Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Object Detection in Fine Art Photography

Bachelor thesis, spring semester 2020

Fabian Meyer

Supervisor: Dr. Simone Lionetti

Lucerne University of Applied Sciences and Arts
School of Information Technology
Bachelor in Information Technology
June 3, 2020

Bachelorarbeit an der Hochschule Luzern - Informatik

Titel: Object Detection in Fine Art Photography

Student: Fabian Meyer

Studiengang: Bachelor Informatik

Abschlussjahr: 2020

Betreuungsperson: Dr. Simone Lionetti

Experte: Roman Bachmann, Swisscom

Eidesstattliche Erklärung
Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt habe, alle verwendeten Quellen, Literatur und andere Hilfsmittel angegeben haben, wörtlich oder inhaltlich entnommene Stellen als solche kenntlich gemacht haben, das Vertraulichkeitsinteresse des Auftraggebers wahre und die Urheberrechtsbestimmungen der Fachhochschule Zentralschweiz (siehe Merkblatt Studentische Arbeiten auf MyCampus) respektieren werde.

Ort / Datum, Unterschrift: _____

Abgabe der Arbeit auf der Portfolio Datenbank

Bestätigungsvisum Studentin / Student
Ich bestätige, dass ich die Bachelorarbeit korrekt gemäss Merkblatt auf der Portfolio Datenbank abgelegt habe. Die Verantwortlichkeit sowie die Berechtigungen habe ich abgegeben, so dass ich keine Änderungen mehr vornehmen kann oder weitere Dateien hochladen kann.

Ort / Datum, Unterschrift: _____

Verdankung
Danke liebe Famile, Freunde und Bekannte. Speziellen Dank geht an Simone Lionetti für die umfassende und freundliche Unterstützung während des gesamten Projekts.

Eingangsvisum (durch das Sekretariat auszufüllen):

Rotkreuz, den _____ Visum: _____

Hinweis: Die Bachelorarbeit wurde von keinem Dozierenden nachbearbeitet. Veröffentlichungen (auch auszugsweise) sind ohne das Einverständnis der Studiengangleitung der Hochschule Luzern Informatik nicht erlaubt.

Abstract

Convolutional neural networks are a branch of deep learning models that are capable of recognising and locating objects. However, these models are trained with standard data sets, containing of everyday photography. In this work, different models from different frameworks of types object detection and instance segmentation were evaluated on a newly created dataset, consisting of 42 pictures from four different artists. Model Mask R-CNN and framework MMDetection have been chosen to work with, as they showed the best overall performance and the best usability. It was realised that the performance of the model depends to a great extent on the style of the respective artist. A web application was developed in order to detect, segment and classify objects in images and to display them in a grid. The web application has been deployed on the EnterpriseLab and is publicly reachable. In a final step, a custom model was created by retraining it with images from fine art photography and its performance was compared to the out-of-the-box Mask R-CNN model. It became obvious that the newly trained model is no longer able to recognise previously detected objects.

Contents

1. Problem, question and vision

Object detection is an important computer vision and machine learning task that consists in the localisation and classification of items within digital images. The business and industry applications of this technology are growing in number and relevance, especially because of the tremendous progress, largely driven by deep learning, that has been made in recent years.

Uses of object detection technologies include applications in security measures for example images from surveillance cameras, search engines, object tracking or counting in traffic, autonomous driving cars, robotics in general and the field of medical- and bioinformatics.

1.1. Task description

The aim of this work is to develop a web-application that is built around an object-detection model. This web-application has to take in a picture, looks for objects in it and has to return a "tidied up" image with the objects found in it. The "tidied up" image is inspired by "Kunst aufräumen", a series of fine-art photographies by Swiss artist Urs Wehrli, that consists of two images, a "messy" one and a "tidy" one. In the "tidy" image, similar objects are shown next to each other, a task that is analogue to classification in computer vision applications. One example picture from "Kunst aufräumen" is shown here:

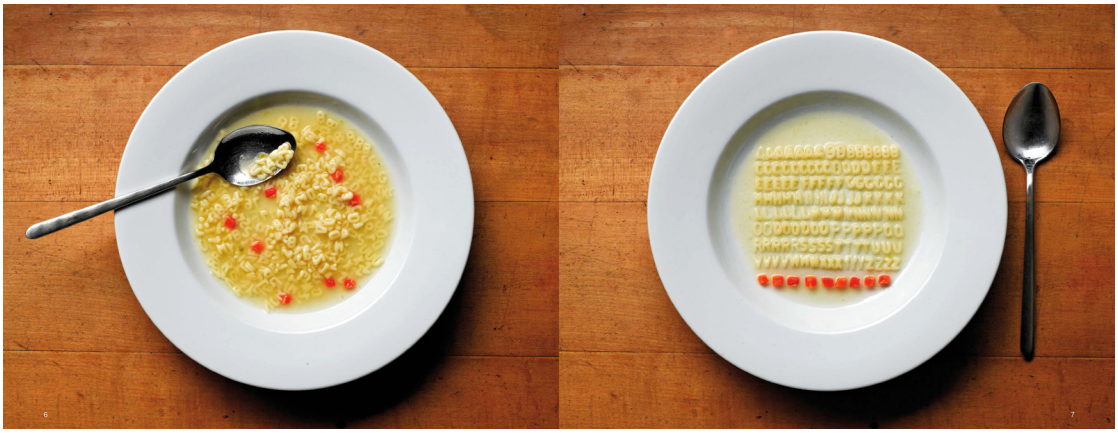


Figure 1.1.: Sample image from "Kunst aufräumen" by Urs Wehrli

In a further step, an existing object detection model has to be refined by retraining it on a dataset with images from artistic photography and the performance of the two models have to be examined. One important element in the research of object detection is computation

of scores to evaluate the quality of the model. An additional tasks was to develop an own metric to evaluate the different models when applying them to data of artistic photography.

1.2. Why fine art photography?

Models for object detection are usually trained on datasets, containing pictures that show objects in a very clear manner. These images usually contain only a few objects and are shot in natural lighting with natural colours. Composition-wise they are simple as it is the goal to depict the object in a most clear way.

Figure 1.2.: Sample image from COCO dataset

Fine art photography pictures on the other hand is in strong contrast to these images, as they are often depicting objects who do not belong together usually. These images are often shot in studios with artificial lighting and heavy post-processing or even digital manipulation. And they often contain a lot more objects in it which even can overlap and obscure each other.

Figure 1.3.: Sample image from artist David LaChapelle

The questions that now arises are, how do object detection models that are trained on traditional datasets perform on given images of artistic photography? What are the limitations of object detection models? How do they deal with extreme data material?

2. State of research and practice

2.1. A brief introduction into discrete two-dimensional convolution

Discrete convolution operation has been proved useful when applied on digital images to detect structural features like edges and corners. When applying discrete convolution to an image (a two dimensional array of values), a filter with a given two dimensional kernel is used to perform convolution on the image. With the given kernel, a segment that has the same size as the kernel is taken from the input image and an elementwise multiplication between these two matrices is performed. The sum of the results gets written in the output (also called feature map). Because of the linear nature of this operation, discrete two-dimensional convolution can be executed on a computer in s short time. [2]

As most kernels are much smaller than the input image, this inevitable results in smaller feature maps than input images, especially after applying convolution to the same image multiple times. To solve this problem, one can add a padding (increase the image size by adding specific values). An example of a two dimensional convolution operation with padding is shown here:

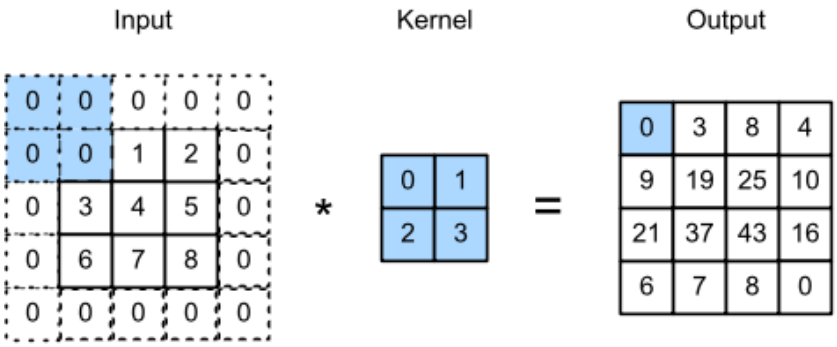


Figure 2.1.: Example of a two dimensional convolution operation with padding

After the convolution is computed for a specific segment, the kernel shifts further some steps in the image array, called stride. There exists a lot of different kernels for different purposes. To detect edges and corners there exist specific kernels especially for that kind of task. For example if one wants to detect vertical edges, a kernel like the following can be used: $P_y = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$, to detect diagonal edges, one can use a kernel like the

following: $S_d = \begin{pmatrix} -0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{pmatrix}$

It is also possible to learn the optimal kernel in the process of neural network training.

2.2. About convolutional neural networks

The model that is used for this project is from the family of convolutional neural networks or ConvNets or CNNs. These are models that are especially useful to perform operations on digital images. CNNs contain at least one convolutional layer that computes a mathematical operation that is called discrete convolution.

The advantage of CNNs compared to traditional multi layer perceptrons (MLPs) is that discrete convolutional operation is translational invariant. This means it is irrelevant where in a given image a specific object is located: The model will not learn the position of the specific object. Another huge advantage of CNNs over MLPs is that CNNs usually have sparse interactions. This means that it is possible to yield a good performance even if some adjacent layers are not fully connected with each other. Another way to speed up the training process is to use parameter sharing: Some weigths will be used at multiple places at the same time. These two techniques, sparse interactions and parameter sharing does make it possible to train very deep convolutional neural networks in a feasible amount of time. [3]

It is important to note that by applying discrete convolution on digital images, only primitive features can be extracted. However by combining convolutional layers with other layer types, it is possible to extract much more complex features that can be used to classify objects as humans or animals or vehicles and so on.

Another important kind of layer that is usually employed in CNNs are pooling layers. Also called subsampling or downsampling layers sometimes, they reduce the size of a given image, by computing a statistic metric to summarize multiple values. Some frequently used pooling layers are max- and average-pooling. The advantages of pooling layers are smaller input, parameter reduction and higher invariance to scaling and transformations. [3]

The third important kind of layer is ReLU (rectified linear unit), which computes an activation function used by the neural network. The mere use of this layer is to preprocess the image before the next convolutional operation step. This just adjusts the image brightness in a ways that all negative values (values that are darker than the middle grey of an image) got adjusted to middle grey. [3]

The last layer of a CNN contains the number of classes that the network should predict. This layer is usually fully connected to the layer before (called a dense layer).

2.3. From CNNs to Mask R-CNN

The very first CNN appeared in the year 1994, it was named LeNet5, after Yann LeCun. This fundamental work was the first network that used convolutional and pooling layers to

process images. In a time long before consumer graphic processing units (GPUs), where even CPUs were slow, it was crucial to reduce the number of parameters to a bare minimum (accomplished with sparse connect layers) [4]. An image, showing LeNet5 with its different layers and operations in between is shown here:

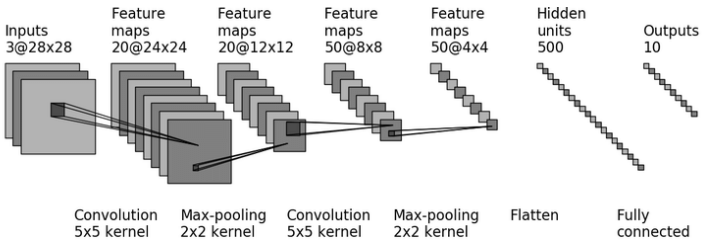


Figure 2.2.: Overview of LeNet5 from the year 1994

Starting with AlexNet by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton in 2012, CNNs gained significantly performance gains when applied on image classification tasks. AlexNet took the idea of a CNN from LeNet5 and added more layers to the network and was the first to include ReLU layers. It also introduced dropout techniques to avoid overfitting during the training process [5]. AlexNet was still an image detection or image classification model, trained to label images. This means a single label (class) is given to a whole input image.

ResNet (residual neural network) in 2015 was the first CNN that included residual blocks or identity shortcut connections. These are connections in the network that skip one or more layers and just use the input value as an output (called mathematical identity). This was another step to reduce computation cost and lead to even deeper networks [6]. ResNet is still used today as part of the backbone in a lot of object detection and instance segmentation models.

With R-CNN (regions with CNN features) in 2013, the rise of object detection models began. These researcher asked themselves, how can one use the techniques from CNNs to not only classify an image but to classify multiple objects in that image? These family of models are capable of labelling multiple objects depicted in the same image with each a bounding box and a class prediction. R-CNN models do that by proposing regions in that potential objects may lie. R-CNN is thus called a two stage model: The first stage scans the image and generates region proposals with the help of a region proposal network (RPN), whereas the second stage uses these regions to extract CNN features from it and to ultimately classify objects in it. For the classification step in the last layer, R-CNN uses a support vector machine (SVM) as a classifier. In the very last step, a regression is used to further tighten the coordinates of the bounding boxes of each object [7].

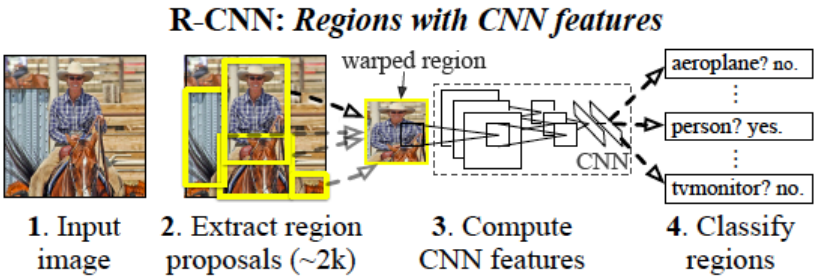


Figure 2.3.: Architecture of R-CNN

In 2015, the team from Ross Girshick, one of the creators of R-CNN, delivered Fast R-CNN an improved, faster version of it. In R-CNN there are a lot of overlapping region proposals and every computation gets calculated again, even if the regions are very similar to each other. To circumvent this they invented region of interest pooling (RoIPool). RoIPool shares these computation across the regions of an image and can speed up computation time a lot. The other improvement was to put all computations in a single network (compared to R-CNN where classification ran in a single network) [8].

Also in 2015 by the team from Ross Girshik, the second iteration of R-CNN got released, called Faster R-CNN. The main improvement was to use just one CNN that produces a single feature map for both region proposal and classification. Features used by both the first and the second stage of the model can be shared to speed up the inference process [9].

2.4. Mask R-CNN

With Mask R-CNN from 2017 (also by Ross Girshik et. al) it is possible to not only predict the bounding box of an object but also to predict a mask that shows the exact shape of the object (called pixel level segmentation). These models are called instance segmentation models, because they produce a mask for every object in the image. This is accomplished by adding a branch to the Faster R-CNN model that computes a binary mask for a given object in the image [10]. An overview of Mask R-CNN can be seen here:

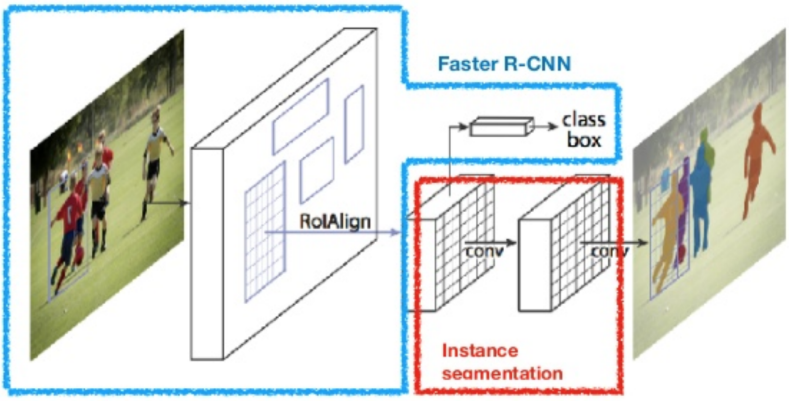


Figure 2.4.: An overview of Mask R-CNN

2.5. Feature pyramid networks

Mask R-CNN also implements a technique called feature network pyramid (FPN). As objects on images often contain very different sizes compared to each other, it is important to employ a model that is invariant to scaling of objects in a given input image. In short, an FPN generates an array of feature maps of a given input image of different sizes that are used as input for prediction. As with every step the feature map gets smaller due to pooling layers, information content increases due to convolution layers. This is called bottom-up pathway. To create a high resolution feature map containing all information content, a top-down pathway is used. Conversely to downsampling, in the top-down pathway, feature maps get upsampled. With the help of lateral connections between these two pathways, it is possible to yield both exact predictions and fast computation time simultaneously [11]. These two pathways are shown here:

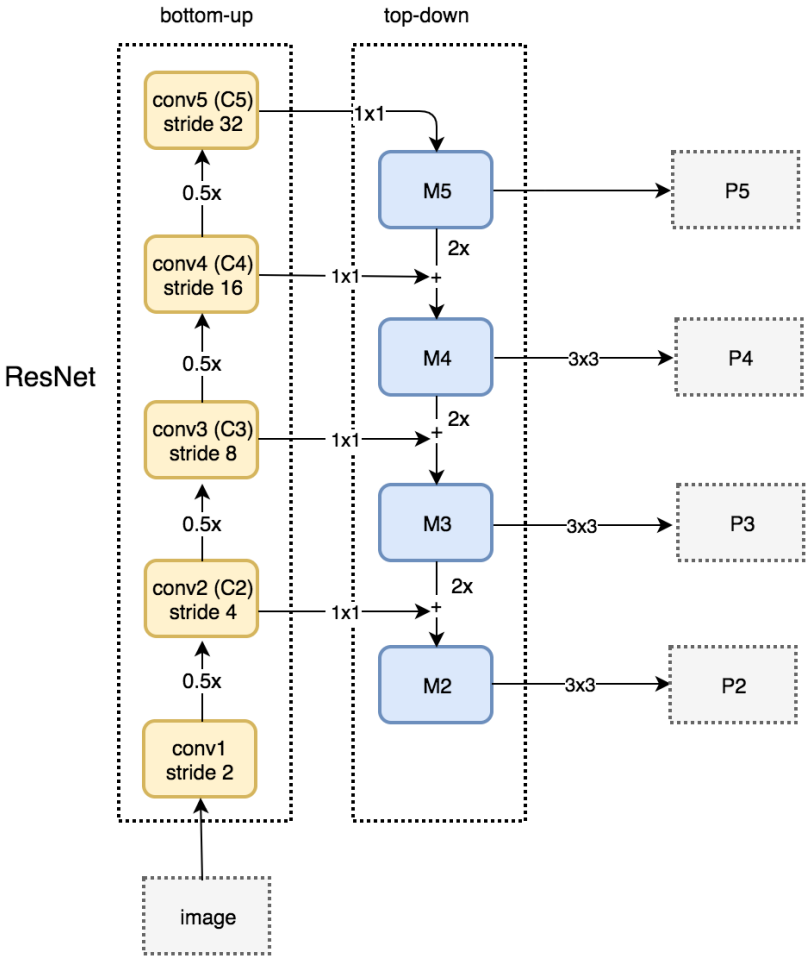


Figure 2.5.: Feature pyramid network with its two pathways

FPN along with ResNet is used in Mask R-CNN as the backbone of the network.

2.6. Overview of object recognition tasks and approaches

An overview of the different tasks and approaches and their models in the field of object recognition can be seen here:

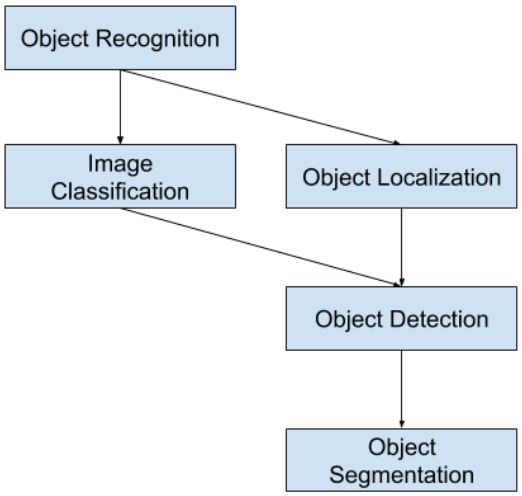


Figure 2.6.: Tasks and approaches in object recognition

As can be seen here, instance segmentation (object segmentation) is a recently invented technology based on object-detection.

3. Ideas and concepts

3.1. Basic idea

The basic idea of creating an application that takes in images and returns a repository of objects found in it is original to my best knowledge and leaves much room for personal interpretation and customization.

3.2. Object detection frameworks and models

As can be seen in the overview graphic of different object recognition tasks in figure 2.6 on page 10, there are multiple possibilities to detect objects in an image. As it gives a nicer output, an instance segmentation model was chosen in favour over an object detection model.

Another idea that came to my mind was to use an object of the family of image captioning models. Image captioning models are capable of generating a text that describes the scene depicted in an input image. These use a CNN in a first step to analyse the image, combined with a generative network to generate a text from it afterwards. Image captioning would have been especially interesting when applied to fine art photography, because many of these pictures show messages that are loaded fully with irony or socio-critical or political messages. An image captioning model thus would also output text and this idea could be reused in a future work.

3.3. Web application tools and approaches

There exist many different tools and approaches to create a web application with Python. The classic approach to accomplish this, would have been to use Python as a backend programming language used to compute the logic of the application and to use a frontend programming language like JavaScript to develop the user interface. There is also a tool available that can take in a Jupyter notebook and turn it into a running web application, called Voilà. Voilà works with any Jupyter kernel and thus can be used to develop an application from other languages than just Python.

3.4. Alternative research questions

The most obvious research quesiton would have been: What happens if a model gets trained on images by artist X and gets tested on images by artist Y, that contain the equal classes

of objects in it? It would have been interesting to find out, whether the model will learn the distinct style of the artist, like colours, contrast, camera angle, lighting and so on. Of course another model could have been trained on the corresponding images from artist Y and the two models could have been compared with each other and also with the base (pretrained) model. As this needs at least two images from different artists that contain the same objects in it, it was rejected. However with a suitable performance metric it would allow to make a concluding statement about object detection in fine art photography.

Another, more humorous task would have been to create something new with the found objects in the image. For example a fruit bowl generator algorithm, that takes in an image, searches for objects of different class of fruits and places them into a bowl and returns the new "fine art"-image. This could be accomplished with just hard-coding the features needed, or even better, with a generative model. As this is a complete different topic this project idea was rejected.

4. Methods

4.1. Project management

In the beginning of the project, a project management plan had to be developed. The project management plan contains a timeline with all (then known) tasks and a list of all milestones. Trello together with a timeline add-on (Elegantt, a gant chart tool) was used as project management tool. The project management plan and all tasks and milestones can be reached via this link: `https://trello.com/b/srqnMstX/object-detection-in-fine-art-photography`. A picture, showing the full timeline is shown here:

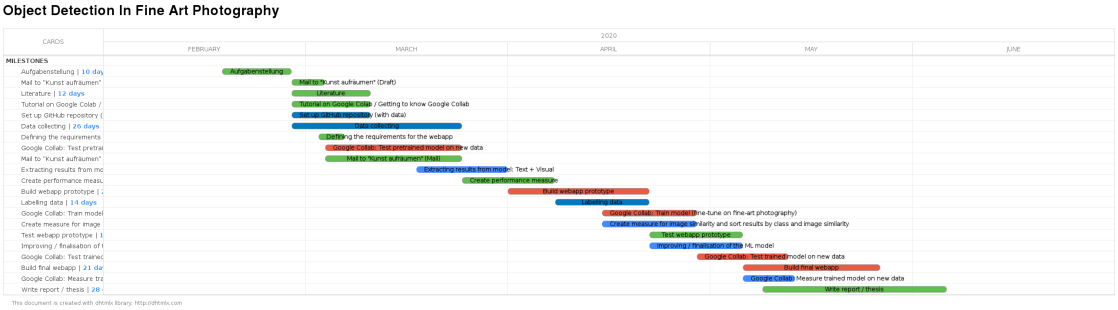


Figure 4.1.: Timeline view of the project management plan

To better structure the project, a number of milestones have been chosen:

1. Test pretrained models on Google Colab with new data: 24th March 2020
2. Build web application prototype: 28th April 2020
3. Finetune model with fine art photography data: 12th May 2020
4. Test finetuned model on new data: 19 May 2020
5. Finish web application with new model: 2nd June 2020

4.2. Toolchain

The whole project can be summarized in four single steps or tasks:

1. Data: Collect a dataset, for inference, for training and for testing
2. Model: Get to know at least one model, for inference and for retraining
3. Tidy: For a given input picture, return an image that displays a repertoire of all found objects in the input picture
4. App: Turn the former three tasks into an application that can be reached from a web browser

In order to develop a minimal viable product, all these four tasks have to be solved first. The way of proceeding was to develop a minimal viable product first and then to start with refining the model and adjusting the web application. This means going through the full circle first, before adjustments and proceeding into retraining are made.

An extra task would have been to develop and compute an own metric for better evaluation of object detection results, when applied to fine art photography.

4.3. Codebase

To develop the project, three different code repositories have been created. A git repository containing different datasets: This repository was used for testing out different models and frameworks when applying inference on the images. When developing the tidied up images, it was used too. It can be reached here: `https://gitlab.enterpriselab.ch/iameyer/odifap`. Another git repository was created to develop the web application. It can be accessed here: `https://gitlab.enterpriselab.ch/iameyer/odifap-program`. The third git repository was created to serve all necessary files for retraining the models. This repository contains a config.py file for every model and a .json file, that contains the bounding boxes and masks used for retraining. This repository can be reached with this URL: `https://gitlab.enterpriselab.ch/iameyer/odifap-training` Large files, like .pth files (checkpoint files, that are generated during training process) were saved to Google drive with the help of a custom Python script, as these files can be larger than 100MB in size.

4.4. Testing

In favour of having more time available for the development cycle and retraining of the model, no testing strategy has been selected. However during the project development process, needed functionality and user experience was reviewed continuously.

5. Realisation

5.1. Data collection

The first step in this project was to collect some data. Two pictures of fine art photography were already shown in the project description: An image by german photographer Andreas Gursky and one by Swiss artist Urs Wehrli. The main motivation behind data collection was to obtain images from different artists that each show a lot of different objects. This lets examine the performance of an object detection model applied to fine art photography, by using images that contain a lot of objects that optimally are also included in the COCO dataset.

5.1.1. COCO-dataset

The COCO (Common Objects in COntext) dataset is among the most popular image datasets created for object recognition tasks. It was lanced by Microsoft in 2014 and the most recent version was published in 2017. It uses 80 different classes for object detection tasks that can be seen here:

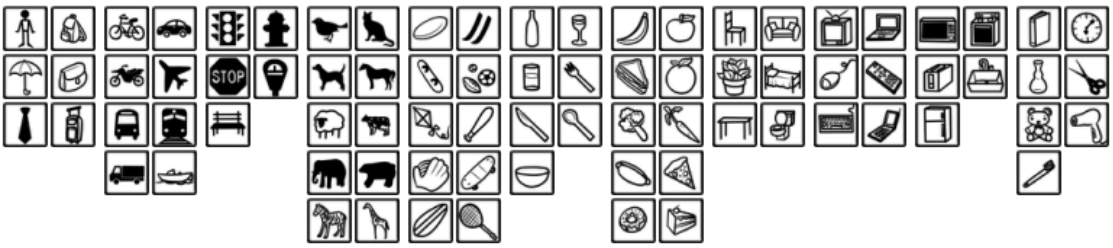


Figure 5.1.: All 80 classes from COCO dataset 2017

The COCO dataset not only contains bounding boxes and binary masks for object detection and segmentation, it also includes informations for keypoint estimation, panoptic segmentation and image captioning tasks. It is one of the big baselines for object detection tasks for many years now.

5.1.2. Collected dataset

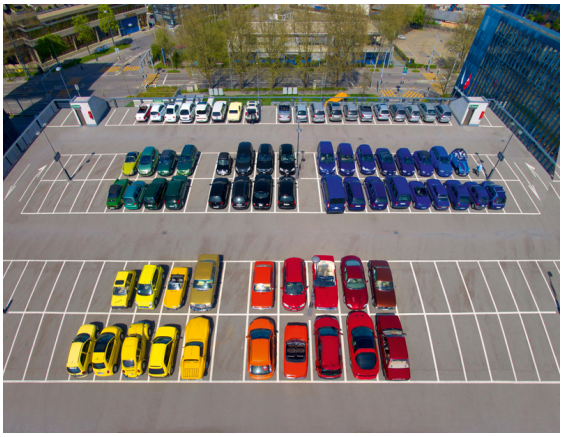
Urs Wehrli, the swiss artist behind "Kunst aufräumen" was asked to give his permission to use his images for this project. Thankfully he gave permission to do so. In total, a data set, containing of 42 different images from the four artists Urs Wehrli, Andreas Gursky, David LaChapelle and Jeff Wall has been gathered. The styles of these artists differ highly, as

Andreas Gursky, for example, uses a viewpoint from above and objects are shown rather small (because his images usually are very large in size). David LaChapelle, on the other hand, shows objects (e.g. people) in a rather standardised size, but by combining objects that do not belong together in a usual scenario, he puts these objects into a new context and uses massive postprocessing or digital manipulation to enhance his pictures. One sample picture from every artist is shown here:

(a) Andreas Gursky

(b) David LaChapelle

(c) Jeff Wall



(d) Urs Wehrli

Figure 5.2.: Sample picture shown from each of the four chosen artists

This dataset has been examined with different object segmentation models on Google Colab. Google Colab is a free to use infrastructure, powered by Google, that offers a zero-configuration Python and Jupyter notebook environment. Running on Linux Ubuntu with the latest Nvidia GPUs, this is a good option to get started with deep learning, as there are a lot of notebooks about every single kind of neural network tasks available.

5.2. Toolchain selection

5.2.1. Model

With the gathered dataset, different object segmentation models from different frameworks have been tried out on the dataset in inference mode. All these models have been pretrained on the COCO-dataset. The tried out models were: Mask R-CNN, CenterNet, Detectron2 and ShapeMask. In general, Mask R-CNN outperformed the other models, when inference is run on the dataset. Detectron2 also delivered a good performance but it threw an error when running on many images in a loop on Google Colab. The chosen model is of type Mask R-CNN with a ResNet-101-FPN backbone (a ResNet backbone that is 101 layers deep, coupled with a feature pyramid network, as explained in 2.5 on page 8).

5.2.2. Framework

At the same time different frameworks have been tested out. These were Tensorflow from Google, Detectron from Facebook and MMDetection, which is a part of the OpenMMLab project developed by Multimedia Laboratory by the Chinese University of Hong Kong. All these frameworks do at least contain one Mask R-CNN model. MMDetection has been chosen as the framework to develop the project in, because it worked out-of-the box when tried out on Google Colab. It returned results when running in inference mode on our dataset in about seven Minutes. MMDetection has a vast number of state-of-the-art models for computer vision tasks available and offers a high-level API that speeds up its usage. It is built on top of PyTorch (primarily developed by Facebook) and delivers a very good performance. An overview of available models in MMDetection (called the "Model Zoo") is shown here:

	MMDetection	maskrcnn-benchmark	Detectron	SimpleDet
Fast R-CNN	✓	✓	✓	✓
Faster R-CNN	✓	✓	✓	✓
Mask R-CNN	✓	✓	✓	✓
RetinaNet	✓	✓	✓	✓
DCN	✓	✓	✓	✓
DCNv2	✓	✓		
Mixed Precision Training	✓	✓		✓
Cascade R-CNN	✓		*	✓
Weight Standardization	✓	*		
Mask Scoring R-CNN	✓	*		
FCOS	✓	*		
SSD	✓			
R-FCN	✓			
M2Det	✓			
GHM	✓			
ScratchDet	✓			
Double-Head R-CNN	✓			
Grid R-CNN	✓			
FSAF	✓			
Hybrid Task Cascade	✓			
Guided Anchoring	✓			
Libra R-CNN	✓			
Generalized Attention	✓			
GCNet	✓			
HRNet	✓			
TridentNet [17]				✓

Figure 5.3.: Overview of MMDetection models

5.3. Programming language

Because most of the deep learning frameworks are using Python, Google Colab is using Python and also of its ease of use, Python has been used as the sole programming language to develop this project. In addition Python does offer a lot of visual computing libraries that were used to create the tidied up image.

5.4. Creating the tidied up image

5.4.1. Masking and cutting out the objects

After the dataset has been collected, and the model and the framework have been chosen, the next step was to extract all single objects from the output of the model when running

in inference mode on an image. Mask R-CNN model outputs for every input image among other things a list with with each a list of bounding boxes, masks, confidence score and predicted class of all objects found in the image. These four results were saved in a list object. The output is per default limited to 100 objects per image. The confidence score was used to filter all objects to get the objects with the highest quality as predicted by the model. The binary mask of the found objects was then applied to the image, creating binary images of the object. An example can be seen here:

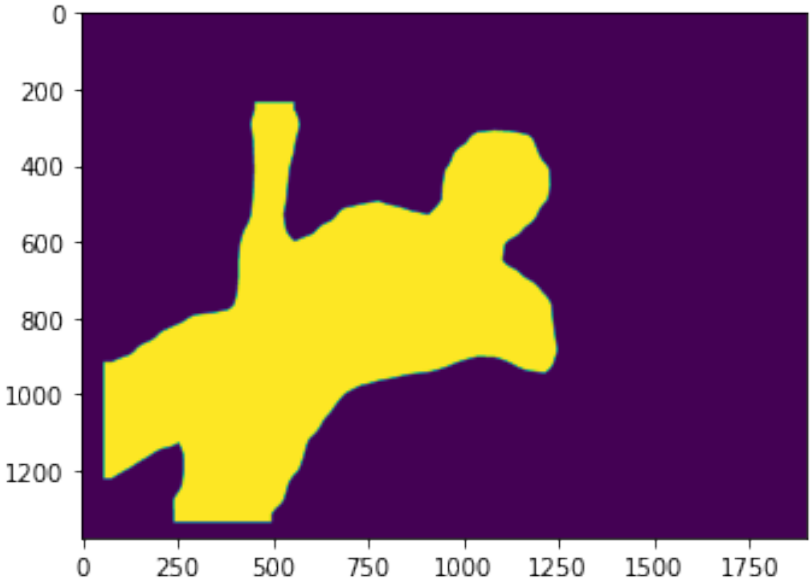


Figure 5.4.: Sample binary mask image

With help of the binary masks, the background of the found object was coloured white whereas the foreground stayed as is. Subsequently the corresponding bounding boxes were taken to cut out the objects from the image.

5.4.2. Creating the grid

The last step is to create a grid with all the found objects. This was done with the help of Python's Matplotlib. The idea is to create a grid with the number of rows according to the number of classes and insert every object as a new column. The problem with using Matplotlib's. `plt.subplots` is that it creates a grid where every image has to have the same size. To circumvent this, a more complicated approach was taken, by creating a grid manually with the help of the width and the height of all the found objects. A sample input of a tidied up image together with its output can be seen here:

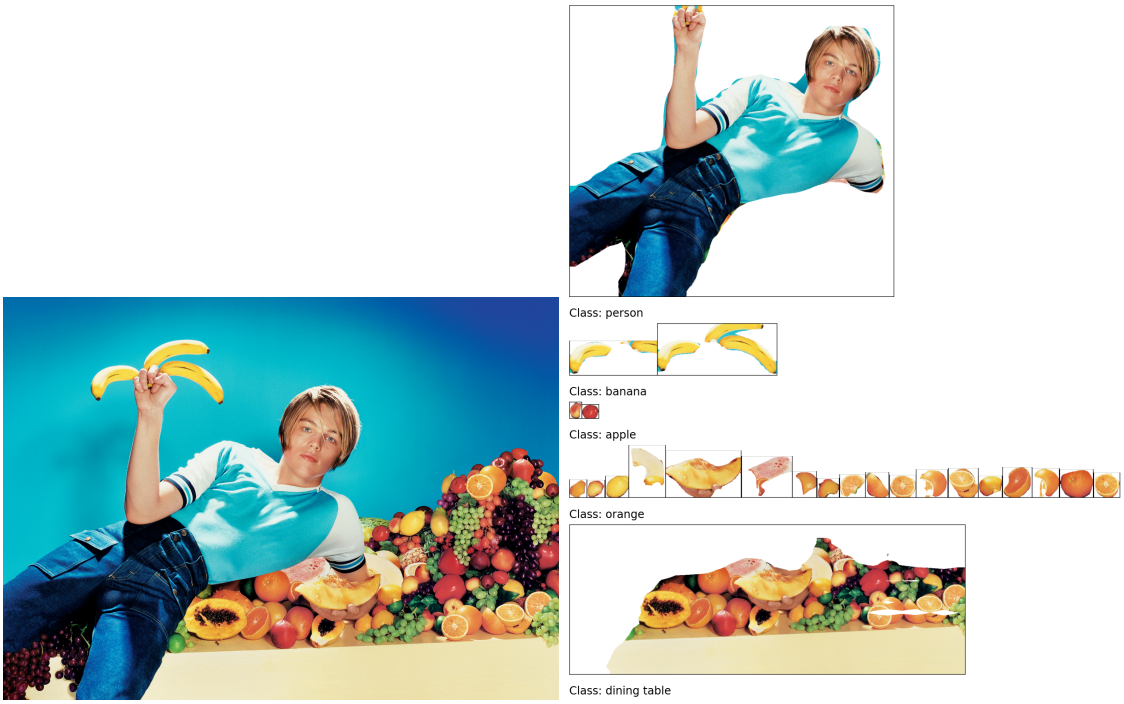


Figure 5.5.: Sample input and output image

5.5. Building the application

5.5.1. Running inference on CPU

After completing the tidied up image, the next step was to build an application that can load an image, run the model in inference mode and return the tidied up image from the input image. A difficulty was that MMDetection, like most deep learning frameworks, needs a Nvidia GPU to run even in inference mode. To solve this problem, a wrapper called SMD (simple MMDetection inference on CPU) was used, that takes in MMDetection models and can run the inference on an arbitrary CPU. To use SMD, many data structures had to be adjusted slightly. More information about SMD can be found here: [1].

5.5.2. Creating the web application

To convert the program into a full-blown web application, a framework, called Plotly-Dash has been used. Plotly-Dash is using Flask to create a webserver and is using HTML-, CSS- and JavaScript-technologies under the hood to create a running web application from a Python program. It offers out-of-the-box user interface (UI) components, that are useful to rapid prototype a web application. With the help of UI-elements like buttons and dropdown-lists, the user is given the possibility to upload and select images and models and to start

the inference by himself. There are also two sliders built in: One to adjust the confidence threshold score of the model and one to adjust the input image size. Executed time was measured and gets written too to get an insight into performance of the model.

One of the goals of the web application was to build it as modular as possible. This was achieved with the possible selection of the model and three different modes to select an image from (predefined list, upload and retrieve via URL). One difficulty with Plotly-Dash was that images when uploaded to the web server, got encoded with base64 encoding. It took some time to find out, how to decode them for further use. A screenshot showing the web application and its UI can be seen here:

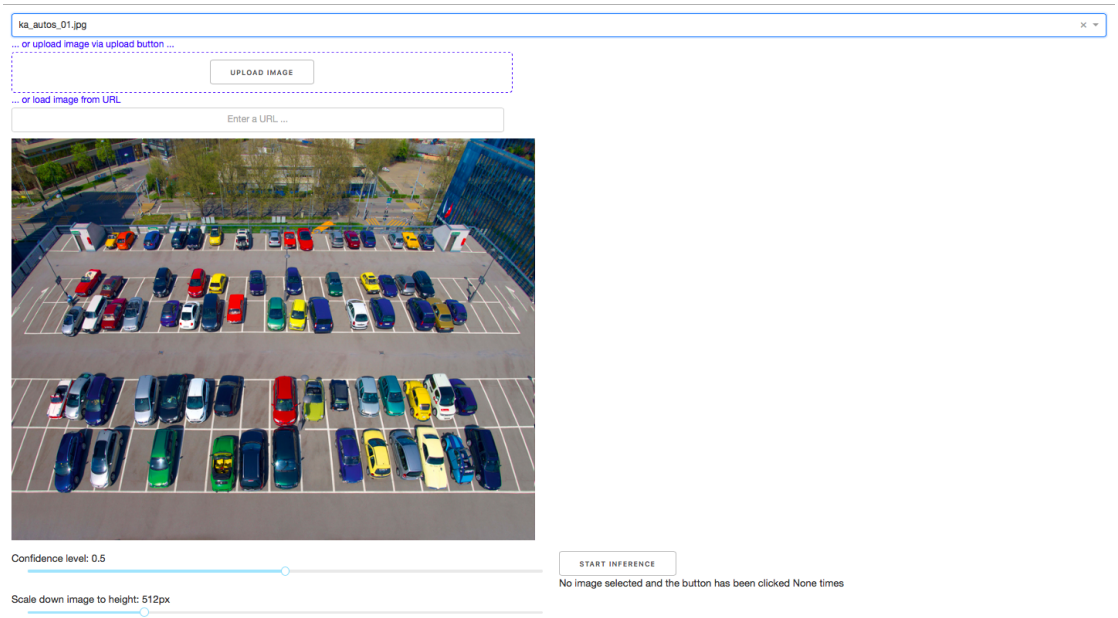


Figure 5.6.: Screenshot showing the web application

5.6. Deployment

To deploy the application, server space on the EnterpriseLab has been allocated. The web application can be accessed at: `http://bdaf20-iameyer.enterpriselab.ch`. One difficulty was to let the application run on port 80 (HTTP). The support team from EnterpriseLab hat to redirect port 80 to 8000 in order to let the application run without specifying another port in the URL.

5.7. A more precise research question

As Urs Wehrli luckily gave his confirmation to use his photos from the series "Kunst aufräumen", I decided to use these images for retraining one or more models. Each model gets

trained on the tidy version of an image and gets tested on the messy version of it. Optimally, the model should find all objects that are visible in the messy image and classify them correctly. There are some difficulties though. Per example in some pictures most objects are hidden because they are obscured by other objects in the foreground, as can be seen in the following picture:

Figure 5.7.: Most of the objects on the image are obscured by other objects

Optimally, the output of the model should contain exactly the same objects which are shown in the tidy version. For some photographies in his series, Urs Wehrli does use a particular technique to sort objects to a further degree in the "tidy" image. Per example using the colour or the size of the found objects.

5.8. Data labelling

A dataset containing of three images from the series "Kunst aufräumen" by Urs Wehrli has been chosen to refine the model. As each of these three images contains a messy and a tidy version, it does make a good template to use the tidied up version as a training image and the messy version as a test image. To accomplish training with own images, one has to label these images first. When training an object segmentation model, it is necessary to train the classifier with images that contain masks in it.

There are several tools that offer object masking with polygons or brushes. The chosen tool is named RectLabel. A total of six images (three messy ones and three tidy ones) have been labelled in order to use for refining the model. During labelling, a .xml-file gets generated, that after completion can be converted into the COCO-format as a .json-file.

5.9. Refining the model

Google Colab was used again as the platform to retrain the model. With just three training images it is quite extreme to train a classifier. This reflected in the poor performance in the beginning of the retraining process. After deciding to retrain a single model for every picture pair ("messy" and "tidy" one), performance got better. For a task like this, when training with a small dataset, data-augmentation is crucial: It lets the model use the same image over and over again after applying different kind of transformations and distortions to it.

When retraining with MMDetection, one has to adjust several things: A .json-file, containing all the classes, masks and bounding boxes in COCO-format and a config.py-file that contains all the needed settings for the training process (shown in the appendix). During retraining, checkpoint-files get saved in a defined interval. These checkpoint-files can be used resume training on a later point of time. After one training cycle (between 1000 and 2000 epochs), the last checkpoint-file was taken and examined on the test-image (the "tidy" image).

6. Evaluation and validation

6.1. Results

6.1.1. Pretrained on COCO-dataset model

When applied to images of fine-art photography we can observe that performance of object-detection models is highly variable to some extend because different photographers use different styles to depict their objects in their images.

For example when using an image with a lot of smaller objects (like Andreas Gursky does), performance is rather poor compared to an image with fewer and bigger objects. An example of the pretrained on COCO-dataset Mask R-CNN model applied to an Andreas Gursky picture is shown here:

Figure 6.1.: Pretrained model applied on an image from Andreas Gursky

As can be seen here, only the largest objects in the foreground (of class "person") got detected. The reason could be because the objects are rather small and photographed from above. However the usage of a FPN should adjust performance when using pictures with small objects depicted. An interesting task would be to train a model merely with pictures from Andreas Gursky to further examine this anomaly.

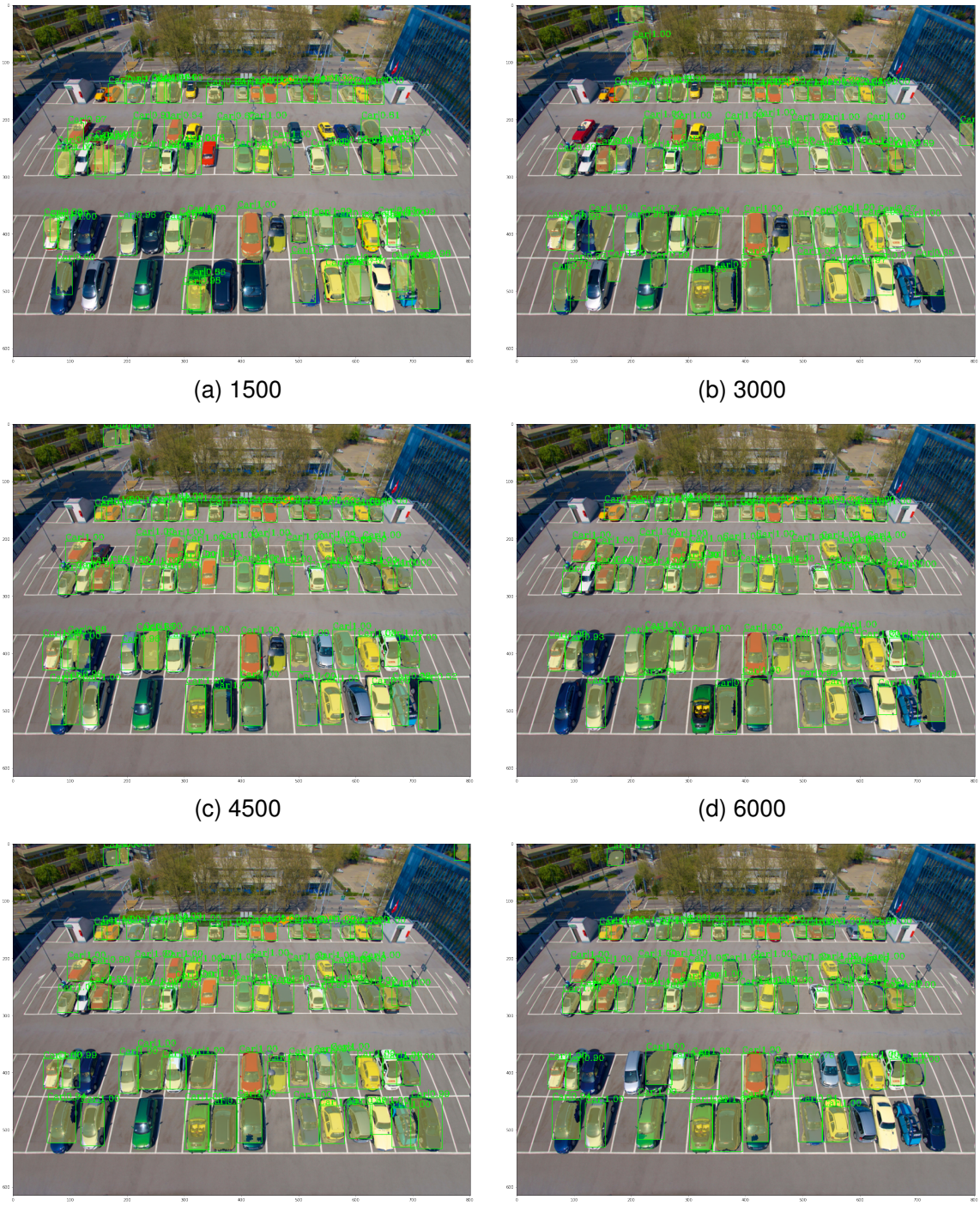
Another oddity that was observed multiple times, was that an object of type "cell phone" was detected when a person puts their hand close to their face. One explanation could be that in photos, people are usually have their cell phones near their ear. An example is shown here (detail shown):

Figure 6.2.: Detail from a picture from David LaChapelle where the models falsely detects a cell phone

In sum, the performance of the pretrained on COCO-dataset Mask R-CNN model depends on the following criteria: The class of objects depicted in the image (included in COCO-dataset or not), the size of objects in the image, the perspective and angle from which the object was photographed and the degree to which an object is visible and is shown in a natural style.

6.1.2. Retrained on "Kunst aufräumen" data model

As explained in section 5.9 on page 23, an own model was created by retraining the out-of-the-box R-CNN model via images from "Kunst aufräumen" by Urs Wehrli. The first model, that got retrained was with the car photo. A model could get trained without use of much data augmentation. The reason, as it is my assumption, is that in the car photos, the car objects are just rotated in two different positions: 0° or 180°. Therefore it is not necessary to include random rotations in the training pipeline.



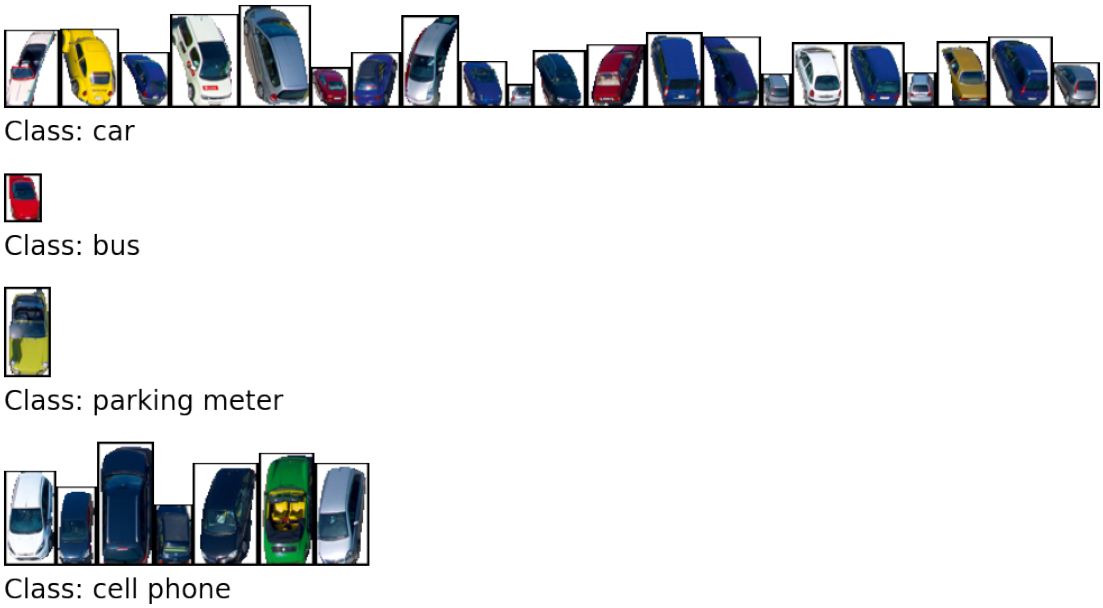
(a) 1500 (b) 3000

(c) 4500 (d) 6000

(e) 7500 (f) 9000

Figure 6.3.: Output of the retrained on "Kunst aufräumen" model, trained with 1500, 3000, 4500, 6000, 7500 and 9000 epochs each.

As one can easily see, there was a steady improvement up to about 4500 epochs. From 7500 to 9000 periods there seems to be even a degradation of the performance of the model. To illustrate the performance of the retrained model compared to the pretrained model further, the output from each model on the "messy" car photo as input into the "tidy" algorithm is shown:



(a) pretrained on COCO-model output as "tidy" image (21 objects got detected as "Car")

(b) retrained model output as "tidy" image (65 objects got detected as "Car")

Figure 6.4.: Comparing the output as "tidied up" images when using the pretrained and the retrained model

When using the retrained model on image material from the COCO-dataset, one can easily see that performance has decreased very significantly. The model has not only forgotten many previously known classes, but has also become too accustomed to the training data (called overfitting). To illustrate this, the output from both models on an input image with cars from the COCO-dataset is shown:

Figure 6.5.: Testing the pretrained and the retrained model on a COCO-dataset image

Apparently the model has learned the angle from which the object is viewed and the size of the object.

6.2. Reviewing the web application

The web application is able to fulfil the requirements and is able to solve the given project task. A nice and clean user interface could be designed and developed. However there are some flaws that should be resolved.

6.2.1. Weaknesses

The first thing that got noticed is the very long waiting time until inference is completed. For a large picture, with a lot objects shown, this can be as long as up to 18 seconds or more (using the "Leonard DiCaprio unspoiled" picture from David LaChapelle with a height of 1024px). For a much smaller picture with only few objects, inference time is still about 9 seconds (using the sample COCO-dataset picture from section 1.2 on page 2. To yield a decent user experience, calculation time should not exceed two seconds. By switching to a GPU-powered infrastructure, this lack of performance could possibly be solved very easily.

6.2.2. Known bugs

6.3. Comparison with given requirements

The following tasks were successfully accomplished: A dataset, containing of 42 images from four different artist, has been gathered. Different object detection models and frameworks have been tested out on this dataset on Google Colab. The most promising model

has been chosen to implement in a web application. The web application has been developed and deployed on the EnterpriseLab successfully. An own model has been created by refining the pretrained on COCO-dataset model with new data from "Kunst aufräumen".

Unfortunately there was no time left over to develop an own metric to examine and compare different models when applied to pictures of artistic photography.

6.4. Evaluation of technical tools

Retrospectively, the used toolchain emerged to be highly effective. Google Colab is a very powerful infrastracture, especially when one does not possess an own Nvidia GPU. To share and comment whole notebooks or single cells was a big plus.

MMDetection turned out to be a fast, modular and powerful framework for computer vision projects. Very helpful was the ability to save checkpoints and to resume training at a later point of time. MMDetection received a lot of traction in the last years and is in a steady change, version 2.0.0 just got released in April 2020.

Having only little prior experience with python, it was nevertheless possible to implement the entire project. Python has proven to be very powerful for working with images and processing images efficiently. This is mainly owed to powerful libraries, like numpy, open-cv or Python Image Library (PIL).

Plotly-Dash was a joy to work with, as it lets the developer turn a Python program into a full-fledged web application very quickly with minimal overhead. As it is built upon JavaScript technologies (React), it is highly customizable too. There exists a lively community that is eager to help.

Unluckily, to this day, most frameworks do not support operating on CPU in inference mode (however this was fixed in MMDetection 2.0.0 in April 2020). To accomplish this, an indirection via SMD, as explained in 5.5.1 on page 20 had to be taken. Running the inference on CPU has one major drawback: It slows down computation time badly. To serve the best performance and user experience, a GPU-powered server infrastructure should be taken into consideration.

6.5. Evaluation of used method

The used approach, to proceed to retraining after the whole toolchain got finished, turned out to be the right decision. However not all milestones could be reached in the agreed point of time, as can be seen in the next section.

6.5.1. Evaluation of project management and project procedure

Due to several project difficulties, the achievement of certain milestones was delayed for some time:

Milestone	Agreed date	Actual date
Test pretrained models on Google Colab with new data	24th March 2020	24th March 2020
Build web application prototype	28th April 2020	5th May 2020
Finetune model with fine art photography data	12th May 2020	31th May 2020
Test finetuned model on new data	19 May 2020	31th May 2020
Finish web application with new model	2nd June 2020	3rd June 2020

Table 6.1.: Agreed and actual dates of all milestones

6.5.2. Project difficulties

Because MMDetection did not support inference via CPU before version 2.0.0, a detour through the wrapper SMD had to be made. To successfully integrate SMD into the application, some data structures had to be modified. This changes resulted in the workload of some working days. When uploading an image file to a Plotly-Dash application, data gets encoded with a base64-encoding. To find out how to decode this data back into an image object cost about another one or two working days. Retraining a model with MMDetection was not simple either. As there was only a very small training set, it was difficult to judge if the configuration was wrong or if the data set was just too small.

7. Outlook

7.1. Conclusion

By and large, it was an inspiring and exciting experience to engage with deep learning through the use of fine art photography. Since the topic is highly complex and abstract, and the project was very extensive, it was unfortunately not feasible to go into every detail in depth. Therefore, I am still not able to understand and cover every aspect of CNNs fully. However, it has awakened interest and the desire to continue to study the topic of deep learning.

7.2. Outlook

In a future work one could for example train a model on images from one photographer and test it on images from another photographer which contain the same classes of objects in it. From a technical point of view it would be interesting to decrease the computation time of the inference as much as possible, either by using a GPU-powered environment for deployment or by using a lighter model that provides sufficiently fast computation on CPU.

List of Figures

List of Tables

Formelverzeichnis

Bibliography

[1] Karaniewicz, A. (n.d.). SMD (simple mmdetection). Retrieved May 30, 2020, from https://github.com/akarazniewicz/smd

[2] 6. Convolutional Neural Networks. (n.d.). Retrieved May 30, 2020, from https://d2l.ai/chapter_convolutional-neural-networks/index.html

[3] Goodfellow, I., Bengio, Y.„ Courville, A. (2016). Deep Learning. MIT Press.

[4] LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE (p./pp. 2278–2324), .

[5] Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou & K. Q. Weinberger (ed.), Advances in Neural Information Processing Systems 25 (pp. 1097–1105) . Curran Associates, Inc. .

[6] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770-778.

[7] Girshick, R.B., Donahue, J., Darrell, T., & Malik, J. (2014). Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. 2014 IEEE Conference on Computer Vision and Pattern Recognition, 580-587.

[8] Girshick, R.B. (2015). Fast R-CNN. 2015 IEEE International Conference on Computer Vision (ICCV), 1440-1448.

[9] Ren, S., He, K., Girshick, R.B., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Transactions on Pattern Analysis and Machine Intelligence, 39, 1137-1149.

[10] He, K., Gkioxari, G., Dollár, P., & Girshick, R.B. (2017). Mask R-CNN. 2017 IEEE International Conference on Computer Vision (ICCV), 2980-2988.

[11] Lin, T., Dollár, P., Girshick, R.B., He, K., Hariharan, B., & Belongie, S.J. (2017). Feature Pyramid Networks for Object Detection. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 936-944.

[12] Hui, J. (n.d.). Understanding feature pyramid networks for object detection (FPN). Retrieved June 1, 2020, from https://medium.com/@jonathan_hui/understanding-feature-pyramid-networks-for-object-detection/-fpn-45b227b9106c

A. Appendix

A.1. Config.py

```
# model settings
model = dict(
    type='MaskRCNN',
    pretrained='torchvision://resnet101',
    backbone=dict(
        type='ResNet',
        depth=101,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        frozen_stages=1,
        style='pytorch'),
    neck=dict(
        type='FPN',
        in_channels=[256, 512, 1024, 2048],
        out_channels=256,
        num_outs=5),
    rpn_head=dict(
        type='RPNHead',
        in_channels=256,
        feat_channels=256,
        anchor_scales=[8],
        anchor_ratios=[0.5, 1.0, 2.0],
        anchor_strides=[4, 8, 16, 32, 64],
        target_means=[.0, .0, .0, .0],
        target_stds=[1.0, 1.0, 1.0, 1.0],
        loss_cls=dict(
            type='CrossEntropyLoss', use_sigmoid=True, loss_weight=1.0),
        loss_bbox=dict(type='SmoothL1Loss', beta=1.0 / 9.0, loss_weight=1.0)),
    bbox_roi_extractor=dict(
        type='SingleRoIExtractor',
        roi_layer=dict(type='RoIAlign', out_size=7, sample_num=2),
        out_channels=256,
        featmap_strides=[4, 8, 16, 32]),
    bbox_head=dict(
        type='SharedFCBBoxHead',
```

```
        num_fcs=2,
        in_channels=256,
        fc_out_channels=1024,
        roi_feat_size=7,
        num_classes= 2,
        target_means=[0., 0., 0., 0.],
        target_stds=[0.1, 0.1, 0.2, 0.2],
        reg_class_agnostic=False,
        loss_cls=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0),
        loss_bbox=dict(type='SmoothL1Loss', beta=1.0, loss_weight=1.0)),
    mask_roi_extractor=dict(
        type='SingleRoIExtractor',
        roi_layer=dict(type='RoIAlign', out_size=14, sample_num=2),
        out_channels=256,
        featmap_strides=[4, 8, 16, 32]),
    mask_head=dict(
        type='FCNMaskHead',
        num_convs=4,
        in_channels=256,
        conv_out_channels=256,
        num_classes= 2,
        loss_mask=dict(
            type='CrossEntropyLoss', use_mask=True, loss_weight=1.0)))
# model training and testing settings
train_cfg = dict(
    rpn=dict(
        assigner=dict(
            type='MaxIoUAssigner',
            pos_iou_thr=0.7,
            neg_iou_thr=0.3,
            min_pos_iou=0.3,
            ignore_iof_thr=-1),
        sampler=dict(
            type='RandomSampler',
            num=256,
            pos_fraction=0.5,
            neg_pos_ub=-1,
            add_gt_as_proposals=False),
        allowed_border=0,
        pos_weight=-1,
        debug=False),
    rpn_proposal=dict(
        nms_across_levels=False,
```

```
        nms_pre=2000,
        nms_post=2000,
        max_num=2000,
        nms_thr=0.7,
        min_bbox_size=0),
    rcnn=dict(
        assigner=dict(
            type='MaxIoUAssigner',
            pos_iou_thr=0.5,
            neg_iou_thr=0.5,
            min_pos_iou=0.5,
            ignore_iof_thr=-1),
        sampler=dict(
            type='RandomSampler',
            num=512,
            pos_fraction=0.25,
            neg_pos_ub=-1,
            add_gt_as_proposals=True),
        mask_size=28,
        pos_weight=-1,
        debug=False))
test_cfg = dict(
    rpn=dict(
        nms_across_levels=False,
        nms_pre=1000,
        nms_post=1000,
        max_num=1000,
        nms_thr=0.7,
        min_bbox_size=0),
    rcnn=dict(
        score_thr=0.05,
        nms=dict(type='nms', iou_thr=0.5),
        max_per_img=100,
        mask_thr_binary=0.5))
# dataset settings
dataset_type = 'CocoDataset'
data_root = '/content/training/odifap-training/data/'
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True, with_mask=True),
    dict(type='RandomCrop', crop_size=(256, 256)),
    dict(type='Resize', img_scale=(400, 300), keep_ratio=True),
```

```
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', img_scale=(400, 300), keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
data = dict(
    imgs_per_gpu=4,
    workers_per_gpu=4,
    train=dict(
        type=dataset_type,
        ann_file=data_root + 'train_autos/annotations_train_autos.json',
        img_prefix=data_root + 'train_autos/images/',
        pipeline=train_pipeline),
    val=dict(
        type=dataset_type,
        ann_file=data_root + 'test_autos/annotations_test_autos.json',
        img_prefix=data_root + 'test_autos/images',
        pipeline=test_pipeline),
    test=dict(
        type=dataset_type,
        ann_file=data_root + 'test_autos/annotations_test_autos.json',
        img_prefix=data_root + 'test_autos/images',
        pipeline=test_pipeline))
evaluation = dict(interval=150, metric=['bbox', 'segm'])
# optimizer
optimizer = dict(type='SGD', lr=0.04, momentum=0.9, weight_decay=0.0001)
optimizer_config = dict(grad_clip=dict(max_norm=35, norm_type=2))
# learning policy
```

```
lr_config = dict(
    policy='step',
    warmup='linear',
    warmup_iters=1,
    warmup_ratio=1.0 / 3,
    step=[8, 11])
checkpoint_config = dict(interval=150)
# yapf:disable
log_config = dict(
    interval=150,
    hooks=[
        dict(type='TextLoggerHook'),
        # dict(type='TensorboardLoggerHook')
    ])
# yapf:enable
# runtime settings
total_epochs = 1500 # start with 1500
dist_params = dict(backend='nccl')
log_level = 'INFO'
work_dir = './mask_rcnn_r101_fpn_1x_2'
load_from = None
resume_from = None
workflow = [('train', 1)]
```