Dynamics Group (M-14)
Schloßmühlendamm 30
21073 Hamburg

Implementation and Tuning of YOLO for brake NVH sounds

Project Work by Lucky Aubrey Adam, (B.Sc.)

Hamburg, August 2022

Prüfer: Prof. Dr. N. Hoffmann
Betreuer: Dr. M. Stender

Eidesstattliche Erklärung

Ich, Lucky Aubrey Adam (Student der Mechatronik an der Technischen Universität Hamburg, Matrikelnummer 21510774), versichere an Eides statt, dass ich die vorliegende Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, August 31, 2022

Contents

List of Figures

List of Tables

List of Code

Nomenclature

One common convention is that matrices and vector are represented with bold letters. Constants and variables are depicted with normal letters. If time-discrete quantities are considered, the expression is defined for a specific time.

List of Symbols

Latin Symbols

variable	unit	meaning
A_{A}	$[-]$	Area of box A
A_{B}	$[-]$	Area of box B
A_{overlap}	$[-]$	Area of overlap
A_{union}	$[-]$	Area of union
$\mathrm{AP}_{0.50}$	$[-]$	Average precision for $\mathrm{IoU}^{\mathrm{thresh}} = 0.50$
$\mathrm{AP}_{0.75}$	$[-]$	Average precision for $\mathrm{IoU}^{\mathrm{thresh}} = 0.75$
$\mathrm{AP}_{0.90}$	$[-]$	Average precision for $\mathrm{IoU}^{\mathrm{thresh}} = 0.90$
AP_{c}	$[-]$	Average precision for class c
B_g	$[-]$	Number of anchors per grid
B_{A}	$[-]$	Box A
$B_{\mathrm{anchor},i}$	$[-]$	Anchor box i
B_{B}	$[-]$	Box B
B^{*}_{anchor}	$[-]$	Best anchor box
B_{label}	$[-]$	Bounding box of the label
C	$[-]$	Number of classes in a dataset
F_1	$[-]$	F_1-score
FN	$[-]$	False Negative
FP	$[-]$	False positive
$\mathrm{IoU}^{\mathrm{thresh}}$	$[-]$	IoU-threshold
mAP	$[-]$	Mean average precision
$\mathrm{mAP}_{0.50}$	$[-]$	Mean average precision for $\mathrm{IoU}^{\mathrm{thresh}} = 0.50$
$\mathrm{mAP}_{0.75}$	$[-]$	Mean average precision for $\mathrm{IoU}^{\mathrm{thresh}} = 0.75$
$\mathrm{mAP}_{0.90}$	$[-]$	Mean average precision for $\mathrm{IoU}^{\mathrm{thresh}} = 0.90$
S_g	$[-]$	Grid-scale for the grid g
TN	$[-]$	True negative g
TP	$[-]$	True positive g

variable	unit	meaning
a_{h}	$[-]$	example anchor height in normalized image coordinates
a_{w}	$[-]$	example anchor width in normalized image coordinates
a_{h}^{*}	$[-]$	height of the best anchor in normalized image coordinates
a_{w}^{*}	$[-]$	width of the best anchor in normalized image coordinates
$\tilde{a}_{\mathrm{h}}$	$[-]$	example height width in cell coordinates
$\tilde{a}_{\mathrm{w}}$	$[-]$	example anchor width in cell coordinates
$a_{\mathrm{h},gj}$	$[-]$	height of the anchor j from the grid g in normalized image coordinates
$a_{\mathrm{w},gj}$	$[-]$	width of the anchor j from the grid g in normalized image coordinates
$\tilde{a}_{\mathrm{h},gj}$	$[-]$	height of the anchor j from the grid g in cell coordinates
$\tilde{a}_{\mathrm{w},gj}$	$[-]$	width of the anchor j from the grid g in cell coordinates
b_{h}	$[-]$	ground-truth box height in cell coordinates
b_{w}	$[-]$	ground-truth width in cell coordinates
b_{x}	$[-]$	x-position of ground-truth box in cell coordinates
b_{y}	$[-]$	y-position of ground-truth box in cell coordinates
$\hat{b}_{\mathrm{h},gij}$	$[-]$	predicted box width in cell coordinates
$\hat{b}_{\mathrm{w},gij}$	$[-]$	predicted box prediction in cell coordinates
$\hat{b}_{\mathrm{x},gij}$	$[-]$	predicted x-axis position of a box center in cell coordinates
$\hat{b}_{\mathrm{y},gij}$	$[-]$	predicted y-axis position of a box center in cell coordinates
c_{x}	$[-]$	ground-truth x-axis shift of the box center in cell coordinates
c_{y}	$[-]$	ground-truth y-axis shift of the box center in cell coordinates
$\hat{c}_{\mathrm{x},gij}$	$[-]$	predicted x-axis shift of the box center in cell coordinates
$\hat{c}_{\mathrm{y},gij}$	$[-]$	predicted y-axis shift of the box center in cell coordinates
d	$[-]$	depth of a tensor (third dimension)
f	$[-]$	filter dimension
h	$[-]$	height
$\hat{h}_{gij}$	$[-]$	predicted box height in grid g, cell i, anchor section j
k	$[-]$	number of filters
$\mathcal{L}$	$[-]$	loss
$\mathcal{L}_{g}$	$[-]$	loss for grid g
$\mathcal{L}_{\mathrm{class},g}$	$[-]$	class loss for grid g
$\mathcal{L}_{\mathrm{coord},g}$	$[-]$	coordinate loss for grid g
$\mathcal{L}_{\mathrm{noobj},g}$	$[-]$	no-object loss for grid g
$\mathcal{L}_{\mathrm{obj},g}$	$[-]$	object loss for grid g
$\mathcal{L}_{\mathrm{wh},g}$	$[-]$	box dimension loss
$\mathcal{L}_{\mathrm{xy},g}$	$[-]$	center coordinate loss
o	$[-]$	feature map dimension
$p_{\mathrm{class},cgij}$	$[-]$	target probability for class c
$p_{\mathrm{conf},gij}$	$[-]$	target confidence
$p_{\mathrm{score}}^{\mathrm{thresh}}$	$[-]$	score-threshold

variable	**unit**	**meaning**
$\hat{p}_{\text{class},cgij}$	$[-]$	predicted probability for class c
$\hat{p}_{\text{conf},gij}$	$[-]$	prediction confidence
precision	$[-]$	precision
$\text{precision}_{c,n}$	$[-]$	n-th precision for class c
r	$[-]$	image dimension
r_1	$[-]$	first image dimension
r_2	$[-]$	second image dimension
recall	$[-]$	recall
$\text{recall}_{c,n}$	$[-]$	n-th recall for class c
s	$[-]$	stride
$t_{\text{h},gij}$	$[-]$	target height of the box in output format
$t_{\text{w},gij}$	$[-]$	target width of the box in output format
$t_{\text{x},gij}$	$[-]$	target x-position of the box center in output format
$t_{\text{y},gij}$	$[-]$	target y-position of the box center in output format
$\hat{t}_{\text{class},cgij}$	$[-]$	raw output for the probability of class c
$\hat{t}_{\text{conf},gij}$	$[-]$	output value for confidence
$\hat{t}_{\text{h},gij}$	$[-]$	raw output for the height of a bounding box
$\hat{t}_{\text{w},gij}$	$[-]$	raw output for the width of a bounding box
$\hat{t}_{\text{x},gij}$	$[-]$	raw output for the x-position of a bounding box center
$\hat{t}_{\text{y},gij}$	$[-]$	raw output for the y-position of a bounding box center
w	$[-]$	width
$\hat{w}_{gij}$	$[-]$	predicted box width in grid g, cell i, anchor section j
x	$[-]$	x-axis position
x_1	$[-]$	x-axis position of the top left corner
x_2	$[-]$	x-axis position of the bottom right corner
$x_{\text{A},1}$	$[-]$	x-axis position of the top left corner of the box A
$x_{\text{A},2}$	$[-]$	x-axis position of the bottom right corner of the box A
$x_{\text{B},1}$	$[-]$	x-axis position of the top left corner of the box B
$x_{\text{B},2}$	$[-]$	x-axis position of the bottom right corner of the box B
$x_{\text{overlap},1}$	$[-]$	x-axis position of the top left corner of the overlap
$\hat{x}_{gij}$	$[-]$	predicted x-axis position in grid g, cell i, anchor section j
y	$[-]$	x-axis position
y_1	$[-]$	x-axis position of the top left corner
y_2	$[-]$	y-axis position of the bottom right corner
$y_{\text{A},1}$	$[-]$	y-axis position of the top left corner of the box A
$y_{\text{A},2}$	$[-]$	y-axis position of the bottom right corner of the box A
$y_{\text{B},1}$	$[-]$	y-axis position of the top left corner of the box B
$y_{\text{B},2}$	$[-]$	y-axis position of the bottom right corner of the box B
$y_{\text{overlap},1}$	$[-]$	y-axis position of the top left corner of the overlap
$\hat{y}_{gij}$	$[-]$	predicted y-axis position in grid g, cell i, anchor section j

variable	unit	meaning
$\hat{\mathbf{t}}_{gi}$	$[-]$	Output cell array
$\hat{\mathbf{t}}_{gij}$	$[-]$	Output cell section
$\hat{\mathbf{t}}^{\text{cell}}_{gij}$	$[-]$	Predicted cell section in cell coordinates
$\hat{\mathbf{pr}}_{gi}$	$[-]$	Prediction of the i-th cell in grid g
$\hat{\mathbf{pr}}_{gij}$	$[-]$	Prediction of the j anchor section in cell i in grid g
$\mathbf{tr}_{gi}$	$[-]$	target cell of the i-th cell for grid g in image coordinates
$\mathbf{tr}_{gij}$	$[-]$	target section of the j anchor section in cell i for grid g in image coordinates

Greek Symbols

variable	unit	meaning
λ_{class}	$[-]$	class loss weight
λ_{coord}	$[-]$	coordinate loss weight
λ_{noobj}	$[-]$	no-object loss weight
λ_{obj}	$[-]$	object loss weight

Abbreviations

abbr.		meaning
AP	=	Average Precision
BCE	=	Binary Crossentropy
CCE	=	Categorical Crossentropy
CNN	=	Convolutional Neural Network
COCO	=	Common Objects in Context (dataset)
FN	=	False Negative
FP	=	False Positive
FPN	=	Feature Pyramid Networks
FPS	=	Frames per second
mAP	=	mean average precision
NMS	=	Non Maximum Suppression
RCNN	=	Region-based Convolutional Neural Network
ReLU	=	Rectified Linear Unit
R-FCN	=	Region-based Fully Convolutional Network
SCCE	=	Sparse Categorical Crossentropy
SSD	=	Single Shot Detector
TN	=	True Negative
TP	=	True Positive
VOC	=	Visual Object Classes
YOLO	=	You Only Look Once

1 Introduction

Due to the popularity of battery electric vehicles, understanding noise, vibration, and harshness becomes more crucial. Research focused primarily on noise reduction from combustion engines while less attention was paid to other noise sources. Braking systems can cause friction-induced noise. In order to understand the generation and behaviour of noise, brake test systems are set up and studied. In this case a method to monitor occurring noises is needed. The sound is recorded with microphones, however the detection of frequency, duration, intensity of sounds and identifying different sounds by classification, remains a challenge when considering noise-contaminated measurements and a variety of different sounds.

A data transformation can generate pictures from the sound recordings, thereby making the microphone measurements available to computer vision techniques. Object detection methods are used to locate and classify objects on images. This is done by training a convolutional neural network model (CNN) on training data. The conventional approach of detecting squeal is to use a sliding window on sound recordings and apply Fourier transformation. The detector signals positive when the sound pressure level of a frequency exceeds the neighbouring frequencies and a certain threshold. The problem is that the detector is limited to squeals only and susceptible to other noises, such as clicks, wire-brushes and artefacts. For the detection of braking noises, earlier work proposed to use faster RCNN and R-FCN models[24], which are two-stage detectors. *YOLO* is the first one-stage detector, which gained popularity in the object detection community fast. YOLO promises to achieve faster prediction of objects, while maintaining high accuracy. Since then, the accuracy and inference time have been advanced[13, 20, 21]. Therefore, the aim of this thesis is to investigate the usage of the YOLO algorithm on the brake noise problem.

1.1 Project Goal

The goal in this project is to understand the potentials and limitations of deep learning algorithms for sound detection and to implement a YOLOv3 model that detects sound noises on spectrogram data well. A noise detector performs well if the following requirements are fulfilled. First, it is capable of accurately classifying noise recordings. Second, duration and frequency band must be determined with low error.

1.2 Project Outline

This project work is divided in six chapters. Chapter 2 gives detailed explanation behind the theory of the YOLOv3 algorithm. This includes explanation about the network architecture, the loss function and the non maximum suppression algorithm. In chapter 3 an implementation of a YOLOv3 model using the python library tensorflow is explained and different training strategies are discussed. The strategies are tested on the same brake noise dataset as in [24], which are evaluated using two tests in chapter 4. The first test is an object classification task and the second is an object detection task. For both tests average precision and mean average precision are determined for different training strategies. In chapter 5 the results are discussed and compared to previous models. Chapter 6 summarizes the project.

2 Theory of YOLOv3

Object detection is the localization and classification of objects on images. Figure 2.1 shows a general representation of an object detector. Object detection is used in a variety of fields, for instance in tracking of traffic vehicles or detection of cancer cells on x-ray scans. YOLO is such an detector. There exist multiple versions of YOLO, but the introduction of YOLOv3 gained the most popularity. This chapter explains the fundamental building blocks of a YOLOv3 model and the mechanics of the algorithm. First, other object detection algorithms are presented. The *intersection over union* as a metric is introduced. YOLOv3 uses *intersection over union* to determine the best anchors in cells for transforming the annotations, to calculate the ignore mask in the loss function and to process the prediction output in *non maximum suppression*. Then the required data is described. Furthermore the neural net and it's building blocks are characterized and the loss function is explained. Training strategies to increase performance of YOLOv3 are presented. Finally, the prediction of bounding boxes with *non maximum suppression* is demonstrated.

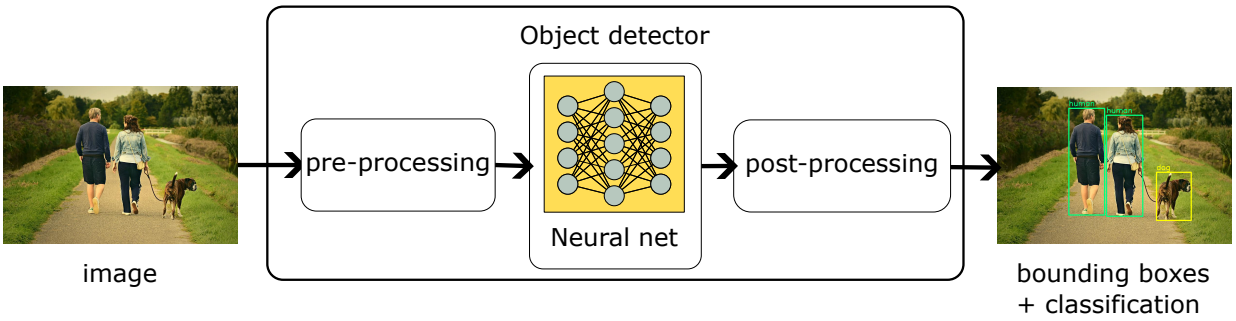

Figure 2.1: An image is taken as input and the result are bounding box predictions with classification for detected objects.

2.1 Background

Object detection algorithms have improved a lot over the last 20 years. Early algorithms used sliding windows, which go through an image at all locations, with different scales and shapes. They try to classify an object for each combination of scale, shape and location, but this process is very inefficient. Afterwards, two-stage detectors were proposed, which use the region proposal technique. In the first stage of the prediction a rectangular shaped region is given as a candidate, where an object is suspected. Then the region is classified with an image classifier in the second stage[29]. RCNN[23], fast RCNN[7], faster RCNN [22] and R-FCN[6] are examples of two-stage detectors. In 2016 *Redmon et al.* proposed the first one-stage detector with *YOLO*, which stands for *You only look once*. Soon after *SSD* [13] was proposed and further improvements came with *YOLOv2*[20] and *YOLOv3*[21]. All one-stage detectors have in common that the localization and classification are done in a single step. The table 2.1 from [15] gives an overview for modern object detection models whose accuracies are measured in *mean average precision* (mAP) on the COCO dataset. This dataset contains images and labels for 80 different classes which includes, for instance, different vehicles, animals and fruits. mAP is an accuracy metric and is explained in Chapter 4. Two-stage object detectors, for instance RCNN, FCN and FPN [12], achieve higher accuracy, but one-stage object detectors are significantly faster in training and prediction time which makes real time object detection possible. Additionally, YOLOv3 is faster and more accurate then previous one-stage detectors. For example, YOLOv3-416 has an inference time of 29ms compared to R-FCN with 85ms [19].

In the next section the data needed for training and prediction and its characteristics are discussed.

Table 2.1: Object detection models with performance on COCO dataset from [15]

Model	Data	Backbone	mAP (%)
Fast RCNN	train	VGG-16	19.7
Faster RCNN	trainval	VGG-16	21.9
R-FCN	trainval	VGG-16	22.6
CoupleNet	trainval	ResNet-101	34.4
Faster RCNN+++	trainval	ResNet-101-C4	34.9
Faster RCNN w FPN	trainval35k	ResNet-101-FPN	36.2
Deformable R-FCN	trainval	Alignmed-inception-ResNet	37.5
umd-ted	trainval	ResNet-101	40.8
Mask RCNN	trainval35k	ResNetXT-101	39.8
DCNv2+Faster RCNN	train118k	ResNet-101	44.8
YOLOv2	trainval35k	DarkNet-19	21.6
YOLOv3	trainval35k	DarkNet-53	33.0
DSSD321	trainval35k	ResNet-101	28.0
SSD513	trainval35k	ResNet-101	31.2
DSSD513	trainval35k	ResNet-101	33.2
RetinaNet500	trainval35k	ResNet-101	34.4
RetinaNet800	trainval35k	ResNet-101-FPN	39.1
M2Det512	trainval35k	ResNet-101	38.8
M2Det800	trainval35k	VGG16	41.0
RefineDet320+	trainval35k	ResNet-101	38.6
RefineDet512+	trainval35k	ResNet-101	41.8
FPN	trainval35k	ResNet101	39.8
NAS-FPN	trainval35k	RetinaNet	40.5
NAS-FPN	trainval35k	AmoebaNet	48.0
Granulated CNN	trainval35k	ResNet-101	32.0

2.2 Data Format for Object Detection

A YOLOv3 model uses image-like data as input. These can be photos, scans or other types of array-like data. The origin of the coordinate system is in the top left corner of an image and the y-axis points downwards. Additionally, the axes are scaled by the width and height of the image, so that $x = 1$ indicates the right edge of an image and $y = 1$ the bottom edge. Pixel values are also scaled to $[0, 1]$; for instance, for RGB images each pixel value is divided by 255. These image-like data contains objects that are to be identified. Before a YOLOv3 can detect these objects, it has to learn how to do this. This is done by supervised learning using a training set. Each example is a pair of an input image and labels which describe the bounding

boxes and class for each object on the image. The bounding boxes are determined by either the x and y coordinates of 2 opposite corners $(x1, y1, x2, y2)$ or the x and y coordinate of the center and the width and height of the box (x, y, w, h). Single class labels are usually given by label encoding: Every class gets a unique tag in the form of a number. Each object is assigned to only one class. An example of a labeled image is shown in Figure 2.2 and in Table 2.2 the corresponding annotations are given where the class is labeled encoded and the bounding box is described in *xywh*-format.

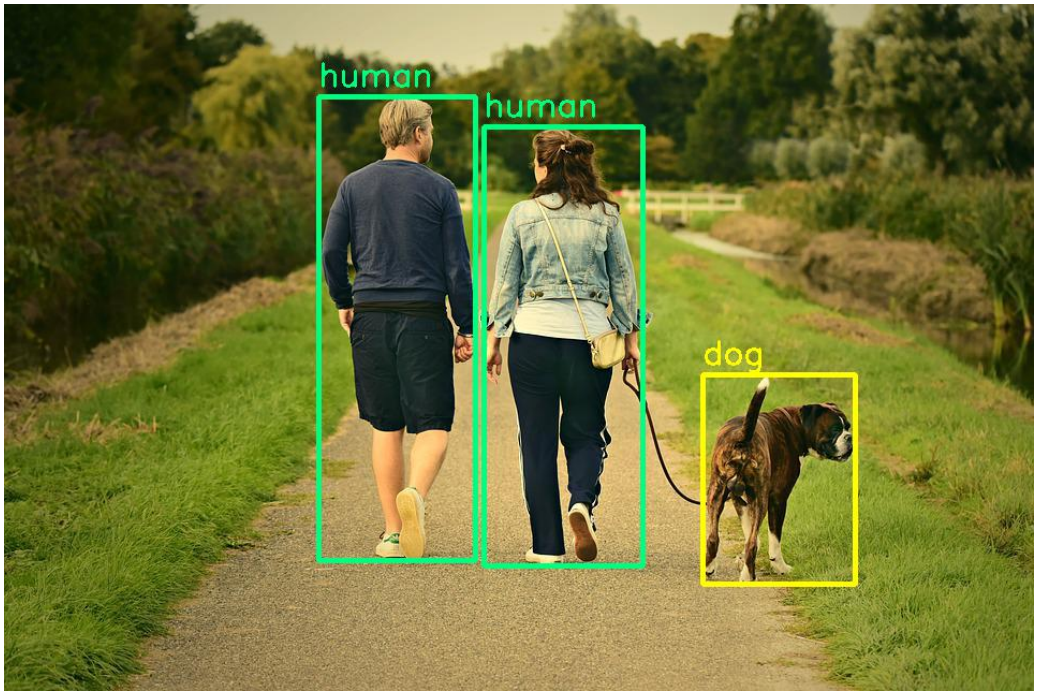

Figure 2.2: 3 objects are labeled by bounding boxes and class annotations

Table 2.2: Single-class labeling with label encoding: The bounding box coordinates are described with x,y,w and h. The class is annotated with label encoding where 0 corresponds to *human* and 1 to *dog*.

class	x	y	w	h
0	0.382	0.472	0.152	0.674
0	0.543	0.498	0.154	0.639
1	0.752	0.693	0.148	0.305

Sometimes we have to use x_1, y_1, x_2, y_2-format, but the x, y, w, h-format is given. For this a simple conversion is done with

$$\begin{aligned} x_1 &= x - w/2 \\ y_1 &= y - h/2 \\ x_2 &= x + w/2 \\ y_2 &= y + h/2. \end{aligned} \tag{2.1}$$

where x and y are the box center coordinates and w, h are width and height. x_1, y_1 describe the top left corner and x_2, y_2 the bottom right corner of a box.

The hand-labeled bounding boxes are ground-truth bounding boxes. The goal of this project is to implement a model that takes images as input and outputs bounding boxes that match the ground-truth well and identifies the class. In the next section a metric is introduced which measures how well two boxes match.

2.3 Intersection over Union

IoU is also known as *Jaccard Index* [26]. It is a metric to measure the similarity of two sample sets. In object detection, IoU is commonly used for evaluating the accuracy of the localization and size estimation by an object detector. The IoU is defined by

$$\text{IoU}(B_A, B_B) = \frac{A_{\text{overlap}}(B_A, B_B)}{A_{\text{union}}(B_A, B_B)}, \tag{2.2}$$

where $A_{\text{overlap}}(B_A, B_B)$ is the area of overlap and $A_{\text{union}}(B_A, B_B)$ is the area of union of the boxes B_A and B_B. Figure 2.3 illustrates the IoU of two boxes. The IoU ranges from 0 to 1. IoU = 0 indicates that the boxes do not overlap and IoU = 1 means that the boxes are identical. Then the *area of overlap* is calculated with

$$\begin{aligned} x_{\text{overlap},1} &= \max(x_{\text{A},1}, x_{\text{B},1}) \\ y_{\text{overlap},1} &= \max(y_{\text{A},1}, y_{\text{B},1}) \\ x_{\text{overlap},2} &= \min(x_{\text{A},2}, x_{\text{B},2}) \\ y_{\text{overlap},2} &= \min(y_{\text{A},2}, y_{\text{B},2}), \end{aligned} \tag{2.3}$$

where $x_{\text{A},1}, y_{\text{A},1}, x_{\text{A},2}$ and $y_{\text{A},2}$ are the coordinates of the box A and $x_{\text{B},1}, y_{\text{B},1}, x_{\text{B},2}$ and $y_{\text{B},2}$ are the coordinates of the box B, while $x_{\text{overlap},1}, y_{\text{overlap},1}, x_{\text{overlap},2}$ and $y_{\text{overlap},2}$ are the coordinates

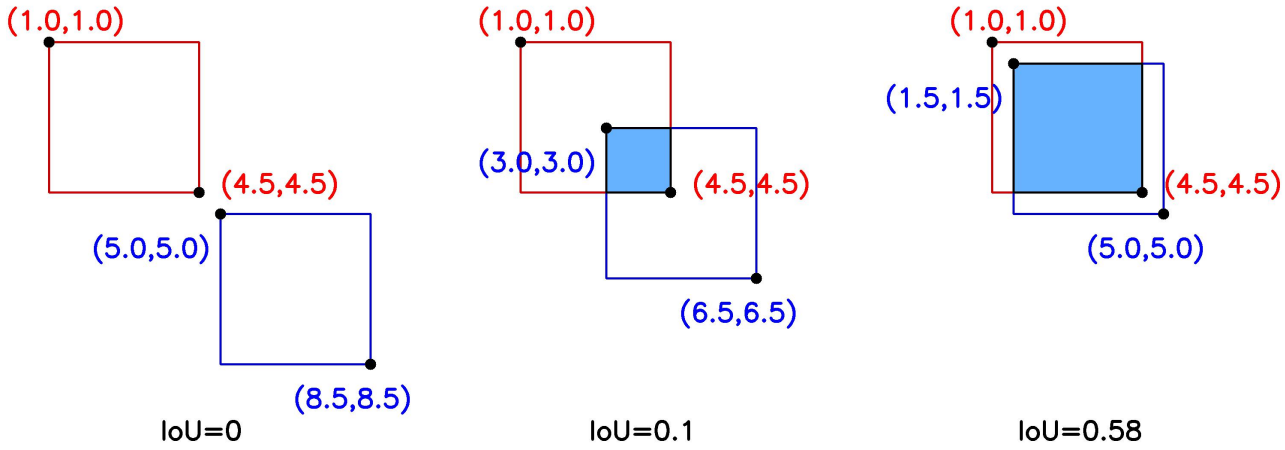

Figure 2.3: IoU examples: The left boxes are not overlapping resulting in IoU $= 0$. The middle and boxes overlap with IoU $= 0.53$ and IoU $= 0.73$. The intersection area is marked in light-blue color.

of the intersection candidate. With these coordinates the area of overlap between boxes A and B is computed with

$$A_{\text{overlap}} = \max(0, x_{\text{overlap},2} - x_{\text{overlap},1}) \cdot \max(0, y_{\text{overlap},2} - y_{\text{overlap},1}). \tag{2.4}$$

The *area of union* is needed for the denominator of IoU. It is computed with

$$\begin{aligned} A_{\text{A}} &= (x_{\text{A},2} - x_{\text{A},1}) * (y_{\text{A},2} - y_{\text{A},1}) \\ A_{\text{B}} &= (x_{\text{B},2} - x_{\text{B},1}) * (y_{\text{B},2} - y_{\text{B},1}) \\ A_{\text{union}} &= A_{\text{A}} + A_{\text{B}} - A_{\text{overlap}}, \end{aligned} \tag{2.5}$$

where A_{A} and A_{B} are the areas of boxes A and B. Now the IoU can be calculated with Equation (2.2). It is used in the calculation of the loss, processing of the prediction output and as evaluation metric.

In the next section anchor boxes are introduced.

2.4 Anchors

Anchors are boxes, which are designed to be of similar width and height as the bounding boxes encountered across the training dataset. They are used as starting points for the bounding box predictions. Using anchors makes predicting new bounding box easier. They act as priors

to potential bounding box shapes, thereby providing better initialization for the bounding box regression task.

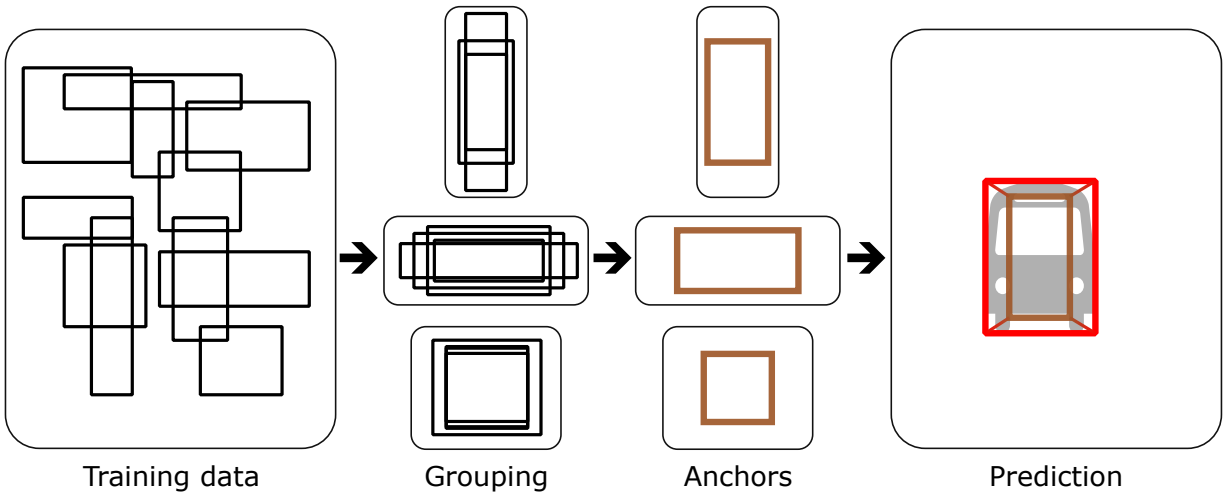

Figure 2.4: Similar shaped bounding boxes are grouped together and for each group an anchor box is generated. Anchor boxes are then reshaped to produce bounding box predictions.

The anchors are determined before the training phase of the neural network begins. Figure 2.4 illustrates the basic process of creating anchors and using them. Anchor boxes are obtained from the training data set. First, the number of anchors are chosen. Then the dimensions of each anchor box are determined by analyzing the training dataset with statistical methods. One way is to hand pick anchor boxes. Figure 2.5 is an example of a scatter plot with objects from the pascal VOC dataset and the original YOLOv3 anchors. Every red point denotes the width and height of a bounding box from the dataset. The black points are anchor boxes. These were computed by applying a clustering method on the COCO dataset.

A more accurate method is to run a *k-means-clustering* algorithm on the training dataset, with k being the number of anchor boxes. It was shown that this increases the perfomance of a YOLO model [20]. It is also possible to use the anchor boxes generated from different datasets; for instance, the anchors of the original YOLOv3 implementation used anchors which were generated from the *COCO*-dataset. In the original YOLOv3 model 9 anchors were used, which are different in shape and size. 3 small anchors for the (52×52)-grid, 3 medium-sized anchors for the (26×26)-grid and 3 big anchors for the (13×13)-grid outputs.

In the next section the basic operation of object detection is discussed.

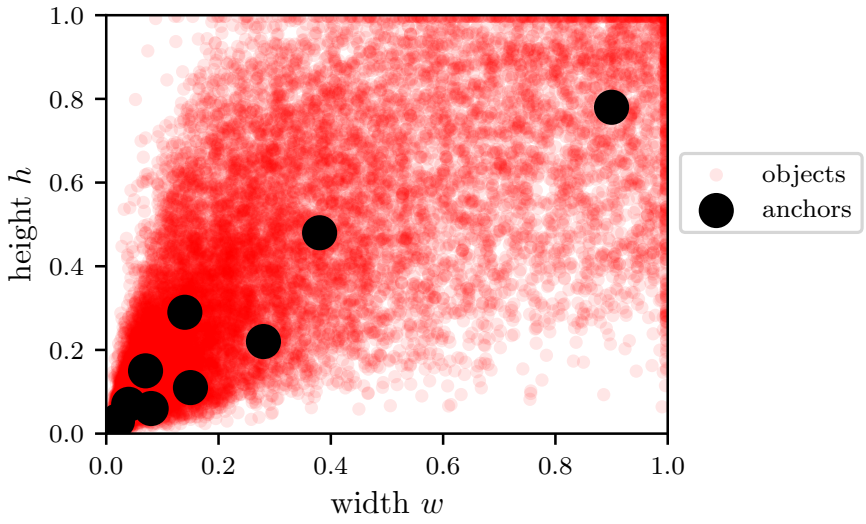

Figure 2.5: All labelled boxes are plotted with respect to their height and their width. 9 anchor boxes were hand picked, so that every labelled box has an anchor with similar dimensions.

2.5 Discrete Convolution

The YOLOv3 architecture [21] consists of a convolutional neural network which takes images as inputs. Images can have multiple color channels. For the original YOLOv3 structure the dimensions of an image must be a multiple of 32, which is usually 416×416 [21]. It uses discrete convolution to identify shapes and patterns in an image. The convolution operation is carried out by the filter, which is a matrix, that is usually smaller than the image. The filter is applied to the image like a sliding window and for each position the overlapping entries of the matrices are multiplied entry-wise. The sum is stored in a feature map. An example of this is shown in Figure 2.6. The feature map can be interpreted as an identifier for a specific *shape* defined by the filter. In the example of Figure 2.6 the *shape* of the filter resembles an X. It weights the value of the middle pixel and its diagonal neighbors with 1 and ignores its vertical and horizontal neighbors. The feature map displays at which position the shape of the filter matches the values in the image the best.

From Figure 2.6 we can see that the feature map is smaller than the original image, because the filter does not fully convolute the pixels at the edges. These pixels contribute less to the feature map which results in the loss of information at the edges [17]. To avoid this the image is padded with zeros. Then, the size of the feature map stays the same after convolution and all pixels contribute equally to the feature map. This technique is called *same padding* while using no padding is called *valid padding*. The step size of the sliding window is called the *stride*

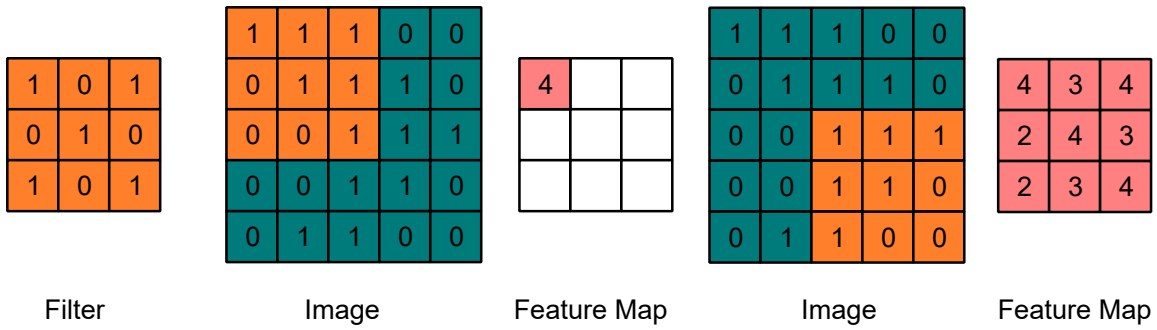

Figure 2.6: Valid convolution: The filter is highlighted with orange on the image (green). The 9 top left entries of the image are weighted by the filter and summed up. The result is 4 on the top left entry of the feature map. The image is convoluted with stride $s = 1$ and the resulting feature map is on the right side.

s. In Figure 2.6 the stride in x and y-direction is 1, because the filter moves to every pixel. s = 2 indicates that the filter is applied only to every second pixel, which results in a smaller feature map. Given an image $\in \mathbb{R}^{(r \times r)}$, a filter $\in \mathbb{R}^{(f \times f)}$ and the stride s the dimensions of a feature map $o \times o$ are determined with

$$o = r \backslash s + 1 - f, \tag{2.6}$$

where $\backslash$ is the integer division operation. If the top and left edges of an image are zero-padded, then using a filter with $s = 2$ and filter dimensions (3×3) always results in a feature map with exactly half the dimensions of the input. A filter dimension of (1×1) means that the convolutional layer evaluates only 1 pixel. Usually input matrices are 3-dimensional, but we refer to the third dimension as depth. RGB images have a depth of 3, because they have 3 color channels, and are therefore matrices with dimensions $\mathbb{R}^{(r_1 \times r_2 \times 3)}$, where r_1 and r_2 are the width and height of the image. To convolute an image with RGB-channels the dimensions of a 2D-filter are extended by the depth of the image. Figure 2.7 shows an example of convolution of an RGB image.

Convolution is the basic operation of convolutional layers which is the basic component of a YOLOv3 neural network. The convolutional layers are discussed in the next section.

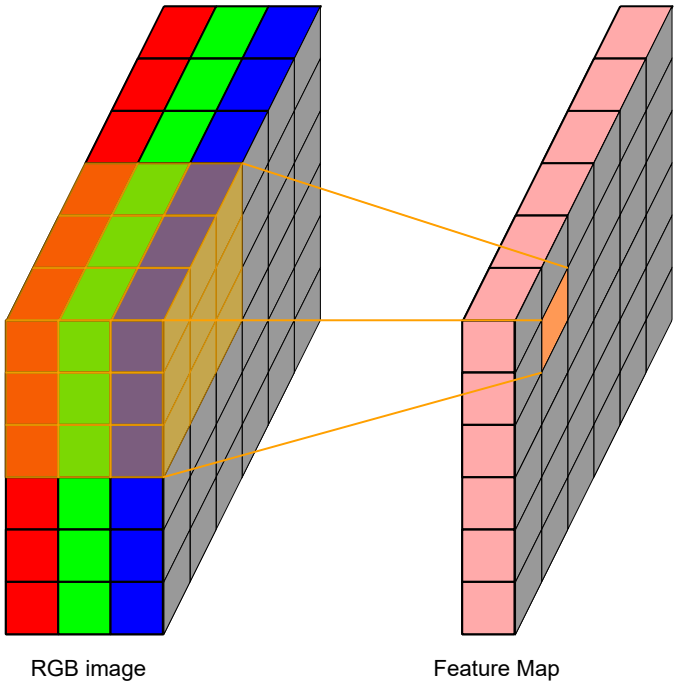

Figure 2.7: RGB image convolution with a matrix $\in \mathbb{R}^{(3\times3\times3)}$ as filter using valid padding

2.6 Convolutional Layer

The convolution of an image is carried out by a convolutional layer with multiple different filters which results in multiple feature maps for each filter. This is represented by a 3D-tensor with dimensions $\mathbb{R}^{(r_1\times r_2\times k)}$, where m and n are the dimensions of each filter and k is the number of filters. Each entry of each filter is learned by the neural network in the training step. 3-dimensional filters are able to convolve 3D-tensor. A filter with dimension $\mathbb{R}^2(3\times3\times3)$ convolves an RGB image in to a feature map with dimension $\mathbb{R}^{(r_1\times r_2)}$ using same padding. The convolution of an RGB image with a convolutional layer is shown in Figure 2.8. 16 filters are used to process an RGB image with dimensions $32\times32\times3$ into a $32\times32\times16$ feature map. Each layer of the feature map corresponds to a different filter with different weights for each individual entry. Therefore, each filter favors a different pattern on the image. If such a pattern is found, it is marked at the corresponding cell in the feature map with a high value. In CNNs the output of a convolutional layer is used as input for another convolutional layer again. The result is another set of feature maps which stores information about shapes and patterns inside the first feature map tensor. These patterns have a more complicated form. The more convolutional layers are chained, the more complicated patterns can be detected.

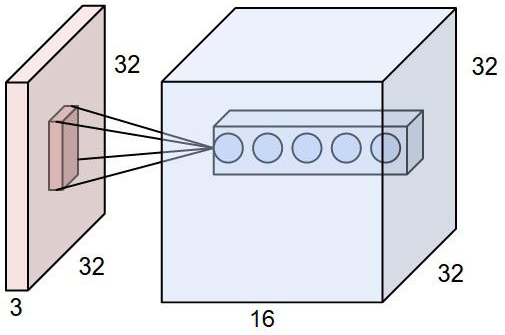

Figure 2.8: Convolution with multiple filters from [25]: A $32 \times 32 \times 3$ tensor is processed by convolution operation to a $32 \times 32 \times 16$ tensor using 16 filters and valid padding.

Convolutional layers are the main component of the YOLOv3 model, which will be explained in the next section.

2.7 Neural architecture of YOLOv3

The neural architecture of YOLOv3 consists of 2 parts. The first part is the feature extractor, which consists of multiple convolutional layers. The purpose of it is to create a feature map representation of input images. Then the feature maps are passed to the second part, which is called the detection head, where the actual detection is performed. Its output are 3 prediction tensors for 3 different object scales. Figure 2.9 illustrates the architecture.

YOLOv3 uses different types of layers with different filter dimensions ($f \times f \times d$) and strides s:

1. $f = 3$ and $s = 1$
2. $f = 3$ and $s = 2$
3. $f = 1$ and $s = 1$

A convolutional layer with f = 1 is used to reduce the depth of an input tensor before another convolution layer with bigger filter size is applied which results in better performance for object classification tasks [11]. It is also used as an output layer for the YOLOv3 neural net. In the training step the images are fed to the network in batches. After each convolutional layer we apply batch normalization to all output tensors of a batch. For information about batch normalization see the appendix A. Finally a *leaky ReLU* is used as an activation function. Convolutional, batch normalization and leaky ReLu layer form a convolutional block as shown in Figure 2.10. A residual unit is made up of 2 convolutional blocks and a *residual skip*

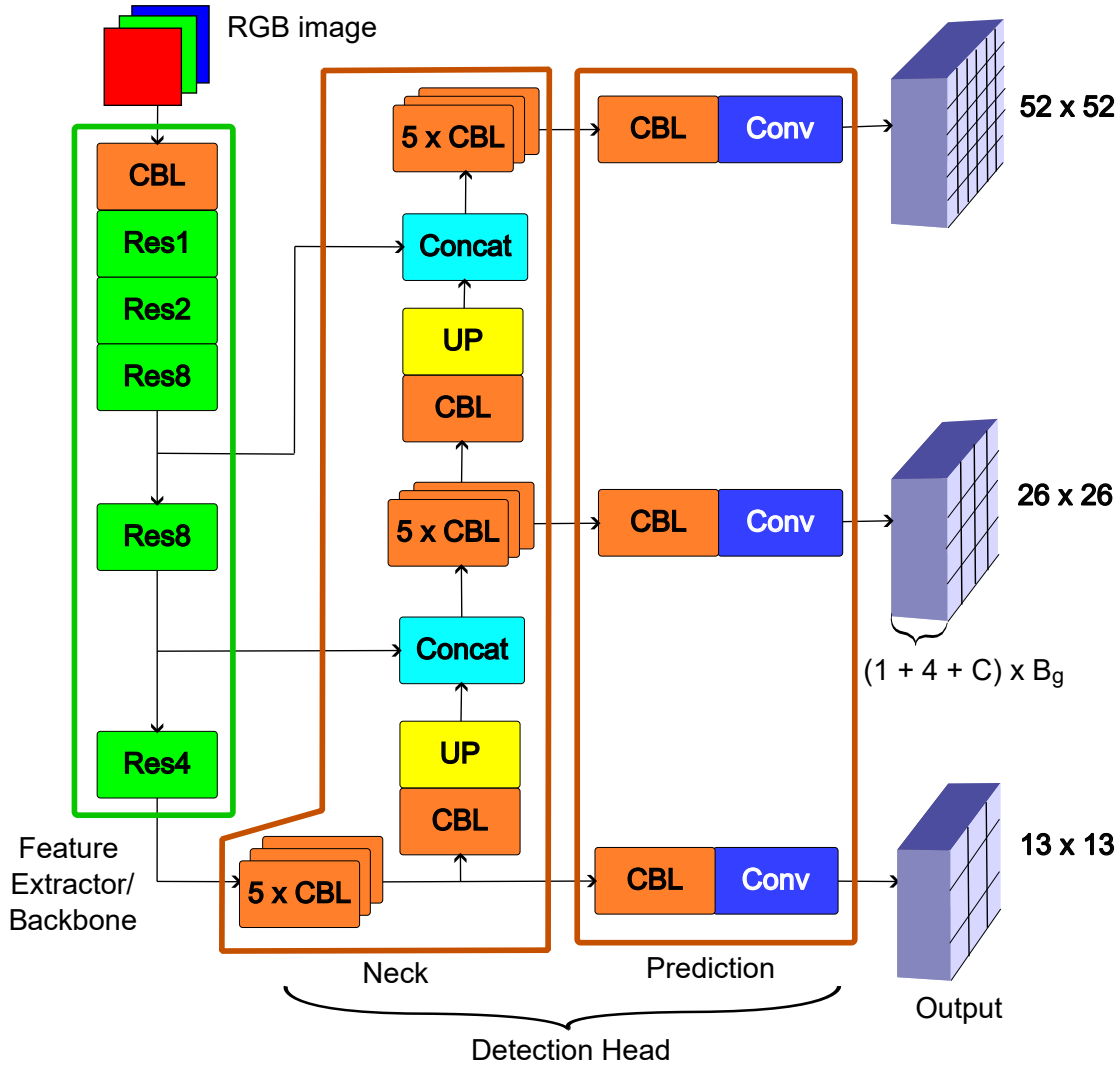

Figure 2.9: YOLOv3 net structure: The left side is the *darknet53* feature extractor. On the right side is the *YOLOv3* detection head. *UP* is an up-sampling layer which doubles the resolution of a tensor (except the depth). *Concat* is a concatenate operation for merging two tensors with shared dimensions.

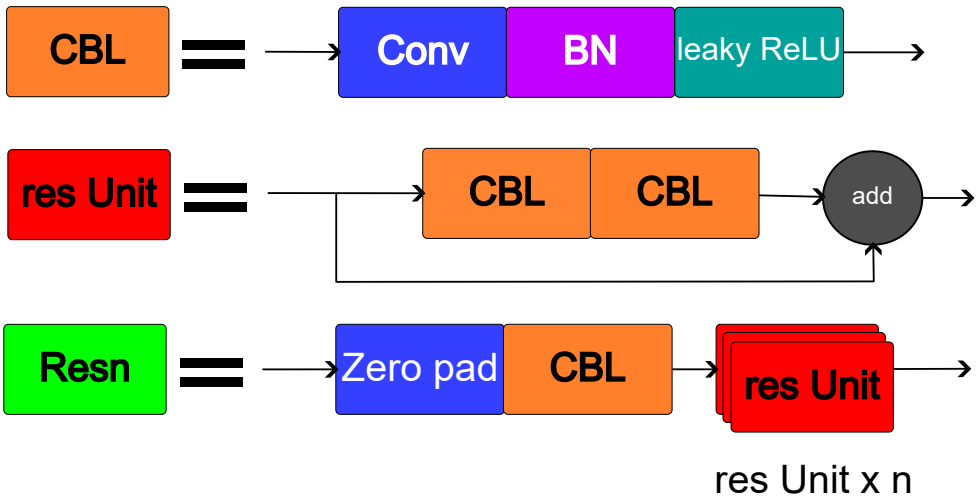

Figure 2.10: Different blocks used in YOLOv3 architecture: The top shows a convolutional block, the middle a residual unit and the bottom a residual block.

where the input tensor is added to the output of the convolutional blocks. This addresses the *degradation* problem of neural networks with a high amount of layers [8]. A residual block contains zero padding for the top and left edges and a convolutional block using a convolutional layer with $s = 2$ followed by residual units as shown in Figure 2.10. After each residual block the dimensions of an input image or feature map are reduced by half; for instance, a 416×416 image is transformed into a 208×208 feature map. In the Table 2.3 the backbone of the YOLOv3 model is shown, which consists of 53 convolutional blocks. For an input image $\in \mathbb{R}^{(416 \times 416)}$ the feature maps of the last 3 residual blocks are of dimensions 52×52, 26×26 and 13×13, which are also called grids. The lowest resolution grid stores the biggest and most complicated patterns. Each cell in a grid contains information about a potential object in the cell location of the input image. This is why the backbone is also called the *feature extractor* which is the first part of the YOLOv3 architecture.

The second part is the detection head. The complete YOLOv3 architecture is shown in Figure 2.9. The detection head consists of the *neck* and the *dense prediction*. In the neck the 13×13 feature map of the last residual block is passed to a convolutional set which contains 5 convolutional blocks. This is repeated for the output of the 2nd last and 3rd last residual blocks, but in addition the lower resolution tensor after the convolutional set is *up-sampled* and concatenated with the next higher resolution feature map before applying a convolutional set. The up-sampling is done by a convolution block with an *up-sampling* layer which doubles the resolution of the tensor preserving the depth. This enables the high-resolution network to process more complex information from the low resolution grid. The resulting tensors are then

Table 2.3: Darknet-53 [21]: The object classifier was invented by *Redmon et al.* and has 53 convolutional blocks. The residual units are marked with boxes. The left number next to a residual residual unit specifies how often the unit is repeated. In YOLOv3 the layers after the last residual block are omitted.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

passed to the dense prediction layers, which consists of a convolutional block and a convolutional layer with $f = 1$, which is the output layer. The result is the neural net output with 3 grids which differ in resolution:

- 13×13 for big objects
- 26×26 for medium-sized objects
- 52×52 for small objects

The depth of each grid is determined by the number of filters in the output layer. Each cell is responsible for the detection of objects inside the corresponding cell boundaries. This and the transformations of the prediction output will be explained in the next section.

2.8 Network Output

The output of the network are three 3D-tensors, which are also called grids, that have the center of the bounding box in that cell. Given a spatial descretization of $13 \cdot 13$ cells, this will allow the network to make predictions about $(13 \times 13 \times 3)$ objects. A cell of a grid is an array with bounding box and class label predictions. In the standard implementation of YOLOv3 each cell can predict up to 3 objects. Therefore, the number of anchors per grid is $B_g = 3$. A cell in a grid is a 1D tensor defined as

$$\hat{\mathbf{t}}_{gi} = [\hat{\mathbf{t}}_{gi,1}, \hat{\mathbf{t}}_{gi,2}, ..., \hat{\mathbf{t}}_{gi,\mathrm{B_g}}] \ \in \mathbb{R}^d \ , \tag{2.7}$$

where g is the grid index and i is the cell index of a grid with $i \in 1, ..., S_g^2$, S_g is the grid size and B_g is the number of anchors per grid g. Each $\hat{\mathbf{t}}_{gi,j}$ is an anchor section corresponding to the j-th anchor with $j \in 1, .., B_g$. Each section is defined as

$$\hat{\mathbf{t}}_{gi,j} = [\hat{t}_{\mathrm{conf},gij}, \hat{t}_{\mathrm{x},gij}, \hat{t}_{\mathrm{y},gij}, \hat{t}_{\mathrm{w},gij}, \hat{t}_{\mathrm{h},gij}, \hat{t}_{\mathrm{class},1,gij}, ..., \hat{t}_{\mathrm{class},C,gij}] \in \mathbb{R}^{d/B_g} \ , \tag{2.8}$$

where C is the number of classes in the dataset, $\hat{t}_{\mathrm{conf},gij} \in [0, 1]$ is the raw output confidence which encodes the probability to detect an object centered in that cell. $\hat{t}_{\mathrm{x},gij}$, $\hat{t}_{\mathrm{y},gij}$, $\hat{t}_{\mathrm{w},gij}$ and $\hat{t}_{\mathrm{h},gij}$ are the raw spatial location information for the corresponding bounding box prediction and $\hat{t}_{\mathrm{class},1,gij}$ to $\hat{t}_{\mathrm{class},C,gij}$ determine the class prediction in the sense of a one-hot encoded class label. The total length of $\hat{\mathbf{t}}_{gi}$ is the depth d of the output tensor, which is $d = (1 + 4 + C) \cdot B_g$. A cell is illustrated in Figure 2.11. The raw output of the network are transformed to obtain bounding box and class predictions. Each element in a cell section $\hat{\mathbf{t}}_{gi,j}$ is transformed to

$$\begin{aligned}
\hat{p}_{\mathrm{conf},gij} &= \sigma(\hat{t}_{\mathrm{conf},gij}) \\
\hat{b}_{\mathrm{x},gij} &= \sigma(\hat{t}_{\mathrm{x},gij}) + \hat{c}_{\mathrm{x},gij} \\
\hat{b}_{\mathrm{y},gij} &= \sigma(\hat{t}_{\mathrm{y},gij}) + \hat{c}_{\mathrm{y},gij} \\
\hat{b}_{\mathrm{w},gij} &= \tilde{a}_{\mathrm{w},gj} e^{\hat{t}_{\mathrm{w},gij}} \\
\hat{b}_{\mathrm{h},gij} &= \tilde{a}_{\mathrm{h},gj} e^{\hat{t}_{\mathrm{h},gij}} \\
\hat{p}_{\mathrm{class},1,gij} &= \sigma(\hat{t}_{\mathrm{class},1,gij}) \\
&\vdots \\
\hat{p}_{\mathrm{class},C,gij} &= \sigma(\hat{t}_{\mathrm{class},C,gij}) \ ,
\end{aligned} \tag{2.9}$$

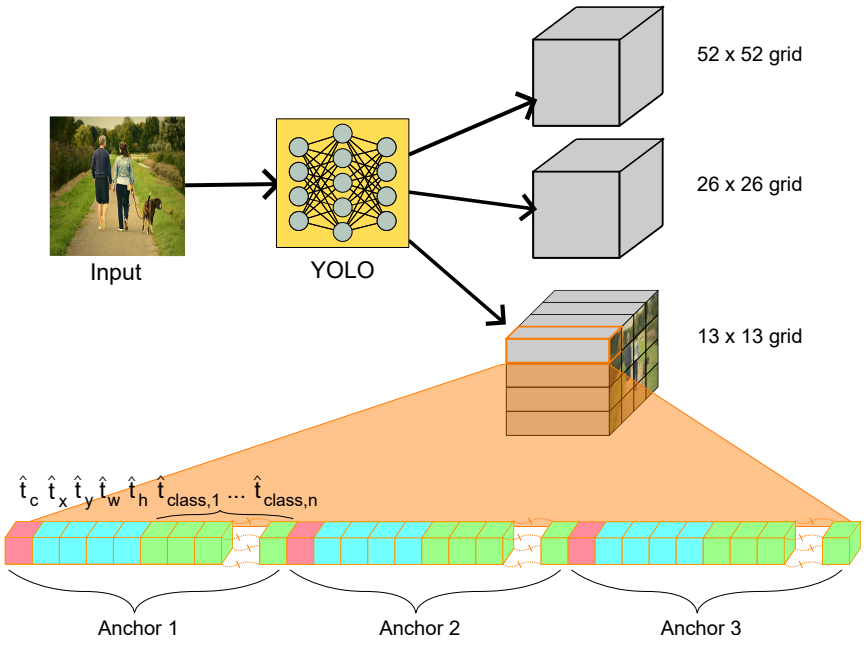

Figure 2.11: Cell array: For every cell there is one prediction for each anchor. In this example there are 3 predictions. Each prediction estimates the bounding box coordinates, the confidence and the probability of a class.

where $\sigma(x)$ is the sigmoid function and $\hat{c}_{\mathrm{x},gij}$, $\hat{c}_{\mathrm{y},gij}$ are the predicted box center shift in cell coordinates. $\hat{p}_{\mathrm{conf}}$ is the *prediction confidence* and determines if an object is predicted with the corresponding anchor. A high confidence value means that a prediction has high certainty. $\hat{b}_x$, $\hat{b}_y$, $\hat{b}_w$ and $\hat{b}_h$ determine the location and dimensions of the predicted box in *grid-scaled* coordinates. To get the grid-scaled coordinate system, the axis of the w,h-scaled coordinate system are multiplied by the number of grid cells in x- and y-direction S_g, which is shown in Figure 2.12.

Figure 2.13 shows an example of the resulting bounding box prediction of the array section

$$\hat{\mathbf{t}}_{gij}^{\mathrm{cell}} = [0.9, 1.7, 1.5, 1.4, 2.0, ...] \,, \tag{2.10}$$

where the box center and dimension are given in cell coordinates.

The cell is located in the middle of a (3×3)-grid, which corresponds to $\hat{c}_x = 1$ and $\hat{c}_y = 1$. Inside the cell the center of the box is shifted by $\sigma(\hat{t}_x) = 0.7$ in x-direction and by $\sigma(\hat{t}_y) = 0.5$ in y-direction. In general the sigma function maps input values from $[-\inf, \inf]$ to $[0, 1]$. This restricts the box center to the cell boundaries. For the prediction of the box dimensions a grid-

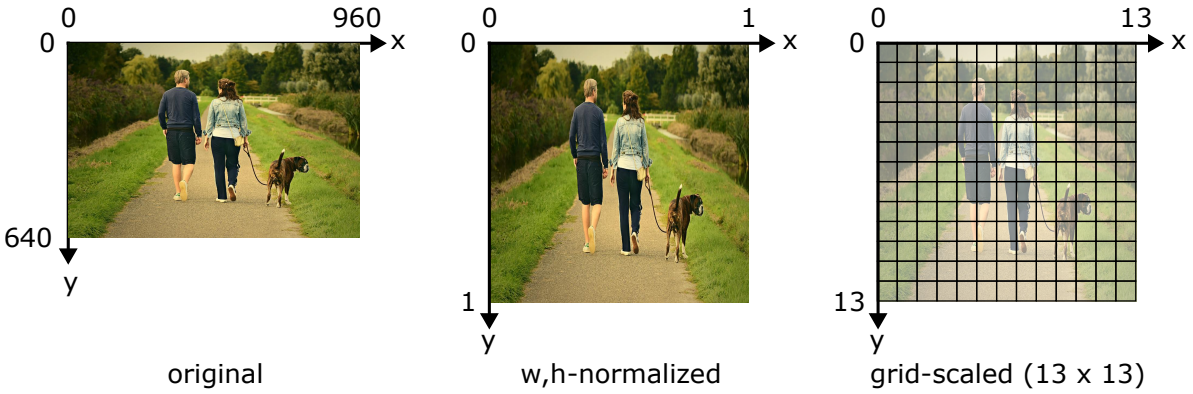

Figure 2.12: On the left is the original image with the pixel positions as coordinates. In the middle image the width and height are normalized to 1. The right image shows the same image with grid-scaled coordinates corresponding to a (13 × 13)-grid.

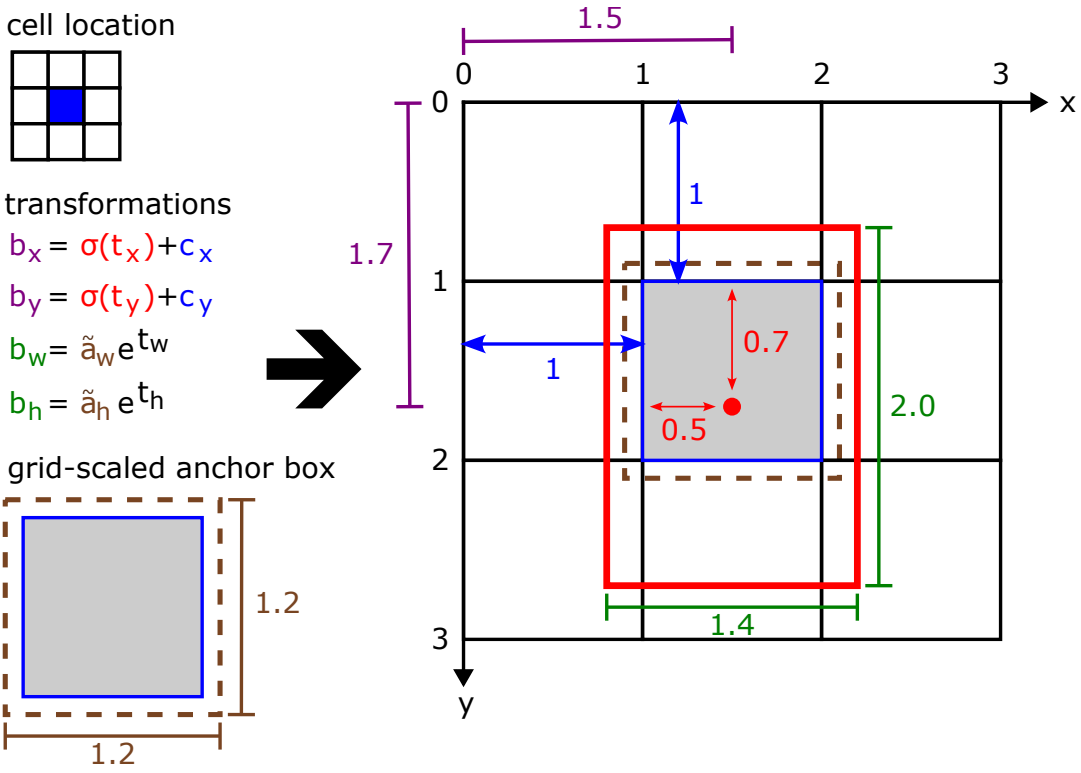

Figure 2.13: $\hat{t}_x$, $\hat{t}_y$, $\hat{t}_w$ and $\hat{t}_h$ are outputs of the highlighted cell. A box center is located inside the cell boundaries and is shifted by the value of the sigmoid functions $\sigma(t_x)$ and $\sigma(t_y)$. For more information on the sigmoid see the appendix A. The width and height of an anchor box are adjusted by multiplying with the exponential functions $\exp t_w$ and $\exp t_h$.

scaled anchor box is used with width $\tilde{a}_w$ and $\tilde{a}_h$. We get these by multiplying the normalized anchor box width $a_{\mathrm{w},gj}$ and height $a_{\mathrm{h},gj}$ with the grid-scale $\mathrm{S_g}$, which results in

$$\begin{aligned} \tilde{a}_{\mathrm{w},gj} &= a_{\mathrm{w},gj} \cdot \mathrm{S_g} \\ \tilde{a}_{\mathrm{h},gj} &= a_{\mathrm{h},gj} \cdot \mathrm{S_g} \ . \end{aligned} \tag{2.11}$$

In the example of Figure 2.13 the grid-scale is 3 and the grid-scaled anchor width and height are $\tilde{a}_{\mathrm{w}} = 1.2$ and $\tilde{a}_{\mathrm{h}} = 1.2$ which are then multiplied with exponential functions. This prevents predicting bounding boxes that have negative widths or heights. The result is a bounding box of width $\hat{b}_{\mathrm{w}} = 1.4$ and height $\hat{b}_{\mathrm{h}} = 2.0$.

In general the box center for all predictions is restricted to the cell boundaries of the corresponding cell, but the widths and heights are not restricted by the cell boundaries and can take only positive values.

The last entries of a cell array section $\hat{p}_{\mathrm{class},1,gij}$ to $\hat{p}_{\mathrm{class},C,gij}$ determine the class. Each entry gives a probability for belonging to a specific class. In a multi-label classification task the predicted object is assigned to the class probability with the highest value. For example, with a 4 class dataset a cell array section is given as

$$\hat{\mathbf{t}}^{\mathrm{cell}}_{gij} = [..., 0.1, 0.2, 0.7, 0.9] \ . \tag{2.12}$$

If the threshold for classification is lower than 0.7, the classes 3 and 4 are predicted. The last step is to transform $\hat{b}_{\mathrm{x}}$, $\hat{b}_{\mathrm{y}}$, $\hat{b}_{\mathrm{w}}$ and $\hat{b}_{\mathrm{h}}$ to image coordinates. To get the values for the normalized coordinate system as shown in Figure 2.12, they are divided by the grid-scale $\mathrm{S_g}$. The result is

$$\begin{aligned} \hat{x}_{gij} &= \hat{b}_{\mathrm{x},gij} / \mathrm{S_g} \\ \hat{y}_{gij} &= \hat{b}_{\mathrm{y},gij} / \mathrm{S_g} \\ \hat{w}_{gij} &= \hat{b}_{\mathrm{w},gij} / \mathrm{S_g} \\ \hat{h}_{gij} &= \hat{b}_{\mathrm{h},gij} / \mathrm{S_g} \ , \end{aligned} \tag{2.13}$$

where $\hat{x}_{gij}$, $\hat{y}_{gij}$, $\hat{w}_{gij}$ and $\hat{h}_{gij}$ are the center coordinates and dimensions. The full transformations from output to prediction are

$$\begin{aligned}
\hat{p}_{\text{conf},gij} =& \sigma(\hat{t}_{\text{class},gij}) \\
\hat{x}_{gij} =& (\sigma(\hat{t}_{\text{x},gij}) + \hat{c}_{\text{x},gi})/\text{S} \\
\hat{y}_{gij} =& (\sigma(\hat{t}_{\text{y},gij}) + \hat{c}_{\text{y},gi})/\text{S} \\
\hat{w}_{gij} =& (\tilde{a}_{\text{w},gj} e^{\hat{t}_{\text{w},gij}})/\text{S} \\
\hat{y}_{gij} =& (\hat{a}_{\text{h},gij} e^{\hat{t}_{\text{h},gij}})/\text{S} \\
\hat{p}_{\text{class},1,gij} =& \sigma(\hat{t}_{\text{class},1,gij}) \\
& \vdots \\
\hat{p}_{\text{class},C,gij} =& \sigma(\hat{t}_{\text{class},C,gij}) \ .
\end{aligned} \tag{2.14}$$

A predicted anchor section is then defined as

$$\hat{\mathbf{pr}}_{gij} = [\hat{p}_{\text{conf},gij}, \hat{x}_{gij}, \hat{y}_{gij}, \hat{w}_{gij}, \hat{h}_{gij}, \hat{p}_{\text{class},1,gij}, ..., \hat{p}_{\text{class},C,gij}] \in \mathbb{R}^{d/B_g}. \tag{2.15}$$

A cell prediction consists of B_g sections, which can be written as

$$\hat{\mathbf{pr}}_{gi} = [\hat{\mathbf{pr}}_{gi,1}, \hat{\mathbf{pr}}_{gi,2}, ..., \hat{\mathbf{pr}}_{gi,B_g}] \in \mathbb{R}^d \ , \tag{2.16}$$

where $\mathbf{pr}_{\text{i}}$ with $i \in \{1, 2, 3\}$ are anchor sections. Therefore, a cell prediction with transformations is defined as

$$\begin{aligned}
\hat{\mathbf{pr}}_{gi} =& [\hat{p}_{\text{conf},gi,1}, \hat{x}_{gi,1}, \hat{y}_{gi,1}, \hat{w}_{gi,1}, \hat{h}_{gi,1}, \hat{p}_{\text{class},1,gi,1}, ..., \hat{p}_{\text{class},C,gi,1}, ..., \\
& \hat{p}_{\text{conf},gi,2}, \hat{x}_{gi,2}, \hat{y}_{gi,2}, \hat{w}_{gi,2}, \hat{h}_{gi,2}, \hat{p}_{\text{class},1,gi,2}, ..., \hat{p}_{\text{class},C,gi,2}, ..., \\
& \vdots \\
& \hat{p}_{\text{conf},gi,B_g}, \hat{x}_{gi,B_g}, \hat{y}_{gi,B_g}, \hat{w}_{gi,B_g}, \hat{h}_{gi,B_g}, \hat{p}_{\text{class},1,gi,B_g}, ..., \hat{p}_{\text{class},C,gi,B_g}] \in \mathbb{R}^d.
\end{aligned} \tag{2.17}$$

The transformations lead to predictions about the confidence value, the box center coordinates, box dimensions and the class. However before the model is able to predict it needs to be trained. For this the outputs and the ground-truth labels need to be transformed in a format which enables the calculation of the error between prediction and ground-truth. This is explained in the next section.

2.9 Target preparation

In the training step the error of the predictions to the ground-truth is calculated with the loss function. To do this the label information on global image level are used to construct the *targets*. They need to be of the same format as the prediction outputs of the models, which are 3 tensor at different spatial resolution. Additionally, each grid cell has 3 sections. The elements of each anchor section contain the information of a bounding box prediction and its classification. Therefore, the global labels on image level, which are given in image coordinates, are also transformed to 3 grids with cells. A target cell for the grid g is a 1D tensor, which is given as

$$\mathbf{tr}_{gi} = [\mathbf{tr}_{gi,1}, \mathbf{tr}_{gi,2}, ..., \mathbf{tr}_{gi,B_g}] \in \mathbb{R}^d \tag{2.18}$$

with each anchor section $\mathbf{tr}_{gij} \in \mathbb{R}^{d/B_h}$. This is similar to Equation (2.16). Figure 2.14 illustrates the steps for transforming the labels. The first step is to determine the grid and the anchor section by comparing ground-truth bounding boxes with the anchor boxes. This is done by determining the *best anchor* with

$$B^*_{\text{anchor}} = \arg\max_a \text{IoU}^*(a, B_{\text{label}}) \ , \ a \in \{B_{\text{anchor},1}, B_{\text{anchor},2}, ..., B_{\text{anchor},8}, B_{\text{anchor},9}\} \ , \tag{2.19}$$

where B_{label} is the bounding box of a single label entry and $B_{\text{anchor},i}$ is the i-th anchor box. IoU* is the *broadcast intersection over union* and is similar to the IoU which is computed on bounding boxes centered about their centers, i.e. solely rating width and height differences. The highest IoU* determines which anchor box is the most similar to the ground-truth box in shape and size. Since each anchor represents a scale and a section in the target array $\mathbf{tr}_{gi}$, the ground truth bounding box information is located in the corresponding target tensor representing the ground truth for training. The location and dimensions of the box is computed to grid-scaled coordinates with

$$\begin{aligned} b_{\text{x}} &= x \cdot \text{S}_{\text{g}} \\ b_{\text{y}} &= y \cdot \text{S}_{\text{g}} \\ b_{\text{w}} &= w \cdot \text{S}_{\text{g}} \\ b_{\text{h}} &= h \cdot \text{S}_{\text{g}} \ , \end{aligned} \tag{2.20}$$

where x,y,w and h are the box center and box dimensions in image coordinates and S_g is the grid-scale. Therefore, b_x,b_y,b_w and b_h are the box center and box dimensions in grid-scaled coordinates. The bounding box and classification information need to be located on the grid. The corresponding cell is located by computing c_x and c_y from b_{x} and b_{y}.

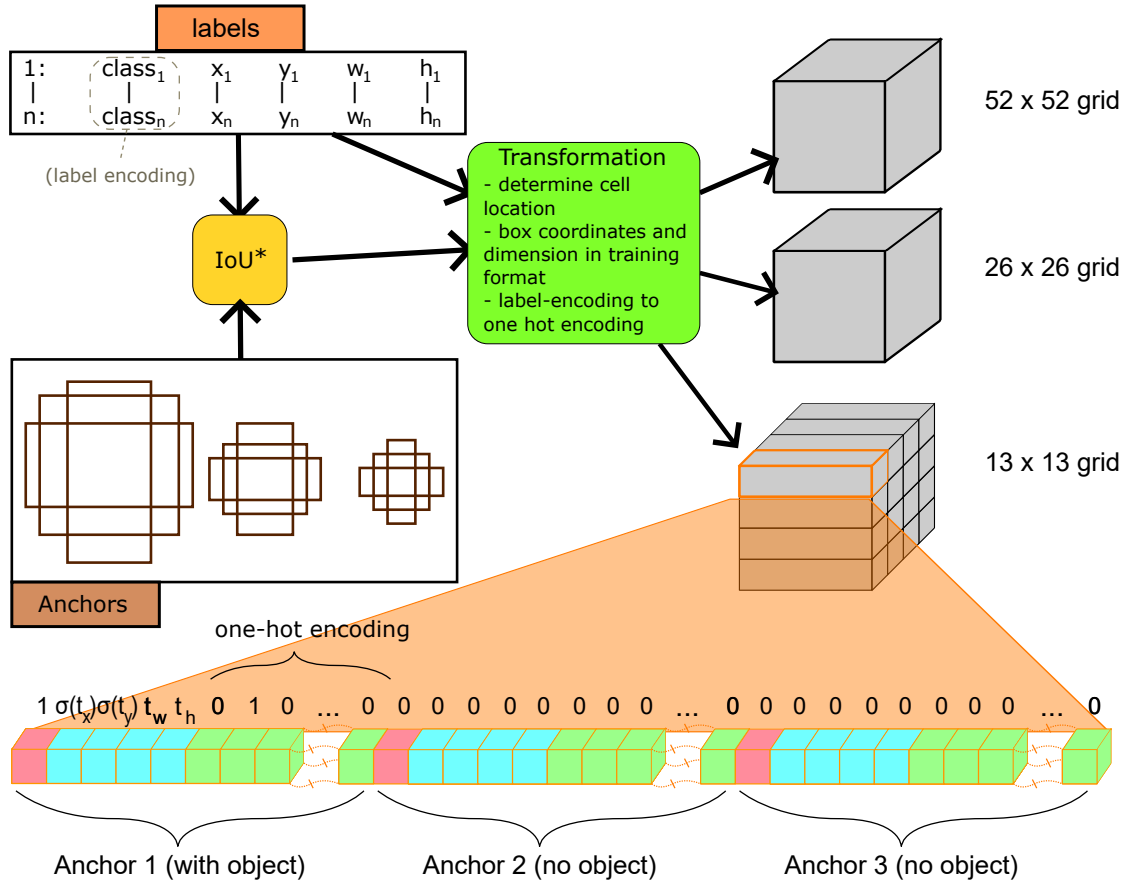

Figure 2.14: In the first step the grid and the anchor section are determined through the calculation of a ground-truth box with all anchors using IoU*. In the second step the label values are transformed and stored in the corresponding cell.

In the corresponding Equation (2.9) we can see that b_x must be calculated by adding $\sigma(t_x)$ and c_x. The same holds for b_y with $\sigma(t_y)$ and c_y. $\sigma(t_x)$ and $\sigma(t_y)$ are the shift inside the cell boundaries, which ranges from 0 to 1. c_x and c_y is the cell location and the values are always integer numbers which is used to split b_x and b_y into integer and non-integer number with range $[0, 1)$. As a result, the cell location is obtained with

$$\begin{aligned} c_x &= b_x - \sigma(t_x) \\ c_y &= b_y - \sigma(t_y) \ . \end{aligned} \tag{2.21}$$

To give an example, the label of an image is $[1, 0.8, 0.4, 0.5, 0.5]$. Additionally, the anchors $[0.01, 0.01]$, $[0.02, 0.02]$, $[0.03, 0.03]$, are given for a 12×12 grid, $[0.06, 0.06]$, $[0.08, 0.08]$, $[0.1, 0.1]$

for a 6×6 grid and $[0.3, 0.3]$, $[0.5, 0.5]$, $[0.7, 0.7]$, for a 3×3 grid. The best anchor is computed with IoU* which is the anchor $[0.5, 0.5]$. Therefore, the label information on target level is located in the second section of the 3×3 grid. The center coordinates are transformed to cell coordinates by multiplying with the grid-scale, which results in

$$\begin{aligned} b_\mathrm{x} &= 0.8 \cdot 3 = 2.4 \\ b_\mathrm{y} &= 0.4 \cdot 3 = 1.2 \ . \end{aligned} \tag{2.22}$$

The cell location is obtained by rounding down to integer numbers resulting in $c_\mathrm{x} = 2$ and $c_\mathrm{y} = 1$. This corresponds to the third cell in x-direction and the second cell in y-direction. After that, the location of the target anchor section $\mathbf{tr}_{gij}$ is determined with $g = 1$ for the lowest resolution grid, $i = 3 + 3 = 6$ by going row wise from left to right and $j = 2$, which corresponds the second anchor of a grid.

The next step is to compute the elements of every cell. In general a target anchor section is defined as

$$\mathbf{tr}_{gij} = [p_{\mathrm{conf},gij}, \sigma(t_{\mathrm{x},gij}), \sigma(t_{\mathrm{y},gij}), t_{\mathrm{w},gij}, t_{\mathrm{h},gij}, p_{\mathrm{class},1,gij}, p_{\mathrm{class},2,gij}, .., p_{\mathrm{class},C,gij}] \in \mathbb{R}^{d/B_g}, \tag{2.23}$$

where $p_{\mathrm{conf},gij}$ is the true confidence and is either 0 for no object or 1, if an object is located in that cell section. If it is 0, the values of the remaining elements do not matters. Otherwise, $\sigma(t_{\mathrm{x},gij})$ and $\sigma(t_{\mathrm{y},gij})$ are the box center location relative to the cell in cell coordinates. They are calculated with

$$\begin{aligned} \sigma(t_\mathrm{x}) &= b_\mathrm{x} - c_x \\ \sigma(t_\mathrm{y}) &= b_\mathrm{x} - c_y \ . \end{aligned} \tag{2.24}$$

$t_{\mathrm{w},gij}$ and $t_{\mathrm{h},gij}$ are the box dimensions transformed to match the direct output of the network. They are determined with

$$\begin{aligned} t_w &= \ln(w / a^*_\mathrm{w}) \\ t_h &= \ln(h / a^*_\mathrm{h}) \ , \end{aligned} \tag{2.25}$$

where a^*_w and a^*_h are the dimensions of the best anchor B^*_anchor. The elements $p_{\mathrm{class},1,gij}$ to $p_{\mathrm{class},C,gij}$ are the elements of the one-hot encoded label. Taking the example from before and setting the number of classes C to be 4, the target anchor section results in

$$\mathbf{tr}_{1,6,2} = [1, 0.4, 0.2, 1.5, 1.5, 0, 1, 0, 0] \ . \tag{2.26}$$

The targets and the transformed outputs are used in the training step to calculate the loss, which is explained in the next section.

2.10 YOLOv3 Loss Function

The loss function computes the error between the predictions and targets which is needed to adjust the weights of the neural network through back-propagation. For every grid $g \in [1, 2, ..., G]$ the loss $\mathcal{L}_g$ is calculated and summed up, which results in

$$\mathcal{L} = \sum_{g=1}^{G} \mathcal{L}_g \ , \tag{2.27}$$

where G is the number of grids and $\mathcal{L}_g$ is the grid loss. The grid loss is calculated with

$$\mathcal{L}_g = \mathcal{L}_{\text{coord},g} + \mathcal{L}_{\text{obj},g} + \mathcal{L}_{\text{noobj},g} + \mathcal{L}_{\text{class},g} \ , \tag{2.28}$$

where $\mathcal{L}_{\text{coord},g}$ is the coordinate loss, $\mathcal{L}_{\text{obj},g}$ the object loss, $\mathcal{L}_{\text{noobj},g}$ the no-object loss and $\mathcal{L}_{\text{class},g}$ the class loss. Before each loss term is explained, *masks* are introduced. A mask is a binary tensor which is used to filter elements of another tensor. This is done by multiplying each element with 0 or 1. Figure 2.15 shows a basic mask operation. One mask used is the *object*

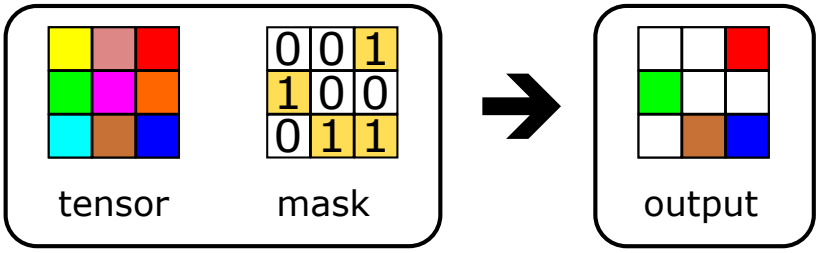

Figure 2.15: A tensor is filtered using a mask.

mask, which is constructed from the target grids. Each target cell has 3 sections for each anchor. Each section stores information of an object that is constructed with the corresponding anchor. If a target section $\mathbf{tr}_{gij}$ has no object, the corresponding prediction section $\mathbf{pr}_{gij}$ is omitted. This is illustrated in Figure 2.16. In contrary the *no-object mask* filters every prediction sections that have no object in the corresponding target section. Figure 2.17 shows how the no-object mask is constructed.

The purpose of these mask is to split the prediction sections $\mathbf{pr}_{gij}$ into sections with and without target object. The loss of sections with target objects is computed differently from

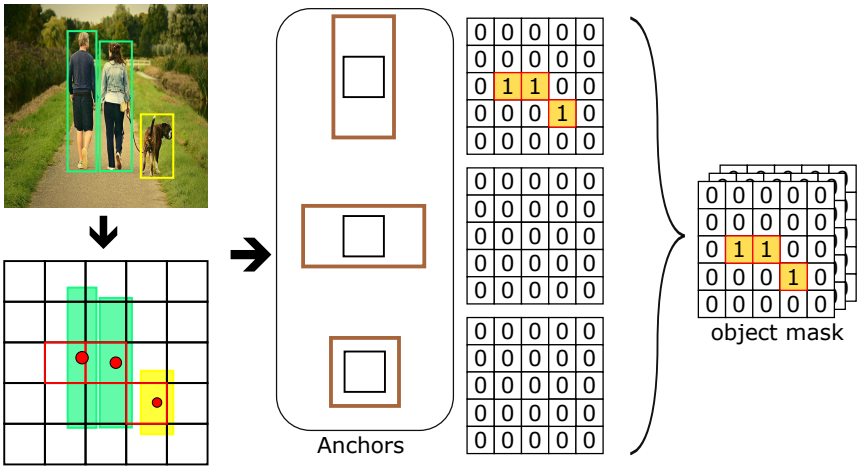

Figure 2.16: The object mask is computed from the box center location and best matching anchor.

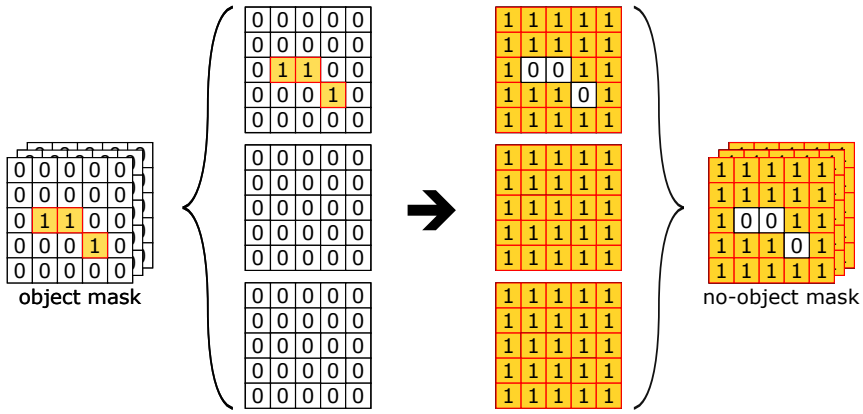

Figure 2.17: The no-object mask is the inverse of the object mask and indicates the cell sections without objects.

those without target objects. First we look at the loss terms which are calculated with the object mask. The object loss is defined as

$$\mathcal{L}_{\text{obj},g} = \lambda_{\text{obj}} \sum_{i=1}^{S_g^2} \sum_{j=1}^{B_g} \mathbb{1}_{gij}^{\text{obj}} \text{BCE}(p_{\text{conf},gij}, \hat{p}_{\text{conf},gij}) \ , \tag{2.29}$$

where λ_{obj} is a weighting factor, S_g^2 is the number of cells in the grid g, B_g is the number of anchors in grid g, $\mathbb{1}_{gij}^{\text{obj}}$ is an element of the object mask, $\text{BCE}(p_{\text{conf},gij}, \hat{p}_{\text{conf},gij})$ is the binary cross-entropy (BCE) with $p_{\text{conf},gij}$ and $\hat{p}_{\text{conf},gij}$ being the confidences of the ground-truth and the prediction [14].

The object loss computes how well the model predicts wether there is an object center in a cell with respect to a specific anchor. It evaluates the prediction confidence $\hat{p}_{\text{conf},gij}$ against the target confidence $p_{\text{conf},gij}$ by computing the BCE, which is calculated with

$$\text{BCE}(p_{\text{conf},gij}, \hat{p}_{\text{conf},gij}) = -p_{\text{conf},cgij} \cdot \log_2(\hat{p}_{\text{conf},cgij}) + (1 - p_{\text{conf},cgij}) \cdot \log_2(\hat{p}_{\text{conf},cgij}) \ , \tag{2.30}$$

First, only the cell sections with target objects are filtered by applying the object mask. Then, the BCE is calculated for each pair of $p_{\text{conf},gij}$ and $\hat{p}_{\text{conf},gij}$. For sections with objects, the corresponding $p_{\text{conf},gij}$ is 1. When $\hat{p}_{\text{conf},gij}$ is close to 1, the BCE approaches 0 and it is large when $\hat{p}_{\text{conf},gij}$ is close to 0. The results are summed up over all filtered cell sections.

Another loss with object mask is the coordinate loss which is defined as

$$\mathcal{L}_{\text{coord},g} = \mathcal{L}_{\text{xy},g} + \mathcal{L}_{\text{wh},g}, \tag{2.31}$$

where $\mathcal{L}_{\text{xy},g}$ and $\mathcal{L}_{\text{wh},g}$ are the center coordinate loss and the box dimension loss. The center coordinate loss is defined as

$$\mathcal{L}_{\text{xy},g} = \lambda_{\text{coord}} \sum_{i=1}^{S^2} \sum_{j=1}^{B_g} \mathbb{1}_{gij}^{obj} ((\sigma(t_{\text{x},gij}) - \sigma(\hat{t}_{\text{x},gij}))^2 + (\sigma(t_{\text{y},gij}) - \sigma(\hat{t}_{\text{y},gij}))^2) \ , \tag{2.32}$$

where λ_{coord} is a weighting factor, $\sigma(t_{\text{x},gij})$ and $\sigma(t_{\text{y},gij})$ are ground-truth center coordinates without the cell location shift, see Equation (2.24). $\sigma(\hat{t}_{\text{x},gij})$ and $\sigma(\hat{t}_{\text{y},gij})$ are the corresponding predicted center coordinates. The error of the center coordinates is calculated with *squared distance*. In analogy, the box dimension loss is defined as

$$\mathcal{L}_{\text{wh},g} = \lambda_{\text{coord}} \sum_{i=1}^{S^2} \sum_{j=1}^{B_g} \mathbb{1}_{gij}^{obj} ((t_{\text{w},gij} - \hat{t}_{\text{w},gij})^2 + (t_{\text{h},gij} - \hat{t}_{\text{h},gij})^2) \ , \tag{2.33}$$

where $t_{\text{w},gij}$ and $t_{\text{h},gij}$ are the ground truth box dimensions in net output format with $\hat{t}_{\text{w},gij}$ and $\hat{t}_{\text{h},gij}$ as the corresponding net outputs. The error is also calculated with squared distance. The coordinate loss is only measured for sections which are filtered by the object mask. Otherwise there would be no target bounding box to which a predicted bounding box is compared to. The center coordinate loss measures how far the predicted box center is from the target box center and the box dimension loss determines how well the shape of the target box is predicted.

The next step is to measure the how well the model predicts the right class label. This is measured with the class loss, which is defined as

$$\mathcal{L}_{\text{class},g} = \lambda_{\text{class}} \sum_{i=1}^{S^2} \sum_{j=1}^{B_g} \mathbb{1}_i^{obj} \text{BCE}(\mathbf{p}_{\text{class},gij}, \hat{\mathbf{p}}_{\text{class},gij}) \ , \tag{2.34}$$

where λ_{class} is a weighting factor, $\mathbf{p}_{\text{class},gij}$ is the vector of the one-hot encoded label and $\hat{\mathbf{p}}_{\text{class},gij}$ is the vector for class probabilities. For multi-label classification the error between these vectors is calculated with the BCE, which is calculated with

$$\text{BCE}(\mathbf{p}_{\text{class},gij}, \hat{\mathbf{p}}_{\text{class},gij}) = -\frac{1}{C} \sum_{c=1}^{C} p_{\text{class},cgij} \cdot \log_2(\hat{p}_{\text{class},cgij}) + (1 - p_{\text{class},cgij}) \cdot \log_2(\hat{p}_{\text{class},cgij}) \ , \tag{2.35}$$

where C is the number of classes in the dataset, $p_{\text{class},cgij}$ is the target class probability and $\hat{p}_{\text{class},cgij}$ is the predicted class probability. For single-label classification, the categorical crossentropy (CCE) is used instead. The vector $\hat{\mathbf{p}}_{\text{class},gij}$ is passed to a softmax function, which normalizes the vector by the sum of its elements. Then, the CCE is calculated with

$$\text{CCE}(\mathbf{p}_{\text{class},gij}, \hat{\mathbf{p}}_{\text{class},gij}) = -\frac{1}{C} \sum_{c=1}^{C} p_{\text{class},cgij} \cdot \log_2(\hat{p}_{\text{class},cgij}) \ . \tag{2.36}$$

For the cell sections without ground-truth objects not all elements in the prediction tensor contribute to the loss. The values of the box center and box dimension predictions as well as the class predictions are ignored. Only the confidence values of the prediction cell are compared to the ground-truth. This is measured with the no-object loss, which is defined as

$$\mathcal{L}_{\text{noobj},g} = \lambda_{\text{noobj}} \sum_{i=1}^{S_g^2} \sum_{j=1}^{B_g} \mathbb{1}_{gij}^{\text{noobj}} \mathbb{1}_{gij}^{\text{ignore}} \text{BCE}(p_{\text{conf},gij} \hat{p}_{\text{conf},gij}) \ , \tag{2.37}$$

where λ_{noobj} is a weighting factor and $\mathbb{1}_{gij}^{\text{noobj}}$ are the elements of the no-object mask. $\mathbb{1}_{gij}^{\text{ignore}}$ is the *ignore mask*. It neglects cell sections whose bounding box predictions are close to a ground-truth box. Figure 2.18 illustrates how the ignore mask is contructed. In this example the red box is the bounding box of the object. The two blue boxes and the purple box are predicted bounding boxes. The blue boxes are generated with the best anchor B_{anchor}^* and the purple box is generated an anchor from different anchor section. The center of one blue box is predicted from a different cell. Because all three predicted bounding boxes overlap with the ground truth box by an $IoU > 0.5$, the corresponding cell sections are ignored in the no-object loss. Without the ignore mask the loss function is too restrictive, because high confidence

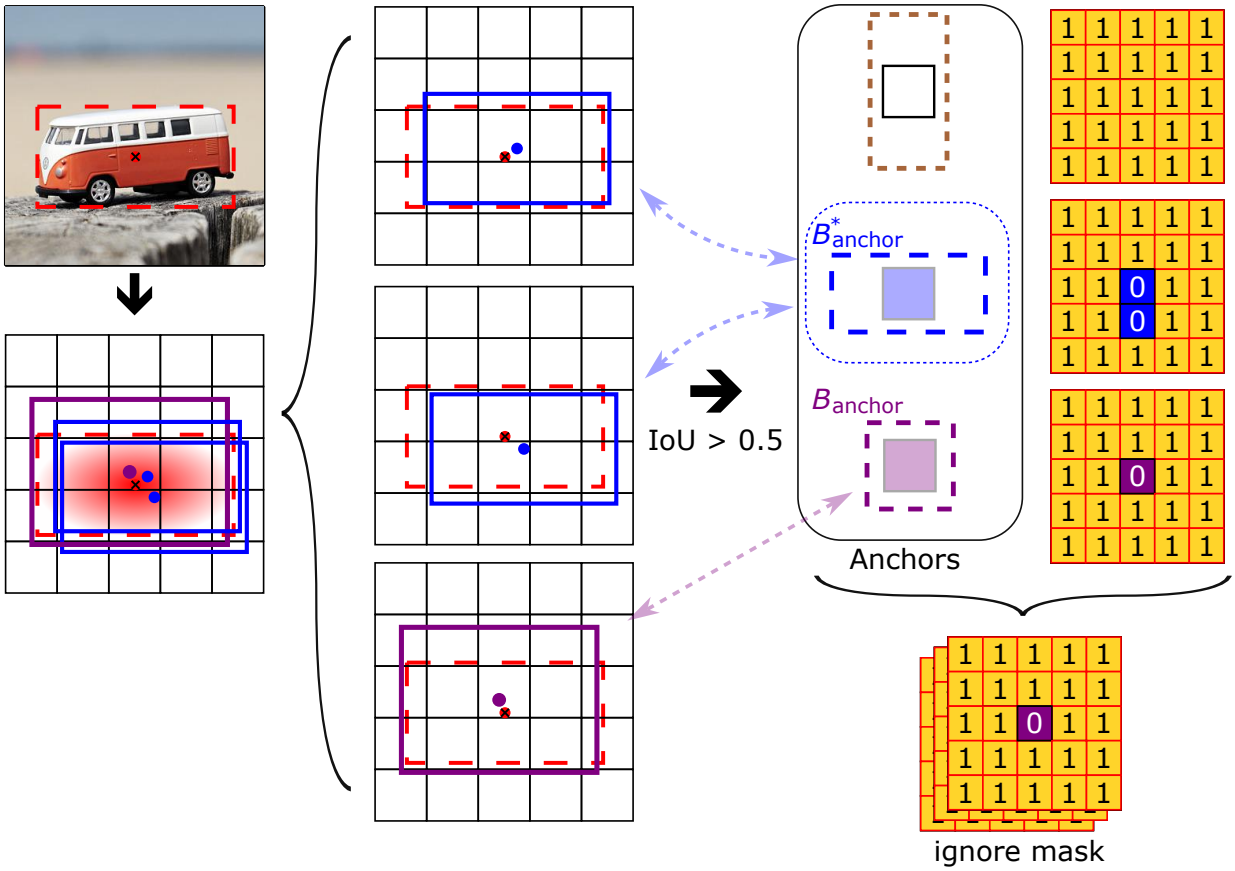

Figure 2.18: The figure shows bounding box predictions with 1 grid. The red box is the ground-truth bounding box. The blue bounding boxes are predictions with the best anchor B^*_{anchor}. The green bounding box is a prediction with a different anchor. The IoUs of the predictions with the ground-truth are calculated. If an IoU is bigger than the threshold 0.5, the corresponding anchor section of a cell is ignored. The result is the ignore mask on the right.

values in cells without objects result in a high loss. With the ignore mask, cells that are close to a ground-truth object are able to predict accurate bounding boxes. The same is true for suboptimal anchors which are not the best anchor. The ignore mask allows these imperfect predictions as long as the IoU is bigger than a certain threshold.

In summary, the full loss function is defined as

$$\begin{aligned}\mathcal{L} = &\sum_{g=1}^{G}\sum_{i=1}^{S_g^2}\sum_{j=1}^{B_g}(\lambda_{\text{obj}}\mathbb{1}_{gij}^{\text{obj}}\text{BCE}(p_{\text{conf},gij},\hat{p}_{\text{conf},gij})... \\ &+\lambda_{\text{coord}}\mathbb{1}_{gij}^{obj}((\sigma(t_{\text{x},gij})-\sigma(\hat{t}_{\text{x},gij}))^2+(\sigma(t_{\text{y},gij})-\sigma(\hat{t}_{\text{y},gij}))^2)... \\ &+\lambda_{\text{coord}}\mathbb{1}_{gij}^{obj}((t_{\text{w},gij}-\hat{t}_{\text{w},gij})^2+(t_{\text{h},gij}-\hat{t}_{\text{h},gij})^2)... \\ &+\lambda_{\text{class}}\mathbb{1}_{i}^{obj}\text{BCE}(\mathbf{p}_{\text{class},gij},\hat{\mathbf{p}}_{\text{class},gij})... \\ &+\lambda_{\text{noobj}}\mathbb{1}_{gij}^{\text{noobj}}\mathbb{1}_{gij}^{\text{ignore}}\text{BCE}(p_{\text{conf},gij},\hat{p}_{\text{conf},gij}))\ ,\end{aligned} \tag{2.38}$$

where λ_{obj}, λ_{coord}, λ_{class} and λ_{noobj} are weighting factors for the individual loss terms. In the original YOLOv3 implementation all weights are 1.

The transformed outputs of the network are many predictions with all kinds of confidence and class probability values. In addition some bounding box predictions are detecting the same object. A filtering algorithm needs to be applied to get a single prediction for each object. The algorithm which does that is *non maximum suppression*. This is explained in the next section.

2.11 Non maximum suppression

After the training step, the neural network is able to generate new outputs for new images. These are transformed based on Equation (2.14), which results in a prediction for every cell and anchor. The predictions include bounding box estimations with low confidence and class probability values. Furthermore, some predictions detect the same object. Therefore, a filtering algorithm is used which is called non maximum suppression (NMS). NMS is illustrated in Figure 2.19. It works in two steps. First the confidence and highest class probability for each bounding box prediction are multiplied with each other which results in a score. Prediction scores that are lower than the score-threshold $p_{\text{score}}^{\text{thresh}}$ are omitted. The threshold is typically 0.5. Second, the remaining bounding box predictions are sorted by their score and by the predicted class. All predictions with the same class are compared to pairwise with each other using IoU starting from the prediction with the highest score. If the IoU is greater then the IoU-threshold $\text{IoU}^{\text{thresh}}$, which is typically 0.5, the two predictions detect the same object. Only one prediction per object is needed, therefore the prediction with the lower score is omitted. The result are a single bounding box and a class prediction with a high score for each detected object.

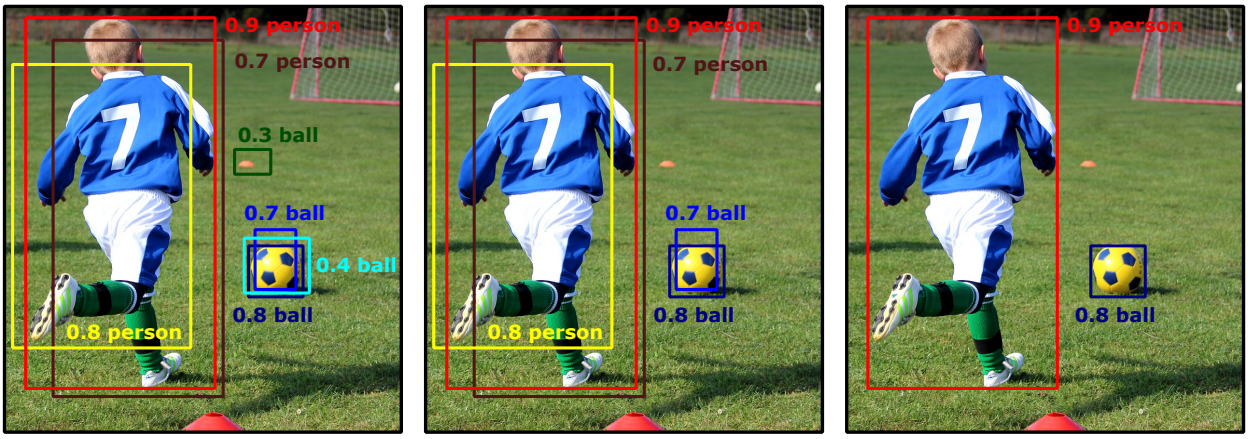

Figure 2.19: On the left are unfiltered predictions for a person and a ball. These predictions are filtered by a score threshold, which results in the predictions shown in the middle. After that, the IoU between all bounding box with the same class prediction are calculated. If two boxes predictions have an IoU greater than 0.5, the one with the lower score is omitted.

2.12 Pre-training of the feature extractor

Training the whole YOLOv3 network from scratch is difficult for datasets with complicated patterns and objects; for instance, persons and animals. Instead, part of the network uses weights of a pre-trained image classifier. The feature extractor of the YOLOv3 is based of the darknet-53 image classifier (see Table 2.3) . In the original YOLO implementation the darknet-53 classifier was trained on the *ImageNet database* [21]. It consists of images that are annotated according to the WordNet hierarchy, which is a lexical database. The image classifier predicts classes as probabilities, which is carried out by a pooling layer, a dense layer and a softmax activation layer. The weights of the trained image classifier are loaded to the feature extractor of the YOLOv3 network. The weights act as a starting point for the feature extractor and are trained with the weights of the detection head in the main training step. Here the model is able to train on the dataset for object detection more effectively. Figure 2.20 illustrates this process.

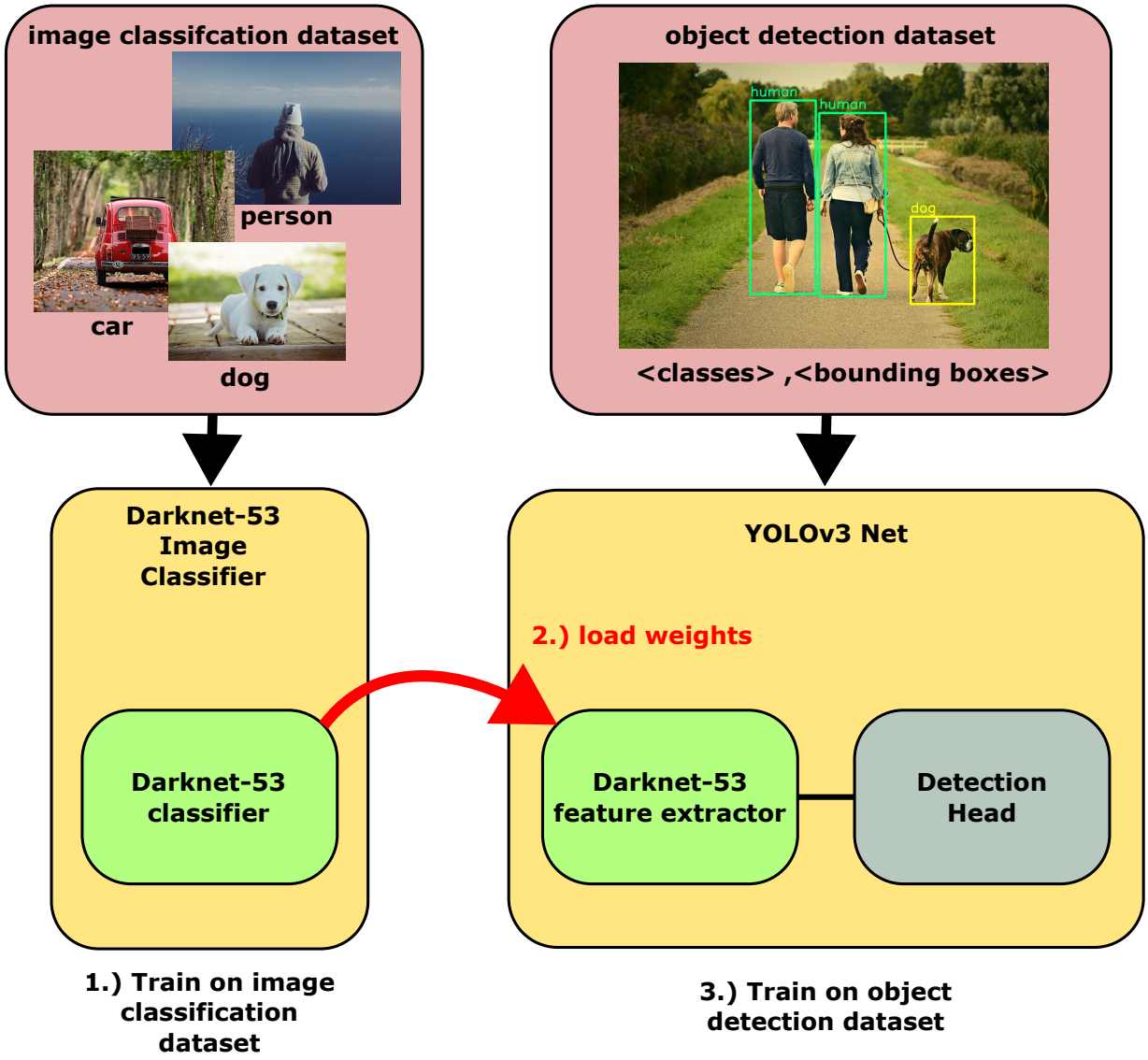

Figure 2.20: First an object classifier which uses darknet-53 is trained on an image classification dataset. The weights of the darknet-53 are loaded into the corresponding darknet-53 network in the YOLOv3 model. Then the YOLOv3 network is trained on the object detection dataset.

3 Implementation of a Noise Detector

A YOLOv3 model is implemented to detect noise objects on sound dataset. The sound dataset was created by recording a braking test with a microphone, which are then transformed to images with *short time Fourier Transformation* [24].The original YOLOv3 implementation was written in C++ [18]. The noise detector is implemented with Python. The model is build with the Tensorflow framework using the high-level API *keras* and trained on a Nvidia GeForce TITAN Xp 12GB graphics card. The implementation is inspired by the open source models [16, 28, 10, 27, 2]. Parts of the YOLOv3 implementation are shown in the following sections. In addition different hyperparameters and training strategies are implemented.

3.1 Tensorflow

Tensorflow is a an open-source framework for machine learning and artificial intelligence. It was developed by the Google Brain team and released in November 2015. It supports different programming languages, such as Python, Java, C++ and Javascript. Tensorflow allows the programmer to build deep learning algorithms by using symbolic math. Big datasets are processed using dataflow graphs [1]. An example for a simple dataflow graph is given in Figure 3.1. The edges are tensors, which are multidimensional data arrays. The nodes are mathematical operations. A tensorflow implementation for the example can be seen in the code snippet 3.1.

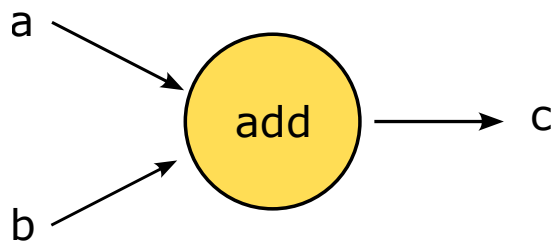

Figure 3.1: The edges are tensors and the node is a mathematical operation.

```
1 import tensorflow as tf
2
3 a = tf.constant([[1,3,2],[4,7,0],[2,1,3]])
4 b = tf.constant([[4,1,0],[4,2,0],[2,1,1]])
5 c = tf.math.add(a,b)
```

Listing 3.1: Example for creating two tensors and adding them

3.2 Keras

Keras is a deep learning API for implementing neural networks using Python. It is designed to make neural network programming easier by using a high-level of abstraction. It supports different frameworks, which are Tensorflow, Theano, PlaidML, MXNet and CNTK. Keras has become the official high-level API for tensorflow. There are two APIs for implementing models with Keras. The first is the *sequential API*. It is appropiate for networks with a plain stack of layers. The downside is that it can take only one input and has only one output. The second is the *functional API*, which enables the use of multiple inputs and multiple outputs [5]. An example is shown in snippet 3.2. Since the YOLOv3 network uses residual skips and outputs multiple tensors, the functional API is used for the implementation of the noise detector.

```
1 import keras
2 import keras.layers
3
4 inputs = keras.Input(shape=(784,))
5 x = layers.Dense(64, activation="relu")(x)
6 x = layers.Dense(64, activation="relu")(x)
7 outputs = layers.Dense(10)(x)
8 model = keras.Model(inputs=inputs, outputs=outputs)
```

Listing 3.2: Example for a simple network implemented with Keras functional API

3.3 Dataset

The dataset contains spectrograms as images and annotations for objects on the spectrogram. These were generated in previous work by applying short time Fourier transform on the sound

recordings [24]. The images have pixel dimensions 384×256. The x-axis represents the time, the y-axis the frequencies and the color of a pixel indicates the amplitude. The frequencies range from 1 to 16 kHz. There are four object classes, which are squeals, wirebrushes, clicks and artefacts. An image for each object class is shown in Figure 3.2. Squeals are tonal frequency sounds that range from 1 to 16 kHz. They have long durations and their frequency band is narrow. The opposites are clicks, which span the complete frequency range and are very short sound events. Wirebrushes are a combination of many short-time sounds. Noise artefacts have high amplitudes for a broad frequency range. Each object is assigned only to one class. Therefore, a single-label classification task is given.

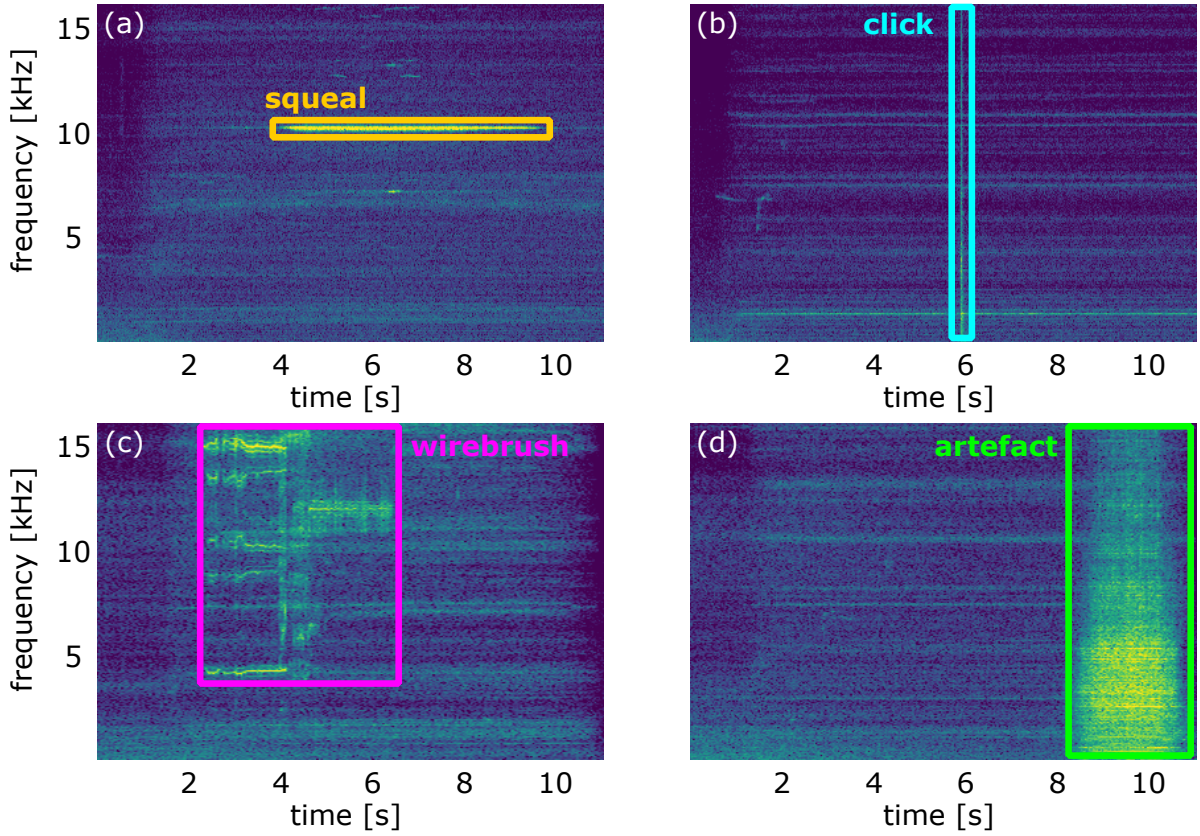

Figure 3.2: Four noise objects are shown, where (a) is squeal, (b) is a click, (c) a wirebrush and (d) is an artefact.

Each object has a bounding box in x_1,y_1,x_2,y_2-format and is assigned to a class by label encoding. In total there are $3,476$ images and annotations for the objects. The dataset is split into 3 sets. Two sets are used in the training step which are the training set with 2387 and the validation set with 599 images and annotation files. This corresponds to a 80-20 split. The third set is the test set which contains only images with objects of the same class and the

corresponding annotations. These are 200 images for squeals, 50 images for wirebrush noises, 25 images for click sounds and 15 images for artefacts. Additionally, 200 images without any noise object are added to the test set. Table 3.1 describes the data split in more detail. The annotations are stored in xml-files.

Table 3.1: Noise dataset is split into the number of single-class images and multi-class images. For instance, training dataset has 1764 images with squeal noises only. In addition, the number of noise objects for each class is shown for the datasets.

	Image class			Number object per class		
Category	Training	Validation	Test	Training	Validation	Test
squeal	1764	442	200	3257	780	302
wirebrush	231	58	50	403	118	58
click	143	36	25	405	109	49
artefact	7	2	15	68	11	15
multi-class	242	61	0	-	-	-
silent	0	0	200	-	-	-

3.4 Image Preprocessing

As input the YOLOv3 network expects images, which have quadratic pixel dimensions and are divisible by 32. For this an input shape of (416×416) is chosen. This is done by applying the resize function from the tensorflow library. Then the pixel values are normalized from $[0, 255]$ to $[0, 1]$. The function that does both is shown in the code snippet 3.3.

```
1 import tensorflow as tf
2
3 def transform_images(x_train, size):
4     x_train = tf.image.resize(x_train, (size, size))
5     x_train = x_train / 255
6     return x_train
```

Listing 3.3: Transforming images

3.5 Target Preparation

The annotation for each image is transformed to the target shape, which is needed for the loss calculation. In the code snippet 3.4 the best anchor for each label is determined using broadcast IoU, which is stored in the anchor idx and concatenated with the labels. In the for-loop the labels for an image are transformed into target shape, which results into three 3D-tensors, where the classes are label encoded. This is different than the target shape introduced in Section 2.9, where the labels are one-hot encoded. The advantage to one-hot encoding is that label encoding uses less memory.

```
1  def transform_targets(y_train, anchors, anchor_masks, size):
2      y_outs = []
3      grid_size = size // 32
4
5      # calculate anchor index for true boxes
6      anchors = tf.cast(anchors, tf.float32)
7      anchor_area = anchors[..., 0] * anchors[..., 1]
8      box_wh = y_train[..., 2:4] - y_train[..., 0:2]
9      box_wh = tf.tile(tf.expand_dims(box_wh, -2),
10                      (1, 1, tf.shape(anchors)[0], 1))
11     box_area = box_wh[..., 0] * box_wh[..., 1]
12     intersection = tf.minimum(box_wh[..., 0], anchors[..., 0]) * \
13         tf.minimum(box_wh[..., 1], anchors[..., 1])
14     iou = intersection / (box_area + anchor_area - intersection)
15     anchor_idx = tf.cast(tf.argmax(iou, axis=-1), tf.float32)
16     anchor_idx = tf.expand_dims(anchor_idx, axis=-1)
17
18     y_train = tf.concat([y_train, anchor_idx], axis=-1)
19
20     for anchor_idxs in anchor_masks:
21         y_outs.append(transform_targets_for_output(
22             y_train, grid_size, anchor_idxs))
23         grid_size *= 2
24
25     return tuple(y_outs)
```

Listing 3.4: Tranforming targets

3.6 Anchors

Two different anchor sets are used for the model. One is the anchor set of the original YOLOv3 implementation [21], which was generated from the COCO-dataset with k-means-clustering. These are shown in the snippet 3.5.

```
ANCHORS = [
    [(0.28, 0.22), (0.38, 0.48), (0.9, 0.78)],  # 13x13 grid
    [(0.07, 0.15), (0.15, 0.11), (0.14, 0.29)], # 26x26 grid
    [(0.02, 0.03), (0.04, 0.07), (0.08, 0.06)], # 52x52 grid
]
```

Listing 3.5: Original anchors obtained from COCO

The other set of anchors is created by plotting the bounding boxes of the training dataset in a scatter plot which is shown in Figure 3.3, and then hand picking 9 anchor boxes. Each point on the plot denotes an object. The position describes the bounding box dimensions and the color indicates the class. The anchors were positioned so that each class has is represented by at least two anchor boxes. The anchor values are shown in the snippet 3.6.

```
ANCHORS = [
    [(0.40, 0.93), (0.90, 0.45), (0.70, 0.93)], # 13x13 grid
    [(0.08, 0.60), (0.08, 0.80), (0.50, 0.40)], # 26x26 grid
    [(0.20, 0.08), (0.50, 0.12), (0.75, 0.08)], # 52x52 grid
]
```

Listing 3.6: Noise Object Anchors

3.7 Neural Architecture

The neural network uses different blocks, that are made of different layers. These are convolutional blocks, residual units and residual blocks. In addition there are two up-sampling layers and two concatenate operations. The neural network is implemented in < model.py >.

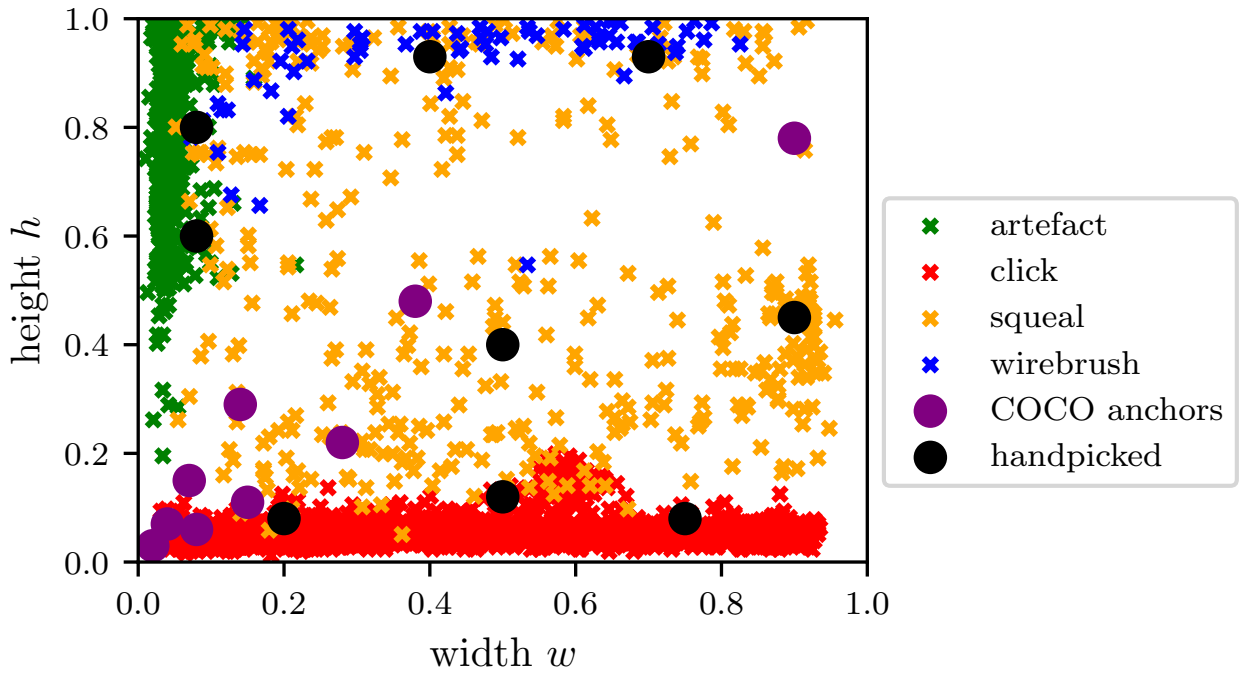

Figure 3.3: The bounding boxes of the training dataset as scatter plot. Each color indicates a different object class. The purple points are the original YOLOv3 anchors. The black points are hand-picked anchors.

3.7.1 Convolutional Block

The implementation of the convolutional block is shown in the code snippet 3.7. A convolutional block takes the input tensor <x> and processes it through multiple functions. It consists of a convolutional layer, where the number of filters is set with <filter_number>. It uses valid padding for stride 1 and top-left zero padding for stride 2. In all convolutional blocks a batch normalization layer and a leaky ReLU activation function follow the convolutional layer, except for the outputs of the network.

```
1 from keras.layers import (
2     BatchNormalization, Conv2D, ZeroPadding2D, LeakyReLU
3     )
4
5 def conv_block(x, filter_number, kernel_size, stride=1, \
6                                     batch_norm=True):
7     # padding = 'same'
8     if stride == 1:
9         padding = 'same'
```

```
10      else:
11          x = ZeroPadding2D(((1, 0), (1, 0)))(x)  # top left padding
12          padding = 'valid'
13      x = Conv2D(filters=filter_number,
14                 kernel_size=(kernel_size,kernel_size),
15                 strides=(stride,stride),
16                 padding = padding,
17                 use_bias=not batch_norm)(x)
18      if batch_norm is True:
19          x = BatchNormalization()(x)
20          x = LeakyReLU(alpha=0.1)(x)
21      return x
```

Listing 3.7: Convolutional Block

3.7.2 Residual Block

A residual block consists of at least one residual unit, which has a convolutional block with a (1×1)-filter, a convolutional block with a (3×3)-filter and a skip connection. The number of filters for each convolutional block is determined with <filter_list>. Multiple residual units are chained by a certain number of times, which is specified by <repeats>.

```
1  from keras.layers import add
2
3  def res_block(x, filter_list, repeats):
4      for i in range(repeats):
5          res_skip = x
6          x = conv_block(x, filter_list[0], 1)
7          x = conv_block(x, filter_list[1], 3)
8          x = add([res_skip, x])
9      return x
```

Listing 3.8: Residual Block

3.7.3 Feature Extractor

The feature extractor is the darknet-53 classifier without the pooling layer, dense layer and activation function. The code snippet 3.9 shows the implementation of building the feature extractor. The resulting darknet-53 model takes RGB-images as input and has three outputs, which are three tensors with different resolutions.

```
1  from keras import Model
2  from keras.layers import Input
3
4  def darknet(name):
5
6      x = inputs = Input([None, None, 3])
7      x = conv_block(inputs, 32, 3)
8      x = conv_block(x, 64, 3, 2)
9      x = res_block(x, [32, 64], 1)
10
11     x = conv_block(x, 128, 3, 2)
12     x = res_block(x, [64, 128], 2)
13
14     x = conv_block(x, 256, 3, 2)
15     x = x_36 = res_block(x, [128, 256], 8)
16
17     x = conv_block(x, 512, 3, 2)
18     x = x_61 = res_block(x, [256, 512], 8)
19
20     x = conv_block(x, 1024, 3, 2)
21     x = res_block(x, [512, 1024], 4)
22     return tf.keras.Model(inputs, [x_36, x_61, x], \
23                           name=name)
```

Listing 3.9: Darknet-53 Feature Extractor

3.7.4 Neck

The neck is part of the detection head and can be split into three segments. Each segment has as stack of five convolutional blocks. The convolutional blocks are implemented with the code snippet 3.10.

```
# Format for the arguments of filters_numbers, kernel_sizes, strides
# [32,64,...], [1,3,...], [1,2,...] <- all same length
def conv_block_2(x, filter_numbers, kernel_sizes, strides):

    for i, filter_number in enumerate(filter_numbers):
        kernel_size = kernel_sizes[i]
        stride = strides[i]
        x = conv_block(x, filter_number, kernel_size, stride)
    return x
```

Listing 3.10: Stack of convolution blocks used for the outputs of the feature extractor

Depending on the segment it can have one input tensor or two input tensors with different dimensions, for instance, $13 \times 13 \times d$ and $26 \times 26 \times d$. The additional input comes from the output of another segment (see Figure 2.9). In that case, an up-sampling layer is used to increase the dimensions of the smaller input tensor. Then both tensors are concatenated. The code snippet 3.11 shows the implementation of a segment.

```
from keras import UpSampling2D, Concatenate

def yolo_conv(x_in, filters, kernels, strides, filters_up, name):
    if isinstance(x_in, tuple):
        inputs = Input(x_in[0].shape[1:]), Input(x_in[1].shape[1:])
        x, x_skip = inputs

        # concat with skip connection
        x = conv_block(x, filters_up, 1, 1)
        x = UpSampling2D(2)(x)
        x = Concatenate()([x, x_skip])
    else:
        x = inputs = Input(x_in.shape[1:])

```

```
15      x = conv_block_2(x, filters, kernels, strides)
16      return tf.keras.Model(inputs, x, name=name)(x_in)
```

Listing 3.11: Implementation of a neck segment

3.7.5 Output Layers

Each of the three outputs from the neck are passed to two convolutional blocks, which are the output layers. The output layers are implemented with the code snippet 3.12. The last convolutional block uses no batch normalization and no leaky ReLU activation. For convenience, the output tensors are reshaped, so that a grid cell is a 2D-tensor with the shape $((1+4+C) \times B_g)$.

```
1  from keras.layers import Lambda
2
3  def yolo_output(x_in, filters, anchors, num_classes, name):
4      x = inputs = Input(x_in.shape[1:])
5      x = conv_block(x, filters, 3, 1)
6      x = conv_block(x, (5+num_classes)*3, 1, 1, False)
7      x = Lambda(lambda x: \
8                 tf.reshape(x, (-1, tf.shape(x)[1], tf.shape(x)[2], \
9                                anchors, num_classes + 5)))(x)
10     return tf.keras.Model(inputs, x, name=name)(x_in)
```

Listing 3.12: Output layers

3.7.6 YOLOv3 Network

The code snippet 3.13 shows the implementation of the complete YOLOv3 network. It builds a darknet-53 feature extractor, which is then connected with the detection head. The detection head consists of three < yolo_conv(...) > functions, are connected to each other according to Figure 2.9. < yolo_output(...) > defines the last two convolutional blocks.

```
1
2  def build_model(num_classes, shape=(416, 416, 3)):
3      x = inputs = Input(shape=shape, name='input')
4      #Darknet-53
5      x_36, x_61, x = darknet('yolo_darknet')(x)
```

```
6     # Detection Head, 13 x 13 Output
7     x = yolo_conv(x, [512, 1024, 512, 1024, 512, ],
8                       [1,3,1,3,1],
9                       [1,1,1,1,1],
10                      None,
11                      name='yolo_conv_0')
12    out0 = yolo_output(x, 1024, 3, num_classes, name='yolo_output_0')
13    # 26 x 26 Output
14    x = yolo_conv((x, x_61), [256, 512, 256, 512, 256, ],
15                      [1,3,1,3,1],
16                      [1,1,1,1,1],
17                      256,
18                      name='yolo_conv_1')
19    out1 = yolo_output(x, 512, 3, num_classes, name='yolo_output_1')
20    # 52 x 52 Output
21    x = yolo_conv((x,x_36), [128, 256, 128, 256, 128, ],
22                      [1,3,1,3,1],
23                      [1,1,1,1,1],
24                      128,
25                      name='yolo_conv_2')
26    out2 = yolo_output(x, 256, 3, num_classes, name='yolo_output_2')
27    outputs = [out0, out1, out2]
28    return tf.keras.Model(inputs=inputs, outputs=outputs, name='darknet_53')
```

Listing 3.13: Building the YOLOv3 Network

A network is then build with < build_model(num_classes=4) > and it can be plotted with < tf.keras.utils.plot_model(model)> which is shown in Figure 3.4.

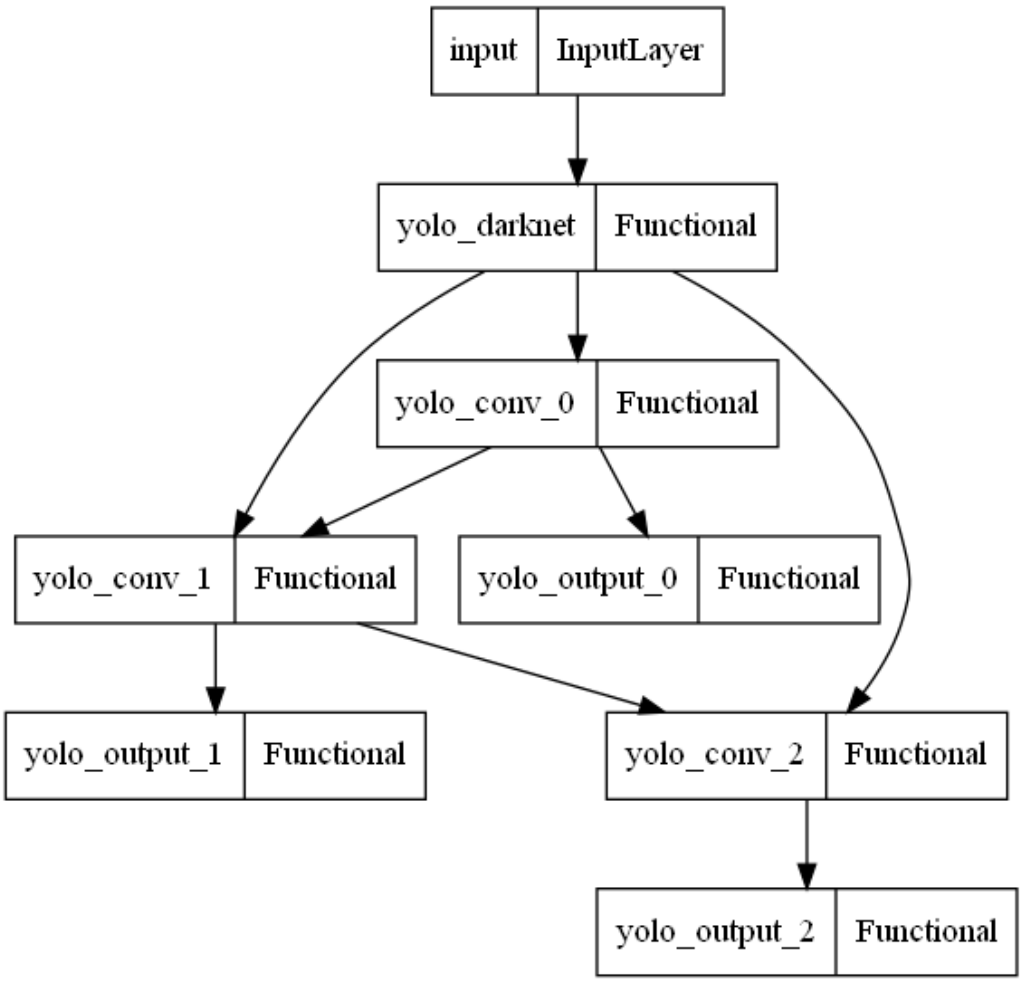

Figure 3.4: Plot of the YOLOv3 Network

3.8 Loss function

The transformations of the outputs are implemented together with the loss function. It is implemented as < yoloLoss(...) > in < model.py >. The classification loss is implemented with *sparse categorical crossentropy* (SCCE), because single-label classification is given. The SCCE takes label encoded classes as argument. The code snippet 3.14 shows an example loss function which contains only the object loss and the corresponding transformations. For each of the three grid outputs a loss function with the corresponding anchors is created, which is then passed to the training step.

```
1 import tensorflow as tf
2 from tensorflow.keras.losses import (
3     BinaryCrossentropy, SparseCategoricalCrossentropy
4 )
```

```
bce = BinaryCrossentropy(reduction=tf.keras.losses.Reduction.NONE)
scce = SparseCategoricalCrossentropy( \
        reduction=tf.keras.losses.Reduction.NONE)
# Object loss only
def yoloLoss(anchors, weights, num_classes):
    def Lossfunction(y_true, y_pred):
        # conf at [4]
        true_obj = y_true[...,4]
        true_obj = tf.expand_dims(true_obj,-1)

        pred_obj = tf.sigmoid(y_pred[...,4])
        pred_obj = tf.expand_dims(pred_obj,-1)

        # obj_mask
        obj_mask = tf.squeeze(true_obj, -1) #

        # obj_loss + noobj_loss
        obj_loss_all = bce(true_obj, pred_obj)

        #============================
        #...other transformations...
        #============================

        obj_loss = obj_mask * obj_loss_all

        # sum over (batch, gridx, gridy, anchors) => (batch, 1)
        xy_loss = ...
        wh_loss = ...
        obj_loss = tf.reduce_sum(obj_loss, axis=(1, 2, 3))
        noobj_loss = ...
        class_loss = ...

        total_loss =  weights[0] * xy_loss + \
                      weights[1] * wh_loss + \
                      weights[2] * obj_loss + \
                      weights[3] * noobj_loss +\
```

```
41                          weights[4] * class_loss
42
43          return total_loss
44      return Lossfunction
```

Listing 3.14: YOLOv3 Loss function with object loss only

3.9 Training

The code snippet 3.15 creates and trains a YOLOv3 model. The hyperparameters are loaded from config.py. Weights from the original YOLOv3 model which was trained on the COCO dataset is converted into tensorflow format with <convert.py>. These are used for loading the feature extractor of the noise detector.

```
1  import config
2  img_size = config.IMG_SIZE # 416, multiple of 32
3  grid_list = config.GRID_LIST # [13,26,52]
4  num_classes = config.NUM_CLASSES # 4 for noise data
5  weights = config.WEIGHTS #
6  anchors = config.ANCHORS #
7  epochs=config.EPOCHS
8  batch_size=config.BATCH_SIZE
9  learning_rate = config.LEARNING_RATE
10 file_name = f'yolov3_scratch_bs{batch_size}_'
11
12 from load_datasets import get_sound_datasets
13 train_dataset, val_dataset = get_sound_datasets()
14
15 from model import build_model, yoloLoss
16 import tensorflow as tf
17 yolo = build_model(num_classes)
18 loss_function = [yoloLoss(anchors[0], weights, num_classes),
19                  yoloLoss(anchors[1], weights, num_classes),
20                  yoloLoss(anchors[2], weights, num_classes)]
21 yolo.compile(optimizer=tf.keras.optimizers.Adam(  \
22
```

```
23                 loss=loss_function)
24 from tensorflow.keras.callbacks import (
25     ReduceLROnPlateau, EarlyStopping, ModelCheckpoint)
26 callbacks = [ReduceLROnPlateau(verbose=1),
27                 ModelCheckpoint('checkpoints/'+file_name+'.tf',
28                                 verbose=1, save_weights_only=True),]
29
30 yolo.fit(train_dataset, batch_size=batch_size, epochs=epochs,
31                       validation_data=val_dataset,
32                       )
```

Listing 3.15: Creating and training a YOLOv3 model

3.10 Predictions on Image Level

After training, the model is used in the prediction step. For this the transformation from network output to image level is implemented as shown in the code snippet 3.16. It takes a tensor from the raw output of the network and converts the elements. Therefore it converts the raw values of the (13×13), (26×26) and (52×52) output grid into prediction values. The results are tensors for center coordinates and box dimensions in image coordinates and tensors for confidence values and class probabilities. These tensors are then reshaped in the format for the non maximum suppression function, which is shown in the code snippet 3.17.

```
1  def yoloBoxes(output, anchors, num_classes):
2      grid_size = tf.shape(output)[1:3]
3
4      bboxes_xy, bboxes_wh, conf, p_class = tf.split(
5          output, (2,2,1,num_classes), axis = -1)
6
7      bboxes_xy = tf.sigmoid(bboxes_xy)
8      conf = tf.sigmoid(conf)
9      p_class = tf.sigmoid(p_class)
10
11     grid = _meshgrid(grid_size[1],grid_size[0])
12                 # [gy, gx, 1, 2]
13     grid = tf.expand_dims(tf.stack(grid, axis=-1), axis=2)
```

```
14
15
16      bboxes_xy = (bboxes_xy + tf.cast(grid, tf.float32)) / \
17          tf.cast(grid_size, tf.float32)
18      bboxes_wh = tf.exp(bboxes_wh) * anchors
19
20      bboxes_x1y1 = bboxes_xy - bboxes_wh / 2
21      bboxes_x2y2 = bboxes_xy + bboxes_wh / 2
22
23      bboxes = tf.concat([bboxes_x1y1, bboxes_x2y2], -1)
24
25      return bboxes, conf, p_class
```

Listing 3.16: Transformation of raw network output to bounding box and class predictions

```
1  def reshapeYoloBoxes(outputs):
2      b,c,p = [],[],[]
3      for o in outputs:
4          b.append(tf.reshape(o[0], (tf.shape(o[0])[0], \
5                                     -1, tf.shape(o[0])[-1])))
6          c.append(tf.reshape(o[1], (tf.shape(o[1])[0], \
7                                     -1, tf.shape(o[1])[-1])))
8          p.append(tf.reshape(o[2], (tf.shape(o[2])[0], \
9                                     -1, tf.shape(o[2])[-1])))
10     b = tf.concat(b,1)
11     c = tf.concat(c,1)
12     p = tf.concat(p,1)
13     return b,c,p
```

Listing 3.17: Transforming prediction tensor to lists of bounding box coordinates (<b>), confidence values (<c>) and class probabilities (<p>)

3.11 Non Maximum Suppression

The code snippet 3.18 shows a non maximum suppression implementation for batch size 1. It contains the < reshapeYoloBoxes(outputs) > implementation. It calculates the score and then

uses the tensorflow implementation for NMS. It takes the unfiltered predictions as input. The thresholds $\text{iou}^{\text{thresh}}$ and $p_{\text{score}}^{\text{thresh}}$ are determined with $<\text{iou}_{\text{t}}\text{hreshold}>$ and $<\text{score}_{\text{t}}\text{hreshold}>$.

```
def _nms(outputs, num_classes, max_output_size, iou_threshold, \
        score_threshold, soft_nms_sigma):
    bbox,conf,prob_class = reshapeYoloBoxes(outputs)

                scores = conf * prob_class
    dscores = tf.squeeze(scores, axis=0) # Squeeze for batch size 1
    scores = tf.reduce_max(dscores,[1])
    bbox = tf.reshape(bbox,(-1,4))
    classes = tf.argmax(dscores,1)

    selected_indices, selected_scores = \
                tf.image.non_max_suppression_with_scores(
                        boxes=bbox,
                        scores=scores,
                        max_output_size=max_output_size,
                        iou_threshold=iou_threshold,
                        score_threshold=score_threshold,
                        soft_nms_sigma=soft_nms_sigma
                )
    num_valid_nms_boxes = tf.shape(selected_indices)[0]
    boxes=tf.gather(bbox, selected_indices)
    boxes = tf.expand_dims(boxes, axis=0)
    scores=selected_scores
    scores = tf.expand_dims(scores, axis=0)
    classes = tf.gather(classes,selected_indices)
    classes = tf.expand_dims(classes, axis=0)
    valid_detections=num_valid_nms_boxes
    valid_detections = tf.expand_dims(valid_detections, axis=0)
    return boxes, scores, classes, valid_detections
```

Listing 3.18: Non maximum suppression for batch size 1

3.12 Prediction

The image <noise.png> is loaded to the working directory and loaded with <plt.imread(file)> to the python environment. Then, it is reshaped and passed to model network. The raw outputs are transformed for the NMS function. The final predictions are stored in the lists for bounding boxes (<boxes>), scores (<scores>) and (<classes>).

```
import matplotlib.pyplot as plt
import numpy as np
import cv2
from yolo_utils import yoloBoxes, _nms
file = r'C:/Users/Lucky/yolov3/yolov3-keras/noise.png'
img = plt.imread(file)
img = cv2.resize(input_img, (416,416), interpolation = cv2.INTER_AREA)
img = img / np.max(img)
img = np.expand_dims(img, axis = 0)

output = yolo(input_img)
y1 = yoloBoxes(output[0], anchors[0], num_classes)
y2 = yoloBoxes(output[1], anchors[1], num_classes)
y3 = yoloBoxes(output[2], anchors[2], num_classes)
outputs=(y1,y2,y3)

boxes, scores, classes, valid_detections = _nms(outputs, \
      num_classes, \
      max_output_size=100, \
      iou_threshold=0.5, \
      score_threshold=0.5, \
      soft_nms_sigma=0.)
```

Listing 3.19: Predicting bounding boxes and class

3.13 Hyperparameters

The hyperparameters are set in < config.py >. Table 3.2 shows the hyperparameters used in training. In addition a validation set is used to measure the validation error.

Table 3.2: Hyperparameters

category	value
image size	416×416
learning rate	0.001
epochs	30
batch size	16
optimizer	Adam
λ_{obj}	1
λ_{coord}	1
λ_{class}	1
λ_{noobj}	1

3.14 Training Strategies

Multiple YOLOv3 models are implemented and trained with different training strategies.

In the first setup the original anchor boxes are used and the feature extractor is loaded with weights from the original YOLOv3 implementation which was trained on the COCO dataset (model 1). It attempts to give the model a head start for training. Since the objects of the COCO dataset and the noise dataset might be too different, two models are trained without pre-loading weights. The first model trains with the original anchors which were obtained from the COCO dataset (model 2). The second model trains with anchors hand-picked for the noise dataset (model 3). Table 3.3 summarizes the different setups.

Table 3.3: Model Setups

model	weights	anchors
1	pre-loaded	original (COCO)
2	scratch	original (COCO)
3	scratch	hand-picked

4 Results

The three implemented YOLOv3 models (see Table 3.3) are tested in two tests. The tests have similar implementation to [24]. The first test evaluates the models in an image classification task. The second test is an object detection task, where the models predict the location and class of noise objects. For these, test accuracy metrics are needed. If a model predicts a positive class correctly, then the result is a true positive (TP). If it predicts a positive class incorrectly, then the result is a false positive (FP). Similarly, if a model predicts a negative class correctly, the result is a true negative (TN) and the result is a false negative, if it predicts the negative class incorrectly. For all tests, the number of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) are determined. Furthermore, other metrics are calculated, which are recall, precision, F_1-score, average precision (AP) and mean average precision (mAP). Precision, recall and F_1-score are defined as

$$\begin{aligned} \text{precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}} \\ \text{recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}} \\ F_1 &= 2 \cdot \frac{\text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}} \ . \end{aligned} \tag{4.1}$$

Precision measures how many of the predicted positives are actual positives. Recall measures how many of the actual positives are predicted correctly as positive. The F_1-score measures the accuracy using precision and recall. To determine the AP, the precision-recall curve for each class is determined, which is approximated in both tests by varying the score threshold for the NMS. The score thresholds is varied from 0 to 1 in steps of 0.1. Then, the AP for a class c is defined as the area under the precision-recall curve. For the class c, AP is defined as

$$\text{AP}_c = \sum_n (\text{recall}_{c,n} - \text{recall}_{c,n-1}) \text{precision}_{c,n} \ , \tag{4.2}$$

where $\text{precision}_{c,n}$ and $\text{recall}_{c,n}$ are the precision and recall value at the n-th threshold for class c. The mAP is defined as

$$\text{mAP} = \frac{1}{C} \sum_c \text{AP}_c \ , \tag{4.3}$$

where C is the number of classes.

4.1 Model Convergence

Figure 4.1 shows the trainings histories for each model. Figure 4.1a is the loss history of the model 1, Figure 4.1b shows the loss history of the model 2 and Figure 4.1c shows the loss history of the model 3. Training takes about 40 minutes for 30 epochs with a Nvidia GeForce TITAN Xp 12GB graphics card, which is about 80 seconds per epoch. Model 1 starts to converge after 30 epochs. The minimum validation loss is approximately 20 for all models. For all models the weights at the epoch with the lowest validation loss are saved. This is for model 1 at 13 epochs, for model 2 at 14 epochs and for model 3 at 19 epochs. In comparison, a YOLOv3 model with COCO weights for the feature extractor does not reach an optimum after 30 epochs. However, with frozen pre-loaded feature extractor weights the model reaches its optimum after 4 epochs (See Appendix A.1). The real-time prediction speed (batch size 1) is about 1.09 seconds (0.92 FPS) on a notebook with Mobile Gfx 2.30 GHz (CPU) and is about 0.088 seconds (11,36 FPS) for a Nvidia GeForce TITAN Xp 12GB graphics card.

The models for noise detections are tested in the following tests.

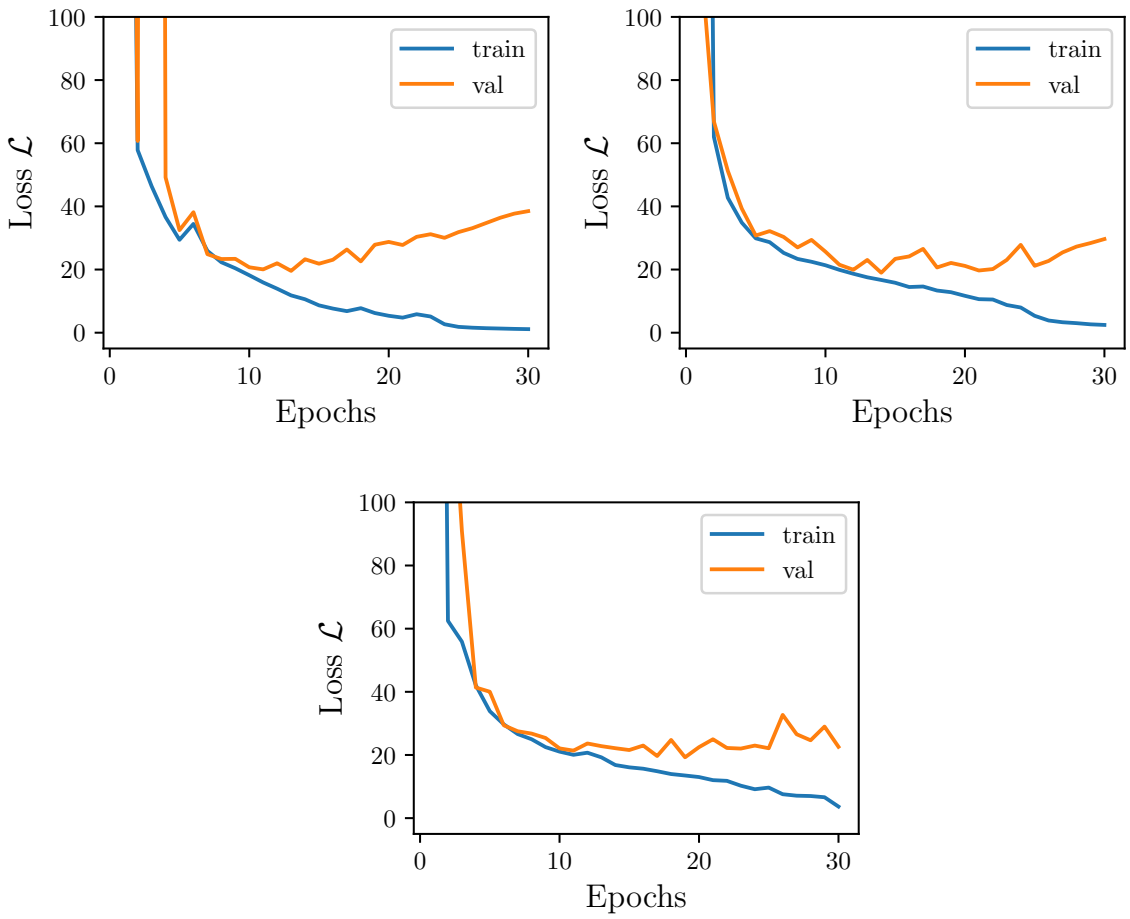

Figure 4.1: Loss history for each model

4.2 Image Classification Test

Each model is tested in an image classification task. It measures how well a model predicts wether there is a noise object of a certain class in an image. The test set contains images which has objects of a single class only. These are squeals, wirebrushes, clicks and artefact sounds. The predicted image class label is compared to the true class label. Take for example, an image which has a squeal noise. Then, the image label is [squeal: 1, wirebrush: 0, click: 0, artefact: 0]. The model predictions are squeals and clicks. Therefore the predicted image label is [squeal: 1, wirebrush: 0, click: 1, artefact: 0]. The squeal class is predicted correctly (TP). The model predicts falsely a click *no-click* (FP) and predicts correctly no wirebrush and no artefact (TN). If the example image has a wirebrush instead of squeal noise, the result would be a FN.

Table 4.1 shows the results of the classification test where the score-threshold and IoU-threshold are set to $p_{\text{score}}^{\text{thresh}} = 0.5$ and $\text{IoU}^{\text{thresh}} = 0.5$. Recall, precision and F_1-score are determined. The

APs are computed by determining the precision-recall curve first. For this, the $p_{\text{score}}^{\text{thresh}}$ is varied from 0 to 1 with steps of 0.1 and the corresponding precision and recall for each class is calculated. Then, the APs are approximated with (4.2) and the mAP is determined. All three models have high accuracies for squeal images, but model 2 performs slightly better. Model 1 has the highest accuracy for wirebrush images. Model 2 performs on click image significantly better than the other two models. Model 1 performs well on artefact images, while model 2 has low accuracy on artefact images and model 3 does not predict artefacts. Overall, Model 1 performs well on all classes and has the highest mAP.

Similar results can be seen on the precision-recall curves which are shown in Figure 4.2. In general, there is a trade-off between precision and recall. Therefore, accurate detectors have high precision for high recall, which means that their precision-recall curve is close to the top right corner. Figure 4.2a shows high performance in predicting squeal images for all models. Figure 4.2b and Figure 4.2d show that model 1 more accurate on wirebrush and artefact images than the other models. Figure 4.2c shows that Model 2 performs on click noise images the best.

Table 4.1: Classification results for model 1 (pre-loaded, original anchors), the model 2 (scratch, original anchors) and model 3 (scratch, hand-picked anchors)

model	category	TP	FP	FN	TN	recall	precision	f1	AP	mAP
1	squeal	198	72	2	218	0.99	0.73	0.84	0.96	0.86
	wirebrush	24	2	26	438	0.48	0.92	0.63	0.84	
	click	10	3	15	462	0.4	0.77	0.53	0.77	
	artefact	11	3	4	472	0.73	0.79	0.76	0.86	
2	squeal	198	34	2	256	0.99	0.85	0.92	0.98	0.68
	wirebrush	23	14	27	426	0.46	0.62	0.53	0.66	
	click	18	2	7	463	0.72	0.9	0.8	0.89	
	artefact	3	3	12	472	0.2	0.5	0.29	0.17	
3	squeal	194	29	6	261	0.97	0.87	0.92	0.96	0.54
	wirebrush	10	3	40	437	0.2	0.77	0.32	0.54	
	click	4	0	21	465	0.16	1.0	0.28	0.66	
	artefact	0	0	15	475	0.0	None	None	0.0	

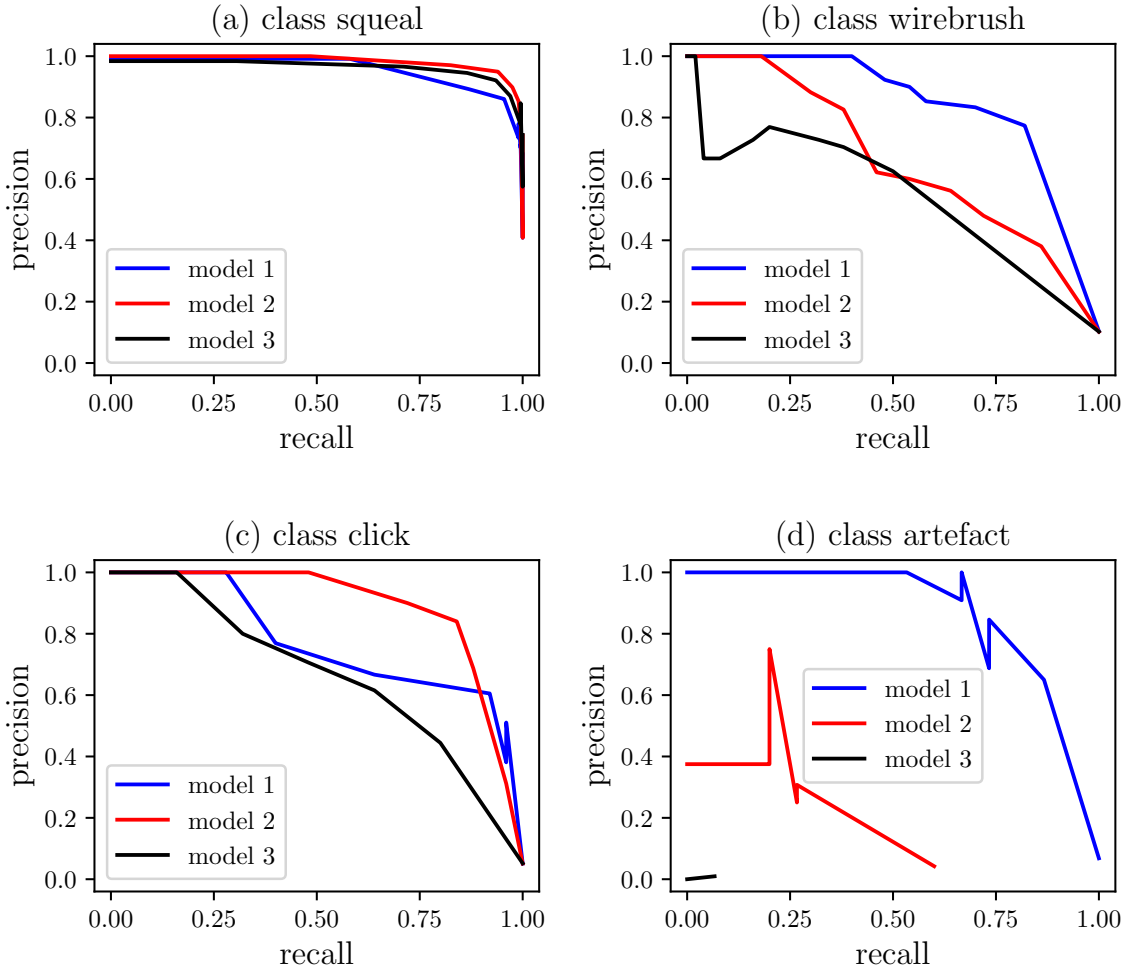

Figure 4.2: Precision-recall curves for the classification test

4.3 Object Detection Test

The models are tested in locating and classifying noise objects. For this TP, FP and FN are determined. Figure 4.3 illustrates the detections of the model 1 for six different images. It detects the squeal noises and the wirebrush correctly for Figure 4.3a and Figure 4.3d (TP). In Figure 4.3b and in Figure 4.3c, the wirebrushes and clicks are not detected (FN) and it falsely predicts a squeal (FP) for Figure 4.3c. Figure 4.3e shows an image with multiple noise objects which are detected correctly. Figure 4.3f is a noiseless image.

Each model is tested on the test set (see Table 3.1), which contains only single-class images. AP and mAP are computed for different IoU-thresholds which are $\text{IoU}^{\text{thresh}} = 0.5$, $\text{IoU}^{\text{thresh}} = 0.75$ and $\text{IoU}^{\text{thresh}} = 0.9$. A higher IoU-threshold indicates a stricter accuracy metric. These are same thresholds as in [24]. The results are shown in Table 4.2. The model 2 has the highest

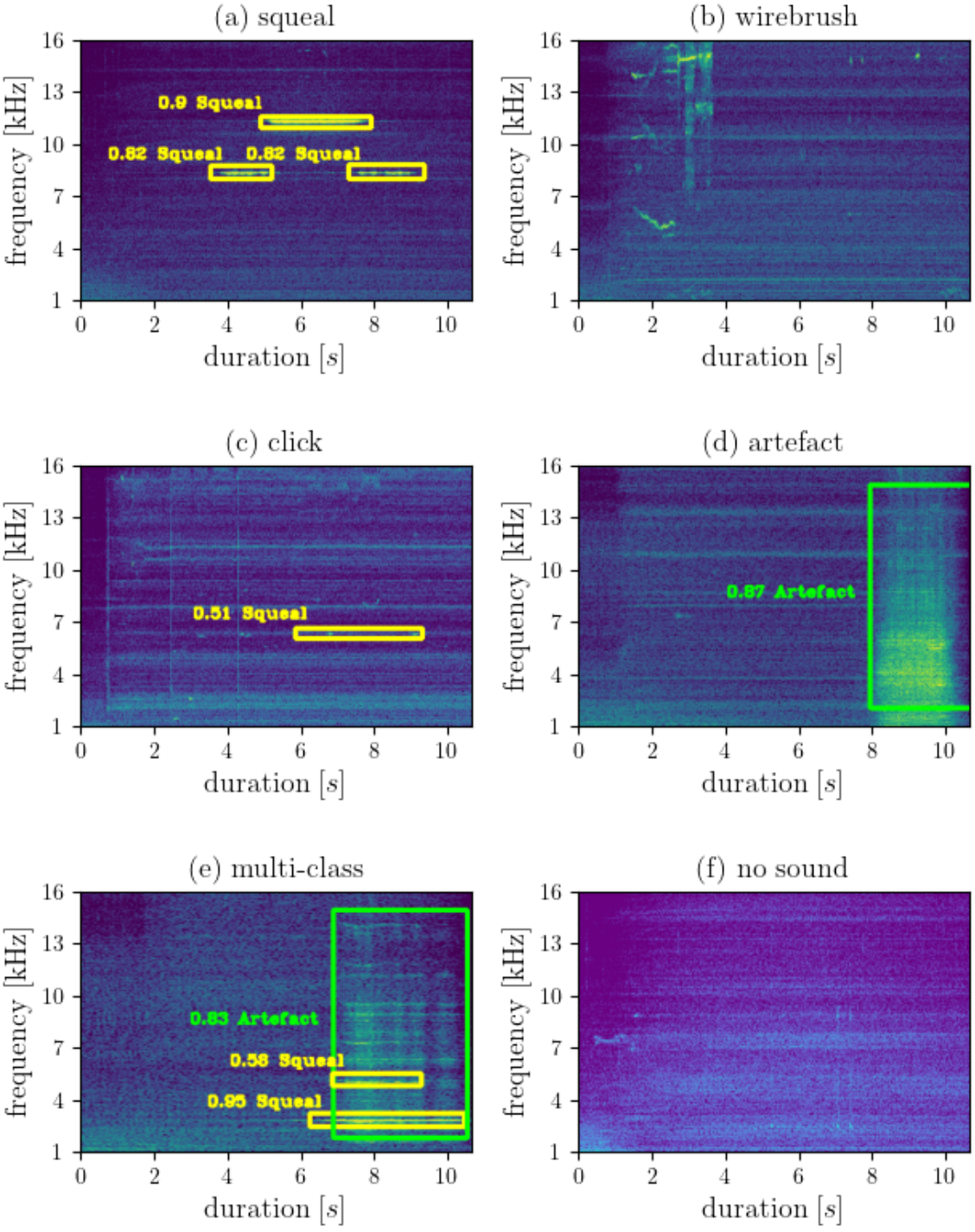

Figure 4.3: Model 1 detections for squeal image (a), wirebrush image (b), click image (c), artefact image (d), multi-class image (e), noiseless image (f)

APs for squeal noises and model 3 has the lowest. Model 1 and model 2 have similar accuracy on wirebrushes. All three models have low accuracy for click noises. Only model 1 has a high accuracy for artefacts. Model 3 does not predict any artefact noises. Model 1 has the highest mAP, followed by model 2.

Figure 4.4 shows the precision-recall curves for each class. The curves for each class are similar, except for Figure 4.4d. Here, the accuracy of model 1 is significantly higher. The model 3 has no precision-recall curve for artefact noises.

Table 4.2: Object detection results for model 1 (pre-loaded, original anchors), the model 2 (scratch, original anchors) and model 3 (scratch, hand-picked anchors)

model	category	$AP_{0.50}$	$AP_{0.75}$	$AP_{0.90}$	$mAP_{0.50}$	$mAP_{0.75}$	$mAP_{0.90}$
1	squeal	0.53	0.46	0.4	0.46	0.36	0.3
	wirebrush	0.42	0.22	0.2			
	click	0.2	0.21	0.16			
	artefact	0.7	0.53	0.44			
2	squeal	0.54	0.54	0.48	0.33	0.28	0.22
	wirebrush	0.41	0.24	0.16			
	click	0.27	0.27	0.2			
	artefact	0.1	0.08	0.05			
3	squeal	0.48	0.45	0.34	0.24	0.21	0.14
	wirebrush	0.24	0.16	0.1			
	click	0.26	0.22	0.11			
	artefact	0.0	0.0	0.0			

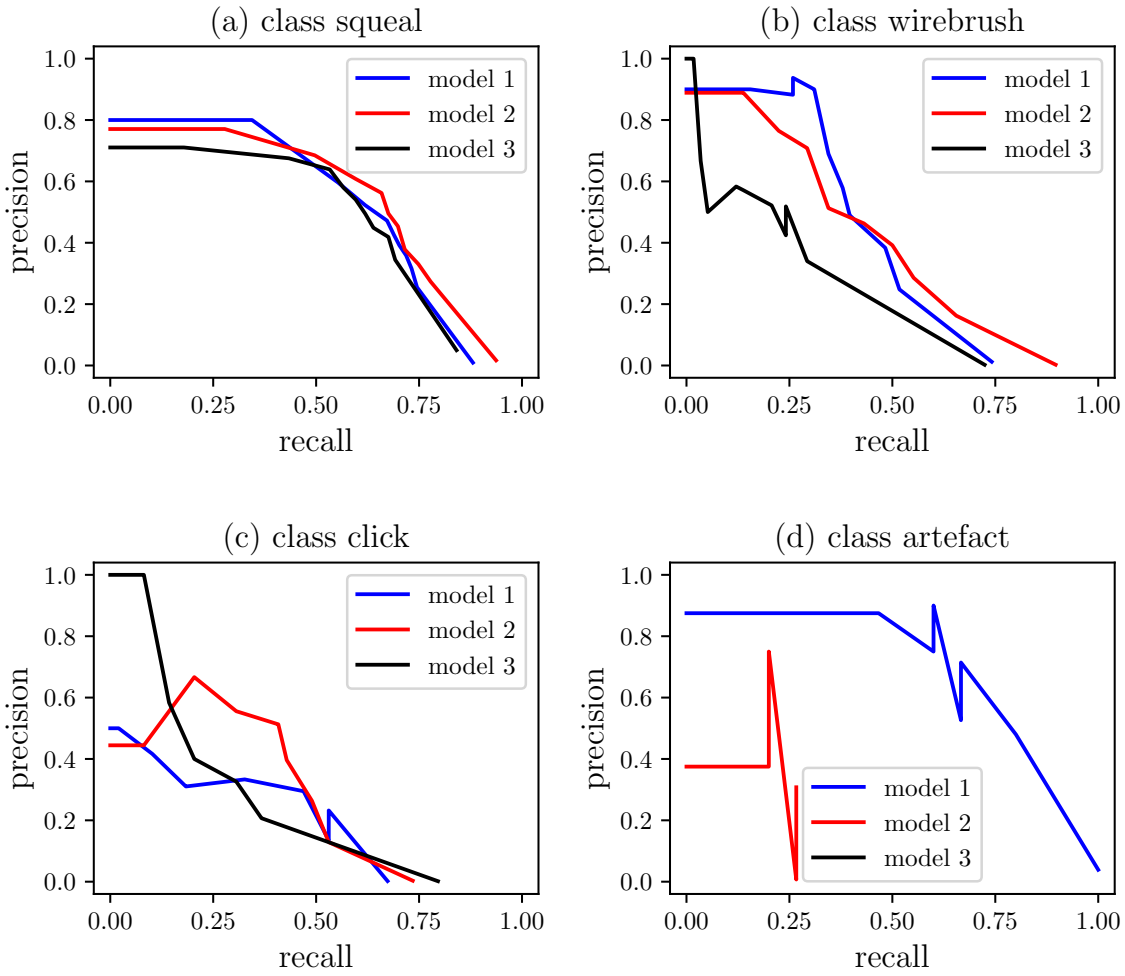

Figure 4.4: Precision-recall curves for object detection test with $\text{IoU}^{\text{thresh}} = 0.5$

5 Discussion

The results indicate that a YOLOv3 noise detector is able to perform well on the noise image classification task. Model 1 has better AP for squeal, wirebrush and artefact images in comparison to the two Faster R-CNN models and the R-FCN model [24], which are two-stage detectors. Model 2 has the highest accuracy for click images and is closely followed by the Faster R-CNN inception architecture. Model 1 has the highest mAP with 0.86, which is slightly better than the highest mAP of the models in [24]. However, the more relevant results are measured in the object detection test. In addition to the classification performance, the ability to determine the frequency band and the duration of noises is evaluated.

The object detection results show worse accuracy for all YOLOv3 models compared to the Faster R-CNN models. The YOLOv3 model with the highest mAPs (model 1) has similar $mAP_{0.50}$ to the R-FCN model, which is the model with the worst mAPs from [24]. Model 1 is outperformed in $mAP_{0.75}$ and $mAP_{0.90}$, which means that the bounding boxes which are predicted by the YOLOv3 models overlap less with the ground-truth boxes in comparison to the two-stage detectors. Therefore, predictions for the frequency band and the duration of noise objects are less accurate. Training time of the YOLOv3 models are significantly faster with 40 minutes for 30 epochs in comparison to a full day for 10^6 epochs [24]. The YOLOv3 models perform better for squeal noises compared to other noises. This is likely due to the class imbalance in the training dataset between squeals (3257) and the other objects (881). This creates a bias towards squeal noises (see Table 3.1). Furthermore it is recommended to have at least 2000 images for each object class in the training dataset [3]. The noise training dataset contains 1764 squeal images and only 231 wirebrush images, 143 click images, 7 artefact images and 242 multi-class images. All models perform significantly bad on clicks, which was unexpected because of the simplicity of the object shape. This might be because the bounding box labels for clicks are not accurate enough. The true width of click objects is only one pixel. Therefore, clicks should be relabeled with narrower bounding boxes. In addition, data-augmentation could be used to increase the amount of data and compensate the class imbalance by creating copies from the training dataset and modifying them.

Different YOLOv3 setups have a big impact on the model performance. Pre-loading is a favorable training strategy. The model trains faster and detects complex noise objects (wirebrush, artefact) more accurately, even if the feature extractor weights are loaded from a YOLOv3

model which was trained on a complete different dataset (COCO). Different weights from models that were trained on more similar datasets or pre-training the feature extractor on an image classification dataset are training options that could be studied. Hand-picked anchors did not improve object detection. Using k-means-clustering instead of hand-picking could improve accuracy. Future work should also include studying the newer YOLO models [4, 9].

A difficulty on this project was the lack of detailed explanations behind the theory of YOLO. *Redmon et al.* provided minimal explanation, blog posts had to read with caution and open source implementation are documented sparsely.

6 Summary

YOLOv3 as an noise detector was introduced and the theory of YOLOv3 algorithm was explained. The network structure, the transformations for the output, the loss function and non maximum suppressed were detailed. YOLOv3 models were implemented as noise object detectors. Keras API was used with the Tensorflow framework. Different training strategies were introduced and used for training. One model was trained with pre-loaded feature extractor weights and original anchors. Two models were trained from scratch with one having original anchors and the other model having hand-picked anchors. The models were trained on noise data which are labeled spectrogram images. Then, the models were evaluated on an image classification task and an object detection task. The accuracy on the image classification task is high for all models. For the object detection task, all models are able to detect noise objects, however the YOLOv3 model with pre-loaded weights in the feature extractor achieves the highest accuracy. In comparison, the YOLOv3 models have lower accuracy to previously investigated noise detector models, but they train and predict significantly faster. The implemented models do not fulfil the requirements for a noise object detector. Hand-picking anchors did not improve the performance and k-means-clustering is suggested. Pre-loading weights or pre-training the feature extractor should be included in future training strategies. The main problem that influenced the accuracy is the dataset. The noise dataset has class imbalance towards squeal noises and it is too small for effective training. Click noises are can be labeled more accurately. Data augmentation should be considered to increase the dataset. Improvements from newer YOLO models should also be included in future work.

A Appendix

A.1 Pascal VOC training

For comparison of model convergence two YOLOv3 models were trained on pascal VOC dataset with different setups:

- loading COCO weights into feature extractor
- loading COCO weights into feature extractor and freeze it

The training histories for 30 epochs are shown in Figure A.1.

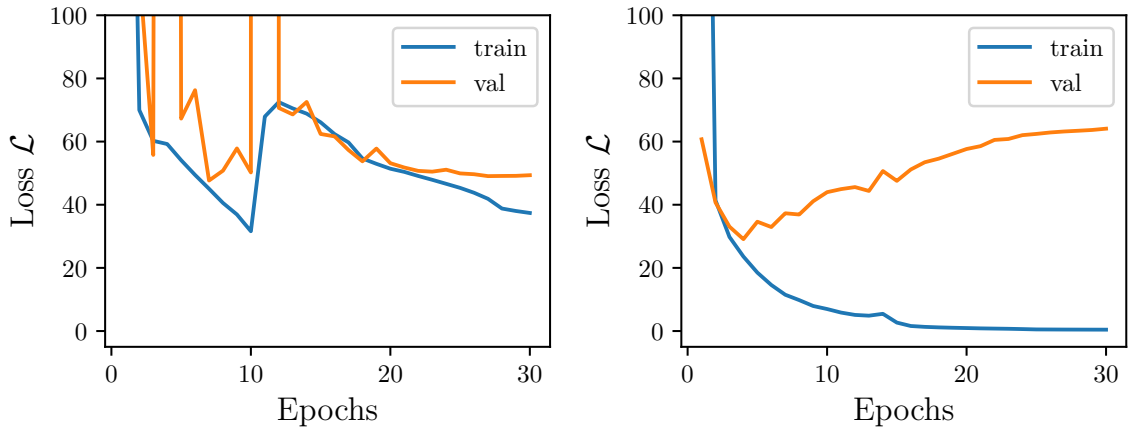

Figure A.1: Loss history for pascal VOC: left shows model with unfrozen darknet weights and right shows model with frozen weights

Bibliography

[1] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dandelion ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. `https://www.tensorflow.org/`. Version: 2015. – Software available from tensorflow.org

[2] Anh, Huynh N.: *keras-yolo3*. `https://github.com/experiencor/keras-yolo3`, 2018

[3] Bochkovskiy, Alexey: *darknet*. `https://github.com/AlexeyAB/darknet`, 2022

[4] Bochkovskiy, Alexey ; Wang, Chien-Yao ; Liao, Hong-Yuan M.: Yolov4: Optimal speed and accuracy of object detection. In: *arXiv preprint arXiv:2004.10934* (2020)

[5] Chollet, François u. a.: *Keras*. `https://keras.io`, 2015

[6] Dai, Jifeng ; Li, Yi ; He, Kaiming ; Sun, Jian: R-fcn: Object detection via region-based fully convolutional networks. In: *Advances in neural information processing systems* 29 (2016)

[7] Girshick, Ross: Fast r-cnn. In: *Proceedings of the IEEE international conference on computer vision*, 2015, S. 1440–1448

[8] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, S. 770–778

[9] Jocher, Glenn: *darknet*. `https://github.com/ultralytics/yolov5`, 2020

[10] Li, Ethan Y.: *YOLO*. `https://github.com/ethanyanjiali/deep-vision/tree/527bb3da655ac6245568942e252e27c61d0b4ca2/YOLO`, 2019

[11] Lin, Min ; Chen, Qiang ; Yan, Shuicheng: Network in network. In: *arXiv preprint arXiv:1312.4400* (2013)

[12] Lin, Tsung-Yi ; Doll'ar, Piotr ; Girshick, Ross ; He, Kaiming ; Hariharan, Bharath ; Belongie, Serge: Feature pyramid networks for object detection. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, S. 2117–2125

[13] Liu, Wei ; Anguelov, Dragomir ; Erhan, Dumitru ; Szegedy, Christian ; Reed, Scott ; Fu, Cheng-Yang ; Berg, Alexander C.: Ssd: Single shot multibox detector. In: *European conference on computer vision* Springer, 2016, S. 21–37

[14] Murphy, Kevin P.: *Machine learning: a probabilistic perspective*. MIT press, 2012

[15] Pal, Sankar K. ; Pramanik, Anima ; Maiti, Jhareswar ; Mitra, Pabitra: Deep learning in multi-object detection and tracking: state of the art. In: *Applied Intelligence* 51 (2021), Nr. 9, S. 6400–6429

[16] Persson, Aladdin: *YOLOv3*. `https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/object_detection/YOLOv3`, 2021

[17] Programmathically contributor Sebastion: *Understanding Padding and Stride in Convolutional Neural Networks*. `https://programmathically.com/understanding-padding-and-stride-in-convolutional-neural-networks/`. Version: 2021. – [Online; accessed 6-August-2022]

[18] Redmon, Joseph: *darknet*. `https://github.com/pjreddie/darknet`, 2013

[19] Redmon, Joseph ; Divvala, Santosh ; Girshick, Ross ; Farhadi, Ali: You only look once: Unified, real-time object detection. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, S. 779–788

[20] Redmon, Joseph ; Farhadi, Ali: YOLO9000: better, faster, stronger. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, S. 7263–7271

[21] Redmon, Joseph ; Farhadi, Ali: Yolov3: An incremental improvement. In: *arXiv preprint arXiv:1804.02767* (2018)

[22] Ren, Shaoqing ; He, Kaiming ; Girshick, Ross ; Sun, Jian: Faster r-cnn: Towards real-time object detection with region proposal networks. In: *Advances in neural information processing systems* 28 (2015)

[23] Sande, Koen E. d. ; Uijlings, Jasper R. ; Gevers, Theo ; Smeulders, Arnold W.: Segmentation as selective search for object recognition. In: *2011 international conference on computer vision* IEEE, 2011, S. 1879–1886

[24] Stender, Merten ; Tiedemann, Merten ; Spieler, David ; Schoepflin, Daniel ; Hoffmann, Norbert ; Oberst, Sebastian: Deep learning for brake squeal: Brake noise detection, characterization and prediction. In: *Mechanical Systems and Signal Processing* 149 (2021), S. 107181

[25] Wikipedia contributors: *Convolutional neural network — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=1103715044`. Version: 2022. – [Online; accessed 29-August-2022]

[26] Wikipedia contributors: *Jaccard index — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Jaccard_index&oldid=1100399116`. Version: 2022. – [Online; accessed 3-August-2022]

[27] Yang, Yun: *TensorFlow2.0-Examples*. `https://github.com/YunYang1994/TensorFlow2.0-Examples`, 2019

[28] Zhang, Zihao: *yolov3-tf2*. `https://github.com/zzh8829/yolov3-tf2`, 2019

[29] Zou, Zhengxia ; Shi, Zhenwei ; Guo, Yuhong ; Ye, Jieping: Object detection in 20 years: A survey. In: *arXiv preprint arXiv:1905.05055* (2019)