Tackling Uncertainty in Mobile Computer Vision Applications

Ana Cecilia López González
Ana Tatiana López González

Supervisor: Prof. Piero Fraternali

Department of Electronics, Informatics and Bioengineering
Polytechnic University of Milan

This dissertation is submitted for the degree of
Master of Science

April 2018

"Mischief Managed".

Abstract

In this work we present a mobile outdoor augmented reality application, which uses data coming from the sensors of the mobile device to identify mountain peaks in the skyline, analyze the *uncertainties* present in this type of applications and focus on how they can be overcome so as to make the *digital augmentation* of the physical world meaningful. Although uncertainties come in a wide range of possibilities, here we address uncertainties as imperfect information about what the user is seeing, namely wrong estimates of the phone orientation and the presence of objects occluding the skyline. Leveraging on recent advances in Computer Vision algorithms and significant progress in object class recognition using Deep Learning techniques, we propose a hybrid model that is able to perform real-time accurate classification of skyline pixels in images captured with a real outdoor augmented reality application; the proposed approach combines two binary classifiers in cascade, one for skyline detection and the other categorically tailored for occlusion detection, so as to provide an accurate alignment between the panorama the user is seeing through the camera and the virtual panorama computed from the digital terrain model of the corresponding location. The final combined model respects the efficiency constraints associated with low-powered mobile devices and exhibits a good balance between accuracy, memory consumption and runtime execution overhead. In addition to the development of the occlusion management module, the thesis also reports on the re-design of the sensor capture sub-system for iOS, which has been used to explore the uncertainties that arise in outdoor mobile augmented reality applications and set the requirements for the improved Computer Vision approach.

Sommario

In questo lavoro presentiamo una applicazione mobile di realtà aumentata, che usa dati dei sensori di dispositivi mobili per identificare vette di montagne sull'orizzonte, analizziamo le *incertezze* presenti in questo tipo di applicazioni e studiamo come esse possano essere corrette per rendere il *digital augmentation* del mondo fisico più significativo. Sebbene le incertezze possono avere diverse forme, noi ci focalizziamo su incertezze intese come informazioni imperfette su quello che l'utente sta vedendo, specificamente, errori nella stima dell'orientamento del telefono e la presenza di ostacoli all'orizzonte. Sfruttando i recenti sviluppi nel campo di Computer Vision e i progressi nell'identificare classi di oggetti con tecniche di Deep Learning, presentiamo un modello ibrido in grado di classificare accuratamente e in tempo reale i pixel sull'orizzonte delle immagini all'aperto proveniente dalle applicazioni di realtà aumentata; l'approccio descritto contiene due classificatori binari in cascata, uno per l'identificazione dell'orizzonte e l'altro addattato alla identificazione di ostacoli, per garantire un corretto allineamento tra l'orizzonte che l'utente sta vedendo attraverso la camera e il panorama virtuale generato dai modelli digitali di elevazione della corrispettiva zona. Il modello finale rispetta i requisiti di efficienza richiesti per dispositivi mobili di bassa potenza e ottiene un buon equilibrio tra l'accuratezza, consumo di memoria e l'overhead del tempo di esecuzione. Oltre allo svilluppo del modulo di gestione delle occlusioni, la tesi descrive il re-design del sottosistema del sensore per iOS, che è stato usato per esplorare le incertezze caratteristiche delle applicazioni di realtà aumentata all'aperto e definire i requisiti per migliorare l'approccio Computer Vision.

Acknowledgements

All of us face victories and defeats, is part of life, it is more important though, to be able to recognize and celebrate victories, that although few, keep our engines running. We are more than happy to consider this one as a victory, so inevitably, as the occasion demands, there are many people to thank, some were next to us, day by day, others, on the other side of the pond. For that, this work is dedicated to her, our greatest source of inspiration, our Mother, the one who kept telling us that we could do it all, and if not, at least we could try, we blame you Blanquita, we believed in you, and here we are, seven thousand miles away and about to finish one of our favorite adventures, with the biggest realization so far, the older we get, the smarter you become.

This has been quite a ride, with many ups and downs, a journey in which we have learned so much, and for that we thank you Darian, we know we are a little bit moody and for that, we apologize. We are extremely grateful for the time, effort, and enthusiasm you put forward in our work together.

And finally, we don't know where to begin to start thanking you Prof. Fraternali, you have been the ideal adviser, hands off when the sailing was smooth and deeply supportive when it wasn't. The completion of this thesis couldn't have been possible without your expertise, watching you work has been a great joy of our time in the lab. You have honestly made a great impact on our careers. We thank you for making this experience, the best experience it ever could be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Augmented Reality (AR) promises to change the way information is conveyed, creating an impact in military, industrial, medical, and entertainment domains. Outdoor applications, for instance, represent an interesting challenge for AR in that they have a *world* full of objects that could be augmented; nonetheless, having a *world* to augment also means having a *world* full of uncertainties, especially, when little is known a priori about the real world we wish to augment.

Outdoor applications heavily rely on information provided by mobile devices; exploiting the sensor data and motion patterns is, therefore, the way to go. Nevertheless, if we only take into account the position and orientation signals, we are neglecting an important source of information, *what the user is actually seeing*, more precisely, the user's interest. To provide accurate information, we need to address the limitations of the aforementioned approaches, not only by considering the stream data received from the camera, but also tackling the specific uncertainties associated with it.

Uncertainty comes in many shades, however, in an outdoor domain the primary source of uncertainty is the constant rearrangement of objects composing the scene due to change of viewpoints. These positional uncertainties directly derive from objects -*partially or completely*- covering the user's scene of interest; hence, handling occlusions is the main challenge for providing accurate matching between the image of the physical world and the virtual generated model of the world that the application exploits to perform augmentation. Since the detection of objects composing the scene need to be computed per frame and achieved in real-time, we focused on solving the problem of identifying occlusion of the scene while respecting efficiency

constraints associated with low-powered mobile devices.

Despite the remarkable progress of recent object class recognition systems, partial occlusion remains a major challenge to state-of-the-art detectors; constituting one of the main barriers to accurate recognition results in complex scenes. Furthermore, handling real-time unconstrained images which are ubiquitous in outdoor augmented reality applications, is a fundamental problem in computer vision, given that real world scenes usually contain more than one object and it is very likely that some parts of an object are occluded by other objects in the scene; even more, the combinatorial nature of occlusion patterns poses an extra significant difficulty. Furthermore, object recognition becomes more challenging when dealing with moving cameras in real-time applications. In spite of all the recent developments and even when significant progress has been achieved in these tasks, the long-standing problem of partial occlusion is still facing a number of challenges, one of which is the fast and accurate detection in low-power devices without having any detrimental effect on memory fingerprint and battery consumption.

After carefully analyzing the potential issues affecting *real world* scenes, we introduce what we call *the occlusion problem*, consisting in the fact that occluders -*any object occluding the skyline*- are treated as second class citizens, when they should be explicitly considered in both problem and solution, and not simply treated as yet another source of noise.

In this thesis we present an outdoor mobile application for mountain peak detection that not only accounts for sensor-based data retrieved by the device, but also uses images captured by the camera to make an *accurate* alignment between the skyline the user is seeing at and the terrain, represented by a virtual skyline computed from the the "Digital Elevation Model of the Earth" (DEM), the position of the user and the orientation of the device.

To avoid placing augmented content onto objects that are not visible in the current camera frame, we need to acknowledge the presence of possible occluders in the horizon and define an architecture capable of identifying occluded peaks in phone camera frames, so as to accurately display augmented content only when relevant.

The proposed approach explicitly represents occlusions by leveraging a fully convolutional neural network that predicts whether occlusions exist in the skyline shown in the actual camera frame.

As a starting point to understand the current image analysis capability of PeakLens [10], we performed several experiments to assess the model presented in [50] -hereinafter referred to as *baseline*-, which is included in a Computer Vision module that implements the skyline detection in the deployed version of PeakLens.

Based on the results of such analysis, we extend the Computer Vision module so as to handle more informative input data to better understand the relationship among local spatial features. In addition, we model the network to learn not only the appearance of different occlusion patterns but features that directly represent mountainous skylines as well. This network is later used in a cascade hybridization manner along with the network presented in [50]. The final combined system is able to perform classification on images captured in a real outdoor scenery and omni-directional AR mobile application, using only the information from image sequences retrieved by the camera phone.

As a result of the analysis presented in this dissertation the main contribution of the thesis is twofold:

First, we have developed an iOS outdoor AR mobile application for mountain peak detection that takes fully advantage of Apple's standardized hardware and software with the aim to provide not only a meaningful user experience but hit more accurate readings by leveraging on the device's on-board sensors. Using specific operating system frameworks, the application effectively tracks the user's position in space in real-time by means of the Device Motion Tracking Service, and collect events coming from the accelerometer, gyroscope and magnetometer, in this way we are able to recognize motion with a significant degree of accuracy. For AR applications in which interacting with virtual objects is innate, effective motion tracking is key, thus, we account for the distance between the motion sensors and cameras as well as the device's place in the world by relying on the Sensor Fusion Algorithm to remove bias such as the effect of gravity. Once the pose is estimated, a panorama based on the acquired user's position is obtained, later, an alignment *wrt* the orientation and field of view of the device is performed, consequently, mountain peaks are located in the refined panorama. We collect data at every frame so as to compute the tilt of the device to efficiently update the position of the augmented content that will be presented to the user. Accordingly, the augmentation experience is successfully provided by placing on the screen tags with pertinent information about the mountain peaks the user is seeing through the camera.

As one of the main aims in this stage was to extend the existent Computer Vision module used in [10], and guarantee that it will work indistinctly in both iOS and Android devices, an in-depth study was conducted to identify the mandatory means in order to use framework-specific data, such as the device's attitude (in any of its mathematical representations: Euler angles, quaternions or rotation matrices), indistinctly of the operating system. To provide a standard representation of the device's true orientation, a series of procedures were carried out on the mathematical representations of the device's attitude, to standardize the reference coordinate system between both platforms.

Promising results were obtained and we succeeded at locating and rendering peaks into the camera view, providing the user with a complete AR experience. Through the analysis and development of this application, we explored the uncertainties that arise in outdoor AR applications such PeakLens and the implications involved in relying solely on sensor-based data.

Second, we have used *Deep Learning* models to address the occlusion problem and, consequently, enhanced the mobile application capabilities. In particular, we have proposed a combined model tailored to skyline and occlusion detection, composed by a skyline detector [50] and a fully convolutional network devised for occlusion handling. The strength of our contribution lies on this model's acquired knowledge that let it discriminate accurately skylines from occlusions. Significantly, our technique shows a clear advantage over the skyline detector presented in [50] as we are able to improve their performance by almost **15%** in the detection of occlusions over a large dataset of challenging mountainous images depicting all sorts of uncertainties such as sunlight, clouds, bushes, buildings and people occluding the skyline. These results support the validity of our approach to tackle occlusions in the skyline.

The sequential nature of the proposed combined model results in less than 70% increase in the execution time in mid-range devices, as such, we are able to process around 3 images per second. We stress the fact that this rate is totally compatible with the usability requirements of a real-time AR application.

In this sense, we evidence that learning a deep ConvNet effectively handles the occlusion problem inherent to outdoor AR applications. Furthermore, throughout the course of this study, we have experimented with different sampling techniques to address the imbalanced nature of the occlusion problem, additionally, we have demonstrated that image information

arranged in non-conventional shapes (columns instead of squares) can provide a deeper context to the network and consequently, make it learn to discriminate among features that better define occlusions.

The rest of the thesis is structured as follows:

- In the following chapter we present an extensive literature review of the state-of-the-art approaches to tackle uncertainties in outdoor augmented reality applications.
- In Chapter 3 we explain the task of the non-intelligent system and discuss the limitations that are involved.
- In Chapter 4 we present the proposed heuristics in the intelligent system and we give a detailed explanation of the various components conforming it.
- In Chapter 5, we first describe the design of the experiments and then demonstrate the experimental results.
- Finally, in the last chapter, we summarize our work and discuss future extensions and improvements.

Chapter 2

Related Work

In this section we report the current state of the art of the areas involved in this work, more specifically, we are interested in: *Augmented Reality* applications, the *Computer Vision* problem and the *Analysis of Mountain Images*.

2.1 Augmented Reality

Augmented reality is the integration of virtual content into the user's environment in real-time, its main objective is then, to enhance the user's current perception of reality by superimposing virtual objects in such a way that the virtual objects appear to be part of the real world [96]. AR enhances the physical world adding graphics, sounds and haptic feedback, in that sense, it differs from *Virtual Reality*, which is purely synthetic and has no direct correlation with the natural environment the user inhabits [44].

Traditionally, AR has been viewed as a wearable technology, yet the greatest uptake recently has been on mobile platforms, such smartphones and tablets, due to increase in computational power and a sort of hardware standardization –*camera* and *location and motion sensors*– to run AR applications. AR has evolved enormously since its beginning in 1968, when a head-mounted display showed, *now* rudimentary, computer-generated graphics [93]. Today's trend is to include Machine Vision, object recognition and gesture recognition technologies into AR applications so as to let them not just *see* the world, but understand it; therefore, Computer Vision has become essential for AR growth.

2.1.1 AR for Outdoor Applications

Outdoor applications represent an interesting challenge for AR, in contrast to indoor AR applications they face problems strictly related with being outdoors; for instance, accurate tracking indoors is quite challenging, however, accurate tracking outdoors is even more challenging mainly for 2 reasons, **first**, we do not have control over the physical environment and, **second**, AR devices posses *limited* resources such battery life and computational power [22], thus, outdoor AR applications should be aware that memory efficiency and power consumption are key, when working outdoors.

Irrespective of the type of industry –*from gaming to tourism*– the possibilities of outdoor AR applications are endless, for instance, the Apple Design Award winner **Star Walk** [14], is an augmented reality astronomy guide that shows celestial objects in their exact position in the sky, its AR view overlays useful information by harnessing the device's GPS capabilities to present an on-screen view of what stars and constellations should be visible on a clear night from the user's current location. The augmentation respond to the device's tilt i.e if the device is pointing towards the sky, it shows all the stars and constellations in the sky above, instead, if it points towards the ground, it shows what all the stars are in the other hemisphere. Another good example is **CityViewAR** [74], a mobile outdoor AR application that provides augmentation on a city scale, it was developed to provide geographical information about the city of Christchurch, which was hit by several major earthquakes in 2010 and 2011. **CityViewAR** presents information about destroyed buildings and historical sites that were affected by the earthquakes. The augmentation is provided in different formats, for instance, it includes 2D map views, AR visualization of 3D models of buildings on-site and immersive panorama photographs.

In this work we present an interesting type of outdoor AR application, one that identifies points of interest (POI) in nature scenarios. In our context, a POI is a mountain peak, and the augmentation is done by overlaying relevant information on top of the identified peaks. For instance, **PeakFinder** [9], an application that let users to explore mountains and peaks with a 360° panorama display, its AR view shows names, heights and locations of just about any peak in the world. **ShowMeHills** [13], an application that superimposes the names of mountains and hills, each peak can show its name, height, distance and bearing, all happening in real-time. **PeakVisor** [12], a mountain guide that uses AR to display, in real-time, elevation markers atop nearby mountain summits. **Peak.AR** [8], an application that

displays a panoramic augmented reality view of the surrounding peaks with their respective name and elevation. Although these applications, roughly speaking, target our goal, we do not only consider data coming from the location and motion sensors, but we also consider the user's interest i.e what the user is seeing through the camera.

2.1.2 AR for iOS

When interacting with virtual objects as if they are in the real world, a device has to accurately identify the position of the object in relation to itself, thus, effective motion tracking is key to make an AR application *a good AR application*. Standard hardware and software let Apple devices provide a good AR experience, furthermore, the newest iOS has been designed to precisely account for the distance between the motion sensors and cameras as well as the device's place in the world, letting it for a far more interactive experience.

Prior to iOS 11[1], building AR applications with iOS was done either using third-party frameworks, such as Vuforia [16] or Wikitude [17], or building iOS *native* frameworks i.e integrating data coming from the gyroscope, accelerometer, compass or any other sensor, with cameras and microphones, and overlaying 2D/3D graphics. Although time consuming, the latter approach is a viable choice when simple *augmentation* is needed.

Within iOS 11, Apple has included many features and technologies, undoubtedly one of the most interesting is its own AR SDK *ARKit* [4], Apple's proprietary framework specifically designed to build unparalleled augmented reality applications. *ARKit* leans on Apple's powerful hardware to bring the virtual world seamlessly into the real world. *ARKit* makes it easy for developers to create vivid AR experiences and by combining information from the device's motion sensors with data from its cameras, *ARKit* helps to analyze the surroundings more accurately. One of the key benefits of *ARKit* is a feature called *scene understanding*, in charge of finding horizontal and vertical planes within the scene, and tracking and placing objects. However, since tracking is not the only thing needed for AR applications, the rendering of virtual objects is done natively via three main frameworks: SpriteKit, SceneKit, and Metal.

ARKit includes world tracking through *Visual Inertial Odometry* (VIO), plane detection, real world hit testing, and light estimation.

[1] iOS 11 was released on September 19, 2017.

- VIO tracks the environment and places virtual objects with great accuracy and without any calibration. VIO uses several sensors to track where the device is: the camera, accelerometer, and gyroscope.
- Boasts advanced scene analysis capabilities. *ARKit* can estimate the amount of light in each scene and adjust the lighting of virtual objects accordingly.
- The performance can be optimized by popular third-party tools such as Unity and Unreal Engine. These tools allow developers to create compelling virtual objects with advanced graphics.
- Enables revolutionary face tracking capabilities.

Although *new*, *ARKit* has already been used to create interesting augmented reality applications, for instance, *MeasureKit* uses *ARKit* to measure almost anything using just an iPhone or iPad camera, it measures dimensions, angles, it can also verify if something is perfectly level. Another successful example of an application using *ARKit* is *Ikea Place*, an application that helps people to decide what to shop by overlaying Ikea's furniture in any space.

It is worthwhile noting that in despite of *ARKit* many advantages, we did not use it in this work, mainly for 2 reasons, *first*, the augmentation needed for the AR application is not complex as no placement of 3D virtual objects within the scene nor plane detection is needed. Therefore, all the *augmentation* created for the AR view was handled with the native framework we developed throughout this work. *Second*, by the time *ARKit* was released, the development of the AR module was finished.

2.2 Image Understanding

Human brain and eyes' senses are unimaginably advanced. Within fractions of seconds, we are able to identify objects inside our field of view. Not only can we easily name objects we are looking at, but we are perfectly capable of perceiving their depth, discriminate their contours, colors, textures, and infer objects even when they blend into the background. Our eyes take in raw voxels of color data and pass them to our brain, which later translates this information into more significant primitives, such as lines, curves, and shapes, that might indicate, for instance, that we are looking at a car rather than a truck.

It emerged then the desire of mimicking the human visual system, furthermore, letting machines do it. One early breakthrough came in the mid 1950's [100], when Computer Vision pioneers started working on recognition tasks using single images of 2D scenes, such as photomicrographs and high altitude views of the Earth's surface. In the late 1950's scientist tried to break the enigma of Computer Vision by analyzing how brains process visual inputs from the eyes, a study conducted by Hubel and Weisel [60] showcased the importance of edge detection to understand visual mechanics, to achieve this, they experimented on cats; they were awarded the Nobel Prize for this work. In the late 1960's, discussions on the possibility of extracting 3D information from 2D perspective views [20], opened the possibility of endowing robots with intelligent behavior [95].

From a biological science point of view, Computer Vision aims to replicate the functionality of components responsible for the human sense of sight. Recognizing animals, describing a view, differentiating among visible objects are things that us humans do without thought or hesitation, furthermore, we are able to recognize objects under all kinds of variations in illumination and viewpoint, our brain and eyes' senses are simply too good at this task which allow us, for instance, to recognize a friend in a photograph taken many years ago a really cake-walk. Conversely, it took many years of research to grant the ability of detecting objects to a computer with reasonable accuracy, by all means, Computer Vision systems still suffer by comparison. Programming machines to replicate human vision has huge implications, all lie on the inherit perception of an object, us humans possess, where we automatically see lines, contours, and objects, computers just see large matrices of numbers.

Nonetheless, since the early attempts and efforts done back in the 1950's, there have been remarkable improvements in device's capabilities, such as, computational power, memory capacity, power consumption and image sensor resolution, all these have improved the performance and cost-effectiveness of Computer Vision applications. These advancements are accompanied by the development of sophisticated algorithms for tasks such as face recognition, image classification, object detection, etc.

For instance, Machine Learning is driving a revolution in vision-based applications. More recently, the breakthrough of Deep Learning has enormously influenced improvements in Machine Learning techniques, in particular Computer Vision, subsequently, new scenarios arose for Computer Vision applications and its usage on trending fields such as Augmented Re-

ality. Compared to traditional Computer Vision techniques, Deep Learning provides greater accuracy in tasks such as image classification. Given that neural networks used in Deep Learning are trained and not programmed, applications relying on it take better advantage of the enormous amount of imaging and video data available in the websphere. By leveraging on the sophistication and versatility of Deep Learning, frameworks, possibly, can be utilized in any kind of domain, compared to Computer Vision algorithms that tend to be more purpose-specific. Nowadays, Deep Learning-powered image recognition is performing better than human vision on many tasks, making applications of Deep Learning in vision such as self-driven cars possible in near future.

Before diving into Deep Learning models, let us take a quick review of the challenges associated with Computer Vision and the traditional approaches used before Deep Learning became popular.

2.2.1 Computer Vision Overview

The most basic application *-yet the building block for others-* of Computer Vision is *Object detection.* On a daily basis, us humans unconsciously detect objects by the mere action of opening our eyes, it is intuitive as it is innate, it is very simple for us and we do it with such utter ease. However, several challenges arise when trying to design systems similar to the human vision sense, issues directly associated with the recognition of objects, we explain them as follows:

- **Viewpoint variations** The same object can have different positions inside an image, furthermore, it can be seen from different angles depending on the relative position between the object and the observer. Figure 2.1a shows an example of this condition. Although it is obvious that these pictures contain the same object, to impart this knowledge to a computer, it is not a trivial task.

- **Difference in Illumination** The same object can be portrayed under different light conditions as shown in Figure 2.1b, however, we will still be able to recognize the same object, making a computer capable of understand this phenomenon the way we do, is a difficult challenge.

- **Occluded parts** Occlusions depict a major problem, humans can intuitively infer an occluded object, we fill out the missing parts by using our previously acquired knowledge, we tend to see complete images even when they are clearly not. However, when small or large

portions of images are hidden as illustrated in Figure 2.1c, computers tend to misinterpret what they are seeing.

- **Background Clutter** Figure 2.1d shows a good example of clutter, that basically is images blending into the background. We humans, may fail to recognize the man in the picture in a first attempt, but when observing carefully, we see there is a man in the image. As simple as it may seem, it is an uphill task for a computer to learn.

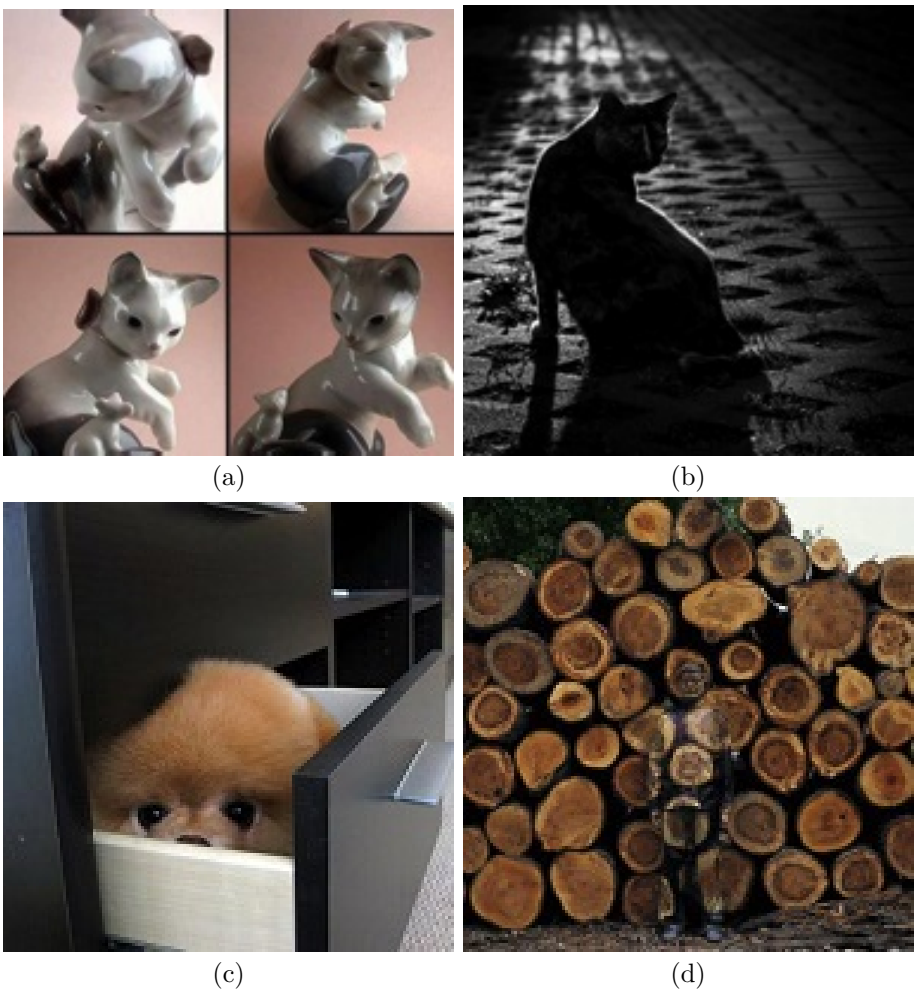


Figure 2.1: Challenging Images: (a) Cat observed from different viewpoints. (b) Ambient light changes. (c) A dog's head partially occluded. (d) Man blending into the background.

Nonetheless, by means of Computer Vision existent techniques, we are able to solve most of these challenges individually, unfortunately, we are still

decades away from a system which can get close to the human eye, in this sense, Computer Vision systems will continue suffering by comparison.

2.2.2 Traditional Approaches

Several techniques exist, other than Deep Learning, that could possibly enhanced Computer Vision achievements. Albeit useful when dealing with simple problems, when data and task complexity increase, these approaches are no longer suitable alternatives to Deep Learning. In the following, we discuss three *simple approaches* to image classification, please note that other -*more*- sophisticated techniques can be used but they would rarely outperform a Deep Learning model.

Linear Classifiers

A linear classifier uses the characteristics of a given object to identify to which class it belongs to, it is called *linear* since its decision is based solely on a linear combination of the object's characteristics. To classify an image, it uses a parametric approach in which each pixel value is considered a parameter. Broadly speaking, we could interpret it as a weighted sum of the pixel values with the dimension of the weights matrix depending on the number of outcomes. To this end, the weighted sum of pixels forms a sort of *template image* that is later contrasted again all images awaiting classification. It becomes clear then, that this approach is not generalizable, not only it will face difficulty in overcoming the challenges discussed in Section 2.2.1 but it will pose an extra difficulty when designing one single *template* for all the different cases.

K-Nearest Neighbors

K-Nearest Neighbors or simply KNN [40] [43] is a supervised learning technique used to classify objects based on closest training samples in the feature space. During the image classification process, each image is matched with all remaining images in the training set. The top k with minimum distances are selected. The majority class of those top k is predicted as output class of the image. Several distance metrics are used to select the top k, for instance $L1$ distance (sum of absolute distance) and $L2$ distance (sum of squares).

Although KNN performs well with multi-modal classes because the basis of its decision is based on a small neighborhood of similar objects, a major disadvantage is that it uses all the features equally in computing for similarities. This can lead to classification errors, especially when there is only a

small subset of features that are useful for classification [64], meaning KNN may fail when classifying an image containing the same object under the same light conditions and orientation, but, for instance, in different regions of the image as seen in Figure 2.2. Despite being the same object, KNN is likely to give highly non-zero distance for these 2 images.

Figure 2.2: KNN Classical Errors. A classical error for KNN is to give non-zero distance for the same object but in different positions.

Support Vector Machines

Despite being mainly used in other real world problems like voice and tone recognition and text categories, SVMs [39] are also used in image classification and object detection. In the late 2000s gained popularity in Machine Learning as it showed practical performance [79].

SVMs produce a model, based on the training data, which will be able to predict class labels of the test data accurately. It works by using optimal hyperplanes produced via maximum margin between two different classes in a high dimensional feature space [69]. When used as an image classifier, it applies this classification process to all the features extracted, then the test points are subsequently mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

A main advantage of SVMs is that perform well on high-dimensional datasets even when available data to train is scarce. However, feature-based classifiers such SVMs, although usually outperform KNN classifiers for image classifications tasks [64], do not substantially improve their performances after reaching a plateau, and usually no gain is shown when introducing more data, whereas other techniques such neural networks are able to fully take advantage from additional data and constantly outperform feature-based classifiers. Furthermore, not only it will face the issues described in

Section 2.2.1 but additional limitations arise regarding size and speed during training and testing, and the selection of the kernel function parameters.

With this overview, we introduced some intuition into the challenges faced by approaches other than Deep Learning. We have further examined how human and computer vision extract features from raw pixels, and suggested how Deep Learning could be of good use to tackle the problem of learning more complex features out of raw pixel values.

2.2.3 Deep Learning Models

Before exploring into more specialized concepts and models such as Convolutional Neural Networks, we will briefly discuss the basics of artificial neural networks.

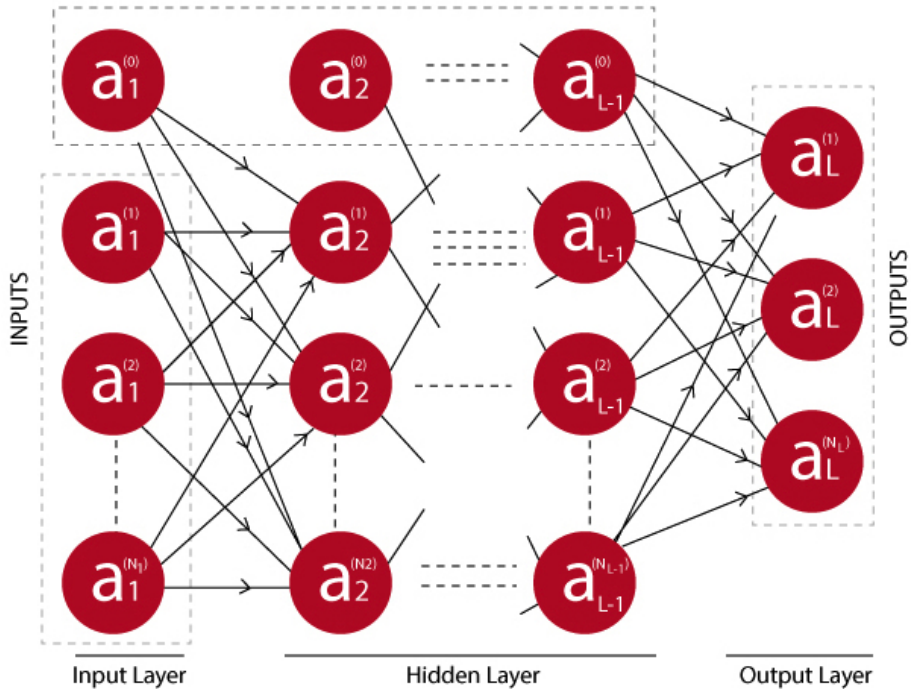
2.2.3.1 Feedforward Neural Networks

A Neural Network is a computational model that works in a similar way to the neurons in the human brain, in the sense that each neuron takes an input, performs seldom operations then passes the output to the following neuron, and so on and so forth. In 1957, an early breakthrough appeared in the form of the *Perceptron* machine, a very early *artificial neural network* capable of sorting images into very simple categories like triangles and squares, invention of psychologist and Computer Vision pioneer Frank Rosenblatt [85]. Broadly speaking, learning emerges from the firing of neuron cells in the brain, thus, it happens when links among neurons get significantly stronger, and to get stronger connections neurons need to connect more often. Rosenblatt's insight was that the same process could be applied in computers.

Despite being a very promising idea, it was demonstrated that the *Perceptron* would fail to recognize different classes of patterns. However, later it was brought to the attention of researchers that more layers of perceptron could be stacked along each other in the form of a feedforward neural network, furthermore, insights arose suggesting that not only training them using the back-propagation algorithm [86] would be possible but likely to yield significant results.

We call these models *feedforward* because their topology is defined so that information flows from the *input* a_1, through the intermediate computations used to define f, all the way to the *input* a_L as can be seen in Figure 2.3. We can observe that there are no feedback connections -*links*- in which functions being evaluated from *input* a_1 are fed back into the model. As we will learn

later back-propagation posed a major advancement in how networks are, nowadays, modeled.



Source: Analytics vidhya. Fundamentals of Deep Learning.

Figure 2.3: Feedforward Fully-Connected Neural Network. L layers: 1 input layer, 1 output layer and L-2 hidden layers. N_i is the number of neuron in i-th layer excluding the bias unit, and $a_i^{(j)}$ is the output of the j-th neuron in ith layer.

Neural Network models are arranged into several layers composed by many units performing in parallel where each of them resembles a human brain neuron as they also receive input from many other units and compute its own activation value. A neural network transforms the information carried through a series of hidden layers -*those that are neither input or output layers*- turning it into a new representation that better addresses the, for instance, classification task. Samples in the training set denoted as (x_i, y_i) pairs, are very important since they establish the behavior of the output layer associated to each input sample but only that, in any case the behavior of the remaining layers are not directly specified by the training data. Thus, the learning algorithm must decide how to use the intermediate layers to produce the desired output. Given the fact that the training data does not have a saying on the desired output for each of the intermediate layers, these are called *hidden layers*. The term *Deep Learning* comes from the

analogy this process posed, piling several layers all together to later train them jointly. There is a rather remarkable intuition when we talked about Deep Learning, and that is that is based on the evidence that a deep, hierarchical model can be exponentially more efficient than a shallow one [27] as recent research supporting this intuition shows [70] [42] [81].

Designing a feedforward network involves taking decisions about the hyper-parameters we use to design a network, this however, is no different for any other Machine Learning technique. Among the most important parameters to be considered when designing a feedforward network are the optimizer, the cost function, the kind of hidden and output units, the activation functions of each hidden layer and the depth and topology of the network. We proceed to briefly explain some major concepts behind these hyper-parameters.

Gradient Based Learning

Training neural networks is typically done by means of iterative, gradient-based optimization methods that try to drive the cost function to very low values. This being said, the cost function these methods intend to lower is very sensitive to nonlinearities and high dimensionality, which causes for it to be highly non convex, making the optimization very hard. With this understanding, various mutations of the classic gradient descent algorithm have emerged, basically, they are improvements of the Stochastic Gradient Descent (SGD) algorithm [83] [63] [32], which is an approximation of the classic gradient descent algorithm that uses an estimate of the gradient of the loss function based only on a single example of the training set.

New optimization methods trying to ameliorate the performance of the SGD algorithm variants using adaptive learning rates and second-order curvature informations such as Quasi-Newton methods have been proposed [65] [102]. Luckily, deep networks optimization is a very active area of research, which poses quite a promising horizon.

Weight Initialization

In order to properly let the optimization algorithm to rapidly converge to a good solution, selecting a robust initialization method for the parameters of the network becomes of greater importance. Some heuristic approaches have been proposed, let us discuss some of them:

- **All Zeros** Although biases can generally be initialized to zero, weights

need to be initialized carefully to break the symmetry between hidden units of the same layer, otherwise all the neuron will generate the same output initially and similar gradients would flow back in back-propagation [26].

- **Gaussian Random Variables** The weights can be initialized with a zero-mean Gaussian with a small standard deviation around 0.1 or 0.01. This works for shallow networks, i.e. around 5 hidden layers but not for deeper networks where small weights make outputs also small and while moving towards the end, the values become even smaller. Thus, the gradients will also become small resulting in gradient harm at the end. [59]

- **Fan-in, Fan-out** Techniques based on the activation of the hidden units such as the ones proposed by [52] and [58] have been well received. [71] and [52] recommend scaling by the inverse of the square root of the fan-in. For hyperbolic tangent units an-in is the number of inputs of the unit, in such like manner, for sigmoid units, fan-in is also the number of inputs of the unit.

Back-Propagation Algorithm

Introduced by Rumelhart in the 1980's [86], was conceptualized to optimize multi layered neural networks by determining the loss (or error) at the output and then propagating it back into the network. Interestingly, Fully-connected Feedforward Neural Network became very popular after the introduction of this algorithm. Broadly speaking, back-propagation provides an efficient and exact way of computing the gradient of the cost function in order to train a neural network.

Training neural networks is done in a feedforward fashion, where information flows forward from the input layer passing through the hidden units and producing the output prediction. This process goes under the name of *forward propagation*. After the forward is complete, the cost function is evaluated and the parameters of the network (weights) are updated to minimize the error resulting from each neuron. The first step in minimizing the error is to determine the gradient of each node *wrt* the final output, here, the algorithm is exploited by allowing the information given by the cost function to flow backwards through the network in order to compute the gradients. Given that it is a gradient-based learning method, it minimizes the error function by looking at the direction of the gradient. Once the gradient of the cost function is computed, the gradients are used to update

the parameters, hence, weights are learned propagating back, through all the layers of the network, the prediction error.

Cost Functions

Another important aspect when designing a deep neural network is deciding the cost function to use $J(\Theta)$. When using deep neural networks as classifiers, the model defines a distribution $p(y|x;\Theta)$ over a set of classes, hence, the maximum likelihood can be used to estimate the parameters required. This means using the cross-entropy cost function to measure the error between the training set and the predictions thrown by the model. What cost function to use depends on various factors, such as the domain, the complexity, the data distribution, and the output; for instance, if the outputs are real numbers we are likely to be estimating the parameters for a regression task, thus, a usual choice would be the classic *Mean Squared Error*.

Activation Functions

Hidden units as well as deep network optimization are a very active area of research. In recent years various nonlinear activation functions have been proposed aiming to improve the performance and trying to simplify the optimization. Let us discuss some of the most popular options:

- **Sigmoid Function** Sigmoidal units 2.1 saturate to a high value when their input is very positive and saturate to a low value when the input is very negative. As we know, gradients get multiplied during back-propagation causing this small gradient stop back-propagation into further layers, thus, harming the gradient. Such trait makes sigmoidal units inoperative, as they function only in a very narrow range around 0, hence, gradient-based learning very hard. For this reason they have been substituted by other kind of hidden units in feedforward networks, such as linear units.

$$\begin{aligned} g(z) &= \sigma(z) \\ \sigma(z) &= 1/(1+e^x) \end{aligned} \tag{2.1}$$

  As can be seen in Figure 2.4 all the outputs are between 0 and 1, meaning sigmoidal units are not zero-centered. As these become inputs to the next layer, all following gradients -*belonging to next layer*- will

be either positive or negative, causing the path to optimum to be a zig-zag.

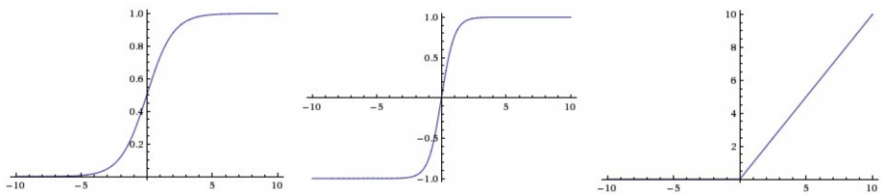
- **Tanh Activation** It is simply a hyperbolic tangent function 2.2 as shown in Figure 2.4. It is always preferred over sigmoid when outputs fall in the range [-1,1]. However, it will still harm the gradient, thus, nowadays they are mostly used as output units and other settings, for instance, auto-encoders or Recurrent Neural Networks.

$$g(z) = tanh(z) \tag{2.2}$$

- **ReLU (Rectified Linear Unit)** First proposed for restricted Boltzmann machines [78] it finds its way into neural networks [41] [53] proving to be very successful.

$$g(z) = max(0, z) \tag{2.3}$$

It is the most commonly used activation function for Convolution Neural Networks since typically avoid the *vanishing gradient problem* [28] given that their derivative is 1 for positive values and 0 elsewhere, thus, the gradient would not saturate in the positive region. It is also computationally very efficient as simple thresholding is required. Empirically found to converge faster than *sigmoid* or *tanh*. However, its outputs are not zero-centered and always positive, furthermore, gradients at $z < 0$ and $z = 0$ are tricky. If the caveats are kept in mind, ReLU can be used very efficiently.



(a) Sigmoid unit (b) Tanh unit (c) ReLU unit

Figure 2.4: Activation Functions. Sigmoid non-linearity (a) squashes real numbers to range between [0, 1]. Tanh non-linearity (b) squashes real numbers to range between [-1, 1]. Rectified Linear Unit (ReLU) activation function is zero when $x < 0$ and then linear with slope 1 when $x > 0$

2.2.3.2 Convolutional Neural Networks

One of the most popular techniques used in improving the accuracy of image classification is *Convolutional Neural Networks* (CNNs or ConvNets for short). Its breakthrough dates back to 1989, when a group of researchers demonstrated they were very effective at classifying hand-written images [72].

Along the years, deep convolutional networks have outperformed state of the art classifiers in many visual recognition tasks. However, it is the *recent* availability of both large datasets and massive amounts of computational power that have made them the crux of Deep Learning applications in Computer Vision. Popularized by the outstanding results obtained by [67], where a network with 8 layers trained with one million samples proved to be specially successful in the Computer Vision field while breaking records in the *ImageNet Large Scale Visual Recognition Competition* [87] back in 2012. Consequently, ConvNets have become the *default* choice for almost every Image Classification problem. Since then, even larger and deeper networks have been trained, such is the case of GoogleNet [94], ResNet [57], among others.

ConvNets are a special type of neural network that work in the same way of a regular feedforward networks except that they include convolution layers at the beginning of the network topology. Deep ConvNets work by consecutively modeling small pieces of information and combining them deeper into the network. Instead of feeding the entire image as an array of numbers, the image is broken up into a number of tiles, the machine then tries to predict what each tile is. One way to understand this concept is by picturing, for instance, that the first layer is trying to detect edges and form templates for edge detection. Then, all subsequent layers will try to combine them into simpler shapes and eventually into templates of different object positions, illumination, scales, etc. At the end, the final layers will match an input image with all the templates and the computer will predict what is in the picture based on the prediction of all the tiles, as a sort of weighted sum of all of them. Such trait enables the computer to parallelize the operations carried out and detect the object regardless of where it is located inside the image.

Fully ConvNets use a special architecture that exploits the spatial structure of the images. More in general these kind of neural networks are specialized in processing data that has a grid-like topology such as time-series (1D grid) or images (2D grid) [54]. In particular, a ConvNet is a neural

network that instead of using a matrix multiplication uses convolution operations to compute the activation of a layer. It is called **fully convolutional** since the neural network is only composed of convolutional layers without any fully-connected layers or MLP usually found at the end of the network. A fully convolutional net tries to learn representations and make decisions based on local spatial input [76].

Another kind of neural network exist called Fully-Connected Neural Network that has shown to perform well in many different fields, however, they are not really used in practice for image recognition tasks since given its full connectivity, the number of parameters grow exponentially and is likely to lead to overfitting. Furthermore, in contrast with Fully Convolutional Neural Networks, they do not take into account the inherent spatial structure of an image, thus, input pixels far apart and close together are treated instinctively, which means that appending a fully connected layer enables the network to learn something using global information where the spatial arrangement of the input falls away and need not apply. The main difference however, is that the fully convolutional net is learning filters everywhere, even in the decision-making layers at the end of the network are filters.

In order to function, ConvNets use 2 major constructs as primitives:

- **Filters**, also known as feature detectors, and
- **Feature Maps**, also known as pre-activations or convolved features.

A *filter* is represented by a small matrix that conveys a feature that we are interested to find in the original image. In Figure 2.5, the filter on the top attempts to discover the parts of the original image with vertical lines, while the filter on the bottom tries to discover parts of the image with horizontal lines.

The actual detection process works by taking the convolution of the filter with the original image. Figure 2.5 shows us the results of performing the convolution on the right side. The outputs of the convolutions, which locate the positions of the features in the original image, are the *feature maps*.

Refer to Figure 2.3 as we will use it as a scheme to describe how these primitives are turned into concrete structures. Layers of neurons in a feedforward neural network represent either the original image or a feature map. Filters represent combinations of connections or links that get replicated across the entirety of the input. Finally, the output layer at each stage, is the feature map generated by these filters. A neuron in the feature map

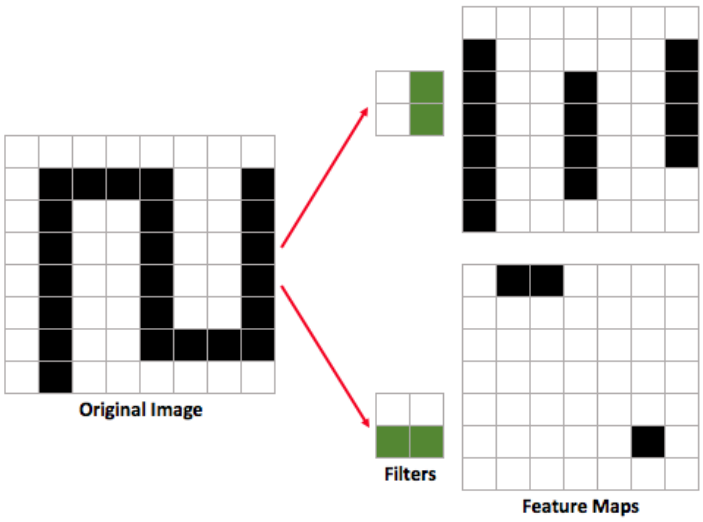


Figure 2.5: Primitives: Filter and Feature Maps. Filters drawn in green. The filter on the top attempts to discover the parts of the original image with vertical lines, while the filter on the bottom tries to discover parts of the image with horizontal lines.

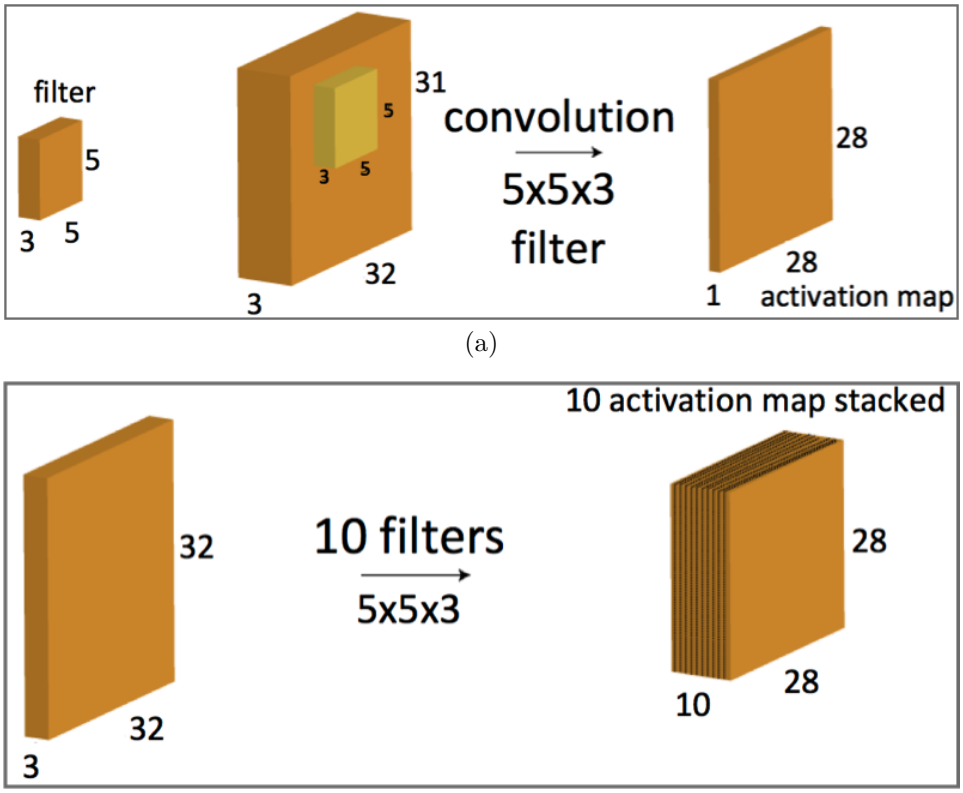
is activated if the filter contributing to its activity detected an appropriate feature at the corresponding position in the previous layer.

A typical ConvNet architecture consists in a pipeline of three operations: (a) a *convolution* represented by means of a convolutional layer, (b) a nonlinear transformation, represented using activation functions and (c) a pooling stage. First and foremost, the ConvNet computes several parallel convolutions to produce a set of feature maps. Later, the nonlinear activation function is applied to the generated feature maps. Finally, a pooling layer may be inserted in-between successive convolutional layers in a deep ConvNet architecture. Let us discuss these operation-facilitators in detail.

Convolution Layer

The primary purpose of convolution in case of a ConvNet is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small structures of input data. The convolution creates a map of where certain features appear in the input. Moving the object in the input, its representation will move the same amount in the output. This is important for tasks such as image recognition where the same local feature is useful everywhere in the input. For instance,

when object's edges appear along the image, as one might expect in any sort of image, it is very useful to have the network learned a robust edge detector. On the contrary, in face recognition tasks for instance, we might want to extract features at different locations without paying attention to soft edges, thus, the network subset in charge of processing the top of the face, it also needs to be very cautious to the eyes, in contrast, the network subset processing the bottom of the face needs to look after the mouth.



Source: Analytics vidhya. Deep Learning for Computer Vision.

Figure 2.6: Convolution Layer. (a) This filter is a set of: $5x5x3 = 75 + 1bias = 76weights$. At each position, the weighted sum of the pixels is calculated and a new value is obtained. A single filter will result in a volume of size 28x28x1 (b) Multiple filters are run at each step. In the image 10 filters are used.

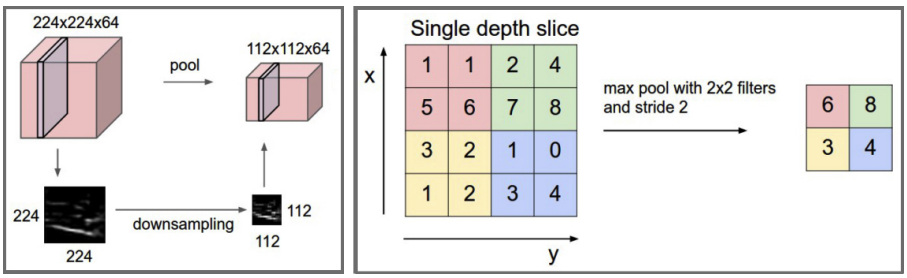
The filter slides by 1 or more pixels -called *stride*- over the input image to produce a feature map, for every position, an element wise multiplication is computed and later added to get the final integer which forms a single element of the output matrix. If we were to choose a different filter, the

convolution over the same image would give a different feature map as can be seen in Figure 2.6. It is worth noticing that this convolution operation captures the local dependencies in the original image.

Pooling

Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc. The most widely used type of pooling is called Max Pooling, it works under a simple approach, it defines a spatial neighborhood or window and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average, called Average Pooling. In practice, Max Pooling has proved to work better.

Pooling layers work by sampling in each layer using filters. Consider the layer presented in Figure 2.7. If we use a 2x2 filter with stride 2 and max-pooling, we get the response illustrated in the Figure 2.7.



(a) (b)

Figure 2.7: Pooling Layer. Input volume of size 224x224x64 is pooled with filter size 2, stride 2 into output volume of size 112x112x64. The volume depth is preserved (b) A 2x max pooling operation where each max is taken over 4 numbers on a patch of dimension 2x2.

The important fact behind the use of pooling is that it helps to make the representation invariant to small transformations, distortions and translations in the input image, meaning a small distortion in input will not change the output of pooling. Furthermore, it helps us arrive at an almost scale invariant representation of the input image. This particularly is very powerful as it allows us to detect objects inside an image regardless where they are located.

Putting all these concepts together, we are ready to tackle any interesting Computer Vision problem.

2.3 Mountain Image Analysis

Mountainous sceneries analysis is quite ambitious, since they have little structural information, furthermore, season, weather and geographical location have a deep impact in the appearance of mountains, making the standard approaches not adequate for this purpose. Nonetheless, this problem has been addressed from different viewpoints, some using traditional Machine Learning algorithms with Dynamic Programming, others, Computer Vision with Deep Learning techniques.

Mountain peak detection has been a research topic of some interesting papers, for instance, in [75] Support Vector Machines are used to predict possible skyline segments, additionally, linking incomplete fragments of skyline is treated as a shortest path problem, hence, Dynamic Programing is used to solve it, furthermore, to identify peaks a 2D curve matching is carried out on the extracted skylines. In [48], the authors perform an edge-based matching between the visual content of each photo and a terrain view synthesized from the DEM that uses the location coordinates included in the geo-tags of each photo. Both approaches show significant results, however, the estimation of the position of the peaks in [48] is more accurate than the one proposed by [75], since contextual information is included in the geo-tags. A system for the annotation and augmentation of mountain photographs is presented in [24], the proposed technique is able to automatically derive the pose of the camera relative to the geometric terrain model by using an edge detection algorithm, the technique searches for the best match with silhouette edges rendered using the synthetic model, although interesting, this approach focuses on applications for annotation of the mountains and not in building environmental models. A similar approach is presented in [84], the authors introduce an approach to identify mountain peaks and their corresponding edges based on the estimation of the field of view and the direction of the camera using also a matching algorithm on the edge map and the mountain silhouettes, in contrast with previous works, they include an algorithm to extract the visible part of the identified mountains.

PeakLens [10] has already addressed the problem of mountain peak detection in mobile applications, by using the framework proposed in [47] to detect mountain peaks considering as an input not only the position and orientation of the user's device but also the content of the current view and the DEM, they estimate the virtual panorama visible from the user's viewpoint. The framework proposed in [47] has been improved for mobile AR applications in which the captured images can be taken in adverse weather

conditions, more importantly, as the presence of transient occlusions in the skyline is even more frequent in these conditions, an approach based on *only* edge filtering is not longer the best choice, since objects occluding the skyline can produce misleading edges, consequently, erroneous classification that may harm the alignment with the DEM and the positioning of peaks may be thrown, for that matter, the Convolutional Neural Network for pixel-wise skyline detection presented in [50] has been included as a filtering procedure that retains only edge pixels of the skyline so as to improve the alignment *wrt* the terrain. The ConvNet presented in [50] is used as a binary classifier at pixel level, a set of scores –*or probabilities*– for each pixel is obtained; a threshold that maximizes accuracy is chosen and a further post-processing step is done, one that only retrieves a single pixel per column, as shown in Figure 2.8.

(a) Photo (b) Photo

Figure 2.8: PeakLens ConvNet. **Left:** Test image. **Right:** The ground truth is represented in white. Correctly predicted skyline is represented in green. Miss-classified skyline is represented in red [50].

The approach presented in [50] may contribute to the detection of *occlusions*, after the alignment of the landscape skyline and the skyline pixels classified by the ConvNet, the skyline-interrupted fragments could be considered as occlusions, thus, *occlusion* detection can be achieved to some degree of accuracy. In contrast, the approaches presented in [24] and [23] rely on edge-based heuristics using Dynamic Programming to connect edges, such approach has proved to work poorly on images taken in bad conditions, in which a cloud, a high voltage cable, or a building could negatively impact on the heuristic edge filter, what is more, some of these objects' edges could possibly be recognized as skyline leading to occlusion segments being misinterpreted. As it will be further discussed in this work, these approaches leave a *big* room for improvement when the problem is to detect *occlusion* segments.

2.4 The Occlusion Problem

Many approaches used in image classification and object detection simply ignore *occlusions* and treat them as *outliers*, which may be reasonable when the main purpose of the modeling is to *simply* recognize an object. However, by ignoring *occlusions* we are *blurring* features that given its singular appearance are just hard to detect, for instance, in PeakLens [10] the extracted skyline is used to perform an alignment *wrt* the terrain, therefore, ignoring occlusions would be detrimental.

Another common approach to model occlusions is to treat them in a post-process step, thus the *occlusion-detection* stage is no longer part of the *learning* process, this is certainly true for some techniques used in Computer Vision, where *visibility* is treated as a binary variable that could be inferred later in the process. Other examples include those in which the **stereo matching problem** [15] -*also known as correspondence problem*- is prevalent, making the formal prediction then, through dynamic programming, as in [55].

Although the former techniques are quite successful, there are others that do include the *occlusion-detection* stage as part of the *learning* process, to do so, for starters, *all* the *occlusion*-hypotheses could be captured, nonetheless capture all the *occlusion*-hypotheses for mountain peak detection is infeasible due to its sole nature i.e. in this type of scenarios there are an infinite number of occluders and their positioning is entirely arbitrary. Whereas some part-detection models have successfully captured and scored all of these hypotheses as in [49] and [99], this is not realistic for scenarios that are far less constrained.

Models though, can explain features generated by occlusions as in the case of [51], when pose estimation is coupled with segmentation, however, such models require several orders of magnitude for parameters as for training. Conversely, in this work, we propose a model that does not required such orders of magnitude for parameters nor for training, but instead relies on the shape of the input data we are feeding to our network, and as it will be demonstrated, the approach presented in this work is able to provide more meaningful context to our network, thus, making it able to recognize occlusions in the horizon.

Chapter 3

Non-Intelligent System

In this chapter we introduce what we call the *non-intelligent* system, an outdoor mountain peak detection application for iOS devices. We describe in detail the architecture of the system and further design choices, the system modules and submodules, and all main functionalities to finally present *the* working prototype named *PeakLens-iOS*.

3.1 Background

PeakLens [10] is an AR Android application for mountain peak detection in real-time. Within PeakLens, *peak detection* can be achieved either using GPS and compass signals, or exploiting a Computer Vision module specifically tailored to identify mountain peaks. The former, uses raw data from the Sensor API provided by Android, the latter, fixes data errors coming from the sensors –*misreadings introduced by the sensor's hardware*– by leveraging a ConvNet able to classify skyline pixels.

In this chapter we present the iOS version of such application that takes fully advantage of Apple's standardized hardware and software with the aim to provide not only a meaningful user experience but hit more accurate readings by leveraging on the device's on-board sensors, so as to explore the specific uncertainties that arise in outdoor mobile augmented reality applications. However, since data coming from the device's physical hardware are entirely dependent on the operating system from which are obtained, developing the iOS application is **far from trivial**, simple *porting* code cannot be done, instead, a complete redesign of the application architecture is needed.

Figure 3.1 shows a *scene capture*, using the PeakLens Android application, in which the augmentation was done using only sensor's data. In this example we can observe that, although the application is able to detect mountains peaks (*Pizzo del Tre Termini* and *Monte Colmegnone*) and place them on the AR view, there are some obvious misplacements, for instance, the alignment of the peak *Monte Colmegnone* is not 100% correct, although the displacement of the virtual tag with respect to the actual skyline is not completely detrimental to the AR experience, is evident that inaccurate sensor readings do affect at some degree the final outcome and can reflect on the user's perception of the correctness of the application. A very noticeable error in this example is the alignment of the peak *Pizzo del Tre Termini*, although the peak is present in the scene, is not visible from the user's viewpoint as there is a house occluding it, this specific misalignment also affects the outcome results and is detrimental to the overall AR experience. As we will learn through this chapter, this particular scenario cannot be overcome using only sensor data but additional task needs to be carried out in order to handle it.

Figure 3.1: PeakLens Capture. A scene capture made with PeakLens Android application.

3.2 Overview

As a first attempt to detect mountain peaks inside a scene and provide users with meaningful augmented content of the peaks seen through their device's camera, we have developed an iOS application that takes fully advantage of

Apple's standardized hardware and software with the aim to improve not only the user experience but hit more accurate readings by leveraging on the device's on-board sensors. Additionally, the massive number of users[1], and consequently, user's feedback, comes just as a *bonus*.

However, as we will show throughout the course of this study, empowering an AR mobile application for mountain peak detection with only sensor-based data is not sufficient, although useful to explore the uncertainties that arise in outdoor mobile AR applications, can fairly provide a meaningful user experience as we are undoubtedly neglecting the *actual* user's interest -*what the user is seeing though the camera*, for this reason, we consider this first approach as *Non-Intelligent* as no learning step is included to correctly understand the interest of the user. Nonetheless, we validate the usefulness of this approach as a way to examine uncertainties related with the physical world we are trying to augment, foremost, based on the findings of this approach we set the requirements for the improved Computer Vision technique discussed in Chapter 4.

3.3 PeakLens-iOS

PeakLens-iOS is and outdoor AR mobile application designed for mountain peak detection, developed for Apple devices. To ensure proper functioning, devices running *PeakLens-iOS* must be shipped with a minimum set of required hardware, for instance, a camera, an accelerometer, a gyroscope and a compass must be present.

As for the technical specifications, to build *PeakLens-iOS* we used *Swift*, a general-purpose, multi-paradigm programming language created by Apple, currently considered the *default* choice when building iOS and Mac applications. Given the nature of *PeakLens-iOS*, some specific built-in sensors are considered mandatory, to fulfill this constraint, *PeakLens-iOS* runs strictly on iOS 7[2] -*or later*-, iPhone 4[3] -*or later*-, iPad 2[4] -*or later*- and any version of iPad Mini and iPad Air. As can be seen in Table 3.1, devices running prior operating systems do not complied with *PeakLens-iOS* hardware constraints, thus, compatibility is not guaranteed.

Figure 3.2 shows a high-level architecture diagram of the *PeakLens-iOS* and the modules it interacts with. *PeakLens-iOS* is built upon 3 main com-

[1]By the end 2016 there were over 700 million iPhones in use worldwide [6]
[2]iOS7 was released on September 18, 2013.
[3]iPhone 4 was released on June 24, 2010.
[4]iPad 2 was released on March 11, 2011.

Device	Model	Sensors			Compatibility
		Accel.	Gyro.	Compass	(>= iOS 7)
iPhone	<= 3G	✓	✗	✗	✗
	3G	✓	✗	✓	✗
	>= 4	✓	✓	✓	✓
iPad	1	✓	✗	✗	✗
	>= 2	✓	✓	✓	✓
	Mini	✓	✓	✓	✓
	Air	✓	✓	✓	✓

Table 3.1: Technical Specifications. The information shown above was collected from Apple's products technical specification [3].

ponents, each responsible of well-defined functionalities, named as follows: the *augmented reality framework*, the *mountain peak detection module* and the *standardization module*.

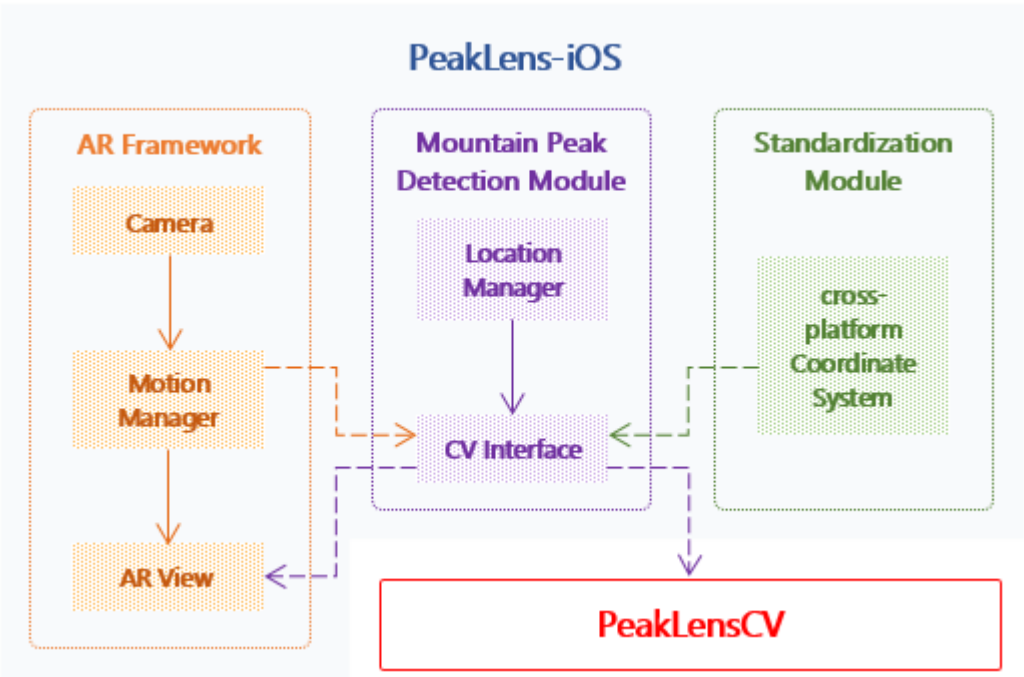


Figure 3.2: PeakLens Architecture. A high-level architecture diagram of the application presented in this chapter.

- The **Augmented Reality Framework**, that is in charge of overlaying augmented content of detected peaks, responsive to the device

positioning and user's movements.

- The **Mountain Peak Detection** module, that is in charge of identifying mountain peaks present in the live camera images, using the current user's location.

- The **Standardization** module, that is in charge of generating *standardized data* that can be used in the scene replay of Android devices running PeakLens [31], as well as to be used in the application's Computer Vision module.

3.4 Augmented Reality Framework

When an application adds additional content, such as 2D or 3D elements, to the live camera image, users experience *augmented reality*, which is nothing but the *illusion* that those elements are part of -*or inhabit*- the real world. In that way, AR applications give us an *enhanced* version of reality by bringing elements of the virtual world into our real world. As discussed in the previous chapter, outdoor applications present a perfect scenario to include *augmentation*, since they provide the user an experience that lies in that *spectrum*, just in between what is real and what is virtual i.e. it does not diminish the activity been held but enhances the things the user sees, hears or feels.

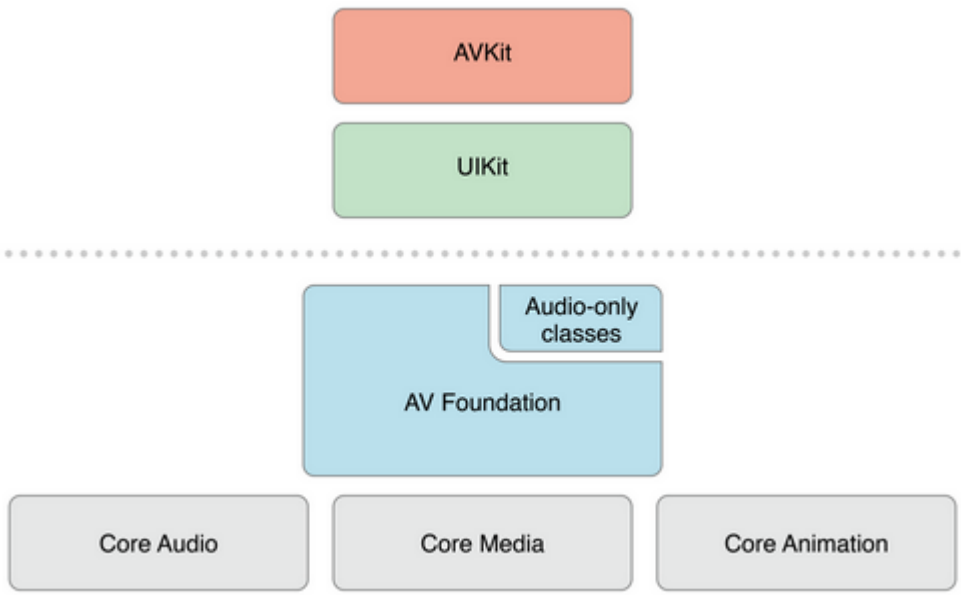
In *PeakLens-iOS*, the *augmentation* is made by superimposing information about the mountains seen through the camera, to provide such *augmentation* on iOS devices, we must create a framework that allows us to use all resources needed from the device on-board hardware, as well as from the operating system itself. In order to set-up an *augmented reality* environment in our application, we combined the camera scene capture and the device motion tracking service, so as to accurately track the world around us and blend all virtual content smoothly into the user's environment. Our main goal then, is to bring the virtual world seamlessly into the real world in such a way that a realistic experience can be provided to the user.

3.4.1 Camera Scene Capture

In order to provide an *augmented reality* environment, we first need to stream a live view of what the user is seeing at the moment, to then add virtual elements on top of this view, to do so, we must be in full control of the device's camera. iOS provides access and control of the media physical

devices such as cameras and microphones, by means of the `AVFoundation` framework [2].

`AVFoundation` captures, processes and controls all the audiovisual media on Apple platforms, specifically `AVFoundation` includes a *Camera and Media Capture* subsystem that is needed for applications that require a full control of the physical hardware. Figure 3.3 shows the architecture of `AVFoundation` for iOS, a similar architecture is provided for OS X.



Source: Apple's Developer Documentation. About AVFoundation.

Figure 3.3: AVFoundation Stack on iOS. It lets access to input streams from devices and manipulate video during real-time capture and playback.

To handle photo and video capturing within our application, we have created a controller named `BasicCameraViewController`. Likewise, to manage the capture from the camera we need to gain access to the physical device, create an input, setup the session using this input and then save it on an output. Figure 3.4 shows a diagram depicting this process.

While recording input from the camera, a session is required, this session is in charge of coordinating data flow from input devices such the camera, to desired outputs, such movie files and photos. *PeakLens-iOS* constantly provides the user a *preview* of what the camera is recording, in order to so,

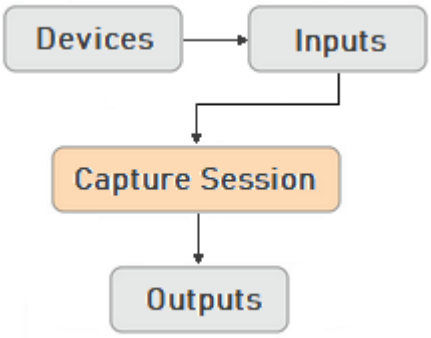


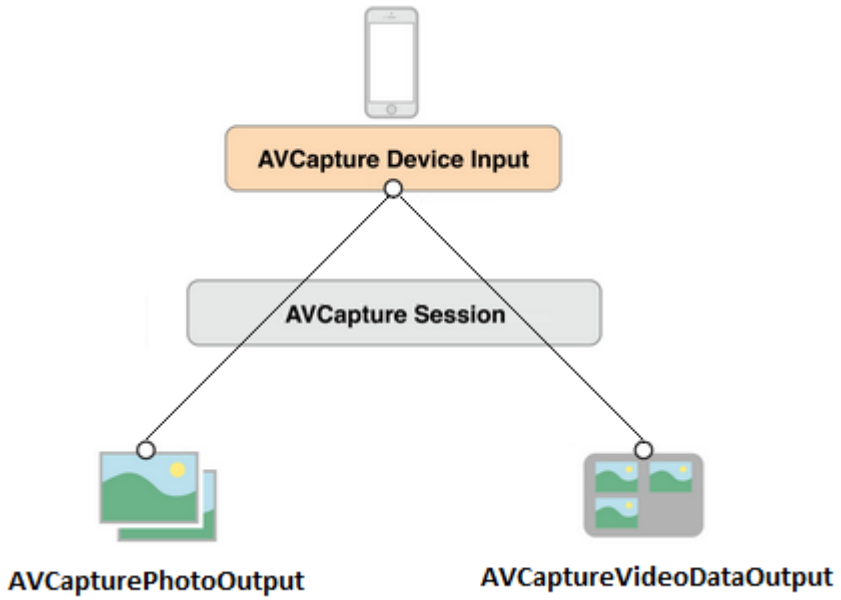
Figure 3.4: Camera Capture Flow Diagram. Sequence of objects needed during a capture session, following the specification of the `AVFoundation` architecture.

it outputs a video stream by attaching it to a *preview layer*. So as to be able to properly handle the Camera Scene Capture, we use the following objects:

- An instance of `AVCaptureDevice` to represent the physical capture device and configure the properties of the underlying hardware as well as to provide input for capture sessions.
- An instance of `AVCaptureDeviceInput` to configure the ports from the selected input device i.e to provide media from a capture device to a capture session.
- An instance of `AVCaptureVideoDataOutput` to process uncompressed frames from the video being captured or to access compressed frames to output or stream them to a movie file.
- An instance of `AVCapturePhotoOutput` to provide an interface for capture workflows related to still photography, such Live Photo, RAW-format capture and more.
- An instance of `AVCaptureSession` to manage all capture activities, on real-time or off-line, as well as to coordinate the data flow from input devices to capture outputs.
- An instance of `AVCaptureVideoPreviewLayer` to manage the image-based content by showing the user a preview of the video as it is being captured by the selected input devices.

Figure 3.5 shows the configuration of the capture session on *PeakLens-iOS*, given that we do not need to record the audio, we have only considered

one input for the video data coming from the rear camera, however, not only do we need to stream the images captured by the camera but we also need to take pictures that include all the virtual elements rendered on the user's view as well, therefore, we have included 2 outputs.



Source: Apple's Developer Documentation. Still and Video Media Capture.

Figure 3.5: Camera Capture Flow Diagram. A single session can configure multiple inputs and outputs, in our application, one single input AVCaptureDeviceInput and 2 outputs, AVCaptureVideoDataOutput and AVCapturePhotoOutput are needed.

After the session has been properly configured, we need to process the video frames by setting a delegate for the `AVCaptureVideoDataOutput` object, in order to ensure that frames are delivered to the delegate in the proper order, we need to specify a serial *queue* for the video output as well. Frames are delivered as instances of the `CMSampleBuffer`, by default the buffers are created with the camera's most efficient format, however, we have specified a custom output format of `kCVPixelFormatType_32BGRA` or `BGRA` that better suits our purposes.

As explained in Section 3.3, we established iOS 7 as the minimum operating system version, given the fact that from this version and on iOS introduces high frame rate video capture support, thus *AVFoundation* supports full 720p (1280 x 720 pixels) resolution at 60 frames per second (fps) which is more than an ideal delivery rate for augmented reality applications

[5]. It is worth noticing that when no data generation is required i.e we do not need to save images from every frame plus additional meta-data such values from sensors, device's information and image specifications, compulsory for the *scene replay functionality* (see Section 3.6 for further details), *PeakLens-iOS* is able to attain 60 *fps*, in contrast, when data generation is required, we can only process 35 *fps*, nonetheless, we guarantee smoothly mixed scenes in both scenarios.

The second output we have included in the session handles the photo output, we have preset the session with a HD 1280x720 resolution. Photos are also delivered as instances of the `CMSampleBuffer`, for this purpose we created a JPEG representation of the buffer and we have also included all the 2D elements rendered on the user's camera view, such elements are represented as tags on top of the mountain peaks, displaying pertinent information such as names, elevation and more.

3.4.2 Device Motion Tracking

For the sake of tracking the device's motion, we must capture environment-related data from the built-in sensors and trigger events accordingly. In *PeakLens-iOS*, events coming from the accelerometer, gyroscope and magnetometer are extremely important for capturing motion, these events are used to control the magnetic field, the rotation in $^{\circ}/s$ and the acceleration in m/s^2 of the device at some degree of accuracy. Each sensor provides the following specific data:

- **Accelerometer.** A motion sensor that detects the change in movement relative to the current device orientation, in 3 dimensions along the x, y, and z axis, see Figure 3.6.
- **Gyroscope.** A motion sensor that detects the rotation with respect to Earth gravity, in 3 dimensions along the x, y, and z axis, see Figure 3.6.
- **Magnetometer.** A motion sensor that measures the strength of the Earth's magnetic field relative to the device, determining the *heading* and acting as a digital compass.

When interacting with virtual objects, effective *motion tracking* is key, so, to accurately report the position of the object *wrt* the device, iOS accounts for the distance between the motion sensors and cameras as well as the device's place in the world. For such purposes, iOS includes `CoreMotion`,

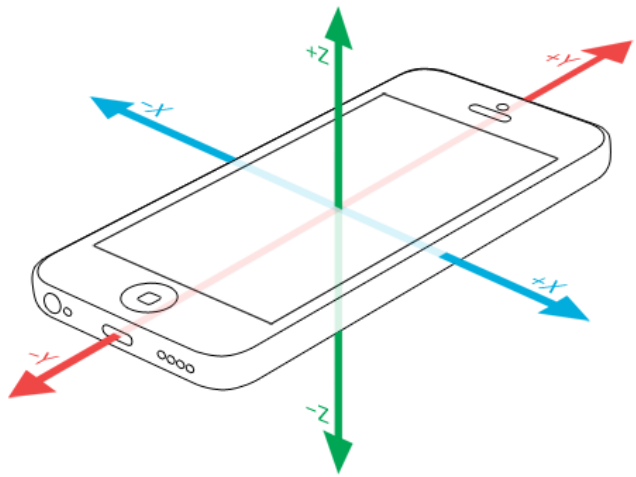


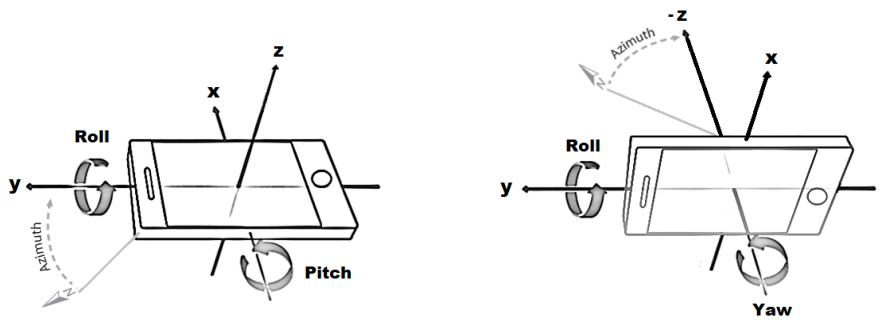
Figure 3.6: iOS Coordinate System. The X-axis runs through the device from left (-) to right (+), the Y-axis through the device from bottom (-) to top (+) and the Z-axis runs perpendicularly through the screen from the back (-) to the front (+).

a framework that makes easy to harness the sensors by reporting all motion events and exposing both raw and a processed values -*those that do not include bias*. Additionally, `CoreMotion` offers a service called `DeviceMotion` that uses the so called *Sensor Fusion Algorithm* to remove bias such as the effect of gravity, in the raw data coming from the *Accelerometer* and *Gyroscope*. Thus, the *device motion* data contains the exact *attitude*, *rotation rate*, *gravity* and *user acceleration* of the device at a specific point in time. It is much easier and safer then, to work with the device motion data, than to compute the values from the raw data.

PeakLens-iOS uses the *Device Motion service* to handle all motion events and collect all the environment-related data, to do so, we must include an instance of `CMMotionManager` in order to get the latest sample of device-motion data at every frame. Once the data has been collected, we compute the *tilt* of the device to efficiently update the position of the virtual objects in the camera view, such information is present on the device's *attitude*. The attitude is included in the `CMDeviceMotion` object within `CMMotionManager` in 3 different mathematical representations: a *quaternion*, a *rotation matrix* and *Euler angles* (*roll*, *pitch* and *yaw*), we use the *rotation matrix* to obtain the correspondent 2D point to re-locate the virtual components in the camera view.

An additional element in the camera view exist, that although it is not considered within the augmented reality framework, it does get affected by

the device's motion, the *Compass*. The *Compass* presented in the camera view uses the sensor's data to show the user's location *wrt* to the north, however, since the device is not used in portrait mode, which is the default mode for tracking location on iOS devices, the *heading* needs to change from the top of the device -*when is held in portrait orientation*-, to the left or right side -*when is held in landscape orientation*-, in order to update the *Compass* with the right values so as to show the true *heading*, we do not longer use the Y-axis but the Z-axis to reflect the true heading, Figure 3.7 depicts this new configuration.



(a) (b)

Figure 3.7: Azimuth on iOS Devices. (a) Device held in portrait orientation and azimuth measured *wrt* to the Y-axis (roll). (b) Device held in landscape orientation and azimuth measured *wrt* to the Z-axis (yaw); the final landscape orientation, say left or right, changes the offset.

Once the *heading* has been aligned, what remains to be done is merely computing the *azimuth*, that is, the angle between the device's reference frame and a reference vector pointing towards the *north*. Further explanation about the reference frames used on iOS devices will be given in Section 3.6, for now, it is enough to understand how the azimuth is computed.

- When handling the device on landscape-left -*home button on the right side*-, yaw goes from -PI to PI, with north at 0° and going to PI, counterclockwise, and to -PI, clockwise [2], therefore only a basic normalization is needed.

- When handling the device on landscape-right -*home button on the left side*- all measures have an offset of 180°, from PI to -PI [2].

3.5 Peak Detection

The Peak-detection stage comprises (a) the *localization*, based on the device's on-board sensors, (b) the *panorama matching*, to align the user's current location and obtain the peaks inside the panorama, and (c) the *rendering*, to display a 2D element for each matched peak.

3.5.1 Sensor-based Localization

Sensor-based localization describes the process of getting the device's geographic location and orientation based only in the device's built-in hardware, which includes the GPS, WiFi, Bluetooth and others. On iOS, the *location* information is available through the `CoreLocation` framework and `CLLocationManager` is the entry point to all its functionalities [2], however, unlike `CMMotionManager`, `CLLocationManager` do not expose the sensors nor their raw data in any form, instead, it provides processed data considering only what best suits the request i.e if the desired accuracy, in *kilometers* or *meters*, is big enough, the `CLLocationManager` has the flexibility to turn off GPS hardware and rely solely on the WiFi or cell radios, which can lead to considerable power savings. Although we are not permitted to choose the sensors used during the triangulation of the user's location, `CMMotionManager` guarantees algorithms that effectively capture data considering not only accuracy but speed and energy efficiency, which is crucial for applications that are constantly monitoring the user's location.

PeakLens-iOS uses an instance of `CLLocationManager` to configure, start, and stop all location services, and accounts for the following location-related activities:

- Tracking changes in the user's current location.
- Reporting heading changes from the on-board compass.

To handle all location and motion activities within our application, we included a *tracking manager* component named `TrackingManager` which acts as a delegate of `CLLocationManager` i.e it receives all location-related events. Events triggered by the built-in sensors work under the following settings:

- **Desired accuracy**, that tells the framework what level of accuracy we expect, the higher uses GPS information, the lower, only cell tower data, although the receiver does its best to achieve the requested accuracy; the actual accuracy is not guaranteed. We have considered

necessary to use as desired accuracy `kCLLocationAccuracyBest` i.e to use GPS information whenever possible.

- **Distance filter**, that tells the framework the minimum distance -*measured in meters*- a device must move horizontally before an update event is generated, this distance is measured relative to the previously delivered location. We have set this property to 50 meters, since the peaks included in the panorama -*on 360°*- do not change significantly in smaller distances.

Once a location-related event that fits the previous configurations is received, the following step is to send the location data, said, *altitude*, *latitude* and *longitude*, to the *panorama matching* service defined in Section 3.5.2, to obtain all peaks inside the field of view, to later draw their correspondent augmented content. Certainly, we do not call this service indistinctly, instead we carefully filter out the events that do not reflect a real change in the user's panorama, to do so, we discard all events coming from the cached measurements and we only consider those in which the radius of uncertainty for the location is less than 60 meters.

3.5.2 Panorama Matching

We refer to *Panorama Matching* as the process of identifying the mountain peaks inside the current panorama. We heavily rely on the framework presented in [47], that proposes a sensor data matching algorithm by performing an initial estimation based only in the raw sensor data to later refine the peak's positioning using the 3D model of the Earth. Additional algorithms are included in this framework to better align the digital skyline coming from the sensors and camera, and the real panorama coming from the Earth's model, however, we have not included them in this work due to timing constraints.

We reckon the need of an additional module in charge of executing the *Panorama Matching* process, to fulfill such requirement we have implemented *PeakLensCV-iOS*, an iOS framework thought to work with alignments based on sensor's and motion data to then use some Computer Vision techniques to refine the peaks positioning in the extracted panorama, at this point we want to remind the reader that we have not developed the Computer Vision module at its fullest, however, we designed *PeakLensCV-iOS* architecture in such a way that can be easily extended.

The most widely used library for Computer Vision tasks is *OpenCV*,

an open source computer vision and machine learning software library used to provide a common infrastructure for computer vision applications [7]. Currently, *OpenCV* offers interfaces for C++, Java, MATLAB and Python, to work with *OpenCV* in iOS or OS X, we need to create an Objective-C or C wrapper so as to be able to use the C++ interface.

PeakLensCV-iOS uses an online service, hereinafter *PeakLensRenderer* [11], that retrieves a list of peaks given a latitude, longitude and the desired panorama width e.g 360° (a panorama is a representation of the skyline with all the peaks included in their respective positions). To this end, a reliable Internet connection is required for proper functioning. Each returned peak includes name, elevation, distance from the user and position in the skyline. It is worth noticing that the peak's position in the skyline is retrieved in 2D space, however, its position in the real world is estimated in the 3D space.

PeakLensCV-iOS is invoked in 2 different times. *First*, when a location-related event is received, at this point, takes the data coming from the service *PeakLensRenderer* to get the peaks inside the panorama. *Second*, on every frame captured by the camera, in this particular case, takes the device's orientation as well as the rotation matrix that represents it, to *correctly* align all peaks inside the panorama, and discard those that lie out of the camera view, this information is later used in the previously described AR framework described so as to make the augmentation precise.

3.5.3 Peak Rendering

When the application starts, the *panorama matching service* retrieves a set of peaks -*if any*- based on the user's current location, the following step is then, add this information and smoothly blend it into the camera view so as to provide the *augmented reality* experience.

In *PeakLens-iOS* we call *Peak Rendering* the process of displaying such additional information as 2D virtual objects called *annotations*, Figure 3.8 illustrates how annotations are rendered on the screen.

Each annotation is created using the *augmented reality* framework introduced in Section 3.4, thus, to represent the peak in the virtual environment we need to use an instance of `ARAnnotation`. The `ARAnnotation` object contains the peak's *id*, *name*, *elevation* and its *position* as a 2D point that represents the position of the peak in screen coordinates, however, to actually render the annotation on the screen, every `ARAnnotation` needs to be embedded in an `ARAnnotationView`. All `ARAnnotationView` are added to

Figure 3.8: Peaks AR Annotations. Camera view containing a single annotation for every detected peak, each annotation shows the peak's position, name and elevation.

a data source, named `ARViewController`, that at any moment is able to retrieve the list of views that are currently visible in the camera image.

As mentioned before, the user's location and device motion are constantly monitored, consequently, what is shown in the screen should also be constantly updated, we manage to do this in a very simple but effective way, the `ARViewController` updates in every frame what is shown and what is not. When new information of the *matching* service is received, the `ARViewController` verifies if new peaks are present within the set retrieved by the service, if so, adds them to the data-source, it also verifies if the peaks present in the data-source have changed their position *wrt* the previous recorded information, in any case, since annotations keep a strong reference to their respective `ARAnnotationView`, the (re)positioning of the views runs smoothly.

`ARViewController` contains all annotations subsequently fetched after the initialization of the application, furthermore, annotations that are not currently within the device's field of view are not removed but only their visibility status is changed. Although keeping all annotations in memory could be perceived as costly or detrimental to the application's performance, it is not, keeping the annotation's view and updating its position is less costly than removing the views and creating them from scratch after each

frame, this of course works under the assumption that the number of peaks inside the user's field of view will never grow exponentially causing memory crushes, which is intuitively true since the panorama is not likely to change drastically within the same scene.

3.6 Standardization Module

Android devices running the *debug* version of PeakLens [10] have a very interesting feature called *capture and replay*, which is nothing but the capability of storing an outdoor usage session as a set of geo-tagged images to later reproduce them simulating their performance in controlled conditions for *debug* purposes. The meta-data included in the images as geo-tags is used to feed the external Computer Vision module -*meant to be used in PeakLens-iOS as well*- presented in Section 3.5.2. The device's orientation, the user's location and additional information are used to identify the peaks in each frame and to display the augmented content as annotations. This functionality becomes of greater importance for evaluation purposes since it allows for a visual inspection of the results thrown by the application in lab conditions, by means of a simulation of the application execution on the field, taking in as input all captured data streams [31]. For the purposes of this study, the development of a complete *capture and replay* module was not required, thus, *PeakLens-iOS* does not include such feature.

To let the Computer Vision module presented in Section 3.5.2 be capable of processing data indistinctly from the operating system it comes from, and to be able to replay usage sessions taken with both Android and iOS applications, and examine all captured meta-data, we should tag images in a standardized format so as to guarantee that the Android application, for instance, can recognize and understand the information inside the meta-data files and vice-versa. Although some information included in the meta-data is generic, for instance, the *FOV*, *latitude*, *longitude* and *altitude* among others, there is other information that cannot be used directly in any other operating system since its raw values are meaningful only in the operating system in which they were generated, thus, further processing is required.

To draw 3D/2D geometry on the screen, both operating systems use the device's attitude in any of its mathematical representations, such *Euler angles*, *quaternions* or *rotation matrices*, since conceptually this information is standard there should not be a need for major processing on this information, however, there is a mismatch in how these values are computed and it lies on the *reference frame* used by each operating system. A reference frame

is a basic orientation of the device's coordinate system taken as a reference. Both frameworks, iOS and Android, see the world in different ways, thus, the device's attitude cannot be used indistinctly. In the following section we review in detail how each operating system use the *reference frame* so as to provide a cross-platform representation of the os-specific data.

3.6.1 Android's Coordinate System

In Android, the framework in charge of monitoring positioning and three-dimensional device movements is called *Sensor*, the *coordinate system* used by the Sensor API is defined relative to the device's frame of reference and it is always based on the natural orientation of the device (some devices include screens with orientations that are naturally landscape e.g. tablets), hence, the sensor's coordinate system never changes.

Figure 3.9 illustrates Android's coordinate system in which the axes are distributed as follows:

- The **X** axis is horizontal and points to the right.
- The **Y** axis is vertical and points up.
- The **Z** axis points toward the outside of the screen face.

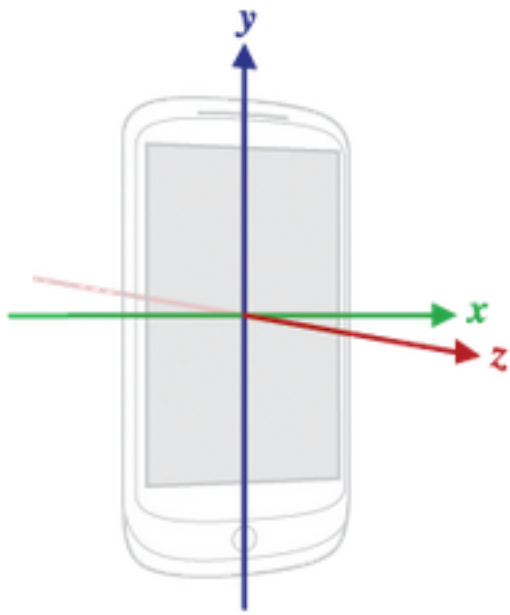
In the context of its sensors, the Android operating system is set up to calculate a rotation matrix **R** which is defined as follows:

$$\mathbf{R} = \begin{bmatrix} -E_x & -E_y & -E_z \\ N_x & N_y & N_z \\ G_x & G_y & G_z \end{bmatrix} \tag{3.1}$$

In the Rotation matrix defined in equation 3.1, **E** represents a unit vector that points *East*, **N** a unit vector that points *North* and **G** a unit vector that points away from the center of the earth (Gravity vector).

In Android, the rotation matrix from `SensorManager.GetRotationMatrix` translates from device to world coordinates [1], therefore **R** transforms a vector from the device's coordinate system to the world's coordinate system which is defined as a direct orthonormal basis, where:

- **X** is defined as the vector product Y.Z (It is tangential to the ground at the device's current location and roughly points to the *East*).



Source: Android's Developer Documentation.

Figure 3.9: Android's Coordinate System. Device in portrait-mode.

- **Y** is tangential to the ground at the device's current location and points towards the magnetic *North Pole*.
- **Z** points towards the sky and it is perpendicular to the *ground*.

R would be the identity matrix if the device is aligned with the world's coordinate system, that is, if the device's **X** axis points towards the *East*, the **Y** axis points towards the *North Pole* and the device is facing the sky.

3.6.2 iOS's Coordinate System

As mentioned in Section 3.4, iOS provides access to its sensors through the `CoreMotion` framework. We can obtain data directly from the *gyroscope*, *accelerometer* or *magnetometer* (suited for our purposes), however `CoreMotion` also includes a *Device Motion* service in charge of processing and refining the raw sensor data coming from both, the *accelerometer* and *gyroscope*, by measuring the device's attitude, rotation rate and the gravity metrics using the so called *Sensor Fusion Algorithm*.

The device's true orientation in space is represented using 3 different models: *Euler angles* (roll, pitch, and yaw), *quaternion*, and *rotation matrices*, each of these is outputted under full concordance to a given reference

frame from which all attitude samples are referenced, a sort of *"zero reference point"* [2]. To retrieve the device's orientation, the iOS framework provides 4 different pre-set configurations, which we explain as follows:

- **xArbitraryZVertical.** The Z axis is vertical and the X axis points in an arbitrary direction in the horizontal plane.
- **xArbitraryCorrectedZVertical.** As above but the magnetometer is used to improve long-term yaw accuracy.
- **xMagneticNorthZVertical.** The Z axis is vertical and the X axis points toward magnetic north.
- **xTrueNorthZVertical.** The Z axis is vertical and the X axis points toward true north.

3.6.3 Cross-platform Generalization

Both, Android and iOS, use the same device coordinate system, however, when obtaining data from their sensors, the *reference frame* in which these data are based is different, therefore the rotation matrices retrieved, and subsequently the Euler angles and/or quaternions cannot be used interchangeably in both operating systems, at least not without any adjustment.

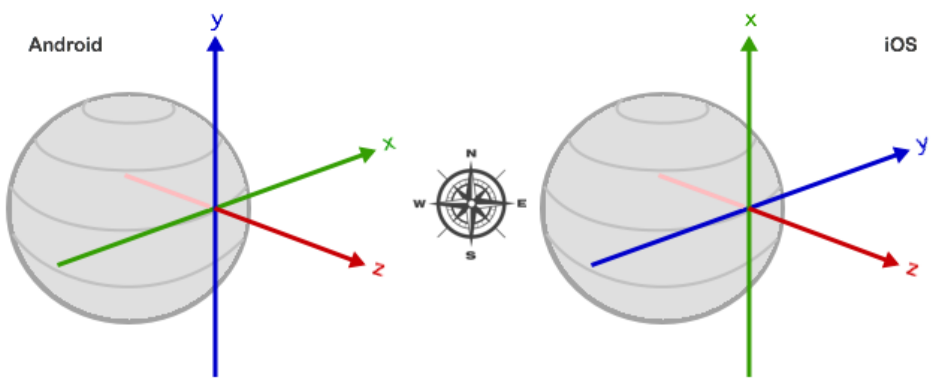


Figure 3.10: Reference Frame in Android vs iOS. Android translates from device to world coordinates. iOS translates from world to device coordinates

The configuration presented in Figure 3.10 assumes that the *reference frame* used by iOS is either `xMagneticNorthZVertical` or `xTrueNorthZVertical`

(the remaining configurations are meaningless for our purposes) whereas the *reference frame* used by Android is the only one provided by its framework. Once we start the sensors readings the reference frames cannot change, thus, the data obtained from the APIs at any moment is, in fact, the actual deviation of the device with respect to its *reference frame* i.e if we want to use data provided by the framework, such as the *rotation matrix*, of one operating system on the other one, we need to apply some transformations.

3.6.4 Strategy

To let the framework presented in Section 3.5.2 be used indistinctly in Android or iOS devices and, to effectively use data coming from one operating system into the other one, the standardization of the rotation matrices is mandatory, thus, we need to consider some important facts:

- In iOS the *rotation matrix* comes from `CMDeviceMotion` and translates from world to device coordinates.
- In Android the *rotation matrix* comes from `SensorManager` and translates from device to world coordinates.

For our purposes, we want all device's orientation data in an absolute frame of reference, such as relative to the earth. To transform the device's orientation data to use an earth coordinate base, we need to use a change of basis matrix. This is also referred to as a rotation matrix. Rotation matrices hold an important property, they are *orthogonal*, thus, their columns and rows are orthogonal unit vectors, a property that allow us to easily invert a matrix by transposing it.

In order to make a *unique* representation of the rotation matrices, we need to use either the rotation matrix provided by Android or the one provided by iOS as the standard rotation matrix, and apply a series of transformations to the other one, so as to make them compatible. To this end, we have chosen Android's rotation matrix as the standard rotation matrix to be used in the Computer Vision module, therefore, the following transformations will be applied to the iOS rotation matrix.

First we inverse iOS rotation matrix (so we can change base from device to world), this operation is rather simple considering that matrices provided by both frameworks are indeed *rotation matrices*, meaning they are orthogonal, transposing them will have the same effect. Additionally, we must align the reference frame in both operating systems, that is, changing iOS device's

motion manager reference frame to be in full concordance with the Android's sensor manager, to this end, we must have the *North* on the **Y axis** and the *gravity* aligned with the **Z axis**, the remaining axis is, therefore, aligned with the *East*. We will achieve this result by applying a rotation of 90° on the Z axis (*yaw*) to each reading of the device's attitude.

The transformations are applied in the following order:

$$\mathbf{R} = Z_\theta \times A^T \tag{3.2}$$

where:

A^T: iOS original rotation matrix transposed.
Z_θ: Rotation matrix of θ on the Z axis.

Applying the rotation of 90° in 3.2 we obtain:

$$R = \begin{bmatrix} \cos 90 & \text{-}\sin 90 & 0 \\ \sin 90 & \cos 90 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \tag{3.3}$$

From 3.3:

$$R = \begin{bmatrix} \text{-}a_{12} & \text{-}a_{22} & \text{-}a_{32} \\ a_{11} & a_{21} & a_{31} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \tag{3.4}$$

The matrix obtained in 3.4, named `correctedRotationMatrix`, is the only rotation matrix used in *PeakLens-iOS*, and it is used to process each frame received by the camera, so as to align the peaks accordingly. Additionally, the `correctedRotationMatrix` is also used when the data needed for the *replay* is being recorded, thus, data produced by iOS can safely be reproduced on the *replay* function of Android devices running PeakLens.

Chapter 4

Intelligent System

In this chapter we describe the *intelligent* system build to enhanced the augmented reality application we have described in Chapter 3 by complementing the ConvNet proposed in [50] with a *second*-ConvNet that is able to improve the detection of *occlusions* in the skyline.

4.1 Occlusion Modeling

The ConvNet described in [50] uses a *Fully Convolutional Network* that has been trained to recognize skylines, this ConvNet is able to achieve considerable results while detecting skylines, however, it clearly has some limitations when dealing with unconstrained images in which a significant percentage of the skyline is occluded. A very obvious explanation of such *phenomenum* is given by the fact that when modeling ConvNets, *occlusions*, represent a very challenging task, for instance, a basic intuition is that an occlusion could be virtually *anything*, from buildings to transmission lines -*or the combination of both*- and even when the object that is occluding the skyline is known such as in Figure 4.1, there is a enormous variety of possible representations i.e. the disposition of the occlusion *wrt* the object being recognized -*the skyline*- is entirely unconstrained.

In this study, we do include the *occlusion-detection* stage in the learning process, unlike the former approaches, and unlike the latter approach we heavily rely on how we train our ConvNet, therefore, before the inference and learning is done, we carefully consider 2 main aspects:

- First, when extracting the occluded regions, we try to find all figure-ground cues that best serve as positive proof for occlusion events,

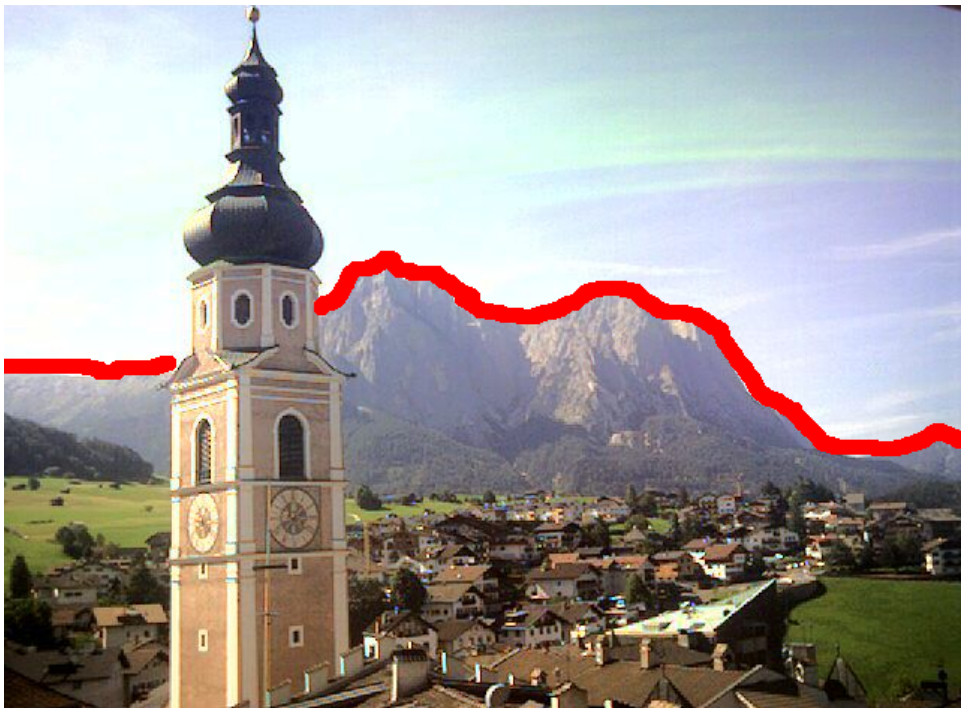


Figure 4.1: Example of a Typical Occluded Scenery. The figure above depicts a skyline in the background interrupted by a steeple in the foreground, notice that the background contains other steeples, however those are not considered occlusions. The ground-truth (manual annotation) is shown in red.

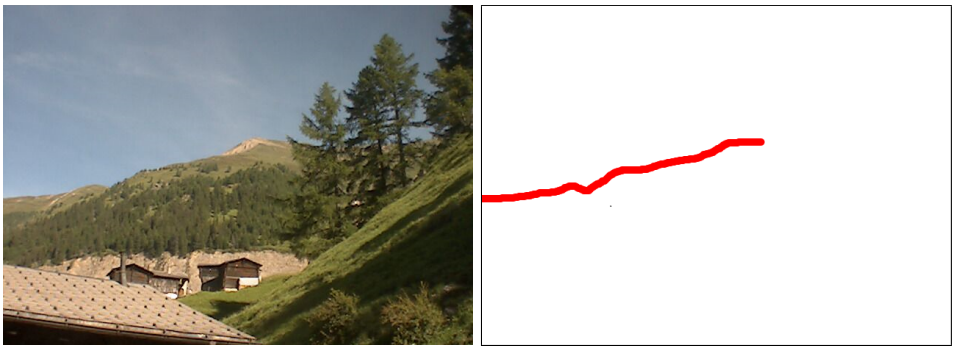
therefore, instead of treating occlusions as part of the background objects, such is the case of terrain and sky -*that are also part of our model as negative samples*-, we try to model the visual aspect of an *occlusion*, this means that after training, a pixel is classified as an occlusion if it falls under a threshold, otherwise it is consider as a skyline.

- Second, we do not use square patches to feed our ConvNet -*as is regularly done*-, instead we propose the use of non-square patches to provide a better *context* to our ConvNet, as explained in Section 4.2.2.

We are aware that occlusions can be confused with background objects, as stated in [51], however, we strongly believe that by combining these 2 key aspects we can overcome most of the miss-predictions that could be generated in other models of such kind.

4.1.1 Labeling

A crowd-sourcing task has been conducted to manually annotate the images on the dataset [50] used in this work. As shown in Figure 4.2 the ground-truth only annotates skylines, all remaining pixels are considered negative, which means, there is no knowledge of what constitutes an occlusion.



(a) Photo (b) Annotation

Figure 4.2: Image - Annotation (Ground-Truth) Pair. (a) A typical scenery interrupted by a set of trees in the most right part of the picture. (b) The annotation or ground-truth -marked in red- of the picture on the left.

The ConvNet presented on [50] is a pixel-level classifier that has been trained to learn whether there is a skyline -*positive samples*- or not -*negative samples*, all possible negative samples are treated indistinctly, it is fair to say then, in that context, the annotation of occluders was not considered necessary. Although this approach has proven to be quite successful when detecting skylines, such broad definition of negative samples has place for improvement in order to predict *occlusions* with better accuracy.

We have not conducted any extra process for explicitly labeling *occlusions*, however, in order to introduce a fine-grained definition of negative samples, we take heed of the *data preparation* stage, as it will be explained in Section 4.1.2.

Within the scope of mountain peak detection applications, we propose the following definitions of **negative** samples:

- **Sky.** All pixels that lie above the skyline.
- **Terrain.** All pixels that lie bellow the skyline.
- **Occlusion.** All pixels that lie within a fixed region. Dimension of such region is delimited on the left by the previously found column

with skyline and on the right, by the immediately following column with skyline, and with parameterizable height.

4.1.2 Image Patch Extraction

As stated in the previous section, we do not explicitly label occlusions in the ground-truth, nevertheless, we fed our ConvNet with different types of negatives samples that were extracted considering the *new* negative classes introduced before.

Our ConvNet was fed with *image patches* instead of *whole images*, for each image patch, we located it on the edge map -*annotation*- and picked the value at its center column as the corresponding ground-truth label. Intuitively, our ConvNet will learn to differentiate patches with occlusion, terrain or sky pixels close to the center.

For this purpose, we have implemented a sampling tool that carefully labels the different types of samples as **positive** and both, *generic* and *in-detail* **negative**. Further characteristics of each type of extracted patch are explained as follows:

- **Positive.** Patches that contain one or more skyline-pixels across the center column.
- **Negative.** Patches that do not contain any skyline-pixel across the center column.

Edge maps cannot provide further information than the one previously presented, since their classification is binary i.e an edge is a skyline, everything else is not, such broad classification is not sufficient when the *occlusion problem* is prevalent, therefore, we need to extend the above configuration to introduce the *new* set of negative classes defined earlier in this section. Figure 4.3 shows an example of how patches are extracted using the new configuration.

Patches will be labeled with these *new* negative classes if one of the premises presented below is true.

- **Sky.** There is one or more skyline-pixels in the center column *below* the region the patch is enclosing.
- **Terrain.** There is one or more skyline-pixels in the center column *above* the region the patch is enclosing.

- **Occlusion.** There exist a bounding box -*with parameterizable height*- in which one of the following premises is true.
  - There is one or more skyline-pixels on the *right* of the region the patch is enclosing but none on the *left*.
  - There is one or more skyline-pixels on the *left* and on the *right* of the region the patch is enclosing, with a gap greater or equal than the patch's width.
  - There is one or more skyline-pixels on the *left* of the region the patch is enclosing but none on the *right*.

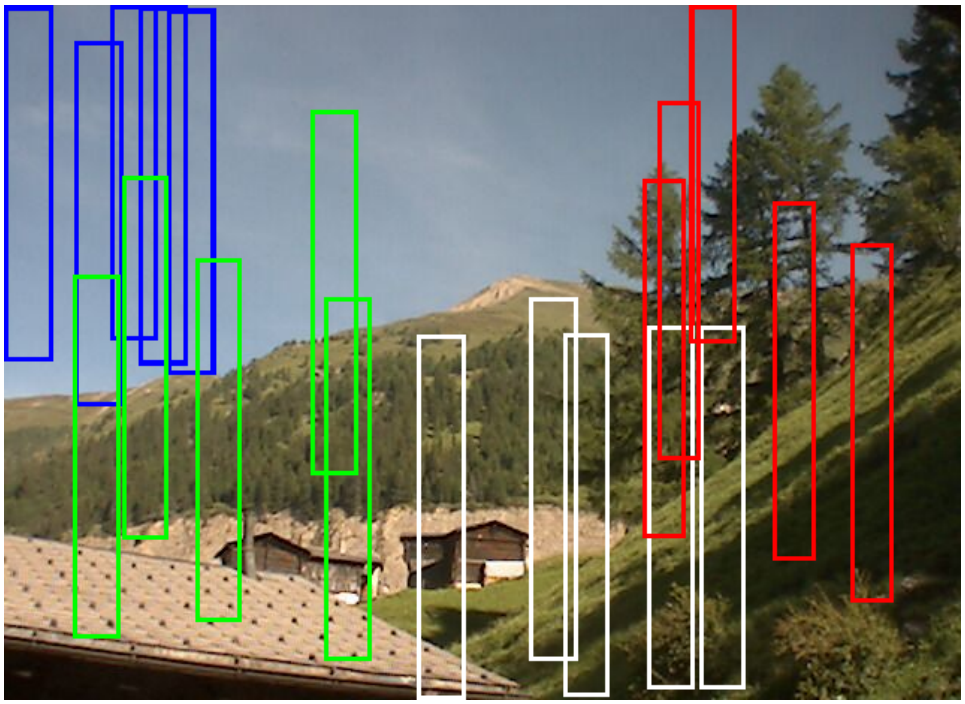


Figure 4.3: Fine-grained Classes. An example of the patch extraction process. In **green**, *positive* patches. In **blue**, *sky* patches. In **white**, *terrain* patches. In **red**, *occlusion* patches.

Our sampling tool considers padding and margin as parameterizable properties among other properties.

- **Padding.** Excludes all skyline pixels that lie in the space between the patch content and its boundaries.
- **Margin.** Adds the number of pixels to be considered on top and bottom when creating the bounding boxes for occlusions classes.

4.2 Heuristics

Common heuristics to detect horizon lines are heavily based on edge detection, which can be done using traditional Computer Vision algorithms or using learning techniques such as ConvNets. The former heavily relies on the pixel's gradient magnitude [21] but is usually combined with other techniques to yield a better result, such is the case of *canny* detection [35]. However, assuming that changes on color intensity are a good measure to detect horizon lines is not always true, a large color gradient can appear within an object and a small color gradient can also appear on its boundaries, thus, such assumptions could lead to non-stable results due to parameter choices. Conversely, ConvNets can learn hierarchical feature representations and encode very particular feature representations within their multiple layer structure. Very interesting approaches have been presented in [19] and [98] demonstrating that the extraction of the horizon line can be made from the classification map.

Within the *occlusion problem* scope, feature-based heuristic methods may work adequately, however, we need to specifically address this hypothesis, otherwise obstacles such buildings, trees, or even sunlight could be detected as edges of the skyline, negatively impacting on the performance of our model. To overcome such problems, we introduce 2 techniques that we consider are key in the occlusion detection problem.

4.2.1 Sampling

As mentioned in Section 4.1.1, we have introduced a fine-grained definition of negative samples, thus, our ConvNet is fed with 4 different classes of patches: *positive*, *sky*, *terrain* and *occlusion*. In the following, we further analyze how we carefully selected the sampling heuristics we used in this study.

Table 4.1 depicts the heuristics used to sample our data, we have considered 3 experiments named *Experiment 1*, *Experiment 2* and *Experiment 3*, and we introduced 2 different heuristics for sampling the negative classes of each experiment, *random* and *distributed*, we have also considered balance our data with 3 different factors, detailed as follows:

- *Experiment 1*, in which patches were sampled as a {positive : negative} ratio of 1:1 e.g one positive sample for each negative sample. In this way, positives and negatives are balanced, meaning we extract the

same amount of positives and negatives as a whole with no discrimination among negatives subclasses, thus, it is possible some negative subclasses are under-represented.

- *Experiment 2*, in which patches were sampled as a {positive : negative-sky : negative-occlusion : negative-terrain } ratio of $1 : \frac{1}{3} : \frac{1}{3} : \frac{1}{3}$ e.g three positive samples for each negative subclass. In this way, the presence of positives takes precedence, meaning we extract more positives patches than specific-negative patches, but we guarantee the same amount among negative subclasses.

- *Experiment 3*, in which patches were sampled as a {positive : negative-sky : negative-occlusion : negative-terrain } ratio of 1:1:1:1 e.g one positive sample for each subclass of negative sample. In this way, we ensure the same amount of positives and each negative subclass.

Name	Type	Factor			
		Positive	Sky	Terrain	Occlusion
Experiment 1	random	1	1		
Experiment 2	distributed	3	1	1	1
Experiment 3	distributed	1	1	1	1

Table 4.1: Sampling Heuristics. Selection of 3 subsets (statistical samples) of patches from within the entire dataset.

Type. Defines the way in which *negatives* samples are extracted.

- **Random.** All 3 negative classes -*sky, terrain and occlusion*- are created from each image when possible, however, since the discrimination of negatives subclasses is not needed, patches are extracted randomly and no guarantee of the same distribution among negative subclasses is provided. When using a random distribution, the 3 different negative subclasses are grouped and labeled as one, therefore, our ConvNet learns to differentiate 2 classes only, *positive* and *negative*.

- **Distributed.** All 3 negative classes -*sky, terrain and occlusion*- are evenly extracted from each image when possible. We ensure the final distribution of negative subclasses to be the same. When using a balanced distribution, our ConvNet learns to differentiate 4 classes, *positive*, *sky*, *terrain* and *occlusion*.

Factor. Defines the amount of generated patches for each class, this number determines if a dataset is balanced aka even -*when all the classes are evenly represented*- or imbalanced -*when the distribution among the different classes is imbalanced*- [92]. It has been *empirically* demonstrated that balanced datasets outperform imbalanced datasets [56] [97] by reducing the bias towards the most-represented classes, however, most learning techniques assume a balanced distribution, thus, when the training data and the unknown data that needs to be classified have a different distribution they usually under-perform [56], for instance, if we only aim to minimize the overall error rate, the under-represented classes, *occlusions*, would be discarded.

In practice, the available data is often imbalanced, this is particularly true in our context where the fraction of occluded columns is minimal *wrt* the non-occluded columns. Nonetheless, several methods exist to lessen the impact of imbalanced datasets, in this work we use sampling techniques to create balanced distributions by extracting only a portion of the available classes, such technique is *aka* under-sampling -*removing instances from over-represented classes*-. Although imbalanced datasets may under-perform balanced datasets, we have also included an experiment with imbalanced datasets.

- **Balanced datasets.** *Experiment 1* and *Experiment 3* consider 1 positive sample for each negative sample e.g. 1:1.
- **Imbalanced datasets.** *Experiment 2* considers 1 positive sample for each type negative sample e.g. 3:1, this is used only in the *distributed* heuristic.

4.2.2 Column-wise Classifier

The *baseline* ConvNet proposed in [50] has been trained using an imbalanced dataset 1:2 *{positive:negative}* composed by square patches in the form of *small RGB images of 29x29 pixels*. These patches are sampled using a random *heuristic*. A set of patches is shown in Figure 4.4 (images have been scaled for visualization purposes). As we can observe, constrained by their fixed dimension, patches hold little information about the scene, and are labeled based on the positivity of the middle pixel.

In this work, we use our proposed ConvNet as a powerful *column* classifier that directly operates on non-square patches, hereinafter called *column-patches*. Conceptually this is very simple, by feeding our ConvNet with

(a) Positive patches

(b) Negative patches

Figure 4.4: Random Square Patches. Generated with [50]. (a) Patches that contain skyline in the middle pixel. (b) Patches that do not contain skyline in the middle pixel. From left to right: sky, terrain and occlusion.

column-patches we convolve the set of learnable filters across the width and height of the *column*, therefore, the network will learn filters that activate with a certain combination of features in the same region size as our columns, intuitively, such combination of features will be captured within a larger *context* if small patches are used. For example, on the first layer, filters will activate if they see simple interruption among sky pixels within the same column. On the second layer, filters could look at complex structures in between sky and terrain, and so forth and so on until occlusions are entirely captured. At the end, we test our ConvNet on images with height equal to the *column*'s height, such the column-wise classifier will output a vector with width equal to the input image width, each element of the vector will predict whether there is an occlusion within the correspondent column or not.

In order to set the *minimum* height for *columns* and not having to stretch the images, we have only considered heights bigger than or equal to 240 pixel, we carefully selected this number by evaluating the images minimum -*114 pixels*- and maximum -*3456 pixels*- height as well as the height distribution among images in the dataset, Figure 4.5 illustrates this information. After some inspection we managed to detect and discard only 82 images from the dataset.

Additionally, in order to choose the most suitable height, we have also considered the concept of *context*, previously introduced in this section, for such matter, we have analyzed the disposition of the skyline pixels within an image, Figure 4.6 shows that the majority of the skyline pixels are located on

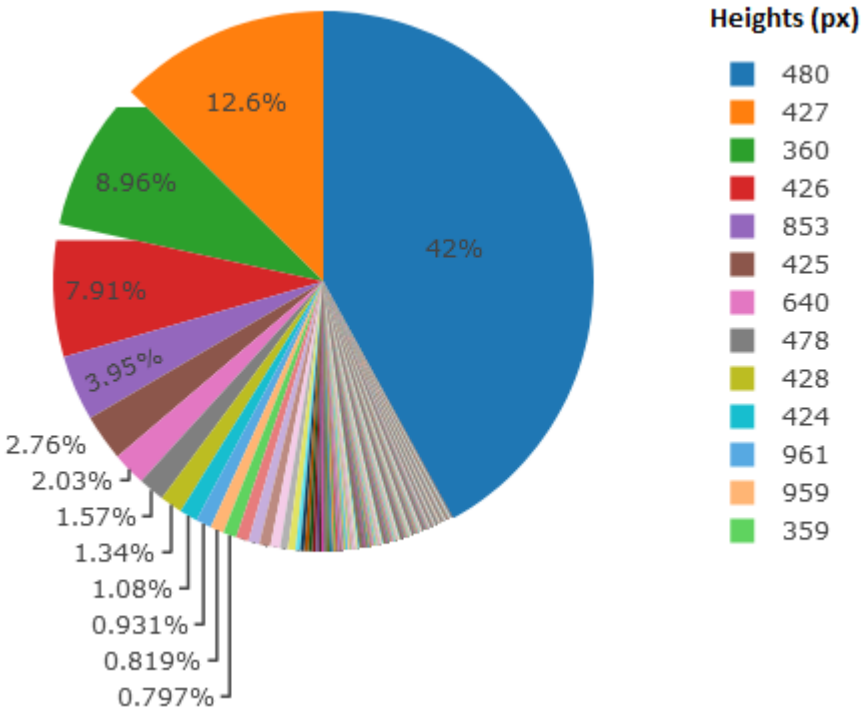


Figure 4.5: Images Height Distribution on Dataset [50]. A pie chart of the height distribution of all images in dataset [50]. Almost 50% images have a height of 480 pixels. This distribution was used to manually choose the most suitable height for the non-square patch introduced in this section.

the second and third fourths of the image, this validates the *context* we want to provide our ConvNet with, by showing that the extracted columns will contain skyline pixels evenly distributed along the column height. Figure 4.6 also shows that a significant number of skyline pixels are located in the first row of the image, starting from the left-top corner, however, this is handled by means of a parameter called *padding* defined in our sampling tool described in Section 4.1.2, that specifically avoids considering skyline pixels when these are located very close to the column boundaries.

Finally, our *columns* are RGB images of 29x240 pixels, Figure 4.7 shows a set of columns extracted by our sampling tool.

4.3 ConvNet

As stated in Section 4.2, ConvNets have an interesting property that make them suitable in solving problems such edge detection, since edges are usu-

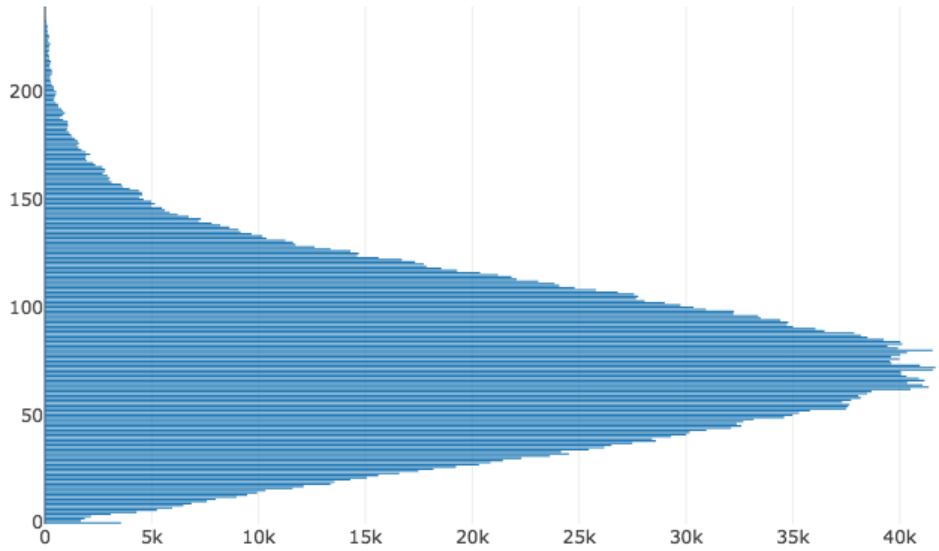
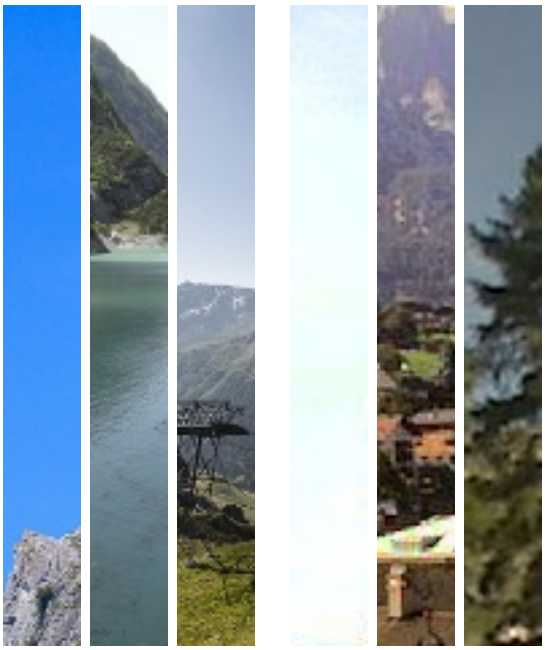


Figure 4.6: Skyline Distribution on Dataset [50]. The figure above shows the distribution of skyline pixels across all images in the dataset, we have considered a standardize height of 240 pixels for demonstration purposes. The row number (from 0 to 240) is shown in the vertical axis and the number of skyline pixels is shown in the horizontal axis (from 0 to approximately 40k).

ally locally correlated and exhibit specific patterns, such as straight lines, corners, T-junctions and Y-junctions, ConvNets can capture such hierarchical patterns within their *convolutional* layers.

In this section we present a small -*in terms of memory footprint*- and easy to follow architecture based on LeNet [73], it is worth mentioning that we have made some alterations on LeNet's architecture to better suit our purposes, such alterations include the transition from 1 to 3 input channels and from a *fully-connected* network to a *fully-convolutional* network.

Fully Convolutional Network (FCN). One of the main characteristics of a ConvNet, when trained with significant and representative amount of data, is what commonly is referred as *translation invariance*, which is the ability of a ConvNet to properly recognize objects regardless if their appearance varies in some way -*e.g. translation, rotation, size, illumination, etc.*-, therefore, it allows to abstract an object's identity from the specifics of the visual input by operating on local regions, and depending only on relative spatial coordinates. This is a very attractive property when capturing *occlusions*, since all occluded pixels lie within the local spatial coordinates, and



(a) Positive columns (b) Negative columns

Figure 4.7: Random Column Patches. (a) Columns that contain skyline in the center pixel. (b) Columns that do not contain skyline in the center pixel. From left to right: sky, terrain and occlusion.

do not depend on the full spatial coordinates. Furthermore, *fully-connected* layers generally cause loss of local spatial information, since the moment the inputs are obtained from the last *convolutional* layer, the spatial arrangement of the feature map is discarded and each *convolutional* layer output is connected to each input neuron, therefore, general ConvNets (those that contain *fully-connected* layers) learn a general nonlinear function, instead, FCNs learn a nonlinear filter [76], that is, even the decision-making layers at the end of the network are filters.

As stated before, the only difference between *fully-connected* layers and *convolutional* layers is how neurons are connected, furthermore, in *convolutional* layers many of the neurons share parameters, hence, any *fully-connected* layer can be converted to a *convolutional* layer For instance, the first *fully-connected* layer in LeNet, *ip1*, has 500 outputs and is looking for an input volume of size **7x7x512**, it could be equivalently expressed as a *convolutional* layer with **F=7**, **P=0**, **S=1**, **K=500** where `F` is the spatial extent, `P` is the zero-padding, `S` is the stride and `K` is the number of filters. In other words, we are setting the filter size to be exactly the size of the input volume to make the output size be **1x1x500** giving an identical result

as the initial *fully-connected* layer.

In this study, we did not simply replace the 2 *fully-connected* layers on LeNet for one identical *convolutional* layer, instead we have changed the number of filters and, therefore, the kernels and their respective strides which allowed us to keep reasonable computational requirements as shown in Section 4.3.1. Additionally, by not adding any *fully-connected* layer in our architecture, we could *potentially* use the network on images of virtually any size as opposite of *fully-connected* networks that expect inputs of a predetermined dimension. However, it is important to *stress* that our classifier was thought to work with images of a fixed height (240 pixels), so as to only perform horizontal convolutions that allow us to have a unique prediction for each column, to that end, the *kernel*'s sizes in our model obey a non-square shape in some convolutional layers to better represent the column-based patches described in 4.2.2.

4.3.1 Architecture

The architecture of our network is depicted in Figure 4.8, bellow are listed all the alterations made on LeNet's architecture along with the main characteristics of our own network:

- Instead of using `TANH` as activation function, we use `RELU` since it tends to give much better classification accuracy due to sparsity and reduced likelihood of the gradient to vanish i.e. the gradient has a constant value that results in faster learning.
- Given the non-square shape of our training *patches*, we have changed the size of the *kernels*, from squared ones to rectangular ones, in both, *convolutional* and *pooling* layers.
- We used an input image with 3 channels (RGB images) instead of 1 channel (gray-scale images), given that in the edge detection context, variation in the color scale can enclose important information regarding the object's boundaries.
- Instead of inserting a MAX-pooling layer in between convolutions, that results in a much faster reduction of the spatial size, we have placed a single MAX-pooling layer in the middle of the architecture.
- We have removed the 2 *fully connected* layers and replaced them with a single *convolutional* layer, making the whole architecture a *Fully Convolutional Network* for the aforementioned reasons.

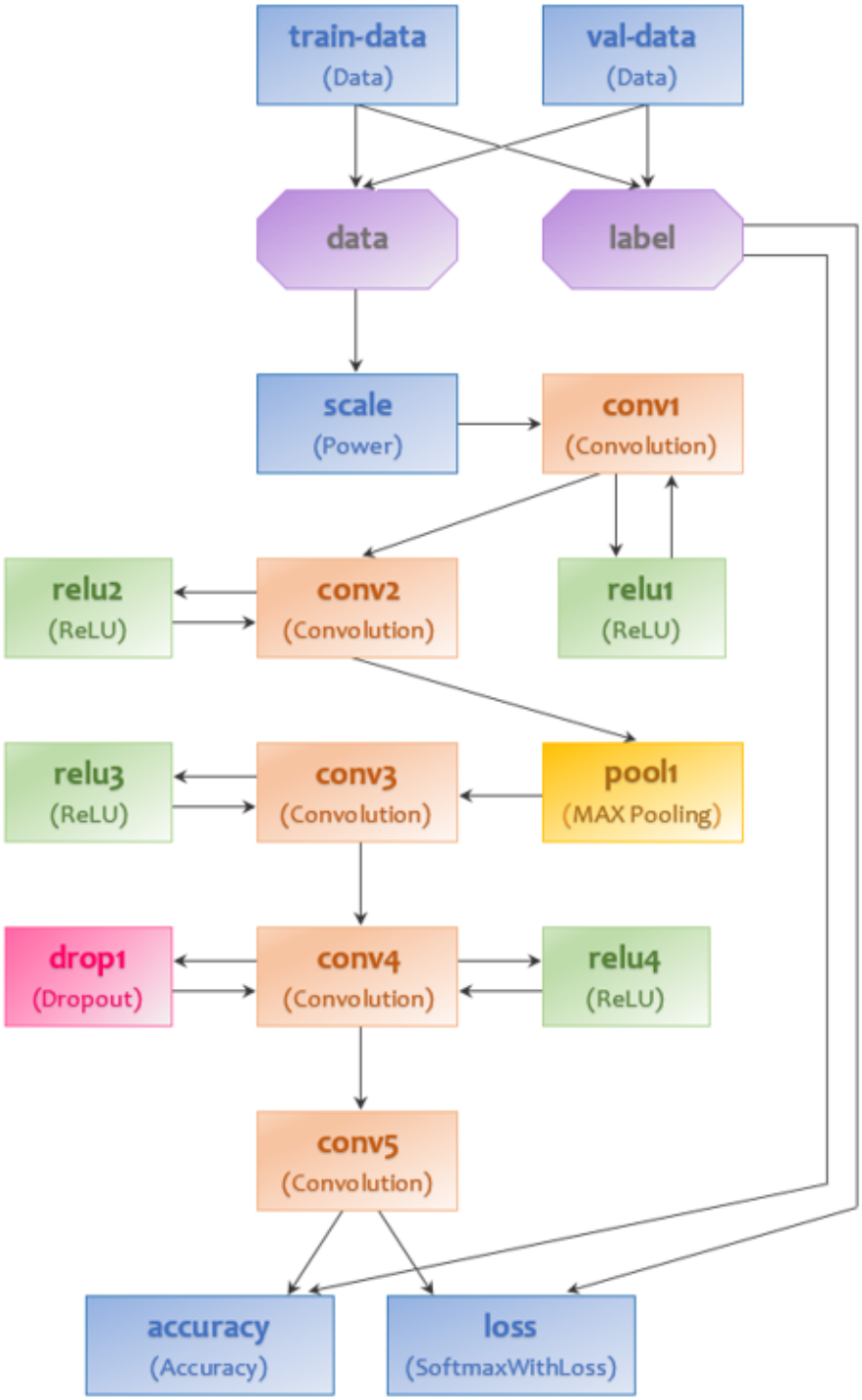


Figure 4.8: Proposed ConvNet Architecture. The proposed architecture consists of 2 sets of convolutional and activation (RELU) layers, followed by a max pooling layer, another 2 sets of convolutional and activation (RELU) layers followed by a dropout (for reducing over-fitting), another convolutional layer (as a decision-making layer) and finally a soft-max classifier (to convert the output of the last layer into a probability distribution).

It is worth noting that we have experimented with multiple variations of different state-of-the-art networks, such as *AlexNet* [68], *FaceNet* [89] and *SqueezeNet* [61], however, we encountered that in some cases the error curve converged and in others the error rate on the validation set underperformed the one already presented in this section, training was stopped in both scenarios.

Table 4.2 shows the bias, kernel's sizes, strides and weights generated after each convolution for an input RGB image (3 channels) of 29x240 pixels. After training is completed, the network has a total of **1 001 538** weights when is trained to predict 2 output classes -*positive and negative*- and **1 001 732** when is trained to predict 4 output classes -*positive, sky, terrain and occlusion*-.

Name	Output			Bias	Kernel		Stride		Weights
	H	W	C		H	W	H	W	
INPUT	240	29	3						
CONV-1	236	25	16	16	5	5	1	1	1 216
CONV-2	228	21	32	32	9	5	1	1	23 072
POOL-1	38	7	32	33	6	3	6	3	
CONV-3	30	5	64	64	9	3	1	1	55 360
CONV-4	1	1	96	96	30	5	1	1	921 696
CONV-5 (a)	1	1	2	2	1	1	1	1	194
									1 001 538
CONV-5 (b)	1	1	4	4	1	1	1	1	388
									1 001 732

Table 4.2: Proposed ConvNet Architecture. The proposed architecture include 4 Convolutional Layers (CONV-1, CONV-2, CONV-3, CONV-4), 1 Max Pooling Layer (POOL-1) and a final Convolutional Layer (CONV-5) replacing the commonly used Fully Connected Layer present on architectures such as LeNet. **CONV-5 (a)** Final Convolutional Layer of the 2-class predictor. **CONV-5 (b)** Final Convolutional Layer of the 4-class predictor.

4.3.2 Hyper-parameters

In order to improve the *Accuracy* of our model, we have experimented with several hyper-parameter variations. In these experiments we tuned the *learning rate* and *batch size* accordingly, taking into consideration the rules of thumb for each *optimization method* used. Nevertheless, we do not claim this work to be an in-depth exploration of the space of the hyper-parameters of our model, however, we attempted to have a reasonable degree of sensitivity on the main parameters. Please refer to Appendix A for more details on the configuration of the Hyper-parameters.

4.3.2.1 Optimization Method

Gradient descent is one of the most popular algorithms to perform optimization and by far, the *default* choice for neural networks optimization. As such, state-of-the-art deep learning frameworks provide several implementations of common algorithms used to optimize gradient descent. When choosing an optimizer, specific properties of the addressed problem should be considered, such as the type of data and their class distribution, the type of layers composing the network as well as its depth, for instance, we will probably benefit from per-weight learning rates if the network is deep.

The academic literature seems to mainly suggests SGD and Nesterov as solid choices when dealing with shallow networks, and Adam and RMSprop for deeper networks. Conversely, choosing one over the other does not yield significant boost in performance, making them a sort of black-box optimizers.

In Caffe [62], in order to improve the *loss*, model optimization is done through a parameter call *Solver Type*. The *solvers* included in Caffe are listed below.

- Stochastic Gradient Descent (type: SGD).
- AdaDelta (type: AdaDelta).
- Adaptive Gradient (type: AdaGrad).
- Adam (type: Adam).
- Nesterov's Accelerated Gradient (type: Nesterov).
- RMSprop (type: RMSProp).

How these *Solvers* differ among each other lies on how much data are used to compute the gradient of the objective function, so a trade-off between the accuracy of the parameter update and the time it takes to perform an update, needs to be made. Caffe forces *Solvers* to use the *mini*-batch strategy in order to overcome performance issues, hence, a proper combination of *learning rate* and *batch size* needs to be addressed.

In our experiments, although convergence was noisier while using *SGD*, we managed to obtain good results at **Experiment 2** and **Experiment 3** using *SGD*, in contrast, better results in **Experiment 1** were shown when using *Nesterov*. Although there was an improvement in terms of *Accuracy* and *loss*, it is important to stress that this improvement was minimal, which led us to conclude that both gradient variances will perform similarly within the same-*ish* ConvNet architecture.

4.3.2.2 Batch Size

Batch size defines the number of samples that are going to be propagated through the network. Typically, networks train faster with *mini*-batches, that is because the weights are updated after each propagation. Batches fully leverage the GPU, hence, have a huge impact not only on memory consumption but training time efficiency too, thus, using batches instead of individual training samples allows for greater parallelism.

A common intuition is then, the smaller the **batch** the less accurate is the estimation of the gradient i.e gradients will become more unstable. Conversely, a *batch* size too big will cause gradients to become less noisy but it will affect the rate at which the gradient converges as well as the quality of the final solution [66]. In theory, when you reduce the *batch-size* by a factor of **X** then you should increase the **learning rate** by a factor of **sqrt(X)** to keep the variance in the gradient expectation constant, however, as stated in [66], using a factor of X yields to promising results despite what theory suggests.

In our experiments, we have successfully used batches of **512** -*for training*- and **256** -*for validation*- in all our proposed experiments. We did not benefit from using larger *batch* sizes, in any case, the performance achieved was the same as with the smaller ones, thus, we kept the mini batches.

4.3.2.3 Learning Rate

Choosing a proper *learning rate* poses a rather difficult task. If the learning rate is low, training is more reliable, however, since the steps towards the minimum of the loss function are really small, the model would get sensitive to *high-frequency* noise in the data, therefore, arriving at the area close to the *minima* might cause *over-fitting*. Instead, if the learning rate is high, training may not easily converge or even diverge, intuitively, steps will be far apart enough to possibly miss the area near the local *minima* and make the loss worse, furthermore, the model could become insensitive to the data itself and might cause *under-fitting*. Consequently, when using batch training, a common convention is to rescale the *learning rate* as the *batch size* changes [90], such scaling process should be done accordingly.

Given that at the beginning of the training process, random weights are far from optimal, we could start from a relatively large *learning rate*. However, as proposed in [66], we could also start the training process with a low *learning rate* and increasing it exponentially for every batch, to then select the one with the fastest decrease in the *loss*.

In our experiments, we have followed the suggestions proposed by [66] and have started with a low value of 0.1 and later, exponentially lower the values to 0.01 and 0.001. We stopped the iterations when the *loss* began to reach an apparent *plateau*, in all our proposed experiments the best *learning rate* was **0.001**.

4.3.3 Execution

We deployed our ConvNet using DIGITS [101], since it simplifies the data management, training, visualization and performance monitoring of the network in real time. DIGITS includes several open-source frameworks for defining and training ConvNets such as Theano [30], Torch [37], Google's TensorFlow [18] and Caffe [62]. Although learning time depends, among others, on the architecture of the ConvNet, we have chosen to work with Caffe since it is one of the most matures frameworks.

Due to availability issues, we ran 3 experiments in 3 different servers, experiment **Experiment 1** on **Server 1**, experiment **Experiment 2** on **Server 2**, and **Experiment 3** on **Server 3**, a detailed information about the results of each experiment will be given in Chapter 5. Table 4.3 summarizes information about the servers.

Server	CPU		GPU			Digits	Caffe
	Name	f	Name	Mem.	#	ver.	ver.
Ubuntu 16.04.1	i5-7640X	4.0GHz	GeForce GTX 1080	8G	1	5.0	0.16.4
Ubuntu 16.04.1	i7-7800X	3.5GHz	GeForce GTX 1080Ti	11G	2	6.0	0.15.14
Ubuntu 16.04.4	E5-2690V3	2.6GHz	Tesla K80 GK210GL	12G	2	6.0	0.15.14

Table 4.3: Server Specifications. 3 different servers were used for training the ConvNet presented in this chapter, relevant technical specification about the hardware and software are shown above.

Table 4.4 illustrates additional information regarding the execution of the 3 conducted experiments, such as the training time and the size of the file containing the model weights generated by the framework, named *.caffemodel. *Experiment 2* and *Experiment 3* have the same network architecture and use similar amounts of data, however, their training time is significantly different. Although **Server 3** -*used for Experiment 3*- is the most powerful server we have used in this work, the training time is slower than the others because training samples were read from a HDD instead of a SSD.

Experiment	Server	Training time	Model size
Experiment 1	1	18:49:00	3.82 MB
Experiment 2	2	14:21:00	4.00 MB
Experiment 3	3	22:07:00	4.00 MB

Table 4.4: Execution's Statistics. General stats for each experiment. Training time (in Digits) of the ConvNet proposed in this chapter, and the size in disk of the model weights file.

4.4 Combined Model

Throughout this thesis, we have reviewed existing skyline detection models and analyzed their performance, this understanding yielded us to propose

the ConvNet described along this chapter. We now turn to the task of explaining how we leverage the capability of our model by embedding it into the model proposed by [50].

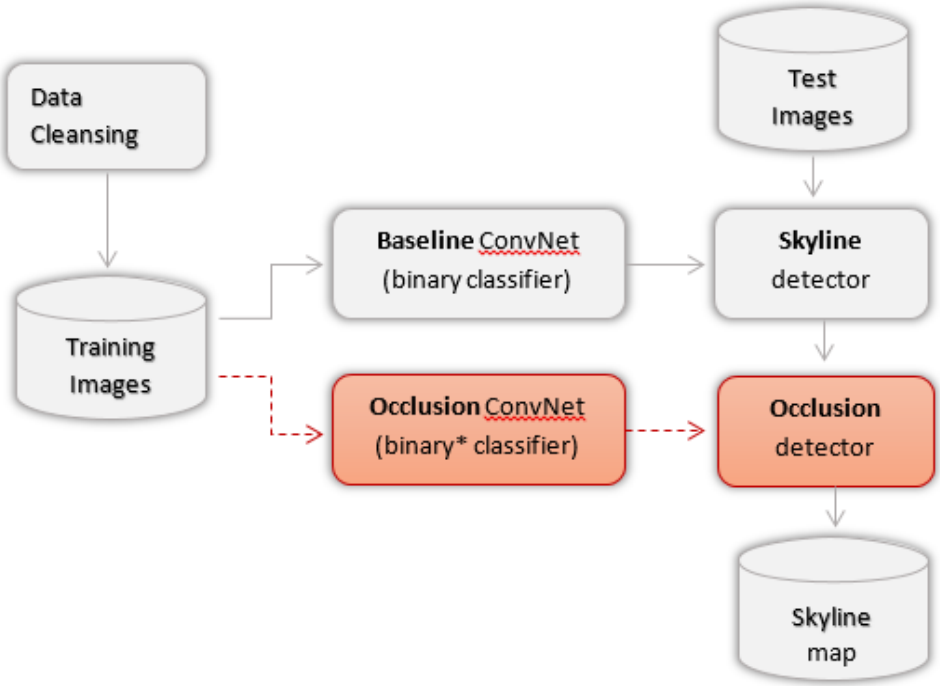


Figure 4.9: Combined Model. New work-flow of the learning process including the proposed Combined Model. The ConvNet specifically tailored for automated occluded skyline detection is highlighted in red.

As depicted in Figure 4.9, both ConvNets are used in a cascade hybridization fashion that can be analyzed as 2 sequential stages.

- In the **first** stage, a tailored mountainous skyline detection system -*Baseline ConvNet*- is used to produced a coarse binary horizon matrix identifying the skyline pixels in the form of a heat-map [50]. This ConvNet has been trained with square patches of 29x29 pixels, possibly containing a skyline in the center pixel -*for positive samples*-; since test images fed into the network have a height of 240 pixels, and a proportionally variable width, such heat-map is not longer the same size as the testing image, thus an additional post-processing step needs to be carried out.

- In the **second** stage, a tailored occlusion detector system -*Occlusion ConvNet*- produces a horizon vector that represent the columns that

contains a skyline pixel, but also those with occlusions, this occlusion detector is embedded on top of the previous results in order to clean all the miss-classified pixels. i.e to filter out all occlusions present in the image. It is worth mentioning that although the *Occlusion ConvNet* has also been trained to recognize skylines, we do not use it to do so for 2 main reasons, *first*, the *Occlusion ConvNet* is not able to predict the position of the skyline, only whether there is a skyline or not, *second*, the skyline accuracy in this ConvNet does not have a significant improvement over the *Baseline ConvNet* [50].

The resulting combined system is able to perform classification on images captured in a real outdoor scenery with omnidirectional augmented reality applications using only image sequences retrieved by the devices' camera.

Chapter 5

Evaluation

In the following, we present a detailed analysis of the different ConvNets implemented based on the notion of occlusion detection in skyline scenes that we introduced in Chapter 4. In a series of experiments we report both results according to generalized metrics in unconstrained instances, as well as a closer look at specific occlusion cases.

5.1 Dataset Collection and Preprocessing

The final goal of an augmented reality system is to enhance user experience through the accurate placement of virtual objects in virtue of smoothly generated mixed scenes. In the particular case of an outdoor application such PeakLens [10], this means being able to recognize the horizon line as well as segments of interrupted skyline covered by clouds, bushes, buildings, people, etc. so to be able to correctly align the skyline *wrt* the terrain and to overlay pertinent information tags about those peaks only on top of pixels where a skyline actually exists, making the *augmented content* precise.

Such compelling task required using a dataset that extends basic segmented labeling of mountains and sky to a finer labeled dataset suited for training our ConvNet. Although various real-world image datasets, broadly used in machine vision, are publicly available [23] [33] [38] [46] [80], to the best of our knowledge, no pixel-wise annotated dataset of diverse non-continuous skyline images compatible to [50] exists to quantify the generalization of the trained networks.

To evaluate the performance of the proposed approach, we have experimented on a large dataset [50] containing mountainous images randomly

collected from over 2,000 publicly available touristic web-cams and fetched from *Flickr*. This dataset is suited for our purposes since it is enriched with pictures showing various viewpoints, geographical and seasonal variations that add up to the occlusion patterns we require. The dataset comprises a total of **8,940** mountainous images manually annotated by crowd-sourcing means, deriving a ground-truth in a skyline binary format.

Despite the dataset being a rich source of interesting cases, a further step was required to overcome the imbalanced nature of the class distribution in the pictures, where as expected, skyline pixels are more prominent than non-skyline pixels. To this end, we have extracted hundreds of rectangular tiles from every image in the dataset, supposedly holding sufficient contextual information along their pixels; such technique bears a close resemblance to the one proposed by [36], [98] and [50]. To our purposes, we have used each image annotation mask to determine the presence of skyline pixels referenced as *positive columns*, and terrain, sky and occlusion pixels referenced as *negative columns*; later we used these labeled column-patches for training our ConvNet as outlined in Section 4.2 and evaluate the ability of our system to detect the presence of occlusions, more details on this will be given in Section 5.3.

5.1.1 Data Cleansing

Prior to train our ConvNet we identified the need for data cleansing. Detecting and removing errors and inconsistencies from the dataset in order to improve its quality became an essential step given that most of the images included in the dataset are collected from the web, furthermore, the dataset we worked on is a consolidation of different sources which increased significantly the need for cleansing. In order to feed accurate and consistent data to our ConvNet, we performed a series of steps which we detail as follows:

- A manual inspection of the data samples and their respective annotation was carried out with aims to correct particular instance problems. Although this approach is often perceived as naive, several instances were purged due to major noisy annotations, resulting in a new total of **8,913** images.

- In order to fulfill the column dimension requirement for all sample column-patches used during training and validation, as outlined in Section 4.2.2, images smaller in height than 240 px were discarded since they were not suited for the chosen network architecture. In total,

57 images were withdrawn from the dataset giving a final amount of **8,856** images and their respective annotations.

5.1.2 Data Preprocessing

Building an effective neural network model requires both careful consideration of the network architecture as well as the input data format. In this section, we discuss the latter.

There are a number of preprocessing steps we might wish to carry out before using our dataset as input to the chosen network architecture. However, for the sake of this work, we perform only one.

Uniform aspect ratio

One of the first steps we took in our experiments was to ensure images having the same size and aspect ratio as their annotations. To guarantee that subsequent evaluations will be carried out on coherent image-annotation pairs, we further analyzed the dataset to confirm full consistency between the sample and its ground-truth mask at file level, such analysis highlighted some discrepancies in the dimension of the images and their correspondent annotation. A total of **2,556** images exhibited different dimensions with respect to their annotations, the reason for this rather contradictory outcome, although not confirmed, could yield in the nature of the data acquisition step, since annotations were crowd-sourced through an online application, that might have lead to issues during conversion and storage tasks. To solve this conflict and ensure that all images in the dataset were consistent and uniform with regard to the ground-truth, a resizing step was performed in all uneven pairs; so as to not alter the annotation quality, images were resized in respect of their annotation width and height, meaning a conflictual image was scaled up or down to the dimension ratio of its correspondent annotation.

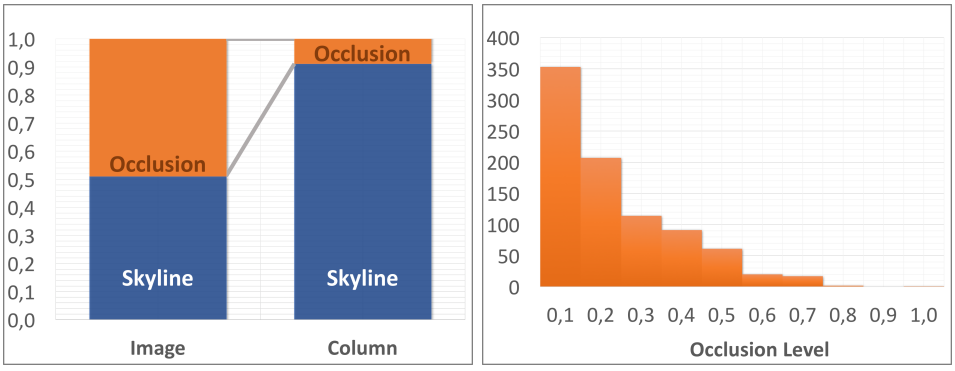
5.1.3 Occlusion Statistics

The dataset [50] is a fair source of challenging occlusion cases, as shown in Table 5.1, it contains thousands of images of which almost half contain occluded skylines. Nonetheless, at a closer look we can observe that less than 10% of the columns present in the entire dataset are occlusion objects. Since our main aim was, as mentioned earlier, to accurately detect occluded skylines, it is plausible then, that a number of limitations arise due to the nature of the chosen dataset.

	#	%
Occluded Images	4 327	48.86%
Non Occluded Images	4 529	51.14%
Occluded Columns	485 920	8.80%
Non Occluded Columns	5 038 444	91.20%

Table 5.1: Dataset Class Distribution. Half the images present non-continuous skyline. Interestingly, non continuity seems negligible, less than 10% of columns are occlusions.

As can be seen in Figure 5.1a, a very low occlusion/skyline ratio at column level prevails in our dataset. In like manner, it is apparent from Figure 5.1b that most images are occluded to a very small degree, showing a mode around 10% occlusion, this trait will be explored in detail in Section 5.3.3, where we will unfold the characteristics that lead to a good occlusion detector.



(a) Occlusion distribution (b) Occlusion histogram

Figure 5.1: Dataset Class Distribution. (a) Many images are occluded, conversely, few occluder segments are present (b) Occlusions form patterns: the distribution over relative orientations of occluder-skyline is highly peaked around one mode.

5.2 Experimental Setup

In order to evaluate the effectiveness of our system, we first compare the performance of the proposed ConvNet in isolation and then proceed to assess the quality of the combined model described in Section 4.4. In addition, to provide more insights, the implementation of the various heuristics proposed in this work are evaluated to investigate the effects of different sampling techniques on the *skyline-occlusion prediction* performance.

5.2.1 Protocol

For each experiment designed (refer to Section 4.2.1), we performed a *hold-out* segmentation; the dataset D was split into the training set D_{train}, the validation set D_{val}, and the test set D_{test}. We then trained our ConvNet with a `80-20` setting, where 80% of the images were used for training and validation (64% and 16% respectively) and the remaining 20% for testing.

We have successfully trained our networks and evaluate them on the validation set using various sampling approaches addressed in Section 4.2.1, afterwards, we tested the results on the level of detections at both, column-patch and image level. All experiments were carried out on the ConvNet architecture and optimization settings depicted in Section 4.3. Table 5.2 further details on the segmentation of the dataset.

		D_{train} **(64%=5 668)**	D_{val} **(16%=1 417)**	D_{test} **(20%=1 771)**
Experiment 1	Positive	1 272 797	317 476	395 853
	Negative	1 272 797	317 476	395 853
Experiment 2	Positive	1 278 852	318 972	398 316
	N-Terrain	426 284	106 324	132 772
	N-Occlusion	426 284	106 324	132 772
	N-Sky	426 284	106 324	132 772
Experiment 3	Positive	717 952	179 073	223 617
	N-Terrain	717 952	179 073	223 617
	N-Occlusion	717 952	179 073	223 617
	N-Sky	717 952	179 073	223 617

Table 5.2: Dataset Segmentation. D_{train} to fit the model. D_{val} to estimate prediction error for model selection and tune hyper-parameters. D_{test} to assess the generalization error of the final chosen model.

5.2.2 Evaluation Metrics

We adopted widely-used metrics from common classification evaluations and report their results on the 3-experiment configuration depicted in Section 4.2.1. The metrics described below were used during the evaluation at column-patch level.

Accuracy (A); averages all corrected classified test cases:

$$A = \frac{TP + TN}{|D_{test}|}$$

Precision (P); measures positive predictions against the number of positive class values predicted. Can be thought of as a measure of *exactness*:

$$P = \frac{TP}{TP + FP}$$

Recall (R); also called *Sensitivity*, measures positive predictions against the number of positive class values in the test data. Can be thought of as a measure of *completeness*:

$$R = \frac{TP}{TP + FN}$$

F1 Measure (F); conveys the balance between precision and recall.

$$F = 2 \times \left(\frac{P \times R}{P + R}\right)$$

True Negative Rate (TNR); also called *Specificity*, measures positive predictions against the number of positive class values in the test data. Can be thought of as a measure of *completeness*:

$$TNR = \frac{TN}{TN + FP}$$

However, as stated by [50], accuracy metrics at column-patch level may fail to represent the quality of our model on whole full images. Thus, a series of tailored metrics for skyline detection evaluation that assess quality at image level are required; the metrics proposed by [50] are suited for this purpose since their estimations are calculated by contrast, meaning we will compare the extracted skyline using our ConvNet against the ground-truth. We proceed to explain in detail how these metrics work.

Let `CNN(i, j)` be a function that returns `1` if the image pixel at coordinates `(i,j)` belongs to the skyline extracted by the ConvNet (`0` otherwise) and let `GT(i, j)` be a function that returns `1` if the pixel `(i,j)` belongs to the ground-truth skyline (`0` otherwise).

Average Skyline Accuracy (ASA); measures the fraction of image columns that contains ground-truth skyline pixels and in which at least one of the positive pixels extracted by the ConvNet matches one of the ground truth pixels.

$$ASA = \frac{\sum_{j=1}^{cols} I_{GT \wedge CNN}(j)}{\sum_{j=1}^{cols} I_{GT}(j)}$$

Average No Skyline Accuracy (ANSA); measures the fraction of columns that do not contain any ground-truth skyline pixel and for which also the ConvNet output does not contain positive pixels; this metric evaluates false positives in images with an interrupted skyline.

$$ANSA = \frac{\sum_{j=1}^{cols} I_{\overline{GT} \wedge \overline{CNN}}(i,j)}{cols - \sum_{j=1}^{cols} I_{GT}(j)}$$

Average Accuracy (AA); measures the fraction of columns in which the ground-truth and the ConvNet skyline coincides, considering agreement when none contain pixels or otherwise at least one of the ConvNet pixels matches one of the ground truth pixels.

$$AA = \frac{1}{cols} \sum_{j=1}^{cols} I_{agree}(j)$$

given that: $I_{GT}(j) = 1 \; if \; \exists i | \; GT(i,j) = 1; \; 0 \; otherwise$
$I_{GT \wedge CNN}(j) = 1 \; if \; \exists i | \; GT(i,j) = 1 \; \wedge \; CNN(i,j) = 1; \; 0 \; otherwise$
$I_{\overline{GT} \wedge \overline{CNN}}(j) = 1 \; if \; \forall i | \; GT(i,j) = 0 \; \wedge \; CNN(i,j) = 0; \; 0 \; otherwise$
$I_{agree}(j) = 1 \; if \; I_{GT \wedge CNN}(j) = 1 \; \vee \; I_{\overline{GT} \wedge \overline{CNN}}(j) = 1; \; 0 \; otherwise$

Although various sampling techniques were used to overcome the imbalanced distribution between the different classes in hopes to minimize bias towards the more frequent class -*positive columns*- as explained in Section 4.2, *negative columns* are still under-represented. We recognize then, pure accuracy-based metrics albeit representative, may not be the most appropriate in measuring our model success since they may falsely suggest above-chance generalizability. When these metrics are applied indistinctly on a test set that is imbalanced in the same direction, we could possibly yield unreal pessimistic estimates; under such conditions we reckon the need to weight the chosen accuracy metrics, in order to do so, we adopt the *balanced*

accuracy metric proposed by [34]. The metrics described below were used during the evaluation of the *combined model* at image level.

Balanced Accuracy (BA); symmetric about the type of class, it measures the number of correct predictions divided by the number of predictions of each class:

$$BA = \frac{1}{2} \times \left(\frac{TP}{P} + \frac{TN}{N} \right)$$

It can be extended to include a cost `c` associated with the misclassification of one particular class by dropping the symmetry:

$$BA = c \times \frac{TP}{P} + (1 - c) \times \left(\frac{TN}{N} \right), where\ c \in [0, 1]$$

5.2.3 Evaluated Baseline

As part of the evaluation protocol of this thesis, we undertake a comparison analysis of the conditions and further achievements of the ConvNet, presented in Section 4.3, with respect to a fully-working skyline detection model [50] already deployed as part of an augmented reality application designed for outdoor activities [10] which we consider as a **baseline** for the purpose of this study.

5.3 Experimental Results

In this section, we present the experimental results of the proposed approach. Firstly, we expose the performance achieved by our ConvNet at column-patch level, as explained in Section 4.3, our whole training was done effectively by sampling in column-patch-wise like manner, in order to correct class imbalance and mitigate the spatial correlation of dense column-patches. Secondly, we unfold the results of our experiments at image level in both occlusion and unconstrained cases.

5.3.1 Column Patch-wise Evaluation

We commence by evaluating the ability of our ConvNet to correctly classify patterns as occlusion or skyline labels in isolation. To that end, we fully trained our ConvNet and reviewed its performance in the test set. This evaluation is performed contrasting the prediction outputted by the ConvNet and the pixels located in the middle column of a sampled rectangular

column-patch. For the purpose of this assessment, no further post-processing on images was required.

In Table 5.3 we report the results obtained on the *first experiment* of the dataset (Experiment 1), in which column-patches were sampled as a {positive : negative} ratio of 1 : 1. As can be seen in the table, the model yields considerable performance and it is capable of recognizing occlusions in almost 91% of the cases, we can also observe that the model achieves good balance between its capability of detecting occlusions and skylines, so it is sensitive as it is specific. It is worth noticing that this particular *experiment* comprises an implicit binary classification in which both positive and negative entries were sampled randomly, furthermore, given that the distribution of negative samples is imbalanced towards occlusion patches, the ConvNet presumably could learn to discriminate from the features of more prominent negative classes such as *negative-terrain* or *negative-sky*, and it would not give much importance to the fact that misclassified the data of the scanty class, *negative-occlusion*; consequently, we presume that most of the misclassification errors in the negative class, almost 9%, arise on instances labeled as *occlusions*. The values reported in Table 5.3 were obtained using a cutoff value for positivity of *109* from a [0-255] range.

As shown in Table 5.3, results obtained on the *second experiment* (Experiment 2), in which column-patches were sampled as a {positive : negative-sky : negative-occlusion : negative-terrain} ratio of $1 : \frac{1}{3} : \frac{1}{3} : \frac{1}{3}$ reveal interesting insights; it is worth noticing that, although sampling discriminates among 3 different negative classes, the reported metrics were computed on the basis of a binary classification. We observe that the induced balanced distribution among **negative classes only**, does not provide a finer distinction of the features that represent an *occlusion*, moreover, it harms the accuracy level in more than 4% with respect to the *Experiment 1*; before interpreting these results, we remind the reader of our main aim which is detecting occlusions in a skyline, for that purpose, we are more interested in assessing the capability of the model to detect negative samples, although outperformed by the others, this *experiment* reaches an acceptable level of performance, achieving almost 85% of negative cases correctly classified. However, the model exhibits a tendency to be biased towards the positive class, annotating skylines more often than it should. The values reported in Table 5.3 were obtained using a cutoff value for positivity of *132* from a [0-255] range.

Finally, Table 5.3 portrays the results obtained from the third *experi-*

ment (experiment 3), in which column-patches were sampled as a {positive : negative-sky : negative-occlusion : negative-terrain} ratio of 1 : 1 : 1 : 1, as with the *second experiment*, here, again, the reported metrics were computed on the basis of a binary classification. We observe that regarding non-skyline accuracy (TNR), this experiment reaches a better performance than the others, correctly predicting occlusions in 93% of the cases, suggesting that the induced balanced distribution among **all classes**, does help the network to better understand features representing occlusions. Subsequently, at the expense of this gain, levels of Precision and Recall fall. The values reported in Table 5.3 were obtained using a cutoff value for positivity of *143* from a [0-255] range.

	Accuracy	Precision	Recall	F	TNR
Experiment 1	**0.9258**	**0.9140**	**0.9417**	**0.9396**	0.9096
Experiment 2	<u>0.8800</u>	0.8584	0.9101	0.8835	<u>0.8499</u>
Experiment 3	0.8906	<u>0.7909</u>	<u>0.7645</u>	<u>0.7775</u>	**0.9326**

Table 5.3: Accuracy Metrics at Column-Patch Level. Best results are shown in **bold**, lowest estimates are <u>underlined</u>. Comparison against baseline is not meaningful, thus, not performed.

Let us explore these results with more detailed by looking at the confusion matrix derived in each experiment. As we can observe in Figure 5.2a, *Experiment 1* shows around 7% misclassifications of true skylines, furthermore, occlusions are mislabeled as skyline in 10% of the cases. Had we employed this ConvNet to detect both skyline and occlusions, we consider it as the best one, as it shows a good balance between TP and TN rates, however this is not the case. In like manner, less than 10% skyline column-patches have been misclassified as occlusions and in 18% of the cases the model in *Experiment 2* has failed to correctly distinguished occlusions, as shown in Figure 5.2b. Interestingly, we observe in Figure 5.2c that *Experiment 3* erroneously classified around 30% of the skyline column-patches as occlusions, however, roughly 7% of true occlusions were mislabeled as skyline, showing that this model is the best at recognizing occlusion patterns. Although the last experiment shows a considerable larger rate of skyline mispredictions compared to the other 2, this outcome should be assessed having in mind that the ConvNet we introduced in Section 4.3 is designed to better detect occlusions as it will later be concatenated to another ConvNet that detects skyline quite accurately, as such, the 3% improvement in no-skyline accuracy gained using *Experiment 3* is relevant for our purposes,

whereas the loss in skyline accuracy is not representative.

		Observed Class	
		Positive	Negative
Predicted Class	Positive	380 241	35 770
	Negative	23 547	360 083

(a) Experiment 1

		Observed Class	
		Positive	Negative
Predicted Class	Positive	362 512	59 805
	Negative	35 804	338 511

(b) Experiment 2

		Observed Class	
		Positive	Negative
Predicted Class	Positive	170 964	45 209
	Negative	52 652	625 639

(c) Experiment 3

Figure 5.2: Confusion Matrices. Derived from the conducted experiments. (a) Experiment 1 results (b) Experiment 2 results (c) Experiment 3 results.

Taken together, these results point to the likelihood that, based solely on the sampling technique, *Experiment 3* is the most promising setting and should be the preferred configuration when training our ConvNet, since it is the one that reaches the best TNR. Interestingly, *Experiment 1* does also present comparable results, showing good balance among TNR, Precision and Recall. These findings, nevertheless, need to be interpreted with caution since so far, we have tested our ConvNet with column-patches only. In the following section we will further investigate if this assumption can be extended to a more realistic scenario where whole images are assessed. Please refer to Appendix B for a detailed visualization of the chosen metrics.

5.3.2 Unconstrained Detection

In this section we conduct the evaluation of our various experiments using a more realistic approach, in which we no longer test prediction in patches but assess the quality of the proposed model in unconstrained whole images containing both continuous and non-continuous skyline. For this purpose, the complete set D_{test} was tested. As outlined in Section 4.4, a *combined model* -composed of the **baseline** approach and our **ConvNet** in a cascade hybridization fashion- was designed to overcome limitations of earlier works [50], given that our main goal is to tackle the *occlusion problem*, we further explore this new enhanced model and treat it as the subject of all following evaluations.

It can be seen in Figure 5.3 that, while the baseline approach is comparable in terms of *ASA* and *AA*, the *combined model* achieves the best performance in detecting non skyline patterns *ANSA*, around 40% in all experiments, improving over the baseline detector by a significant margin of 14.14% with *Experiment 1*, 13.22% with *Experiment 2* and 14.50% with *Experiment 3*, such results have further strengthened our confidence, validating the usefulness of the undertaken approach. Detailed results are shown in Table 5.4.

Interestingly, our model (85.48%, 85.32% and 85.23%) outperforms the baseline (84.82%) at the *AA* level as well, with a slight increase of 0.66% on our best detector. While the latter improvement could be interpreted as modest at first glance, we point out that this result is significant as it translates to obtaining almost **4 000** more true positive and negative detections, which clearly suggests that an occlusion handling-oriented model can boost the overall detection accuracy significantly. Nonetheless, we also observe that a slight decrease in *ASA* is introduced, such result was anticipated and

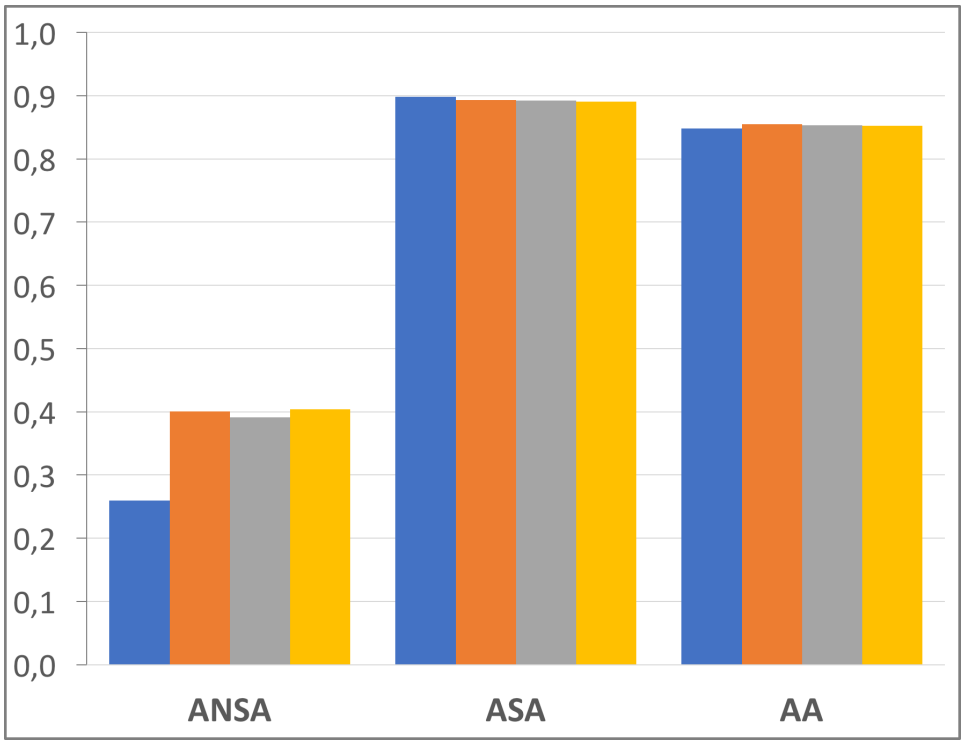


Figure 5.3: Performance Metrics in Unrestricted Images. *Baseline* results are shown in blue. Results of *Experiment 1* are shown in orange, results of *Experiment 2* are shown in silver. Finally, results of *Experiment 3* are shown in yellow.

lies on the cascade nature of the combined model we proposed, given that our detector works on top of the baseline predictions, refining the pixels previously classified as *skyline* by validating such classification against our ConvNet, that is tailored to recognize occlusions.

It should be noticed that although the raise in the number of occlusions correctly predicted might seem a windfall, it comes at a price, around 0.75% loss in the skyline detection task. Nonetheless, this result was anticipated and can be explained in part by the significant relationship between the skyline-occlusion discriminative features our ConvNet was entrusted to learn, furthermore, although the modeled classes had the same weight in the loss function during training, the sampling techniques described in Section 4.2.1 further encouraged our ConvNet to learn less frequently occurring classes, such is the case of occlusions, with weights getting a higher value in the loss function, a drop in the average skyline prediction is to be expected. Please refer to Appendix B for a detailed visualization of the chosen metrics.

Let us now look an alternative evaluation, in which we take into account

	ANSA	ASA	AA	BA	BA *c=0.6*	BA *c=0.2*
Baseline	0.2593	**0.8980**	0.8482	0.7520	0.7899	0.6382
Experiment 1	0.4007	0.8929	**0.8548**	0.8007	0.8384	0.6875
Experiment 2	0.3915	0.8927	0.8532	0.7957	0.8350	0.6777
Experiment 3	**0.4043**	0.8905	0.8523	0.8180	0.8525	0.7142

Table 5.4: Performance Metrics in Unrestricted Images. Best results are shown in **bold**, lowest estimates are underlined. Considerable increase in the detection of occlusions achieved by all 3 different experiments.

the imbalanced nature of our dataset. To this end, we use the *Balanced Accuracy* metric discussed in Section 5.2.2. BA allows us to weight the misclassification errors by providing generic safeguards against reporting optimistic or pessimistic estimates. If our classifier performs equally well on either class (skyline and occlusion), this term reduces to the conventional accuracy. In contrast, if our classifier is taking advantage of the imbalanced nature of the test set, BA will drop to chance [34].

Table 5.4 depicts further evidence that supports the hypothesis we planted in this study. All 3 variants of our model are superior than the baseline when accounting for different numbers of representatives from each class. Furthermore, when dropping the symmetry of the proposed metric and introducing a mild penalization of **0.4** *(when c=0.6)* and a severe penalization of **0.8** *(when c=0.2)* associated with the misclassification of a skyline, we observe that the generalizability of the *combined model* is still significant with more than 85% accuracy in our best predictor when a non-severe penalization is introduced; furthermore, when more importance is given to the correct classification of occlusions *(when c=0.2)*, our predictors achieve considerable results, surpassing the baseline in more than 5%. Results reported on Table 5.4 confirm that our ConvNet *-at a non-negligible degree-* has learnt to discriminate among samples according to their features rather than be biased towards the more frequent class.

5.3.3 Detecting Occlusion Patterns

We now proceed to evaluate the ability of our model, to reliably identify occlusions in the skyline, under a constrained subset of images where only images containing occluded skylines are evaluated, we call this new set $\mathrm{D}_{\mathrm{occ}}$. This new set of images is built upon the test set ($D_{\mathrm{occ}} \subset D_{\mathrm{test}}$).

In this evaluation, we considered a series of increasingly difficult scenarios for comparing the *combined model* performance, corresponding to increasing levels of occlusions. Expressly, we look at the following 4 scenarios: the full subset D_{occ}, D_{occ} restricted to at most 10% occluded columns, D_{occ} restricted to occluded columns between 10% and 30%, and D_{occ} with more than 30% occluded columns. Table 5.5 reports the results of the evaluation, in particular, we contrast the performance of the 3 variants we have experimented on (Experiment 1, Experiment 2 and Experiment 3) with one baseline, the standard model proposed by [50], which is partially aware of occlusions as we will see as follows. We consider this targeted evaluation essential to our experiments, on account of the fact that this will draw meaningful conclusions about the role of different variants of occluded skylines inside captured images.

		ANSA	ASA	AA
Baseline	D_{occ}	0.2593	**0.8772**	0.7771
	0-10%	0.3127	**0.8930**	**0.8776**
	10-30%	0.2978	**0.8753**	0.7790
	30%+	0.2200	**0.8458**	0.5892
Experiment 1	D_{occ}	**0.4679**	0.8601	**0.7973**
	0-10%	**0.4902**	0.8739	0.8635
	10-30%	0.5144	0.8599	**0.8021**
	30%+	**0.4330**	0.8316	**0.6689**
Experiment 2	D_{occ}	0.4402	0.8612	0.7915
	0-10%	0.4808	0.8763	0.8655
	10-30%	0.5007	0.8575	0.7975
	30%+	0.3825	0.8310	0.6463
Experiment 3	D_{occ}	0.4667	0.8539	0.7903
	0-10%	0.4852	0.8706	0.8601
	10-30%	**0.5254**	0.8511	0.7967
	30%+	0.4147	0.8209	0.6525

Table 5.5: Performance Metrics in Occluded Images. Best results are shown in **bold**, lowest estimates are underlined. For each evaluation stage, 4 settings were tested, (a) full occlusion test set (b) test set with: images containing up to *10%* of occluded skyline (c) images containing up to *30%* of occluded skyline (d) images containing more than *30%* of occluded skyline.

We observe that the trends from the evaluation in unconstrained images transfer to the more specific subset of occluded skylines. We then, make the following observations: firstly, we observe that our 3 detectors achieve

a relatively higher *ANSA* with respect to the baseline, the best one reports an increase in more than 20% of accuracy when images are fairly covered with occlusions (with a mode around 30%) reaching a remarkable 52% of accuracy on the non-skyline detection task. Interestingly, our model does not seem to benefit from higher levels of occlusion inside an image, conversely, accuracy drops to 38% when images are largely occluded. Although the reasons for this result are not yet entirely understood, we believe the foremost cause of this discrepancy could be explained by patterns present in these largely covered skylines that were not included in training samples, or if they were, it was not sufficiently represented. Secondly, we observe our detectors performing on a comparable level to the baseline *ASA*, 89.30% vs 87.63% in the best scenario when images are occluded up to 10%, and around 2% less in the remaining scenarios, proving that to better understand occlusions we might have to risk our understanding of skylines. The overall accuracy *AA* however, is boosted with the newly introduced rise in ANSA, nearly 8% of gain in the best predictor. Interestingly, these tests revealed that the baseline fails to accurately detect occlusions when these are largely present in an image, classifying only 22% of the occlusion cases correctly.

5.3.4 Efficiency Evaluation

Efficiency is defined as the degree in which software fulfills its purpose without wasting resources [88]. One measure of efficiency is the execution speed of the various modules that compose the assessed software, such trait becomes of high importance in mobile oriented settings, hence, we must guarantee that the proposed models are not only reliable (by achieving high recognition accuracy) but efficient as well.

Executing the proposed model in desktop PCs shows negligible execution time per image, however, such results are not meaningful since we are more interested in the suitability of our model on mobile devices. To this end, we assessed the execution time per image in smartphones with different hardware specification. Table 5.6 compares the efficiency performance of the proposed combined model across several devices.

For this reason, we have selected an input image of 320 x 240 pixels. Whereas the majority of smartphones in the market support capturing frames of larger size, after several experimental trials we observed that the chosen dimension had the best balance of accuracy, memory consumption (**13.44** MB on average), and execution time, on a broad spectrum of devices. To conduct this evaluation the skyline extraction process was repeated 1 000

Device	Time (ms) Baseline	Combined Model
MacBook Pro		
2,9 GHz Intel Core i5 (2 cores)	73	90
16GB		
Google Pixel		
2,15 GHz Qualcomm Snapdragon 821 (4 cores)	199	363
4GB		
Nexus 6		
2,65 GHz Qualcomm Snapdragon 805 (4 cores)	273	477
3GB		
One Plus A0001		
2,46 GHz Qualcomm Snapdragon 801 (4 cores)	296	501
3GB		
Nexus 5X		
1,82 GHz Qualcomm Snapdragon 808 (6 cores)	437	686
2GB		

Table 5.6: Efficiency Performance on Mobile Devices. Time required to execute the skyline extraction component in different low-powered devices.

times on the selected image, in each device tested. Thence, execution times were averaged and reported in Table 5.6. We make the following observations: first, naturally, the execution time in low power mobile devices is much higher than PC's, where skyline extraction can be performed at a frequency of **10** images per second, conversely, smartphones could barely achieve **2** images per second. These results thus need to be interpreted with caution, first and foremost, we need to acknowledge that real-time AR applications demand special usability requirements that do not compare to PC's large capabilities, and secondly, good execution times are consistent among all mid-range smartphones. On account of the fact that rates shown in Table 5.6 are compatible with AR mobile applications' constraints, we consider these results significant, however, a noticeable increase with respect to the baseline's reported results [50] is evident, on average around 71% more time is needed per image. We associate this increase to the sequential nature of the proposed *combined model*, although we could decrease the times by parallelizing the 2 components composing the model, it may not be worth to put an extra computational effort to the device for what we estimate will be a mild gain, nonetheless, further investigations in this direction need to be carried out, in order to explore different ensemble models.

Devices running PeakLens handle the entire image processing at background; which means that when no sudden camera movements occur, as expected in a mountain peak recognition application, the skyline extraction and the subsequent DEM alignment step could possibly be done at a frequency lower than 15 frames per second, usually considered viable for video play. Such trait comes at a price in terms of side effects, some jitter in the camera view when the extraction and alignment is performed, and whenever an update of the peak positions inside the camera view is required.

5.4 Discussion on the Performance

5.4.1 On the Accuracy

We commence by confirming the ability of our system to detect occlusion patterns in unconstrained mountainous images. Furthermore, our experiments have demonstrated that such statement is still valid when facing not only complicated but frequent scenarios typical of outdoor applications' usage. Interestingly, we have observed that when moving from the detection under unconstrained scenarios comprising both occluded and non-occluded skylines of varying difficulty (refer to Section 5.3.2) to an isolated setting (refer to Section 5.3.3), the undertaken approach provides additional support to the *occlusion handling* requirement, widening our knowledge of the uncertainties related to real-time AR applications. As reported in Table 5.4, our model exhibits comparable performance in terms of Average No Skyline Accuracy **ANSA** when compared to the original detection method, showing an increase of almost 15% w.r.t [50]. It is worth noticing that even when Experiment 2 presents lower accuracy than the other 2 experiments, it still boosts the baseline model performance, as expected.

The afore-mentioned findings confirm the usefulness of the proposed *combined model*. Our technique clearly has an advantage over the baseline's capability to accurately detect occlusions without compromising the true-skyline detection accuracy. Broadly speaking, these results are a direct outcome of the heuristics followed over the course of this study. Our method comprises an efficient binary classification model that is able to detect skylines and localize occluded segments out of the visible continuous horizon line.

Contrary to other research carried out in this area, in which the identification estimations seem to treat the scarce class representatives -occluders- as outliers and ignore image evidence in occluded sections, our ConvNet

leverages the appearance of occlusion boundaries throughout the skyline. To be precise, our network, as most classification models, do not yield a binary decision, but rather a continuous decision value, using the decision values outputted from our model we rank test samples, from *'almost certainly positive'* to *'almost certainly negative'*. Based on the decision value, we assign for each experiment a *probability cutoff* that configures the classifier in such a way that a certain fraction of data is labeled as `skyline` and the remaining as `occlusion`; in this way we have introduced a non-arbitrary **cost function** into our model to handle the implicit cost of false negatives to false positives (FN/FP); all sampling techniques applied in our experiments (refer to Section 4.2.1) are orthogonal to this technique. So as to determine an appropriate threshold we have used receiver operating characteristic (ROC) curves and chose a threshold based on what fits our specific needs. Please refer to Appendix B for further details on the probability cutoffs chose for each experiment.

Let us now explore different scenarios that further demonstrate our model's strengths, foremost, have let us identify possible opportunities for improvement. We evaluate the *occlusion problem* by means of a qualitative analysis, to this intent, we prepared a handful set of examples that will help us examine, visually, the results obtained.

Ordinary Detection

Figure 5.4 illustrates the detection quality comparisons of the 3 experiments we have conducted with respect to the baseline and ground-truth. Before interpreting our results, we remind the reader of our main aim, that is, detecting occlusions, focus will be given then, to the improvement achieved with our model on this basis.

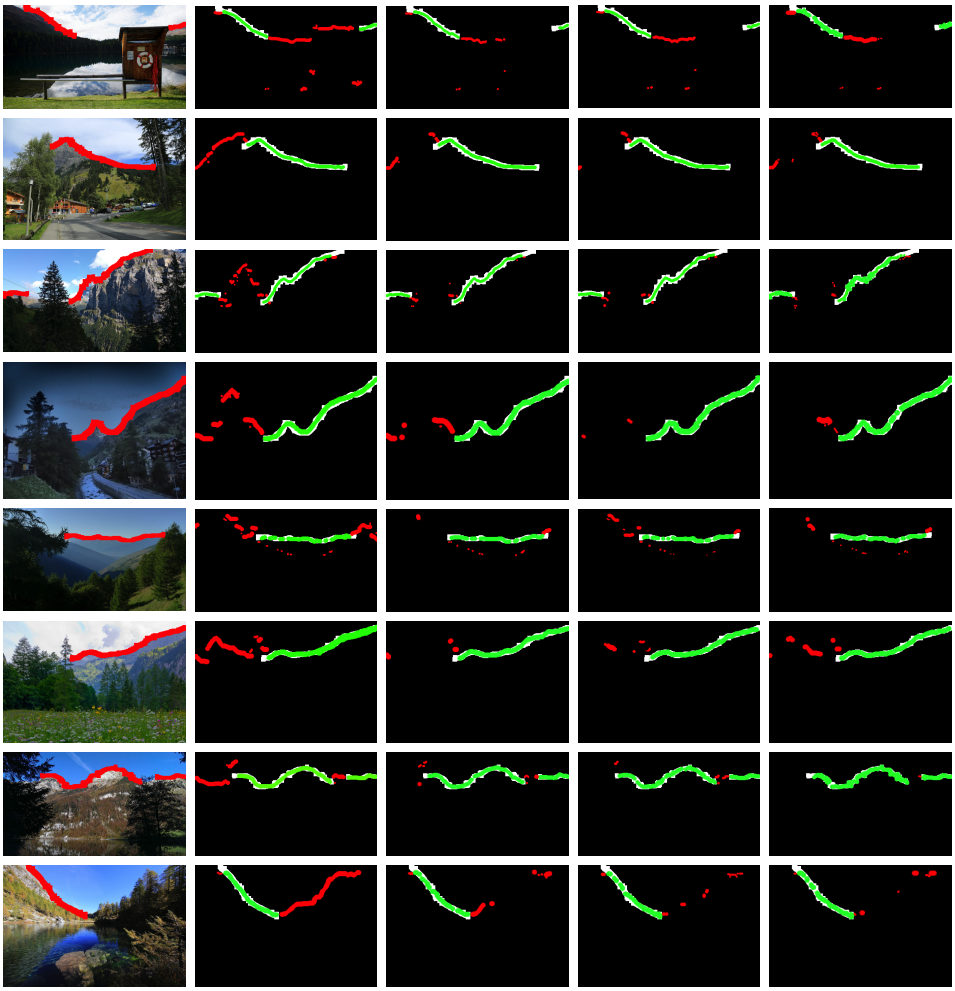
We derive from the output prediction masks, that by coupling our model to the baseline, we are able to correct its prior inferences in most cases. These corrections are not negligible and are worth to be highlighted, they vary from mild to highly noticeable, for instance, we observe that our model corrects almost 55% of the misclassification done by the baseline in the second scene; a more noticeable improvement is observed in the seventh scene, in which the first variant of our model amends almost 90% of the misclassifications.

A very interesting case of correction is depicted in the third scene, the baseline is able to recognize the true skyline almost perfectly, however, it fails to detect a tall pointy tree occluding the horizon and misclassifies it as *skyline*, our model though, corrects the misclassifications in more than 80%

by understanding that this tree is in fact, an occluder. Similar conclusions arise in the fourth, fifth, sixth and seventh scene; we suspect the baseline fails to discriminate well defined objects such as trees, with similar texture and color of the peaks they are occluding. By training our ConvNet with column-patches containing more context we are able to overcome this issue.

Observing the samples shown in Figure 5.4 we reckon that most of the images have the same structure, i.e., skylines are usually located on the upper half, and often they are occluded by objects of different nature, being trees the predominant occluder. There are only few samples in which the structure of the scene changes, such is the case of the first and last scene in Figure 5.4. In these cases, it appears that trees are not actually occluding the peaks, or the horizon they do occlude is not substantially visible and the region in which the skyline blends with the nature is a bit fuzzy. These cases are very tricky and apparently, our ConvNet struggles to distinguish these features, thus, scenes like these are of higher uncertainty for the network. We fear the reason behind this phenomenon is the small amount of training samples with this spatial arrangement, but most importantly, it is likely that the reason for this is that images displaying this pattern were *inconsistently* annotated -not surprising though-, if we carefully observe these images, we will come to the realization that their annotations are highly subjective, and depends entirely of what a human thinks of an occlusion.

Regardless we encounter visible flaws, inconsistencies or even improvement opportunities, we can easily observe that our experiments provide not only a generalized improvement with respect to the baseline in all cases, but pose a significantly powerful technique tailored to the mountain peak detection task. We fully associate the gain in both **ANSA** and **AA** to the learning process proposed throughout this study.



(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 3

Figure 5.4: Ordinary Detection. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

Occlusion Patterns

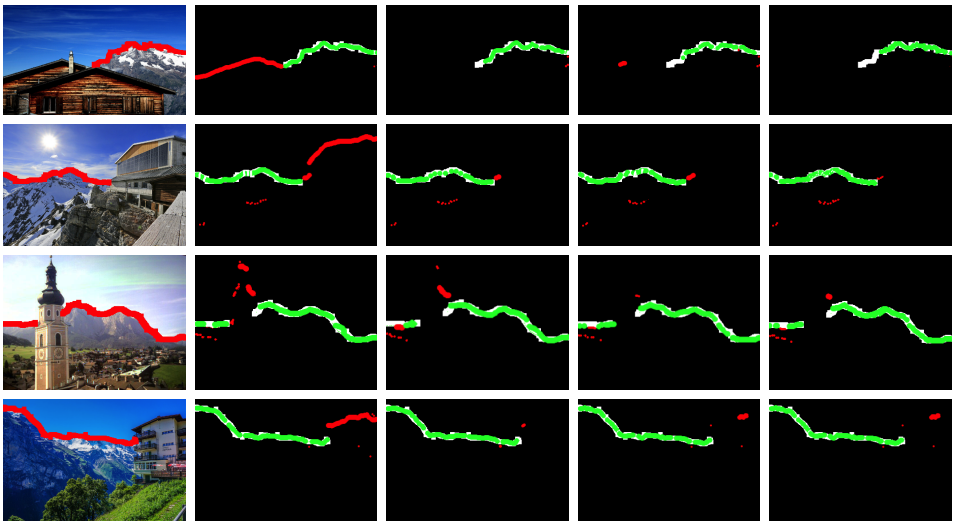
Throughout the course of this study we have recognized different highly frequent patterns in the scenes we used to train our networks and the overall dataset. We have observed that our model showcases significant results while inferring them. In such like manner, we encounter some other occlusion patterns that pose an extra challenge to our model, we will further discuss them in Section *Challenging Patterns*. Following, we showcase frequent patterns in which our model performs very well.

Frequent patterns found in the dataset, for which our model shows a clear advantage over the compared baseline, are (a) buildings, (b) trees and bushes, (c) snowy and (d) cloudy mountains. Most of the test scenes that fall in these categories yield satisfactory results, it can thus be asserted that the *combined model* presented in this thesis, will be well for use by a real application, in which these patterns are highly frequent, which is the case of an outdoor AR application such PeakLens.

Less predominant are people and power cable towers, albeit present, their distribution is not sufficiently representative. We are of the opinion that our model could be better exploited by the inclusion of more images portraying patterns such these, that represent a *real* world distribution of classes, especially *people*, given that it is very likely to encounter scenes where people are occluding the landscape when interacting with an augmented reality application. As such, we are aware that our work may have been limited by the frequency of these desired patterns which could have prevented us from a good generalization performance. However, we have observed promising results when assessing the capability of our model to detect people as occluders, as well as power cable towers.

Buildings

Detecting high frequent occlusion patterns in images is feasible, such is the case of buildings overlapping the skyline. Achieving both, sufficiently high accuracy while detecting skyline (already provided by the baseline) and comparable non-skyline accuracy (provided by our ConvNet). We account these results viable evidence that the combined detector has the potential to aid recognition when occlusions are present as can be seen in Figure 5.5, all cases shown a remarkable advantage introduced by the use of our model, which is capable of recognizing almost every non-skyline pixel accurately. We reckon *buildings* to be our model's most accurate prediction.



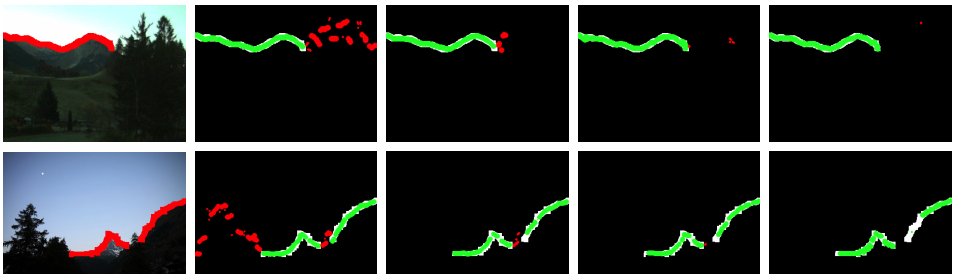
(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 3

Figure 5.5: Occlusion Patterns: Buildings. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

Trees

Out of all type of occluders, trees are the most common, yet the most variable objects we can encounter in an outdoor scene. Although a general structure of a tree can be inferred by our model, their irregular edges and textures make them a very interesting case of study. Contrary to buildings for example, trees do not have hard edges and do not introduce high contrast shifts in the scene, they blend smoothly with mountains and peaks, specially in non-harsh seasonal pictures.

Not surprisingly, our dataset contains thousands of images with various patterns in which we find trees occluding the skyline the most frequent of all, this fact positively contributed the learning process of our ConvNet, making it able to learn specific features of this pattern. Figure 5.6, pinpoints examples of the inference level of the proposed model when facing trees as occluders. Following the success achieved using the proposed model in scenes belonging to the *Building* pattern, we can observe our model is able to correctly detect occluding trees, amending almost all misclassifications done by the baseline.



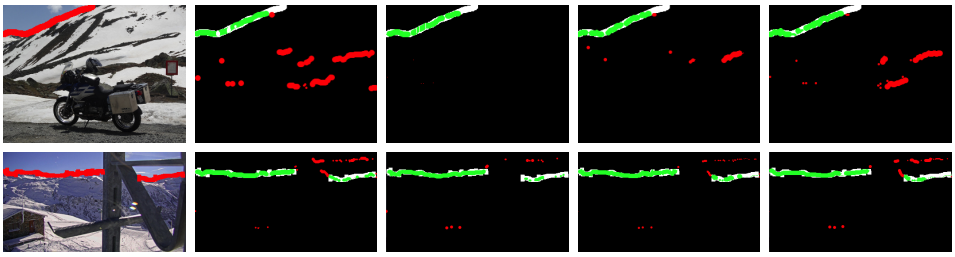
(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 3

Figure 5.6: Occlusion Patterns: Trees. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

Snowy Mountains

Despite not being as frequent as buildings and trees, our model seems to perform good in the task of recognizing winter-*ish* pictures. Peaks and mountains covered with snow, proved to be a challenging case for the baseline as we can observe in Figure 5.7. The soft lines emerging from the ground could resemble a horizon line under a cloudy sky, interfering with the baseline's knowledge of what defines a mountain peak.

Although it may be perceived as not remarkable as previous patterns, we consider these results significant as they show that the proposed model has learned to discriminate the soft edges recurrent in this pattern, and specially, sufficiently better than the baseline.

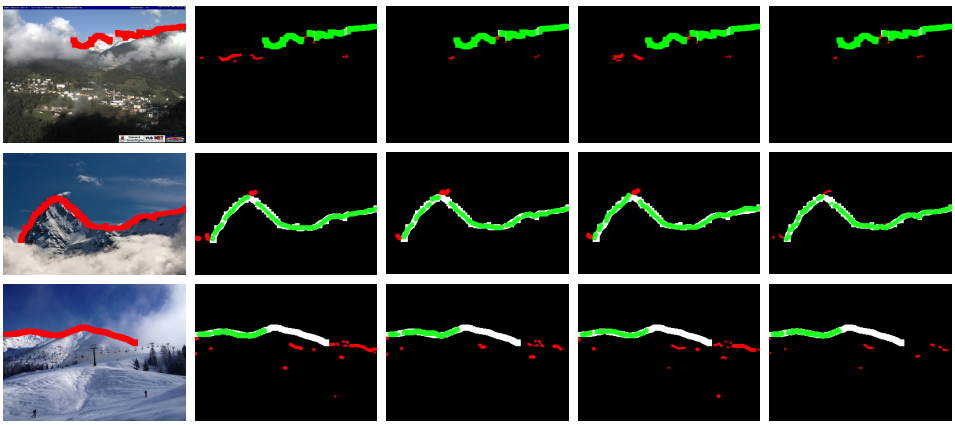


(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 2

Figure 5.7: Occlusion Patterns: Snowy Mountains. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

Cloudy Mountains

Another frequent pattern is that of skylines being occluded by clouds. We have observed this pattern to be quite challenging depending on the composition of colors with respect to the mountain peaks and the *blurriness* of such clouds. The soft lines blending with the peaks may lead to some mispredictions, however, we confirm our model working better than the baseline at recognizing these kind of occlusions as can be seen in Figure 5.8. There are more complex cases of blurry clouds covering the skyline in which our model evidences some problems, we will address them in Section *Challenging Patterns*.



(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 2

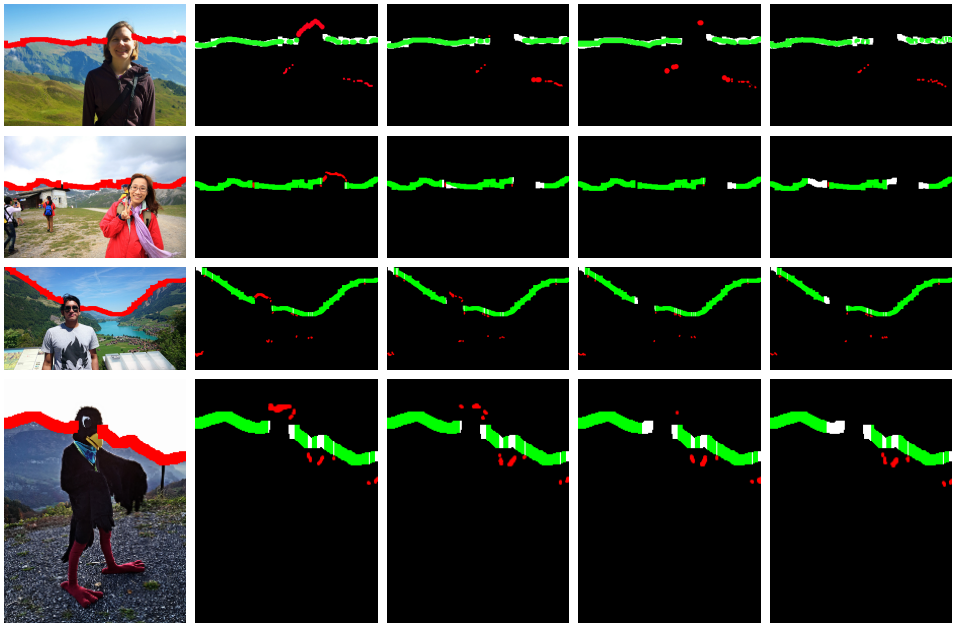
Figure 5.8: Occlusion Patterns: Cloudy Mountains. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

People

As was discussed in the previous section, people -as occluders- are scanty in our dataset, thus, we feared our model being slightly unaware of the intrinsic features that compose a regular non-occluding people, let alone people as occluders. However, as can be seen in Figure 5.9, our model seems to have generalized features that characterize more frequent occluders, naturally extending such acquired knowledge into more specific cases, for instance, people overlapping the skyline. Despite not being representative in the dataset used for training and test, this pattern is very common in outdoor mobile ap-

plications where users tend to heavily interact with the views and scenes. These images are evidence of the practical usefulness of our model.

We confirm that the 3 variants of our model are quite capable of recognizing the occlusions people are causing, performing much better than the baseline, which, unfortunately, fails to detect these occlusions and as we can observe, often misinterprets the top of the humans' head as being part of a continuous skyline rather than understand that it belongs to a different object, one that is occluding the horizon. Although in 2 of the pictures presented (the first and last picture in the figure), some pixels have been misclassified as skyline, we noted that, these misclassifications are not related to the intrinsic difficulty of the occluder, thus, they do not indicate that our model is not capable of recognizing people occluding the skyline; it is evident that our model is not confusing humanly shapes with skylines but the landscape edges present in the background.



(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 3

Figure 5.9: Occlusion Patterns: People. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

Challenging Patterns

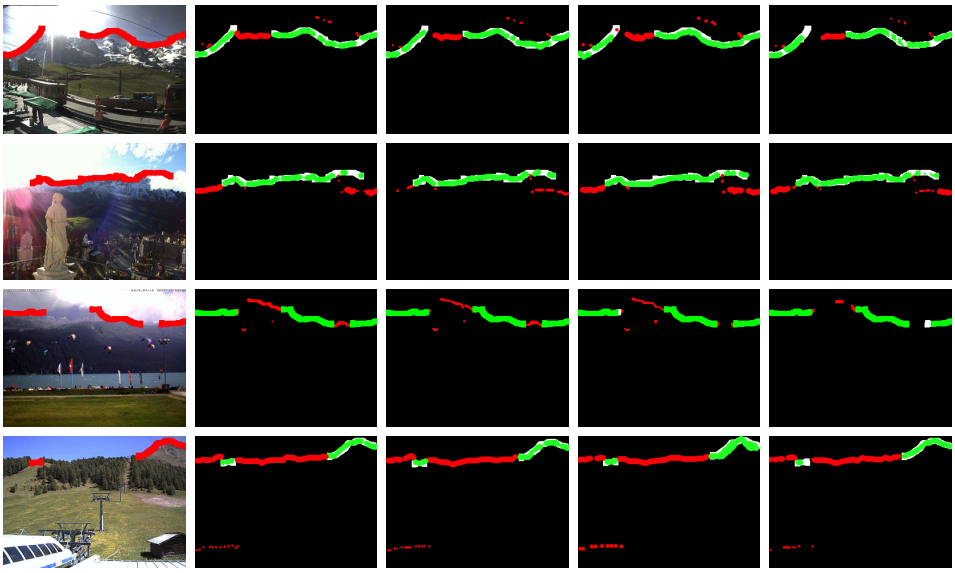
In the same way we were able to spot patterns in which the proposed model outperformed the baseline and showed considerable improvement and correction of misclassified occlusions, we have come to the realization that a couple of patterns exist that pose an extra challenge for our model, thus, predictions given on these patterns are not as accurate as in other cases. Figure 5.10 illustrates some examples of this nature.

The most recurrent scenarios in which misclassifications are high in number, are those containing blurry clouds covering the mountains and strong sunlight projected into the peaks present in the scene. The composition of elements inside images belonging to this category is quite difficult to interpret, it is not always clear -not even for humans- where an element ends and the other begins, the boundaries of the clouds or illumination are not crystal clear, in fact, not only improving our learning process to better handle these patterns becomes necessary but the ground-truth manual annotation process turns into a very complicated task, resulting in the introduction of some noise, indirectly affecting the overall learning process.

Surprisingly, we found severe misclassifications in scenes where *bushes* are covering the skyline (refer to the last picture in Figure 5.10). Such outcome is contradictory to what we stated before, when we confirmed our model's ability to distinguish bushes and trees as occlusions when pertinent; however, please do take care in observing this pattern as it is not depicting regular trees overlapping the skyline but hundreds of trees arranged together as a whole, although we were not able to find a significant amount of images with this amount of mispredictions, we consider it worth mentioning given the high false negative ratio per image output in images with this pattern; the reason for this rather contradictory result may lay in the image disposition of woods and peaks as well as its semantic difficulty.

We paid special attention to a peculiar pattern in which both the baseline and our model present a consistent behavior in the sense that they struggle to detect occlusions. *Blurry and Cloudy Mountains.* Figure 5.11 illustrates several pictures containing mountain peaks highly covered by clouds. Our model in all cases, performs better that the baseline, however, it throws poor predictions in most of the cases.

Picture number 2 demonstrates that high contrast between the skyline itself and the occluding cloud provides extra insights about the edges of the occlusion, thus, prediction becomes somehow manageable. Nonetheless, pic-

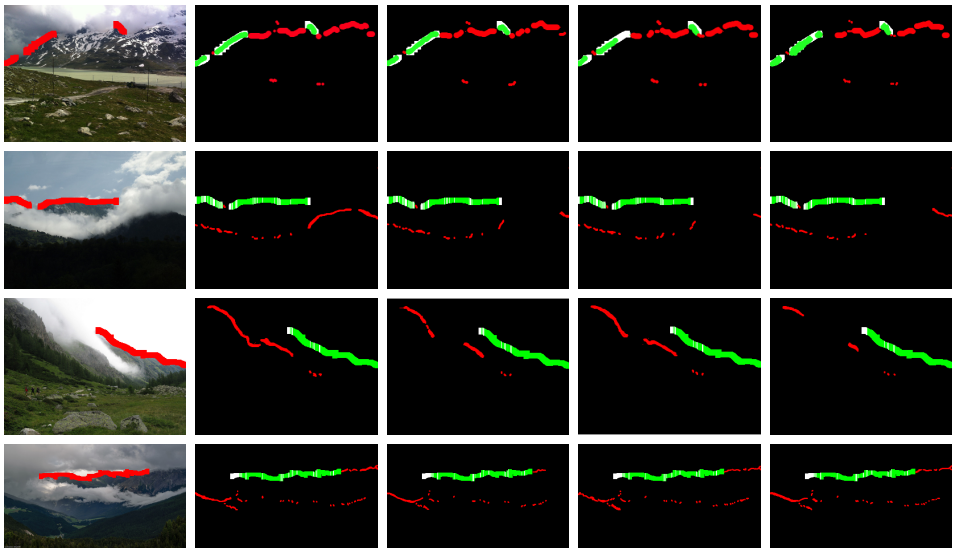


(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 3

Figure 5.10: Challenging Patterns. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

tures 1, 3 and 4, are a clear example of uncertainty, we are able to predict just small segments as occlusions, but whenever clouds blend so naturally -yet confusing- with the peaks, our model fails to provide consistent predictions, depicting the difficulty of these cases. It is worth pointing out that all these *occlusion* cases are very difficult to understand even for us humans, it is likely that while assessing many of these images, we would not easily agree in which regions we do find clouds being part of the background rather than covering the peaks.

These inaccuracies, however, were mildly anticipated. It is likely that the reason for this is the restricted amount and low variability of samples of this nature, preventing our ConvNet to learn such peculiarities. Further data collection would be needed to determine exactly how training samples depicting these patterns influence our model inference capabilities. Furthermore, a major source of uncertainty comes from the manual annotations used as ground-truth, in which several cases depicting similar scenes to the ones presented in Figure 5.10 and Figure 5.11 are *erroneously* annotated, due to the subjectivity associated with what defines an occlusion in complex images.



(a) Input (b) Baseline (c) Exp. 1 (d) Exp. 2 (e) Exp. 3

Figure 5.11: Challenging Patterns: Cloudy Mountains. Left column shows the ground-truth with the skyline annotated in red. The second column shows results obtained using the baseline [50], third, fourth and fifth show our results for experiments 1, 2 and 3 respectively. Skyline matches are shown in green, Skyline misses are shown in white, occlusion errors are shown in red.

5.4.2 On the Efficiency

One of the aims of this study, was to enhanced the occlusion detection accuracy of an existent model [50], such demand needed to be accomplished while respecting efficiency constraints associated with low powered devices, hence, the final component could be of practical use on, for instance, the various smartphones in the market. Fulfilling these constraints is not a trivial task and due care must be paid; even when, nowadays, mobile devices are packed with advanced computing capabilities and connectivity, decreasing the inference time of the proposed model becomes more decisive in *PeakLens-iOS*, where the identification, positioning and alignment of peaks are done by processing camera frames that come at a high frequency, hence, we rely on the maximum allowed speed.

We would like to remind the reader that the skyline extraction *combined model* described in this thesis is embedded in an AR mobile application that provides real-time mountain peak detection by processing camera frames at the maximum allowed speed a device can attain. Although, some efficiency metrics were reported in this work, showcasing results obtained at image level, it is also very important to account for the final user experience, in

which not only the speed of inference is assessed but the overall experience.

After inference is done, the application overlays pertinent augmented content of all visible mountain peaks inside the user's field of view. In *PeakLens-iOS* the initial peak positioning is done using only the DEM and the GPS and all compass sensors, the virtual panorama in view is estimated and peaks are projected onto the camera frame. No significant delay is encountered in sensor-based applications such *PeakLens-iOS*, peak positioning then, is not detrimental to the user experience. However, this method is extremely prone to errors in the DEM, GPS and compass. To minimize the effects of such errors on the user's perception of our application, the proposed *combined model* is exploited; by updating all peaks' position using the inputs from the camera view, the skyline extracted by our model and the skyline of the virtual panorama, the application is able to automatically correct substantial errors in the DEM, GPS position and compass, in real-time.

At what level the ConvNet execution time is perceived by the user is unknown and rather difficult to assess, however, being the model a vital component of the AR application pipeline, its inference velocity becomes of great importance, as we have demonstrated in Section 5.3.4, we are able to balance accuracy and execution time, hence, non prejudicial impact on the final user experience is introduced with the proposed approach.

Chapter 6

Conclusions and Future Work

In this work we have described a method to incorporate the uncertainties associated with the physical world into an outdoor AR mobile application for real time mountain peak detection. In AR applications, much of what the user sees and interacts with are non perfectly arranged physical objects about which the system has imperfect information, especially regarding their position and orientation relative to the user. Such uncertainties may either cause the information presented by the application to be misleading or make the augmentation meaningless.

The primary source of uncertainty in an application such PeakLens, is the arrangement of objects composing the scene with respect to the user's viewpoint. Positional uncertainties not only derive from noisy sensor-data readings but more importantly, from objects partially or completely occluding the skyline, hence, to provide accurate alignment between the virtual panorama and the physical world the application is trying to augment, we need to handle these uncertainties very carefully.

This thesis underlines the importance of expressing the aforementioned task as an image understanding problem in which not only the user's current position and device orientation are exploited but the actual scene the user is seeing at is treated as a rich source from where skyline occluders can be recognized and explicitly modeled in the solution.

A discussion on different mechanisms to cope with *the* mountain peak detection problem has been provided. In particular, we proposed alternatives using both, non-intelligent and intelligent approaches. Our contribution here

is twofold. First, *PeakLens-iOS*, a sensor-based iOS mobile application for mountain peak detection was developed; exploiting out-of-the-box resources and frameworks provided by the vendor, we tackle the uncertainties of such AR applications. Second, we took on the the path of artificial intelligence and introduced a convolution neural network to tackle the occlusion problem in the skyline detection task.

We presented a sensor-based application that effectively tracks the user's position in space in real-time by means of the Device Motion Tracking Service and use the best estimation of the output readings from the accelerometer and the gyroscope to estimate the *real* user's pose. Once the pose is estimated, a panorama based on the acquired user's position is obtained, later, an alignment *wrt* the orientation and field of view of the device is performed, consequently, mountain peaks are located in the refined panorama. Accordingly, the augmentation experience is successfully provided to the user by placing on the screen tags with pertinent information about the mountain peaks the user is seeing through the camera.

Promising results were obtained and we succeeded at locating and rendering peaks into the camera view, providing the user with a complete AR experience, however, understanding the correct relative position between the virtual objects and the several real objects composing a scene proved to be of great importance. Through the analysis of these results we explored the uncertainties that arise in AR applications such PeakLens, and came to the realization that estimating the user's interests based solely on data retrieved by the device-on-board position and orientation sensors -*which tend to be sensible to errors*-, is likely to result in misplaced augmented information about the peaks, making the whole AR experience less satisfactory. Furthermore, using only data provided by the sensors makes detecting occlusions in the skyline difficult to achieve, thus, they remained a problem at this stage. In our view, the main reason behind this, lies on the vision understanding that this approach fails to provide. In spite of being able to *see* what the user is seeing through the camera, pure sensor-based applications are not able to actually see.

As we learned from the drawbacks of this approach, we explored the power of artificial intelligence and focus on Deep Learning techniques, specifically ConvNets, that are known to be the *default* choice in most Computer Vision applications.

In particular, we have proposed a combined model (integrated in a cascade fashion) for detecting both skylines and occlusions, powered by a base-

line skyline detector [50] and a tailored 5-convolutional layer ConvNet devised for occlusion handling, learned from a large annotated training data. The strength of our work lies on this model's understanding of what constitutes a skyline and what constitutes an occlusion inside a scene. More importantly, using the proposed model presented in this work, we have been able to improve the performance of the skyline detector presented in [50] by almost **15%** in the detection of occlusions, over a large dataset of challenging mountainous images. These satisfactory results further demonstrate the validity of our approach to tackle occlusions in the skyline.

The sequential nature of the proposed combined model results in less than *70%* increase in the execution time in mid-range devices, resulting in around 3 images processed per second. Nonetheless, processing images at this rate is fully compatible with the usability requirements of a real-time AR application, since image processing is done in background with respect to the user interface; if the camera view movements are not too sudden, as one expects in a mountain peak recognition use case, the skyline extraction and the subsequent DEM alignment step could be done at a frequency lower than the 15 frames per second normally considered viable for video play.

Our work has provided further evidence that learning a deep ConvNet that consecutively models small pieces of information and combines them deeper in the network effectively handles the occlusion problem inherent to outdoor AR applications. To this end, we have experimented with 3 different sampling ratios of positive and negative classes (1:(1), 3:(1:1:1) and 1:(1:1:1)) to address the imbalanced nature of the occlusion problem, throughout the analysis of the experiments we have conducted, no evidence of the superiority of one experiment sampling ratio over the others was found; Experiment 3 showed to be slightly better than the other 2 experiments; however, the gain is not significant.

On this path, our most important findings are: (a) Information arranged in non-conventional shapes (columns instead of squares) can provide a deeper context to the network, (b) An occlusion-tailored model can be embedded in an already working model and (c) Despite all efforts on improving occlusion detectors, they will remain challenging in terms of dataset annotation.

Finally, our work has led us to conclude that compared to sensor-based techniques, the introduction of ConvNets in this domain provides greater support and detection accuracy. The fact that ConvNets are trained rather than programmed, makes applications using this approach, to take better advantage of the enormous amount of data available in nowadays' websphere.

However, benefits do not come without trade-offs and challenges. ConvNets (and other Deep Learning techniques) require non-negligible amounts of computing resources, for both training and inferencing stages. Furthermore, ConvNets are quite data hungry, they rely not only on large datasets but foremost, good quality data. In an image classification task, results depend directly on the images fed into the network, in order to achieve adequate performance, it is required to have, if not high, at least good resolution images and proper ground-truth labels. This becomes especially important for applications such PeakLens, in which it is necessary to detect objects (skyline and occluders) in the distance.

Future Work

Future work concerns deeper analysis of particular techniques, trying different methods, correcting sources of discrepancy identified during this study, completing the application cycle or simply curiosity. Due to time limitations several adaptations and experiments that we would have liked to try during the implementation of the network architecture in Chapter 4 have been left for the future. Broadly speaking, this thesis has been mainly focused on the development of an outdoor AR mobile application and the use of ConvNets for skyline detection where most of the resources used to achieve comparable results where adapted from previous work, leaving a deep exploration of several state-of-the-art networks outside the scope of the thesis. In the following, we detail several approaches and suggestions that we consider could contribute important insights to the findings we presented.

iOS Integration

Throughout this thesis we have emphasized how *learning* techniques, such as ConvNets, can improve the mountain peak detection by recognizing occluders in the skyline, although the proposed *combined model* introduced in Section 4.4 has been successfully tested on Android devices running PeakLens, future work should concentrate on completing the Computer Vision framework introduced in Section 3.5.2, so as to embed the final combined model into *PeakLens-iOS* and turning it into an intelligent system.

Dataset distribution

It is worth noticing that the dataset we have worked upon is a fair source of challenging pictures. Nonetheless, this work has given rise

to many insights on the appropriateness of this dataset to handle the *occlusion problem*, which is a fundamental issue for future research on this topic. Thus, we consider future work should concentrate on the quality of the dataset and variability of images included. It is recommended to enrich the dataset in such a way that it represents a real distribution of outdoor scenes, where images have all sort of obstacles rather than being professional-like, taken under perfect-*ish* conditions. To this end, not only more images containing occluded skyline must be included, but more complex scenes should also be present.

Fine-coarse Ground-truth Annotation

As with the distribution of the dataset, another interesting insight emerged in the course of this study. At present, the ground-truth distinguishes skyline pixels only, everything else is, thus, treated as a negative pixel, indistinctly of its semantic connotation with respect to the skyline in the scene. Consequently, a promising step towards better occlusion detection is the explicit annotation of occlusions. We foreseen such measure could positively impact the accuracy of the model. This, however, is not a trivial task as it is costly and highly subjective when performed by humans.

Class Balancing

We evidence that occlusion handling in the skyline detection task is inherently an unbalanced classification problem due to the uneven nature of real world images containing skylines vs occlusions. We demonstrated that one way to avoid learning a trivial classifier that always detects skylines, is using a *sample technique* in which we balanced the dataset by down-sampling the high-occurrence class before feeding it into the network. In fact, we took a step forward and experiment with different sampling ratios corresponding to various non-arbitrary cost functions. An alternative solution will be to work directly on the loss function and train a more balanced model by re-weighting each class in the loss function. *Median frequency balancing* is often used for this purpose since encourages the network to learn less frequently occurring classes, for instance, *occlusions* [25] [45] [77]; it works by assigning in the loss function corresponding weights to every class depending on the frequency in which the class occurs in the train dataset. Other alternative techniques exist, such as optimizing

the Intersection-Over-Union, that have been proven to be very helpful, however they are most commonly used in segmentation problems [82].

Exploration of the hyper-parameter space

As stated before, this work did not focus on an exhaustive exploration of hyper-parameters when learning our model. In fact, we carried out quite a superficial manual search of the hyper-parameter space, in which we used our knowledge about the problem, guess parameters and observe the results. However, this is likely to lead suboptimal solutions. With computational and time constraints we were not able to explore more different combinations of hyper-parameters that could possibly yield better performance. Nonetheless, we believe our work can be considered a good starting point for further explorations, a *grid search* could be applied, or even a *random search* that usually works better than other methods [29]. More recent work has been focused on *Bayesian Optimization*, in which information gained from an experiment is later used to decide how to adjust hyper-parameters for the next experiment [91].

Further Data Preprocessing

Normalizing image inputs

In this work, we have not experimented normalizing images before feeding them into our ConvNet, however, we do have set up our experiments to subtract mean pixels in both training and inference stages. Given that this type of normalization has been proven to make convergence faster while training a network, an interesting test should include this preprocessing method and study its consequences, if any. One of the most common image normalization methods is *mean-centering*, it acts by ensuring each input parameter having a similar data distribution and is done by subtracting the mean from each pixel, and then dividing the result by the standard deviation. After mean-centering, each mean-centered pixel shows only how it differs from the average sample in the original image.

Standard deviation of image inputs

Throughout a visual inspection of our dataset, we have observed that many images have skylines located in the upper half, it could be use-

ful though, to look at the *mean image* obtained by taking the mean values for each pixel across all samples. Furthermore, we could take the standard deviation of all images and analyze where the higher variance lies. Observing this could give us insight into some underlying structure in the images such as variations in boundaries or corners. With this understanding, we may choose to augment our dataset with richer viewpoints so as to not have input images with only one innate structure.

Outcome Post-processing

Results obtained with our model are very promising, as reported earlier, around **15%** improvement, with respect to the baseline, in the accuracy of non-skyline detection was introduced with our proposed *combined model*. Currently, we do not perform any sort of post-processing on the final outcome. One potential step towards finer detection could imply post-processing the output of our model. Throughout a qualitative analysis of the test set predictions, we have identified that several predictions were made in such a way that narrow segments of around 20 pixels surrounded by true non-skyline, were labeled as *skylines*. Skylines being that narrow in a real scene, are almost certainly *unreal*, to overcome this issue, we could put in good use a post-processing step in which we eliminate these discrepancies. Despite this would only improve the non-skyline accuracy slightly, we consider it as a viable course. We estimate this could be done at a minimal cost in the execution time.

In suchlike manner, we encountered various predictions in which small sections (around 20 pixels) surrounded by true skyline, were labeled as *occlusions*. Occlusions being that narrow in a real scene, is almost certainly *unreal*, so the post-processing step could also include removal of errors of this kind. Although such measure would improve the overall skyline accuracy rather than the non-skyline accuracy, we consider it an important step towards a better quality model.

Bibliography

[1] Android developer documentation. https://developer.android.com/index.html. Accessed: 2017-10-06.

[2] Apple developer documentation. https://developer.apple.com/documentation. Accessed: 2017-08-14.

[3] Apple products technical specifications. https://support.apple.com/specs/iphone. Accessed: 2017-09-02.

[4] Arkit. https://developer.apple.com/arkit/. Accessed: 2018-01-23.

[5] Augmented reality in ios. https://www.apple.com/lae/ios/augmented-reality/. Accessed: 2018-01-23.

[6] iphone market. http://fortune.com/2017/03/06/apple-iphone-use-worldwide/. Accessed: 2017-09-11.

[7] Opencv. https://opencv.org/about.html. Accessed: 2017-09-19.

[8] Peakar. https://peakar.salzburgresearch.at. Accessed: 2018-01-16.

[9] Peakfinder. https://www.peakfinder.org/. Accessed: 2017-11-19.

[10] Peaklens - mountain identification android mobile app. http://peaklens.com/. Accessed: 2018-01-10.

[11] Peaklens renderer. https://render.peaklens.com/api/mobilebundle. Accessed: 2017-09-19.

[12] Peakvisor. https://peakvisor.com. Accessed: 2018-01-16.

[13] Showmehills. http://www.showmehills.com/. Accessed: 2018-01-16.

[14] Starwalk. http://vitotechnology.com/star-walk.html. Accessed: 2018-01-02.

[15] Stereo matching. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT11/node5.html. Accessed: 2018-01-15.

[16] Vuforia. https://vuforia.com/. Accessed: 2017-10-08.

[17] Wikitude. https://www.wikitude.com/. Accessed: 2017-10-08.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[19] Touqeer Ahmad, George Bebis, Monica Nicolescu, Ara Nefian, and Terry Fong. An edge-less approach to horizon line detection. In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on*, pages 1095–1102. IEEE, 2015.

[20] Yiannis Aloimonos. Guest editorial: Qualitative vision. *International Journal of Computer Vision*, 14(2):115–117, 1995.

[21] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 33(5):898–916, 2011.

[22] Ronald T Azuma. The challenge of making augmented reality work outdoors. *Mixed reality: Merging real and virtual worlds*, pages 379–390, 1999.

[23] Georges Baatz, Olivier Saurer, Kevin Köser, and Marc Pollefeys. Large scale visual geo-localization of images in mountainous terrain. In *Computer Vision–ECCV 2012*, pages 517–530. Springer, 2012.

[24] Lionel Baboud, Martin Čadík, Elmar Eisemann, and Hans-Peter Seidel. Automatic photo-to-terrain alignment for the annotation of mountain pictures. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 41–48. IEEE, 2011.

[25] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.

[26] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[27] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[28] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[29] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[30] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[31] Carlo Bernaschina, Roman Fedorov, Darian Frajberg, and Piero Fraternali. A framework for regression testing of outdoor mobile applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 179–181. IEEE Press, 2017.

[32] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[33] Jan Brejcha and Martin Čadík. Geopose3k: Mountain landscape dataset for camera pose estimation in outdoor environments. *Image and Vision Computing*, 66:1–14, 2017.

[34] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M Buhmann. The balanced accuracy and its posterior distribution. In *Pattern recognition (ICPR), 2010 20th international conference on*, pages 3121–3124. IEEE, 2010.

[35] John Canny. A computational approach to edge detection. In *Readings in Computer Vision*, pages 184–203. Elsevier, 1987.

[36] Dan C Cireşan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. Mitosis detection in breast cancer histology images with

deep neural networks. In *International Conference on Medical Image Computing and Computer-assisted Intervention*, pages 411–418. Springer, 2013.

[37] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[38] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[39] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[40] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[41] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE, 2013.

[42] Olivier Delalleau and Yoshua Bengio. Shallow vs. deep sum-product networks. In *Advances in Neural Information Processing Systems*, pages 666–674, 2011.

[43] Satyan L Devadoss and Joseph O'Rourke. *Discrete and computational geometry*. Princeton University Press, 2011.

[44] Rae A Earnshaw. *Virtual Reality Systems*. Academic press, 2014.

[45] David Eigen and Rob Fergus. Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2650–2658, 2015.

[46] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[47] Roman Fedorov, Darian Frajberg, and Piero Fraternali. A framework for outdoor mobile augmented reality and its application to mountain peak detection. In *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*, pages 281–301. Springer, 2016.

[48] Roman Fedorov, Piero Fraternali, and Marco Tagliasacchi. Mountain peak identification in visual content based on coarse digital elevation models. In *Proceedings of the 3rd ACM International Workshop on Multimedia Analysis for Ecological Data*, pages 7–11. ACM, 2014.

[49] Robert Fergus, Pietro Perona, and Andrew Zisserman. Object class recognition by unsupervised scale-invariant learning. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 2, pages II–II. IEEE, 2003.

[50] Darian Frajberg, Piero Fraternali, and Rocio Nahime Torres. Convolutional neural network for pixel-wise skyline detection. In *ICANN*, page 8, 2017.

[51] Golnaz Ghiasi. *Recognizing and Segmenting Objects in the Presence of Occlusion and Clutter*. PhD thesis, UC Irvine, 2016.

[52] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[53] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[54] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*, volume 1. MIT press Cambridge, 2016.

[55] J. Ha and H. Jeong. Occlusion filling in dynamic programming with simple index treatment. In *2012 12th International Conference on Control, Automation and Systems*, pages 2163–2166, Oct 2012.

[56] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. corr abs/1512.03385 (2015), 2015.

[58] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[59] Geoffrey E Hinton. A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade*, pages 599–619. Springer, 2012.

[60] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[61] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[62] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[63] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

[64] JINHO KIM[1], Byung-Soo Kim, and Silvio Savarese. Comparing image classification methods: K-nearest-neighbor and support-vector-machines. *Ann Arbor*, 1001:48109–2122, 2012.

[65] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[66] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

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

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

[69] Eep Kumar, Zeeshan Khan, and Anurag Jain. A review of content based image classification using machine learning approach. 2012.

[70] Nicolas Le Roux and Yoshua Bengio. Deep belief networks are compact universal approximators. *Neural computation*, 22(8):2192–2207, 2010.

[71] Y LeCun, L Bottou, G Orr, and K Muller. Efficient backprop in neural networks: Tricks of the trade (orr, g. and müller, k., eds.)[j]. *Lecture Notes in Computer Science*, 1524.

[72] Yann Lecun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L.D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[73] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, page 20, 2015.

[74] Gun A Lee, Andreas Dünser, Seungwon Kim, and Mark Billinghurst. Cityviewar: A mobile outdoor ar application for city visualization. In *Mixed and Augmented Reality (ISMAR-AMH), 2012 IEEE International Symposium on*, pages 57–64. IEEE, 2012.

[75] Wei-Han Liu and Chih-Wen Su. Automatic peak recognition for mountain images. In *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*, pages 1115–1121. Springer, 2014.

[76] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[77] Richard F Lyon and Larry S Yaeger. On-line hand-printing recognition with neural networks. In *Microelectronics for Neural Networks, 1996., Proceedings of Fifth International Conference on*, pages 201–212. IEEE, 1996.

[78] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[79] Kamini Nalavade and BB Meshram. Data classification using support vector machine. In *National Conference on Emerging Trends in Engineering & Technology (VNCET)*, volume 2, pages 181–184, 2012.

[80] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012.

[81] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

[82] Md Atiqur Rahman and Yang Wang. Optimizing intersection-over-union in deep neural networks for image segmentation. In *International Symposium on Visual Computing*, pages 234–244. Springer, 2016.

[83] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[84] Fedorov Roman, Martinenghi Davide, Tagliasacchi Marco, and Castelletti Andrea. Exploiting user generated content for mountain peak detection. In *2nd International Workshop on Social Media for Crowdsourcing and Human Computation (SoHuman 2013)*, pages 21–28, 2013.

[85] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para.* Cornell Aeronautical Laboratory, 1957.

[86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[87] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[88] Stephen R Schach. *Software Engineering.* McGraw-Hill Professional, 1999.

[89] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings*

of the IEEE conference on computer vision and pattern recognition, pages 815–823, 2015.

[90] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[91] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[92] Yale Song, Louis-Philippe Morency, and Randall Davis. Distribution-sensitive learning for imbalanced datasets. In *Automatic Face and Gesture Recognition (FG), 2013 10th IEEE International Conference and Workshops on*, pages 1–6. IEEE, 2013.

[93] Ivan E Sutherland. The ultimate display. *Multimedia: From Wagner to virtual reality*, pages 506–508, 1965.

[94] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[95] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[96] DWF Van Krevelen and Ronald Poelman. A survey of augmented reality technologies, applications and limitations. *International Journal of Virtual Reality*, 9(2):1, 2010.

[97] Slobodan Vucetic and Zoran Obradovic. Classification on data with biased class distribution. In *European Conference on Machine Learning*, pages 527–538. Springer, 2001.

[98] Ruohui Wang. Edge detection using convolutional neural network. In *International Symposium on Neural Networks*, pages 12–20. Springer, 2016.

[99] Markus Weber, Max Welling, and Pietro Perona. Towards automatic discovery of object categories. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 101–108. IEEE, 2000.

[100] Joseph N Wilson and Gerhard X Ritter. *Handbook of computer vision algorithms in image algebra*. CRC press, 2000.

[101] Luke Yeager, Julie Bernauer, Allison Gray, and Michael Houston. Digits: the deep learning gpu training system. In *ICML 2015 AutoML Workshop*, 2015.

[102] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Appendix A

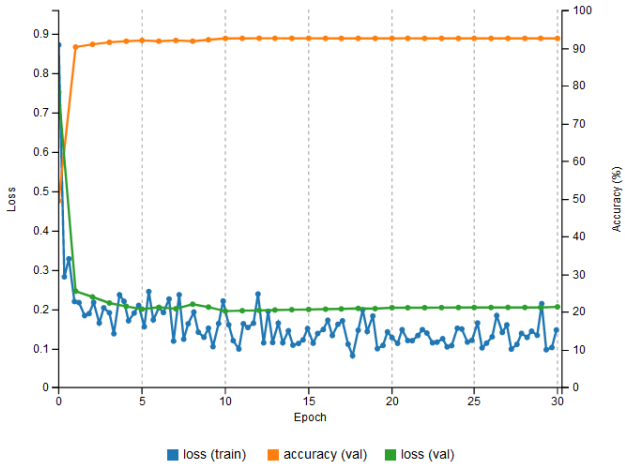
Occlusion ConvNet

A.1 Accuracy and Loss

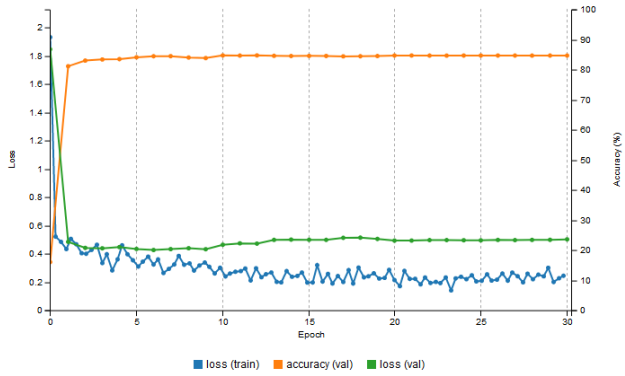
The *epoch* with the best trade-off between accuracy and loss in the validation set was chosen for generating the weights. Table A.1 shows the selected Epoch, Accuracy and Loss in the validation set. Figure A.1 depicts the Loss and Accuracy obtained by the network at each *epoch*.

	Epoch	Accuracy	Loss
Experiment 1	10	0.9304	0.1924
Experiment 2	6	0.8458	0.4291
Experiment 3	6	0.8305	0.4460

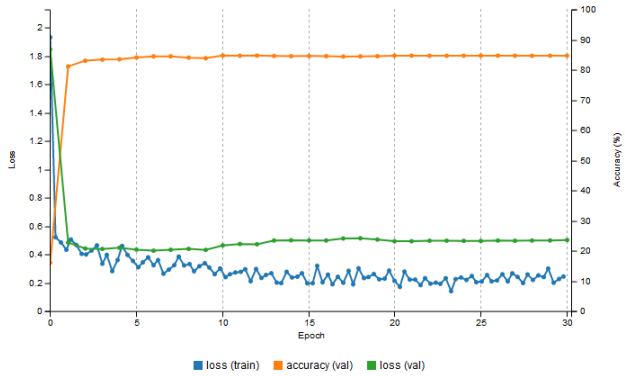
Table A.1: Validation Metrics. Selected *epoch*, *loss* and *accuracy* in the validation set.



(a) Experiment 1



(b) Experiment 2



(c) Experiment 3

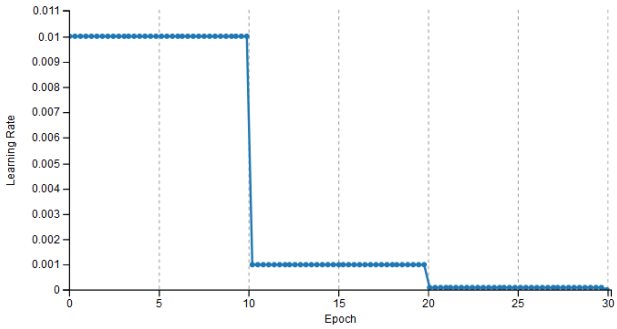
Figure A.1: Loss (training and validation) and Accuracy (training) reported for all conducted experiments.

A.2 Learning Rate Decay

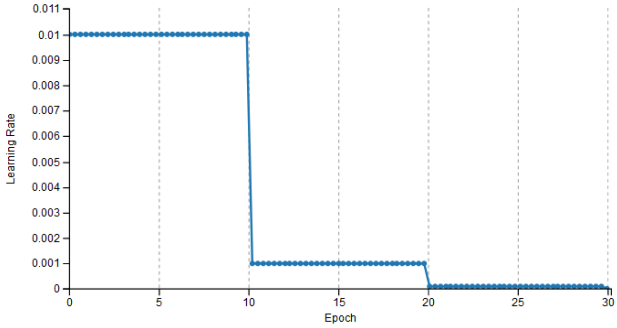
Table A.2 shows the configured learning rate `lr` and the batch size used in the training and validation sets. Figure A.2 shows the learning decay.

	lr	Batch Size	
		training	validation
Experiment 1	0.0010	64	32
Experiment 2	0.0100	512	256
Experiment 3	0.0100	512	256

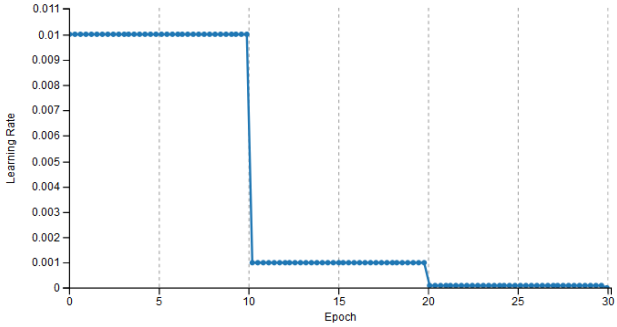
Table A.2: Learning Rate and Batch Size. Learning rate and batch size used during training and validation.



(a) Experiment 1



(b) Experiment 2



(c) Experiment 3

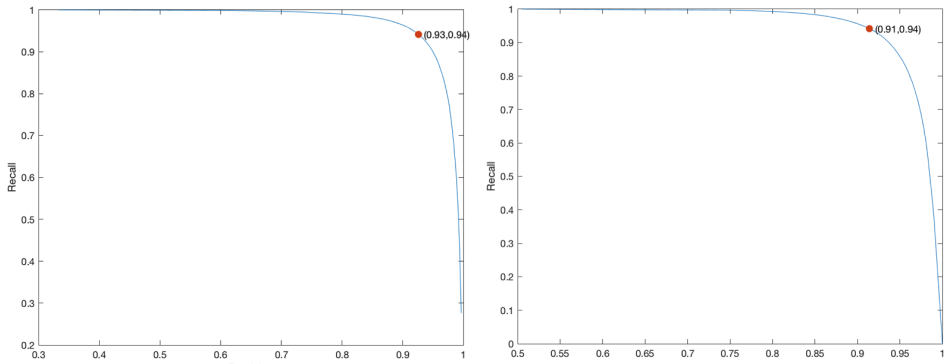
Figure A.2: Learning rate decay for each conducted experiment.

Appendix B

ConvNet Evaluation

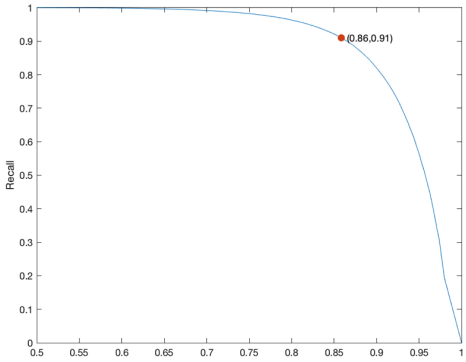
B.1 Precision-Recall Curves

Figure B.1 shows the Precision-Recall curves obtained during the evaluation of the Baseline and the Column Patch-wise evaluation of the test set performed in the 3 experiments presented in this work.

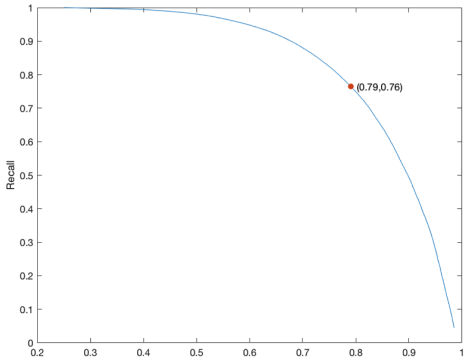


(a) Baseline

(b) Experiment 1



(c) Experiment 2



(d) Experiment 3

Figure B.1: Precision-Recall Curves. Values obtained in the evaluation of the (a) Baseline and the evaluation at Column Patch-wise level conducted on (a) Experiment 1 (b) Experiment 2 and (c) Experiment 3.

B.2 Performance Metrics

Baseline

Figure B.2 depicts the Average Accuracy, Average No Skyline Accuracy and the Average Accuracy obtained during the evaluation of the test set conducted using the *Baseline* model.

Experiment 1

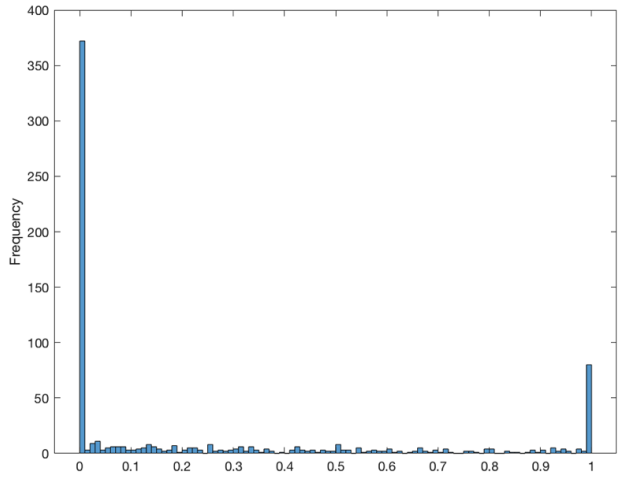
Figure B.3 portrays the Average Accuracy, Average No Skyline Accuracy and the Average Accuracy obtained during the evaluation of the test set conducted on *Experiment 1*.

Experiment 2

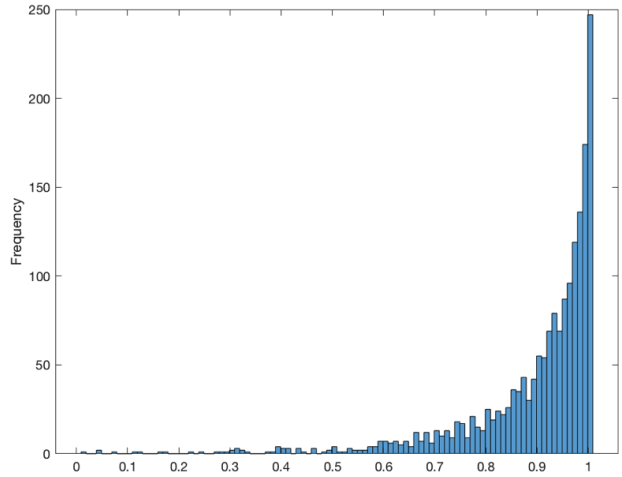
Figure B.4 depicts the Average Accuracy, Average No Skyline Accuracy and the Average Accuracy obtained during the evaluation of the test set conducted on *Experiment 2*.

Experiment 3

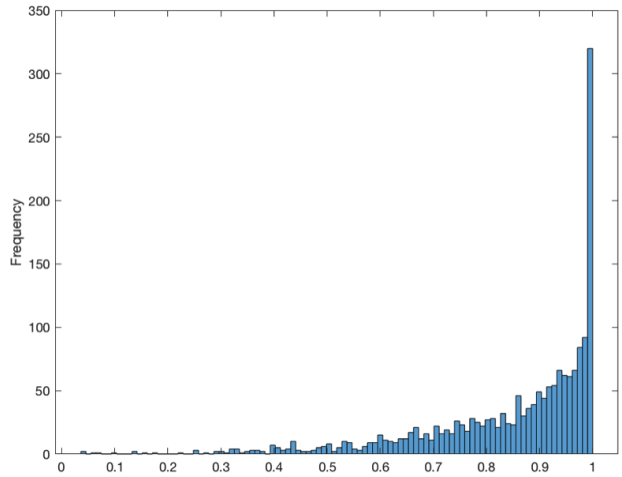
Figure B.5 depicts the Average Accuracy, Average No Skyline Accuracy and the Average Accuracy obtained during the evaluation of the test set conducted on *Experiment 3*.



(a) ANSA

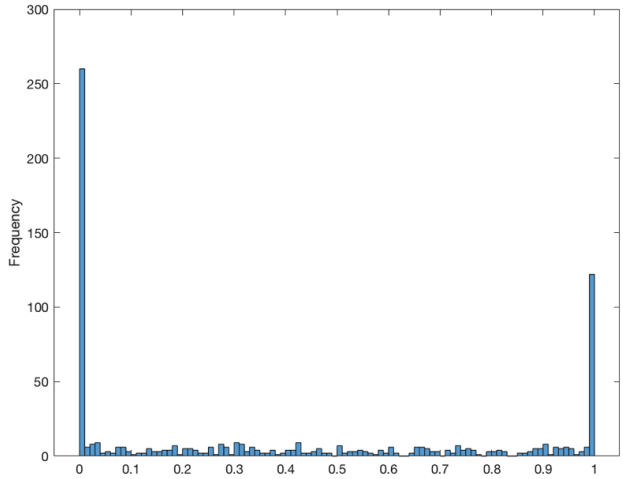


(b) ASA

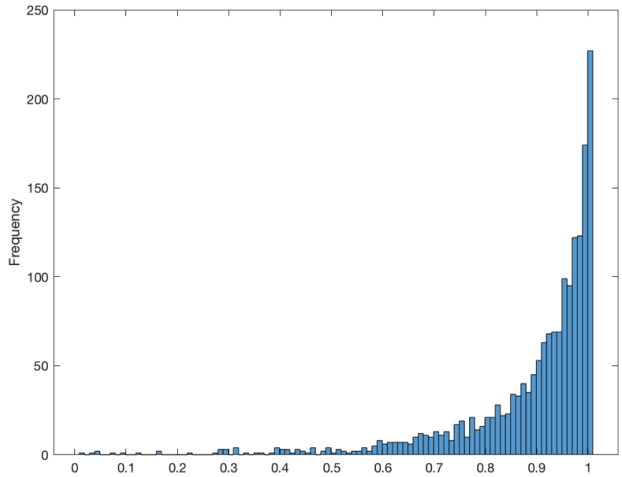


(c) AA

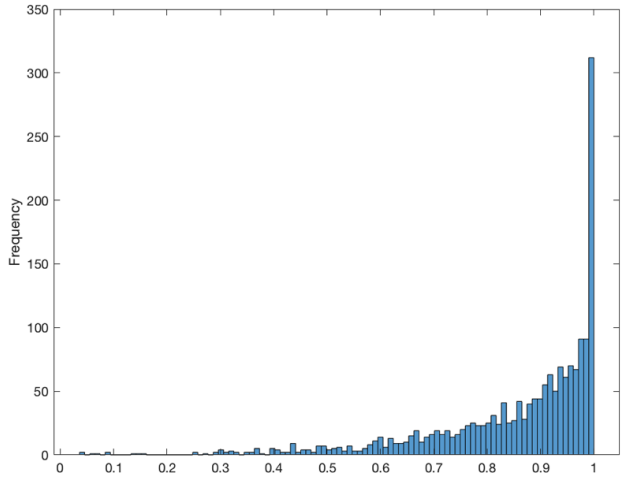
Figure B.2: Baseline Performance Metrics. Test set results in (a) Average Non-Skyline Accuracy (b) Average Skyline Accuracy (c) Average Accuracy.



(a) ANSA

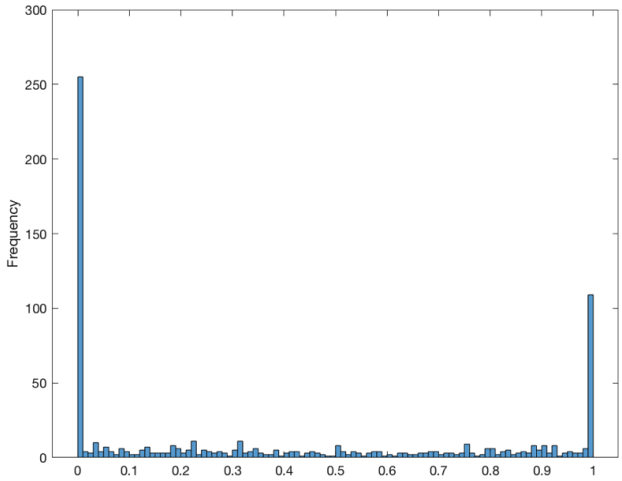


(b) ASA

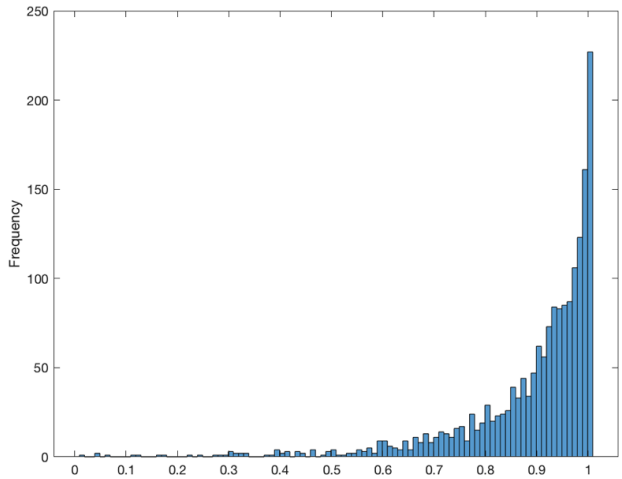


(c) AA

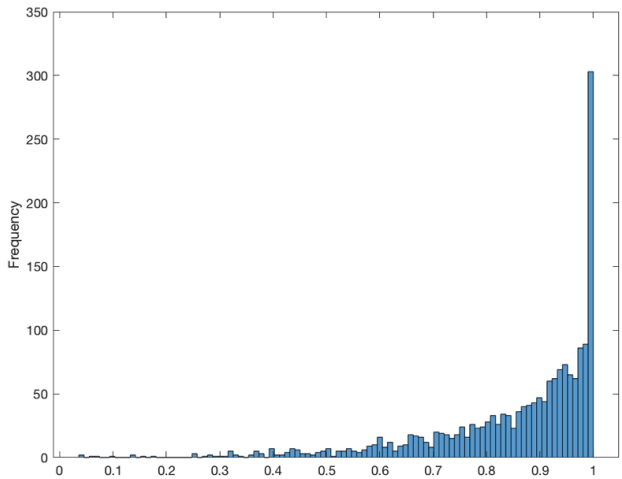
Figure B.3: Experiment 1 Performance Metrics. Test set results in (a) Average Non-Skyline Accuracy (b) Average Skyline Accuracy (c) Average Accuracy.



(a) ANSA

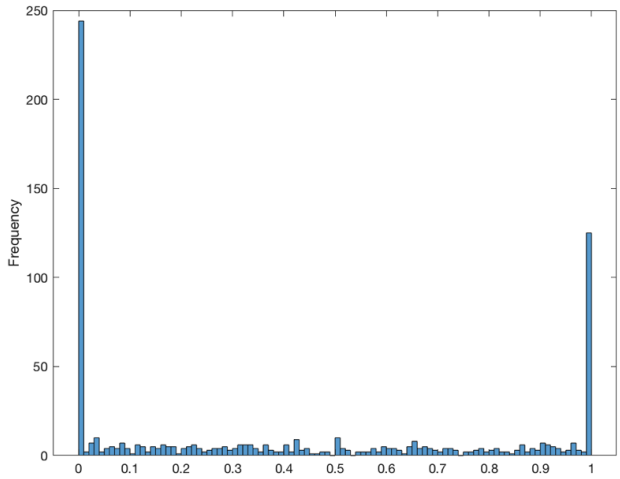


(b) ASA

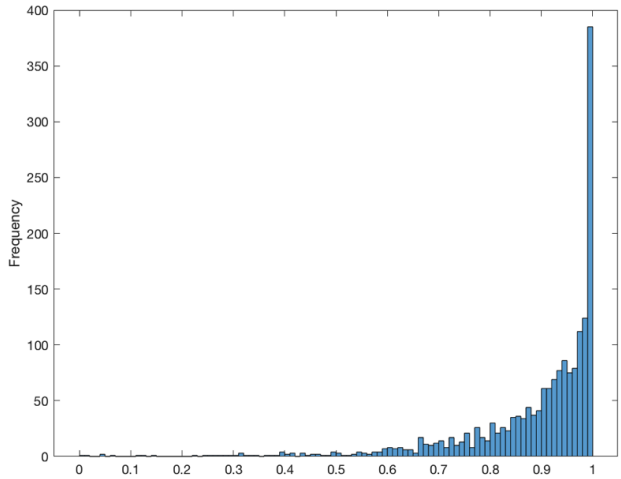


(c) AA

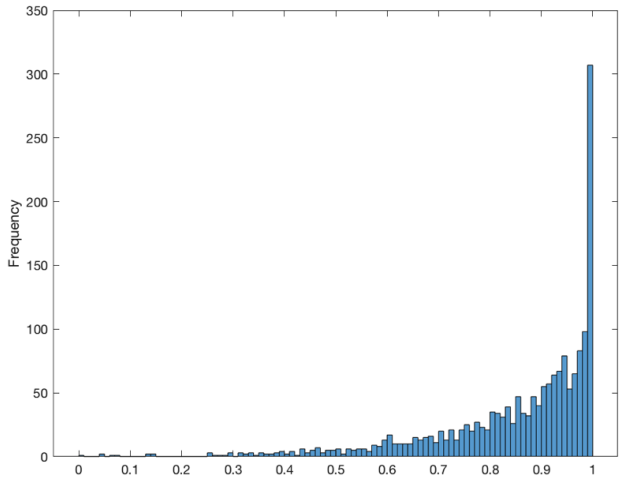
Figure B.4: Experiment 2 Performance Metrics. Test set results in (a) Average Non-Skyline Accuracy (b) Average Skyline Accuracy (c) Average Accuracy.



(a) ANSA



(b) ASA

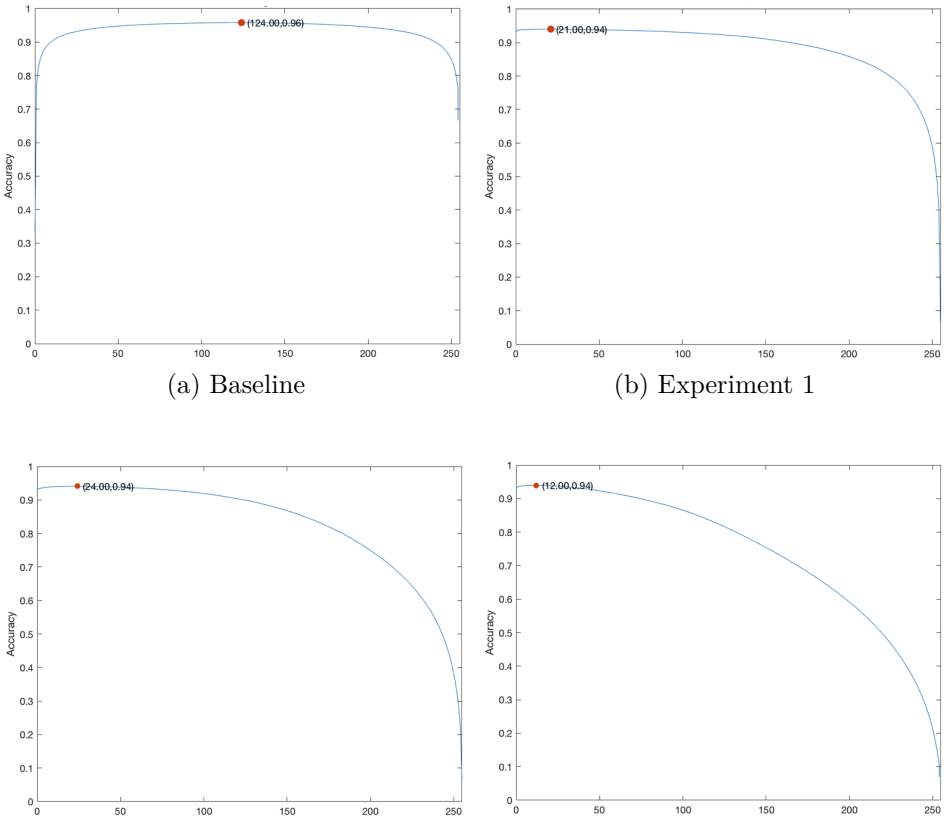


(c) AA

Figure B.5: Experiment 3 Performance Metrics. Test set results in (a) Average Non-Skyline Accuracy (b) Average Skyline Accuracy (c) Average Accuracy.

B.3 Decision Cutoff

Figure B.6 depicts the Threshold-Accuracy ROC curves used to determine an appropriate threshold (cutoff value) in all our experiments and the baseline. To configure the classifier, we computed this threshold using samples from the validation set.



(a) Baseline (b) Experiment 1

(c) Experiment 2 (d) Experiment 3

Figure B.6: Decision Threshold Values. Cutoffs obtained from the validation set using the (a) Baseline (b) Experiment 1 (c) Experiment 2 and (d) Experiment 3.