Efficient Deep Learning for Plant Disease Classification in Resource Constrained Environment

by

Ola Mustafa Alqahtani

(Under the Direction of Lakshmish Ramaswamy)

ABSTRACT

Deep Neural Networks (DNNs) have been widely used in today's applications. In many applications such as video analytics, face recognition, computer vision, and classification problems like plant disease classification, etc. DNN models are constrained by efficiency constraints (e.g., latency). Many deep learning applications require low inference latency, which must fall within the parameters set by a service level objective. The prediction of the inference time of DNN models raises another problem which is the limited resources of Internet of Things devices. These devices need an effective way to run DNN models on them. One of the most widely discussed technological developments since the Internet of Things is edge machine learning (Edge ML), and with good reason. Edge Machine Learning is a fast-growing well-known technological improvement since the existence of the Internet of Things (IoT). Edge ML allows smart devices to use machine learning and deep learning techniques to analyze data using servers locally or at the device level, which reduces the need for cloud networks. This is caused by a variety of issues, including poor internet access, expensive cloud resources, low-resource edge devices, and a high failure rate of Internet of Things (IoT) devices, either because of battery or connection issues. Finding a way to effectively run the DNN models locally on IoT devices is crucial.

Keywords— Deep Nural Network, Internet of Things, Edge ML, University of Georgia, Machine Learning, Inference Time, Latency

Efficient Deep Learning for Plant Disease Classification in Resource Constrained Environment

by

Ola Mustafa Alqahtani

B.S., Imam Mohammad Ibn Saud Islamic University, Kingdom of Saudi Arabia, 2010

M.S., Claremont Graduate University, USA, 2016

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA 2024

© 2024

Ola Mustafa Alqahtani

All Rights Reserved

Efficient Deep Learning for Plant Disease Classification in Resource Constrained Environment

by

OLA MUSTAFA ALQAHTANI

Major Professor: Lakshmish Ramaswamy

Committee: Thiab Taha

Khaled Rasheed

Electronic Version Approved:

Ron Walcott
Vice Provost for Graduate Education and Dean of the Graduate School
The University of Georgia
May 2024

ACKNOWLEDGEMENTS

It is a great pleasure to acknowledge my deepest thanks and gratitude to my committee chair, Professor. Lakshmish Ramaswamy, Professor and Associate Director of the School of Computing of The University of Georgia, for guiding me from the inception to the completion of this dissertation, and his kind supervision. It is a great honor to work under his supervision.

I would like to express my deepest appreciation to my committee member, Professor. Khaled Rasheed, Professor of the School of Computing and the Director of the Institute for Artificial Intelligence of The University of Georgia, who has the attitude and the substance of a genius and without his guidance and persistent help this until this work came to existence.

I would like to thank my committee member, Professor. Thiab Taha, Professor of the School of Computing of The University of Georgia, whose work demonstrated to me that concern for his kind endless help, generous advice, and support during the study.

I cannot express enough thanks to my committee for their continued support and encouragement. I offer my sincere appreciation for the learning opportunities provided by my committee.

My completion of this dissertation could not have been accomplished without the support of “My Family”. My father, my mother, and my siblings – thank you for allowing me time away from you to study and research.

I would like to express my special thanks to my brother Ammar Alqahtani, who was the best brother I could have ever asked for. You made my PhD years so much fun, and always had my back even though we are both thousands of miles away from our family. You were the family to me.

I would like to thank all the employees of the Saudi Cultural Mission and Prince Sattam bin Abdulaziz University, for provided me a huge amount of their precious time and effort to me. I feel really lucky to be able to work under their direction.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS....iv

LIST OF TABLES.... vi

LIST OF FIGURES vii

CHAPTER 1: INTRODUCTION

1 PLANT DISEASES CLASSIFICATION 1

2 RESEARCH CONTEXT AND MOTIVATION 2

3 RESEARCH OBJECTIVES AND CONTRIBUTION 5

4 RESEARCH ORGANIZATION 12

CHAPTER 2: BACKGROUND AND RELATED WORK

1 INTRODUCTION 13

2 EDGE-BASED MACHINE LEARNING SYSTEMS 14

3 INFERENCE LATENCY OF DEEP LEARNING MODELS 25

4 EXISTING APPROCHES AND LIMITATIONS 33

CHAPTER 3: HARNESSING DEEP LEARNING ON IOT TO CLASSIFY TOMATO PLANT DISEASES

1 CHAPTER OVERVIEW 50

2 MODEL DESCRIPTION....51

3 EXPERIEMNTAL SETUP 54

4 RESULTS....56

5 CHAPTER SUMMARY....63

CHAPTER 4: INFERENCE TIME PREDICTION OF DEEP LEARNING ARCHITECTURES FOR PLANT DISEASE CLASSIFICATION

1 CHAPTER OVERVIEW 65

2 PROPOSED SOULATION....66

3 METHODOLOGY 76

4 RESULTS.... 83

5 CHAPTER SUMMARY....87

CHAPTER 5: MODEL COMPRESSION FOR EFFICIENT DEEP NEURAL NETWORKS ON PLANT DISEASE CLASSIFICATION

1 CHAPTER OVERVIEW 88

2 CHALLENGES AND LIMITATIONS 89

3 MODEL COMPRESSION ……………....90

4 PROPOSED SOLUTION....107
5 EXPERIEMNTAL RESULTS....109
6 DISCUSSION....120
7 CHAPTER SUMMARY....121
CHAPTER 6: CONCLUSION 123
1 FUTURE DIRECTIONS....126
REFERENCES 132

LIST OF TABLES

Page

Table 1: [Inference Time Estimation Techniques on Mobile Devices Advantages/Disadvantages] 47

Table 2: [The selected features] 53

Table 3: [The Coloration between each Parameter and Inference Time] 76

Table 4: [Lenovo ThinkPad L14 Gen 2] 80

Table 5: [Software Requirements] 82

Table 6: [Inference Time Per Image Vs. Batch Size of our CPU] 83

Table 7: [Inference Time with Top 5 Models] 84

LIST OF FIGURES

Page

Figure 1: [Overview of method] 51

Figure 2: [Prediction model architecture includes 3 KNN Models each predicts either use one of the three image classification models] 51

Figure 3: [Accuracy comparison of multiple machine learning techniques for constructing our prediction model] 52

Figure 4: [Tabular output from the Nvidia-smi command showing details of the NVIDIA Tesla T4 GPU] 55

Figure 5: [Accuracy and our approach performance against individual models] 56

Figure 6: [Inference time and our approach performance against individual models] 57

Figure 7: [Memory usage and our approach performance against individual models] 58

Figure 8: [Throughput (No.of. Images/s) and our approach performance against individual models] 58

Figure 9: [Comparable modeling techniques for Prediction Model Priority is Accuracy] 59

Figure 10: [Comparable modeling techniques for Prediction Model Priority is Inference Time] 60

Figure 11: [Comparable modeling techniques for Prediction Model Priority is Memory Usage] 60

Figure 12: [Features Correlation against different indices when priority (accuracy, inference time)] 61

Figure 13: [Features Correlation against different indices (Memory usage, Throughput)] 61

Figure 14: [Feature size and accuracy impact when priority is accuracy] 62

Figure 15: [Feature size and accuracy impact when priority is memory usage] 62

Figure 16: [Feature size and accuracy impact when priority is inference time] 63

Figure 17: [System Architecture] 66

Figure 18: [Coloration Matrix of All Selected convolutional parameters used to predict Inference Time] 74

Figure 19: [Coloration Matrix of All Selected Dense parameters used to predict Inference Time] 75

Figure 20: [Data Distribution in Case of Backpropagation or No Backpropagation] 78

Figure 21: [RMSE of Inference Time Predictions and Number of FLOPS for Convolutional Layers on CPU Using Models with Different Numbers of Hidden Layers] 84

Figure 22: [Predicted Inference Time for Convolutions on CPU Versus the Measured Time for the Deep Neural Network Model of a Forward Pass Only] 85

Figure 23: [Predicted Inference Time for Convolutions on CPU Versus the Measured Time for Linear Regression Model for a Forward Pass Only] 85

Figure 24: [The common groups of DNN model compression techniques] 91

Figure 25: [Generalized visualization of structured pruning] 93

Figure 26: [Generalized visualization of unstructured pruning] 95

Figure 27: [Generalized loss approach design to knowledge distillation] 102

Figure 28: [Pruning compression techniques for trained Resnet18 Model on Tomato Plant Diseases] 111

Figure 29: [Pruning compression techniques for trained Googlenet Model on Tomato Plant Diseases] 112

Figure 30: [Pruning compression techniques for trained Mobilenet Model on Tomato Plant Diseases] 112

Figure 31: [Knowledge Distillation compression techniques for trained Resnet18 Model on Tomato Plant Diseases] 115

Figure 32: [Knowledge Distillation compression techniques for trained Googlenet Model on Tomato Plant Diseases] 115

Figure 33: [Pruning compression techniques for trained Mobilenet Model on Tomato Plant Diseases] 116

Figure 34: [Quantization compression techniques for trained Resnet18 Model on Tomato Plant Diseases] 118

Figure 35: [Quantization compression techniques for trained Googlenet Model on Tomato Plant Diseases] 119

Figure 36: [Quantization compression techniques for trained Mobilenet Model on Tomato Plant Diseases] 120

CHAPTER 1

INTRODUCTION

One of the trends we're seeing in food and agriculture is more and more consumers wanting to know things about their food and where and how it's grown and what's in it.

— Dan Glickman [1998]

Tomato Plant Diseases Classification

A key vegetable crop that boosts the world economy is the tomato. The red tomato, a tasty fruit, originated on the Americas continent. Tomatoes are attacked by a variety of plant diseases, including mosaic virus, late blight, and leaf mold. The treatment and complete control of viral infections are generally challenging, even with excellent plant management efforts. Thus, prompt, and accurate identification of plant diseases is necessary in order to address the problem at an early stage.

In the world of agriculture, the identification and classification of plant diseases are extremely important. The timely identification of diseases is essential to the health, quantity, and productivity of the plants.

DNN model inference can be accelerated by offloading computing to the cloud, although this is not always possible due to latency or connectivity concerns. A latency problem is raised when offloading DNN inference to the cloud. Our method of model selection enables one to choose a model based on the input. Hence, a locally designed automated system on an IoT device is required in order to make the detection process autonomous with the least amount of human involvement [104].

Although current deep learning systems frequently require a significant amount of CPU resources to operate, image classification tasks are crucial for IoT devices, particularly for sensing and diseases detection applications on smart farms. Even for straightforward operations, running these models on IoT devices can result in long runtimes and a significant resource usage, including CPU, memory, and electricity [120]. The efficient implementation of machine learning (ML) algorithms is the goal of several projects. However even though they are frequently the most time-consuming processes for analysts, three crucial difficult and iterative practical tasks in employing ML—feature engineering, algorithm selection, and parameter tuning—have largely been ignored by the data management community.

Compressing the model to lower its resource and computational requirements is a typical technique for accelerating DNN models on IoT devices, however, doing so reduces precision [108]. Other approaches involve offloading some, or all, computation to a cloud server. Due to latency, a quick, reliable network connection is not always guaranteed, so this is not always possible [102].

This work proposes a novel runtime strategy for DNN model selection on IoT devices that aims to reduce inference time while still satisfying user needs. In order to choose the best model to apply at runtime, we use machine learning to automatically create predictors. On the Nvidia system administration interface [105] referred to as (NVSMI).

Our method will support farmers in early illness diagnosis and enable them to take mitigation actions before it's too late. Using three convolution neural network architectures and a probabilistic measure model selection strategy, we use deep learning to identify the diseases [128]. For a given input, our system will guarantee the best DNN model to use, which will be faster and use less power on IoT devices.

Dissertation Context and Motivation

Farmers get extensive information about every aspect of their operation from drones, and remote sensors, such as soil moisture, nutrition levels, harvest information, and more. In smart farms, plant stress, disease, and insect infestation can all be identified using crop photos taken by robots, and drones. Machine learning based image classification is a scalable way to identify plant

diseases. These models can help us to recognize healthy plants by comparing data points from millions of photographs. Farmers, urban planners, and data scientists are investigating new ideas on how data can be used to assist make better decisions while using less energy and resources. It is frequently impractical to shift tasks to the cloud in smart farms.

This is caused by a variety of issues, including poor internet access, expensive cloud resources, low resource edge devices, and a high failure rate of Internet of Things (IoT) devices, either because of battery or connection issues. Finding a way to effectively run the DNN models locally on IoT devices is crucial. We present an efficient ML-based system to classify tomato plant leaf diseases. Our system is specifically optimized to run on local IoT device coupled with "NVSMI". A unique feature of our system is that it dynamically chooses the most appropriate ML model considering the tradeoff between inference time, accuracy, memory usage, throughput, and battery consumption. We demonstrate the efficacy of our approach through a series of experiments using a real-world dataset. In recent years, deep learning models have been widely adopted in lots of fields. such as computer vision, pattern recognition, and classification problems like plant disease classification.

Due to the large diversity among the computing devices that these models may run on, we need to choose between the appropriate device based on cost and performance. Furthermore, finding the suitable optimal device for a given project is a complex process that needs significant time and resources. Prediction of inference latency DNN models is necessary for many tasks where measuring the latency on real devices is either infeasible or too costly. This is a very challenging problem, and most existing approaches fail to achieve high accuracy of prediction. While some research has been carried out to predict the inference time of DNN models – most existing techniques assume that training time is linearly related to the number of floating-point operations. This paper designs and develops a framework to predict the inference time for deep learning models and is generic to be easily extended for a large set of devices. Our key idea is decomposing a given model inference into layers and conducting layer-level prediction. Our experiments demonstrate that this strategy provides significant benefits in terms of prediction accuracy.

Deep Neural Networks (DNNs) have been widely used in today's applications. In many applications such as video analytics, face recognition, computer vision, and classification problems like plant disease classification, etc. DNN models are constrained by efficiency constraints (e.g.,

latency). Many deep learning applications require low inference latency, which must fall within the parameters set by a service level objective. The prediction of the inference time of DNN models raises another problem which is the limited resources of Internet of Things devices. These devices need an effective way to run DNN models on them.

Finding a way to effectively run the DNN models locally on IoT devices is crucial. Presently, due to the expeditious advancement of deep learning, deep neural networks (DNNs) have been extensively utilized in diverse computer vision endeavors. Nevertheless, in the quest for improved performance, sophisticated deep neural network (DNN) models have grown increasingly intricate, resulting in a substantial consumption of memory and significant computational requirements. Consequently, the use of these models in real-time scenarios poses challenges. In order to tackle these concerns, the research community has directed its attention into the field of model DNN compression.

In addition, the utilization of model compression approaches holds significant importance in the deployment of models on edge devices. We conducted an analysis of diverse techniques for model compression with the aim of aiding researchers in the reduction of device storage capacity, acceleration of model inference, simplification of model complexity and training expenses, and enhancement of model deployment. Therefore, we provide a comprehensive overview of the latest advancements in model compression methods. These methods encompass model pruning, parameter quantization, low-rank decomposition, knowledge distillation, and lightweight model design. Furthermore, we evaluated the beforementioned approaches of DNN compression on practical examples and analyzed the obtained results.

The motivation for our approach is to fill the gap by providing a prediction framework that can deal with these non-linearities while also generalizing to previously unseen neural network models or hardware. We also want to match out the limitations of the computation resources of our hardware to provide good accuracy and avoid any failure during running the model. In our approach, we train a deep learning network to predict the inference time for parts of a deep learning network. Timings for these individual parts can then be combined to provide a prediction for the whole model. Our approach reduces the computational time while maximizing the types of layers that we can predict by employing atomic operations. The proposed approach can inform choices

when selecting appropriate hardware for training or inferring models, as well as aiding in the decision-making of model design.

Dissertation Objectives and Contribution

The goal of the majority of the growing number of Internet of Things projects is cost reduction. The present limitations of edge devices impose constraints on the ability to do real-time inference of deep learning models for applications. Moreover, the act of sending data via a network necessitates a greater expenditure of energy compared to local processing, mostly due to the high cost associated with network transmission.

This dissertation has the following contributions, introducing a novel machine learning-based method for automatically learning how to choose DNN models depending on input and precision requirements, and developing a machine learning-based running locally on IoT device with higher speed and lower power consumption by considering the model inputs and requirements solution to classify tomato plant leaf diseases.

It is challenging to create DNNs for mobile platforms since many good DNNs require a lot of computing, and most mobile devices have computational, storage, and power limitations. The influence of efficient deep-learning algorithms on distributed systems, embedded devices, and field-programmable gate arrays (FPGAs) for artificial intelligence has been well recognized in the literature [256]. As an illustration, for instance, one of the most well-known DNNs for image classification, AlexNet [257], needs 724M MACs and 61M weights to analyze a single image. Downloading those DNN-powered applications to local devices may become prohibitively expensive. If used regularly, these DNN-based applications can potentially quickly deplete the battery. Based on those findings, in this dissertation, we utilize some of the most common compression techniques in the experiments. in order to try to address the aforementioned problems of running the interference on low-power IoT devices.

We successfully got three models with a very small inference time. After that we train and test these three models on image classification problem for tomatoes plant diseases. Since our IOT device has limited storage and computational resources, we decided to design a selective model to select between these three models based on different criteria such as accuracy, inference

time, memory usage and throughput, that is Chapter 3. Finally, in Chapter 5, we decided to use model compression techniques to reduce the size of these three models which will save memory and reduce inference.

Reduced computer costs during inference are compression's main advantage. The primary motivation behind model compression is the decrease of computational resources. Model compression decreases disk storage, memory, and CPU/GPU consumption. It can make a model that would have previously been too expensive, too slow, or too big suitable for manufacturing.

Our work proposes a novel runtime strategy for DNN model selection on IoT devices that aims to reduce inference time while still satisfying user needs. In order to choose the best model to apply at runtime, we use machine learning to automatically create predictors. On the Nvidia system administration interface [105] referred to as (NVSMI).

Our method will support farmers in early illness diagnosis and enable them to take mitigation actions before it's too late. Using three convolution neural network architectures and a probabilistic measure model selection strategy, we use deep learning to identify the diseases [128]. For a given input, our system will guarantee the best DNN model to use, which will be faster and use less power on IoT devices. We select the best model feature values depending on the input and number of evaluation criterion such as inference time, accuracy, memory usage, throughput, and battery requirement. It is a novel runtime strategy for DNN model selection on IoT devices that aims to reduce inference time while still satisfying user needs. Hence, determining which model to use is non-trivial. What we need is a technique that can automatically choose the most efficient model to use for any given input using NVSMI [105].

Dissertation Organization

The following five chapters represent the remaining part of the dissertation. Chapter 2 presents an analysis of relevant background and related work materials. Chapter 3 focuses on deep learning and IoT to classify tomato plant diseases. In chapter 4, we show how inference time prediction of deep learning architecture can help in classify plant diseases effectively.

Chapter 5 will be towards model compression for deep neural networks. Chapter 6 of the dissertation contains an overall conclusion.

In Chapter 4, we successfully get three models with a very small inference time. After that we train and test these three models on image classification problem for tomatoes plant diseases. Since our IOT device has limited storage and computational resources, we decided to design a selective model to select between these three models based on different criteria such as accuracy, inference time, memory usage and throughput, that is Chapter 3. Finally, in Chapter 5, we decided to use model compression techniques to reduce the size of these three models which will save memory and reduce inference.

CHAPTER 2

BACKGROUND AND RELATED WORK

Introduction

Deep learning systems frequently require a significant amount of CPU resources to operate, image classification tasks are crucial for IoT devices, particularly for sensing and diseases detection applications on smart farms. Even for straightforward operations, running these models on IoT devices can result in long runtimes and a significant resource usage, including CPU, memory, and electricity [120]. The efficient implementation of machine learning (ML) algorithms is the goal of several projects. However, even though they are frequently the most time-consuming processes for analysts, three crucial difficult and iterative practical tasks in employing ML—feature engineering, algorithm selection, and parameter tuning—have largely been ignored by the data management community.

Recent ground-breaking developments in deep neural networks (DNNs) make them appealing for IoT devices. Machines are increasingly better at handling complicated tasks like object recognition [112], facial recognition, audio processing [97], and image classification due to recent developments in deep learning [98].

Our method will support farmers in early illness diagnosis and enable them to take mitigation actions before it's too late. Using three convolution neural network architectures and a probabilistic measure model selection strategy, we use deep learning to identify the diseases [128]. For a given input, our system will guarantee the best DNN model to use, which will be faster and use less power on IoT devices.

Compressing the model to lower its resource and computational requirements is a typical technique for accelerating DNN models on IoT devices, however, doing so reduces precision. Other approaches involve offloading some, or all, computation to a cloud server. Due to latency, a quick, reliable network connection is not always guaranteed, so this is not always possible [102].

Our work proposes a novel runtime strategy for DNN model selection on IoT devices that aims to reduce inference time while still satisfying user needs. In order to choose the best model to apply at runtime, we use machine learning to automatically create predictors. On the Nvidia system administration interface referred to as (NVSMI).

On the Nvidia simulator, we propose performing a Tomato plant diseases classification [105]. We evaluate the performance of the Googelnet, ResNet, and MobileNet convolutional neural network designs [103]. These models were all trained using the Plant Village dataset [94] and are all based on TensorFlow. For inference, the GPU is utilized. Our research demonstrates that the input and assessment criterion affect the best model feature values. It is therefore not simple to choose which model to apply. What we desire is a method that can select the most effective model to apply for any given input automatically.

Edge-Based Machine Learning Systems

There has recently been a boom in the number of AI-powered goods and services entering the market. In recent years, advances in computer power, storage options, machine learning technology, and artificial intelligence have all contributed to the rapid development of machine learning technology. In fact, if we peek behind the curtain, we may find several instances of ML technology being used today in a variety of sectors, from manufacturing and financial services to consumer goods and social media. But it still begs the issue of how, in such a short time, ML went from science fiction to reality. After all, Arthur Lee Samuel, a data scientist, only succeeded in creating a computer program that could teach itself how to play checkers in 1959. Let's map the evolution of machine learning to identify the solution by looking at the past, present, and potential future developments.

The development of ML can be linked to a series of pivotal events that occurred in the 1950s, when ground-breaking research demonstrated that computers are capable of learning. The English mathematician Alan Turing created the renowned "Turing Test" in 1950 to evaluate whether a machine demonstrates intelligent behavior equivalent to or similar to that of a human. In 1952, Arthur Lee Samuel, a data scientist, was able to instruct an IBM computer program to learn the game of checkers and get better every time it plays. Then, in 1957, the first neural network was ever made for computers by Frank Rosenblatt. Experimentation increased after that. Probabilistic interference such as Bayesian techniques in machine learning was first presented in

the 1960s. And in 1986, the machine learning community was exposed to the deep learning technique by computer scientist Rina Dichter, which is based on artificial neural networks.

Utilizing a Data-Driven Strategy

The knowledge-driven approach of machine learning transmission to the data-driven approach didn't exist until the 1990s. Scientists began developing computer algorithms that could analyze a huge quantity of data and draw conclusions from the findings. Kernel approaches for algorithm pattern analysis, including Support Vector Clustering, gained popularity in the 2000s. During this time, support vector machines became very popular as well as recurrent neural networks.

The next major advance that helped ML become what it is today was the advancement of technology in the early 2000s. The creation of graphics processing units for embedded devices has dramatically accelerated algorithm training from weeks to days. A potent deep neural network that could identify untagged YouTube cat photos was developed in 2009 using Nvidia GPUs. A new paradigm of edge ML for software services and applications may begin if deep learning proves to be practically feasible.

As businesses from a variety of industries use their data and collect the rewards of machine learning, the demand for GPUs is now on the rise. Targeted advertising, machine maintenance prediction, and medical diagnostics are some examples of current applications of machine learning. However, there is a particular roadblock that is impeding development when it comes to using ML models in the real world. And such difficulty is referred to as latency.

More enterprises are storing their data in the cloud. This implies that before the final results can be delivered again to the edge device, model comparison data must be moved from the edge device to the cloud. Sometimes, thousands of kilometers separate clouds. When time is of the essence, as it is in fall detection, this is a major and potentially dangerous concern. Due to latency difficulties, many enterprises are increasingly switching to the edge. With Intelligence on the Edge, Edge AI, or Edge ML, data is processed locally by algorithms stored on a physical device or edge rather than by algorithms maintained in the cloud. With real-time operations made possible, electricity consumption and security hazards when processing data in the cloud are also greatly diminished.

There are some issues with Edge ML Power Constraints, as efforts are made to deploy Edge ML and AI to ever-smaller devices and wearables, resources are limited. How can we use Edge ML applications while preserving precision and speed. To scale to the small models now on the market that fit microprocessors, learning (ML) models still require a lot of memory and processing power. By creating complex hardware, software, and algorithms, or by deftly combining these components, this issue is attempted to be addressed.

As new advancements and landmarks are made in the present, the future of ML is constantly changing. That makes it difficult to make precise predictions, but we can still spot some major trends.

The following benefits are provided by the edge-cloud architecture:

Reliability

Edge-Cloud systems are particularly reliable because of redundancy and global data access. Modern edge systems can be highly flexible because of remote edge device administration. To save edge nodes in the event of fail issues or when hardware has to be fixed or replaced, some systems build cloud devices two times of the particular edge devices [132], [133].

Scalability

Continuously adding or removing edge devices from the network is made possible by the seamless scalability of edge nodes. Scalable deployments of real-world systems are possible when more edge devices sign up [134]–[136].

Computational Expenses

The ability to reduce computing expenses is a significant benefit. Particularly for computationally demanding jobs like computer vision using neural networks (NNs) and processing resources in the cloud are still quite expensive. Although there are few resources available at the edge device level, processing data locally is far more cost-effective and results in sending a lot less data to the cloud. In any event, cloud computing requires network access, relies on external security, raises privacy issues because of data transfer and storage, and has a longer latency than local computing. Edge computing, on the other hand, offers workable, scalable, low-latency computing [137]– [139].

Updated Privacy and Security

Edge ML is the only way to meet updated business needs, such as those requiring extra privacy, high security applications such as public safety organizations, government, and situations where network connectivity is constrained or temporarily unavailable for example intellectual property and privacy laws [140], [141].

Smart and Intelligent Systems

The largest retailers, e-commerce companies, telecoms, and logistics companies in the world are moving beyond machine learning (ML) in order to create and manage smart intelligent systems. Better profits, a route to a better customer experience, and an increase in operational efficiency are all made possible by AI-driven management, intelligent video analytics, and customer and store analytics. Typical edge devices that could be modeled using ML include Intel NUCs and SoC PCs, both of which are well-known embedded computing platforms. Since there is no predetermined form factor on Intel or ARM platforms, edge computers can be edge servers, mobile devices, or other desktop computing devices with standard or embedded hardware [142]–[144].

Powerful Integration

The potent pairing of Jetson GPUs with NVIDIA T4 at the edge for intelligent video analytics and machine learning (ML) applications can be used for any application [152], [153].

Cutting-Edge Services

Edge ML enables telecom companies to provide greater client care and AI solutions, generating new revenue streams [154]– [156].

Some of possible Areas of Edge ML in the Future, Amazon's Echo and Google's Home are just two examples of smart speakers that are already supported by Edge ML. Some businesses in the energy and industrial sectors have created Edge ML systems with anticipatory sensors and algorithms that track the health of the parts and alert workers when repair is necessary. Other Edge ML systems keep an eye out for situations like equipment meltdowns or failures. Future healthcare and assisted living facilities may implement Edge ML-based systems to keep an eye on things like patient heart rates, blood sugar levels, and falls (using radar sensors, cameras, and/or motion sensors). These technologies have the potential to save lives, and if the

data is analyzed locally at the edge, staff members would be alerted in real-time when immediate action is required to save lives.

Sustainable Edge ML

The spectrum of potential solutions for developing very sustainable systems has increased with the usage of Edge ML applications. They produce cost-efficient, intelligent, mobile, low-power technology. Advanced supply chains, sustainable distributed energy grids, environmental monitoring, agricultural applications, and weather and disaster predictions are just a few of the applications that Edge ML can be used for. An estimated 200-terawatt hours of power are used by data centers every year, which is more than some nations actually require. Furthermore, it's estimated that they produce 2% of global CO2 emissions. Therefore, Edge ML can aid in lowering data center power usage.

Unsupervised Machine Learning

The bulk of AI and ML programs dedicate the majority of their development time to classifying and sorting data. According to the research firm Coanalytic, the time spent on an average AI project (80%) is spent obtaining, filtering, categorizing, and enhancing data for inclusion in ML models. Unsupervised learning can be so revolutionary because of this. Extra machines will be able to recognize discovered patterns in unlabeled, uncategorized data sets all alone. When it is unclear what the desired outcome will be, unsupervised learning is particularly helpful for identifying previously undetected patterns in data collecting. Applications like evaluating consumer data on edge devices to discover target audiences for new products or spotting data abnormalities like fraudulent transactions or corrupted hardware can both benefit from this.

At-the-Edge Hardware Acceleration

Semiconductor firms and startups try to make edge machine learning workloads—from training to inference on the edge path—faster, less expensive, and more energy efficient. A brand-new category of dedicated accelerator is starting to take shape. These accelerators promise a huge increase in performance for edge hardware and edge machine learning systems. For example, by relieving the CPU of edge devices from the challenging and taxing mathematical work needed to run deep learning models. This is crucial to prepare for quicker projection. Many

organizations are creating Edge ML chips designed for edge performance and low power consumption, including Arm, GreenWaves, Syntiant, and many others.

Expanding Edge ML

As the cost of AI and machine learning technologies continues to decline, the Internet of Things will eventually affect more elements of our daily life. But the infrastructure we already have must be able to support the increasing number of edge ML devices.

In a recent post, it is stated that one trillion IoT devices are anticipated to exist by the year 2035, bringing considerable infrastructure and design challenges [131]. Our technology must progress if we are to keep up. It suggests that Arm will continue to invest heavily in developing the tools, software, and hardware required to support rational decision-making across the whole infrastructure stack at the forefront of edge computing. Furthermore, it makes use of heterogeneous computation across the network, not just at the processor level, but also at the cloud, edge, and endpoint devices levels.

Key Components of Edge ML

The three most crucial components of an intelligent application are said to be data, model, and computation [126]. If an intelligent application were a "person," the "body" would be a model, and the "heart" would be computation. The "book" is then data. The "person" develops their skills by absorbing information from the "book." After learning, the "human" begins to use the new information. In accordance, the majority of intelligent applications' full deployment consists of three parts: data collection and administration (creating the "book"), training (learning), and inference [169]. Unsupervised learning-based applications are not included [127]. A fourth component that is not visible but is crucial to the other three is computation. These three obvious components become edge cache (data storage at the edge), edge training (training at the edge), and edge inference (inference at the edge), respectively, when combined with an edge environment. Keep in mind that edge computers and edge servers are typically weak. Offloading is typically used to do computation at the edge. The hidden component becomes edge offloading as a result (computation at edge). Therefore, Edge ML's classification of data is based on these four components. Each component's key development concerns are listed, and these

considerations are then divided into a number of smaller ones to provide a step-by-step explanation.

Edge Caching

In order to provide intelligent applications to consumers at the edge, edge caching refers to distributed data systems that are located close to clients and then collect (store) data from the internet, edge devices, and surroundings. BS caches maintain data that the client has lately acquired from the Internet in order to identify the user's interest more accurately in suggested applications [183]. In order to develop edge caching, the following three questions are addressed: what, where and how to cache. Edge devices to macro and micro-BSs have fewer resources and more mobility. Therefore, the issue of caching on a single edge device receives less attention. For instance, the issue of what to cache based on communication and computation redundancy in some specialized applications, such as computer vision, is studied in [184]–[186]. The majority of researchers employ edge device collaboration for caching, especially in networks with many users. The caching problem is often formulated as an optimization issue including content replacement [187]–[190], association policy [191]–[193], and incentive mechanisms [194], [195]. Content replacement must be considered since the storage capacity of macro-BSs, micro-BSs, and edge devices are constrained. Studies on this issue concentrate on developing replacement strategies, such as popularity-based schemes, and ML-based schemes to maximize service quality [168], [196], [197].

Edge Training

Edge training is a decentralized training algorithm that finds the optimal weights and biases, or hidden patterns, using an edge-cached training set. For instance, Google developed the G-Board intelligent input tool, which leverages previous input to learn usage patterns and more precisely forecast the user's upcoming input [182]. As opposed to conventional centralized training methods on powerful servers or compute clusters, edge training is often conducted on edge servers or edge devices that are less powerful than the central servers or compute clusters. Thus, it must consider four major issues in addition to the challenge of the training sample (Caching) when using edge training.

- How to learn (the learning framework).
- How to accelerate the learning.
- How to optimize the learning.
- How to measure the unpredictability of the model's outcome.

Edge training is substantially slower than centralized training paradigms, where strong CPUs and GPUs might provide a decent output with little training time. The acceleration of edge training is a topic of interest for several academics. Works on training acceleration are split into two groups in accordance with the training architecture: acceleration for collaborative training [200], [201] and solo training [198]–[199].

Collaborative training refers to numerous devices that work together to train a single model or algorithm, as opposed to solo training, which is conducted on a single device alone. The majority of the literature now in existence focuses on collaborative training designs since solo training puts a larger demand on the hardware, which is frequently unavailable.

Solo training is a closed system in which obtaining the ideal parameters or patterns only requires iterative calculation on a single device. Collaborative training, in contrast, relies on the collaboration of several devices and necessitates recurrent communication for updating. Update frequency and update cost are two variables that impact training effectiveness and communication success. Researchers in this field mostly concentrate on how to retain the performance of the model or algorithm with reduced update frequency and cost because collaborative training is public, and malevolent people can exploit it. There is also some literature that focuses on security [202]–[205] and privacy problems. The output results of DL training could be mistakenly regarded as model confidence. While estimating uncertainty is simple for traditional intelligence, edge training requires a lot of resources. Some literature draws attention to this issue and makes various suggestions for how to cut back on computation and energy use [206], [207].

Edge Inference

Edge inference is the process of computing the output on edge servers and devices using a trained model or algorithm to infer the testing instance in a single forward pass. As an

illustration, programmers created a face verification application based on DL and used on-device inference [208], [209], which achieves high accuracy and cheap computing cost.

The majority of edge ML models now in use are created with strong CPUs and GPUs in mind; hence, they cannot be used in edge environments. Therefore, the key challenges of using edge inference are how to develop new models or compress old models so that they can be deployed on edge devices or servers, and how to speed edge inference so that real-time results can be obtained.

In order to construct models that are appropriate to edge situations, researchers generally concentrate on two research paths. Creating new models and methods that are more naturally suited to edge contexts and have lower hardware requirements, as well as minimizing operations during inference by compressing current models and deleting superfluous models. In the first path, there are two ways of developing a new model. In order to find the optimal model to construct a human-invented framework [215]–[217], there are group convolution [210], [211] and depth-wise separable convolution [212]–[214], or let the computer use architectural search.

In the second technique, known as model compression, scientists concentrate on thinning and condensing existing models in order to produce smaller, more potent, and computationally efficient models with minimal to no accuracy loss. Five popular techniques for model compression are Low-rank approximation [218]–[220], knowledge distillation [221], [222], compact layer design [223]–[225], network pruning [80]–[82], and parameter quantization [44]–[47]. Like edge learning, edge devices and servers are less potent than central servers and computer clusters. Therefore, edge inference takes longer.

Edge Offloading

A key element of edge ML is edge offloading, a distributed software framework that provides computer service for all previous edge components caching, training, and inference. If a single-edge device is poorly equipped to handle a certain edge ML application, it may delegate those tasks to edge devices and servers. The edge offloading layer allows seamless computing services to the other edge ML components. The most important aspect of edge offloading is the offloading strategy, which should carry out the tasks at the edge environment to the fullest extent possible.

Distributed cloud servers, edge servers, and edge devices are the available computing resources. Accordingly, the majority of the literature now in circulation focuses on four techniques: hybrid offloading, device-to-device (D2D) offloading, device-to-edge server (D2E) offloading, and device-to-cloud (D2C) offloading. Studies on the D2C offloading approach [48]–[50] pre-processing jobs on edge devices and offloading the remaining duties to a cloud server, which might drastically reduce the quantity of uploaded data and latency. This technique is also used by working on the D2E offloading approach [51], [52], which might further minimize latency and cellular network dependence.

The majority of D2D offloading method studies [53], [54] concentrate on smart home situations where smartphones, smartwatches, and IoT devices work together to accomplish training and inference tasks. The strongest ability of adaptability, which makes the most of all the resources available, is found in hybrid offloading schemes [55].

The rapid development of intelligent methods, including deep learning and machine learning techniques, provides the theoretical foundation for the usage of edge intelligence [56]–[58]. Autonomous piloting, speech recognition, and behavior prediction are just a few of the fields where intelligent techniques are leading the way. Our lifestyle has significantly changed as a result of these efforts. People desire to have access to smart services wherever they are and whenever they want. Clients find it bothersome that the bulk of smart applications now available are cloud-based. For instance, more and more people are using voice assistants like Apple Siri on their smartphones. But in order to work, these applications need networks.

Choosing the Right Application based on Edge ML

Machine learning can be implemented on any computing device using contemporary co-processors and AI accelerators. While there are big disparities between edge devices in terms of computational power and technological design, recent advancements have proven that AI model optimization is now feasible, leading to significantly smaller neural networks with significantly greater effectiveness. Certain sorts of models are intended for use with low-power edge devices such as TensorFlow Lite [158]. Due to the wide range of deep learning models, their frameworks and application architectures, it is very difficult to compare the cost and performance of different edge devices. Now, expensive GPUs can be replaced by scalable, affordable technologies that cost a lot less and offer the same performance. For computer vision applications, it is essential to

choose a platform that permits cross-architecture deployments that can be upgraded to new hardware. Technological agility will have a substantial impact on the utility and responsiveness of edge AI systems in the future.

Edge ML in Cloud Environment

As a development of cloud computing, edge ML brings cloud services closer to clients. To provide computing, storage, and networking resources that are often found on the network edge, edge computing offers virtual computing and ML platforms [129], [130], [157]–[159].

Edge servers, which can include IoT gateways, routers, and tiny data centers in mobile network base stations, on automobiles, among other places, are the devices that deliver services for end devices. Edge devices such as mobile phones, Internet of Things (IoT) devices, and embedded devices ask edge servers for services. Three factors might be used to summarize the edge ML paradigm's key benefits in a cloud environment.

I. Extremely low latency: processing typically occurs close to the data sources, reducing the amount of time needed for data transfer. Responses from edge servers are delivered to end devices very instantly [160]–[162].

II. Energy conservation for end devices: By delegating computational workloads to edge servers, end devices might reduce their energy usage. As a result, end-user devices' batteries would last longer [163],[165].

III. Scalability: Cloud computing is still accessible if edge servers or devices lack sufficient resources. The cloud server would assist in carrying out duties in such a scenario [166], [167].

Edge ML in Edge Intelligence

The main issues with ML-based applications are resolved by using (edge computing), and Machine Learning in combination with edge computing suggests a potential solution. This new ML paradigm is referred to as edge [168], [169], or mobile intelligence [170]. Edge machine learning (ML) refers to a group of interconnected systems and a number of devices that are used for data collecting and processing, caching, and analysis that is close to the site of collection in order to enhance the quality and speed up the processing and make sure the confidentiality and privacy of data. Traditional cloud intelligence protects user privacy, speeds up inference times,

and uses less bandwidth. Also, it requires devices to gather and send the data to a remote cloud where edge ML processes and analyzes data locally, [171], [172].

By using their own data to train ML/DL models, users may further customize intelligent apps [173], [174]. Edge intelligence is a very important aspect of the 6G network [175]. In addition to that, edge computing may benefit greatly from AI. This paradigm, which differs from edge intelligence, is intelligent edge [176], [177]. Edge intelligence places a strong emphasis on leveraging edge computing to build intelligent applications in the edge environment whereas intelligent edge concentrates on applying AI solutions to address edge computing issues like resource allocation.

Numerous works that have used an edge intelligence paradigm in real-world application fields have demonstrated the viability of edge intelligence. A facial recognition program is implemented by Yi et al. on a smartphone and edge server [128]. According to the results, the latency is lower when compared to the cloud-based paradigm, dropping from (900 ms) to (169 ms). In order to conduct recognition tasks, Ha et al. deploy a cloudlet, which reduces energy usage by 30%–40% [178]. Some academics focus on how AI performs in the setting of edge computing. A limited DL model for activity recognition has been successfully implemented on smartphones by Lane et al. [179]. The demo outperforms shallow models in terms of performance, proving that common smart devices are suitable for basic DL models. Both embedded devices [180] and wearable devices [181] undergo a similar verification process. The most well-known example of edge intelligence is Google G-board, which employs federated learning [182] to jointly train the typing prediction model on mobile devices. To teach G-board, each user utilizes their own typing logs. As a result, the trained G-board could be put to use right away, powering experiences that were customized by the way people interacted with this application.

Traditional and Edge Intelligence

In traditional centralized intelligence, all edge devices submit data to the central server first before doing intelligent operations, such as model training or inference. The main server or data center is typically, but not always, situated in a distant cloud. Findings, such as recognition or prediction results, are communicated back to edge devices following processing on the central server. Whereas in edge intelligence, tasks like recognition and prediction are either carried out

by edge servers and peer devices or by the edge-cloud cooperation paradigm. The data is either sent to the cloud in very modest amounts or not at all. For instance, cloudlets, or BS and IoT gateways, might operate fully intelligent models and algorithms to offer services to edge devices in some areas. In another area, through various edge devices which are carried out, a model is broken up into several components with distinct functions. Together, these edge devices complete the process.

Inference Latency Of Deep Learning Models

Deep learning has flourished in the last years, due to its very high performance. Development in deep learning is so rapid that it can solve a larger number of problems and be adopted in more systems. Deep learning is an advanced level of machine learning that utilizes a multi-layered hierarchical level of artificial neural networks to carry out the process of machine learning and deliver high accuracy in tasks such as speech recognition, object detection, and language translation.

Deep learning models need to be equipped with adequate processing power to solve complex real-world problems. To ensure better efficiency and less time consumption, data scientists switch to multi-core high-performing GPUs and similar processing units. These processing units are costly and consume considerable power. We have a large set of hardware to run deep learning models on and each is suitable for a certain system. Thus, we are confronted with quite a few design choices. It is important to understand the performance of different hardware for running deep learning models, in terms of their inference times, memory usage and power consumption, as these are vital for architecting systems to meet certain design requirements.

In deep learning, inference time is the amount of time it takes for a machine learning model to process live data points and make a prediction/classification. Inference time is an essential component that must be performed to achieve the required level of accuracy. Predicting inference time is crucial in tuning DNN models (e.g., hyperparameter settings) to achieve required accuracy and performance. Once the model is trained, it can be used to make predictions on new data very quickly. This makes them well suited for use in real-time applications such as text recognition, speech recognition, and facial recognition.

There have been several previous works in predicting inference time. One approach considers FLOPs (floating-point operations) [5] [11] to be the atomic operations of the deep learning model.

Such operator-level methods sum up the latencies of all operators. The problem with this approach is that it does not consider any type of runtime optimizations or data loading time. Another approach is BRP-NAS [1], which considers the deep learning model graph representation to predict inference latency. However, this approach does not work well for many unseen model structures.

In this thesis, we propose training deep neural networks based on the features that come from computational resources such as RAM, CPU, GPU, etc. Here we propose a framework to predict the inference time of deep learning models on a CPU device, by summing the inference time prediction for each layer of a given DNN model. Training our proposed model is done by generating a dataset of 60000 data points which will be discussed in this paper. Testing the model is done using 13 Benchmarked deep learning architectures from [2]. The technical contributions of this paper include providing a prediction framework that can deal with non-linearities while also generalizing to previously unseen neural network models or hardware. In addition, predicting the inference time by training our proposed model on many features that can potentially influence the performance of deep learning inference times and providing flexibility while reducing the computational time needed to train our approach and maximizing the types of layers.

Real-Life Applications Where Latency Prediction Served as a Key Role

The few real-world applications listed below where latency estimate, and prediction played a key role are as follows:

1. In real-time Internet applications, network latency prediction is crucial for server selection and quality-of-service estimate [86].

2. Optimal dataflow plans and/or assignments of operators to nodes are searched for a given set of continuous queries. Re-optimization, which is the periodic modification of the continuous queries on the basis of identified changes in input behavior, is an issue that is closely connected to this one [88].

3. We must promptly and precisely forecast the related impact on the system when attempting to add or delete a continuous query from the system [80].

4. The system administrator must assess the impact of adding or removing CPU cycles or nodes from the system's available resources under the system's present continuous query load.

5. Users require an accurate assessment of how their ongoing inquiries will behave. Such an estimate ought to be based on a user-relevant statistic. The system's performance guarantees might potentially be based on this [49].

6. One of the most significant applications is (vehicular-hoc) networks (VANETs), which have been installed in the USA, Europe, and Asia, and have been envisioned to be promising in road safety and many other commercial applications. The following are some characteristics of vehicular ad hoc networks:

• Because of the great flexibility of the vehicles, the topology of (VANETs) changes quickly. Due to inconsistent connection, VANETs often are unable to create a reliable end-to-end channel to send data packets.

• The flexibility of vehicle nodes in this system is restricted by established highways. The data transfer based on the V2V connections across VANETs is also limited by the roads due to some obstructions. The most important issue is to select a forwarding path with the smallest packet delivery delay [50].

7. Fog computing (FC) is a developing field of computer science that works in a dispersed setting. FC wants to offer cloud computing capabilities to devices on the edge. The method is anticipated to meet the minimal latency requirement for Internet-of-Things (IoT) devices in the healthcare industry [41].

If we want to summarize the main advantages in a few points, we will say they are firstly, our method provides a predictive framework that can manage nonlinearities while generalizing to previously unseen neural network models or hardware, and secondly, through train our proposed model to predict inference time on many parameters that may affect DL performance. Third, our method provides flexibility while reducing the computational time and type of maximization layers required to train our method. The benefit of our approach is to bridge the gap by providing a predictive framework that can manage these nonlinearities while generalizing to previously unseen neural network models or hardware. We would also like to tune the computational

resource limits of our hardware to ensure good accuracy and avoid errors during model execution. In this approach, we train DNN to predict the inference time of the DL network parts. These individual parts of time can then be provided predictions for the entire model.

The approach reduces computation time while maximizing the types of layers we can predict using atomic operations. To train our method, we generate convolutional layers with random parameters on our hardware device for convenience. However, our method can be generalized to any hardware by adding hardware parameters to the training data, e.g., (clock frequency, FLOPS frequency). It can be done by just forward propagation or the full cycle. Therefore, the method we propose can make decisions when choosing the appropriate hardware to train or derive the model and help with model design decisions. We have a highly flexible model that generalizes to various network models, data sizes, and hardware configurations. Our method produces incredibly good results by using only convolutional layers, as this is the main contributor to inference time.

Our approach here is to divide any deep learning model into layers and treat layers as atomic operations in a DNN model. Then we train an FFNN (feedforward neural network) on a large number of random parameters layers. Using the recorded time as the target, we test the reliability of our model versus the benchmark model to validate our method. Prediction is done by summing all predicted inference times for selected layers in the DNN architecture. This approach allows us to save a lot of resources and time and eliminates the need to evaluate models on hardware to determine if they are suitable for a particular application.

Compare the results of predicting the inference time of the feedforward path using our method with the results of the linear regressor (FLOPs approach). The linear regressor clearly fails to capture the complexity of the model and produces a large amount of inaccuracy (RMSE 9.3 ms). In contrast, the deep learning predictor (our approach) provided consistent predictions compared to the actual measurement time (RMSE 2.98 ms). We can see these two results in Figure 4 and Figure 5. This suggests that deep learning-based predictors are more suitable to use in this case. This may be because linear regressors cannot account for the nonlinear relationship between parameters and inference time. Also, different operations may take various times to perform.

Furthermore, this is both implementation and hardware specific. Several devices, such as GPUs, have a very non-linear dependence of inference time on operands. The data movement

operation is not considered at all, since it is not a formal computational operation, but it takes some time and can be significant at times. On the other hand, our approach shows that accurate inference time predictors play a key role in NAS (Neural Architecture Search) where latency on target hardware is important and existing latency predictors are too error-prone.

In addition, BRN-NAS is a new prediction-based NAS framework that gathers binary relational accuracy predictor models and a data selection method to enhance the performance of NAS [22]. The state-of-the-art BRP-NAS uses a GCN to predict the latency (inference time) of the NASBench201 dataset on varied hardware. It captures runtime by using a representation of the model graph and predicting latencies. BRP-NAS [22] outperforms previous FLOPs [21] methods in productivity and accuracy. On the other hand, the GCN is dependent on the tested model structure and is not working for many unseen model cases. Over and above that, comparing our method (Layer Level prediction) with nn-Meter [20], a kernel-based prediction system predicts the inference of (DNNs) models on different hardware very well. nn-Meter proposes kernel detection, which records various operator fusion behaviors [20].

By collecting the most useful data, nn-Meter builds a kernel-level inference predictor. nn-Meter provides high accuracy on unseen models which outperform previous BRN-NAS methods [22]. nn-Meter addresses this problem by developing a more fine-grained approach to characterizing kernel-level models. However, this approach involves extensive testing of all kernels, which can finish in 1 to 4.4 days, depending on execution time. We propose a Layer Level prediction method. Using our results, we have shown that by describing operator-level hardware devices and runtimes, our model can accurately estimate neural optimization-based, previously unseen runtimes in just in few minutes not many days, and the end-to-end latency of the network architecture is executed. This means our method outperforms FLOPs [21], BRN-NAS [22], and nn-Meter [20]. In our work, we propose a simple but efficient latency method that yields the most accurate latency estimates for CPU devices, including optimization graphs. We also observed additional gains when we produced a targeted uniform sampling and normalization strategy. Our method shows particularly strong gains in estimating latency when trained on only 12 model architectures from a single device, yielding 95% accuracy.

The execution time required for the forward pass through the neural network is limited by the floating-point operations (FLOPs). These FLOPs rely on the DNNs architecture and the size of the data. The time consumed for each FLOP relies on the hardware features. Also, when data

transfers are managed and memory bandwidth is used effectively, the time of communication matches with data size.

Recent efforts in the literature have focused on this type of linear model. However, other features, such as the activation function and the optimizer used, introduce sources of nonlinearity into the system. Furthermore, computing resources and memory bandwidth are not fully utilized in all cases. Consequently, more sophisticated methods produce unsatisfactory results and generalize poorly across multiple types of hardware. Our approach helps to close this gap by proposing a predictive framework that can handle these nonlinearities, delivering accurate results while generalizing to previously unknown neural network models or hardware. We will describe here the hardware we used to evaluate our method. We train the model on CPU and just use it to compare different methods. We expect that a single model spanning more platforms will not only be equally accurate but will also make predictions on previously unknown hardware.

However, due to the limited hardware we have available, we believe that combined models may not currently achieve good accuracy compared to individual models.

Limitations of Latency Prediction in Deep Neural Networks (DNNs)

Deep neural networks (DNNs) have made great strides in the previous ten years, which has prompted their use in a number of study areas where complex systems need to be represented or understood, such as earth observation, medical picture analysis, or robotics. DNNs have gained popularity in high-risk industries like medical image analysis [76][78], or autonomous vehicle control [79][81]; however, their use in mission- and safety-critical real-world applications is still in its infancy. The primary causes of this restriction include:

- It is difficult to trust the outcomes of a deep neural network's inference model since it lacks expressiveness and transparency [77].
- The sensitivity to changes in a particular domain [82], the difficulty to distinguish between samples that are inside and outside of a certain domain [83][84].
- Considering the prevalence of overly optimistic predictions and the inability to provide reliable uncertainty estimates for a deep neural network's decision [85][86].
- Due to their susceptibility to adversarial assaults, deep neural networks are exposed and insecure [87][88].

These problems primarily stem from a lack of understanding of the neural network or uncertainty that is already present in the data (data uncertainty) (model uncertainty). It is crucial

to offer uncertainty estimates so that doubtful forecasts can be discounted or forwarded to human specialists in order to get around these restrictions [89]. Giving uncertainty estimates is vital for safe decision-making in high-risk domains as well as in those where labeled data is scarce and data sources are extremely homogeneous, like in remote sensing [90]. Uncertainty estimates are critical not just for sectors where uncertainties play a major role in learning procedures, but also for active learning [91] and reinforcement learning [86][80].

Researchers' interest in assessing DNN uncertainty has grown recently [66][71]. By separating modeling, the uncertainty caused by the model (also known as epistemic or model uncertainty) and the uncertainty caused by the data, the most popular method for estimating the uncertainty on a prediction (the predictive uncertainty) is possible (aleatoric or data uncertainty). The second one cannot be reduced; however, the first one can be by enhancing the model that the DNN learns. Bayesian inference [86][34] ensemble approaches [57][78] test time data augmentation approaches [45][36] or single deterministic networks with explicit components to represent the model and the data uncertainty [52][83] are the most crucial methods for modeling this separation.

For safe decision-making, estimating the forecast uncertainty is insufficient. Furthermore, it is necessary to guarantee the accuracy of the uncertainty estimates. In order to do this, the level of dependability of DNNs has been examined, and re-calibration techniques have been suggested [57][62].

Following are the main challenges associated with latency prediction in DNNs:

Low-Quality Training Data

With real-world tabular data sets, data quality is frequently a problem. In contrast to the high dimensional feature vectors created from the data, they frequently contain missing data [28], extreme data (outliers), incorrect or inconsistent data [29], and have a limited overall size [30]. Tabular data are usually out of class due to the pricey nature of data collecting. All machine learning algorithms face these difficulties; nevertheless, the majority of contemporary decision tree-based algorithms can deal with missing data or different variable scopes internally by seeking out suitable approximations and split values [31][32].

Missing Irregular Spatial Dependencies

In tabular data sets, there is frequently no spatial connection between the variables [86], or there are quite complex irregular dependencies between features. Working with tabular data

requires learning from the start the structure and connections between its characteristics. Convolutional neural networks, a popular model for homogenous data, use inductive biases that are inappropriate for modeling this type of data [33][35].

Dependency on Preprocessing

One major benefit of homogeneous data in deep learning is that it incorporates an implicit step for learning representations [36], necessitating little to no explicit feature building or preprocessing. However, the preprocessing approach chosen may have a significant impact on performance for tabular data and deep neural networks [37]. Managing the category features is still a difficult task [38] and can quickly result in a very sparse feature matrix or provide a synthetic ordering of previously ordered items such as using an ordinal encoding scheme. Last but not least, deep neural network preprocessing techniques may result in information loss and a decline in predicting accuracy [39].

Importance of Single Features

Single features are important because, unlike changing the class of an image, which often needs coordinated changes to multiple characteristics, such as pixels, a single categorical or binary feature can completely affect the outcome of a prediction on tabular data [40]. Decision-tree algorithms, as opposed to deep neural networks, can handle shifting feature relevance remarkably well by choosing a single feature and suitable threshold (ex, splitting) values and ignoring the remainder of the data sample. Individual weight regularization may lessen this difficulty, according to [40][42].

Existing Approches And Limitations

DNNs have demonstrated remarkable performance in a variety of activities that initially appeared challenging [123, 117]. Even though many IoT devices need highly accurate sensing capabilities, the implementation of DNN models on such systems has moved very slowly [125]. This is mainly because computation-intensive tasks like DNN-based inference execute slowly on IoT devices because of their constrained resources.

There are several ways to minimize the computational requirements of a deep model, including compressing a pre-trained network [122, 109, 114], developing small networks directly [108, 92], or combining the two [113]. With the use of these methods, a user must now choose

when to employ a certain model in order to satisfy the overall accuracy criterion with the least amount of latency. Making such a crucial choice is not easy because the application context (such as the model input) is frequently uncertain and ever-changing. By automatically choosing a suitable model to detect and categorize tomato diseases, our approach lessens the strain on farmers.

Neurosurgeon determines when it is advantageous to offload a DNN layer to be calculated in the cloud (for example, in terms of energy usage and end-to-end latency) [116]. We strive to reduce on-device inference time, unlike Neurosurgeon, without compromising prediction accuracy. For each layer of a CNN, the Pervasive CNN generates numerous computing kernels, which are then dynamically chosen in accordance with user requirements and inputs. By selecting the best model to utilize at runtime, our approach enables having a broad selection of models rather than just allowing the ability to fine-tune a single network structure.

Many software-based solutions have recently been made out to speed up CNNs on IoT devices. By utilizing task parallelism [118, 126], parameter tuning [119], computational kernel optimization [99, 110], and trading precision for time, among other strategies, they seek to reduce the length of the inference process. Since one model is unlikely to satisfy all the accuracy, inference time, and energy consumption requirements across inputs [120], it is desirable to have a method for dynamically choosing the best model to utilize. Our work complements these earlier strategies by precisely offering this capability.

DNN model inference can be accelerated by offloading computing to the cloud, although this is not always possible due to latency or connectivity concerns [102]. The latency problem that arises when offloading DNN inference to the cloud is somewhat addressed in the work reported by Ossia et al [132]. Our method of model selection enables one to choose a model based on the input. Running our models locally on an IoT device is especially advantageous when cloud offloading is unfeasible due to the demand for latency or a lack of connectivity.

Machine learning has been used for a number of optimization tasks [93], including work scheduling [100, 107] and code optimization [121, 124, 129]. Our strategy is quite similar to supervised learning, which uses numerous models to address an optimization issue. This method has been demonstrated to be helpful in optimizing the use of application memory and scheduling simultaneous processes.

On the Nvidia simulator, we propose performing an image classification [105]. We evaluate the performance of the Googelnet, ResNet, and MobileNet convolutional neural network designs [103]. These models were all trained using the Plant Village dataset [94] and are all based on TensorFlow. For inference, the GPU is utilized. Our research demonstrates that the input and assessment criterion affect the best model feature values. It is therefore not simple to choose which model to apply. What we desire is a method that can select the most effective model to apply for any given input automatically.

In our previous work [103]. We come up with the top 3 influential convolutional neural network architectures: Googelnet, ResNet, and MobileNet that got the most accurate predicted inference time using our A Layer Decomposition Approach to Inference Time Prediction of Deep Learning Architectures [103].

We demonstrate that our approach reduces computational time while maximizing the types of layers that we can predict by employing atomic operations. Inference time was proved as a very effective and efficient criterion for identifying the best network architecture to implement and determining the amount of data required for training within the required cost range. We build up our research [103] to show how to select the best model feature values depending on the input and number of evaluation criterion such as inference time, accuracy, memory usage, throughput, and battery requirement. It is a novel runtime strategy for DNN model selection on IoT devices that aims to reduce inference time while still satisfying user needs. Hence, determining which model to use is non-trivial. What we need is a technique that can automatically choose the most efficient model to use for any given input using NVSMI [105].

Deep Neural Networks (DNNs) have been widely used in today's applications [3]. In many applications such as video analytics, face recognition, AR/VR, etc. DNN models are constrained by efficiency constraints (e.g., latency). To design a model with both high accuracy and efficiency, model compression [4] [7] and the recent Neural Architecture Search (NAS) [8] [9] consider the inference latency of DNN models as the hard design constraint.

However, measuring the inference latency for DNN models is laborious and expensive. In practice, it requires developers to perform a deployment process on the physical device to obtain the latency. The engineering effort is tremendous for diverse devices (e.g., mobile CPU/GPU and

various AI accelerators) and different inference frameworks (e.g., TFLite and Pytorch). A high cost can hinder the scalability and make the measurement-based method practically infeasible to support the fast-growing number of devices.

There are many approaches to predict the inference latency of deep learning models. First, predicting inference latency considering FLOPs (floating-point operation) [6] [12] to be the atomic operation of the deep learning model. Such operator-level methods sum up the latencies of all operators. The problem with this approach is that it doesn't consider any type of runtime optimizations or data loading time.

Second, the state-of-the-art BRP-NAS [1] uses graph convolutional networks (GCN) to predict the latency of the NASBench201 [14] dataset on various devices. Individual network levels serve as the vertices of a graph, and the edges represent the data flow between those layers.

A feature vector, or layer representation, identifies each vertex. How to create effective layer representations that contain sufficient data to forecast the latency of real-world DNN architectures is a significant challenge. For instance, the network layer type one-hot encodings are used by BRP-NAS. The unique collection of DNN architectures supplied in the NAS-Bench-201 dataset complements this extremely straightforward technique effectively [48]. However, in general, the parameters of the layers, such as the shape of the input and output tensors, as well as the layer types, affect the latency of individual network layers and, consequently, of entire DNNs. As a result, more realistic DNN designs do not generalize well to this layer representation.

BRP-NAS is a new prediction-based NAS framework that gathers a binary relational accuracy prediction architecture and a purified data selection strategy to improve the performance of NAS. BRP-NAS outperforms previous NAS methods in both sample efficiency and accuracy on NAS-Bench-101 and NAS-Bench-201, and also beats DARTS in its search space. However, this model graph-based approach is highly dependent on the tested model structure and may not work for many unseen model structures.

There are two phases for BRP-NAS. The first phase is to predict the ranking of all candidate models (accuracy comparisons between two models) includes the outputs from a binary relation predictor, which is trained to forecast the binary relation. In the second phase, the models with high predicted ranks are fully trained based on the predicted rankings, and the model with the best-trained accuracy is then chosen.

Third, the NN-meter [15] approach uses kernel-level operations to predict inference latency. nn-Meter is a successful and precise inference time-predicted method of DNN models on edge devices. By dividing the model-wide inference into kernel or device execution units, the basic idea behind nn-Meter is to make predictions at the kernel level. The two main methods that nn-Meter develops are kernel detection, which employs a carefully crafted set of test cases to automatically identify a model's execution units, and the optimal configuration, where adaptive sampling effectively selects the space for the development of precise kernel-level latency predictors. Three popular types of edge hardware are employed for its implementation ex, mobile CPU/mobile GPU. When put to the test on a huge dataset of 25,000 models, it remarkably beats the prior state-of-the-art [54].

Depending on the hardware and runtime, a kernel can be a single operator or a combination of many operators. The latency sum of all the kernels in the model is used by nn- Meter to forecast the overall latency of a model. This kernel-level prediction design decision is based on two findings. First, in deep learning frameworks, particularly on edge devices, the kernel is the functional unit (for example, GPU kernels). As a result, the term "kernel" naturally encompasses a variety of runtime improvements, including operator fusion, the most significant optimization that has the most potential to reduce latency. The types of operators and kernels are stable with a very limited collection, despite the fact that there are a lot of DNN models available. Any model is basically a collection of various operator and kernel combinations. As a result, kernel-level prediction can enable previously unseen new models.

To nn-meters, there are two basic challenges. The first challenge is figuring out how to divide the model effectively into a number of kernels for each edge device. The operating kernel will differ from device to device due to different runtime improvements. In contrast to mobile CPUs and Intel VPUs, conv-add is not a concatenation operator on mobile GPUs. Numerous inference frameworks are also closed-source. The kernel ought to be created for open-source kernels that demand hardware expertise. Second, creating precise kernel predictors is challenging. The issue may be resolved using two methodologies, automated kernel discovery and adaptive transmission sampling [54].

Fourth, Paleo based DNN model. It is currently unclear how to employ parallel and distributed computing resources to speed up their deployment and training, although various adjustable deep learning packages have been suggested. Furthermore, the efficiency of contemporary parallel and

distributed systems varies significantly depending on the neural network design and dataset in issue. To efficiently explore the universe of adjustable deep learning systems and measure the efficiency for a particular issue instance, the PALEO analytical performance model is presented.

The computational requirements for building and analyzing a neural network are declaratively stated in the architecture of the network. From a specific architecture's requirements, we can match them to a location within the design space of hardware, software, and communication approaches, PALEO may efficiently and precisely forecast the predicted performance and scalability of a hypothetical deep learning system. PALEO is unaffected by the choices of hardware, network architecture, software, and communication protocols, or parallelization strategies. It can accurately replicate some recently reported DNN scalability findings [42].

The convolutional layer is one of the most often utilized and computationally costly types of layers in contemporary DNNs. There have been numerous highly efficient implementations as a result [43][44]. It is simple to derive reasonable FLOPs counts for other kinds of layers. The selection of the algorithm matrix multiplication depends on the filter size, strides, input size of the convolutional layers, and memory workspace and is thus problem-specific. It is crucial for PALEO to make decisions similar to real-world systems in order to produce credible estimations for user-specific DNNs equivalent to actual executions. PALEO is resilient to changes in network design, hardware, software, communication protocols, and parallelization techniques resulting in reduction of inference time.

To choose between these algorithms, existing DNN software frameworks and libraries use two typical methods:

✓ Using pre-established heuristics based on offline benchmarks

✓ Using auto-tuning to experimentally assess available algorithms on the provided specification.

Since auto-tuning depends on platform and software implementations, PALEO by default employs the first strategy for maximum universality [42].

Fifth, NeuralPower based DNN model. It is a sparse polynomial regression-based layer-by-layer prediction approach for estimating the serving energy consumption of a DNN deployed on any GPU platform. NeuralPower provides an accurate prediction for runtime and energy across all layers in the entire network given the design of a DNN, assisting machine developers in diagnosing the

power, runtime, or energy bottlenecks. In order to help machine learners, choose an energy-efficient DNN architecture that better balances energy usage and prediction accuracy, the energy precision ratio (EPR) metric is assessed. NeuralPower exceeds the most accurate prediction model that has been released so far, improving prediction accuracy by up to 68.5%.

The accuracy of predictions is assessed at the network level by forecasting the runtime, power, and energy of state-of-the-art CNN architectures, with average accuracy ratings of 88.24% in runtime, 88.34% in power, and 97.21% in energy. Neural Power's effectiveness is a measurement that affirms it as a solid machine learning framework by putting it to the test on various GPU systems and Deep Learning software programs [45]. Better estimation is obtained by NeuralPower model as it involves simpler computations based on the idea of regression.

Some other related work is Blackthorn based DNN model. As deep neural network (DNN) accelerators and more robust and efficient embedded devices become approachable, machine learning is turning into a crucial aspect of edge computing. Finding the ideal platform for a particular application gets harder as the number of these devices increases. Application developers frequently struggle to find the best affordable DNN and device combo that satisfies accuracy and latency criteria. Blackthorn (Layer-wise based DNN model) is a layered latency estimation method for embedded Nvidia GPUs that is based on analytical models. Accurate forecasts for each layer, assistance for developers in locating bottlenecks, and DNN design optimization for target platforms are some of its benefits [46].

Many issues with data collection are resolved by selecting the benchmarking websites that provide the most information to decrease the essential measures. Analytical layer designs based on characteristic templates are used to increase the generalizability of the estimator while keeping the sparseness of the underlying dataset. This is also the only technique for producing layer-wise models by mapping a limited number of characteristics and linear functions to the observed latency based on the authors' knowledge. With Blackthorn, networks can be evaluated fast, compression algorithms can be guided to make better use of a platform (including platform-aware pruning), and network improvements may be foreseen, and their effects predicted network architecture seek (NAS).

Utilizing vendor-specific inference frameworks, the profiling tool creates estimation models based on measurement results collected from a platform. The estimation tool uses the derived

estimate models to forecast the inference time of a network. Blackthorn can save developers time and money by assisting them in swiftly determining the optimum platform and optimizations for their use-case [46].

Layer-wise predictors have been proposed that can efficiently aggregate the delays of individual neural network layers. Blackthorn, while more accurate than proxy, has a distinct disadvantage: it cannot capture the complexity of multiple layers running on real hardware.

Another related work is COBRA, which analyzed the source code and then extracts the corresponding network graph and layer implementations. The layer implementations are made up of little pieces of code that invoke deep learning framework functions. Second, it incorporates each layer implementation into a representation that is ideal for latency estimation using a transformer encoder [49]. To learn source code embeddings that are particularly useful for DNN latency prediction, a novel training approach is suggested for the transformer encoder. These extracted layer representations are then combined by a GCN that measures the DNN's latency and captures the data dependencies between function calls.

COBRA is contrasted with BRP-NAS, a predictor built on the GCN that makes use of hand-crafted layer representations that include the kind of layer, all of the layer's characteristics, and the measured layer latency [49]. It's interesting to note that COBRA routinely beats GCN-based predictors that make use of such hand-crafted representations.

COBRA goes through three phases:

• The layer implementations and their call graph are extracted from the code after it has been interpreted.

• A transformer encoder that incorporates layer implementations into feature vectors one at a time.

• A GCN that computes an estimate of the defined DNN's latency using both the call sequence and the layer embeddings

COBRA offers two benefits:

• In contrast to BRP-NAS, it leverages learnt properties rather than custom-made ones to encode distinct function calls. As a result, adding additional layer types is simple.

• It produces equivalent performance to BRP-NAS++ without requiring the measured delay of individual function calls.

In addition, a related work Hardware-adaptive Efficient Latency Predictor (HELP) that models the device-specific latency estimation issue as a meta-learning problem, allowing developers to predict the latency of a model's performance for a particular task on an unknown device with a small number of samples. To do this, innovative hardware embeddings are presented to embed any devices, treating them as black box functions that output latencies, and meta-learn the hardware-adaptive latency predictor in a device-dependent way using the hardware embeddings.

Its validity is demonstrated on unidentified platforms, where it outperforms all relevant baselines and achieves good estimate performance with as low as 10 measurement samples. In latency-constrained environments, end-to-end NAS is assessed frameworks utilizing HELP versus ones without it, and the results demonstrate that it significantly decreases the overall time cost of the fundamental NAS technique [50].

HELP is incredibly flexible and all-encompassing because it can be used with any hardware platforms and architecture search areas. In order to alleviate its computational constraint and produce latency-constrained topologies, HELP may also be connected with any NAS framework. In particular, HELP may complete the full latency-constrained NAS procedure for a new device very rapidly when combined with quick NAS techniques like MetaD2A [51], OFA [26], and HAT [52]. Utilizing the latency dataset for a large device pool in the HW-NAS-Bench dataset [53], the authors evaluated the latency estimate performance of HELP on the NAS-Bench-201 space [48] with numerous devices from different hardware platforms.

In brief, HELP is a new latency predictor that uses meta-learning to generalize a few-shot regression model across hardware devices and can estimate the latency of an unknown device using a small number of observations [50]. With a relatively low overall time cost, HELP achieves noticeably better latency estimate performance than baselines.

On the other hand, HELP has a drawback, while HELP and the NAS methods perform are more accurate than LUT-based estimators, the research presented in nn-Meter [54] shows that existing LPMs (Latency Prediction Models) have difficulty estimating latency when running inference at runtime that optimizes the model graph. HELP investigates this issue in this paper [50]. For example, frameworks such as TensorRT (TRT) perform kernel auto-tuning and layer and tensor

fusion to accelerate inference. In other words, some layers and operations can be merged or run in parallel compared to the sequential execution order set by the model designer.

Last related work is A regression network that is trained on architecture-latency pairings in combination with a hardware-runtime descriptor called MAPLE-Edge, a variant of MAPLE designed specifically for edge devices. MAPLE-Edge, a version of MAPLE created exclusively for edge devices, uses hardware execution descriptors together with architectural and latency combinations to train regression connections. This cutting-edge delay analyzer is made for multipurpose hardware. This state-of-the-art latency predictor is built for general-purpose hardware. With only 10 measurements from an unobserved target device, MAPLE-Edge outperforms the former state-of-the-art methods on optimized edge device runtimes by up to +49.6%. This is in comparison to MAPLE, which uses a bigger set of CPU performance counters to describe the runtime and target device platform [55].

By using a method to normalize performance counters by the operator delay in the observed hardware-runtime, MAPLE-Edge can generalize across runtimes more successfully than MAPLE since it provides better when run on devices having a shared runtime. Collecting more samples from the target device can improve performance for runtimes that don't achieve the requisite precision; an additional 90 samples result in improvements of about 40% [55] [56].

A hardware-aware regression model known as MAPLE-Edge is capable of reliably estimating the inference latency of deep neural network architecture on an embedded system that is not yet visible [55].

When there are several devices, MAPLE Edge produces a 26.08% improvement over HELP and a 7.65% improvement over MAPLE. When trained on about 800 designs from a single device, MAPLE-Edge shows very substantial increases for predicting latency, with an average improvement over HELP and MAPLE of 17.04% and 21.59%, respectively. MAPLE Edge illustrates that it is feasible to accurately estimate latency on optimized edge runtimes by using substantially less adaption samples than other methods, opening the door for more effective architectures for embedded vision to be found via Neural Architecture Search [55] [57].

Hardware gets old and slow by the time of calculating floating points operations. We always need to predict inference time. Inference time is a very important aspect when training complex networks and analyzing big quantities of data. We must consider the computational cost.

It will take time to overcome humans' ability or just achieve an acceptable level of accuracy. Inference time plays a prominent role in training models. It is very effective and efficient for identifying the best network architecture that is possible to implement, determining the best hyper-parameters for the network, determining the amount of data required for training within the required cost range.

Prediction of inference latency DNN models is necessary for different cases in which measuring the latency on real machines is either infeasible or too costly. This is a very challenging problem, and most existing approaches lack high accuracy of prediction. While some research has been carried out to predict the inference time of DNN models – most existing techniques assume that training time is linearly related to the number of FLOPs. We designed and developed a framework to predict the inference time for deep learning models that is generic to be easily extended for a large set of devices. Our key idea is decomposing a given model inference into layers and conducting layer-level prediction. Our experiments demonstrate that this strategy provides significant benefits in terms of prediction accuracy.

In our work, we propose training deep neural networks based on the features that come from computational resources such as RAM, CPU, GPU, etc. we propose a framework to predict the inference time of deep learning models on a CPU device, by summing the inference time prediction for each layer of a given DNN model. Training our proposed model is done by generating a dataset of 60000 data points which will be discussed in this paper. Testing the model is done using 13 Benchmarked deep learning architectures. The contributions of our paper include providing a prediction framework that can deal with non-linearities as well as including previously unseen NN models or hardware, predicting the inference time by training our proposed model on many features that can potentially influence the performance of deep learning inference times, and providing flexibility while reducing the running time needed to train our approach and maximizing the types of layers.

Further benchmarking of our model on a wider variety of hardware will be helpful, along with further investigation of a wider variety of appropriate 'features' used for training it. With this additional work, we will ultimately have a highly flexible model and can generalize across a wide variety of network models, data sizes, and hardware configurations.

Latency Estimation in Mobile GPU

Deep neural network-based algorithms can perform well in a wide range of computer vision applications, including segmentation, object detection, and image recognition [58], [59], among many others [60], [61]. Numerous applications necessitate real-time computer vision issue solving at the end devices, including mobile phones, embedded systems, in-car computers, etc. The hardware, software, and architecture of each of the devices is distinct.

Most often, researchers consider the accuracy FLOPs trade-off while optimizing neural network architecture. The issue, especially for mobile computing devices, is that the actual inference time of the employed neural networks can vary greatly from the theoretical. For instance, in comparison to MobileNet [62], fast and accurate ShuffleNet [63] achieved actual speedup at Qualcomm Snapdragon 820 processor is more than 1.5 less than theoretical. More examples can be found on the TensorFlow [64] Lite (TFLite) benchmark comparison [65], which shows how pervasive the problem is.

Many deep learning applications would like to run on mobile platforms. For most of them, it is important to provide accuracy and inference. Although FLOPs are frequently employed as a proxy for NN latency, they may not always be the ideal option. The researchers employ LUT of all feasible layers for the calculation of the inference on a mobile CPU in order to get a better estimate of latency. There aren't many experiments necessary. Unfortunately, this approach exhibits low accuracy and is not easily usable on a mobile GPU.

Finding a neural network model that is both quick and precise for each target device and implementation is the key challenge. The task of hand-crafting, an architecture that directly optimizes the accuracy-latency trade-off for each new device is too expensive and time consuming, despite the fact that each new device has some valuable differences in inference time for the same architecture.

Finding a neural network model that is both quick and precise for each machine and implementation is the key challenge. The task of hand-crafting, an architecture that directly optimizes the accuracy-latency trade-off for every new machine is very costly and takes so long time, despite the fact that new machines are different in inference time for the same architecture.

The potential NAS (automatic neural architecture search area) may hold the answer. A specific optimization approach is being used in this domain to search for neural network architecture while maximizing the speed-accuracy trade-off. NAS techniques have already surpassed manually created systems on a number of tasks, including semantic segmentation, object detection, and picture categorization [66]. It results in the creation of the unique datasets, NAS Benchmark 101 [67] and NAS-Benchmark 201 [48], for the purpose of an efficient automatic architectural search. These files include every cell that might be created using different numbers (4–7) of blocks.

On those datasets, a number of effective NAS algorithms, including DARTS [68] and a number of others [69], show efficiency, although they focus on architectural optimization in terms of FLOPs or server runtime. Instead of the actual performance of architectural search techniques, our focus is on the proxy used to quantify architecture complexity and inference time. Once apps prioritize target device speed above the complexity of the number of processes. In their work on MnasNET, the authors provided (NAS) for architectural search and came up with evaluations for each recommended model on a portable CPU [48]. Because of this, they proposed an exceptionally efficient architecture for exploiting mobile devices that did not require the mobile GPU. In order to directly measure real-world inference time, they developed models in TensorFlow Lite and ran the model on mobile devices for NAS. This strategy inevitably calls for doing several actual field tests on the target machinery.

The other approach is used by researchers in Efficient Net-Edge TPU [70]. They created an excellent simulator of the required machine (the mobile CPU), which can operate concurrently on conventional clusters but requires a significant amount of technical work. TFLite is used to implement it as well. The creators of FBNet [27], ChamNet [71], and ProxylessNAS [72] all adopt a lighter and less expensive methodology. Models in these works were implemented using a very effective int8 implementation using Caffe2. The lookup table of all blocks was built by the authors. Next, the total of the individual latencies of the relevant blocks is used to calculate latency. This method does not necessitate extensive testing on the target devices and is extremely effective for modeling mobile CPU delay.

Surprisingly, despite the fact that GPU or dedicated NPU are selected for NN latency (inference), there is relatively few research regarding latency modeling for mobile GPU. The

authors of MOGA [73] investigated GPU-awareness for the various search spaces and discovered that, in situations where latency is estimated for all blocks of the equal structure and input form, the aforementioned lookup table is efficient even on mobile GPUs. In that work, TensorFlow Lite was used to implement the models.

Because they thought a lookup table technique was inadequate for GPU delay prediction, the authors of BRP-NAS [47] offered their own solution Graph Convolution Network (GCN), however no source code or open datasets are provided. On a smartphone without the popular Snapdragon Neural Processing Engine architecture, they ran experiments (SNPE). According to the research, layer-wise neural network latency prediction on a portable GPU is affected by LUT to a very substantial extent.

The field of latency modeling is still in its infancy, and there are no established methods for estimating the speed or time of neural network inference, particularly on mobile GPU. With the suggested technique and LETI pipeline, this research aims to close this gap. The CPU or GPU architecture, the number of cores and processing units, the memory hierarchy, the bandwidth between memory levels, and other factors all have a significant impact on latency. Additionally, it's crucial to tailor the software's implementation to the intended architecture (how to layout data and perform processing of them). It is worthwhile to draw attention to a crucial point: in addition to technology, implementation has a significant impact on how quickly neural networks can predict inferences [74].

There are currently just a few methods for estimating inference time on mobile devices [75]. Every one of them has a strength and a weakness. Generally, they are expressed in Table 1.

Table 1. Inference Time Estimation Techniques on Mobile Devices Advantages/Disadvantages

Sr. No.	Inference Time Estimation Techniques	Advantages	Disadvantages
1	FLOPs	✓ Simple to calculate. ✓ Informs about the quantity of computing processes, which is obviously related to latency.	✓ Time requirements for various operations can vary. ✓ That is additionally hardware- and implementation dependent. ✓ Many devices, including the GPU, have a fairly nonlinear relationship between inference time and number of operations. ✓ Because moving data is not technically a computational operation, it is not considered at all even if it takes time and occasionally is important.
2	Lookup Table	• Only a few numbers of experiments are necessary to reach the whole number of potential blocks.	• It does not consider model loading on a device or data transit across blocks. • It can require a lot of inference.
3	Build Simulator (target device)	➢ Provides accurate latency without the need for actual experiments. ➢ Unlike trials on a mobile device, more durable. ➢ It can be executed in parallel on clusters.	➢ Significant technical work is needed to build a simulator for each piece of target hardware. ➢ Difficult to use on non-CPU machines.
4	Direct measurements on device for each model	○ Allows for accurate inference time estimation using a direct approach. ○ It most closely matches the actual user experience.	○ Inference is very different from different runs, necessitating a number of tests to obtain a reliable estimate.
5	Direct measurement of models (small subset)/construction prediction model	✓ Requires a timetable of numerous field tests for even one test gadget. ✓ Achieves accurate experimental latency results that match user experience.	✓ A model is not accurate as actual measurements. ✓ The nn training is insufficiently representative of the entire search space.

Deep learning models have demonstrated remarkable success in a myriad of applications, ranging from image and speech recognition to natural language processing. However, the proliferation of large-scale deep neural networks has led to significant challenges in deploying these models on resource-constrained devices, such as mobile phones and edge devices. This necessitates the exploration of techniques to compress deep learning models without compromising their predictive performance. The aims and motivations behind the compression of deep learning models are rooted in addressing the inherent limitations associated with model size, computational complexity, and memory requirements [258]–[260]. This section provides a comprehensive overview of the key motivations and objectives driving the research and development of model compression techniques.

In general, there are three common aims when deciding to use DNN model compression:

1. Deployment on Resource-Constrained devices: Large-scale deep learning models often demand substantial computational resources and memory, making them unsuitable for deployment on devices with limited hardware capabilities. Model compression aims to reduce the size of these models, enabling their efficient deployment on resource-constrained devices, such as smartphones and Internet of Things (IoT) devices [258].

2. Enhanced interference speed: Compressed models facilitate faster inference times, a critical factor in real-time applications where quick decision-making is imperative. By reducing the number of parameters and operations, model compression techniques contribute to improved inference speed, making deep learning models more practical for time-sensitive applications [259].

3. Energy-efficient interference: The energy efficiency of deep learning models is paramount, particularly in applications where devices are powered by batteries or operate in environments with limited power resources. Model compression aids in the reduction of computational requirements, leading to energy-efficient inference, thereby extending the operational life of battery-powered devices [260].

On the other hand, the motivation to utilize DNN model compression is commonly dedicated to scalability, network bandwidth and latency reduction, and privacy and security. The scalability of model deployment becomes a pressing concern as deep learning models continue to grow in

size and complexity. Compression techniques motivate the development of scalable models that can be deployed across a diverse range of devices, ensuring that advancements in deep learning are accessible and applicable across various technological landscapes [259]. Transmitting large neural network models over network connections introduces latency and consumes considerable bandwidth. Compression addresses these challenges by reducing the size of models, facilitating faster model transfer and lowering communication costs, which is crucial in distributed systems and edge computing scenarios [258]. Model compression contributes to privacy and security by enabling on-device processing. This reduces the need for transmitting sensitive data to external servers, mitigating privacy concerns associated with data transfer and storage. Additionally, compressed models are less susceptible to adversarial attacks, enhancing the overall security of deployed systems [259].

CHAPTER 3

HARNESSING DEEP LEARNING ON IOT TO CLASSIFY TOMATO PLANT DISEASES [1]

Chapter Overview

Finding a way to effectively run the DNN models locally on IoT devices is crucial. In this chapter, we present an efficient ML-based system to classify tomato plant leaf diseases. Our system is specifically optimized to run on local IoT device coupled with "NVSMI". A unique feature of our system is that it dynamically chooses the most appropriate ML model considering the tradeoff between inference time, accuracy, memory usage, throughput, and battery consumption. We demonstrate the efficacy of our approach through a series of experiments using a real-world dataset.

The timely identification of diseases is essential to the health, quantity, and productivity of the plants. Hence, a locally designed automated system on an IoT device is required in order to make the detection process autonomous with the least amount of human involvement. Using the Plant Village dataset [94], we test the three DNN architectures and calculate accuracy, power consumption, throughput, memory usage, and inference time. Additionally, we examine the relationships between these performance indices to gain information into which DNN architecture would best meet the resource limitations of real-world deployments and applications [103].

This chapter proposes a novel runtime strategy for DNN model selection on IoT devices that aims to reduce inference time while still satisfying user needs. In order to choose the best model to apply at runtime, we use machine learning to automatically create predictors. On the Nvidia system administration interface [105] referred to as (NVSMI).

Method Description

Our method relies on a prediction model to determine which of the three trained image classification models should be used for a specific input given a new, unseen tomato image. Our prediction models are based on a number of input image features, including the brightness, contrast, aspect ratio, and number of edges. The input image is sent to the selected model after it has been selected, and it then tries to classify the image (tomato disease). Finally, the selected model's classification data are returned as outputs. The use of our prediction models will function

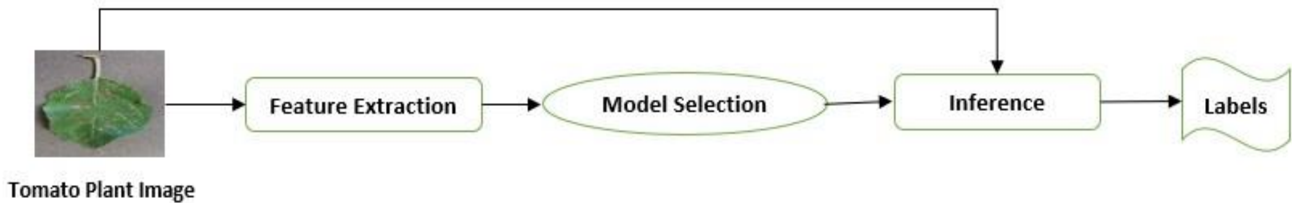


Figure 1. Overview of method

precisely like the use of any other model, with the exception that we have the ability to dynamically select the best model to employ. Figure 1 depicts the overall process of our method.

Figure 2 shows the three k-Nearest Neighbors (KNN) classification models that make up our prediction model. Our model receives an image as input, analyzes it for features, makes a prediction, and then outputs a label indicating which image classification model should be applied.

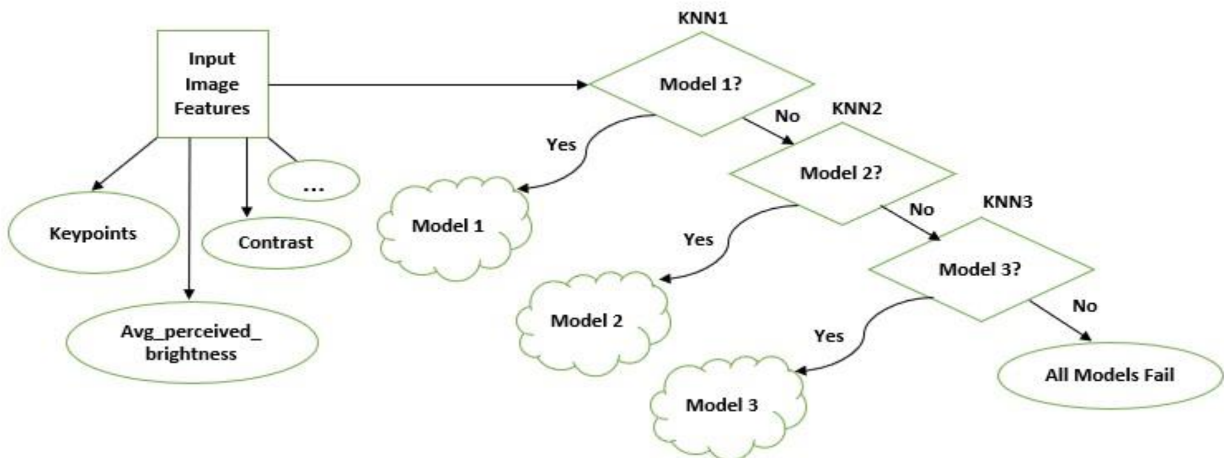


Figure 2. Prediction model architecture includes 3 KNN Models each predicts either use one of the three image classification models

A. Model Description

To build our prediction model, we compare a variety of machine learning methods, such as Decision Trees (DT), Support Vector Machines (SVC), k-Nearest Neighbors (KNNs), and Convolutional Neural networks (CNNs). While developing an inference model selection method for an IoT device, two key objectives to consider are rapid execution time and high accuracy. In our work, we predict the best inference model to use, including MobileNetV2, Googlenet, and Resnet18.

In Figure 3, we evaluated the efficacy of the various models DT, SVC, KNN, and CNN. We selected KNN since it has a rapid prediction time (1ms) and the highest accuracy at solving our challenge. We deployed three different KNN models, each of which makes a prediction about whether or not to use a single image classifier. Lastly, we selected a collection of image features (7 features) to represent each image; the feature selection procedure is covered in more detail in the following section.

Figure 3. Accuracy comparison of multiple machine learning techniques for constructing our prediction model

B. Criteria Selection

We will explain the procedure we used to decide which DNNs to include in our prediction model. The first DNN we include is always the one with the highest accuracy in our training data, or based on other indices such as inference time, throughput, or memory utilization. As a result, a prediction model is created, with Model1 being Googlenet, Model2 being MobileNetV2, and Model3 being Resnet18.

C. Feature Selection

Obtaining the appropriate features to describe the input is one of the crucial components in creating a good predictor. Edge-based features (more edges result in a more complicated image), contrast (lower contrast makes it harder to see image content), and other features that have a major impact on the image were chosen. We choose features based on correlation. We maintain the other feature and discard the other if any pair of features exhibits high pair wise correlation. The relevance of each of our remaining features was then assessed. We first trained and used accuracy to evaluate our prediction model before determining feature relevance. We considered a total of 7 features, shown in Table 2.

Table 2. The selected features

Feature	**Description**
Area_by_perim	Area/perimeter of mail object
Edge_length {1-7}	A7-bin histogram of edge lengths
Avg_brightness	Average brightness
Key_points	Number of key points
Contrast	The level of contrast
Hue	Histogram of the different hues
Aspect_ratio	The aspect ratio of the main object

D. Prediction Model Training

We train our model to predict the same for any new, unseen inputs by first determining which DNN works best for each of our tomato training images. The feature values and associated optimal model for each tomato image in the training dataset are based on an evaluation criterion.

The candidate DNN models must be used on tomato images that have not yet been seen in order to assess their performance. To create training data for our prediction model, we use the approximately 23,000 image Plant Village [94] validation set. The results of each candidate model's exhaustive execution on the images are then measured, together with the inference time, power usage, accuracy, throughput, battery usage, and prediction results. Lastly, we create our entire training dataset by extracting the above feature values from each tomato image and pairing them with the best classifier for that image.

The training data is utilized to choose the best classification models to use together with their appropriate hyper-parameters. We employ a conventional supervised learning technique to train our prediction model because we chose KNN models to build it. The training data utilized in KNN classification is used to label each point in the model, and during prediction, the model uses the Euclidian distance to find the K nearest points (example, K=2). The output label is the one with the most points relative to the prediction point.

Experimental Setup

A. Models and Platforms

We evaluate our approach on (NVIDIA-SMI) NVIDIA System Management Interface program. Nvidia-SMI (also NVSMI) which monitors and manages NVIDIA GPUs such as Tesla, Quadro, GRID, and GeForce. It is installed along with the CUDA toolkit and provides us with meaningful insights. Below Figure 4 is an output of “Nvidia-SMI” command line. The GPU’s is Tesla T4 with memory usage of 15360 MiB and power capacity of 70 watt.

```
!nvidia-smi

Mon Apr 17 13:29:41 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   46C    P8     9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 4. Tabular output from the nvidia-smi command showing details of the NVIDIA Tesla T4 GPU

For System Software, we use TensorFlow v.2.12.0, CUDA (v12.0), Pytorch V. 2.0.0+cu118. Our prediction model is implemented using the Python scikit-learn package. Three trained DNN models for image classification from the TensorFlow-Slim library are considered when discussing deep learning models. The models are created with TensorFlow and trained using images of tomato diseases. The ResNet-18 model has 68 layers, 11.181,.642 total parameters, 11.181.642 trainable parameters, 0.57 MB input size, 62.79 MB Forward/backward pass size, and 42.65 MB parameters size. GoogleNet model has 196 layers, 5.610.154 total parameters, 5.610.154 trainable parameters, 0.57 MB input size, 94.10 MB Forward/backward pass size, and 21.40 parameters size. Mobilenetv2 has 158 layers, 3.504.872 total parameters, 3.504.872 trainable parameters, 0.57 MB input size, 152.87 MB Forward/backward pass size, 13.37 MB parameters size.

B. Models Evaluation

On the Plant Village validation set [94], we analyze our prediction model using 10-fold cross-validation. To be more precise, we divide the 27,500 validation tomato images into 10 sets of equal size, each of which has 2750 images. The remaining nine sets are used as training data, and we keep one set for evaluating our prediction model. Each of the 10 sets is used only once as the

testing data in this process, which is repeated ten times (folds). A machine-learning model's capacity for generalization is assessed using this accepted methodology. We assess our strategy using the following parameters:

- Accuracy is the percentage of tomato images with accurate labels to all the test images and greater accuracy is better.

- Inference time is the time between an input and an output produced by a model, including our prediction model overhead and smaller is better.

- Battery usage is the power consumed by a model to draw conclusions. This also covers our prediction model's energy usage according to our method. The static power consumed by the hardware when the system is not in use is subtracted.

Results

A. Model Performance

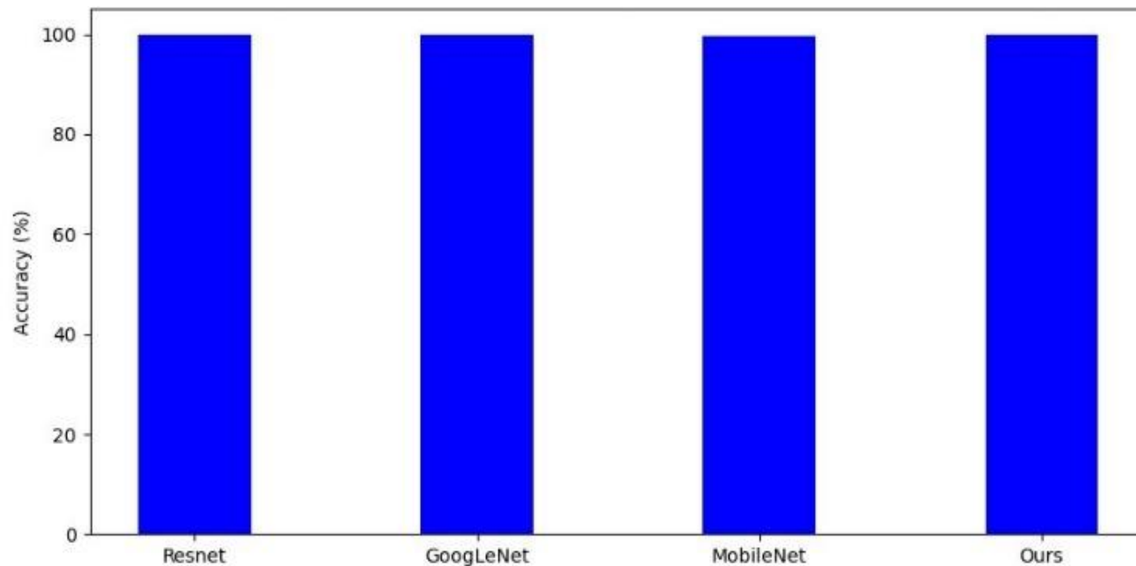


Figure 5. Accuracy and our approach performance against individual models

Figure 5 contrasts the accuracy level attained using each model. We also demonstrate the greatest accuracy attainable for model selection provided by an optimum combination predictor. It should be noted that there are situations in which all DNNs fail, therefore it does not provide 100% accuracy. Yet not all DNNs are equally ineffective; for example, ResNet can classify some

tomato photos that MobileNet cannot. Thus, our approach surpasses all individual inference models by successfully utilizing several models. Despite the fact that we trained with high overall accuracy, this result demonstrates how our strategy can increase each model's inference accuracy.

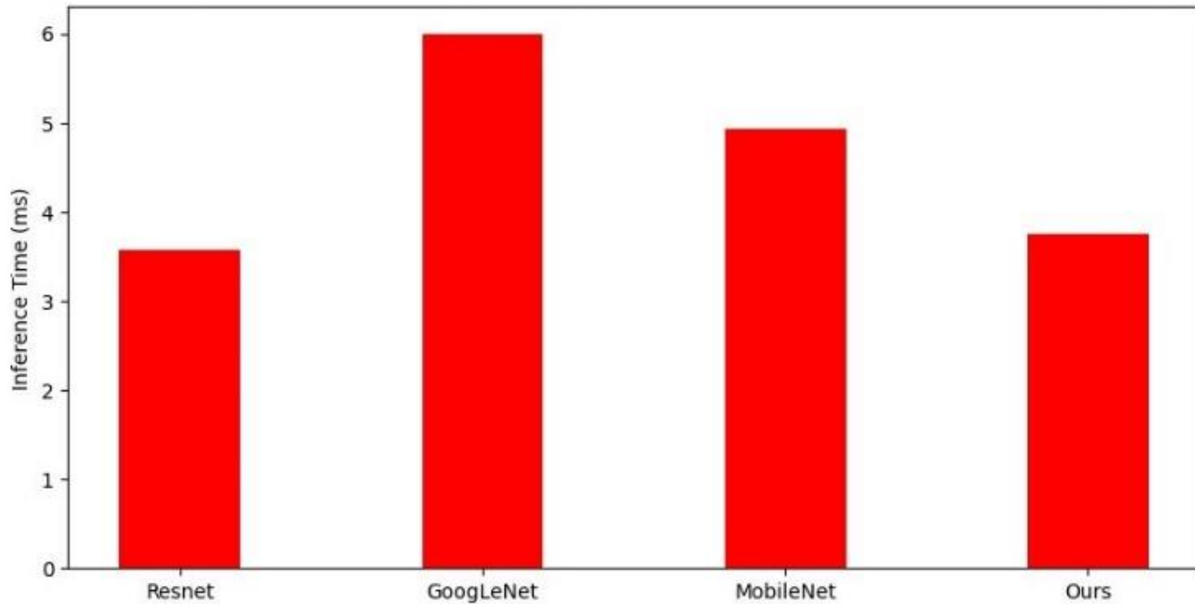


Figure 6. Inference time and our approach performance against individual models

Figure 6 illustrates how our approach and several DNN models compare in terms of inference time. Resnet is the least accurate but the fastest inferencing model, beating both MobileNet and Googlenet, respectively. Our prediction model is faster than Resnet on its own. Feature extraction is where our prediction model involves most of its overhead. Our method's average inference time is less than one second, which is a little bit longer than MobileNet' s average inference time of 0.7 seconds. The most precise inference model in our model set, Resnet, is slower than our method. Given that our strategy can greatly increase Mobilenet's prediction accuracy, we think our prediction model's moderate cost is reasonable.

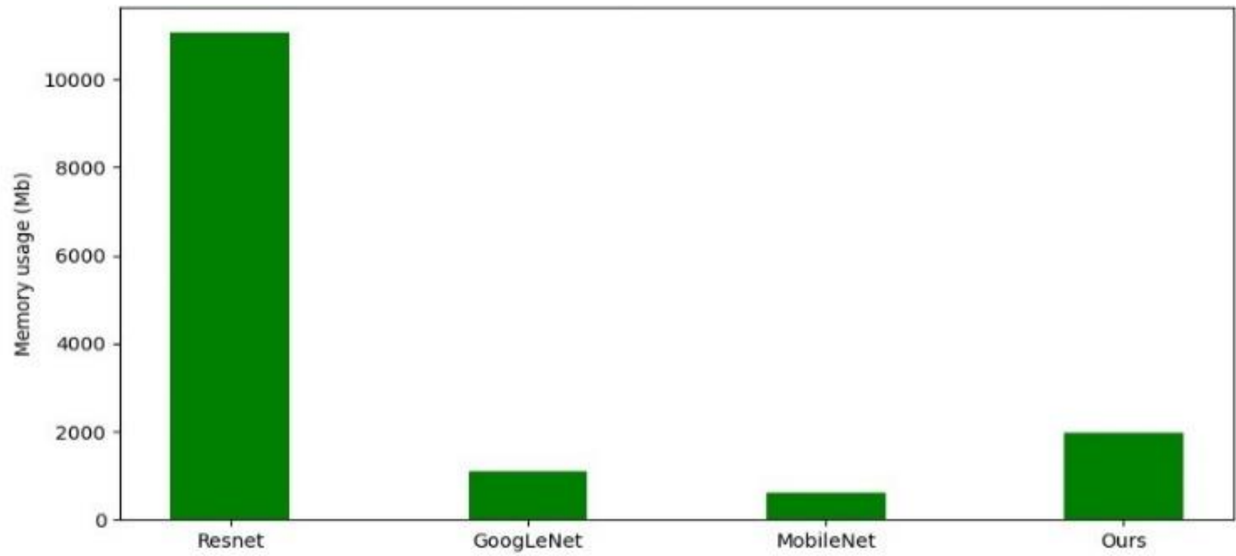


Figure 7. Memory usage and our approach performance against individual models

Figure 7 illustrates how our method and several DNN models use memory. Compared to Googlenet and Resnet, respectively, Mobilenet requires less storage but is less accurate. Our prediction model requires less storage than Resnet on its own. Given that our approach can greatly reduce the amount of RAM used.

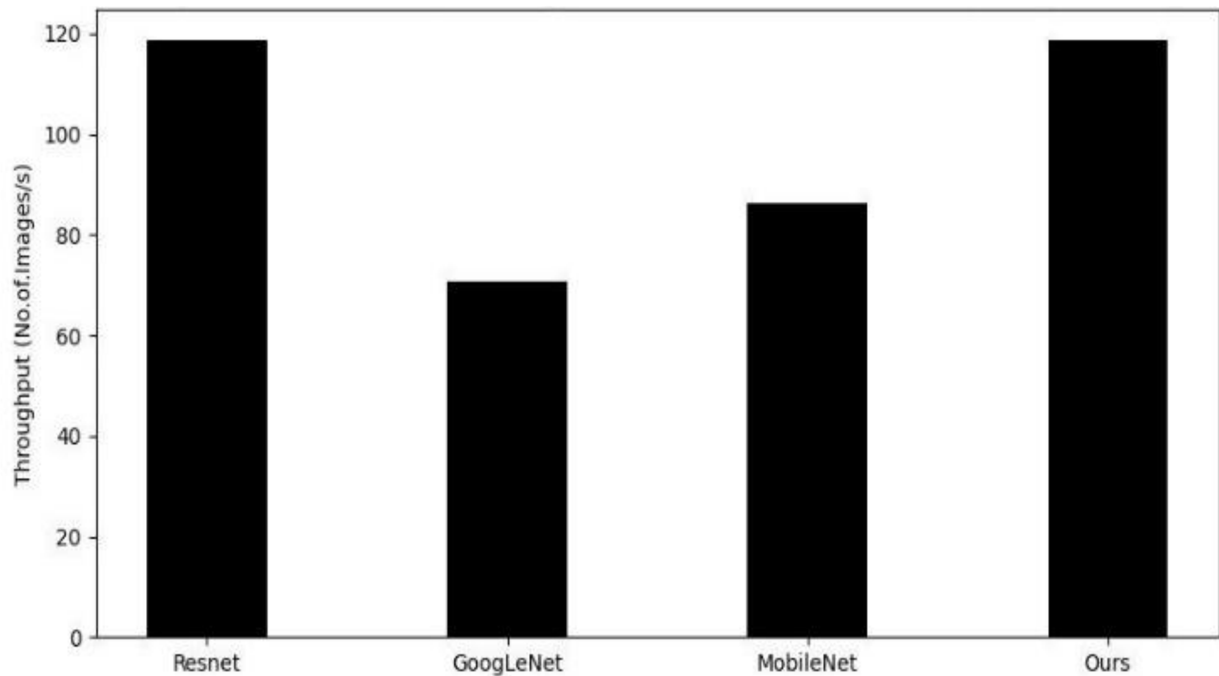


Figure 8. Throughput (No.of. Images/s) and our approach performance against individual models

Figure 8 shows that our approach outperforms individual DNN models in other evaluation metrics. Specifically, our approach gives the highest overall throughput, which in turn leads. In our approach, the selected model will always be the one with the highest throughput.

B. Comparable Models for Prediction Model

The next three Figures 9, 10, and 11 demonstrate how well various methods for building the prediction model worked. Predicting which of the inference models—MobileNet, Googlenet, and ResNet—to utilize in this situation is the learning task. In addition to K-Nearest Neighbor (KNN), we also consider Naive Bayes (NB), Decision Trees (DT), Random Forest (RF), and Support Vector Machines (SVM), as well as Logistic Regression (LR) and AdaBoost (Ada).

To create the prediction model, we employ a neural network structure that is intended for embedded inference. We use the same training examples to train all of the models. The KNN, DT, and SVM feature sets are also utilized. We optimize the training parameters for the NN using a hyper-parameter tuner, and we train the model for more than 200 epochs. In all instances of various sorts of priority, our chosen KNN model has superior accuracy that is comparable to the DT and the SVM, as well as another model. Our general approach to feature selection and model selection is still valid even if the optimum method changes when the application domain and training data size change.

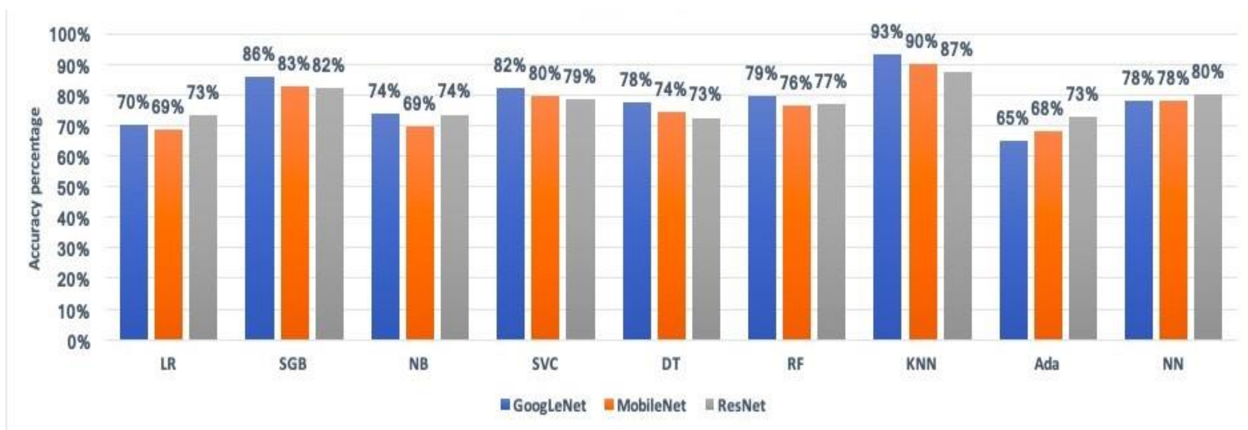


Figure 9. Comparable modeling techniques for Prediction Model (Priority is Accuracy)

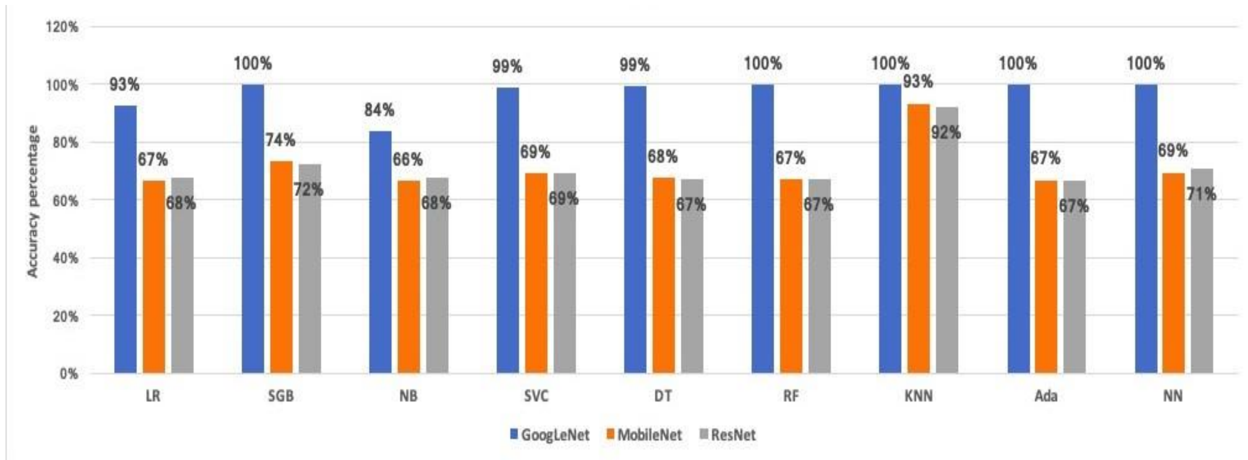


Figure 10. Comparable modeling techniques for Prediction Model (Priority is Inference Time)

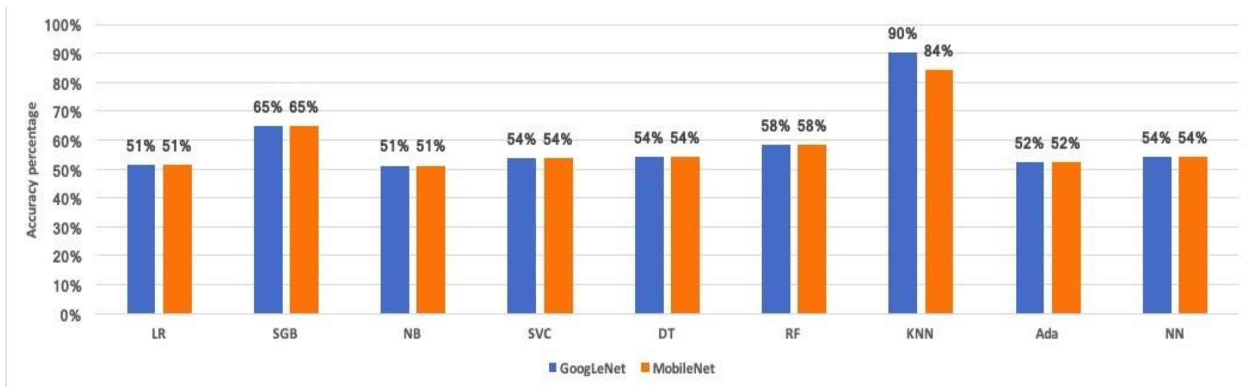


Figure 11. Comparable modeling techniques for Prediction Model (Priority is Memory Usage)

C. Importance of Features

We discuss the process we used to choose the features that would be used to represent each image in our prediction model in the third section, part c. Below, in Figure 12 and Figure 13 we highlight the significance of each of the seven features we looked at in Table 7. It is evident from observation that the seven features we kept are the most crucial; the significance of features dramatically decreases at feature #7 (level of contrast).

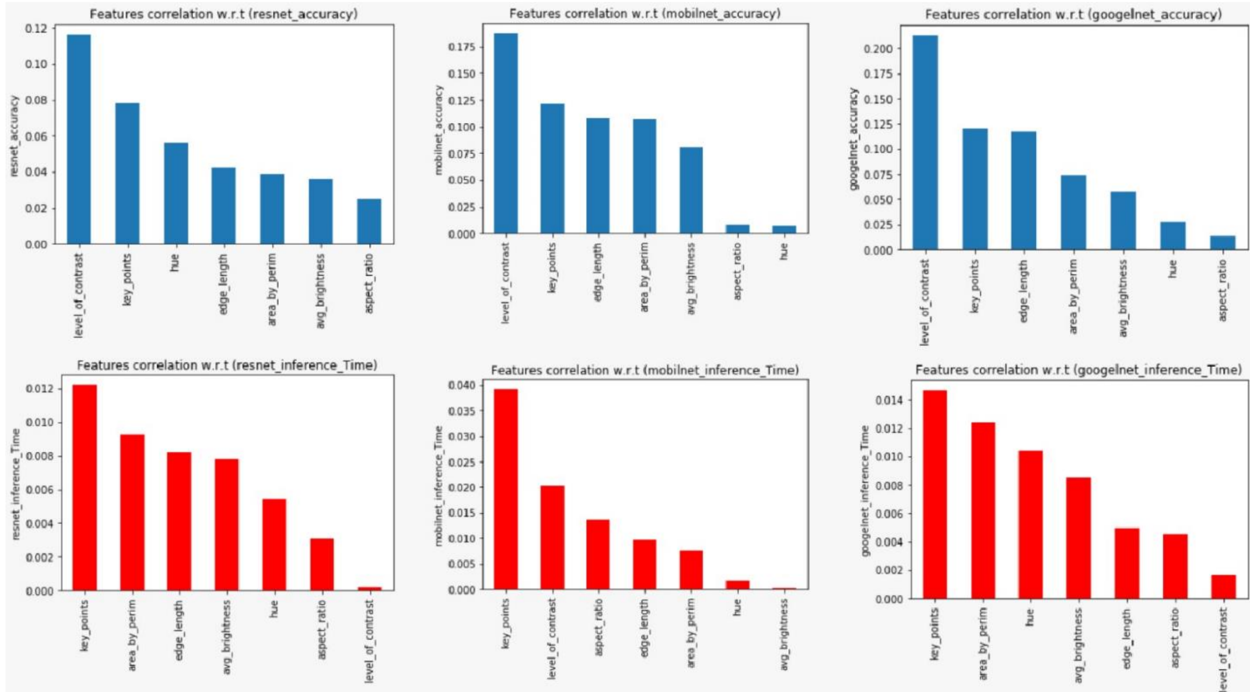


Figure 12. Features Correlation against different indices when priority (accuracy, inference time)

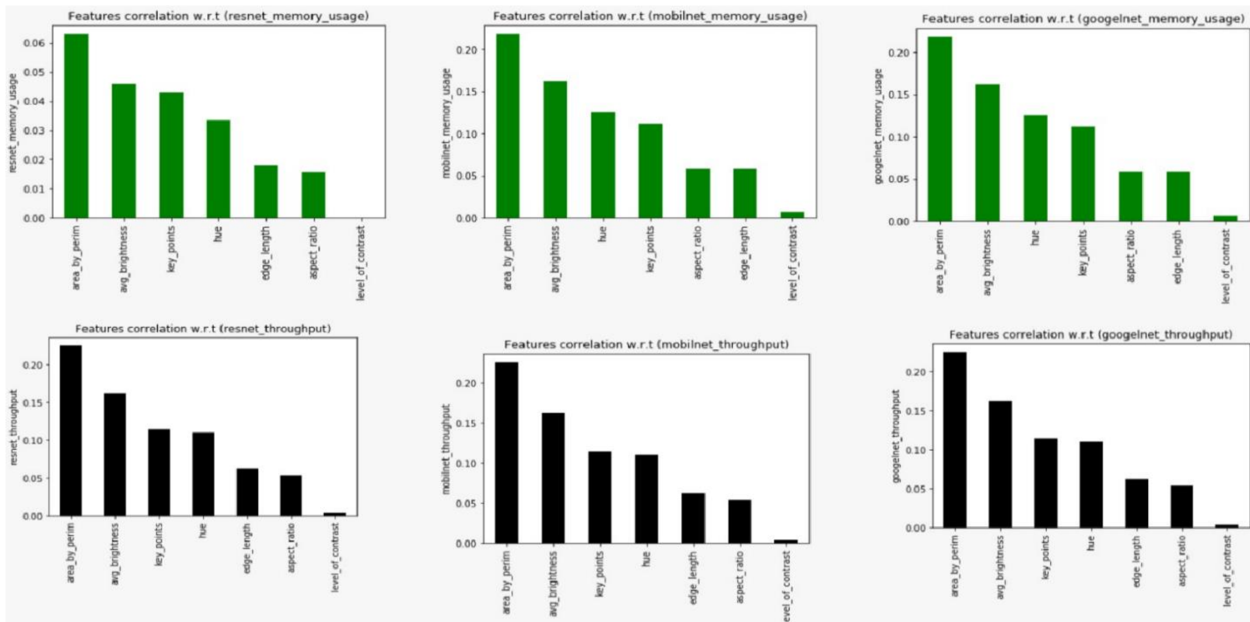


Figure 13. Features Correlation against different indices (Memory usage, Throughput)

D. Impact of Feature Sizes

In the next Figure 14 and Figure 15, we demonstrate how changing the number of features we use has an effect on prediction model accuracy. Accuracy dramatically changes when the number of features is reduced. It is evident that the overhead increases significantly as the feature number increases, but, interestingly, accuracy also slightly decreases. This leads us to the conclusion that using six features is the best size when priority is accuracy or memory usage.

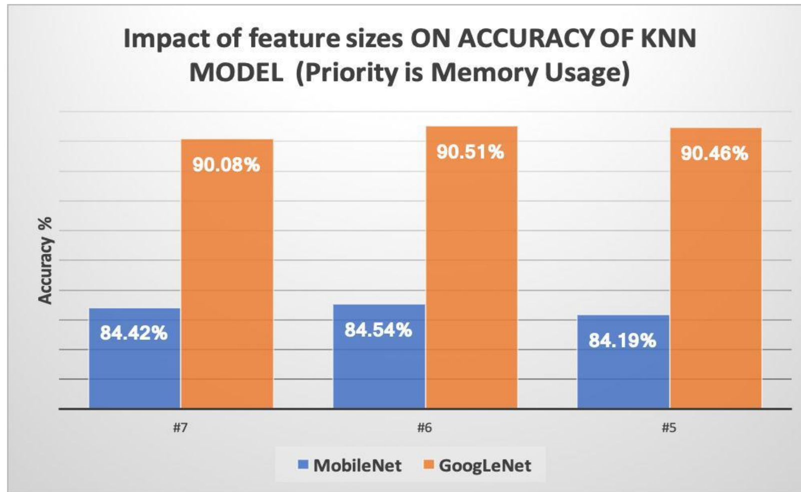


Figure 14. Feature size and accuracy impact when priority is accuracy

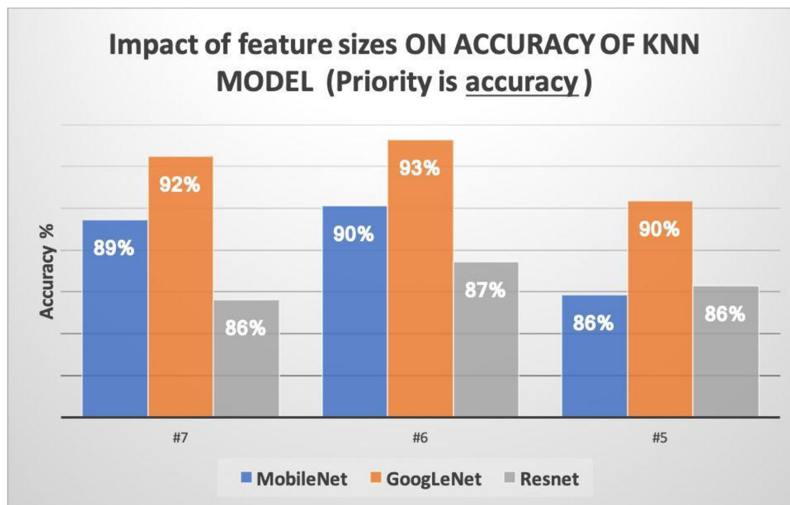


Figure 15. Feature size and accuracy impact when priority is memory usage

The accuracy of the prediction model is demonstrated in the following Figure 16 by showing how it is impacted by the number of features we employ. When inference time being the priority, this leads us to the conclusion that employing seven features is the best size of feature.

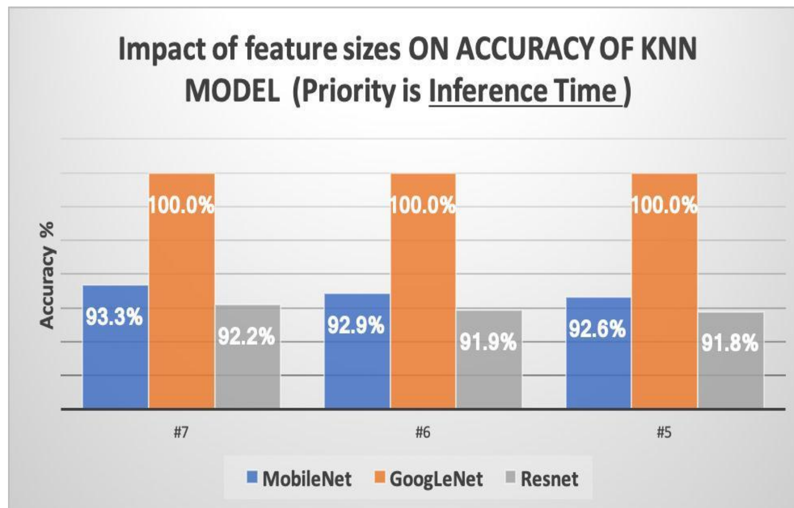


Figure 16. Feature size and accuracy impact when priority is inference time

Chapter Summary

In a few years, we will have billions of connected devices installed in homes, cities, cars, and businesses around the world. User and device environments interact with resource-constrained devices. To interpret the behavior in sensor data, make accurate predictions, and make judgments, many of these devices rely on machine learning models. The number of connected devices that can block the network will become a bottleneck. Therefore, machine learning techniques are needed to integrate intelligence into end devices. Using machine learning on these edge devices can reduce network congestion by being able to perform computations close to the data source. In addition, servers and other powerful computers have traditionally been the only platforms used for machine learning. But with advances in chip technology, we now have portable libraries that can fit in our pockets. Therefore, due to the current advancements in processing power, energy storage, and storage capacity of these devices, the opportunity to gain significant benefits from machine learning on Internet of Things (IoT) devices has emerged. Implementing machine learning inference on edge devices holds great potential but is still in its infancy. With the rapid development of IoT and AI, research efforts to fully realize Edge ML will increase in the coming decades. This study provides a detailed assessment of the past, present, and future of the Edge ML literature. Additionally, caching difficulties and best practices for Edge ML training are clearly discussed. Additionally, Edge ML computational latency for designing realistic and responsive applications is studied. In this

chapter, an innovative method for dynamically choosing the best deep learning model between three CNN models for an IoT device is provided. In terms of accuracy, inference time, and energy usage, our method offers a significant advantage over individual deep learning models. Our strategy relies on designing a Machine learning based system to classify tomato plant leaf diseases running with higher speed and lower power consumption; and depending on the model input and the required level of precision that is running locally on IoT device. A number of input features that are tuned and chosen by our automated approach form the basis of the prediction. Using the Plant Village validation dataset, we applied our method to the tomato image classification task and assessed it on the Nvidia system administration interface referred to as (NVSMI).

CHAPTER 4

INFERENCE TIME PREDICTION OF DEEP LEARNING ARCHITECTURES FOR PLANT DISEASES CLASSIFICATION [2]

Chapter Overview

Prediction of inference latency DNN models is necessary for many tasks where measuring the latency on real devices is either infeasible or too costly. Due to the large diversity among the computing devices that these models may run on, we need to choose between the appropriate device based on cost and performance. Furthermore, finding the suitable optimal device for a given project is a complex process that needs significant time and resources. Prediction of inference latency DNN models is necessary for many tasks where measuring the latency on real devices is either infeasible or too costly. This is a very challenging problem, and most existing approaches fail to achieve high accuracy of prediction. This paper designs and develops a framework to predict the inference time for deep learning models and is generic to be easily extended for a large set of devices. Our key idea is decomposing a given model inference into layers and conducting layer-level prediction. Our experiments demonstrate that this strategy provides significant benefits in terms of prediction accuracy.

---

[2] This chapter partially appears as:
Alqahtani, Ola et al., "A Layer Decomposition Approach to Inference Time Prediction of Deep Learning Architectures ", ICMLA by 21st IEEE

Proposed Solution (System Design)

In this section, we describe the overall architecture of our system and the datasets used. Figure 17 illustrates the system architecture. It shows the components to realize the accurate latency prediction for a DNN model. The idea is to train an FFNN (Feedforward neural network) to predict the inference time of different deep learning architectures.

For training our approach, we generated convolutional layers with random features on one [8] [2] [15] hardware device for ease. However, this approach can be generalized to any hardware by adding hardware features to the training data such as (clock speed, frequency of FLOPS, …). It can be performed either by only forward propagation or full cycle. Training data features and model design are mentioned in the next sections.

For testing our approach, we used 12 DNN model architectures unseen in the training data and pretrained them on the ImageNet dataset [16]. Then, we fine-tuned the models on the plant village dataset [17]. The dataset includes classes with over 87,000 images. We benchmarked these models on our hardware like in [18]. These architectures will be split into layers and each layer will be an input to our prediction DNN Model.

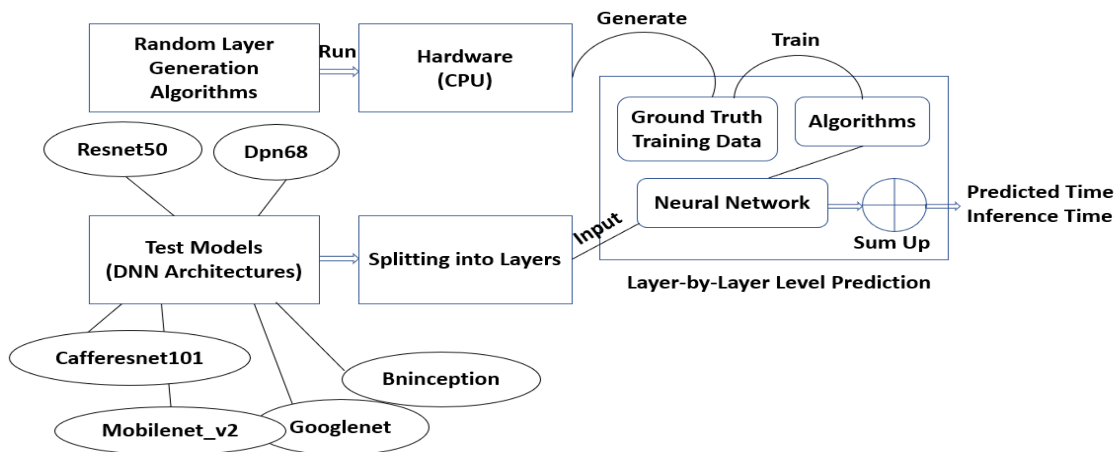


Figure 17. System Architecture

Training Data Features

In this section, we define the features which can influence the inference time of deep learning models. There can be multiple hardware and layer-specific features. Here, we will show some of these features. Notice that lots of secondary features can be defined based on these core features. Our approach has got very good results by only working with convolutional layers as it's the main contributor to the inference time.

A. Layers Features

These relate to features regarding a particular layer within the neural network, and all these features can be encoded in the feature set by one-hot encoding. Some of them are the activation function, optimizer, and batch sizes. Activation functions features can include ReLU, leaky ReLU, softmax, sigmoid, and None. Then, optimizer features that can be used for updating the weight of neural network models, and this can include Adam, SGD, Adadelta, Adagrad, RMS prop, and None. Lastly, batch Size that represents the number of data that gets processed together through the neural network.

B. Layer Specific Features

There are lots of layers in a deep learning model (convolutional layers, pooling layers, fully connected layers, etc.). we will show some of them in the following list:

1) Convolutional layer features: There are many convolutional layer features such as kernel Size, matrix size, strides, padding, input channels, and output channel. Kernel size is the size of filters applied to processed data. Matrix Size is the size of the input image. Strides are the size of strides to be used with kernels. Padding is the size of padding applied to processed data. Input Channels are the number of input channels to a conv layer. Output Channels are the number of output channels from a conv layer.

2) Pooling layer features: Kernel Size is the size of the pooling kernel. Stride Size is the size of the stride with the kernel. Input Padding is the size of added padding to processed data.

3) Fully connected layer features: A few inputs are the number of outputs from the previous layer as all layers are fully connected. Several neurons are the number of outputs of the layer within this layer.

Main parameters for predicting the inference time for DNNs:

In order to predict inference time for a DNN with desired quality accuracy, the correct choice of parameters is critical. Many deep learning applications require low inference latency, which must be within the parameters set by service level objectives. To achieve service level goals, the inference application developer must design or choose neural networks (NNs). But often application developers cannot determine the inference latency of the application until the NN has been deployed on the inference service system. Numerous variables such as network architecture, software environment, and hardware platform can affect inference latency.

Here we discuss inference time prediction for a DNNs. Many methods for predicting DNN latency have been proposed. The most advanced of these is probably the GCN-based predictor (graph convolutional network) currently proposed for BRP-NAS [22]. From the network graph representation of DNN, it is necessary to use GCN to predict its latency. Individual network levels are used as vertices of the graph, and edges represent the flow of data between these levels. A feature vector or layer representation identifies each vertex. How to create an efficient hierarchical representation containing enough data to predict the latency of real-world DNN architectures is a major challenge. For example, BRP-NAS uses network layer type one-hot encoding. This extremely simple technique is effectively complemented by the unique collection of DNN architectures contained in the NAS Bench 201 dataset [21]. In general, however, the parameters of a layer, such as the shape of the input and output tensors, and the layer type affect the latency of a single network layer and thus the entire DNN. As a result, more realistic DNN designs do not generalize well to such layer representations.

Here we define parameters that may affect execution time predictions when performing training. We divide these parameters into layer parameters, layer-specific parameters, implementation parameters, and hardware parameters. Each of these categories can contain an almost infinite list of parameters. As such we outline here a core subset of those parameters but argue that other features could easily be added.

A) Layer parameters refer to the parameters of all layers within the neural network and the hyperparameters associated with those layers. These include but are not limited to:

The activation function is applied to a single neuron. These can include ReLU, softmax, sigmoid, tanh, and none. They can be encoded into parameter sets using one-hot encoding. Second, the optimizer, which is a key element in the learning process of an ML model. Since the main goal of an ML model is to reach the minimum of the loss function, to locate the minimum within the loss function range the optimizer is used. This is where the optimizer comes into play. It defines how to tune the parameters to approach the minimum value. These may include gradient descent, RMS Prop, Adagrad, Adam, Momentum, and Adadelta. These optimizers can be encoded into parameter sets using one-hot encoding. Third, batch size is a very important parameter here since it represents the number of training samples which are processed together as part of the same batch. It should be stated that each individual layer within the network may hold different values for these parameters. As each layer is predicted independently this is not a problem.

B) Layer specific parameters, here we will discuss parameters specific to a specific layer type in a neural network:

1) MLP parameter (Multi-Layer Perceptron): Since this represents the (fully) connected layer in the network, we don't only care about how many neurons are in this layer, but also the neurons in the front and rear layers. First enter the count of layers. Since all layers are fully connected, this value is actually the number of outputs from the previous layer. Second, the count of neurons in the layer, that is, the number of outputs of the corresponding layer.

The selected Dens Layer Specific parameters used for predicting inference time are:

['batchsize', 'dim_input', 'dim_output', 'memory_weights', 'peak performance', 'cores', 'clock', Activation Function, optimizer].

2) Convolutional parameters: relate to those parameters relevant for convolutional layers within a network. These include "matrix size", which is of the input data to train on, "kernel size", which is the filter applied to the image data, and "input depth", which is the count of channels or levels in the input data , "output depth" is the count of channels or layers in the output data, "stride" is the step size to be taken by the convolution kernel, and "input padding" is the count of edge layers that add zeros to the outside of the matrix to allow correct analysis of edge pixels [25]. The selected Convolutional Layer Specific parameters used for predicting inference time are:

['batchsize','matsize','elements_kernel', 'channels_in', 'channels_in', 'kernel_size', 'padding', 'strides', 'use_bias', 'memory_in', 'memory_out', Activation Function, optimizer].

3) Pooling parameter: It is a parameter unique to the pooling layer in the network. Parameters include: 'Kernel Size' of the pooling kernel, 'Stride Size' is the stride of the pooling kernel, 'Input Padding' is the edge layer matrix of zeros added to the outer to allow correct analysis of edge pixels.

4) Recurrent parameters: Indicates those parameters that define the RNNs. RNN contains the same parameters as MLP (Multi-Layer Per Ceptron) i.e., the count of inputs and the count of neuroses. Additionally, they include: First, "Recurrent types", there are many types of recurrent that define one-hot encodings for these types, including: default, LSTM, and GRU. Second, "Bidirectional" is a binary value indicating whether or not the RNN is bidirectional. Additional parameter sets and individual parameters can be added, but these are considered core parameters that cover most deep neural networks in use today.

In order to obtain an execution time prediction model for a deep learning network, we developed our own fully connected feed-forward deep learning network, trained on the set of parameters defined above and the actual 'training' run of the deep learning network results. Our neural network architecture consists of m layers with jn neutrons in the nth layer. Each layer is followed by a dropout layer for the training only, and the last dense layer produces a single output. To create an accurate deep learning predictor, we evaluated multiple runs of our predictor using various parameters. Optimization is the process of finding optimal parameters for a model that significantly reduces the error function. The parameters considered in this optimization process are m, jn ($n \in [0, m]$), dropout rate, loss function, activation function, and model optimizer. Activation function (ReLU): An activation function is a simple mathematical function that transforms a given input into a desired output with a specific range. It is so called because it activates the neuron when the output reaches the set threshold of the function. Basically, they are responsible for turning neurons on/off. Neurons receive the sum of input products and randomly initialized weights, as well as predicting static biases for each layer in the model [25].

An activation function is applied to this sum and produces an output. The activation function helps applying to this sum and provides an output. The activation function provides (non-linearity) to allow the DNN to learn complex types in the data, like images of plants.

Without an activation function, our model works like a linear regression model with small learning ability. We already mentioned in our answer that activation functions can contain ReLU, Softmax, Sigmoid, Tanh and None. A linear activation function (ReLU) is a linear function that outputs immediately if the input is regular, and zero otherwise.

It is the most effective activation function in (CNN). So (ReLU) is more efficient than Sigmoid or Tanh. If an optimizer like SGD (Stochastic Gradient Descent) is used during backpropagation, it works like a linear function with positive values, so it becomes easier to compute the gradient. This near-linearity allows property preservation and makes it easier to optimize linear models using gradient-based algorithms. Additionally, ReLU increases the sensitivity of the weighted sum, preventing the neuron from becoming saturated (i.e., when there is little or no output change). Dropout Rate: is a very effective parameter that ignores randomly selected neurons (nodes) during training. They will be "eliminated" at random. This collaboration to activate downstream neurons is temporarily eliminated in the forward pass where there is no weight updates to the neurons in the backward pass. If a neuron drops out of our DNN during training, other neurons must step in and process the required representation to predict inference-time for the lost neuron. This causes the network to learn multiple independent internal representations. Dropout is only used during model training, not when evaluating model skills. Dropout can be applied in the body of a DNN model. We used a dropout rate of 20% and limited the weights of these layers. Loss function: to measure the error, we took the contrast of the actual output and the predicted output. The function we used to measure the error is called the (loss function) or (cost function). The main focus is the parameter estimation using mean squared error (MSE). MSE is a very important parameter. It is often used in linear regression as a performance measure.

Model Optimizer: The optimizer is helpful to update the weights and biases to reduce errors. for prediction error or loss minimization, the model updates the model's weights parameters as it learns examples from the training set. These error calculations, when plotted against weights, are also called cost function plots, as they determine the model's cost. The most important technique and foundation for our training and optimization of models is the use of gradient descent.

It is used when we plot a cost function to find a way to reach this minimum cost. In this algorithm, it starts with random model parameters, computes the error for each learning iteration,

and then upgrades the model parameters to approach the values that provide the minimum cost. The gradient is measured with respect to each model parameter. The cost function takes the variable of the number of parameters [1 to n]. (α or alpha), is the learning rate of how fast we want to approach the minimum. The learning rate we use in the prediction model is equal to 0.01.

There are three ways to perform gradient descent: One is batch gradient descent, which uses all training samples to upgrade the parameters at each iteration. Second, Mini-Batch GD splits the training set into smaller sizes called batches. This mini-batch is used to upgrade the model parameters in each iteration. Third, Stochastic Gradient Descent (SGD), uses only one training sample per iteration to upgrade parameters. Training instances are usually chosen randomly. When there are hundreds of thousands or more training examples, (SGD) is often preferred to optimize the cost function for faster convergence than batch gradient descent. There are other important optimizers, I'll mention them and talk about the specific optimizer (Adam) we used in our prediction models.

They are RMSprop, Adagrad, and Adam. 'Adam' is Adaptive Moment Estimation, which is another method that uses previous gradients to find out the current gradients. Adam also uses the concept of momentum by adding the number of the previous gradient to the current gradient. This optimizer is very effective to train DNNs. We did a parametric search of various fully connected feedforward deep learning networks and identified different depths required for different problems. Also, using the dropout layer after the last hidden layer, using the ReLU activation function, and L2 regularization with a regularization constant of 10-5 by each layer of the network, helps generalization and maximizes the overall accuracy. To emphasize that in the importance of accurate results in common low-cost operations, we used log mean squared error (RMSE) in the loss function. To minimize the loss, Adam's decreasing learning rate optimizer is used.

Ou model allows for deriving the execution time of each layer of the DL model. Therefore, it is used to predict the influence of arbitrary changes on the model architecture or parameters such as batch size or optimizer used. Our deep learning predictor approach nicely demonstrates the different computational time scaling properties of different batch sizes and different layers. While batch size has little effect on the execution time of fully connected layers, the execution time of convolutional layers increases greatly with increasing batch size.

Therefore, for small batches, fully connected layers dominate, while for larger batches, convolutional layers dominate. While these results show some variation, especially for large batch sizes, they are helpful to infer the computational complexity of a given model and estimate the training time for an epoch. Furthermore, it is possible to predict the impact of different hardware and different model properties, such as, optimizer on the resulting execution time. Let me discuss some hardware parameters that can be included in predicting the execution time of a DNNs. Hardware parameters describe the CPU or GPU used and the system on which the GPU card is located. For example, some available hardware parameters like number of cores are the number of processing units (core-count), maximum clock speed recorded in Hertz (clock-speed), hardware memory is the memory available per hardware i.e., per CPU or per GPU, Hardware Count is the number of CPUs in the system (or GPU), Hardware Memory Bandwidth is recorded in GB/s, and lastly, hardware peak performance is recorded in GFLOPS that is a result of CPU clock speed and CPU core count.

From the above considerations it can be concluded that speed, feed, and depth of cut are the most important parameters. Optimizing these parameters results in a better surface finish. The use of parametric design techniques is considered to be a successful and efficient method for optimizing machining parameters, which tends to reduce inference time and increase productivity. The results of this model can be used to predict the execution time of a full deep neural network. This allows the model to provide a good basis for making informed decisions when choosing the appropriate hardware to train the model on or derive predictions from, while also helping to make informed decisions about model design and layout.

Figure 18 shows the coloration matrix of selected convolutional parameters that we used to predict the inference time. Also, Figure 19 shows the coloration matrix of selected dense parameters used to the inference time.

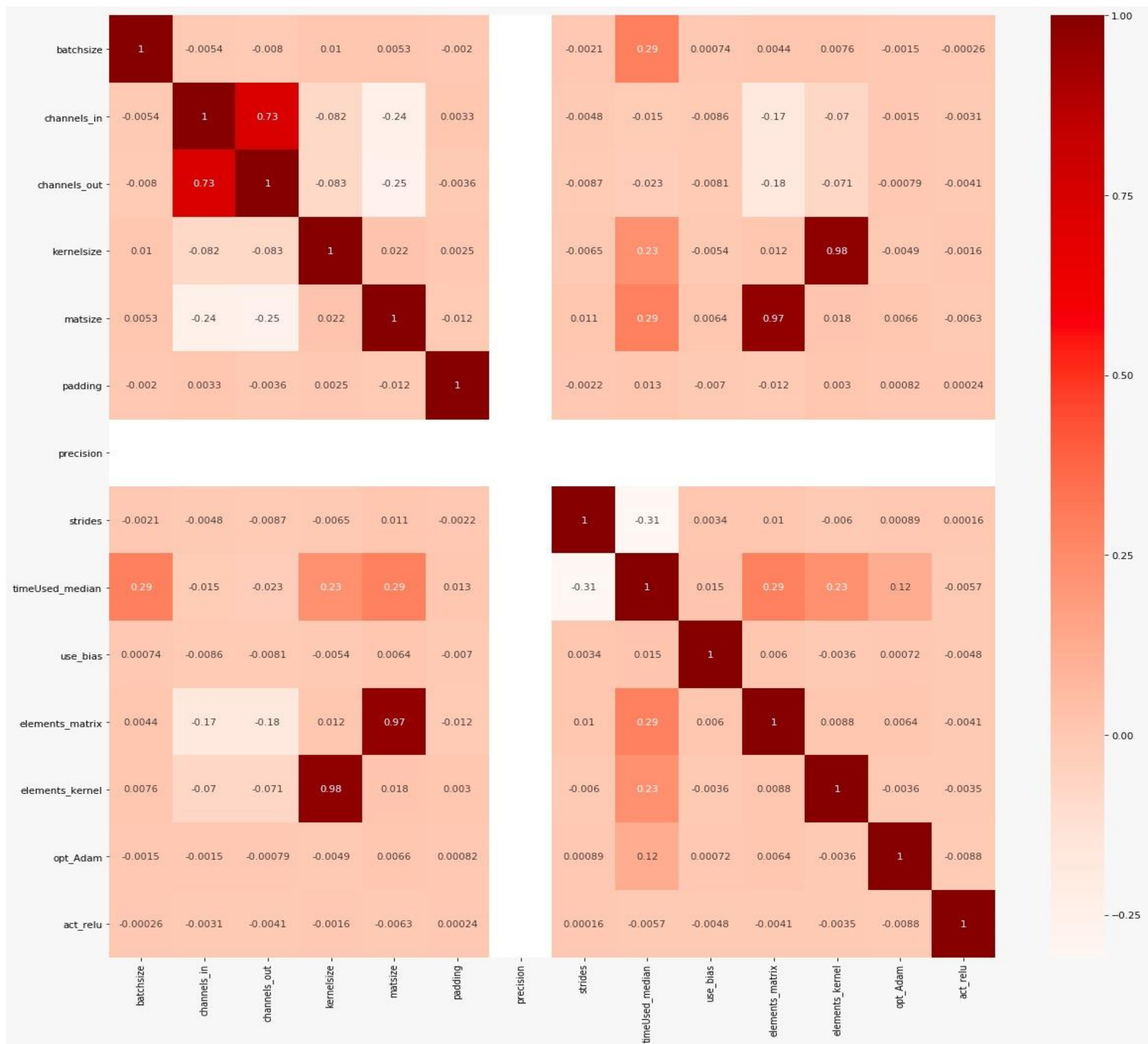


Figure 18. Coloration Matrix of All Selected convolutional parameters used to predict Inference Time

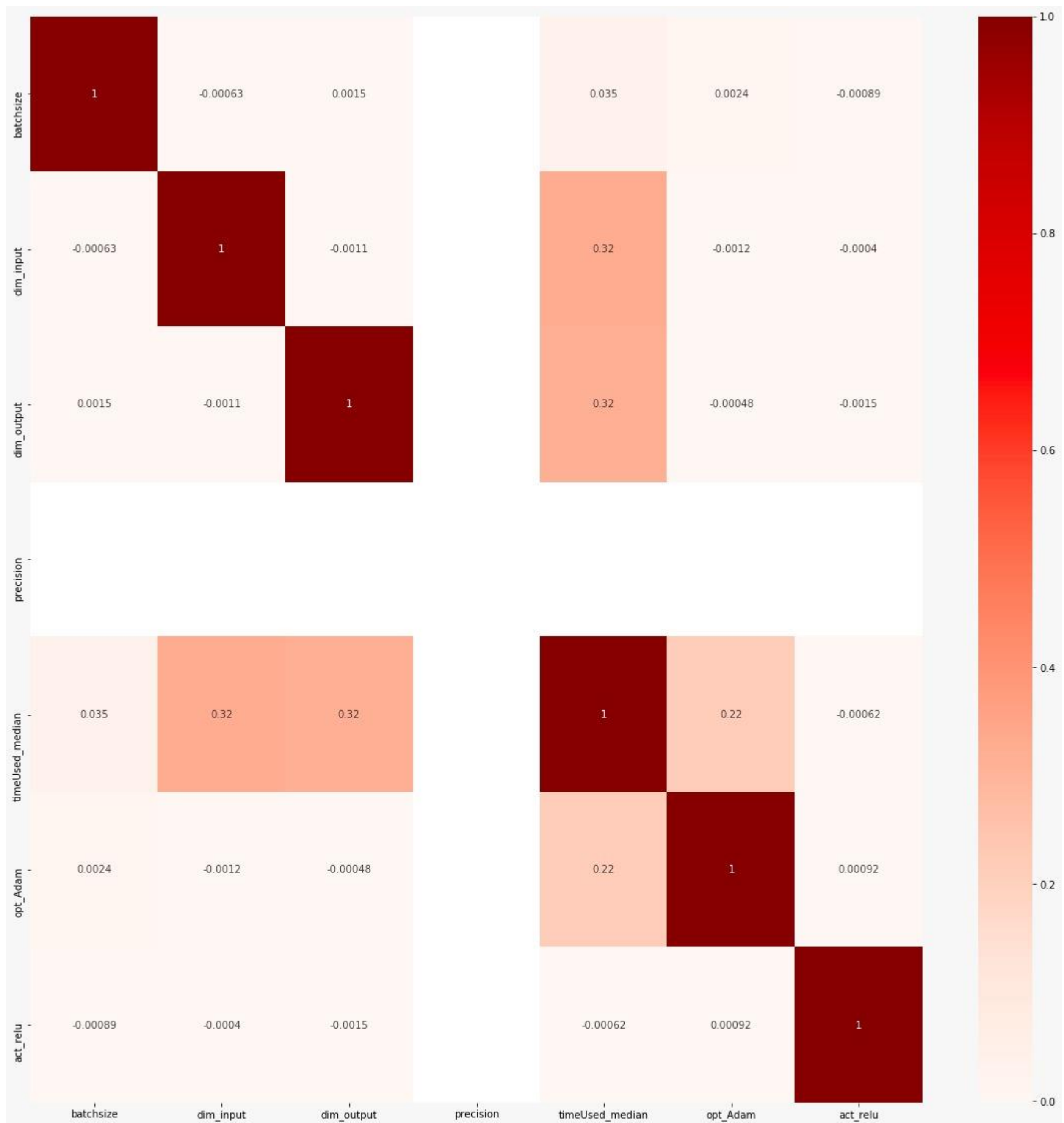


Figure 19. Coloration Matrix of All Selected Dense parameters used to predict Inference Time

The following Table 3 shows the correlation between each parameter we discussed above and the target (inference time). In order to select a highly correlated parameter, the correlation between the parameter and the inference time is assumed to be greater than (0.01). By this table, we conclude our answer for this question by defining the main parameters that need to be included in predicting the execution times for a DNN.

Table 3. The Coloration between each Parameter and Inference Time

Parameter	Coloration with Inference Time
batchsize	0.292164
matsize	0.286154
kernelsize	0.226384
channels_in	0.0159185
channels_out	0.0249471
padding	0.054595
elements_matrix	0.287445
elements_kernel	0.226123
memory_weights	0.030074
memory_in	0.404868
memory_out	0.378331
act_None	0.3316411
act_sigmoid	0.020235
opt_ASGD	0.10551
opt_Adam	0.1153059
dim_input	0.323868
dim_output	0.318699
bandwidth	0.298580
cores	0.279249

Methodology

Our approach here is to break up each deep learning model into layers and consider layer as the atomic operation in DNN models. Then, we train an FFNN (Feedforward neural network) on a large set of layers of random features. By using the time recorded as a target, we test the model accuracy over benchmarked models to validate our approach. The prediction is done by summing all the predicted inference times of chosen layers in a DNN architecture. By this approach, we

can significantly save resources and time, and eliminate the need to test models on hardware to determine if they will fit a certain application.

A. Model Architecture

To obtain a predictor for our approach, we used the same model architecture used in [18] to predict the training time of deep learning models. This model has m layers and Jn neurons in layer n. Each layer is followed by a dropout layer except for the last layer which consists of one neuron to produce the predicted inference time.

To obtain an accurate deep learning predictor, we evaluated numerous runs of our model with different parameters. The parameters considered during the optimization process included dropout rate, loss function, and the model optimizer.

B. Model Prediction

After training the model as discussed, we need to split the benchmarking models into layers to produce output. Then, we use the convolutional layers as an input to our model, and sum over all the layer inference times to get the inference time for the whole model:

$$T = \sum_{i=0}^{N} Ti$$

Where *"N"* is the total number of layers, *"i"* is the count of the current layer, *"Ti"* is the predicted inference time of *"i"* the layer, and *"T"* is the total predicted inference time for a model.

Experimental Setup

The CPU features are an 11th Generation Intel(R) Core i5 device, 4 cores, 8 logical processors, and 8 GB of RAM. Multiple platforms can be worked with such as (CPU, GPU, TPU, VPU), but for the sake of simplicity, we went with CPU only.

A. Test Case: Fully Connected Layer

To evaluate our deep learning predictor for fully connected layers we used the implementation torch.nn.Linear to generate fully connected layers. We tested 30,000 of the possible parameter

combinations. Our batch sizes are in the range of 1 to 64, input and output dimensions in the range of 1 to 4096.

B. Test Case: Convolutional Neural Layer

To consider the convolutional layer, we used a torch. Conv2d layers with features in ranges of 1 to 64 batch size, 1x1 up to 512x512 matrix size, etc in the process of testing the model, and as a part of the generated model. Then, we predict the time used for each convolutional layer.

C. Data Collection and Preparation

Features sets were randomly selected with no independence on the previous choice. For each feature, we performed two benchmark runs and used the median of used time as input to our model. The data was split into (80%) training data, (10%) validation data, and (10%) testing data. We can see in Figure 20, the data distribution in case of backpropagation or no backpropagation between the number of operations for a layer on the "x" axis and time recorded for inferring this layer on the "y" axis.

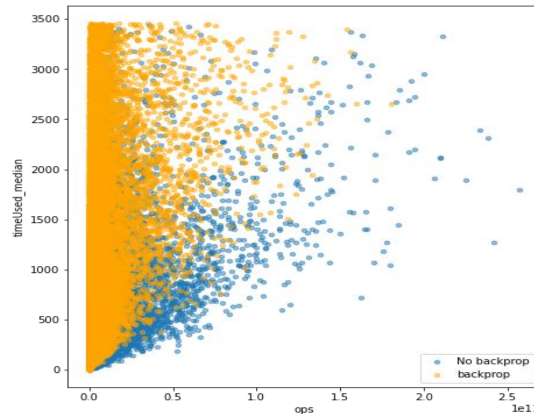


Figure 20. Data Distribution in Case of Backpropagation or No Backpropagation

Minimum requirements in terms of hardware and software to achieve the results in a reasonable amount of time and resources:

Hardware

This section describes the hardware features used to predict the inference time for deep neural networks. Despite the high financial cost of training deep learning networks, little research has been done on predicting execution time and choosing the best hardware.

In order to find the suitable hardware for inference time prediction for DNNs. It depends on the operations we want to perform, which in turn depends on the size and the datasets we'll mostly be working with. Let's take a look at the CPU first. However, GPUs are what deep learning is all about. After all, the deep learning "ImageNet moment" is based on an introduction by Alex [23]. Training convolutional neural networks like AlexNet on GPUs are more efficient than on plain old CPUs. Older CPU hardware has very long execution times and larger absolute deviations in predictions. A suitable CPU makes a big change, mostly in training that inherits from good multithreading, e.g., running multiple physics simulations for reinforcement learning can achieve optimal performance while ensuring cost-effectiveness.

The inference performance directly affects the results. This means it is critical to carefully analyze the hardware parameters that affect model performance and make sure that the selected hardware is suitable for performance targets. Depending on performance targets, we consider different hardware parameters when finding the hardware such as memory, clock speed, memory bandwidth, core count, and peak performance. We will explain the two examples below to illustrate this relationship. If the performance goal is to minimize latency, then we need to consider memory when choosing hardware because memory is not a bottleneck, we can choose cheaper hardware and reduce costs. If maximizing throughput is the performance goal, then the hardware must be able to support high memory bandwidth to handle large batches [24].

Starting from the model and the resources required by the hardware; we definitely realize which hardware can best process the data at a given time. For example, let's check the latency performance of (ResNet-50) on a number of the hardware. When using the hardware (Skylake int8 CPU), the inference time is (116ms), but when using a (2080TI TRT16), the inference time is (1.27ms). This is a very big difference in performance [24]. On a GPU or CPU device, the inference time (latency) of a model depends on the ability to work with large batches. The reason is that running the same model on varied hardware provides a range of throughput performance, it is essential to realize whether the hardware you choose can support the needed target and performance. Our approach is aimed to match the computation resources limitations of the hardware to provide good accuracy and avoid any failure during running the model.

Our CPU power is an 11th Gen Intel(R) Core i5 - Device, 4 cores, 8 logical processors, and 8 GB RAM. It can be used with multiple platforms such as (CPU, GPU, TPU, and VPU), but for simplicity, we have chosen only CPU. Related work [25] made a comparison of performance of

predicted execution times for different hardware by using five different NVIDIA Tesla GPU cards (V100, P100, M60, K80, K40). Each has a different clock, memory, cores, and memory bandwidth features. V100 has 1455 MH_z clock, 16 GB memory, 5120 cores, and 900 GB/s memory bandwidth. P100 has 1303 MH_z clock, 16 GB memory, 3584 cores, and 732 GB/s memory bandwidth. M60 has 1178 MH_z clock, 16 GB memory, 4096 cores, and 320 GB/s memory bandwidth. K80 has 875 MH_z clock, 12 GB memory, 2496 cores, and 240 GB/s memory bandwidth. K40 has 875 MH_z clock, 12 GB memory, 2880 cores, and 288 GB/s memory bandwidth. For this approach evaluation [25], the model was trained on these individual GPU cards as well as on a combined set of GPU cards. The measured time and predicted time for convolutions in (Tesla V100), the RMSE is 1.86 ms. In (Tesla P100), the RMSE is (3.50 ms). In (Tesla M60), the RMSE is (9.07 ms). In (Tesla K80), the RMSE is (8.12 ms). In (Tesla K40), the RMSE is (12.73 ms).

In the following Table 4, we show the hardware requirements that we used in our inference time prediction model for a DNN that performed 2.98 ms of predicted time for convolutions versus measured time.

Table 4. Lenovo ThinkPad L14 Gen 2

Feature	Specification
Processing (CPU)	11th Gen Intel(R) Core i5 - Device, 4 cores, 8 logical processors (2.40GHz)
Memory	8 GB DDR4-3200MHz
Hard Drives	256 GB SSD M.2 2280 PCIe TLC Opal
Display	14" FHD (1920 x 1080),4K 282ppi IPS LCD glossy
Graphics (GPU)	NVIDIA GTX 1050 GPU with 4GB RAM

Software

This section describes the software features that have been used in predicting inference time of DNNs. We implemented a benchmark framework for DNN comparison in Python. We used PyTorch package for NNs processing with cuDNN-v5.1 and CUDA-v9.0 as backend. All code used to estimate hypothetical performance metrics, and all considered DNN models are written in Python. All pre-trained models expect the input image to be normalized in the same way, i.e., mini-stack of RGB images of the form 3 × H × W, where H and W are expected to be

(330) pixels for the NASNet-A-Large model; - (220) pixels for InceptionResNet-v2, and (223) pixels for all other models. For testing our approach, we used 12 DNN model architectures unseen in the training data and pretrained them on the ImageNet dataset. We will briefly describe the analyzed 12 model architectures. We have chosen different architectures; some are designed to be more effective, and others are more efficient and therefore better suited for certain applications. The model architectures are Bninception, Cafferesent101, Mobilnet_v2, Densenet121, Densenet161, Densenet201, Dpn68, Googlenet, and Nasnetamobile. Resnet34, Resnet18, and Resnet50. In some cases, the name of the schema is followed by a number. Such numbers indicate the number of convolutional layers or fully connected layers that are containing the parameters to be learned such as:

1. Alex Net; VGG architecture family (VGG-11, -13, -16, and 19), with and without batch normalization (BN) layers.
2. BN-Inception, GoogleNet, ResNet34, DenseNet-121 and -201 have a growth rate of 32, and DenseNet-161 has a growth rate of 48.
3. ResNet50 (32x4d) and ResNet18 (64x4d), where the numbers in parentheses indicate the number of groups and the bottleneck width of each convolutional layer, respectively.

In addition, we also consider the following efficiency-oriented models, such as MobileNet-v2. In the result section below, we found the average inference time per frame over 10 runs for the 12 DNN models considered. In our approach, we used 7 batch sizes on our device (CPU) equal to 1, 3, 6, 12, 24, 48 and 96. Inference time is measured in milliseconds, and the entries in Table 1 are color-coded for easy conversion to frames per second (FPS). We found that the mobilenet_v2 architecture can achieve hyper-real-time performance (0.12 ms) on our CPU when the batch size is 3. Other models achieve the lowest real-time performance under the same batch size 3. Namely: densenet121, desnet34, densenet201, cafferesnet101, and densenet161.

This DNN prediction approach illustrates different computation time scaling performances for different batch sizes and different layers. In our paper, we show that we created a deep learning model that can outperform classical methods by predicting inference time. By summing the inference time of the deep learning architecture layer by layer and treating each layer as an atomic operation of the model. Our method can be generalized to include more hardware and software features. This is time and money-consuming and extends the time-to-market for many products that use deep learning for each task.

The following Table 5 shows the software requirements that we used in our inference time prediction model for a DNN.

Table 5. Software Requirements

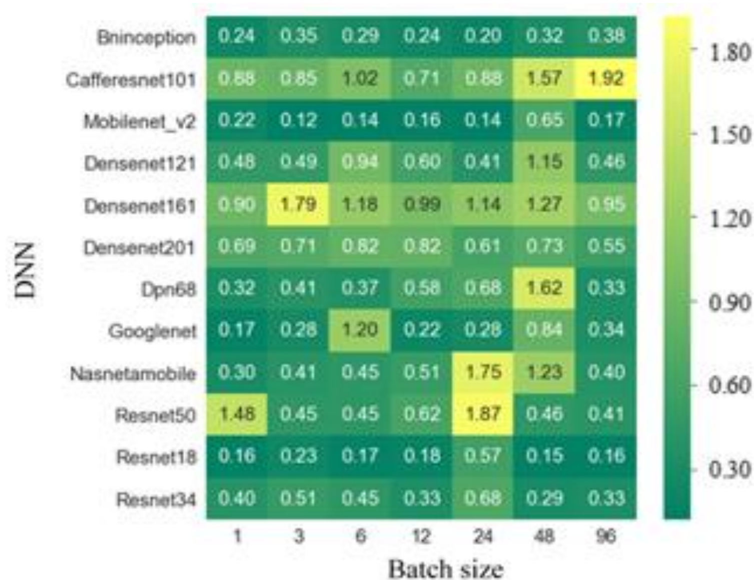
Software	**Platform**	**Cost**	**Written in language**	**Algorithms or Features**	**Version**
Scikit Learn	Linux, Mac OS, Windows	Free.	Python, Cython, C, C++	Classification Regression Clustering Preprocessing Model Selection Dimensionality reduction.	0.20
PyTorch	Linux, Mac OS, Windows	Free	Python, C++, CUDA	Autograd Module Optim Module nn Module	1.9.0
TensorFlow	Linux, Mac OS, Windows	Free	Python, C++, CUDA	Provides a library for dataflow programming	Version 2
Keras.io	Cross-platform	Free	Python	API for neural networks	2.3.0
Anaconda	Mac OS, Windows	Free	Python, C	Supports libraries of PyTorch, Keras, TensorFlow, and OpenCV	2.1.0
Python	Linux, Mac OS, Windows	Free	C	Supports libraries of PyTorch, Keras, TensorFlow, and OpenCV	2.7

Results

A. Model Architecture Inference Time

Here we find the Average per image inference time over 10 runs for the 12 DNN models considered. They are reported in Table 6. We used 7 batch sizes equal to 1, 3, 6, 12, 24, 48, and 96 on our device (CPU). Inference time is measured in milliseconds and the entries in Table 5 are color-coded to easily convert them in frames per second (FPS). From the table provided, the architecture mobilenet_v2 can achieve super real-time performances (0.12 ms) on our CPU, when a batch size of 3 is considered. Other models achieved the lowest real-time performances with the same batch size 3, namely: densenet121, desnet34, densenet201, cafferesnet101, and densenet161.

Table 6. Inference Time Per Image Vs. Batch Size of our CPU



DNN	1	3	6	12	24	48	96
Bninception	0.24	0.35	0.29	0.24	0.20	0.32	0.38
Cafferesnet101	0.88	0.85	1.02	0.71	0.88	1.57	1.92
Mobilenet_v2	0.22	0.12	0.14	0.16	0.14	0.65	0.17
Densenet121	0.48	0.49	0.94	0.60	0.41	1.15	0.46
Densenet161	0.90	1.79	1.18	0.99	1.14	1.27	0.95
Densenet201	0.69	0.71	0.82	0.82	0.61	0.73	0.55
Dpn68	0.32	0.41	0.37	0.58	0.68	1.62	0.33
Googlenet	0.17	0.28	1.20	0.22	0.28	0.84	0.34
Nasnetamobile	0.30	0.41	0.45	0.51	1.75	1.23	0.40
Resnet50	1.48	0.45	0.45	0.62	1.87	0.46	0.41
Resnet18	0.16	0.23	0.17	0.18	0.57	0.15	0.16
Resnet34	0.40	0.51	0.45	0.33	0.68	0.29	0.33

B. Best DNN at Given Time

Table 7 shows the best DNN architectures in terms of inference time. This analysis is done for our device (CPU). The high-quality model was obtained with the lowest inference time shown in Table 7. Mobilenet_v2 was the Top 1 model where the inference time was (0.12 ms). Then, resnet34 was the fifth model with an inference time of (0.288 ms). Also, Googlenet ranked average among the top five deep networks.

Table 7. Inference Time with Top 5 Models

N.	Architecture Models	Inference time (ms)
1	Mobilenet v2	0.119
2	Resnet18	0.150
3	Googlenet	0.171
4	Bninception	0.201
5	Resnet34	0.288

C. Estimating Inference Time for Convolutional and Fully Connected Layers

In this section, we show the results of training our model using the features we defined above. We explained that our approach uses CPU and can be easily generalized to any hardware and any type of layer.

Here we train a predictor deep learning model with architecture as stated above in the model section. Then, we generate predictions of inference time for fully connected layers implemented using the TensorFlow framework and hardware specifications as mentioned above.

Figure 21 shows the RMSE and number of FLOPS when predicting CPU performance using models with different numbers of hidden layers. We can notice that there is an inverse relationship between RMSE and number of FLOPS with several hidden layers. The 6-layer has the lowest RMSE and the highest number of FLOPS. However, the benefits to the 4-layer network are negligible when compared to the increased training time. Following these results, we use four hidden layer neural networks to predict CPU performance for less computational complexity and less RMSE.

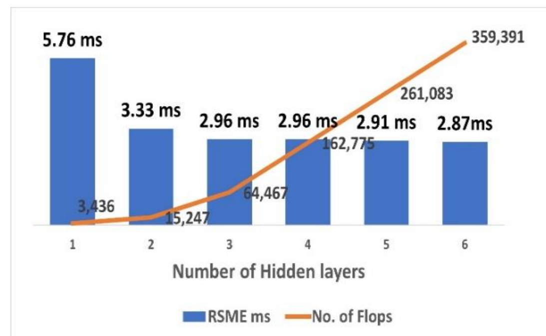


Figure 21. RMSE of Inference Time Predictions and Number of FLOPS for Convolutional Layers on CPU Using Models with Different Numbers of Hidden Layers

Figure 22 shows the result of using a deep learning predictor to predict the inference time of the feedforward path. Figure 23 shows the result of a linear regressor [19]. The linear regressor clearly cannot capture the complexity of the model and produces a large amount of inaccuracy (RMSE 9.3 ms). In contrast, deep learning predictors provide consistent predictions compared to the actual measured time (RMSE 2.98 ms). This suggests that deep learning-based predictors are much better to use in such cases. This is probably the result of the linear regressor not being able to explain the non-linear relationship between the feature and the inference time.

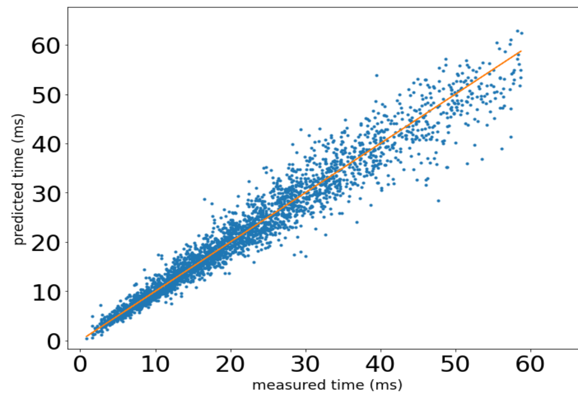


Figure 22. Predicted Inference Time for Convolutions on CPU Versus the Measured Time for the Deep Neural Network Model of a Forward Pass Only (RMSE 2.98 ms)

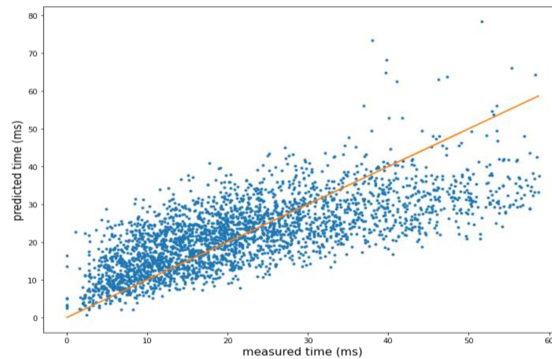


Figure 23. Predicted Inference Time for Convolutions on CPU Versus the Measured Time for Linear Regression Model for a Forward Pass Only (RMSE 9.3 ms)

Our work designs and develops an inference time prediction framework for deep learning models that is general and easily scalable to a large number of devices. Our key idea is to decompose a given model inference into layers and perform hierarchical predictions. Our experiments show that this strategy has a significant advantage in prediction accuracy. Compared to linear methods, this has the advantage that more complex situations can be simulated. But it is also able to predict inference time for cases not seen in the training data.

In our approach, we propose to train deep neural networks using parameters from computing resources such as RAM, CPU, etc. Here, we propose a framework to predict the inference time of deep learning models on CPU devices by performing inference time prediction for each layer of a given DNN model. Our proposed model is trained by generating a dataset of 60,000 data points, which is discussed in our article. The model is evaluated against 13 benchmark deep learning architectures.

In this approach, we train DNN to predict the inference time of the DL network parts. These individual parts of time can then be provided predictions for the entire model. The approach reduces computation time while maximizing the types of layers we can predict using atomic operations. To train our method, we generate convolutional layers with random parameters on our hardware device for convenience. However, our method can be generalized to any hardware by adding hardware parameters to the training data, e.g., (clock frequency, FLOPS frequency). It can be done by just forward propagation or the full cycle. Therefore, the method we propose can make decisions when choosing the appropriate hardware to train or derive the model and help with model design decisions. We have a highly flexible model that generalizes to various network models, data sizes, and hardware configurations. Our method produces incredibly good results by using only convolutional layers, as this is the main contributor to inference time.

Our approach here is to divide any deep learning model into layers and treat layers as atomic operations in a DNN model. Then we train an FFNN (feedforward neural network) on a large number of random parameters layers. Using the recorded time as the target, we test the reliability of our model versus the benchmark model to validate our method. Prediction is done by summing all predicted inference times for selected layers in the DNN architecture. This approach allows us to save a lot of resources and time and eliminates the need to evaluate models on hardware to determine if they are suitable for a particular application.

Chapter Summary

While highly desirable, latency prediction is also expensive. Due to the varied models inference latency brought on by the runtime optimizations on different edge devices, it is highly difficult and current techniques are unable to attain a high level of prediction accuracy. This chapter reviews the possible challenges involved in the prediction of latency. Further, it comprehensively discusses the various DNN models that don't only reduce the inference time but also predict it accurately. It is found from this chapter that inference time is dependent upon the computational complexity, data size, and features. Accuracy is reciprocal of inference time. Computational complex models and large number of features reduce the inference time. Also, here we discussed here the strengths and weaknesses of various deep neural network models for latency prediction, various real-world application domains where latency played a crucial role, and the limitations and challenges of current methods for latency predicting in Deep Neural Networks (DNNs) in real-world applications. The resolution of these limitations would be the possible future directions in latency prediction. We demonstrated that we created a deep learning model that can outperform the classical approaches by predicting inference time. By summing the inference time layer by layer for deep learning architectures and think of each layer as the atomic operation of a model. This approach can be generalized more to include more hardware and software features. This will save lots of time and money and will increase the time to market many products that are using deep learning for any task.

CHAPTER 5

MODEL COMPRESSION FOR EFFICIENT DEEP NEURAL NETWORKS BASED ON PLANT DISEASE CLASSIFICATION

Chapter Overview

Presently, due to the expeditious advancement of deep learning, deep neural networks (DNNs) have been extensively utilized in diverse computer vision endeavors. Nevertheless, in the quest for improved performance, sophisticated deep neural network (DNN) models have grown increasingly intricate, resulting in a substantial consumption of memory and significant computational requirements. Consequently, the use of these models in real-time scenarios poses challenges. In order to tackle these concerns, the research community has directed its attention into the field of model DNN compression. In addition, the utilization of model compression approaches holds significant importance in the deployment of models on edge devices. This chapter conducted a diverse technique for model compression with the aim of aiding researchers in the reduction of device storage capacity, acceleration of model inference, simplification of model complexity and training expenses, and enhancement of model deployment. Therefore, this chapter provides the latest advancements in model compression methods. These methods encompass model pruning, parameter quantization, low-rank decomposition, knowledge distillation, and lightweight model design. Furthermore, we evaluated the beforementioned approaches of DNN compression on our classification models and analyzed the obtained results.

Challenges and limitations

In addition to the evident benefits discussed in the preceding section, there are certain acknowledged difficulties and constraints associated with the application of deep neural network compression methods. One of the foremost obstacles is in the task of compressing models while minimizing any substantial degradation in their predictive capabilities. Achieving a harmonious equilibrium between the decrease of size and the preservation of accuracy is a goal that is not easily accomplished [259]. The development of efficient compression methods necessitates the consideration of the algorithmic complexity associated with deep learning models. In order to achieve a balance between compression ratios and minimal performance loss, it is necessary to employ advanced approaches [258]. The development of efficient compression methods necessitates the consideration and mitigation of the algorithmic complexity associated with deep learning models. The achievement of optimal compression ratios while minimizing performance degradation necessitates the utilization of advanced methodologies. The existence of heterogeneous hardware configurations across various devices presents a significant obstacle when attempting to develop compression techniques that can be universally used. The inclusion of several hardware architectures in the optimization of models introduces a level of intricacy to the compression process [259].

In relation to the constraints associated with the implementation of deep neural network (DNN) compression methodologies, a predominant limitation arises from the application-specific issues. These challenges pertain to the efficacy of compression strategies, as their effectiveness may fluctuate depending on the particular application under consideration. The utilization of specific compression techniques, such as quantization, has the potential to result in a reduction in the interpretability of models. The importance of comprehending model decisions is evident in applications, as highlighted by Han et al. [258]. Finding a compromise between attaining optimal compression ratios and ensuring flexibility for various datasets and applications might provide a significant challenge. According to Cheng et al. [259], an excessive application of compression techniques can result in a decrease in the flexibility of the model.

Model Compression

Deep learning models have achieved remarkable success across various domains, showcasing unprecedented accuracy and capabilities. However, the widespread adoption of these models is hindered by their inherent resource-intensive nature, posing challenges for deployment on devices with limited computational power and memory. The problem at hand revolves around the urgent need to compress deep learning models without sacrificing their predictive performance, thus addressing the following critical issues. As a result, the models are difficult to apply in real time. To address these issues, model compression has become a focus of research. Furthermore, model compression techniques play an important role in deploying models on edge devices. Considering these challenges, the compression of deep learning models emerges as a critical research area. Effectively addressing these issues requires the development of compression techniques that strike a balance between model size reduction and the preservation of predictive performance. The goal is to enable the deployment of efficient, fast, and energy-conscious deep learning models across a spectrum of devices and applications. Therefore, we will analyze various model compression methods to assist researchers in reducing device storage space, speeding up model inference, reducing model complexity and training costs, and improving model deployment. The state-of-the-art techniques for model compression, including model pruning, parameter quantization, low-rank decomposition, knowledge distillation, and lightweight model design.

DNN (either fully-connected, CNN or RNN) is made of a variety of layer types. Convolutional Neural Networks (CNNs) such as AlexNet [257] and VGG16, as described by Simonyan and Zisserman [249], incorporate both convolutional layers and dense layers in their architecture. Previous studies have indicated that deep neural networks (DNNs) impose significant demands on both storage and computational resources [258]. The majority of the parameters in deep neural networks (DNNs) are derived from the dense layer, while a significant portion of the computational resources is dedicated to executing multiply-accumulate (MAC) operations within the convolutional layer [261]. Li et al. [262] found that within the VGG16 model, the proportion of parameters in the dense layer compared to the convolutional layer is 90:10. Additionally, the study revealed that around 99% of the multiply-accumulate (MAC) operations are attributed to the convolutional layer. Significantly reducing the storage and

computation overheads of deep neural networks (DNNs) can be achieved by decreasing the number of parameters in the dense layer and the multiply-accumulate (MAC) operations in the convolutional layer. This section provides an elucidation of several strategies in Figure 24, provided by researchers to reduce the parameters of the dense layer, and minimize the number of multiply-accumulate (MAC) operations in feed-forward neural networks (NN) and convolutional neural networks (CNN).

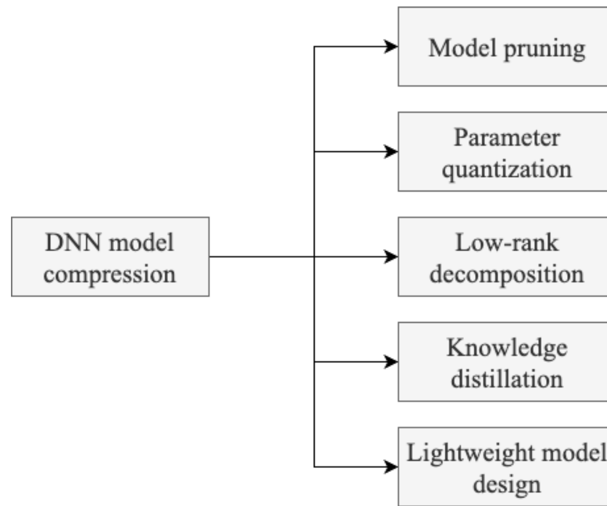


Figure 24. The common groups of DNN model compression techniques

Model pruning

The initial technique employed for pruning was known as biased weight decay [263]. During the 1990s, researchers employed the goal function within a Taylor expansion technique to identify the neuron that had the minimal effect on the loss [264]. The aforementioned techniques were primarily concerned with eliminating non-essential elements from deep neural networks (DNNs) while minimizing any noticeable impact on their overall performance. As the investigation advanced, the process of model pruning was categorized into two distinct methods: structured and unstructured.

1. Structured pruning

The conventional methodology for structured pruning entails employing a channel (or filter) as the basic element for pruning, as illustrated in Figure 25 [241], [264]–[266]. The process of channel pruning leads to the elimination of the corresponding channels along with it [262]. The method of channel-based structured pruning was employed to evaluate the importance of

particular channels. Li et al. [262] measured the relative significance of channels in each layer by calculating the total absolute weights of the channels. The methodology employed did not need the use of a sparse convolution library, nor did it result in the establishment of sparse connections. In contrast, it demonstrated a reduction in time consumption in comparison to the method of repeatedly refining the model layer by layer. The observed increase in efficiency resulting from time-saving measures was particularly evident while performing the pruning process on deep neural networks. However, as a result, there was a decrease in the model's performance. Therefore, Lin et al. [267] proposed a thorough and flexible pruning methodology with the objective of removing unnecessary channels.

In the first stage, a global discriminant function was utilized to remove the irrelevant channels across all layers, considering the historical global knowledge associated with each channel. Following this, the precision of the filters was continuously adjusted by comparing the pruned and sparse networks, to correct any mistakenly pruned channels. Following that, the model was subjected to a process of retraining with the aim of improving its overall performance. Furthermore, Li et al. [268] proposed a novel methodology that integrates max-average pooling with an upgraded channel-attention mechanism within deep neural networks (DNNs) to boost the representation of features. Kuang et al. [268] investigated to ascertain the importance of a channel. This was achieved by evaluating the influence of each channel on a loss function that is contingent upon the specific task. A reduction in the loss of function value is indicative of a drop in the relevance of the channel. The authors Li et al. [269] proposed a method for refining pruning techniques in deep neural networks (DNNs) at the software level. This approach led to enhanced efficiency in the hardware-level inference process [270].

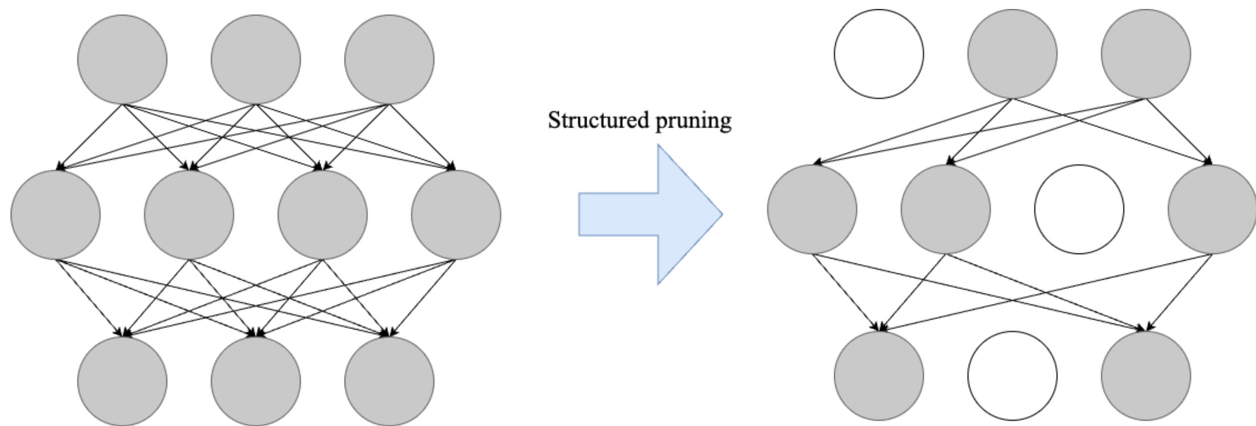


Figure 25. Generalized visualization of structured pruning

Channel-based pruning has been utilized in the domains of picture segmentation and object detection as well. In their study, Sawant et al. [271] introduced a strategy known as optimal-score-based filter pruning (OSFP) that aims to eliminate unnecessary filters based on their similarities in the feature space. The OSFP protocol implemented the removal of redundant filters, resulting in enhanced segmentation performance and quicker network learning. Sparse training [272] and mask learning [273] have been proposed as pruning techniques that introduce new connections while performing the pruning procedure.

In the domain of object identification, Chu et al. [274] introduced a model-compression approach consisting of three stages: (1) dynamic sparse training, (2) group channel pruning, and (3) spatial attention distilling. The technique of group channel pruning involves partitioning the network into several groups based on the size of the feature layers and the similarity of the module design within the network. Subsequently, the channels within each group underwent pruning based on varying criteria. Furthermore, Chang et al. [275] introduced an automated technique for channel pruning. The proposed approach initially conducts hierarchical channel clustering by simultaneously considering feature map similarity and early network trimming. Subsequently, a technique for population initialization was introduced with the aim of converting the trimmed architecture into a set of potential candidate populations.

Ultimately, the most efficient compression architecture was discovered through the utilization of particle-swarm optimization. Liu et al. [276] introduced a technique for network slimming that

involved assessing the efficacy of its parameters. Notably, this approach did not necessitate the utilization of specialized software or hardware accelerators for the model. Throughout the training phase, channels that were deemed unnecessary were automatically detected and subsequently eliminated. The utilization of a $L1$ regularization [277] was implemented to induce sparsity in the weights of the batch-normalization (BN) [278] layers. Subsequently, the technique of repetitive pruning was employed to get elevated rates of pruning.

The energy-aware pruning algorithm was proposed by Yang et al. in their study [279]. The procedure was directed by the algorithm, which utilized the computing resources of the convolutional neural network (CNN). The process of pruning was executed in a sequential manner, with each layer being pruned individually. This approach proved to be more efficient compared to previously suggested pruning techniques, since it focused on reducing mistakes in the output feature map rather than modifying the weights of the filters. To achieve this, the weights were initially trimmed on a per-layer basis. Subsequently, a process of local fine-tuning was conducted by closed-form least squares in order to restore the accuracy following the pruning procedure. Ultimately, the layers underwent pruning, and afterwards, the entire network underwent global fine-tuning through the utilization of back-propagation. In their 2021 study, Fan et al. [280] introduced a hierarchical channel pruning technique that involves grouping several layers to decrease the model accuracy of the pruned network. Following a predefined order, the network underwent retraining after pruning each layer. The network model experienced a slight decline in accuracy, while the computing resources allocated to the hardware were significantly diminished. To mitigate the computational burden associated with many training iterations, Chen et al. [281] introduced a training and pruning framework known as only-train-once (OTO).

The OTO approach significantly simplifies the intricate multi-stage training process employed in existing pruning methods. In addition, a novel approach known as the half-space random projection gradient method was introduced to address the issue of structural sparsity-induced regularization. In contrast to the need for several fine-tuning processes, OTO (One-Time Only) pruning only necessitated a single technique, hence greatly streamlining the pruning procedure. In their study, Chung et al. [282] conducted channel pruning on specific convolutional channels inside the initial layer of a pre-trained convolutional neural network (CNN).

The act of pruning the initial layer significantly aided in the process of compressing the channels in the subsequent convolutional layers. Nevertheless, the initial input to the first layer consisted of only one channel. To tackle these concerns, Chen et al. [283] put out a proposition to strategically modify neurons by the process of "grafting" suitable levels of linearized unimportant rectified linear unit (ReLU) neurons, with the aim of eliminating the non-linear elements. Nevertheless, to restore the performance of the model, it was necessary to optimize the corresponding slopes and intercepts of the replacement linear components. The original multiple pruning and fine-tuning methodologies were designed to be employed only once, despite the ongoing advancements in structured pruning algorithms, including layer-based and filter-based techniques.

2. Unstructured pruning

The technique of unstructured pruning in Figure 26 was developed using a heuristic methodology to eliminate insignificant characteristics, such as weight magnitude [258], gradients [284], and hessian [241] statistics. Historically, this phenomenon has commonly led to enhancements in competitive performance. However, the process of acceleration has encountered challenges mostly attributed to irregular sparsity [285].

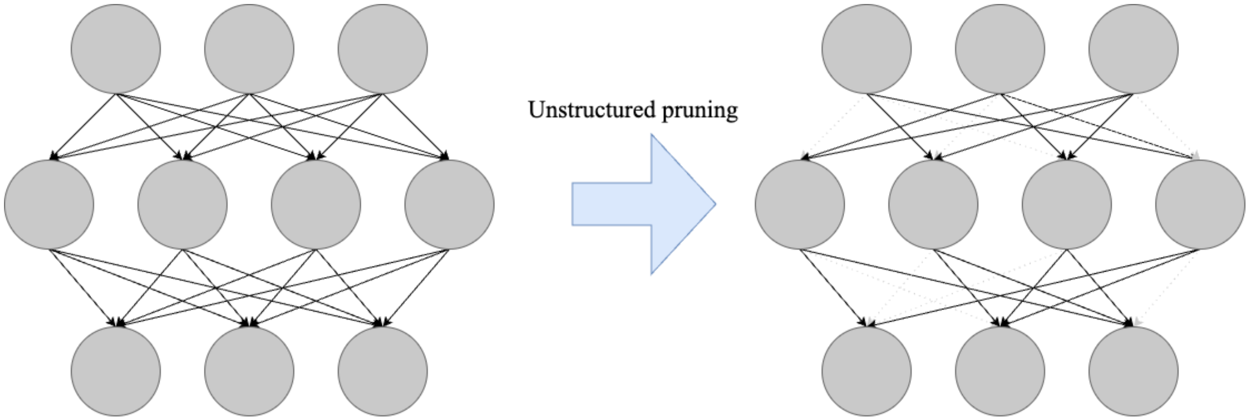


Figure 26. Generalized visualization of unstructured pruning

In the year 1989, LeCun et al. [241] proposed the notion of optimal brain damage, wherein second-derivative information was utilized to strike a balance between the complexity of the

network and the inaccuracy in the training set. This approach aimed to eliminate insignificant weights from the network.

In their study, Han et al. [258] introduced a technique known as train-prune-retrain, which aims to minimize the memory usage and computational requirements of neural networks by selectively learning the significant connections. The performance was enhanced by a factor of ten without compromising the precision. In their study [280] the authors employed the energy consumption of individual layers as a criterion for determining the order of pruning. In their study, Yang et al. [286] developed latency tables that employed a greedy algorithm to ascertain the optimal layers for cropping.

Additionally, Yang and colleagues [286] conducted comparative experiments employing L1 and L2 regularization techniques. Based on the empirical findings, it was observed that the use of L1 regularization in the pruning process resulted in higher accuracy compared to the utilization of L2 regularization, both in the context of post-pruning and without further retraining. This phenomenon transpired due to the application of L1 regularization, which resulted in a higher number of parameters being shifted towards zero. However, it was observed that L2 regularization had superior performance compared to L1 regularization following the process of retraining pruning. The authors Guo et al. [287] introduced a method called dynamic network surgery, which effectively decreased the complexity of a network by selectively removing connections in real-time. In contrast to the methodology employed in the previous study, Guo et al. incorporated linked splicing throughout the entire process to mitigate the risk of erroneous pruning. The incorporation of a learning process into the procedure of distinguishing between significant and insignificant parameters enabled a more precise determination of the ideal parameters. In their study, Neill et al. [288] used two weight regularizes with the objective of enhancing the alignment between units of pruned and unpruned networks.

This approach was employed to address alignment issues in pruned cross-lingual models. The implementation of unstructured pruning resulted in a significant reduction in both the number of parameters and computational requirements. Nevertheless, in the context of neural networks, unstructured pruning involves setting superfluous neurons to zero instead of completely removing them from the network [289]. Consequently, the potential benefits of non-regular sparsity in enhancing model acceleration were not fully realized in accordance with existing

hardware architectures. Hence, it is imperative to do further investigation into the optimization of unstructured pruning approaches for their application on contemporary hardware architectures.

Parameter quantization

The process of parameter quantization has been shown to effectively decrease the dimensions and inference duration of models [290], [291]. The process of parameter quantization has a high degree of versatility and may be effectively applied to a wide range of models and hardware devices. The process of parameter quantization in neural networks involves the conversion of the weights and activation values of a network model from a high precision representation to a lower precision one. There are several advantages associated with the use of this technology in an academic context. Firstly, it offers reduced storage overhead and bandwidth requirements, which may be highly beneficial for managing large amounts of data efficiently. Additionally, this technology is characterized by lower power consumption, which is advantageous in terms of energy efficiency and sustainability. Lastly, it enables faster calculation speed, which can significantly enhance computational performance.

Parameter quantization establishes a data-mapping relationship between fixed-point and floating-point data, allowing for better gains at a smaller cost in terms of accuracy loss. The generalized process is presented in following equations [292]:

$$S = \frac{R_{\max} - R_{\min}}{Q_{\max} - Q_{\min}}$$

$$Z = Q_{\max} - \frac{R_{\max}}{S}$$

where R represents a real floating-point number, Q represents a quantization fixed-point value, Z represents the quantization fixed-point value that corresponds to the zero floating-point value, and S represents the scale factor of quantization. Furthermore, S and Z serve as quantization parameters within the context of this study. It is worth noting that the data type of S is FP32, while the data type of Z is INT8. The values of Q and R are obtained from below. Specifically, Q represents the quantization value, while R represents the floating-point value that is back-propagated. If the values are beyond the upper limit of their respective ranges, it becomes

necessary to apply rounding procedures. The equation for converting floating point numbers to fixed point numbers is as follows.

$$Q = \frac{R}{S} + Z$$

The equation for inverse quantization from fixed point to floating point is as presented below:

$$R = (Q - Z) * S$$

where S and Z are found by the following the equation:

$$Z = Q_{\max} - \frac{R_{\max}}{S}$$

Following the process of quantization, it is typically necessary to adjust the parameters of the model. The process of acquiring a model through retraining is sometimes referred to as quantization-aware training (QAT). The process of acquiring a model without the need for retraining is referred to as post-training quantization (PTQ).

1. Quantization-Aware Training

The process of quantization introduces disturbances into the parameters of the trained model, resulting in a greater deviation from the convergence point compared to training with floating-point precision [293]–[295]. To achieve convergence towards a more optimal loss point, it is possible to address the issue by retraining the quantization parameters. One often utilized approach in this context is the Quantification during both forward and backward propagation (QAT) method [296], [297].

Nevertheless, the parameters of the model are quantified subsequent to every gradient update. It is crucial to execute this computation after the adjustments in weight using floating-point precision.

In the same vein, it is crucial to execute the backward transfer using floating-point operations. This is because the accumulation of gradients with quantization accuracy has the potential to result in significant mistakes in zero gradients or gradients, particularly when employing low-precision quantization.

2. Post-Training Quantization

Post-training quantization emerged as a viable alternative to Quantization-Aware Training (QAT) due to its ability to perform quantization and weight adjustment without the need for fine-tuning [294], [298], [299]. Hence, the expense associated with PTQ was exceedingly minimal and inconsequential. Furthermore, PTQ could be applied with limited or no labeling of data, which was a major advantage. Nevertheless, the process of Post-training Quantization (PTQ) necessitated a substantial amount of training data for retraining purposes. Regrettably, the resulting accuracy rate was relatively low, especially when it came to low-precision quantization.

In order to tackle the issue of declining accuracy in PTQs, scholars have put forth a range of methodologies. Banner et al. [300] and Finkelstein et al. [301] identified a fundamental bias in the average and variability of the quantified weight values. To address this issue, they put forth a bias-correction approach. Meller et al. [302] and Nagel et al. [303] showed that balancing the weight ranges across the layers or channels could reduce the quantization errors. ACIQ [300] performed an analytical calculation to determine the ideal range for clipping and the appropriate channel bit width settings for PTQ.

While ACIQ did see a decrease in precision, the implementation of channel-wise activation quantization in hardware devices proved to be challenging. In order to tackle this issue, the OMSE approach [304] implemented a solution that removed channel quantization during activation. Additionally, they offered PTQ (Post-Training Quantization) by maximizing the L2 distance between the quantized tensor and its corresponding floating-point tensor. Furthermore, in order to address the negative consequences of outliers more effectively in PTQ, Zhao et al. [305] introduced a technique known as outlier channel-splitting. This method involves duplicating and subsequently halving the channels that include outliers. An additional significant contribution was made by AdaRound [306], which introduced an adaptive rounding technique that demonstrated improved efficacy in minimizing losses. While AdaRound limited the range of variation in quantization weights to ± 1, AdaQuant [307] introduced a more comprehensive technique that enabled the adjustment of quantization weights as required. In the process of Post-training Quantization (PTQ), the weights and activation quantization parameters were established without conducting any retraining of the neural network models. Hence, the employment of PTQ proved to be an

expeditious method for quantifying neural network models. Nevertheless, the PTQ method exhibited a lower level of accuracy compared to the QAT approach.

Low-rank decomposition

The techniques of low-rank decomposition involve utilizing a matrix with a low rank to provide an approximation of the weight matrix within a neural network [308]. Utilizing such low-rank approximation for the weight matrix yields notable efficacy, resulting in a compression ratio of three times on the fully-connected layer. Nevertheless, the acceleration of the model is not substantial as the primary computational activities of the Convolutional Neural Network (CNN) predominantly occur in the convolution layer. Hence, the compression rate is enhanced by decreasing the quantity of convolution layers.

The derivation of the idea of low-rank decomposition was based on the hypothesis that a 3-dimensional (3D) tensor possesses a structural capacity. In the study conducted by [309], the convolution kernel was conceptualized as a three-dimensional tensor, while the fully connected (FC) layer was seen as either a two-dimensional matrix or a three-dimensional tensor. The utilization of low-rank filters was employed as a means to expedite convolutional procedures. The proposal made by [310] introduced the concept of acquiring separable 1D filters through the utilization of a dictionary learning methodology. In their study [311], Denton et al. introduced clustering algorithms that incorporate low-rank decomposition and convolution kernel techniques into basic DNN models. A twofold boost in speed was attained in a solitary convolution layer. Nevertheless, there was a drop in the classification accuracy of 1.00%.

Multiple techniques exist for using low ranks in fully connected (FC) layers (97,101). As an illustration, Denil and colleagues [242] employed a low-rank technique to decrease the quantity of dynamic factors within a deep model. The study conducted by Sainath et al. [312] investigated the application of low-rank matrix factorization to the final weight layer in deep neural networks (DNNs) used in auditory models. In their study, Lu et al. [313] employed a truncated singular value decomposition (SVD) technique to breakdown the fully connected (FC) layers, with the aim of developing compact multi-task deep neural network (DNN) models.

The utilization of the low-rank decomposition method proved to be a straightforward approach in the context of model compression. Nevertheless, the implementation of the low-rank

decomposition method proved to be challenging primarily due to the inherent complexity of the decomposition procedure.

Another issue that arose was the utilization of layer-by-layer low-rank decomposition in recent techniques, which hindered the ability to compress global parameters due to the presence of varying information across different levels. The identification of redundant parameters in deep neural networks (DNNs) was accomplished through the utilization of matrix and tensor decomposition techniques. The tensor representing the filter in a neural network is commonly conceptualized as having four dimensions: width ($\mathcal{W}$), height ($\mathcal{H}$), number of channels ($\mathcal{C}$), and number of convolution kernels ($\mathcal{N}$). The impact of $\mathcal{C}$ and $\mathcal{N}$ on the network architecture is significant, prompting the utilization of low-rank decomposition methods for network compression. These approaches leverage the properties of information redundancy present in the convolution kernel ($\mathcal{W} \times \mathcal{H}$) matrix and its low-rank property.

Due to the predominant concentration of weight vectors within a low-rank subspace, the convolution kernel matrix was recreated using a limited set of basis vectors to mitigate memory constraints. Low-rank decomposition techniques have demonstrated significant advancements in compression and computational efficiency, particularly in the context of large convolutional kernels and networks of small to medium sizes. In recent years, there has been a growing trend in the utilization of 1×1 convolutions in emerging networks. The utilization of low-rank decomposition is not advantageous when employing a 1×1 convolution. Furthermore, the matrix decomposition procedure incurs high computational costs. The decomposition performed layer-by-layer does not facilitate efficient global parameter compression and necessitates extensive retraining to attain convergence. Tackling the mentioned issue, Jaderberg et al. [314] introduced a two-step approach aimed at enhancing the performance of convolution layers in extensive convolutional neural networks. This method relies on tensor decomposition and discriminative fine-tuning [315].

Knowledge distillation

Knowledge distillation [316] is a common method for efficiently passing on information from a more complex model (the instructor) to a simpler one (the student). Knowledge distillation (KD) typically involves the formulation of a loss function aimed at minimizing the discrepancy in output or intermediate features between the student and the teacher. The designs of loss functions

employed in recent studies on distilling NLP models are summarized, as depicted in Figure 27. The approaches can be further classified into logit-based knowledge distillation, feature-based knowledge distillation, knowledge distillation with a dynamic target, and module replacing, depending on the designs of the loss function. Commonly we divide the approaches to KD into two groups: Logit-based KD and Feature-based KD. Those are explained more in-depth in the following.

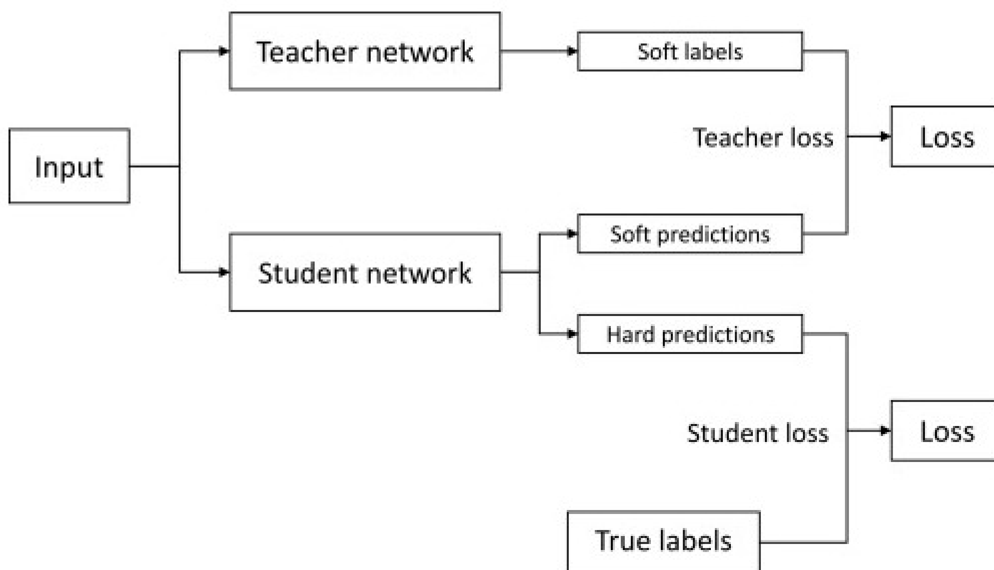


Figure 27. Generalized loss approach design to knowledge distillation

1. Logit-based KD

According to Hinton et al. [316], the utilization of logit-based knowledge distillation (KD) techniques represents the initial endeavors in compressing a big pretrained language model in order to enhance its computational efficiency. Logit-based knowledge distillation (KD) uses the KL divergence or mean squared error (MSE) as optimization objectives to reduce the discrepancy in logits between the student and the teacher. In their study, Tang et al. [317] employed a task-specific approach to distill a BiLSTM model from a fine-tuned BERT model. The BiLSTM model trained with knowledge distillation significantly beats the BiLSTM model trained without knowledge distillation by a substantial margin. To distill BERT model, as proposed by Sanh et al. [318], undergoes a process known as distillation. This process involves pretraining BERT on the specific

objective of masked language modeling (MLM). The loss function comprises three constituent elements: the initial loss of the MLM model, the cosine distance, and the KL divergence. Following the process of distillation, the model has the potential to undergo fine-tuning, hence enabling its performance in various downstream activities. In this study, Turc et al. [319] investigate the impact of initiation on the student. The researchers discovered that employing a student BERT model that was pretrained with Masked Language Modeling (MLM) yielded superior results compared to random initialization and shortened teacher models [318], [320] when utilized for initializing the student model.

2. Feature-based KD

The concept of feature-based knowledge distillation (KD) refers to a method in which a teacher model transfers its knowledge to a student instead of solely relying on the ultimate result, feature-based knowledge distillation (KD) endeavors to synchronize the intermediate characteristics of both the teacher and the learner.

In the distillation process, the initial step involves training the instructor model using the provided training data in order to acquire task-specific features that are pertinent to the given task. The student model is subsequently trained to acquire identical characteristics by decreasing the disparity between the features acquired by the teacher model and those acquired by the student model. The process is commonly executed by employing a loss function that quantifies the disparity between the acquired representations of the teacher and student models, such as the mean squared error or the Kullback-Leibler divergence [321].

One of the primary benefits of feature-based knowledge distillation is in its capacity to facilitate the acquisition of more informative and resilient representations by the student model, surpassing what it might achieve through independent learning. The reason for this is that the teacher model has already acquired the most pertinent and enlightening characteristics from the material, which may be imparted to the student model via the process of distillation. In addition, the utilization of feature-based knowledge distillation exhibits a broad applicability across various tasks and models, hence demonstrating its versatility as a technique.

Nevertheless, feature-based knowledge distillation does have its inherent limitations. The computational cost of this technique may be higher compared to other forms of knowledge

distillation due to the necessity of extracting the internal representations from the instructor model throughout each iteration. Moreover, it is important to consider that a feature-based approach may not be appropriate for tasks in which the internal representations of the instructor model are not transferable or pertinent to the student model.

The authors of the study conducted by Sun et al. [320] propose the utilization of a mean squared error (MSE) loss function to measure the discrepancy between layer representations. A comparable methodology is also expounded upon in the work of Aguilar et al. [322]. Building upon the concept of Progressive Knowledge Distillation (PKD), Jiao et al. [323] proposed the integration of an attention loss mechanism in TinyBERT. This attention loss mechanism is designed to align the attention matrices at different levels between the instructor and the student models. The findings of TinyBERT further support the notion that including knowledge distillation (KD) in both the pretraining and finetuning phases might lead to enhanced performance.

In a similar vein, the attention matrix and value-value scaled dot-product (referred to as value relation loss) are aligned in MiniLM [324], [325]. The additional functionality serves as a supplement to the attention matrix, specifically the queries-keys scaled dot-product, enabling the seamless transfer of multi-head self-attention. In their recent study, Wu, Wu, and Huang [326] put up a novel approach for distilling a student model by using intermediate features and soft labels from different professors. This framework aims to enhance the performance of the student model. The DynaBERT model, as proposed by Hou et al. in [327], employs a layer-wise knowledge distillation (KD) loss mechanism to transfer knowledge from a teacher model to a student model. The student model is designed with sub-networks of varying widths and depths. Hence, the identical model can be applied across diverse devices that possess varying computational resources.

The MobileBERT model, as proposed by Sun et al. [328], presents a modified BERT architecture specifically tailored for utilization on mobile devices. In addition to the methodologies proposed by Sun et al. [80] for layer-wise feature distillation and Jiao et al. [323] for attention distillation, the authors of this study provide a progressive knowledge transfer mechanism that involves distilling the model layer by layer, as opposed to distilling all layers simultaneously. Liu et al. [329] utilize structural characteristics at both the token and span levels to establish alignment between the student and the teacher.

Lightweight model design

The concept of lightweight deep neural network (DNN) model design involves modifying the existing DNN structure to decrease the number of parameters and computational complexity. Table 4 presents an overview of the design skills pertaining to the lightweight model. In their study, Iandola et al. [330] introduced SqueezeNet as a novel approach that involved substituting 3 × 3 convolution kernels with 1 × 1 convolution kernels. The number of parameters in a 1 x 1 convolution kernel is one-ninth of the number of parameters in a 3 × 3 convolution kernel. Nevertheless, the utilization of this approach resulted in a reduction in the quantity of input channels accessible in comparison to the 3 × 3 convolution technique.

Through the acquisition of knowledge pertaining to ResNet and the subsequent integration of bypass branches into the initial network architecture, a discernible enhancement in classification accuracy of roughly 3% was seen. The MobileNet architecture, as presented by Howard et al. [260], introduces a novel approach to convolution by separating it into two distinct operations: depth-wise convolution and point-wise convolution. In the context of depth-wise convolution, each individual convolution kernel filter exclusively conducts convolutional operations on a singular input channel. The technique of point-wise convolution involves utilizing a convolution kernel with a size of 1 × 1 to merge the outputs of the depth-wise convolution layer, which consist of many channels. In their study, Zhang et al. [331] introduced ShuffleNet, a novel approach that incorporates channel shuffling to enhance the learning capacity of the model. By rearranging the input groups into channels, ShuffleNet ensures that the receptive fields of each convolutional kernel are distributed over multiple input groups, hence facilitating improved learning capabilities.

In their study, Gao et al. [331] made enhancements to the efficacy of lightweight models within the context of self-supervised learning. In their study, Tan et al. [332] introduced MnasNet, a method for neural architecture search (NAS). The integration of the model's time consumption on the device was achieved by means of multi-objective optimization within the search space. Subsequently, the utilization of a deconstructed hierarchical search space enabled the network to preserve layer diversity while simultaneously simplifying the search space. This facilitated a more optimal balance between precision and efficiency in the search algorithm. Huang and colleagues [333] postulated that the concept of group convolution may be effectively learned. The learning group convolution technique was further developed by integrating the training process with

pruning to enhance the accuracy of the pruning procedure. Mehta and colleagues [334], [335] introduced an end-to-end speech processing network (ESPNet) as a lightweight network designed for semantic segmentation. The key component of ESPNet was an ESP module.

The ESP module included point-wise convolution and a spatial pyramid of dilated convolution, demonstrating superior efficiency compared to MobileNet and ShuffleNet. The utilization of depth-wise separable convolution resulted in a reduction in both the computational time and the number of parameters within the network. Conversely, point-wise convolution exhibited the highest number of parameters. In response to this, Gao et al. [336] introduced a convolution technique that separates channels and depths. The ChannelNet architecture was developed by substituting the fully connected (FC) layer and the global pooling layer of the network. The utilization of group convolution in the interleaved group convolution (IGC) series network was found to be quite pronounced, as evidenced by previous studies [337]–[339].

The IGC approach involves decomposing the conventional convolution operation into numerous group convolutions, resulting in a reduction in the overall number of parameters. Moreover, the idea of complementarity and the sorting operation facilitated the exchange of information among groups while minimizing the number of parameters involved. The FBNet series, specifically FBNet-143 to FBNet-145, represents a collection of lightweight network architectures that were developed exclusively using the Neural Architecture Search (NAS) methodology. The FBNet architecture [340] integrates the principles of DNAS (Differentiable Neural Architecture Search) with the consideration of resource restrictions. FBNetV2 [342] has incorporated a search feature for channel and input resolution. The FBNetV3 model [342] employed accuracy prediction as a means to conduct an efficient exploration of network architectures. At present, efforts are being made to optimize the performance of deep neural networks (DNNs) to enhance performance in the following three domains: enhance the network's width, augment the network's depth, and enhance the resolution of input photos.

Enhancing the precision of a network can be readily achieved by adjusting a single dimension. Nevertheless, the process of simultaneously modifying two or three aspects of the network necessitates laborious human adjustment and poses challenges in terms of optimization. To tackle these issues, Tan et al. [343] introduced a hybrid scaling approach for model scaling. This method offers improved selection of breadth, depth, and resolution dimensional scaling, hence facilitating

the attainment of enhanced accuracy in the model. In their study, Han et al. [344] introduced a Ghost module as a means to extract a greater number of features while utilizing a reduced number of parameters. Initially, the Ghost module employed the output that entailed the least number of raw convolution processes. Subsequently, a sequence of elementary linear operations was applied to the resulting data to develop additional characteristics. The GhostNet architecture was introduced as a modification of the Ghost module, wherein the conventional convolution layer is substituted with the Ghost module. The empirical findings demonstrated that GhostNet exhibited favorable compression capabilities while still preserving high levels of accuracy.

In their study, Ma et al. [345] introduced a novel dynamic generative network called WeightNet. This network effectively combined the characteristics of two existing models, namely squeeze and excitation networks (SENet) and CondConv, within the weight space. WeightNet is a convolutional neural network that employs a dynamic approach to build kernel weights by considering sample characteristics. Additionally, it optimizes the hyperparameters to strike a balance between accuracy and computational efficiency.

In their study, Li et al. [346] introduced MicroNet, a novel architecture that incorporates micro-factorized convolution with dynamic shift-max. The utilization of micro-factorized convolution in this study ensured the preservation of input-output connectivity while simultaneously reducing the overall number of connections by employing low-rank decomposition. Self-supervised representation learning (SSL) has garnered considerable interest in recent times. Nevertheless, subsequent research has determined that there is a significant decline in performance as the size of the model is reduced. The authors Gao et al. [331] introduced a novel approach termed distillation contrast learning (DisCo) as a potential solution to address the limitations of existing SSL approaches that mainly depend on contrast learning for network training. The final embedded limitations of lightweight pupils were aligned with those of teachers by DisCo, aiming to optimize the transfer of teacher knowledge.

Proposed solutions

The process of training and testing deep neural network models can impose significant computational demands. The particular software and hardware needs are contingent upon the intricacy of the models, magnitude of the datasets, and the financial resources at our disposal.

Although not obligatory, possessing a GPU (Graphics Processing Unit) is strongly advised for the purpose of training our deep neural networks. Deep learning frameworks have the capability to utilize graphics processing units (GPUs) in order to achieve notably accelerated training speeds. NVIDIA graphics processing units (GPUs) are widely used and enjoy extensive compatibility with a majority of frameworks.

In order to enable GPU acceleration, it is necessary to install the CUDA (Compute Unified Device Architecture) and cuDNN (CUDA Deep Neural Network) libraries if an NVIDIA GPU is being used. These factors are crucial in facilitating accelerated training. Data manipulation libraries are software tools that are used to perform various operations on data. These libraries provide a set of functions and methods that enable users to manipulate, transform, and Libraries such as NumPy, pandas, and scikit-learn are widely employed in the field of data science for the purpose of data manipulation, preprocessing, and analysis.

Depending on the specific requirements, it may be advantageous to utilize supplementary tools such as Tensor Board for data visualization, Jupyter Notebooks for interactive programming, and scikit-learn for machine learning applications in conjunction with deep learning methodologies.

For the hardware aspect of model compression, a contemporary multi-core central processing unit (CPU) is necessary, although the majority of the computational workload during the training process will be delegated to the graphics processing unit (GPU). It is strongly advised to utilize a specialized graphics processing unit (GPU) for the purpose of training deep neural networks. NVIDIA graphics processing units (GPUs), namely those belonging to the GeForce and Quadro series, are extensively employed in the realm of deep learning applications. Deep learning work often utilizes high-end graphics processing units (GPUs) such as the NVIDIA GeForce RTX series because of their widespread popularity.

The utilization of memory in deep learning models can be substantial. It is advisable to possess a minimum of 16 gigabytes of random-access memory (RAM), while a greater amount is preferable when dealing with larger models and datasets.

Sufficient storage capacity is essential for accommodating datasets, model checkpoints, and experiment logs. The utilization of a Solid-State Drive (SSD) is recommended in order to

enhance the speed of data loading and model saving processes. It is imperative to consider the power supply unit (PSU) capacity in relation to high-end GPUs, since they have a substantial power consumption. Therefore, it is crucial to ascertain that the PSU is capable of accommodating the power demands imposed by these GPUs.

Deep learning tasks have the potential to create substantial thermal energy. Sufficient cooling, encompassing effective airflow and appropriate cooling mechanisms, is important in order to avert the occurrence of overheating. Cloud services such as AWS, Google Cloud, and Azure provide GPU instances for deep learning, which can be particularly beneficial in cases where individuals lack access to high-performance local machines. This approach offers a cost-effective means of obtaining the necessary hardware. The hardware and software requirements may exhibit variability contingent upon the magnitude and intricacy of the deep learning endeavor. It is advisable to consult the specific guidelines provided by the selected deep learning framework and GPU manufacturer in order to determine the most appropriate combinations.

Experimental Results

1. Model Pruning Compression Technique

Pruning is a model compression approach wherein certain neurons or connections are eliminated from a deep neural network model in order to decrease its dimensions and computing demands. The utilization of pruning techniques on a deep neural network model can yield several benefits as well as potential limitations.

There are several benefits associated with the practice of pruning. The process of pruning results in a substantial reduction in the size of the model, hence resulting in smaller files and a decrease in memory usage. The adoption of this technology on devices with limited resources is advantageous. The utilization of smaller models often leads to accelerated inference times, rendering them well-suited for real-time applications and edge devices. Pruned models exhibit a decreased memory footprint throughout the inference process, hence offering a notable advantage for devices that possess constrained random-access memory (RAM) capacity.

The utilization of pruned models in the context of energy efficiency during inference is of significant importance, particularly in the case of battery-powered devices. The utilization of pruned models enhances scalability, enabling their deployment across a broader spectrum of

hardware configurations. One advantage of smaller models is the potential for cost savings due to reduced storage space requirements. This can result in lower expenses associated with hosting and distributing the models. The transfer speed can be enhanced by lowering the size of model files, resulting in less latency during internet transmission.

The process of pruning has the potential to result in a decrease in the accuracy of a model, particularly if executed without due caution. The removal of critical neurons or connections has the potential to detrimentally impact the performance of the model. Fine-tuning is a commonly employed strategy to address the issue of accuracy loss in pruned models. Additionally, strategies such as repeated pruning are sometimes utilized to further offset this loss. However, it is important to note that these approaches typically result in an increase in training time.

The concept of complexity refers to the level of intricacy or difficulty in a particular system or the process of determining which neurons or connections to prune and the degree to which they should be pruned can be a multifaceted undertaking. Additional hyperparameter adjustment may be necessary. The efficacy of pruning is contingent upon the particular task and architecture at hand. Certain jobs may exhibit a higher degree of tolerance towards pruning compared to others.

The concept of balance refers to the state of equilibrium or stability achieved when opposing forces or elements Trade-off: A trade-off exists between the reduction in model size and the level of accuracy achieved. Achieving an optimal equilibrium is of utmost importance. Certain pruning strategies may not exhibit direct compatibility with all deep learning systems, necessitating the need for special code during implementation.

In brief, the utilization of pruning as a strategy holds significant potential in diminishing the dimensions and processing demands of deep neural network models, hence rendering them more appropriate for implementation on edge devices and real-time applications. Nevertheless, achieving an optimal equilibrium between reducing model size and preserving satisfactory model performance necessitates meticulous deliberation and empirical investigation. Moreover, the specific outcomes will be contingent upon the model, task, and the pruning methodologies employed. Below Figure 28 represents the result of pruning compression techniques that applied to our trained resnet18 model on the Tomato Plant Diseases dataset.

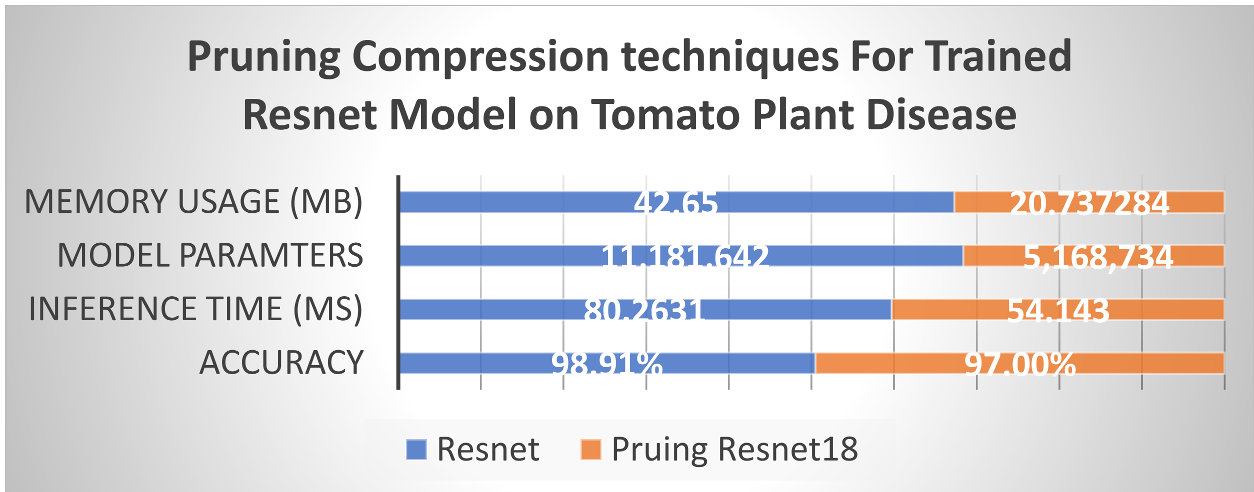


Figure 28. Pruning compression techniques for trained resnet18 Model on Tomato Plant Diseases

We can observe from the above-mentioned figure 28 that both measured properties of the model (parameter size and number of model parameters) are significantly reduced. Also, the inference time is significantly reduced from 80.26 MS to 54.143 MS. Regarding the downside of this compression approach, there is a noteworthy predictive performance penalty resulting in the predictive performance (accuracy) decreasing from 98.91% to 97.00%. Therefore, as mentioned before, we have to accept a tradeoff between the compressed model properties and reduced predictive performance.

The figure 29 below represents the results of applied pruning compression techniques on Googlenet model trained on the Tomato Plant Disease dataset. Observing the results, we can also detect the significantly reduced model size, model parameters as well as the inference time. On the other hand, there is still some predictive performance penalty but not nearly as significant as can be observed in previous figure on the resnet18 model. In this case, the predictive accuracy dropped only by 5% which can be acceptable while reducing the model by significant amount as well as reducing the inference time, which can be crucial in the domain specific applications.

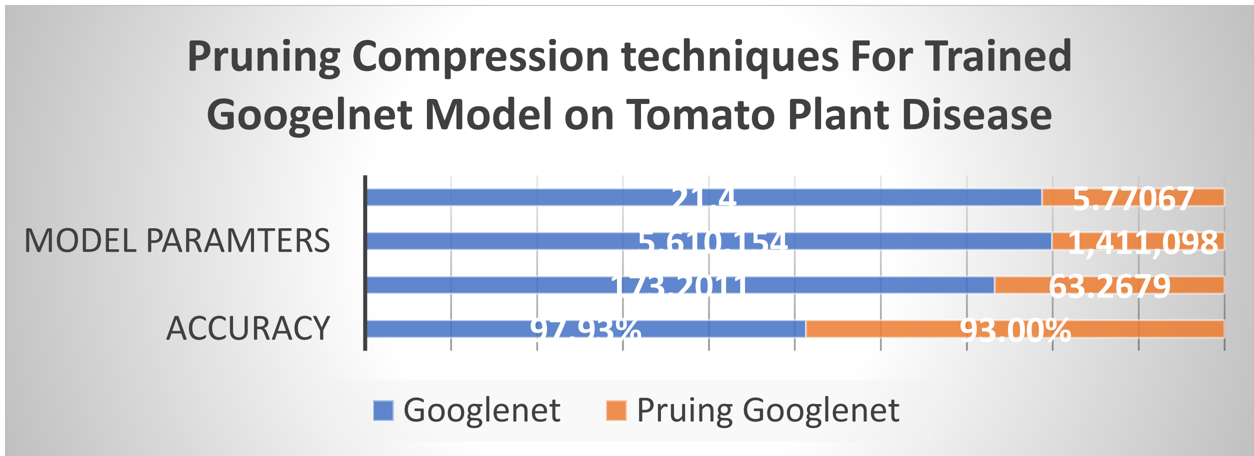


Figure 29. Pruning Compression techniques for trained Googlenet Model on Tomato Plant Diseases

Presented in figure 30 below, we can show the results of the applied pruning compression techniques to the Mobilenet model trained on our Tomato Plant Diseases dataset. Similar to the results of the resnet18 model, we can observe a significant drop of model parameter size as well as the number of model parameters. There is also a decrease of the inference time by 10 MS but on the other hand there is also no difference in the predictive accuracy of the Mobilenet model from 98% to 98% providing us a good productive performance.

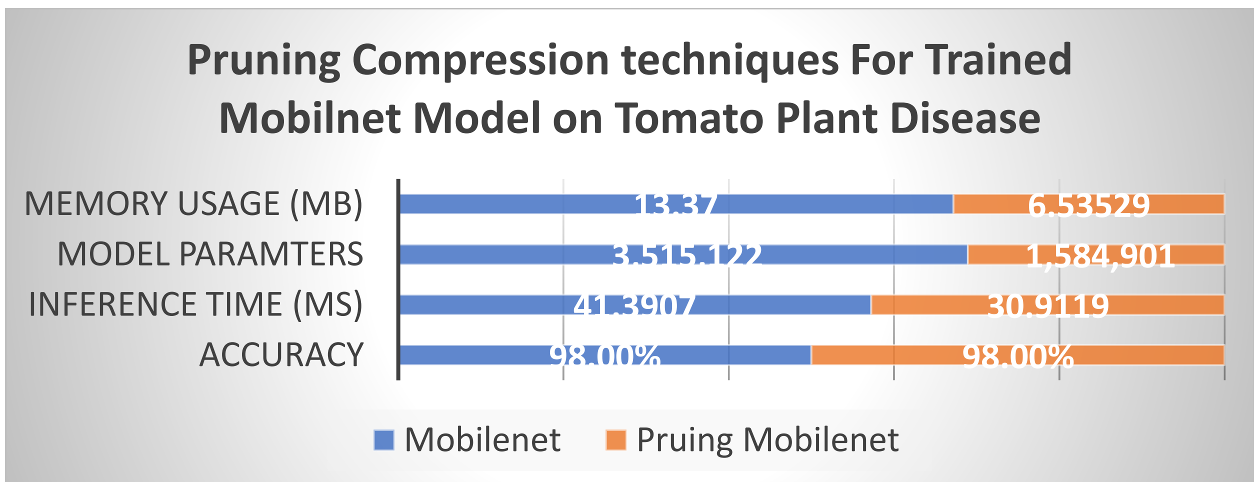


Figure 30. Pruning Compression techniques for trained Mobilenet Model on Tomato Plant Diseases

Based on the presented results of our experimental pruning techniques applied to different model architectures, we can conclude that the Googlenet model is more prone to the model pruning technique than the resnet18 and the Mobilenet model, since it has the least predictive performance

decrease than the other two models. Therefore, Googlenet is the most appropriate model to choose when we are going for the utilization of pruning model compression techniques.

2. Knowledge Distillation Compression Technique

The process of knowledge distillation involves training a smaller model, referred to as the student, to emulate the behavior and performance of a bigger model, known as the teacher. When the technique of knowledge distillation is employed in the context of a deep neural network model, numerous significant outcomes and benefits can be discerned. The size of the student model is generally smaller compared to that of the teacher model, resulting in decreased memory and storage demands. The adoption of this technology on devices with limited resources is advantageous.

The speed of inference is influenced by the size of the model, with smaller models generally exhibiting faster inference times. This characteristic renders them well-suited for real-time applications and edge devices. The utilization of knowledge distillation has been observed to enhance the generalization capabilities of the student model, surpassing those achieved through training from scratch. The transfer of valuable information from the instructor model to the learner is facilitated through the distillation of knowledge.

Another technique that can be employed to regulate the level of confidence in the predictions made by the student model is temperature scaling. This feature enables the user to optimize the trade-off between the accuracy of the model and its ability to generalize to unseen data. The regularization effect of knowledge distillation is observed as it serves as a mechanism to regularize the student model, hence mitigating the risk of overfitting.

The technique of knowledge distillation can also be employed to extract knowledge from a collection of instructor models, thereby inducing an ensemble effect within the student model. This process has the potential to enhance the student model's resilience and precision. The transferability of the knowledge encapsulated in the student model can be enhanced across other domains or activities, as the student is acquiring knowledge from the experiential expertise of the teacher.

Compression and distribution of student models can be facilitated by reducing their size, resulting in potential benefits such as decreased bandwidth usage and storage expenses during

model deployment. Nevertheless, it is crucial to acknowledge that there are certain potential trade-offs.

There is generally a trade-off between the accuracy of a model and its level of compression. Although knowledge distillation has the potential to enhance generalization, it is important to note that the student model may not attain the same level of performance as the teacher model. The incorporation of knowledge distillation in the training process introduces an additional layer of complexity since it necessitates the training of both a teacher model and a student model. It is imperative to engage in meticulous hyperparameter tweaking and demonstrate a keen focus on details.

The process of selecting a teacher model. The selection of the teacher model holds significant importance, as the efficacy of the distilled information is contingent upon the instructor's performance and architectural attributes. In brief, information distillation is an advantageous methodology for diminishing model dimensions, enhancing generalization, and generating more compact and efficient models. The utilization of models on edge devices or in situations with restricted computational resources can be particularly advantageous. Nevertheless, achieving the ideal balance between accuracy and compression necessitates meticulous implementation and the careful selection of suitable hyperparameters.

Figure 31 represents the results of the applied knowledge distillation compression techniques for Resnet18 model trained on the Tomato Plant Diseases in the experimental phase of our work. As we can observe from figure 31 there is a small amount of reduced parameter size and number of parameters, however the interference time is significantly reduced from 20.66 MS to 19.91 MS. Regarding accuracy, we can observe a significant reduction in predictive performance where accuracy has dropped from 98.91% to 93.67, which translates to the margin of 5.24% decrease.

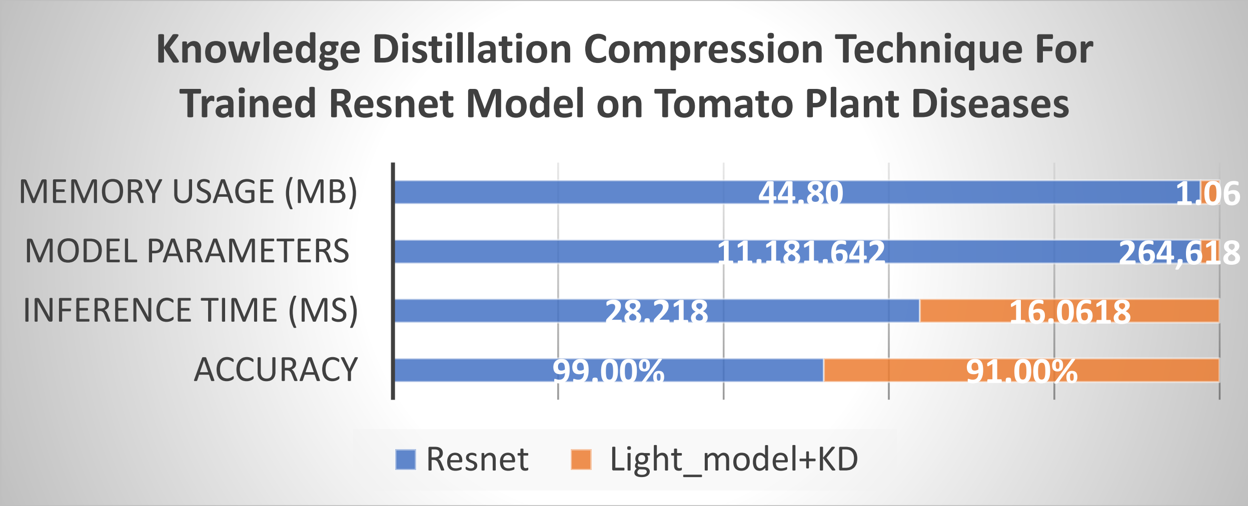


Figure 31. Knowledge Distillation compression techniques for trained Resnet18 Model on Tomato Plant Diseases

Figure 32 illustrates the outcomes of employing knowledge distillation compression methods on the Googlenet model that was trained using the Tomato Plant Diseases dataset. Similar to the findings in the Resnet18 experiment, it is evident that there is a smaller reduction in the size of the parameters, as well as a decrease in the number of model parameters. In this scenario, the inference time is reduced in a comparable manner to that observed in the Resnet18 model. In comparison to the Resnet18 model, the drop in classification performance seen in this scenario is rather minor, with a difference of only 2.1%.

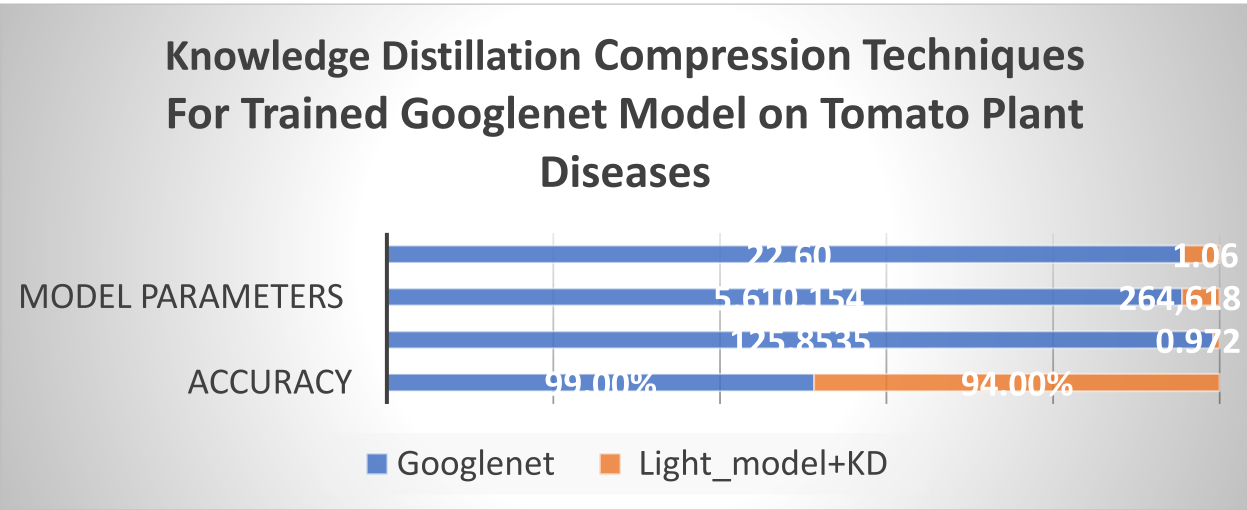


Figure 32. Knowledge Distillation compression techniques for trained Googlenet Model on Tomato Plant Diseases

Represented in Figure 33 are the results of applied knowledge distillation techniques for trained Mobilenet model against the Tomato Plant Diseases dataset. Similar to the previously presented experimental results, here also we can observe the same behavior in terms of reduced parameter size, number of parameters, and decreased inference time. Similar to the Googlenet model, the Mobilenet model also experiences a quite smaller performance accuracy reduction. The accuracy in this case was only decreased by 1.45%, which is the least of all of the evaluated model architectures.

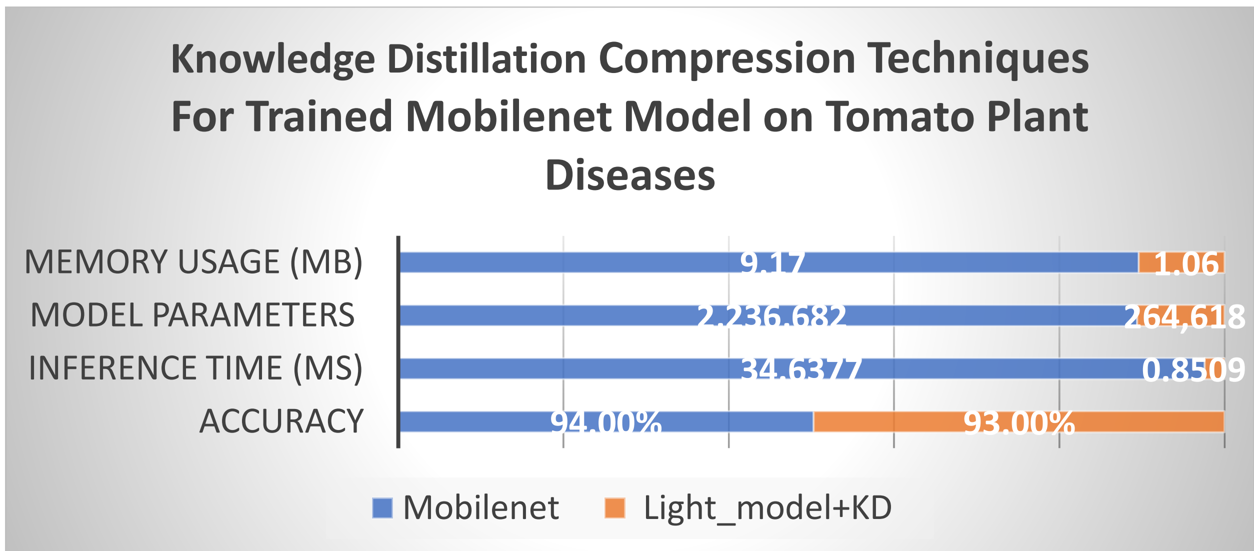


Figure 33. Knowledge Distillation compression techniques for trained Mobilenet Model on Tomato Plant Diseases

Examining results of the three presented architectures, the one with the least downsides when it comes to the applying the knowledge distillation techniques turns out to be the Mobilenet, which while reducing the parameter size, number of parameters as well as the inference time it manages to achieve classification accuracy with least accuracy reduction. Therefore, when applying the knowledge distillation technique, the most suitable of the compared models would be a Mobilenet architecture.

3. Parameter Quantization Compression Technique

Quantization is a model compression technique that entails the reduction of precision in the weights and/or activations within a deep neural network model. When implementing quantization techniques on a deep neural network model, several outcomes and impacts might be noticed.

The advantages of quantization often include a reduction in model size, speedier interference, scalability, cost savings, and compatibility. However, there are also certain negatives associated with quantization, such as a reduction in accuracy, increased complexity, and dependence on the specific task at hand.

The process of quantization is employed to decrease the memory usage of the model by utilizing lower bit precision to represent weights and activations. This is particularly important when deploying the model on devices with limited resources. The utilization of lower-precision computations can lead to enhanced inference speed, rendering quantized models well-suited for real-time applications and edge devices. Energy efficiency is enhanced by reduced precision operations, as they generally consume less power. This, in turn, contributes to more efficient inference processes. This holds particular significance in the context of gadgets that rely on battery power.

The deployment of quantized models allows for broader hardware configuration compatibility, hence facilitating scaled deployment. Cost savings can be achieved by using smaller model sizes, which can effectively minimize the expenses associated with storing and distributing models. The expedited transmission of smaller model files via the internet results in decreased latency during the process of model downloads. In general, quantization is widely compatible with most deep learning frameworks, as several frameworks offer dedicated tools for facilitating the process of quantization.

The utilization of quantization techniques, particularly with low bit precision, has the potential to diminish the correctness of a model. Achieving a balance between accuracy and the level of quantization is of utmost importance. Fine-tuning, also known as retraining, is a crucial step following the quantization process to potentially restore the accuracy that was diminished. The process of determining the appropriate quantization level and bit accuracy for weights and activations is intricate and necessitates empirical investigation. The impact of quantization can differ according to the particular task, dataset, and architecture involved.

Quantization-Aware Training (QAT) involves the utilization of certain training methods that consider the eventual application of quantization techniques. By incorporating this knowledge throughout the training process, the model might potentially get benefits from the subsequent quantization.

Quantization seems to be a helpful strategy in the context of reducing the size of models, raising the speed of inference, and improving energy economy during the deployment of deep neural network models. Nevertheless, achieving an optimal equilibrium between reducing model size and preserving satisfactory model performance necessitates meticulous deliberation and empirical investigation. The outcome will be contingent upon the specific model, task, and quantization techniques employed.

Figure 34 represents the results of our quantization compression techniques which we employed against the Resnet18 model trained on the Tomato Plant Diseases dataset. As we can observe from Figure 34, the parameter size and number of model parameters are significantly decreased as well as the inference time from 20.663 MS to 18.2 MS. We can also observe a significant reduction of predictive performance of trained model. From an initial 98.9% it was reduced to 91%, which is a 7.9% decrease. Utilization of such quantization approach to Resnet18 model architecture would be suitable only on edge devices with some strong hardware constraints which would make the reduction of accuracy acceptable.

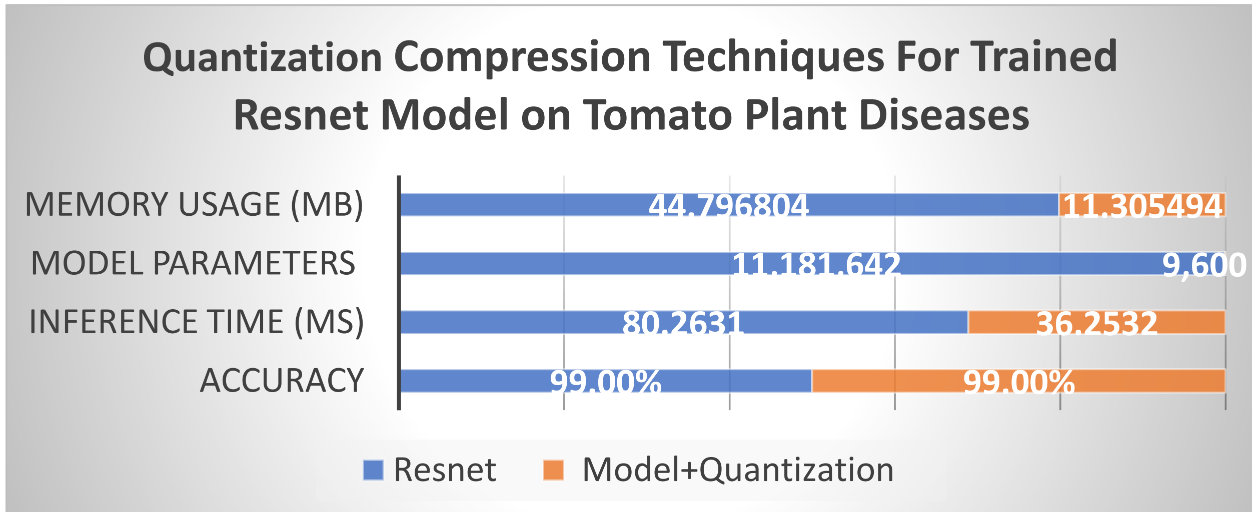


Figure 34. Quantization compression techniques for trained Resnet18 Model on Tomato Plant Diseases

The results of the applied quantization techniques on the Googlenet model trained against the Tomato Plant Disease dataset are presented in Figure 35. Similar to the previous results on the Resnet18 model, also here on the Googlenet model we can observe that the number of parameters and parameter size are significantly decreased: The parameter size is reduced from 21.4 MB to

6.03 MB, which would enable us to deploy such model in really hardware constrained edge devices. Also, the inference time was decreased but not as much as in Resnet18 experiment. The reduction of the predictive accuracy is also significantly noticeable in this experiment. The accuracy after the employment of the quantization techniques dropped by 4.93 % from initial 97,93%, which is significant but not as large as we observed in the Resnet18 model.

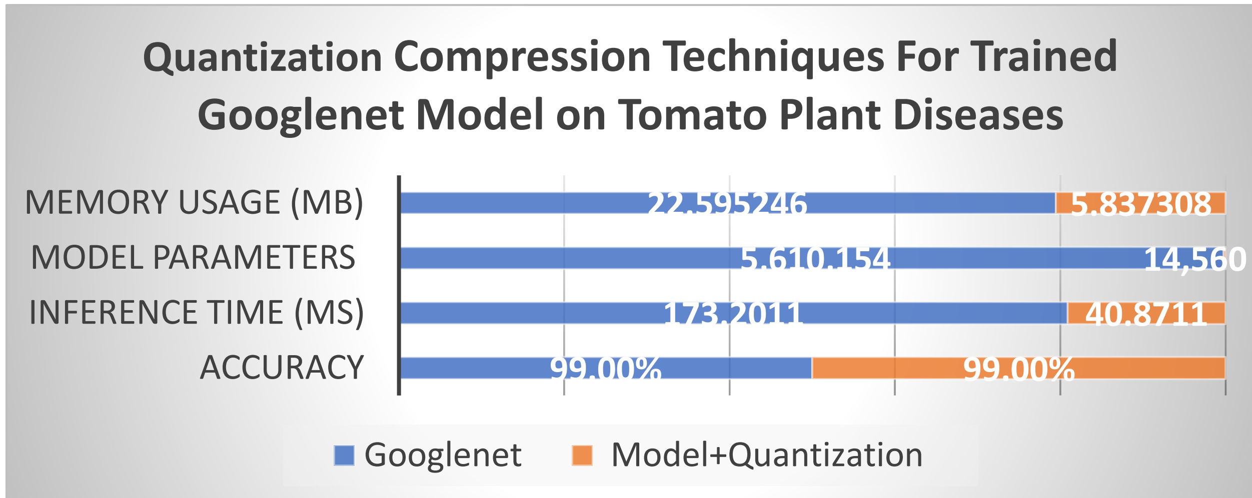


Figure 35. Quantization compression techniques for trained Googlenet Model on Tomato Plant Diseases

Lastly, Figure 36 below represents the results of quantization compression techniques applied to the Mobilenet model trained on the Tomato Plant Diseases dataset. The results of this experiment are quite similar to the previously presented ones utilizing Resnet18 and Googlenet predictive models. From Figure 36, we can observe a significant reduction of parameter size as well as in number of parameters. Regarding the inference time, it is reduced from 22.55 MS to 20.5 MS, which means that the reduced margin is more or less on par with other analyzed model architectures. Similar to the previous two experiments, the decrease in model's predictive performance is also significant. We can observe the performance drop of 5.12%, which is more than we experienced utilizing Googlenet and less than in Resnet18.

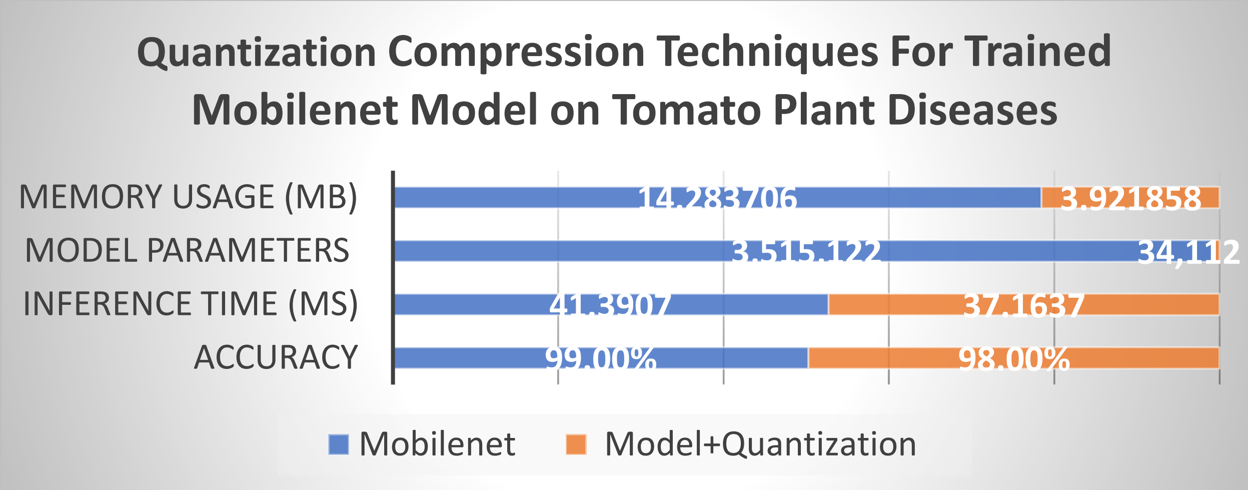


Figure 36. Quantization compression techniques for trained Mobilenet Model on Tomato Plant Diseases

Based on the findings from our experimental analysis, which involved the application of quantization techniques to various model architectures, it can be inferred that the Googlenet model exhibits a higher susceptibility to model quantization techniques compared to the Resnet18 and Mobilenet models. This is evident as the Googlenet model demonstrates a relatively smaller decrease in predictive performance compared to the other two models, while simultaneously achieving substantial reductions in the number of parameters, parameter size, and inference time. Hence, the Googlenet model is seen as the most suitable choice for implementing pruning model compression strategies.

Discussion

In the preceding part, we provided comprehensive findings from our performed experiments, wherein various model compression strategies like as pruning, knowledge distillation, and quantization were applied to three distinct model architectures: Resnet18, Googlenet, and Mobilenet.

Upon analysis of the data, it is evident that the Googlenet model architecture exhibits the highest level of suitability among the compared architectures for the application of diverse compression algorithms. The Googlenet predictive model demonstrated the highest classification accuracy among the applied model compression techniques in two out of three experiments and achieved the second-best score in the remaining experiment. This indicates that the Googlenet model is the most optimal choice among the compared model architectures. Based on the experiments we ran,

the Mobilenet architecture emerged as the second-best performing model, while the Resnet18 architecture performed the least effectively. Naturally, it is imperative to acknowledge and account for the constraints inherent in our experimental study. The experiments in this instance were only conducted on a single dataset, hence limiting the ability to extrapolate the findings of the study. Furthermore, the evaluation of model compression strategies has been limited to only three distinct model designs, hence constraining the ability to derive overarching conclusions.

Chapter Summary

Model compression strategies refer to a set of methodologies employed to decrease the dimensions and processing demands of deep neural network models, while simultaneously maintaining or decreasing their performance. These strategies yield multiple notable effects and benefits such as, the size of deep learning models can be greatly reduced through the implementation of model compression techniques, including quantization, pruning, and knowledge distillation. This aspect holds significant importance when it comes to the implementation of models on devices with limited resources or for expediting the transfer of models over the internet. In addition, the utilization of smaller models frequently leads to expedited inference times. Low latency is a crucial need in real-time applications, mobile devices, and edge computing.

Compressed models exhibit a reduced memory footprint, hence rendering them compatible with devices possessing constrained RAM capacities, such as smartphones and embedded systems. The process of model compression has the potential to enhance the energy efficiency of inference, a critical factor for devices reliant on battery power and applications like mobile and Internet of Things (IoT). The utilization of compressed models facilitates the implementation of deep learning solutions on a broader spectrum of hardware, hence enabling efficient and scalable deployment. The generalization performance of a smaller student model can be improved through the utilization of compression techniques, such as knowledge distillation, which involves the transfer of knowledge from a bigger instructor model.

One potential benefit of utilizing smaller models is the potential for savings on storage costs. This is due to the fact that smaller models require less storage space, resulting in a reduction in expenses associated with hosting and distributing these models. The acceleration of training can

be observed in certain instances through model compression, as the utilization of smaller models necessitates fewer computational operations throughout the training procedure. The utilization of compressed models is advantageous in the context of federated learning, a paradigm in which models are trained on distributed data sources. Compressed models effectively mitigate the burdens associated with communication and computing, making them highly suitable for this decentralized learning approach. The implementation of model compression approaches plays a crucial role in facilitating the deployment of deep learning models on edge devices. These techniques enable the utilization of on-device artificial intelligence (AI) capabilities, eliminating the need for dependence on cloud services. It is imperative to acknowledge that the precise impacts of model compression may differ based on the employed approaches, the nature of the model, and the given task. Although model compression presents various advantages, it entails a compromise with model performance. The degree to which a model may be compressed while preserving satisfactory performance relies on the particular use case and the effectiveness of the compression techniques employed.

CHAPTER 6

CONCLUSION

While highly desirable, latency prediction is also expensive. Due to the varied models inference latency brought on by the runtime optimizations on different edge devices, it is highly difficult and current techniques are unable to attain a high level of prediction accuracy. This chapter reviews the possible challenges involved in the prediction of latency. Further, it comprehensively discusses the various DNN models that don't only reduce the inference time but also predict it accurately. It is found from this chapter that inference time is dependent upon the computational complexity, data size, and features. Accuracy is reciprocal of inference time. Computational complex models and large number of features reduce the inference time. Also, here we discussed here the strengths and weaknesses of various deep neural network models for latency prediction, various real-world application domains where latency played a crucial role, and the limitations and challenges of current methods for latency predicting in Deep Neural Networks (DNNs) in real-world applications. The resolution of these limitations would be the possible future directions in latency prediction. We demonstrated that we created a deep learning model that can outperform the classical approaches by predicting inference time. By summing the inference time layer by layer for deep learning architectures and think of each layer as the atomic operation of a model. This approach can be generalized more to include more hardware and software features. This will save lots of time and money and will increase the time to market many products that are using deep learning for any task.

In a few years, we will have billions of connected devices installed in homes, cities, cars, and businesses around the world. User and device environments interact with resource-constrained devices. To interpret the behavior in sensor data, make accurate predictions, and make judgments, many of these devices rely on machine learning models. The number of connected devices that can block the network will become a bottleneck. Therefore, machine learning techniques are needed to integrate intelligence into end devices. Using machine learning on these

edge devices can reduce network congestion by being able to perform computations close to the data source. In addition, servers and other powerful computers have traditionally been the only platforms used for machine learning. But with advances in chip technology, we now have portable libraries that can fit in our pockets. Therefore, due to the current advancements in processing power, energy storage, and storage capacity of these devices, the opportunity to gain significant benefits from machine learning on Internet of Things (IoT) devices has emerged. Implementing machine learning inference on edge devices holds great potential but is still in its infancy. With the rapid development of IoT and AI, research efforts to fully realize Edge ML will increase in the coming decades. This study provides a detailed assessment of the past, present, and future of the Edge ML literature. Additionally, caching difficulties and best practices for Edge ML training are clearly discussed.

Additionally, Edge ML computational latency for designing realistic and responsive applications is studied. In this chapter, an innovative method for dynamically choosing the best deep learning model between three CNN models for an IoT device is provided. In terms of accuracy, inference time, and energy usage, our method offers a significant advantage over individual deep learning models. Our strategy relies on designing a Machine learning based system to classify tomato plant leaf diseases running with higher speed and lower power consumption; and depending on the model input and the required level of precision that is running locally on IoT device. A number of input features that are tuned and chosen by our automated approach form the basis of the prediction. Using the Plant Village validation dataset, we applied our method to the tomato image classification task and assessed it on the Nvidia system administration interface referred to as (NVSMI).

Model compression strategies refer to a set of methodologies employed to decrease the dimensions and processing demands of deep neural network models, while simultaneously maintaining or decreasing their performance. These strategies yield multiple notable effects and benefits such as, the size of deep learning models can be greatly reduced through the implementation of model compression techniques, including quantization, pruning, and knowledge distillation. This aspect holds significant importance when it comes to the implementation of models on devices with limited resources or for expediting the transfer of models over the internet. In addition, the utilization of smaller models frequently leads to expedited

inference times. Low latency is a crucial need in real-time applications, mobile devices, and edge computing.

Compressed models exhibit a reduced memory footprint, hence rendering them compatible with devices possessing constrained RAM capacities, such as smartphones and embedded systems. The process of model compression has the potential to enhance the energy efficiency of inference, a critical factor for devices reliant on battery power and applications like mobile and Internet of Things (IoT). The utilization of compressed models facilitates the implementation of deep learning solutions on a broader spectrum of hardware, hence enabling efficient and scalable deployment. The generalization performance of a smaller student model can be improved through the utilization of compression techniques, such as knowledge distillation, which involves the transfer of knowledge from a bigger instructor model.

One potential benefit of utilizing smaller models is the potential for savings on storage costs. This is due to the fact that smaller models require less storage space, resulting in a reduction in expenses associated with hosting and distributing these models. The acceleration of training can be observed in certain instances through model compression, as the utilization of smaller models necessitates fewer computational operations throughout the training procedure. The utilization of compressed models is advantageous in the context of federated learning, a paradigm in which models are trained on distributed data sources. Compressed models effectively mitigate the burdens associated with communication and computing, making them highly suitable for this decentralized learning approach. The implementation of model compression approaches plays a crucial role in facilitating the deployment of deep learning models on edge devices. These techniques enable the utilization of on-device artificial intelligence (AI) capabilities, eliminating the need for dependence on cloud services. It is imperative to acknowledge that the precise impacts of model compression may differ based on the employed approaches, the nature of the model, and the given task. Although model compression presents various advantages, it entails a compromise with model performance. The degree to which a model may be compressed while preserving satisfactory performance relies on the particular use case and the effectiveness of the compression techniques employed.

Future Directions

The advantages of edge ML are clear: it prepares the way for the final mile of AI, enables the delivery of highly effective intelligent services to users, drastically reduces reliance on centralized cloud servers, and may successfully safeguard data privacy. Reiterating that there are still some unresolved obstacles to achieving edge intelligence is important. It is essential to recognize, examine, and seek out fresh theoretical and technical answers to these problems. In this perspective, some significant edge ML issues are highlighted along with some potential solutions.

If the model is trained centrally, this issue could be resolved quickly. The ability to learn the invariant features of the variations is ensured by the centrally located huge training set. But this is outside the purview of edge intelligence. Future work on this issue should concentrate on finding ways to counteract the variation's detrimental impact on model accuracy. In order to do this, representation learning, and data augmentation are two potential research topics. To make the model more resistant to noise, data augmentation could be used to supplement the data during model training. For instance, speech recognition software for mobile devices introduces a variety of background noises to mask the fluctuation brought on by the environment.

In addition, noise brought on by sensor hardware might also be included to address the inconsistent problem. The models are more resistant to these variations thanks to the training with the augmented data. The effectiveness of models is significantly impacted by data representation. When creating models, representation learning focuses on learning how to represent data to extract more useful characteristics, which may also be used to disguise hardware disparities. For this issue, when using the same data source, the model's performance would be greatly enhanced if a "translation" could be established between two sensor representations. Therefore, representation learning is a viable approach to lessen the effects of inconsistent data. Future initiatives in this direction could include creating processing pipelines and data transformations that are more efficient.

The model is often first trained on a central server before being distributed on edge devices in edge ML-based AI applications. Once the training process is complete, the trained model won't need to be retrained. These statically trained models perform poorly and provide poor user experience because they are unable to handle fresh, unexpected data and tasks in unfamiliar

situations. On the other hand, local knowledge is the only input for models trained using a decentralized learning approach. As a result, these models might only become authorities in their specific localities. The quality-of-service declines as the serving region grows.

Two options may be taken into consideration to deal with this issue: sharing knowledge and lifetime machine learning. An advanced learning paradigm called lifetime machine learning (LML) permits ongoing knowledge building and self-learning on new tasks. Instead of being instructed by humans, machines are taught to acquire new knowledge on the basis of previously acquired knowledge. Meta learning, which enables computers to automatically learn new models, and LML are slightly distinct from one another. Lifelong machine learning (LML) could be used by edge devices with a variety of learnt tasks such as directory updating, apps updating or data adding/ deleting to adjust to changing environments and deal with unseen data. It is important to remember that the LML is not primarily intended for edge devices, therefore significant computing hardware is anticipated for the machines. Therefore, if LML is used, model design, model compression, and offloading mechanisms should also be considered.

The exchange of knowledge across various edge servers is made possible via knowledge sharing. If an edge server receives a task for which it lacks the knowledge necessary to deliver quality service, it may send knowledge requests to other edge servers. The server with the necessary knowledge answers the query and completes the task for users because the knowledge is distributed among various edge servers. In such a knowledge sharing paradigm, a method for knowledge appraisal and a system for knowledge querying are necessary.

The two most crucial phases of edge ML are data collecting and model training/inference. It is difficult to guarantee the accuracy and usefulness of the information contained in the collected data. When sensing and gathering data, data collectors use their own resources, such as battery life, bandwidth, and processing time. It is unrealistic to anticipate that all data collectors will be willing to contribute, let alone for resource-intensive preparation like feature extraction, data cleaning, and encryption. All participants must work together to complete a task in order to train or infer models.

References

[1] D. Łukasz , C. Thomas , S. A. Mohamed , L. Royson , K. Hyeji and D. L. Nicholas , "BRP-NAS: Prediction-based NAS using GCNs," 2021.

[2] B. Simone, C. Remi, C. Luigi and N. Paolo, "Benchmark Analysis of Representative Deep Neural Network Architectures," 2018.

[3] X. Mengwei , L. Jiawei , L. Yuanqiang , L. Felix Xiaozhu , L. Yunxin and L. Xuanzhe , "A First Look at Deep Learning Apps on Smartphones," 2021.

[4] C. Han , G. Chuang , W. Tianzhe , Z. Zhekai and H. Song , 2020.

[5] H. Song , M. Huizi and J. D. William , "DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION," 2016.

[6] H. Yihui He, L. Ji , L. Zhijian , W. Hanrui , L. Li-Jia and H. Song , "AMC: AutoML for Model Compression and Acceleration on Mobile Devices," 2018.

[7] L. Zechun , M. Haoyuan , Z. Xiangyu , G. Zichao and Y. Xin , "MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning," 2019.

[8] C. Han, Z. Ligeng and H. Song , "PROXYLESSNAS: DIRECT NEURAL ARCHITECTURE SEARCH ON TARGET TASK AND HARDWARE," 2019.

[9] T. Mingxing , C. Bo , P. Ruoming , V. Vijay , S. Mark , H. Andrew and V. L. Quoc V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," 2019.

[10] W. Bichen , . D. Xiaoliang, Z. Peizhao , W. Yanghan , S. Fei , W. Yiming , T. Yuandong , V. Peter Vajda, J. Yangqing and K. Kurt , "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search," 2019. [Online].

[11] Z. Li Lyna, Y. Yuqing, J. Yuhang , Z. Wenwu and L. Yunxin , "Fast Hardware-Aware Neural Architecture Search," 2020.

[12] L. Hao, K. Asim, D. Igor , S. Hanan and P. G. Hans , "Pruning Filters for Efficient ConvNets," 2017.

[13] L. Hanxiao , S. Karen and Y. Yiming , "DARTS: Differentiable Architecture Search," 2019.

[14] D. Xuanyi and Y. Yi, "NAS-BENCH-201: EXTENDING THE SCOPE OF REPRODUCIBLE NEURAL ARCHITECTURE SEARCH," 2020.

[15] Z. Li Lyna, H. Shihao, W. Jianyu, Z. Ningxin, C. Ting, Y. Yuqing and L. Yunxin, "nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices," July 2021.

[16] D. Jia, D. Wei , S. Richard , L. Li-Jia , L. Kai and F.-F. Li , "ImageNet: A Large-Scale Hierarchical Image Database," 2009.

[17] S. Mohanty,"PlantVillage-Dataset,"pp. https://github.com/spMohanty/PlantVillage-Dataset, 2018.

[18] J. Daniel, B. John, B. Stephen and M. Andrew, "Predicting the Computational Cost of Deep Learning Models," 2018.

[19] F. Galton, "Regression Towards Mediocrity in Hereditary Stature," The Journal of the Anthropological Institute of Great Britain and Ireland, 1886. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[20] L. L. Zhang et al., “nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices,” in Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services, New York, NY, USA, Jun. 2021, pp. 81–93. doi: 10.1145/3458864.3467882.

[21] C. Li et al., “HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark,” presented at the International Conference on Learning Representations, Feb. 2022. Accessed: Oct. 14, 2022. [Online].

[22] Ł. Dudziak, T. Chau, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane, “BRP-NAS: Prediction-based NAS using GCNs,” 2020, doi: 10.48550/ARXIV.2007.08668.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” Jul, 2017, doi: 10.1145/3065386.

[24]AI, D. (2022, March 25). 4 Parameters to Consider When Choosing Hardware for Deep Learning Inference. Deci AI. https://medium.com/deci-ai/4-parameters-to-consider-when-choosing-hardware-for-deep-learning-inference-b5a06eefd621

[25] J. Daniel, B. John, B. Stephen and M. Andrew, "Predicting the Computational Cost of Deep Learning Models," 2018.

[26] H.-L. Truong, T. Truong-Huu, and T.-D. Cao, “Making distributed edge machine learning for resource-constrained communities and environments smarter: contexts and challenges,” J Reliable Intell Environ, May 2022, doi: 10.1007/s40860-022-00176-3.

[27] M. Wolf, “Machine Learning + Distributed IoT = Edge Intelligence,” in 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Jul. 2019, pp. 1715–1719. doi: 10.1109/ICDCS.2019.00170.

[28] “What is Edge AI? Machine Learning + IoT,” Digi-Key Electronics. https://www.digikey.com/en/maker/projects/4f655838138941138aaad62c170827af (accessed Oct. 20, 2022).

[29] A. Marchisio et al., “Deep Learning for Edge Computing: Current Trends, Cross-Layer Optimizations, and Open Research Challenges,” in 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Jul. 2019, pp. 553–559. doi: 10.1109/ISVLSI.2019.00105.

[30] C. P. Filho et al., “A Systematic Literature Review on Distributed Machine Learning in Edge Computing,” Sensors, vol. 22, no. 7, Art. no. 7, Jan. 2022, doi: 10.3390/s22072665.

[31] Q. Xia, W. Ye, Z. Tao, J. Wu, and Q. Li, “A survey of federated learning for edge computing: Research problems and solutions,” High-Confidence Computing, vol. 1, no. 1, p. 100008, Jun. 2021, doi: 10.1016/j.hcc.2021.100008.

[32] “A Vision for the Future with Edge ML-Powered Devices - Data + AI Summit 2022 | Databricks.” https://databricks.com/dataaisummit/session/vision-future-edge-ml-powered-devices (accessed Oct. 26, 2022).

[33] N. Elgendy and A. Elragal, “Big Data Analytics: A Literature Review Paper,” in Advances in Data Mining. Applications and Theoretical Aspects, vol. 8557, P. Perner, Ed. Cham: Springer International Publishing, 2014, pp. 214–227. doi: 10.1007/978-3-319-08976-8_16.

[34] M. K. Abiodun et al., “Cloud and Big Data: A Mutual Benefit for Organization Development,” J. Phys.: Conf. Ser., vol. 1767, no. 1, p. 012020, Feb. 2021, doi: 10.1088/1742-6596/1767/1/012020.

[35] “Big Data using Cloud Computing - Opportunities for Small and Medium-sized Enterprises | European Journal of Economics and Business Studies,” Nov. 2021, Accessed: Oct. 26, 2022. [Online]. Available: https://revistia.com/index.php/ejes/article/view/5281

[36] W. Choi, J. Kim, S. Lee, and E. Park, “Smart home and internet of things: A bibliometric study,” Journal of Cleaner Production, vol. 301, p. 126908, Jun. 2021, doi: 10.1016/j.jclepro.2021.126908.

[37] Y. Li, Y. Zuo, H. Song, and Z. Lv, “Deep Learning in Security of Internet of Things,” IEEE Internet of Things Journal, pp. 1–1, 2021, doi: 10.1109/JIOT.2021.3106898.

[38] D. C. Nguyen et al., “6G Internet of Things: A Comprehensive Survey,” IEEE Internet of Things Journal, vol. 9, no. 1, pp. 359–383, Jan. 2022, doi: 10.1109/JIOT.2021.3103320.

[39] L. Yang, Z. Li, S. Ma, and X. Yang, “Artificial intelligence image recognition based on 5G deep learning edge algorithm of Digestive endoscopy on medical construction,” Alexandria Engineering Journal, vol. 61, no. 3, pp. 1852–1863, Mar. 2022, doi: 10.1016/j.aej.2021.07.007.

[40] T. A. Nguyen, D. Min, E. Choi, and J.-W. Lee, “Dependability and Security Quantification of an Internet of Medical Things Infrastructure Based on Cloud-Fog-Edge Continuum for Healthcare Monitoring Using Hierarchical Models,” IEEE Internet of Things Journal, vol. 8, no. 21, pp. 15704–15748, Nov. 2021, doi: 10.1109/JIOT.2021.3081420.

[41] K. He, J. Gong, L. Xie, X. Zhang, and D. Xu, “Regions Preserving Edge Enhancement for Multisensor-Based Medical Image Fusion,” IEEE Transactions on Instrumentation and Measurement, vol. 70, pp. 1–13, 2021, doi: 10.1109/TIM.2021.3066467.

[42] V. Hayyolalam, M. Aloqaily, Ö. Özkasap, and M. Guizani, “Edge Intelligence for Empowering IoT-Based Healthcare Systems,” IEEE Wireless Communications, vol. 28, no. 3, pp. 6–14, Jun. 2021, doi: 10.1109/MWC.001.2000345.

[43] R. Bogdan, A. Tatu, M. M. Crisan-Vida, M. Popa, and L. Stoicu-Tivadar, “A Practical Experience on the Amazon Alexa Integration in Smart Offices,” Sensors, vol. 21, no. 3, Art. no. 3, Jan. 2021, doi: 10.3390/s21030734.

[44] W. Ahmad, A. Rasool, A. R. Javed, T. Baker, and Z. Jalil, “Cyber Security in IoT-Based Cloud Computing: A Comprehensive Survey,” Electronics, vol. 11, no. 1, Art. no. 1, Jan. 2022, doi: 10.3390/electronics11010016.

[45] A. H. Shaikh and B. B. Meshram, “Security Issues in Cloud Computing,” in Intelligent Computing and Networking, Singapore, 2021, pp. 63–77. doi: 10.1007/978-981-15-7421-4_6.

[46] A. S. AlAhmad, H. Kahtan, Y. I. Alzoubi, O. Ali, and A. Jaradat, “Mobile cloud computing models security issues: A systematic review,” Journal of Network and Computer Applications, vol. 190, p. 103152, Sep. 2021, doi: 10.1016/j.jnca.2021.103152.

[47] G. Bhatti, H. Mohan, and R. Raja Singh, “Towards the future of smart electric vehicles: Digital twin technology,” Renewable and Sustainable Energy Reviews, vol. 141, p. 110801, May 2021, doi: 10.1016/j.rser.2021.110801.

[48] P. Dixit, P. Bhattacharya, S. Tanwar, and R. Gupta, “Anomaly detection in autonomous electric vehicles using AI techniques: A comprehensive survey,” Expert Systems, vol. 39, no. 5, Jun. 2022, doi: 10.1111/exsy.12754.

[49] J. Wei, T. Dingler, and V. Kostakos, “Developing the Proactive Speaker Prototype Based on Google Home,” in Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems, New York, NY, USA, May 2021, pp. 1–6. doi: 10.1145/3411763.3451642.

[50] M. J. Sánchez-Franco, F. J. Arenas-Márquez, and M. Alonso-Dos-Santos, “Using structural topic modelling to predict users’ sentiment towards intelligent personal agents. An application for Amazon’s echo and Google Home,” Journal of Retailing and Consumer Services, vol. 63, p. 102658, Nov. 2021, doi: 10.1016/j.jretconser.2021.102658.

[51] S. Yi, Z. Hao, Z. Qin, and Q. Li, “Fog Computing: Platform and Applications,” in 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), Nov. 2015, pp. 73–78. doi: 10.1109/HotWeb.2015.22.

[52] “Cisco’s Global Cloud Index Study: Acceleration of the Multicloud Era - Cisco Blogs.” https://blogs.cisco.com/news/acceleration-of-multicloud-era (accessed Oct. 28, 2022).

[53] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, “To offload or not to offload? The bandwidth and energy costs of mobile cloud computing,” in 2013 Proceedings IEEE INFOCOM, Apr. 2013, pp. 1285–1293. doi: 10.1109/INFCOM.2013.6566921.

[54] W. Hu et al., “Quantifying the Impact of Edge Computing on Mobile Applications,” in Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, New York, NY, USA, Aug. 2016, pp. 1–8. doi: 10.1145/2967360.2967369.

[55] Y. Yu, “Mobile edge computing towards 5G: Vision, recent progress, and open challenges,” China Commun., vol. 13, no. Supplement2, pp. 89–99, 2016, doi: 10.1109/CC.2016.7833463.

[56] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge Computing: Vision and Challenges,” IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[57] B. A. Mudassar, J. Hwan Ko, and S. Mukhopadhyay, “Edge-Cloud Collaborative Processing for Intelligent Internet of Things: A Case Study on Smart Surveillance,” in 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), Jun. 2018, pp. 1–6. doi: 10.1109/DAC.2018.8465862.

[58] A. Yousefpour et al., “All one needs to know about fog computing and related edge computing paradigms: A complete survey,” Journal of Systems Architecture, vol. 98, pp. 289–330, Sep. 2019, doi: 10.1016/j.sysarc.2019.02.009.

[59] “The past, present and future of edge ML.” https://www.imagimob.com/blog/the-past-present-and-future-of-edge-ml (accessed Nov. 02, 2022).

[60] P. Peniak, E. Bubeníková, and A. Kanáliková, “The Redundant Virtual Sensors via Edge Computing,” in 2021 International Conference on Applied Electronics (AE), Sep. 2021, pp. 1–5. doi: 10.23919/AE51540.2021.9542888.

[61] A. Paszkiewicz et al., “Network Load Balancing for Edge-Cloud Continuum Ecosystems,” in Innovations in Electrical and Electronic Engineering, Singapore, 2022, pp. 638–651. doi: 10.1007/978-981-19-1677-9_56.

[62] J. Otterbach and T. Wollmann, “Chameleon: A Semi-AutoML framework targeting quick and scalable development and deployment of production-ready ML systems for SMEs.” arXiv, May 08, 2021. doi: 10.48550/arXiv.2105.03669.

[63] J. Vykopal, P. Čeleda, P. Seda, V. Švábenský, and D. Tovarňák, “Scalable Learning Environments for Teaching Cybersecurity Hands-on,” in 2021 IEEE Frontiers in Education Conference (FIE), Oct. 2021, pp. 1–9. doi: 10.1109/FIE49875.2021.9637180.

[64] K. Ray and A. Banerjee, “Horizontal Auto-Scaling for Multi-Access Edge Computing Using Safe Reinforcement Learning,” ACM Trans. Embed. Comput. Syst., vol. 20, no. 6, p. 109:1-109:33, Oct. 2021, doi: 10.1145/3475991.

[65] A. Abouaomar, S. Cherkaoui, Z. Mlika, and A. Kobbane, “Resource Provisioning in Edge Computing for Latency-Sensitive Applications,” IEEE Internet of Things Journal, vol. 8, no. 14, pp. 11088–11099, Jul. 2021, doi: 10.1109/JIOT.2021.3052082.

[66] B. Wang, A. Ali-Eldin, and P. Shenoy, “LaSS: Running Latency Sensitive Serverless Computations at the Edge,” in Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, New York, NY, USA, Jun. 2021, pp. 239–251. doi: 10.1145/3431379.3460646.

[67] A. Biswas, A. Jain, and Mohana, “Survey on Edge Computing–Key Technology in Retail Industry,” in Computer Networks and Inventive Communication Technologies, Singapore, 2021, pp. 97–106. doi: 10.1007/978-981-15-9647-6_7.

[68] R. Dave, N. Seliya, and N. Siddiqui, “The Benefits of Edge Computing in Healthcare, Smart Cities, and IoT,” JCSA, vol. 9, no. 1, pp. 23–34, Oct. 2021, doi: 10.12691/jcsa-9-1-3.

[69] B. Parekh and K. Amin, “Edge Intelligence: A Robust Reinforcement of Edge Computing and Artificial Intelligence,” in Innovations in Information and Communication Technologies (IICT-2020), Cham, 2021, pp. 461–468. doi: 10.1007/978-3-030-66218-9_55.

[70] M. Adhikari, A. Munusamy, A. Hazra, V. G. Menon, V. Anavangot, and D. Puthal, “Security in Edge-Centric Intelligent Internet of Vehicles: Issues and Remedies,” IEEE Consumer Electronics Magazine, vol. 11, no. 6, pp. 24–31, Nov. 2022, doi: 10.1109/MCE.2021.3116415.

[71] Mohd. Sarim, M. S. Ansari, N. Kanwal, and M. Asghar, "Improved Privacy-Ensuring Data-Fusion and Service Recommendation for Users in Smart Cities," in 2021 IEEE International Smart Cities Conference (ISC2), Sep. 2021, pp. 1–7. doi: 10.1109/ISC253183.2021.9562777.

[72] M. Eisoldt et al., "ReconfROS: Running ROS on Reconfigurable SoCs," in Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings, New York, NY, USA, Feb. 2021, pp. 16–21. doi: 10.1145/3444950.3444959.

[73] M. Flottmann et al., "Energy-efficient FPGA-accelerated LiDAR-based SLAM for embedded robotics," in 2021 International Conference on Field-Programmable Technology (ICFPT), Dec. 2021, pp. 1–6. doi: 10.1109/ICFPT52863.2021.9609934.

[74] B. M. Magnussen, T. Kawasumi, H. Mikami, K. Kimura, and H. Kasahara, "Performance Evaluation of OSCAR Multi-target Automatic Parallelizing Compiler on Intel, AMD, Arm and RISC-V Multicores," in Languages and Compilers for Parallel Computing, Cham, 2022, pp. 50–64. doi: 10.1007/978-3-030-99372-6_4.

[75] R. Ding, Z. Zhang, X. Zhang, C. Gongye, Y. Fei, and A. A. Ding, "A Cross-Platform Cache Timing Attack Framework via Deep Learning," in 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Mar. 2022, pp. 676–681. doi: 10.23919/DATE54114.2022.9774612.

[76] A. Kelkar and C. Dick, "NVIDIA Aerial GPU Hosted AI-on-5G," in 2021 IEEE 4th 5G World Forum (5GWF), Oct. 2021, pp. 64–69. doi: 10.1109/5GWF52925.2021.00019.

[77] Z. Xue et al., "A Resource-Constrained and Privacy-Preserving Edge-Computing-Enabled Clinical Decision System: A Federated Reinforcement Learning Approach," IEEE Internet of Things Journal, vol. 8, no. 11, pp. 9122–9138, Jun. 2021, doi: 10.1109/JIOT.2021.3057653.

[78] H. A. Abdelhafez, H. Halawa, K. Pattabiraman, and M. Ripeanu, "Snowflakes at the Edge: A Study of Variability among NVIDIA Jetson AGX Xavier Boards," in Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, New York, NY, USA, Apr. 2021, pp. 1–6. doi: 10.1145/3434770.3459729.

[79] H. Xue, B. Hein, M. Bakr, G. Schildbach, B. Abel, and E. Rueckert, "Using Deep Reinforcement Learning with Automatic Curriculum Learning for Mapless Navigation in Intralogistics," Applied Sciences, vol. 12, no. 6, Art. no. 6, Jan. 2022, doi: 10.3390/app12063153.

[80] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking." arXiv, Mar. 18, 2019. doi: 10.48550/arXiv.1903.07486.

[81] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration," IEEE Communications Surveys & Tutorials, vol. 19, no. 3, pp. 1657–1681, 2017, doi: 10.1109/COMST.2017.2705720

[82] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, "Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things," IEEE Internet of Things Journal, vol. 7, no. 8, pp. 6722–6747, Aug. 2020, doi: 10.1109/JIOT.2020.3004500.

[83] J. Pan and J. McElhannon, "Future Edge Cloud and Edge Computing for Internet of Things Applications," IEEE Internet of Things Journal, vol. 5, no. 1, pp. 439–449, Feb. 2018, doi: 10.1109/JIOT.2017.2767608.

[84] G. Demosthenous and V. Vassiliades, "Continual Learning on the Edge with TensorFlow Lite." arXiv, May 05, 2021. doi: 10.48550/arXiv.2105.01946.

[85] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12, Helsinki, Finland, 2012, p. 13. doi: 10.1145/2342509.2342513.

[86] P. Garcia Lopez et al., "Edge-centric Computing: Vision and Challenges," SIGCOMM Comput. Commun. Rev., vol. 45, no. 5, pp. 37–42, Sep. 2015, doi: 10.1145/2831347.2831354.

[87] M. Adhikari and A. Hazra, "6G-Enabled Ultra-Reliable Low-Latency Communication in Edge Networks," IEEE Communications Standards Magazine, vol. 6, no. 1, pp. 67–74, Mar. 2022, doi: 10.1109/MCOMSTD.0001.2100098.

[88] R. Gupta, D. Reebadiya, and S. Tanwar, "6G-enabled Edge Intelligence for Ultra -Reliable Low Latency Applications: Vision and Mission," Computer Standards & Interfaces, vol. 77, p. 103521, Aug. 2021, doi: 10.1016/j.csi.2021.103521.

[89] A. Y. Ding, M. Janssen, and J. Crowcroft, "Trustworthy and Sustainable Edge AI: A Research Agenda," in 2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), Dec. 2021, pp. 164–172. doi: 10.1109/TPSISA52974.2021.00019.

[90] C. Jiang et al., "Energy aware edge computing: A survey," Computer Communications, vol. 151, pp. 556–580, Feb. 2020, doi: 10.1016/j.comcom.2020.01.004.

[91] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A Survey on Edge Computing Systems and Tools," Proceedings of the IEEE, vol. 107, no. 8, pp. 1537–1562, Aug. 2019, doi: 10.1109/JPROC.2019.2920341.

[92] Sujith Ravi. 2015. ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections. arXiv:1708.00630 (2015).

[93] Faiza Samreen et al. 2016. Daleel: Simplifying Cloud Instance Selection Using Machine Learning. In NOMS '16.

[94] S.Mohanty,"PlantVillage-Dataset,". https://github.com/spMohanty/PlantVillage-Datase (2018).

[95] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. 2014. Striving for Simplicity: The All Convolutional Net. CoRR abs/1412.6806 (2014).

[96] JJ Allaire, Dirk Eddelbuettel, Nick Golding, and Yuan Tang. 2016. TensorFlow for R. https://tensorflow.rstudio.com/

[97] Dario Amodei et al. 2016. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In ICML '16.

[98] Dzmitry Bahdanau et al. 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014).

[99] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In Conference on Embedded Networked Sensor Systems.

[100] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resourceefficient and QoS-aware Cluster Management. In ASPLOS '14.

[101] Yi Sun, Yuheng Chen, et al. 2014. Deep learning face representation by joint identification-verification. In NIPS '14.

[102] Surat Teerapittayanon et al. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In ICDCS '17.

[103] Lakshmish Ramaswamy and Alqahtani Ola. 2022. A Layer Decomposition Approach to Inference Time Prediction of Deep Learning Architectures. ICMLA IEEE'22.

[104] Mohamed Said, Belal AA, Abd-Elmabod Kotb, and Shirbeny Mohammed. Smart farming for improving agricultural management. NARSS 2021.

[105] NVIDIA Corporation. 2016. nvidia-smi 367.38 https://developer.nvidia.com/.

[106] Jeff Donahue et al. 2014. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In ICML '14.

[107] Murali Krishna Emani et al. 2013. Smart, adaptive mapping of parallelism in the presence of external workload. In CGO '13.

[108] Petko Georgiev et al. 2017. Low-resource Multi-task Audio Sensing Representations. ACM Interact. Mob. Wearable Ubiquitous Technol. (2017).

[109] Song Han et al. 2015. Learning both weights and connections for efficient neural network. In NIPS '15.

[110] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In ISCA '16.

[111] M Hassaballah et al. 2016. Image features detection, description and matching. In Image Feature Detectors and Descriptors.

[112] Kaiming He et al. 2016. Deep residual learning for image recognition. In CVPR '16.

[113] Andrew G. Howard et al. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017).

[114] Forrest N. Iandola et al. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MBmodelsize. CoRR abs/1602.07360 (2016).

[115] Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. 2015. Flattened Convolutional Neural Networks for Feedforward Acceleration. (2015).

[116] Yiping Kang et al. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In ASPLOS '17.

[117] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In NIPS '12.

[118] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A software accelerator for low-power deep learning inference on mobile devices. In IPSN '16.

[119] Seyyed Salar Latifi Oskouei et al. 2016. Cnndroid: GPU-accelerated execution of trained deep convolutional neural networks on android. In Multimedia Conference.

[120] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An Analysis of Deep Neural Network Models for Practical Applications. CoRR (2016).

[121] Shizhao Chen et al. 2018. Adaptive Optimization of Sparse MatrixVector Multiplication on Emerging Many-Core Architectures. In HPCC '18.

[122] Wenlin Chen et al. 2015. Compressing Neural Networks with the Hashing Trick. In ICML '16.

[123] Kyunghyun Cho et al. 2014. Learning phrase representations using RNNencoder-decoder for statistical machine translation. In EMNLP '14.

[124] Chris Cumminset al. 2017. End-to-end Deep Learning of Optimization Heuristics. In PACT '17.

[125] Honglak Lee et al. 2009. Unsupervised Feature Learning for Audio Classification Using Convolutional Deep Belief Networks. In NIPS '09.

[126] M. Wolf, "Machine Learning + Distributed IoT = Edge Intelligence," in 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Jul. 2019, pp. 1715–1719. doi: 10.1109/ICDCS.2019.00170.

[127] "What is Edge AI? Machine Learning + IoT," Digi-Key Electronics. https://www.digikey.com/en/maker/projects/4f655838138941138aaad62c170827af (accessed Oct. 20, 2022).

[128] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog Computing: Platform and Applications," in 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), Nov. 2015, pp. 73–78. doi: 10.1109/HotWeb.2015.22.

[129] Y. Yu, "Mobile edge computing towards 5G: Vision, recent progress, and open challenges," China Commun., vol. 13, no. Supplement2, pp. 89–99, 2016, doi: 10.1109/CC.2016.7833463.

[130] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[131] "The past, present and future of edge ML." https://www.imagimob.com/blog/the-past-present-and-future-of-edge-ml (accessed Nov. 02, 2022).

[132] P. Peniak, E. Bubeníková, and A. Kanáliková, “The Redundant Virtual Sensors via Edge Computing,” in 2021 International Conference on Applied Electronics (AE), Sep. 2021, pp. 1–5. doi: 10.23919/AE51540.2021.9542888.

[133] A. Paszkiewicz et al., “Network Load Balancing for Edge-Cloud Continuum Ecosystems,” in Innovations in Electrical and Electronic Engineering, Singapore, 2022, pp. 638–651. doi: 10.1007/978-981-19-1677-9_56.

[134] J. Otterbach and T. Wollmann, “Chameleon: A Semi-AutoML framework targeting quick and scalable development and deployment of production-ready ML systems for SMEs.” arXiv, May 08, 2021. doi: 10.48550/arXiv.2105.03669.

[135] J. Vykopal, P. Čeleda, P. Seda, V. Švábenský, and D. Tovarňák, “Scalable Learning Environments for Teaching Cybersecurity Hands-on,” in 2021 IEEE Frontiers in Education Conference (FIE), Oct. 2021, pp. 1–9. doi: 10.1109/FIE49875.2021.9637180.

[136] K. Ray and A. Banerjee, “Horizontal Auto-Scaling for Multi-Access Edge Computing Using Safe Reinforcement Learning,” ACM Trans. Embed. Comput. Syst., vol. 20, no. 6, p. 109:1-109:33, Oct. 2021, doi: 10.1145/3475991.

[137] A. Abouaomar, S. Cherkaoui, Z. Mlika, and A. Kobbane, “Resource Provisioning in Edge Computing for Latency-Sensitive Applications,” IEEE Internet of Things Journal, vol. 8, no. 14, pp. 11088–11099, Jul. 2021, doi: 10.1109/JIOT.2021.3052082.

[138] B. Wang, A. Ali-Eldin, and P. Shenoy, “LaSS: Running Latency Sensitive Serverless Computations at the Edge,” in Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, New York, NY, USA, Jun. 2021, pp. 239–251. doi: 10.1145/3431379.3460646.

[139] A. Biswas, A. Jain, and Mohana, “Survey on Edge Computing–Key Technology in Retail Industry,” in Computer Networks and Inventive Communication Technologies, Singapore, 2021, pp. 97–106. doi: 10.1007/978-981-15-9647-6_7.

[140] M. Adhikari, A. Munusamy, A. Hazra, V. G. Menon, V. Anavangot, and D. Puthal, “Security in Edge-Centric Intelligent Internet of Vehicles: Issues and Remedies,” IEEE Consumer Electronics Magazine, vol. 11, no. 6, pp. 24–31, Nov. 2022, doi: 10.1109/MCE.2021.3116415.

[141] Mohd. Sarim, M. S. Ansari, N. Kanwal, and M. Asghar, “Improved Privacy-Ensuring Data-Fusion and Service Recommendation for Users in Smart Cities,” in 2021 IEEE International Smart Cities Conference (ISC2), Sep. 2021, pp. 1–7. doi: 10.1109/ISC253183.2021.9562777.

[142] M. Eisoldt et al., “ReconfROS: Running ROS on Reconfigurable SoCs,” in Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings, New York, NY, USA, Feb. 2021, pp. 16–21. doi: 10.1145/3444950.3444959.

[143] M. Flottmann et al., “Energy-efficient FPGA-accelerated LiDAR-based SLAM for embedded robotics,” in 2021 International Conference on Field-Programmable Technology (ICFPT), Dec. 2021, pp. 1–6. doi: 10.1109/ICFPT52863.2021.9609934.

[144] B. M. Magnussen, T. Kawasumi, H. Mikami, K. Kimura, and H. Kasahara, “Performance Evaluation of OSCAR Multi-target Automatic Parallelizing Compiler on Intel, AMD, Arm and RISC-V Multicores,” in Languages and Compilers for Parallel Computing, Cham, 2022, pp. 50–64. doi: 10.1007/978-3-030-99372-6_4.

[145] A. Kelkar and C. Dick, “NVIDIA Aerial GPU Hosted AI-on-5G,” in 2021 IEEE 4th 5G World Forum (5GWF), Oct. 2021, pp. 64–69. doi: 10.1109/5GWF52925.2021.00019.

[146] Z. Xue et al., “A Resource-Constrained and Privacy-Preserving Edge-Computing-Enabled Clinical Decision System: A Federated Reinforcement Learning Approach,” IEEE Internet of Things Journal, vol. 8, no. 11, pp. 9122–9138, Jun. 2021, doi: 10.1109/JIOT.2021.3057653.

[147] Md. I. Uddin, Md. S. Alamgir, Md. M. Rahman, M. S. Bhuiyan, and M. A. Moral, “AI Traffic Control System Based on Deepstream and IoT Using NVIDIA Jetson Nano,” in 2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST), DHAKA, Bangladesh, Jan. 2021, pp. 115–119. doi: 10.1109/ICREST51555.2021.9331256.

[148] S. Valladares, M. Toscano, R. Tufiño, P. Morillo, and D. Vallejo-Huanga, “Performance Evaluation of the Nvidia Jetson Nano Through a Real-Time Machine Learning Application,” in Intelligent Human Systems Integration 2021, vol. 1322, D. Russo, T. Ahram, W. Karwowski, G. Di Bucchianico, and R. Taiar, Eds. Cham: Springer International Publishing, 2021, pp. 343–349. doi: 10.1007/978-3-030-68017-6_51.

[149] H. M. Mohan, S. Anitha, R. Chai, and S. H. Ling, “Edge Artificial Intelligence: Real-Time Noninvasive Technique for Vital Signs of Myocardial Infarction Recognition Using Jetson Nano,” Advances in Human-Computer Interaction, vol. 2021, pp. 1–19, Aug. 2021, doi: 10.1155/2021/6483003.

[150] H. A. Abdelhafez, H. Halawa, K. Pattabiraman, and M. Ripeanu, “Snowflakes at the Edge: A Study of Variability among NVIDIA Jetson AGX Xavier Boards,” in Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, New York, NY, USA, Apr. 2021, pp. 1–6. doi: 10.1145/3434770.3459729.

[151] H. Xue, B. Hein, M. Bakr, G. Schildbach, B. Abel, and E. Rueckert, “Using Deep Reinforcement Learning with Automatic Curriculum Learning for Mapless Navigation in Intralogistics,” Applied Sciences, vol. 12, no. 6, Art. no. 6, Jan. 2022, doi: 10.3390/app12063153.

[152] T. Aboneh, A. Rorissa, R. Srinivasagan, and A. Gemechu, “Computer Vision Framework for Wheat Disease Identification and Classification Using Jetson GPU Infrastructure,” Technologies, vol. 9, no. 3, Art. no. 3, Sep. 2021, doi: 10.3390/technologies9030047.

[153] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, “Dissecting the NVidia Turing T4 GPU via Microbenchmarking.” arXiv, Mar. 18, 2019. doi: 10.48550/arXiv.1903.07486.

[154] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, “On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration,” IEEE Communications Surveys & Tutorials, vol. 19, no. 3, pp. 1657–1681, 2017, doi: 10.1109/COMST.2017.2705720.

[155] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, “Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things,” IEEE Internet of Things Journal, vol. 7, no. 8, pp. 6722–6747, Aug. 2020, doi: 10.1109/JIOT.2020.3004500.

[156] J. Pan and J. McElhannon, “Future Edge Cloud and Edge Computing for Internet of Things Applications,” IEEE Internet of Things Journal, vol. 5, no. 1, pp. 439–449, Feb. 2018, doi: 10.1109/JIOT.2017.2767608.

[157] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC ’12, Helsinki, Finland, 2012, p. 13. doi: 10.1145/2342509.2342513.

[158] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, “Collaborative Mobile Edge Computing in 5G Networks: New Paradigms, Scenarios, and Challenges,” IEEE Commun. Mag., vol. 55, no. 4, pp. 54–61, Apr. 2017, doi: 10.1109/MCOM.2017.1600863.

[159] P. Garcia Lopez et al., “Edge-centric Computing: Vision and Challenges,” SIGCOMM Comput. Commun. Rev., vol. 45, no. 5, pp. 37–42, Sep. 2015, doi: 10.1145/2831347.2831354.

[160] M. Adhikari and A. Hazra, “6G-Enabled Ultra-Reliable Low-Latency Communication in Edge Networks,” IEEE Communications Standards Magazine, vol. 6, no. 1, pp. 67–74, Mar. 2022, doi: 10.1109/MCOMSTD.0001.2100098.

[161] M. A. Siddiqi, H. Yu, and J. Joung, “5G Ultra-Reliable Low-Latency Communication Implementation Challenges and Operational Issues with IoT Devices,” Electronics, vol. 8, no. 9, Art. no. 9, Sep. 2019, doi: 10.3390/electronics8090981.

[162] R. Gupta, D. Reebadiya, and S. Tanwar, “6G-enabled Edge Intelligence for Ultra -Reliable Low Latency Applications: Vision and Mission,” Computer Standards & Interfaces, vol. 77, p. 103521, Aug. 2021, doi: 10.1016/j.csi.2021.103521.

[163] A. Y. Ding, M. Janssen, and J. Crowcroft, “Trustworthy and Sustainable Edge AI: A Research Agenda,” in 2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), Dec. 2021, pp. 164–172. doi: 10.1109/TPSISA52974.2021.00019.

[164] C. Jiang et al., “Energy aware edge computing: A survey,” Computer Communications, vol. 151, pp. 556–580, Feb. 2020, doi: 10.1016/j.comcom.2020.01.004.

[165] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, “A Survey on Edge Computing Systems and Tools,” Proceedings of the IEEE, vol. 107, no. 8, pp. 1537–1562, Aug. 2019, doi: 10.1109/JPROC.2019.2920341.

[166] D. C. Nguyen et al., “Federated Learning Meets Blockchain in Edge Computing: Opportunities and Challenges,” IEEE Internet of Things Journal, vol. 8, no. 16, pp. 12806–12825, Aug. 2021, doi: 10.1109/JIOT.2021.3072611.

[167] F. Paissan, A. Ancilotto, and E. Farella, “PhiNets: a scalable backbone for low-power AI at the edge,” ACM Trans. Embed. Comput. Syst., Feb. 2022, doi: 10.1145/3510832.

[168] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, “In-Edge AI: Intelligentizing Mobile Edge Computing, Caching and Communication by Federated Learning,” IEEE Network, vol. 33, no. 5, pp. 156–165, Sep. 2019, doi: 10.1109/MNET.2019.1800286.

[169] E. Li, Z. Zhou, and X. Chen, “Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy,” in Proceedings of the 2018 Workshop on Mobile Edge Communications, Budapest Hungary, Aug. 2018, pp. 31–36. doi: 10.1145/3229556.3229562.

[170] Z. Wang, Y. Cui, and Z. Lai, “A First Look at Mobile Intelligence: Architecture, Experimentation and Challenges,” IEEE Network, vol. 33, no. 4, pp. 120–125, Jul. 2019, doi: 10.1109/MNET.2019.1700470.

[171] H. Khelifi et al., “Bringing Deep Learning at the Edge of Information-Centric Internet of Things,” IEEE Communications Letters, vol. 23, no. 1, pp. 52–55, Jan. 2019, doi: 10.1109/LCOMM.2018.2875978.

[172] N. D. Lane and P. Warden, “The Deep (Learning) Transformation of Mobile and Embedded Computing,” Computer, vol. 51, no. 5, pp. 12–16, May 2018, doi: 10.1109/MC.2018.2381129.

[173] F. Chen, M. Luo, Z. Dong, Z. Li, and X. He, “Federated Meta-Learning with Fast Convergence and Efficient Communication.” arXiv, Dec. 14, 2019. doi: 10.48550/arXiv.1802.07876.

[174] Y. Chen, J. Wang, C. Yu, W. Gao, and X. Qin, “FedHealth: A Federated Transfer Learning Framework for Wearable Healthcare.” arXiv, May 11, 2021. doi: 10.48550/arXiv.1907.09173.

[175] E. Peltonen et al., “6G White Paper on Edge Intelligence.” arXiv, Apr. 30, 2020. doi: 10.48550/arXiv.2004.14850.

[176] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, “Convergence of Edge Computing and Deep Learning: A Comprehensive Survey,” IEEE Communications Surveys & Tutorials, vol. 22, no. 2, pp. 869–904, 2020, doi: 10.1109/COMST.2020.2970550.

[177] “Toward an Intelligent Edge: Wireless Communication Meets Machine Learning | IEEE Journals & Magazine | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/8970161 (accessed Oct. 26, 2022).

[178] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards wearable cognitive assistance,” in Proceedings of the 12th annual international conference on Mobile systems, applications, and services, Bretton Woods New Hampshire USA, Jun. 2014, pp. 68–81. doi: 10.1145/2594368.2594383.

[179] “Squeezing Deep Learning into Mobile and Embedded Devices | IEEE Journals & Magazine | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/7994570 (accessed Oct. 26, 2022).

[180] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, “An Early Resource Characterization of Deep Learning on Wearables, Smartphones and Internet-of-Things Devices,” in Proceedings of the 2015 International Workshop on Internet of Things towards Applications, Seoul South Korea, Nov. 2015, pp. 7–12. doi: 10.1145/2820975.2820980.

[181] V. Radu et al., “Multimodal Deep Learning for Activity and Context Recognition,” Proc. ACM Interact. Mob. Wearable Ubiquitous Technol., vol. 1, no. 4, pp. 1–27, Jan. 2018, doi: 10.1145/3161174.

[182] B. McMahan and D. Ramage, “Federated Learning: Collaborative Machine Learning without Centralized Training Data,” Apr. 06, 2017. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html (accessed Oct. 26, 2022).

[183] D. Liu and C. Yang, “A Learning-Based Approach to Joint Content Caching and Recommendation at Base Stations,” in 2018 IEEE Global Communications Conference (GLOBECOM), Dec. 2018, pp. 1–7. doi: 10.1109/GLOCOM.2018.8647827.

[184] L. Cavigelli and L. Benini, “CBinfer: Exploiting Frame-to-Frame Locality for Faster Convolutional Network Inference on Video Streams,” IEEE Transactions on Circuits and Systems for Video Technology, vol. 30, no. 5, pp. 1451–1465, May 2020, doi: 10.1109/TCSVT.2019.2903421.

[185] L. N. Huynh, R. K. Balan, and Y. Lee, “Demo: DeepMon: Building Mobile GPU Deep Learning Models for Continuous Vision Applications,” in Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, Niagara Falls New York USA, Jun. 2017, pp. 186–186. doi: 10.1145/3081333.3089331.

[186] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, “Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices,” in Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, Seoul South Korea, Nov. 2015, pp. 155–168. doi: 10.1145/2809695.2809711.

[187] B. Chen, C. Yang, and Z. Xiong, “Optimal Caching and Scheduling for Cache-Enabled D2D Communications,” IEEE Communications Letters, vol. 21, no. 5, pp. 1155–1158, May 2017, doi: 10.1109/LCOMM.2017.2652440.

[188] N. Giatsoglou, K. Ntontin, E. Kartsakli, A. Antonopoulos, and C. Verikoukis, “D2D-Aware Device Caching in mmWave-Cellular Networks,” IEEE Journal on Selected Areas in Communications, vol. 35, no. 9, pp. 2025–2037, Sep. 2017, doi: 10.1109/JSAC.2017.2720818.

[189] L. Qiu and G. Cao, “Popularity-Aware Caching Increases the Capacity of Wireless Networks,” IEEE Transactions on Mobile Computing, vol. 19, no. 1, pp. 173–187, Jan. 2020, doi: 10.1109/TMC.2019.2892419.

[190] D. Malak and M. Al-Shalash, “Optimal caching for device-to-device content distribution in 5G networks,” in 2014 IEEE Globecom Workshops (GC Wkshps), Dec. 2014, pp. 863–868. doi: 10.1109/GLOCOMW.2014.7063541.

[191] S. Krishnan, M. Afshang, and H. S. Dhillon, “Effect of Retransmissions on Optimal Caching in Cache-Enabled Small Cell Networks,” IEEE Transactions on Vehicular Technology, vol. 66, no. 12, pp. 11383–11387, Dec. 2017, doi: 10.1109/TVT.2017.2721839.

[192] N. Golrezaei, A. G. Dimakis, and A. F. Molisch, “Wireless device-to-device communications with distributed caching,” in 2012 IEEE International Symposium on Information Theory Proceedings, Jul. 2012, pp. 2781–2785. doi: 10.1109/ISIT.2012.6284029.

[193] N. Naderializadeh, D. T. H. Kao, and A. S. Avestimehr, “How to utilize caching to improve spectral efficiency in device-to-device wireless networks,” in 2014 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton), Sep. 2014, pp. 415–422. doi: 10.1109/ALLERTON.2014.7028485.

[194] Z. Chen, Y. Liu, B. Zhou, and M. Tao, “Caching incentive design in wireless D2D networks: A Stackelberg game approach,” in 2016 IEEE International Conference on Communications (ICC), May 2016, pp. 1–6. doi: 10.1109/ICC.2016.7511284.

[195] M. Taghizadeh, K. Micinski, S. Biswas, C. Ofria, and E. Torng, “Distributed Cooperative Caching in Social Wireless Networks,” IEEE Transactions on Mobile Computing, vol. 12, no. 6, pp. 1037–1053, Jun. 2013, doi: 10.1109/TMC.2012.66.

[196] P. Blasco and D. Gündüz, “Learning-based optimization of cache content in a small cell base station,” in 2014 IEEE International Conference on Communications (ICC), Jun. 2014, pp. 1897–1903. doi: 10.1109/ICC.2014.6883600.

[197] E. Baştuğ, J.-L. Guénégo, and M. Debbah, “Proactive small cell networks,” in ICT 2013, May 2013, pp. 1–5. doi: 10.1109/ICTEL.2013.6632164.

[198] O. Valery, P. Liu, and J.-J. Wu, “CPU/GPU Collaboration Techniques for Transfer Learning on Mobile Devices,” in 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), Dec. 2017, pp. 477–484. doi: 10.1109/ICPADS.2017.00069.

[199] T. Miu, P. Missier, and T. Plötz, “Bootstrapping Personalised Human Activity Recognition Models Using Online Active Learning,” in 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, Oct. 2015, pp. 1138–1147. doi: 10.1109/CIT/IUCC/DASC/PICOM.2015.170.

[200] H. Zeng and V. Prasanna, “GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms,” in Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Feb. 2020, pp. 255–265. doi: 10.1145/3373087.3375312.

[201] V. Smith, C.-K. Chiang, M. Sanjabi, and A. Talwalkar, “Federated Multi-Task Learning.” arXiv, Feb. 27, 2018. doi: 10.48550/arXiv.1705.10467.

[202] B. Biggio, B. Nelson, and P. Laskov, “Poisoning Attacks against Support Vector Machines.” arXiv, Mar. 25, 2013. doi: 10.48550/arXiv.1206.6389.

[203] J. Steinhardt, P. W. W. Koh, and P. S. Liang, “Certified Defenses for Data Poisoning Attacks,” in Advances in Neural Information Processing Systems, 2017, vol. 30. Accessed: Oct. 27, 2022. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/9d7311ba459f9e45ed746755a32dcd11-Abstract.html

[204] C. Fung, C. J. M. Yoon, and I. Beschastnikh, “Mitigating Sybils in Federated Learning Poisoning.” arXiv, Jul. 15, 2020. doi: 10.48550/arXiv.1808.04866.

[205] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, “Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates,” in Proceedings of the 35th International Conference on Machine Learning, Jul. 2018, pp. 5650–5659. Accessed: Oct. 27, 2022. [Online]. Available: https://proceedings.mlr.press/v80/yin18a.html

[206] Y. Gal and Z. Ghahramani, “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning,” in Proceedings of The 33rd International Conference on Machine Learning, Jun. 2016, pp. 1050–1059. Accessed: Oct. 27, 2022. [Online]. Available: https://proceedings.mlr.press/v48/gal16.html

[207] S. Yao et al., “RDeepSense: Reliable Deep Mobile Computing Models with Uncertainty Estimations,” Proc. ACM Interact. Mob. Wearable Ubiquitous Technol., vol. 1, no. 4, p. 173:1-173:26, Jan. 2018, doi: 10.1145/3161181.

[208] C. N. Duong, K. G. Quach, I. Jalata, N. Le, and K. Luu, “MobiFace: A Lightweight Deep Learning Face Recognition on Mobile Devices,” in 2019 IEEE 10th International Conference on Biometrics Theory, Applications and Systems (BTAS), Sep. 2019, pp. 1–6. doi: 10.1109/BTAS46853.2019.9185981.

[219] S. Chen, Y. Liu, X. Gao, and Z. Han, “MobileFaceNets: Efficient CNNs for Accurate Real-Time Face Verification on Mobile Devices,” in Biometric Recognition, Cham, 2018, pp. 428–438. doi: 10.1007/978-3-319-97909-0_46.

[210] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 6848–6856. Accessed: Oct. 27, 2022. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2018/html/Zhang_ShuffleNet_An_Extremely_CVPR_2018_paper.html

[211] Z. Qin, Z. Zhang, S. Zhang, H. Yu, and Y. Peng, “Merging-and-Evolution Networks for Mobile Vision Applications,” IEEE Access, vol. 6, pp. 31294–31306, 2018, doi: 10.1109/ACCESS.2018.2843341.

[212] A. G. Howard et al., “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” arXiv, Apr. 16, 2017. doi: 10.48550/arXiv.1704.04861.

[213] Y. Zhang, N. Suda, L. Lai, and V. Chandra, “Hello Edge: Keyword Spotting on Microcontrollers.” arXiv, Feb. 14, 2018. doi: 10.48550/arXiv.1711.07128.

[214] D. Wofk, F. Ma, T.-J. Yang, S. Karaman, and V. Sze, “FastDepth: Fast Monocular Depth Estimation on Embedded Systems,” in 2019 International Conference on Robotics and Automation (ICRA), May 2019, pp. 6101–6108. doi: 10.1109/ICRA.2019.8794182.

[215] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710. Accessed: Oct. 27, 2022. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2018/html/Zoph_Learning_Transferable_Architectures_CVPR_2018_paper.html

[216] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, Art. no. 01, Jul. 2019, doi: 10.1609/aaai.v33i01.33014780.

[217] M. Tan et al., "MnasNet: Platform-Aware Neural Architecture Search for Mobile," presented at the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 2820–2828. Accessed: Oct. 27, 2022. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2019/html/Tan_MnasNet_Platform-Aware_Neural_Architecture_Search_for_Mobile_CVPR_2019_paper

[218] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up Convolutional Neural Networks with Low Rank Expansions." arXiv, May 15, 2014. doi: 10.48550/arXiv.1405.3866.

[219] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," in Advances in Neural Information Processing Systems, 2014, vol. 27. Accessed: Oct. 27, 2022. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/2afe4567e1bf64d32a5527244d104cea-Abstract.html

[220] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications." arXiv, Feb. 24, 2016. doi: 10.48550/arXiv.1511.06530.

[221] S. Zagoruyko and N. Komodakis, "Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer." arXiv, Feb. 12, 2017. doi: 10.48550/arXiv.1612.03928.

[222] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, New York, NY, USA, Aug. 2006, pp. 535–541. doi: 10.1145/1150402.1150464.

[223] A. Wong, M. Famuori, M. J. Shafiee, F. Li, B. Chwyl, and J. Chung, "YOLO Nano: a Highly Compact You Only Look Once Convolutional Neural Network for Object Detection," in 2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS), Dec. 2019, pp. 22–25. doi: 10.1109/EMC2-NIPS53020.2019.00013.

[224] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778. Accessed: Oct. 27, 2022. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

[225] C. Szegedy et al., "Going Deeper With Convolutions," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9. Accessed: Oct. 27, 2022. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deeper_With_2015_CVPR_paper.html

[226] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for Energy-Efficient Neuromorphic Computing," in Advances in Neural Information Processing Systems, 2015, vol. 28. Accessed: Oct. 27, 2022. [Online]. Available: https://proceedings.neurips.cc/paper/2015/hash/10a5ab2db37feedfdeaab192ead4ac0e-Abstract.html

[227] J. Wang, H. Bai, J. Wu, and J. Cheng, "Bayesian Automatic Model Compression," IEEE Journal of Selected Topics in Signal Processing, vol. 14, no. 4, pp. 727–736, May 2020, doi: 10.1109/JSTSP.2020.2977090.

[228] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization." arXiv, Dec. 18, 2014. doi: 10.48550/arXiv.1412.6115.

[229] R. Alvarez, R. Prabhavalkar, and A. Bakhtin, "On the efficient representation and execution of deep acoustic models." arXiv, Dec. 16, 2016. doi: 10.48550/arXiv.1607.04683.

[230] F. Wang, M. Zhang, X. Wang, X. Ma, and J. Liu, "Deep Learning for Edge Computing Applications: A State-of-the-Art Survey," IEEE Access, vol. 8, pp. 58322–58336, 2020, doi: 10.1109/ACCESS.2020.2982411.

[231] Y. Huang, X. Ma, X. Fan, J. Liu, and W. Gong, "When deep learning meets edge computing," in 2017 IEEE 25th International Conference on Network Protocols (ICNP), Oct. 2017, pp. 1–2. doi: 10.1109/ICNP.2017.8117585.

[232] A. E. Eshratifar and M. Pedram, "Energy and Performance Efficient Computation Offloading for Deep Neural Networks in a Mobile Cloud Computing Environment," in Proceedings of the 2018 on Great Lakes Symposium on VLSI, New York, NY, USA, May 2018, pp. 111–116. doi: 10.1145/3194554.3194565.

[233] C. Streiffer et al., "ePrivateeye: to the edge and beyond!," in Proceedings of the Second ACM/IEEE Symposium on Edge Computing, New York, NY, USA, Oct. 2017, pp. 1–13. doi: 10.1145/3132211.3134457.

[234] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in Proceedings of the 9th international conference on Mobile systems, applications, and services, New York, NY, USA, Jun. 2011, pp. 43–56. doi: 10.1145/1999995.2000000.

[235] P. Liu, B. Qi, and S. Banerjee, "EdgeEye: An Edge Service Framework for Real-time Intelligent Video Analytics," in Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking, New York, NY, USA, Jun. 2018, pp. 1–6. doi: 10.1145/3213344.3213345.

[236] M. Song et al., "In-Situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems," in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb. 2018, pp. 92–103. doi: 10.1109/HPCA.2018.00018.

[237] A. Morshed, P. P. Jayaraman, T. Sellis, D. Georgakopoulos, M. Villari, and R. Ranjan, "Deep Osmosis: Holistic Distributed Deep Learning in Osmotic Computing," IEEE Cloud Computing, vol. 4, no. 6, pp. 22–32, Nov. 2017, doi: 10.1109/MCC.2018.1081070.

[238] S. Ochella, M. Shafiee, and F. Dinmohammadi, "Artificial intelligence in prognostics and health management of engineering systems," Engineering Applications of Artificial Intelligence, vol. 108, p. 104552, Feb. 2022, doi: 10.1016/j.engappai.2021.104552.

[239] A. Bahrammirzaee, "A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems," Neural Comput & Applic, vol. 19, no. 8, pp. 1165–1195, Nov. 2010, doi: 10.1007/s00521-010-0362-z.

[240] A. He et al., "A Survey of Artificial Intelligence for Cognitive Radios," IEEE Transactions on Vehicular Technology, vol. 59, no. 4, pp. 1578–1592, May 2010, doi: 10.1109/TVT.2010.2043968.

[241] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," Adv Neural Inf Process Syst, vol. 2, 1989.

[242] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, "Predicting parameters in deep learning," Adv Neural Inf Process Syst, vol. 26, 2013.

[243] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," arXiv preprint arXiv:1710.01878, 2017.

[244] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," arXiv preprint arXiv:1702.03044, 2017.

[245] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," IEEE Trans Wirel Commun, vol. 19, no. 1, pp. 447–457, 2019.

[246] C.-Y. Lin, T.-C. Wang, K.-C. Chen, B.-Y. Lee, and J.-J. Kuo, "Distributed deep neural network deployment for smart devices from the edge to the cloud," in Proceedings of the ACM MobiHoc workshop on pervasive systems in the IoT era, 2019, pp. 43–48.

[247] M. Verhelst and B. Moons, "Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices," IEEE Solid-State Circuits Magazine, vol. 9, no. 4, pp. 55–65, 2017.

[248] T. H. Vu, L. Dung, and J.-C. Wang, "Transportation mode detection on mobile devices using recurrent nets," in Proceedings of the 24th ACM international conference on Multimedia, 2016, pp. 392–396.

[249] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

[250] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in Proceedings of the IEEE international conference on computer vision, 2015, pp. 2722–2730.

[251] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," Adv Neural Inf Process Syst, vol. 28, 2015.

[252] X. Yue, H. Li, Y. Fujikawa, and L. Meng, "Dynamic dataset augmentation for deep learning-based oracle bone inscriptions recognition," ACM Journal on Computing and Cultural Heritage, vol. 15, no. 4, pp. 1–20, 2022.

[253] R. Ishibashi, H. Kaneko, X. Yue, and L. Meng, "Grasp Point Calculation and Food Waste Detection for Dish-recycling Robot," in 2022 International Conference on Advanced Mechatronic Systems (ICAMechS), 2022, pp. 41–46.

[254] H. Li, Z. Wang, X. Yue, W. Wang, T. Hiroyuki, and L. Meng, "A comprehensive analysis of low-impact computations in deep learning workloads," in Proceedings of the 2021 on Great Lakes Symposium on VLSI, 2021, pp. 385–390.

[255] M. O. Lawal, "Tomato detection based on modified YOLOv3 framework," Sci Rep, vol. 11, no. 1, p. 1447, 2021.

[256] X. Zhang, Y. Wang, G. Geng, and J. Yu, "Delay-optimized multicast tree packing in software-defined networks," IEEE Trans Serv Comput, 2021.

[257] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", Accessed: Aug. 13, 2017. [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[258] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint arXiv:1510.00149, 2015.

[259] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," arXiv preprint arXiv:1710.09282, 2017.

[260] A. G. Howard et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv preprint arXiv:1704.04861, 2017.

[261] Z. Yang et al., "Deep fried convnets," in Proceedings of the IEEE international conference on computer vision, 2015, pp. 1476–1483.

[262] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," arXiv preprint arXiv:1608.08710, 2016.

[263] S. Hanson and L. Pratt, "Comparing biases for minimal network construction with back-propagation," Adv Neural Inf Process Syst, vol. 1, 1988.

[264] B. Hassibi and D. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," Adv Neural Inf Process Syst, vol. 5, 1992.

[265] Q. Xiang, X. Wang, Y. Song, L. Lei, R. Li, and J. Lai, "One-dimensional convolutional neural networks for high-resolution range profile recognition via adaptively feature recalibrating and automatically channel pruning," International Journal of Intelligent Systems, vol. 36, no. 1, pp. 332–361, 2021.

[266] J.-H. Luo, H. Zhang, H.-Y. Zhou, C.-W. Xie, J. Wu, and W. Lin, "Thinet: pruning cnn filters for a thinner net," IEEE Trans Pattern Anal Mach Intell, vol. 41, no. 10, pp. 2525–2538, 2018.

[267] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang, "Accelerating Convolutional Networks via Global \& Dynamic Filter Pruning.," in IJCAI, 2018, p. 8.

[268] J. Kuang, M. Shao, R. Wang, W. Zuo, and W. Ding, "Network pruning via probing the importance of filters," International Journal of Machine Learning and Cybernetics, vol. 13, no. 9, pp. 2403–2414, 2022.

[269] H. Li et al., "Optimizing the deep neural networks by layer-wise refined pruning and the acceleration on FPGA," Comput Intell Neurosci, vol. 2022, 2022.

[270] H. Li, X. Yue, Z. Wang, W. Wang, H. Tomiyama, and L. Meng, "A survey of Convolutional Neural Networks—From software to hardware and the applications in measurement," Measurement: Sensors, vol. 18, p. 100080, 2021.

[271] S. S. Sawant et al., "An optimal-score-based filter pruning for deep convolutional neural networks," Applied Intelligence, vol. 52, no. 15, pp. 17557–17579, 2022.

[272] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in International Conference on Machine Learning, 2020, pp. 2943–2952.

[273] Q. Huang, K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks," in 2018 IEEE Winter Conference on Applications of Computer Vision (WACV), 2018, pp. 709–718.

[274] Y. Chu, P. Li, Y. Bai, Z. Hu, Y. Chen, and J. Lu, "Group channel pruning and spatial attention distilling for object detection," Applied Intelligence, vol. 52, no. 14, pp. 16246–16264, 2022.

[275] J. Chang, Y. Lu, P. Xue, Y. Xu, and Z. Wei, "Automatic channel pruning via clustering and swarm intelligence optimization for CNN," Applied Intelligence, vol. 52, no. 15, pp. 17751–17771, 2022.

[276] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in Proceedings of the IEEE international conference on computer vision, 2017, pp. 2736–2744.

[277] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," ACM Journal on Emerging Technologies in Computing Systems (JETC), vol. 13, no. 3, pp. 1–18, 2017.

[278] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in International conference on machine learning, 2015, pp. 448–456.

[279] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 5687–5695.

[280] Y. Fan, W. Pang, and S. Lu, "HFPQ: deep neural network compression by hardware-friendly pruning-quantization," Applied Intelligence, pp. 1–13, 2021.

[281] T. Chen et al., "Only train once: A one-shot neural network training and pruning framework," Adv Neural Inf Process Syst, vol. 34, pp. 19637–19651, 2021.

[282] G. S. Chung and C. S. Won, "Filter pruning by image channel reduction in pre-trained convolutional neural networks," Multimed Tools Appl, vol. 80, pp. 30817–30826, 2021.

[283] T. Chen et al., "Linearity grafting: Relaxed neuron pruning helps certifiable robustness," in International Conference on Machine Learning, 2022, pp. 3760–3772.

[284] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 11264–11272.

[285] X. Dong, S. Chen, and S. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," Adv Neural Inf Process Syst, vol. 30, 2017.

[286] T.-J. Yang et al., "Vivienneand Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications," in The European Conference on Computer Vision. Springer International Publishing, 2018.

[287] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," Adv Neural Inf Process Syst, vol. 29, 2016.

[288] J. O. Neill, S. Dutta, and H. Assem, "Aligned weight regularizers for pruning pretrained neural networks," arXiv preprint arXiv:2204.01385, 2022.

[289] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," arXiv preprint arXiv:1803.03635, 2018.

[290] X. Yue, H. Li, and L. Meng, "An Ultralightweight Object Detection Network for Empty-Dish Recycling Robots," IEEE Trans Instrum Meas, vol. 72, pp. 1–12, 2023.

[291] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," arXiv preprint arXiv:2106.08295, 2021.

[292] Z. Li, H. Li, and L. Meng, "Model Compression for Deep Neural Networks: A Survey," Computers, vol. 12, no. 3, p. 60, 2023.

[293] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," Adv Neural Inf Process Syst, vol. 28, 2015.

[294] J. Fang, A. Shafiee, H. Abdel-Aziz, D. Thorsley, G. Georgiadis, and J. H. Hassoun, "Post-training piecewise linear quantization for deep neural networks," in Computer Vision--ECCV 2020: 16th European Conference, Glasgow, UK, August 23--28, 2020, Proceedings, Part II 16, 2020, pp. 69–86.

[295] S. Garg, J. Lou, A. Jain, Z. Guo, B. J. Shastri, and M. Nahmias, "Dynamic precision analog computing for neural networks," IEEE Journal of Selected Topics in Quantum Electronics, vol. 29, no. 2: Optical Computing, pp. 1–12, 2022.

[296] R. Ni, H. Chu, O. Castañeda, P. Chiang, C. Studer, and T. Goldstein, "Wrapnet: Neural net inference with ultra-low-resolution arithmetic," arXiv preprint arXiv:2007.13242, 2020.

[297] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," arXiv preprint arXiv:2008.05000, 2020.

[298] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 13169–13178.

[299] J. Fang, A. Shafiee, H. Abdel-Aziz, D. Thorsley, G. Georgiadis, and J. Hassoun, "Near-lossless post-training quantization of deep neural networks via a piecewise linear approximation," arXiv preprint arXiv:2002.00104, vol. 10, pp. 973–978, 2020.

[300] R. Banner, Y. Nahshan, and D. Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," Adv Neural Inf Process Syst, vol. 32, 2019.

[301] A. Finkelstein, U. Almog, and M. Grobman, "Fighting quantization bias with bias," arXiv preprint arXiv:1906.03193, 2019.

[302] E. Meller, A. Finkelstein, U. Almog, and M. Grobman, "Same, same but different: Recovering neural network quantization error through weight factorization," in International Conference on Machine Learning, 2019, pp. 4486–4495.

[303] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1325–1334.

[304] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), 2019, pp. 3009–3018.

[305] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," in International conference on machine learning, 2019, pp. 7543–7552.

[306] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in International Conference on Machine Learning, 2020, pp. 7197–7206.

[307] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving post training neural quantization: Layer-wise calibration and integer programming," arXiv preprint arXiv:2006.10518, 2020.

[308] H. Li, Z. Wang, X. Yue, W. Wang, H. Tomiyama, and L. Meng, “An architecture-level analysis on deep learning models for low-impact computations,” Artif Intell Rev, vol. 56, no. 3, pp. 1971–2010, 2023.

[309] S. Lin, R. Ji, C. Chen, D. Tao, and J. Luo, “Holistic cnn compression via low-rank decomposition with knowledge transfer,” IEEE Trans Pattern Anal Mach Intell, vol. 41, no. 12, pp. 2889–2905, 2018.

[310] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, “Learning separable filters,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2013, pp. 2754–2761.

[311] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” Adv Neural Inf Process Syst, vol. 27, 2014.

[312] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, “Low-rank matrix factorization for deep neural network training with high-dimensional output targets,” in 2013 IEEE international conference on acoustics, speech and signal processing, 2013, pp. 6655–6659.

[313] Y. Lu, A. Kumar, S. Zhai, Y. Cheng, T. Javidi, and R. Feris, “Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 5334–5343.

[314] M. Jaderberg, A. Vedaldi, and A. Zisserman, “Speeding up convolutional neural networks with low rank expansions,” arXiv preprint arXiv:1405.3866, 2014.

[315] S. Swaminathan, D. Garg, R. Kannan, and F. Andres, “Sparse low rank factorization for deep neural network compression,” Neurocomputing, vol. 398, pp. 185–196, 2020.

[316] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” arXiv preprint arXiv:1503.02531, 2015.

[317] R. Tang, Y. Lu, L. Liu, L. Mou, O. Vechtomova, and J. Lin, “Distilling task-specific knowledge from bert into simple neural networks,” arXiv preprint arXiv:1903.12136, 2019.

[318] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter,” arXiv preprint arXiv:1910.01108, 2019.

[319] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, “Well-read students learn better: On the importance of pre-training compact models,” arXiv preprint arXiv:1908.08962, 2019.

[320] S. Sun, Y. Cheng, Z. Gan, and J. Liu, “Patient knowledge distillation for bert model compression,” arXiv preprint arXiv:1908.09355, 2019.

[321] C. Xu and J. McAuley, “A survey on model compression and acceleration for pretrained language models,” in Proceedings of the AAAI Conference on Artificial Intelligence, 2023, pp. 10566–10575.

[322] G. Aguilar, Y. Ling, Y. Zhang, B. Yao, X. Fan, and C. Guo, “Knowledge distillation from internal representations,” in Proceedings of the AAAI Conference on Artificial Intelligence, 2020, pp. 7350–7357.

[323] X. Jiao et al., “Tinybert: Distilling bert for natural language understanding,” arXiv preprint arXiv:1909.10351, 2019.

[324] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, “Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers,” Adv Neural Inf Process Syst, vol. 33, pp. 5776–5788, 2020.

[325] W. Wang, H. Bao, S. Huang, L. Dong, and F. Wei, “Minilmv2: Multi-head self-attention relation distillation for compressing pretrained transformers,” arXiv preprint arXiv:2012.15828, 2020.

[326] C. Wu, F. Wu, and Y. Huang, “One teacher is enough? pre-trained language model distillation from multiple teachers,” arXiv preprint arXiv:2106.01023, 2021.

[327] L. Hou, Z. Huang, L. Shang, X. Jiang, X. Chen, and Q. Liu, “Dynabert: Dynamic bert with adaptive width and depth,” Adv Neural Inf Process Syst, vol. 33, pp. 9782–9793, 2020.

[328] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, “Mobilebert: a compact task-agnostic bert for resource-limited devices,” arXiv preprint arXiv:2004.02984, 2020.

[329] C. Liu, C. Tao, J. Feng, and D. Zhao, “Multi-granularity structural knowledge distillation for language model compression,” in Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 1001–1011.

[330] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size,” arXiv preprint arXiv:1602.07360, 2016.

[331] Y. Gao et al., “Disco: Remedy self-supervised learning on lightweight models with distilled contrastive learning,” arXiv preprint arXiv:2104.09124, 2021.

[332] M. Tan et al., “Mnasnet: Platform-aware neural architecture search for mobile,” in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 2820–2828.

[333] G. Huang, S. Liu, L. der Maaten, and K. Q. Weinberger, “Condensenet: An efficient densenet using learned group convolutions,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 2752–2761.

[334] S. Mehta, M. Rastegari, A. Caspi, L. Shapiro, and H. Hajishirzi, “ESPNet: efficient spatial pyramid of dilated convolutions for semantic segmentation. Comput Sci.” 2018.

[335] S. Mehta, M. Rastegari, L. Shapiro, and H. Hajishirzi, “Espnetv2: A light-weight, power efficient, and general purpose convolutional neural network,” in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 9190–9200.

[336] H. Gao, Z. Wang, and S. Ji, “Channelnets: Compact and efficient convolutional neural networks via channel-wise convolutions,” Adv Neural Inf Process Syst, vol. 31, 2018.

[337] T. Zhang, G.-J. Qi, B. Xiao, and J. Wang, “Interleaved group convolutions,” in Proceedings of the IEEE international conference on computer vision, 2017, pp. 4373–4382.

[338] K. Sun, M. Li, D. Liu, and J. Wang, “Igcv3: Interleaved low-rank group convolutions for efficient deep neural networks,” arXiv preprint arXiv:1806.00178, 2018.

[339] G. Xie, J. Wang, T. Zhang, J. Lai, R. Hong, and G.-J. Qi, “Interleaved structured sparse convolutional neural networks,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8847–8856.

[340] B. Wu et al., "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 10734–10742.

[341] A. Wan et al., "Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In 2020 IEEE," in CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 12962–12971.

[342] X. Dai et al., "Fbnetv3: Joint architecture-recipe search using predictor pretraining," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 16276–16285.

[343] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in International conference on machine learning, 2019, pp. 6105–6114.

[344] K. Han, Y. Wang, and Q. Tian, "Ghostnet: More features from cheap operations [C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition," in Ghostnet: More features from cheap operations [C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020.

[345] N. Ma, X. Zhang, J. Huang, and J. Sun, "Weightnet: Revisiting the design space of weight networks," in European Conference on Computer Vision, 2020, pp. 776–792.

[346] Y. Li et al., "Micronet: Improving image recognition with extremely low flops," in Proceedings of the IEEE/CVF International conference on computer vision, 2021, pp. 468–477.