

الجمهورية الجزائرية الديمقراطية الشعبية
DEMOCRATIC AND POPULAR REPUBLIC OF ALGERIA
وزارة التعليم العالي و البحث العلمي
Ministry of Higher Education and Scientific Research

ÉCOLE NATIONALE POLYTECHNIQUE



Electronics Engineering Department

Final Year Project Dissertation
in partial fulfilment of the requirements for:
Electronics Engineer's Degree

Deep Neural Networks Optimization for Embedded Platforms

Authors:

Abderrahim BENAOUA and Anouar LAOUICHI

Presented and defended in July 8th, 2020 before the members of jury:

President	Adel	BELOUCHRANI	Prof.	ENP, Algiers
Supervisor	Sid Ahmed	BERRANI	PhD.	ENP, Algiers
Co-supervisor	Hamza	YOUS	PhD.	USTHB, Algiers
Examiner	Nesrine	BOUADJENEK	PhD.	ENP, Algiers

ENP 2020

École Nationale Polytechnique (ENP)
10, Avenue des Frères Oudek, Hassen Badi, BP. 182, 16200 El Harrach, Alger, Algérie
www.enp.edu.dz

الجمهورية الجزائرية الديمقراطية الشعبية
DEMOCRATIC AND POPULAR REPUBLIC OF ALGERIA
وزارة التعليم العالي و البحث العلمي
Ministry of Higher Education and Scientific Research

ÉCOLE NATIONALE POLYTECHNIQUE



Electronics Engineering Department

Final Year Project Dissertation
in partial fulfilment of the requirements for:
Electronics Engineer's Degree

Deep Neural Networks Optimization for Embedded Platforms

Authors:

Abderrahim BENAOUA and Anouar LAOUICHI

Presented and defended in July 8th, 2020 before the members of jury:

President	Adel	BELOUCHRANI	Prof.	ENP, Algiers
Supervisor	Sid Ahmed	BERRANI	PhD.	ENP, Algiers
Co-supervisor	Hamza	YOUS	PhD.	USTHB, Algiers
Examiner	Nesrine	BOUADJENEK	PhD.	ENP, Algiers

ENP 2020

ملخص - إن هذا المشروع يقتضي بتحسين الشبكات العصبية العميقة لاستدلال أكثر فعالية في الأنظمة المضمنة. أنشأت لهذا الغرض تقنيتي *Pruning* و *Quantization* الشبكات في بيئة *PyTorch* و تم قياسها على *ResNet50*. النتائج المتحصل عليها و المتمثلة في معدلات الضغط و السرعة تؤكد مدى صحة المبدأ. ولإظهار إمكانيتهما العملية، تم تطبيق التقنيتين على كاشف أشياء *RetinaNet*. يوضح هذا العمل أنه يمكن إجراء الاستدلال على الحافة من خلال تقليل مدة المعالجة و مساحة الذاكر المشغلة، و الذي ينتج عنه خفض زمن الاستجابة و الطاقة المستهلكة إضافة إلى رفع درجة خصوصية البيانات. يؤدي هذا إلى فتح آفاق لتطبيقات ثورية في الأنظمة المضمنة

كلمات مفتاحية - الذكاء الاصطناعي، الشبكات العصبية العميقة، الأنظمة المضمنة، الاستدلال، *Pruning*، *Quantization*، كشف أشياء، *PyTorch*.

Résumé—Ce projet porte sur l’optimisation des réseaux de neurones profonds pour une inférence embarquée efficace. Pour cela, les techniques d’élagage et de quantification des réseaux ont été implémentées dans l’environnement PyTorch, et testées sur ResNet50. Les résultats obtenus en terme de taux de compression et de vitesse de traitement, valident avec succès la faisabilité du concept. Afin de montrer leur intérêt applicatif, ces deux techniques ont été appliquées sur le détecteur d’objets RetinaNet. Ce travail montre que l’inférence peut être effectuée localement en réduisant le temps d’exécution et l’espace mémoire occupé par le modèle, ce qui permet une diminution de la latence et de la consommation d’énergie. Il permet également de contribuer à une meilleur protection des données, et ce, en les traitant localement. Ce travail ouvre ainsi des perspectives intéressantes pour les applications systèmes embarqués.

Mots-clés : Intelligence artificielle, réseaux de neurones artificiels profonds, systèmes embarqués, inference, Elagage, Quantification, détection d’objet, PyTorch.

Abstract—This project deals with the optimization of Deep Neural Networks for efficient embedded inference. Network Pruning and Quantization techniques are implemented under the PyTorch environment and benchmarked on ResNet50. The obtained results, consisting of compression and speed-up rates, successfully validate the feasibility and the effectiveness of the concept. To show their practical potential, the two schemes have been applied on RetinaNet object detector. Additionally, this work demonstrates that inference can be performed at the edge by reducing the model’s memory footprint and the processing time, resulting in reduced latency and energy consumption as well as improved data security. Hence, new horizons of applications in embedded systems are opened up.

Index-terms : Artificial intelligence, Deep Neural, Embedded Systems, Inference, Networks, Pruning, Quantization, Object detection, Pytorch.

Acknowledgement

First and foremost, we would like to offer sincere thanks to our supervisor Mr Sid Ahmed BERRANI, Associate Professor with Ecole Nationale Polytechnique, who has been an exceptional supervisor, and we have been very fortunate to receive his guidance throughout this project. In each step, Mr BERRANI gave us extremely visionary advice and most generous support. He taught us, the methodology of a researcher, and the way to understand the scope of any research we tackle. We are honored to say he is one of our best professors and was a very good advisor as well.

We would like to express our profound gratitude to Ph.D Hamza YOUS, data scientist at Intel corp, for his availability and for his invaluable support. We had the unique privilege to have access to his professional expertise and brilliant thinking. The great advice he offered us, have dramatically helped us to learn a great deal while completing our project.

We give special thanks to Ph.D Sofiane YOUS, Lead scientist at Intel corp, for introducing us to the topic as well for the support along the way. We would like to thank him for giving us the chance to engage in a cutting edge research.

We would like to thank the committee members for reviewing our dissertation and for their valuable and insightful comments.

Dedication

We would like to dedicate our work to our parents and loved ones, who always helped us throughout the entire educational pathway. They always stood alongside us during tough moments, offering us their wholehearted support and elating words of encouragement.

We offer this dissertation to all our professors at the Electronics Department at Ecole Nationale Polytechnique, who had spared no effort to teach us, and who have been always supportive and available.

We also dedicate this project to our friends, both from the school and outside the school. All those who have supported us throughout the process. We will always appreciate everything they have done.

Contents

Abstract

Acknowledgement

Dedication

List of Figures

General Introduction	10
1 Deep Neural Networks	13
1.1 Introduction	13
1.2 Neural Networks	13
1.2.1 The Artificial Neuron	14
1.2.2 Activation Functions of Artificial Neurons	15
1.2.3 Feed Forward Neural Networks	17
1.2.4 Training the Neural Network	18
1.2.5 Testing the Neural Network	20
1.2.6 Evaluation Metrics	21
1.3 Convolutional Neural Networks	23
1.3.1 Convolutional Layer	23
1.3.2 Batch Normalization	27
1.3.3 Pooling	28
1.3.4 Putting it all together	28
1.4 Residual Neural Networks	29
1.5 Datasets	32
1.5.1 ImageNet	32
1.5.2 CIFAR10	34
1.6 Pytorch Deep learning framework	35
1.7 Conclusion	35

2	Optimization Methods	36
2.1	Introduction	36
2.2	Software Optimizations for Embedded Systems	36
2.2.1	Prior to Optimization in the Compiler	37
2.2.2	Optimizing Software for Performance	38
2.2.3	Optimizing Software for Memory	40
2.2.4	Optimization with Pytorch mobile	43
2.3	Neural Network Pruning	46
2.3.1	Filter pruning	49
2.3.2	Pruning Scheme	51
2.3.3	Experiments and Results	54
2.4	Neural Network Quantizaion	58
2.4.1	Quantization scheme	60
2.4.2	Quantization modes in Pytorch	63
2.4.3	Experiments and Results	66
2.5	Conclusion	67
3	Application and Deployment	68
3.1	Introduction	68
3.2	COCO dataset	70
3.3	RetinaNet	71
3.3.1	The Backbone	71
3.3.2	Classification Sub-network	72
3.3.3	Box Regression Sub-network	72
3.4	RetinaNet-50 Optimization	73
3.5	Mobile application development	75
3.5.1	Android General Concepts	75
3.5.2	Profiling the application	79
3.6	Conclusion	81
	Conclusion	83

List of Figures

1.1	An artificial neural network with N hidden Layers [9].	14
1.2	Artificial neuron in a deep neural network [5].	15
1.3	Graph of the Sigmoid function [71].	15
1.4	Graph of the <i>tanh</i> function [29].	16
1.5	Graph of the ReLu function [18].	17
1.6	A feed-forward neural network with 3 layers [5].	18
1.7	Visualization of the data splits [70].	21
1.8	2D-convolution operation.	24
1.9	(a) : fully connected layer (b): locally connected layer (convolutional) [46]. . . .	24
1.10	Examlpe of zero padding [72].	25
1.11	Generation of output feature maps by convolving filters with input feature maps. .	26
1.12	Top-right: max pooling, bottom-right: average pooling [67].	29
1.13	Convolutional neural network architecture [8].	29
1.14	Structure of a residual block: a direct path, and an identity mapping [23].	30
1.15	(a): bottleneck with no expansion; (b): bottleneck with expansion [8].	31
1.16	Architecture of ResNet50 [53].	32
1.17	Example of a hierarchy of the concept: Carnivore.	33
1.18	Classes that belong to the <i>Canine</i> synset.	33
1.19	Sample images of the CIFAR-10 dataset [1].	34
2.1	The optimisation process.	37
2.2	Aliasing: Two pointers that store the same memory address.	39
2.3	Examples of control graphs; nodes represent basic blocks, edges represent flow of control [76]	41
2.4	left: traditional matrix multiplication; right: QNNPACK matrix multiplication. [11]	44
2.5	Addition of the indirection buffer [11].	45
2.6	Depthwise 3×3 convolution in QNNPACK [11].	46
2.7	Weights and Neurones Pruning in a fully-connected network.	47
2.8	Transforming to compressed sparse format.	47

2.9	Two different structures of sparsity.	48
2.10	Accuracy-Sparsity Curve of AlexNet with different grain sizes. X-axis: sparsity of conv layers (percentage of zero weights). Y-axis: top-5 accuracy on ImageNet validation set [49].	49
2.11	Illustration of data-dependency in filter pruning (side view).	50
2.12	Parallel Conv layers Pruning (front view).	51
2.13	Iterative Pruning.	54
2.14	Illustration of 1st and 2nd trials, (Batch Normalization and ReLu are not presented for simplicity).	55
2.15	Top-1 Accuracy on CIFAR10 Test set against Pruning rate.	55
2.16	Inference time for a batch size of 200 against Pruning rate.	56
2.17	Model size against Pruning rate.	57
2.18	Top-1 Accuracy on CIFAR10 Testset against Pruning rate.	57
2.19	Weight distribution for GoogleNet [19].	58
2.20	Converting real values to integer values [62].	60
2.21	Floating point.	62
2.22	Fixed point.	62
2.23	Convolutional layer: Before and After Graph Transformation [33].	62
2.24	Illustration of a quantized computation graph.	63
2.25	Training with simulated quantization [33].	64
2.26	Optimized matrix on matrix multiplication.	65
3.1	Example of object detection and segmentation [50]	69
3.2	An image with bounding boxes from COCO dataset [44].	70
3.3	RetinaNet network architecture [43].	72
3.4	RetinaNet network architecture [43].	73
3.5	Retinanet-50 - iterative pruning: Precision/Recall.	73
3.6	Size reduction through Iterative pruning.	74
3.7	Activity lifecycle.	76
3.8	Android project organization.	77
3.9	The View class and its children.	78
3.10	Profiling memory usage on a smartphone	80
3.11	CPU usage profiling, the application uses 4 threads to perform inference, CPU peaks at 65 %	80
3.12	(a): Launch screen. (b): Image loaded. (c): Inference performed.	81

Acronyms

AI Artificial Intelligence. 10

AUC Area Under Curve. 21

BFLOPS Billion Floating Point Operation per Second. 10

CIFAR Canadian Institute for Advanced Research. 34

CNN Convolutional Neural Networks. 28

COCO Common Object in Context. 70

DNNs Deep Neural Networks. 10

DRAM Dynamic Random Access Memory. 10

IoT Internet of Things. 10

QNNPACK Quantized Neural Networks PACKage. 43

ReLU Rectified Linear Unit. 16

SRAM Static Random Access Memory. 11

YOLO You Only Look Once. 70

Introduction

Deep Neural Networks (DNNs) are nowadays widely employed in a variety of Artificial Intelligence (AI) applications, including computer vision, natural language processing, robotics, and so on. Whereas DNNs achieve state-of-the-art accuracy on many machine learning tasks, it comes at the cost of high computation and memory usage. As a result, it is quite challenging to deploy computing systems like these DNNs on embedded devices such as smartphones, drones, self-driving cars, and other IoT devices.

Generally, deep learning can be carried out in the cloud or by utilizing high-performance computing platforms, often relying on multiple graphics cards to accelerate the process. Inference can be performed in the cloud as well, which works fine for non-time-critical workflows. However, it is now commonly being carried out on a device local to the data being analyzed, which is known as inference at the edge.

The motivation for performing inference computation at the edge, where the data resides, is that sending data across the network is expensive and time-consuming. This incurred latency and cost limit the user experience that could be delivered to the consumer. Removing these barriers opens up new applications that would otherwise not be possible. Clearly, for real-time applications such as the detection of defective products in a production line (anomaly detection), it is important that the result is generated as quickly as possible so that the faulty product can be quickly discarded. Data security is another area that can benefit from performing inference at the edge; removing the necessity to transmit personal data across the network reduces the chances of a malicious user from intercepting it.

That being said, bringing all the computation to end devices comes with numerous challenges, because of the limited computational resources of the devices. For example, a modern object detection model like YOLO-v3 is 237MB in size, and requires the hardware to deliver 56.45 BFLOPS to run at 30 frames/second [58]. Running such a large neural network requires a lot of memory bandwidth to fetch the weights. These weights usually do not fit in on-chip memory. Hence, they require costly DRAM accesses. To give an insight into energy consumption, take a device

that runs on 45nm CMOS technology. In this device, a 32-bit floating-point addition consumes 0.9pJ, a 32bit SRAM cache access takes 5pJ, while a 32bit DRAM memory access consumes 640pJ. The latter type of access is 3 orders of magnitude of an add operation. Running a neural network with 1 billion connections, for example, at 20fps would require $(20\text{Hz})(1\text{G})(640\text{pJ}) = 12.8\text{W}$ just for DRAM access [19], on top of that, extensive arithmetic operations also contribute their fair share in energy consumption. Embedded systems can hardly afford all these demands, and in many cases, it is beyond their capabilities.

Despite those challenges and constraints, we have witnessed rapid progress in the area of Deep Neural Networks optimization. In this dissertation, we investigate how to simplify and compress DNN models to make them less computation and memory intensive. The most popular techniques are Pruning and Quantization. They have been able to achieve surprisingly high compression rates and a stunning speed-up with minimal accuracy drop. Plenty of works focus on both techniques (i.e. pruning and quantization). Han et al. [61] implemented both techniques in addition to *Huffman coding*, however their approach requires specialized hardware to work effectively. Other works introduce more sophisticated variants of pruning, notably those proposed by He et al. [25] and Lin et al. [24]. Quantization has its fair share of proposals as well. The most outstanding approach is the one published by engineers from Google Jacob et al. [30] that is widely adopted. Some quantization approaches do however present a number of practical downsides [30]. The aim of this project is to explore previous researches conducted in this field of study and effectively implement Pruning and Quantization, then apply them on a modern DNN model. The model will then be deployed on an embedded system.

The main contributions of our work can be summarized as follows:

1. Implement two well-established techniques in the field of deep neural networks optimization, i.e. Network pruning and quantization
2. Apply these above-mentioned techniques on two contexts: Image classification and object detection.
3. Compare the performance of the baseline model and the fully optimized model on a smart-phone (typical embedded device)

This dissertation is organized as follows:

- Chapter I provides a background on the context of why DNNs are important, their history, and applications. It gives an overview of the basic components of DNNs, as well as a brief presentation of the utilized datasets and the framework which we used for implementation.

- Chapter II describes software optimization methods for embedded systems along with the two aforementioned techniques explained in meticulous detail. It also shows the achieved results on the optimization of ResNet50 trained on the CIFAR10 dataset.
- Chapter III gives an overview of object detection, then presents the targeted model, RetinaNet, as well as the dataset it is trained on. Afterward, it describes the mobile application development workflow that is used to deploy the backbone model of the object detector (ResNet50) on a smartphone. Finally, it shows the inference results of the original and the optimized model on the target phone.
- The conclusion lays out a brief summary of what this dissertation is about. In addition, it explores the challenges that we faced during implementation. It also explores contributions that could be made in future works.

Chapter 1

Deep Neural Networks

1.1 Introduction

The aim of this chapter is to acquaint the reader with the fundamental concepts that constitute the field of deep learning. This area of research is becoming increasingly fruitful as it is paving the way for modern machine learning. The information is presented such that each concept builds on its previous counterparts, going from the single artificial neuron and the biological analogy to feedforward and convolutional networks. The models of great interest to us are those that have a deep convolutional architecture like ResNet50. Deep networks have more parameters and therefore, take more space in memory, which gives a good incentive to compress them. Convolutional layers are also computationally demanding, which gives a good incentive to accelerate operations that they perform.

1.2 Neural Networks

Artificial Neural Networks (ANNs) are computing entities that mimic the way biological nervous systems operate. This similarity can be linked to two main aspects of ANNs. The first one is based on the observation that much like the nervous system in living beings, artificial neural networks are comprised of a collection of interconnected computational nodes. These nodes are also known as artificial neurons; these neurons exchange and process signals between one another. The second aspect resides in the fact that the network is able to learn and adapt its behavior according to the input, in order to get as close as possible to the desired output, just like their biological counterparts.

Out of the huge number of nodes that the network consists of, the ones found at the extremities are the most notable to us. On one side, the input units are located. They are designed to receive different forms of information signals from the outside environment. In the training phase, the network will attempt to learn about these signals and extract residing patterns. The response of

the network will emerge on the other side, i.e. the output units. In between these extremities, the rest of the units constitute what is known as the hidden layers, which form the majority of the neural network. These hidden layers are what makes the network deep. This apparent characteristic of the network is what popularised the term "Deep Learning" which is the field of study associated with everything related to ANNs. Figure 1.1 below illustrates the general structure of a plain neural network.

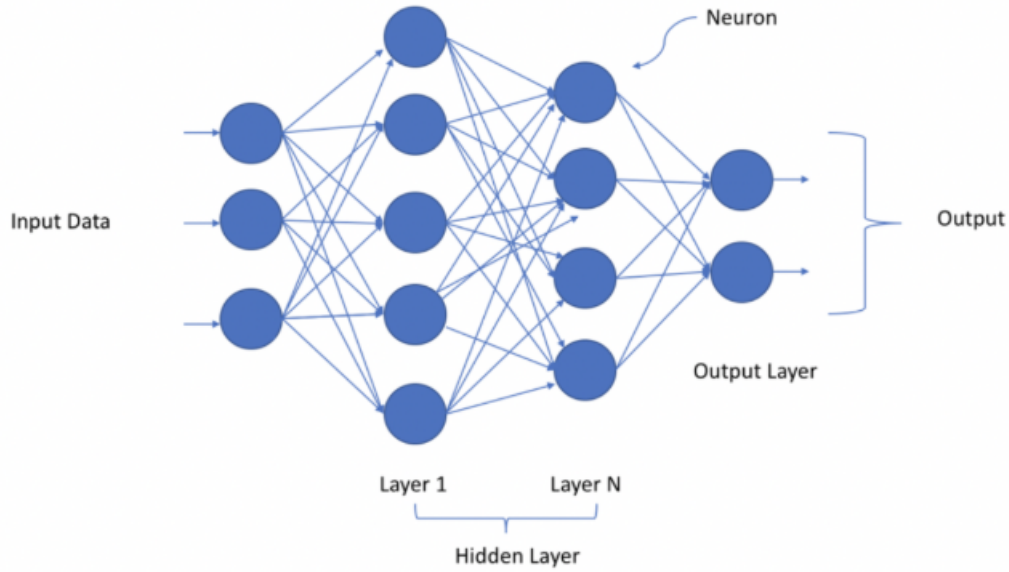


Figure 1.1: An artificial neural network with N hidden Layers [9].

1.2.1 The Artificial Neuron

The understanding of the functional behavior of biological neurons can be relayed to computing systems in the form of what is known as the artificial neuron. To put it bluntly, an artificial neuron is an object that can be represented in our computers and with which, far more complex systems can be built. The artificial neuron can be thought of as a point with multiple connections that incorporates a function. This function takes the inputs, $x_1, x_2, x_3, \dots, x_n$, scaled with the weights $w_1, w_2, w_3, \dots, w_n$ respectively. These weighted inputs are added together along with a constant bias b to produce z .

$$z = \sum_{i=1}^n w_i x_i + b \quad (1.1)$$

z is then passed to an activation function to produce the output of the neuron, y .

$$y = f(z) \quad (1.2)$$

The activation function f illustrated in Figure 1.2 can be linear or nonlinear. In the linear case it

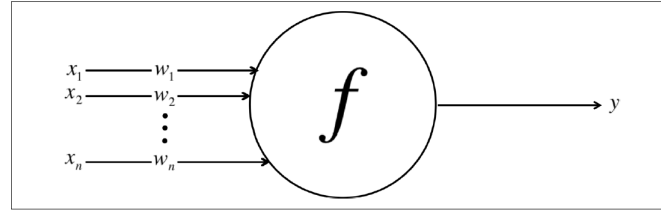


Figure 1.2: Artificial neuron in a deep neural network [5].

can be expressed as:

$$f(x) = ax + b \quad (1.3)$$

where a is the slope, and b is the offset (or bias). In the nonlinear case, multiple functions have been proposed each with its advantages and drawbacks.

1.2.2 Activation Functions of Artificial Neurons

The Sigmoid activation function

This function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1.4)$$

The Sigmoid function is differentiable and monotonic, but with a non-monotonic derivative. It maps the input x to the interval $[0; 1]$, where the smaller the value of x gets, the closer it is to zero. Conversely, the bigger it gets, the closer it is to 1. Figure 1.3 depicts a representation of the Sigmoid function.

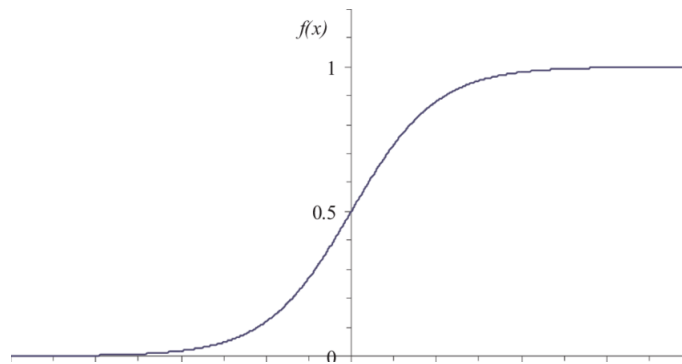


Figure 1.3: Graph of the Sigmoid function [71].

Tanh function

The tanh function shares some similarities with the Sigmoid function. It is also differentiable and monotonic, and with a non-monotonic derivative. The interval it maps to is $[-1; 1]$ and it has the expression:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \quad (1.5)$$

The graph of this function is s-shaped and is represented in Figure 1.4.

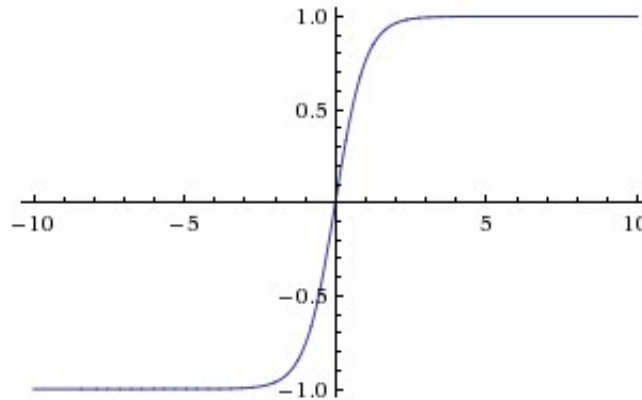


Figure 1.4: Graph of the *tanh* function [29].

ReLU function

Rectified Linear Unit (ReLU) is a piece-wise linear function (Figure 1.5) that has 0 output if the input is negative, and the identity output otherwise. This type of activation is preferred in many models due to the fact that it has a non-saturating gradient, meaning that it does not have the back-propagation errors that other activation functions like Sigmoids suffer from. On top of that, the calculations performed during both the training and inference phases are much faster. This is because the expression of the ReLU function is simple and does not involve complex mathematical operations like exponential functions (for the Sigmoid).

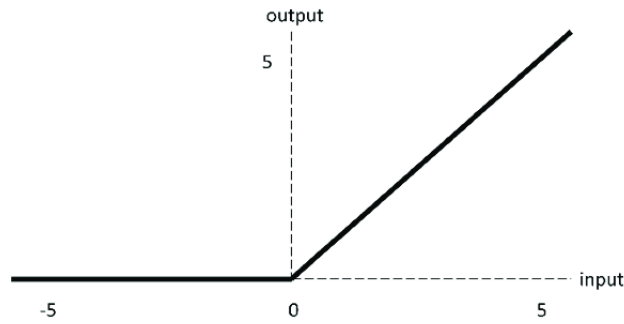


Figure 1.5: Graph of the ReLu function [18].

1.2.3 Feed Forward Neural Networks

In order to be able to treat complex real-life problems, a single neuron will clearly not suffice. If one looks at the architecture of the artificial neural network, one can easily deduce that it would be impossible for it to differentiate between complex patterns, like those of handwritten digits, if it was comprised of one neuron only. So to tackle these kinds of problems, the artificial neuron is used as a building block for a more complex system. This system is constructed by grouping multiple neurons in regular structures known as layers, and then stacking these layers against each other.

The first layer takes in the input information, and the last layer computes the output. The layers found in the middle who are hidden have the role of extracting patterns. These patterns get increasingly more complex as we move forward along the network, up until we reach the output layer.

In Figure 1.6, we observe that between the i^{th} neuron in the k^{th} layer and the j^{th} neuron in the $k + 1^{st}$ layer, there is a connection that has a weight of $w_{i,j}^{(k)}$ associated with it. This model is called a feed-forward neural network because the information moves from one layer to the next layer only. There are no connections between neurons in the same layer, nor are there connections that transmit data from a higher layer to a lower one.

Softmax activation function

The Softmax activation function outputs a vector representing a probability distribution over a set of mutually exclusive labels. It differs from other activation functions that we have seen before because the output of each individual neuron in the layer associated with it depends on the output of the rest of the neurons of the same layer. This dependency is the result of the law of probability that requires the sum of all outputs to be equal to 1. This is achieved by normalizing

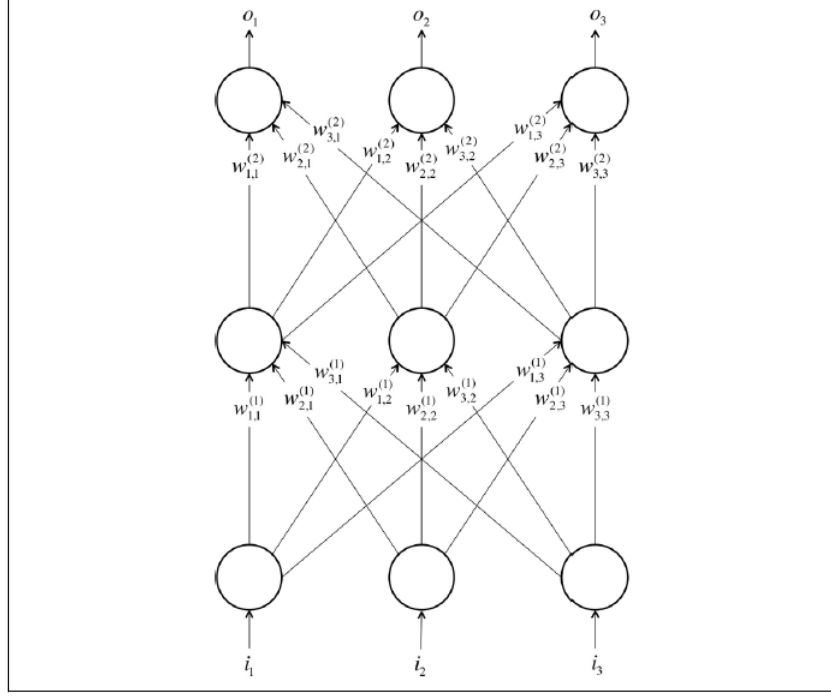


Figure 1.6: A feed-forward neural network with 3 layers [5].

z and setting the output of the neuron to:

$$y_{oi} = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad (1.6)$$

A good prediction would have one of the entries of the output vector close to 1, whereas the remaining entries are close to 0. Contrariwise, a bad prediction would have multiple possible labels roughly equal to the same value. Therefore, Softmax is used specifically for the output layer of the network.

1.2.4 Training the Neural Network

During the training process, the values of the parameters are gradually learned. The model parameters, in this case, refer to the weights and biases of each layer. The first phase is the initialization, where the parameters are assigned initial values. The most trivial method is to initialize the parameters randomly. Though, this technique is known to slow down or even stall the convergence process [51]. Many initialization methods have been proposed to circumvent this problem [51], [63], [68]. The next phase, called forward propagation, occurs when the network gets exposed to a predetermined amount of training data (batch). The data traverses the entire neural network until the final layer is reached, at which point, label predictions are delivered. With these outputs, the loss function that models the error is calculated. The calculation is

performed based on a given expression (Cross entropy, mean squared error, etc). After that, an optimization algorithm is applied to adjust the parameters of the model so as to minimize the error. The algorithm works by computing the gradient of the loss function with respect to each weight, computing the gradient one layer at a time, propagating backward from the last layer up to the first layer. This phase is called the back-propagation phase. When all the weights and biases have been successfully updated, the whole process is repeated for another incoming batch, until all the training dataset has been entirely traversed, in which case, one epoch of training is achieved.

Optimization Algorithms for Deep Learning

The entire premise of model training relies on optimization algorithms, these are used to adjust the parameters of the network such as weights and biases, with the objective of reducing the error in the output. The errors are modeled by a loss function $L(\theta, \Gamma)$ where Γ is the complete training dataset, and θ is a vector representing the parameters to optimize. The end goal is to find the values of the parameter vector θ that would minimize the value of the loss function. This is mathematically described as:

$$\theta = \arg \min_{\theta} L(\theta, \Gamma) \quad (1.7)$$

The way weights are updated is different depending on the optimization algorithm used.

Vanilla Gradient Descent

This optimization algorithm is the most basic among optimizers, but also the most widely used. This is especially true for linear regression and classification problems. It is a first-order optimizer which depends on the first-order derivative of the loss function. The model variable vector θ of the current iteration τ is calculated as follows:

$$\theta^{(\tau)} = \theta^{(\tau-1)} - \eta \cdot \nabla_{\theta} L(\theta^{(\tau-1)}; \Gamma) \quad (1.8)$$

In the equation above, $\nabla_{\theta} L(\theta^{(\tau)}; \Gamma)$ is the first-order derivative of the loss function with respect to θ based on the complete training set, and η denotes the learning rate. This iterative process is perpetuated until convergence over a minimum is achieved.

Stochastic Gradient Descent

One of the cons of the vanilla gradient descent algorithm is that the loss function is computed over the complete training data set, which is very slow and inefficient if the dataset is large.

To circumvent this problem, the stochastic gradient descent updates the model's parameter vector more frequently. In this algorithm, the parameter vector is altered after the loss function is

calculated in each training example. This way, the stochastic gradient descent can be much quicker than its vanilla counterpart. But it comes at the cost of heavy fluctuations of the loss function throughout the updating process. The updating process is described as follows:

$$\theta^{(\tau)} = \theta^{(\tau-1)} - \eta \cdot \nabla_{\theta} L(\theta^{(\tau-1)}; (x_i, y_i)) \quad (1.9)$$

where $L(\theta^{(\tau)}; (x_i, y_i))$ designates the loss introduced by the data sample (x_i, y_i) .

As mentioned previously, the stochastic gradient descent may fluctuate on multiple occasions, but this can sometimes be helpful, in the sense that it could give this algorithm the ability to jump out of a local minimum. A good way to approach this is to manually select a small learning rate η and gradually decrease its value as more epochs are completed. This way, the convergence to the local or global minimum is much more likely.

Mini-batch Gradient Descent

This algorithm can be thought of as middle-ground between the vanilla gradient descent and the stochastic gradient descent. In this instance, the model parameters are updated in each mini-batch B which is a subset of the complete dataset. Formally,

$$\theta^{(\tau)} = \theta^{(\tau-1)} - \eta \cdot \nabla_{\theta} L(\theta^{(\tau-1)}; B) \quad (1.10)$$

The Concept of Momentum

To reduce high variance and soften the convergence in the stochastic and mini-batch gradient descent algorithms, Momentum has been introduced [54]. It accelerates the convergence towards the relevant direction and reduces the fluctuation that is irrelevant. Momentum is mathematically modeled with the help of one more hyper-parameter ρ and the parameter vector is updated within each iteration as follows:

$$\theta^{(\tau)} = \theta^{(\tau-1)} - \eta \cdot \Delta V^{(\tau)} \quad (1.11)$$

where $\Delta V^{(\tau)} = \rho \Delta V^{(\tau-1)} + (1 - \rho) \cdot \nabla_{\theta} L(\theta^{(\tau-1)})$ and τ is the current iteration.

1.2.5 Testing the Neural Network

Before the training process, the available dataset is split into three sets: The training set, the validation set, and the test set (Figure 1.7).



Figure 1.7: Visualization of the data splits [70].

Train dataset

This is the actual dataset that is used for the training process explained in Section 1.2.4. Batches of this dataset are fed to the model to adjust its parameters and reduce the training error as much as possible.

Validation dataset

This dataset is used to provide an unbiased evaluation of a model trained on the training dataset. The validation process can be performed multiple times while varying the model's hyperparameters, so as to improve its effectiveness even more. The evaluation on this dataset becomes more biased as more training-validation cycles are performed to find the optimal hyperparameters.

Test dataset

The test dataset provides the gold standard for model evaluation. It is only used once a model is completely trained under optimal hyperparameter values (that result in the smallest error on validation set). The test set is generally carefully selected. It contains samples that span the various classes that the model would face when used in a real-world scenario.

1.2.6 Evaluation Metrics

After preparing and implementing a deep neural network, using whatever approach suited for the targeted classification problem. The next step is to evaluate how effective the model is based on some metric using test datasets. There are different performance metrics used for classification problems such as Accuracy, Log-Loss, AUC (Area Under Curve), precision, recall, etc. The choice of metrics to evaluate the effectiveness of the model is very important because it can give us some insights regarding how the model stacks up with respect to the other models. These metrics also rely on the type of problem at hand.

Confusion Matrix

A confusion matrix is used in describing a classification model in terms of its performance, which is measured on a set of testing data for which the true results are established and defined by the experts.

The confusion matrix describes the performance of a classifier on a set of test data for which the true values are known. If we take an example of a binary classifier for the sake of simplicity (though, it can easily be extended to more than two classes) which predicts two classes, either positive (1) or negative (0). To measure the performance of such a classifier, it is important to distinguish four basic terms:

1. **True Positives (TP)**: Refer to the number of cases in which the actual class of the input data was 1 (True) and the prediction of the network is 1 (True) as well.
2. **True Negatives (TN)**: Refer to the number of cases in which the actual class of the input was 0 (False) and the prediction of the network is 0 (False) as well.
3. **False Positives (FP)**: Refer to the number of cases in which the actual class of the input was 0 (False) but the predicted value by the network is 1 (True).
4. **False Negatives (FN)**: Refer to the number of cases when the actual class of the data point was 1 (True) but the predicted is 0 (False).

Table 1.1 gives a concise illustration of how a confusion matrix can be represented:

Table 1.1: Confusion matrix for two classes.

	Predicted: class 0	Predicted: class 1
Actual: class 0	TN	FP
Actual: class 1	FN	TP

Ideally, the model ought to give 0 cases of false positives and 0 cases of false negatives. Which is equivalent mathematically to a diagonal confusion matrix.

Accuracy is one of the metrics that are often used, and it represents the rate of correctness of a neural network, it is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + FN + TN + FP} \quad (1.12)$$

Precision represents the percentage of the results that are relevant and are correctly classified by the neural network.

$$Precision = \frac{TP}{TP + FP} \quad (1.13)$$

Recall represents the relative number of total relevant results that are correctly classified by the neural network.

$$Recall = \frac{TP}{TP + FN} \quad (1.14)$$

Sometimes a higher priority is given to maximizing precision, while at other times, maximizing recall is more important. But to include both of these metrics in a single quantity for ease of comparison between models, the F1-score has been proposed. It is the harmonic mean of precision and recall:

$$F1\text{-Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (1.15)$$

1.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of deep neural networks characterized by their shift, scale, and distortion invariance [38]. They have been proven to be excellent in real-world problems involving computer vision, and natural language processing [34], [64], [79]. These are attributable to their architecture that inherently incorporates local connectivity, shared weights, and sub-sampling. These properties are achieved through the use of a combination of convolutional layers, pooling layers, as well as fully connected layers.

1.3.1 Convolutional Layer

The convolution layer is a fundamental building block of the CNN model. Its main function is feature extraction. It takes an input in the form of a tensor^a known as input feature maps, and produces a tensor of activations (output feature maps). Typically, it combines linear and nonlinear operations, namely, a convolution operation followed by an activation function.

2D-Convolution

The 2D-convolution is a fairly simple operation. It starts with a kernel, which is simply a small matrix of weights. This kernel "slides" over the 2D input data, performing an element-wise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel. This operation is often referred to as a scalar product.

The kernel repeats this process for every location it slides over, converting a 2D matrix of features into yet another 2D matrix of features. The output values are essentially the weighted sums (with the weights being the values of the kernel itself) of the inputs taken from its receptive-field^b. Figure 1.8 shows an example of a 2D-convolution.

^aN-dimensional array of values.

^bthe region of the input space that affects a particular the output pixel.

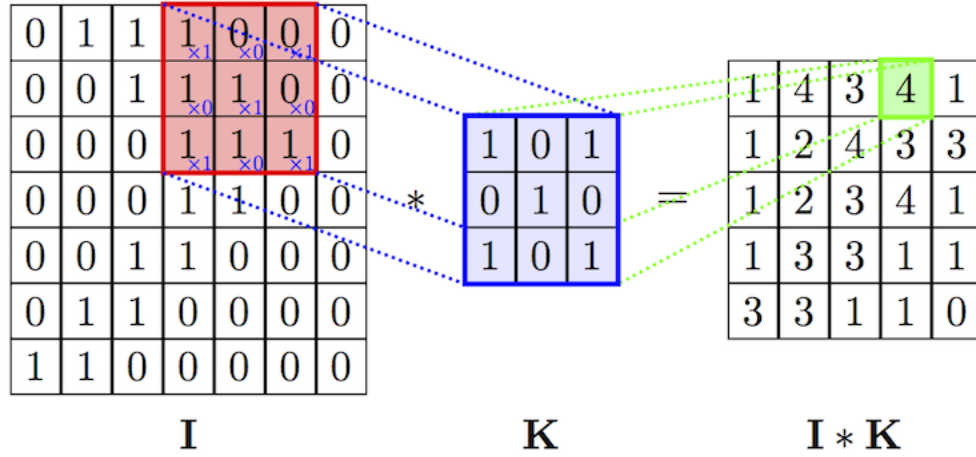


Figure 1.8: 2D-convolution operation.

All the output pixels share the same weights and together, they form a *feature map*. This weight sharing property grants the convolution the ability to extract basic local visual features such as edges, corners, and end-points (Figure 1.9). These local features can then be exploited by subsequent convolutions to detect higher-order local features. The term *local* here is very important because it leads to robustness against the spatial shift of these features in the image.

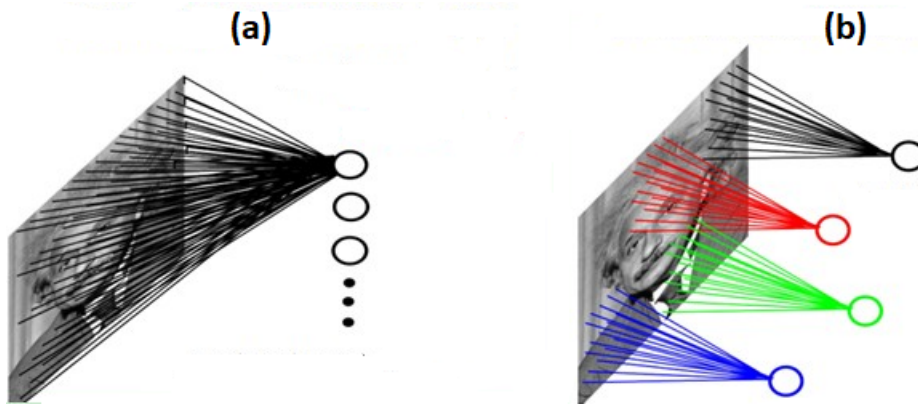


Figure 1.9: (a) : fully connected layer (b): locally connected layer (convolutional) [46].

The kernel in the convolution described above cannot overlap the elements found in the borders of the input image. As a result, the output feature map has a height and width smaller than those of the input. To address this issue, the zero-padding technique is used. With it, rows and columns of zeroes are added on each side of the input image. This way, the border elements can be overlapped by the kernel (Figure 1.10). Thus, the height and width dimensions of the input can be preserved. In modern CNN architectures, zero padding is employed to retain input height and width in order to apply more layers. Without zero-padding, the feature maps would get smaller after each convolutional layer. The distance between two consecutive kernel positions

is known as a stride. The usual choice of stride is 1. However, a larger stride is used at times to perform downsampling (i.e. reduction in the size of the input) of the feature maps. The dimensions (height or width) of a $f \times f \times 1$ feature map can be calculated with the formula:

$$f = \frac{n + 2p - k}{s + 1} \quad (1.16)$$

where:

s is the stride size

p is the padding size

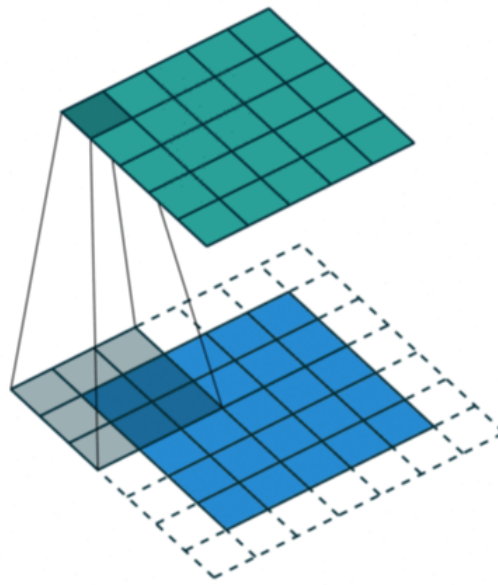


Figure 1.10: Examlpe of zero padding [72].

Convolution in a Convolutional Layer

In general, a convolution layer will transform an input into a stack of feature mappings of that input (output feature maps). The 2D-convolution operation is repeated by convolving multiple kernels on the input feature maps. The results are summed to produce a single output feature map, as shown in Figure 1.11. Kernels stacked against each other form a filter. The components of a filter (kernels) are called *channels*. The filter is trained to extract unique features from the input feature maps of the convolutional layer. Thus, the filter *depth* (number of channels) must match that of the input. For example, a filter with 6 channels will be convolved with 6 input feature maps to produce a single output feature map (Figure 1.11).

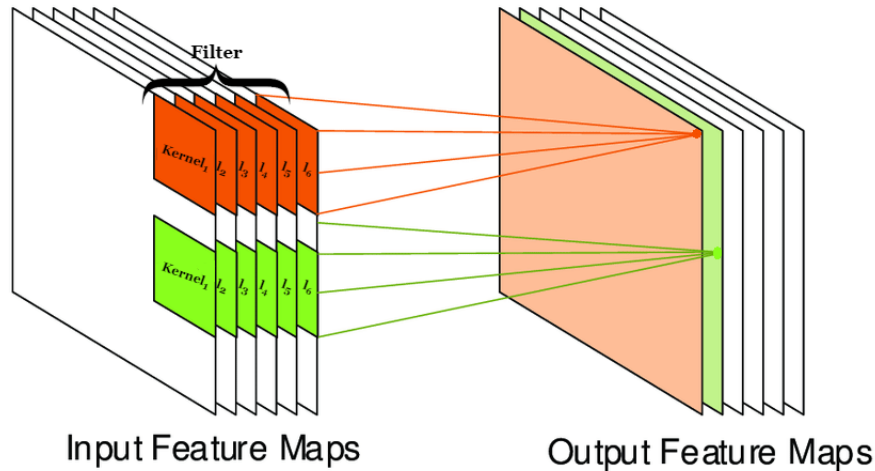


Figure 1.11: Generation of output feature maps by convolving filters with input feature maps.

Convolutional neural networks do not learn a single channel; they, in fact, learn multiple features in parallel for a given input. Essentially, the number of channels in a convolutional layer represents the number of output feature maps.

The convolution operation is defined by two important hyperparameters: The kernel size and the number of channels. Usually, the former is 1×1 , 3×3 , or 5×5 . The latter is defined during the construction of the model, where each convolutional layers contains a predetermined number of channels.

Point-wise Convolution

It refers to a convolution of kernel size 1×1 . At first thought, it may seem futile to construct a convolution layer with only one scalar in each of its filters, but it turns out that this operation is very useful as it allows for:

- Depth control of the input feature maps, as they are passed to the subsequent layer. The depth can be either expanded or shrunk. This control is attained by appropriately choosing the number of channels. As for the width and height dimensions, they can be controlled by adjusting channel sizes and padding parameters, or even by adding a pooling layer (Section 1.12) in some instances.
- Reduction of the computation intensity by decreasing the dimensions. This is especially useful when the model contains convolution layers of channel sizes larger than 1 (3×3 , 5×5 , ... etc). By adding a 1×1 convolutional layer that decreases the depth of feature

maps before a large channel size layer, calculation speed improves greatly. After that, another point-wise convolutional layer can be added to restore the original depth.

1.3.2 Batch Normalization

One problem that emerges during the training phase of the model is that of the internal covariate shift, which happens when there is a change in the input distribution of feature maps. In reaction to this change, the hidden layers try to adapt to this new change throughout the training. The problem is that this shift occurs from one input batch to the next, so in each time, the weights are adjusted according to the new distribution. This can drastically slow down convergence to a minimum of the cost function. To solve this problem, the *batch normalization* technique has been proposed [28], in which the input mini-batches to the layers in the convolutional neural network are standardized. Therefore, stabilizing the learning process and substantially reducing the required number of learning epochs. During training time, the batch normalization layer performs the following:

1. Calculate the batch mean μ_B and batch variance σ_B^2 of the input tensor with m elements:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad (1.17)$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad (1.18)$$

2. Normalize the input tensor's elements using the previously calculated batch statistics:

$$\bar{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (1.19)$$

where ϵ is a small constant used to avoid division by zero (usually 10^{-5})

3. The output tensor elements of the layer are shifted and scaled versions of the input tensor elements:

$$y_i = \gamma \bar{x}_i + \beta \quad (1.20)$$

γ and β are learned during training along with the other parameters of the model.

During inference, the batch mean and batch variance values are fixed. They are an estimation of the previously computed means and variances of the training batches.

1.3.3 Pooling

Another important element of a convolutional neural network is the pooling layer. This layer is obtained through the application of a pooling operator with the goal of aggregating information within each region found in the input feature maps. The results are then down-sampled. In other words, it tries to preserve the more useful information and at the same time, it discards details that are irrelevant. This has the effect of reducing the dependency of the distinctive features to their position in the image. The deployment of this type of layer in CNNs aims at achieving some degree of invariance when the patterns that are sought after change in position or their lighting conditions change.

Feature pooling also increases the robustness to clutter and renders the representation more compact. In the output of the layer, the resolution of the input feature maps is decreased by pooling over local neighborhoods on the feature maps that the previous layer outputs.

In CNNs, two pooling techniques are usually used: max pooling and average pooling. In the max-pooling technique, the largest value in each pooling region is selected, formally:

$$y_{kij} = \max_{(p,q) \in R_{ij}} x_{kpq} \quad (1.21)$$

Where y_{kij} is the pooling operator output associated to the k^{th} feature map, and x_{kpq} is the value at (p, q) in the R_{ij} neighborhood of the position (i, j) .

In the average pooling layer, the output is the average of the elements belonging in the neighborhood region R_{ij} of the position (i, j) :

$$y_{kij} = \frac{1}{|R_{ij}|} \sum_{(p,q) \in R_{ij}} x_{kpq} \quad (1.22)$$

Figure 1.12 depicts an example of a matrix that get down-sampled by pooling its elements using either max pooling or average pooling:

1.3.4 Putting it all together

The CNN model is going to be containing different versions of the layers mentioned in the previous subsections, and the way they are connected to one another differs also from a convolutional neural network model to another. In general, the input image will go through a series of convolutional layers, pooling layers, fully connected layers, and in the end, a softmax function is applied to classify an object by delivering a probability vector.

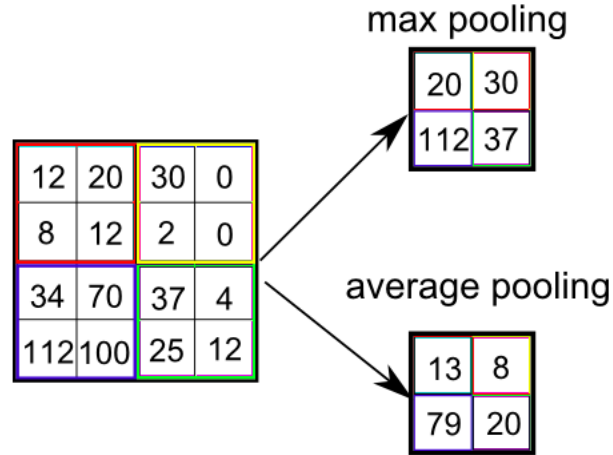


Figure 1.12: Top-right: max pooling, bottom-right: average pooling [67].

Figure 1.13 illustrates a typical CNN architecture designed for images classification. It comprises a feature extraction block composed of convolutional layers and a classification block that contains fully connected layers.

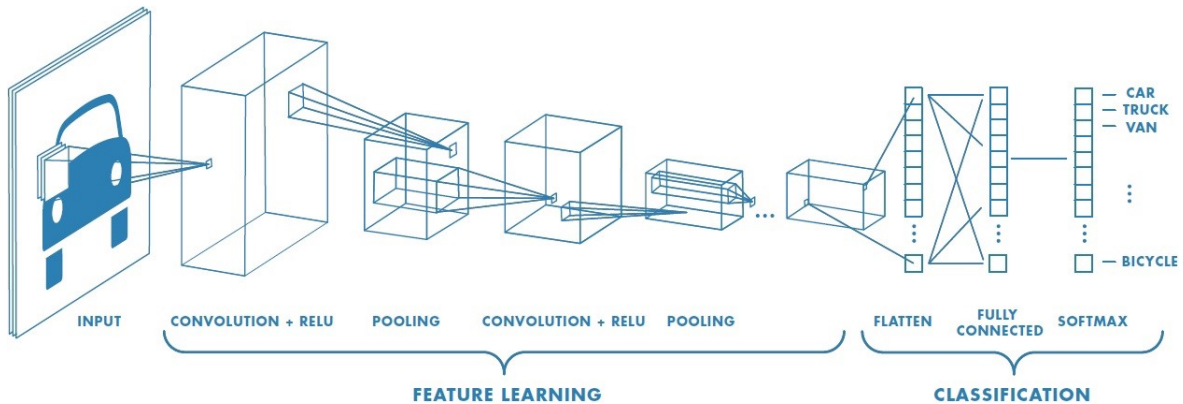


Figure 1.13: Convolutional neural network architecture [8].

1.4 Residual Neural Networks

Deep residual networks are considered to be a major breakthrough. This is because not only do they enable the design of much deeper networks, but also circumvent the problem usually associated with them. Specifically, the vanishing gradient problem [4], [15].

In general, deeper networks are capable of extracting more complex features from the input data [65], [69]. This naturally leads to better performance, yet experiments showed that if the model

gets too deep, the training error increases. Therefore, performance starts to degrade rapidly [22], [66]. This is where residual blocks were proposed as a remedy to this apparent degradation [23].

The alleviation of the deep networks training problem came with the introduction of a new structure called the residual block.

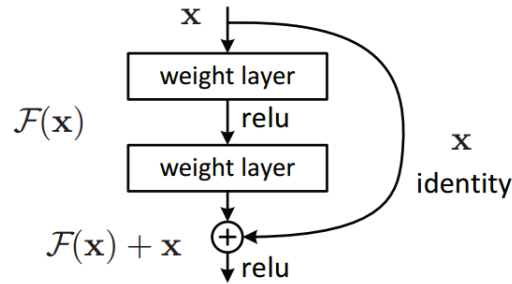


Figure 1.14: Structure of a residual block: a direct path, and an identity mapping [23].

Figure 1.14 above illustrates the most important contribution of the residual block: the addition of a skip connection also referred to as identity mapping. This connection is devoid of parameters and it takes the input x to directly add it to the output of the current convolutional layer $F(x)$. The Skip Connections between layers results in the ability to train much deeper networks than what was previously possible. To eliminate the potential problem of unmatched dimensions between the feature maps x and $F(x)$, a linear projection is applied to the identity mapping in order to expand or shrink the channels of the shortcut, thus, allowing for the possibility to combine x and $F(x)$ as inputs that get fed to the subsequent layer. Formally, this process can be described by:

$$y = F(x, \{W_i\}) + W_s x \quad (1.23)$$

where $F(x, W_i)$ is the direct path that contains numerous convolution layers, whose weights are represented by W_i , and $W_s x$ is the shortcut path, with W_s being the linear projection that matches the dimensions of the input x to that of the output of the direct path.

ResNet model

A ResNet model contains four major layers. Each layer contains a predefined set of bottlenecks, and each bottleneck is composed of a direct path and a shortcut path. In the direct path, three convolutional layers are stacked on top of one another, along with their respective batch normalization and activation layers. The number of bottlenecks dictates the number of total layers in the model. For example, in ResNet 50, the number of bottlenecks is distributed among the four major layers according to Table 1.2

In some bottleneck modules of the network, the output of the 3×3 convolution is expanded

Table 1.2: Distribution of bottlenecks in Resnet50.

Layer	N ^o of Bottlenecks
1	3
2	4
3	5
4	3

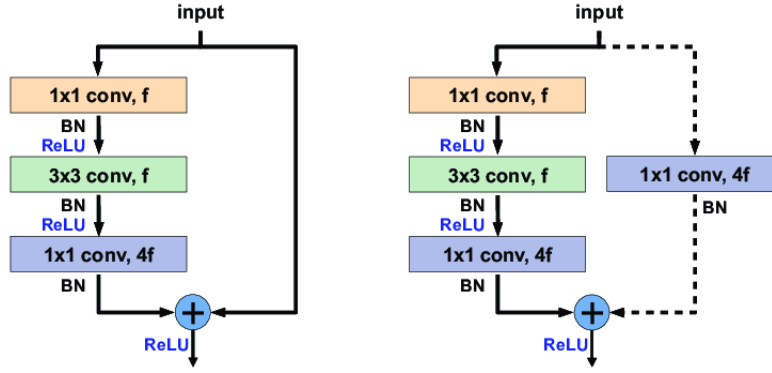


Figure 1.15: (a): bottleneck with no expansion; (b): bottleneck with expansion [8].

(Figure 1.15 (b)). The 1×1 convolutional layer that follows plays the role of a linear projection as it takes the set of output feature maps of dimensions $[C, D, H, W]$ and expands it to $[C * f, C, H, W]$ where f is the expansion factor which, in the case of ResNet50, has the value of 4. This set of four major layers is sandwiched between a 7×7 convolutional module at the input side, and a fully connected module at the output side.

The overall architecture of ResNet50 is illustrated in Figure1.16.

ResNet50 has been the subject of our experiments that involve optimization using pruning and quantization. Subsequently, the performance results of the baseline model and the optimized model are contrasted and discussed in the next chapter.

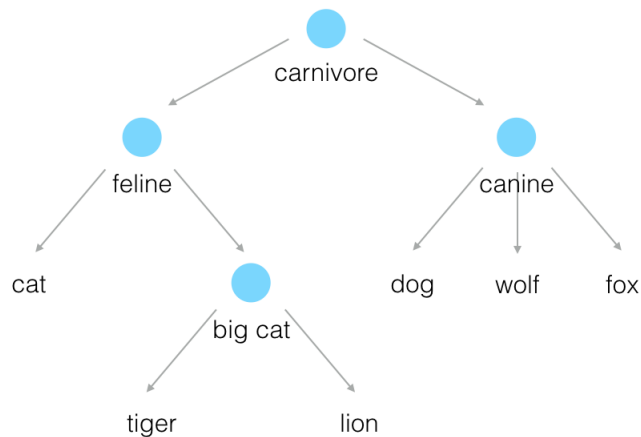


Figure 1.17: Example of a hierarchy of the concept: Carnivore.

Figure 1.17 showcases an example of a hierarchy of synsets that revolve around the *Carnivore* concept. In WordNet, each meaningful concept is potentially described by a multitude of words or word phrases. This concept is referred to as a synonym or a synset. WordNet contains over 100 thousand synsets. A big portion of them are nouns, which amount to approximately 80 thousand synsets. The creators of ImageNet set the goal of providing at least 1000 images for each of these 80 thousand words. So far, researchers have been able to gather over 14 million labeled images across more than 21,000 synsets. Figure 1.18 depicts some examples of the synsets that have the *Canine* synset as a distant parent.

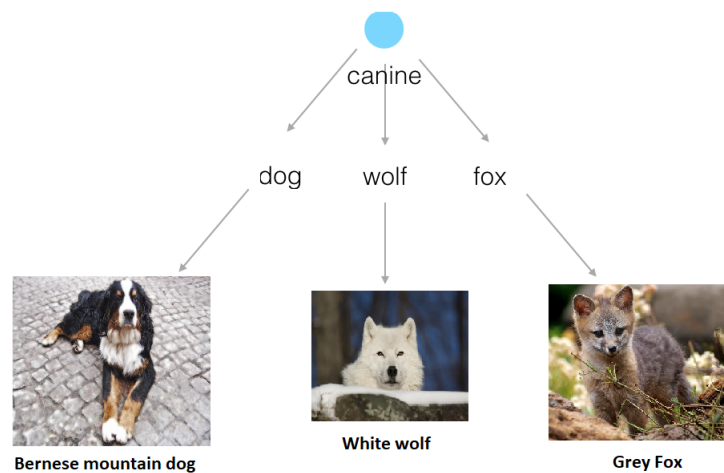


Figure 1.18: Classes that belong to the *Canine* synset.

1.5.2 CIFAR10

CIFAR-10 dataset is a collection of low-resolution images that make it possible for researchers to quickly deploy algorithms, test them, and observe the results. CIFAR-10 comprises 60,000 images with a fixed size of 32x32. These images are distributed among 10 mutually exclusive classes with 6000 images for each class. A subset of 50,000 images is reserved for training and structured in a way such that we have 5 training batches containing randomly-selected images from each class. It is important to note that the training images are not distributed evenly amongst the 10 classes. Some training batches contain more images from one class than another. As for the testing set, it contains exactly 1000 randomly selected images per class. Figure 1.19 shows 10 sample images for each of the 10 classes of the dataset:

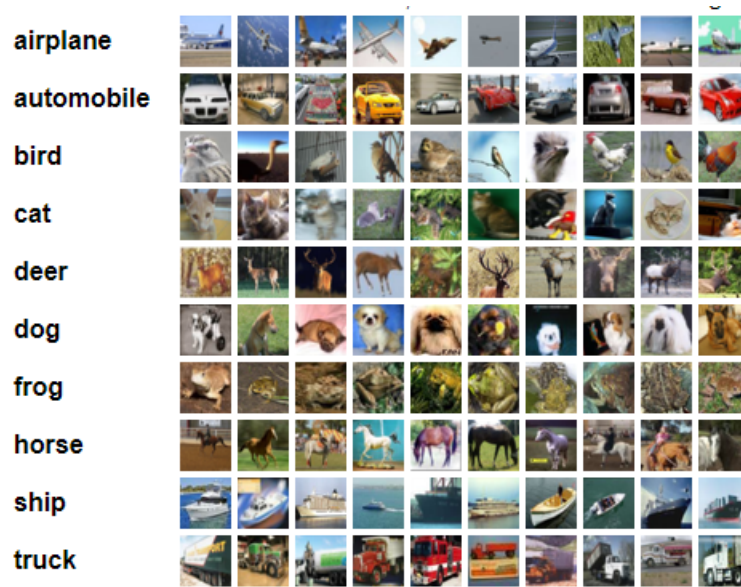


Figure 1.19: Sample images of the CIFAR-10 dataset [1].

1.6 Pytorch Deep learning framework

When it comes to implementing machine learning algorithms, one must make use of a *Deep learning framework* to make development easier. A deep learning framework is an interface, library, or a tool which allows us to build deep learning models more easily and quickly, without getting into the details of underlying algorithms. It provides a clear and concise way of defining models using a collection of pre-built and optimized components. Pytorch¹ is an open-source framework. It has been developed primarily by Facebook AI Research lab (FAIR) in 2016. It is built to be flexible and modular for research, with the stability and support needed for production deployment.

Pytorch applies a graph-meta programming based approach — the executive code for defining layers, composing models, loading data, and running optimizer are all expressed by general-purpose programming. Such a design ensures that any new neural network architecture can be easily generalized and implemented with PyTorch. We have chosen Pytorch over other frameworks such as Tensorflow² because, as the latter relies on the non-intuitive computational graphs, PyTorch shares many similarities with imperative programming languages that us, electronics engineers, are more familiar with. Additionally, its *pythonic* style made it more intuitive to use and easier to learn.

1.7 Conclusion

This chapter presented general information regarding deep neural networks, starting with the most basic building block, the neuron, and going through the different constituting layers, their characteristics, and usage in the network, we also talked about the training phase since some of the methods presented later on involve adjusting the weights in a process known as fine-tuning.

One particular layer of interest to us is the convolutional layer. It is notorious for being computationally expensive and hard to parallelize. This is largely due to its implementation being serial in nature [48], optimizing this layer will result in an impressive improvement in inference time using a residual-block-based model, i.e. Resnet50. We tried to keep the material presented in this chapter as relevant as possible so as to not diverge from the focus of our dissertation, which we shall explore in detail in the following chapters.

Pytorch¹: Official Pytorch website: <https://pytorch.org/>

Tensorflow²: Official Tensorflow website: <https://www.tensorflow.org/>

Chapter 2

Optimization Methods

2.1 Introduction

In this chapter, we first and foremost explore the literature concerning the advancements made in the general field of code optimization. The methods involved apply universally to any software source code, including deep neural network implementations. Afterward, we treat in great lengths the two principal components of our work: **Network Pruning** and **Quantization**. We discuss the principles governing them, lay out the state-of-the-art progress, and implement them on our framework of choice.

The convolutional model on which these methods are applied is *ResNet50*. We have chosen this model for two main reasons. Firstly, it offers excellent classification accuracy on datasets like ImageNet and CIFAR10. Secondly, many researchers use *ResNet50* as a backbone upon which their own model proposals are built, case in point, the *RetinaNet* model designed for object detection, that we will explore in Chapter 3. As for the dataset, we have opted to conduct the experiments on CIFAR10 instead of ImageNet since it is a relatively small collection of low-resolution images, something which renders it very convenient for us. It makes it possible to quickly test and fine-tune the model that goes through the optimization pipeline.

2.2 Software Optimizations for Embedded Systems

Contrary to traditional computers like PCs, laptops, or servers, embedded systems operate under arduous restrictions when it comes to resource availability. Some micro-controllers have word-lengths of only 8 or 16 bits and have to read from or write data to a small-size memory. In addition, the entire system runs on a clock speed in the tens of Megahertz. However, if the system is not constrained enough, sometimes it must fulfill additional requirements.

The example of real-time systems comes to mind, where they have to finish computations, and deliver the results before the next batch of data arrives.

These systems have to also be energy efficient, and follow stringent constraints with respect to size and weight. Which is why optimization plays an important role in the process of designing an embedded system. The software that runs on this type of system needs to fulfill all requirements with the scarce resources that are at its disposal. There are some optimizations that need to be considered by the programmer, while others can be directly handled by the compiler. This section covers most of the two types.

2.2.1 Prior to Optimization in the Compiler

Determining functional accuracy prior to beginning the optimization process is paramount. A baseline function needs to be established by generating tests in order to enable easy identification and localization of potential errors that occurred when optimizing the program. These errors can be induced through incorrect changes in the code done by the programmer, or through an extremely aggressive optimization by a compiler. Figure 2.1 showcases the basic workflow of software optimization.

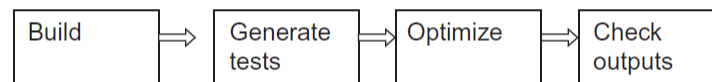


Figure 2.1: The optimisation process.

The Profiler

The profiler is a tool that performs dynamic analysis on the program source code or its executable binary form. Not only can it measure memory and time complexity, but also monitor the usage of particular instructions of interest, or the duration and frequency of function calls.

Software profilers rely on a number of different techniques which include event-based, statistical, instrumented, and simulation methods. If the programmer wants to optimise the code for speed for example, he/she can use the profiler to analyse where cycles are spent, to determine critical regions in the code, and to proceed with the optimisation process.

Below is an example of C code to be optimised:

```
1 int x=0;
2 int y = 2;
3 z = 2*y;
4 for ( int i = 0; i < 50; i ++ ) {
5     x = x + z;
6 }
7 printf("value of x is %d",x);
```

One can determine with simple visual scrutiny that the program above spends more than 90% of its execution time in the **for** loop (lines 4-5). Thus, optimization in this region can significantly improve the running speed of this code. In our case, we use the profiler in the embedded system application to monitor the CPU, and memory usage. Further details are given in the later sections.

2.2.2 Optimizing Software for Performance

These techniques focus more on improving execution speed with no regard to code size in the memory.

Data-types and Intrinsics

Leaning the sizes of different data types on the core prior to starting to write code is very important. Opting to store data in one type over another can have drastic performance implications, and a lot of it has to do with the compiler. It is required to support all the necessary *data-types* so it will try to find a workaround if a data-type incompatible with the hardware is fed. For example, a CPU may not support 32-bit multiplication. Writing code that involves this operation will cause the compiler to generate a sequence of instructions to construct 32-bit multiplication using 16-bit operations that the CPU supports.

Intrinsics are functions native to the target hardware. They represent a way to express some operations that are not possible or practical to express in a low-level programming language like C. *Intrinsics* can be combined with custom data types that allow for the use of non-standard data sizes or types. They can be used to access application-specific instructions that cannot be generated automatically from ANSI C¹ by the compiler. At first glance, they appear to be function calls, but what actually happens is that the compiler will replace them with the intended sequence of instructions that perform the custom operation. Thus, calling overhead is virtually non-existent. Examples of hardware-specific features accessible through *intrinsics* include:

¹ANSI C¹: a collection of successive standards for the C programming language published by the American National Standards Institute (ANSI)

- Arithmetic saturation.
- Fractional types.
- Enabling/disabling interrupts.

Pointers and Memory Access

Some embedded specialized processors like DSPs are capable of loading multiple data values through buses to ensure that all the arithmetic units are kept busy. These operations are called multiple data moves. During the process, adjacent values are moved in from memory to the processor registers. Usually, a compiler aligns variables in memory according to their access width. To leverage multiple data moves, the data has to be placed in a higher alignment. In other words, to load two 16-bits values at once, the data must be aligned to 32-bits in a memory that supports 32-bits word lengths. In our work, after *quantizing* a neural network from 32-bits floating-point precision to an 8-bits integer value, the QNNPACK library [11] is capable of ensuring an alignment of four 8-bit values in a single 32-bit GPR in the ARM processor.

One important thing that the programmer needs to keep in mind is *pointer aliasing*. If pointers are to be used in the same piece of code, great care must be given to avoiding pointers that direct to the same memory location. If this is guaranteed never to happen, the compiler can ensure accesses to memory locations pointed to by those pointers in parallel.

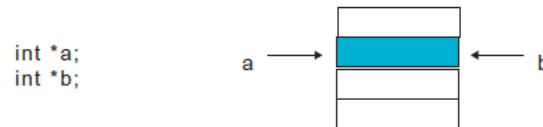


Figure 2.2: Aliasing: Two pointers that store the same memory address.

If pointer aliasing cannot be guaranteed not to happen, the compiler will assume that pointers could alias, and parallel access is avoided. Otherwise, a fatal run-time error could occur.

General Loop Transformations

The optimization techniques that are presented in this section are critical in taking full advantage of processors with modern multi-ALU architectures. A modern compiler has the ability to perform most of these optimizations, if possible, even simultaneously.

Loop Unrolling

The *unrolling* is a type of code transformation in which the body of the loop is duplicated multiple times. Each copy of the loop can be slightly tweaked, and different registers can be used in each case. If the loop performs N iterations originally and gets unrolled K times, instruction level

parallelism (ILP) can be achieved, the execution speed is increased by a factor of K [26]. For example, the following loop,

```
1 for(int i=0;i<100;i++)
2   a[i]=2;
```

is unrolled by a factor of 4.

```
1 for(i=0;i<100;i+=4){
2   a[i]=2;
3   a[i+1]=2;
4   a[i+2]=2;
5   a[i+3]=2;
6 }
```

By using this code transformation, the execution is speeded-up by a factor of 4.

Multisampling Partial Summation

This iteration method maximizes the usage of ALUs units during execution. The output values that share common input source data values are calculated in parallel. Multisampling extends to any signal-processing calculation that has the form:

$$y[n] = \sum_{m=0}^M x[n+m]h[n] \quad (2.1)$$

If the outputs $y[n]$ are independent of each other, and at the same time, their inputs overlap, then they can be calculated simultaneously. Thus, calculation time is considerably reduced.

In partial summation, one output sum is divided into numerous smaller sums, which are added at the end. Like multisampling, partial summation enables us to leverage parallelism inside the processing unit. However, unlike multisampling, partial summation is prone to errors since serial dependency between the partial sums is broken.

These unrolling and partial summation methods forms the basis of convolution optimization in deep neural networks, since convolution can be implemented using nested loops, which are optimized using a GEMM-based algorithm [48]

2.2.3 Optimizing Software for Memory

Software optimization for memory designates the process of reducing code size in order to make it fit into a smaller memory location than it did originally. In this section, we will cover the two

techniques for compacting code: The pattern-based procedural abstraction approach, and the multiple-memory access approach [31]. But before we delve into the details of these methods, some key concepts need to be cleared in order to better understand the optimization process.

Control Flow and Data Flow Graphs

The control flow graph (CFG) is a representation of a program that involves specifying the paths that might be traversed in the code during execution. In a control graph structure, each node denotes a sequence of instructions without any jumps or jump targets. The block of instructions that fulfill this requirement is called a basic block. A basic block starts after a jump target and ends after a jump. The edges in the CFG represent the flow of control between blocks.

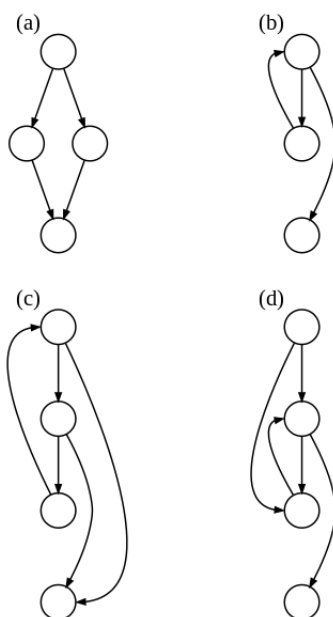


Figure 2.3: Examples of control graphs; nodes represent basic blocks, edges represent flow of control [76]

The DFG (data flow graph) follows the same representational pattern as the CFG. The difference is in the edges. In the case of DFG, they represent the flow of data between blocs.

Program Dependence Graph

The program dependence graph fuses the control flow and data flow graphs into one single representation. The nodes still represent sequential instructions. But the edges comprise two types, one for representing the passing of control from one block to another, and one for representing relevant data flow relationships between blocks. This approach combines the most important aspects of a program and therefore, has some advantages. The main leverage is that there is only one data structure to hold and maintain in the compiler. Therefore, many optimizations can be

carried out within a single walk of the PDG and optimizations that required complex analysis of CFG and DFG are achieved more easily (vectorization [3]).

Procedural Abstraction

This method involves reducing code size by replacing similar code occurrences with compiler-generated functions. While this can be achieved by directly analyzing the program source code, a graph representation is much more pertinent because of the normalizing effect graph representations like the PDG, which increases the probability of finding similar patterns in the program structure. The average code size reduction using this approach in non-object-oriented programming languages is up to 30% [35]. Algorithm 1 describes the procedure.

Algorithm 1: Procedural Abstraction Algorithm.

input : Program P
output: A semantically equivalent P' program with reduced size
Method:
START: Generate all patterns in P ;
Find a pattern π with greatest benefit ;
if π is found **then**
 generate abstract f_{π}^A function ;
 insert calls to f_{π}^A at every occurrence of π ;
 remove now-redundant nodes ;
 go back to **START**
else
 end ;

Multiple Memory Access

Multiple memory access refers to the process of loading and storing data using two or more registers simultaneously. Numerous modern microprocessors use MMA instructions for the purpose of reducing code size. These instructions include multiple loads and stores like the LDM and STM instructions found in the ARMV7 series of microprocessors. These allow for a significantly more compact code by making it possible to express a large set of registers with few instruction word bits. For example, the ARMV7 processor uses only 16 bits to encode up to 16 register loads (512 bits if the LDM is not used) [35] The MMA algorithm looks for potential candidate load/store instructions that can be replaced by MMA instructions.

2.2.4 Optimization with Pytorch mobile

In recent years, many platforms and frameworks have been introduced by the major technology companies intended for machine learning applications on mobile systems. Facebook's solution goes under the name of Pytorch Mobile¹. It is a portable version of Pytorch capable of supporting fully-fledged end-to-end workflows, going from project development using Python to deployment on mobile operating systems like IOS or Android. It provides APIs for preprocessing, as well as the integration routines required for a successful machine learning incorporation in mobile applications. It also comes with built-in support for QNNPACK quantized libraries, as well as support for ARM-based CPUs.

QNNPACK library

QNNPACK (short for Quantized Neural Networks PACKage) is a library specifically optimized for mobile systems. Its main design purpose is the acceleration of neural network inference at a minimal cost of precision loss. This library presents an algorithm called the "indirect convolution algorithm" [10] which is a modified version of the GEMM-based algorithms [6]. Additionally, it offers support for quantized 8-bit tensors and 8-bit tensor operations which improve inference time even further. In the next chapter, we will deal with quantization in the QNNPACK library, but for now, our focus is on how the convolution operation is accelerated.

Most computer vision architectures are built upon convolutional layers with large filter sizes. These layers are notorious for their computational intensity [48]. If they were to be implemented directly, it would mean the involvement of many multiply-add operations that are associated with every loaded element, something which would make inference on edge devices impractical.

Recently, the trend is heading towards using several other types of convolution that do not leverage fast convolution algorithms, for example, 1×1 convolutions. Others also include grouped, strided, dilated, and depth-wise convolutions. In addition to that, they have a low arithmetic intensity which reduces the required memory bandwidth for their utilization through the exploitation of their low-precision computation capabilities.

Matrix-Matrix multiplication

In Figure 2.4, matrix A ($M \times K$) is multiplied by matrix B ($K \times N$) to produce matrix C ($M \times N$). One of the optimizations that QNNPACK provides is found in matrix multiplications. Instead of calculating each element in the output matrix C one at a time, several rows and columns of A and B respectively are considered, meaning, MR elements of A and NR elements of B are loaded and $MR \times NR$ multiply-accumulate operations are performed. This would avoid the case where

Pytorch Mobile¹: Official website: <https://pytorch.org/mobile/home/>

the processing unit becomes bottlenecked by memory or cache bandwidth, and the performance, in this case, is solely dependent on the computational power of multiply-add units.

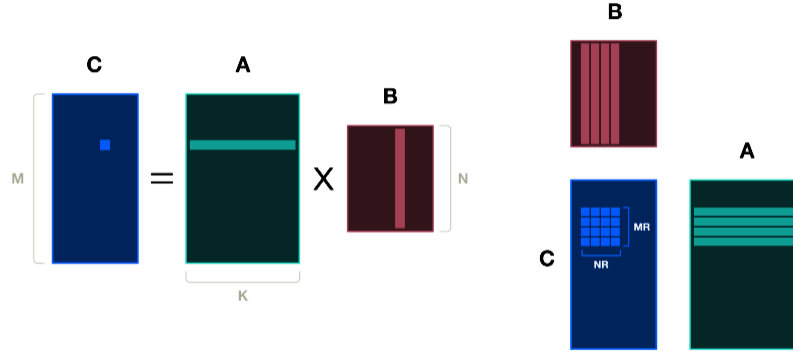


Figure 2.4: left: traditional matrix multiplication; right: QNNPACK matrix multiplication. [11]

Convolution with QNNPACK

For a convolution with 1×1 kernel size, direct matrix-matrix multiplication is performed. But this is not the case for convolutions with larger strides, padding, or kernel sizes. These more complex convolutions can be performed by mapping them to matrix-matrix multiplications through the use of a transformation called *im2col*, which aligns the data in such a way that the convolution operation can be achieved using matrix-matrix multiplication.

This technique is much more efficient because, in traditional element-wise convolution, a lot of irregular memory accesses need to be called. *Im2col* arranges the data so that memory accesses are regular, which improves speed drastically. But this comes at the price of a significant overhead because for each pixel value in the output, *im2col* copies segments of the input image required to compute it. And given that each pixel is dependent on $k_h \times k_w \times C$ input pixel, (k_h and k_w are the kernel's height and width respectively, C being the number of channels in the input image), the resulting buffer matrix is $k_h \times k_w$ times bigger than the input image.

In QNNPACK, a more efficient algorithm is implemented. The multiply-add micro-kernel is adapted to perform *im2col* transformation on the fly. Instead of copying actual data values from input images to the *im2col* buffer, a buffer of pointers is created where each pointer stores the address of the input data on which the output pixel depends. The pointer buffer, also called indirection buffer, eliminates the overhead of *im2col* transformation. Figure 2.5 depicts a visual comparison between the GEMM-based (*im2col*-based) approach, and the QNNPACK approach.

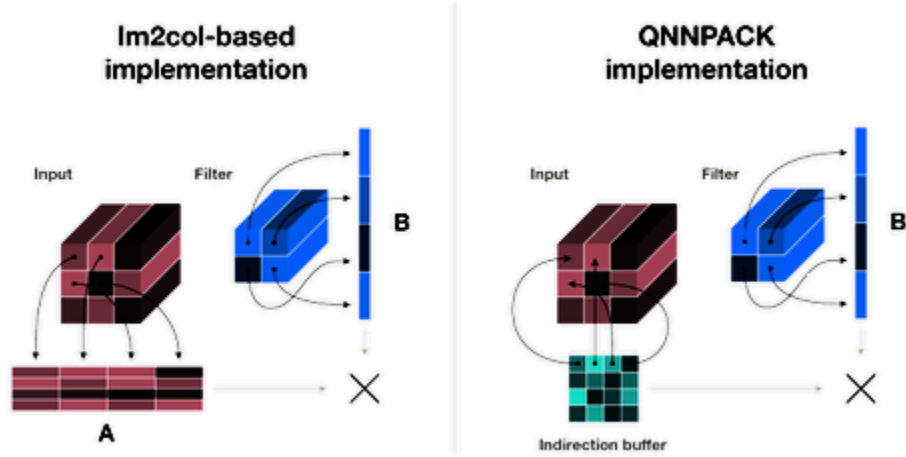


Figure 2.5: Addition of the indirection buffer [11].

Depthwise convolutions with QNNPACK

QNNPACK provides a separate implementation for depth-wise convolutions which has a vastly different computational pattern in comparison to normal convolution. Depth-wise convolution is a special case of the broader concept called grouped convolution, where the input and output channels are split into several groups. Each group is processed independently. In the edge case, when the number of groups equals the number of channels, the convolution is called depth-wise. This latter type of convolution is used commonly in state-of-the-art neural network architectures. It carries out spacial filtering of each channel independently in a way that does not resemble normal convolution.

QNNPACK includes a highly optimized version of 3×3 depthwise convolution (Figure 2.6). The QNNPACK implementation achieves a high performance, and one of the reasons is that it perfectly utilizes general-purpose registers (GPRs) in order to unroll the looping over filter elements, and at the same time, avoid reloads for address registers in the hot loop.

In the 32-bit ARM architecture, 14 GPRs are implemented. In 3×3 depth-wise convolution, the most frequently used in mobile-based neural networks, 9 input rows and 9 filter rows are required. Thus, a total of 18 addresses must be stored to fully unroll the loop. Since the filters do not change at inference time, their previous shape of $C \times k_h \times k_w$ is packed into the new shape $\frac{C}{8} \times k_h \times k_w \times 8$. Where C is the number of channels, k_h and k_w represent the channel height and width respectively. With this repacking, only one GPR is used, and all filters are accessed with address increments. The factor 8 is used because 8 elements can be loaded with one instruction, then the zero point is subtracted, producing eight values of 16 bits in length aligned in a 128-bit

NEON register. Regarding the input row pointers, 9 of them can be fit along with the repacked filters in only 10 GPRs, thus, the loop can be fully unrolled over filter elements. In the 64-bit ARM architecture, the number of GPRs is doubled. The extra GPRs are leveraged by QNNPACK to hold pointers to 3×5 input rows and calculate 3 output rows at a time.

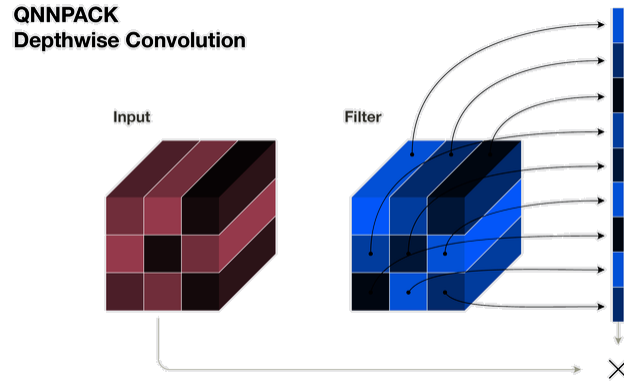


Figure 2.6: Depthwise 3×3 convolution in QNNPACK [11].

2.3 Neural Network Pruning

Despite being so accurate at so many kinds of tasks, DNNs have the drawback of being over-parameterized, which has been shown in many works [20], [52]. Therefore, a large amount of weights in a network are redundant and can be removed (i.e., set to zero). This process is called network pruning.

It was first proposed in 1989 through a technique called optimal brain damage [37]. The idea was to compute the impact of each weight on the training loss, referred to as the weight saliency. The low-saliency weights were removed and the remaining weights were fine-tuned; this process was repeated until the desired weight reduction and accuracy were reached.

With the resurgence of deep learning in 2012 and Neural Networks getting deeper and bigger size-wise, pruning has increasingly attracted attention recently, due to its ability to reduce an overwhelming amount of parameters. Many papers have been published proposing various methods [24], [25], [47].

Weights pruning, or model pruning, is a set of methods designed to increase the sparsity (amount of zero-valued elements in a tensor) of a network's weights. In general, the term 'parameters' refers to both weight and bias tensors of a model. Biases are rarely, if ever, pruned because there are very few bias elements compared to weight elements. Subsequently, they are just not worth removing since they have little impact on performance.

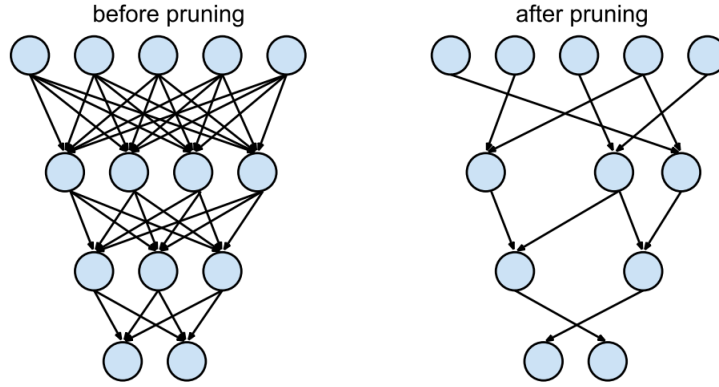


Figure 2.7: Weights and Neurons Pruning in a fully-connected network.

DNNs consist of two types of layers: fully-connected layers and convolutional layers. They perform matrix-vector and matrix-matrix multiplication, respectively. Weight pruning techniques [20], [21], [37] measure the importance of each weight individually and remove those deemed unimportant, as shown in Figure 2.7. This results in both memory storage and computation reductions. After weight pruning, the weight matrices in the pruned networks become sparse. Thus, the remaining weights are stored in a sparse matrix format. Figure 2.8 shows the transformation of a sparse into the Compressed Sparse Row format. This new presentation requires less memory as we store 3 vectors only, instead of the entire matrix. Eventually, Storing a sparse neural network in this format leads to a significant reduction in overall memory usage.

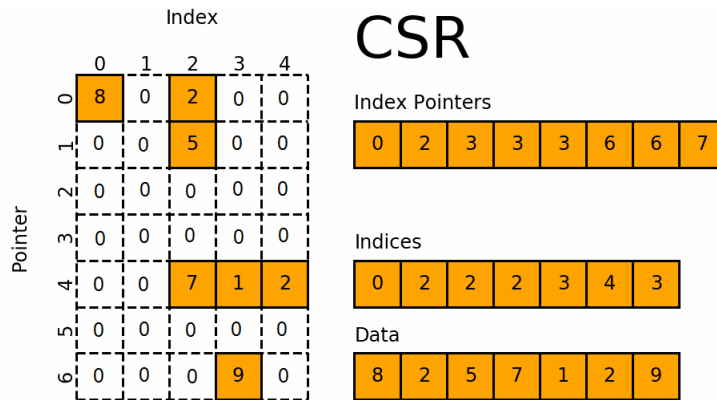


Figure 2.8: Transforming to compressed sparse format.

Though it is able to achieve a high theoretical speed-up ratio, the sparsity of pruned networks often results in a performance decrease in DNN computation. Sparse weight matrices lose the regular structure of dense matrices, thus causing irregular memory accesses, i.e. a complex storage structure that requires extra computation to decode the sparse format, and therefore,

losing optimal usage of the supported hardware, e.g. memory coalescing. Moreover, dense matrix optimizations, like matrix tiling, would be less effective.

Therefore, to run pruned models effectively, it requires either special hardware accelerators or a special implementation, making use of libraries like cuSPARSE for NVIDIA GPUs and MKL SPBLAS for intel CPUs. These libraries are optimized for sparse matrix-vector and matrix-matrix multiplications. This Makes the implementation hardware-dependent and more complicated. This is the reason why we opt for filter pruning where we remove certain filters (weight tensors) entirely. It can be seen clearly in Figure 2.9, that filter-wise pruning is much more regular, making it easier for hardware acceleration. Although this method is more hardware friendly, it causes

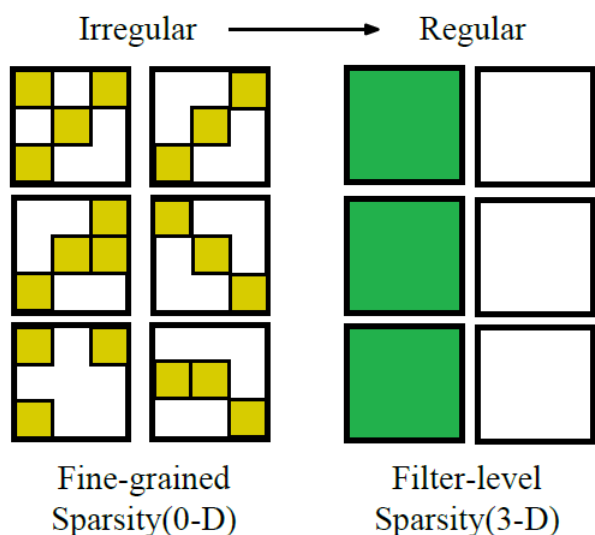


Figure 2.9: Two different structures of sparsity.

larger accuracy loss than pruning individual weights [49]. As can be seen in Figure 2.10. The green curve, which represents filter pruning, decreases rapidly as pruning rates increases. Other less regular structured forms of sparsity do not hurt accuracy as much. Fortunately, there are some techniques to mitigate the severeness of this downside.

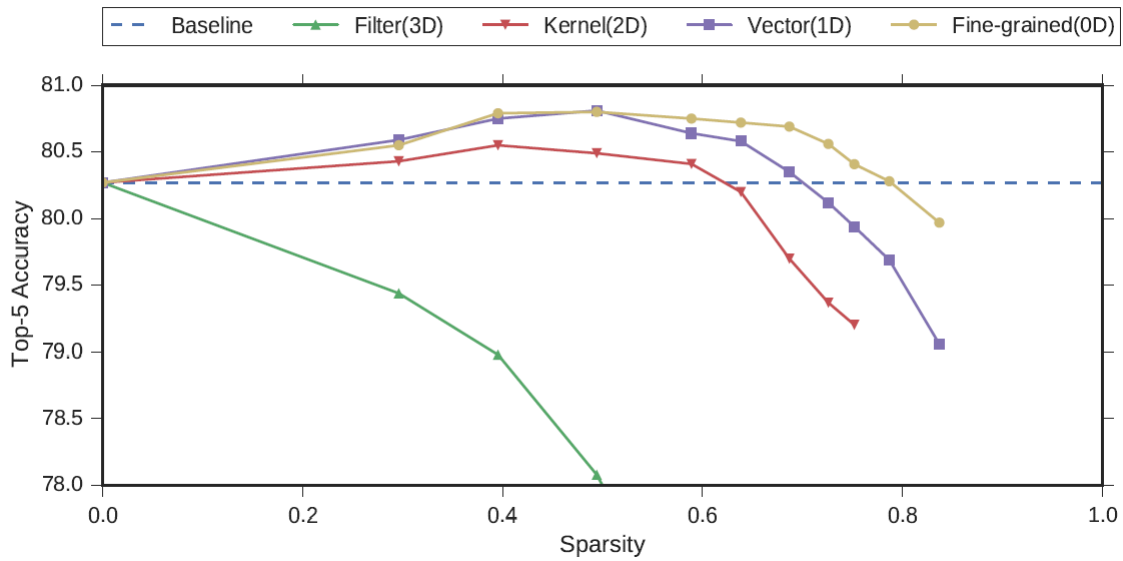


Figure 2.10: Accuracy-Sparsity Curve of AlexNet with different grain sizes. X-axis: sparsity of conv layers (percentage of zero weights). Y-axis: top-5 accuracy on ImageNet validation set [49].

2.3.1 Filter pruning

Filter pruning involves removing the entire filter in the weight tensor. It is a form of structured pruning that compresses a model in such a way that does not require special hardware or special implementations to reap its benefits performance-wise. Removing hardware/implementation dependency made this type of pruning quite popular in the research community. In DNN models that have serial data dependencies, defining and pruning channels is reasonably straight-forward. That being said, models with a more complex design, that includes parallel data dependencies (paths) - such as GoogLeNet (Inception layers) and ResNets (skip connections) – things become increasingly more challenging and require an elaborate understanding of how the data flows in the model, in order to define the pruning schedule.

In filter pruning we use some criterion to determine which filters are important and which are not. We will expand on that in the next section. But for now, we won't take notice of how the filters to prune are determined. Suppose that in Figure 2.11, we choose to prune the selected grey filters (The * symbol denotes the convolution operation).

The fact that the convolution layer is stripped of two filters means that there will be two less output feature maps, meaning the structure of the data is also altered. Therefore, when we prune filters, we not only need to reconfigure the immediate convolution layer, but we also need to take into account that the input FMs of the next layer is smaller. Hence, we need to reduce the next layer's filters, by removing the channels that correspond to the pruned filters. We say that there is

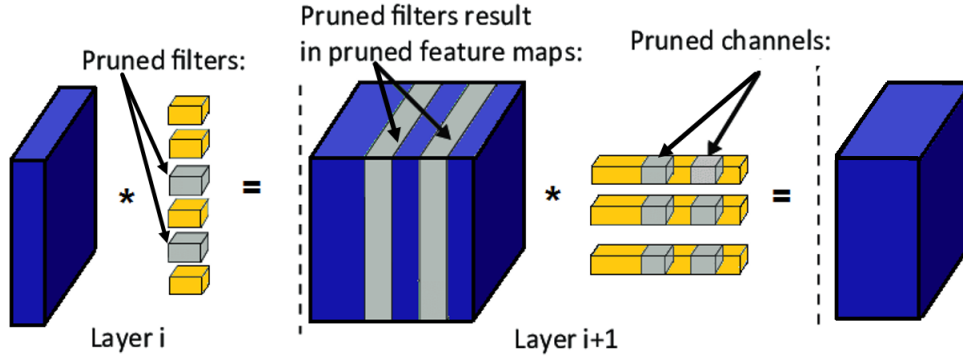


Figure 2.11: Illustration of data-dependency in filter pruning (side view).

a data dependency between the two adjacent Convolution layers (see Figure 2.11). The activation functions that immediately follow these convolution layers are devoid of trainable parameters and are, therefore, not sensitive to the shape of their input.

The Batch Normalization (BN) layer is parameterized by a couple of tensors that contain information per input feature map (i.e. scale and shift). Since the output FMs of the preceding convolution layer is reduced in dimensions, and being the input to the Batch Normalization layer, we also need to reconfigure the Batch Normalization layer. In addition, we need to physically shrink the Batch Normalization layer's scale and shift tensors, which are coefficients in the BN input transformation. Moreover, the scale and shift coefficients that we remove from the tensors, must correspond to the filters (or output feature-maps channels) that have already been removed from the Convolution layer.

In another common case, we might have two parallel convolutional layers whose outputs undergo an element-wise summation that is fed into a third Convolution, as is the case with residual blocks (see 1.4). Figure 2.12 shows how *conv3* is dependent on both *conv1* and *conv2*, and there are two implications to this dependency.

The more obvious implication is that the number of filters pruned in *conv1* and *conv2* must be equal. Since we apply element-wise addition on the outputs of *conv1* and *conv2*, they must have the same shape - and they can only have the same shape if *conv1* and *conv2* prune the same number of filters. The second implication of this triangular data-dependency is that we ought to prune the exact same filters in *conv1* and *conv2* (as shown in Figure 2.12 because otherwise, we would not know what channels to prune from the following convolutional layer. This means that we need to decide whether to use the pruning on *conv1* and *conv2* according to the filter-pruning choices of *conv1* or of *conv2*.

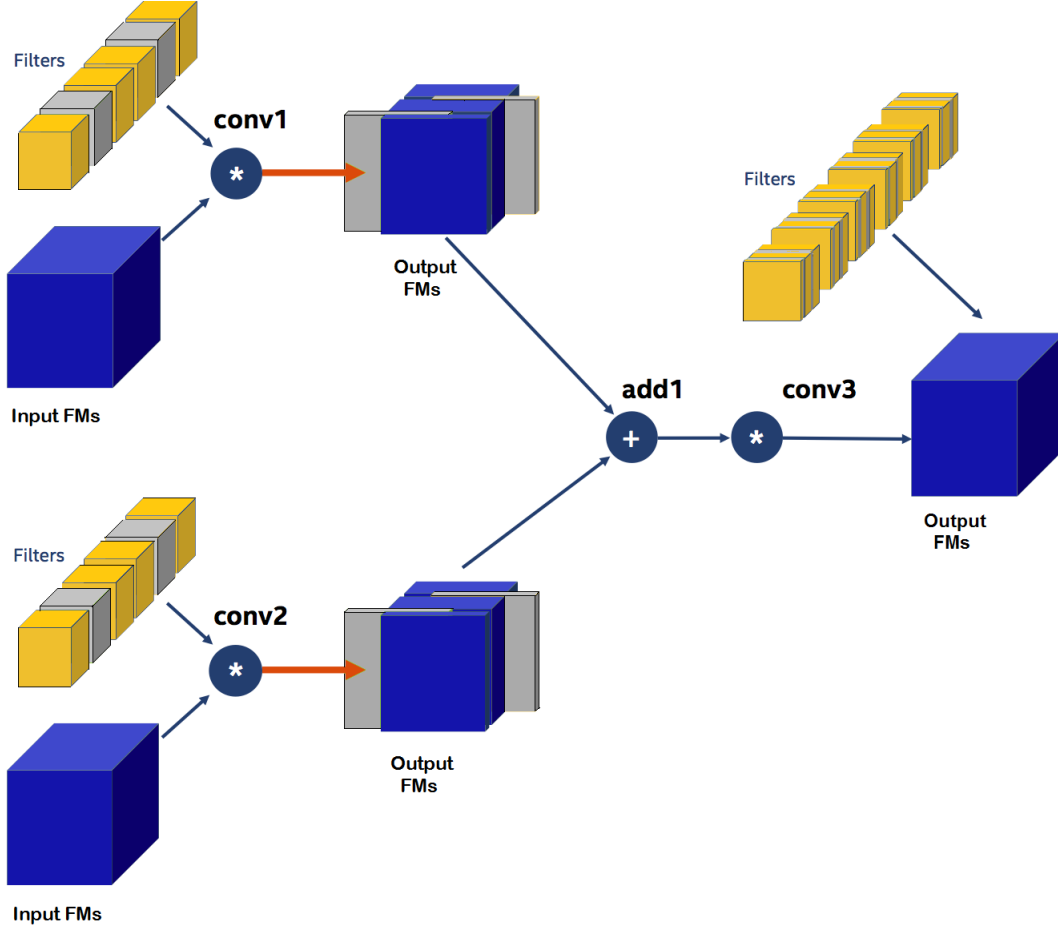


Figure 2.12: Parallel Conv layers Pruning (front view).

2.3.2 Pruning Scheme

The challenge in filter pruning is how to decide what filters are the most redundant in the network, and thus should be removed. Researchers have been extensively working on this area and proposed various pruning schemes that are more and more effective. Basically, pruning schemes can be categorized mainly into two groups: Criteria-based and Regularization-based methods.

The first group prunes weights based on some established importance criteria. For instance, Optimal Brain Damage [37] and Optimal Brain Surgery [21] propose an importance criteria based on the second-order derivatives of the loss function derived from Taylor expansion. Han et al [20], rather than using the saliency as a metric, which is too difficult to compute for the large-scaled DNNs, they simply prune small magnitude weights. The obtained compression is impressive but may not always translate directly to faster inference since modern hardware exploits regularities in the computation for high throughput as explained in paragraph 2.3. Taylor Pruning [52], also introduces a new importance criterion based on Taylor expansion, but they use the

first-order derivatives considering their easy access during back-propagation. Their method is shown to be effective in pruning filters on transfer learning tasks. Li et al. [40] use L1-norm to guide one-shot filter pruning (as we will see in the next subsection), which is simple and proves effective on CIFAR-10 and ImageNet with VGG-16 and ResNet, but the reported speed-up is limited. He et al. [25] alternatively used LASSO-regression-based channel selection and feature map reconstruction to prune filters, and achieves one of the state-of-the-arts on VGG-16. Xu et al. [77] sensitivity aware pruning consists of performing a fast sensitivity test to help identify the sensitivity of each layer in a network for a given accuracy tolerance. AMC proposed by He et al. [24] augment Channel Pruning with an optimized pruning ratio combination for different convolutional layers based on searching via reinforcement learning.

The second set of methods is regularization-based methods, which add group regularization terms to the objective function to penalize unimportant parameters and prune redundant filters directly by minimizing the objective function during training. Among these methods we find Group-wise Brain Damage [36] and Structured Sparsity Learning [75] that embed Group LASSO [78] into CNN regularization and obtain regular-shape sparsity. Both works use a constant regularization factor to all the filters in the network, which is problematic because this scheme is a "one size fits all", it implicitly assumes that all weights in different groups are equally important, which does not hold true. A recent method by Wang et al. [73] proposes an incremental regularization scheme that enables the network to adapt during pruning. Lately, structured sparsity regularization (SSR) [41] was introduced that achieves a state of the art performance. It forms a new optimization problem based on two structured sparsity with $l_{2,1}$ -regularization and $l_{2,0}$ -regularization, both of which can be efficiently solved by a novel Alternative Updating with Lagrange Multipliers (AULM).

In our work we favored a criterion-based over a regularization-based filter pruning since the latter is slow in convergence for filter selection, thus resource-intensive, which in our case was not possible. Hence, we have opted for the L2-norm selection criterion in the first stage due to its simplicity. It is a reasonable choice as other approaches are comparable to it performance-wise, despite that, their implementation is much more complex.

L1/L2-norm

We measure the relative importance of a filter in each layer by calculating the sum of its absolute weights $\sum \|F_{i,j}\|$, i.e., its L1-norm $\|F_{i,j}\|_1$. This value gives an expectation of the magnitude of the output feature map. Filters with smaller kernel weights tend to produce feature maps with weak activations as compared to the other filters in that layer. Thus they contribute little to the final result (low saliency) and can be removed. To further improve this method, we can choose a specific threshold for each layer. The threshold is calculated from the pruning percentage that we

want to achieve, which in turn is intertwined with the desired compression and speed-up rates. In practice, we increase the pruning percentage until we reach the targeted performance.

$$threshold(F_i) = \begin{cases} F_i & \text{if } \|F_i\|_1 > \lambda \\ 0 & \text{if } \|F_i\|_1 \leq \lambda \end{cases} \quad (2.2)$$

The same can be said with L2-norm. The difference is that for L1-norm, large weight values and small weight values have the same impact on the group's importance. However, in real DNNs, larger weights tend to be more important and should have a larger impact. L2-norm considers all the weight values, and larger weight value will have a larger impact on the group's importance. Therefore, compared to the L1-norm, it is expected to help remove more redundant filters.

Iterative Pruning

When employing a pruning-followed-by-retraining regimen, we can achieve much better results (higher sparsity at practically no accuracy loss). This is called *iterative pruning*, and the retraining that follows pruning is often referred to as *fine-tuning*. How the pruning criteria changes between iterations; how many iterations we perform and how often; and which filters are pruned - this is collectively called the *pruning schedule*. Iterative pruning could be thought of as repeatedly learning which weights are important, removing the least important ones based on some importance criterion, and then retraining the model to let the accuracy "recover" from the pruning by adjusting the remaining weights. At each iteration, more weights are pruned.

The decision of when to stop pruning is also expressed in the schedule, and it depends on the pruning algorithm. For example, if we are trying to achieve a specific sparsity level, then we stop when the pruning achieves that level. If we are pruning weight structures in order to reduce the required compute budget, then we stop the pruning when this compute reduction is achieved. In our case, we chose the former.

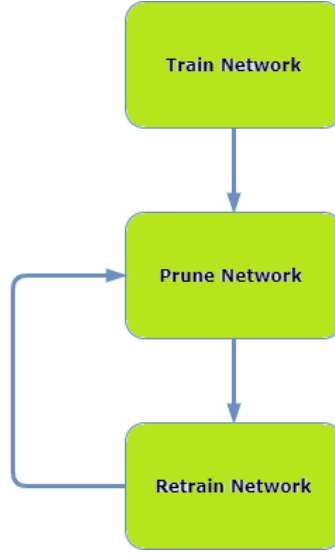


Figure 2.13: Iterative Pruning.

2.3.3 Experiments and Results

We have implemented Network pruning in Pytorch. We have conducted the experiments on intel Xeon(R) CPU. We have pruned a modern classification model, ResNet 50 trained on CIFAR10 image dataset.

As explained in Subsection 1.4, ResNet 50 consists of 10 residual blocks, each residual block has 3 bottlenecks comprising 3 convolutional layers each, *conv1*, *conv2* and *conv3* of kernel size being 1×1 , 3×3 and 1×1 respectively. As shown in Figure 2.14, we pruned at first stage only on *conv2* of all bottlenecks, where the convolution is most expensive computation-wise, (*conv3* will be channel pruned systematically as explained earlier). In the second trial we pruned both *conv1* and *conv2*, similarly *conv3* is pruned as well.

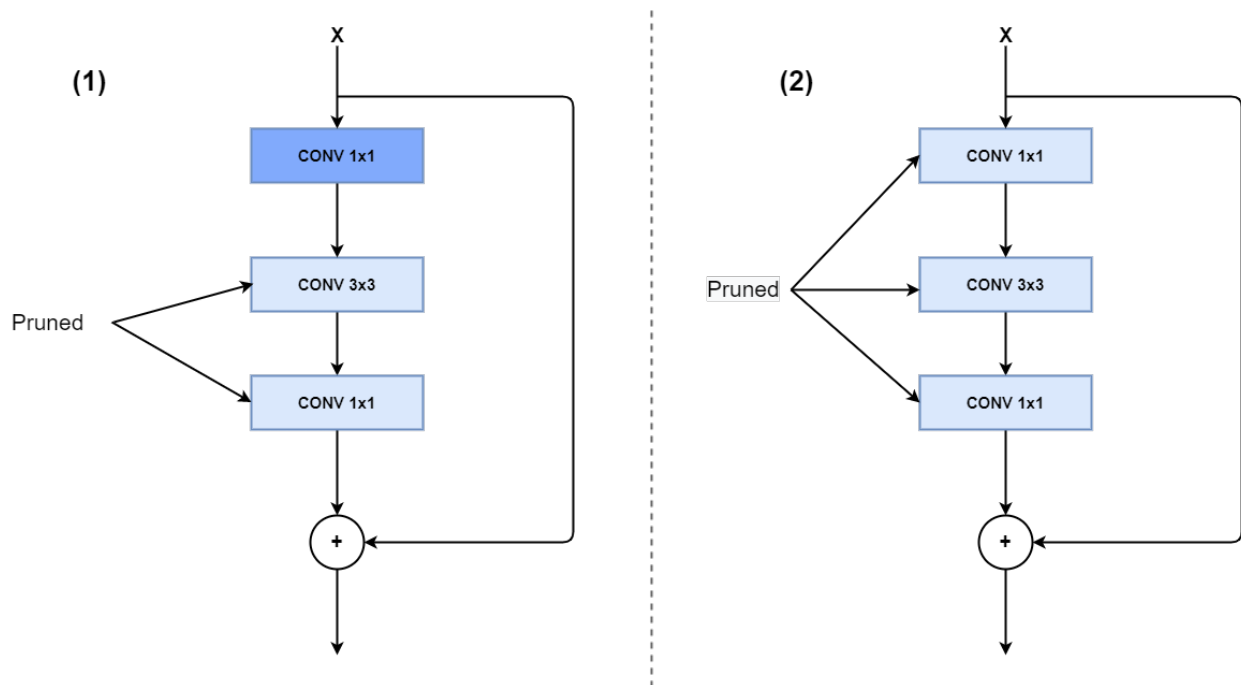


Figure 2.14: Illustration of 1st and 2nd trials, (Batch Normalization and ReLu are not presented for simplicity).

We have first applied a *one-shot pruning*, which is a strait forward approach: Take a pretrained model and prune it at once. The compression and speed-up are impressive, however, we have noticed a significant drop in accuracy after 40% in sparsity as can be seen in Figure 2.15.

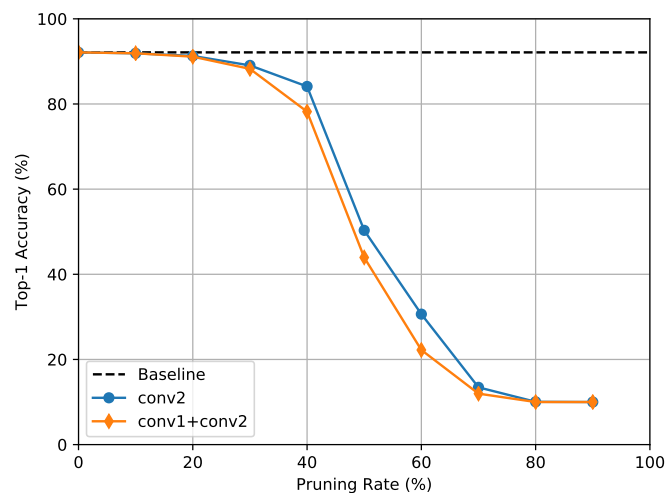


Figure 2.15: Top-1 Accuracy on CIFAR10 Test set against Pruning rate.

Figure 2.16 presents the average inference time of a batch size 200 image for each pruning rate. We have observed that the inference time decreases as more redundant filters are removed. Clearly, pruning both *conv1* and *conv2* gives better results since more weights are discarded. We have also noticed that at the 40% mark - where accuracy loss is relatively acceptable - there is only a small decrease in execution time in both methods, beyond 40% pruning rate, the inference time becomes irrelevant as the accuracy is far too low to be taken into account.

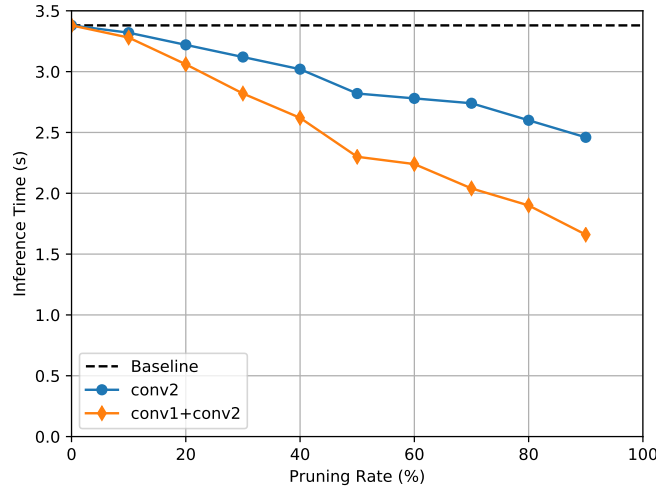


Figure 2.16: Inference time for a batch size of 200 against Pruning rate.

In Figure 3.6, we see that the size of the model gradually decreases as the pruning rate increases. At each pruning rate, the size reduction in the cases of pruning both *conv1* and *conv2* is almost double that of the method where we prune *conv2* only.

We have noticed that pruning both 1st and 2nd convolutional layers of each bottleneck has shown to cause nearly the same accuracy loss as pruning 2nd layer alone. On the other hand, it results in much more compression and speed-up as shown in Figures 3.6 and 2.16. Therefore, it is more reasonable to proceed with the first approach.

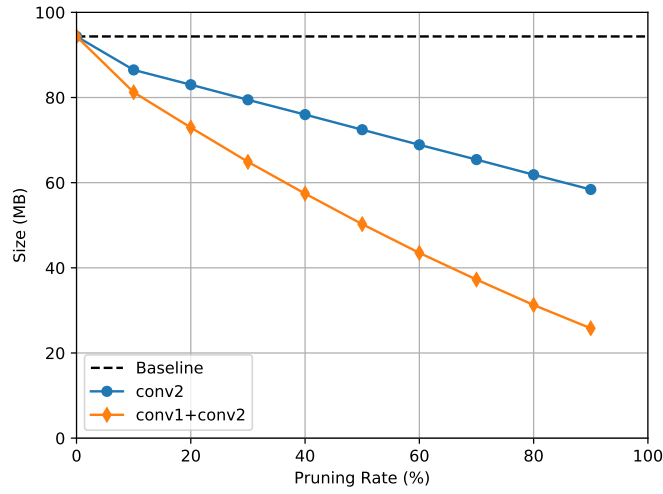


Figure 2.17: Model size against Pruning rate.

In the next phase, we performed *iterative pruning*. At each iteration, we increased the sparsity level by 10% and then we retrained the network until it recovers (each time 10% of the new model). Eventually, High pruning rates required more epochs to regain accuracy. At first, 4 epochs were enough. However, as pruning became aggressive we added more epochs and reduced the learning rate each time the loss plateaus. This is a clear case of model over-parametrization with respect to the dataset.

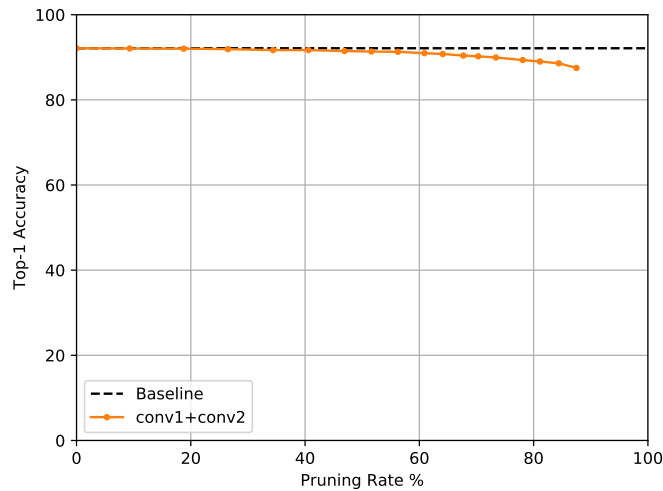


Figure 2.18: Top-1 Accuracy on CIFAR10 Testset against Pruning rate.

The results obtained of iterative pruning, exposed in Figure 2.18, show that there is little to no significant accuracy loss. Indeed, the baseline starts at 92.12% and manages to keep the accuracy at 89% even though more than 80% of the total weights have been removed.

2.4 Neural Network Quantizaion

Quantization refers to the process of reducing the number of bits that represent a number. In the context of deep learning, the predominant numerical format used for research and deployment has so far been a 32-bit floating-point or FP32. However, the desire for reduced bandwidth and compute requirements of deep learning models has driven research into using lower-precision numerical formats. It has been extensively demonstrated that weights and activations can be represented using 8-bit integers (or INT8) without incurring a significant loss in accuracy [2], [30], [80]. Broadly speaking, there are, two reasons for this.

First, DNN parameters are known to be quite robust to noise and other small perturbations once trained. This means that even if we subtly round off numbers, we can still expect reasonably accurate results. In addition, the weights and activations by a particular layer often tend to lie in a small range, which can be estimated beforehand hence allowing us to concentrate our precious fewer bits within a smaller range, for example, -1 to +1.

Below in Figure 2.19 is an example of a histogram representing the weight distribution of GoogleNet [69]. We can notice that most values are concentrated in a small range.

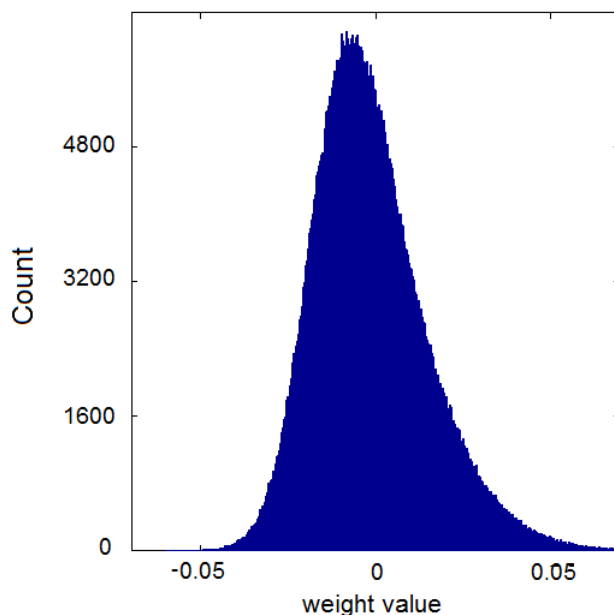


Figure 2.19: Weight distribution for GoogleNet [19].

There are two main attributes when discussing a numerical format. The first is dynamic range, which refers to the range of representable numbers. The second one is how many values can be represented within the dynamic range, which in turn determines the **precision / resolution** of the format (the distance between two numbers). For all integer formats, the dynamic range is $[-2^{n-1}; 2^{n-1} - 1]$, where n is the number of bits. So for INT8 the range is $[-128; 127]$. The number of representable values is $2n$. In contrast, with FP32, the dynamic range is $\pm 3.4 \times 10^{38}$, and approximately 4.2×10^9 values can be represented.

There are two key benefits to representing the data in integers using int8:

- Using integer arithmetics instead of floating-point math leads to faster computation as most processors allow for faster processing of 8-bit data, especially embedded systems (subject of our work).
- Reduce data storage requirements by a factor of 4, since a single-precision floating-point requires 32 bits to represent a number (default data type in many frameworks including PyTorch). The result is a reduction in the memory used to store all the weights and biases, and in the power consumed in transferring all the data, since energy consumption is dominated by memory access.

The two factors above translate into faster inference, with a typical speed-up of 2-4x due to the reduced precision for both memory accesses and computations. Further improvements in speed and power consumption are possible with processors and hardware accelerators optimized for low precision vector arithmetic. That is why machine learning library developers are building extensive framework support for quantized inference.

However, it is until 2018 when quantization has seen the light, after developers from google proposed a new approach for quantizing neural network [30], that is different from previous approaches. As other quantization approaches do not deliver verifiable efficiency improvements on real hardware. Approaches that quantize only the weights, for instance, [7], [16], [61] are primarily concerned with on-device storage and less with computational efficiency. We point out that there are some exceptions notably binary, ternary, and bit-shift networks [27], [39], [55] that employ weights that are either 0 or powers of 2, which allow multiplication to be implemented by bit shifts. However, while bit-shifts can be efficient in custom hardware, they provide little benefit on existing hardware with multiply-add instructions that, when properly used (i.e. pipelined), are not more expensive than additions alone. Moreover, multiplications are only expensive if the operands are wide, and the need to avoid multiplications diminishes with bit depth once both weights and activations are quantized.

B.Jacob’s and *al.* [30] approach become popular to a large extent due to its efficiency. Google published afterward a white paper describing it in more detail. Recently, it has been implemented in many frameworks like TensorFlow, Pytorch, and Caffe.

2.4.1 Quantization scheme

In this paragraph, we describe the general quantization scheme, that is, the correspondence between the bit-representation of values (denoted q below, for “quantized value”) and their interpretation as mathematical real numbers (denoted r below, for “real value”).

At first, the quantization scheme [30] has a couple of requirements:

- An affine mapping of integers q to real numbers r : This means that it permits efficient implementation of all arithmetic using only integer arithmetic operations on the quantized values (no use of lookup tables because these tend to perform poorly compared to pure arithmetic).
- Always represent 0.f accurately: Zero-point is an integer, ensuring that zero is quantized with no error is critical so that common operations like zero padding do not cause quantization error. Moreover, having 0 map to another value that’s higher/lower than zero will introduce a bias in the quantization scheme.

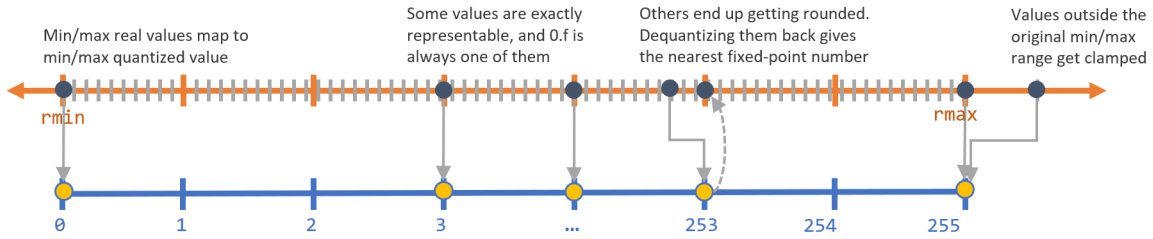


Figure 2.20: Converting real values to integer values [62].

Overall, a quantization scheme will simply be shifting and scaling of the real number line to a quantized number line. For a given set of real values, the minimum/maximum real values are in this range $[x_{min}, x_{max}]$ to map to the minimum/maximum integer values $[-2^{B-1}; 2^{B-1} - 1]$ respectively, with everything in between linearly distributed.

Thus, the quantization goes as follows:

$$x_{int} = \text{round}\left(\frac{x}{\Delta}\right) + z \quad (2.3)$$

$$x_Q = \text{clamp}(-N_{levels}/2, N_{levels}/2 - 1, x_{int}) \quad (2.4)$$

The de-quantization operation is:

$$x_{float} = (x_Q - z)\Delta \quad (2.5)$$

- x is the real value (float32).
- x_Q is its quantized representation as a B-bit integer (int8, int16, etc.)
- x_{float} is the rounded value of x (float32).
- Δ (float32) and z (int8) are the factors by which we scale and shift the number line. z is the quantized 'zero-point' which will always map back exactly to 0.f.

$$\Delta = \frac{x_{max} - x_{min}}{2^B - 1} \quad (2.6)$$

The FP32 scale factor is approximated using fixed-point multiplication followed by a shift operation (detailed explanation can be found in [30, sec 2.2]). In most cases the effect of this approximation on accuracy is negligible.

The quantization scheme uses a single set of quantization parameters (Δ and z) for all values within each activation array and within each weight array; separate arrays use separate quantization parameters.

While the uniform affine quantizer allows for storing weights and activations at 8-bits of precision, there is an additional cost. Consider a 2D convolution between a weight and an activation:

$$y(k, l, n) = conv(w(k, l, m, n), x(k, l, m)) \quad (2.7)$$

$$y(k, l, n) = \Delta_w \Delta_x conv(w_Q(k, l, m, n) - z_w, x_Q(k, l, m) - z_x) \quad (2.8)$$

$$y(k, l, n) = conv(w_Q(k, l, m, n), x_Q(k, l, m)) - z_w \sum_{k=0}^{K-1} \sum_{l=0}^{K-1} \sum_{m=0}^{N-1} x_Q(k, l, m) - z_x \sum_{k=0}^{K-1} \sum_{l=0}^{K-1} \sum_{m=0}^{N-1} w_Q(k, l, m, n) + z_x z_w \quad (2.9)$$

Other than the first term is extra computation due to the zero-point.

To accelerate the process, the equation above is optimized by noting that the last two terms are constant and can be computed off-line. This is further improved by noting that the sum over activations is identical for all convolutional kernels of the same size.

We show below an example presenting the graph transformation for a convolutional layer is shown in 2.23.

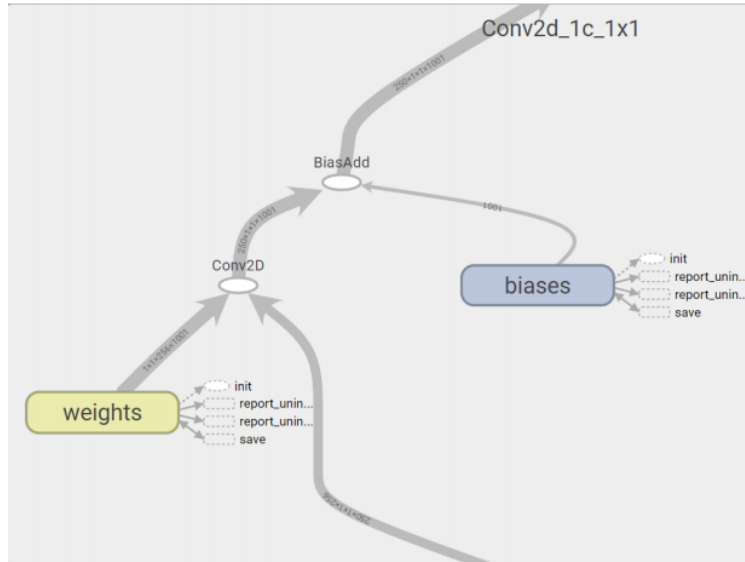


Figure 2.21: Floating point.

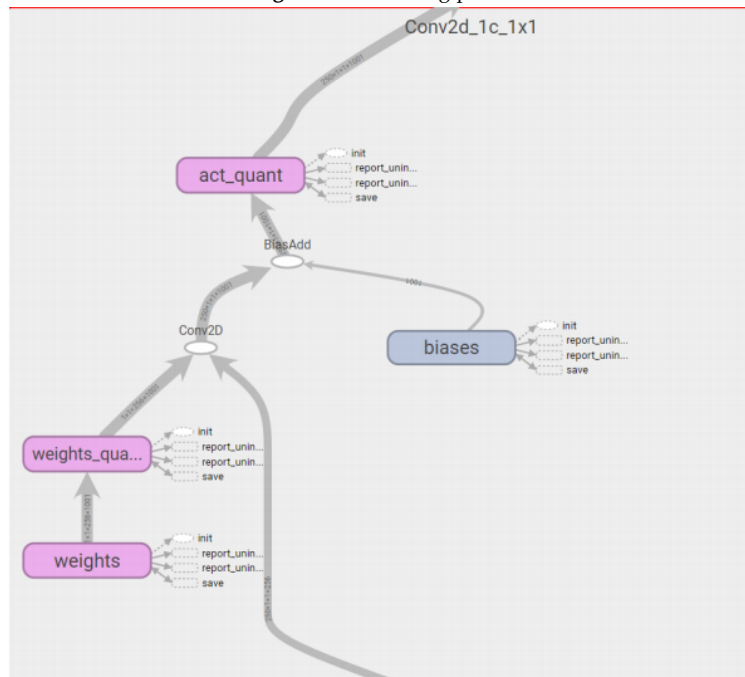


Figure 2.22: Fixed point.

Figure 2.23: Convolutional layer: Before and After Graph Transformation [33].

Avoiding Overflows

Convolution and fully connected layers involve the storing of intermediate results in accumulators, and if we would use the same bit-width for the activations and weights, and for the accumulators, it is almost certain that we would overflow due to the limited dynamic range of

integer formats. Therefore, accumulators are usually implemented with higher bit-widths.

The result of multiplying two n -bit integers is, at most, a $2n$ -bit number. In convolutional layers, these multiplications are accumulated $c.k^2$ times, where c is the number of input channels in a filter, and k is the kernel size (single channel), assuming a square kernel. Thereby, to avoid overflowing, the accumulator should be $2n + M$ -bits wide, where M is at least $\log_2(c.k^2)$. In our cases, accumulators of 32-bit width are used. Figure 2.24 shows data widths before and after a matrix multiplication in computation graph.

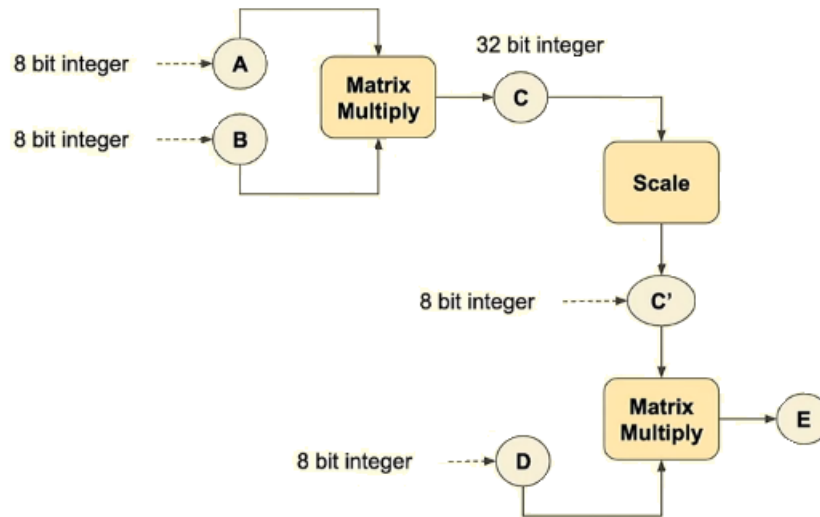


Figure 2.24: Illustration of a quantized computation graph.

2.4.2 Quantization modes in Pytorch

Pytorch developers have recently implemented three techniques for quantizing neural networks as part of quantization tooling in the torch. quantization name-space:

Dynamic quantization

The easiest method of quantization PyTorch supports is known as dynamic quantization. In this quantization, variant weights are converted ahead of time whereas the activations are dynamically quantized during inference (on the fly), just before the computation (hence “dynamic”). However, the activations are stored and loaded from memory in a 32-floating point format.

It is typically used in a situation where execution time is dominated by loading weights from memory rather than matrix computation. The typical use cases for this are LSTM and Transformer type models with small batch sizes.

Post-Training Static Quantization

Static quantization involves converting activations in addition to weights. This results in more speed-up as the execution of the networks uses both integer arithmetic and int8 memory accesses. The scale factor and zero-point for the activation tensors are pre-computed based on observing the behavior of the model during a *calibration process*, where batches of data are fed into the network to compute the resulting distributions of activations. This information is used to determine how the different activations tensors should be quantized at inference time.

Quantization Aware Training

Quantization-aware training (QAT) is the one that typically results in the highest accuracy of these three. With QAT, all weights and activations are “fake quantized” during both the forward and backward passes of training; meaning that float values are clamped and rounded to simulate the effects of INT8 quantization, but all computations are still done in float-32 format. Thus, all the weight adjustments during training are made while “aware” of the fact that the model will ultimately be quantized; after quantizing, therefore, this method usually yields higher accuracy than the other two methods.

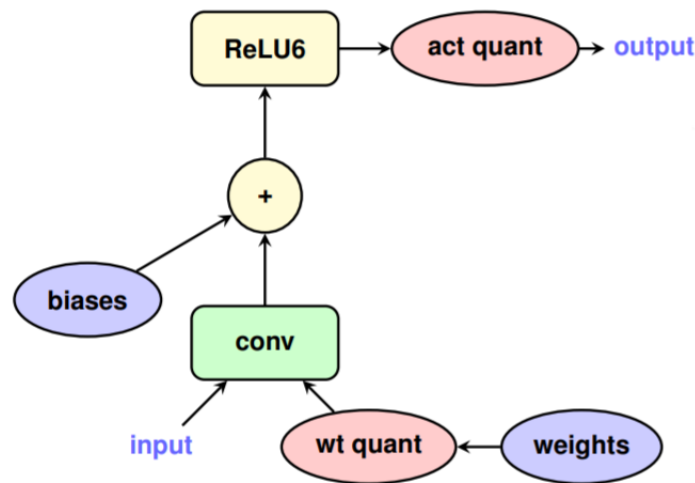


Figure 2.25: Training with simulated quantization [33].

Neural Network Quantization with QNNPACK

As described in Section 2.2.4, QNNPACK is a backend kernel dedicated to optimized mobile deep learning. It uses a linear quantization scheme, which is compatible with Android Neural Networks API. QNNPACK’s optimization focus is on 8-bit quantized tensors that mobile

AI use is designed for. Other optimization libraries that specifically target performance improvements in scientific computing, expecting matrices to be very large, containing thousands of double-precision floating-point values, in the case of QNNPACK, the input matrices contain low-precision elements and come at a variety of dimensions. In usual mobile applications (e.g: computer vision task), the number of in-channels K and out-channels N do not exceed 1024. They typically lie within the 32-256 range.

Mobile architecture constraints require that the loaded elements that take part in the simultaneous multiply-add operations, i.e. MR and NR , do not exceed 8 (Section 2.2.4). In the extreme case, the biggest models include 1024 channels in some layers. Therefore, the whole memory block read in the QNNPACK microkernel is at most 16KB. As a result, it can fit without a problem in the L1 cache on mobile devices, even those with low-end specifications. This property enables QNNPACK to process the whole panels of A and B in a single call of the microkernel, thus, eliminating the need to accumulate 32-bit intermediate results outside of the microkernel. Furthermore, this library is able to fuse re-quantization of the 32-bit intermediate results into the microkernel and output the quantized 8-bit final result, this saves both memory bandwidth and cache footprint.

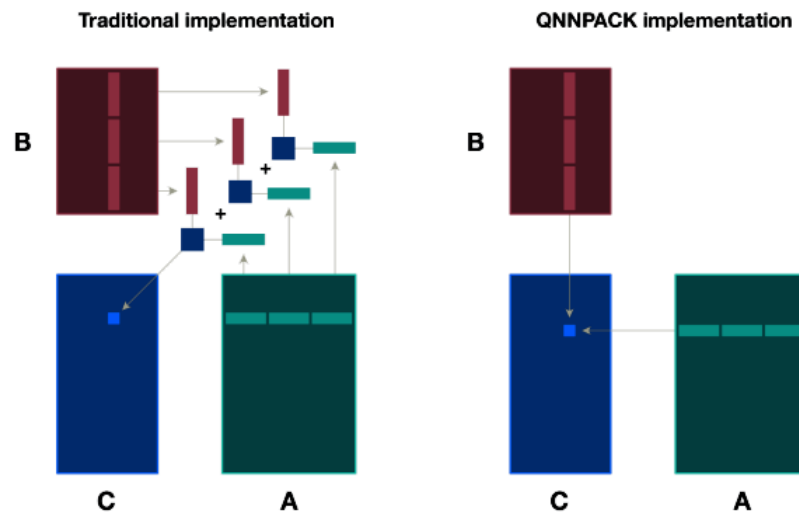


Figure 2.26: Optimized matrix on matrix multiplication.

Squeezing the whole panels of A and B into cache memory allows for another optimization in QNNPACK: Matrix A repacking is removed. Repacking is the process of taking an object in memory, a matrix for example, and transform its layout to another shape or form in order to process it. In the case of matrix B, the elements that it contains are static weights and can be transformed into any memory layout at the cost of a one-time operation. But for matrix A, which contains feature maps, its elements are changing from one inference to the next. Ergo, repacking matrix A induces overhead within each inference run.

In traditional implementations like GEMM, matrix A repacking cannot be avoided on account of limited cache associativity and microkernel efficiency. Without repacking, the microkernel is forced to read rows of matrix A that could be separated by large strides. If these strides are multiples of large powers of 2, elements from different rows of matrix A could potentially fall into the same cache set in the panel. Hence, the performance decreases dramatically if the number of colliding rows exceeds cache associativity. This situation cannot occur when the entire panel fits into L1 cache memory, as is the case with models for which QNNPACK is optimized.

2.4.3 Experiments and Results

We have applied Static Quantization that quantizes weights on a per-channel basis and picks quantization parameters for activations in an optimal manner as explained before in Paragraph 2.4.2.

This technique has been first applied on the baseline ResNet50 (trained on CIFAR10 as before), then on the ResNet50 that has been pruned using the iterative pruning scheme. We have chosen the one that had 90.23% accuracy, which was obtained at 70% pruning rate (further pruning brought accuracy below 90%).

As we can see in the Table 2.1, impressive results were achieved by quantizing these networks. Memory usage was reduced by 75% right away ($4\times$ compression), due to the conversion from FP32 to INT8 for both models, pruned and original. As for execution time, it decreased by more than half for when applied on the original model, and 74% (i.e $3.85\times$ speed-up) when applied on the pruned ResNet50.

Table 2.1: Optimization statistics for ResNet50.

Model	Accuracy (%)	Size (MB)	Speed-up	Compression
Baseline	92.12	94.34	-	-
Pruned	90.23	37.77	$1.66\times$	$2.5\times$
Quantized	91.90	23.60	$2.13\times$	$4\times$
Pruned+Quantized	89.56	9.44	$3.85\times$	$10\times$

2.5 Conclusion

In this chapter, we have presented general optimization techniques for software in embedded systems and focused on DNN pruning and quantization. Both techniques are applied on ResNet-50 trained on CIFAR 10 dataset. We have achieved impressive results: almost $10\times$ compression, and $3.85\times$ speed-up with less than 3% drop in accuracy. We point out, however, that ResNet-50 is originally created and trained to classify ImageNet dataset that contains 1000 classes. Thus, applying these methods on the original model and eventually on ImageNet dataset, would probably not yield the same performance as for CIFAR10 (images distributed into only 10 classes). Nevertheless, it does serve as a solid proof of concept.

Chapter 3

Application and Deployment

3.1 Introduction

After successfully implementing and assessing two powerful techniques of DNN optimization, we have applied them on one of the recent models that perform a common computer vision task, that is, object detection. The considered model that we have worked on is *RetinaNet* trained on COCO dataset. The model has been put through the optimization pipeline. Afterward, the results of the baseline and those of the optimized model are contrasted.

In Addition, this chapter covers the deployment of an object classifier on an Android smartphone. The aim behind this is to find out if the two techniques implemented offer the same level of performance improvements that we have seen on a PC. But before getting into the details, we first clear out the new concept of object detection, and highlight its potential.

Humans glance at an image and instantly know what objects are in the image, where they are, and how they interact. Detection and recognition of objects in the observed scenes is a natural biological ability. The human visual system is fast and accurate, allowing us to perform complex tasks like driving with little conscious thought. Computer vision is a field of artificial intelligence that trains computers to interpret and understand the visual world. Using digital images from cameras and deep learning models, machines can accurately locate, identify, and classify objects. Each month, tens of papers are published with new ideas, algorithms, etc. For this reason, we focus our work on optimizing this task (object detection).

Object detection deals with predicting the location of an object as well as its class in an image. This is done regardless of scale, location, pose, view with respect to the camera, partial occlusions, and illumination conditions.

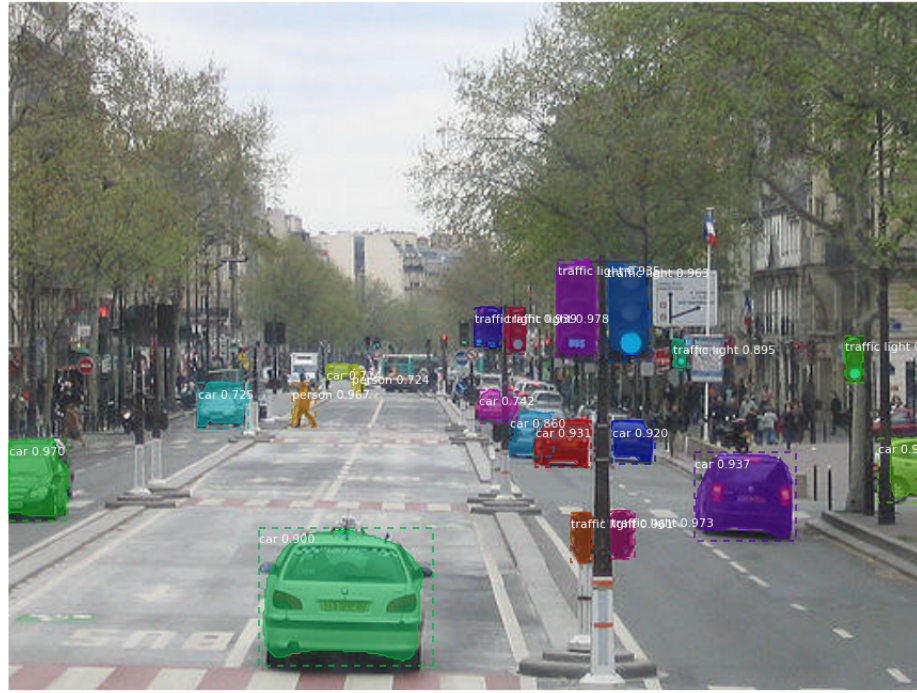


Figure 3.1: Example of object detection and segmentation [50]

This task is breaking into a wide range of industries, with use cases ranging from personal security to productivity in the workplace. it is applied in many areas with the most popular ones being:

- Automotive: allowing computers to drive cars without specialized sensors.
- Security: intelligent video surveillance systems
- Healthcare: lesion detection in medical images, computer-assisted diagnosis, and intervention.

In learning-based object detection, there are four primary object detector families one can find:

- Region-Based Convolutional Neural Network(R-CNN) [59], [14].
- Single Shot Detector (SSD) [45], [32], [74].

- You Only Look Once (YOLO) [56], [57], [58].
- RetinaNet [43], [13]

These families revolve around two design approaches. The first approach is based on a two-stage architecture, such as Faster R-CNN (Region-based Convolutional Neural Networks). The first stage is composed of a Region Proposal Network (RPN) that generates regions of interest. These regions are fed to the second stage for object classification and bounding-box regression. The other design uses one-stage detectors such as YOLO-v3 (You Only Look Once) and RetinaNet. These models treat object detection as a simple regression problem by taking an input image and learning the class probabilities and bounding box coordinates. We have opted for RetinaNet since it has a number of characteristics that we have been looking for. More specifically, it has ResNet50 as a backbone. This model is familiar to us since we have performed extensive optimization experiments on it. Additionally, it utilizes ReLU as an activation function which is supported in the Pytorch quantization API, unlike YOLO-v3 for example, that relies on leaky-ReLU (unsupported).

3.2 COCO dataset

Microsoft COCO: Common Object in Context [44], is a large-scale dataset that addresses three core research problems in scene understanding: Object detection, segmentation and captioning. The 2014 version that we have used contains images, bounding boxes, and labels. Images are gathered from complex everyday scenes containing common objects in their natural context. This dataset contains photos of 80 object types that can be recognized easily. Instances are labeled in 164,062 images. The total size of the dataset is 37.57GB. The images are split into 3 groups, the training set that comprises 82,783 images, validation containing 40,504 images, and the rest (40,775 images) are the test set (the split is approximately $\frac{1}{2}$ train, $\frac{1}{4}$ validation, and $\frac{1}{4}$ test).

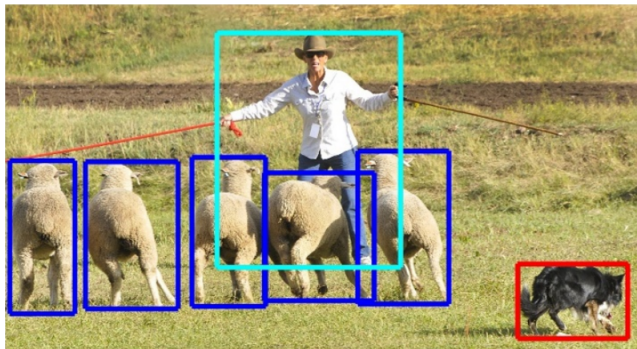


Figure 3.2: An image with bounding boxes from COCO dataset [44].

3.3 RetinaNet

RetinaNet has been developed by Facebook AI Research (FAIR) in 2018, by Lin and *al* [43]. It is a one-stage object detection model that consists of a backbone network and two specialized subnetworks. The backbone is a combination of ResNet50 (or 101) and a feature pyramid network (FPN). This backbone is mainly responsible for computing a convolutional feature map across the entire input image. The output of this block is fed to two sub-networks. The first sub-network performs convolutional object classification; the second sub-network is tasked with performing convolutional bounding-box regression (Figure 3.10).

3.3.1 The Backbone

The pattern computing backbone is composed of:

- A ResNet (50 or 101) model is used for deep feature extraction (Section 1.4).
- Feature Pyramid Network (FPN) is used on top of ResNet for constructing a rich multi-scale feature pyramid from one single resolution input image [42].

The scale of objects in an image poses a huge challenge in the object detection task. In a traditional bottom-up convolutional networks (Figure 3.3 b), the object detection at different scales is not very accurate. It is especially true if there is a big size discrepancy. One way to solve this problem is to feed the network multiple scales of the same image (called featurized image pyramid ,Figure 3.3 a), but this method is too slow because these scales are fed sequentially to the network. A much faster way of extracting features at different scales is to use the inherent pyramidal feature hierarchy that a CNN computes layer by layer (with sub-sampling layers). This feature hierarchy is treated as if it were a featurized image pyramid (Figure 3.3 c). The problem with this approach is that the resulting featurized image pyramid suffers from large semantic gaps because of the different depths from which it is extracted. The high-resolution feature maps have low-level features that harm their representational capacity for object recognition.

The FPN network circumvents the problems of the semantic gaps with the addition of lateral connections along with a top-down pathway. Thus, this approach combines low-resolution, semantically strong features with high-resolution, semantically weak features (Figure 3.3 d), which results in a lot more robustness to scale.

Anchors

The anchors are bounding boxes with pre-defined scales and aspect ratios that cover objects of different shapes. At the center of each sliding-window location in the FPN output, and at

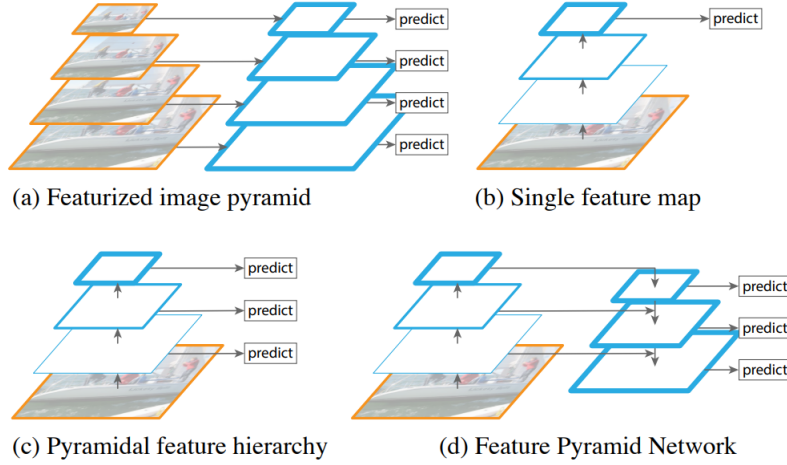


Figure 3.3: RetinaNet network architecture [43].

each pyramid level, the anchor boxes are simultaneously generated. The maximum number of possible boxes for each location is denoted as A . RetinaNet uses these anchors in the matching phase throughout the training. During the process, the appropriate anchor box is matched with the bounding boxes of each ground truth object within an image. Essentially, the anchor box that has the highest extent of overlap with an object is responsible for predicting the class and the location of that object. The RetinaNet model accounts for three aspect ratios (1:2, 1:1, and 2:1) as well as three scales (2^0 , $2^{\frac{1}{3}}$, $2^{\frac{2}{3}}$) in all the levels of the pyramid. With a total of 9 anchor boxes (combination of the 3 scales and 3 ratios) for each pyramid level, the model is able to detect objects with a great variety of shapes and scales.

3.3.2 Classification Sub-network

This classification sub-network predicts the probability of object presence at each spatial position for each of the A anchors and K object classes (80 in COCO). The sub-network is a fully convolutional network (FCN¹) that applies four 3×3 convolutional layers, each with C channels and each followed by ReLU activations, followed by a 3×3 convolutional that has a channel for each anchor associated with a class. There are K classes and A anchors for each class, therefore, the number of channels of this last layer is KA .

3.3.3 Box Regression Sub-network

This sub-network is an FCN connected to each pyramid level for the purpose of regressing the offset from each anchor box to a nearby ground-truth object if it exists. It is identical to the classification sub-network except that it terminates With $4A$ (4 coordinates for each of the A

¹FCN: A convolutional network that does not contain fully connected layers

anchors) linear outputs per spatial location. It is a class-agnostic bounding box regressor that uses fewer parameters.

Figure 3.10 depicts how the different sub-networks are connected in the RetinaNet model.

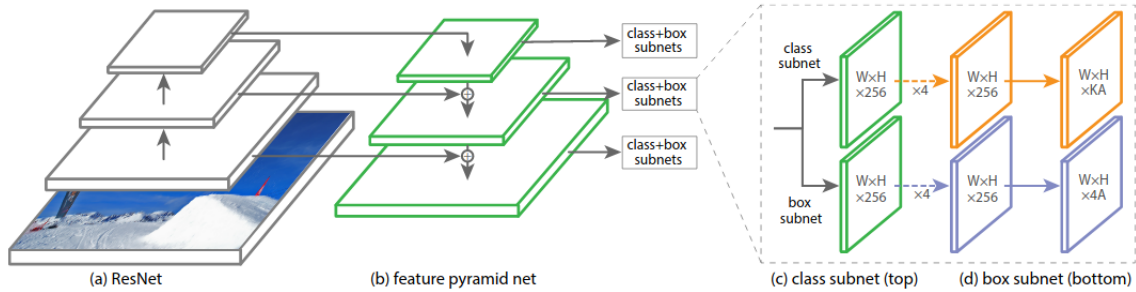


Figure 3.4: RetinaNet network architecture [43].

3.4 RetinaNet-50 Optimization

Our objective is to optimize RetinaNet-50 in order to evaluate the pipeline's (pruning and quantization) effectiveness on an object detector model. At the first stage, we have performed iterative pruning, with a 5% step at each iteration. We have chosen a step of 5% because we have noticed that RetinaNet could hardly recover with a greater pruning step. 40% was the maximum pruning rate we could achieve. Further pruning results in an intolerable loss in the original accuracy (more than 5% incurred error in both Average Recall and Average Precision).

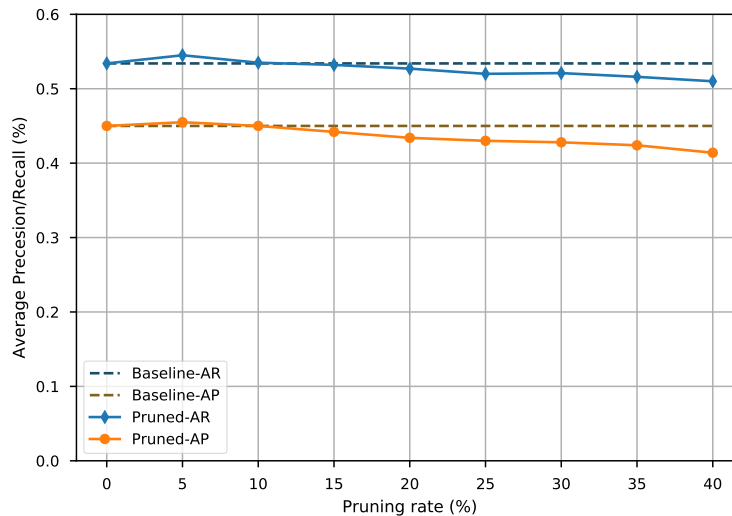


Figure 3.5: Retinanet-50 - iterative pruning: Precision/Recall.

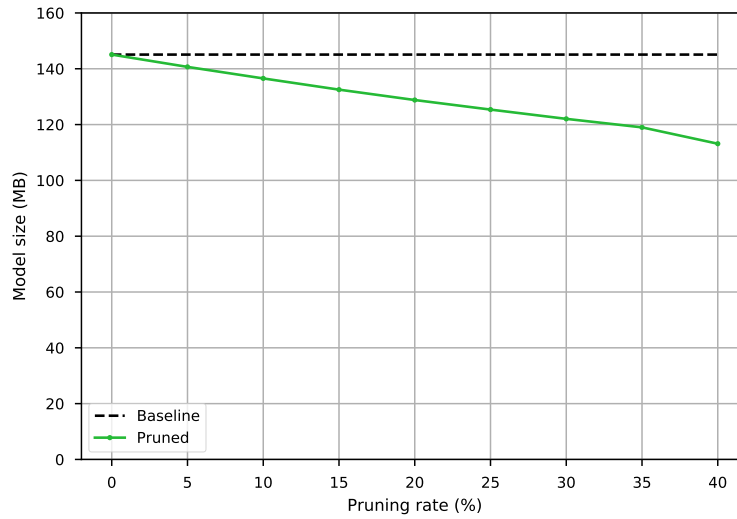


Figure 3.6: Size reduction through Iterative pruning.

A remarkable note in Figure 3.5 above is that RetinaNet-50 performed better after 5% pruning (with fine-tuning of course), and that is explained by the fact that Pruning is a form of regularization.

Table 3.1 showcases the results of optimizing RetinaNet-50-500¹.

Table 3.1: Optimizations statistics for Retinanet-50-500.

Model	Average Precision	Average Recall	Size (MB)	Inference (ms)	Compression	Speed-up
Baseline	45.0	53.4	145.08	536	-	-
Pruned 40%	41.4	51.0	113.14	266	1.3×	2×
Quantized	43.6	50.8	36.27	282	4×	1.9×
Fully optimized	38.9	49.2	28.81	101	5×	5.3×

RetinaNet-50-500¹: A RetinaNet model with ResNet50 as a backbone that takes an image with a size of 500 pixels.

Upon the application of an iterative pruning that removes 40% of the channels in each 3×3 convolutional layer, we observe in Table 3.1 that the size of the model drops from 145.08 MB to 113.14 MB ($1.3\times$ compression). The inference time also drop to 266 ms when it was originally 536 ms ($2\times$ speed-up). Applying quantization alone also gave impressive results. The size if reduced with a compression factor of $4\times$ whereas the speedup almost doubled. When the full pipeline is applied (pruning + quantization), we observe and impressive $5.3\times$ speed-up as well as a compression factor of $4\times$. The accuracy drop was a tolerable 6.1%.

3.5 Mobile application development

Our embedded platforms of choice are mobile devices. They provide a rich environment in terms of deep neural network deployment. With that being said, a lot of progress needs to be made in order for it to be considered a fully functional deep learning framework. Since we have used Facebook's Pytorch for the implementation of the optimization techniques, it is only natural that we use Pytorch Mobile to ensure a consistent end-to-end workflow to deliver the edge-inference tests. But before that, a brief description of the Android development tools used is a must. The next section deals with the important concepts, covering both the executed background modules as well as user interface layouts.

3.5.1 Android General Concepts

Application components

Activities

An activity is a class that represents one screen of the user interface. It is very much comparable to a window in Microsoft Windows. An android app may contain one or more screens, meaning, one or more activities. In our case, we have opted for a minimalist design with only one activity. a sequence of callback methods are associated with an activity life cycle, Figure 3.7 illustrates the chain of events when an activity is launched. The main activity is the first activity that is created, much like the main function is the first function that gets executed in a C/C++ program.

- **onCreate()** is called when the activity is created for the first time.
- **onStart()** is called as soon as the activity is visible on the screen.
- **onResume()** is executed when interaction with the application starts.
- **onPause()** is called when the the activity is no longer receiving input from the user.
- **onStop()** executes when the activity is no longer visible on the screen.

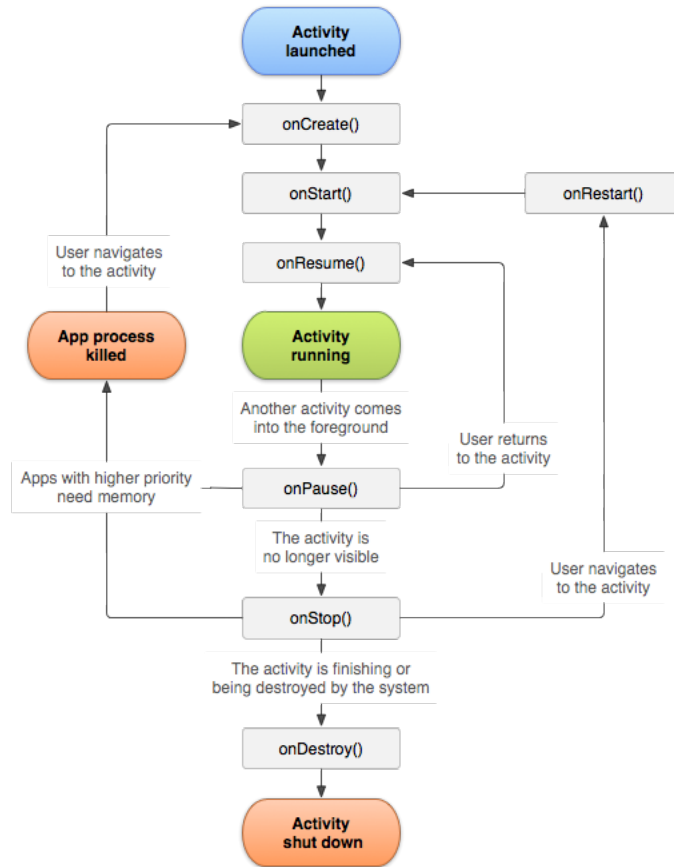


Figure 3.7: Activity lifecycle.

- **`onDestroy()`** is executed before the activity is destroyed by the operating system.

For our purposes, the only method that we need to override is the `onCreate()` method. the other callbacks will be called from the parent class.

Services

A service is an application component that performs long-running operations in the background. no user interface is provided during its life cycle. A service can be in two states, *Bound*, and *Started*. When a service is in the *Started*, that means that an application component such as an activity has started it by calling `startService()`. the service runs in the background indefinitely even if the calling element is destroyed. To create a service, a new class that extends the *Service* base class has to be created. In our case, we will need to run the inference in the mobile device using a service.

Intents

an *Intent* is an abstract description of an operation. It can be used for various actions, launching

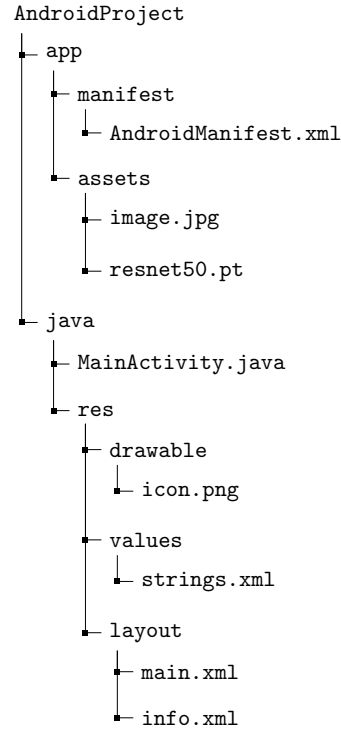


Figure 3.8: Android project organization.

an activity, broadcasting a message to other apps, or communicating with background services. For our purposes, we will use intents to communicate with the gallery application in order to retrieve the image for inference.

Resources

Android application is comprised of more elements other than code. These elements include images, videos, layout definitions, or any other type of static content that the code may need throughout the life cycle of the application. These resources are organized in such a way that they are always maintained separately in their respective sub-directories. A typical android studio project is organised in the hierarchy shown below:

The */manifest* directory contains what's called the android manifest, which is a file that defines the structure and metadata of the application, including its components and requirements. for example, if the application needs to access the camera or Bluetooth modules, then it must be stated in the manifest file as a requirement for the application to run properly, and access permissions to these modules will have to be declared as well. the */java* directory contains all the java code necessary, including the classes of the different activities and other custom classes that are used. *MainAcvity.java* is the main activity as its name suggests. this is the entry point to the application when it is first launched.

User Interface

The most basic, yet fundamental element for the user interface is a **View** object which is an instantiation of the **View** class. It takes for the form of a rectangle on the screen and takes care of drawing shapes as well as handling events. View is also the base class for many UI components such as buttons, text fields, etc.

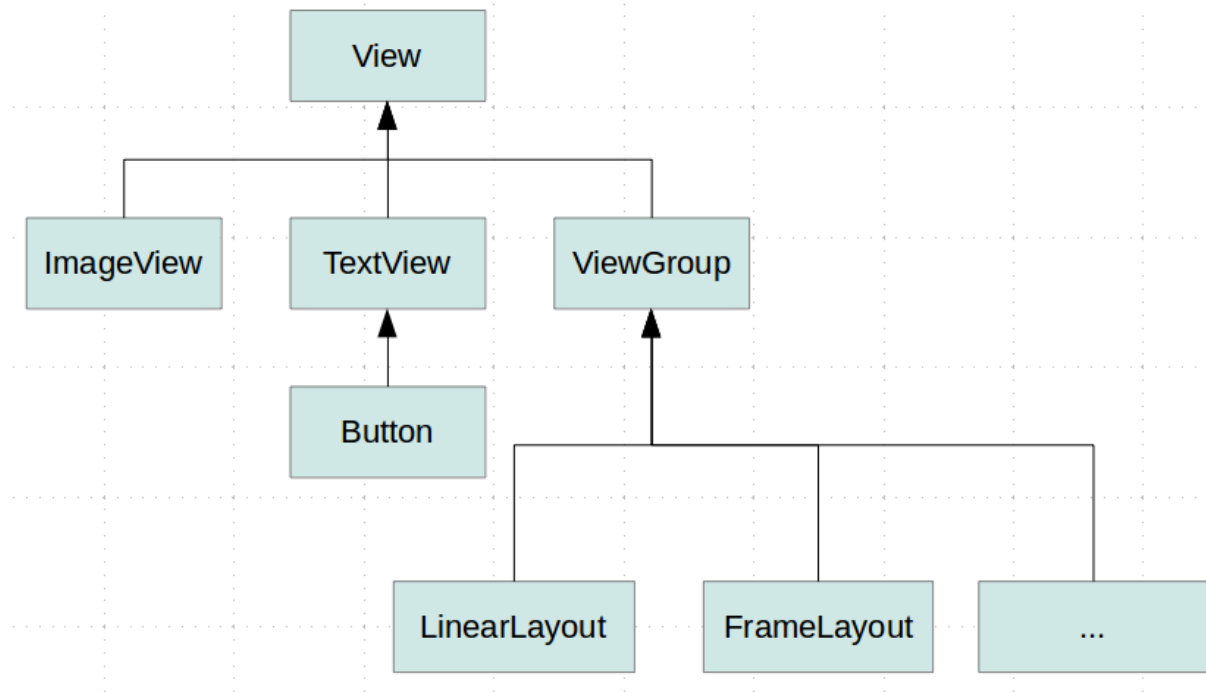


Figure 3.9: The View class and its children.

In our application we need the **View** class to contains the buttons and the image that will be fed to the CNN model.

Controls

As previously stated, we tried to keep the application as minimal as possible, containing only the necessary functionality for loading the model, the input image, and running the inference, as a result, we opted to include two buttons, one for loading the image, and one for running the classification process.

Event Handling

Events are a way to gather information about the user's interaction with application components that allow it. like button presses or screen touch. The Android operating system treats such events as a queue, meaning that events are handled on a first-in, first-out basis. We bind event listeners to the buttons displayed in our application such that when the user clicks on them, the

corresponding code will be executed immediately. the event handler in this case is the *onClick()* callback.

Application development workflow

The Resnet50 model is optimized by applying iterative pruning and reducing the number of weights of some layers by 60%. Then per-channel quantization is applied, which converts the parameters and activations of the model from float32 to int8. After that, we convert the model to what is called a Torchscript format that can be deployed in a mobile device. once the model is ready, we put it in the assets folder of the android project (Figure 3.8).

3.5.2 Profiling the application

Upon inspection of the memory usage in time, which was performed on a *Condor Allure M1* smartphone that uses an ARMv7 CPU, one notices right off the bat that both applications (one using the baseline model and one using the optimized model) start at about the same size in the memory heap. As soon as the models are loaded (after around 3 seconds), a huge size discrepancy is observed. The baseline application maintains a 117 MB memory usage, while the optimized application reserves 47 MB. At the 10 seconds mark, however, after inference is initiated, we see that the baseline application peaks at 309 MB and does not stop running until 37 seconds are passed, whereas the optimized application stops much earlier (17 seconds mark) with a peak memory usage of only 120 MB.

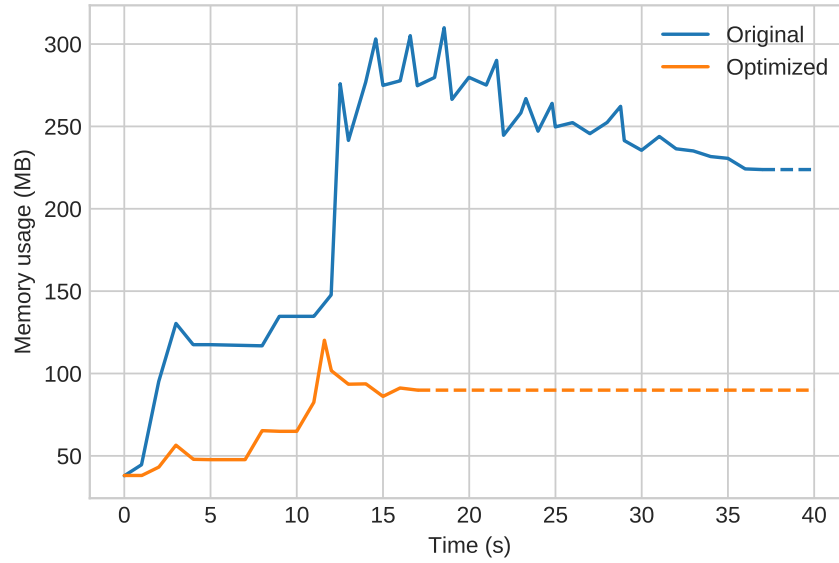


Figure 3.10: Profiling memory usage on a smartphone

Figure 3.11 is a capture from Android profiler that shows the CPU work load. Colored in green is the time where inference dominated CPU usage. We see how intensive inference is, therefore keeping it in short duration is extremely beneficial.

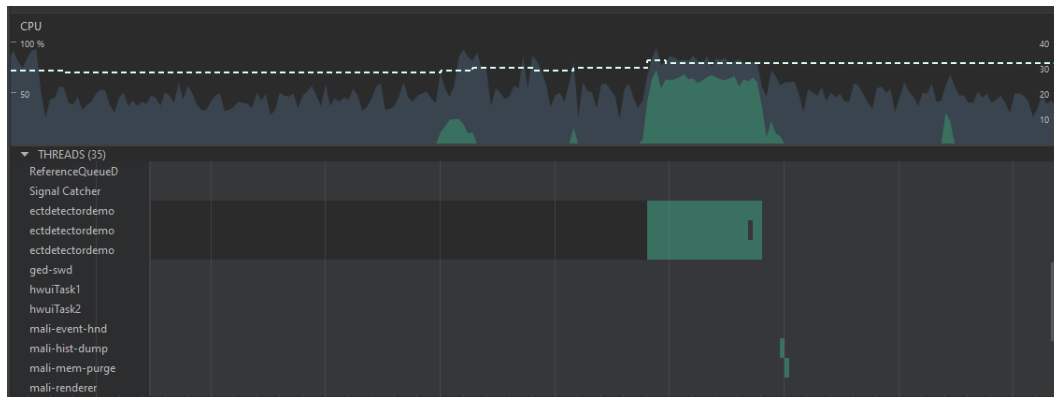


Figure 3.11: CPU usage profiling, the application uses 4 threads to perform inference, CPU peaks at 65 %

Table 3.2 presents the Loading and Inference duration for both baseline model and optimized one. After the model was optimized we gained an impressive $4.8\times$ speed-up (on PC it was $3.3\times$). We note however that the loading time becomes greater. We suspect that it is due to additional quantization support libraries that may be loaded as well.

Figure 3.12 shows screenshots of the Android application.

Table 3.2: Statistics for loading and inference times

Trial	Loading time (ms)		Inference time (ms)	
	Baseline	Optimized	Baseline	Optimized
1	1076	1197	25055	5128
2	1065	1172	25556	5390
3	1069	1237	25212	5126
4	1072	1222	25342	5261
5	1058	1229	24865	5156
Average	1068	1211	25206	5212

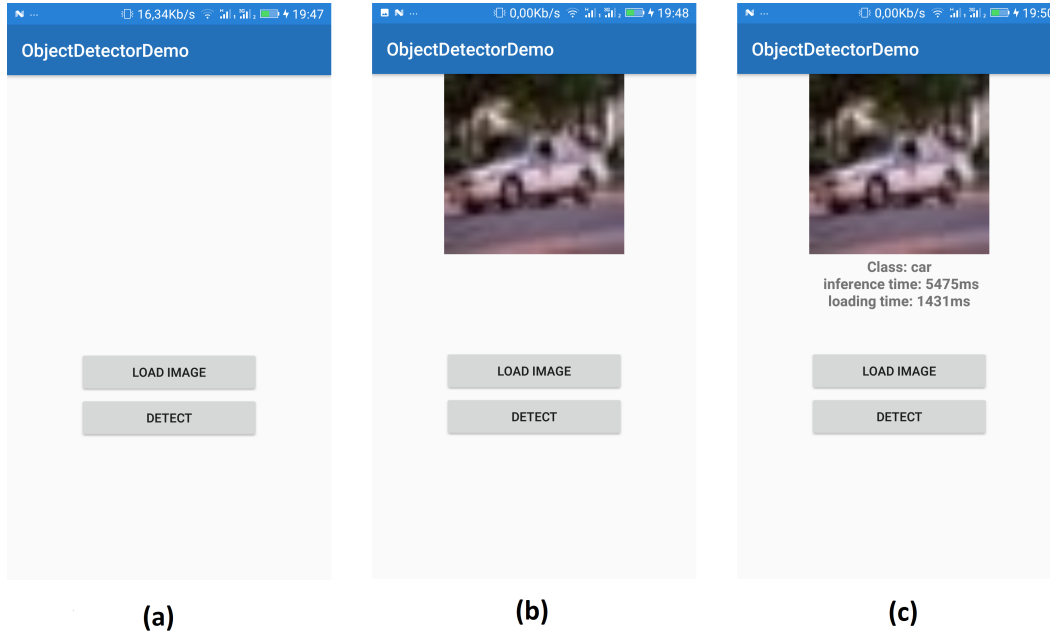


Figure 3.12: (a): Launch screen. (b): Image loaded. (c): Inference performed.

3.6 Conclusion

In this last chapter:

- We shed light on object detection, one of the most challenging computer vision tasks out there, which truly puts the methods that we implemented to the test. The optimization pipeline has been applied on the Retinanet-50 trained on COCO dataset. We have been able to showcase the effectiveness of the two techniques in the context of object detection. The performance improvements were comparable to those obtained by optimizing ResNet50 on CIFAR10.
- A fully optimized classification model (ResNet50) has been deployed in an embedded de-

vice (smartphone). The inference performance is contrasted with that of the baseline model. In the absolute, the optimized model showed that real-time inference is still hard to achieve on a smartphone. However, relatively speaking, the performance improvement over the baseline model is remarkable.

Conclusion

In many Artificial Intelligence applications such as smart industrial robots, autonomous vehicles, and speech recognition, deep neural networks are constrained by latency, energy consumption, and model size. This dissertation has focused on optimizing DNNs for efficient inference on embedded platforms with limited resources. The objective is to compress the model and accelerate the inference (i.e: reducing its memory footprint and execution time) while preserving the accuracy.

We have started our work by conducting thorough research in the literature. Many techniques have been proposed to address this problem. Most of them revolve around two techniques: Network *pruning* and *quantization*. Each technique comes with many ingenious approaches. The first technique that we implemented is Filter Pruning, based on the L2-norm criterion. The second technique is post-training Quantization which is applied at the channel level. The optimization pipeline that we devised integrates pruning and quantization which have the property of being orthogonal, i.e. they are able to compress the network without interfering with each other. Thus, leading to surprisingly high optimization rates. We set up the benchmark of our pipeline on ResNet50. Then to demonstrate its efficiency in a typical embedded system application, the model has been deployed on an Android smartphone. Furthermore, we applied the optimization procedure in the context of object detection. The model of choice has been RetinaNet object detector trained on COCO dataset.

Future work The implemented techniques in this dissertation showed some promising results. However, there still is plenty of room for future improvement, such as implementing state-of-the-art pruning methods (e.g. AMC [24]) or developing new approaches, since the criterion that we used (L2-norm) is still not as accurate as what state-of-the-art offers. Network Quantization could be further improved as well. Very recently, a paper entitled "Training with quantization noise for extreme model compression"[12] sets a new state-of-the-art for combining high accuracy and small model size. Hence, exploiting it would be on our top priorities in future work.

The optimization API of the Pytorch deep learning framework could be improved further as well. At the time of writing this dissertation, this API is still in beta version, and additional features need to be implemented in order to truly leverage the power of DNN optimization. For example,

some special object detection operations are still not supported, like quantization support for the leaky-ReLU activation function. Something which made it very difficult for us to optimize the original target object detector model, YOLO-v3 (which forced us to switch to Retinanet). Additionally, considering GPU support for quantization on PCs, as well as GPU support for mobile DNN inference on Pytorch Mobile would bolster optimization performance even further. The latter feature is a work in progress with the introduction of the Android Neural Network API [17], but for now, it only supports Tensorflow.

Removing these obstacles would open a vast array of possibilities in this promising field. Of course, it goes without saying that we will keep on investigating future contributions, and we will improve on our work as more features are supported.

Bibliography

- [1] Alex Krizhevsky, *The cifar-10 dataset*, [accessed 9-July-2020]. [Online]. Available: <https://www.cs.toronto.edu/~kriz/cifar.html>.
- [2] J. D. A. Mishra E. Nurvitadhi, "WRPN: wide reduced-precision networks," *Computing Research Repository*, vol. abs/1709.01134, 2017.
- [3] W. Baxter and H. R. Bauer, "The program dependence graph and vectorization," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89, Austin, Texas, USA: Association for Computing Machinery, 1989, 1–11, ISBN: 0897912942. DOI: 10.1145/75277.75278. [Online]. Available: <https://doi.org/10.1145/75277.75278>.
- [4] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994. [Online]. Available: <https://ieeexplore.ieee.org/document/279181>.
- [5] N. Buduma, *Fundamentals of Deep Learning, Designing Next-Generation Machine Intelligence Algorithms*. O'REILLY, 2017, ch. The Neural Network. [Online]. Available: <https://www.oreilly.com/library/view/fundamentals-of-deep/9781491925607/>.
- [6] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," *international conference on frontiers in handwriting recognition*, Oct. 2006. [Online]. Available: <https://hal.inria.fr/inria-00112631/document>.
- [7] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," *CoRR*, vol. abs/1504.04788, 2015.
- [8] Cole Murray, *Deep learning cnn's in tensorflow with gpus*, [accessed 22-June-2020]. [Online]. Available: <https://hackernoon.com/deep-learning-cnns-in-tensorflow-with-gpus-cba6efe0acc2?gi=d95857a52c74>.
- [9] Cyrille Kone, *A layman's guide to deep neural networks*, [accessed 13-June-2020]. [Online]. Available: <https://towardsdatascience.com/a-laymans-guide-to-deep-neural-networks-ddcea24847fb>.

- [10] M. Dukhan, "The indirect convolution algorithm," *CoRR*, vol. abs/1907.02129, 2019. arXiv: 1907.02129. [Online]. Available: <http://arxiv.org/abs/1907.02129>.
- [11] Facebook Engineering developpers, *Qnnpack: Open source library for optimized mobile deep learning*, [accessed 9-July-2020]. [Online]. Available: <https://engineering.fb.com/ml-applications/qnnpack/>.
- [12] A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jégou, and A. Joulin, "Training with quantization noise for extreme model compression," *ArXiv*, vol. abs/2004.07320, 2020.
- [13] C.-Y. Fu, M. Shvets, and A. C. Berg, "Retinamask: Learning to predict masks improves state-of-the-art single-shot detection for free," *ArXiv*, vol. abs/1901.03353, 2019. [Online]. Available: <https://arxiv.org/pdf/1901.03353.pdf>.
- [14] R. B. Girshick, "Fast r-cnn," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448, 2015. [Online]. Available: <https://arxiv.org/pdf/1504.08083.pdf>.
- [15] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, Jan. 2010. [Online]. Available: https://www.researchgate.net/publication/215616968_Understanding_the_difficulty_of_training_deep_feedforward_neural_networks.
- [16] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," *CoRR*, vol. abs/1412.6115, 2014.
- [17] Google, *Android neural network api*, [accessed 19-June-2020]. [Online]. Available: <https://developer.android.com/ndk/guides/neuralnetworks>.
- [18] H. H. Sultan, N. Salem, and W. Al-Atabany, "Multi-classification of brain tumor images using deep neural network," *IEEE Access*, vol. PP, pp. 1–1, May 2019. DOI: 10.1109/ACCESS.2019.2919122. [Online]. Available: <https://ieeexplore.ieee.org/document/8723045>.
- [19] S. Han, "Efficient methods and hardware for deep learning," PhD thesis, Stanford university, Sep. 2017. [Online]. Available: <https://stacks.stanford.edu/file/druid:qf934gh3708/EFFICIENT%20METHODS%20AND%20HARDWARE%20FOR%20DEEP%20LEARNING-augmented.pdf>.
- [20] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 1135–1143. [Online]. Available: <https://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf>.

- [21] B. Hassibi, D. G. Stork, and G. Wolff, "Optimal brain surgeon: Extensions and performance comparisons," in *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., Morgan-Kaufmann, 1994, pp. 263–270. [Online]. Available: <http://papers.nips.cc/paper/749-optimal-brain-surgeon-extensions-and-performance-comparisons.pdf>.
- [22] K. He and J. Sun, "Convolutional neural networks at constrained time cost," *CoRR*, vol. abs/1412.1710, 2014. arXiv: 1412.1710. [Online]. Available: <http://arxiv.org/abs/1412.1710>.
- [23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: <http://arxiv.org/abs/1512.03385>.
- [24] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *ECCV*, 2018. [Online]. Available: <http://arxiv.org/abs/1802.03494>.
- [25] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *CoRR*, vol. abs/1707.06168, 2017. [Online]. Available: <http://arxiv.org/abs/1707.06168>.
- [26] J. C. Huang and T. Leng, "Generalized loop-unrolling: A method for program speedup," in *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, 1999, pp. 244–248. [Online]. Available: <https://ieeexplore.ieee.org/document/756775>.
- [27] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in neural information processing systems*, 2016, pp. 4107–4115. [Online]. Available: <http://papers.nips.cc/paper/6573-binarized-neural-networks>.
- [28] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: <http://arxiv.org/abs/1502.03167>.
- [29] Irene Lizihui, *An introduction to activation functions*, [accessed 17-June-2020]. [Online]. Available: <https://ychai.uk/notes/2019/05/10/NN/Activation-functions-introduction/>.
- [30] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2018. [Online]. Available: <https://arxiv.org/pdf/1712.05877.pdf>.
- [31] N. Johnson and A. Mycroft, "Using multiple memory access instructions for reducing code size," vol. 2985, Mar. 2004, pp. 265–280. DOI: 10.1007/978-3-540-24723-4_18. [Online]. Available: <https://www.cl.cam.ac.uk/~am21/papers/cc04.pdf>.

- [32] T. Kong, F. Sun, H. Liu, Y. Jiang, and J. Shi, "Consistent optimization for single-shot object detection," *ArXiv*, vol. abs/1901.06563, 2019. [Online]. Available: <http://arxiv.org/abs/1901.06563>.
- [33] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018. [Online]. Available: <https://arxiv.org/pdf/1806.08342.pdf>.
- [34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, issn: 0001-0782. doi: 10.1145/3065386. [Online]. Available: <https://doi.org/10.1145/3065386>.
- [35] A. Kumar, "New trends and challenges in source code optimization," *International Journal of Computer Applications*, vol. 131, no. 16, pp. 27–32, 2015. doi: 10.5120/ijca2015907609. [Online]. Available: <https://doi.org/10.5120/ijca2015907609>.
- [36] V. Lebedev and V. S. Lempitsky, "Fast convnets using group-wise brain damage," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2554–2564, 2016. [Online]. Available: <http://arxiv.org/abs/1506.02515>.
- [37] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," D. S. Touretzky, Ed., pp. 598–605, 1990. [Online]. Available: <http://papers.nips.cc/paper/250-optimal-brain-damage.pdf>.
- [38] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, Contour and Grouping in Computer Vision*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 319–345. doi: 10.1007/3-540-46805-6_19. [Online]. Available: https://doi.org/10.1007/3-540-46805-6_19.
- [39] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016. [Online]. Available: <https://arxiv.org/pdf/1605.04711.pdf>.
- [40] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016. [Online]. Available: <http://arxiv.org/abs/1608.08710>.
- [41] S. Lin, R. Ji, Y. Li, C. Deng, and X. Li, "Toward compact convnets via structure-sparsity regularized filter pruning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, pp. 574–588, 2020. [Online]. Available: <http://arxiv.org/abs/1901.07827>.
- [42] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016. arXiv: 1612.03144. [Online]. Available: <http://arxiv.org/abs/1612.03144>.

- [43] T.-Y. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2999–3007, 2017. [Online]. Available: <http://arxiv.org/abs/1708.02002>.
- [44] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," *ArXiv*, vol. abs/1405.0312, 2014. [Online]. Available: <http://arxiv.org/abs/1405.0312>.
- [45] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, 2016, pp. 21–37. [Online]. Available: <https://arxiv.org/pdf/1512.02325.pdf%EF%BC%89>.
- [46] I. Lizihui, *Deep learning 05: Talk about convolutional neural networkscnn*, [accessed 17-June-2020]. [Online]. Available: <https://ireneli.eu/2016/02/03/deep-learning-05-talk-about-convolutional-neural-networkcnn/>.
- [47] J. Luo and J. Wu, "An entropy-based pruning method for CNN compression," *CoRR*, vol. abs/1706.05791, 2017. [Online]. Available: <http://arxiv.org/abs/1706.05791>.
- [48] N. Ma, X. Zhang, H. Zheng, and J. Sun, "Shufflenet V2: practical guidelines for efficient CNN architecture design," *CoRR*, vol. abs/1807.11164, 2018. arXiv: 1807.11164. [Online]. Available: <http://arxiv.org/abs/1807.11164>.
- [49] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 13–20. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2017_workshops/w29/papers/Mao_Exploring_the_Granularity_CVPR_2017_paper.pdf.
- [50] (2018). Mask r-cnn for object detection and segmentation, [Online]. Available: https://gitee.com/DSCjxgzdzd/Mask_RCNN.
- [51] D. Mishkin and J. E. S. Matas, "All you need is a good init," *CoRR*, vol. abs/1511.06422, May 2016. [Online]. Available: <https://arxiv.org/abs/1511.06422>.
- [52] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," 2016. arXiv: 1611.06440. [Online]. Available: <https://arxiv.org/pdf/1611.06440.pdf>.
- [53] J. Peng, S. Kang, Z. Ning, H. Deng, J. Shen, Y. Xu, J. Zhang, W. Zhao, X. Li, W. Gong, J. Huang, and L. Liu, "Residual convolutional neural network for predicting response of transarterial chemoembolization in hepatocellular carcinoma from ct imaging," *English (US), European Radiology*, vol. 30, no. 1, pp. 413–424, Jan. 2020, issn: 0938-7994. doi: 10.1007/s00330-019-06318-1. [Online]. Available: <https://pubmed.ncbi.nlm.nih.gov/31332558/>.

- [54] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999, ISSN: 0893-6080. DOI: [https://doi.org/10.1016/S0893-6080\(98\)00116-6](https://doi.org/10.1016/S0893-6080(98)00116-6). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0893608098001166>.
- [55] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," pp. 525–542, 2016. [Online]. Available: <https://arxiv.org/pdf/1603.05279.pdf>.
- [56] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," pp. 779–788, 2015. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf.
- [57] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2017. [Online]. Available: <http://arxiv.org/abs/1612.08242>.
- [58] —, "Yolov3: An incremental improvement," *ArXiv*, vol. abs/1804.02767, 2018. [Online]. Available: <https://arxiv.org/pdf/1804.02767.pdf>.
- [59] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, pp. 1137–1149, 2015. [Online]. Available: <https://arxiv.org/pdf/1506.01497.pdf>.
- [60] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014. arXiv: 1409.0575. [Online]. Available: <http://arxiv.org/abs/1409.0575>.
- [61] H. M. S. Han and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *Computing Research Repository*, vol. abs/1510.00149, 2016. [Online]. Available: <https://arxiv.org/pdf/1510.00149.pdf>.
- [62] M. Sahni. (). Making neural nets work with low precision. [accessed 20-June-2020].
- [63] A. Saxe, J. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," Jan. 2014, pp. 1–22. [Online]. Available: <https://arxiv.org/abs/1312.6120>.
- [64] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. Lecun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *International Conference on Learning Representations (ICLR) (Banff)*, Dec. 2013. [Online]. Available: <https://arxiv.org/abs/1312.6229>.

- [65] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv 1409.1556*, Sep. 2014. [Online]. Available: <https://arxiv.org/abs/1409.1556>.
- [66] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *CoRR*, vol. abs/1505.00387, 2015. arXiv: 1505.00387. [Online]. Available: <http://arxiv.org/abs/1505.00387>.
- [67] Sumit Saha, *A comprehensive guide to convolutional neural networks — the eli5 way*, [accessed 22-June-2020]. [Online]. Available: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
- [68] D. Sussillo, "Random walks: Training very deep nonlinear feed-forward networks with smart initialization," *CoRR*, vol. abs/1412.6558, 2014. arXiv: 1412.6558. [Online]. Available: <http://arxiv.org/abs/1412.6558>.
- [69] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. arXiv: 1409.4842. [Online]. Available: <http://arxiv.org/abs/1409.4842>.
- [70] Tarang Shah, *About train, validation and test sets in machine learning*, [accessed 9-July-2020]. [Online]. Available: <https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7>.
- [71] B. van der Zwaag, "Using domain-specific basic functions for the analysis of supervised artificial neural networks," Undefined, SAS 34-039, PhD thesis, EL/SAS, Netherlands, Dec. 2003, ISBN: 90-365-2008-8. [Online]. Available: <https://research.utwente.nl/en/publications/using-domain-specific-basic-functions-for-the-analysis-of-supervi>.
- [72] Vincent Dumoulin, Francesco Visin, *Convolution arithmetic*, [accessed 9-July-2020]. [Online]. Available: https://github.com/vdumoulin/conv_arithmetic.
- [73] H. Wang, Q. Zhang, Y. Wang, and H. Hu, "Structured pruning for efficient convnets via incremental regularization," *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2019. [Online]. Available: <http://arxiv.org/abs/1811.08390>.
- [74] T. Wang, R. M. Anwer, H. Cholakkal, F. S. Khan, Y. Pang, and L. Shao, "Learning rich features at high-speed for single-shot object detection," in *The IEEE International Conference on Computer Vision (ICCV)*, 2019. [Online]. Available: https://openaccess.thecvf.com/content_ICCV_2019/papers/Wang_Learning_Rich_Features_at_High-Speed_for_Single-Shot_Object_Detection_ICCV_2019_paper.pdf.
- [75] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082. [Online]. Available: <http://arxiv.org/abs/1608.03665>.

- [76] Wikipedia contributors, *Control-flow graph* — *Wikipedia, the free encyclopedia*, [accessed 23-June-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=942915804.
- [77] X. Xu, M. S. Park, and C. Brick, "Hybrid pruning: Thinner sparse networks for fast inference on edge devices," *CoRR*, vol. abs/1811.00482, 2018. [Online]. Available: <http://arxiv.org/abs/1811.00482>.
- [78] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society Series B*, vol. 68, pp. 49–67, Feb. 2006. doi: 10.1111/j.1467-9868.2005.00532.x. [Online]. Available: <http://www.columbia.edu/~my2550/papers/glasso.final.pdf>.
- [79] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. arXiv: 1311.2901. [Online]. Available: <http://arxiv.org/abs/1311.2901>.
- [80] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *Computing Research Repository*, vol. abs/1702.03044, 2017. [Online]. Available: <https://dblp.org/rec/journals/corr/ZhouYGXC17.bib>.