UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

CONVOLUTIONAL NEURAL NETWORK-BASED OBJECT DETECTION WITH LIMITED EMBEDDED COMPUTATIONAL RESOURCES

PROJET DE MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN SCIENCES ET TECHNOLOGIES DE L'INFORMATION

PAR

ICARO CAMELO

AOÛT 2024

Ce mémoire a été évalué par un jury composé des personnes suivantes :

Dr. Péricles De Lima Sobreira ..... Président du jury

Dre. Rokia Missaoui ..... Membre du jury

Dre. Ana-Maria Cretu ..... Directeur de recherche

Mémoire accepté le : 16 Juillet 2024

Dedication

I dedicate this thesis to my wife Anndreza, and children Michael and Melissa, for their understanding and love throughout this journey. To my grandfather Jose Augusto Camelo (in memoriam) and father Augusto Camelo (in memoriam).

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Ana-Maria Cretu, for her invaluable guidance, encouragement, and support throughout the course of my research.

I would like to thank my family for their unwavering love, support, and understanding.

Lastly, I would like to acknowledge the support of my mother, Maria do Carmo Veras da Silva, and my mother-in-law, Maria do Socorro Barros, who have helped and supported me in various ways during the completion of this thesis. Your support has been invaluable, and I am truly grateful.

Table of Contents

List of Figures

List of Tables

List of abbreviations and acronyms

CNN Convolutional Neural Network

GPU Graphics Processing Unit

CPU Central Processing Unit

TPU Tensor Processing units

FPGA Field Programmable Gate Arrays

SENet Squeeze-and-Excitation Networks

Résumé

Les réseaux de neurones convolutifs ont été largement utilisés dans diverses applications telles que la reconnaissance d'images et de la parole, le traitement du langage naturel et les voitures autonomes. Cependant, le déploiement de modèles CNN vastes et complexes sur des appareils aux ressources limitées tels que les smartphones et les appareils Internet des objets (IoT) n'est pas réalisable en raison de leur mémoire, de leur stockage et de leur puissance de calcul limitées. Les CNN compacts, dont les modèles sont plus petits et nécessitent moins de ressources informatiques, sont apparus comme une solution à ce défi. De plus, les CNN compacts sont moins sujets au surajustement, un problème courant avec les CNN volumineux et complexes lorsque l'on travaille avec de petits ensembles de données, et peuvent améliorer l'interprétabilité des modèles.

Les CNN sont particulièrement utiles en vision par ordinateur, un domaine qui étudie comment comprendre et identifier les caractéristiques des images et des vidéos, grâce aux progrès de la conception des algorithmes, à la croissance de la puissance de calcul et à la disponibilité des données. Les CNN sont largement utilisés dans les tâches de vision par ordinateur telles que la classification d'images, la détection d'objets et la segmentation sémantique. Grâce aux améliorations des architectures CNN, elles peuvent entraîner des modèles avec des milliards de paramètres, même dans des appareils aux ressources limitées, ce qui en fait une option solide pour l'utilisation dans les images 2D et 3D.

L'objectif de cette recherche est d'implémenter et d'évaluer les performances de différents algorithmes CNN compacts sur la plateforme matérielle NVIDIA Jetson Xavier NX. L'évaluation des algorithmes CNN compacts sera menée à l'aide de mesures telles que la taille du modèle, le coût de calcul, le F-score, la précision et le rappel. Les résultats de cette recherche fourniront une comparaison complète des compromis entre AlexNet, ShuffleNet V2, SqueezeNet, ResNet50 et MobileNet V2.

Comme prévu, les modèles pré-entraînés ont obtenu de bons résultats par rapport à ceux entraînés à partir de zéro sur la plateforme Jetson Xavier, en utilisant le modèle

de base entraîné sur ImageNet. Le modèle le plus performant sur l'ensemble de données CIFAR-10 est le ShuffleNet pré-entraîné et l'AlexNet pré-entraîné sur l'ensemble de données STL-10, tous deux entraînés pendant 30 époques. Sur l'ensemble de données OrganMNIST {A,C,S} le AlexNet pré-entraîné et ensuite entraînés pendant 100 époques est le plus performant. Il convient de noter que le Jetson Xavier NX est efficace en termes d'utilisation des ressources, utilisant près de 100% du GPU et de la mémoire RAM. Les principales limitations rencontrées lors de l'entraînement des CNN sur la carte NVIDIA Xavier NX sont dues aux ressources mémoire limitées.

Abstract

Convolutional Neural Networks have been widely used in various applications such as image and speech recognition, natural language processing, and self-driving cars. However, deploying large and complex CNN models on resource-constrained devices such as smartphones and Internet of Things (IoT) devices is not feasible due to their limited memory, storage, and computational power. Compact CNNs, which have smaller model sizes and require less computational resources, have emerged as a solution to this challenge. In addition, compact CNNs are less prone to overfitting, a common issue with large, complex CNNs when working with small datasets, and can improve the interpretability of the models.

CNNs are especially helpful in computer vision, a field that studies how to understand and identify features in images and videos, thanks to the progress of algorithms design, the growth of computing power, and data availability. CNNs are widely used in computer vision tasks such as image classification, object detection, and semantic segmentation. With the improvements of the CNNs structures, it can train models with billions of parameters even in devices with limited resources making it a strong method for further development for 2D and 3D visual data.

The objective of this research is to implement and evaluate the performance of different compact CNN algorithms on the NVIDIA Jetson Xavier NX hardware platform. The evaluation of compact CNN algorithms will be conducted using metrics such as model size, computational cost, F-score, precision and recall. The results of this research will provide a comprehensive comparison of the trade-offs between AlexNet, ShuffleNet V2, SqueezeNet, ResNet50, and MobileNet V2.

As expected, the pre-trained models performed well compared to those trained from scratch on the Jetson Xavier platform, using the base model trained on ImageNet. The best performing model on the CIFAR-10 dataset was the pre-trained ShuffleNet and the pre-trained AlexNet on the STL-10 dataset, both trained for 30 epochs. On OrganMNIST

{A,C,S} dataset, AlexNet pre-trained for 100 epochs obtained the best performance. It is worth noting that the Jetson Xavier NX was efficient in terms of resource utilization, utilizing almost 100% of GPU and RAM. The primary limitations encountered when training CNNs on the NVIDIA Xavier NX board were due to limited memory resources.

Chapter 1

Introduction

1.1 Context

Nowadays, artificial intelligence (AI) has become very prominent and impactful owing to its proficiency in accomplishing a wide variety of tasks with high levels of effectiveness and efficiency. Some of the areas where AI has demonstrated its capabilities include, but are not restricted to, visual recognition tasks like image classification, object detection, sensor data and natural language processing. Deep learning is an advanced sub-discipline of machine learning that emphasizes refining artificial neural networks with multiple layers to apprehend intricate representations of data. It can learn useful features from raw data without manual feature engineering. In contrast, the advent of Internet of Things devices having inbuilt sensors opens novel prospects for implementing convolutional neural networks (CNNs) directly on resource-limited devices.

1.2 Classical vs. compact CNN models

There are basically two approaches for choosing CNNs depending on the task and the computational resources available. The classical CNN models are powerful but are also resource-intensive whereas compact CNN models are lighter without compromising performance. The growing demand for CNN deployments on resource-constrained devices, including smartphones, Remotely Piloted Aircraft System (RPAS), and IoT devices, is one key reason for the utilization of compact convolutional neural networks since a large, complex CNN cannot run efficiently on these devices due to their limited memory, sto-

rage, and computational power. These devices are more suitable for deploying compact CNNs, which have smaller model sizes and require less computational resources. Compact CNNs are designed to address several challenges associated with traditional CNNs such as : overfitting, and the high computational cost and memory requirements, as we briefly discuss in the following paragraphs.

Classic, large, and complex CNNs are also prone to overfitting, particularly when working with small datasets. Compact CNNs, with smaller model sizes and fewer parameters, are less prone to overfitting and can achieve good performance even with limited data. Compact CNNs help to reduce complexity, improve efficiency, and limit the risk of overfitting in CNNs by reducing the amount of parameters and computation in the network.

Having fewer parameters makes it easier to figure out what the model is doing or, in other words, it improves its interpretability. Often, a compact CNN reduces the number of parameters compared to a larger model, which allows one to better interpret the results.

Implementing compact CNNs with smaller models and computational needs on IoT devices enables localized capabilities like object recognition without relying on the cloud. This reduces latency while improving privacy and reliability. To facilitate model training and inference, several types of specialized hardware have emerged such as CPUs, graphics processing units (GPUs)/ tensor processing units (TPUs), and field-programmable gate arrays (FPGAs).

Researchers have been investigating the training and inference performance of models in resource-constrained devices. Ajit et al. [1] provide a broader review of CNNs without directly addressing the impact of training using different hardware. Nevertheless, the paper offers valuable context regarding the algorithmic steps and applications of CNNs across various fields. Recent studies [2] indicate that both GPU and TPU significantly improved the performance and accuracy of CNN models, with TPU outperforming GPU in certain cases. This suggests that the choice of hardware can have an important impact on model performance and overall performance. Other work focuses on GPU and TPU deployment for image classification tasks [3]. Only a few papers explore the use of the NVIDIA Jetson Xavier NX platform for deploying imaging applications. Among these, Jabłoński et al. [4] evaluate the performance of Jetson Xavier NX for real-time image processing for plasma diagnostics. The authors implement several image processing algorithms on the platform and evaluate their performance in terms of speed and

accuracy. They found that the platform is able to achieve good performance on the image processing tasks, and that it is well-suited for real-time applications due to its fast-processing speeds. Kortli et al. [5] propose a hybrid model that combines a CNN with a long short-term memory (LSTM) network for lane detection and implement and demonstrate the ability to achieve good performance for this task on the Jetson Xavier NX.

1.3 Objectives

The goal of this thesis is to evaluate various compact CNN architectures, such as AlexNet, ShuffleNet, SqueezeNet, Resnet50, and MobileNetV2, for object recognition in 2D and 3D data trained on the NVIDIA Jetson Xavier NX. The key objectives are to analyze resource usage such as CPU/GPU and RAM used to train models, the performance of the CNNs, identify trade-offs, and find optimized deep learning solutions tailored for training and real-time inferencing on devices with tight resource constraints. In particular, the thesis aims to achieve these objectives pursuing the following steps :

1. Setting up NVIDIA Jetson Xavier NX board : Install the operational system and prepare the external hard drive to accommodate future training of CNNs.
2. Deploying CNNs on the board : Install the necessary libraries and software to download and train CNN models, as well as prepare code scripts to monitor the models during training.
3. Training CNNs on the board : Code experiments leveraging CNN models on NVIDIA Xavier NX, including downloading datasets, as well as the code needed for assessing the models performance trained from scratch and pre-trained.
4. Evaluating CNN performance : The performance of each CNN architecture will be evaluated in order to identify the strengths and weaknesses of each architecture and how they perform in object recognition tasks on various image datasets.
5. Analyzing resource usage : The research will analyze the resource usage such as CPU/GPU and RAM used to train models. This will help in identifying the resource requirements of each CNN architecture and the trade-offs involved in using them.

6. Identifying trade-offs : The research will identify the trade-offs involved in using each CNN architecture aiming to select the best architecture for object recognition on the available resources and performance requirements.

Overall, this research aims to provide insights into the performance and resource requirements of various compact CNN architectures and identify optimized deep learning solutions for object recognition tasks on devices with tight resource constraints.

The thesis is organized as follows :

- Chapter 1 introduces the research problem, research questions, objectives, and an overview of the thesis structure.
- Chapter 2 presents the literature review going through the relevant papers in the literature related to deep learning use cases in resource-constraint devices, focusing on the compact convolutional network architectures and their applications, the optimization techniques used to improve model size and performance, and the hardware platforms for deep learning
- Chapter 3 describes the methodology, the design, data collection and analysis methods.
- Chapter 4 provides a summary of the research findings and contributions, limitations, and future work.
- Chapter 5 presents the conclusions of this work.

Chapter 2

Literature review

This chapter provides an introduction to deep learning and CNNs, presents briefly some relevant architectures and techniques, as well as discusses the relevant papers in the literature related to deep learning use cases using resource-constraint devices. The focus is to identify the architectures used in this work, the hardware platform chosen, and to present the benchmarks captured.

2.1 Introduction to deep learning

Deep learning is an advanced sub-discipline of machine learning that emphasizes on refining artificial neural networks with multiple layers to apprehend intricate representations of data. It has gained significant prominence in contemporary times due to its proficiency in attaining first-rate performance across a diverse spectrum of tasks, including but not limited to image and speech recognition, natural language processing, and game playing.

Deep learning is important because it can learn useful patterns from raw data without manual feature engineering. This makes it a great choice for complex tasks, like image recognition where the structure of the data is hard to capture using traditional methods. In this case, deep learning models can recognize patterns and shapes in images (like edges and curves) to make predictions with accuracy.

Neural networks mimic the human brain expressing its behaviour through several algorithms. A neural network is composed of several layers such as : an input layer, hidden layer(s), and an output layer.

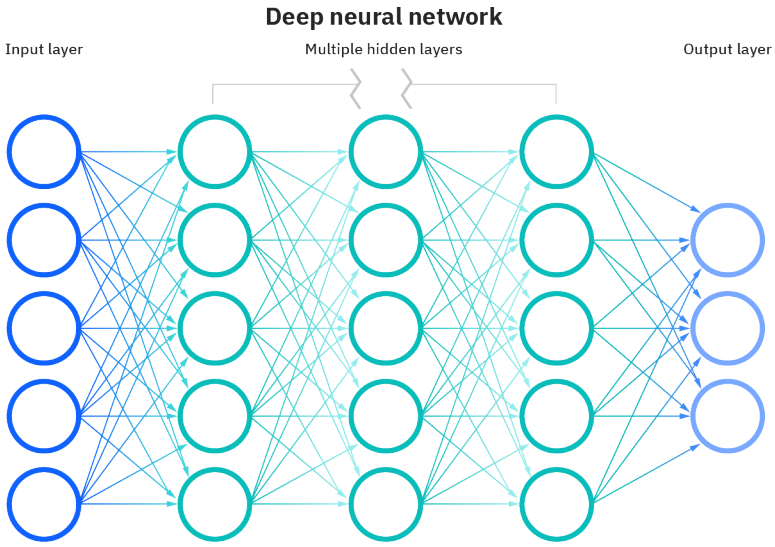

FIGURE 2.1 – Deep Neural Network (DNN) [1]

Deep learning poses significant challenges despite its advantages. Among these hurdles is the requirement for substantial quantities of labeled data to train models optimally due to deep learning algorithms consisting of millions of parameters that necessitate tuning and subsequently, massive datasets. Moreover, training deep learning models can be computationally expensive because it requires specialized hardware such as graphics processing units (GPUs) or tensor processing units (TPUs).

2.2 Convolutional Neural Networks Architectures

2.2.1 The classical CNN Architecture

Convolutional Neural Networks are a specialized type of Deep Neural Networks designed for image recognition tasks whereas DNNs are general-purpose, being used in several types of tasks, such as image and speech recognition, and natural language processing. Even though CNNs and DNNs can be used for image recognition tasks, CNNs include a few specific layers such as pooling layers, convolutional layers, batch normalization, and fully connected layers, as shown in 2.2, as well as and activation layers.

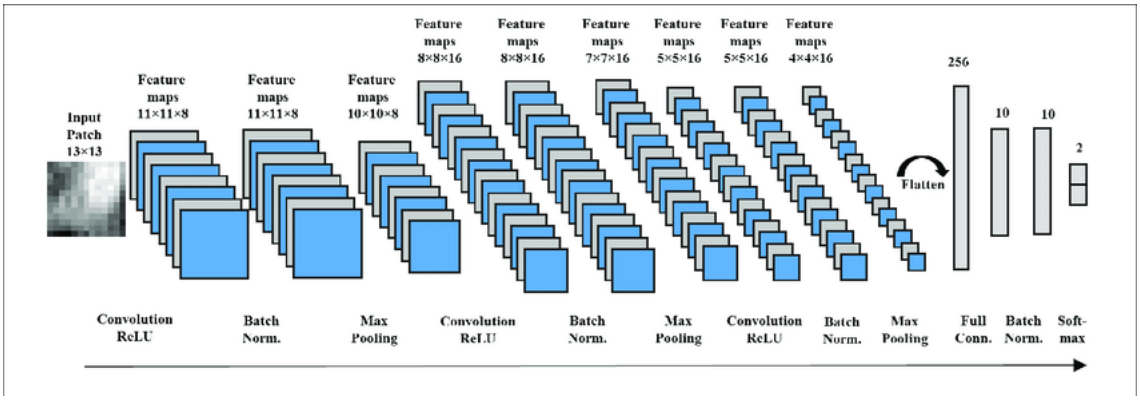

FIGURE 2.2 – Convolutional Neural Networks layers [2]

Convolutional layers serve as the foundation, accommodating the obtained kernels (filters) that assume the main function of extracting distinctive features. During a convolution operation, a kernel swipes the input image as a sliding window shifting a given number of pixels at a time. The number of pixels a kernel is shifted is defined by a hyper parameter called stride. The stride defines the pace of the sliding window, for example, a stride of '1' will make the sliding window shift one pixel at a time. An example is shown in Fig. 2.3. A stride of '2', shifts two pixels at a time, and so forth. The size of the kernels and stride are defined in each network. Also, layers are connected to one another and each connection represents a distinct kernel, utilized in the convolution operation to generate the neuron's output or feature map.

As per learning, CNNs extract features through convolution and pooling layers. The train of filters/kernels happen through backpropagation which adjusts the filter weights and minimize errors by leveraging optimization algorithms such as gradient descent. Backpropagation is a key element of the CNN learning process as it calculates and minimizes errors by leveraging optimization algorithms such as gradient descent and adjusts parameters by updating filter weights to improve predictions.

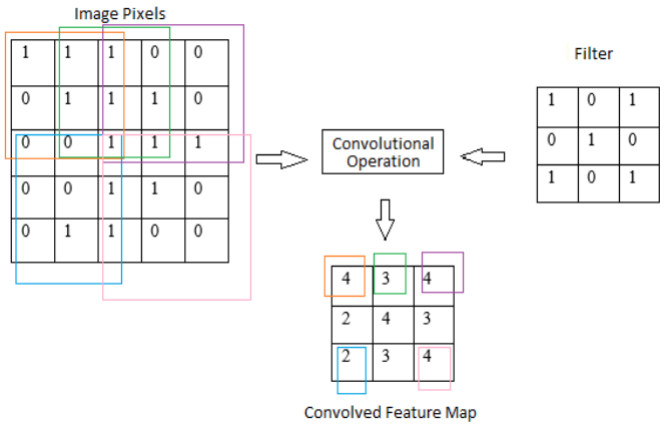

FIGURE 2.3 – Convolution operation. [3]

The goal of a pooling layer is to gradually reduce the spatial size of the network. This reduction helps to decrease the number of parameters and the overall computational cost on the network. Various CNN architectures utilize different types of pooling layers such as max pooling and average pooling. For example, max pooling calculates the maximum value for each patch on the feature map, as shown in Fig. 2.4.

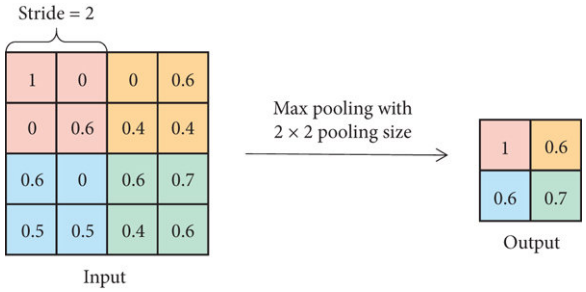

FIGURE 2.4 – Max pooling [4]

During the training process, the model sees the dataset multiple times in a number of epochs. However, it is trained on multiple chunks of the dataset instead of the entire dataset. Each individual chunk is called a batch. The main objective of the training process is to reduce the loss. Meaning, on each epoch, the model finds the values of the model's parameters in order to make the loss as small as possible through backpropagation.

The batch normalization layer reduces input values to the range of '0' and standard deviation to '1' which target overfitting issues making the network to generalize better, and speed up the training process. It is applied during the training phase using mean and

standard deviation within each mini-batch, instead of in the input layer. The activation layer is responsible for adding non-linearity to the network. This is particularly important because it makes the network learn more complex representations. Without non-linearity, neural networks would behave as though they have a single perceptron as the sum of multiple linear functions also results in a linear function. Some examples of known activation functions are : ReLu, Tanh, and Sigmoid. Finally, the Fully connected layer is composed of multiple neurons connected to each other and takes the results of the activation and convolutional layers as input to make predictions.

The next subsections present a few well-known CNN architectures that bring specific changes in the classical CNN architecture to improve accuracy, reduce parameters, and computational complexity, including the ones that we chose for experimentation in this work.

2.2.2 AlexNet

Developed by researchers from the University of Toronto, the AlexNet [16] CNN architecture was a significant breakthrough in the field of computer vision. It achieved a top-5 error rate of 15.3% on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012, which was more than 10.8 percentage points lower than the previous state-of-the-art model. The AlexNet architecture has been used as a foundation for many subsequent CNN architectures, and it continues to be a valuable tool for computer vision tasks. The AlexNet architecture has eight layers with learnable parameters. The model consists of five convolutional layers with a combination of max pooling layers followed by three fully connected layers. The ReLU activation function

$$Relu(z) = max(0, z)$$

is used in each of these layers except the output layer.

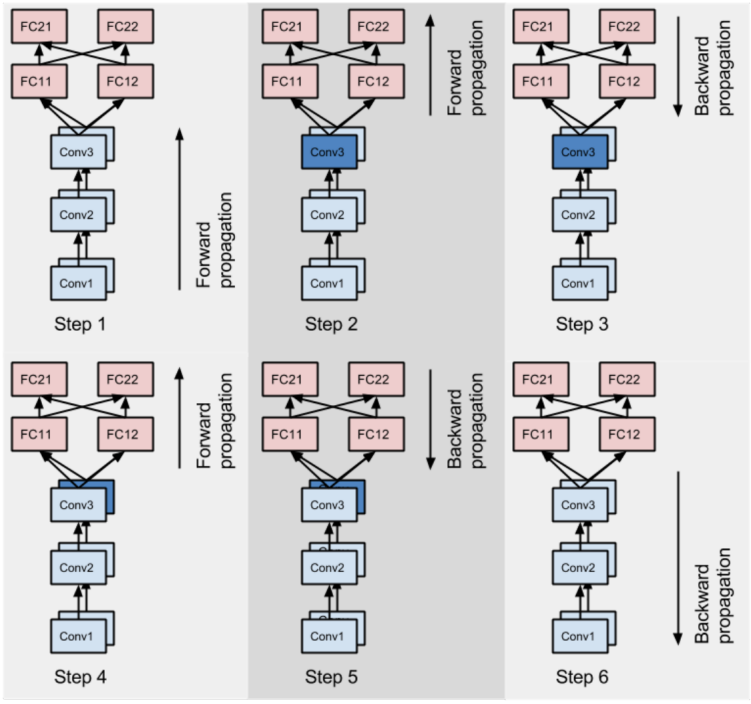

FIGURE 2.5 – AlexNet architecture from Pytorch [5]

The researchers found that using the ReLU activation function accelerated the speed of the training process by almost six times. They also used dropout layers, which prevented their model from overfitting. The model was trained on the ImageNet [17] dataset, which has almost 14 million images across a thousand classes.

The AlexNet architecture has had a significant impact on the field of computer vision. It has been used to achieve state-of-the-art results on a variety of tasks, including image classification, object detection, and scene segmentation.

2.2.3 ResNet

ResNet, or Residual Network [6], is a groundbreaking Convolutional Neural Network architecture introduced in 2015 that uses residual connections to improve training of deep neural networks. It addressed the challenge of training deep networks by employing skip connections, allowing gradients to flow directly to earlier layers. ResNet's key components include residual blocks, stacked together to form the network, and a bottleneck design for deeper versions. This architecture enabled the training of extremely deep networks, up to 152 layers, as seen in Fig. 2.6 :

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				

FIGURE 2.6 – ResNet architectures : 18, 34, 50, 101, 152 layers [6]

2.2.4 MobileNets

MobileNets [7] CNNs architecture is a class of convolutional neural networks that have been specifically designed for mobile and embedded vision applications. These CNNs are highly efficient and lightweight, making them suitable for deployment on low-power devices such as mobile phones, drones, and security cameras. Google developed this architecture based on depth-wise and point-wise separable convolutions, as illustrated in Fig. 2.9 that allow for a reduction in the number of parameters and computations required by the network.

Depth-wise separable convolution splits the standard convolution into two steps that are more efficient. A depth-wise convolution applies one single filter for each input channel reducing the amount of computations as compared to the conventional convolution whereas a point-wise convolution is performed to combine the outputs of the depth-wise convolution utilizing a simple 1x1 convolution across channels, as demonstrated in Fig. 2.7.

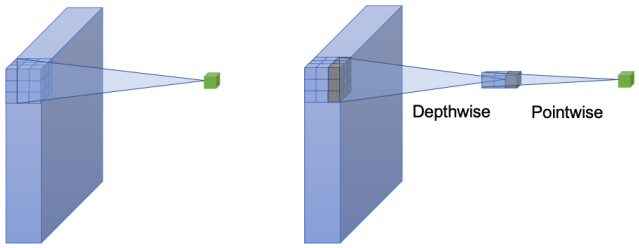

FIGURE 2.7 – Depth-wise and point-wise convolutions [7]

By dividing the convolution process into two, the model requires less computation power for training with a slight decrease in accuracy. In this architecture, each convolution is followed by a batch normalization [18] and an activation function ReLu. In its second version, MobileNetV2, introduced in 2018 by Sandler et al [19] the authors introduce the concept of "inverted residual blocks".

Residual blocks establish a connection between the beginning and the end of a convolutional block by skipping connections. Through these two stages, the network is capable of accessing earlier activations that have not been altered within the convolutional block. MobileNetV2 broadens the network by applying a 1x1 convolution, followed by a 3x3 depth-wise convolution, and subsequently appending a 1x1 convolution to align with the initial count of channels, as shown in Fig. 2.8.

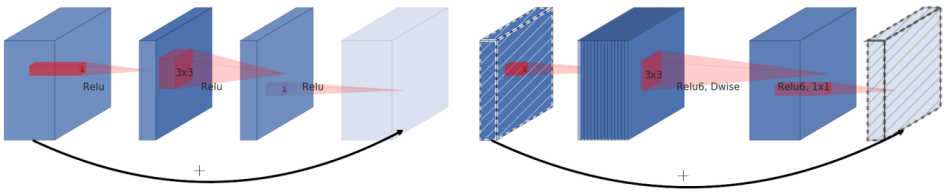

FIGURE 2.8 – Residual and inverted blocks [7]

This is a modification of the standard residual block found in architectures like ResNet. Instead of initially restricting the input with 1x1 convolutions (as in the original residual blocks), inverted residual blocks first expand the input to a higher dimension, apply a lightweight depth-wise convolution, and then project it back to a lower dimension. This streamlined architecture achieves greater efficiency and ease of training, which ultimately leads to improved performance on a variety of computer vision tasks.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

FIGURE 2.9 – MobileNet V2 architecture. Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size. [7]

2.2.5 ShuffleNet

ShuffleNet [8] is a convolutional neural network architecture that has been designed to be highly efficient and lightweight for mobile and embedded vision applications. The architecture is based on a shuffle operation, as illustrated in Fig. 2.10, that enables channel interleaving, thereby reducing the number of computations required by the network. ShuffleNet was introduced by Xiangyu Zhang and others in a paper titled "ShuffleNet : An Extremely Efficient Convolutional Neural Network for Mobile Devices." The primary goal of ShuffleNet is to enable efficient and accurate computer vision on mobile devices with limited computational resources. The network achieves this by using a shuffle operation that interleaves channels to reduce computations while still maintaining a high accuracy. This architecture is particularly useful in applications where the model needs to be deployed on low-power devices with limited computational resources. Its architecture is composed of 50 layers and incorporates two operations, point-wise group convolution and channel shuffle, which significantly reduce computational costs while still preserving accuracy.

The key idea behind channel shuffling is to enable the flow of information between channel groups in order to combine their capabilities, while keeping computational com-

plexity low. By allowing cross-channel information sharing, the network can better utilize its resources and potentially learn improved representations of the data as it has early access to combined information from different channels.

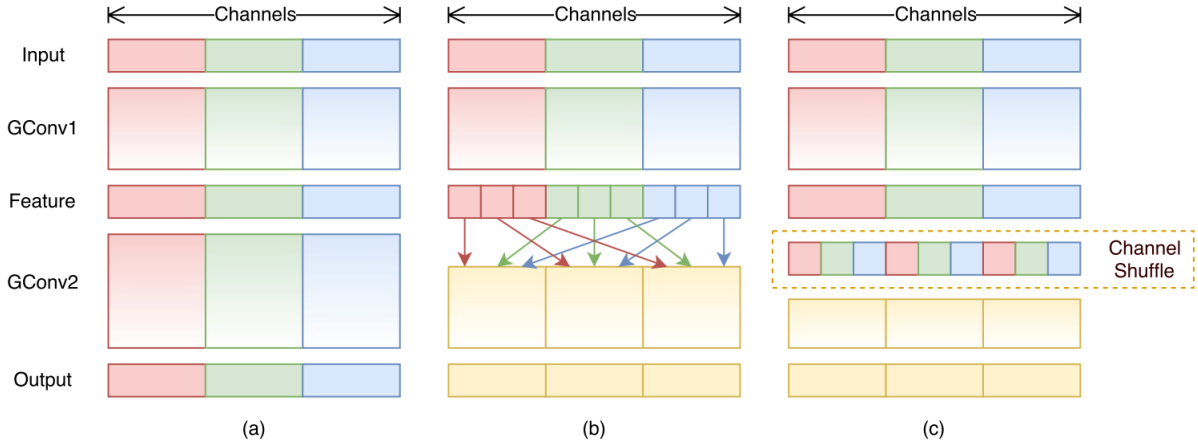

FIGURE 2.10 – Channel shuffle [8]

2.2.6 CondenseNet

CondenseNet [9] like other efficient CNN models, such as MobileNets and ShuffleNet, proposes a new method for making dense CNNs more efficient. CondenseNet has been designed to be suitable for use on mobile and embedded devices with limited computational resources. The architecture achieves this by using a combination of dense blocks and learned group convolutions as shown in Fig. 2.11. Dense blocks are blocks of densely connected layers, while learned group convolutions reduce the number of parameters and computational complexity of the network. Unlike traditional architectures, CondenseNet incorporates the concepts of "growth" and "compression" to adaptively determine the number of layers based on the complexity of the input data and available resources. The growth rate parameter controls the addition of new feature maps at each layer, allowing the network to expand or contract as needed. This technique allows for the creation of a highly efficient architecture while still maintaining high accuracy. As per compression, it applies a pruning technique in order to reduce the superfluous connections and reduce the network size without compromising accuracy.

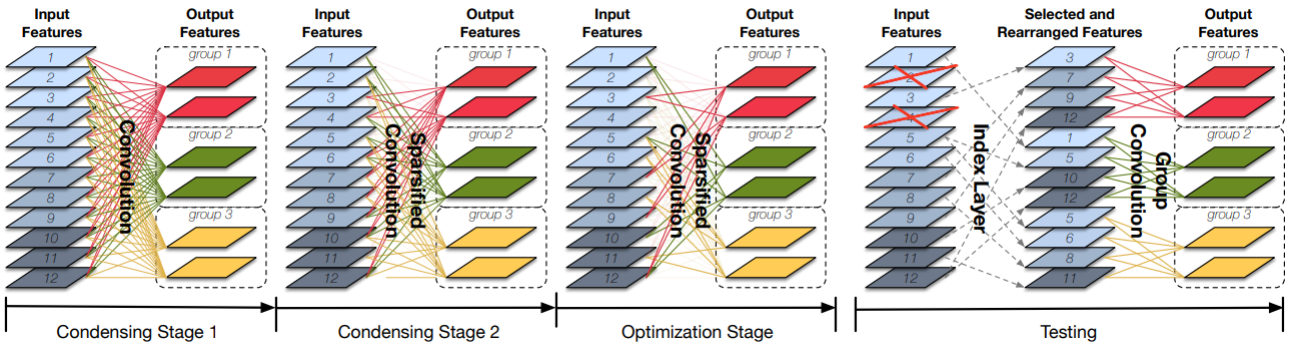

FIGURE 2.11 – CondenseNet - dense connectivity with learned group convolution [9]

The CondenseNet architecture has been shown to achieve state-of-the-art performance on a range of computer vision tasks, including image classification, object detection, and segmentation.

2.2.7 Squeeze-and-Excitation Networks (SENet)

SENet is a type of CNN architecture that was introduced in the paper "Squeeze-and-Excitation Networks" [10] published by Jie Hu, Li Shen, and Gang Sun. This CNN model uses a squeeze-and-excitation block that adaptively recalibrates the channels of the intermediate feature maps, improving the accuracy of the network. The main idea behind SENet is to allow the network to weight the importance of different channels in a feature map adaptively, rather than using a fixed set of weights. This is achieved by introducing a new block called a "squeeze-and-excitation" (SE) block that allows the network to re-calibrate the feature maps by considering inter-dependencies between channels as shown in Fig. 2.12. The SE block consists of two steps : "squeeze," which reduces the dimensionality of the feature maps by taking the global average pooling of the channels, and "excitation," which uses this reduced representation to adapt the weight of the channels. By adding SE blocks to the network architecture, the network is able to learn more features and improve the performance on various tasks. SENet can be used in any application where CNNs are used, such as image classification, object detection, and segmentation.

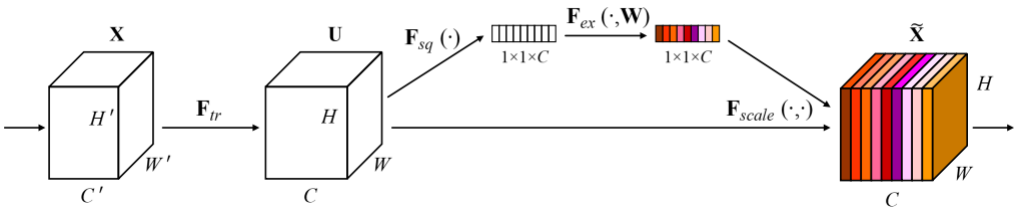

FIGURE 2.12 – Squeeze-and-Excitation (SE) block [10]

2.2.8 SqueezeNet

The SqueezeNet [11] convolutional neural network architecture illustrated in Figure 2.13 was designed to attain high accuracy with an extremely compact model size. Several improvements enable SqueezeNet reduction in number of parameters. It introduces the concept of a Fire module, shown in Figure 2.14, which is comprised of two components : a squeeze convolution layer and an expand layer. The squeeze layer employs 1x1 filters, whereas the expand layer integrates a combination of 1x1 and 3x3 convolution filters to perform channel-wise squeezing, which does nothing but to reduce the number of filters in the squeeze layers. Even though downsampling operations may reduce accuracy, they actually increase accuracy depending on where they are placed in the network. In SqueezeNet, downsampling is intentionally postponed to later stages so convolutional layers can first run on larger activation maps, enabling greater accuracy with fewer filters, and model compression techniques such as quantizing weights to 8-bits lower precision shrink the model's disk footprint.

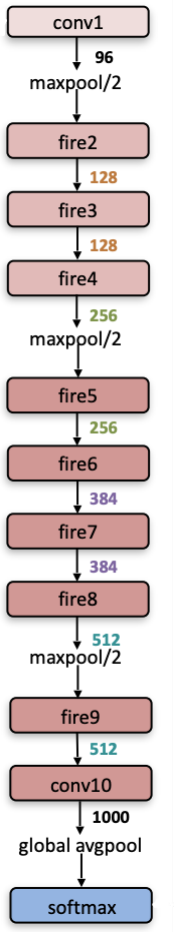

FIGURE 2.13 – SqueezeNet V2 architecture [11]

The original SqueezeNet contains around 1.2M parameters, matching AlexNet's (60 million parameters) accuracy on ImageNet with 50 times fewer parameters. This dramatic reduction makes SqueezeNet ideally suited for embedded and mobile applications where minimal model size is critical.

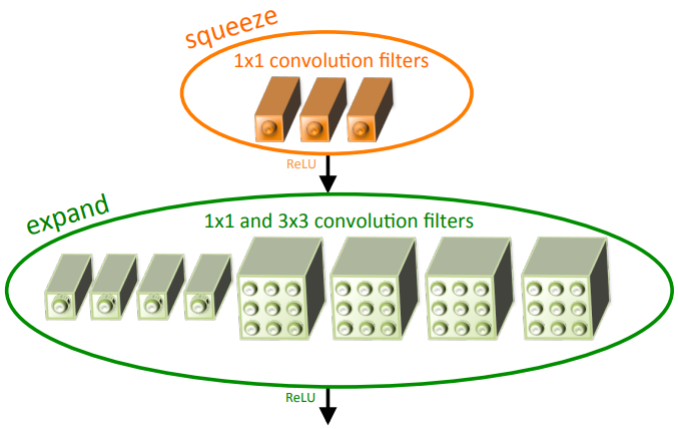

FIGURE 2.14 – Fire module : squeeze and expand layers [11]

2.2.9 EfficientNet

EfficientNet [12] is a CNN model introduced in a paper titled "EfficientNet : Rethinking Model Scaling for Convolutional Neural Networks" by Mingxing Tan and others. It uses a combination of scaling techniques to improve accuracy, reduce parameters, and computational complexity. EfficientNet employs network scaling and compound scaling to adjust the dimensions of the network, including depth, width, and resolution, to better fit input data. By leveraging network scaling, it adjusts the size and complexity of the model by changing the number of layers, the number of neurons in each layer, and the size of the input image, whereas compound scaling, simultaneously adjusts the width, depth, and resolution of the model.

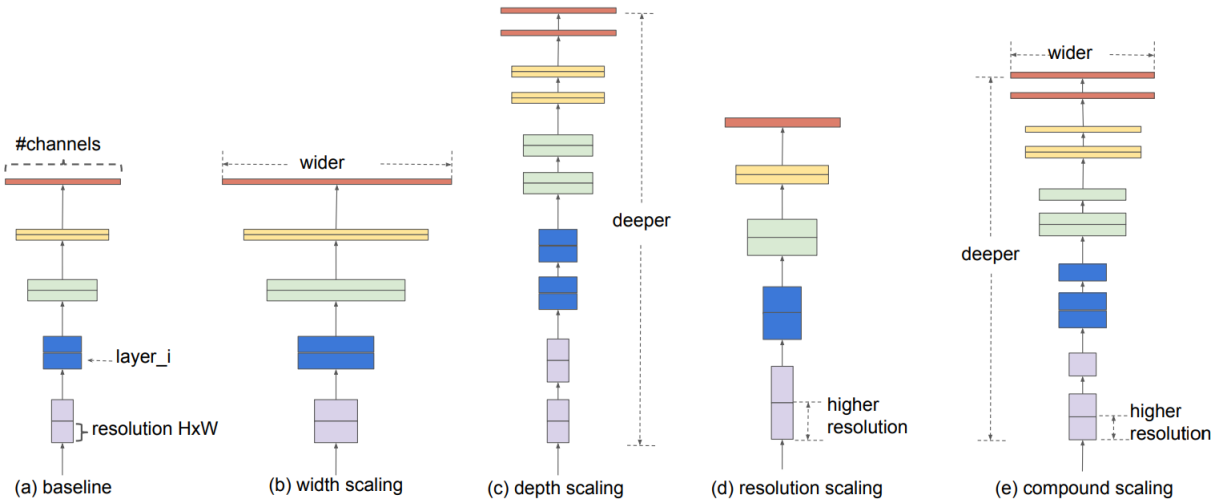

FIGURE 2.15 – EfficientNet compound scaling [12]

This is done to improve the performance, efficiency, or resource utilization of the model allowing the model to achieve state-of-the-art performance on various benchmarks with fewer parameters and computation compared to other models as shown in Figure 2.15.

EfficientNet has been widely used in image and video recognition, natural language processing, and speech recognition, as well as in mobile and embedded vision applications due to its efficiency. It has achieved top performance on tasks like image classification on ImageNet, object detection, and segmentation on COCO dataset [20]. EfficientNet is particularly useful for resource-constrained devices like mobile phones or edge devices where efficiency is crucial.

2.2.10 Other architectures

The VGG was introduced by [21] and stands for Visual Geometry Group. It consists of a standard deep CNN with multiple layers and has multiple versions, such as VGG-11, VGG-13, VGG-16 (total of 138 million and 200 million parameters) or VGG-19, depending on how many layers it has. It receives as input an image size of 224x224. Although it appears to be a complex network, given the number of parameters aforementioned, it is rather a simple and uniform one. Unlike other models, VGG employs stacked convolutional and pooling layers. It's composed of several kernels with depth ranging from 64

up to 512, followed by a Max pooling layer, ending with fully connected layers, and a Softmax activation function.

SSD, short for Single Shot Multibox Detector [22], is a CNN model based on VGG-16 introduced in 2016 by Liu et al. It replaces the fully connected layers present in VGG-16 with auxiliary convolutional layers to extract features at different scales and reduce input size. It also makes use of bounding boxes, typically present in the Region-Convolutional Neural Network (R-CNN) outputs [23]. By applying additional convolutions on top of the base VGG-16 network and the extra layers, SSD is able to capture features at multiple scales. This design enhances efficiency and allows detection of objects with different sizes and shapes. By using anchor boxes and predicting offsets and confidences, SSD accurately locates and classifies objects.

The YOLO (You only look once) model [24] is an object detection CNN model that stands out for its real-time processing capabilities, detecting objects in images or videos in a single pass through the network. It was introduced in 2015 and uses a CNN architecture that divides the image into a grid and predicts bounding boxes and object classes, similar to SSD, for each grid cell. Unlike other CNN models that perform object detection on various regions, YOLO only goes through the entire image once, hence its name. By doing so, it is able to be faster and avoid inefficient and resource-consuming operations. YOLO is known for its speed and accuracy, making it popular for real-time applications such as autonomous driving, surveillance, and robotics. It has been widely used and improved upon in various versions.

All these models can potentially be adapted to work with both 2D and 3D data, as well as used in tasks such as object recognition, object detection, instance segmentation, among others.

2.3 Optimization Techniques

Due to the large size and intricate nature of models, alongside the extensive resources required to train Convolutional Neural Network (CNN) models, experts have also developed optimization strategies to enhance the efficiency of both model training and model inference on resource-limited devices. By utilizing these techniques either independently or in conjunction with each other, significant model compression can be achieved, resulting in reduced complexity and fewer parameters used during model training. As a

consequence, the overall performance of the models can be vastly improved, making them more accessible and adaptable to a broader range of applications.

Firstly, the Pruning approach proposed by Ye et al. [25] proposes a channel pruning technique that simplifies the channel-to-channel computation graph of a CNN (connections between the channels of the input and output of each layer) resulting in a compact model that can be quickly fine-tuned. It suggests that dense neural networks contain sparse sub-networks that contain only a small number of active connections between neurons and can be trained to achieve similar performance using the same number of iterations as the original network. The goal of this technique is to reduce the number of parameters by eliminating redundant or low impact weights in order to reduce model size without compromising the model accuracy.

Secondly, the quantization technique developed by Micikevicius et al., [26] combines the use of lower-precision arithmetic for forward and backward propagation with higher-precision arithmetic for weight updates. This approach allows models to maintain accuracy while reducing memory and computation demands leveraging the fact that real-world data often exhibit low-rank structure, meaning that the weight matrices can be approximated by a product of two smaller matrices, resulting in a reduction in the number of models parameters and improving their efficiency.

Denton et al., [27] introduces a method known as low-rank factorization, which aims to approximate the initial weight matrices of a neural network by employing lower-rank matrices. This technique effectively reduces the quantity of parameters and computational complexity. This reasoning is based on the observation that numerous weights in a neural network are superfluous or exhibit a high degree of correlation, and thus can be expressed using a reduced set of parameters. Consequently, this strategy permits the model to preserve a substantial portion of its expressive capacity while employing a reduced number of parameters.

In addition, Hinton et al., [28] defines knowledge distillation as a model compression technique that involves training a smaller student network to mimic the behavior of a larger, pre-trained teacher network. The student network learns from the teacher network's output probabilities or intermediate representations, allowing it to achieve comparable performance with fewer parameters.

Last but not least, Tan and Le [12] propose a new method finding a compound scaling method that jointly scales network width, depth, and resolution, resulting in highly compact and efficient models. The traditional approach to creating larger models

involves adding layers, filters, and resolving inputs, which results in increased computation costs and resources usage. In contrast, this technique shows improved precision over contemporary CNN methods, offering a beneficial combination of high quality outputs and computational economy using fewer parameters than alternative designs.

2.4 Hardware platforms for deep learning

The increasing popularity of deep learning has led to a growing demand for hardware solutions that can handle various tasks efficiently. To facilitate training and inference, several types of specialized hardware have emerged such as CPUs (Central Processing Unit), GPUs (Graphics Processing Unit)/TPUs (Tensor Processing units), and FPGAs (Field Programmable Gate Arrays).

CPUs are general-purpose processors that are designed to handle a wide range of computing tasks. They are capable of executing complex operations and are essential components in many computing systems. When it comes to machine learning, CPUs are ideal for handling tasks that require a lot of sequential processing, such as data pre-processing, small-scale model training, and running lightweight models.

Graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are two types of hardware that are commonly used to accelerate the training of deep learning models. They have some differences in terms of their performance characteristics that can impact their suitability for different tasks. One of the main differences between GPUs and FPGAs is their level of programmability.

GPUs are highly programmable and can be used to accelerate a wide range of tasks, including deep learning. They are also widely available and are well-suited for tasks that require high-throughput and high-bandwidth processing. However, they can be relatively power-hungry, which can limit their use in certain applications, whereas TPUs are designed specifically for machine learning workloads, and they are optimized for handling large matrix multiplication operations that are commonly found in deep neural networks. TPUs are particularly well-suited for large-scale training and inference tasks and can deliver significant performance improvements over CPUs and GPUs for these types of workloads.

FPGAs, on the other hand, are less programmable than GPUs but offer a higher level of customization. They can be tailored to specific tasks, making them well-suited for tasks that require highly optimized hardware. They are also generally more power-

efficient than GPUs, which makes them well-suited for tasks that require low power consumption or that need to be performed on low-power devices. However, FPGAs can be more difficult to program than GPUs and may require specialized expertise. In terms of their performance for deep learning, TPUs and GPUs are generally faster than FPGAs for training deep learning models. This is because they are designed specifically for high-throughput and high-bandwidth processing, which makes them well-suited for tasks such as matrix operations that are commonly used in deep learning. However, FPGAs can still be a good choice for certain tasks, particularly if power efficiency is a key consideration.

The performance differences between GPUs and FPGAs depend on the specific requirements of the task at hand. Both types of hardware have their own strengths and weaknesses, and the best choice will depend on the specific needs of the application.

As per computing solutions, NVIDIA Xavier NX and AMD-Xilinx are both high-performance solutions that are designed for a range of applications, including data centers, communications, and industrial automation. However, they differ in several key ways.

The NVIDIA Xavier NX is a system-on-a-chip (SoC) developed by NVIDIA. It is designed for use in a wide range of applications, including autonomous machines, robotics, and edge computing. The Xavier NX is based on the NVIDIA Volta architecture and features a 6-core Arm Cortex-A57 processor, a 512-core NVIDIA Volta GPU, and a deep learning accelerator (DLA). It is designed to be highly energy efficient and has a small form factor, making it suitable for use in devices with limited space and power resources. The Xavier NX can deliver high performance for a range of tasks, including machine learning, image and video processing, and computer vision. It is targeted at developers and OEMs who are looking to build advanced, high-performance systems for a variety of applications.

Xilinx Ultra96v2 is an embedded field-programmable gate array (FPGA) device developed by Xilinx. It is designed to be used in a wide range of applications, including machine learning, video processing, and data acceleration. The Ultra96v2 is based on the Xilinx Zynq UltraScale+ MPSoC (multi-processing system-on-chip) architecture, which integrates a quad-core Arm Cortex-A53 processor with a Xilinx UltraScale+ FPGA. The device has a range of features, including high-bandwidth memory, programmable logic, and multiple I/O options. It is suitable for use in applications where low power consumption, high performance, and flexibility are important.

The impact of training CNNs using different hardware accelerators, such as GPU/TPU, CPU, and FPGA, on model accuracy has been widely discussed in the literature. Research has demonstrated that the choice of hardware can significantly affect the performance and accuracy of CNNs.

Ravikumar [29] conducted a study that explored the effects of GPU and TPU acceleration on the performance and accuracy of CNNs, specifically in the context of image analytics. The findings indicate that both GPU and TPU significantly improved the performance and accuracy of the models, with TPU outperforming GPU in certain cases. This suggests that the choice of hardware can be critical in achieving optimal results.

Sharma [30] corroborates these findings, observing that TPU outperformed GPU in terms of epoch time across different batch sizes. This further supports the notion that the selection of hardware accelerators can have a substantial impact on model accuracy and overall performance.

In another study, Lee [31] demonstrated that integrating existing techniques for enhancing CNN performance could lead to notable improvements in accuracy and robustness while minimizing the loss of throughput. Although the study did not directly compare the impact of different hardware accelerators, it highlights the importance of optimization techniques in maximizing model performance.

2.5 Applications of CNN on 2D and 3D data

CNNs can be trained to recognize different types of images based on the features present in the image and classify them accordingly. The most popular application of CNNs on 2D datasets is image classification. As one of the most common tasks in computer vision, this is something that is used in a wide variety of applications, from self-driving cars to medical imaging to security systems.

Object detection and object recognition applications, common applications of CNNs on 2D and 3D datasets, are two distinct types of computer vision tasks. The object recognition task processes the entire input (image or video) and identifies the object category outputting a label, whereas object recognition task takes part of the input identifying and localizing where objects are located at. It usually leverages bounding boxes to localize and label objects.

CNNs are trained to identify objects within images and can be used for multiple objects at the same time. In computer vision, this is an important task that is used in self-

driving cars, robotics, and security systems. Besides 2D datasets, CNNs can also be used to analyze 3D datasets. For example, one common application for 3D object recognition is the recognition of 3D objects, which is similar to the detection of 2D objects, but has the advantage of being modeled in three dimensions. Robotics is a field where this is an important skill that is used for applications such as autonomous navigation and grasping. An application of CNNs to 3D datasets is 3D semantic segmentation. CNNs are trained to classify each voxel in a 3D image into different semantic classes. Medical imaging relies heavily on this task and uses it for surgical planning and radiotherapy. The ability to segment medical images is crucial to the success of these procedures and a vital skill for medical doctors. This is also used for security purposes to detect weapons and bombs from security cameras. One potential field for this application is autonomous cars, where the car has to decide on what it should be doing based on the environment in front of it. It is also possible for CNNs to be applied to 3D point clouds, which are a representation of 3D data in which each point corresponds to a 3D coordinate. The use of point clouds is common in computer vision and robotics, especially for tasks like detecting objects, segmenting them semantically, and registering them. Overall, CNNs are versatile tools that can work with 2D and 3D datasets. They've been used for image classification, object detection, 3D object recognition, 3D semantic segmentation, and point clouds analysis.

Extending to RBG, RGB-D combines both color (RGB) images and depth (D) maps into one sensor package. This enables the capture of 3D point cloud data, which can provide valuable additional information compared to traditional 2D imaging systems. Some of the potential benefits are the enhancement of object detection and tracking by utilizing both visual and depth information, RGB-D cameras enable more accurate object detection and tracking in complex scenes.

Enhanced human computer interaction (HCI) area can also benefit from RGB-D images. Depth sensors allow computers to perceive hand gestures and movements at close range. Combined with machine learning techniques, this capability can open up new possibilities for immersive gaming interfaces or even control mechanisms for people with disabilities.

Researchers have been investigating the training and inference performance of models in resource-constrained devices.

Meyer [32] introduces a radar-centric automotive dataset tailored for 3D object detection. This dataset comprises high-resolution radar data and manually refined 3D ground truth data, which can be used to train and evaluate CNNs for 3D object detection tasks.

Ajit et al. [33] provides a broader review of CNNs without directly addressing the impact of training using different hardware. Nevertheless, their review offers valuable context regarding the algorithmic steps and applications of CNNs across various fields.

Köpüklü et al. [34] converts resource-efficient 2D CNNs to 3D CNNs and evaluates their performance on different benchmarks, showing that they can be utilized for real-world applications with real-time performance and considerable accuracy and memory usage. The findings of the investigation indicate that these models can be employed in a variety of practical scenarios due to their ability to deliver real-time performance while maintaining high levels of accuracy and efficient use of memory. His assessment suggests that optimizing 3D CNNs for resource efficiency requires avoiding excessively shallow or narrow designs in order to minimize complexity.

Bochkovskiy et al. [35] proposes a novel CNN architecture that achieves state-of-the-art results on the MS COCO dataset. This architecture demonstrates impressive real-time performance, with a speed of approximately 65 frames per second (FPS) when deployed on a Tesla V100 GPU. These findings underscore the potential of CNNs for efficient and effective 3D object detection when implemented on powerful hardware accelerators such as GPUs.

Simonelli et al. [36] presents a unique disentangling transformation for 2D and 3D detection losses, as well as a self-supervised confidence score for 3D bounding boxes. This approach establishes a new state-of-the-art performance benchmark for the KITTI 3D [37] Car class. The results of this study further emphasize the importance of innovative CNN architectures in enhancing the performance of 3D object detection on GPU and TPU.

Huang et al. [38] proposed the Adaptive Precision Training (APT) approach that aims to save both training energy cost and memory usage in resource constrained devices. However, it only focuses on the optimization techniques and does not get into further details on the hardware used to validate assumptions.

Liu et al. [39] proposes a ground-aware monocular 3D object detection algorithm that leverages the ground plane as additional information for depth reasoning. This approach achieves state-of-the-art performance on the KITTI 3D object detection benchmark,

once again highlighting the impact of innovative CNN architectures on the performance of 3D object detection when deployed on GPU and TPU.

Kim [40] performs a literature review on transfer learning in the context of medical image classification. The study highlights that the majority of the papers surveyed did not present any details on hardware nor training and test time performance.

Okochi et al. [41] presents the design and development of a lightweight, portable 3D spatial sensing device equipped with a compact LiDAR-type sensor. Even though the device used is based on a RaspberryPi Compute Module 4 (CM4) that has an ARM Cortex-A72 CPU (1.50 GHz, four cores) with 4 GB of RAM, the paper does not mention any training details on a resource-constrained device. The aforementioned device is used for inference only.

Only a few papers explore the use of the NVIDIA Jetson Xavier NX platform for deploying imaging applications. Among these, Jabłoński et al. [42] evaluate the performance of the NVIDIA Xavier NX platform for real-time image processing in the context of plasma diagnostics. Plasma diagnostics involves the analysis of images of plasma to understand the physical and chemical properties of the plasma. This is an important task in a variety of fields, including nuclear fusion and space propulsion. However, it can be challenging to perform real-time image processing on these images due to the large amount of data and the need for fast processing speeds. The authors investigate the use of the Xavier NX platform for real-time image processing in the context of plasma diagnostics. They implement several image processing algorithms on the platform and evaluate their performance in terms of their speed and accuracy. The authors find that the Xavier NX platform is able to achieve good performance on the image processing tasks, and that it is well-suited for real-time applications due to its fast processing speeds. They also discuss the potential benefits of using the platform for other image processing tasks in plasma diagnostics, such as image analysis and feature extraction.

Kortli et al. [43] proposes a new deep learning model for lane detection implemented on the NVIDIA Jetson Xavier NX. The authors propose a hybrid model that combines a CNN with a long short-term memory (LSTM) network. The CNN is used to extract features from the input images, while the LSTM is used to model the temporal dependencies between the images in a video sequence. This allows the model to make more accurate predictions about the lane positions over time. The authors also describe the implementation of the model on the Jetson Xavier NX, and show that it is able to achieve good performance on a benchmark dataset for lane detection. They also compare the

performance of the model to other state-of-the-art lane detection methods, and show that it is able to outperform them in some cases.

In summary, recent work shows promise for efficient object detection using innovative CNN architectures, especially when leveraging GPUs/TPUs. The potential of the Xavier NX platform has just begun to be explored for real-time imaging applications. In this context and as previously mentioned in section 1, the goal of this thesis is to evaluate various compact CNN architectures for object recognition in images trained on the NVIDIA Jetson Xavier NX. The objective is to train these models on the Xavier NX and evaluate their performance on an object recognition task in terms of accuracy, efficiency, and computational resources required. This will provide insight into the most suitable compact CNN architectures for efficient deployment on embedded devices with limited compute power. Details on the methodology are provided in the next chapter.

Chapter 3

CNN-Based 2D and 3D Object Recognition Using Nvidia Jetson Xavier NX

This chapter describes the methods and materials used in the research. It starts with a detailed walkthrough on the environment, the configuration and setup of the NVIDIA Jetson Xavier NX board, followed by an introduction to the JetPack NVIDIA Software Development Kit. It continues by describing the datasets utilized for experimentation, and finally explains how the selected CNN models are trained and assessed.

3.1 Guidelines

In the upcoming chapter, an extensive analysis of the NVIDIA Jetson Xavier NX environment will be undertaken. Our discussion starts with an in-depth walkthrough on the configuration and setup of the NVIDIA Jetson Xavier NX board, elaborating on the necessary steps essential for fully harnessing the advanced capabilities of this hardware. This chapter not only provides detailed instructions on how to set up the board, but also gives a thorough understanding of the system's architecture and potential applications.

Subsequently, the focus is shifted towards the NVIDIA JetPack Software Development Kit (SDK). This comprehensive toolkit enhances AI applications by providing robust libraries dedicated to several computational tasks, including but not limited to deep learning, computer vision, and GPU computing. An elaborate description of the

JetPack SDK's components will be provided, alongside its integration and usage with the Xavier NX platform.

3.1.1 Environment

The NVIDIA Jetson Xavier NX is a good embedded system that brings cutting-edge AI capabilities in a compact hardware. Because it has a compact size and good computational power, the Jetson Xavier NX enables developers and AI enthusiasts to develop and deploy AI applications. The Jetson Xavier NX comes with a set of components that contribute to its performance. Here is some of its main features :

- **GPU** : At its core lies an NVIDIA Volta GPU with 384 CUDA cores, delivering accelerated parallel processing capabilities for AI workloads and high-performance computing tasks.
- **Deep Learning Accelerator (DLA)** : The Jetson Xavier NX is equipped with a dedicated DLA, designed to efficiently execute deep learning inference tasks and optimize energy efficiency by offloading the workload from the GPU.
- **CPU** : A 6-core NVIDIA Carmel ARMv8.2 CPU provides high-performance computing capabilities.
- **Memory** : 8 GB of LPDDR (Low-Power Double Data Rate) 4x RAM.
- **Storage** : A microSD card slot enables expandable storage, accommodating datasets, models, and essential files required for AI development.
- **Connectivity** : The Jetson Xavier NX supports several connectivity options, including Gigabit Ethernet, USB 3.1 ports, HDMI, and DisplayPort, enabling integration with peripherals and external devices.

3.1.2 Setting Up the NVIDIA Jetson Xavier NX

The process of installing NVIDIA SDK Manager and Jetpack [44], flashing an SD card, and installing an SSD drive while changing the root file system (rootfs) to the SSD involves a series of specific procedures.

The following steps in figure 3.1 guide users through the process of setting up the Jetson Xavier NX, preparing it for AI development and deployment.

FIGURE 3.1 – NVIDIA Xavier NX configuration process

The first step is to download the NVIDIA SDK Manager from the NVIDIA SDK Manager download page. This step requires one to have an NVIDIA Developer account to access the download. Once the file has been downloaded, the terminal must be opened, and one must navigate to the directory where the file is saved. The permission of the file is then changed to make it executable with a specific command. The SDK Manager is then installed by running the following command in the terminal :

```
sudo rsync -axHAWX --numeric-ids --info=progress2 \
--exclude {"/dev/","/proc/","/sys/","/tmp/","/run/","/mnt/",
"/media/*","/lost+found"} / /mnt
```

After the SDK Manager has been installed, it can be executed by typing *'sdkmanager'* into the terminal and then logging in with the NVIDIA Developer account credentials. Within the SDK Manager, one must select the appropriate hardware configuration in the 'Target Hardware' section. Then, the desired Jetpack version is selected in the 'SDKs' section. One must then follow the prompts to complete the installation process.

Flashing the SD card is a task handled by the SDK Manager during the Jetpack installation process, requiring the SD card to be connected to the host PC machine. If a manual flash of the SD card is needed, a tool like Etcher [45] can be utilized. One can download and install Etcher, select the image file they want to flash, select the SD card, and start the flashing process.

The installation of the SSD drive and the changing of the rootfs to point to the SSD first necessitate physically connecting the SSD to the device. After this, the SSD must be formatted, which, in Linux, can be done using the 'sudo mkfs -t ext4 /dev/sdX' command, replacing *'sdX'* with the appropriate device id. One is then guided through a series of prompts to create a new partition and format it.

After the SSD has been formatted, it is mounted by running the 'sudo mount /dev/sda1 /mnt' command. The contents from the SD card are then copied to the SSD using the following *'rsync'* command :

```
sudo apt install ./sdkmanager_[version]-[build#]_amd64.deb
```

One then must edit the *'/boot/extlinux/extlinux.conf'* file on the SD card to point to the SSD, changing *'root=/dev/mmcblk0p1'* to *'root=/dev/sdX1'*. The device is then rebooted, after which the system should boot from the SSD.

It is crucial to remember to replace *'sdX'* with the user's SSD drive id and *'/dev/mmcblk0p1'* with the actual root partition. Additionally, it is of paramount importance that one backs up any vital data before proceeding with these steps and proceeds with caution when modifying any system files or disk partitions.

Once the above steps are completed, one can begin creating and training AI models. Code development can be made more effective by installing any Integrated Development Environment (IDE), in our case Visual Studio Code [46] aarch64 on the Jetson Xavier NX or by establishing a remote connection over SSH. Both these steps are optional.

3.1.3 Docker

Docker [47] is a platform that allows applications to be deployed into standardized containers, making development and deployment much easier across different environments. For machine learning model training, Docker provides key benefits like environment isolation, portability, reproducibility, and efficient resource use.

Specifically, Docker containers run on an isolated environment with consistent dependencies and configurations. This ensures reproducibility. The containers can also be easily shared as images and run anywhere with Docker, enabling portability. Docker enables efficient use of resources by allowing multiple isolated environments to run on a single machine. Finally, Docker streamlines onboarding by abstracting away low-level setup details.

With that in mind, NVIDIA provides several Docker images [48] to ease development of machine learning applications. It provides Docker images with pre-installed packages in several types of architectures and machine learning frameworks providing predictable, portable environments that simplify machine learning model development, training, and deployment. A list of all software and libraries used can be seen in the Table 3.1.

3.1.4 NVIDIA JetPack SDK

For NVIDIA Jetson platforms, such as the Jetson Xavier NX, the NVIDIA JetPack SDK provides as a full software development kit. To fully utilize the Jetson platform

TABLE 3.1 – Software and libraries utilized

Software	**Version**
Ubuntu	20.04 Focal Fossa
Python	3.8.17
Docker	20.10.21, build 20.10.21-0ubuntu1 20.04.2
Docker container image	nvcr.io/nvidia/l4t-ml :r35.2.1-py3
Jetpack	5.1.1 [L4T 35.3.1]
	l4t-ml :r35.2.1-py3
	TensorFlow 2.11.0
	PyTorch v2.0.0
	torchvision v0.14.1
	torchaudio v0.13.1
	onnx 1.13.0
	CuPy 11.5.0
	numpy 1.21.1
	numba 0.56.4
	PyCUDA 2022.2
	OpenCV 4.5.0 (with CUDA)
	pandas 1.5.3
	scipy 1.10.0
	scikit-learn 1.2.1
	JupyterLab 3.6.1
Etcher	1.18.12

for AI development and deployment, it offers developers a comprehensive set of tools, libraries, and frameworks.

With a variety of features and parts designed especially for developing AI applications, the JetPack SDK offers an extensive solution for software development on Jetson devices. Its essential parts consist of :

- **Operating System (OS) :** The JetPack SDK comes with the Jetson operating system, giving AI workloads a solid and well-optimized base. Its Linux-based platform guarantees compatibility and enables effective use of the hardware capabilities of the Jetson.
- **CUDA Toolkit and GPU Libraries :** The NVIDIA CUDA Toolkit, a parallel computing platform and programming model, is a component of the JetPack SDK that enables programmers to use parallel processing to fully utilize the enormous computational capabilities of the Jetson's GPU. In addition to CUDA, the SDK incorporates GPU-accelerated libraries like TensorRT (TensorRT Inference

Optimization library) and cuDNN (CUDA Deep Neural Network library), which optimize AI workloads, improve performance, and increase efficiency.

- **Deep Learning Frameworks :** PyTorch and TensorFlow are two prevalent open-source frameworks used in the field of machine learning and deep learning. They facilitate the creation, training, and deployment of a wide range of machine learning models, offering a high-level, user-friendly interface. PyTorch [49], developed by Facebook's artificial intelligence research group, is recognized for its dynamic computational graph and effective memory usage, which simplifies the development and understanding of the models. It provides a rich API for tensor computation and supports GPU acceleration, distributed training, various optimization methods, and much more. Furthermore, PyTorch includes a package called Torchvision, which has datasets loaders, model architectures, and image transformation tools for computer vision. On the other hand, TensorFlow [50], a product of the Google Brain team, utilizes a static computational graph, which is advantageous for distributed computing and optimization. TensorFlow caters to a broad spectrum of tasks, including neural networks, and supports a wide array of platforms, from mobile and embedded platforms to server farms. TensorFlow also incorporates TensorFlow Lite for mobile and embedded devices and TensorFlow.js for browser-based applications. Both PyTorch and TensorFlow are available and supported on the NVIDIA Jetson platform.
- **Vision and Multimedia Libraries :** A wide range of vision and multimedia libraries, including OpenCV and GStreamer, are included in the JetPack SDK. With the help of these libraries, developers can easily handle tasks like camera input handling, real-time picture analysis, and multimedia integration for AI applications.
- **Development Tools :** The NVIDIA Nsight suite is one of the many development tools available through the SDK that provides options for debugging, profiling, and optimizing GPU-accelerated apps, these tools expedite the workflow of AI development. In order to promote effective AI development on Jetson platforms, the SDK also contains performance analysis tools, system monitoring utilities, and code optimization resources.
- **Documentation and Samples :** The JetPack SDK offers a wealth of documentation, tutorials, and code samples because it understands how important guidance and reference resources are. These tools help programmers comprehend

the SDK's parts, set up their work environment, and refine their AI models for effective use on Jetson platforms. The documentation provided by the SDK serves as a thorough knowledge base that enables programmers to explore and successfully use the features of the Jetson platform.

In summary, the NVIDIA JetPack SDK is an empowering software development kit that equips developers with a comprehensive suite of tools, libraries, and frameworks to harness the full potential of the Jetson platform.

3.2 Datasets

Our work focus on both 2D and 3D object recognition task in images. We chose two 2D datasets for experimentation, STL-10 and CIFAR-10 to observe how well the selected CNN models can generalize to different quantity of data and deal with different image sizes. To test the capabilities of Nvidia Jetson Xavier NX on 3D data, we used a subset of the MedMNIST v2 dataset.

3.2.1 STL-10

STL-10 [13] was introduced in 2010 by researchers at Stanford University and is commonly employed to benchmark machine learning models, particularly in unsupervised representation learning and transfer learning given the abundance of unlabeled samples. It contains 96x96 color images (a few samples are illustrated in Fig. 3.2 across 10 classes with 500 training and 800 test images per class, totaling 5,000 labelled training images and 8,000 labelled test images. An additional 100,000 unlabeled images are provided to facilitate unsupervised learning. The images originate from the ImageNet dataset but were resized to 96x96 resolution.

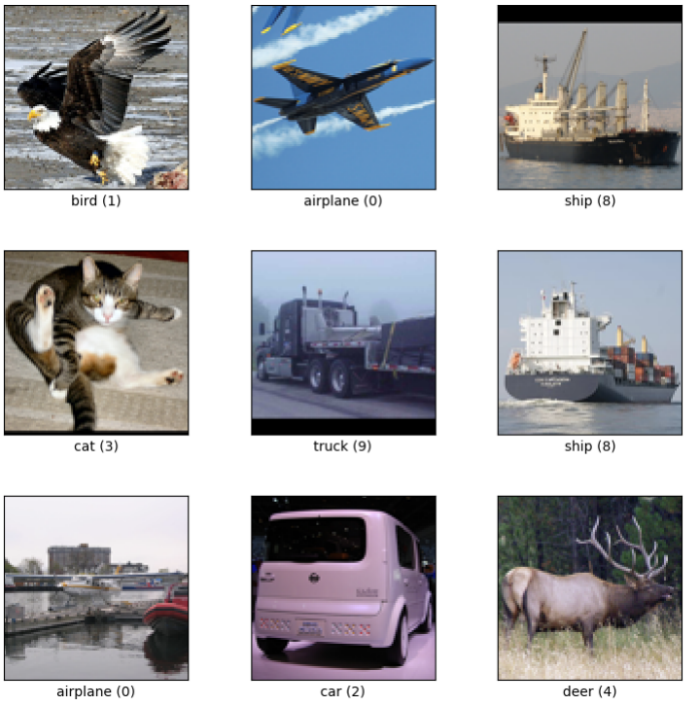

FIGURE 3.2 – STL-10 dataset [13]

3.2.2 CIFAR-10

The CIFAR-10 dataset [14] is a well-known dataset in computer-vision. It has been utilized for the purpose of object recognition. This particular dataset, a segment of the 80 million tiny images dataset, showcases an 60,000 32x32 color images (of which a few samples are shown in Figure 3.3), all of which contain one of the 10 distinct object classes. Each class is comprised of 6000 images, rendering a grand total of 10 unique object classes. The test batch contains exactly 1000 randomly-selected images from each class. It was obtained by the joint efforts of Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

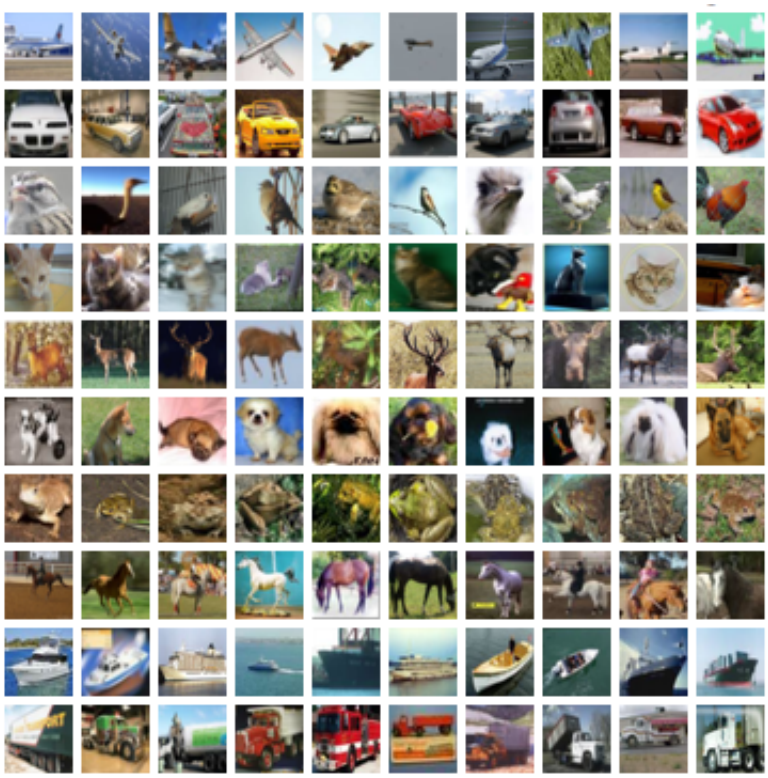

FIGURE 3.3 – CIFAR-10 dataset [14]

As the STL-10 dataset has fewer labeled training images with higher resolution (32x32 for CIFAR-10 vs. 96x96 for STL-10), we used both datasets to observe how well models can generalize to different quantity of data and deal with different image sizes.

3.2.3 MedMNISTv2

The MedMNIST v2 dataset [15] is a comprehensive collection of biomedical images. It includes 12 datasets for 2D and 6 datasets for 3D. The images are standardized into 28 x 28 (2D) or 28 x 28 x 28 (3D) and come with classification labels. MedMNIST v2 covers various data scales and tasks, making it suitable for classification on lightweight images. The dataset consists of 708,069 2D images and 9,998 3D images, which can be used for research and education purposes.

Due to the limited processing resources of Jetson Xavier NX, we decided to transform it into a 2D classification task that could be treated on-board. This is done by projecting

the 3D data onto 2D and using several 2D views to represent each 3D object. For a real object, this would look like taking pictures from several viewpoints and providing them to an algorithm to recognize the object. In this thesis, we chose to work on medical data, namely on a subset of the MedMNISTv2 dataset, the OrganMNIST. In this context, the images collected for a 3D organ are created using a technique called 'slicing'. It consists in extracting multiple 2D views/images namely axial, coronal, and sagittal 2D views from a 3D image.

The OrganMNIST {A,C,S} dataset, the subset utilized in our experiments, is based on 3D computed tomography (CT) with axial/coronal/sagittal views (2D) split into three datasets : OrganMNIST A (Axial), OrganMNIST C (Coronal), and OrganMNIST (Sagittal), as illustrated in Fig. 3.4, Fig. 3.5, Fig. 3.6, respectively. These are extracted from its analogous OrganMNIST3D dataset. The CT images in the OrganMNIST {A,C,S} dataset are of a size of 28x28 in grey scale. It has 11 annotated classes to perform multi-class classification of body organs. In total, the OrganMNIST {A,C,S} dataset has 61,521 images for training and 34,875 images for testing.

3.3 Methodology

As mentioned in the objectives, the research in this thesis is conducted using the NVIDIA Jetson Xavier NX as the hardware platform. Different compact CNN algorithms are implemented and evaluated on the Jetson Xavier platform. The performance of five common compact CNN algorithms for object recognition, namely AlexNet, ShuffleNet, SqueezeNet, Resnet50, and MobileNetV2 is evaluated using metrics such as model size, computational cost, precision, recall, and F-score. As transfer learning was demonstrated during experimentation to yield better results in shorter time, for 3D data, we only trained models using transfer learning.

As we mentioned before, due to resource constraints we reduce the 3D classification task to 2D. To perform our experiments, we combine all 2D views (axial, coronal, and sagittal) generated from 3D images into one large dataset. The CT images in the OrganMNIST {A,C,S} dataset are in grey scale and contain only one channel. However, all models evaluated expect an input image with three channels. Hence, to avoid tweaking the models' architecture, we decided to convert all images to RGB (three channels) by utilizing the PIL (Python Image Library) library [51] which copies the grey scale intensity value to all three channels (R, G, B) in the new RGB image.

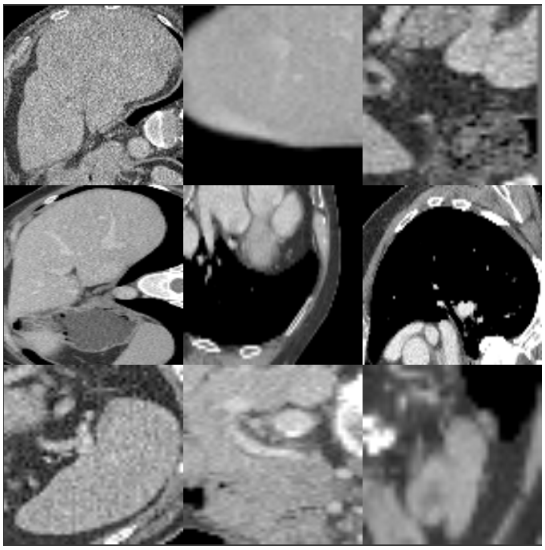

FIGURE 3.4 – OrganMNIST Axial dataset [15]

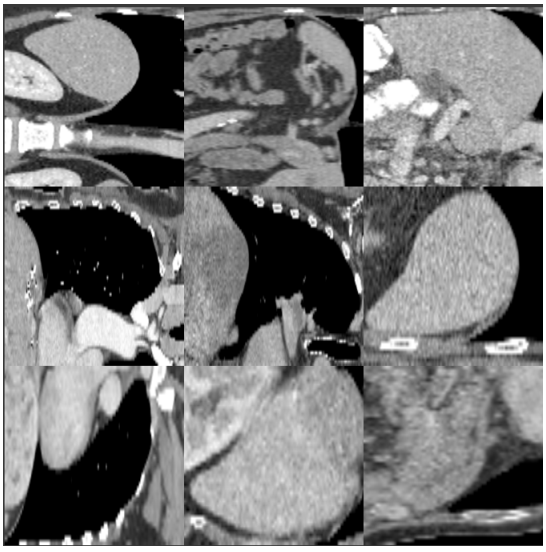

FIGURE 3.5 – OrganMNIST Coronal dataset [15]

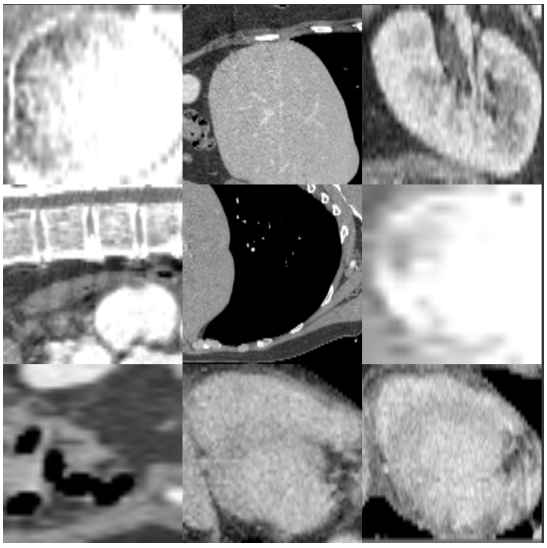

FIGURE 3.6 – OrganMNIST Sagittal dataset [15]

The training process is demonstrated in the sequence diagram in the Figure 3.7.

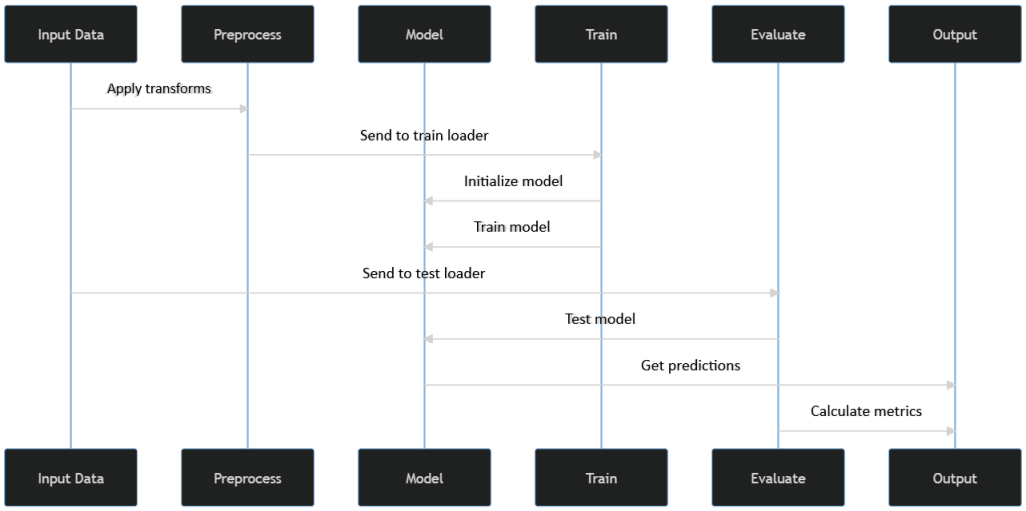

FIGURE 3.7 – Sequence diagram of the training process

The CNNs are trained both from scratch and leveraging pre-trained weights trained on the ImageNet dataset. The ImageNet dataset is one of the largest database and key reference in computer vision containing 1000 object classes and contains 1,281,167 training images, 50,000 validation images and 100,000 test images. We leverage ImageNet

for the transfer learning process to reduce the time required to train a model as it freezes the hidden layers that contain general knowledge (i.e. uses pre-trained weights obtained during learning on ImageNet dataset) and retrains only a limited number of layers, particularly those towards the output layer that contains the specific knowledge of the target task. It achieves a good performance quicker with reduced computation costs.

The images are pre-processed either via resizing, cropping and normalization transforms to conform to the expected input shape and data distribution of the models. The training and test splits are employed through PyTorch data loaders for iterative batch training and evaluation. Models are then initialized with and without pre-trained weights depending on the training session and fine-tuned via Stochastic Gradient Descent as the optimizer. All models leveraging transfer learning use the default weights in *ImageNet 1K V1* available in the Pytorch library (pre-trained on the ImageNet dataset). The Cross Entropy loss is optimized over multiple epochs with batch sizes of 64, except when mentioned otherwise in the text, due to resource limitations. Periodic loss updates are printed during training. Model evaluation on the test set computes metrics like precision, recall and F-score allowing quantitative assessment of the real-world performance of the models. Moreover, we freeze model's hidden layers and retrain only the last layer, either a fully connected one or a classifier, depending on the model.

In order to train models and assess their performance, we used the three datasets in section 3.2. Pytorch and Tensorflow are both available in the NVIDIA Jetson SDK. The PyTorch library was our choice for implementation. Moreover, we also use Torchvision, which is a PyTorch add-on library that provides datasets, model architectures, and image transformations for computer vision, NumPy [52] for numerical operations, and Scikit Learn [53] that provides utilities for machine learning, including model evaluation metrics. The Pytorch profiler, 'torch.profiler', is also used for profiling model inference to analyze GPU/CPU usage and memory consumption. For maximizing performance and get the best out of the NVIDIA Xavier NX, we activated all CPUs, enabling its 6 cores, which makes the board to consume 20 Watts of power. NVIDIA provides a script called *'jetson_clocks'*. The script is provided by NVIDIA to optimize the board performance through the implementation of static max frequency settings for CPU, GPU, and EMC clocks. It is also recommended to activate fans, but we found that the CPU and GPU temperatures are not high when the board is managed with the default values.

Each model is trained initially with 10 epochs and the number of epochs is increased to 30, 50, 60, 100, and 200 epochs, for a fixed batch size of 64, unless explicitly mentioned

otherwise in the text, due to resource limitations, until we notice that there is no further meaningful improvement.

For monitoring the training process, a script runs in background collecting CPU/GPU/RAM utilization from the board. Also, the loss is printed every batches. In each epoch, the model performs a forward pass, computes the loss, performs a backward pass, and updates the model parameters. The best model is identified by the highest F-score with less computation cost. However, the time required to train, and accuracy of a model should also be considered depending on the use case.

After training, the model's performance is evaluated on the test sets mentioned in section 3.2. The precision, recall, and F-score for each model are calculated using the Scikit Learn library's functions, using the following equations :

$$Precision = \frac{TP}{TP + FP} \tag{3.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.2}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{3.3}$$

The next chapter presents the experimental results on the NVIDIA Javier Jetson NX we obtained for the selected CNN models over the three datasets, with the methodology described. We trained AlexNet, ShuffleNetV2, SqueezeNet, Resnet50, and MobileNetV2 on 2D datasets, namely CIFAR-10 and STL-10. AlexNet and SqueezeNet were also trained on the OrganMNIST A,C,S 3D dataset, a subset of the MedMNISTV2 dataset.

Chapter 4

Experimental results

In this section, we present the experimental results we achieved using the chosen architectures, on the three selected datasets using the methodology described in section 3, organized by data type (e.g. 2D data or 3D data). As mentioned in the 'Methodology' section, for 2D data, we consider both training from scratch and leveraging pre-trained weights, whereas for 3D data we only consider leveraging pre-trained weights.

4.1 Results on 2D data

4.1.1 AlexNet

4.1.1.1 Training from scratch

Table 4.1 summarizes the training results on the CIFAR-10 dataset when training from scratch.

AlexNet shows consistent improvement in all metrics with increased training epochs, particularly evident in the F-score rates. The growth trend is linear, indicating a stable learning curve.

TABLE 4.1 – AlexNet - Performance metrics trained from scratch on CIFAR-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.658	0.642	0.640
50	0.828	0.824	0.824
100	0.831	0.831	0.831
200	0.846	0.846	0.845

F-score steadily improves from a modest 0.640 after only 10 epochs to 0.824 after 50 epochs, and to 0.831 after 100 full epochs of training. Moreover, the precision, recall and F-score remain consistently balanced in the low 0.8, signifying the well-trained model has an ideal balance of minimizing both false positives and false negatives. The smooth, steady accuracy increase demonstrates the model continues progressively learning and improving throughout all 100 epochs without plateauing or degradation. Overall, the good achievement of 0.831 F-score and robust precision and recall metrics conclusively indicate this AlexNet implementation is training well on the CIFAR-10 dataset across the full 100 training epochs. Finally, the model achieves its best performance when trained on 200 epochs with precision and recall 0.846, and F-score 0.845.

TABLE 4.2 – AlexNet - Performance metrics trained from scratch on STL-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.260	0.260	0.226
30	0.629	0.602	0.588
50	0.811	0.806	0.797
100	0.985	0.984	0.984
200	0.999	0.999	0.999

Table 4.2 summarizes the results achieved when training AlexNet from scratch on the STL-10 dataset. AlexNet attained 0.260 precision and recall on the STL-10 dataset with 10 initial training epochs, clearly showing signs of underfitting. Continuing to 30 epochs substantially improved the precision and recall to 0.629 and 0.602 respectively, constituting noticeable yet incomplete convergence given remaining room for enhancement. Further training to 50 epochs yielded additional gains, achieving strong precision and recall of 0.811 and 0.806 respectively, steadily improving its F-score to 0.797 at this point. Peak results were attained at 100 epochs, with AlexNet producing near perfect precision and recall measures of 0.985 and 0.984 respectively. Extending training to 200 epochs maintained the near perfect accuracy, with very few false positive and false negative errors. More investigation is required to assess if the model is actually overfitting at this point.

4.1.1.2 Transfer learning

As shown in Table 4.3, the first experiment with 10 epochs with pre-trained weights, the model was able to achieve impressive value of 0.802 for the F-score with values of

0.803 precision and 0.802 recall, respectively, representing an increase in F-score of 18% with respect to the AlexNet trained from scratch. At 50 epochs, it reaches 0.907 F-score with precision 0.908 and recall 0.908, thus showing a 10% performance improvement with only 40 extra epochs and taking 5 hours of training.

TABLE 4.3 – AlexNet - Performance metrics with pre-trained weights on CIFAR-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.803	0.802	0.802
50	0.908	0.908	0.907
100	0.912	0.911	0.911

However, it is not able to improve further after reaching 100 epochs, achieving an F-score of 0.911. Increasing the number of epochs to 200 epochs did not bring any additional improvements.

On the STL-10 dataset, as shown in Table 4.4, the pre-trained AlexNet model achieves a 0.938 F-score at the very first experiment with 10 epochs. At 50 epochs, it achieves a precision, recall, and F-score of 0.997, achieving better results than its counterpart from scratch with less than half epochs.

TABLE 4.4 – AlexNet - Performance metrics with pre-trained weights on STL-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.940	0.938	0.938
30	0.995	0.995	0.995
50	0.997	0.997	0.997

4.1.2 ShuffleNet V2

4.1.2.1 Training from scratch

The training that spans across 10, 30, 100, and 200 epochs for the ShuffleNet V2 model on CIFAR-10 dataset shows that after only 10 initial training epochs, the model was able to achieve an F-score of just 0.666, as shown in Table 4.5. Continuing training out to 30 epochs significantly improved the F-score to 0.718 indicating the model continued to improve. Further training to 100 epochs enabled the model to reach its near best F-score of 0.741. Extending the training to 200 epochs achieved very marginal further improvements with both precision and F-score at 0.743, and a recall of 0.745.

TABLE 4.5 – ShuffleNet V2 - Performance metrics training from scratch on the CIFAR-10 dataset

Epochs	**Precision**	**Recall**	**F-score**
10	0.667	0.667	0.666
30	0.718	0.719	0.718
100	0.740	0.741	0.741
200	0.743	0.745	0.743

Throughout the training process, the precision and recall metrics remained consistently balanced in the 0.70-0.80 range, demonstrating that the model maintained a good balance of maximizing true positives while minimizing false positives. Despite the fact that the model achieved higher values when trained on 200 epochs, 100 epochs gave the best generalizability and performance for ShuffleNet V2 on the CIFAR-10 dataset, while requiring half the number of epochs.

On STL-10, as shown in Table 4.6, the model seems to have difficulty learning, and it scores low overall. In the initial 10 epochs, it achieves a performance of 0.322 for the F-score, of 0.347 for precision and 0.342 for recall. Continuing the training to 30 epochs enhance the results by improving both precision and recall to 0.428 and 0.434 respectively. At 50 epochs, it kept improving slowly with 0.02% increase in precision, totalling 0.448 and recall merely increasing 0.005%, totalling 0.439.

The best results were obtained at 200 epochs with 0.475 precision, 0477 recall, and 0476 F-score. However, the model obtained virtually the same results with 100 epochs, which it is considered a poor performance, given they are below the 50% probability.

Summarizing, the model shows consistent but gradual improvement as the number of epochs increase despite showing signs of plateauing at higher epochs.

TABLE 4.6 – ShuffleNet V2 - Performance metrics training from scratch on the STL-10 dataset

Epochs	**Precision**	**Recall**	**F-score**
10	0.347	0.342	0.322
30	0.428	0.434	0.429
50	0.448	0.439	0.442
100	0.475	0.475	0.474
200	0.475	0.477	0.476

4.1.2.2 Transfer learning

Training ShuffleNet V2 using pre-trained weights on the CIFAR10 dataset has dramatically improved accuracy with very few epochs. In the first experiment with 10 epochs, the model was able to achieve a value of 0.916 for precision, recall and F-score. Unlike the model trained from scratch, this model did not see any significant improvement as the number of epochs increased and peaked at 0.929 for all metrics in both experiments with 100 and 200 epochs.

TABLE 4.7 – ShuffleNet V2 - Performance metrics with pre-trained weights on the CIFAR-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.916	0.916	0.916
30	0.924	0.924	0.924
50	0.927	0.928	0.927
100	0.929	0.929	0.929
200	0.929	0.929	0.929

On STL-10, as shown in Table 4.8, ShuffleNetV2 demonstrates a commendable start in the initial 10 epochs, with precision at 0.787, recall at 0.720, and an F-score at 0.718. As the training progresses from 50 to 200 epochs, there is a notable and consistent improvement in performance metrics. The model precision keeps on increasing, reaching its peak at 100 epochs. At this point, ShuffleNetV2 attains a precision score of 0.914, and 0.913 for recall and F-score. Pushing the training to 200 epochs provided no benefit ; in fact the results are worse in terms of metrics performance.

TABLE 4.8 – ShuffleNet V2 - Performance metrics with pre-trained weights on the STL-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.787	0.720	0.718
50	0.880	0.880	0.880
100	0.914	0.913	0.913
200	0.911	0.911	0.911

4.1.3 SqueezeNet

4.1.3.1 Training from scratch

The results for the SqueezeNet trained on the CIFAR-10 dataset over 10, 30, 50, and 100 epochs are shown in Table 4.9. One can notice that after just 10 initial training epochs, the model achieved only a value of 0.571 for F-score, indicating that more training would be required for the model to converge. Continuing the training to 30 epochs significantly improved the F-score (to 0.702). Further training to 50 epochs enabled the model to reach its peak F-score of 0.761. Ultimately, training to 100 epochs resulted in the F-score decreasing slightly to 0.755, suggesting that 50 epochs seem to be the optimal training duration before some performance degradation occurs.

TABLE 4.9 – SqueezeNet - Performance metrics trained from scratch on the CIFAR-10 dataset

Epochs	**Precision**	**Recall**	**F-score**
10	0.599	0.573	0.571
30	0.719	0.711	0.702
50	0.767	0.763	0.761
100	0.779	0.753	0.755

As Table 4.10 demonstrates, SqueezeNet attained limited 0.266 precision and recall on the STL-10 dataset with 10 initial training epochs, significantly underfitting at this point. With 30 epochs, the model improved both precision and recall to 0.457 and 0.374 respectively, indicating that the model is learning and there is still room for improving the performance. Moving forward training to 50 epochs continues to show gains, achieving better precision and recall (i.e 0.559 and 0.486 respectively), implying that further training could still improve learning. The best performance for SqueezeNet, for all metrics, is achieved for 100 epochs.

TABLE 4.10 – SqueezeNet - Performance metrics trained from scratch on the STL-10 dataset

Epochs	**Precision**	**Recall**	**F-score**
10	0.266	0.274	0.221
30	0.457	0.374	0.342
50	0.559	0.486	0.471
100	0.613	0.567	0.571

4.1.3.2 Transfer learning

The outcome of training SqueezeNet with pre-trained weights on the CIFAR-10 dataset demonstrated in Table 4.11 was outstanding. In the very early stages of training, with only 10 epochs, the model was able to achieve a value of 0.874 for the F-score with values of 0.875 for recall and 0.878 for precision. In contrast, its non-pretrained model solely achieved 0.571 F-score, as shown in Table 4.9 above. Going forward, training the model with 30 epochs results in an F-score, recall, and precision of 0.902. However, it seems to show signs of overfitting when trained with 50 epochs, decreasing its F-score to 0.891 with 0.894 precision and 0.891 recall. At 100 epochs the model shows signs of stabilization with a mere 0.1% improvement in F-score.

TABLE 4.11 – SqueezeNet - Performance metrics with pre-trained weights on the CIFAR-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.878	0.875	0.874
30	0.902	0.902	0.902
50	0.894	0.891	0.891
100	0.904	0.902	0.903

On the STL-10 dataset, demonstrated in Table 4.12, SqueezeNet attains remarkably strong precision, recall, and F-score metrics of 0.862, 0.862, and 0.861 respectively, after mere 10 epochs. Continuing training on more epochs did not bring substantial gains and the model show signs of stabilization.

TABLE 4.12 – SqueezeNet - Performance metrics with pre-trained weights on the STL-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.862	0.862	0.861
30	0.864	0.861	0.860
50	0.867	0.866	0.866
100	0.866	0.866	0.866

4.1.4 Resnet50

4.1.4.1 Training from scratch

On the CIFAR-10 dataset, ResNet50 trained from scratch was able to achieve merely a value of 0.258 for the F-score after 10 initial training epochs, as illustrated in Table 4.13. Proceeding with training out to 30 epochs improved the F-score to 0.338 still with clear room for additional learning, hinting that the model remained incompletely converged. At 50 epochs, the model exhibited continued steady convergence and achieved an F-score 0.416, peaking its F-score at 0.649 at 100 epochs. At this point, the model stopped showing signs of learning as it decreases F-score to 0.611, precision to 0.609, and recall to 0.613. At 100 epochs, the model achieved virtually the same performance as observed for 200 epochs, thus representing a good compromise between training time and performance.

TABLE 4.13 – ResNet50 - Performance metrics trained from scratch on the CIFAR-10 dataset

Epochs	Precision	Recall	F-score
10	0.275	0.271	0.258
30	0.343	0.343	0.338
50	0.421	0.423	0.416
100	0.655	0.647	0.649
200	0.609	0.613	0.611

On the STL-10 dataset, indicated in Table 4.14, after only 10 training epochs, the ResNet50 model attained limited F-score of 0.368, a precision of 0.389, and a recall of 0.387 on the STL-10 dataset, clearly indicating underfitting at this early stage.

The model shows continuing improvement in performance, peaking at 100 epochs. At 200 epochs, the model decreases its performance, hinting that a possible overfitting may have happened. More investigation is needed to understand the reasons behind this variation. The results obtained are subpar as the model performance is barely better than the 50% probability.

4.1.4.2 Transfer learning

Leveraging transfer learning, the model results start off with a value of 0.823 for the F-score with both precision and recall at 0.823 with only 10 epochs. Moving up to 30

TABLE 4.14 – ResNet50 - Performance metrics trained from scratch on the STL-10 dataset

Epochs	**Precision**	**Recall**	**F-score**
10	0.389	0.387	0.368
30	0.444	0.435	0.436
50	0.472	0.468	0.468
100	0.449	0.452	0.448
200	0.464	0.470	0.464

epochs the model slightly improves its F-score to 0.841 with precision and recall at 0.842. Going for 100 epochs makes the model reach an F-score of 0.847, thus seeming to have plateaued. The model shows minor improvements when trained for 200 epochs achieving a value of 0.853 for the three metrics. The best trade-off is observed for 100 epochs, with both precision and F-score of 0.847, and recall of 0.848, as no drastic improvement in performance is observed with double the training time in 200 epochs. Overall, training the model with pre-trained weights gives considerably better results with only 10 epochs. Table 4.15 summarizes the training results of ResNet50 trained with pre-trained weights on the CIFAR-10 dataset.

TABLE 4.15 – ResNet 50 - Performance metrics with pre-trained weights on the CIFAR-10

Epochs	**Precision**	**Recall**	**F-score**
10	0.823	0.823	0.823
30	0.842	0.842	0.841
50	0.846	0.846	0.846
100	0.847	0.848	0.847
200	0.853	0.853	0.853

On the STL-10 dataset, indicated in Table 4.16, ResNet50 achieves its best performance in early stages for 10 epochs, with an F-score, precision, and recall of 0.914. This is impressive since it requires very little training time for such good results. Training the model for more epochs actually decrease its performance indicating signs of stabilization and plateauing.

TABLE 4.16 – ResNet 50 - Performance metrics with pre-trained weights on the STL-10

Epochs	Precision	Recall	F-score
10	0.914	0.914	0.914
30	0.911	0.911	0.911
50	0.909	0.908	0.908
100	0.905	0.904	0.904
200	0.908	0.909	0.908

4.1.5 MobileNetV2

4.1.5.1 Training from scratch

As shown on Table 4.17, the training results for the MobileNet V2 architecture on the CIFAR-10 dataset show that after only 10 initial training epochs, the model was able to achieve a very low F-score of 0.258. Training for 30 epochs slowly improved the F-score to 0.338, an increase of 0.08 for 20 extra epochs, indicating model convergence is improving as the number of epochs increase. Further training for 50 epochs and 100 epochs continue to lead to more improvement with F-score values of 0.416 and 0.649 respectively, peaking when trained for 150 epochs with a F-score and precision values of 0.750 and recall of 0.751. Similar progression was also observed in ResNet50 trained from scratch. After 200 epochs, the model performance decreases, with F-score of 0.611, a value lower than what it achieved with 100 epochs, thus indicating signs of overfitting.

Trained on the STL-10 dataset, MobileNet V2 performed poorly in all three metrics on every epoch tested with F-score values ranging from 0.121 (trained for 10 epochs) to 0.257 (trained for 150 epochs). It reached its peak performance when trained for 150 epochs with a precision value of 0.328, recall of 0.276, and F-score of 0.257. It is very likely that the model could not learn properly due to not sufficient data, since the STL-10 dataset is rather a small dataset. The lack of sufficient data may also explain the poor performance of MobileNet V2 in CIFAR-10, which is not a big dataset either. Further investigation is required to pinpoint the root cause. Table 4.18 summarizes the training results of Mobilenet V2 trained from scratch on the STL-10 dataset.

Overall, MobileNet V2 showed signs of difficulty in learning, requiring either more data or more epochs to improve its performance.

TABLE 4.17 – MobileNet V2 - Performance metrics trained from scratch on the CIFAR-10 dataset

Epochs	Precision	Recall	F-score
10	0.275	0.271	0.258
30	0.343	0.341	0.338
50	0.421	0.423	0.416
100	0.655	0.647	0.649
150	0.750	0.751	0.750
200	0.609	0.613	0.611

TABLE 4.18 – MobileNet V2 - Performance metrics trained from scratch on the STL-10 dataset

Epochs	Precision	Recall	F-score
10	0.208	0.159	0.121
30	0.222	0.196	0.161
50	0.214	0.206	0.160
100	0.282	0.237	0.212
150	0.328	0.276	0.257
200	0.330	0.283	0.248

4.1.5.2 Transfer learning

Again, by leveraging transfer learning, MobileNet V2 achieves an F-score of 0.798, a precision of 0.799 and a recall of 0.798 with only 10 epochs. This is an outstanding improvement compared to the model trained from scratch on the CIFAR10 dataset alone. Moving up to 30 epochs, it improves the F-score to 0.825 with a value for precision of 0.825 and for recall of 0.826. Going for 100 epochs makes the model's metrics to stay at 0.82 range, hinting that the learning has stabilized. Similar results are achieved when the model gets trained for 200 epochs. The best results from the model trained from scratch with 100 epochs peaks at values for both F-score and precision of 0.828 and of 0.830 for recall. Table 4.19 summarizes the training results of MobileNet V2 with pre-trained weights on the CIFAR-10 dataset.

On the STL-10 dataset, similar to the results observed in all previous pre-trained tests, the model performed substantially better than its version trained from scratch, as demonstrated in Table 4.20. The model performance steadily improves as the number of epochs increases still showing signs of improvement even after the 200-epochs mark

TABLE 4.19 – Mobilenet V2 - Performance metrics trained with pre-trained weights on the CIFAR-10 dataset.

Epochs	Precision	Recall	F-score
10	0.799	0.798	0.798
30	0.825	0.826	0.825
50	0.825	0.824	0.824
100	0.828	0.830	0.828
150	0.825	0.825	0.825
200	0.824	0.823	0.823

with a F-score of 0.761, 0.774 precision and 0.758 recall. The best results were found at 150 epochs with F-score and recall values of 0.781 and precision of 0.786.

TABLE 4.20 – MobileNet V2 - Performance metrics with pre-trained weights on the STL-10

Epochs	Precision	Recall	F-score
10	0.604	0.592	0.591
30	0.711	0.697	0.696
50	0.491	0.459	0.458
100	0.738	0.723	0.721
150	0.786	0.781	0.781
200	0.774	0.758	0.761

A discussion of these results as well as a comparison in terms of performance and computation time is provided in section 4.3.

4.2 Results on 3D data

4.2.1 AlexNet

On the OrganMNIST{A,C,S} dataset, as shown in Table 4.21, AlexNet's performance continues to increase up to 100 epochs, achieving a F-score of 0.845, and both precision and recall of 0.846, indicating the benefits of extended training. However, these metrics peak at 100 epochs and actually decline slightly by 200 epochs. This suggests the model starts overfitting after 100 epochs. Training for 200 epochs takes 19h59min, about twice as long as for 100 epochs, while providing no improvement in metrics.

TABLE 4.21 – AlexNet - Performance metrics with pre-trained weights on OrganMNIST {A,C,S}

Epochs	Precision	Recall	F-score
10	0.820	0.818	0.817
30	0.836	0.836	0.834
50	0.839	0.839	0.837
100	0.846	0.846	0.845
200	0.845	0.844	0.843

4.2.2 ShuffleNetV2

As illustrated in Table 4.22, ShuffleNetv2 performs surprisingly poor on OrganMNIST{A,C,S} when trained for 10 epochs with an F-score of 0.078, a precision of 0.199, and a recall of 0.210. Subsequently, we continued training the model for the usual 30, 50, 100, and 200 epochs. The metrics improved steadily all the way to 200 epochs, implying the model has not converged yet and could still benefit from more training data. The ShuffleNetV2 performance slowly increased peaking at 200 epochs with a precision of 0.638, recall of 0.623, and F-score of 0.611.

TABLE 4.22 – ShuffleNetV2 - Performance metrics with pre-trained weights on OrganMNIST {A,C,S}

Epochs	Precision	Recall	F-score
10	0.199	0.210	0.078
30	0.443	0.422	0.355
50	0.556	0.496	0.443
100	0.603	0.567	0.534
200	0.638	0.623	0.611

4.2.3 SqueezeNet

The performance of SqueezeNet on the OrganMNIST{A,C,S} dataset, is shown in Table 4.23. One can notice that there is a steady increase in all metrics as the number of epochs increases from 10 to 100, with the F-score varying from 0.737 up to a maximum of 0.762. The model metrics peak at 30 epochs with a precision of 0.766, a recall of 0.763, and an F-score reaching 0.762. This indicates that the model hardly benefits from more training over additional training epochs. In this particular case, there has not been a significant improvement in performance to justify training over 30 epochs

which takes 3h52min to complete. Increasing number of epochs does not translate in increasing performance. It actually diminishes returns as F-score goes down from 0.762 (at 30 epochs) to 0.741 (at 200 epochs).

TABLE 4.23 – SqueezeNet - Performance metrics with pre-trained weights on OrganMNIST {A,C,S}

Epochs	Precision	Recall	F-score
10	0.745	0.743	0.737
30	0.766	0.763	0.762
50	0.765	0.757	0.752
100	0.773	0.759	0.759
200	0.767	0.751	0.741

4.2.4 Resnet50

On OrganMNIST{A,C,S}, Resnet50 takes significantly more time to train than other models, as one can notice in Table 4.24. In our initial experiments, we were not able to train it using the usual batch size of 64 used to train other models due to resource limitation, in particular to insufficient memory on the board. We continuously encountered 'Out of Memory' issues. Hence, we tried a couple of different batch sizes and we finally were able to train ResNet50 on 32 batches. It took 3h19min to train for 10 epochs yielding a precision of 0.723, a recall of 0.720, and an F-score of 0.714. At 50 epochs, we noticed a slightly improvement in performance with a precision of 0.750, a recall of 0.751 and an F-score of 0.748. However, it took 16h08min to be complete. Nothing impressive in terms of performance that could justify the time it took to be trained. At this point, we decided to stop further training as the estimated time required to go to the next target epochs (100 and 200) was around 30h and 60h respectively.

TABLE 4.24 – Resnet50 - Performance metrics with pre-trained weights on OrganMNIST {A,C,S}

Epochs	Precision	Recall	F-score
10	0.723	0.720	0.714
30	0.752	0.734	0.731
50	0.750	0.751	0.748
100	-	-	-
200	-	-	-

4.2.5 MobileNetV2

Finally, the results for MobilenetV2 on the OrganMNIST{A,C,S} dataset are presented in Table 4.25, MobileNetV2 is also trained with 32 batches size, instead of 64, as in the case of ResNet50, due to 'Out of memory' issues. It is able to achieve decent performance after 10 epochs with precision, recall, and F-score of 0.716, 0.713 and 0.708, respectively. No significant improvements are observed in the three metrics as the number of epochs increases. The F-score peaks at 200 epochs with precision of 0.713, recall of 0.716 and F-score of 0.713 right after a small decline in performance at 100 epochs.

TABLE 4.25 – MobileNetV2 - Performance metrics with pre-trained weights on OrganMNIST {A,C,S}

Epochs	**Precision**	**Recall**	**F-score**
10	0.716	0.713	0.708
30	0.715	0.715	0.710
50	0.718	0.718	0.712
100	0.729	0.713	0.709
200	0.719	0.716	0.713

The next section will summarize the obtained results and offer a performance comparison for the studied CNN architectures in terms of performance metrics and computational time.

4.3 Comparison of performance and training time for the three datasets

Table 4.26 and 4.27 summarize the results we obtained on the two 2D datasets using the five tested CNN architectures when trained from scratch and when using the pre-trained weights computed by transfer learning. Table 4.28 summarizes the results we obtained on the 3D dataset using the five tested CNN architectures only leveraging pre-trained weights.

On the CIFAR-10 dataset, the AlexNet model trained from scratch improved its F-score from a modest 0.640 after only 10 epochs to an impressive F-score of 0.824 after 50 epochs and finally to 0.845 after 200 full epochs of training. Using transfer learning, the performance of the model increases significantly with fewer epochs, achieving an F-score of 0.911 for 100 training epochs. The pre-trained ShuffleNetV2 model achieved an

F-score of 0.916 with only 10 epochs. Unlike the model trained from scratch, this model did not see significant improvement as the number of epochs increased and peaked at an F-score of 0.924 for 30 epochs. The SqueezeNet with transfer learning scored very well (an F-score of 0.902 trained for 30 epochs) with only 3 hours of training. The pre-trained ResNet50 achieved an F-score of 0.841 with 30 epochs. On the other hand, MobileNetV2 showed a modest improvement of 0.078 in F-score when using pre-trained weights versus training from scratch, but in less than half training time.

TABLE 4.26 – Summary of best model performance on CIFAR-10

Model	**Training**	**Epochs**	**Precision**	**Recall**	**F-Score**	**Time to Train**
AlexNet	Scratch	200	0.846	0.846	0.845	21h46min
	Pre-trained	100	0.912	0.911	0.911	10h47min
ShuffleNet	Scratch	100	0.740	0.741	0.741	13h02min
	Pre-trained	30	0.924	0.924	0.924	4h07min
SqueezeNet	Scratch	50	0.767	0.763	0.761	11h40min
	Pre-trained	30	0.902	0.902	0.902	3h
ResNet50	Scratch	100	0.655	0.647	0.649	38h15min
	Pre-trained	30	0.842	0.842	0.841	1h24min
MobileNetV2	Scratch	150	0.750	0.751	0.750	5h
	Pre-trained	100	0.828	0.830	0.828	2h12min

TABLE 4.27 – Summary of best model performance on STL-10

Model	**Training**	**Epochs**	**Precision**	**Recall**	**F-Score**	**Time to Train**
AlexNet	Scratch	100	0.985	0.984	0.984	1h33min
	Pre-trained	30	0.995	0.995	0.995	30min
ShuffleNet	Scratch	100	0.475	0.475	0.474	1h20min
	Pre-trained	100	0.914	0.913	0.913	1h15min
SqueezeNet	Scratch	100	0.613	0.567	0.571	2h30min
	Pre-trained	10	0.862	0.862	0.861	8min
ResNet50	Scratch	200	0.449	0.452	0.464	3h42min
	Pre-trained	10	0.914	0.915	0.914	10min
MobileNetV2	Scratch	150	0.328	0.276	0.257	33min
	Pre-trained	150	0.786	0.781	0.781	32min

As expected, the pre-trained models performed very well compared to their counterparts trained from scratch (average of 23.7% over the two datasets), as the base model

trained on ImageNet is suitable for the task. Also, the pretrained model took less training time (average of 74.6% shorter over the two 2D datasets). The pretrained AlexNet performed better (8.4% improvement) on STL-10 compared to CIFAR-10, with only 30 epochs instead of 100 epochs on CIFAR-10. The same stays true for the counterpart trained from scratch, i.e., a 13.9% improvement on CIFAR for half the training epochs. This suggests that AlexNet can learn well with fewer images but with higher resolution samples. The pre-trained ShuffleNet performed well on CIFAR-10 with only 30 epochs and on STL-10 took more time (100 epochs) to achieve an F-score greater than 0.90. The ShuffleNet model trained from scratch obtained subpar performance (less than 0.5) on STL-10, suggesting that the model struggles with less data. SqueezeNet achieved an F-score of 0.861 with only 10 epochs on the STL-10 dataset and the performance didn't improve with more epochs. However, it achieves an F-score of 0.902 with 30 epochs on CIFAR-10. Similar to ShuffleNet, the SqueezeNet model trained from scratch obtained a low performance (0.57). This implies that SqueezeNet requires more data to increase performance. ResNet50 achieves an F-score of 0.914 with only 10 epochs on the STL-10 dataset but only scores 0.842 with 30 epochs on CIFAR-10. Like AlexNet, ResNet50 does well with few data but with higher resolution images. MobileNetV2 did not show a big discrepancy in terms of F-score across the two datasets, performing slightly better (4% improvement) on the CIFAR-10 dataset with 50 fewer epochs.

On OrganMNIST {A,C,S}, AlexNet achieved a F-score of 0.845 trained for 100 epochs, taking 10h13min to be trained. SqueezeNet did not perform as well as AlexNet, but performed reasonably well with only 30 epochs and under 4h of training, achieving an F-score of 0.762. Still on OrganMNIST {A,C,S}, ResNet50 was the slowest model to train forcing us to stop further training at 50 epochs as the estimated time required to train for 100 and 200 epochs was around 30h and 60h respectively. The increased training time is expected given the reduced number of batches per epoch, scaled down to 32 due to memory issues, and the number of layers it has (50) compared to other models. MobilenetV2 showed slightly worse results compared to SqueezeNet. The worst performance came from ShufflenetV2 that only achieved an F-score slightly higher than 0.6 after being trained for 200 epochs and take around 18h. The best performance on this dataset came from AlexNet trained for 100 epochs with precision, recall, and F-score of 0.846, 0.846, 0.845, respectively for 10h13min of training. The second best was SqueezeNet trained for 30 epochs with precision, recall, and F-score of 0.766, 0.763, 0.762, taking 3h52min to complete.

It is worth mentioning that those results were the best in terms of F-score. Nonetheless, some models still achieved a decent performance, slightly lower than their best results, taking considerably less training time. For example, on OrganMNIST {A,C,S}, SqueezeNet took 3h52min trained for 30 epochs to achieve its highest F-score of 0.762. However, it took just 1h57min to achieve an F-score of 0.737. A slightly decrease in F-score with about 2h shorter training time. AlexNet took 10h13min to achieve an F-score of 0.845 but took only 1h33min to achieve an F-score of 0.817. A mere 0.028 reduction in F-score performance with around 9h shorter training time. Similarly, MobilenetV2 took 25h06min to achieve an F-score of 0.713 but it took just 1h18min to achieve an F-score of 0.708 which is nearly a 24h reduction in training time for a 0.005 decrease in F-score.

TABLE 4.28 – Summary of best model performance on OrganMNIST {A,C,S}

Model	**Training**	**Epochs**	**Precision**	**Recall**	**F-Score**	**Time to Train**
AlexNet	Pre-trained	100	0.846	0.846	0.845	10h13min
SqueezeNet	Pre-trained	30	0.766	0.763	0.762	3h52min
ResNet50	Pre-trained	50	0.750	0.751	0.748	16h08min
MobileNetV2	Pre-trained	200	0.719	0.716	0.713	25h06min
ShuffleNetV2	Pre-trained	200	0.638	0.623	0.611	17h56min

The models overall did not perform as well as they did on the CIFAR-10 and STL-10 datasets. A few possible explanations for these results are : 1) the CT images size is too small (28x28) which some models may have had difficulty learning ; 2) the grey scale to RGB conversion (from one channel to three channels), and 3) the pre-trained weights used from ImageNet which do not contain any medical or CT-like images. All these three factors could have influenced to the slightly worse performance of the models.

As previously mentioned, the best performing model on the CIFAR-10 dataset was the pre-trained ShuffleNet trained on 30 epochs with an F-score of 0.924, whereas the pre-trained AlexNet model achieved an F-score of 0.995 on the STL-10 dataset when trained with 30 epochs. On average, ShuffleNet achieved an F-score of 0.918 over the two datasets, whereas AlexNet achieved 0.953. Even though AlexNet scored slightly better than ShuffleNet, the training time is another determining factor. On average, ShuffleNet only needed 2h41min to achieve good performance on both 2D datasets, while AlexNet took 5h30min, making ShuffleNet the preferred model. Regarding the speed training, the pre-trained ResNet50 model was the fastest to train on CIFAR-10,

taking 1h24min to obtain an F-score of 0.841, which is approximately half the time. SqueezeNet, the second fastest model, took 3h to achieve an F-score of 0.902 on CIFAR-10. On STL-10, the pre-trained SqueezeNet was the fastest, showing an F-score of 0.861 in 8 minutes. On the other hand, the pretrained ResNet50 model took only two extra minutes (10 minutes total) to reach an impressive F-score of 0.914. On OrganMNIST {A,C,S}, the best performance was achieved by AlexNet trained for 100 epochs followed by SqueezeNet trained for 100 epochs in 2h43min. Overall, AlexNet, SqueezeNet, and ShuffleNet achieved good results in all three datasets tested.

Ultimately, deciding on what is best in terms of training time and accuracy depends on the use case. In fields like medicine and finance, high accuracy is usually preferred even if it means longer training time whereas in other fields sacrificing accuracy may be acceptable if training time is shorter.

4.3.1 Computational resource usage

As one of the objectives is to analyze the resource usage, this section presents, as an example, the results of the computational resource usage observed on the NVIDIA Jetson Xavier NX captured while training AlexNet for 10 epochs on the OrganMNIST {A,C,S} dataset achieving a precision of 0.758, recall of 0.744, and F-score of 0.746. Similar results were observed during all our experiments.

The NVIDIA Jetson Xavier NX was overall very efficient in terms of resource utilization, using almost 100% of GPU and RAM available as seen in Fig. 4.1 and 4.2 :

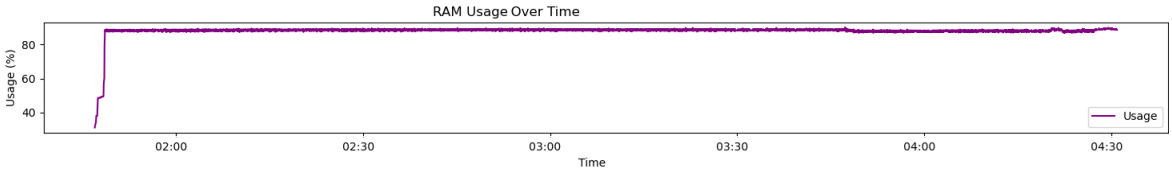

FIGURE 4.1 – Memory usage (%) over time

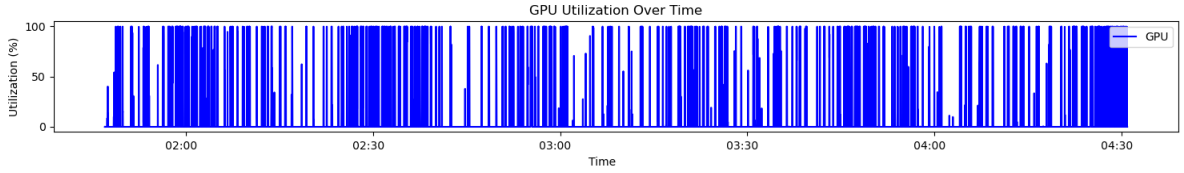

FIGURE 4.2 – GPU usage (%) over time

In terms of CPU utilization, the average usage of all 6 CPUs was around 34% and, as shown in Fig. 4.3, the CPU load was equally balanced among all CPUs :

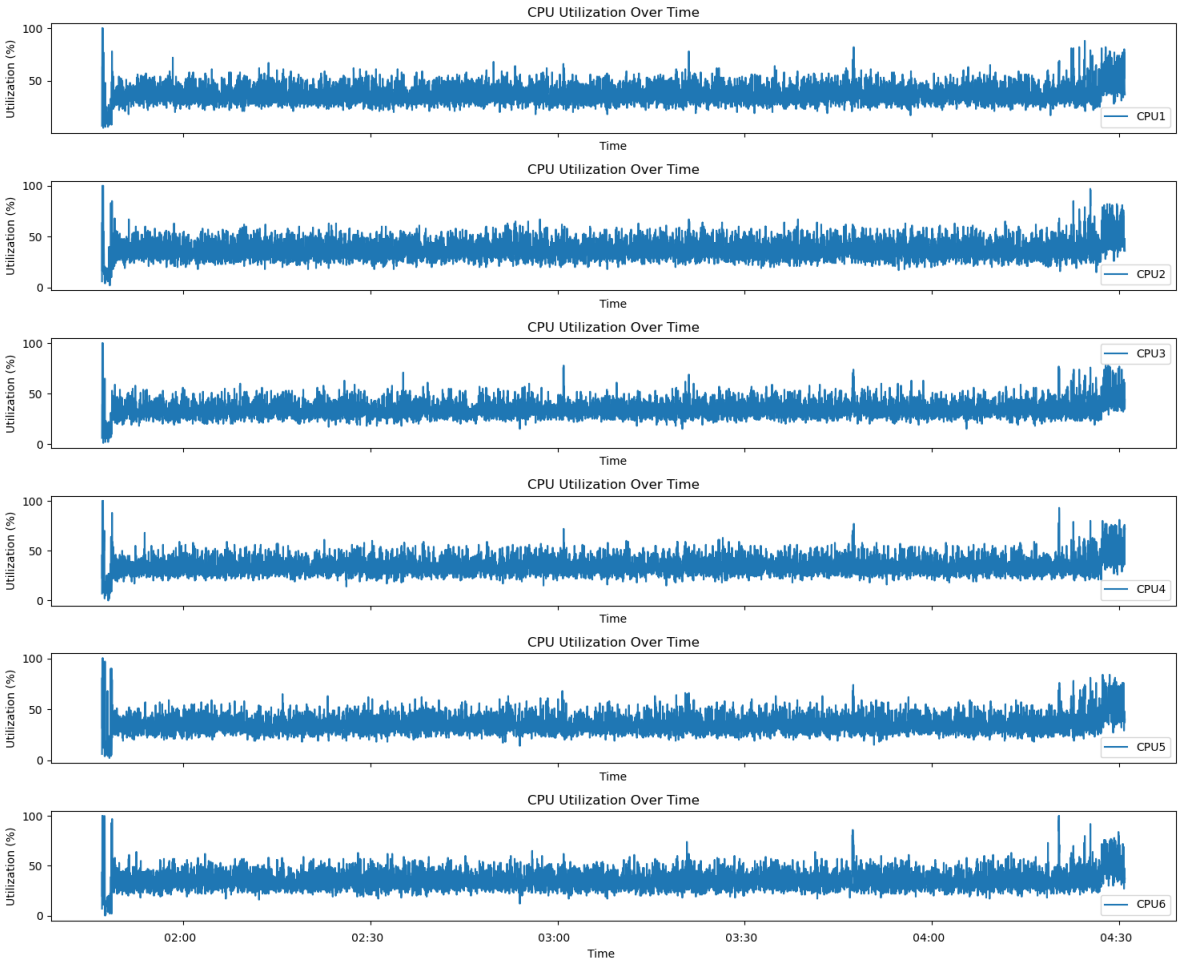

FIGURE 4.3 – CPUs usage (%) over time for the 6 CPUs of Jetson Xavier NX

In terms of CPU and CPU temperatures, they both stayed between 50°C and 74°C with an average of 71°C for CPU and 69°C for GPU. Table 4.29 illustrates the summary statistics of the temperatures while training AlexNet for 10 epochs on the OrganMNIST {A,C,S} dataset.

	Temp CPU (°C)	Temp GPU (°C)
mean	71.08	69.46
min	50.00	48.00
- 25%	72.00	69.50
- 50%	72.00	70.50
- 75%	72.50	70.50
max	73.50	74.00

TABLE 4.29 – NVIDIA Jetson NX Xavier CPU and GPU temperature summary statistics

	GPU (%)	CPU1 (%)	CPU2 (%)	CPU3 (%)	CPU4 (%)	CPU5 (%)	CPU6 (%)	RAM (GB)
mean	13.96	36.85	36.87	34.23	34.30	34.96	34.98	5.89
min	0.00	5.00	2.00	1.00	0.00	2.00	0.00	2.09
- 25%	0.00	32.00	31.00	29.00	29.00	30.00	29.00	5.90
- 50%	0.00	36.00	36.00	33.00	33.00	33.00	33.00	5.92
- 75%	0.00	41.00	41.00	38.00	38.00	39.00	39.00	5.94
max	99.80	100.00	100.00	100.00	100.00	100.00	100.00	6.04

TABLE 4.30 – NVIDIA Jetson NX Xavier summary statistics

The primary limitations encountered while training CNNs on the NVIDIA Xavier NX board stemmed from the constrained memory resources available. We encountered some challenges while training larger models such as VGG-11 and VGG-19 on CIFAR-10, Resnet50 on OrganMNIST {A,C,S}, with the container repeatedly exiting due to 'Out of Memory' issues. After several attempts, we managed to find an appropriate batch size that works for these models. While the Xavier NX platform enables training in many moderate CNN architectures, memory constraints impose clear limits on model and dataset scale versus high-end GPUs or cloud-based accelerators with abundant RAM. In summary, RAM availability represents the primary bottleneck for more advanced deep learning tasks on this embedded hardware. In our future work we will be exploring alternative optimizers and loss functions that could potentially improve convergence speed, model performance, and robustness. Additionally, leveraging hardware-specific

libraries such as Nvidia's TensorRT could also improve inference performance on the Xavier NX via strategies tailored to the GPU architecture.

Chapter 5

Conclusion

We successfully setup the NVIDIA Xavier NX, installed the OS, extended its original disk capacity by using an external SSD drive, installed all necessary libraries in order to deploy CNN models, implemented scripts to capture the board and the models metrics and plot its results. We were able to identify trade-offs in terms of performance of studied metrics and time required to train while selecting a CNN architecture on 2D and 3D datasets. We also published a paper called "Evaluating Compact Convolutional Neural Networks for Object Recognition using Sensor Data on Resource-Constrained Devices" [54], published on 15 November 2023 by MDPI in the Engineering Proceedings Journal, ECSA 2023.

5.1 Contributions

As stated in section 1.3, this thesis aimed to assess the effectiveness of different compact CNN architectures, including AlexNet, ShuffleNet, SqueezeNet, Resnet50, and MobileNetV2, for object recognition in both 2D and 3D data when trained on the NVIDIA Jetson Xavier NX. The main goals were to examine the utilization of resources, such as CPU/GPU and RAM during model training, evaluate the CNNs' performance, and identify any trade-offs. Moreover, this work aimed to bridge the gap in the literature by investigating the training of CNN models on resource-constraint platforms such as NVIDIA Jetson Xavier NX. In a nutshell, the thesis brought these specific contributions :

- Set up NVIDIA Jetson Xavier NX board in section 3.1.2
- Deployed and trained CNNs on the board in sections 4.1 and 4.2

- Evaluated performance and compared of five CNN models in on 2D and 3D data in section 4.3
- Monitored and analyzed board resource utilization in section 4.3.1
- Identified trade-offs on training CNNs on resource constrained devices in section 4.3.

5.2 Future work

We currently deployed and assessed the selected models on the NVIDIA Xavier NX board, training them on 2D and 3D data. While the current research has focused on training various CNN architectures on the CIFAR-10, STL-10, and OrganMNIST {A,C,S} datasets using the Nvidia Xavier NX platform with a standard SGD optimizer and Cross Entropy loss function, there are several promising avenues for further exploration.

First, training the models on additional hardware platforms such as Intel CPUs, Google TPUs, AMD Xilinx, or other embedded devices would provide valuable comparative data. Optimizing the neural network implementations and hyperparameter tuning specifically for each hardware backend's strengths and limitations would likely lead to performance improvements. Extensive benchmarking of accuracy, latency, throughput, power consumption, and other metrics across platforms would give essential insights into real-world deployment trade-offs.

Second, exploring alternative optimizers such as Adam, RMSprop, or AdaGrad and loss functions like softmax cross-entropy, mean absolute error, or hinge loss could potentially improve convergence speed, accuracy, or model robustness. Hyperparameter tuning per optimizer and loss function combo would extract the best possible performance. Comparing results across this matrix would shed some light on the strengths and weaknesses of different optimization strategies.

Third, as future work, we are going to explore the ability of training and assessing the performance of these models on a pure 3D dataset leveraging 3D convolutions instead of the 2D convolutions employed in this work.

Additionally, leveraging hardware-specific libraries such as Nvidia's TensorRT could significantly improve inference performance on the Xavier NX via optimized quantization, pruning, and compilation tailored to the GPU architecture. Learning rate tuning via advanced schedulers like cosine annealing, cyclical or step-wise policies, is another proven technique that could enhance convergence and accuracy. Comparing TensorRT-

optimized inference versus baseline PyTorch model performance, paired with extensive learning rate hyperparameter searches fit to each model, would provide further opportunities for optimization atop the cross-hardware and cross-software training exploration already proposed.

Moreover, training various architectures on multiple datasets besides CIFAR-10, STL-10, OrganMNIST {A,C,S} would provide much more robust validation of the approaches. Evaluating model performance across several datasets would improve generalizability and reduce the risk of overfitting. Comparing model's performance, convergence speed, overfitting tendencies, and other metrics across datasets along with the hardware, software, optimization, and learning rate search outlined earlier would deliver a holistic assessment of real-world viability. Broad dataset testing is a crucial component for ensuring the reliability and applicability of the trained computer vision models.

Bibliography

[1] Y. Al-Nasiri, M. S. Al-Hafid, and A. P. D. H. al bayaty, "Five-component load forecast in residential sector using smart methods," *Iraqi Journal for Electrical And Electronic Engineering*, vol. 18, 04 2022.

[2] H. Li, N. Parikh, J. Wang, S. Merhar, M. Chen, M. Parikh, S. Holland, and L. He, "Objective and automated detection of diffuse white matter abnormality in preterm infants using deep convolutional neural networks," *Frontiers in Neuroscience*, vol. 13, 06 2019.

[3] R. Vadlamani and A. Patel, "Deep convolutional net," 03 2021.

[4] J. Guo, Y. Wu, L. Chen, G. Ge, Y. Tang, and W. Wang, "Automatic detection of cracks in cracked tooth based on binary classification convolutional neural networks," *Applied Bionics and Biomechanics*, vol. 2022, pp. 1–12, 09 2022.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch : An imperative style, high-performance deep learning library," 2019.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets," *arXiv preprint arXiv :1704.04861*, 2017.

[8] P. K. Gadosey, Y. Li, and P. T. Yamak, "On pruned, quantized and compact cnn architectures for vision applications : An empirical study," Association for Computing Machinery, 2019.

[9] G. Huang, S. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Condensenet : An efficient densenet using learned group convolutions," 2018. CondenseNet : An Efficient DenseNet Using Learned Group Convolutions.

[10] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-excitation networks," 2019.

[11] 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 <0.5mb model size," 2016.

[12] M. Tan and Q. V. Le, "Efficientnet : Rethinking model scaling for convolutional neural networks," vol. 2019-June, 2019. EfficientNet : Rethinking model scaling for convolutional neural networks.

[13] A. Coates, A. Ng, and H. Lee, "An Analysis of Single Layer Networks in Unsupervised Feature Learning," in *AISTATS*, 2011. `https://cs.stanford.edu/~acoates/papers/coatesleeng_aistats_2011.pdf`.

[14] A. Krizhevsky, "Learning multiple layers of features from tiny images," pp. 32–33, 2009.

[15] J. Yang, R. Shi, D. Wei, Z. Liu, L. Zhao, B. Ke, H. Pfister, and B. Ni, "Medmnist v2-a large-scale lightweight benchmark for 2d and 3d biomedical image classification," *Scientific Data*, vol. 10, no. 1, p. 41, 2023.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," 2012.

[17] K. Yang, K. Qinami, L. Fei-Fei, J. Deng, and O. Russakovsky, "Towards fairer datasets : Filtering and balancing the distribution of the people subtree in the imagenet hierarchy," in *Conference on Fairness, Accountability, and Transparency*, 2020.

[18] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn : Towards real-time object detection with region proposal networks," 6 2015.

[19] "Mobilenetv2 : Inverted residuals and linear bottlenecks," pp. 4510–4520, IEEE, 2018.

[20] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco : Common objects in context," vol. 8693 LNCS, 2014.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv :1409.1556*, 2014.

[22] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD : single shot multibox detector," in *Computer Vision - ECCV 2016 - 14th*

European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), vol. 9905 of *Lecture Notes in Computer Science*, pp. 21–37, Springer, 2016.

[23] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 00, pp. 580–587, June 2014.

[24] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once : Unified, real-time object detection," 2015. cite arxiv :1506.02640.

[25] J. Frankle and M. Carbin, "The lottery ticket hypothesis : Finding sparse, trainable neural networks," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019.

[26] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," 2017. cite arxiv :1710.03740Comment : Published as a conference paper at ICLR 2018.

[27] 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 27 : Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 1269–1277, 2014.

[28] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015. cite arxiv :1503.02531Comment : NIPS 2014 Deep Learning Workshop.

[29] A. Ravikumar, H. Sriraman, P. M. S. Saketh, S. Lokesh, and A. Karanam, "Effect of neural network structure in accelerating performance and accuracy of a convolutional neural network with gpu/tpu for image analytics," *PeerJ Computer Science*, vol. 8, 2022.

[30] V. Sharma, G. K. Gupta, and M. Gupta, *Performance Benchmarking of GPU and TPU on Google Colaboratory for Convolutional Neural Network*. 2021.

[31] J. Lee, T. Won, T. K. Lee, H. Lee, G. Gu, and K. Hong, "Compounding the performance improvements of assembled techniques in a convolutional neural network," 1 2020.

[32] M. Meyer and G. Kuschk, "Automotive radar dataset for deep learning based 3d object detection," 2019.

[33] A. Ajit, K. Acharya, and A. Samanta, "A review of convolutional neural networks," 2020.

[34] O. Kopuklu, N. Kose, A. Gunduz, and G. Rigoll, "Resource efficient 3d convolutional neural networks," 2019.

[35] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4 : Optimal speed and accuracy of object detection," 4 2020.

[36] A. Simonelli, S. R. R. Bulò, L. Porzi, M. López-Antequera, and P. Kontschieder, "Disentangling monocular 3d object detection," 5 2019.

[37] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving ? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[38] T. Huang, T. Luo, and J. T. Zhou, "Adaptive precision training for resource constrained devices," 2020.

[39] Y. Liu, Y. Yixuan, and M. Liu, "Ground-aware monocular 3d object detection for autonomous driving," 2 2021.

[40] "Transfer learning for medical image classification : a literature review," 2022.

[41] Y. Okochi, H. Rizk, T. Amano, and H. Yamaguchi, "Object recognition from 3d point cloud on resource-constrained edge device," *2022 18th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 369–374, 2022.

[42] B. Jablonski, D. Makowski, P. Perek, P. N. V. Nowakowski, A. P. Sitjes, M. Jakubowski, Y. Gao, and A. Winter, "Evaluation of nvidia xavier nx platform for real-time image processing for plasma diagnostics," *Energies*, vol. 15, 2022. Evaluation of NVIDIA Xavier NX Platform for Real-Time Image Processing for Plasma Diagnostics.

[43] Y. Kortli, S. Gabsi, L. F. Y. Voon, M. Jridi, M. Merzougui, and M. Atri, "Deep embedded hybrid cnn–lstm network for lane detection on nvidia jetson xavier nx," *Knowledge-Based Systems*, vol. 240, p. 107941, 3 2022.

[44] NVIDIA, "Nvidia jetson developer guide." `https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide%2Fgetting_started.html`, 2023. Accessed on July 11th, 2023.

[45] Etcher, "Etcher image flasher." https://etcher.download/, 2023. Accessed on August 31th, 2023.

[46] Microsoft, "Visual studio code." https://code.visualstudio.com/, 2024. Accessed on February 26th, 2024.

[47] Docker, "What is a container ?." https://www.docker.com/resources/what-container/, 2023. Accessed on August 31th, 2023.

[48] NVIDIA, "Containers for deep learning frameworks user guide." https://docs.nvidia.com/deeplearning/frameworks/user-guide/index.html, 2023. Accessed on August 31th, 2023.

[49] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch : An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow : A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[51] S. L. AB, "Pil - python image library." https://web.archive.org/web/20200914010747/https://effbot.org/imagingbook/, 2024. Accessed on March 5th, 2024.

[52] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[53] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn : Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[54] I. Camelo and A. Cretu, "Evaluating compact convolutional neural networks for object recognition using sensor data on resource-constrained devices," in *Proceedings of the 10th International Electronic Conference on Sensors and Applications*, (Basel, Switzerland), MDPI, November 15–30, 2023 2023.