TANSZÉKVEZETŐ

DIPLOMATERVEZÉSI FELADAT

Pásztor Gábor Dávid
szigorló villamosmérnök hallgató részére

Robusztus objektumdetektálás vizsgálata szimulált ipari környezetben

A gépi tanulás új módszerei az intelligens érzékelés számos területét forradalmasították az elmúlt évtizedben. Ezen módszerek közül külön figyelmet érdemel a mély tanulás (Deep Learning), amely a gépi tanulás legtöbb területén state of the art megoldásnak számít. A mély tanulás egyik legfontosabb alkalmazása a robotikában történik, ahol az összes fő feladat (érzékelés, irányítás, stratégia) megoldására alkalmazható.

Az utóbbi néhány évben a mély tanulás kutatásának egyik fontos fókusza az önfelügyelt tanulási módszerek fejlesztése, ahol az ágens az érzékelési feladatokat címkézett tanító példák helyett a környezetének megértése segítségével sajátítja el. Tipikus öntanulási módszerek közé tartozik a képekről kitakart részletek behelyettesítése, vagy a robot által megtett akciók következményeinek megbecslése.

A diplomatervezés során a hallgató feladata egy olyan algoritmus késztése, amely képes szimulált környezetben különböző objektumokat, és azoknak releváns vizuális tulajdonságait felismerni, valamint az akciónak ezekre való hatását megbecsülni.

A hallgató feladatának a következőkre kell kiterjednie:

- Tanulmányozza át a téma releváns szakirodalmát. Vizsgálja meg, hogy más műhelyek milyen megoldásokat alkalmaznak.
- Készítsen rendszertervet egy megoldásra, amely képes az objektumok és tulajdonságaik felismerésére és predikálására.
- Végezze el a szimulációs környezet kialakítását.
- Végezze az algoritmus fejlesztését és tanítását.
- Tesztelje a megoldás pontosságát és hatékonyságát, valamint végezze el a tanuló algoritmus validációját.

Tanszéki konzulens: Szemenyei Márton

Budapest, 2020.09.14

Dr. Kiss Bálint
egyetemi docens
tanszékvezető

(Tanszéki levélpapír hivatalos lábrésze)

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

H-1117 Budapest, Magyar Tudósok krt. 2. IE444.
Tel: +36 1 4632057, Fax: +36 1 4634112
http://www.iit.bme.hu • e-mail: bkiss@iit.bme.hu

Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Control Engineering and Information Technology

Robust object detection in simulated industrial environment

Master's Thesis

Author

Gábor Dávid Pásztor

Advisor

Márton Szemenyei

May 21, 2021

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Related Work **3**

3 Computer vision **5**

3.1 Neural Networks 6

3.1.1 Neural Network architectures 6

3.1.2 Loss functions 8

3.1.2.1 Cross Entropy Loss 8

3.1.2.2 Dice Loss 8

3.1.3 Optimization with gradient method 9

3.1.4 Backpropagation 9

3.1.5 Difficulties 10

3.2 Deep Neural Networks 10

3.2.1 Convolutional Neural Networks 11

3.2.1.1 Pooling 11

3.2.1.2 Activation 12

3.2.2 Training Convolutional Neural Networks . . . 13
3.2.2.1 Convergence Problems . . . 13
3.2.2.2 Weight Initialization and Data Normalization . . . 13
3.2.2.3 Data Split . . . 14
3.2.2.4 Dropout . . . 14
3.2.2.5 Batch Normalization . . . 15
3.2.2.6 Hyperparameter optimisation . . . 15
3.2.2.7 Data augmentation . . . 16
3.2.3 Transfer learning . . . 16
3.3 Semantic Segmentation and Object Detection . . . 17
3.3.1 Object Detection . . . 17
3.3.1.1 R-CNN . . . 18
3.3.1.2 Fast R-CNN . . . 18
3.3.1.3 Faster R-CNN . . . 19
3.3.1.4 YOLO . . . 19
3.3.2 Semantic Segmentation . . . 21
3.3.2.1 Sliding Window . . . 21
3.3.2.2 Fully Connected Network . . . 22
3.3.3 Instance Segmentation . . . 23
3.3.3.1 Mask R-CNN . . . 23

4 Simulated Environment **24**

4.1 Requirements . . . 24
4.2 Platform . . . 25
4.3 Unity Development Framework . . . 26
4.3.1 Unity User Interface . . . 27

4.3.2 Scene . . . 28
4.3.3 Game Object . . . 28
4.4 Simulation Setting . . . 29
4.4.1 Scenario design . . . 29
4.4.2 Random objects . . . 29
4.5 Lighting and Post-Processing . . . 31
4.5.1 Lighting . . . 32
4.5.2 Post-processing . . . 34
4.5.2.1 Color Grading . . . 34
4.5.2.2 Bloom . . . 35
4.5.2.3 Chromatic Aberration . . . 35
4.5.2.4 Grain . . . 35
4.5.2.5 Screen space reflection . . . 35
4.6 Data generation . . . 36
4.6.1 Perception Package . . . 37
4.6.1.1 Bounding Boxes . . . 38
4.6.1.2 Occlusion Segmentation . . . 39
4.6.2 High Definition Render Pipeline . . . 40
4.6.2.1 Postprocessing metadata . . . 41

5 Algorithm development **44**

5.1 The data used for training . . . 44
5.2 Architecture . . . 45
5.2.1 You Only Look Once . . . 45
5.3 Modification . . . 46
5.3.1 U-Net . . . 46

5.3.1.1 U-Net Architecture 46
5.3.2 Joint network with multiple outputs 47
5.3.2.1 Single-stage Detector 48
5.3.2.2 Expanded detector 49
5.4 Output 50

6 Test Results **51**

6.1 Evaluating the output 51
6.1.1 Performance measurement 51
6.1.1.1 Loss Functions 51
6.1.2 Confusion matrix 52
6.1.2.1 Precision 53
6.1.2.2 Recall 53
6.1.2.3 F_1-score 53
6.1.2.4 Mean Average Precision 54
6.1.3 The mask output 54
6.1.4 Comparing the two approaches 55

7 Future development opportunities **61**

8 Summary **62**

Bibliography **63**

HALLGATÓI NYILATKOZAT

Alulírott *Pásztor Gábor Dávid*, szigorló hallgató kijelentem, hogy ezt a diplomaterv meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. május 21.

Pásztor Gábor Dávid
hallgató

Kivonat

A mély tanulás kutatásának egyik fókusza az önfelügylet tanulási módszerek fejlesztése, ahol az ágens az érzékelési feladatokat a címkézett tanító példák helyett a környezezetének megértésével sajátítja el. Az öntanuló módszerek tipikus használata a képekről kitakart részletek behelyettesítése, vagy a robot által megtett akciók hatásának megbecsülése. Jelen dolgozat célja egy olyan algoritmus készítése, amely képes szimulált környezetben különböző objektumokat, és azok releváns vizuális tulajdonságait felismerni, valamint a különböző akcióknak ezekre való hatását megbecsülni. A dolgozat során nemcsak egy ilyen robosztus objektum detektáló algoritmus mutatok be, hanem egy olyan szimulált ipari környezetet is megalkotok, mely képes a tanításhoz szükséges adatok gyors és egyszerű generálására.

Abstract

The main focus of deep learning research is the development of self-supervised learning methods, where the agent completes different detection tasks by understanding his environment, and not by relying on labeled training examples. A typical usage of self-supervised learning methods is to substitute obscured details from images or to estimate the impact of actions taken by robots. The main goal of the following thesis is to create an algorithm that is able to recognize different objects and their relevant visual properties in a simulated environment and to estimate the effect of different actions on them. In this thesis, I will not only present such a robust object detection algorithm, but also create a simulated industrial environment capable of generating the data needed for training quickly and easily.

Chapter 1

Introduction

In recent decades Deep Learning based solutions revolutionized the field of intelligent perception and machine vision. Moreover, the application of these technologies had a significant impact on robotics, where they provide outstanding solutions for the problem of perception, control, and strategy.

One of the widely used and emerging fields of deep learning is self-supervised learning. A typical task in this field is to estimate and substitute the unseen and uncovered part of the image or predict the effect of an action. Both the substitution and action prediction tasks are applicable in an industrial environment, where there is a camera on the end-effector. The main goal of this thesis project is to develop an algorithm that provides a solution for robust object detection and covered part estimation in an industrial environment.

The first responsibility of this work is to provide a system design capable of simulating, predicting, and detecting different objects and their features. For that, a simulated industrial environment is built capable of creating generic objects and scenarios with a large variety of object features, image distortions, and aberrations. In addition, the environment should be capable of saving datasets with images and metadata for deep learning algorithms.

After the industrial simulation is ready, a deep learning-based algorithm is developed and trained with the generated dataset. The main goal of this method is to detect and classify objects on images and also predict their covered parts. These two

different information combined are efficient to predict the robotic arm's movements to make the covered objects visible. The deep learning algorithm is also tested, validated, and evaluated for its efficiency.

In the first part of this work, an introduction is given to the related works and solutions available in both the industrial and academic fields. After this, the relevant topics and ideas of deep learning that are necessary to overcome the challenges of this task are introduced. The main chapters of the thesis are about the simulation environment with the dataset generation. For that, I introduce the selected simulation engine and how I modified and used it to generate not only the factory model but also the data generation pipeline. In the last part, I describe how I created the deep learning algorithm based on existing solutions. The performance and prediction accuracy of the object detection is also tested end evaluated thoroughly.

Chapter 2

Related Work

In recent years, the research of industrial applications of deep learning and artificial intelligence has had considerable momentum. One of the main goals of the fourth industrial revolution is to create intelligent machines for automated manufacturing. However, it is a difficult task to develop and test deep learning algorithms for industrial purposes since the uninterrupted operation of these advanced robots is usually critical for safety and economic reasons as well. Because of this, these algorithms are developed in a simulated environment, and their initial inputs are images from the virtual environments.

Object detection is widely used to search defects, distortions during smart monitoring tasks [5]. Although building a robust object detection model could be a difficult but necessary task as industrial images are often blurry and noisy [15].

Another effect of the industrial applications is that the object class could be unique. Therefore it is hard to find pre-trained networks with the required object classes in the available datasets. When this occurs, one needs an enormous dataset that results in long training times. To solve this problem, transfer learning is applied, which applies initialized weights of a pre-trained model and only a small number of annotated application-specific training samples [7].

Deep Reinforcement learning is also used for industrial purposes; for example, OpenAI created a proof of concept solution to solve Rubik's Cube with a robotic arm.

After training the algorithm in a simulated environment, the achievements were tested in a real-life scenario [14].

Representational learning techniques provide methods for object detection to eliminate the need for human-annotated datasets to learn visual representations of the data. Self-supervision tasks for deep learning could use auxiliary information for tasks like predicting camera motion between the two images taken in the same environment or predicting the movement of the robotic arm between two states of the environment. Moreover, there is another subsection of self-supervision applications that works with raw pixels to complete different tasks as color recovering [12] or filling patches on images.

Chapter 3

Computer vision

In the following chapter I will give a short introduction to the field of modern computer vision systems, with special attantion to the state of the art deep learning methods used for object detection and semantic segmentation. I learned about the presented topics during my studies, primarily from the relevant chapters of the notes of the Computer Vision Systems course: [21][24][25][22][23].

The main goal of computer vision is to extract high-level information from the images to give human meaning to the scenes. Although, we have to tackle some difficulty if we aim to interpret images using computers. First of all, the same object could have different representations in a 3-channel RGB image in varying lighting conditions. For example, a white surface can be seen as light blue in low light. Similar problems could be caused by scaling, rotating, perspective distortion, as these operations also cause significant changes in the same object's numerical representation. On the other hand, objects could deform significantly, which could cause severe difficulties in object detection. The reader is encouraged to imagine how many different poses the same cat can have on an image. In addition to deformation, actual tasks should manage scenarios where objects are partially or fully occluded [21].

However, we want to recognize not a single specific object but a semantic class in the classification and detection tasks. A class can contain several different objects, between which (depending on the type) there can be significant differences. Moreover, in the real world, classes, where individual specimens show no visual similarity

at all, are common. Here, the class assignment could be determined on the basis of some physical or functional similarity (think, for example, of differently designed cars, e.g., sedan, pickup, Cabrio, SUV). This problem is called intra-class variation and is one of the most significant difficulties in classifying semantics [21].

There is also another difficulty: the so-called semantic gap, which complicates the situation further. It describes the irreconcilable difference between the human interpretation and the digital representation of the image, making it quasi-impossible to formulate a complex computer vision problem into a simple algorithm [21].

3.1 Neural Networks

To overcome the problems mentioned above and extract high-level semantic information from the images, modern computer vision systems use a particular group of artificial intelligence methods called learning algorithms.

These procedures provide a general, parameterizable model for solving problems. During the learning process, a training dataset is used to define these parameters in such a way that the initial, general model will be specialized in solving the given problem. The huge advantage of these algorithms is that they can be used to solve problems where an exact, closed-form algorithmic solution is not available, provided we can produce a database suitable to train the algorithm with[24].

3.1.1 Neural Network architectures

Neurons in neural networks are the mathematical functions, inspired by the functioning of biological neurons. A single neuron's input is a certain number of input signals x_i. The neuron performs a dot product on x_i with its weights w_i, adds the bias, and applies the ϕ activation function on the output. The mathematical model of a single neuron can be seen in Figure 3.4

Neural Networks can be seen as a group of connected neurons, organized in different layers, where connections can usually be found between different layers only. A layer

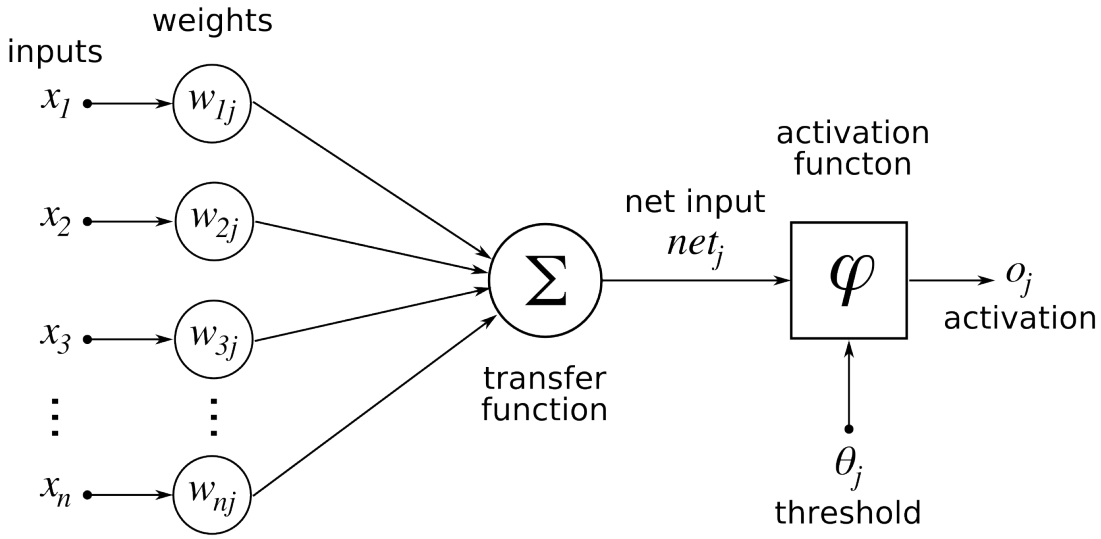


Figure 3.1: Mathematical model of a single neuron[1].

where every neuron is connected with all of the preceding layer's neurons is called a fully connected or linear layer.

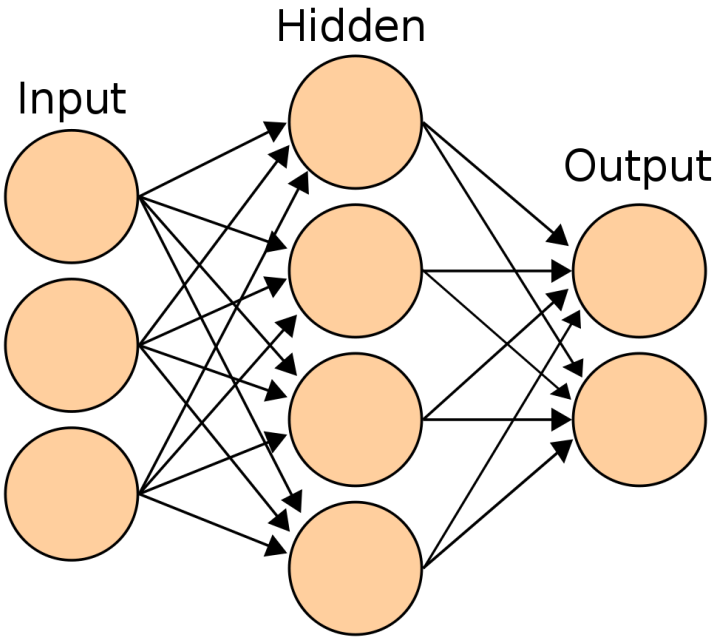


Figure 3.2: Simple neural network[2]

[1]Wikipedia. URL https://upload.wikimedia.org/wikipedia/commons/6/60/ArtificialNeuronModel_english.png

[2]Wikipedia. URL https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Artificial_neural_network.svg

3.1.2 Loss functions

Neural networks are mainly used for classification tasks, where the main goal is to minimize the error of misclassified data, which could be described with a loss function. For regression problems, Mean Square Error is a widely used function to measure the algorithm's accuracy. However, there are better solutions for classification problems to measure the difference between the expected and the actual output. In this section, I introduce two loss functions that are used for this project.

3.1.2.1 Cross Entropy Loss

A widely used loss function is Binary Cross Entropy Loss. In information theory, cross entropy measures the average number of bits needed to identify an event drawn from the set between two probability distributions p and q over the same underlying set, where p is the original distribution, and q is the estimated one [24].

$$H(p, q) = -\sum_{i} p_i \log q_i \tag{3.1}$$

3.1.2.2 Dice Loss

As we will in Section 4.6.1.2, in the segmentation task, it is critical to predict the correct boundaries of the objects. But Cross Entropy Loss computes the average of the discretely calculated per-pixel loss without knowing anything about the global environment of the pixel. This could cause problems as objects usually have edges different edges, which could be predicted as false boundaries (e.g., a curtain or leaves).

This could be solved with the usage of Dice Loss originated from Sørensen-Dice (or also known as F1-score see 6.1.2.3) coefficient, which measures the similarity between two sets. In our case, the two sets are the predicted pixels (p_i) and the ground truth pixels q_i

$$D = \frac{2\sum_i^N p_i q_i}{\sum_i^N p_i^2 + \sum_i^N q_i^2} \tag{3.2}$$

3.1.3 Optimization with gradient method

To minimize the loss function defined in the last section, gradient-based iterative optimization methods are used since both the loss function and the model equations are differentiable.

In practice, we split the training set into randomly chosen mini-batches. The randomness reduces the possibility of getting stuck for the algorithm, and the mini-batches allow parallelization of execution. Although the gradient calculated on mini-batch are not equal to that of the whole dataset, it gives a sufficient approximation for that. This method called Stochastic Gradient Descent (SGD), and it updates the weights with the following formula, where α is a hyperparameter to influence the learning rate :

$$W_{k+1} = W_k - \alpha \frac{\partial \|E\|^2}{\partial W} \tag{3.3}$$

To prevent the stochastic gradient descent from stuck in local minima, momentum is added to the gradient. The most popular optimization algorithm is called Adam, which combines momentum and gradient-scaling techniques[24].

3.1.4 Backpropagation

Backpropagation is a widely used method in feedforward neural networks to compute the gradients of the loss function for each weight by the chain rule. Since every node of the computational graph implements differentiable functions in the neural network, we can calculate the output of each node based on the input. But in the same manner, if we know the derivative of the loss function and the function represented by the graph node, we can calculate the derivative of the node's weight [24].

3.1.5 Difficulties

Of course, neural networks also have different limitations. The most common problems we need to be aware of are the under- and overfitting. It could be the case where the algorithm is not complex enough to solve the given task. In this case, we find that the error in the training set is quite large, but when the algorithm is tested on new data, a similarly significant difference can be observed. This phenomenon is called *underfitting*.

If we increase the complexity, both training and validation errors decrease. Nonetheless, after a while, the validation error will start to increase, yet the training error will remain on the same track: this is the phenomenon of *overfitting*.[24].

The reason for overfitting is that the data used for training is imperfect, finite, and loaded with noise. Thus the primary goal of the algorithms is to learn how to generalize the knowledge extracted from the training set. However, if the training and validation set images are noisy, the training error will remain even with perfect generalization. To overcome this, the algorithm will reduce the abstraction and start to memorize every single answer in the training set, which will turn it similar to associative memory, implying less efficiency for predictions for data not present in the training set[24].

3.2 Deep Neural Networks

Deep Neural Network is based on the architecture described in the last section, just with several hidden layers. Although linear layers could cause several issues as they have plenty of parameters, which can easily lead to overfitting, and it is computationally expensive. Moreover, it does not exploit the spatial features of the image.

3.2.1 Convolutional Neural Networks

The most commonly used artificial neural networks for image analysis are Convolutional Neural Networks, which can solve the problems of the linear layer mentioned in the last paragraph. The Convolutional Layer consists of N convolutional filters slid across the input image (usually has 1-4 channels) and produces an N-channel filtered image. The following convolutional layers will be executed on this N-channel filtered image. The filter area is called the receptive field (usually 3×3 or 5×5), and the depth of the layer (N) are hyperparameters in most cases. Usually, to maintain the size of the original input image after convolution, the pictures are zero-padded [25].

Two essential parameters should be mentioned, called stride and dilation. If stride equals 1, then the filter is executed at every position; if it is 2, then only at every second position, effectively downscaling the output activation by a factor of 2. An l-factor dilatation means that the kernel only samples the signal at every l-th entry. This will increase the receptive field of the layers by drawing apart from the filter. For example, a kernel with a dilation factor of 2 means that the filter is "drawn-apart" by one pixel, as it can be seen in Figure 3.3 [1].

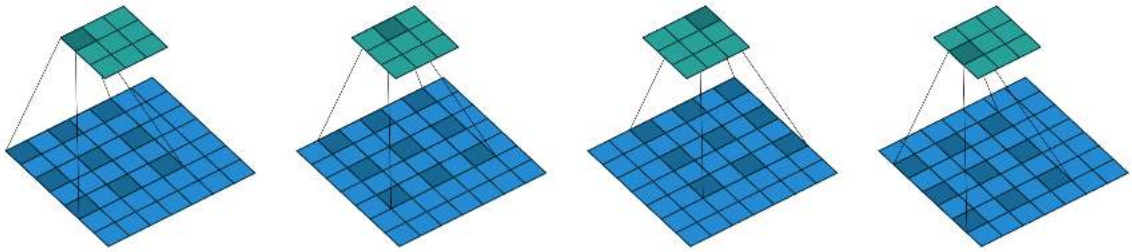


Figure 3.3: Convolving a 3x3 kernel over a 7x7 input with dilation factor of 2. [8]

3.2.1.1 Pooling

If we apply several convolutional layers stacked on top of one another, the size of the activation array at the output will be too large. To decrease the size of the activation array, we can use a stride more immense than one or pooling. Pooling is a sliding-window operation, which substitutes overlaying part of the window with a

single number. This number could be the average or the maximum of the values. [25]

3.2.1.2 Activation

The last essential block of neural networks is the activation layer, which typically follows every convolutional and fully connected layer. As these layers perform linear operations, their composition also remains linear. To overcome this limitation, we embed nonlinear functions between them, which are executed element-wise. In traditional neural networks, the sigmoid and hyperbolic tangent functions were typical choices, but they are only extremely slow-changing in all but a fraction of their input range; thus, their derivative is zero almost everywhere. If we use several of these activation layers, then most derivatives will be zeroed out eventually because of the chain rule. This means that the weights will be unaffected and the training unsuccessful. [25]

The most popular activation function for Convolutional Neural Networks is ReLU (Rectified Linear Unit), which simply thresholds the activation at zero.

$$f(x) = max(0, x) \tag{3.4}$$

ReLU is easy to implement and can greatly accelerate the convergence of stochastic gradient descent compared to hyperbolic tangent and sigmoid functions. Unfortunately, ReLU layers can also be stuck, so the network's weights will not change. To solve this problem, Leaky ReLU is used most often. Instead of the function being zero for the negative range, Leaky ReLU will have a small negative slope (0.01 or so). The slope in the negative region can also be made into a parameter of the activation, as seen in the PReLU (Parametric ReLU activation function). Another variant is the so-called ELU (Elastic Linear Unit), which is completely smooth and differentiable.

3.2.2 Training Convolutional Neural Networks

In the previous sections, I introduced the basics of deep learning and convolutional neural networks. Although these methods may seem easy to use in theory, in practice, several difficulties arise.

3.2.2.1 Convergence Problems

During the training of neural networks, the finite resolution of floating-point numbers causes several problems related to convergence. Generally speaking, working with floating-point arithmetic is stable if the distribution of the numbers used is 0-centered and its standard deviation is roughly 1. The main reason for that is that during backpropagation, several matrix multiplications are carried out. If the numbers are greater than 1, the algorithm will diverge to infinity (exploding gradients) or converge to zero (vanishing gradients). To stay in the domain of nonzero but finite values, the norm of the matrices should be near one [22].

3.2.2.2 Weight Initialization and Data Normalization

It has been described before how a Neural Network is constructed, but the parameters need to be initialized before the training begins. We have no information about the final value of the weights after the training. Still, with proper data normalization, we can assume that approximately half of the weights will be positive and half negative. The natural idea is to initialize the network with zero mean random weights with the correct variance. It is a crucial question how the variance is chosen: too large weights result in too large activations; thus, exploding gradients become more likely, causing the algorithm to diverge. On the other hand, if the weights are small, the network can get stuck in the initial values due to vanishing gradients [22].

To select the correct variance, the Xavier and He initialization schemes are the most widely-used methods, where the initial standard deviation:

$$\mu = 0 \tag{3.5}$$

$$\sigma_{Xavier} = \frac{2}{n_i + n_o} \tag{3.6}$$

$$\sigma_{He} = \frac{2}{n_i} \tag{3.7}$$

Where n_i and n_o are the in- and outputs of the given layer.

Due to similar convergence reasons, it is recommended to have an approximately normal distribution of pixels in the input. If the input values are large, the result will be similar, as in the case of having large weights.

3.2.2.3 Data Split

To avoid overfitting, 2 different datasets are used: one for training and one for validation to monitor overfitting and update hyperparameters. However, it is not enough to use only these 2 sets. However, we cannot state anything about the accuracy of the algorithm on new data. Therefore a third dataset is used: the test set. To ensure the method's reliability, we should separate these three databases and use them only for training, validation, and testing. The dataset split depends on the number of hyperparameters: the more hyperparameters the model has, the more validation data is required. Generally, the data is split as follows: 60% training and 20%-20% for validation and testing, or 70% training and 15%-15% for validation and testing.

3.2.2.4 Dropout

Dropout is an extremely effective and straightforward way to avoid unwanted activations. It is implemented by only keeping a neuron active with some **p** probability or setting it to zero otherwise. This approach significantly decreases overfitting by forcing the network to be redundant and preventing complex co-adaptations on training data. [1]

3.2.2.5 Batch Normalization

Another technique called **Batch Normalization**, developed by Sergey Ioffe and Christian Szegedy helps avoid overfitting by having a regularization effect on mini-batches consist of randomly selected images. During the training we evaluate a minibatch of images in parallel instead of one single image, and the mean and standard deviation of each activation is calculated in each iteration, and they are normalized based on the calculated statistics [1].

The advantages of the Barch Normalization:

- It reduces overfitting because it is computed over minibatches, therefore the model sees some noise each time on the data, which has a regularization effect.
- It improves convergence as the activations of the layers are approximatly in the same order of magnitude.
- The weights of the layers are changing simultaneously during optimization; thus, the distribution of the inputs changes for every layer, forcing them to adjust continuously. Batch Normalization making the optimization more stable and faster by canceling this continuous change.

3.2.2.6 Hyperparameter optimisation

Deep learning algorithms require numerous different parameters that control the learning process. These are the hyperparameters (i.e., mini-batch size, learning rate, etc.), and in contrast to the weights, they are not learned during the training process, but they have a significant effect on the speed and quality of the training. Unfortunately, there is no exact way to determine their values; thus, hyperparameters can only be set by trial and error method.

However, training a complex deep neural network with thousands of training data takes a long time and significant resources, even with high-performance GPUs (Graphics Processing Unit). Therefore it is highly advised to tune these parameters only on a small subset of the data. This is also useful to test the model and look for potential errors, such as memory leaks.

The traditional way to perform hyperparameter optimization is *Grid Search*, which is a simple exhaustive search in a predefined subset of the parameters. The quality of the Grid search method is exceeded by *Random Search*, especially if the optimized parameters are chosen with due diligence. Performing the exhaustive search on random samples in the same space has not only better resolution, but it is way easier to implement.

As Deep Learning Networks can be seen as noisy-black box functions, it is possible to use the well-known *Bayesian Optimization* for hyperparameter optimization. The core idea of this method is to build a probabilistic model of the hyperparameter values and then perform an iterative search on the promising combinations, balancing exploration (parameters with uncertain outcomes) and exploitation (with parameters excepted to be relevant) during the search. In many cases, Bayesian Optimization can outperform Random Search [22].

3.2.2.7 Data augmentation

Supervised Deep Learning models are greatly reliant on the quality, diversity, and amount of the training data. However, creating brand new labeled data points can be an expensive and time-consuming task, especially when we need thousands of them. Therefore data augmentation is a widely used practice to enlarge the size of the dataset by adding randomly modified copies of already existing data. Fortunately, it is relatively easy to use data augmentation for computer vision compared to other deep learning applications. Here, we can randomly crop, rotate, scale, mirror images or use intensity transformations. These operations do not modify the labels of the dataset (or the new labels can be computed automatically), so this method can be used to enlarge the dataset with a minimum amount of effort [1].

3.2.3 Transfer learning

As I mentioned in the previous paragraphs, training data is crucial in deep learning applications. However, it could be a really challenging and time-consuming task to generate thousands or millions of training data, let alone train the convolutional

neural network with them. In practice, it is also relatively rare to have a dataset with a sufficient amount of data, so scientists usually apply Transfer Learning.0

Transfer Learning is the reuse of knowledge of a pre-trained machine learning problem to solve a different task. This method works because the first layers of the network serve as feature extractors. This means that if we already have a pre-trained model, we can use it for a similar task by retraining only the last few layers. Fewer layers mean fewer weights to train, so less data is required.

Another Transfer Learning strategy is the fine-tuning of the earlier layers of the convolutional network while completely replacing the classifier (just like in the previous method) and continuing backpropagation until the earlier layers to tune weights. This is especially useful if we want the network to learn different details of the new data. However, the earlier the stage, the more general the information it extracts, the last layers could contain crucial but specialized feature descriptors that often need to be modified [22].

3.3 Semantic Segmentation and Object Detection

Deep learning has several use-cases in modern technology, from image colorization and style transfer to reconstruction and synthesis. This section will discuss the three main applications relevant to this project: object detection, semantic - and instance segmentation.

3.3.1 Object Detection

Object detection aims to classify all objects into a pre-defined set categories or subcategories and localize them on the image. This task is executed by defining the bounding box of the object on the image. As a regression task, neural networks can easily determine the corners of a single object's bounding box. To estimate the position of the bounding boxes, the same convolutional neural network that performed the classification can be used. We only need to add four outputs to the

network: the coordinates of the bounding box's center (x,y) and the size of the bounding box (height, width), and we also need correctly labeled input data.

3.3.1.1 R-CNN

It is important to note that pictures usually contain multiple objects, and our main goal is to detect all of them. But it is easy to see that we do not know how many objects the image has in most cases. Thus, if we try this method for detection, then the length of the output layer would be a variable, not a constant value. Of course, we could select different regions from the image and classify them separately. Still, to do so, a huge number of regions should be selected due to the variance of the object's spatial location and aspect ratio, which is computationally infeasible.

Fortunately, we can solve this problem with a method proposed by Ross Girshick et al. called R-CNN: Region with CNN features. The key concept of Region Convolutional Neural Net is to select ca. 2000 candidate region proposal with a selective search method. First, the algorithm generates many candidate regions with an initial sub-segmentation. It uses a greedy algorithm to recursively combine similar regions into larger ones that are going to be the final candidates for region proposals. These extracted regions are the input pictures for the CNN which acts as a feature extractor. The output feature vector of the CNN will be fed into a Support Vector Machine (SVM), which classifies the proposed region of the image and also predicts four offset values to adjust the region for increased precision [10][23].

3.3.1.2 Fast R-CNN

R-CNN can sufficiently propose regions potentially containing objects; nevertheless, it still takes a large amount of time to train the network with a couple of thousands of proposed regions per image. This makes it also impossible to use it in real-time applications such as self-driving cars, where the required reaction time is well below 200 milliseconds. The authors of R-CNN bypassed this issue to make object detection faster in an algorithm called Fast R-CNN. Fast R-CNN feeds the whole image to the convolutional network instead of multiple region proposals. Then,

the region proposals are identified from the activation map, and they are fed to an RoI (Region of Interest) Pooling and into a fully connected layer that predicts the class and the bounding boxes. With this solution, Fast R-CNN could be up to 20 times faster than R-CNN. Although if we check testing performance, Fast R-CNN is significantly slower if region proposal is included [9][23].

3.3.1.3 Faster R-CNN

As we have seen, the bottleneck of the Fast R-CNN method is the region proposal, which takes more than 90% of the runtime. This is because the proposed regions are chosen with selective search, which is not only slow, time- and resource-consuming but also maladaptive. To improve the way potential object regions are defined, Shaoqing Ren at al. created a method that lets the network learn the region proposals.

Similar to Fast R-CNN, Faster R-CNN feeds the whole image into the convolutional neural network. Still, instead of using extensive search on the feature map, the algorithm predicts the region with a separate network, called Region Proposal Network. The further parts of the network (RoI Pooling and Linear Layer) remain the same. With these modifications, the Faster R-CNN is fast enough to be used in real-time applications [18][23].

3.3.1.4 YOLO

You Only Look Once (YOLO) is a state-of-the-art, real-time object detection system, which uses a single neural network. YOLO operates with a Fully Convolutional Neural Network that has only convolutional layers with shortcuts and upsampling layers. for the downsampling the feature map 2 strided convolution is used instead of pooling, which helps preserve low-level information about the features.

The feature map learned by the FCN is passed to the classifier, which generates the predictions for the coordinates of bounding boxes and class labels with a 1x1 convolution. For regression, YOLO divides the input image into an $N \times N$ grid, and it predicts B bounding boxes and confidence scores for each cell of the grid. The

system predicts the center coordinates, the width, and height of the bounding box and marks the prediction with a confidence value, representing how likely the box contains an object and how accurate the box is. It means that the algorithm will generate $(Bx(5+C))$ outputs for each cell in the feature map where $5+C$ represents the dimensions and center coordinates of the bounding boxes with objectness score and C class scores.

With this method, each cell of the activation map predicts an object through one of its bounding boxes if the center of the object falls in the receptive field of that neuron. It is important that YOLO could find an object more than once, and in that case, it only keeps the prediction with the highest confidence value: this step is called non-maximum suppression.[17]

An important part of state-of-the-art object detection systems are anchor boxes. Predicting the width and height of the bounding boxes directly might sound a great idea, but it could lead to unstable gradients due to the large variance of these values. To solve this issue, object detectors usually predict log-space transforms or simply offsets pre-defined bounding boxes called anchors.

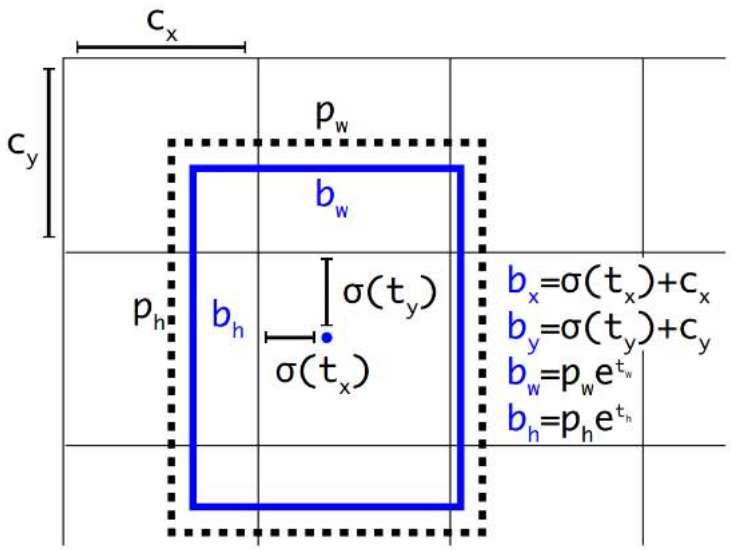


Figure 3.4: Bounding box prediction with coordinates and dimension in yolov3 [16]

In summary, YOLO predicts the coordinates of the bounding box's center relative to the grid cell, and the dimensions, which are also normalized by the dimensions

of the image, are predicted by applying a log-space transform and then multiplying it with the anchor.

YOLO has many different versions: version 2 introduced the usage of anchor boxes, and version 3 improved its performance by detecting objects with 3 different scaling factors. [16] YOLOv4 uses two categories of methods to improve the object detector's accuracy.

- **Bag of freebies(BoF):** These methods can make the object detector to receive better accuracy without increasing the inference cost. These methods (such as data augmentation, normalization and different regularization techniques) only change the training strategy or only increase the training cost.
- **Bag of specials(BoS):** These are different plugin modules (e.g., Spatial Attention Module, Spatial Pyramid Pooling) or post-processing methods that only increase the inference cost by a small amount but can significantly improve the accuracy of object detection. [6]

In contrast with region-based solutions, YOLO applies a single convolutional network to predict both the class probabilities and the locations of the objects, which makes YOLO considerably faster. The biggest advantage of YOLO over R-CNN is its speed, which makes it fast enough for real-time applications, although it also has some limitations, as it struggles to detect small objects.

3.3.2 Semantic Segmentation

Another well-known task for Deep Learning Neural Networks in the field of Computer Vision is Semantic Segmentation, which can be seen as pixel-level classification, where every pixel of the image is assigned to a class label [23].

3.3.2.1 Sliding Window

Of course, the semantic segmentation could be carried out naively by using a neural network trained for classification. All we need to do is to slide the architecture over

the image creating prediction for each pixel instead of classifying the whole image. But it is easy to see that this is not the most computationally efficient way to solve the problem with a sliding window: first of all, it takes a long time; secondly, it does not use the shared features from the overlapping sections; therefore features have to be re-computed over and over.

3.3.2.2 Fully Connected Network

We could make the classification parallel by stacking multiple Convolutional Layers on top of each other with padding to preserve the original resolution of the image. The number of the output channels will be the same as the number of classes, so each channel represents different class predictions about the pixels. This solution performs better than a simple sliding window, but preserving the same resolution through the process is quite expensive and requires significant computational power.

We could operate with fewer layers to reduce the parameters, but it could impact the performance dramatically, unlike in the case of object detection. It happens because, in semantic segmentation, we classify the image on pixel level; therefore, important details of the image may be lost due to downsampling. However, in object detection applications, the network predicts the presence of a single object in a given region based on a set of high-level features. Hence, periodical downsampling with pooling causes no significant harm.

A popular solution to bypass this drawback is to use Downsampling and Upsampling in Fully Connected Network. The main idea is that we want to get our output in the same resolution as the input image, but we want to execute the algorithm mostly in lower resolutions. FCNs contains skip connections between early pooling layers and the upsampling layers with the same resolution. These shortcuts have two main advantages:

1. Integrate detailed lower-level features usually lost with downsampling, which contain boundary information for label predictions.

2. Improves convergence by avoiding information degradation during the training process.

Information degradation is a practical problem in deep neural networks, which is caused by vanishing gradients. In theory, deep neural networks act as universal function approximators that could learn any simple or complex problem if we stack enough layers on top of each other. Nevertheless, due to practical problems such as vanishing gradients and the curse of dimensionality, deep neural networks can fail to learn simple functions. With the increasing number of layers, the training accuracy will saturate and eventually degrade rapidly because of this problem. An obvious solution to solve the information degradation problem is to skip the layers with residual connections and skip connections or shortcuts. Residual blocks can improve the accuracy of the predictions in deep neural networks by mitigating the problem of degradation and improving the effective Field of View [23].

3.3.3 Instance Segmentation

The third important application of deep learning in the field of computer vision is Instance Segmentation. Instance segmentation combines elements of object detection and semantic segmentation by obtaining the precise segmentation and correct detection of the individual objects together.

3.3.3.1 Mask R-CNN

Understanding a Deep Learning architecture that solves Instance Segmentation is an easy task based on the knowledge we gained in the previous sections. Mask R-CNN extends Faster R-CNN architecture with improved backbone convolutional network and parallel pathway to generate object masks after the Region Proposal Network.

To overcome the problem of degradation, detailed in Section 3.3.2.2, Mask R-CNN uses Feature Pyramid Network (FPN) as the backbone, consisting of a bottom-up pathway, a top-bottom pathway, and lateral connections. In the first stage, the Region Proposal Network scans the top-bottom pathway of the FPN and proposes regions with the help of the well-known anchors. At the second stage, another neural network assigns the region proposals to specific areas of the activation map and generates class, bounding-box, and mask predictions [11].

Chapter 4

Simulated Environment

As mentioned, the first stage of this work is to create a simulated industrial environment that can be used to generate images and metadata for the deep learning algorithm. The simulation should also be modifiable and capable of generating randomized scenarios sufficient to create diverse train datasets.

4.1 Requirements

The main requirements of the simulated environment are the following:

- Create a 3D model of an industrial area
- The environment should contain different scenarios, e.g.:
  - assembly workshop
  - storage
  - conveyor belt
- Include generic objects with random features (e.g., quantity, orientation, position, color)
- Simulate image errors, such as distortion and visibility reduction with post-processing effects.

- Reduce the visibility of the objects (e.g., glare) with realistic lightning and different visual effects.
- Enable to use scripts, especially for saving the dataset, and for setting up simulation parameters (e.g., number of objects in the Scene)
- Capture video and image with metadata automatically during the simulation.

4.2 Platform

First, a development framework capable of creating an environment is selected, which satisfies the requirements described above. After the initial research, only four potential platforms were left that seemed sufficient for this task: MoJuCo, NVIDIA Isaac Sim, Unreal Engine, and Unity.

MuJoCO stands for Multi-Joint dynamics with Contact, and it is developed by Emanuel Todorov. MuJoCO is a physics engine aiming to facilitate research and development in robotics and other areas where fast and accurate simulation is needed. This is the first full-featured simulator designed from the ground up for model-based optimization. It is widely used in academic research for creating a 3D environment both for robotic simulations and deep learning algorithms. The main advantage of this framework is the extensive documentation, realistic physics, and support for robot simulations. However, the personal license for non-commercial use is costly.

NVIDIA Isaac Sim is a simulation platform that provides photo-realism with real-time ray and path tracing through RTX. This platform aims to provide realistic simulations for industrial and robotic applications. There is satisfactory documentation provided by NVIDIA for the toolkit; however, a high-end RTX GPU is required to run this platform.

Game engines such as **Unity** or **Unreal Engine** are also widely used for robotic simulations and machine learning environments. An excellent example for this is *CARLA*, an open-source simulator developed with Unreal Engine to support the development, training, and validation of autonomous driving systems. On the other hand, Unity provides out-of-the-box solutions and libraries to support machine learn-

ing applications. One notable project is *Unity Machine Learning Agents*, which is an open-source Unity module that enables games and simulations to serve as environments for reinforcement learning agents. Unity ML-Agents has more than 15 pre-built example environments to train reinforcement learning agents with, and users can also create new ones. Unity ML-Agents also supports deep reinforcement learning algorithms written in Python.

Furthermore, Unity Computer Vision provides a framework called *Perception Package* for generating large-scale datasets for computer vision training and validation, which provides many valuable tools for this project, e.g., depth image generation, instance and semantic segmentation masks, and ground truth information for object detection and classification tasks.

Unity was chosen to create the simulation environment for this project because of its friendly User Interface and practical documentation. Unity is widely used for 3D modeling, creating simulated realities, and developing games. The engine has been used by industries outside video gaming, such as architecture, construction, film, automotive, and engineering. It provides a vast amount of learning materials for inexperienced game developers. It also has a built-in asset store; however, it is also possible to import different assets created by a third party. Another significant advantage is that Unity provides a free-to-use license for students.

4.3 Unity Development Framework

In the previous chapter, I introduced the different options the create a simulated industrial environment that supports dataset generation for computer vision and deep learning tasks. In this chapter, a quick introduction is given about the selected environment: Unity.

In 2005, Unity technologies announced its Mac OS X-exclusive game engine at Apple's WWDC (World Wide Developer Conference). Although it was only available on Mac at first, Unity supports more than 25 different platforms by now. As a result, developers can create games on Windows, Linux, or OS X, and they can port their game to a number of different platforms, including consoles and mobile.

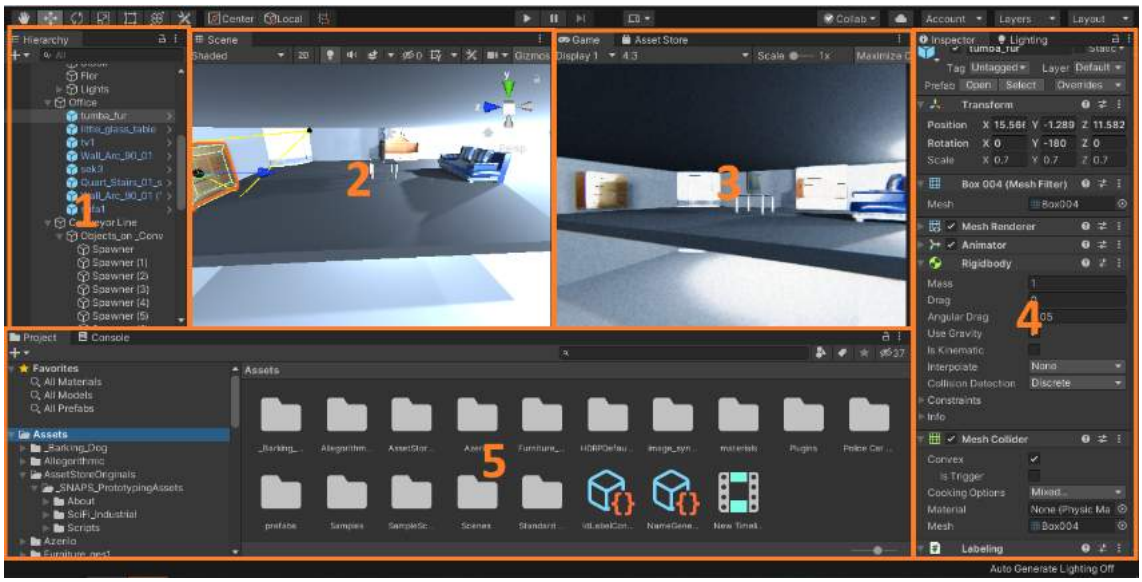


Figure 4.1: Unity User Inferface

4.3.1 Unity User Interface

The user interface is fully customizable, every panel and window can be hidden or resized. The most common windows in their default position (Figure 4.8) are as follows [28]:

1. **Hierarchy Window:** This is a hierarchical representation of the Scene's Game Objects. The Scene is inherently linked to this view, and the selected objects are marked in both windows. The hierarchy reveals the structure of how the GameObjects are attached to each other.

2. **Scene View:** Displays a preview of the designed Scene. It visualizes the Scene from a 3D or 2D perspective, depending on the type of the project. It is also possible to visually navigate and edit the Scene and the Game Objects in it.

3. **Game View:** Simulates how the final game will look like through the Main Camera. This view is only used for visualization, so roaming and editing are not enabled. The simulation can be started by clicking on the Play button.

4. **Inspector Window:** This window is used to edit and view the currently selected GameObject's properties. The layout of this window changes based on the different properties of the Game Objects.

5. **Project Window:** Displays the library of Assets and Packages that are available to use in the Project. Every project resource (e.g., sound, animation, model, script) is accessible from this window.

4.3.2 Scene

The previous chapter suggests that one of the main components of the project is the Scenes. In Unity, Scenes are assets that contain a part of a game or application. When a new project is created and opened for the first time, it opens with a sample scene that contains only a Light Source and a Camera [30].

4.3.3 Game Object

In addition to the camera and light source, other game elements can be created, which are referenced in Unity as GameObjects. GameObject is an essential concept in the Unity Editor: it represents anything that exists in the Scene from special effects and characters to cameras and lights[27]. Although they cannot accomplish anything on their own, different properties need to be assigned to them in the form of Components that implement various functions. The most common components are:

- **Transform:** This is used to store the position, rotation, scale, and parenting state of the GameObject; therefore, it is the most crucial Component, which is always attached and cannot be deleted.
- **Mesh Renderer, Mesh Filter:** The Mesh Filter takes a mesh (which is the main graphics primitive) from the assets and passes it to the MeshRenderer for rendering on the screen at the position defined by the GameObject's Transform component.
- **Collider:** This component defines the shape of the GameObject for the purpose of physical collision, but it does not necessarily have the same shape as the mesh of the object. With the appropriate simplification of the mesh,

the difference is indistinguishable in the simulation, but it can save significant computational resources. Frequently used Colliders are the Box Collider, Sphere Collider, Capsule Collider, Mesh Collider.

- **Rigidbody:** This component enables GameObjects to move in a realistic way according to the laws of physics by receiving torque and forces. It is necessary for the object to be influenced by gravity and other scripted forces or interact with other objects through the NVIDIA PhysX engine.

4.4 Simulation Setting

4.4.1 Scenario design

Once the requirements were defined and the simulation engine selected, the simulation scenario needed to be designed. As mentioned before, the simulation should contain different scenarios; therefore, the simulated industrial building will have three different parts:

1. A conveyor belt with different small objects on it like barrels or tools.
2. A workshop for larger items, such as cars and industrial platforms.

and a floor that functions as an office. Different robotic arms stand at the two sides of the conveyor belt and at the workshop. The design plan of the building is shown in Figure 4.8.

4.4.2 Random objects

In the initial setting, there are 67 different objects. First, the 3D models were downloaded from the Asset Store or other third-party providers [2][31]. Then, they are imported into the project and saved with the required features as *Prefabs*. Unity's Prefabs are stored GameObjects with all its components, properties, values, and child GameObjects as a reusable asset. They allow developers to configure and save GameObjects with different properties and reuse them as a template in the game.

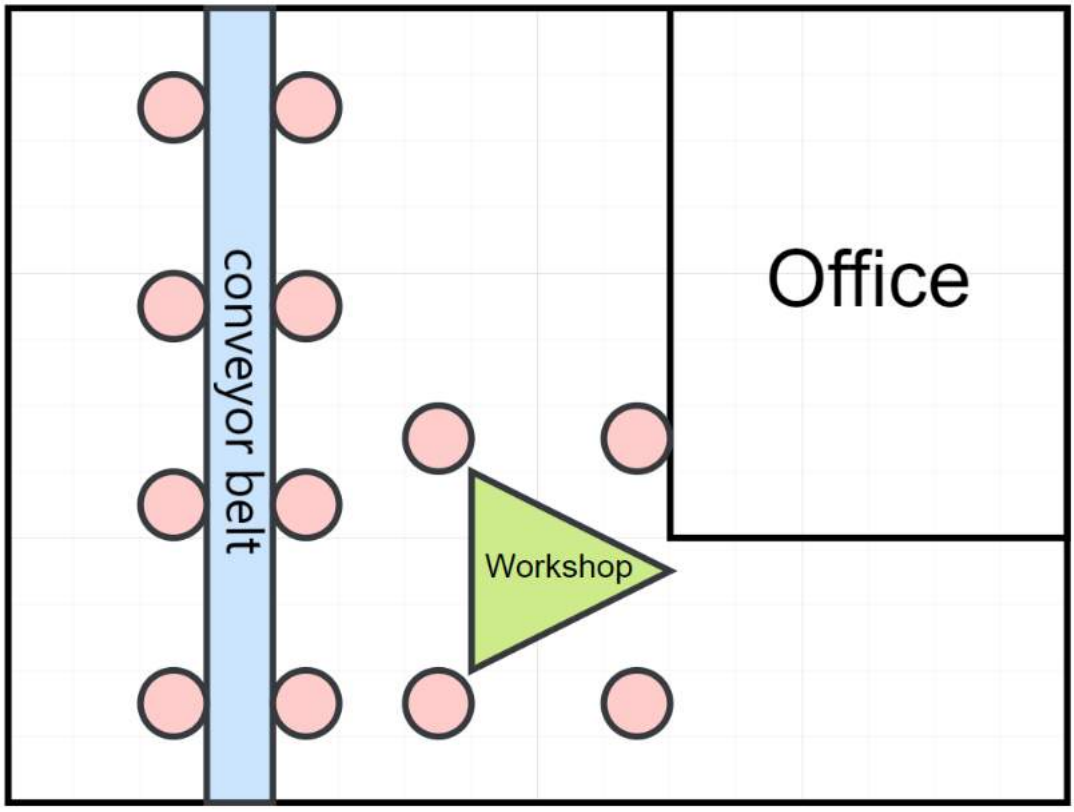


Figure 4.2: Design plan of the virtual environment.

In addition, the Prefab system automatically keeps all copies in sync, which makes it easier to change the details of the object as opposed to using traditional copies.

A vital requirement was that the objects should be generated with random features. To generate random objects, it is crucial to save slightly different copies from them, in such way that all copy can be modified at once. For this use case, Unity Prefabs provides an excellent solution. I created many different prefabs (e.g., barrels with different sizes and colors) and saved them in the project. These prefabs are spawned on the map with random position and orientation with the C# script. At the workshop, there are three generated elements chosen randomly from a list of big-size prefabs. The objects are created through an empty game object that has a script attached to it. The C# script chooses an object from the list of prefabs and rotates it randomly around the y-axis if this function is enabled. However, this function can be switched off because we do not want every spawned prefab to be randomly rotated. E.g., the robotic arms are also generated from a list of prefabs with the help of the Spawner script, but they need to face in the direction of the conveyor belt.

The small objects on the conveyor belt are spawned from 36 different Spawner objects above the production line. Therefore, the spawned objects fall and land on the assembly line with different positions and orientations while also occluding or hiding each other. This allows the self-supervised deep learning algorithm to estimate and substitute the obscured details on the images. Of course, there will be some objects that fall to the ground, but there is an option to hide or delete them later.

4.5 Lighting and Post-Processing

In the initial setting, the light came from 11 Point Light sources placed on the ceiling. Later on, this could be replaced with light-emitting surfaces, which result in more realistic lighting.

Figure 4.3: Office.

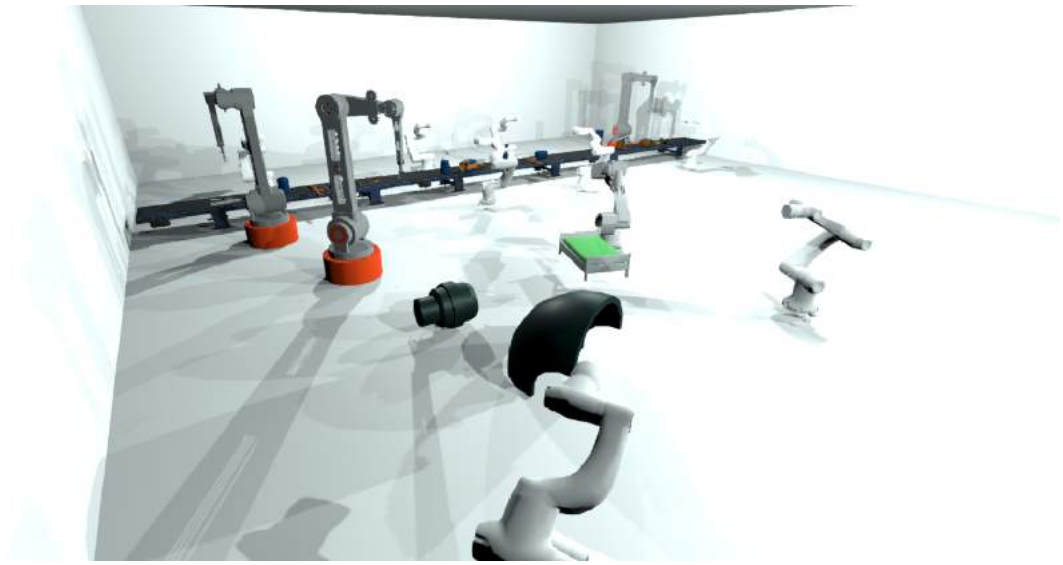


Figure 4.4: Workshop.

4.5.1 Lighting

The Lighting in Unity works by modeling how realistic light behaves in the environment. Therefore, the detail of the lighting depends on how realistic or simplified the model is [29].

Unity has three lighting modes:

1. **Real-time lighting:**

Figure 4.5: Conveyor belt.

As the name suggests, Unity calculates and updates the real-time lighting for every frame at runtime without any pre-calculations. This lighting model could be helpful if there are moving elements in the Scene, as their shadows should be changed in real-time, although it has some limitations. First of all, it could be costly, especially in complex scenes or on low-end hardware. Moreover, as this mode is computation-intensive to use, special lighting effects, i.e., color bounce is disabled, which might cause unrealistic lighting on the Scene.

2. **Baked Lighting:**

   In contrast with Real-time light, Baked Light Mode calculates the lighting of the Scene and saves the lightmap to load during the simulation: this called baking. This operation reduces shading and rendering cost at the price that Baked Light mode does not contribute to illuminating the dynamic Game Objects.

3. **Mixed Lighting:**

   Mixed lights combine the properties of real-time and baked lighting. The exact behavior depends on the specific lighting mode, which are:

   (a) Baked Indirect: Real-time direct combined with baked indirect lighting.

   (b) Shadowmask: Real-time direct combined with baked indirect lighting. It is similar to baked indirect but enables baked shadows, which will au-

tomatically be blended with real-time shadows. Whereas Baked indirect Lighting mode only uses real-time shadows.

(c) Subtractive: Baked direct and indirect lighting. This is the least realistic lighting mode, but it is efficient for stylized art, and low-end hardware [29].

My choice for this project was to use Shadowmask lighting mode, as the main goal is to create a realistic simulation as possible, and since the simulation is only used to generate images, so the lower framerate due to complex calculations and hardware limitations is allowed.

4.5.2 Post-processing

The project's principal focus is to create an algorithm capable of detecting objects and estimating the occluded parts in an industrial environment, where the visibility may below. To reduce the visibility in this virtual factory, different post-processing effects were used. The effects were made with the Unity Post-processing Stack, which is built-in into the Unity engine. To use the effects, a Post-processing Layer should be added to the Camera Object, and another Post-process Volume component should be set up with the applied effects. The used effects are introduced in the following paragraphs.

4.5.2.1 Color Grading

Unity operated with a balanced color scheme by default, which gives a pleasant look, especially when neutral white-light is used. However, the color spectrum of industrial image sensors is usually smaller compared to consumer cameras. To recreate this property, the Color Grading effect is used with reduced Saturation and Contrast. In addition, ACES (Academy Color Encoding System) is used for tone-mapping, which is widely used in the film industry.

4.5.2.2 Bloom

There are two ways to occlude an object from the camera to create challenging estimation tasks for the algorithm. One approach is when other objects are used, as mentioned before in Section 4.4.2. The second way is to create glare on the objects. These glares also can be eliminated from the items by moving the camera to a different position. This glare is created with the bloom effect. To make it work, the effect intensity is set to 50 with a two threshold and 0.678 Soft Knee. With this setup, the items with Materials with a 0.86 Smoothness value will have a tiny glare on them which is enough to trick the algorithms.

4.5.2.3 Chromatic Aberration

The object detection algorithms look for different features on the images, such as edges, corners, or other patterns. Therefore it is advised to make the object boundaries harder to detect. For example, to blur the edges, a small amount of Chromatic Aberration is used in a small portion with a 0.24 value to avoid making the task too difficult.

4.5.2.4 Grain

Difficulties introduced by low resolution are not so common as the modern images usually have at least 720p resolution. However, there could be issues with object detection in specific use-cases with low light conditions, which cause noise in the sensor. To mimic this noise, a Grain effect could be used with varied intensity and grain size.

4.5.2.5 Screen space reflection

Screen Space Reflection is a post-processing effect that helps create more realistic lighting in the Scene without additional computation. This provides a way more efficient solution than calculating the reflections in real-time via ray-tracing.

Figure 4.6: Workshop after post-process.

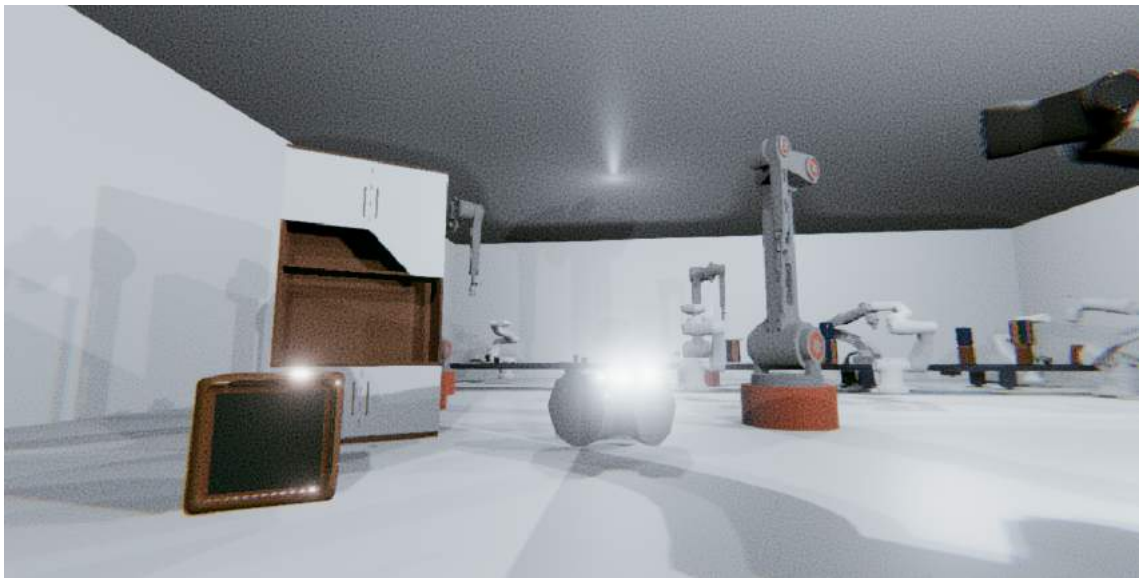


Figure 4.7: Workshop with objects after post-process.

4.6 Data generation

Once the environment is designed and created in a way that allows it to meet all the criteria described in Section 4.1, the data set can be generated.

To achieve this, a video is captured through the Main Camera module with the help of C# scripts. Screenshots from the simulation are saved periodically after every few frames in *PNG* format with the marked 2D bounding boxes around the objects. Moreover, metadata is also saved, containing the ground truth information for the Deep Learning algorithm, including bounding box position, class label, and 3D

Figure 4.8: Conveyor belt with objects after post-process.

information about the objects' position in the simulation space. For object detection, the state-of-the-art YOLOv5 Object Detector is used with some modifications to predict the occluded areas as well, which will be discussed in more depth later in Section 5.

4.6.1 Perception Package

Fortunately, Unity provides an official open-source toolkit - named Perception Package - for generating large-scale datasets for computer vision training and validation. The Perception package is modifiable and has a detailed tutorial and documentation that helps understand how the different modules work together, making it easier to modify for individual tasks. By default, the Perception Package saves data, for instance, segmentation, semantic segmentation, object detection, and classification. The most recent version is also capable of saving depth images from the Scene.

The depth image, the 3D and 2D Bounding Box data, and custom-built overlap mask information are used for this work.

4.6.1.1 Bounding Boxes

The most critical data is the ground truth information about the bounding boxes of objects seen in the Scene. The 2D bounding boxes can be generated with the following algorithm:

The 3D unity objects are rendered with a MeshRenderer, which has its bounds in the space. These bounds can be accessed through an AABB (*Axis-Aligned Bounding Box*), which is a box aligned with coordinate axes and fully enclosing the object. Because the box is never rotated with respect to the axes, it can be defined by just its center and extents. The only task left is to enclose this 3D box into a 2D rectangle which can be done quickly.

However, this method returns the 2D bounding box of the AABB, so this could be used for the deep learning algorithm. However, if higher precision is required, the bounding box could be calculated directly based on the vertices of the object's mesh in the following way:

1. Loop through all the vertices of the mesh:
2. Transform the point from local to world coordinates
3. Transform the point from word to screen coordinates
4. Search for minimal and maximal X and Y coordinates.

To create the bounding boxes around the objects, the Perception Package uses a similar method described above, but it uses the advantages of the pre-built Scriptable Render Pipelines (see 4.6.2); thus, it generates the information about the objects in real-time.

The Perception package saves the ground truth data in JSON format with the following pieces of information:

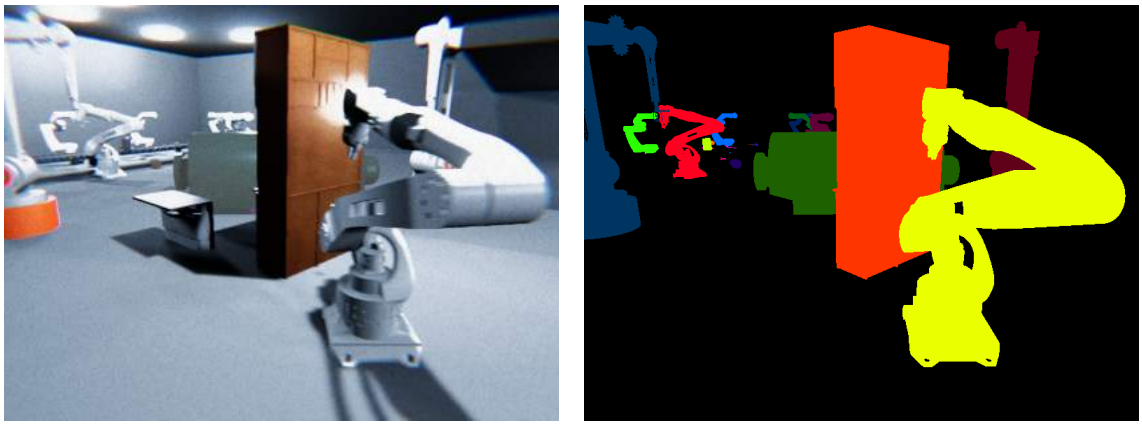
- about the 2D Bounding Boxes: instance id, label id, label name, place on the image
- about 3D Bounding Boxes: 3D position, size, rotation, velocity, and acceleration

The deep learning algorithm will use this information completed with occlusion information.

4.6.1.2 Occlusion Segmentation

For the segmentation task, the most important information is the ground truth image. Of course, the ideal way to get mask images would be to create a segmentation mask for the covered parts of the objects. Nevertheless, that is a rather difficult task because of the way these images are generated.

To create the segmentation mask in Unity, the textures of the objects are changed temporarily based on the category of the labeled object. The objects are now textured with a single color representing their category, and after that, a single screenshot is saved from the objects in front of the black background. The output of the instance segmentation laberer can be seen in Figure 4.9 with the original Scene to compare with.



(a) Original **(b)** Segmentation mask

Figure 4.9: Original camera output and the instance segmentation mask generated by Perception module.

To save the covered parts in a segmentation mask, the instance segmentation module should be modified in a way that it should save multiple screenshots from each frame with objects separated based on their place relative to the camera, and after this, images should be combined with a post-processing module.

This procedure is easy to explain but difficult to implement, and the time required for implementation exceeds the time available. Yet, in order to show that our

concept works, we do not need the exact covered parts. It is much easier to use the information about the covered 2D bounding boxes. With a simple modification, additional data for bounding boxes have been generated that shows which objects' bounding boxes are covered by another one. A post-processing module can create a segmentation mask with the use of this additional metadata. This information provided about the hidden parts is sufficient to test our method.

4.6.2 High Definition Render Pipeline

In Unity, Render pipelines are designed to perform a series of operations that take the contents of a Scene and display them on the screen. The operations performed by the render pipelines at a high level are culling, rendering, and post-processing.

Different render pipelines have varying performance and capabilities, and content creators can choose between them based on their project's needs. However, switching a project from one render pipeline to another can be challenging because different pipelines have different shader outputs and might not have the same features. By default, Unity offers three pre-built render pipelines with different characteristics, but users can create their own custom render pipelines too.

Unity offers the following pre-built render pipelines:

- *Built-in Render Pipeline*: This is Unity's default render pipeline for general purpose usage, with limited options for customization.
- *Universal Render Pipeline (URP)*: A Scriptable render pipeline that is easy and quick to customize and allows users to create optimized graphics across a wide range of platforms from mobile to high-end consoles and PC.
- *High Definition Render Pipeline (HDRP)*: A Scriptable render pipeline that allows developers to create cutting-edge high-fidelity graphics for high-end platforms.

Because these render pipelines often have different features and output shaders, they are not compatible with each other; therefore, developers need to understand

the different pipelines that Unity provides. They need to make the right decision between the options early in development.

The pictures were shown of the project thus far were made with the Built-in Render Pipeline. Unfortunately, the default pipeline has very limited customization options; thus, it is not compatible with the Perception Package. Unfortunately, I was not aware of the limitations of the Unity Computer Vision modules nor the differences of different render pipelines, so changing the Render pipeline on-the-fly in the development phase proved to be a challenging task. The most challenging part was to make existing Materials compatible with the new shaders. *Shaders* are assets that contain instructions and code for the graphics card.

After careful consideration, High Definition Rendering Pipeline seemed to be the best decision to migrate the project into, because it had better graphics, and most of the Materials were compatible with the basic HDRP Lit Shader, which enables the creation of realistic materials with subsurface scattering, iridescence, and vertex or pixel displacements. Another aspect of my decision was that in Universal Render Pipeline, real-time shadows are not supported with the point or directional light sources. However, in this project, most of the objects are deployed randomly in real-time, and inconsistent shadows could make the Scene really disturbing and less realistic.

After changing the render pipeline from Built-in to High Definition, the Scene looked similar but with more realistic colors and lighting. The only defect was that in the post-processing layer, specific effect values are capped or restricted. Therefore the glare effects are not as vivid as before.

4.6.2.1 Postprocessing metadata

Although Unity with the Perception module is capable of saving most of the required data to train the deep learning neural network, formatting the input data before the training is crucial.

First of all, the properly formatted metadata needs to be extracted from the captured json files. In this step, the ground truth input of the 2D bounding boxes for YOLO

(a) High Definition Render Pipeline **(b)** Built-in Render Pipeline

Figure 4.10: Conveyor belt with different render pipelines after post-processing.

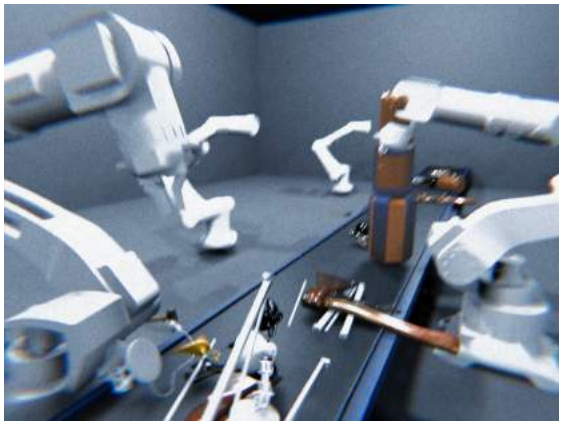
are created. YOLO traditionally expects the annotation file to be in the following format in a *.txt* named the same as the image it belongs to:

```
<object class> <x> <y> <width> <height>
```

- <object class> is the index of the object class (from 0 to #classes-1)
- <x> and <y> are the center of the rectangles relative to the to the width and height of the image (0.0 to 1.0]
- <widht> and <height> are the width and height of the bounding box relative to the width and height of the image (0.0 to 1.0]

Once the annotation file is created, the next task is to create the proper input images. The input consists of the screenshots of the Scene, the depth images, and the segmentation mask. The screenshots are combined with the right depth images based on the frame number to a four-channel RGBD image that will be used as input for the object detection and classification module, while the segmentation mask is generated using the overlap information provided by the customized Bounding Box Labeler from the Perception Package.

The segmentation mask for the covered bounding boxes and the combined images is shown in Figure 4.11



(a) RGB image

(b) Depth image

(c) Combined RGBD image

(d) Segmentation mask

Figure 4.11: Input for the neural network: Segmentation mask from covered object parts and the RGBD image

Chapter 5

Algorithm development

In the last chapter, I introduced how I created the simulated industrial environment with the Unity game engine and modified the Perception Package to generate the dataset required for the training. Then, in Chapter 5, I present the deep learning algorithms I used, and discuss how I modified them to solve the problem and to improve their performance.

5.1 The data used for training

After I had the pipeline to generate the dataset with the algorithms I described in Section 4.6, I created numerous different images of the scene. My goal was to make them as diverse as possible, moving the camera around the main parts of the simulated industrial camera. The camera took different traits around the generated items, ensuring all types of objects are included in the shots. To increase the complexity of the images, I added extra objects to the scenes sometimes. As a result, more than 6000 pictures and metadata were generated. The dataset is split into three different sets: train, test, and validation, where I aimed to keep the recommended data split ratio mentioned in Section 3.2.2.3. Table 5.1 shows the data split I used.

Dataset	Number of data	Percentage
Train Set	4806	69.74%
Test Set	982	14.25%
Validation Set	1103	16.01%

Table 5.1: Dataset split

5.2 Architecture

YOLO is one of the most popular object detection algorithms nowadays due to its speed and efficiency. Therefore, I chose YOLO v5 for this project to start with, and I modified this architecture to detect the occluded part of the objects.

5.2.1 You Only Look Once

The short description with its main advantages of the YOLO object detector can be found in Section 3.3.1.4. As I mentioned before, the first three versions of YOLO were published by Joseph Redmon from 2016 to 2018. However, he stopped his computer vision research in 2020 due to the potential military applications and privacy concerns.

Fortunately, other developers continued improving YOLO after Redmon's withdrawal. Alexey Bochovskiy et al. [6] published a new version of the algorithm as a fork from the original repository. YOLO v4 significantly improved the detector's accuracy, with only a modest trade-off in the training cost.

As YOLO v4 released as a fork, it uses the same C++ darknet architecture as the original one. Glen Jocher introduced several PyTorch-based YOLO implementations before, but within two months after the release of YOLO v4, he uploaded another fully PyTorch implementation named YOLO v5. This implementation has exceptional improvements, including auto-learning bounding box anchors and mosaic data augmentation. I chose YOLO version 5 to work with due to the PyTorch implementation, the performance.

5.3 Modification

To make the deep neural network able to produce a segmentation output, I investigated two different options:

1. In the beginning, I added a separate network for segmentation, which works parallel with the existing model.

2. Another approach is to extend the neural network already used by YOLO with a segmentation output.

5.3.1 U-Net

U-Net is a neural network architecture based on Fully Convolutional Networks initially developed for biomedical image segmentation at the University of Freiburg. U-Net requires fewer training data than a simple FCN architecture and yields more precise segmentation. This feature is advantageous in biomedical applications, where the amount of the training data is insufficient. The improvement is reached by replacing the max-pooling layers with upsampling operations, called successive layers; hence these layers increase the output resolution. Another modification is that the successive layers also have a large number of feature channels. This helps to propagate context information to the high-resolution layers and results in a symmetrical u-shaped architecture [20].

5.3.1.1 U-Net Architecture

The U-Net constists of two parts:

1. **Contracting path:**

   It is built like a typical convolutional network: the main building blocks are two repeated $3x3$ convolution, with a ReLU activation function and a $2x2$ max pooling with stride 2 for downsampling.

2. **Expansive path:**

The expansive path contains $2x2$ "up-convolutions" that halves the number of feature channels, a concatenation with the cropped feature map, which is necessary due to the loss of border pixels, and two $3x3$ convolutional layer with ReLU.

The final classifier is a 1x1 convolutional layer that maps each 64 component feature vector to the desired number of classes [20]. The illustrated network can be seen in Figure 5.1.

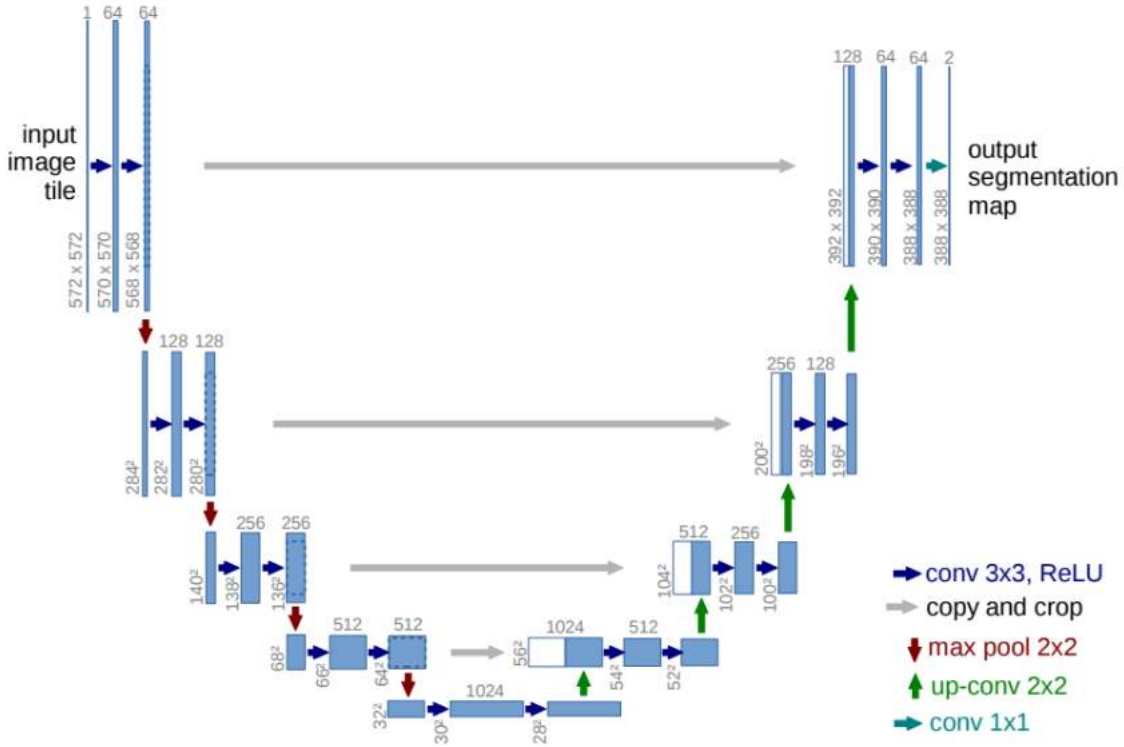


Figure 5.1: Illustrated U-Net architecture, where each blue box corresponds to a multi-channel feature map. [20]

5.3.2 Joint network with multiple outputs

My other approach was to modify the neural network YOLO used by adding them upscaling layers. Before describing how I modified the existing architecture, let me have a quick introduction, how the single-stage detector in YOLO made up.

5.3.2.1 Single-stage Detector

As to the head part, object detectors are categorized into two main categories: one-stage and two-stage object detectors. The architecture of these detectors is illustrated in Figure 5.2.

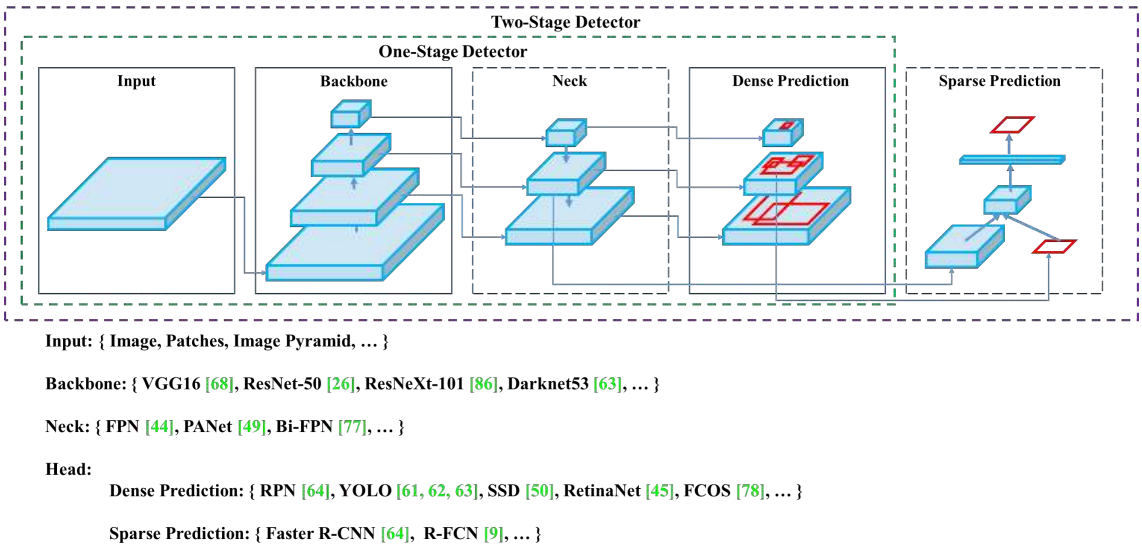


Figure 5.2: One- and two-stage object detector architectures with examples. [6]

The 3 parts of the one-stage detector:

1. Model Backbone
2. Model Neck
3. Model Head

Cross Stage Partial Backbone: A meaningful improvement of YOLO version 4 and 5 is the usage of the Cross Stage Partial Network (CSPNet) backbone, which is capable of reducing the cost of the computations by 20% with equivalent or superior accuracy on the ImageNet dataset by integrating from the beginning and the end of a network stage. Furthermore, this architecture is easy to implement and combine with existing model architectures. It is also used in YOLO v4 and v5 to improve the feature extractor [32].

PANet as Model Neck: The Model Neck is mainly used to generate feature pyramids that help generalize well on object scaling, which is essential for the models to perform better on brand new, unseen data, as objects vary in size and scale on different images[26].

In YOLO v5 Path Aggregation Network (PANet) is used as Feature Pyramid Network (FPN) to generate a multi-scale feature map, with adaptive feature pooling and accurate localization signals in lower layers by bottom-up, which shortens the information path between lower and topmost layers. [13]

Model Head: The Model Head applies the anchor boxes on features and generates the output vectors with bounding box coordinates, objectness scores, and probabilities. YOLO v5 uses the same head as the previous versions and generates the bounding boxes in the same way as discussed in Section 3.3.1.4.

5.3.2.2 Expanded detector

The second approach I used was to expand the neural network used by YOLO with an auxiliary segmentation output. As I discussed before, the single-shot object detector has a highly efficient feature detector layer with the application of CSPNet and PANet.

For the semantic segmentation, I added three successive blocks of the following layers:

1. Upsample layer
2. Shortcut
3. 2x2 convolution with Batch Normalization and Leaky ReLU activation function
4. CSP Bottleneck layer that used to obtain a representation of the input with reduced dimensionality.

5.4 Output

As I suggested before, the algorithm has two different output:

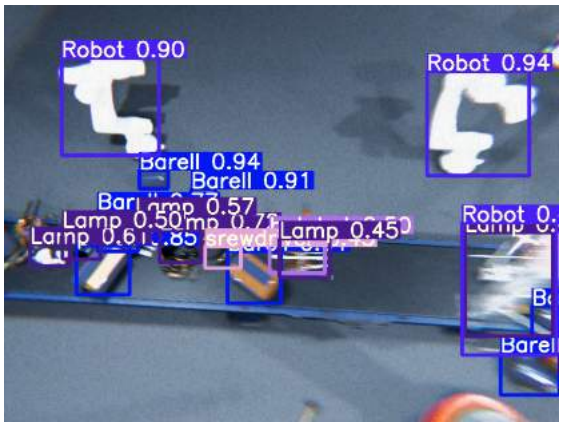
1. Predicted bounding box coordinates and class labels from the YOLO-based object detector.

2. Predicted segmentation mask, which shows the covered regions of the objects.

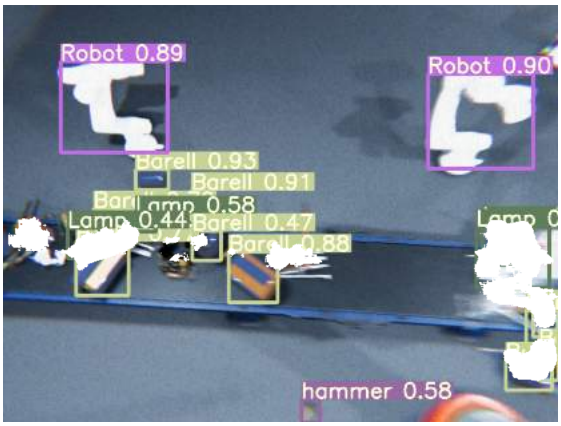
The final output of the cover detection process is a combined image of these two images, where the found objects with bounding boxes and class labels are shown with white cloud-like regions, where covered parts are predicted by the deep neural network. The outputs are shown on Figure 5.3

(a) Original cover mask

(b) Predicted cover mask



(c) Object Detector output



(d) Combined output

Figure 5.3: Outputs of the detector.

Chapter 6

Test Results

This section introduces, analyzes, and tests the algorithm's results with the different types of underlying neural network architecture discussed in the previous section. The images represented here must be from the test set; thus, the algorithm has not seen these images during the training or validation; hence these represent the performance of the algorithm on unseen images.

The majority of the tests were executed with Google Colab, which is an online collaborative environment, that is capable of running Python code with GPU hardware acceleration. It is designed specifically for students, data scientists, and AI researchers.

6.1 Evaluating the output

6.1.1 Performance measurement

6.1.1.1 Loss Functions

Before validating on unseen data, however, it is essential to measure how good the training performance is. As I mentioned in Section 3.1.2, the main goal of the deep learning methods is to minimize the loss function, which defines the average precision of the predictions; as a result, loss functions provide a great metric to track whether the algorithm actually learns. For semantic segmentation, I used the actual Binary

Cross-Entropy loss on both the training and validation sets, as well as the Dice score (3.1.2.2), to get a more complete picture of the accuracy of the segmentation.

Our single-shot object detector has three outputs for each prediction: the bounding boxes, the objectness score, and the class labels. Three different losses are calculated for the three output during the object detection task:

1. **Classification loss** (cls_loss) **:** Measures the correctness of the classification for each bounding box prediction, where each box contains a classified object or just a plain background. Calculated with Cross Entropy Loss.

2. **Generalized Intersection over Union loss** (giou_loss) **:** GIoU is an evaluation metric to characterise the accuracy of the predicted bounding boxes. [19]. Traditionally IoU is calculated as the fraction of the area of overlap and the area of union for the areas of the predicted and ground truth bounding boxes.

3. **Object loss loss** (obj_loss) **:** Loss to measure the predicted objectness score. The loss is usually computed with a Binary Cross Entropy function.

6.1.2 Confusion matrix

In the field of machine learning and classification problems, the *Confusion Matrix* is an excellent way to visualize the performance of an algorithm. In a confusion Matrix, the rows represent the actual class of the instance, whereas the columns represent the predicted class. For binary classification problems a confusion matrix is shown in Figure 6.2.

For object detection tasks, the confusion matrix metrics are calculated based on the IoU threshold:

- True Positive: The ground truth and the predicted bounding boxes overlap with a higher value than the threshold.

- False Positive: The predicted bounding box does not overlap enough with the ground truth or is classified with the wrong class label.

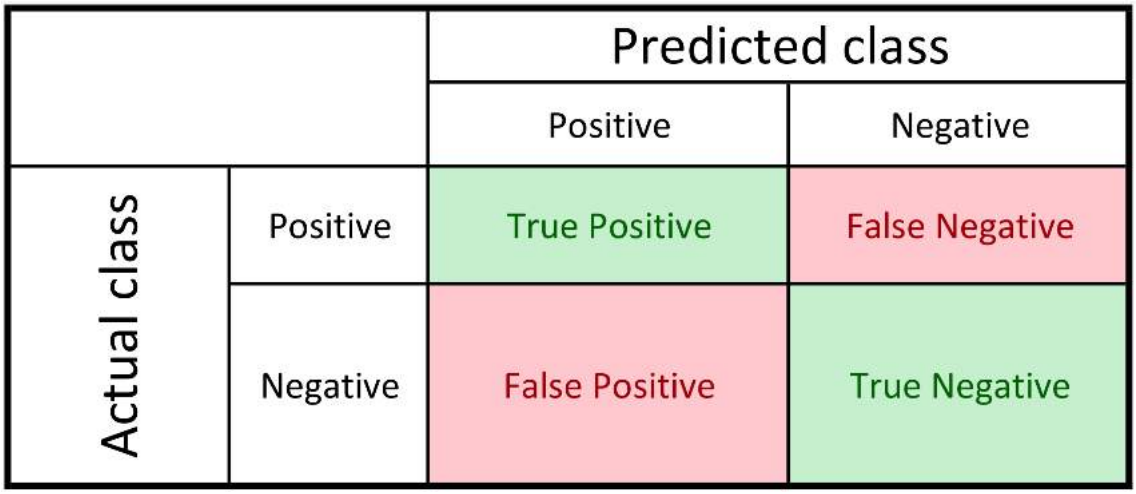


Figure 6.1: Confusion Matrix

- False Negative: No prediction matched with the ground truth object.
- True Negative: The algorithm does not predict the bounding box, which seemed a correct decision.

6.1.2.1 Precision

Precision is the fraction of the relevant, correctly identified positive instances among all positive identifications.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{6.1}$$

6.1.2.2 Recall

Recall represents the percentage of good predictions compared to all predictions interpreted as correct.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{6.2}$$

6.1.2.3 F_1-score

F_1-score (also known as Dice-Sørensen coefficient see. Section 3.1.2.2) is a measure of the test's accuracy, and calculated as the harmonic mean of the recall and precision.

We have seen how the coefficient is expressed in terms of vector operations over binary vectors in Section 3.1.2.2. Equation 6.3 shows the measure when it is applied to Boolean data.

$$F_1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} = \\ = \frac{2 * TruePositive}{2 * TruePositive + FalsePositive + FalseNegative} \tag{6.3}$$

6.1.2.4 Mean Average Precision

The calculated precision and recall values can be visualized on a joint chart for each prediction. This is called the Precision-Recall curve. The curve will start in the upper-left corner, as the algorithm has a high precision (more predictions means more False Positive detections) and high precision (fewer predictions means more False Negative detections). As the training continues, more predictions are made, which will increase the Recall value. If the model works appropriately, the Precision-Recall curve stays high with increasing recall values, which means more correct detection with fewer mistakes. The Average Precision is meant to summarize the Precision-Recall Curve by averaging the precision values across all recall values. In practice, Average Precision is calculated as the area under the interpolated Precision-Recall curve. However, the precision and recall values are calculated for each individual class, and therefore if we take the mean of the Average Precision for each class, we get a valuable metric to compare the quality of different models. [4]

6.1.3 The mask output

Looking carefully at Figure 5.3b, you can notice that the segmentation output does not follow the boundaries of the rectangles on the segmentation mask (Figure 5.3a). At first glance, you might think that the inaccuracy of the prediction reduces the efficiency of the algorithm. However, when you combine the two outputs of the algorithm, you see something interesting. In Figure 5.3d it can be noticed that

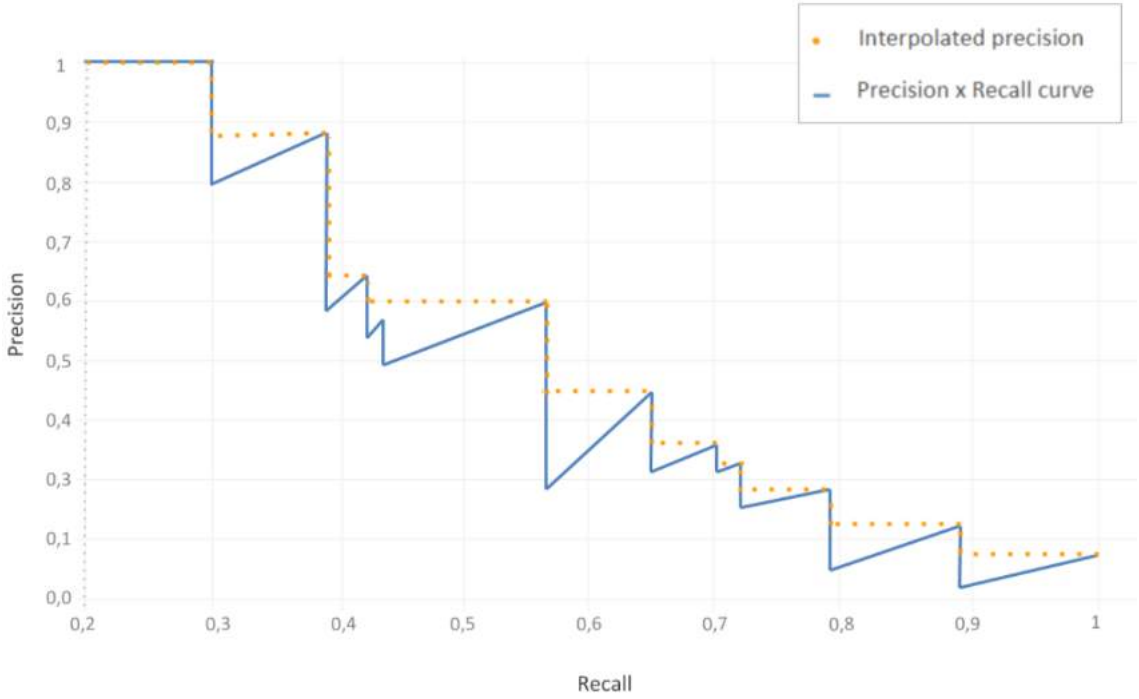


Figure 6.2: Precision-Recall Curve[3]

the overlaps marked by the network in the segmentation output do not follow the overlaps of the bounding boxes but the covered parts of the actual objects. What is the reason for this phenomenon? The underlying neural networks extract different features from the images that can be used for generalized object classification later. However, at the edges of the bounding box overlaps, there is often no suitable image feature to define the bounding box boundaries, only a grey background. This results that the neural network could only classify the parts of the image covered with complete certainty, where an object is actually present.

6.1.4 Comparing the two approaches

To compare the different runs, I used the same hyperparameters each time and executed the training cycle for 30 epochs with a minibatch size of 4. The hyperparameters are documented in Table 6.1.

In Section 5.3 I described the two different neural networks I used for the training. The following paragraph is made to discuss the performance, accuracy, advantages, and disadvantages of the different network architectures. On the next page, Figure

Hyperparameter	Value
initial learning rate (Adam=1e-3)	0.01
SGD momentum	0.937
optimizer weight decay	5e-4
giou loss gain	0.05
cls loss gain	0.58
cls BCELoss positive_weight	1.0
obj loss gain	1.0
obj BCELoss positive_weight	1.0
iou training threshold	0.20
anchor-multiple threshold	4.0
focal loss gamma	0.0
HSV-Hue augmentation	0.014
image HSV-Saturation augmentation	0.68
image HSV-Value augmentation	0.36
image rotation (+/- degree)	0.0
image translation (+/- fraction)	0.0
image scale (+/- gain)	0.5
image shear (+/- degree)	0.0

Table 6.1: Hyperparameters

6.3 shows 4-4 images of the combined output from both networks: with separate UNet and YOLO on the left side and one joint neural network with two separate outputs on the right side.

As it is shown in the figure, the neural network with auxiliary segmentation output (right side of Figure 6.3) produces more accurate predictions. The occlusion segmentation is not so fuzzy, and estimates the boundaries of the hidden objects with higher precision. This could happen because the segmentation loss value is backpropagated through the same subset for the computational graph as the object detection loss, which means another reinforcement or confirmation in the learning process.

During the initial test runs, I noticed that the segmentation loss was not changing during the training session. The segmentation output was also inaccurate, and the predicted territories were too bulky compared to the ground truth, especially with the parallel U-Net. In order to improve the performance of the neural network, I halved the segmentation cost. This modification stabilized the quality of the segmentation for the parallel UNet, but it has not led to any particular progress. The segmentation loss for the validation remained around the same values for the joint network, as you can see in Figure 6.4 To know the impact of the segmentation

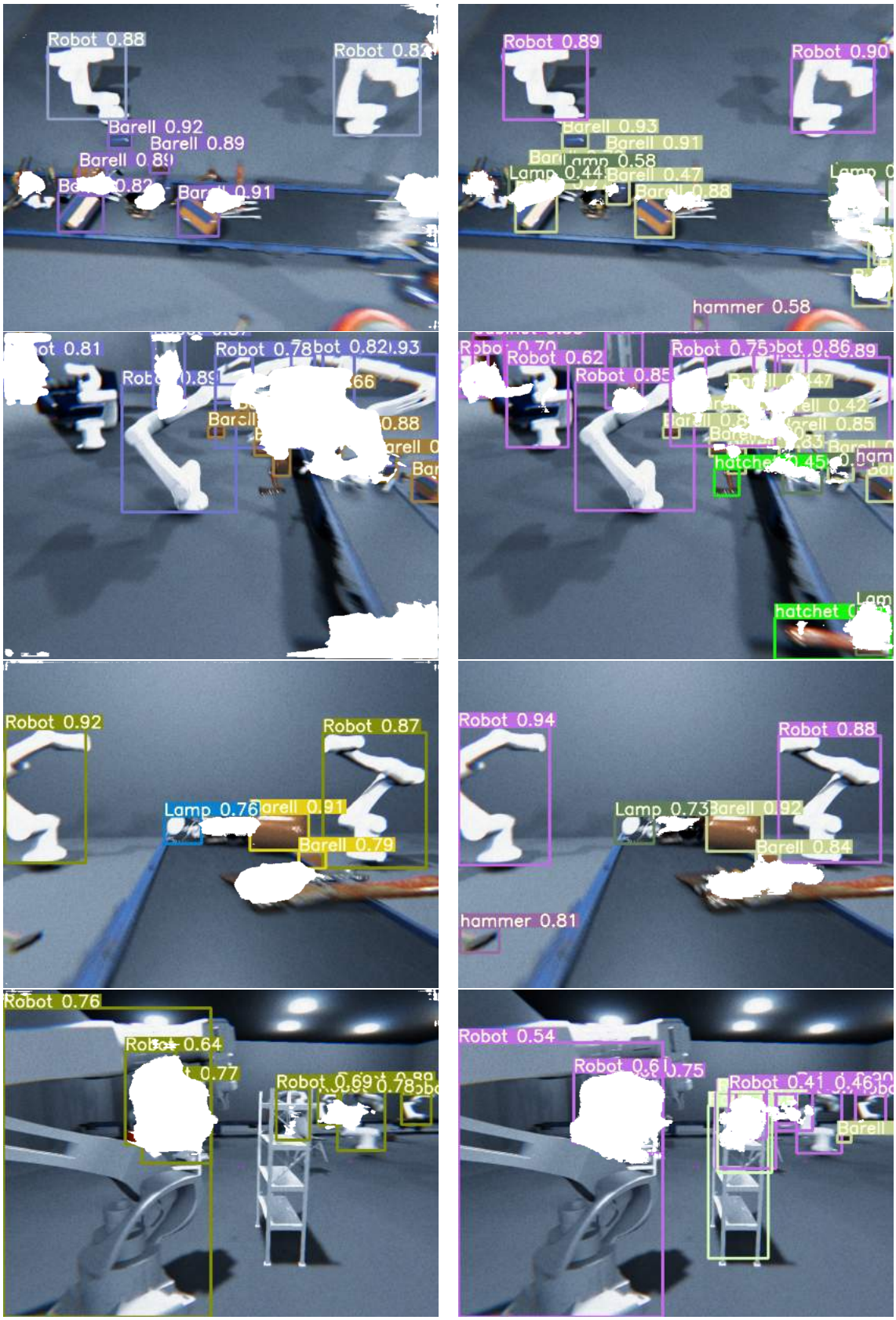
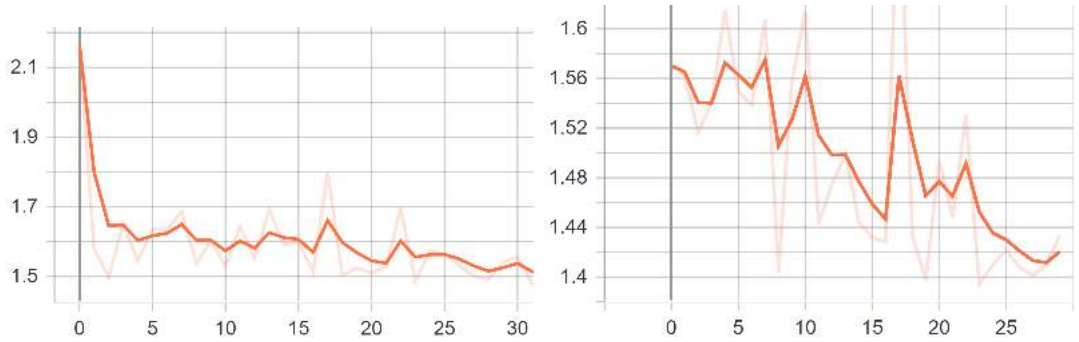
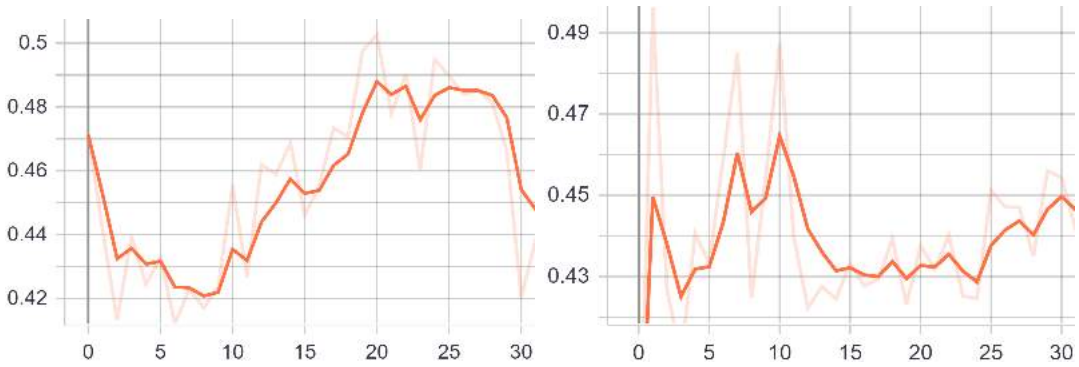


Figure 6.3: Comparing the two architectures output. **Left side:** YOLO with parallel UNet. **Right side:** One network with auxiliary segmentation output.



(a) Separated Network **(b)** Separated Network - 0.5 * Segmentation Loss

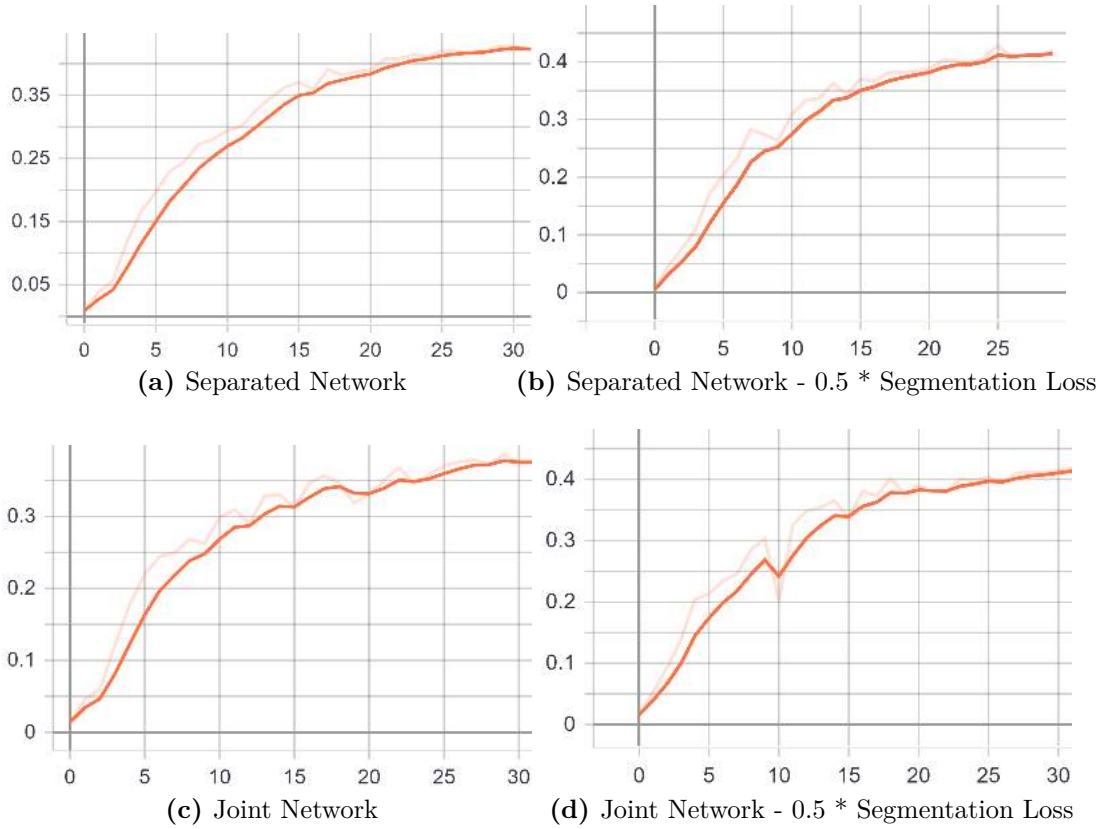


(c) Joint Network **(d)** Joint Network - 0.5 * Segmentation Loss

Figure 6.4: Segmentation Loss during validation

loss modification for object detection, let us take a look at Figure 6.5. Halving the segmentation loss helps reach the same level of precision of the object detection faster while having a stabilizing effect on the segmentation process. When using a shared network, the two types of loss affect the same parts of the network during backpropagation, so changing the value of the segmentation loss has a direct effect on object detection as well.

I also experimented with changing the IoU threshold to see its effect on object detection. Halving the value of the threshold has not changed the Mean Average Precision curve. The Recall and the Precision also stayed similar, which means that the algorithm does not have more predicted bounding boxes with a lower threshold, so the missing objects are likely not recognized at all. If we take a look at the images of Figure 6.3, we can see that the object detection is surprisingly accurate, even though a lot of objects on the conveyor belt are largely occluded. Knowing the effect of the IoU threshold modification, we can conclude that objects at the edge of



(a) Separated Network **(b)** Separated Network - 0.5 * Segmentation Loss

(c) Joint Network **(d)** Joint Network - 0.5 * Segmentation Loss

Figure 6.5: mAP@0.5

the image or largely occluded are undetected due to lack of information and image noise.

The recorded metric values are shown by Table 6.2.

Metric	Separated	Separated - 0.5	Joint	Joint - 0.5
f1	0.22	0.21	0.19	0.21
map@0.5	0.43	0.43	0.37	0.42
precision	0.35	0.33	0.33	0.35
recall	0.46	0.46	0.44	0.46
Dice	0.48	0.56	0.45	0.41
Seg train mean loss	0.26	0.89	0.26	0.93
Seg val loss	1.49	1.4	0.42	0.45
Train cls loss	0.018	0.019	0.017	0.018
Train giou loss	0.049	0.049	0.047	0.049
Train obj loss	0.09	0.09	0.09	0.094
Val cls loss	0.029	0.029	0.03	0.029
Val giou loss	0.057	0.057	0.058	0.057
Val obj loss	0.11	0.11	0.112	0.11

Table 6.2: Metrics for algorithm validation

As we can see, the different networks have similar performance for the object detection task. The joint network's precision and recall values are the same as the separated, even with the additionally backpropagated segmentation loss. The validation losses that describe the object detection are also similar for the joint network. In conclusion, the augmented network part does not significantly affect the one-staged detector's performance.

However, the segmentation loss measured during validation reveals a significant improvement. The joint network predicts the covered segmentation output with similar Dice-Sørensen coefficient as the separated UNet. Nevertheless, the most crucial factor is the segmentation loss value during the validation, which is the third in this case.

As a result, the proposed joint network, which consists of the one-stage detector and an auxiliary segmentation output, could predict not only the object's class and place on the image also its visual details such as its covered parts. The developed method is robust enough to execute the prediction even with low visibility and blurred images. Another advantage is that the segmentation mask used as ground truth does not need to be the same as the actual covered parts. Thus, the algorithm can robustly mark the invisible area based only on the information extracted from the objects bounding rectangles. This means a considerable advantage, as the proposed solution does not require any additionally generated metadata for the dataset. The inaccurate cover masks do not affect the algorithm performance. As a result, the proposed solution is accurate, robust and it is less computationally intensive due to the joint neural network architecture.

Chapter 7

Future development opportunities

The deep learning algorithm is capable to detect both objects and their hidden parts with surprising precision; however, both the simulation and the algorithm's performance can be improved further.

To use this technique in the real industrial environment, the 3D object models and textures need to be more realistic. The time and hardware constraints of this thesis make it impossible to use custom-made textures and effects for the photorealistic visualization, but high-resolution simulations are capable of creating high-quality training datasets that can be used to train real industrial robots. To improve the quality of the ground truth images, it would be ideal to create a segmentation mask that actually follows the covered parts of the objects. This requires drastic modification of both the Unity Render Pipeline and the Perception Package with custom scripts, as I described in Section 4.6.1.2.

For safety-critical applications, such as real-life robots, it is ideal for working with a real-time video stream. This requires the deep learning algorithm to run faster, and for that, the neural network (especially the segmentation part) should be redesigned. I am also curious whether more accurate segmentation can be achieved with the currently used labeling technique by augmenting the cost function.

Chapter 8

Summary

My first task during this thesis work was to create a system design both for the simulated industrial environment and for the deep neural network-based object detector. After that, I selected a suitable framework to create the simulation that satisfies the requirements. This selection took a while, as it was time-consuming to collect and assess the information about the physics engines on the market.

After due diligence, I chose to work with Unity Game Engine. I created the game environment with Unity and its modules. The 3D models I used were collected from free and open-source marketplaces. I also used and modified the Perception module, which was developed by Unity Computer Vision, to generate ground truth labels for deep learning applications.

At the end of the semester, I created an object detection and cover segmentation algorithm based on existing deep learning solutions. I ran several training cycles with different parameter settings and documented every test I had. During the tests, the algorithm was capable of solving the robust object detection tasks even on noisy images, and it also outperformed the initial requirements for the accuracy of its prediction.

Bibliography

[1] Cs231n convolutional neural networks for visual recognition, convolution / pooling layers. URL https://cs231n.github.io.

[2] URL https://free3d.com/.

[3] Evaluating object detection models: Guide to performance metrics, Oct 2019. URL https://manalelaidouni.github.io/Evaluating-Object-Detection-Models-Guide-to-Performance-Metrics.html.

[4] How to calculate the mean average precision (map) - an overview, Apr 2021. URL https://hungsblog.de/en/technology/how-to-calculate-mean-average-precision-map/#identifier_8_3802.

[5] Ioannis D. Apostolopoulos and Mpesiana Tzani. Industrial object, machine part and defect recognition towards fully automated industrial monitoring employing deep learning. the case of multilevel vgg19, 2020.

[6] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

[7] Yann Dauphin, Xavier Glorot, Salah Rifai, Yoshua Bengio, Ian Goodfellow, Erick Lavoie, Xavier Muller, Guillaume Desjardins, David Warde-Farley, Pascal Vincent, Aaron Courville, James Bergstra, and et al. Unsupervised and transfer learning challenge: a deep learning approach, 2011.

[8] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[9] Ross Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015. DOI: 10.1109/ICCV.2015.169.

[10] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524.

[11] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

[12] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Learning representations for automatic colorization, 2017.

[13] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation, 2018.

[14] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019. URL http://arxiv.org/abs/1910.07113.

[15] Steven Puttemans, Timothy Callemein, and Toon Goedemé. Building robust industrial applicable object detection models using transfer learning and single pass deep learning architectures. *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2018. DOI: 10.5220/0006562002090217. URL http://dx.doi.org/10.5220/0006562002090217.

[16] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[17] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[18] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[19] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression, 2019.

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL http://arxiv.org/abs/1505.04597.

[21] Márton Szemenyei. *Computer Vision Systems*, chapter Introduction to Computer Vision. Budapest University of Technology and Economics Faculty of Electrical Engineering and Informatics Department of Control Engineering and Information Technology", . URL http://deeplearning.iit.bme.hu/notesFull.pdf.

[22] Márton Szemenyei. *Computer Vision Systems*, chapter Deep Learning in Practice. Budapest University of Technology and Economics Faculty of Electrical Engineering and Informatics Department of Control Engineering and Information Technology", . URL http://deeplearning.iit.bme.hu/notesFull.pdf.

[23] Márton Szemenyei. *Computer Vision Systems*, chapter Detection and segmentation. Budapest University of Technology and Economics Faculty of Electrical Engineering and Informatics Department of Control Engineering and Information Technology", . URL http://deeplearning.iit.bme.hu/notesFull.pdf.

[24] Márton Szemenyei. *Computer Vision Systems*, chapter Neural Networks. Budapest University of Technology and Economics Faculty of Electrical Engineering and Informatics Department of Control Engineering and Information Technology", . URL http://deeplearning.iit.bme.hu/notesFull.pdf.

[25] Márton Szemenyei. *Computer Vision Systems*, chapter Convolutional Neural Networks. Budapest University of Technology and Economics Faculty of Electrical Engineering and Informatics Department of Control Engineering and Information Technology", . URL http://deeplearning.iit.bme.hu/notesFull.pdf.

[26] Towards AI Team. Yolo v5 - explained and demystified, Jul 2020. URL https://towardsai.net/p/computer-vision/yolo-v5%E2%80%8A-%E2%80%8Aexplained-and-demystified.

[27] Unity Technologies. Gameobjects, . URL https://docs.unity3d.com/560/Documentation/Manual/GameObjects.html.

[28] Unity Technologies. Unity's interface, . URL https://docs.unity3d.com/Manual/UsingTheEditor.html.

[29] Unity Technologies. Introduction to lighting, . URL https://docs.unity3d.com/Manual/LightingInUnity.html.

[30] Unity Technologies. Scenes, . URL https://docs.unity3d.com/Manual/CreatingScenes.html.

[31] Vitalii et al. Uspalenko. The best 3d viewer on the web. URL https://sketchfab.com/.

[32] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of cnn, 2019.