Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Region Proposal Network for Simple Objects in Grasping Experiments

Ruben Bauer

Studiengang:	Informatik
Prüfer/in:	Dr. Jim Mainprice
Betreuer/in:	Janik Hager, M.Sc.
Beginn am:	17. Juni 2020
Beendet am:	17. Dezember 2020

Abstract

Particularly in applied robotics, the grasping of objects is a major field which comes with various difficulties. Multiple objects with simple or complex shapes as well as different colors can be scattered on a surface in random positions and orientations. However, with the knowledge about the correct object positions, a robot has high chance of grasping them. Object detection systems can determine bounding boxes of objects in images, which can help to calculate the correct object positions. Current state-of-the-art object detection systems such as the popular Faster R-CNN and the Mask R-CNN, often use multi-stage architectures. Both models utilize a region proposal network to obtain regions which are likely to contain objects. This thesis introduces and evaluates multiple architecture variations of single-stage and two-stage models. These variations include a region proposal network, yet in the setting of grasping experiments. Usually, the training of these models is done in a supervised manner which requires lots of data with ground truth information. Generating this kind of data in a real world environment is expensive, yet it is cost-efficient to generate the same kind of data in a simulated environment. Therefore, this thesis introduces a framework to generate artificial data in a simulated grasping experiment environment. This framework implements several domain randomization techniques in order to randomize this simulation environment. The training data contains only artificial images with objects of simple geometry. The results have shown that models, which were trained only on these artificial images, can still generalize well to images of a real environment. Furthermore, the generalization to images which contain objects of complex geometry is equally possible. This thesis performs ablation studies on the employed domain randomization techniques which reveal both degradation and improvement of different techniques. Benchmarks on the model variations show that a significantly faster inference is possible compared to the originally Faster R-CNN and Mask R-CNN, while still achieving pleasant prediction results. This was made possible by using different configurations for the region proposal network, and by introducing faster feature extraction backbone architectures.

Kurzfassung

Insbesondere in der angewandten Robotik ist das Greifen von Objekten ein großes Forschungsgebiet, welches mit verschiedenen Schwierigkeiten verbunden ist. Verschiedene Objekte mit einfachen oder komplexen Formen sowie in unterschiedlichen Farben können in zufälligen Positionen und Orientierungen auf einer Oberfläche verteilt sein. Sollten jedoch die korrekten Objektpositionen bekannt sein, dann kann ein Roboter diese mit hoher Wahrscheinlichkeit Greifen. Objekterkennungssysteme können Objekte in Bildern erkennen und ihre Bounding Boxen bestimmen, welche die Objekte auf den Bildern einrahmen. Diese Informationen werden oft genutzt, um dann echte Objektpositionen zu berechnen. Aktuelle Objekterkennungssysteme, wie das bekannte Faster R-CNN und das Mask R-CNN, verwenden häufig mehrstufige Architekturen. Beide Modelle verwenden unter anderem ein Region Proposal Netzwerk, welches wichtige Vorarbeit leistet, um aus einem Bild Regionen zu bestimmen, die mit hoher Wahrscheinlichkeit Objekte enthalten. In dieser Arbeit werden mehrere Architekturvarianten von einstufigen und zweistufigen Objekterkennungsmodellen entwickelt und diese im Kontext von Greifexperimenten ausgewertet. Diese Modelle verwenden ebenfalls ein Region Proposal Netzwerk. Das Trainieren solcher Modelle erfordert in der Regel große Mengen an Trainingsdaten, die korrekte Ground-Truth Informationen beinhalten. Das Generieren solcher Daten ist in der echten Welt teuer und aufwändig. Abhilfe können Simulationsumgebungen schaffen, welche es erlauben, kostengünstig große Mengen an künstlicher Trainingsdaten zu generieren. In dieser Arbeit wurde ein Framework entwickelt, welches zur Erzeugung künstlicher Trainingsdaten im Kontext von Greifversuchen, genutzt werden kann. Das Framework implementiert verschiedene Domain Randomization Techniken, um diese Simulationsumgebung so zufällig wie möglich zu gestalten. Die generierten Trainingsdaten enthalten dabei ausschließlich künstliche Bilder mit Objekten einfacher Geometrie, welche genutzt wurden, um die verschiedenen Modellvarianten zu trainieren. Die Ergebnisse haben gezeigt, dass diese Modelle, die nur auf diesen künstlichen Bildern trainiert wurden, immer noch gut sowohl simple als auch komplexe Objekte in Bildern einer ähnlichen, aber realen Umgebung erkennen können. Außerdem wurden verschiedene Experimente durchgeführt, um die Effekte der eingesetzten Domain Randomization Techniken zu untersuchen. Benchmarks für die verschiedenen Modellvarianten haben gezeigt, dass das Erkennen von Objekten auf Bildern, im Vergleich zu den ursprünglichen Faster R-CNN und Mask R-CNN Modellen, deutlich beschleunigt werden kann, ohne dabei große Verluste in der Genauigkeit der Objekterkennung in Kauf nehmen zu müssen. Dies wurde unter anderem durch die Entwicklung eines kleinen Autoenkodierer Netzwerkes, dessen vortrainierter Enkodierer für das Extrahieren von Features verwendet wurde, möglich, sowie durch die Anpassung der Konfigurationen für das Region Proposal Netzwerkes auf den Kontext von Greifexperimenten.

Contents

1 Introduction

This thesis introduces different model architectures which utilize a region proposal network for object detection and evaluates the different architectures in the setting of a grasping experiment. These model architectures are supposed to be trained only on data from a simulated environment. Although, they should generalize well on previously unseen data from a real environment. Furthermore, the models should be able to detect objects of simple geometry, as well as arbitrary objects with more complex geometry, correctly. The models should be practical in order to be used in real-time applications.

This chapter provides a short motivation to the ambitions in section 1.1. It also gives a brief overview on how this thesis is organized as well as a short summary of the main contributions in this thesis in sections 1.3 and 1.2 respectively.

1.1 Motivation

In the field of robotics, the grasping of objects is a major challenge. It is essential to automate many complex processes in which objects have to be moved from one position to another. For instance, the automated selection of tools such as a drill or spray flask for further tasks, or an automated sorting process in which a pile of different objects should be arranged correctly. With the knowledge about correct object positions a robot would be able to grasp them. Object detection systems can determine bounding boxes of objects in images, which can help to calculate the correct object positions. However, the system must correctly identify objects and their locations. This can be difficult as the exact object positions are often not known beforehand, even though they usually are located on a specific surface area as for instance a table. The actual objects are also unknown. Thus, they can come in the form of simple or complex shapes as well as each arbitrary color. Distinguishing the objects from the background is often difficult, since the background might be textured and therefore appears to contain objects even though it does not. Cameras and sensors are usually attached to robots in order to scan the environment by capturing it with RGB images or depth images. Hence, the camera position and gaze can change during movements which further increases the difficulty of detecting objects as the appearance of them as well as of the background changes with the perspective of the cameras. This motivates the first main objective of this thesis: developing an object detection model which can successfully and efficiently detect and locate simple as well as complex objects in the setting of grasping experiments.

Current state-of-the-art results in the task of object detection have been achieved by powerful neural networks. Some of them utilize special region proposal network architectures. Training neural networks is usually done in a supervised manner which requires lots of data with ground truth information. Generating this kind of data in a real world environment is expensive, yet it is cost-efficient to generate the same kind of data in a simulated environment. This motivates the second main objective of this thesis: developing a simulated grasping experiment environment to generate artificial training data which allows to train an object detection model that utilizes a region proposal network architecture. The trained model should be able to also generalize well to data from a corresponding real world environment.

1.2 Contributions

The core contributions of this thesis are the following:

1. The development of a simulation framework which allows to simulate the environment of a grasping experiment and generate artificial datasets. The simulation framework implements multiple domain randomization techniques. It is covered in section 4.1.
2. The development of a large variety of models, which all utilize a region proposal network, see section 5.2.
3. Ablation studies were performed and together with the evaluation of the developed models they reveal informative insights on the effect of different domain randomization techniques (subsection 5.2.2) and how different model architectures perform (subsections 5.2.1 and 5.2.1).
4. An auto-encoder architecture was developed to utilize its pretrained encoder part as feature extraction network, see section 4.2.2. This enables really fast and accurate object detection, see subsection 5.2.1.
5. A training, evaluation, and visualization pipeline was developed to ease and accelerate the development of new models.

1.3 Organization

This thesis is organized as follows. Chapter 2 covers related work on the subjects covered in this thesis, which are “object detection,” “grasping experiments,” and “domain randomization techniques.” Chapter 3 provides background information which allows to build a theoretical foundation for understanding the developed and used methods in this thesis. It covers the basics of neural networks, the operating principle of the used region proposal network and object detection systems, as well as core concepts on how to transfer models from a simulated environment to a real world environment. The developed and used methods are presented in chapter 4. It goes into detail about the developed simulation framework; the different architectures, components, and algorithms that are used by the developed models; and how the developed models were trained and evaluated. Chapter 5 presents the performed experiments and investigates the evaluation results on the developed models w.r.t. several metrics. Finally, chapter 6 gives a brief discussion on the summarized results, some hints on alternative approaches, and a final conclusion.

2 Related Work

The subjects covered in this thesis are based on and related to a considerable amount of previous work. They can be divided in the following three topics: “Object Detection,“ “Grasping Experiments,“ and “Domain Randomization.“ Each topic comes with a long history of research, hence, just a few can be briefly referred to and only a small selection of them will be mentioned in detail in the following sections.

Section 2.1 covers some related work on the topic of object recognition. There exist different methods to the problem of object detection, yet the most recent and common ones are approaches using either multi-stage or single-stage network models. The section covers related work on multi-stage models in detail as it is the model type of choice in this thesis.

While the major focus of this thesis lies on object detection and domain randomization, section 2.2 provides a small outlook to work which relates to the overall goal, the grasping of objects. One approach to solving this problem proposes similar architectures for generating grasp predictions as it is for instance used in this thesis for object detection, thus skipping the step of object detection or implicitly solving it by defining a more specific objective.

Domain randomization plays a big role for machine learning methods and can also assist to solve problems of transfer learning. It is a major objective in this thesis to develop a good model which has been trained on synthetic data only, where utilizing domain randomization techniques seems to be promising. Section 2.3 is concerned with related work on this topic and presents two distinct approaches to domain randomization.

2.1 Object Detection

Object detection is a major field in computer vision and has continuously grown in relevance due to many breakthroughs in recent years. There are many applications for object detection, such as autonomous driving, face recognition or grasping experiments in robotics. Most recent object detectors can be divided in either single-stage detectors or multi-stage (mostly two-stage) detectors. Multi-stage detector usually perform better in object localization and object classification but lack in inference speed due to expensive processing steps between stages, whereas single-stage detectors are far superior when it comes to inference speed but often lack in localization and recognition performance. A common characteristic for multi-stage detectors is an additional stage generating region proposals, usually defined as rectangular bounding boxes. They define a region on the original image which suggests the existence of a possible object in this region. Multi-stage detectors often implement this step and use these proposals as additional information when performing RoI-Pooling 3.2.2. In contrast, single-stage detectors omit this step and directly predict the bounding boxes on the input image. [13]

A well known multi-stage detectors is for example the R-CNN proposed by Girshick et al. [8]. It was the first region-based CNN to outperform all of the previous models on VOC 2012 [5], and was a breakthrough in image detection, as it increased the baseline performance by more than 30%. On the basis of the R-CNN, Girshick published an improved version called Fast R-CNN [7] with drastically reduced training and inference time as well as a slightly better performance on VOC 2012. Shortly after, Ren et al. proposed the Faster R-CNN [25], a likewise further improved version of the Fast R-CNN in which a region proposal network (RPN) has been introduced for the first time. The Faster R-CNN has been designed for the task of object localization and object classification; however, its architecture serves as base for future models. For instance, the Mask R-CNN which has been proposed by He et al. [9], and augments the capabilities of the Faster R-CNN by additionally performing instance segmentation and keypoint detection.

Since the Faster R-CNN serves as foundation for the Mask R-CNN, the model of choice in this thesis, more of their core contributions are elaborated in subsections 2.1.1 and 2.1.2 respectively. However, their specific architectures are further presented in subsections 3.2.5 and 3.2.6.

Single-stage predictors do not use region proposals and predict the bounding boxes from the input image directly. A popular single-stage predictor is YOLO (You Only Look Once) [22], proposed by Redmon et al., which focuses on inference speed and real-time detection, see 2.1.3 for more detail. Further improved versions are the YOLOv2 [23]. It even outperforms multi-stage models like the Faster R-CNN on VOC 2012, while still having significantly faster inference. Another version is the YOLOv3 [24], which uses for example a new backbone for more robust feature extraction compared to YOLOv2. Other known single-stage methods are SSD [18], DSSD [6], RetinaNet [16], M2Det [36], and RefineDet [35], as presented in [13].

2.1.1 Faster R-CNN

The Faster R-CNN network [25] has been proposed by Ren et al. and has been a breakthrough in object detection. At the time, it achieved state-of-the-art accuracy in object detection and did run on 7 frames per seconds (fps), which made it significantly faster compared to the previous Fast R-CNN model. Faster R-CNN is a multi-stage model for the task of object detection and object recognition. It builds on top of the previous models, R-CNN [8] and Fast R-CNN [7] and introduces a novel fully convolutional region proposal network (RPN), allowing to predict cheap object boundaries and objectness scores in form of region proposals. These proposals then serve as input for the Fast R-CNN module. In order to enable the prediction of different sized objects on potentially different sized images, the Faster R-CNN uses so called anchor boxes which exist in different scales and aspect ratios. A technique called RoI-Pooling, see 3.2.2, generates fixed sized feature maps from the region proposals where the anchors are being applied on. In Faster R-CNN, the RPN and the Fast-RCNN have been merged into one single network, sharing convolutional features and allowing the Faster R-CNN to be trained end-to-end. Its architecture is explained in section 3.2.5 in detail.

The Faster R-CNN is mainly used for the task of object detection in a general setting. Usually, the input images describe complicated environments that consist of many complex objects. In contrast, object detection in this thesis is applied to the setting of grasping experiments. For this reason, there are less objects, the objects have simple shapes, and the classification head is reduced to predict for binary object or no object, as it is described in subsection 4.2.2. Thus, the models discussed here are class-agnostic classifier which allow to generalize to objects of complex shapes that have not been seen before.

2.1.2 Mask R-CNN

The Mask R-CNN [9], which has been proposed by He et al., represents a framework for object instance segmentation. By using the Faster R-CNN as core architecture, He et al. showed in their approach how it could be easily extended for other tasks, e. g., instance segmentation and human pose estimation. At the time, it outperformed all other existing models for each of these tasks and provides a good example on how to extent given models to different objectives in a modular manner. Moreover, they added an additional branch for predicting the segmentation mask of an object without changing the existing branch used for object detection. The mask branch is a small fully convolutional network; which can be easily implemented and adds only a little computational overhead. The Mask R-CNN runs with 5 fps, while the Faster R-CNN runs with 7 fps. It can be trained by using the Faster R-CNN framework, in which the mask-task is added as additional loss to the multi-task loss of the Faster R-CNN. A small adjustment to the implementation is the usage of RoI Align instead of RoI Pooling (see subsection 3.2.2). It allows to produce output masks that are correctly aligned with the input. This improved the mask loss by a significant margin, given that a correct alignment has not been relevant for the task of object detection in the Faster R-CNN.

Although the main objective in this thesis lies on object detection rather than on instance segmentation, the model of choice is still the Mask R-CNN. Particularly because it provides an easy way to additionally perform instance segmentation while still predicting bounding boxes and classes. Also, [9] showed, that the additional task of instance segmentation even leads to slightly better results on the task of object detection. The architecture of the Mask R-CNN is presented in detail in subsection 3.2.6.

2.1.3 You Only Look Once (YOLO)

Compared to multi-stage models like the Faster-RCNN or the Mask R-CNN, single-stage models like YOLO [22] significantly outperform the known multi-stage models in terms of inference speed. In YOLO, object detection is formulated as a regression problem where bounding boxes and class probabilities are directly predicted from the input image's feature map. Even though it has a worse localization accuracy than for example the Faster R-CNN, it outputs far less false positive predictions and runs with over 45 fps, compared to the 7 fps for the Faster R-CNN. This makes its use attractive for real-time applications.

A more recent version is the YOLOv2 [23]. It introduces multiple improvements and greatly increases the detectors accuracy while still keeping most of its speed. It achieves a inference speed of up to 91 fps. The focus of this thesis lies on evaluating models using a region proposal network, in particular the Mask R-CNN. However, bearing the knowledge about real-time detectors in mind, also different approaches are tested to speed up the existing multi-stage model, as it is presented in section 4.2.2.

2.2 Grasping Experiments

Object detection has likewise a field of application in robotic vision. There exist different approaches to tackle the problem of grasping objects. For example, Schmidt et al. [27] proposed a method to grasp unknown objects by predicting grasp configurations from depth images. Their comparatively small neural network takes a depth image as input which is then processed by two convolutional layers. This step is followed by two fully connected layers that finally output a single grasp configuration.

Real-World Multi-Object, Multi-Grasp Detection

Chu et al. [1] introduces a deep learning architecture to predict possibly multiple grasping positions for objects in the input image. Their approach uses RGB-D images as input. Similar to [27], they do not compute the object locations explicitly but directly tackle the overall grasping problem for general and unseen objects. However, [27] outputs the grasp in camera coordinates with roll, pitch and yaw values, a total of 6 parameters, whereas [1] outputs an grasp oriented bounding box with a total of 5 parameters encoding one grasp position. This limits the grasps to the normal vector of the image plane. Their network architecture is similar to the Faster R-CNN and also utilizes a region proposal network which is trained to output grasp region proposals. These grasp region proposals are then further refined in the network head to contain an additional orientation value and slightly adjusted bounding box parameters to match the best grasp positions as closely as possible. Due to the architecture, the network allows predicting multiple grasps for multiple objects in the input almost by default.

In[1], training and testing were both performed on real data. In contrast, in this thesis, the models are trained on only synthetic data. Furthermore, this thesis focuses on object detection and predicting axis aligned bounding boxes for the objects in the input image, instead of predicting grasp positions directly.

2.3 Domain Randomization

In order to avoid the overfitting of deep neural networks, it is necessary to provide a large enough dataset with multiple diverse training samples. The amount of existing real data is often not sufficient, yet there exist different techniques to circumvent this problem. Domain randomization is a technique which eases the transfer of models between different domains and improves the overall generalization to other datasets, especially when it comes to transferring models from an simulated environment to the real world. Due to a lack of real world data for the specific setting of object detection in grasping experiments, this thesis thoroughly exploits domain randomization techniques. Their investigation is a major objective of this thesis, see subsection 4.1.5.

Different previous work engage in the topic of domain randomization. A significant work is [29]. It focuses on randomizing the generation of the data and is further covered in subsection 2.3.1.

Whereas [29] and several other scholars pursue the approach of applying almost arbitrary randomizations in order to hopefully increase the variety in data, Zakharov et al. propose in [34] a different approach for domain randomization. It focuses on data augmentation, see subsection 2.3.2.

2.3.1 Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World

In [29], Tobin et al. explore domain randomization to tackle the “reality gap"problem, which describes the difference between the real world and simulation. The core idea is, that “with enough variability in the simulator, the real world may appear to the model as just another variation" [29].

In their work, they investigate this approach w.r.t. the task of object localization and show that it is also applicable for the task of grasping in cluttered environments. They train on simulated RGB images, while testing is done on real world images. Similar to this thesis, they use a setting in which the objects are placed on a table. The randomizations of their simulator include for example the amount and shape of objects; their position, orientation, and texture; the texture of the floor and table; the position, orientation, and field of view of the camera; the amount of lights in the scene; the position, orientation, and specular characteristic of the lights; as well as the type and amount of random noise which is being added to the images. Although this thesis adopts many of their proposed randomizations, there are some minor differences. For instance, the fact that objects do not have a texture and are plain-colored; as well as the camera’s orientation and fixed field-of-view. Its position and view direction is randomized. Due to the used framework, only one light exists. It is also noteworthy that adding random noise to the images is no part of the rendering but happens separately during training together with many other image augmentations.

Tobin et al. uses a modified version of the VGG16 [28] to predict 3D-coordinates of the objects of interest. The performance on real world data was found to be promising and although some overfitting exists, the results are comparable to other traditional techniques. They also discovered that using pretrained weights is not necessary for achieving good performance, as training from scratch yields to similar good results, if enough training samples are being used. However, the use of pretrained weights still leads to less iterations needed for convergence. The results have also shown, that texture randomizations seem to be more important than for instance the randomization of object positions, particularly in cases in which only few data is used. Furthermore, adding small amount of noise makes training more robust against local minima and leads to faster convergence.

In this thesis, many of their techniques for domain randomization have been adapted and evaluated by using the powerful Mask R-CNN network.

2.3.2 DeceptionNet: Network-driven Domain Randomization

As mentioned above, most applications of domain randomization techniques use either the trial-and-error approach or orient themselves on previous works which have successfully implemented randomizations. Zakharov et al. propose in [34] a novel approach to close the reality gap by introducing a method to learn “useful augmentations which maximize the uncertainty of the output."

They use a deception network to find the most destructive augmentations w.r.t. the task architecture, which is in their case a recognition net for class and pose estimation. This has been achieved by using an alternating optimization scheme with two alternate phases during training. In the first phase, the weights of the recognition net are frozen and the weights of the deception net are updated w.r.t. the maximized recognition net objective. This first phase leads to an increasing amount of variance in the images produced by the deception net. In the second phase, the weights of the deception net are frozen and the weights of the recognition net are updated w.r.t. its minimized objective, yielding a more and more robust recognition net against these domain changes.

Their novel approach yields comparable performance to other existing techniques and shows a significantly better generalization ability of their recognition network.

In contrast to [29], this approach performs data augmentation, i.e., modifies or creates data based on already existing data and does not change the way the data is generated in the first place, e. g., by a renderer and a simulation.

In this thesis, multiple data augmentation techniques are used to induce and increase the effect of domain randomization, as described in subsection 4.2.3. However, the augmentations are hand-selected, applied randomly and in consequence, not learned.

3 Background

This chapter covers relevant background information to build a theoretical foundation for understanding the different methods presented in chapter 4.

The Mask R-CNN can be understood as big neural network. Neural networks have shown to be quite promising in solving different tasks. What needs to be understood when talking about neural networks and what the core concepts are, is covered in section 3.1.

The model used, is the Mask R-CNN [9]. It expands on multiple previous other network architectures and is one of the latest generations in the evolution of R-CNN. Section 3.2 provides background information to the history of the Mask R-CNN and explains different important techniques which have been introduced on this path and are still being used in the modern Mask R-CNN architecture.

Section 3.3 goes more into the depth of domain randomization and data augmentation techniques. It covers the core ideas and shows examples of where and how these can be used.

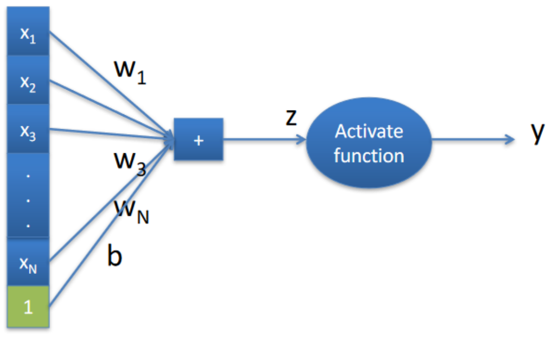
3.1 Neural Networks

Artificial neural networks mimic the processing of information in the human brain, which occurs via multiple neural connections between neurons. They have proven to be great function approximators and have been used to successfully tackle many different tasks. In this thesis, the term ‘neural network‘ will from now on always refer to some artificial neural network.

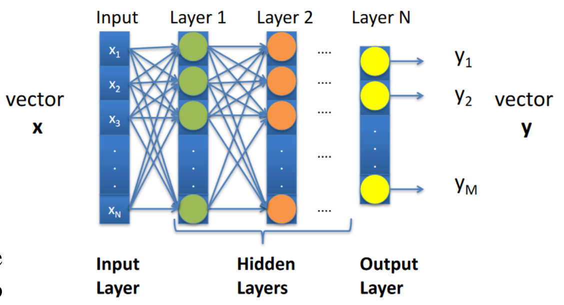
A neural network represents a parameterized function which parameters can be optimized w.r.t. a objective and given data [30]. The data usually contains corresponding input and output samples of the underlying function or distribution that needs to be learned. The universal approximation theorem states that any continuous function can be approximated to a certain degree by a neural network with one hidden layer under certain assumptions to the activation functions. It has been proven in different previous work for sigmoid activations [3] and for Rectified Linear Unit (ReLU) activations [19].

3.1.1 Neurons, Activation Functions, and Optimization

The core elements of neural networks are neurons which perform a linear mapping from some input vector $\vec{x}$ to some scalar output z using some weight vector $\vec{w}$ and add a scalar bias b to the result, thus z can be computed as $z = \vec{w}^T * \vec{x} + b$. The scalar output z serves as input for a non-linear activation function σ, yielding the final output $y = \sigma(\vec{w}^T * \vec{x} + b)$, as depicted in figure 3.1a. A single neuron outputs only one scalar value. Usually, multiple neurons are stacked to form a layer which then can



(a) A single neuron multiplies each element of the input vector with a weight and adds a bias to the result, which is then fed to some activation function that yields the neuron's output which is scalar. Note that the input vector here has been augmented with a scalar 1 and the bias has been absorbed as additional weight such that the neuron's operation (excluding the activation function) still represents a linear mapping. [32]



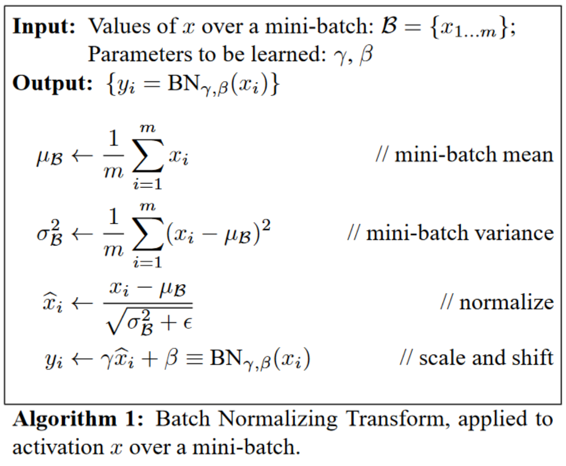
(b) An illustration of a fully connected neural network with multiple hidden layers. Each layer may have a different amount of neurons. [32]

be used to output higher dimensional information. The neuron's transposed weight vectors can be represented as a matrix W. The output of such a layer is computed as $\vec{y} = \sigma(W * \vec{x} + \vec{b})$, where σ is an element-wise operation.

Typically, a loss function is defined on the networks output which represents the loss / error / discrepancy between the expected output and the actual output of the network for some training data. The training data consists of input and expected output tuples. This loss can be used to compute gradients for the different layers via a backpropagation algorithm. There exist different optimization algorithms which use these gradients to compute the weight updates with the objective to minimize the loss [32]. One such optimization algorithm is gradient descent. The idea of gradient descent is to start at a random point of the function to optimize, which corresponds to a random weight initialization, and then follow the slope of the function by taking small steps to a minimum. This is achieved by iteratively computing: $gradient\_step := gradient * learningrate$ and then $new\_weights := old\_weights - gradient\_step$. Gradient descent computes the loss and the gradient to update the weights w.r.t. the complete training data in every iteration. As this is quite unpractical for big datasets, it is more common to use stochastic gradient descent. In each iteration, naive stochastic gradient descent picks one data element by random and uses this to compute the weight updates. However, this induces many weight updates and sometimes even worsens convergence. Therefore, the middle way, in which in each iteration multiple random samples of the training data, called a mini-batch, are chosen to compute the gradient and weight updates, is the preferred way to train neural networks nowadays.

Note that more sophisticated optimization algorithms exist. While they all follow the core idea of gradient descent, they introduce additional mechanisms that can improve learning. See subsection 4.2.4 for the optimization algorithm used in this thesis.

A neural network consisting of only one single layer (only the output layer) is always limited to be linear in the input. However, a neural network with two layers, i. e., one hidden layer and one output layer (not counting the input layer), forms a universal function approximator when certain



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                        // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$       // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                    // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$           // scale and shift
```

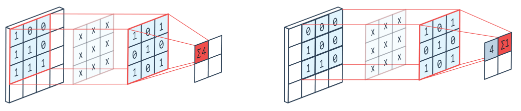
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 3.2: The batch normalization algorithm to normalize activations of hidden layers and to increase the stability of the network. [12]

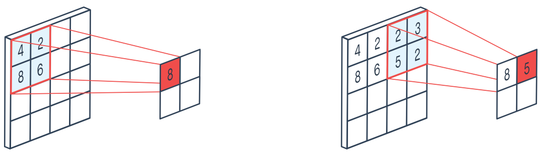
assumptions to the used activation functions hold. As shown in 3.1a, multiplying with a weight and adding a bias can be rewritten as a linear mapping. Since multiple successive linear mappings can be expressed in one single linear mapping, a multi-layer neural network with linear activation functions would be as powerful as a neural network without any hidden layers [32]. The output of any hidden layer can be seen as some intermediate representation of the input, and it might be higher or lower dimensional. Neural networks are often used for feature extraction. A feature is just some property or characteristic of what will be observed and is expressed as a value. The original input itself or any intermediate representation are features.

The input features for neural networks are often normalized in a preprocessing step to improve learning. This is sometimes also done for the output of hidden layers, and is then called **Batch Normalization**. Batch normalization increases the stability of a neural network and introduces a regularization effect. It works by computing the mean and standard deviation per (mini-)batch, and then use this to normalize the batch. By introducing two additional learnable parameters (per layer where batch normalization is being applied), this normalization can be undone by the neural network if needed. This is called denormalization. Figure 3.2 shows the batch normalization algorithm [12].

Two famous activation functions used in neural networks are the sigmoid and the Rectified Linear Unit (ReLU) function. Both have an easy computable derivative which is important for the backpropagation algorithm. The sigmoid function is defined as follows: $\sigma(x) = \frac{1}{1+e^{-x}}$. Its derivative is $\sigma'(x) = \sigma(x) * (1 - \sigma(x))$. The ReLU function is defined as: $ReLU(x) = max(0, x)$. Its derivative is $ReLU'(x) = \begin{cases} 0 & x > 0 \\ 1 & x < 0 \end{cases}$. The derivative of the ReLU function for $x = 0$ is undefined, hence, it is often chosen to be zero. In deeper neural networks with multiple layers, sigmoid functions are less popular because they reduce the absolute value of the gradient during backpropagation with each layer, due to their derivative. Hence, ReLU activation functions are preferred in deep neural networks.



(a) A two dimensional filter is sled over the input. At every position, each weight is multiplied with the corresponding input value and the results are summed up. No padding is used, thus the output size is a little smaller[1].



(b) A window is sled over the input. At each position, the maximum value in the window is taken[2].

A typical fully connected neural network with multiple layers is depicted in figure 3.1b. Fully connected means that every neuron of one layer is connected to every output value of the previous layer, or to every input value if it is the first layer. A fully connected layer expects the input to be a vector of features. Its output is a vector of features, too. This makes it difficult to apply them on images directly as a image is represented as matrix instead of a vector, and also spatial information might be important. Convolutional neural networks are a special type of neural networks which are especially useful for this kind of input.

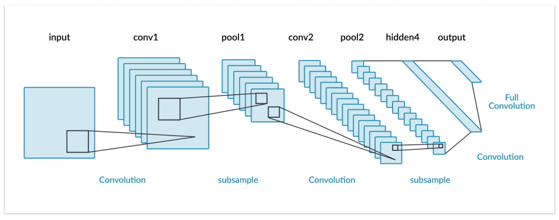
3.1.2 Convolutional Neural Network

Convolutional neural networks (CNNs) can handle two (or three-) dimensional input particularly well. They can preserve spatial information of the input and are often used for image processing tasks. CNNs are often used to reduce the huge amount of input features occurring in high resolution images, and output a small as well as informative representation of the input in form of feature maps. Most CNNs are built of two different types of layers. Firstly, the convolutional layer which is used to extract abstract features of the input. Secondly, max-pool layers, which are used to downsample and reduce the number of input features. Note that in this thesis the 2D-convolutional layers with two dimensional filters are meant when talking about CNN, if not specified otherwise.

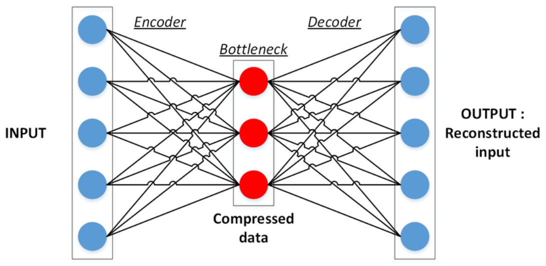
A convolutional layer is not fully connected to its input and it has its weights grouped in filters. The operation of a convolutional layer can be visually imagined by sliding each of its filter over the input image and performing a convolution between the image and the filter at each position. See an illustration of this in figure 3.3a. A max-pool operation works in a similar manner as it slides a window instead of a filter over the input and takes the maximum value in the window at each position. It is illustrated in figure 3.3b. A stride specifies how many grid cells the filter / window is shifted each time it is moved. As illustrated in figure 3.3a, the center of the filter is initially aligned at the boundary of the input. This results in smaller output size because there are less shifts than input value in both dimensions, and the number of shifts specify the number of values in the output. Padding the input at the edges, for instance with zeros, allows for more shifts. This is usually done to maintain the input dimensions in the output.

[1] https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/2d-convolution-block (last visited 12-17-2020)

[2] https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/2d-max-pooling-block (last visited 12-17-2020)



(a) A typical architecture of deep convolutional neural networks. Multiple convolutional- and pooling layers downsize the input and extract higher-level features which then serve as input for a fully connected neural network[3].



(b) The autoencoder network minimizes a reconstruction error between input and output, which requires the to be as similar to the input as possible. A bottleneck in the middle part of the network forces the network to find a good lower dimensional representation of the input[4].

While, the number of parameters in a fully connected layer is directly coupled to the amount of input features ($\#number\_parameters = (\#input\_features + 1) * \#output\_features$), the amount of parameters (weights) of a convolutional layer is independent of the height or width of the two dimensional input. It depends only on the size and number of chosen filters. Therefore, CNNs can be used to extract small information rich feature maps, even for high resolution images with huge amounts of input features, whilst using a comparatively small amount of weight parameters.

Deep convolutional neural networks have proven to be excellent feature extractors in the field of image processing. They are often used as ‘backbones‘ to extract features that serve as input for a following fully connected neural network, as it is illustrated in figure 3.4a. Popular deep CNN architectures which are often used as backbones and have also been utilized in this thesis, are the ResNet [10] in different variations and the VGG16 [28]. They were designed to tackle the task of image recognition. Thus, they learn to extract features that characterize images and enable to distinguish different types of images.

Autoencoder

An autoencoders is a neural network which consists of an encoder and decoder part as illustrated in figure 3.4b. Autoencoder networks can be convolutional, fully connected or a mix of both. The encoder part is used to compress the input to fewer, yet more informative features. They will be used by the decoder to reconstruct the input. This is achieved by defining a reconstruction loss that is to be minimized, on the original- and reconstructed inputs. By using the reconstruction loss, an autoencoder can be trained with arbitrary images. The encoder part can then be used to serve as feature extractor, similar as ResNet or VGG16, as mentioned in subsection 3.1.2.

In this thesis, the encoder part of a simple auto-encoder architecture will be additionally used as backbone. See subsection 4.2.2 for more information about the used backbones.

[3]https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-architecture-forging-pathways-future/ (last visited 12-17-2020)

[4]https://www.researchgate.net/profile/Jeremie_Sublime/publication/333038461/figure/fig3/AS:757767321169921@1557677216019/Basic-architecture-of-a-single-layer-autoencoder-made-of-an-encoder-going-from-the-input.ppm (last visited 12-17-2020)

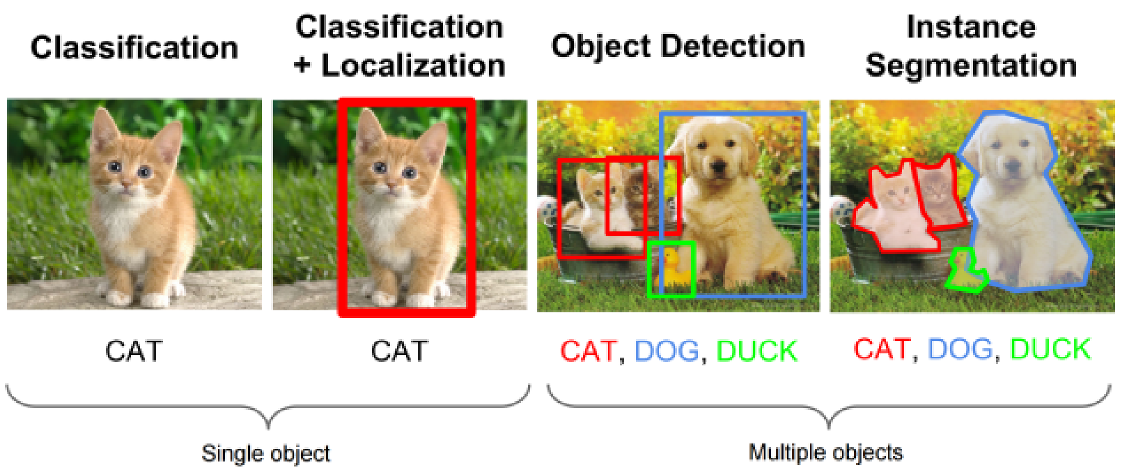


Figure 3.5: Illustration of the differences between object localization, image classification, object detection and instance segmentation.[5]

3.2 The Evolution of R-CNN

CNNs have had great success in image classification. It seems natural to also use them for object detection. However, this comes with some issues as one image might contain many different objects and a classifier would need to settle on one specific class, thereby the classifier would be compelled to ignore all the other classes. Likewise, a standard classifier would not output the objects location.

This section is about the basics of object detection, covered in subsection 3.2.1, as well as the different techniques which finally lead to fully complete object detection networks, such as the Faster R-CNN, described in subsection 3.2.5.

3.2.1 Object Detection Basics

Object detection is a subcategory of object recognition. Object recognition relates to a field in computer vision which involves different tasks, such as object localization, image classification, object detection, and segmentation.

Object Localization, Classification, Detection and Segmentation

Object localization is the task of locating one or multiple objects in an image. Hence, an object localization algorithm should take an image as input and output a set of locations on the image, i. e., one location per object. These locations are usually represented as 2D bounding boxes. Note that in this thesis it will be spoken of 2D-bounding boxes only.

[5] https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/object_localization_and_detection (last visited 12-17-2020)

A bounding box defines a rectangle area on an image. Axis-aligned bounding boxes are aligned with the x and y axis of the image. They can be encoded by providing two points on the image $(x0, y0, x1, y1)$ or one point with a width and a height value (x, y, w, h). Either way, the encoding uses a total of four values. However, one could use an additional value representing an angle, to define object oriented bounding boxes. In this thesis, bounding boxes will always be axis aligned and not object oriented.

Image classification is the task of predicting some type or class for an image. In the field of object recognition, image classification is typically used to classify an image depending on the most dominant object class visible in the image. Thus, an image classification algorithm takes an image as input and outputs one class label for the most dominant object in the image.

Object detection combines object localization and image classification. For some given input image, a object detection algorithm outputs bounding boxes and corresponding class labels for the detected objects in the image.

Image segmentation algorithms take an image as input and output a segmentation mask of the same size as the input image, in which each pixel is labeled with the class of the object / thing it belongs to. Instance segmentation is basically the same, but it distinguishes between object instances. Thus, two different objects of the same class will have either different labels or some other additional information which marks them as separate instances.

Figure 3.5 illustrate the differences between object localization, image classification, object detection and instance segmentation.

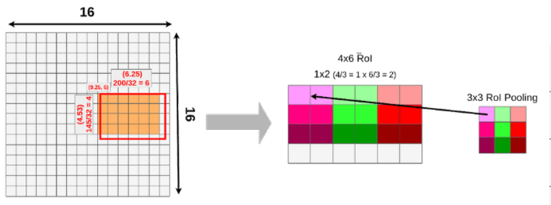
3.2.2 Region Proposals

When going from image classification to object detection, region proposals play an important role. They suggest regions on the image where objects may be located, usually in form of bounding boxes. In the rest of this thesis, region proposals will be assumed to be represented as axis aligned bounding boxes, although they could be of any shape or orientation. If such proposals are given and a good image classifier exists, one can run the image classifier for each of the proposed regions of the input, also called region of interest (RoI), and validate each region if it does indeed contain some expected object or not, i. e., is background. Therefore, combining a good region proposal algorithm with a good image classifier yields to a two-stage object detection model that does both generating bounding boxes and classifying labels for the objects in the image. Note that, as it was mentioned in the related work chapter, there also exist single-stage algorithms which do not use a separate stage generating region proposals 2.1.3.

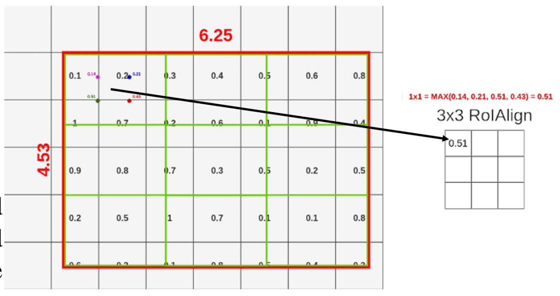
Region of Interest Pooling

[6] https://towardsdatascience.com/understanding-region-of-interest-part-1-roi-pooling-e4f5dd65bb44 (last visited 12-17-2020)

[7] https://towardsdatascience.com/understanding-region-of-interest-part-2-roi-align-and-roi-warp-f795196fc193 (last visited 12-17-2020)



(a) Illustration of RoI Pooling on a 4×6 RoI and a fixed 3×3 output size. The RoI is mapped to a 16×16 feature map. Due to the discrete grid size some information is already lost during this mapping. The pooling layer has to be of size $(4/3) \times (6/3)$ which is rounded down to 1×2. Due to quantization, the information of the last RoI row is lost when applying the pooling operation. [6]



(b) RoI Align uses bi-linear interpolation to map the RoI onto the feature map and pools its values without skipping information. [7]

Region proposals define regions of interest on the input image. As described in subsection 3.2.2, the combination of good region proposals and a good classifier which can classify these regions of interest, already yield an object detection model. A trivial solution for a region proposal would be to define the whole image as region of interest. The objects of interest would be completely captured by the region proposal. However, this does not help with localizing them and is basically the same as image classification. A good region proposal algorithm should output tight bounding boxes for the proposed objects in the image. Due to the objects being usually of different sizes, the region of interest are also differently sized. This poses a problem, since classifiers usually expect the input to have a fixed size, depending on its architecture. This problem is tackled by Region of Interest Pooling (RoI Pooling).

RoI Pooling works like max-pooling, but with a pooling size dependent on the size of the input. The pooling size is calculated such that the output always has the same size for every input. RoI Pooling is a special case of Spatial Pyramid Pooling [11], yet with only one pyramid layer. It is used to extract a fixed sized feature vector from arbitrary sized feature maps. RoIs can have any size, whereas the size of the pooling output is fixed. Therefore, some information might get lost due to quantization, if the input size is not divisible by the output size. Note that region of interests are defined on the input image. If a backbone is used to extract a small feature maps from the input image, the RoI have to be mapped to them before pooling can be applied. In standard RoI Pooling this mapping introduces additional quantization, similar to the pooling process, if the RoI coordinates are not divisible by the size of the feature maps. Figure 3.6a illustrates this. The mapping depends on the architecture of the backbone. Different backbones may have a different stride. The stride of a backbone is the factor an input image is reduced compared to the size of its resulting feature map. Suppose that the input image is of size 512×512 and the backbone outputs a 32×32 feature map, then the backbone would have an overall stride of $16x16$, or just 16. Note that the stride of a backbone is independent of the input size and solely depends on its architecture. As it was mentioned before, there might be an additional loss of information, if the size of the input image is not divisible by the network's stride.

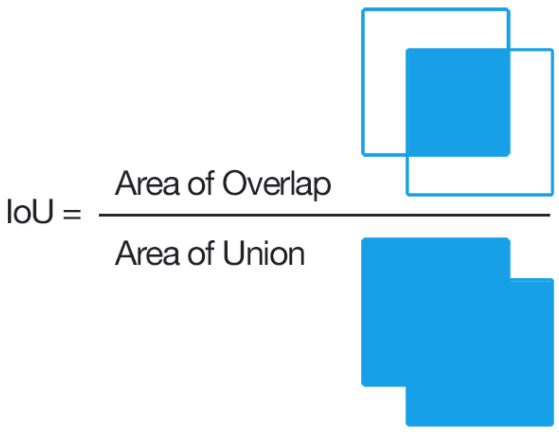


Figure 3.7: The intersection over union. A measure of similarity between bounding boxes. [8]

RoI Align works similar as RoI Pooling. However, it circumvents both the quantization when mapping the RoI to the feature map and the quantization when pooling the RoI. This is possible by mapping the RoI to the feature map and dividing it in the appropriate sections with sub-grid precision and then using bi-linear interpolation to evaluate the values for the new grid position in the divided RoI. Figure 3.6b illustrate this.

Intersection over Union (IoU)

The intersection over union (IoU) is a measure of similarity between two areas (here the areas are defined by bounding boxes). It is computed by the area of overlap divided by the area of union. Let box one be B_1 and box two be B_2, then the intersection over union of those two is computed as follows:

$$IoU(B_1, B_2) = \frac{area\_of\_overlap}{area\_of\_union} = area(B_1 \cap B_2)/area(B_1 \cup B_2) \tag{3.1}$$

This is illustrated in figure 3.7. An IoU of 1 would indicate, that the bounding boxes are identical, while an IoU of 0 would indicate, that they have no common area at all.

3.2.3 R-CNN

R-CNN is short for “Regions with CNN featuresänd refers to an object detection architecture proposed in [8]. It combines traditional image classification using a CNN with a region proposal method. In [8] they use selective search [31], which is not further relevant in this thesis. The region proposal method is not part of the network.

First, the region proposal method generates all of the region proposals. Then, the regions of interest are directly pooled from the input image and resized to fit the CNN input which then computes a feature map per given RoI. These features serve as input for a linear regression classifier which

[8] `https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/` (last visited 12-17-2020)

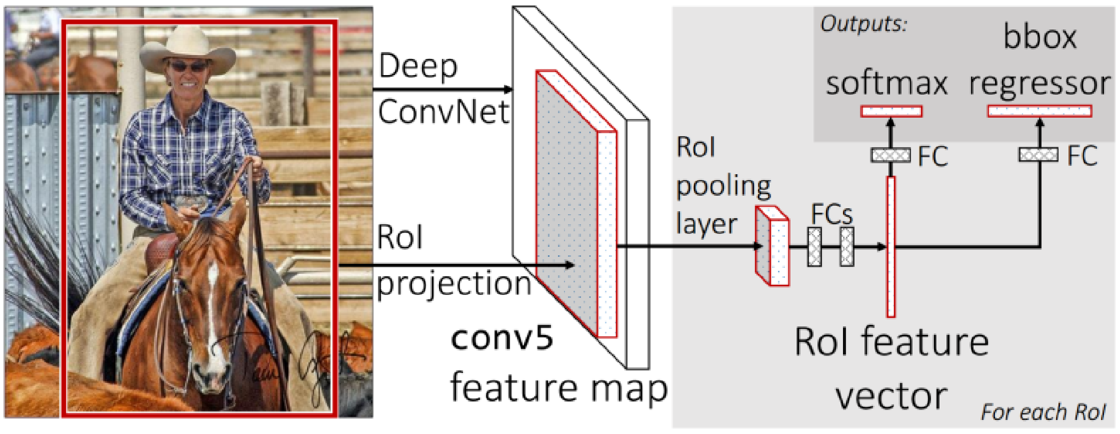


Figure 3.8: The Fast R-CNN architecture. [7]

performs the bounding box regression and outputs four values on how to adjust the original proposals in order to fit a tight bounding box optimally. Simultaneously, the same features are also fed to support vector machine (SVM) classifiers which determine the classes of the objects or background. Due to its architecture and the way the proposals are generated and processed, it was really slow which is the main issue of the R-CNN.

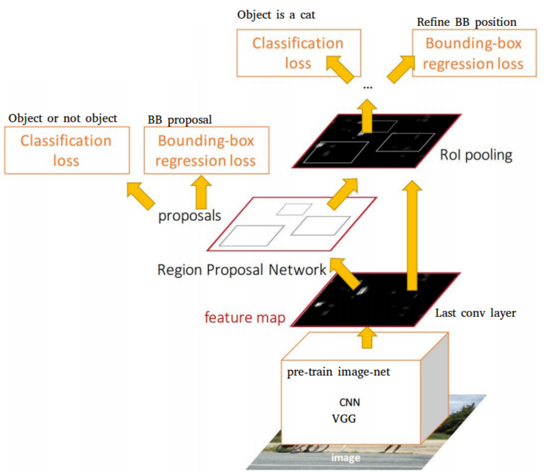
3.2.4 Fast R-CNN

Instead of computing features per proposal, the more sophisticated Fast R-CNN [7] builds on the idea of the R-CNN. It computes the feature map for the complete image once and then applies RoI Pooling, as described in subsection 3.2.2. Another change is, that it uses fully connected layers with a softmax classifier to classify the RoI instead of support vector machines (SVMs). Its architecture is depicted in figure 3.8.

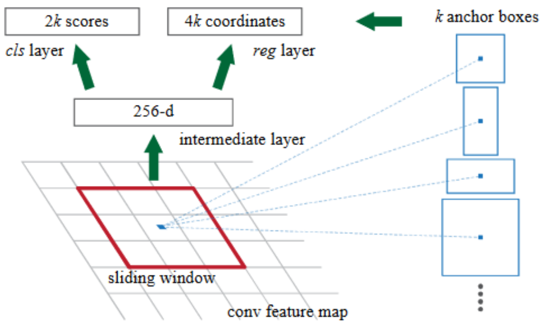
3.2.5 Faster R-CNN

The Faster R-CNN introduces a region proposal network (RPN) and makes the proposal generation method part of the network itself. In contrast to the Fast R-CNN, the Faster R-CNN has an additional stage between the convolutional layers, which compute the feature maps, and the RoI pooling. Note that this allows to omit the external region proposal method which was necessary in the R-CNN and Fast R-CNN and is now no longer a part of Faster R-CNN. Figure 3.9a illustrates the base architecture of the Faster R-CNN network, including the new stage for proposal generation. In this stage, the RPN generates region proposals based on the feature maps generated from the convolutional layers. Thus, the RPN and RoI Pooling share the same features. Afterwards, the region proposals are used together with the feature maps as input for RoI Pooling and further processing, which is the same as it was in the Fast R-CNN architecture, see subsection 3.2.4.

Including the RPN in the network allows to train the whole network altogether by using a multi-task loss, which is the sum of two losses from the RPN and two losses from the Fast R-CNN, as it is shown in figure 3.9a.



(a) The Faster R-CNN architecture, including four different losses that contribute to a multi-task loss which can be used to train the networks as a whole. [25] [5]



(b) The region proposal network is sled over the convolutional feature map in a sliding window like manner, generating one region proposal with a confidence score for each of the k anchors at each feature map position. [25]

Region Proposal Network

The region proposal network is a small neural network which generates region proposals in form of bounding boxes and a confidence score per proposal. The network consists of one regression layer and one classification layer. The region proposals are generated by sliding a small window over the feature maps. At each position on the feature map, the features inside the window are used as input for the RPN. Thus, the RPN can be imagined as being sled over the feature map.

For every window of features, the regression layer of the RPN generates a fixed amount of region proposals in a single evaluation step of its layers. Let this amount be denoted by k. Thus, the regression layer has $4 * k$ outputs which correspond to the encoding of k region proposals. The classification layer has $2 * k$ outputs which estimate the probability of object or no object for each of the proposals. The classification layer is evaluated in parallel with the regression layer.

Each generated region proposal encoding is associated with one of the so called k anchor boxes. There exists a one-to-one mapping between region proposals and anchor boxes. Anchor boxes exist in different sizes and aspect ratios. The center of the anchor boxes are placed at the center of the window. Figure 3.9b illustrates this process. A region proposal encoding defines how its corresponding anchor box needs to be adjusted to yield / decode the actual region proposal. The proposals are sorted by their confidence score and further processed by non-maximum suppression.

Non-Maximum Suppression (NMS) is a technique to filter overlapping object proposals that most likely detect the same object. It only leaves those with the highest confidence scores. Starting with an empty final output, it iterates over the sorted input list until it is empty. Before each iteration it pops the proposal with the highest score and adds it to the final output. During each iteration, it calculates the intersection over union between the recently popped proposal and each of the remaining proposals. Additionally, it deletes all proposals that have an IoU that is bigger than some threshold.

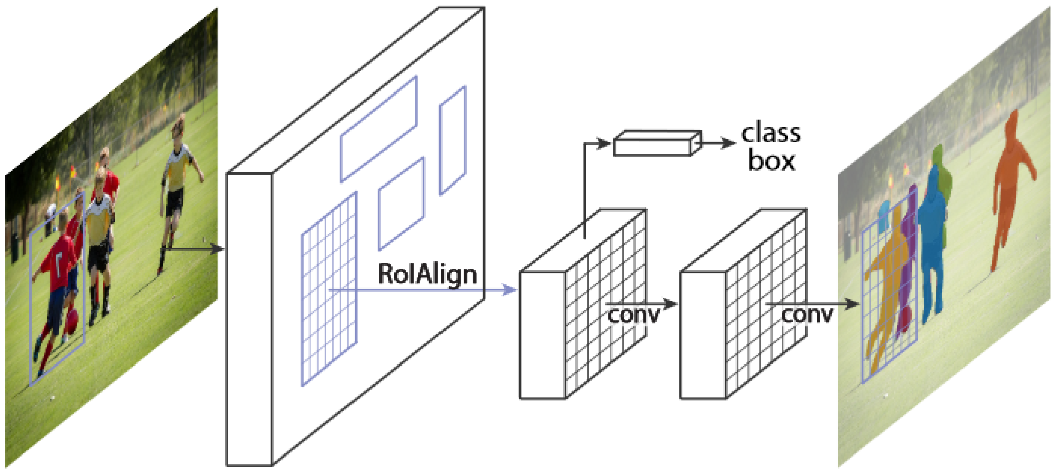


Figure 3.10: The Mask-RCNN architecture. An additional branch produces a segmentation mask per RoI. This branch is in parallel to the regression and classification branch from the Faster R-CNN. It uses RoI Align instead of RoI pool for a correct pixel-to-pixel alignment between the input and the produced segmentation masks. [9]

After applying NMS, the remaining top-k proposals will be selected and returned as final output of the RPN. [25]

3.2.6 Mask RCNN

The Mask R-CNN [9] is an extension to the Faster R-CNN. It performs instance segmentation by extending the existing Faster R-CNN with an additional branch for predicting segmentation masks. The mask branch is parallel to the existing classification and bounding box regression branches from the Fast R-CNN, as shown in figure 3.10. Thus, the segmentation shares the region of interests and corresponding features with classification and regression.

Instead of using RoI Pooling, as in the Faster R-CNN, the Mask R-CNN uses RoI Align (see 3.2.2). This allows for a correct pixel-to-pixel alignment from the input image to each RoI. RoI Align makes the predicted segmentation mask to properly map to the corresponding region of interest on the input image.

For each RoI the mask branch simultaneously produces a $m * m$ segmentation mask for each of the K possible classes. Hence, the output size of the mask branch is $K * m^2$. During training, the mask loss is only defined on the outputs corresponding to the correct output class. Therefore, each output is trained only for its corresponding class which makes it independent to other classes. This makes class prediction separated from the segmentation mask prediction. Class prediction now relies on the classification branch of the Fast R-CNN only.

For training a multi-task loss consisting of the new mask loss, the classification, and the regression loss of the two other branches, is defined on each RoI. This allows to train the branches together.

3.3 From Simulation to the Real World

To train supervised learning algorithms, it is required for the training data to have corresponding input and output pairs. As it was mentioned in section 3.1, neural networks can learn to approximate continuous functions. They are usually trained in a supervised learning manner which means that for a given input a loss is defined to measure the discrepancy between the current output of a network and the expected output. During training of such a network its weights are updated in a way that this loss is optimized.

Generating data with corresponding input and output / ground-truth pairs turns out to be a non-trivial chore, especially if real world data is needed. Often real world training datasets contain hand-labeled ground-truth information which is a time-consuming task. The use of a simulation allows to automatically generate artificial input and corresponding ground-truth pairs.

Subsection 3.3.1 covers a concept called ‘Transfer Learning‘ which relates to the idea of training a model on data of some domain and transferring the learned knowledge to a model which works in another domain. In this thesis, this concept is applied to a setting in which one domain is a simulated environment and the other domain is the real world.

Subsection 3.3.2 and subsection 3.3.2 refer to two techniques which assist to implement this concept.

3.3.1 Transfer Learning

Transfer learning is a broad concept and describes the idea of using knowledge which has been learned by solving one task, to help solve another task. Both tasks might be in the same or different domain. Transfer learning also applies if the tasks are essentially the same, but the models work with different domains.

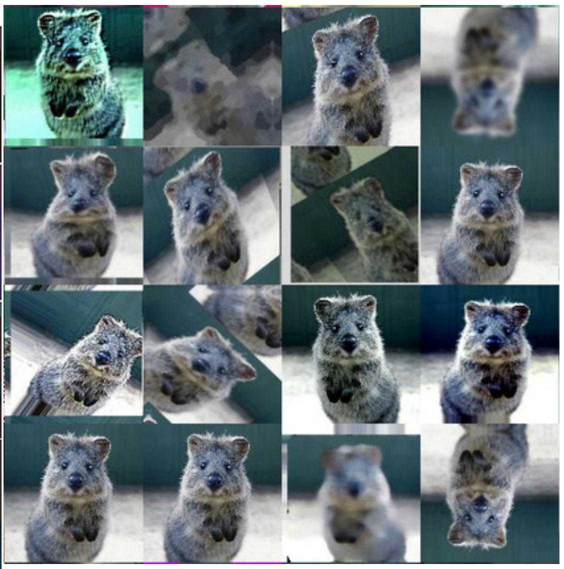
In the context of neural networks this is commonly used. For example, pretrained CNNs are often used to extract features of images which then serve as input for arbitrary other image processing tasks. These CNNs were mostly trained on image classification tasks during which they have learned to extract important features of images and to distinguish as well as classify them correctly. This knowledge of ‘how to extract important features‘ can be reused when such a pretrained CNN is used as preprocessing step for arbitrary other image processing tasks in different domains.

This poses a contrast to traditional machine learning approaches, in which models have been specifically trained on one specific task in one specific domain.

3.3.2 Domain Randomization

Domain randomization is a technique for training models on simulated images which can transfer to real images by randomizing the rendering in the simulator. [29] The model learns to solve a task in the artificial domain and applies this knowledge to solve the same task in the real world domain. Here the domains change, but the tasks remain the same. Thus, the task to be solved and the target domain, i. e., the real world, are known beforehand.

(a) A real world setting which is emulated with a simulator, but the simulated environments have been randomized to differ in each data element. [29]



(b) Different image augmentations applied to the same source image result in many different images but which still represent the same original information. [14]

A simulation which can mimic the setting of the real world implements the idea of domain randomization by randomizing the simulated environment as much as possible while maintaining the same key elements of the real world. Then, by randomizing the data, the real world appears as just another randomization, i. e., the model learns to solve a task by recognizing the important key elements in the data and ignoring the overall appearance. Figure 3.11a shows different randomizations of a setting where objects are placed on a table. The size of the objects, their color, texture, and other characteristic such as the lightning conditions, are randomized to increase the variety of data.

3.3.3 Data Augmentation

Data augmentation techniques can be used to increase the available data by creating modified versions of the existing data elements. Given one data element in the original dataset, its corresponding modified versions all rely on the same original information and often retain most of the original properties. Figure 3.11b shows different augmentations applied to the same initial image. Even though the images differ greatly in their appearance, it is still obvious that they belong together.

Data augmentation techniques can be used in the sense of domain randomization as well. Nonetheless, this thesis distinguishes between the domain randomization and data augmentation in the sense that domain randomization refers to randomizing the information the data carries. Thus, two data elements rely on different information as they are created. While, data augmentation may produce many different new data elements, it always relies on already existing data.

4 Methods

This chapter covers the methods which were used for training and evaluation of the different models. It can be split into two main sections.

Section 4.1 is about the simulation framework that has been developed to generate the artificial datasets. It describes the different simulation settings, the objects used and the domain randomization techniques that have been employed w.r.t. the simulation.

Section 4.2 goes into detail of the different architectures and methods used during training and evaluation. It explains how the models are designed, trained, and which metrics were used to evaluate them.

4.1 Simulation Framework

The simulation framework has been written in the programming language Python[1]. It makes use of the `Bullet Real-Time Physics Simulation` which has been written in `C++` but is also available as a Python module named `PyBullet` [2].

The frameworks core is a simulation class which is described in subsection 4.1.1. It wraps the PyBullet simulation client, which is a physics engine, and extends it with multiple different functionalities in order to simplify its use for grasp experiments and domain randomization.

The simulation is used to generate the artificial data. Each data element is stored to disk as a encoded so called sample object which is further described in subsection 4.1.2.

The simulation class supports different types of objects. Some are of simple and some of rather complex geometry, see subsection 4.1.3.

The framework provides a functionality which considerably simplifies the data generation, while it continues to provide arbitrary flexibility to create scenes. This is further described in subsection 4.1.4.

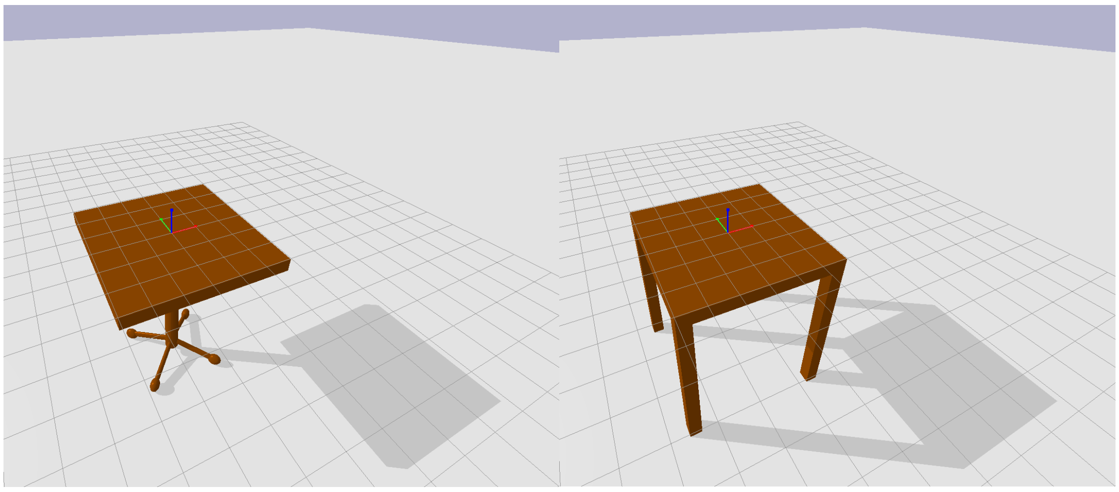


Figure 4.1: The left figure shows the table with only one stand in the middle. This variant is the same as the table that has been used for the data generation in the real world. The right figure shows the variant with four legs. Their overall sizes are the same, i.e., they are equal in width, length and height.

4.1.1 Simulation

The simulation class can be seen as wrapper of the PyBullet simulation. In use, it connects to the PyBullet physics server and holds the corresponding client id as a member variable. This allows the simulation class to access and control the underlying PyBullet simulation. A PyBullet simulation contains only one single light source. It is not able to manage more, thus this simulation has always exactly one light source which is defined by a direction and distance vector w.r.t. the world origin.

The standard simulation consists of a white plane representing the ground, a table that is available in two different styles, and a single light source. The table is available with either one stand in the middle and four attached rolls on the bottom, or with four legs at each corner. Figure 4.1 shows both table variants used in the simulation. Note that these figures were rendered via the the PyBullet graphical user interface (GUI) by using an OpenGL renderer. PyBullet does not allow to use the OpenGL renderer in headless mode. Thus, the actual images of the generated samples will be rendered by a different renderer with different lightning and camera settings.

The simulation class allows to load a bunch of textures from a given directory which can be applied to the objects inside the simulation.

There are different options available to configure the lightning conditions for the actual rendering of images. As mentioned, the light source can be adjusted in distance and angle w.r.t. the world origin. In figure 4.1, the world origin is displayed at the middle of the table and shows the orientation of the three axes of the simulation's coordinate system. Further lightning options to configure are: the ambient coefficient, the diffuse coefficient, and the specular coefficient of the light. Finally, it is possible to enable and disable the computation of shadows.

[1]https://www.python.org/downloads/release/python-385/ (last visited 12-17-2020)

The simulation allows to add different types of objects. They can be added to the simulation in three different ways:

1. The standard method would be to place the new object randomly on the table. This can happen with one single function call. Its position, orientation, size and color is randomized. The shape depends on the type of the object. Section 4.1.3 elaborates further on the different object types and how their initial positions, orientations and colors are determined.
2. The simulation also allows to add multiple objects with one single function call at once. It works by continuously adding objects until the desired amount has been reached. However, before adding a new object, its initial position, orientation, and size is used to compute whether it collides with another object. If the object does not collide, it can directly be added to the simulation. If the object does collide, a new random position, orientation, and size for it will be computed until it mo longer collides with other objects, or a maximum number of trials has been performed which stops the adding of new objects.
3. Similar to 2, but the height of the objects will also be randomized. Thus, this places multiple objects in a certain height range over the center of the table. After all object positions have been determined and the objects were placed in the simulation, the simulation will be run for multiple steps. Meanwhile, a second function will continuously compute the sum of the object base velocities. The simulation will stop if the base velocities have reached almost zero, i. e., they have stopped moving. This simulates multiple objects being dropped on the table. To prevent the objects from falling off the table and laying around on the ground, every simulation step performs a check on the object positions and removes them if they are too far away from the table.

Option 3, i. e., dropping the objects on the table, is the preferred way to add objects and resembles best how the experiments are expected to look like in the real world.

Especially for round objects such as spheres, the dropping often causes them to roll of the table. Therefore, before the dropping simulation, there are artificial walls created around the table in order to stop the objects from falling off. Those walls are removed again when the objects have stopped moving. Although the walls are invisible, removing them is needed, otherwise they would cast shadows.

The simulation is supposed to generate multiple different scenes. A scene represents how the objects are arranged in the simulation, i. e., their position and orientations, their color and texture, but also the lightning conditions in the scene. To generate multiple scenes by using the same simulation, the simulation implements a function that will remove all objects except the table, plane and the light source, and allows to build a new scene.

As mentioned before, the generated data from a simulation comes in the form of so called samples. A sample describes a scene from a specific camera perspective, which will be further explained in subsection 4.1.2. Additionally, it contains a rendered image of a scene which is a snapshot of the scene taken by a virtual camera as well as information about the objects seen in the image. The perspective is defined by a view matrix and projection matrix and may change for every sample. The simulation has a function to create such a sample for the current scene. It includes for instance, a rendered RGB image, an associated segmentation mask, and a corresponding depth image.

Rendering may result in sharp edges or other artifacts. Hence, it is implemented that the images can be generated with doubled resolution, but are afterwards downsampled using bi-linear interpolation in order to smooth edges. The function also collects the other information contained in a sample, such as, among other things, the objects states which hold the objects bounding boxes in image coordinates (see subsection 4.1.3).

4.1.2 Sample

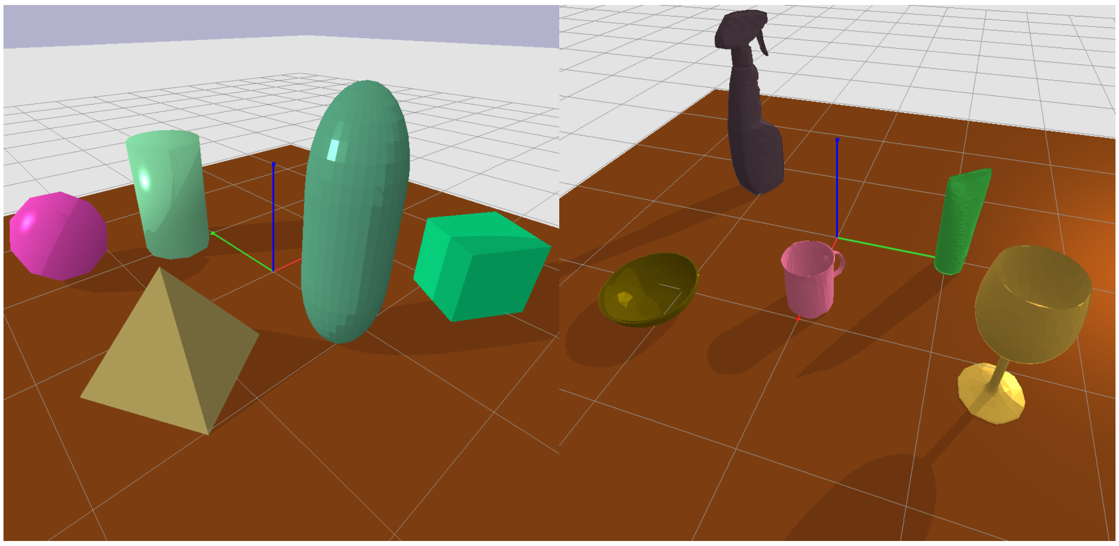
A sample is used to gather the data of a specific scene. The corresponding sample class has functionality to encode and decode the sample objects to and from a disk. Thus, it is used to store the collected data. As it was mentioned before, each sample object holds the information about a certain scene from a specific perspective. Hence, it is possible to have multiple samples from the same scene.

The specific data a sample holds is the following:

- A `unique identifier` which is a `uuid4` string which uniquely identifies each sample. It is also used in the file names when encoding and storing a sample to disk and allows to identify the files that belong to the same sample.
- The `width` and `height` of the rendered image stored by the sample. Note that this is not affected by oversampling.
- The encoded `view` and `projection` matrices that were used to render the image.
- The rendered `RGB image`.
- The associated `segmentation image`. It is of the same size as the other RGB and depth image. At each of its image coordinates, it has the object id of the visible object at this coordinate as integer pixel value.
- A corresponding `depth image`.
- A `object state` dictionary, i. e., a key-value mapping of the `object ids` in the segmentation image to the corresponding `objects states` of the objects. The content of a `object state` is further explained in subsection 4.1.3.

The simulation allows to generate such samples of scenes, yet they are still in memory. Therefore, the sample class also contains the needed functionality to encode and decode a sample object to the disk. A sample is encoded and stored in four different files:

- The samples information, except for the `RGB`, `segmentation`, and `depth` image, is encoded as a `json` object and stored as `sample_{uuid4}.json` file. The json encoding is handled by a specific SampleEncoder class, which is needed to handle more complex data such as the `view` or `projection` matrix and the `object states`.
- The `RGB` image is saved as `rgb_{uuid4}.png` file.
- The `depth` image is saved as `depth_{uuid4}.tiff` which has the actual depth values in the simulation, i. e., in units, encoded as 32 bit floating point pixel values.



(a) The different objects with simple geometry in randomized shape and color on the table. From left to right: Sphere, Cylinder (flipped), Pyramid, Capsule (flipped), and Cube.

(b) The different objects with complex geometry in randomized shape and color. From back left to front right: spray flask, glass bowl, flower cup, toothpaste, and wineglass.

- The `segmentation` image is saved as `seg_{uuid4}.json` file. The pixels are eight bit integer values corresponding to the `object ids` of the objects in the `RGB` image.

Decoding a stored sample into a sample object is done by using the SampleDecoder class.

The sample class allows to load one or more samples in different ways:

1. Providing a path to a directory together with a given sample `{uuid4}`.
2. Providing the path to its `.json` file.
3. Providing a path to a directory which contains `.json` files of samples. This will return a list of all samples that were stored in the specified directory.

The sample class also allows to store a single sample as well as to store a complete list of samples at once.

4.1.3 Objects

The simulation supports different object types, each type has its own shape. There are a few basic standard objects with shapes that exist in the PyBullet engine by default. It is possible to extent the simulation with arbitrary objects whose shapes are determined by some mesh that can be loaded into the simulation. For example, the shapes for the two table variations, were designed by using Blender[2]. Most of the simple geometries have been already available in the simulation as they

[2]`https://www.blender.org/` (last visited 12-17-2020)

are standard PyBullet shapes. All of the complex geometries come from external sources such as blender. Note that multiple instances of the same object have the same overall shape but may be scaled in different sizes or colored differently.

Each available object in the simulation has its own file that defines two classes: a `base object` class, e. g., `BaseCube`, and a corresponding `real object` class, e. g., `RealCube`. Each base class inherits from the `BaseObject` class, and each corresponding real class inherits from the `RealObject` class. The base classes define the shapes of the objects as well as a static function which will return such a new object instance with randomized position, orientation, size, and color. When a new object is created, this static function is called. It then randomly selects the parameters defining the size of the shape as well as its color, position, and orientation, from a specific range, and returns this contained in a base object instance. Note that the base object instance has not been added to the simulation yet. However, the base object instance is used to define a corresponding real object instance which adopts and realizes the information in the base object by placing it in the actual simulation. The real instances hold the information about such objects after they have been added to the simulation. As the position and orientation may change per object, the corresponding real class provides functionality to access the current position and orientation of the object. This for instance, allows to compute the bounding boxes of the objects. Real object instances also hold the `object id` which identifies the object in the simulation and in the segmentation image, see 4.1.2.

Simple Geometries

The following lists the objects defined to have simple geometries.

- `Sphere` - a standard PyBullet geometry. Its size is defined by a `radius`.
- `Cube` - a standard PyBullet geometry. Its size is defined by a `extent` in each axis direction.
- `Cylinder` - a standard PyBullet geometry. Its size is defined by a `length` and `radius` value. It may be flipped, i. e., with a 50 % probability the cylinder is placed vertically instead of horizontally.
- `Capsule` - originally a standard PyBullet geometry but due to false rendering it was remodeled with blender. Its mesh can be scaled in `length` and `radius` similar to the `Cylinder`. It also may be flipped.
- `Pyramid` - a mesh designed via blender. Its mesh can be scaled in `length`, `width`, and `height`.

Figure 4.2a displays the different available simple geometries.

Complex Geometries

The shapes of the objects with complex geometries are all defined by meshes. The available complex geometries originate from [15] and are the following:

- `Spray Flask` - its shape is defined by as mesh. Its mesh can be scaled in `width` and `height`.
- `Glass Bowl` - its shape is defined by as mesh. Its mesh can be scaled in `width` and `height`.
- `Flower Cup` - its shape is defined by as mesh. Its mesh can be scaled in `width` and `height`.

- `Toothpaste` - its shape is defined by as mesh. Its mesh can be scaled in `width` and `height`.
- `Wineglass` - its shape is defined by as mesh. Its mesh can be scaled in `width` and `height`. It may be flipped upside down with a 50 % chance.

Figure 4.2b displays the different available complex geometries.

Object State

A sample contains a {`object id`: `object state`} mapping. Each object instance in the rendered image has an unique `object id`. This is also used in the segmentation image to indicate where the objects / instances are located. Every real object has an `object state` which can be retrieved via a function and gives information about the object in the current simulation regarding size and orientation. Simultaneously, it includes the information of the `base object`. An `object state` contains the following information:

- The `position` of the object in world coordinates when the state was retrieved.
- The `orientation` of the object w.r.t. the world origin when the state was retrieved.
- The `size` of the object (the same as for its base object).
- The `color` of the object (the same as for its base object).
- The `x`, `y`, `width`, and `height` of the object's bounding box in image coordinates with floating point precision.
- The `image frame points` of the object, if they were computed (see below for a description).
- The object's `segmentation mask` in run-length encoding (RLE)[3].

The most relevant information in an `object state` is the computed bounding box as well as the segmentation mask of the corresponding object instance. They are part of the ground truth annotations of a sample, and are computed by the simulation during the generation of a sample. Note that objects which are not visible in the rendered image, i. e., their bounding box is outside the image bounds or its visible area is zero, will not be included in a sample.

Bounding Box Computation

The simulation allows to compute the axis aligned bounding box of an object in two different ways.

The first, more complex approach defines points on an object that form a convex hull and tightly encompass the object. These are the so called `frame points`. They are defined in the objects local coordinate system, but can be mapped to world coordinates with the knowledge about the world position and orientation of the objects. Furthermore, knowing the view and projection matrix, those

[3]`https://github.com/cocodataset/cocoapi/blob/master/PythonAPI/pycocotools/mask.py` (last visited 12-17-2020)

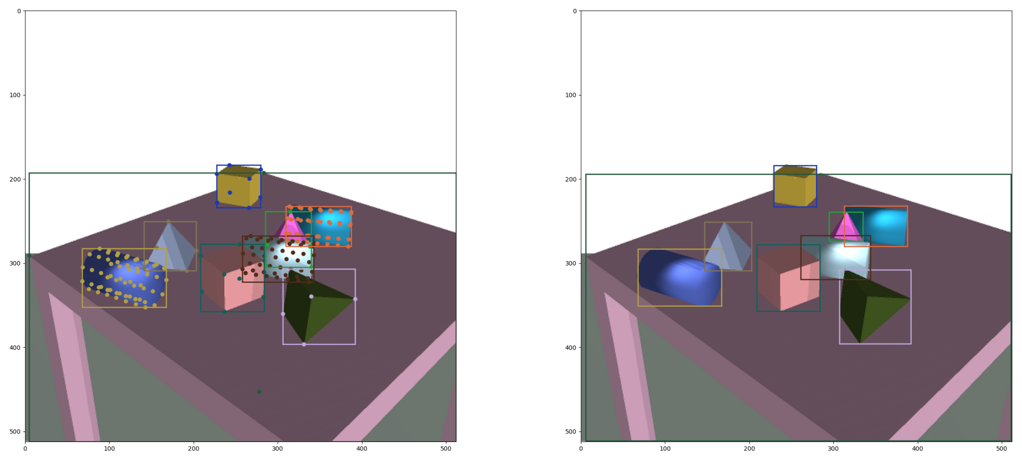


Figure 4.3: In the left image, frame points are projected on each object to encompass its convex hull. These are then projected on the image and used to compute the bounding boxes. In the right image however, the segmentation mask of each object is used to compute its bounding box. Thus, only the visible parts of the objects contribute to the computation of the bounding boxes. Note, how the bounding box of the pink pyramid between the two cylinders differs between the images.

can then be mapped to image coordinates. By using these image coordinates, the bounding box of an object can be computed by taking the minimum and maximum values of all the points w.r.t. the x- and y-axis.

The second approach, which is easier and independent to the shapes of the objects, is to compute the bounding boxes given the segmentation image. The segmentation image is returned by default and the `object id` is known for every object, therefore it is easy to extract the segmentation mask per object. Assuming the segmentation mask represents frame points in image coordinates, the bounding box can be computed in the same way as in the first approach. The computation of bounding boxes via segmentation masks is independent to the shapes of the objects. Therefore, this is the preferred way of computing the bounding boxes.

The only difference in the result of both approaches is that the first approach does not consider overlapping objects during the computation of bounding boxes. Thus, even if the object would be partially or completely occluded by other objects, its bounding box would still be computed as if there were no other objects. The second approach does compute the bounding box only w.r.t. the visible pixels of the object. Figure 4.3 illustrates how the two approaches differ in the resulting bounding boxes.

4.1.4 Data Generation

The main purpose of the simulation is to generate data for training, validation, and testing of supervised learning models. As described in subsection 4.1.1, the simulation is able to generate scenes and to sample them. In subsection 4.1.2, it was shown that these samples contain the rendered images and `objects states`. Subsection 4.1.3 have shown, that these `object states` contain the bounding boxes and segmentation masks for the objects in the scene. Hence, w.r.t. the task of object detection, the encoded samples include the input and expected output pairs of an object detection- as well as an instance segmentation algorithm and form the data needed to train, validate, and test those algorithms.

For this purpose, there exist different `utility` functions that can interact with the simulation. There also exists the concept of so called `scene functions` which generate a random scene each time they are called.

Utility

The utility functions enable to generate a complete dataset consisting of a training, validation, and test set. The overall data is split by default into 80 percent training data, 10 percent validation data, and another 10 percent test data. The corresponding function can be called with a directory path to where the data should be saved, a scene function, a total amount of scenes, and an amount of how many times each scene should be sampled. Thus, the total amount of samples is: $total\_samples = amount\_scenes \times amount\_samples\_per\_scene$, which will be divided into train, validation, and test set. For each of those sets, a function is called which generates the corresponding portion of samples. This works as follows:

1. Define the middle of the table, here this is the world origin.
2. Create a simulation object. This starts the simulation and connects its physic client to the PyBullet physic server. It is done in `DIRECT` mode, consequently there is no GUI and rendering of the images will be performed using a `ER_TINY_RENDERER` instead of an `OpenGL Renderer`.
3. For the expected amount of scenes:
    a) The `scene function` is called, which creates a new scene by adding objects or changing textures. The scene function returns the parameters that define the random ranges for the creation of the view matrix.
    b) For the expected amount of samples per scene, a new random view matrix is created and the scene is sampled out of this perspective.
    c) The samples are encoded and saved in the previously provided directory.
    d) The scene is cleared, i. e., all created objects from the scene function will be removed. Texture changes persist but can be overwritten by the next call to the scene function.

The view matrix is computed via a camera position, a target position, and a camera up vector. The up-vector is always the same and it points upwards, parallel to the z-axis, i. e., $\vec{u} = (0, 0, 1)^T$. The camera position and target position are sampled from the ranges returned by the scene function. The middle of the table as well as a random offset that is sampled from the given range is used as

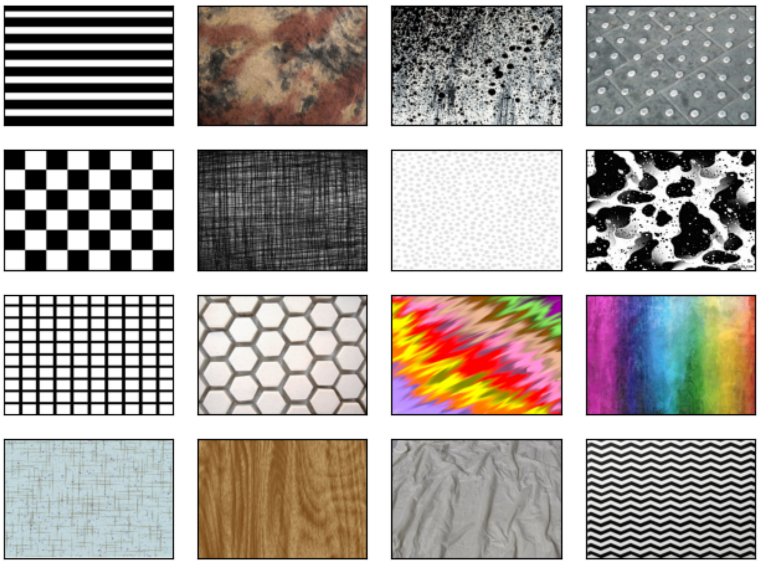


Figure 4.4: The different textures available in the simulation.

target position. The camera position is computed via a specific distance to the target and a direction vector from the target to the camera. Distance and direction are again sampled from a given range, but the direction is always set to point slightly upwards in order for the camera position to be above a specific minimum height value. Hence, the camera is always directed downwards.

Another utility function generates random lightning coefficients, which specify the ambient, diffuse, and specular lightning conditions for the light from the single light source.

Scene Functions

As described before, the generate data utility functions take a scene function as argument. Scene functions are supposed to generate a new scene in the simulation when they are called, and they return the ranges to sample the view matrix parameters. A scene function gets the simulation as argument and has full control over the simulation. It can add and remove objects; load and apply textures; change colors or positions; run multiple simulation steps, e. g., when dropping objects; configure the lightning conditions and define the ranges for sampling the view matrix. Hence, the scene function defines how the created dataset is going to look like.

4.1.5 Domain Randomization Techniques

The previous subsections illustrate most of the possibilities, the simulation has to offer. They have been designed to suit different domain randomization techniques. This subsection quickly summarizes them.

The scene always consists of a bottom plane, which represents the ground, and a table which stands on top of it. The table exists in two variations as shown in figure 4.1, one variation with only one stand in the middle but with rolls at the footing, and the other variation with four table legs, one at every corner. All objects can have textures applied but this will only be done to the table and the bottom. There are 16 different textures available which are shown in figure 4.4. The simulation offers five object types with simple a geometry. These are: sphere, capsule, cylinder, cube, and pyramid, shown in figure 4.2a. It also offers five object types which have a rather complex geometry. These are: spray flask, toothpaste, glass bowl, flower cup, and wineglass, shown in figure 4.2b. One or more instances of these objects can be put in arbitrary positions and orientations. They can even be put above the table to simulate them dropping down. Their sizes are adjusted randomly and each instance has a random color assigned to it. The lightning conditions are randomized. Shadows are always set to be enabled. However, the PyBullet rendering engine renders in few cases none or false shadows with artifacts. There is one light source, its position, direction and color is randomized. Its ambient, diffuse, and specular light coefficients are also randomized. The rendered images are taken by a virtual camera whose position and orientation is randomized. However, it always points roughly in the direction of the table's center, and its position is at least slightly elevated. Section 5.1 presents examples of differently generated artificial datasets and how their randomized scenes look like.

4.2 Model Research

The used models in this thesis were implemented, trained, and evaluated with the help of the Detectron2 [33] framework. Detectron2 is a software system, which has been developed by Facebook AI Research, and it implements various state-of-the-art object detection algorithms. The framework has been written in the programming language Python. It is based on the PyTorch [21] deep learning library and is compatible to its concepts. PyTorch is a deep learning library that allows to utilize GPUs and CPUs for tensor computations. A tensor in PyTorch "is a multi-dimensional matrix containing elements of a single data type." [21] A usual image has three channels, one for red, green, and blue respectively. In PyTorch, a tensor containing such an image of width $W = 512$ and height $H = 512$ would have a shape, i. e., dimensionality of $(3, 512, 512)$, hence more generally: (C, H, W), where $C = amount\_channels$. In the following, the 'size' of a tensor will refer to its width and height dimension, if not otherwise specified. Multiple images (or other data) can be gathered in a batch. If the images or the data have the same dimensions, such a batch can be represented in a single tensor of shape (N, C, H, W), where $N = batch\_size$.

The detectron2 framework can be used as a Python module, i. e., as library. Its concepts and components were designed to be both, easy to use, and highly customizable. State-of-the-art models, such as the Faster R-CNN or Mask R-CNN, are supported in a generalized form which enables them to be easily modified, see subsection 4.2.2. Hence, this library has been chosen to serve as a tool in this thesis to implement the training, testing, evaluation, and visualization of the models, as well as the models themselves.

Subsection 4.2.1 is about `configurations` in detectron2. They allow to separate different models by writing the construction plan for the model, as well as the settings for its training, testing, evaluation and visualization in one configuration file per model. Almost all components in detectron2 are configurable via key-value pairs in a configuration file.

Subsection 4.2.2 explains detectron2's concept of the `generalized r-cnn`, which allows to easily exchange or modify specific parts of a two-stage r-cnn network without changing the rest. All evaluated models were trained as an instance of a generalized r-cnn, except for the auto-encoder, see subsection 4.2.2.

The data used for training, testing, and evaluation is loaded via so called `Dataloaders` which are covered in subsection 4.2.3. Dataloading takes care of mapping the original data to the expected input format of each model. Dataloading also takes care of applying online augmentations to the data. However, dataloaders work only for data that has been registered to the framework. To register data to the framework, it has to be of a specific format, which is also covered in this subsection.

The standard training loop which will be used for the training of each of the models is described in subsection 4.2.4. Subsection 4.2.5 describes the evaluation process of the models including the different metrics that are evaluated.

4.2.1 Configurations

Configuration files play an important role if several different models have to be evaluated. They allow to define arbitrary parameters which often change during the development, training, testing, and evaluation of different models. Consequently, configuration files allow to keep track of different model versions and to keep the different model configurations separated from each other. This makes it easy to try out multiple different approaches without changing the code of the training or evaluation pipeline.

Almost all components which are implemented in the detectron2 framework are configurable via a configuration file[4]. This thesis uses configuration files to define the different models. In addition to the default configuration keys that are being used from the detectron2 framework, several custom keys have been added to configure the customized augmentations during data loading, and to allow the loading of weights for the backbone model, e. g., when using the encoder backbone, see subsections 4.2.3 and 4.2.2 respectively. In addition, a key has been added to specify that a depth image is part of the model input.

The configuration settings that affect the training and evaluation of the different models, and those which vary between the models are specified in section 5.2. Common configuration settings which do not change among the evaluated models are covered in the following subsections.

4.2.2 Generalized R-CNN

Detectron2 defines so called `Meta Architectures`. A meta architecture represents a specific type of model. The meta architecture defines of what different components a model consists, and how they interact with each other. The components may be exchanged or configured individually.

One such meta architecture in detectron2 is the `generalized r-cnn`. It is defined to consist of three components.

[4]The configuration reference for the standard detectron2 config keys. `https://detectron2.readthedocs.io/modules/config.html#config-references` (last visited 12-17-2020)

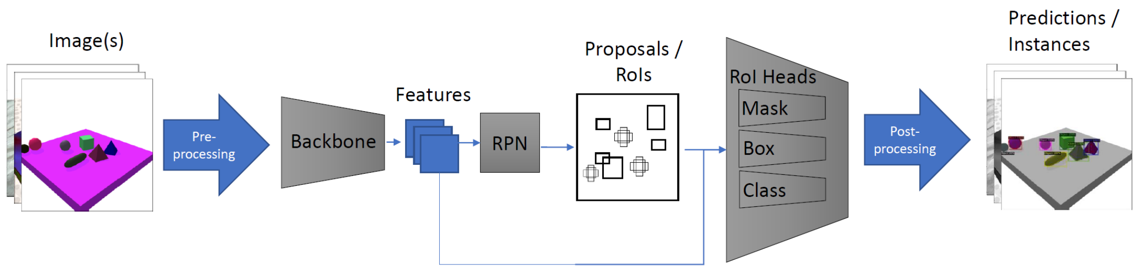


Figure 4.5: The architecture of a generalized r-cnn. The modules `Backbone`, `RPN`, and `RoI Heads` are configurable and even completely exchangeable.

1. A `backbone` which is used for the feature extraction per image. It is usually a convolutional neural network which outputs feature maps, see subsection 4.2.2.
2. A region proposal network (`RPN`) which generates the region proposals / region of interests. The implementation used here is the same as it is used in the Faster R-CNN and Mask R-CNN which is described in subsection 3.2.5. However, it is configured slightly differently.
3. A module which extracts the features for each region of interest, and computes the predictions on those using possibly multiple prediction heads. The implementation of this module is again similar to the Mask R-CNN, see subsection 3.2.6. This module is called `RoI Heads`.

The generalized r-cnn represents two-stage models. The backbone and the RPN form together the first stage. The RoI Heads represents the second stage.

Evaluating the model will do the following:

1. Take a list of input images and preprocess them. Preprocessing normalizes the batch of images and pads the possibly different-sized images to the same size in order to form a single image tensor. This padding respects the size divisibility of the backbone, thus, the images are always padded to a size which is divisible by the stride of the backbone.

   Normalization is done by subtracting the `mean pixel value` and dividing by the `pixel standard deviation`, i. e., $image := \frac{(image - mean\_pixel\_value)}{pixel\_standard\_deviation}$, where subtraction and division is a per-pixel operation, and $mean\_pixel\_value$ and $pixel\_standard\_deviation$ are vectors containing the mean and standard deviation per color channel respectively.
2. Extract features from the images by using the backbone.
3. Use the region proposal network to compute region proposals (region of interests) on the features from the backbone.
4. Compute arbitrary predictions given the features from the backbone and the RPN region proposals using the RoI Heads module per region of interest. Possible predictions are the following: box, classification, and or segmentation mask predictions. The predictions for one roi is summarized in a so called `instance` object.
5. Postprocess the `instances` and scale the predictions back to the target input sizes.

Figure 4.5 illustrates the steps involved of evaluating a generalized r-cnn.

All the evaluated models were trained as instances of a generalized r-cnn but differ in the used backbones and their initialization. See subsection 4.2.2 for the different backbone architectures and 5.2 for the different model configurations. Additionally, the predictions of the RoI Heads will be omitted in some cases in order to evaluate how good the original region proposals already are. Thus, those which are evaluated without the RoI Heads predictions can be considered as one-stage models.

Backbones

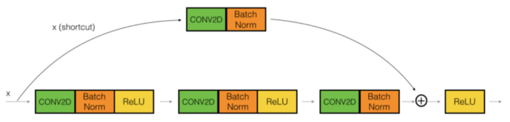
There are different backbone architectures that are used to train, evaluate, and benchmark the generalized r-cnn. These are the ResNet50, the VGG16, and the encoder part of a simple auto-encoder. The results can be seen in section 5.2.

The backbones have different architectures which result in a different amount of resulting feature map channels. The networks differ also in their overall stride, which determines the down sampling factor of the size of the resulting feature map compared to the input size. However, they are all fully convolutional and accept input of any size. Note that 'size' in this context always refers to the width and height dimension of the features, not the amount of channels. Furthermore, the input images have usually three channels as they are mostly RGB images and a specific convolutional layer can only accept a specific amount of channels. Some datasets which are going to be used for training and evaluation have a different amount of channels, e. g., an extra channel for depth information. The backbones are then modified in a way they can still accept this kind of data. This can be done by exchanging their first convolutional layer through an almost identical layer which accepts the expected amount of input channels. This causes the filters of the first convolutional layer to have a different shape, i. e., a different amount of weights, the possibly pretrained weights of a network that used to accept a different amount of channels cannot be used in this layer, and the weights of the first convolutional layers have to be initialized by random.

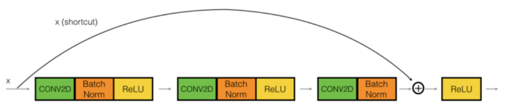
ResNet50

The ResNet50 [10] is a residual network architecture with a depth of 50 convolutional layers. It is a 'very deep convolutional neural network' and has great success in various image classification competitions. As mentioned in section 3.1.1, deep networks have to deal with the problem of vanishing gradients due to the multiplications at each layer. Residual networks tackle this problem by introducing so called skip connections, which allow the gradient to flow through the shortcuts, and therefore prevent it from vanishing. Additionally, skip connections ensure that features extracted by lower level layers are at least as good as features extracted by higher level layers. This is possible because the network is allowed to learn identity functions where the input passes through by using the skip connection and ignores the regular layers.

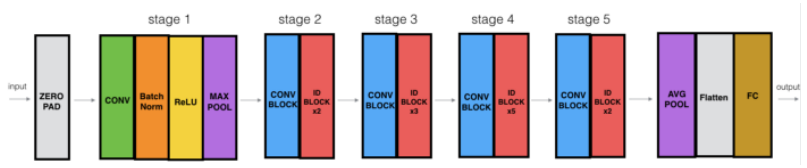
[5] https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33 (last visited 12-17-2020)



(a) A resnet convolutional block with skip connection. The input and output dimensions may be different.



(b) A resnet identity block with skip connection. The input and output dimensions stay the same.



(c) The ResNet50 stages built with identity and convolutional blocks.

Figure 4.6: Illustration of the ResNet50. Subfigure 4.6c shows the ResNet50 with its different stages. The stages consist of (4.6b) identity- and (4.6a) convolutional blocks[5].

ResNet50 consists of multiple stages which are illustrated in figure 4.6c. The first stage is special, let it be called the 'stem' of the residual network. It consists of one convolutional layer which is followed by a batch normalization, ReLU activations, and finally a max pool layer. As it was mentioned before, this stage can be built to accept any number of input channels. The stem is configured to output a 64 channel feature map. The second stage is configured to accept the 64 channel features and to output a 256 channel feature map. The following other stages each consist of a convolutional block followed by multiple identity blocks. The amount of identity blocks varies among the stages. Stage three, four, and five reduce the width and height dimension of its input features by a factor of two. However, they double the amount of input channels. Hence, they each effectively reduce the amount of total features by a factor of 2.

An identity block performs three convolutional layer operations as shown in figure 4.6b. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Identity blocks do not alter the input's dimension. Therefore, the skip connection can be used to directly transfer the input to the output by a simple element-wise addition with the result of the skipped layers.

A convolutional block, as shown in figure 4.6a, is basically the same as an identity block. However, it does alter the dimension of the input. As the input can no longer be added directly to the output because the dimensions do not match, the skip connection needs a convolutional layer of its own to make the dimensions match.

The implementation of the ResNet50 in the detectron2 framework allows to use the output features of any of the five stages. However, only the output features of the fourth stage are used, which is referred to as ResNet-C4 backbone in the original Mask R-CNN paper [9]. Note that in this implementation, the fifth stage is still used to further process the pooled region of interest features w.r.t. the predicted region proposals, see subsection 4.2.2. The ResNet50 backbone, given that the final output are the features of the fourth stage, has a stride of 16 and outputs a 1024 channel feature map at the fourth stage. As it was mentioned in subsection 3.2.2, a stride of 16 means that the resulting feature map's width and height dimension is 16 times smaller than the width and height of the input. Furthermore, the ResNet50 stages are configured to use frozen batch normalization

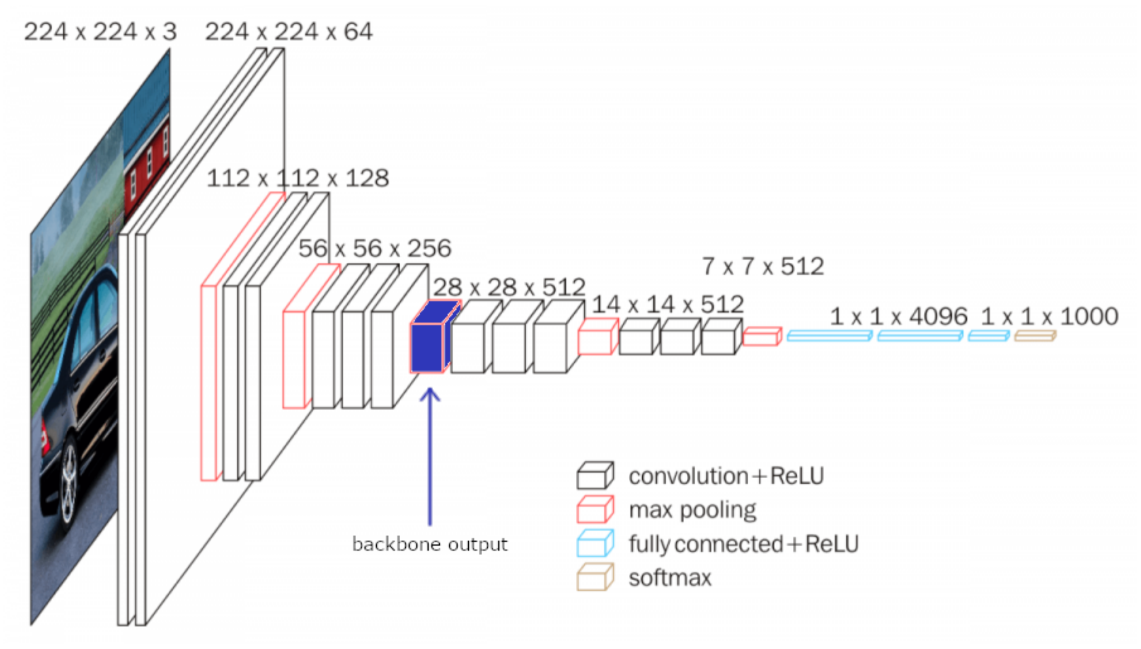


Figure 4.7: The VGG16 architecture.[6] Only the first few layers are part of the backbone, the rest is ignored. The third max pooling layer is marked blue. It is the last layer that will be included in the backbone and thus, it will produce the backbone's output.

layers, i. e., the batch normalization parameters are not learnable but fixed. Hence, if weights of a pretrained model are used for initialization, these parameters will stay the same during training. Here, for a randomly initialized model, a batch normalization layer does only normalize but does not denormalize.

VGG16

The VGG16 [28] is a rather traditional convolutional neural network. It has no skip connections like the ResNet50 does. It was designed for the task of image classification and was state-of-the-art when published. Its architecture consists of multiple convolutional layers, each followed by a ReLU activation function, and multiple max pooling layers to reduce the size of the input. For image classification, its convolutional layers are followed by multiple fully connected layers. Its architecture is illustrated in figure 4.7. Due to the fact that its convolutional layers are followed by fully connected layers, the image expects a fixed sized input. In the original VGG16, images has to be of size $224 \times 224 \times 3$. However, similar to the ResNet50 backbone, the fully connected layers are not used here, which enables the network to process the input independent of the size.

[6] https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33 (slightly modified) (last visited 12-17-2020)

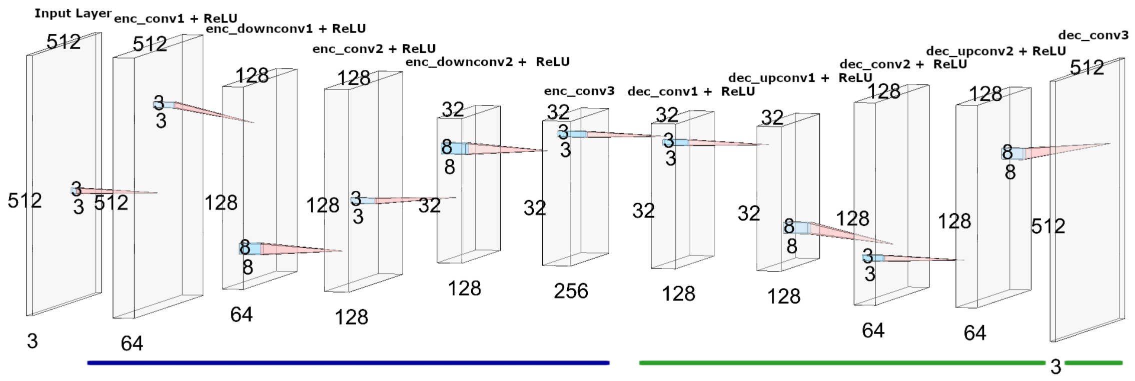


Figure 4.8: The architecture of the auto-encoder.[7] The encoder part on the left is blue underlined and its layers are prefixed with 'enc_'. The decoder part on the right is green underlined and its layers are prefixed with 'dec_'.

As shown in figure 4.7, only the layers up to and including the third max pooling layer are used as a backbone. The convolutional layers do not alter the input's width and height. This backbone has a stride of 8 and produces a 256 channel feature map. Note, that the original VGG16 accepts RGB images with three channels. However, the first convolutional layer can be exchanged to accept a different amount of channels.

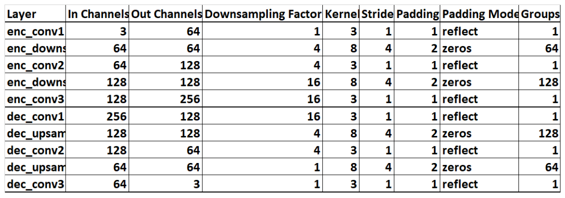
Auto-Encoder

While the ResNet50 and the VGG16 were designed for the task of image classification, an auto-encoder is designed for the task of image encoding and decoding, see subsection 3.1.2. In order to evaluate how a different type of pretrained model performs as backbone in the generalized r-cnn for the task of object detection and instance segmentation, a simple auto-encoder architecture has been developed. It consists of an encoder and decoder part, shown in figure 4.8. Similar to the generalized r-cnn, it is defined to be a meta architecture.

It works as follows:

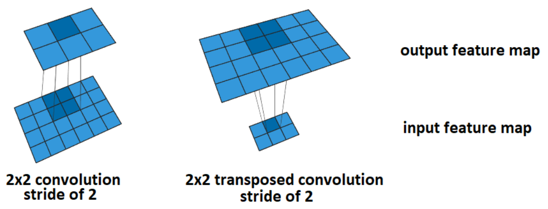
1. Take a list of input images and preprocess them. This step is similar to the preprocessing in the generalized-rcnn. It performs a normalization and padding of the images, which results in a batch of images which are stored in a tensor.
2. Compute the features with the encoder part.
3. Compute the still normalized reconstructed images with the decoder part.
4. Postprocess the reconstructed result and perform denormalization by multiplying with the standard deviation and adding the pixel mean, i. e., $reconstructed = decoder\_output * pixel\_std + pixel\_mean$. Finally, the pixel values are clipped to a range of $[0, 255]$, in order to be in valid color ranges.

[7] created using: `http://alexlenail.me/NN-SVG/AlexNet.html` (last visited 12-17-2020)



Layer	In Channels	Out Channels	Downsampling Factor	Kernel	Stride	Padding	Padding Mode	Groups
enc_conv1	3	64	1	3	1	1	reflect	1
enc_downs	64	64	4	8	4	2	zeros	64
enc_conv2	64	128	4	3	1	1	reflect	1
enc_downs	128	128	16	8	4	2	zeros	128
enc_conv3	128	256	16	3	1	1	reflect	1
dec_conv1	256	128	16	3	1	1	reflect	1
dec_upsam	128	128	4	8	4	2	zeros	128
dec_conv2	128	64	4	3	1	1	reflect	1
dec_upsam	64	64	1	8	4	2	zeros	64
dec_conv3	64	3	1	3	1	1	reflect	1

(a) The specification of the layers in the auto-encoder. Padding mode 'reflect' means that the image is mirrored at the edges as padding. Groups with a value of one is the standard convolutional operation, where all inputs are convolved to all outputs. For the down and upsampling layers, groups is equal to the amount of channels, which means that there is a separate set of filters per channel[8].



(b) The left shows a classical 2d convolution operation with a filter size and stride of two. This results in downsampling with a factor of two. The right shows a 2d transposed convolutional operation, again with a filter size and stride of two. This results in an upsampling operation with a factor of two.

Figure 4.9

Note that the original and postprocessed reconstructed images are used to compute the loss, not their normalized representations.

The auto-encoder is trained using a reconstruction loss between the original and the reconstructed image. In this case, the reconstruction loss is the mean squared error. Let the original image be I and the reconstructed image be $\hat{I}$, then the mean squared error is computed as:

$$(4.1)\quad MSE(I,\hat{I}) = \frac{1}{n} * \sum_{i=1}^{n}(I_i - \hat{I}_i)^2$$

Here, n is the total amount of elements in the input. If the input is a batch of images I_B, the loss L will be computed as:

$$(4.2)\quad L(I_B,\hat{I}_B) = \frac{1}{N} * MSE(I_{Bi},\hat{I}_{Bi})$$

N denotes the amount of images in the batch.

Figure 4.8 shows the complete auto-encoder architecture. The encoder part consists of a total of five convolutional layers. Each convolutional layer, except for the last output layer of the encoder (enc_conv3), is followed by a ReLU activation. The first convolutional layer takes a batch of images. In this case with three input channels and outputs a feature batch with 64 channels, yet the size is preserved. The second layer is used to downsample the size by a factor of four. It uses a stride of four and a kernel size of eight. The third layer doubles the amount of channels, yet again preserves the size, whereas the fourth layer reduces the size by a factor of four, similar to the second layer. The fifth layer again doubles the amount of channels, thus the encoder outputs a feature map with 256 channels and has a total stride of 16.

The decoder part also consists of a total of five convolutional layer. Again, each layer except the last (dec_conv3) is followed by an ReLU activation. Its layers are basically the encoder but back-to-front instead. The first and third layer decrease the amount of channels by a factor of two, and the

[8]https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d (last visited 12-17-2020)

second and fourth layer upsample the size of the input by a factor of four. The last layer takes the 64 channel input map and outputs the reconstructed normalized image with three channels. The exact specification of each layer in the auto-encoder is represented in figure 4.9a.

While the first, third, and fifth layer perform normal convolutional operations, the upsamling of the second and fourth layer occurs by performing a transposed convolution. The transposed convolution simulates a deconvolution operation, i. e., the inverse of the convolution operation. It takes the input feature map and slides a filter over it, similar to the regular convolution. However, at each position on the feature map, only the value at this position is used to multiply with the weights of the filter, which gives the output. Hence, this does not result in one single value but in a field of values in the size of the filter, in which each position in this field corresponds to a weight of the filter multiplied with the single feature value. A stride specifies how many steps this field moves on the output feature map for every step of the filter on the input feature map. Figure 4.9b compares regular 2d convolution with 2d transposed convolution.

Note that by simply changing the first layer of the encoder and the last layer of the decoder, the architecture can be modified to accept any amount of input channels and produce the same amount for reconstruction in the output layer.

Only the encoder part will be used as a backbone for the generalized r-cnn.

RPN

The region proposal network used by the generalized r-cnn models is generally the one used by the Mask R-CNN model. However, it has been slightly modified.

The RPN implementation in detectron2 is also a meta-architecture. It consists of two modules: a module for anchor generation, the anchor generator, and a module for predicting the objectness scores and anchor deltas, the RPN head. The anchor deltas correspond to the region proposal encoding that were mentioned in subsection 3.2.5, and are used to decode or alter the generated anchors which yield the actual region proposals.

The RPN head is used to predict the region proposals. It is very thin and consists of three convolutional layers in total. The first convolutional layer generates a hidden representation of the input features. It is followed by two parallel convolutional layers, of which each takes the hidden representation of the features. One predicts the objectness scores per anchor, and the other one predicts the anchor deltas. Note that both are evaluated only once, thus, they predict the objectness score and anchor deltas for all anchors at once. Suppose there are k reference anchors, then the convolutional layer for objectness scores prediction produces a k channel output of the same size as the feature map, and the convolutional layer for the anchor deltas prediction produces a $k * 4$ channel output which is also of the same size as the feature map. Note that in contrast to the RPN description of the Faster R-CNN in subsection 3.2.5, the implementation here only predicts one objectness score per region proposal. Hence, the objectness 'classification layer' has only k outputs, i. e., channels, instead of $2 * k$.

The evaluation of the overall RPN works as follows:

1. Take the feature map output of the backbone as input.

2. Use the anchor generator to generate k anchor boxes per feature map position. The amount of anchors, i. e., k, results from the different defined anchor sizes and aspect ratios, and is computed as $k = num\_sizes * num\_aspect\_ratios$.
3. Predict objectness scores and anchor deltas using the RPN head.
4. Decode the anchors using the predicted anchor deltas which yields preliminary region proposals.
5. Select a specified amount of top region proposals:
    a) Sort the preliminary region proposals by their objectness score in descending order.
    b) Select the pre_nms_topk proposals and discard the others.
    c) Run non-maximum suppression (NMS) on the remaining proposals.
    d) Sort the remaining proposals again by their objectness score and return the top $post\_nms\_topk$.

The anchor generator which is used here has been configured to use four different sizes, i. e., $num\_sizes = 4$, where the sizes are 32, 64, 128, and 256. The sizes correspond to the square root of the area of the generated anchors in image coordinates. For each size, the anchor generator generates anchors with aspect ratios: 0.5, 1, and 2, i. e., $2 : 1$, $1 : 1$, and $1 : 2$ respectively. Hence, here $k = num\_sizes * num\_aspect\_ratios = 3 * 4 = 12$. Furthermore, pre_nms_topk has been set to be 12000 during training and 6000 during testing. The $post\_nms\_topk$ has been set to 128 during training and to 16 during testing. The threshold for NMS is set to be 0.7. This is the intersection over union (IoU) threshold for discarding region proposals during NMS, see subsection 3.2.2 and 3.2.5 respectively.

Note that only a sampled subset of the generated anchors is used to train the weights of the RPN head and the associated backbone in each training iteration. Here, the subset is of size 256 proposals per image and is sampled in a way that a fraction of approximately 50 percent consists of region proposals labeled as foreground, and the other 50 percent consists of region proposals labeled as background. Region proposals are labeled as foreground if they have an $IoU > 0.7$ with a ground-truth box. If there is no ground-truth box such that they have an $IoU >= 0.3$, they will be labeled as background instead. Proposals that can neither be labeled foreground nor background are completely ignored during the training of the RPN. The RPN contributes to the overall loss of the generalized r-cnn with a L1 regression loss on the anchor delta predictions w.r.t. the corresponding ground-truth deltas, and a binary cross entropy loss on the objectness score predictions w.r.t. the ground truth labels. However, this loss is computed w.r.t. the sampled subset of the generated anchors / proposals and only affects the weights of the RPN head and the backbone during optimization. It does not affect the weights of the following RoI heads.

ROI Heads

The RoI Heads module performs all per-region computations of the generalized r-cnn. Here, it performs the following steps:

1. Only during training: similar to what the RPN does during training, the predicted proposals given from the RPN are labeled as foreground and background proposals, and sampled here for a total of 512 proposals per image, so that a fraction of 25 percent is labeled as foreground and the rest is labeled as background. Here, the threshold for the IoU between a predicted proposal and a ground-truth box to be labeled as foreground is 0.5. Proposals which have no IoU with a ground-truth box greater than 0.5 are labeled as background.

   Note that during testing, the following computations are done on all predictions which result from the RPN. Furthermore, the RoI Heads will only consider proposals with a confidence score greater equal 0.2 in order to reduce the amount of false positives during evaluation.

2. Crop the regions and pool the region of interests, i. e., this extracts the per-region features from the feature map generated by the backbone.

3. Perform the per-region predictions for each of the heads. Here, two different heads are used, one box predictor head which predicts bounding boxes and classes, and one mask head which predicts the segmentation masks. The box predictions of the RoI Heads undergo non-maximum suppression like for the RPN before they are returned, which should remove remaining overlapping proposals. The threshold for here is set to 0.5.

   Note that heads used here have been configured to predict only w.r.t. one 'object'-class or background, i. e., the predictions are class agnostic.

The pooling operation is `RoI Align v2`. It returns fixed sized feature maps of 14×14 resolution, see subsection 3.2.2. `RoI Align v2` is the similar as RoI Align, yet the pixels are shifted by 0.5 for a better alignment of neighboring pixel indices[9]. The pooled region features have a fixed size and are further processed by the fifth stage of the ResNet50 before they serve as input for the different heads. Thus, the heads do share feature computations per region of interest which is illustrated in figure 3.10. The features produced by the fully convolutional fifth stage of the ResNet50 corresponds to the RoI feature vector of the Fast R-CNN head in figure 3.8.

The box predictor consists of the two fully connected layers. These are the layers that follow the RoI feature vector in figure 3.8. One layer for class predictions, and one layer for bounding box regression which again predicts box deltas w.r.t. the region proposals. Note that the RoI Heads have only one object class to predict, which is 'Object', as they are class agnostic. This means that the classification layer has only two outputs, one for 'background' and one for 'Object'. Analogous, the bounding box regression layer has only four outputs because it predicts the box deltas for only one class.

Similar to the RPN, the box predictor contributes with two losses to the overall generalized r-cnn loss. One cross entropy loss computed on the class predictions, and one L1 regression loss w.r.t. the box deltas and region proposal.

For the ResNet50 backbone, the output of the fourth stage is a 1024 channel feature map, which is then accepted by the fifth stage in the RoI Heads. However, using the VGG16 or the encoder part from the auto-encoder as backbone instead, yields only a 256 channel feature map. Therefore, instead of taking the fifth stage of the ResNet50 for the shared feature computation in the RoI Heads,

[9]RoI Align v2 equals the PyTorch RoI Align implementation with `aligned=True`. https://pytorch.org/docs/stable/torchvision/ops.html#torchvision.ops.roi_align (last visited 12-17-2020)

in these cases, the fifth stage of the ResNet18 is used instead. This stage accepts a 256 channel feature map. Due to it being a ResNet18 block, it consists of blocks that only have two convolutional layers instead of three. Furthermore, the fifth stage of the ResNet18 consists of one convolutional block followed by two identity blocks instead of three identity blocks, as it would be the case for the fifth stage of the ResNet50.

Note that the computed losses during training do not contribute to the weight updates of the RPN and the backbone. They only affect the trainable weights of the fifth stage of the ResNet50 (or ResNet18) and the layers of the box and mask head. Hence, even though the total loss of the generalized r-cnn is the sum of the intermediate losses from the RPN and the losses from the RoI Heads, the weights of RPN and backbone are updated and therefore trained separately to the weights of the RoI Heads during the optimization of the total loss.

4.2.3 Dataloading

Loading the data for training and evaluating models consists of two steps.

1. Register the dataset and its metadata to the detectron2 framework. Note that the dictionaries do not contain memory expensive information, such as the data of images, but only reference the filenames of the images on the disk.
2. Create a dataloader object which takes the information from a registered dataset and provides the actual input to the models. The dataloader can modify the data during loading with a data mapper, e. g., apply augmentations, see subsection 4.2.3.

In the configuration file, it can be specified which datasets are to be used for training and which for the evaluation of a model. An additional key in the configuration file also specifies whether to merge the classes when registering a dataset or not. Here, all models do class agnostic prediction, i. e., they only know the classes ’Object’ and ’Background’. Thus, the different objects are all merged to one ’Object’ class.

Step 1 registers the datasets using a function ’load_data_set’ which can read the data and returns it in the expected format of the framework. It also registers the corresponding metadata for the dataset. Note that if a dataset consists of a train, validation, and test set, each set is here considered to be a single dataset. Therefore, the registration of a ’complete’ dataset consisting of a ’train’, ’val’, and ’test’ set, as generated by the simulation (see subsection 4.1.4), is done by registering each of these sets as a different dataset. Datasets need to be registered before a dataloader can access them.

A registered dataset is a list containing the data elements, in which each data element is represented as a dictionary (key-value mapping). The metadata of a dataset is also a dictionary. It contains a list of object names, a list of object colors for the visualization of the predictions, and a mapping from the object ids in the dataset to the internal ids of the classes that correspond to the model outputs. However, since there will be only one class to predict, the names list only contains ’Object’, the corresponding color is yellow, and the object id is just ’0’, whereas background will have the id ’1’. Once a dataset is registered to the detectron2 framework, it can be accessed by a dataloader in order to produce the inputs for the model during training or evaluation.

All raw data comes in the form of samples, see subsection 4.1.2, as it was either generated as samples by the simulation or transformed to the sample encoding, if it came from another source, e. g., the real data. Hence, the 'load_data_set' function operates with samples. It loads all samples, which are stored in a provided directory, by using the 'Sample' class from the simulation, and converts them to the encoding expected from the detectron2 framework. Each sample is transformed into a dictionary which contains the following key-value pairs[10]:

- `file_name` - the path to the RGB image of the sample
- `height` - the height of the RGB image
- `width` - the width of the RGB image
- `image_id` - the sample id
- `annotations` - a list of instance annotations. Each instance in the image corresponds to its own annotation. Note that in a sample, the instances / objects are described by a dictionary which contains `object states`. Each object state corresponds with one instance, thus there is one separate annotation for every object state. An annotation describes the instance's ground-truth and is again a key-value mapping which contains the following:
  - `bbox` - $[x, y, w, h]$, the axis aligned bounding box of the instance in image coordinates, encoded as four values: x and y specify the top left corner of the box in the image, and w and h specify the width and height of the box.
  - `bbox_mode` - a value indicating the encoding of the `bbox`. This is needed as (axis aligned-) bounding boxes can also be encoded differently, e. g., by providing four values that specify the top left and bottom right corner of the box.
  - `category_id` - the category of the instance. Note that when the configuration specifies to merge the classes, the category_id is set to '0' for all instances. Furthermore, the object state of the table, which is also stored in the sample, is ignored.
  - `segmentation` - the RLE encoded segmentation mask of the instance
- `depth_file_name` - the path to the depth image of the sample.

The 'load_data_set' function returns a list containing these dictionaries which is then registered by the detectron2 framework and can later be retrieved by the dataloader.

Step 2 creates a dataloader object. A dataloader takes the information from the registered dataset and processes the data to a format which can be directly used as input by the model. The dataloader consists of a datasampler and a dataset mapper. During training, the datasampler samples the registered dataset and represents an infinite stream of random samples from the dataset. During evaluation, the datasampler iterates only once over the dataset. The dataset mapper is a component which loads the RGB and depth images from the disk into memory and applies the data augmentations that will be used during training or testing, see subsection 4.2.3. It does this for every sampled data dictionary and returns the mapped input for the models.

[10] `https://detectron2.readthedocs.io/tutorials/datasets.html` (last visited 12-17-2020)

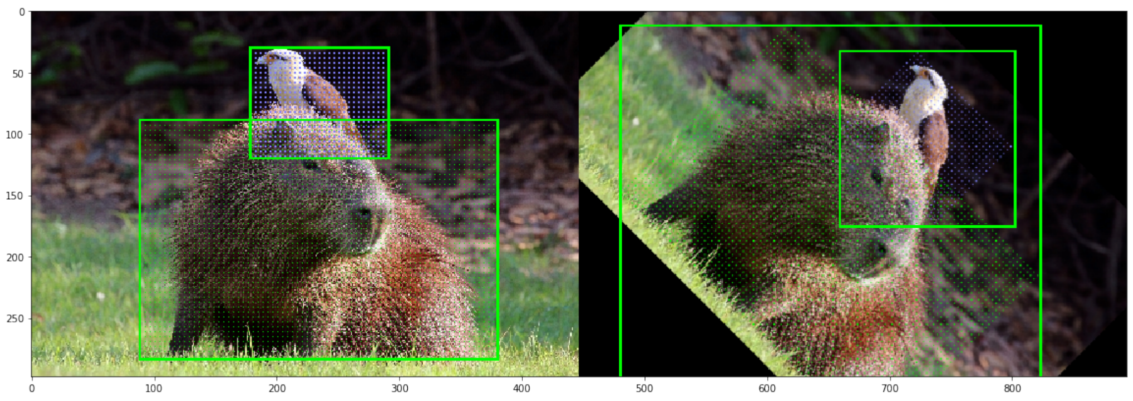


Figure 4.10: The left image shows two axis aligned bounding boxes and the pixels that they encompass. Rotating the image and the corresponding pixels leads to loose axis aligned bounding boxes on the right. They are still computed correctly w.r.t. the pixels they did encompass before, but not w.r.t. the objects they should cover. [14]

During training or evaluation, the dataloader can be iterated. In each iteration it returns a batch of images. For the majority of models, the batch size is eight, but during testing it is always one. The images in the batch have been mapped by using the dataset mapper and they were sampled with the datasampler. This batch is the input for the preprocessing of the generalized r-cnn or the auto-encoder model.

The dataset mapper component has been customized in order to also create single and multi channel input images. This allows to have an additional channel for depth information, i. e., RGB-D images, as well as to use depth information only, i. e., D images. By default, it produces three channel RGB images.

The dataset mapper works as follows:

1. Take the dictionary of a single data element from a registered dataset as input. The dictionary is cloned and the mapping only affects the cloned version.
2. Depending on the information to be used, i. e., , RGB, RGB-D, or D only, load the RGB image and or depth image from the disk by using the corresponding filenames from the data dictionary.
3. When loading a depth image, its values are going to be normalized to a range of $[0.0, 255.0]$ by using one-dimensional linear interpolation.
4. Apply the configured augmentations on the loaded images separately.
    - If RGB and depth images are being used, the augmentations will be applied on the RGB image first. An augmentation is associated with a parameterized deterministic transformation. Applying an augmentation will compute random parameters which define the actual deterministic transformation that has to be applied, and returns this deterministic transformation in which the random parameters have been absorbed into. This allows to apply the exact same transformations to, e. g., the depth image afterwards, too. Multiple augmentations can be applied one after another. They result in an ordered list of deterministic transformations.

   - Depth images only have a single channel. However, some augmentations only work with RGB images which have three channels. Therefore, the depth image are stacked three times which results in a DDD image, i. e., a fake RGB image. The augmentations can then be applied as usual on the DDD image. Afterwards, only the first channel of the result is taken to represent the augmented depth image. Note that the DDD image has still floating point precision before applying the augmentations. However, some augmentations do not work with floating points. Thus, in those cases, the depth image values are converted to integers before applying the augmentations which introduces some information loss.

5. Apply the transformations that result from augmenting the previous image(s) also on the segmentation mask of each instance. The segmentation masks are stored in the annotations of the data dictionary. Note that the segmentation masks are RLE encoded. They are decoded into binary mask images before applying the transformations. Afterwards, they are RLE encoded and stored in the dictionary again.
6. Recompute the bounding boxes by using the corresponding transformed segmentation masks, as described in section 4.1.3. This is needed as some augmentations alter the image geometrically, e. g., rotating or flipping. This results in loose bounding boxes, as shown in figure 4.10. Therefore, they are recomputed by using the new segmentation mask which then again results in tight bounding boxes.
7. Return the altered copy of the data dictionary with the additional keys `image` and `instances`. The key `image` maps to the augmented image. The key `instances` maps to an efficient representation of the augmented annotations that can be processed by the model during training.

Augmentations

The dataset mapper applies different augmentations depending on how it is configured. Note that the augmentations applied by the dataset mapper are online augmentations. This means that this is done during data loading by the dataset mapper. This is different to offline augmentation. Here, the augmented dataset results in a new dataset containing the augmented images, i. e., the augmentations have been stored to the disk. Thus, a model may see the exact same data multiple times during training. In contrast, for online augmentations, each augmentation computes a new randomness for its transformations which likely results in differently augmented data after every augmentation.

The augmentations during training or testing can be configured separately. However, it has been configured that there are no augmentations applied during testing.

The detectron2 framework already provides some augmentations[11] that are partly also used during training. The following lists these augmentations and how they have been configured.

- RandomFlip - performs a horizontal flip with a 50 percent chance on the image

[11] `https://detectron2.readthedocs.io/_modules/detectron2/data/transforms/augmentation_impl.html` (last visited 12-17-2020)

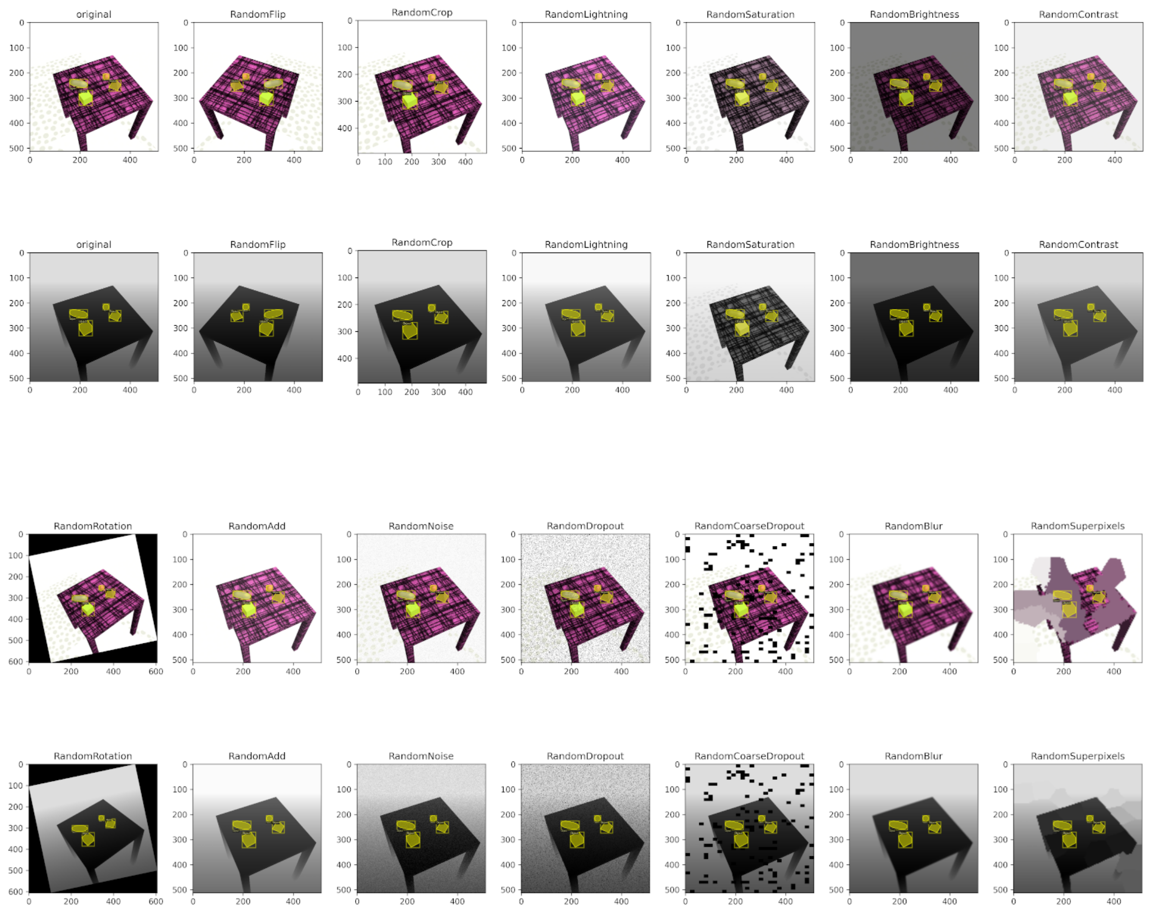


Figure 4.11: The available image augmentations. During training, each data element is augmented by selecting four of the available image augmentations randomly and applying them in a random order, in which each is only applied with a chance of 50 percent.

- RandomCrop - performs a random crop on the image, i. e., it extracts a rectangle region of the original image which gives the augmented image. The size of the rectangle is randomly computed but in the range of 80- to 100 percent of the original image size.
- RandomLightning - performs a 'lightning' augmentation. It uses the 'fixed PCA (principal component ananlysis) over ImageNet' and a normal distribution with a standard deviation (scale) of 150 to sample the degree of color jittering [33].
- RandomSaturation - samples a saturation weight parameter in the range of $[0.25, 1.75]$ and changes the saturation of the image.
- Random Brightness - samples a brightness weight parameter in the range of $[0.25, 1.75]$ and adjusts the overall brightness of the image.
- RandomContrast - samples a contrast weight parameter in the range of $[0.25, 1.75]$ and adjusts the contrast of the image.
- Rotation - rotates the image clockwise or counter-clockwise by an angle sampled from a range of $[0, 30]$ degrees.

Besides the already existing augmentations of the detectron2 framework, the following other augmentations from the imgaug library [14] have been made available for use during training. The following lists these augmentations and how they were configured.

- RandomAdd - samples a value from a given range which is then added to all pixels on all channels. With a 50 percent chance, this value is sampled new per channel. Note that this has been configured to always select the value 30, i. e., there is no range given to sample from.
- RandomNoise - add a noise to the image element-wise, i. e., the noise is computed per pixel, that is sampled from a Gaussian distribution with a scale factor that is sampled from the range $[0, 0.1 * 255]$.
- RandomDropout - zeros a given fraction of all the pixels in the image. This fraction is in the range of $[0, 15]$ percent.
- RandomCoarseDropout - similar to RandomDropout but instead of single pixels, full rectangles of pixels are dropped. This works by performing RandomDropout on a downscaled image and mapping the dropped pixels back to the original size, thus dropping full areas on the actual image instead of single pixels. The RandomDropout is performed on an image that has 5 percent to 10 percent the size of the original image. The RandomDropout drops between 2 percent and 20 percent of the pixels from the downscaled image.
- RandomBlur - blurs the image with a Gaussian kernel filter using a sigma of 3 for the gaussian.
- RandomSuperpixels - splits the image in 50 to 120 segments. With a chance of 50 percent per segment, make it to a superpixel by setting all its pixels to its mean pixel value, or leave them to their original value otherwise.

The effect of applying each augmentation itself is shown in figure 4.11. The dataset mapper does not apply all of the augmentations. Per data element that is going to be mapped, four of all available augmentations are selected, their order randomized, and each of the selected augmentations is only applied with a chance of 50 percent.

4.2.4 Training Loop

Training a model is always done w.r.t. a configuration file which specifies the model and all the parameters related to its training or testing. The training code performs the following steps:

1. Load the configuration file.
2. Register the datasets needed for training or evaluation
3. Build the model as specified in the configuration file.
4. Load pretrained weights of some checkpoint from previous training or initialize the weights by random.

   If the model uses a backbone, its weights can be loaded separately while the rest of the model will be initialized randomly.

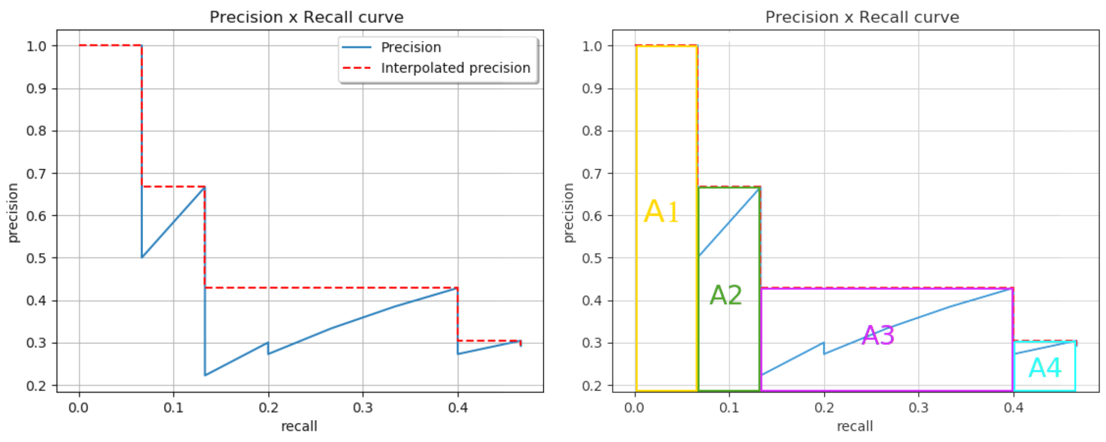


Figure 4.12: The left figure shows an example precision-recall curve with interpolated precision values. The right figure shows the area under the interpolated precision-recall curve. The average precision (AP) is computed as $AP = A1 + A2 + A3 + A4$ [20].

5. Build an optimizer which performs the optimization algorithm including the backpropagation. The optimization algorithm used here is a stochastic gradient descent with momentum, i. e., it is basically a stochastic gradient descent but it uses a moving average of the gradients to compute the weight updates, in which the older gradients decay exponentially w.r.t. the amount of updates since. [26]

6. Build a learning rate scheduler which allows to alter the learning rate based on the number of iterations.

   Often the learning rate starts almost at zero and increases up to the actual learning rate in the first few iterations, which is called ’warm up’. This might prevent ’early overfitting’. Early overfitting means that the model adjusts too heavily on the first few samples of the data and keeps biased during training afterwards.

   The learning rate scheduler might also decrease the learning rate after a specific amount of iterations. This allows to better fine tune on the data by taking smaller steps to the minimum, which usually leads to improved convergence.

7. Build the dataloader used during training.

8. Run the training loop for the specified amount of iterations on the data which is being retrieved from the dataloader. In every iteration, compute the total loss of the model (the sum of all losses returned by it or its sub-modules), and update the weights with the learning rate scheduler and optimizer w.r.t. the total loss.

   In addition, in a specific iteration interval, the current model is evaluated on a small random subset of the validation or testing data.

4.2.5 Evaluation Metrics

Evaluation is performed with the help of the object-detection-metrics library [20]. The library implements the object detection metrics of the Pascal VOC Challenge [5]. The evaluation will compute the following metrics w.r.t. some IoU threshold:

- Total positives - total amount of ground truth objects.
- Total true positives (TP) - total amount of detections that have an $IoU >= threshold$ with some ground truth bounding box. For every ground truth box only the detection with the highest score that still has a sufficient IoU with the ground truth box is labeled as TP. All other detections that have an IoU above the threshold are labeled as FP w.r.t. this ground truth box.
- Total false positives (FP) - total amount of detections that have no $IoU >= threshold$ with any ground truth bounding box
- Total false negatives (FN) - total amount of ground truth boxes that have no $IoU >= threshold$ with any detection. $FN = total\_positives - total\_TP$.
- Total precision (P) - $total\_P = \frac{total\_TP}{total\_TP+total\_FP} = \frac{total\_TP}{amount\_detections}$
- Total recall (R) - $total\_R = \frac{total\_TP}{total\_TP+total\_FN} = \frac{total\_TP}{amount\_ground\_truth\_boxes}$
- Average precision (AP) - 'the precision averaged across all recall values between 0 and 1', instead of taking only eleven points as it is often done.

  Assume that there are n predictions in a list sorted by their confidence score and each has been labeled as TP or FP. For every index i in the list, calculating the accumulated TP and accumulated FP over all predictions from index zero to the current index i, results in two new lists: $accTP$ for the accumulated TP, and $accFP$ for the accumulated FP. The list of precision values is computed via the element-wise operations:
  $accP = \frac{accTP}{accTP+accFP}$. The corresponding list of recall values is computed via the element-wise operation: $accR = \frac{accTP}{total\_positives}$. This yields n precision-recall pairs that can be plotted to a precision-recall curve as shown in figure 4.12. The precision values are interpolated over all recall points where the interpolated precision $p_{interp}(r)$ at recall r is the maximum precision for all recall values greater equal than r. Calculating the area under the interpolated curve yields the average precision (AP).

  The average precision (AP) is computed as: $AP = \sum_{i=0}(r_{i+1} - r_i) * p_{interp}(r_{n+1})$, where i iterates over the index of all **distinct** recall values in $accR$. [20]

Note that the 'total' prefix refers to the summed TP, FP, and FN among all predictions returned by evaluating a dataset.

The used IoU thresholds to compute the above metrics are 0.5 and 0.75, i. e., 50 percent and 75 percent. The metrics will be marked with a trailing '@$\{IoU\}$', e. g., $AP@50$ or $AP@75$.

5 Experiments

This chapter covers the evaluation results on different configuration for models. All model configurations which were evaluated are of the type generalized r-cnn, see subsection 4.2.2.

Section 5.1 first illustrates the different datasets which were used to train or test the models. The training datasets were generated with the simulation from section 4.1. However, for the ablation studies, which are shown in subsection 5.2.2, they differ slightly in the implemented domain randomization techniques. For testing and evaluation, artificial as well as real datasets were used.

Section 5.2 covers the different model configurations and their evaluation results. The model configurations differ in the used backbone, whether they use pretrained weights or not, the use of the RoI Heads module, the artificial training datasets, image format, and the testing datasets.

The model configurations are grouped in three categories: architecture variations 5.2.1 and their benchmarks 5.2.1, domain randomization 5.2.2, and depth information 5.2.3 The results were evaluated with the evaluation metrics of subsection 4.2.5. However, main focus lies on the commonly used average precision metric with a IoU threshold of 50 percent (AP@50).

5.1 Datasets

The following subsections describe the different available datasets. All datasets contain samples as data, see subsection 4.1.2. Therefore, they all have corresponding RGB and depth images as well as ground truth instance annotations with a bounding box and segmentation mask per object instance visible in the images.

5.1.1 Artificial Data

The artificial datasets were all generated with the simulation framework described in section 4.1. Every sampled scene contains a table and a ground plane as well as a single light source. Each scene was sampled out of two different perspectives.

The default **scene function**, which creates all of the scenes for the default artificial datasets ’AD. SO.’ and ’AD. CO.’, is configured to do the following during the generation of a scene:

- Of the two available table designs, which are shown in figure 4.1, one is chosen with a 50 percent chance. The table is placed in the scene with a width and length of 6 units. The table plane is 7 units above the bottom plane. The color for the table is chosen uniformly by chance.
- The table stands on the bottom plane. The bottom plane has a width and length of 20 units. The color of the bottom plane is chosen uniformly by chance.

- Table and bottom plane can each have a texture assigned to them. For both, this texture is sampled uniformly by chance out of all the available textures which are presented in figure 4.4. However, with a 20 percent chance no texture is applied at all. The assigned colors of the table and the plane also influences the appearances of their assigned textures.
- Of all the currently available object classes, which are either simple objects or complex objects, a random amount between zero and four instances are placed in the scene. However, the total amount of instances among all object classes does not exceed eight instances per scene.
- All object instances are placed in a random position above the table plane. In regard to the table plane, each object position is between 0 and 1.5 units away from the table center. Furthermore, each vertical object position is chosen in a height range between 0 and 10 units above the table.
- All instances are randomized in their size, orientation, position, and color. Their sizes are around 0.3 units in width, length, and height, yet they can vary significantly which also depends on the object class. The color is chosen uniformly by chance.
- After placing the objects, the simulation is run until all objects have stopped moving in order to simulate the objects being dropped down on the table. During this simulation, artificial walls are set up to prevent objects from falling over the table edge, yet these walls are removed after the dropped objects have stopped moving.
- The camera position is randomized for every sample. However, the camera position is configured to be at least 6 units and at most 14 units away from the table center, and it is always at least 3 units above the table. It always looks roughly at the center of the table with an offset of up to 1 unit. The camera upwards orientation is $\vec{u} = (0, 0, 1)^T$.
- Shadow casting is enabled.
- Ambient, diffuse, and specular lightning coefficients are chosen randomly from the ranges: $[0.25, 0.75]$, $[0.3, 0.9]$, and $[0.3, 0.9]$ respectively.
- The incoming direction of the light is randomized. However, the light source is always set to be above the table surface, and its position is between 10 to 16 units away from the center of the table surface.

The behavior of this scene functions is slightly modified for the datasets which were used in the ablation studies of the domain randomization techniques.

Artificial Data - Simple Objects (AD. SO.)

This dataset is used for training and validation of the models with different backbones and image formats, see subsection 5.2.1 and 5.2.3 respectively. The available objects are only of simple geometry. These are sphere, capsule, pyramid, cube, and cylinder. It uses shadows and randomized lightning, random textures, random sizes, and random object colors. Figure 5.1 shows some example samples. The 'AD. SO.' training set contains 26214 samples, the validation set contains 3276 samples, and the test set contains 3276 samples. The instances are distributed as follows:

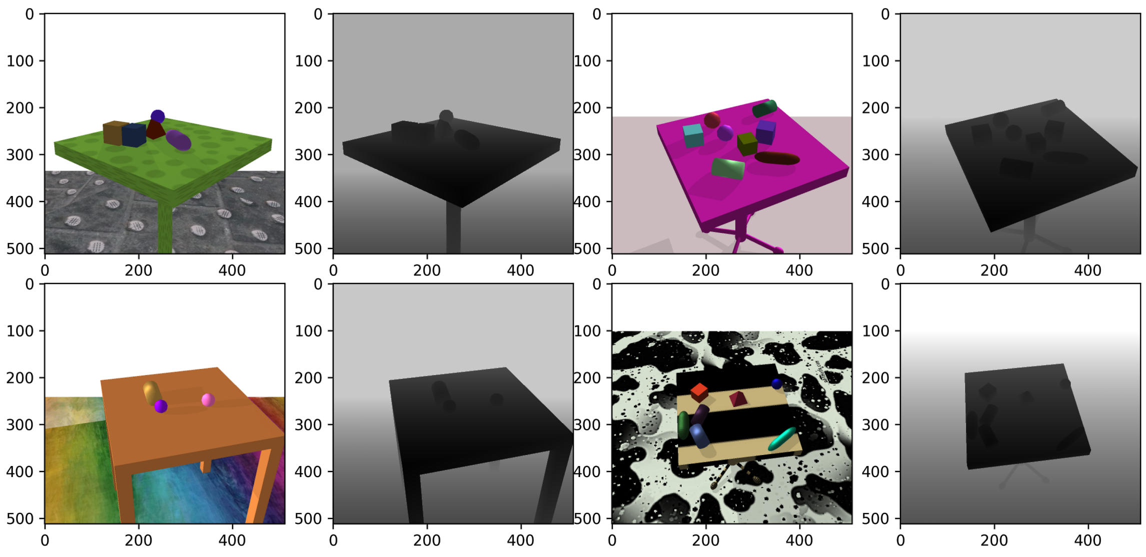


Figure 5.1: Some RGB images and corresponding depth images of samples from the 'AD. SO.' dataset. The 'AD. SO.' dataset implements simple objects, randomized textures, randomized lightning and shadows, and randomized object colors. The left shows the RGB image, the right shows the corresponding depth image in grayscale format after it was normalized to pixel values in the range of $[0, 255]$.

Object	Training Set	Validation Set	Test Set
Capsule	25006	3057	3062
Cube	35225	4514	4303
Cylinder	27560	3480	3508
Pyramid	30967	3856	3790
Sphere	32103	3913	4218
Total	150861	18820	18881

Table 5.1: The distribution of the instances in the training, validation, and test set of the 'AD. SO.' dataset.

Artificial Data - Complex Objects (AD. CO.)

The 'AD. CO.' dataset only differs from the 'AD. SO.' dataset w.r.t. the geometry of its objects. It uses only objects with complex geometry. These are, flower cup, glass bowl, spray flask, toothpaste, and wineglass. It still implements randomized lightning, random textures, and random object colors and sizes. Figure 5.1 shows some of its samples. This dataset is used to evaluate whether the models, which were trained on the 'AD. SO.' dataset with simple geometries, can generalize to this dataset which implements complex geometries instead. It is not used for training. It contains a total of 3276 samples. The instances are distributed as shown in table 5.2.

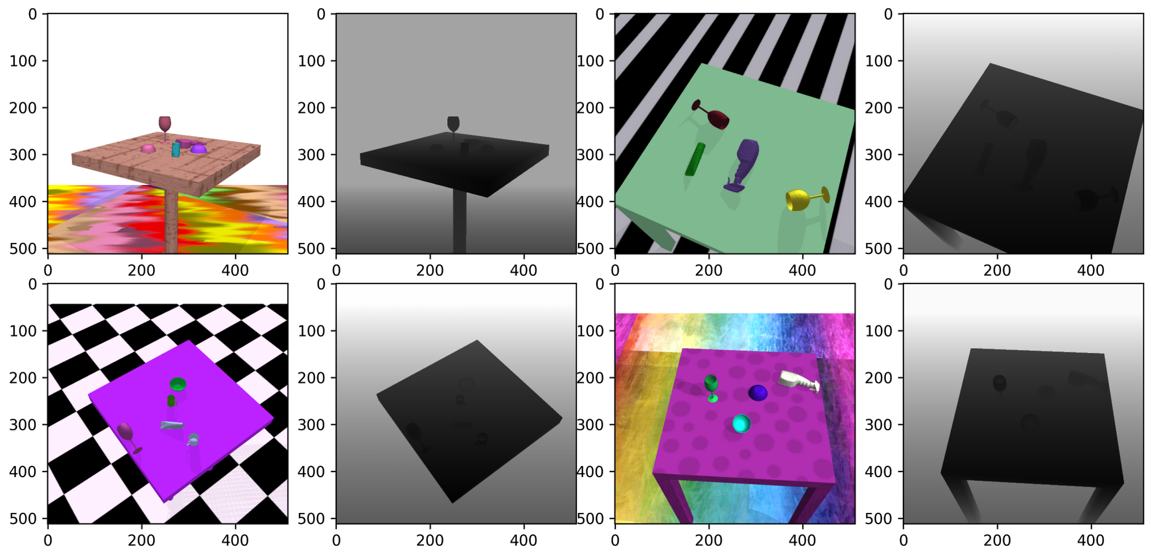


Figure 5.2: Some data samples of the 'AD. CO.' dataset, each with a RGB and corresponding normalized depth image. It is the same as the 'AD. SO' dataset, but it contains objects with complex geometry instead of objects with simple geometry. It still implements randomized textures, randomized lightning and shadows, and randomized object colors and sizes.

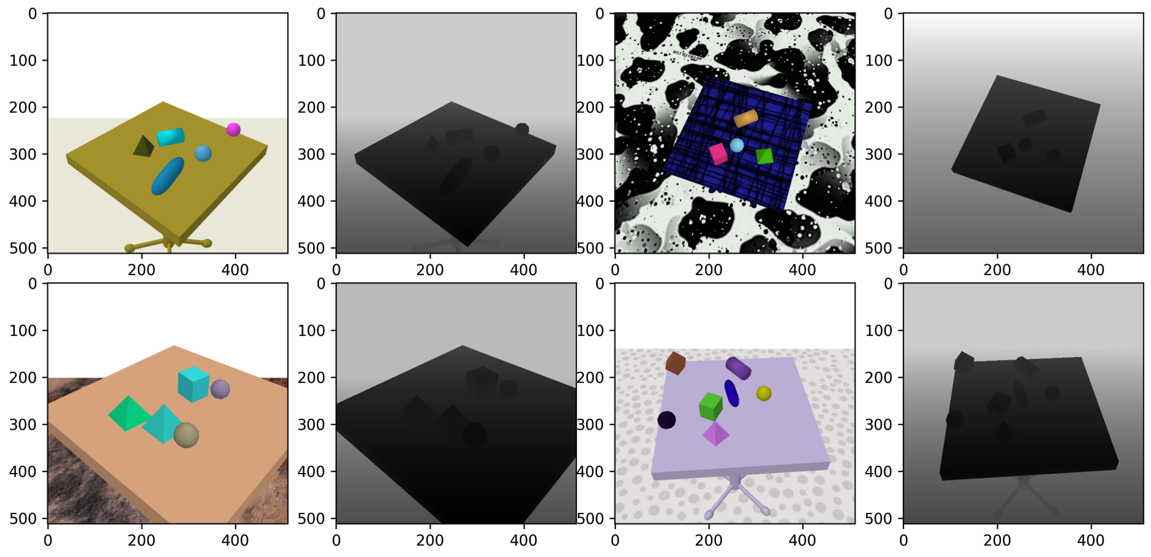


Figure 5.3: Some data samples of the 'AD. NL. SO.' dataset, each with a RGB and corresponding normalized depth image. It is similar to the 'AD. SO.' dataset with the exception that the lightning conditions are not randomized, and shadow casts are disabled.

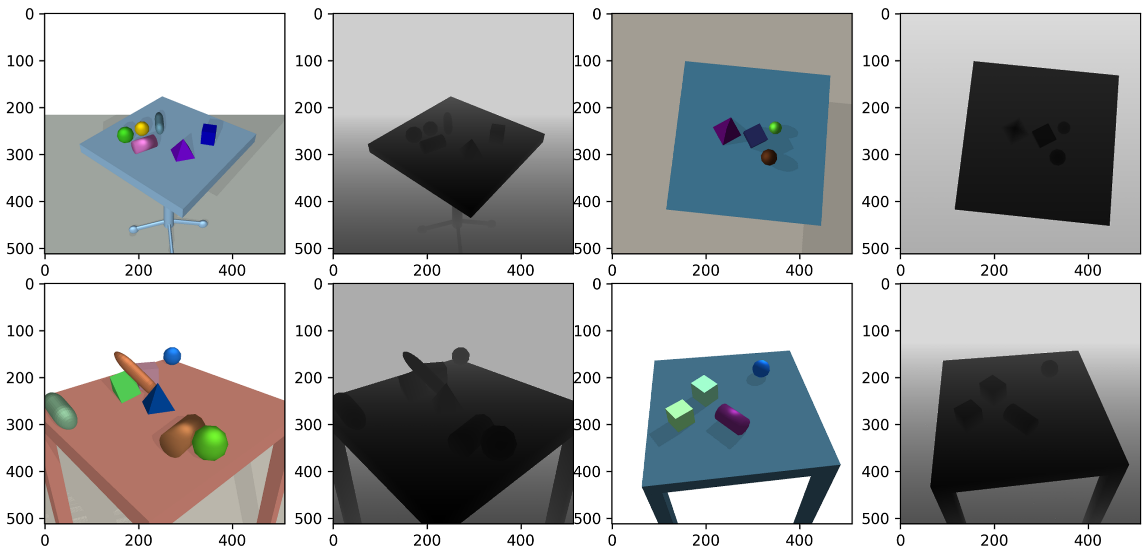


Figure 5.4: Some data samples of the 'AD. NT. SO.' dataset, each with a RGB and corresponding normalized depth image. It is similar to the 'AD. SO.' dataset with the exception that neither table nor bottom plane do have textures applied to them.

Artificial Data without randomized Lightning - Simple Objects (AD. NL. SO.)

The 'AD. NL. SO.' dataset was created by using the same scene function as for the 'AD. SO.' dataset 5.1.1, yet shadows were disabled and the lightning conditions were not randomized. The available objects have simple geometry. Figure 5.3 shows some example samples.

Artificial Data without Textures - Simple Objects (AD. NT. SO.)

Similar to the 'AD. NL. SO.' dataset, the 'AD. NT. SO.' dataset was created by using the same scene function as for the 'AD. SO.' dataset 5.1.1, yet no textures were applied to the table and bottom plane. The available objects are of simple geometry. Figure 5.3 shows some example samples.

Artificial Data without randomized Object Colors - Simple Objects (AD. NRC. SO.)

The 'AD. NRC. SO.' dataset was created by using the same scene function as for the 'AD. SO.' dataset 5.1.1, with the exception that the objects do not have randomized colors. They all are colored light gray to white. The objects are of simple geometry. Figure 5.5 shows some example samples.

5.1.2 Real Data (RD.)

The 'RD.' dataset contains real data. There is always one object instance per sample visible. The available objects are: cube, metal cylinder, plastic cylinder, fruit press, computer mouse, pipe, sec, and tape. Figure 5.6 shows some example samples. It is used for testing and consists of a total of 231 samples. Table 5.2 shows the instance distribution of the overall dataset.

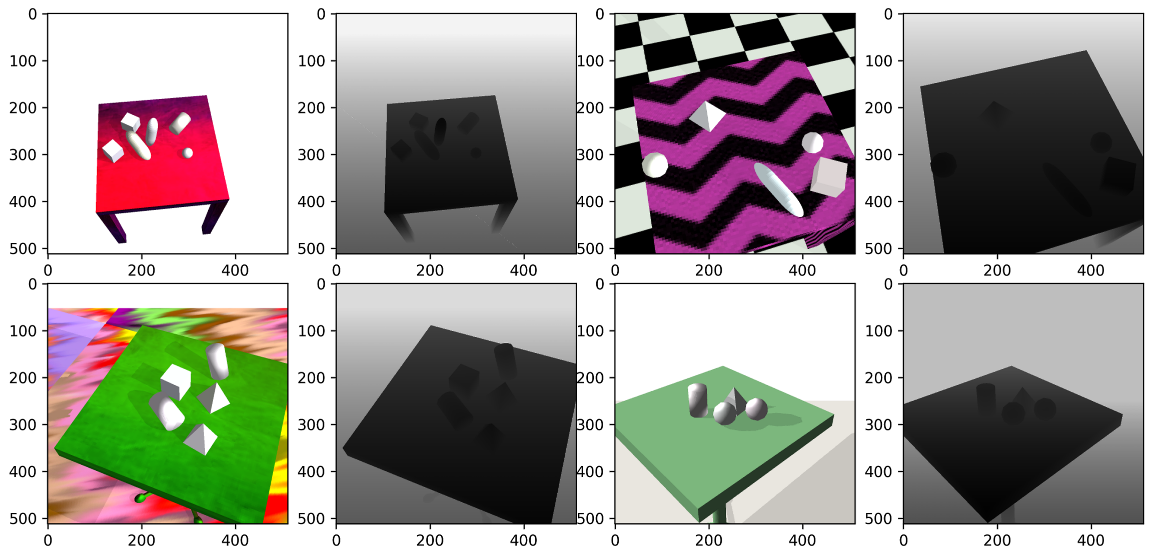


Figure 5.5: Some data samples of the 'AD. NRC. SO.' dataset, each with a RGB and corresponding normalized depth image. It is similar to the 'AD. SO.' dataset with the exception that the objects do not have arbitrarily randomized colors. They are colored light gray to white. However, shadows are cast and lightning conditions, object sizes, and textures are randomized.

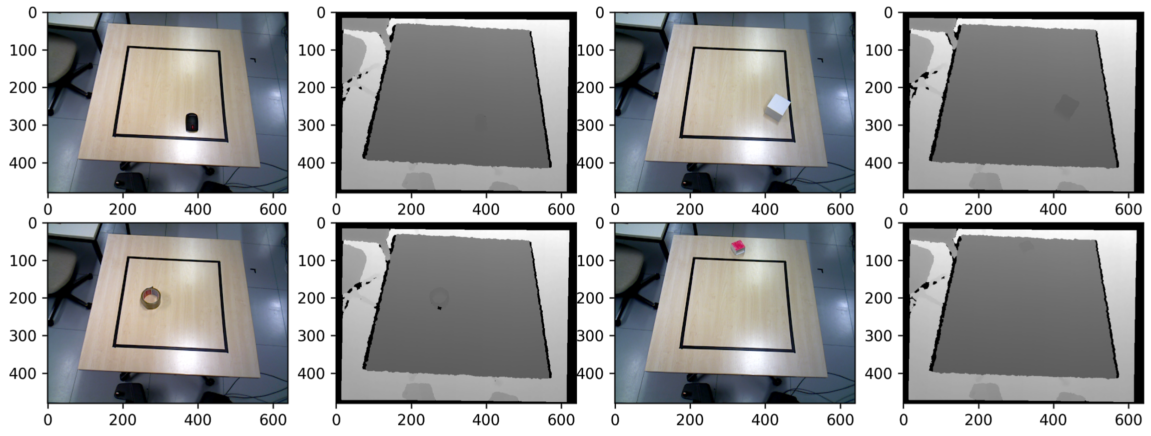


Figure 5.6: Some data samples of the 'RD.' dataset, each with a RGB and corresponding normalized depth image. The real data only ever has one object at a time visible on the table surface.

Object	'RD.' and 'RDC.'	'Mixed'
Capsule	0	865
Cube	23	1292
Cylinder	62	1232
Fruit-press	23	92
Mouse	22	88
Pipe	32	128
Pyramid	0	1050
Sec	44	173
Sphere	0	1214
Tape	25	100
Total	231	6234

Object	'AD. CO.'
Flower Cup	4258
Glass Bowl	3880
Spray Flask	3356
Toothpaste	3130
Wineglass	4186
Total	18810

Table 5.2: The instance distribution in the 'RD.', 'RDC.' and 'Mixed' datasets on the left, and of the 'AD. CO.' dataset on the right.

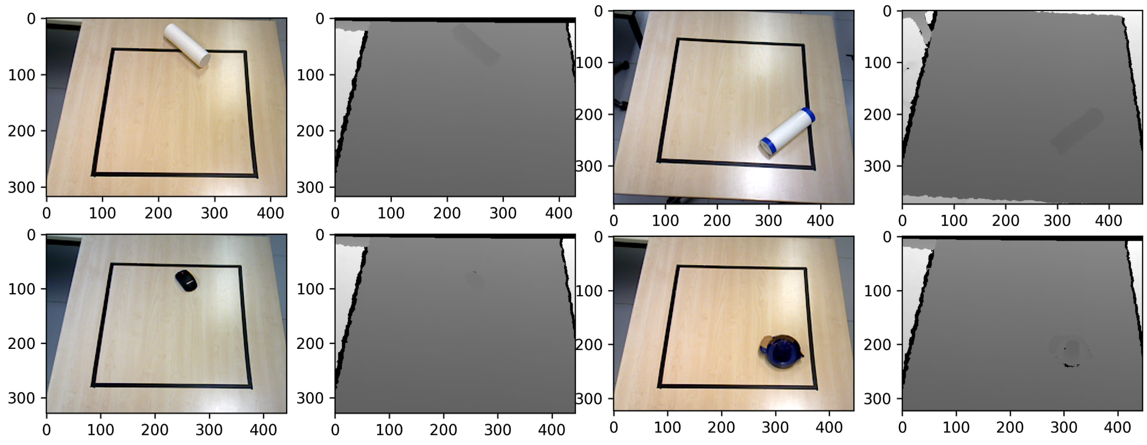


Figure 5.7: Some data samples of the 'RDC.' dataset, each with a RGB and corresponding normalized depth image. It contains the cropped images of the 'RD.' dataset.

Real Data Cropped (RDC.)

The 'RDC.' dataset contains cropped versions of the samples in the 'RD.' dataset. It is a approach to remove eventual false positive predictions in the background. The images have been cropped to the bounding box of the table, yet with some small random offset at each side of the table's bounding box. Figure 5.7 shows some example samples of the 'RDC.' dataset. It has a total of 231 samples likewise the 'RD.' dataset, and it has the same distribution of instances as shown in table 5.2.

5.1.3 Artificial and Real Data Mixed (Mixed)

The 'Mixed' dataset contains artificial as well as real data. The ratio between artificial and real data samples is 50 : 50. The real data of the 'Mixed' dataset is the data of the 'RD.' dataset which has been additionally augmented once with a horizontal flip and twice with a random crop. Hence, the

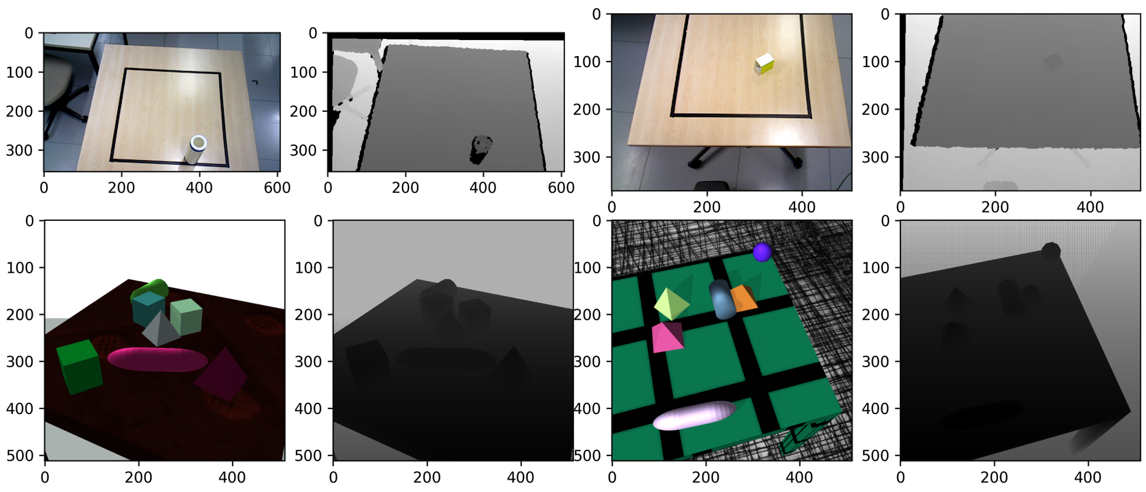


Figure 5.8: Data samples of the 'Mixed.' dataset with RGB and corresponding normalized depth image. The 'Mixed' dataset consists of images from the 'AD. SO.' dataset as well as images from the 'RD.' dataset.

'Mixed' dataset contains four times the amount of real data than the 'RD.' dataset. The artificial data was sampled randomly from the 'AD. SO.' dataset to match the amount of augmented real data. Figure 5.8 shows some examples. The mixed dataset consists a total of 1848 samples. Table 5.2 shows its distribution of instances.

5.2 Model Evaluations

The evaluation of the different model configurations covers three parts. The first part, subsection 5.2.1, investigates the performance of different network architectures. They differ in the used backbones and whether the RoI Heads module is being used or not. It is also examined how the different architectures perform with a pretrained weight initialization.

The second part, subsection 5.2.2, focuses on one specific model architecture. It performs ablation studies on the employed domain randomization techniques. The different models are trained with datasets that differ in one specific characteristic. It also explores the effect of image augmentations during training.

The third part, subsection 5.2.3, examines how different image formats during training and evaluation impact the performance of some models. In particular, two model architectures were in addition trained and evaluated with RGB-D images and D images, and their performance compared to their counterparts that were trained and evaluated on RGB images.

The differently trained models are separated in configurations, see subsection 4.2.1. All models were built as a type of the generalized r-cnn meta architecture which was mentioned in subsection 4.2.2. Their architectures resemble the Mask R-CNN. However, only the box predictions will be considered. In the Mask R-CNN architecture, which is used here, the mask head and the box head of the RoI Heads module share some features. Hence, training with an additional mask prediction loss also influences these features for the box predictor. In [9], it is stated that training with an additional mask loss improves the box predictions. This is why both the mask head and the box head are used as part of the RoI Heads module. Nevertheless, subsection 5.2.1 also investigates the impact the RoI Heads module has on the performance in contrast to just using the RPN predictions directly.

The models are configured as described in section 4.2. However, other configurations which apply to all models are the following:

- The preprocessing step of the models use the following values for normalization. The mean pixel value and standard deviation are based on the ImageNet [4]: $mean\_pixel\_value = [123.765, 116.28, 103.53]$ and $pixel\_standard\_deviation = [58.395, 57.12, 57.375]$, in RGB channel order. If depth data is used, it is normalized with: $mean\_depth\_value = 128.0$ and $depth\_standard\_deviation = 60$. The values for the depth information are leaned on the mean and standard deviation for the RGB values and do not represent the actual mean and standard deviation of the depth data in the datasets.

  Only when using the pretrained weights from the detectron2 instance segmentation baseline in model 1 (see table 5.3), the standard deviation is set to 1.0 on all channels as it has already been absorbed by the weights of the first layer[1].

- All convolutional and fully connected layers in the models are trainable. This includes layers which were initialized with pretrained weights.

- During evaluation, all images are resized by the datamapper to 512×512 pixels in width and height. The resizing process uses bi-linear interpolation. This resizing is a augmentation, thus during training this happens only if the appropriate resizing augmentation has been selected by the datamapper.

- All model configurations are evaluated on the 'AD. CO.,' 'RD.,' 'RDC.,' and 'Mixed' datasets. During evaluation, no image augmentations are applied except for resizing the images.

  Also recall that the evaluation will only consider proposals with a confidence score of at least 0.2 to reduce the amount of false positive predictions.

- Intermediate evaluation results are obtained by evaluating a small subset of 20 random samples every 1000 iterations. This gives insight on how the performance changes with increasing training iterations. Note that these intermediate results only represent a small subset of the actual test set and are more likely to result in spikes or outliers or are shifted for the better or worse due to the smaller amount of evaluated data. However, the smoothed intermediate results on the subsets should at least correlate with the performance on the complete test sets. The smoothing was performed with a 1-dimensional Gaussian filter with $\sigma = 2$.

- The random seed is fixed to 424242.

The optimization during training of the models is configured as follows:

- A mini-batch is of size 8 images.

  Only model 1 in table 5.3 uses a batch size of 10 images instead.

[1] _C.Model.PIXEL_STD in https://detectron2.readthedocs.io/modules/config.html#config-references (last visited 12-17-2020)

ID	Backbone	RoI Heads shared Conv.	Input	Pretrained	Training Set
1	ResNet50	ResNet50 5th stage	RGB	COCO Instance Seg. Baseline[2]	AD. SO.
2	ResNet50	ResNet50 5th stage	RGB	False	AD. SO.
3	VGG16	ResNet18 5th stage	RGB	only the VGG16 backbone[3]	AD. SO.
4	VGG16	ResNet18 5th stage	RGB	False	AD. SO.
5	Encoder	ResNet18 5th stage	RGB	Encoder only[4]	AD. SO.
6	Encoder	ResNet18 5th stage	RGB	False	AD. SO.

Table 5.3: The configurations resemble the Mask R-CNN architecture with different backbones. For each backbone, there exists a model with a pretrained and a model with a random weight initialization.

- The base learning rate is 0.02.

  In the first 1000 iterations, the learning rate is linearly increased via a learning rate scheduler until it reaches the base learning rate at 1000 iterations.
- After 35000 iterations the learning rate is reduced to 0.002, and after 120000 iterations, it is again reduced to 0.0002.
- The maximum amount of iterations is 160000. However, w.r.t. the performance on the validation set, some models were evaluated at earlier iterations, if the performance suggested convergence.
- The optimization algorithm uses gradient descent with momentum. The weight decay is set to 0.0001.

All models were trained and evaluated on a workstation with two `Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz` by utilizing one `NVIDIA GeForce GTX 1080 Ti` GPU.

5.2.1 Backbone Variations

The used backbones vary in structure and type. The ResNet50 is a 'very deep network' and produces a 1024 channel feature map from the input image. This results in a lot of trainable parameters for the overall model. Not only the backbone itself has lots of layers and weights, but also the amount of weights of the RPN and the RoI Heads scale with the amount of channels in the input feature map. The VGG16 and the encoder backbone each have less layers and they both output only a 256 channel feature map. Hence, the amount of parameters is significantly smaller, as it is illustrated in table 5.4. The model architectures do also impact the benchmarking results of their inference, see subsection 5.2.1.

[2]The backbone and the shared convolutions in the RoI Heads are initialized with pretrained weights: the model ID for the detectron2 COCO instance segmentation baseline: '137849525' - https://github.com/facebookresearch/detectron2/blob/master/MODEL_ZOO.md (last visited 12-17-2020)

[3]The weights of the VGG16 have been pretrained on the ImageNet, see https://pytorch.org/docs/stable/torchvision/models.html#id2 (last visited 12-17-2020)

[4]Only the encoder part has been pretrained as part of the auto-encoder and with a reconstruction loss. The pretraining has been performed on the AD. SO. training data.

Module	ResNet50 backbone	VGG16 backbone	Encoder backbone
Backbone	8290304	1735488	383296
RPN	9499708	605500	605500
Model without RoI Heads	**17790012**	**2340988**	**988796**
RoI Heads shared conv.	14942208	8650752	8650752
RoI Heads - Box Head	12294	3078	3078
RoI Heads - Mask Head	2097665	524801	524801
RoI Heads Total	17052167	9178631	9178631
Complete Model	**34842179**	**11519619**	**10167427**

Table 5.4: The amount of trainable parameters of the different modules per architecture style. Note that these values are for models with the RGB input format. The RGB-D input format would result in slightly more, and the D input format would result in slightly less parameters in the backbone due to more weights in the filters of the first layer. The ResNet50 produces a 1024 channel feature map while the VGG16 and the encoder only produce a 256 channel feature map. The amount of channels in the feature map influences the amount of features in the RoI Heads module.

ID	Iter.	AP@50 | AP@75 AD.SO. val.	AP@50 | AP@75 AD.CO.	AP@50 | AP@75 RD.	AP@50 | AP@75 RDC.	AP@50 | AP@75 mixed
1	50000	99.51% | 98.68%	56.45% | 31.20%	**79.30**% | 46.05%	**84.38**% | **46.22**%	**97.35**% | 92.39%
1	160000	**99.52**% | **98.89**%	55.30% | 30.65%	73.69% | **46.44**%	77.01% | 43.68%	96.37% | **92.48**%
2	50000	98.71% | 97.40%	58.58% | 32.83%	59.82% | 35.11%	69.03% | 41.68%	93.91% | 89.54%
2	130000	98.97% | 97.66%	59.53% | 33.67%	60.83% | 38.49%	64.86% | 39.54%	94.14% | 90.14%
3	20000	97.69% | 96.92%	53.42% | 29.99%	30.09% | 19.70%	18.03% | 9.80%	88.50% | 86.15%
3	160000	98.70% | 98.03%	57.40% | 34.07%	34.75% | 24.64%	28.07% | 13.86%	89.93% | 87.52%
4	45000	98.03% | 97.24%	58.02% | 34.12%	38.19% | 23.86%	33.74% | 18.00%	89.86% | 87.32%
4	150000	98.36% | 97.61%	**59.70**% | **36.31**%	34.74% | 23.82%	30.16% | 16.62%	89.49% | 87.27%
5	160000	96.35% | 94.46%	57.80% | 35.27%	56.66% | 25.77%	55.89% | 21.79%	91.28% | 85.28%
6	105000	96.29% | 94.55%	58.20% | 35.32%	53.78% | 19.55%	51.23% | 16.05%	91.07% | 84.80%

Table 5.5: The AP@50 and AP@75 evaluation results of the different models at the specific iterations for different datasets.

Table 5.3 shows the different model configurations in which each configuration corresponds to a different model. Each model architecture exists in a pretrained and non-pretrained variant. In the following, the models will also be referred to by their model id. The model ids are are defined in table 5.3, table 5.9, and table 5.11. All model in this subsection use RGB data as input during training and evaluation. The models were trained on the 'AD. SO.' dataset. Augmentations during training were enabled as described in subsection 4.2.3. For every input image, out of all the available augmentations, four were randomly selected. Each of the selected augmentations was then applied with a 50 percent chance, one after another.

Model 1 uses a ResNet50 backbone which is described in subsection 4.2.2. The weights of its backbone as well as the weights of its shared convolutional layers in the RoI Heads were initialized by a pretrained model. The pretrained model is an instance segmentation baseline of the Mask R-CNN which was trained on the COCO dataset[17]. Model 2 has the same architecture as model 1, yet all its trainable weights were randomly initialized.

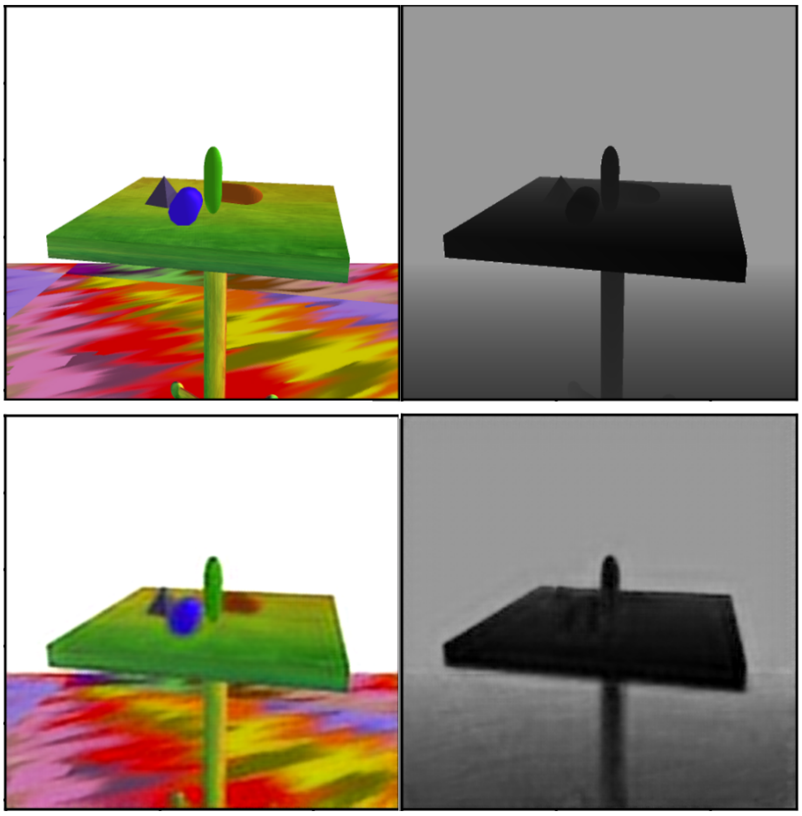


Figure 5.9: The left figure shows the original RGB image above, and the reconstructed RGB image below. The right figure shows the original depth image above, and the reconstructed depth image below. The auto-encoder which was used here was trained on RGB-D data. To illustrate the results, the input and output was split in RGB and D images respectively. It can be seen that the texture in the originally RGB image has an impact on the reconstructed depth image.

Model 3 uses a shortened VGG16 backbone which is described in subsection 4.2.2. Here, only the backbone was initialized from a pretrained model. The rest of the model was initialized randomly. The pretrained model was trained with an image classification task on the ImageNet[4] dataset. Model 4 is has the same architecture as model 3, but all its trainable weights were randomly initialized.

Model 5 uses the encoder part of an auto-encoder, see subsection 4.2.2. The auto-encoder, thus including the encoder, was pretrained on the 'AD. SO.' dataset by using a reconstruction loss. The rest of the model was randomly initialized. The auto-encoder used the same values for normalization as the generalized r-cnn models. Note that the reconstructed normalized images were denormalized with those same values before computing the reconstruction losses. Hence, the encoder backbone expects the same normalized input images as the other backbones.

The auto-encoder was trained with a batch size of 10 images for a total of 140000 iterations. In the first 5000 steps, a linear learning rate warm-up was performed, which reached the base learning rate of 0.000015 at 5000 iterations. After 70000 iterations and after 1000000 iterations, the base learning rate was decreased by a factor of 10. The optimization algorithm used gradient descent with momentum. The weight decay rate was set to 0.005. During training, the same augmentations were used as for the other models in this subsection. Figure 5.9 shows an example of a RGB and depth image of a sample in the 'AD. SO.' validation set as well as the corresponding reconstructed images which were reconstructed by using the trained auto-encoder. Model 6 has the same architecture as model 5, yet all its trainable weights were randomly initialized.

Table 5.5 shows the evaluation results of the different configurations. Some models were evaluated at multiple different iterations. This was done to examine if overfitting on the artificial data occurs. The total loss and average precision on the validation data converges already at early iterations. However, the overall goal is to have the models generalize well to unseen real data and to unseen artificial data with complex objects. Continuing training after convergence on the validation data might result in worse generalization to other domains like the real data or the artificial data with complex objects. Therefore, the models will also be evaluated at earlier iterations when their total loss and average precision on the validation data stops changing and converges.

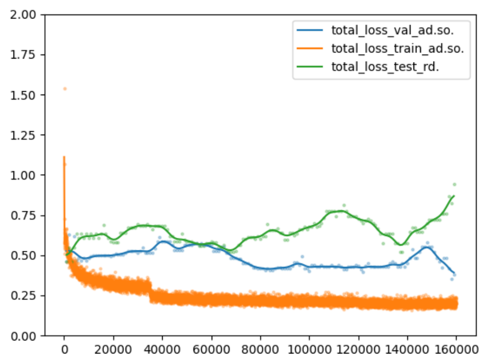
ResNet50 (Model 1 & Model 2)

Figure 5.10 shows the total losses and intermediate AP@50 for model 1 and model 2 on the training and validation set of 'AD. SO.' as well as on 'RD.' at multiple training iterations.

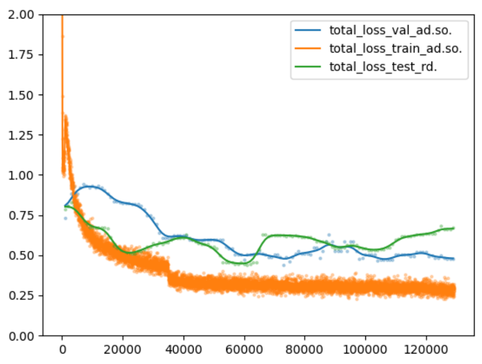
For both models, the total training loss is in overall continuously decreasing and converges after around 35000 iterations. However, the validation loss behaves differently. The pretrained model, i. e., model 1, starts off with a significantly smaller validation loss than model 2, which is only slightly decreasing afterwards. In contrast, the validation loss of model 2 keeps decreasing but still stays in general worse than the validation loss of model 1. The small jump of the training loss at around 35000 iterations is the effect of the learning rate scheduler, which decreased there the learning rate by a factor of 10. This happens again at 120000 iterations, yet this seems to have a less significant effect on the training loss.

The intermediate evaluation of the average precision on the validation set for model 1 shows that it starts off well after only about 1000 iterations, and finally converges at after 50000 iterations. This possibly results due to the initialization of model 1 with pretrained weights. Hence, less iterations are needed for the network to learn good features and correctly predict the objects. In contrast, model 2 was initialized randomly. The intermediate results show that the AP@50 on the validation set starts of low after 1000 iterations but increases fast until around 20000 iterations, where it slows down for a moment until it increases again and finally converges around 50000 iterations.

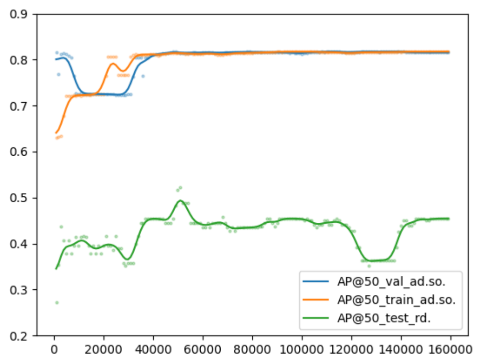
Even though the loss and average precision on the 'RD.' dataset is shown in figure 5.10, these values were not used to select the iterations to evaluate the models. This was done w.r.t. the validation data of the 'AD. SO.' dataset. Also, they represent a small subset and therefore might differ from the actual evaluation result on the complete datasets. However, it can be clearly seen, that model 1 with pretrained weights performs significantly better in early iterations than the randomly initialized model 2. The average precision of model 1 on the 'RD.' dataset stays relatively high even after it has converged on the training and validation set of the 'AD. SO.' dataset. Its peak is around iteration 50000. The AP@50 for model 2 increases fast early and has its peak at around 35000 iterations. Afterwards, it decreases again and does not seem to converge as it keeps alternating increasing and decreasing. A possible reason for it to decrease after 35000 iterations would be that overfitting occurs on the 'AD. SO.' dataset. In contrast, model 1 does not show this behavior. It seems unnatural



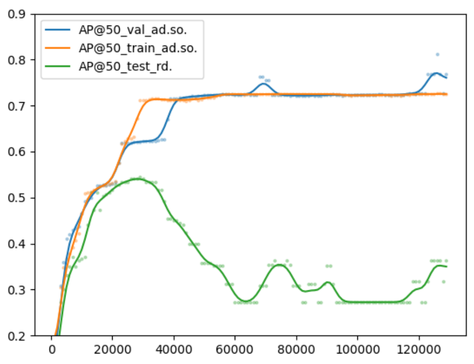
(a) Model 1 total loss



(b) Model 2 total loss



(c) Model 1 AP@50



(d) Model 2 AP@50

Figure 5.10: The upper two images show the total losses on the 'AD. SO' training and validation dataset as well as the total losses on the 'RD.' dataset for model 1 (5.10a) and model 2 (5.10c). The lower two images show the AP@50 from the intermediate evaluation results on the same datasets for model 1 (5.10c) and model 2 (5.10d) respectively.

that the pretrained model 1 has a worse peak AP@50 on the real data as its weights were pretrained on the COCO data set which uses real data too. Hence, the ResNet50 backbone should already be trained well to extract features out of images. However, due to the training on a completely different domain, these well learned weights might be annihilated. Also, the 'AD. SO.' and 'RD.' dataset do indeed have a similar setting. Due to the model 1 being pretrained on a completely different domain, the pretrained weight initialization might even worsen the generalization to the 'RD.' dataset. This would justify the lower peak performance for the AP@50.

The final evaluation results on the different test datasets for model 1 and model 2 are shown in table 5.5. Model 1 was evaluated both at 50000 iterations and at 160000 iterations, while model 2 was evaluated at 50000 iterations and at 130000 iterations. The results endorse the assumption that overfitting occurs on the 'AD. SO.' dataset, i. e., on the domain of the artificial dataset. Model 1 performs on all test sets better with less training iterations. The generalization to complex objects seems to work reasonable well with an average precision (AP@50) of 56.45 percent on the 'AD. CO.' dataset. The generalization on the real data works even better with an AP@50 of 79.30 percent on the 'RD.' dataset. On the cropped real data of the 'RDC.' dataset, it achieves an even higher AP@50 of 84.38 percent, which is possibly due to less visible background which would encourage more false positive predictions. The AP@50 on the 'Mixed' dataset is almost perfect with 97.35 percent. However, this is not suitable to imply the generalization to the real data. Although the amount of samples is distributed evenly among real data and artificial data in the 'Mixed' dataset, each artificial data sample does contain in average four times the amount of instances which leads to four times the amount of predictions on artificial instances, when assuming that the predictions are correct. Hence, the artificial data weighs significantly more in the evaluation of the 'Mixed' dataset.

Model 2 performs on most test sets similar or slightly worse with less iterations, except for the cropped real data (RDC.) dataset. At 130000 iterations, it achieves 59.53 percent on the 'AD. CO.' dataset and therefore generalizes better to it than the pretrained model 1 does. However, for the real data, it performs significantly worse compared to model 1. At 130000 iterations, it achieves 60.8 percent on the 'RD.' dataset and 64.86 percent on the 'RDC.' dataset. Using pretrained weights improves the AP@50 on the 'RD.' dataset by almost 20 percentage points. Overall does model 1 with pretrained weights, outperform its counterpart model 2 which has been initialized randomly. The generalization to complex objects, i. e., the AP@50 on the 'AD. CO.' dataset is better for the non-pretrained model 2.

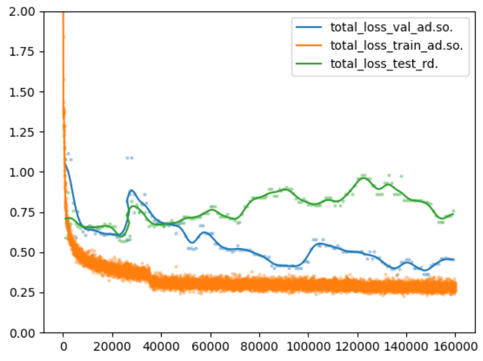
Note that the evaluated average precision of the intermediate evaluations does indeed differ by a significant margin and should not be considered as valid absolute values.

VGG16 (Model 3 & Model 4)

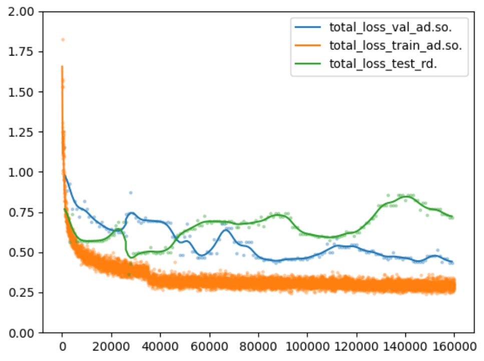
The validation loss of the pretrained VGG16 (model 3) depicted in figure 5.11 behaves likewise the validation loss of the pretrained ResNet50 (model 1) during training. However, it is initially significantly higher but decreases with higher iterations to a similar value. An explanation for model 1 having a smaller loss initially could be that the shared convolutions in the RoI Heads of model 1 were also initialized with pretrained weights, whereas this is not the case for model 3.

The average precision of model 3 on the validation set converges faster than for model 1. It already converges before 20000 iterations even though the validation loss keeps decreasing. The intermediate evaluations on 'RD.' suggest that model 3 achieves here an even higher AP@50 than model 1. However, the actual evaluation results will show that this is not the case. Anyways, the AP@50 peak on the 'RD.' dataset is already at around 5000 iterations, and afterwards it decreases significantly.

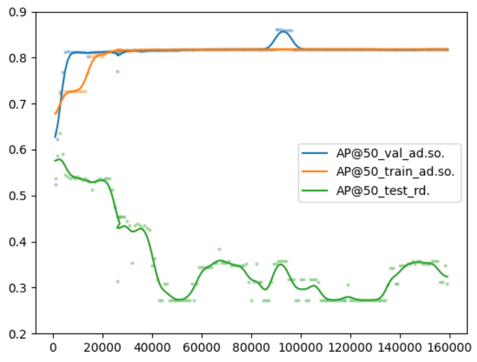
The validation graph of the AP@50 seen in figure 5.11 for the non-pretrained model 4 looks similar to the graphs of the non-pretrained model 2 in figure 5.10. It starts off significantly lower and it takes more iterations to converge. The AP@50 on the validation data for model 4 converges at around 45000 iterations. Again, the AP@50 for the real data does peak early at around 45000 iterations, too, and then decreases again but does not seem to converge.



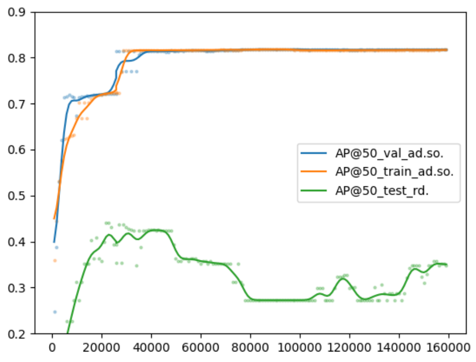
(a) Model 3 total loss



(b) Model 4 total loss



(c) Model 3 AP@50



(d) Model 4 AP@50

Figure 5.11: The intermediate evaluation results of model 3 and model 4. The upper two images show the total loss on the 'AD. SO' training and validation dataset as well as the total loss on the 'RD.' dataset for model 3 (5.11a) and model 4 (5.11d). The lower two images show the average precision with an IoU threshold of 50 percent on the same datasets for model 3 (5.11c) and model 4 (5.11d).

The final evaluation results on the test datasets for both models are also shown in table 5.5. Compared to model 1 and model 2, which use the ResNet50 backbone, model 3 and model 4 only reach less than half of the AP@50 on the real data. On the 'RD.' dataset, the pretrained model 3 reaches only about 34.75 percent at 160000 iterations, and the non-pretrained model 4 reaches about 38.19 percent at 45000 iterations.

The results for model 3 show that it performs worse with less iterations than with more iterations, on the real data, i. e., 'RD.' and 'RDC.' as well as on the 'RDC.' dataset. This contradicts the intermediate evaluations w.r.t. the real data.

The results for model 4 show that at earlier iterations the performance on the real datasets increased, which approves the results of the intermediate evaluations for model 4. However, the intermediate evaluation results do not seem to be reliable for the general case.

Note that the version with less iterations of model 4 is at 45000 iterations while for model 3 it is at 20000 iterations. This might suggest that training for 20000 iterations is not enough while training with the full 160000 iterations is too much and results in overfitting. The checkpoint at 45000 iterations for model 4 might be just between overfitting and not enough training. Overall does model 3 with pretrained weights perform worse on all evaluated datasets than its counterpart model 4 which has been initialized randomly. This is in contrast to the previous models (1 and 2) which both use the ResNet50 backbone instead of the VGG16 backbone, for which the pretrained model did perform significantly better, at least on the datasets which use real data, i. e., the 'RD.,' 'RDC.,' and 'Mixed' datasets.

As for the models with the ResNet50 backbone, the non-pretrained model at high iterations perform best on the 'AD. CO.' dataset. Model 4 achieves an AP@50 of 59.70 percent on the 'AD. CO.' dataset.

Encoder (Model 5 & Model 6)

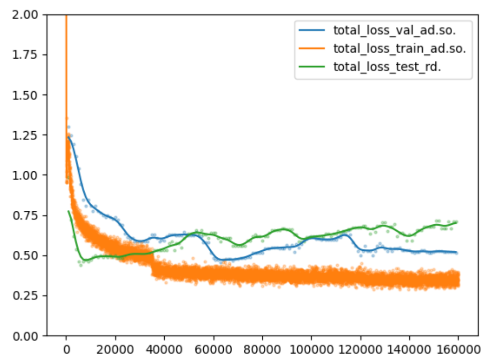
Model 5 and model 6 are equipped with the encoder backbone. In contrast to the previous model pairs, the pretrained model 5 was trained via a reconstruction loss on the training set of 'AD. SO.'. Thus, the encoder was trained to extract important features of the images which enable for a good image reconstruction. However, as depicted in figure 5.12, the validation loss of model 5 even starts higher and decreases slower than the validation loss of model 6. It converges on a significantly higher value than it does for model 6.

This correlates with the measured average precision from the intermediate evaluations. The average precision of model 5 on the validation data does increase slower and converges at 130000 iterations with a significantly lower value than model 6 which already converges after around 30000 iterations. The intermediate evaluations on the 'RD.' datasets seem to correlate with the behavior of the validation loss and the AP@50 on the validation data.

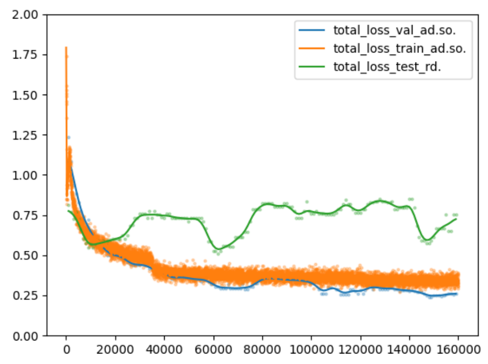
Table 5.5 includes the final evaluation results of model 5 and model 6. The intermediate evaluation results on the validation data, which are shown in figure 5.12, suggest that the non-pretrained model 6 is superior compared to the pretrained model 5. However, the complete evaluation results of model 5 and model 6 show that the non-pretrained model 6 is actually inferior to the pretrained model 5 by a substantive margin of around 3 to 4 percentage points on the AP@50 w.r.t. the 'RD.' and 'RDC.' datasets. Model 5 performs slightly better on the validation dataset and the 'Mixed' dataset, yet with a difference of less than 0.25 percentage points. Only on the artificial dataset with complex objects ('AD. CO.') does the non-pretrained model perform slightly better with an advantage of 0.4 percentage points.

Overall do the models with an encoder backbone achieve significantly better results w.r.t. AP@50 metrics than the models with the VGG16 backbone w.r.t. the real data. Model 5 achieves an AP@50 of 57.80 percent on the 'AD. CO.' dataset, 56.66 percent on the 'RD.' dataset, and 55.89 percent on the 'RDC.' dataset. Model 5 achieves a better AP@50 by 18.47 percentage points than model 4 on the 'RD.' dataset.

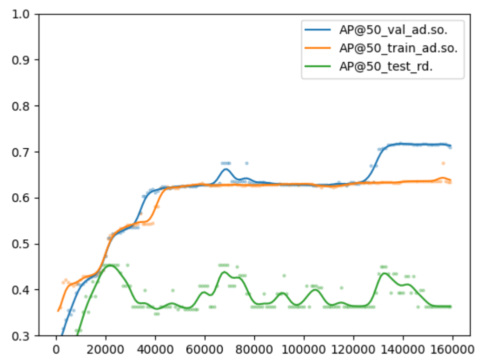
This is notable as the encoder network consists of around 4.53 times less trainable parameters than the VGG16 backbone, as shown in table 5.4. However, the best AP@50 on the 'RD.' dataset is still achieved by model 1 which utilizes the ResNet50 backbone. It achieves 22.64 percentage



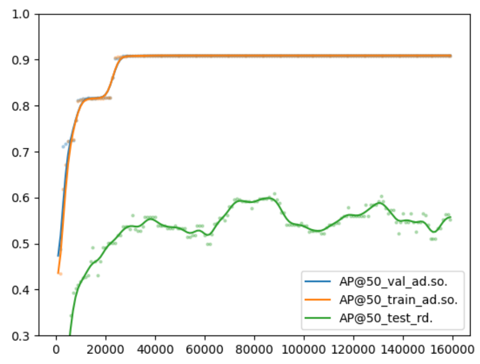
(a) Model 5 total loss



(b) Model 6 total loss



(c) Model 5 AP@50



(d) Model 6 AP@50

Figure 5.12: The intermediate evaluation results of model 5 and model 6. The upper two images show the total loss on the 'AD. SO' training and validation dataset as well as the total loss on the 'RD.' dataset for model 5 (5.12a) and model 6 (5.11d). The lower two images show the average precision with an IoU threshold of 50 percent on the same datasets for model 5 (5.12c) and model 6 (5.12d). Note that even though model 5 seems to be inferior given the intermediate evaluation results, table 5.5 reveals that model 6 performs better in the final evaluations.

points of AP@50 more than model 5 with an encoder backbone does. This is justified as it is the most powerful backbone used here and it was initialized with weights in the backbone and in the RoI Heads module which come from training on a dataset containing real data. Thus, it is not completely unfamiliar with real data and real lightning conditions or other characteristics of a real environment. Note that it has 21.63 times more trainable parameters than the encoder backbone, and 4.78 times more parameters than the VGG16 backbone. It achieves best results on all datasets except for the 'AD. CO.' dataset. Even though the models, which utilize the VGG16 backbone, achieve far worse AP@50 results on the real data than the models with an encoder or ResNet50 backbone, model 4 which uses a randomly initialized VGG16 backbone generalizes best to the 'AD. CO.' dataset. Although, only by a small margin of 0.17 percentage points compared to model 2 and of 1.50 percentage points compared to model 6.

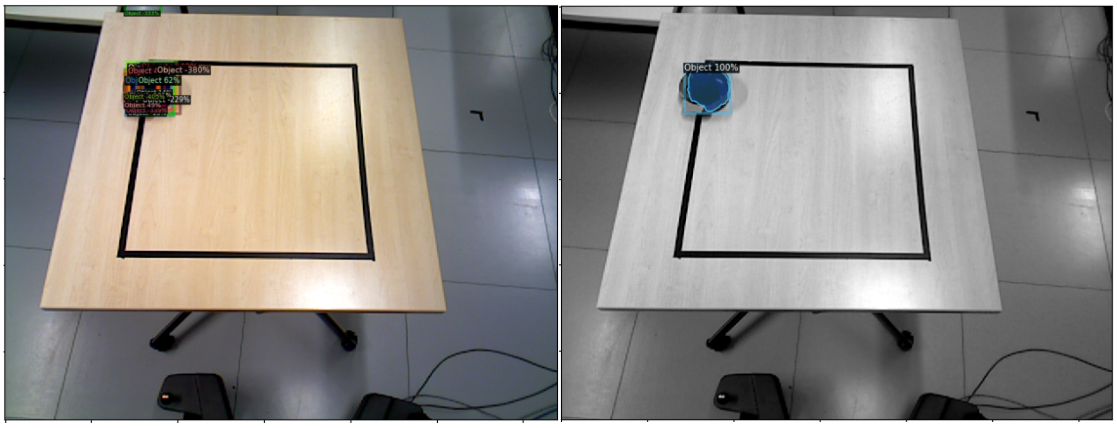


Figure 5.13: Comparison of prediction results of model 1 without the RoI Heads module on the left, and with the RoI Heads module on the right. On the left, the RPN module does always output 16 proposals of which all except one are close to the actual object. On the right, the RoI Heads module did correctly classify the outlying proposal as background and filtered the other overlapping proposals by correcting their bounding box and removing overlaps by applying non-maximum suppression (NMS) to the resulting proposals.

Single Stage Evaluation

This subsection investigates how the previous two-stage models, which include the RoI Heads module, perform compared to their single-stage variants which do not use the RoI Heads module. The evaluation of the single-stage model variants will consider all region proposals returned by the NMS of the RPN module as box predictions. The NMS in the RPN module is configured to always output 16 proposals, and its IoU threshold is 0.7. In contrast, the RoI Heads module does also apply NMS before outputting the final predictions, yet it is not configured to keep a specific amount of predictions and its threshold is only 0.5 which will remove overlapping predictions more aggressively. This will lead to a massive amount of false positive predictions for the single-stage models because the average amount of instances in the data samples is only 1 for the real data and about 4 for the artificial data. Thus, when computing the AP@50 (see subsection 4.2.5) for the real data out of 16 predictions will always be 15 predictions labeled as false positive, while for the artificial data at least 8 predictions but in average 12 predictions will result in false positives. Nevertheless, the computation of the average precision does consider the objectness scores of the proposals. Therefore, the false positives with a lower objectness score will have less of an impact on the overall average precision. Figure 5.13 shows an example prediction of model 1 at 50000 iterations without the RoI Heads and once with the RoI Heads. Applying NMS a second time for the RPN proposals without keeping a specific amount of proposals and with a smaller threshold of 0.5, or appropriately modifying the existing NMS algorithm in RPN module, should result in significantly less false positives and a drastically increased precision for all models which do not use the RoI Heads module. However, this was not implemented here. In the following, only the artificial dataset with complex objects ('AD. CO.') and the real dataset ('RD.') will be considered in the evaluation.

ID	Iter.	RoI Heads	AP@50 | AP@75 AD.CO.	AP@50 | AP@75 RD.
1	50000	True	56.45% | 31.20%	**79.30**% | **46.05**%
1	50000	False	56.87% | 25.46%	74.40% | 34.59%
2	130000	True	**59.53**% | 33.67%	60.83% | 38.49%
2	130000	False	58.03% | 26.60%	69.45% | 26.56%
3	160000	True	57.40% | 34.07%	34.75% | 24.64%
3	160000	False	49.10% | 22.74%	28.91% | 9.21%
4	45000	True	58.02% | 34.12%	38.19% | 23.86%
4	45000	False	50.26% | 23.17%	29.73% | 5.86%
5	160000	True	57.80% | 35.27%	56.66% | 25.77%
5	160000	False	52.38% | 19.38%	49.26% | 7.49%
6	105000	True	58.20% | **35.32**%	53.78% | 19.55%
6	105000	False	52.00% | 18.54%	53.74% | 5.80%

Table 5.6: Comparison of the evaluated average precision of the different models of which each is evaluated once with a RoI Heads module and once without a RoI Heads module. In general, the additional RoI Heads module increases the AP@50 on both 'AD. CO.' and the 'RD.' dataset by up to 8.46 percentage points. Only model 2 seems to achieve a worse AP@50 with the RoI Heads module with a difference of around 8.62 percentage points.

The direct comparison of the evaluated average precision among the models is shown in table 5.6. The counterparts which use the additional RoI Heads module achieve in general a similar or better AP@50 by up to 8.46 percentage points on both evaluated datasets. Only model 2 performs worse with the RoI Heads module than without the RoI Heads module on the real data with a difference of around 8.62 percentage points. The results also show, that the additional RoI Heads module significantly increases the predictions quality w.r.t. the AP@75 metric which uses a higher IoU threshold.

Table 5.7 directly compares the models given the evaluated precision and recall with an IoU threshold of 50 percent, i. e., P@50 and R@50. It shows that the precision without the RoI Heads module and its including NMS is far worse, which can be expected. This is due to the many false positives that result from the fixed amount of 16 proposals which are generated by the RPN module. The overall precision on the 'AD. CO.' dataset compared to the 'RD.' dataset is among the models without a RoI Heads module significantly higher. The higher average amount of instances per image in the 'AD. CO.' dataset contributes to this difference.

The results in table 5.7 also reveal the precision and recall values of the previous models which have RoI Heads modules. They reveal that model 5 with a pretrained encoder backbone and a RoI Heads module achieves the best precision values among the other backbone variations on the 'RD.' dataset with a P@50 of 76.60 percent. It beats the next best model, model 2, which has a P@50 of 71.83 percent by 4.77 percentage points, and the third best model, model 4, which has a P@50 of 70.14 percent by 6.46 percentage points. Table 5.7 shows that all three different backbone architectures can achieve pretty good precision results. However, the encoder and the VGG16 backbone with a RoI Heads module significantly lack in recall performance with a R@50 of 43.72 percent and 62.34 percent respectively, compared to the ResNet50 with a RoI Heads module which still achieves a R@50 of 88.31 percent.

ID	Iter.	RoI Heads	P@50 | R@50 AD.CO.	P@50 | R@50 RD.
(1)	50000	True	58.81% | 77.44%	50.75% | 88.31%
1	50000	False	29.75% | 82.91%	6.14% | **98.27**%
(2)	130000	True	65.27% | 78.11%	71.83% | 66.23%
2	130000	False	30.32% | **84.49**%	6.09% | 97.40%
(3)	160000	True	67.89% | 73.10%	66.92% | 38.53%
3	160000	False	29.49% | 82.18%	4.71% | 75.32%
(4)	45000	True	70.64% | 76.32%	70.14% | 43.72%
4	45000	False	29.38% | 81.87%	4.65% | 74.46%
(5)	160000	True	72.92% | 72.98%	**76.60**% | 62.34%
5	160000	False	29.64% | 82.60%	5.06% | 80.95%
(6)	105000	True	**73.08**% | 73.09%	69.42% | 61.90%
6	105000	False	29.62% | 82.54%	5.49% | 87.88%

Table 5.7: Comparison of the evaluated precision P@50 and recall R@50 for each model once with a RoI Heads module and once without a RoI Heads module. Including the RoI Heads module drastically increases the evaluated precision up to 71.54 percentage points (model 5 w.r.t. the 'RD.' dataset). However, the evaluated recall also decreases by a large margin of up to 36.79 percentage points.

ID	Backbone	RoI Heads	Inference Time (compute time)	Time per Image (compute time)	fps
1	ResNet50	True	1:15 min	0.023198 s/img	43.1 fps
1	ResNet50	False	0:50 min	0.015555 s/img	64.3 fps
3	VGG16	True	1:06 min	0.020372 s/img	49.1 fps
3	VGG16	False	0:51 min	0.015745 s/img	63.5 fps
5	Encoder	True	0:47 min	0.014373 s/img	69.6 fps
5	**Encoder**	**False**	**0:35 min**	**0.010896 s/img**	**91.8 fps**

Table 5.8: The total inference time, average time per image, and the average frames per second (fps) of the different models. The models all perform significantly faster without the RoI Heads module. The timings only refer to the pure compute times which omit an overhead of around 10 seconds on the total inference time per model. It is noteworthy that the VGG16 backbone without RoI Heads performs slower than the corresponding model which use the ResNet50 backbone, even though the VGG16 has less layers and weights.

Model 5 with the non-pretrained encoder backbone and a RoI Heads module achieves the best precision results among the other models on the 'AD. CO.' dataset with a P@50 of 73.08 percent. It again beats model 1 and model 2 which implement the powerful ResNet50 backbone. Thus, when using a RoI Heads module, the encoder backbone does in general perform well w.r.t. the P@50 metric, compared to the other models. In terms of recall, model 1 beats all other models which also use a RoI Heads module, and model 2 beats all other model which use no RoI Heads module. The recall of model 2 without the RoI Heads module is almost perfect with around 98 percent.

Benchmarks

This subsection investigates the difference in inference speed of the different configurations. Deep models like model 1, which uses the deep ResNet50 as backbone, are expected to perform slower during inference than shorter models like model 3 which uses a VGG16 backbone, or model 5, which uses an encoder backbone. See subsection 4.2.2 for a detailed description of these backbones. The same models are also evaluated without the RoI Heads module. This should speed up the inference time for each model as this omits multiple layers and expensive per region computations.

Table 5.8 shows the inference benchmark results on the 'AD. SO.' validation set. The 'AD. SO.' validation set contains 3276 samples, thus 3276 RGB images are evaluated during the inference per model. Recall, the benchmarks for all models were performed on a workstation with two `Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz` while utilizing one `NVIDIA GeForce GTX 1080 Ti` GPU. As expected, for all model pairs, the version without the RoI Heads module is faster due to less computations being made. The pure inference time when using the encoder backbone improves by 25.5 percent from 47 seconds down to 35 seconds without a RoI Heads module. This is equivalent to 91.8 fps, i. e., over 91 images were processed per second. The pure inference time when using the VGG16 backbone improves by 22.7 percent from 66 seconds down to 51 seconds without a RoI Heads module. This is equivalent to 63.5 fps. The pure inference time when using the ResNet50 backbone improves by 33.3 percent from 75 seconds down to 50 seconds without a RoI Heads module. This is equivalent to 64.3 fps.

It shows that the less layers and parameters the backbones have, the faster the inference is. However, model 3 which uses a VGG16 backbone with significantly less layers and parameters than the ResNet50, takes around the same amount of time during inference compared to model 1 which utilizes the ResNet50 backbone, if both do not use the RoI Heads module. A possible reason might be that although the VGG16 has less layers and parameters than the ResNet50, it produces a feature map with less channels but with doubled width and height, i. e., four times the amount of sliding window positions for the RPN module. This likely increases the inference time of the RPN.

Table 5.4 shows the amount of parameters for the different architecture styles. A reason for model 1 with RoI Heads being still slower than model 3 with RoI Heads might be the following: Although the previous stage for both models do perform similar fast and the produced region proposals for both models is always set to be 16 proposals, the pooled features of the ResNet50 do have 1024 channels, whereas for the VGG16 they have only 256 channels. This slows down the computation in the RoI Heads of model 1 as it requires much more parameters than the RoI Heads of model 3. Overall do model 5 and model 6 perform faster than all other combinations. This meets the expectations as the encoder is the smallest backbone network and produces only a 256 channel feature map of the same size as the produced feature maps of the ResNet50 backbone.

5.2.2 Domain Randomization Techniques

This subsection investigates the effect that the available different domain randomization techniques have on the evaluation performance of the model. Therefore, the non-pretrained Mask R-CNN architecture, which uses the ResNet50 backbone, was trained on each of the different datasets presented in subsection 5.1. Each of these datasets has one significant domain randomization

ID	Backbone	RoI Heads shared Conv.	Input	Pretrained	Training Set
(2)	ResNet50	ResNet50 5th stage	RGB	False	AD. SO.
7	ResNet50	ResNet50 5th stage	RGB	False	AD. NT. SO.
8	ResNet50	ResNet50 5th stage	RGB	False	AD. NL. SO.
9	ResNet50	ResNet50 5th stage	RGB	False	AD. NRC. SO.
10	ResNet50	ResNet50 5th stage	RGB	False	AD. SO. (no augmentations)

Table 5.9: The configurations resemble the Mask R-CNN architecture with the ResNet50 backbone, similar to model 2 which is used as reference. The models are all initialized randomly using the same random seed. The training data differs in the employed domain randomization techniques. Model 10 uses the same training data as model 2, yet without online augmentations.

ID	Type	AP@50 | AP@75 AD.CO.	AP@50 | AP@75 RD.	AP@50 | AP@75 RDC.	AP@50 | AP@75 mixed
2	reference	58.58% | 32.83%	59.82% | 35.11%	69.03% | 41.68%	93.91% | 89.54%
7	no textures	57.28% | **33.20**%	**69.27**% | **41.43**%	**79.57**% | **49.96**%	**93.97**% | 88.79%
8	no shadows & fixed lightning	55.85% | 31.31%	32.85% | 20.60%	38.15% | 20.19%	89.53% | 85.91%
9	no random colors	16.15% | 7.34%	35.95% | 15.52%	39.40% | 15.66%	40.28% | 32.00%
10	no augmentations	**59.59**% | 31.04%	**61.34**% | **38.30**%	57.28% | 31.23%	93.76% | 89.51%

Table 5.10: The AP@50 and AP@75 evaluation results of the different models of which each was trained with one specific domain randomization technique missing. Model 2 serves as reference. It was trained on the 'AD. SO.' dataset which implements all the available domain randomization techniques. Furthermore, its training did include online augmentations. The metrics which surpass the corresponding reference of model 2 are marked bold.

technique missing, except the 'AD. SO.' dataset which implements all the available domain randomization techniques and which was used to train model 2. All configuration variations and their corresponding model id are presented in table 5.9:

- Model 2: It was trained on the artificial 'AD. SO.' dataset, and serves as reference. All domain randomization techniques have been employed, including online augmentations during training.
- Model 7: It was trained on the 'AD. NT. SO.' dataset. It is the same as the 'AD. SO.' dataset with the exception that the table and bottom plane do not have any textures applied to them. Its training also includes the online augmentations of model 2.
- Model 8: It was trained on the 'AD. NL. SO.' dataset. It is the same as the 'AD. SO.' dataset with the exception that there is no randomized lightning. The shadows are disabled and the lightning conditions are fixed. Its training also includes the online augmentations of model 2.
- Model 9: It was trained on the 'AD. NRC. SO.' dataset. It is the same as the 'AD. SO.' dataset, however, the instances do not have randomized colors. The colors differ only in a small range and are between white or grey. Its training also includes the online augmentations of model 2.
- Model 10: It was trained on the 'AD. SO.' dataset. The datamapper does not apply any augmentations during training except for resizing the image to a fixed size of 512×512 pixels.

Model 2 was trained for 50000 iterations. The other models were trained for 80000 iterations. The evaluated average precision on the different test datasets for each model is shown in table 5.10. No augmentations, except for resizing the input image, were performed during the evaluations.

Texture Randomizations On / Off

Training without textures seems to significantly increase the average precision on the datasets which contain real data, i. e., ’RD.,’ ’RDC.,’ and ’Mixed.’ For the ’RD.’ dataset, the AP@50 increases by 9.45 percentage points from 59.82 percent up to 69.27 percent. For the ’RDC.’ dataset, the AP@50 increases by a similar margin of 10.54 percentage points from 69.03 percent up to 79.57 percent. The applied textures seem to make the images less recognizable to the network w.r.t. the setting of the real data. They severely change the appearance of the images, and the setting of the real data does not contain such diverse textures. Hence, the data of the artificial dataset ’AD. NT. SO.’ is closer to the real dataset ’RD.’ than the ’AD. SO.’ dataset, and training on the ’AD. NT. SO.’ dataset improves the generalization to the ’RD.’ dataset compared to training on the ’AD. SO.’ dataset.

However, training with the additional textures increases the AP@50 on the artificial ’AD. CO.’ dataset by a small margin of 1.3 percentage points from 57.28 percent up to 58.58. The ’AD. CO.’ dataset does also contain textures. This makes it appear natural that the performance on it increases if the training set also uses textures.

Lightning Randomizations On / Off

Training without shadows and fixed lightning conditions significantly decreases the performance on the real data and slightly decreases the performance on the artificial dataset with complex objects. For the ’RD.’ dataset, the AP@50 decreases by 26.97 percentage points from 59.82 percent down to 32.85 percent. For the ’RDC.’ dataset, the AP@50 decreases by 30.88 percentage points from 69.03 percent down to 38.15 percent. This shows that training without shadows and randomized lightning conditions makes the artificial data appear less real, hence the performance on the real data decreases. However, it might be surprising that these additional randomizations have such a notable impact on the evaluated performance, as the data of the ’AD. SO.’ dataset does not seem to appear that different compared to the ’AD. NL. SO.’ dataset. Thus, shadows and randomized lightning in the input images do contribute to important features for object detection w.r.t. real data.

Without shadows and randomized lightning, the performance on the ’AD. CO.’ dataset does slightly decrease by a small margin of 2.73 percentage points from 58.58 percent down to 55.85 percent. The overall setting of the artificial data in ’AD. SO.’ and ’AD. CO.’ seems to be close enough to still produce similar results.

Object Color Randomizations On / Off

The results show that randomizing object colors does contribute significantly to the performance on the real data, and training without it decreases the AP@50 by a notable margin. For the ’RD.’ dataset, the AP@50 decreases by 23.87 percentage points from 59.82 percent down to 35.95 percent.

ID	Backbone	RoI Heads shared Conv.	Input	Pretrained	Training Set
(2)	ResNet50	ResNet50 5th stage	RGB	False	AD. SO.
12	ResNet50	ResNet50 5th stage	RGB-D	False	AD. SO.
14	ResNet50	ResNet50 5th stage	D	False	AD. SO.
(5)	Encoder	ResNet18 5th stage	RGB	Encoder only	AD. SO.
11	Encoder	ResNet18 5th stage	RGB-D	Encoder only[5]	AD. SO.
13	Encoder	ResNet18 5th stage	D	Encoder only[6]	AD. SO.

Table 5.11: The configurations resemble the Mask R-CNN architecture with the ResNet50 and encoder backbone. The additional models are configured to have RGB images and D images (RGB-D) or depth information only (D) as input.

For the 'RDC.' dataset, the AP@50 decreases by 29.63 percentage points from 69.03 percent down to 39.40 percent. However, randomized lightning conditions and shadows still seems to be of more importance.

The results on the 'AD. CO.' dataset with complex objects are far worse. For the 'AD. CO.' dataset, the AP@50 decreases by 42.43 percentage points from 58.58 percent down to 16.15 percent. This can be expected as the objects do not look alike w.r.t. their color anymore. The 'AD. NRC. SO.' dataset contains only objects which are colored white to grey. With regard to their color, they do rather resemble the background's white skybox instead of colored objects which would justify them to not being predicted well.

Augmentations On / Off

Augmentations introduce a regularization effect, thus it is justified that the performance on a similar artificial dataset decreases. The results show that the augmentations do not contribute to the generalization for complex objects which are in the same artificial setting as the additional augmentations slightly decrease the performance on the 'AD. CO.' dataset by 1.01 percentage points from 59.59 percent to 58.58 percent.

However, w.r.t. the 'RD.' dataset, the additional augmentations do also decrease the performance by a small margin of 1.52 percentage points from 61.34 percent down to 59.82 percent. In contrast, for the cropped real data, i. e., the 'RDC.' dataset, the additional augmentations did significantly increase the average precision by 11.75 percentage points from 57.28 percent up to 69.03 percent. This seems unexpected as both contain real data. A possible yet rather unlikely explanation could be, that random cropping is also part of the augmentations. Thus, when augmentations are enabled, the model trains on more cropped data and in result responses better to the cropped real data.

ID	Input	AP@50 | AP@75 AD.CO.	AP@50 | AP@75 RD.	AP@50 | AP@75 RDC.	AP@50 | AP@75 mixed
(2)	RGB	**58.58**% | **32.83**%	**59.82**% | **35.11**%	**69.03**% | **41.68**%	**93.91**% | **89.54**%
12	RGB-D	58.10% | 30.84%	44.14% | 31.67%	58.83% | 36.11%	91.61% | 88.48%
14	D	29.93% | 7.03%	38.24% | 15.48%	48.50% | 10.18%	91.11% | 85.21%
(5)	RGB	**57.80**% | **35.27**%	56.66% | 25.77%	55.89% | 21.79%	91.28% | 85.28%
11	RGB-D	55.46% | 29.77%	37.12% | 22.17%	36.78% | 15.65%	89.05% | 85.70%
13	D	34.01% | 9.35%	**74.22**% | **50.58**%	**72.65**% | **44.58**%	**94.41**% | **89.76**%

Table 5.12: Model 2, model 12, and model 14 use a ResNet50 backbone. Model 5, model 11, and model 13 use an encoder backbone. The table shows the average precision performance of the models when trained and evaluated with RGB, RGB-D (additional depth information), and D only (only depth information) images.

5.2.3 Depth Information

This subsection investigates the impact of training and evaluation of models which have additional or only depth information as input. Therefore, two backbones were selected. Firstly, the ResNet50 backbone which is used in model 2, model 12, and model 14. Secondly, the encoder backbone which is used in model 5, modelInformatik1, and model 13. Table 5.11 shows the different configurations. The models, which use the ResNet50 backbone, were all randomly initialized, while the models, which use the encoder backbone, had the backbone initialized with pretrained weights. Similar to model 5, as described in subsection 5.2.1, the encoder of model 11 and model 13 were pretrained as part of an auto-encoder with a reconstruction loss on the 'AD. SO.' dataset, each with the appropriate input format. Note that online augmentations were applied on RGB-D and D input, too.

Model 2 was evaluated after 50000 training iterations, and model 12 and model 14 each after 80000 training iterations. Model 5 and model 13 were evaluated after 160000 training iterations, and model 11 after 100000 training iterations. The iteration were chosen w.r.t. the best results of the intermediate evaluations on the validation set of the 'AD. SO.' dataset.

The results of the evaluated average precision on the test datasets are shown in table 5.12. Both backbone variations generalize to the 'AD. CO.' dataset better without any depth information than with depth information. Additional depth information slightly decreases the average precision by 0.48 percentage points from 58.58 percent down to 58.10 percent for the models with a ResNet50 backbone, and by 2.34 percentage points from 57.80 percent down to 55.46 percent for the models with the encoder backbone.

Omitting the RGB data completely, thus using depth information during training and evaluation only, severely worsens the AP@50 on the 'AD. CO. dataset by 28.65 percentage points from 58.58 percent down to 29.93 percent for the models with a ResNet50 backbone, and by 23.79 percentage points from 57.80 percent down to 34.01 percent for the models with the encoder backbone. This suggests that the additional depth information confuses the network w.r.t. the artificial dataset 'AD. CO.' and leads to a worse feature extraction for object detection. When using depth information only,

[5]The encoder is pretrained as part of an auto-encoder with a reconstruction loss on the 'AD. SO.' dataset but including the depth information, i. e., the input was RGB-D data.

[6]The encoder is pretrained as part of an auto-encoder with a reconstruction loss on the 'AD. SO.' dataset but using only the depth information, i. e., the input was D data.

the images of the ’AD. CO.’ and ’AD. SO.’ datasets have even less similarities, due to the missing colors and textures. Thus, the decrease in performance on the ’AD. CO.’ dataset is more significant when using only depth information compared to using RGB and D images.

For the ResNet50 backbone models, the AP@50 with additional depth data also decreases on the ’RD.,’ ’RDC.,’ and ’Mixed’ datasets which contain real data. However, in this case the effects of the additional depth data is more significant. For the ’RD.’ dataset, the AP@50 decreases by 15.68 percentage points from 59.82 percent down to 44.14 percent. For the ’RDC.’ dataset, the AP@50 decreases by 10.2 percentage points from 69.03 percent down to 58.83 percent.

Using depth data only decreases the AP@50 on the real data, when using the ResNet50 backbone, even more. For the ’RD.’ dataset, the AP@50 decreases by 21.58 percentage points from 59.82 percent down to 38.24 percent. For the ’RDC.’ dataset, the AP@50 decreases by 20.53 percentage points from 69.03 percent down to 48.50 percent.

For the models which use the encoder backbone, the results on the real data show a similar effect when using additional depth information. For the ’RD.’ dataset, the AP@50 decreases by 19.54 percentage points from 56.66 percent down to 37.12 percent. For the ’RDC.’ dataset, the AP@50 decreases by 19.11 percentage points from 55.89 percent down to 36.78 percent. However, in contrast to the models which use the ResNet50 backbone, using only depth data does increase the AP@50 on the real data. For the ’RD.’ dataset, the AP@50 increases by 17.56 percentage points from 56.66 percent up to 74.22 percent. For the ’RDC.’ dataset, the AP@50 increases by 16.76 percentage points from 55.89 percent up to 72.65 percent. Thus, the combination of RGB and depth information in one input image leads to worse results for all of the model configurations. However, the results on the models which use the encoder backbone imply that the depth information holds valuable information, as they perform significantly better when trained and evaluated on depth images compared to training and evaluation on RGB images.

6 Discussion

The results in chapter 5 have shows that a generalization to unseen real data or complex objects is possible by training only on artificial data from a simulated environment which contains simple objects. The results of chapter 5 are briefly summarized and discussed in section 6.1.

Section 6.2 discusses some alternative approaches that might further improve the found results. The key insights of this thesis are briefly summarized in section 6.3, and a final conclusion is given in section 6.4.

6.1 Results

The following subsections briefly recap and discuss the results of chapter 5 w.r.t. different topics. It covers the ability of generalization of the evaluated models w.r.t. unseen domains, the effect of using pretrained weights to initialize the models, the impact of using different architectures, and the results of the ablation studies of the employed domain randomization techniques. Furthermore, it covers how additional or less information in the input changes the results.

6.1.1 Generalization Ability

This subsection discusses the generalization ability of models, which were only trained on artificial data and simple objects, to unseen real data and to unseen artificial data with complex objects. The training dataset implemented all introduced domain randomization techniques. Online augmentations for further domain randomization were used during training.

Artificial to Real Data

The results have shown that training on artificial data, which employs the correct domain randomization techniques, allows the models to generalize effectively to unseen real data. The best evaluation result on the real data, with respect to the AP@50 metric and RGB information as input, were achieved by a Mask R-CNN model which utilizes the powerful ResNet50 backbone. It achieves an AP@50 of up to 69.27 percent.

Simple Objects to Complex Objects

The results have also shown that training models on artificial data, which contains only objects of simple geometry and implements appropriate domain randomization techniques, allows to generalize them to artificial data with complex objects. The best evaluation results on the data containing complex objects w.r.t. the AP@50 metric and RGB information as input, was achieved by a Mask R-CNN model with the VGG16 backbone. The achieved AP@50 is 59.70 percent.

6.1.2 The Effect of Pretrained Weights

The initialization of parts of the models with pretrained weights had different effects among the models depending on their architecture. Two models, one with the ResNet50 backbone and one with the VGG16 backbone, were initialized with weights that have been previously trained with tasks on real data. One third model uses an encoder part of an auto-encoder as a backbone. This backbone was initialized with weights, which were learned during the training of the auto-encoder on the artificial dataset with simple objects.

Pretrained on Real Data

Pretraining on real data did improve the AP@50 of the model significantly by using the ResNet50 backbone. The achieved AP@50 on the real test data is 79.30 percent, which makes it the best AP@50 any evaluated model did achieve on this test data. In contrast, the model which uses a VGG16 backbone actually performed worse when initialized with pretrained weights compared to a random initialization. Both backbone variation generalize worse to artificial data with complex objects if they were initialized with weights that were trained on real data.

Pretrained on Artificial Data

The third model uses an encoder backbone which was pretrained with a reconstruction loss on artificial data. The results have shown that this increases the performance on the real data considerably. Thus, pretraining with a reconstruction loss appears to help the model to find good features which also work well on real data. However, it slightly worsens the performance on the artificial data with complex objects.

6.1.3 Architecture Variations

As shown before, different architectures behave differently. In chapter 5, the Mask R-CNN architecture was evaluated in different variations. The used backbones of the Mask R-CNN were the following: the ResNet50; the VGG16; and a encoder network which is part of an auto-encoder. Among these backbones, the ResNet50 is by far the most powerful network with the highest amount of parameters. The VGG16 is rather smaller and has only about a third of the amount of parameters the ResNet50 has. The encoder network is by far the smallest network. Additionally, a variation was evaluated for each backbone which omitted the second stage of the Mask R-CNN. This results in single-stage models which only consist of a backbone and a region proposal network.

In general, the single-stage variations produced many false positives. This is due to a missing additional non-maximum suppression (NMS) at the output of the region proposal network. Nevertheless, they partly achieve similar values w.r.t. the AP@50 metric, even if most perform worse. A non-pretrained model, which uses a ResNet50 backbone, even achieves better results when evaluated as single-stage variant.

Precision vs. Recall

The single-stage variants are far worse w.r.t. the measured precision due to the several false positives. An additional NMS to postprocess the predictions would significantly increase the evaluated precision. However, it would still likely be worse than for the two-stage variants. This is because the NMS does not remove false predictions at different positions but only duplicates at similar positions, whereas the additional RoI Heads module in the two-stage variants can also remove these false predictions.

The results have shown that the single-stage models achieve significantly higher recall values, which shows that the region proposals of the region proposal network cover most objects in the image, even though the generated fixed amount of proposals is small with only 16 proposals per image.

Overall, the two-stage ResNet50 variations work best w.r.t. the recall metric, and they also achieve considerable precision values compared to the other two-stage variants. However, the best precision for two-stage variants is achieved with the encoder backbone. Nevertheless, the recall is in this case slightly worse than for the ResNet50.

6.1.4 Average Precision and Inference Speed Tradeoff

The two-stage models, which use the ResNet50 backbone, are the slowest among the other two-stage variations. They achieve only 43.1 fps. Even though they achieve the best average precision values. The two-stage models with the VGG16 backbone are only slightly faster with 49.1 fps, but achieve significantly worse average precision values compared to the other variations. The two-stage models with an encoder backbone are the smallest and fastest two-stage variation with 69.6 fps, yet they achieve almost as high average precision values as the two-stage ResNet50 variants. Note that if high precision would be more important than high recall, then the encoder models would be the best choice of all three two-stage variants. This is likely to be the case for grasping experiments, since unpredictable movements to arbitrary empty positions are often less acceptable as no movement at all.

If speed is the main factor, single-stage models with an encoder backbone would in this case be the best choice as their inference is the fastest with over 90 fps. As mentioned before, additionally adding NMS would introduce only a tiny overhead, but increases the precision significantly which would make them practical.

Note that the Mask R-CNN in the original paper[9] also uses the ResNet50 backbone, but achieves only about 5 fps during inference. The speed up of the corresponding models here is reasoned by a significantly reduced amount of proposals that are generated by the region proposal network. Here, the RPN outputs only 16 proposals, whereas the RPN of the corresponding original Mask R-CNN outputs 300 proposals during inference. This leads to a massively increased amount of 'per region of interest'-computations in the mask and box head of the model, which slows down inference.

6.1.5 Domain Randomization Techniques

Training with additional textures notably decreases the generalization ability to the real data. It shows that training with textures does in fact has a strong effect on the results, even if it turns out worse, as in this case. Note that the chosen textures did introduce heavy image mutations which let the images appear far less natural and real. However, different textures or applying them in a less dominant way could as well turn this effect around for the better and improve the results with additional textures compared to the results without using textures.

Shadows and randomized lightning make the artificial data appear more real. This additional technique significantly improves the generalization to real data. A similar effect was observed for the randomization of object colors.

The effect of online augmentations during training is less clear. They worsen the performance on the regular real images but notably increase the performance for cropped real images. Similar to the randomized textures, choosing the augmentations carefully might also improve the overall performance for regular real images.

6.1.6 Input Information

Training and evaluating with RGB-D images, i. e., the input contains additional depth information as an extra channel in the image leads to a worse prediction performance compared to training and evaluating with RGB images. However, the same training with D images instead leads to a significantly improved performance compared to RGB images, and an even better performance compared to RGB-D images, at least for models that use the encoder backbone. This implies that depth images do contain valuable information w.r.t. the objects in the image. However, combining a RGB image and D image into one single image does lead to worse results as it might confuse the network due to the different nature of the data in the input.

6.2 Alternatives

The evaluated single-stage models improve the recall and inference speed. As it was mentioned in the results, applying additional non-maximum suppression to the output of the region proposal network might improve the precision significantly. Thus, this represents a plausible alternative, if inference speed matters.

The results have shown that domain randomization techniques are able to affect the generalization ability of models. Furthermore, the results have shown that heavy randomizations and modifications can lead to worse results. Therefore, less severe or sophisticated randomizations such as textures and augmentations could improve the results.

In general, it is rather uncommon to have real depth data available because it is more complex and difficult to be obtained. However, the results have shown that depth images hold valuable information for predicting objects. Even though, including depth data directly in the input image yields worse results and confuses the network. A possible alternative to construct a model, which is independent to whether RGB images, depth images, or both are available during training and evaluation, would be to have another parallel first stage which includes a second backbone and a second region proposal network to the existing one. The second backbone will accept depth images instead of RGB images, while the first backbone will accept RGB images instead. Both generate region proposals w.r.t. their input data. The generated proposals can then be concatenated and serve together as input for the second stage, i. e., the box or mask head. Hence, depending on which data is available, the corresponding 'first stage' is executed, or both, if RGB and depth data is available. The 'first stages' can be executed in parallel and should only add a small overhead. The predictions will benefit from the best of both RGB data and depth data.

6.3 Insights

The evaluation results lead to following insights:

- Training only on artificial data with the correct domain randomization techniques enables models to generalize effectively on unseen real datasets. Thus, domain randomization allows to transfer a model from a simulated environment to the real world.
- The domain randomization techniques should be chosen carefully. More and heavier randomizations do not necessarily contribute to an improved generalization.
- The amount of training iterations does affect the performance of the trained models. Both, training for not enough iterations as well as training for too many iterations, may result in worse results. The sweet spot for the best results will be obtainable in between. However, this sweet spot depends on the goal of how the model should perform and also on which data.
- Intermediate evaluation results are not always reliable.
- Models with fewer parameters may perform almost as well as models which have plenty more parameters.
- Appropriate configurations can considerably speed up the inference of the Mask R-CNN while still achieving pleasant results.
- The initialization of models with pretrained weights usually improves their performance, yet not necessarily.
- RGB and depth images should be considered as separate images to extract higher level features. Combining them into one image leads to worse results.

6.4 Conclusion

This thesis did develop a simulation framework which allows to generate artificial data of a simulated grasping experiment environment. The simulation allows to employ multiple different domain randomization techniques during data generation. With the help of the detectron 2 framework, different model architectures, which are based on the Mask R-CNN network, were trained and evaluated. It was investigated how well the models can generalize to real images and artificial images that contain unseen complex objects. The training was performed only on artificial data with simple objects. Both intentions were found to be successful. Multiple evaluations of combinations with different architectures and employed domain randomization techniques were performed, also in form of ablation studies. They have shown that choosing different architectures leads to a tradeoff of inference speed and prediction quality. The ablation studies give insight about the effect of the employed domain randomization techniques. They show that more randomizations of the artificial training data do not necessarily improve the generalization to real testing data. Furthermore, more input information such as information from depth images does not necessarily improve the performance but can also act worsening, if RGB and depth images are combined into one single image. The evaluation on a real dataset using depth images led to a model with an average precision (at IoU=50) of up to 74.22 percent at 69.6 fps. A different model which uses RGB images instead, achieved an AP@50 of up to 69.27 percent at 43.1 fps on a real dataset. For complex objects, another model achieved an AP@50 of 59.70 percent at 49.1 fps.

Bibliography

[1] F.-J. Chu, R. Xu, P. A. Vela. *Real-world Multi-object, Multi-grasp Detection*. 2018. arXiv: 1802.00520 [cs.RO]. URL: https://arxiv.org/pdf/1802.00520 (cit. on p. 14).

[2] E. Coumans, Y. Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org. 2016–2019. URL: https://github.com/bulletphysics/bullet3 (cit. on p. 31).

[3] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314. URL: https://link.springer.com/article/10.1007/BF02551274 (cit. on p. 17).

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei. “ImageNet: A Large-Scale Hierarchical Image Database”. In: *CVPR09*. 2009 (cit. on pp. 69, 72).

[5] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, A. Zisserman. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html. URL: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/ (cit. on pp. 12, 59).

[6] C.-Y. Fu, W. Liu, A. Ranga, A. Tyagi, A. C. Berg. *DSSD : Deconvolutional Single Shot Detector*. 2017. arXiv: 1701.06659 [cs.CV]. URL: https://arxiv.org/pdf/1701.06659 (cit. on p. 12).

[7] R. Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV]. URL: https://arxiv.org/pdf/1504.08083 (cit. on pp. 12, 26).

[8] R. Girshick, J. Donahue, T. Darrell, J. Malik. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV]. URL: https://arxiv.org/pdf/1311.2524 (cit. on pp. 12, 25).

[9] K. He, G. Gkioxari, P. Dollár, R. Girshick. *Mask R-CNN*. 2018. arXiv: 1703.06870 [cs.CV]. URL: https://arxiv.org/pdf/1703.06870 (cit. on pp. 12, 13, 17, 28, 45, 68, 91).

[10] K. He, X. Zhang, S. Ren, J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/pdf/1512.03385 (cit. on pp. 21, 44).

[11] K. He, X. Zhang, S. Ren, J. Sun. “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition”. In: *Lecture Notes in Computer Science* (2014), pp. 346–361. ISSN: 1611-3349. DOI: 10.1007/978-3-319-10578-9_23. URL: http://dx.doi.org/10.1007/978-3-319-10578-9_23 (cit. on p. 24).

[12] S. Ioffe, C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/pdf/1502.03167v3 (cit. on p. 19).

[13] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, R. Qu. “A Survey of Deep Learning-Based Object Detection”. In: *IEEE Access* 7 (2019), pp. 128837–128868. ISSN: 2169-3536. DOI: 10.1109/access.2019.2939201. URL: http://dx.doi.org/10.1109/ACCESS.2019.2939201 (cit. on pp. 11, 12).

[14] A. B. Jung, K. Wada, J. Crall, S. Tanaka, J. Graving, C. Reinders, S. Yadav, J. Banerjee, G. Vecsei, A. Kraft, Z. Rui, J. Borovec, C. Vallentin, S. Zhydenko, K. Pfeiffer, B. Cook, I. Fernández, F.-M. De Rainville, C.-H. Weng, A. Ayala-Acevedo, R. Meudec, M. Laporte, et al. *imgaug*. https://github.com/aleju/imgaug. Online; accessed 01-Feb-2020. 2020. URL: https://github.com/aleju/imgaug (cit. on pp. 30, 54, 57).

[15] A. Kasper, Z. Xue, R. Dillmann. “The KIT object models database: An object model database for object recognition, localization and manipulation in service robotics”. In: *The International Journal of Robotics Research* 31.8 (May 2012), pp. 927–934. DOI: 10.1177/0278364912445831 (cit. on p. 36).

[16] T.-Y. Lin, P. Goyal, R. Girshick, K. He, P. Dollár. *Focal Loss for Dense Object Detection*. 2018. arXiv: 1708.02002 [cs.CV]. URL: https://arxiv.org/pdf/1708.02002 (cit. on p. 12).

[17] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, P. Dollár. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV] (cit. on p. 71).

[18] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A. C. Berg. “SSD: Single Shot MultiBox Detector”. In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2 (cit. on p. 12).

[19] Y. Lu, J. Lu. *A Universal Approximation Theorem of Deep Neural Networks for Expressing Distributions*. 2020. arXiv: 2004.08867 [cs.LG]. URL: https://arxiv.org/pdf/2004.08867 (cit. on p. 17).

[20] R. Padilla, S. L. Netto, E. A. B. da Silva. “A Survey on Performance Metrics for Object-Detection Algorithms”. In: *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*. 2020, pp. 237–242 (cit. on pp. 58, 59).

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 41).

[22] J. Redmon, S. Divvala, R. Girshick, A. Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV]. URL: https://arxiv.org/pdf/1506.02640 (cit. on pp. 12, 13).

[23] J. Redmon, A. Farhadi. *YOLO9000: Better, Faster, Stronger*. 2016. arXiv: 1612.08242 [cs.CV]. URL: https://arxiv.org/pdf/1612.08242 (cit. on pp. 12, 13).

[24] J. Redmon, A. Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767 [cs.CV]. URL: https://arxiv.org/pdf/1804.02767 (cit. on p. 12).

[25] S. Ren, K. He, R. Girshick, J. Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: 1506.01497 [cs.CV]. URL: https://arxiv.org/pdf/1506.01497 (cit. on pp. 12, 27, 28).

[26] D. E. Rumelhart, G. E. Hinton, R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0 (cit. on p. 58).

[27] P. Schmidt, N. Vahrenkamp, M. Wächter, T. Asfour. "Grasping of Unknown Objects Using Deep Convolutional Neural Networks Based on Depth Images". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 6831–6838. DOI: 10.1109/ICRA.2018.8463204. URL: https://h2t.anthropomatik.kit.edu/pdf/Schmidt2018.pdf (cit. on p. 14).

[28] K. Simonyan, A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV]. URL: https://arxiv.org/pdf/1409.1556 (cit. on pp. 15, 21, 46).

[29] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, P. Abbeel. *Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World*. 2017. arXiv: 1703.06907 [cs.RO]. URL: https://arxiv.org/pdf/1703.06907 (cit. on pp. 14–16, 29, 30).

[30] M. Toussaint. *Introduction to Machine Learning (Lecture Slides)*. URL: https://www.user.tu-berlin.de/mtoussai//teaching/Lecture-MachineLearning.pdf (cit. on p. 17).

[31] J. Uijlings, K. Sande, T. Gevers, A. Smeulders. "Selective Search for Object Recognition". In: *International Journal of Computer Vision* 104 (Sept. 2013), pp. 154–171. DOI: 10.1007/s11263-013-0620-5. URL: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf (cit. on p. 25).

[32] T. Vu. *Lecture WS 2018/2019 Deep Learning for speech and Language Recognition*. University of Stuttgart (cit. on pp. 18, 19).

[33] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, R. Girshick. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019. URL: https://github.com/facebookresearch/detectron2 (cit. on pp. 41, 56).

[34] S. Zakharov, W. Kehl, S. Ilic. *DeceptionNet: Network-Driven Domain Randomization*. 2019. arXiv: 1904.02750 [cs.CV]. URL: https://arxiv.org/pdf/1904.02750 (cit. on pp. 14, 15).

[35] S. Zhang, L. Wen, X. Bian, Z. Lei, S. Z. Li. *Single-Shot Refinement Neural Network for Object Detection*. 2018. arXiv: 1711.06897 [cs.CV]. URL: https://arxiv.org/pdf/1711.06897 (cit. on p. 12).

[36] Q. Zhao, T. Sheng, Y. Wang, Z. Tang, Y. Chen, L. Cai, H. Ling. *M2Det: A Single-Shot Object Detector based on Multi-Level Feature Pyramid Network*. 2019. arXiv: 1811.04533 [cs.CV]. URL: https://arxiv.org/pdf/1811.04533 (cit. on p. 12).

All links were last followed on December 17, 2020.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 17.12.2020 [illegible]

Ort, Datum, Unterschrift