BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

HORIZON DETECTION IN IMAGE

DETEKCE HORIZONTU VE FOTOGRAFII

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **NATÁLIA HOLKOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. ROMAN JURÁNEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Bachelor's Thesis Specification

Student: **Holková Natália**
Programme: Information Technology
Title: **Horizon Detection in Image**
Category: Image Processing

Assignment:

1. Study methods for automatic detection of horizon line in images. Focus on deep learning methods.
2. Find suitable datasets or acquire your own data.
3. Design a new method or modify existing method for horizon line detection
4. Evaluate the method and compare it to other existing methods.
5. Discuss opportunities for improvements of your method.
6. Create presentation materials (e.g. poster)

Recommended literature:

- Zhai et al, Detecting Vanishing Points using Global Image Context in a Non-Manhattan World, CVPR 2016
- Workman et al,Horizon Lines in the Wild, BMVC 2016
- Simon et al, A-Contrario Horizon-First Vanishing Point Detection Using Second-Order Grouping Laws, ECCV 2018

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Juránek Roman, Ing., Ph.D.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: January 14, 2021

Abstract

This thesis aims to implement a method of detecting the horizon line in images using deep learning to prevent any constraints on input data. A training dataset is created by downloaded images from large metropolitan cities around the world using the Google Street View service. Several popular architectures for convolutional neural networks are chosen, and their performance is evaluated on existing benchmark datasets.

Abstrakt

Cieľom tejto práce je naimplementovať metódu detekovania horizontu vo fotografii pomocou hlbokého učenia, aby sa zabránilo obmedzeniam pre vstupné dáta. Trénovací dataset bol vytvorený sťahovaním obrázkov z miest z celého sveta pomocou služby Google Street View. Bolo vybratých niekoľko populárnych architektúr pre konvolučné neurónové siete a po natrénovaní boli vyhodnotené na existujúcich testovacích datasetoch.

Keywords

CNN, neural network, deep learning, horizon line estimation, Keras, Google Colab

Kľúčové slová

CNN, neurónové siete, hlboké učenie, odhad pozície horizontu, Keras, Google Colab

Reference

HOLKOVÁ, Natália. *Horizon Detection in Image*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Roman Juránek, Ph.D.

Rozšírený abstrakt

Táto práca sa zaoberá problematikou detekcia horizontu vo fotografii pomocou hlbokého učenia. Využitím práve hlbokého učenia sa má zabrániť tomu, aby boli kladené požiadavky na vstupné obrázky, ako napríklad prítomnosť zbiehajúcich sa úbežníkov. Takáto podmienka podstatne by sťažovala detekciu horizontu na miestach s menšou alebo neexistujúcou zástavbou.

Práca popisuje niekoľko existujúcich riešení pre detekciu horizontu, z ktorých niektoré využívajú práve metódy hlbokého učenia. Na týchto metódach sa zakladá táto práca. Ďalej sa v tejto práci popisuje základná teória ohľadom konvolučných neurónových sietí, vrátane niekoľkých populárnych architektúr, ktoré sú použité pri trénovaní modelu.

Pre testovanie validity detektora horizontu existuje niekoľko datasetov: York Urban Dataset, Eurasian Cities Dataset a z nich najnovší a najväčší Horizon Lines in the Wild, ktorý čiastočne slúži aj na trénovanie. Avšak nie je dostupný vhodný trénovací dataset, preto bolo potrebné vytvoriť vlastný. Na toto bola použitá služba Google Street View, ktorej API umožňuje sťahovať fotografie z takmer celého sveta. Pri sťahovaní dovoľuje špecifikovať parametre ako sklon kamery alebo veľkosť zorného poľa. Vďaka tomuto sa dá automaticky vypočítať pozícia horizontu a nie je potrebné ručné anotovať jednotlivé obrázky. Takto získané obrázky nemajú žiadne otočenie a preto ich dodatočne treba rotovať.

Drvivá väčšina modelov hlbokého učenia predikuje jednu triedu alebo hodnotu. Horizont, the priamku, nie je možné popísať jedinou hodnotou. Je ho ale možné popísať dvoma hodnotami, a to uhlom priamky a vzdialenosťou priamky od stredu obrázku. Trénovaný model preto bude produkovať dve výstupné hodnoty.

Inštinktívne je možné poňať úlohu určenia horizontu ako regresnú, teda produkujúce konkrétnú hodnotu. V tejto práci je to avšak brané ako úloha klasifikačné, kde model vracia príslušnosť k triede. Tento prístup dovoľuje sa ľahšie riešiť problém odchýlok vrámci datasetu. Použité architektúry konvolučných sietí museli byť upravené, aby umožnili viacero rôznych výstupov. Toto je docielené tak, že sa ponechá vstupná vrstva spolu s konvolučnými vrstami a následne sa model rozvetví.

Implementácia prebiehala v jazyku Python a boli využitá knižnica Keras a TensorFlow. Modely boli trénované na platforme Google Colaboratory. Kvôli veľkosti trénovacieho datasetu bolo potrebné vytvoriť dátový generátor, aby sa umožnilo trénovať model po dávkach.

Natrénované modely boli vyhodnoté na dvoch testovací datasetoch. Metrikou je vzdialenosť predikované horizontu od skutočného anotovaného horizontu normalizovaná výškou obrázku. Pri testovaní architektúra AlexNet dosiahla spomedzi použitých architektúr najhoršie výsledky, zatiaľ čo architektúra Inception V3 dosiahla najlepšie.

Avšak aj tieto výsledky nedosiahli hodnoty najmodernejších metód. Je preto stále priestor pre zlepšenie, či získaním väčšieho trénovacieho datasetu alebo uplatnením metód na regularizáciu, aby sa predišlo nadmernému pretrénovaniu. Plne funkčný model by bolo zaujímavé do budúcna využiť a zakomponovať do aplikácií, ako webová aplikácia pre korekciu natočenia fotografií podľa horizontu.

Horizon Detection in Image

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Roman Juránek, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Natália Holková
May 12, 2021

Acknowledgements

I would like to thank my supervisor for his guidance and help with the thesis.

Contents

Chapter 1

Introduction

This thesis deals with horizon detection in photos using deep neural networks. The Horizon line represents valuable information for a variety of tasks. While many methods on horizon line estimation from a single image have been proposed, most do not use deep learning. Furthermore, they often require the image to contain at least some clues, such as vanishing points.

This thesis analyzes state-of-the-art methods for horizon line estimation, which use deep learning, and implements a method that should solve some of their shortcomings. Existing methods are described in chapter 2. chapter 3 introduces convolutional neural networks and architectures that are used in this thesis. chapter 4 describes the metric used for measuring the accuracy of the horizon line detectors and the existing datasets used for evaluation. chapter 5 discusses the creation of a training dataset and approach to detecting horizon lines. In chapter 6 are described used tools and implementation of convolutional neural network. chapter 7 details evaluation of trained models on benchmark datasets and discusses possible improvements.

Chapter 2

Analysis of Existing Methods to Horizon Detection

2.1 Problem Definition

The horizon is the line that separates the earth from the sky. Frequently, the horizon line is obscured by, for example, buildings, especially in urban areas, and may not be easily seen. However, it can still be determined using the vanishing points to see where the mutually parallel lines appear to converge. Horizon line has great importance in aviation, navigation, or even in art.

2.2 State-of-the-art Approaches

Detecting Vanishing Points using Global Image Context in a Non-Manhattan World

This method[21] is used for detecting the vanishing points and the horizon line in man-made environments. While other methods first seek possible vanishing points and then remove the outliers, this method reverses the process. Furthermore, unlike other methods, it does not make a Manhattan-world assumption[4], which means that all surfaces are aligned with three dominant directions (X, Y, and Z axes). Due to this, it can analyze even scenes with only a single vanishing point.

Global Image Context

The method first uses a deep convolutional neural network to extract global image context. Horizon priors are extracted from the global image context.

The horizon line is parametrized by two values:

- slope angle
- offset

Popular architecture AlexNet[13] was used, with some modifications. General AlexNet architecture is described in section 3.2. The first five convolutional layers from the original architecture remain unchanged. Fully connected layers are removed and replaced by two disjoined sets of fully connected layers, one for slope angle and the other for offset.Both

horizon line parameters are converted into categorical labels by dividing their domains into bins.

The network is trained using stochastic gradient descent with a multinomial logistic loss function. The learning rates are progressively increased.The network outputs a categorical probability distribution for the slope and offset.

A training database for the model was created by downloading a large number of panoramas from the Google Street View service.

After the priors from the global image context, the algorithm consists of the following steps:

1. detect zenith VP - firstly, an initial set of line segments is chosen from the global image context using the zenith direction. The RANSAC algorithm is used to refine it and handle the presence of outlier line segments.

2. detect horizontal VPs - at first, a set of horizon line candidates that are perpendicular to the zenith direction is chosen. For each candidate, a set of horizontal vanishing points is identified. It is done by selecting points along the horizon line where a lot of line segments intersect. The assumption is that if the horizon line is true, the horizontal VPs will be close to many intersection points and those intersections will be more clustered as opposed to non-horizon lines.

3. score horizon line candidates with horizontal VPs - to each horizon line candidate, a score is assigned based on the total consistency of lines with selected VPs.

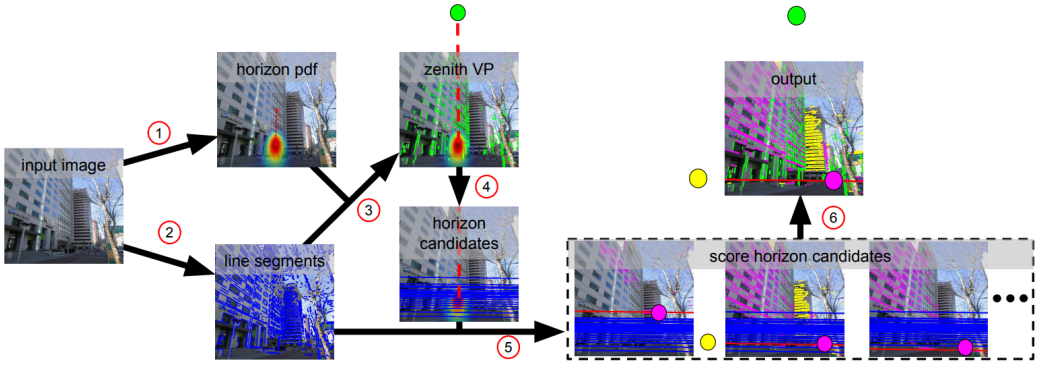

Figure 2.1: Principle of Detecting Vanishing Points using Global Image Context, taken from [21]

Horizon Lines in the Wild

The method proposed in [19] takes a learning-based approach and applies convolutional neural networks to directly estimate the horizon line. It does not rely on explicit geometric constraints like vanishing points. It can be used either in isolation or in conjunction with some geometric approach. While other horizon detection methods already used CNN, at that time none used it to directly estimate horizon line.

This method also introduced a new dataset which is now used as one of the benchmark datasets. This dataset is further described in section 4.2.

In order to automatically label images in the dataset, authors have used the structure from motion (SfM) model similar to the one used in [9]. These models mostly consist of

tourist landmarks located in urban areas. They do not contain many images of, for instance, residential streets or forests, which would create a bias. This was solved by adding such images to the dataset using panoramas downloaded from the Google Street View service.

Horizon Line Estimation using CNN

The cost of using CNN to estimate the horizon line only depends on the size of the image. GoogleNet architecture[17] was used as a basis due to the fact it requires fewer parameters. The general structure of GoogleNet architecture is described in section 3.3. There are two CNN variants - one uses a classification approach and the other with a regression approach.

The horizon line is described here either by:

1. slope and offset
2. vertical offset on the left and right side of the image

The network was implemented using the Caffe[7] deep learning toolbox.

Classification Approach

Horizon line estimation is framed here as a classification problem. The output of a CNN is then a probability distribution over categories. For each of the parameters are generated bins.

The standard GoogleNet architecture is adapted by:

1. duplicating each softmax classifier to occur once for each parameter
2. modifying the fully connected layer for each softmax classifier to output a vector with dimensions corresponding to the number of bins

Regression Approach

This approach is more challenging than classification. The default architecture is adapted by:

1. replacing each softmax classifier with a regressor - once for each parameter
2. modifying the corresponding fully connected layer to output a scalar value

Only using a regression objective did not achieve as good results as classification. However, this could be fixed by changing the initialization strategy.

Figure 2.2: Example of predicted and true horizon lines, taken from [19]

A-Contrario Horizon-First Vanishing Point Detection Using Second-Order Grouping Laws

The method[16] proposes horizon line detection in man-made environments. In such environments, the horizon line can be hypothesized based on a-contrario (meaning from the opposite) detections of second-order grouping events. A vanishing point is classified as a second-order grouping event, or a second-order gestalt according to the Gestalt theory of perception[6]. This way horizontal vanishing points on that line are extracted.

The method allows detecting the zenith and all horizontal vanishing points. Furthermore, it automatically associates a Manhattan frame to the scene. This allows calculating the camera focal length and camera orientation. Lastly warp is added to an image in order to make all vertical planes in the image appear as they would in a frontal view. If the Manhattan frame is found, the image proportions are changed to appear as in the real world.

This method avoids computationally expensive processes changing the 2D search of useful vanishing points to three separate 1D searches of zenith line, horizon line, and VPs.

Chapter 3

Convolution Neural Networks and their Architectures

When dealing with problems such as image classification or recognition, convolutional neural networks can be used, as they are most suitable for such tasks. Convolutional neural networks (CNNs) are a special Deep learning algorithm.

3.1 Convolutional Neural Networks

They take an image as an input. The CNN aims to transform a raw image into a form that is easier to process by computer. The neurons in the early layer will extract local visual features and then neurons in later layers will combine those features to form higher-order features.[8] In face of a face recognition task, the first layer might look for lines at a specific angle, those lines would then combine to extract corners and later the previous features would combine into parts of the face like the eye.

Layers

CNN consists of an input, output layer, and multiple hidden layers in between. As their name suggests, at least one of the hidden layers is a convolutional layer. Apart from convolutional layers, there are also other possible layers such as the pooling layer.

Input Layer

The input to a CNN is a tensor with a shape defined as:

$$shape = number\,of\,images \times image\,height \times image\,width \times input\,channels$$

Convolutional Layer

The convolutional layer is based around learnable filters or kernels, which are small in size. The filter is convolved across the input volume and computes the dot product between the entries of the filter and the input This produces a 2D activation map of a filter. This way the network learns kernels that activate when they see a specific feature at a given position of the input.[15]

Convolution can be expressed by the following formula[10]:

$$f_l^k(p,q) = \sum_c \sum_{x,y} i_c(x,y) \cdot e_l^k(u,v) \tag{3.1}$$

where $i_c(x,y)$ is an input image element, that is element wise multiplied by $e_l^k(u,v)$ index of the k^th kernel k_l of the l^th layer.

Convolutional layers are mainly defined by following parameters:

- the filter/kernel - represented as m x n matrix
- the stride - defines how we slide the filter across, bigger strides reduces the overlapping and the output size
- zero-padding - adding zeroes on the input image border

Each of these parameters will alter the final output size, which can be calculated as:

$$\frac{(V-R)+2Z}{S+1} \tag{3.2}$$

Where V represents input dimensions, R is receptive field size (kernel size), Z equals zero-padding, and S is stride.

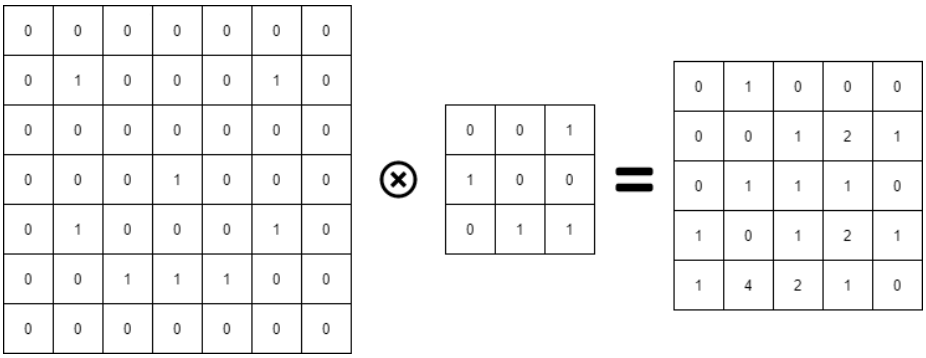

Figure 3.1: Convolution applied on an image with zero padding

Pooling Layer

The purpose of the pooling layers is to reduce the dimensionality which will reduce the number of parameters needed and the computational complexity of the model.[15] Pooling layers operate over the activation map and uses the chosen function to scale down the input. In a CNN architecture, a common practice is to insert a pooling layer between the convolutional layers.

Most CNNs use a max-pooling layer that has the kernel size of 2 x 2 and uses a stride of 2. This scales down the activation map to 25% of the original size. Max-pooling uses the max function - reports the maximal values.

Apart from max-pooling, there is also overlapping pooling, where the kernel size is bigger than the stride and thus overlaps, or the general pooling which reports the average values.

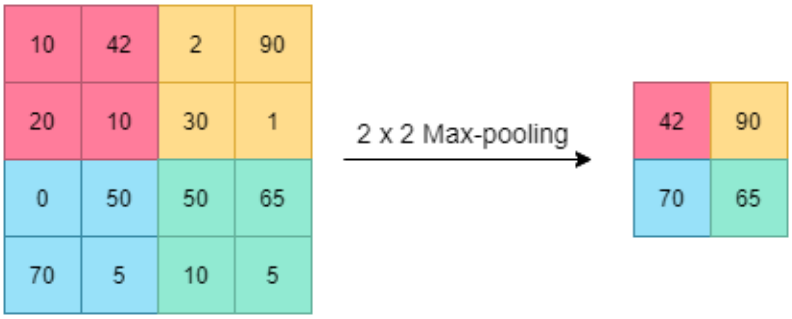

Figure 3.2: 2×2 Max-pooling

Fully-Connected Layer

A fully connected layer provides high-level reasoning and occurs after several convolutional and pooling layers. It attempts to produce class scores from the activations, to be used for classification.[15] Neurons in a fully connected layer have connections to all activations in the previous layer.

Activation Functions

The activation function defines the output of that node in the network depending on the input. The most used activation functions are:

- Sigmoid[14] - sigmoid is a non-linear activation function that is computed as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

- The rectified linear unit (ReLU)[14] - ReLU performs a threshold operation to each input element where values less than zero are set to zero thus the ReLU is given by

$$f(x) = max(0, x) = \begin{cases} x_i & if\ x_i \geq 0 \\ 0 & if\ x_i < 0 \end{cases} \tag{3.4}$$

- Softmax[14] - it is used to calculate probability distribution from a vector of real numbers. The softmax function is calculated as:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.5}$$

  The difference between sigmoid and softmax is that sigmoid is used for binary classification as opposed to the softmax which is used for multiple class classification.

Loss Functions

The loss function or the cost function is a function used to evaluate a candidate solution (the set of weights). It penalizes the deviation between the predicted and true labels during the model training. Loss functions can be divided into multiple categories:

- regression loss functions - model predicts a real value

- binary classification loss functions - model predicts between only two classes
- multi-class classification loss functions - model predicts between multiple classes

Multi-class classification loss functions include, for instance, categorical cross entropy or sparse categorical cross entropy.

3.2 AlexNet

AlexNet is a convolutional neural network that competed in a won the ImageNet challenge[1] in 2012. It achieved an error of 15.3%, which was more than 10.8% lower than the runner up. The paper publishing the results is considered to be among the most influential in the computer vision field as it has influenced more papers to use CNNs.

AlexNet architecture consists of 8 layers[13] - 5 convolutional layers and 3 fully-connected layers. It brought new approaches to convolutional neural networks by using:

- ReLU activation function - before AlexNet, the tanh activation function was the standard. However, using the ReLU resulted in significantly lower training time.
- multiple GPUs - by splitting the model's neurons between two GPUs the training time was reduced and a bigger network could have been trained.
- overlapping pooling - introducing the overlapping pooling saw an error reduction and also helped to prevent overfitting.

To reduce overfitting, the authors resorted to data augmentation to make their data more varied. They have used image translations and horizontal reflections, which greatly increased the size of the training set. They also changed the intensities of RGB channels.

Apart from data augmentations, they used the technique called dropout, where every iteration uses a different sample of the model's parameters. This forces neurons to have more robust features. The downside is the increase in the training time.

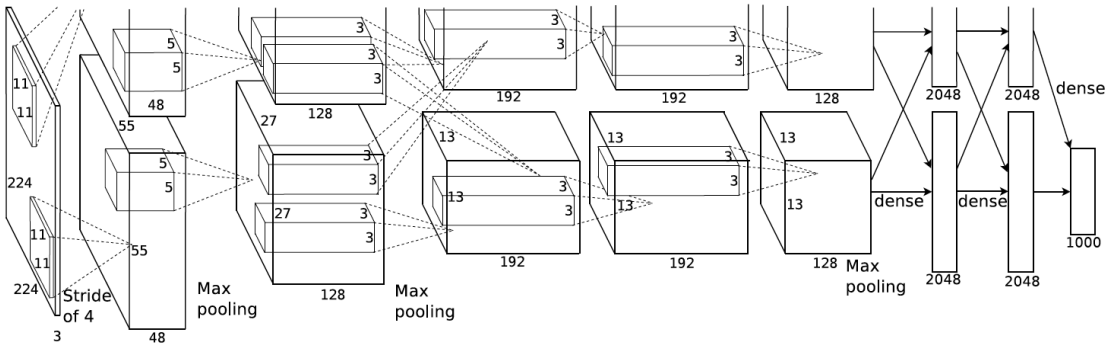

Figure 3.3: AlexNet architecture[13]

[1] http://www.image-net.org/challenges/LSVRC/

3.3 GoogLeNet

GoogLeNet is a deep convolutional neural network architecture that is a variant of the Inception Network developed by researchers at Google. It was first presented at the ImageNet challenge in 2014.[17] GoogLeNet is now a staple architecture within most common machine learning libraries such as TensorFlow, Keras, or PyTorch.

GoogLeNet architecture was designed to have higher computational efficiency than some of its predecessors. It achieves this by reducing the input image size, while still retaining important spatial information. The input layer takes in an image of 224 x 224 pixels.

The GoogLeNet architecture consists of 27 layers including the pooling layers. Part of the layers is 9 inception modules. An inception module can be seen in 3.4.

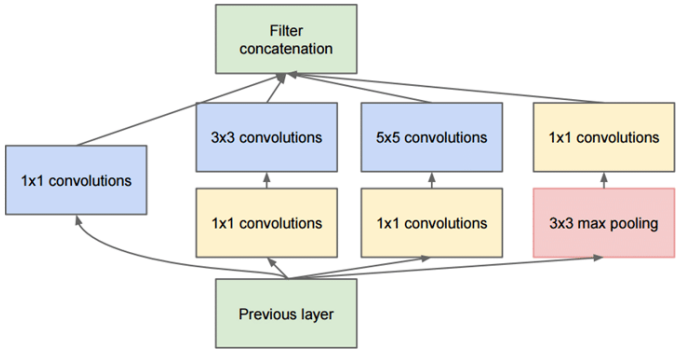

Figure 3.4: GoogLeNet inception module[17]

Networks with too many deep layers may face the problem of overfitting. Authors of GoogLeNet solve this by having filters with multiple sizes that can operate on the same level. This way the network becomes wider instead of deeper.

The convolution is performed with filters of sizes 1x1, 3x3, and 5x5. Max-pooling is performed alongside convolutions to be sent into the next inception module. Due to how neural networks are time-consuming to train, the number of input channels is limited to one by adding another 1x1 convolution before the 3x3 and 5x5 convolutions.

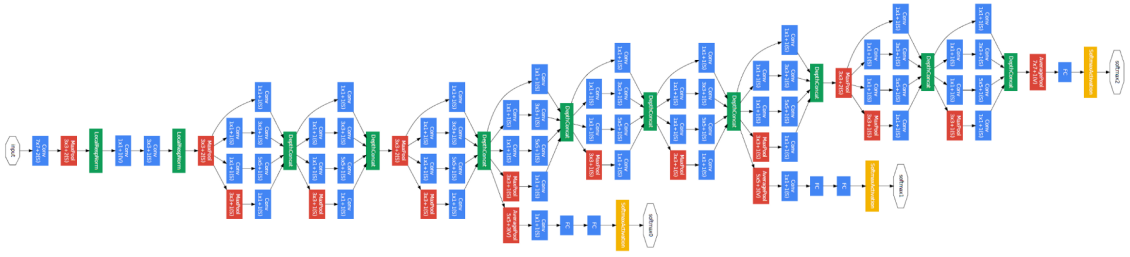

Figure 3.5: GoogLeNet architecture[17]

3.4 InceptionV3

Inception v3 is a convolutional neural network architecture that was first introduced in [18]. This architecture builds upon the previous iteration Inception V2 and improves it. It focuses on image analysis and object detection.

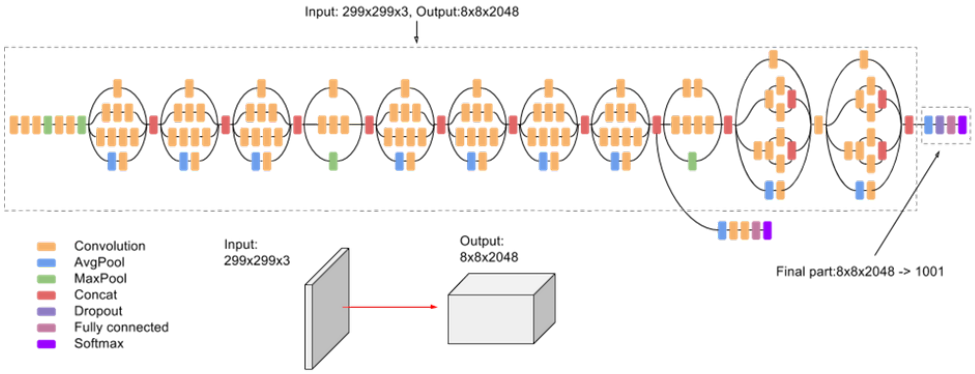

Figure 3.6: Inception v3 architecture - source: https://paperswithcode.com/method/inception-v3

Chapter 4

Evaluation Metrics and Existing Datasets

4.1 Metrics

New methods for the horizon line estimation are evaluated on three existing benchmark datasets which are discussed in section 4.2 - the York Urban Dataset, Eurasian Cities Dataset, and Horizon Lines in the Wild.

Horizon detection error is defined as the distance from the detected horizon to the ground-truth horizon line and then normalized by the image height.[21] Traditionally, a cumulative histogram of errors is shown along with information about the size of the area under the curve (AUC).

4.2 Datasets

There exists only a small number of publicly available datasets with annotated horizon lines. Usually, creators of horizon detection methods in their papers evaluate their implementation using three datasets:

- York Urban Dataset[1]
- Eurasian Cities Dataset[2]
- Horizon Lines in the Wild[3]

York Urban Dataset

York Urban Dataset (YUD), which was first introduced in [5] contains a total of 102 photos, out of which 45 are of indoor environments and 57 are of outdoor urban environments. The images were taken around York University and Toronto, Canada, which reflects in its name. All the images were hand-labeled. Each image has a resolution of 640 x 480 pixels.

[1] https://www.elderlab.yorku.ca/resources/york-urban-line-segment-database-information/
[2] http://graphics.cs.msu.ru/en/research/projects/msr/geometry
[3] http://mvrl.cs.uky.edu/datasets/hlw/

Figure 4.1: Example of images in York Urban Dataset, taken from [5]

Eurasian Cities Dataset

The Eurasian Cities Dataset (ECD) is another one of the benchmark datasets for horizon line estimation. It was first introduced in [2]. Unlike the YUD, it contains many non „Manhattan“ worlds.

Figure 4.2: Example of images in Eurasian Cities Dataset, taken from [2]

Horizon Lines in the Wild

The Horizon Lines in the Wild (HLW) was first used in [19].

The HLW contains a significantly more significant number of annotated images. However, there is a significant number of mislabeled images in HLW, which might be one reason why all methods perform significantly worse on this dataset. Its size is another factor, the original version of the dataset totals around 13GB, while the newer version contains 69GB of data. Unlike previous entries, this dataset is split between the image for training models and images withheld for test and validation.

Figure 4.3: Example of images in Horizon Lines in the Wild dataset, taken from [19]

Chapter 5

Design of Horizon Detector

This thesis aims to create a method capable of detecting the horizon line in an image using deep learning. This chapter describes the approach to designing a convolutional neural network and creating a training dataset for such a network.

5.1 Creating a Training Dataset

To train a convolutional neural network capable of detecting a horizon line in the image, it was necessary to create a training image dataset. The two existing datasets, the York Urban and Eurasian Cities datasets, are very small, containing only around 100 images. Furthermore, they are both used as benchmark datasets.

The only other publicly available dataset is the Horizon Lines in the Wild dataset. While this dataset is considerably more extensive, it is not ideal to use it as the sole training dataset. The training portion of the dataset contains only about 16000 images. Additionally, there are instances of incorrectly labeled images. However, after filtering outliers, this training dataset can be used in combination with our own.

The final training dataset should contain a variety of images from both urban and more rural locations. Each picture needs to have annotated position of the horizon line. There are several ways of defining the horizon line, ranging from parametric, normal, or slope-intercept form.

Our annotation method was based on the slope-intercept form. One of the parameters of the line is its slope or angle. The second parameter is the distance of the line from the center of the image. The distance from the bottom left corner of the image to the point of interception of the line with the left image border was also considered a potential parameter instead of the distance from the center. However, this was deemed less intuitive and would be more challenging to learn by the neural network.

Used services

The process of creating the dataset was inspired by [21], where they used the Google Street View[1] service, which has an API that allows users to download images from around the world using either city names or specific latitude and longitude. While the service is no longer free ever since it changed its pricing model, it provides a 200$ free monthly credit for its users. Each user is assigned a unique key signature to be used in all requests.

[1] https://developers.google.com/maps/documentation/streetview/overview

Most importantly, the API allows several parameters when downloading images. Apart from specifying the exact location using latitude and longitude, users have the option to specify, among others, the following parameters:

- `heading` - heading direction of the camera in degrees
- `fov` - horizontal field of view of the image
- `pitch` - up or down angle of the camera relative to the Street View vehicle

Those are the most critical parameters because they are needed to calculate the horizon's position in the image. Other parameters such as `source` can limit results to only outdoor images. However, for our purposes, indoor images are, too, downloaded since benchmark datasets also include several examples of indoor imagery.

The OpenCage Geocoding API[2] is a service that can convert city names to latitude and longitude coordinates and vice versa. When converting text name to coordinates, it returns coordinates of the city center.

Approach to downloading images

Firstly, we have created a list of cities from which we intend to download images for the dataset. Each city on the list also needs its country in order to avoid possible confusion. We have chosen cities based on their population and size. The reasoning behind this is that those cities typically have a denser infrastructure with various types of buildings. Such metropolitan cities on the list are Tokyo, New York, or London. Apart from those large cities, several smaller cities were also included to provide more rural imagery.

For the training part of the dataset, the list contained 51 cities worldwide, except for Australia. The purpose of this was to separate training data from validation data. The list of cities for validation contained eight cities from around Australia, such as Sydney or Melbourne.

We created a separate script for downloading image data. It downloads a defined number of photos from the chosen city. Cities from our created lists are fed to the downloaded script. The images are in .jpg format, and the default size is 400 x 400 px. By using the square ratio, we significantly simplify all necessary equations.

The `opencage.geocoder` Python library, which implements the OpenCage Geocoder API, converts the city name with its country to latitude and longitude coordinates of the city center. The program will calculate several locations in said city within a defined radius from the city center.

For our purposes, we have chosen to use a radius of 5 km, and in each city, download from 125 locations. Moreover, in each location, we download four pictures by changing the heading direction. In the ideal case, this results in 500 downloaded images per city in single program execution. However, the Google Street View often does not offer panorama at chosen coordinates and returns a blank image. To prevent downloading pointless blank images and wasting credit, we check the size of the first downloaded image from a location. If the size is below a certain threshold, we remove the image, and the program proceeds with the following random location.

To get the new latitude and longitude coordinates from where to download images, we first need to choose an angle and distance from the city center. The distance ranges from

[2] https://opencagedata.com/

0 to our previously chosen radius. The angle has to be in radians. We use a uniform distribution from both of those parameters.

We calculate the first the new latitude coordinate[1]:

$$\begin{aligned} delta\_latitude &= sin(angle) * distance/110.574 \\ new\_latitude &= city\_center\_latitude + delta\_latitude \end{aligned} \tag{5.1}$$

Using the new latitude coordinate, we calculate the longitude as:

$$\begin{aligned} delta\_longitude &= cos(angle) * distance/(111.320 * cos(new\_latitude) \\ new\_longitude &= longitude + delta\_longitude \end{aligned} \tag{5.2}$$

Once we have these coordinates, we can download images in that location using the Google Street View API. We randomly, with uniform distribution, choose values for `fov`, `heading`, and `pitch` parameters. The ranges for each parameter are based on those in [cite]:

- `fov` ∈ <40°,80°>
- `heading` ∈ <0°,360°>
- `pitch` ∈ <-30°,30°>

We include the city name, timestamp, field of view, heading, and pitch value in the image name for better organization.

After an image is downloaded, we calculate the horizon's position. Using fov and pitch values in radians, and given that we know image width and height, we calculate two points of horizon line by first determining the offset h as:

$$h = tan(pitch) * (\frac{(width/2)}{tan(fov/2)}) \tag{5.3}$$

Then the offset h is then adjusted due to the fact that when the pitch is equal to 0, the horizon line is precisely in the middle of an image:

$$h = (height/2) + h \tag{5.4}$$

Finally, we get the two points:

$$A = [0, h], B = [width, h] \tag{5.5}$$

Given two points A and B, we automatically determine the angle (which will be 0 since the image is not yet rotated) and line offset with the following:

$$\begin{aligned} slope &= \frac{B[1] - A[1]}{B[0] - A[0]} \\ angle &= arctan(slope) \\ offset &= A[1] - slope * A[0] \end{aligned} \tag{5.6}$$

We store metadata for downloaded images in a .csv file, where we write image filename, angle in radians, and offset normalized by image height.

Figure 5.1: Example of downloaded images

Rotating images

The images are not yet rotated. We rotate them all at once after finishing the downloading. From a single unrotated image, we produce three rotated images. The degree of rotation is randomly chosen with a range of <-20°,20°>. This also serves as data augmentation.

We rotate the image using the function already provided to us by the scipy library. While this image is correctly rotated, it does have black spots left by rotation. To avoid this effect, we zoom in using the smallest zoom factor possible and clip the image.

After rotating each image, the new position of the horizon is determined. After extracting angle and offset from the previously created .csv file, we determine two points on the line.

Next, we apply the same rotation on the image to the two points of the unrotated horizon line. The origin of rotation is the center of the image, and the angle is the roll. Then we use the zoom on the new points with the same factor as was used for the whole image.

Lastly, we convert the two points representing the line back into the angle and offset the same way as in [equation]. New angle and offset values are written into the new .csv file containing the information on rotated images.

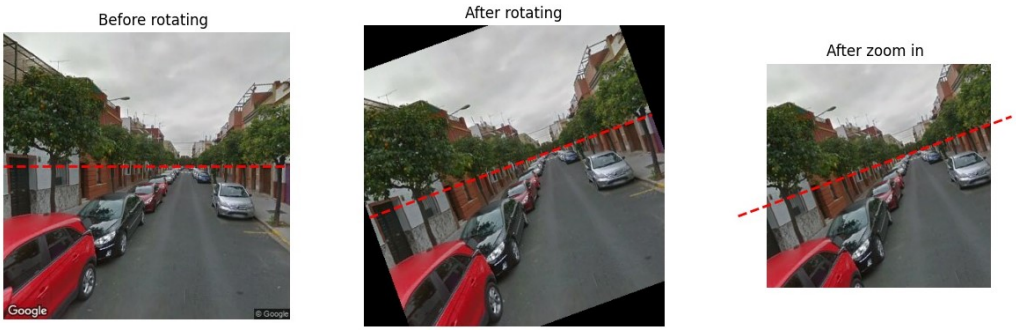

Figure 5.2: Process of preparing images for dataset

Converting offset to distance from the center

Since parametrizing horizon line with angle and distance from center was deemed more intuitive than offset from left image border, we need to convert offset to distance from the center.

Given the angle and offset from .csv file, we first compute the horizon line definition in slope-intercept form, y = k*x + q. We obtain k as tan(angle), and q is essentially the offset already. Using the slope-intercept form, we obtain two points residing on the horizon line:

$$A = [0, q], B = [width, k * width + q] \tag{5.7}$$

Afterward, we determine the center of the image as:

$$P = [width/2, height/2] \tag{5.8}$$

The final distance from the center is calculated as the cross product (B-A) × (P-A) and then divided by the vector norm of (B-A).

In total, the dataset contains 103392 files totaling approximately 2,7 GB. The training portion consists of 91263 images and the validation portion of 12126 images.

Additional training images

Training images from Horizon Lines in the Wild dataset[20] were used to increase the image dataset's size further. In total, the HLW dataset provides 16919 training photos of mostly touristic locations. However, these images are of various resolutions with non-square ratios, unlike our training dataset, where all images should be 400x400 pixels. Therefore, all these photos had to be first transformed, and the horizon line position in them had to be recalculated. We cropped the center square of each image. Afterward, this square was resized down to the final size of 400x400 pixels.

The additional images from the Horizon Lines in the Wild dataset bring the total number of training images for the neural network to 108182.

5.2 Method of detecting the horizon line

To detect horizon lines in images, I have decided to use a convolutional neural network similar to [19]. Several existing architectures for the convolutional neural network would be tested to see which would provide the best results. Those architectures will be the AlexNet(section 3.2), GoogleNet(section 3.3) and Inception V3(section 3.4) architectures.

The input of this network would be a single image. The output, however, would two separate information:

- the **angle** of the horizon line in radians
- the **distance** of the horizon line from the center of the image

In order to have the trained model output both at once and not have two separate models for each parameter, changes had to be made to the network architecture. The base structure containing the input layer and all the convolutional layers remains the same, no matter which architecture we are using.

After the final convolutional layer, the model will be split into two separate sets of fully connected layers. One branch will output predictions for the angle and the other for the distance parameter of the horizon line.

After some thought, we decided to treat the task of horizon line detection as a classification problem instead of a regression problem. The model, therefore, will not output the exact numerical values but the class to which it belongs. Using the classification approach instead of regression also helps to deal with outliers. Normally, in cases like the imagenet challenge [3], the classes are categories such as „cat“ or „dog“. Instead, we define a possible range for each parameter and the number of classes within this range. The real value is then transformed into the classes using the following formula:

$$\begin{aligned} step &= \frac{range\_length}{number\_of\_classes} \\ class &= \frac{parameter\_value + (range\_length * 0.5)}{step} \end{aligned} \tag{5.9}$$

For the angle of the horizon line, the range was chosen to be <-45°,45°>, or <-$\frac{\pi}{4}$, $\frac{\pi}{4}$> in radians. This range was chosen due to the nature of our training dataset. For the distance from the center, the range was chosen to be <-500,500> px. This was chosen through a trial and error approach to see which range would provide the best results. The distance range is intentionally more extensive than the input image size to accommodate possible instances where the horizon line is way too high or low to be directly in the picture.

The number of classes was chosen to be 100. This number, once again, was done by testing which value returned the best results. The number of classes had to be big enough so that the deviation between actual value and value obtained by reversing the formula for getting the class was not significant.

5.3 Evaluation method

The trained models will be evaluated on the Eurasian Cities dataset (section 4.2) and York Urban dataset (section 4.2). Both these datasets have horizon position data stored in the form of three parameters a, b and c for the normal form where $a * x + b * y + c = 0$. We transform this form into two points residing on the horizon line and image borders.

The trained models will require the input images to be of square ratio with a specific size. Nevertheless, the images in those two benchmark datasets all have different sizes and are not always square. Therefore we will first extract the square image from the center and then resize it to the correct size. We also need to recalculate the horizon position at this point.

The model will predict the angle and distance from the center, from which we get points where the horizon line crosses the left and right border of the image. We use the metric for horizon detection error described in section 4.1 where we get distance from the actual horizon as was annotated and predicted. We do this by subtracting y values of points intersecting the left image border and normalizing it by image height. Each image will follow a similar process. In the end, the AUC score will be calculated.

[3] https://www.image-net.org/challenges/LSVRC/

Chapter 6

Implementation

This chapter follows the chapter Design. This chapter describes tools, libraries, and services used. The most important aspects of implementing CNN for horizon detection are addressed in detail. Chapter Experiments and Evaluation follow the chapter.

6.1 Tools used

For the implementation of the CNN and subsequent evaluation scripts, the Python programming language was used, with the following libraries and packages:

- *Keras*[1] and *Tensorflow*[2] the deep learning frameworks
- *Numpy*[3] used for math operations
- *Pandas*[4] for working with dataset metadata
- *Matplotlib*[5] to visualize the results of testing trained models on benchmark datasets

Google Colababoratory[6] was used to implement and train the models and the environment for the subsequent evaluation of trained models on the benchmark datasets. It provides its users with free access to GPUs and removes the need to train on local hardware. Furthermore, it allows mounting user's *Google Drive* and accessing data there directly. Because of that, all datasets used in both training and evaluation were stored on Google Drive.

6.2 CNN implementation

Data preparation

Before the model could be build and training started, the input data and its labels had to be prepared to be in the correct format.

[1] https://keras.io/
[2] https://www.tensorflow.org/
[3] https://numpy.org/
[4] https://pandas.pydata.org/
[5] https://matplotlib.org/
[6] https://colab.research.google.com

This involved resizing each image from the original 400x400 px down to the size requested by specific architecture for the image data. The AlexNet architecture expects a 227x227 px image as input, GoogleNet expects a 224x224 px, and finally, Inception V3 architecture expects images to be of size 299x299 px.

Image data also needed to be properly normalized. Pre-trained models available in Keras API[footnote] provide their preprocessing functions. It was necessary for architectures not already available in Keras API to define preprocessing function from scratch. In this function, the image loaded as a Numpy array was first divided by 255 to get all image pixels within range <0,1>. Afterward, the mean value from the training dataset, which was calculated beforehand, was subtracted from the image. Lastly, the image was divided by the value of standard deviation from all training images.

The labels for both train and validation data were loaded from .csv files using the Pandas library. There is a separate file for the training data and for the validation data. The .csv file contained image location, angle in radians, and distance from the center, and each of these parameters is transformed into a separate Numpy array. Around 10% of validation data is used as testing data.

Data generator

Due to the large size of the training dataset, it was impossible to load the whole dataset directly into the memory. This would involve uploading a compressed .zip file to the active Python notebook on Google Colaboratory, which would sooner run out of available time than actually finish extracting the files from the archive. Furthermore, it would have to be repeated every time the runtime was restarted.

The solution to this problem is data generators. Generators break down the problem, the large dataset, into smaller batches. The program works on a single batch and moves onto the next one after it has finished with the previous one. The Keras API provides by default a class capable of generating batches of images for our model, the `ImageDataGenerator` class.

The `ImageDataGenerator` class can generate the image batches by giving it only the dataset path in function `flow_from_directory`. For classification tasks, this function can create labels for images based on the subdirectory's name in which they are stored. However, given our dataset's directory structure is not related to its labels, we cannot use this function. Furthermore, even if the dataset structure was not an issue, the `ImageDataGenerator` class does not support well the models, which have multiple outputs. Therefore it was needed to create a custom data generator.

The custom data generator inherits the `Sequence` class to use the advantage of multiprocessing. Our generator initializes with an array of image locations and a dictionary of the corresponding labels. Most important methods are `__getitem__` and `on_epoch_end`.

The `__getitem__` returns a single batch of data provided by the `__data_generation` method. In the `__data_generation` function, the image is finally loaded from Google Drive. The image is then resized and preprocessed, either by the function provided by API or the custom method mentioned in [link].

The `on_epoch_end` method is responsible for shuffling the dataset at the end of each training epoch. Shuffling the dataset is essential to prevent the model from learning the order of the training and help the training converge faster.

Before training the model, separate generators for the training and the validation data are initialized.

Loss function

Since we decided to treat the horizon detection problem as a classification and have multiple classes, we have decided to use *sparse categorical crossenthopy* as the loss function for both of the outputs:

$$L_{CE} = -\sum_{i=1}^{n} t_i \log(p_i) \tag{6.1}$$

where the t_i are the true labels, p_i are predicted labels and n number of classes[12].

Another possible loss function to use is *categorical crossenthropy*. For our purposes, those two loss functions are practically identical, except that the *sparse categorical crossenthopy* removes the need first to transform our class labels into a one-hot vector. Instead, the labels are integers. The loss returns a probability vector for all classes.

Because our model has two separate outputs and two losses, it is necessary to define the loss weight coefficients for each loss function. Since we see the angle and distance parameters as both equally important, incorrectly estimating the horizon line, their respective loss weights are both 1.0. The final loss is calculated as the sum of angle and distance loss.

Modifying network architecture

Multiple convolutional neural network architectures were used, namely: AlexNet, GoogleNet, and Inception V3 architecture. As each of these architectures was designed to classify images in the imagenet challenge, they output only a single class prediction; it was necessary to make specific changes.

AlexNet

The Keras API does not provide AlexNet architecture as a pre-trained model, unlike Inception V3. Following architecture, as it is described in [13], we copied the input layer along with the five convolutional layers. The input layer expected image shape of 227 x 227 px.

The last convolutional layer was followed by one final Pooling layer, and the model split into two separate branches. Each branch contained a Dropout layer to reduce overfitting and a Dense layer with a softmax activation function.

GoogleNet

Like the AlexNet architecture, the GoogleNet is also not implemented as a pre-trained model in Keras API. Following the same procedure as before, we implemented the architecture described in [17] up until the last convolutional layer. This architecture required input images to be 224 x 224 px.

Similarly, after the last convolution, a pooling layer is added, and the model branches out. Both branches contain a Dropout layer and a final Dense layer with a softmax—one branch outputs prediction for the angle parameter, the second for the distance parameter.

Inception V3

The Inception V3 architecture is included in Keras library[7]; hence, there is no need to implement it from scratch. It also offers the possibility of initializing the model with

[7] https://keras.io/api/applications/inceptionv3/

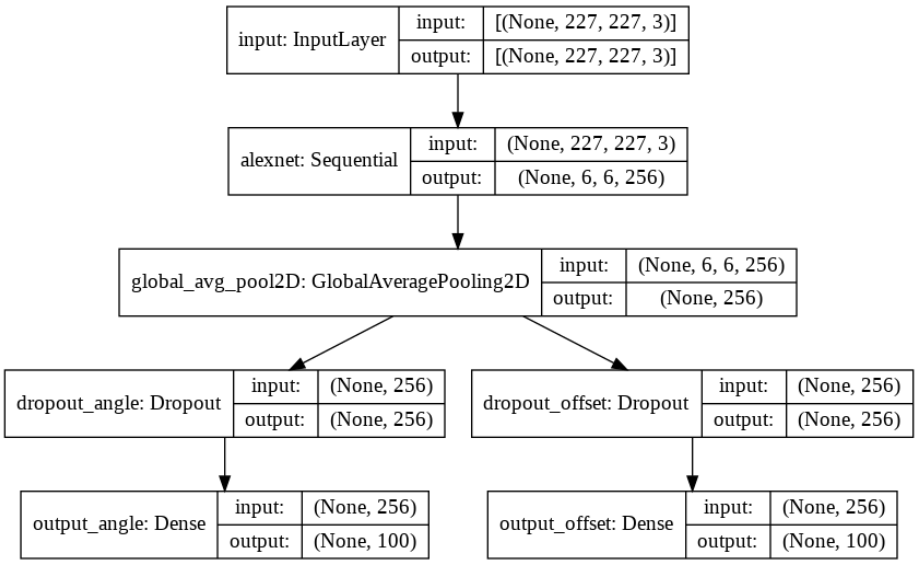

Figure 6.1: AlexNet based model structure visualized

weights from the imagenet challenge. We later experiment with both initializing weights randomly and using the imagenet weights.

This architecture works with images of size 299 x 299 px. We do not include the top part of the model, meaning the fully connected layers. To the base model is attached a Pooling layer, then the model splits into two branches. The branches contain multiple fully connected layers followed until the last Dense layer with softmax activation.

Training

All training was done on the Google Colaboratory platform with datasets stored on Google Drive. Three different architectures were trained, and one of them, the Inception V3 architecture, was trained with both imagenet initialization and random initialization.

The models based on the AlexNet, GoogleNet and Inception V3 with random weights initialization were trained for 25 epochs. The model based on Inception V3 with imagenet weight initialization was trained with a transfer-learning approach. All layers except the final added layers were frozen, and the model was trained for a small number of epochs. Afterward, the layers were unfrozen, and the model was trained for entire 25 epochs.

All model was trained with Adam optimizer[11], the learning rate of 1e-3 and batch size of 32. The training dataset was shuffled after each epoch.

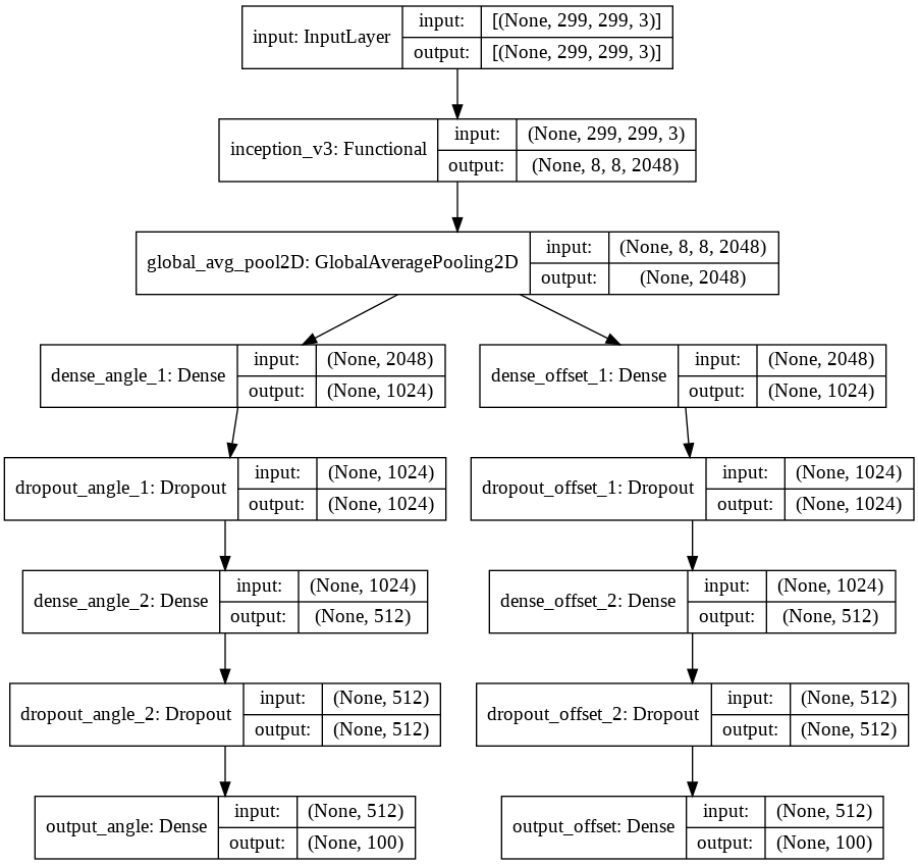

Figure 6.2: Inception V3 based model structure visualized

Chapter 7

Evaluation

This chapter contains details about trained models and their evaluation on benchmark datasets and compares them to available state-of-the-art methods. Lastly, we discuss possible improvements and future practical applications.

7.1 Trained Models

This section details the training process. We trained models described in section 6.2. All models were trained on the dataset specified in section 5.1. In the graphs below, we show a combined loss and separate losses for each parameter. Graphs Figure 7.1 show results of training the model with AlexNet architecture, graphs Figure 7.2 corresponds to the model with GoogleNet architecture, graphs Figure 7.3 present the model based on Inception V3 architecture with random weight initialization. Finally, graphs Figure 7.4 also display an Inception V3 based model, but one that was initialized with imagenet weights.

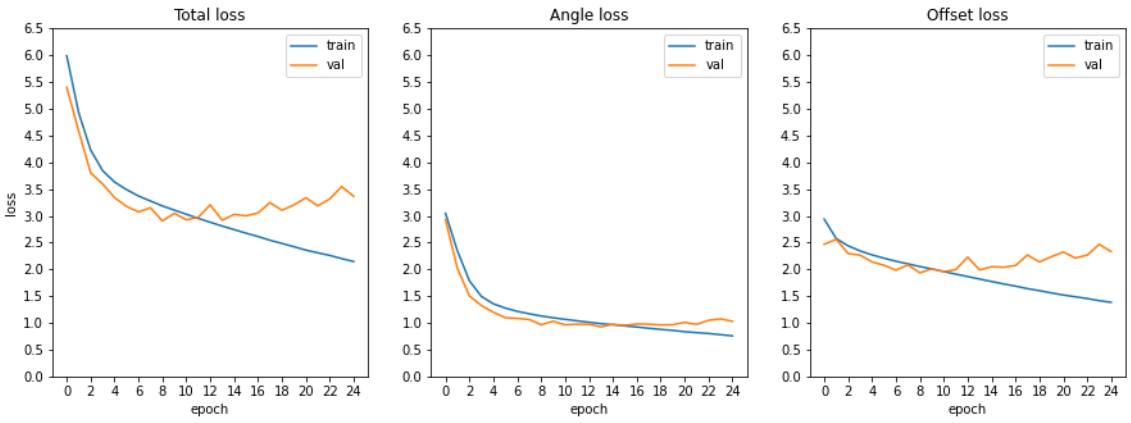

Figure 7.1: Loss on the AlexNet model

By analyzing the loss function, we see that during the training of all models occurred overfitting. This means that models focused too much on the training data, as evidenced by the fact that the validation loss stops decreasing and is higher than the training loss.

From the graph Figure 7.2, it is evident that the GoogleNet architecture was the first to stop learning. Both the AlexNet model and Inception V3 with random initialization

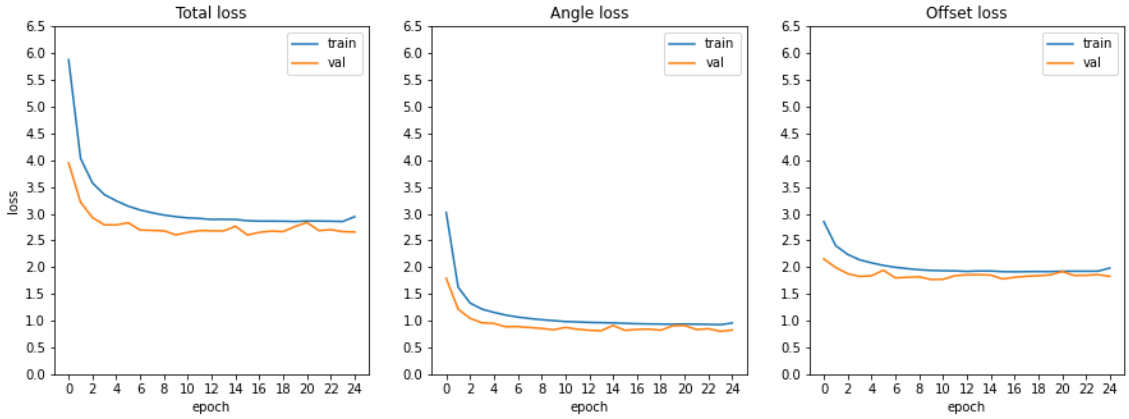

Figure 7.2: Loss on the GoogleNet model

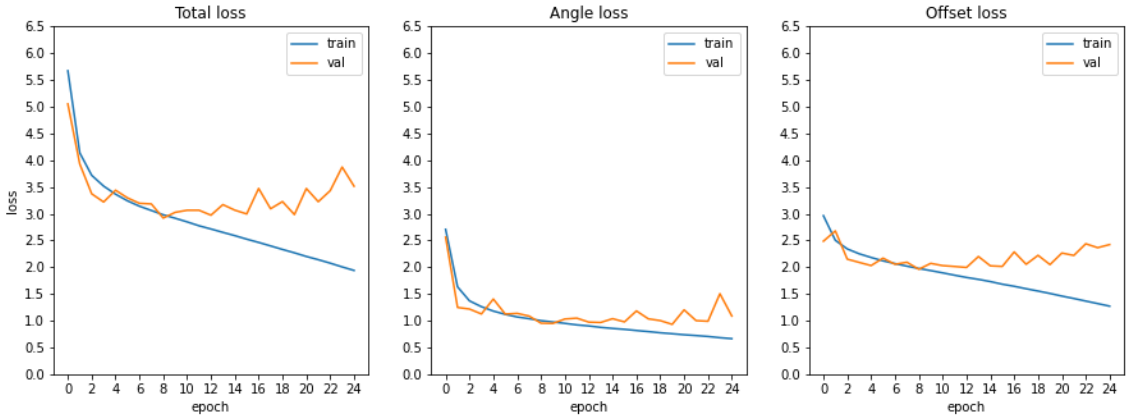

Figure 7.3: Loss on the Inception V3 model with random initialization

continued learning; however, they soon began to overfit. The Inception V3 model with imagenet initialization took more time to start learning. In the first three epochs, only the last layers were updated; the rest was frozen.

Looking at the individual losses, it is evident that the model had more problems with learning the horizon line offset or the distance from the image center than the line angle.

Judging only from the loss data during the training, the Inception V3 architecture overall yielded the best results from the trained models. The different weight initialization did not make a significant difference, except for the fact that imagenet initialization appeared more prone to overfitting.

7.2 Evaluation on Benchmark Datasets

The trained models were tested on the two benchmark datasets, the Eurasian Cities Dataset (section 4.2) and the York Urban Dataset (section 4.2). Testing our models on those datasets is a much better indicator of the model's validity. The metric used for evaluation was described in section 4.1.

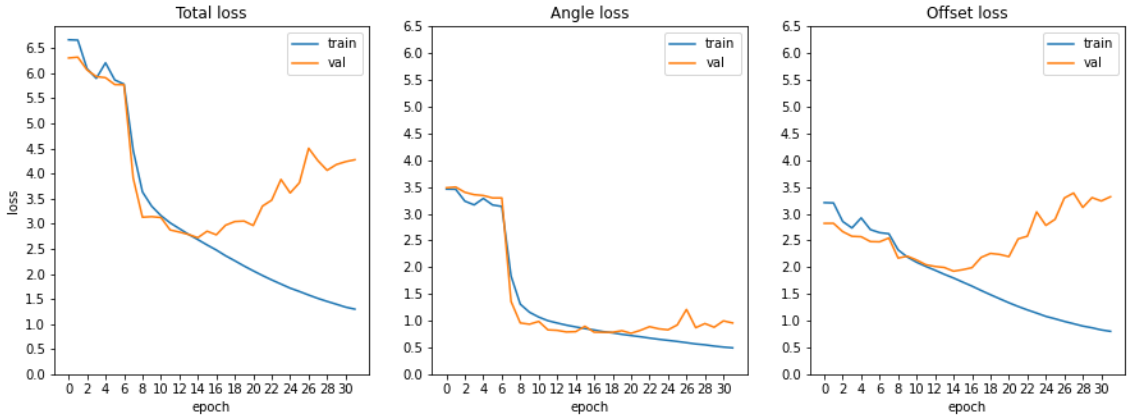

Figure 7.4: Loss on the Inception V3 model with imagenet initialization

AlexNet

The model based on AlexNet architecture scored 76.12% on Eurasian Cities Dataset and 72.23% on York Urban Dataset. This model had overall the worst results.

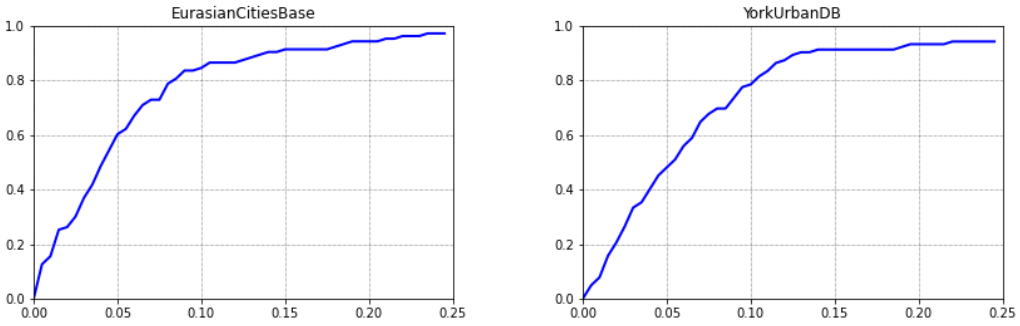

Figure 7.5: AUC of AlexNet model

GoogleNet

The model based on GoogleNet architecture scored 80.87% on Eurasian Cities Dataset and 74.09% on York Urban Dataset.

Inception V3

The models based on Inception V3 had the best results. The variant with random layer weights initialization scored slightly better on Eurasian Cities Dataset with a score of 82.20% but had worse results on York Urban Dataset with a score of 83.63%. The other variant with imagenet layer weights initialization scored 81.48% on Eurasian Cities Dataset and performed better on York Urban Dataset with 86.39%.

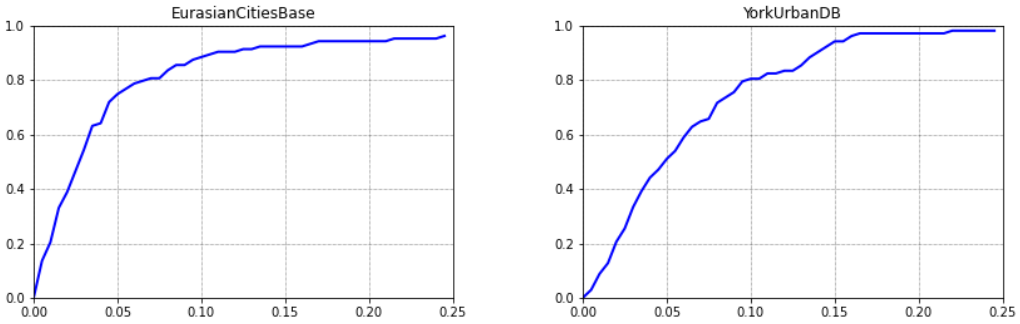

Figure 7.6: AUC of GoogleNet model

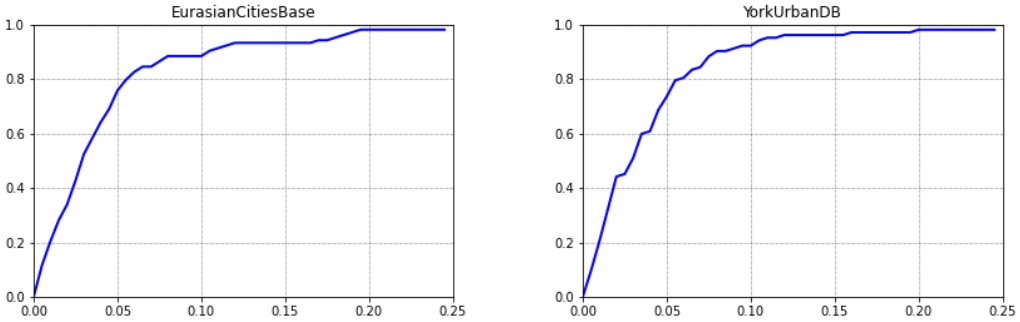

Figure 7.7: AUC of Inception V3 model - random initialization

7.3 Comparison to existing methods and future improvements

Compared to existing methods, our trained models yielded worse results. However, the score, particularly by the Inception V3 based models, was not low enough for them to be considered entirely unusable. Table 7.1 shows how specifically how our models scored compared to existing methods:

There is room for improvement in our approach. The most pressing issue is overfitting. One way to reduce overfitting would be to increase the size of training data [3]. Additional data would be downloaded from the Google Street View service. We could manually anno-

Method	YUD	ECD
AlexNet	72.23%	76.12%
GoogleNet	74.09%	80.87%
Inception V3 - random init	83.63%	**82.20%**
Inception V3 - imagenet init	**86.39%**	81.48%
GoogleNet[19]	86.41%	83.6%
CNN+FULL[21]	94.78%	90.80%

Table 7.1: Comparing AUC of existing methods with our implementation.

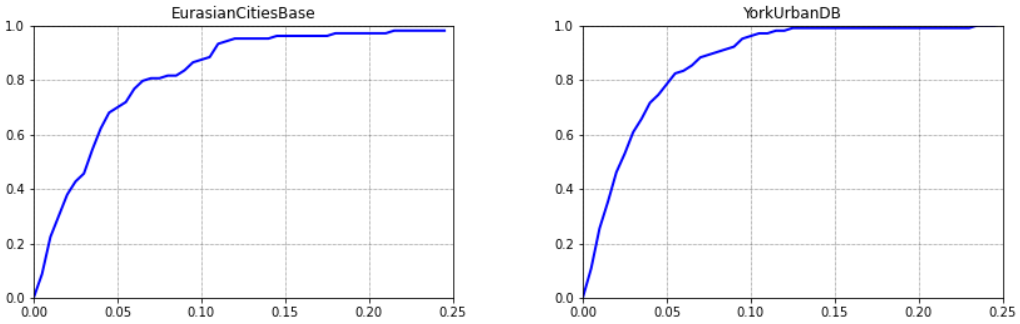

Figure 7.8: AUC of Inception V3 model - imagenet initialization

tate photographs to introduce greater variety. Alternatively, we could focus on improving the model by applying regularization, forcing the model to learn only the relevant patterns in data.

Once the model's performance improves, the trained model could be used in practical applications, for instance, a web-based application capable of automatically correcting photographs where the horizon line is skewed.

Chapter 8

Conclusion

This thesis dealt with the task of detecting the horizon line in an image using a deep learning method.

Training dataset had to be created by downloading images from Google Street View service and automatically calculating horizon position based on pitch and field of view. Images were downloaded from a combination of large metropolitan cities and small towns. Training data from already existing Horizon Lines in the Wild dataset were used as a supplement.

The horizon line can be described by two parameters, the angle and the line's distance from the center of the image. The problem was approached as a classification task where the actual values of line parameters are transformed into classes during training. When predicting, classes are transformed back into actual values. Multiple architectures for convolutional neural networks were chosen as a basis, namely the AlexNet, GoogleNet, and Inception V3 architectures. These architectures had to be adapted to have two outputs.

Trained models were evaluated on two benchmark datasets, the Eurasian Cities dataset, and York Urban dataset. The Inception V3 architecture yielded the best results. However, the trained model performed worse than the existing state-of-the-art methods, as our models were prone to overfitting.

Training models on a more extensive dataset could improve the results by reducing overfitting. Once satisfactory accuracy is obtained, the model could be used in practical applications, for example, a web application that automatically corrects uploaded images with horizon line askew.

Bibliography

[1] *Calculate distance, bearing and more between Latitude/Longitude points*. Cit. 2020-05-10]. Available at: http://www.movable-type.co.uk/scripts/latlong.html.

[2] Barinova, O., Lempitsky, V., Tretiak, E. and Kohli, P. Geometric Image Parsing in Man-Made Environments. In: *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6312, p. 57–70. Lecture Notes in Computer Science. ISBN 9783642155512.

[3] Carremans, B. *Handling overfitting in deep learning models*. August 2018. Cit. 2020-05-10]. Available at: https://towardsdatascience.com/handling-overfitting-in-deep-learning-models-c760ee047c6e.

[4] Coughlan, J. M. and Yuille, A. L. Manhattan World: Orientation and Outlier Detection by Bayesian Inference. *Neural Computation*. MIT Press. 2003, vol. 15, no. 5, p. 1063–1088. ISSN 0899-7667.

[5] Denis, P., Elder, J. H. and Estrada, F. J. Efficient Edge-Based Methods for Estimating Manhattan Frames in Urban Imagery. In: *Computer Vision – ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5303, p. 197–210. Lecture Notes in Computer Science. ISBN 9783540886853.

[6] Desolneux, A., Moisan, L. and Morel, J.-M. *From Gestalt Theory to Image Analysis: A Probabilistic Approach*. Springer-Verlag, collection "Interdisciplinary Applied Mathematics", 2008. 276 p. Available at: https://hal.archives-ouvertes.fr/hal-00259077.

[7] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J. et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *ArXiv.org*. Ithaca: Cornell University Library, arXiv.org. 2014. ISSN 2331-8422. Available at: http://search.proquest.com/docview/2084581566/.

[8] Kelleher, J. *Deep Learning*. MIT Press, 2019. MIT Press Essential Knowledge series. ISBN 9780262537551. Available at: https://books.google.sk/books?id=1wICwQEACAAJ.

[9] Kendall, A., Grimes, M. and Cipolla, R. PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2015, vol. 2015, p. 2938–2946. ISBN 9781467383912.

[10] Khan, A. and Zahoora, U. A Survey of the Recent Architectures of Deep Convolutional Neural Networks. *ArXiv.org*. Ithaca: Cornell University Library, arXiv.org. 2020, vol. 53, no. 8. ISSN 02692821. Available at: http://search.proquest.com/docview/2169187870/.

[11] Kingma, D. and Ba, L. Adam: A Method for Stochastic Optimization. *ICLR 2015*. 2015.

[12] Koech, K. E. *Cross-Entropy Loss Function*. October 2020. Cit. 2020-05-10]. Available at: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e.

[13] Krizhevsky, A., Sutskever, I. and Hinton, G. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*. ACM. 2017, vol. 60, no. 6, p. 84–90. ISSN 00010782.

[14] Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *ArXiv.org*. Ithaca: Cornell University Library, arXiv.org. 2018. ISSN 2331-8422. Available at: http://search.proquest.com/docview/2131541613/.

[15] O'Shea, K. and Nash, R. An Introduction to Convolutional Neural Networks. *ArXiv.org*. Ithaca: Cornell University Library, arXiv.org. 2015. ISSN 2331-8422. Available at: http://search.proquest.com/docview/2083864399/.

[16] Simon, G., Fond, A. and Berger, M.-O. A-Contrario Horizon-First Vanishing Point Detection Using Second-Order Grouping Laws. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. September 2018.

[17] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. et al. Going Deeper with Convolutions. 2014.

[18] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z. Rethinking the Inception Architecture for Computer Vision. 2015.

[19] Workman, S., Zhai, M. and Jacobs, N. Horizon Lines in the Wild. 2016.

[20] Workman, S., Zhai, M. and Jacobs, N. Horizon Lines in the Wild. In: *British Machine Vision Conference (BMVC)*. 2016. Acceptance rate: 39.4%.

[21] Zhai, M., Workman, S. and Jacobs, N. Detecting Vanishing Points using Global Image Context in a Non-Manhattan World. 2016.