MACHINE LEARNING FOR MOBILE HEALTHCARE

by

Chiranjivan Krishnakumar Nirmala
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

______________________	Dr. Weiwen Jiang, Thesis Director
______________________	Dr. Sai Manoj PD, Committee Member
______________________	Dr. Tolga Soyata, Committee Member
______________________	Dr. Brian L. Mark, Department Head
Date: ______________________	Spring Semester 2024 George Mason University Fairfax, VA

Machine Learning for Mobile Healthcare

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Chiranjivan Krishnakumar Nirmala
Bachelor of Engineering, Electronics and Communication Engineering
Sri Sairam Engineering College, 2019

Director: Dr. Weiwen Jiang, Professor
Department of Electrical and Computer Engineering

Spring Semester 2024
George Mason University
Fairfax, VA

Copyright © 2024 by Chiranjivan Krishnakumar Nirmala
All Rights Reserved

Dedication

I dedicate this dissertation to my parents, Krishnakumar C and Nirmala T, for their constant motivation and support. I would also like to thank my friends for their unwavering support and encouragement.

Acknowledgments

I would like to thank the following people who made this possible. I wish to express my profound appreciation to Dr. Weiwen Jiang, my advisor during my master's studies, for his consistent encouragement and support whenever I encountered obstacles that hindered the progress of my research. His consistent guidance and counsel have been of great assistance in ensuring the success of this thesis.

Table of Contents

Page

List of Tables . . . vii

List of Figures . . . viii

Abstract . . . ix

1 Introduction . . . 1

1.1 AI in Healthcare . . . 1

1.1.1 Machine Learning . . . 2

1.1.2 Deep Learning . . . 7

1.1.3 Difference between Machine Learning and Deep Learning . . . 9

1.1.4 Deep Learning model types and applications . . . 11

1.2 Motivation . . . 13

1.3 Flow of thesis . . . 15

2 Edge Computing and Fairness in AI . . . 16

2.1 Rise of Edge Computing . . . 16

2.2 Efficient CNNs for Mobile & Edge Devices . . . 17

2.2.1 SqueezeNet . . . 17

2.2.2 MobileNet . . . 19

2.2.3 MnasNet . . . 21

2.2.4 FaHaNa . . . 23

2.3 Comparison of NAS Frameworks . . . 25

2.4 Importance of Fairness in Dermatology AI . . . 27

2.5 Challenges in Existing AI Systems . . . 28

2.6 Contribution . . . 29

3 Development of the Android Application . . . 31

3.1 Why Android? . . . 31

3.2 Fundamentals of Android Application Architecture . . . 31

3.2.1 Android General Concepts . . . 32

3.2.2 Lifecycle Management . . . 33

3.2.3 Lifecycle callbacks . . . 33

3.2.4 Essential Files and Configuration . . . 34
3.3 Dynamic Interaction in Activity_main . . . 35
3.4 Camera Functionality in activity_camnew . . . 36
3.4.1 Image Processing Techniques in module_imageprocessing . . . 39
3.4.2 Configuration Management in module_param . . . 40
3.4.3 Integration of PyTorch Models in Mobile Applications . . . 41
3.4.4 Displaying Results in activity_result . . . 43
3.5 Implementing User Preferences in activity_usr_pref . . . 44
3.5.1 Data Mapping and Error Handling in module_mapping . . . 45
3.6 Design for Software Information in activity_abt_software . . . 45
4 Experimental Results . . . 47
4.1 Experimental Setup . . . 47
4.1.1 Dataset . . . 47
4.1.2 FaHaNa Settings . . . 47
4.1.3 Edge Devices . . . 48
4.2 Workflow . . . 49
4.3 Performance Evaluation . . . 53
4.4 Performance Comparison . . . 56
5 Conclusion . . . 57
5.1 Conclusion . . . 57
5.2 Future Works . . . 58
Bibliography . . . 59

List of Tables

Table Page

1.1 Difference between Machine Learning and Deep Learning 10

4.1 Comparison of the existing models and FaHaNa-Models 54

4.2 Size comparison of ML Models 55

List of Figures

Figure		Page
1.1	Supervised Learning	3
1.2	Unsupervised Learning	4
1.3	Semi-supervised Learning	5
1.4	Reinforcement Learning	6
1.5	Perceptron	7
1.6	Shallow Neural Network	8
1.7	Deep Neural Network	9
2.1	Fire Module	18
2.2	MobileNetV3 figure	20
2.3	MnasNet Architecture	22
2.4	FaHaNa Framework Figure	24
3.1	Activity Lifecycle	32
3.2	Android App Architecture	35
4.1	Conversion to ptl	49
4.2	Structure of FaHaNa-Small	50
4.3	Screenshot of Android Application	51
4.4	Screenshot of Diagnosis Result	52
4.5	Screenshot of User Preference	52
4.6	Screenshot of About Software	53

Abstract

MACHINE LEARNING FOR MOBILE HEALTHCARE

Chiranjivan Krishnakumar Nirmala, MS

George Mason University, 2024

Thesis Director: Dr. Weiwen Jiang

As AI becomes increasingly ubiquitous across industries, there is a growing demand for ML models to be deployed on edge devices, driven by the democratization of AI. However, the decision-making processes of AI systems often exhibit biases, prompting a renewed focus on fairness, particularly in industries prioritizing equitable outcomes such as security surveillance, face recognition, and medical applications like dermatology. This research addresses the need for fairness in mobile healthcare, specifically in dermatology, by developing an Android application for skin disease detection and mobile dermatology assistance in remote areas.

While existing AI systems boast high overall accuracies, they often neglect fairness considerations, resulting in subpar performance, especially on datasets representing diverse skin tones. Despite the importance of fairness, most neural network architectures prioritize other metrics, disregarding the need for models to run efficiently on edge devices. To bridge this gap, there is a call for smaller networks optimized for hardware constraints, without compromising fairness.

This study explores the paper "The Larger The Fairer? Small Neural Networks Can Achieve", presented at the Design Automation Conference – 2022. Which introduces an

automatic neural architecture search (NAS) methodology called as Fairness and Hardware-aware Neural architecture search (FaHaNa) for network selection. FaHaNa employs a freezing method to accelerate optimization while preserving fairness, effectively minimizing network size and latency for edge devices.

The thesis discusses about the successful application of the FaHaNa framework on Android devices illustrates its potential to democratize healthcare diagnostics across diverse demographic and geographic landscapes, making advanced healthcare solutions more accessible and reducing disparities in medical care availability.

This work not only showcases the feasibility of achieving fairness in mobile healthcare applications but also sets a solid foundation for future innovations in the domain of equitable, AI-enabled healthcare solutions.

Chapter 1: Introduction

1.1 AI in Healthcare

Artificial intelligence (AI) has emerged as a transformative force in healthcare, revolutionizing various aspects of medical diagnosis, treatment, and patient care. With its ability to analyze vast amounts of data, identify patterns, and make predictions, AI holds the promise of enhancing clinical decision-making, improving patient outcomes, and optimizing healthcare delivery.

In recent years, AI technologies have been increasingly integrated into healthcare systems worldwide, offering innovative solutions to longstanding challenges. One of the key areas where AI has made significant strides is in medical imaging interpretation. AI algorithms trained on large datasets of medical images can assist radiologists in detecting abnormalities, such as tumors or fractures, with greater accuracy and efficiency than traditional methods.

Furthermore, AI-powered predictive analytics tools are being used to forecast disease progression, identify patients at risk of developing certain conditions, and personalize treatment plans. By analyzing electronic health records (EHRs), genetic data, and other patient information, AI algorithms can help clinicians make more informed decisions tailored to individual patient needs.

In addition to clinical applications, AI is also transforming healthcare operations and administrative tasks. From streamlining appointment scheduling and optimizing resource allocation to automating billing and coding processes, AI-driven solutions are improving efficiency and reducing administrative burdens on healthcare providers.

Moreover, AI-powered virtual assistants and chatbots are enhancing patient engagement and access to care by providing round-the-clock support, answering medical questions, and delivering personalized health recommendations.

Despite its immense potential, the widespread adoption of AI in healthcare is not without challenges. Ethical considerations surrounding data privacy, algorithm bias, and the potential for job displacement are among the issues that need to be carefully addressed. Furthermore, the complexity of healthcare systems and the need for regulatory compliance pose additional hurdles to the seamless integration of AI technologies.

Nevertheless, the ongoing advancements in AI research and technology hold promise for transforming healthcare delivery, making it more efficient, equitable, and patient-centered. As we continue to harness the power of AI in healthcare, it is imperative to ensure that these technologies are developed and deployed in a responsible and ethically sound manner, with a focus on improving patient outcomes and advancing the delivery of quality care.

1.1.1 Machine Learning

Machine Learning (ML) is an integral component of Artificial Intelligence (AI), focusing on leveraging data and algorithms to enable AI systems to learn and improve over time, close to the way humans learn. By iteratively analyzing data patterns and adjusting its predictions, a machine learning system gradually enhances its accuracy and effectiveness. Machine learning system or algorithm can be broken into three main components:

1. A Decision Process: Commonly, machine learning algorithms are aimed to work as a prediction or classification system based on input data. Whether the data is labeled (supervised learning) or unlabeled (unsupervised learning), the ML algorithm will produce an estimate about a pattern within the dataset to generate meaningful insights or predictions.

2. An Error Function: An error function evaluates the prediction of the model. By comparing the model's predictions against known examples or ground truth data, the error function quantifies the discrepancy or error between the predicted outcomes and the actual observations. This assessment allows the algorithm to gauge its accuracy.

3. A Model Optimization Process: To fit the model better to the data points in the training set, the weights are adjusted using technique such as gradient descent, thus reducing the discrepancy between the known example and the model estimate. The iterative "evaluate and optimize" process lies at the heart of machine learning, driving continuous improvement and adaptation of the model to evolving datasets and scenarios. As the algorithm learns from additional data and refines its predictions, it becomes increasingly adept at capturing complex patterns and making accurate decisions.

Further, ML algorithms encompass a diverse range of methodologies, each tailored to address specific types of learning tasks and data structures. Broadly categorized, ML algorithms can be classified into the following types:

1. Supervised machine learning : Supervised learning, constitutes a fundamental approach in machine learning where algorithms are trained using labeled datasets. In this paradigm, each data point in the training set is associated with a corresponding label or outcome, allowing the algorithm to learn the mapping between input features and target variables.

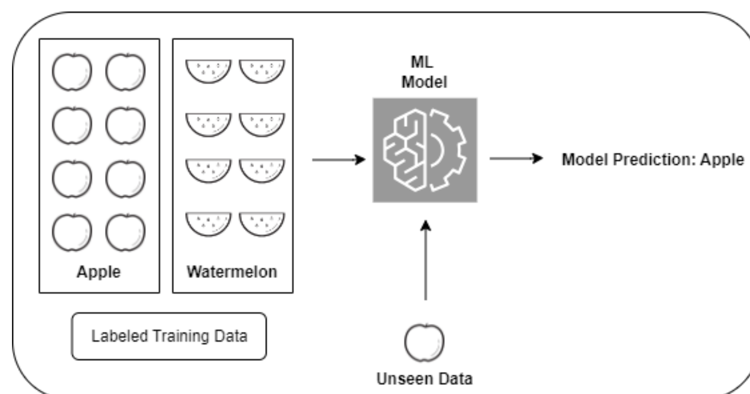

Figure 1.1: Supervised Learning

Through iterative optimization processes, supervised learning algorithms adjust their weights, to minimize prediction errors and accurately classify new, unseen data. In order to prevent the model from overfitting or underfitting, this happens as part of the cross validation process. The supervised learning framework is crucial for solving a wide range of real-world problems at scale. For instance, it enables email service providers to automatically classify incoming emails as spam or non-spam, facilitating efficient inbox management for users. Common algorithms employed in supervised learning include neural networks, Naïve Bayes, linear regression, logistic regression, random forest, and Support Vector Machines (SVM).

2. Unsupervised machine learning: In contrast to supervised learning, unsupervised learning involves analyzing and clustering unlabeled datasets to uncover hidden patterns or structures within the data. Without the presence of explicit labels, unsupervised learning algorithms autonomously identify similarities and differences among data points, organizing them into meaningful clusters or subgroups. This approach is particularly useful for exploratory data analysis, customer segmentation, anomaly detection, and dimensionality reduction.

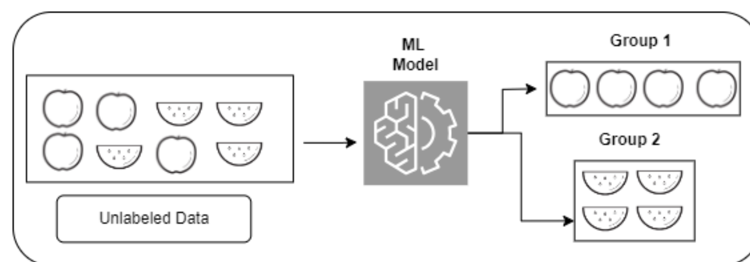

Figure 1.2: Unsupervised Learning

Popular algorithms in unsupervised learning include k-means clustering, hierarchical clustering, and probabilistic clustering methods. Additionally, techniques such as

Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) are commonly utilized for dimensionality reduction, enabling the extraction of essential features from high-dimensional data.

3. Semi-supervised learning: Semi-supervised learning bridges the gap between supervised and unsupervised learning by leveraging both labeled and unlabeled data during the training process. In scenarios where acquiring labeled data is expensive or time-consuming, semi-supervised learning offers a pragmatic solution by utilizing a small labeled dataset to guide the classification and feature extraction from a larger, unlabeled dataset.

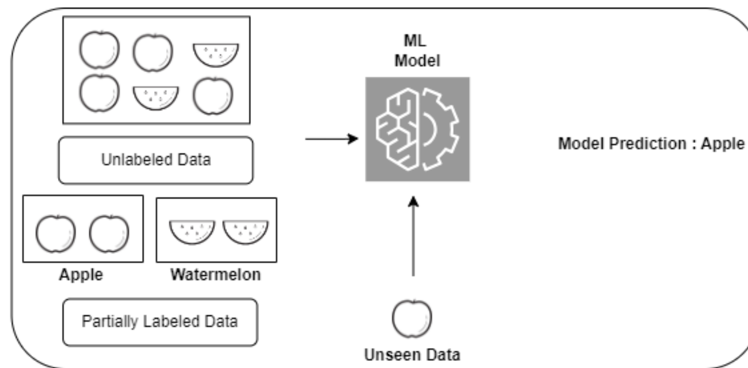

Figure 1.3: Semi-supervised Learning

This approach is particularly beneficial when the availability of labeled data is limited or when labeling data instances is cost-prohibitive. Semi-supervised learning algorithms enhance the efficiency and scalability of machine learning models while maintaining a balance between the supervision provided by labeled data and the autonomy of unsupervised learning. In summary, the trio of supervised, unsupervised, and semi-supervised learning methodologies constitutes essential pillars of machine learning, each offering unique capabilities and applications across diverse domains

and industries. By understanding and harnessing the strengths of these approaches, researchers and practitioners can develop robust machine learning solutions tailored to specific tasks and objectives.

4. Reinforcement machine learning: Reinforcement machine learning represents a distinct paradigm within the realm of machine learning, characterized by its dynamic interaction with the environment and reliance on trial-and-error feedback mechanisms. Unlike supervised learning, where algorithms are trained on labeled sample data, reinforcement learning algorithms learn iteratively through direct interaction with the environment, receiving feedback in the form of rewards or penalties based on their actions. This feedback loop enables the algorithm to autonomously discover optimal strategies or policies for solving a given problem, with the goal of maximizing cumulative rewards over time.

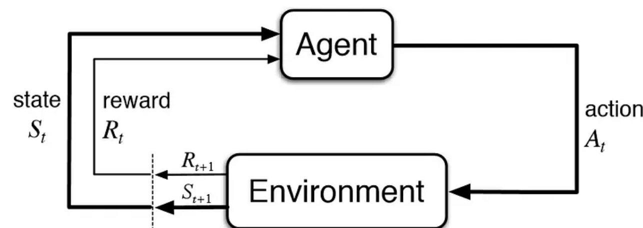

Figure 1.4: Reinforcement Learning

The idea of reinforcement lies at the heart of reinforcement learning, where a series of successful results or actions are reinforced to direct the model toward the best course of action. Trial, error, and delay are essential components of reinforcement learning because the model iteratively investigates various actions and their effects in an effort to enhance performance.

The Q-function, which calculates the expected reward of a particular action in a given state, is a fundamental concept in reinforcement learning algorithms. There are a few popular algorithms for reinforcement learning, such as Q-learning, State-Action-Reward-State-Action (SARSA) and Deep Q-learning.

1.1.2 Deep Learning

Deep learning is a subset of machine learning that uses multi-layered neural networks, called deep neural networks, to simulate the complex decision-making power of the human brain. In today's AI-driven world, deep learning underpins a myriad of applications, playing a pivotal role in tasks ranging from image recognition and natural language processing to autonomous driving and medical diagnostics.

At the core, deep learning are artificial neural networks, which aim to replicate the functioning of biological neurons through a combination of interconnected nodes, weighted connections, and bias terms. These elements collaboratively process input data to accurately identify, classify, and describe objects or patterns within the data.

The following section provides an overview of the architecture behind deep learning, focusing on the components of a perceptron, the basic building block of neural networks, and the structure of shallow and deep neural networks.

Perceptron Structure:

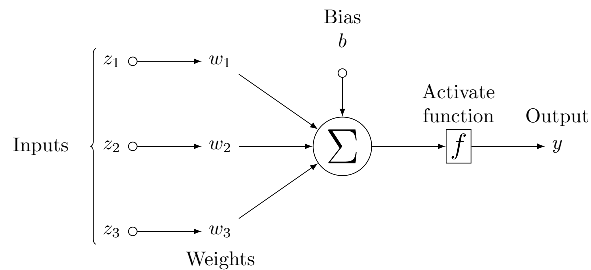

Figure 1.5: Perceptron

A perceptron, the fundamental unit of a neural network, comprises several interconnected components that facilitate data processing and decision-making. The process begins with the calculation of a weighted sum, where input features are multiplied by corresponding weights and summed together along with a bias term. This weighted sum is then passed through an activation function, which introduces non-linearity into the system and determines whether the neuron should be activated.

Shallow Neural Network Architecture:

A neural network consists of interconnected layers, including an input layer, one or more hidden layers, and an output layer. In its simplest form, with only one hidden layer, it is termed a shallow neural network. Each neuron within the network performs the computations outlined above, collectively undergoing forward propagation to generate predictions. Subsequently, during backpropagation, the network adjusts its weights based on the differences between predicted and actual values, iteratively minimizing the overall error through gradient descent.

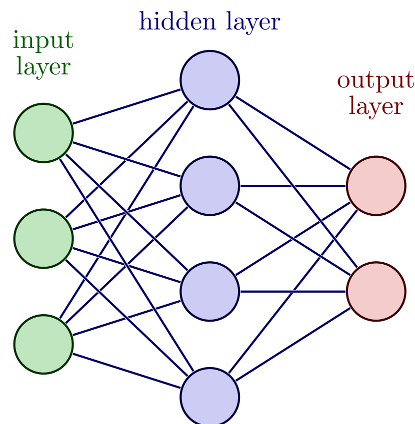

Figure 1.6: Shallow Neural Network

Deep Neural Network Architecture:

A deep neural network extends the concept of a shallow neural network by incorporating multiple hidden layers. Each hidden layer comprises interconnected neurons, with each neuron connected to many others through weighted connections. The depth of the network, attributed to its multiple hidden layers, enhances its capacity to capture intricate data representations and learn complex patterns. The number of hidden layers and neurons therein is determined based on the problem's nature and the dataset's size, with deeper architectures often yielding superior performance in capturing nuanced relationships within the data.

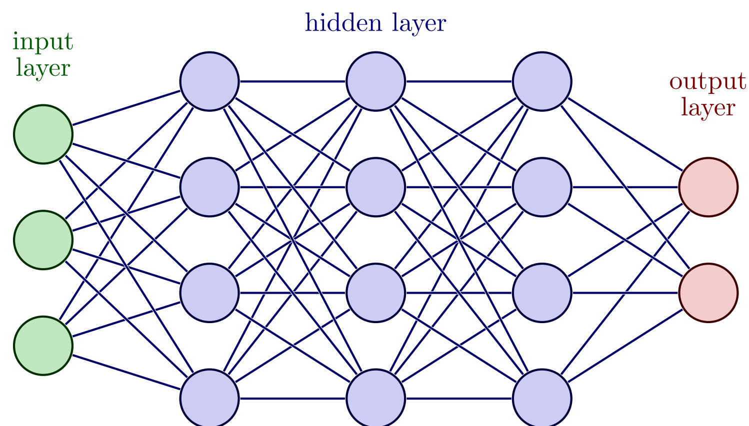

Figure 1.7: Deep Neural Network

1.1.3 Difference between Machine Learning and Deep Learning

Deep learning and machine learning, often used interchangeably, represent distinct yet interconnected fields within artificial intelligence. While both are subsets of AI, they differ in their approaches to learning and processing data. Understanding the nuances between deep learning and machine learning is essential for navigating their respective applications and capabilities effectively.

Table 1.1: Difference between Machine Learning and Deep Learning

Aspect	Machine Learning	Deep Learning
Learning Approach	Statistical algorithms	ANN architecture
Dataset Size	Effective with small amounts of data	Larger
Task Complexity	Low-level tasks	Complex tasks
Training Time	Less time	More time
Feature Extraction	Manual	Automatic
Model Interpretability	Easy to interpret	Black box
Computing Resources	CPU or less computing power	GPU and HPC

Machine learning encompasses a broad range of techniques that apply statistical algorithms to learn hidden patterns and relationships within datasets. It can operate with labeled datasets, known as supervised learning, or ingest unstructured data in its raw form, albeit with more human intervention to determine relevant features. In contrast, deep learning, a subfield of neural networks, leverages artificial neural network architectures to automatically extract features from data, enabling it to handle complex tasks such as image processing and natural language processing [1].

Neural networks, the foundation of deep learning, consist of interconnected node layers, including input, hidden, and output layers. The "deep" in deep learning refers to the presence of multiple hidden layers within a neural network, facilitating the extraction of intricate data representations. These networks have played a pivotal role in advancing fields like computer vision, speech recognition, and natural language processing.

This section explores the polarity between deep learning and machine learning, highlighting their respective learning processes, dataset requirements, and applications. By elucidating these differences, we gain insight into the unique strengths and capabilities of each approach, empowering us to harness their potential in solving diverse real-world problems.

These distinctions make each approach suitable for specific applications and highlight the importance of selecting the appropriate technique based on the task at hand and available resources.

1.1.4 Deep Learning model types and applications

Deep learning models have revolutionized various fields by automatically learning features from data, making them highly effective for tasks such as image recognition, speech recognition, and natural language processing. Among the myriad architectures in deep learning, three stand out as the most widely used: Feedforward Neural Networks (FNNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs).

1. Feedforward Neural Networks (FNNs): FNNs represent the simplest form of artificial neural networks, characterized by a linear flow of information through the network. These networks have found extensive application in tasks such as image classification, speech recognition, and natural language processing. Despite their simplicity, FNNs demonstrate remarkable effectiveness in handling various types of data and tasks.

2. Convolutional Neural Networks (CNNs): CNNs are specifically designed for image and video recognition tasks. By automatically learning features from images through convolutions, CNNs excel in tasks such as image classification, object detection, and image segmentation. The hierarchical architecture of CNNs enables them to capture intricate spatial dependencies within images, leading to superior performance in visual recognition tasks.

3. Recurrent Neural Networks (RNNs): RNNs are tailored to process sequential data, including time series and natural language. Unlike feedforward networks, RNNs possess recurrent connections that allow them to maintain an internal state, capturing information about previous inputs. This capability makes RNNs well-suited for tasks such as speech recognition, natural language processing, and language translation, where context and temporal dependencies are crucial.

In summary, each type of neural network—FNNs, CNNs, and RNNs—offers unique advantages and capabilities, catering to specific data types and tasks. Understanding the characteristics and applications of these neural network architectures is essential for leveraging the full potential of deep learning in various domains, including healthcare, finance,

and automotive industries. In the subsequent sections, we delve deeper into the implementation and performance of these neural network models in the context of developing a mobile dermatology assistant.

Computer Vision:

In computer vision, deep learning models empower machines to interpret and analyze visual data, paving the way for a myriad of applications:

- Object Detection and Recognition: Deep learning facilitates the identification and localization of objects within images and videos, facilitating tasks like autonomous driving, surveillance, and robotics.
- Image Classification: Deep learning models classify images into distinct categories such as animals, plants, and buildings. Additionally, it is used for applications in medical imaging, quality control, and image retrieval.
- Image Segmentation: Deep learning techniques enable the segmentation of images into different regions, allowing for precise identification of specific features within images.

Natural Language Processing (NLP):

In NLP, deep learning models enable machines to comprehend and generate human language, leading to various applications:

- Automatic Text Generation: Deep learning algorithms trained on text corpora can automatically generate summaries, essays, and other textual content.
- Language Translation: Deep learning models proficiently translate text from one language to another, facilitating cross-linguistic communication.
- Sentiment Analysis: Deep learning algorithms analyze the sentiment of text, enabling tasks such as customer service feedback analysis, social media sentiment monitoring, and political discourse analysis.

- Speech Recognition: Deep learning models accurately recognize and transcribe spoken words, enabling applications such as speech-to-text conversion, voice search, and voice-controlled devices.

Reinforcement Learning:

In reinforcement learning, deep learning algorithms train agents to make decisions in dynamic environments to maximize rewards. This approach finds applications in various domains:

- Game Playing: Deep reinforcement learning models have surpassed human expertise in games like Go, Chess, and Atari, showcasing their adaptability and strategic capabilities.
- Robotics: Reinforcement learning techniques train robots to perform intricate tasks such as object manipulation, navigation, and grasping.
- Control Systems: Reinforcement learning is employed in controlling complex systems such as power grids, traffic management, and supply chain optimization, enabling efficient and adaptive decision-making.

In summary, the widespread adoption of deep learning across computer vision, NLP, and reinforcement learning domains underscores its transformative impact on diverse industries, driving innovation and enabling new possibilities in artificial intelligence.

1.2 Motivation

The continuous progress of AI democratization has led to the proliferation of deep learning models deployed in edge and mobile devices for various AI applications. These applications span across diverse domains, including healthcare, where mobile dermatology assistants, eye cancer detection systems, vital signs monitoring, and medical imaging diagnostics have

shown promising results. However, despite the advancements in model compression, accelerator design, and hardware/software co-design techniques to optimize efficiency, existing AI system designs often overlook fairness among diverse groups in the dataset.

The emergence of fairness concerns in AI systems is a critical issue highlighted by recent studies. Commercial AI systems have been found to exhibit gender and skin-type biases, with significant disparities in accuracy rates between different demographic groups. For instance, facial-analysis software has demonstrated considerably higher error rates for dark-skinned human compared to light-skinned human. Similarly, racial disparities have been observed in skin condition identification applications, where accuracy rates are notably lower for individuals with dark skin.

While research efforts have been made to address fairness issues, existing approaches primarily focus on modifying neural network models or data collection practices. However, achieving fairness on resource-constrained edge devices presents new challenges, particularly in balancing model size, computational complexity, and fairness metrics. Traditional methods often involve manual fine-tuning of models, which may not be feasible or scalable in practice.

The motivation behind this research is driven by the need to develop fair and efficient AI systems that can be deployed on edge and mobile devices. By addressing fairness concerns in neural architecture design, this thesis aims to contribute to the development of more equitable AI technologies that uphold principles of fairness and inclusivity. Through empirical evaluations and case studies, a framework with a particular focus on medical AI applications, such as dermatological disease diagnosis.

In summary, this research is motivated by the desire to use the state-of-the-art in fairness-aware neural architecture design and contribute to the development of AI systems that are both accurate and equitable. By combining insights from deep learning, hardware optimization, and fairness-awareness, this thesis seeks to pave the way for a new generation of AI technologies that prioritize fairness and accessibility across diverse populations.

1.3 Flow of thesis

The second chapter focuses on the necessity of compact machine learning models for edge devices and a list of currently utilized models in this context. Moreover, it also highlights the drawbacks of the current models. The third chapter provides a concise overview of the concepts and tasks involved in developing the Android application, with an in-depth explanation of the functions integral to the development process. The fourth chapter provides a detailed account of the experimental setup, methodology, software utilized, and performance evaluation of the application using the FaHaNa ML model. The thesis finishes with a final summary and an exploration of potential future projects using FaHaNa.

Chapter 2: Edge Computing and Fairness in AI

2.1 Rise of Edge Computing

Edge computing has become more popular in recent years due to the spread of Internet of Things (IoT) devices and the growing need for real-time data processing and low-latency applications. Unlike traditional cloud computing, where data is processed and stored in centralized data centers, edge computing brings computational resources closer to the data source, enabling faster response times and reducing bandwidth requirements [2].

At its core, edge computing is driven by the need to overcome the limitations of centralized cloud infrastructure, particularly in scenarios where latency, bandwidth, and privacy concerns are paramount. By distributing computational tasks to edge devices located closer to end-users or IoT endpoints, edge computing minimizes the distance data needs to travel, resulting in faster processing and reduced network congestion.

One of the primary drivers behind the adoption of edge computing is the exponential growth of IoT devices and the massive volumes of data they generate. From smart sensors and wearable devices to autonomous vehicles and industrial machinery, IoT deployments generate vast amounts of data that require real-time processing and analysis. Edge computing enables this data to be processed locally, without the need to transmit it to remote data centers, thereby reducing latency and ensuring timely decision-making.

Moreover, edge computing offers inherent advantages in scenarios where network connectivity is limited or unreliable. By processing data locally at the edge, devices can continue to operate autonomously even in offline or low-bandwidth environments, ensuring uninterrupted functionality and resilience.

Furthermore, edge computing enhances data privacy and security by minimizing the need to transmit sensitive information over public networks. By keeping data localized

and processing it closer to the source, edge computing reduces the risk of data breaches and unauthorized access, thus addressing growing concerns around data sovereignty and compliance with regulatory requirements.

The applications of edge computing span across various industries, including healthcare, manufacturing, transportation, and smart cities. In healthcare, for example, edge computing enables real-time monitoring of patient vital signs, remote diagnostics, and personalized treatment recommendations, all while ensuring patient data privacy and confidentiality.

As the adoption of edge computing continues to grow, fueled by advancements in hardware, software, and networking technologies, it is poised to become an integral part of the modern computing landscape. By bringing computational capabilities closer to the data source, edge computing promises to unlock new opportunities for innovation, efficiency, and scalability across a wide range of industries, shaping the future of digital transformation and IoT-driven applications.

2.2 Efficient CNNs for Mobile & Edge Devices

2.2.1 SqueezeNet

The main goal of SqueezeNet is to strike a balance between high accuracy and low complexity, making it an ideal choice for devices with limited resources like mobiles and embedded systems.

SqueezeNet adopts three strategic approaches to minimize model size while preserving accuracy:

1. Replacement of 3x3 filters with 1x1 filters: This strategy drastically reduces the number of parameters, as a 1x1 filter has 9 times fewer parameters than a 3x3 filter.

2. Reduction of input channels to 3x3 filters: By incorporating squeeze layers that limit the number of input channels, the model's total number of parameters is further diminished.

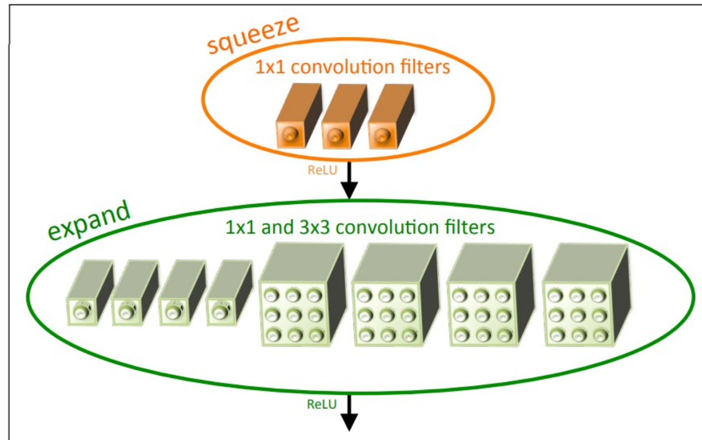

Figure 2.1: Fire Module Figure.

3. Late downsampling: By deferring downsampling within the network, SqueezeNet enables larger activation maps, potentially enhancing classification accuracy within a constrained parameter budget.

The Fire Module

At the heart of SqueezeNet's architecture lies the Fire module, which comprises a squeeze layer of 1x1 filters followed by an expand layer containing a mix of 1x1 and 3x3 filters. [3] This modular design not only reduces the model's size but also maintains a balance between computational efficiency and predictive performance.

SqueezeNet starts with a standalone convolution layer, followed by eight Fire modules and concludes with a final convolution layer. The architecture methodically increases the number of filters in each successive Fire module. Notably, max-pooling operations are strategically placed after certain modules to effectively reduce the spatial size of feature maps, further aligning with the goal of creating a highly efficient model.

SqueezeNet's evaluation highlighted its capability to match AlexNet's accuracy on the ImageNet dataset with substantially fewer parameters, demonstrating the potential of smaller models to achieve high performance.

2.2.2 MobileNet

MobileNet stands as a pivotal development in the domain of efficient neural networks, particularly tailored for mobile applications where computational resources are limited. MobileNetV3 [4], represents a significant leap forward, combining automated search algorithms with novel architectural advancements to optimize performance on mobile devices. The development of MobileNetV3 involved a holistic approach, incorporating hardware-aware network architecture search (NAS), the NetAdapt algorithm, and breakthroughs in network design to fine-tune the model for optimal accuracy-latency trade-offs on mobile CPUs.

Key Innovations in MobileNetV3

1. Automated Search Techniques: MobileNetV3's development employed NAS to automate the design of network architecture, ensuring that the models are specifically optimized for mobile environments. This approach was complemented by the NetAdapt algorithm, which further refines the network by adjusting the number of filters per layer, achieving a delicate balance between computational efficiency and model performance.

2. Efficient Nonlinearities: The introduction of modified swish nonlinearities, specifically tailored for mobile settings, marked a significant enhancement in MobileNetV3. These nonlinearities, adapted to be more quantization-friendly, contribute to the model's efficiency without sacrificing accuracy.

3. Squeeze-and-Excite Optimization: The integration of the squeeze-and-excite mechanism, adjusted to operate efficiently within the constraints of mobile devices, further exemplifies the model's innovative design. This feature allows the network to focus on the most informative features, enhancing its predictive capabilities.

4. Lite Reduced Atrous Spatial Pyramid Pooling (LR-ASPP): For tasks requiring dense pixel predictions, such as semantic segmentation, MobileNetV3 introduces LR-ASPP,

a novel efficient segmentation decoder. This component underscores the model's versatility and its ability to provide state-of-the-art results across various applications, including classification, detection, and segmentation.

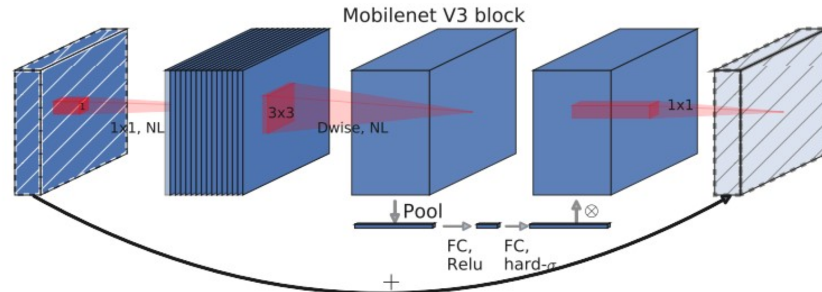

Figure 2.2: MobileNetV3 figure.

MobileNetV3's architecture demonstrates superior performance on standard benchmarks, including ImageNet classification, Common Objects in Context (COCO) dataset detection, and Cityscapes segmentation, showcasing its versatility and efficiency. The model achieves this with significantly reduced computational costs and model sizes, making it ideal for deployment in mobile applications. Such advancements not only enhance the user experience by enabling more sophisticated on-device AI capabilities but also open up new possibilities for mobile vision applications, ranging from real-time object recognition to augmented reality, all while adhering to the stringent power and storage constraints of mobile devices.

In conclusion, MobileNetV3 represents a significant milestone in the evolution of mobile-centric neural networks. By ingeniously blending automated architecture search techniques with innovative design principles, MobileNetV3 sets new standards for efficiency and performance, paving the way for the next generation of mobile vision applications.

2.2.3 MnasNet

MnasNet represents a significant advancement in the field of efficient neural networks, designed specifically for mobile devices. The core challenge addressed by MnasNet[5] is to develop neural networks that are not only accurate but also lightweight and fast enough to operate within the stringent computational limits of mobile devices. Traditional approaches to neural network design have struggled to balance these factors effectively, often requiring extensive manual tuning and compromises between model size, speed, and accuracy. Key Features of MnasNet:

- Automated Neural Architecture Search (NAS): MnasNet introduces an automated NAS framework that incorporates model latency directly into the search objective. This contrasts with previous methods that relied on less direct measures such as the number of floating-point operations (FLOPs) to estimate model performance and efficiency. By measuring real-world inference latency on mobile devices, MnasNet ensures that the resulting architectures are practically efficient for mobile deployment.

- Factorized Hierarchical Search Space: To manage the complexity of the search process and allow for layer diversity, MnasNet employs a novel factorized hierarchical search space. This structure enables the model to explore a variety of layer configurations by dividing the network into blocks, each with its sub-search space. This approach facilitates the discovery of diverse architectures that can better adapt to the varied demands of different parts of the network, leading to improved performance and efficiency.

- Direct Measurement of Inference Latency: One of the innovative aspects of MnasNet is its focus on directly measuring the inference latency of models on mobile devices. This direct measurement addresses the inaccuracies associated with proxy metrics like FLOPs, ensuring that the search process prioritizes architectures that are genuinely efficient in practical deployment scenarios.

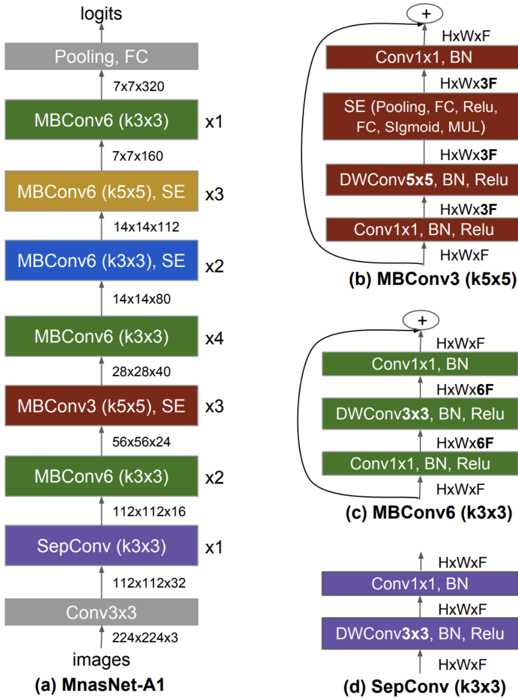

Figure 2.3: MnasNet Architecture.

- Multi-Objective Optimization: MnasNet's search process is guided by a multi-objective optimization goal that seeks to maximize accuracy while minimizing latency. This dual focus ensures that the resulting models achieve a balanced trade-off, delivering high performance without compromising on speed or increasing computational demands beyond what is feasible for mobile devices.

MnasNet has demonstrated superior performance compared to previous state-of-the-art mobile models across multiple vision tasks. For instance, on the ImageNet classification task, MnasNet achieved a top-1 accuracy of 75.2% with a latency of only 78ms on a Pixel phone, outperforming MobileNetV2 and NASNet in terms of both accuracy and speed. Furthermore, MnasNet has shown to be adaptable across different model scaling techniques, consistently outperforming alternatives when adjusted for depth multipliers or input sizes.

The introduction of MnasNet has significant implications for the deployment of AI on mobile devices. By automating the search for efficient neural architectures and directly

optimizing for real-world performance constraints, MnasNet paves the way for more sophisticated and capable mobile AI applications. Whether for image classification, object detection, or other vision tasks, MnasNet provides a robust foundation for developing AI models that can operate within the limited computational resources of mobile devices without sacrificing accuracy or performance.

In conclusion, MnasNet represents a groundbreaking approach in the field of efficient neural networks, offering a powerful tool for the automated design of high-performance, resource-efficient models tailored for mobile applications. Its focus on practical efficiency and the novel methodologies it introduces for NAS set a new standard for the development of AI models in the mobile context.

2.2.4 FaHaNa

The Fairness and Hardware aware Neural Architecture Search (FaHaNa) framework integrates four key components to optimize the neural architecture search (NAS) for mobile healthcare applications, focusing on performance, fairness, and efficiency. The components are:

1. RNN-based Controller: Utilizes reinforcement learning via the Monte Carlo policy gradient algorithm to dynamically predict and update the hyperparameters of child networks based on accuracy, unfairness scores, and latency, ensuring compatibility with both software and hardware requirements.[6].

2. Block-based Search Space: Features modular blocks inspired by MobileNetV2 and ResNet-18 designs, which allow for versatile hyperparameter adjustments. This setup aims to enhance the fairness of networks without compromising performance, making it ideal for varied healthcare applications on mobile devices. In addition, we can facilitate the skip operation within a block to enhance the adaptability of the neural network's depth

3. Backbone Architecture Producer:

- Feature Map Analysis: Streams data from different demographic groups through a pre-trained model to capture feature maps at various layers.
- Feature Variation Assessment: Analyzes these maps using the L2-norm metric to identify demographic disparities.
- Threshold Determination and Layer Selection: Establishes a threshold for feature variation to identify critical layers for further optimization, freezing less variable layers to streamline the search process.

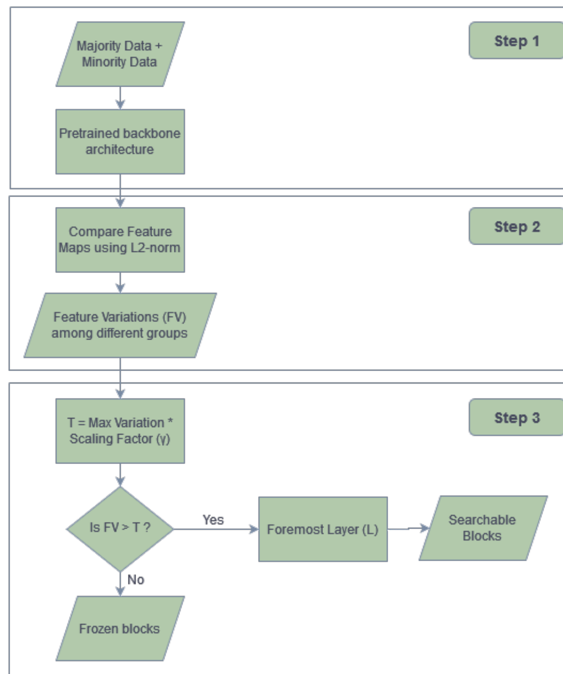

Figure 2.4: FaHaNa Framework Figure

For an detailed comprehension of the Backbone Architecture Producer's process, the figure 2.4 flowchart illustrates its operational flow, from data streaming to the separation of the blocks.

4. Evaluator and Trainer:

    - Hardware Specification Assessment: Quickly verifies if a child network meets the hardware requirements, bypassing training for non-compliant designs.
    - Offline Block Performance Testing: Evaluates block performance on specific hardware to refine the architecture, ensuring efficient operation.
    - End-to-End Evaluation and Training: Conducts comprehensive testing of the final architecture across various datasets to assess both accuracy and fairness, followed by training of adaptable blocks to optimize performance.
    - Training of Searchable Blocks: If the hardware specifications are feasible, the searchable blocks in the child network are trained to learn the function for the designated dataset. After training, the model is tested on the dataset and subgroups to assess accuracy and unfairness.

These components are designed to specifically developed to accelerate the Neural Architecture Search (NAS) process, focusing on creating neural architectures that are efficient, fair, and suitable for deployment on mobile healthcare devices.

2.3 Comparison of NAS Frameworks

The evolution of Neural Architecture Search (NAS) frameworks has significantly advanced the field of machine learning, particularly in the development of optimized neural networks for diverse applications. We discussed about the FaHaNa framework, which introduces a novel approach by integrating fairness and hardware efficiency into the architecture search process. This section provides a comparative analysis of FaHaNa with other leading NAS frameworks to highlight its unique contributions and potential advantages in the landscape of neural architecture design.

- Traditional NAS Frameworks: Traditional NAS frameworks, such as NASNet [7] and AmoebaNet [8], utilize reinforcement learning or evolutionary algorithms to optimize

network architectures for maximum accuracy. These frameworks primarily focus on improving performance metrics like accuracy and computational efficiency without considering fairness or specific hardware constraints.

Comparison: Unlike traditional frameworks, FaHaNa explicitly incorporates fairness as a core component of the search objective. This integration ensures that the resulting architectures not only perform well but also provide equitable predictions across diverse demographic groups, a critical aspect often overlooked in conventional NAS.

- ProxylessNAS: ProxylessNAS [9], focuses on reducing the search time and computational resources required by the NAS process. These frameworks leverage techniques like gradient-based optimization to directly learn architecture parameters without the need for proxy tasks.

  Comparison: FaHaNa incorporates a model freezing method to enhance the efficiency of the search process. This method reduces the search space significantly by freezing certain layers of the network based on their impact on fairness, thus accelerating the optimization process without compromising the quality of the search.

- EfficientNet: Developed by Tan and Le [10], EfficientNet is another framework known for optimizing model scaling, thus achieving remarkable balance between accuracy and efficiency. EfficientNet uses a compound scaling method to uniformly scale network width, depth, and resolution, which is a computationally effective strategy for enhancing model performance.

  Comparison: FaHaNa's block-based search space and RNN-based controller allow for more dynamic architecture adjustments tailored specifically to dermatological image analysis, providing an advantage in scenarios requiring customized solutions for skin-type diversity and fairness.

- Hardware-Aware NAS: Frameworks like MnasNet [5] and MobileNetV3 [4] represent a shift towards hardware-aware NAS, where the search process considers the computational constraints of the deployment platform. These frameworks optimize for a

balance between accuracy and latency or power efficiency, making them suitable for mobile and embedded devices.

Comparison: FaHaNa extends beyond simple hardware awareness by additionally emphasizing fairness. While frameworks like MnasNet focus on hardware constraints, FaHaNa simultaneously addresses the need for fairness in predictions, making it particularly valuable for applications in sensitive areas such as healthcare.

The FaHaNa framework stands out in the NAS landscape by uniquely addressing fairness alongside accuracy and hardware efficiency. This comprehensive approach not only meets the technical requirements of modern AI applications but also addresses the ethical implications of automated decision-making systems. As AI continues to permeate diverse sectors, the importance of frameworks like FaHaNa that prioritize equitable outcomes cannot be overstated.

2.4 Importance of Fairness in Dermatology AI

The importance of fairness in dermatology AI is underscored by recent research aiming to address and mitigate the biases present in these systems, particularly those related to skin type and ethnicity. These biases can significantly affect the accuracy and reliability of skin lesion diagnoses, thereby impacting patient care and outcomes. Following are some works that prioritize fairness:

- Fairness Through Disentanglement and Contrastive Learning: FairDisCo [11] , a framework that improves fairness in dermatology AI by disentangling sensitive attributes (like skin type) from representations used for diagnosis. This approach demonstrated fairer and superior performance in classifying skin lesions across diverse skin types [11].

- Multi-Exit Frameworks for Fairness: proposed a multi-exit framework [12] designed to enhance fairness in dermatological disease diagnosis. This framework allows for

early exits for instances with high confidence from internal classifiers, thus improving fairness without sacrificing accuracy [12].

- Multi-Dimensional Fairness with Muffin: Muffin [13] , a framework aimed at addressing multi-dimensional fairness by uniting multiple models. This method significantly improved fairness across different attributes, demonstrating the importance of considering various factors contributing to bias in dermatology AI[13].

So ensuring fairness in dermatology AI is critical for its reliable and equitable application across diverse populations. By addressing biases related to skin type and other sensitive attributes, researchers aim to create more accurate and fair diagnostic tools that can improve patient care regardless of ethnicity or skin color. This is an essential step towards achieving equitable healthcare outcomes in dermatology and beyond.

2.5 Challenges in Existing AI Systems

The integration of Artificial Intelligence (AI) in dermatology, particularly in the context of mobile healthcare, presents a host of challenges that necessitate thoughtful consideration and innovative solutions. These challenges span technical, ethical, and logistical domains, directly impacting the development, deployment, and acceptance of AI systems in dermatological practices.

- Data Quality and Quantity: High-quality, diverse datasets are critical for training robust AI models. In dermatology, there's a pressing need for datasets that accurately represent various skin types, conditions, and stages of disease across different populations. The lack of such comprehensive datasets can lead to biased algorithms with reduced accuracy and effectiveness in clinical settings [14].

- Algorithmic Explainability: AI models, particularly those based on deep learning, often operate as "black boxes", where the decision-making process is not transparent. In healthcare, where decisions can have significant consequences, the inability to

understand and interpret AI recommendations limits its utility and trustworthiness among clinicians and patients [15].

- Ethical Considerations and Patient Consent: Ethical challenges, including patient privacy, consent for using personal data, and the potential for algorithmic bias, are paramount concerns. Ensuring fairness and mitigating biases in AI systems are critical to prevent exacerbating existing health disparities [16].

- Access and Equity: As mobile healthcare and AI dermatology tools proliferate, ensuring equitable access to these technologies becomes increasingly important. Challenges include addressing the digital divide, ensuring AI tools are affordable and accessible to underserved populations, and tailoring solutions to meet diverse healthcare needs.

Addressing these challenges is essential for realizing the full potential of AI in dermatology and mobile healthcare. Collaborative efforts among researchers, clinicians, industry stakeholders, and regulatory bodies, alongside the development of equitable, transparent, and patient-centered AI solutions, are imperative for advancing dermatological care and improving patient outcomes.

2.6 Contribution

This thesis contributes to the integration of the Fairness and Hardware aware Neural Architecture Search (FaHaNa) framework within Android applications, specifically tailored for edge devices. By embedding advanced neural network capabilities directly into mobile devices, this work facilitates real-time, fair, and efficient processing on edge, making significant advancements in mobile healthcare applications. The key contributions of this thesis are outlined as follows:

1. Development of an Android Application Utilizing FaHaNa: Successfully implemented the FaHaNa framework in an Android application, enabling the deployment of fair and

efficient neural networks on mobile devices. This is one of the first applications to integrate a fairness-aware neural architecture search directly into a mobile environment, addressing the dual challenges of achieving fairness in neural network predictions and adhering to the computational constraints of mobile devices.

2. Enhancement of Real-time Processing Capabilities: Enhanced the application's capability to process data in real-time by integrating optimized neural architectures, which are capable of running efficiently on Android devices without compromising on performance or accuracy. This contribution is pivotal for applications requiring immediate processing, such as those used in medical diagnostics or real-time decision-making scenarios.

3. Experimental Validation on Dermatological Disease Diagnosis: Conducted a comprehensive evaluation of the application using a dermatology dataset to validate the effectiveness of the FaHaNa framework in a real-world scenario. This thesis not only demonstrated the app's capability to handle diverse skin types fairly but also highlighted its efficiency and accuracy in processing diagnostic imaging on mobile devices.

4. Usability and Accessibility Improvements: Focused on user interface design and experience to make the application accessible and practical for everyday users, particularly for those in remote or resource-constrained environments. This involves simplifying the interaction process and ensuring that the app provides clear, understandable outputs for non-specialist users.

These contributions showcase the potential of fairness-aware neural architectures in enhancing mobile healthcare technologies. It also provides a foundation for future research into the development of robust, fair, and efficient applications that are capable of operating within the constraints of mobile and edge devices, thereby broadening the scope and impact of artificial intelligence in everyday healthcare solutions.

Chapter 3: Development of the Android Application

3.1 Why Android?

The decision to build an Android application using the FaHaNa model was driven by several key factors which are listed below:

1. Wide Accessibility: Android devices make up a significant portion of the mobile market, ensuring the application can reach a broad audience. This widespread accessibility is crucial for applications intended for medical or other socially impactful purposes.

2. Flexibility and Openness: Android offers a more open environment compared to other operating systems, which allows for deeper integration and customization of applications. This flexibility is essential for implementing complex machine learning models like FaHaNa, which require specific configurations and optimizations to function efficiently.

3. Efficient ML model deployment: Android supports efficient deployment of the advanced ML model with libraries like Pytorch Mobile, Tensorflow Lite.

All these make Android an ideal platform for deploying advanced neural network technologies.

3.2 Fundamentals of Android Application Architecture

In the context of developing an Android application, it is crucial to have a robust understanding of the Android platform's fundamental concepts and components. This knowledge forms the foundation upon which the application's functionality is built. In the following

section we will see an expanded and refined explanation of the Android-specific elements and the files used in this Android App.

3.2.1 Android General Concepts

An activity [17] is a core component of an Android application that provides a screen with which users can interact. A typical application consists of multiple activities that are interconnected. For instance, the main activity is the first screen presented to users upon launching the application.

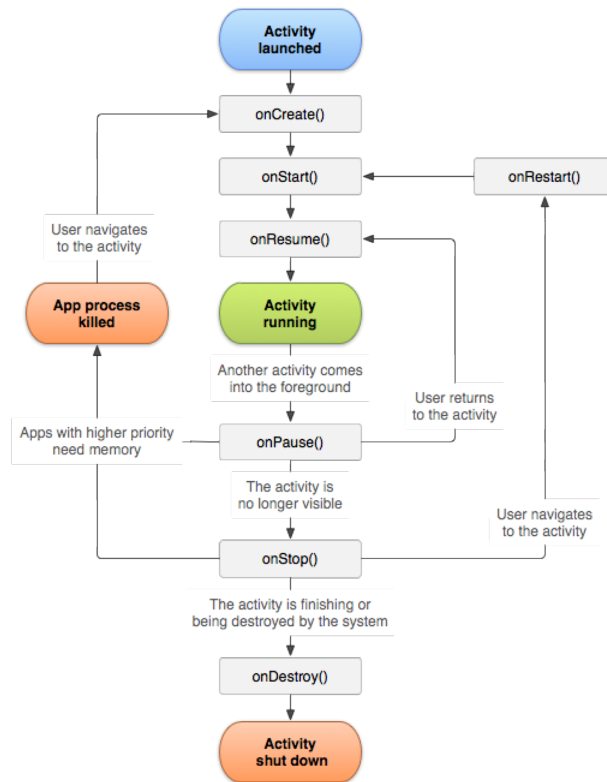

Figure 3.1: Activity Lifecycle

In our project, 'ML for Mobile Healthcare', the initial launch activity (activity_main.kt) and subsequent activities are launched based on the users interaction. For example, if the "Start New Session button" is pressed by the user, it will direct to the activity_camnew.kt activity and end up in activity_result.kt to display the result of our diagnosis. If the user wants to select between the ML models for the diagnosis, he can use the "User Preferences" button which in turn will direct to activity_usr_perf.kt. We can also view a short description about the app by selecting the "About software" button which leads to the activity_abt_software.kt.

3.2.2 Lifecycle Management

Managing an activity's lifecycle is fundamental for creating efficient, user-friendly applications. Activities primarily exist in three states:

- Resumed: The activity is active and interacting with the user.
- Paused: The activity is partially visible but does not have focus, often because another activity is active.
- Stopped: The activity is completely obscured by another activity and is not visible to the user.

3.2.3 Lifecycle callbacks

Each activity manages its visibility and interactivity through a series of lifecycle methods, which developers can override to tailor the application's behavior during state transitions. But the main activity carefully manages its visibility and interactivity through lifecycle methods. The lifecycle events include:

- onCreate(): Called when the activity is first created. This is where you set up the layout and initialize your app components.
- onResume(): Called just before the activity starts interacting with the user, making it ideal for initializing components that are paused or stopped.

- onPause(): Invoked when the user is leaving the activity, which is a good stage to save data or release system resources.
- onDestroy(): Called before the activity is destroyed, where you clean up any resources that were used.

```
public class ExampleActivity extends Activity {
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
// The activity is being created.
}
@Override
protected void onResume() {
super.onResume();
// The activity has become visible (it is now "resumed").
}
@Override
protected void onPause() {
super.onPause();
// Another activity is taking focus (this activity is about to be "paused").
}
@Override
protected void onDestroy() {
super.onDestroy();
// The activity is about to be destroyed.
}
}
```

3.2.4 Essential Files and Configuration

All activities and other application components must be declared in the AndroidManifest.xml file, which outlines essential details such as the application's package name, components (activities, services, etc.), permissions, and the minimum SDK version required by the app. Here is the list of the kotlin files that are used in this android project:

- activity_abt_software
- activity_camnew

- activity_main
- activity_result
- activity_usr_pref
- module_imageprocessing
- module_mapping
- module_param
- module_pytorch

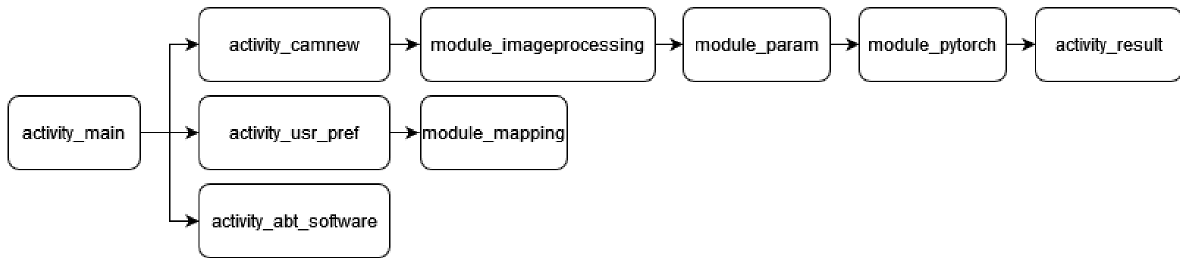

Figure 3.2: Android App Architecture

Figure 3.2 illustrates the architecture of our Android application, providing a visual representation for better understanding of its structure and workflow. In the following sections, we will delve into a detailed explanation of each file, exploring its role and functionality within the project.

3.3 Dynamic Interaction in Activity_main

Overview of activity_main The activity_main class, a pivotal component of the "ML for Mobile Healthcare" application, serves as the primary user interface where users begin their interaction with the app. This class extends AppCompatActivity, which provides

support for features like the action bar along with backward-compatible versions of other important features on older versions of the Android platform.

Upon launching the application, the onCreate method is invoked. This method sets up the initial state of the activity, including the layout configuration and initial animations. It starts with setting the content view to activity_main.kt and hiding the support action bar to provide a full-screen experience.

Dynamic User Interface Elements Within onCreate, the main activity dynamically interacts with various UI components:

- Main Logo: The main logo is added to a mutable set of views, which is used later to manage animations across all views.
- Start New Session Button: This button initiates the core functionality of capturing images. Upon clicking, it plays a bounce animation and then transitions to activity_camnew.kt using an intent with a fade transition effect.
- User Preferences Button: Similar to the start button, this button also uses bounce animation. Clicking this button transitions the user to activity_usr_pref.kt, where they can set their preferences.
- About Software Button: This button leads to activity_abt_software.kt, providing users with information about the app.

In the following sections, we will explore the complete process from capturing an image to obtaining a diagnostic result, beginning with activity_camnew.

3.4 Camera Functionality in activity_camnew

Overview of activity_camnew The activity_camnew class is central to the "ML for Mobile Healthcare" application, handling the camera functionalities that enable users to capture images for analysis. This class extends AppCompatActivity and uses features from

the CameraX library to simplify camera operations, ensuring compatibility and ease of implementation across Android devices.

Permission Handling Upon creation of the activity, the application checks and requests necessary permissions from the user to access the camera and storage. This is vital for the application's functionality, as Android security norms mandate user consent for accessing sensitive hardware and user data.

```
if (allPermissionsGranted()) {
    startCamera()
} else {
    ActivityCompat.requestPermissions(
        this, REQUIRED_PERMISSIONS, REQUEST_CODE_PERMISSIONS
    )
}
```

Camera Initialization and Setup The startCamera method orchestrates the setup of the camera. It involves configuring the camera provider, setting up a preview use case, and binding these components to the lifecycle of the activity. This method ensures that the camera lifecycle is tied to that of the activity, enhancing resource management and user experience.

```
val preview = Preview.Builder().build().also {
    it.setSurfaceProvider(viewBinding.prevCamnewPreviewView.surfaceProvider)
}

val cameraSelector = CameraSelector.DEFAULT_BACK_CAMERA

cameraProvider.bindToLifecycle(
    this, cameraSelector, preview, imageCapture
)
```

Capturing Images The takePhoto method encapsulates the functionality to capture images. It defines how images are saved, including the file name formatting and metadata

storage, which are crucial for later retrieval and usage. The method uses an ImageCapture use case from CameraX, which simplifies the process of capturing and storing images.

```
imageCapture.takePicture(
    outputOptions,
    ContextCompat.getMainExecutor(this),
    object : ImageCapture.OnImageSavedCallback {
        override fun onImageSaved(output: ImageCapture.OutputFileResults) {
            val msg = "Photo capture succeeded: ${output.savedUri}"
            Toast.makeText(baseContext, msg, Toast.LENGTH_SHORT).show()
        }
    }
)
```

Lifecycle and Transition Management The activity also manages transitions and animations, ensuring a smooth user experience when navigating between activities. This is handled in lifecycle methods such as onStart and onPause, where UI elements are animated to fade in or out based on the activity state.

User Interface and Interactivity The UI interactions and their intended functionalities are highlighted below:

- Take Photo Button: Initiates the takePhoto method with a bounce animation to provide visual feedback.
- Switch Camera Button: Could potentially switch between front and back cameras.
- Import from Gallery: Allows users to pick an image from the gallery, which is essential for applications allowing historical data analysis or secondary image processing.
- View Binding for Direct UI Manipulation: Utilizes Android's view binding to interact directly with UI components, minimizing redundant code and improving performance.

From activity_camnew, the application transitions to module_imageprocessing, where the captured image is converted into a bitmap format suitable for analysis by the machine learning model.

3.4.1 Image Processing Techniques in module_imageprocessing

Overview of module_imageprocessing The module_imageprocessing class is designed to facilitate various image processing operations crucial for preparing images for analysis. It includes methods for loading images from the application's assets and file system, resizing images, and performing scaling operations conditioned on specific parameters.

Loading and Scaling Bitmaps The obtainBitmapFromAsset and obtainBitmapFromFilePath methods in the class handle the loading of bitmap images from the application's assets and external storage, respectively. These methods are essential for the preprocessing steps like normalisation which are required before feeding the images into a machine learning model.

```
fun obtainBitmapFromAsset(context: Context, pictureFilePath: String): Bitmap {
    // Load and possibly scale the bitmap from assets
}

fun obtainBitmapFromFilePath(pictureFilePath: String): Bitmap {
    // Decode and scale bitmap from file path
}
```

Adaptive Image Scaling The image scaling functionality demonstrates an adaptive approach based on parameters defined in module_param. The use of conditional scaling (using pytorch_param_useScaling) allows the application to dynamically adjust the image size based on the requirements of the processing algorithm, which can be critical for performance optimization and accuracy in machine learning applications.

```
if (module_param.pytorch_param_useScaling) {
    bitmap = Bitmap.createScaledBitmap(
        bitmap,
        module_param.pytorch_param_bitmapScaleFactor.first,
        module_param.pytorch_param_bitmapScaleFactor.second,
        module_param.pytorch_param_useFilterWhileScaling
    )
}
```

Asset Management The assetFilePath method highlights a method for copying files from the application's asset directory to a writable directory. This operation is often necessary when working with libraries that require file paths (like PyTorch Mobile) because assets within the APK are not accessible via absolute file paths.

```
fun assetFilePath(context: Context, asset: String): String {
    // Copy an asset to a writable location and return the file path
}
```

Once the user-provided input image is processed, we proceed with module_param to configure the necessary parameters for the machine learning model.

3.4.2 Configuration Management in module_param

Overview of module_param The module_param class plays a crucial role in managing the configuration parameters of the "ML for Machine Learning" application. It encapsulates settings for machine learning operations, including model paths, scaling options, and performance tweaks. This centralized approach to parameter management ensures that changes to configurations require minimal code adjustments and can be easily tracked and maintained.

Machine Learning Parameters This class defines several properties to configure the behavior of machine learning models, allowing for easy adjustments that cater to different operational needs or optimization strategies. The parameters include:

- Model Name: Specifies the filename of the PyTorch model.
- Use Lite Loader: A boolean indicating whether a lite version of the loader should be used, potentially improving performance on mobile devices.
- Use Scaling: Controls whether input images should be scaled according to the specified dimensions, crucial for models expecting a certain input size.

- Bitmap Scale Factor: Defines the dimensions to which the input images are resized, ensuring they meet the model's input size requirements.
- Use Filter While Scaling: Indicates whether a filtering process should be applied during image scaling, which can affect the quality of the input images and consequently the model output.

```
public var pytorch_param_modelName: String = "FahaNa-Small.ptl"
public var pytorch_param_useLiteLoader: Boolean = false
public var pytorch_param_useScaling: Boolean = false
public var pytorch_param_bitmapScaleFactor: Pair<Int, Int> = Pair(224, 224)
public var pytorch_param_useFilterWhileScaling: Boolean = false
}
```

Initialization of Classification Mapping The class also includes a lambda function, initClassificationMapping, which initializes the mapping module. This function encapsulates the logic required to load classification mappings from a CSV file, demonstrating a functional programming approach to initializing critical data components. Based on the machine learning model selected by the user, we assign and integrate the corresponding model into the application through the subsequent module.

3.4.3 Integration of PyTorch Models in Mobile Applications

Overview of module_pytorch The module_pytorch class is instrumental in implementing PyTorch functionalities within the "ML for Mobile Healthcare" app. It manages the loading of trained machine learning models and performing inference operations, thereby serving as the bridge between the app's core functionalities and its machine learning capabilities.

Model Loading and Initialization This class provides a loadModule function that dynamically loads a PyTorch model based on the application settings stored in module_param. This flexibility allows the application to switch between standard and lite versions of the PyTorch loader, accommodating various device capabilities and performance considerations.

```
fun loadModule(context: Context, modelName: String) {
    ptModule = Module.load(module_imageprocessing.assetFilePath(context, modelName))
    Log.d("Q_PYTORCH", "loadModule - modelName $modelName SUCCESS.")
    isReady = true
}
```

Running Inference The runInference method illustrates how a bitmap image is processed and analyzed using the loaded PyTorch model. This method converts the bitmap to a tensor, runs the model inference, and then interprets the output to determine the most likely classification result. Such functionality is central to applications that require real-time data processing and decision-making.

```
fun runInference(bitmap: Bitmap, context: Context): Int {
    val tensorInput = TensorImageUtils.bitmapToFloat32Tensor(bitmap, ...)
    val tensorOutput = ptModule.forward(IValue.from(tensorInput)).toTensor()
    val floatArrayOutput = tensorOutput.dataAsFloatArray
    return floatArrayOutput.indexOfFirst { it == floatArrayOutput.maxOrNull() }
}
```

Asynchronous and Timed Execution This class also includes an advanced method, runTimedInferenceOnAnotherThread, which executes the inference process on a separate thread and imposes a time limit for the operation. This method uses Java's concurrency utilities to manage execution timeouts, ensuring that the application remains responsive, even during intensive computational tasks.

```
fun runTimedInferenceOnAnotherThread(bitmap: Bitmap, context: Context, timeout: Int): Int
{
    val callable = Callable { runInference(bitmap, context) }
    val future = scheduledExecutorService.submit(callable)
    return future[timeout.toLong(), TimeUnit.SECONDS]
}
```

Next, we proceed to the activity_result module which is utilized to present the inferred diagnosis results to the user.

3.4.4 Displaying Results in activity_result

Overview of activity_result The activity_result class extends AppCompatActivity and is primarily focused on the final display of diagnostic results derived from the machine learning model's analysis. It manages bitmap manipulations, user interface updates, and integrates closely with other modules like module_pytorch and module_mapping to fetch and display the relevant information.

Initialization and Image Handling Upon creation, the activity sets up the user interface from a predefined layout and hides the action bar to maximize screen usage for content display. It retrieves a bitmap image passed through intents, which is then used for further processing.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_result_scene_done)
    supportActionBar?.hide()

    val scaledBitmap: Bitmap = getTransferredBitmap()
    imageBuffer = scaledBitmap
    findViewById<ImageView>(R.id.imag_result_imageView).also {
        it.setImageBitmap(scaledBitmap)
    }
}
```

Bitmap Processing and Model Inference The core functionality of activity_result is to use the bitmap image obtained either directly from the camera or passed via another activity and use it to perform inference using the PyTorch model. The getResultIndexFromBitmap method manages this by calling runInference from module_pytorch, which returns the index of the diagnosed condition.

```
private fun getResultIndexFromBitmap(bitmap: Bitmap): Int {
    return module_pytorch.runInference(bitmap, this).also {
        Log.d("Q_ACTIVITY_RESULT", "GETRESULTINDEXFROMBITMAP(): RECEIVED DECISION $it")
```

```
    }
}
```

Diagnostic Results Once the inference is completed, the displayDisease method uses the index obtained to fetch and display the disease details. It interacts with module_mapping to retrieve the corresponding disease name and description based on the index.

```
private fun displayDisease(diseaseIndexKey: Int) {
    module_mapping.getPairFromIndex(diseaseIndexKey).let {
        findViewById<TextView>(R.id.text_result_diseaseTitle).text = it.first
        findViewById<TextView>(R.id.text_result_diseaseDetail).text = it.second
    }
}
```

The activity also sets custom typefaces for textual elements, enhancing the user interface's aesthetics and readability. Such details are crucial for maintaining an engaging user experience, particularly in applications dealing with medical data where clarity is crucial.

3.5 Implementing User Preferences in activity_usr_pref

Overview of activity_usr_pref This activity is triggered when the user selects the "User Preferences" button. The activity_usr_pref class extends AppCompatActivity and is primarily focused on enabling users to select their preferred machine learning (ML) model via a radio button interface. This functionality demonstrates the app's flexibility in adapting to user choices.

The activity_usr_pref involves setting up a listener for the RadioGroup that detects when the user selects a different radio button. Each selection triggers feedback mechanisms including a Snackbar and a Toast, ensuring the user is well-informed about the choice made. This is critical in maintaining transparent communication with the user, enhancing the interactive experience. The selected choice is stored and sent to the module_mapping, where it is stored in CSV and will be further discussed in the coming section.

3.5.1 Data Mapping and Error Handling in module_mapping

Overview of module_mapping The module_mapping class is designed to provide a reliable way to fetch mapped values based on integer keys. This functionality is essential for an application where mappings need to be dynamically loaded and accessed, such as decoding indices into meaningful descriptions within the app.

Initialization of Mappings The class utilizes a companion object to hold static members and functions, which allows the data to be loaded and accessed throughout the app without needing an instance of the class. The init function is responsible for loading the mapping data from a CSV file stored in the application's assets. This approach ensures that the mapping data is available immediately after the app is launched and throughout the app's lifecycle.

Fetching Data The getPairFromIndex function demonstrates robust error handling and logging practices, crucial for maintaining the reliability of the application. If the mappings are not ready (perhaps due to a failed initialization), the function throws a RuntimeException, preventing any further unreliable operations. This method logs detailed information about the operation, which aids in debugging and maintaining the software.

In module_mapping, by default the FaHaNa-Small model is selected. The selected model is stored and then utilized by the module_param.

3.6 Design for Software Information in activity_abt_software

Overview of activity_abt_software The activity_abt_software class extends AppCompatActivity and is primarily designed to provide users with detailed information about the "ML for Mobile Healthcare" application. It utilizes view binding for efficient interaction with the layout components, ensuring a robust implementation free from the common pitfalls of manual view handling.

Implementation of View Binding View binding is a feature that allows for more precise and safe interaction with views in the code. By using ActivityAbtSoftwareBinding, the class binds directly to the layout, eliminating the need for findViewById, reducing boilerplate code, and preventing null pointer exceptions associated with view interaction.

```
private lateinit var binding: ActivityAbtSoftwareBinding

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityAbtSoftwareBinding.inflate(layoutInflater)
    setContentView(binding.root)
}
```

The core functionality within activity_abt_software involves setting a descriptive text about the application's purpose and functionality. This text is a key element in providing users with an understanding of what the application offers, particularly in terms of its advanced AI capabilities for healthcare. The class includes methods for handling specific Android lifecycle events, such as onStart and onPause, ensuring that the app maintains good performance and proper state management. These methods can be used to manage resources or adjust UI elements based on the app's state changes.

Chapter 4: Experimental Results

4.1 Experimental Setup

4.1.1 Dataset

The dermatology dataset is built based on patient images collected in the field and the open-access datasets including ISIC 2019 [18] for light-skin, Dermnet [19], and Atlas dermatology [20] for dark-skin. Using all the images present in the dermatology dataset, we can perform the classification task of the following dermatology diseases:

- 1 - Dermatofibroma [DF]
- 2 - Basal cell carcinoma [BCC]
- 3 - Melanoma [MEL]
- 4 - Squamous cell carcinoma [SCC]
- 5 - Melanocytic nevus [NV]

4.1.2 FaHaNa Settings

During the evaluation phase, the parameters α and β of the recurrent neural network (RNN) controller were both set to 1. This configuration aimed to guide the controller in seeking neural architectures that strike a balance between accuracy and fairness. The producer component of FaHaNa utilized MobileNetV2 as the backbone architecture. The scaling factor (γ) was set to 0.5 to facilitate the selection of frozen blocks within the backbone architecture. This setting enabled efficient freezing of specific blocks to expedite the search process while maintaining fairness considerations. To facilitate training, validation, and evaluation, the dataset was partitioned into three distinct sets:

- Training Set: Consisting of 60% of the total images, this set was utilized for training the neural architectures.
- Validation Set: Comprising 20% of the images, this set facilitated hyperparameter tuning and model selection.
- Test Set: The remaining 20% of the images constituted the test set, which was reserved for assessing the generalization performance of the trained models.

Reinforcement Learning Episodes: The reinforcement learning process within FaHaNa was conducted over 500 episodes. This iterative approach allowed the framework to explore the search space effectively and identify optimal neural architectures.

FaHaNa-Nets: Upon completion of the search process, a series of neural architectures were identified and denoted as FaHaNa-Nets. We have obtained two kinds of neural net which are FaHaNa-Fair and FaHaNa-Small. FaHaNa-Fair is used to achieve highest Fairness and FaHaNa-Small is set to have smallest size and lowest latency. These architectures represent the culmination of the optimization process, embodying the principles of fairness-aware neural architecture search.

These settings and configurations provided a standardized framework for conducting experiments within the FaHaNa framework, facilitating consistent and reproducible results across multiple trials.

4.1.3 Edge Devices

To assess the real-world performance of FaHaNa-Nets and benchmark them against state-of-the-art small neural networks, we conducted latency comparisons on two different edge devices:

1. Raspberry PI Model B [21] : Broadcom BCM2711 equipping a 1.5 GHz quad-core ARM Cortex-A72 processor and 8 GB memory

2. Xiaomi Redmi Note-4: Snapdragon 625 equipping ARM Cortex-A53 octa-core processor and 3 GB memory.

The latency is determined by executing the trained models on both devices for inference using a standard PyTorch framework.

4.2 Workflow

To incorporate the FaHaNa-Small model, which is stored as a pickle file (.pkl), into an Android application for mobile devices, we undertook a multi-step process centered around model conversion and application development. Initially, a Python script was developed to convert the trained Pytorch model into a format suitable for mobile interpretation. This involved transforming the model into a TorchScript file (.pt) and subsequently into a Lite version (.ptl) to optimize it for mobile platforms. Figure 4.1 goes through part of the script.

```
import torch
from torch.utils.mobile_optimizer import optimize_for_mobile
from mobilenet import Net  # Importing FaHaNa model class

# Load the original .pkl model
model = Net()  # Instantiate the model
model.load_state_dict(torch.load('/path/to/your/FaHaNa-Small.pkl', map_location=torch.device('cpu')))
model.eval()

# Convert the model to a ScriptModule
scripted_model = torch.jit.script(model)

# Optimize the model for mobile
optimized_model = optimize_for_mobile(scripted_model)

# Save the optimized model as a .ptl file
output_model_path = "/path/to/your/optimized_model.ptl"
torch.jit.save(optimized_model, output_model_path)

print(f"Model converted to .ptl and saved as {output_model_path}")
```

Figure 4.1: Conversion to ptl

The development and iterative testing of the mobile application were conducted using Android Studio, a comprehensive development environment that facilitated the coding,

debugging, and simulation of the app. Android Studio's virtual device emulator was particularly valuable for preliminary testing, enabling us to refine the application in a controlled environment before real-world deployment.

```
Net(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU(inplace=True)
  (pool1): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Block(
      (conv1): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=384, bias=False)
      (bn2): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): Block(
      (conv1): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=384, bias=False)
      (bn2): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer11): Sequential(
    (0): Block(
      (conv1): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=384, bias=False)
      (bn2): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer21): Sequential(
    (0): Block_96(
      (conv1): Conv2d(64, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=96, bias=False)
      (bn2): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(384, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```
  (layer3): Sequential(
    (0): Block_bais(
      (conv1): Conv2d(96, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): Block_bais(
      (conv1): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(32, 32, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(32, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu1): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(32, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu1): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      )
    )
  )
  (linear): Linear(in_features=256, out_features=5, bias=True)
)
```

Figure 4.2: Structure of FaHaNa-Small

Following successful emulation and testing within Android Studio, the application was deployed on the designated target mobile devices. This stage of real-world testing was critical for evaluating the app's performance, usability, and reliability, ensuring that the integration of the FaHaNa-Small model met our operational and user experience goals. Through this methodical approach, we were able to effectively integrate advanced machine learning capabilities into a mobile application, showcasing the potential for FaHaNa-Small to enhance mobile computing experiences.

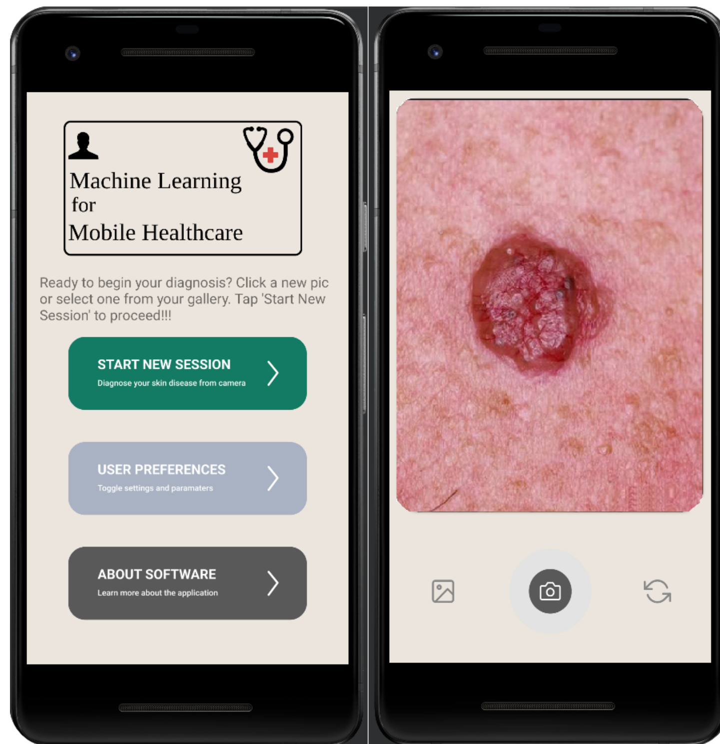

Figure 4.3: Screenshot of Android Application

Figure 4.3 displays the welcome screen of the application, which presents users with three primary options. The first option, 'Start New Session,' allows users to initiate a diagnostic session for skin diseases. The second option, 'User Preference,' enables users to choose between different FaHaNa models. The third option, 'About Software,' provides a detailed description of the application's functionality and purpose.

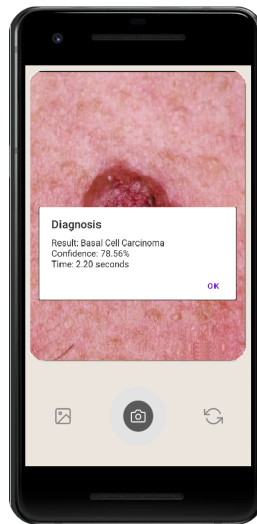

Figure 4.4: Screenshot of Diagnosis Result

Figure 4.4 shows the result screen of the application, which displays the diagnosed disease name, confidence score of the diagnosis, and the time taken. If a diagnosis cannot be determined, the screen displays 'Unknown Carcinoma Type.'

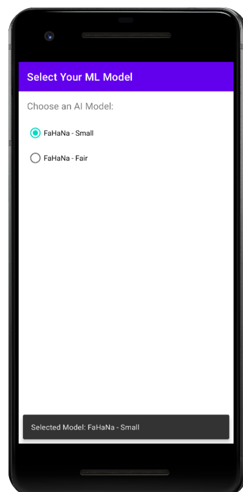

Figure 4.5: Screenshot of User Preference

Figure 4.5 presents the option for users to select between two FaHaNa ML models prior to inference.

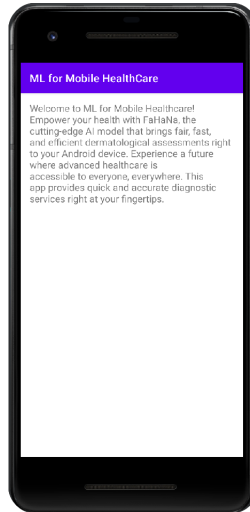

Figure 4.6: Screenshot of About Software

Figure 4.6 displays a brief description of the Android application.

4.3 Performance Evaluation

The performance evaluation of the FaHaNa framework, specifically focusing on the FaHaNa-Small model, was conducted rigorously to assess their efficacy in dermatological disease classification across diverse skin types. The evaluation was centered on a comprehensive analysis that compared these models not only to each other but also against established benchmarks such as MobileNetV2, MnasNet 0.5, MobileNetV3, and MnasNet 1.0. This comparative study aimed to highlight the strengths of the FaHaNa framework in terms of accuracy, fairness, model size, and computational efficiency.

Unfairness Score: In classification models, fairness is assessed by examining the accuracy of predictions across different demographic groups. The *"unfairness score"* quantifies disparities in accuracy across these groups and is calculated using the L1-norm, which sums the absolute differences in accuracy between groups. This score serves as a crucial measure, indicating variations in predictive performance among diverse groups and highlighting any potential biases in the model. By integrating the unfairness score into model evaluation, we aim to ensure that the models are not only accurate but also equitable across all groups.

Table 4.1: Comparison of the existing models and FaHaNa-Models

Model	# of Parameters	Accuracy	Light Dark	Unfairness Score
MobileNetV2	2,230,277	81.05%	81.27% 58.02%	0.2325
MnasNet 0.5	943,917	78.12%	78.54% 33.33%	0.4521
MobileNetV3	1,522,981	80.38%	80.68% 48.15%	0.3253
MnasNet 1.0	3,108,717	80.71%	80.98% 51.85%	0.2913
FaHaNa-Small	422,341	81.28%	81.46% 61.73%	0.1973
FaHaNa-Fair	5,502,469	84.06%	84.22% 66.67%	0.1755

Accuracy and Fairness: FaHaNa-Small achieved an accuracy of 81.28%, slightly outperforming MobileNetV2's 81.05% but with a significantly smaller number of parameters (422,341 vs. 2,230,277), showcasing its efficiency. Moreover, FaHaNa-Small exhibited the highest fairness score, with a stark improvement in accuracy for dark skin types (61.73%) compared to MobileNetV2 (58.02%) and drastically outperforming MnasNet 0.5's 33.33%. This underscores FaHaNa-Small's ability to deliver more equitable healthcare outcomes across different skin types.

Table 4.2: Size comparison of ML Models

Model	Storage (MB)	Latency (ms)	
		Android	RaspberryPi
MobileNetV2	8.51	4264.55	1939.40
MnasNet 0.5	3.60	2312.05	714.19
MobileNetV3	5.81	1954.14	658.84
MnasNet 1.0	11.86	7033.29	3855.72
FaHaNa-Small	1.61	1036.22	347.30
FaHaNa-Fair	21.5	2183.34	648.80

Model Size and Latency: In terms of model size and latency, FaHaNa-Small was the most lightweight model, with a storage requirement of just 1.61 MB. This is significantly lower than the next best model, MnasNet 0.5, which requires 3.60 MB. The latency measurements on edge devices further emphasized FaHaNa-Small's superiority, recording the lowest latency of 1036.22 ms on Android and 347.30 ms on Raspberry Pi. This contrasts sharply with the heaviest model, MnasNet 1.0, which required 11.86 MB of storage and exhibited latencies of 7033.29 ms on Android and 3855.72 ms on Raspberry Pi.

Comparative Analysis: The comparative analysis revealed that FaHaNa-Small not only excels in delivering high accuracy and fairness but also stands out for its compact size and low latency, making it an ideal candidate for deployment on mobile and edge devices. These attributes are crucial for real-time applications in mobile healthcare, especially in resource-constrained settings where computational power and storage are limited.

4.4 Performance Comparison

The comparative performance of FaHaNa-Small against other state-of-the-art small neural networks highlights its innovative approach to balancing the trade-offs between model complexity, computational efficiency, and fairness in AI-driven dermatology applications. By leveraging a smaller parameter set and optimizing for edge devices, FaHaNa-Small facilitates faster, more accessible, and equitable dermatological assessments.

This evaluation substantiates the FaHaNa framework's potential to revolutionize mobile healthcare through AI by ensuring that models like FaHaNa-Small can be seamlessly integrated into mobile applications. Such advancements promise not only to enhance diagnostic accuracy across diverse patient populations but also to democratize access to dermatological care, irrespective of geographical and resource limitations.

In conclusion, the experimental results underscore FaHaNa's capabilities in addressing critical challenges in AI dermatology, particularly for mobile and edge computing applications. Its emphasis on fairness and efficiency, combined with robust performance metrics, positions FaHaNa as a significant step forward in the pursuit of equitable and accessible healthcare solutions powered by artificial intelligence.

Chapter 5: Conclusion

5.1 Conclusion

This thesis explored the application of the FaHaNa model within an Android application, focusing on mobile healthcare. We tackled the challenge of developing neural networks that are fair, accurate, and hardware-efficient for edge devices. The crux of our work involved leveraging the FaHaNa framework, which intelligently navigates the trade-offs between model fairness and computational constraints.

Our findings demonstrate that FaHaNa can efficiently identify neural architectures that excel in fairness and accuracy while being significantly smaller and faster on edge devices compared to existing models like MobileNetV2. This advancement proves pivotal for deploying sophisticated AI models on mobile devices, particularly in healthcare diagnostics, where such qualities are crucial.

The successful application of FaHaNa on Android underscores the potential to democratize healthcare diagnostics across diverse demographic and geographic landscapes. By bringing fair and efficient AI-driven diagnostics to mobile platforms, we envision a future where advanced healthcare solutions are accessible to all, reducing disparities in medical care availability.

In summary, this thesis not only showcases the feasibility of achieving fairness in mobile healthcare applications but also sets a foundation for future innovations in the field. The integration of the FaHaNa framework into Android applications marks a significant stride towards equitable, AI-enabled healthcare, heralding a new era of mobile health solutions that are accessible, fair, and efficient.

5.2 Future Works

Building upon the groundwork laid by this thesis in applying the FaHaNa model to mobile healthcare through Android applications, several avenues for future research and development emerge. These directions not only promise to extend the capabilities and applications of the FaHaNa framework but also to address existing challenges and open up new possibilities for equitable AI-driven healthcare solutions. The potential areas for future work include:

1. Cross-Platform Adaptability: While this thesis focused on Android applications, extending the framework's compatibility to other mobile operating systems like iOS could significantly broaden the reach and impact of AI-driven healthcare solutions. Investigating cross-platform development strategies and optimization techniques will be crucial.

2. Enhanced Model Generalization: Further research could focus on improving the generalization capability of models identified by the FaHaNa framework. This involves developing techniques to ensure that these models perform consistently across diverse and unseen datasets, enhancing their reliability in real-world healthcare applications

3. Collaborative Learning Models: Exploring collaborative learning approaches, such as federated learning, could enhance the FaHaNa framework by enabling models to learn from decentralized data sources without compromising patient privacy.

By addressing these areas, future work can not only enhance the efficacy and applicability of the FaHaNa framework for mobile healthcare but also contribute significantly to the broader goal of achieving equitable, efficient, and accessible healthcare services through AI technologies.

Bibliography

[1] Ibm Data Team, Ai, Ibm Data Team, and Ai. Ai vs. machine learning vs. deep learning vs. neural networks: What's the difference?, Jul 2023.

[2] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[3] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[4] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.

[5] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2820–2828, 2019.

[6] Yi Sheng, Junhuan Yang, Yawen Wu, Kevin Mao, Yiyu Shi, Jingtong Hu, Weiwen Jiang, and Lei Yang. The larger the fairer? small neural networks can achieve fairness for edge devices, 2022.

[7] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2018.

[8] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search, 2019.

[9] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware, 2019.

[10] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[11] Siyi Du, Ben Hers, Nourhan Bayasi, Ghassan Hamarneh, and Rafeef Garbi. Fairdisco: Fairer ai in dermatology via disentanglement contrastive learning, 2022.

[12] Ching-Hao Chiu, Hao-Wei Chung, Yu-Jen Chen, Yiyu Shi, and Tsung-Yi Ho. *Toward Fairness Through Fair Multi-Exit Framework for Dermatological Disease Diagnosis*, page 97–107. Springer Nature Switzerland, 2023.

[13] Yi Sheng, Junhuan Yang, Lei Yang, Yiyu Shi, Jingtongf Hu, and Weiwen Jiang. Muffin: A framework toward multi-dimension ai fairness by uniting off-the-shelf models, 2023.

[14] Dipayan Sengupta. Artificial intelligence in diagnostic dermatology: Challenges and the way forward. *Indian Dermatology Online Journal*, 14:782 – 787, 2023.

[15] Christopher J. Kelly, Alan Karthikesalingam, Mustafa Suleyman, Greg C. Corrado, and Dominic King. Key challenges for delivering clinical impact with artificial intelligence. *BMC Medicine*, 17, 2019.

[16] Arieh Gomolin, Elena Netchiporouk, Robert Gniadecki, and Ivan V. Litvinov. Artificial intelligence applications in dermatology: Where do we stand? *Frontiers in Medicine*, 7, Mar 2020.

[17] Introduction to activities. `https://developer.android.com/guide/components/activities/intro-activities`, 2023. Accessed 10-May-2023.

[18] International skin imaging collaboration dataset. `https://challenge.isic-archive.com/landing/2019/`, 2023. Accessed 10-May-2023.

[19] Dermnet dataset. `https://dermnet.com/`, 2023. Accessed 11-May-2023.

[20] Dermatology atlas. `https://www.atlasdermatologico.com.br/`, 2023. Accessed 11-May-2023.

[21] Raspberry pi 3 model b. `https://www.raspberrypi.com/documentation/`, 2024. Accessed 19-Jan-2024.

Biography

Chiranjivan Krishnakumar Nirmala pursued his Bachelor in *Electronics and Communication Engineering* from Sri Sairam Engineering College, India. Following that, worked as *Software engineer* at Vembu Technologies, India. Chiranjivan is currently pursuing his Master of Science in *Computer Engineering* at George Mason University, Virginia, USA. Chiranjivan's interests are FPGA-based projects, Hardware Accelerators for ML Applications, and CUDA.

Chiranjivan Krishnakumar Nirmala

Fairfax, Virginia | 703-215-6562 | ckrishn@gmu.edu | www.linkedin.com/in/chiranjivan | https://chiranjivan-kn.github.io | @github

Education

George Mason University, **MS in Computer Engineering** *Aug 2022 – May 2024*
Relevant Coursework: *ML for Embedded Systems, HW Accelerators for ML, GPU Architecture and Programming, Learning From Data, ML Security and Privacy, Big Data Technologies, Neuromorphic Computing.*

Anna University, **BE in Electronics and Communication** *Aug 2015 – May 2019*

Technical Skills

Languages	: C, C++, C#, Embedded C, CUDA, Python, VHDL, Verilog, RTL Design, TCL, SQL, Scala
Technologies & Protocols	: Azure, AWS, Hyper-V, HTML, GPIO, ADC, UART, I2C, SPI, MQTT, RTOS, UVM, STA, Clock gating
Tools	: Xilinx Vivado, Quartus Prime, ModelSim, Cube IDE, Cadence Virtuoso, Synopsis Design Compiler
Boards	: STM32, Arduino, RaspberryPi, Jetson Nano, ESP32, Spartan6, pynq z2

Experience

Graduate Teaching Assistant | *George Mason University* *Aug 2023 – Present*
- Graded labs, assignments, and proctored exams for ENGR 107, ECE 554, and ECE 556 courses.

Graduate Research Assistant | *George Mason University* *Summer 2023*
- ML for Mobile Healthcare : (MS–Thesis) Developed a ML based Android app for skin diseases, leveraging ***FaHaNa*** model to eliminate gender and skin-type bias in dermatology.
- Achieved edge device deployment, enhancing the accessibility and accuracy of dermatology assistance.

Software Developer | *Vembu Technologies Pvt.Ltd* *Jun 2019 – May 2021*
- Restore to Azure : Developed a workflow for restoring VMs to Azure Cloud Environment, ensuring ***minimal downtime*** during critical incidents.
- Used tools such as MS Azure Management Fluent SDK for C# and PS scripts and integrated ***CLR to CPP*** for seamless execution.
- Tape Backup : Implemented a robust backup feature for standalone tape drives, utilizing ***Win32 API (winbase.h, winnt.h)*** , which enabled customization of retention policies for backups stored in a Tape Infrastructure, improving data retention and reliability.
- Microsoft AD : Involved in collection and management of MS Active Directory structure data using ***LDAP*** protocols to facilitate backup processes for AD Users, Groups, and OUs.

Intern | *Air Liquide Medical System Pvt.Ltd* *Summer 2018*
- Electronic Board Sub-Assembly & Testing : Contributed to the production and quality assurance process including sub-assembly and testing of the Orion-G (ventilator) and its printed circuit boards.

Projects

Reliable Multiplier Design Implementation on FPGA with Adaptive Hold Logic | *VHDL* *Mar 2019*
- Implemented an Adaptive Hold Logic (AHL) circuit and Error Detection Correction Pulsed Latch (ECPL) to address delay issues stemming from transistor aging effects. - *Published in Journal of Emerging Technologies and Innovative Research (JETIR)*
- Validated using a radix-4 booth multiplier with proposed methodologies on Xilinx Spartan6 FPGA, resulting in improved system performance.

Designing Hyperdimensional Computing Systems with FPGA Technology | *VHDL* *Spring 2022*
- Pioneered FPGA-based Hyperdimensional Computing (HDC) framework, optimizing resource utilization and demonstrating the feasibility of HDC in tasks like image classification using MNIST.

Quantum Gate Simulation Optimization - CUDA Performance Analysis with Memory Configurations | *CUDA* *Fall 2022*
- CUDA program for qubit gate simulations, conducted performance comparisons between Unified Virtual Memory and Separate Host/GPU Memory configurations, and provided insights to optimize quantum gate simulations on GPUs.

Dynamic Instruction Scheduling Simulator | *C++* *Spring 2023*
- Simulated a dynamic instruction scheduling for out-of-order processor, implementing the Tomasulo algorithm with a Reorder Buffer to optimize execution performance.

Cache Simulator | *C++* *Spring 2023*
- Designed and implemented a C++ cache simulator that emulates Level 1 (L1) and Level 2 (L2) cache behavior, analyzes performance metrics (miss rate, memory access time), and employs LRU eviction for various cache configurations and write policies, including tracking Valid, Dirty bits, and Tag fields.

Combating Against SEM IC Reverse Engineering through Localized Adversarial Perturbations | *Python, ART* *Fall 2023*
- Developed a defense mechanism against SEM-based IC reverse engineering by introducing localized adversarial patches using ***Adversarial Robustness Toolbox*** into logic gate GDSII.
- This strategy effectively fools CNNs, strengthening intellectual property protection and HW security.

Leadership Skills

Events volunteer, *PatriotHacks, 2023* **- Multimedia Team,** *IC3IoT, 2018* **- Organizer,** *Zenista 2018*

Licenses & certifications

IBM Qiskit Global Summer School 2023 | *IBM* *Sept 2023*
Scaling Workloads Across Multiple GPUs with CUDA C++, Building Transformer-Based NLP Applications | *NVIDIA* *Nov 2023*