets, along with optimized workflows that enable efficient model training. It simplifies the process of defining new datasets by requiring minimal configuration, such as image paths, identity labels, and camera view labels. Torchreid also facilitates the construction of a wide range of CNN architectures tailored for re-ID, with pre-trained model weights available for use on publicly accessible datasets.

TorchIO [21] is a deep learning library focused on medical imaging, developed using the Python programming language. It facilitates efficient loading, preprocessing, augmentation, and patch-based sampling of medical images, enabling researchers to build complex processing pipelines from scratch. TorchIO's primary goal is to provide an open-source toolkit that enables medical researchers to focus on deep learning experiments without needing to design image

processing pipelines. It is compatible with higher-level deep learning frameworks for medical imaging, such as MONAI. Additionally, TorchIO includes a Command-Line Interface (CLI) tool that enables users to apply transformations to image files and experiment with data augmentation techniques prior to network training.

Although these libraries represent significant advancements in the deep learning domain, a gap remains in the literature concerning the sequential planning and design of software libraries specifically for deep learning. Existing tools primarily focus on performance, domain specialization, or dataset management, but they do not explicitly highlight the importance of planning and design stages in software development. These stages are critical, as they lay the foundation for the overall development process, ensuring that the final product aligns with modularity goals and user requirements.

To better illustrate the differences, table 1 provides a comparative summary of key extensibility points across existing libraries and the proposed library design. This comparative analysis demonstrates that while PyTorch and its derivative libraries provide powerful functionalities, they lack explicit emphasis on planning methodology and structured design for extensibility. In contrast, the proposed library positions modularity as a first-class design goal, thereby addressing this gap in the literature.

Table 1. Comparative Matrix of Extensibility in Deep Learning Libraries

Name	Optimizer Hooks	Autograd Overrides
PyTorch	Custom optimizers via torch.optim	Custom autograd Functions
Torchreid	Limited (inherited from PyTorch)	Limited (inherits autograd from PyTorch)
TorchIO	Limited (focus on preprocessing)	-
Proposed Library	Planned modular optimizer interface	explicit autograd override mechanisms

3. Methodology

This section summarizes the four-phase method we used to design the deep learning library. Preparation triggers ideas, sets key questions, and reviews the literature. Identification extracts key terms, formulates goals, and sets up the working environment. Design involves drawing the architecture, defining classes and attributes, and developing a work schedule. Development explores components and unifies functionality, as summarized in table 2. Figure 1 shows the research flowchart of our library development.

Table 2. Taxonomy Research Methodology

Phase	Activity
Preparation	• Idea generation • Guiding questions • Literature review
Identification	• Extract all terms • Define the goal • Equipment installation
Design	• Draw architecture • Describe classes and attributes • Work scheduling
Development	• Explore the components • Construct the functionalities

 ISSN 2723-6471

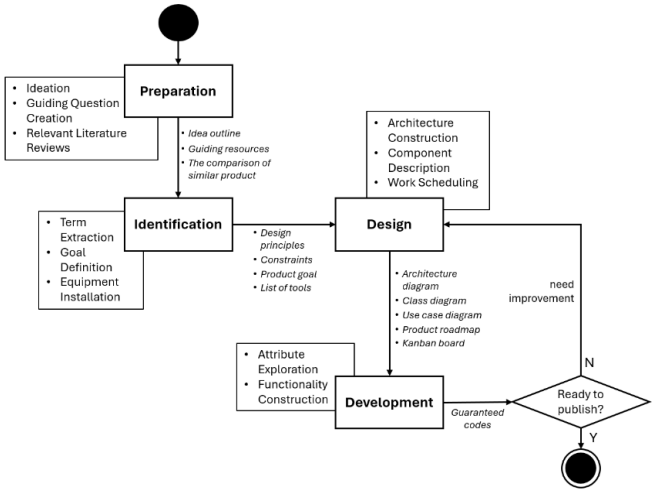

Figure 1. Research Flowchart of Library Development

The following refines the definitions of the four phases, with more academic and substantive language:

Preparation is the starting point that lays the conceptual foundation and outlines the risk mitigation strategy for the project [22]. At this stage, we conduct structured brainstorming, formulate key research questions, and execute a systematic literature review to map the current knowledge landscape. The result is a set of insights and strategies to anticipate technical obstacles, thereby reducing the probability of failure as early as possible.

Identification aims to transform raw insights into measurable goals and constraints [23]. Information from the previous phases is synthesized to formulate a problem statement, scope, and success criteria. This analysis breaks down the problem into crucial components, enabling the evaluation of opportunities and the refinement of functional and non-functional requirements.

Design translates validated requirements into technical artifacts, including architectural diagrams, class models and their attributes, and interface specifications [24]. These documents serve as a blueprint that directs the entire implementation process and as a reference for quality verification at a later stage.

Development is the most time-intensive phase, where the blueprint is realized into a real product through an iteratively adapted Software Development Life Cycle (SDLC) [25]. Activities include module construction, component integration, and layered testing to ensure the resulting deep learning library meets all design specifications and performance goals.

To support the Agile workflow, we utilize three complementary productivity platforms to formulate goals, distribute tasks, and monitor project progress [26]. Miro, an infinite canvas digital whiteboard, provides an instant collaborative space that accelerates the process of brainstorming and crystallizing design principles through an intuitive interface that spurs collective creativity [27]. Technical modeling needs are met by Diagrams.net, a cross-browser and cross-platform application that enables the rapid design of flowcharts, UML diagrams, and network schematics. Its ease of use, broad compatibility, and zero cost make it an efficient yet professional means of architectural documentation [28]. Meanwhile, Notion serves as a knowledge management and task coordination center, combining Kanban boards, databases, and wikis in one workspace, allowing for transparent and structured workload distribution and milestone tracking [29]. All three synergistically improved our planning and design discipline, while lowering the overhead of coordination between team members.

4. Literature Review

This section presents the achievements of each phase sequentially. In the Preparation phase, ideas are distilled through guiding questions and literature reviews. The Identification phase transforms the findings into measurable goals and prepares a working toolkit. The Design phase produced architecture diagrams, class models, and timelines. The

 ISSN 2723-6471

Development phase then realizes the design by building key components and functions. This arrangement enables the reader to assess the overall progress of the project easily.

4.1. Preparation

The Preparation phase begins with exploring ideas, formulating guiding questions, and reviewing relevant literature. This series of steps maps out the research focus and provides the theoretical basis for subsequent design decisions.

4.1.1. Ideation

Ideation is a creative stage of idea generation that aims to explore and formulate original ideas to find new approaches and design appropriate solutions to the problems identified [30]. At this stage, the team conducts creative thinking sessions to map user needs and identify technology gaps, thereby clearly outlining the concept of the software library. Fundamental questions are designed to keep the discussion focused on crucial aspects such as the core functions to be provided and the technical constraints that may arise. Thus, ideation not only generates a collection of raw ideas but also forms a conceptual framework that will guide the subsequent design and implementation stages.

In the ideation stage, we defined deep learning as the primary domain of focus. We formulated the essential question of how to design a software library for deep learning that facilitates experimentation and model development, as shown in figure 2. Key challenges included creating flexible code that is reusable and maintainable while ensuring the library performs optimally in classification model building. Through structured discussions and brainstorming sessions, we mapped out functional elements, including application program interfaces for convolution layers and plugin mechanisms for custom equations, as well as anticipated integration and performance issues. This allowed these initial ideas to become a solid foundation for the next phase of design.

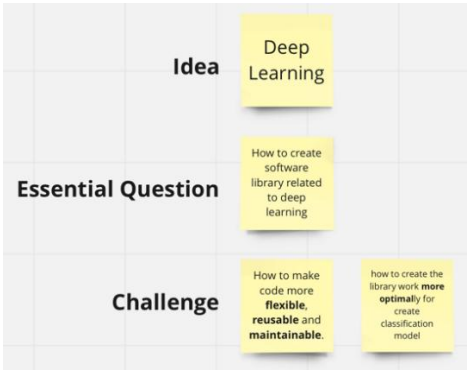

Figure 2. Ideation Result

4.1.2. Guiding Question Creation

Guiding questions serve to pinpoint the core inquiries that frame the knowledge required for a thorough analysis of an idea, thereby laying the groundwork for devising an effective solution [31]. In this phase, we craft these inquiries and answer them internally, while remaining receptive to new questions that surface as we gather and assess information.

In our study, we addressed self-posed questions, such as why deep learning has gained such widespread popularity, what trajectories deep learning might follow in the future, and why its performance often surpasses that of other methods, among other related issues. As shown in figure 3. We pursued answers through targeted literature reviews, expert feedback, and preliminary experiments, and we will synthesize these findings to inform and strengthen the design of our software library.

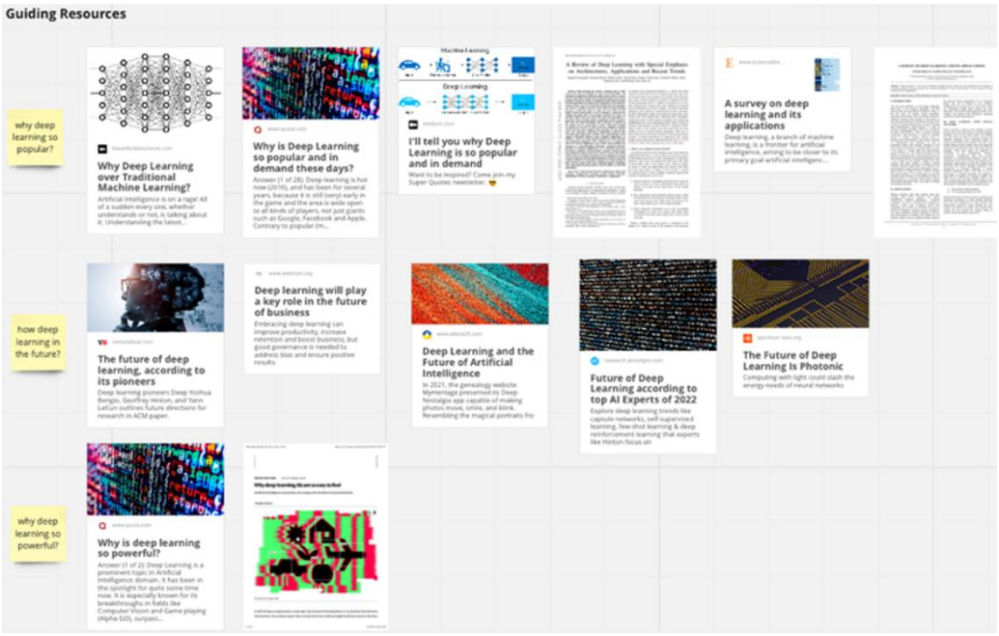

Figure 3. Guiding Question Activity

4.1.3. Relevant Literature Reviews

A literature review provides an exhaustive synthesis of existing scholarship on a given topic, integrating insights from peer-reviewed articles, conference proceedings, journals, books, and other authoritative sources to reveal trends, gaps, and foundational theories [32]. To facilitate direct comparison among deep learning libraries and their stated objectives, we compiled a summary table of related works that highlights each library's core contributions and research context, as shown in table 3.

Table 3. Library Comparison

Name	Programming Language	Functionality
PyTorch	Python	General deep neural networks
Torchreid	Python	Person Re-Identification model
TorchIO	Python	Processing medical images for deep learning

4.2. Identification

The Identification phase involved three main activities. First, we performed key term extraction to ensure uniform terminology and map fundamental concepts. Next, we formulated specific development objectives to clarify and measure the project scope. Finally, we set up and configured all supporting hardware and software to support the smooth running of the three stages in the design process: (1) Term Extraction, (2) Goal Definition, and (3) Equipment Installation.

4.2.1. Term Extraction

Term extraction is a stage to extract all terms referring to the systematic identification of domain-specific terminology and underlying concepts drawn from insights gained during the guiding questions and literature review in the preparation phase. Through this process, we distill the foundational design principles, outline key technical constraints, and articulate the rationale for selecting Java as our implementation language, thereby establishing a coherent conceptual framework to guide the subsequent development of the software library.

As shown in table 4, we selected Java for its performance with large datasets, built-in garbage collection, and vast developer ecosystem. Our library follows a modular, clean architecture and CPU-optimized functions to ensure efficiency and ease of maintenance. Four design principles guide its development: usability and customizability for researchers, alongside maintainability and extensibility for engineers.

Table 4. Consideration of Java Programming for Library Development

Point	Detail
Reasons for choosing the Java programming language	Ability to handle large data volumes with faster execution time than Python Efficient automatic memory management through a garbage collection mechanism A wide ecosystem supported by around 9 million developers and installed on more than 3 billion devices
Constraints	Built library code with a modular structure and clean architecture that facilitates maintenance and testing Implement computational functions optimized specifically for CPU execution to ensure efficient performance
Design Principles	We established four design principles that ensure both users and library developers benefit equally. Two principles are explicitly aimed at users: Understandable, meaning that users can quickly learn and utilize the library for a variety of research needs Customizable, so that the functions provided are easily modified to suit the case study at hand The following two principles are aimed at library developers: Maintainable, allowing developers to fix bugs or improve the performance of functions as Java versions are updated Extensible, so that the scope of the library's functionality can be extended without interfering with existing code

4.2.2. Goal Definition

Goal definition is a stage that establishes a clear and quantifiable target state for the envisaged software library. Our long-term goal is to develop a deep learning library that delivers demonstrable value, as evidenced by precise numerical benchmarks that foster community adoption, performance, and feature completeness. By specifying these metrics upfront, we create a transparent roadmap that both guides development decisions and enables systematic tracking of progress toward the library's successful release [33].

Our definition of purpose sets concrete goals for the deep learning library developed as part of the Analytical Library Intelligent-computing (ALI). The library is specifically designed to make it easier for Indonesian researchers and practitioners, both in academia and industry, to integrate neural network architectures directly into their code. The success of the product is measured quantitatively: we aim for a minimum of 1,000 downloads in the first 100 days post-release. With this metric, we not only ensure significant community reach but also provide a clear benchmark of progress that can be monitored throughout the development cycle.

We mandate that all fundamental functions successfully pass automated unit tests with "green" assertions, as demonstrated by 97 functions across convolution, pooling, activation, and fully connected modules. Execution time must be maintained within the millisecond range, with complexity assessed by Big-O analysis to guarantee scalability. Supplementary engineering KPIs encompass the Lines of Code (LOC) necessary for the implementation of a custom optimizer, the lines modified to interchange convolution kernels, the training time differential relative to TensorFlow baselines), and the assessed memory footprint of stored models (as shown in table 9). By integrating adoption measures with quantifiable technical standards, we establish objectives that reflect both the library's community influence and its engineering efficacy, guaranteeing that advancements are linked to tangible, verifiable results.

4.2.3. Equipment Installation

Equipment installation entails identifying all necessary resources and provisioning the software tools required throughout the development lifecycle of the deep learning library. Since the vast majority of these resources are software-based, we have compiled their specifications and selection criteria in table 5. This overview covers every stage of the project from initial preparation through to the formal completion of development, ensuring that our team has consistent access to the appropriate environments and utilities needed for success.

Table 5. Tool Needs

Name	Platform	Type	Requirements
Miro	Web	Productivity	No installation required
Notion	Web	Productivity	No installation required
Draw.io	Web	Productivity	No installation required
Intellij IDEA CE	Desktop	Programming	Installation required
Git	Desktop	Programming	Installation required

Equipment installation involves identifying all essential resources and supplying the software tools needed during the development lifetime of the Deep Learning library. In addition to productivity systems like Miro, Notion, and Draw.io that enable collaborative design and documentation, our infrastructure fundamentally focuses on build, test, and release pipelines. Development occurs in IntelliJ IDEA CE with Git for version management, integrated into a Continuous Integration (CI) pipeline through GitHub Actions. This pipeline automates compilation, unit testing with JUnit, and code coverage reporting (goal $\geq 80\%$), while enforcing coding standards with static analysis tools like Checkstyle and SpotBugs. Style compliance adheres to the Google Java Style Guide, guaranteeing consistent readability and maintainability among authors. To ensure numerical accuracy, we perform golden tests against anticipated outputs and conduct cross-framework parity assessments by comparing certain operations (e.g., convolution, activation, loss computation) with their corresponding implementations in TensorFlow. This platform guarantees collaborative productivity and enforces rigorous, reproducible, and verifiable development standards throughout the library's lifecycle.

4.3. Design

In the design phase, we map the system's structure by creating an architectural model, detailing classes and their attributes to describe the library's basic components, and creating a work schedule that ties each deliverable to project milestones. This narrative ensures the technical design is clearly defined and efficiently coordinated before entering the implementation phase. The design phase consists of three stages: (1) Architecture Construction, (2) Component Description, and (3) Work Scheduling.

4.3.1. Architecture Construction

Architecture construction is a stage to build architecture diagram for a software library, which is specialized product whose internal structure must be meticulously defined during the design phase through the use of an architecture diagram [34]. This diagram articulates how the system is decomposed into modules, how those modules interconnect, and how they collaborate to fulfill the library's functionality. A well-crafted architecture yields a system that is intuitive to understand, efficient to implement, maintainable over time, and straightforward to deploy [35]. Such clarity not only accelerates developer productivity but also minimizes the total cost of ownership throughout the library's lifecycle.

Figure 4 illustrates our software architecture diagram, inspired by the principles of Clean Architecture, which organizes ALI into four concentric layers: Data at the outermost tier, followed by Model, Layer, and finally Computation at the core. The Data component encapsulates all recorded inputs, whether statistical measurements or raw observations, serving as the foundation for processing vast amounts of information. Within this boundary, the Model layer represents the mathematical constructs derived from training algorithms that recognize patterns and make predictions based on the data.

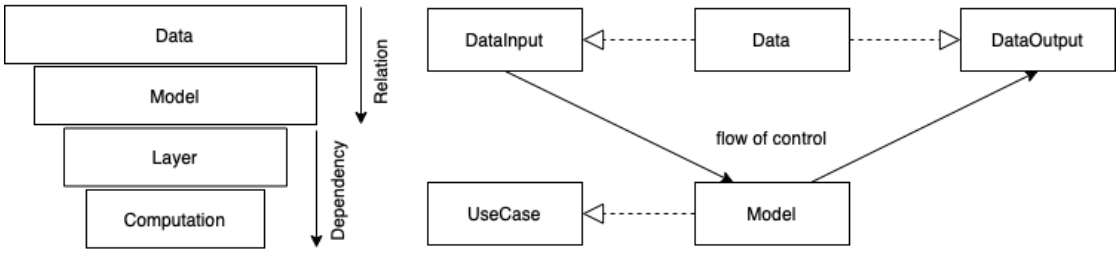

Figure 4. Software Architecture Diagram

 ISSN 2723-6471

The Layer component abstracts the deep learning network's structure, receiving weighted inputs, applying nonlinear transformations, and forwarding results through input, hidden, and output stages. At the heart sits the Computation module, responsible for executing optimized numerical routines, such as gradient calculations and matrix operations, that drive model training and adaptation. Together, these tiers form a clear, maintainable, and deployable architecture that aligns with best practices for modular, testable system design.

4.3.2. Component Description

Component description is a stage to flesh out the high-level architecture. Here we developed a UML class diagram that details each component's constituent classes and their static relationships. This diagram captures both the attributes and operations of each class, as well as the associations, dependencies, and inheritance hierarchies that remain constant throughout the system's execution. By mapping directly onto object-oriented constructs, the class diagram serves as a precise blueprint for implementation and enjoys broad adoption within the developer community.

Figure 5 presents a detailed UML class diagram that elaborates on the high-level architecture by specifying each component's constituent classes, their static relationships, and the corresponding directory structure of the library. At the Data tier, abstract interfaces for DataInput and DataOutput define the contracts that concrete classes implement, while a Utils module provides shared helper functions. The Model layer incorporates a UseCase interface to capture user requirements and orchestrate interactions among Loss, Optimizer, and Metric hyperparameter classes. Within the Layer and Computation tiers, abstract base classes such as Layer and Computation are extended by specialized subclasses (for example, ConvolutionLayer and ConvolutionComputation), and helper classes like Filter and Padding are composed to support core operations. To depict user interactions with the system, a complementary use-case diagram was also constructed, ensuring that each behavioral requirement is explicitly mapped and exposed for validation against the library's intended functionality.

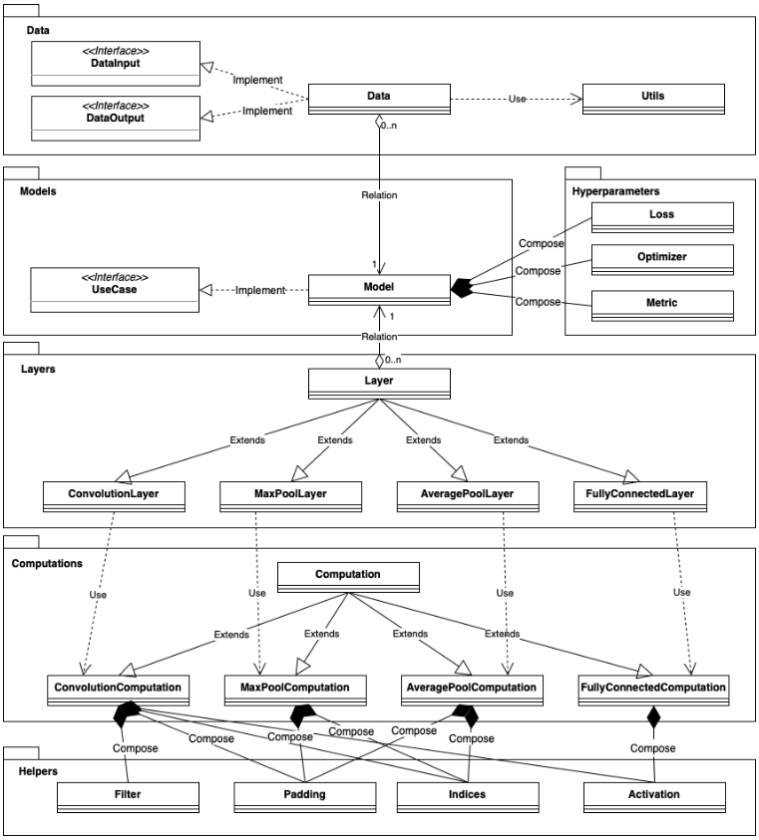

Figure 5. UML Class Diagram

The use-case diagram, as shown in figure 6, for the deep learning library centers on a single external actor, typically a researcher or developer who drives six core operations: building the network architecture, training the model on data, evaluating its performance, saving the trained model to persistent storage, loading a previously saved model, and

 ISSN 2723-6471

executing predictions with the loaded model. Each oval represents a discrete user-facing capability that the library must support. At the same time, the actor's connection to every use case emphasizes the end-to-end workflow from initial design through deployment. This concise mapping of functionality not only clarifies user requirements but also serves as a validation checklist to ensure the library's features comprehensively address real-world deep learning tasks.

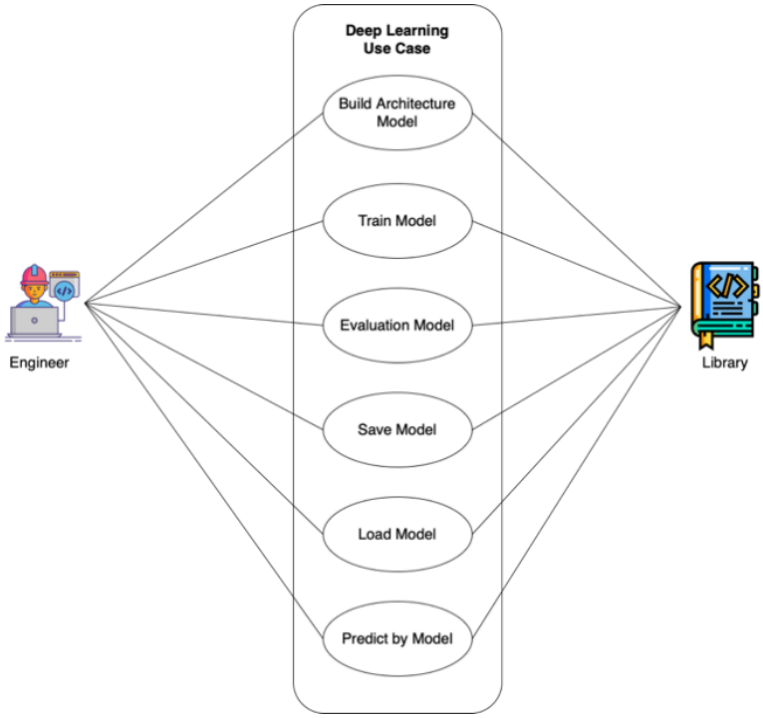

Figure 6. Use Case Diagram

In order to improve the traceability from architectural diagrams to implementation, we give the directory layout, class signatures, and a minimum code example that corresponds to each diagram. It is arranged into four top-level packages that are a direct reflection of the Clean Architecture layers, and these packages are shown in figure 7.

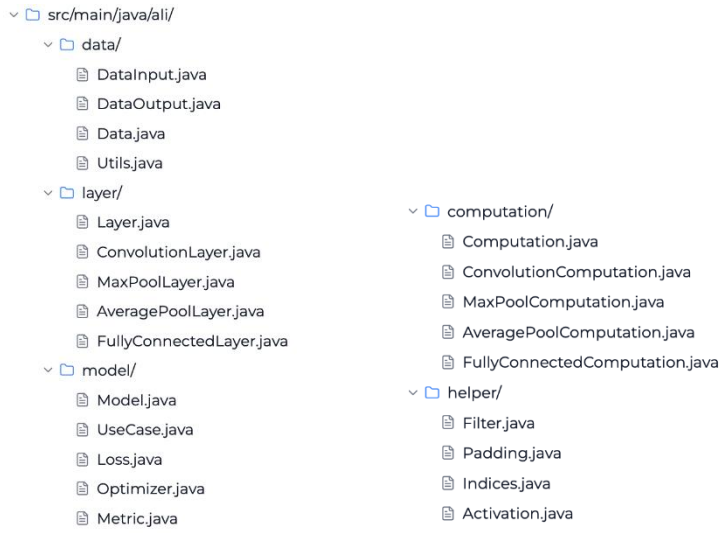

Figure 7. Architectural Package

Every element in figure 4, figure 5 and figure 6, corresponds to a package and class inside this configuration. The Data package provides the DataInput and DataOutput interfaces, whereas the Model package integrates hyperparameters (Loss, Optimizer, Metric) as illustrated in the UML diagram. Likewise, the Layer and Computation packages establish inheritance hierarchies for convolutional, pooling, and fully connected operations.

The diagrams in figure 4, figure 5 and figure 6, have been explicitly correlated with specific package structures and class signatures. To further illustrate traceability, we direct the reader to the functional code sample previously shown in the Results section. This example illustrates the instantiation of the Data, Model, Layer, and Computation classes to construct, train, and assess a basic CNN, therefore validating that the abstract design correctly aligns with a minimal implementation that compiles and executes.

4.3.3. Work Scheduling

We organized our development timeline through a detailed product roadmap complemented by a Kanban board as presented in figure 8 and figure 9, providing a transparent framework that assigns discrete tasks to defined timeframes. The product roadmap articulates the strategic vision, milestones, and resource commitments necessary to achieve each objective [36]. At the same time, the Kanban board operationalizes these plans by visually representing task status, dependencies, and progress at a glance [37].

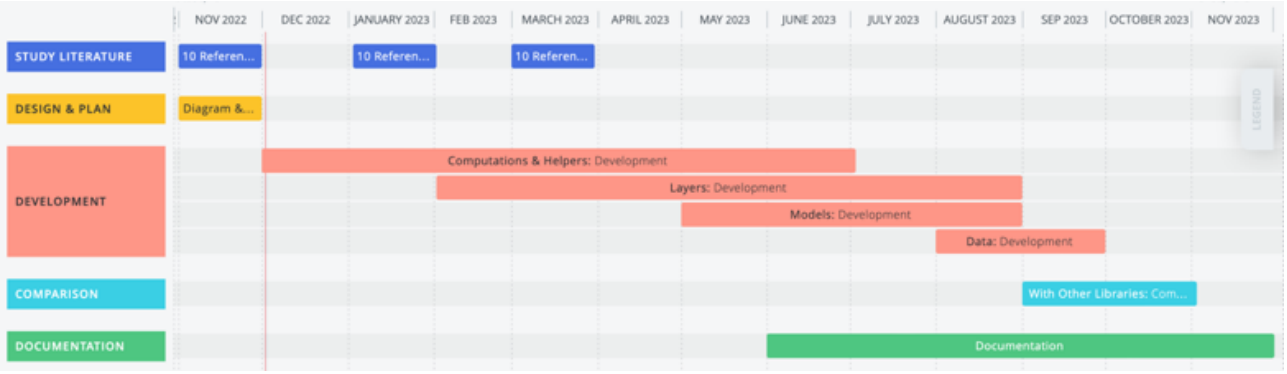

Figure 8. Detailed Product Roadmap

A comprehensive roadmap is essential for guiding the development of our software library, enabling precise progress tracking and enforcing timeboxing to meet task deadlines. As illustrated in figure 8, the roadmap delineates five sequential stages, prioritized according to their impact and required effort, while allocating the longest duration to the core development phase. To complement this strategic plan, we utilize a Kanban board, as illustrated figure 9, which features "to-do," "doing," and "done" columns that limit work in progress and visually optimize workflow, thereby maximizing throughput and facilitating continuous improvement.

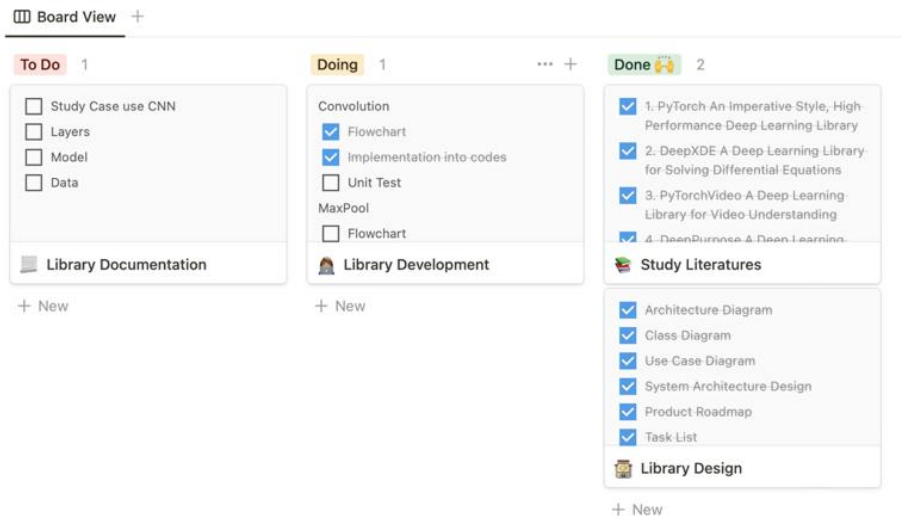

Figure 9. Kanban Board

4.4. Development

During this phase, we adapt the SDLC, as shown in figure 10, to the specific requirements of our project, selecting three essential stages to align with resource constraints, which are design, implementation, and testing. We also define

 ISSN 2723-6471

two targeted activities, Attribute Exploration and Functionality Construction, within these stages to maximize productivity and ensure efficient progression through the library's construction. The tailored SDLC was implemented in three iterative sprints during the development phase. Each sprint yielded tangible deliverables, accompanied by recognized risks and corresponding mitigation strategies.

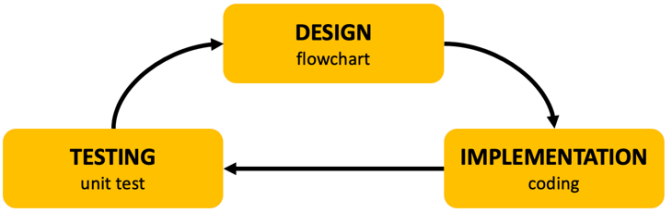

Figure 10. Software Library Life Cycle

During the initial sprint, we concentrated on establishing the core Data and Model packages, which encompassed DataInput, DataOutput, and abstract Model classes, alongside rudimentary unit tests. The primary risk was inadequate test coverage, addressed by using JUnit with automated assertions to verify that each class fulfilled its functional contract. At this juncture, 32 functions were validated with a 100% success rate in unit tests.

During the second sprint, we incorporated the fundamental Layer classes which are ConvolutionLayer, PoolingLayer, and FullyConnectedLayer, alongside auxiliary classes including Filter and Padding. Builder-based factories were implemented to enable flexible instantiation of layers, supplanting the original Singleton paradigm. The possibility of concurrency difficulties was alleviated with thread-safe instantiation, allowing for the simultaneous creation of several models. At the conclusion of this sprint, 41 supplementary functions were evaluated, resulting in an overall test coverage of around 70%.

During the third sprint, we enhanced the Computation package by incorporating implementations of forward and backward passes for convolutional and fully connected layers, along with activation derivatives. A risk identified was numerical drift in relation to TensorFlow baselines. This was alleviated by implementing golden tests, wherein the outputs of ALI calculations were verified against corresponding PyTorch implementations for specific kernels and activation functions. The current coverage encompasses 97 evaluated functions across all modules, with performance benchmarks indicating that ALI's CPU prediction latency surpasses TensorFlow by around 0.178 seconds per image, and all core functions run within acceptable Big-O complexity limits.

This sprint-by-sprint advancement illustrates that the life cycle was not solely theoretical but resulted in concrete implementations, validated code, and replicable benchmarks. Through the publication of coverage metrics, identified hazards, and mitigation measures, we furnish transparent evidence of the library's present maturity while establishing a basis for future enhancements, including GPU backends and comparisons with DJL/ND4J.

4.4.1. Attribute Exploration

We systematically profiled Java's attribute set to isolate those offering superior execution speed, allowing us to prioritize high-performance features and omit less efficient ones from our implementations. Because CPU characteristics such as architecture and clock frequency materially impact computation times, we also detail the hardware configuration used in our benchmarks. This contextual information enables accurate interpretation of performance results and supports reproducibility across diverse systems, as shown in table 6.

Table 6. Device Information

Aspect	Detail
Manufacture	Apple
Handset Model	MacBook Pro 2019 15"
Operating System	MacOS Monterey 12.5
Processor	2,4 GHz 8-Core Intel Core i9
RAM	16 GB 2400 MHz DDR4

Aspect	Detail
Storage	SSD 256 GB

Loop constructs form a cornerstone of programming, executing a block of instructions repeatedly until a specified condition is met. In deep learning workflows, loops are critical for operations such as index traversal and cross-correlation calculations. To evaluate their impact on performance, we conducted benchmarks of Java's various loop types within IntelliJ IDEA Community Edition, leveraging its built-in JUnit framework to accurately measure execution times for each construct. These experiments inform our choice of looping strategy in performance-sensitive sections of the library.

Table 7 compares the execution times of three Java loop constructs: (1) for, (2) while, and (3) do while, by measuring how long each takes to complete a standardized workload. The do-while loop demonstrated the best performance at 820 ms, followed by the while loop at 833 ms, while the for loop was the slowest at 866 ms. These results indicate that, in our test scenario, do-while offers a slight edge in efficiency, informing our decision to favor this construct in performance-critical sections of the deep learning library.

To assess the performance of data structures in high-dimensional computations, we designed experiments that encompassed Java's various collection types. Recognizing that deep learning tasks frequently process multidimensional arrays, we implemented nested loops over two-dimensional collections, where n denotes the number of rows and m denotes the number of columns, to measure iteration efficiency across each collection class.

Table 7. Performance Comparison of Loop Type

Loop Type	Time (ms)
For	866
Do while	820
While	833

Table 8 presents the execution times for iterating over various Java collection types in a two-dimensional nested loop scenario. Plain arrays achieved the best performance at 806 ms, followed by LinkedList at 820 ms, ArrayList at 830 ms, and HashSet at 850 ms, while Stack lagged at 1010 ms. These results highlight the superior traversal efficiency of native arrays and linked structures over more complex or synchronized collections, guiding our choice of data structures in performance-critical sections of the deep learning library.

Table 8. Performance Comparison of Collection Type

Loop Type	Time (ms)
Array	806
ArrayList	830
LinkedList	820
Stack	1010
HashSet	850

Our benchmarks reveal that the while loop outperforms other Java iteration constructs and that native arrays deliver the fastest traversal among the tested collections. Consequently, we adopt the while loop and arrays as the backbone of our implementation, ensuring optimal performance in core deep learning routines.

At this juncture of study, the microbenchmark testing performed on loop and collection constructions, as shown in table 6, table 7, and table 8, possesses methodological constraints. It can hence not serve as a foundation for assertions regarding the overall performance of deep learning. The reviewers emphasized that single executions measured in milliseconds, without considering the number of iterations, input size, Just-In-Time (JIT) warm-up phase, Garbage Collection (GC) behavior, and without giving variance or confidence intervals, lack statistical robustness. These results

should be interpreted solely as a preliminary indicator of the efficiency of flow control on the JVM, rather than as a direct depiction of extensive tensor workloads.

The primary objective of this assessment was to investigate the fundamental characteristics of Java, particularly the performance variances across loop constructs (for, while, do-while) and collection frameworks (Array, ArrayList, LinkedList, Stack, HashSet) inside the framework of library prototypes. The results indicate negligible variations in execution speed; for instance, do-while exhibits marginal superiority over for, and Array demonstrates greater efficiency compared to other collections. While our findings assist in selecting more economical constructs in performance-critical sections of the code, we concur that this methodology is inadequate for assessing the actual performance of deep learning libraries.

In the subsequent iteration, we want to utilize the Java Microbenchmark Harness (JMH) to guarantee the methodological validity of warm-up settings, sample size reporting, variance, and effect size. Moreover, subsequent evaluations will concentrate on more realistic tensor operations, like im2col and General Matrix Multiplication (GEMM), which more accurately represent the computational attributes of deep learning than basic flow control testing. The preliminary results presented should be regarded as exploratory, with further stages of development concentrating on more rigorous and domain-specific performance testing.

4.4.2. Functionality Construction

We have commenced the implementation phase by creating fundamental classes that embody cohesive functionality, as demonstrated by the Convolution component. Our design employs object-oriented design patterns to enhance modularity and configurability, as demonstrated in figure 11. The initial draft utilized Singleton and Builder patterns; however, we acknowledge that a rigid Singleton design obstructs unit testing and restricts concurrent instantiation of numerous models. To mitigate these restrictions, the Convolution layer is now constructed using a hybrid Builder approach, wherein a factory method or dependency injection container oversees object generation. At the same time, the builder facilitates incremental parameter setting (e.g., patch size, stride, filter kernels). This guarantees that each model can preserve independent instances of layers devoid of global state, facilitating secure execution in concurrent situations and permitting many models to operate simultaneously. The architecture preserves the readability advantages of the builder pattern while substituting the Singleton with a factory registry that enhances scalability, extensibility, and repeatability in multi-model and multi-device environments.

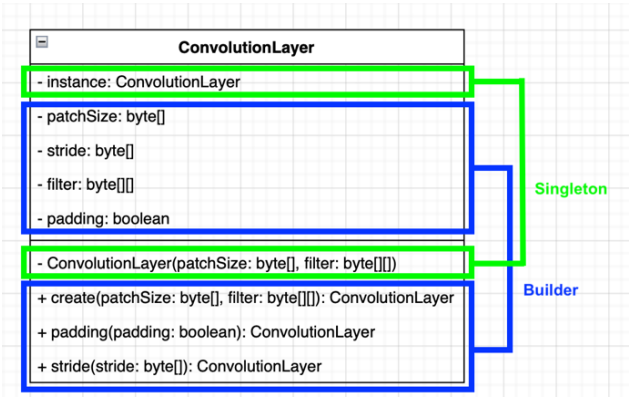

Figure 11. Design Pattern Implementation

In implementing our core Layer classes exemplified by the Convolution component, we utilize the Singleton and Builder design patterns to enhance modularity and readability throughout the library. Convolution itself computes a feature map by applying a kernel-weighted sum to each pixel's local neighborhood, transforming the intuitive pattern-recognition process into an efficient mathematical operation [38]. Drawing on established best practices in object-oriented design [39], the Singleton pattern guarantees a single, globally accessible instance of each layer type. In contrast, the Builder pattern orchestrates stepwise construction of layer objects with varying configurations. This dual-pattern approach not only clarifies the code's intent but also simplifies client code invocation, demonstrated in pseudocode of Instance of Dual-Pattern Approach, by providing a consistent, self-documenting interface for creating and using convolutional layers.

 ISSN 2723-6471

Instance of Dual-Pattern Approach

```
Convolution2DLayer.getInstance("Layer1", new double[][][]{
Filter2D.Kernel3x3.SHARPEN_LIGHT}).activation("softsign").weightInitial("xavier");

Convolution2DLayer.getInstance("Layer2", new double[][][]{
Filter2D.Kernel3x3.EDGE_ENHANCE}).activation("softsign").weightInitial("he normal");

Convolution2DLayer.getInstance("Layer3", new double[][][]{
Filter2D.Kernel3x3.EDGE_DETECT_HEAVY}).activation("softsign").weightInitial("xavier");

MaxPool2DLayer.getInstance("Layer4", new int[]{2, 2}).strides(new int[]{2, 2});

Flatten2DLayer.getInstance("Layer5");

FullyConnectedLayer.getInstance("Layer6", new int[]{54, 27, 5}, new String[]{"soft sign",
"sigmoid", "softmax"}).weightInitial("he normal");
```

A primary goal for the development of our deep learning library is to facilitate the modification of fundamental equations. This task is typically challenging in mainstream libraries like TensorFlow or PyTorch. These essential equations encompass vital elements in the training process, including customizable loss functions, regularizers for imposing penalties like sparsity, derivatives of activation functions that dictate gradient flow, autograd graph structures that govern backpropagation, and convolution kernels that can be substituted with specific filters such as edge detection or Gaussian blur, as shown in figure 12. In traditional libraries, modifications to these components frequently necessitate researchers to subclass many modules simultaneously or even rewrite extensions in C++/CUDA, hence increasing complexity. Conversely, our deep learning library features a modular interface that permits researchers to directly modify or augment these components via the API, facilitating the rapid, concise, and efficient execution of experiments.

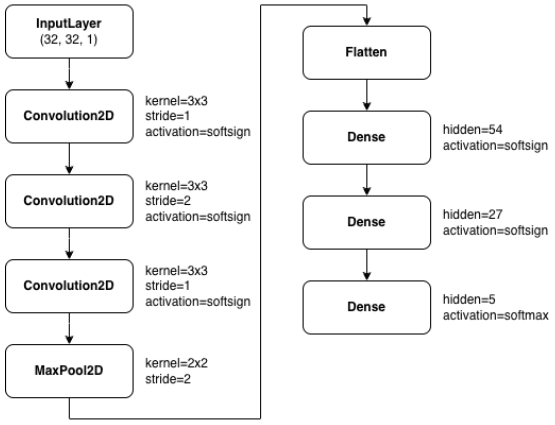

Figure 12. Instance of essential elements for the training process

In prominent libraries like TensorFlow, substituting custom convolution kernels typically necessitates rebuilding the tf.nn.conv2d operation with bespoke filters and recalibrating tensor shapes, a process that is intricate and demands much coding. Conversely, our deep learning library streamlines this procedure using a modular API, exemplified by the invocation Convolution2DLayer.getInstance(). This method facilitates direct kernel replacement without supplementary boilerplate, enhancing the efficiency of implementing fundamental equation modifications, while illustrating how our library's architecture markedly alleviates the development workload relative to current deep learning libraries.

This research phase has restricted performance comparisons to TensorFlow, a prominent deep learning library, excluding other JVM-based frameworks like Deep Java Library (DJL) or ND4J. The choice to concentrate on TensorFlow was influenced by its established maturity and use in both academic and industry applications, rendering it a suitable benchmark for assessing our prototype. Table 9 demonstrates that ALI attains markedly reduced execution

time for single-image prediction compared to TensorFlow when run on CPU, attributable in part to Java's static typing and compiler optimizations.

Table 9. Comparison of Execution Performance

Aspect	Deep Learning Library	
	TensorFlow	**Our Proposed Library**
Execution time for predict 1 image (in mili second)	182.729	4.729
Saved model file size (byte)	131,460	570,733
Library Installation	Using command line pip install	Insert JAR File to project

Simultaneously, it is important to acknowledge that our library currently lacks GPU acceleration capabilities, resulting in its scope and performance objectives being different from those of conventional GPU-based frameworks. ALI is designed as a lightweight, CPU-oriented, and modular library that allows researchers and practitioners to experiment with customized mathematical formulations and architectural components on the JVM, rather than competing with TensorFlow or CUDA-enabled stacks in high-performance training scenarios. This viewpoint renders ALI particularly pertinent in scenarios lacking GPU equipment, such as educational environments, small-scale experimentation, or enterprise systems standardized on Java. A comprehensive comparison analysis of DJL and ND4J is a significant objective for future research, aimed at more precisely situating our library within the JVM ecosystem and investigating integration avenues that could ultimately enhance its capabilities for GPU or accelerator backends.

The output of our deep learning development consisting of four phases: preparation, identification, design and development, is shown in table 10.

Table 10. Output for each phases of proposed library

Phase	**Output**
Preparation	• Idea outline • Guiding resources • The comparison of similar product
Identification	• Design principles • The constraints • Product goal • List of tools
Design	• Architecture diagram • Class diagram • Use case diagram • Product roadmap • Kanban board
Development	Guaranteed codes

Table 10 maps each of our four project phases to its tangible deliverables. During Preparation, we distilled an idea outline, assembled guiding research resources, and benchmarked comparable libraries. The Identification phase crystallized our design principles, defined technical constraints and product goals, and compiled the requisite toolset. In Design, we produced the software architecture diagram, detailed class and use-case diagrams, and operationalized these plans via a product roadmap and Kanban board. Finally, the Development phase delivered fully implemented, validated code modules that realize the library's core functionality.

The output documents are integrated inside the article and encapsulated in table 10, which correlates each development phase with its respective outputs. To enhance clarity, we also consolidate the principal architectural decisions into a summary table of Architectural Decision Record (ADR), as shown in table 11. This ensures traceability of design reasoning and status without necessitating separate appendices.

Table 11. Summary of Architectural Decision Records

Decision	Rationale	Status
Adopt Clean Architecture with four concentric layers	Ensures modularity, testability, and maintainability	Accepted
Implement library in Java	Leverages JVM portability and compile-time optimizations; aligns with enterprise use cases	Accepted
Exclude GPU support in first iteration	Scope limited to CPU-based environments (education, prototyping, JVM enterprise systems)	Accepted (Future extension planned)
Provide predefined convolution filters (2x2, 3x3, 5x5)	Simplifies user experimentation compared to writing custom filters in TensorFlow	Accepted
Use Kanban board for development management	Enables iterative tracking of tasks and design artifacts	Accepted

As shown in table 11, this simplified ADR table enhances table 10 by emphasizing the deliverables of each phase, as well as the rationale and current status of significant design decisions, thus achieving the objective of documenting the eleven design objects within the paper.

5. Conclusion

In this paper, we presented the development of a Deep Learning software library in Java Programming with four-phase methodology: Preparation, Identification, Design, and Development, the focus of this paper is to lay the groundwork for the development of this software library, outlining the planning phases and the architectural design of the library's components. Our methodology unfolds across four sequential phases, each with distinct objectives and deliverables. The initial three: (1) Preparation, (2) Identification, and (3) Design, establish the foundational terminology, goals, and architectural blueprint of the deep learning library, defining its scope and guiding principles. In the final Development phase, we apply a tailored Software Development Life Cycle to translate these designs into working code modules and features.

During Preparation, we distilled an idea outline, assembled guiding research resources, and benchmarked comparable libraries. The Identification phase crystallized our design principles, defined technical constraints and product goals, and compiled the requisite toolset. In Design, we produced the software architecture diagram, detailed class and use-case diagrams, and operationalized these plans via a product roadmap and Kanban board. Finally, the Development phase delivered fully implemented, validated code modules that realize the library's core functionality.

By documenting our end-to-end planning and design workflow for a deep learning software library, this paper presents a comprehensive set of artifacts that reflect the activities and outputs of each phase, thereby enabling rigorous evaluation and continuous refinement in subsequent development cycles. Our current implementation prioritizes the construction of convolutional neural network layers and their supporting structures for image classification. This manuscript focuses exclusively on the preparatory and design stages, rather than the detailed adaptations of the SDLC. Accordingly, we defer a complete exposition of our customized SDLC to future publications, where we will report on its execution, lessons learned, and further enhancements as the project advances.

The primary focus of our proposed library is on CNNs, although the architecture was deliberately designed with generic abstractions which are Layer and Computation, to facilitate extension beyond CNN tasks. These abstractions separate the mathematical activity from its network context: a Layer specifies input/output agreements, whereas Computation encapsulates the numerical procedure. This division facilitates the integration of recurrent components (RNNLayer, LSTMLayer) and Transformer-style modules (AttentionLayer, SequenceMasking) without modifying the current CNN

interfaces. Attention kernels may be executed as specific Computation classes, while sequence masking can be incorporated at the Layer level to preprocess input batches. Support for mixed-precision arithmetic can be implemented by augmenting the Computation base class with precision-handling algorithms instead of altering convolution-specific routines. The essential refactoring needed to avert CNN-centric leaking is maintaining basic interfaces (such as tensor shape management and gradient propagation) as independent of spatial operations, while restricting CNN-specific code to subclasses. The present release prioritizes CNNs, although the design establishes a robust platform for the incorporation of RNNs, Transformers, and other deep learning frameworks in subsequent versions.

6. Declarations

6.1. Author Contributions

Conceptualization: A.R.B., O.C.R.R., and T.K.; Methodology: A.R.B.; Software: O.C.R.R.; Validation: A.R.B., O.C.R.R., and T.K.; Formal Analysis: A.R.B., O.C.R.R., and T.K.; Investigation: O.C.R.R.; Resources: A.R.B.; Data Curation: O.C.R.R.; Writing Original Draft Preparation: O.C.R.R., A.R.B., and T.K.; Writing Review and Editing: A.R.B., O.C.R.R., and T.K.; Visualization: T.K. All authors have read and agreed to the published version of the manuscript.

6.2. Data Availability Statement

The data presented in this study are available on request from the corresponding author.

6.3. Funding

The authors received no financial support for the research, authorship, and/or publication of this article.

6.4. Institutional Review Board Statement

Not applicable.

6.5. Informed Consent Statement

Not applicable.

6.6. Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E. D. Lusiana, S. Astutik, Nurjannah, and A. B. Sambah, "Using machine learning approach to cluster marine environmental features of Lesser Sunda Island," *Journal of Applied Data Sciences*, vol. 6, no.1, pp. 247-258, Jan. 2025, doi: 10.47738/jads.v6i1.478.

[2] A. Apandi, A. B. Mutiara, and Dharmayanti, "Generating image captions in indonesian using a deep learning approach based on Vision Transformer and IndoBERT architectures," *Journal of Applied Data Sciences*, vol. 6, no.2, pp. 1191-1202, May 2025, doi: 10.47738/jads.v6i2.672.

[3] V. Sumathi, B. L. Shivakumar, S. S. Maidin, and W. Ge, "Optimizing emergency logistics identification: utilizing a deep learning model in the big data era," *Journal of Applied Data Sciences*, vol. 5, no.3, pp.1440-1448, Sep. 2024, doi: 10.47738/jads.v5i3.369.

[4] Y. Liu, "Research on deep learning-based algorithm and model for personalized recommendation of resources," *Journal of Applied Data Sciences*, vol. 4, no.2, pp. 68-75, May 2023, doi: 10.47738/jads.v4i2.85.

[5] S. A. Qureshi, "Breast cancer detection using mammography: Image processing to deep learning," *IEEE Access*, vol. 13, no. 1, pp. 60776–60801, 2025, doi: 10.1109/ACCESS.2024.3523745.

[6] B. Jain, P. Pawar, D. Gada, and T. Patwa, "Comprehensive analysis of machine learning and deep learning models on prompt injection classification using natural language processing techniques," *International Research Journal of Multidisciplinary Technovation*, vol. 7, no. 2, pp. 24–37, Feb. 2025, doi: 10.54392/irjmt2523.

[7] H. Javed, S. El-Sappagh, and T. Abuhmed, "Robustness in deep learning models for medical diagnostics: security and adversarial challenges towards robust AI applications," *Artificial Intelligence Review*, vol. 58, no. 12, pp. 1-127, 2025, doi: 10.1007/s10462-024-11005-9.

[8] M. Mishra and J. G. Singh, "A comprehensive review on deep learning techniques in power system protection: trends, challenges, applications and future directions," *Results in Engineering*, vol. 25, no. 1, pp. 1-30, Mar. 2025, 10.1016/j.rineng.2024.103884.

[9] B. Ozdemir and I. Pacal, "A robust deep learning framework for multiclass skin cancer classification," *Scientific Reports*, vol. 15, id. 4938, no. 1, pp. 1-12, Feb. 2025, doi: 10.1038/s41598-025-89230-7.

[10] S. Imambi, K. B. Prakash, and G. R. Kanagachidambaresan, "PyTorch," in *EAI/Springer Innovations in Communication and Computing*, vol. 2021, no. 1, pp. 87–104, Jan. 2021, doi: 10.1007/978-3-030-57077-4_10.

[11] A. K. Jha, A. Ruwali, K. B. Prakash, and G. R. Kanagachidambaresan, "Tensorflow Basics," in *EAI/Springer Innovations in Communication and Computing*, vol. 2021, no. 1, pp. 5–13, Jan. 2021, doi: 10.1007/978-3-030-57077-4_2.

[12] O. C. R. Rachmawati, A. R. Barakbah, and T. Karlita, "Programming language selection for the development of deep learning library," *JOIV International Journal on Informatics Visualization*, vol. 8, no. 1, pp. 434-441, Mar. 2024, doi: 10.62527/joiv.8.1.2437.

[13] M. N. Shodiq, D. H. Kusuma, M. G. Rifqi, A. R. Barakbah, and T. Harsono, "Neural network for earthquake prediction based on automatic clustering in Indonesia," *JOIV International Journal on Informatics Visualization*, vol. 2, no. 1, pp. 37–43, Feb. 2018, doi: 10.30630/joiv.2.1.106.

[14] N. Ramadijanti, A. R. Barakbah, and F. A. Husna, "Automatic breast tumor segmentation using Hierarchical K-means on mammogram," *2018 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC),* Bali, Indonesia, vol. 2018, no. 1, pp. 170-175, Oct. 2018, doi: 10.1109/KCIC.2018.8628467.

[15] M. Subhan, A. Sudarsono, and A. R. Barakbah, "Classification of radical web content in indonesia using web content mining and k-Nearest Neighbor algorithm," *EMITTER International Journal of Engineering Technology*, vol. 5, no. 2, pp. 328–348, Jan. 2018, doi: 10.24003/emitter.v5i2.214.

[16] O. C. R. Rachmawati, A. R. Barakbah, and T. Karlita, "The comparison of activation functions in feature extraction layer using sharpen filter," *Journal of Applied Engineering and Technological Science (JAETS)*, vol. 6, no. 2, pp. 1254–1267, Jun. 2025, doi: 10.37385/jaets.v6i2.5895.

[17] F. Yang, J. M. Caballero, and R. A. Juanatas, "Designing of Chinese ancient poetry app based on iOS platform augmented reality and machine learning," *2022 7th International Conference on Business and Industrial Research (ICBIR)*, vol. 5, no. 1, pp. 285–288, 2022, doi: 10.1109/ICBIR54589.2022.9786385.

[18] Y. Cheon, "Multiplatform Application Development for Android and Java," *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, Honolulu, HI, USA, vol. 2019, no. 1, pp. 1-5, 2019, doi: 10.1109/SERA.2019.8886800.

[19] F. J. Agbo, S. S. Oyelere, J. Suhonen, and M. Tukiainen, "Design, development, and evaluation of a virtual reality game-based application to support computational thinking," *Educational Technology Research and Development*, vol. 71, no. 2, pp. 505–537, Oct. 2022, doi: 10.1007/s11423-022-10161-5.

[20] M. Fruhner and H. Tapken, "From persons to animals: transferring person re-identification methods to a multi-species animal domain," in *Proceedings of the 2024 9th International Conference on Multimedia and Image Processing*, Osaka, Japan, vol. 2024, no. 1, pp. 39-43, 2024, doi: 10.1145/3665026.3665032.

[21] F. Pérez-García, R. Sparks, and S. Ourselin, "TorchIO: A Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning," *Computer Methods and Programs in Biomedicine*, vol. 208, id. 106236, no. 1, pp. 1-12, Sep. 2021, doi: 10.1016/j.cmpb.2021.106236.

[22] D. N. Shibaeva, D. A. Asanovich, K. A. Malodushev, and D. A. Shamshura, "Development of a software package for systematization and analysis of research results of iron ore preparation characteristics: goals, objectives, first data," *Mining Industry Journal (Gornay Promishlennost)*, vol. 6, no. 1, pp. 99–106, 2024, doi: 10.30686/1609-9192-2024-6-99-106.

[23] Y. Romani, O. Tibermacine, and C. Tibermacine, "Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification," *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Honolulu, HI, USA, vol. 2022, no. 1, pp. 15-19, 2022, doi: 10.1109/ICSA-C54293.2022.00010.

[24] M. Zorzetti, I. Signoretti, L. Salerno, S. Marczak, and R. Bastos, "Improving agile software development using user-centered design and lean startup," *Information and Software Technology*, vol. 141, no. 1, pp. 1-12, 2022, doi: 10.1016/j.infsof.2021.106718.

[25] M. Kuhrmann, "What makes agile software development agile?," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3523-3539, Sep 2022, doi: 10.1109/TSE.2021.3099532.

[26] B. Şarlak, "Agile Methodology for Project/Process Management IT System Infrastructure," *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT),* vol. 2020, no. 1, pp. 1-7, 2020, doi: 10.1109/ICCCNT49239.2020.9225593.

[27] T. A. C. H. Chan, J. M.-B. Ho, and M. Tom, "Miro: promoting collaboration through online whiteboard interaction," *RELC Journal*, vol. 55, no. 3, pp. 871–875, Mar. 2023, doi: 10.1177/00336882231165061.

[28] E. Hendrawan, M. Meisel, and D. N. Sari, "Analysis and implementation of computer network systems using software Draw.io," *Asia Information System Journal*, vol. 2, no. 1, pp. 9-15, May 2023, doi: 10.24042/aisj.v2i1.18227.

[29] W. Tamrat, "The notion of relevance in academic collaboration," *International Journal of African Higher Education*, vol. 9, no. 3, pp. 133–151, Dec. 2022, doi: 10.6017/ijahe.v9i3.16051.

[30] S. Davies, P. S. Hakkarainen, and K. Hakkarainen, "Idea generation and knowledge creation through maker practices in an artifact-mediated collaborative invention project," *Learning, Culture and Social Interaction*, vol. 39, id. 100692, no. 1, pp. 1-12, Apr. 2023, doi: 10.1016/j.lcsi.2023.100692, doi: 10.1016/j.lcsi.2023.100692.

[31] K. Doulougeri, J. D. Vermunt, G. Bombaerts, and M. Bots, "Challenge-based learning implementation in engineering education: A systematic literature review," *Journal of Engineering Education*, vol. 113, no. 4, pp. 1076–1106, Oct. 2024, doi: 10.1002/jee.20588.

[32] O. C. R. Rachmawati and Z. M. E. Darmawan, "The comparison of deep learning models for Indonesian political hoax news detection," *CommIT (Communication and Information Technology) Journal*, vol. 18, no. 2, pp. 123–135, Aug. 2024, doi: 10.21512/commit.v18i2.10929.

[33] N. Singh, "Framework of goal-driven risk management in software development projects using the socio-technical Systems approach," *FIIB Business Review*, vol. 13, no. 4, pp. 437–451, Sep. 2021, doi: 10.1177/2319714521103241.

[34] M. Loor, A. Tapia-Rosero, and G. D. Tre, "An open-source software library for explainable support vector machine classification," *2022 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, vol. 2022, no. 1, pp. 1–7, 2022, doi: 10.1109/FUZZ-IEEE55066.2022.9882731.

[35] B. Singh and S. Gautam, "The Impact of Software Development Process on Software Quality: A Review," *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN),* Tehri, India, vol. 2016, no. 1, pp. 666-672, 2016, doi: 10.1109/CICN.2016.137.

[36] K. L. Petry, E. OliveiraJr, and A. F. Zorzo, "Model-based testing of software product lines: Mapping study and research roadmap," *Journal of Systems and Software*, vol. 167, id. 110608, no. 1, pp. 1-12, Sep. 2020, doi: 10.1016/j.jss.2020.110608.

[37] N. Damij and T. Damij, "An approach to optimizing Kanban board workflow and shortening the project management plan," *IEEE Transactions on Engineering Management*, vol. 71, no. 1, pp. 13266–13273, Nov. 2021, doi: 10.1109/TEM.2021.3120984.

[38] A. F. Del Carpio and L. B. Angarita, "Trends in Software Engineering Processes using Deep Learning: A Systematic Literature Review," *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA),* Portoroz, Slovenia, vol. 2020, no. 1, pp. 445-454, 2020, doi: 10.1109/SEAA51224.2020.00077.

[39] M. Hasan, "Finding the design pattern from the source code for developing reusable object oriented software," *2009 Second International Conference on the Applications of Digital Information and Web Technologies*, London, UK, vol. 2009, no. 1, pp. 157-162, 2009, doi: 10.1109/ICADIWT.2009.5273947.