POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

A deep learning approach to estimate population density of urban areas to compute risk maps for UASs

Supervisors

Prof. Alessandro RIZZO

Dott. Stefano PRIMATESTA

Candidate

Filippo BOLLETTA

July 2021

Summary

Unmanned Aircraft Systems (UASes) have gained interest in recent years in many different fields. UASes are used for remote sensing, surveillance, precision agriculture and package delivery, to name a few. The growing interest in their use leads to safety concerns which can be faced using a systematic approach to define safe operations. The proposed approach consists in the definition of risk maps that quantify the risk connected with a flight operation over a certain area. Starting from these maps, a path planning algorithm that finds the trajectory that minimizes the risk for the UAS can be developed. Risk is measured as the hourly probability of having a fatality due to a drone crash. To correctly compute its numerical value it is necessary to have maps describing the area of interest: the value of the risk is determined starting from population density, sheltering factor, no-fly zones and obstacles. The main goal of this thesis is to obtain population density maps. Knowing how people are distributed in a given area is essential to understand how dangerous the impact of a vehicle on the ground could be. Generally speaking, sparsely populated areas will be considered safer than highly populated ones to fly over. In the final part of the thesis, an approach based on the direct estimate of the risk with the same layers structure used for the creation of population density maps is proposed.

In this thesis, a deep learning-based approach for the creation of population density maps is adopted. Some of the most promising results in this field have been obtained in the last years using Convolutional Neural Networks (CNNs). A CNN is a machine learning model in which specific layers are employed, with the goal of learning features from databases composed of images. The pre-trained VGG16 network has been employed as feature extractor, on the top of which dense layers were added to reduce the output dimension. Techniques like fine tuning and test-time augmentation have been employed to increase the precision of the maps obtained. The deep learning model was implemented using Tensorflow and Keras, which are libraries that provide ready-to-use tools to train and evaluate machine learning models. The model was trained in an Amazon Web Service (AWS) EC2 instance with a dedicated GPU, that is designed for deep learning applications.

The CNN was trained on the area of Turin (city centre and surrounding area),

and satellite imagery were downloaded from Bing Maps REST Services. After the evaluation of the network, an analysis has been performed looking at the most and less precise estimates. This process is useful to understand the limits of the model used and the problems of the database (for example, limited resolution and perspective of the images).

The thesis is divided into 5 chapters. A brief introduction is provided in chapter 1. In chapter 2 a state of the art analysis is carried out to understand the methods used by researchers to handle this problem. In chapter 4 the software used and the hardware on which the network was trained are described. The development flow of the thesis is presented in details in chapter 5: all the steps taken are resumed and the related results are listed. Conclusions are drawn in chapter 6, where also some suggestions for the future development of this work are proposed.

Table of Contents

List of Figures

Acronyms

AI
artificial intelligence

API
appllication programming interface

AWS
Amazon Web Services

CNN
convolutional neural network

CPU
central processing unit

FTP
file transfer protocol

GPU
graphics processing unit

ML
machine learning

NN
neural network

SSH
secure shell protocol

TPU

tensor processing unit

TTA

test time augmentation

UAS

unmanned aerial system

UAV

unmanned aerial vehicle

XML

extensible markup language

Chapter 1

Introduction

The goal of this thesis is to continue and enhance the work done by the research group about the adoption of a risk-aware path planning strategy for UAVs in urban environments. In particular, the possibility of estimating population density starting from satellite imagery of an area is investigated. To do so, a machine learning approach is employed. The interest in building high-definition population density maps is motivated by its great practical significance: these maps are used all over the world to optimize resource allocation and disaster response, to understand many economical, environmental and social problems, and to take risk-based decisions.

The safety of UAV operations is an open issue that aviation agencies like ENAC (Ente Nazionale per l'Aviazione Civile) and FAA (Federal Aviation Administration) are facing with specific rules. Most of the operations with UAVs must be performed in Visual Line-Of-Sight (VLOS), while operations Beyond Visual Line-Of-Sight (BVLOS) are strictly limitated. A complete legislation about this topic has been proposed by EASA (European Aviation Safety Agency), which adopts a risk-based approach.

Risk maps are cell-based maps that quantify the risk of flying over an area. The value of the risk is computed taking into account several layers: population density, sheltering factor, no-fly zones and obstacles (figure 1.1). These maps are fundamental for the path planning algorithm, because they provide the search space, that is, the set of trajectories among which the algorithm can choose, along with the associated value of the risk.

As previously mentioned, this thesis is based on a project of the research group to compute safe routes for UAVs in urban areas. This project uses risk maps to estimate the risk to the population on ground due to a UAV flight. Risk maps are matrices of size $n \times m$, where each entry of the matrix refers to a cell. The dimension of the cells defines the resolution of the risk map, so it is a fundamental parameter to choose. On one hand, high resolution maps are preferable, since they

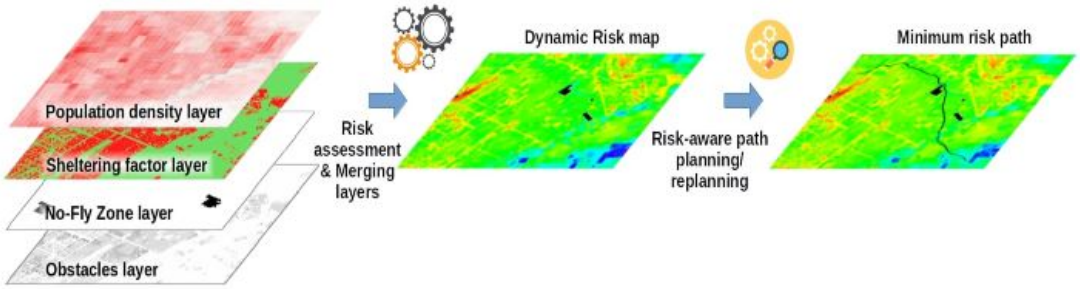

Figure 1.1: Workflow

provide a more precise description of the area; on the other hand, to obtain reliable population density and sheltering factor maps it is necessary to use cells that are not too small, in order to make possible for the CNN to learn general features from satellite imagery. Each cell has an associated value of the risk expressed in casualties per hour, a common measure unit used in aviation.

The risk is defined using a probabilistic risk assessment approach: its value is proportional to the probability of having a fatal accident per hour of flight. Three conditional events are required to happen in order to have a fatality: the loss of control of the vehicle, its impact with a person and the injury resulting fatal for that person, so the formula is ([1]):

$$P_{casualty}(p_{i,j}) = P_{crashing} \times P_{impact}(p_{i,j}) \times P_{fatality}(p_{i,j}) \tag{1.1}$$

$p_{i,j}$ refers to a cell of the grid map. The term that depends on the population density is $P_{impact}(p_{i,j})$, that is defined as the product of the area occupied by the crash of the drone (also called lethal area) and the population density in that area:

$$P_{impact}(p_{i,j}) = \rho(p_{i,j}) \times A_{crash} \tag{1.2}$$

$P_{fatality}(p_{i,j})$, instead, depends on the sheltering factor, that takes into account the presence of obstacles that can absorb the kinetic energy of the UAV, reducing the probability of fatalities. $P_{crashing}$ represents the probability that the UAV loses control, and the descending behavior can be of different types (parachute, ballistic, uncontrolled or fly-away), and the risk changes depending on the type.

Then, a risk-aware path planning algorithm computes the minimum risk path in the risk map. The optimal path is computed off-line using a path planning algorithm, such as riskA* or riskRRT*. These algorithms find the best trade-off between the risk and the flight duration, given as inputs the risk maps, the initial point and the target point.

In recent years, the development of convolutional neural networks in the field of image recognition and analysis has encouraged the use of machine learning in a

context where traditionally census data were the only available source. Census data are reliable, but they have many problems associated: the process of gathering them is long and expensive, and for this reason they are not updated frequently, especially in poor countries. For example, the national census in countries like Italy, China and USA is carried out every 10 years, while there are six countries which have been without census since 1990([2]): Afghanistan (1979), Eritrea (1984), Lebanon (1932), Somalia (1985), Uzbekistan (1989), and the area of Western Sahara (1970).

The increasing availability in the last decade of high-resolution satellite imagery was important for the creation of bigger and better databases. Moreover, transfer learning played a crucial role, as it has made possible for researchers all around the world to use already trained CNNs as feature extractors. This, in particular, has reduced the training time dramatically, making possible to save time and computational resources. This concept is further analyzed in chapter 5. Fine tuning is a technique sometimes used in combination with transfer learning, especially when the application for which the CNN is used is different from the original one. The possibility of using it in this work has been explored, as discussed in section 5.6.

All these improvements are crucial to make possible the application of machine learning solutions to problems like population density estimation, and the most important consequence is the reduction of cost and time required to have the same data with respect to the traditional methods.

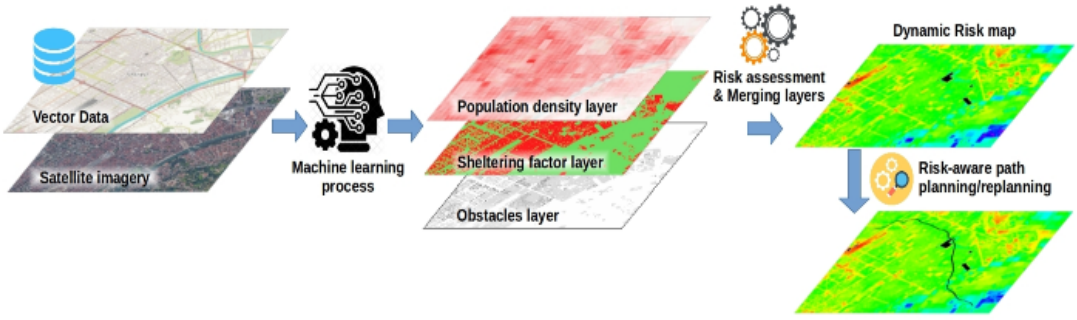

Figure 1.2: Workflow of the ML-based approach

Following the same approach, a new method is proposed for the computation of risk maps. Since both population density and sheltering factor maps are obtained using a CNN-based approach, the possibility of direct estimation of the risk through a deep learning model is investigated. The idea is the following: the workflow shown in figure 1.1 is used to build a database that, in turn, is used to train a convolutional neural network with the same structure as that used to estimate the population density. The advantage of this approach is that once we have a trained

CNN, we can use it to estimate the risk in a very short time over big areas. Of course, the database should be of good quality and big enough to make the CNN training possible, and this requires that all the layers that contribute to the risk are well defined and properly merged.

The work developed in this thesis is part of the activities of the Amazon Research Award "From Shortest to Safest Path Navigation: An AI-Powered Framework for Risk-Aware Autonomous Navigation of UASs" granted to Prof. Rizzo.

Chapter 2

State of the art

Many approaches have been proposed in recent years to build population density maps of different resolution starting from remote sensing data. In many research activities ([3], [4], [5]), the VGG16 network [6] was used as a starting point, to which more specific layers were added to properly combine the extracted features. The benefits of doing fine-tuning to adjust the parameters of the last layers of the pre-trained network are demonstrated by [3], and they are due to the fact that VGG16 was trained on Imagenet database, that contains only natural images. These are quite different from satellite imagery: as an example, for those, the concept of "centre" is meaningless, and they don't have an orientation. This also explains why the results obtained by VGG16 when used with satellite imagery are worse than when it is used with natural images, even with a smaller database.

Another possibility consists in performing a classification over a large number of classes and then take the median value as estimate of the population density [4]. In the same work, results are compared with US census data obtained using traditional county population projection methods, and they turn out to be worse. However, they are influenced by errors made by the model due to noisy input data, that are investigated qualitatively. Sparsely populated areas are sometimes classified as highly populated because of the presence of uninhabited buildings. In figure 2.1 three examples are shown: a Disney park, a laboratory, and an army depot.

Recently, the research group Facebook AI used an approach based on machine learning and satellite imagery, along with the available census data, to create population maps characterized by high resolution and accuracy [7, 8]. The population estimate is obtained counting the buildings present in the area. Satellite imagery were pre-processed using classical computer vision techniques to keep into account that most of the world's land does not contain a building, so the dataset in unbalanced. The network used is ResNet-50, and fine-tuning was employed. The number of buildings in a given area is obtained with an error estimated to be

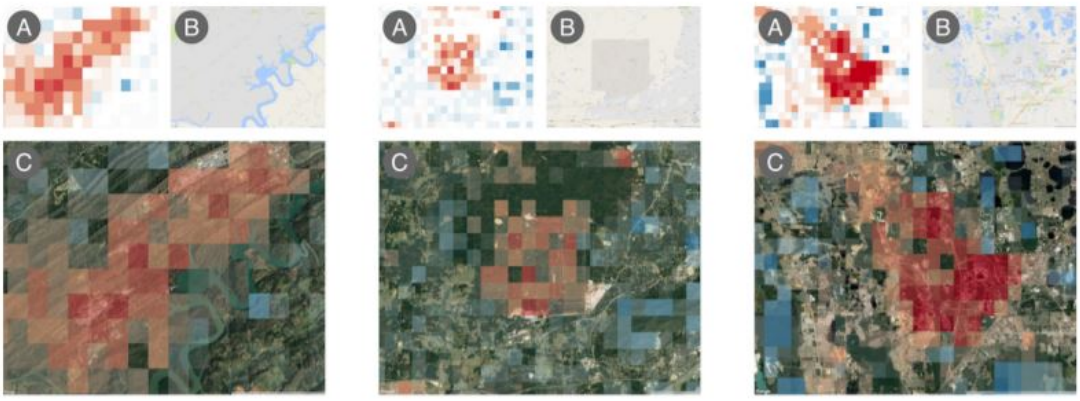

Figure 2.1: Examples of some big errors

few percent, then population counts (available from census) are distributed using proportional allocation. Results are 30m resolution population density maps for 18 countries, which show that both precision and recall are high (0.95 and 0.91, respectively), and the model can effectively be used in different geographic regions with good results. The performance of the model is slightly worse in urban areas with respect to rural areas: this is because of the larger diversity of buildings' height and use in cities. The results are shown in figure 2.2: the first 2 images refer to two traditional methods used to estimate population density, while the last one is the ground truth (for the red area, data are not available). The map obtained through this novel approach is more precise than the other methods, especially in rural areas.

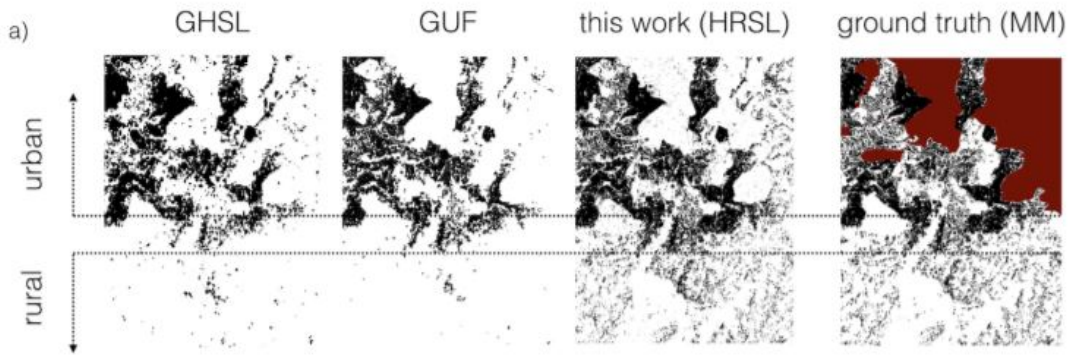

Figure 2.2: Facebook AI results

A novel and promising approach towards estimation of population distribution

consists in using mobile phone data. Call activity data have been used to investigate the population distribution and its dynamics. The main advantage is the capability of this approach to provide a dynamic picture of population distribution. This possibility was investigated recently by [10], that shows that the accuracy of the results is strongly dependent on the density of the phone cells. The same research reveals, using phone calls data of France and Portugal, that good estimates of population density maps can be obtained, and the model implemented turned out to be robust, even if some coefficients must be tuned when considering countries with a considerably smaller mobile phone penetration. Some drawbacks are also present: privacy issues must be tackled, and the correlation between the call (or caller) volume and underlying population may be not so strong. This fact was investigated by [11], and two main problems about this approach have been suggested. The first one is that the group of mobile phone users is not representative of the world's population over several points of view (social and economic status, age, etc), the second is that mobile phone users are nor evenly spatially distributed. The same issues are partially recognized by [7], in which population distribution maps with resolution 235x235m are obtained combining census data, phone calls data and satellite imagery of Milan.

An interesting work that involves satellite imagery without the use of machine learning algorithms is presented in [12] and [13]. Here, nighttime satellite imagery from the Defense Meteorological Satellite Program are used to analyze the correlation between light levels and population density. Basically, a nighttime image (figure 2.3) is transformed into a binary image (figure 2.4), where pixels are white if they are associated with a value of light above a threshold. Then, saturated pixels were turned into urban clusters based on the adjacency to other saturated pixels. Then, the distance between each pixel and the edge of the cluster to which it belongs was used to assign a value of the population density. Different functions and different pixel resolutions were used, and the best result was obtained with a Gaussian function and a 10 km^2 resolution (the resulting value of the coefficient of determination R^2 is 0.709). This method is very simple, it is hard to extend to other countries, as the relationship between nighttime lights and population density varies a lot. Once again, it is confirmed that population density depends on many cultural, economical and environmental factors that must be taken into account.

Figure 2.3: Nighttime image

Figure 2.4: Saturated image

Chapter 3

Machine Learning and Convolutional Neural Networks

3.1 Introduction to machine learning

Machine learning (ML) is the discipline that studies computer algorithms that are able to improve autonomously their performance using data. An important part of ML is deep learning, that studies artificial neural networks and their learning algorithms.

The first work published about machine learning followed the idea of emulating the behaviour of neurons: the Threshold Logic Unit (TLU) was proposed by Warren McCulloch and Walter Pitts in 1943. It was a simple model of the neurons built using electrical circuits. It performs a weighted sum of the inputs and, if this sum overcomes a threshold, it outputs a quantity. Using these networks it is possible to build any Boolean function. In 1958 the first artificial neural network model was created by Rosenblatt, and it was called Perceptron, a simple linear classifier. A few studies in this field were published in the following years, but the interest started to grow at the beginning of the 21^{st} century. The limited computational capability of computers has been a problem for the development of complex neural network models. With the development of GPUs and (in 2016) TPUs, computations on vectors and matrices has become much faster.

A NN is made of several layers, each of which has a certain number of neurons. A neuron is able to generate on output a signal depending on the input signals it receives. NNs are formed by an input layer, that is the layer that receives data, an output layer that generates the output signal, and several hidden layers (figure

3.1). Those are called hidden because their inputs and outputs are not visible, and

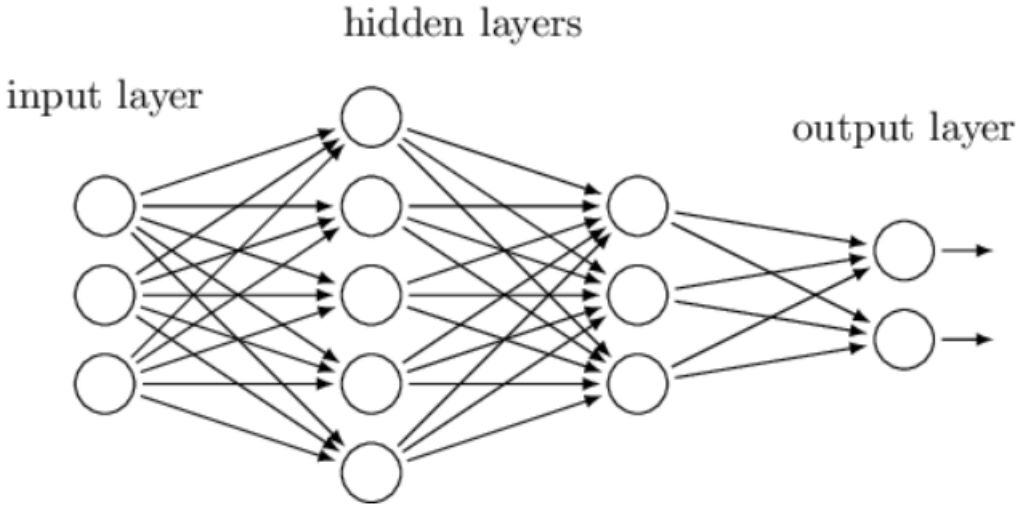

Figure 3.1: Neural network structure

their task is to transform the data. The output of each neuron depends on the activation function, that is the function that links the inputs to the output.

Machine learning algorithms can be divided into 3 groups: supervised learning, unsupervised learning and reinforcement learning algorithms. In supervised learning, the data available are composed of the input and the corresponding output that the model should give. The NN should be able to reproduce a function that is the mapping between the input and the output. The algorithm used in this thesis belongs to this group. In unsupervised learning, the problem consists in structuring the dataset given as input, without having information about what the output should look like. Some possible applications of this technique are market segmentation, social analysis and astronomical data analysis. Figure 3.2 shows two examples. In the first problem data are labeled and the goal is to classify them, while in the second one the goal is to cluster data having no information about them: the NN algorithm should find patterns in the data.

In reinforcement learning, instead, an agent learns to find the best possible action in a given situation on the basis of its experience, and achieves a reward if the action is correctly carried out. The agent should maximize the cumulative reward.

Each neuron of a NN has a vector of weights and a bias unit associated. In supervised learning, the training phase consists in tuning the values of these units using labeled data with the goal of minimizing the loss. Loss measures how a model is working: it is zero in the case of a perfect prediction, and it grows as the

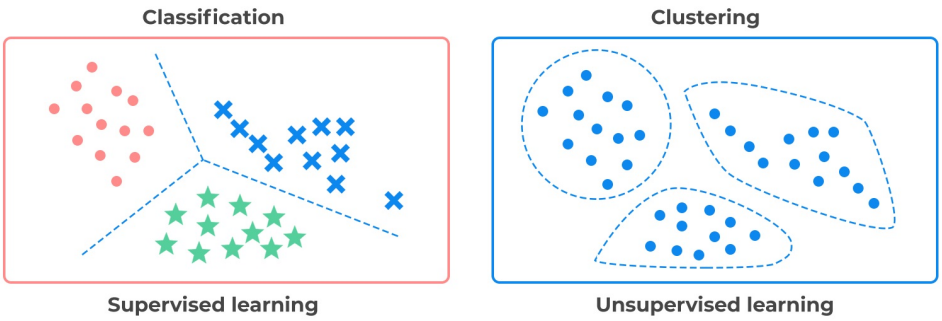

Figure 3.2: Supervised vs unsupervised learning

prediction diverges from the correct value. The cost function sums the loss of all the samples, and an optimization algorithm is used to minimize its value.

Gradient descent is an iterative optimization algorithm used to minimize the cost function. Given a cost function $J(\theta_1, \theta_2)$, the goal is to find the values of the parameters θ_1 and θ_2 that lead to the minimum value of J. To do that, an initial guess is made, and then the values are updated until the minimum is reached. The parameters are moved in the direction that implies the biggest possible decrease of the cost function. The parameters are updated according to:

$$\theta_j = \theta_j - \alpha \times \frac{d}{dt} J(\theta_0, \theta_1), j = 0{,}1 \tag{3.1}$$

α is called *learning rate*, and it controls how big the steps of gradient descent are. If it takes a small value, the algorithm will be too slow to converge. On the other hand, a big value implies big updates of the parameters which can lead to a diverging behaviour. The two situations are shown in figure 3.3. There exist several extensions of this algorithm, designed to speed up the optimization process, and the most popular one is Adaptive Movement Estimation (Adam). This is an algorithm that uses adaptive learning rates methods to find individual learning rates for each parameter.

Backpropagation is an algorithm used to compute the gradient of the cost function with respect to the variables of the model. The idea of using backpropagation for neural networks was introduced in 1986 ([14]). In the training phase of a neural network, each sample is passed to the NN model, that gives the output value (or output vector, in the case of classification). To get the output, the input is forward propagated through the network. The error can be computed as the difference between the output of the model and the true value used as ground truth. To tune the weights and biases, this error is propagated back all the way to the first layer, iteratively applying the chain rule. This is a rule from calculus useful to obtain the

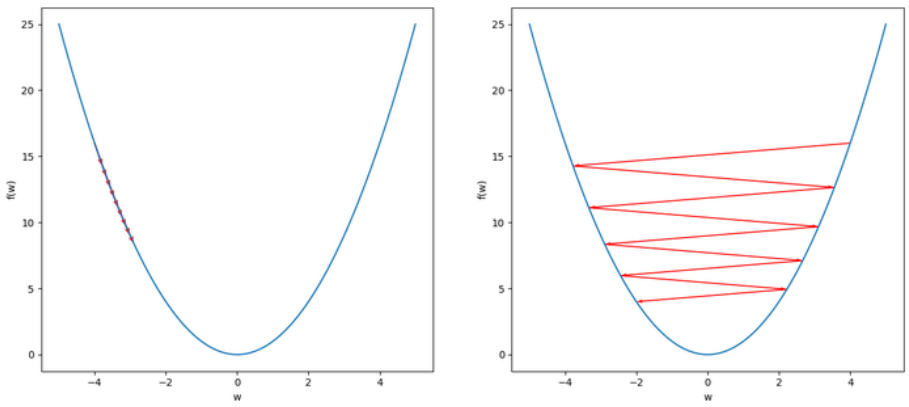

Figure 3.3: Learning rate

derivative of a function using other functions which have known derivatives.

3.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a particular type of deep neural network architecture, widely used for image-related tasks. CNNs were introduced by Yann LeCun in [15], where they were used to classify handwritten digits. The architecture of CNNs is inspired by how the human brain recognizes objects ([16]): the visual cortex is organized into several layers, made by cells which can be classified into 2 categories: simple cells and complex cells. Simple cells respond to bars of light and dark, and each cell responds differently depending on the orientation of the bar. Complex cells have instead a less strict response profile. The visual cortex can be modeled as a series of alternated layers of complex and simple cells.

Convolving a set of filters with an input image, a features mapping effect similar to that of simple cells is obtained. Subsampling, instead, plays the role of a complex cells layer. Several sequences of convolutional and subsampling layers make a CNN model. The comparison between the model of the human cortex and CNNs is shown in figure 3.4.

Figure 3.5 shows the typical architecture of CNN models, that is composed of two parts: the first one serves as feature extractor, and this part is often imported from a pre-trained model (like VGG16), following the principle of transfer learning. Early layers of this part extract low-level features (egdes, textures, shapes and so on), while later layers are linked to complex features, that are used by the second part of the model. This is made of fully connected layers, that have the effect of learning non-linear combinations of high level features. The dimension (that is,

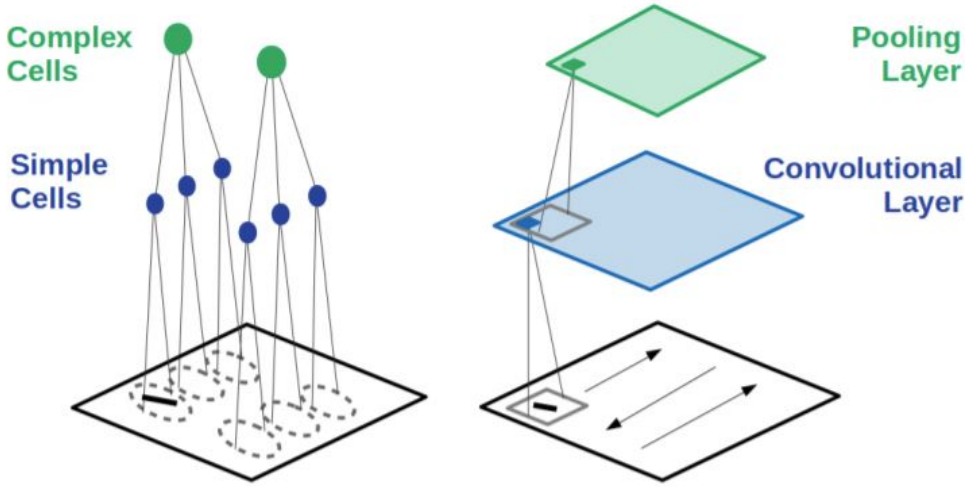

Figure 3.4: Visual cortex model vs CNNs

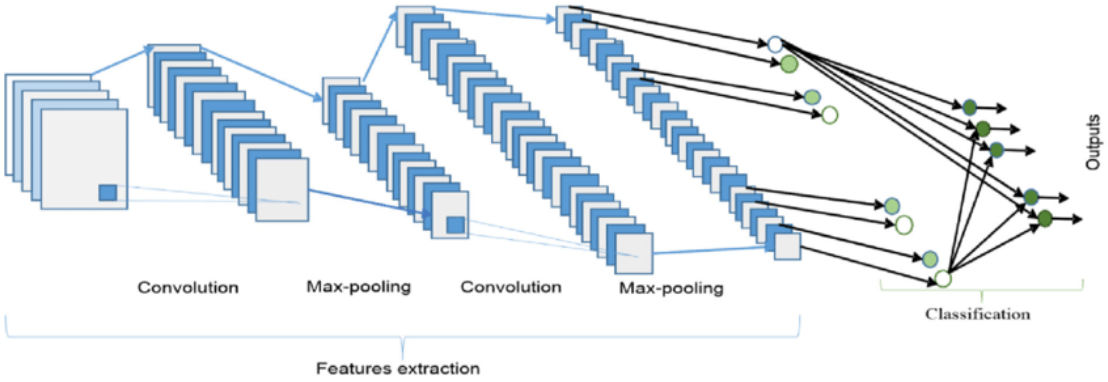

Figure 3.5: CNN structure

the number of neurons) of the last layer of the model is equal to the number of categories in which we want to classify the images. Each neuron is associated with a class, and the value on output is the probability that the image belongs to that class. The activation function of the neurons is "relu", that means "rectified linear unit" (see figure 3.6).

In regression applications, the output is a real value and the last layer of the model consists of one neuron, that has "linear" as activation function. The difference between classification and regression is highlighted in figure 3.7: in the first case, data must be divided into two or more category, while in the second one the goal is to find a function that approximates the data.

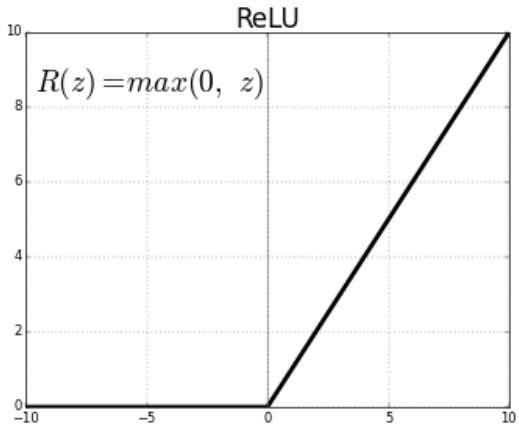

Figure 3.6: Relu activation function

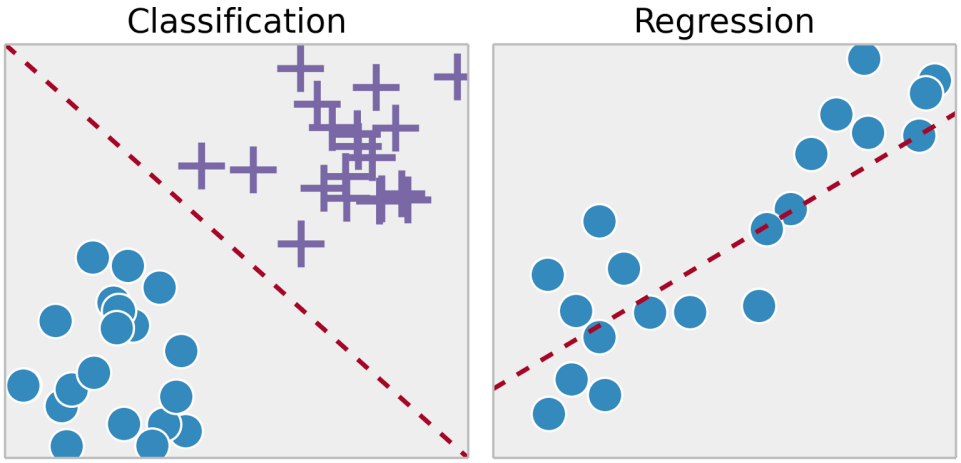

Figure 3.7: Classification vs. Regression

3.3 Convolutional layers

Convolution is a mathematical operation between vectors, defined as:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y_i = \sum_{k=-\infty}^{+\infty} x_{i-k} w_k \tag{3.2}$$

$\mathbf{x}$ is the input vector, while $\mathbf{w}$ is the filter (or kernel). The index k goes from $-\infty$ to $+\infty$, and to make it possible the vector $\mathbf{x}$ is zero-padded: zeros are added to both sides of $\mathbf{x}$, and the resulting vector $\mathbf{y}$ will be of infinite size. In practice, x is padded only with a finite number of zeros, denoted with p. If $\mathbf{x}$ and $\mathbf{w}$ have

dimensions n and m respectively, the padded vector $\mathbf{x_p}$ will have $n + 2 \times p$ elements, and equation 3.2 becomes:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y_i = \sum_{k=0}^{m-1} x^p_{i+m-k} w_k \tag{3.3}$$

Note that $\mathbf{x}$ is indexed in the opposite direction of $\mathbf{w}$, and that is equal to make the sum of the products once one vector is reversed.

Images can be seen as matrices of pixels, so we can perform convolution with a filter matrix. The 2D convolution between matrices $\mathbf{X}$ and $\mathbf{W}$ is defined as:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} \rightarrow Y_{ij} = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X_{i-k_1, j-k_2} W_{k_1,k_2} \tag{3.4}$$

The sizes of $\mathbf{X}$ and $\mathbf{W}$ are $n_1 \times n_2$ and $m_1 \times m_2$ respectively, with $m_1 \leq n_1$ and $m_2 \leq n_2$.

An example of convolution between a matrix $\mathbf{X}$ and a filter $\mathbf{W}$ is shown in figure 3.8. Zero-padding with $p = 1$ is applied to $\mathbf{X}$, stride is equal to 2 and the filter matrix is:

$$W = \begin{bmatrix} 0.5 & 0.7 & 0.4 \\ 0.3 & 0.4 & 0.1 \\ 0.5 & 1 & 0.5 \end{bmatrix}$$

In the figure below, this matrix is rotated according to the convolution formula: it is sufficient to take the sum of element-wise product. The kernel is like a sliding window which is moved each time of a value equal to the stride, until it reaches the last element of the matrix $\mathbf{X}$.

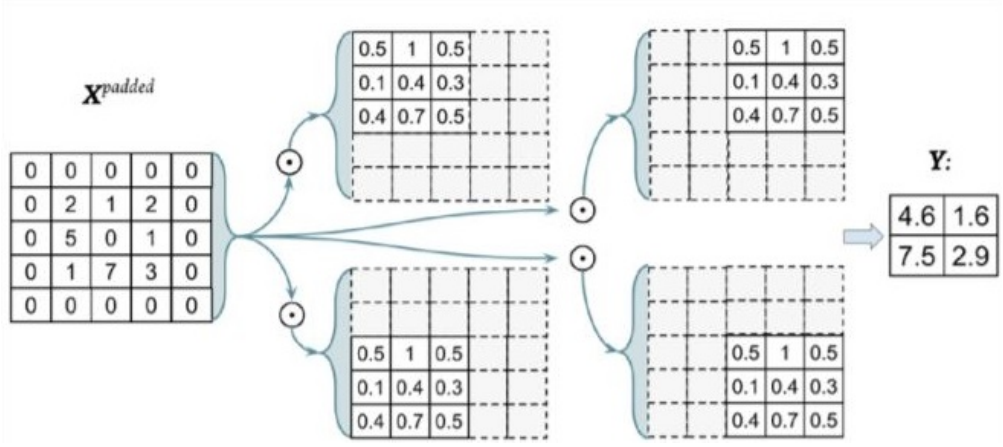

Figure 3.8: Convolution operation

Convolutional layers can be easily implemented using the *Conv2d* Keras function, that is called giving as input the filters, the kernel size, the padding and the activation function.

3.4 Subsampling layers

Subsampling is an operation that is typically performed in the form of max-pooling or mean-pooling. A pooling layer has size $n_1 \times n_2$, and its effect is simple: max pooling takes the maximum value in each set of elements of dimension $n_1 \times n_2$, while mean pooling takes the average value in the same set. An example of max and mean pooling is reported in figure 3.9.

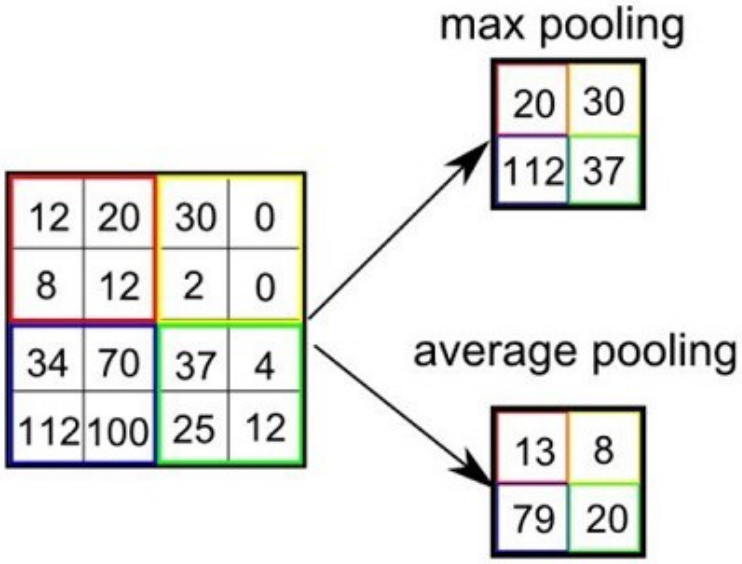

Figure 3.9: Max and mean pooling

Pooling layers do not have weights associated with them: it is sufficient to define the size and the stride. The beneficial effects of pooling layers are two: they introduce local invariance, making the learned features less sensitive to small variations of the input, and they also decrease the size of features.

Depending on the pooling technique used, there are several Keras functions available to add a pooling layer to a model.

3.5 Dropout layers

A neural network model has a certain capacity, that is defined as the level of complexity of the function it can learn to approximate. Models with a small number of layers are likely to underfit, while too complex model will tend to overfit.

The goal of a NN algorithm is to have a model that preforms well on both training and unseen data. The ability of a model to fit well data that were not used for training is called *generalization*. Sometimes it happens that a model performs

well on the data on which it was trained, but it is not able to fit unseen data. This problem is called overfitting, and it is caused by the fact that the model learns very fine trends present in the data. On the other hand, a model that does not learn the sufficiently the problem will perform poorly on both the training and validation data. This is called underfit. The two problems are shown in figure 3.10, where the goal is to separate the labeled points into two classes. The first model underfits the data, the second one fits them well and the third one is affected by overfitting.

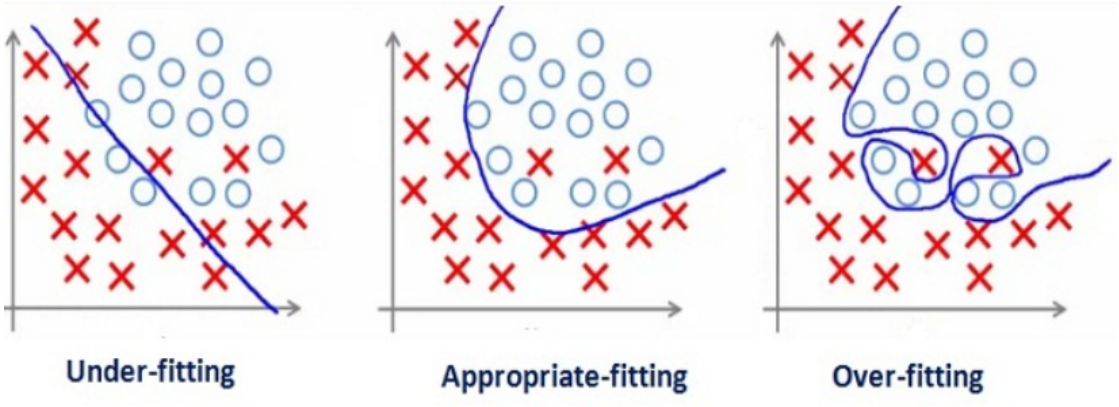

Figure 3.10: Underfitting and overfitting example

The two main reason for overfitting are the size of the database and the complexity of the model. Since there are a lot of weights to tune in the model, it is necessary to train it on a proper number of images. Deep neural network continue to improve their performance as the size of the database increases. The complexity is another important aspect to consider. Reducing the complexity reduces the likelihood that the model overfits the database. The complexity or capacity of a NN model depends on the number of weights and layers present.

Underfitting, instead, is the opposite problem, and it arises when the model chosen is too simple (that is, there are not enough weights to tune), so that the results on the training data are bad.

A possible approach to design a NN model consists in building a model with high capacity, and then apply regularization schemes to avoid overfitting. Dropout is one of these schemes, and it is described in [18]. It is a useful technique adopted to avoid overfitting in deep NNs, and it is applied to the hidden layers. During training, at each iteration hidden units are dropped with probability p_{drop}, that is usually chosen as 0.5; the weights of active neurons are updated to account for the missing units. This results in the fact that the network is forced to learn more robust patters from data. Neurons are randomly dropped out during the training phase, while they are all active in the evaluation phase (figure 3.11).

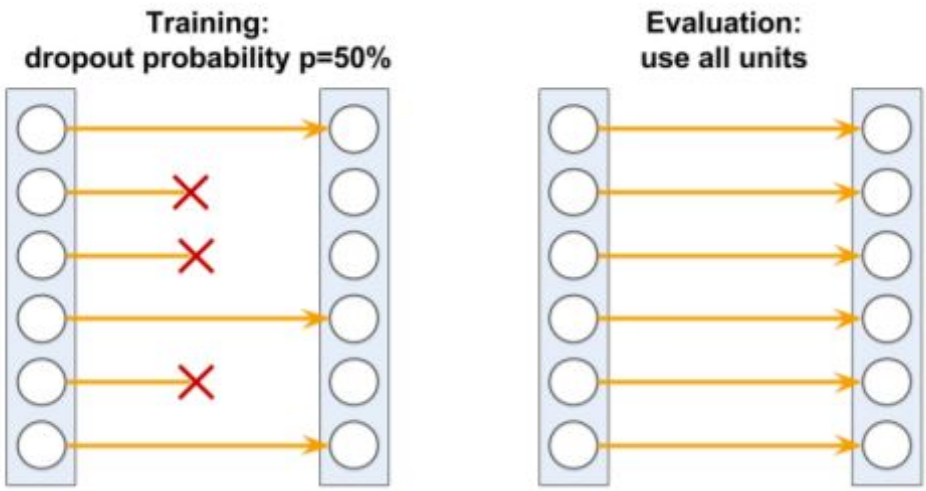

Figure 3.11: Dropout layer

Adding a dropout layer in a sequential model only requires to call the *Dropout* Keras function and give to it the probability of dropping a unit.

3.6 Dense layers

After the layers imported from the VGG16 model (figure 3.13), dense layers are added to reduce the dimension of the output. This is an important operation that must be done gradually. In the case of this thesis, two convolutional 2D layers and a flatten layer have been added after the VGG16 model. The flatten layer has the function of reshaping the tensor given as input, unrolling it into a column vector. The dimension of this vector resulted to be 6272 (figure 3.14). This number had to be gradually reduced to reach the last layer where only one neuron is present, and this have been done using dense layers with dimensions corresponding to some of the powers of 2. Some dropout layers have been inserted among dense layers to reduce overfitting. The final structure consists of 71 million parameters (weights and biases), 56 millions of which are trainable.

Dense layers are the Keras implementation of fully-connected layers, that are the building blocks of neural networks. Their main characteristic is that each neuron of a layer is connected to every neuron of the previous layer (see figure 3.1).

The output they produce depends on the activation function. The most simple activation function is the "linear" one, where the output is the product of the input vector $\mathbf{X}$ and the weight $\mathbf{W}$, plus the bias b:

$$y = \mathbf{XW} + b \tag{3.5}$$

This activation function is very simple, but it is not sufficient in many applications: a non-linear function is needed to avoid instability problems. Several functions can be used (four among the most common are represented in figure 3.12), and the choice depends on the application.

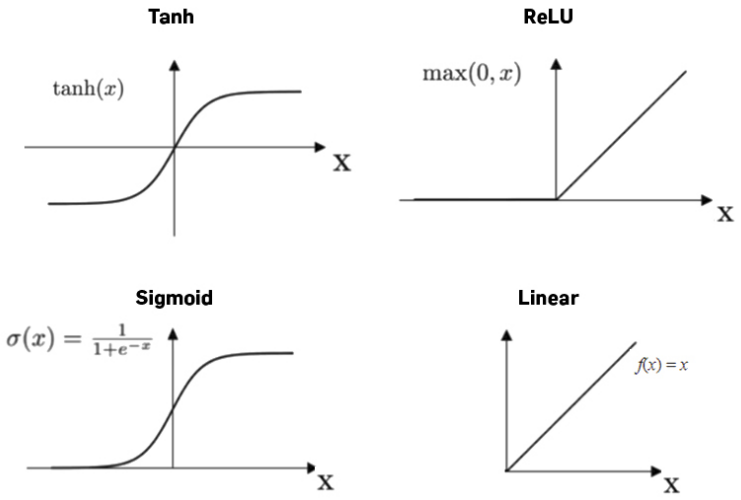

Figure 3.12: Activation functions

The *Dense* function of Keras can be used to add a Dense layer to a sequential model, passing to the function the number of output units of the layer and its activation function.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

Figure 3.13: Layers of the VGG16 model

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 7, 7, 512)         14714688
_________________________________________________________________
conv2d (Conv2D)              (None, 7, 7, 128)         3211392
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 128)         147584
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 4096)              25694208
_________________________________________________________________
dropout (Dropout)            (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2048)              8390656
_________________________________________________________________
dropout_2 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_3 (Dense)              (None, 1024)              2098176
_________________________________________________________________
dropout_3 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               524800
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_6 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_7 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 33
=================================================================
Total params: 71,638,849
Trainable params: 56,924,161
Non-trainable params: 14,714,688
```

Figure 3.14: Layers of the CNN used

Chapter 4

Software used

4.1 Introduction

The thesis has been developed using Python as programming language and some packages for machine learning applications. Matlab was also used for a small portion of the work, that is described in section 5.4. The training of the CNN was carried out using the AWS EC2 cloud computing platform, as it required high computing capability due to the large number of weights to tune in the model.

4.2 Tensorflow and Keras

Tensorflow is an open source platform for machine learning, useful for the implementation of deep neural network algorithms ([19]). It was developed by the Google Brain team and released in 2015. It is an ecosystem of libraries, tools and other resources that facilitates the development and distribution of machine learning-based applications. Tensorflow has an architecture that makes possible to run applications in CPUs, GPUs and also TPUs.

Keras is an open source library for machine learning that acts as an interface with Tensorflow. It was created by Francois Chollet, a Google engineer, who released the first version in 2015. His goal was to make available for researchers an easy-to-use tool to develop deep learning applications. Keras is written in Python and it needs a backend to work, which has to build the network graph, run the optimizer and perform many other low-level operations. Originally, its backend was Theano, but as of now Keras is fully integrated in Tensorflow, in the form of the *tf.keras* high level API, while the standalone version of Keras is no longer maintained.

When running a Tensorflow application, if a GPU is available the code will run on it, without any explicit code configuration. The GPU version of Tensorflow

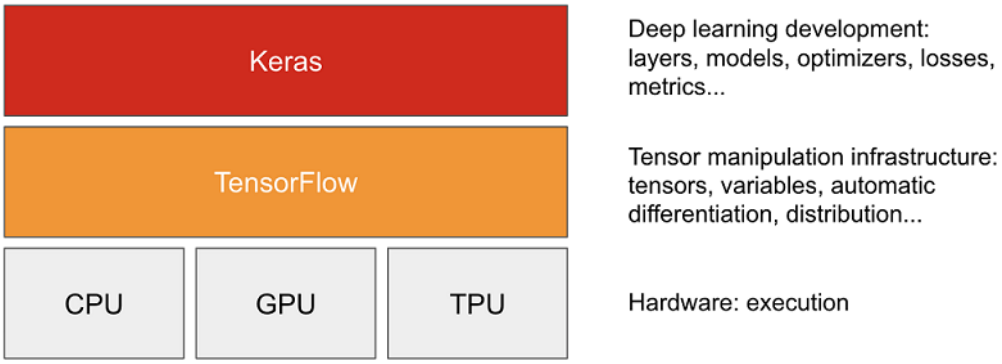

Figure 4.1: Keras and Tensorflow hierarchy

is currently available for Windows and Ubuntu. To run a Tensorflow application on a GPU, a NVIDIA GPU card compatible with CUDA is required. CUDA is a parallel computing platform and an API created by NVIDIA which makes available the computing power of GPUs to speed up computing-intensive applications.

Tensorflow can also run on TPUs. A TPU is an AI accelerator application-specific integrated circuit developed by Google for machine learning applications, designed for high volumes of low precision computation. TPUs are proprietary, so they are available only in the Google Cloud Platform services.

In the following, the most relevant functions imported from Keras and Tensorflow are briefly described.

First of all, the model used is *Sequential*, that means, it is a linear stack of layers where each layer has exactly one input tensor and one output tensor. When adding layers to the model, different parameters must be specified depending on the layer, as discussed in chapter 3. Then, the model is compiled using the *compile* function, and in this phase the loss function, the optimizer and the metrics must be specified. The loss function (or cost function) is the function that gets minimized by the optimizer, and the *mean square error* was chosen, that is defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{4.1}$$

y_i is the prediction, $\hat{y}_i$ is the ground truth, and N is the number of samples.

The optimizer is the mathematical function applied to the model that computes the weights of each neuron in order to minimize the loss, and *Adam* was chosen. As metrics, the *mean absolute error* is used. Using a metric is useful to judge the performance of the model. Anyway, the choice of the metrics does not influence the optimization process at any level.

At this point, the training is launched using the Keras function *fit*. Several parameters are passed to this function:

- input data: NumPy vector or Tensorflow tensor containing the input of the model (in this case, the satellite imagery transformed into arrays);
- target data: Numpy vector or Tensorflow tensor (constistent with the input), containing the ground truth for the model;
- batch size: number of samples per gradient update. A big batch size results in a faster training, but it also requires more CPU or GPU memory;
- epochs: number of complete iterations over the entire data x and y. A trade-off between precision and training time has to be done, and the choice can be made with the aid of learning curves;
- verbose: it determines the level of information given about the training, and it can be chosen between 0 (silent), 1 (progress bar) and 2 (one line per epoch).

Another important parameter that can be used is *callbacks*, that is used to stop the training when a monitored metrics has stopped improving, or if the improvement is too small. The purpose of using callbacks is to save time and computational resources. Callbacks have not been employed in the final model, because the training done with the GPUs provided by the AWS EC2 instance chosen was quite fast.

To evaluate the performance of a model after the training and to build the learning curves, the *evaluate* function was used, giving as input the predicted values (output of the model), the ground truth and the *verbose* parameter. The *pyplot* function of the Matlplotlib Python library was used to plot the learning curves.

The last step consists in making the predictions on the images of interest. This is done calling the function *predict* on the model, giving to it the image array as input. To compute the value of R^2, the *r2_score* function from sklearn library was used with two vectors as input: the true and the predicted values of the population density.

4.3 Other Python libraries

Some Python libraries other than Tensorflow and Keras have been used for the development of the thesis, and they are listed here, along with a brief description:

- Sklearn: Scikit-learn is a Python library for ML that contains tools to implement different classification and regression algorithms, and it also provides functions to compute some of the most common metrics used to evaluate the performance of a ML model. The functions used in this thesis are *MinMaxScaler* for data preprocessing and *r2_score* to compute the coefficient of determination R^2.

- MatPlotLib: it is a popular library that provides an easy way to plot graphs providing the *Pyplot* function. Functions are available to easily add elements like the title, the legend and the axes labels to plots.

- NumPy: it is the most used library to work with numerical data in Python. In particular, NumPy contains multidimensional arrays and matrix data structures. The most popular structure is *ndarray*, that is an n-dimensional object that contains data, and it comes with methods to efficiently perform mathematical operations on it. Differently from Python lists, a *ndarray* object contains elements of the same data type and they are also faster and more compact than lists.

- PIL: the Python Image Library contains functions to open, modify and save images. It has been used to crop and save the images downloaded from Bing Maps API, as described in section 5.3.

- Pandas: it is a Python library for data analysis. It is used to build data structures and manipulate time series. In the thesis, it is used to save the training history.

- xml.etree.ElementTree: xml is a markup language that is designed to store and transport data and to be both human-readable and machine-readable. It does not have pre-defined tags, and for this reason it is extensible. The structure of a xml file is tree-like: there is the root element that has several child elements. Each child has in turn other children. An example of xml code and the corresponding structure is shown in figure 4.2. Python offers a solution to interface directly with xml files: the *xml ElementTree* module. This contains the *parse* function that makes the tree corresponding to the code available, on which the *getroot* method can be called to access the root. To access a child element, the *attrib* function can be used.

- requests: it is a library that allows to make HTTP requests simpler for humans. To make a request, the *get* method can be used, giving as input the URL of the page to download. The value returned by a get request is an object of type *response*, and it contains three parts: a first line that describes the message, a block of headers containing the attributes and the body, that contains the data. It is possible to check whether the request succeeded by calling the *status_code* attribute of the response object. In this work, the requests library was used to make requests to download images from the Bing Maps API. When saving an image, the *write* function is used. It is important to split the image in chunks of reasonable size in order to avoid that the process occupies too much memory.

a) XML document

```
<library>
   <book>
        <year>1990</year>
        <title>XML Labeling</title>
        <publisher>Smart Publ.</publisher>
   </book>
   <magazine>
         <issue>1</issue>
         <title>Technology</title>
         <publisher>Ideal Publ.</publisher>
   <magazine>
</library>
```

b) XML Tree representation

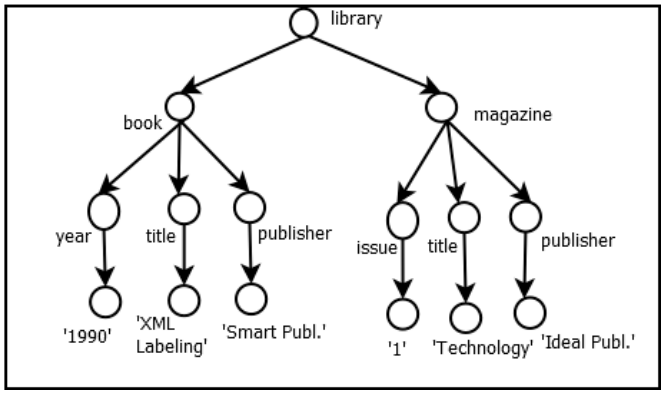

Figure 4.2: Xml example

- pyproj: it is the Python interface to PROJ, a coordinate transformation software. The *geod* was used in this work to determine the coordinates (latitude and longitude) of a point, given the initial point, the distance and the azimuth, with the goal of avoiding overlapping between images.

In addition, it is also to mention the use of Bing Maps REST Services API, used to download satellite imagery from Bing Maps. This API offers a high degree of customization on the maps obtained. The area of interest can be specified using the centre point and the dimensions, or defining the 4 points of the bounding box. It is also possible to choose the type of map (for example, street maps or aerial, with or without labels and so on). The *get* function from *requests* was used to interface with the service.

4.4 Amazon Web Services

The training of a CNN is an heavy task from the computational point of view. That is because the architecture of the network used consists of millions of weights that have to be tuned.

Amazon Web Services (AWS) is a subsidiary of Amazon that provides on-demand cloud computing services. Cloud computing is beneficial for users, as it is an alternative to buying and maintaining physical data centres and servers. The main features are the pay-as-you-go pricing model and the high availability rate. The first consists in the fact that the user pays just for what he uses, so that the service can be easily adapted to changing needs. The second one is made possible by using specific clusters, designed to guarantee the minimum possible amount of down-time, so that the service is highly reliable.

In the context of this thesis, the Amazon Elastic Compute Cloud (EC2) service has been used. This allows users to rent a virtual machine, called Amazon Machine Image (AMI), that runs on AWS servers, in which users can deploy their applications.

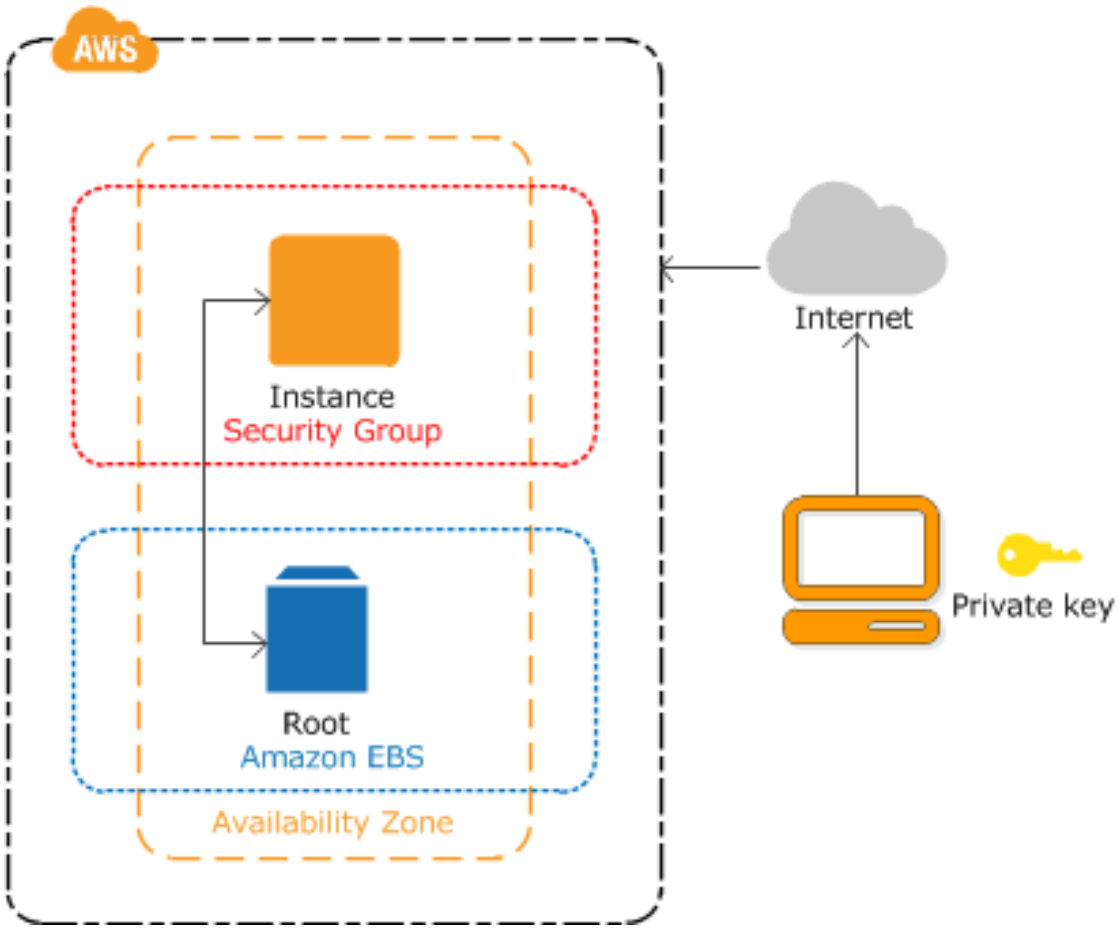

Figure 4.3: AWS architecture

An AMI contains a software configuration, composed by an operating system, an application server, and the applications. One or more instances can be launched as copies of an AMI, running on the virtual server on the cloud. Both AWS and the community of users publish many AMIs containing common configurations that are ready-to-use. In addition, it is possible to create custom AMIs depending on the needs.

An instance can be launched and arrested as needed, and this feature explains the word "elastic" of the name EC2. AMIs are preconfigured with different operating systems that can be chosen depending on the needs (Windows and Ubuntu, for instance). A lot of different instance types are available, comprising a huge of

combinations of CPU, memory, storage and networking capacity, covering different needs. The AMI have been used to train a Neural Network model, so an Accelerated Computing instance was chosen: this uses hardware accelerators to process functions much more efficiently than what is possible relying on CPU only.

Moreover, the chosen instance is backed by Amazon EBS, which means that the root device for the instance is an Amazon Elastic Block Store (EBS) volume. EBS is a block storage service designed to be integrated with EC2 AMIs and providing high performance, scalability and security.

It is possible to connect to EC2 instances using an SSH client like PuTTY. To have access to the instance, it is necessary to know its DNS and to have the SSH key pair generated for that instance. Files can be transferred using a FTP application. FTP is a protocol used to transfer files from a server to a client on a computer network.

Chapter 5

Convolutional Neural Network design and testing

5.1 Previous work

Since this thesis is the continue of the work done in the past year by the research group, it is of interest to recall the main results obtained before jumping into the description of the steps taken to improve the precision of the estimated population density. The configuration of the neural network used is shown in figure 5.1. This structure of layers is chosen after many trials, and it is made up by two parts: the first one is composed of the layers that are part of the VGG16 network, from which the last layer has been eliminated, as that is the layer on which the final classification was performed. The second part of the network is made by layers that are chosen to gradually reduce the dimension of the output neurons from 6272 (output size of the last convolutional layer) to 1.

The database used consists of 20382 images, corresponding to the area of Turin shown in figure 5.2. This area contains different types of scenarios: the city with shops and apartment buildings, suburbs with individual houses, industrial areas with industrial buildings and warehouses, and the hilly area with few houses and a lot of trees. It is important to recall and remark these differences, as they imply different population distributions: they must be recognized by the neural network algorithm to obtain good results.

The scatter plot obtained is shown in figure 5.3. The best value of R^2 (coefficient of determination) obtained in the previous work is 0.54. The goal now is to improve this value, to obtain better population density estimates. The coefficient of determination R^2 is defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{5.1}$$

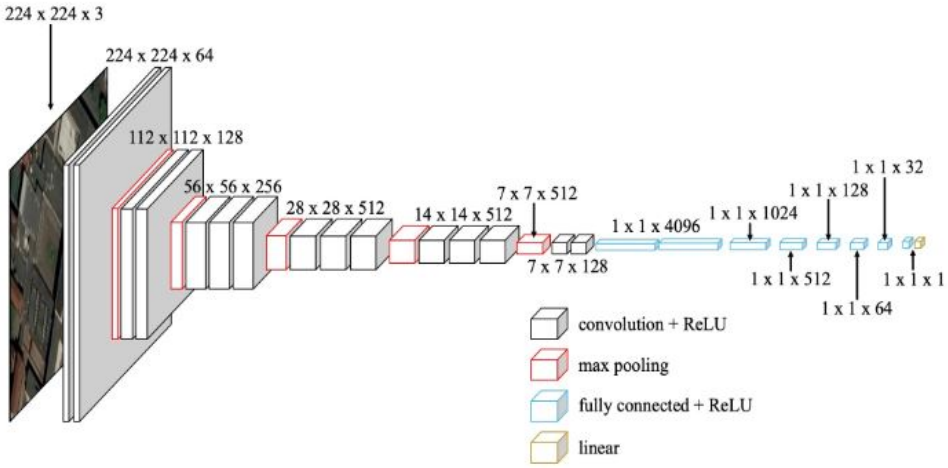

Figure 5.1: Convolutional Neural Network configuration

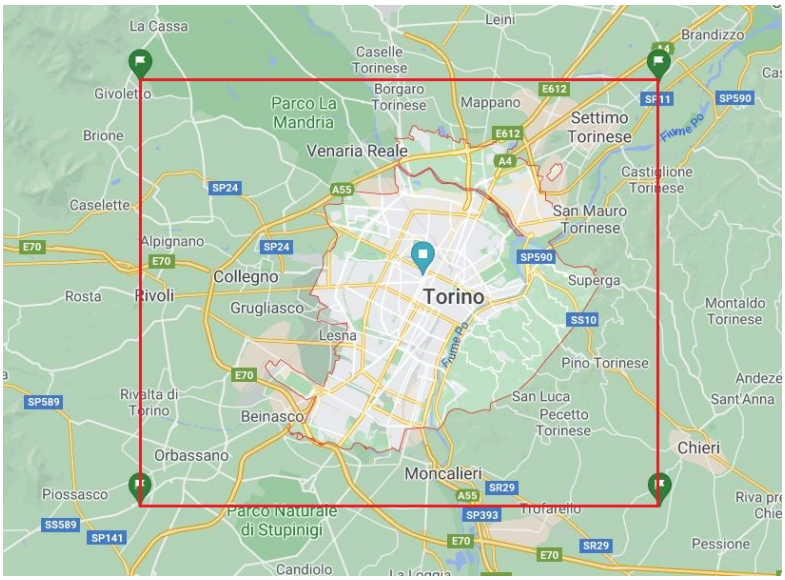

Figure 5.2: Database area

In the previous equation, y_i represents the true values, while $\hat{y}_i$ stands for the predicted values. The R^2 score is the proportion of the variance in the dependent variable that is predictable from the independent variable ([20]), so it quantifies how well new images are likely to be predicted correctly by the model. The biggest possible value of R^2 is 1, and it can also take negative values. The quality of the predictions obtained can also be represented graphically using a scatter plot: on

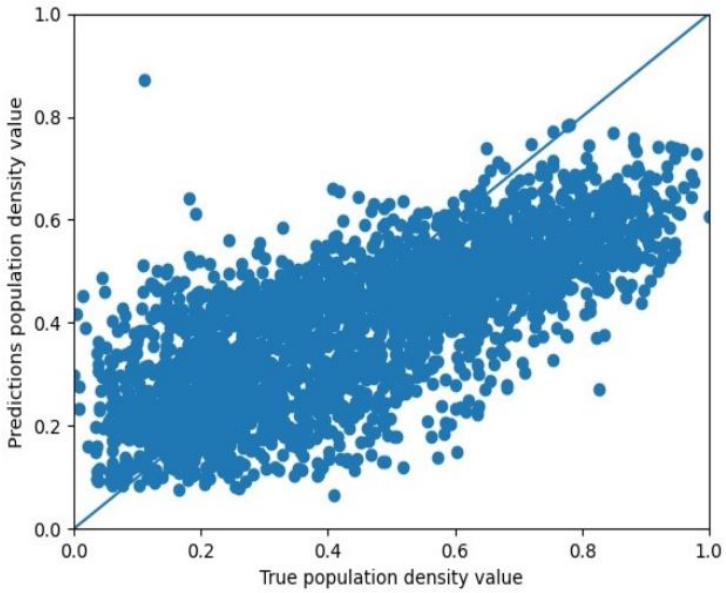

Figure 5.3: Scatter plot after the previous work

the x axis the true values contained in the database are reported, while on the y axis there are the estimated values. A perfect model has $R^2 = 1$ and all the points of the scatter plot are on the bisector of the first quadrant.

5.2 Data distribution

A brief discussion about the distribution of population density data is necessary. First of all, the absolute values are very small, ranging from 0 to 0.0175. Also, they are not equally distributed in this interval (not even remotely): most of the samples are near the lower bound. This is clearly visible from histogram 5.4, which shows how many samples there are in each range. A range is defined dividing the population density into 1000 intervals of the same size. To overcome this issue, the natural logarithm was applied to all the samples to obtain a better data distribution. This is a kind of data pre-processing, that is required because otherwise the machine learning algorithm would not work well. In fact, if we left data in the original form, a model that predicts a constant value near zero for all the samples would give high metrics values, even if it is clearly unacceptable. The distribution of data after the logarithm has been applied is shown in histogram 5.5: the number of samples in each range is now distributed more equally.

The application of the logarithm to data raises a number of problems. The final goal of this algorithm is training a CNN model that is able to estimate the

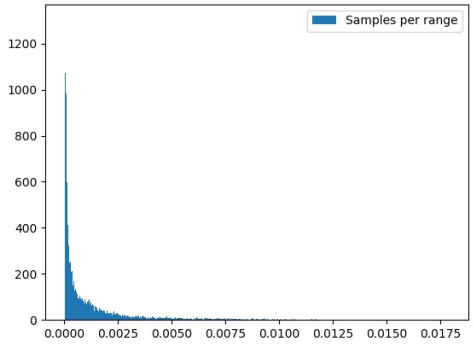

Figure 5.4: Original data

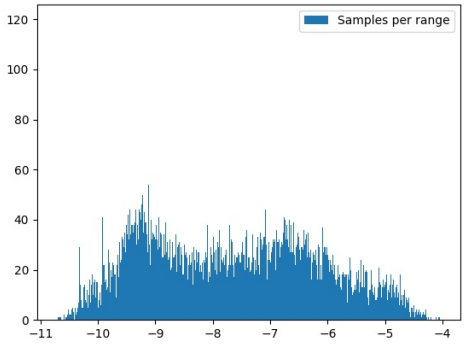

Figure 5.5: Logarithm data

population density once a satellite image is given on input. Anyway, the network is trained with data that are pre-processed applying the logarithm function and also the MinMaxScaler operator. The latter projects all the values on the interval $[0, 1]$, and it is useful to avoid numerical problems. At the end, to obtain the values of the population density, we need to apply the inverse of the logarithm (that is, the exponential function) and the inverse of the scaler. About the log-exp operation, a problem arise when we train the network with logarithmically scaled data: suppose that the prediction of 0.2 is made over an area whose correct value is 0.1. The absolute error would be 0.1, and the relative error 100%. Say, then, that an area characterized by a value of 10 is predicted as 20: the relative error is still 100%, but the absolute error is 10, that is much larger. When the logarithm is applied, the CNN model considers the two errors with the same relevance, even if the relative error is very different.

5.3 Image cropping

The first step taken to improve the results consists in cropping the images downloaded from Bing Maps Rest API. In practice, when an image is downloaded, a label is added in the bottom-right corner (as shown in figure 5.6). The label occupies a significant part of the image (estimated to be around 9.5%), and in some cases it hides relevant elements like buildings or streets, that are not taken into account by the network. To remove the labels, images have been downloaded again so that they cover a wider area, then cropped using the Python Image Library (PIL) so that the image corresponds to the correct area and the label is not present. Thanks to this simple process, a significant improvement in the R^2 score is obtained: from

0.54 to 0.65. The scatter plot obtained after this step is shown in figure 5.8

Of course, the additional operation that is performed implies that the creation of the database requires more time than before.

Figure 5.6: Original image

Figure 5.7: Cropped image

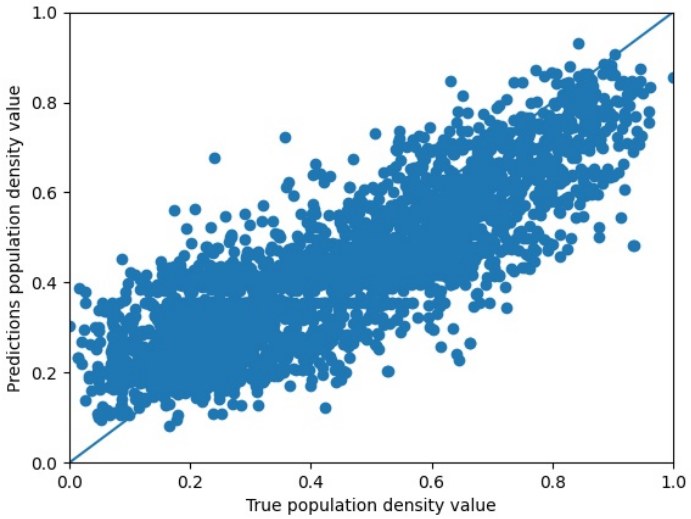

Figure 5.8: Scatter plot after cropping

5.4 Test time augmentation (TTA)

Test-time augmentation is a technique used to improve the results of machine learning algorithms based on neural networks. Basically, each original image is taken and modified into several versions (with flips, rotations, shifts and so on), which are all passed to the model that makes a prediction for each image, and the mean of the obtained output values is taken as predicted value. The idea behind this technique is that by averaging predictions, also the errors are averaged, leading to a reduced generalization error. The main disadvantage of this method is that the time required to predict over an image will increase, as a prediction over each modified image is required.

TTA is performed rotating each image of 90, 180 and 270 degrees, and each rotated image was flipped along the horizontal and vertical axes. The result is that from each original image, 12 modified versions of it are created. An example of the modified images is shown in figure 5.9.

TTA does not work for every application: as an example, digit recognition would not work well if we considered flipped or rotated digits, as they would have no meaning. Instead, with satellite imagery this problem does not arise: flipping or rotating an image, the information contained in it does not change, as it is instead contained in features like the number of buildings and trees and in the colours. A complete study of the effects of TTA is contained in [21]. In this article, two popular deep learning classification models (ImageNet and Flowers-102) were studied, and the effects of two types of TTA (standard and extended) were evaluated, taking as metrics the percentages of predictions corrected and corrupted by TTA. It turned out that the net effect of this technique is nearly always positive. However, the improvement is related to many factors. For example, the more accurate the model, the lower the benefits of TTA, and the reason for this is that a model trained on more data generalizes better, and it is less sensitive to image modifications. Moreover, the improvements obtained for the Flowers-102 network are significantly smaller than those obtained for ImageNet: flower pictures are typically taken centred and from the same perspective, while ImageNet object are very different.

After TTA had been performed performed, an analysis was carried out on the results, to find the images that have benefited the most from this process. It turned out that the best improvements are obtained on images that have inside geometric objects and shapes, like straight streets shifting them into various parts; some examples are reported in figures 5.10 to 5.13.

The value of the metrics R^2 obtained is 0.74, and the scatter plot is shown in figure 5.14.

After this process, an analysis on the estimated values was also done. In fact, since 12 different values of the estimated population density value have been

Figure 5.9: On the top left corner the original image, while the others modified versions

obtained and their average has been taken as population density estimate, the possibility of removing outliers has been investigated. First of all, some of the estimates have been plotted along with their mean and the corresponding exact value: the results are shown in figure 5.15, in which each row corresponds to a sample.

The Matlab function "rmoutliers" [22] was used. By default, a value that is more than three scaled median absolute deviations (MAD) of the vector under analysis is considered as an outlier. Another possibility is to manually define a threshold and define outliers the points outside of the percentiles specified by the threshold.

The results (figure 5.16) obtained with both methods are not better than the original estimates. This is because sometimes a result far from the other 11 values may be beneficial to take the mean nearer the exact value, while in other cases the

Figure 5.10: Aerial imagery of an area in which the TTA improves a lot its prediction of Population Density - 1

Figure 5.11: Aerial imagery of an area in which the TTA improves a lot its prediction of Population Density - 2

Figure 5.12: Aerial imagery of an area in which the TTA improves a lot its prediction of Population Density - 3

Figure 5.13: Aerial imagery of an area in which the TTA improves a lot its prediction of Population Density - 4

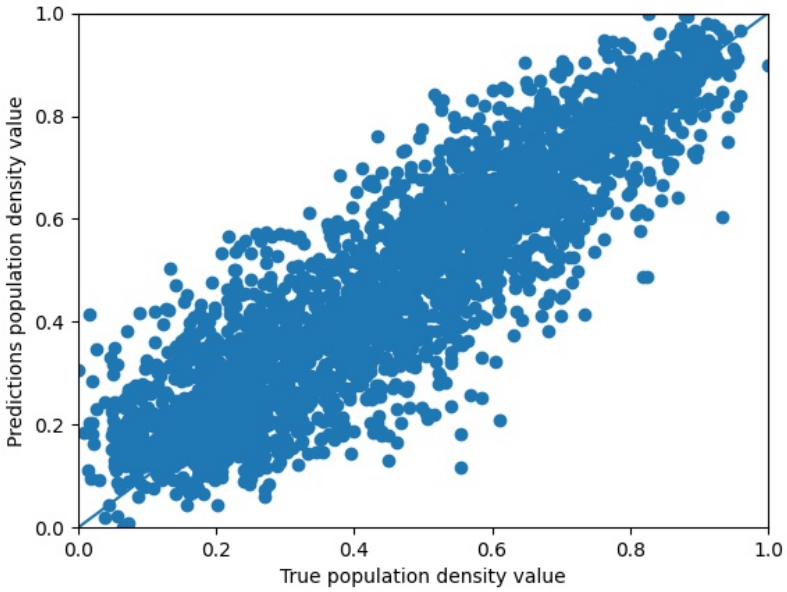

Figure 5.14: Scatter plot after TTA

opposite situation happens. At the end, a net effect is obtained, so that the value of the metrics R^2 does not change significantly with respect to the original one.

5.5 Database augmentation

In order to further improve the performance of the CNN, another step is performed: the database is doubled in order to train on more data and reduce overfitting. This is a problem that often arises when the database on which the neural network is trained is too small. The result is a model that fits well the data on which it has been trained, but it does not generalize well, that is, it is not able to make correct predictions on fresh data. Moreover, the performance of deep learning models continue to scale with the size of the training set.

To double the database, it was chosen to take each image and rotate it of 90° counterclockwise, and then the CNN is trained on both the original and rotated images. Figures 5.17 and 5.18 show an image of the database before and after the rotation.

This step will reduce the beneficial effects of TTA, as the model will be able to generalize better and so the 12 estimates will be nearer to each other. Anyway, what matters is the final result, that is expected to improve.

Now, the training requires twice the time as before, and the results are the

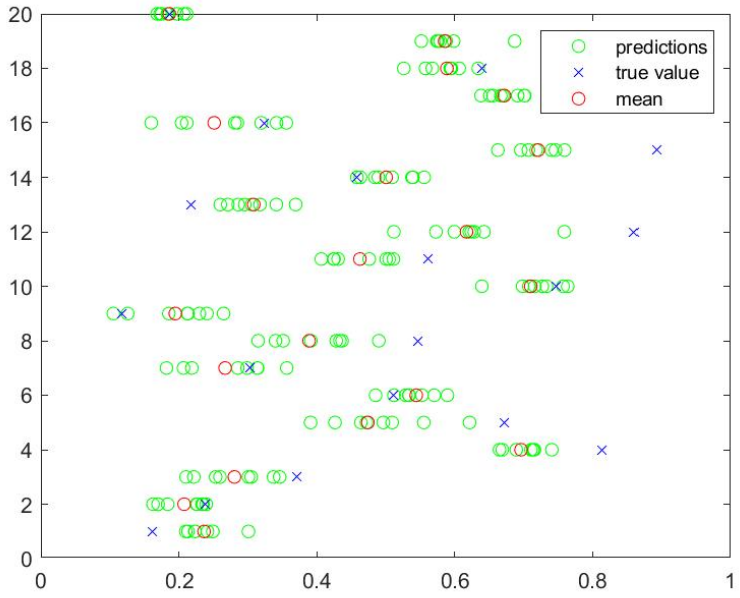

Figure 5.15: Predictions before filtering

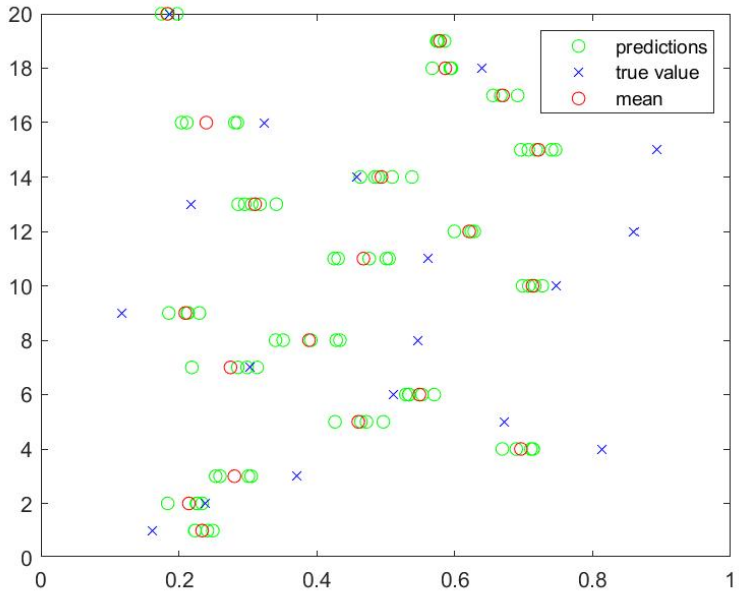

Figure 5.16: Predictions after filtering

Figure 5.17: Original image

Figure 5.18: Rotated image

following: the value of the metrics R^2 becomes about 0.82, and the scatter plot obtained is reported in figure 5.19.

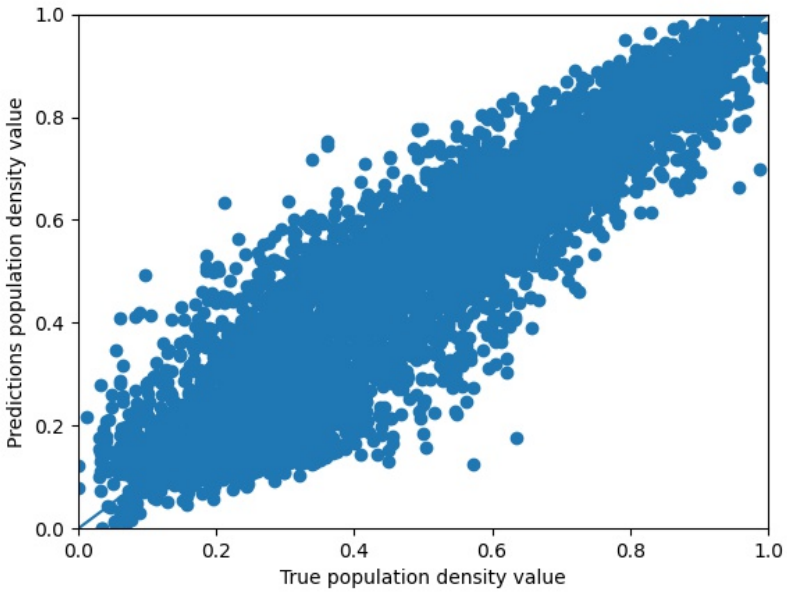

Figure 5.19: Scatter plot after database augmentation

5.6 Fine tuning

The distribution of the points in figure 5.19 is around the bisector. The goal now is to further reduce the thickness of the cloud. A tentative is done using fine tuning. This is a well known technique in machine learning that is linked to transfer learning. When a model trained for an application is used for a different task, as it has been done in this work, it may be helpful to tune the weights of the imported model. The more different are the two applications, the more beneficial fine tuning is. The network imported in this case, VGG16, is a model trained on images of natural objects, that are quite different from satellite imagery that constitute the database used. Fine tuning makes possible for the model to understand new information from the data used for the final application. Looking at the imported model, later layers are linked to features that are high-level and specific to the original task, so their weights have to be tuned. On the contrary, early layers should be freezed, because they are connected to low-level features like edges and shapes.

Tensorflow provides an easy way to perform fine tuning on a model: when the base model is uploaded, it is possible to set all the layers until a certain point as freezed, writing:

```
for layer in base_model.layers[:fine_tune_at]:
  layer.trainable =  False
```

fine_tune_at is a variable containing the number associated with the last layer to keep freezed.

In the case of VGG16, fine tuning was done on the layers after the 15^{th}. Due to the limits of computational power available on the AWS EC2 instance used, the training was done on the original database (composed of 21382 images), and not on the doubled one. This implies that the values to take as reference are $R^2 = 0.75$ and the scatter plot of figure 5.20. The results obtained show that the model performance gets worse: R^2 is now 0.68, and the scatter plot is reported in figure 5.21.

The fact that fine tuning does not work in this case is due to the dimension of the database, that is too small in comparison to the database on which VGG16 was originally trained. This in fact is called ImageNet, and it is composed of 15 million images that belong to 15000 different categories. The database used here, instead, is composed of 20382 images that are also very different from those of ImageNet.

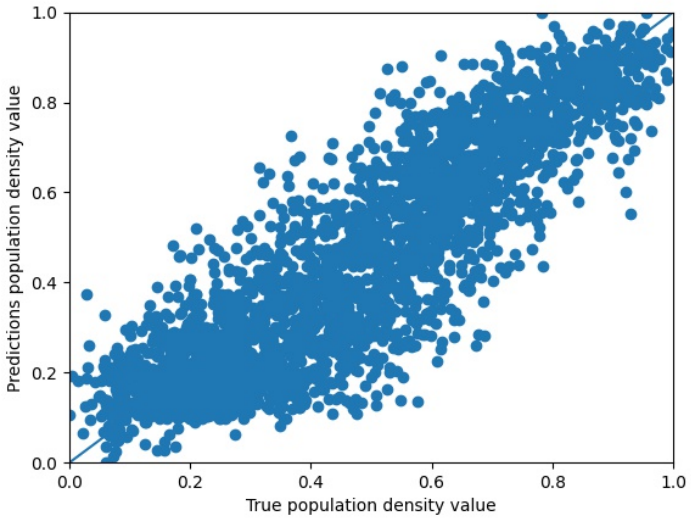

Figure 5.20: Reference Scatter plot for fine tuning

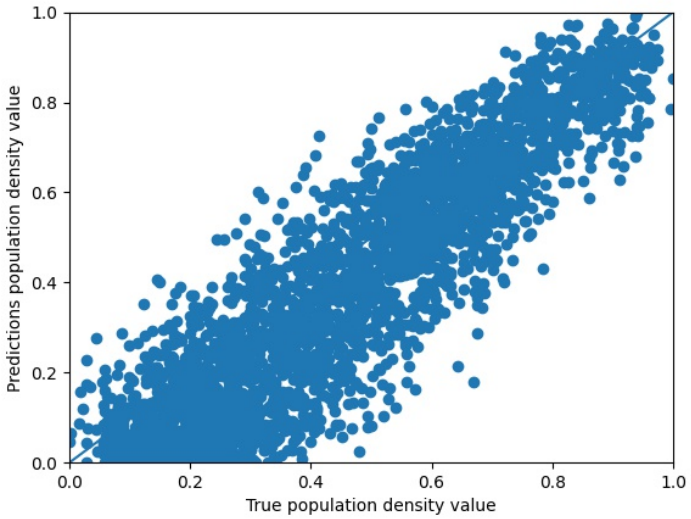

Figure 5.21: Scatter plot after fine tuning

5.7 Analysis of the biggest and smallest estimate errors

After the evaluation phase, an analysis of the estimated values of particular images can be performed, in order to better understand which are the problems still affecting the model. To do that, the 50 images linked to the biggest estimation errors and the 50 linked to the smallest estimation errors were saved, along with the estimated values and the true values. It is important to notice that, at this point, population density values have been modified using the logarithm function (as discussed in 5.2) and the MinMaxScaler operator. This is a data preprocessing method that maps the vector given as input into the range [0, 1] (figure 5.22), and in the context of machine learning it is used to scale the features. Scaling is an operation which purpose is to change the the range of a set of values, without changing their distribution. The optimization algorithms used in machine learning converge faster if features are scaled.

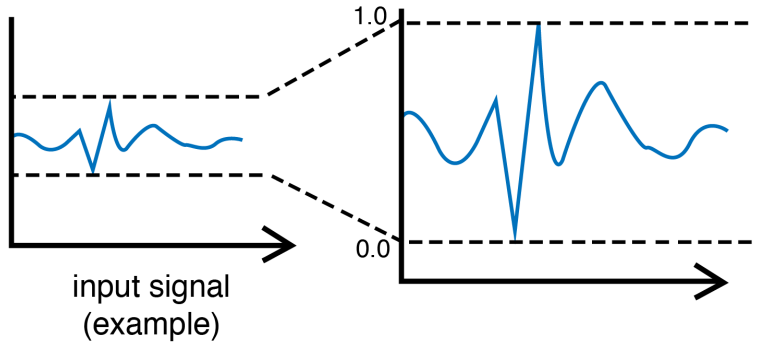

Figure 5.22: MinMaxScaler

The part of the database which is used for testing is composed of 5948 images. These are randomly chosen from the entire database, so in order to draw conclusions from the results of the algorithm, it have been run many times. For this reason, in the following part intervals are reported instead of a simple value.

The first interesting point to notice is that the model tends to underestimate the population density value. In fact, a percentage from 61% to 66% of the images were given a value bigger than the real one. Among the 100 images connected to

the biggest estimation errors, this is even more evident: the percentage is 76% to 84%. The fact that the algorithm underestimates the population density should be carefully taken into account, given that the context of this work is safety-related. Anyway, to better understand the problem it is convenient to look at some of the images that are underestimated the most. Four relevant example figures are reported in figures from 5.23 to 5.26. They are all related to a high value of the population density, meaning that a lot of people are present in the areas shown, but the model assigns a low population density to them. The reason why the CNN algorithm gives in output a small population density estimate is clear looking at the buildings present: some of them are tall, other have strange shapes. The height of the buildings is a problem, because satellite imagery flatten every object on the ground. The skyscraper in figure 5.26 is not recognized at all by the model. Moreover, in figures 5.24, 5.25 and 5.26 the shadows projected by high buildings hide relevant areas.

Figure 5.23: Underestimation example 1

Figure 5.24: Underestimation example 2

Figure 5.25: Underestimation example 3

Figure 5.26: Underestimation example 4

Another class of images that seems to cause problems to the model are those containing big industrial buildings. The population density in these areas is big due to the presence of workers, but from above they look like plain rectangles that cover most of the area, so that the model is not able to find relevant features. Two examples are shown in figures 5.27 and 5.28.

Figure 5.27: Underestimation example 5

Figure 5.28: Underestimation example 6

It is also important to look at some of the overestimated images. A first category is composed of images of crops, that are assigned by the algorithm to a small population density value, since no buildings are present, but the database labels them as highly populated. This can be seen as an error in the database, and the cause of it is how data are collected: phone cells are located everywhere, and they are not perfectly distributed, so a cell installed in the middle of crops catches signals from its surrounding area that can include highly populated buildings. Two areas that have this problem are reported in 5.29 and 5.30.

Figure 5.29: Overestimation example 1

Figure 5.30: Overestimation example 2

Another class of overestimated images is composed by satellite imagery over residential areas where there are buildings and trees. At the time when data were taken from phone cells (that is, 9 a.m.) few people were present in the area, resulting in a very low population density. The model, anyway, predicts high population density values due to the presence of relevant elements. Two representative images are 5.31 and 5.32.

An important feature that some of the images that are estimated with big errors have in common is the presence of colors that are not common in the database. In fact, most of the images are made of colors like green and brown (due to crops, trees and buildings). Colors like white, blue and red, instead, are not so common among the images, and this causes the model to fail the estimation. Two examples of images belonging to this class are shown in figures 5.33 and 5.34.

The last part of this analysis involves the images for which the estimation of the population density is most precise. Looking at the 50 better predicted images, it is

Figure 5.31: Overestimation example 3

Figure 5.32: Overestimation example 4

Figure 5.33: Color example 1

Figure 5.34: Color example 1

possible to conclude that in all of them there are a lot of relevant objects like trees, buildings and streets. The relation is simple: it is fundamental for the CNN to find elements already found during the training to make correct predictions. The four images that have been better predicted with the best model obtained are shown in figures 5.35 to 5.38.

Figure 5.35: Best prediction 1

Figure 5.36: Best prediction 2

Figure 5.37: Best prediction 3

Figure 5.38: Best prediction 4

5.8 Direct estimation of the risk

The computation of the risk is complex, and it is described in [1]. An alternative approach is proposed here, with the goal of facilitate the computation. The method is inspired by the promising results obtained in the population density estimation. Basically, the idea is to use the same CNN structure to implement a model that

is able to estimate directly the risk, avoiding the need of building and merging different layers. The values of the risk used as database for training are those obtained with the probabilistic approach. These data also constitutes the ground truth of the testing phase, and it is important to note that they are strongly dependent on the population density.

The results are encouraging: the value of R^2 is 0.78, and the TTA technique is still beneficial for the model. The scatter plot is reported in figure 5.39. An

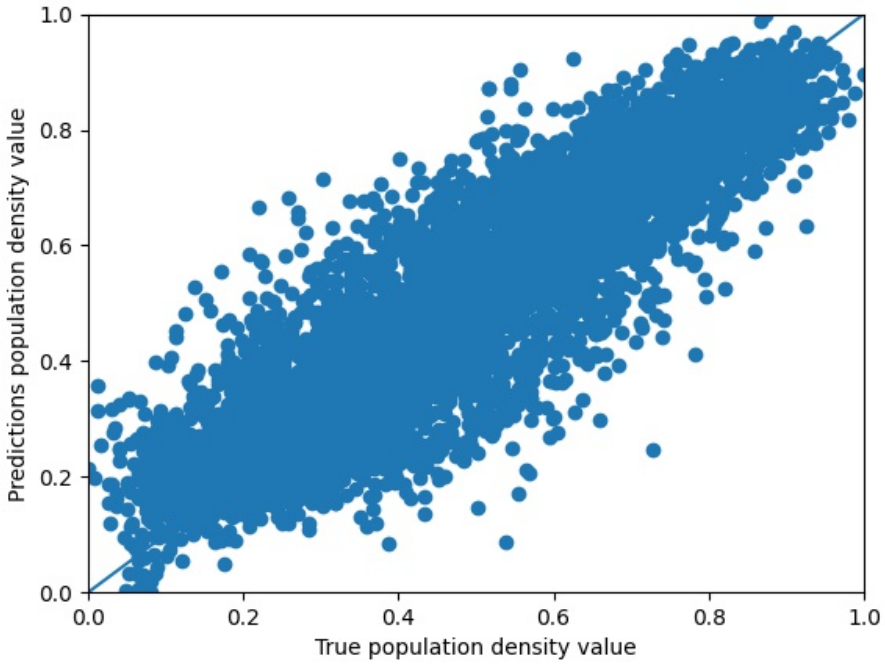

Figure 5.39: Risk estimation scatter plot

analysis similar to that done before shows that the algorithm tends to slightly underestimate the risk: the percentage of underestimated images is around 58%.

Looking at the images for which the risk was better estimated, the same problems discussed before are still present, with the biggest errors being related to images of industrial building and crops (figures 5.40 and 5.41).

5.9 Population density maps

In the last two sections of this chapter the main results obtained are presented. In fact, the goal of the thesis was to obtain population density maps and the risk

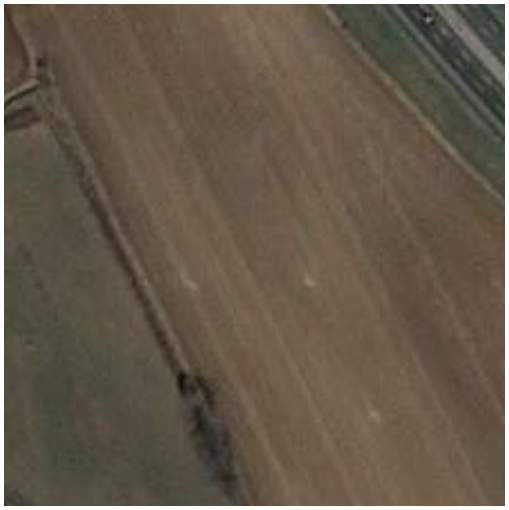

Figure 5.40: Risk error example 1

Figure 5.41: Risk error example 2

maps using a machine learning-based approach. The best way to see how the model works in practice is taking an area, obtaining the maps for which the models have been trained and comparing them with the maps obtained with the true values. This process is also useful to understand the environments on which the model is able to make good predictions and those on which the model does not work well.

The color scale that is used to build the maps is shown in figure 5.42. The red color is associated with the most populated areas, while violet corresponds to the less populated ones (and the same holds for the risk). Different maps have been created for each area: two for the population density and three for the risk. For each area, the population density maps share the same scale, and the same for the risk maps. This means that the red color is associated, for example, to the maximum value of the true and estimated population density. This choice makes possible to understand, from these maps, if the model is able to reproduce the distribution of population.

Three different areas have been selected to make this analysis. The first one includes almost all the Valentino park and the adjacent residential area, the Po river and its shores shown in figure 5.43.

The true and estimated population density maps of this first area are shown in 5.44 and 5.45. Some important conclusions about how the algorithm works can be drawn from these maps. First, the estimated population density and the real one are distributed similarly: the residential area on the left of the maps is associated with high values (red and yellow), while the right side is characterized by low values. It is also visible the trend that the model has to underestimate

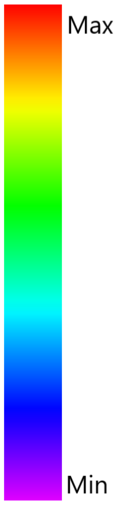

Figure 5.42: Color scale

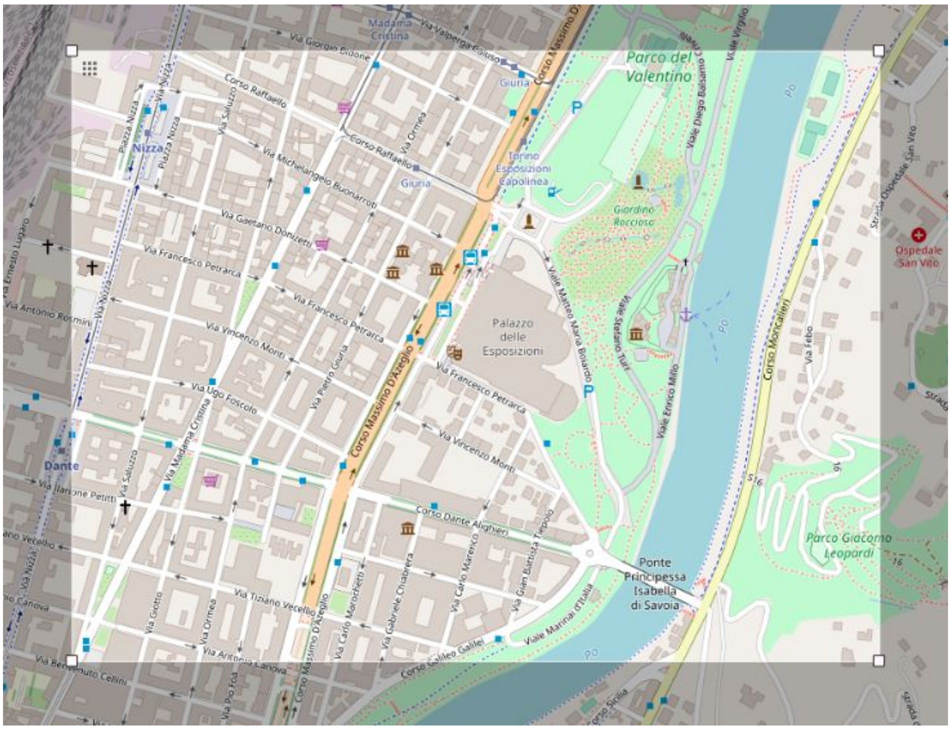

Figure 5.43: Test area 1

the population density: the colors are all shifted towards those representing a low population density.

To better understand these behaviours, a bigger area that is similar and adjacent

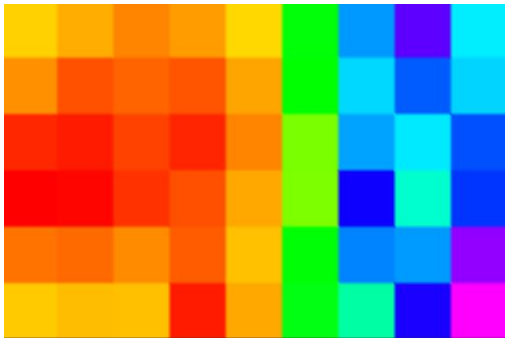

Figure 5.44: True population density map - Area 1

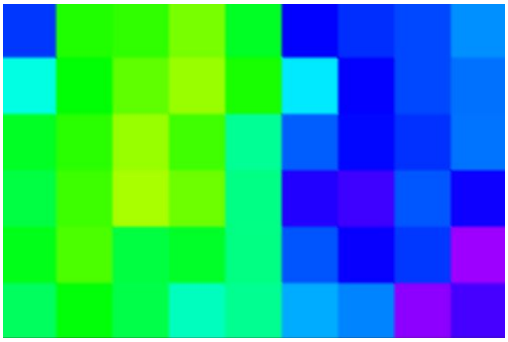

Figure 5.45: Estimated population density map - Area 1

to the first one is taken: figure 5.46. As before, it is visible the bias towards underestimation of the population density that the model shows. This map can be divided into 3 parts: the first one corresponds to a residential area (San Salvario district) on the left side of the river, the second one is the Po river and its shores, and the last one is the hilly area on the other side of the river. The algorithm is able to distinguish these three areas and to categorize them quite correctly. Some problems are still present: the red area in the bottom left part of the map is not distinguished from the surrounding area. This happens because from above the area does not show big differences (the buildings are more or less similar in this area). Anyway, the use of these buildings may differ a lot: there are apartments, offices, shops and also a hospital, and recognizing them from the satellite view is an hard task.

The population density maps are shown in figures 5.47 and 5.48.

The third and last area on which the model has been tested is different from the previous two: it is a more homogeneous area, corresponding to Barriera di Milano district (figure 5.49). This includes residential buildings and the related serviced like schools, shops and restaurants. The results (figures 5.50 and 5.51) confirm the fact that population density is underestimated. Moreover, the distribution is less precisely reproduced than before, with the most populated area that is wrongly predicted as sparsely populated. The right side of the area, that is less populated, is predicted with a higher accuracy.

The discrepancy between estimated and real values is due to the difficulty of the CNN to understand the type of buildings (residential, offices, markets, etc.) in the area. In fact, using a top view image it is hard to estimate the population density with an high accuracy. However, the CNN shows promising results and it is able to distinguish populated areas and non-populated areas, as well as estimating the distribution of population as demonstrated by the tests above. Moreover, the CNN

Figure 5.46: Test area 2

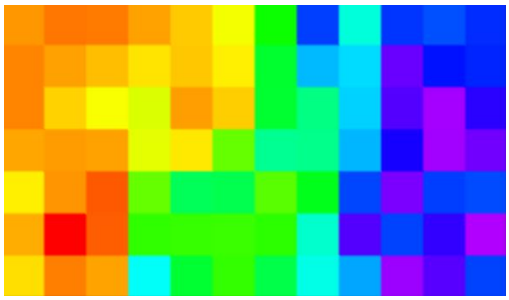

Figure 5.47: True population density map - Area 2

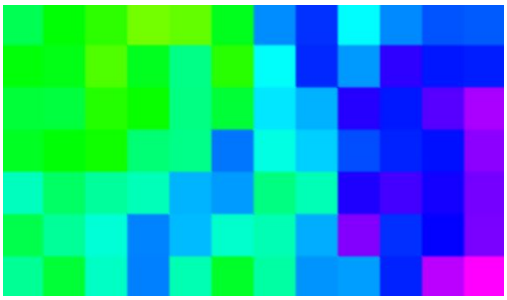

Figure 5.48: Estimated population density map - Area 2

tends to underestimate the population density because it is trained considering the whole Turin municipality and, then, evaluating also areas with a low population density. As a consequence, the CNN tends to estimate an average population density. In fact, the tests shown are focused in the city center where we have the higher population density distribution in the city.

5.10 Risk maps

For each area chosen for the test, three different risk maps have been generated. The first one is obtained using the probabilistic approach, using for the population

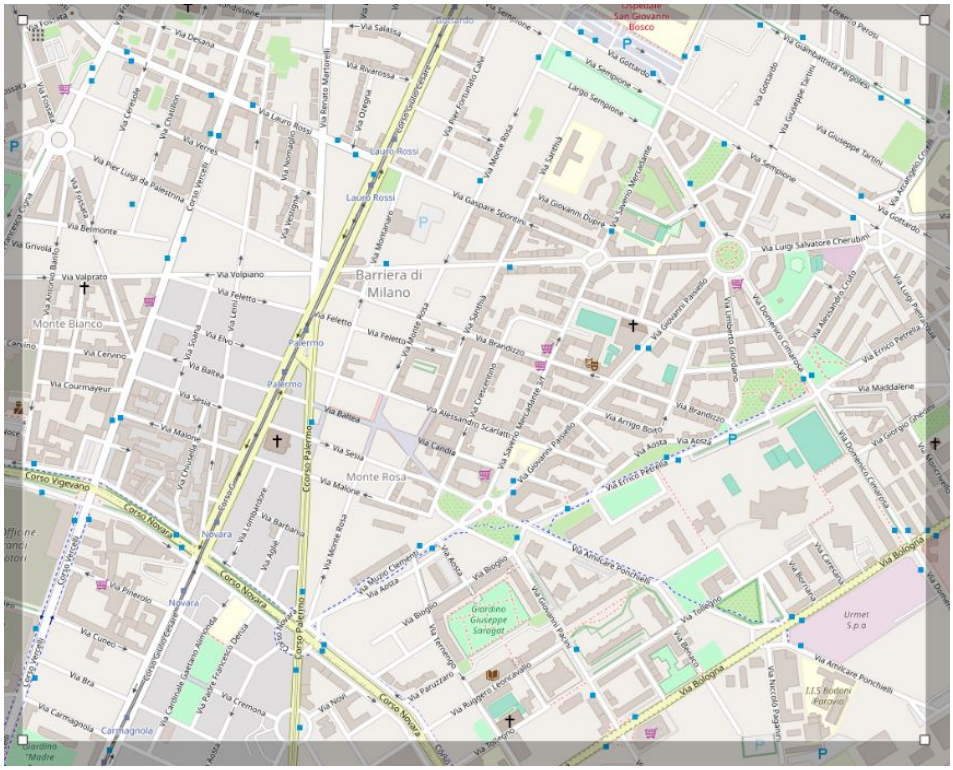

Figure 5.49: Test area 3

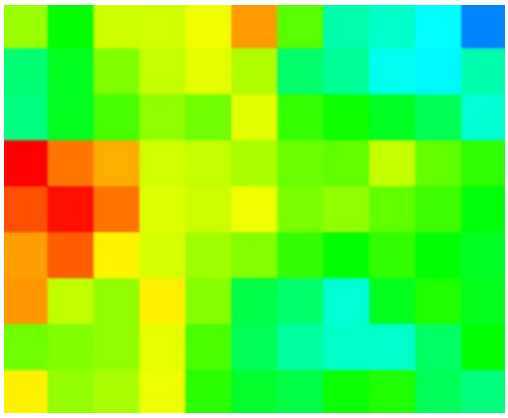

Figure 5.50: True population density map - Area 3

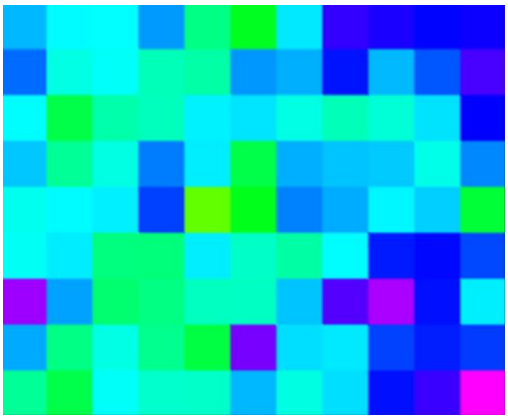

Figure 5.51: Estimated population density map - Area 3

density layer the correct values on which the CNN has been trained. The second map is obtained through the same approach, but the estimated population density

values have been used in place of the correct ones. For the third map, the risk computed via the direct approach is used (as discussed in section 5.8). The results of the first area are shown in figures 5.52 to 5.54.

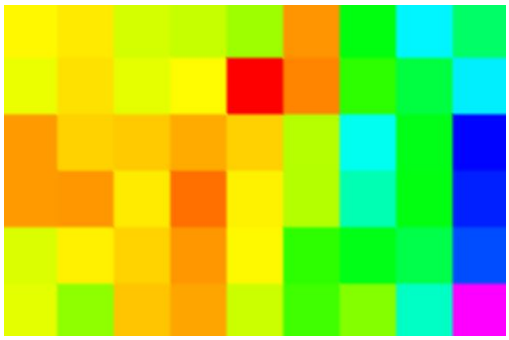

Figure 5.52: Risk map - Area 1

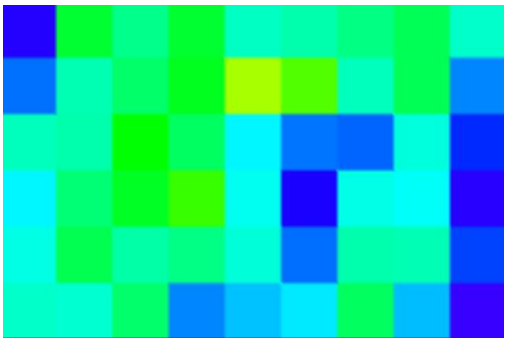

Figure 5.53: Estimated risk map - Area 1

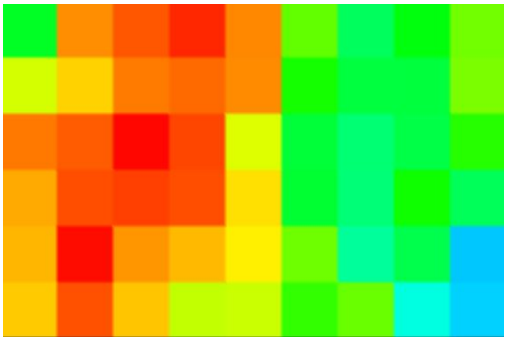

Figure 5.54: Directly estimated risk map - Area 1

The risk computed using the population density estimated with the CNN has a distribution similar to the reference map. In fact, the risk is low in the right half, and higher (even if it is underestimated) in the left half of the map. The direct estimation of the risk also reproduces well the distribution of the true map, but in this case there is an overestimation problem. Anyway, both approaches lead to promising results.

The second area, similar but bigger than the first one, reveals similar results (figures 5.55 to 5.57). As before, the direct approach shown an overestimating trend, while the map obtained with the estimated population density layer underestimates the risk. Again, both the distributions are similar to the original one.

In the last area tested (results in figures 5.58 to 5.60), the map obtained with the estimated population density is very similar to the original one, considering the

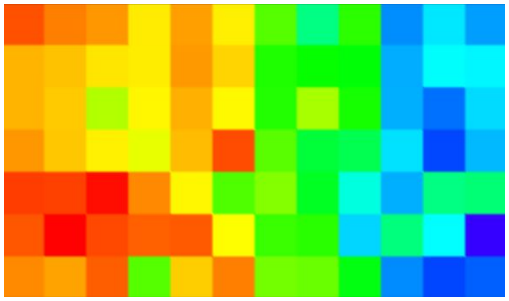

Figure 5.55: Risk map - Area 2

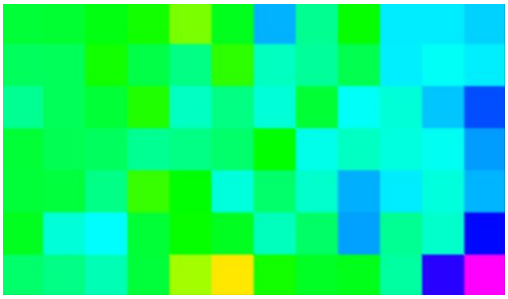

Figure 5.56: Estimated risk map - Area 2

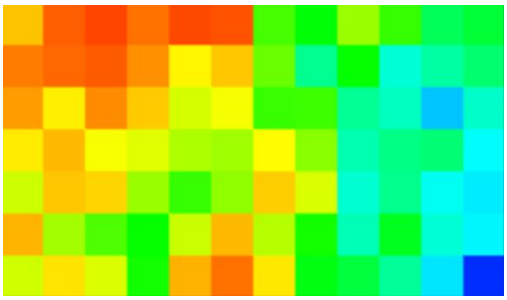

Figure 5.57: Directly estimated risk map - Area 2

underestimation problem. In fact, the areas with the highest population density are red in the original map and yellow/green in the estimated map, while green areas are turned into blue ones. The directly estimated map, instead, shows that almost every square has the same value of the risk. This because, as discussed before, this area from above looks quite homogeneous.

After the analysis of the results, some conclusions can be drawn. First, the risk obtained via the standard probabilistic approach with the population density obtained using the CNN model has a problem of underestimation. This comes from the fact that population density is underestimated, and then it is used to compute the risk. The values of the risk are not correctly reproduced, but the distribution of the different areas is correct (this is the same problem observed for population density maps). The direct approach, instead, shows some problems in areas like the third one analyzed. In fact, the model is not able to distinguish the risk in some situations, while a correct distribution is replicated for areas containing different scenarios, like the first two test cases.

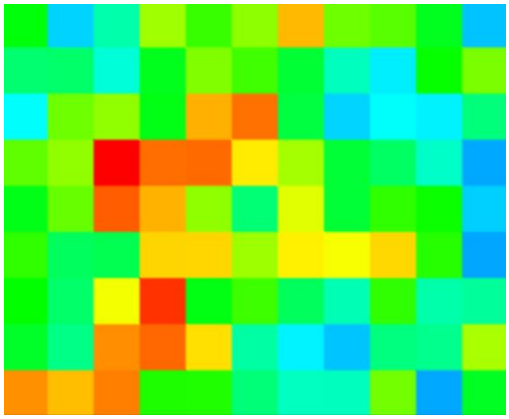

Figure 5.58: Risk map - Area 3

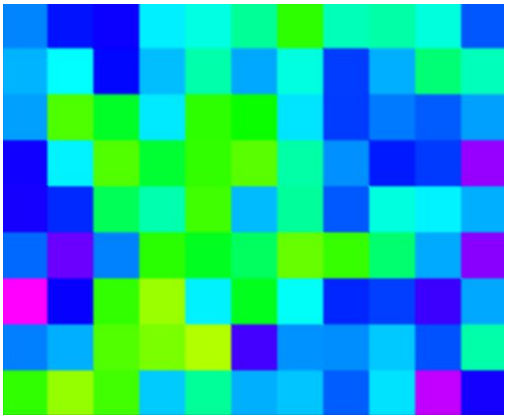

Figure 5.59: Estimated risk map - Area 3

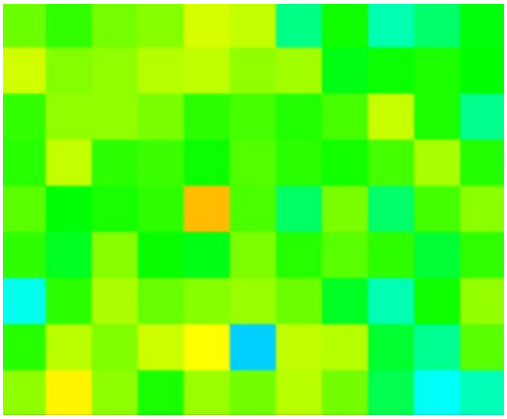

Figure 5.60: Directly estimated risk map - Area 3

Chapter 6

Conclusions and future work

The main goal of this thesis was to train a CNN model able to estimate the population density of an urban area, starting from the satellite imagery of that area. This is a novel approach in this field where traditionally census data were used. The final results show that the model is able estimate population density with a good degree of accuracy. The database used has some limitations, due to both the images and the numerical data that form the ground truth. Moreover, images and population density data refer to a different date and time, and this affects the precision of the final results: the distribution of people in a city changes dynamically and continuously. This issue can be partially solved using data from phone cells to build a database, considering that this would give a dynamic picture of how people are distributed. In fact, a possible future development of the present work is to build different CNN models, all with the same structure but trained with data collected at different times in a day and also in different days in the week.

Several machine learning techniques have been employed to improve the performance of the CNN model. First, the VGG16 model was imported with its weights already tuned to save time. Fine tuning turned out to be not useful in this case, due to the size of the new database, that is too small. Test time augmentation, instead, boosted the performance of the algorithm. The database was also doubled to train the model better, improving its generalization capability. This was possible considering that satellite imagery are particular, in that they don't have an orientation other than the conventional one, and the information is equally distributed in the whole image.

The main advantage of using a machine learning-based approach to obtain population density maps is the cost: the process of obtaining census data is expensive and long, and it involves many people. A CNN model instead, once its

weights have been tuned, can be used to obtain the population data of wide areas in a short time.

It is also important to remember the goal of the maps obtained: they are one of the layers used to form risk maps, that are used to run a risk-aware path planning algorithm for UAVs. Census data provide a fine and precise description of the population density, that is necessary for many application. In this context, knowing the exact number of people living in an area is not so important. What is important, instead, is to have a model that is able to correctly distinguish highly populated areas from areas where there are few people, on which it is safe to fly over. The model obtained is able to correctly understand the differences between residential areas and unpopulated areas, as attested by the tests performed (figure 5.51). When it comes to distinguish areas at a finer level, as in the last test, the model reveals its limits.

For the development of this work, a possible improvement may be obtained including in the database other cities. This would result in a bigger database, on which the model can be trained with better results. It is also interesting to see how a model trained on multiple cities performs when it is used to predict over a different city, not included in the database.

Last, the results obtained using a direct approach to obtain risk maps are promising. The computation of the risk with the probabilistic approach is complex, as it requires the definition of several layers and their combination. Using a ML-based approach would dramatically reduce the time required to build a risk map, with all the advantages discussed above. Anyway, it is necessary to have a database of high quality and big enough to train the network: as of now, the probabilistic approach still gives the best results.

Acknowledgements

Bibliography

[1] Stefano Primatesta, Giorgio Guglieri, and Alessandro Rizzo. «A risk-aware path planning strategy for UAVs in urban environments». In: *Journal of Intelligent & Robotic Systems* 95.2 (2019), pp. 629–643 (cit. on pp. 2, 49).

[2] *Milestones and Moments in Global Census History*. URL: `https://www.prb.org/resources/milestones-and-moments-in-global-census-history/` (cit. on p. 3).

[3] Adrian Albert, Jasleen Kaur, and Marta Gonzalez. *Using convolutional networks and satellite imagery to identify patterns in urban environments at a large scale*. 2017. arXiv: `1704.02965 [cs.CV]` (cit. on p. 5).

[4] Caleb Robinson, Fred Hohman, and Bistra Dilkina. *A Deep Learning Approach for Population Estimation from Satellite Imagery*. 2017. arXiv: `1708.09086 [cs.AI]` (cit. on p. 5).

[5] Patrick Doupe, Emilie Bruzelius, James Faghmous, and Samuel G Ruchman. «Equitable development through deep learning: The case of sub-national population density estimation». In: *Proceedings of the 7th Annual Symposium on Computing for Development*. 2016, pp. 1–10 (cit. on p. 5).

[6] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: `1409.1556 [cs.CV]` (cit. on p. 5).

[7] Rex W Douglass, David A Meyer, Megha Ram, David Rideout, and Dongjin Song. «High resolution population estimates from telecommunications data». In: *EPJ Data Science* 4 (2015), pp. 1–13 (cit. on pp. 5, 7).

[8] Tobias G Tiecke et al. «Mapping the world population one building at a time». In: *arXiv preprint arXiv:1712.05839* (2017) (cit. on p. 5).

[9] *Population Density Maps*. 2021. URL: `https://dataforgood.fb.com/tools/population-density-maps/`.

[10] Pierre Deville, Catherine Linard, Samuel Martin, Marius Gilbert, Forrest R Stevens, Andrea E Gaughan, Vincent D Blondel, and Andrew J Tatem. «Dynamic population mapping using mobile phone data». In: *Proceedings of the National Academy of Sciences* 111.45 (2014), pp. 15888–15893 (cit. on p. 7).

[11] Chaogui Kang, Yu Liu, Xiujun Ma, and Lun Wu. «Towards estimating urban population distributions from mobile call data». In: *Journal of Urban Technology* 19.4 (2012), pp. 3–21 (cit. on p. 7).

[12] Paul Sutton, Dar Roberts, Chris Elvidge, and Henk Meij. «A comparison of nighttime satellite imagery and population density for the continental United States». In: *Photogrammetric engineering and remote sensing* 63.11 (1997), pp. 1303–1313 (cit. on p. 7).

[13] Paul Sutton. «Modeling population density with night-time satellite imagery and GIS». In: *Computers, Environment and Urban Systems* 21.3-4 (1997), pp. 227–244 (cit. on p. 7).

[14] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536 (cit. on p. 11).

[15] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. «Handwritten digit recognition with a back-propagation network». In: *Advances in neural information processing systems*. 1990, pp. 396–404 (cit. on p. 12).

[16] Grace W Lindsay. «Convolutional neural networks as a model of the visual system: past, present, and future». In: *Journal of cognitive neuroscience* (2020), pp. 1–15 (cit. on p. 12).

[17] Sebastian Raschka. *Python machine learning*. Packt publishing ltd, 2015.

[18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (cit. on p. 17).

[19] Peter Goldsborough. «A tour of tensorflow». In: *arXiv preprint arXiv:1610.01178* (2016) (cit. on p. 23).

[20] Wikipedia contributors. *Coefficient of determination — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-June-2021]. 2021. URL: `https://en.wikipedia.org/w/index.php?title=Coefficient_of_determination&oldid=1024391042` (cit. on p. 32).

[21] Divya Shanmugam, Davis Blalock, Guha Balakrishnan, and John Guttag. «When and why test-time augmentation works». In: *arXiv preprint arXiv: 2011.11156* (2020) (cit. on p. 36).

[22] *Detect and remove outliers in data - Matlab*. URL: `https://www.mathworks.com/help/matlab/ref/rmoutliers.html` (cit. on p. 37).