University of Liège

Faculty of Applied Sciences

# Machine Learning under resource constraints

*Author*
Nathan Greffe

*Supervisors*
Prof. Pierre Geurts
Jean-Michel Begon


Master thesis submitted for the degree of MSc in Computer Science and Engineering

Jury Members: Prof. Quentin Louveaux, Prof. Gilles Louppe, Prof. Louis Wehenkel

Academic Year 2018-2019

# Abstract

Nowadays, Machine learning on embedded devices, for example smartphones, is a popular topic. This arises from the growing concern of the public for data privacy and the general usefulness of running a service without the need of an external server.

Many methods exist to reduce the inference time of different algorithms but they are not often compared together or combined. The goal of this thesis is thus to offer a review of some of these methods. The scope of this work is limited to image classification using Convolutional Neural Networks on a Raspberry Pi 3B. CIFAR-10 was used as a dataset and out of the many benefits of embedded devices friendly CNNs, we limited ourselves to inference time. In other words, our goal was to classify images on CIFAR-10 as accurately as possible for a given inference time.

The methods investigated and our main conclusions are the following:

- We compared different architectures between each other and modified them to increase their performances. There, we managed to improve the error rates by adding Squeeze-and-Excitation blocks to existing MobileNetv1/v2 and MnasNet architectures.

- Based on recent works [Crowley et al., 2018, Liu et al., 2018], we modified several pruning algorithms to adapt architectures by changing the number of channels per layer. This did not show promising results in our case. We suspect, however, that this is related to the dataset we used and might be worthwhile on bigger datasets like ImageNet.

- We used knowledge distillation on the architectures obtained from our search. Knowledge distillation takes profit of the predictions of a big network to help training a smaller one. This pushes the accuracies of some networks appreciably further.

- We tried to use Tensorflow's quantization to decrease the inference time of the previous architectures at moderate costs in accuracy. However, these methods are not mature as of now and did not give any result.

In conclusion, our initial objective of reviewing many methods and testing their interaction has been completed. Some of these methods showed rather shy results in our experiments on the CIFAR-10 dataset and the Raspberry Pi 3B. We believe, however, that more significant improvements could be obtained in different settings, as shown in several publications. Our breadth-first study also allowed to highlight several directions that would deserve deeper exploration.

# Acknowledgments

I want to express my gratitude towards the persons that helped me throughout this thesis here:

- First, both of my supervisors, Pierre Geurts and Jean-Michel Begon, who helped me a lot during these 4 months. For all our meetings, their pointers towards the papers we built this work upon, their many pieces of advice and their very useful and thorough feedback.
- All of the people involved in the elaboration of the alan cluster, for the 833+ gpu-hours I consumed there. Especially to Joeri Hermans that let me run a few experiments in priority towards the end of my thesis.
- My friend Maxime Noirhomme that helped me with the correction of the writing of this thesis.

To all of them, a huge thank you.

# Contents

# Chapter 1

# Introduction

Nowadays, deep neural networks are omnipresent in computer vision and natural language processing. Several applications in these fields require or would benefit from embedding on mobile devices (face identification on a picture on a smartphone, semantic segmentation of the video input of a drone or a car, digital assistants,...). A metric of primary importance in this regard is inference time. However, what architecture/training procedure to use in order to get the optimal performance/inference time tradeoff remains unknown. The corresponding literature also often uses abstract metrics (number of parameters, number of FLOPS – floating point operations) that are not always well correlated with inference time on real devices [Ma et al., 2018, Tan et al., 2018, Yang et al., 2018]. Additionally, several techniques (creating new architectures from scratch, pruning architectures at runtime, quantizing/binarizing weights and possibly activations) exist but are not compared between one another.

The goal of this work is to review and inspect the state-of-the-art and to compare different approaches. Additionally to the accuracy of the network, we mostly consider the inference time as the metric of interest. We choose to measure inference time on a Raspberry Pi 3B, to have a device that was both representative of the embedded world and well known. The scope of this thesis is also limited to image classification problems.

The body of this work is divided into several chapters, with the idea of splitting different techniques between different chapters:

In Chapters 1 and 6, we respectively introduce this work and draw a conclusion.

In Chapter 2, we present the nomenclature used throughout this work as well as some details on the methodology.

In Chapter 3, we introduce several popular architectures to perform efficient inference as well as their main building blocks. We then compare those between each other and modify them to increase their performances.

In Chapter 4, we modify several pruning algorithms to find the number of channels to allocate to each layer of a WideResNet. Pruning algorithms consists in modifying a wide, pretrained, network and to iteratively prune it while updating its weights. The idea developped here is to use the architecture discovered by pruning, while diregarding the obtained weights. Thus, performing some sort of architecture search.

In Chapter 5, we use two different techniques to improve further the architectures obtained from the previous chapters. The first technique is knowledge distillation which consists in using the predictions of a bigger network to help a smaller one training. The second one is quantization, which consists in reducing the representational power of the

weights of the network, for example, to use 8-bit integers instead of 32-bit floating-points to perform faster inference. We use Tensorflow to perform quantization.

We realized this work to give a broad exploration of the fascinating topic of inference time efficient deep learning image classification. This was chosen over an in-depth overview to test as many different approaches as possible. This means we opened several doors for improvements and deeper analyses in future works.

# Chapter 2

# Concepts and Methodology

This chapter describes the nomenclature used in the present report as well as the test settings used throughout the thesis.

## 2.1 Nomenclature

To avoid confusions between different *neural networks* parts, we will define some nomenclature used in the rest of this thesis. Although we redefine the nomenclature to avoid confusions, the basic working of a neural network is assumed to be known:

- A *neural network* is composed of several *subnetworks*, these *subnetworks* handle *tensors* having the same width and height and are a repetition of *blocks* having the same structure.
- A *block* is composed of several layers, the most simple example is the Bottleneck *block* which corresponds to a $1 \times 1$ convolutional *layer* followed by a $3 \times 3$ convolutional *layer* (see Section 3.2.2 for more details).
- A *layer* is either a linear operation, for example, a convolution or a matrix multiplication (fully-connected layer) or a pooling operation (pooling layer). *Layers* often come with *activations*, *batch normalization* and/or *dropout*.

  Mathematically, a fully-connected layer takes as input a 1-D *tensor* $x$ and returns $y = W \times x + b$, where $W$ is a 2-D tensor and $b$ a 1-D *tensor* with the same shape as $y$. A convolution layer is an operation taking as input a 3-D *tensor* of shape $Tensor_w \times Tensor_h \times NumChannels_{in}$ and computing its cross-correlation (convolution is an abuse of language actually) with $NumChannels_{out}$ *tensors* of shape $Conv_w \times Conv_h \times NumChannels_{in}$.
- *Activations* are non-linear operations, for example applying a Relu activation to a *tensor* will output another *tensor* with the same shape where all the elements smaller than 0 in the first tensor are replaced by 0's.
- *Batch Normalization* is an operation consisting in normalizing its input with respect to the rest of the batch, i.e. subtracting the per-component mean and dividing by the per-component variance of the batch of data.
- *Dropout* is an operation consisting in randomly zeroing out a proportion of its input at training time.

- *tensors* are the successive transformations that are made to the input data by convolutional layers. They usually are 4-dimensional, in which case their dimensions are *batch* size, number of *channels*, *width* and *height*. The *batch* size is sometimes omitted for simplification.

- A *batch* is a group of images (or more generally tensors) that are processed together.

- A *channel* is the third dimension of an image, for example an RGB image is composed of 3 channels (Red, Green and Blue). The number of channels increases in the network and tensors can contain several hundreds of those.

Figure 2.1 presents a schematic illustration of these concepts.

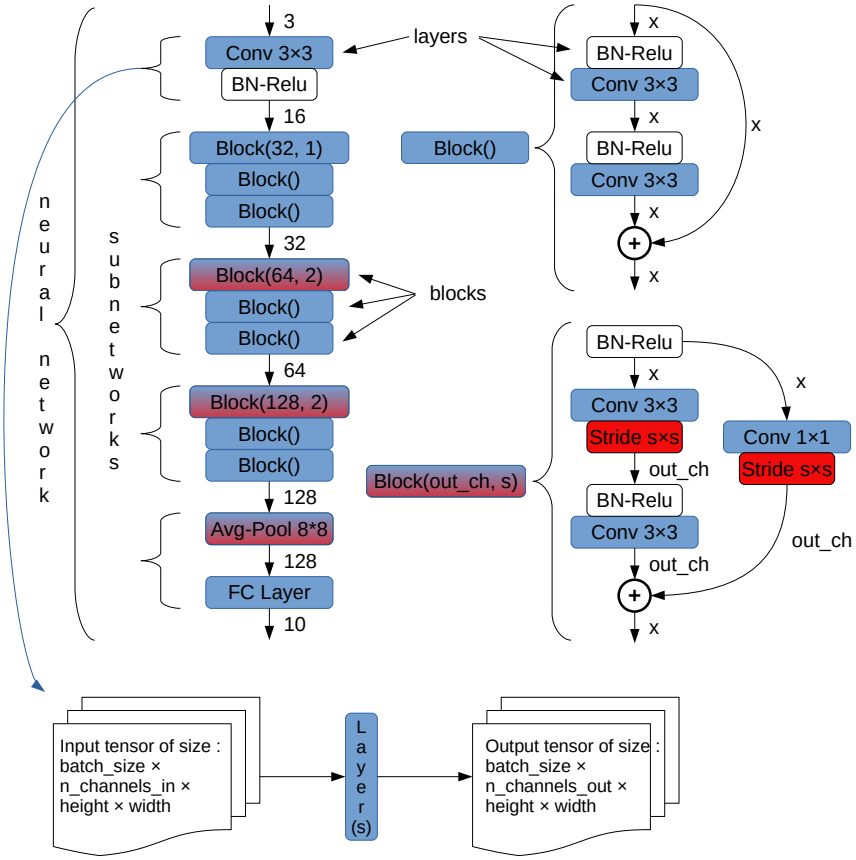



Figure 2.1: Illustration of the nomenclature used on a WideResNet-22-2 on CIFAR-10 [Crowley et al., 2018]
See Figure 2.2 for the meaning of the elements of this scheme

## 2.2 Scheme conventions

To make that document as clear and understandable as possible, we made the network block diagrams ourselves. Figure 2.2 shows the conventions we used for the different blocks.

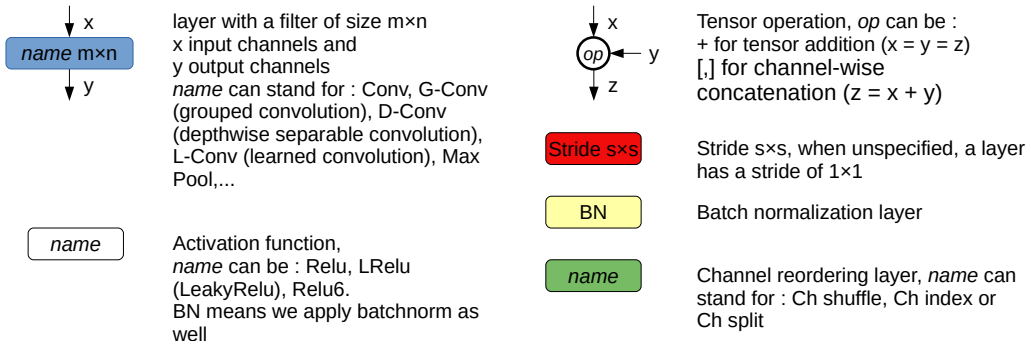



Figure 2.2: The conventions used in the schemes

## 2.3 Performance measurement setup

As explained in the Introduction, the goal of this work is to build models that are as efficient as possible while remaining accurate. To do so, we used Tensorflow lite 1.13.1 on a Raspberry Pi 3B. Tensorflow lite is a part of the Tensorflow deep learning framework that is dedicated to perform inference on embedded devices. The Tensorflow lite version running on the Raspberry Pi was cross-compiled using Docker.

The Raspberry Pi 3B is a lightweight and cheap ($\approx$ 40 €) computer designed to be affordable to anyone. The card posses a Quad Core 1.2GHz Broadcom BCM2837 CPU (ARM) processor (OS running on 32 bits) and 1GB of RAM.

The inference time measurement given in the rest of this work come from tf-lite files executed on the Raspberry Pi. We used a C++ benchmark program that came alongside tf-lite and a small bash script we coded ourselves that wrapped around the C++ program.

To compute the inference time of a network, our script report the average time over 3 requests to the C++ program. We computed the inference time for several networks at a time and the relative order of the networks was randomly changed between each of the 3 runs. To avoid overheating the CPU, the process sleeps for 10 seconds between each call to the benchmark tool.

To compute the inference time, the C++ program makes warming up predictions for a second and then report the average inference time over 50 runs (each inference predicts only one image).

We launched our script 2 times on around 20 networks and the maximal relative difference between the 2 predictions ($rel\_pred\_diff = 2 \times \frac{\|pred\_1 - pred\_2\|}{pred\_1 + pred\_2}$) was of 2%. We thus decided to consider the inference time as a deterministic signal in the other chapters.

To train our networks, we used Tensorflow 1.13.1 and PyTorch 1.0.2, which were the most recent versions of these libraries when we started this Master Thesis. They both run on top of CUDA 10.1 and cuDNN 7.4. The hardware used for training was a desktop computer with a GTX 970 graphics card, an Intel i5 4670 and 16 GB of RAM in addition to Montefiore's Arya and later Alan clusters (GTX 1080Ti and later RTX 2080Ti).

We limited ourselves to the CIFAR-10 [Krizhevsky, 2009] dataset. An interesting extension would be to perform the same experiments on more complex datasets like ImageNet [Russakovsky et al., 2015]. CIFAR-10 is a well-known dataset that is often used to evaluate pruning techniques. It consists of 50.000 training and 10.000 testing RGB images

of size $32 \times 32 \times 3$. Each images is to be assigned to one of 10 different classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck). The main disadvantage of this dataset is that the minimal size of its input images is not very realistic, i.e. there are few computer vision problems in which we have to classify $32 \times 32$ images.

The data augmentation we used throughout this project was straightforward, at both training and testing time, we normalized each image (normalize each of the 3 channels with respect to the training dataset statistics). Additionally, we performed padding of 4 pixels in every spatial dimension by reflecting the borders of the image and then took a $32 \times 32$ random crop in the resulting $40 \times 40$ image. Finally, with a probability of 50%, we flip the image horizontally.

## 2.4 Code

The code is also available on Github[1]. Regarding the separation between what we implemented and did not, the contents of the folders *CondenseNet*, *morph_net*, *PyTorch-prunes* were initially cloned from other repositories (https://github.com/ShichenLiu/CondenseNet, https://github.com/Tensorflow/models/tree/master/research, and https://github.com/BayesWatch/PyTorch-prunes respectively). *CondenseNet* was barely modified, *morph_net* did not contain the training code (data loading, optimizer, model architecture,...), which we added and we experienced on *PyTorch-prunes* quite a lot. We implemented the contents of *NetAdapt* and *training_from_scratch* entirely ourselves.

[1] https://github.com/NatGr/Master_Thesis

# Chapter 3

# Model Architectures

## 3.1 Introduction

The simplest way to have a Convolutional Neural Network that is accurate while being reasonably fast and lightweight is to design it from scratch by paying attention to its resource consumption. Initially, the only goal was to perform as well as possible on ImageNet [Lin et al., 2014]. AlexNet [Krizhevsky et al., 2012] and VGG [Simonyan and Zisserman, 2014] are good examples of that approach. On the other hand, GoogLeNet [Szegedy et al., 2015] was designed while taking his computational budget into account and SqueezeNet [Iandola et al., 2016] was created to stay as accurate as possible while diminishing the number of parameters. Recently, some works on reinforcement-learning based architecture search (for example [Zoph et al., 2018, Tan et al., 2018]) have been able to produce lightweight architectures. However, these require an immense amount of GPU time for the architecture search and are thus only affordable to big companies.

There are several advantages to use small and optimized architectures, the first one that drew attention was the memory footprint, an example of that focus is SqueezeNet [Iandola et al., 2016] that was explicitly built to reduce model size. A smaller network is also faster to train for the researcher that builds it as well as faster to predict while consuming less energy for the final user. Finally, it can be expected to generalize better (assuming it is not too small) due to the well known bias-variance tradeoff, although recent works go in the opposite direction [Belkin et al., 2018].

In this work, we only focus on the inference time.

Initially, the number of FLOPS, or even worse, the number of parameters, was considered as a proxy for inference time. [Wong, 2018] used a metric based on both FLOPS and parameters. The number of FLOPS was shown to be proportional to the inference time by [Canziani et al., 2016]. This analysis is, however, mostly based on classical architectures and does not hold for more recent and original choices.

More recent works like [Sandler et al., 2018, Tan et al., 2018, Ma et al., 2018] use inference times on real devices to compare different models, [Gholami et al., 2018] discussed the implications of their design on real hardware.

Finally, most architectures are based on the repetition of a given block in the same fashion as ResNets [He et al., 2016a]. The search problem thus reduces into the problem of finding the best block. Doing so is simpler but also suboptimal since blocks at different depths operate on different tensors having different spatial resolutions (width and height).

Thus, the optimal block at depth $i$ might not be the same as the optimal block at depth $j$. A notable exception is [Tan et al., 2018], a network search approach that designed its search space so as to allow using different types of blocks at different depths. Section 3.2 sums up the different building blocks used to build more efficient CNNs, while Section 3.3 presents different architectures of interest based on the blocks.

## 3.2 Base Building Blocks

### 3.2.1 Tensor addition and concatenation

In basic neural network architectures, the input of block $i$ is simply the output of block $i-1$, the latter being the result of a single convolution operation. As demonstrated by ResNets [He et al., 2016a], introducing skip connections, i.e. giving the outputs of block $i-1$ and $i-2$ as inputs to block $i$ (see Figure 3.1), can help the model converge. Based on the success of the later, [Huang et al., 2017] introduced DenseNets in which the input of a block inside of a subnetwork is connected to the output of all the previous blocks (these outputs are concatenated). Tensor concatenation (on the number of channels axis) was also used in Inception [Szegedy et al., 2015] but this time, to combine the outputs of different convolutions operations (see Figure 3.2). A constraint of these operations is that the tensors need to have the same width and height, in the case of addition, they must also have the same number of channels.

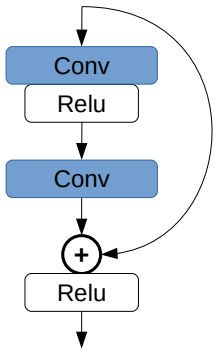



Figure 3.1: A skip connection (number of channels not specified), as used in ResNets [He et al., 2016a]

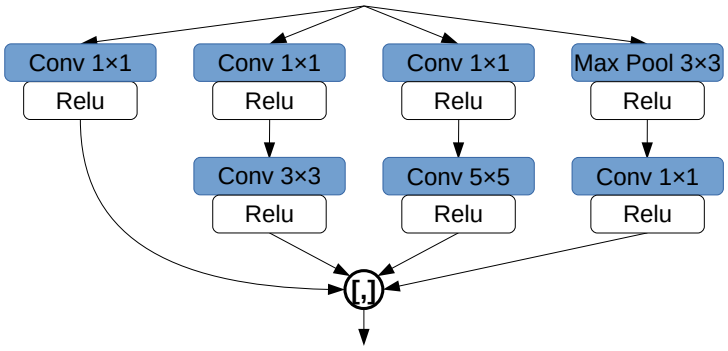



Figure 3.2: An inception module (number of channels not specified), see [Szegedy et al., 2015]

### 3.2.2 Bottleneck

The bottleneck structure is very intensively used, among others by [Szegedy et al., 2015, He et al., 2016a, Huang et al., 2017, Iandola et al., 2016, Zagoruyko and Komodakis, 2016, Gholami et al., 2018, Sandler et al., 2018]. The idea is to reduce the number of channels by using the first convolution before performing a second convolution that restores the number of channels to its original size. This idea is similar to embeddings, where a low dimensional embedding can keep much information about a larger space [Szegedy et al., 2015]. The computational price of a convolutional layer being given by

$$Tensor_w \times Tensor_h \times Conv_w \times Conv_h \times NumChannels_{in} \times NumChannels_{out}$$

where $w$ and $h$ subscripts mean width and height. Thus, dividing the number of channels inside of the bottleneck by $n$ is $n$ times less costly than not having a bottleneck.

There exist many different flavours of bottlenecks, the first convolution is usually of size $1 \times 1$ and the second $3 \times 3$ or $5 \times 5$, which would be denoted $B(1,3)$ and $B(1,5)$ respectively. There are also variants with 3 layers, for example, $B(1,3,1)$ in [He et al., 2016a] which has the advantage to be much cheaper since the $3 \times 3$ convolution that is 9 times more resource consuming than an equivalent $1 \times 1$ has a smaller number of input and output channels. For example, the 2 middle columns of Figure 3.2 are $B(1,3)$ and $B(1,5)$ respectively.

### 3.2.3 Depthwise separable convolutions

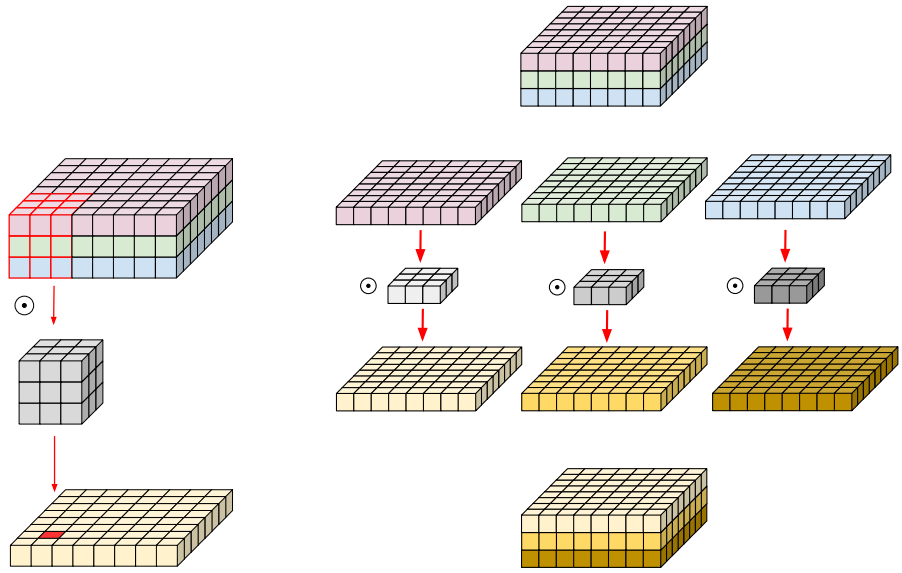
The idea behind depthwise separable convolutions is to only use the channel at depth $i$ of the input to compute the channel at depth $i$ of the output. Figure 3.3 illustrates that approach. This choice makes depthwise convolutions $NumChannels_{in}$ times cheaper to compute than their classical equivalent. The different channels are then recombined by regular $1 \times 1$ Convolutions (called pointwise convolutions) in most cases. This approach is used by [Chollet, 2017, Howard et al., 2017, Zhang et al., 2018, Freeman et al., 2018, Sandler et al., 2018].

On the other hand, [Gholami et al., 2018] argued that depthwise separable convolutions had weak hardware performance because of their poor ratio of computing to memory operations. Indeed, in depthwise convolutions, an element of the input is only used in $Conv_w \times Conv_h$ computations whereas it is used $Conv_w \times Conv_h \times NumChannels_{out}$ times in the classical convolution. One mitigation to that problem could be to use the same tensors for several depthwise separable convolutions and then add/concatenate the outputs.

Whether there are activations or not vary between the different works, [Chollet, 2017] (Xception) do not put any after the depthwise separable convolution, [Sandler et al., 2018] (MobileNetv2) do not put any after the pointwise convolution and [Howard et al., 2017] (MobileNetv1) put them after both the depthwise separable and pointwise convolution.

### 3.2.4 Grouped convolutions

Grouped convolutions are a less extreme variant of depthwise convolutions where the input and output tensors are divided into $g$ independent groups, i.e. the output in one of these group does not depend on the input of the other groups. This is illustrated in Figure 3.4, additionally depthwise separable convolutions can be seen as grouped convolutions where $g = NumChannels_{in}$. Grouped convolutions reduce the cost of the convolution by a factor $g$ since we perform $g$ convolutions that are $g^2$ times less costly instead of one.



(a) Classical convolution (b) Depthwise convolution

Figure 3.3: Depthwise convolution, figures taken from eli.thegreenplace.net

These convolutions are used by CondenseNets and ShuffleNetv1 [Huang et al., 2018, Zhang et al., 2018]. It was also used by AlexNet [Krizhevsky et al., 2012] where it was to be able to train on 2 GPUs rather than to optimize inference time.

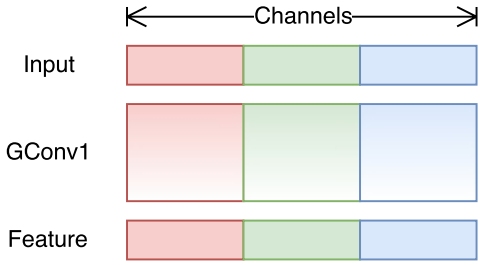



Figure 3.4: An illustration of grouped convolutions, figure taken from [Zhang et al., 2018]

### 3.2.5 Channel shuffling

When a network uses grouped convolutions, it can be interesting to reorder the channels in order to avoid having several groups of channels that are independent throughout the network, i.e. the first quarter of channels never interact with the last quarter of channels. This alternative is way less powerful than a fully interacting option. A solution to this problem, used by [Huang et al., 2018, Zhang et al., 2018, Ma et al., 2018], is to reshuffle the channels inside each block. This means that the channels are reordered after a grouped convolution as shown in figure 3.5.

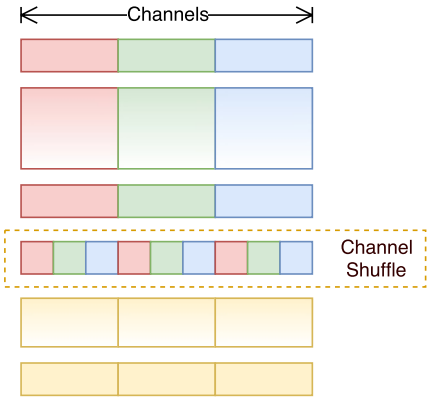



Figure 3.5: A channel shuffle layer inside of a ShuffleNetv1 block, see [Zhang et al., 2018]

### 3.2.6 1-D convolutions

Until now, convolutions always had the same width and height, the idea here is to decompose a $n \times n$ convolution into a $n \times 1$ and a $1 \times n$ convolution. By doing so, the number of FLOPS is reduced by $\frac{n^2}{2n} = \frac{n}{2}$. It is thus more interesting the more significant the filter gets. Since activations are introduced between the $n \times 1$ and $1 \times n$ layers, it can also be argued to increase the depth of the network. That property might, however, be undesirable since it reduces parallelism (thinner and deeper networks). This approach was used in Inception-v2/v3 [Szegedy et al., 2016b] and Inception-v4/ResNet [Szegedy et al., 2016a] in 2016 as well as EffNet [Freeman et al., 2018] and SqueezeNext [Gholami et al., 2018] in 2018. Inception variants used it on $7 \times 7$ and $3 \times 3$ filters while EffNet and SqueezeNext used it on $3 \times 3$ filters.

### 3.2.7 Squeeze-and-Excitation

Squeeze-and-Excitation blocks [Hu et al., 2018] are an improvement that can be added on top of every type of convolutional layers. The idea consists of learning a weight for each of the output channels of a block. These blocks allow giving more importance to specific channels depending on the input image. To give a crude example, the Squeeze-and-Excitation block might learn that horizontal edge detectors are essential for classification when many vertical edges are detected. Thus when channel $x$ that detects vertical edges has a significant average value, channel $y$ that detect horizontal edges is given substantial weight, whereas it won't be the case if channel $x$'s average value is low.

The design is explained on Figure 3.6, in the case that interests us, the Squeeze-and-Excitation blocks are placed after a convolution operation. We first use average pooling to compute the average value of each channel, these are called statistics about each channel (we could use something else than average pooling of course). Then, we compute the weights through a two fully connected layers network, more formally,

$$weight = Sigmoid(\boldsymbol{W}_2 \times Relu(\boldsymbol{W}_1 \times statistics)) \tag{3.1}$$

where $\boldsymbol{W}_2 \in \mathbb{R}^{C \times C/r}$ and $\boldsymbol{W}_1 \in \mathbb{R}^{C/r \times C}$, $r$ being an hyperparameter $> 1$ called reduction factor and $C$ the number of channels. This design was shown to improve results at a little cost since the Squeeze-and-Excitation block is much less computationally hungry than the convolution operation that precedes it.

This design can be used on any network by simply adding Squeeze-and-Excitation blocks after the last convolution of the base network's blocks.

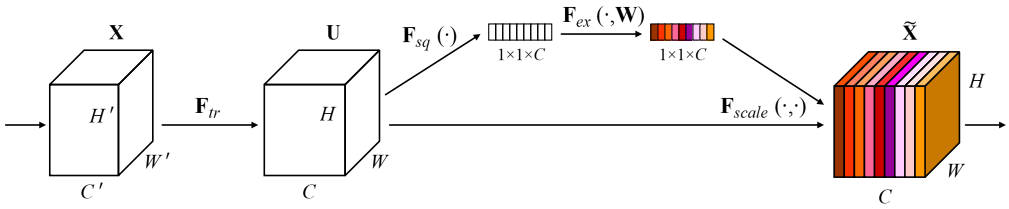



Figure 3.6: Squeeze-and-Excitation block, figure taken from [Hu et al., 2018] $\mathbf{F}_{tr}$ denotes a transfer operation (e.g. convolution), $\mathbf{F}_{sq}$ a squeeze operation that gets global channel statistics (e.g. average pooling), $\mathbf{F}_{ex}$ a non linear learnable mapping between the statistics and the weights of each channel and $\mathbf{F}_{scale}$ a scaling operation involving the channels weights (e.g. element-wise product)

## 3.3 Complete Architectures

This Section introduces many architectures built upon the blocks illustrated previously. Those will later be compared to see which ones are performing better for our problem. An excpetion is DenseNet that is introduced here because it is needed to understand CondenseNet. It is also used in Chapter 4.

### 3.3.1 (Wide)ResNet

ResNets [He et al., 2016a] introduced the concept of skip connection that we mentionned in Section 3.1. Although this architecture is pretty old relatively to the others, it is still very often used as a baseline. This is because of the groudbreaking impact [He et al., 2016a] had.

WideResNet [Zagoruyko and Komodakis, 2016] are an adaptation of ResNets [He et al., 2016a] that simply consists in multiplying the number of channels at each layer by a constant. For example, WideResNet-40-2 denotes a ResNet of depth 40 where the number of channels is two times bigger at each layer. [Zagoruyko and Komodakis, 2016] shows that it is more interesting to have wider ResNets than excessively deep ones both in terms of achievable accuracy and in terms of computational cost. WideResNets were also used extensively for the pruning chapter and are a hard to beat baseline for the other architectures. A WideResNet-40-2 architecture with inputs of size $32 \times 32$ is shown in Figure 3.7.

### 3.3.2 DenseNet

DenseNet [Huang et al., 2017] takes the idea of residual connections from [He et al., 2016a] even further by connecting the input of layer $i$ of a subnetwork to the outputs of layers $0, ..., i-1$ of the same subnetwork as illustrated in Figure 3.8. The channels to use as input are concatenated together rather than added. This is a fascinating idea because it reuses channels and allows to use the tensors of different depth, i.e. that represent information at a different level of coarseness in the same layer. The problem with this approach is

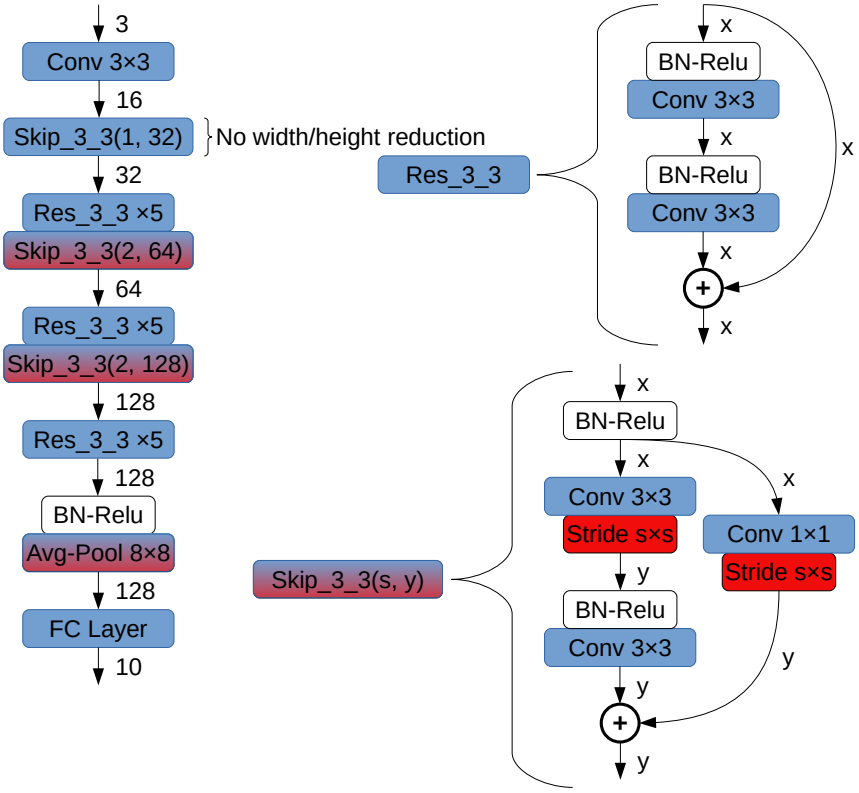



Figure 3.7: WideResNet-40-2 network (on CIFAR-10), see [Zagoruyko and Komodakis, 2016]
[Zagoruyko and Komodakis, 2016] chose to put the skip connection before applying batchnorm and activation, this was not done for other networks which is why the figure might look unfamiliar but is a detail

that concatenation is not efficiently implemented in current deep learning frameworks. As advocated by [Pleiss et al., 2017], DenseNet and its variations could be much more efficient if tensors were represented in Channel first order, i.e. $mem\_buffer[0..x]$ for channel 1, $mem\_buffer[x..2x]$ for channel 2 and so on... but they are represented with the batch size as the first dimension (because that is the way cuDNN chose). Indeed, in a Channel first ordering, one would have to allocate a large buffer and then to put the output of each layer next to another in memory which would perform the desired concatenation for free.

DenseNets have several hyperparameters, namely:

1. **Growth Rate:** If we denote the number of channels of the output of a layer by $k$, it follows that the $i^{th}$ layer of a subblock has $k_0 + k \times (i-1)$ input channels, thus $k$ can be small (e.g. $k = 12$), this parameter is denoted as growth rate.

2. **Bottleneck layers:** Since the number of input channels of a layer is much larger than its number of output channels, it is computationally interesting to introduce $B(1,3)$ bottlenecks ($1 \times 1$ convolution before the $3 \times 3$ convolution). The number of middle channels can for example be 4 times the output size, which is small with respect to the input.

3. **Compression Rate:** To reduce the size of the network, a good idea is to reduce the number of channels at transition blocks (between two subnetworks - when we

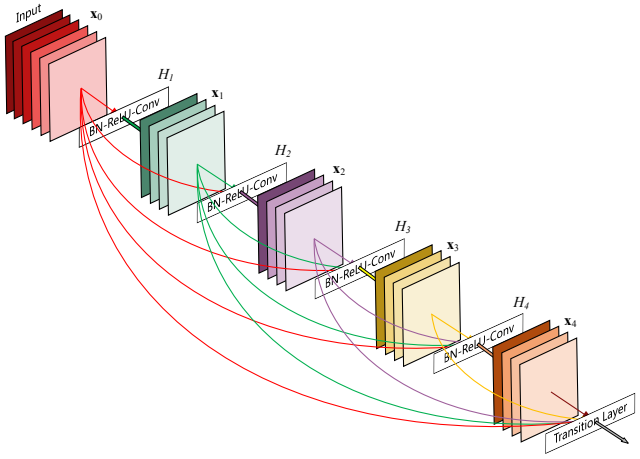



Figure 3.8: An illustration of a DenseNet subnetwork, figure taken from [Huang et al., 2017]

reduce the tensors spatial size). A good possibility is to reduce the number of layers by a factor of 2, here 2 would be called the compression rate.

A DenseNet with both compression and bottlenecks is referred to as DenseNet-BC.

As we can see from figure 3.9, which uses the average absolute filter weight as a metric of input importance, the output of some layers is more used than others. A basic modification to be done to DenseNets would thus be to severe some of the intermediary connections, for example, having the input of layer $i$ be only the output of layers $i-1$, $i-2$ and $i-3$ or $i-1$, $i-2$, $i-4$ and $i-8$. This idea was explored in [Hu et al., 2017] and gave birth to Log-DenseNet; it also inspired [Ma et al., 2018] to introduce their channel-split layer.

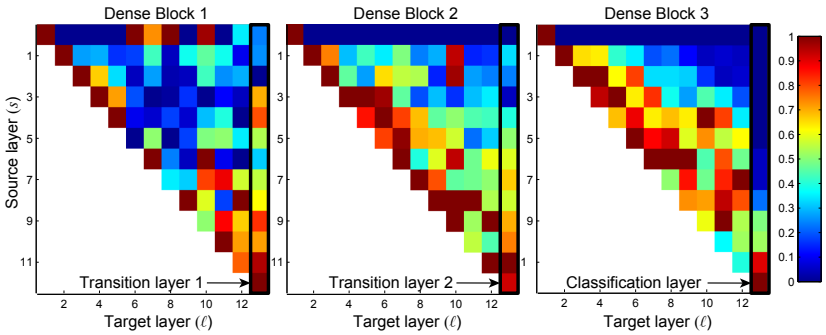



Figure 3.9: DenseNet connectivity diagram, Figure taken from [Huang et al., 2017] The color highlights the importance of the output of source layer $s$ for target layer $l$, there is one map per subnetwork

### 3.3.3 CondenseNet

CondenseNet [Huang et al., 2018] is an improvement over DenseNet that can be seen as a form of connections pruning. The main idea is to use grouped convolutions instead of the classical convolutions in a DenseNet, but instead of separating the channels based on

their offsets which would result in independent groups of channels trained separately, the mapping between channels and groups is learned. Connections between several blocks are also pruned.

We first define some hyperparameters and architecture specifications.

$G$ denotes the number of groups in the grouped convolutions. A each group in a grouped convolution can use $\frac{1}{C}$ of the input channels, The parameter $C$ is called condensation factor. In standard group convolutions, $C = G$ and this is the value used in practice since doing other renders the implementation more difficult, but the authors let the possibility to do otherwise.

The base DenseNet architecture was also modified in two ways: First, the growth rate increases exponentially, i.e. a channel at depth $i$ (starting from 0) has $2^i k_0$ output channels, where $k_0$ is a constant. Second, the dense connectivity is not limited to a subnetwork anymore, i.e. layers from different subnetworks are directly connected as well (difference in tensors width and height are handled through average pooling).

A CondenseNet block is composed of a learned group convolution followed by a shuffling block and then a grouped convolution (see Figure 3.10). Initially, the learned convolutions are separated into $G$ groups, each being connected to all of the input channels. The training occurs in $C$ steps, at the end of each of the first $C-1$ steps, $NumChannels_{in}/C$ channels are removed from each group, such that ultimately only $NumChannels_{in} \times (1 - (C-1)/C) = NumChannels_{in}/C$ remains. The removed channels are the ones whose associated filters have the smallest $L_1$ norm, and group lasso is introduced to encourage groups to use the same subset of input channels. This training procedure is illustrated in figure 3.11. After training, the learned group convolution can be replaced by an index (reorder the channels with repetition) and a normal group convolution layer. By its nature, this approach might seem quite unstable, i.e. different runs would result in different connections being pruned, but this was shown not to be the case in the paper. CondenseNet works also way better than traditional pruning approach applied on DenseNets [Liu et al., 2017].

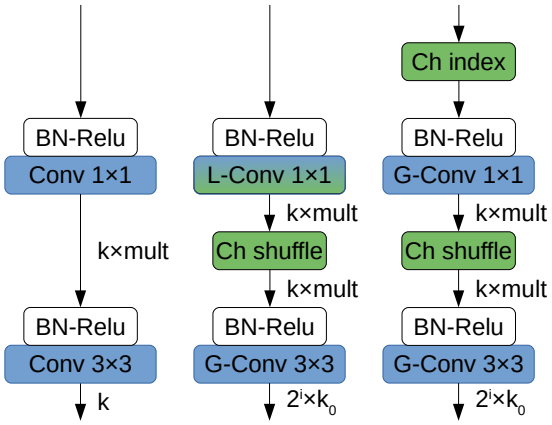



Figure 3.10: DenseNet block (left) - CondenseNet block during training (middle) - CondenseNet block after training (right), see [Huang et al., 2018]
*mult* is the bottleneck channel multiplier, usually, $mult = 4$, $k$ denotes the growth factor, $i$ the depth of the block and $k_0$ a constant

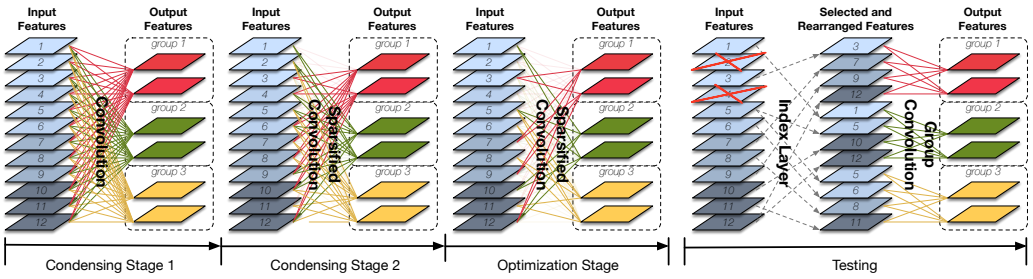



Figure 3.11: CondenseNet training procedure, figure taken from [Huang et al., 2018] Please note that several groups can take the same channels as input

A weak point of this technique is that, although it is very efficient in terms of FLOPS, it suffers the same memory allocation problem as DenseNets. Worse, the efficient approach used for vanilla DenseNets would not work since the order of the different channels is perturbed at each block.

### 3.3.4 EffNet

EffNet [Freeman et al., 2018] structure is based on the idea that bottlenecks with a huge channel-reduction factor can be harmful when applied to thin networks because it creates a bottleneck in the number of floats that flow through the network at a certain point. The authors thus designed their architecture with the idea of having a monotonically decreasing number of floats per layer.

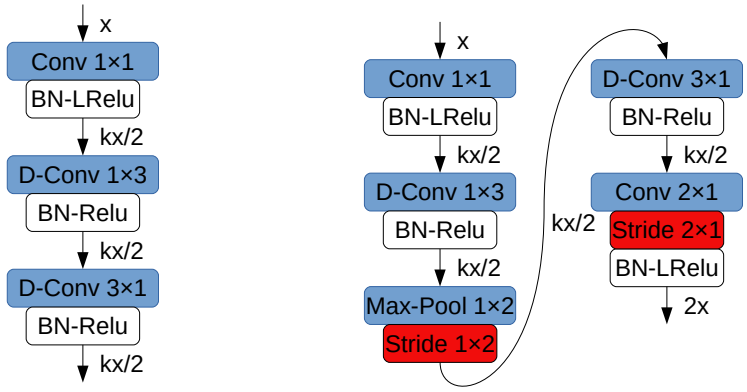
It was decided to use 1-D, depthwise separable convolutions to reduce the number of FLOPS. They also consider using a stride of two to be too aggressive, and, rather than reducing the height and width of tensors at the same layer, they do it at two different layers of the same block by using max-pooling $2 \times 1$ and a stride of $1 \times 2$ (see Figure 3.12). They also use LeakyRelu as activation for the first and last layer of the block. LeakyRelu are a variation of Relu whose equation is:

$$out = \begin{cases} \alpha \times in & \text{if } in < 0 \\ in & \text{otherwise} \end{cases} \tag{3.2}$$

where $\alpha$ is a hyperparameter whoose typical value is 0.3.

Residual connections, although very popular, were also shown by [He et al., 2016a] not to be interesting on smaller models and were thus not used here.

The aforementioned design choices were justified by experiments. A downside of the paper is, however, that the model was benchmarked in a shallow depth regime with only 3 blocks throughout the entire network. These benchmarks are thus not really of interest to us since it produces very inaccurate networks. The authors also do not provide inference time measurements on which 1-D convolutions might be less attractive than they are in terms of FLOPS due to the considerable optimization of $3 \times 3$ convolutions by libraries.




(a) Stride-less block (b) Strided block, see [Freeman et al., 2018]

Figure 3.12: EffNet blocks
For the first layer of the network, a Relu is used instead of a LeakyRelu, $k$ is a hyperparameter called expansion ratio, it is usually $> 2$.
(a) [Freeman et al., 2018] only designed the strided blocks, we inferred this block structure from the strided one. The number of input channels is $kx/2$ and not $x$ if the block was preceded by a stride-less block.

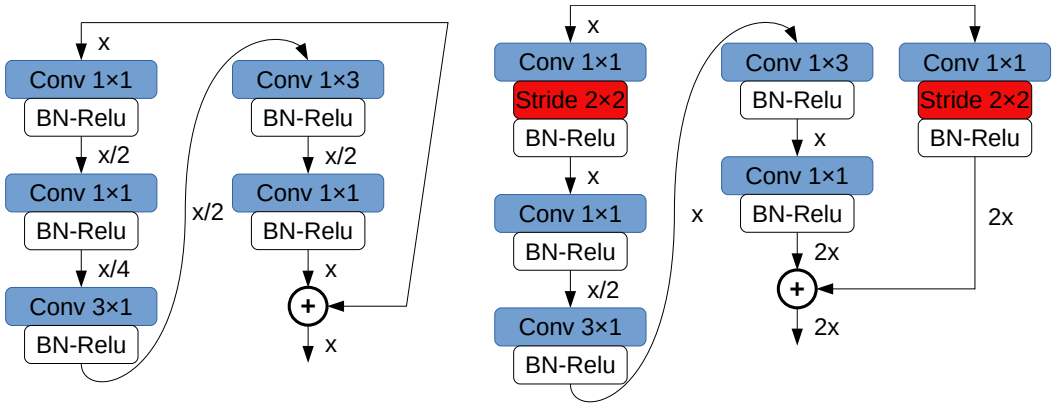
### 3.3.5 SqueezeNet and SqueezeNext

SqueezeNet [Iandola et al., 2016] was one of the early works in efficient image classification. The paper focused on reducing the number of parameters as much as possible and made much noise by claiming AlexNet's accuracy with 50× fewer parameters (510× under compression). The main idea of the paper is the use of bottleneck blocks whose second $3 \times 3$ convolution is replaced by a $1 \times 1$ and a $3 \times 3$ convolution whose outputs are concatenated. The first layer of the bottleneck was called squeeze layer, and this is where the name of the architecture comes.

More recently the SqueezeNext [Gholami et al., 2018] architecture was proposed. Its main building block is a bottlenecked block with several $1 \times 1$ conv layers, a $3 \times 1$ and a $1 \times 3$ layers, the $1 \times 1$ conv layers are used so that both input and output of the $1 \times 3$ and $3 \times 1$ layers have a reduced number of channels. A skip connection is also used (see Figure 3.13). Whether the $1 \times 3$ conv or the $3 \times 1$ conv comes first alternates throughout the network.

The authors proposed to incorporate a final bottleneck layer to the network to reduce the number of parameters in the fully connected layer.

The network design was validated by simulating its performance on a hypothetical hardware accelerator with a large number of processing elements. This allowed the authors to experiment with several variations of their architecture. First, as this is often the case, the authors proposed deeper/wider modifications of the base design, denoted *width_mult*-SqNxt-*depth*. In a second time, several new versions of the meta-architecture were proposed by varying the number of blocks per subnetwork, see table 3.1 for more details.




(a) without stride (b) with stride

Figure 3.13: SqueezeNext block, see [Gholami et al., 2018]

| subnetwork offset | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| subnetwork tensors width/height | 55 | 55 | 28 | 14 |
| subnetwork nbr of channels | 64 | 32 | 64 | 128 |
| version 1 & 2 | 6 | 6 | 8 | 1 |
| version 3 | 4 | 8 | 8 | 1 |
| version 4 | 2 | 10 | 8 | 1 |
| version 5 | 2 | 4 | 14 | 1 |

Table 3.1: Number of blocks per subnetwork in SqueezeNext architecture variations (on ImageNet's $227 \times 227$ input size)
From v2 onward, the first conv layer is of size $5 \times 5$ instead of $7 \times 7$, only main subnetworks are shown

### 3.3.6 MobileNetv1/v2

MobileNetsv1 [Howard et al., 2017] are another pioneer in efficient inference, their main idea was to use $3 \times 3$ depthwise separable convolutions followed by $1 \times 1$ pointwise convolutions. A MobileNetv1 block is shown in Figure 3.14a. To balance accuracy and latency, the authors proposed to use a width multiplier and a resolution multiplier (using a lower resolution than $224 \times 224$ for the input).

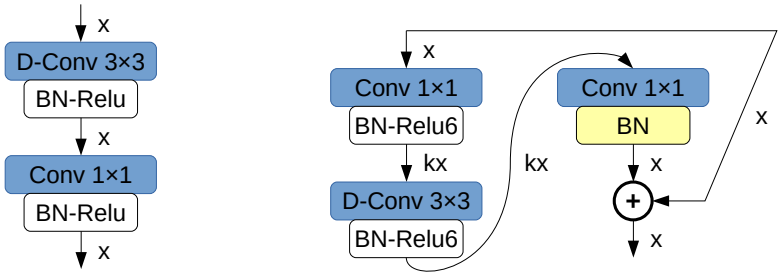
One year later, the second version of this network came out [Sandler et al., 2018]. The network is still based on depthwise separable convolutions, but the structure of the block has changed (see Figure 3.14b).

A particularity of this design is that bottleneck blocks have a bigger number of intermediate channels while their inputs/outputs have less (the authors have a much bigger discussion about the architecture, see [Sandler et al., 2018]).

The authors also used Relu6 instead of Relu because they argue it performs better with low-precision computations ([Krishnamoorthi, 2018] advocates the contrary based on experiences on MobileNetv1). Relu6 equation is:

$$out = \begin{cases} 6 & \text{if } in > 6 \\ in & \text{if } 0 \leq in \leq 6 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Finally, they did not put an activation after the last layer of each block because this layer has a huge number of input channels (see paper for more details). Mobilenetv2 also use width and resolution multipliers (although width multipliers smaller than 1 do not affect the last convolutional layer).




(a) MobileNetv1 block, see [Howard et al., 2017]

(b) MobileNetv2 block, see [Sandler et al., 2018]

Figure 3.14: Mobilenet Blocks

Stride might be used in the depthwise separable convolution in which case the skip connection is omitted for MobileNetv2, $k$ is called expansion ratio and is $> 1$

Another idea developed in the paper is a way to trade inference speed for inference maximal memory consumption. In MobileNetv2 blocks, the biggest tensors are the ones produced inside of the feature blocks. Thus by avoiding the storage of the bottlenecks intermediate tensors, we can reduce the maximal memory consumption during inference. This improvement is possible since the intermediate tensors are depthwise separable. Indeed, if we denote the output of the block by $\mathcal{F}(x)$ (not taking the skip connection into account), we have that $\mathcal{F}(x) = \mathcal{B}(\mathcal{N}(\mathcal{A}(x)))$, with $\mathcal{N} = Relu6(BatchNorm(DepthwiseConv(Relu6())))$ being a per-channel non-linear operation, $\mathcal{A} = BatchNorm(Conv())$ and $\mathcal{B} = BatchNorm(Conv())$ being linear operators, the whole operation is factorizable channel-wise, i.e. if we group the intermediate channels into $t$ groups, $\mathcal{F}(x) = \sum_{i=1}^{t} \mathcal{B}_i(\mathcal{N}(\mathcal{A}_i(x)))$. This idea reduces the maximal memory consumption since we do not have to store the whole tensors but is more computationally costly (more cache misses).

This will not be developed further in this work since we had no RAM consumption problem and our main interest was inference time. It is however mentioned as we found the idea interesting.

### 3.3.7 ShuffleNetv1/v2

ShuffleNetv1's [Zhang et al., 2018] idea is that in a classical depthwise separable block, the $1 \times 1$ pointwise convolutions accounts for the vast majority of the operations and is thus a bottleneck. To solve that problem, the authors proposed to use $1 \times 1$ grouped convolutions instead. To avoid the issue of having an output channel only depending on a subset of the input channels, they use a channel shuffle layer, like CondenseNet (Section 3.3.3). The authors show the advantage of their design through experiments. The block architecture is shown in Figure 3.15 alongside a ShuffleNetv2 block. We can see that they do not use an activation function after the depthwise separable convolution, it was shown in [Chollet, 2017] to be harmful, the latter argued it might be due to loss of information. Width multipliers are used to change the accuracy/latency tradeoff. Same goes for v2.

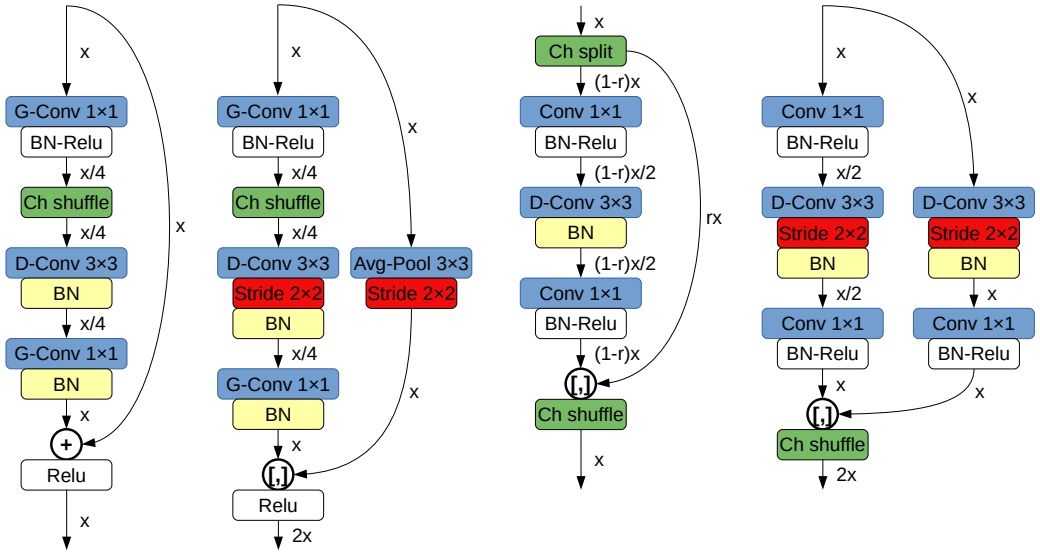



Figure 3.15: ShuffleNetv1 base block (leftmost), ShuffleNetv1 transition block (left), ShuffleNetv2 base block (right), ShuffleNetv2 transition block (rightmost); see [Zhang et al., 2018, Ma et al., 2018]

The design of the second version of these networks [Ma et al., 2018] is based on a series of design principles that are shown to hold on real hardware in the paper.

1. Equal channel width minimizes memory access costs (MAC):
   An important player in the latency is the memory access costs, for a $1 \times 1$ convolution (that accounts for most of the costs), the authors show that the MAC is minimized (under $NumChannels_{in} + NumChannels_{out}$ constant) when the number of input and output channels are equal. This principle discourages the use of bottlenecks.

2. Excessive group convolution increases MAC:
   As argued by [Gholami et al., 2018], the grouped convolutions have a large bandwidth over compute ratio, thus, by fixing the number of FLOPS, the MAC increase with the number of groups. In consequence, one should avoid using a too big number of groups.

3. Network fragmentation reduces the degree of parallelism:
   Network fragmentation is the idea to use several paths, like in the Inception architecture (Figure 3.2). Doing so is quite unoptimal for heavily parallelizable hardware like GPUs but less for CPUs.

4. Element-wise operations (Relu, addTensor, addBias) are non-negligible:
   This is because, although they have few FLOPS, they induce huge MAC (even though some of these could be optimized, for example, doing the Relu right after the convolution and not once the convolution is over).

The authors proposed thus several modifications to the ShuffleNetv1 architecture, their base building block is presented in Figure 3.15. They introduce a new operation, channel split, which consists in separating the number of channels into two groups of different sizes that go along different paths. This idea is exciting because it allows feature reuse, if $r$ is the fraction of channels reused in the next block and $x$ the number of channels, the block $i+j$ reuses $r^j \times x$ channels of block $i$, this is analogous to feature reuse in DenseNet

and its variations (Section 3.3.2). There are only 2× fewer channels in the bottlenecks than outside to follow principle 1; the $1 \times 1$ convolutions are no longer group-wise to follow principle 2; one branch remains as identity following principle 3 and the operations "concatenate", "channel shuffle" and "channel split" are done in place to reduce MAC and follow principle 4.

### 3.3.8 NASNet

NASNet [Zoph et al., 2018] architectures are found through reinforcement learning [Zoph and Le, 2016]. This consists of maintaining a probability distribution over the different architectures, to sample one of the later and to update the probability distribution based on its results. The problem with the previous approach is that it is incredibly resource consuming, thus, the authors chose to use CIFAR-10 instead of ImageNet as dataset assuming that the result would transfer and to only search for a block architecture instead of the full network architecture (the network architecture they use to combine the blocks is inspired from ResNets [He et al., 2016a] and inception [Szegedy et al., 2015]). Two different blocks must be found, one that changes the size of the tensors (transition block) and one that does not.

The search space from which layers are sampled is called *NASNet search space*. The investigated architectures are composed of 5 branches that are concatenated, each of those is the result of either an add or concatenation operation from two of the layers of the search space taking either this state or the previous one as input. The layers can be of the following types:

- Identity
- $3 \times 3$, $5 \times 5$ or $7 \times 7$ depthwise separable convolutions followed by $1 \times 1$ plain convolutions (repeated 2 times)
- $1 \times 7$ into $7 \times 1$ convolutions or $1 \times 3$ into $3 \times 1$ convolutions
- $3 \times 3$, $5 \times 5$, $7 \times 7$ max-pooling or $3 \times 3$ average pooling
- $1 \times 1$, $3 \times 3$ convolution or $3 \times 3$ dilated convolution.

See Figures 3.16 and 3.17 for the discovered architecture. It is important to note that this architecture did not take any of parameters, FLOPS or inference time into account for its search. It only cared about accuracy which is an important difference with the next approach.

The authors also developed a variation of DropPath [Larsson et al., 2016] to improve the performance of NASNets, in DropPath, each path (output of a blue operation) is dropped with a fixed probability during training. Their variation, called ScheduledDropPath, consists of linearly increasing the probability of dropping a path throughout the training.

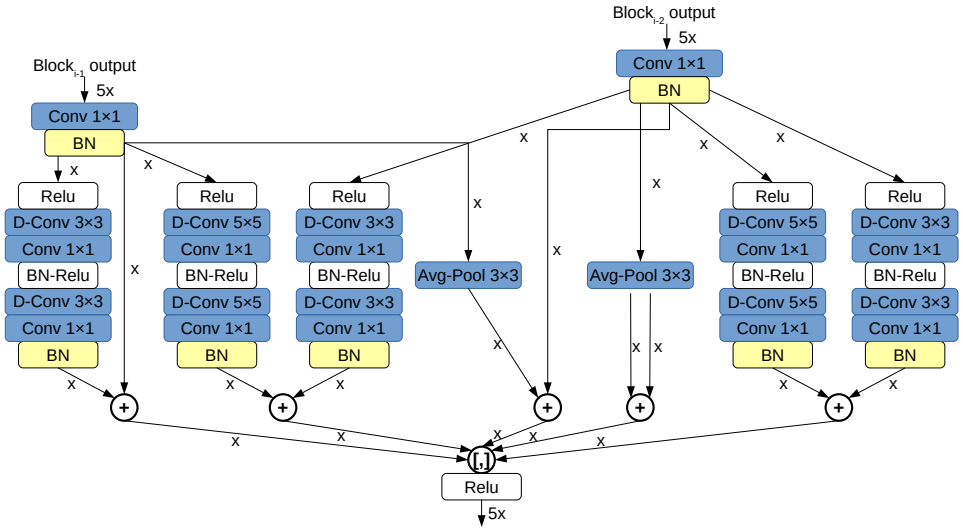



Figure 3.16: NASNet-A normal block see [Zoph et al., 2018]
In the case where the height/width of $Block_{i-2}$ output is not the same, its dimension must be reduced, this is done by using an elaborate scheme (that the paper never specifies)

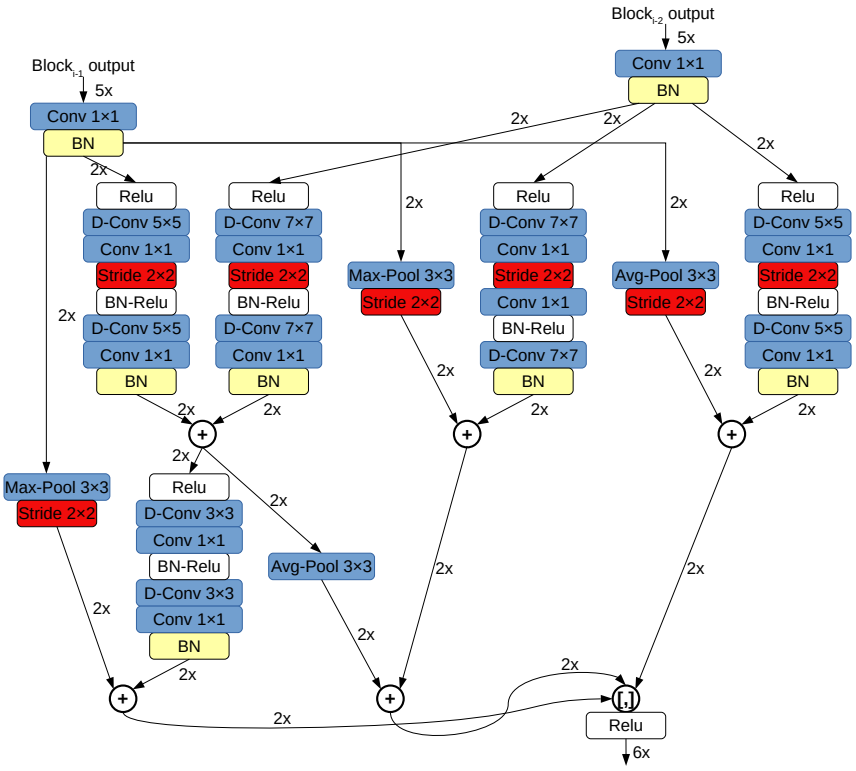



Figure 3.17: NASNet-A transition block see [Zoph et al., 2018]
Same remark as in the normal block for the height/width

### 3.3.9 MnasNet

MnasNet [Tan et al., 2018] is a reinforcement learning approach as well, important differences with NASNet are that:

- they take the inference time (on a google pixel) into account and integrate it into a

mixed objective function

- they do not use CIFAR-10 as a proxy for ImageNet (it showed bad results) but rather do not train until convergence
- they do not search for an optimal block structure but rather for the entire network

The network is divided into 7 subnetworks defined by the width and height of their tensors (each subnetwork begins with a strided layer). Each of the following parameters can vary inside of a subnetwork:

- the number of blocks
- the number of channels throughout the subnetwork
- the convolutional operation: regular convolution, depthwise convolution and inverted bottlenecks with various ratios (as in MobileNetv2, see Figure 3.14b)
- the convolutional kernel size: $3 \times 3$ or $5 \times 5$
- the skip operation used for each block of the subnetwork: max-pooling, avg-pooling, residual skip, no skip path

The final network is presented in Figure 3.18, interestingly, $5 \times 5$ convolutions are used, and we can see variations between each layer.

The authors report better results by searching with different accuracy/latency tradeoffs than scaling their architecture. Finally, experiments were performed to show that the diversity between each subnetwork is important.

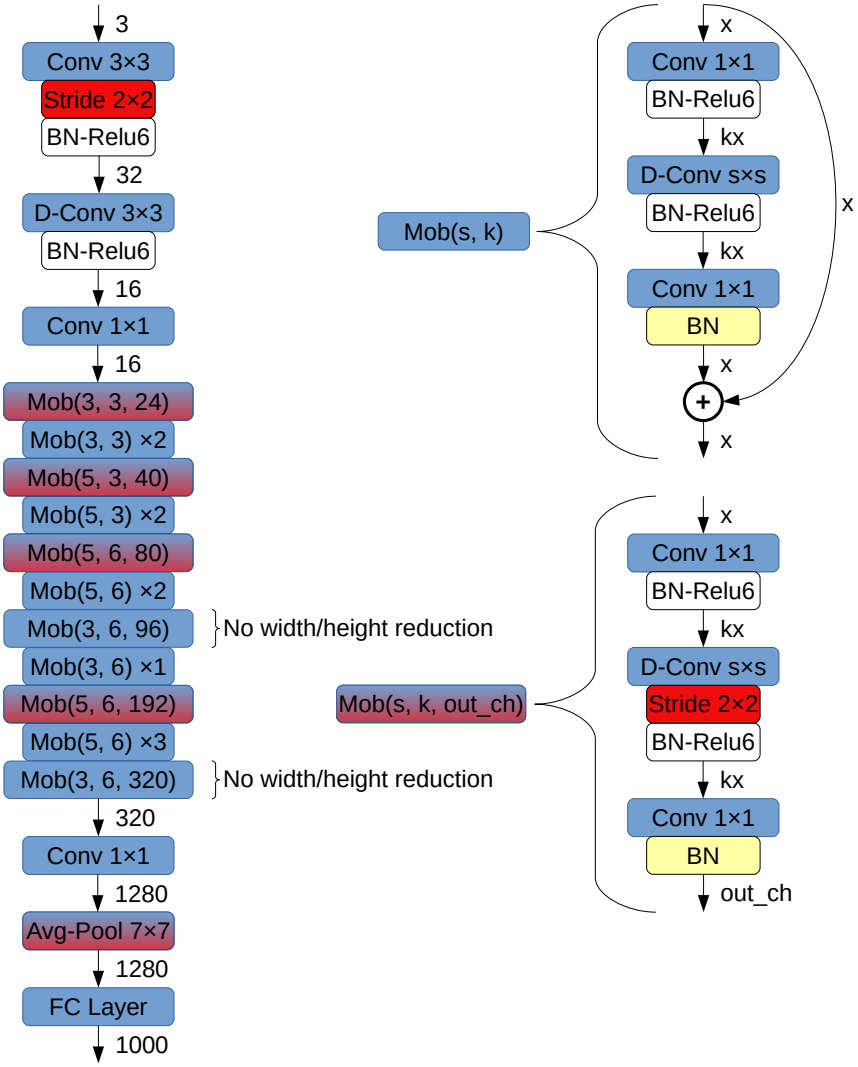



Figure 3.18: MnasNet network (on ImageNet), see [Tan et al., 2018]
Mob stands for Mobilenetv2 block

## 3.4 Comparison

### 3.4.1 Implementation details

#### Architecture

Since we are using the CIFAR-10 dataset as the benchmark, we had to reduce the size of some networks that did not provide an implementation with inputs of size $32 \times 32$. Below are listed the implementation decisions we made for the networks we had to implement (the modifications were tested on a validation set of 5,000 of the 50,000 training images from CIFAR-10), we first tested many different combinations of the width and depth as well as some network specific parameters to get networks performing well at different accuracies:

- **WideResNet** [Zagoruyko and Komodakis, 2016]: we followed the CIFAR-10 implementation of the authors. We tried different width and depths with widen factors between 1 (width = 16) and 2 (width = 32). Squeeze-and-Excitation blocks were

also added.

Since it was adapted to many other networks below, we will detail the layout (agencement of the different blocks) of WideResNets on CIFAR-10 here. First, we divide the network into 3 subnetworks of equal size, a stride of 2 is applied in the first block of the second and third subnetwork, and the number of channels is multiplied by two each time the width and height are divided by two. Before applying the subnetworks, we first use a plain $3 \times 3$ convolution with 16 output channels. After the subnetworks, the tensors go through an average pooling layer (of dimension $8 \times 8$) followed by a final fully-connected layer.

Thus, if we want to build a network containing 12 blocks (referred to as depth for all the other networks) with an initial width of 32, it would look like Figure 3.19

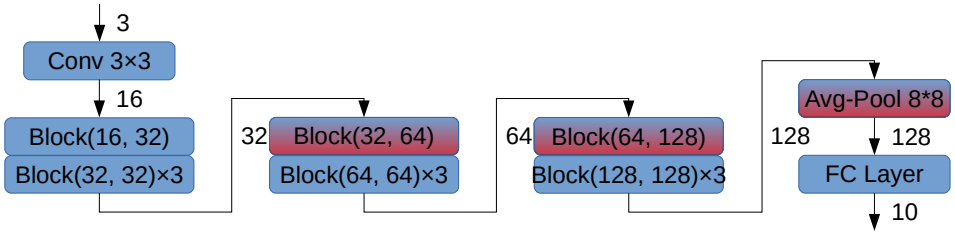



Figure 3.19: Global network architecture

- **CondenseNet** [Huang et al., 2018]: we used the official implementation[1] written in PyTorch. Weights and optimization parameters for CIFAR-10 were provided on GitHub.

  We initially planned to convert this network from PyTorch to tf-lite through ONNX (Open Neural Networks eXchange format), a library designed to exchange neural networks files between several frameworks. Converting from PyTorch to ONNX ran without problems but converting from ONNX to tf-lite (by using Tensorflow as an intermediary step) is not currently supported. ONNX also provides a runtime environment, but it does not work on 32 bits environments. We did not try to go further and that approach was abandoned.

- **EffNet** [Freeman et al., 2018]: the base network has very few layers (achieves less than 85% accuracy while we aim for 92 to 95%). We modified it ourselves to allow more layers. The small network was trained with the parameters provided in the paper. To increase the depth of the network, we just added the same number of blocks in all subnetworks.

- **SqueezeNext** [Gholami et al., 2018]: while the block structure exactly follows the paper, the layout of the blocks in networks for inputs of size $32 \times 32$ were unspecified. We thus decided to copy the design of WideResNet.

- **MobileNetv1** [Howard et al., 2017]: We followed the block structure and implemented the network using the WideResNet layout.
  Additionally, we implemented four modifications. First, we copied the kernel width

[1] https://github.com/ShichenLiu/CondenseNet

layout of MnasNet, i.e. we used $5 \times 5$ instead of $3 \times 3$ depthwise separable convolutions in the same layers as our implementation of MnasNet (see details below). Second, we added a skip connection to potentially improve the performances of deep networks. Third, we added a dropout layer (drop rate of 20%) before the final fully-connected layer. Fourth, we added Squeeze-and-Excitation blocks [Hu et al., 2018] after each convolution in which case the reduction factor was another hyperparameter to optimize.

- **MobileNetv2** [Sandler et al., 2018]: We followed the block structure and implemented the network using the WideResNet layout. We, however, made an exception for the number of channels per block that, rather than being constant on a subnetwork, grows linearly because the number of channels per block grows by small increments in the paper as well. To give an example, rather than having 32, 32 and 32 output channels for the blocks of the first subnetwork, we would have 32, 48 and 64. We also tested the network with the WideResNet layout.
  We also explored the performances for different values of the expansion ratio and tried variations that used dropout before the fully-connected layer and/or Squeeze-and-Excitation blocks.

- **ShuffleNetv1** [Zhang et al., 2018]: We followed the block structure and implemented the network using the WideResNet layout.
  We explored the performances for different number of groups in the $1 \times 1$ grouped convolution layers.

- **ShuffleNetv2** [Ma et al., 2018]: We followed the block structure and implemented the network using the WideResNet layout.
  We tried variations that used dropout before the fully-connected layer and/or Squeeze-and-Excitation blocks, as well as additionnal skip connections.

- **NASNet** [Zoph et al., 2018]: We followed the block structure from the paper and the official implementation (for what was not specified in the paper) and implemented the network using the WideResNet layout.
  This architecture was not even tested on the test set because the results it obtained on the validation set were inferior. We did not tried to use ScheduledDropPath (see Section 3.3.8) either.

- **MnasNet** [Tan et al., 2018]: Transferring the MnasNet layout to CIFAR-10 was not simple because it optimizes the network layout based on existing blocks rather than building a new block and using it throughout the network. We nevertheless noticed a strong resemblance with MobileNetv2. Thus we decided to use our CIFAR-10 version of MobileNetv2 with two modifications. First, we replaced the $3 \times 3$ depthwise separable convolutions by $5 \times 5$ ones at all the blocks but the last one of the 2nd and 3rd subnetworks. Second, the expansion factor is divided by two in the first block of the first subnetwork.
  We also tried variations that used dropout before the fully-connected layer and/or Squeeze-and-Excitation blocks.

### Optimizers

We also had to choose the optimizer to use, the different choices we tested are summarized in Table 3.2. Surprisingly, the relative performances of the optimizers were nearly independent of the choice of architecture. We can also see that weight decay is of prime

importance here. Weight decay is an additional penalty term added to loss to reduce overfitting. It penalizes large weights in convolutions filters/batchnorm scaling factors/fully connected filters. Since we deal with lightweight networks with few parameters here, it is useful to use little weight decay ($5 \times 10^{-4}$ is a typical value).

| optimizer name | init lr | lr decay policy | weight decay | source | relative accuracy |
|---|---|---|---|---|---|
| SGD with cosine lr | .1 | lr shrinks like a cos fct | $10^{-4}$ | [Huang et al., 2018] | 0 |
| multistep SGD | .1 | $lr \times = 0.2$ at epoch (60,120,160) | $5 \times 10^{-4}$ | [Crowley et al., 2018] | $-3\%$ |
| multistep SGD | .1 | $lr \times = 0.2$ at epoch (60,120,160) | $10^{-4}$ | - | a bit less |
| Adam | $10^{-3}$ | - | $5 \times 10^{-4}$ | [Freeman et al., 2018] | $-8\%$ |
| Adam | $10^{-3}$ | - | $10^{-4}$ | - | fails to converge |
| RMSprop | $4.5 \times 10^{-2}$ | $lr \times = 0.99$ at each epoch | $4 \times 10^{-5}$ | [Sandler et al., 2018] | fails to converge |

Table 3.2: Relative Performance with respect to *SGD with cosine lr* and characteristics of the different optimizers
The performance is given in % of accuracy relative to the best (cosine learning rate), we always trained for 200 epochs.
Adam is trained with $\beta_1 = .75$, multistep SGD with a heavy-ball momentum of 0.9 and SGD with cosine lr with a Nesterov momentum of 0.9

At the light of these results, we decided to use cosine learning rate [Loshchilov and Hutter, 2017] by using the hyperparameters from [Huang et al., 2018].

### 3.4.2 Results

Now that we have defined our training settings, we still have to find good combinations of the different hyperparameters. This is done in section 3.4.2. Next, we have to compare the different architectures in terms of error as a function of the inference times on our dataset and hardware. This is done in Section 3.4.2. Finally, we draw a short conclusion about the results of this chapter.

#### Hyperparameter search

The final hyperparameters chosen for the networks used in Section 3.4.2 are shown in Table 3.3. These hyperparameter settings are the result of 114 trainings on the validation set. Since we want to cover an error range, networks of different size were trained, we limited ourselves to 3 networks per architectures maximum.

Since CIFAR-10 is a relatively easy benchmark, we targeted low error rates between 8 and 5% ideally. We chose low error rates because networks that have bad accuracy, although being extremely fast, are not precise enough to be used in practice.

An important thing to notice regarding the error achieved by the different networks is that, at some point, increasing the depth/width did not improve the accuracy on the validation set (or at the price of significant inference time increase). Thus, some architectures only covers a small error range and do not have 3 different sets of hyperparameters.

Some remarks concerning the hyperparameters we chose:

- **ShuffleNetv1**: A number of groups of 2 (2 groups in the $1 \times 1$ grouped convolutions) achieved better results than one of 3. Using only one group would have been interesting, but not respect the design (and grouped convolutions were removed in ShuffleNetv2).

| architecture | inference time (ms) | number of blocks | width | others |
|---|---|---|---|---|
| WideResNet | 48.9 | 9 | 16 | - |
| WideResNet | 100.8 | 9 | 24 | - |
| WideResNet | 235.4 | 12 | 32 | - |
| SqueezeNext | 52.7 | 9 | 48 | - |
| EffNet | 21.5 | 12 | 32 | exp. ratio of 4 |
| ShuffleNetv1 | 50.7 | 9 | 80 | 2 groups (in grouped convs) |
| ShuffleNetv2 | 44.1 | 12 | 64 | - |
| ShuffleNetv2 | 93 | 18 | 80 | - |
| MobileNetv1 | 22.6 | 15 | 32 | - |
| MobileNetv1 | 52.8 | 12 | 64 | - |
| MobileNetv1 | 150.1 | 15 | 100 | - |
| MobileNetv2 | 27.2 | 9 | 16 | exp. ratio of 4, dropout |
| MobileNetv2 | 50.3 | 9 | 24 | exp. ratio of 4, dropout |
| MobileNetv2 | 107.7 | 12 | 24 | exp. ratio of 6, dropout |
| MnasNet | 30.7 | 9 | 16 | exp. ratio of 4, dropout |
| MnasNet | 64.4 | 12 | 16 | exp. ratio of 6, dropout |
| MnasNet | 121.2 | 12 | 24 | exp. ratio of 6, dropout |

Table 3.3: Chosen hyperparameters
The width reported here represents the width outside of the blocks in the first subnetwork, for e.g. a width of 32 means 32, 64 and 128 for the first, second and third subnetworks respectively. When using a Squeeze-and-Excitation block, the reduction factor was always set to 2

- **ShuffleNetv2**: Using dropout on the final layer did not have any significant influence on the results and was thus not used. Skip connections proved to be harmful and were thus not used.
- **MobileNetv1**: The variations to include $5 \times 5$ convolutions and skip connections were significantly worse, we thus stuck with the original architecture. The dropout did not have any significant influence on the results and was thus not used.
- **MobileNetv2**: The structural variation of increasing the number of channels inside of the subnetwork proved to be harmful, it was probably too aggressive, and a lighter block size increase could have given better results but was not tested. We thus stuck with WideResNet's layout. Dropout also helped and was thus used.
- **MnasNet**: Like for MobileNetv2, dropout improved results and was used. Different architectural adaptations (expansion ratio divided by 2 for the first subnetwork and not only the first layer) would be interesting to test but were not due to lack of time.
- **Squeeze-and-Excitation blocks:** reduction factors of 1, 2 and 4 were tested for all architectures but 2 always performed better.

### Architecture comparisons

We can see in Figure 3.20 that ShuffleNetv1, EffNet and SqueezeNext perform very poorly, both in term of accuracy and in terms of lowest possible error. If we go back to the architecture descriptions, we can see that EffNet and SqueezeNext are the only networks using 1-D convolutions (replacing a $3 \times 3$ convolution by a $3 \times 1$ and a $1 \times 3$) suggesting these might work poorly in our desired error-range.

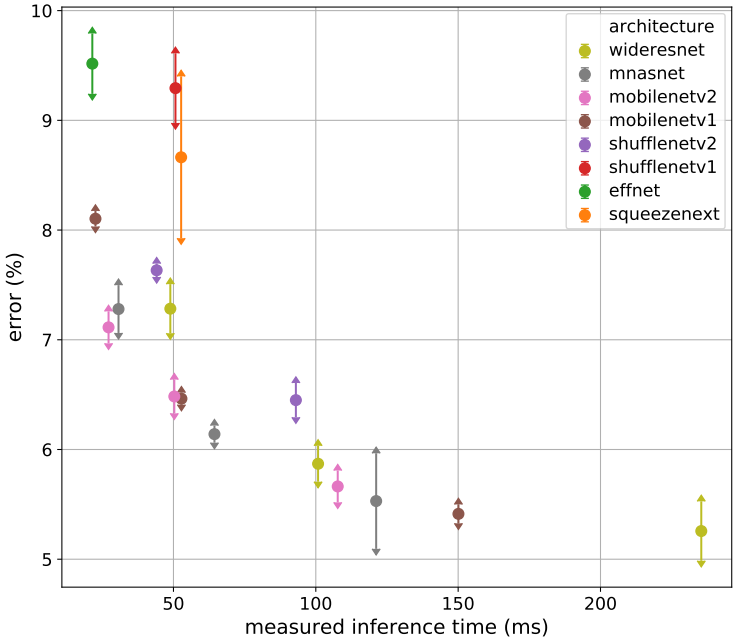



Figure 3.20: Error vs inference time scatterplot for different network architectures without Squeeze-and-Excitation blocks
The error point are the means over 3 trials. The errors bars show the standard derivation computed over the 3 trials.

ShuffleNetv1 bad performances are harder to explain, they might come from the design choices that reduce the complexity more aggressively than ShuffleNetv2 (grouped $1 \times 1$ convolutions and a smaller number of channels in bottlenecks). We can also note that grouped convolutions are not first-class citizens in Keras, which probably causes them to be slower. Indeed, since they do not exist explicitly in the API, we had to implement them using *num_groups* full convolutions, each using a fraction of the input tensor. The outputs of these convolutions were later concatenated together. This probably results in *num_groups* useless tensor allocations (the outputs of the full convolutions) that are afterwards merged into a bigger tensor that must be allocated. Instead, directly allocating the big tensor and writing the outputs of each group in place would be faster.

ShuffleNetv2 performs better than the 3 previous models both in terms of performances and achievable error. It is, however, less attractive than WideResNet, MobileNetv1/v2 and MnasNet. Although its authors ([Ma et al., 2018]) report better results than MobileNetv1/v2, the tests they performed were based on a custom implementation that was probably much more efficient in terms of management of memory (for example for the tensors concatenations) than Tensorflow Lite. This could explain the performance gap observed here.

WideResNets berforms a bit better than ShuffleNetv2 (and can achieve low errors). It is surprising than such a simple baseline using plain convolutions performs so well.

Finally, we can see that our best results are obtained by MobileNetv1, MobileNetv2 and MnasNet (which is heavily inspired by MobileNetv2). It is quite surprising that MobileNetv1 performs as well as MobileNetv2 since the latter is the next version of the former.

### Squeeze-and-Excitation blocks

An improvement that can be added to all architectures are the Squeeze-and-Excitation blocks (see Section 3.2.7 for a summary). We decided only to try those on ShuffleNetv2, WideResNet, MobileNetv1/v2 and MnasNet. These are displayed in Figure 3.21. We can still see the corresponding networks without SE blocks under a smaller alpha.

We can see that applying this block nearly always results in accuracy improvement, while the increase in inference time is minimal for MobileNetv2 and MnasNet. We can also notice that adding Squeeze-and-Excitation blocks to WideResNets is not an interesting modification. There, we barely see error improvements and this even sometimes increases the error (not much when taking the confidence intervals into account). This might be due to the fact that SE blocks are much more interesting when using depthwise separable convolutions but we lack experiments to suggest that rigorously.

SE blocks also have the benefit to push the best achievable loss boundary further.

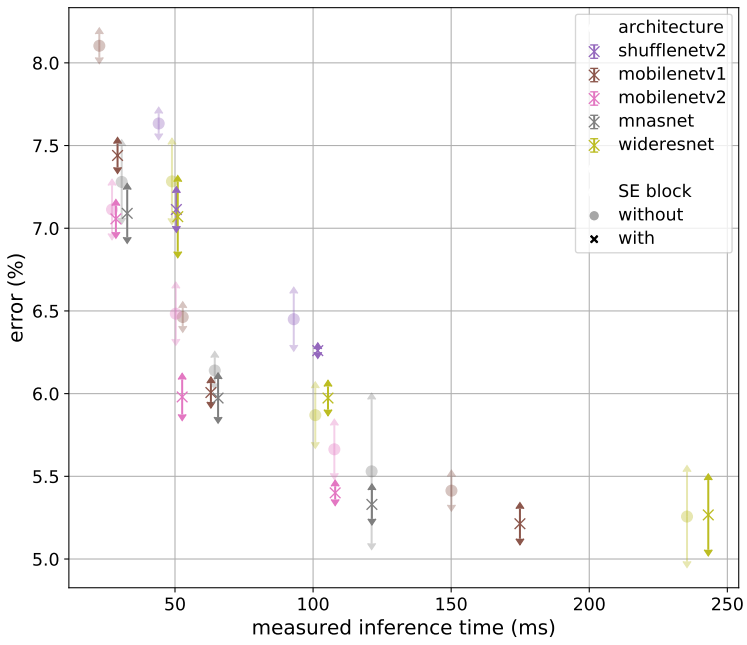



Figure 3.21: Error vs inference time scatterplot for different network architectures with Squeeze-and-Excitation blocks
The error point/crosses are the means over 3 trials. The errors bars show the standard derivation computed over the 3 trials. The same architectures without SE blocks are displayed as well for comparison

The increase in inference time and, to a lesser extent, in accuracy is more critical for MobileNetv1 and ShuffleNetv2. This is because we add a SE block at the end of each

network block. Since MobileNetv1 blocks are simpler, MobileNetv1 are, for the same accuracy, deeper and wider. As a consequence, there are more SE blocks, each bigger than for WideResNet, MobileNetv2 And MnasNet. This result in a significant increase in inference time.

When adding SE blocks to the picture, MobileNetv2 seems to perform better than the others at a given error rate.

### 3.4.3 Conclusion

In this chapter, we introduced several architectures that we compared afterwards on our benchmark. We also modified some of them by adding Squeeze-and-Excitation blocks. The unmodified architectures can be split in three groups:

1. ShuffleNetv1, EffNet and SqueezeNext that performed poorly and could not achieve low errors
2. ShuffleNetv2 and WideResNet that showed average results. ShuffleNetv2 was also more limited than WideResNet in terms of lowest achievable error
3. MobileNetv1, MobileNetv2 and MnasNet that showed good performances and could achieve small errors

Adding Squeeze-and-Excitation blocks to the networks of the last group have two benefits. First, it improves the accuracy of the networks (at a given error rate), even though this is not always noticeable. Second, it allows the netwoks to achieve lower errors while retaining a low inference time.

We can conclude that MobileNets blocks are particularily interesting, especially when adding SE blocks. The best performing network over all the architectures tested are MobileNetv2 with SE blocks, while MobileNetv1 and MnasNet (with SE blocks) fall shortly behind.

In the next chapter, we will study a technique based on pruning algorithms that allows to chose the number of channels in a more principled way than what we did until now. In this chapter, we fixed the number of channels to be multiplied by two each time stride is applied. Next, we will use a trained network and its error/loss to decide at which layers to put more or less channels or even to decide in which subnetworks to put more or less layers.

# Chapter 4

# Channel Pruning

## 4.1 Introduction

Another popular method to reduce the resource consumption of neural networks is pruning, i.e. removing elements of the networks in a way that has little impact on the accuracy. Pruning can be performed at several levels. The most commons are weight and channel. Weight pruning has received much attention (see [Frankle and Carbin, 2019, Lee et al., 2018] for recent works). It performs better in terms of parameter reduction since it does not constrain the choice of the parameters to prune. Unfortunately, sparse convolutions are not efficiently leveraged by modern hardware [Turner et al., 2018], and weight-level pruning will thus not be investigated in this work. Channel pruning [Molchanov et al., 2017, Theis et al., 2018], on the other hand, produces networks that are as easily leveraged as their unpruned counterparts since we only played on the number of channels of each layer.

The main idea behind this chapter is using pruning as an architecture search procedure. This means that we are not interested in the weight obtained through pruning but only by the architectures (number of channels per layer) discovered.

This is faster than finding the best number of channels for each layer through reinforcement learning as well as potentially much more interesting than using hand-coded, default, values. First, as pruning uses pieces of information from the dataset, we can adapt our network for the given data. For example, a dataset depending on finer features (edges are important for classification) would spare channels in the early layers while being more aggressive towards later layers; On the other hand, a dataset depending on higher-level features would spare channels in the last layers of the network. The same reasoning applies for differences in hardware; different devices might have different relative costs for the channels of each layer. Finally, state-of-the-art networks all have a very regular number of channels (e.g. 32 for $x$ layers, then 64 for $x$ layers, then 128 for $x$ layers) but nothing says that the optimal is not something fancier, like $18, 24, 24, 18, 32, 48, 64, 48, \ldots$.

The different criteria used for channel pruning are explained in Section 4.2. Section 4.3 explains why pruning should be used as an architecture search procedure rather than a post-processing operation. Sections 4.4, 4.6 and 4.7 present different approaches we tested and adapted. These were only tested on a WideResNet [Zagoruyko and Komodakis, 2016] architecture with a depth of 40 and a widen factor of 2. That architecture is interesting to prune since it uses skip connections. We chose only one due to the important training times needed to prune an architecture. Finally, Section 4.8 compares the latter approaches.

## 4.2 Pruning Mechanisms

There are two main types of pruning mechanisms. The first is to prune channels (or weights) according to a given criteria associated with them. These criteria are obtained from fully-trained networks and the channels with the smallest scores are pruned iteratively, either one at a time or several at once. This approach is taken by all the mechanisms detailed here but the last one (Section 4.2.5).

Another possibility is to add a penalization term to the loss to enforce some sparsity in the network. This is the approach taken by MorphNet [Gordon et al., 2018] (Section 4.7). In the following, we only refer to pruning channels, but the reasoning is the same for pruning weights.

### 4.2.1 Weight norms

The simplest criterion to prune convolution layer's channels is the $L_1$ or $L_2$ norm of the weights that compose this channel. This metric has the advantage to be very straightforward to compute. Assuming the network is trained with weight decay, i.e. there exists a penalty for high weights in the objective function, the norm of the weights can be seen as a metric related with the variance of the weights inside the channel. Since $Var\{\mathcal{X}\} = E\{\mathcal{X}^2\} - E\{\mathcal{X}\}^2$, the square of the $L_2$ norm and the variance are equal if not for $E\{\mathcal{X}\}^2$ which is assumed to be small because of the weight decay. Channels with high variance convey more information than others. So, keeping them makes sense intuitively in the absence of more formal arguments. This, however, does not take the rest of the network into account. Indeed, the output of a channel whose weights have a high variance might not be used so much by the rest of the network. This technique does not performs as well as others because of its simplicity [Hassibi and Stork, 1993, Molchanov et al., 2017, Theis et al., 2018].

### 4.2.2 Activation based metrics

This technique looks a lot like weight-norm based pruning. The difference is that rather than using norms of the weights of the channels, they use the norm of the tensors after the activations. This makes sense since, after a Relu activation function, for example, half of the elements of the tensors are zeroed-out. This approach works better than weight norm [Molchanov et al., 2017, Theis et al., 2018] but is still not so well developed theoretically and does not take the rest of the network into account.

### 4.2.3 Taylor-based approaches

There exist more sophisticated and theoretically motivated approaches. These are based on approximations of the Taylor expansion of the loss when removing a channel from the network. Both [Molchanov et al., 2017, Theis et al., 2018] use such approaches and end up with nearly the same signal ([Molchanov et al., 2017] takes the absolute value of something as criterion instead of its square and uses normalization). Since [Theis et al., 2018]'s approach, called Fisher pruning, is better motivated, we do not detail [Molchanov et al., 2017] here nor use it in the rest of this work.

Fisher pruning (the following is just a rephrasing of the proof of [Theis et al., 2018])

considers the case of a network trained to minimize a cross-entropy loss, denoted:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_P[-log\mathcal{Q}_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{I})] \tag{4.1}$$

where $\boldsymbol{\theta}$ are the parameters of the model, $P$ is a data distribution, $\boldsymbol{z}$ are the outputs of the model and $\boldsymbol{I}$ its inputs.
The effect of a change of parameters $\boldsymbol{d}$ on the loss can be approximated with a 2nd order Taylor expansion, where $\boldsymbol{g} = \nabla\mathcal{L}(\boldsymbol{\theta})$ and $\boldsymbol{H} = \nabla^2\mathcal{L}(\boldsymbol{\theta})$:

$$\mathcal{L}(\boldsymbol{\theta} + \boldsymbol{d}) - \mathcal{L}(\boldsymbol{\theta}) \approx \boldsymbol{g}^T\boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T\boldsymbol{H}\boldsymbol{d} \tag{4.2}$$

And if $\boldsymbol{d} = -\theta_k\boldsymbol{e}_k$, where $\boldsymbol{e}_k$ is a unit vector:

$$\mathcal{L}(\boldsymbol{\theta} - \theta_k\boldsymbol{e}_k) - \mathcal{L}(\boldsymbol{\theta}) \approx -g_k\theta_k + \frac{1}{2}H_{kk}\theta_k^2 \tag{4.3}$$

The network is assumed to be at a local optimum and the first term thus vanishes. The diagonal of the Hessian can be approximated assuming that $Q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{I})$ is close to $P(\boldsymbol{z}|\boldsymbol{I})$ (network trained to convergence):

$$H_{kk} \approx \mathbb{E}_P[(\frac{\partial}{\partial\theta_k}log\mathcal{Q}_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{I}))^2] \tag{4.4}$$

See annex of [Theis et al., 2018] for more details. The second term can be seen as an empirical estimate of the Fisher information of $\theta_k$. Hence the name "Fisher pruning".

If we use $N$ data points (images), the increase in loss finally becomes

$$\Delta_k = \frac{1}{2N}\theta_k^2\sum_{n=1}^{N} g_{nk}^2 \tag{4.5}$$

where $\boldsymbol{g}_n$ is the gradient of the parameters with respect to the $n$th data point.

To prune an entire channel at a time, we can assume the presence of a binary mask $m \in \{0,1\}^K$ of size $K$ (with one component for each channel we desire to prune in the network) and denote by $a_{nkij}$ the activation of the $n$th data point corresponding to channel $k$ at spatial indexes $i, j$.

We thus have: $a'_{nkij} = m_k a_{nkij}$. Computing the gradients, $\boldsymbol{g}_n$, of equation 4.5 with respect to the binary mask $m_k$ gives:

$$\begin{aligned}
g_{nk} &= \frac{\partial}{\partial m_k} - log\mathcal{Q}(\boldsymbol{z_n}|\boldsymbol{I_n}) && (4.6)\\
&= -\sum_{ij}\frac{\partial}{\partial a'_{nkij}}(log\mathcal{Q}(\boldsymbol{z_n}|\boldsymbol{I_n}))\frac{\partial a'_{nkij}}{\partial m_k} \qquad \text{chain rule} && (4.7)\\
&= -\sum_{ij} a_{nkij}\frac{\partial}{\partial a_{nkij}}log\mathcal{Q}(\boldsymbol{z_n}|\boldsymbol{I_n}) \qquad \text{for channels not already pruned} && (4.8)\\
&= -\sum_{ij} a_{nkij}\frac{\partial}{\partial a_{nkij}}log\mathcal{Q}(\boldsymbol{z}_n|\boldsymbol{I}_n) && (4.9)
\end{aligned}$$

The pruning signal thus being

$$\Delta_k = \frac{1}{2N}\sum_n g_{nk}^2 = \frac{1}{2N}\sum_n(\sum_{ij} a_{nkij}\frac{\partial}{\partial a_{nkij}}log\mathcal{Q}(\boldsymbol{z}_n|\boldsymbol{I}_n))^2 \tag{4.10}$$

since $\theta_k^2 = 1$ before pruning because $\theta_k$ is a boolean parameter of one of the masks. $\frac{\partial}{\partial a_{nkij}} log \mathcal{Q}(\boldsymbol{z}_n|\boldsymbol{I}_n)$ is computed during the backward pass and is thus "free" to compute which makes this criterion nearly as efficient as simpler ones like weight norm.

The variation in loss $\mathcal{L}(\boldsymbol{\theta} + \boldsymbol{d}) - \mathcal{L}(\boldsymbol{\theta})$ could also be computed explicitly. While being more accurate, this approach would have a significant computational cost. Indeed, the number of channels in a layer could be as high as 1000. So instead of one backward pass (for $n$ inputs), we would have to compute 1000 forward passes (for $n$ inputs).

This approach links the pruning of a channel with an accuracy cost. However, it is agnostic of the benefits we get in terms of performance for removing that channel. The authors thus proposed to use the ratio $\frac{\Delta \mathcal{L}}{\Delta \mathcal{C}}$, where $\mathcal{C}$ is the cost of the network in terms of FLOPS as a criterion instead. [Yang et al., 2018] proposed to use real-time measurements instead of FLOPS to weight the predicted decrease in accuracy in a different context.

### 4.2.4 Importance-based metrics

Another approach to prune channels that takes the whole network into account is to prune the layers based on some importance metric related to the predictions of the network. Taylor-based approaches can be seen as a particular case of this. The only example we consider here is [Yu et al., 2018], where the importance is computed from the influence of the channel to the second to last layer of the network. In the latter, the authors first applied feature ranking on the inputs of the last layer. Once this is done, a fixed fraction of channels is pruned at each layer to minimize the reconstruction error, this is formulated as a binary integer optimization problem, and an upper bound of the corresponding objective function is minimized. This results in having to minimize an importance score, which is the product of the weights matrices of the following layers over the network. It is computable in a single backward pass.

This algorithm was not inspected in the rest of this thesis because of a lack of time and because we found out about it too late. It is nevertheless mentioned here for completeness and would be interesting to test in a future work since it showed interesting results.

### 4.2.5 Batchnorm-based metrics

This approach is the only mechanism described in this work used as a penalization term in the loss.
All of the recent neural network architectures make use of batch normalization [Ioffe and Szegedy, 2015]. Since the batch normalization principle is to rescale each element of a tensor separately, we can use the scaler associated with all of the outputs of a channel to get an idea of the importance of that channel. These scaling factors are penalized in the objective function so that the output of some channels are pushed towards 0 which is equivalent to pruning them. Network-Slimming [Liu et al., 2017] and MorphNet [Gordon et al., 2018] use that idea.

## 4.3 Pruning as an architecture search

Usually, pruning is considered as a three staged pipeline:

1. training an overparameterized network from scratch

2. pruning some channels based on a given criterion (with a bit of fine-tuning between the consecutive prunings)

3. fine-tuning the pruned network obtained previously

However, both [Liu et al., 2018] and [Crowley et al., 2018] recently showed that the interest in pruning a network does only come the structure (number of channels for each layer) discovered through the pruning and not from the inherited weights. They showed that fine-tuning the pruned network obtained at step 2 gave worse results than simply take the structure and retrain the new network from scratch (from random weights). Both pipelines are illustrated in Algorithms 1 and 2 respectively (only the last lines differs).

**Algorithm 1:** Illustration of the basic pruning pipeline with an algorithm pruning a channel at a time

**Input:** the plain, pretrained network: *network*, the number of channels to prune: *num_channels_to_prune*
**Output:** the pruned network

```
pruned_net = network
for i = 0 to num_channels_to_prune - 1 do
    pruned_net = prune_one_channel(pruned_net)
    pruned_net = retrain_on_few_batches(pruned_net)
end
pruned_net = retrain_on_many_batches(pruned_net) /* optionnal but quite
    common                                                          */
return pruned_net
```

**Algorithm 2:** Illustration of the modified pruning pipeline that discards the pruning weights with an algorithm pruning a channel at a time

**Input:** the plain, pretrained network: *network*, the number of channels to prune: *num_channels_to_prune*
**Output:** the pruned network, already retrained from scratch

```
pruned_net = network
for i = 0 to num_channels_to_prune - 1 do
    pruned_net = prune_one_channel(pruned_net)
    pruned_net = retrain_on_few_batches(pruned_net)
end
/* we reinitialize the weights randomy in retrain_from_scratch      */
pruned_net = retrain_from_scratch(pruned_net)
return pruned_net
```

Both works compared the two pipelines and observed that the second yields better results. This means that the networks obtained from the basic pipeline have weights that achieves a local optimum due to the greedy training strategy that is worse than the local optimum reached by training the pruned network from scratch.

[Liu et al., 2018] makes comparisons on VGG [Simonyan and Zisserman, 2014], ResNet [He et al., 2016a], DenseNet [Huang et al., 2017] and PreResNet [He et al., 2016b] using 6 pruning channel or weight pruning algorithms including [Liu et al., 2017]. [Crowley

et al., 2018] uses Fisher pruning and pruning based on the $L_1$ norm of the weights on WideResNet [Zagoruyko and Komodakis, 2016] and DenseNet [Huang et al., 2017]. Both made experiments on ImageNet [Russakovsky et al., 2015] and CIFAR-10 [Krizhevsky, 2009] (CIFAR-100 was also used by [Liu et al., 2018]). Thus, we can see that their conclusion is well motivated.

The main difference between both papers is that [Liu et al., 2018] seems to apply a fixed learning rate for retraining from scratch and chose the number of epochs to have the same computation time than the training of the unpruned network (i.e. a 2 times smaller model would have 2 times more epochs). Whereas [Crowley et al., 2018] used an adaptive learning rate schedule and the same number of epochs for training the unpruned and pruned networks from scratch. We chose the same approach as [Crowley et al., 2018] since we see no reason to treat the (re)training of the pruned and unpruned networks in a different way.

Batchnorm-based approaches do not work exactly in the same way (the three steps are merged into a single step with constraints), but the same reasoning holds: we can still retrain the pruned architectures from scratch afterwards.

The following experiments were based on [Crowley et al., 2018]'s work and, like them, we only pruned the layers inside of the bottlenecks (see Figure 4.1) and do not take the cost (number of FLOPS) of a channel into account when pruning. Limiting the pruning to the bottlenecked layers has the advantage to avoid having to come up with a solution to prune several layers at once since both layers whose output are added (for example after a skip connection) must have the same number of channels. However, when pruning the network heavily, we end up with a disparity between the number of channels inside and outside the bottlenecks, which is very likely not optimal. The effect of both of these choices is further investigated in Section 4.4.2. The paper results are shown as a baseline to compare with other approaches and to see how the networks are pruned.

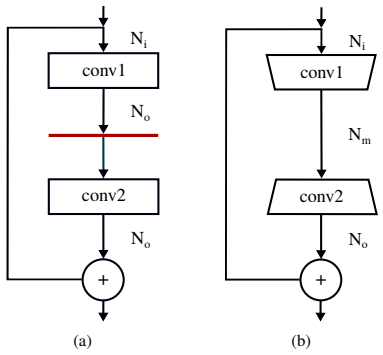
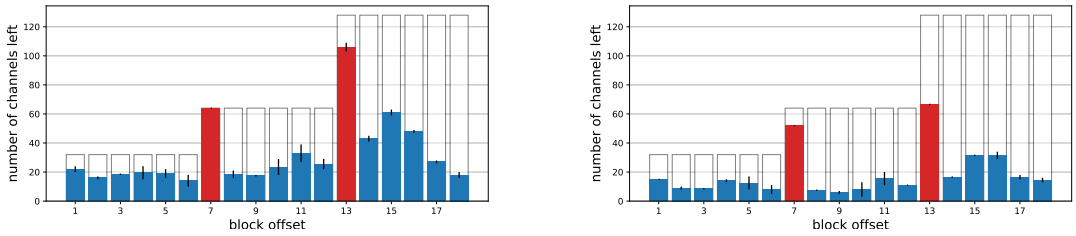



Figure 4.1: Pruning limited to bottlenecks, figure taken from [Crowley et al., 2018] (a) represents one of the base blocks that compose the network; only the channels on the red line will be diminished through pruning to be reduced to $N_m$ finally (b). Thus, after heavy pruning, $N_m$ could be around 10 while $N_i = N_o$ could be around 100 (since the corresponding tensors are added at the end of the block, $N_i$ and $N_o$ must be equal)

Since we consider pruning as an architecture search, the result of interest of the pruning algorithm is the number of channels that are kept for each layer. We display the results of pruning 750 and 1000 layers (out of 1344) of a WideResNet [Zagoruyko and Komodakis, 2016] with a depth of 40 and a width-multiplier of 2 in Figure 4.2. As we can see on the latter, the channels where stride is present are very lightly pruned, and the first channels of the network are proportionally less pruned than the others.




(a) 750 channels pruned

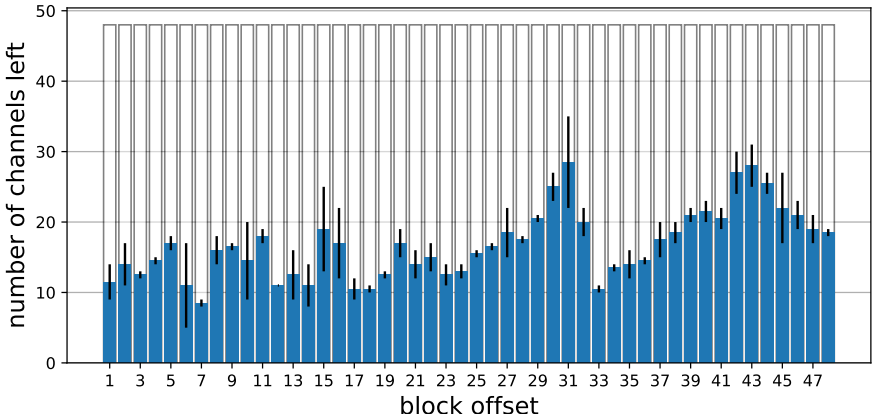
(b) 1000 channels pruned

Figure 4.2: Number of channels per block (since we only prune inside of blocks) of a WideResNet-40-2 after Fisher pruning
The error bars represent the min and max over two trials. The red colour is used to show tensors spatial resolution reduction. The black borders show the initial number of channels

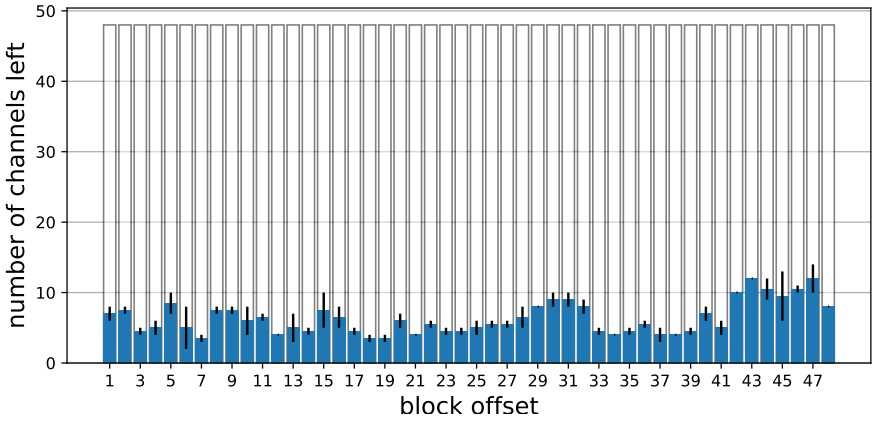
We then do the same for pruning 1500 and 2000 channels (out of 2304) of a DenseNet [Huang et al., 2017] with a depth of 100, a growth rate of 12 and a reduction-factor of 0.5 (see Figure 4.3). Here, since we do not prune the strided layer directly, we do not observe the same striding reduction, although there is a peak around block 32 after some depression in the 10 previous blocks. We observe the same kind of phenomenon around block 43.

We can also observe the error curves during pruning (see Figure 4.4). We can see that Fisher pruning performs better than random pruning or pruning based on the $L_1$ norm of the weights and that we get much better results by retraining the network from scratch in each case. Please note that random pruning is very close to uniform pruning (dividing the number of channels per layers by a constant) since each channel has the same probability of being removed.

Now that we have explained why we want to retrain architectures obtained through pruning from scratch, we will introduce modifications to several pruning algorithms as well as a method to compute the inference times of all the pruned variations of a network efficiently.




(a) 1500 channels pruned




(b) 2000 channels pruned

Figure 4.3: Number of channels per block (since we only prune inside of blocks) of a DenseNet of depth 100, growth-rate 12 and reduction-factor 0.5 after Fisher pruning
There is no red here because the layers that reduce tensor spatial resolution are not pruned, strides occur after blocks 16 and 32

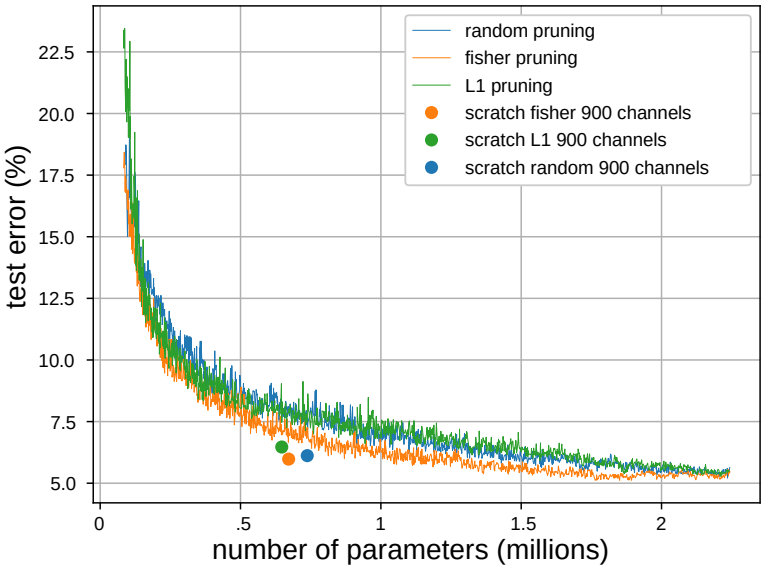



Figure 4.4: Evolution of the test error when {fisher, $L_1$, random} pruning a WideResNet-40-2
The points correspond to architectures obtained through pruning that were retrained from scratch

## 4.4 Fisher pruning modifications

We now investigate two modifications of the Fisher pruning algorithm.

### 4.4.1 Full Retrain at some steps during pruning

If we assume that pruning works as an architecture search, it means that the pruning pipeline of Algorithm 2 could be improved because we pruned a network based on non-optimal weights during the entire pruning. Indeed, since retraining from scratch after having pruned $k$ channels greedily gives better accuracy, we should benefit from retraining from scratch after having pruned $1, ..., n-1$ channels since the weights used for pruning decision would be "better".

This, however, does not scale at all since we have to train the network from scratch several thousand times rather than fine-tuning it with a few batches each time. An intermediary approach is thus to retrain the network from scratch a few times during the pruning process, which gives Algorithm 3. The bigger *num_retrain_scratch*, the slower the pruning is and the more accurate it should be. In Figure 4.5, we observe improvements on the pruning curve. However, when comparing the results obtained by both architectures, i.e. when retraining from scratch, we do not see much difference. Additionally, even with a few restart points, the training times become huge, and this method is thus not usable in the frame of this work. We can see the produced architectures in Figure 4.6. We still observe the weak pruning on the strided blocks but apart from that and the fact that the number of channels is monotonically decreasing in the last layers, there is no significant difference.

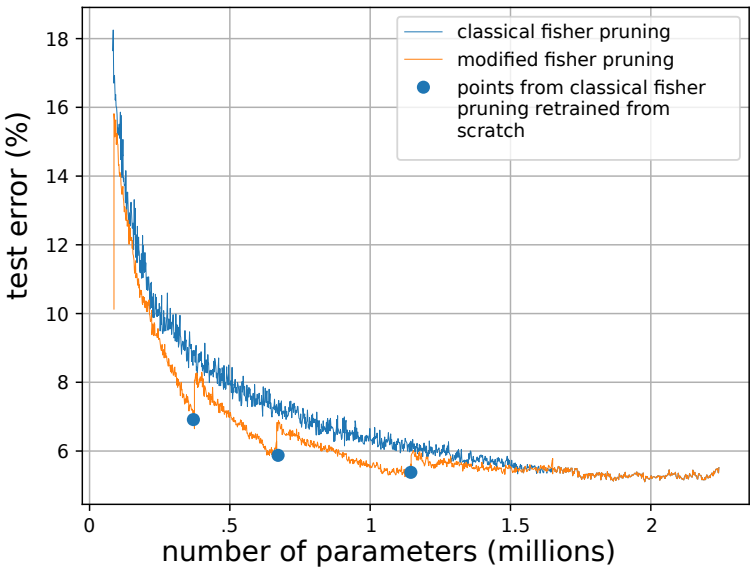
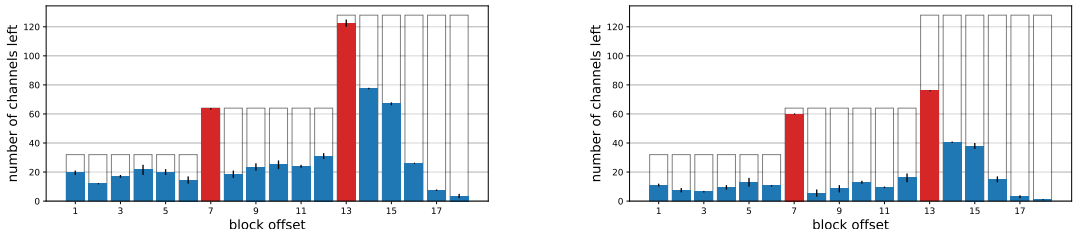



Figure 4.5: Comparison between classical and modified Fisher pruning of a WideResNet-40-2
Training curves and retrained from scratch points are averaged on two runs.




(a) 750 channels pruned (b) 1000 channels pruned

Figure 4.6: Number of channels per block (since we only prune inside of blocks) of a WideResNet-40-2 after Fisher pruning interleaved with retraining from scratch
The error bars represent the min and max over two trials. The red colour is used to show tensor spatial resolution reduction. The black borders show the initial number of channels

**Algorithm 3:** Modified greedy pruning algorithm

**Input:** the plain, pretrained network: *network*, the number of channels to prune: *num_channels_to_prune*, the number of times the network is retrained from scratch: *num_retrain_scratch*

**Output:** the pruned network, already retrained from scratch: *pruned_net*

`/*` *num_retrain_scratch* `is assumed to be a divider of` *num_channels_to_prune* `here */`

```
pruned_net = network
num_consecutive_greedy_prune = num_channels_to_prune/num_retrain_scratch
for i = 0 to num_retrain_scratch - 1 do
    for j = 0 to num_consecutive_greedy_prune - 1 do
        pruned_net = prune_one_channel(pruned_net)
        pruned_net = retrain_on_few_batches(pruned_net)
    end
    pruned_net = retrain_from_scratch(pruned_net)
end
return pruned_net
```

### 4.4.2 Improvements on Fisher pruning

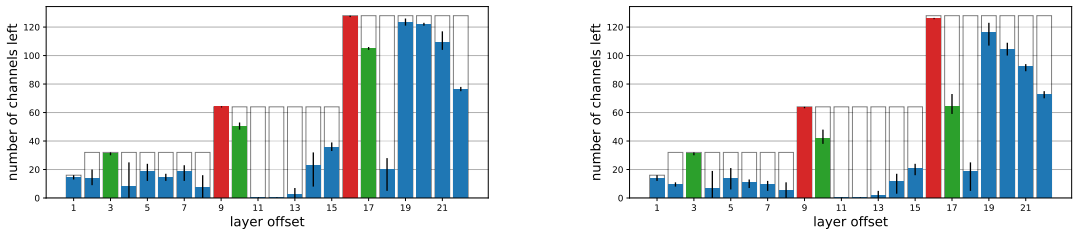
The approach of [Crowley et al., 2018] can easily be improved by applying two modifications:

1. Allowing to prune layers that are connected with other layers through skip connections by pruning the channels with a given offset in all of those at the same time (conv2 in Figure 4.2).

2. Using [Yang et al., 2018]'s look-up tables to compute the inference time cost of pruning a channel. Look-up tables are a method that allows using real hardware measurements to compute the inference time of a network (see Section 4.5). Instead of pruning the channel that has the smallest influence on the loss, we can then prune the channel that has the smallest ratio $\frac{\Delta fisher\_score}{\Delta cost}$.

By using these improvements, we still notice that we are better off retraining the obtained architectures from scratch rather than using the values of the weights coming from the original model. By having a look at the pruned architectures on Figure 4.7, we can see that

the layers of the second subnetwork are more pruned and the layer of the third subnetwork are less pruned than with classical Fisher pruning. This makes sense since the layers of the second subnetwork have a bigger cost than the layers of the third one, taking the cost into account will thus benefit the latter and result in a more aggressive pruning of the former. We can also notice that, as before, the layers applying stride are spared and that this is also the case for the layer linked through skip connections.




(a) 600 prunings (b) 750 prunings

Figure 4.7: Number of channels per layer of a WideResNet-40-2 after Fisher pruning all layers using performance look-up tables
The error bars represent the min and max over three trials. The red colour is used to show tensor spatial resolution reduction. The black borders show the initial number of channels. Since we prune more layers than on Figure 4.6, we have new errors bars, (1) the leftmost blue layer is the first layer of the network (taking the image as input), (2) the green layers each represents 6 layers connected with a skip connection and thus sharing the same number of output channels

## 4.5 Real inference time look-up tables

### 4.5.1 Introduction

The **layer-wise look-up tables** are a promising idea to compute the inference time of all the pruned variations of a network. Usually, inference time is measured through abstract metrics (ex: number of FLOPS). Their main advantage is that they are straightforward to compute: given the architecture and number of channels at each layer, a simple calculation allows us to get the number of FLOPS or parameters. On the other hand, these metrics are not always good proxies of the real inference time of the network on a given device. Measuring inference time directly can be a solution [Tan et al., 2018] but is not affordable for pruning since the number of measures to make increases exponentially with the depth of the network. An important contribution of [Yang et al., 2018] was to introduce look-up tables inference computations. The main idea is to factorize the inference time between each layer of the network, i.e. to compute the total cost of the network as the sum of the cost of each layer. The inference times of these layers can then be measured independently (for several numbers of input and output channels) which makes the number of measures increases linearly with the number of layers (if not less since we can reuse the measurements made on two identical layers). Figure 4.8 illustrates how to compute the latency of a network of two layers.

### 4.5.2 Implementation

The central part of the implementation is to perform the actual inference time measurement on the Raspberry Pi. Initially, we tried to perform these measurements directly

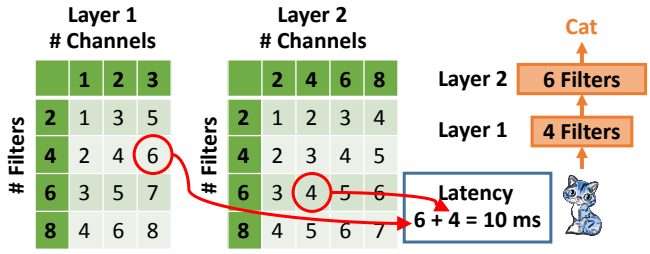



Figure 4.8: Illustration of the look-up tables mechanism, figure taken from [Yang et al., 2018]
In the notation used in the rest of this Thesis, "Filters" is called "output channels" and "Channels" is called "input channels"

on our training machine and within the python interpreter as a proof of concept. Figure 4.9 displays inference times computed at that stage, as we can see on the graph, we implemented several variants:

1. Since the bulk of the code was written in PyTorch, we decided to use PyTorch as well on top of the python interpreter. The problem with this approach is that there are huge spikes in some places (and, as we discovered later, the cost is significantly higher than tf-lite)

2. To try to solve the pikes problem, we had the idea to run a prediction on a batch of images of small size, this did not improve the results however

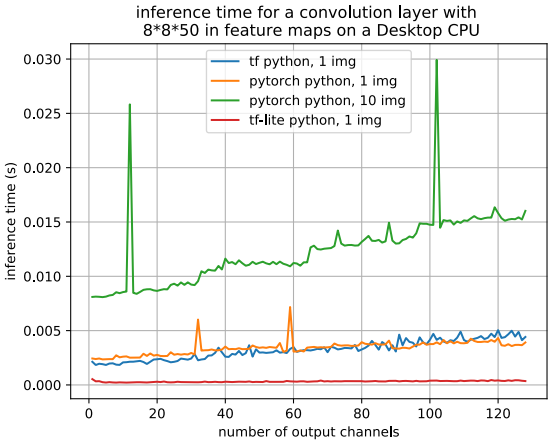
3. We then tried to use Tensorflow (still on top of python) thinking the problem might come from the fact that PyTorch uses dynamic graphs, at this point the huge pikes disappeared, but we still had a quite fuzzy graph

4. Using tf-lite inside of python did significantly speed-up the computation, but we still had the fuzziness problem

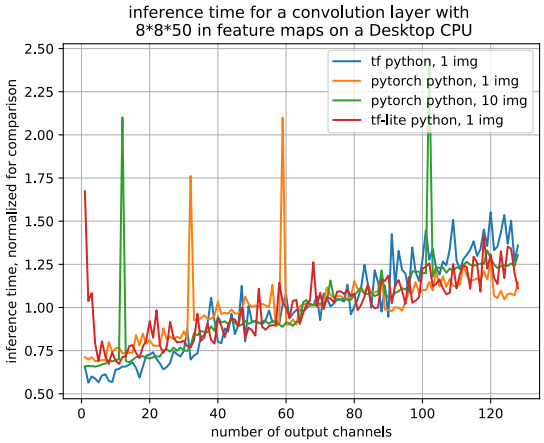
We then proceed to try running tf-lite on the Raspberry Pi, at first, we compiled the "models"[1] to tf-lite files on the Raspberry Pi (which had the advantage of not to have to store the whole pack of them at the same time) but was unfeasible in terms of time (it needed one day to compute a small fraction of a table).

To solve that problem, we performed the compilation to .tflite files on a desktop computer (this still took one full day) and ran the predictions on the Raspberry Pi with the help of a small C++ benchmark program that comes alongside tf-lite and a tiny bash script we coded ourselves. As we can see in Figure 4.10, there are still two types of pikes, the first one is a tiny reduction of the inference times when the number of output channels is a multiple of 4, we do not know what it is due to but it looks more like a consequence of the library implementation than to measurement noise. On the other hand, there are also positive pikes. We think these are due to measurement errors (for example, it could be that the processor runs slower to cool down during some periods). Some observation going in that direction is that the numbers of channels at which the spikes occur change when we take the measurements a second time.

[1] for each layer type, each possible number of input channels and each possible number of output channels of that layer, we need to make a measurement, i.e. build a trivial (one layer) model, we thus have thousands of models to build



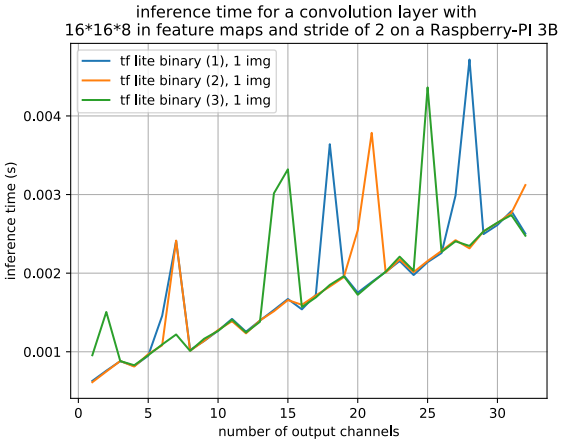

(a)



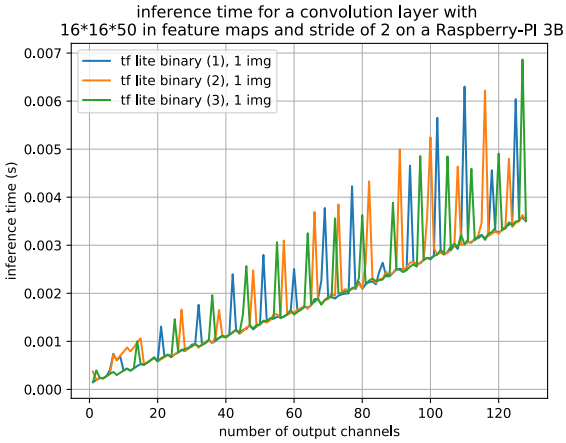

(b) same graph where signal intensities are normalized (the average over the number of output channels of the inference time is 1)

Figure 4.9: Inference times while varying the number of output channels for a convolution layer as measured on a Desktop CPU with different frameworks inside of a python process.

A solution to this problem would be to take the mean over several measurements (this is not too costly to compute, building the models takes a day on a desktop computer while running the measurements on a Raspberry Pi only takes a few hours) and then to apply 2D Gaussian filtering to it. The results are displayed in Figure 4.11.




(a)




(b)

Figure 4.10: Inference times while varying the number of output channels for a convolution layer as measured on a Raspberry Pi 3B with tf-lite binaries

We can also draw several conclusions on the inference times of different layers on the Raspberry Pi 3B from that figure. The number of FLOPS seems to be a good proxy on the inference time on this hardware between different plain convolutional layers. All other things being equal, $1 \times 1$ conv seems much cheaper than $3 \times 3$ (there should be a factor of 9). Reducing the tensor width and height by a factor of 2 seems to reduce the inference time by a factor of 4. We can also see that $Stride\_x$ and $No\_Stride\_x$ have the same

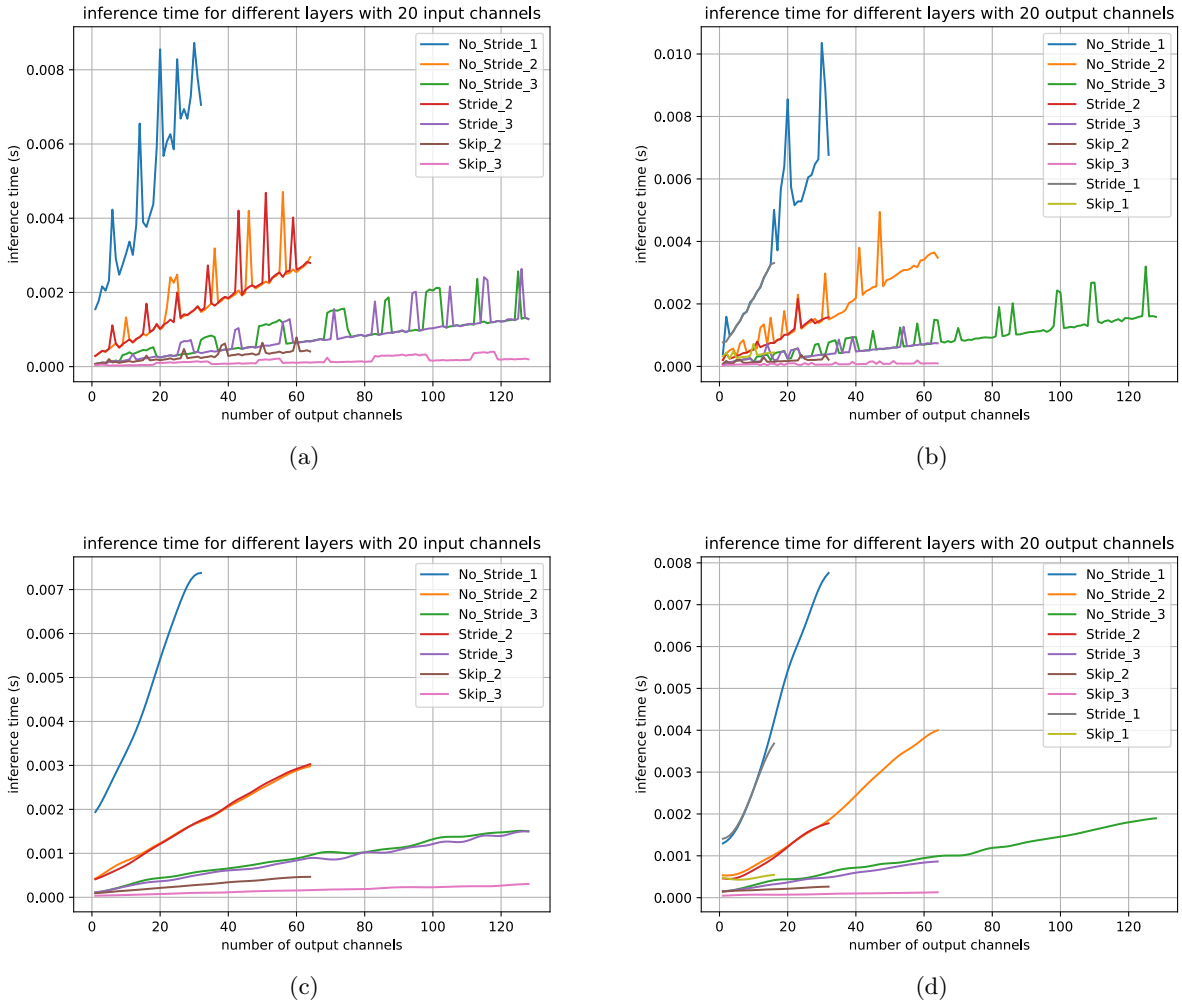



Figure 4.11: Inference times while varying the number of channels for several convolution layers as measured on a Raspberry Pi 3B with tf-lite binaries
Stride and No_Stride are $3 \times 3$ convolutions with or without f.m. resolution diminution, Skip layers have $1 \times 1$ filters and 1, 2 or 3 indicates the depth in the network ($1 = 32 \times 32$ output tensors, $2 = 16 \times 16$ output tensors, $3 = 8 \times 8$ output tensors).
In (a) and (b), lines are a single prediction. In (c) and (d), lines are the application of 2D Gaussian filtering on the average over 3 predictions.

inference times which is coherent with the number of FLOPS (*Stride_x* has 2 times wider and higher inputs, but a stride of 2 is applied). We can, however, notice a difference when varying the number of channels, the FLOPS predict lines of an equation $y = ax$ with a bias of 0, whereas we observe a non-negligible bias here.

Lastly, we checked on all the pruned architectures we obtained in the following sections that the predicted inference times were sufficiently precise. This is indeed the case, the correlation is significant $> 99\%$, and the scaling factor between prediction and reality seems very close to 1. This is surprising considering the prediction do not take the skip layers (i.e. tensor addition) into account. A scatterplot is shown in Figure 4.12. This is excellent news. It shows that look-up tables are a great proxy for real-time inference (at least in our use case).

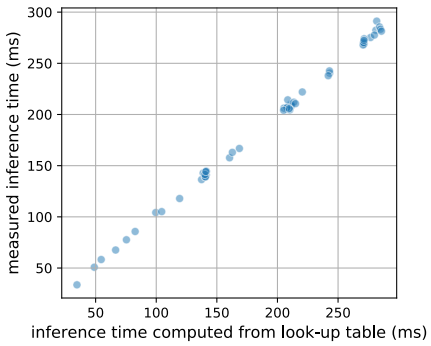



Figure 4.12: Jointplot between predicted and measured inference time for pruned WRN-40-2

## 4.6 NetAdapt

NetAdapt [Yang et al., 2018] is a quite unorthodox pruning meta-algorithm. Additionally, the authors introduced the look-up table mechanism detailed in Section 4.5. The paper's reported results also looked very promising, beating width multipliers.

The meta-algorithm consists in pruning the network several channels at a time. At each step, we compute, for every layer $1 \leq l \leq L$, how many channels $c_l$ should be pruned to achieve a given inference-time reduction objective without pruning the others. Then, we build $L$ models, where model $1 \leq i \leq L$ is the original model where $c_i$ channels of layer $i$ have been pruned according to some criterion and fine-tune the pruned models on a few batches of data. The authors use the $L_2$ norm of the weights as criterion. Once this is done for every layer, we compare the error increase of all the new networks on a holdout set (data not used for training nor testing the accuracy of the network). We keep the pruned network with the smallest error and use it as a baseline for another iteration until some performance criterion is met. This is illustrated in more details in Algorithm 4.

### 4.6.1 Implementation

There was no implementation of NetAdapt available online. We thus had to implement it ourselves. Due to the complexity of the task, we decided to implement that network using the PyTorch framework since we are more familiar with and have a much better user-experience using it. However, the only working solution to compute look-up tables was to use tf-lite and thus, we had to use Tensorflow and PyTorch alongside each other which introduced code repetition.

We wanted our network object to handle Fisher pruning (see Section 4.6.2).

We needed to be able to build a copy of a pruned model (to build the pruned models online 7 in the algorithm).

We shared look-up tables between several layers having the same parameters (i.e. same input resolution, no. of strides, ...) to avoid useless measurements.

To handle the parallel fine-tuning of several networks (inner loop on line 5). We had to move models from GPU to CPU and the other way around, as well as destroying several

**Algorithm 4:** NetAdapt pruning algorithm

**Input:** the plain, pretrained network with $K$ layers: *network* and its initial cost *init_cost*, the inference cost target: *objective_cost*, the inference cost reduction achieved at the first step: $first\_step\_cost\_red$, the step-wise cost reduction decay factor: *step_cost_red_dec*

**Data:** a train set to perform fine tuning and a holdout set to pick the best candidate

**Output:** the pruned network: *pruned_net*

1 $pruned\_net = network$
2 $cur\_cost = init\_cost$
3 $step\_cost\_red = first\_step\_cost\_red$
4 **while** $cur\_cost > objective\_cost$ **do**
5  **for** $k = 0$ *to* $K$ *-* $1$ **do**
6  $num\_chan_k, res\_gains_k =$ choose_num _filters$(pruned_net, k, step_cost_red)$ `/* computes the number of channels to prune and the corresponding inference gains, if it is impossible to achieve` *step_cost_red* `by pruning this layer, it is skipped */`
7  $pruned\_net\_candidate_k =$ prune$(pruned_net, k, num_chan_k)$ `/* prunes the` $num\_chan_k$ `channels whose weights have the smallest` $lL_2norm$ `out of layer` $k$ `*/`
8  $pruned\_net\_candidate_k =$ fine_tune$(pruned_net_candidate_k, train_set)$
9  **end**
10  $pruned\_net, res\_gains =$ pick_best_network$(pruned_net_candidate_:, res_gains_:, holdout_set)$
11  $cur\_cost = cur\_cost - res\_gains$
12  $step\_cost\_red = step\_cost\_red \times step\_cost\_red\_dec$
13 **end**
14 **return** *pruned_net*

`/* the function pick_best_network on line 10, takes the best network according to both the performances gains and the accuracy losses, the paper used a function called pick_best_accuracy (name which is indicating that it only took the accuracy into account) which took both the variation of accuracy and of performances as arguments. This was not clarified in the paper, and we thus chose to take both accuracy loss and performance gains into account. When accuracy improved (which is possible), we only maximized the accuracy. Otherwise, we minimized the positive ratio` $\frac{-\Delta acc}{\Delta perfs}$ `*/`

models throughout the algorithm without causing memory leaks.

We wanted to prune layers linked by skip convolutions as well because skip-connections are very common in state-of-the-art architectures ([Crowley et al., 2018] only pruned convolutions in the bottleneck blocks, see Figure 4.1). This introduces a problem since, to prune the $k$th channel of a given layer, we have to prune the $k$th channels of all the layers it is linked to by skip convolutions. To do so we considered the $n$ connected layers as one, i.e. in function choose_num _filters (line 6), we used the sum of the inference costs of the $n$ tables (and the other tables affected by the pruning) and in prune (line 7) we chose which offsets to prune based on all of the channels.
Finally, we also wanted to remove a channel from the network once it is pruned. There are

two ways to handle pruning: 1) to add a binary mask after each layer and to zero-out the outputs of pruned channels (the approach taken by [Crowley et al., 2018]) 2) to replace the pruned layers by new objects having fewer channels but keeping the un-pruned weights of the layer. The second technique is more efficient since once the network is half pruned, we perform backpropagation on two times fewer parameters. It is also more complex to design.

For the reasons mentioned above, the NetAdapt algorithm was quite complicated to implement.

### 4.6.2 Variations

A variation that comes easily to mind after having looked at the previous sections is to use Fisher pruning instead of $L_2$ norm pruning to decide which channels to prune inside of a layer.

According to Algorithm 4, we only prune channels inside of a layer if this allows us to achieve the reduction objective for this iteration (see Line 6). This means that, if a layer ends up having a few channels, the algorithm never prunes the last channels of that layer because it does not give enough inference time reduction. [Yang et al., 2018] designed their algorithm in this way because they tested it on MobileNetv1 [Howard et al., 2017]. The latter is a network without skip connections. This means that completely pruning a channel is unthinkable because it would cut the network in half. We thus propose a modification called *allow small prunings* where we prune the layer having the best accuracy/cost ratio without limiting ourselves to the layers that can achieve the current iteration's reduction objective.

A comparison between the four combinations of these two modifications is shown in Figure 4.13. We do not see any difference between the performances of these methods. We did not notice any difference when looking at the number of channels pruned either.

The lack of influence of the quality of the pruning algorithm (Fisher vs $L_2$ norm) can be explained by the fact that NetAdapt performs several prunings in parallel and only takes the pruning that performs better on a holdout set.

Regarding the adaptation to prune the last channels of a layer, it appears that having a layer with only a few channels is not that problematic in our case. This seems to contradict the lookup tables from which we can infer that extremely thin layers are much more time consuming than others at an equivalent number of FLOPS (because of the bias in the accuracy as a function of inference plots). A possible explanation for this would be that channels of thin layers are much more useful to predict channels of wider layers.

A drawback of this algorithm is its important training times. A modification to mitigate that issue could be to randomly sample a proportion of the layers at each iteration and only prune among these layers to create new networks. Due to the iterative nature of the algorithm, skipping some layers which will be seen at a later stage might not prove harmful while allowing for much faster training time. This idea was suggested by Jean-Michel Begon during the end of the thesis and was thus not developed in this work. We think it could be very useful to alleviate the training time problem.

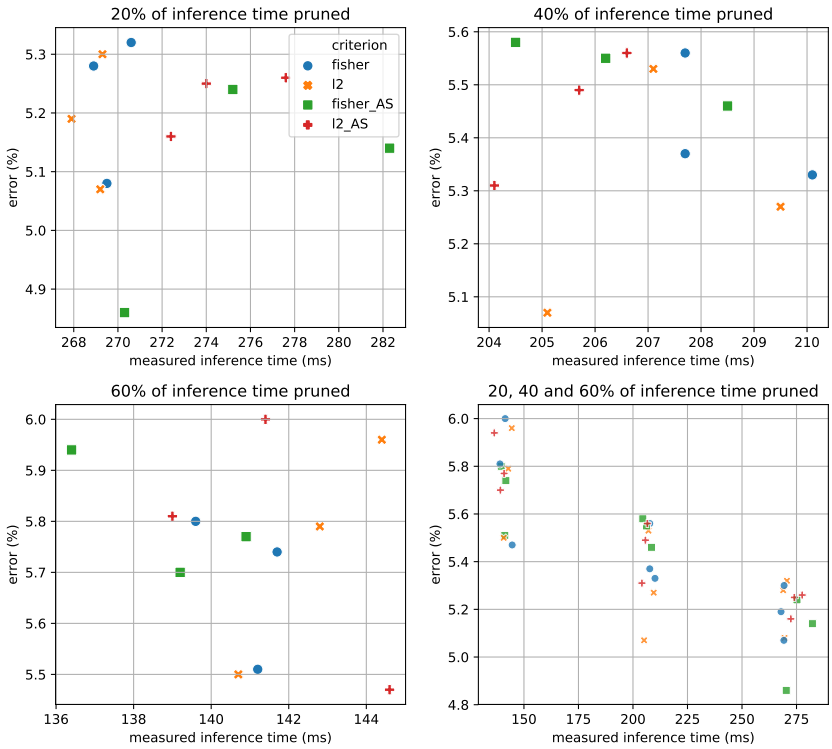



Figure 4.13: Performance plots between the 4 variations of NetAdapt (networks retrained from scratch), the bottom right graph shows all the data points, and the 3 others are zooms over the latter
The acronym "AS" stands for "allows small prunings", i.e. the second adaptation of NetAdapt.
In this chapter, we show several points per run rather than the mean and std for $x$ runs, this was done because different runs ends up with lightly different architectures having inference time differences

### 4.6.3 Retraining from scratch

The paper used the weights that resulted from pruning and fine-tuned them for an important number of epochs, i.e. they did not retrain the weights from scratch. Figure 4.14 shows the difference between the paper's method and retraining from scratch (including the 3 variations of Section 4.6.2), as we can see, it is preferable to retrain from scratch. The results without retraining from scratch are also surprisingly less good than what was announced in the paper. It could be explained by the fact we prune a different architecture on a very different dataset measuring inference on different hardware.
We also hypothesized it could be due to an error from our part where we used too few fine-tuning steps between each pruning iteration. We thus repeated a subset of the experiment with eight times more fine tuning (purple points on Figure 4.14) but this did not change anything to the fact retraining from scratch is better nor the accuracy of the base method.

We can have a look at the channels pruned in Figure 4.15. We can directly notice that it resulted from pruning important numbers of channels at a time due to the look of the number of channels per layer and the large error bars. These also result from the fact that NetAdapt is not extremely precise on the fraction of inference time pruned since the algorithm only takes a lower bound as input. Regarding which layers were pruned, we can

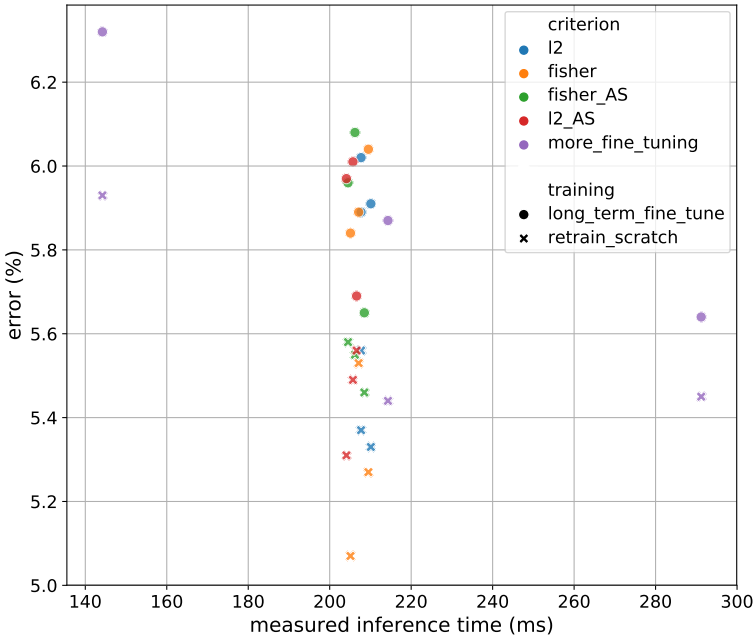
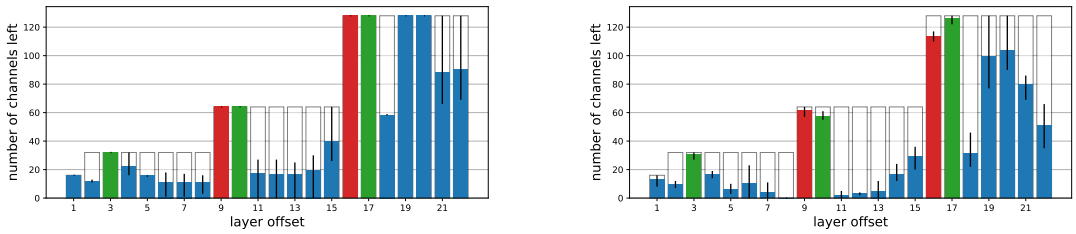



Figure 4.14: Comparison between retraining from scratch and [Yang et al., 2018]'s approach (called "long_term_fine_tune")
"AS" stands for "allow small prunings" (see Section 4.6.2), "more_fine_tuning" is "fisher_AS" where we applied 8× more fine-tuning

notice a preference for the two first sub-networks, this was also the case for Fisher pruning with the help of performance tables (see Figure 4.7).




(a) 40% of inference time pruned (b) 60% of inference time pruned

Figure 4.15: Number of channels per layer of a WideResNet-40-2 after pruning using the NetAdapt Algorithm
The error bars represent the min and max over three trials. The red colour is used to show tensor spatial resolution reduction. The black borders show the initial number of channels. Since we prune more layers than on Figure 4.6, we have new errors bars, (1) the leftmost blue layer is the first layer of the network, (2) the green layers each represents 6 layers connected with a skip connection and thus sharing the same number of output channels

## 4.7 MorphNet

We decided to consider MorphNet [Gordon et al., 2018] in our study because it has a different approach to pruning. Indeed, instead of training the network from scratch and then iteratively removing channels, the network is trained with a penalty that zero-out some outputs of the layers, the pruned architecture can then be extracted from this network. The penalty added to the loss (with a weight factor $\lambda$) takes the form of $\sum_{all\_channels} |cost_{channel} \times \gamma_{channel}|$, where $cost_{channel}$ is the cost in terms of FLOPS of the channel and $\gamma_{channel}$ is the scaling factor associated with the channel in the next batchnorm layer (this is the pruning criterion introduced in Section 4.2.5).

To handle the problems that layers combined by skip connections must have the same number of channels, group LASSO using $L_\infty$ norm is introduced. For example, if $a$ and $b$ are two channels that are added together in a skip connection; pruning neither $a$ nor $b$ does not introduce a penalty from the group LASSO, pruning one of them or both introduces a penalty from the group LASSO of the same magnitude. That encourages SGD to either prune both $a$ and $b$ or neither of them.

The authors also introduced the possibility to, once a pruned network is trained to convergence, multiply its number of channels by a constant factor and reapplying the pruning procedure to the widened network. This is interesting since it allows a pruning algorithm to give more channels to specific layers than they had in the initial network. On the other hand, it is also more time consuming and did not induce significant improvements in [Gordon et al., 2018]. We thus did not explore that approach.

MorphNet does not allow us to directly select the amount of inference-time gains we would like to benefit from or the number of channels to prune. Instead, we have to modify the weight $\lambda$ of the penalty term in the loss function. This proved to be quite annoying in practice since modifying $\lambda$ leads to networks that might not converge or might not prune any layer (although, as said in the paper, whether the network would converge or prune anything was apparent after some epochs). We also had to fine-tune a threshold on the batchnorm scaling factors under which we considered that the corresponding layer was pruned.

Finally, we retrained the pruned-architecture from scratch after the pruning phase, as suggested in Section 4.3.

We have now introduced all the variations to pruning algorithms we developed. In the next section, we will compare their performances in terms of accuracy at a given error on a WideResNet-40-2.

## 4.8 Comparison

All the previously described algorithms are compared on Figure 4.16. We also included uniform pruning and the smaller WideResNets (without SE blocks) defined in the previous chapter (Table 3.3) as a baseline. For all of the pruning algorithms, we started from a plain WideResNet-40-2 and pruned so as to cover a wide range of latencies.

We can, unfortunately, see that there is not any noticeable difference between the results of the pruning algorithms, although some pruning algorithms seemed much more promising than others or uniform pruning and that the results in terms of number of channels pruned per layer vary as well. We can also see that for large prunings ($350 \rightarrow 100$

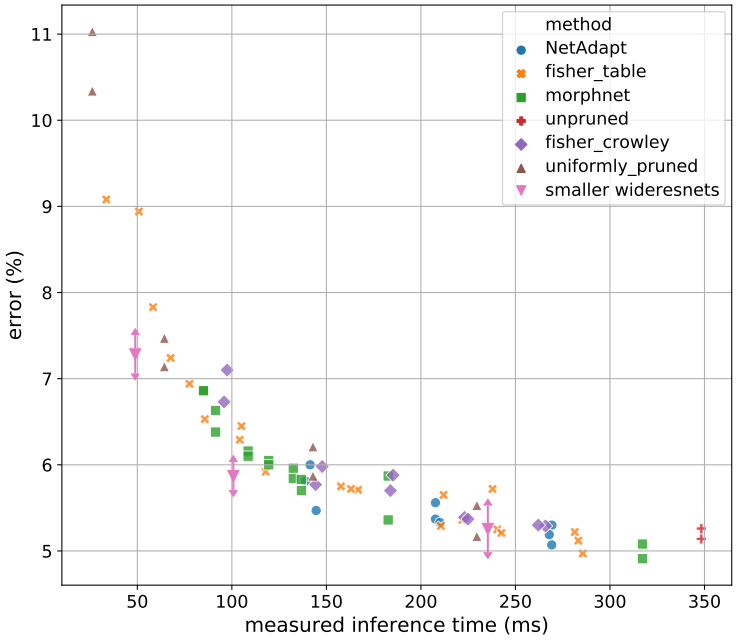



Figure 4.16: Comparison between all the pruning algorithms (with retraining from scratch) presented in this section as well as uniform pruning and smaller architectures
"fisher_crowley" is the basic fisher prunig as used by [Crowley et al., 2018], i.e. not including all the modifications we discussed. For the smaller architectures, we display the error and std over 3 runs.

ms of inference), we get better results by performing grid search on different combinations of width and depth. This makes sense for uniform pruning since we could end up with too thin architectures. It is however disappointing to be the case for NetAdapt and our variation of Fisher pruning using the performance tables since these allow to remove entire layers and could thus have found the smaller architectures by themselves. It is also important to remember that these results are only shown for WRN-40-2 on a Raspberry Pi 3B using CIFAR-10 as the dataset. Things might be different with other networks, datasets or hardware, but we did not get the chance to test it in this work.

## 4.9 Conclusion

In this chapter, we compared several pruning methods and made some modifications by mixing them. All of these methods were modified to be retrained from scratch, assuming it would improve accuracy. We studied the effect of retraining several times the network from scratch during architecture search. Next, we extended the method of Fisher pruning to handle correctly skip connections and take (some proxy of) the inference time into account. Finally, we dwelled on a more sophisticated pruning algorithm, NetAdapt, and analyzed how the different novelties impacted its performance.

We can draw several conclusions from this chapter:

1. Retraining from scratch, i.e. formulating the problem as an architecture search one,

yields better accuracies for all methods.

2. Although there are differences in the channels pruned by each algorithm. They all spare the layers that reduce the spatial resolution of the tensors.

3. Estimating inference time from look-up tables works very well. The main difference with FLOPS is the important bias when presenting the inference time as a function of the number of input or output channels.

4. Unfortunately, there is no clear winner between all of those methods. They are also less performant than a simple grid search on the width and depth of the networks when the inference time reduction is important.

Initially, the main goal of this chapter was to use one of those pruning methods to improve the architectures developed in the previous section. However, since it showed unpromising results on WideResNet, we decided to focus on other approaches instead. This bring us to the next chapter, that introduces methods to improve the accuracy/inference time tradeoff once the architecture is fixed.

# Chapter 5

# Knowledge distillation and quantization

## 5.1 Introduction

Other than modifying the architecture used and its depth/width, we can also use training time optimizations to improve the accuracy/inference time of our network. We explore two very different methods that allow doing so in this chapter.

First, we use knowledge distillation [Hinton et al., 2015, Mirzadeh et al., 2019] to improve the accuracies of our networks in Section 5.2. Second, we use quantization [Krishnamoorthi, 2018] to reduce the inference times of our networks while preserving the accuracy as much as possible in Section 5.3.

## 5.2 Knowledge distillation

### 5.2.1 Method

Knowledge Distillation [Hinton et al., 2015, Mirzadeh et al., 2019] is a technique consisting of using the predictions of a big network, the teacher, to help a smaller network, the student, improve its accuracy. More specifically, instead of just training to reduce the error with the true classes, we (partly) train to mimic the predictions of the teacher. Let us first introduce some terminology before explaining what it exactly means.

Multilabel classification networks always use softmax as a final activation, this activation turns logits, which are unconstrained real numbers into so-called probabilities[1] by normalizing those according to:

$$prob_i = \frac{exp(logit_i/T)}{\sum_j exp(logit_j/T)} \tag{5.1}$$

where $T$, the temperature, is conventionally set to 1. These probabilities are called soft targets. Increasing $T$ will result in a softer probability distribution.

This distribution gives an idea on how close several labels are, for example, we can infer from a model giving a final score of 0.7 to "car", 0.2 to "plane" and 0.001 to "horse"

[1] These are not probabilities in the sense that the network considers there is a $x_i$ probability the correct label is $i$

that a plane is closer from a car than a horse. This is hypothetized to be the reason why the predictions of the teacher helps the student to train. Using a bigger temperature also softens those probabilities, which help the knowledge distillation in practice.

The classical cross entropy loss for a signle labelled sample[2] is given by

$$CE\_loss = \mathbb{E}_{label}(-log(prob)) \tag{5.2}$$

$$= -\sum_{i}^{C} label_i \; log(prob_i) \tag{5.3}$$

$$= -\sum_{i}^{C} label_i \; log(\frac{exp(logit_i)}{\sum_j exp(logit_j)}) \tag{5.4}$$

where *label* are the true labels of the $C$ classes of the dataset (1 if the offset of the class is $i$, otherwise 0) and *prob* denotes the output probabilities of the network (after softmax activation).
Knowledge distillation consists in adding a new weighted term to this loss that we call $soft\_CE\_loss$:

$$KD\_loss = \lambda \; CE\_loss + (1-\lambda)T^2 \; soft\_CE\_loss \tag{5.5}$$

$$= -\lambda \sum_{i}^{C} label_i \; log(\frac{exp(logit_i)}{\sum_j exp(logit_j)}) \tag{5.6}$$

$$- (1-\lambda)T^2 \sum_{i}^{C} teacher\_soft\_target_i \; log(\frac{exp(logit_i/T)}{\sum_j exp(logit_j/T)})$$

where $teacher\_soft\_target$ are the probabilities given by the teacher network while using the temperature $T$. We can see that the second term corresponds to a cross-entropy computed between the soft targets of the teacher and the student. The $T^2$ factor is there to compensate that the gradient of the $soft\_CE\_loss$ scales as $1/T^2$.

Another use of Knowledge distillation introduced in [Hinton et al., 2015] is to train specialists networks that focus on a subset of the classes, but we do not consider this approach in more details here.

Very recently, [Mirzadeh et al., 2019] studied the impact of the difference in accuracy between the teacher and the student. They showed that, given a fixed student architecture, increasing the size (and thus accuracy) of the teacher decreases the performances of the student model with knowledge distillation at some point. This is due to opposing effects. On the one hand, the more accurate the teacher, the better the information transmitted through its soft targets. On the other hand, a more accurate teacher, is more confident about its predictions making its soft targets contain less information. In addition, a more accurate teacher is harder to mimic.

To solve this problem, the authors introduced the usage of several teacher assistant (TA) networks. These would be networks of intermediary size between the student and the teacher that would learn from the teacher (or from more accurate TA networks) and transmit knowledge to the student (or smaller networks). That approach shows promising

[2] we consider this case for simplicity

results.

Another property of that algorithm is that increasing the number of TA networks always increases the performances of the student. However, this also importantly increases the training costs since each new TA networks is a new network to train. Additionally, training networks must be done sequentially since we need the predictions of the teacher to train the student. The authors also suggested empirically that, given a teacher and a student, the best TA was the one with an accuracy equal to the mean of both teacher's and student's accuracies.

### 5.2.2 Implementation

Since there is a tradeoff in the number of TA to have, we chose to use the MnasNets, MobileNetsv1 and v2 with SE blocks of Figure 3.21, i.e. the same as in Chapter 3, as TA's and student. The teacher is a WideResNet of depth 40 and width factor 4 with Squeeze-and-Excitation blocks (4.33% of error on the test set, averaged on 3 runs), to achieve better accuracy than our most accurate networks. The knowledge distillation setup is illustrated in figure 5.1. The goal of this experiment is to show the benefits KD could have on the best performing networks of Chapter 3.

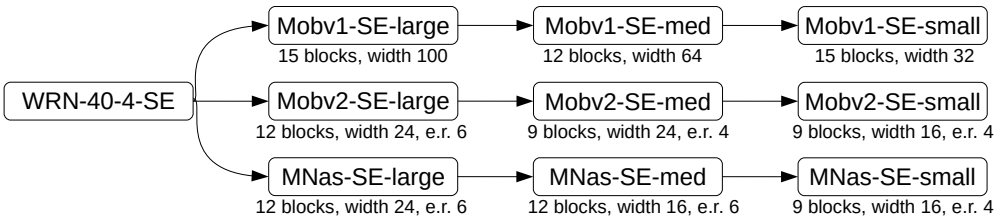



Figure 5.1: trainer, TA, student setup for knowledge distillation
An arrow from A to B means we use A's predictions to train B

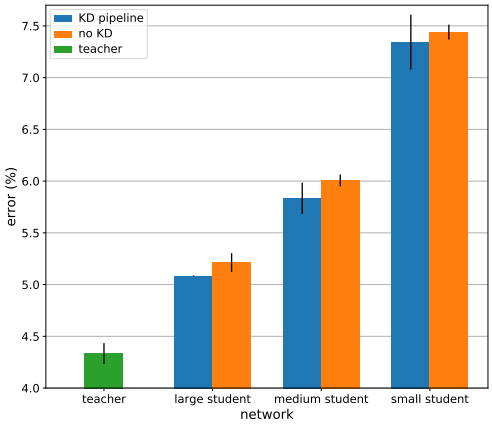
Considering the two hyperparameters used in knowledge distillation, namely $\lambda$ and $T$ in equation 5.5, we chose to set $\lambda$ to 0.2 and $T$ to 2. These results were chosen on a grid search on $\lambda \in \{0.02, 0.1, 0.2\}$ and $T \in \{2, 5, 10\}$ on knowledge distillation from a WideResNet of depth 40 and widen factor 4 with SE blocks (WRN-40-4-SE) to a large MobileNetv1 (on the validation set). We could have performed a more thorough hyperparameter fine-tuning but were caught by time. Regarding the other hyperparameters used for training, we used the exact same settings as in Chapter 3, only changing the loss to add knowledge distillation.

Another practical problem of importance to perform knowledge distillation is the time at which to compute the predictions of the teacher. When no knowledge distillation is performed it is very interesting to memoize the outputs of the teacher since these do not change durng the whole training. It results in *num_epochs* times less computations. Additionally, when performing the teacher predictions at inference, the teacher must often run on CPU since the student already fills the GPU, which is much slower. Unfortunately, when using data augmentation, the number of outputs to memoize increases exponentially.

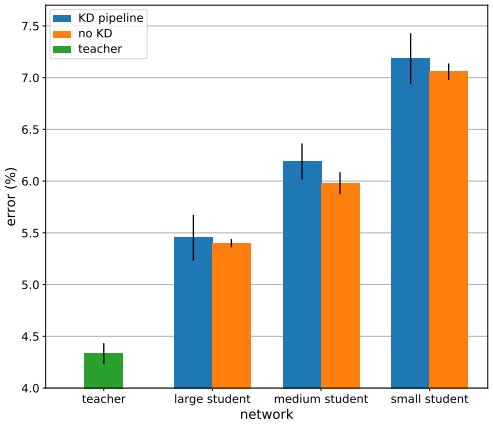
In our case, since we use lightweight data augmentation, we "only" have 128 data augmentation possibilities (2 horizontal flip $\times$ 8 x-axis random crop $\times$ 8 y-axis random crop). Since we train for 200 epochs, memoization is more computationally interesting ($128 \times 50,000$ predictions instead of $200 \times 50,000$), and we used that approach.

### 5.2.3 First results and improvements

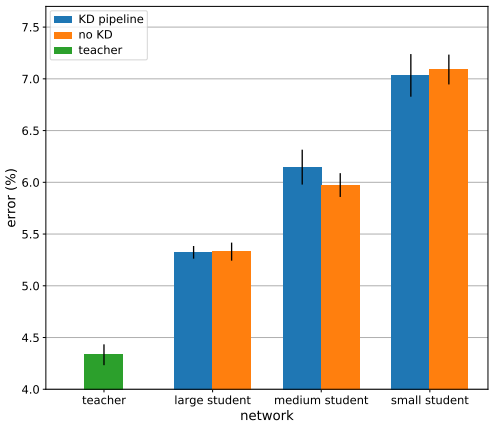
As we can see in Figures 5.2a, 5.2b and 5.2c, knowledge distillation did not improve the accuracies for networks from the pipeline of Figure 5.1. Indeed, although MobileNetv1 results seem overall positive (error decrease when using KD on 2 out of 3 cases). This is not even the case for the other two networks. Additionally, results are generally small both with respect to the standard derivations and in absolute to conclude a noticeable difference in our opinion.



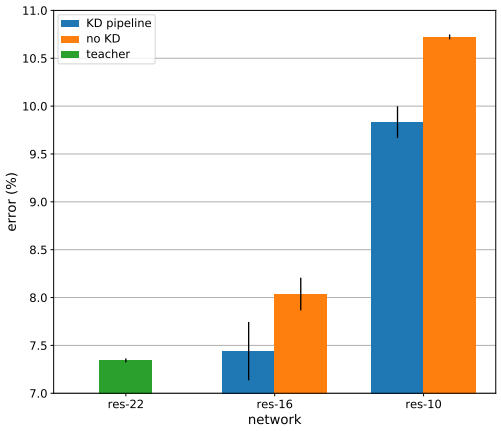

(a) KD on MobileNetv1




(b) KD on MobileNetv2




(c) KD on MnasNet




(d) KD on resnets (for less accurate networks)

Figure 5.2: Error evolution when using knowledge distillation
Displays the mean and std computed on 3 runs. Teacher is displayed in green, TAs/students in blue and the same networks retrained without KD in orange. Please note that Figure 5.2d has higher error rates

This surprised us. Even though we use different networks and aim for higher accuracies, [Mirzadeh et al., 2019] reported accuracies improvements over the normal training procedure. We decided to try our knowledge distillation on settings analogous to an ex-

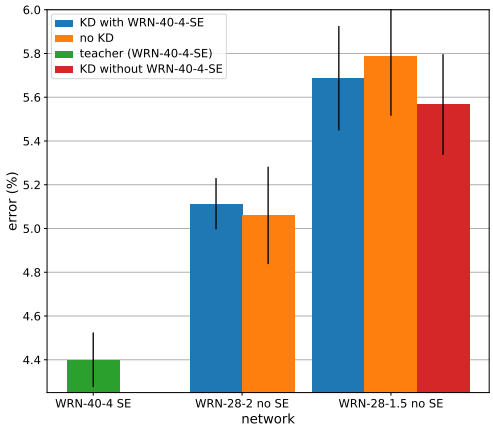


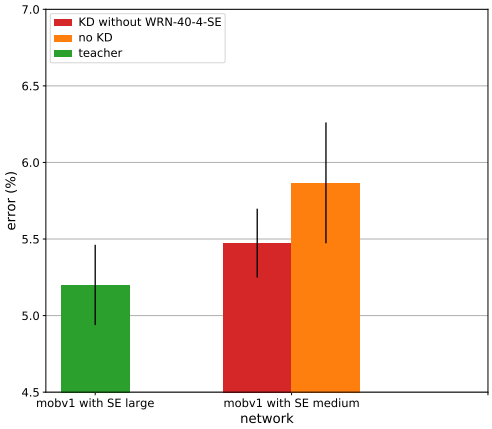

Figure 5.3: Error (on the validation set) evolution when using knowledge distillation on ResNet with and without WRN-40-4-SE as teacher
Displays the mean and std computed on 3 runs. Teacher is displayed in green, the networks retrained without KD in orange and the TA/students in blue when using WRN-40-4-SE as a teacher and red otherwise.

periment of [Mirzadeh et al., 2019]. As we can see in Figure 5.2d, using teaching assistant knowledge distillation (teacher: ResNet of depth 22, TA: ResNet of depth 16) on ResNets allows a significant accuracy improvement of about 1% on ResNet of depth 10. The accuracy gains for the TA being of the order of 0.5%. We thus concluded that our KD implementation was correct.

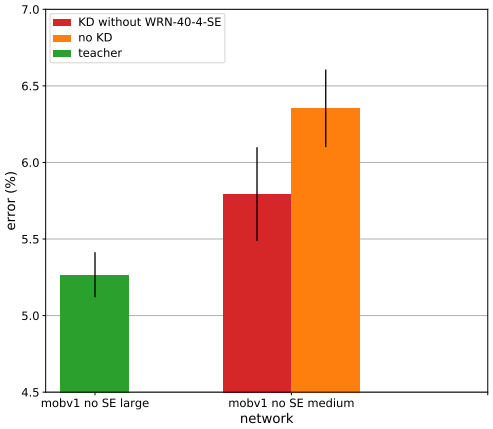
We thought about three explanations for the failure of knowledge distillation in the previous experiment. First, it could be due to the architecture of the networks, for some reason, knowledge distillation would fail on MobileNetv1, MobileNetv2 and MnasNet with SE blocks. Second, it could be due to the error range, knowledge distillation fails to help when our error is small (here between 5 and 8%). Third, it could be that the teacher, WRN-40-4-SE cannot transfer knowledge, and, worse, corrupts its TA so that they are not helpful for KD either. Unfortunately, we were urged by time while doing these experiments, we thus did not run as much tests as we would have liked but we think they are numerous enough to draw conclusions from them.

To test all of these hypotheses, we ran several experiments on our validation set. First, we tried to distill knowledge to WideResNets using the WRN-40-4-SE as teacher. The setting is the same as in Figure 5.1 except that we do not train small networks. The WideResNets used are those from Table 3.3. We can see on Figure 5.3 that this technique does now show any improvement. To test if the problem came from the WRN-40-4-SE, we tried to distill the knowledge from a large WideResNet (without SE), trained without KD from the WRN-40-4-SE, to a medium WideResNet (without SE). To our delight, we found out that this reduces the error of the medium WideResNet of 0.22%. This is not so important but is still appreciable, especially since the teacher is quite close in terms of error to the student ($\Delta = 0.6\%$).

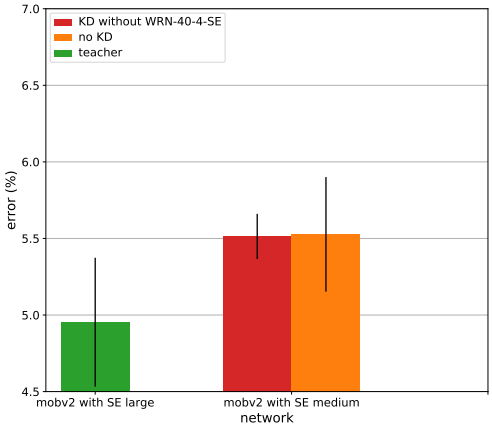
We can thus conclude that using WRN-40-4-SE is very harmful to the performances. We decided to run additional experiments to test whether Squeeze-and-Excitation blocks



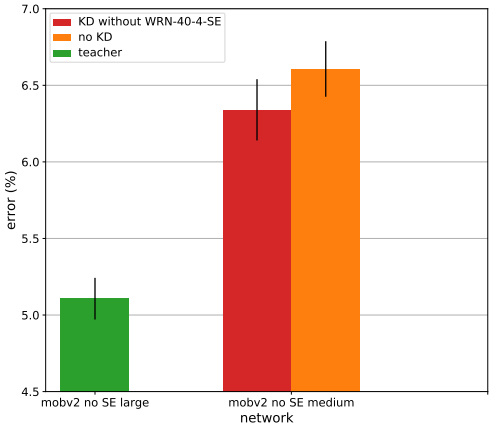

(a) KD on MobileNetv1 with SE without using WRN-40-4-SE as teacher




(b) KD on MobileNetv1 without SE without using WRN-40-4-SE as teacher




(c) KD on MobileNetv2 with SE without using WRN-40-4-SE as teacher




(d) KD on MobileNetv2 without SE without using WRN-40-4-SE as teacher

Figure 5.4: Experiments to determine the cause of the bad performances of knowledge distillation. All graphs represent the error (on the validation set) evolution when using knowledge distillation.
Displays the mean and std computed on 3 runs.

were to be incriminated or if the cause was more likely to be structural differences between teacher and student(s). We decided to try knowledge distillation from a large MobileNetv1 to a medium one, both having SE blocks (Figure 5.4b) or not (Figure 5.4a). We can see that the error decrease in both cases with the help of knowledge distillation. When using SE blocks, the error falls from 5.87 to 5.47 ($\Delta = 0.4$), when not, 6.35 to 5.79 ($\Delta = 0.56$). Both of these diminutions are appreciable (even though the standard derivations of the errors are important as well). It would be interesting to test whether the difference between both error diminutions comes from the SE blocks or from the fact that the SE networks have a lower initial error.

We repeated the same experiment with MobileNetv2 (see Figures 5.4c and 5.4d). We

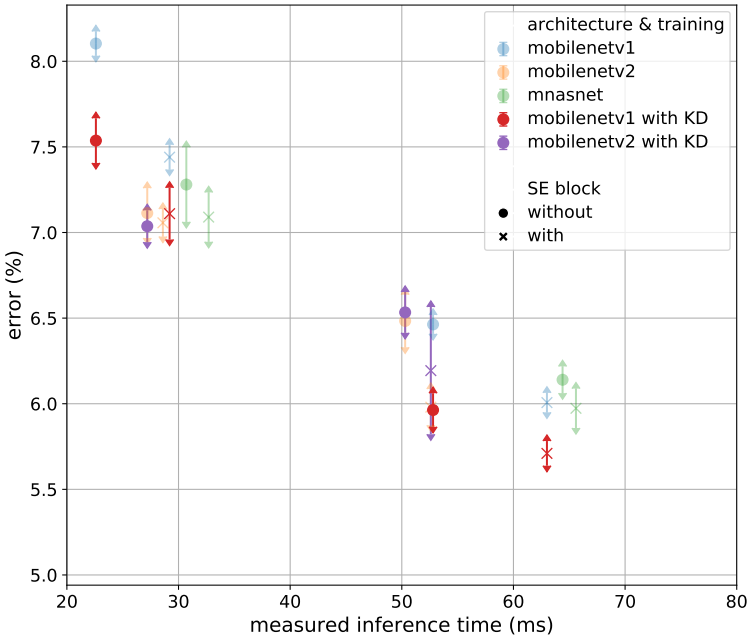



Figure 5.5: Error (on the test set) improvement when using knowledge distillation on MobileNetv1/v2
Displays the mean and std computed on 3 runs.

can see that the benefits of knowledge distillation are much less visible for this network. The error decrease is negligible ($\Delta = 0.01\%$) when using Squeeze-and-Excitation blocks. When not using those, the error goes from 6.61 to 6.34 ($\Delta = 0.27\%$), with a smaller base error, MobileNetv1 engaged benefits of 0.56% in similar settings which is much better. The reason behind that difference remains unknown and would be an interesting extension to investigate in future work.

Finally, we decided to use KD to improve MobileNetv1/v2 with/without SE blocks. This is represented on Figure 5.5. As we could expect, Knowledge distillation is uninteresting on MobileNetv2. On the other hand, MobileNetv1 + KD benefits from significant improvements and we can see that MobileNetv1 with KD are competitive with MobileNetv2 with KD (on the displayed error range). We can also notice that the inference time range displayed here is quite limited (inference times $< 80$ms). This is because, lacking a more accurate teacher, we did not perform KD on our large MobileNets, solving this issue is an interesting direction for future work.

## 5.3 Quantization

### 5.3.1 Method

Quantizing a neural network consists in reducing the number of bits used for the weights of a network (activations might be reduced as well). When quantizing the weights, the "only" benefit is model size reduction. When quantizing both weights and activations

(i.e. tensors outputted by the different layers), all the computations can be done using (for example) 8-bit integer multiplications that are faster to compute than floating-points multiplications. This results in substantial speed gains, especially if the network is run on custom hardware that is optimized to perform such operations.

Quantization can use, among others, 4 or 8 bits integers [Krishnamoorthi, 2018] or even go up to binarizing the weights and/or activations [Rastegari et al., 2016].

Quantization is, for the moment, not well integrated into common deep learning frameworks. A notable exception is Tensorflow that integrates 8-bits integer quantization [Krishnamoorthi, 2018]. Tensorflow's quantization is, however, quite experimental, as we learned while using it. For this work, we limited ourselves to the usage of Tensorflow's integrated quantization.

Tensorflow's quantization [Krishnamoorthi, 2018] consists in quantizing weights and/or activations to 4 or 8 bit integers. Without entering into details, [Krishnamoorthi, 2018]'s approach consists in turning 32-bits floating-points weights, $x$ into integers, $x_Q$ by applying the following equation:

$$x_Q = min(2^{nbr\_bits} - 1, max(0, round(\frac{x}{\Delta}) + z)) \tag{5.7}$$

where $\Delta$ and $z$ are a scaling factor and an offset. Both of these are determined so that the quantized range includes $x_{min}$, $x_{max}$ and 0 (0 is needed so that zero-padding did not generate quantization errors). The weight can then be converted back to floating point precision by using:

$$x_{float} = (x_Q - z)\Delta \tag{5.8}$$

When using floating point activations, equation 5.8 is used to compute a floating point weight back. With quantized activations, we can use the fact $z$ and $\Delta$ are common to many different weights to speed things up. Activations are quantized on a per-tensor basis (i.e. one $z$ and one $\Delta$ per tensor), while convolutions layers are quantized on a per-channel basis ($num\_out\_channels$ $z$ and $\Delta$ per convolution layer).

[Krishnamoorthi, 2018] proposes 3 different methods to perform quantization that are implemented in Tensorflow:

- **Post-training weight only quantization**: This method consists in quantizing only the weight of a pretrained network, this has the advantage to be easy to use but do not offer any gains in performances and does generate nearly the same accuracy degradation as Post-training weights and activations quantizations.

- **Post-training weights and activations quantizations**: This method consists in quantizing both weights and activations of a pretrained network, it is easy to use as well although calibration data are needed to compute the $z$ and $\Delta$ of the different tensors.

- **Weights and activations quantization aware training**: This method consists in quantizing both weights and activations at "training time" (fine tuning on an already trained network while performing quantization). It gives better accuracy than Post-training. However, it is more complex to perform.

### 5.3.2 Implementation

We tried the three approaches:

- **post-training weight only quantization**: We did manage to use this approach; however, since it does not prune the activations, no performance gain is observed. We only reduce the size of the tf-lite models (which were only on a few MB to start with) at the cost of several percents of accuracy. It is thus not useful to us.
- **Post-training weights and activations quantizations**: We did manage to compile a model using this approach by using the very last versions of Tensorflow (tf-nightly). However, inference on both the Raspberry Pi and tf-lite interpreter failed. This approach is not documented in Tensorflow. We found out about it by having a look at recent commits. It is thus not so surprising it generates bugs but was worth a try.
- **Weights and activations quantization aware training**: We did not manage to train a model using this method, at the current time, this method should be supported on Tensorflow's lower level APIs. We tried several approaches, but all failed when creating a tf-lite model.

## 5.4 Conclusion

In this chapter, we investigated two methods that were orthogonal to the previous approaches.

Knowledge distillation tries to leverage the predictions of a heavy model, the teacher, to improve the training of a smaller model, the student. We tried a recent algorithm that adds teaching assistant networks between the teacher and the student. By doing so, knowledge distillation helps pushing further the accuracies of MobileNetv1, allowing it to achieve extremely interesting results on the small and medium inference time range. We also showed that using a teacher of different architecture can prove very harmful for the performances. Finally, we think diving deeper into this method: trying other KD methods, investigating the effects of the architecture of the teacher and finding a substitute for WRN-40-2-SE, performing a more thorough hyperparameter search, and inspecting the bad knowledge distillation performances of MobileNetv2 can be worthwhile search directions for future works.

Quantization is a promising method that uses coarser representation of the weights and elements of the tensors to reduce inference time. Tensorflow's integrated quantization does not seem to be reliably integrated into the main framework now. We expect this to change in the following months with future releases.

# Chapter 6

# Final Conclusion and perspectives

## 6.1 Conclusion

The main goal of this thesis was to explore the topic of modern deep convolutional neural networks in a constraint environment. More precisely, we tackle the task of minimizing inference time on Raspberry Pi 3B while maintaining good predictive performances. Getting networks to run fast on affordable, embedded devices is paramount to the development of intelligent end devices.

To compare various methods, we relied on the CIFAR-10 dataset, sometimes referred to as the smallest hard dataset for image classification. This dataset combines a non-trivial challenge with affordable learning times. For this task, we targeted an error ranging from 8 to 5% with minimal inference times.

To do so, we investigated several architectures specifically designed for this (Chapter 3), as well as pruning (Chapter 4), knowledge distillation and quantization (Chapter 5).

Although the latter approach seems promising, it is not well integrated in popular frameworks as of today. Knowledge distillation did show interesting results on MobileNetv1 and is a promising direction for a deeper inspection.

In chapter 4, we propose several extensions to existing pruning methods and use pruning as an architecture search procedure. We noticed that formulating pruning as an architecture search problem systematically improves the accuracy of the resulting network. The discovered architectures are also interpretable and different between the modified methods. On the other hand, these methods all give very similar results which are not outperforming a grid search baseline. We suspect this might not (less) be the case on a dataset with bigger image spatial resolutions.

In Chapter 3, we review the state-of-the-art architectures for constrained environments and propose to combine them with Squeeze-and-Excitation blocks. When using those, MobileNetv2 seems to be the best performing architecture, followed closely by MnasNet and MobileNetv1. When adding the effect of knowledge distillation, MobileNetv1 shows very interesting results for small to medium inference times as well.

## 6.2 Perspectives

This thesis raises many more questions than it answered. First is the question of how the conclusions of this work can be extrapolated in other contexts. Indeed, changing the dataset may change which networks perform the best in terms of accuracy for a given inference time. On the other hand, changing the inference device might impact the

inference time of each model differently. In another context, using pruning algorithms, like NetAdapt, might perform well on architecture search.

The question of quantization remains of great interest and exploring it further in several months might show new interesting developments. We also think that exploring knowledge distillation further, integrating other algorithms and performing a more throughout hyperparameter search could be worthwhile. The two approaches could also be combined [Mishra and Marr, 2018].

Implementing some architecture ourselves (ShuffleNetv2, for instance) instead of using tf-lite would allow us to include custom optimizations, such as tensor concatenation on the channel axis.

Additionally, during the last week of this thesis, we discovered two papers whose contributions could be used to improve this work:

1. [Tan and Le, 2019]: In this paper, the authors introduced a simple and efficient way to scale simultaneously the depth, width and resolution[1] of a given network. The authors showed that scaling the width, depth and resolution of a network by $\alpha^\phi$, $\beta^\phi$ and $\gamma^\phi$ respectively gave much better results than only scaling one of these at a time. $\phi$ is a factor depending on the performance we are willing to sacrifice in the exchange of better accuracies. Given an initial network, $\alpha$, $\beta$ and $\gamma$ are determined by performing a grid search to find the best performing network under the constraint $\alpha \times \beta^2 \times \gamma^2 \approx 2$. $\beta$ and $\gamma$ are here squared because the number of FLOPS scales quadratically with width and resolution.

   [Tan and Le, 2019] scaled their networks from medium-sized to big sized network, achieving state-of-the-art results on ImageNet at a much smaller cost than other approaches. It would be interesting to see how their approach performs in the small to medium size scale with respect to our manual hyperparameter fine-tuning.

2. [Howard et al., 2019]: In this paper, MobileNetv3 was introduced. In addition to comparing the architecture, they developed with the other ones that were investigated in chapter 3 and other improvements they made. We noticed that, in parallel to this work, they also used Squeeze-and-Excitation blocks and the NetAdapt algorithm to prune pre-trained architectures. Like us, they also modified the NetAdapt algorithm to retrain the final architecture from scratch and to use the ratio $\frac{\Delta acc}{\Delta latency}$ instead of only $\Delta acc$. This seems to show that although using NetAdapt to search for new architectures did not show any good results on WideResNets on CIFAR-10 while using a Raspberry Pi 3B to perform inference, this might not be the case for MnasNet-like architectures on ImageNet while performing inference on Google Pixels.

[1] we did not mention input resolution here because it is not used on $32 \times 32$ CIFAR-10 but networks trained on ImageNet often vary the height/width of their inputs to trade accuracy for performance

# Bibliography


[Belkin et al., 2018] Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2018). Reconciling modern machine learning and the bias-variance trade-off. *arXiv:1812.11118 [cs, stat]*. arXiv: 1812.11118.

[Canziani et al., 2016] Canziani, A., Paszke, A., and Culurciello, E. (2016). An Analysis of Deep Neural Network Models for Practical Applications. *arXiv:1605.07678 [cs]*. arXiv: 1605.07678.

[Chollet, 2017] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807.

[Crowley et al., 2018] Crowley, E. J., Turner, J., Storkey, A., and O'Boyle, M. (2018). Pruning neural networks: is it time to nip it in the bud? *arXiv:1810.04622 [cs, stat]*. arXiv: 1810.04622.

[Frankle and Carbin, 2019] Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR 2019*.

[Freeman et al., 2018] Freeman, I., Roese-Koerner, L., and Kummert, A. (2018). Effnet: An efficient structure for convolutional neural networks. *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 6–10.

[Gholami et al., 2018] Gholami, A., Kwon, K., Wu, B., Tai, Z., Yue, X., Jin, P., Zhao, S., and Keutzer, K. (2018). Squeezenext: Hardware-aware neural network design. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1638–1647.

[Gordon et al., 2018] Gordon, A. D., Eban, E., Nachum, O., Chen, B., Yang, T.-J., and Choi, E. (2018). Morphnet: Fast & simple resource-constrained structure learning of deep networks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1586–1595.

[Hassibi and Stork, 1993] Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal Brain Surgeon. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. Morgan-Kaufmann.

[He et al., 2016a] He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

[He et al., 2016b] He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity mappings in deep residual networks. In *ECCV*.

[Hinton et al., 2015] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[Howard et al., 2019] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. (2019). Searching for MobileNetV3. *arXiv:1905.02244 [cs]*. arXiv: 1905.02244.

[Howard et al., 2017] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861 [cs]*. arXiv: 1704.04861.

[Hu et al., 2017] Hu, H., Dey, D., Del Giorno, A., Hebert, M., and Bagnell, J. A. (2017). Log-DenseNet: How to Sparsify a DenseNet. *arXiv:1711.00002 [cs]*. arXiv: 1711.00002.

[Hu et al., 2018] Hu, J., Shen, L., and Sun, G. (2018). Squeeze-and-excitation networks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7132–7141.

[Huang et al., 2018] Huang, G., Liu, S., van der Maaten, L., and Weinberger, K. Q. (2018). Condensenet: An efficient densenet using learned group convolutions. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2752–2761.

[Huang et al., 2017] Huang, G., Liu, Z., and Weinberger, K. Q. (2017). Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269.

[Iandola et al., 2016] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360 [cs]*. arXiv: 1602.07360.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*.

[Krishnamoorthi, 2018] Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv:1806.08342 [cs, stat]*. arXiv: 1806.08342.

[Krizhevsky, 2009] Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. *Technical Report*, page 60.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60:84–90.

[Larsson et al., 2016] Larsson, G., Maire, M., and Shakhnarovich, G. (2016). FractalNet: Ultra-Deep Neural Networks without Residuals. *arXiv:1605.07648 [cs]*. arXiv: 1605.07648.

[Lee et al., 2018] Lee, N., Ajanthan, T., and Torr, P. H. S. (2018). SNIP: Single-shot Network Pruning based on Connection Sensitivity. *arXiv:1810.02340 [cs]*. arXiv: 1810.02340.

[Lin et al., 2014] Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. (2014). Microsoft COCO: Common Objects in Context. *arXiv:1405.0312 [cs]*. arXiv: 1405.0312.

[Liu et al., 2017] Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. (2017). Learning efficient convolutional networks through network slimming. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2755–2763.

[Liu et al., 2018] Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. (2018). Rethinking the Value of Network Pruning. *arXiv:1810.05270 [cs, stat]*. arXiv: 1810.05270.

[Loshchilov and Hutter, 2017] Loshchilov, I. and Hutter, F. (2017). Sgdr: Stochastic gradient descent with warm restarts. In *ICLR*.

[Ma et al., 2018] Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*.

[Mirzadeh et al., 2019] Mirzadeh, S.-I., Farajtabar, M., Li, A., and Ghasemzadeh, H. (2019). Improved Knowledge Distillation via Teacher Assistant: Bridging the Gap Between Student and Teacher. *arXiv:1902.03393 [cs, stat]*. arXiv: 1902.03393.

[Mishra and Marr, 2018] Mishra, A. and Marr, D. (2018). Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *International Conference on Learning Representations*.

[Molchanov et al., 2017] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2017). Pruning convolutional neural networks for resource efficient inference. In *ICLR*.

[Pleiss et al., 2017] Pleiss, G., Chen, D., Huang, G., Li, T., van der Maaten, L., and Weinberger, K. Q. (2017). Memory-Efficient Implementation of DenseNets. *arXiv:1707.06990 [cs]*. arXiv: 1707.06990.

[Rastegari et al., 2016] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer.

[Russakovsky et al., 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Bernstein, M. S., Fei-Fei, L., Berg, A. C., and Khosla, A. (2015). Imagenet large scale visual recognition challenge. *Springer US*.

[Sandler et al., 2018] Sandler, M. B., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520.

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

[Szegedy et al., 2016a] Szegedy, C., Ioffe, S., and Vanhoucke, V. (2016a). Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9.

[Szegedy et al., 2016b] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016b). Rethinking the inception architecture for computer vision. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826.

[Tan et al., 2018] Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. (2018). MnasNet: Platform-Aware Neural Architecture Search for Mobile. *arXiv:1807.11626 [cs]*. arXiv: 1807.11626.

[Tan and Le, 2019] Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114.

[Theis et al., 2018] Theis, L., Korshunova, I., Tejani, A., and Huszár, F. (2018). Faster gaze prediction with dense networks and Fisher pruning. *arXiv:1801.05787v2*.

[Turner et al., 2018] Turner, J., Cano, J., Radu, V., Crowley, E. J., O'Boyle, M., and Storkey, A. (2018). Characterising across-stack optimisations for deep convolutional neural networks. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 101–110. IEEE.

[Wong, 2018] Wong, A. (2018). NetScore: Towards Universal Metrics for Large-scale Performance Analysis of Deep Neural Networks for Practical On-Device Edge Usage. *arXiv:1806.05512 [cs, stat]*. arXiv: 1806.05512.

[Yang et al., 2018] Yang, T.-J., Howard, A. G., Chen, B., Zhang, X., Go, A., Sze, V., and Adam, H. (2018). Netadapt: Platform-aware neural network adaptation for mobile applications. In *ECCV*.

[Yu et al., 2018] Yu, R., Li, A., Chen, C.-F., Lai, J.-H., Morariu, V. I., Han, X., Gao, M., Lin, C.-Y., and Davis, L. S. (2018). Nisp: Pruning networks using neuron importance score propagation. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9194–9203.

[Zagoruyko and Komodakis, 2016] Zagoruyko, S. and Komodakis, N. (2016). Wide Residual Networks. *arXiv:1605.07146 [cs]*. arXiv: 1605.07146.

[Zhang et al., 2018] Zhang, X., Zhou, X., Lin, M., and Sun, J. (2018). Shufflenet: An extremely efficient convolutional neural network for mobile devices. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856.

[Zoph and Le, 2016] Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

[Zoph et al., 2018] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8697–8710.