POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Politecnico
di Torino

Master's Degree Thesis

DEPTHWISE ADDERNET: Energy-efficient deep neural networks for edge devices

Supervisors

Prof. Luciano LAVAGNO

Prof. Mihai LAZARESCU

Candidate

Teodoro URSO

APRIL 2022

Abstract

Today machine learning techniques and, in particular, deep neural networks are widely used for multiple tasks. Convolutional neural networks improve the performance of artificial intelligence algorithms in many applications (e.g. image recognition, object detection, natural language processing); but tend to be energy-intensive due to the large amount of multiplications involved.
This type of model is unsuitable for IoT devices or edge devices (such as smartphones) which are limited in terms of memory and computational capacity.
For this reason, many studies are focusing on developing more efficient deep network architectures.

This thesis analyses two modern techniques: depthwise separable convolution and Addernet. The first allows designing deep networks with fewer parameters while the second reduces the usage of hardware resources and decreases the computational and energy cost by substituting multiplications with additions.

In particular, a new layer is presented which combines the strengths of the two models. It is compatible with many networks in use today and it presents opportunities for further enhancements.

Training and testing are performed on a GPU using the PyTorch framework. The DNN architectures employing the proposed layer present a reduction up to 75% in the parameters memory occupation. Moreover, when implementing the models on a dedicated hardware platform (e.g., FPGA) an improvement in terms of computational resource utilization and power consumption can be expected.

Results obtained on different networks against the CIFAR-10 and CIFAR-100 datasets show the feasibility and potential of this new "depthwise adder kernel".

Ai miei nonni Anna Maria, Rocco e Rosa

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

BN
batch normalization

CNN
convolutional neural network

DL
Deep Learning

DNN
Deep Neural Network

DS-CNN
Depthwise separable convolutional neural network

DS-conv
Depthwise separable convolution

DW-conv
Depthwise convolution

FPGA
Field programmable gate array

GPU
graphics processing unit

IoT

internet of things

MAC

Multiply and accumulate

ML

machine learining

PW-conv

Pointwise convolution

ReLU

Rectified linear unit

SM

Streaming multiprocessor

Chapter 1

Introduction

1.1 Machine learning and deep learning

Artificial intelligence (AI) [1] is a computer science field in which a machine (i.e. a computer) is instructed to perform tasks that that maximize its chance of achieving its goals. The advancements in this field have brought enormous progress in recent years in a broad field of applications, such as image recognition, object detection, language understanding and problem solving.

Machine Learning (ML) [1] is one of the methods for performing AI tasks. Depending on the interpretation, it can be considered as subset of artificial intelligence or a separate field. It is based on the ability of machines to process data and learn autonomously, modifying variables and algorithms based on the information they receive.
ML can be divided in three main categories [2]:

- Supervised learning : the computer system is trained to autonomously predict the output value, which can be a continuos (regression) or discrete (classification) one, corresponding to a certain input. The training phase is based on a set of examples, consisting of input and output pairs, which are provided to instruct the machine about which output has to be assigned to a certain input.

- Unsupervised learning : the machine does not predict an output value; instead, it searches for common patterns and similarities in the data in order to organise them in categories/groups.

- Reinforcement learning : this approach deals with sequential decision problems, where the action to be taken depends on the current state of the system and determines its future state. The algorithm aims at creating autonomous agents capable of choosing actions to be performed in order to achieve certain

objectives through interaction with their environment. The quality of an action is associated to a numerical value of "reward", inspired by the concept of reinforcement, which encourages the correct behaviour of the agent.

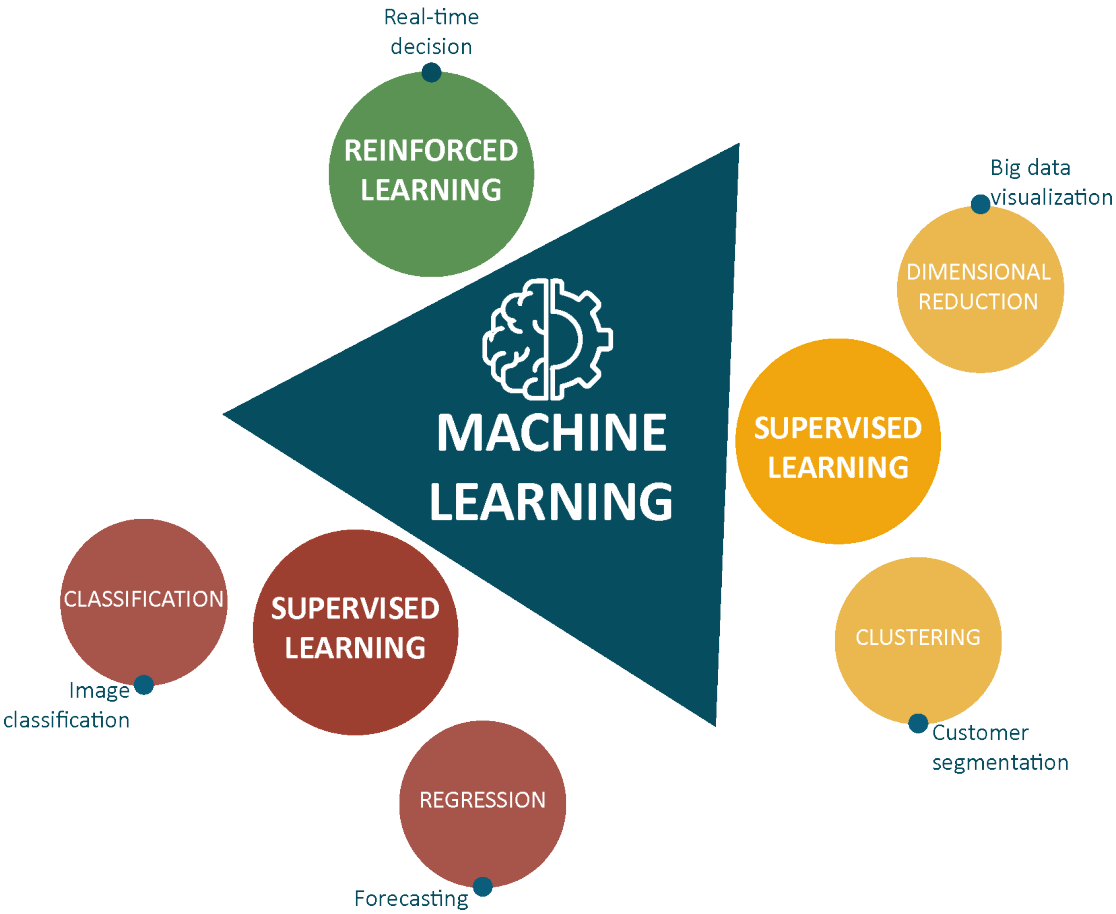

Figure 1.1: Machine learning approaches.

Other approaches [3] have been developed that do not fit into the above list, and sometimes more than one can be used by a machine learning system.
When machine learning (whether supervised or unsupervised) is performed using multi-layered structures, it is called deep learning (DL) [1], which is characterised by the fact that the high-level features of the inputs are autonomously extracted by the machine itself, instead of being hand-crafted by humans as in conventional ML algorithms. Other approaches are clustering, logic programming and Bayesian networks [1].

1.2 Artificial Neural Networks

An artificial neural network (ANN) is a mathematical model inspired by the human brains neurons structure. In particular, in feedforward neural networks, each neurons performs an algebric operation on its input x_i, which equation is reported in (1.1).

$$y_i = w_i \cdot x_i + b_i \tag{1.1}$$

w_i represents the weight of the i-th neuron, while b_i its bias. One can notice that the output y_i is a linear function of the input x_i.

An ANN is a collection of neurons, that are organized in layers, i.e. groups of neurons that are interconnected in a sequence-like structure, as shown in Figure 1.2. It can be noticed that the neurons of two layers are all inteconnected to each other.

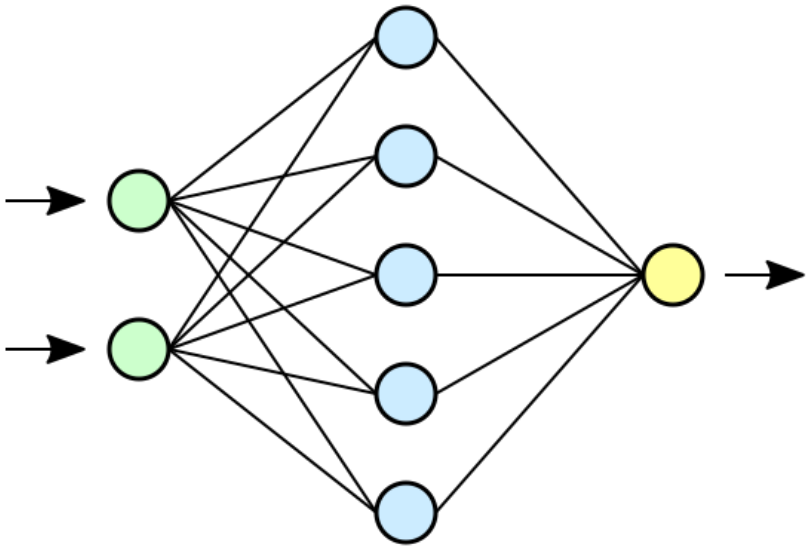

Figure 1.2: Schematic structure of neural network.[4]

Considering a neuron with multiple inputs, (1.1) can be rewritten in the following way:

$$y_i = \sum_{j=1}^{n} w_{ij} \cdot x_j + b_i \tag{1.2}$$

n is the number of neurons which outputs are connected to the i-th neuron under consideration. One can derive a matrix-like description of the network from (1.2),

in which each layer is described by a matrix of weights, $\boldsymbol{W_l}$, and a vector of biases, $\boldsymbol{b_l}$. In particular, given the l-th layer, one can write:

$$\boldsymbol{y_l} = \boldsymbol{W_l} \cdot \boldsymbol{x_l} + \boldsymbol{b_l}$$

An ANN can be divided in three sections: the input layer, the hidden layer (i.e. the intermediate one) and the output layer. When multiple hidden layers are used, the model is called Deep Neural Network (DNN).

1.2.1 Activation layer

The main purpose of an activation function is to introduce non-linearity into the model and thus increase the ability to handle more complex patterns.
A good activation function should be zero centered (to avoid that the gradient tends towards a particular direction during backpropagation phase). It should be differentiable at least in parts, computationally cheap and should not shift the gradient towards zero (vanishing gradient problem).

Relu and Relu6

Rectified Linear Unit(ReLU) is one of the most commonly used activation functions, especially in convolutional networks. It is defined as:

$$ReLU(x) = max(0, x) \tag{1.3}$$

The advantages of this function are that it is not affected by the vanishing gradient and it requires low computational effort.
The main disadvantage is that it is non-zero centered and also assumes null value for negative input values. This causes "dying ReLU" problem so that some nodes in the network do not learn anything.
Another drawback of ReLUs is the lack of an upper limit, which leads to some output values "exploding". A solution to the gradient explosion (and other minor ReLU problems) is to use ReLU 6 defined as follows:

$$ReLU6(x) = min(max(0, x),6) \tag{1.4}$$

Hence, for the single neuron, (1.2) can be modified as follows:

$$y_i = g(\sum_{j=1}^{n} w_{ij} \cdot x_j + b_i) \tag{1.5}$$

where $g(x)$ is the non linear function employed.

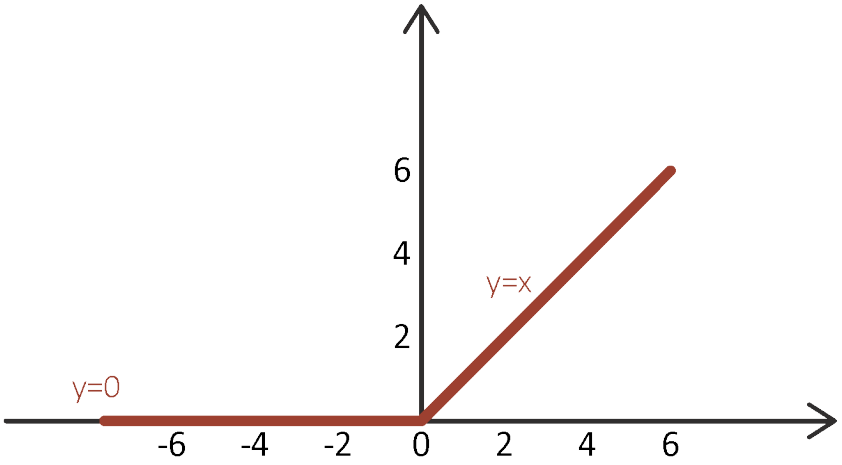

Figure 1.3: ReLU.

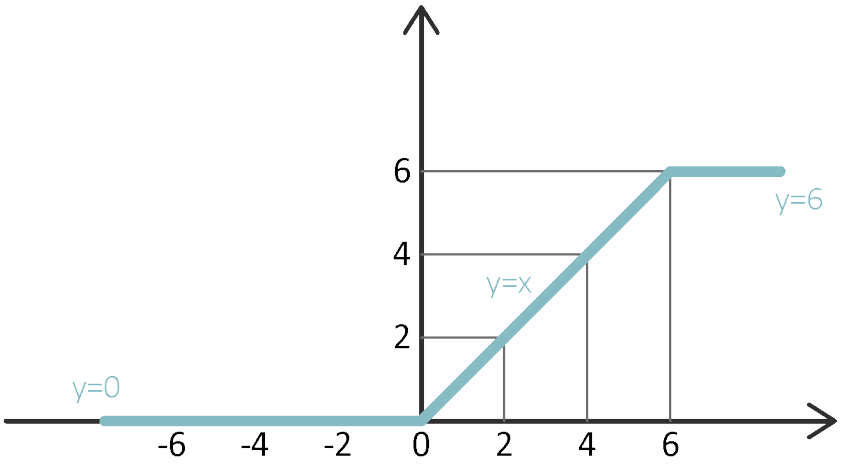

Figure 1.4: ReLU6.

1.2.2 How a neural network learns

Consider a classification problem. A training set is provided in order to "teach" the network to correctly assign a set of inputs, $\boldsymbol{x}$, to a set of labels, $\boldsymbol{y}$. To achieve this, the set of inputs is provided to the network.
The following naive equation can be written:

$$\hat{\boldsymbol{y}} = NN(\boldsymbol{x})$$

where $\hat{\boldsymbol{y}}$ is the output provided by the network.
The training process consists in updating the weights and biases of the network

through multiple iterations across the training set so that $\hat{\boldsymbol{y}} \approx \boldsymbol{y}$. In order to achieve this, a metric has to be defined to measure the "distance" between the provided output, $\hat{\boldsymbol{y}}$, and the desired one, $\boldsymbol{y}$. This function is referred to as the loss function [1].
Depending on the application, different loss functions can be used. Usually, for regression the mean-squared error (MSE) is used, while, in classification tasks, the cross entropy is employed [1], which formula for a binary classification task is reported in (1.6) :

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(1-\hat{y}_i) + (1-y_i) \cdot log(1-\hat{y}_i) \tag{1.6}$$

where N represent the training set size.

Once the loss function is chosen, a way to update the weights of the networks so that the loss (also called cost function) is minimised, has to be found. In neural networks, the gradient descent [1] algorithm is employed: the backward derivatives of each layer with respect to the loss value are computed, and their values are used to update the weights. This operation is carried out by an optimization algorithm called optimizer.
The most commonly used optimization algorithms (e.g. SGD and Adam [1]) require the calculation of the gradient with respect to the weights in order to find the minimum of the cost function.
The gradient represents the variation of the output with respect to the variation of the single weight and consequently establishes the intensity of its contribution to the loss function. These gradients are calculated through an algorithm called back-propagation which involves the recursive application of the chain rule from calculus starting from the output layer and proceeding backwards.
The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known.
This process is usually performed several times (epochs) on the entire dataset.
In algorithm 1, an example of optimizer, Adam, is reported.

Once the network has been trained, a metric has to be defined to evaluate the network performance on a certain task. For classification, the accuracy metric [1] is usually employed, which can be computed as the ratio between the correctly classified samples and the dataset size.

Algorithm 1 Adam optimizer algorithm. All operations are element-wise, even powers. Good values for the constants are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. ϵ is needed to guarantee numerical stability[5].

procedure ADAM($\alpha, \beta_1, \beta_2, f, \theta_0$)
  ▷ α is the stepsize
  ▷ $\beta_1, \beta_2 \in [0, 1)$ are the exponential decay rates for the moment estimates
  ▷ $f(\theta)$ is the objective function to optimize
  ▷ θ_0 is the initial vector of parameters which will be optimized
  ▷ Initialization
  $m_0 \leftarrow 0$ ▷ First moment estimate vector set to 0
  $v_0 \leftarrow 0$ ▷ Second moment estimate vector set to 0
  $t \leftarrow 0$ ▷ Timestep set to 0
  ▷ Execution
  while θ_t not converged **do**
    $t \leftarrow t + 1$ ▷ Update timestep
    ▷ Gradients are computed w.r.t the parameters to optimize
    ▷ using the value of the objective function
    ▷ at the previous timestep
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
    ▷ Update of first-moment and second-moment estimates using
    ▷ previous value and new gradients, biased
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    ▷ Bias-correction of estimates
    $\hat{m}_t \leftarrow \dfrac{m_t}{1 - \beta_1^t}$
    $\hat{v}_t \leftarrow \dfrac{v_t}{1 - \beta_2^t}$
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \dfrac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ ▷ Update parameters
  end while
  return θ_t ▷ Optimized parameters are returned
end procedure

1.3 Convolutional Neural Networks

Convolutional neural networks (CNN or ConvNet) are one of the most widely used classes of network, especially for image and video recognition.
The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution [1].
Usually the convolution operation is performed by the hidden layers.
In Figure 1.5, a description of the most common layers in a CNN is reported.

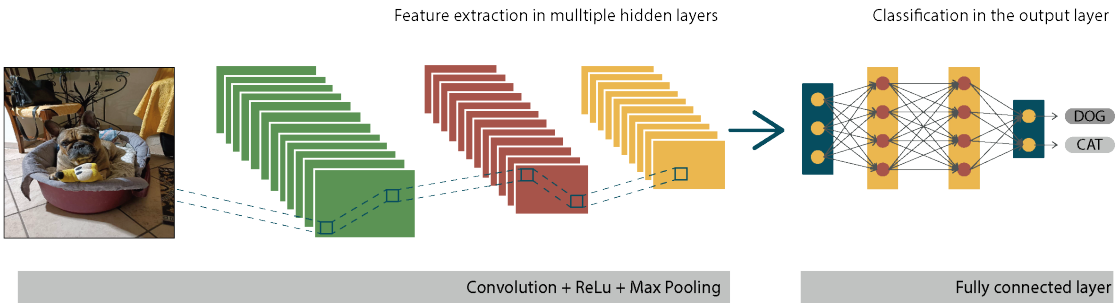

Figure 1.5: Typical structure of a convolutional neural network.

1.3.1 Convolutional layer

The core of CNNs is the convolutional layer. This type of layer is made up of filters that perform convolutions with the aim of extracting and identifying features from the input data. In the case of images, can be identified straight lines, particular shapes or colours.
In mathematics, in particular in functional analysis, convolution is an operation between two functions of one variable that consists of integrating the product between the first and the second after the latter has been translated by a certain value.
Consequently, it is a special type of integral transformation defined as:

$$(f * g)(t) := \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{1.7}$$

This operation could be performed between two matrices. For example, considering A and B both of size 3x3:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

In this case the convolution between A and B will be given by :

$$A * B = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} b_{ij}$$

Thus, each element in A is multiplied by the corresponding element in B and then the output value is calculated as the sum of all multiplications.
In image processing, the convolution is performed between an input matrix consisting of the image and a filter also called kernel. The latter is usually small and of odd size (1x1,3x3,5x5) because it is important to identify the centre of the kernel matrix.
Usually, the input image would be bigger than the filter. In this case the output matrix is computed gliding through the input and performing the scalar product between a sub-matrix with the same dimensions of the kernel (called sliding window) and the filter (Figure 1.6).

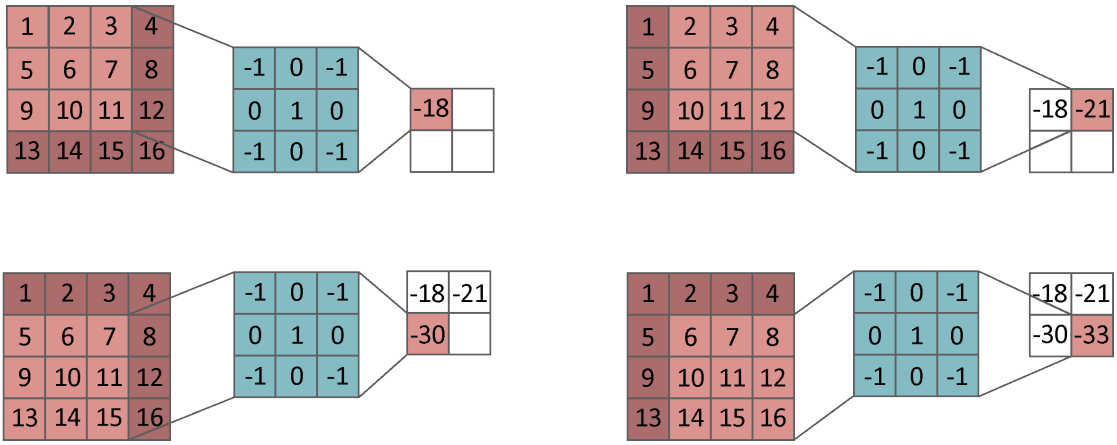

Figure 1.6: Example of 2D Convolution.

Stride and padding

In Figure 1.6, a convolution is performed between a matrix of size 4×4 and a 3×3 kernel, obtaining as output a 2×2 matrix. In general, supposing that the dimensions of the input are n_h and n_w and the kernel has dimensions k_h and k_w, the output size can be expressed as $(n_h - k_h + 1) \times (n_w - k_w + 1)$.
A convolution operation leads to a reduction in the image size from one layer to the other; as a consequence, information is lost through the network; in particular, the pixels on the edges are removed.
To overcome this problem, a commonly used technique is to surround the input matrix with a frame of extra pixels (usually zeros) as shown in the Figure 1.7.

This approach is called padding.
Considering the insertion of p_h rows (usually half on top and half on bottom) and p_w columns (usually half on the left and half on the right) the output shape can be easily derived as:

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$

Padded convolutions can be useful to preserve the shape of the input after the operation ($p_h = k_h - 1$ and $p_w = k_w - 1$) bringing some benefits in calculations and model performance.

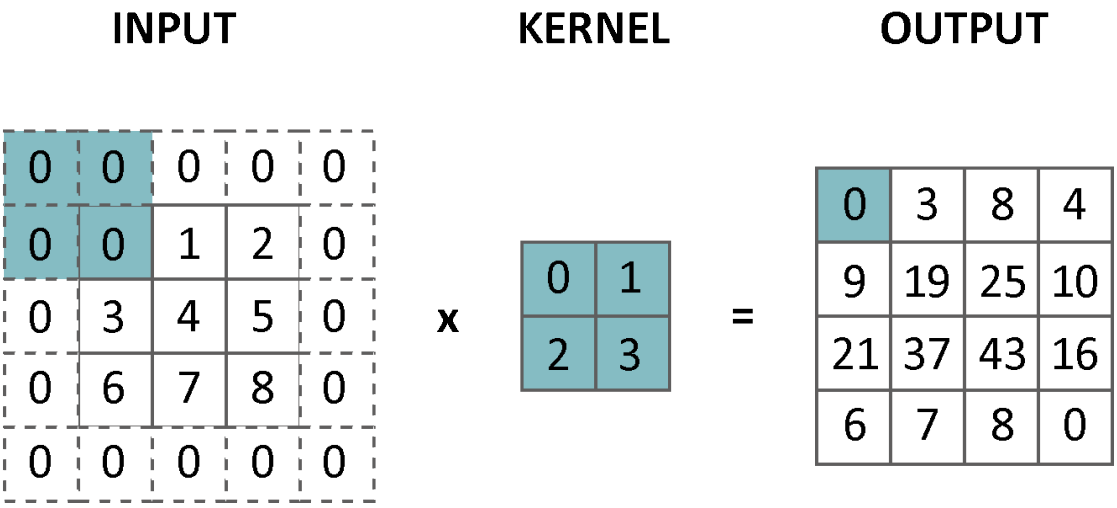

Figure 1.7: Padding.

For computational efficiency reasons or to shrink the sample, it may be necessary to reduce the feature matrix size from one layer to the other. Usually the convolutions are carried out shifting the input sliding window of 1 pixel (to the right or bottom). By increasing the displacement factor called "stride", the strided convolution is carried out.
In these cases, assuming a vertical stride equal to s_h and a horizontal stride equal to s_w, the output shape will be:

$$\frac{(n_h - k_h + p_h + s_h)}{s_h} \times \frac{(n_w - k_w + p_w + s_w)}{s_w}$$

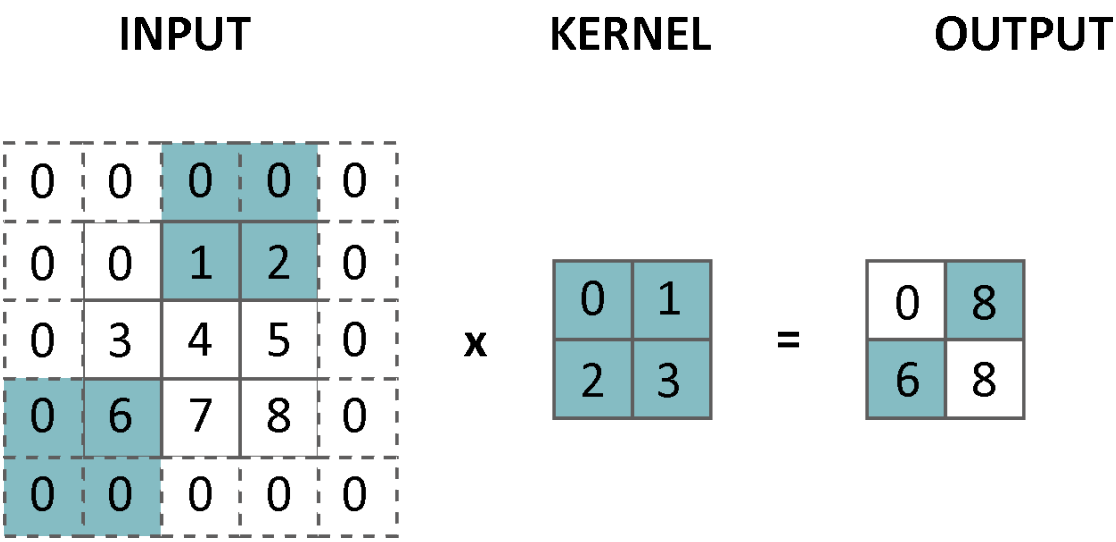

Figure 1.8: Stride example with $s_w = 2$ and $s_h = 3$.

1.3.2 Batch Normalization layer

In CNNs, data are often processed in groups called "batches".
Batch normalization (BN) is a technique first introduced in 2015 [6], which allows to make the training phase of a deep neural network faster and more stable.
BN can be inserted immediately before or after the activation function and in AI frameworks this method is often used as a standard layer.
For each hidden layer the mean value μ_B (1.8) and the variance σ_B^2 (1.9) of the activation value on the batch are computed.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad (1.8) \qquad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \qquad (1.9)$$

Then the input is normalized

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}. \qquad (1.10)$$

The parameter ϵ is used to prevent division by zero if σ_B becomes null during the training phase.
Finally the output of the layer is calculated using two trainable parameters (γ and β) to perform a linear transformation:

$$y_i = \gamma\hat{x_i} + \beta \qquad (1.11)$$

In each step the model calculates μ and σ^2 and trains γ and β using the gradient descent. During the evaluation phase the mean value and the variance are computed using the values obtained during the training phase.
The reason why this technique is so efficient in many models is still a matter of debate and more recent studies [7] have formulated different hypotheses from the first paper.

1.3.3 Pooling layer

The pooling layer consists of a filter that reduces the size of the feature map allowing the reduction of the number of operations, trainable parameters and complexity. It is also helpful to summarise the information of a given input and make the network more robust to changes.
There are essentially two methods:

- Average pooling - The average of the input features covered by the filter is calculated.

- Max pooling - The output corresponds to the highest values among the regions of the input feature map.

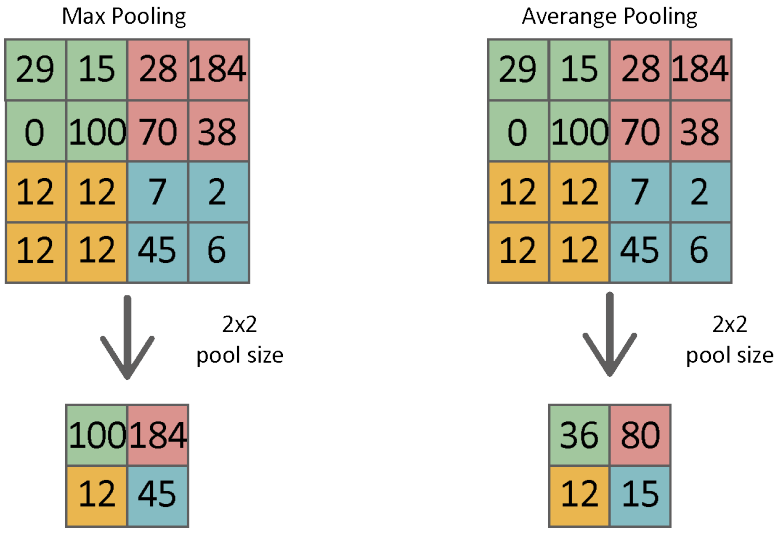

Figure 1.9: Pooling methods.

1.3.4 Fully connected layer

The output of the last pooling layer (or convolutional) layer is usually flattened into a 1D vector. This is followed by a fully connected (FC) layer, i.e. a feedforward network, in which each input is associated to each output through a trainable weight.
In classification problems usually the last fully connected layer has a number of outputs equal to the classes to be identified.

1.4 Neural Networks on edge devices

With the development of modern Graphics Processing Units (GPUs), it has been possible to use deeper and deeper neural networks with more parameters and to reduce training and inference time.
However, GPUs have a high power consumption that makes them unusable on edge devices such as mobile phones or cameras. These types of devices have low memory and computational capacities, which makes them unsuitable for deep neural network models. Consequently, many models still depend on cloud computing, which for some applications presents problems of weak privacy, long latency and low reliability. Moreover, on-device computing is essential for some IoT applications that demand real-time data.
In order to meet these needs, studies aimed at analysing on-device computing focus mainly on three fields:

- Pruning : in deep learning this technique is used to develop more efficient deep learning models. DNNs have redundancy, pruning allows the elimination of "neurons" or links between them, reducing the memory occupation and speeding up the training of the model.
  The usage of pruning depends on the application and methods used. Sometimes fine-tuning or multiple iterations of pruning may be unnecessary, based on how much the network is pruned.

- Quantization : this approach focuses on optimizing the representation of data and weights. It is based on the trade off between occupied memory and loss of accuracy of the metrics.
  By using fewer bits, the operations are less computationally demanding but the resolution of the inputs and outputs deteriorates, degrading performance. Different types of quantization have advantages and disadvantages that often depend on the type of model considered.

- Design of model architecture : this field explores possible alternative backbones or layer structures that allow the development of efficient models.

The goal of this thesis is to combine two techniques used to reduce the memory occupation of the parameters and the computational cost of the operations performed by implementing a new type of layer that can replace the classical convolutional layers. The above mentioned techniques are respectively the depthwise separable convolutions (Chapter 2) and the addernet (Chapter 3).

Chapter 2

Depthwise Separable Convolutions

Depthwise separable convolutions were first introduced in a PhD thesis [8] in 2014 and have subsequently been used in DNNs such as MobileNet [9] and Xception [10].
This technique makes possible a considerable reduction in the number of parameters and multiplications compared to a classical convolution with an acceptable loss of accuracy (or other metrics).
This is why it has attracted a lot of interest in recent years in order to optimize the architecture of CNNs and make them more "lightweight".
The origin of this approach can be traced back to Spatial Separable Convolutions which consist of dividing a 2D kernel ($w \times h$) into two one-dimensional kernels. The classic example is the Sobel filter used for edge-detection [11].

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

Figure 2.1: Sobel kernel.

Using this method, instead of performing a convolution with a 3×3 kernel (i.e. 9 multiplications for each output pixel), 2 kernels are used, each performing 3 multiplications (6 multiplications for each output pixel), getting the same result. Unfortunately, not all kernels can be divided in this way and consequently this method is not widely used in deep learning.

Depthwise Separable Convolutions (DS-convs) also apply to kernels that are not separable into two smaller ones and owe their name to the fact that they act not

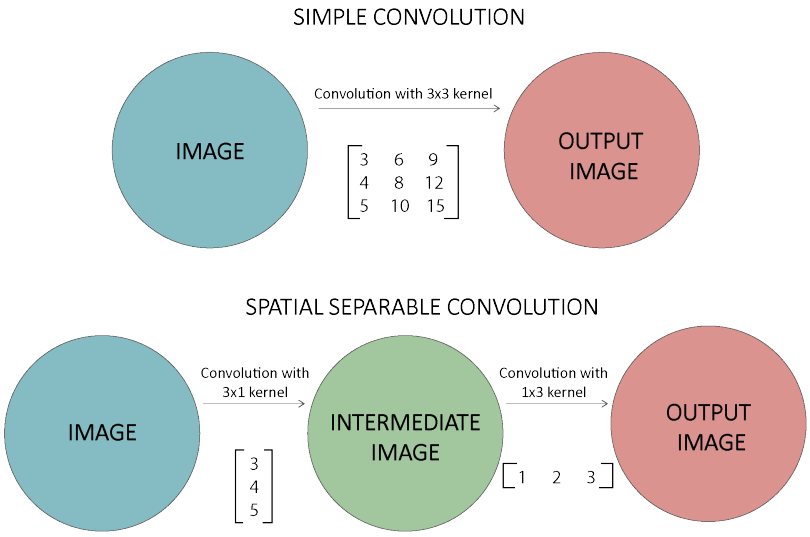

Figure 2.2: Spacial Separable Convolution.

only on the spatial dimension but also on depth.
A colour image has several channels (e.g. RGB), i.e. it is represented by several matrices, each showing the colour intensity value of the pixels. When layer operations are performed, the number of channels in the feature map may vary (e.g. according to the number of filters used).
When DS-conv are used, the classical convolution is replaced by two stages of convolutions, the first acting individually on each channel (depthwise) and the second intersects the information along the depth dimension (pointwise).

2.1 Depthwise Convolution

In DW-conv, a single $C_{in} \times K \times K$ filter is used (usually K is equal to 3 or 5), with a number of channels C_{in} equal to that of the input. On each channel of the input feature map a 2D convolution is performed with the corresponding channel of the filter. This generates an output with same number of channels of the input.
Consider a feature map of size $C_{in} \times D \times D$ (here $W = H = D$ for simplicity) and padding and stride equal to 1; the computational cost in terms of number of multiplications (additions are considered negligible) can be derived:

$$DW_{op} = C_{in} \times D \times D \times K \times K \tag{2.1}$$

This operation makes it possible to search for patterns in the channels considered separately but, unlike convolution, it does not intersect the information between them.

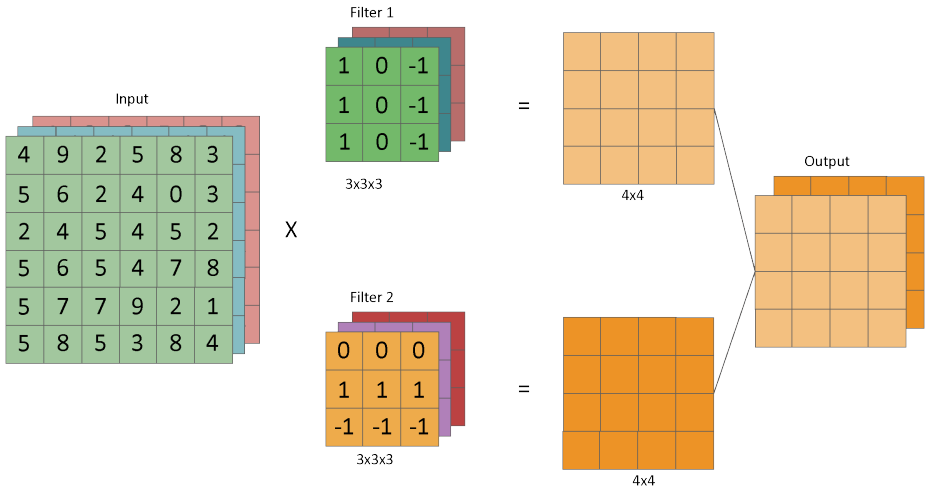

Figure 2.3: Multi-channel convolution.

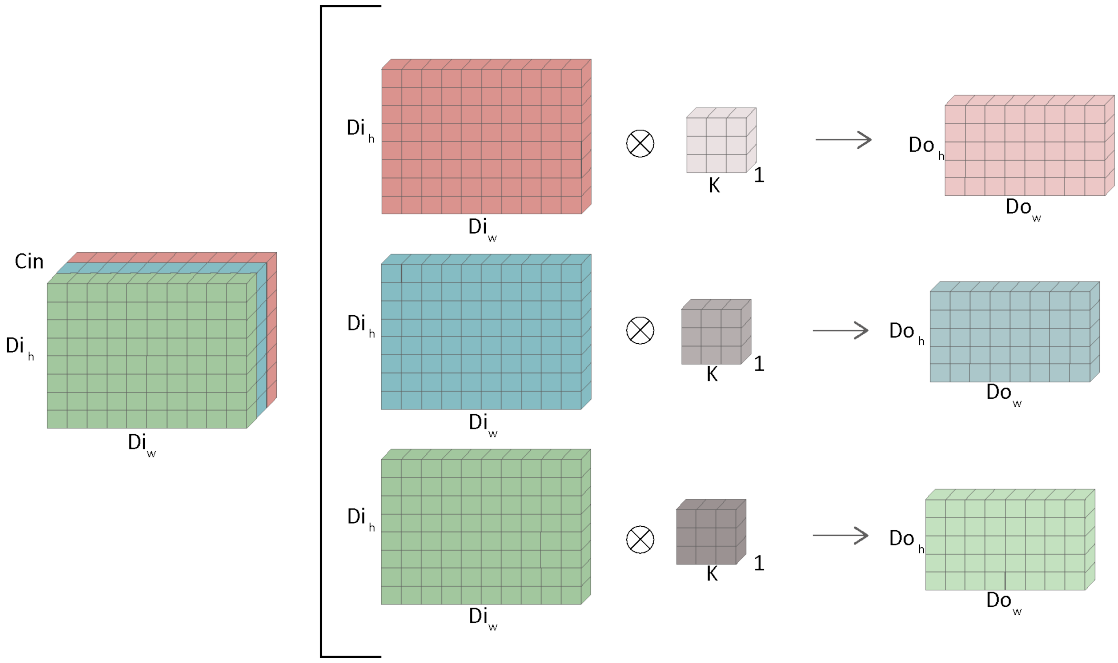

Figure 2.4: Depthwise Convolution.

2.2 Pointwise Convolution

A convolution is called pointwise when it is performed using 1x1 filters.
In the case of the DS-conv this takes the output of the DW-conv as input and preserves the width and depth dimensions by performing a linear combination

along the channels.
The number of filters (of dimension $C_{in} \times 1 \times 1$) varies depending on the number of desired output channels C_{out}.
In this case the number of multiplications performed is given by:

$$PW_{op} = C_{in} \times C_{out} \times D \times D \tag{2.2}$$

The chaining of DW-conv and PW-conv makes it possible to generate an output with dimensions coinciding with those obtainable from a multi-channel convolution. Therefore in CNNs the Conv2D layers can be replaced by DS-conv layers without changing the global structure of the model.

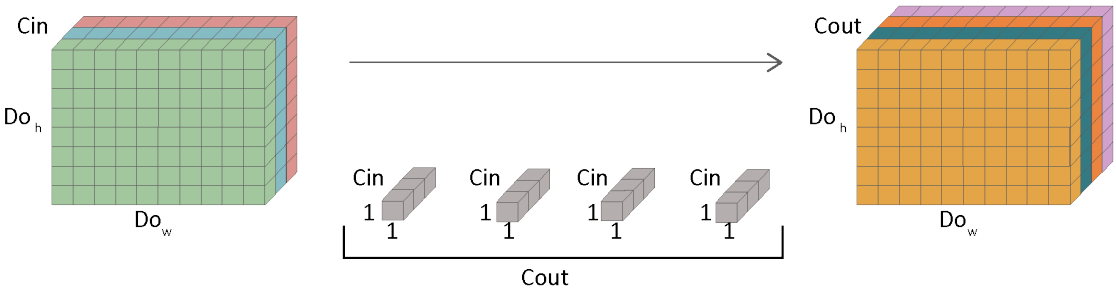

Figure 2.5: Pointwise Convolution.

2.3 Comparison with standard convolutions

The advantages of depthwise separable convolution can be quantified by considering the number of operations performed (multiplications) with respect to standard convolution.
Consider an input of size $C_{in} \times D \times D$; to obtain as output a feature map of size $C_{out} \times D \times D$, $N = C_{out}$ filters of size $C_{in} \times K \times K$ have to be used; consequently, the number of multiplications required is:

$$Conv_{op} = C_{in} \times C_{out} \times D \times D \times K \times K \tag{2.3}$$

By combining (2.1) , (2.2) and (2.3), the ratio between the computational costs of a DS-conv and a CNN is given by:

$$\frac{\text{DS} - \text{conv}_{op}}{\text{CNN}_{op}} = \frac{C_{in} \times D \times D \times K \times K + C_{in} \times C_{out} \times D \times D}{C_{in} \times C_{out} \times D \times D \times K \times K} =$$

$$= \frac{1}{C_{out}} + \frac{1}{K^2} \tag{2.4}$$

A similar estimation regarding the filters parameters can be made, which leads to the following cost ratio:

$$\frac{\text{DS} - \text{conv}_{param}}{\text{CNN}_{param}} = \frac{C_{in} \times K \times K + C_{in} \times C_{out}}{C_{in} \times C_{out} \times K \times K} =$$

$$= \frac{1}{C_{out}} + \frac{1}{K^2} \tag{2.5}$$

2.4 MobileNet

In order to provide a clearer view of depthwise separable convolution technique, the results of one of the first article [9] analysing this type of deep neural network are reported in Table 2.1.
The detailed structure of the DNN in question, called MobileNet, is described in the Appendix D.1.

Model	Dataset	Method	# param (M)	Mult-Adds (M)	Accuracy
Mobilenet	ImageNet	CNN	29.3	4866	71.7%
		DS-CNN	4.2	569	70.6%

Table 2.1: Comparison between MobileNet and CNN on ImageNet.

It can be observed that the MobileNet has about 85% fewer parameters on the same backbone.
Besides the direct advantage on the memory occupation, the strong reduction of multiplications leads to a saving on the computational resources and therefore reduces the energy consumed by the hardware device.
This network can be used in solving many tasks such as image classification, object identification, fine grained recognition and large scale geolocalization.

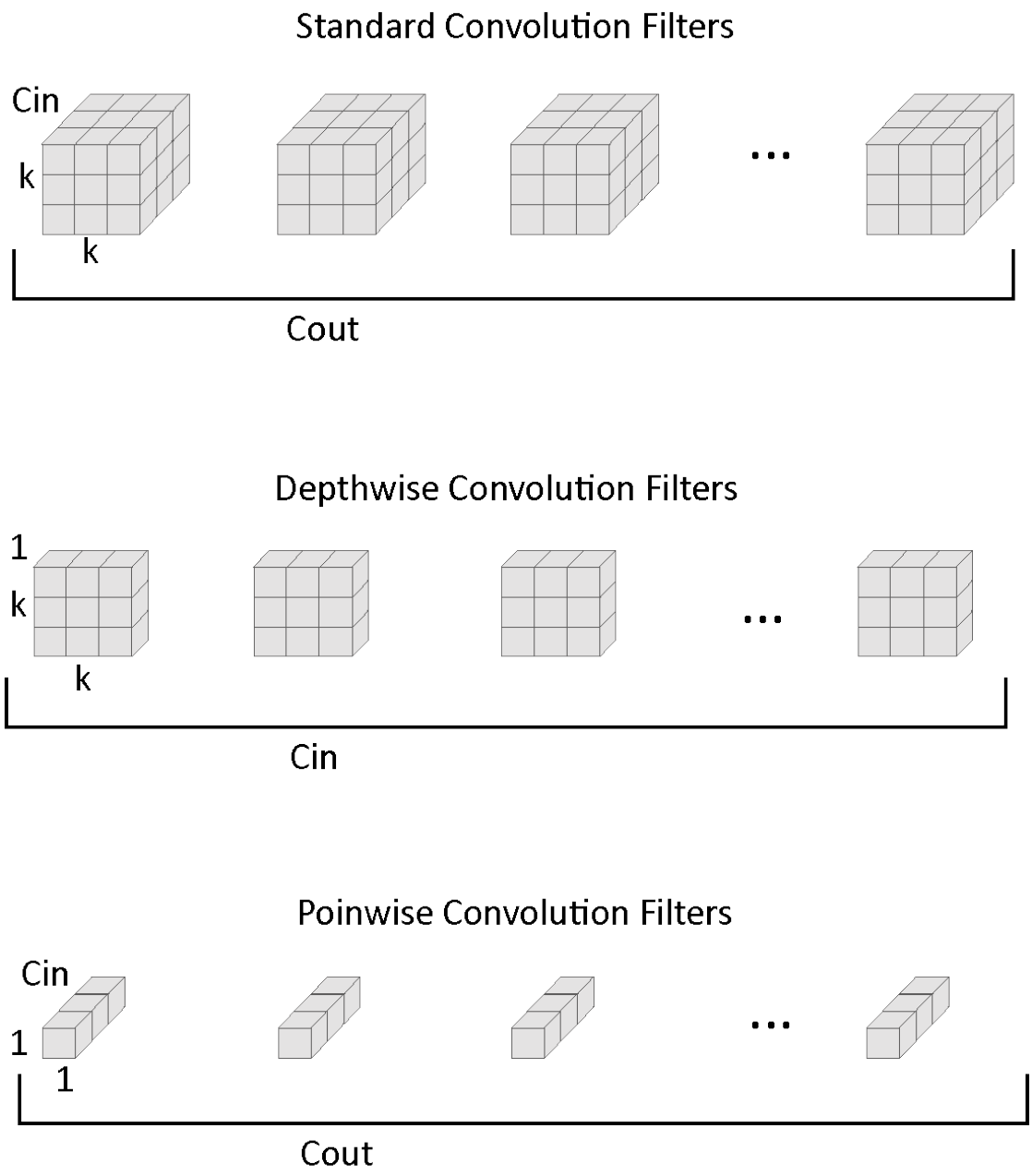

Figure 2.6: Comparison between filters in differet types of convolution.

Chapter 3

AdderNet

3.1 The adder layer

Multiplications constitute the most computationally intensive part of CNNs; as mentioned in Section 1.3 this type of networks exploits the convolution operation to find patterns within the feature map.
Recent studies have attempted to explore new techniques for replacing multiplication with addition by developing a new distance metric that does not significantly degrade performance.
Addernets [12] have been proposed for this purpose, replacing convolutional layers with adder layers.Additive kernels essentially calculate the L^1 distance between weights and inputs according to (3.1) :

$$Y(m,n,t) = -\sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}}|X(m+i,n+j,k) - F(i,j,k,t)| \tag{3.1}$$

This expression computes the similarity between the weights and the feature map. It can be seen that while the output of a convolutional layer can be positive or negative, adder kernels always generate negative output.
In order to use the same activation functions as in CNNs, these layers are followed by batch-normalization layers to obtain values in a suitable range. Nowadays, many network structures already have built-in normalization layers, so these new modules can be inserted directly to replace the standard Conv2D layers.

3.2 Gradient computation in AdderNets

In the training phase, in order to update the weights, backpropagation is carried out and the gradients are calculated. The partial derivative of the outputs of the

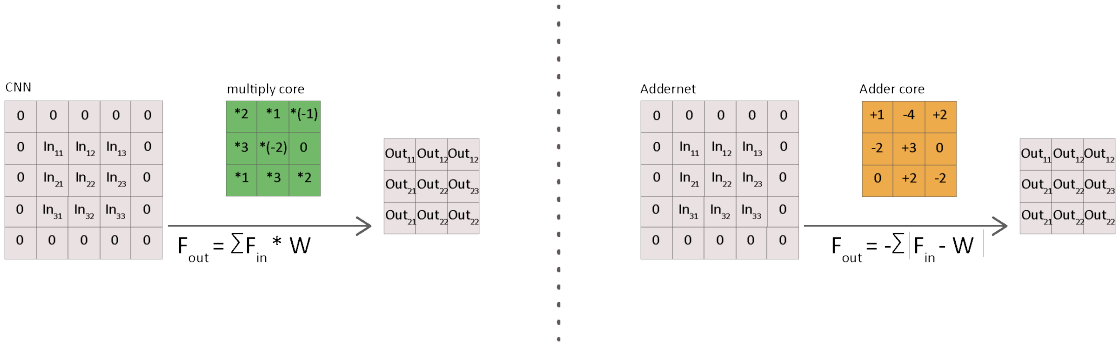

Figure 3.1: Convolutional kernel(left) and adder kernel(right).

convolutional layers can be easily derived:

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = X(m+i,n+j,k) \tag{3.2}$$

On the other hand, in adder networks the derivative of the output Y with respect to the filter F is given by:

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = sgn(X(m+i,n+j,k) - F(i,j,k,t)) \tag{3.3}$$

Due to the sign function, (3.3) only allows 1, 0 or -1 as a result, and its use leads to a singSGD [13]. This approach is inefficient for networks with many parameters and has convergence problems as the size increases [14]. Consequently in the original AdderNet paper [12] it is proposed to use the L^2-norm derivative (3.4) to perform the weight update.

$$\frac{\delta Y(m,n,t)}{\delta F(i,j,k,t)} = X(m+i,n+j,k) - F(i,j,k,t) \tag{3.4}$$

When using the full-precision gradient to calculate the gradient of the output with respect to the X_i input, it must be borne in mind that it will not only affect the calculation for the i-th layer but also the previous layers according to the chain-rule. As a result, the modulus of the gradient increases in the previous layers and this leads to instability and convergence problems.
In order to solve this issue the gradient is clipped between -1 and 1 using the HardTanh function (3.5, 3.6).

$$\frac{\delta Y(m,n,t)}{\delta X(m+i,n+j,k)} = HT(F(i,j,k,t) - X(m+i,n+j,k)) \tag{3.5}$$

$$HT(x) = \begin{cases} x, & \text{if -1<x<1} \\ 1, & \text{if x>1} \\ -1, & \text{if x<-1} \end{cases} \tag{3.6}$$

3.3 Adaptive learning rate

Considering the weights and inputs as independent and identically distributed, the variance in classical CNNs can be estimated as:

$$Var[Y_{CNN}] = d^2 c_{in} Var[X] Var[F] \tag{3.7}$$

Instead, the variance of the adder layers [12] can be roughly approximated as:

$$Var[Y_{AdderNet}] = \sqrt{\frac{\pi}{2}} d^2 c_{in} (Var[X] + Var[F]) \tag{3.8}$$

Usually the variance of the filters Var(F) is very small [15], consequently the variance of the output of Addernets is higher than in CNNs.
The high value of the variance causes the value of the gradient with respect to the input X in the AdderNets to be much lower than in the standard convolutional layers. Consequently, due to the chain rule, the gradient with respect to the filters of the adder kernels also decreases [12].
If the gradient magnitude is too low, the update of weights in the Addernet may slow down and network convergence may not be achieved.
A first approach would consist of increasing the learning rate, i.e. the hyper-parameter that determines the step size at each iteration while moving towards a minimum of a loss function [16]. Learning rate can be imagined as the speed at which an ML algorithm "learns".
However, it is worth noting that there is considerable variation in the modulus of the filter gradients across different layers (Table 3.1) of the network and so a more specific technique is needed.

Model	Layer 1	Layer 2	Layer 3
AdderNet	0.0009	0.0012	0.0146
CNN	0.2261	0.2990	0.4646

Table 3.1: The L^2-norm of gradient of weight in each layer using different networks at 1st iteration [12].

In order to avoid this unwanted effect, an adaptive learning rate is introduced [12]. In particular, a local learning rate is computed to take account of the filter

modules.

$$\alpha_l = \frac{\eta\sqrt{k}}{||\Delta L(F_l)||_2} \tag{3.9}$$

In (3.9) η is a tunable hyperparameter, k is the number of elements in kernel F_l and $\Delta L(F_l)$ represent the gradient of the filter in layer l.
Then the update of the filter F to the adder layer l is calculated as follows:

$$\Delta F_l = \gamma \times \alpha_l \times \Delta L(F_l) \tag{3.10}$$

Where α_l is the local learning rate and γ is the global learning rate of the entire neural network.

3.4 Comparison with CNN

Various articles have proved the potential of the Addernet, in order to have a starting point for working on the adder kernels it was necessary to initially replicated the results of the original layer.
In particular PyTorch is used (an open source framework used for AI applications with python as interface described in Appendix A) and trained Resnet-20 (Appendix C) on CIFAR10 and CIFAR100 (Appendix E).
In Table 3.2 are the results obtained, which are congruent with those of the reference article.

Model	Dataset	Method	param	param. size (MB)	Accuracy
Resnet-20	CIFAR10	CNN	272,484	1.09	92.5%
		AddNN	272,484	1.09	91.4%
	CIFAR100	CNN	278,424	1.11	68.7%
		AddNN	278,424	1.11	67.6%

Table 3.2: Comparison of AddResNet-20 with and without depthwise separable adder layers.

In contrast to DS-convs, addernets do not provide a direct reduction in the number of parameters and in memory occupation. The main advantage is that, if batch normalization layers are neglected, this model contains no multiplications.
At the cost of a reasonable loss of accuracy, AdderNet provides a faster model. Compared to multiplication, the results of addition operation (or subtraction in this case) does not need to be rescaled because weights, input and output tend to be in the same range. This saves logic resources and further reduces power consumption.

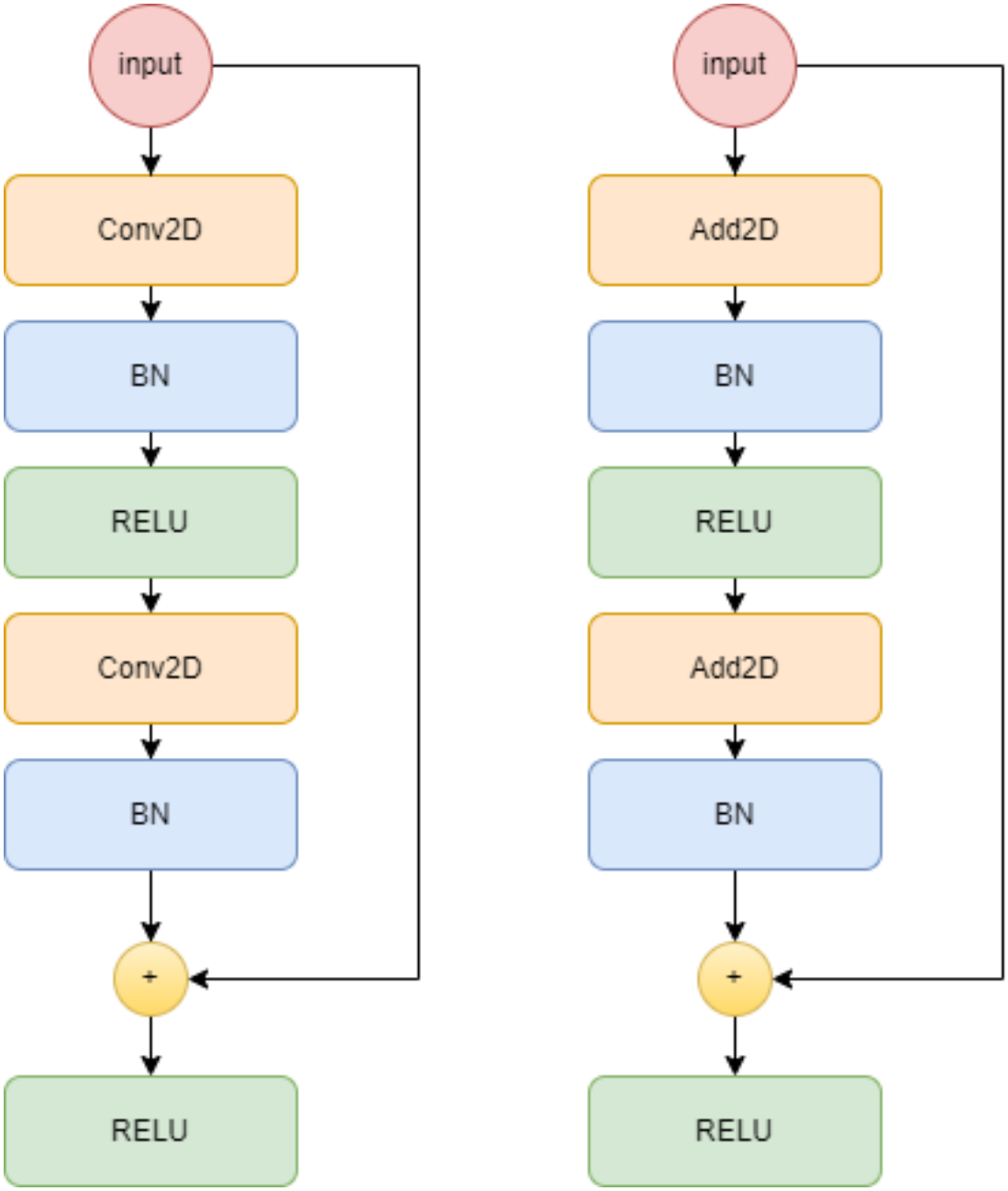

Figure 3.2: Comparison between basic block in the standard ResNet-20 (left) and the AddResNet-20 (right).

3.5 Hardware implementation

It should be noted that modern GPUs are optimised for convolution, and in particular have modules that perform MAC operations. As a result, AdderNet is slower than CNN on this type of device, especially in the training phase.
However, graphics cards are particularly expensive hardware from an energy point of view and the power they require to run is far greater than what mobile and edge devices can provide at the moment. By implementing an adder kernel on a device capable of performing additions more efficiently (such as FPGAs/ASICs), the time required and resources used are significantly reduced compared to CNNs. In fact, the core of a convolutional kernel is a MAC unit, whereas in the case of the adder layer, a comparator and an adder (1C1A) or two adders (2A) can be used, as shown in Figure 3.3 [17].

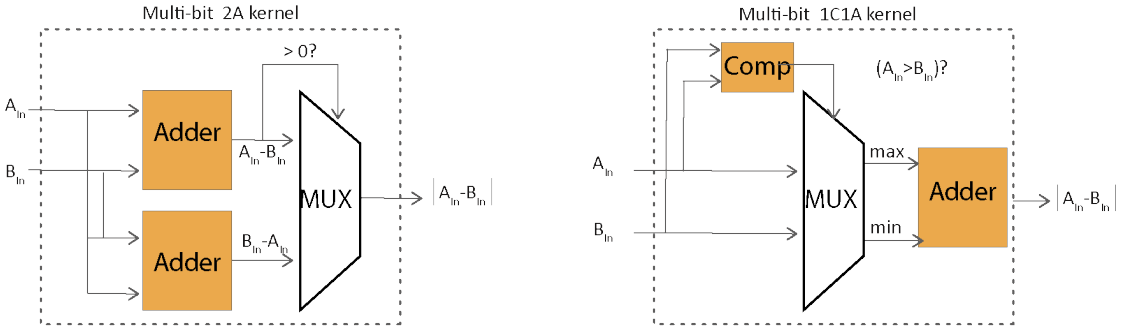

Figure 3.3: Two typical design of adder convolution kernel in 1C1A and 2A, respectively [17].

A possible implementation of FPGA accelerator structure for convolutional kernels is shown in Figure 3.4. It is divided in 4 parts: data storage unit and Input/Output port, data path control module,structures for other operations (such as the Pooling and BN unit) and parallel kernel operation core. The latter is usually the most expensive in terms of logic resources due to the Single Instruction Multiple Data (SIMD) architecture organization of FPGAs [18]. In fact, with a 16 bits data parallelism and 64 input channels to be summed in the adder tree, AdderNet theoretically uses 81.6% less logical resources and power than convolutional networks [17].
However, an appropriate estimation of the energy benefits of AdderNet is still a challenge. Data overhead is of great importance in the implementation of the model on FPGAs, as the data transfer from DRAM to computational resources constitutes the most important energy and time bottleneck.

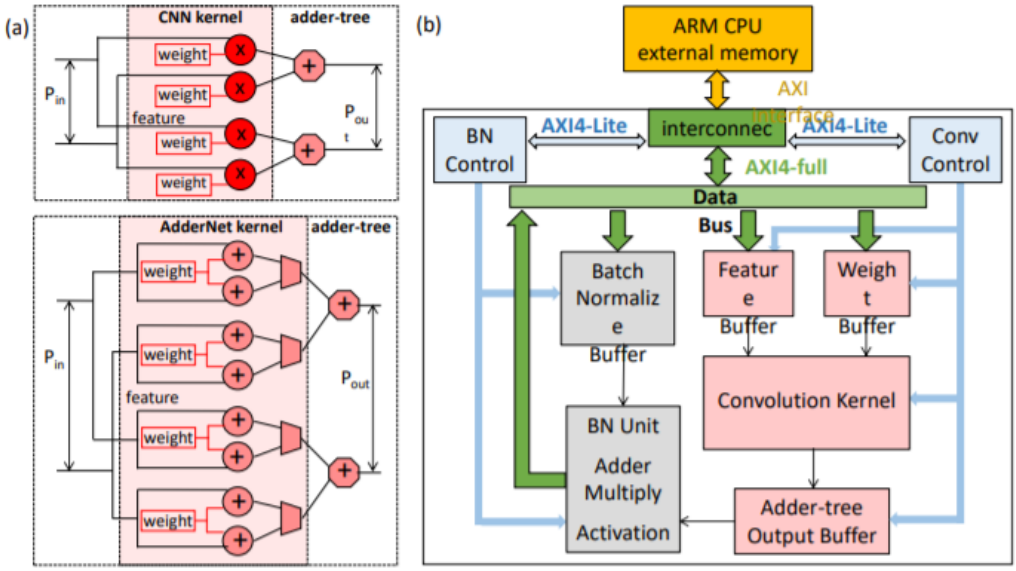

Figure 3.4: Universal AdderNet accelerator. (a) Detailed structure of the FPGA accelerator that can be used for universal Convolutional Neural Networks. (b) Parallel computation on the convolution kernel. The AdderNet convolutional kernel mainly contains two adders and one multiplexer, and the CNN kernel contains one multiplier. Though the AdderNet kernel seems to be more complex, actually it is much lightweight than that of CNN[17].

Chapter 4

Depthwise Separable AdderNet

4.1 Depthwise Adder Layer

Both DS-CNN and AdderNet have energy, memory and efficiency advantages, especially when considering edge devices as application platforms.
These techniques are not mutually exclusive, but can be used simultaneously by constructing a new "depthwise separable adder" neural network. In order to combine Addernets with depthwise separable convolutions two blocks have to be used: a depthwise adder layer and a pointwise adder layer.
The first step in order to verify the feasibility of the idea is to develop this two layers in Python and train a model with them on a GPU.
GPUs are more suitable for the model development and characterization: once the architecture and its parameters are obtained, the VHDL/Verilog code for the hardware platform (e.g, FPGA) is generated using high-level synthesis CADs, and the model performance can be properly evaluated.

4.1.1 The proposed adder layer

The pointwise layer is created by employing an Adder2D layer with 1×1 kernels. From now on, this layer is called "pwAdder2D".
The adder-depthwise part is developed using PyTorch, by inheriting the class `Layer` and constructing the new operator on top of it. As discussed in Chapter 3, the forward operation is not differentiable; consequently, the backpropagation function is rewritten in order to correctly perform the wieights update during training.
Moreover, in this implementation, unbiased kernels are used, as in the reference work [12].

4.2 CUDA kernel

One of the major drawbacks of using a high-level PyTorch descripition of the layer is the long training time. This is due to the fact that GPUs are optimized to perform MAC operations, typical of standard convolutions. Even if, in principle, an addition operation is characterized by a lower latency with respect to the multiplication one, longer training times are registered switching from a CNN to an AdderNet.
One way to reduce the problem is to use custom C++ and CUDA extensions with which PyTorch allows to write kernels using a customized implementation on GPU hardware [19].
The basic structure of the code used to parallelize and improve the calculation of additive kernels proposed in [20] is presented; this is then modified to be used also for the Depthwise Adder Layer.
In order to produce the output feature map, Depthwise operation iterates over 6 dimension: batch size (`b`), channels (`c_in = c_out`), output width (`o_w`), output height (`o_h`), filter width (`f_w`), filter height (`f_h`).
The used GPU is an NVIDIA GTX 1070, which has a maximum number of 1024 threads per block parallelizable over a maximum of three dimensions.
There are three different kernels:

- Inference kernel.
- Input gradient backpropagation kernel.
- Weight gradient backpropagation kernel.

To decide how to assign computations to thread and blocks the following procedure is performed.

4.2.1 Inference and Input Gradient Kernels

For inference/input gradient kernels, each thread computes a set of output values/input gradient values of the feature maps considered; this means that the kernel iterates over `dimSET`, `f_h`, and `f_w`.
`dimSET` represents the total number of elements calculated by one single thread and depends from computation partitioning over threads.
The number of pixels of the output feature map is given by :
`n_pxs = b * c_in * o_w * o_h`.
Considering a maximum number of blocks `MAX_BLOCKS`, the number of blocks used is given by:

```
n_blks=min(MAX_BLOCKS, (n_pxs-MAX_THREADS+1)/(MAX_THREADS+1))
```

Where the constant `MAX_THREADS` is 1024 for the GPU. This formula ensures that at least one block and at most `MAX_BLKS` blocks are allocated.
Given:

- G_i = grid index
- B_n = `n_blks`
- T_i = thread index
- T_n = `MAX_THREADS`

It follows that:

```
k=0
SET=0
offSET = B_i*T_n + T_i
addSET = B_n*G_i
for all k > 0 with SET < n_pxs do
  SET = offSET + k*addSET
end for
```

Where `SET` represents the index of output element computed by a specific thread, `offSET` and `addSET` are respectively the index of the first ouput element and the distance (in terms of memory address) between two consecutive elements computed by the thread.
This means that each thread will compute the output values that correspond to a multiple of the `n_blks` shifted by the thread number, given a 1D view of the output feature map.

4.2.2 Weight Gradient Kernel

For Weight gradient backpropagation kernel, each thread will compute part of one element of the weight gradient tensor. Each thread will iterate over `b_p` and `k_p`, where `b_p` is a portion of the batch size and `k_p` is a portion of the feature map of each single image of the batch.
The number of elements of the weight tensor is `c_in * f_w * f_h` and is equal to the grid dimension (`n_blks`).
Differently from previous kernels, in this case the concept of warp is introduced. The warp is a group of threads in a block that cannot run concurrently if the execution flow diverges (i.e. if a conditional statement is used and they are not following the same branch).

In NVIDIA GPUs usually the `WARP_SIZE` is equal to 32. This concept is necessary because the element is not computed entirely in a thread and there is an "if" statement in the kernel to take care of the padded values. Efficiently allocation of warps allows to reduce the groups of thread which are not completely parallelized by the scheduler.
The number of threads per block used is:

```
num_threads = min(MAX_THREADS, (batch_size * WARP_SIZE))
```

It is important to ensure that each thread is assigned a specific amount of feature maps per batch needed to compute the gradient value.
Given:

- B_i = block index
- T_i = thread index
- T_n = `num_threads`

It follows that:

```
k=0
offBATCH = T_i div WARP_SIZE
addBATCH = T_n div WARP_SIZE
for all k > 0 with xBATCH < batch_size do
  xBATCH = offBATCH + k*addBATCH
end for
```

The formula below ensures that each thread of each warp is assigned a specific set of batch elements for computations:

```
k=0
offELEM = T_i div WARP_SIZE
addELEM = WARP_SIZE
for all k > 0 with xELEM < o_w*o_h do
  xELEM = offELEM + k*addELEM
end for
```

After the calculation of the branches, the threads are synchronized again. The partial values sum up in order to obtain the final gradient of the element.
The thread with index 0 of each block will store it in global memory.

4.3 Training setup

Model training and testing is performed using Pytorch as framework and developed on the GeForce GTX 1070 GPU.
The networks were trained for 400 epochs using a batch size of 256.
For data augmentation, the same approach as in [21] is used: 4 pixels are padded on each side, and a 32×32 crop is randomly sampled from the padded image or its horizontal flip.
For testing,it is evaluated the single view of the original 32×32 image.
The optimizer used is SGD and the models are trained using a cosine learning rate decade [22].
In particular, the learning rate scheduling is calculated using (4.1):

$$l_r = k_l \cdot \left(1 + \cos \frac{e}{e_{max}} \cdot \pi\right) \tag{4.1}$$

Where e represents the epoch considered, e_{max} is the number of total epochs and k_l is a parameter to establish the starting value of the learning rate.
This value constitutes the global learning rate η which is then used in (3.9) to calculate the local learning rate.
For the ResNet-20 model, the learning rate starts from a value of 0.1, while in the case of MobileNetV2 0.01 is used.

4.4 Experiment on Resnet20

As shown in Chapter 3, Resnet20 is one of the first networks in which AdderNets were used. In order to make the model "depthwise separable" is suffficient to substitute the Adder2D layer with a sequence of dwAdder2D, BN, Relu, pwAdder2D.
So the structure of the basic block of the ResNet-20 with the depthwise separable additive convolutions will be the one in Figure 4.1.

Model	Dataset	Method	param	param. size	Accuracy
Resnet-20	CIFAR10	DS-AddNN	61,732	0.25 (MB)	86.6%
		AddNN	272,484	1.09 (MB)	91.4%
	CIFAR100	DS-AddNN	67,672	0.27 (MB)	59.3%
		AddNN	278,424	1.11 (MB)	67.6%

Table 4.1: Comparison of AddResNet-20 with and without depthwise separable adder layers.

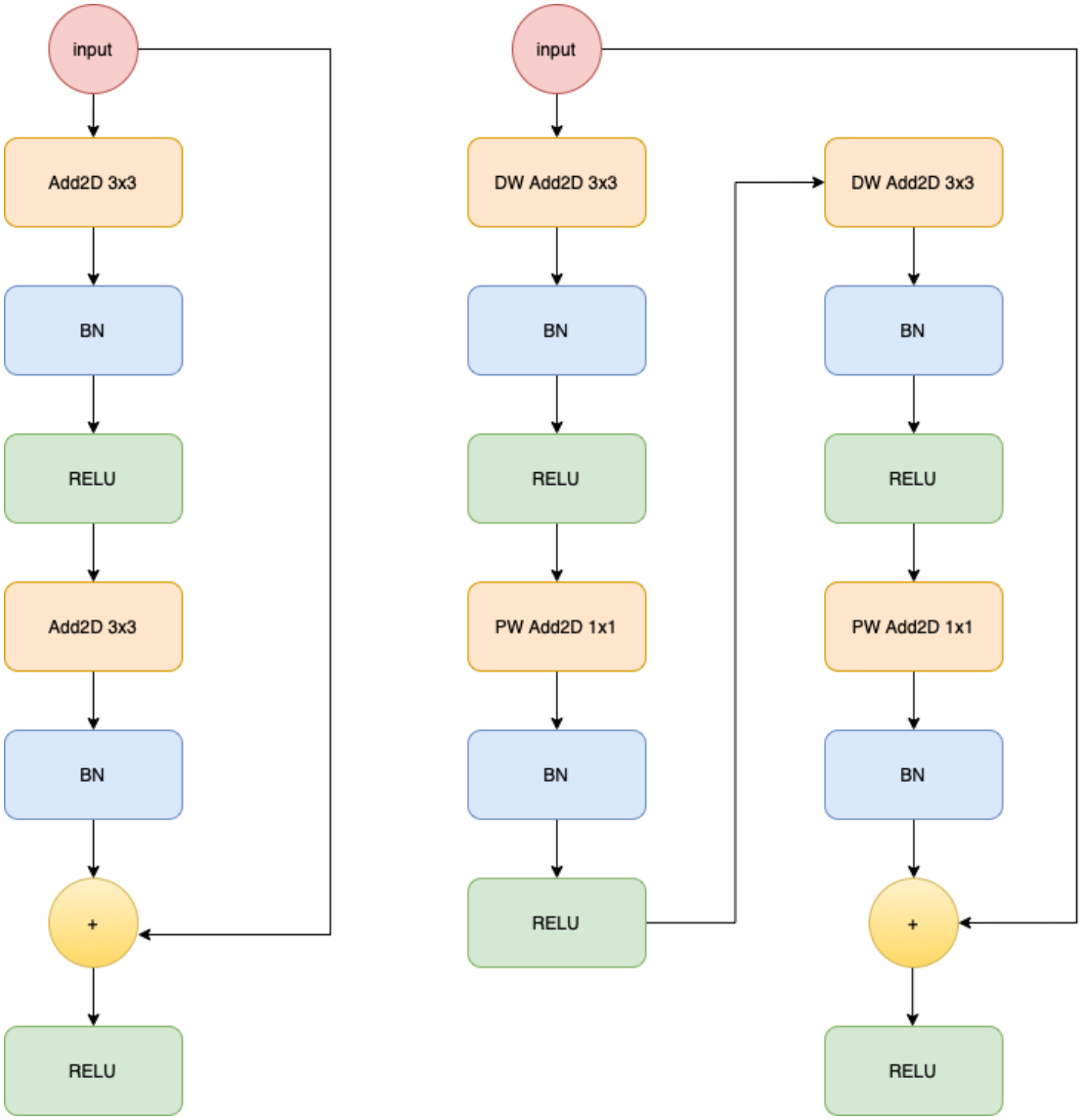

Figure 4.1: Comparison between basic block in AddResNet-20 (left) and Depthwise Separable AddResNet-20 (right).

Observing the results it is possible to notice how the contribution of the deptwhise separable convolution leads to a reduction of about 70% in the memory occupation of the parameters.
In the case of CIFAR-10 there is a loss of accuracy equal to 5% while considering CIFAR-100 as a dataset the accuracy decreases by 8%.
In spite of the metric decrease, the results demonstrate the convergence of the algorithm and, considering the large memory savings it represents a good starting

point for the development of models suitable for edge computing.

4.5 Experiment on Mobilenet

As discussed in Chapter 2, MobileNet is a network that uses natively the deptwhise separable convolutions so in this case there are no advantages in terms of memory occupation by parameters (considering the same quantization).
Therefore, it is sufficient to replace the existing pointwise and depthwise convolutional kernels with the corresponding addition kernels without the need to introduce any further changes to the structure of the model.
For these tests, an improved version of the network called MobileNetV2 [22] is used, details of which can be found in Section D.2.

Model	Dataset	Method	param	param. size	Accuracy
MobileNetV2	CIFAR10	DS-AddNN	2,296,922	9.19 (MB)	89.5%
		CNN	2,296,922	9.19 (MB)	93.8%
	CIFAR100	DS-AddNN	2,412,212	9.65 (MB)	64.6%
		CNN	2,412,212	9.65 (MB)	72.5%

Table 4.2: Comparison of MobileNetV2 with and without adder layers.

In the standard convolutional model the layers perform the same number of multiplications and additions, in the Adder version the operations are all additions. As seen in the previous chapter, this leads to a strong reduction in the power consumption and the logical resources used.
Consequently, the loss in accuracy is acceptable, considering that the results can be further improved in this case as well.

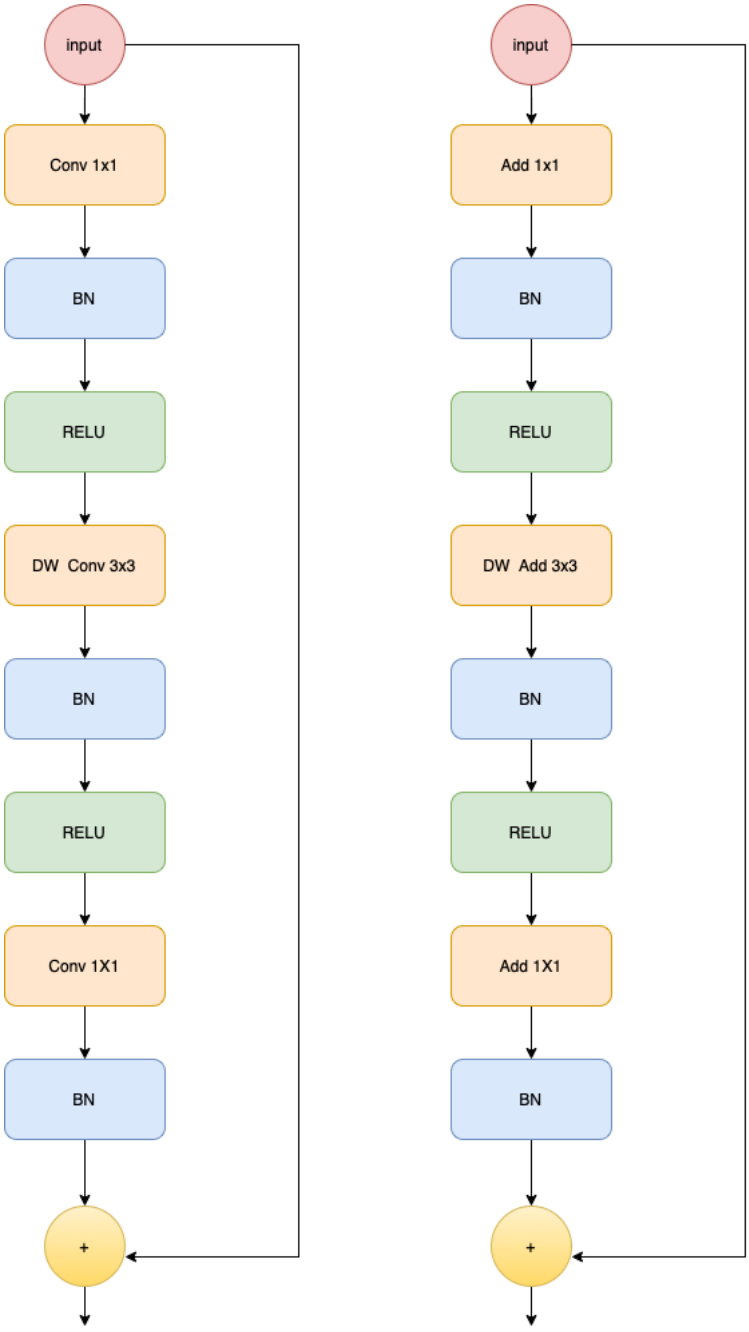

Figure 4.2: Comparison between inverted residual block in the standard MobileNetV2 (left) and the AddMobileNetV2 (right).

4.6 Considerations on the results

It is necessary to highlight that these results have been obtained by performing few trainings (for time reasons) and without a proper fine tuning (only few different values of learning rate have been tested).
The different type of model may require special reasoning with regard to the selection of the hyper-parameters and consequently the results obtained have room for improvement.
Furthermore, estimating the energy benefits of the contribution of the depthwise separable Adder2D layers requires further tests on FPGAs considering quantized models.
The advantages can however be expected to be similar to those found with AdderNet.

Chapter 5

Conclusions

In this work, two DL techniques called AdderNet and Depthwise Separable Convolutions are used to improve the efficiency of two DNNs, ResNet-20 and MobileNetV2, in image classification tasks. The results highlight that by substituting the conventional convolutional layer with an Adder one, it is possible to completely replace the multiplications with additions. This leads to reduced inference latency and power consumption on edge devices, which are characterized by lower computational capabilities, when compared with modern GPUs.

In particular, when deployed on FPGA platforms, the proposed AdderNet is expected to produce a 5x reduction of the hardware resource utilization. The introduction of DS-Conv, moreover, leads to a drastic reduction in the number of parameters and operations for the architecture.
The proposed architecture is trained and tested on two datasets, CIFAR-10 and CIFAR-100, by implementing the DNNs cited above. In particular:

- the ResNet-20 has been compared with the AdderNet version proposed in [12].
- the MobileNetV2 has been compared with the standard CNN implementation.
- on CIFAR-10, an accuracy of 86.6% and 89.5% for the ResNet-20 and MobileNetV2, respectively, are obtained.
- on CIFAR-100, an accuracy of 59.3% and 64.6% for the ResNet-20 and MobileNetV2, respectively, are obtained.
- on the ResNet-20, the substitution of the Adder layer with a DS-Adder one allows to reduce the memory occupation due to the parameters by 70%.

Hence, it is shown that the proposed DS-AdderNet converges during training and provides an acceptable accuracy, if one considers that the model is deployed on devices with reduced hardware capabilities. Moreover, for some architectures (e.g. ResNet-20), a large reduction in memory resource utilization can be obtained.

5.0.1 Future work

This work can be improved from different points of view:

- fine-tuning can be performed to furtherly improve classification accuracy and training time.
- more datasets and classification tasks (e.g. object detection, tracking etc.) can be used to consolidate the proposed architecture performance and drawbacks.
- actual deployment of the proposed DNN would allow to measure more precisely the impact of the new layer on resources utilization, power consumption and inference latency.

Appendix A

Framework Pytorch

Pytorch [23] is a Python-based framework aimed at research prototyping and production implementation in the field of machine learning[22].
This library has become a popular resource in the deep learning research community by integrating a focus on user friendliness with careful performance considerations. This scientific computing package serving two broad purposes:

- A replacement for NumPy to use the power of GPUs and other accelerators.
- An automatic differentiation library that is useful to implement neural networks.

It uses dynamic computation graphs and allows to run and test portions of the code in real-time. Thus, users don't have to wait for the entire code to be implemented to check if a part of the code works.
The most important data type is the tensor, which is a container that can hold data of various sizes. It can be a number, a vector, or a matrix and can be easily handled by the CPU or GPU to make calculations faster and more efficient.
By default PyTorch uses 32-bit Float as the data type for the tensor elements, so when we talk about unquantized data, this representation is intended.

A.1 Common modules

A short description of the modules that were useful for the realization of this work is presented:

- nn : the nn module includes various classes that help to build neural network models. All modules in PyTorch subclass the nn module.

- Optim : the Optim module is a package with pre-written algorithms for optimizers that can be used to build neural networks.
- Autograd : the autograd module is PyTorch's automatic differentiation engine that helps to compute the gradients in the forward pass in quick time. Autograd generates a directed acyclic graph where the leaves are the input tensors while the roots are the output tensors.

PyTorch provides a plethora of operations related to neural networks, arbitrary tensor algebra, data wrangling and other purposes.
In addition, the ability to use C++ extensions to create operators defined out-of-source (separate from the PyTorch backend) is included.

Appendix B

The CUDA Programming Model

The CUDA programming model [19] represents an abstraction of the GPU hardware structure that connects an application to its possible implementation on the GPU. It divides the computational structure in:

- Grids: group of blocks
- Blocks: group of threads, a streaming multi-processor
- Threads: computational unit

The CUDA kernel is a function that runs on the GPU. The parallel part of applications is executed K times in parallel by K separate CUDA threads, as opposed to only once as normal C++ operations.
Each CUDA block is executable by a streaming multiprocessor (SM) and cannot be moved to other SMs (except during dynamic CUDA parallelism, preemption, or debugging). One SM can run several CUDA blocks depending on the amount of resources required by the CUDA blocks. Each kernel runs on one GPU and CUDA supports running multiple kernels on one device at one time.
Each thread has its own local memory and registers, shares the "shared memory" with the other threads of his block and the constant/global memory with the threads of the other blocks (Figure B.1).

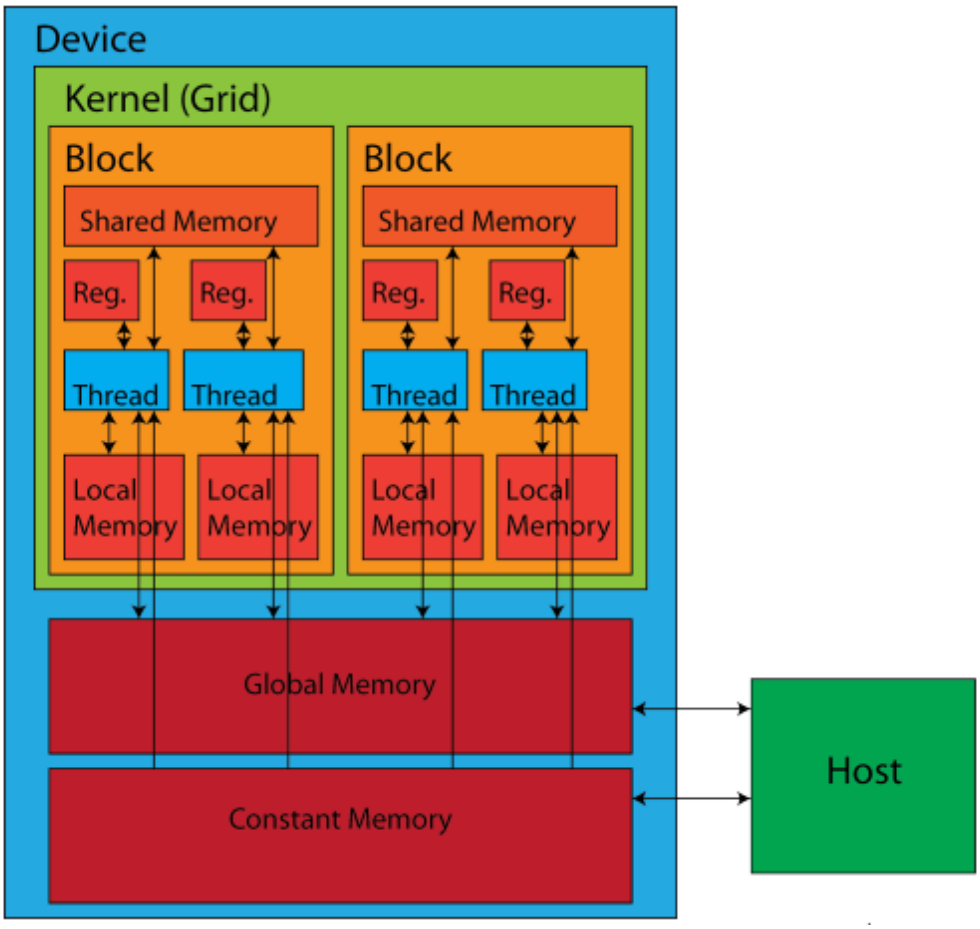

Figure B.1: CUDA memory programming model.

Appendix C

ResNet

ResNet[21] is a deep convolutional neural network which uses a technique known as skip connection paved the way for residual networks.
This type of architecture was introduced in 2015 and won in the ILSVRC (ImageNet Large Scale Visual Recognition Challenge).
A problem common to many DNNs [24] is summarised by two concepts:

- Vanishing gradient : the gradient becomes too small resulting in minimal updating of the weights and causing a slowdown in the training process.
- Exploding gradient : the gradient becomes excessively large causing instability problems and generating weights that exceed those manageable by the computer (overflow) resulting in values that cannot be updated further.

These issues become progressively important as the number of layers in the network increases and the gradient tends to degrade due to the chain rule.
In order to circumvent this issue, residual blocks have been introduced.
These structures consist of sequences of standard convolutional layers (usually two or three) that contain nonlinearities (ReLU) and batch normalisation in between with the addition of skip connectors.
The basic structure of the residual block is shown in Figure C.1, the skip connections perform identity mapping without adding parameters or increasing computational cost.
Thus the function representing the otput of a residual block can be expressed as:

$$y = F(x) + x \tag{C.1}$$

Considering a loss function $L(y(x))$ then its partial derivative will be given by:

$$\frac{dL}{dx} = \frac{dL}{dy}\frac{dy}{dx} = \frac{dL}{dy}\Big(\frac{dF}{dx} + 1\Big) = \frac{dL}{dy}\frac{dF}{dx} + \frac{dL}{dy} \tag{C.2}$$

The Equation C.2 approximates the calculation of the gradient during backpropagation and allows us to appreciate the benefit of the skip connetion.

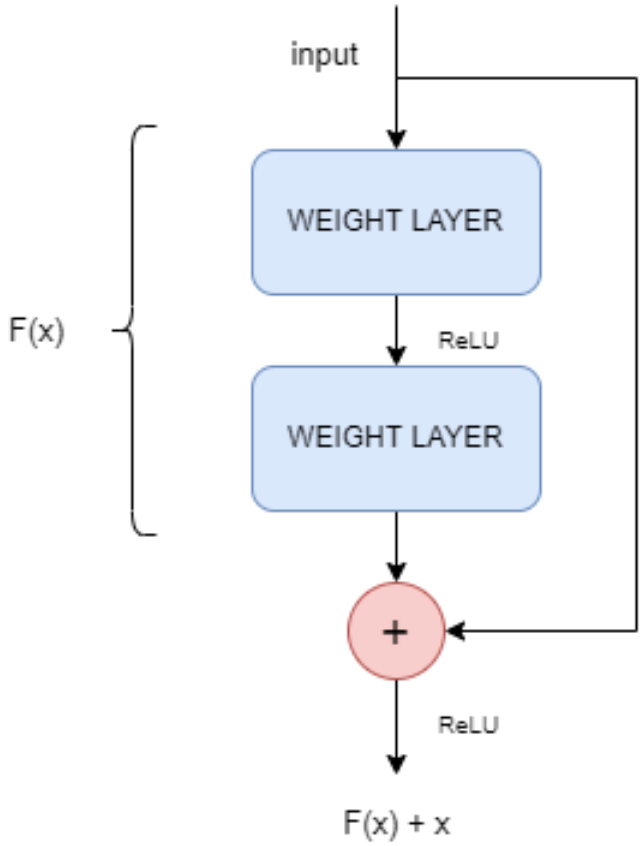

Figure C.1: Residual block with skip connetion.

This method allows the development of very deep models that do not suffer from the stability problems caused by the gradient calculation.
Models with several parallel skips are referred to as DenseNets [25].
The Table C.1 shows the structure of the ResNet-20 used in this work, the structure of the residual block is presented in the Figure 3.2.
In the residual block 4 and in the residual block 7 the skip connection is carried out through a convolutional layer that downsamples the input (increasing the size of the channels) in order to make it compatible with the dimensions of the output and carry out the addition.
Each convolutional operation is followed by a batch normalization and ReLU.

Type / Stride	**Filter Shape**	**Input Size**
Conv	$3 \times 3 \times 3 \times 16$	$32 \times 32 \times 3$
Residual Block 1	$3 \times 3 \times 16$ x 16	32 x 32 x 16
	$3 \times 3 \times 16$ x 16	32 x 32 x 16
Residual Block 2	$3 \times 3 \times 16$ x 16	32 x 32 x 16
	$3 \times 3 \times 16$ x 16	32 x 32 x 16
Residual Block 3	$3 \times 3 \times 16$ x 16	32 x 32 x 16
	$3 \times 3 \times 16$ x 16	32 x 32 x 16
Residual Block 4	$3 \times 3 \times 16$ x 32	32 x 32 x 16
	$3 \times 3 \times 32$ x 32	16 x 16 x 32
Residual Block 5	$3 \times 3 \times 32$ x 32	16 x 16 x 32
	$3 \times 3 \times 32$ x 32	16 x 16 x 32
Residual Block 6	$3 \times 3 \times 32$ x 32	16 x 16 x 32
	$3 \times 3 \times 32$ x 32	16 x 16 x 32
Residual Block 7	$3 \times 3 \times 32$ x 64	16 x 16 x 32
	$3 \times 3 \times 64$ x 64	8 x 8 x 64
Residual Block 8	$3 \times 3 \times 64$ x 64	8 x 8 x 64
	$3 \times 3 \times 64$ x 64	8 x 8 x 64
Residual Block 9	$3 \times 3 \times 64$ x 32	8 x 8 x 64
	$3 \times 3 \times 64$ x 64	8 x 8 x 64
Avg Pool / s1	Pool 8×8	8 x 8 x 64
FC / s1	64×10	1 x 1 $\times$ 64
Softmax / s1	Classifier	1 x 1 x 10

Table C.1: ResNet-20 architecture for CIFAR10.

Appendix D

Mobilenet

D.1 MobileNet

The first version of MobileNet[9] replaced the classic convolutional layers with a sequence of depthwise and pointwise layers, as shown in the Figure D.1.

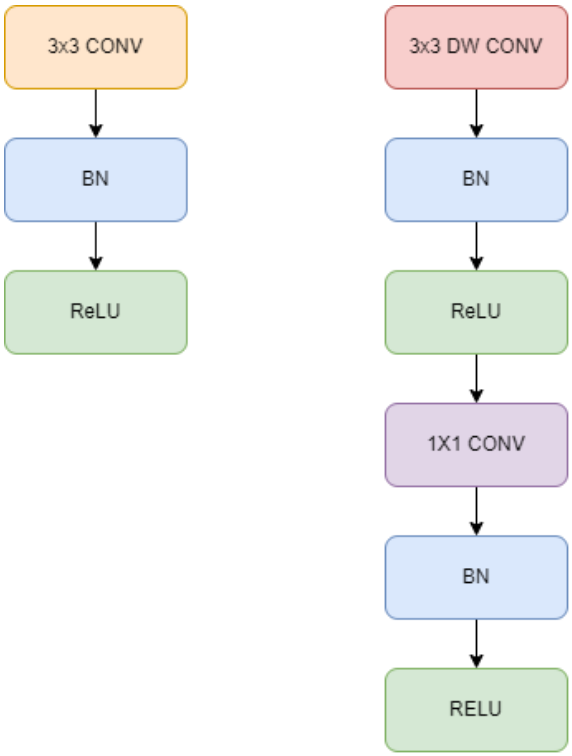

Figure D.1: Standard convolutional layer (left) with batch normalization and ReLu and Depthwise Separable Convolution (right).

The complete network structure is presented in Table D.1, it has more than 4 million parameters and achieves an accuracy of 71.7% on ImageNet.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$56 \times 56 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 256 \times 256$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 256$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5x Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	1x1x1000

Table D.1: MobileNet architecture.

D.2 MobileNetV2

An improved version of MobileNet[22] was introduced in 2018 based on an "inverted residual block" in which the shortcut links are between the "bottleneck layers".

D.2.1 Bottleneck Layer

In [21] a residual block (Appendix C) is introduced with a three-layer structure instead of the standard one with two layers.
This design choice, called bottleneck, was mainly aimed at improving the training

time for very deep networks and larger datasets.
The three layers that form the block (bottleneck) are respectively 1×1, 3×3, and 1×1 convolutions.
The 1×1 layers are responsible for reducing and then restoring the dimensions, leaving the 3×3 layer a bottleneck with smaller input/output sizes.
In the version of the bottleneck block used in the ResNets, each convolutional layer is followed by a batch normalization and a ReLU, the skip connection is made between the input of the block and the output of the last layer, immediately before the activation function.

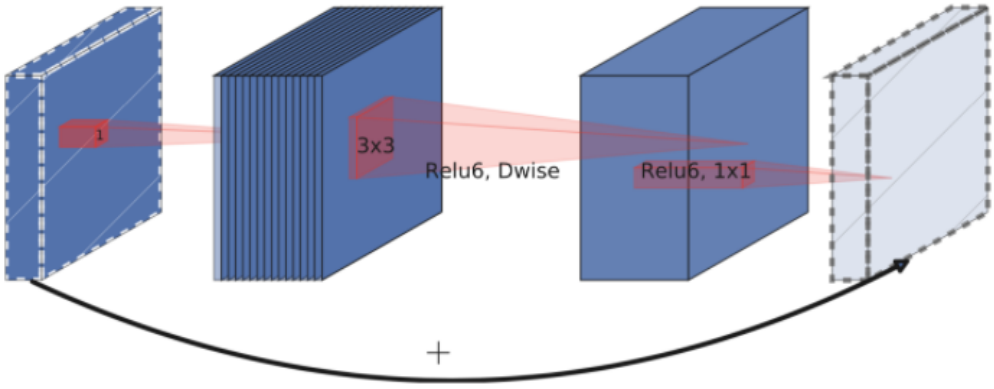

Figure D.2: Visualization of the intermediate feature maps in the inverted residual layer [26].

D.2.2 Inverted residual

MobilenetV2 uses a block very similar to Residual Bottleneck called Inverted residual.
The latter, however, replaces the standard 3x3 convolution with its depthwise counterpart.
Inspired by the intuition that the bottlenecks actually contain all the necessary information, while an expansion layer acts merely as an implementation detail that accompanies a non-linear transformation of the tensor, this architecture use shortcuts directly between the bottlenecks.[22]
The structure of the MobileNetV2 basic blocks is presented in the Figure D.3, while the complete architecture is schematized in the Table D.2

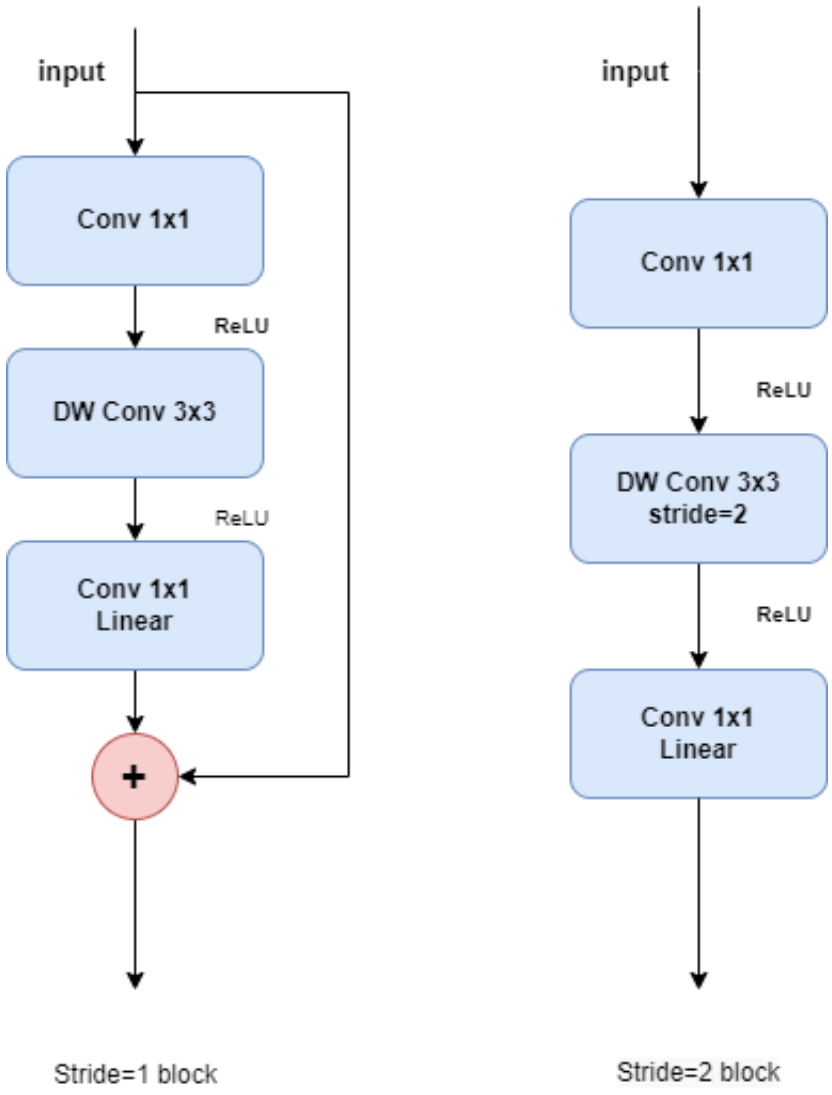

Figure D.3: Basic blocks of MobileNetv2 : the inverted residual block with *stride* = 1 (left) and the depthwise bottleneck with *stride* = 2 for downsizing (right). Note that the third layer (1×1 convolution) has no non-linearity.

Type / Stride	**Filter Shape**	**Input Size**
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
bottleneck / s1	$3 \times 3 \times 16$ dw	$112 \times 112 \times 32$
bottleneck / s2	$3 \times 3 \times 24$ dw	$112 \times 112 \times 16$
bottleneck / s2	$3 \times 3 \times 32$ dw	$56 \times 56 \times 24$
bottleneck / s2	$3 \times 3 \times 64$ dw	$28 \times 28 \times 32$
bottleneck / s1	$3 \times 3 \times 96$ dw	$14 \times 14 \times 64$
bottleneck / s2	$3 \times 3 \times 160$ dw	$14 \times 14 \times 96$
bottleneck / s1	$3 \times 3 \times 320$ dw	$7 \times 7 \times 160$
Conv / s1	$1 \times 1 \times 320 \times 1280$	$7 \times 7 \times 320$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1280$
FC / s1	1280×1000	$1 \times 1 \times 1280$
Softmax / s1	Classifier	1x1x1000

Table D.2: MobileNetV2 architecture. The table does not show the 1x1 filters of the bottleneck blocks.

Appendix E

Datasets

E.1 CIFAR10

CIFAR-10 (Canadian Institute For Advanced Research) is a dataset mainly used to train machine learning algorithms for computer vision.
It consists of 6000 colour images of size 32x32 divided into 10 classes.
Each class comprises 6000 images and the data is divided into 50000 samples for training and 10000 for testing.
The 10 classes are: airplane, automobile, bird, deer, dog, frog, horse, ship, truck.
They are completely mutually exclusive (there is no overlap between automobiles and trucks).
CIFAR-10 is a labeled subset of the 80 million tiny images dataset. When the dataset was created, students were paid to label all of the images [27].

E.2 CIFAR100

CIFAR-100 is very similar to CIFAR-10 in that it consists of the same number of images with the same format.
However, this dataset is divided into 100 classes, each including 600 images (500 for training and 100 for testing).
The 100 classes are grouped into 20 superclasses. Each image comes with a "fine" label (the class it belongs to) and a "coarse" label (the superclass to which it belongs). In Table E.1 lists the classes of the dataset.

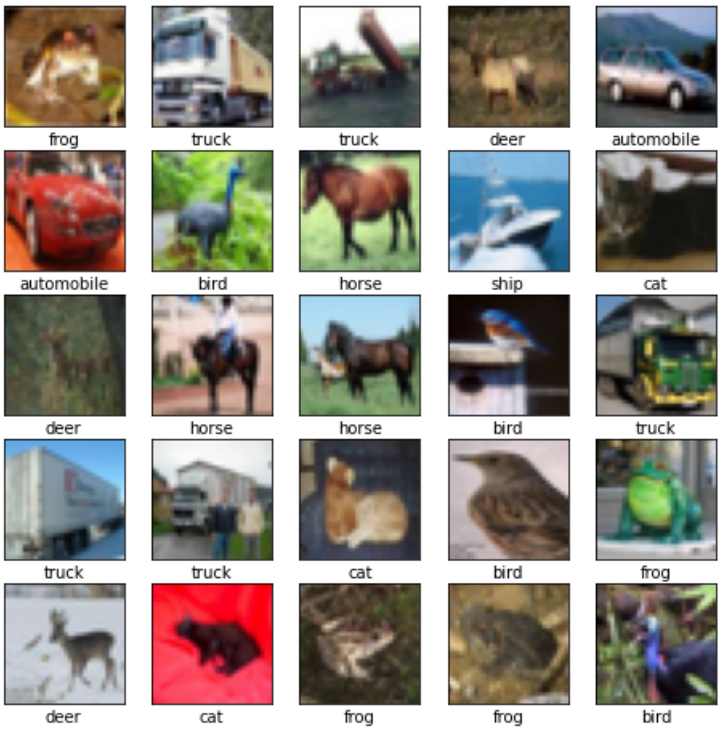

Figure E.1: First 25 labeled images from CIFAR10.

E.3 ImageNet

ImageNet is a large image database, realized for artificial vision and object recognition. The dataset consists of more than 14 million images which have been annotated manually with an indication of the objects represented in them and the bounding box that delimits them[28].
The identified objects have been classified into more than 20,000 categories.
The database with third-party image annotations is available for download from ImageNet, even though the images are not part of the project (only the link to the images is provided).
Since 2010, a competition called ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has been held every year, where programs compete to classify and correctly detect objects and scenes in the images.
A reduced list of images with objects from a thousand non-overlapping categories is used in the competition.

SUPERCLASS	CLASSES
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Table E.1: Superclasses and classes of CIFAR-100.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 1, 2, 6, 8).

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006 (cit. on p. 1).

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, Dec. 2002. ISBN: 0137903952 (cit. on p. 2).

[4] Wikimedia Commons. *File:Neural network.svg — Wikimedia Commons, the free media repository*. [Online; accessed 24-March-2022]. 2021. URL: https://commons.wikimedia.org/w/index.php?title=File:Neural_network.svg&oldid=606652357 (cit. on p. 3).

[5] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 2014. URL: http://arxiv.org/abs/1412.6980 (cit. on p. 7).

[6] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 (cit. on p. 11).

[7] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. *How Does Batch Normalization Help Optimization?* 2019. arXiv: 1805.11604 (cit. on p. 12).

[8] Laurent SIfre and Stéphane Mallat. *Rigid-Motion Scattering for Texture Classification*. 2014. arXiv: 1403.1687 (cit. on p. 15).

[9] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 (cit. on pp. 15, 19, 46).

[10] François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2017. arXiv: 1610.02357 (cit. on p. 15).

[11] Irwin Sobel. *History and Definition of the Sobel Operator*. 2014 (cit. on p. 15).

[12] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. «AdderNet: Do We Really Need Multiplications in Deep Learning?» In: *CVPR* (2020) (cit. on pp. 21–23, 28, 37).

[13] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. *signSGD: Compressed Optimisation for Non-Convex Problems*. 2018. arXiv: 1802.04434 (cit. on p. 22).

[14] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. *Convergence rate of sign stochastic gradient descent for non-convex functions*. 2018 (cit. on p. 22).

[15] X. Glorot and Y. Bengio. *Understanding the difficulty of training deep feedforward neural networks*. 2010 (cit. on p. 23).

[16] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.]: MIT Press, 2013. ISBN: 9780262018029 0262018020 (cit. on p. 23).

[17] Yunhe Wang, Mingqiang Huang, Kai Han, Hanting Chen, Wei Zhang, Chunjing Xu, and Dacheng Tao. *AdderNet and its Minimalist Hardware Design for Energy-Efficient Artificial Intelligence*. 2021. arXiv: 2101.10015 (cit. on pp. 26, 27).

[18] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. *Optimizing fpga-based accelerator design for deep convolutional neural networks*. ACM, 2015 (cit. on p. 26).

[19] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. Version 3.2. 2010 (cit. on pp. 29, 41).

[20] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. «ShiftAddNet: A Hardware-Inspired Deep Network». In: *CoRR* abs/2010.12785 (2020). arXiv: 2010.12785. URL: https://arxiv.org/abs/2010.12785 (cit. on p. 29).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2016. DOI: 10.1109/CVPR.2016.90 (cit. on pp. 32, 43, 47).

[22] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. «Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation». In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381 (cit. on pp. 32, 34, 39, 47, 48).

[23] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 39).

[24] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. Ed. by Yee Whye Teh and D. Mike Titterington. 2010. URL: http://dblp.uni-trier.de/db/journals/jmlr/jmlrp9.html#GlorotB10 (cit. on p. 43).

[25] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. «Densely Connected Convolutional Networks». In: *CoRR* abs/1608.06993 (2016). arXiv: 1608.06993. URL: http://arxiv.org/abs/1608.06993 (cit. on p. 44).

[26] Andrew Howard et al. «Searching for MobileNetV3». In: *CoRR* abs/1905.02244 (2019). arXiv: 1905.02244. URL: http://arxiv.org/abs/1905.02244 (cit. on p. 48).

[27] Alex Krizhevsky. «Learning Multiple Layers of Features from Tiny Images». In: *University of Toronto* (May 2012) (cit. on p. 51).

[28] John Markoff. «For Web Images, Creating New Technology to Seek and Find». In: (2012). URL: https://www.nytimes.com/2012/11/20/science/for-web-images-creating-new-technology-to-seek-and-find.html (cit. on p. 52).