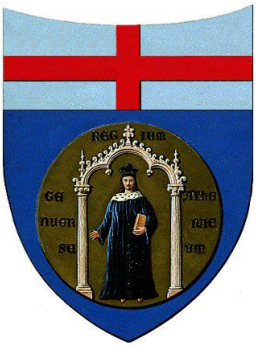
UNIVERSITY OF GENOA



Application-aware optimization of Artificial Intelligence for deployment on resource constrained devices

by

Alessio Canepa

A thesis submitted in partial fulfillment for the degree of
Doctor of Philosophy in Science and Technology for Electronic and Telecommunication Engineering,
Curriculum: Electromagnetism, Electronics, Telecommunications and Interactive Cognitive Environments

in the
Faculty of Engineering
Department of naval, electric, electronic and telecommunications engineering

Supervisors: Prof. Paolo Gastaldo, Rodolfo Zunino
Co-Supervisor: Prof. Edoardo Ragusa
Coordinator of PhD Course: Prof. Maurizio Valle

February 2023

Declaration of Authorship

I, Alessio Canepa, declare that this thesis titled, 'Application-aware optimization of Artificial Intelligence for deployment on resource constrained devices' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

'La felicità non dipende da quanto hai, ma da quanto lo sai apprezzare.'

UNIVERSITY OF GENOA

Abstract

Faculty of Engineering
Department of naval, electric, electronic and telecommunications engineering

Doctor of Philosophy in Science and Technology for Electronic and Telecommunication Engineering,
Curriculum: Electromagnetism, Electronics, Telecommunications and Interactive Cognitive Environments

by Alessio Canepa

Artificial intelligence (AI) is changing people's everyday life. AI techniques such as Deep Neural Networks (DNN) rely on heavy computational models, which are in principle designed to be executed on powerful HW platforms, such as desktop or server environments. However, the increasing need to apply such solutions in people's everyday life has encouraged the research for methods to allow their deployment on embedded, portable and stand-alone devices, such as mobile phones, which exhibit relatively low memory and computational resources. Such methods targets both the development of lightweight AI algorithms and their acceleration through dedicated HW.

This thesis focuses on the development of lightweight AI solutions, with attention to deep neural networks, to facilitate their deployment on resource constrained devices. Focusing on the computer vision field, we show how putting together the self learning ability of deep neural networks with application-specific knowledge, in the form of feature engineering, it is possible to dramatically reduce the total memory and computational burden, thus allowing the deployment on edge devices. The proposed approach aims to be complementary to already existing application-independent network compression solutions. In this work three main DNN optimization goals have been considered: increasing speed and accuracy, allowing training at the edge, and allowing execution on a microcontroller. For each of these we deployed the resulting algorithm to the target embedded device and measured its performance.

Acknowledgements

I would like to thank Edoardo, Paolo and Rodolfo for the trust you have placed in me and for your guidance through this long journey. I hope I managed to give you something back for the help you gave me.

My family, I owe you everything. Thanks for always supporting me in this long work, for believing in me, and for teaching me what's really important in life. You'll always be my haven.

I thank all the friends who are always there, and who understood me when my businesses didn't let me dedicate them all the time I wanted.

I'm also grateful to IVECO for having supported my growth through this journey, the trust you put on me has been foundamental to live these years happily.

Last but not least, I thank you, Alice, for the immense love you give me everyday, and for supporting my efforts in such an intense moment of our lives. This work would have been much harder without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the recent years, Artificial Intelligence (AI) has been an always increasing research field, given the game-changing innovations it has shown to be capable of. One of the most interesting subfields of AI is machine learning, which deals with algorithms able to automatically learn from high amounts of data. Among machine learning methods, neural networks have gained particular interest in the last decade, given their ability to outperforms other approaches in many applications.

Actually, neural networks have not been invented recently. The first prototypes of neural networks date back to the 1960's. However, they became increasingly popular only in the last decade, thanks to the exploding amount of data available (mainly from the internet) to train them, and thanks to the higher availability of computational resources at low cost, which are necessary to train and run them. In fact, by mimicking the way the human brain learns, neural networks exhibit outstanding learning capability, but at the expense of intensive memory and computational cost. The bigger the network, the higher the computational burden.

Researchers soon realized that in many tasks, such as Computer Vision (CV) ones, neural networks perform better when they feature a high number of layers, thus deserving the title "deep". The increasing popularity of Deep Neural Networks (DNNs) gave birth to the Deep Learning (DL) research field. The research for achieving unprecedented accuracy in tasks such as image classification and object detection, thus making the impossible possible and aiming to beat human performance, was initially the main driver in the DL field. For this reason, less attention was put on the memory and computational footprint of such solutions. DNNs in fact can be easily run and trained on desktop or server environments featuring all the required computing power.

More recently, we're assisting to a growing interest in deploying DL-based solutions on embedded devices. Some of the main drivers of this interest are:

- The wide availability of compact, affordable, low power and efficient hardware platforms suitable to run heavier algorithms, although very far from server-like performance. Smartphones are a great example of these, but in this thesis we will see how we were able to deploy such solutions also to much more resource constrained devices such as microcontrollers.
- A rapidly growing market of applications runnable on portable devices (such as smartphones) delivering DL-powered services and solutions to a wide range of customers. Among these we have in particular solutions for semi-automatic image manipulation, picture enhancement and virtual reality.
- The increasing interest in the development of autonomous systems, which make use of artificial intelligence for sensing the environment. A great example of this are autonomous cars, which use sensors like cameras to understand the surroundings. In this case, DNNs for example can be used to process the video from a camera to detect objects such as other cars or pedestrians.

Deploying DNNs to embedded, resource constrained devices is a challenge both from a computational and memory point of view.

Computationally, DNNs are intrinsically based on a huge number of computations, or MACs (multiply and accumulate operations). When targeting real time embedded systems, the neural network could need to process the input data in a fraction of a second. A typical example of this is a CV-based pedestrian detection system in autonomous cars. The need for speed requires designers to reduce the computational cost of the DNN, or exploit more powerful hardware, which would in turn increase costs, space, and energy consumption.

From a memory perspective, DNNs require memory both for storing the network's parameters (constants) and the layers' activations (dynamic). If the lack of computational power can lead to a slower execution of the network, the lack of sufficient memory can make a DNN not even executable.

To face such challenges, researchers designed several methods aimed at deploying DNNs on resource constrained devices. These methods are deeply discussed in Chapter 3 of this thesis and can be splitted in two main categories: hardware-based and software-based. The first ones are incentrated on HW techniques to accelerate the execution of AI algorithms speeding up both computations and access to memory. The second ones

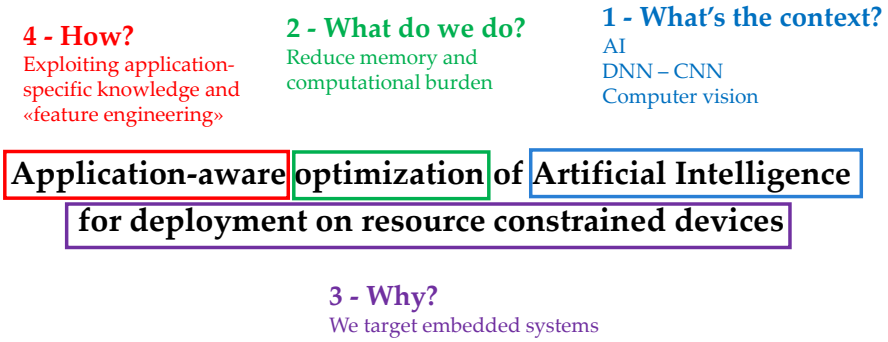


FIGURE 1.1: Thesis title explanation

aim at making the software more lighweight, by reducing the amount of computations, the number of parameters and the memory required for temporary calculations, such as to store layers' activations.

1.1 Contribution

In the above explained context, the contribution of this thesis can be introduced with reference to the thesis' title "Application-aware optimization of Artificial Intelligence for deployment on resource constrained devices", as shown in Figure 1.1:

1. **What's the context?** We're in the context of AI, with particular focus on DNN and CNN applied to computer vision.
2. **What do we do?** We research methods to optimize AI algorithms, mostly based on DNN, in order to have a low memory and computational burden.
3. **Why?** We need to do so to allow their deployment on resource constrained devices such as embedded systems.
4. **How?** We perform this optimization by exploiting application-specific knowledge and "feature engineering".

Point 4 is the main distinguishing element of this thesis. As discussed in Chapter 3, several software-based methods do already exist in order to reduce the footprint of DNNs. However, most of the research in the field targets application-independent methods, that is methods able to compress the neural networks without any knowledge about the target application. This approach is very valuable, since it allows for greater generalisation. Commercial as well as free tools exist that take in input a neural network and try to output a compressed version of the same, with less computations or parameters.

However, the great generalization of these solutions comes at the cost of lower optimization, with respect to what is achievable with an application-aware design of the solution. The wide availability of off-the-shelf models (and pretrained networks) to perform common tasks, such as object detection in CV, tends to devalue the culture for researching application-specific optimizations, despite the great advantage they can bring in terms of solution efficiency. In other words, if in the pre-machine learning era computational problems were mostly approached hand-crafting algorithms according to domain-specific knowledge (consider edge detectors in CV as an example), now the common trend is to heavily rely on the self-learning capabilities of neural networks. The first approach is often referred to as "feature engineering", while, in the second one, features are learned from the NN itself.

The goal of this thesis is to show that merging together the learning ability of DNNs and the contribution from domain specific knowledge, in form of feature engineering, is of foundamental importance when trying to optimize artificial intelligence solutions for their deployment on resource constrained devices. In particular, we targeted three main optimization goals: improve accuracy and speed; allow on-target network retraining; deploy the DNN on a microcontroller. As outlined in the paragraph below, for each optimization goal we considered a reference application.

The thesis is divided in the following chapters:

Chapter 2 provides the background of this work. Given the focus we have on Computer Vision solutions, this Chapter gives an outlook of the major milestones which in the CV field led to the advent of DL-based techniques, and in particular to Convolutional Neural Networks (CNNs). Then, the Chapter explains the basics of CNNs, with focus on the most common types of layers used. Finally, it is shown how CNNs are used in the CV field, with particular focus on feature extraction and object detection architectures, since they'll be widely used in the scope of this work.

In Chapter 3 we provide an analysis of the challenges to deploy DNNs on resource constrained devices, with focus on what determines their computational and memory footprint. Moreover, we give an outlook of already existing techniques to overcome the above challenges, considering both HW and SW-based ones. In this work we make use of off the shelf HW accelerators to run our tests, while we provide direct contribution to the category of SW-based solutions by performing optimization at DNN architecture level (number of layers, hyperparameters, etc.), doing feature engineering, and with application-aware algorithm design.

Chapter 4 presents the case study we considered for the first optimization goal: optimizing speed and accuracy. As a reference application we used an automatic embedded

detector of fast and small moving objects. In the chapter we show how performing feature engineering on the input image can lead to a dramatic reduction of the computational footprint of the solution, thus leading to achieve much better accuracy and speed with respect to state-of-the-art methods.

Chapter 5 is dedicated to the second optimization goal, that is allowing retraining of a NN-based solution directly on the target. Normally, in fact, the computationally intensive training procedure is carried out on a desktop or server environment, and then only the trained network is deployed to the edge device. Taking as a reference application a hand image classifier, we show how preprocessing the raw input image with an already trained and fixed neural network allows to dramatically reduce the portion of the network to be trained, thus allowing the procedure to be carried out on the edge device.

Chapter 6 targets the last optimization goal, which aims at deploying a DNN on a microcontroller. Differently from microprocessor based architectures, microcontrollers are designed with focus on low power consumption, reduced cost and weight. Therefore, microprocessors are extremely resource constrained, in particular with respect to the avilable memory, thus making the deployment of DNN on them a very challenging task. The reference application in this case is a person detection system for surveillance applications. In order to implement the solution on a microcontroller, we deeply analize the impact of neural network architecture choices on the memory footprint, thus obtaining the best compromise in terms of memory footprint and accuracy. Moreover, a handcrafted postprocessing of the neural network output is used to achieve the detection goal minimizing the overall computational burden.

In Chapter 7 we approach the same problem as in Chapter 6, but this time the optimized neural network is obtained with a semi-automatic approach based on Neural Architecture Search (NAS). In an evolutionary manner, a set of parent models generate, through small perturbations of the architecture, successor candidates, of which only the most performing ones are selected to take, in the next iteration, the place of lower performing parents. Given the target of deploying the solution on an embedded device, the concept of performance is defined using a custom loss function which considers both accuracy as well as computational cost of the given architecture.

Chapter 8 concludes our research on optimizing AI for embedded devices describing a route prediction algorithm for electric vehicles, that we patented, specifically designed to run on low resources microcontrollers. The algorithm makes use only of the residual battery energy, of the vehicle status (driving, recharging, standby), and of the vehicle position to build a graph of the habitual driving routes of the vehicle. This graph is

explored at each new driving cycle to identify if the current route corresponds to an habitual one, in order to provide an estimate of the energy needed until the next recharge, which would be very beneficial to on-board energy optimization algorithms. This contribution is complementary to the previous ones since considers a form of lightweight AI not relying on DNNs.

Chapter 2

Background

In this chapter we provide an outlook of the technologies and methods which form the underlying basis of our work. We will focus on applications of AI to computer vision since it's the main application field considered in this work.

First, in Section 2.1, we explain how a Deep CNN works, which are its main building blocks, and how does the learning mechanism work. In this section we also give an overview of the broad role of CNN in computer vision.

Following, in Section 2.2, we go through the most important CNN architectures for image classification, which introduce concepts that have been then reused by object detection CNNs.

Finally, in Section 2.3, we explain how a CNN can be used to localize and classify at the same time objects in images, according to two major families of architectures: Region based and single shot detectors.

2.1 Deep Convolutional Neural Networks

2.1.1 Introduction

As the name suggests, Deep Convolutional Neural Networks are a special type of Artificial Neural Networks (ANN) with the properties of being:

- **Deep**: Like in Deep Neural Networks (DNNs), this term refers to the number of layers between the input and the output of the network. Having a relatively high number of layers, DNNs are able to learn complex non linearities of the data, but

they require a higher amount of computational resources, which made them gain popularity only in the last 15 years.

- **Convolutional**: These networks process the input data, typically an image, by means of a sequence of convolutions with kernels (filters) which get learned during the training of the network.

Thanks to the convolution mechanism, CNNs preserve the locality of data, which is what makes them so powerful when applied to image processing. Differently, in classic ANNs, each neuron takes in input the output of all the neurons of the preceding layer. Accordingly, at the output of each convolution layer, we get a series of *feature maps*, which are image-like tensors resulting from the application of the convolution filters to the feature maps in input to that layer. The convolution makes each point of the output feature map a function of the corresponding point in the input feature map and a small set of its neighbours.

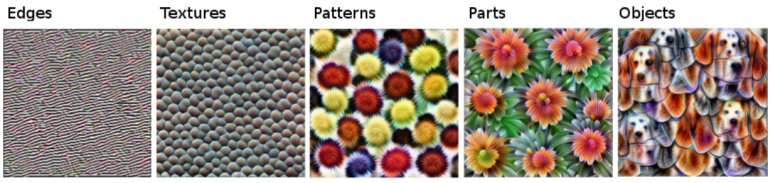


FIGURE 2.1: Type of features which the various layers of a CNNs learns to detect, from low level structures to high level, more abstract, information, according to the depth of the layer. Figure from [1].

CNNs process the image in a hierarchical manner, like shown in Figure 2.1. The first layers learn to detect low level structures of the image, such as edges; deeper layers contain neurons which get activated in the presence of higher level structures like textures or patterns; the last layers are sensitive to contextual information, such as the overall scene (streets, mountains, etc.). For this reason, CNNs for image processing tend to have a pyramidal shape, which means that, from the input to the output of the network, we see a reduction of the resolution of the feature maps (less focus on details) and an increment in the number of feature maps outputted by each layer (increased diversity of high level semantic information). Figure 2.2 depicts the CNN VGG16 as an example.

In practical applications, such as object classification or detection, CNNs are normally composed of two parts:

- A backbone, or feature extraction network, which processes the input image as explained above, with the aim of extracting relevant data from it;
- A head, which is the final part of the network that elaborates the features extracted by the backbone to produce the final output, such as a class prediction (in case of

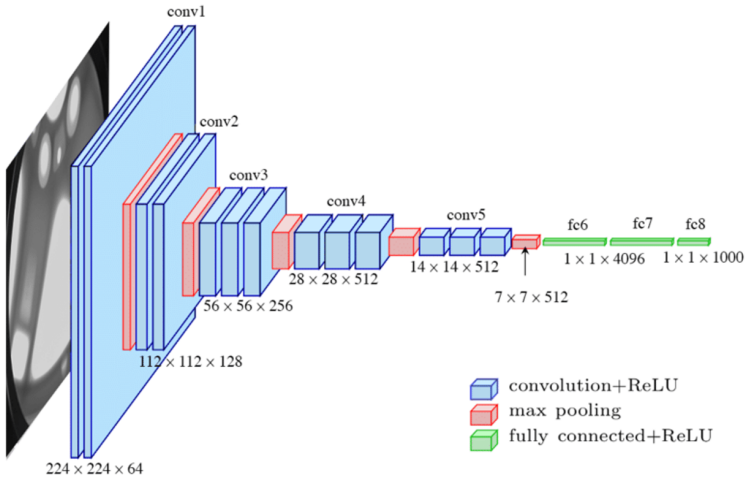


FIGURE 2.2: Schematic representation of the VGG16 CNN. Figure from [2].

image classification tasks) or object class and position (in case of object detection tasks).

In the previous VGG16 example, the green part is the classification head, while the rest of the network is the backbone. In the head, often fully connected layers are used.

In the next session an overview of the main layer types used in the backbone as well as in the classification head is given. It must be noticed that in this work we will not discuss topics common to the neural networks in general and which does not have a specific link with CNNs. As an example, these include activation functions, batch normalization, regularization methods, how training works, dataset splitting and others.

2.1.2 Main layer types

2.1.2.1 Convolution layers

Convolution layers are the main building block of convolutional neural networks. Given an input image patch P and a kernel K, both of size (m, n), the convolution of P with K, in the given context, is performed as:

$$P * K = \sum_{i=1}^{m} \sum_{j=1}^{n} P_{i,j} \cdot K_{i,j} \tag{2.1}$$

that is the sum of the product of the elements in the corresponding positions in the patch and kernel.

Normally, the input image I is bigger than the kernel K. In this case, the output image is computed sliding the kernel along all the positions in the input image and computing, per each of them, the convolution with the corresponding patch of the same size of the kernel. This case corresponds to the so called *stride* set to 1. A stride s greater than one means that the convolution is computed sliding the kernel every time by s positions along the image. As a consequence, the output image will be s times smaller than the input one.

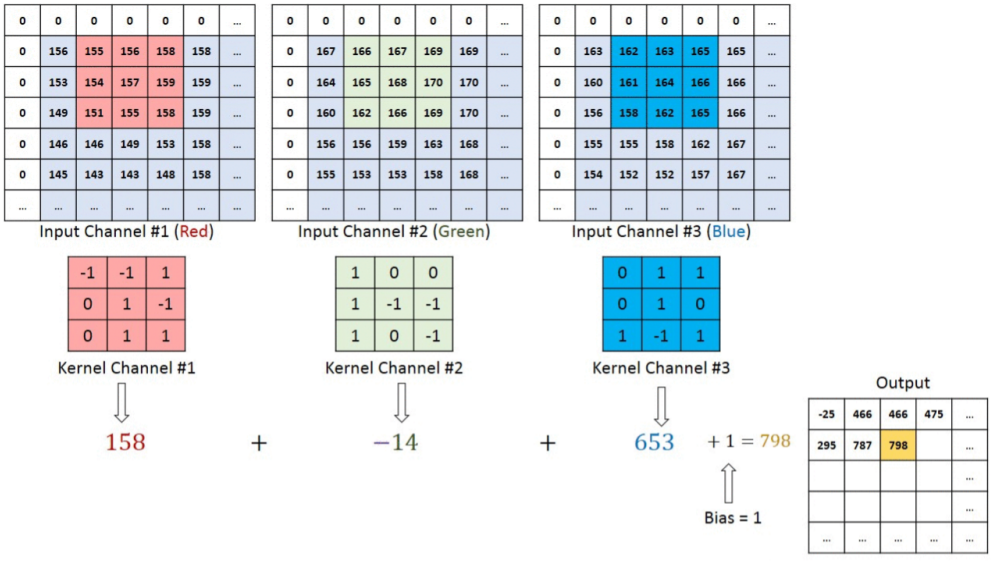


FIGURE 2.3: Graphical representation of the computation of a convolution layer. [3]

When the input image, or tensor, or feature map (according to the CNN dictionary) has multiple channels, like RGB channels in colored images, the kernel needs to have an equivalent number of channels, and the computation described above is simply extended as follows:

$$P * K = \sum_{k=1}^{c} \sum_{i=1}^{m} \sum_{j=1}^{n} P_{i,j,k} \cdot K_{i,j,k} \tag{2.2}$$

where c is the channel number. Following the same approach of classic ANNs, convolutional layers also consider a bias term b to be added to the result of the convolution, which improves the network's learning performance. Figure 2.3 shows a graphical representation of the explained computation. Notice that, in this case, a *padding* of zeros has been added to get an output of the same size (width and height) of the input.

It can be noticed that the so obtained output feature map has only one channel. Normally, in deep CNN architectures, a single convolution layer applies several different filters to the input image, thus outputting a multichannel feature map.

Important hyperparameters of the convolution layer are the kernel size (m and n, normally equal) and the stride s.

While training a CNN, the values of each filter get learned as well as the bias term.

2.1.2.2 Pooling layers

To optimise the available computational power and speed up the CNN execution, we need to forward through the network only the dominant features of each feature map. Pooling layers are designed to this purpose. As shown in Figure 2.4, a pooling layer reduces the dimension of a feature map by converting a region of adjacent numbers in the input feature map (orange square) into a single number of the output feature map, preserving the relative position in the map.

There are two types of pooling layers: Average pooling and Max pooling. In the first one, the output value is computed as the average of all the input values in the relative region, while, in the second one, it is computed as the maximum of the same values.

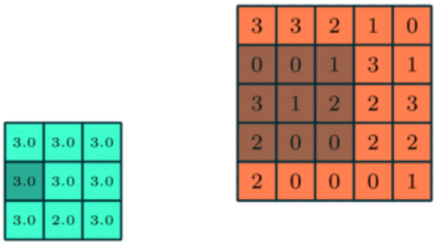


FIGURE 2.4: Graphical representation of the computation of a pooling layer.

2.1.2.3 Fully connected layers

Fully connected (FC) layers date back to the very origins of ANNs, and are not a peculiarity of CNNs. In fact, these layer do not perform any form of image manipulation through filtering, like convolution layers do. Instead, in CNNs they're normally used to learn, from the feature maps outputted by the convolutional part of the network, how to classify the image (in image classification tasks) or an object in the image (in object detection tasks). Figure 2.5 gives a conceptual representation of the role of FC layers in a CNN.

In fully connected layers each neuron computes its output as the weighted sum, linear combination, of all the input values (normally the result of flattening the last feature map tensor), plus a bias term. All the weights and the biases are trainable parameters.

When fully connected layers shall output an image classification label, they normally have a number of neurons in the last layer equal to the number of possible classes of the image. Then, the activation (output value) of each of these last neurons is an indicator of the likelihood of that class for the given input image.

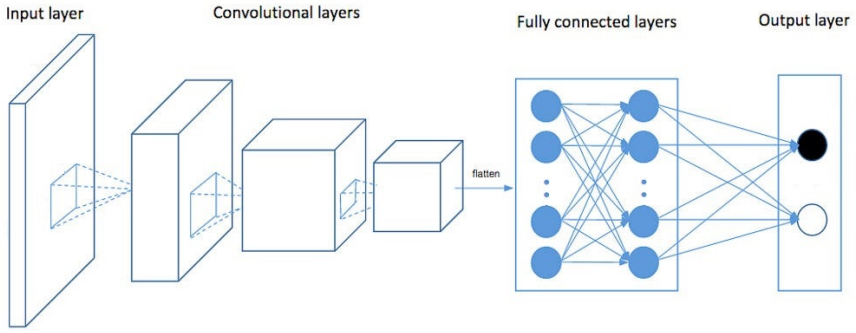


FIGURE 2.5: Graphical representation of the fully connected layers in a CNN.

2.2 Common feature extraction CNN architectures

As anticipated, in applications such as image classification and object detection, CNNs are normally built by two main parts: a backbone, or feature extraction network, and a head. In this section we give an overview of three of the most common feature extraction networks based on CNN which can be found in the literature.

The aim of a feature extraction network is to process the input image via a series of NN layers, most of which are convolutions and pooling layers, to extract meaningful feature maps which can then be the input of the classification or object detection part of the network, according to the specific application.

Although many other network do exist which achieve similar targets, the three networks here explained have been selected since they are recognised by the research community as important milestones and they introduced concepts which inspired a lot of other relevant works. Moreover, implementations of these architectures can be easily found with open access and, therefore, they are often taken as a reference for comparison with modern works.

2.2.1 VGG

VGG was proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" [15]. The name is the acronym of Visual Geometry Group, the department the two

researchers worked in. It became famous thanks to the improvement it achieved with respect to previous CNNs, AlexNet in particular [16], in the image classification competition ILSVRC-2014, where it achieved 92.7% accuracy in the ImageNet dataset.

The key idea behind VGG was simply to leverage the same strategy that made AlexNet improve over the previous LeNet: using a deeper and bigger network.

The structure of VGG16 (one of the two variants of the VGG architecture, as proposed by the authors) is depicted in Figure 2.2. It is composed of several blocks, where each block has a sequence of convolution layers followed by a max pooling layer. At each convolution, the output feature map preserves the size of the input one. The resolution is instead decreased by two at the level of the 2x2 max pooling layer, which uses a stride of 2.

All the convolutions in the network use a kernel of size 3×3. This choice was the result of investigations carried out by the authors, which proved that several convolution layers with smaller kernel size give better results than fewer convolution layers with larger kernels, such as 5×5. Using 3×3 kernels has become a standard de facto in subsequent works and is still the preferred choice in CNNs.

Since the network was initially designed for image classification, at the end of the network it uses a sequence of three fully connected layers to predict the likelyhood of each of the 1000 classes of the ImageNet dataset. However, this part of the network is task specific and can be considered the *head* of the *feature extraction network* which comes before it.

If the main thing which makes VGG so famous is its simplicity and modularity (VGG16 and VGG19 are just two instances of the architecture), on the other side it has the drawback of being very heavy, not suitable to resource constrained devices, and log to train. As an example, VGG16 counts 138 M and, at its origin, it took weeks to train it on ImageNet.

2.2.2 ResNet

Although both AlexNet and VGG successfully adopted the concept of increasing the number of layers to improve the performance of the CNN, the effectiveness of this technique saturates when considering architectures even deeper than VGG19. As proven in [17], simply adding layers may also decrease the performance of the CNN. The simple explanation relies in the phenomenom of vanishing or exploding gradients due to the high amount of computations an non linearities between the input and the output of the network, which make training the neural network less effective.

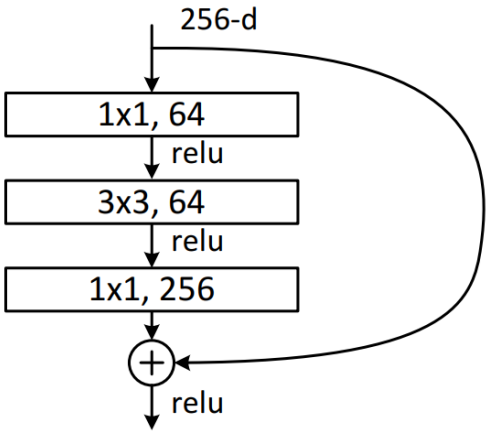


FIGURE 2.6: Bottleneck block and skip connection, as used in ResNet-50 -101 and -152. The rectangles represent convolutions, and specify the filter size and the number of output channels.

To overcome the problem, Microsoft researchers He et al. in 2015 proposed ResNet [17], which introduced the concept of skip connections (also known as shortcut connections or residuals) while building deeper models. Figure 2.6 shows how they are used in ResNet-50 (one of the versions of ResNet) to form the so called bottleneck block, which will also be used in our contributions. Skip connections are parallel neural network branches that let a feature map (input of the bottleneck block) "jump over" convolution layers being directly added to the feature map in output from the bottleneck block.

These skip connections have a twofold positive effect: first, they contribute to the solution of the vanishing gradient problem, since they represent an alternate shortcut for the gradient to pass through; second, they make it easier for the model to learn an identity function. This ensures that the higher layers of the model do not perform any worse than the lower layers.

As a result, ResNet, which is composed mainly by a series of these bottleneck blocks, can be at the same time much deeper than other networks (in particular his ResNet-101 and ResNet-152 variants) while remaining easy to train and respectful of computational complexity. Moreover, ResNet is efficient from a computational and memory point of view with respect to VGG, as shown in Table 2.1.

	# Params	**Bln FLOPs**	**Top-1 Err Rate [%]**
VGG-16	138 M	15.3	28.07
VGG-19	144 M	19.6	28.07
ResNet-34	21.5 M	3.6	24.52
ResNet-50	23.9 M	3.8	22.85

TABLE 2.1: Comparison between VGG and ResNet in terms of number of parameters, computational complexity and error rate in the ImageNet dataset. From ResNet paper.

2.2.3 Inception

Rather then being a single CNN, the Inception [18] one is a series of networks designed by researchers at Google between 2014 and 2016. These networks share the principle of substituting the approach of building CNN simply stacking convolutions in series with a more engineered approach aimed at improving the feature extraction capabilities of the network and its speed.

The main versions of the Inception series are Inception v1, v2, v3, v4 and Inception-ResNet. Here, we will not go through all the details of these versions, but we will give an overview of the key ideas behind its success, which is what is also relevant in the scope of our work.

The key idea the Inception CNN was built on is that the huge variation in the location of information in an image (target object size variation, for example) makes it hard to define the right filter size for convolution layers. Although in part the problem can be overcome by using deeper networks, these are more prone to overfitting, are harder to train, and are computationally expensive.

The solution of the Inception network to this problem is to make the CNN "wider" instead of "deeper". Therefore, the network is built by stacking blocks (Inception modules) which contain parallel (instead of sequential) convolutions on the same input, and then concatenate all the results of these operations. The parallel computational paths process the input in different ways, thus making the network "wider".

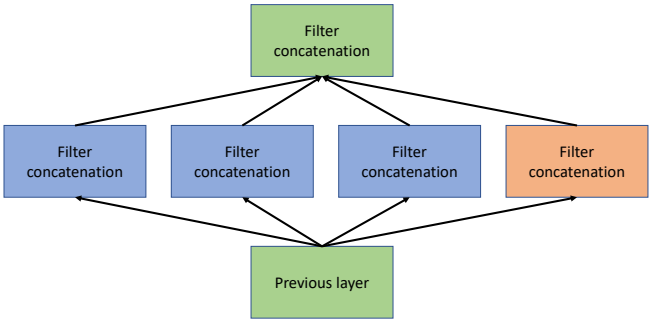


FIGURE 2.7: Conceptual view of an Inception module used in Inception CNNs. The input is processed in parallel computation paths which apply convolutions with different kernel size. Then, the outputs are concatenated.

Figure 2.7 shows how an Inception module works conceptually, with different filter sizes applied to the same input tensor. In practice in the various versions of the Inception network, this concept is implemented by exploiting several optimizations, the most important of which are:

- 1×1 convolutions are added in front of convolutions with a kernel bigger than 3×3, to reduce the number of channels and improve speed.
- Convolutions with a kernel bigger than 3×3 are factorized into sequences of 3×3 convolutions.
- 3×3 convolutions are factorized in parallel 1×3 and 3×1 convolutions.
- Factorized 7×7 convolutions are added
- The residual mechanism introduced in ResNet is introduced to improve learning.

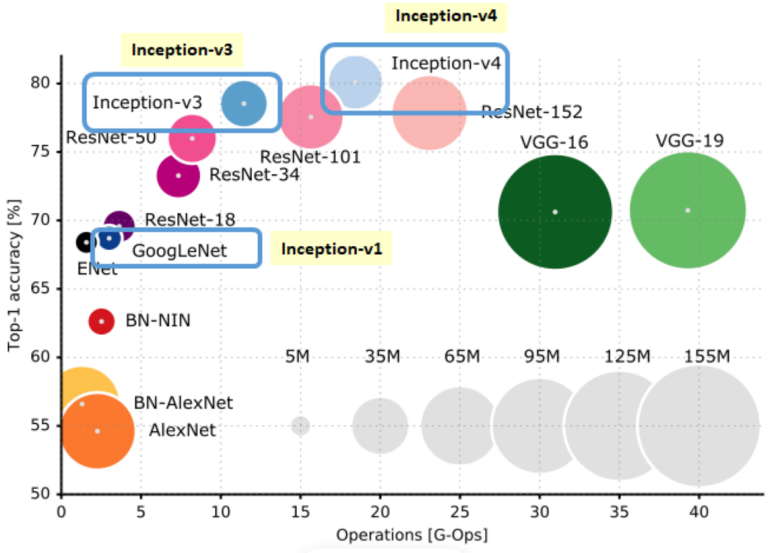


FIGURE 2.8: Positioning of the inceptions models with respect to other deep learning models in terms of accuracy, number of operations and number of parameters. Figure from [4].

As we will see in the next chapters of this thesis, the two main lesson we learned from the Inception series are: (i) the effectiveness of processing the same input with different convolution layers, each designed to be more sensitive to specific features; (i) the power of "engineering" neural networks rather than simply stacking convolution layers.

Figure 2.8 gives an overview of the computational complexity, accuracy and number of parameters of the different versions of Inception, ResNet, VGG and other feature extraction networks, when applied to the image classification task on the ImageNet dataset.

2.3 Object detection CNN architectures

In the previous section we have gone through three CNN architectures for feature extraction. As anticipated, these CNNs can be used as backbone architectures for several

computer vision tasks, such as image classification and object detection. The difference between one task and another is done via the network head, which is the last part of the CNNs, that processes the extracted features to produce the final output.

In the case of image classification tasks, the classification head is usually made up of fully connected layers, whose last layer has a number of neurons equal to the number of possible prediction classes.

In the case of object detection, the network head has to predict the coordinates of the bounding boxes around the discovered objects, and the object class. To achieve this task two main approaches exist: Region based (or two-stage) detectors and Single shot (or single-stage) detectors.

2.3.1 Region based detectors

Region based object detection CNN, also known as Two-Stage Detectors, process the input image in two steps: in the first step, regions of the image which are likely to contain an object of interest are relected; in the second step, each of these regions gets analyzed to produce the final object class and position in the image, as the coordinate of its bounding box.

The first region based detector was R-CNN , proposed in 2014. R-CNN gave birth to the R-CNN series, composed by the first R-CNN [19], and the two successors Fast R-CNN [20] and Faster RCNN [21]. Although other important region based detectors exist, such as Mask R-CNN [22] and FPN [23], in this section we focus on the R-CNN series, since it's a good reference to analyse the mechanism behind region based detectors, and since they're often used as a benchmark for object detection algorithms, given their notoriety in the this research field. Figure 2.9 gives a clear view of the structure of each of the networks in this family.

R-CNN In R-CNN the input image gets first processed by a "region proposal" algorithm, whose aim is to select regions of the image that *could* belong to a particular object. The authors used the selective search algorithm, which works by generating segmentations of the image which appear to be objects. It does so by looking at shapes, color and texture, independently from the actual target objects of a specific implementation of the R-CNN object detector. Selective search generates 2000 region proposals category-independent.

After this stage, each of the 2000 region proposals is reshaped into 227 x 227 pixels images, which are then processed independently from each other by an Alex-Net inspired feature extraction network, thus generating a 4096 dimensional feature vector for each

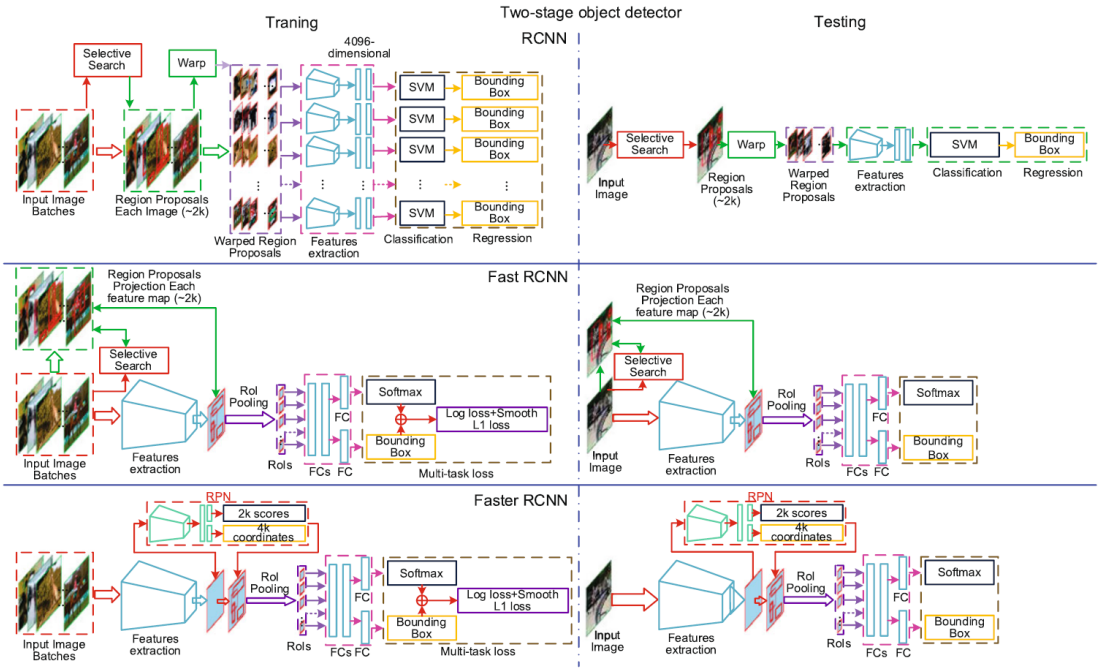


FIGURE 2.9: Diagram of R-CNN, Fast R-CNN and Faster CNN, highlighting the differences between the three methods and between the training and testing (inference) phases. Figure from [5].

region. The feature extraction network is first trained on the ILSVRC2012 classification dataset, and then fine tuned on the specific target object classes. During training. a classification head is added to the feature extraction network, just for training purposes, then removed.

The final stage has to output the position on the object in the region (bounding box regression) and its predicted class.

The bounding box regression is achieved by learning, in a least-squares fashion, the parameters of the transformation function from the last feature map of the feature extraction network to the bounding box coordinates, minimizing the error with respect to the ground-truth bounding box.

The classification is achieved via an individual SVM (Support Vector Machine) classifier for each class.

Fast R-CNN The main disadvantage of R-CNN is that, a shown in Figure 2.9, each region passes through the feature extraction network independently. This is highly inefficient since overlapping patches of the proposed regions gets processed multiple times in the same way. Fast R-CNN addresses this problem by passing the input image only once through the feature extraction network, thus producing an unique output feature map. The region proposals obtained through selective search are mapped from

regions of the source image to the corresponding regions of the computed feature map, so that no recomputation is needed.

Faster R-CNN Faster R-CNN improves even more by substituting the Selective Search algorithm with a much faster region proposal network, directly applied on the output of the once-computed feature map.

Despite all these optimizations, these Region Based Detectors share the need of looping over the region proposals. Although this aspect in principle might guarantee higher accuracy, on the other hand it dramatically affects computational time.

2.3.2 Single shot detectors

Single shot (or single stage) object detectors are able to perform object detection passing the input image only once through the computations of the neural network. In other words, redundant computations over the same portion of the image (due to regions overlaps) are avoided. This makes single shot detectors much faster than region based ones, which is the main reason why they're of great interest for embedded and real time systems, such as in the autonomous car field.

Among Single Shot detectors, we have two main families: YOLO [24] and SSD [6], which will be briefly described here below.

YOLO The first version of YOLO was proposed in 2015 by Redmond et al. Its name stands for You Only Look Once, underlying its nature of a single stage detector. As shown in Figure 2.10, the YOLO network processes the input image first using a simple feature extraction CNN inspired from VGG, which takes the name od Darknet. Considering a 448x448 input size (according to the original paper), at the output of the feature extraction network we have a 7x7 feature map with 1024 channels. Since CNNs preserve data locality, this feature map can be seen as a 7x7 grid, where each cell of the grid corresponds to a patch in the input image.

Through two fully connected layers, for each cell the network infers the parameters (position and size) of B bounding boxes, the "objectness" (likelyhood of containing an object) of each bounding box, and a likelyhood value for each of C possible classes. Therefore, at the output of the fully connected layers, the network gives a SxSx$(5B+C)$ tensor, where S is the grid size (7 in the original paper).

Table 2.2 outlines the results obtained in [14] comparing Faster RCNN with YOLO on a combination of the VOC2007 and VOC2012 datasets. YOLO proved much faster than previous approaches, thus preserving comparable accuracy (even thou lower on average, when considering many other datasets).

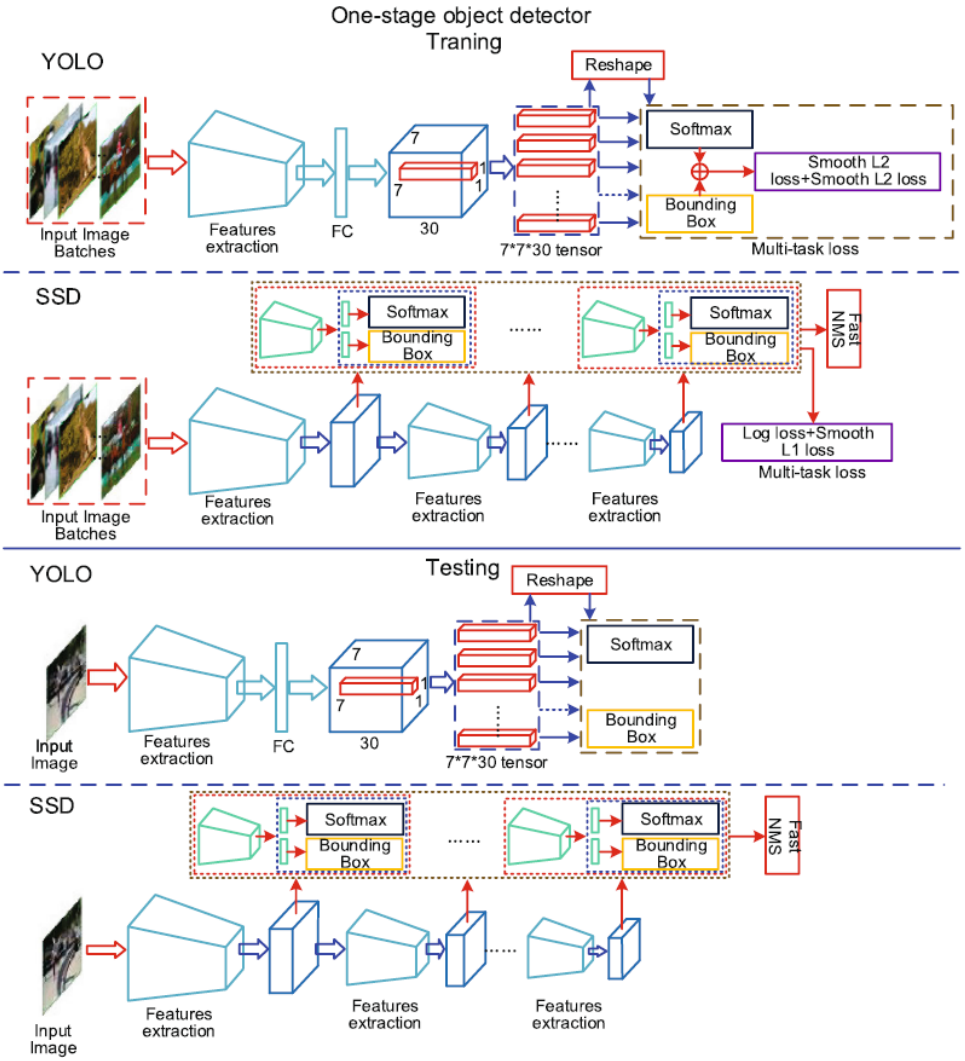


FIGURE 2.10: Diagram of YOLO and SSD, highlighting the differences between the two methods and between the training and testing (inference) phases. Figure from [5].

Method	Backbone	mAP	FPS
Faster RCNN	AlexNet	62.1	18
Faster RCNN	VGG16	73.2	7
YOLO	DarkNet	63.4	45
YOLO	VGG16	66.4	21

TABLE 2.2: Comparison of Faster RCNN with YOLO on a combination of the VOC2007 and VOC2012 datasets [14].

However, as mentioned in the original paper, the first version of YOLO had at least two important shortcomings:

- Since the bounding box regression and object classification are performed starting from feature maps taken at the end of the network (where low level details have been filtered out by the preceeding CNN layers), YOLO does not perform well in the detection of small objects.

- The network finds it hard to detect objects it has been trained for, when they appear in the test set with an aspect ratio different from the training set.

Several other versions of YOLO have been designed in the years following its publication. Relevant versions are YOLOV2 and YOLOV3 [25]. The main improvements to the original YOLO are:

- Batch normalisation has been added, thus increasing the networks average precision.
- The *anchor box* mechanism has been added, thus allowing the network to detect inside the same cell multiple objects at different scales and aspect ratios.
- A new 106-layers network which makes use of residual blocks is used in YOLOv3, and features for performing object detection are extracted at multiple levels in the network, thus capturing both low level details (for detecting small objects) and high level contextual information.

SSD The Single Shot Multibox Detector was proposed in 2015 by W. Liu et al. Learning from the shortcomings of its predecessor YOLO (v1), SSD was designed to address the problem of multireference (detecting multiple objects closed to each other) and multiresolution (detecting small as well as big objects). As shown in Figure 2.10, the multiresolution feature was built by using VGG16 (with modified convolutional blocks in the last layers) as feature extraction network, and performing detection at multiple levels of the network, using from high to low resolution feature maps.

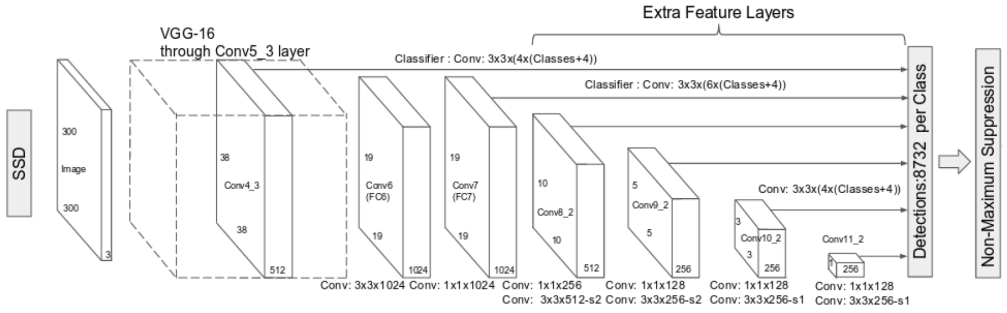


FIGURE 2.11: Architecture of the Single Shot Detector (SSD).Figure from [6].

In particular, 6 feature maps at different resolution are selected from the backbone network. Each of these is then processed by a last convolutional block which works as a classifier by using a 3×3 kernel with a number of output channels equal to $B\text{x}C + 4$, where (following the same mechanism seen in YOLO) B is the number of anchor boxes and C is the number of object classes. In other words, as it can be seen in Figure 2.11,

for each feature maps 4 to 6 boxes are computed, and for each of them the likelihood of each object class is computed. Note that the position of the boxes is inferred as 4 parameters, related to the height, width, horizontal and vertical position of the box with respect to the cell (position in the feature map) they're positioned in.

As the reader might have noticed, since the third version of YOLO also implements the anchor boxes and multi-resolution mechanisms, the differences with SSD are minimal, and mostly in the backbone network.

The simplicity of SSD made it one of the networks with the highest number of open source implementations available in the internet, therefore being often used as a reference for high speed object detectors, together with YOLO. As we will see in this work, one of its variants uses the lightweight MobileNet backbone network to boost the detection speed.

Chapter 3

Embedding Deep Neural Networks

After the presentation given in the previous chapter of the basics of DNN and the main methods to apply them to image classification and object detection, in this chapter we move forward exploring the challenges and related methodologies related to the problem of bringing Deep Learning based CV solutions to resource constrained embedded systems, which is the focus of the present research work.

The chapter is structured as follows: in the first section, we discuss about the importance of being able to run Deep Learning based CV algorithms on embedded devices, limited in weight, space, computational and memory resources, and we give an outlook of modern applications of these implementations; in the second section we explore the challenges related to embedding these solutions, analysing their computational and memory requirements, with respect to the available resources on embedded devices; in the third section we provide the most common methods used at software level to allow the execution of DNNs on embedded systems; finally, in the fourth and last section, we explore the complementary solutions, this time at hardware level, aiming at the same goal.

3.1 The need for embedded DNNs

As the name suggests, Deep Neural Networks are a form of neural networks characterized by a relatively high amount of layers. As seen in the previous chapter, they started to be adopted in applications, such as computer vision, for their superior performance with respect to previous approaches. Seeking for algorithms capable of solving, for the first

time, problems such as object recognition with human-comparable accuracy, the research community did not initially focus on the computational and memory implications of such DNNs.

Once that works such as AlexNet proved the above mentioned problems to be effectively approachable by increasing the number of layers of neural networks ("going deep"), a whole new era started, where researchers in academics mostly focused on achieving better and better accuracy.

At the same time, the industry started to gain interest in making this kind of DNN based CV algorithms available on embedded devices. Great examples of fields which drive this interest are:

- **Industry 4.0**: In industrial applications embedded computer vision is starting to be used for example to design collaborative robots, that use vision to identify and grab objects, or to receive motion commands from the operator. Vision-enhanced robots can also be used to identify problems in fabrication processes (visual inspection).
- **Autonomous robots**: many autonomous robots make use of computer vision to understand the environment around them and be able to autonomously navigate it. Modern commerically available drones, for example, exploit computer vision to avoid obstacles or to recognise and follow a target, such as the operator.
- **Healthcare**: Computer vision is being implemented in machines for the analysis of visual data such as ultrasounds images or radiographies. A practical example is the help these algorithms can give to the medical equipe in identifying tumor masses.
- **Mobile phones**: Mobile phones are one of the major fields of application of embedded CV. Applications range from face recognition for security purposes, to virtual reality, passing through techniques to intelligently improve the quality of photographs taken with the device itself. It must be said that although mobile phones are becoming increasingly powerful, techniques to optimize the execution of DNNs on these platforms are still crucial, in particular due to the need of optimizing battery consumption.
- **Automotive**: The automotive industry is also an important player in this field, thanks in particular to the search in autonomous driving and, more in general, in Advanced Driving Assistance Systems (ADAS). These technologies are in fact based on the ability of the vehicle to see and understand the environment around it and inside it (for example in the case of driver drowsiness detection systems).

Sending the raw vision data to a remote server for processing, and waiting for the output of the computation, is a solution that an embedded system may adopt to overcome the lack of computational and memory resources on the device itself. However, this work targets the applications where executing CV algorithms on the edge device is crucial. This need may come for several reasons, above which:

- **Reliability**: Streaming vision data to a remote server requires a reliable communication channel which cannot always be given for granted. Using for example Wifi or the cellular network to reach internet servers would not be suitable for critical applications.
- **Low latency**: in many applications such as autonomous cars, computations on the visual data need to be carried out in real time, that is in a few milliseconds. The additional latency introduced by the need of sending and receiving the data to and from a remote server would never be acceptable.
- **Privacy**: vision data might be confidential and transferring it to remote server might expose to the risk of data leakage.
- **Network bandwidth**: in particular when processing high resolution videos, the bandwidth offered by the communication channel might not be sufficient to transfer the needed data in a reasonable amount of time. Moreover, availability of higher network bandwidth is normally associated with additional costs.
- **Power consumption**: transmitting data remotely, in particular using cellular networks, is power demanding, and for battery powered edge devices using more power leads to lower autonomy.

3.2 Challenges

Deploying Convoutional Neural Networks on resource constrained embedded devices represents a challenge in particular due to their high memory requirements and computational burden. Most microcontrollers for embedded applications, in fact, feature up to a few hundreds of KBytes of memory, and have a clock speed in the order of the MHz instead of GHz. If on one side lower computational resources lead to longer inference time (therefore mainly impacting time critical applications, such as autonomous driving), on the other side the lack of sufficient memory can completely prevent the execution of the network.

In the next two subsections we give an outlook of the reasons behind these two types of need, whose comprehension at the basis of techniques aimed at making CNNs executable on edge devices.

3.2.1 Memory requirements

CNNs, and neural networks in general, use memory for two main reasons: storing the parameters (or weights) of the network, and storing the activations (or feature maps) at the output of each layer.

Since the main building blocks of CNN, in particular in the CV field, are convolutional layers and fully connected layers, we put our focus on these.

3.2.1.1 Parameters

The parameters represent the learnable (or trainable) part of the CNN. Considering a convolutional layer with a kernel of size kxk, C_o output channels and applied to an input feature map of C_i channels, the total number of parameters required for this layer is

$$P_{CL} = k \times k \times C_i \times C_o + C_o \tag{3.1}$$

Notice that the last term added in equation 3.1 comes from the scalar bias added to the result of the convolution of the input feature maps with the kernel, and is different for each output channel (see Section 2.1.2.1). Assuming a 4 Bytes floating point representation in memory of these parameters, the final memory footprint would be $4P$.

It must be noticed that the final number or parameters does not depend on the height and width of the input and output feature maps.

Figure 3.1 shows an example of computation of memory footprint in a fraction of the VGG network.

In the case of fully connected layers (refer to Section 2.1.2.3) the number of parameters is simply given by

$$P_{FC} = C_i \times C_o + C_o \tag{3.2}$$

Since fully connected layers are often used at the end of a series of convolution layers, the last feature map of the convolutional sequence is flattened, so that each element

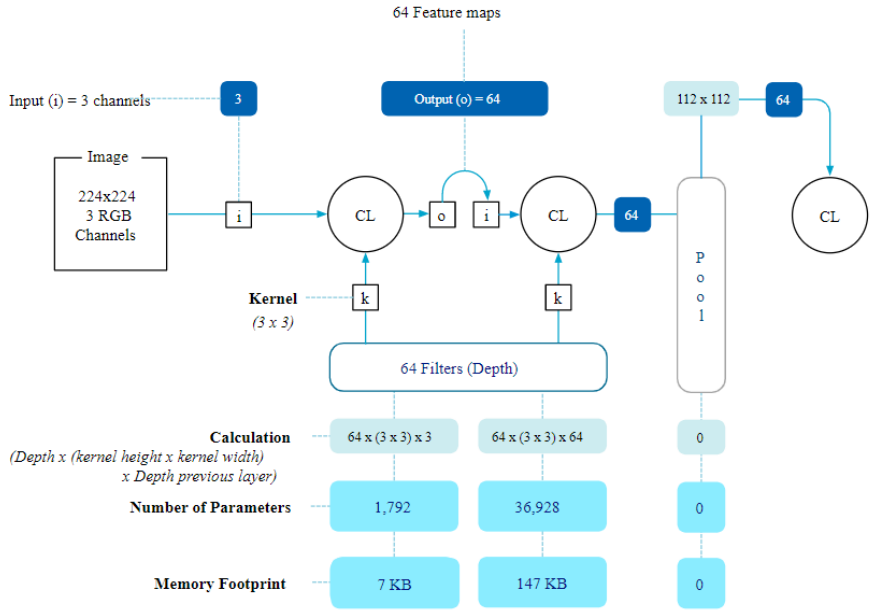


FIGURE 3.1: Example of computation of the number of parameters in a fraction of the VGG architecture. [7]

of the feature map gets mapped to one of the C_i input neurons of the following fully connected layer. Therefore, in this context, the height and width of the final feature map affect the total number of parameters of these layers.

3.2.1.2 Activations

The activations, or feature maps, require memory to store the output of a layer and make it available for all the following operations that operate on that feature map. The first CNNs were often designed with one single data path where a feature map is used only by the directly following layer. However, modern CNNs usually make use of "shortcuts", where a feature map is not only used by the directly succeeding layer, but is also taken in input by deeper layers. This case requires a feature map to be kept in memory for longer, thus increasing the total memory footprint.

The amount of memory required to store a feature map can be simply computed from the size of the feature map itself. Considering a feature map of height H, width W, with C channels and assuming 4-bytes floating point representation, the required memory, in bytes, would be

$$M = 4 \times H \times W \times C \tag{3.3}$$

In the case of fully connected layers, the memory required to store the activations would be simply equal to 4 times the number of output neurons, always assuming floating point representation.

It is important to note that, differently from the parameters' case, the amount of memory in CNNs depends on the width and height of the feature maps, and therefore, on the resolution of the input image.

Another important difference is that while parameters need to be always kept in memory, activations store only temporary results, thus allowing the corresponding memory to be released once these values have been processed by all the succeeding layers. In practice, this means that the overall memory footprint for the activations depends on the maximum number of activations that need to be kept in memory at the same time, rather than on the sum of all the activations' memory footprint.

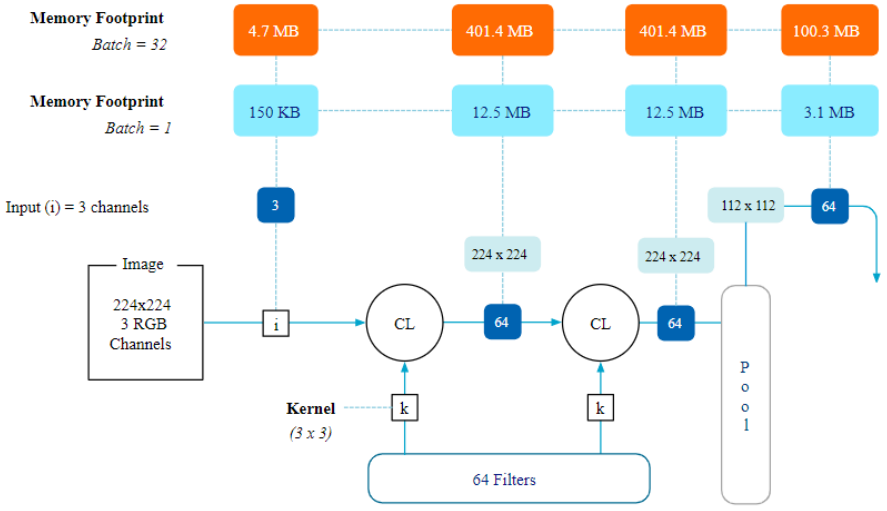


FIGURE 3.2: Example of computation of the activation's memory footprint in a fraction of the VGG architecture. [7]

Figure 3.2 shows an example of computation of the memory footprint in a fraction of the VGG network.

3.2.1.3 Difference between training and inference

In the above calculations we focused on the memory requirements to perform a forward propagation, that is inference, on a CNN. This is also what matters most in designing embedded CNN, since the training of the network is normally done only once during development, and can be performed on a powerful server.

Training a CNN, and a DNN in general, is in fact much more computationally demanding. To train a CNN, in addition to the forward propagation, an additional process called backward propagation (or backpropagation) is needed. The need for a backprogation is related to the fact we're doing supervised learning, that is we train the network to produce, given a training set of images, outputs whose true value (for example the image

label) is known. While training the network, the error in the CNN's output resulting from a forward pass on the train dataset is used by this backpropagation step to update the weights and biases of the network to improve the prediction, or "to learn".

Since, starting from a prediction error, all the weights which contributed to this error are updated according to their "contribution" to the error, that is according to the partial derivative of the error (loss) function with respect to that weight, partial derivatives for all the weights need also to be stored in memory during network training, and this dramatically increases the memory requirements. Moreover, to compute these derivatives, also all the activations computed in the forward pass need to be kept in memory.

The usage of training optimizers like ADAM, further increse the memory requirements, requiring previous gradients to be kept in memory.

Finally, effective training requires batches, which means performing the forward propagation on a set of images (usually 32) at the same time. Therefore, the batch size is an additional factor which has implications on the total memory footprint for training.

As will be explained in the next chapters, in this work we have also experimented training a neural network on a resource constrained embedded device for specific applications.

3.2.2 Computational burden

A practical way to get an idea of the time a neural network needs to perform an inference is to count the number of computations it has to perform and compare it with the capabilities of the HW platform it will run on. The number of computations is typically counted as FLOP or MACCs. FLOP stands for floating point operations, and is more commonly used with a final S (FLOPS) indicating "per second" when referred to the HW computational power. MACC instead stands for multiply-and-accumulate operations.

MACC is a unit frequently used in the neural network field, since most computations in a neural network are actually *dot products* and therefore implemented as a series of multiply and accumulation operations. For simplicity, it is usually considered a dot product between two vectors of size n to use n MACCs, which are equivalent to $2n - 1$ FLOPs, since there are n multiplications and $n - 1$ additions.

Considering for example the case of a fully connected layer with C_i input neurons and C_o output neurons, the computation that this neuron performs can be written as in equation 3.4.

$$y = Wx + b \tag{3.4}$$

being $y \in \mathbb{R}^{C_o}$ the output array, $W \in \mathbb{R}^{C_i \times C_o}$ the matrix of weights, $x \in \mathbb{R}^{C_i}$ the input array and $b \in \mathbb{R}^{C_o}$ the array of biases. Following the above definition, the total number of MACCs for the fully connected layer is $C_i \times C_o$. Conveniently, the final addition of the bias is not considered since actually the dot product "misses" one addition.

In the case of a convolutional layer, the input and output are not arrays, but three-dimensional feature maps. Considering a square kernel of size K, and assuming C_i input channels and an output feature map of size $H_o \times W_o$ with C_o channels, the total number of MACCs would be $K \times K \times C_i \times H_o \times W_o \times C_o$. Interestingly, this number does not depend on the spatial (height and width) of the input feature map, but only on that of the output. For this reason, using a *stride* higher than one, thus compressing the resolution of the feature map by the same amount, dramatically decreases the number of MACC for the layer.

Although convolutional and fully connected layers are not the only parts of the network which require computations. Batch normalizations and activations functions do also use computing power. However, convolutional and fully connected layers account for the very majority of the MACCs. Therefore, the parameters involved in the above computations must be considered to design lightweight neural networks suitable for deployment on resource constrained devices.

3.3 Embedding DNNs: Software

Given the challenges explained in the previous section, in the recent years many researchers focused on methods to make Deep CNNs executable on embedded devices. In practice, the goal is to make them lighter in terms of memory and computational burden. Solutions have been identified both working at software and hardware level. In this section, we focus on software level, while the hardware level is treated in section 3.4

We splitted techniques at software level into two subcategories: Architecture design level and Architecture implementation level.

3.3.1 Architecture design level

Optimizing CNNs at architectural level means trying to find different, lighter, configurations of the CNN (in terms of number of layers, types of layers, hyperparameters) while trying to preserve as much as possible the performance of a heavier parent CNN which meets the application needs.

Three main methods are considered in this context: Knowledge distillation, Neural Architecture Search and Feature engineering.

3.3.1.1 Knowledge distillation

First introduced by Bucilua et al. [26] in 2006 and later generalized by Hinton et al. [27] in 2015, Knowledge distillation is a method to compress DNN (including Deep CNNs) where a smaller model gets trained to mimic a pre-trained larger model. This method is also called the "teacher-student", where the larger model is the teacher and the smaller one is the student.

The idea behind this concept relies on the fact that in order to be able to learn hard non linearities (such as, in the image classification task, the function mapping an input image into its class) while training we need to use much deeper neural networks than what would be actually required to implement the function that the neural network has learnt in the end. One of the reasons of this phenomenon is that a DNN is intrinsically capable of learning a generalisation of the knowledge extracted from a training set, thus being able to apply it to unknown samples. However, considering for example the image classification task, while training, samples which are similar to each other, but belong to different classes, are "taught" to the network like if they where completely different, thus requiring higher effort for the network to learn it.

As an example, Figure 3.3 shows the number 3 from the MNIST dataset. In training data, the number three is encoded in a one-hot-vector as 0001000000, thus loosing the information about its similarity with the number 8.

FIGURE 3.3: Number three from the training set of the MNIST dataset.

In Knowledge distillation, the simpler student model is directly trained on the "soft predictions" (that in this training context work as labels) learnt by the more complex teacher model on the training dataset. Soft predictions are more effective than the original hard predictions (like one hot encoding in the MNIST dataset) of the original dataset, since they preserve the information about similarity of data, as learned by the teacher DNN. In practice, these soft predictions are obtained via a variant of the softmax layer, which, through a parameter T called temperature, sets the balance between a one-hot like output, and a more equally distributed output among classes. In other words,

when the temperature is set to $T = 1$, we get a standard softmax function which tends to increase the probability difference between the correctly predicted class and the others, while when the temperature is set to higher value, the distribution of the predicted probabilities for each class gets increasingly flattened, thus better preserving the learnt similarities between the classes.

Figure 3.4 shows a graphical representation of the concept. According to the original paper, as shown, the student model is actually trained considering an overall loss function which takes into account, in a weighted manner, both the teacher's soft output and the original dataset.

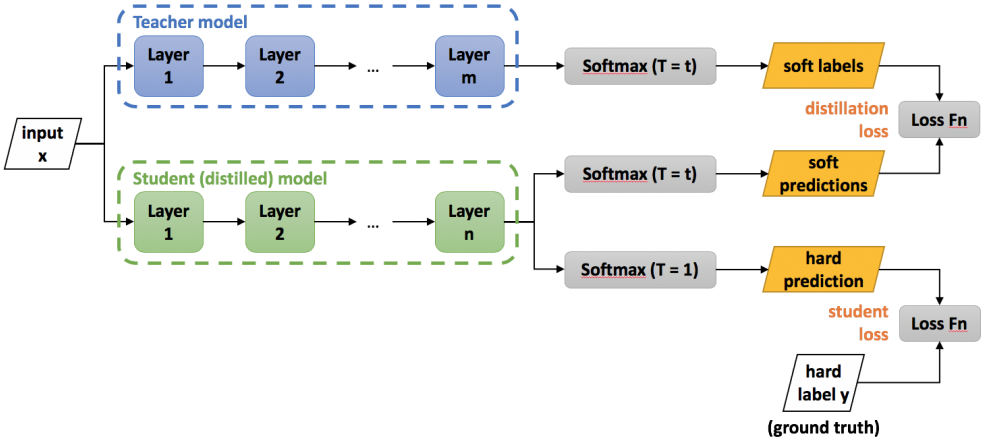


FIGURE 3.4: Graphical representation of the teacher-student training model.

3.3.1.2 Neural architecture search

Designing an effective DNN architecture for a given task requires a mixture of application specific knowledge and experience with the many options this design activity involves. However, predicting the performance of a DNN architecture without a test campaign is still very hard also for experienced people. For this reason, trial and error is still a widely used approach, even when it's driven by experience. Since evaluating the performance of a new architecture trial is a computationally demanding task which might require up to days or weeks, the recent increase of need for application specific high performing (in accuracy and/or efficiency, like in embedded scenarios) DNNs has encouraged the research for automated neural architecture search methods (NAS). NAS date back to 2016, when Zoph and Le used reinforcement learning algorithms to successfully achieve state-of-the-art architectures for image recognition and language modeling [28].

The NAS domain comprises of a set of tools and methods that will test and evaluate a large number of architectures across a *search space* using a *search strategy* and select

the one that best meets the objectives of a given problem by maximizing a *performance function*. Figure 3.5 gives a graphical representation of the concept.

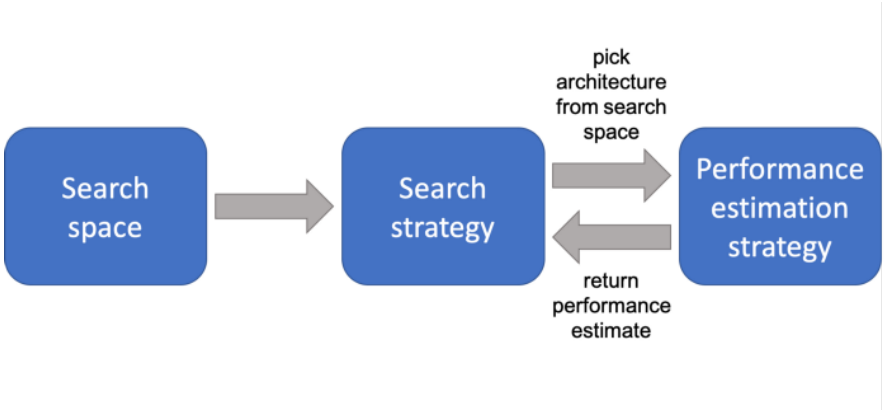


FIGURE 3.5: Graphical representation of Neural Architecture Search concept.

Search space The search space represents the set of possible architectures that the NAS can consider along its research. The search spaces is defined by the degrees of freedom of the architecture, over which the NAS can scan. The search space might involve, at higher level, the number of layers and the way they're connected (skip connections or branching architectures for example), while, at lower level, it might include also the hyperparameters of each single layer. It is strightforward that the bigger is the search space, the higher the potential performance outcome, the longer the research could become. In order to limit the dimensionality of the search space, it can be restricted to assembling pre-defined submodules (portions of DNNs) which are known to be well performing. When restricting the search space to a single architecture with different sets of hyperparameters, NAS crosses over to hyperparameter optimisation.

Search strategy The search strategy is the mechanism used to navigate the search space according to the results obtained by previous trials, with the goal of converging to a local or global optimum. Among the most common strategies we find (i) grid search, (ii) random search, (iii) evolutionary algorithms and (iv) reiforcement learning algorithms.

Grid search consists in systematically screening the search space. In random search, instead, architectures to be tested are randomly extracted from the search space. Naturally, these two search strategies can be adopted only in case of small search spaces.

Evolutionary algorithms work according to the following process: first, a pseudo-random set of architecture candidates is selected from the search spaces and evaluated; secondly, the lower performing architectures are deleted from the candidate pools and substituted

with mutations of the surviving ones; then, the new candidates are evaluated and the whole process continues in an evolutionary manner, as shown in Figure 3.6. Mutations include modifications in the number of layers, their connections or hyperparameters, such as changing the size of a kernel. Also hyperparameters of the training phase can be the subject of the search space. The evaluation of each candidate is performed via network training and test. Given the computational cost of training many networks, as explained below, early stopping criteria are used to limit the training time.

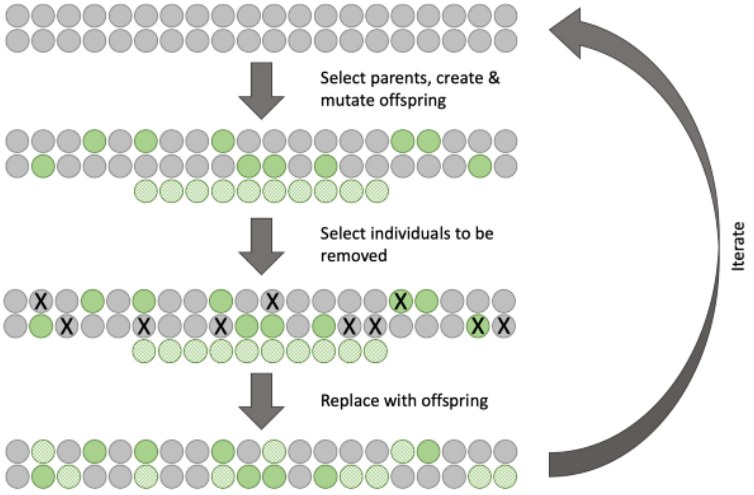


FIGURE 3.6: Neural Architecture Search performed using evolutionary algorithms.

Reinforcement learning algorithms started to be developed also for NAS only in the recent years. In this case, the candidate DNN is sampled from the search space according to a certain probability distribution given by an additional controller network, usually a recurrent neural network (RNN). The sampled architecture is trained and evaluated using the performance estimation strategy. According to the obtained performance, the controller network is updated. This process is iterated until convergence or timeout.

Performance function Every NAS approach requires a method to evaluate the goodness of a candidate DNN. A commonly used method consists simply in training the candidate network on the training set and then evaluating its accuracy on the test set. However, especially when the number of candidates increases, performing this procedure in a complete form for all the candidates could easily become too much time consuming. For this reason, methods are used to estimate, instead of fully evaluate, the performance of a candidate via approximations such as reduced training time (via early stopping), reduced training and validation data, weight initialization from previous well performing models or learning curve extrapolation.

According to the specific application, the performance function could also take into account measures such as the computational and memory footprint of a candidate. As

will be shown in the present work, this becomes particularly important when applying NAS to find embedded systems-compatible DNNs.

3.3.1.3 Feature engineering and application specific design

Before the neural network era, a common trend in the computer vision field was to try to design algorithms able to extract from the images the features considered as meaningful from experts in the field. The simplest example of this are edge detectors, since edges are considered relevant features for the understanding of an image. The advent of neural networks dramatically revolutionised traditional approaches, since neural networks are intrinsically capable of learning which are the important features of an image for a given task. The increasing ability of Depp CNNs, in particular, to learn in an end-to-end fashion (from the raw input image to the final output, such as a classification label) weakened the importance of feature engineering, that is the process of manually designing the rules to process an input.

However, as will be seen in this work, when targeting algorithms suitable for execution on resource constrained devices, combining feature engineering with the self-learning ability of Deep CNN is of fundamental importance. For example, in the task of tracking a moving object on a video recorded by a fixed camera, giving in input to a CNN the pixel-wise difference between two consecutive frames might be helpful for the network to identify the moving object.

Proving the effectiveness, in terms of optimization, of combining application-aware feature engineering with CNNs is one of the major drivers of this work. We will investigate how domain-specific knowledge can be effectively used not only to pre-process the CNN's input, like in the previous example, but also in the design of the CNN architecture itself.

3.3.2 Architecture implementation level

Given a DNN architecture, techniques exist to implement it in compressed ways to reduce its computational and memory footprint while minimizing the impact on its accuracy. In this section, we give an overview of three of the most used methods, namely Pruning, Weight Sharing and Quantization.

3.3.2.1 Pruning

Pruning weights is one of the most commonly used techniques to reduce the number of parameters in a pretrained DNN. Through pruning it is possible to dramatically reduce

the memory footprint of the network and its execution time. Network pruning consists in removing from a DNN the connections which provide minimum contribution to the NN function. This process is similar to what has been found to happen in the human brain during the first years of life: the number of neurons and connections among them at birth is at its maximum, and then decreases while the life experience determines which are the useful connections to keep and those to discard, thus shaping our brain. In the same way, when pruning a network, weights are removed and the network is retrained iteratively, looking for the desired network size and accuracy tradeoff. In fact, retraining the pruned network is a form of fine tuning which affects only the weights which have already proved to be the impacting ones. Although the most common pruning method is at weight level, pruning can be performed also at neuron level, where a complete neuron is removed including all its ingoing and outgoing connections, or at layer level, where a whole layer is removed from the network. Figure 3.7 shows an example of pruning at weight and neuron level, applied on a network of fully connected layers. However, the same concept can be extended to CNNs.

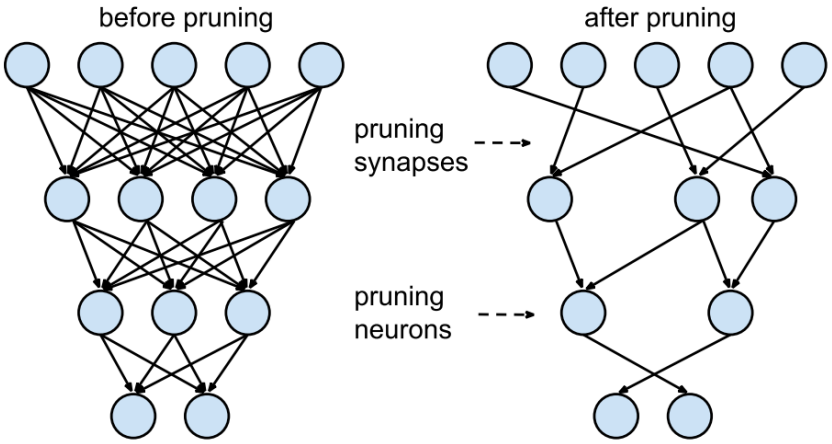


FIGURE 3.7: Example of pruning applied to fully connected layers. Both weight and neuron pruning are shown.

Pruning requires a method for the selection of the weights to prune. Although random pruning might also be applied, this technique has a higher probability of negatively impacting the network performance and requires extensive retraining after pruning [29]. Commonly used methods are in the cathegory of magnitude-based pruning (MBP), which means that the "importance" of a weight is determined according to its magnitude, as it is straightforward to infer from the basics of neural network computing. A simple yet effective pruning strategy consists in discarding weights lower than a defined threshold. When pruning is applied to neurons, the threshold can be set on the sum of the incoming weights of the neuron [30].

Thresholds can be set per layer, according to the distribution of the magnitude of the weights of that layer, or can be set once for the whole network. Other methods [31] propose instead to prune the weights with lowest absolute value of the normalized gradient multiplied by the weight magnitude, given a set of mini-batch inputs. Also in this case, the method can be applied globally or on a layer-wise manner.

Instead of setting a threshold, one can predefine a percentage of weights to be pruned based on their magnitude, layer-wise or globally.

As an example, Figure 3.8 shows the results obtained applying several pruning techniques to the ResNet-18 network on Imagenet dataset. It can be seen how the accuracy gets preserved even with a compression ratio of 4, using a pruning technique based on a global threshold.

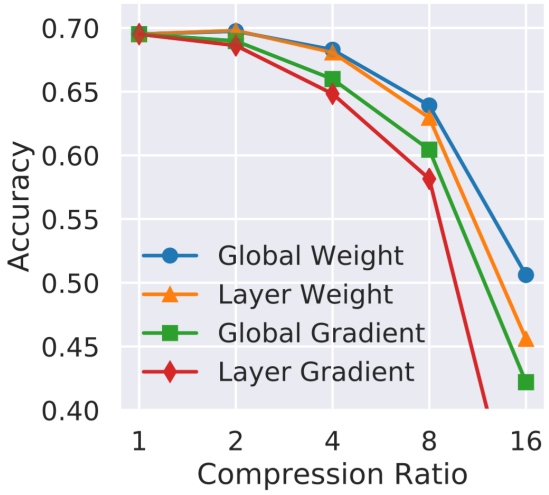


FIGURE 3.8: Results obtained applying several pruning techniques to the ResNet-18 network on Imagenet dataset [8].

3.3.2.2 Weight sharing

If the main goal of network pruning is reducing the number of parameters, the weight sharing technique aims at reducing the number of bits needed to store the weights. The idea behind weight sharing is that weights that are very close to each other (in terms of magnitude) can be grouped together in a cluster and substituted with a single weight representative of all of them, computed for example as the centroid of the weights. Then, each weight in memory can be replaced with a number indicating which cluster the original weight belongs to. A "code book" is generated, containing a map of correspondence between the cluster number and the reference weight for that cluster. The upper-left part of Figure 3.9 gives a conceptual representation of the concept: 16

weights of 32 bits each (assuming floating point representation) are grouped into 4 clusters; this allows them to be replaced by 2 bit pointers to the corresponding cluster; the amount of memory required to store the parameters decreased from $16 \times 32 = 512$ bits to $16 \times 2 + 4 \times 32 = 160$ bits.

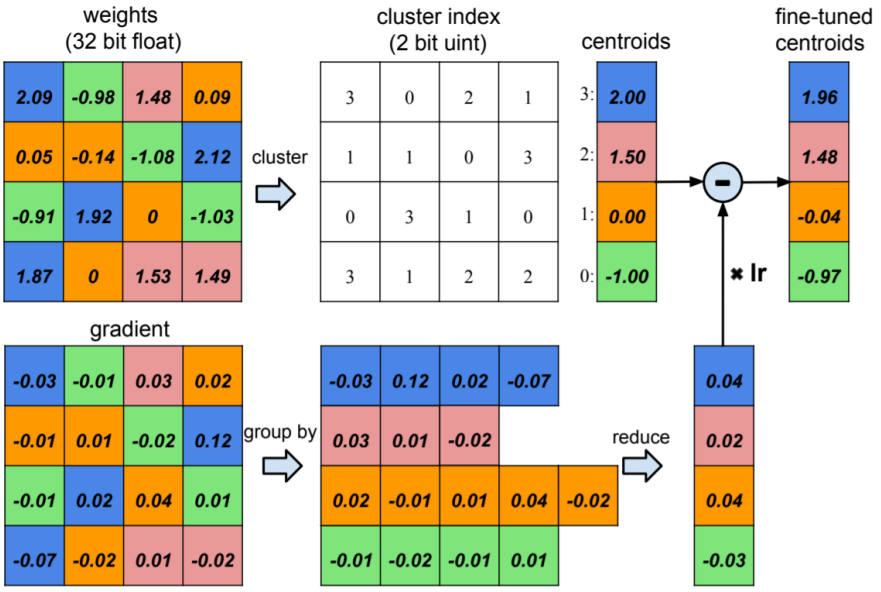


FIGURE 3.9: Conceptual representation of the weight sharing technique and of the update of cluster's centroids.

While training, centroids need to be updated. According to the Stochastic Gradient Descent (SGD) method, while training, the gradient of each weight is computed. As shown in the lower part of Figure 3.9, the gradients of the weights belonging to the same cluster are grouped together and summed up. Then, after multiplication by the learning rate, the centroids are updated. This process is repeated at every iteration of the SGD. Figure 3.10 from [32] shows the potential of combining weight sharing and pruning to achieve model compression of AlexNet on the ImageNet dataset. In the original figure the term quantization is used indicating weight sharing.

To achieve high compression rates, weight sharing can be integrated with Huffman Coding to choose the number of bits to represent each cluster: according to it, frequent clusters can be represented using a lower number of bits, while a higher number of bits are used to represent in-frequent clusters.

3.3.2.3 Quantization

As shown in Figure 3.11, the computational speed, energy demand and area cost of running neural networks strongly depends on the data types involved, even in the case of HW specifically designed for neural network acceleration. In particular, decreasing

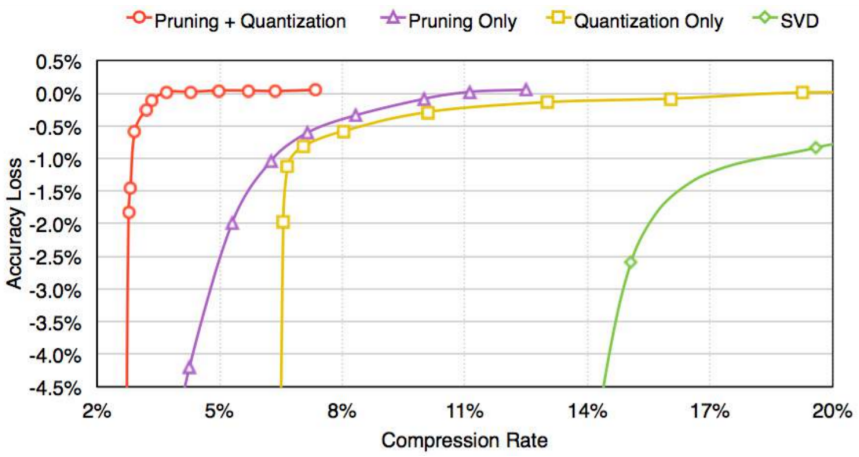


FIGURE 3.10: Accuracty loss with respect to different DNN compression methods, as a function of the achieved compression rate. The term quantization refers to the weight sharing approach explained in this section. SVD stands for singular value decomposition.

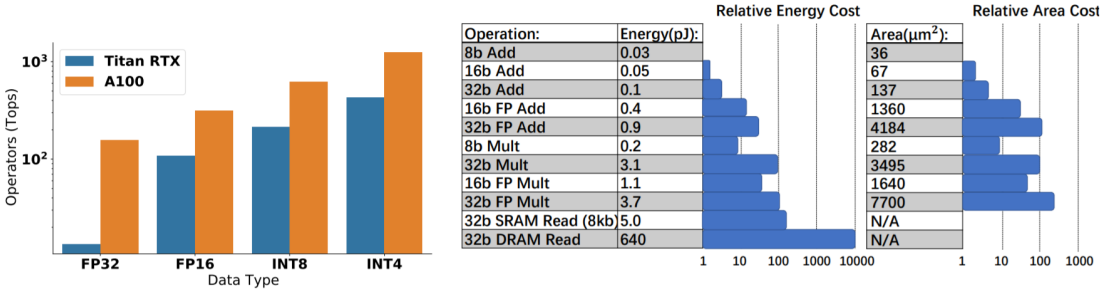


FIGURE 3.11: (Left) Comparison between peak throughput for different bit-precision logic on Titan RTX and A100 GPU. (Right) Comparison of the corresponding energy cost and relative area cost for different precision for 45nm technology. As one can see, lower precision provides exponentially better energy efficiency and higher throughput. [9]

the number of bits of datatypes leads to exponentially better energy efficiency and higher throughput.

For this reason, quantization is a widely adopted method in compressing neural networks, especially when targeting embedded devices. The goal of quantization is to substitute 32-bit floating point operations with lower-bits operations, such as using 8-bit fixed point representations.

Although several variants of the technique exist, a common workflow consists in the following operations. First, the NN is trained using floating point representations. After training, according to the statistical distribution of weights and activations (the latter computer on a batch of input data), an appropriate scaling factor S is chosen to represent the data using a lower number of bits. As an example, given r the real value which a set of activations or weights might have, we can choose S as

$$S = \frac{max(|r|)}{2^n - 1} \tag{3.5}$$

where n is the number of bits of the target representation (8 in INT8 for example). The quantization factor can be chosen with different levels of granularity, such as per network, per layer or per filter in CNNs. Moreover, several methods exist for defining the quantization factor, such as uniform and non-uniform quantization. The above formula refers to the widely adopted *uniform* quantization scheme.

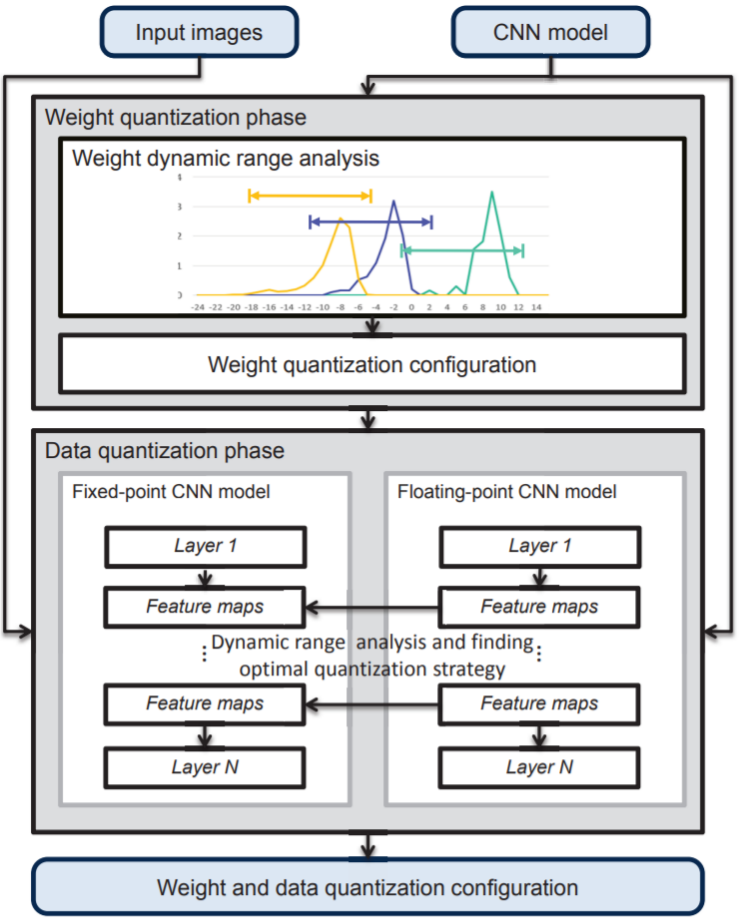


FIGURE 3.12: Neural network quantization workflow. [10]

A variant of the quantization approach just described is called "quantization aware training" and consists in performing, while training, the forward propagation step using quantized data and updating the quantization factor, while the backward propagation step is performed in floating point.

3.4 Embedding DNNs: Hardware level

Despite their high potential DNNs are a composition of basic computational operations such as multiplications, sums and data accesses, which make them executable, in principle, on any type of computing platform. However, if on one side the limited computing power of a hardware device may dramatically increase the processing time, in case of insufficient memory the DNN could not even be executable. For this reason, HW platforms play an important role in the challenge of embedding DNNs.

In the following section, we give an overview of the HW families that can be used for execution of DNNs in embedded environments. Then, in Section 3.4.2 we highlight the most important factors involved in the design of hardware accelerators. Finally, in Section 3.4.3 we provide some commercial examples of HW accelerators for DNNs, frequently used in the research community.

3.4.1 HW families

When deploying DNNs to embedded HW, we can consider 4 main types of HW families, grouped as follows:

- **General purpose**: Comprises the HW platforms designed in an application independent manner. They're normally less optimized but more flexible. Are useful to minimize time to market and for fast prototyping.
  - **CPU**: Central Processing Unit. In this cathegory we include common microprocessors and microcontrollers. Although they can be used for the scope, these devices are normally not specifically designed for running neural networks. Microprocessors are used in computing architectures where they can benefit from an external, potentially large, memory, and therefore the main difficulties in using them for running DNN relies in their slow computation and memory access speed. On the other sude, microcontrollers are more suitable for resource constrained embedded systems and are optimized in space and power consumption. They feature a very small (roughly up to 1 MB) internal memory, which requires using extensive SW Techniques (like those presented in Section 3.3) to compress a DNN and make it runnable on these devices.
  - **GPU**: Graphic processing units. As the name suggests, GPUs where initially designed to accelerate graphic computations, with particular attention to the very demanding field of gaming and animation. However, they soon became

of interest for DNN applications, and CNN in particular, given the similarities between the two tasks, which are both strongly based on operations between matrices. With respect to CPUs, which have a few cores (typically 4 or 8) each of which can perform a variety of complex tasks, GPUs are characterized by thousands of cores, optimized for the execution of relatively simple operations in parallel. To feed all these cores with the needed data, GPUs feature a high amount (several GB) of high speed local memory, and several caching levels. As presented in Section 3.4.3, GPUs are being readapted also for embedded applications, with designs that focus more on reducing their size and power consumption. Despite this, their power consumption still remains much higher than that of a microcontroller.

- **Specialized**: Comprises the HW platform designed and optimized for a specific task or set of tasks. Require more design effort, but can provide higher energy and space efficiency.
  - **FPGA**: Field Programmable Gate Array. FPGAs are HW platforms which can be configured to implement a specific set of operations on the input data, with focus on latency and power consumption minimization. FPGAs consist of an array of programmable circuits that can each individually do a small amount of computation, as well as a programmable interconnect that connects these circuits together. The large number of programmable gates in the FPGA makes it a naturally highly parallel device. FPGAs can therefore be used to implement DNNs, as an intermediate solution between the power demanding GPUs and the ASICs, which have the characteristics explained here below.
  - **ASIC**: Application specific integrated circuit. As the name suggests, ASICs are HW boards designed for a specific task. The reduced need for generalization, enables a higher optimization of the design of the device, given the target application and performance. However, these gains comes at a high cost: the time to market of a ASICs is much longer that that of the other solutions already presented; moreover, the design of an ASIC is a very costly complex task. The optimum design of ASICs allows to reduce piece price cost in series production, therefore leading to an economic gain in case of high production quantities. However, as per the authors knowledge, the fast evolving sector of artificial intelligence tends to discourage HW makers from committing on long-term ASICs projects for very specific applications, while preferring more flexible GPUs and FPGA based platforms.

3.4.2 HW Acceleration

In the context of DNNs, HW accelerators are devices which implement techniques aimed at improving the performance, typically in terms of latency and throughput, in the execution of DNN. Although GPUs, FPGAs and ASICs can all be used as HW accelerators both for DNN inferencing and training, GPUs are nowadays the most common form of hardware accelerator. In the last years, Google proposed the Tensor Processing Unit (TPU) as a new type of high performing hardware accelerator.

Hardware accelerators have to face two challenges: optimizing computations and memory access.

From the computational point of view, most of the operations of a DNN are Multiply and Accumulate (MAC), which are at the basis of matrix multiplication. Computations are accelerated by mean of two main approaches: (i) making use of several parallel Arithmetic Logic Units (ALUs); (ii) speeding up each ALU with hardware optimized for the MAC operation. In devices such as the Google's TPU, the second point gets also advantages from using an 8 bit quantization, as shown in Figure 3.11.

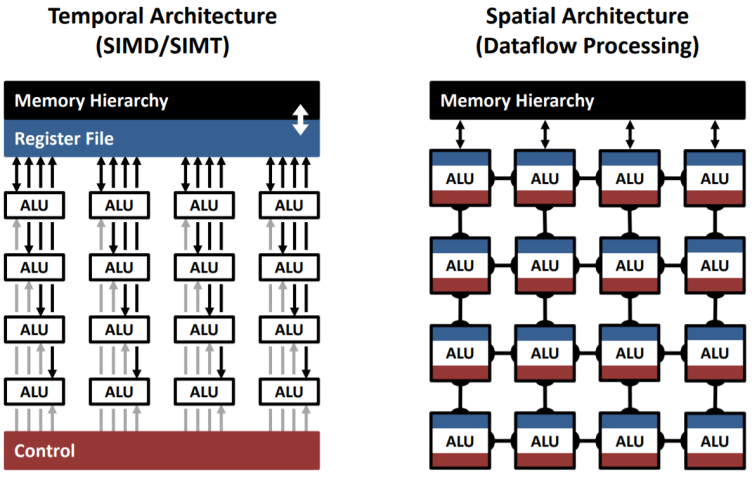


FIGURE 3.13: Highly-parallel compute paradigms. [11]

From the memory access point of view, two main parallel compute paradigms exist, depicted in Figure 3.13:

- **Temporal architecture**: in this paradigm all the ALUs are controlled by a central controller and exchange data with a common register file, which then communicates with the DRAM. Therefore, data is exchanged between the ALUs always passing through the register file. This paradigm is typical of Single Instruction Multiple Data (SIMD) or Single Instruction Multiple Thread (SIMT) devices, such as CPUs and GPUs respectively.

- **Spatial architecture**: in this paradigm each ALU has its own register file and controller, forming the so called Processing Element (PE). This architecture allows implementing a dataflow among the several PEs and enables the usage of a more sophisticated and optimized memory structure which guarantees much higher speedup with respect to the temporal architecture case. In fact, as shown in Figure 3.14, fetching the data from the local register file (for example to reuse the same weights for multiple convolutions) or an adjacent PE (for example to read the activations it computed) is much faster than accessing the DRAM or the global buffer. Of course, in case the spatial architecture is used, a proper design of the dataflow needs to be carried out to maximize "close data" reuse.

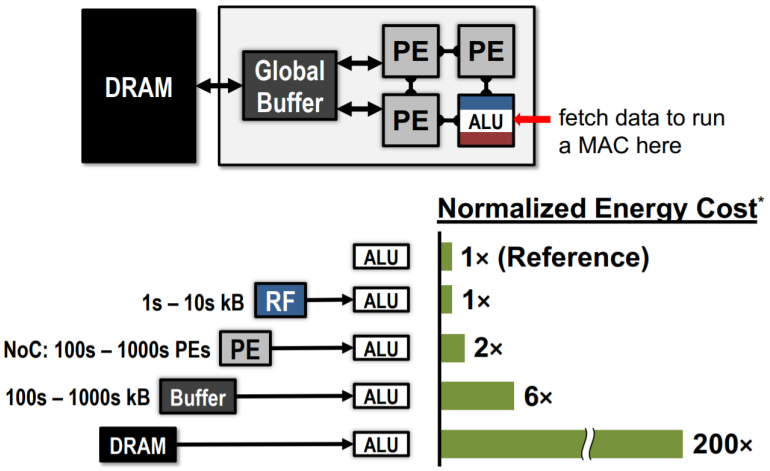


FIGURE 3.14: Memory access cost as a function of the memory storage type. [11]

3.4.3 Commercial solutions

In this section we give a brief overview of three popular general purpose commercial solutions for running DNN on Embedded systems. They're widely used both in the industry as well as in academics. Each of them represents one of the three main cathegories that we have identified, namely: System on Chip, Stick, CPU Based.

3.4.3.1 System on Chip: NVIDIA Jetson Series

A system-on-chip (SoC) is an integrated circuit which integrates most of the electronics of a complete computer. Alongside the CPU, a SoC includes input/output interfaces, volatile as well as non-volatile memories, power management units as well as other custom units for example for digital signal processing, wireless communications and more. SoC have the advantage ob being extremely compact and of dramatically reducing the development costs and time to market of the final application, since they're normally

designed by silicon manufacturers in high quantities and adaptable to many different applications.

The NVIDIA Jetson is a series of SoC HW platforms including an operative system and a complete SW framework to accelerate the development and execution of AI solutions on embedded systems. The HW platform range from more compact devices such as the Jetson Nano, suitable for example for drone applications, to high performance units targeting autonomous machines such as the Jetson AGX Orin.

Just as an example, the Jetson Nano is a compact device which consumes 5 to 10W, able to compute up to 472 GFLOPS, equipped with a Quad-core ARM Cortex-A57 MPCore with a clock frequency up to 1.43 GHz.

The AGX Orin, on the other side, is capable of performing up to 275 TOPS by exploiting an integrated 2048 core GPU, also featuring 64 tensor cores, specifically designed to accelerate DNN. It has a 64-bit CPU ARM Cortex-A78AE with a clock frequency up to 2.2 GHz.

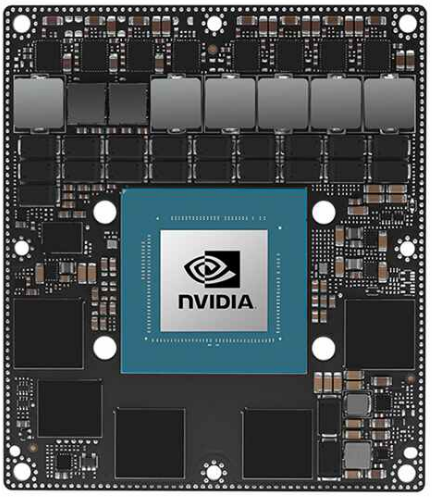


FIGURE 3.15: NVIDIA Agx Orin.

3.4.3.2 Stick: Intel Neural Compute Stick

In some cases, developers might need to update an already existing embedded solution to allow execution of neural networks. For these cases, AI HW accelerators in the form of USB sticks exists. These solutions minimize development cost and time-to-market, in particular when upgrading already existing systems. The USB stick can be connected to an existing host computer (also embeeded ones, like a Raspberry Pi) and receive

from a running application computation-intensive tasks to perform. Once the stick has computed the output, it is sent back to the application.

Intel's Neural Compute Stick 2 (NCS2), shown in Figure 3.16, is an example of these kind of accelerators.

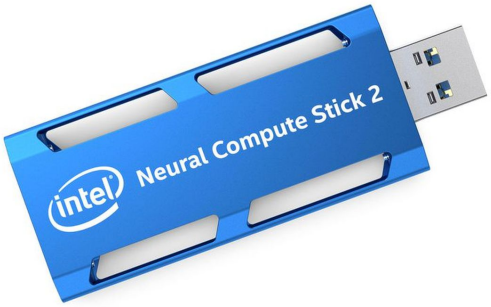


FIGURE 3.16: Intel Neural Compute Stick 2.

Intel offers the NCS2 coupled with the Open VINO SW framework, which simplifies the development of applications exploiting its potential. The NCS2 is equipped with the Intel's Movidius Myriad X Vision Processing Unit (VPU), featuring 700 MHz of base frequency and capable of reaching 4 TOPS of processing power, while keeping power consumption as low as 2 Watts. Open VINO is also compatible with common frameworks such as Caffè and TensorFlow for DNN design.

3.4.3.3 CPU Based: STM X-CUBE-AI

As previously stated, also standalone CPUs, equipped with internal or external memory, can be used to execute DNNs. In these cases, the device might also not be provided with a dedicated HW acceleration. Of course, the limited computing power significantly increases execution time with respect to other solutions, and the low amount of memory can also undermine the possibility of executing the network. On the other hand, these solution can fit very small devices and require less then 1 Watt of energy.

ST Microelectronics provides to its users a software framework named X-CUBE-AI that, working as an add-on of the STM32CUBE toolchain, simplifies the deploy of neural networks on the STM32 family of microcontrollers. X-CUBE-AI offers an engine for the automatic generation of optimized C code for the execution of neural networks, as well as a set of tools to analize and configure the memory usage of the implementation. Since the tool works independently from the specific microcontroller choice, the designer can select the HW device most suitable for the desired application. Figure 3.17 depicts the reference workflow for developers given by ST. As will be seen in the nect chapters, this

workflow has also been used in the context of this work to deploy a NN to an STM32 microcontroller.

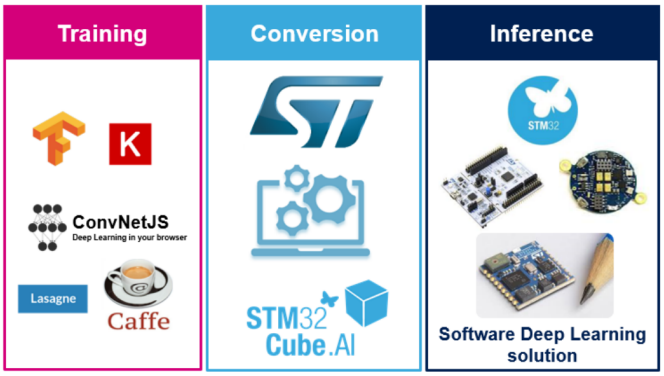


FIGURE 3.17: Reference workflow for the deploy of NN on STM32 devices using X-CUBE-AI.

Applications

In the next chapters we present some reference applications we considered in order to investigate the potential of application-aware neural network design and optimization, in particular when targeting resource constrained embedded devices.

Most of the research on the deplyment of AI solutions to embedded systems targets very general purpose methods, like those presented in Chapter 3, which can often be applied in an application independent manner. Our intent is to contribute in a complementary way: we want to underline that, although the indoubtedly powerful self-learning capability of many AI algorithms, the contribution of the so called "feature engineering" and application-specific design is still crucial when it is needed to push the peformance of the system (also in terms of reduced memory and computational burden) to the limit.

Each of the works presented in the next chapters focuses on a different optimization target. In Chapter 4 we focus on fast and accurate DNN inference; in Chapter 5 we focus on making an image classifier retrainable directly on an embedded target device; in Chapter 6 we focus on making an object detector implementable on a microcontroller without HW acceleration; in Chapter 7 we present an alternative method of the previous chapter, using Neural Architecture Search; finally, Chapter 8 presents a lightweight optimized AI algorithm that we patented to let electric vehicles learn habitual routes, with the goal of predicting and optimizing the energy consumption.

Chapter 4

Optimizing speed and accuracy: Fast small object tracking

4.1 Introduction

The recent growth of industrial applications for object detection stimulates the research community toward novel solutions. Intelligent video analysis is the core of several industry applications such as transportation [33], sentiment analysis [34], and sport [35, 36].

As shown in Chapter 2, Deep Learning lies today at the core of state-of-the-art techniques for object detection, such as Faster RCNN[21], YOLO[25] and SSD[6]. Thanks to GPUs, object detection solutions based on deep learning can support real time applications; the edge-computing market now offers a variety of relatively inexpensive devices for Artificial-Intelligence (AI): microprocessors [37], hardware accelerators [38], up to complete Systems on Module (SoM), such as the Jetson series by NVIDIA [39], and machine vision cameras such as the JeVois A33 and Sipeed Maix Bit, used in [40]. These tools rely on GPUs and a collection of software optimisations to deploy computationally intensive tasks, such as AI inference, on resource-constrained hardware. Real-time object detection on embedded devices still represents a major issue, as that goal involves quite complex architectures for deep learning. In practice, one needs a trade-off between accuracy and latency to tune each method to the target scenario.

In this work we search for a method to perform fast and accurate detection of small moving objects, which typically take up a few tens of pixels, by using fixed cameras. State-of-the-art approaches often exhibit poor performances when dealing with very small objects, due to the apparent difficulty in discriminating these features from one another and from the background [41]. Figure 4.1 presents an example, including three

candidate sub-regions extracted from as many frames in a tennis-match video. While the rightmost frame actually includes the ball, the other patches do resemble a tennis ball but represent misclassification errors.

Human observers face a similar challenge when looking for tiny objects in a wide scene. The detection task, in fact, gets simpler if the target moves with respect to a still background, since the human vision system can combine motion information with the visual aspect of the object. Figure 4.2 clarifies this concept: the image on the left is the frame (at time t_n) drawn from the tennis video. The image on the right merges the frames from time t_{n-5} up to t_{n+3}. In the former case, the ball is hardly distinguishable even by a human viewer, not just for its small size, but also because motion blur hinders the detection of fast-moving objects. In the rightmost image, instead, motion information makes the tennis ball clearly detectable.

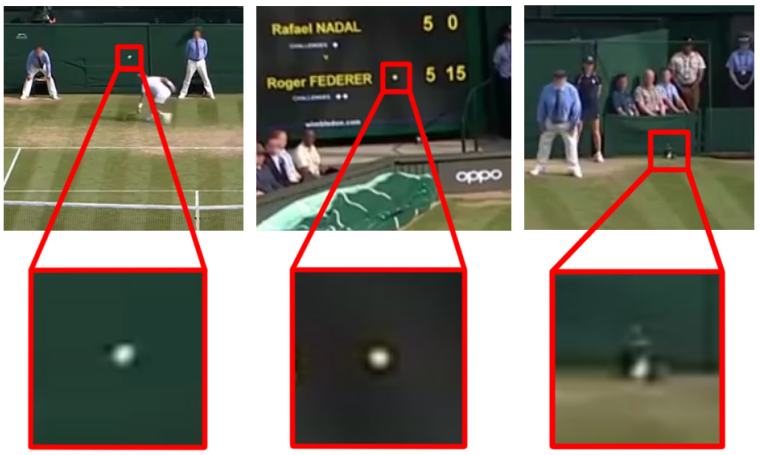


FIGURE 4.1: Three examples of patches which show how easily a small object might appear similar to other objects. Only the rightmost patch is a tennis ball, while the other two objects appear similar to it without actually being a tennis ball. Without a mean to discriminate the real object from potential false positives, a neural network might fail to learn how to recognise the sought object.

The approach presented here deploys the detection of tiny moving objects in wide scenes on limited hardware resources and using fixed cameras. The method adjusts the basic building blocks of resource-constrained computer vision, and proposes a custom deep neural network for the recognition task called T-RexNet. The T-RexNet framework improves over generic hardware-aware detectors, which only rely on visual features, and combines those features with motion information. The framework processes three consecutive frames from the video source, and prompts a set of bounding boxes around the detected objects. The overall architecture includes two stacked blocks, for feature extraction and subsequent object detection.

The dedicated pair of parallel convolutional paths in the network support that image/-motion fusion process. As compared to generic object detectors, the computational

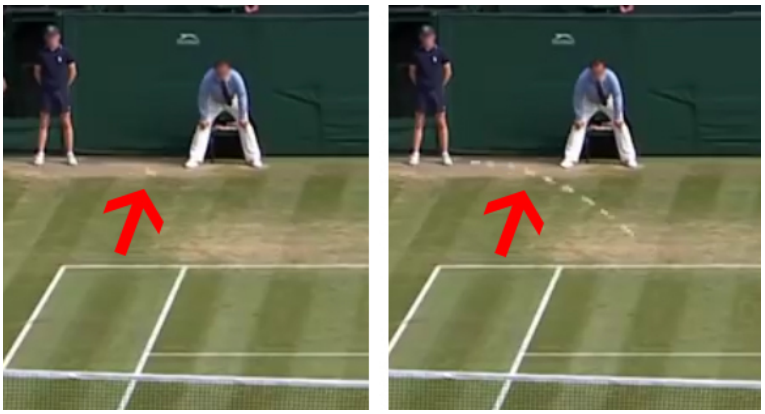


FIGURE 4.2: The left image shows a single frame as it is extracted from the video. The tennis ball is indicated by the red arrow and is almost undistinguishable. The right image overlays the position of the tennis ball in the previous and following frames and shows how the motion information is foundamental for its detection.

overhead brought about by the two-tiered feature-extraction network is mitigated by reducing the network depth. As a matter of fact, focusing on tiny objects allows to leave out the deep layers operating at low resolution.

Single-Shot-Detector (SSD) architectures are quite popular for resource-constrained object detection. The custom feature-extraction module overcomes the well-known limitations of SSD in detecting tiny objects. The resulting feature-extraction architecture is quite shallow, and the object detection block relies on one of the least demanding available State-of-Art (SoA) solutions. In summary, the integration of these two features yields a viable solution for the real-time detection of small objects by constrained devices.

Experimental results prove that, in that context, T-RexNet improves significantly over state-of-the-art methods for generic object detection. As compared to application-specific solutions, T-RexNet exhibits a satisfactory accuracy vs/speed balance in several complex scenarios such as aerial and/or civilian surveillance and high-speed detection, tackling medium-sized to tiny objects, and varying target densities. In other words, it manages to achieve high detection rates without sacrificing accuracy too much.

The following is organised as follows. Section 4.2 overviews the state-of-the-art in object detection, moving-object detection, and in the specific domains used for testing. Section 4.3 presents the T-RexNet approach in detail. Section 4.4 discusses the test scenarios considered, whereas Section 4.5 makes some concluding remarks. .

4.2 Tiny Moving Object Detection: State of the Art

The identification of small moving objects is a subset of a wider research field in object detection. Existing solutions and techniques can be arranged into three main groups, namely, Single-image solutions, Background-subtraction solutions, and Spatio-temporal CNNs .

4.2.1 Single-Image General-Purpose Solutions

Typical object-detection models handle one image at a time, even when spatio-temporal information might be available. FOllowing the cathegorization done in 2 State-of-the art approaches, relying on deep learning, can be divided into region-based and single-shot detectors.

In the former models, such as R-FCN [42] and Faster R-CNN [21], a dedicated algorithm first extracts a set of Regions-of-Interest (ROIs), that is, sub-portions of the image that are likely to contain an object; then fine-detection and classification modules analyze each ROI. Single-shot detectors such as YOLOv3 [25], SSD [6] and DSSD [43], instead, avoid looping over several ROIs, and tackle the input image in a single shot. These methods apply a library of predefined bounding boxes (anchor boxes), which have various shapes and sizes and cover the likely locations of objects in the image. The inference phase takes care of fine tuning each anchor box in terms of size and position.

Region-based detectors usually prove more accurate that single-shot detectors, but are computationally demanding, as they require a loop for each single ROI [44]. In the case of small objects at low resolutions, both region-based detectors and single shot detectors tend to exhibit poor performances. Several techniques have been proposed recently to overcome that issue [45]:

- **Multi-scale representation**: high- and low- resolution feature maps stem from different levels of a feature-extraction network; after super-sampling low-resolution maps, features fuse together by applying either element-wise sum (Multi-scale deconvolutional single shot detector (MDSSD) [46]) or concatenation (Diverse region-based CNN (DR-CNN), [47]).
- **Contextual information**: the network takes into account explicitly the contextual information around a candidate object. For example, ContextNet [48] applies a custom region-proposal network specifically aimed to small objects, and for each candidate region an enlarged region is used to process contextual information.

- **Super resolution**: generative adversarial networks generate a higher-resolution version of the candidate object, thus improving accuracy in the detection of small objects (Perceptual generative adversarial networks (PGAN)) [49]).
- **Mixed methods**: features with distinct scales are extracted from different layers of a convolutional neural network; they are concatenated together, and then used to generate a series of pyramid features [50].

These methods all exhibit an increase in both computational and memory load. This brings about lower update frequency, higher latency, and ultimately might compromise implementations on resource-constrained devices for embedded applications.

4.2.2 Background Subtraction and Frame-Difference Solutions

In complex applications such as aerial surveillance, camera views can cover wide areas. Target objects (e.g., pedestrians and cars) usually span just a few tens of pixels, and the detection techniques discussed above [51] are ineffective. At the same time, in those applications the majority of input images are quasi-static and only target objects move in the scene, hence conventional background-subtraction approaches are widely adopted, even in the era of deep learning. The basic idea consists in working out the difference between a frame and the background model of the scene acquired by the same camera; the time-difference information highlights the changes caused by moving objects.

Methods differ in terms of computational cost, robustness and accuracy—Mixture of Gaussians (MOG) [52] approaches model each pixel as a random variable with a gaussian mixture model; mean-filtering [53] techniques extract the background by averaging the values of each pixel over the last N frames, whereas methods for frame-difference background subtraction [53] only consider the pixel differences between the current frame and the previous one. The latter approach is very fast but possibly less robust to noise; moreover, by disregarding any sequence of past frames, frame differences only apply when the camera is slowly moving.

Since these methods typically process gray-scale (or even B/W after threshold) images that highlight changes at a given time, the actual detection of moving objects requires some post-processing. This might possibly include morphological transformations, blob detection [54], or more complex computations [55–57], to the detriment of detection speed.

4.2.3 Spatio-Temporal Convolutional Neural Networks (CNNs)

The literature witnesses the growth of spatio-temporal CNNs, which take into account both visual and motion data. In MODNet [58], the authors proposed a two-stream neural network that processed input RGB images and optical flows, thus learning object detection and motion segmentation at the same time. The research presented in [59] adopted an end-to-end approach for video classification. A pseudo-3D neural network learned spatio-temporal information by considering multiple consecutive frames, which were processed by a series of convolutional filters in both the spatial ($1\times\ 3\ \times\ 3$) and the temporal ($3\ \times\ 1\ \times\ 1$) domains. The 3D neural networks virtually replaced explicit image pre-processing steps such as background subtraction or optical-flow computation.

A spatio-temporal CNN supported the detection of vehicles in Wide Area Motion Imagery (WAMI) [60]. In the 2-stage approach, a CNN first handled 5 consecutive images (taken by an aerial surveillance system) and highlighted promising regions. The second stage completed fine detection within each region. The TrackNet approach [35] applied spatio-temporal CNNs to track small fast-moving objects in sport applications; a fully convolutional neural network could accurately track a tennis ball by processing 3 consecutive video frames (taken by a steady camera). The CNN prompted a heatmap of the possible positions of the ball, subsequent blob detection eventually yielded the predicted location.

Spatio-temporal CNNs for object detection can prove effective, but also exhibit some drawbacks: they are often computationally heavy; the various approaches are normally tailored to specific applications, and application-independent detection of small objects has not been proved yet.

4.2.4 Summary of Contribution

The methods discussed above all exhibit some features that make them unsuitable to support the Real-Time detection of small moving objects on resource-constrained devices; specific shortcomings possibly include the inability to recognize tiny objects, impractical computational loads, or lack of general applicability. The approach described in this work can perform detection of small moving objects by maintaining some crucial features: it is lightweight and suitable for embedded devices, accuracy keeps comparable to SoA approaches and improves over them in particularly challenging conditions, the system is end-to-end trainable, and finally the method is application independent, as it performs satisfactorily in different scenarios.

4.3 Methodology

T-RexNet combines several of the techniques mentioned above to detect small moving objects in a fast, lightweight manner, by using a fixed camera. The system benefits from the versatility of an end-to-end fully convolutional neural network, it processes differences between frames to involve motion information, and relies on the efficiency of MobileNet-based convolutions to integrate visual and motion data. Single-shot detectors attain real-time performances. Thus T-RexNet can be regarded as a spatio-temporal, single-shot, fully convolutional deep neural network, as per Section 4.2. With only 2.38 M parameters, T-RexNet turns out to be one of the most lightweight networks in the object detection field (the SoA lightweight MobileNetV2 has 3.4 M parameters).

Figure 4.3 outlines the three-step structure of T-RexNet. Three time-consecutive gray-scale images I_{t-1}, I_t, I_{t+1} make up the system input, where $I_{\{\cdot\}}$ denotes the 2D matrixes of pixel intensities at different time steps. The algorithm first works out a pair of motion-augmented pictures, M and K, which undergo a feature-extraction process based on two separate parallel convolutional paths. The actual object-detection results stem from the third SSD-based step.

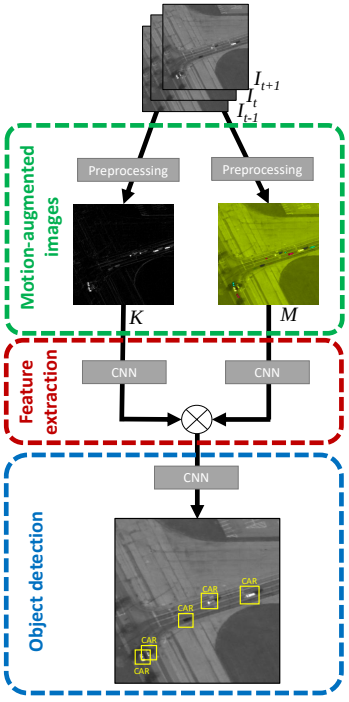


FIGURE 4.3: T-RexNet macro architecture, showing the two parallel "Motion-only" and "Mixed Visual-Motion" MobileNetv2-Based feature extractors. Their output is concatenated (circled X symbol) and then processed by an SSD network.

4.3.1 Step 1: Extracting Motion-Augmented Images

This module received in input three gray-scale input frames, I_{t-1}, I_t, I_{t+1}. Since gray-scale images are represented as matrices of size $[height \times width \times 1]$, stacking three of them we obtain a $[height \times width \times 3]$ matrix, which is equivalent to the size of a single colored image. In other words, compared to traditional object detection methods, we substituted color with temporal data. The input of the network is processed in order to generate the pair $\{M, K\}$ of motion-augmented images, as explained in the following.

The image M includes three channels that are worked out as:

$$M_t^1 = |I_{t+1} - I_t|, \quad M_t^2 = I_t, \quad M_t^3 = |I_t - I_{t-1}|,$$

where the superscripts $(1, 2, 3)$ refer to the channel number and the $|\cdot|$ is the absolute-value operator. Figure 4.4 illustrates the overall process in a graphic form. Channel M^2 preserves visual features, while channels M^1 and M^3 bring in motion information via frame differencing, which proves much faster than conventional background-subtraction techniques. It must be noted that preserving single-frame visual features in one of the three channels of the image makes the network able to detect, in principle, also non-moving object.

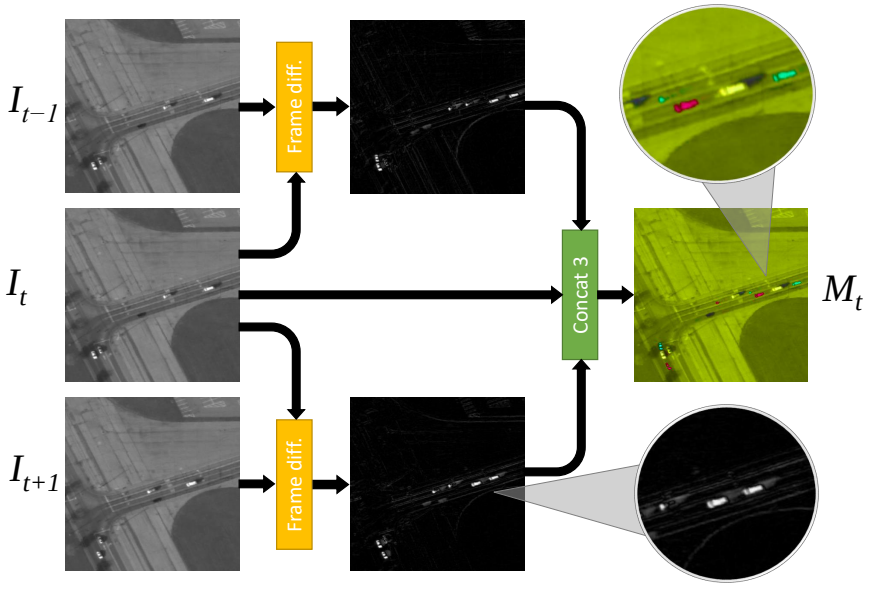


FIGURE 4.4: Computation of motion-augmented image M. For visualization purposes, after the concatenation, each of the three channels is displayed as a single color channel like in RGB images. Here, with respect to RGB, for visualization purposes the hue of the whole image has been modified. In the zoomed area of the final M image we can see that a moving car appears as 3 cars, corresponding to time instants $t-1$, t and $t+1$.

The image K is the concatenation of the first and the last channels of M, hence it only holds motion data without any visual feature.

4.3.2 Step 2: Feature Extraction

Feature-extraction networks typically include stacks of convolutional layers and pooling layers, in which lower layers involve the details of the image, whereas the topmost layers extract object-related information [61]. From a spatial point of view, the deeper is the feature map in the network, the larger is the receptive field of each of its "pixels".

T-RexNet aims to detect small objects, hence high level information can be disregarded, and the number of stacked layers in the feature extraction network reduces accordingly. This feature also entails a beneficial effect on latency. In principle, high-level features might provide context information and therefore help localise small objects; at the same time, reducing contextual information makes the feature extractor more independent of any specific scenario and therefore maximally flexible. Feature extraction in T-RexNet involves two convolutional paths that process visual-motion mixed data (image M), and only motion-related data (image K), respectively.

The rightmost path in Figure 4.3 processes image M and relies on a custom network drawn from the MobileNet [62] model. This is a family of Neural-Networks (NN) architectures specifically designed for low-latency execution on mobile devices, and yields a promising balance between computational cost and accuracy. T-RexNet inherits from MobileNet the use of **bottleneck residual block** as a main building block, as shown in Figure 4.5, to limit the sensitivity to high-level, context-dependent information.

The leftmost path in Figure 4.3 takes into account the motion-related data held in image K. The architecture features a stack of several 2D convolutions, as per Figure 4.5. The stride is set to 2, hence the input image is downsampled to match the output resolution of the parallel convolutional path.

Finally, the outputs of the two paths are concatenated channel-wise.

4.3.3 Step 3: Object Detection

The object detection block relies on SSD [6], which mitigates computational costs as compared with region-based approaches and better fits real-time applications. Since, in the inference phase, the method prompts predictions for the whole list of predefined anchors, execution time turns out to be image independent.

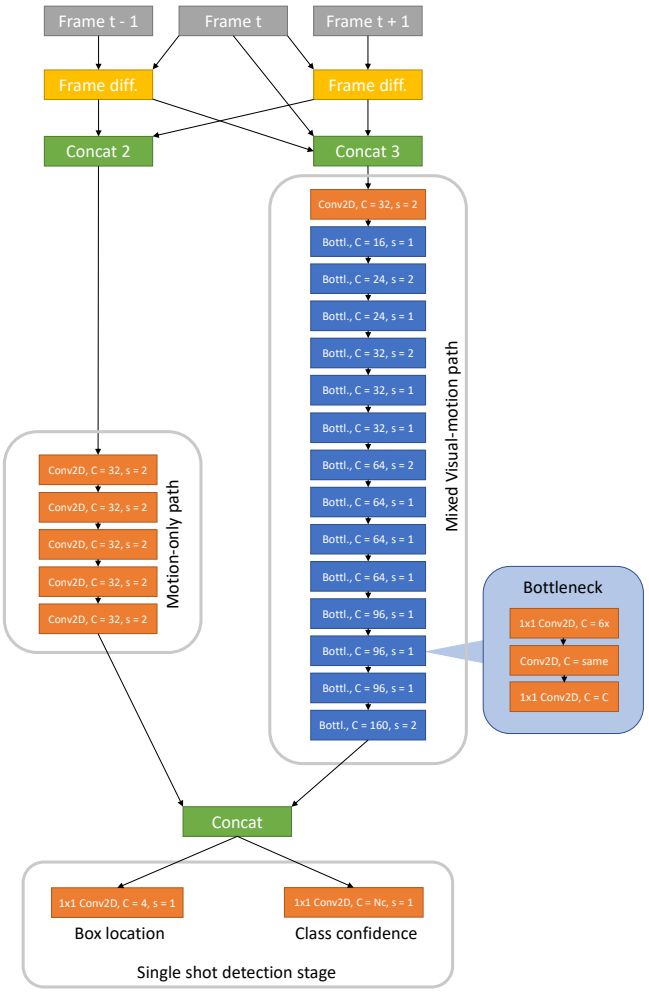


FIGURE 4.5: T-RexNet full architecture and image processing high level view. All the Conv2D blocks in the motion-only path use a 3×3 kernel. C is the number of output channels, s is the stride. Box locations are encoded with 4 numbers according to [6]. Bottleneck block [6] is highlighted: $C = 6x$ means that the first block of the bottleneck is an expansion block which increments by a factor of 6 the number of channels; $C = same$ means that the number of output channels is equal to the input ones; Nc is the number of classes.

Detection in the basic SSD involves several feature maps that are extracted at different levels of the feature-extraction network (the *base network* in [6]). This technique improves the robustness to different object scales. Since T-RexNet is targeted at detecting small objects, the output of the first stage just involves one feature map to contain computational costs.

T-RexNet associates each element of the feature map (i.e., each position in the map grid) with the dimension/position information and the classification (car/pedestrian/background etc.) of the corresponding anchors. The anchor size is set to $0.2 \cdot \text{size}(I)$, where I is a squared input image and $\text{size}(\cdot)$ is a function which returns the height and width of the image. The anchor's aspect ratios depend on the shapes of the target objects.

Scenario	**# of obj.**	**Obj. size**	**Obj. Speed**	**Im. size**
Aerial surv.	High	Small	Mid	2000
Civilian surv.	Medium	Med. & small	Low	512
Fast obj. track.	Single	Small	High	300

TABLE 4.1: Overview of the test scenarios considered in this work. Object size and speed are relative to the image frame. Image size is in pixels and measures the side of a squared image.

The standard values are $\{0.5, 1, 2\}$, which correspond, respectively, to horizontal shape, squared shape and vertical shape.

4.4 Experimental Setup

4.4.1 Scenarios

Three heterogeneous scenarios formed the test-bench for assessing the performances of T-RexNet, namely, aerial surveillance, civilian surveillance, and fast object tracking. Table 4.1 summarises the characteristics of the three scenarios and gives four quantities: the number of objects to be detected, the target object size, the speed of objects, and the overall image size in pixels.

To ensure fair tests, comparisons included methods with the following features:

1. the research community proved the comparison's effectiveness in object detection and its implementation on embedded devices; the experiments focused on each method's ability to detect small moving objects;
2. the various methods had been targeted to their specific test scenario, hence comparisons with T-RexNet could highlight the latter's balance between accuracy and speed.

The Appendix 4.7 gives details about the training procedures adopted for T-RexNet, whereas the actual experimental outcomes are discussed in Section 4.5.

4.4.1.1 Aerial Surveillance

Aerial-surveillance tests addressed the Wright-Patterson Air Force Base (WPAFB) 2009 dataset, which is a well-established benchmark in Wide-Area-Motion-Imagery (WAMI). Surveillance relies on powerful camera set-ups (and software) to detect and track hundreds of targets, usually people and vehicles, possibly over areas of several squared kilometres. This typically calls for airborne systems. Targets can be so small that motion information is required to distinguish them from the background or noise. Background-subtraction techniques are therefore popular for object detection in this field [63].

The WPAFB dataset holds images taken by an airborne system and focuses on moving vehicles. Each frame roughly includes 315 million pixels and merges the shots by six, partially overlapping, gray-scale camera sensors [60]. The total area covered by each frame is around 19 squared km and the frame rate is about 1.25 Hz. On average, each target vehicle covers a region of about 100 pixels. Due to the considerable size of each raw image, state-of-the-art methods address a set of Areas of Interest (AOI); this allows fair comparisons between the various approaches [63].

The tests presented in this work involved AOI 1, 2 and 3, as they covered a variety of layouts with different intensities of traffic. The size of each AOI was 2000 $\times$ 2000 pixels. The WPAFB dataset gave the position of a vehicle within an AOI by means of the target coordinates, (x, y); the position was mapped into a squared bounding box of 31 $\times$ 31 pixels. Stationary vehicles were not taken into account to focus on moving targets; thus cars whose positions changed less than 15 pixels between two consequent frames were removed.

T-RexNet was trained on AOIs 1 and 3, which covered high-intensity and low-intensity traffic situations, respectively. The images were taken from a moving camera, and the test phase involved the remaining set, AOI 2. To ensure fair comparisons with other approaches, time-consecutive images were recorded to support frame differencing. Due to the excessive size of input images (for T-RexNet as well as other object detectors), in compliance with the approach [64] each 2000 $\times$ 2000 image was split into a set of 16 partially overlapping pictures, each holding 512 $\times$ 512 pixels. To be consistent with the literature [63], true positives were only considered when the center positions of the detected boxes lied within a 20-pixel distance from the ground-truth location.

4.4.1.2 Civilian Surveillance

The CUHK Square dataset [65] addressed people detection, and included videos recorded (@30 Frames-Per-Second, FPS) by a surveillance camera monitoring a square and a road

crossing. Spatial resolution was 720 × 576 pixels. Since the original dataset featured some misdetections [33], the proper labels were manually added. The overall set of videos included 2105 detections for training and 593 detections for testing.

The quasi-horizontal inclination of the camera affected the depth of the scene and the perspective; as a consequence, people close to the camera appeared much bigger than people on the background. Thus CUHK also allowed to test the effectiveness of T-RexNet in detecting medium-sized objects.

At 30 FPS, the slow advance of walking people resulted in minimal changes between time-consecutive frames, hence the input videos were downsampled to 1 Frame-Per-Second (FPS), and the spatial resolution was resized to 512 × 512 pixels. That downsampling factor set the trade-off between the amount of motion data captured and the update frequency of the detections.

The experiments only considered valid detections when an IoU exceeded the 50% threshold with respect to the ground truth. Whenever a bounding box was associated with multiple ground truth points, the tests only considered one candidate, on a minimum-distance basis.

4.4.1.3 Tennis Ball Tracking

This setup only included one target object per frame. The example presented in Figure 4.2 points out the difficulty of tennis ball tracking in real-time detection: the small size of the ball and the motion blur caused by its fast movement made it almost undetectable even by human observers without the aid by motion data.

Tennis ball tracking lacks publicly available benchmarks, hence the training set collected 18,220 labeled frames extracted from videos of various matches and recorded at 30 FPS. To avoid overfitting, the test set included 5160 frames taken from three videos taken in different courts and with the camera placed at different heights, as per Figure 4.6c. In the following, we will refer to these videos as Court A, Court B, and Court C. In both the training and test set, the ground-truth labels were generated by using TrackNet [35], whose precision, according to the authors and to our observations, exceeded 95%. The result of such an automatic labelling method was anyway checked manually for correctness.

This scenario aimed to assess the suitability of T-RexNet for fast, real-time detections; in both the training and the test set, the frames were downsampled to 300 × 300 pixels. Only the detected boxes whose center was closer than 16 pixels to the ground truth location were considered as true positives.

FIGURE 4.6: The three test scenarios considered in this work: (**a**) aerial surveillance, WPAFB 2009 dataset; (**b**) civilian surveillance, CUHK dataset; (**c**) Tennis ball tracking, custom dataset. In each image a sub-area is zoomed to highlight the small size of the target objects.

4.4.2 Deployment

To allow a fair comparison with other methods in the literature and make the repeatibility of our experiments easier, we first performed our tests on a Desktop PC provided with an NVIDIA GTX 1080 Ti graphic card.

Then, to prove the suitability of our method to embedded edge-AI devices, we deployed it on an NVIDIA Jetson Nano [66], using its development board. This is the more resource constrained device of the NVIDIA Jetson series, a suite of hardware platforms specifically designed for bringing Artificial Intelligence to the edge. It is a System-On-Module (SoM) which features HW acceleration for deep learning and runs a proprietary modified version of Ubuntu 18.04. Basic characteristics of the SoM (not including development board) are reported in Table 4.2.

AI Performance	472 GFLOPs
GPU	128-core NVIDIA Maxwell GPU
CPU	Quad-Core ARM Cortex-A57 MPCore
Memory	4 GB 64-bit LPDDR4 25.6GB/s
Storage	16 GB eMMC 5.1
Power	5W / 10W
Mechanical	69.6 mm x 45 mm 260-pin SO-DIMM connector

TABLE 4.2: Characteristics of the NVIDIA Jetson Nano System-on-Module.

Users can set hardware utilization using a software interface. Two optimized configurations called 5W and Max-N are available. The first one limits power consumption setting a clock frequency of CPU and GPU to 0.90 and 0.64 GHz, respectively. In addition, two

cores of the CPU are turned off. In Max-N configuration all the hardware resources are set to maximize performance, at the expense of power consumption.

NVIDIA provides a toolchain based on TensorTRT. This tool provides an optimized implementation of common deep learning layers for Jetson devices. For the case of TF models, the output of TensorTRT is again a TF frozen graph where the computed layers are replaced with optimized versions. TensorTRT can adopt different data sizes when deploying a network: standard floating-point representation (FP32), half-precision floating point (FP16) and 8-bit integer representation (INT8). The experiments were conducted with the FP16 format since this provides a good trade-off between accuracy and power consumption [67]. In addition, the results proved that FP16 is indeed sufficient to reach good frame rates using Jetson Nano.

The code was developed in Python using the CV2 module and TensorFlow. The computed latency considered only network processing. Each frame was elaborated in real-time when acquired without the use of any batching strategy. The measures involved 100 images. The tests involved two versions of T-RexNet, one with input size 300×300 and another with input size 512×512. The networks were optimized using tensorRT with FP16 representation. The results measured the average inference time for optimized and non-optimized models, using the Jetson Nano with different power settings and different input sizes.

4.5 Results

This section illustrates the results of the experiments performed in the three test scenarios. According to our previous findings [67] we observed that moving the same architecture from the Desktop to the embedded platform has negligible impact on the detection accuracy, while it mostly affects speed and memory footprint. Therefore, in Sections 4.5.1–4.5.3 we first illustrate, for each scenario, our achievements using the Desktop platform and, then, in Section 4.5.4, we analyze the impact of deploying T-RexNet on the Jetson Nano.

Tables 4.3–4.5 give an overview of the comparisons, in terms of F1 scores, with existing methods in the literature.

4.5.1 Aerial Surveillance

Figure 4.7 shows the ROC curves achieved by T-RexNet and other state-of-the-art algorithms on the AOI 2 test set. The ROC curve for T-RexNet was added to the original

	Aerial surveillance
T-RexNet	0.91 (3)
ClusterNet	0.95* (0.3*)
Median BG+N	0.89*

TABLE 4.3: Comparison of F1 scores achieved in the Aerial surveillance scenario. Numbers in brackets represent the measured speed, in frames per second, with the Desktop platform. The asterisk indicates that the number is retrieved from the original paper.

	Civilian surveillance	
	Normal	**Small**
T-RexNet	0.77 (44)	0.79 (44)
Faster R-CNN	0.69 (23)	0.5 (23)
SSD512	0.73 (41)	0.59 (41)

TABLE 4.4: Comparison of F1 scores achieved in the Civilian surveillance scenario. Numbers in brackets represent the measured speed, in frames per second, with the Desktop platform. This scenario is splitted into the sub-cases of normal and small object size to highlight the results of our method when objects are particularly small.

	Tennis ball tracking		
	A	**B**	**C**
T-RexNet	0.78 (47)	0.84 (47)	0.67 (47)
SSD300	0.34 (43)	<0.2 (43)	0.23 (43)
TrackNet	>0.84* (2.2)	>0.84* (2.2)	>0.84* (2.2)

TABLE 4.5: Comparison of F1 scores achieved in the Tennis ball tracking scenario. Numbers in brackets represent the measured speed, in frames per second, with the Desktop platform. This scenario is splitted into the three videos we considered, with different camera view, court and environment. The asterisk indicates that the number is retrieved from the original paper.

plot reported in [60]. To assess the balance between recall and precision the experiments applied various threshold values on the detection confidence. We remind the reader that Recall $= TP/(TP+FN)$; Precision $= TP/(TP+FP)$; where TP, FP, FN are True/False Positives/Negatives.

T-RexNet outperformed the other comparisons in terms of accuracy, with the exception of ClusterNet [60], which scored near-optimal performances. As reported in [60], however, ClusterNet required 2–3 s per image on a Titan X GPU board, depending on the

number of selected regions; a time span of 3 s covered the inference phase to inspect the whole image for fine detection. By contrast, the inference time for T-RexNet was 310 ms per image on our Desktop platform featuring an NVIDIA GTX 1080 Ti board, which is similar to a Titan X in terms of hardware resources and computational performances.

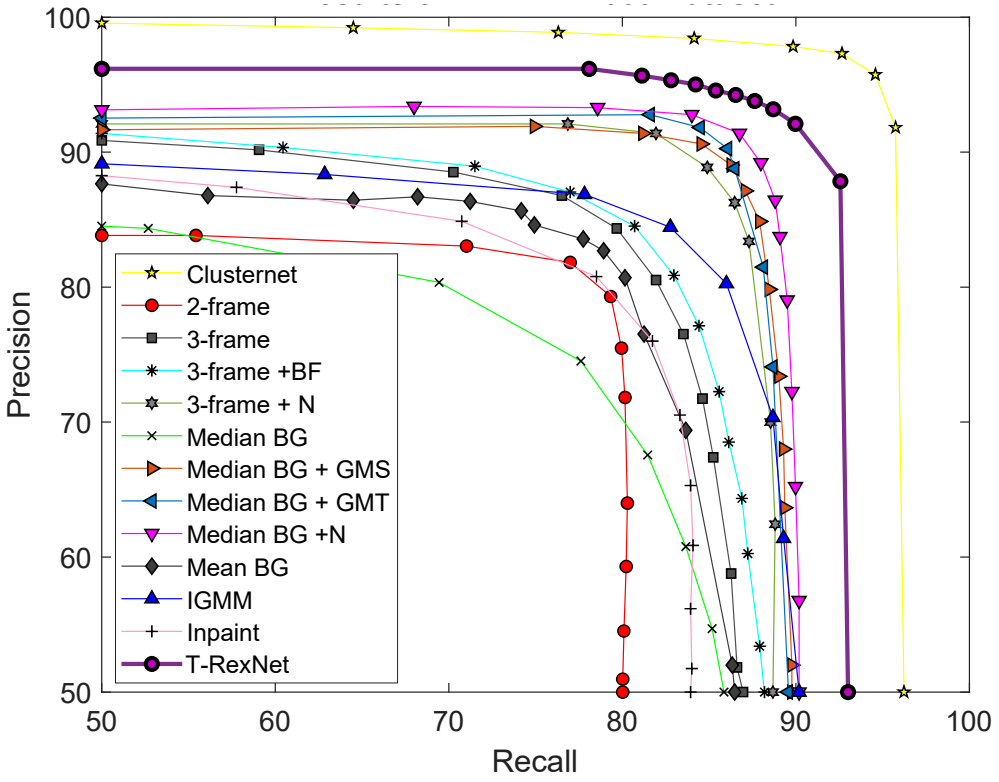


FIGURE 4.7: Comparison between the results achieved by our T-RexNet and other State-of-Art (SoA) approaches in the aerial surveillance (WAMI) scenario. We took the comparison in [60] and added the results of using T-RexNet. Despite Clusternet has better performance, our T-RexNet is ∼10 times faster. For information about the other methods we compare with refer to [60].

4.5.2 Civilian Surveillance

Figure 4.8 gives the ROC curve scored by T-RexNet in object detection within one image. The obtained results are compared with the corresponding curves attained by SSD [6] (with MobileNetv2 [62] as backbone network) and Faster R-CNN [21] (with ResNet50 [17] as backbone network). The Figure gives two curves for each comparison: the Full mark refers to experiments on whole images, whereas Small curves refer to tests only performed on the upper halves of images, where perspective made people appear smaller.

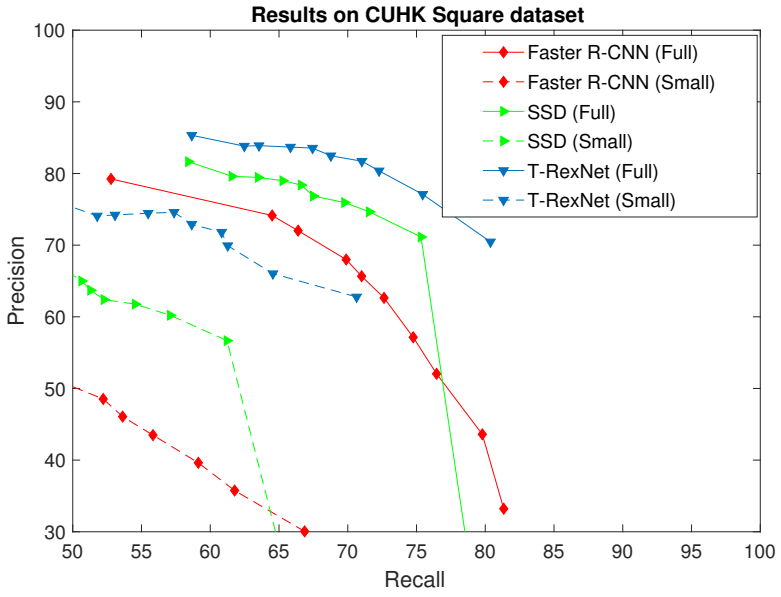


FIGURE 4.8: Comparison between the results achieved by our T-RexNet, SSD and Faster-RCNN in the civilian surveillance test case using the CUHK square dataset. Full and Small indicate whether the test has been conducted over the whole image or the upper half only, where perspective makes people much smaller and the gap between our approach and others is even more pronounced.

The ROC curves in Figure 4.8 witness that motion information greatly helped T-RexNet achieve the best performance. More, T-RexNet was the only architecture that attained satisfactory results when focusing on tiny objects.

4.5.3 Tennis Ball Tracking

Figure 4.9 shows the ROC curves measured by applying T-RexNet on the test sets Court A, Court B, and Court C. The graph also give the associate ROC curves obtained by MobileNetv2-SSD, which represented the single-image architecture from which T-RexNet evolved. The comparison pointed out the significant impact of involving motion data in the detection of the target object.

Experimental outcomes prove that T-RexNet featured a remarkable improvement over State-of-the-Art, application-independent approaches. When considering application-specific solutions, TrackNet [35] had generated our ground-truth labels and proved more accurate than T-RexNet in tennis-ball tracking. As reported in the original paper, TrackNet attained on average higher F1 scores than 0.84, which was consistent with the test performed in this research. At the same time, TrackNet proved significantly heavier than T-RexNet: Python implementations of both, running on the Desktop platform, resulted in 2.2 FPS for TrackNet and 47 fps for T-RexNet, that is ∼21 times faster. The limited resolution of input images allowed to increase the batch size in the inference

phase up to 10 consecutive frames, while still fitting the memory of the test GPU. This batch approach allowed T-RexNet to run at 96 fps, at the price of an increased latency from 21 ms to 104 ms.

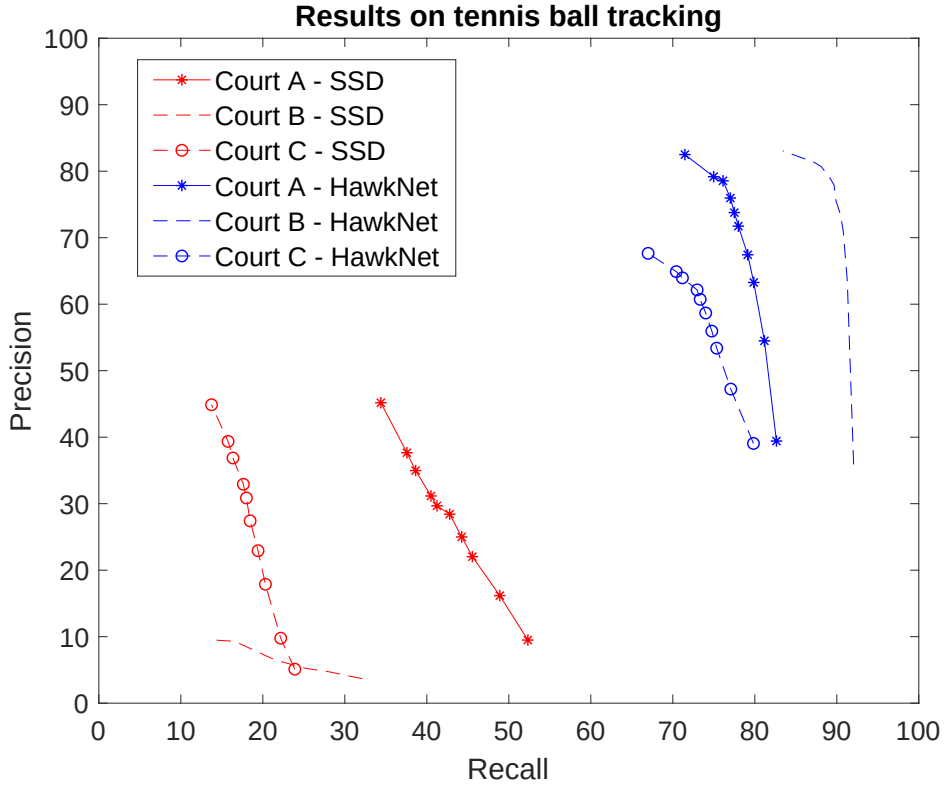


FIGURE 4.9: Comparison between the results achieved by our T-RexNet and SSD in the tennis ball tracking test case. Due to the motion blur and the small scale of the ball, it becomes almost undetectable by SSD, since it does not exploit motion data.

4.5.4 Deployment of T-RexNet on the Jetson Nano

This section presents the results of the deployment on Jetson Nano. Table 4.6 shows on the rows the power setting of the board. Columns are divided into couples. The first pair reports the result for input size 512 × 512, the second refers to 300 × 300. The first column of each pair refers to an optimized model with FP16 representation. The second column indicates the original TF model.

The results reveal that T-RexNet can be deployed in embedded systems with real-time performances. In Max-N configuration, the network can process a frame in 70.28 ms. In other words, the device could elaborate 13 FPS, which is acceptable for many applications. The comparison with native TensorFlow solutions highlights the importance of optimization combined with FP16. A similar observation holds for 5W power mode.

Power mode	**512x512**		**300x300**	
	TRT(ms)	TF(ms)	TRT(ms)	TF(ms)
Max-N	70.28	437.15	65.45	431.28
5W	108.77	616.74	98.28	661.14

TABLE 4.6: Inference time measured on the NVIDIA Jetson Nano device for every combination of image size, power mode and optimization level.

Memory requirements for this network are quite limited. The *pb* file, that is the TensorFlow's ProtoBuf file containing the description of the network, measures around 3.0 MB. The memory strategy implemented on Jetson Nano allocates a large amount of memory that is not directly dependent on the model size. Accordingly, a direct measure would yield biased results. Indeed, literature proves that similar models can be deployed in devices using a smaller memory footprint [67].

4.6 Conclusions

The T-RexNet approach involves a deep neural network for the detection of small moving objects. Using a fixed camera, it exploits motion data as a discriminant contribution whenever visual-only information is limited due to the small target sizes. The T-RexNet architecture includes a two-path network, while keeping computational costs low. The method's relevant features consist in limiting computational and memory costs, allowing real-time execution, ensuring reuse in several applications with an end-to-end approach, and yielding remarkable accuracy performances that favourably compare with SoA approaches. T-RexNet was tested in three real-world scenarios covering a wide range of applications. Accuracy results confirmed that the proposed method outperformed most of SoA approaches; conversely, when considering execution speed, T-RexNet improved over the most accurate methods. Tests performed on an NVIDIA Jetson Nano proved that our solution is suitable for deployment on embedded edge devices. In conclusion, we believe T-RexNet can be regarded as an easy-to-use alternative, suitable for embedded to high-end devices, to deal with tiny moving targets observer through a fixed camera.

4.7 Appendix: Hyperparameters and training details

In this section we use S1, S2, S3 to denote our three scenarios: aerial surveillance, civilian surveillance, tennis ball tracking.
T-RexNet has been implemented in Python, version 3.6.9, using the TensorFlow library,

version 1.14. All the models have been trained from scratch using an NVIDIA GTX 1080 Ti graphic card. The batch size has been set to 32, and the number of training steps to 20.000 in S1 and S2, and 60.000 in S3. RMSprop optimizer has been used, with a learning rate of 0.004, momentum and decay equal to 0.9. Hard negative mining was used to balance positive vs negative (background) classes, with a maximum of 3 negative per positive examples. All the convolutions use L2 regularization with a weight of 0.00004. No dropout has been implemented. The default size of the anchor boxes has been set to 0.2 times the size of the image, while the aspect ratio has been choosen on a scenario basis: 1 in S1 and S3, 2 in S2. Training time lasted approximately 10 hours per scenario.

Chapter 5

Allowing on-target retraining: Hand image classifier

In this Chapter our AI optimization goal has moved from focusing on speed and accuracy, like in Chapter 4, to allowing on-target retraining. As a reference application, we considered a hand image classifier.

State of the Art solutions for image classification are nowadays mostly based on Convolutional Neural Networks (CNNs). With respect to running them for inference, training CNNs is a much more resource demanding job. For this reason, embedded devices designed to execute neural networks are normally not suitable to train the same networks. Hand image classification is becoming increasingly popular in several application areas, by empowering embedded systems with the ability to recognise hand gestures or grasp types. In this field, making it possible to retrain the hand image classifier directly on the embedded device would allow an easy customization of classification task according to application specific needs. In this Chapter we present a method based on transfer learning which approaches this problem by first extracting hand keypoints, using with a dedicated neural network, and then applying on the resulting data a lightweight classifier, easily retrainable directly on the target embedded system. The proposal has been validated on a new dataset collected by the authors, outperforming state of the art solutions in the specific task.

5.1 Introduction

State of the Art techniques for image classification typically rely on Convolutional Neural Networks. For each specific classification task, CNNs need to be trained by using

relatively large datasets of labelled images. Even when the final classifier is designed to be deployed on an embedded device, the network training normally needs to be run on a desktop or server environment, given the higher amount of computational resources needed. In many applications, this limitation does not represent an issue, since the classification task is not supposed to change over the product lifetime and can therefore be learned only once. However, there are fields where making the user able to customise the image classification task directly on the target embedded device would be beneficial. Among these, in this work we consider the always growing field of prostheses and rehabilitation technologies empowered with computer vision to perform classification of hand images. In upper limb amputees, robotized prostheses enhanced with computer vision can be made aware of visual information of the good hand in order to perform autonomous context dependent actions and enable bi-manual interaction [68, 69]. For example, the user could use the good hand to control the robotic one via gestures, or the robotized hand could reshape itself according to the type of grasp the good hand is doing [70].
Clearly, in these fields, users would greatly benefit from the possibility to customise by themselves the hand classification task to adapt it to application specific usages, by performing a semi-automated training procedure. This requires the neural network based classifier to be trainable on the target embedded device. Training deep learning models on single edge devices is a relatively new topic because most work is still spent on the implementation and optimization of the inference phase [71]. The most active research line for training on constrained environment envisions an edge server that coordinates the training process with multiple edge devices [72]. Each device performs part of the global computation and the results are sent back to the server and combined. Other interesting options to share the work load on different nodes came from distributed training strategies inspired by online distillation [73]. However, all these solutions assume a sufficient support from hardware resources. This assumption is rarely meet by most of the constrained devices typically employed in real-world applications.

In fact, making it possible to let the user collect a new custom dataset and train a CNN in a reasonable amount of time on an embedded device, one has to overcome two main issues:

- The memory and computational complexity required for training the neural network must be contained.
- The amount of data (labelled images) the user has to manually collect to train the network has to be reasonably low.
- The neural network must be able to learn relevant features even in case of a bad dataset, for example avoiding overfitting on the image background or illumination.

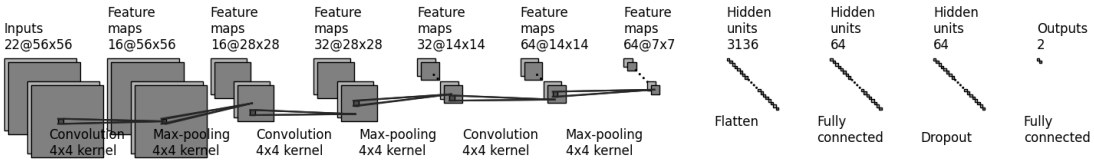


FIGURE 5.1: Architecture of the classifier we selected. The input is a series of 22 heatmaps, each corresponding to one of the hand keypoints. The output is the probability for each of the possible classes: in our experiments pinch and grasp.

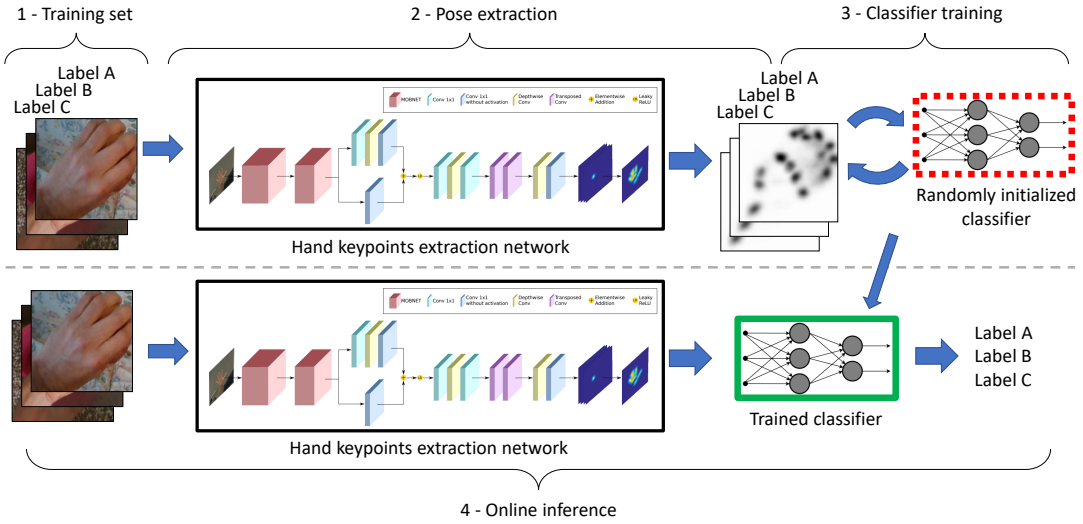


FIGURE 5.2: Overview of the steps proposed in our methodology to train our custom hand image classifier (upper part) and use it for inference (lower part).

Considering the use cases of classification of hand images already cited, in this work we focus on retraining a classifier of grasp types. Relevant work in this field is represented by [74] and [75], where CNNs are used for hand detection and then grasp classification. Focusing on embedded systems, [76] introduced a novel framework for video-based grasping classification on an edge device. In [77] an automatic hand pose annotation system is used to enhance the grasp classification task on embedded devices. However, to the best of the author's knowledge, no study has been conducted on the possibility to customize a grasp classifier by retraining the neural network directly on the embedded system. We propose a method which addresses this task overcoming the above explained issues and being able to: (i) be trained directly on the embedded target device and (ii) learn the classification task from only a few hundreds of video frames. To validate our solution, we created a dataset for grasp type classification. Although the literature recognises several types of grasp [78], in this work we distinguished only between *power grasp* (used to lift a heavy object) and *pinch grasp* (used for example to hold a pen). Since the previously cited [76] and [77] prove that MobileNetV2 [79] is a valuable backbone CNN for grasp classification in resource constrained environments,

we considered it as our reference. In the specific case of challenging image background and low amount of training data, which represent a realistic scenario in the context of this work, our solution proved to be able to outperform a MobileNetV2-based classifier. Moreover, we proved that our classifier, differently from MobileNetV2, can be trained in a few minutes on an NVIDIA Jetson Nano, that is a resource constrained embedded device for deep learning applications.

5.2 Method

The focus of this work is the design of an hand image classifier that can be hosted on embedded devices. Actually, the design of the classifier should allow also the training process to be run directly on the device, i.e., without exploiting computational resources provided by other devices. To this purpose, the proposed research relies both on the setup of a lightweight architecture and on the adoption of a learning process that reduces the amount of training samples.
In this section, we present our method in detail. For clarity, the section is divided into three parts: in a first subsection, we explain the algorithmic foundations of our method; in the second subsection we show how it can be deployed on an embedded device; in the last subsection we discuss the particular use case we considered for validating the proposed idea.

5.2.1 Model

Our method is based on the key idea that in most applications one of the most important sources of information when classifying images of hands is the hand pose. The classification of hand gestures or grasp types provide two good examples: such problems may be tackled focusing on the shape of the hand, rather than on the specific object the hand is using or on the background. This approach seems indeed convenient when the goal is to limit the amount of training samples required to achieve a reliable classifier. In fact, one can discard unuseful features such as those characterizing the hand color tone or the background, which only increase the risk of overfitting in the training process.

In the proposed design, a lightweight architecture is obtained by exploiting transfer learning. Thus, a hand pose estimator is utilised as feature extractor; on top of it a custom lightweight classifier represents the only trainable part of the neural network.

Figure 5.2 gives an overview of the complete framework we propose. The upper part of the figure refers to the training process; the lower part of the figure refers to online inference exploiting the trained model. Here we summarise all the steps involved:

1. **Training set**: a set of images centred and focused on a single hand is acquired; each image shall be assigned a custom label corresponding to the action, gesture, or grasp type the hand is performing.

2. **Pose extraction**: the input image is processed by a CNN-based pose estimator to get the hand pose tensors and filter out unuseful information such as the background. The details on the pose estimator adopted in this research are discussed later in this section.

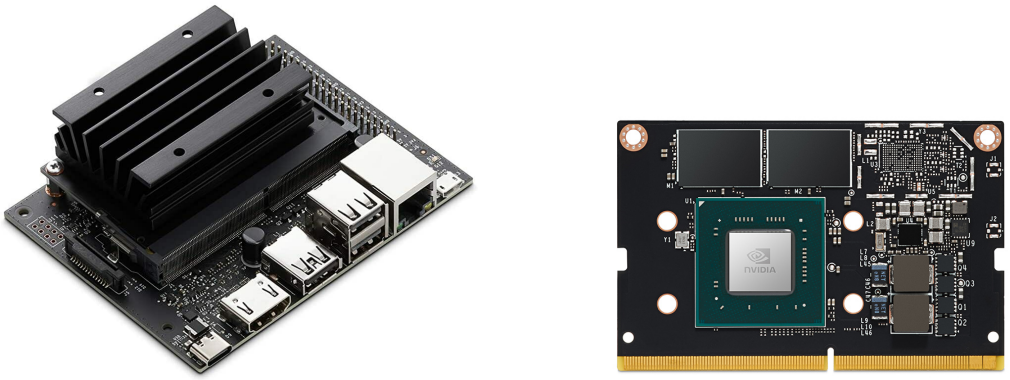
3. **Classifier training**: A classifier is trained to infer the image label starting from the hand pose tensor. Optionally, model selection can be performed running this task for several classifier configuration trials and then selecting the one with the best performance. The classifier is discussed in more detail later in this section.

4. **Online inference**: in the online inference phase, the trained system classifies in real time the input images. Again, the input image is first processed by the pose extractor; then, the trained classifier provides the prediction on the extracted hand pose.

The two main blocks of the proposed model are the pose estimator and the classifier. In the following we give more details about the architecture supporting those blocks.

The *hand pose estimation network* adopted in this research is taken from the work of Gouidis et al. [80]. It is based on an encoder-decoder network: the first layers of the network take in input a 224x224x3 colored image and perform encoding using the same feature extractor of MobileNetV2 with minimal changes; the final decoding layers compute the 22 output heatmaps following an approach inspired by [81]. Each heatmap is 56x56 pixels wide and corresponds to the probability of having in that point of the image one of 21 identified hand joints. One additional heatmap represents the likelihood of the background. This pose estimation network is particularly lightweight and suitable for execution on embedded devices, including smartphones [80].

The *classifier network* takes in input the 56x56x22 tensor computed by the pose estimator and gives as output the likelihood of each label for the given input. The detailed architecture of the classifier is depicted in Figure 5.1. It is composed by a series of three pairs of 2D Convolution and max pooling layers, followed by a fully connected layer with dropout, and then a last fully connected layer to get one output per class. On the final output, a softmax function is applied to get the final likelihood per each class. The final classifier architecture was the result of a model selection procedure we performed on our specific use case; details will be discussed in Section 5.3. Actually, model selection is an optional step that the user can perform to increase the classifier performance at the cost



(A) Development board (B) Production board

FIGURE 5.3: NVIDIA Jetson Nano board. Figure (a) shows the development kit we used, where the core board is integrated with additional HW to enable the usage of all the provided interfaces. Figure (b) shows only the core board ready for integration in custom embedded systems for production purposes.

of a longer training time. Training different models in series, in fact, does not increase the hardware requirements of the embedded device.

Overall, the key feature of the proposed model is the structure of the pipeline, where only the classifier is subject to training. In fact, the hand pose estimator can be trained offline only once by exploiting high-end computing platforms with a dataset large enough to achieve robustness. Thus, by exploiting transfer learning one can design a model that is at the same time fast on edge devices -both in training and inference- and suitable for the specific hand image classification task. In this sense, such architecture is expected to be more effective than a state-of-the-art, general- purpose image classifier such as MobileNetV2. The experimental results presented in 5.3 will confirm such statement.

One may observe that transfer learning can be applied also to MobileNetV2; in this case, one would train only its classifier. Section 5.3.1 will address this aspect by comparing the performance of the proposed architecture with that of a MobileNetV2.

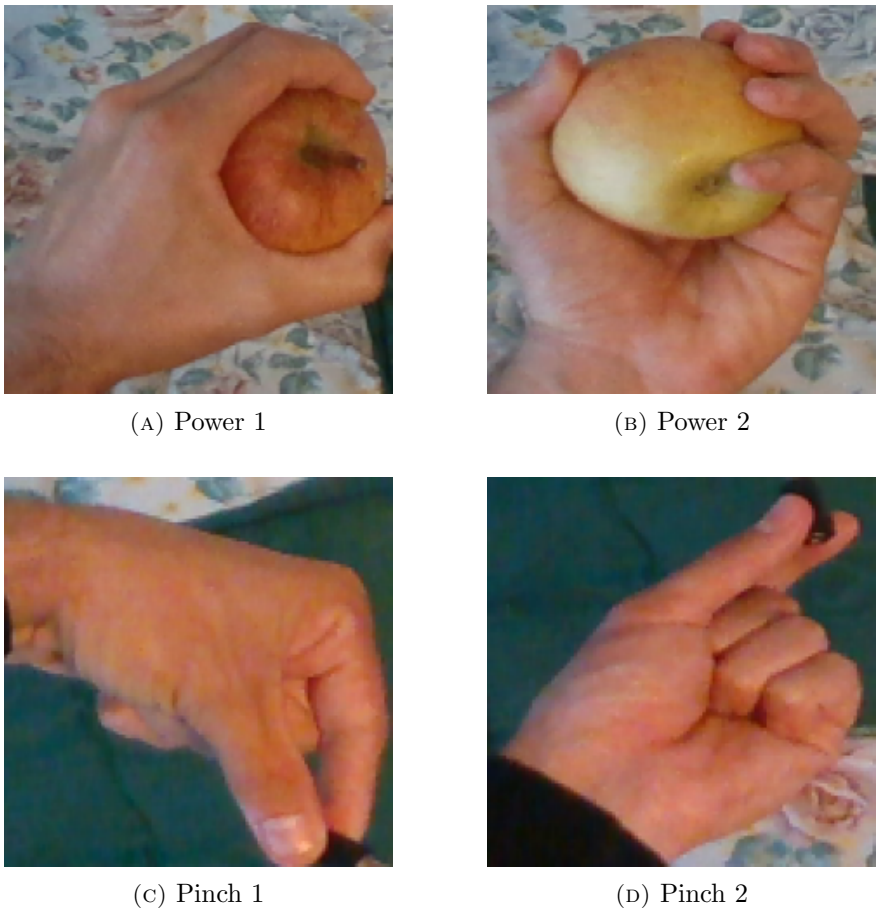
5.2.2 Deployment to embedded device

The complete pipeline described above can be implemented on an edge device suitable for deep learning applications. In this research we adopted the Jetson Nano, from NVIDIA [39]. Figure 5.3 shows both the development kit and the production board, which is only the core of the device, suitable for integration in custom embedded devices for production. The Jetson Nano features a Quad Core ARM A57 CPU, 4 GB of RAM, a dedicated GPU, several interfaces, and can host non volatile data on a SSD card.
When deploying our method to the embedded device, the *dataset acquisition* was completed by using a camera directly connected as an input peripheral to the device; the

whole dataset was stored in the SD card of the board. Details of the dataset and use case we choose are given in Section 5.2.3. In general, by using a portable embedded device, the user would be allowed to collect his custom training dataset by wearing the camera in such a way to have the desired hand in the camera's field of view. Through an automatic routine running on the embedded device, the user can be guided through the collection of the dataset by simply performing with the hand the actions or gestures in the scope of the custom classification task he wants to achieve.
In the *training phase*, pose extraction and classifier training are executed in series on the device; intermediate data are stored in the non volatile memory. The resulting trained classifier is also stored in the non volatile memory.
Online inference is executed in real time by acquiring the hand images directly from the peripheral camera.

5.2.3 Use case: grasp classification

Although our method can be in general applied to the classification of actions and gestures performed by hands, in this work we considered the use case of grasp type classification. [78] presents a detailed categorisation of human grasp types: 33 different grasp types are identified and categorised on the basis of the "strength" (precision vs power) of the grasp, the position of the thumb and other features. In the present case, we got inspired by the above classification and we considered only the coarser classification into "power" grasp, used for holding heavy or big objects, and "pinch" grasp, suitable for precision. To this aim, we collected a dataset whose details are reported here below. Using a camera mounted on the head of a person, we collected a total of 4.000 pictures of his left hand holding an object. 40 of the 80 considered objects were held in a "power" manner, that is with a considerable part of the hand around the object, due to his size or high weight; the other 40 object were held in a "pinch" manner, that is, like in the case of a pen, to have more precision. "Power" and "pinch" were the target classes of our classification task. A total of 50 images per object have been taken, covering many possible camera frame angles with respect to the hand. In order to challenge the classifier to be robust to learning only the meaningful content of the image, the backgrounds have intentionally been chosen particularly different from one object to another. Figure 5.4 shows 4 samples, 2 per grasp type, of our dataset. In order to get images well centred on the desired hand, we did manual cropping of the original images. In applications where the camera is embedded in a wearable device mounted, for example, to have an egocentric view, this task shall be automatised by mean of a hand detector. However, state of the art solutions for this task already exist in the literature and are not in the interests of this work [82].



(A) Power 1 (B) Power 2

(C) Pinch 1 (D) Pinch 2

FIGURE 5.4: Samples from our dataset of grasp types. For each of the Power and Pinch classes of grasp type we considered 40 object, and for each object we shot 50 images at different angles, getting a total of 4000 images.

5.3 Experiments and results

In this section we describe the experiments we performed and comment the results. Notice that all the experiments have been conducted on the grasp classification use case and related dataset, described in 5.2.3.

5.3.1 Experiment 1: Classifier model selection

As outlined in Figure 5.2, the proposed solution consists of a feature extraction part, where the hand pose is detected, and a classifier network. To choose the specific architecture of our classifier network, we tested 8 different variations of the architecture depicted in Figure 5.1, by changing only (α) the number of convolution-max polling consecutive pairs, (β) the number of channels in each convolution-max pooling pair, and (γ) the number of neurons in the first fully connected layer. To get a meaningful result, each variant has been tested on the dataset 10 times in a leave-one-out manner: in each

of the 10 trials, the dataset was split to have, for each of the two labels ("power" and "pinch") 36 objects for training, 2 for validation and 2 for testing. Objects left out from the training fold were changed in the 10 trials in order to cover the whole dataset uniformly. Since the classifier is trained on the features extracted by the pose detection network, the dataset has first been preprocessed to get these features (step 2 in Figure 5.2) and then the classifier has been trained (step 3 in Figure 5.2).
The network was implemented in Python 3.8 using Keras 2.4.3.; the batch size was set to 32; the learning rate was fixed to 0.001. An early stopping strategy has been used to avoid overfitting the training set: the training was automatically stopped in case the best achieved validation accuracy had not been improved in any of the last 10 epochs (training patience). Since these tests were only intended to select the best classifier architecture and measure its classification accuracy, they were ran on a desktop environment. Results achieved training directly on the embedded device are exposed in Section 5.3.3.
Table 5.1 summarises the results. The average accuracy over the 10 trials has been considered.
Interpreting the table, we can see that the best result is achieved by using the classifier depicted in Figure 5.1, where a total of three 2D Convolution layers are used, each with 16,32 and 64 filters respectively, and having 64 nodes in the fully connected layer. Table 5.1 also shows that a much lighter network which uses a sequence of 3 2D convolution layers with only 8 filter each and 32 nodes in the fully connected layer is still able to achieve 0.68 of accuracy, thus allowing the user to select the appropriate classifier configuration according to the available hardware resources.

		# of filters in fully connected layers	
# of convol / max pool layers pairs	**# of filters per convol / max pool layer pair**	**32**	**64**
3	**4,8,16**	0.67	0.68
	8,8,8	0.68	0.67
	16,32,64	0.68	<u>0.71</u>
4	**4,8,16,32**	0.67	0.69

TABLE 5.1: Results achieved during the classifier architecture selection. Numbers in bold are parameters. On the botton right of the table we have the results in terms of accuracy in the test set eachieved by each of the 8 configurations. The best result is underlined.

5.3.2 Experiment 2: Accuracy comparison with MobileNetV2

To get confirmation that the advantages of our method do not come at the expense of accuracy, we ran the same 10 trials using MobileNetV2 as classification network, given its proven suitability for grasp classification on embedded devices [76][77]. Hyperparameters of the training were not changed compared to the previous test. MobileNetV2 can be trained (i) from scratch, (ii) by transfer learning, (iii) by transfer learning and fine tuning. Considering the network composed by a feature extraction and a classifier subparts, the three methods differ in the way the network is initialized before training, and in which parts of it are trainable. In the first case, the network is randomly initialized and all its layers are trained; in the second case a pretrained network is taken, and only the classifier part gets retrained; in the third case also the feature extraction gets retrained. Following common practices in image classification, we did not take into account training the network from scratch, which would also be out of scope for our work. We started from MobileNetV2 pretrained on the ImageNet dataset, then we performed first transfer learning only, and finally we applied fine tuning.
The results are reported in Table 5.2 and compared with our method. Given the widely recognition in the literature of MobileNetV2 as a well performing architecture in the generic image classification task, the reason of our better results must be found in our adoption of a preprocessing strategy specifically designed for the case of images containing hands, which resulted able to filter out unrelevant information, such as the background. In case of a limited number of training samples, in fact, a generic classifier like MobileNetV2, may easily tend to overfit the training data, possibly learning background features instead of hand features.

	MobileNetV2		
	TL	**TL + FT**	**Our method**
Accuracy	0.62	0.65	0.71

Table 5.2: Accuracy of the MobileNetV2 classifier by using the transfer learning (TL) and transfer learning plus fine tuning (TL + FT), compared to our method.

5.3.3 Experiment 3: Training on the embedded device

To prove the suitability of our method for training on an embedded device, we implemented this procedure on an NVIDIA Jetson Nano board. On this device, we repeated the training procedure done in Experiment 1 (step 3 in Figure 5.2), but, instead of performing model selection on the classifier, we considered the classifier architecture which

gave the best accuracy (0.71) in Experiment 1, depicted in Figure 5.1.
Results showed a training time of roughly 9 seconds per epoch. Due to our patience-based early stopping criteria explained in Section 5.3.1, trainings stopped on average after 20 epochs, that is a total average training time of 3 minutes. Although here model selection has not been considered, it is possible to implement it simply at the cost of higher training time. It must be noticed that the total training time shall include, in addition, the time taken to extract the hand pose from the raw image. This step corresponds to step 2 in Figure 5.2 and is executed both for the training as well as for the inference phase (step 4). Its execution time is assessed in Experiment 4. As already mentioned, trying to train MobileNetV2 on the embedded device as an alternative solution, turned out to be impossible due to its higher memory demand. To give a quantitative idea of the "training effort", Table 5.3 summarises the number of trainable parameters for the two solutions. Our proposal has roughly 10 times less trainable parameters than MobileNetV2. It must be noticed that training MobileNetV2 with transfer learning only (without feature extraction fine tuning) would lead to have only the 1.281 parameters of the classifier to train. However, as already highlighted in Table 5.2, in this way the neural network does not manage to learn the task robustly.

	MobileNetV2	**Our proposal**
Feature extraction	2,225,153	0
Classifier	1,281	246,602

TABLE 5.3: Number of trainable parameters between MobileNetV2 and our proposed method. Notice that the hand keypoints feature extractor does not need to be retrained since it is application independent.

5.3.4 Experiment 4: Inference on the embedded device

Once the classifier is trained, it is ready for online use on the embedded device. This step corresponds to number 4 in Figure 5.2. Like we did for the training part, two substeps are needed: extraction of hand keypoints through a dedicated Neural Netowrk (feature extraction), and classification through our custom trained classifier. In this test, the Jetson Nano was set in performance mode. Table 5.4 reports the inference time for each substep and the total time. With 72.8 ms of average inference time, corresponding to 13.7 frames per second, our method proved to be suitable for real time execution. Since our classification is based on the pose of the hand, which normally is not supposed to change at very high frequency, we consider the achieved speed as sufficient for most online applications of our method.

Feature extraction	Classifier	Total
56.9 ms	15.9 ms	72.8 ms

TABLE 5.4: Average inference time in milliseconds on the NVIDIA Jetson Nano for each part of our complete neural network.

5.4 Conclusions

Training an image classifier on a resource constrained device is nowadays still considered a challenging task. The problem is even more difficult When the training dataset has to be collected by the end user, thus being reduced in number of samples and quality. In this work we showed how, in the specific case of hand image classification, this task can be made possible by first extracting from the hand image a representation of the hand keypoints, and then training the classifier only on this more compacted and meaningful representation. Results confirmed that (i) our method has better accuracy compared to other approaches used in the literature for the specific task; (ii) it can be trained in a few minutes on an embedded device; (iii) it requires a relatively low amount of training samples; (iv) it can be executed in real time.

Chapter 6

Deploying DNN to microcontroller via manual algorithm design: object detection for surveillance

In this Chapter we face the challenge of deploying a DNN to a microcontroller, which represents an extreme scenario given the very low resources that typical microcontrollers exhibit. As a use case, we considered an object detection system for surveillance applications.

Using object detection techniques, nowadays mostly based on deep neural networks, new intelligent camera-based surveillance systems can be designed, capable of generating alerts only in the presence of specific objects, like persons, in the camera field of view. However the memory and computational load required by these techniques makes it challenging to use them on low power, miniaturised and resource constrained surveillance devices designed for harsh environments. In this Chapter, we show an efficient method to detect the presence of a specific object in surveillance video frames using deep neural networks on an STM32 microcontroller, suitable for harsh environments. Our solution achieved 97% precision and 93% recall, while consuming less than 400 mW.

6.1 Introduction

In many cases computer vision models, often realized via DNNs, are expected to run on resource-constrained embedded systems. Drones [83], wearables [84], (semi-)autonomous

robots and vehicles [85] are examples of applications where one should deal with constraints on power, size, costs and computational performance of the embedded system entitled to run DL models. Accordingly, the market has started to provide embedded devices designed to host DL architectures; the Jetson series from NVIDIA [39] and hardware accelerators such as Google Coral [86] and Intel Movidius [87] are good examples of such devices. At the same time, software frameworks have been developed for the design of lightweight versions of state-of-the-art DL architectures (e.g., TensorFlow Lite [88]). In this case, techniques such as binarization and quantization are exploited to make feasible the deployment of a given architecture on a resource-constrained device. Furthermore, novel DL architectures are being proposed that specifically aim at the minimisation of both the memory footprint and the computational load.

The present research focuses on DL architectures for object detection. On the one hand, state-of-the-art architectures can attain very good performance on object detection. On the other hand, resource-constrained embedded systems in general struggle in adopting such solutions. This issue emerges in particular when targeting very low-resources edge devices. This work tackles such issue by proposing a DL-based solution for object detection that can be deployed on the STM32 family of 32-bit microcontrollers. The target device features 1 Mbyte of program memory and 320 Kbyte of RAM. Hence, the goal is to show that DL solutions for object detection can be customised to match the constraints of low-cost edge devices, even without sacrificing target object generalisation as in face detection systems. Such outcome obviously can have a significant impact when looking at the applications of computer vision.

The use case analysed in this work is a semi-autonomous battery-operated video surveillance device. The device is expected to periodically grab a frame from the video captured by the camera to the purpose of triggering an alarm whenever specific events happen (e.g., an object that enters the camera's field of view). In the envisioned scenario, the device must a) be as small as possible, and b) suit very low-power applications to enable its utilisation in harsh environments or as on board system in drones. Indeed, a relaxation of timing constraints is allowed to meet power constraints.

This work shows that in applications where the object detection task is aimed more at revealing the *presence* of an object in the camera's field of view rather than object counting or defining bounding boxes, an approach based on segmentation Deep Neural Networks (DNNs) helps to dramatically decrease the memory footprint of the algorithm. As a major result, the inference process can be executed on a very resource constrained device. We implemented the proposed method on an STM32 microcontroller, achieving 97% precision and 93% recall on a video surveillance dataset for person detection. Our implementation requires only the selected microcontroller, an external SDRAM and the

camera, thus being extremely compact, lightweight and cost-effective. It does not exceed 400mW of power consumption when working at maximum performance. To the best of the author's knowledge, this is the first implementation of this kind of object detector in such resource constrained conditions.

The rest of the Chapter is organized as follows: Section 6.2 describes our work starting from the selected HW platform, then moving to an analysis of the challenges of deploying DNNs for object detection on embedded systems, and finally giving the details of our method; Section 6.3 provides details on our dataset and person detection use case; finally, in Section 6.4 we describe our experiments and results.

6.2 Material and Methods

The present research focuses on the design of a DL-based model for object detection that can be hosted on a low-power low-memory edge device. In this work, the target device is a 32-bit microcontroller of the STM32 family: the STM32F746NG. Thus, the envisioned solution does not exploit any DL-oriented hardware accelerator.

	STM32F746NG	**Jetson Nano**
Type	Microcontroller	System-On-Chip
Unit price	17$	129 $
Data bus	32 bit	64 bit
Max frequency	216 MHz	1479 MHz
Core	Arm Cortex M7	Arm Cortex A57
Floating point unit	Yes	Yes
Program memory	1024 KByte	SD Card
RAM	320 KByte	4 GByte
Operating temp.	-40 , +105°C	0 , +60°C
Size	13,15 x 13,15 mm	69,6 mm x 45 mm
Interfaces	SPI, I2C, USB, USART and more	SPI, I2C, USB, USART, HDMI, HD cam. and more
Power (RUN mode)	345 mW	5.000 - 10.000 mW
Power (STOP mode)	7 - 891 uW	NA

TABLE 6.1: Comparison between the STM32F746NG used in this project and the NVIDIA Jetson Nano which outlines the hard resource constraints we face in this work.

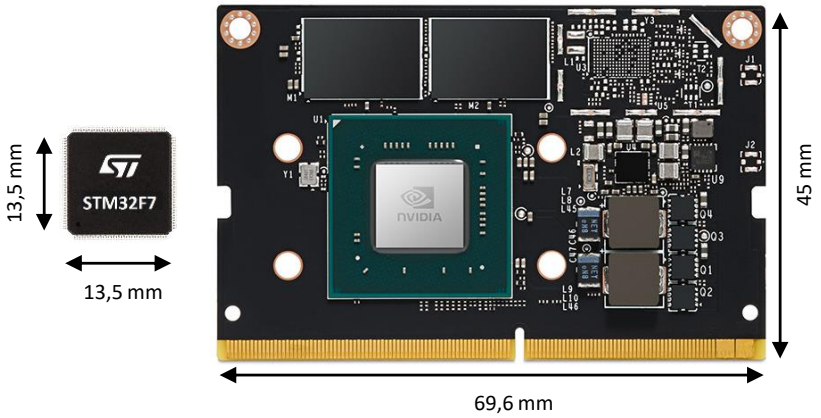


FIGURE 6.1: Our STM32 device (left) vs NVIDIA Jetson Nano (right) footprint comparison.

6.2.1 The edge device: STM32F746NG

The STM32F746NG microcontroller is based on the ARM Cortex-M7 32-bit RISC core, which operates at up to 216 MHz and is designed for high performance applications in environments from -40 to +105 °C. The Cortex-M7 core features a single floating point unit precision which supports all ARM®single-precision data-processing instructions and data types. It incorporates a 1 Mbyte Flash memory, 320 Kbytes of SRAM and an extensive range of enhanced I/Os and peripherals. It also supports external memory access, which can be efficiently used via a Flexible Memory Controller (FMC) and standard as well as advanced communication channels. In terms of sizing, this device extends on an area of 13.15mm x 13.15mm.

The STM32F746NG must be supplied with a voltage between 1.8V and 3.6V, and features several low power modes which ease the development of energy constrained applications. In stop mode, where the device can be woken up by an internal real time clock or external interrupt, it can use as low as 7 µW, making it suitable also for non continuous operation in battery powered conditions. Table 6.1 outlines the differences between our device and the NVIDIA Jetson Nano, a well known embedded platform for implementing artificial intelligence at the edge: at the cost of reduced memory and computational resources, our device is smaller, requires less power, is more suitable for harsh external environments, and comes at a cheaper price.

6.2.2 Object detection on a low cost microcontroller: challenges

The deployment of DNN-based object detection on low-power, general purpose microcontrollers involves a few challenges. The memory footprint of the DNN in particular

plays a major role, as one should deal with a very limited amount of flash memory and RAM:

- **Flash memory - program**: in the case on DNN execution, this memory stores the actual code which implements the inference function, i.e., the sequence of operations which process the DNN input down to the output. Since most DNNs perform repeated operations from a limited set of known layer types, this part can easily be arranged in reusable functions, and therefore does not represent a significant contribution to the overall memory footprint.
- **Flash memory - data**: this segment of memory should store all the parameters of the DNN, i.e., a very critical factor for a DL architecture.
- **RAM**: this memory is exploited to store temporary data (mostly tensors at the output of a layer) while executing the inference function.

The size of the required data segment in the flash memory and the size of the required RAM memory strongly depend on the specific DNN architecture. In particular, for convolutional DNN, the amount of parameters to be stored in the flash memory is mostly influenced by the number of channels and size of the different kernels involved in each single layer.

The RAM memory should store temporary data. In the case of the most simple implementation of a DNN inference process, the RAM should mainly store the intermediate tensors. Assuming that a feedforward network is implemented and no pipelining mechanism are involved, each intermediate tensor is stored in RAM from the moment it gets computed to when all the following layers which take that tensor in input have been executed. Figure 6.2 gives a graphical representation of this concept. According to it, the total RAM requirement M_{req} depends on the sum of the size of all the tensors T_x that need to be kept in memory at the same time, due to data dependency. In practice, skip connections (T_{3b} in the figure) will increase the usage of RAM, while in a network without skip connections the total RAM requirement would depend only on the largest tensor along the network. Table 6.2 summarizes in a qualitative way the effects of network parameter changes on Program Memory, RAM, and number of operations.

One can refer to MobileNetV3 for object detection for a quantitative example. This DNN is one of the most lightweight architectures for the task. According to its authors, the "small" version uses 1.77 M parameters, which would need 7.08 MBytes of data segment in the flash memory. This amount of memory is not available in common microcontrollers. Although one could consider adding an external Read-Only-Memory (ROM) just to store the DNN parameters, this choice would have several drawbacks: it

	Impact		
Network parameter	**Program mem.**	**RAM**	**# of oper.**
Input tensor dim	-	↑	↑
# channels in conv	↑	↑	↑
Kernel size in conv	↑	-	↑
Stride in conv	-	↓	↓
Skip connections	-	↑	↑

TABLE 6.2: Examples of qualitative impact on network footprint due to changes to commonly used arameters in convolutional networks. ↑ and ↓ means, respactively that the footprint increases or decreases when the parameter increases.

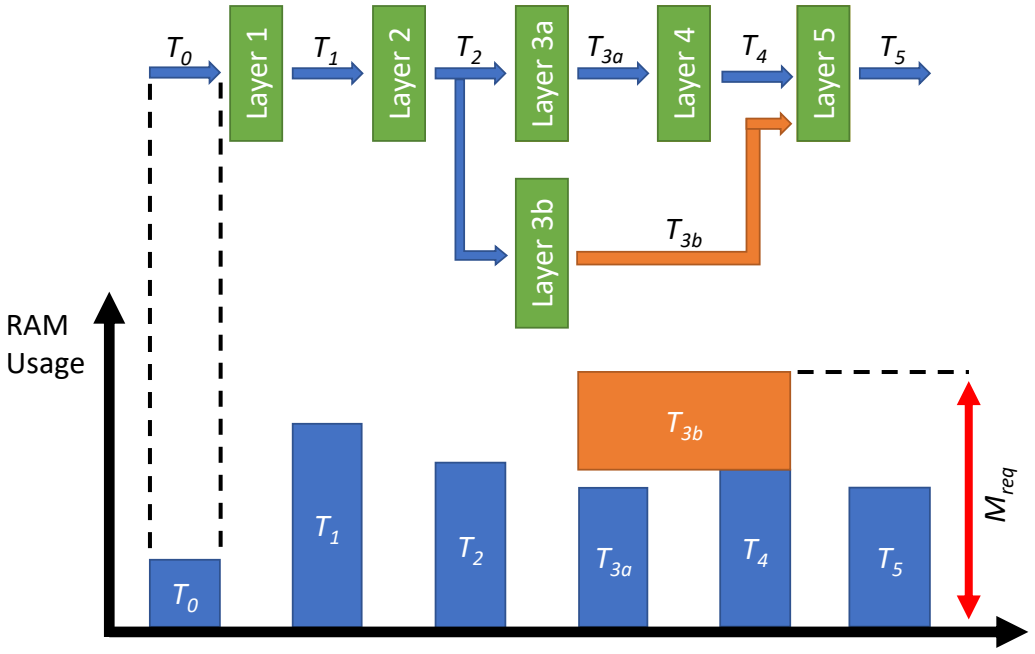


FIGURE 6.2: Graphical representation of the computation of the minimum amount of RAM memory, M_{req}, required to run a DNN. T_x are tensors, whose size depends on network parameters. Skip connections tend to increase memory usage.

would increase (i) the overall PCB size, (ii) complexity, and (iii) costs. When looking for an extremely lightweight, compact and cost effective solutions, all these elements are of interest.

A similar issue concerns RAM usage. This lightweight version of MobileNetv3 requires at least 1.6 MBytes of RAM just to compute the first bottleneck block, which is much higher, for example, than the 320 KBytes available in the STM32F746NG. Again, one

can address this issue by adding an external RAM. In fact, one would face the same drawbacks discussed above.

Overall, it is clear that the deployment of state-of-the-art DNNs for object detection on a general purpose, low cost microcontroller involves major challenges. The next section discusses and analyses the procedures adopted in this research to deploy a lightweight DNN on the STM32F746NG.

6.2.3 Design of a very lightweight DNN for object detection

Memory constraints are a major obstacle when targeting the deployment of a DNN on the STM32F746NG device. The first goal to achieve is to design an architecture with a limited amount of parameters. The lightweight MobileNetv3 DNN represents the starting point for the design of such architecture.

In its original paper, the MobileNetV3 architecture was used as backbone network to target two different applications: object detection and semantic segmentation. In general, a DNN designed to target object detection give as output a number of bounding boxes, with each box associated to a given class. Conversely, a DNN designed to target semantic segmentation gives as outputs bit-wise classification of the pixels in the form of masks. Although the segmentation task might look more complex than object detection, the paper shows that semantic segmentation can be implemented with a DNN that features much less parameters. The two DNNs for detection and segmentation used MobileNetV3-Small as backbone and differed only in the topmost block, as obviously the feature map provided by the backbone should be processed in a different manner according to the target (semantic segmentation or object detection). Eventually, the DNN targeting semantic segmentation featured a total of 0.47 M parameters, while the object detector featured a total of 1.77 M parameters. A possible explanation is that the output of the segmentation head is basically a heatmap indicating for each class the probability density of the presence of that object. Hence, such representation has a high semantic connection to the feature maps provided by the backbone network. Differently, the detection head has to convert the feature maps into bounding box coordinates, passing through the anchor box mechanism described in [89].

For these reasons, in this work, MobileNetV3-Small for semantic segmentation has been used as starting point for the backbone. In Section 6.3 we will explain how the segmentation mask has been exploited for detection. The segmentation DNN has been further customised to meet the constraints set by the target device. Figure 6.3 shows the final architecture. In particular, the following action has been executed:

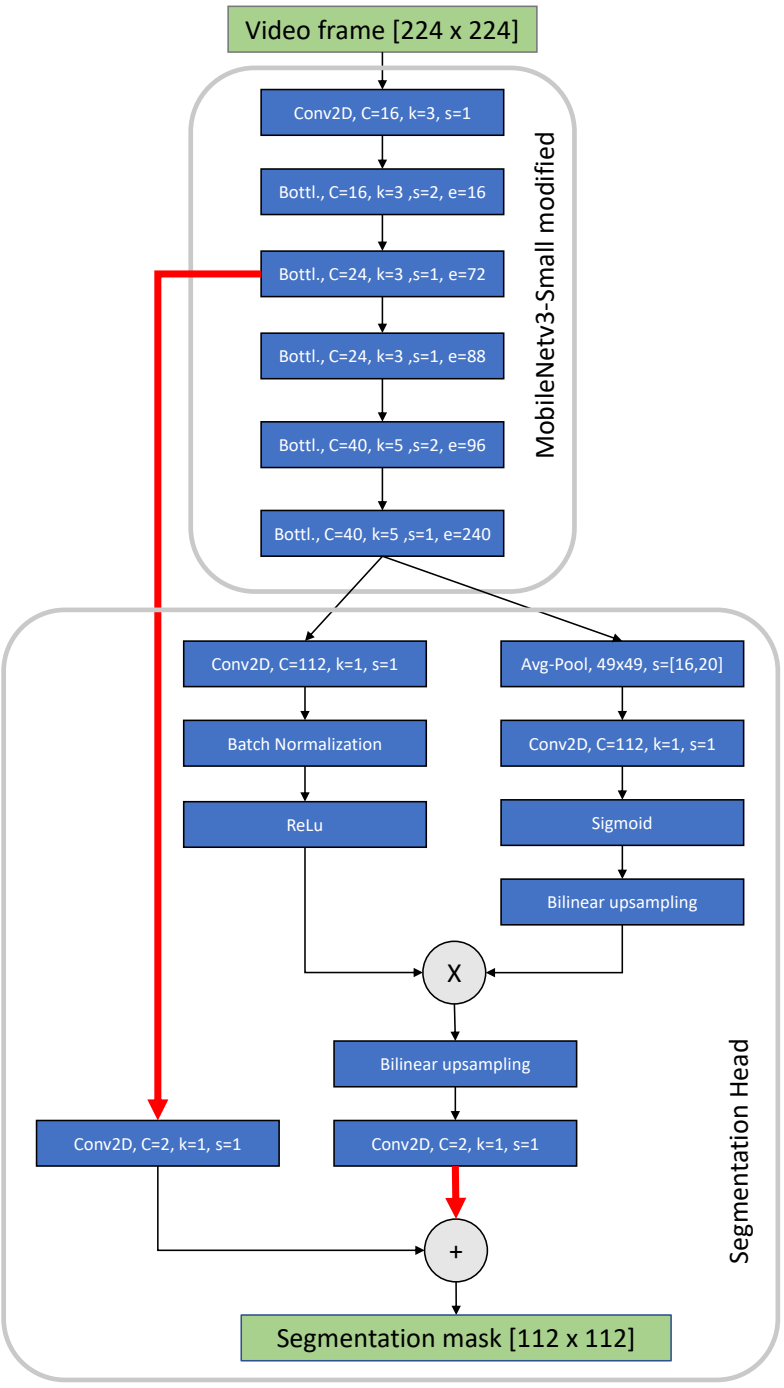


FIGURE 6.3: DNN architecture proposed in this work. The segmentation mask is then postprocessed to get the detections. *Bottl.* stands for Bottleneck block. C is the number of output channels, e the expansion, k the kernel size, s the stride. The red arrows represent memory critical paths.

- the size of the input image has been reduced from 1024 x 2048 pixels to 224 x 224 pixels. A high resolution allows accurate object segmentation, but it also impacts on execution speed and RAM usage. In our case, accurate object *segmentation* is not required, since we target the more generic *detection* of a given object.

- The stride of the first convolution layer and second bottleneck layer have been decreased from 2 to 1. In fact, the stride acts as a compression factor on the image size. Since a 224 x 224 pixels input image was adopted, one should avoid a harsh compression.

- The feature maps feeding the segmentation head have been changed with respect to the architecture provided in the original paper. Again, the goal was to not loose information due to network compression. Hence, in the proposed DNN the output of the 2nd and 5th bottleneck blocks become the inputs to segmentation head. Originally, the 9th bottleneck block and the last 2D convolution fed the segmentation head [12]. This means that the layers from the 5th bottleneck block to the end of the backbone network do not need to be computed anymore, thus improving execution speed and reducing the total number of parameters.

The resulting network featured a total of 191K parameters, which correspond to 764 KByte; such outcome fitted the 1024 KBytes available on the selected STM32F746NG microcontroller. The actual RAM memory required for running the network on the microcontroller was assessed by exploiting the STM32Cube development environment distributed by ST Microelectronics. That quantity was estimated in 8.7 MByte, i.e., an amount of memory significantly larger than the available 320 KBytes. According to the discussion provided in 6.2.2, it is easy to show that the 8.7 MByte are due to the parallelism of the computational paths shown in red in Figure 6.3, which carry memory-heavy tensors. On the other hand, fitting the 320 KBytes of RAM (with our or any other performing DNN for the task) would require an excessive downsizing of network parameters, thus undermining the whole DNN architecture performance. Hence, the only available solution to address a convenient trade-off among costs, size, and performance was to include an external SDRAM in the eventual device. Reducing from 128 to 112 the number of channels in the tensors involved in the two mentioned red paths made us achieve a final footprint of 764 KByte of Program Data and 7.2 MByte of RAM, which eventually require a 8 Mbyte SDRAM.

The final output of the designed DNN is a heatmap H of size 112 x 112 x N_c, where N_c is the number of detection classes. A *softmax* layer has been added at the very end of the network; as a result, all the values of the heatmap H are in the range $[0, 1]$ and, for a given pixel and a given class, correspond to the probability of having that class

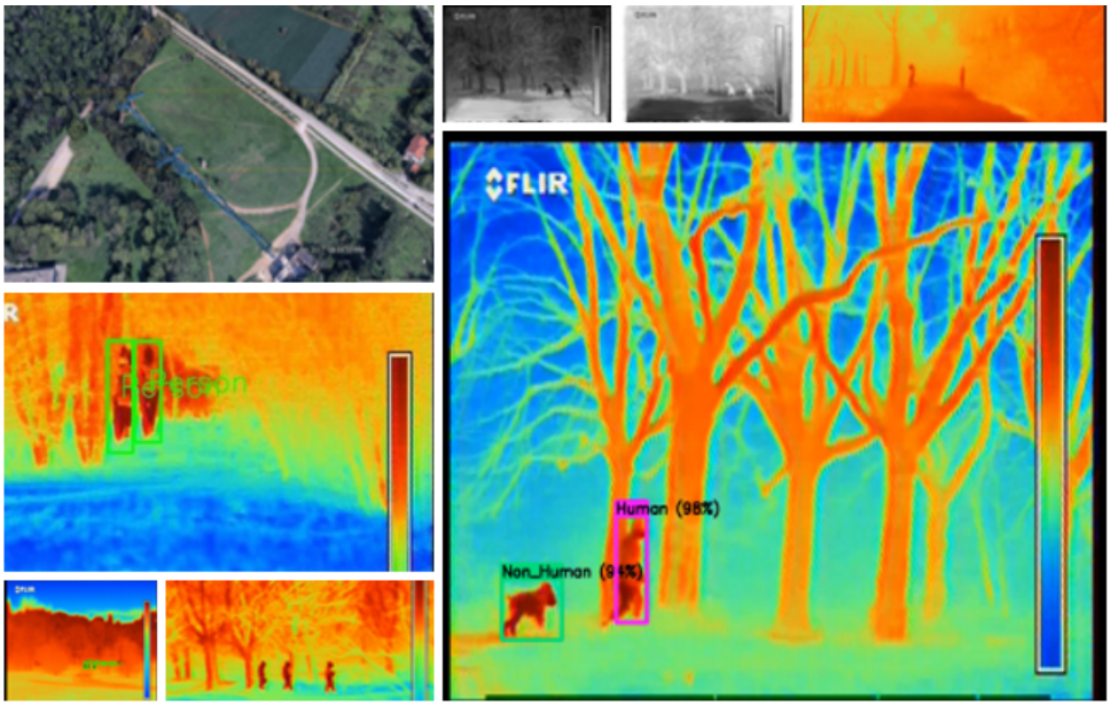


FIGURE 6.4: Preview of the UNIRI-TID dataset for person detection on thermal images.

in the corresponding position of the network input image. The *softmax* layer ensures that, for each pixel, the sum of the probabilities of each class is equal to one, that is $\sum_{k=1..N_c} H_{i,j,k} = 1 \ \ \forall i, j$.

6.3 Use case: person detection in outdoor thermal images

The use case addressed in this work is a semi-autonomous battery-operated video surveillance device. The device is designed to periodically grab a frame from the video captured by the camera to the purpose of triggering an alarm whenever specific events happen. In this research, the event to be detected is the presence of people in protected areas. In the following, Sec. 6.3.1 describes the dataset adopted to implement and test the device on the use case.

6.3.1 The dataset

The device has been tested by using the "Thermal image dataset for person detection (UNIRI-TID)" openly distributed by the University of Rijeka [90]. The dataset, a preview of which is shown in Figure 6.4, is comprised of a set of thermal images that simulate illegal movements of people around the border and in protected areas. The

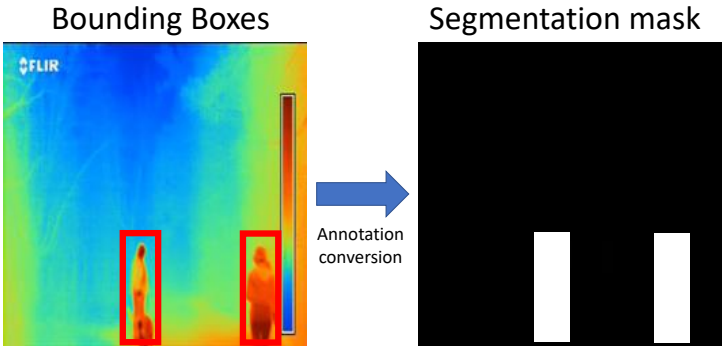


FIGURE 6.5: Example of the annotation conversion we performed on our dataset.

images are taken from the frames of a thermal camera. The videos are recorded in areas around the forest, at night, in different weather conditions – in the clear weather, in the rain, and in the fog, and with people in different body positions (upright, hunched) and movement speeds (regular walking, running) at different distances from the camera. In addition to using standard camera lenses, the authors used also telephoto lenses to test their impact on the quality of thermal images and person detection in different weather conditions and distance from the camera. The dataset comprises a total of 7412 images extracted from video frames captured in the long-wave infrared (LWIR) a segment of the electromagnetic (EM) spectrum. For each image, the position of people in the image is annotated in the form of bounding boxes, provided on separated files.

On the original dataset we performed the following operations:

- **Augmentation**: we flipped all the images horizontally to double the size of the dataset while maintaining realism.
- **Resize**: we resized the images to our DNN input size of 224 x 224 pixels.
- **Annotation conversion**: since our method exploits a segmentation DNN, for training, annotations (groundtruths) must be provided in the form of pixel-wise input image classification instead of bounding boxes. To achieve this conversion, we simply considered all the pixels inside a given bounding box as person-classified pixels. All the remaining pixels were marked as background. An example is shown in Figure 6.5.

One might observe that, due to our label conversion, discrimination between more people whose bounding boxes are adjacent or partially overlapping could be lost, since all the included pixels would result in a contiguous region of "person" classified pixels, with no information on the actual number of people involved. However, this does not represent an issue for our purpose, since in our application we're not interested into counting the

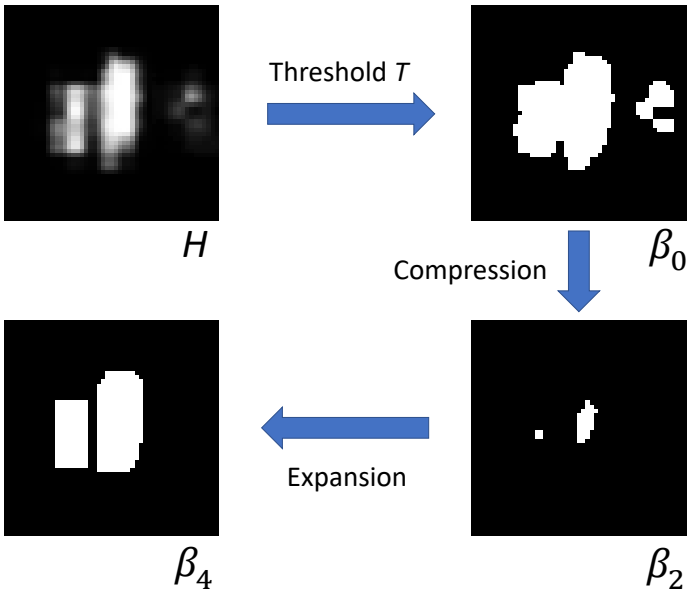


FIGURE 6.6: Postprocessing of DNN output for the person detection use case. False positives, on the right of the image, are filtered out.

number of people, but just on detecting their presence in the camera's field of view.

6.3.2 DNN-based model for person detection

The architecture designed in Sec. 6.2.3 has been further customised to tackle the use case. Accordingly, the final output of the designed DNN was a heatmap H of size 112 x 112 x 2, as the goal is to detect a person over a background.

However, in our application the final output must be a boolean signal which indicates the presence, or not, of the object O in the camera's field of view. This means we need to process the output heatmap H to compute the boolean value. To this end, we perform the steps described hereafter and depicted in Figure 6.6.

First, we considered a threshold T on the 112 x 112 heatmap $H_{\cdot,\cdot,k_o}$, where k_o is the channel corresponding to the target object (person) O. After applying this threshold, we get a bitmap β_0 where 1 corresponds to the detection of the object and 0 to background. Then, to filter out detections (bits set to 1) which are likely false positives, we apply on the bitmap β_0 the following operations, divided into a *compression* stage (steps 1 and 2) and *expansion* stage (steps 2 and 4):

1. We apply on β_0 a 2D convolutional filter using a $n \times m$ pixels all-ones kernel K_1:

$$K_1 = ones(n, m) = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}$$

   This is a rectangular filter of height m and width n designed to highlight, at the output of the convolution, contiguous regions of pixels set to 1. We call the output β_1.

2. We set to 1 all the pixels whose value equals $n \cdot m$. Given the convolution in step 1, this means that only the points that in β_0 were at the center of an all 1 area of *at least* $n \times m$ pixels are set to 1. The other pixels are set to zero. We call the resulting bitmap β_2. In our use case we set $n = 15$ and $m = 7$.

3. Detections (1s in the bitmap) which survived to steps 1 and 2, at this stage extend over a smaller area with respect to their original size. In order to compensate this loss, we compute β_3 by applying on β_2 again a 2D convolutional filter using the same kernel K_1. This step works as an expansion filter.

4. Finally, we set to 1 all the non zero values, thus obtaining the final bitmap β_4.

The algoritm described is efficiently implemented with two convolution operations (corresponding to steps 1 and 3), each with a condition on the output of every convolution step (corresponding to steps 2 and 4). Therefore, the computational and memory impact on the overall DNN execution is negligible.

The final boolean value indicating the presence or not of the specific object in the video frame would simply be *true* only if β_4 contains at least one 1. This step can be efficiently included in the computation of step 4 above, thus not impacting the total execution time.

6.3.3 Deployment on device

The final DNN-based solution for person detection has been tested on the development board 32F746GDISCOVERY, which mounts the STM32F746NG including the necessary circuitry to power it up, connect a debugger and test many of its features. The board also mounts an external high speed SDRAM, MT48LC4M32B2B5-6A from Micron, which provides 16 MBytes of additional memory (only 8 MBytes accessible) that can be directly accessed from the microcontroller. The size of the package of the SDRAM is 13mm x 8mm.

Even if the development board contains many integrated circuits, to run our specific application only the microcontroller and the external memory, including the minimum circuitry for their power supply and communication, are needed. We computed that a dedicated PCB would require an area of about 2 x 2 cm only.

Our target application would also require an external camera. However, our contribution focuses on the detection of objects on a given video frame, which is a task independent from the specific embedded camera used. Therefore, we did not mount a camera, and we used USART communication with our laptop to stream to the board the input image data.

6.4 Experiments

6.4.1 DNN implementation, training and deployment on target device

The DNN has been first implemented by using target independent tools and then has been trained on a workstation. Finally, the trained network has been deployed on the STM32 device by using a target specific toolchain.

- Implementation: we implemented our DNN using the TensorFlow framework for Python. We used TensorFlow 2.7.0 and Python 3.8. In particular, we used the Keras library of TensorFlow, which is particularly handy for custom DNN design. Our implementation started on the base of an existing implementation of MobileNet V3 for segmentation and detection freely available on GitHub [91], which we checked and modified according to our needs and following the steps described in 6.2.3.

- Training: we trained our DNN from scratch on the dataset described in Section 6.3.1, which has been randomly split into a training (60%) set, a validation set (20%) and a test set (20%). We used again the TensorFlow framework. We set the batch size to 32, number of epochs to 30, loss function to categorical crossentropy, and we used the Adam optimizer with a learning rate of 0.001. We trained the network using only the training set. We considered as our final trained model, the one with the highest accuracy on the validation set, after each epoch.

- Deployment: the trained DNN has been deployed using the X-Cube-AI expansion package for the STM32Cube development environment, distributed by ST Microelectronics to ease the deployment of Neural Networks on STM32 products. This tool can import the trained NN and generate a template application code, in C,

which includes the code necessary for the NN inference. We set this tool in order to allocate the DNN weights in the microcontroller internal Program Data Memory, while, as anticipated in Section 6.2.3 we used the external SDRAM as the memory for all the internal DNN partial results, that are mainly the intermediate tensors between the layers. In order to minimize RAM usage, whenever possible X-Cube-AI reuses the allocated RAM, thus resulting in a total memory consumption which is in line with the theoretical considerations discussed in Section 6.2.2, that are at the basis of our DNN design. For further optimization, we set the tool to include in this shared memory mechanism also the input and output buffers of our DNN, that are the very first and very last tensors. We manually added the code for the output filtering described in Section 6.3.2, and the necessary code to configure and initialize the external SDRAM.

According to the SW build report, our final application used a total of 791 KBytes of internal Program Data Memory (77%), 164 KBytes of internal RAM (51%) and 7.207 KBytes of external SDRAM (85%).

To verify the correctness of the DNN execution on the embedded device, we used a validation tool included in X-Cube-AI, which automatically compares the output of the DNN execution on the STM32 with the output obtained running the same DNN in Python, when both the networks are fed with the same set of random input tensors. Results running inference on the test dataset are discussed in Section 6.4.2.

Setting the device to maximum performance (216 MHz clock) performing inference on one single video frame took 33 seconds. This result is acceptable in non time critical surveillance applications, such as the use case at hand, when large areas are in the camera's field of view.

In terms of power consumption, during the inference our microcontroller used about 356 mW, to which we need to add an average power consumption of 1 mW for the camera (considering for example the OV7670 camera and taking one picture every 33 seconds).

6.4.2 Detection performance and comparison

Object detection algorithms are conventionally evaluated by standard metrics such as the Precision-Recall curve, where, for binary classification like in our case, each is computed as:

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN}$$

with TP, FP and FN being, respectively, True Positives, False Positives and False Negatives.

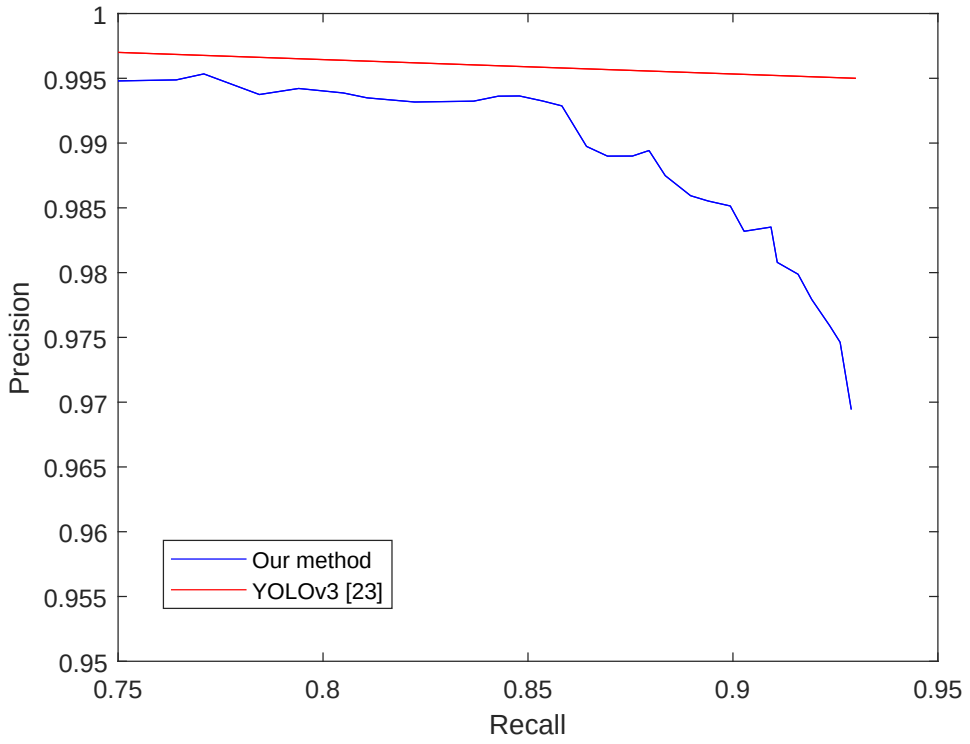


FIGURE 6.7: Results obtained using our method, and qualitative comparison with the results obtained on the same dataset using YOLOv3 in [23].

Since our system is designed to output only a boolean signal which indicates if a specific object is seen in the surveillance camera's field of view, such a metric cannot be directly applied. In fact, these metrics assume detections in the form of bounding boxes, which is out of scope for our work.

However, considering that at the output of our segmentation DNN (just before the conversion step to a single boolean value, explained in Section 6.3.2) we have a detection bitmap, segmentation mask, which preserves the object position information, we could compute the Precision-Recall curve of the detection capabilities of our system by using SOTA segmentation metrics. That is, we considered a detection as: true positive when its intersection over union (IoU) with any of the groundtruth bounding box was greater than a threshold t_{IOU}, that we set to 30%; false positive when its IoU was lower than t_{IOU} with everyone of the groundtruth bounding boxes. All the remaining groundtruth boxes, not covered by the true positive detections, were considered as false negatives.

Since the execution of our DNN on the target device is equivalent, in terms of input/output relation, to its execution on a desktop environment (equivalence verified in Section 6.4.1), we tested its performance on the test set of our dataset, running the DNN on a laptop. Figure 6.7 shows the resulting Precision-Recall curve. Each point of the map is obtained by varying the detection threshold T explained in Section 6.3.2.

As a qualitative comparison, we show in red an approximation of the results achieved

in [92] doing object detection on our same dataset using the YOLOv3 DNN [93] for object detection. The comparison is qualitative since YOLOv3 is also able to count the number of objects in the image. Although YOLOv3 achieves slightly better results (1-2% improved precision) it requires 65M parameters, which is 85 times more than the 766k parameters used by our network.

6.5 Conclusions

In this work we have shown an approach to enable the execution of DNN-based object detection on a cost-effective, lightweight, small, low-power and resource constrained edge device. As a use case, we considered the case of person detection in video-frames for non real-time critical surveillance applications. Results demostrate that relaxing the need of getting precise *bounding boxes* for detected objects and adopting a more *presence* detection approach, it is possible to design a DNN to perform object detection on a STM32 microcontroller, using only an external 8 MByte RAM. Our final system has the following features: requires only (i) 791 KBytes of internal Program Data Memory, (ii) 164 KBytes of internal RAM, (iii) 7.207 KBytes of external SDRAM, (iv) can be implemented on a 2 x 2 cm board, (v) is relatively cheap, (vi) can be used in harsh environments, (vii) requires less than 400 mW doing inference and (viii) is accurate.
We believe that this example and our analysis of the correlation between DNN parameters and their memory and computational footprint might inspire future research focused on adapting SOTA DNN for their execution on edge devices.

Chapter 7

Deploying DNN to microcontroller via Neural Architecture Search: landing pad detection

In this Chapter we present an alternative approach to design DNN-based solutions suitable to be deployed on a resource constrained device. Differently from the previous Chapter, in this case the Deep Neural Network is found through the use of Neural Architecture Search (NAS), that is, a semi-automated way to generate Neural Networks with the aim of minimizing a cost function. Given the target of having a lightweight DNN, we designed the search algorithm in order to take into account not only the network's performance, but also its computational cost. As a reference application, we considered landing pad detection.

7.1 Introduction

Landing pad detection is a crucial problem in the operation of Unmanned Aerial Vehicles (UAV)s [94]. Detecting safe landing areas by using simple passive tags can be critical for many applications[95–97], including smart farm monitoring, rescue operations, home delivery, and autonomous guidance.

The development of a framework for landing pad detection in UAVs poses two major issues. The first issue concerns the localization of small, colored patterns within a possibly complex background. Even if the target has a basic pattern and is very visible,

the operation environment can make the detection goal quite challenging. Secondly, UAVs can typically only host compact, low-power computing platforms, featuring limited computational resources. Cloud-based approaches might not represent a viable solution due to the energy consumption for data transmission and the requirement of a stable connection.

In this application we approach landing-pad detection as a computer vision problem and exploits the effectiveness of deep neural networks (DNNs). The main contribution is a design strategy for the development of lightweight neural architectures, that can fit the operational constraints set by low-power micro-controllers. As a result, the inference (forward) phase of the detector can run on on-board edge devices without energy-demanding hardware accelerators. In this work, the target platform is the STM32F746NG [1] micro-controller, which embeds an ARM Cortex-M7 32-bit RISC core.

In this Chapter we present a novel design strategy for lightweight DNNs, which combines knowledge distillation [98] and Neural Architecture Search (NAS) [99]. A genetic-algorithm procedure spawns a generation of candidate architectures (the Students). The neural training makes each Student adhere to a reference feature representation, provided by the Teacher. Students compete in terms of both recognition accuracy and compliance to hardware constraints; the most promising device-compatible architecture is the seed for the next generation of Students. This cyclic process progressively converges to an effective, hardware-compatible neural system. The competing architectures, generated via NAS, include Bottleneck Residual Blocks[100] as the network core components. The lightweight, yet powerful DNN MobileNetV3 [100] architecture supports the Teacher model, thanks to its known effectiveness in computer vision applications.

The experimental session aimed to assess the design capability of effective architectures, applied to landing-pad detection supported by an STM32F746NG micro-controller. The tests involved a dataset of aerial-like images. Experimental outcomes confirmed that the design strategy compared favourably with state-of-the-art solutions, yet featuring lower computing requirements.

The contribution of the work presented in this Chapter can be summarized as follows:

- an automatic strategy for designing lightweight neural networks for landing pad detection;
- a strategy for real-time inference phase supported by arm cortex, avoiding any hardware accelerator for deep learning;

[1] `https://www.st.com/en/microcontrollers-microprocessors/stm32f746ng.html`

- a tiny architecture supported by a micro-controller yielding a considerably energy-efficient inference phase;
- deployment of the deep neural network inference phase with real-time performance on a commercial microcontroller;

7.2 Related works

7.2.1 Hardware-Aware NAS

Naural Architecture Search has already been introduced in Chapter 3. With Hardware-Aware NAS, HW-NAS, we identify NAS-based approaches which take into account, as an optimization goal, the memory and computational footprint of the target architecture. Three main features characterize HW-NAS strategies [101]. The search space is the first issue, spanning the set of eligible networks, i.e. the set of all admissible architectures. From among the several proposals in the literature, Mobile (M)NAS space is a popular solution [102] designed on the blocks of MobileNet, hence the resulting candidate architectures only include computationally efficient building blocks. Secondly, The actual search algorithm adopted is another crucial aspect, which considers the various hypothesis by selecting architectures from the search space. Finally, the choice of an effective evaluation criteria is of paramount importance, since hardware devices can vary in latency performances even when they support the same number of FLOPS and parameters [99]. Furthermore, software implementations can heavily affect speed, as the same device can score quite differently when it relies on either general-purpose solutions (such as TfLite and TfMicro) or optimized libraries [103, 104]. For that reason, the MCUnet solution [105] integrates both the selection of the architecture and the setup of the computing layer in one optimization procedure, thus attaining excellent performances at the expense of a 'closed' system [106].

Large-scale benchmarks offer a pre-computed evaluation of huge search spaces in the presence of different hardware setups. For example, the research presented in [107] compares 46k architectures deployed in 6 devices. A major issue in the NAS paradigm lies in the cost of architecture search; a popular approach consists in picking candidate networks from a unique 'super' network, which contains all possible architectures spanned by the search space at hand[108]. That strategy sharply reduces training time at the expense of a bias in the search procedure.

As compared with the approach presented in [109], the design strategy presented in this work introduces a teacher network that simplifies the overall procedure, while measurements of FLOPS drive the architecture-selection process.

7.2.2 DNNs for the navigation of UAVs

Bringing intelligence to UAVs is a challenging task due to the limited hardware resources [110]. Popular, cloud-based solutions delegate expensive computations to remote servers [111]; the literature offers cloud-based, object-detecting neural systems for UAVs [112, 113].

When one cannot rely on stable connections or is subject to low-latency inference timings, a platform featuring on-board intelligent behaviour seems the preferred approach. Embedded solutions hosting hardware accelerators, such as the Jetson TX2 board [114, 115] or Raspberry SBCs, have been presented [116], but physical occupation, energy consumption, and timing constraints still remain crucial issues in the overall design process. This is even more important in the presence of multiple smart sensors [117, 118].

Cameras represent the primary input sources to support the autonomous navigation of UAVs [119–122], also including the detection of a safe landing area. The method described in [116, 123] applies conventional computer vision algorithms to extract relevant features and feed Deep Neural Networks (DNN) accordingly. In [124], a Convolutional Neural Network cooperates with a Kalman filter to control landing operations.

In the specific landing-pad detection context [13], the application domain and a simplified version of MobileNetV3 [100] allowed to overcome the (otherwise) computationally demanding requirements of the network. That research confirmed that the adoption of a segmentation network for the detection problem could sharply reduce computational costs.

Several approaches tackled the landing-pad detection either by using a variety of additional sensors or by adopting specific landing pads. The method recently described in [125] relied on a custom landing pad containing specific RGB colors. That paper also reviewed a variety of landing systems based on traditional computer vision techniques, including edge detection [126] and geometry-based template matching [127]. That comparative analysis confirmed that the use of DNNs for landing-pad detection could deserve further investigation.

7.3 Automated Design of efficient DNNs for landing-pad detection

The design approach presented in this work yields a lighweight DNN architecture, which processes the frames captured by the UAV's on board camera and prompts the coordinates of the landing pad in the image plane. The iterative design strategy embeds the Knowledge Distillation paradigm within a Neural Architecture Search process. The idea is to start from a reference "Teacher" model, which is indeed best performing but typically proves cumbersome to implement on low-performance microcontrollers. The method progressively searches for the best "Student" architecture that replicates the Teacher behaviour and, at the same time, satisfies computational constraints.

The search space is a refined version of MNAS. The overall process relies on a straightforward evolutionary algorithm, which combines simplicity and effectiveness. Empirical evidence justifies the adoption of FLOPS as the basic evaluation parameter. Indeed, the target device exhibits a quasi-linear correlation between FLOPS and run-time inference timings, possibly due to the single computing core within the device, which forces a sequential flow of operations.

7.3.1 Knowledge distillation

In knowledge distillation, a 'Student' architecture is optimized to replicate the behaviour of a 'Teacher' model. In the target application, a Teacher's architecture includes a *backbone* to carry out feature extraction, and a *head*.

In the specific case of landing-pad detection, the head component processes those features and prompts the coordinates of the object of interest. The Teacher model actually looks for a simple pattern, and high-level semantic information (usually residing in the head section) is less critical. As a consequence, the intermediate layers in the backbone may already work out the relevant features. The overall strategy, therefore, is to select the Student that best replicates the behaviour of the Teacher's backbone.

To optimize the weights of the Student one measures the discrepancy between the Teacher's and the Student's responses to the same input. The associate cost compares the internal feature representations of the Teacher and of the Student at selected corresponding locations of the networks. The Teacher's backbone relies on a fixed architecture; one denotes, for the $i-th$ input, as $\boldsymbol{\mathcal{F}}_{Ii}^{T}$ and $\boldsymbol{\mathcal{F}}_{Hi}^{T}$ the representations at some intermediate level and at the higher level, respectively. Those values are compared with the corresponding feature values, $\boldsymbol{\mathcal{F}}_{Ii}^{S}$ and $\boldsymbol{\mathcal{F}}_{Hi}^{S}$ in the Student model. Figure 7.1 exemplifies the comparative approach.

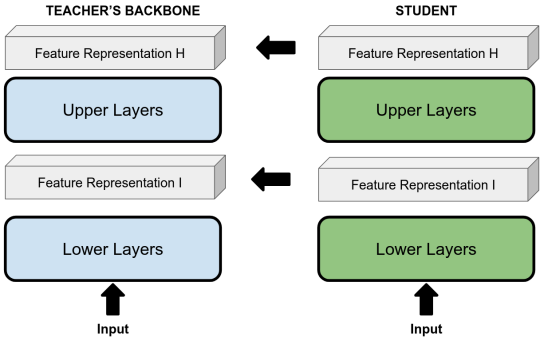


FIGURE 7.1: Block-wise representation of the CNNs with the features maps. On the left is the Teacher's backbone which produces two representations (tensors). On the right is the Student network that aims to provide the same feature sets, using different building blocks.

The following notation helps formalize the cost D:

- $\boldsymbol{\mathcal{X}}_i \in R^{H \times W \times 3}$ is the tensor representation of the i-th input image (having size $H \times W$).
- $\boldsymbol{\mathcal{F}}_{li} \in R^{H_l \times W_l \times C_l}$ is the tensor that holds the output feature representations at the l-th layer of the neural network, where H_l, W_l, C_l are the height, width and number of channels of the tensor, respectively.
- V= $\{(\boldsymbol{\mathcal{X}}_i, y_i); i = 1, .., Z\}$ is the validation set, i.e., a collection of images and labels that are never involved in the training of both the Teacher and the Students.

The overall distillation loss, $\mathcal{L}_D$, is the expected discrepancy value, measured at the selected positions in the networks:

$$\mathcal{L}_D = A_V(|\boldsymbol{\mathcal{F}}^T_{Ii} - \boldsymbol{\mathcal{F}}^S_{Ii}| + |\boldsymbol{\mathcal{F}}^T_{Hi} - \boldsymbol{\mathcal{F}}^S_{Hi}|) \tag{7.1}$$

where $A_V()$ denotes the average over a validation set. Figure 7.2 outlines the knowledge-distillation approach. The distillation loss function (7.1) makes the Student's intermediate representation match the Teacher model's backbone.

7.3.2 Neural architecture search

The NAS process gathers a set of lightweight, candidate architectures that should replace the Teacher's backbone. For its definition, one should consider the search space, the search strategy, and the selection criteria.

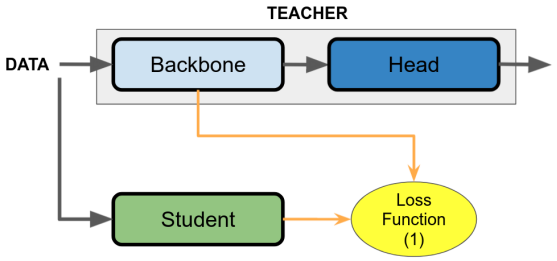


FIGURE 7.2: Proposed teacher/student learning schemata

The search space is in fact vast, as one (mild) constraint simply admits any architecture that includes a pair of main blocks (i.e., lower and upper layers, respectively). To shrink that space, the design strategy only considers a sequential combination of parametric building blocks, made by single-branch neural networks. This conventional setup [99], however, cannot by itself support an extensive search, since typical building blocks involve up to 6 parameters.

The search strategy selects the best Student architecture, and embeds a basic evolutionary algorithm (Figure 7.3). Given a 'parent' architecture, the process yields a set of N 'child' networks by applying random mutations to the parent. Admissible mutations include changes in the parameters of the blocks or in the number of blocks itself. Child architectures all undergo a distillation-based training procedure. The selection criteria highlights the best Student from among the N children. The selected candidate now plays as the new parent architecture, and the whole selection strategy iterates until a stop condition is fulfilled (that condition will be detailed in the following).

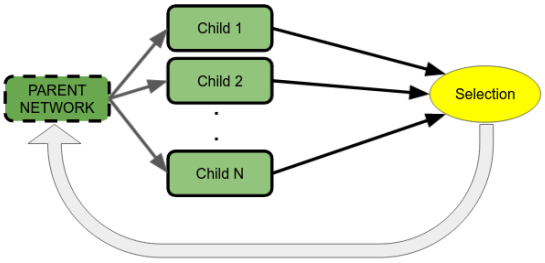


FIGURE 7.3: Block scheme of the evolutionary algorithm

The adoption of a straightforward evolutionary algorithm stems from several reasons. First, the training of a child network is fast as lightweight models are involved. Secondly, an evolutionary algorithm supports any kind of mutation in the network architecture and is not subject to the optimization procedure. In addition, random mutations allow a wider, unbiased exploration of the search space as compared with other NAS approaches [128]. Fourth, the evolutionary algorithm can admit a non-differentiable cost criterion for the child-comparison task. This is a major advantage when considering that formalizing hardware constraints explicitly may prove quite difficult: for instance, the

relationship between the network architecture and the corresponding RAM occupation might also depend on the optimization tool used to deploy the overall architecture itself. Finally, an independent selection process can indirectly help the children-training procedure. Common practice suggests that child networks are trained for a limited number of epochs, mostly to speed up the search procedure; widening the search area in the space of architectures can actually integrate the basic weight-adjustment process.

The selection criteria is the last issue to consider to define the overall NAS process. It relies on a cost function that integrates the Teacher-Student representation mismatch and the associate computational cost, measured in Floating Point Operations per Second (FLOPS). The resulting overall cost function, S, for the $n-th$ model is therefore written as:

$$S_n = \mathcal{L}_{Dn}, + \beta C_n \tag{7.2}$$

where $\mathcal{L}_{Dn}$ measures the discrepancy between the Teacher and the n-th candidate Student as per (7.1), C_n is the computational cost in FLOPS associated with that candidate, and β is a parameter that rules the relative weights of the two terms.

Measuring the computational cost in FLOPS might appear, in general, insufficient to characterize the hardware awareness of a Student model. In the present context, however, it is a significant indicator when considering the architectures of basic microcontrollers, which support limited or null parallel computation.

7.3.3 Integrated Neural Architecture Search with Knowledge Distillation

A cyclic process combines the paradigms discussed above, and supports the progressive performance-driven neural architecture search; the overall approach is depicted in figure 7.4 and evolves as follows:

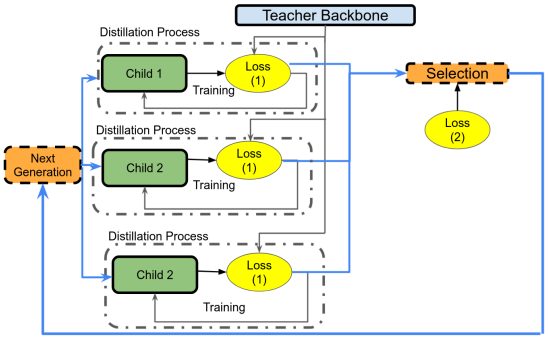


FIGURE 7.4: The integrated design strategy

1. The next generation block generates a set of candidate children networks introducing mutation into a parent architecture;

2. In compliance with the Knowledge-Distillation paradigm, the set of candidate Student architectures are trained (as per function 7.1) to approximate the Teacher's behaviour; in figure 7.4 the dashed boxes identifies the training procedures of the children;

3. The trained children are compared, by also taking onto account the computational complexity of each candidate as per expression 7.2 by the selection block;

4. The resulting 'best' child architecture is used to spawn a next generation of candidates as per step 1, which undergo the same selection process.

For simplicity, the stopping condition in the above cyclic procedure just relies a preset number of iterations. The procedure yields a lightweight backbone that can work out similar features with respect to those extracted by a large network. The backbone is finally retrained in an end-to-end fashion together with a head on the target landing-pad detection problem.

7.4 Deployment of the Landing Pad Detector

7.4.1 Edge devices

The reference platform for the deployment of the landing pad detector is the STM32F746NG microcontroller, which features an ARM® Cortex-M7 32-bit RISC core. The microcontroller unit (MCU) operates at up to 216 MHz and includes a single-precision floating point unit that supports all ARM® single-precision data-processing instructions. It holds 1 Mbyte of Flash memory and 320 Kbytes of SRAM. The device also supports external memory access, which can be efficiently used via a Flexible Memory Controller (FMC) and either standard or advanced communication channels. The device extends on an area of 13.15mm x 13.15mm. To cope with memory requirements, an 8 Mbyte SDRAM was added.

The ARM® Cortex-M7 is a member of the energy-efficient Cortex-M processor family. Cortex-M targets low-power applications featuring reduced clock frequencies (up to a few hundreds of MHz) and supporting the indexing of small-size memories (up to a few MB). Hence, the deployment of DNN-based computer-vision solutions on such processors represented a challenge. ST's software tool to optimize artificial neural networks on

STM32 (STM32 X-Cube-AI) only supports the deployment of very tiny architectures; this also holds true in the case of TensorFlow Lite for microcontrollers.

In the experimental session (Sec. 7.5) the landing pad detector was also deployed on a processor of the ARM® Cortex-A family, to provide a baseline for comparison. The processors of the A (Application level) family feature high clock frequencies and support the indexing of large memories in the range of GB. Those processors are typically hosted in microcomputers and smartphones. The specific device used for the experiments was the 1.5 GHz quad-core Arm® Cortex®-A72 CPU that supports the Raspberry Pi 4.

7.4.2 Hardware-aware landing pad detector

To cope with the tight constraints imposed by ARM® Cortex-M7 MCUs the Teacher model was inherited from [13]. In [13], the detection task was suitably approached as a pixel-wise problem. The network's output was a mask that marked the pixels belonging to the landing pad. That approach allowed to rely on the lightweight LR-ASPP head for image segmentation, and to avoid the popular, yet computationally hungry single-shot detectors (SSDs). Eventually, applying simple heuristics on the output mask provided the coordinates of the landing pad [13].

The ablation study presented in [13] showed that, given the small set of bottleneck layers, the squeeze and excite layers could be removed without significant loss in accuracy. The landing pad detector [13] was implemented with a network with 60,612 parameters that required 1.289 GFlops to process an input image of size 320×320 pixels. In the approach presented here, the candidate Student models only involved Bottleneck Residual Blocks, which admitted different settings for the number of filters, the kernel size, the expansion value, the stride, and the non-linear component. The loss function 7.2 drove the distillation process.

The severe constraints set by the STM32F746NG microcontroller imposed to customize also the segmentation head of the final detector, which in principle should inherit the LR-ASSP architecture. This simple segmentation head involved a few convolutional layers and upper sampling layers. The overall architecture is presented in Fig. 7.5: green blocks refer to the input tensors coming from the backbone, while the red block refers to the output mask. Input images have size 320x320. From a computational viewpoint, the scheme highlights three main bottlenecks:

- The number of filters in the lowest convolutional layers (marked in red); this quantity not only sets the number of parameters and operations for the two layers, but also impacts on the size of tensors in the following layer.

- The size of the output mask (marked in yellow), which in the original implementation halves the size of the input image. In principle, one can use a smaller output mask to cut the number of operations.
- The size of the input image, which heavily affects the number of floating point operations.

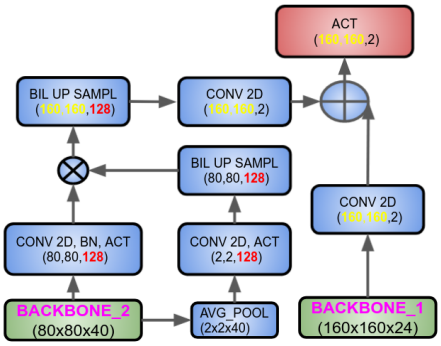


FIGURE 7.5: Block scheme of the teacher segmentation head [12]

In the proposed implementation, input images held 160×160 pixels; the number of filters was set to 8 (instead of 128), as the landing pad had a simple shape. The size of the output mask was four times smaller than the size of the input image.

7.5 Experiments

The experimental dataset included 21 videos of landing pads and featured a total of 29,415 frames. The videos were grabbed at two different heights, approximately 4 and 8 meters. The dataset covered three possible landing pads, two in color (orange and blue), and one in grayscale, all with a black 'H' mark printed on a standard A4 sheet. A collection of 9,200 frames (drawn from 9 of those videos) formed the training sets. The images from the remaining 12 videos composed the test set.

7.5.1 Distillation

The distillation procedure was implemented by using Keras. The Teacher network was trained on the dataset for 20 epochs. The candidate children networks were trained for 4 epochs, following the early stopping paradigm to speed up NAS procedure [102]; batch size was set to 4, and the learning rate was 1e-4. Every pool of child networks included 6 candidates; each candidate stemmed from two mutations of the parent network. The

Input size	Kernel size	Filters	Expsize	Act.	stride
320x320x3	3x3	24	72	Relu	2
160x160x24	5x5	40	96	h-swish	2

TABLE 7.1: Architecture summary of the first parent architecture in the NAS procedure.

distillation procedure was performed considering images having size 320×320; the original segmentation head of the teacher networks that relied on a 160×160 segmentation mask. The number of filters was 128 as per the original implementation.

The first parent network featured a tiny architecture holding a pair of Bottleneck Residual Blocks. Table 7.1 gives the settings of the two layers; the columns give, for each layer, the size of the input tensor, the kernel size, the number of filters, the expansion factor (Expsize), the activation function, and the stride. Using a tiny architecture at the beginning of the procedure is a popular approach called *hot start* [129]. In practice, the search process is expected to progressively increase the complexity of the architecture while targeting a higher accuracy.

The parameter β trades off the quantities D_n and C_n in the cost function 7.2. Those terms typically differ significantly in their orders of magnitude: D_n varies in the order of units, whereas C_n may range from millions to billions. In the experiments, β was set using the following equation:

$$\beta = \frac{D_0}{C_0}\beta_E \tag{7.3}$$

where β_E sets the actual trade-off between the two terms in eq. 7.2, D_0 and C_0 are two normalization terms: when $\beta = \frac{D_0}{C_0}$ the contributions of the two terms are equal. The values D_0 and C_0 are the respective cost terms in 7.2 worked out for the Teacher architecture computed after 4 epochs of training, i.e. the training epochs used for Students architectures.

Two distillation experiments were conducted for as many different settings of the parameter β_E:

- Balanced configuration ($\beta_E = 1$). In this experiment, the stopping criterion for the distillation process was set to 50 iterations. Therefore, a total of 300 children architectures were evaluated. Eventually, the selected Student network included three Bottleneck Residual Blocks; the number of parameters of the resulting network composed of backbone and segmentation head (LR-ASSP) was 39,012, while an inference required 648 MFlops. Table 7.2 gives all the details about the selected architecture (denoted as *BAL_BB*) by adopting the same format of Table 7.1. In

Input size	Kernel size	Filters	Expsize	Act.	stride
320×320x3	2x2	24	72	Relu	2
160×160x24	5x5	46	96	Relu	2
80×80x46	5x5	40	128	Relu	1

TABLE 7.2: Balanced architecture distilled with the proposed method.

Input size	Kernel size	Filters	Expsize	Act.	stride
320×320x3	1x1	24	10	Relu	2
160×160X24	5x5	40	40	Relu	2
80×80x40	6x6	40	56	Relu	1

TABLE 7.3: Small architecture distilled with the proposed method

practice, starting from a parent network with two blocks, a block was added to find a trade-off between the discrepancy D and the computation cost C (since $\beta_E = 1$).

- Small architecture ($\beta_E = 10$). This experiment privileged the minimization of the computation component, C. The number of iterations in the distillation process was set to 100 because the child networks were smaller in size and the training phases were faster. A total of 600 architectures were evaluated. Again, the selected Student network involved three Bottleneck Residual Blocks; in this case, the number of parameters of the resulting network composed of backbone and segmentation head (LR-ASSP) amounted to 23,232 and one inference step required 283 MFlops. Table 7.3 gives the architecture details and adopts the same format of Table 7.1. Notably, the first block uses a kernel of size 1×1, while in general low-level kernels have larger sizes in standard architectures [130]. In the following, this architecture will be referenced to as *SMALL_BB*.

7.5.2 Generalization performance of the landing pad detector

The experiment assessed the generalization performance of the architectures embedding the two Student networks discussed above. In total, four architectures for landing pad detection were tested:

1. *BALANCED*: this architecture stacked the original segmentation head (LR-ASSP, as per Fig. 7.5) on the BAL_BB backbone.
2. *SMALL*: this architecture stacked the original segmentation head (LR-ASSP) on the SMALL_BB backbone.
3. *STM32*: this architecture stacked the customized version of the segmentation head described in Sec. 7.4 on the SMALL_BB backbone.

4. *STM32_TINY*: this architecture stacked the customized version of the segmentation head described in Sec. 7.4 on a customized version of the SMALL_BB backbone, where the expansion factor has been divided by 2. In this way, the number of floating point operations has been reduced without changing the size of the tensors propagated through the architecture.

The input size for the two latter architectures was set to 160×160 pixels instead of 320×320 pixels. The images of size 160×160 were obtained by zooming the original images not to lose definition of the target landing pad in the image. All the networks were trained for 20 epochs; performance analysis involved a test set of 20,215 images never used in the training process.

The four networks were compared with three recent baselines suitable for solving the same task. The first baseline was the architecture adopted as the Teacher model [13]. The second baseline was the network proposed in [116], which supports recognition and classification of the landing pad on an edge device embedded in the drone. Finally, the third baseline was the SSD-lite MobileNetV3 architecture, i.e., a general-purpose object detector that suits an edge paradigm [112, 113]. The Teacher network was trained using the aforementioned settings. The detection method proposed in [116] was replicated following the setup of the original paper. The SSD-lite MobileNetV3 architecture pre-trained on coco-dataset was fine-tuned for 20 epochs with a learning rate 10^-3 and ADAM optimizer. Hyper-parameters were set using a subset of the training data as a validation set.

The landing pad detectors supported by the four architectures listed above extracted a probability mask, i.e., a mask containing for each pixel the probability of being a landing pad. The mask was converted into a box using a blob search algorithm. In general, a mask can generate multiple boxes. Here, only the largest box was considered. This setup corresponded to a worst-case analysis.

Figure 7.6 gives the results of the experiments involving the seven different implementations of the landing pad detector. On the x-axis the plot gives the confidence threshold, i.e., the minimum probability level for a pixel to be classified as a landing pad. The y-axis gives the percentage of true positives (TPs), i.e., images where the Intersection over Union (IoU) between prediction and ground truth was higher than 0. Threshold 0 was set because landing pads covers a small portion of the image therefore even small values of IoU identify valid detections. In this plot, the two baseline methods that did not exploit a pixel-wise classification are obviously characterized only in terms of TP percentage.

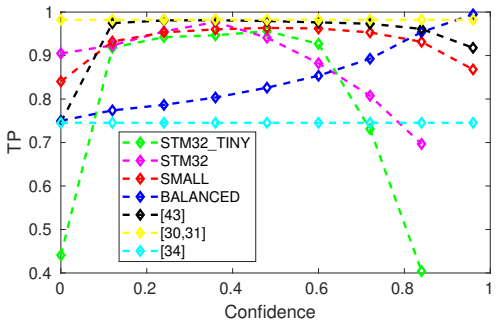


FIGURE 7.6: Generalization performance of the architectures under analysis: NSE [13], NAS-1 tiny architecture, NAS-2 balanced architecture

The results confirmed that a general purpose object detector with backbones for general purpose computer vision represents a valuable option to obtain very high accuracy. Similarly, traditional computer vision pipelines, even being more convenient in term of computing cost, features lower generalization capabilities with respect to DNNs. The teacher network [13], for intermediate values of confidence, confirm to be a suitable option to extract effective features. The four distilled network shows two trends. The BALANCED network improves monotonically its performances when the confidence threshold value grows. With high levels of confidence, the network becomes the most accurate among all the predictions. The results suggest that using a smaller set of parameters acted as a regularizer. The three versions based on the small backbone exhibit a similar trend. Interestingly, the difference with respect to the teacher becomes very high for low or high confidence levels, but for intermediate values, the drop in accuracy remains low.

Figure 7.7 shows the standard precision-recall plot obtained by varying the confidence levels. Here, an image was considered a false negative (FN) when the detector did not identify any landing pad and was considered a false positive (FP) when IoU was zero, i.e. the network identified the landing pad in a wrong position. In this plot, the x-axis gives the recall while the y-axis gives the precision. The chart focuses on a specific portion of the plot, i.e., precision and recall greater than 0.95. The color scheme is the same as figure 7.6.

The results confirm that all the deep learning-based solutions yield a high level of precision and recall for at least one confidence value confirming the suitability of the proposed methodology to detect landing pads from medium distances.

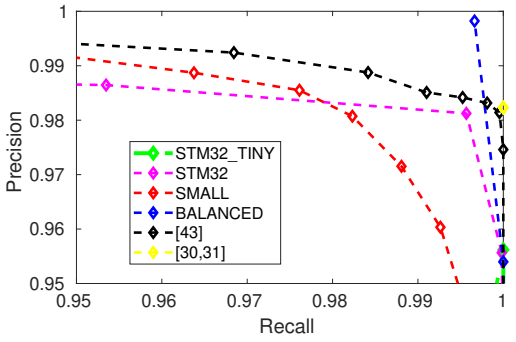


FIGURE 7.7: Generalization performance: precision-recall analysis

7.5.3 Computational performance

The computational performances of the four proposed implementations of a landing pad detector have been evaluated by using as reference the Cortex A and Cortex M platforms, as anticipated in Sec. 7.4.

Two commercial tools supported the deployment of the architectures on the two platforms. The deployment on RaspberryPi 4 (featuring a Cortex A core) was obtained using the TFLite suit, which actually admits only the 32-bit floating-point data type for run-time operations. In fact, quantizations with a 16-bit floating point or 8-bit integer only impacts memory storage.

The deployment on the STM32 microcontroller (featuring a Cortex M core) was performed in two steps. First, the network was converted via TFLite; then, the STM32 X-Cube-AI suit was exploited to optimize the model. Eventually, the memory indexing was tuned to use the external memory, when necessary, to host the tensors during the propagation along the layers of the network. Data representation was set to 32-bit because STM32 X-Cube-AI supports 8-bit representation only for fully connected layers, but the architectures tested don't use these operators. Eventually, one can consider the performance measured as a worst-case analysis considering that quantization can improve performance in terms of latency.

Table 7.4 summarizes the outcome of the experiments. The first two columns (in red) give, respectively, the network name and the input size (in pixel). The third and fourth columns (in yellow) show the computational features of the network that are independent of the implementation, i.e., respectively, the FLOPS and the number of parameters. Columns from fifth to seventh (in blue) refer to the implementation on the STM32 microcontroller. These columns give, respectively, the latency in seconds (per frame), the amount of flash memory required to store the parameters, and the peak of RAM usage. Finally, the last column (in green) refers to the implementation on RaspberryPi.

Model Name	Input Size	FLOPS (M)	PAR.S (K)	STM32 Lat. (s)	Flash (KiB)	Ram (MiB)	RASP Lat. (ms)
BALANCED	320	564	33	NaN	121,57	29,54	305,36
	160	141	33	8,4	121,57	7,4	106,5
SMALL	320	283	23	NaN	83,81	17,19	189,2
	160	70	23	4,2	83,81	4,3	58,4
STM32	320	203	12	11,2	44,44	7,52	135,3
	160	50	12	2,7	44,44	1,88	34,2
STM32 TINY	320	110	6	6,3	22,12	5,57	108,2
	160	27	6	1,5	22,12	1,39	32,5
Teacher [13]	320	1289	60	NaN	216,27	21,3	453,1
	160	322	60	17,5	216,27	5,22	164,9
SSD [112, 113]	320	3758	2760	-	-	-	-

TABLE 7.4: Hardware measures

This column shows the latency in milliseconds (per frame). Since the available memory in the RaspberryPi platform was definitely abundant with respect to the networks involved in the experiment, the table does not provide any information about memory usage.

The table compares the four proposed implementations (BALANCED, SMALL, STM32, STM32_TINY) with the model proposed in [13]. In addition, as a reference, the last row gives the FLOPS and the number of parameters of the SSD architecture for object detection. SSD was not deployed on the devices; here, the table refers to the original implementation available in Tensorflow v1 object detection API model zoo [2]. Eventually the deployment would have led to biased results due to the different implementation. In addition, one can easily see that both the number of flops and parameters are slightly larger than the other network involved in the experiments therefore hardware requirements are likely to be significantly larger.

The trend shown by the yellow indicators remarks that the proposed architectures can save parameters and flops operations. In particular, the STM32TINY network is the smallest one among the proposed ones uses one-tenth of the parameters with respect to the teacher model and around 0.08 of the flops for both the input sizes. As expected raspberry can support real-time landing pad detection using the proposed architectures. Even the balanced model with the largest size achieves more than 3 FPS which is a very interesting result when considering that these devices don't feature dedicated accelerators. The indicators about STM32 report the hardest benchmark. As for the number of FLOPS, the peak RAM usage is a direct function of the input size because the size of this memory is roughly set by the size of the largest tensor propagated along the architecture. The most important indicator is the latency which is very related to the

[2] `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md`

ram occupation. The lowest latency, as obvious, has been achieved by STM32TINY which supports the inference phase in 1.5 seconds. This latency on a commercial microcontroller is a major result for a computer vision task. This observation is further strengthen by the recently proposed [106] that uses the NAS procedure directly aiming toward MCU real-time inference. As shown in the software release, a tiny object detector runs on cortex A, but no one is supported by serie M.

The pipeline proposed in [116] envision a deep CNN to perform the final classification of the landing pad. This model was not included in the table because refers to a different pipeline. MobileNetV2 was one of the architecture tested in [116] for the classification stage. As notorious, classification architecture are significantly expensive in terms of hardware resources. In fact, the lone MobileNetV2 with input size 100x100 uses 3538 K parameters and 156 M floating point operations, that is far away from the performance of the tested models.

All the frame-rates measured on STM32 are lower than 1 frame per second failing the constraints of real-time control but can be sufficient to implement a very low power core that awaken a powerful unit that executes predictions in real time. For example, one can think to a system equipped with two computing modules. A low power core equipped with the STM32 micro-controller monitor the ground. When a landing pad is retrieved the UAVs switch the inference operation to a middle/high performance unit like the raspberry 4 to perform high speed operations only for the amount of time required.

7.6 Conclusions

The work presented a design strategy for landing pad detection on ARM cortex based microprocessors. The strategy uses an automatic procedure to extract the neural network architecture that balances, in the best way, hardware requirements and generalization performance. The results confirmed the suitability of the proposed approach leading to real time deployment on STM32 microcontrollers.

Chapter 8

Patent: Lightweight path learning-based algorithm for prediction of veihicle driving routes

8.1 Introduction

This chapter is dedicated to a final example of application-aware optimized AI, which differentiates from the others since it does not make use of DNNs, but relies solely on a hand-crafted algorithm. Also in this case, we target resource constrained embedded devices, microcontrollers in particular. The present work has been proposed as a patent, and it has already received a positive research report, so it will be soon officially published. The activity has been carried out in cooperation with IVECO S.p.A., an italian company specialized in the production of commercial, military, industrial vehicles and buses.

The context of this work is the electrified mobility. Although their constantly increasing popularity in the automotive market, electric vehicles still have to face two important challenges: the relatively low availability of recharging points and the high amount of time needed to recharge them, which is still orders of magnitude higher than the time required to refill a thermal vehicle with fossil fuel. These challenges translates into the commonly called "range anxiety", which is the fear of the driver to run out of battery while driving. For this reasons, modern electric cars are starting to be equipped with smart systems designed to prevent this fear by providing to the driver informations like

the estimated residual range, the availability of nearby charging spots, and alerts in case of risk of running out of battery. These systems are normally embedded into the vehicle infotainment system and can make use of informations about the status of the vehicle, such as the battery residual energy, the vehicle speed and position. At the same time, car manufacturers have the need to design intelligent control systems aimed at minimizing the vehicle energy consumption. Among these we have logics for energy recuperation from braking and smart thermal management of the cabin and vehicle electronics. In the case of hybrid vehicles, the vehicle control system is also responsible for managing the power split between the thermal engine and the electric motor, as well as strategies to turn one of them fully off. Obviously, the more the data the control system can use, the better the achievable optimization result. As one can easily understand, being able to predict the future power needs while driving would dramatically the energy minimization strategies. For example, knowing that the residual energy needed to reach the target destination is lower than the available battery energy could lead the control system to automatically shut off auxiliary power demanding systems, such as air conditioning. Such predictions may come from the navigation system in case the user would set the destination on it. However, navigations systems are normally not used on habitual routes.

In this work we propose a method to predict the future energy consumption of the vehicle up to the next charging point, by exploiting hystorical data about habitual driving routes. In the context of this thesis, the proposed method represents an alternative form of Artificial Intelligence which relies on manual algorithmic design instead of machine learning-based solutions. As in previous works, our focus is the optimization of the method in terms of effectiveness versus computational and memory requirements. Our proposal, in fact, has the peculiarity of being particularly lightweight with respect to pattern (in this case driving route pattern) recognition solutions, like Markov-chain based ones. The algorithm can be easily run on a very resource constrained device, like a microcontroller on a vehicle's electronic control unit, and is designed with parameters allowing to adapt it to the amount of available resources. The output of our algorithm is intended to be used by an energy optimization strategy.

8.2 Algorithm

The proposed algorithm works by building a *graph* of connected *nodes*, representing the habitual *routes* that the vehicle does, and storing for each route the energy it required. A *route* starts when the vehicle leaves a recharging point, after charging, and ends when the vehicle connects to a charging point (which can also be the starting one). At any

point in time, while driving, the graph can be explored to get if the vehicle is driving on a habitual route and, in that case, to get an estimate of the residual energy needed to complete that route, based on past observations.

The proposed algorithm is supposed to run on an embedded device on the vehicle, and requires, at each iteration, only three informations:

1. Vehicle position
2. Vehicle status (recharging or not)
3. Residual battery energy

Therefore, no street map is needed. The algorithm is flexible to whatever place it is used in. The low amount of data required is one of the main strength points of the solution.

8.2.1 Data structures

The proposed method is designed to use the least possible amount of memory. To achieve this goal, it makes use of a graph of connected nodes to store all its data. Each node represents a point on a route, and has the following properties:

1. *Position*: possibly taken from the vehicle's GPS, stores the geographical position (2D coordinates) of the point on the route.
2. *Age*: used in case of lack of memory, to remove non-habitual routes from the memory.
3. *TotalConsumption*: is the energy consumption needed to reach that node from its eldest parent (which represents the start of the driving route from the last recharge). The total consumption is stored only in the *TerminatorNode*, at the end of a route.

Each node is connected with zero or one parent node and with zero or many children nodes. By plotting the nodes according to their position and connecting them with arrows from the parent to the children, like shown in Figure 8.1, we get an intuitive view of the routes stored in our graph.

In addition to the nodes, four arrays of nodes are used:

1. *StartNodes*: stores pointers to the nodes that represent the starting point of a route, that is when the vehicle leaves a recharging point.

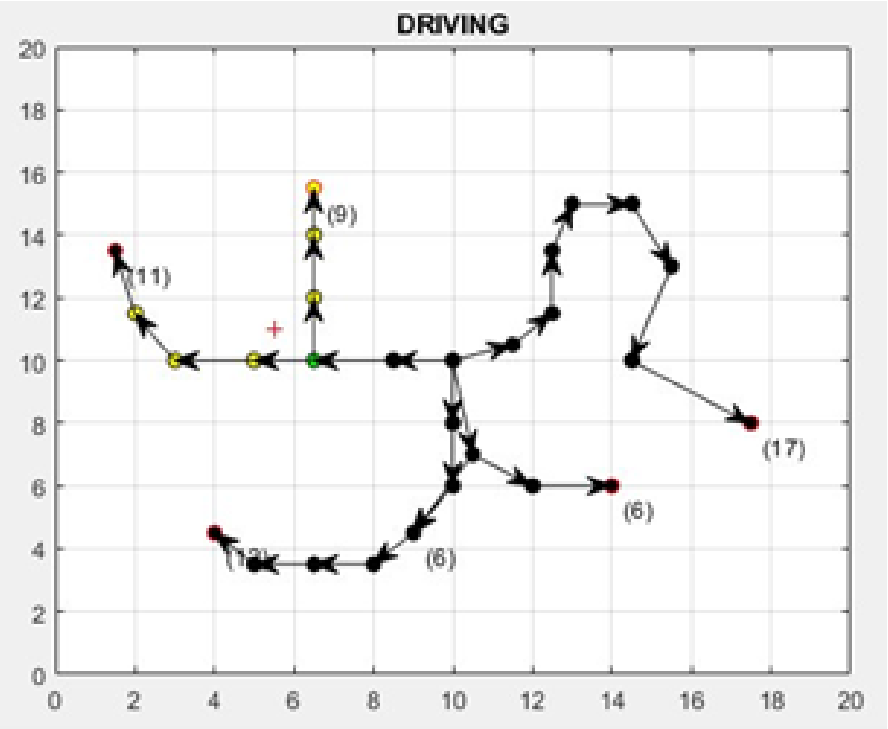


FIGURE 8.1: Graphical representation of the graph of nodes storing habitual routes. Black dots are the nodes. Arrows connect parent nodes with their children. The red cross is the current position. Green dots are the *CurrentNodes*; yellow dots are *TestNodes*; dots with a red countour are *TerminatorNodes*. Numbers in parenthesys represent the *TotalConsumption* recorded in that *TerminatorNode*.

2. *CurrentNodes*: stores pointers to the nodes representing the current vehicle position. They can be nodes already existing in the graph (from previous routes), thus representing the current position of the vehicle along an already existing route; or newly added points, when the vehicle is outside an already stored route.

3. *DevelidatedNodes*: stores pointers to the nodes which have just been removed from the *CurrentNodes* since the vehicle has moved far from their position.

4. *TestNodes*: stores pointers to nodes belonging to routes already present in the graph (from hystorical data) and which are candidates to become *CurrentNodes* in case their position is close to the current vehicle position.

5. *TerminatorNodes*: stores pointers to the nodes which represent the termination point of a route, that is the point where the vehicle gets connected to a recharging point.

The next section will help in the comprehension of these arrays.

8.2.2 Graph update

Figure 8.2 supports the explanation of the algorithm which updates the graph.

The three arrays of nodes are initialized empty.

Provided that the vehicle position, status, and residual battery energy are available, the graph update algorithm can be executed on a time-base or according to an event. In this case, we considered the event of "incoming GPS position signal".

From a high level view, the algorithm is divided into three steps: update *TestNodes*, update *CurrentNodes* and update *TerminatorNodes*

Update *TestNodes* Whenever the vehicle starts a new driving cycle (VehiclePreviousStatus was RECHARGING and VehicleCurrentStatus is not RECHARGING) the *TestNodes* are selected as the *StartNodes* already present in the graph. For improved robustness to GPS errors, we add to the *TestNodes* list also the successors of the *StartNodes*, down to a configurable N-th level, for example $N = 2$. If the vehicle was already driving, instead, we do the same process but taking from the *CurrentNodes* instead of the *StartNodes*.

Update *CurrentNodes* First, we remove from *CurrentNodes*, the nodes which have a position further than a parameter $K1$ from the current position of the vehicle. The removed nodes are temporarily added to *DevalidatedNodes* list. Then, for each *TestNode*, we make it a *CurrentNode* if its distance from the current vehicle position is lower than a parameter $K3$, while we remove it from the *TestNodes* if it is greater than $K2$. In case after this procedure the *CurrentNodes* list is empty, we have to add a new node to the graph, in the current position. In case of a new driving cycle, the newly added node is placed into *StartNodes*, otherwise it is set as a successor of the *DevalidatedNode* closer to the current position. As shown in Figure 8.2, in case no memory is available for a new node, space is obtained by removing from the graph the old (not habitual) route branches. Note that the above mentioned parameters adjust the balance between creating a new graph branch in case of deviations from already existing routes and considering the vehicle still on an existing route, without the need to create a new one.

Update *TerminatorNodes* When the vehicle is connected to a charging point, add the *CurrentNodes* to the *TerminatorNodes*, and set its *TotalConsumption* property equal to the energy consumption from the last recharge. In case the same *TerminatorNodes* is

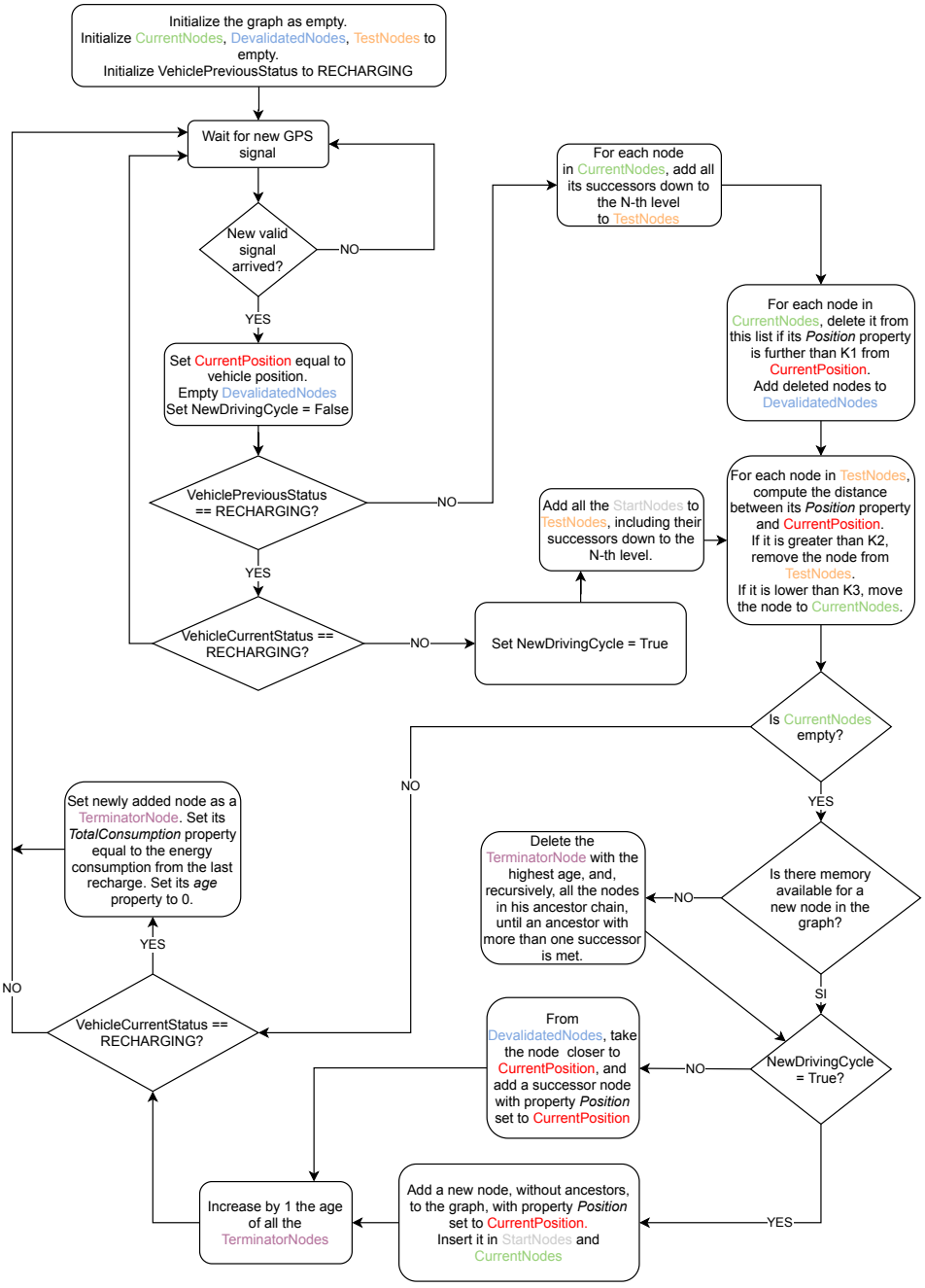


FIGURE 8.2: Graph update algorithm.

reached several times, the corresponding energy consumption can be updated simply averaging all the recorded consumptions. The *TerminatorNodes* also stores an *occurrency* variable, which stores the number of times it has actually been reached.

It must be notices that by setting the above mentioned parameters one can adjust the computational burden of the algorithm. The *TestNodes* selection mechanism allows not to have the need to explore the whole graph at every iteration looking for *CurrentNodes*

close to the current position. This solution makes the algorithm very robust to extended graphs.

8.2.3 Energy prediction

At any point in time, the graph can be explored looking for an estimate of the energy needed to complete the driving cycle. Starting from the *CurrentNode*, passing through all its successors and down to the *TerminatorNodes*, it is possible to get if we're driving on one or many habitual routes, where all of these end, and how much energy they require to complete. The *occurrency* and *TotalConsumption* of all the reachable terminator nodes can be for example used to create a statistical distribution of the possible future energy consumption. The exact method used is out from the scope of this work, which aims only at providing the above mentioned data and store it in an efficient way.

8.3 Conclusion

In this chapter we illustrated a very lightweight algorithm to estimate the residual energy needed to drive a vehicle along a habitual route, according to the previous routes abservations. The method is specifically designed for resource constrained devices, thus being deployable on an embedded microcontroller, and avoiding the need for connection to an external server. The algorithm makes use of very few information from the vehicle, therefore being extremely easy to implement in real world scenarios.

This work complements what has been presented in the previous chapters by representing a form of artificial intelligence not relying on machine learning, but on pure algorithmic design, following a more traditional approach. In particular, the exposed method wants to stress one of the main messages we want to communicate in this thesis: although the undoubtedly great power of modern machine learning-based approaches, solution designers shall avoid excessive bias toward their application, and, in particular when targeting edge devices, they should always consider the added value that application-specific algorithm design can bring, even in the development of Artificial Intelligence solutions.

Chapter 9

Conclusion

In this thesis we analyzed the problem of deploying Artificial Intelligence solutions on resource constrained devices, an increasingly important topic in the modern world of smart battery powered devices and autonomous machines. Although also HW based approaches exists, in this thesis the challenge has been faced from a SW perspective, aiming at optimizing the computational and memory footprint of AI algorithms, thus speeding up computation and allowing their execution down to tiny microcontrollers for edge devices.

One of the main peculiarities of this work is the "Application-aware" approach to AI optimization, which wants to be complementary to the several already existing methods for Application-independent neural network acceleration. We showed that, if on one side end-to-end-machine learning-based solutions can be extremely powerful in solving problems such as image classification or object detection, on the other side mixing machine learning with application-specific knowledge in the form of feature engineering and manual algorithm design enables much higher optimization, and becomes crucial when targeting deployment on embedded devices.

As a first contribution, we performed a structured survey of the methods used nowadays to accelerate neural networks on embedded devices. We grouped these methods into two main categories, HW- and SW-based, highlighting for each of them the corresponding subcathegories, weak and strength points. According to this classification, we placed our work in the subcathegory of SW-based solutions which operate at architecture design level, thus providing a clear contextualization of our contribution with respect to other works.

Secondly, taking as a reference an object detection application, we have seen the great potential that feature engineering can have in boosting the performance of a neural

network. Preprocessing the input image, in our case performing a form of frame differencing which highlights moving objects, highlights important features to be learnt by the neural network. In this context, we've also seen how beneficial can be having parallel branches in the same network to process different types of information and merging their outputs along the network. In our work, applying these methods dramatically improved the performance that general purpose object detection algorithms achieved, and made it possible to deploy the solution on an embedded device to run in real time.

As a third contribution we've shown how the same principles expressed above can be used to dramatically simplify the portion of a neural network which needs to be retrained for application-specific needs, thus making it also possible to perform the training directly on an embedded device in a reasonable time. In a reference hand pose classification application, we achieved this result by splitting the network in a hand keypoint detection part, which doesn't require retraining, and a classification head. Differently from the previous contribution, in this case the relevant features (the hand keypoints) are not extracted with an handcrafted algorithm but by using a dedicated pretrained and fixed neural network. However, also in this case, an application-aware choice of these features, on paper, was crucial to minimize the classifier size, training time, and the overall performance of the solution.

Then, in this thesis we addressed the problem of implementing deep neural networks on a microcontroller featuring very limited computational and memory resources. Taking as a reference an object detection application, to fit the constraints of such a resource limited device we had to put together a lightweight object segmentation network with handcrafted postprocessing of its output. Moreover, we conducted a deteailed analysis of the impact of each architectural choice and hyperparameters on the memory footprint of the solution in order to apply the chirurgical changes needed to reduce the memory requirements without impacting the overall accuracy. The final solution was implemented on an STM32 microcontroller. As an alternative approach, we also experimented the use of Neural Architecture Search to design an extremely lightweight neural network for object detection on a microcontroller. We've demonstrated how NAS can be used in conjunction with a cost function which takes into account not only the network accuracy, but also its weight, to generate a resource-aware network.

Finally, we presented an algorithm that we patented, which is a great example of empowering resource constrained devices with a form of artificial intelligence obtained through very lightweight algorithms designed in a more traditional approach instead of using machine learning.

We believe that our work will inspire future researchers and engineers, promoting a way of approaching the design of artificial intelligence for embedded systems which is more

aware of the consequences of each design choice on the memory and computational footprint of the solution, and which emphatizes the importance of performing optimization not only by using general purpose off-the-shelf methods, but also in an application-aware fashion following the principles illustrated in our work.

Bibliography

[1] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2(11):e7, 2017.

[2] Max Ferguson, Ronay ak, Yung-Tsun Lee, and Kincho Law. Automatic localization of casting defects with convolutional neural networks. pages 1726–1735, 12 2017.

[3] A comprehensive guide to convolutional neural networks. `https://towardsdatascience.com`, 2022. Accessed: 2022-10-30.

[4] A. Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *ArXiv*, abs/1605.07678, 2016.

[5] Xiao Youzi, Zhiqiang Tian, Jiachen Yu, Yinshu Zhang, Shuai Liu, Shaoyi Du, and Xuguang Lan. A review of object detection based on deep learning. *Multimedia Tools and Applications*, 79, 09 2020. doi: 10.1007/s11042-020-08976-6.

[6] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[7] Frank Denneman. Training vs inference, memory consumption by neural networks. `https://frankdenneman.nl`, 2022. Accessed: 2022-06-08.

[8] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[9] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[10] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016*

ACM/SIGDA international symposium on field-programmable gate arrays, pages 26–35, 2016.

[11] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 2017.

[12] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019. URL `http://arxiv.org/abs/1905.02244`.

[13] Andrea Albanese, Tommaso Taccioli, Tommaso Apicella, Davide Brunelli, and Edoardo Ragusa. Design and deployment of an efficient landing pad detector. In *International Conference on System-Integrated Intelligence*, pages 137–147. Springer, 2023.

[14] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL `http://arxiv.org/abs/1409.4842`.

[19] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[20] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[21] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[22] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[23] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[24] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[25] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[26] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541, 2006.

[27] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015.

[28] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[29] Shimeng Yu. Neuro-inspired computing with emerging nonvolatile memorys. *Proceedings of the IEEE*, 106(2):260–285, 2018.

[30] Masafumi Hagiwara. Removal of hidden units and weights for back propagation networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, volume 1, pages 351–354. IEEE, 1993.

[31] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

[32] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[33] Ala Mhalla, Thierry Chateau, Sami Gazzah, and Najoua Essoukri Ben Amara. An embedded computer-vision system for multi-object detection in traffic surveillance. *IEEE Transactions on Intelligent Transportation Systems*, 20(11):4006–4018, 2018.

[34] Edoardo Ragusa, Christian Gianoglio, Rodolfo Zunino, and Paolo Gastaldo. Image polarity detection on resource-constrained devices. *IEEE Intelligent Systems*, 2020.

[35] Yu-Chuan Huang, I-No Liao, Ching-Hsuan Chen, Tsì-Uí İk, and Wen-Chih Peng. Tracknet: A deep learning network for tracking high-speed and tiny objects in sports applications. In *2019 16th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–8. IEEE, 2019.

[36] HawkEye Innovations Ltd. Hawkeye system. `http://www.hawkeyeinnovations.com`, 2022. Accessed: 2020-06-08.

[37] ST Microelectronics NV. Stm32 32bit arm cortex mcus. `https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html` todo, 2022. Accessed: 2020-06-08.

[38] Intel Corporation. Intel movidius neural compute stick. `https://software.intel.com/content/www/us/en/develop/articles/intel-movidius-neural-compute-stick.html`, 2022. Accessed: 2020-06-08.

[39] NVIDIA Corporation. Nvidia autonomous machines. `https://www.nvidia.com/autonomous-machines/embedded-systems/`, 2022. Accessed: 2020-06-08.

[40] Deebul Nair, Amirhossein Pakdaman, and Paul G Plöger. Performance evaluation of low-cost machine vision cameras for image-based grasp verification. *arXiv preprint arXiv:2003.10167*, 2020.

[41] Yang Liu, Peng Sun, Nickolas Wergeles, and Yi Shang. A survey and performance evaluation of deep learning methods for small object detection. *Expert Systems with Applications*, page 114602, 2022.

[42] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

[43] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C Berg. Dssd: Deconvolutional single shot detector. *arXiv preprint arXiv:1701.06659*, 2017.

[44] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International journal of computer vision*, 128(2):261–318, 2020.

[45] Guang Chen, Haitao Wang, Kai Chen, Zhijun Li, Zida Song, Yinlong Liu, Wenkai Chen, and Alois Knoll. A survey of the four pillars for small object detection: Multiscale representation, contextual information, super-resolution, and region proposal. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2020.

[46] Lisha Cui, Rui Ma, Pei Lv, Xiaoheng Jiang, Zhimin Gao, Bing Zhou, and Mingliang Xu. Mdssd: Multi-scale deconvolutional single shot detector for small objects. *arXiv preprint arXiv:1805.07009*, 2018.

[47] Mengmeng Zhang, Wei Li, and Qian Du. Diverse region-based cnn for hyperspectral image classification. *IEEE Transactions on Image Processing*, 27(6):2623–2634, 2018.

[48] Chenyi Chen, Ming-Yu Liu, Oncel Tuzel, and Jianxiong Xiao. R-cnn for small object detection. In *Asian conference on computer vision*, pages 214–230. Springer, 2016.

[49] Jianan Li, Xiaodan Liang, Yunchao Wei, Tingfa Xu, Jiashi Feng, and Shuicheng Yan. Perceptual generative adversarial networks for small object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1222–1230, 2017.

[50] Hu Lin, Jingkai Zhou, Yanfen Gan, Chi-Man Vong, and Qiong Liu. Novel up-scale feature aggregation for object detection in aerial images. *Neurocomputing*, 411: 364–374, 2020.

[51] Kinjal A Joshi and Darshak G Thakore. A survey on moving object detection and tracking in video surveillance system. *International Journal of Soft Computing and Engineering*, 2(3):44–48, 2012.

[52] Pakorn KaewTraKulPong and Richard Bowden. An improved adaptive background mixture model for real-time tracking with shadow detection. In *Video-based surveillance systems*, pages 135–144. Springer, 2002.

[53] Massimo Piccardi. Background subtraction techniques: a review. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 4, pages 3099–3104. IEEE, 2004.

[54] Gábor Váraljai and Sándor Szénási. Projectile detection and avoidance using computer vision. In *2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000157–000160. IEEE, 2020.

[55] Rupali S Rakibe and Bharati D Patil. Background subtraction algorithm based human motion detection. *International Journal of scientific and research publications*, 3(5):2250–3153, 2013.

[56] Thanarat Horprasert, David Harwood, and Larry S Davis. A statistical approach for real-time robust background subtraction and shadow detection. In *Ieee iccv*, volume 99, pages 1–19. Citeseer, 1999.

[57] ZuWhan Kim. Real time object tracking based on dynamic feature grouping with background subtraction. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[58] Mennatullah Siam, Heba Mahgoub, Mohamed Zahran, Senthil Yogamani, Martin Jagersand, and Ahmad El-Sallab. Modnet: Moving object detection network with motion and appearance for autonomous driving. *arXiv preprint arXiv:1709.04821*, 2017.

[59] Zhaofan Qiu, Ting Yao, and Tao Mei. Learning spatio-temporal representation with pseudo-3d residual networks. In *proceedings of the IEEE International Conference on Computer Vision*, pages 5533–5541, 2017.

[60] Rodney LaLonde, Dong Zhang, and Mubarak Shah. Clusternet: Detecting small objects in large scenes by exploiting spatio-temporal information. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4003–4012, 2018.

[61] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[62] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[63] Lars Wilko Sommer, Michael Teutsch, Tobias Schuchert, and Jürgen Beyerer. A survey on moving object detection for wide area motion imagery. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–9. IEEE, 2016.

[64] Yun Liu, Krishna Gadepalli, Mohammad Norouzi, George E Dahl, Timo Kohlberger, Aleksey Boyko, Subhashini Venugopalan, Aleksei Timofeev, Philip Q Nelson, Greg S Corrado, et al. Detecting cancer metastases on gigapixel pathology images. *arXiv preprint arXiv:1703.02442*, 2017.

[65] Meng Wang, Wei Li, and Xiaogang Wang. Transferring a generic pedestrian detector towards specific scenes. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3274–3281. IEEE, 2012.

[66] NVIDIA Corporation. Jetson nano. `https://developer.nvidia.com/embedded/jetson-nano`, 2022. Accessed: 2020-06-08.

[67] Edoardo Ragusa, Tommaso Apicella, Christian Gianoglio, Rodolfo Zunino, and Paolo Gastaldo. Design and deployment of an image polarity detector with visual attention. *Cognitive Computation*, Available Online, 10.1007/s12559-021-09829-6.

[68] Marko Markovic, Strahinja Dosen, Dejan Popovic, Bernhard Graimann, and Dario Farina. Sensor fusion and computer vision for context-aware control of a multi degree-of-freedom prosthesis. *Journal of neural engineering*, 12(6):066022, 2015.

[69] Ghazal Ghazaei, Ali Alameer, Patrick Degenaar, Graham Morgan, and Kianoush Nazarpour. Deep learning-based artificial vision for grasp classification in myoelectric hands. *Journal of neural engineering*, 14(3):036025, 2017.

[70] Taiqian Wang, Yande Li, Junfeng Hu, Aamir Khan, Li Liu, Caihong Li, Ammarah Hashmi, and Mengyuan Ran. A survey on vision-based hand gesture recognition. In *International Conference on Smart Multimedia*, pages 219–231. Springer, 2018.

[71] Mário P Véstias, Rui Policarpo Duarte, José T de Sousa, and Horácio C Neto. Moving deep learning to the edge. *Algorithms*, 13(5):125, 2020.

[72] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[73] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E Dahl, and Geoffrey E Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[74] Yezhou Yang, Cornelia Fermuller, Yi Li, and Yiannis Aloimonos. Grasp type revisited: A modern perspective on a classical feature for vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 400–408, 2015.

[75] Minjie Cai, Kris M Kitani, and Yoichi Sato. An ego-vision system for hand grasp analysis. *IEEE Transactions on Human-Machine Systems*, 47(4):524–535, 2017.

[76] Edoardo Ragusa, Christian Gianoglio, Rodolfo Zunino, and Paolo Gastaldo. Data-driven video grasping classification for low-power embedded system. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 871–874. IEEE, 2019.

[77] Edoardo Ragusa, Christian Gianoglio, Filippo Dalmonte, and Paolo Gastaldo. Video grasping classification enhanced with automatic annotations. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society*, pages 23–29. Springer, 2020.

[78] Thomas Feix, Javier Romero, Heinz-Bodo Schmiedmayer, Aaron M Dollar, and Danica Kragic. The grasp taxonomy of human grasp types. *IEEE Transactions on human-machine systems*, 46(1):66–77, 2015.

[79] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[80] Filippos Gouidis, Paschalis Panteleris, Iason Oikonomidis, and Antonis Argyros. Accurate hand keypoint localization on mobile devices. In *2019 16th International Conference on Machine Vision Applications (MVA)*, pages 1–6. IEEE, 2019.

[81] Dushyant Mehta, Srinath Sridhar, Oleksandr Sotnychenko, Helge Rhodin, Mohammad Shafiei, Hans-Peter Seidel, Weipeng Xu, Dan Casas, and Christian Theobalt. Vnect: Real-time 3d human pose estimation with a single rgb camera. *ACM Transactions on Graphics (TOG)*, 36(4):1–14, 2017.

[82] Andrea Bandini and José Zariffa. Analysis of the hands in egocentric vision: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[83] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. Ultra low power deep-learning-powered autonomous nano drones. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2018)*. ETH Zurich, 2018.

[84] Edoardo Ragusa, Christian Gianoglio, Strahinja Dosen, and Paolo Gastaldo. Hardware-aware affordance detection for application in portable embedded systems. *IEEE Access*, 9:123178–123193, 2021.

[85] Shahrokh Paravarzar and Belqes Mohammad. Motion prediction on self-driving cars: A review. *arXiv preprint arXiv:2011.03635*, 2020.

[86] Google LLC. Coral. `https://coral.ai`, 2022. Accessed: 2022-01-22.

[87] Intel Corporation. Intel movidius vision processing units. `https://www.intel.co.uk/content/www/uk/en/products/details/processors/movidius-vpu.html`, 2022. Accessed: 2022-01-22.

[88] Google LLC. Tensorflow lite. `https://www.tensorflow.org/lite`, 2022. Accessed: 2022-01-22.

[89] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[90] Mate Krišto, Marina Ivasic-Kos, and Miran Pobar. Thermal object detection in difficult weather conditions using yolo. *IEEE Access*, 8:125459–125476, 2020.

[91] GitHub user xiaochus. Mobilenetv3 segmentation implementation. `https://github.com/xiaochus/MobileNetV3`, 2022. Accessed: 2022-01-22.

[92] Mate Krišto, Marina Ivasic-Kos, and Miran Pobar. Thermal object detection in difficult weather conditions using yolo. *IEEE Access*, 8:125459–125476, 2020.

[93] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[94] Carlo Giorgio Grlj, Nino Krznar, and Marko Pranjić. A decade of uav docking stations: A brief overview of mobile and fixed landing platforms. *Drones*, 6(1):17, 2022.

[95] Abdul Hafeez, Mohammed Aslam Husain, SP Singh, Anurag Chauhan, Mohd Tauseef Khan, Navneet Kumar, Abhishek Chauhan, and SK Soni. Implementation of drone technology for farm monitoring & pesticide spraying: A review. *Information Processing in Agriculture*, 2022.

[96] Linjie Xing, Xiaoyan Fan, Yaxin Dong, Zenghui Xiong, Lin Xing, Yang Yang, Haicheng Bai, and Chengjiang Zhou. Multi-uav cooperative system for search and rescue based on yolov5. *International Journal of Disaster Risk Reduction*, 76: 102972, 2022.

[97] Lucas Prado Osco, José Marcato Junior, Ana Paula Marques Ramos, Lúcio André de Castro Jorge, Sarah Narges Fatholahi, Jonathan de Andrade Silva, Edson Takashi Matsubara, Hemerson Pistori, Wesley Nunes Gonçalves, and Jonathan Li. A review on deep learning in uav remote sensing. *International Journal of Applied Earth Observation and Geoinformation*, 102:102456, 2021.

[98] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

[99] Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. Fast hardware-aware neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 692–693, 2020.

[100] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.

[101] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. A comprehensive survey on hardware-aware neural architecture search. *arXiv preprint arXiv:2101.09336*, 2021.

[102] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[103] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[104] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[105] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

[106] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3: 517–532, 2021.

[107] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Lin. Hw-nas-bench: Hardware-aware neural architecture search benchmark. *arXiv preprint arXiv:2103.10584*, 2021.

[108] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *European conference on computer vision*, pages 544–560. Springer, 2020.

[109] Albert Shaw, Daniel Hunter, Forrest Landola, and Sammy Sidhu. Squeezenas: Fast neural architecture search for faster semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[110] Su Yeon Choi and Dowan Cha. Unmanned aerial vehicles using machine learning for autonomous flight; state-of-the-art. *Advanced Robotics*, 33(6):265–277, 2019.

[111] Laith Abualigah, Ali Diabat, Putra Sumari, and Amir H Gandomi. Applications, deployments, and integration of internet of drones (iod): a review. *IEEE Sensors Journal*, 2021.

[112] Jangwon Lee, Jingya Wang, David Crandall, Selma Šabanović, and Geoffrey Fox. Real-time, cloud-based object detection for unmanned aerial vehicles. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 36–43. IEEE, 2017.

[113] Anis Koubâa and Basit Qureshi. Dronetrack: Cloud-based real-time object tracking using unmanned aerial vehicles over the internet. *IEEE Access*, 6:13810–13824, 2018.

[114] Sunggoo Jung, Sunyou Hwang, Heemin Shin, and David Hyunchul Shim. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. *IEEE Robotics and Automation Letters*, 3(3):2539–2544, 2018.

[115] Nils Tijtgat, Wiebe Van Ranst, Toon Goedeme, Bruno Volckaert, and Filip De Turck. Embedded real-time object detection for a uav warning system. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 2110–2118, 2017.

[116] Andrea Albanese, Matteo Nardello, and Davide Brunelli. Low-power deep learning edge computing platform for resource constrained lightweight compact uavs. *Sustainable Computing: Informatics and Systems*, page 100725, 2022.

[117] Mohamad Hazwan Mohd Ghazali and Wan Rahiman. Vibration-based fault detection in drone using artificial intelligence. *IEEE Sensors Journal*, 22(9):8439–8448, 2022.

[118] Siddharth Gupta, Prabhat Kumar Rai, Abhinav Kumar, Phaneendra K Yalavarthy, and Linga Reddy Cenkeramaddi. Target classification by mmwave fmcw radars using machine learning on range-angle images. *IEEE Sensors Journal*, 21(18):19993–20001, 2021.

[119] Muhammad Hafidz Fazli Md Fauadi, Suriati Akmal, Mahasan Mat Ali, Nurul Izah Anuar, Samad Ramlan, Ahamad Zaki Mohd Noor, and Nurfadzylah Awang. Intelligent vision-based navigation system for mobile robot: A technological review. *Periodicals of Engineering and Natural Sciences*, 6(2):47–57, 2018.

[120] Shuo Li, Michaël MOI Ozo, Christophe De Wagter, and Guido CHE de Croon. Autonomous drone race: A computationally efficient vision-based navigation and control strategy. *Robotics and Autonomous Systems*, 133:103621, 2020.

[121] Ranzhen Ren, Lichuan Zhang, Lu Liu, and Yijie Yuan. Two auvs guidance method for self-reconfiguration mission based on monocular vision. *IEEE Sensors Journal*, 21(8):10082–10090, 2021.

[122] Bedada Endale, Abera Tullu, Hayoung Shi, and Beom-Soo Kang. Robust approach to supervised deep neural network training for real-time object classification in cluttered indoor environment. *Applied Sciences*, 11(15):7148, 2021.

[123] Marcin Paszkuta, Jakub Rosner, Damian Peszor, Marcin Szender, Marzena Wojciechowska, Konrad Wojciechowski, and Jerzy Paweł Nowacki. Uav on-board emergency safe landing spot detection system combining classical and deep learning-based segmentation methods. In *Asian Conference on Intelligent Information and Database Systems*, pages 467–478. Springer, 2021.

[124] Pranay Mathur, Yash Jangir, and Neena Goveas. A generalized kalman filter augmented deep-learning based approach for autonomous landing in mavs. In *2021 International Symposium of Asian Control Association on Intelligent Robotics and Industrial Automation (IRIA)*, pages 1–6. IEEE, 2021.

[125] JA García-Pulido, G Pajares, and S Dormido. Uav landing platform recognition using cognitive computation combining geometric analysis and computer vision techniques. *Cognitive Computation*, pages 1–21, 2022.

[126] Miguel Saavedra Ruiz, Ana Maria Pinto Vargas, and Victor Romero Cano. Detection and tracking of a landing platform for aerial robotics applications. In *2018 IEEE 2nd Colombian Conference on Robotics and Automation (CCRA)*, pages 1–6. IEEE, 2018.

[127] Phong Ha Nguyen, Ki Wan Kim, Young Won Lee, and Kang Ryoung Park. Remote marker-based tracking for uav landing using visible-light camera sensor. *Sensors*, 17(9):1987, 2017.

[128] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

[129] Weiwen Jiang, Lei Yang, Sakyasingha Dasgupta, Jingtong Hu, and Yiyu Shi. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4154–4165, 2020.

[130] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2019.