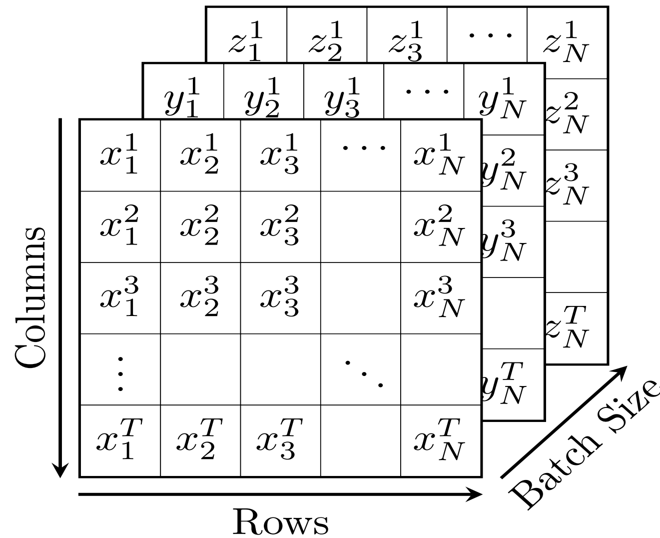
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Introducing the ModularML Framework

A transparent and modular machine learning framework made as a tool for research and education

Bachelor's thesis in Computer science and engineering

Måns Bremer
Tim Carlsson
Sander Carlsson
Elias Nilsson
William Norland
Gabriel Sättemo

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Bachelor's thesis 2025

Introducing the ModularML Framework

A transparent and modular machine learning framework made as a tool for research and education

Måns Bremer
Tim Carlsson
Sander Carlsson
Elias Nilsson
William Norland
Gabriel Sättemo

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

ModularML
A transparent and modular machine learning framework made as a tool for research and education
Måns Bremer Tim Carlsson Sander Carlsson Elias Nilsson William Norland Gabriel Sättemo

© Måns Bremer, Tim Carlsson, Sander Carlsson, Elias Nilsson, William Norland, Gabriel Sättemo 2025.

Supervisor (Handledare): Mateo Vázquez Maceiras, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering
Graded by teacher (Rättande lärare): Birgit Groher, Department of Computer Science and Engineering

Bachelor's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of a three-dimensional tensor where the dimensions are labeled rows, columns, and batch size.

Typeset in LaTeX
Gothenburg, Sweden 2025

ModularML
A transparent and modular machine learning framework made as a tool for research and education
Måns Bremer, Tim Carlsson, Sander Carlsson, Elias Nilsson, William Norland, Gabriel Sättemo

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis explores the process of creating a highly modular machine learning framework in C++, without performance compromises. The framework can parse ONNX models into dynamic C++ objects, modify the implementation of core computational functions (like GEMM), and use the models to perform inference. The framework reproduces results achieved in peer frameworks like PyTorch and TensorFlow. The usefulness of this framework stems from its pure C++ implementation, with no API layers to compiled modules or other black-box functionality. This makes it highly suitable for use in education and research, where debuggability, modularity, and ease of use are paramount.

Sammandrag (Swedish)

Denna avhandling utforskar processen att skapa ett mycket modulärt ramverk för maskininlärning i C++ utan prestandaförluster. Ramverket kan tolka en ONNX modell till dynamiska C++ objekt, modifiera implementationen av centrala beräkningsfunktioner (som GEMM) och använda modellerna för att utföra inferens. Ramverket uppnår resultat jämförbara med ramverk som PyTorch och TensorFlow. Användbarheten med detta ramverk är dess rena C++ implementation, utan API-lager till kompilerade moduler eller annan dold funktionalitet. Detta gör det särskilt lämpligt för utbildning och forskning, där felsökbarhet, modularitet och användarvänlighet är avgörande.

Keywords: Machine-Learning, Artificial-Intelligence, Computer-Vision, Neural-Network, Framework, LeNet, AlexNet, MNIST, ImageNet, Research-Tool.

Acknowledgements

We are very grateful for our inspiring and committed supervisor Mateo Vázquez Maceiras, whose guidance during this project has been invaluable. We would also like to extend our gratitude towards all the amazing developers who have worked, contributed, and are still maintaining all the previous frameworks that have provided us with a foundation for this project. We especially want to thank the developers and maintainers of ONNX without whom this would not be possible.

Måns Bremer, Tim Carlsson, Sander Carlsson, Elias Nilsson, William Norland, Gabriel Sättemo, Gothenburg, May 2025

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Using Neural Networks for Image Classification . . . 1

1.1.2 Contemporary Frameworks . . . 2

1.2 Purpose . . . 3

1.3 Goals . . . 3

1.4 Limitations / Demarcations . . . 3

1.5 Ethics and Model Transparency . . . 4

2 Theory **5**

2.1 Machine Learning Fundamentals . . . 5

2.1.1 Deep Neural Networks . . . 5

2.1.2 Fully Connected Layer . . . 6

2.1.3 Convolutional Layer . . . 7

2.1.4 Pooling Layer . . . 9

2.1.5 Local Response Normalization . . . 10

2.1.6 Flatten . . . 10

2.1.7 Activation Functions . . . 11

2.1.8 Vectorization and Normalization of Images . . . 13

2.2 General Matrix Multiplication . . . 13

2.3 Optimization . . . 14

2.3.1 Blocked GEMM . . . 14

2.3.2 Single Instruction, Multiple Data . . . 14

2.3.3 Basic Linear Algebra Subprograms . . . 15

2.4 Dataset Challenges . . . 15

2.4.1 MNIST Challenge . . . 15

2.4.2 ImageNet Challenge . . . 16

2.5 Machine Learning Models . . . 16

2.5.1 LeNet . . . 17

2.5.2 AlexNet . . . 17

2.5.3 ResNet-18 . . . 18

2.6 ONNX . . . 19

3 Methods **21**
3.1 Requirements Analysis 21
3.2 Research and Design 21
3.3 Implementation 22
3.4 Parser Implementations 22
3.5 GEMM Implementations 22
3.6 ONNX Standard and Documentation 22
3.7 Standard Functions 23
3.8 Testing and Evaluation 23
3.8.1 Dataset Comparisons 23
3.8.2 Model Loading and Execution 23
3.8.3 Performance Evaluation 23
3.8.4 Code Evaluation 24
3.8.5 Modularity and Transparency Evaluation 24

4 Results **25**
4.1 Dataset Comparisons 25
4.2 Model Loading and Execution 26
4.3 Performance Evaluation 27
4.3.1 Naive GEMM Backend 27
4.3.2 Blocked GEMM Backend 27
4.3.3 OpenBLAS GEMM Backend 27
4.3.4 AVX2 GEMM Backend 29
4.3.5 AVX512 GEMM Backend 29
4.3.6 Overall GEMM Improvement 29
4.4 Test Coverage Results 30
4.5 Modularity and Transparency 30

5 Conclusion **33**
5.1 Achieved Goals 33
5.2 Data Challenge Results 34
5.3 Performance 34
5.4 Lessons Learned 34
5.5 Further Knowledge and Future Work 35

Bibliography **37**

Bibliography **37**

A Appendix 1 **I**
A.1 Project Source Code I
A.2 ResNet-18 Architecture I
A.3 First Convolution in LeNet, outlined in JSON parsed by ModularML II

List of Figures

2.1 An overview of CVNN and DNN. The type of network displayed can be called a fully connected feed-forward neural network. 6
2.2 Visualization of an input tensor connected to an output by a fully connected layer. 7
2.3 An overview of how a kernel performs a convolution. 8
2.4 Illustration of 2×2 max pooling (kernel size = 2×2, stride = 2). . . . 9
2.5 Visualization of Local Response Normalization 11
2.6 Visualization of a tensor before and after a flatten operation 11
2.7 Plot of the ReLU activation function. 12
2.8 Image from the 2012 ILSVRC [20], CC BY-NC: Attribution-NonCommercial. Should be predicted as label 970, “Alps”. 16
2.9 Overview of the LeNet-5 architecture. 17
2.10 Overview of the AlexNet architecture 18
2.11 $y = F(ReLU(x)) + ReLU(x)$ identity shortcut 18

3.1 Flowchart of how the framework works. 21

4.1 Handwritten digit 5, used as input for LeNet-5 running on ModularML. 26
4.2 American (great) egret, used as the input image for AlexNet and ResNet-18 running on ModularML. 26
4.3 Pie charts illustrating the runtime breakdown using naive GEMM backend. 27
4.4 Pie charts illustrating the runtime breakdown when using the Blocked GEMM backend. 28
4.5 Pie charts illustrating the runtime breakdown when using the Open-BLAS GEMM backend. 28
4.6 Pie charts illustrating the runtime breakdown when using the AVX2 GEMM backend. 29
4.7 Pie charts illustrating the runtime breakdown when using the AVX512 GEMM backend. 30
4.8 Example of multiple implementations in their respective directories. . 31

A.1 ResNet-18 Architecture I
A.2 Resulting coverage report of the framework. III

List of Tables

4.1 Dataset results 25
4.2 GEMM improvements compared to naive implementation. 30

1

Introduction

This chapter introduces the historical development and application of neural networks within the field of Artificial Intelligence (AI). It provides background information on fundamental neural network architectures, outlines the objectives and goals of the thesis, and defines the scope and limitations of the work.

1.1 Background

The use of AI for image classification has grown from being a novelty that had difficulty recognizing handwritten digits in problems like The Modified National Institute of Standards and Technology (MNIST) dataset [1], to producing good results in even harder issues such as the ImageNet challenge. Currently, AI is widely utilized, and the ImageNet challenge has been solved [2]. This progress is partly due to breakthrough architectures such as LeNet-5 (1998) [1] and AlexNet (2012) [3]. Parallel to these strides in architecture is the rise of modular, easy to use, Machine Learning (ML) frameworks like PyTorch [4] and TensorFlow [5], which are today household names and powerful tools for engineers, researchers, and students alike.

Still, these frameworks do not provide adequate transparency. This is an effect of having the actual implementation of their different functions obfuscated behind multiple code and API layers. This separation of user client (what the user sees when using the framework) and computational back end (where the computations are performed) leads to difficulties using code profiling tools like Performance Application Programming Interface (PAPI) [6] or Likwid [7], using simulators like gem5 for profiling or running on bare metal [8], low-level debugging and the ability to change fundamental parts of the code. This work aims to explore a solution to this problem by creating a new, self-contained, ML framework that does not rely on an API for other modules—making it simple and more intuitive to modify all parts of the framework, solving the issues stated above.

1.1.1 Using Neural Networks for Image Classification

The use of the Convolutional Neural Network (CNN) architecture for tasks such as image classification (An ML model that takes an image as input and classifies it) stretches back even further than this current peak in public interest for AI. This is caused by advancements in Large Language Model (LLM) technology [9] and releases of AI chatbots such as ChatGPT [10]. The origins of CNN stem from

attempts at beating a challenge known as the MNIST Challenge [1], which involves a computer program looking at a dataset containing handwritten digits (images) and attempting to predict the corresponding number. This proved a hard task for contemporary Feed-Forward Neural Networks (FFNN) and as such, there was a need for a new type of architecture. A remedy came from the seminal work of Yunn LeCun et al. where in 1998 they presented the LeNet-5 architecture [1], which among other things, brought the use of Convolutional Layers (CL) and the CNN. LeNet-5 achieved an accuracy of around 99% on MNIST leaving very little room for improvement. This gave birth to a new challenge known as the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). Instead of digits, ILSVRC involves identifying what a given item in a picture is. The breakthrough architecture of this challenge came to be the CNN AlexNet [3] which in 2012 won the challenge by a significant margin. The ImageNet challenge was last held in 2017 when it was decided that this challenge, like MNIST before it, had served its purpose and, had become too easy considering 2017 winner SENet got a top-5 score of 97.749% [11].

1.1.2 Contemporary Frameworks

To create or use neural network models, including CNNs, there are several different frameworks available in several different programming languages. According to an empirical study made in 2022, the most commonly used (by plurality) were the frameworks PyTorch and TensorFlow (used in combination with Scikit-learn and Keras) [12]. These frameworks enable the user to create and train their models as well as import third-party models. For this purpose (using other people's models) files in the Open Neural Network Exchange (ONNX) format can be used, which contains the data necessary to reconstruct a model in a framework of choice [13]. Whilst frameworks like PyTorch and TensorFlow give the user a powerful set of tools, they lack transparency. Transparency, in this context, means to which extent the user can see and modify the implementation of back-end functionality. For reference, Abadi et al. [5] shows the clear separation existing between the Python/C++ client and the C API. Naturally, this creates an issue for some users. Users learning about neural networks would benefit greatly by seeing and implementing the step-by-step processes that the model goes through during inference themselves. Users dealing with hardware and algorithms would also benefit from having an easy way to modify the model's inference steps or change individual layers. It would improve users' ability to optimize the performance of models. In summary, users would benefit from:

- Gaining a readable source code level insight into how the model computes its results.
- Being able to make their implementations of back-end functions for the sake of learning and/or optimization.
- Being able to inject profiling code into back-end functions.
- Being able to switch out layers, activation functions, and other components from imported models.

1.2 Purpose

The purpose of this project was to create a new ML framework designed to offer a more transparent and flexible platform to explore ML models. A framework enabling users to implement, modify, and compare core components in ML models to foster a greater understanding of the underlying processes that make ML possible. Fostering understanding, primarily by users learning how neural network components function through a flexible base for experimentation.

1.3 Goals

The primary, overarching, goal of this project was to design and implement a working ML framework software in C/C++ that serves the purpose defined in 1.2.

To support and break down this overarching goal into achievable, and measurable pieces the following five sub-goals guided the project:

G.1 Learn how to structure and design an ML framework from the ground up.
G.2 Dynamically generate data structures that model machine learning models using C/C++.
G.3 Enable users to easily select among multiple implementations of core back-end functions.
G.4 Develop functionality to read and execute existing models and their associated weights in the ONNX format.
G.5 Create easily debuggable code that can be followed step by step.

These goals bridge the gap between high-level ML frameworks and the underlying hardware/software operations, allowing for better understanding and experimentation for both students and researchers alike.

1.4 Limitations / Demarcations

ModularML is designed strictly as a framework. It is implemented in C/C++ and, in its most fundamental form, enables the execution of pre-trained ML models that adhere to the ONNX format [13]. Because of time constraints, training models using ModularML is considered outside the scope of the project. Time constraints are also the reason why formats other than ONNX were not explored. The primary focus and goal of this project is to provide a transparent environment for running existing models. Given the focus on transparency (avoiding existing pre-compiled backends that are often heavily optimized) and the limited time available, the primary goal has not been to compete with the performance of well-established platforms but rather to ensure clarity and accessibility in model execution. However, this does not mean that ModularML is limited to its current performance as it can easily be extended.

ModularML does, in the scope of this thesis, focus exclusively on Image-classifying

models.

ModularML does not include functionality to be able to parse ONNX directly as creating an ONNX parser would require a substantial amount of time to fully cover the ONNX specification. Furthermore, the parser itself would not address the core problems that the framework intends to solve, it is therefore deemed as out of scope for this project.

ModularML could not rely on API layers to other modules or compiled code for computations. This restriction ensured that every component of the framework remained fully observable, debuggable, modifiable, and replaceable by the user.

ModularML is built to run on and is verified on Linux systems. Other UNIX-based systems, especially MacOS have limited support.

Mechanisms to ensure that the models executed within ModularML always produce ethical content [14] are not explored as they are outside the scope of the project. Additionally, for the same reason, ModularML does not incorporate any form of online verification nor model checking to validate generated outputs [15].

1.5 Ethics and Model Transparency

The field of ML as a whole raises a lot of ethical "what-ifs" that must be considered. A multitude of higher-level ethical frameworks — addressing transparency, justice, fairness, non-maleficence, responsibility, and privacy [16] — have been proposed to guide the ethical development and deployment of AI systems. ModularML focuses on the evaluation phase and aims to assess the ethical issues relevant to this role.

During evaluation, it is crucial to detect potential ethical concerns before deploying a model. Developers must examine all issues outlined in ethical frameworks [16] using various tools and techniques, though importantly no single tool can address every concern.

ModularML might be seen as a tool that promotes more transparent AI models, though it is not intended to achieve the broad AI transparency emphasized in many guidelines. In contrast, the framework provides what could be referred to as algorithmic transparency, focusing on the technical mechanics of how ML models are executed. Larsson and Heintz make the distinction as, AI transparency considers the entire system, influenced by factors such as literacy, information asymmetries, model-close explainability, and governance, while algorithmic transparency focuses on individual algorithms or components [17].

ModularML will achieve algorithmic transparency by allowing students and researchers to tinker with the underlying components of ML models, permitting users to inspect, modify, or implement their components. By enhancing the understanding of model behavior at a lower level, ModularML aims to heighten ML literacy.

2
Theory

This chapter provides the theoretical foundation behind the development of the ModularML framework. It outlines key machine learning concepts, architectural design principles, and performance optimization strategies that guide the project. Emphasis is placed on model representation standards, such as ONNX, and profiling tools relevant to low-level execution analysis.

2.1 Machine Learning Fundamentals

An ML model is a statistical model that learns patterns from training data, which the model then uses to make predictions to generate or classify content. A model consists of several different components, often referred to as “layers”. Each layer can differ greatly in terms of what operation it performs. Data within the model is represented by tensors, which are multidimensional arrays. To structure the flow of tensors through a model and its different layers, a model is generally represented as a directed computational graph, where each node represents an operation.

2.1.1 Deep Neural Networks

Deep Neural Networks (DNNs) can be described as a graph divided up into layers where each layer is connected to the preceding layer to some degree. The main difference between DNN and conventional neural networks (CVNN) is the depth or number of layers included in the model. A CVNN contains at most 3 layers while a DNN has 4 or more [18]. An illustration of the difference between CVNNs and DNNs can be seen in Figure 2.1. Note that layers do not have to be uniform in size. The larger amount of layers in a DNN allows it to capture a more complex, nonlinear relationship compared to a CVNN, and this can lead to more accurate predictions. A DNN is a statistical model, that tries to predict an output from a given input. This becomes more clear when it is understood that, even though often visualized as a graph or a set of layers, a DNN mainly consists of matrices (or more generalized, tensors) of numbers called weights (w_i) and biases (b_n). These matrices are multiplied with each other in a process referred to as Forward Propagation (FP) finally resulting in a vector of values representing the prediction of the model. In Equations 2.1-2.4 you can see, mathematically, how forward propagation could work in the CVNN from 2.1. The whole process of the model receiving input and producing output is referred to as inference, to differentiate inference from FP you can consider FP as the internal computational process and inference as an action available to the

model user.

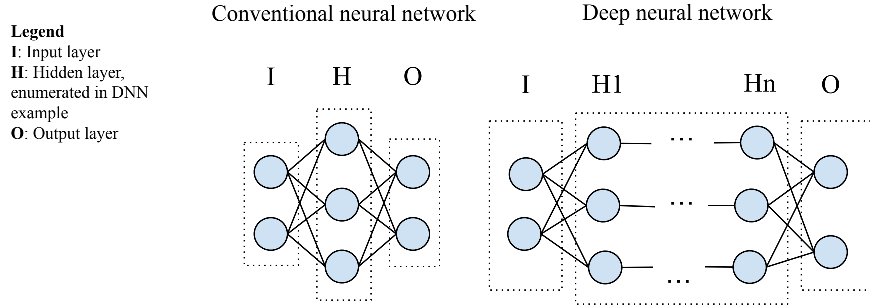


Figure 2.1: An overview of CVNN and DNN. The type of network displayed can be called a fully connected feed-forward neural network.

$$\overrightarrow{V_{in}} = [i_1, i_2]^\top \quad \text{(input)} \tag{2.1}$$

$$f_a : \mathbb{R}^n \to \mathbb{R}^n \quad \text{(activation function)} \tag{2.2}$$

$$\overrightarrow{V_{p1}} = f_a \left(\overrightarrow{V_{in}} \cdot \begin{bmatrix} w_{111} & w_{112} & w_{113} \\ w_{121} & w_{122} & w_{123} \end{bmatrix} + \begin{bmatrix} b_{111} \\ b_{112} \\ b_{113} \end{bmatrix} \right) \tag{2.3}$$

$$\overrightarrow{V_{out}} = f_a \left(\overrightarrow{V_{p1}} \cdot \begin{bmatrix} w_{211} & w_{212} \\ w_{221} & w_{222} \\ w_{231} & w_{232} \end{bmatrix} + \begin{bmatrix} b_{221} \\ b_{232} \end{bmatrix} \right) \quad \text{(prediction)} \tag{2.4}$$

2.1.2 Fully Connected Layer

Figure 2.2 shows a fully connected layer (FC), also known as a dense layer is a very common and important layer for machine learning models. As the name suggests, the layer connects all neurons to all the neurons of the preceding layer. Each connection between neurons has a weight that determines how much influence the input neuron has on the output neuron. Since weights connect all neurons, all features will be accounted for when calculating the result, making it possible to recognize complex and intricate patterns.

In practice, the layer takes an input tensor and then multiplies it with its weights, also in the form of a tensor. This will effectively perform a linear operation on the input, allowing the network to recognize linear relationships. After the weighted sum is computed a bias term is added. The purpose of the bias is to enable each neuron to shift its output of the linear transformation independently of the input and additionally enable the output of each neuron to not have to be constrained to

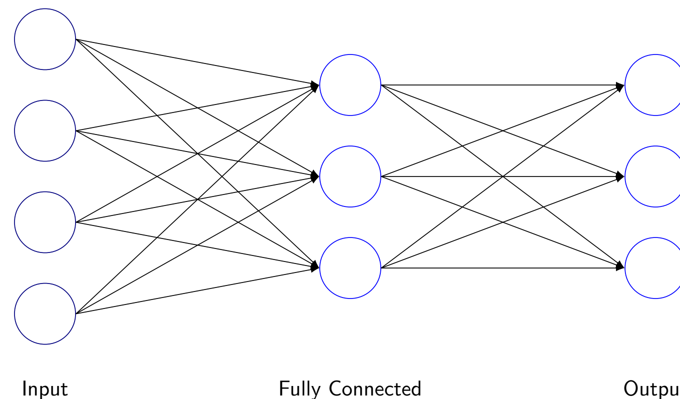


Figure 2.2: Visualization of an input tensor connected to an output by a fully connected layer.

the origin. This increases the range of patterns that can be modeled and thus its accuracy [19].

Although optional, an activation function is often included. The activation function performs a mathematical operation on each element of the output tensor individually, which in turn allows the layer to detect relationships that are not linear. This is crucial in most networks, considering most real-life scenarios are rarely possible to model with a linear model. Equation 2.5 defines the final layer.

$$\mathbf{y} = activation(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.5}$$

Where:

- $\mathbf{x} \in \mathbb{R}^n$ is the input vector,
- $\mathbf{W} \in \mathbb{R}^{m \times n}$ is the weight matrix,
- $\mathbf{b} \in \mathbb{R}^m$ is the bias vector,
- $activation(\cdot)$ is a non-linear activation function (e.g., ReLU, sigmoid),
- $\mathbf{y} \in \mathbb{R}^m$ is the output vector.

2.1.3 Convolutional Layer

Convolutional Layers (CL), also contain matrices of weights. These weights, however, work a bit differently than the ones in an FC, instead consisting of k learnable filters referred to as kernels and represented by tensors. The kernel's dimensions are derived from the dimensions of the input. These kernels are then convolved or slid over the channels of the input image pixel values creating feature maps (see Figure 2.3). Consider the reference Figure 2.3 where the pixels are encoded in red, green, and blue color values (RGB), then, this image would be divided into three initial channels. If the first layer has k kernels, then there would be k channels after the image passed through it. The rate at which the kernel slides across the input is called the stride; the stride affects the size of the feature map as a lower stride means more iterations until the kernel has traversed all the input data. As you might have noticed this operation can reduce the original size of the image. To help with

this a CL can apply padding to the channels of an input image (numbers, often zeroes) that is placed around the images mitigating the shrinking effect to some degree.

The purpose of the CL is to try to capture invariant parts of the image despite these parts having some variations. Consider for example a hand-drawn number six. Regardless of who drew the number six, we would expect it to contain a loop and a curved line, this is what makes a six a six and this is precisely what the CL is trying to capture [20]. Figure 2.3 shows an overview of how a kernel performs

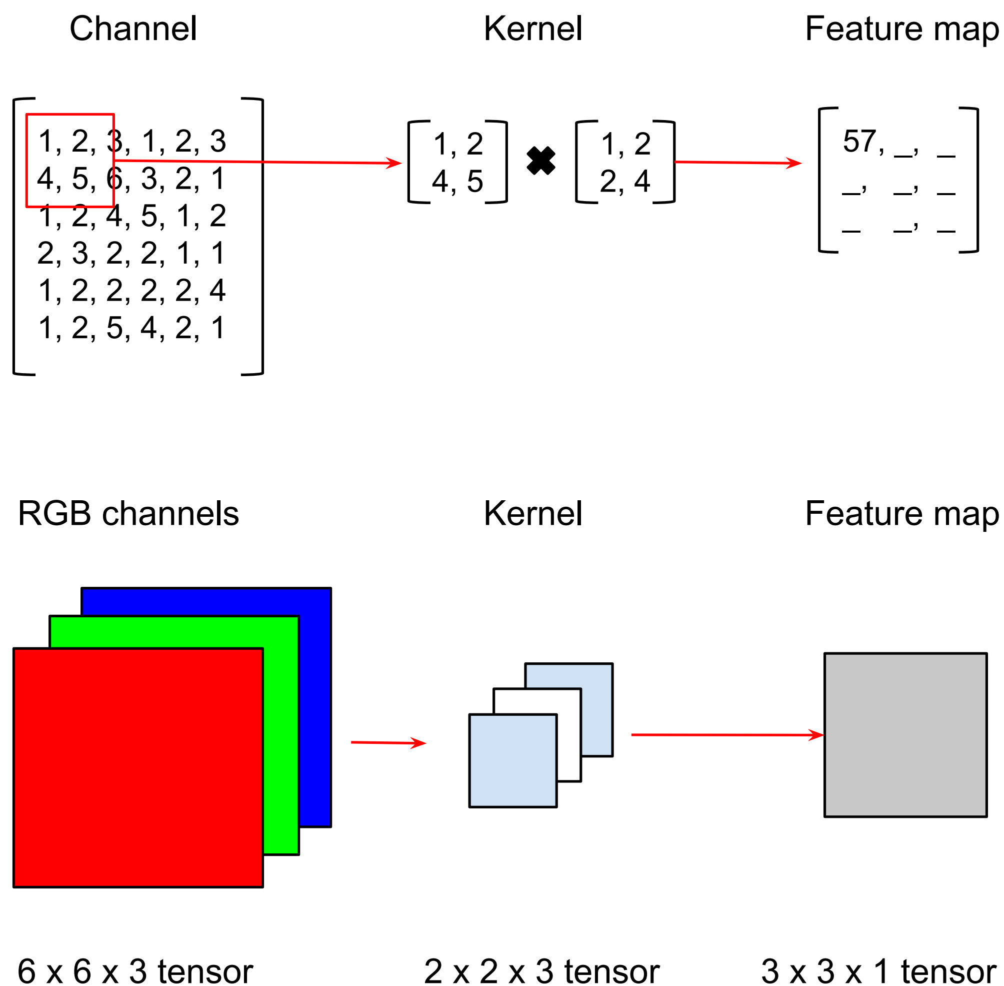


Figure 2.3: An overview of how a kernel performs a convolution.

a convolution with a stride of two and padding of zero. The result of the matrix multiplication is summed to a single element in the feature map. Note that the top visual can be seen as a cross-section of how one channel interacts with one layer of the kernel.

2.1.4 Pooling Layer

Pooling layers play a vital role in CNNs by reducing the spatial dimensions (height and width) of feature maps. This downsizing cuts down on both the number of parameters and the computational load, while also introducing some tolerance to small shifts in the input. As a bonus, smaller feature maps mean fewer operations in the layers that follow, boosting overall processing speed.

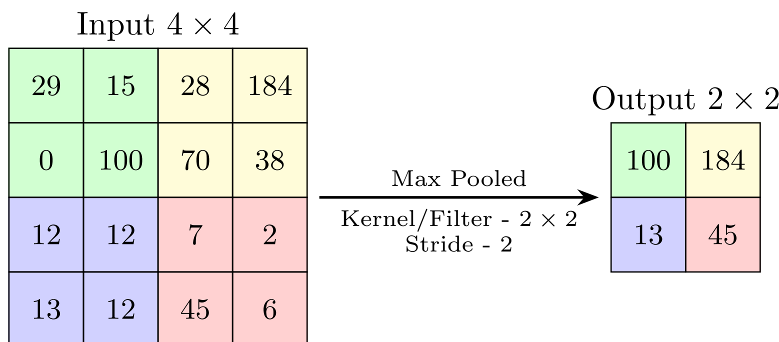


Figure 2.4: Illustration of 2 × 2 max pooling (kernel size = 2 × 2, stride = 2).

Figure 2.4 demonstrates how a 2×2 window with stride two summarizes each block of four pixels into a single summary value — converting a 4×4 input into a 2×2 output.

Max Pooling

Max pooling replaces each $k \times k$ window with its highest activation. Formally, for an input tensor (Equation 2.6) using window size k and stride s, the output (Equation 2.7) is defined by Equation 2.9 with index ranges defined as 2.10.

$$X \in \mathbb{R}^{H \times W \times C} \tag{2.6}$$

$$Y \in \mathbb{R}^{H' \times W' \times C} \tag{2.7}$$

$$H' = \left\lfloor \frac{H-k}{s} \right\rfloor + 1, \qquad W' = \left\lfloor \frac{W-k}{s} \right\rfloor + 1, \tag{2.8}$$

$$Y_{i,j,c} = \max_{0 \le p,q < k} X_{s\,i+p,\; s\,j+q,\; c} \tag{2.9}$$

$$i = 0, \ldots, H'-1, j = 0, \ldots, W'-1, c = 0, \ldots, C-1, (p,q) \in [0,k) \tag{2.10}$$

By selecting only the strongest response in each region, max pooling helps preserve critical features like edges, making the network less sensitive to slight translations. For example, in object detection, this ensures that key contours remain highlighted even if they shift by a pixel.

Average Pooling

Average pooling summarizes each window by its mean activation (see Equation 2.11).

$$Y_{i,j,c} = \frac{1}{k^2} \sum_{p=0}^{k-1} \sum_{q=0}^{k-1} X_{s\,i+p,\; s\,j+q,\; c} \tag{2.11}$$

Using the same index ranges as defined as 2.10, averaging produces a gentler representation of each patch, which helps carry through subtle gradients and textures. In applications like scene segmentation, this smoother representation aids in distinguishing large regions based on overall color and texture patterns.

2.1.5 Local Response Normalization

Local Response Normalization (LRN) is a normalization layer. It implements lateral inhibition, which is a concept that stems from neurobiology. In a biological brain, when a neuron is strongly activated, it will suppress its close neighbors. This will create a clear local maximum, which in turn will increase sensory perception [21].

The concept is used in ML to increase characteristics by highlighting stronger neurons. In turn, this creates stronger contrasts between neurons and thereby makes local key points stand out more prevalently. The lateral inhibition from neurobiology is in this context implemented on neurons at the same spatial location, across nearby channels. It enhances features (such as edges, textures, etc.) by enhancing strongly activated neurons while suppressing lesser ones. Using the square sum of activations across nearby neurons, the relevant neuron weight can be increased when the surrounding neuron weights are lesser in the presence of larger neurons. The mathematical formula is represented in Equation 2.12. A visualization of what happens during LRN is provided in Figure 2.5.

$$b_c = a_c \left(k + \frac{\alpha}{n} \sum_{c'=\max(0,c-n/2)}^{\min(N-1,c+n/2)} a_{c'}^2 \right)^{-\beta} \tag{2.12}$$

Where:

- $n \in \mathbb{N}$ is the size of the local neighborhood of channels around the channel $\mathbf{c}$,
- $\mathbf{a_c} \in \mathbb{R}^m$ is the activation of the neuron at channel $\mathbf{c}$,
- $\mathbf{b_c} \in \mathbb{R}^{m \times n}$ is the normalized output of the neuron at channel $\mathbf{c}$,
- k is a small hyperparameter used to stabilize the results,
- α, β are hyperparameters used for scaling and sensitivity,

2.1.6 Flatten

After pooling or a CL, the tensor dimensions can vary. For the data to apply to an FC layer, it needs to be reshaped. Therefore, a flatten layer is commonly applied after pooling or a CL. The flatten layer takes the tensor, of whatever dimension, and reshapes each sample to one dimension. The resulting tensor will be of two dimensions since the batch dimension persists. A visualization of this can be seen in Figure 2.6.

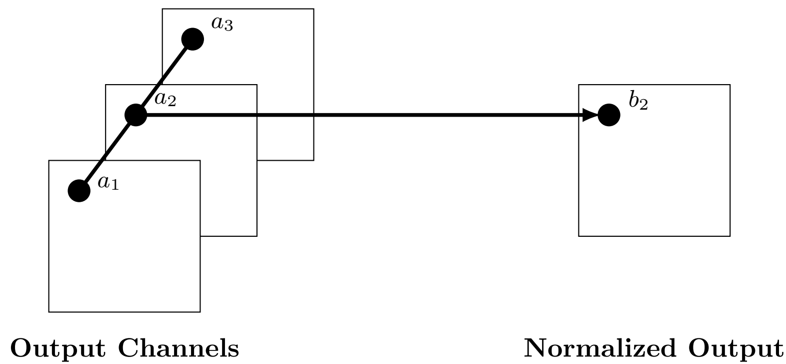


Figure 2.5: Visualization of Local Response Normalization

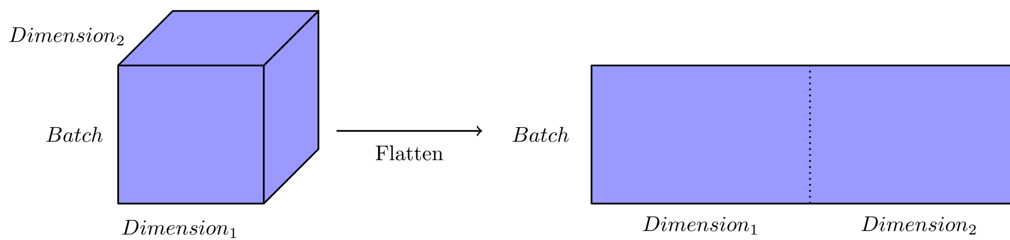


Figure 2.6: Visualization of a tensor before and after a flatten operation

2.1.7 Activation Functions

This subsection describes some common and important activation functions. Some are more briefly described than others due to their simplicity or relative insignificance.

ReLU

Rectified Linear Unit activation function (ReLU), is applied elementwise to tensors defined by Equation 2.13 and visualized in Figure 2.7.

$$\text{ReLU}(x) = \begin{cases} x, & x > 0, \\ 0, & x \leq 0. \end{cases} \tag{2.13}$$

During inference, ReLU introduces nonlinearity by passing forward only positive values and setting negatives to zero. This usually reduces the output because many activations become zero, which in turn can reduce computation and speed up the forward pass. Since the operation is just a simple comparison and assignment, it is very efficient to compute, making ReLU one of the standard choices for activations in modern convolutional and fully connected networks.

Leaky ReLU

A variation of ReLU that allows a small gradient when the unit is inactive (negative inputs). Unlike the standard ReLU function, Leaky ReLU assigns a small, non-zero value (often 0.01) to negative values.

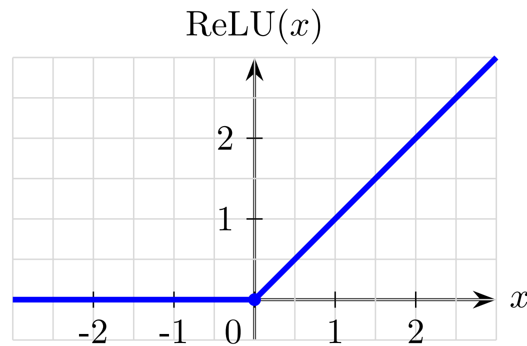


Figure 2.7: Plot of the ReLU activation function.

Softmax

The Softmax activation function converts a vector of numerical values with a length of K, into a probability distribution. Given an input vector $\mathbf{z} = (z_1, z_2, \ldots, z_K)$, Softmax is defined by Equation 2.14.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \ldots, K \tag{2.14}$$

In other words, each output is the exponential of its input score divided by the sum of exponentials of all scores, ensuring Equation 2.15 holds.

$$\sum_{i=1}^{K} \text{Softmax}(z_i) = 1 \tag{2.15}$$

During inference, Softmax is typically used in the final layer of a classification network. It turns the network's raw output scores into non-negative values that sum to one, which can then be interpreted as the probability of each class, rendering it straightforward to select the class with the highest probability as the model's top prediction. This also allows for analysis if the model's top prediction is incorrect and if any of its other predictions are correct.

Log Softmax

Applies the natural logarithm to the result of the softmax function, often used for numerical stability in classification tasks.

$$\text{LogSoftmax}(z_i) = \log\left(\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) = z_i - \log\left(\sum_{j=1}^{K} e^{z_j}\right) \tag{2.16}$$

Sigmoid

An activation function that normalizes its input to the range (0, 1), is often used in binary classification. It's defined as:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.17}$$

Swish

A smooth, non-monotonic (a function that doesn't consistently increase or decrease, it can switch direction.) activation function defined as $x \cdot \text{Sigmoid}(x)$ 2.17 is known for improving model accuracy in some contexts.

TanH

An activation function that maps inputs to the range (-1, 1), centered around zero. It's defined as:

$$\text{TanH}(x) = \frac{\sinh x}{\cosh x} \tag{2.18}$$

2.1.8 Vectorization and Normalization of Images

As mentioned before, neural networks operate on tensors, which are multidimensional arrays that represent data. To process an image it must first be converted into a tensor, a step often referred to as image vectorization. This transformation restructures the pixel data into a format suitable for input into a neural network. In addition to restructuring the data to a tensor and depending on needs, it may also be necessary to normalize it.

Image normalization scales pixel values to a standardized range, commonly between (0,1) or (-1,1) for each channel, and often centers the distribution around zero. This standardization helps neural networks converge faster during training and can reduce the risk of overfitting, as it ensures no single feature disproportionately influences the learning process [22].

2.2 General Matrix Multiplication

One of the most important operations in ML is General Matrix Multiplication (GEMM). It is a software implementation of matrix multiplication, the most used mathematical operation in ML. It is the backbone of common operations such as FCs and CLs. GEMM is mathematically defined by Equation 2.19.

$$C = \alpha AB + \beta C \tag{2.19}$$

Where:

- A is a matrix of size m x k,
- B is a matrix of size k x n,

- α is a scalar multiplier applied to the matrix product of AB,
- β is a scalar multiplier applied to the original matrix C,
- C is both the input and updated output matrix

GEMM is often in focus in ML frameworks since it does most of the heavy lifting. It will therefore have a significant impact on the performance of each model. At its most elementary implementation, it performs inner product matrix-matrix multiplication, described by Equation 2.20.

$$C_{ij} = \alpha \sum_{l=1}^{k} A_{il} B_{lj} + \beta C_{ij} \tag{2.20}$$

For example, the element C_{11} is calculated by taking the dot product of the first row of matrix A and the first column of matrix B. Equation 2.21 provides a scaled explanation of this.

$$C_{11} = \alpha(A_{11}B_{11} + A_{12}B_{21} + ... + A_{1k}B_{k1}) + \beta C_{11} \tag{2.21}$$

When k becomes larger, i.e. when the matrix dimensions increase, this operation will become quite heavy. Requiring a lot of calculations from the processor which will affect the runtime performance of the model. To optimize this, several implementations utilize parallelization, hardware acceleration, and other clever optimization techniques.

2.3 Optimization

This section introduces several different optimizations of GEMM 2.2 that were implemented in ModularML.

2.3.1 Blocked GEMM

Blocked GEMM is an optimization that leverages data locality to improve performance. Instead of operating across single matrices, the blocked approach splits the computation across several sub-matrices or 'blocks' based on a set block size. By choosing an appropriate block size that matches the host's Central Processing Unit (CPU) cache sizes, the sub-matrices can fit exactly into the CPU cache during program execution. Re-accessing elements needed to calculate each block can then be retrieved from fast cache memory, as opposed to slower main memory.

2.3.2 Single Instruction, Multiple Data

Single instruction, multiple data (SIMD), is a type of computational architecture that enables the CPU to use a single instruction to perform computations in parallel on multiple pieces of data. The use of SIMD is applicable when an operation is performed uniformly on large pieces of data, as is the case in matrix-multiplication which entails a large number of additions and multiplications.

The majority of modern CPUs offer native SIMD support through special instruction sets such as Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) [23] for x86 architectures or NEON for ARM architectures [24]. Intrinsics is the interface through which developers interact with these types of instructions, which are assembly-coded functions that enable developers to use C++ function calls and variables instead of inline assembly code, improving readability and ease of use [23].

The Advanced Vector Extensions 2 (AVX2) instruction set is an extension to the x86 SIMD instruction set created by Intel, providing 256-bit wide SIMD registers with support for vector operations. This enables data to be processed in parallel, offering a significant performance increase when utilized in matrix and vector operations such as GEMM.

The Advanced Vector Extensions 512 (AVX512) instruction set is a further extension to AVX2, providing 512-bit wide SIMD registers, doubling the number of elements that can be processed in parallel with a single instruction. However, AVX512 is not as widely supported as AVX2 and is significantly more power-hungry when utilized.

2.3.3 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) is a standard low-level interface for dense linear-algebra kernels [25]. Its Level-3 component, among other things, centers on GEMM. In code, conforming to the BLAS interface allows utilization of several available BLAS libraries, such as OpenBLAS [26], Math Kernel Library [27], and cuBLAS [28].

2.4 Dataset Challenges

A dataset challenge, often called a benchmark competition, is an organized benchmark in which the organizers release a fixed dataset, publish a standard train/test split, and prescribe a single evaluation metric and submission protocol. Since every participant must train and test on the same data and report results in the same way, differences in scores can be attributed primarily to the models or networks themselves, providing a fair and reproducible basis for comparing competing methods.

2.4.1 MNIST Challenge

MNIST is a dataset challenge consisting of 70,000 grayscale images of handwritten digits (0-9), each normalized and centered within a 28×28 pixel frame. An example of such an image can be seen in Figure 4.1. Out of these 70,000 digits, 60,000 are used for training and 10,000 are used for validation [1].

2.4.2 ImageNet Challenge

There are many iterations of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [29]. One such is the 2012 ILSVRC challenge [3]. Consisting of around 1.2 million 224 × 224 RGB images used for training, whilst covering 1000 different object classes, the 2012 ILSVRC challenge tests a model against a total of 50,000 validation images, comparing the output to the expected object class for each one. An example of an image from the 2012 ILSVRC can be seen in Figure 2.8.

Figure 2.8: Image from the 2012 ILSVRC [20], CC BY-NC: Attribution-NonCommercial. Should be predicted as label 970, “Alps”.

2.5 Machine Learning Models

A machine learning model is a statistical model that consists of a set of operations with adjustable parameters often called ‘weights’ and ‘biases’, that learns to turn inputs (like images or feature vectors) into outputs, such as class labels or numerical predictions.

Training is the process of finding the best parameter values so that these operations closely match a set of example input-output pairs (the training data). An optimization algorithm (e.g. gradient descent) nudges the weights and biases step by step, minimizing the difference between the model’s predictions and the true labels.

Inference happens once training is complete. By providing the model with a new input, it applies its learned parameters to produce an output that reflects what it ‘learned’ during training.

In the following subsections three relevant Image-classifying models are introduced. Each of these models where responsible for improvements in the accuracy of the Dataset challenges brought up in 2.4.

2.5.1 LeNet

LeNet refers to a family of early CNNs, the best-known variant being LeNet-5, which Yann LeCun et al. introduced in 1998 for handwritten digit recognition on the MNIST dataset [1]. The LeNet-5 architecture is illustrated in Figure 2.9. As

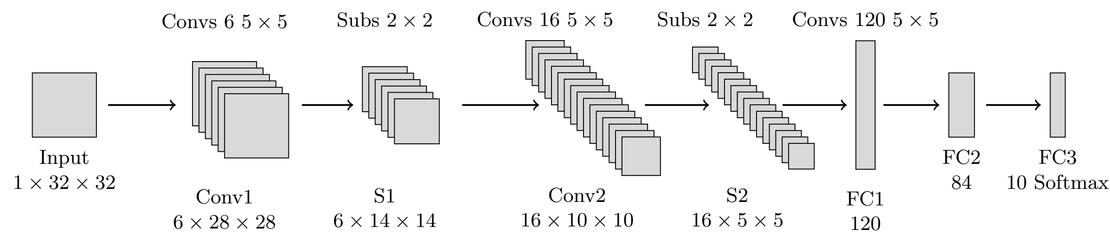


Figure 2.9: Overview of the LeNet-5 architecture.

seen in Figure 2.9 LeNet-5 accepts a single channel 32×32 image. Since MNIST images are 28×28, each digit is first padded by two zero pixels on every side to reach 32×32. The padded image then passes through alternating convolution and pooling layers. Conv1 first applies six 5×5 kernels that convolve the 32×32 input to produce a $6 \times 28 \times 28$ feature volume that detects low-level edges and strokes. S1 applies subsampling in the form of average-pooling over a 2×2 window with stride 2; effectively halving the spatial dimensions to $6 \times 14 \times 14$ and adding translation invariance. Conv2 uses sixteen 5×5 filters (stride 1) on the $6 \times 14 \times 14$ volume to yield a $16 \times 10 \times 10$ output to detect patterns of higher complexity. S2 then subsamples with average-pooling another 2×2 region (stride 2), shrinking the result to $16 \times 5 \times 5$. After the subsampling layer S2, the feature maps have shape $16 \times 5 \times 5$, to which 120 convolutional filters of size 5×5 (stride 1, no padding) are applied. Since each filter covers the 5x5 spatial grid, each one collapses to a single value of a $120 \times 1 \times 1$ tensor (a stack of 120 scalars).

Not shown in Figure 2.9 this tensor is then flattened into a 120-element vector so that it can be fed into ordinary (fully connected) layers. FC1 has 120 neurons, each neuron takes all 120 inputs, computes a weighted sum plus its bias, and then applies its activation function. The output is another 120-element vector, where each entry represents a higher-level combination of the features detected by the preceding convolutions. This is then fed into FC2 where the process of FC1 happens again, except with 84 neurons and without flattening, producing an 84-element vector. Finally, the previous 84-element vector is fed into FC3 where it goes into a fully connected layer of 10 neurons, one for each digit (0 through 9). The raw output of this is then passed into a softmax function; normalizing the neurons and converting it into a probability distribution. The class (digit) with the highest probability becomes the network's prediction.

2.5.2 AlexNet

Although the term AlexNet has no formally standardized specification, it is generally understood to denote the original 2012 architecture with five convolutional

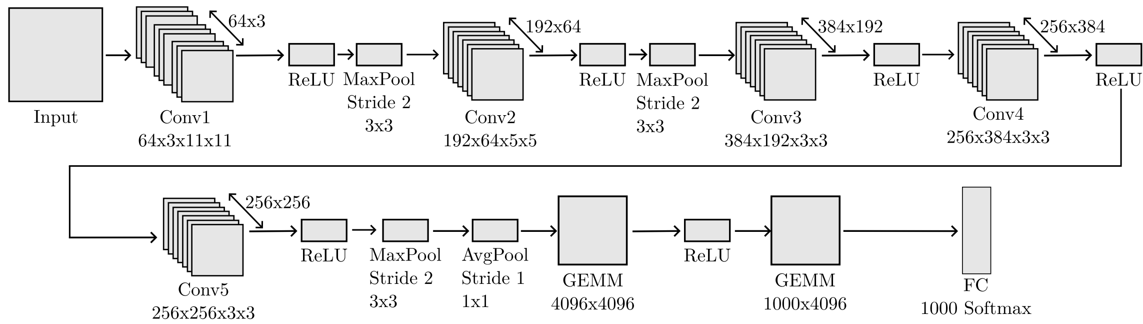


Figure 2.10: Overview of the AlexNet architecture

layers, three fully connected layers, and a final softmax classifier [3]. Figure 2.10 displays an overview of the AlexNet architecture. Instead of digits AlexNet processes images, identifying what is presented in them. The input dimensions of an image expected by an AlexNet model depend on the dataset used for training. For the 2012 ILSVRC configuration, each color image is first resized so that its shorter side is 256 pixels, then center-cropped to 224×224 pixels, converted to a three-channel (RGB) tensor, and finally normalized with the standard ImageNet statistics (channel means [0.485, 0.456, 0.406] and standard deviations [0.229, 0.224, 0.225]). On the output of the final layer, a softmax operation is usually performed. Which outputs a 1000-dimensional probability vector, whose highest scoring index corresponds to one of the 1000 ImageNet object categories.

2.5.3 ResNet-18

Although ResNet designates a family of residual networks of varying depth, ResNet-18 conventionally refers to the original 2015 model that begins with a 7×7 convolution and max-pool layer, followed by four residual stages, each containing two 3×3 convolutions with identity shortcuts, for a total of 18 weighted layers. A global average pooling and a single fully connected softmax layer complete the architecture [30]. An identity shortcut (or identity skip connection) simply adds the input of a residual block x to the output of its internal convolutional stack, yielding $y = F(x) + x$ [30]. An example of the type of identity shortcut used in ResNet-18 can be seen in Figure 2.11. A diagram of the full structure is provided in Appendix A.2 for reference. ResNet-18 uses the same ImageNet preprocessing pipeline already described for AlexNet, and its softmax output also maps one-to-one to the ImageNet object categories.

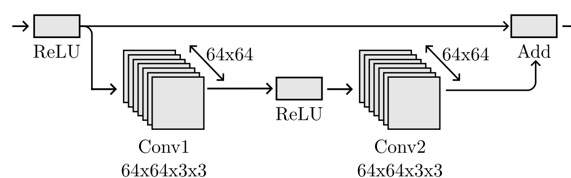


Figure 2.11: $y = F(ReLU(x)) + ReLU(x)$ identity shortcut

2.6 ONNX

The ONNX format was created to provide a standardized representation for AI models, solving the challenges developers often face when porting models between different environments, as each framework has its internal representation of models [31].

At its core, an ONNX model is a Directed Acyclic Graph (DAG) [32] in which nodes specify operators (e.g. Conv, Relu, or Gemm), where each node defines attributes, inputs, and outputs. These inputs and outputs link nodes to other nodes and also to initializers which define operator parameters like weights and biases. Through extensive documentation, ONNX provides specifications (such as attributes, input-output mappings, type-support, etc.) for all standard operators [33].

By implementing these operators, following the standardized specifications, frameworks can achieve interoperability between theirs and all other frameworks supporting ONNX. With ONNX, models trained in one framework (such as PyTorch [4] or TensorFlow [5]) can be exported to another, allowing researchers and engineers to compare and deploy models across different environments. This interoperability facilitates the integration of backend optimizations, allowing models to be effectively deployed on diverse hardware platforms. Applications requiring performance tuning, such as low-level profiling tools, benefit significantly from this [34]. A growing number of tools and libraries now support ONNX, which has promoted an ecosystem around model optimization, debugging, and runtime acceleration.

The file format of ONNX is based on Protocol Buffers (Protobuf) [13], a binary serialization format for structured data. Protobuf is commonly used in place of classes or structs, it is especially well-suited for defining data structures that need to be shared across many different programming languages [35].

3
Methods

The methodology for this project followed a structured approach beginning with an assessment of the framework's requirements, followed by research into existing C++ ML frameworks. The initial development was focused on creating a Minimum Viable Product (MVP). The framework was implemented in an iterative process with concurrent testing.

3.1 Requirements Analysis

Initially, the project was broken down into several requirements that needed to be fulfilled to define it as successful. These requirements were:

- Modularity: It should be easy and intuitive to replace the current implementation of computational functions like GEMM.
- Compatibility: ModularML should be compatible with (be able to parse) current methods of storing ML model blueprints, namely ONNX.
- Usability: ModularML should be able to parse ONNX blueprints of models Lenet and AlexNet and run them.

3.2 Research and Design

To get an understanding of the state-of-the-art within the ML field the group researched existing C++ ML frameworks (PyTorch [36], TensorFlow [37]), essential algorithms, and relevant optimization techniques. The goal here was to understand how the requirements could be implemented conventionally and efficiently.

This research accumulated in the decision to design the framework as illustrated in Figure 3.1.

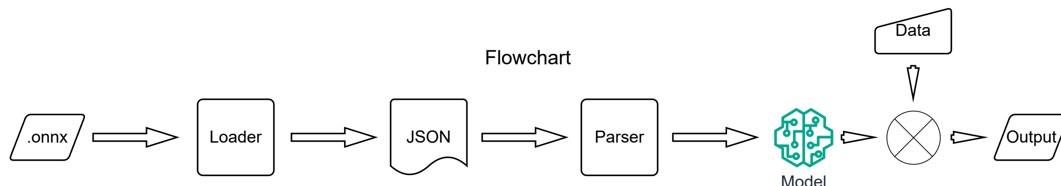


Figure 3.1: Flowchart of how the framework works.

3.3 Implementation

The framework was implemented iteratively using an agile-inspired workflow, starting with an MVP adhering to Figure 3.1.

- Loading models in ONNX format to create a local JavaScript Object Notation (JSON) file.
- Utilization of tensor data structures for computation.
- Flexibility allowing future development.

During development, the Scrum Master held group meetings in person at least once a week. Feature tickets (containing clear implementation instructions) were added to the Kanban board. Each new feature was implemented following test-driven development, ensuring that every feature added did what it was supposed to do and did so reliably. We continuously followed up on coverage reports during our weekly meetings to ensure that the software's integrity was maintained.

3.4 Parser Implementations

The intermediate step of converting the ONNX model into JSON before parsing it into the framework was made for several reasons. Tools already exist to convert between ONNX and JSON [38], leaving both viable. Since JSON is a text-based, Unicode-encoded format [39], defining models as JSON enables models to be inspected and adjusted in any text editor. JSON being human readable simplifies debugging and serves as a helpful guideline when defining and parsing models. Given JSON's advantage of readability outweighs the slight performance benefit of ONNX, models were parsed as JSON. As it was touched on in 1.4 ModularML does not provide an organic (created as a part of the framework) ONNX to JSON parser. This is instead provided by an external open source script [38].

3.5 GEMM Implementations

As discussed in Section 2.2, GEMM dominates the computational cost of running modern CNNs. To make large-scale networks practical, the framework has integrated alternative GEMM kernels in addition to a naive GEMM implementation. These include a Blocked, AVX2, AVX512, and an OpenBLAS implementation.

3.6 ONNX Standard and Documentation

Determining the functionality needed to enable the framework to parse ONNX models required studying the ONNX documentation and examining operator specifications (such as input-output mappings, attributes, etc.). To accomplish ONNX support, all the operations implemented in ModularML were made to follow their respective documentation on the ONNX website [33].

3.7 Standard Functions

Several different computational graph nodes reused code functions that perform mathematical operations on tensors. This reuse of code ensured modularity (changing implementation applies it everywhere), and robustness (utilizing code that has already been demonstrated to function effectively), and made the development process easier since functionality did not need to be remade. The best example of this is GEMM, but the same principle is also applied to some simpler operations such as tensor addition.

3.8 Testing and Evaluation

To minimize issues during the later stages of development, continuous testing and evaluation were conducted throughout all phases of the project. Upon reaching the MVP stage, a comprehensive assessment was carried out from both a functional and a performance perspective. The result of this evaluation informed the prioritization of subsequent performance enhancements.

3.8.1 Dataset Comparisons

To further evaluate and validate the ability of the framework to execute models correctly across a broader set of inputs, inputs from two dataset challenges were employed. The full MNIST dataset, described in Section 2.4.1, was used for LeNet, while 1000 images from the 2012 ILSVRC dataset, described in Section 2.4.2, were used for AlexNet and ResNet-18. For each input, the model's output was compared against the dataset's associated object class for said input. The results were then reported using two metrics, 'Top-1' accuracy, which measures the proportion of inputs where the model's most likely prediction matches the correct label, and 'Top-5' accuracy, which measures the proportion of inputs where the correct label appears among the five most likely predictions.

3.8.2 Model Loading and Execution

To validate that the network meets the criterion of supporting the loading and execution of any ONNX model for which the relevant nodes are implemented, a series of tests were conducted. These tests consisted of executing LeNet, AlexNet, and ResNet-18. The latter was not previously examined during the development process. Each model is loaded and executed using input data appropriate to their respective model types. After this, the output was evaluated as correct or incorrect for the given input.

3.8.3 Performance Evaluation

To examine the performance of the framework two benchmark programs using the ModularML framework were created. The first benchmark, the MNIST benchmark,

operates as follows: provided a LeNet model (JSON) the program will use the ModularML framework to parse and construct the internal representation of the model, then use the model to run inference on 100 MNIST test images. The second benchmark, the ImageNet benchmark, is similar to the MNIST benchmark, except it provides an AlexNet model and runs inference on 10 ImageNet validation images.

The command-line tool Performance Counters for Linux (perf) — a framework for analyzing hardware and software performance [40] — makes it possible to capture the runtime call stack of the benchmarks. Using the visualization tool Flamegraph [41], the data captured was used to create stack-based graphs illustrating the full runtime call-stack hierarchy. Analyzing call-stack runtime durations enabled more efficient identification of bottlenecks within the framework, thus allowing targeted optimizations.

3.8.4 Code Evaluation

Throughout the development of the project, a comprehensive suite of tests using the Google Test framework [42] was implemented to verify the functionality of each operation and as many individual functions as possible within the codebase. To assess the thoroughness of the testing, coverage reports were generated using the GNU coverage (gcov) [43] tool in combination with gcovr [44] (a graphical frontend for gcov), which provides a detailed overview of the portions of the code that were exercised by the tests and those that were not. This testing strategy ensured that ongoing modifications to the codebase did not introduce errors in the behavior of operations and functions.

3.8.5 Modularity and Transparency Evaluation

The resulting modularity of the framework was evaluated in two primary ways. First, by illustrating how easily a user can replace an existing implementation with one of their designs. Second, by showing how a user can select among existing implementations to configure the framework for optimal performance in their specific use case.

4

Results

This chapter will outline the results derived from the testing and evaluation procedures detailed in the methodology chapter. It will also outline the successive improvements made to ModularML's initial performance bottleneck: the GEMM function. Apart from this, the chapter will also showcase the final implementation's code coverage, modularity, and transparency.

4.1 Dataset Comparisons

Table 4.1 displays the Top-1 and the Top-5 accuracy for LeNet-5, AlexNet, and ResNet-18 models when used with their respective datasets, running on ModularML. Each model processed the entire input set in a single pass. No network reloads or reparses were required, and no memory leaks were detected, demonstrating the framework's ability to run continuously and handle arbitrarily large image streams. The large difference in accuracy between LeNet and AlexNet is primarily due to the difference in complexity between the two datasets. The expected LeNet-

Model	**Top-1**	**Top-5**	**Dataset**	**Inputs**
LeNet	97.62%	99.98%	MNIST	10 000
AlexNet	55.50%	79.60%	ILSVRC2012	1 000
ResNet-18	67.80%	87.70%	ILSVRC2012	1 000

Table 4.1: Dataset results

5 result for running the MNIST dataset is around 99.2% Top-1 [1] and 100% Top-5 accuracy. Running LeNet-5 on ModularML yielded 97.62% and 99.98% respectively.

The expected result for AlexNet running the full 2012 ILSVRC dataset is 62.5% Top-1 and 83.0% Top-5 accuracy [3]. Running it on ModularML yielded 55.50% and 79.60% respectively.

For ResNet-18 running on the same dataset, the expected results are 69.3% Top-1 and 89.2% Top-5 accuracy. Running it on ModularML yielded 67.80% and 87.70% respectively [30].

The discrepancies in the results achieved by our framework, PyTorch, and the articles are expected (Even running the models in PyTorch could not replicate the results from the articles exactly). Small differences in framework implementations

and weights are some examples of factors that aren't unimportant but too costly to fix for the scope of this project.

4.2 Model Loading and Execution

LeNet-5, AlexNet, and ResNet-18 were all successfully loaded from their respective JSON files, which in turn had been created by ONNX representations of the models. A portion of a JSON file successfully parsed by ModularML can be found in Appendix A.3. For LeNet-5 a handwritten digit (Figure 4.1) was preprocessed according to the process described in Section 2.5.1. For AlexNet and ResNet-18 an image of an American egret (Figure 4.2), an object class included among the 1000 possible ImageNet categories, was preprocessed according to the procedure described in Section 2.5.2 and used as input. The model outputs were subsequently processed following the methods described in the same sections, and evaluated to determine whether the predicted output correctly identified the digit or object present in the input image. LeNet-5 successfully classified the input digit as five, while both AlexNet and ResNet-18 correctly identified the American egret. This confirms that the parser instantiated every operation in the proper sequence and configuration. Moreover, ResNet-18 loaded and ran immediately, without previously having been a part of the development process, demonstrating that adherence to the ONNX operation specifications achieved a form of seamless import of different ONNX models on the framework.

Figure 4.1: Handwritten digit 5, used as input for LeNet-5 running on ModularML.

Figure 4.2: American (great) egret, used as the input image for AlexNet and ResNet-18 running on ModularML.

4.3 Performance Evaluation

Performance evaluation of the MVP indicated that the GEMM function accounted for the majority of the inference time. Consequently, alternative backend implementations were developed and tested to optimize GEMM performance.

4.3.1 Naive GEMM Backend

The default backend configuration uses a naive GEMM implementation. Profiling the benchmarks and analyzing the results as illustrated in Figure 4.3 showed that GEMM was a significant bottleneck, consuming 69.3% and 86.2% of the total runtime for the two respective benchmarks.

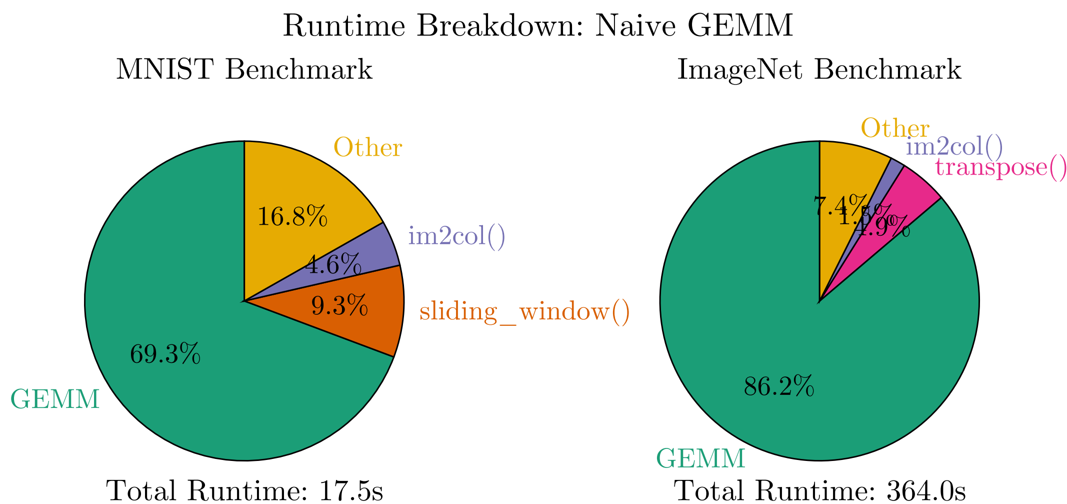


Figure 4.3: Pie charts illustrating the runtime breakdown using naive GEMM backend.

4.3.2 Blocked GEMM Backend

Introducing a backend configuration using a blocked GEMM implementation as explained in Section 2.3.1 showed an overall improvement in the runtime for both benchmarks, having decreased by 14% and 23% for MNIST and ImageNet benchmark respectively in comparison to the naive GEMM backend. Evidently due to less time spent on performing GEMM as illustrated by the runtime distribution in Figure 4.4. An increase in time for memory handling can be observed as well for the MNIST benchmark.

4.3.3 OpenBLAS GEMM Backend

Using a backend configuration leveraging an OpenBLAS GEMM implementation, explained in Section 2.3.3 showed a further performance improvement. The total runtime across the MNIST benchmark has decreased by 18% and 83% for the

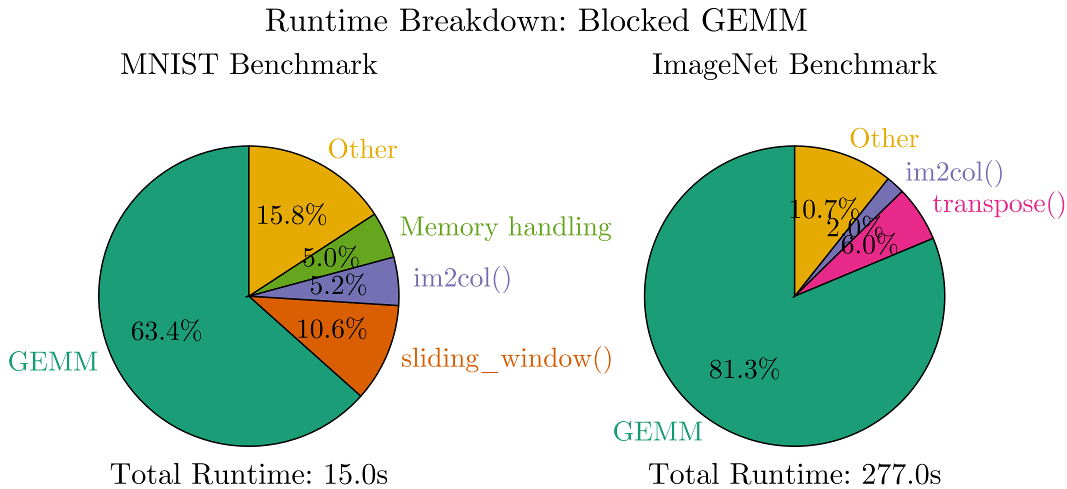


Figure 4.4: Pie charts illustrating the runtime breakdown when using the Blocked GEMM backend.

ImageNet benchmark in comparison to the original naive GEMM backend. As illustrated by Figure 4.5 The major discrepancy between the results, GEMM usage across both benchmarks decreased to 20% for MNIST and 17% for ImageNet.

This large discrepancy can be attributed to the nature of the benchmarks: ImageNet involves far more internal computations and larger matrix operations than in the MNIST benchmark, allowing the OpenBLAS backend to be more fully utilized. In contrast, the lighter computational load of the MNIST benchmark provides less room for such optimizations, leading to a more modest improvement.

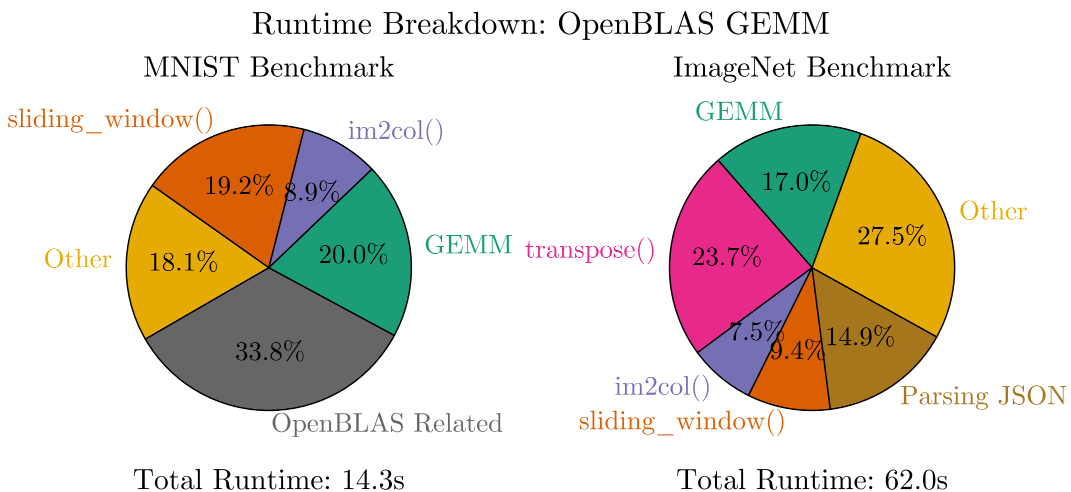


Figure 4.5: Pie charts illustrating the runtime breakdown when using the OpenBLAS GEMM backend.

4.3.4 AVX2 GEMM Backend

To leverage SIMD acceleration, an AVX2-based GEMM backend configuration was implemented, explained in Section 2.3.2. This provided a further decrease in the total runtime across both benchmarks, decreasing by 71% and 87% for the MNIST and ImageNet benchmarks respectively in comparison to the naive GEMM backend. As illustrated by Figure 4.6 the MNIST benchmark is mainly limited by the 'im2col' function performed within the CL and 'sliding window' function performed within the max-pooling layer, combined consuming 55% of the total 5-second runtime, while the ImageNet benchmark in addition to 'im2col' and 'sliding window' is limited by the transpose function. All three combined consume roughly 51% of the total 49-second runtime.

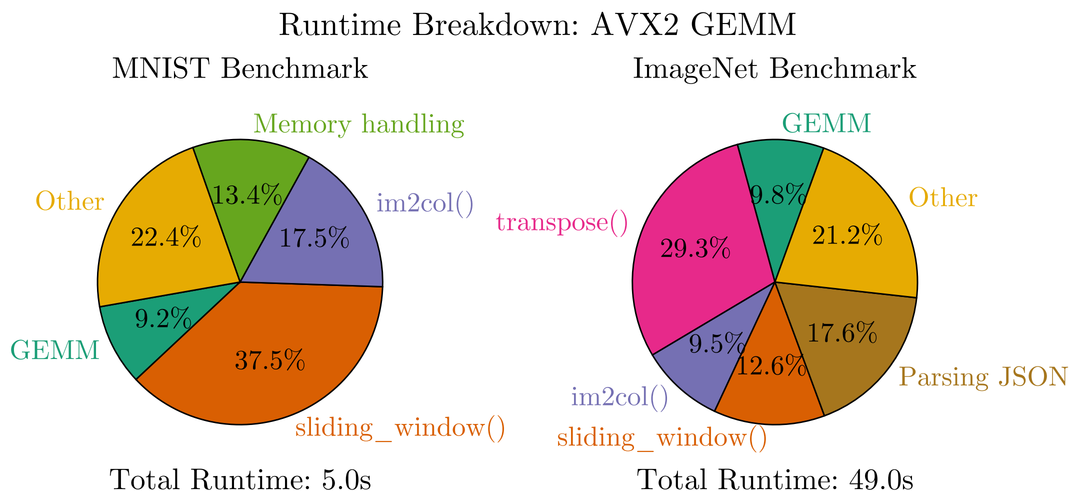


Figure 4.6: Pie charts illustrating the runtime breakdown when using the AVX2 GEMM backend.

4.3.5 AVX512 GEMM Backend

Leveraging SIMD acceleration further, an AVX512-based GEMM backend was implemented, also explained in Section 2.3.2. When profiled the benchmarks showed, as illustrated in Figure 4.7 that there was only a marginal improvement in the runtime for the ImageNet benchmark and an increase in the runtime for the MNIST benchmark. Both benchmarks spent an increased time on memory handling relative to the AVX2 backend, an area of the framework that can be improved upon to be able to fully leverage the AVX512 backend.

4.3.6 Overall GEMM Improvement

Comparing the values in Table 4.2 reveals a significant performance improvement achieved with the AVX512 implementation of GEMM. Specifically, an improvement of 90.3% was observed for the MNIST benchmark and 93.5% for the ImageNet benchmark. These results indicate that, after all the performance optimizations, the

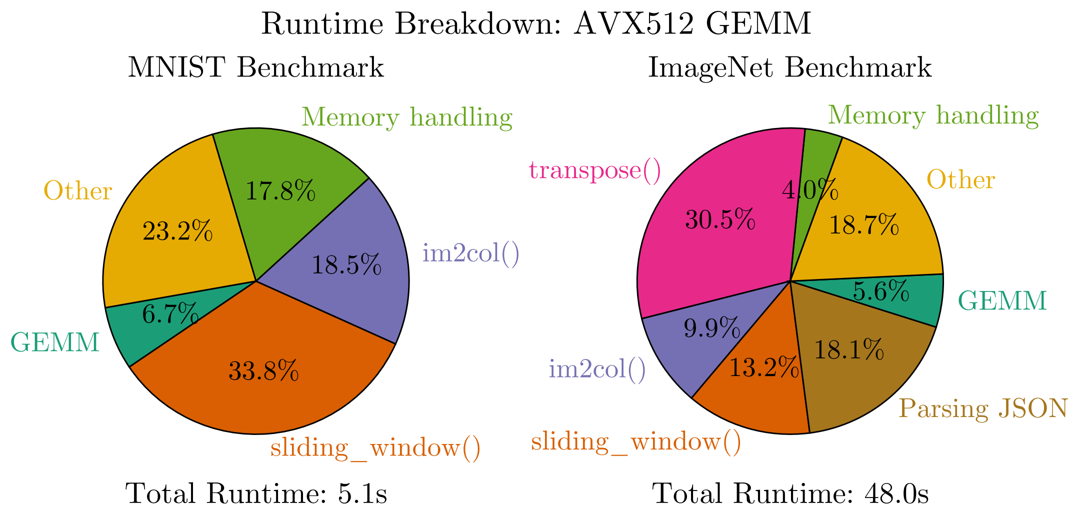


Figure 4.7: Pie charts illustrating the runtime breakdown when using the AVX512 GEMM backend.

AVX512 version provides substantial improvement. The initial GEMM bottleneck has been optimized to the point where it no longer accounts for a significant portion of the inference time.

Implementation	MNIST Improvement (%)	ImageNet Improvement (%)
Blocked GEMM	8.5	5.7
OpenBLAS GEMM	71.1	80.2
AVX2 GEMM	86.7	88.6
AVX512 GEMM	90.3	93.5

Table 4.2: GEMM improvements compared to naive implementation.

4.4 Test Coverage Results

Figure A.2 presents the gcovr test coverage report for the framework. The results indicate a high level of coverage, with 77.7% of all 336 functions being covered. The remaining uncovered functions primarily perform minor utility tasks that have not yet been included in the test suite. These functions are generally trivial and are not critical to the core functionality of the framework.

4.5 Modularity and Transparency

To substitute the default implementation of a source file, a user may create a directory representing a specific backend (e.g. `cuda/` or `avx/`). Within this directory, the user can add a custom source file using the same relative path and filename as the corresponding file in the default implementation directory. This is illustrated in Figure 4.8. After this, the user can pass the option `-DBUILD=...` with their implementation directory. During configuration, CMake will then detect the file and

configure the build system to compile it in place of the default.

For additional demands such as custom compiler options or toolkit requirements, the user may also include a CMakeLists.txt file within the same custom directory (as illustrated in Figure 4.8). If present, this file will be included during the configuration of this implementation. In turn, this allows the user to define build settings that might be needed, such as enabling specific toolkits (e.g. Compute Unified Device Architecture (CUDA) [45] toolkit) or enabling compiler options for specific hardware. This modularity means multiple implementations, optimized for specific hardware,

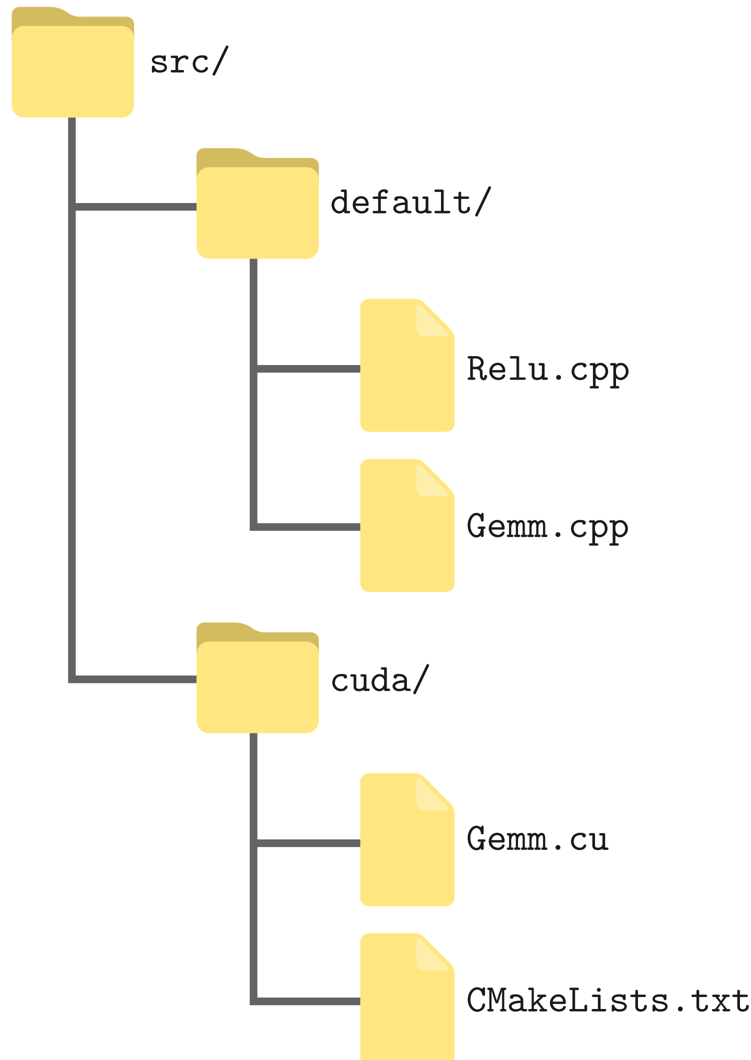


Figure 4.8: Example of multiple implementations in their respective directories.

can be present within the project at the same time. In turn, this enables users to build for their specific hardware, a user with hardware that supports AVX512 can build an optimized version for that hardware by using the CMake command-line option `-DBUILD=avx512`. If that same user also has a graphics card that supports the CUDA toolkit and instead wants to use its optimized backend then they can build again but with `-DBUILD=cuda`.

The framework provides algorithmic transparency by not utilizing any hidden API layers and by stating which implementations will be used for compilation during the build configuration. There are no limitations on using a debugger or profiler during any step of the execution.

5

Conclusion

This chapter will discuss the project about the achieved goals, performance, and data challenge results. It will also discuss lessons learned as well as what future work would benefit ModularML.

5.1 Achieved Goals

The overarching aim of ModularML — designing and implementing a transparent, modular C/C++ ML framework with fine-grained control — has been fully realized. All goals defined in Section 1.3 have been met as follows.

Goal **G.1** was fulfilled by constructing every ML model data structure and algorithm of the framework from first principles, yielding a coherent architecture that correctly executes ML models (see Sections 4.1 and 5.2). As stated in 1.4 we didn't have the time to implement training models and chose to implement only inference of pre-trained models. With the limitations of the project taken into consideration, we believe this goal has been achieved. Enabling users to train models in ModularML will be left for future work.

Goal **G.2** was satisfied through the ability to load a model into our dynamic C++ data structures during runtime and execute it an arbitrary amount of times. This means that static C++ implementations of different models are not required.

Goal **G.3** has been addressed by the multiple interchangeable implementations of critical routines. Among these routines is GEMM, whose multiple implementations can be viewed in Section 4.3. Apart from already existing implementations to choose from, as outlined in Section 4.5 the resulting framework also makes it easy for users to create their implementations.

Goal **G.4** was achieved through the framework's ability to parse a JSON file that had been created from an ONNX file. In turn permitting seamless import and inference of externally defined models, as demonstrated in Section 4.2, where three different ONNX models and their weights were loaded and executed successfully. We believe that the choice to use an ONNX parser to turn into JSON was crucial for this project to be possible. As with the case of training, making an organic parser would be too costly of an endeavor. Especially when there is working open-source software available.

Goal **G.5** was realized by maintaining a pure C++ codebase that exposes the entire code to the user, thereby ensuring a complete overview of what computational steps are performed when debugging.

5.2 Data Challenge Results

The outcomes from the two data challenges, as summarized in Table 4.1, are critical for evaluating and validating model execution within ModularML. The results obtained are consistently close to the reference results for each respective model, demonstrating that ModularML executes models reliably and reflects the expected behavior. As mentioned in 2.4 there are small discrepancies in the results but these are to be expected.

5.3 Performance

The primary goal of the project was to create a framework focused on modularity and transparency, with performance considered secondary. However, profiling revealed that GEMM consumed up to 86.2% of the total runtime, which was deemed unsatisfactory, leading to targeted optimization efforts for that specific functionality. By evaluating successive GEMM implementations of blocked, OpenBLAS, AVX2, and finally, AVX512 major performance improvements were made. The ImageNet benchmark reduced its runtime from 364 seconds to only 48 seconds with the AVX512 implementation. These results demonstrate not only that a modular design approach does not inherently introduce significant performance overhead but also that it allows for easy optimization and profiling.

Overall, ModularML achieves reasonable performance as a result of utilizing the additional backend configurations. With additional improvements to caching strategies and the utilization of even more optimized backend configurations, the framework could be a viable tool for experimenting with even larger models that would otherwise take too long to process.

5.4 Lessons Learned

From the outset, learning was a central objective of this project. By adopting an approach of building all components from the ground up, the project provided valuable insights, not only into ML concepts but also within C++ programming.

For several contributors, this was their first experience developing an organized software project at such a scale. This led to a number of organizational programming concepts having to be explored for the first time. Among these concepts was the structured testing that took place as well as the heavy utilization of GitHub [46] features, such as the Kanban board, as well as many Git features.

While there was initially only a superficial understanding of CNNs, the process of implementing and testing them led to a significantly deeper comprehension. In addition, exposure to performance-oriented technologies such as BLAS libraries and SIMD contributed to a broader understanding of low-level optimization in a context where it was necessary, even on modern computer hardware.

Overall, the project served as a comprehensive learning experience in both theoretical and practical aspects of ML design. It also served as a gateway to further one's understanding even deeper within this particular domain of ML, providing a good basis for a low-level understanding of current highly relevant LLMs.

5.5 Further Knowledge and Future Work

The ONNX standard currently defines approximately 180 unique operators [33]. To support full compatibility with the ONNX ecosystem, ModularML would ultimately need to implement the complete set of these operators. The process of adding new operators follows the same structure established during this project, and extending support should remain an ongoing objective. This task can be pursued in parallel with continued performance optimization.

If sufficient performance improvements are achieved, the training of models could also be explored, at least to an extent adequate for educational purposes. Such capability could also build highly upon the ONNX standard.

Future work could emphasize optimizations to a greater extent by implementing hardware acceleration or BLAS-based versions of computationally intensive operations that currently lack such backends. Doing so would not only shorten the runtime for even larger models but would also offer valuable opportunities to explore advanced optimization techniques.

Another potential improvement for the future is the ability to load ONNX files directly, without requiring an intermediate JSON conversion step. While this would streamline the model loading process, the JSON-based workflow should still be retained as an optional feature, since it offers users a convenient way to inspect and modify models using any standard text editor.

Making it possible to train models within the framework is also a very viable route of expansion for the Framework. This would in a sense make it "full-fledged" like its peers PyTorch and TensorFlow.

Lastly, extending ModularML to support model types beyond image classification, such as LLMs or other non-vision architectures, would constitute a logical next step. This would broaden the framework's applicability and further demonstrate its general-purpose capability as an ML inference platform. In turn, this would also allow for a deeper knowledge of current highly relevant ML LLMs.

Bibliography

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[2] ImageNet, *Imagenet: About*, https://www.image-net.org/about.php, Accessed: 2025-05-18, 2025.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, Describes the original 5-conv / 3-FC AlexNet architecture, vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[4] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, Vancouver, Canada, 2019, pp. 3–5.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi, Savannah, GA, USA, Nov. 2016.

[6] ICL-UTK, *Papi - performance api*, GitHub repository, 2024. [Online]. Available: https://github.com/icl-utk-edu/papi.

[7] RRZE-HPC, *Likwid - performance tools for linux*, GitHub repository, 2024. [Online]. Available: https://github.com/RRZE-HPC/likwid.

[8] gem5 Community, *Gem5 - computer architecture simulator*, GitHub repository, 2024. [Online]. Available: https://github.com/gem5/gem5.

[9] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, Long Beach, CA, USA, 2017.

[10] OpenAI, *Chatgpt*, [Online]. Available: https://openai.com/index/chatgpt/, 2022.

[11] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[12] M. R. E. aoun, L. N. Tidjon, B. Rombaut, F. Khomh, and A. E. Hassan, *An empirical study of library usage and dependency in deep learning frameworks*,

2022. arXiv: 2211.15733 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2211.15733.

[13] ONNX, *Introduction to onnx*, https://onnx.ai/onnx/intro/index.html, [Online; Accessed: Jan. 29, 2025], 2025.

[14] M. Findlay and J. Seah, "An ecosystem approach to ethical ai and data use: Experimental reflections," in *Proceedings of the 2020 IEEE/ITU International Conference on Artificial Intelligence for Good (AI4G)*, https://doi.org/10.1109/AI4G50087.2020.9311069, 2020, pp. 192–197. DOI: 10.1109/AI4G50087.2020.9311069.

[15] J. Martins, R. Barbosa, N. Lourenço, J. Robin, and H. Madeira, "Online verification through model checking of medical critical intelligent systems," in *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, https://doi.org/10.1109/DSN-W50199.2020.00015, 2020, pp. 32–37. DOI: 10.1109/DSN-W50199.2020.00015.

[16] A. Jobin, M. Ienca, and E. Vayena, "The global landscape of ai ethics guidelines," *Nature Machine Intelligence*, vol. 1, pp. 389–399, 2019. DOI: 10.1038/s42256-019-0088-2. [Online]. Available: https://doi.org/10.1038/s42256-019-0088-2.

[17] S. Larsson and F. Heintz, "Transparency in artificial intelligence," *Internet Policy Review*, vol. 9, no. 2, 2020. DOI: 10.14763/2020.2.1469. [Online]. Available: https://policyreview.info/concepts/transparency-artificial-intelligence.

[18] ScienceDirect, *Deep neural network*, [Online]. Available: https://www.sciencedirect.com/topics/computer-science/deep-neural-network.

[19] S. Wang, Q. Yao, Y. Zhang, and Y.-S. Wei, "Bias also matters: Bias attribution for deep neural network explanation," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, vol. 97, PMLR, 2019, pp. 6638–6647. [Online]. Available: https://proceedings.mlr.press/v97/wang19p.html.

[20] ImageNet Team, *Imagenet dataset download*, https://www.image-net.org/download.php, Accessed: 2025-05-14, 2025.

[21] S. direct, *Lateral inhibition*, https://www.sciencedirect.com/topics/neuroscience/lateral-inhibition, Accessed: 2025-05-02.

[22] P. Kashyap. "Image normalization in pytorch: From tensor conversion to scaling." Accessed: 2025-05-02. (2023), [Online]. Available: https://medium.com/@piyushkashyap045/image-normalization-in-pytorch-from-tensor-conversion-to-scaling-3951b6337bc8.

[23] Intel, *Intrinsics*, https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics.html, Version 2021.10, accessed May 2025, 2023.

[24] ARM, *Neon programmer guide for armv8-a coding for neon*, https://developer.arm.com/documentation/102159/latest/, Version 4.0, accessed May 2025, 2023.

[25] L. S. Blackford, A. Petitet, R. Pozo, *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[26] OpenBLAS Development Team, *OpenBLAS: An optimized blas library*, https://www.openblas.net/, Version 0.3.x, accessed 30 Apr 2025, 2025.

[27] Intel Corporation, *Intel® oneapi math kernel library*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, Version 2024.0, accessed 30 Apr 2025, 2024.

[28] NVIDIA Corporation, *Cublas library user's guide*, https://docs.nvidia.com/cuda/cublas/, Version 12.1, accessed 30 Apr 2025, 2024.

[29] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[31] T. L. Foundation, *Onnx: Open neural network exchange*, https://onnx.ai/about.html, Accessed: 2025-04-03, 2025.

[32] Wikipedia contributors. "Directed acyclic graph — wikipedia, the free encyclopedia." Accessed: 2024-04-8. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Directed_acyclic_graph.

[33] T. L. Foundation, *ONNX Operators (version 1.18)*, https://onnx.ai/onnx/operators/, Accessed: 2025-04-04.

[34] onnx, *End-to-end ai for nvidia-based pcs: Transitioning ai models with onnx*, GitHub repository, 2025. [Online]. Available: https://github.com/onnx/optimizer.

[35] Wikipedia contributors. "Protocol buffers — wikipedia, the free encyclopedia." Accessed: 2025-05-08. (2025), [Online]. Available: https://en.wikipedia.org/wiki/Protocol_Buffers.

[36] PyTorch, *Pytorch/pytorch*, https://github.com/pytorch/pytorch, [Online; Accessed: Feb. 6, 2025], 2025.

[37] TensorFlow, *Tensorflow/tensorflow*, https://github.com/tensorflow/tensorflow, [Online; Accessed: Jan. 29, 2025], 2025.

[38] PINTO0309, *Onnx2json: Convert onnx model files to json*, https://github.com/PINTO0309/onnx2json, [Online; Accessed: 2025-05-08], 2023.

[39] Wikipedia contributors. "Json — wikipedia, the free encyclopedia." Accessed: 2025-05-08. (2025), [Online]. Available: https://en.wikipedia.org/wiki/JSON.

[40] Linux Kernel Organization, *Perf - performance analysis tools for linux*, Accessed: 2025-05-01, 2024. [Online]. Available: https://www.man7.org/linux/man-pages/man1/perf.1.html.

[41] B. Gregg, *Flamegraph*, https://www.brendangregg.com/flamegraphs.html, Accessed: 2025-05-01, 2014.

[42] Google, *Googletest - google testing and mocking framework*, https://github.com/google/googletest, Accessed: 2025-04-28, 2025.

[43] G. Project, *Gcov: A test coverage program*, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, Accessed: 2025-04-28, 2025.

[44] Gcovr, *Gcovr: Code coverage report generator for c/c++*, https://gcovr.com/en/stable/, Accessed: May 2, 2025.

[45] NVIDIA Corporation, *Cuda c programming guide*, Accessed: Feb. 03, 2025, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[46] GitHub, Inc., *GitHub: A platform for version control and collaboration*, Accessed: 2025-05-15, 2024. [Online]. Available: https://github.com.

A

Appendix 1

This appendix provides supplementary material referenced in the main body of the report.

A.1 Project Source Code

The source code, documentation, and installation guidance can all be found on GitHub (Using this link)

A.2 ResNet-18 Architecture

Figure A.1 displays the ResNet-18 architecture and is intended to support the theory description in Section 2.5.3.

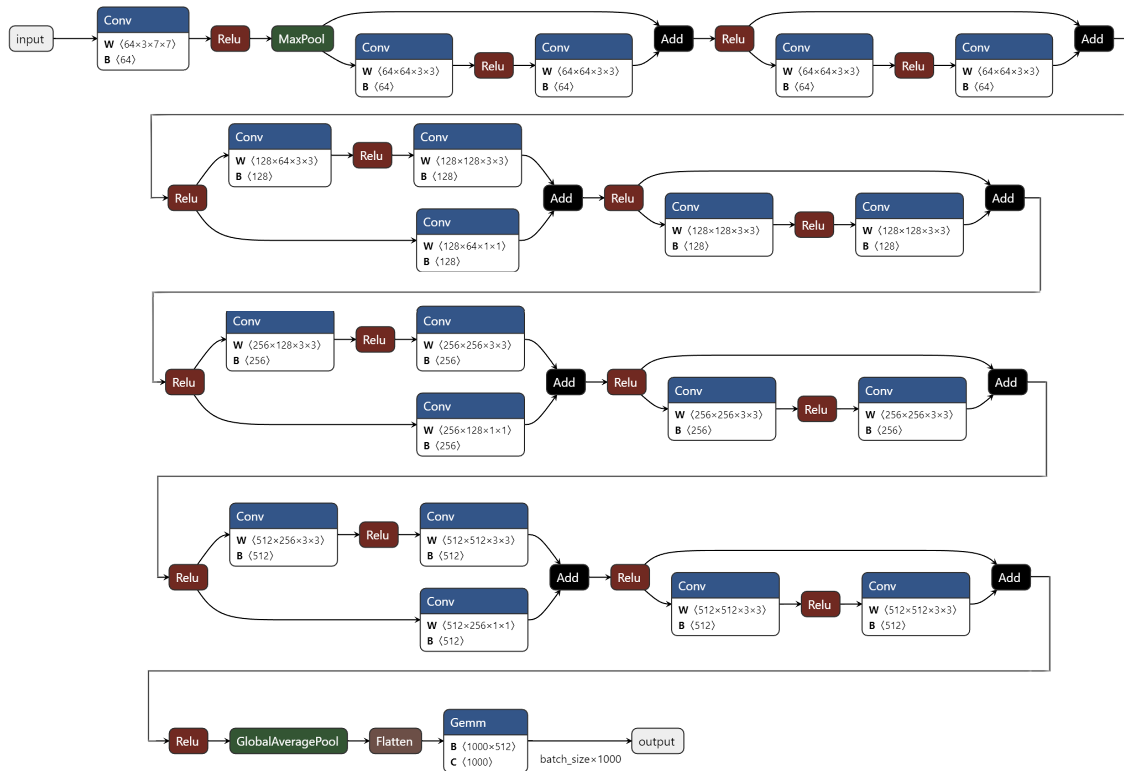


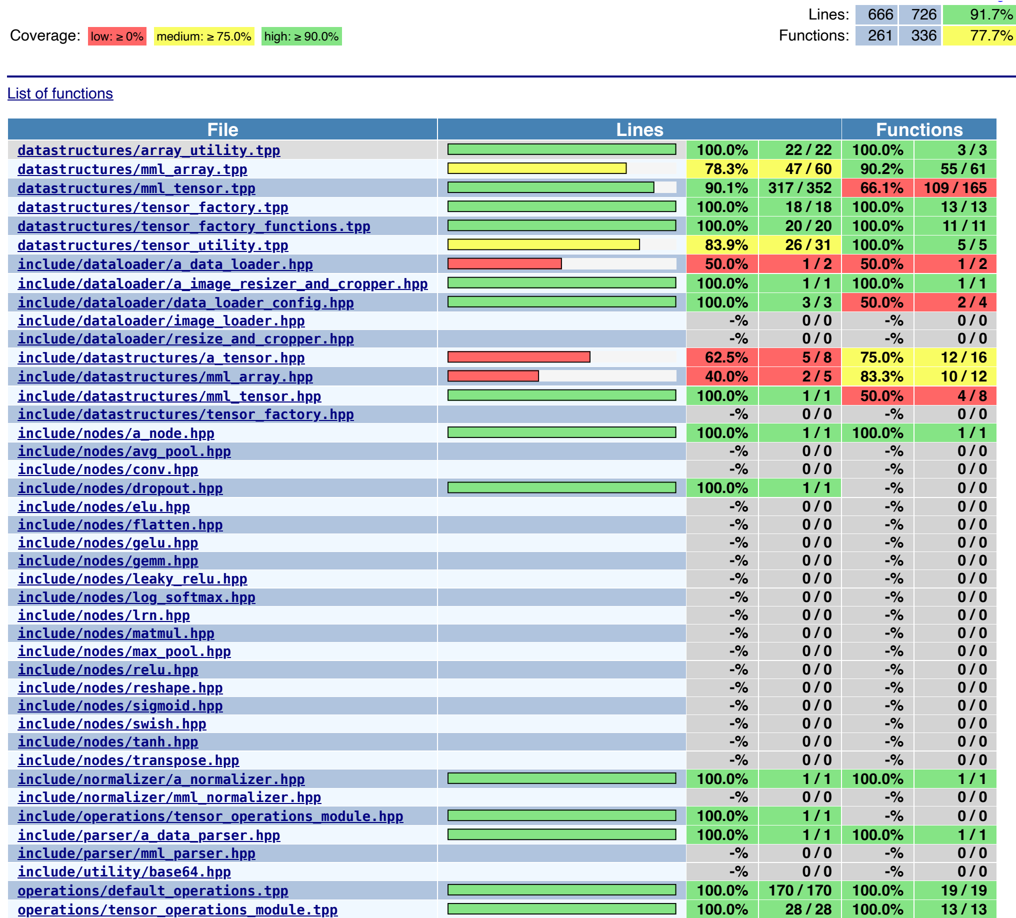
Figure A.1: ResNet-18 Architecture

A.3 First Convolution in LeNet, outlined in JSON parsed by ModularML

Listing A.1 displays how the first convolutional layer in the LeNet model appears in JSON form and is intended to support the results presented in Section 4.2.

```
1      "node": [
2        {
3          "name": "Conv1",
4          "opType": "Conv",
5          "input": ["input", "base_model.conv1.weight"],
6          "output": ["Conv1_out"],
7          "attribute": [
8            {"name": "dilations", "ints": ["1", "1"], "type": "INTS" },
9            {"name": "group", "i": "1", "type": "INT" },
10           {"name": "kernel_shape", "ints": ["5", "5"], "type": "INTS" },
11           {"name": "pads", "ints": ["2", "2", "2", "2"], "type": "INTS" },
12           {"name": "strides", "ints": ["1", "1"], "type": "INTS" }
13         ]
14       }
```

Listing A.1: First Convolution in LeNet, outlined in JSON



Lines:	666	726	91.7%
Functions:	261	336	77.7%

Coverage: low: ≥ 0% medium: ≥ 75.0% high: ≥ 90.0%

List of functions

File	Lines		Functions	
datastructures/array_utility.tpp	100.0%	22 / 22	100.0%	3 / 3
datastructures/mml_array.tpp	78.3%	47 / 60	90.2%	55 / 61
datastructures/mml_tensor.tpp	90.1%	317 / 352	66.1%	109 / 165
datastructures/tensor_factory.tpp	100.0%	18 / 18	100.0%	13 / 13
datastructures/tensor_factory_functions.tpp	100.0%	20 / 20	100.0%	11 / 11
datastructures/tensor_utility.tpp	83.9%	26 / 31	100.0%	5 / 5
include/dataloader/a_data_loader.hpp	50.0%	1 / 2	50.0%	1 / 2
include/dataloader/a_image_resizer_and_cropper.hpp	100.0%	1 / 1	100.0%	1 / 1
include/dataloader/data_loader_config.hpp	100.0%	3 / 3	50.0%	2 / 4
include/dataloader/image_loader.hpp	-%	0 / 0	-%	0 / 0
include/dataloader/resize_and_cropper.hpp	-%	0 / 0	-%	0 / 0
include/datastructures/a_tensor.hpp	62.5%	5 / 8	75.0%	12 / 16
include/datastructures/mml_array.hpp	40.0%	2 / 5	83.3%	10 / 12
include/datastructures/mml_tensor.hpp	100.0%	1 / 1	50.0%	4 / 8
include/datastructures/tensor_factory.hpp	-%	0 / 0	-%	0 / 0
include/nodes/a_node.hpp	100.0%	1 / 1	100.0%	1 / 1
include/nodes/avg_pool.hpp	-%	0 / 0	-%	0 / 0
include/nodes/conv.hpp	-%	0 / 0	-%	0 / 0
include/nodes/dropout.hpp	100.0%	1 / 1	-%	0 / 0
include/nodes/elu.hpp	-%	0 / 0	-%	0 / 0
include/nodes/flatten.hpp	-%	0 / 0	-%	0 / 0
include/nodes/gelu.hpp	-%	0 / 0	-%	0 / 0
include/nodes/gemm.hpp	-%	0 / 0	-%	0 / 0
include/nodes/leaky_relu.hpp	-%	0 / 0	-%	0 / 0
include/nodes/log_softmax.hpp	-%	0 / 0	-%	0 / 0
include/nodes/lrn.hpp	-%	0 / 0	-%	0 / 0
include/nodes/matmul.hpp	-%	0 / 0	-%	0 / 0
include/nodes/max_pool.hpp	-%	0 / 0	-%	0 / 0
include/nodes/relu.hpp	-%	0 / 0	-%	0 / 0
include/nodes/reshape.hpp	-%	0 / 0	-%	0 / 0
include/nodes/sigmoid.hpp	-%	0 / 0	-%	0 / 0
include/nodes/swish.hpp	-%	0 / 0	-%	0 / 0
include/nodes/tanh.hpp	-%	0 / 0	-%	0 / 0
include/nodes/transpose.hpp	-%	0 / 0	-%	0 / 0
include/normalizer/a_normalizer.hpp	100.0%	1 / 1	100.0%	1 / 1
include/normalizer/mml_normalizer.hpp	-%	0 / 0	-%	0 / 0
include/operations/tensor_operations_module.hpp	100.0%	1 / 1	-%	0 / 0
include/parser/a_data_parser.hpp	100.0%	1 / 1	100.0%	1 / 1
include/parser/mml_parser.hpp	-%	0 / 0	-%	0 / 0
include/utility/base64.hpp	-%	0 / 0	-%	0 / 0
operations/default_operations.tpp	100.0%	170 / 170	100.0%	19 / 19
operations/tensor_operations_module.tpp	100.0%	28 / 28	100.0%	13 / 13

Figure A.2: Resulting coverage report of the framework.