ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

**Scuola di Ingegneria e Architettura**
**Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche**

# Edge AI: Deep Learning techniques for Computer Vision applied to embedded systems

**Elaborato finale in Machine Learning**

**Relatore:**
**Prof. Davide Maltoni**

**Co-relatore:**
**dott. Vincenzo Lomonaco**

**Presentata da:**
**Giacomo Bartoli**

**II Sessione**
**Anno Accademico 2018/19**

*Alla mia famiglia,*

*per avermi sempre appoggiato in ogni mia scelta.*

*When machines can see, doctors and nurses will have extra pairs of tireless eyes to help them to diagnose and take care of patients. Cars will run smarter and safer on the road. Robots, not just humans, will help us to brave the disaster zones to save the trapped and wounded. We will discover new species, better materials, and explore unseen frontiers with the help of the machines.*

*Little by little, we're giving sight to the machines. First, we teach them to see. Then, they help us to see better. For the first time, human eyes won't be the only ones pondering and exploring our world. We will not only use the machines for their intelligence, we will also collaborate with them in ways that we cannot even imagine.*

*This is my quest: to give computers visual intelligence and to create a better future for the world.*

Fei-Fei Li,  Head of Stanford AI Lab

# Introduction

More than half a century ago the computer was invented. Since that day many felt the essence of thinking, the heart of intelligence was found. Reproducing intelligence seemed to be possible by the way computers worked. All of a sudden, it became possible to simulate thinking, problem solving, even natural language: Artificial Intelligence was born. The human brain was interpreted as a computer. We all agree on the fact that computers have been one of the biggest achievements in history, but computers have not fulfilled the expectations of producing intelligence as we normally understand it. Here are some cases where AI seemed to be more smarter than humans:

- In 1997, world chess champion Garry Kasparov played against Deep Blue, a program built by IBM. As you already probably know, Kasparov was defeated by the machine.
- In 2011, Watson, another software made by IBM, won against a human player. The game was called Jeopardy and it was, essentially, a quiz show where players were asked questions concerning any domain: general knowledge, history, politics, economics etc.

Both these cases do not necessarily imply that a computer must be smart in order to achieve these goals. In fact, a chess algorithm can perform very well in searching through many positions and possible moves without requiring so much intelligence. Even retrieving information from a pre-built archive is not something we can define as smart. While scientists and engineers were pushing themselves to build robots capable of doing commonplace activities (path navigation, obstacle avoidance, face recognition) they realized that these tasks were extremely hard for machines. What is easy for people is extremely hard for computers and, viceversa, what is often hard for humans is easy for computers. By the mid-1990s, researchers from Artificial Intelligence had to admit that perhaps the idea of computers as smart machines was misguided. As Rolf Pfeifer and Christian Scheier wrote in their book “Understanding Intelligence” [1], the brain does not simply ”run programs” but it does something completely different. What the brain does is reinforcing connections among neurons, which are activated depending on specific stimuli. Essentially, the brain learns from experience. The idea of creating a class of algorithms that learn from experience is summed up by a subfield of Artificial Intelligence called Machine Learning. The more examples we give them, the more they learn and, subsequently, this knowledge can be applied for inference over new examples.

When little children still struggle to talk, parents still talk to them as if they could understand everything. This way, the brain of the children is nourished by examples. Furthermore, when children start uttering their first words and they use a term wrong, the parents correct them with the right word and, again, children learn. This is what in Machine Learning is called Supervised Learning: a model is fed by thousands of examples and then it is able to predict, act, classify. However, if the model is wrong about a prediction it can be automatically corrected, learning from its own mistakes.

In the last decade, Machine Learning techniques have been used in different fields, ranging from finance to healthcare and even marketing. Amongst all these techniques, the ones adopting a Deep Learning approach were revealed to outperform humans in tasks such as object detection, image classification and speech recognition.
This thesis introduces the basic concepts of Machine Learning and Deep Learning, and then deepens the convolutional model (CNN). The second chapter specifically introduces Deep Learning architectures present in the scientific literature for object recognition. Then we introduce the concept of

"Edge AI", that is the possibility to build learning models capable of making inference locally, without any dependence on expensive servers or cloud services. A first case study we consider is based on the Google AIY Vision Kit, an intelligent camera equipped with a graphic board to optimize Computer Vision algorithms. Then we focus on two applications: we want to test the performances of CORe50, a dataset for continuous object recognition, on embedded systems. The techniques developed in the previous chapters will then be used to solve a challenge within the Audi Autonomous Driving Cup 2018, where a mobile car equipped with a camera, sensors and a graphic board must recognize pedestrians and stop before hitting them.

# Introduzione

Più di mezzo secolo fa il computer fu inventato. A partire da quel giorno, molte persone intuirono che l'essenza del ragionamento, il cuore dell'intelligenza fu trovato. Sembrava a tutti possibile sfruttare il modo in cui i computer operavano per riprodurre intelligenza. All'improvviso sembrò possibile simulare il pensiero, svolgere in maniera automatizzata attività di problem solving ed anche riprodurre il linguaggio naturale: l'Intelligenza Artificiale era appena nata. Il cervello umano veniva interpretato proprio come un computer.
Siamo tutti d'accordo sul fatto che il computer sia stato uno dei più grandi traguardi nella storia, tuttavia i computer non hanno raggiunto l'aspettativa di riprodurre intelligenza così come noi la intendiamo. Analizziamo i seguenti casi in cui l'AI sembrava essere più intelligente di esseri umani:

- Nel 1997, Garry Kasparov, campione mondiali di scacchi, giocò contro Deep Blue, un programma costruito da IBM. Come noto, Kasparov fu battuto dal calcolatore.
- Nel 2011, Watson, un altro programma fatto da IBM, vinse contro un altro campione. Il gioco in questione si chiamava Jeopardy e consisteva in una specie di quiz con domande relative ad ogni possibile ambito: cultura generale, storia, politica, economia ecc ecc.

Entrambi i casi non richiedono necessariamente intelligenza da parte della macchina. Infatti, la ricerca di possibili combinazioni e mosse, come nel caso degli scacchi, non richiede troppa intelligenza. Anche reperire informazioni

da un archivio è qualcosa che non si può considerare intelligente. Mentre scienziati ed ingegneri concentravano le loro forze nello sviluppo di robot capaci di intraprendere attività per umani considerate comuni (muoversi in un ambiente, evitare ostacoli, riconoscere facce, afferrare oggetti) si resero conto che questi task erano estremamente difficili per le macchine. Ciò che è facile per le persone è molto difficile per i computer e, viceversa, cio che è difficile per le persone è facile per i computer.
A cavallo degli anni 90', ricercatori nell'ambito dell'Intelligenza Artificale dovettero ammettere che forse l'idea di computer inteso come una macchina intelligente era sbagliata. Come Rolf Pfeifer e Christian Scheier scrivono nel loro libro "Understanding Intelligence" [1], il cervello non manda semplicemente programmi in esecuzione ma fa qualcosa di completamente diverso. Ciò che il cervello fa è rinforzare le sinapsi, le connessioni tra diversi neuroni a seconda degli stimoli che ricevono dall'esterno. Più stimoli riceviamo, più le connessioni si rinforzano. In poche parole, il cervello apprende dall'esperienza. L'idea di creare una classe di algoritmi che impara dall'esperienza è riassunta in una sottobranca dell'Intelligenza Artificiale chiamata Machine Learning. Più esempi diamo in pasto a questi algoritmi e più loro apprendono e, di conseguenza, sono in grado di utilizzare questa conoscenza per fare inferenza su nuovi esempi.
Quando i bambini, ancora piccoli, stentano a parlare, i genitori parlano loro come se potessero intendere ogni cosa. In questa maniera, il cervello dei bambini è nutrito in maniera continua di esempi. In più, quando il bambino impara le prime parole e le usa in maniera sbagliata il genitore lo corregge. Questo è ciò che in Machine Learning viene chiamato come Apprendimento Supervisionato: un modello è alimentato da migliaia di esempi e, di conseguenza, diventa in grado di predire, agire, classificare. Tuttavia, se il modello si sbaglia in uno di questi task può essere automaticamente corretto. Si apprende dai propri errori.
Negli ultimi dieci anni le tecniche di Machine Learning sono state applicate ai più svariati ambiti, dalla finanza alla medicina fino al marketing. Tra tutte

queste tecniche, quelle basate sul Deep Learning hanno dimostrato di avere performance migliori degli essere umani in task come rilevazione di oggetti, classificazione di immagini e riconoscimento del parlato.
L'elaborato in questione introduce i concetti base dell'apprendimento automatico e del Deep Learning, per poi approfondire il modello convoluzionale (CNN). Il secondo capitolo espone e confronta le architetture di Deep Learning presenti in letteratura per il riconoscimento di oggetti. Si introduce poi il concetto di "Edge AI", ovvero la possibilità di costruire modelli di apprendimento in grado di fare inferenza localmente, senza alcuna dipendenza da servizi cloud o server costosi. Il caso di studio è basato sul Google AIY Vision Kit, una camera intelligente dotata di una scheda grafica per l'ottimizzazione di algoritmi di Computer Vision. Lo scopo finale è duplice: da una parte si vogliono testare le performance di CORe50, dataset per il riconoscimento continuo di oggetti, su sistemi embedded. In seguito, le tecniche sviluppate nei capitoli precedenti saranno utilizzate per risolvere una challenge all'interno dell'Audi Autonomous Driving Cup 2018, dove una macchina dotata di camera, sensori e scheda grafica deve riconoscere i pedoni e fermarsi.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Machine Learning

Machine Learning is a subject located at the intersection among Statistics, Data Analysis, Pattern Recognition and Artificial Intelligence. As already explained in the introduction, the main idea is that of feeding a model with many examples and, later on, applying inference. Thus, machines "magically" learn from data. The point is to define the correct model for the learning phase. More formally, we can say that "a computer program is said to learn from experience E with respect to some classes of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [2] . A dataset is normally split into three parts: the first part, called the training set, is used for the training phase; the second one, the test set, is aimed at the evaluation, while the last one, the validation set, is suitable for tuning hyper parameters. The training phase is characterized by the learning process and knowledge acquisition of the model. The subsequent application of said knowledge is the testing phase.

If, during the training phase, data are labelled then it is called Supervised Learning. The goal is to find a function that maps the input data on its corresponding classes. In case there are no labels, which makes the problem harder than before, it is called Unsupervised Learning. The training set can even be partially labelled: this is what Semi-supervised Learning means. The last paradigm of Machine Learning is Reinforcement Learning: an agent takes action interacting with the environment. To each action there corresponds a reward. The goal of the agent is to maximize the sum of the total rewards. This approach has shown to be incredibly effective when an agent must learn behaviours: control theory, simulation, gaming.

## 1.1 Machine Learning Data

Machine Learning algorithms can handle different kinds of data:

- Numerical: these are values associated with measurable characteristics. There is an order among them and they can be both discrete or continuous. They can be represented as vectors in a multidimensional space. Ex: finding the height, weight or foot size of a given person .

- Categorical: values associated with qualitative characteristics. Binary values are considered categorical as well. Ex: finding the sex or blood type of given a person.

- Sequences: these data express a relationship between time and space. What really matters is their position into a sequence and the reference with predecessors and successors. Ex: a sequence of words, streams of data.

## 1.2 Machine Learning Problems

Machine Learning techniques can be applied for facing several problems, which are:

- Classification: starting from a labelled dataset, the aim is to correctly classify new patterns as belonging to their classes. Ex: given the weight and height of a person, is that a male or female? Traditional approaches to classification tasks are: supported vector machines (SVM), decision trees, perceptron, nearest neighbors and other models.

- Clustering: grouping data that share the same characteristics. Data are not labelled. The purpose is to minimize intra-cluster distance and maximize inter-cluster distance. Clustering algorithms are: K-means, K-median, X-means, etc.

- Regression: the task of approximating a mapping function from input variables to a continuous output variable. Even if it may seem close to classification, the main difference is the fact that regression considers continuous variables as output, while classification works only with discrete values.

- Dimensionality Reduction: mapping a space $K^n$ to $K^m$ where $m < n$. This operation surely implies the loss of some information that is not supposed to be important. The most used technique is called PCA (Principal Component Analysis) and it exploits eigenvectors to discard the variance.

- Representation Learning: identifies a set of algorithms aimed at the automatic processing of data provided during the learning phase, for the discovery of a better representation of said data. Representation Learning takes the raw data provided during the pre-processing phase before they are classified. Many of the deep learning techniques (such as convolutional neural networks) operate this way, using raw data as input and automatically extracting the necessary features to solve the problem.

During the training phase we want to find a model that maps features to labels and then we proceed with the test of the model on the validation set. The model must be able to generalize and find relationships between features and labels even on a new dataset. This is how we measure performances. However, it can happen that the model seems to perform very well but then it underperforms on the validation set. This is what is known as *overfitting*: the model knows how to fit the data, but there is no real understanding of

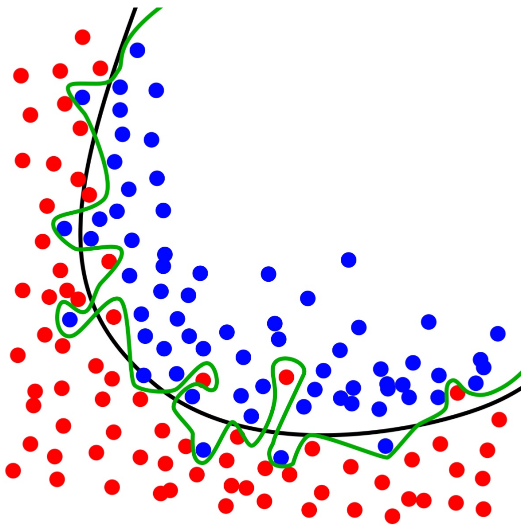


Figure 1.1: Example of overfitting

the relationship between data and labels. Figure 1.1 shows an example of overfitting.

Machine Learning algorithms have a performance limit. When this limit is reached, even if data input is increased, performances do not improve. To overcome this limit it is necessary to introduce more powerful techniques based on a Deep Learning approach. In figure 1.2 it is possible to see the difference between Machine Learning and Deep Learning algorithms in terms of performances.

## 1.3 Introduction to Deep Learning

Deep learning is the field of research in Machine Learning that is based on different levels of representation, corresponding to hierarchies of characteristics of factors, where high-level concepts are defined on the basis of low-level ones. Deep learning based techniques have been applied successfully in computer vision, automatic speech recognition, natural language processing, audio recognition and bioinformatics. This chapter introduces the concept of artificial neuron, Neural Network (ANN), Multilayer Perceptron (MLP) and

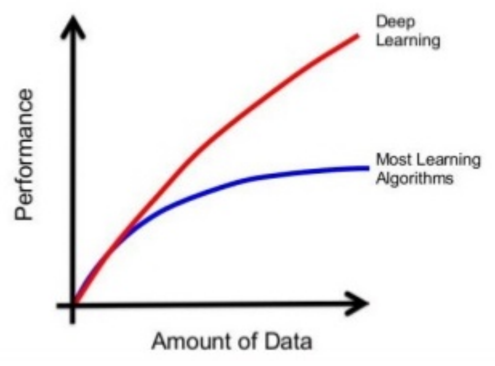


Figure 1.2: Comparing Machine Learning and Deep Learning performances

Convolutional Neural Network (CNN).
The following are just a few examples to give an idea of how Deep Learning is being applied in different contexts [3]:

- Autonomous Guide: researchers in the automotive industry have developed deep learning algorithms for automatic detection of objects, such as stop signs and traffic lights. Furthermore, deep learning is used to detect the presence of pedestrians, helping to reduce the risk of accidents.

- Aerospace and Defense: Deep learning is used to identify objects from satellites that are able to help locate areas of interest and identify safe or unsafe areas for troops.

- Medical Research: researchers use deep learning to automatically detect cancer cells. Some teams at UCLA have built an advanced microscope that produces a large data set used for the training of a deep learning application that can accurately identify cancer cells.

- Industrial Automation: Deep learning helps improve the safety of workers when using heavy machinery by automatically detecting the presence of people and objects at an unsafe distance from the machines.

- Assistive Technologies: Deep learning is used in automatic auditory and vocal translation. For example, home assistance devices with voice recognition and knowledge of user preferences are supported by deep learning applications.

### 1.3.1 Artificial Neuron

Taking inspiration from biology, in 1943 McCulloch and Pitts introduced the notion of artificial neuron, which is shown in figure 1.3.

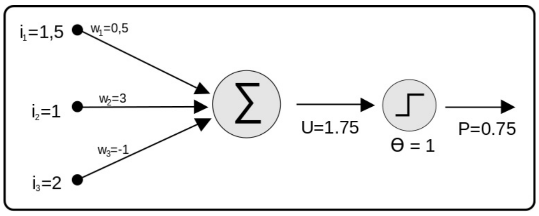


Figure 1.3: Artificial Neuron

An artificial neuron is essentially a function that takes input from other neurons. Each input is associated to a weight. Inside the neuron, which is represented as a circle in the figure above, the dot product (input x weights) is calculated and then an activation function is applied.

The activation function is typically the Standard Logistic Function, or any function that belongs to the Sigmoid family:

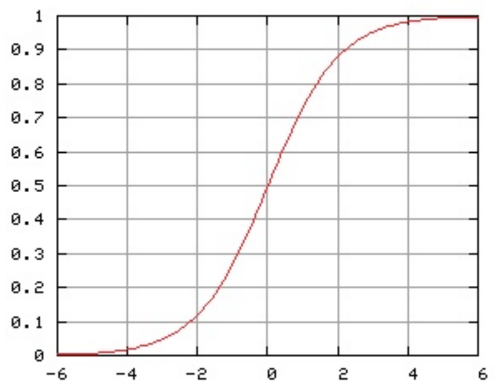


Figure 1.4: Standard Logistic Function

$$f_(x) = \frac{1}{1 + e^{-x_\mathrm{i}}}$$

Where $x_\mathrm{i}$ is the dot product given a neuron at index i:

$$x_\mathrm{i} = \sum w_\mathrm{i} \times p_\mathrm{i}$$

Properties of the Standard Logistic Function:

- Dom: [-$\infty$,$\infty$], Cod: [1,0]
- $\lim_{x\to-\infty} f(x) = 0$, $\lim_{x\to\infty} f(x) = 1$

Another common activation function is the Hyperbolic Tangent, shown in figure 1.5:

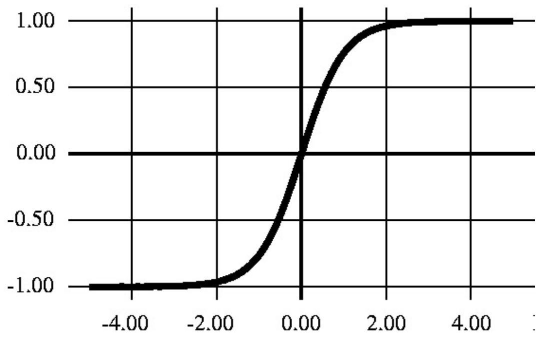


Figure 1.5: Hyperbolic Tangent

$$f(x) = \tanh(x)$$

Properties of the Hyperbolic Tangent:

- Dom: [-$\infty$,$\infty$], Cod: [-1,+1]
- $\lim_{x\to-\infty} f(x) = -1$, $\lim_{x\to\infty} f(x) = 1$

Sigmoid functions are often used in Neural Networks (ANN) to introduce non-linearity into the model and to ensure that certain signals remain within specific ranges. A popular artificial neural element computes the linear combination of the corresponding input signals and applies a sigmoid function limited to the result. This model can be seen as a ”regular” variant of the classical threshold neuron.

### 1.3.2 Artificial Neural Network

An Artificial Neural Network (ANN) is obtained by grouping different artificial neurons into different layers. Figure 1.6 shows an example of ANN.

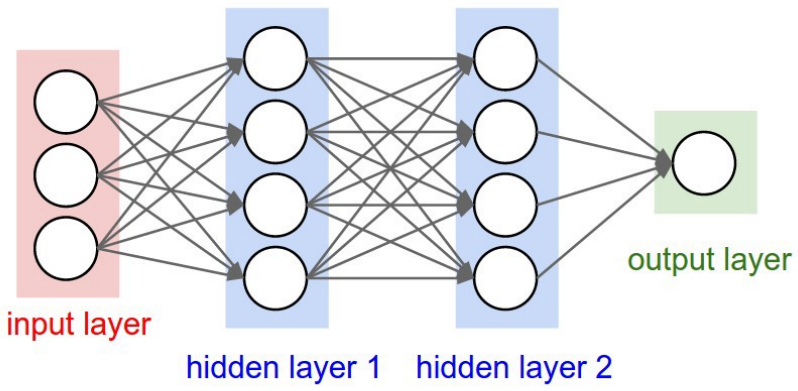


Figure 1.6: Artificial Neural Network

An Artificial Neural Network (ANN) is composed of a series of neurons, represented as circles in the figure, which take as input the weights of the edges coming from the neurons of the previous level. The ANN then calculates the dot product and applies the activation function to the result and it propagates the output to the next level. A network with these characteristics is called *feedforward* because the output always goes towards the next layer. Another feature of the network shown in the figure is that of being *fully connected*: each neuron propagates the result of its computation to each neuron of the following layer. A network that is both feed-forward and fully connected is called Multilayer Perceptron (MLP). A neural network is considered ”deep” when several layers are stacked and each layer contains thousands of neurons. Recent models present between 7 and 50 layers.

The core of the Deep Learning algorithms consists in finding the right weights at the edges of the network so that the sum of the errors of the output neurons is minimized. To achieve this goal it is necessary to establish:

- which function to use in each neuron (activation function).
- which function to use to calculate the sum of errors (loss function).
- which algorithm to use to find the right weights of the network (back-propagation algorithm).

We have already explained the meaning of the activation function, thus no further clarification is needed. Concerning the loss function, one of the most used, especially for regression problems, is the Mean Square Error (MSE):

$$\mathrm{J}(w, x) = \frac{1}{n} \sum t_{\mathrm{c}} - {z_{\mathrm{c}}}^2$$

Where:

- $z$ represents the output vector produced by the network $z = [z_1, .., z_{\mathrm{n}}]$
- $x$ represents the input vector $x = [x_1, .., x_{\mathrm{n}}]$
- $t$ represents the desired output.

The aim of the Deep Learning algorithms is to find parameters (weights) and hyper-parameters to minimize the total MSE. The most used method is the *backpropagation algorithm*, which consists in tracing back the network (from the output level up to the input level) and adjusting the weights, which in principle are initialized randomly. At each step, the new weights are calculated using the *gradient descent*, a well known optimization technique that allows to find local minimum of multi-variable functions, and therefore to minimize the error, according to an additional parameter called *Learning Rate*. This latter parameter determines the speed of convergence of the gradient: if the learning rate is too high, the algorithm converges quickly but risks overshooting, e.g.: not finding the local minimum and continuing to oscillate among more possible solutions. Viceversa, if the learning rate is too small, the gradient descent algorithm risks converging too slowly. Learning is therefore a complex optimization problem because the number of weights involved can be very high. Recent models reach up millions of parameters.

### 1.3.3 Convolutional Neural Network

Artificial Neural Networks, in particular the MLP model, are extremely powerful because, according to the Universal Approximation Theorem, are capable of approximating any function. However, they do not apply well to images. In fact, images are 2D grid structured arrays. Given a few million pixels, the parameters explode. In addition, images have transversely repeated patterns. It is possible to exploit the presence of patterns to give the same weights to the edges and make the problem easier for computers. The same thing happens in the brain: data coming through the retina to the primary visual cortex (v1) pass through neural layers that create hierarchical features. This process is illustrated in figure 1.7.

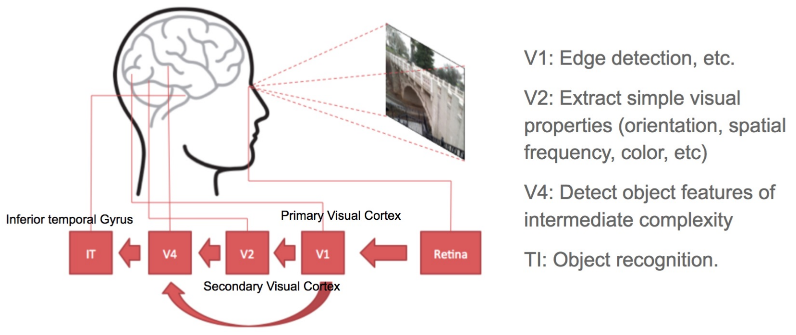

V1: Edge detection, etc.
V2: Extract simple visual properties (orientation, spatial frequency, color, etc)
V4: Detect object features of intermediate complexity
TI: Object recognition.
Inferior temporal Gyrus
Primary Visual Cortex
IT
V4
V2
V1
Retina
Secondary Visual Cortex

Figure 1.7: Visual Cortex System

The intuition stems from an experiment by D.Wiesel and T. Hubel [4], which dates back to 1962, when the two researchers realized that cats, and similarly humans, have two types of cells: simple and complex cells. Simple cells are excited to recognize small and simple patterns, while complex cells aggregate information from simple cells to recognize more generic patterns.

There is a hierarchy of simple and complex cells which is repeated. This repetition creates a hierarchical structure, where initial the layers are able to recognize simple patterns, while complex patterns are distinguished by the last layers.
Convolutional neural networks (CNN) are based exactly on the same hierarchical representation. However, instead of using a fully connected network, a local filter convolution is applied to every area of the input image. The convolution operation is essentially a filter that is passed over every area of the input image of the CNN. This operation, described in figure 1.8, represents exactly the simple cell.

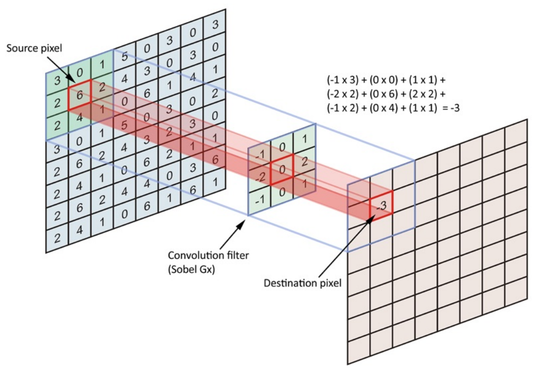

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Figure 1.8: Convolution operation with filter

The complex cell is instead represented by the pooling operation: it takes a set of simple cells and applies an operation that is typically the maximum or the average to aggregate information. The idea is to present a sort of 'summary' to the next level, so we use the maximum or the average. Pooling operation is shown in figure 1.9.

CNNs apply a convolution layer and a pooling layer in series. At the end there is always a fully connected layer, that is the classifier, and it classifies the learned features. The result is a neural network where weights are shared and connections to the next level are local. As a consequence, the number of weights to be found is much lower than an MLP, so the problem, computationally speaking, is easier. The activation function typically used

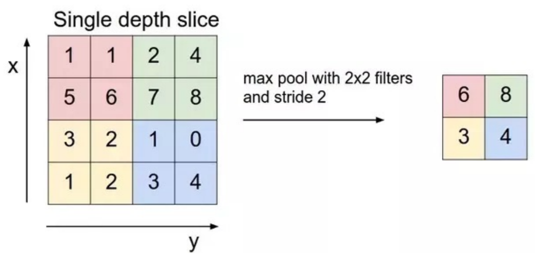


Figure 1.9: Pooling operation using max, 2x2 filter and stride=2

in CNNs is the ReLU (figure 1.10), which is similar to a sigmoid, but has some advantages that favors convergence:

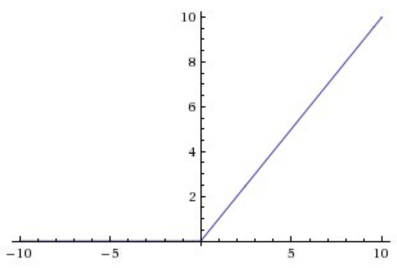


Figure 1.10: ReLU, Rectified Linear Unit

$$f(x) = max(0, x)$$

The ReLU as activation function implies that neurons are activated in a sparse manner. Combining convolution and pooling, CNNs are excellent for image classification problems. Hence, the final architecture for Convolutional Neural Network is shown in figure 1.11:

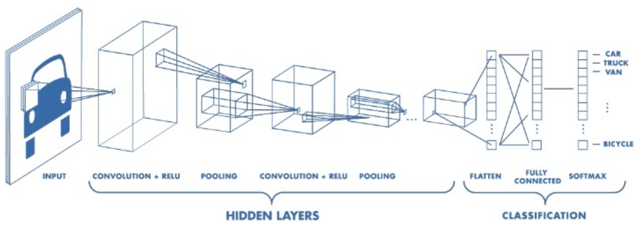


Figure 1.11: Convolutional Neural Network

## 1.4 Training details

During the training phase, hyper-parameters are particularly relevant. This section aims at analyzing the most important parameters, the one that can affect the training phase the most in terms of training time, convergence and accuracy.

### 1.4.1 Stochastic Gradient Discent

Stochastic Gradient Descent (SGD) is one of the most popular optimization algorithms. It is widely used in neural networks as it is the basis of the backpropagation algorithm. The idea of the SGD is to minimize an objective function $J(x)$ formed by N parameters by updating the value of the parameters based on the difference with the negative gradient of $J(x)$. The parameter is then updated step by step, according to a given value LR, called the Learning Rate. In short, we descend a function $J(x)$ step by step, until this leads us to a local minimum value. How long is the step? This corresponds to the Learning Rate value, as shown in figure 1.12.

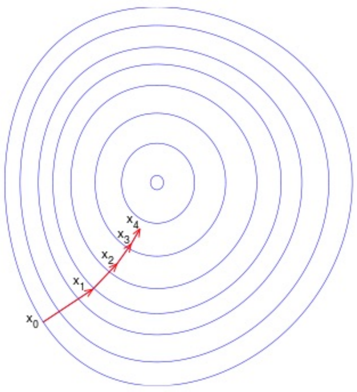


Figure 1.12: SGD and Learning Rate

### 1.4.2 SoftMax as function for the output level

As already explained, it is possible to use both $tan(h)$ or the *Standard Logistic Function* as activation functions on the output layer . In the first case, output values are included between -1 and 1. Using a Standard Logistic Function, output values will be included between 0 and 1. However, in both cases there is no guarantee that the sum on the output neurons is 1, which is a fundamental requirement so that they can be interpreted as a probabilistic distribution.
When a neural network is used as a multi-class classifier, the use of the *Soft-Max* activation function makes it possible to transform the values produced by the last level of the network into class probabilities:

$$softmax(x) = \frac{e^x}{\sum_{1..n} e^x}$$

### 1.4.3 Cross-Entropy as loss function

Using MSE as a loss function is not an optimal choice for classification since the output values do not represent probabilities and the non-imposition of the sum constraint equal to 1 makes learning less effective [5]. Given a multiclass classification problem, the use of Cross-Entropy as a loss function is highly recommended:

$$CE(x) = -\sum p(x) \times log(q(x))$$

Cross-entropy is mostly used to understand the difference between two probability distributions. The target distribution, the one that the model is trying to match, is expressed in terms of a one-hot distribution. How close is the predicted distribution to the real distribution? That is what the cross-entropy loss determines. In the formula above, $p(x)$ stands for the target probability, while $q(x)$ represents the actual probability.

### 1.4.4 Regularization

Regularization techniques can be used to reduce the risk of overfitting by a neural network with many parameters. This trick is very important when the training set is not large compared to the capacity of the model. Neural networks whose weights are close to zero tend to be more stable and this often leads to a better generalization [5]. To force the network to adopt weights of small value, a regularization term to the loss can be added. For instance, in the case of Cross-Entropy Loss:

$$J_{\text{tot}} = CE + J_{\text{reg}}$$

$J_{\text{reg}}$ can be obtained as follows:

- L1: $J_{\text{reg}} = \lambda \sum |w_{\text{i}}|$
- L2: $J_{\text{reg}} = \frac{1}{2} \times \lambda \sum {w_{\text{i}}}^2$

In both cases the $\lambda$ parameter can be set arbitrarily. L1 can have a sparsifying effect (i.e. bring numerous weights to 0) greater than L2. In fact, when the weights assume values close to zero, the calculation of the square in L2 has the effect of excessively reducing the corrections to the weights, making it difficult to reset them.

### 1.4.5 Momentum

While stochastic gradient descent remains a popular optimization strategy, learning with it can sometimes be slow. The method of momentum is designed to accelerate learning, especially in presence of high curvature, small but consistent gradients, or noisy gradients [6]. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The name momentum derives from an analogy with physics, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In figure 1.13 it is possible to see the effect of the momentum, draw in red lines.

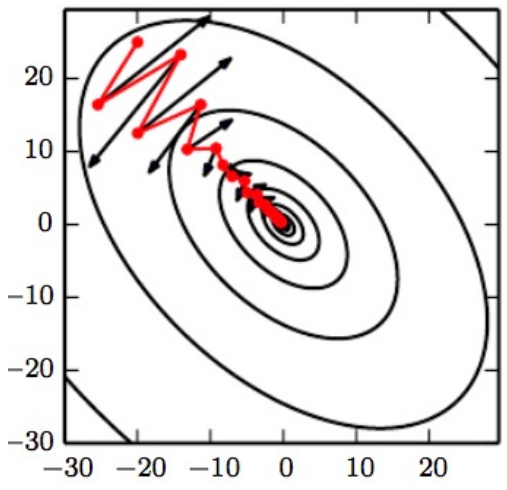


Figure 1.13: The effect of momentum

### 1.4.6 Learning Rate

We said before that learning rate is an hyper parameter that represents how long is the step while moving towards a local minimum. This definition implicitly means that steps must be of the same length. In this scenario, we would probably choose a short step, meaning a low value for the LR, because we want to be sure to find a local minimum and to avoid overshooting, despite the fact that this would be time consuming. However, it is easy to understand that the best choice would be that of taking a ”longer” step when we are far from the minimum, and of progressively decreasing the length of the step as we get closer to the solution. This can be achieved by using predefined learning rate schedules or adaptive learning rate methods.

### Learning Rate Schedules

Learning rate schedules aim at adapting the learning rate during training by planning a predefined schedule. Learning rate schedules methods are: time-based decay, step decay and exponential decay.
Time-Based Decay can be expressed in mathematical form as follows:

$$lr = lr_0/(1 + k_{\mathrm{t}})$$

where $lr$, $k$ are hyperparameters and $t$ is the iteration number. The learning rate is updated by a decreasing factor in each epoch.
Step Decay schedule drops the learning rate by a factor every few epochs. The mathematical form of step decay is :

$$lr = lr_0 \times drop^{floor(\frac{epoch}{epochsdrop})}$$

A typical way is to to drop the learning rate by half every 10 epochs.
Another common schedule is exponential decay:

$$lr = lr_0 \times e^{-kt}$$

where $lr$, $k$ are hyper parameters and $t$ is the iteration number.

### Adaptive Learning Rate

There is still one issue to be discussed concerning the use of the learning rate schedule: hyper-parameters must be defined *a priori* and the same hyper parameters are applied for each update. Sometimes, we may be interested in updating the parameters in different extent. A better way to do this is to use Adaptive gradient descent algorithms: Adadelta, Adagrad, Adam, RMSprop. These methods use a heuristic approach, avoiding extensive fine tuning for the hyper-parameters.

Adagrad adapts the learning rate performing updates: frequently occurring features are associated with smaller updates, while infrequent features are associated with larger updates. Due to this behaviour, Adagrad is well-suited for dealing with sparse data. Previously, an update over all parameters $\Theta$ was performed using the same learning rate $\eta$. Adagrad uses a different learning rate for every parameter $\Theta_\mathrm{i}$ at every time step $t$.

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size [7].

Adam (Adaptive Moment Estimation) calculates adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum [8].

There are still other methods as adaptive learning rate that are not mentioned. At this point, one might be wondering which criteria we must consider for picking up the right learning rate method. The truth is that, despite Adam being considered the state of the art, there is no rule for this task. In fact, training a neural network requires experience, and hyper-parameters tuning depends on the problem, the architecture of the network, the dataset, the domain and other details. Experience in training neural architectures surely plays a key role [9].

# Chapter 2

# Computer Vision

T.S. Huang provides a brilliant definition for Computer Vision [10]. Computer Vision has a dual goal. From the biological science point of view, computer vision aims at providing computational models of the human visual system. From an engineering point of view, computer vision aims at building autonomous systems which could perform some of the tasks which can also be performed by the human visual system (and even surpass it in many cases). Of course, the two goals are intimately related. The properties and characteristics of the human visual system often give inspiration to engineers who are designing computer vision systems. Conversely, computer vision algorithms can offer insights into how the human visual system works.
The tasks of Computer Vision typically are:

- Image Classification: it is the task of assigning a label, from a fixed set of categories, to an input image. This is one of the core problems in Computer Vision that, despite its simplicity, presents a large variety of practical applications.

- Image Classification and Localization: given an image, what we want is the most relevant object and provide a rectangle to identify where the object is.

- Object Detection: given an image, the aim of object detection is to extract relevant objects and their location. This means that object detection is not just a classification task, but also a regression task. In fact, the final output is a label plus a tuple of numbers: $x_0$, $y_0$, width, height.

- Instance Segmentation: it can be considered as a sort of object recognition, with the difference that in this case segmentation is more precise, because it creates an area that overlaps the detected object. The result of image segmentation is a set of segments that collectively cover the entire image, or a set of contours extracted from the image [11].

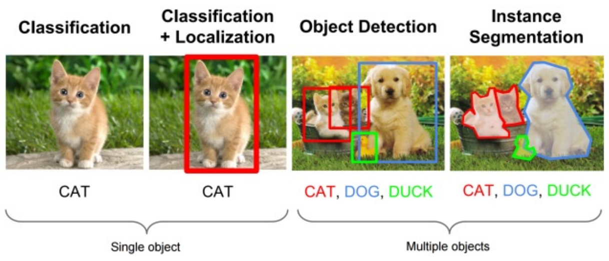


Figure 2.1: Image Classification, Localization, Detection and Segmentation

All these tasks are illustrated in figure 2.1. In this thesis, we will focus mainly on Object Detection.

## 2.1 Deep Architectures for Object Detection

This section will introduce the main architectures that exploit Deep Learning for Object Detection. It should be noted that the architectures proposed are the result of a fairly recent scientific work which is in constant evolution. All these models share the same core: they are all based on Convolutional Neural Networks (CNN).

### 2.1.1 R-CNN

The purpose of the R-CNN (Region CNN) [12] is to identify the main objects given an input image. Therefore, the first step is identifying the regions where the objects could potentially be and subsequently applying CNNs for the identification phase. This phase, called Regions Proposal Phase, is computed through an algorithm called Selective Search [13]. This algorithm considers the image through windows of different sizes, and for each dimension tries to group adjacent pixels by texture, color or intensity, as shown in figure 2.2.

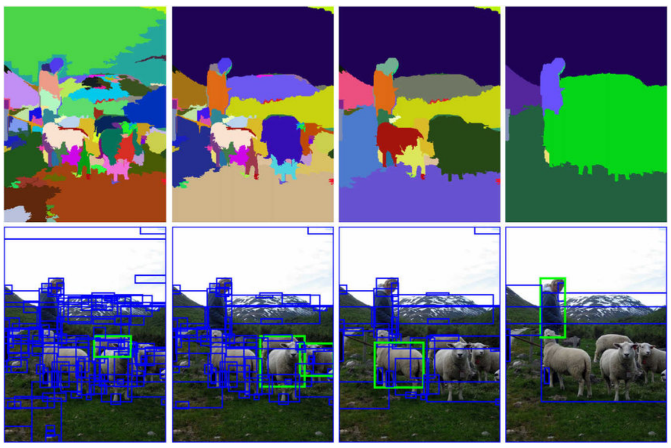


Figure 2.2: Selective Search

The result of this phase are the Regions of Interest, which are areas of the starting image that contain potential objects to be detected. Each Region of Interest is adjusted to ensure that the input of the Region of Interest matches the CNN (warping phase). Each image is processed by CNN, typically AlexNet, and then SVM is applied for the classification task, on the last layer, the fully connected one, while Bounding Box Regression is used for the localization task. The initial regions proposed by the Selective Search algorithm may not be perfectly centered with respect to the subject they contain. Precisely for this reason, the last step of Bounding Box Regression is a further refinement to ensure that the subject is centered in relation to the coordinates of the rectangle in which it is inscribed. Therefore we can consider the R-CNNs just like CNN, where the input is suggested by the Selective Search algorithm and the last layer performs the classification task using SVM. It is evident that the R-CNNs suffer from a critical problem: each Region of Interest becomes the input of a CNN, so the number of CNNs is equal to the number of Regions of Interest, usually a few thousands per image. This makes the problem not suitable for real-time applications, where the detection time must be almost instantaneous. Figure 2.3 shows R-CNN architecture.

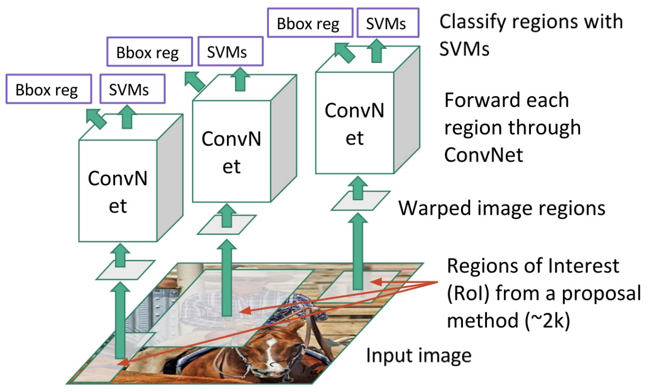

Bbox reg
SVMs
Classify regions with SVMs
Forward each region through ConvNet
ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

Figure 2.3: R-CNN

### 2.1.2 Fast R-CNN

Fast R-CNNs [14] are a further improvement of the R-CNN. The substantial difference consists in placing a RoI Pooling layer between the CNNs and the last fully connected level. While with the R-CNN the components to be trained were CNN, SVM and Bounding Box respectively, the Fast R-CNNs are better because only one component (the RoI Pooling layer) is used for the training phase. The RoI Pooling layer receives as input both the Region Proposals, obtained again by the Selective Search algorithm, and the last layer of CNN. The resulting output are vectors that have the same size that are processed by the fully connected layer. Similarly to the classical R-CNNs, classification and regression are made on the basis of this last level, with the difference that the classification task is no longer assigned to SVM but to the Soft-Max algorithm, which returns a confidence value. The regression part remains unchanged, using the Bounding Box Regressor. Figure 2.4 shows the Fast R-CNN architecture, while figure 2.5 shows just the RoI Pooling Layer.

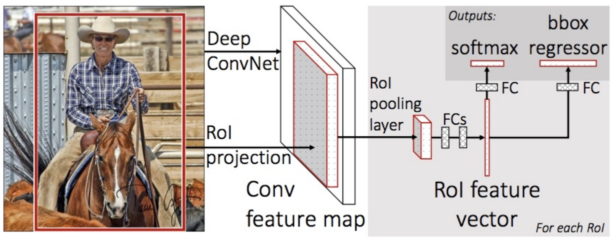


Figure 2.4: Fast R-CNN Architecture

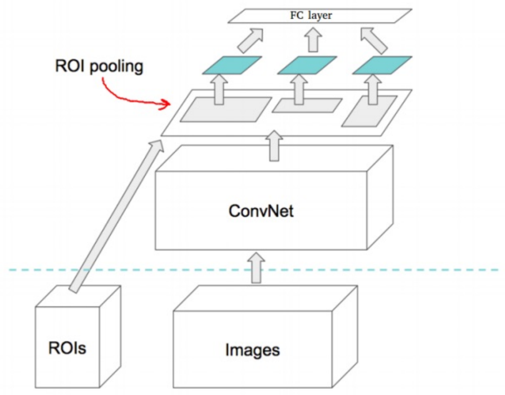


Figure 2.5: RoI Pooling Layer

When analyzing the architecture of a Fast R-CNN network, it is clear that all the inputs and outputs necessary for the classification and localization tasks come from a single network, which therefore proves to be much more efficient than a classic R-CNN, where to each object there corresponds a CNN.

### 2.1.3 Faster R-CNN

Even with all these improvements, a sore point remains in the (Fast) R-CNN: the Region Proposal phase. As we have seen, the first step to discover the position of objects is to generate a lot of potential regions of interest to be tested. The Selective Search algorithm is quite slow and turns out to be the bottleneck of the whole process. The Faster R-CNNs [8] distinguish themselves precisely because they are able to make classification and region proposals thanks to a single CNN. In fact, note that the RoIs strongly depend on the features extracted from the image, which are also calculated by the first levels of CNN. Why not using the same results coming from the CNN for region proposals instead of running the Selective Search separately? This way we obtain the regions of interest almost for free, the only shrewdness is to train the CNN.
Faster R-CNNs consist of two modules:

- RPN (Region Proposal Network): given the input of the convolutional layer, it finds the rectangles for the localization. These rectangles are identified only if there is a relevant subject inside.
- RoI Pooling Layer: classifies each proposal and applies a correction factor so that the subject is centered with respect to the rectangle in which it is inscribed.

The Region Proposal Network (RPN) is definitely the element that makes Faster R-CNN more interesting than the approaches seen so far. Specifically, the RPN uses a sliding window that is scrolled on the feature map and classifies what is below the sliding window as an object/non-object proposing a bounding box in the first case. However, we know that these bounding boxes will have different proportions depending on the object that has been identified. For example, a person will be framed in a rectangle where the height will be significantly greater than the width. Vice versa, a TV monitor will be identified by a rectangle with a 4: 3 aspect ratio. The RPN also deals with finding rectangles with the appropriate proportions. To do so, we propose a series of anchor boxes with associated confidence values. Anchor boxes whose confidence value falls below a certain threshold are discarded, while the rest are passed to the RoI Pooling layer for the classification phase. Figures 2.6 and 2.7 show respectively the Faster R-CNN architecture and the RPN.

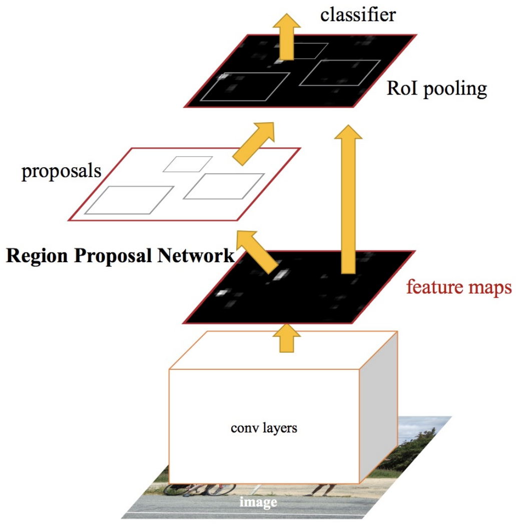


Figure 2.6: Faster R-CNN Architecture

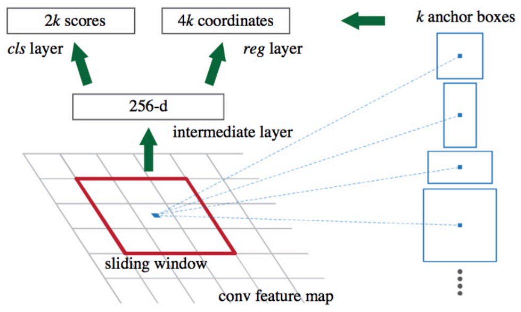


Figure 2.7: Region Proposal Network

### 2.1.4 YOLO

Previous detection systems reuse classifiers or locators to perform detection. They apply the model to an image in multiple positions and scales. Regions with a high image score are considered relevant. YOLO [15] uses a completely different approach. A single neural network is applied to the entire image. In fact, YOLO stands for You Only Look Once. This network divides the image into regions and calculates bounding boxes and probabilities for each of them. This approach has several advantages over classifier-based systems. It makes predictions with a single evaluation of the network, unlike systems like R-CNN that require thousands of evaluations for a single image. How is it possible to do everything with a single neural network? The first step of the processing is to divide the input image into a grid, where each cell is responsible for predicting bounding boxes and their respective confidence values. If the box contains an object, the associated confidence value will be very high, vice versa if it does not contain any object, then the confidence will be very low. This step is shown in figure 2.8.

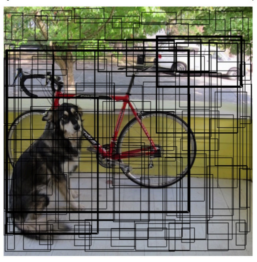


Figure 2.8: YOLO, image divided into bounding boxes

At this point we know exactly how many relevant objects there are in the image, but not what they are. Therefore, each cell also takes care of calculating the probability that for the object to belong to a certain class (class probability), as shown in figure 2.9.

We are talking about conditional probability, which means that the calculated value does not represent the probability to contain that object; it means that if that cell contains an object, then the probability for that ob-

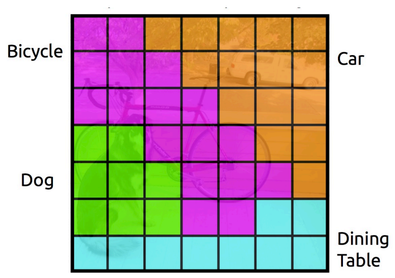


Figure 2.9: YOLO, class probabilities

ject to be a bike, a dog or a car is equal to the calculated value. Subsequently the class probability is multiplied by the conditional probability to obtain a grid where each bounding box considers both the probability of containing this object related to the possibility that there is an object inside. Figure 2.10 illustrates this process.

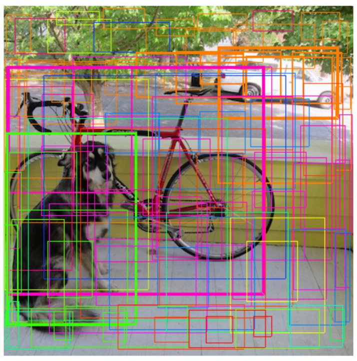


Figure 2.10: YOLO, bounding box and conditional

We then consider only the rectangles whose confidence value is higher than a certain threshold, which is described in figure 2.11.

Despite some generalization errors, YOLO represents the state of the art in object detection algorithms and its speed in detecting objects inside images makes the network particularly adaptable to solve tasks in real time. The architecture of YOLO is summarized in figure 2.12.

The same authors of YOLO have then introduces some improvements in order to enhance the processing time such as anchor boxes generation and a

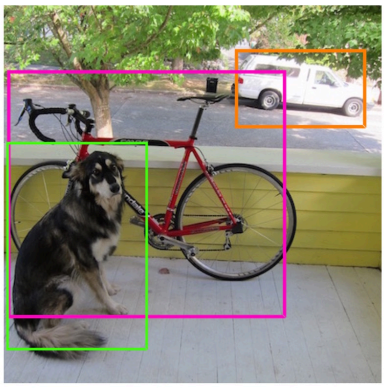


Figure 2.11: YOLO, threshold detection

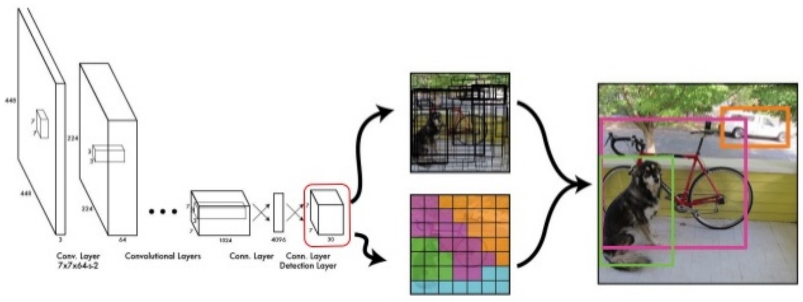


Figure 2.12: YOLO architecture

tree data structure allowing the label sharing between detection and classification, since detection usually has coarse grained label, which corrisponds to the parent node in the tree [16] [17].

### 2.1.5 SSD Multibox

In November 2016, Wei Liu et al.[18] introduced the Single Shot MultiBox Detector. SSD speeds up the processing phase by eliminating the RPN. Starting from a small convolution filters, SSD is able to calculate both location and class score. Then, feature maps are extracted and a convolution filter is applied to each cell. What is new is the multi-scale feature maps: multiple layers are used to detect objects independently. This is done by 6 convolution layers after the VGG-16 layer [19]. In fact, VGG-16 as base network is a brilliant idea due to its strong performances in image classification and its popularity for problems where transfer learning helps in improving results. Thus, a set of convolution layers enables multiple scale feature extractions while decreasing the size of the input layer step by step.

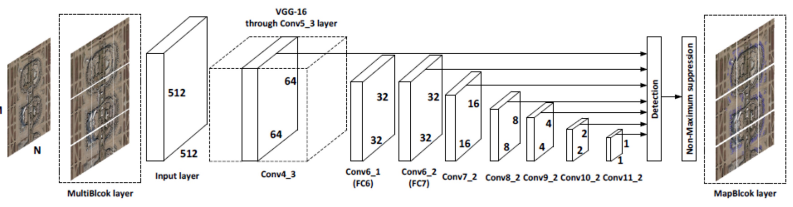


Figure 2.13: SSD architecture

In addition, extra convolutional layers help handling bigger objects, while the non-maxima suppression algorithm is used to filter multiple boxes that may appear. The final architecture is exposed in figure 2.13. The model is quite simple if compared to previous architecture because it discards the proposal generation step. All the processing phase is handled in a single network. Despite SSD could seem similar to YOLO, it is necessary to remark that YOLO uses predefined grid cells, so the aspect ratio is fixed, while SSD allows more aspect ratios (6 in total). Due to this reason, SSD bounding boxes are generally more accurate than the one predicted by YOLO.

### 2.1.6 Performance evaluation

mAP stands for mean Average Precision and it is considered the standard measure for evaluating object detection algorithms by the scientific community. Here we should remember that that object detection means both classification and regression, we need a measure for evaluating both tasks at the same time. Using accuracy as metric would introduce some biases. In fact, a typical data set may contain many classes and their distribution is non-uniform (for example, there might be many more cats than cars). To face this problem, we must introduce the notion of Precision and Recall:

- Precision measures the false positive rate:

$$Precision = \frac{\text{true object detections}}{\text{total number of detections by the system}}$$

A precision value close to 1 means that whatever the classifier predicts as positive is in fact a correct prediction.

- Recall measures the false negative rate:

$$Recall = \frac{\text{true object detections}}{\text{total number of objects in the data set}}$$

A recall value close to 1 means that almost all objects in the dataset will be positively detected.

Precision and Recall are linked by a relationship of inverse proportionality: when one grows, the other decreases and vice versa. For instance, if we want to calculate the AP (Average Precision) for the class 'dog', the Precision/Recall curve is created by varying the threshold that determines what is considered as a model-predicted positive detection of the class, as shown in figure 2.14.

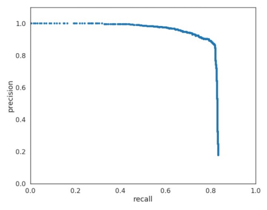


Figure 2.14: Precision-Recall curve for the class dog

Given this curve, AP can be calculate as follows:

$$AP = \frac{1}{n} \sum_{Recall_{\text{i}}} Precision(Recall_{\text{i}})$$

Concerning the localization problem, the most common metric is the Intersection over the Union (IoU). The basic idea is to provide a number scoring how well the ground truth object overlaps the object boundary predicted by the model. IoU can even be explained visually, which makes everything easier to understand. Figures 2.15 and 2.16 explain IoU visually.

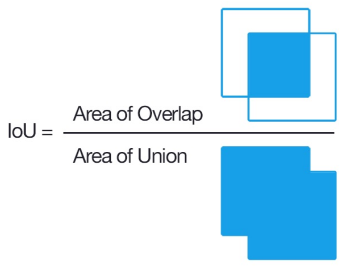


Figure 2.15: Intersection over the Union

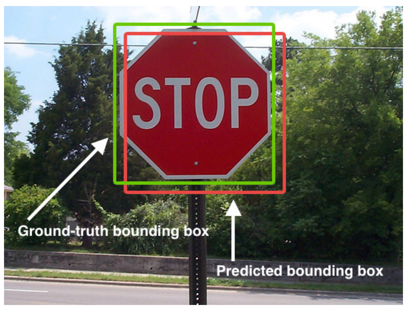


Figure 2.16: IoU stop sign

Again, the IoU threshold is a fundamental parameter and, as previously stated, we can calculate Precision-Recall by varying IoU thresholds.

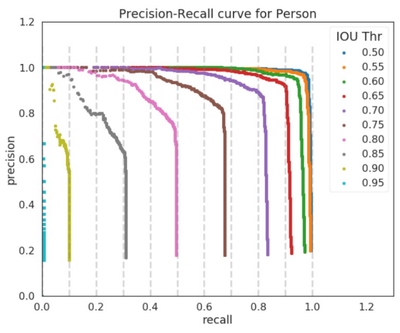


Figure 2.17: Precision/Recall curve for IoU

In figure 2.17 dashes represent spaced recall values where the AP is calculated. Finally, the mean Average Precision or mAP score is equal to the mean AP over all classes and over all IoU thresholds. This way there is only one score that comprehends both classification and localization. However, thresholds may vary depending on the competition or dataset. Right now, we can evaluate all the deep learning architectures for object detection that have been previously explained. Table 2.1 compares different deep architectures for object detection [20].

| | mAP | Speed (FPS) | Speed (s/img) |
|---|---|---|---|
| R-CNN | 62.4 | .05 FPS | 20 s/img |
| Fast R-CNN | 70 | .5 FPS | 2 s/img |
| Faster R-CNN | 78.8 | 7 FPS | 140 ms/img |
| YOLO | 63.7 | 45 FPS | 22 ms/img |
| SSD | 74.3 | 59 FPS | 29 ms/img |

Table 2.1: Object Detection performances, tested on Pascal VOC

It is necessary to specify that the table below has been elaborated to give a summary view of the main characteristics of these networks. To be more precise, each of these neural networks should be evaluated in their versions and updates. In addition, although Pascal VOC is considered the standard by the scientific community, there are also other datasets on which to perform benchmarking such as COCO, ImageNet etc. However, data show that:

- R-CNNs are the worst solution both for accuracy and speed.
- R-CNNs, Fast R-CNNs, Faster R-CNNs clearly show a huge bottleneck due to the selective search algorithm.
- YOLO and SSD are nearly the only two suitable for solving real-time tasks.

In the next chapters we will first introduce Tensorflow for Object Detection and we will then focus on those networks that can solve real-time tasks and, at the same, optimize the use of resources (computing, memory, battery capacity), a typical scenario of embedded systems.

# Chapter 3

# Tensorflow for Object Detection

Tensorflow is the most popular library for Machine Learning. It was initially built for scaling and running over multiple CPUs and now it is even available for mobile operating systems. In Tensorflow models are represented as a dataflow graph that contains a set of nodes described as operations. These are units of computation: they can be simple, as addition or multiplication, but also complicated, such as convolutions. Each operation takes as input a tensor and it provides a new tensor as output. Tensors are just the way data are represented in Tensorflow, they are multidimensional arrays of numbers and they flow among operations. This chapter provides basic notions for building Machine Learning models using Tensorflow, introducing:

- High Level APIs (Keras and Eager Execution)
- Low Level APIs (Tensors, Graphs and Sessions)

The content of this chapter is a small summary of Tensorflow documentation. The aim is to provide a general overview of the concepts that underlie the construction of deep learning models [21].

## 3.1 High Level APIs with Keras

Keras is a high-level API to design deep learning models. It is used for fast prototyping, advanced research and production, with three key advantages:

- User friendly: a simple interface optimized for common use cases. It provides a clear feedback regarding user errors.
- Composable: Keras models are made by connecting configurable building blocks together.
- Easy to extend: writing custom layers, creating new layers, loss functions.

What developers do with Keras is, essentially, composing layers for building models. The basic model is the sequential: tf.keras.Sequential. This is a simple Multilayer Percepetron model:

```
import tensorflow as tf
from tensorflow import keras
model = keras.Sequential()
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

The above mentioned code creates a MLP with 3 layers: 64 neurons for layers 1 and 2 with ReLU as activation function, while the last layer has 10 output neurons using softmax for classification.
We can train and evaluate this model using the 'compile' method with 3 arguments:

- Optimizer: This object specifies the training procedure. We can specify which kind of training policy to adapt, such as AdamOptimizer, RMSPropOptimizer, or GradientDescentOptimizer.

- Loss: The function used to calculate the error during the optimization phase. Common choices include mean square error (MSE) or Cross Entropy.
- Metrics: Used to monitor training. These are string names or callables from the tf.keras.metrics module.

By calling the 'compile' method we can train and evaluate the model:

```
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
             loss='categorical_crossentropy',
             metrics=['accuracy'])
```

Before starting the training phase, we need to set up the dataset. This can be easily done by in-memory Numpy Arrays and using the 'fit' method, that takes 3 arguments: epochs, batch size and validation data.

```
import numpy as np
data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))
model.fit(data, labels, epochs=10, batch_size=32)
```

In case we need bigger datasets or we just need to clean data before processing them, Keras APIs provide all the methods to perform these operations. Lastly, we can evaluate the model and predict new data:

```
model.evaluate(x, y, batch_size=32)
model.evaluate(dataset, steps=30)
model.predict(x, batch_size=32)
model.predict(dataset, steps=30)
```

The final code for a simple fully connected neural network (MLP) is:

```
inputs = keras.Input(shape=(32,))
x = keras.layers.Dense(64, activation='relu')(inputs)
x = keras.layers.Dense(64, activation='relu')(x)
predictions = keras.layers.Dense(10, activation='softmax')(x)
```

```
model = keras.Model(inputs=inputs, outputs=predictions)
model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
       loss='categorical_crossentropy',metrics=['accuracy'])
model.fit(data, labels, batch_size=32, epochs=5)
```

For instance, it is extremely easy to build a MNIST classifier [22] using Keras:

```
import tensorflow as tf
# downloading the mnist dataset
mnist = tf.keras.datasets.mnist
# splitting test and train set
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
# composing the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
# configuring the training parameters
model.compile(optimizer='adam',
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])
# start the training
model.fit(x_train, y_train, epochs=5)
# evaluating the model
model.evaluate(x_test, y_test)
```

## 3.2 High Level APIs with Eager Execution

What makes Tensorflow eager execution different from Keras APIs is the fact that operations are evaluated immediately, with no graph needed. In fact, operations return values without building the entire graph. It can be a good choice for getting started with Tensorflow and for debugging the model. The main advantages are:

- An intuitive interface: structure your code using Python data structures. Quickly iterate on small models and small data.
- Easier debugging: call operations directly to inspect models.
- Natural control flow: use procedural control flow instead of Tensorflow graph.

By enabling tf.enable_eager_execution() TensorFlow operations will return the result immediately:

```
import tensorflow as tf
tf.enable_eager_execution()
x = [[2.]]
m = tf.matmul(x, x)
print(m) #[[4.]]
```

Since there is not a computational graph to build and run later in a session, it is easy to inspect results using 'print()' or a debugger. Evaluating, printing, and checking tensor values does not break the flow for computing gradients [21].
Models can be organized in classes. Here is a model class that creates a two layer neural network that can classify the standard MNIST handwritten digits.

```
class MNISTModel(tfe.Network):
  def __init__(self):
    super(MNISTModel, self).__init__()
    self.layer1 = self.track_layer(tf.layers.Dense(units=10))
    self.layer2 = self.track_layer(tf.layers.Dense(units=10))
  def call(self, input):
    # Actually runs the model
    result = self.layer1(input)
    result = self.layer2(result)
    return result
```

The 'build' method is called the first time the layer is used. It is possible to create variables during __init__() if their full shapes are already known. We use tfe.Network, which is a container for layers. It also contains utilities for inspection, saving and restoring values. Even without training the model, we can imperatively call it and inspect the output:

```
model = MNISTModel()
batch = tf.zeros([1, 1, 784])
print(batch.shape) # (1, 1, 784)
result = model(batch)
print(result)
# tf.Tensor([[[0., ..., 0]]], shape=(1, 1, 10), dtype=float32)
```

To train any model, we define a loss function to optimize, calculate gradients, and use an optimizer to update the variables.

```
def loss_function(model, x, y):
  y_ = model(x)
  return tf.nn.softmax_cross_entropy_with_logits(labels=y,
      logits=y_)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
for (x, y) in tfe.Iterator(dataset):
  grads = tfe.implicit_gradients(loss_function)(model, x, y)
  optimizer.apply_gradients(grads)
```

## 3.3 Low Level APIs: Tensors, Graphs and Sessions

If you really want to know how Tensorflow works at its cor,e then Low Level APIs are needed, which consist of three elements:

- Tensors
- Graphs

- Sessions

The central unit of data in Tensorflow is the tensor. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. Given a tensor, its rank is the number of dimension, and its shape is a tuple of numbers specifying the array lenght for each dimension. Tensorflow uses numpy arrays to represent tensor values. [23]
A computational graph is a series of operations. The graph is composed of two types of objects:

- Operations: these are the nodes of the graph. The input is a tensor and the output is another tensor.
- Tensors: these are the edges in the graph. They are the values that flow through the graph.

A simple computational graph for adding two numbers can be coded as follows:

```
a = tf.constant(3.0)
b = tf.constant(4.0)
total = a + b
print(total) #this will not print 7.0
```

Printing the value of the tensor will not provide the final result. To execute this graph you need to start a session. A session encapsulates the state of the Tensorflow runtime, and runs Tensorflow operations. A tf.Session object is like the python executable:

```
sess = tf.Session()
print(sess.run(total)) # 7.0
```

A graph can be parameterized to accept external inputs, known as placeholders. A placeholder is a promise to provide a value later, like a function argument.

```
x = tf.placeholder(tf.float32)
```

```
y = tf.placeholder(tf.float32)
z = x + y
```

We can use Placeholders for simple experiments, but Datasets are the preferred method of streaming data into a model. To get a runnable tf.Tensor from a Dataset you must first convert it to a tf.data.Iterator, and then call the Iterator's get_next method. Creating an iterator is easy with the make_one_shot_iterator method. For instance, in the code below the next_item tensor will return a row from the my_data array on each run call:

```
my_data = [
    [0, 1,],
    [2, 3,],
    [4, 5,],
    [6, 7,],
]
slices = tf.data.Dataset.from_tensor_slices(my_data)
next_item = slices.make_one_shot_iterator().get_next()
```

We will now see how to built a feedfoward neural network using Low Level APIs:

```
# Define placeholders
X = tf.placeholder(tf.float32, shape=[None, 4])
y = tf.placeholder(tf.int32, shape=[None, 3])
# Define variables
w1 = weight_variable([1], 2])
b1 = bias_variable([2])
w2 = weight_variable([2, 3])
b2 = bias_variable([3])
# Define network
# Hidden layer
z1 = tf.add(tf.matmul(X, w1), b1)
a1 = tf.nn.relu(z1)
# Output layer
```

```
z2 = tf.add(tf.matmul(a1, w2), b2)
y_pred = tf.nn.softmax(z2)
y_one_hot = tf.one_hot(y, 3)
# Define loss function
loss = tf.losses.softmax_cross_entropy(y, y_pred,
    reduction=tf.losses.Reduction.MEAN)
# Define optimizer
optimizer = tf.train.AdamOptimizer(0.01).minimize(loss)
# Metric
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y, axis=1),
    tf.argmax(y_pred, axis=1)), tf.float32))
for _ in range(n_epochs):
    sess.run(optimizer, feed_dict={X: X_train, y: y_train})
```

In this example, variables X_train and y_train contain the whole training set, mini-batches as big as the whole dataset. It is clear that building a MNIST classifier using Low Level API is quite complex. We will not focus on this part, but if you are interested the code is available on Github: https://github.com/cjalmeida/tensorflow-mnist

## 3.4 Tensorflow Object Detection APIs

Designing accurate machine learning models capable of localizing and identifying objects in a single image remains a core challenge in computer vision. The TensorFlow Object Detection API is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. Object Detection is based on low level APIs and has nothing to do with Keras or Eager Execution. This section will provide a general overview about how object detection APIs work, more details will be given in chapter 4.

### 3.4.1 Training a model and export a frozen graph

Object Detection APIs are a set of classes for training custom models for object detection. The good point is that the general architecture does not require to code writing for the training phase. In fact, the training is completely handled by a config file, where the developer can specify:

- Model configuration: this defines what type of model will be trained.
- Train configuration: decides what parameters should be used during the training.
- Evaluation configuration: determines what set of metrics will be reported for evaluation.
- Training input: defines what dataset the model should be trained on.
- Evaluation input: defines what dataset the model will be evaluated on.

Once all these things are defined, we still need a file for the labels and .record files that contain the images for the training and evaluation phases in a Tensorflow own binary storage format. Labels can be expressed in a .pbtxt file according to the following pattern:

```
item {
  id: 1
  name: 'class_name_1'
}
item {
  id: n
  name: 'class_name_n'
}
```

TF record files can be created by simply using using the generate_tfrecord.py class that is included in the Object Detection package. This class needs a .csv as input and it provides a .record file as ouput. After that, it is easy to setup a training pipeline. The library has already some examples available, which may be customized just by changing some parameters. The training phase can be started by running:

```
python legacy/train.py
    --logtostderr
    --train_dir=train_directory
    --pipeline_config_path=CONFIG_FILE
```

Concurrently, we can run the evaluation phase:

```
python legacy/eval.py
    --logtostderr
    --pipeline_config_path=CONFIG_FILE
    --checkpoint_dir=directory_to_save_checkpoints
    --eval_dir=eval_directory
```

The training phase continuously generates checkpoints, which contain the weights of the network, that are evaluated by eval.py. Through Tensorboard it is possible to monitor the whole process. Tensorboard can be started by typing:

```
tensorboard --logdir=train_directory
```

When the model reaches good performances, then it is possible to export it as a frozen graph:

```
python export_inference_graph.py
    --input_type image_tensor
    --pipeline_config_path=CONFIG_FILE
    --trained_checkpoint_prefix=checkpoint
    --output_directory=exported_model_directory
```

A .pb file is written into the exported_model_directory. This file contains the network with weights that ensure certain performances. In fact, during the training phase, weights are held in separate files called checkpoints. When the session is running, graph variables are replaced by values taken from checkpoints. The frozen graph is a single file that contains both the graph and the weights, without the need to use two separate files. Once we have the frozen graph, we can exploit it for out of the box inference.

### 3.4.2 Out of the box inference

Out of the box inference, meaning the exploitation a pre-trained neural network for inference, is pretty easy due to the fact that models are already available online. Normally, developers train their models while monitoring the accuracy. When the accuracy reaches an high peak then the model is exported as a frozen graph. Given a frozen graph, this part can be divided as follows:

- Importing Object Detection libraries
- Loading a frozen graph into memory
- Loading the label map
- Running inference
- Visualizing results

Libraries can be imported this way:

```
from utils import label_map_util
from utils import visualization_utils as vis_util
```

Then we need to load the frozen graph (.pb) into the memory:

```
detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
  with tf.gfile.GFile('frozen_graph.pb', 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')
```

Label maps are important because they match each label with an index:

```
label_map = label_map_util.load_labelmap('my_label_map.pbtxt')
categories =
    label_map_util.convert_label_map_to_categories(label_map,
    max_num_classes='SET_NUM_CLASSES', use_display_name=True)
category_index = label_map_util.create_category_index(categories)
```

Then, we can run inference on a single image:

```
def run_inference_for_single_image(image, graph):
  with graph.as_default():
    with tf.Session() as sess:
      # Get handles to input and output tensors
      ops = tf.get_default_graph().get_operations()
      all_tensor_names = {output.name for op in ops for output in
          op.outputs}
      tensor_dict = {}
      for key in
          ['num_detections','detection_boxes','detection_scores','detection_classes']:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
          tensor_dict[key]=tf.get_default_graph().get_tensor_by_name(tensor_name)
      image_tensor=tf.get_default_graph().get_tensor_by_name('image_tensor:0')
      # Run inference
      output_dict = sess.run(tensor_dict, feed_dict={image_tensor:
          np.expand_dims(image, 0)})
      # all outputs are float32 numpy arrays, so convert types as
          appropriate
      output_dict['num_detections'] =
          int(output_dict['num_detections'][0])
      output_dict['detection_classes'] =
          output_dict['detection_classes'][0].astype(np.uint8)
      output_dict['detection_boxes'] =
          output_dict['detection_boxes'][0]
      output_dict['detection_scores'] =
          output_dict['detection_scores'][0]
  return output_dict
```

At this point, the output_dict object contains everything we need: predicted bounding boxes and the predicted class. It is possible to visualize results by drawing a rectangle on the image:

```
image = Image.open('my_image.jpg')
  image_np = load_image_into_numpy_array(image)
  image_np_expanded = np.expand_dims(image_np, axis=0)
  # Actual detection.
  output_dict = run_inference_for_single_image(image_np,
      detection_graph)
  # Visualization of the results of a detection.
  vis_util.visualize_boxes_and_labels_on_image_array(
      image_np,
      output_dict['detection_boxes'],
      output_dict['detection_classes'],
      output_dict['detection_scores'],
      category_index,
      use_normalized_coordinates=True,
    line_thickness=8)
plt.figure(figsize=IMAGE_SIZE)
plt.imshow(image_np)
```

Figure 3.1 shows the results. All the objects are correctly classified according to a confidence value which is reported at the top right corner label.

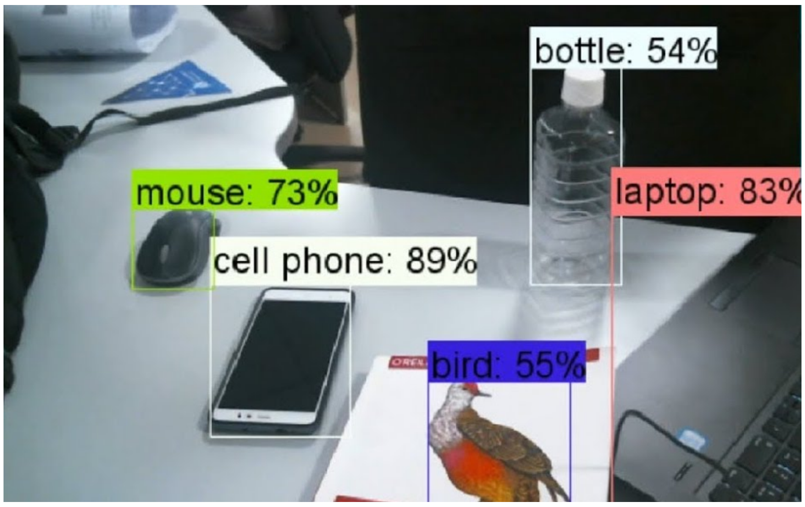


Figure 3.1: Object Detection using Tensorflow APIs

Tensorflow has proven to be an extremely flexible framework, that can be used both for prototyping projects and for a more in-depth use. Object Detection APIs provide a high level of abstraction, enclosing the core of computation in a single configuration file.

# Chapter 4

# Edge AI

Big companies in the ICT sector offer web services to take advantage of the latest Computer Vision technologies. On the one hand, this allows a quick prototyping, but on the other hand, there is a lack of knowledge about what is behind these services. Cloud APIs are easy to use, but in many cases we would like to be cloud independent for the following reasons:

- Connectivity - there might be places where the network is not available.
- Latency - for real time tasks is not suitable to upload an image/video and then get the result back.
- Costs - running inference in the cloud is not free.
- Privacy - some images or videos are personal and we want to make sure no one accesses them.

Edge AI, a term coined by Intel [24], is an enabling technology for the development of ambient intelligence systems: inference runs directly on device and no connectivity is required. This thesis will focus mainly on the Google Vision Kit, but same concepts can be applied to other embedded AI devices such as Amazon Deep Lens, Intel Neural stick etc.

Concerning the pipeline to be adopted while working with these intelligent systems, it is summarized in figure 4.1.

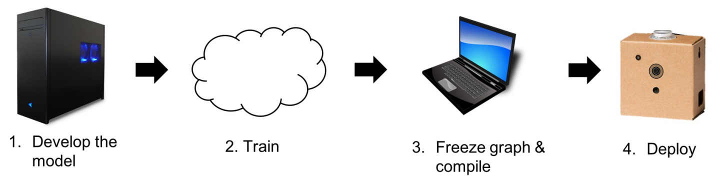


Figure 4.1: Edge AI workflow

The model is developed locally depending on the system's requirements. Then, the model is trained on the cloud. When the model reaches good performances it is 'frozen' and compiled locally. Finally, we move the compiled model to the embedded system. Now, the system is ready to run inference locally.

## 4.1 MobileNets

As soon as Google realized that Edge AI would be a disruptive factor in bringing Artificial Intelligence within everyday products (smart homes, surveillance systems, cars, healthcare image analysis), the company decided to publish a special kind of deep architecture specifically designed for embedded systems, called MobileNets.
A short extract from the original paper [25]:

*MobileNets, a family of mobile-first computer vision models for TensorFlow, designed to effectively maximize accuracy while being mindful of the restricted resources for an on-device or embedded application. MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of a variety of use cases. They can be built upon for classification, detection, embeddings and segmentation similar to how other popular large scale models, such as Inception, are used.*

To sum up, there are two key points introduced in MobileNets:

- Hyperparameters: two simple global hyperparameters efficiently trade off between efficiency and accuracy allowing developers to choose the right model depending on the constraints of their system.

- Depth-wise Separable Convolution: to reduce the computation in the first few layers.

A standard convolution both filters and combines inputs into a new set of outputs in just one step, as shown in figure 4.2. Using the depth-wise separable convolution, these operations are split. Firstly, a single filter is applied to each input channel. Figure 4.3 shows this technique. Later, the pointwise convolution applies a 1x1 convolution to combine the outputs of the depthwise convolution, which is described in figure 4.4. Thus, there is a separate layer for filtering and another layer for combining. This mechanism dramatically reduces computation and model size.

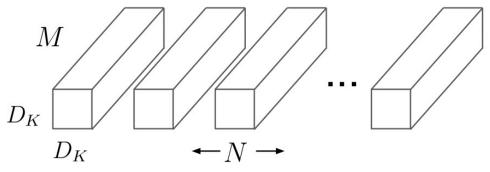


Figure 4.2: Standard Convolution Filter

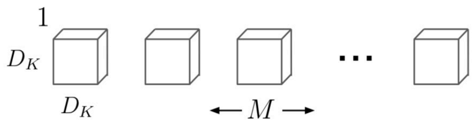


Figure 4.3: Depthwise Convolutional Filter

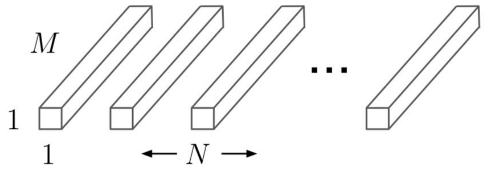


Figure 4.4: Pointwise Convolutional Filter

As already explained, designers can trade-off accuracy and effiency thanks to two hyperparameters: $\alpha$ and $\beta$. $\alpha$ is called width multiplier and its role is that of thinning a network uniformly at each layer. $\alpha$ is ranged between 0 and 1. MobileNet, in its base form, has $\alpha = 1$, while reduced models have $\alpha < 1$. $\beta$ is the resolution multiplier: by applying $\beta$ to the input image, the internal representation of every layer is subsequently reduced by the same value. MobileNet architecture is illustrated in figure 4.5.

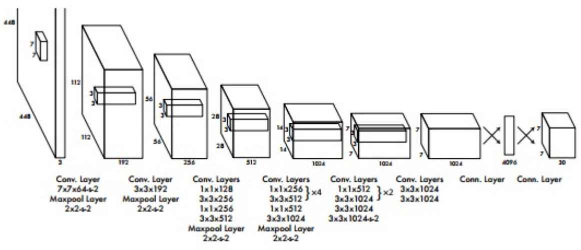
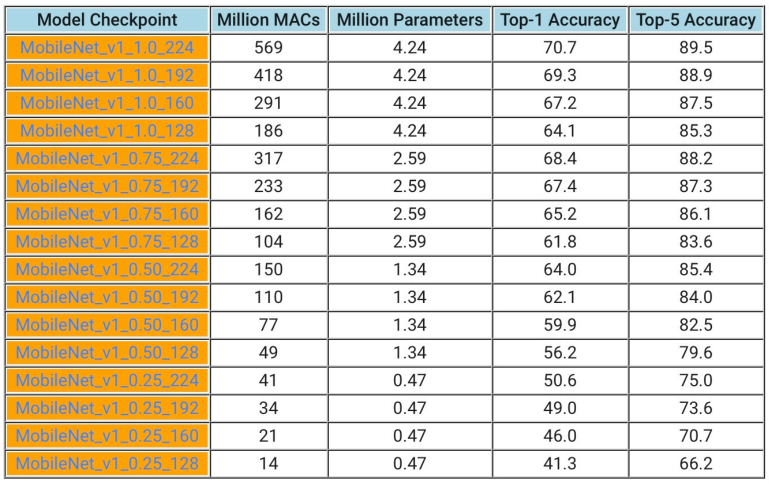

448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 }x4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 }x2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 4.5: MobileNet Architecture

In order to make transfer learning easier, Google released also 16 pre-trained ImageNet classification checkpoints to be used in mobile projects of all sizes. Figure 4.6 shows all the checkpoints available for MobileNets.



| Model Checkpoint | Million MACs | Million Parameters | Top-1 Accuracy | Top-5 Accuracy |
|---|---|---|---|---|
| MobileNet_v1_1.0_224 | 569 | 4.24 | 70.7 | 89.5 |
| MobileNet_v1_1.0_192 | 418 | 4.24 | 69.3 | 88.9 |
| MobileNet_v1_1.0_160 | 291 | 4.24 | 67.2 | 87.5 |
| MobileNet_v1_1.0_128 | 186 | 4.24 | 64.1 | 85.3 |
| MobileNet_v1_0.75_224 | 317 | 2.59 | 68.4 | 88.2 |
| MobileNet_v1_0.75_192 | 233 | 2.59 | 67.4 | 87.3 |
| MobileNet_v1_0.75_160 | 162 | 2.59 | 65.2 | 86.1 |
| MobileNet_v1_0.75_128 | 104 | 2.59 | 61.8 | 83.6 |
| MobileNet_v1_0.50_224 | 150 | 1.34 | 64.0 | 85.4 |
| MobileNet_v1_0.50_192 | 110 | 1.34 | 62.1 | 84.0 |
| MobileNet_v1_0.50_160 | 77 | 1.34 | 59.9 | 82.5 |
| MobileNet_v1_0.50_128 | 49 | 1.34 | 56.2 | 79.6 |
| MobileNet_v1_0.25_224 | 41 | 0.47 | 50.6 | 75.0 |
| MobileNet_v1_0.25_192 | 34 | 0.47 | 49.0 | 73.6 |
| MobileNet_v1_0.25_160 | 21 | 0.47 | 46.0 | 70.7 |
| MobileNet_v1_0.25_128 | 14 | 0.47 | 41.3 | 66.2 |

Figure 4.6: MobileNet Checkpoints

The size of the network in memory and on disk is proportional to the number of parameters. The accuracy and power usage of the network scales with the number of Multiply-Accumulates (MACs), which measures the number of fused Multiplication and Addition operations [25].

## 4.2 A case study: Google AIY Vision Kit

The AIY Vision Kit from Google allows to build your own intelligent camera. Thanks to this kit, it is east develop machine learning and neural network based models. The hardware is very limited: a Raspberry Pi Zero with a special GPU called Vision Bonnet. Right now, the AIY Vision Kit is not available for sale in Europe but it can be sold exclusively in the USA. Figure 4.7 illustrates the new Google Vision Kit.

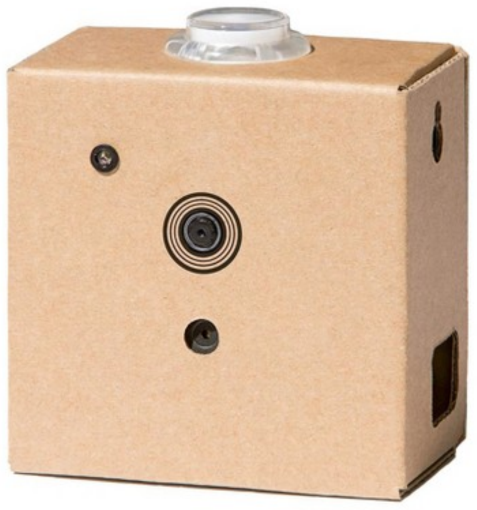


Figure 4.7: Google AIY Vision Kit

### 4.2.1 Vision Bonnet

What makes this kit so special is the presence of the new compact GPU board called Vision Bonnet (figure 4.8). This board is equipped with the Movidius Myriad 2 MA2450 chip, a Vision Processing Unit designed by Intel and intended for machine vision in low-power environments. The Vision bonnet allows the kit to run real-time deep neural networks directly on the device, rather than in the cloud.

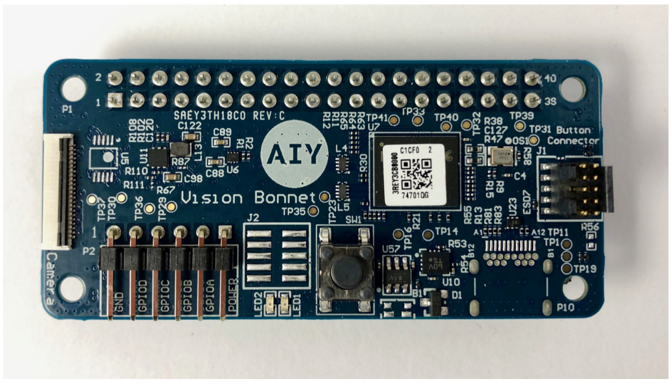


Figure 4.8: Vision Bonnet

The Vision Bonnet reads data directly from the Pi camera through the flex cable, processes them and passes said data to the Raspberry Pi. This approach brings two main benefits:

- while the code is running, the process has complete access to the camera;
- the whole processing phase does not overhead the Raspberry Pi, which is equipped with just 1 GHz ARM single core processor.

The main feature is the Movidius Myriad 2 MA2450 vision processing unit chip (VPU), its architecture is shown in figure 4.9. The VPU presents hardware acceleration that runs neural network graphs at low power. Despite the hardware acceleration, the inference engine has been coded from scratch by Google to enhance performances at runtime.

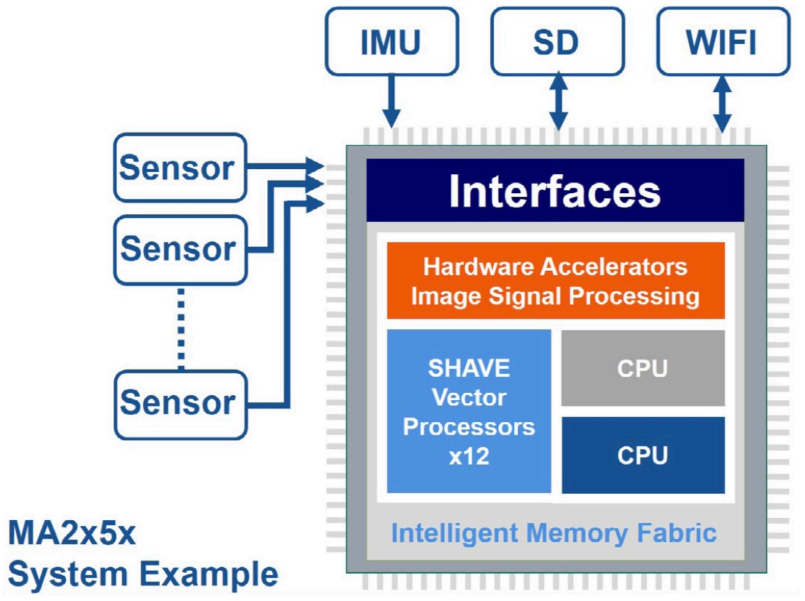

IMU
SD
WIFI
Sensor
Sensor
Sensor
Interfaces
Hardware Accelerators Image Signal Processing
SHAVE Vector Processors x12
CPU
CPU
Intelligent Memory Fabric
MA2x5x System Example

Figure 4.9: Myriad 2 Vision Processing Unit

The HW Accelerators Image Signal Processing part is also known as Vision Engine. It is a set of pre-optimized vision libraries and custom softwares: this way the developer can describe the vision sequence as a directed graph, taking inspiration from the TensorFlow model. Given that, the compiler takes the graph and it distributes the computation among 12 shave cores, vision engine and memory fabric. This enables high performances at low power. Target applications are Deep Neural Network-based Classification, Pose Estimation, 3D Depth, Visual Inertial Odometry (Navigation), Gestures Recognition and Eyes Tracking.

## 4.3 Getting started: AIY demos

The AIY Vision Kit comes with a prebuilt image within Raspbian as operating system, plus some default demos to get started with Computer Vision tasks. The first one is the **Joy Detector** (figure 4.10), which is pretty simple but still surprising. This demo uses machine learning to detect if a person is smiling or frowning. A smile turns the button to yellow, and a frown turns it blue. If expressions are really emphasized, a sound plays. If the camera sees more than one face, it evaluates each person's face and sums the joy score of each face.

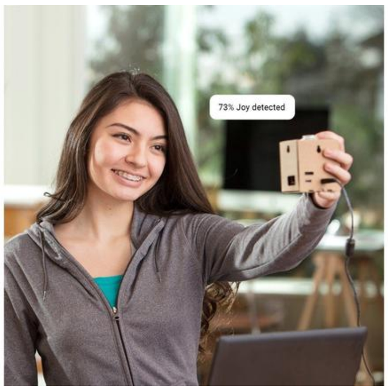


Figure 4.10: Joy Detector demo

The **image classification camera** demo, shown in figure 4.11, is based on Deep Neural Networks. Using the camera you can point at objects randomly and you will see the confidence score appearing on the display, next to the guest object. This model can be run using either MobileNet or SqueezeNet [26]. MobileNet based model, with depth multiplier equal to 1 and 300 pixel as input size, has 59.9% top-1 accuracy on ImageNet, while SqueezeNet based model only 45.3%. The **dog/cat/human detector** is the first demo that uses object detection. It is based on MobileNetV1+SSD, which is a meta-architecture: mixing different parts of image classification architectures for improving their performances. Huang et al.[27] presented a paper which provides an in-depth performance comparison between R-FCN, SSD and Faster R-CNN, finding out that they can be combined with different kinds of feature extractors such as ResNet or Inception. This is the concept of meta-architecture. As the name suggests, this demo is able to draw bounding boxes around dogs, cats and humans given an image.

The **Dish Classifier** demo is designed to identify food in an image. It is based on the MobileNet model, a general-purpose object classification model. It is incredibly accurate and it is even able to distinguish composed food. The

Figure 4.11: Image Classification Camera demo

**Face Detection** demo recognizes faces through the camera in real time and it draws bounding boxes. This is real-time face detection, a core element in Edge AI. Lastly, **Nature Explorer** has 3 machine learning models based on MobileNet, trained on a dataset made of photos uploaded by the iNaturalist community. These models are built to recognize 4,080 different species (960 birds, 1020 insects, 2100 plants). The species and images are a subset of the iNaturalist 2017 Competition dataset, organized by Visipedia.

## 4.4 Custom deploy

After playing a little bit with available demos, we decided to train a model from scratch. The aim of this section is designing a model that can possibly perform real-time inference. We do not only want image classification, our goal is object detection. For sure, our Deep Learning framework will be **Tensorflow**, especially its **Object Detection API** [27]. Developers normally create models capable of detecting their pets: cats, dog, hamsters or even racoons. Since we do not have any pets, we decided to create a "**Pikachu Detector**". Pikachu, illustrated in figure 4.14, is just a sort of little yellow mouse, belonging to the famous Pokemon saga. Truth is that I have a simple Pikachu puppet in my room, so it is easy to test it on the Google Vision Kit. Furthermore, it is even easier to create a dataset with Pikachu labelled images due to its popularity on the web.

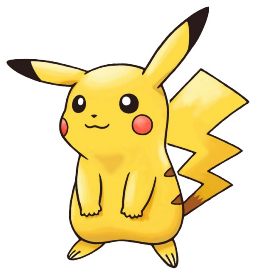


Figure 4.12: Pikachu

Thus, the required steps are:

- Create a small dataset.
- Label each image by hand.
- Convert the dataset into .tfrecords (Tensorflow readable files).
- Create labels in .pbtxt format.
- Create bounding boxes.
- Set TF Object Detection API.
- Create a pipeline for training.
- Run the training to produce a graph.
- Monitor the performances and freeze the graph.
- Export the graph.
- Compile it for the Vision Bonnet.
- Write Python code to make it work with the compiled graph.
- Test the final system.

### 4.4.1 Creating the dataset

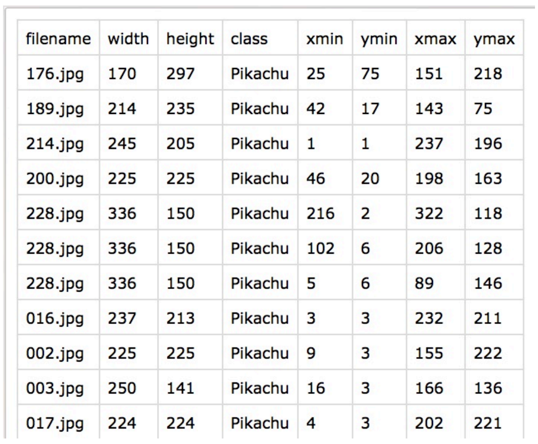
As you can see there are lot of things to do. We shall start by simply installing a Chrome plugin for downloading images in a single batch. This saves a lot of time. The plugin is called "Fatkun Batch Download Images". We will not go into details, because its use is pretty straightforward. A remark: it would be better to download images where the subject is not always in the foreground, but hidden and partially visible sometimes. After that, we must install a program called RectLabel for the labelling phase, which requires to manually label each image. Once this operation is completed, we will see a folder named 'annotations' with many XML files describing the bounding box of each image. The dataset is then split into 70% for the training set and 30% for the test set. Having XML files for bounding boxes and labels is quite useless. In fact, Tensorflow requires .tfrecord files as input. In order to convert these files, we will first convert from xml to .csv and then we will use a class provided by Tensorflow for generating .tfrecords. Concerning the first part, this is the code:

```python
import os
import glob
import pandas as pd
import xml.etree.ElementTree as ET
def xml_to_csv(path):
    xml_list = []
    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        for member in root.findall('object'):
            bndbox = member.find('bndbox')
            value = (root.find('filename').text,
                     int(root.find('size')[0].text),
                     int(root.find('size')[1].text),
                     member.find('name').text,
                          int(bndbox.find('xmin').text),
                          int(bndbox.find('ymin').text),
                          int(bndbox.find('xmax').text),
                          int(bndbox.find('ymax').text)
                          )
                xml_list.append(value)
        column_name = ['filename', 'width', 'height', 'class', 'xmin',
            'ymin', 'xmax', 'ymax']
        xml_df = pd.DataFrame(xml_list, columns=column_name)
        return xml_df

    def main():
        image_path = os.path.join(os.getcwd(), 'annotations')
        xml_df = xml_to_csv(image_path)
        xml_df.to_csv('pikachu_labels.csv', index=None)
        print('Successfully converted xml to csv.')

    main()
```

The result is a .csv with labels and bounding boxes, as shown in figure 4.15.



| filename | width | height | class | xmin | ymin | xmax | ymax |
|---|---|---|---|---|---|---|---|
| 176.jpg | 170 | 297 | Pikachu | 25 | 75 | 151 | 218 |
| 189.jpg | 214 | 235 | Pikachu | 42 | 17 | 143 | 75 |
| 214.jpg | 245 | 205 | Pikachu | 1 | 1 | 237 | 196 |
| 200.jpg | 225 | 225 | Pikachu | 46 | 20 | 198 | 163 |
| 228.jpg | 336 | 150 | Pikachu | 216 | 2 | 322 | 118 |
| 228.jpg | 336 | 150 | Pikachu | 102 | 6 | 206 | 128 |
| 228.jpg | 336 | 150 | Pikachu | 5 | 6 | 89 | 146 |
| 016.jpg | 237 | 213 | Pikachu | 3 | 3 | 232 | 211 |
| 002.jpg | 225 | 225 | Pikachu | 9 | 3 | 155 | 222 |
| 003.jpg | 250 | 141 | Pikachu | 16 | 3 | 166 | 136 |
| 017.jpg | 224 | 224 | Pikachu | 4 | 3 | 202 | 221 |

Figure 4.13: CSV labels and bbox

This operation, meaning the conversion from xml to csv, must be repeated both for the training and the test set. At the end, we will have two .csv files: train.csv, test.csv. At this point we have to put aside these files and install Tensorflow Object Detection API, which is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. The installation is reported on the website and it relies on a lot of dependencies. However, after that, we can exploit a python class for converting .csv files to .tfrecords:

```
> cd models/research/object_detection
> python generate_tfrecord.py
  --csv_input=train.csv
  --output_path=train.record
> python generate_tfrecord.py
  --csv_input=test.csv
  --output_path=test.record
```

TFrecord file format is optimized for being used with Tensorflow in multiple ways. To start with, it makes it easy to combine multiple datasets and integrates with the data import and preprocessing functionality provided by the library. This is suitable for large datasets that cannot be completely stored in memory. Only data required at each batch will be loaded in memory. Another major advantage of TFRecords is that it is possible to store sequence data. For instance, a time series or word encodings in a way that allows very efficient and convenient import of this type of data. After tfrecords are generated, what we need is a label map which consists in a .pbtxt file in the following format:

```
item {
  id: 1
  name: 'Pikachu'
}
```

### 4.4.2 Setting the training pipeline

As already explained in the previous chapter, the complete training process is handled by a config file known as the **pipeline**. Said pipeline is divided into five main structures that are responsible for defining the model, the training and evaluation process parameters, and dataset inputs. The skeleton of the pipeline looks like this:

```
model {
    model config here
}
train_config : {
    train_config here
}
train_input_reader: {
    train_input configuration here
}
eval_config: {
     eval configuration here
}
eval_input_reader: {
     eval_input configuration here
}
```

Concerning the model structure, due to limited hardware resources on Vision Bonnet, there are constraints on what type of models can be run on device. Table 4.1 shows compatible models.

Among all these models, the only one suitable for Object Detection is MobileNetv1+SSD, with input size equal to 256x256 and depth multiplier as 0,125. This specific configuration is called `embedded_ssd_mobilenet`. According to these constraints, the model structure can be set as reported in appendix A.

We must be aware that training the network from scratch is not a good solution. In fact, our dataset is too small and first layers of the network

| MODEL TYPE | SUPPORTED CONFIGURATION |
|---|---|
| MobileNetV1 | input size: 160x160, depth multiplier = 0.5<br>input size: 192x192, depth multiplier = 1.0 |
| MobileNetV1 + SSD | input size: 256x256, depth multiplier = 0.125 |
| SqueezeNet | input size: 160x160, depth multiplier = 0.75 |

Table 4.1: Vision Bonnet constraints

would struggle for learning features. Hence, a good idea is to apply *transfer learning* [28]: we start from a pretrained network and then we train the last layers using our dataset. This can be achieved by setting a proper train configuration:

```
train_config: {
  batch_size: 32
  optimizer {
    rms_prop_optimizer: {
      learning_rate: {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.004
          decay_steps: 800720
          decay_factor: 0.95
        }
      }
      momentum_optimizer_value: 0.9
      decay: 0.9
      epsilon: 1.0
    }
  }
  fine_tune_checkpoint:"training/model.ckpt"
  from_detection_checkpoint: true
  data_augmentation_options {
    random_horizontal_flip {
    }
  }

  data_augmentation_options {
    ssd_random_crop {
    }
  }
```

Please notice that model.ckpt is a checkpoint of the same network trained on the Pascal VOC dataset. This way, during the training phase, weights from this checkpoint are loaded into our network. Training input and evaluation are quite obvious:

```
train_input_reader: {
  tf_record_input_reader {
    input_path: "train.record"
  }

  label_map_path: "label.pbtxt"
}

eval_config: {
  num_examples: 71
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: "test.record"
  }
  label_map_path: "label.pbtxt"
  shuffle: false
  num_readers: 1
}
```

`label_map_path` must point to the .pbtxt file that contains the labels,

while `input_path`, both for train and eval, must point to the tfrecord files we previously generated. `num_examples` set the number of images that are used for the test set.
Finally we have:

- a .pbtxt label file
- tf records both for training and evaluation
- a configuration pipeline

### 4.4.3 Training phase and evaluation

We can start with the training phase. According to the TensorFlow Object Detection APIs, training and evaluation must be run concurrently. The training phase generates some checkpoints that are read by the evaluation phase and then plotted on TensorBoard, an utility that provides useful insights for training deep learning architectures. To start the training it is sufficient to run these commands:

```
python train.py
    --logtostderr
    --train_dir=train
    --pipeline_config_path=myconfig.config
```

Commands for the evaluation are:

```
python eval.py
    --logtostderr
    --pipeline_config_path=myconfig.config
    --checkpoint_dir=train
    --eval_dir=test
```

After that, the training script will start printing the loss value on the terminal, but we must rely on TensorBoard to find out how well the network performs. By monitoring the evaluation, we find that the highest peak is around 20k iterations. So we repeat the same procedure and we evaluate the model using K-Fold Cross Validation (k=5),which can be used when the dataset is too small. This procedure splits the dataset into k non-overlapping subsets of the same size, at every step, the k-th part of the dataset is used as the validation set and the remaining one as the training set. Given k=5, the dataset is divided in 5 parts: P1,P2,P3,P4,P5. Table 4.2 shows all the possible combinations, while table 4.3 reports the results obtained.

| CONFIG | TRAINING | EVAL |
|---|---|---|
| A | P1, P2, P3, P4 | P5 |
| B | P1, P2, P3, P5 | P4 |
| C | P1, P2, P4, P5 | P3 |
| D | P1, P3, P4, P5 | P2 |
| E | P2, P3, P4, P5 | P1 |

Table 4.2: 5-Fold Cross Validation

| CONFIG | embedded_ssd_mobilenet |
|---|---|
| A | 0,5692 |
| B | 0,5873 |
| C | 0,5902 |
| D | 0,5733 |
| E | 0,6084 |
| **avg** | **0,58568 mAP ∼59%** |

Table 4.3: 5-Fold Cross Validation results

Results are clearly not brilliant. This depends mostly on the small dataset of only 350 images. However, they can still be good enough for developing a custom model capable of detecting Pikachu on the Google Vision Kit.

### 4.4.4 Exporting the frozen graph

The model is trained and evaluated. We need to export the Tensorflow frozen graph, compile it and write code for running inference on the Vision Kit. TensorFlow provides all the necessary classes:

```
python export_inference_graph.py
    --input_type=image_tensor
    --pipeline_config_path=myconfig.config
    --trained_checkpoint_prefix=model.ckpt
    --output_directory=exported_model_directory
```

The output is a .pb file located in the output_directory. Later on, the compile phase is possible on a Ubuntu's operating system:

```
./bonnet_model_compiler.par
  --frozen_graph_path=exported_graph.pb
  --output_graph_path=visionkit_graph.binaryproto
  --input_tensor_name=input
  --output_tensor_names=final_result
  --input_tensor_size=input_size
```

### 4.4.5 Coding

What we need now is some Python code to capture data from the Vision Kit camera and to run inference directly on the device. In order to do so, two classes are needed. The first one takes the .protobinary compiled graph and exposes methods that are called by the other class, which is in charge to acquire data from the Pi Camera, apply inference and draw a bounding box as soon as Pikachu is detected.

Firstly, we need to import standard Python libraries plus a subset of AIY libraries, which are not documented but still available within the ISO image.

```
import math
import sys
import os

from aiy.vision.inference import ModelDescriptor
from aiy.vision.models import utils
from aiy.vision.models.object_detection_anchors import
```

Then we create the class, with the costructor and labels:

```
class Object(object):
    BACKGROUND = 0
    PIKACHU = 1
    _LABELS = {
        BACKGROUND: 'BACKGROUND',
        PIKACHU: 'PIKACHU',
    }
    def __init__(self, bounding_box, kind, score):
        self.bounding_box = bounding_box
        self.kind = kind
        self.score = score
        self.label = self._LABELS[self.kind]
```

We must define the model descriptor that reads directly from the .protobinary.

```
def model():
    return ModelDescriptor(
        name='object_detection',
        input_shape=(1, 256, 256, 3),
        input_normalizer=(128.0, 128.0),
        compute_graph=utils.load_compute_graph('graph.protobinary'))
```

Again, we need a method that decodes the bounding boxes and returns a tuple of 4 floats (xmin, ymin, xmax, ymax):

```
def _decode_box_encoding(box_encoding, anchor):
    assert len(box_encoding) == 4
    assert len(anchor) == 4
    y_scale = 10.0
    x_scale = 10.0
    height_scale = 5.0
    width_scale = 5.0

    rel_y_translation = box_encoding[0] / y_scale
    rel_x_translation = box_encoding[1] / x_scale
    rel_height_dilation = box_encoding[2] / height_scale
    rel_width_dilation = box_encoding[3] / width_scale

    anchor_ymin, anchor_xmin, anchor_ymax, anchor_xmax = anchor
    anchor_ycenter = (anchor_ymax + anchor_ymin) / 2
    anchor_xcenter = (anchor_xmax + anchor_xmin) / 2
    anchor_height = anchor_ymax - anchor_ymin
    anchor_width = anchor_xmax - anchor_xmin

    ycenter = anchor_ycenter + anchor_height * rel_y_translation
    xcenter = anchor_xcenter + anchor_width * rel_x_translation
    height = math.exp(rel_height_dilation) * anchor_height
    width = math.exp(rel_width_dilation) * anchor_width
    xmin = _clamp(xcenter - width / 2)
    ymin = _clamp(ycenter - height / 2)
    xmax = _clamp(xcenter + width / 2)
    ymax = _clamp(ycenter + height / 2)
    return (xmin, ymin, xmax, ymax)
```

The output of the neural network must be processed for retrieving objects:

```
def get_objects(result, score_threshold=0.3, offset=(0, 0)):
    assert len(result.tensors) == 2
    logit_scores = tuple(result.tensors['concat_1'].data)
    box_encodings = tuple(result.tensors['concat'].data)

    size = (result.window.width, result.window.height)
    objs = _decode_detection_result(logit_scores, box_encodings,
        ANCHORS,score_threshold, size, offset)
    return _non_maximum_suppression(objs)
```

The network will generate more candidates, so it is necessary to remove some of them that fall below a certain threshold. Also candidates that overlap with existing candidates must be removed.

```
def _non_maximum_suppression(objs,overlap_threshold=0.5):
    objs = sorted(objs, key=lambda x: x.score, reverse=True)
    for i in range(len(objs)):
        if objs[i].score < 0.0:
            continue
        for j in range(i + 1, len(objs)):
            if objs[j].score < 0.0:
                continue
            if _overlap_ratio(objs[i].bounding_box,
                objs[j].bounding_box) > overlap_threshold:
                objs[j].score = -1.0
    return [obj for obj in objs if obj.score >= 0.0]
```

Putting all these pieces together, what we obtain is a library that we called pikachu_object_detection.py. Subsequently, a new file is needed where we capture input from the Pi camera, and then we apply inference using the public methods previously created. Of course, AIY libraries are needed:

```
import argparse
import os
from picamera import PiCamera
from time import time, strftime
from aiy.vision.leds import Leds
from aiy.vision.leds import PrivacyLed
from aiy.toneplayer import TonePlayer
from aiy.vision.inference import CameraInference
from aiy.vision.annotator import Annotator
import pikachu_object_detection
```

Essentially, the CameraInference methods take the model as input. After that, we create an annotator that represents the bounding box. For each detection, the annotator must update its position.

```
def main():
    with PiCamera() as camera, PrivacyLed(Leds()):
        camera.sensor_mode = 4
        camera.resolution = (1640, 1232)
        camera.start_preview(fullscreen=True)
        with CameraInference(pikachu_object_detection.model()) as
            inference:
            last_time = time()
            pics = 0
            save_pic = False
            enable_label = True
            annotator = Annotator(camera, dimensions=(320, 240))
            scale_x = 320 / 1640
            scale_y = 240 / 1232
            def transform(bounding_box):
                x, y, width, height = bounding_box
                return (scale_x * x, scale_y * y, scale_x * (x +
                    width),
                    scale_y * (y + height))
            def leftCorner(bounding_box):
                x, y, width, height = bounding_box
                return (scale_x * x, scale_y * y)
            def truncateFloat(value):
```

```
            return '%.3f'%(value)
        for f, result in enumerate(inference.run()):
            annotator.clear()
            detections =
                enumerate(pikachu_object_detection.get_objects(result,
                0.3))
            for i, obj in detections:
            annotator.bounding_box(transform(obj.bounding_box),
                fill=0)
               if enable_label:
               annotator.text(leftCorner(obj.bounding_box),obj.label
                   + " - " + str(truncateFloat(obj.score)))
                x, y, width, height = obj.bounding_box
            annotator.update()
    camera.stop_preview()
```

However, there is still one thing missing. Our model must be capable of real time detection and for understanding processing time, the following code is needed:

```
def calculate_time():
    now = time()
    duration = (now - last_time)
    annotator.update()
    print("Processing_time:\%s_seconds.Bonnet
        inference_time:_\%s_ms"_\%
    (duration, result.duration_ms))
    last_time = now
```

### 4.4.6 Running the model on the Vision Kit

At this point, we must move to the /home/AIY-projects/src/example/vision directory. The directory structure must be:
- pikachu_detector.binaryproto

- pikachu_object_detection.py
- custom_pikachu_detector.py

and then typing from a shell:

```
./custom_pikachu_detector
```

The camera will start loading the model on the Vision Bonnet. Figures 4.16 show the output. Pikachu is detected and a bounding box is drawn to identify the region where it is located.

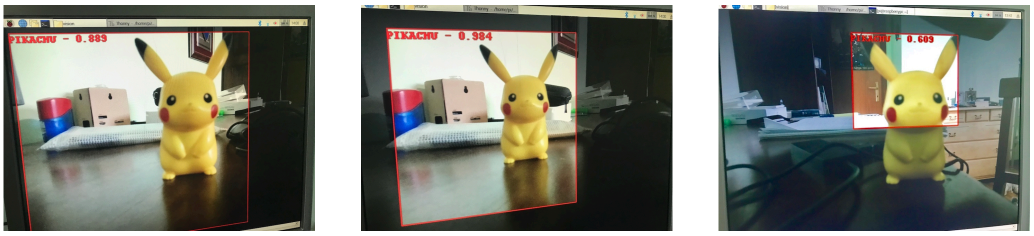


Figure 4.14: Pikachu detector on Google AIY Vision kit

The whole code is available on my Github repository:
https://github.com/giacomobartoli/vision-kit

Processing time is around 20 or 30 ms for each detection, ideal requirement to perform real-time tasks.

### 4.4.7 Running the model as Android app

By Applying more or less the same process, we were able to develop a model even for an Android app. The main differences consist in the fact that the network is pretrained on COCO, while the vision kit model was trained on Pascal VOC, and the network is not an embedded_ssd_mobilenet anymore. We use a **ssd_mobilenet**, where the width multiplier is equal to 1. Perfomances are reported in table 4.4, using 5-Fold Cross Validation:

| CONFIG | ssd_mobilenet |
|---|---|
| A | 0,7916 mAP |
| B | 0,7675 mAP |
| C | 0,7795 mAP |
| D | 0,7982 mA |
| E | 0,7128 mAP |
| **avg** | **0,76992 mAP ~77%** |

Table 4.4: 5-Fold Cross Validation results

It is clear that ssd_mobilenet performs better than embedded_ssd_mobilenet. The main reasons are:

- COCO dataset has 90 classes, while Pascal Voc only 20. For this reason, ssd_mobilenet learns more robust features.
- Using a width mutiplier = 1, makes available 4 times more feature maps per layers with respect to the 0.25 model.

Finally, some visual results are provided in figures 4.17.

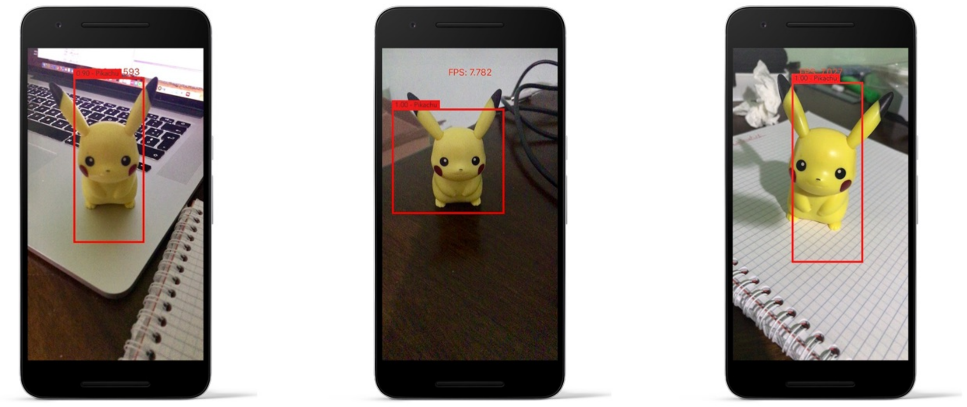


Figure 4.15: Pikachu detector as Android app

# Chapter 5

# CORe50 at the Edge

## 5.1 Introduction

CORe50 [29] is a dataset designed for Continuous Learning. It contains 50 domestic objects belonging to 10 categories. Due to its design, classification can be applied both at an object level or on a category level. There are 11 sessions, 8 indoor and 3 outdoor, for each object. CORe50 has been used exclusively for classification, starting from 128x128 images pre-segmented through a motion algorithm. The aim of this project is to apply detection using CORe50 over original images, in the format 350x350. Bounding boxes are already given and we decided to take advantage of them in order to train a detector. As usual, we will apply transfer learning starting from a pre-trained ssd_mobilenet model on COCO. Being able to distinguish 50 objects is certainly a challenging task. Despite the fact that CORe50 is specifically designed for Continuous Learning, it can be still used as a static dataset. Benchmarks will be reported on the official website: https://vlomonaco.github.io/core50.

## 5.2 Parsing data, setting the TF pipeline

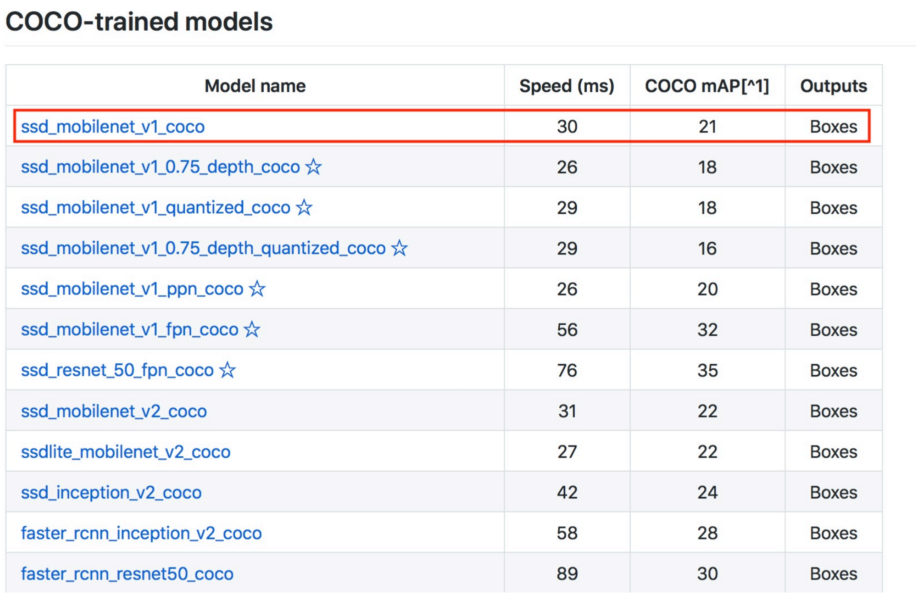
TensorFlow Object Detection requires a valid configuration, a file for mapping the labels and tfrecord files, which contain the images. We need a tfrecord file for the training set and another for the test set. The whole code is available in the appendix, but the procedure can be briefly reported according to these steps:

- Starting from a txt file, we convert the labels in the pbtxt format for Tensorflow (Appendix B).
- We download the entire dataset and we convert images from png to jpeg (Appendix C).
- Given labels, bounding boxes and images, we create two CSV files: one for the training set and another one for the test set (Appendix D).
- CSV files can be easily converted to TF records using TF Object Detection APIs.

Once we have CSV files, we can use Tensorflow Object Detection API for generating tfrecords:

```
python generate_tfrecord.py
    --csv_input=my_csv.csv
    --output_path=core50_train.record
```

Given two tfrecords, training and evaluation, and a .pbtxt for the labels we can setup a tensorflow pipeline by taking a pre-compiled structure of a SSD_mobilenet_v1 from the official website. It is better to start with a pre-trained network that has already learned feature maps on a bigger dataset. For this reason, we can download a pre-trained ssd_mobilenet_v1 on COCO dataset. Checkpoints are available on the Tensorflow website, as shown in figure 5.1.



COCO-trained models

| Model name | Speed (ms) | COCO mAP[^1] | Outputs |
|---|---|---|---|
| ssd_mobilenet_v1_coco | 30 | 21 | Boxes |
| ssd_mobilenet_v1_0.75_depth_coco ☆ | 26 | 18 | Boxes |
| ssd_mobilenet_v1_quantized_coco ☆ | 29 | 18 | Boxes |
| ssd_mobilenet_v1_0.75_depth_quantized_coco ☆ | 29 | 16 | Boxes |
| ssd_mobilenet_v1_ppn_coco ☆ | 26 | 20 | Boxes |
| ssd_mobilenet_v1_fpn_coco ☆ | 56 | 32 | Boxes |
| ssd_resnet_50_fpn_coco ☆ | 76 | 35 | Boxes |
| ssd_mobilenet_v2_coco | 31 | 22 | Boxes |
| ssdlite_mobilenet_v2_coco | 27 | 22 | Boxes |
| ssd_inception_v2_coco | 42 | 24 | Boxes |
| faster_rcnn_inception_v2_coco | 58 | 28 | Boxes |
| faster_rcnn_resnet50_coco | 89 | 30 | Boxes |

Figure 5.1: Screenshot of pre-trained checkpoints

Then, we can setup the training pipeline with the following parameters:

```
num_classes: 50
fine_tune_checkpoint:"downloaded_checkpoint.ckpt"
from_detection_checkpoint: true
label_map_path: "core_50.pbtxt"
train_input_path: "train.record"
train_input_path: "test.record"
```

Again, all these scripts are available on the Github repository: https://github.com/giacomobartoli/vision-kit

## 5.3 Performance evaluation

Monitoring performance trends during training is the best way to understand when a model converges. In fact, performances stabilize around 20k and 30k iterations, with an accuracy up to 66%. The graph taken from Tensorboard is illustrated in figure 5.2, while the loss function is plotted in figure

5.3.

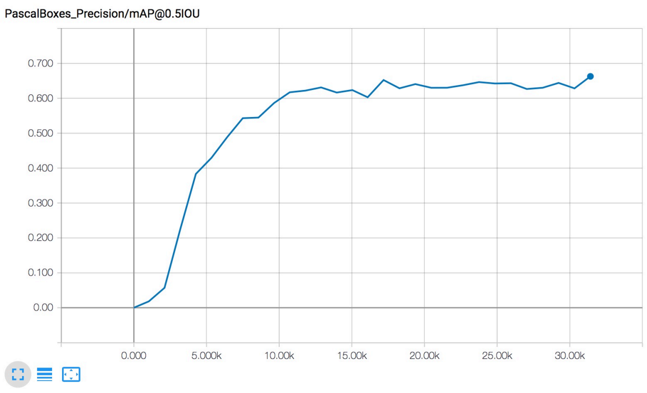


Figure 5.2: SSD_Mobilenet accuracy

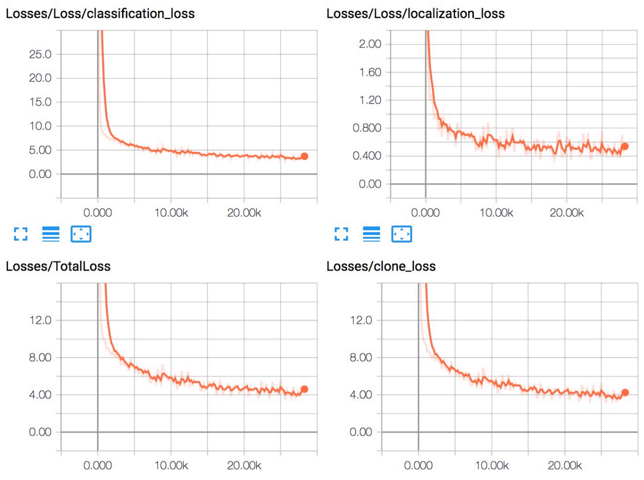


Figure 5.3: SSD_Mobilenet loss

Finally, some visual results. Figure 5.4 shows a light bulb, a ball and a remote control correctly identified.

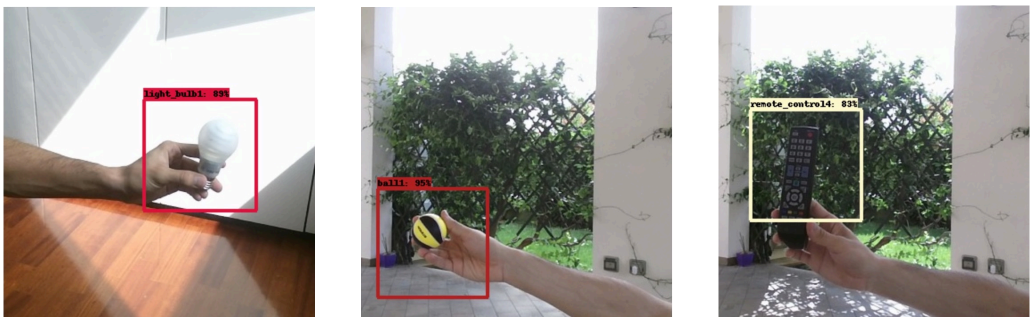


Figure 5.4: CORe50 detection

However, there are still a few mistakes, as reported in figure 5.5, where a can is classified as a mobile phone.

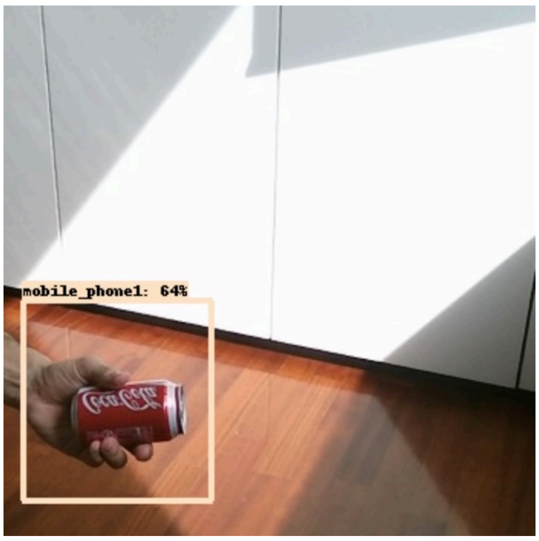


Figure 5.5: SSD_Mobilenet detection

In general, we can say that results are good. We should keep in mind that being able to distinguish among 50 different objects is a challenging task, so 66% is definitely a good performance of the model.

## 5.4 Comparing Mobilenet vs SSD_Mobilenet

SSD_Mobilenet is a meta-architecture for object detection that derives from Mobilenet. We would like to compare performances to understand how mixing parts from different models impact the final performance. CORe50, as static dataset, has already been benchmarked using other deep architectures such as CaffeNet, Nin, GoogleNet and ResNet50. These results are reported in table 5.1.

| model | batch size | accuracy |
|---|---|---|
| CaffeNet | 256 | 74.1% |
| Nin | 256 | 82.26% |
| GoogleNet | 64 | 91.3% |
| ResNet50 | 12 | 92.9% |
| MobileNetV1 | 64 | 91.01% |

Table 5.1: CORe50 benchmarks for classification

However, to compare SSD_Mobilenet and Mobilenet, we need an algorithm to split classification and localization errors. The pseudocode is the following:

```
classification_errors = 0
localization_errors = 0

for each image in test_set:
        BBg #bounding_box ground truth
        class(BBg) # class of BBg
        BBp #predicted_bounding_box
        class(BBp) # class of BBp

        if class(BBg) == class(BBp):
            then classification_errors ++
        else IoU(BBg, BBp) > 0.5 :
```

```
        then localization_errors ++
```

Despite its simplicity, translating this algorithm into Tensorflow is not so trivial. This is due to the fact that the graph must be exported, then we need to run inference for each image in the test set. Moreover, given a test set made of 45'000 images, this algorithm will be very time consuming. The process can be divided into 3 parts:

- a method for running inference on multiple images
- call this method for each image in the dataset
- calculate classification and localization error

The full code is available in appendix E. When running the code, we got 13'591 classification errors, 5'940 of localization. Classification errors make up almost 30% of the test set, so performance of SSD_Mobilenet for accuracy on CORe50 reach up the 70% of the test set. This is close to CaffeNet used in classification mode on 128x128 images: therefore the effectiveness of the proposed detection on original 350x350 images is quite good.

# Chapter 6

# Audi Autonomous Driving Cup

Every year the famous car manufacturing company Audi organizes the Audi Autonomous Driving Cup to test new technologies in the automotive field. This is the first year that the competition is open to teams coming from outside Germany. After a careful selection based on projects submitted from all European universities, only 10 teams were chosen to access the finals. Among these, there is also a team represented by the University of Bologna. The competition involves several challenges, but most of them require object detection for solving tasks such as avoiding pedestrians, recognizing road signs, detecting zebra crossings, allowing emergency vehicles to pass.

## 6.1 The Vehicle

Each team is equipped with one vehicle. The hardware platform, developed by Audi specifically for the contest corresponds to a 1:8 scale replica of the Audi Q2. For the Audi Autonomous Driving Cup 2018, the vehicles were equipped with a miniTX board. This includes:

- an Intel Core i3 processor
- 8 GB RAM
- a fast 128 GB M.2 SSD hard drive
- an NIVIDIA GeForce GTX1050Ti graphics card

In addition to two Gigabit Ethernet ports, the board also has several USB3.0 interfaces and a USB-C port. Furthermore, a Bluetooth and a WLAN module (IEEE 802.11ac) are available. The sensor set of the 2018 AADC model car is getting closer to the real car sensors. A new Laserscanner RPLIDAR A2 is added at the front. The sensor set consists of:

- RPLIDAR A2 (front, $>$ 180 field of view, detection range $<$ 6m, update rate 10Hz)
- 130°mono video camera (front)
- 80°mono video camera (back)
- Ultrasonic sensors (three in the back, one left side, one right side, detection range $<$ 4m, update rate 40Hz)

To measure the speed of the car there are wheel speed sensors. There is also a 9-axis motion tracking sensor to measure accelerations and angular rates.

For development, a tested industry standard environment ADTF (Automotive Data and Time-triggered Framework) is installed. A developer license for the software is available on every vehicle computer so that a convenient development directly on the vehicle is possible. At the front, the car is equipped with a double camera. The first one is the Intel Realsense R200 that can streams at 640x480 or 1920x1080 resolution. The framerate is 15, 30 or 60 FPS. Especially for road sign and lane detection, the car is equipped with an additional front camera that has a 1280x960 resolution at 45 FPS. Both cameras are good enough for real-time scenarios. Figure 6.1 shows the vehicle, the sensors and a general overview of the computer mounted on the car.

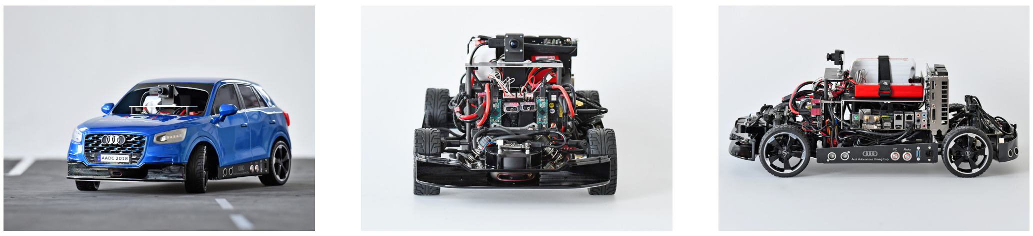


Figure 6.1: AADC vehicle, computer and sensors

## 6.2 The Challenge

Each team is requested to solve a series of tasks that concern the domain of autonomous driving. Tasks are divided into categories:

- Manoeuvre list: parallel parking, cross parking, change lane.
- Sectors: Sectors are specific track sections within the course. They are made up of a list of manoeuvres.
- Basic driving tasks: obstacle avoidance, path navigation.
- Backend communication: the car must send its position continuously to a central server.
- Map data: data concerning the car position can be optionally used to improve the performance of all the driving tasks.

- **Artificial Intelligence-related driving tasks**: distinguish a person from a vehicle, detecting an emergency vehicle and zebra crossing. These are the task we will focus because they can be easily solved by using Deep Learning techniques.

## 6.3 Artificial Intelligence driving tasks

Artificial Intelligence related driving tasks are interesting precisely because it is possible to exploit the deep learning techniques previously illustrated to solve them effectively. These tasks are:

- Zebra crossing
- Crossing task
- Adult versus child
- Yielding to Emergency Vehicles

### 6.3.1 Zebra crossing task

The vehicle must be able to autonomously distinguish between people and other objects. This includes giving way to people crossing the zebra crossing.

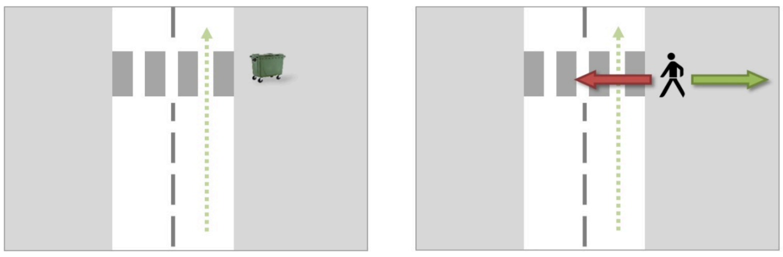


Figure 6.2: Zebra crossing

### 6.3.2 Crossing task

The vehicle must be able to autonomously distinguish between people and vehicles. If a vehicle is detected, right of way must be granted where applicable in accordance with the traffic regulation. If a pedestrian is detected, his/her orientation must be used to determine whether he/she is crossing the road in front of your vehicle, so the vehicle must wait.

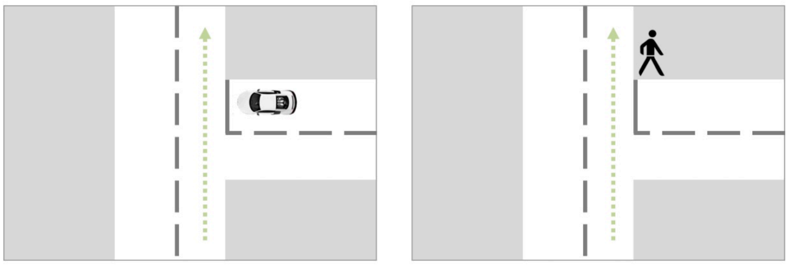


Figure 6.3: Crossing

### 6.3.3 Adult versus child

The vehicle must be able to distinguish between adults and children. If a child is detected, the speed shall be reduced and this shall be indicated by the brake lights. For adults, no actions are required.

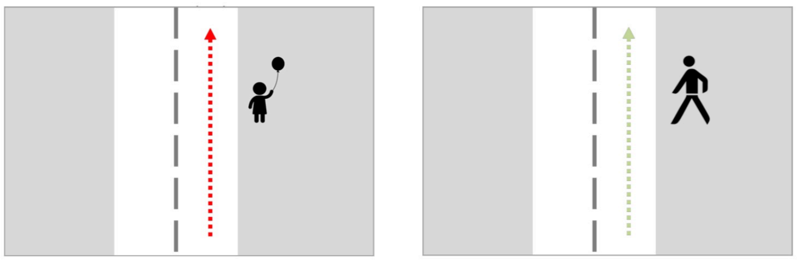


Figure 6.4: Adult versus child

### 6.3.4 Yielding to Emergency Vehicles

The car must yield the right-of-way to any emergency vehicle using a siren and emergency lights. Drive to the right edge of the road and stop until the emergency vehicle has passed.

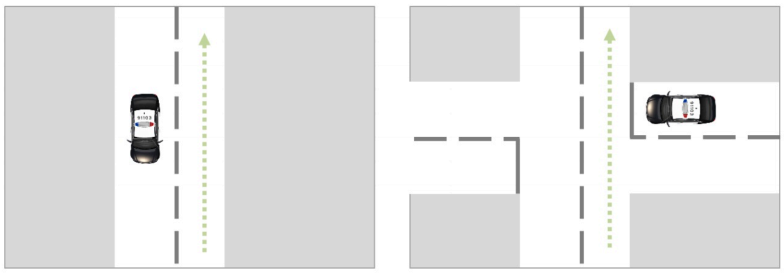


Figure 6.5: Yielding to emergency vehicles

## 6.4 Deep Learning techniques for driving tasks

The graphic card mounted on the car is fully compatible with SSD_Mobilenet, so we decided to create a unique model to detect adults, children, emergency vehicles and normal vehicles. The dataset was created by filming every object and then extracting frames from each video. As total, the dataset is made by 3960 images, 660 of them used as test set and 3300 as training set. In order to avoid biases, each object had almost the same number of images in the whole dataset. Audi already provided dolls for simulating adults and children. The mini car was considered as normal vehicle, while we decided to consider as emergency vehicle only cars that had sirens on it. This means that our model needs only to identify flashing sirens.
Given this, a training pipeline was set up by taking as example a default configuration file. We fine-tuned a pre-trained model on COCO dataset. Each image in the dataset was manually labelled and then labels were exported in Pascal VOC format and converted into TF record files. Starting the training and the evaluation is exactly like repeating what has been done in chapters 4 and 5.

## 6.5 Performance evaluation

After 4k iterations, the algorithm converges with an high accuracy. Figure 6.6 shows the overall accuracy, which reaches almost 90%. Figure 6.7 shows accuracy considering each category as an individual task: adult and car recognition reach almost 100%, while identifying a child performs well in more than 90% of the cases. Results are impressive, except for recognizing emergency sirens, which is up to 60%.

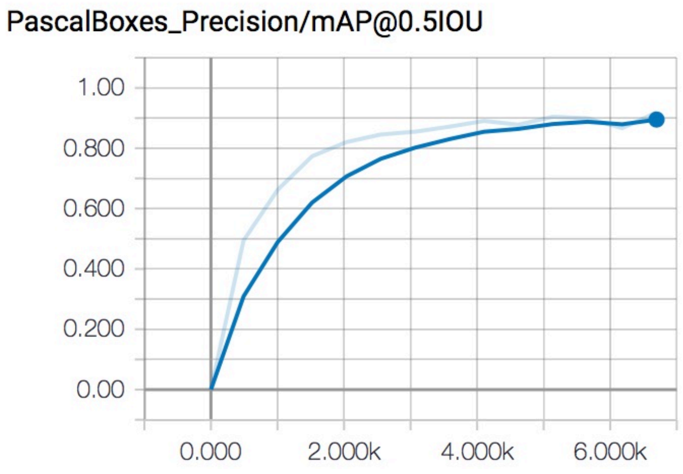


Figure 6.6: Precision

We wrote some code for out of the box inference. This way, we can get inference on images that we want to test. Results are reported in figure 6.8.

I even tried to merge images in a unique image with multiple subjects, to find out if the model was capable to distinguish between a child and an adult, a normal car and an emergency car. As reported in figure 6.9 the model is able to clearly distinguish subjects.

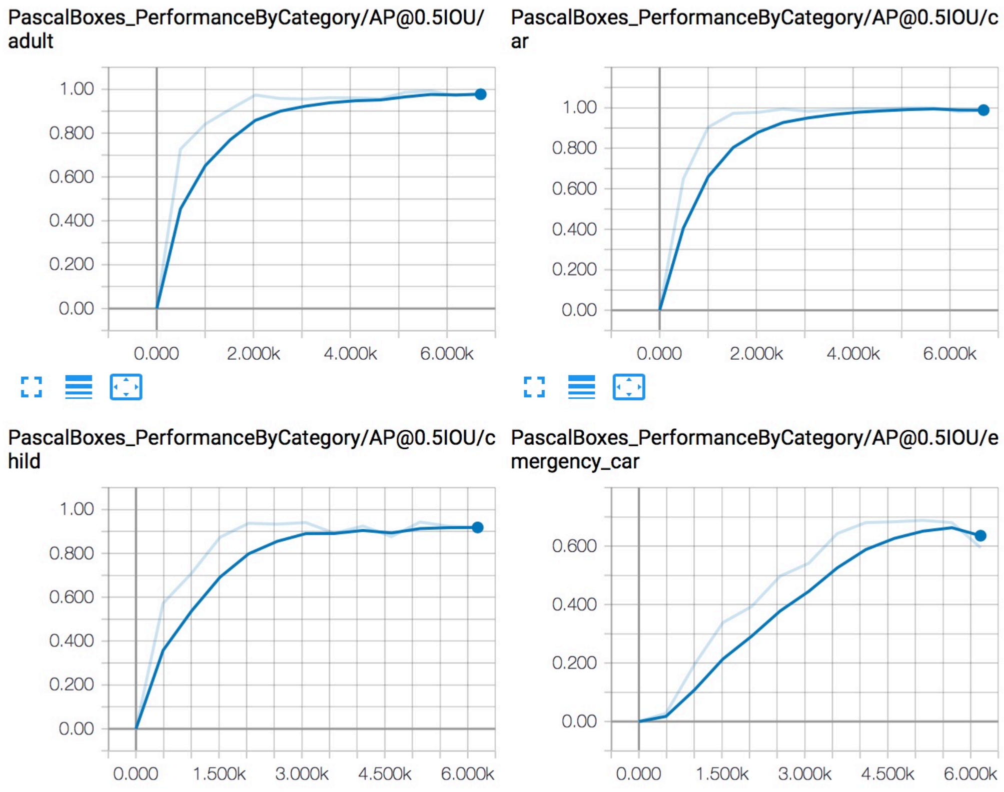


Figure 6.7: Precision per category

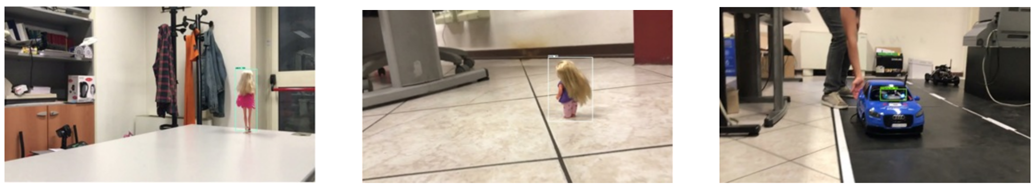


Figure 6.8: Detecting adult, child and emergency car.

## 6.6 Testing the model on the ADTF

Despite results from the test set seem looking excellent, the real test is to deploy the model directly on the car. This way we can understand how well the model behaves in real-world context. This section is divided as follows:

- Test the model to check if inference times are fast enough.

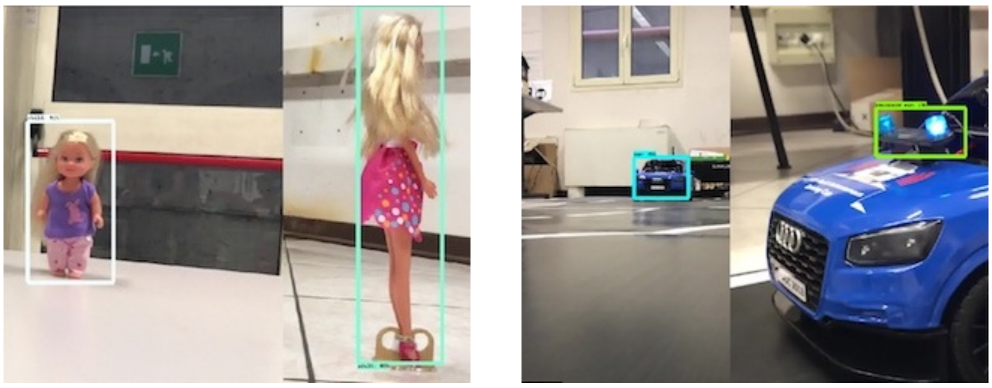


Figure 6.9: Testing multiple detections.

- Deploy the model on the Audi's car.

### 6.6.1 Testing inference speed

The Audi car is equipped with a Linux-based system. Unfortunately, it is not possible run Tensorflow directly on the car. The only way to do inference, given a frozen graph, is to exploit Tensorflow through its C++ APIs. For this reason we cannot reuse the code we wrote before for out of the box inference. Installing C++ APIs for Tensorflow is only possible using a package provided by Audi. Furthermore, this package limits Tensorflow models to the version 1.8. The following code creates a graph, starts a session using the pre-trained model as frozen graph, and feeds this model with a given image. We tested on 5 random images taken from the test. Results show that every image is correctly classified with a bounding box.

```
int main() {
  string image="PATH_TO_IMAGE";
  string graph ="PATH_TO_FROZEN_GRAPH";
  string labels ="PATH_TO_LABELS";
  int32 input_width = 299;
  int32 input_height = 299;
  float input_mean = 0;
  float input_std = 255;
```

```
string input_layer = "image_tensor:0";
vector<string> output_layer ={ "detection_boxes:0",
    "detection_scores:0", "detection_classes:0",
    "num_detections:0" };
bool self_test = false;
string root_dir = "";
// First we load and initialize the model.
std::unique_ptr<tensorflow::Session> session;
string graph_path = tensorflow::io::JoinPath(root_dir, graph);
LOG(ERROR) << "graph_path:" << graph_path;
Status load_graph_status = LoadGraph(graph_path, &session);
if (!load_graph_status.ok()) {
  LOG(ERROR) << "LoadGraph ERROR!!!!"<< load_graph_status;
  return -1;
}
// Get the image from disk as a float array of numbers, resized
    and normalized to the specifications the main graph expects.
std::vector<Tensor> resized_tensors;
string image_path = tensorflow::io::JoinPath(root_dir, image);
Status read_tensor_status =
    ReadTensorFromImageFile(image_path, input_height,
        input_width, input_mean, input_std, &resized_tensors);
if (!read_tensor_status.ok()) {
  LOG(ERROR) << read_tensor_status;
  return -1;
}
const Tensor& resized_tensor = resized_tensors[0];
LOG(ERROR) <<"image shape:" <<
    resized_tensor.shape().DebugString()<< ",len:" <<
    resized_tensors.size() << ",tensor type:"<<
    resized_tensor.dtype();
// Actually run the image through the model.
std::vector<Tensor> outputs;
```

```
 //Starting timer for measuring processing time
 struct timeval start, stop;
 double secs = 0;
 gettimeofday(&start, NULL);
//Inference phase
Status run_status =
    session->Run({{input_layer,resized_tensor}},output_layer, {},
    &outputs);
 //Stopping the timer
 gettimeofday(&stop, NULL);
 secs = (double)(stop.tv_usec - start.tv_usec) / 1000000 +
     (double)(stop.tv_sec - start.tv_sec);
 printf("inference time: %f\n",secs);

 if (!run_status.ok()) {
   LOG(ERROR) << "Running model failed: " << run_status;
   return -1;
 }
 int image_width = resized_tensor.dims();
 int image_height = 0;
 LOG(ERROR) << "size:" << outputs.size() << ",image_width:" <<
     image_width << ",image_height:" << image_height << endl;
 tensorflow::TTypes<float>::Flat scores = outputs[1].flat<float>();
 tensorflow::TTypes<float>::Flat classes =
     outputs[2].flat<float>();
 tensorflow::TTypes<float>::Flat num_detections =
     outputs[3].flat<float>();
 auto boxes = outputs[0].flat_outer_dims<float,3>();
 LOG(ERROR) << "num_detections:" << num_detections(0) << "," <<
     outputs[0].shape().DebugString();
 for(size_t i = 0; i < num_detections(0) && i < 20;++i)
 {
   if(scores(i) > 0.3)
```

```
    {
      LOG(ERROR) << i << ",score:" << scores(i) << ",class:" <<
          classes(i)<< ",box:" << "," << boxes(0,i,0) << "," <<
          boxes(0,i,1) << "," << boxes(0,i,2)<< "," << boxes(0,i,3);
    }
  }
  return 0;
}
```

We fed the model with 5 images taken randomly from the test set. However, we made sure that these 5 images represented all the classes. Table 6.1 reports inference time for each one of those images:

| Image | Processing time |
|---|---|
| 1 | 11,234 ms |
| 2 | 11,265 ms |
| 3 | 11,491 ms |
| 4 | 11,538 ms |
| 5 | 11,332 ms |
| **avg** | **11,372 ms** |

Table 6.1: Audi's car inference time

The results obtained are in line with what is reported into the NVidia whitepaper: GPU-Based Deep Learning Inference, A Performance and Power Analysis [30]. Moreover, 11 ms per image means that our model, during the inference phase, will be able to process almost 90 images per second.

### 6.6.2 Deploy on the Audi car

We can easily reuse code in the previous section to apply inference directly on the car. The difference is that while before we were acquiring images directly from the file system, the ADTF system captures frames from the camera as OpenCV Mat objects. These are essentially matrices made of unsigned integers. Tensorflow instead requires a tensor, which feeds the graph used for inference.

```
Status readTensorFromMat(const Mat &mat, Tensor &outTensor) {
    auto root = tensorflow::Scope::NewRootScope();
    using namespace ::tensorflow::ops;
    // Conversion trick: fill an empty tensor with the opencv
        matrix and feed a TF graph
    float *p = outTensor.flat<float>().data();
    Mat fakeMat(mat.rows, mat.cols, CV_32FC3, p);
    mat.convertTo(fakeMat, CV_32FC3);
    auto input_tensor = Placeholder(root.WithOpName("input"),
        tensorflow::DT_FLOAT);
    vector<pair<string, tensorflow::Tensor>> inputs = {{"input",
        outTensor}};
    auto uint8Caster = Cast(root.WithOpName("uint8_Cast"),
        outTensor, tensorflow::DT_UINT8);
    tensorflow::GraphDef graph;
    TF_RETURN_IF_ERROR(root.ToGraphDef(&graph));
    vector<Tensor> outTensors;
    unique_ptr<tensorflow::Session>
        session(tensorflow::NewSession(tensorflow::SessionOptions()));
    TF_RETURN_IF_ERROR(session->Create(graph));
    TF_RETURN_IF_ERROR(session->Run({inputs}, {"uint8_Cast"}, {},
        &outTensors));
    outTensor = outTensors.at(0);
    return Status::OK();
}
```

After this function, the tensor outputs will contain everything we need for inference: classes, scores, bounding boxes.

```
 // Extract results from the outputs vector
        tensorflow::TTypes<float>::Flat scores =
            outputs[1].flat<float>();
        tensorflow::TTypes<float>::Flat classes =
            outputs[2].flat<float>();
        tensorflow::TTypes<float>::Flat num_detections =
            outputs[3].flat<float>();
        tensorflow::TTypes<float, 3>::Tensor boxes =
            outputs[0].flat_outer_dims<float,3>();
```

Finally, we can iterate over the tensor num_detections and print every every object whose confidence is above a certain threshold.

```
for(size_tensor i = 0; i < num_detections(0);++i){
    if(scores(i) > 0.3) {
      LOG(ERROR) << i << ",score:" << scores(i) << ",class:" <<
          classes(i)<< ",box:" << "," << boxes(0,i,0) << "," <<
          boxes(0,i,1) << "," << boxes(0,i,2)<< "," << boxes(0,i,3);
    }
  }
```

We cannot directly test inference time due to some conflicts with the ADTF libraries, but it is easy to predict that they will not differ so much from what we obtained in section 6.6.1. In that section, we tested only inference while now we should add to that results the time need for loading images in memory.

Some visual results are reported in the video available at this URL: [https://www.vimeo.com/giacomobartoli]

# Conclusions

It is clear that embedded systems empowered by AI will become more significant in the next few years. Given that, we can say for sure that deep models specifically designed for light architectures will be relevant, especially in the field of Computer Vision. This technology can revolutionize entire sectors such as public security, transports, medicine and agriculture. In this direction, deep architectures that can easily trade off effiency and accuracy, as MobileNet, will play a central role. Developers must carefully choose models where inference time is in the order of milliseconds. Choosing the right model is a matter of architectures, purpose, frameworks and hardware compatibility. Despite that, there is still a lack of tools for supporting the creation of the dataset. Video labelling tools, where the user can label objects without repeating this operation for each frame, are still naive. I do believe developers must invest in these softwares to speed up and even automate the dataset creation. However, the pipeline to follow in these cases is the same: creating the dataset, training the model, exporting the model once performances are good enough and applying inference locally. We found out that fine-tuning a pre-trained model is the fastest way to obtain high accuracy. This thesis focused on intelligent cameras, smartphones and autonomous cars but the same concept is valid for drones, unmanned vehicles and any object related to Computer Vision. Finally, we applied those techniques for object detection to the dataset CORe50. Results will be reported in the official website and will be starting points to deepen other Computer Vision tasks such as Image Segmentation and provide other insights about CORe50 as static dataset.

# Appendix A

# embedded_ssd_mobilenet pipeleine

```
model {
  ssd {
    num_classes: 1
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
      }
    }
    matcher {
      argmax_matcher {
        matched_threshold: 0.5
        unmatched_threshold: 0.5
        ignore_thresholds: false
        negatives_lower_than_unmatched: true
        force_match_for_each_row: true
      }
```

```
}
similarity_calculator {
  iou_similarity {
  }
}
anchor_generator {
  ssd_anchor_generator {
    num_layers: 5
    min_scale: 0.2
    max_scale: 0.95
    aspect_ratios: 1.0
    aspect_ratios: 2.0
    aspect_ratios: 0.5
    aspect_ratios: 3.0
    aspect_ratios: 0.3333
  }
}
image_resizer {
  fixed_shape_resizer {
    height: 256
    width: 256
  }
}
box_predictor {
  convolutional_box_predictor {
    min_depth: 0
    max_depth: 0
    num_layers_before_predictor: 0
    use_dropout: false
    dropout_keep_probability: 0.8
    kernel_size: 1
    box_code_size: 4
    apply_sigmoid_to_scores: false
```

```
        conv_hyperparams {
          activation: RELU_6,
          regularizer {
            l2_regularizer {
              weight: 0.00004
            }
          }
          initializer {
            truncated_normal_initializer {
              stddev: 0.03
              mean: 0.0
            }
          }
          batch_norm {
            train: true,
            scale: true,
            center: true,
            decay: 0.9997,
            epsilon: 0.001,
          }
        }
      }
    }
    feature_extractor {
      type: 'embedded_ssd_mobilenet_v1'
      min_depth: 16
      depth_multiplier: 0.125
      conv_hyperparams {
        activation: RELU_6,
        regularizer {
          l2_regularizer {
            weight: 0.00004
          }
        }
        initializer {
          truncated_normal_initializer {
            stddev: 0.03
            mean: 0.0
          }
        }
        batch_norm {
          train: true,
          scale: true,
          center: true,
          decay: 0.9997,
          epsilon: 0.001,
        }
      }
    }
    loss {
      classification_loss {
        weighted_sigmoid {
        }
      }
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      hard_example_miner {
        num_hard_examples: 3000
        iou_threshold: 0.99
        loss_type: CLASSIFICATION
        max_negatives_per_positive: 3
        min_negatives_per_image: 0
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
    normalize_loss_by_num_matches: true
    post_processing {
      batch_non_max_suppression {
        score_threshold: 1e-8
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SIGMOID
    }
  }
}
```

# Appendix B

# Script for converting labels

```
wget https://vlomonaco.github.io/core50/data/core50_class_names.txt

input = 'core50_class_names.txt'
output = 'core_50_labels.pbtxt'

input_file = open(input, 'r').readlines()
output_file = open(output, 'w')
counter = 0
def add_apex(s):
    new_string = "'" + s + "'"
    return new_string
def create_item(s):
    sf='item {\n id:' + str(counter)+' \n name: ' + repr(s) +
        '\n}\n\n'
    return sf
for i in input_file:
    counter+=1
    output_file.writelines(create_item(i.strip()))
output_file.close()
print('done! Your .pbtxt file is ready!')
```

# Appendix C

# Script for converting PNG images to JPEG

```
wget http://bias.csr.unibo.it/maltoni/core50/core50_350x350.zip
unzip core50_350x350
mogrify -format jpg */*.png
find . -type f -iname \*.png -delete
```

# Appendix D

# Script for creating CORe50 csv file

```
import csv
import re
import os
from fnmatch import fnmatch
# header
column_name = ['Filename', 'width', 'height', 'class', 'xmin',
    'ymin', 'xmax', 'ymax']
# CORe50 images width and height
C50_W = 350
C50_H = 350
# CORe50 root: select if training dir or test
root = 'core50_350x350/train'
# root = 'core50_350x350/train'
pattern = "*.jpg"
# Bounding boxes root
bbox = 'bbox/'
pattern_bbox = "*.txt"
# This is an empty list that will be filled with all the data:
    filename, width, height, session..etc
```

```
filenames = []
# some regex used for finding session, obj, frame
re_find_session = '(?<=.{2}).\d'
re_find_object = '(?<=.{5}).\d'
re_find_frame = '(?<=.{8})..\d'

def find_obj(s, regex):
    obj = re.search(regex, s)
    return obj.group()

def find_bbox(session, obj, frame):
    bb_path = 'bbox/'+session+'/'+'CropC_'+obj+'.txt'
    f = open(bb_path, 'r').readlines()
    for line in f:
        regex_temp = 'Color'+frame+': '
        if line.startswith(regex_temp):
            return line[10:]

# c[0] = xmin, c[1] = ymin, c[2] = xmax, c[3] = ymax
def add_bbox_to_list(bbox, list):
    c = bbox.split()
    list.append(c[0])
    list.append(c[1])
    list.append(c[2])
    list.append(c[3])
    return list

# given an object, it returns the label
def add_class_to_list(object):
    index = int(object[1:])
    f = open('core50_class_names.txt', 'r').readlines()
    return f[index-1].strip()
```

```
# scanning the file system, creating a list with all the data
for path, subdirs, files in os.walk(root):
    for name in files:
        if fnmatch(name, pattern):
            listToAppend = []
            listToAppend.append(name)
            listToAppend.append(C50_W)
            listToAppend.append(C50_H)
            session = 's' + find_obj(name,
                re_find_session).strip('0')
            object = 'o' + find_obj(name, re_find_object).strip('0')
            frame = find_obj(name, re_find_frame)
            listToAppend.append(int(object.strip('o')))
            bounding_box = find_bbox(session, object, frame)
            add_bbox_to_list(bounding_box, listToAppend)
            filenames.append(listToAppend)

# writing data to the .csv file
with open('core50_train.csv', 'w') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    filewriter.writerow(column_name),
    for i in filenames:
        filewriter.writerow(i)
print ('Done! Your .csv file is ready!')
```

# Appendix E

# Splitting classification and detection errors

```
def run_inference_for_images(images, graph):
    with graph.as_default():
        with tf.Session() as sess:
            output_dict_array = []
            for image in images:
                # Get handles to input and output tensors
                ops = tf.get_default_graph().get_operations()
                all_tensor_names = {output.name for op in ops for
                    output in op.outputs}
                tensor_dict = {}
                for key in [
                    'num_detections', 'detection_boxes',
                        'detection_scores','detection_classes']:
                    tensor_name = key + ':0'
                    if tensor_name in all_tensor_names:
                        tensor_dict[key] =
                            tf.get_default_graph().get_tensor_by_name(tensor_name)
                image_tensor =
                    tf.get_default_graph().get_tensor_by_name('image_tensor:0')
                image2 = load_image_into_numpy_array(image)
                # Run inference
                output_dict = sess.run(tensor_dict,
                   feed_dict={image_tensor: np.expand_dims(image2,
                   0)})
                # all outputs are float32 numpy arrays, so convert
                   types as appropriate
                output_dict['num_detections'] =
                   str(output_dict['num_detections'][0])
                output_dict['detection_classes'] =
                   output_dict['detection_classes'][0].astype(np.uint8)
                output_dict['detection_boxes'] =
                   output_dict['detection_boxes'][0]
                output_dict['detection_scores'] =
                   output_dict['detection_scores'][0]
                output_dict_array.append(output_dict)
                global tot_images
                tot_images = tot_images - 1
                print('Remaining images: ' + str(tot_images))

    return output_dict_array
```

Then we must call the previous method, sorting the images:

```
# LOADING TEST SET, SORTED ALPHABETICALLY
images = sorted(glob.glob("images/test/*.jpg"))
tot_images=len(images)
print("Test set dimension: "+str(tot_images)+" images")

# ARRAY WITH DETECTION RESULTS
output_dict_array = run_inference_for_images(images,
   detection_graph)

#loading GT labels/bbox from .txt
```

```
ground_truth_list_labels =
    open('ground_truth_labels.txt').read().splitlines()
ground_truth_list_bbox =
    open('ground_truth_bbox.txt').read().splitlines()

# computing class_errors and localization errors
index = -1
for f, b in zip(ground_truth_list_labels, predicted_labels):
    index +=1
    print(str(index))
    if f != b:
        class_errs = class_errs+1
    else:
        boxA = predicted_bbox[index].split(',')
        boxB = ground_truth_list_bbox[index].split(',')
        if bb_intersection_over_union(boxA, boxB) > 0.5:
            localiz_errs = localiz_errs + 1
print('CLASSIFICATION: '+str(class_errs))
print('LOCALIZATION: '+str(localiz_errs))
```

# Bibliography

[1] Rolf Pfeifer and Christian Scheier. *Understanding Intelligence.* Cambridge, MA, USA: MIT Press, 2001. ISBN: 026266125X.

[2] Tom M. Mitchell. *Machine Learning.* New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[3] MathWorks. *Perche' il deep learning e' cosi' importante.* `https://it.mathworks.com/discovery/deep-learning.html`. [Online; accessed 16-August-2018]. 2017.

[4] D. H. Hubel and T. N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of Physiology* 160.1 (), pp. 106–154. DOI: `10.1113/jphysiol.1962.sp006837`. eprint: `https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1962.sp006837`. URL: `https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1962.sp006837`.

[5] Davide Maltoni. *Reti Neurali, materiale didattico.* `http://bias.csr.unibo.it/maltoni/ml/DispensePDF/8_ML_RetiNeurali.pdf`. [Online; accessed 17-August-2018]. 2017.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[7] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: `1212.5701`. URL: `http://arxiv.org/abs/1212.5701`.

[8] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: `1412.6980`. URL: `http://arxiv.org/abs/1412.6980`.

[9] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-49430-0. DOI: `10.1007/3-540-49430-8_2`. URL: `https://doi.org/10.1007/3-540-49430-8_2`.

[10] T. S. Huang. *Computer Vision: Evolution and Promise*. `https://cds.cern.ch/record/400313/files/p21.pdf`. [Online; accessed 19-August-2018]. 1996.

[11] Wikipedia. *Image Segmentation*. [Online; controllata il 20-agosto-2018]. 2011. URL: `https://en.wikipedia.org/wiki/Image_segmentation`.

[12] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013).

[13] J. R. R. Uijlings et al. "Selective Search for Object Recognition". In: *International Journal of Computer Vision* 104.2 (2013), pp. 154–171. URL: `https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013`.

[14] Ross Girshick. "Fast R-CNN". In: *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*. ICCV '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1440–1448. ISBN: 978-1-4673-8391-2. DOI: `10.1109/ICCV.2015.169`. URL: `http://dx.doi.org/10.1109/ICCV.2015.169`.

[15] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *CVPR*. IEEE Computer Society, 2016, pp. 779–788.

[16] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *CVPR*. IEEE Computer Society, 2017, pp. 6517–6525.

[17] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: *CoRR* abs/1804.02767 (2018).

[18] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *CoRR* abs/1512.02325 (2015).

[19] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014).

[20] Arthur Ouaknine. *Review of Deep Learning Algorithms for Object Detection.* [Online; controllata il 3-agosto-2018]. 2018. URL: `https://medium.com/comet-app/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852`.

[21] Google. *Tensorflow documentation.* [Online; controllata il 4-settembre-2018]. 2017. URL: `https://www.tensorflow.org/guide/keras/`.

[22] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: `http://yann.lecun.com/exdb/mnist/`.

[23] Google. *Tensorflow documentation Low Level API.* [Online; controllata il 5-settembre-2018]. 2017. URL: `https://www.tensorflow.org/guide/low_level_intro`.

[24] Intel. *Artificial Intelligence at the Edge.* [Online; controllata il 20-agosto-2018]. 2017. URL: `https://ai.intel.com/artificial-intelligence-at-the-edge/`.

[25] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017).

[26] Forrest N. Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡1MB model size”. In: *CoRR* abs/1602.07360 (2016).

[27] Jonathan Huang et al. “Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 3296–3297.

[28] Hossein Azizpour et al. “From Generic to Specific Deep Representations for Visual Recognition”. In: *CoRR* abs/1406.5774 (2014).

[29] Vincenzo Lomonaco and Davide Maltoni. “CORe50: a New Dataset and Benchmark for Continuous Object Recognition”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. Ed. by Sergey Levine, Vincent Vanhoucke, and Ken Goldberg. Vol. 78. Proceedings of Machine Learning Research. PMLR, 13–15 Nov 2017, pp. 17–26. URL: http://proceedings.mlr.press/v78/lomonaco17a.html.

[30] Nvidia. *GPU-Based Deep Learning Inference: A Performance and Power Analysis*. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf. [Online; accessed 22-September-2018]. 2015.

# Ringraziamenti

L'elaborato in questione è il frutto di un intenso lavoro di circa 7 mesi. Vorrei ringraziare in primis il Prof. Davide Maltoni che ha mostrato massima disponibilità ad accogliermi come tesista, mi ha fornito materiale su cui lavorare, studiare e discutere insieme. Duranto lo svolgimento è stato presente e puntuale nel risolvere ogni mio dubbio. Non avrei potuto chiedere di meglio. Un ringraziamento è d'obbligo anche al dott. Lomonaco Vincenzo, che mi ha fornito supporto tecnico durante tutta l'estate, festivi compresi. Il suo entusiasmo e la sua passione sono contagiosi.

Ci tengo particolarmente a ringraziare, in maniera più generale, tutti i professori che in questi 5 anni hanno contribuito alla mia formazione. Su tutti, vorrei esplicitamente menzionare i prof. Omicini Andrea, Ricci Alessandro, D'Angelo Gabriele e Roli Andrea. Le loro lezioni non sono state mai noiose, ma sempre stimolanti. Un grazie anche al Prof. Panagiotis Papapetrou, docente di Data Mining presso la Stockholm University durante il mio periodo come studente di scambio.

Un grazie ad i miei amici di sempre, per avermi distratto in momenti in cui ero decisamente troppo preso dallo studio.

Un grazie ad i miei amici "internazionali", chi ho avuto modo di conoscere prima in Spagna e chi dopo in Svezia, per avermi fatto capire che certe amicizie vanno oltre le distanze.

Un grazie ad i miei compagni di studio. Solamente loro conoscono il "disagio" di certi momenti, lo sfascio di certi progetti e la tensione di certi esami.

Un grazie ad i miei piccoli cuginetti per le innumerevoli partite a carte che abbiamo fatto quest'estate in giardino. Le risate non si contano. Sappiate che "Giacomon" sarà sempre disponibile per giocare con voi, ma appena sarete più grandi smetterò di farvi vincere come al solito. Giacomon non perde mai.

Un grazie ad i miei ex colleghi del Gruppo Sit. Ho veramente dei bellissimi ricordi dei 9 mesi passati insieme.

Infine, vorrei dedicare questo traguardo ad i miei genitori. E' merito loro se ho avuto l'opportunità di studiare, viaggiare e crescere.
Questo traguardo è prima di tutto il loro.

Giacomo
18 Ottobre 2018