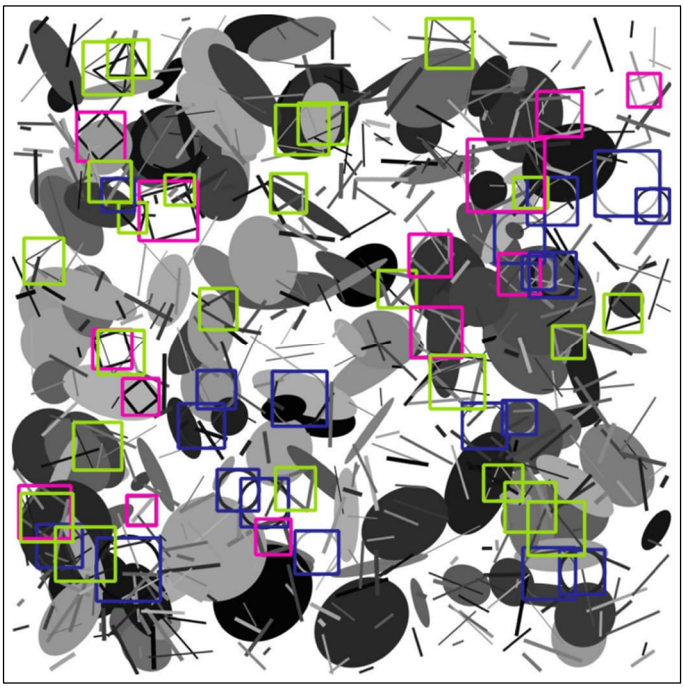
Jürg Krähenbühl Zimmermann
jkraehen@gmail.com

Data Science Project

Detection of simple shapes with Faster R-CNN

July 6, 2023
CAS Applied Data Science, Universität Bern



Abstract

State of the art methods for object detection are based on artificial neural networks: In this project, the Faster R-CNN detection model is applied to a specific computer vision task where simple geometric shapes, i.e., circles, triangles, and squares, are to be detected. The processed images are synthetically generated and labeled by code specifically written for this project.

Table of Contents

1 Introduction

1.1 Project Objective

State of the art methods for object detection are based on artificial neural networks [1]. The goal of this project is to successfully apply neural networks to a specific computer vision task.

On pictures like the example below, with simple geometric shapes on it, i.e., circles, squares, and triangles, each shape and its position on the image should be detected. To make the task more challenging, the shapes are overlapping, and the picture contains extra noise. The synthetic generation of such images is also part of the project objective.

Figure 1, Picture to process (try to find all circles, squares, and triangles, by yourself)

The result of processing a given image is the list of simple geometric shapes contained in the image, such that each entry in the list contains the shape's class name and its bounding box.

1.2 Methods

The main outcome of this project is some trained Faster R-CNN neural networks [2] and an image processing pipeline. All the programming is done in Python [3] and Jupyter Notebooks [4]. Training of the neural networks and processing the images is run in Google Colab [5], where fast GPUs for efficient processing can be activated.

For general evaluation or exploration of data several Python libraries are used, mainly Pandas [6], NumPy [7] and Matplotlib [8]. For handling of neural networks, PyTorch library [9] is used.

GitHub [10] serves as codebase with version control for all written code.

1.3 Preliminary Studies

Starting point for this project is a blog post [11] and a GitHub project [12] by Johannes Rieke, where multiple basic geometric shapes of different type are successfully detected on an image by using basic feed forward neural networks.

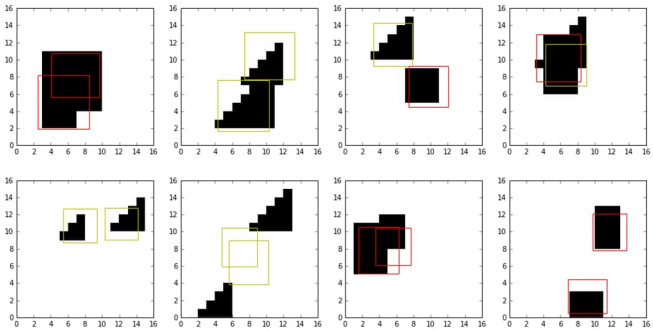


Figure 2, Results of the GitHub project by Johannes Rieke

Inspired by Rieke, I started building my own synthetic images showing simple geometric shapes and some intended noise.

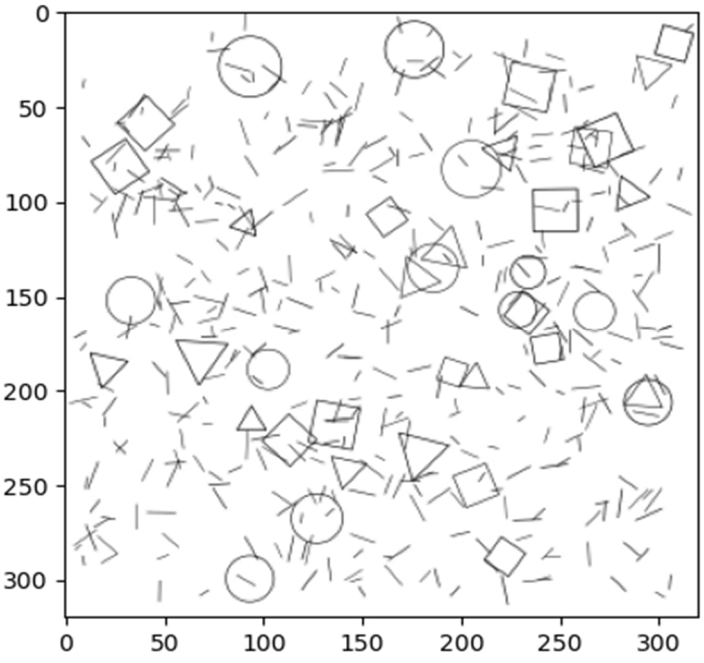


Figure 3, shapes and noise

In the GitHub project [13] these images are used to create detail images for the detection of circles, triangles, and squares, by basic machine learning models.

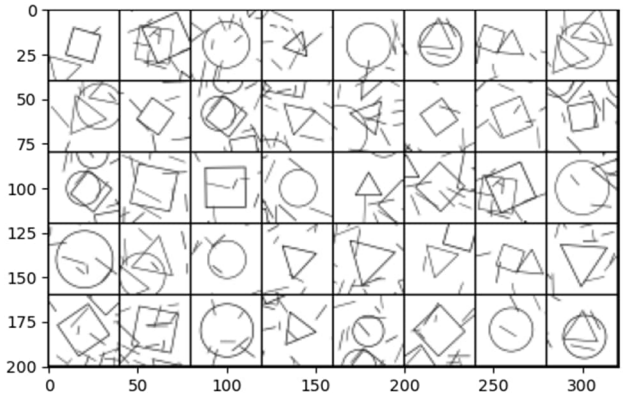


Figure 4, Detail images of shapes

The results of that first project already showed that the simple ML models (without CNNs) are not adequate for the target objective of this project.

1.4 Risks

Failing the project objectives can happen by different reasons:

- The design of the neural network does not fit the problem.
- The training sets are too small.
- The computing power for training is too expensive (time, storage, cost).

Luckily, almost any expectation on the project outcome can be relaxed and the results can be improved by adjusting the generation of "ground truth" images:

- Increase the number of samples in the training set, by just generating more images.
- Reduce the number of shapes in an image.
- Change the size or resolution of the image.
- Reduce the number of types of shapes in the image (e.g., only circles).
- Reduce the differences on size and orientation of the shapes (i.e., just allow limited levels of size, limited set of angles of rotation).
- Reduce or eliminate the disturbing noise in the images.
- Prevent shapes from overlapping.

2 Data

2.1 Data Source

The data is generated from scratch by running Python code with a pseudo-random number generator involved [14]. Labeling of the images happens as part of the image creation. The whole process of data generation can run on-the-fly at any time and in a reproducible manner. For this project the sample data is stored on Google drive after an initial single run of the generator.

Generator code on GitHub: https://github.com/jkcas22/shape-recognition/blob/main/data.ipynb
Some sample Data: https://github.com/jkcas22/shape-recognition/tree/main/data

2.2 Metadata

All the metadata is contained in the Python code for generation of the data [15]. There you also can find information about the specific parametrization involved in the process of data generation.

2.3 Data Quality

Due to the synthetic generation of the data by our own Python code, data quality and labeling of the images are not a problem (other than testing the code itself). The amount of sample data can be increased at will, only limited by the runtime and the storage space of the servers and computers involved.

2.4 Conceptual Data Model

We are working with 4 different types of images. All images contain small circles, triangles, and squares (objects to detect). The picture type differs in complexity from 0 to 3. With a higher type, the pictures become more and more complex since they contain additional noise in the form of filled ellipses in the background and strokes in the foreground.

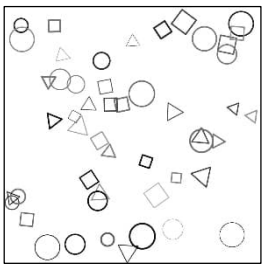


Figure 5, Complexity 0

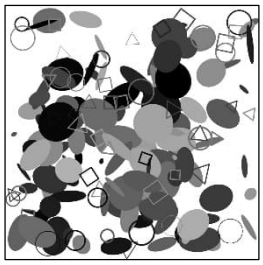


Figure 6, Complexity 1

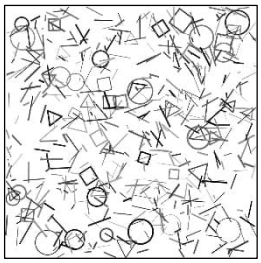


Figure 7, Complexity 2

Figure 8, Complexity 3

2.5 Logical Data Model

All pictures are grayscale images of a fixed size of 1024x1024 pixel. Any image contains exactly 50 objects to detect and depending on its type also noise in the form of 120 ellipses and/or 500 lines. All figures have a randomly varying grayscale value, size, position, and orientation on the image.

Any image is accompanied by a list of label data. For each shape on the image there is one entry in the list. Each entry contains the class name, i.e., circle (2), triangle (3), or square (4), and the corresponding shape's bounding box, i.e., its coordinate values xmin, ymin, xmax, ymax.

```
<annotation>
    <folder/>
    <filename>train0000.jpg</filename>
    <path>train0000.jpg</path>
    <source>synthetic</source>
    <segmented>0</segmented>
    <size>
        <width>1024</width>
        <height>1024</height>
        <depth>1</depth>
    </size>
    <object>
        <name>3</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>960</xmin>
            <xmax>1007</xmax>
            <ymin>414</ymin>
            <ymax>464</ymax>
        </bndbox>
    </object>
    ...
</annotation>
```

Figure 9, Sample annotation xml

2.6 Physical data model

The images are stored in jpg format and the corresponding label data is stored as Pascal VOC2012 Data [16] in xml format.

3 Exploratory Data Analysis

A quantitative exploration of the data is not necessary because we have full control over the creation of the images and labels, see [15], and further we trust in the randomness of the pseudo-random number generator of the NumPy library [14]. Nevertheless, we give a qualitative exploration of the images by having a look at some sample images: On the left side we show the original image, and on the right side there is 49 focus images of particular shapes from the original image.

Images of complexity 0 with different shapes:

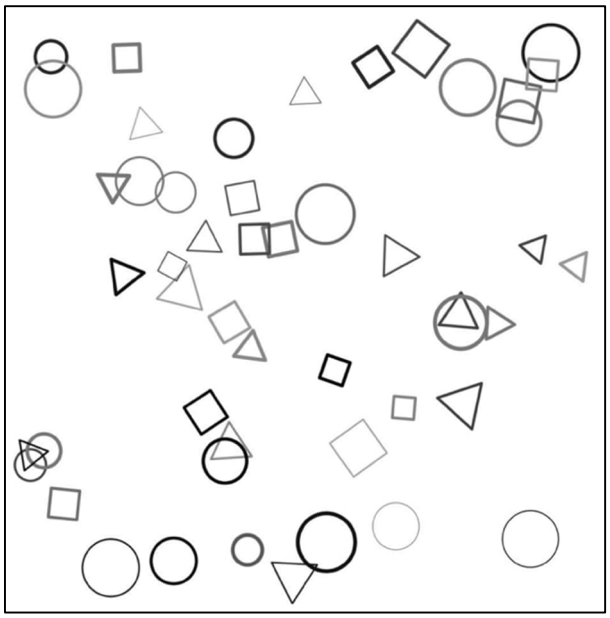


Figure 10, Original Image

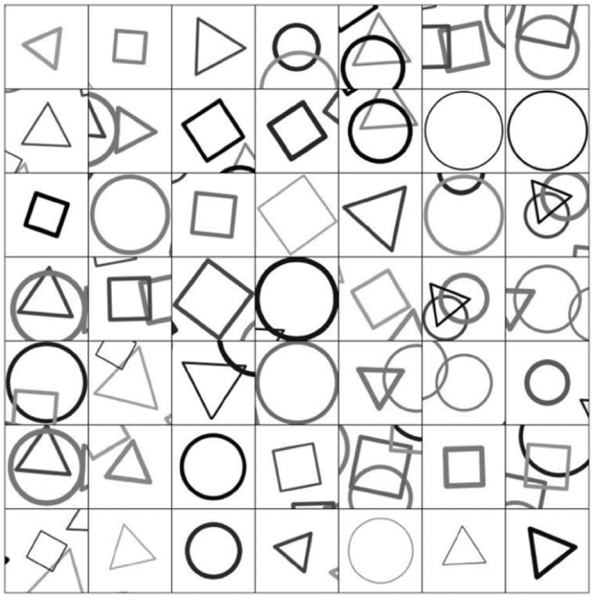


Figure 11, Focus on shapes

Images of complexity 1 with different shapes and varying background:

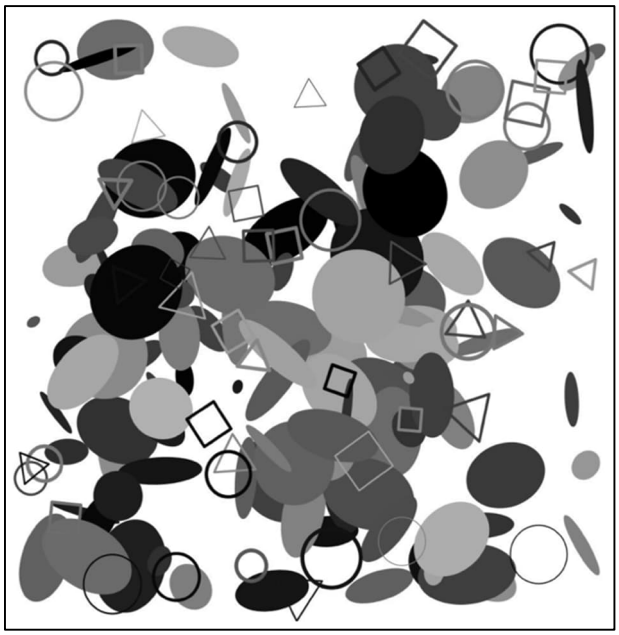


Figure 12, Original image

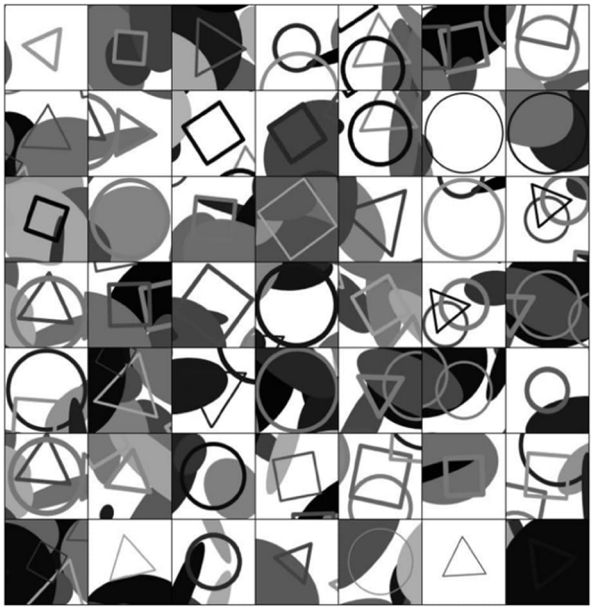


Figure 13, Focus on shapes

Images of complexity 2 with different shapes and noise in the foreground:

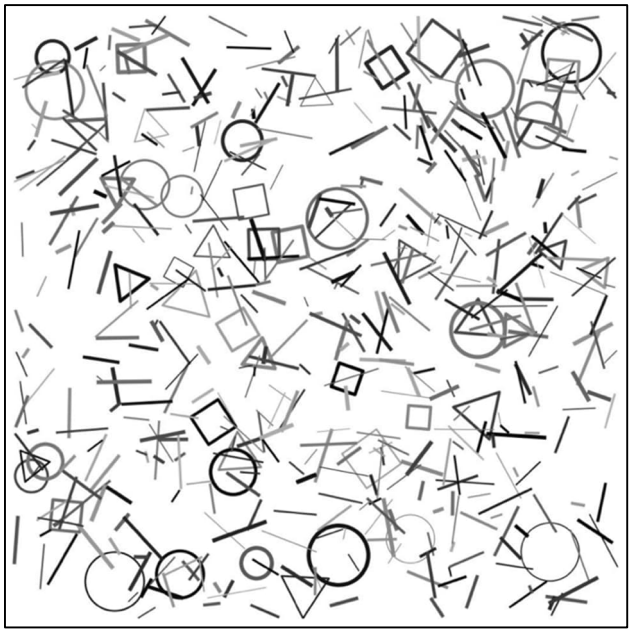


Figure 14, Original image

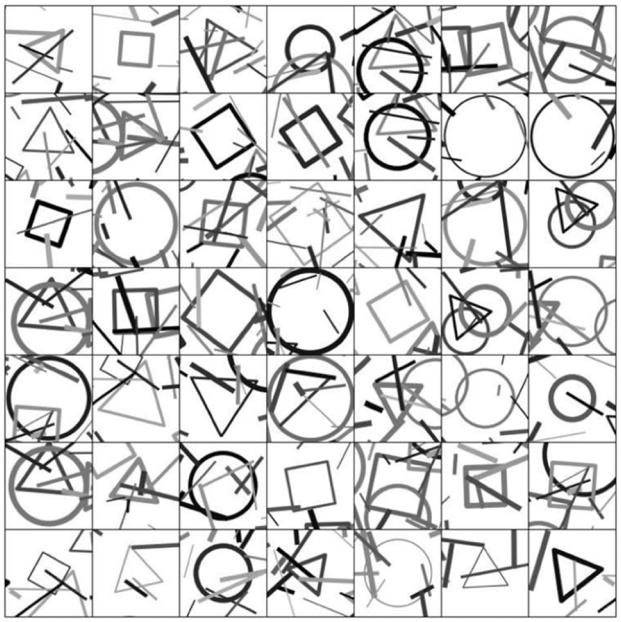


Figure 15, Focus on shapes

Images of complexity 3 with different shapes, varying background, and noise:

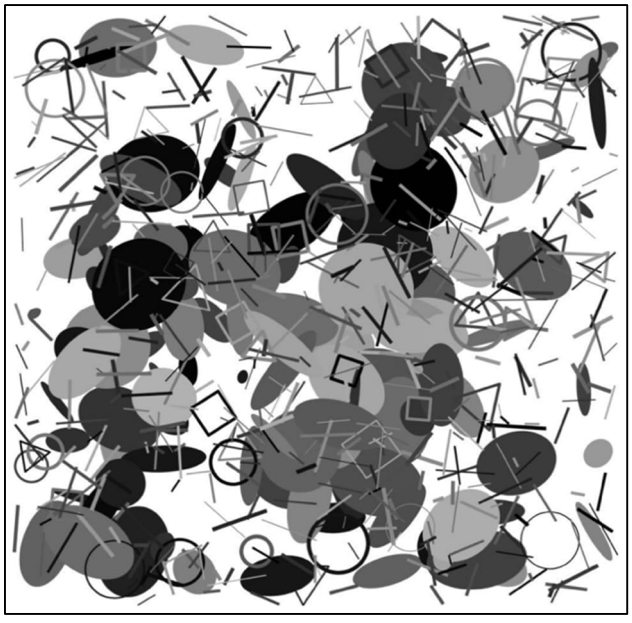


Figure 16, Original image

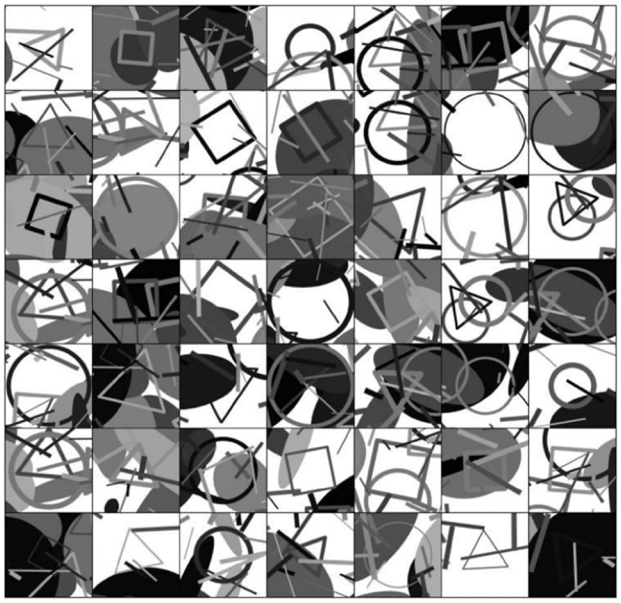


Figure 17, Focus on shapes

4 Machine Learning Analysis

4.1 Data Flow

For object detection we use the ResNet 50 FPN V2 deep neural network from PyTorch [17,18], and for training and evaluation of the Faster R-CNN [2] model on our custom data, we adapt already existing code on GitHub [19]. For a nice explanation of the general design and data flow of Faster R-CNN networks see [20].

The pre-trained model we use is quite big. It has a total of 43'266'403 parameters and consists of the following layers:

```
==========================================================================================
Layer (type (var_name))                            Output Shape              Param #
==========================================================================================
FasterRCNN (FasterRCNN)                            [100, 4]                  --
├─GeneralizedRCNNTransform (transform)             [2, 3, 800, 800]          --
├─BackboneWithFPN (backbone)                       [2, 256, 13, 13]          --
│    └─IntermediateLayerGetter (body)              [2, 2048, 25, 25]         --
│    │    └─Conv2d (conv1)                         [2, 64, 400, 400]         (9,408)
│    │    └─BatchNorm2d (bn1)                      [2, 64, 400, 400]         (128)
│    │    └─ReLU (relu)                            [2, 64, 400, 400]         --
│    │    └─MaxPool2d (maxpool)                    [2, 64, 200, 200]         --
│    │    └─Sequential (layer1)                    [2, 256, 200, 200]        (215,808)
│    │    └─Sequential (layer2)                    [2, 512, 100, 100]        1,219,584
│    │    └─Sequential (layer3)                    [2, 1024, 50, 50]         7,098,368
│    │    └─Sequential (layer4)                    [2, 2048, 25, 25]         14,964,736
│    └─FeaturePyramidNetwork (fpn)                 [2, 256, 13, 13]          --
│    │    └─ModuleList (inner_blocks)              --                        (recursive)
│    │    └─ModuleList (layer_blocks)              --                        (recursive)
│    │    └─ModuleList (inner_blocks)              --                        (recursive)
│    │    └─ModuleList (layer_blocks)              --                        (recursive)
│    │    └─ModuleList (inner_blocks)              --                        (recursive)
│    │    └─ModuleList (layer_blocks)              --                        (recursive)
│    │    └─ModuleList (inner_blocks)              --                        (recursive)
│    │    └─ModuleList (layer_blocks)              --                        (recursive)
│    │    └─LastLevelMaxPool (extra_blocks)        [2, 256, 200, 200]        --
├─RegionProposalNetwork (rpn)                      [1000, 4]                 --
│    └─RPNHead (head)                              [2, 3, 200, 200]          --
│    │    └─Sequential (conv)                      [2, 256, 200, 200]        1,180,160
│    │    └─Conv2d (cls_logits)                    [2, 3, 200, 200]          771
│    │    └─Conv2d (bbox_pred)                     [2, 12, 200, 200]         3,084
│    │    └─Sequential (conv)                      [2, 256, 100, 100]        (recursive)
│    │    └─Conv2d (cls_logits)                    [2, 3, 100, 100]          (recursive)
│    │    └─Conv2d (bbox_pred)                     [2, 12, 100, 100]         (recursive)
│    │    └─Sequential (conv)                      [2, 256, 50, 50]          (recursive)
│    │    └─Conv2d (cls_logits)                    [2, 3, 50, 50]            (recursive)
│    │    └─Conv2d (bbox_pred)                     [2, 12, 50, 50]           (recursive)
│    │    └─Sequential (conv)                      [2, 256, 25, 25]          (recursive)
│    │    └─Conv2d (cls_logits)                    [2, 3, 25, 25]            (recursive)
│    │    └─Conv2d (bbox_pred)                     [2, 12, 25, 25]           (recursive)
│    │    └─Sequential (conv)                      [2, 256, 13, 13]          (recursive)
│    │    └─Conv2d (cls_logits)                    [2, 3, 13, 13]            (recursive)
│    │    └─Conv2d (bbox_pred)                     [2, 12, 13, 13]           (recursive)
│    └─AnchorGenerator (anchor_generator)          [159882, 4]               --
├─RoIHeads (roi_heads)                             [100, 4]                  --
│    └─MultiScaleRoIAlign (box_roi_pool)           [2000, 256, 7, 7]         --
│    └─FastRCNNConvFCHead (box_head)               [2000, 1024]              --
│    │    └─Conv2dNormActivation (0)               [2000, 256, 7, 7]         590,336
│    │    └─Conv2dNormActivation (1)               [2000, 256, 7, 7]         590,336
│    │    └─Conv2dNormActivation (2)               [2000, 256, 7, 7]         590,336
│    │    └─Conv2dNormActivation (3)               [2000, 256, 7, 7]         590,336
│    │    └─Flatten (4)                            [2000, 12544]             --
│    │    └─Linear (5)                             [2000, 1024]              12,846,080
│    │    └─ReLU (6)                               [2000, 1024]              --
│    └─FastRCNNPredictor (box_predictor)           [2000, 4]                 --
│    │    └─Linear (cls_score)                     [2000, 4]                 4,100
│    │    └─Linear (bbox_pred)                     [2000, 16]                16,400
==========================================================================================
Total params: 43,266,403
Trainable params: 43,041,059
Non-trainable params: 225,344
Total mult-adds (G): 559.91
==========================================================================================
```

4.2 Machine Learning

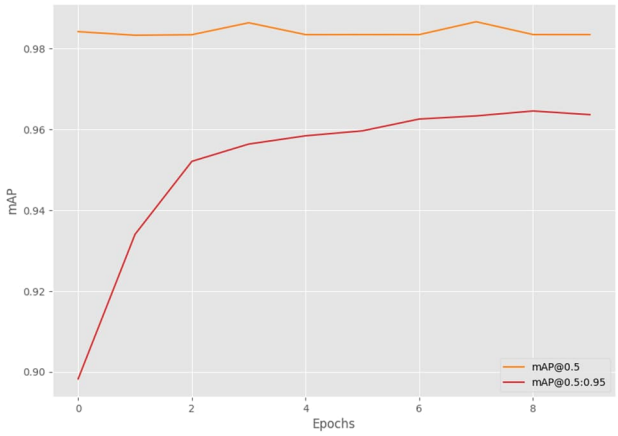
Training is done in 500 iterations with batches of 2 on a set of 1000 images. Validation takes place in 10 or 15 epochs on a set of 100 images.

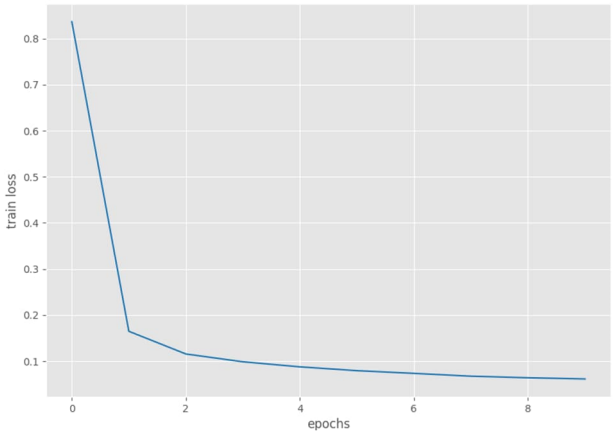
The total loss function here actually is an addition of 4 loss functions:

- Classification loss
- Box regression loss
- Objectness loss
- Region proposal network loss

For validation we use the mean average precision metric.

4.2.1 Training on Complexity 0



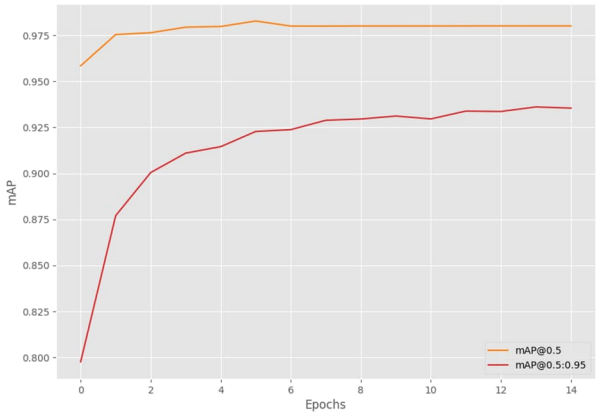


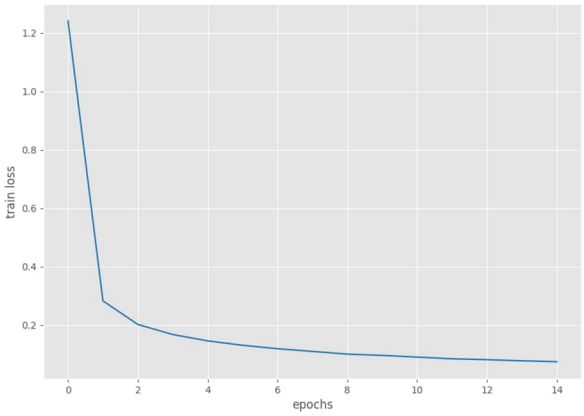
epoch	map	map_05	loss	cls loss	box reg loss	obj loss	rpn loss
1	0.8983	0.9842	0.8365	0.3553	0.4085	0.0529	0.0199
2	0.9340	0.9833	0.1649	0.0567	0.0977	0.0020	0.0085
3	0.9521	0.9834	0.1157	0.0378	0.0699	0.0011	0.0069
4	0.9564	0.9864	0.0987	0.0319	0.0600	0.0007	0.0061
5	0.9584	0.9835	0.0877	0.0277	0.0539	0.0006	0.0056
6	0.9597	0.9835	0.0795	0.0245	0.0491	0.0005	0.0054
7	0.9626	0.9835	0.0736	0.0227	0.0455	0.0004	0.0051
8	0.9634	0.9867	0.0675	0.0205	0.0419	0.0003	0.0048
9	0.9646	0.9835	0.0641	0.0192	0.0399	0.0003	0.0047
10	0.9637	0.9835	0.0615	0.0184	0.0384	0.0003	0.0044

Figure 18, Precision and loss per epoch

➔ https://github.com/jkcas22/shape-recognition/blob/main/faster_rcnn_training-0.ipynb

4.2.2 Training on Complexity 1



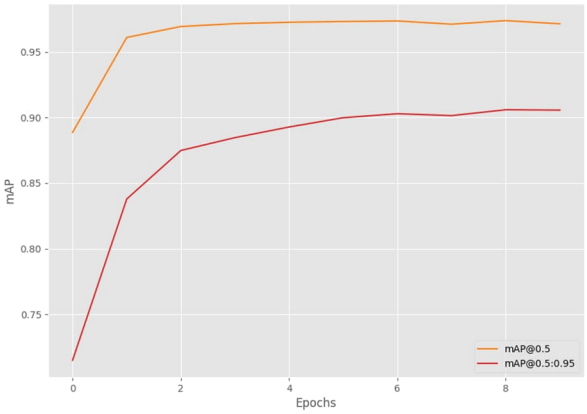


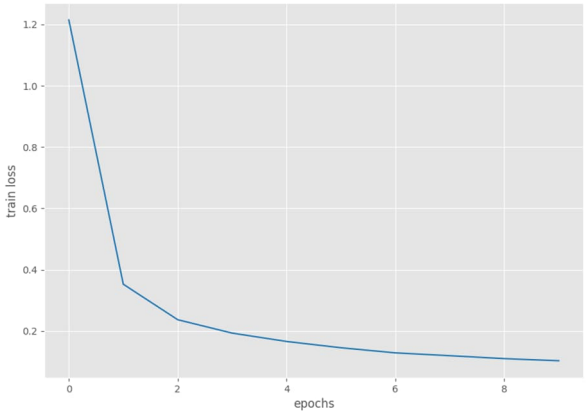
epoch	map	map_05	loss	cls loss	box reg loss	obj loss	rpn loss
1	0.7976	0.9584	1.2408	0.4796	0.4811	0.2331	0.0470
2	0.8770	0.9754	0.2828	0.1036	0.1489	0.0144	0.0159
3	0.9004	0.9764	0.2021	0.0733	0.1096	0.0068	0.0124
4	0.9110	0.9794	0.1678	0.0600	0.0929	0.0041	0.0107
5	0.9145	0.9798	0.1460	0.0509	0.0827	0.0027	0.0097
6	0.9228	0.9828	0.1309	0.0454	0.0745	0.0019	0.0091
7	0.9237	0.9800	0.1191	0.0403	0.0688	0.0015	0.0084
8	0.9288	0.9800	0.1099	0.0361	0.0647	0.0011	0.0080
9	0.9295	0.9801	0.1005	0.0318	0.0601	0.0009	0.0077
10	0.9311	0.9801	0.0962	0.0305	0.0578	0.0007	0.0072
11	0.9296	0.9801	0.0904	0.0283	0.0545	0.0006	0.0069
12	0.9338	0.9801	0.0846	0.0255	0.0518	0.0006	0.0067
13	0.9336	0.9801	0.0815	0.0245	0.0500	0.0005	0.0065
14	0.9361	0.9801	0.0778	0.0232	0.0479	0.0005	0.0062
15	0.9355	0.9801	0.0745	0.0221	0.0460	0.0004	0.0060

Figure 19, Precision and loss per epoch

➔ https://github.com/jkcas22/shape-recognition/blob/main/faster_rcnn_training-1.ipynb

4.2.3 Training on Complexity 2



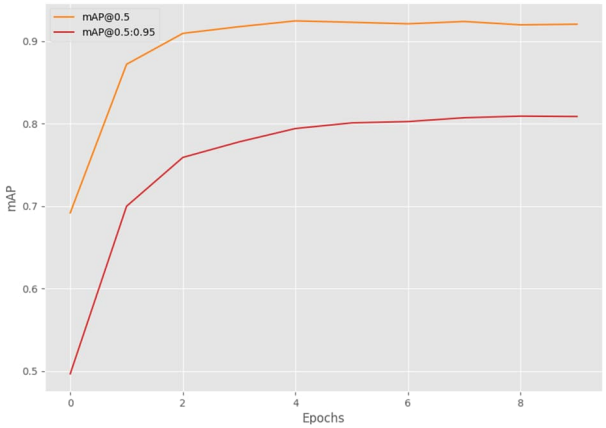


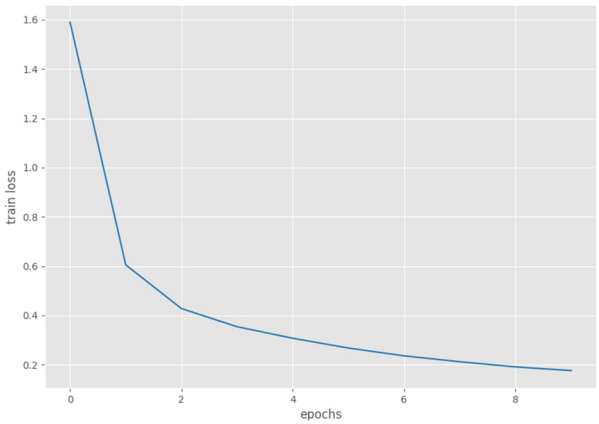
epoch	map	map_05	loss	cls loss	box reg loss	obj loss	rpn loss
1	0.7151	0.8886	1.2145	0.4771	0.4562	0.2361	0.0452
2	0.8379	0.9609	0.3527	0.1432	0.1761	0.0133	0.0201
3	0.8749	0.9693	0.2368	0.0917	0.1230	0.0068	0.0154
4	0.8846	0.9715	0.1932	0.0726	0.1031	0.0043	0.0132
5	0.8927	0.9725	0.1659	0.0593	0.0915	0.0032	0.0119
6	0.8998	0.9731	0.1454	0.0504	0.0816	0.0026	0.0108
7	0.9028	0.9735	0.1287	0.0425	0.0742	0.0018	0.0102
8	0.9014	0.9710	0.1194	0.0379	0.0703	0.0016	0.0096
9	0.9059	0.9738	0.1099	0.0337	0.0658	0.0013	0.0090
10	0.9056	0.9713	0.1030	0.0314	0.0618	0.0011	0.0085

Figure 20, Precision and loss per epoch

➔ https://github.com/jkcas22/shape-recognition/blob/main/faster_rcnn_training-2.ipynb

4.2.4 Training on Complexity 3





epoch	map	map_05	loss	cls loss	box reg loss	obj loss	rpn loss
1	0.4968	0.6919	1.5896	0.6047	0.5193	0.3927	0.0729
2	0.6999	0.8720	0.6051	0.2495	0.2639	0.0582	0.0335
3	0.7591	0.9095	0.4278	0.1736	0.1913	0.0371	0.0258
4	0.7780	0.9176	0.3542	0.1432	0.1622	0.0266	0.0222
5	0.7941	0.9246	0.3073	0.1224	0.1449	0.0200	0.0200
6	0.8010	0.9229	0.2677	0.1053	0.1300	0.0140	0.0183
7	0.8026	0.9211	0.2361	0.0911	0.1186	0.0093	0.0171
8	0.8072	0.9239	0.2120	0.0785	0.1108	0.0067	0.0160
9	0.8091	0.9198	0.1911	0.0684	0.1033	0.0044	0.0150
10	0.8087	0.9206	0.1763	0.0613	0.0968	0.0039	0.0142

Figure 21, Precision and loss per epoch

➔ https://github.com/jkcas22/shape-recognition/blob/main/faster_rcnn_training-3.ipynb

5 Results Discussion

5.1 Model Evaluation

The 4 trained models are evaluated with a test set of another 100 images for each complexity. We use mean average precision and mean average recall metrics for the assessment of the models. In addition, the metrics are also evaluated per class.

5.1.1 Evaluation on Complexity 0

For complexity 0 we have quite a good model with a precision of 97%.

Class	AP	AR
2	0.983	0.989
3	0.951	0.964
4	0.968	0.976
Avg	0.968	0.976

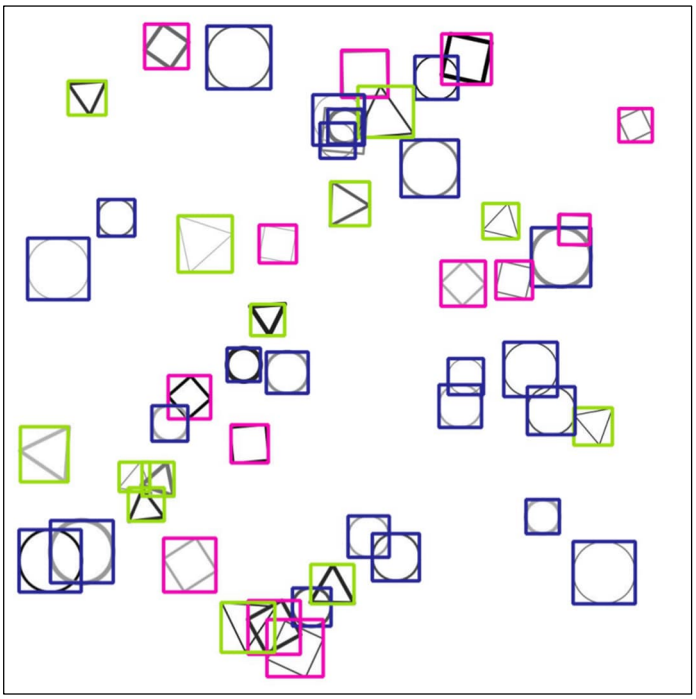


Figure 22, Sample image processing

5.1.2 Evaluation on Complexity 1

By adding variation to the background, the precision drops to 94%.

Class	AP	AR
2	0.97	0.976
3	0.907	0.928
4	0.931	0.946
Avg	0.936	0.95

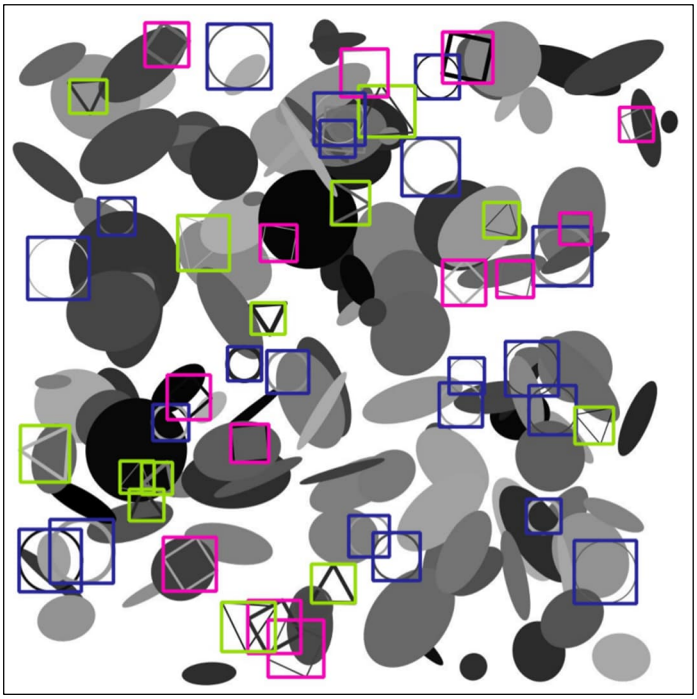


Figure 23, Sample image processing

5.1.3 Evaluation on Complexity 2

By adding noise in the foreground, the precision drops to 91%.

Class	AP	AR
2	0.97	0.978
3	0.852	0.884
4	0.913	0.933
Avg	0.912	0.932

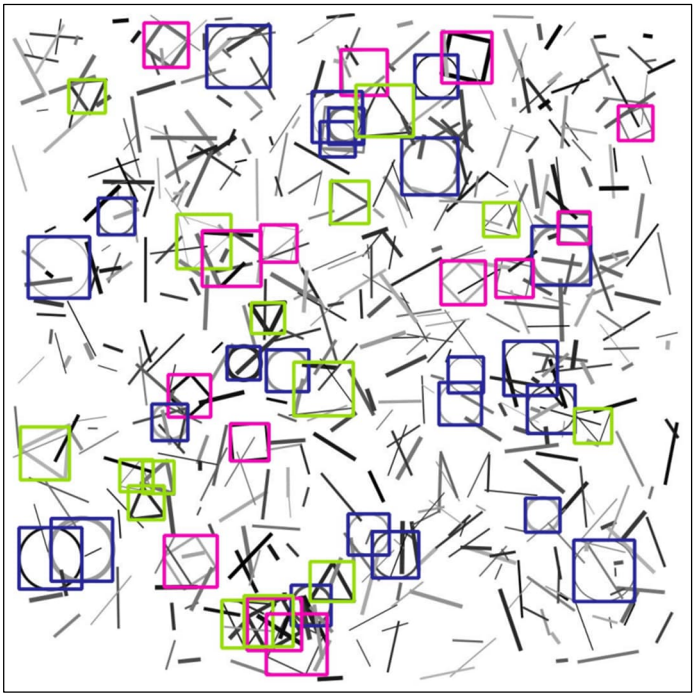


Figure 24, Sample image processing

5.1.4 Evaluation on Complexity 3

Finally, by adding to kinds of disturbances the precision drops significantly to 82%.

Class	AP	AR
2	0.919	0.936
3	0.726	0.772
4	0.819	0.848
Avg	0.821	0.852

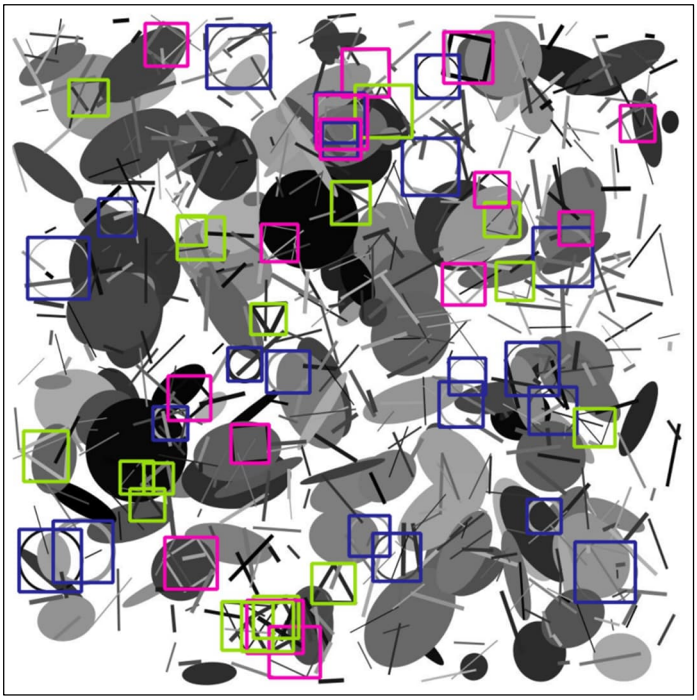


Figure 25, Sample image processing

5.1.5 Evaluation per Class

We can extract a clear common pattern from the tables in the preceding sections: The more rotation symmetry a shape has, the better the precision and recall for its class gets.

5.2 Cross Evaluation

It is also interesting to see how the trained model on complexity 3 is performing on test sets of lower complexity. The following matrix shows the precision of all models evaluated on all other test sets.

Complexity \ Model	0	1	2	3
0	**0.968**	0.964	0.958	0.95
1	0.587	**0.936**	0.649	0.912
2	0.748	0.788	**0.912**	0.908
3	0.341	0.686	0.502	**0.821**

Clearly model 3 manages the lower complexities quite well.

From this table we can see that models 1 and 0 are quite good in complexity 2 already, but models 2 and 0 lose quite a bit of precision on complexity 1, hence noise of type 1 (ellipses in the background) seems to be harder to handle by our ground model ResNet 50 FPN V2, than noise of type 2.

Probably it would also be interesting to have a new model, that is trained on the complete set of images, with different complexities mixed.

6 Conclusion and Outlook

We have successfully applied Faster R-CNN object detection to detect simple shapes on our own synthetic images. With increased image complexity, we still have a good precision of 82%. But this result still has potential to be improved.

For further improvement in the object detection, we could try a few more things:

- Try other base models in the very setting of this project:
  Instead of using ResNet 50 FPN V2, there are some other working models [18]:
  https://github.com/sovit-123/fasterrcnn-pytorch-training-pipeline/tree/main/models
- Try Mask R-CNN [21], the successor of Faster R-CNN. It would be fun to see the image segmentation by Mask R-CNN, i.e., to see the pixels of the shapes in specific colors.
- Besides R-CNN [2], i.e., Region-based Convolutional Neural Networks, also YOLO [22] from the single-shot detector family could be applied. It would be very interesting to compare the two approaches.

We also observed that by using generic synthetic images, we can on hand gradually increase the images visual complexity, and on the other hand observe the behavior of a specific model in the training and evaluation along the variation of the images.

Acknowledgements

First, I want to thank Christine and Emilie for their patience when I get lost in machine learning and in time and space in front of my computer.

Many thanks also go to Sigve, Mykhailo, and all the other docents, and students of the CAS, for all the intellectual and other fun times we had together at quite different geographic locations.

References

[1] https://en.wikipedia.org/wiki/Object_detection, Neural network approaches for object detection

[2] https://arxiv.org/abs/1506.01497, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun

[3] https://www.python.org, Python programming language

[4] https://jupyter.org/, Jupyter Notebook

[5] https://colab.research.google.com, Google Colab

[6] https://pandas.pydata.org, Pandas data analysis and manipulation tool

[7] https://numpy.org, Scientific computing with Python

[8] https://matplotlib.org, Visualization with Python

[9] https://pytorch.org/, PyTorch Library

[10] https://github.com, Software development and version control

[11] https://towardsdatascience.com/object-detection-with-neural-networks-a4e2c46b4491, Media Post, Johannes Rieke

[12] https://github.com/jrieke/shape-detection#readme, GitHub project, Johannes Rieke

[13] https://github.com/jkcas22/geomsha/tree/main/m3#readme GitHub project, Jürg Krähenbühl

[14] https://numpy.org/doc/stable/reference/random/index.html, NumPy Random Sampling

[15] https://github.com/jkcas22/shape-recognition/blob/main/README.md, Detection of simple shapes with Faster R-CNN, Jürg Krähenbühl

[16] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/devkit_doc.pdf, Pascal VOC2012 Data Format

[17] https://debuggercafe.com/object-detection-using-pytorch-faster-rcnn-resnet50-fpn-v2/, Object Detection using PyTorch Faster RCNN ResNet50 FPN V2, Sovit Ranjan Rath

[18] https://pytorch.org/vision/main/models/faster_rcnn.html, PyTorch Faster R-CNN

[19] https://github.com/sovit-123/fasterrcnn-pytorch-training-pipeline, A Simple Pipeline to Train PyTorch FasterRCNN Model, Sovit Ranjan Rath

[20] https://blog.paperspace.com/faster-r-cnn-explained-object-detection/, Faster R-CNN Explained for Object Detection Tasks, Ahmed Fawzy Gad

[21] https://arxiv.org/abs/1703.06870, Mask R-CNN, Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick

[22] https://arxiv.org/abs/1506.02640, You Only Look Once: Unified, Real-Time Object Detection, Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi