Comparison and Optimization of CNN-based Object Detectors for Fisheye Cameras

Payam Goodarzi

Supervisor: Martin Stellmacher

Advisors: Prof. Dr. Dr. hc. habil. Raúl Rojas
Prof. Dr. Tim Landgraf

Department of Mathematics and Computer Science
Freie Universität Berlin

This dissertation is submitted for the degree of
Master of Science

Institute of Computer Science January 2019

I would like to dedicate this thesis to my beloved parents.

Eigenständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Payam Goodarzi
Januar 2019

Acknowledgements

And I would like to express my sincere gratitude to my supervisor Mr. Martin Stellmacher for his tireless support, enthusiasm and immense knowledge. His precious comments and remarks guided me in all the time of experimenting and writing. Besides my supervisor, i would like to thank my professors at the Free University of Berlin: Prof. Dr. Dr. hc. habil. Raúl Rojas and Prof. Dr. Tim Landgraf for the lessons they taught me during my master's degree.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

AI	Artificial Intelligence
BDD	Berkeley Deep Drive
CCVPR	Conference on Computer Vision and Pattern Recognition
CNN	Convolutional Neural Network
COCO	Common Object in Context
ConvL	Convolutional layer
CV	Computer Vision
FC	Fully connected layer
FCN	Fully Convolutional Network
FOV	Field Of View
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IOU	Intersection Over Union
mAP	Mean Average Precision
MSE	Mean Square Error
OPP	Overlapping Pyramid Pooling
R-CNN	Regional CNN
ROI	Region of Interest

RPN	Region Proposal Network
SAT	Speed-Accuracy Trade-off
SGD	Stochastic Gradient Descent
SSD	Single Shot MultiBox Detector
SSIM	Structural Similarity Index Metric
SVM	Support Vector Machine
VOC	Visual Object Classes
YOLO	You Only Look Once

Chapter 1

Introduction

Over the last decade, autonomous driving has been attracting the attention of automobile industrialists more than ever before, although we've been pursuing the dream of traveling between destinations while seating in a driver-less car without any manual intervention since the early 1960s. The major reason of this growing tendency, is the significant and ground-breaking progresses concerning artificial intelligence and computer vision, thanks to machine learning.

An autonomous vehicle can be defined as a vehicle that uses a combination of sensors, cameras, radar and AI to drive toward a destination without a human operator. Experts have defined five levels in the evolution of autonomous driving. Each level describes the extent to which a car takes over tasks and responsibilities from its driver, and how the car and driver interact: 1) Driver assistance, 2) Partly automated driving, 3) Highly automated driving, 4) Fully automated driving 5) Full automation. These levels are obtained from [31]. Among all five levels the entire autonomous driving system can be roughly encapsulated in five main components, as it's nicely illustrated in the figure 1.1.

Sensors capture data from the environment and pass onto the next ring of this chain, **Environment Perception**, where the captured data will be processed for the consecutive

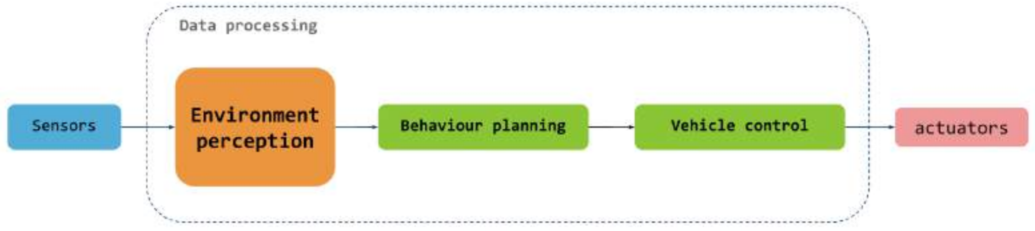


Fig. 1.1 The major components in which an automated driving system is commonly subdivided. source: [24]

decision making processes regarding the next action of the vehicle. Therefore the extracted information at this stage play vital role for an autonomous vehicle. There exist a various sort of sensors to gather data from the surroundings. Our focus here is pointed at *Image sensors* which convert light waves into signals that convey information to generate an image. Mainly each camera consists of lens and sensors. Between different types of lenses, fisheye lens is widely used in automobile industry, due to its wide field of view. This wider field of view can be attained only by generating a hemispherical perspective that produces strong visual distortion, known as barrel distortion. The resulting distortion complicates the detection process, we will investigate the potential issues in the upcoming chapters.

Convolutional neural networks (CNN) has been prevailing in the realm of computer vision during the course of the last ten years, but CV community has been mostly concentrating on development of CNN-based detectors for conventional pinhole images. In this thesis we attempt to equip an existing state-of-the-art detector specifically for fisheye images.

1.1 Objective

The ultimate objective of this thesis it to tackle the lack of CNN-based detectors specialized on fisheye images. Thereby experiments are required, therefore we will compare the performance of several state-of-the-arts and optimize one of them. Along the way we construct a new object detection framework to perform our experiments as convenient as possible. Generally the entire work done in this thesis can be divided into two main part as follow:

- Optimization of a CNN-based detector for fisheye images.
- Construction of an object detection framework.

1.2 Document Structure

The thesis will be structured as follows: in **Chapter 2** we familiarize the reader with the foundation of convolutional neural networks. In **Chapter 3** mathematical model of fisheye camera plus its pros and cons will be discussed. **Chapter 4** breaks down the state-of-the-art detectors and sheds light on their important design related differences. In **Chapter 5**, the actual approaches taken in order to improve the chosen CNN detector's performance on fisheye images will be explained. **Chapter 6** provides information about the structure of our object detection framework and the simulation of fisheye effect. Eventually the successful results of the work done in this thesis will be presented in **Chapter 7**, including example images of the evaluation set used. **Chapter 8** will summarize the system design and results of the evaluation, identify the limitation and draw up conclusion based on discussion brought in the preceding chapters.

Chapter 2

Convolutional Neural Network

2.1 Biological Inspiration

The Evolution took over 500 million years to succeed in creation of a complex and sophisticated neural system, which enables creatures to perceive and sense their surrounding environment. We as human persistently inspect the world around us by gathering data via a multitude of sensors, vision (sight), audition (hearing), gustation (taste), olfaction (smell), and somatosensation (touch). Unlike all other sensory systems *visual system*, composed of components from the eyes to neural circuits, starts to gradually developing from the very first moment of life after birth. Initially the visual system is immature and can only detect variation in brightness, given time the intertwined connection between eyes and the brain, called the *Primary Visual Pathway*, improves and gains the capability of detection and recognition. The upshot is that we can subconsciously detect, track, label and distinguish objects.

As depicted in figure 2.1 the visual information captured by light receptors in eyes goes through optic nerve to the *Primary Visual Cortex*, where the processing takes place. Neurons in the visual cortex fire when visual stimuli appear within their *Receptive Field* (the particular region of the sensory space in which a stimulus will modify the firing of that neuron). Any individual neuron may respond best to a subset of stimuli within its receptive field, in other words each neuron is sensitive to a particular set of patterns. The neuronal responses get optimally tuned to specific patterns through experience, this property is called *Neuronal Tuning*. The tuning becomes later more complex in comparison to the early stage of the visual system maturation process. The collection of the tuned-neurons in visual cortex can extract patterns in input visual information and construct visual perception. The whole visual system develops by repeating the following procedure:

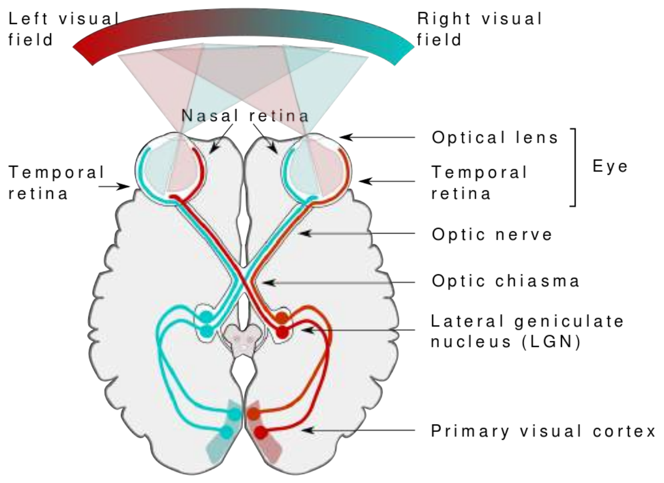


Fig. 2.1 The visual pathway of human brain. source: [1]

1. Reception
2. Transmission
3. Processing
4. Interpretation

as a result of this practice we are able to localize, classify and interpret objects with an extraordinary precision rate.

In computer vision **Convolutional Neural Network** (CNN) is a felicitous attempt to leverage mathematics for emulating the visual perception. If we take the whole visual system as a complex mathematical function, which takes light in numeric form as input, processes it and outputs interpretable numbers, it would be theoretically possible to create such function. The components of this function act like the actual neurons in the visual cortex. It is fair to say that the actual functionality of neurons in brain is way more complex and still poorly understood to be identically replicated with simple mathematical expressions, after all the fundamental nature of neurons inspired the idea of Neuronal Networks (NN), upon which CNNs are built. In context of NNs, a single neuron receives inputs, performs a dot product of its weights followed by a non-linearity, as illustrated in the figure 2.2. An NN is in fact a network of such neurons. CNN as a variant of NNs arranges its neurons in three dimensions (width, height, depth). A regular neuron (or sometime referred to as a *kernel* or *filter*) is an array of weights with the same depth of the input, it slides or convolves over the input with specific step size called *stride*, at each step the neuron computes element wise

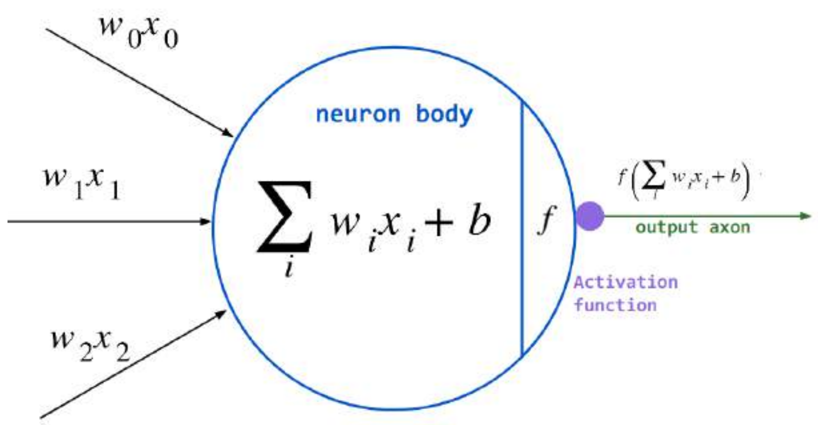


Fig. 2.2 A single neuron is a simple linear classifier followed by an activation function.

multiplication between pixel values of the input and its weights, the multiplications are then all summed up yielding a smaller output. Each of these neurons can be thought of as *feature identifiers*, this means that every individual neuron is sensitive to specific type of input. For instance one could response only to straight edges, colors or curves. For the computer vision specialists from ages before that machine learning placed itself into computer vision community, neurons of CNN resemble the kernels used for blurring, sharpening, embossing, edge detection, and more. The concept of neurons is more or less the same only with one determining difference, that the weights of neurons/kernels are inconstant and randomly initialized. As a result each layer of CNN is a collection of various detectors, specialized and trained to hunt down particular feature.

There is a vast variety of building bricks to construct a network, depending on use-case different structure will be suitable. As described in [34] a CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, RELU layer i.e. activation function, pooling layers, fully connected layers and normalization layers. Next we take a look at these components.

2.2 Layers

Tuning hyper-parameters is undeniably among the daily concerns of any data scientist. Designing a CNN is also no exception, it starts with fixing several architectural hyper-parameters. As already mentioned CNNs have layered architecture, the main building block of CNN is **Convolutional Layer (ConvL)** that performs spatial convolution over input image. Depending on the configuration of a ConvL the resulting output size varies. Several hyper-parameters control the spatial size of the output:

- *Neuron size*: size of the sliding window.
- *Stride*: step size to translate the sliding window.
- *Padding*: Sometimes it is convenient to pad the input volume with zeros around the border

The size of neurons and stride determine the spatial size of the output in each layer, on the other hand total number of kernels/filters controls the depth of the output. Different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. The spatial size of the output volume can be computed as a function of the input volume size **W**, the receptive field of the neurons **F**, the stride **S** and the amount of the zero padding used **P**. The formula $(\mathbf{W} - \mathbf{F} + 2\mathbf{P})/\mathbf{S} + 1$. For example for a 7×7 input and a 3×3 filter with stride 1 and pad 0 we would get a 5×5 output. With stride 2 we would get a 3×3 output.

Convolution is the main characteristic of CNN, that can be realized only if we apply same neurons on input by using the same weights, otherwise it would contradict the definition of convolution. This vital property of CNN is also called **Parameter Sharing**, which affords certain advantages such as *Translation Invariance* and reduction in number of parameters. Total number of parameters of a network is proportional to the number of neurons in each layer. For instance a layer without convolution with output size of $55 \times 55 \times 96$ has $290,400$ neurons, each has $11 \times 11 \times 3 = 363$ weights and 1 bias, totally $290400 \times 364 = 105,705,600$ parameters only for one layer. Adding more layers would rapidly increase the number of parameters, which makes a network computationally expensive and harder to train. However with parameter sharing scheme, the layer in our example would now have only 96 unique set of weights, for a total of $96 \times 11 \times 11 \times 3 = 34,848$ unique weights, or $34,944$ parameters (+96 biases) and is also able to detect its target features regardless of their locations.

2.2.1 Non-Linearity

As shown in figure 2.2 each neuron passes its output to a activation function for the purpose of providing Non-linearity to the network. Lack of dynamic in linear systems makes them unsuitable in most cases in real world applications, since mathematical description based on assumption of constant proportionality between variables is more a naive way of delineating natural behavior due to the high variance in the nature. Stacking linear equations together without activation function yields a linear composition, which is a hyperplane in a high dimensional space acting as separating line. This would turn the whole problem into linear regression, thus inappropriate for finding correlation in input with wide spread variance and

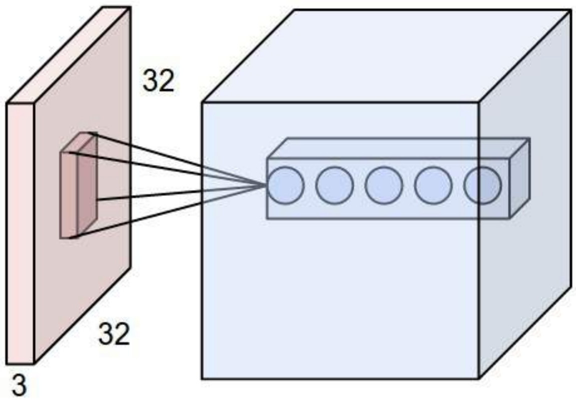


Fig. 2.3 An example input volume in red (e.g. a $32 \times 32 \times 3$ CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth. source: [34]

dimensionality. Therefore non-linearity is vital to NN and it makes it easy for the model to adapt with variety of data and to differentiate between the output. There are several activation functions we may encounter in practice:

Sigmoid: is a variant of *Logistic function* having a characteristic S-shaped curve. Return value of sigmoid is squashed and monotonically increasing in an interval from 0 to 1 as illustrated in figure 2.4a. The sigmoid function has been frequently used since it resembles the behavior of firing neuron nicely: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor. It has two major drawbacks as stated in [35]:

- "Sigmoid saturates and kills gradients. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. During backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn."
- "Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network would be receiving data

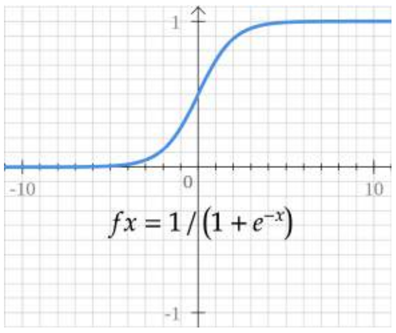
> that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ element-wise in $f = wTx + b$)), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above."

Tanh: the hyperbolic tangent non-linearity shown in figure 2.4b is a rectified version of sigmoid with the purpose of zero-centering the outputs. Technically, it is a scaled sigmoid neuron. The advantage of Tanh is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero but it still suffers from *vanishing gradient* problem.

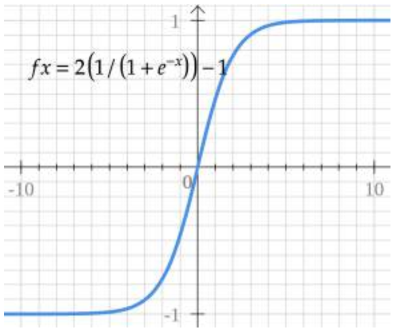
ReLU: the Rectified Linear Unit (figure 3.6b return the maximum between zero and the input. Actually it sets a lower bound at zero and the upper bound can go to infinity, since it's not thresholded. Both the function and its derivative are monotonic. There are several pros and cons about ReLU:

- it significantly accelerates the convergence of stochastic gradient descent compared to the sigmoid and tanh functions, due to its linear non-saturating form.
- computationally it is a lighter operation in comparison to exponential in sigmoid and tanh.
- all the negative inputs turn into zero once they reach ReLU, consequently the network will not be able to map the negative values appropriately.
- In [35] is also pointed out:

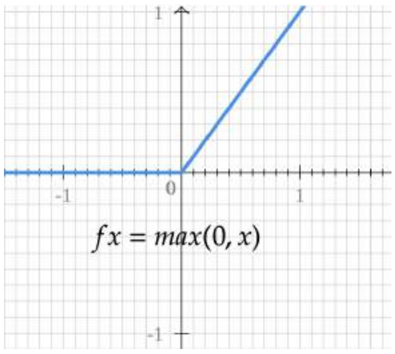
> "Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network



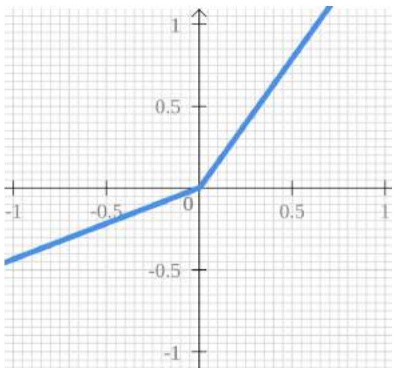
(a) Sigmoid



(b) Tanh



(c) ReLU



(d) Leaky ReLU

Fig. 2.4 Four most used activation functions.

can be “dead” (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue."

Leaky ReLU: to enlarge the range of the ReLU in negative direction, Leaky ReLU uses a small slop (usually 0.01), hence the Leaky ReLU can tend to infinity in both directions. That is, the function computes $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$ where α is a small constant. The difference between ReLU and Leaky ReLU is noticeable in figure 2.4d.

One of the technique to bold out features in an image is to sharpen it. In order to do so we need to drain indistinct and blurry pixels by filtering the strongest values. In CNN this is performed by **Max Pooling Layer (MaxPool)**. In addition to MaxPool, there are some other pooling units, such as average pooling or even L2-norm pooling. Average pooling was often used historically but has recently lost its popularity compared to the max pooling operation, which has been shown to work better in practice. MaxPool can be seen as a down-sampling operation, this reduces the dimensionality of the extracted feature while

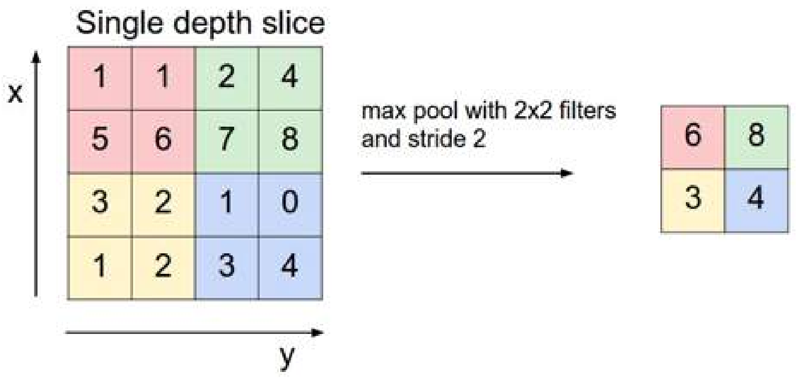


Fig. 2.5 The most common down-sampling operation is max pooling, here shown with a stride of 2. MaxPool applied on each 2×2square yields 4 numbers. source: [34]

preserving information. Figure 2.5 illustrates how a MaxPool operates over a 4×4 feature map.

In CNNs classification might be performed by **Fully Connected Layer (FC)** after convolution and pooling. There exist several networks like [37] or [13], which drop the FC and replace it with ConvL. Such networks are called **Fully Convolutional Network (FCN)**. Next we will see how this replacement is possible. All the neurons in FC are connected to all the activations in the previous layer. Basically FC performs the same operations (matrix multiplication followed by a bias offset) like ConvL with only difference, that in ConvL neurons are connected only to local region in the input and they share parameters [34]. Since the functionality of FC and ConvL is identical we can convert FC to ConvL or visa versa, we take an example from [34]:

> "An FC layer with $K = 4096$ that is looking at some input volume of size $7\times 7\times 512$ can be equivalently expressed as a ConvL with $F = 7, P = 0, S = 1, K = 4096$. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be $1\times 1\times 4096$ since only a single depth column “fits” across the input volume, giving identical result as the initial FC layer. This conversions allows us to slide the original CNN very efficiently across many spatial positions in a larger image, in a single forward pass. Given 224×224 image going through reduction by 32 yields a volume of size $[7\times 7\times 512]$, then forwarding an image of size 384×384 through the converted architecture would give the equivalent volume in size $[12\times 12\times 512]$, since $384/32 = 12$. Assuming that we have 3 following ConvL that we convert from FC layers, it would now give the final volume of size $[6\times 6\times 1000]$, since $(12-7)/1+1 = 6$. Note that instead of a single vector of class scores of size

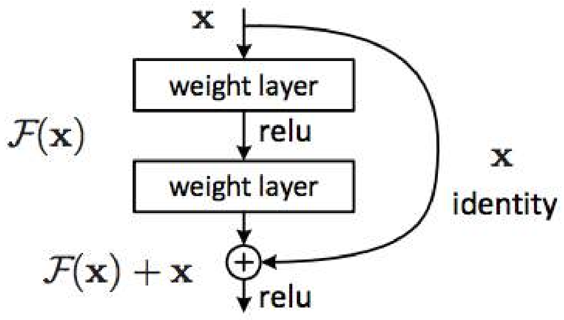

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Fig. 2.6 A residual block - the fundamental building block of residual networks. source: [12]

> $[1 \times 1 \times 1000]$, we're now getting an entire 6×6 array of class scores across the 384×384 image."

The most common pattern for building a CNN is first stacking a few ConvL using ReLU, following them with MaxPool and repeating this pattern until the image has been merged spatially to a small size and finally adding FC for holding the output. The more hidden layers, the more features is believed to be detected at various levels of abstraction. In practice shallow networks are good at memorization but not good at generalization, deeper networks are much better at generalizing because they learn all the intermediate features between the raw data and the high-level classification. Aside from the specter of overfitting, the deeper a network becomes the harder is the training. To address this problem and ease the training of deep networks Microsoft Research team presented *Deep Residual Learning framework(ResNet)* [12]. ResNet is much deeper than their plain counterparts, yet they require a similar number of parameters. As stated in [12]:

> "With network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution to the deeper model by construction: the layers are copied from the learned shallower model, and the added layers are identity mapping. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart."

The proposed residual block is shown in figure 2.6.

ResNet prevents the problem of vanishing gradients by utilizing *Skip Connections* or *Shortcuts*. ResNet reuses activations from a previous layer until the layer next to the current one have learned its weights. During training the weights will adapt to mute the previous layer and amplify the layer next to the current. In the simplest case only the weights for the connection to the next to the current layer is adapted, with no explicit weights for the upstream previous layer. This usually works properly when a single non-linear layer is stepped over, or in the case when the intermediate layers are all linear. If not, then an explicit weight matrix should be learned for the skipped connection. ResNet has fewer layers in the initial phase, which make it easier to learn, it gradually expands the layers and goes deeper as it learns more of the feature space. A neural network without residual parts will explore more of the feature space. This makes it more vulnerable to small perturbations that cause it to leave the manifold altogether, and require extra training data to get back on track.

2.3 Backpropagation

[1] Generally speaking, the main objective of NN is to iteratively modulate all parameters, consisting weights and biases, in a way that minimizes the defined cost function at the bottom of the network, in other words training neurons to generate an output which has possibly the least distance from the desired output. This is what we call *Learning process*. Therefore **Backpropagation** algorithm was introduced in the 1970s. Backpropagation is a way of computing gradients of expressions through recursive application of *Chain Rule*, to put it simply, at the heart of backpropagation is an expression for the partial derivative $\partial C/\partial w$ of the cost function C with respect to any weight w and bias b in the network. The expression tells us how quickly the cost changes when we change the weights and biases. It actually gives us detailed insights into how changing the weights and biases changes the overall behavior of the network. After initializing the weights of the network, the backpropagation algorithm computes the needed corrections on weights. The algorithm can be decomposed into three main steps as follow:

1. Feed-forward computation
2. Backpropagation
3. Weight updates

Recall that a feed-forward neural network is a computational graph whose nodes are computing units and whose directed edges transmit numerical information from node to node. Each computing unit is capable of evaluating a single primitive function of its input. In

[1]Descriptions and annotations of this subsection stem from [28]

fact the network represents a chain of function compositions which transform an input to an output vector. The learning problem consists of finding the optimal combination of weights so that the network function φ approximates a given function f as closely as possible. However, we are not given the function f explicitly but only implicitly through some examples.

Consider a network with n input and m output units. Certainly the number of intermediate hidden units can be variant. Given a training set $\{(x_1,t_1),...,(x_p,t_p)\}$ consisting of p ordered pairs of $n-$ and $m-dimensional$ vectors. When the network is fed with x_i from the training set, it produces an output o_i, usually different from to the target t_i. The goal is to make o_i and t_i identical for $i = 1,...,p$ by using the numerical learning algorithm (Backpropagation). For the sake of simplicity, we take the quadratic error function 2.1, which we want to minimize.

$$E = \frac{1}{2}\sum_{i=1}^{p} \|o_i - t_i\|^2 \tag{2.1}$$

After minimizing this function for the training set, new unknown input patterns are presented to the network and we expect it to interpolate. The network must recognize whether a new input vector is similar to learned patterns and produce a similar output. In order to find a local minimum of the error function, our task is to compute the gradient recursively. Because E is calculated exclusively through composition of the node functions, it is a continuous and differentiable function of the l weights $w_1,w_2,...,w_l$ in the network. We can thus minimize E by using an iterative process of gradient descent, for which we need to calculate the gradient:

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2},..., \frac{\partial E}{\partial w_l}) \tag{2.2}$$

Under assumption that we use a simple optimization method, each weight is updated incrementally

$$\triangle w_i = -\gamma \frac{\partial E}{\partial w_i} \; for \; i = 1,...,l \tag{2.3}$$

where γ represents a learning constant, i.e., a proportionality parameter which defines the step length of each iteration in the negative gradient direction. The chain rule tells us that the correct way to "chain" these gradient expressions in equation 2.2 together is through multiplication (for example $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}.\frac{\partial q}{\partial x}$. This computation of gradient can be nicely understood and visualized with a circuit diagram shown in figure 2.7. Every gate in a circuit diagram gets some inputs and can right away compute two things: 1. its output value and 2. the local gradient of its inputs with respect to its output value. Notice that the gates can do this completely independently without being aware of any of the details of the full circuit that they are embedded in. However, once the forward pass is over, during backpropagation the

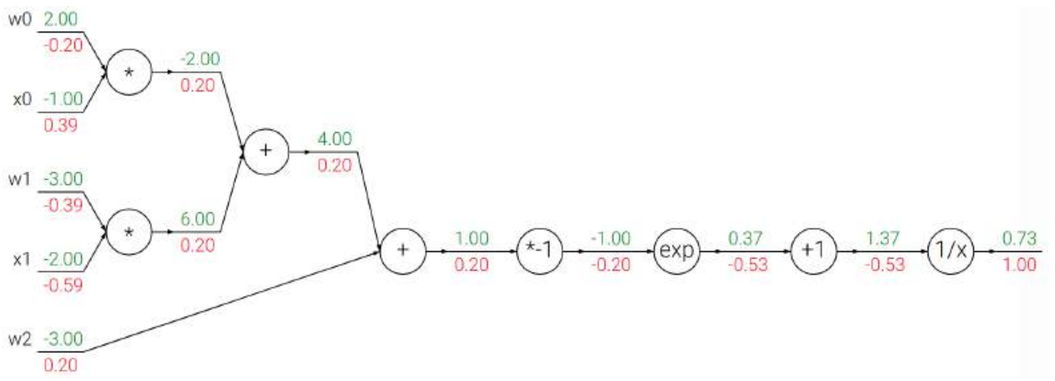


Fig. 2.7 Example circuit for a 2D neuron with a sigmoid activation function. The inputs are $[x0, x1]$ and the (learnable) weights of the neuron are $[w0, w1, w2]$. The forward pass computes values from inputs to output (shown in green). The backward pass then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs of the circuit, gradient of each unit can be seen as signal to other units. This shows that if we alter the last gate by 1.0, this would cause the chain reaction over all the circuit and as a result $w2$ changes by 0.20 for instance. The gradients can be thought of as flowing backwards through the circuit. source: [36]

gate will learn about the gradient of its output value on the final output of the entire circuit. Chain rule says that the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs. "This extra multiplication (for each input) due to the chain rule can turn a single and relatively useless gate into a cog in a complex circuit such as an entire neural network." In fact gates communicate with each other through the gradient signal whether they should increase or decrease their output, so that the final output gets the desired value.

At each iteration/step gradient shows us the direction along which brings about the steepest decline in the value of the loss function, ideally the *Global Minima* as illustrated in figure 2.8. The ultimate goal is to reach the global Minima. Although in search of the global minima the network could converge to the *Local Minima*. The search after global minima in the error landscape turns the whole process into a optimization problem.

2.4 Optimization

As stated before the goal of optimization is to find a particular combination of weights W that minimizes the loss function. It bears mentioning that contrary to the depiction 2.8, optimizing a NN is way beyond a simple convex problem, since we have as many degrees of freedom as the number of trainable variables in NN, this high dimensionality makes it a complex function that exceeds the limit of visualization in our 3D world. For the sake

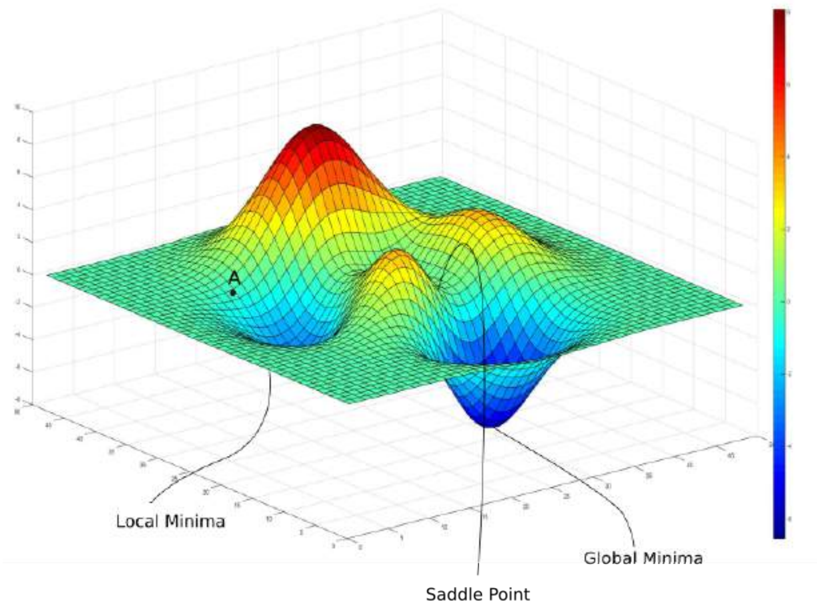


Fig. 2.8 Error landscape created by computing the gradient. The ultimate goal is to reach the global minima, where the gradient is zero. source: [17]

of simplicity we stick to the regular 3D coordinate system. Assuming that the weights are randomly initialized, which is usually the case, we start from point A as shown in figure 2.8 and want to reach the global minima located at the depth of the convex-shape trench colored in blue. Recursively we compute the gradient to figure out the direction, in which we should proceed towards the global minima. The optimal trajectory would be the shortest path with taking large steps. But local gradients could be utterly misleading and shove it into local minima, once it's trapped there it takes large step to climb out of the current hole. Enlarging the steps is not the smartest solution to deal with this situation, since it could throw it too far ahead and just jump over the right spot, instead of scrambling down into the cliff, where the globally deepest point lies. For this very reason we should pick the step size diligently.

The simplest approach is to search in a random direction and then take a step only if it leads downhill (*random local search*). Concretely, we select a random W to start from, generate random perturbations δW to it and if the loss at the perturbed $W + \delta W$ is lower, we perform an update. Although this method could work and not be fully off the track, but is still a blind shot in the dark and quite a leap of faith. Besides it is computationally expensive and time-consuming as well. It is apparent, that logically we should orient ourself by scrutinizing possible steps. For this we make use of *Limit* of a function ($\lim_{x \to p} f(x) = L$), that shows us the behavior of a function (changes in output) while input approaches a particular value. We compute the best direction by utilizing the limit. This is mathematically guaranteed to be the direction of the steepest descend (at least in the limit as the step size goes towards zero). This direction will be related to the **gradient** of the loss function.

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \tag{2.4}$$

The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step. Small steps are likely to lead to consistent but slow progress. Large steps can lead to better progress but are more risky. Note, for a large step size we will overshoot and make the loss worse. The step size (*Learning Rate* η) will become one of the most important hyper-parameters that we will have to carefully tune.

The procedure of repeatedly evaluating the gradient and then performing a parameter update is called **Gradient Descent**. Mathematically expressed, gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley. As it's elaborately declared in [29]: "there are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, they make a trade-off between the accuracy of the parameter update and the time it takes to perform an update."

- **Batch/Vanilla Gradient Descent (BGD):** "it computes the gradient of the cost function w.r.t the parameters θ over the entire training dataset:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2.5}$$

  for each update the gradient is computed again, this retards the convergence and also doesn't allow to update the model *online*, i.e. with new examples on-the-fly. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces."

- **Stochastic Gradient Descent (SGD):** much faster convergence can be achieved in practice by evaluating the gradients of smaller batches to perform more frequent parameter updates. In [29] describes SGD as follow:

SGD performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.6}$$

In contrast to batch gradient descent, that performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update, SGD address this redundancy by doing one update at a time. Therefore is usually faster and makes online learning possible. Frequent updates with high variance causes fluctuation in the objective function. This fluctuation enables it to jump to new and potentially better local minima, or on the other hand jump over them. The fluctuation can be reduced by gradually decreasing the learning rate (*Exponential Decay*). Under this condition SGD shows more or less the same convergence behavior as BGD, converging to a local or the global minimum for non-convex and convex optimization respectively.

- **Mini-batch Gradient Descent:** it's an attempt to find equilibrium between BGD and SGD. Mini-batch gradient descent performs update over every mini-batch of n training examples, [29] formulates it as follow:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.7}$$

  due to the correlation between training data, the gradient over mini-batch would be a good approximation of the gradient of the full objective. Aside of that, mini-batch gradient descent reduces the variance of the parameter updates, thus less fluctuation and more stable convergence.

Several challenges are not met by gradient descent. As mentioned already choosing a proper learning rate is dodgy. Too small learning rate leads to slow convergence, while a too large learning rate can hinder convergence and destabilize the loss function or even diverge. Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous sub-optimal local minima. The difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points (shown in figure 2.9) are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant

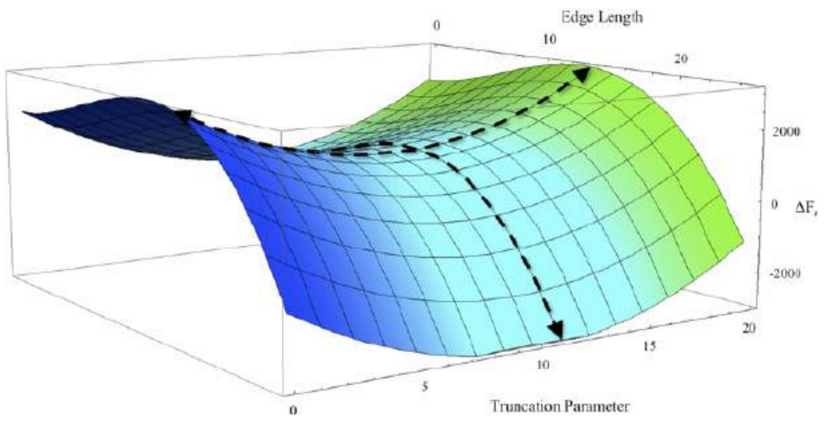
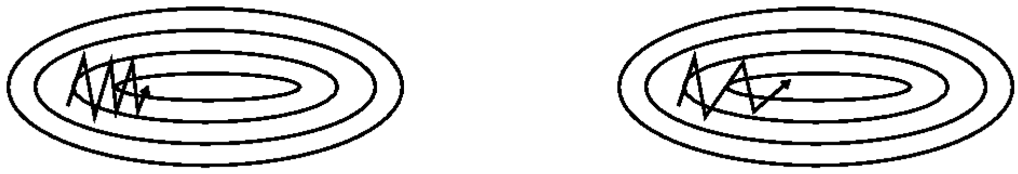


Fig. 2.9 A saddle point. The name stems from its resemblance to a horse's saddle. source: [17]

progress along the bottom towards the local optimum as in figure 2.10b. Once again we draw inspiration from the nature to tackle this issue, this time not from biology but rather physic. Analogously when we push a ball down a hill, it accumulates momentum as it rolls downhill, becoming faster and faster on the way. **Momentum** [23] simulates this physical property. This method accelerates SGD in the relevant direction and dampens oscillations (shown in figure 2.10a) by adding a fraction γ of the update vector of the past time step to the current update vector:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{2.8}$$

So far the same learning rate η is used to update all parameters θ. **Adagrad** [4] adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. In fact Adagrad uses a different learning rate for every parameter θ_i at every time step t. For brevity, we use g_t to



(a) SGD without momentum (b) SGD with momentum

Fig. 2.10 Momentum forces the SGD to go deep faster and depress the oscillation. source: [29]

denote the gradient at time step t. $g_{t,i}$ is partial derivative of the objective function w.r.t the parameter θ_i at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) \tag{2.9}$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{2.10}$$

Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{g_{t,ii} + \varepsilon}} \cdot g_{t,i} \tag{2.11}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t [2], while is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Adagrad eliminates the need to manually tune the learning rate. Adagrad's main weakness is its accumulation of the squared gradients in the denominator, since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

Adadelta [45] is an extension of Adagrad, again we refer to the description of [29]:

> "Adadelta aims to diminish the monotonically decreasing learning rate problem. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w. The inefficient storing of previous squared gradients is replaced with the recursive sum of gradients defined as a decaying average of all past squared gradients. The running $E[g^2]_t$ average at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:
>
> $$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{2.12}$$
>
> the diagonal matrix G_t is replaced with the decaying average over past squared gradients $E[g^2]_t$:
>
> $$\triangle \theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t \tag{2.13}$$

with Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule."

Many other optimization methods like RMSprop, Adam, AdaMax, Nadam and etc. attempt to develop existing methods or come up with a new solution suitable universally to minimize objective function. Generally speaking depending on the input size and the difference in the depth of the network the performance of optimizers differs, but astonishingly many recent networks use either vanilla SGD without momentum with a simple learning rate annealing schedule or momentum with exponential decay. Training a network, that converges and performs properly, requires more than just a smart choice of optimizer, various types of regularization and weight initialization also play a key role.

2.5 Weight Initialization

It is quite evident, that the complexity of finding the sweet spot in the error landscape is strongly contingent upon the sharpness of resultant slopes and elevations from the error function inputted with weights. Thus the **Weight Initialization** becomes substantial. By analogy initial weights are like the first bricks of a tower, the foundation spread underneath, if the foundation is skewed the whole tower will be unsteady and unstable, if ever a tower could be built upon such a grounding. One can initialize all weights randomly, this could potentially lead to two main issues: vanishing gradient or exploding gradient. In order to address these issues weights are drawn from a zero-centered ($\mu = 0$) normal distribution with a constant standard deviation σ. With this formulation, every neuron's weight vector is initialized as a random vector sampled from a multi-dimensional gaussian, so the neurons point in random direction in the input space. The number of inputs determines the magnitude of the variance, to keep the weights within a normal interval we normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its number of inputs. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. In practice mostly *Truncated Normal Distribution* is used, since it has useful property that bounds the random variable from either below or above. Implemented truncated normal distribution in Tensorflow discards and re-draws values more than two standard deviations from the mean.

Xavier [9] suggested a smart way to retain the normality of the weights. Under assumption, that the inputs X_i and the weights W_i are all independent and identically distributed, it initializes the weights from a distribution with zero mean and a suitable variance:

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{2.14}$$

where n_{in} and n_{out} are respectively the number of inputs and outputs of a layer. Xavier initializer allows the neurons to propagate the signals accurately.

2.6 Normalization

There is no guarantee that even a delicately initialized network would converge and perform desirably. Without some refining techniques building a well-functioning network is nearly infeasible. Now we flick through several crucial normalization steps.

Local Response Normalization (LRN) implements a form of *lateral inhibition*. Lateral inhibition is a concept in neurobiology, that refers to the inhibition that neighboring neurons in brain pathways have on each other. For example, in the visual system, neighboring pathways from the receptors to the optic nerve, which carries information to the visual areas of the brain, show lateral inhibition. This means that neighboring visual neurons respond less if they are activated at the same time than if one is activated alone. So the fewer neighboring neurons stimulated, the more strongly a neuron responds. This increases the sensitivity of the visual system to the edges of a surface for instance. In CNN basically we want to detect the high frequency features in images with a large response. If we normalize around the local neighborhood of the excited neuron and subdue its neighbors, it becomes even more sensitive as compared to its neighbors. Aside from that LRN would help specially when we use ReLU activation function and keeps ReLU neurons from growing unboundedly, which can cause gradient explosion, due to the chain rule. The equation 2.15 is the mathematical expression of LRN in CNN, where b is the output of the kernel i at the position (x,y), n is the magnitude of normalization in neighborhood, N number of kernels. α , β and k are hyper-parameters, that need to be fine-tuned.

$$b^i_{x,y} = a^i_{x,y}\left(k + \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^i_{x,y})^2\right)^{\beta} \tag{2.15}$$

Dropout: In CNNs or more generally machine learning *overfitting* is a syndrome, that hinders a model from showing generalized behavior. In simple terms the model performs good only on the training data-set, with which it is trained. Dropout [33] is a technique for addressing this problem. The key idea is to randomly drop units from the neural network during training. This prevents nodes from co-adapting too much and being dependent on other nodes. This significantly reduces overfitting and gives major improvements over other regularization methods. If we assume each node as a gene from biological perspective it makes them able to work well with another random set of gene. Since a gene cannot rely on a large set of partners to be present at all times, it must learn to do something useful on its

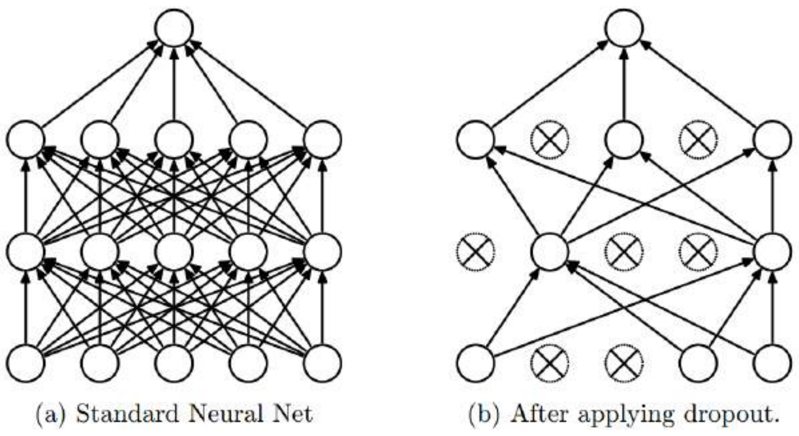


Fig. 2.11 Dropout. source: [33]

own or in collaboration with a small number of other genes. In this way the genes or nodes in a CNN are more robust. Consider a neural network with L hidden layers. Let $l \varepsilon \{1,...,L\}$ index the hidden layers of the network. Let $z^{(l)}$ denote the vector of inputs into layer l, $y^{(l)}$ denote the vector of outputs from layer l ($y^{(0)} = x$ is the input). $W^{(l)}$ and $b^{(l)}$ are the weights and biases at layer l. The feed-forward operation of a dropout neural network (Figure 2.11b) can be described as (for $l \varepsilon \{0,...,L-1\}$ and any hidden unit i)

$$\begin{aligned} r_j^{(l)} &\sim Bernoulli(p), \\ \widetilde{y}^{(l)} &= r^{(l)} * y^{(l)}, \\ z_i^{(l+1)} &= w_i^{(l+1)} \widetilde{y}^{l} + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}) \end{aligned} \tag{2.16}$$

For any layer l, $r^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability p of being 1. This vector is sampled and multiplied element-wise with the outputs of that layer, $y^{(l)}$, to create the thinned outputs $\widetilde{y}^{(l)}$. The thinned outputs are then used as input to the next layer.

Batch Normalization: As training progresses and weights get updated to different extents, consequently the normality of the weights disappears, which slows down the training and amplifies changes as the network becomes deeper. As the input layer is always normalized by adjusting and scaling the activations, reasonably we could benefit the normalization on all hidden layers, batch normalization [15] reestablishes this normalization over each intermediate layer. Technically, batch normalization reduces the *covariate shift*. To increase the stability of a network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}} \tag{2.17}$$

where $x = (x^{(1)}...x^{(d)})$ is layer input. Expectation E and variance Var are computed over the training dataset. Note that simply normalizing each input of a layer may change what the layer can represent. To address this, batch normalization makes sure that the transformation inserted in the network can represent the identity transform. To accomplish this, [15] introduced, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}$, $\beta^{(k)}$, which scale and shift the normalized value. These parameters are learned by training and restore the representation power of the network, as stated by the author [15].

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}. \tag{2.18}$$

2.7 Augmentation

After we paid heed to all prerequisite refinement strategies to construct a CNN, the dataset, on which we train a network plays a key rule in the successful performance of the network. In terms of dataset there is several defining characteristics, which distinguish datasets, for instance the size of a dataset determines the magnitude of the variance in data assuming that examples are widely scattered (this is partially discussed in chapter 4). In the majority of cases it is necessary to boost the manifoldness in the data, in order to enhance the generalization of the network. **Data Augmentation** is a pre-processing technique to realize this. It enlarges the dataset by manipulating the data in a various ways, and thereby balances the proportionality between amount of examples and parameters of a network.

In CNN is common to apply *online augmentation* on input images, mostly consisting of filters such as flip, rotation, scale, crop, translation or gaussian noise (shown in figure 2.12. In other words these filters prepare the network for dealing with different potential scenarios, therewith a CNN learns to spot the same feature regardless of the affine transformations or differences in visual attributes of an image (e.g. brightness, contrast etc.), as a result the network has higher range of adaptivity and won't suffer from overfitting. In this thesis data augmentation is called in action to extend a CNN capabilities in object detection on intense distorted objects in street. This is done by training the network with input images, which

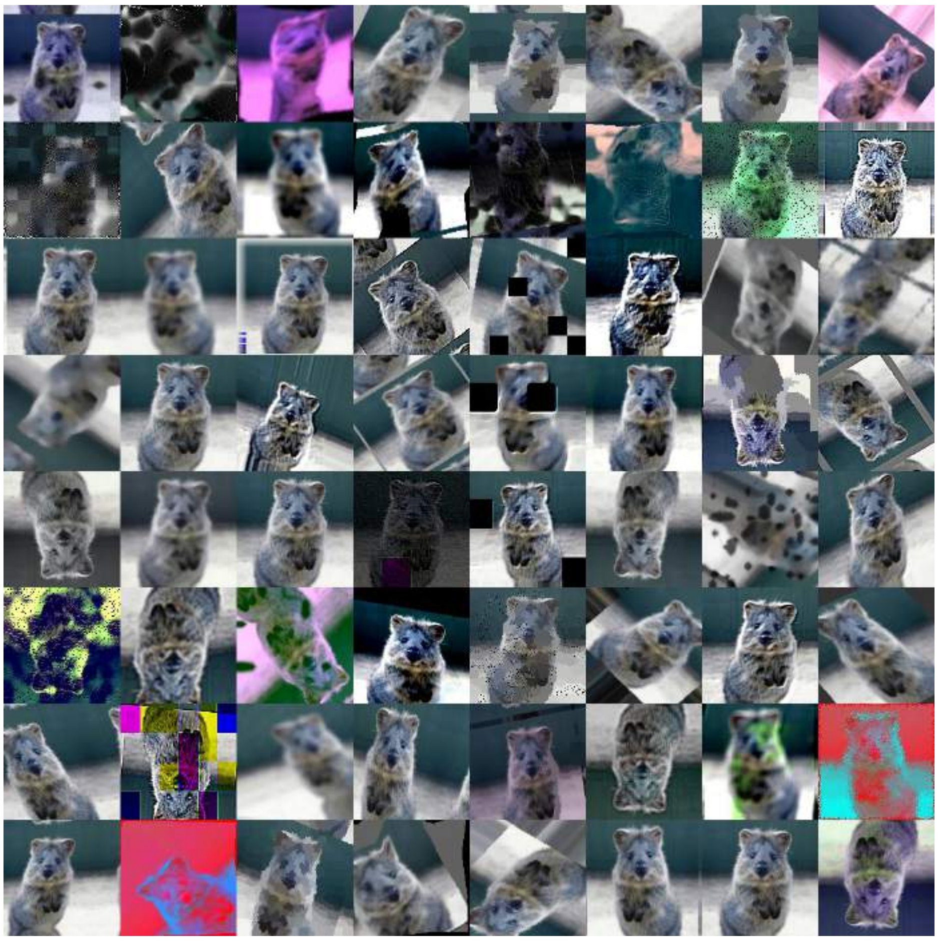


Fig. 2.12 Example augmentations of one input image from python *imgaug* library. (Crop, Pad, Flip, Invert, Gaussian Blur, Sharpen, Emboss, Edge Detect, Dropout, Salt, Pepper, Gray-scale, Contrast, Brightness, Hue, Piecewise Affine, Affine Rotate, Affine Translation and etc. source: [14]

went through a barrel distortion filter. In chapter 5 and 7 we investigate this further to find out whether augmentation truly improves the performance of the network.

Chapter 3

Fisheye Camera

3.1 Pinhole camera vs Fisheye camera

Almost all existing datasets contain images shot by cameras, which are conceptually built based on **Pinhole Camera** model. In pinhole camera model light from a 3D coordinate system (real world) travels through the aperture and will be projected into a 2D coordinate system (image plane) using a perspective transformation. There are various type of formulation and notation for this model, here we follow the model construction from *OpenCV*, since the methods used in this thesis to build fisheye model are based on the OpenCV functions.

The mapping/transformation from 3D space to 2D is formulated as follow:

$$\begin{aligned} s\, m' &= A\,[R|t]\,M' \\ s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} &= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \end{aligned} \tag{3.1}$$

where $[X,Y,Z]$ are the coordinate of a point in the 3D world coordinate space, $[u,v]$ are the coordinates of the point in 2D space after projection and A is the camera matrix containing *intrinsic parameters*. Camera matrix A describes the actual mapping from 3D to 2D on image plane, A is estimated individually for each camera and remains constant as long as the focal length (f_x, f_y) is fixed. (c_x, c_y) denotes the principal point of a camera, that is usually the center of the image. The principal axis is represented as z in the viewing direction of the camera as shown in the figure 3.1. $[R|t]$ is the translation matrix containing the extrinsic parameters. The first three columns is the *rotation matrix R* and t performs the translation. $[R|t]$ is used to describe the camera motion around a static scene, or vice versa,

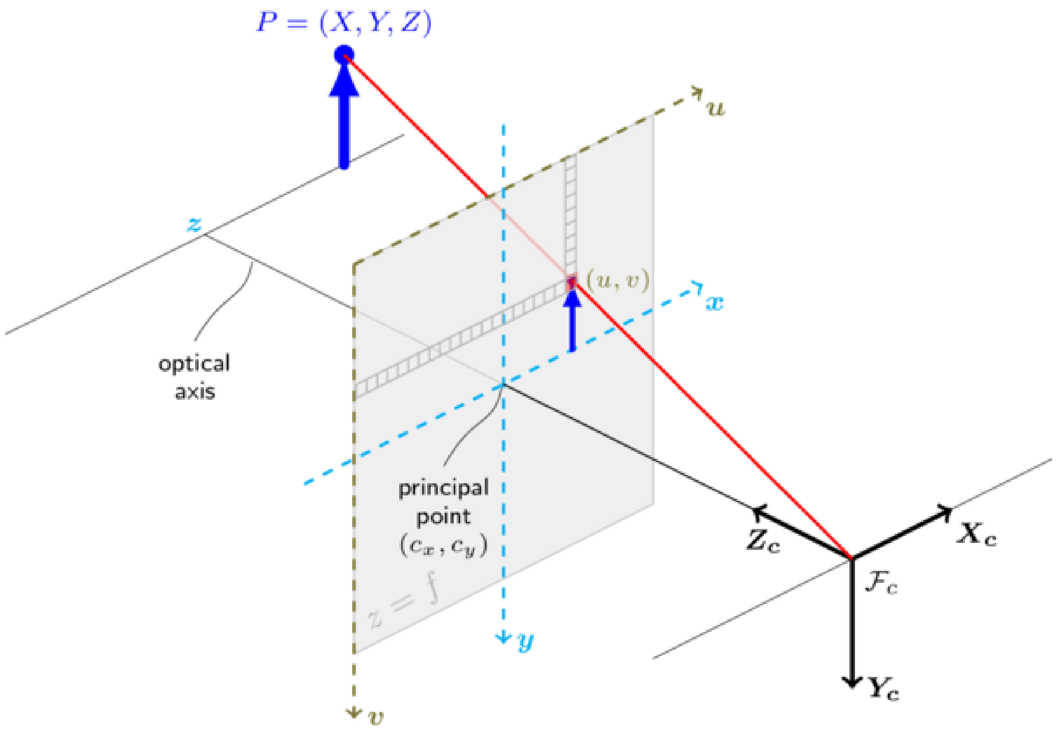


Fig. 3.1 The Pinhole camera model presented in OpenCV. source: [22]

rigid motion of an object in front of a still camera. That is, $[R|t]$ translates coordinates of a point $[X,Y,Z]$ to a coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{aligned} \begin{bmatrix} x \\ y \\ z \end{bmatrix} &= R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\ x' &= x/z \\ y' &= y/z \\ u &= f_x * x' + c_x \\ v &= f_y * y' + c_y \end{aligned} \tag{3.2}$$

A simple pinhole camera carries no lens and therefore is distortion-free, as soon as light passes through another medium than air (a lens for instance), it will be interfered and distracted. This produces distortion, which should be taken into account in order to construct an accurate camera model, the process of estimating these parameters is called *Extrinsic*

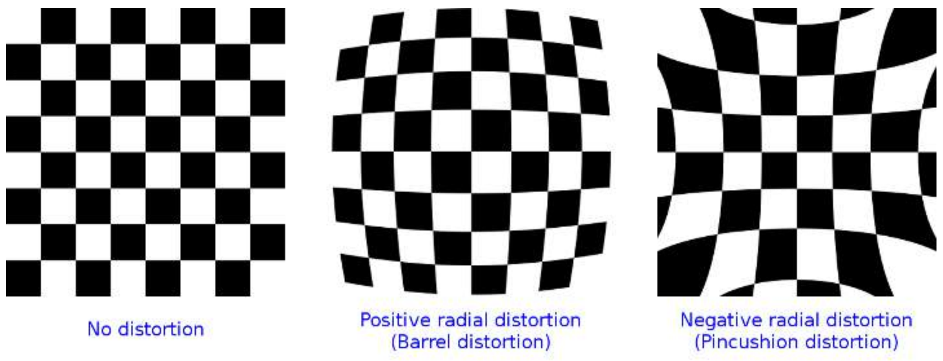

No distortion
Positive radial distortion (Barrel distortion)
Negative radial distortion (Pincushion distortion)

Fig. 3.2 Two common types of radial distortion: barrel distortion (typically $k_1 > 0$ and pincushion distortion (typically $k_1 < 0$). source: [22]

Calibration. The usual distortions caused by lenses are *radial distortion* or slight *tangential distortion*. The pinhole camera model is extended with distortion as follow:

$$\begin{aligned}
\begin{bmatrix} x \\ y \\ z \end{bmatrix} &= R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\
x' &= x/z \\
y' &= y/z \\
x'' &= x' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2\, p_1 x' y' + p_2\, (r^2 + \, 2x'^2) \\
y'' &= y' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + \, p_1\, (r^2 + \, 2y'^2) + \, p_2 x' y' \\
where\ r^2 &= x'^2 + y'^2 \\
u &= f_x * x'' * c_x \\
v &= f_y * y'' * c_y
\end{aligned} \tag{3.3}$$

$(k_1, k_2, k_3, k_4, k_5, k_6)$ are radial distortion coefficients, while (p_1, p_2) tangential distortion coefficients. The figure 3.2 illustrates which type of distortion is generated given distortion coefficients with different signs. The **Barrel Distortion** resembles the so-called *fisheye effect* caused by fisheye lens. As it's clearly evident, at the center the spatial resolution is at maximum and reduces radially outwards from the center of the lens.

The perspective projection of a pinhole camera can be also treated as a hemisphere image projected onto a plane:

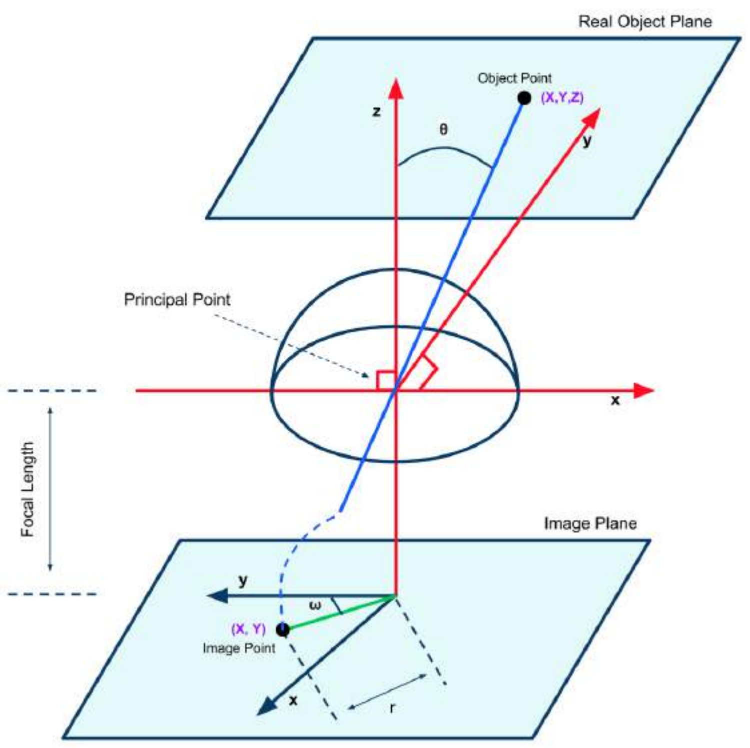


Fig. 3.3 Illustration of the fisheye lens camera projection model. source: [25]

$$r = f\,tan\theta \tag{3.4}$$

where θ is the angle between the optical axis and the incoming ray, r is the distance between the image point and the principal point and f is the focal length. In this model all coordinates are transformed from cartesian to polar in the image plane as follow:

$$\begin{aligned} r &= \sqrt{(x-x_p)^2+(y-y_p)^2} \\ \theta &= arctan(\frac{y-y_p}{x-x_p}) \end{aligned} \tag{3.5}$$

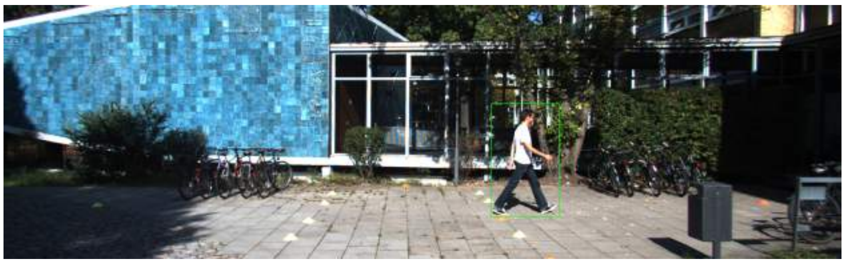
For fisheye lens, the so-called f-theta ($r = f\theta$) mapping is commonly used (also called *equidistance projection*). The general projection model is formalized in [40] as a polynomial with the form:

$$r(\theta) = f\cdot\theta(1+a\theta+b\theta^2+c\theta^3+d\theta^4+e\theta^5...) \tag{3.6}$$

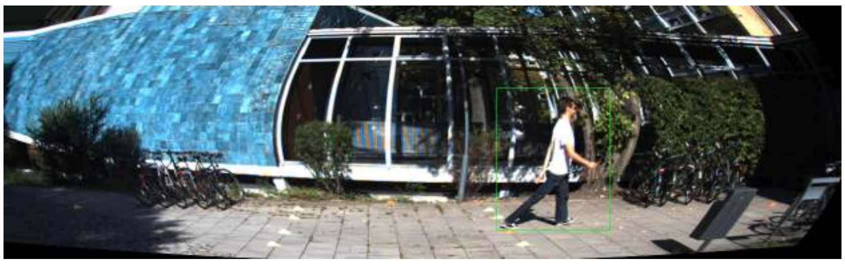
where θ is the angle between the optical axis and the incoming ray, r is the distance between the image point and the principal point, $(a,b,c,d,e,..)$ the distortion coefficients and f is the focal length as depicted in figure 3.3. With this formulation, as θ increases r, escalates also with a factor determined by the distortion coefficients, as a result points from the center to edges appear to keep distancing gradually. This property of fisheye lens is advantageous, since it covers wider field of view (FOV) around 180° unlike conventional lenses $100^\circ - 130^\circ$ (The pinhole camera model can not handle a zero value on the z-coordinate, the axis parallel to the optical axis). But this wide view-angle is achieved at the cost of losing precision in transformation, consequently straight lines appear as curved lines instead.

3.1.1 Difficulties by Object Detection

Fisheye camera has been gaining interest due to its better angle of coverage, which makes it suitable for different sorts of application, but it comes at a price. Assuming we have a fixed size frame, capturing a wider scene inside the frame would mean that each object in the scene will have smaller proportion of the image, since it's represented with lower amount of pixels than the case if it was taken with one standard pinhole camera. Furthermore, the distortion intensely violates the prime structural information of the objects in an image. As an example we pick one image from KITTI [7] and compare it before and after distortion 3.4.



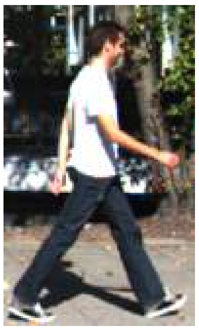
(a) Original Image



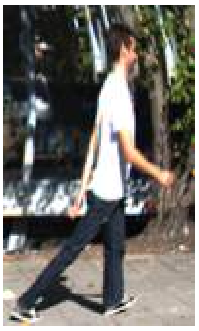
(b) Distorted Image

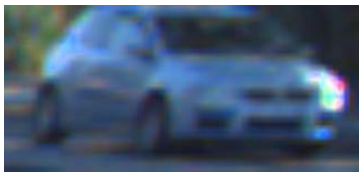
Fig. 3.4 An example from KITTI dataset. (a) in the original form and (b) after applying barrel distortion. In remapping *Cubic Interpolation* is used.

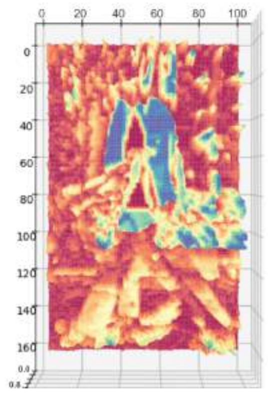
the object of interest is the pedestrian inside the green box. Since the pedestrian is relatively near to the center of the image (Principal Point) it is magnified as expected. As it can be seen, pixels of the object are translated with different θ to new positions causing the distortion. On the side the distortion generates rotation and dislocation, which has an impact on the correlation between location and color value (features) of pixels. As a consequence the distorted image differs strongly from the original one. In order to demonstrate this difference the figure 3.7b shows the error at each location inside the green bounding box in the example image 3.4b (pixel-wise variation in sum of the normalized RGB color). The higher the error, the more uneven and bumpy the error landscape will be. For the sake of comparison figure 3.7a presents another example but with lowest distortion intensity. Two commonly used metrics for image comparison are *Mean Square Error* (MSE) and *Structural Similarity Index* (SSIM) [41], which are calculated for both examples in figure 3.7. Probably MSE or SSIM is not the most accurate metric to assess to what extent does the distortion modify an image but undoubtedly such bold discrepancy (MSE: **98.54**, SSIM: **0.03**) could be problematic, when it comes to the object detection based on detectors which cling tightly to the structural features of the image (Edges, shape, color, texture, etc.).


(a) Original

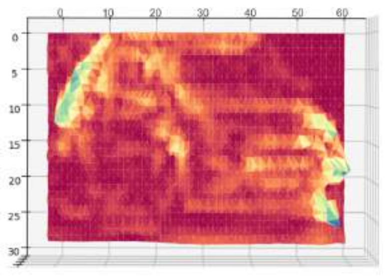
(b) Original


(a) Distorted


(b) Distorted



(a) MSE: **98.54**, SSIM: **0.03**

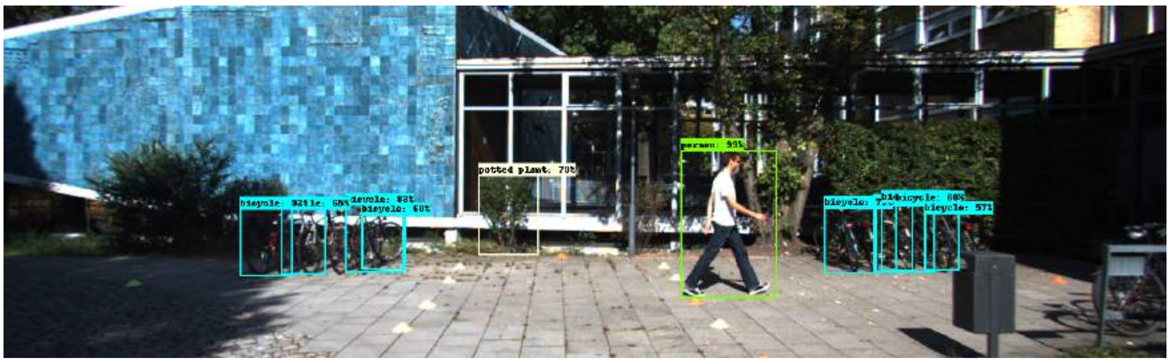


(b) MSE: **63.95**, SSIM: **0.49**

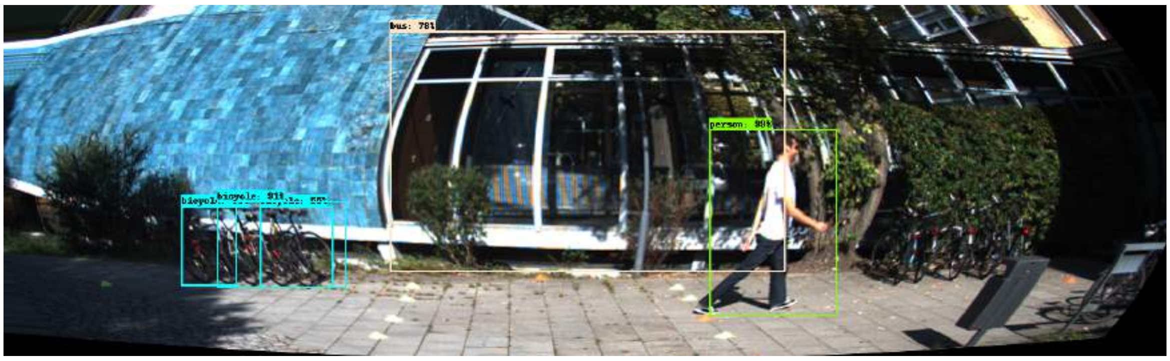
Fig. 3.7 Error landscape between original and distorted object. Highest error value is represented in blue in z-axis, lowest in red. (a) an object affected by intense distortion, (b) an object with slight distortion.

To bring this to proof, we challenge one robust two-stage detector, that is believed to has highest accuracy than all single-stage detectors, Faster-RCNN [27]. In figure 3.8 we see the Faster-RCNN detections on the example image of KITTI. From the first image 3.8a Faster-RCNN detects easily 10 objects including the bicycles at both right and left side of the image, the pedestrian and a plant at the center. Surprisingly when the barrel distortion is applied 3.8b, not only Faster-RCNN misses all bicycles at the right side, it even mistakes the windows at the center to be a bus. As it is obvious, the distortion heavily irritates the detector and makes it uncertain to decide, as the confidence scores of the detector lowers. To investigate further the magnitude of influence of the fisheye distortion we calculate the mean average precision (mAP) of Faster-RCNN over 200 randomly selected images from evaluation set of KITTI. It became apparent the performance of Faster-RCNN is strongly affected by the distortion and as an outcome its mAP dropped by **0.11** from **0.30**.

Having proven the issue of fisheye lens in object detection, this question rises, how to fix it? Logically it could be possible to use the inverse of the same mapping and transformation model to undistort images. In fact what we need to do is the mapping a portion of the surface of a sphere to a flat image. This is called **Rectilinear Projection** (or tangent-plane), and can be envisioned by imagining placing a flat piece of paper tangent to a sphere at a single point, and illuminating the surface from the spheres center. This method is widely used to rectify and straighten distorted images, but this correction is not flawless. The undistortion urges us to cut out considerable amount of pixels from the edges, to get a full straight equiangular quadrilateral shaped image, the demonstration in figure 3.9 helps to clarify this. This might



(a) Detection on image without distortion.



(b) Detection on image with distortion

Fig. 3.8 Impact of the distortion on detection performance of Faster-RCNN [27]. Image source: [7]

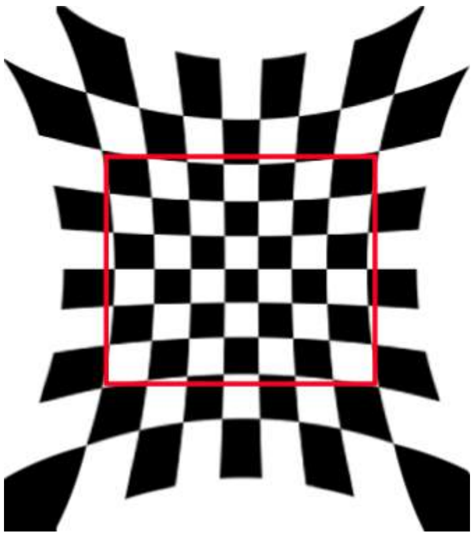


Fig. 3.9 Fisheye distortion correction by Rectilinear Projection.

be thoughtlessly disregarded in use-cases where the central region is more of interest than the borders, but while driving every moving object specially at closer distance matters and needs to be detected and localized in order to avoid potential accident, therefore cropping out the bordering pixels and leaving the potentially critical information out of consideration is not a desired solution. Besides adding extra filter to perform the correction is computationally somewhat expensive and put latency in real-time vision perception pipeline. In an ideal situation the object detector should be resistant against any sort of distortion and be capable of detecting without a hitch even in the presence of deceptive distortions.

This creates the need of having object detector that is trained specially for such scenario. To achieve this we make use of augmentation techniques in this thesis for optimizing a detector on fisheye images. We will see in chapter 7 that training a CNN with distorted images shows promising results and improvements.

Chapter 4

State-of-the-Art

4.1 Object Detection

In our daily life visual perception is the most complicated task performed constantly in our brain. It refers to skills developed from experience relating size, shape, texture, location and the distance and allows us the interpretation of what we see. We examine a scene consisting multiple objects, identify the objects and judge their significance, in other words we localize the objects and classify them. Similar to our visual perception in computer vision we deal mostly with images containing more than only one object and we want to find the location of these object of interest and classify them. We call this task *Object Detection*, which differs from classifying a single image entirely. The figure 4.1 illustrate the difference between object detection and image classification.

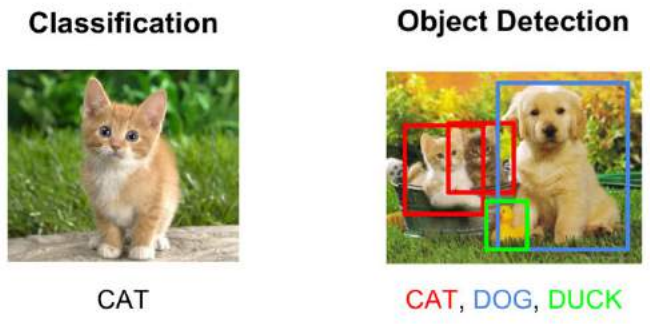


Fig. 4.1 object detection vs Image classification. source: [19]

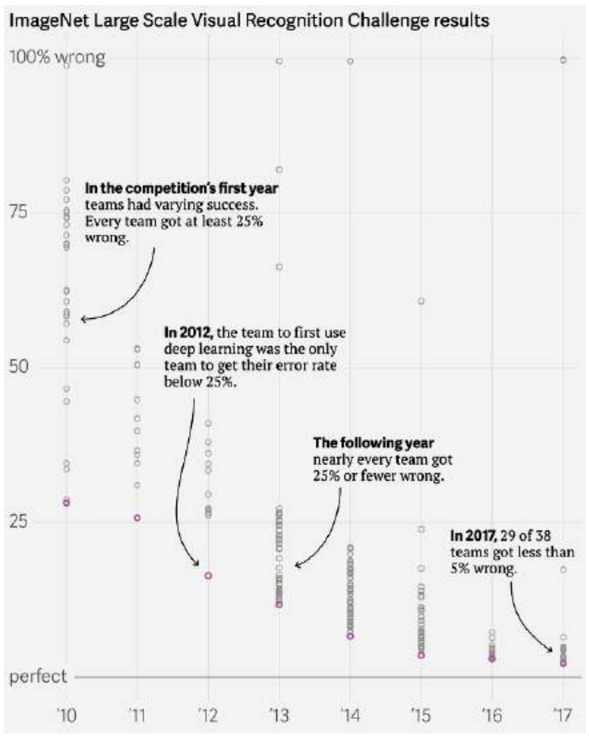


Fig. 4.2 error rate history on ImageNet. source: [30]

4.1.1 Competitions

The primary goal of competitions is to encourage people to push the boundaries of a particular task further and come up with innovative ideas. In recent years many image classification and object recognition competitions have given the computer vision community the opportunity to put their knowledge and skills into competition to push the envelop in object recognition and detection.

One of the first huge competitions was *ImageNet challenge* presented at the 2009 Conference on Computer Vision and Pattern Recognition (CCVPR). In ImageNet Large Scale Visual Recognition Challenge (ILSVRC) researchers evaluate their algorithms on the given dataset and compete to achieve higher accuracy on several visual recognition tasks. Although 29 of 38 competitors in 2017 achieved accuracy with less than 5% loss, but in 2012 was a breakthrough and start of the deep learning revolution with *AlexNet*, a convolutional neural network that achieved a top-5 error of 15.3%, more than 10.8 percentage points ahead of the runner up. It is interesting to see how fast the error rate has been falling down in ImageNet competition since 2011, The Figure 4.2 shows best result per team and up to 10 entries per year.

ILSVRC was a key event, that drew CV community's attention to application potential of deep learning in image processing and started a new era in this field. Consecutively more competitions had been created, some of the most popular competitions are as follow:

- **PASCAL VOC (Visual Object Classes)**: Everingham et al. [5] provided standardized image datasets for object class recognition and ran challenges evaluating performance on object class recognition from 2005 till 2012. (Supported by the EU-funded PASCAL2 Network of Excellence on Pattern Analysis, Statistical Modeling and Computational Learning)
- **COCO (Common Object in Context)**: Lin et al. [20] provides a large-scale object detection, segmentation, and captioning dataset and organizes several challenges. (Sponsored by Microsoft, Facebook, Mighty AI and CVDF)
- **KITTI**: Geiger et al. [7] focuses on stereo, optical flow, visual odometry, 3D object detection and 3D tracking and develop novel challenging real-world computer vision benchmarks. (A project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago)
- **Open Image Dataset**: a new object detection challenge, provides large dataset with 9 million images containing complex scenes with several objects. (Organized and licensed by Google)

All the mentioned challenges make useful datasets available for everybody. There is a wide range of diversity and suitability among the datasets and qualitatively they are considerably variational. This make the choice of a suitable dataset for training supervised algorithm tricky. In the next section we dip into the publicly available datasets, which suite the task of real-time object detection specially for the autonomous vehicle, where object of interests mostly appear on the street and are momentous for decision-making in self-driving car, like pedestrians, autos, traffic lights, etc.

4.1.2 Datasets

Besides the mathematical structure of a supervised object detector the general performance depends strongly on the quality of input data, on which the detector will be trained. In the evaluation section we will see how different datasets cause fluctuation in the performance of CNN, here when we say performance we mean the overall performance, in order to exam a detector appropriately several evaluation metrics should be taken into consideration like Precision, Recall, mean average precision (mAP), etc. At the moment we familiarize ourself

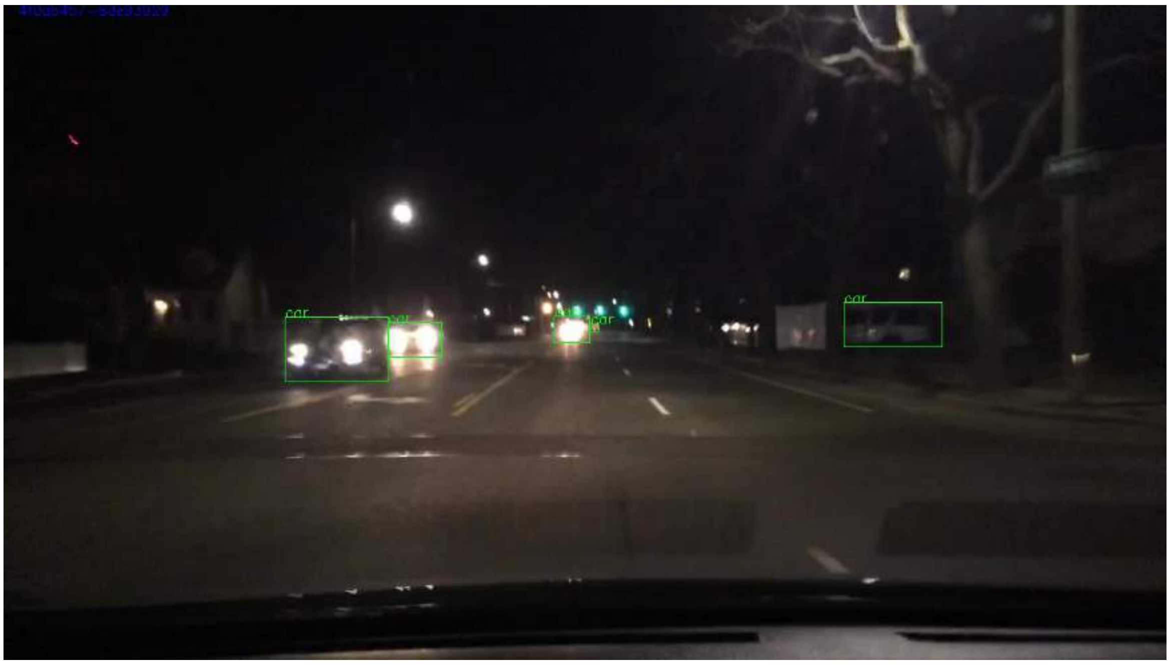
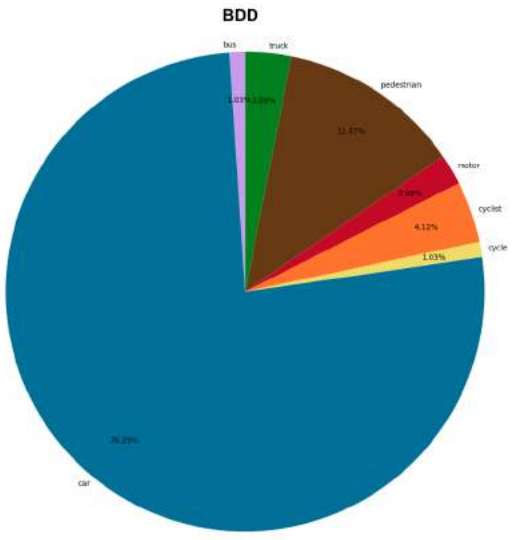


Fig. 4.3 misrepresentative objects in BDD dataset.

with the benchmarks, which are specifically collected for 2D Object detection and investigate their advantages and drawbacks. The fact is all datasets are flawed. That is why choosing the right dataset is the most crucial aspect in supervised learning, this rises the questions which dataset suits our problem at best? which criteria should be considered for assessing a dataset? answering these questions requires examination and analysis, but quantifying them is tedious. In environment perception for autonomous cars particular properties determine the overall qualification of a dataset:

- **Representativeness:** Basically kernels in CNNs perform feature extraction and seek patterns in images. The higher the representation accuracy of objects in a dataset, the better kernels can extract features and therefore a CNN will be able to distinguish objects from each other with higher precision. Feeding a CNN with misrepresentative objects increases classification error rate. Although if we consider this as a kind of noise in dataset it could help to improve the generalization of CNN but more in an uncontrollable manner, on the other hand preprocessing techniques are sufficiently assistant on CNN generalization and overfitting avoidance. Figure 4.3 shows an example of inaccurate and fallacious object representation from BDD Yu et al. [44]. Apart from the reflection of the dashboard on the front glass, which is sort of a noise, in this picture 5 bounding boxes are representing supposedly cars, although those two cars in the center of the picture lie at the horizon of the scene in a pretty long distance,



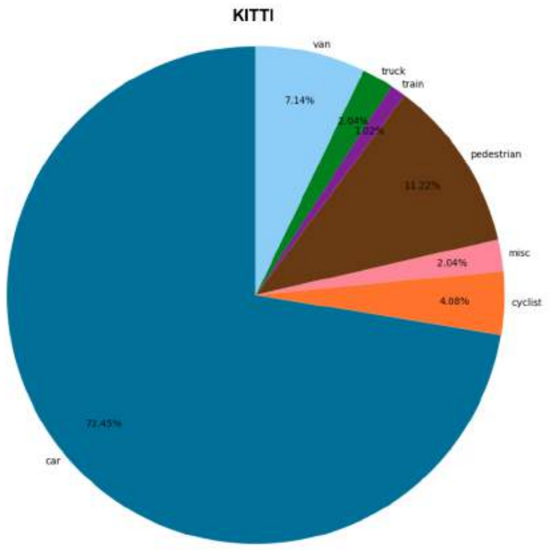


Fig. 4.4 Portion of each class in KITTI and BDD.

which makes them at this particular moment very tiny and irrelevant to be detected, besides they are barely representing any clearly distinguishable optical features of an vehicle, rather they actually resemble more likely an distal indistinct traffic light.

- **Balance:** Uniform distribution of classes in a dataset affects the accuracy of CNN over all classes. In other words equal number of instances in each class makes a CNN put concentration on all classes evenly. It is noteworthy, that accuracy may be an indistinct evaluation metric, it may be better to avoid the accuracy metric in favor of other metrics such as precision and recall for a better class specific evaluation, we will discuss this in more detail in evaluation section. Dataset imbalance is a common syndrome in data science, that has intense consequences like overfitting and biased behavior. Fortunately it is usually curable. Its treatment requires a couple of refinement in preprocessing phase. It's possible to even up the quantity of classes by over-sampling the under-represented classes or conversely down-sample the over-presented classes, this could induce overfitting, therefore must be performed carefully. The other possibility is to synthesize samples to heighten the minority classes. Figure 4.4 demonstrates the proportion of classes in KITTI Geiger et al. [7] and BDD Yu et al. [44] dataset, the main 2 datasets used for the experiments in this thesis. As can be seen from the diagram quantity of *Car* instances overbalance the whole dataset by taking nearly 0.8 of the datasets, consequently detectors fed with such dataset are more likely better at detecting cars than other minor classes like pedestrian, bus or cyclist.

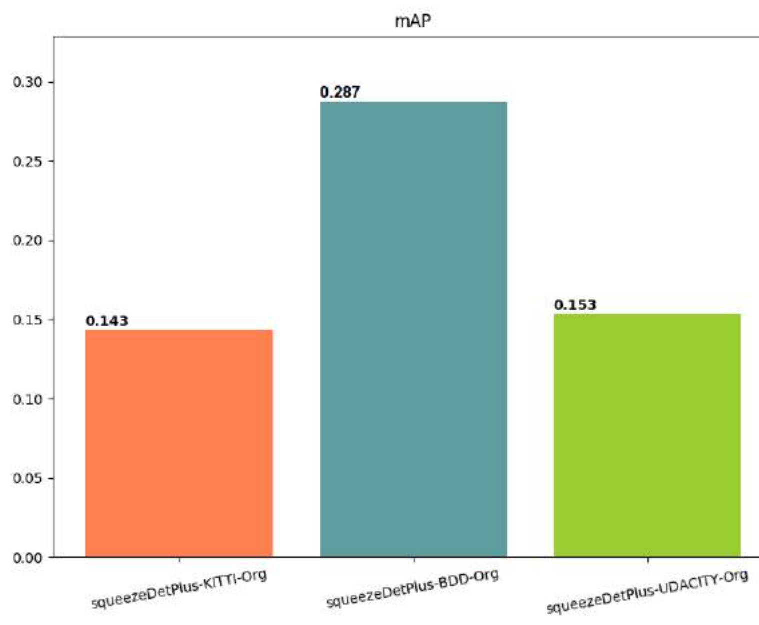


Fig. 4.5 squeezeDetPlus mAP calculated on fisheye images trained with KITTI, BDD and Udacity datasets.

- **Environmental Dynamic:** For autonomous driving applications invariance in illumination, background, daytime, weather conditions, scene, geographic distribution, type of vehicle and etc limits a CNN to specific domain characteristics. A trained CNN with such data easily fails in real world applications and lack of dynamic due to overfitting hinders the CNN to perform adaptively. Hence a fine dataset should provide diverse driving scenes.

- **Annotation Accuracy:** In 2D object detection bounding boxes are *Region of Interests* (ROI) that define the borders of an object under consideration. They contain informations about objects, therefore a falsely drawn bounding box gives irrelevant informations into the network, that are actually background informations but are being taken into account as ROI related informations. This is utterly misleading and could push the training in wrong direction.

Just to depict the influence of the named properties above on detector performance we compare in Figure 4.5 the mAP of the squeezeDetPlus [43] trained with 3 different datasets, KITTI, BDD and Udacity respectively from left to right on the diagram. Surprisingly there is significant difference to see although all named datasets are created for the same use case. BDD improves the precision of squeezeDetPlus considerably from 0.143 to 0.313. BDD contains $\approx$ 900.000 instances, which is roughly 10 times more than KITTI, also with higher illumination variance.

4.2 CNN for Object Detection

It is indeed beyond question that the birth of convolutional neural networks has empowered the CV community and they have been prevailing since the novel network AlexNet was proposed in 2012. AlexNet or similar standard convolutional networks followed by a fully connected layer are designed to solve the image classification problem, while, to put it simply, they map the input image to a constant number of probabilistic outputs. This type of CNN lacks the capability to classify and localize multiple objects in input images, since the length of the output layer is variable. Object detection is composed of two tasks:

- Localization
- Classification

In order to equip a CNN to solve object detection problem, other components need to be attached to the body of the network. In other words the mapping is now from a field with dimensionality of $[Image\_height, Image\_width, Image\_channel]$ to a field $[Number\_of\_objects, Class, Coordinates]$. A naive approach to solve the object detection problem would be to apply a CNN on different ROIs and classify the presence of the object within that region. Objects might have different spatial locations and aspect ratios, as a result number of ROIs grows rapidly, this could be computationally expensive and not applicable in real-world situation. Therefore new network structure have been developed to tackle this problem. In spite of the variety in CNN-based object detector's structures they all follow more or less the same concept. Next we investigate the architectural differences between CNN-based object detectors.

4.2.1 Architectural Differences

Recent CNN-based object detector can be categorized into single-stage and two-stage detectors. The single-stage detector usually targets a sweet-spot very fast speed and reasonably good accuracy. The two-stage detector divides the task into two steps: the first step (*body*) generates many proposals, and the second step (*head*) focuses on the recognition of the proposals. Usually, in order to achieve the best accuracy, the design of the head is heavy. The speed-accuracy trade-off (SAT) seems an inescapable property of decision making in the early stages of designing a network, obviously based on the use case one is favored over the other. Taking balance between them might be a reasonable solution. we compare state-of-the-art of both categories in next segment.

4.2.1.1 Two-stage Detectors

Regional CNN (R-CNN) Girshick et al. [8] was an early application of CNNs to object detection. R-CNNs use a pre-trained network as basis for feature extraction, also referred to as *Transfer Learning*. Choosing the base network is one of the factors, that influences the performance of a R-CNN, there is no real consensus on which network architecture is best. The original Faster R-CNN used ZF Zeiler and Fergus [46] and VGG Simonyan and Zisserman [32] pre-trained on ImageNet but since then there have been lots of different networks with a varying number of weights. The first version of R-CNN addressed the problem of redundant region selection by using *selective search* Uijlings et al. [39], an algorithm to generate possible object locations. The workflow of selective search is as follow [39]:

1. At the initialization stage, apply Felzenszwalb and Huttenlocher [6] graph-based image segmentation algorithm to create regions to start with.
2. Use a greedy algorithm to iteratively group regions together:
   - First the similarities between all neighboring regions are calculated.
   - The two most similar regions are grouped together, and new similarities are calculated between the resulting region and its neighbors.
3. The process of grouping the most similar regions (Step 2) is repeated until the whole image becomes a single region.

Selective search generates 2000 region proposals. These 2000 candidate region proposals are warped into a square and fed into a CNN that yields a 4096-dimensional feature vector as output.The body extracts the features from the image, then the extracted feature will be passed into a *Supper Vector Machine* (SVM) to classify the presence of the object within candidate region proposals. Additionally the network predicts four values which are the offset values between ground-truth and proposed region, this is needed to adjust the bounding box of the region proposals. Figure 4.6 shows the flow of region proposals through R-CNN.

The speed bottlenecks of R-CNN is first selective search and then the exhaustive classification of 2000 region proposals per image, that makes R-CNN impractical for real time tasks, besides the region proposal is solid since the selective search is just an overhead to the CNN and will not be trained. This implies that the network dose not learn directly the

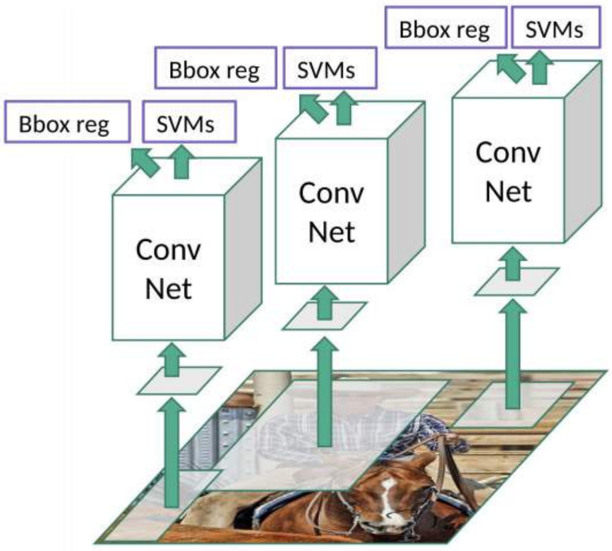


Fig. 4.6 R-CNN. source: [8]

locations of objects but rather is said where to look. After R-CNN Ross Girshick (Author of R-CNN) has attempted to improve the performance of its network and build a faster detector. The resulting work was Faster R-CNN Ren et al. [27].

Faster R-CNN: region proposal is dependent on features of images, that are calculated during the forward pass into base CNN for classification. To remove the selective search Faster R-CNN reuses the extracted features to generate region proposals and adds a Fully Convolutional Network on top of the features of the CNN creating what's known as the *Region Proposal Network*.

As shown in figure 4.7 now this new structure of Faster R-CNN contains two networks:

- Feature Extractor (CNN)
- Region Proposal Network (RPN)

This is why we call these type of networks *Two-stage Detector*. The RPN predicts objectiveness at different regions by sliding a window over the features of the CNN. The sliding window does not have dynamic size, therefore to cover all objects with different sizes and aspect ratios Faster R-CNN uses $k = 9$ so called anchors at each position in 3 different size and aspect ratio. Let's say we have image $600 * 800$, stride 16 there will be $39 * 51 = 1989$ positions and $1989 * 9 = 17901$ anchor boxes. Each anchor box contains 6 parameters, 4 parameters representing bounding box of anchor and two labels the probability of an anchor being background or foreground. Obviously not all the anchor boxes represent an object, anchor boxes need to be associated with ground-truths and only the responsible ones (the

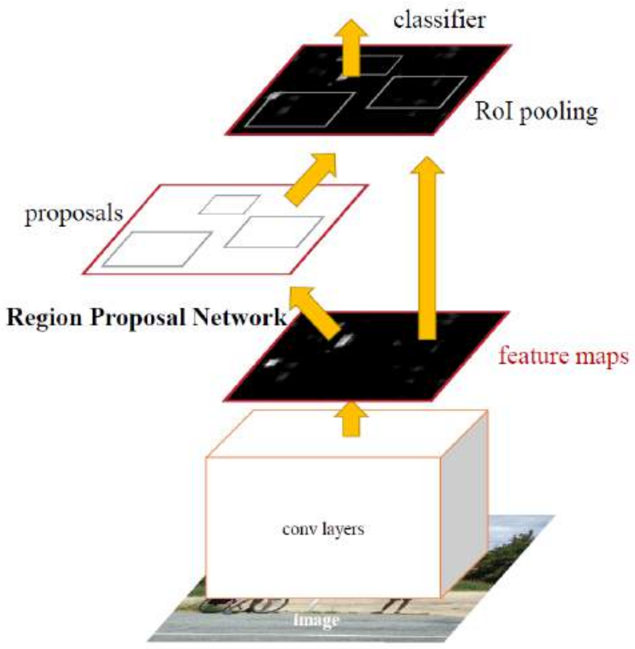


Fig. 4.7 Faster R-CNN is a single, unified network for object detection. source: [27]

anchor boxes truly representing an object/ground-truth) can be brought into loss function. To ensure that a binary class label is assigned to each anchor box. Positive label for anchors with the highest Intersection-Over-Union IoU overlap with a ground-truth box or an anchor that has an IoU overlap higher than 0.7 with any ground-truth box, negative label to a non-positive anchor if its IoU ratio is lower than 0.3 for all ground-truth boxes. Anchors play a key role in Faster R-CNN, because of this multi-scale design based on anchors, Faster R-CNN can simply use the convolutional features computed on a single-scale image. Based on the common shape of objects of interest the size and aspect ratio need to be fine-tuned. For instance to detect pedestrians very short, very big or square boxes are unimportant and should be excluded from training, which affects the accuracy of the network detection and reduces the training time due to the lower number of anchors.

As illustrated in figure 4.8 in next level the feature maps are passed into a bounding box regressor and a classifier. The objective of RPN is to minimize the loss of both bounding box regressor and classifier. The multi-task loss function is defined as:

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i,p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i,t_i^*) \tag{4.1}$$

Here, i is the index of an anchor and p_i is the predicted probability of anchor i being an object. The ground-truth label p_i^* is 1 if the anchor is positive, and is 0 if the anchor is negative. t_i is a vector representing the 4 parameterized coordinates of the predicted bounding

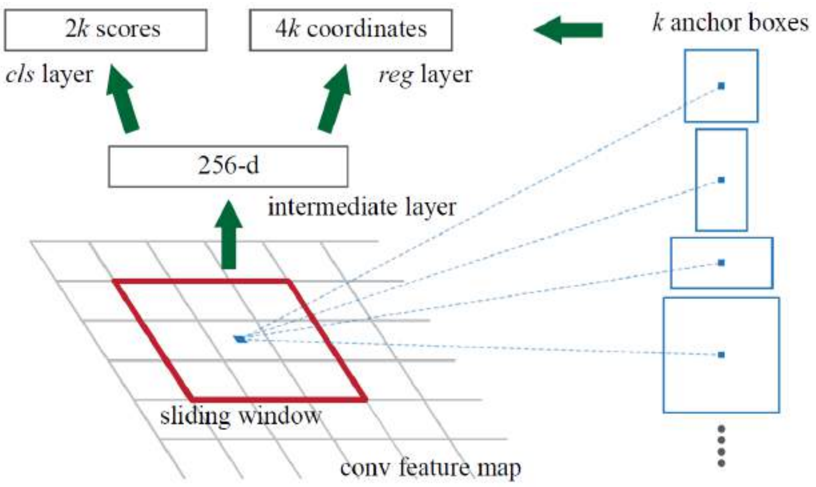


Fig. 4.8 Region Proposal Network (RPN). source: [27]

box ($t = (t_x, t_y, t_w, t_h)$), and t_i^* is that of the ground-truth box associated with a positive anchor. The classification loss L_{cls} is log loss over two classes (object vs. not object). The regression loss is a smooth L_1 defined as follow:

$$L_{reg}(t, t^*) = \sum_{i \in \{x,y,w,h\}} smooth_{L_1}(t_i - t_i^*), \tag{4.2}$$

in which

$$smooth_{L_1}(x) = \begin{cases} 0.5x^2 & if\,|x| < 1 \\ |x| - 0.5 & otherwise, \end{cases} \tag{4.3}$$

Both terms are normalized by N_{cls} and N_{reg} and weighted by a balancing parameter λ.

In the next step the output of RPN and the feature map go through a for object detection specialized max-pooling layer, called *RoI Pooling*. Having bunch of object proposals from RPN it reuses the existing feature map to classify each proposed bounding box. For every region of interest from the input list, it actually takes a section of the input feature map that corresponds to it and scales it to some pre-defined size, the extracted feature maps are used to classify them into a fixed number of classes. Its purpose is to speed up the training and yields fixed-size feature maps from non-uniformed input size, which allows the network to be trained in an end-to-end manner. The scaling in RoI pooling is performed by:

1. Dividing the region proposal into equal-sized sections
2. Extracting the largest value in each section, like a normal max-pooling layer
3. Put these max values together

We could summarize the whole Faster R-CNN training procedure as follow [42]:

1. Pre-train a CNN network on image classification tasks.
2. Fine-tune the RPN (region proposal network) end-to-end for the region proposal task, which is initialized by the pre-train image classifier. Positive samples have $IoU > 0.7$, while negative samples have $IoU < 0.3$.
   - Slide a small n x n spatial window over the conv feature map of the entire image.
   - At the center of each sliding window, we predict multiple regions of various scales and ratios simultaneously. An anchor is a combination of (sliding window center, scale, ratio). For example, $3scales + 3ratios => k = 9$ anchors at each sliding position.
3. Train a Fast R-CNN object detection model using the proposals generated by the current RPN
4. Then use the Fast R-CNN network to initialize RPN training. While keeping the shared convolutional layers, only fine-tune the RPN-specific layers. At this stage, RPN and the detection network have shared convolutional layers!
5. Finally fine-tune the unique layers of Fast R-CNN
6. Step 4-5 can be repeated to train RPN and Fast R-CNN alternatively if needed.

Figure 4.9 illustrates the model design of all three R-CNNs. Replacing the selective search with RPN and reusing the extracted feature improved the speed of R-CNN significantly. Table 4.1 shows the timing on a k40 GPU in millisecond with PASCAL VOC 2007 test set. It stats that a deeper base network like VGG slows the network down.

So far, all the methods discussed handled detection as a classification problem by building a pipeline where first object proposals are generated and then these proposals are send to classification/regression heads. However, there are a few methods that pose detection as a regression problem. Two of the most popular ones are YOLO [26] and SSD [21]. These detectors are also called single-stage detectors.

base model	system	conv	proposal	region-wise	total	rate
VGG	SS + Fast R-CNN	146	1510	174	1830	0.5 fps
VGG	RPN + Fast R-CNN	141	10	47	**198**	**5 fps**
ZF	RPN + Fast R-CNN	31	3	25	**59**	**17 fps**

Table 4.1 Speed comparison between R-CNNs on PASCAL VOC 2007 using k40 GPU. The values are extracted from the original Fast R-CNN paper.

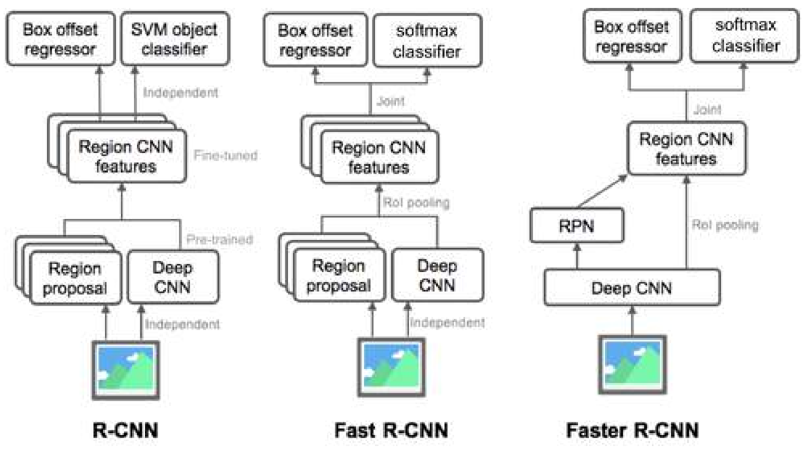


Fig. 4.9 R-CNN models design. source: [42]

4.2.1.2 Single-stage Detectors

Single shot detectors treat the object detection task as a unified regression problem, different from the R-CNN family. They perform classification and localization in one shot, therefor they are usually categorized as real-time detectors. Yolo Redmon et al. [26] is single neural network that predicts bounding boxes and class probabilities directly from full images in one evaluation. Since the whole detection pipeline is a single network, it can be optimized end-to-end directly based on detection performance. The unification of classification and localization problem has advantages and drawbacks, we take a deeper look into single shot family.

Yolo: considering CNN as a non-linear classifier, it has the capability to solve more complex problem than linear classifier like Support Vector Machine (SVM). A standard CNN takes pixel values and finds hyperplanes to separate images, adding few more variables for bounding box coordinates in equation shouldn't cause unsolvable computational complexity. Nevertheless a CNN is still technically able to find numerically an accurate approximation. This is the main idea of Yolo, re-purposing a classifier to perform object detection!

The last two versions of Yolo make use of only convolutional layers, making them *fully convolutional networks* (FCN). FCNs are translation invariant and built only from locally connected layers, such as convolution, pooling and up-sampling. This allows the network to manage different input size. Although Yolov3 uses no form of pooling, instead a convolutional layer with stride 2 is used to down-sample the feature maps. This helps in preventing loss of low-level features often attributed to pooling. Yolov3 has 75 convolutional layers, with skip connections and up-sampling layers. Generally Yolos divide each image into a grid of $S*S$ (as shown in figure 4.11) and each grid cell contains N bounding boxes and a confidence about the object existence within a bounding box. Each bounding box has the following attributes:

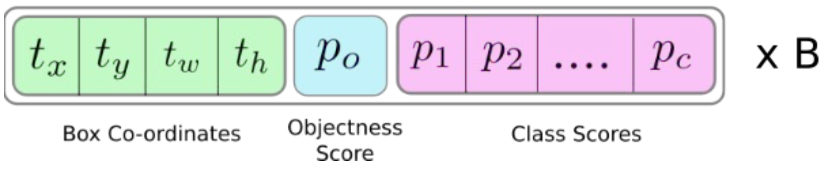


Fig. 4.10 Attributes of bounding box, B is number of the anchors in a cell. source: [18]

The network down-samples the image by a certain stride. The output size of each layer only depends on the stride factor. For instance having an input image $416 * 416$ and network stride of 32, the output will have the size of $13 * 13$. In order to determine object position we should assign it to an anchor box, similar to R-CNNs the association between ground truths and anchors are needed. In Yolo the cell, that contains the center of an object/ground truth is chosen to be the one *responsible* for predicting the object, in other words the one with the highest IoU. The height and width of the bounding boxes are still missing! the network learns the offsets between anchors and ground truth boxes to predict the size of the bounding boxes applies the sigma function to constraint its possible offset range. The following formula describes how the network output is transformed to obtain bounding box predictions:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \end{aligned} \tag{4.4}$$

where (t_x, t_y, t_w, t_h) are output of the network. (b_x, b_y, b_w, b_h) are the predicted bounding box. (c_x, c_y) the top-left coordinates of anchor and (p_w, p_h) width and height of the anchor. The center coordinates prediction are passed through sigmoid function to keep them between 0 and 1. The center coordinates are relative to the top left corner of the grid cell that is predicting the object, normalized by the dimensions of the cell from the feature map. As illustrated in figure 4.10 p_o represents the probability that an object is contained inside a bounding box, since p_o is also output of sigmoid it can be interpreted as probability. The class confidences $p_1, p_2, ..., p_c$ are obtained as follow:

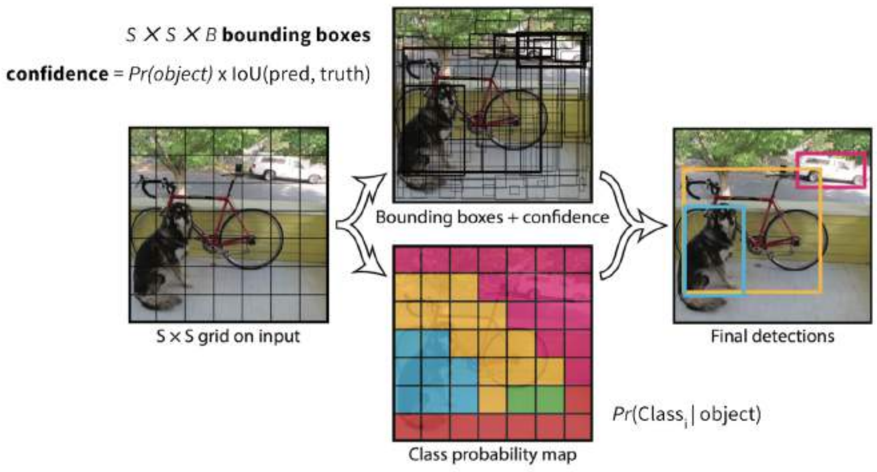


Fig. 4.11 Depiction of Yolo workflow. source: [26]

$$\begin{aligned}
box\ confidence\ score &\equiv P_r(object).IoU \\
conditional\ class\ probability &\equiv P_r(class_i|object) \\
class\ confidence\ score &\equiv P_r(class_i).IoU \\
&= box\ confidence\ score \times conditional\ class\ probability
\end{aligned} \tag{4.5}$$

where $P_r(object)$ is the objectness score. IoU is IoU between the predicted box and the ground truth. $P_r(class_i|object)$ is the probability the object belongs to $class_i$ given an object is presence. $P_r(class_i)$ is the probability the object belongs to $class_i$.

with this we come to the Loss Function of Yolo, which is sum-squared error between the predictions and ground truth. The Loss Function composes of:

- Classification Loss
- Localization Loss
- Confidence Loss

the **Classification Loss** at each cell is squared error of the conditional probability for each class:

$$\sum_{i=0}^{S^2} \mathbb{I}_i^{obj} \sum_c (p_i(c) - \hat{p}_i(c))^2 \tag{4.6}$$

where $\mathbb{I}_i^o bj$ is equal to 1 if an object appears in cell i, otherwise 0. $\hat{p}_i(c)$ denotes the conditional class probability for class c in cell i.

In **Localization Loss** the error in location of bounding boxes and size of them is measured, and of course only the responsible boxes are involved in the equation. As shown blow the error is the distance between vectors of center of predicted bounding box and ground truth and the distance between the width and height of them:

$$\begin{aligned}&\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{obj}\left[(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2\right]\\&+\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{obj}\left[(\sqrt{w_i}-\sqrt{\hat{w}_i})^2+(\sqrt{h_i}-\sqrt{\hat{h}_i})^2\right]\end{aligned}\tag{4.7}$$

$\mathbb{I}_{ij}^{obj}$ is 1 if the jth bounding box in cell i is responsible for detecting the object, otherwise 0. λ_{coord} is weighting parameter to emphasis more on the bounding box accuracy and embolden the localization loss.

the **Confidence Loss** handles two different situation, one to measure the error when an object is detected in the box, two if an object is not detected:

$$\begin{aligned}&\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{obj}(C_i-\hat{C}_i)^2\\&+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{noobj}(C_i-\hat{C}_i)^2\end{aligned}\tag{4.8}$$

where $\mathbb{I}_{ij}^{noobj}$ is the complement of $\mathbb{I}_{ij}^{obj}$, $\hat{C}_i$ is the box confidence score of the box j in cell i. Number of responsible boxes is significantly lower than boxes that don't contain any object, this causes class imbalance problem (Object vs Non-Object). Consequently the network learns more to detect background data, this is exactly contradictory to the objective of the net. To remedy this, the weighting factor λ_{noobj} suppress the loss over non-object detection (It's usually set to 0.5).

The summation of the equations 4.6, 4.7 and 4.8 is the final loss function. According to [26], "Yolo imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that the model can predict. The model struggles with small objects that appear in groups, such as flocks of birds". Furthermore, convolutional layers down-sample the inputs and decrease the spatial dimension gradually. The less resolution, the harder will the detection of smaller object get. To address this issue Yolov3 adopts a different approach called *Pass-through*. To avoid loss of information, it reshapes $28 \times 28 \times 512$ layer to $14 \times 14 \times 1024$. Then it concatenates with the original $14 \times 14 \times 1024$ output layer. Now we apply convolution filters on the new $14 \times 14 \times 3072$ layer to make predictions (Figure 4.12.

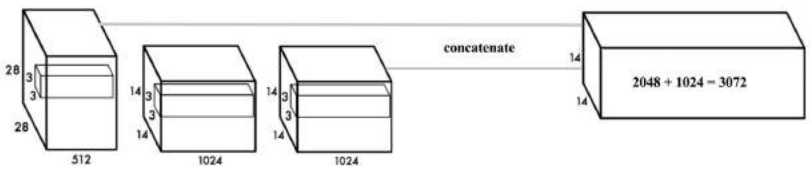


Fig. 4.12 Yolov3 avoids vanishing the features by passing them trough. source: [16]

Yolov2 uses a custom deep architecture darknet-19, an originally 19-layer network supplemented with 11 more layers for object detection. Yolov2 trains the model with multi-scale images. For every 10 batches selects randomly another image size, This acts as data augmentation and forces the network to predict well for different input image dimension and scale. Yolov2 still suffers from inability of detecting small objects due to the loss of fine-grained features. The successor of Yolov2, **Yolov3**, overcomes this issue by performing some modification in the architecture of the net and using several staple elements and methods for information preservation. Yolov3 uses a variant of Darknet, which originally has 53 layer network trained on Imagenet, altogether Yolov3 consists of 106 layer. The entire fully convolutional architecture of Yolov3 is demonstrated in the Figure 4.13.

the main differences of Yolov3 in comparison to its predecessors are as follow:

- Multi-scale architecture: Yolov3 makes detection at three different scales by down-sampling the dimensions of the input image by 32, 16 and 8 respectively. The detection is done by applying 1×1 detection kernels of feature maps at three different stages in the network. The shape of the detection kernel is $1 \times 1 \times (B \times (5 + C))$. B is number of bounding boxes a cell on the feature map can predict, 5 is for four bounding box attribute and one object confidence and C is number of the classes (4.10).

- Skip Connection and Concatenation: Yolov3 uses residual block at the early stages of the network for the feature extraction. After each detection the feature map of detection is subjected to a few convolutional layers before getting up-sampled, thereafter the feature map is depth concatenated with the feature map from previous layer (4.13). This helps the model to preserve the fine-grained features, as a result showing better performance in detecting small objects.

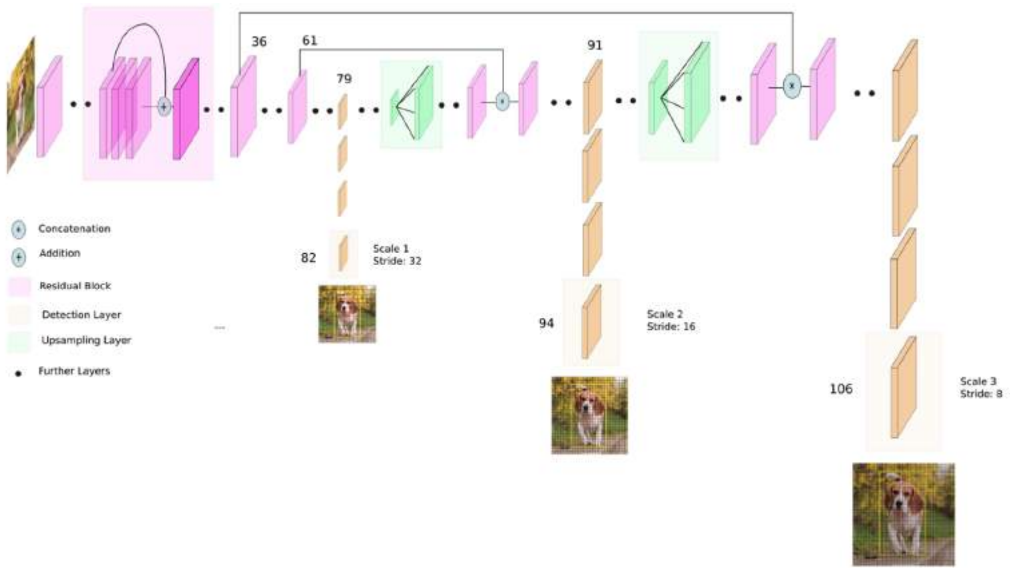


Fig. 4.13 Yolov3 Network Architecture. source: [18]

- Multi-lable classification: Yolov3 replaces the softmax with *Logistic Regression* for prediction. This enables the network to predict multiple classes for an object, since logistic regression dose not rest on the assumption that classes are mutually exclusive, contrary to softmax.
- More anchors: Yolov3 uses 9 anchor boxes, three for each scale. This also means predictions at 3 different scales.At each scale, every grid can predict 3 boxes using 3 anchors. Since there are three scales, the number of anchor boxes used in total are 9, 3 for each scale. Yolov3 predicts 10x the number of boxes predicted by Yolov2.

Table 4.2 shows that Yolov3 performs faster than it predecessors but at the cost of loosing precision. Although the results in 4.2 are taken from original papers and are not absolute reliable measure to evaluate the performance of all the three different variants of Yolo, due to the varied configurations and base model and training dataset used in each model. Yolov3 is one the chosen model to experiment with in this thesis, therefor in evaluation section we will see how Yolov3 performs on distorted images.

SSD: Single Shot Multi-Box Detector Liu et al. [21] is a strong competitor for Yolo. SSD also takes advantage of Multi-scale design like Yolov3. SSD uses VGG-16 Simonyan and Zisserman [32] as truncated base network discarding the fully connected layer followed by convolutional feature layers.

> "These layers decrease in size progressively and allow predictions of detections at multiple scales. Each added feature layer can produce a fixed set of detection

Method	mAP	FPS
YOLO	66.4	21
YOLOv2-544	78.6	40
YOLOv3-608	57.9	51

Table 4.2 Yolo family mAP and FPS comparison on Pascal VOC2007. All timing information is on a Geforce GTX Titan X. The values are extracted from the original Yolo papers.

> predictions using a set of convolutional filters as indicated in Figure 4.14. For a feature layer of size $m \times n$ with p channels, the basic element for predicting parameters of a potential detection is a $3 \times 3 \times p$ small kernel that produces either a score for a category, or a shape offset relative to the default box coordinates. At each of the $m \times n$ locations where the kernel is applied, it produces an output value."

The bounding box offset output values are measured relative to anchor box (Default boxes as named by the authors of SSD) position relative to each feature map location. SSD applies the default boxes to several feature maps of different resolutions. Allowing different default box shapes in several feature maps lets us efficiently discretize the space of possible output box shapes. For training, as commonly performed by other models as well, SSD selects for each ground truth box one box from default boxes by matching each ground truth box to the default box with the best overlap (higher than a threshold 0.5)

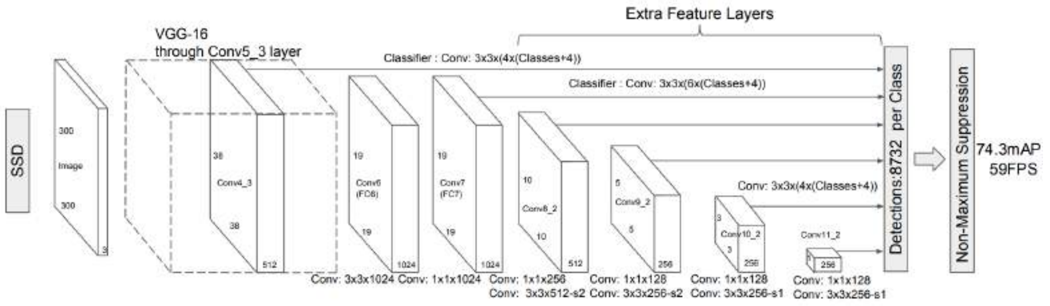


Fig. 4.14 SSD model with a 300 × 300 input size adds several feature layers to the end of a base network, which predict the offsets to default boxes of different scales and aspect ratios and their associated confidences. source: [21]

The overall objective loss function of SSD is a weighted sum of the *Localization Loss* and *Confidence Loss*:

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g)) \tag{4.9}$$

where N is the number of matched default boxes (responsible anchors). If no default box was found ($N = 0$) the whole loss is equal zero. The **Localization Loss** is a *Smooth L1 loss* between the predicted box l and the ground truth box g parameters. Similar to other models, SSD regresses to offsets for the center (cx, cy) of the default bounding box d and for its width w and height h:

$$\begin{aligned}
L_{loc}(x,l,g) &= \sum_{i\in Pos}^{N} \sum_{m\in\{cx,cy,w,h\}} x_{ij}^{k} smooth_{L1}(l_i^m - \hat{g}_j^m) \\
\hat{g}_j^{cx} &= (g_j^{cx} - d_j^{cx})/d_i^w \\
\hat{g}_j^{cy} &= (g_j^{cy} - d_j^{cy})/d_i^h \\
\hat{g}_j^{w} &= log(g_j^w/d_j^w) \\
\hat{g}_j^{h} &= log(g_j^h/d_j^h)
\end{aligned} \tag{4.10}$$

The **Confidence Loss** is the *softmax loss* over multiple classes confidence c:

$$L_{conf}(x,c) = -\sum_{i\in Pos}^{N} x_{ij}^{p} log(\hat{c}_i^p) - \sum_{i\in Pos}^{N} log(\hat{c}_i^o) \tag{4.11}$$

where

$$\hat{c}_i^p = \frac{exp(c_i^p)}{\sum_p exp(c_i^p)} \tag{4.12}$$

The number of negative default boxes outweighs the number of positive ones, to rectify this imbalance SSD sorts the negative default boxes using the highest confidence loss for each default box and pick the top ones so that the ratio between the negatives and positives is at most 3:1, instead of inputting all of them into the loss function.

SSD puts more stress on the mAP than the inference speed, as it can be seen in 4.3 SSD manages to offer a better mean average precision but is still nearly slower than even the first version of Yolo. It's again noteworthy that those results could be misleading, since not all the models are evaluated under the same condition and mAP is highly biased by the number of classes, but at least it gives us a glance of how different architectures show different performances.

Indeed it would not be rational to choose one of Single-stage or Two-stage detectors over the other one as the winner, since choosing a detector depends strongly on the use-case. For real-time application usually accuracy is not preferential and speed is more crucial. Therefore between the introduced state-of-the-art detectors from both families, it seems that Yolov3

Method	mAP	FPS
Faster R-CNN (VGG16)	73.2	7
YOLO (VGG16)	66.4	21
YOLOv3-608	57.9	51
SSD-512	**76.8**	**22**

Table 4.3 Results on Pascal VOC2007.

is currently the most appropriate candidate for the application of real-time traffic objects detection in vehicles.

SqueezeDet: SqueezeDet is fully-convolutional network inspired by Yolo. SqueezeDet first takes an image as input and extract a low-resolution, high dimensional feature map from the image. Then, the feature map is fed into the *ConvDet* layer to compute bounding boxes centered around $W \times H$ uniformly distributed spatial grids, where W and H are number of grid centers along horizontal and vertical axes ($W = 76,\ H = 22$). *ConvDet* is in fact a convolutional layer that maps the extracted features to $K\ \times (4\ +\ 1+\ C)$ parameters at each grid cell, similar to Yolo here K is number of anchors at the center of a grid cell, 4 scalars describing anchor box $(\hat{x}_i, \hat{y}_j, \hat{w}_k, \hat{h}_k)$ and $C+1$ outputs for each anchor encode the confidence score for prediction and conditional class probabilities for C classes. For each anchor ($i \in [1,W],\ j \in [1,H],\ k \in [1,K]$) its relative coordinate is computed as follow:

$$\begin{aligned}\delta x_{ijk}^{G} &= (x^G - \hat{x}_i)/\hat{w}_k,\\ \delta y_{ijk}^{G} &= (y^G - \hat{y}_j)/\hat{h}_k,\\ \delta w_{ijk}^{G} &= log(w^G/\hat{w}_k),\\ \delta h_{ijk}^{G} &= log(h^G/\hat{h}_k).\end{aligned} \tag{4.13}$$

where (x^G, y^G, w^G, h^G) are coordinates of ground truth bounding box. Responsible anchor boxes are the one with highest IoU. SqueezeDet considers the detection as a regression problem, like Yolo. Therefore it puts three components together to form its loss function comprised of detection, localization and classification error as formalized in the equation 4.14.

$$\begin{aligned}
&\frac{\lambda_{bbox}}{N_{obj}}\sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{K} I_{ijk}[(\delta x_{ijk}-\delta x_{ijk}^{G})^2+(\delta y_{ijk}-\delta y_{ijk}^{G})^2 \\
&+(\delta w_{ijk}-\delta w_{ijk}^{G})^2+(\delta h_{ijk}-\delta h_{ijk}^{G})^2] \\
&+\sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{K}\frac{\lambda_{conf}^{+}}{N_{obj}} I_{ijk}(\gamma_{ijk}-\gamma_{ijk}^{G})^2+\frac{\lambda_{conf}^{-}}{WHK-N_{obj}}\bar{I}_{ijk}\gamma_{ijk}^{2} \\
&+\frac{1}{N_{obj}}\sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{K}\sum_{c=1}^{C} I_{ijk} l_c^G log(p_c).
\end{aligned} \tag{4.14}$$

I_{ijk} is 1 if the k-th anchor at position (i, j) is associated with a ground truth, otherwise 0. $[\lambda_{bbox},\ \lambda_{conf}^{+},\ \lambda_{conf}^{-}]$ are coefficients that attach weights to localization and detection error respectively. These hyper-parameters are selected empirically and set initially as follow: $\lambda_{bbox} = 5$, $\lambda_{conf}^{+} = 75$, $\lambda_{conf}^{-} = 100$. The second part in equation 4.14 represents the confidence score regression, its input γ_{ijk} is normalized by *sigmoid* to squash the output of ConvDet between 0 and 1. And finally the last part of the multi-task loss function forms the classification error as cross-entropy loss. $l_c^G \in \{0, 1\}$ is the ground truth label and p_c is normalized by *softmax* and is taken as the probability.

SqueezeDet attempts to reduce the parameter size while preserving accuracy with help of an alternative module called *Fire Module* which is combination of 3 convolutional layers. The input of the fire module (squeeze layer) is a 1×1 convolutional layer that compresses an input tensor with large channel size to one with the same batch and spatial dimension, but smaller channel size. The output of the fire module is a mixture of 1×1 and 3×3 convolution filters that takes the compressed tensor as input, retrieve the rich features and output an activation tensor with large channel size. SqueezeDet has two versions, one light version with 4.72MB model size and the heavier variant SqueezeDet+ with 19MB model size which higher accuracy. SqueezeDet+ is built from 2 input and output convolutional layers, plus 3 pooling layer (second, fifth and eighth layers) and 10 fire layers. The input size of SqueezeDet+ is 375×1242 that will be mapped into $22 \times 76 \times 9 \times (c + 4 + 1)$, where 9 is the number of anchors per grid cell and c number of classes.

the performance of both squeezeDet and squeezeDet+ are compared in the table 4.4. The results are obtained from the original paper published [43].

SqueezeDet was chosen as the main subject for all experiments in this thesis, the reason is the promising performance and the potential for improvement of it in comparison to almost all other single-stage detectors, except Yolov3, a worthy competitor that is a great step ahead due to its multi-scale structure.

Method	Car	Cyclist	Pedestrian	mAP	speed (FPS)
SqueezeDet	73.9	72.1	65.8	76.7	**57.2**
SqueezeDet+	**78.9**	**78.1**	**68.5**	**80.4**	32.1

Table 4.4 mAP and inference speed of both versions of squeezeDet on KITTI dataset using NVIDIA TITAN X GPU.

4.2.2 CNN on Fisheye Images

Without exception all CNN-based detectors are trained with usual pinhole camera images clear of any optical aberration, which resemble the primary form and appearance of objects in real world. Distorted objects in fisheye images appear to them more as an odd object and detecting them gets more challenging. Not so many attempts are made to specialize the existent CNNs for detection of deformed images produced and affected by fisheye lens. Deng et al. [3] is a CNN-based detector for semantic segmentation on urban traffic scenes using fisheye camera. Deng et al. [3] designed a type of data augmentation specially for fisheye images to improve the net's generalization performance, called *Zoom Augmentation*. This network also takes advantage of **Overlapping Pyramid Pooling** (OPP) technique proposed by He et al. [11] to explore local, global and pyramid local region context information.

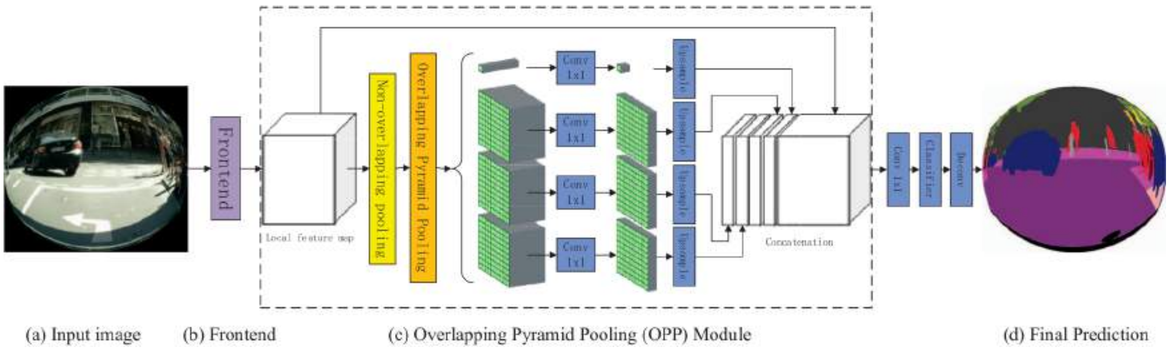


Fig. 4.15 The framework of OPP-net. source: [3]

In Figure 4.15 the front-end (b) generates local feature map from augmented input image (a), and then the overlapping pooling pyramid module (c) gathers local, global and pyramid region context information. The non-overlapping pooling in the OPP module (c) is used to down-sample the feature map. Global and pyramid region feature maps are generated by the overlapping pyramid pooling. The concatenated feature map is reduced by a 1x1 convolutional layer with channel dimension 512 and then classified by a 3x3 convolutional layer. Finally, final prediction (d) is obtained by a convolutional layer. As stated in the paper of Deng et al. [3], in order to model fisheye effect on input images a common equidistance model is constructed as follow:

$$r = f\ tan\ \Theta \tag{4.15}$$

$$r = f\ \Theta \tag{4.16}$$

where Θ is the angle between the principal axis and the incoming ray, r is the distance between the image point and the principal point, and f is the focal length. The mapping from the fisheye image point $P_f = (x_f, y_f)$ to conventional image point $P_c = (x_c, y_c)$ is described by:

$$r_c = f\ tan(r_f\ /\ f), \tag{4.17}$$

where $r_c = sqrt((x_c - u_{cx})^2 + (y_c - u_{cy})^2)$ denotes the distance between the image point P_c and the principal point $U_c = (c_{cx} - u_{cy})$ in the conventional image, and $r_f = sqrt((x_f - u_{fx})^2 + (y_f - u_{fy})^2)$ correspondingly denotes the distance between the image point P_f and the principal point $U_f = (u_{fx}, u_{fy})$ in the fisheye image. The focal length f determines the strength of the distortion map. Each image and its corresponding annotation are then transformed using the resulted mapping function to generate the *fisheye image dataset*. The type of interpolation used for image and annotation mapping is different, respectively bilinear interpolation and nearest-neighbor interpolation.The training set is augmented by employing different scales of focal length, causing various distortion strength as illustrated in figure 4.16. The OPP-net trained with augmented dataset achieves the highest mIoU on generated fisheye image test dataset (Result shown in table 4.5) and is able to capture details in the edge of the images. As the result shows the augmentation enables the network to recognize distorted objects, since it extracts and learns the features of deformed objects, with other words the correlation between transformed pixels, during training phase.

A similar approach is used in this thesis to empower CNN-based detector to handle object detection on fisheye images than semantic segmentation. When it comes to distorting

Method	Car	Rider	Truck	Road	Person
OPP-net	85.8	34.2	40.2	96.5	63.3
OPP-net+AUG	**86.7**	**39.4**	**48.6**	**96.7**	**65.7**

Table 4.5 result on the validation set of the generated fisheye image dataset for semantic segmentation. source: [3]

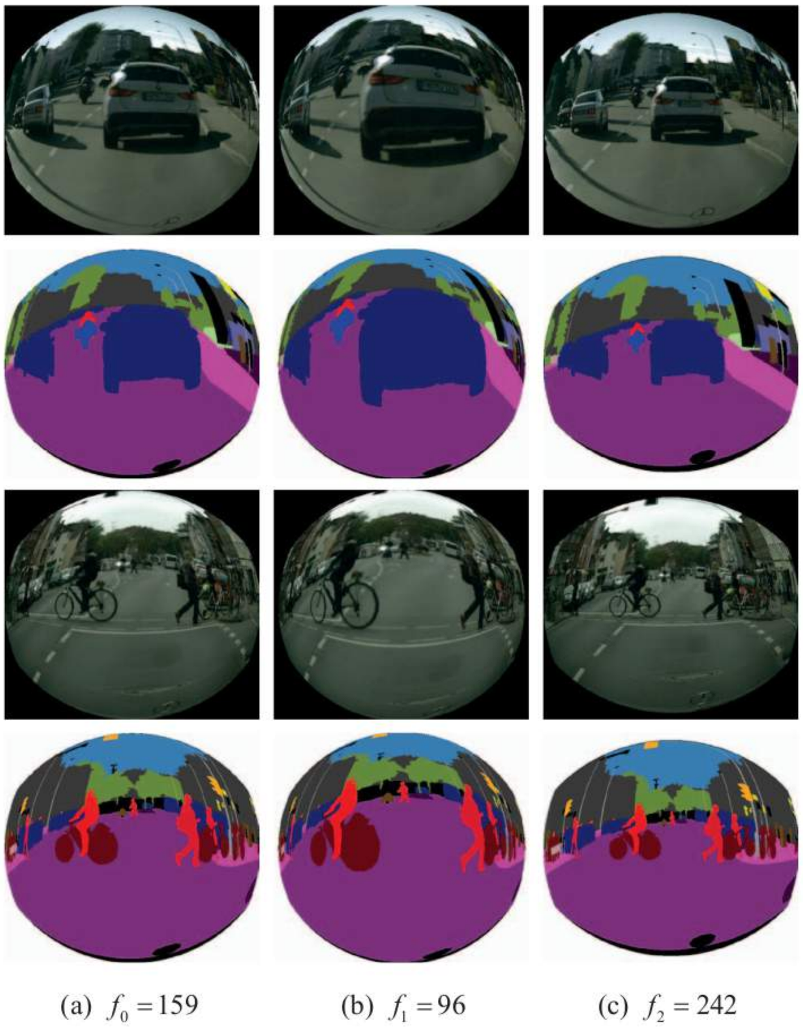


Fig. 4.16 Smaller focal length (b) introduces stronger distortions and bigger focal length (c) introduces weaker distortions. source: [3]

an object within a bounding box, correcting and transforming the bounding box causes instability due to the dislocation of corner coordinates of bounding boxes. We particularize this issue in the optimization chapter.

Chapter 5

Optimization

5.1 Overview

In this chapter we elaborate all approaches taken in order to optimize our chosen detector, SqueezeDetPlus [43] to accomplish better performance over fisheye images. The precise estimation of the improvements that have been achieved will be presented in chapter 7.

5.2 Fisheye Effect

As already discussed in chapter 3, the lack of dataset containing fisheye images leads to the problem that no CNN-based detector is well equipped to cope with highly distorted objects captured by fisheye cameras, and we also argued why undistorting fisheye images for detection is an unfavorable way to tackle this problem. Hence we create synthetical fisheye datasets by transforming the existent datasets images into distorted ones. We use the fisheye camera model described in chapter 3 to construct barrel distortion filter. This fisheye filter is then applied to input images within the pre-processing steps. We take the example image 3.4 to catch a glimpse of how the distortion works. As it's also apparent in our example image 3.4, barrel distortion not only scales down/up an object but also rotates it depending on its location. The only way to fit a rotated object into a non-rotatable quadrilateral with four right angles (rectangle) is to enlarge the rectangle to have more area diagonally, logically bigger area means more pixels. As a consequence unrelated pixels (highlighted in yellow in figure 5.1) step inside the frame unwillingly. This issue is amplified specially when the bounding box drawn by a labeler (in most cases manually by human) is not precise and already encloses background pixels (highlighted in red on right image in figure 5.1), which should be in fact excluded. This is absolutely beyond debate, that slight amount of error

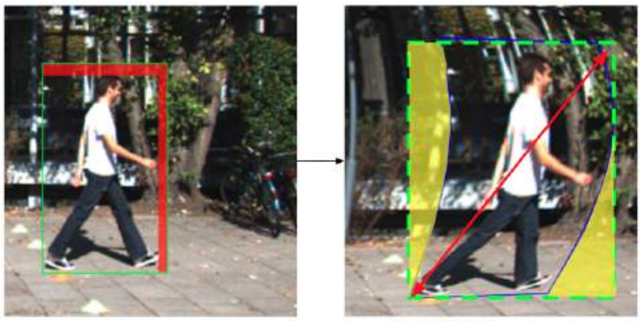


Fig. 5.1 Fitting the object within the box after transformation.

during labeling is out of control, but occurrence frequency of such inaccurate bounding box in KITTI or BDD is considerably so high, as if one would observe only 10 random images of either KITTI or BDD could spot this easily. In our case this minimal error rate elevates with distortion and we need to diminish this somehow.

Adjusting the existing bounding boxes would be very expensive and time consuming, therefore we should preferably regenerate the entire annotation automatically. For this we made use of **MASK-RCNN** to generate instance masks for KITTI and BDD, then bounding boxes were determined on the basis of instance masks. On the one hand this helped us to have much fitter bounding boxes, as the example image in figure 5.2 proves, on the other hand MASK-RCNN detected more objects than all labeled ones in the original dataset. The more objects appear in one image, the more network has to learn. It should be also said, that the detection error of MASK-RCNN is minimal and can be disregarded.

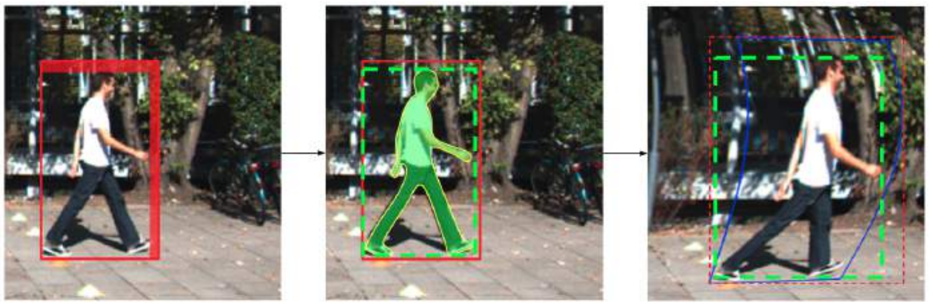


Fig. 5.2 Using instance mask to determine new bounding box coordinates. The bounding boxes in green are extracted from instance mask, and the red one are the original bounding boxes. As it can be observed, the green dashed box in the right picture leaves no gap between its boundaries and the actual object, while the red dashed box is a bit larger and has a sort of padding.

Once the instance masks are generated they will be stored, during training the bounding box's coordinates in distorted image are calculated using the inverse of the map yielded from the fisheye camera model (more details in chapter 6).

5.3 Datasets

SqueezeDetPlus is originally trained with KITTI, although KITTI is widely used in CV community but it has weaknesses, which hinder us from accepting it as the very best dataset for our application.

- Low ground truth density per image: KITTI has totally 40.534 labeled objects from 8 classes in 7.481 images, giving in average $5,41$ objects per image, which is in comparison to BDD with $12,31$ objects per image near 50% less. This means that KITTI images with 375×1224 resolution feed the network more with background informations than object-related informations. Besides there exist images containing unlabeled objects frequently enough, that lowers extra the ground truth density per image of KITTI. Beyond that, unlabeled objects detected will be considered as false negative during training, this could utterly cause confusion for the network.
- Inaccurate bounding boxes: bounding boxes representing objects are not tight-fitting defined (figure 5.1 as example). As a consequence network's predicted height and width will not be precise. This leads to higher localization error specially in situations where several objects fall very close to each other.
- Low environmental variation: scenes of various weather conditions (rainy, snowy, etc.) and different daylights (day and night) are barely found in KITTI. As a result the network could easily fail to detect vehicles covered by snow or appeared in night with completely different reflections for instance. This necessarily needed variance is more minded in BDD.

For the reasons indicated above we merged KITTI and BDD, the resultant dataset (BDD-KITTI) is significantly larger 87481 images (1.026.011 instances) presenting much wider variance. And as already mentioned to enhance the accuracy of ground truths in both KITTI and BDD, annotations are regenerated over again using instance mask. Thereout another variant is created, we call this mixture dataset MASK-BDD-KITTI, which contains even more instances than BDD-KITTI.

5.4 Pre-processing

The author of SqueezeDet used mean over BGR channels of VGG16 to perform mean subtraction on input images. On the contrary we discarded the mean subtraction, since subtracting the mean value extracted from a different dataset on other images would not correctly centralize the pixel values. Instead we performed the normalization through batch normalization during training, and only for inference we divide RGB value of input images by 255 to perform range adjustment (between 0 and 1).

5.4.1 Random Translation

Random translation is a popular augmentation technique, which is also used in original squeezeDetPlus. But in our case fisheye effect distorts pixels differently depending on their locations, with other words objects at the center of an image are handled differently than the ones at edges. Hence we dropped out the translation to prevent any interference in barrel distortion.

5.4.2 Small Objects Exclusion

Datasets contain usually small objects. Such small objects are barely representative and have low resolution due to their size, consequently they send indistinct informations into network, which are staggeringly hard for neurons to decode. Small objects could have high-impact on network performance and increase the recognition error rate if they occur frequently, which is also the case in KITTI and BDD. Normally small objects are located in the distance,this make them dispensable during driving in real world, thus we could exclude them from training. To do so, we set a threshold to filter out objects with area smaller than a specific value (usually 0.75 of the smallest anchor box), this is in average 0.41 of all objects per batch, which is enormously high. The **exclusion of small boxes** not only decreased the amount of false negatives but also accelerated the convergence, since the more we had small objects in a batch the higher was the error (more details in chapter 7).

5.4.3 Multi-scale Input

Ideally an object detector should be able to deal with input images with different sizes, in fact objects that appear in various scales. Usually the variation of object scales in a dataset with fix image size is low, this could be compensated simply by resizing them. Therefore we equipped our input pipeline with **Multi-scale Input** feature. During training instead of

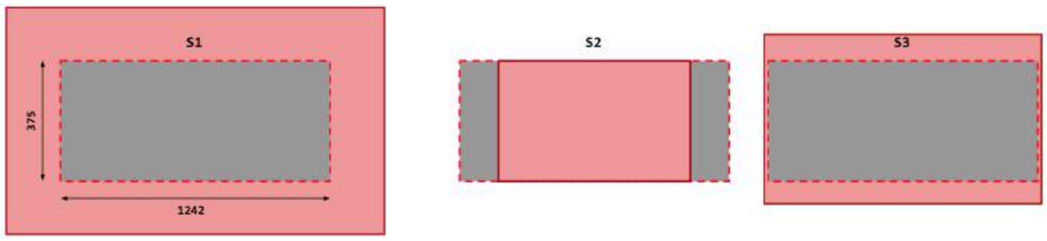


Fig. 5.3 Multi-scale input. The gray rectangle with red dashed border is the input windows of SueezeDetPlus, the pink rectangle is the scaled input image. S_1 scales up and center-crops the input image, S_2 scales down and pads the input image and S_3 just center-crops without scaling.

fixing the input image size every few iterations the input image is resized by three different scaling factors (S_1, S_2, S_3) keeping the aspect ratio solid, S_1 up-samples the input image, S_2 down-samples and pads the image with specific value (by default colored gray $(15,15,15)$) and S_3 is neutral and only crops the image with original size. Up-sampling the input images is beneficial, since on the one hand small ground truths also contribute to the network's learning and on the other hand in datasets like KITTI and BDD where the objects are mostly concentrated at the center of the vertical axis, the useless informations in the top and bottom regions of image representing typically sky and road are cut away. Down-sampling with padding squeezes pixels together at the center of input layer, this has a favorable effect that the informations passed to neurons near to edges will not be vanished away, instead they stay inside the effective receptive field of the network. Note, that resizing the inputs in short intervals destabilizes the convergence flow, to avoid this inputs should be resized interchangeably in appropriate distances, every 10 or 20 batches in our case. This regime forces the network to learn to predict well across a variety of input dimensions.

5.5 Hyper-parameters

The objective of SqueezeDet to minimize is multi-task loss function (Equation 5.1), comprised of localization error, detection error and classification error. Each part of the loss function is weighted with a coefficient normalized by the number of objects in a batch, λ_{bbox}, λ_{conf}^{+} and λ_{conf}^{-} respectively for bounding box and confidence regression, as written in equation 5.1. These Hyper-parameters are selected empirically. To put more stress on localization error and the negative detections we scaled up these penalization factors.

$$
\begin{aligned}
&\frac{\lambda_{bbox}}{N_{obj}} \sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{K} I_{ijk}[(\delta x_{ijk} - \delta x_{ijk}^{G})^2 + (\delta y_{ijk} - \delta y_{ijk}^{G})^2 \\
&+(\delta w_{ijk} - \delta w_{ijk}^{G})^2 + (\delta h_{ijk} - \delta h_{ijk}^{G})^2] \\
&+\sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{K} \frac{\lambda_{conf}^{+}}{N_{obj}} I_{ijk}(\gamma_{ijk} - \gamma_{ijk}^{G})^2 + \frac{\lambda_{conf}^{-}}{WHK - N_{obj}} \bar{I}_{ijk}\gamma_{ijk}^{2} \\
&+\frac{1}{N_{obj}} \sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{K}\sum_{c=1}^{C} I_{ijk} l_{c}^{G} log(p_c).
\end{aligned}
\tag{5.1}
$$

We've tested different values for λ_{bbox} and λ_{conf}^{-} and it ended up with the following values: $\lambda_{bbox} = \mathbf{15}$ and $\lambda_{conf}^{-} = \mathbf{120}$. Since the localization and confidence error are forced up, the network has to search deeper for a suitable combination of weights to minimize them.

Chapter 6

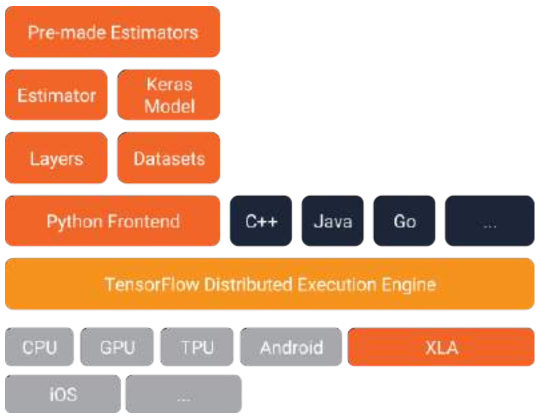
Implementation

There is a quote from Leonardo Da Vinci saying "*simplicity is the ultimate sophistication.*" This quote plays on the idea that being simple is not banal, it's elegant and harmonious. In this thesis we adhere to this idea and put simplicity above all else. Therefore we use a high-level Tensorflow API called **Estimators** as infrastructure for the implementation.

6.1 Estimators

Simplicity and flexibility are prime concern of each machine learning developer. Estimator framework preserves flexibility while it simplifies the most common usage patterns. Estimators include both pre-made models for common machine learning tasks and customize model. Estimator Framework takes the heat off machine learning programmers, as it encapsulates the following actions: Training, Evaluation, Prediction and Export for serving. Estimator Framework are built on *Keras Layers*, which simplifies customization. The most advantageous feature of Estimator Framework is that it provides a safe distributed training loop that controls and coordinates how and when to:

Fig. 6.1 TensorFlow architecture. source: [38]



- build the graph
- initialize variables
- load data

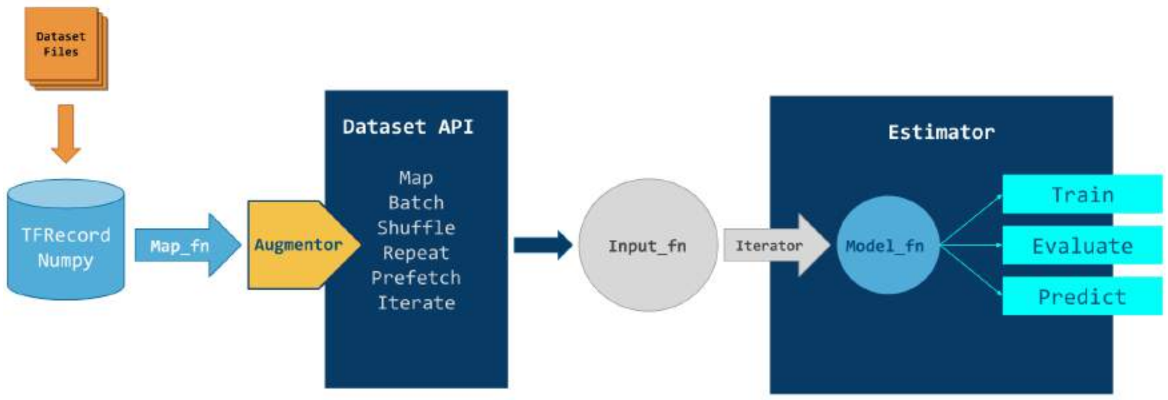


Fig. 6.2 Overall workflow of the estimator framework.

- handle exceptions
- create checkpoint files and recover from failures
- save summaries for TensorBoard

In addition to these the input pipeline is separated from the model. This enables us to easily experiment with different datasets. The heart of a estimator is the **Model Function** (model_fn) that defines the input layer, intermediate layers (Hidden layers) and the output layer. After defining the model next step will be the **Input Function** (input_fn) responsible for supplying the model with data. Model_fn has three different modes: Train, Evaluate and Predict. Depending on the mode model_fn threat the input data accordingly. In a nutshell this is generally the base workflow of Estimators, as it also depicted in the figure 6.2.

6.2 Object Detection Framework

We utilize Estimators API and bundle it with **Datasets API** together, to construct our Object Detection Framework. Hereby our intention is to build up a universal tool specialized for object detection that provides pre-defined functions and components, which are commonly used, and also supplies the infrastructure needed to create a detector from a to z. Therefore during implementation it'd been attempted to unify functions as much as possible. The diagram in figure 6.2 gives us an overview of all components involving in the object detection framework. First of all it should be noted, that all hyper-parameters needed to build a model and run the framework are stored in a dictionary (*config_dict*), that will be handed around for subsequent settings. We start with the input pipeline and proceed to the endpoint of the framework.

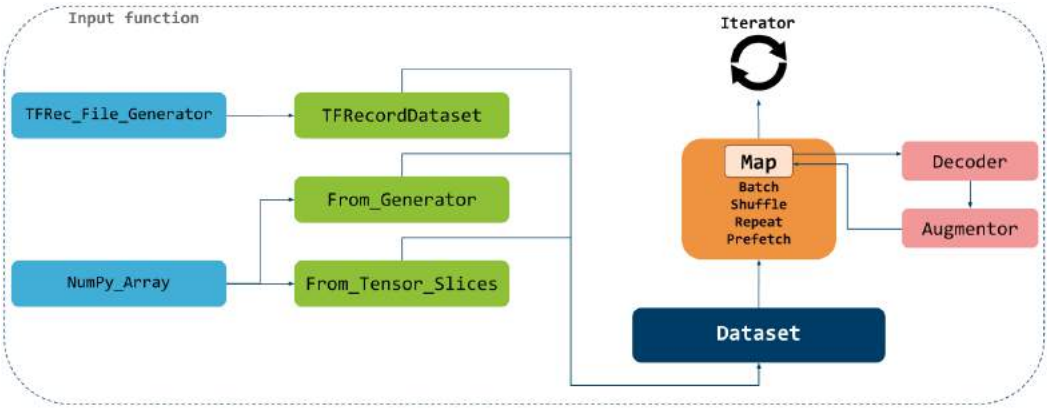


Fig. 6.3 Datasets API at a high level.

6.2.1 Input Function

Since release Tensorflow 1.4, Datasets is a new way to create input pipelines to TensorFlow models. This API is much more performant than using *feed_dict* or the *queue-based* pipelines, and it's cleaner and easier to use. To generate a Dataset at first raw data containing images and labels will be converted to either **TFRecord** files or NumPy Arrays. TFRecord file is a binary storage format created and optimized for Tensorflow applications, it makes it easy to combine multiple datasets and integrates seamlessly with the data import and pre-processing functionality provided by the library. The conversion from raw data to TFRecord files is performed by the generator (*TFRec_File_Generator* shown in figure 6.3). The generator reads both the annotations from text files and images simultaneously, shuffles them to avoid any sequential pattern in data and writes them into TFRecord files, at this point the TFRecord files are divided into Training and Validation sets, the portion of each can be sized by passing a parameter to the generator function. The training and validation sets can be stored either each packed as a single file or fragmented into smaller subsets. The generator accepts only annotations written in accord with KITTI labeling format, due to the lack of a unified annotation format among the published datasets, thus other annotations have to be first transformed into KITTI format. Dataset can be also created from slices of tensors (*From_Tensor_Slices*) containing NumPy arrays, but if the dataset is greater than 2 GB it can run into the limit for the graph protocol buffer. Best practice is to use generator (*From_Generator*) to read data into memory iteratively once it's called.

In the next step TFRecord files or NumPy arrays are respectively passed to *TFRecord-Dataset* or *From_Generator* method of Datasets class. Now dataset is prepared to be fetched for further processes, at this stage dataset class is called into action with a handful of useful tools to wrap up the input pipeline:

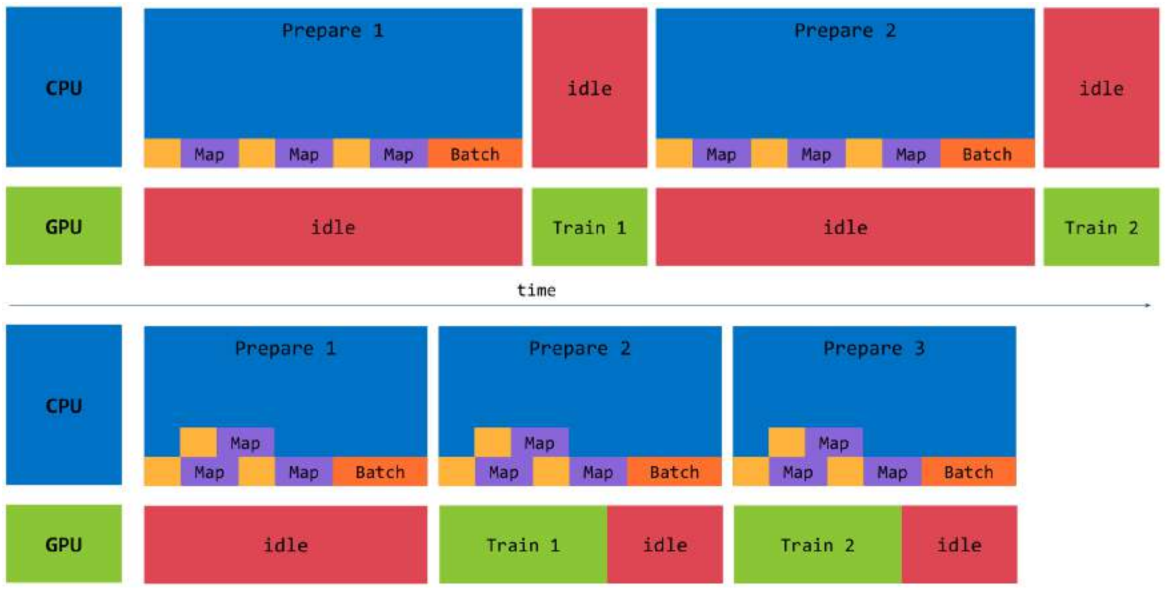


Fig. 6.4 Reduction of idle time by pre-fetching and parallelizing data transformation. source: [38]

- Shuffle: It's necessary to shuffle the elements of dataset before feeding them into the network to have a smooth distribution and various order in input data to prevent overfitting. This is done easily with this method.
- Map: This method applies the given function across the elements of the dataset and outputs the transformed elements in the same order as they appeared in the input. Hereby all the pre-processing modifications including augmentations take place, which are defined in the *map_func*. Map method is able to perform the transformation in parallel, this speeds up significantly the flow of the input pipeline. The pre-processing is parallelized across multiple CPU cores. *num_parallel_calls* argument specifies the level of parallelism. Choosing the best value for the num_parallel_calls argument depends on your hardware, characteristics of your training data (such as its size and shape), the cost of your map function, and what other processing is happening on the CPU at the same time; a simple heuristic is to use the number of available CPU cores.
- Batch: this method combines consecutive transformed elements of the dataset into batches. The size of the batch is set in the configuration dictionary.
- Pre-fetch: To perform a training step, we must first extract and transform the training data and then feed it to a model running on an accelerator. However, in a naive synchronous implementation, while the CPU is preparing the data, the accelerator is sitting idle. Conversely, while the accelerator is training the model, the CPU is sitting

idle. The training step time is thus the sum of both CPU pre-processing time and the accelerator training time. Pipelining overlaps the pre-processing and model execution of a training step. While the accelerator is performing training step N, the CPU is preparing the data for step $N+1$. Doing so reduces the step time to the maximum (as opposed to the sum) of the training and the time it takes to extract and transform the data. Pre-fetch method provides a mechanism to decouple the time data is produced from the time it is consumed. It uses an internal buffer to Pre-fetch elements from the input dataset ahead of the time they are requested. The buffer size should be adjusted with respect to the available RAM. Pre-fetch buffer size is also set as parameter in the configuration dictionary.

- Repeat: performs repetition, when the entire dataset is once pumped into the model.

Overlapping the work of producer and consumer decreases the training time and makes efficient occupation of GPU by utilizing it constantly. Figure 6.4 illustrate the combination of parallel mapping and pre-fetching. After all the flow of the pipeline depends on its delicate configuration (Pre-fetch_buffer_size and num_parallel_calls) and naturally the cost of the map_func. Back to figure 6.3, on the last ring of this chain we instantiate an iterator which enumerates the elements of the dataset. All components mentioned so far are encapsulated in one function as *input_fn*. The input_fn will be passed on to model_fn as a generator and it feeds the model as needed. Input_fn grabs either validation or training dataset (TFRecord files) depending on the current mode of the model_fn. Next we delve into the augmentation part performed in mapping step, where in our implementation most importantly the fisheye effect is simulated.

6.2.1.1 Augmentations

As depicted in figure 6.4 the map_fn consists of *Decoder* and *Augmenter*. Decoder parses single binary example from TFRecord files and decodes them in primary formats before encoding, then it forwards the parsed/decoded image and annotations to the augmenter. The augmenter class provides diverse types of image augmentation methods as follow:

- Random Translation: given a vector v it translates the input image along v, otherwise it chooses v randomly from a uniform distribution in the range $[0, i)$ where i is an integer between 0 and 10 representing the intensity of the translation. The magnitude of v stems from the division between input image size and i.

- Brightness: adjusts the brightness of the input image by adding d to image tensor. d is a float drawn randomly from the interval $[-delta, delta)$ where $delta$ is given as parameter to the method.
- Flip: simply mirrors the input image.
- Fisheye Effect: simulates barrel distortion using OpenCV. More details below.

To instantiate the augmenter an integer f_u should be given, which sets the upper-bound of the interval $i = [0, f_u)$. The *frequency* of applying all methods listed above on input image tensors is controlled by i, whilst an integer t is randomly drawn from a discrete uniform distribution in i. If t is greater than $f - f_u$, the frequency parameter passed individually to each method, the augmentation method will be applied, otherwise it'll pass on the input image untouched. Logically the greater f is, the less likely is that the condition $t > f - f_u$ is satisfied, therefore if we would like to apply a method more frequent we need to select f smaller.

6.2.1.2 Fisheye Effect

We use the same pinhole camera model 3.3 mentioned in the chapter 3 to emulate the fisheye effect, we just need to pick the distortion coefficients $(k_1, k_2, k_3, k_4, k_5, k_6)$ correctly to get the barrel distortion out of the camera model. If fisheye mode in *config_dict* is set to $True$, the augmenter fetches the camera matrix and distortion coefficients given in config_dict plus one sample image from the dataset, with which the network will be trained. This sample image is necessary to retrieve the height and width of the new camera model, besides (c_x, c_y) the principal point of the camera is half of the height and width. OpenCV provides $initUndistortRectifyMap$ function to computes the undistortion and rectification transformation, but in our case since the original images are undistorted and we set k_1 to be negative, the function generates distortion (Barrel distortion in figure 3.2) than performing undistortion. Since the data-sheet and the model of cameras used for capturing dataset images are not available, we find the best suitable focal length (f_x, f_y) and distortion coefficients k_n experimentally. $initUndistortRectifyMap$ actually builds the maps (m_1, m_2) for the inverse mapping algorithm that is used by *remap* function. That is, for each pixel (u, v) in the destination (distorted) image, the function computes the corresponding coordinates in the source image (original image). These maps are calculated initially only once, after that they are dumped as an array into a pickle file for further usages. To prevent redundant reading of the pickle file over and over, the maps are also appended to *config_dict*, thus they can be accessed each time augmenter is called. Once the maps are generated we use the

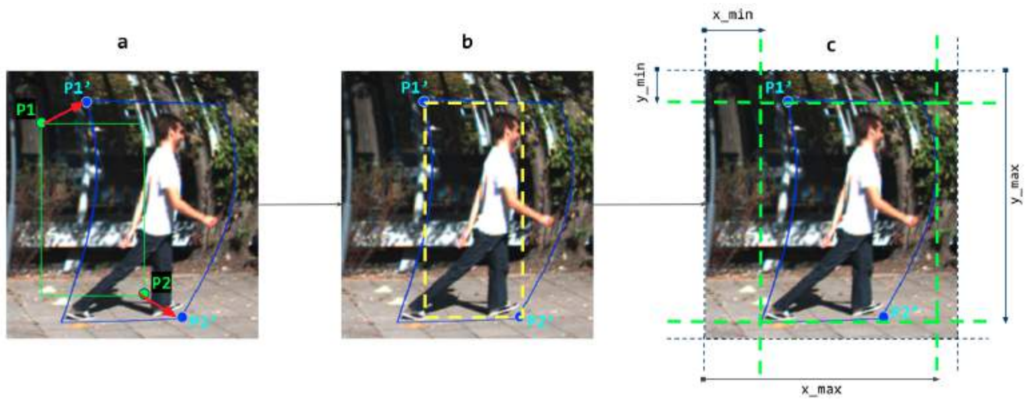


Fig. 6.5 Fitting bounding box to distorted object by looking up new coordinate in m'.

OpenCV function *remap* to apply a generic geometrical transformation using (m_1, m_2). The barrel distortion as explained in chapter 3 magnifies the pixels at the center and shifts the pixels towards edges, consequently pixels get merged and leave gaps. This means that in the destination image there are locations, for which no pixel value are on hand. We need to fill these gaps using interpolation methods. *remap* also deals with interpolations, pre-defined in OpenCV. Our augmenter takes *Cubic Interpolation* for the rectification.

Next, each bounding box has to be adjusted appropriately in a way that it encloses precisely the entire distorted object. Bounding boxes are represented by the top left $p_1 = (x_{min}, y_{min})$ and bottom right $p_2 = (x_{max}, y_{max})$ coordinate (diagonal format). The easiest approach is to look up in the maps (m_1, m_2) to find the new coordinates of p_1 and p_2. As already mentioned, (m_1, m_2) give the coordinates in the source image from where pixels in the destination image come. We need exactly the reverse of (m_1, m_2), a mapping table m' which tells us where in the destination image pixels of the source image land. To generate m' we iterate over m_1 and use its value at each location $(row, column)$ as index to store $(row, column)$ in m'. The yielded matrix is then the reverse map of m_1, obviously m' is still incomplete, since (m_1, m_2) where gappy. To overcome this we use again an interpolation technique. We apply *2D k-nearest-neighbor* on m', this fills a gap (recognizable as *None*), as it takes the mean over its k neighbors in all four directions (left, right, top, bottom). If all k neighbors are also *None*, k will be incremented till it finds at least one neighbor with value other than *None*. After m' is rectified, it can be used as lookup table to retrieve new coordinate in distorted image for any pixel from original image.

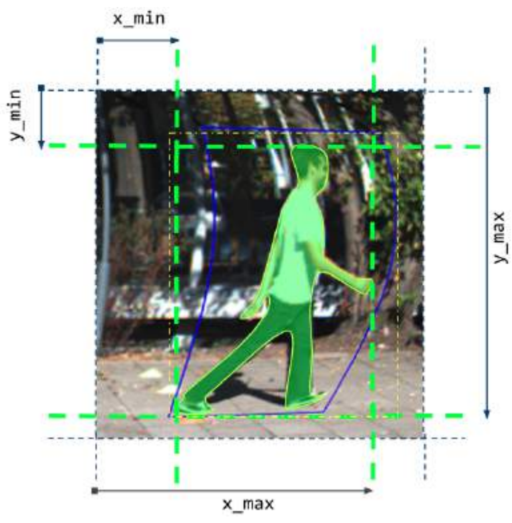
Figure 6.5 demonstrates how we transform a bounding box. The green box in 6.5a is the ground truth (gt) presenting the pedestrian at first place, after distortion gt is deformed, the box colored in blue, and (p_1, p_2) are moved to (p'_1, p'_2). As we can see in 6.5b, if we would simply take the (p'_1, p'_2) as new coordinates of the transformed box, the resultant bounding box (yellow dashed box) would partly cut off the object within, this is clearly not desirable, since we don't want to drop any information from the primal object. To avoid this, we need to take more pixels into account than just (p_1, p_2). In fact we should track how the pixels lying at the four sides of the box (left, right, top, bottom) are moved, to set down new boundaries. Hence we lookup at m' for all pixels on the boundaries, then new coordinates $[x'_{min}, y'_{min}, x'_{max}, y'_{max}]$ are extracted as follow:

- x'_{min}: minimum x on left side
- y'_{min}: minimum y on top side
- x'_{max}: maximum x on right side
- y'_{max}: maximum y on bottom side

thereby the transformed bounding box (green dashed box in 6.5c) encloses the entire object.

The downside of this approach is, if only one background pixel exists that due to barrel distortion falls further far from other pixels at the boundary, the bounding box will be enlarged and stretched out unnecessarily. That is also the case in our example image, as the boundary of the box at the right side is pushed further and consequently the transformed bounding box includes irrelevant pixels. To tackle this issue we should discard background pixels and determine box boundaries based on object-related pixels, which are given as *instance mask*. To be noted, that instance masks, generated by [10], are available only for KITTI and BDD. If instance masks are given, instance boundary pixels are then taken to lookup at m'. In this case new coordinates are extracted as follow:

Fig. 6.6 Fix bounding box using instance mask generated by Mask-RCNN.



- x'_{min}: minimum x of all boundary pixels
- y'_{min}: minimum y of all boundary pixels
- x'_{max}: maximum x of all boundary pixels

- y'_{max}: maximum y of all boundary pixels

as it can be seen in figure 6.6 using instance boundary pixels enables us to narrow the bounding box down to the object boundaries itself. In this way bounding box is focused on the object within, although it is inevitable to not having extra background pixels included, due to the rotation caused by barrel distortion. After all this approach minimizes the amount of unwanted informations after transformation.

6.2.2 Model Function

After performing all pre-processing tasks, the iterator wraps up the *input_fn*. With this we come to the core of the estimator, the model function (*model_fn*) where the beating heart of estimator lies. model_fn contains the actual structure of the network, all layers and components from input layer down to the last detection layer. Once the **Estimator** is instantiated, model_fn builds the graph. Depending on the value of *mode* (Train, Eval, Predict) the corresponding *EstimatorSpec* instance will be created. As depicted in figure 6.7, each mode requires different arguments and operators.

- *Train_EstimatorSpec*: takes ***Loss_fn*** and an optimizer to minimizes the loss_fn with. Optimizer initially creates *global_step* variable and increases it to keep track of training process (number of batches passed to the model). Output of the loss_fn is also regularly stored with example visualized detections in summary for tensorboard.

- *Evaluate_EstimatorSpec*: uses *Loss_fn* to calculated the evaluation metrics given as dictionary (*eval_metrics*). The result is also posted in tensorboard.

- *Predict_EstimatorSpec*: passes the output of the last layer to a parser for further post-processing steps (NMS, etc.), which are all wrapped in another function *detector*

Estimator creates the *tf.Session* based on the configuration parameters (*RunConfig*) given in config_dict. RunConfig includes the options to determine the fraction of the overall amount of memory that each visible GPU should be allocated (*per_process_gpu_memory_fraction*), as well as whether a process is allowed to grow in memory usage (*allow_growth*). Besides the frequency of saving checkpoints and the number of checkpoints to keep are among the parameters of RunConfig as well. Noteworthy, that this feature of Estimator API again takes a load off and simplifies the control over saving and restoring checkpoint. If user starts a training by pointing at an already existing model checkpoints, Estimator will automatically restore previous checkpoint and proceed the training from the point it stops, under assumption that all model structure related configurations are unaltered.

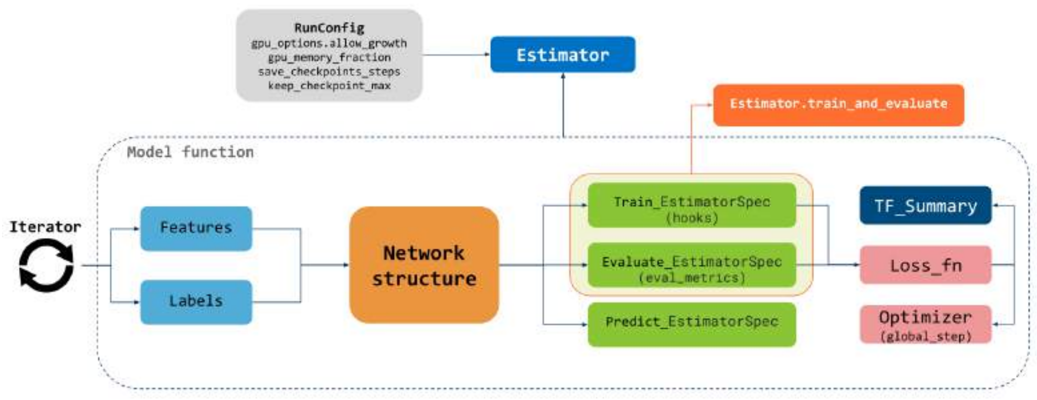


Fig. 6.7 Model function is the main part of the estimator that builds the graph for training, evaluation or prediction.

Both training and evaluation process are packed up in another function (***trainer***), where a handy function of the estimator API is integrated. **estimator.train_and_evaluate** is a utility function that coordinates the both training and evaluation process. It starts the training and simultaneously runs the evaluation in between at regular intervals determined by *steps* and *throttle_secs* parameters passed to evaluate_EstimatorSpec. estimator.train_and_evaluate provides convenient options to creates early-stopping hook. We take advantage of two stop conditions available in the estimator API:

- **stop_if_no_decrease_hook**: stops training if the given metric (in our implementation, total_loss) does not decrease within maximum allowed training steps. This could be also used to monitor the exponential decay of the learning rate l.
- **stop_if_lower_hook**: stops training if the given metric is lower than the threshold. Finding the threshold to stop training at the best stage is not straight forward and should be found on trial, but overfitting is avoidable by establishing this stop criteria easily.

For inference one just need to call ***detector*** function while a checkpoint file is passed, if not the detector initializes the graph randomly. The output is then sliced into [*predicted_bboxes, predicted_classes, predicted_scores*] and forwarded to NMS to filter out best n predictions. n determines the number of predictions with highest score that NMS will observe. NMS is implemented in both Tensorflow and pure python, due to slow pace of tensorflow version we use the python version preferably. At the end the selected detections are visualized on the input images and stored also in a text file.

The entire pipeline presented above can be easily adapted to new network structures. Putting details aside, it is fare to say, that integration of new network into our object detection framework is narrowed down to defining only two main parts of the networks that mostly differ from one to another, the actual skeleton of network in model_fn and the cost function loss_fn. The remaining parts are more or less identical.

6.2.3 SqueezeDetPlus

The graph 6.8 gives an overview of the entire skeleton of SqueezeDetPlus and its connections. The start point is *Dataset_input* node in the graph. The input size of SqueezeDetPlus is 375×1242. SqueezeDetPlus splits each image into a grid of size 22×76, each cell contains 9 anchors with different sizes and scales. In total we have $22 \times 76 \times 9 = 15048$ anchors spread over each image. In output layer each anchor has the following parameters: class scores, box coordinates $[center\_x,\ center\_y,\ width,\ height]$ and confidence. This is the resolution of output layer, a tensor with $[batch\_size,\ 22,\ 76,\ 9,\ c+4+1]$ shape, where c is the number of classes. The shape of each layer is written on the graph 6.8.

The label dictionary created in input_fn consists of the following tensors:

- **input_mask:** for each ground truth gt_i its IoU with all 15048 anchor boxes is calculated, among them the one with highest value will be associated to gt_i. If there is no anchor box intersecting with gt_i, the one with lowest euclidean distance to gt_i will be taken as the responsible anchor box. Then the index of responsible anchor boxes are gathered in an array *idxs*, note that no two ground truths can share a single anchor box. In the end the input_mask is generated by setting all elements at *idxs* to 1 and the rest to 0. input_mask shape: $[batch\_size,\ 15048,\ 1]$.

- **deltas:** for each ground truth gt_i its relative location to its associated anchor box is calculated and stored at location where the responsible anchor box lies, the rest elements are set to zero. deltas shape: $[batch\_size,\ 15048,\ 4]$.

- **labels:** a one_hot tensor. The locations represented by indices in labels take value 1, while all other locations take value 0. labels shape: $[batch\_size,\ 15048,\ c]$.

- **bounding_boxes:** original bounding boxes transformed to center format and stored at locations as their responsible anchor box, the rest elements are set to zero. bounding_boxes shape: $[batch\_size,\ 15048,\ 4]$.

finally a batch takes the following form: $[batch\_size,\ feature,\ label]$. At each pass of a batch through the network, the output of the last layer ($conv12$) will be parsed in $interpret\_output$ node. The sliced predictions and labels are then inputed into loss_fn. As depicted in the graph loss_fn is a collection of three different nodes:

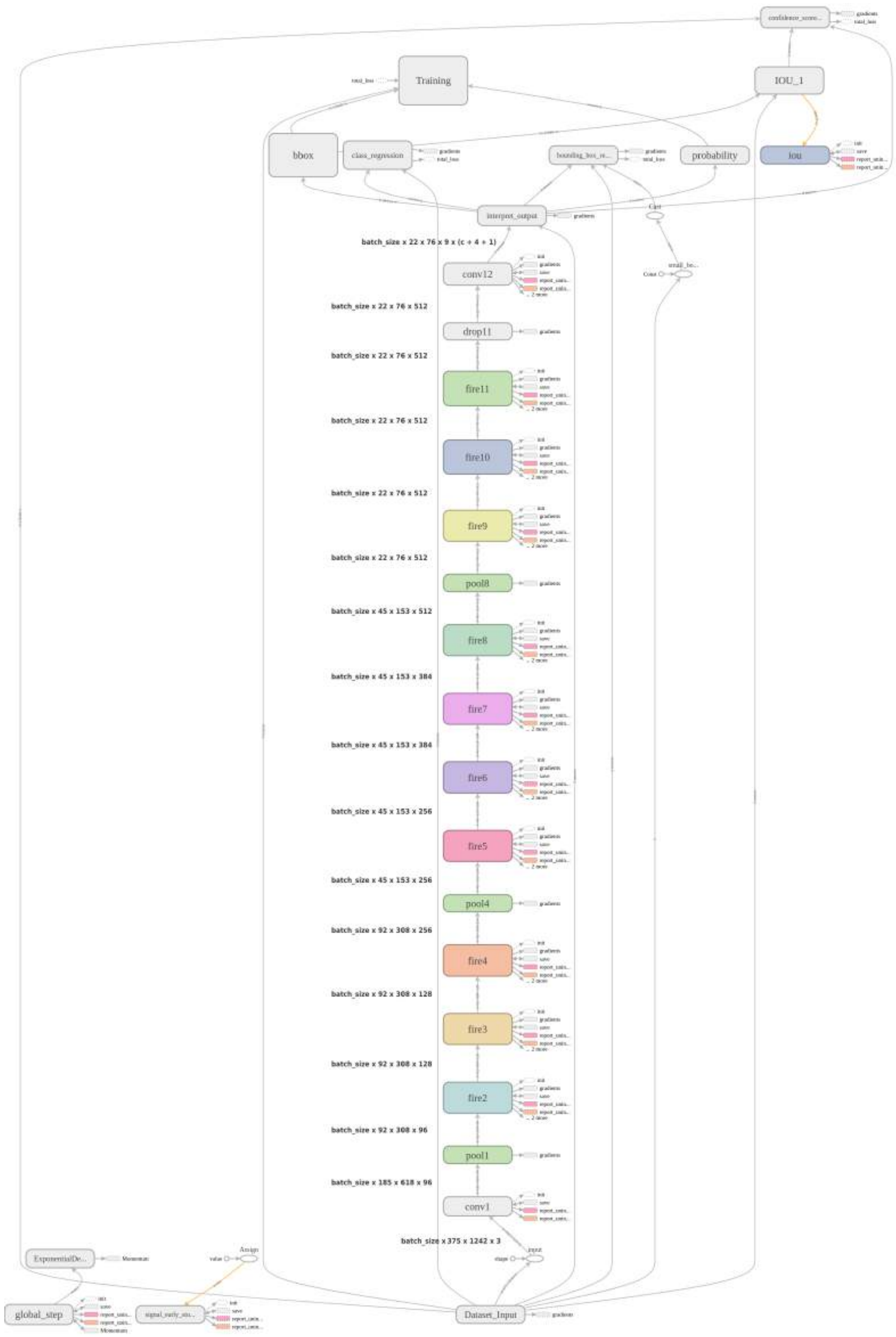

confidence_score...
Training
total_loss
IOU_1
bbox
class_regression
bounding_box_re...
probability
iou
interpret_output
Cast
small_bo...
Const
batch_size x 22 x 76 x 9 x (c + 4 + 1)
conv12
batch_size x 22 x 76 x 512
drop11
batch_size x 22 x 76 x 512
fire11
batch_size x 22 x 76 x 512
fire10
batch_size x 22 x 76 x 512
fire9
batch_size x 22 x 76 x 512
pool8
batch_size x 45 x 153 x 512
fire8
batch_size x 45 x 153 x 384
fire7
batch_size x 45 x 153 x 384
fire6
batch_size x 45 x 153 x 256
fire5
batch_size x 45 x 153 x 256
pool4
batch_size x 92 x 308 x 256
fire4
batch_size x 92 x 308 x 128
fire3
batch_size x 92 x 308 x 128
fire2
batch_size x 92 x 308 x 96
pool1
batch_size x 185 x 618 x 96
conv1
batch_size x 375 x 1242 x 3
input
shape
Assign
value
ExponentialDe...
Momentum
global_step
signal_early_sto...
Dataset_Input
gradients

Fig. 6.8 SqueezeDetPlus graph extracted from TensorBoard.

- class_regression
- bounding_box_regression
- confidence_score_regression

total_loss is the summation of these three nodes, which is in fact the objective to be minimized.

SqueezeDetPlus architecture is not complex and the layers forming the network are not the odd one out. The ***fire layer*** may not be as popular as other commonly used blocks, but technically it is just a combination of three other convolutional layers. The fire module is comprised of a ***squeeze layer*** as input, and two parallel expand layers as output. The squeeze layer is a 1×1 convolutional layer that compresses an input tensor with large channel size to one with the same batch and spatial dimension, but smaller channel size. The ***expand layer*** is a mixture of 1×1 and 3×3 convolution filters that takes the compressed tensor as input, retrieve the rich features and output an activation tensor with large channel size. The alternating squeeze and expand layers effectively reduces parameter size without losing too much accuracy.

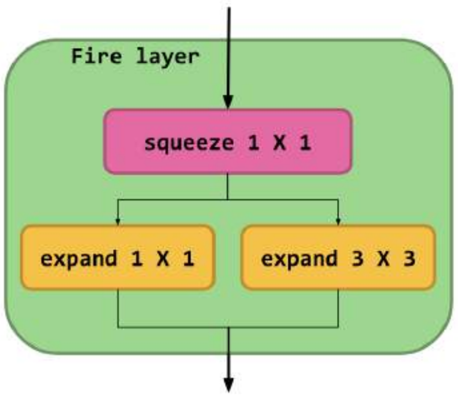


Fig. 6.9 Fire Module of SqueezeDetPlus

All layers are either initialized with pre-trained weights or with Xavier, for this a custom 2D convolutional layer is defined. Pre-trained weights are stored as numpy array in a pickle file. To load them onto layers, the name and shape of arrays are compared with layers, if no match is found the layer will be randomly initialized using Xavier and ReLu.

Chapter 7

Evaluation

7.1 Overview

In this chapter, the evaluation of the approaches applied on SqueezeDetPlus to improve the detection performance on fisheye images will be presented. Several commonly used evaluation metrics are considered to assess the impact of the optimizations performed on SqueezeDetPlus. Additionally to measure the improvement in SqueezeDetPlus performance more specifically after augmentations few other metrics had been taken into account as well. Some of the evaluation metrics used will be discussed briefly to better understand the evaluation results.

7.2 Evaluation Metrics

Generally the overall performance of an object detector is narrowed down to classification and localization accuracy. There has been various proposals, mostly made through competitions like PASCAL VOC [5] or ImageNet, to quantify the classification and localization performance of detectors. Although metrics like mAP became a favored benchmark to judge and compare detectors, but the place for a universal metric is still empty, since they typically neglect some aspects in favor of generality. Over and above, forcibly we have to fix threshold values in most of the measurement procedures, which are used to distinguish ranges of values where the behavior predicted by the model varies in some important way, thus threshold values influence the ultimate conclusion. This could lead to inconsistent outcomes, as thresholds alternate. We shed light upon this issue during this chapter.

Nevertheless, identification of *True Positives* (TP), *False Positives* (FP), *True Negatives* (TN) and *False Negatives* (FN) is the very first step to evaluate a problem in machine learning.

Although for object detection this is not only limited to the classification, as already stated localization error in predicted bounding boxes has to be also rated. To take this into consideration we calculate the *Intersection Over Union* (IoU) between predicted bounding boxes ($bbox_p$) and ground truths ($bbox_{gt}$) for each class, as nicely depicted below in figure 7.1. Hereby the first threshold (iou_t) comes into action, the boundary between TP and FP predictions. Base on iou_t positive detections are interpreted as follow, once a $bbox_p$ exists with IoU over iou_t with a $bbox_{gt}$, that $bbox_{gt}$ is set as detected **TP** assuming that its predicted class matches. Other $bbox_p$ will be considered as **FP**. Finding the proper value of iou_t is debatable, as COCO [20] recommends to measure across various IoU threshold, while PASCAL VOC set a fix value for $iou_t = \mathbf{0.5}$. Note, the lower we set iou_t the less we value the localization accuracy among detectors. As a consequence detectors that are better skilled in localization will be put at a disadvantage, and conversely if we put more emphasis on localization by raising iou_t, detectors, more capable of solving classification problems, will be underrated. The whole issue casts doubt on the reliability of such evaluation metrics, therefore evaluation metrics and their corresponding parameters should be specifically selected and tailored for a problem. For this thesis we stick to a single value $iou_t = 0.5$. Anyhow, Negatives (TN and FN) should be also counted. All locations where no $bbox_p$ are predicted can be considered as **TN**, but this would be infinite, that makes the measurement of true negatives futile. Therefore we discard true negatives and count only **FN**, which are all undetected $bbox_gt$ that a detector missed.

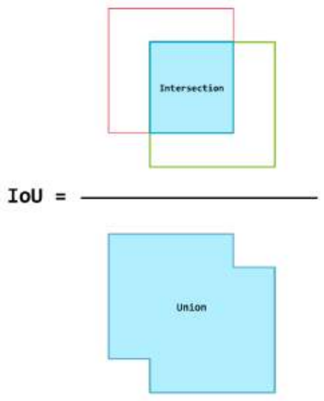


Fig. 7.1 Intersection over union.

- **Precision and Recall:** Having number of TP, FP and FN counted we can calculate the precision and recall for each class. Precision represents the accuracy of prediction, since it's the percentage of correct positive predictions as shown in equation 7.1.

$$precision = \frac{TP}{(TP+FP)} \tag{7.1}$$

  Replacing FP in the denominator of precision with FN yields the recall in equation 7.2, with other words the percentage of positively detected objects among all actual positive objects (ground truths).

$$recall = \frac{TP}{(TP+FN)} \tag{7.2}$$

Unfortunately, precision and recall are often in tension. That is, improving precision typically reduces recall and vice versa. This tension is heavily dependent on a classification threshold, which is confidence score threshold in object detection problem. The confidence score threshold $conf_t$ is to be added to our collection of thresholds as well, the second free parameter in our evaluation system that plays decisive role and has strong impact on the outcomes.

- **Average Precision (AP):** Simply put, AP is the area under the Precision-Recall curve, which shows the correlation between precision and recall at different level of confidence scores. Figure 7.2 gives an example of precision vs recall curve for a single class. In order to generate this curve, we first sort descendingly all predictions according to confidence score. If we imagine this as a table, first column is the rank, second one is either True or False depending on state of the IoU threshold. At each rank (row of table) we calculate precision and recall given the total number of TP and FP occurred up to the current rank. Naturally as we approach the bottom of the table (lower levels of confidence), recall increases steadily, since the total number of FP rises as well. On the other hand precision follows a zigzagging trajectory downward, till it reaches its lowest value where recall is concurrently at its highest stage.

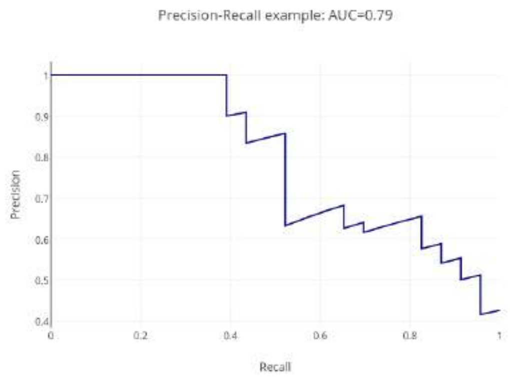


Fig. 7.2 Precision vs Recall. Example curve for a single class.

To compute an approximation of the area under the curve, we smoothen the curve by applying a sort of interpolation. conventionally at 11 different level of recall ($\hat{r} = [0, 0.1, 0.2, ..., 0.9, 1.0]$) we replace the precision value with the maximum precision for any recall equal or greater than $\hat{r}$. Eventually AP is computed as the average of maximum precision at these 11 recall levels:

$$AP = \frac{1}{11} \sum_{\hat{r} \in \{0.0,...,1.0\}} AP_{\hat{r}} \tag{7.3}$$

Ideally the slope of the precision and recall curve should be as lowest as possible, while precision tends tenaciously to stay close to one as recall converges to one. This would state, that a detector regardless of its certainty grade keeps scoring and barely misses out on possible positive objects for a particular class. Indeed no detector shows identical precision and recall curve across all classes due to the imbalance issue in datasets. Noteworthy, that for COCO [20] different AP is used for characterizing the performance of an object detector. AP across different IoUs [0.5, 0.75] and scales $[small : area < 32^2,\ medium : 32^2 < area < 96^2,\ large : area > 96^2]$.

- **Mean Average Precision (mAP):** To gain an overall view of the whole precision and recall curve among all classes mAP had been presented. Basically mAP is mean of all the Average Precision (AP) values across all classes as described above. While computing the mean, all APs are equally weighted, although the distribution of classes is non-uniform, therefore mAP is more a relative metric. By nature relativity brings uncertainty, on account of this one is advised to consider AP of individual class. However mAP might be moderate, but detector performance could be biased to a certain class. After all, mAP provides a single value that eases the comparison by putting particularity aside.

- **Receiver Operating Characteristic (ROC):** ROC is another tool to analyze test result of a detector or classificator generally. It brings *True Positive Rate* (TPR) and *False Positive Rate* (FPR) face to face at various threshold settings. TPR and FPR are known also as sensitivity (probability of detection) and specificity (probability of false alarm) respectively. The TPR is calculated as the number of true positives divided by the sum of the number of true positives and the number of false negatives. Similar to *Recall*, TPR describes also how good the model is at predicting the positive class when the actual outcome is positive.

$$TPR = \frac{TP}{(TP+FN)} \tag{7.4}$$

The FPR is calculated as the number of true negatives divided by the sum of the number of false positives and the number of true negatives. It summarizes how often a positive class is predicted when the actual outcome is negative.

$$\begin{aligned} Specificity &= \frac{TN}{(TN+FP)} \\ FPR &= 1 - Specificity \end{aligned} \tag{7.5}$$

An ROC curve plots TPR vs. FPR at different classification thresholds, which in object detection problem is confidence score. Lowering the classification threshold classifies more items as positive, thus increasing both FPs and TPs. At first glance ROC may appear resembling the precision and recall curve, although quite contrary to Precision, FPR takes true negatives into account. In other words it measures how many of negatives are falsely detected. Depending on how intense the imbalance in a dataset is and how we'd like to treat the negatives, we should choose between ROC and PR curve. Remarkably, ROC curve might not be an appropriate illustration for highly imbalanced data, since FPR does not drop drastically when the TN is large. Whereas Precision is highly sensitive to FPs and is not impacted by a large TN in denominator. Apart from this we will observe both ROC and PR curves in our evaluation. The following figure shows a typical ROC curve on the left plot. The sweet spot of ROC curve lies at upper left corner, where a detector neither misses out any positives (highest TPR) nor identifies any negatives as positive (FPR).

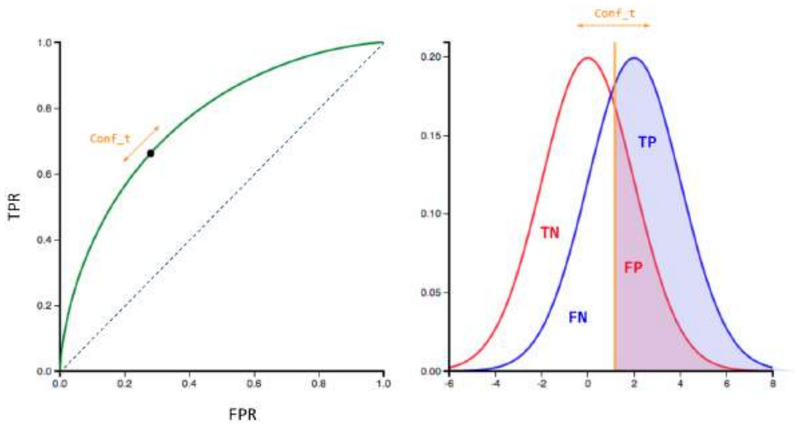


Fig. 7.3 ROC Curve compares the TPR and FPR at different level of confidence score determined with a threshold ($Conf_t$).

If we consider the result of a test in two populations, one population the TPs and the other one FPs we could plot them formed as normal distribution given confidence scores of the test results. As it can be seen on the left plot in figure 7.3 the two distributions overlaps. A perfect separation between the two groups could be barely observed. The vertical yellow line in figure 7.3 represents the confidence score threshold ($conf_t$), as we select different threshold value to discriminate the two populations the area under the two curves vary, which are actually fractions of TP, FP, TN and FN in the test result. Best case scenario is to have overlapping area between the two distributions as low as possible, this happens when the distance between the means of the two distributions is high, or the distribution representing the negatives (colored in red) is way smaller than the other one.

As we have seen the result of a test can be processed in various ways, yielding various interpretations. We put them all together to ensure a reliable judgment. Additionally in this thesis we should assess the performance of SqueezeDetPlus at edges, where objects are at most distorted, therefore we quantify the improvement of SqueezeDetPlus in these specific regions by measuring changes in confidence score and true positives rate of ground truths that fall into these areas. Since there is no reference value to determine the marginal areas and this is utterly use-case dependent, we perform evaluation across different sizes ($edge_t$) for edge area.

7.3 Results and Discussion

All the training and evaluation had been performed on *Nvidia Titan V* (Framebuffer: 12 GB HBM2 and 110 Deep-Learning TeraFLOPS). For evaluation purpose 100 images had been selected from the measurement dataset of IAV. This collected evaluation set (*IAV_Eval_Set*) contains images captured by the front fisheye camera. The images are hand-picked in a way that they cover a wide range of variation in environment (Day, Night, Snowy and etc.) and objects within. The images have 800×1280 resolution, for inference they were padded (filled with gray ($R:15, G:15, B:15$)) and scaled down to the size of the detectors, while preserving the aspect ratio.

At the outset we compare the result of evaluation on the original version of SqueezeDet-Plus trained with KITTI, annotated as *squeezeDetPlus-KITTI-O*, with our optimized version of SqueezeDetPlus using fisheye distortion and techniques explained in the chapter 5. Our modified version of squeezeDetPlus is trained on our combined dataset with regenerated annotations, BDD-KITTI-MASK. Therefore we abbreviate our detector into *squeezeDetPlus-BDD-KITTI-M-F*.

Our training dataset BDD-KITTI-MASK with refined bounding boxes thanks to Mask-RCNN made a major contribution to the promising results, which we'll present in the following, therefore we put BDD-KITTI-MASK and the original KITTI in comparison. The

Dataset	Car	Cycle	Pedestrian	Train	Truck
KITTI	28.742	1.627	4.487	511	1.094
BDD-KITTI-MASK	**588.480**	**15.852**	**101.755**	**10.104**	**25.989**

Table 7.1 Comparison in number of instances of similar class between KITTI and combined BDD-KITTI-MASK dataset with labels regenerated by MASK-RCNN.

table 7.1 shows us the total number of instances per class in both KITTI and BDD-KITTI-MASK. Note, that the classes are not identical, due to different classes in the pre-existing KITTI labels and classes on which MASK-RCNN was trained. To unify them, we merged the similar classes together. For the evaluation we only took *car*, *pedestrian* and *cyclist* into consideration, since squeezeDetPlus was trained only on this three classes originally. As the table 7.1 states, the number of instances in BDD-KITTI-MASK is significantly higher, in average each class in BDD-KITTI-MASK has 19.2 times more examples. Considering the rectified ground truth boxes and higher variance in images plus the total amount of examples, it's hardly surprising that BDD-KITTI-MASK is more of a delicate food for a network than KITTI. It should be mentioned, that putting KITTI and BDD with different sizes generated a sort of multi scale dataset, since we pass KITTI images in original size (which fits the input size of the squeezeDetPlus), and apply scale down and padding on BDD images.

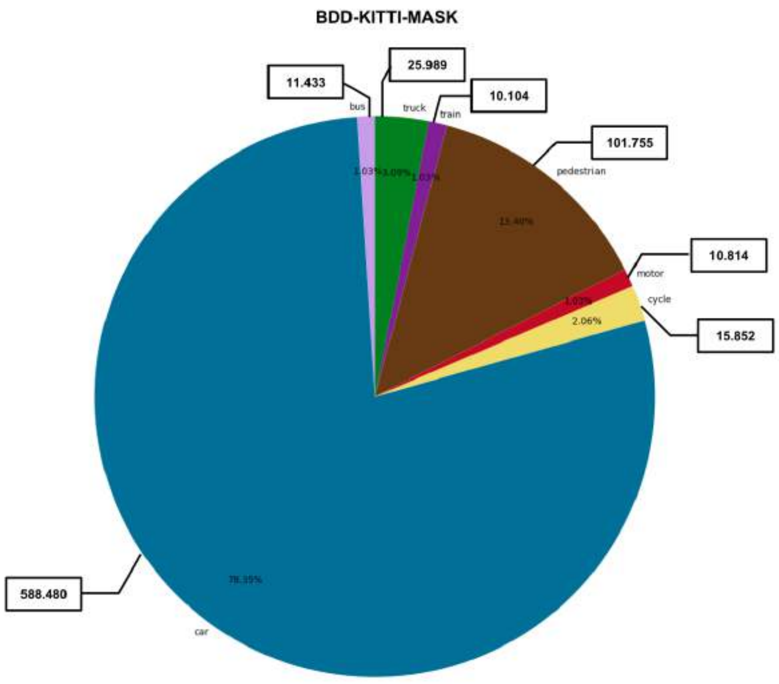


Fig. 7.4 Portion of classes in BDD-KITTI-MASK. It shows also the number of instances of each class.

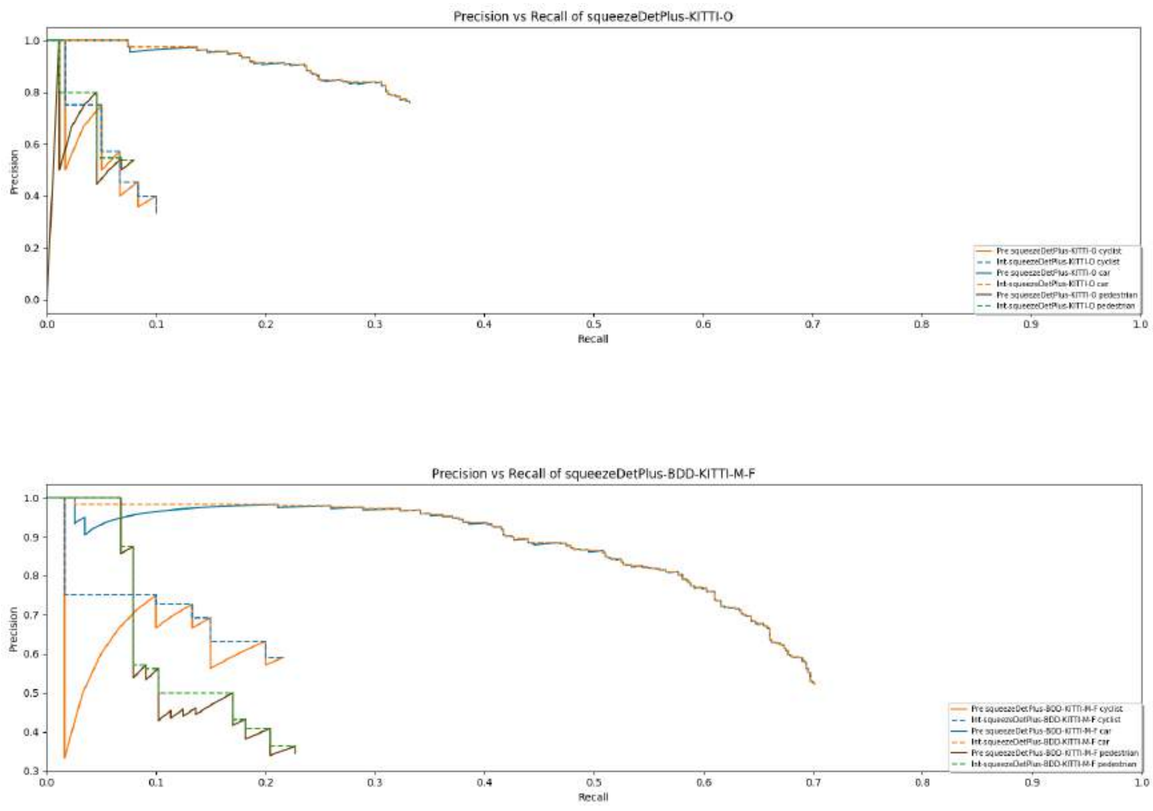


Fig. 7.5 Comparison of Precision and Recall curve between the original version of SqueezeDetPlus trained on KITTI and our modified version trained on BDD-KITTI-MASK applying fisheye distortion.

After all the imbalance in dataset is still hard to resolve, specially when given images contain unequal amount of objects. It is obvious, that one would confront more cars than pedestrians or cycles while driving in streets. The pie chart 7.4 provides us information about what portion of BDD-KITTI-MASK is taken by which class. As it's apparent, the class *car* still predominates as expected. *car* and *pedestrian* have a higher portion of the entire dataset in BDD-KITTI-MASK (car: 78.35%, pedestrian: 13.40%), compared to KITTI (car: 72.45%, pedestrian: 11.22%).

At the first step we compare the precision and recall curve of both squeezeDetPlus-KITTI-O (figure at the top) and squeezeDetPlus-BDD-KITTI-M-F (figure at the bottom) in the figure 7.5. As it can be seen, squeezeDetPlus-BDD-KITTI-M-F simply outperforms squeezeDetPlus-KITTI. Taken the class *car*, as squeezeDetPlus-BDD-KITTI-M-F passes recall 0.7, squeezeDetPlus-KITTI-O barely reaches 0.35. It states, that squeezeDetPlus-BDD-KITTI-M-F first detects with much higher accuracy, second it's incomparably better at finding objects (positives). Although the domination of the class *car* leads to such uneven performance over all classes, as repeatedly discussed, but squeezeDetPlus-BDD-KITTI-M-F shows generally a higher performance in comparison to squeezeDetPlus-KITTI-O. To prove this, we better compare the AP of the both detectors across all classes as well, as shown in the table 7.2.

Model	Car	Cycle	Pedestrian
squeezeDetPlus-KITTI-O	0.396	0.127	0.125
squeezeDetPlus-BDD-KITTI-M-F	**0.731**	**0.291**	**0.164**

Table 7.2 AP over class. Over all three classes significant improvement is to see.

To have a general quantification we might want to cast a glance at mAP, but to not overlook the detail and in order to retain the specification in our evaluation we put mAP across different sizes of ground truth together, as suggested in COCO. Figure 7.6 shows the mAP on various scales of boxes, thereby the number of TPs over the total number of positives. squeezeDetPlus-BDD-KITTI-M-F performs in all scales better than squeezeDetPlus-KITTI-O. Interestingly squeezeDetPlus-BDD-KITTI-M-F precision increases continuously till it reaches its highest mAP where the objects are large, it underpins the fact that small objects are tedious and difficult to detect. Nonetheless squeezeDetPlus-BDD-KITTI-M-F spotted 63 of 150 small objects, while squeezeDetPlus-KITTI-O missed nearly all of them.

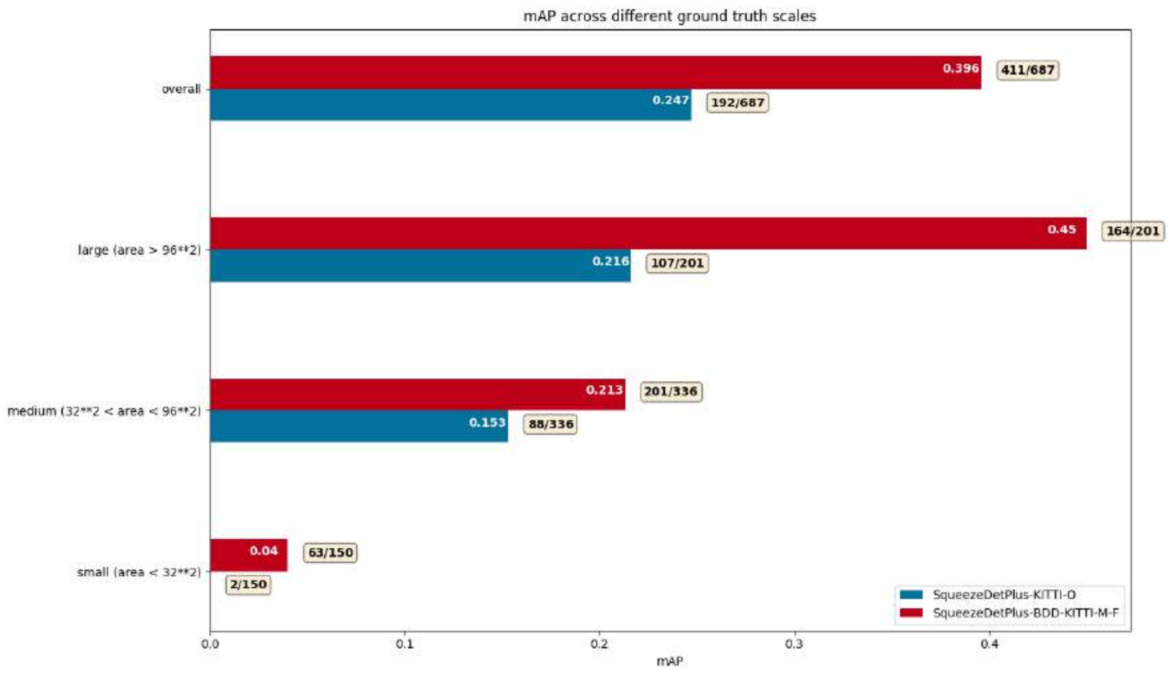
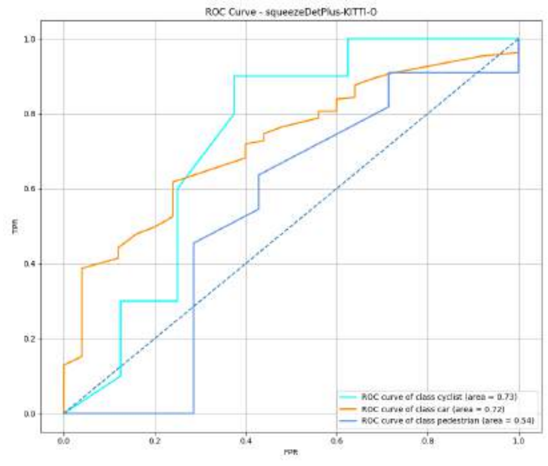
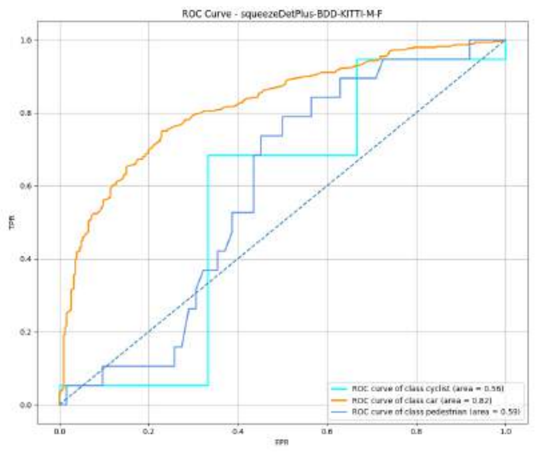


Fig. 7.6 mAP across different ground truth box sizes.



(a)



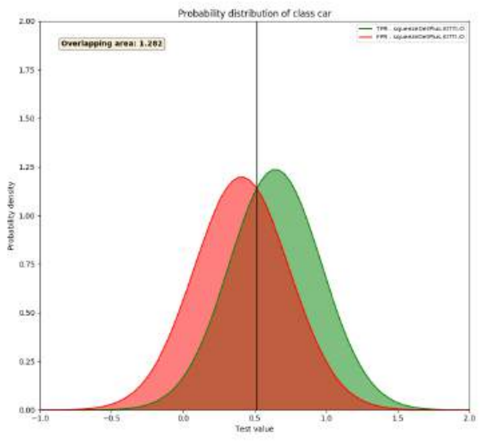
(b)

Fig. 7.7 Comparison of ROC between squeezeDetPlus-KITTI-O and squeezeDetPlus-BDD-KITTI-M-F.

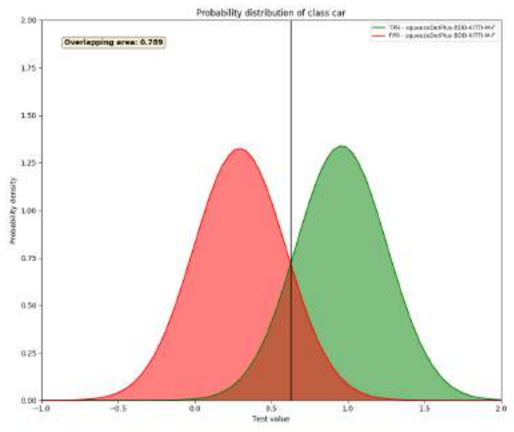
Noteworthy, that the same squeezeDetPlus-BDD-KITTI-M-F without small box exclusion (explained in the chapter 5) detected only **23** objects out of 150, **40** objects less in comparison to squeezeDetPlus-BDD-KITTI-M-F trained under same condition, but excluding the small ground truths out of training.

To shed more light on the impact of our fisheye augmentation, we could scrutinize how training with distorted objects improves the confidence of the network in detections. For this, ROC curve is a suitable tool to examine the performance of the network at different levels of confidence. In the figure 7.7 we compare the ROC of squeezeDetPlus-KITTI-O (7.7a) and squeezeDetPlus-BDD-KITTI-M-F (7.7b). The area under the curve (AUC) can be interpreted as the skill grade of the network in detecting each class, as it's clearly evident the AUC of squeezeDetPlus-BDD-KITTI-M-F is higher than squeezeDetPlus-KITTI-O, which implies squeezeDetPlus-BDD-KITTI-M-F is better skilled to detect cars or pedestrians, due to its familiarity level with features of distorted objects, with other words it has more sensible neurons against characteristics of distorted objects affected by fisheye lens.

This capability of distinguishing positive distorted objects from negatives can be further examined, as we make a comparison between the distribution of confidence scores about positives and negatives detection by the detector. The figure 7.8 illustrates the confidence distributions of squeezeDetPlus-BDD-KITTI-M-F at the right side (7.8b) and squeezeDetPlus-KITTI-O at the left side (7.8a). As discussed earlier in this chapter, we seek a perfect separation between these two population/distribution. The more they distance from each



(a) Overlapping area: 1.282



(b) Overlapping area: **0.789**

Fig. 7.8 Distribution of positives and negatives. a) squeezeDetPlus-KITTI-O b) squeezeDetPlus-BDD-KITTI-M-F.

other the less overlapping area they will have, this would mean that the detector has the ability to discriminate positives from negatives. The figure 7.8 provides the overlapping areas, squeezeDetPlus-BDD-KITTI-M-F with **0.789** and squeezeDetPlus-KITTI-O 1.282. As it is apparent, squeezeDetPlus-BDD-KITTI-M-F succeeded to find a dividing line to have more accurate distinguishability than squeezeDetPlus-KITTI-O. Hereby a considerable amount of credit goes to our fisheye augmentation.

Last but not least, in order to prove the benefit of the fisheye augmentation to empower a CNN-based detector to deal with fisheye images, we peruse the detection accuracy of squeezeDetPlus-BDD-KITTI-M-F in marginal areas, where the objects are deformed at most due to the barrel distortion. The figure 7.9 illustrates average confidence score of true positives in edge areas, given four different ratios. The closer we approach the boundaries of the image, the more uncertain are the both detectors about their decision, as anticipated. But squeezeDetPlus-BDD-KITTI-M-F shows satisfyingly a higher rate of certainty, compared to squeezeDetPlus-KITTI-O.

Based on all evaluation results presented so far, it's evident, that our modified version of squeezeDetPlus simply outperformed the original version. In order to widen the scope of our assessment, next we put the test result of squeezeDetPlus-BDD-KITTI-M-F next to other worthy state-of-the-art detectors.

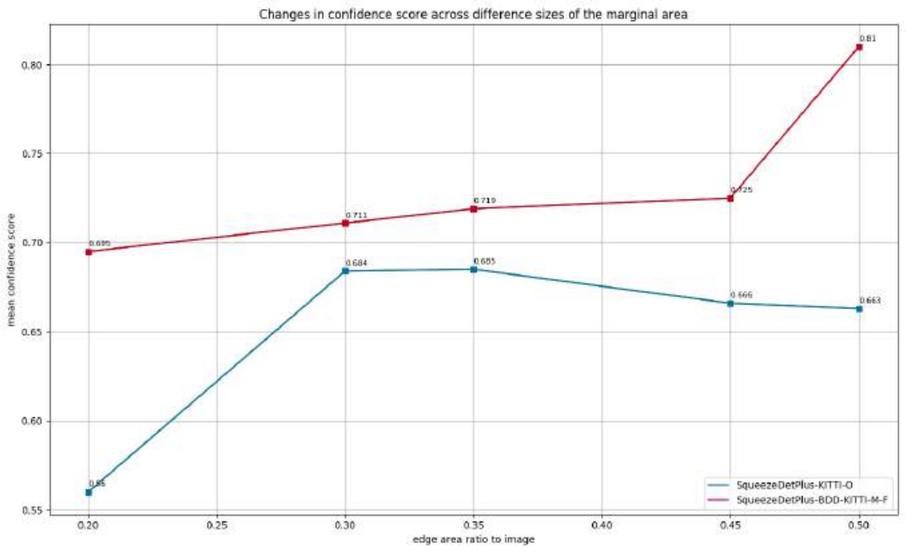


Fig. 7.9 Comparison of mean confidence score of true positives in edge areas. Size of the edge area is plotted on the x axis.

7.4 Comparison to State-of-the-Art

To present a general view of how different squeezeDetPlus-BDD-KITTI-M-F and other state-of-the-art detectors from both single and two stage families perform on our *IAV_Eval_Set*, we plotted their mAP in the figure 7.10. The selected detectors are as follow:

- **Faster-rcnn-resnet50-coco:** Faster-RCNN trained on COCO using ResNet50 as feature extractor.
- **RFCN-resnet101-coco:** Region-based Fully Convolutional Networks (RFCN) trained on COCO using ResNet50 as feature extractor.
- **SqueezeDetPlus-BDD-O:** SqueezeDetPlus trained on BDD with original configurations.
- **SqueezeDetPlus-UDACITY-Org:** SqueezeDetPlus trained on UDACITY with original configurations.
- **SSD-mobilenet-v1-coco:** Single Shot MultiBox Detector (SSD) compressed version for mobile devices trained on COCO.
- **SSD-resnet50-v1-fpn-coco:** Single Shot MultiBox Detector (SSD) full net trained on COCO using Feature Pyramid Network (FPN) as extractor.
- **Yolov3:** Original Yolov3 in darknet framework trained on COCO.

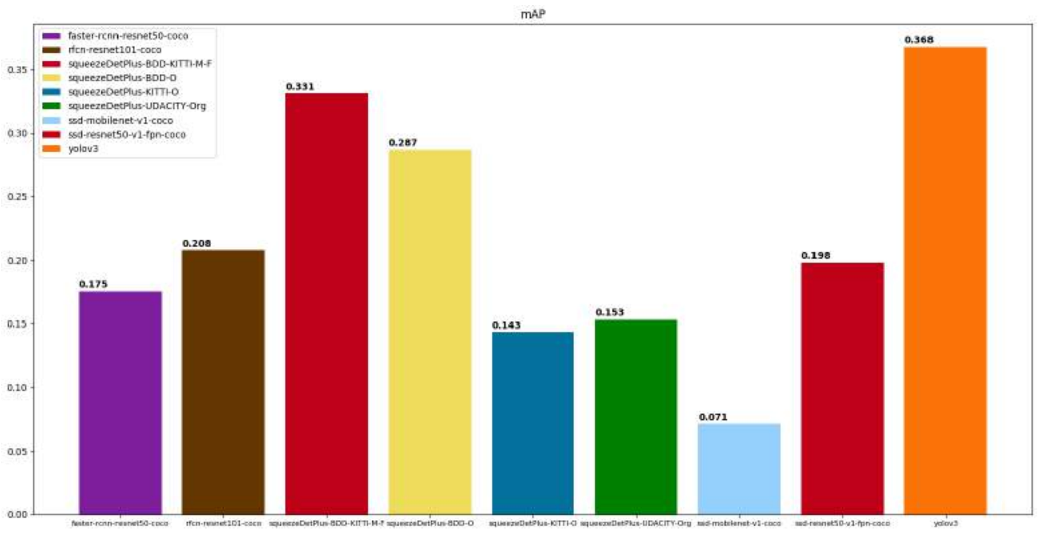


Fig. 7.10 Comparison of mAP between state-of-the-art detectors and squeezeDetPlus-BDD-KITTI-M-F.

To test these models, we used the implementations provided in Tensorflow Object Detection API, except Yolov3 that is originally implemented in Darknet. Contrary to our expectations even two-stage detector like Faster-RCNN fall short when dealing with distorted images. The only detector that tops our squeezeDetPlus-BDD-KITTI-M-F is the novel **Yolov3** with $mAP = 0.368$. The worst performance belongs to SSD-mobilenet-v1-coco, which is not a surprise due to the compression of the model. We pick the front runner, Yolov3, among other to compare more precisely.

As we elaborated before, mAP is a relative metric, since it averages precisions of all classes. To hinder this, once again we compare single AP per class. As it can be spotted in the figure 7.11, squeezeDetPlus-BDD-KITTI-M-F performs on cars slightly even better than Yolov3. But this is not sufficient to put confirmation stamp on this statement that squeezeDetPlus-BDD-KITTI-M-F catches Yolov3 up. The following information may sustain this.

In table 7.3 we compare the mean confidence of Yolov3 and squeezeDetPlus-BDD-KITTI-M-F in marginal areas. On the last column the confidence of detections is presented, which fall at the center of the image within a frame with 0.5 of the image size. Interestingly Yolov3 appears to be more confident regardless of the location of the objects and in centric region performs at its best. Among several possible reasons, Yolov3's larger receptive field and deeper architecture could explain its superior performance to squeezeDetPlus-BDD-KITTI-M-F.

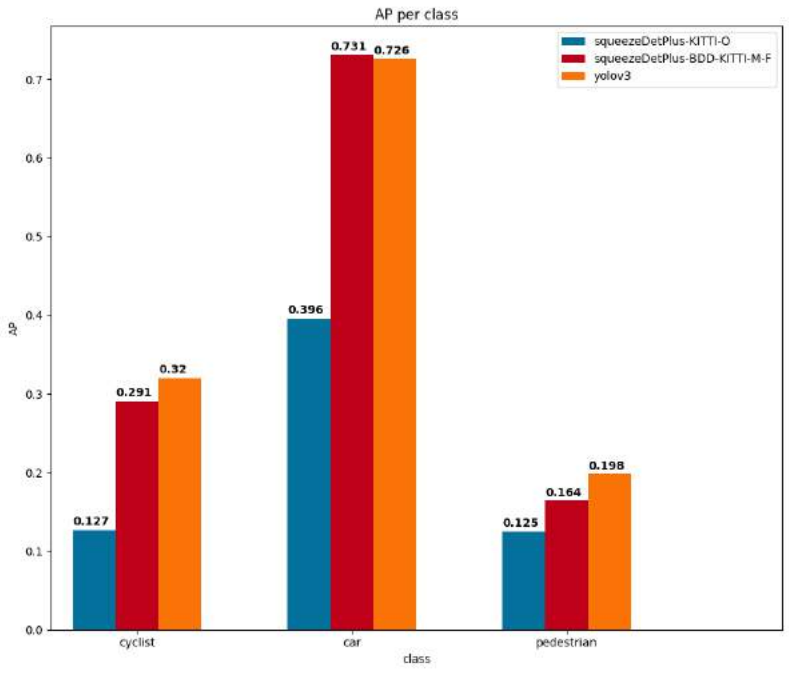


Fig. 7.11 Comparison of AP over classes.

Large object are the Achilles' heel of squeezeDetPlus. For instance in a scene where a pedestrian or a car stands close to the camera, squeezeDetPlus will not be able to see the car or pedestrian. In fact due to its narrow receptive field very large objects are invisible to squeezeDetPlus, while Yolov3 catches large objects unproblematically. To underpin this statement we compare the recall density of Yolov3, squeezeDetPlus-BDD-KITTI-M-F and squeezeDetPlus-KITTI-O over small (figure 7.12), medium (figure 7.13) and large(figure 7.14) objects. For the sake of clarification, we desire to have high density (y-axis) at higher recall values (x-axis), this would mean that a detector has lower FP, intuitively it has sharper eye to spot objects at a particular scale. As figure 7.14 illustrates, Yolov3 has higher recall density pretty much in the upper half of recall ($recall > 0.5$), where squeezeDetPlus-BDD-KITTI-M-F still has difficulties in spite of its improvements in comparison to squeezeDetPlus-KITTI-O.

Model	0.2	0.3	0.35	0.45	0.5 from center
squeezeDetPlus-BDD-KITTI-M-F	0.695	0.711	0.719	0.725	0.715
Yolov3	0.8	0.85	0.868	0.86	**0.881**

Table 7.3 Mean confidence score across different sizes of the marginal area. Due to the larger receptive field and total number of anchors Yolov3 has generally higher confidence score.

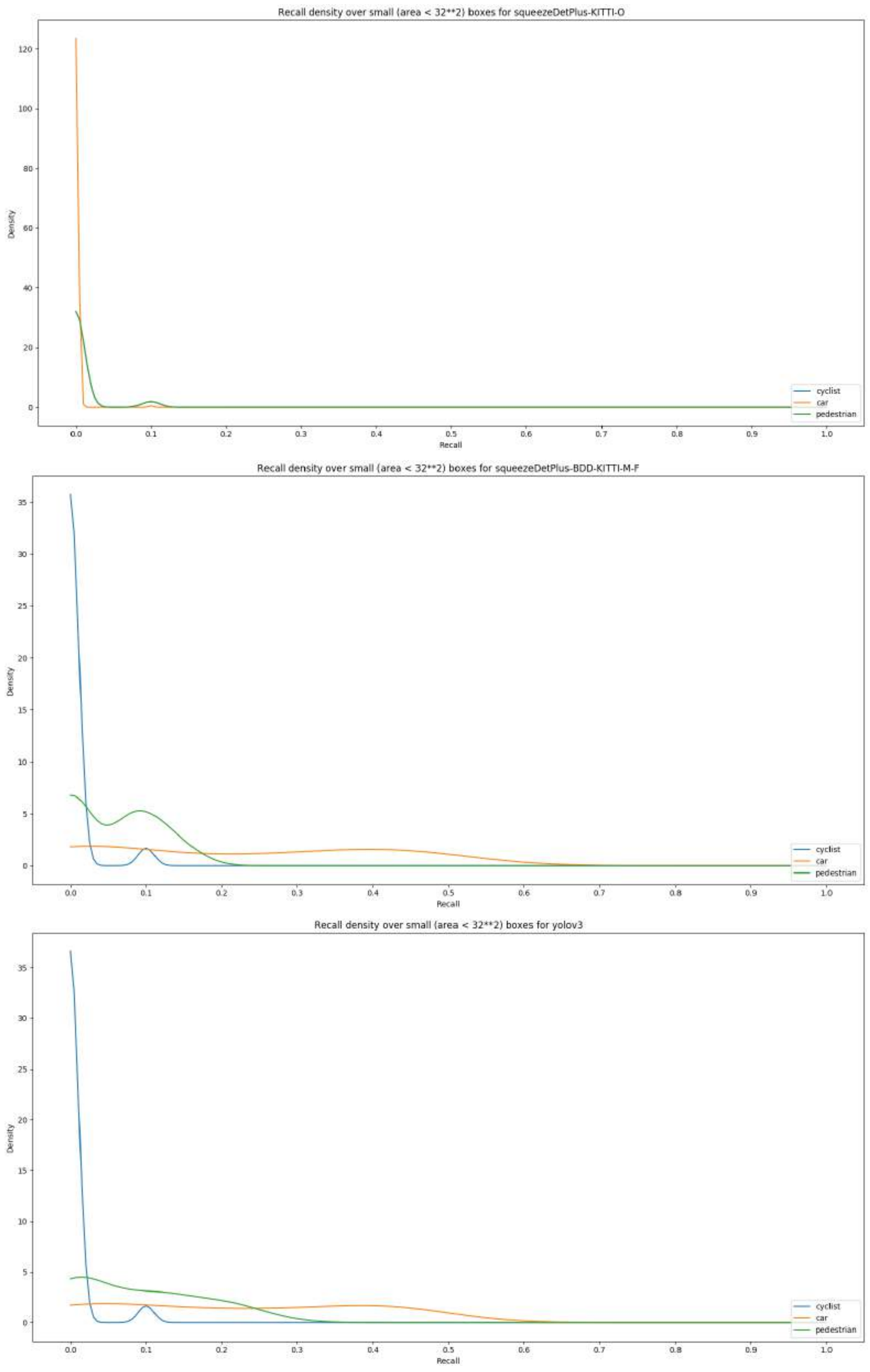


Fig. 7.12 Recall density over **small** boxes.

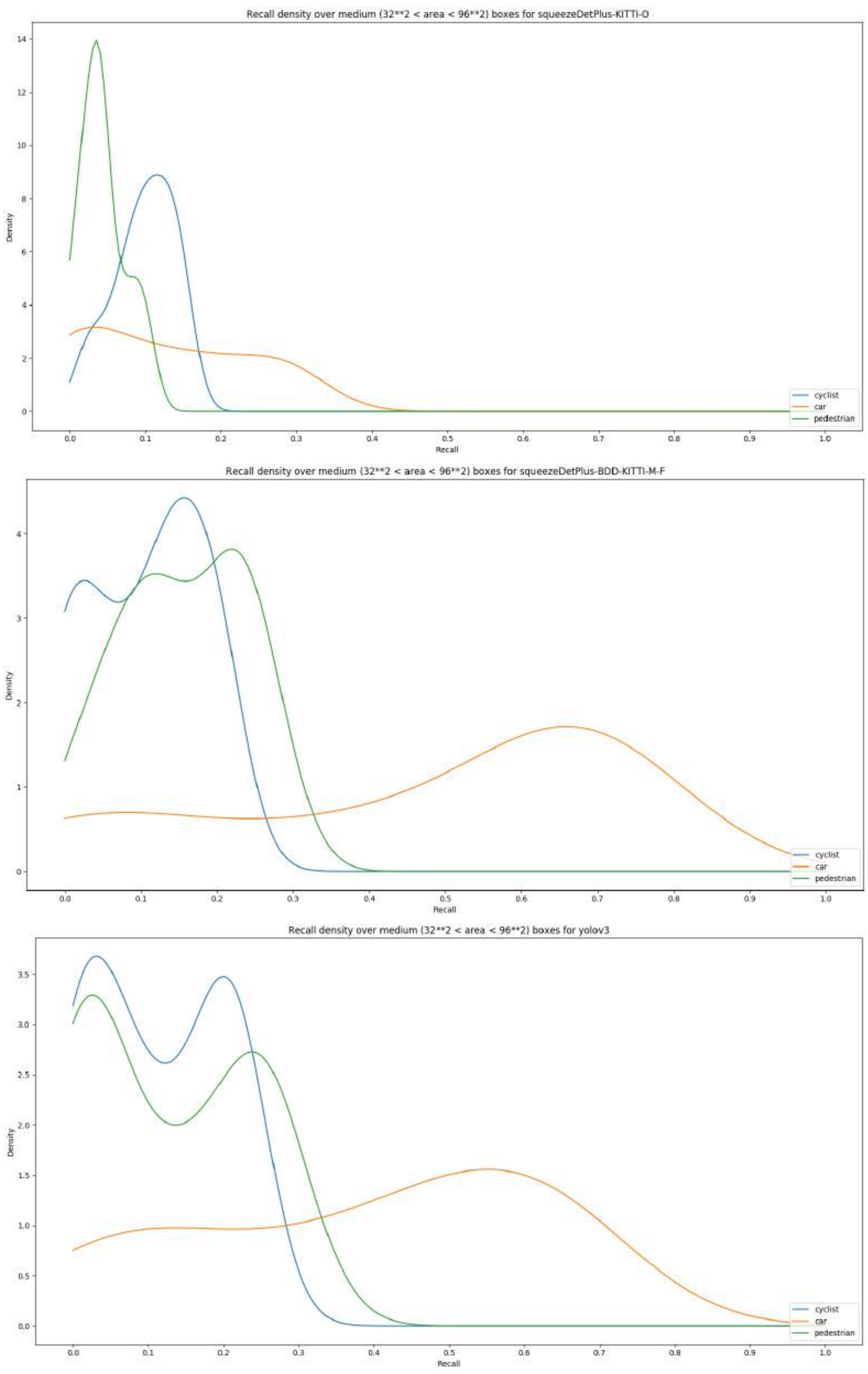


Fig. 7.13 Recall density over **medium** boxes.

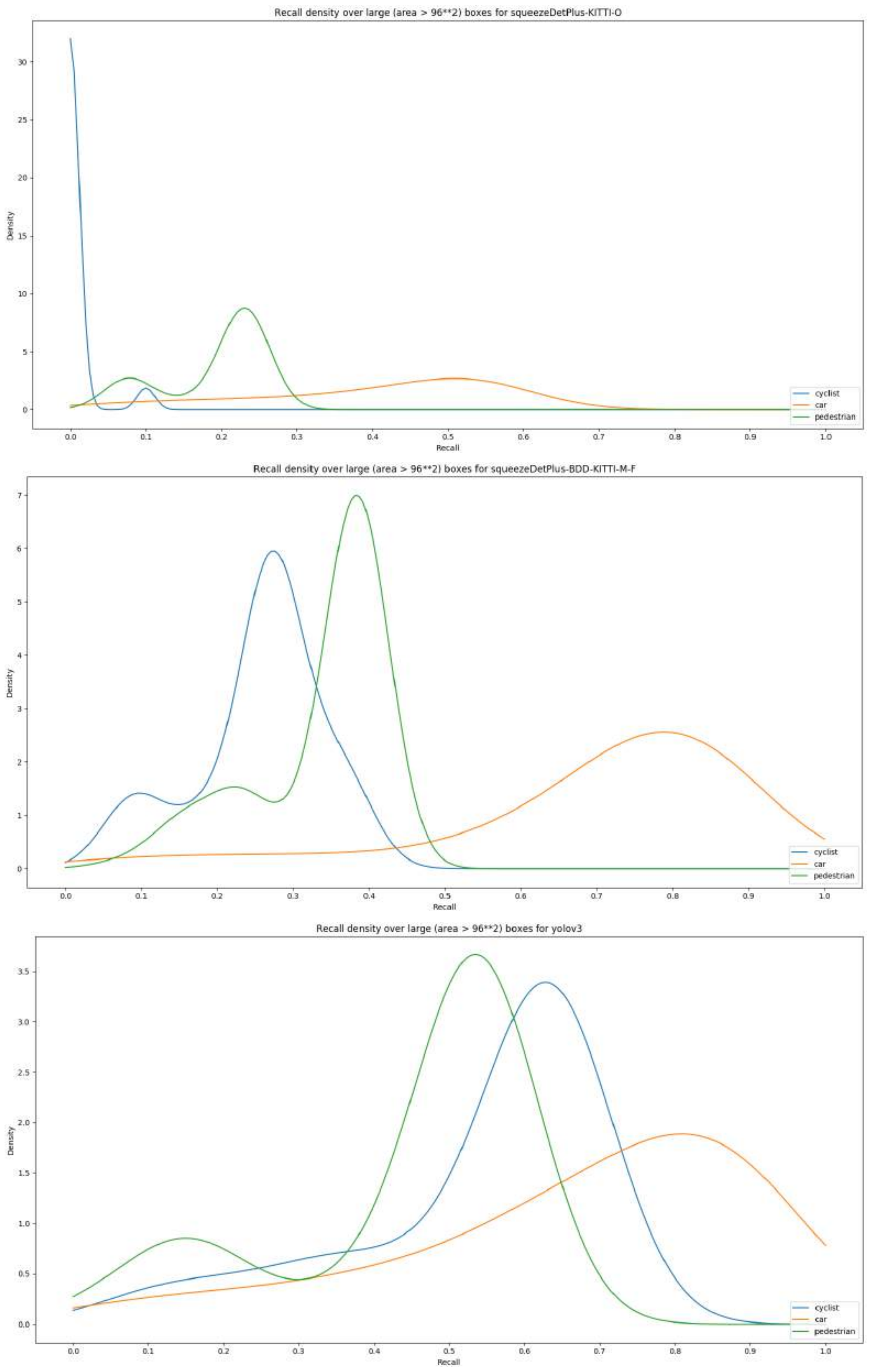


Fig. 7.14 Recall density over **large** boxes.

7.5 Example Images

To see squeezeDetPlus-BDD-KITTI-M-F in action, we show several example images from IAV_Eval_Set, where detection before (squeezeDetPlus-KITTI-O) and after applying our fisheye augmentation (squeezeDetPlus-BDD-KITTI-M-F) are put together.

In figure 7.15 as we can see squeezeDetPlus-KITTI-O misses the car at the left edge of the image plus other cars in the distance, while squeezeDetPlus-BDD-KITTI-M-F easily detects them, interestingly with high confidence score ($0,8$ for the car at the left corner). Probably the rotation caused by the fisheye lens appears to squeezeDetPlus-KITTI-O unusual, as a consequence it's not able to detect it. The example image 7.16 is a similar situation where squeezeDetPlus-KITTI-O incapable of handling the distorted cars, while squeezeDetPlus-BDD-KITTI-M-F flawlessly spot them. Note, that all predicted bounding boxes of squeezeDetPlus-BDD-KITTI-M-F are fitter than ones from squeezeDetPlus-KITTI-O, as table 7.4 also shows squeezeDetPlus-BDD-KITTI-M-F has in average %5 higher IoU than squeezeDetPlus-KITTI-O. The image in figure 7.17 is a dangerous situation in real world where failure could cost a human life, as a pedestrian stands on the street, very close to the car. As driving autonomously all moving objects have to be detected and tracked, squeezeDetPlus-KITTI-O won't be reliable detector for such circumstances. On the other hand squeezeDetPlus-BDD-KITTI-M-F proves once again its skill at tracking down deformed objects due to the fisheye effect.

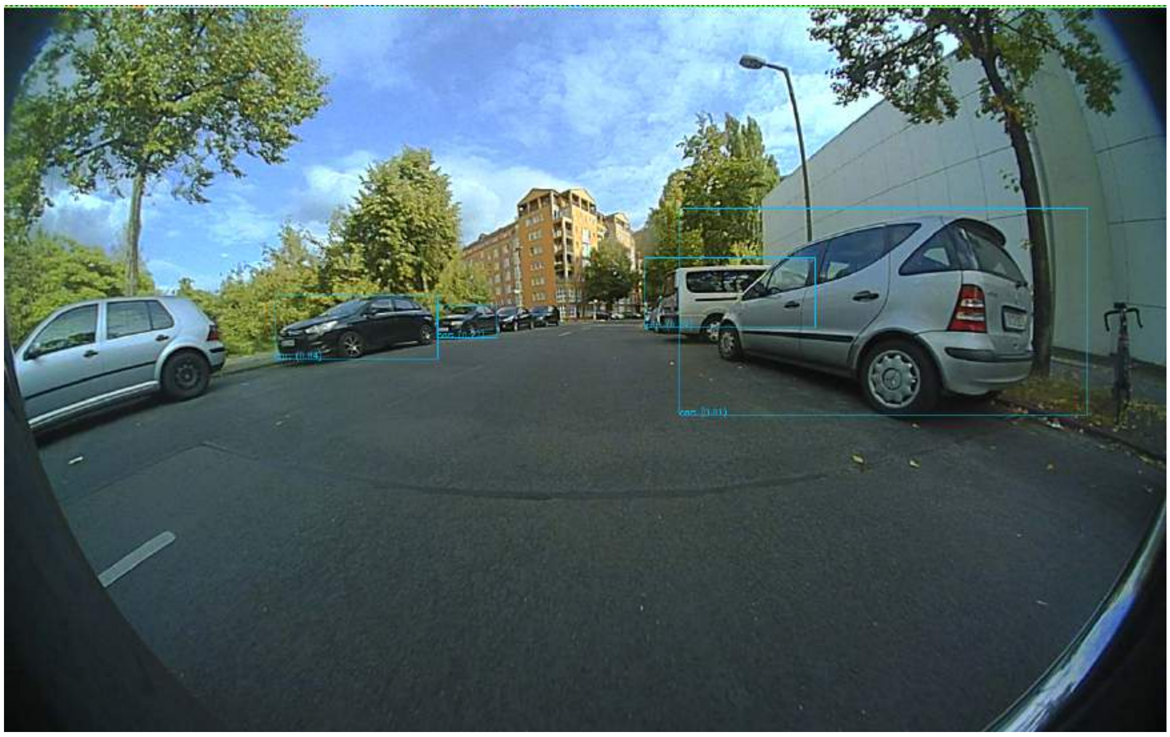
Both figure 7.18 and 7.19 show cars that are magnified through fisheye distortion, very large objects are CNN-based detectors Achilles' heel as well. squeezeDetPlus-KITTI-O is neither an exemption from this, and as it can be seen in 7.19 and 7.18 it has difficulties to find over-scaled objects, astonishingly squeezeDetPlus-BDD-KITTI-M-F gets by.

In order to take the illumination variation into consideration we take look at the example image in the 7.20, where we have a night scene with indistinct cars, a relatively tedious situation for any detector. But squeezeDetPlus-BDD-KITTI-M-F detects both fuzzy cars with confidence higher than 0.8, which is very satisfactory.

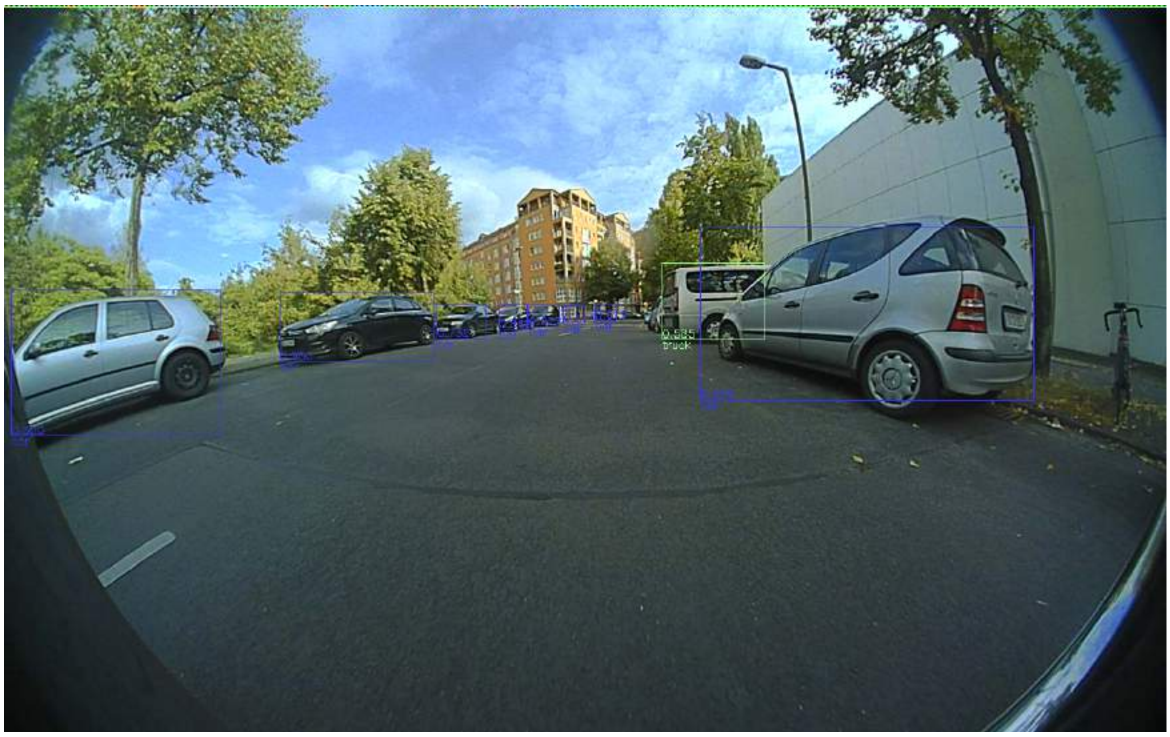
Model	Mean IoU
squeezeDetPlus-KITTI-O	0.65
squeezeDetPlus-BDD-KITTI-M-F	**0.70**
Yolov3	**0.73**

Table 7.4 Mean IoU over TPs with $iou_t = 0.5$.

The last example image 7.21 is another scenario, where flawless detection is vital. squeezeDetPlus-BDD-KITTI-M-F unproblematically detects the cyclist riding across the street from the edge of the image. This is crucial for decision making while driving.

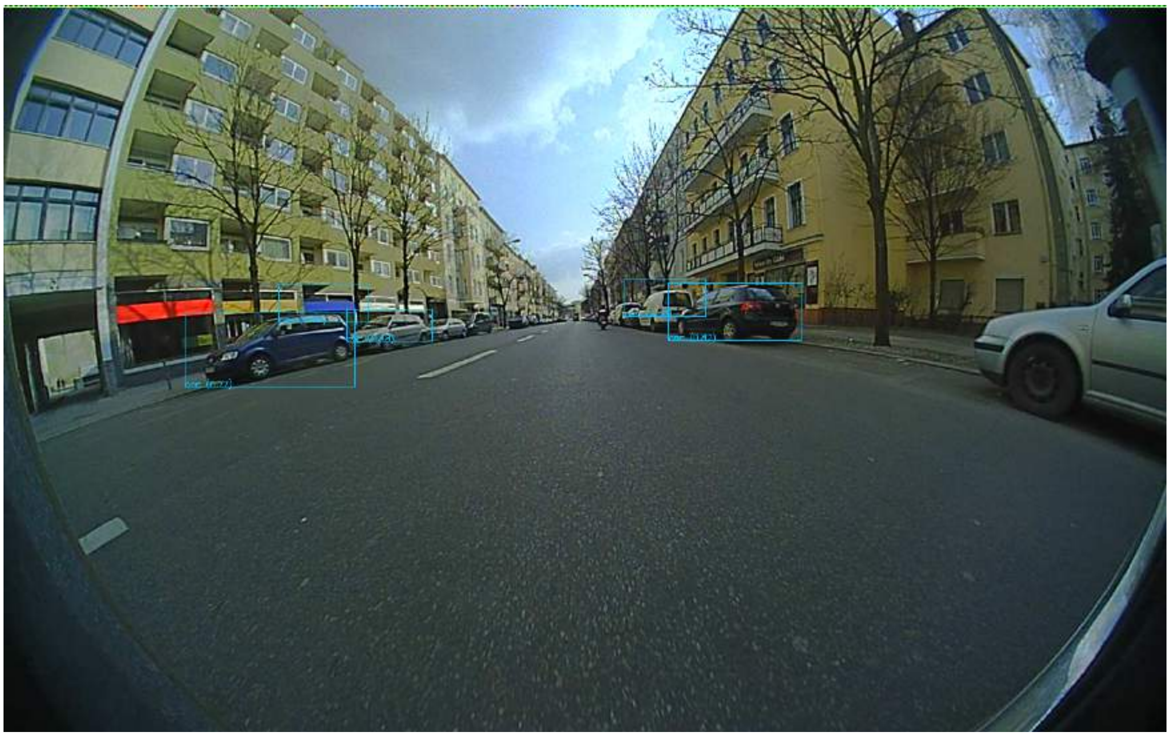


(a) squeezeDetPlus-KITTI-O

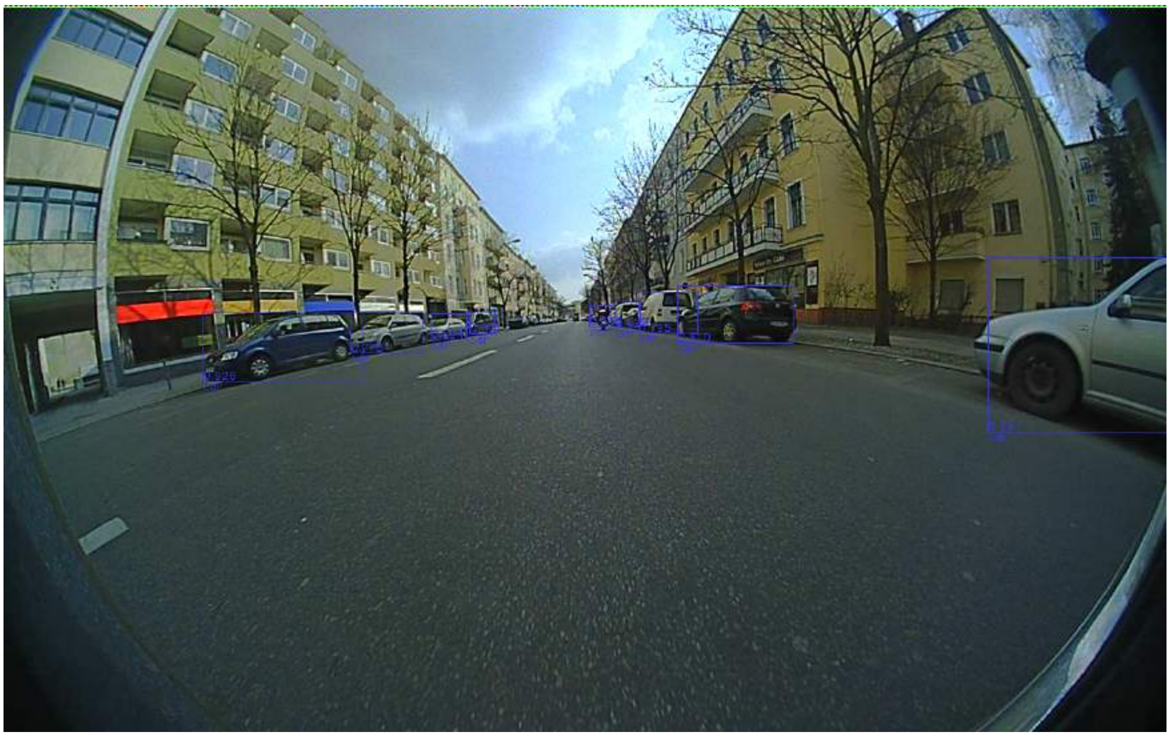


(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.15 Example Image from *IAV_Eval_Set*.

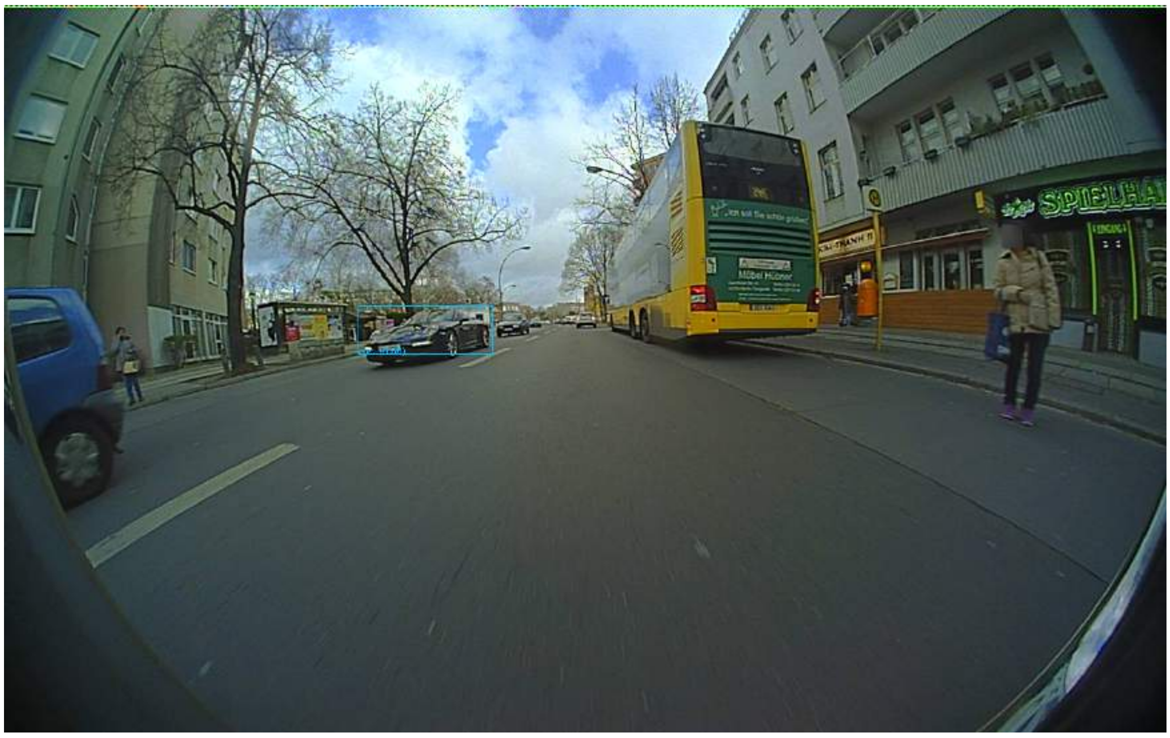


(a) squeezeDetPlus-KITTI-O

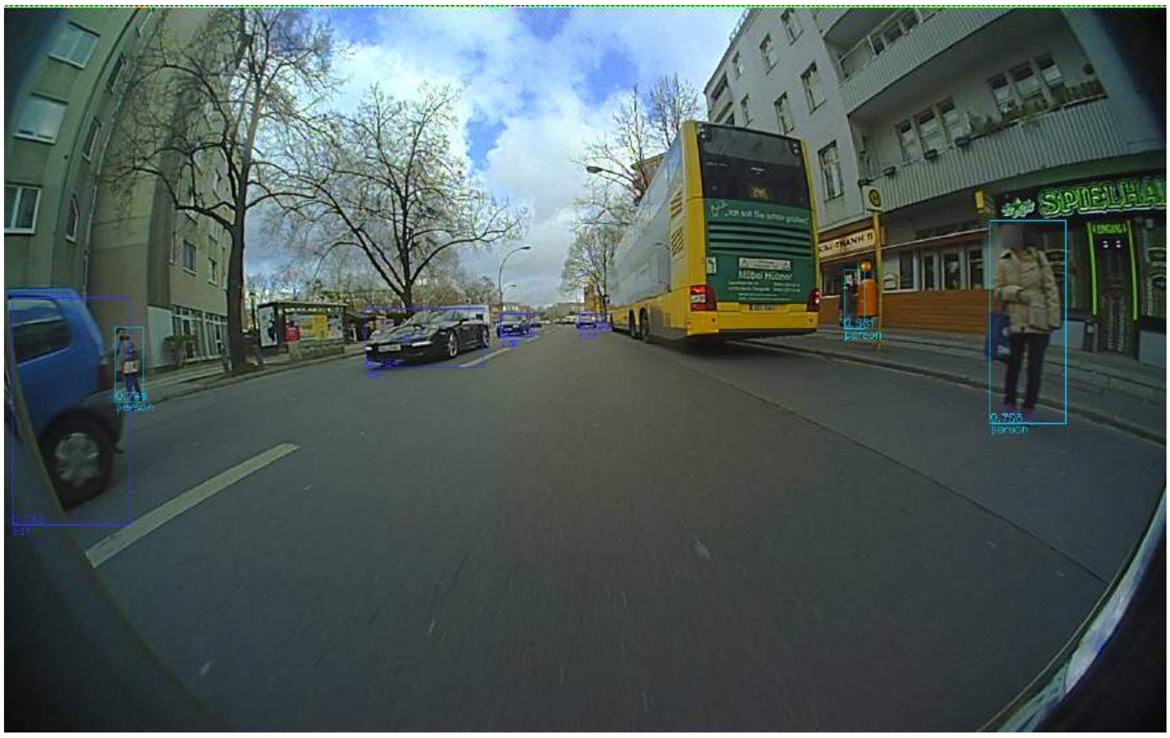


(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.16 Example Image from *IAV_Eval_Set*.

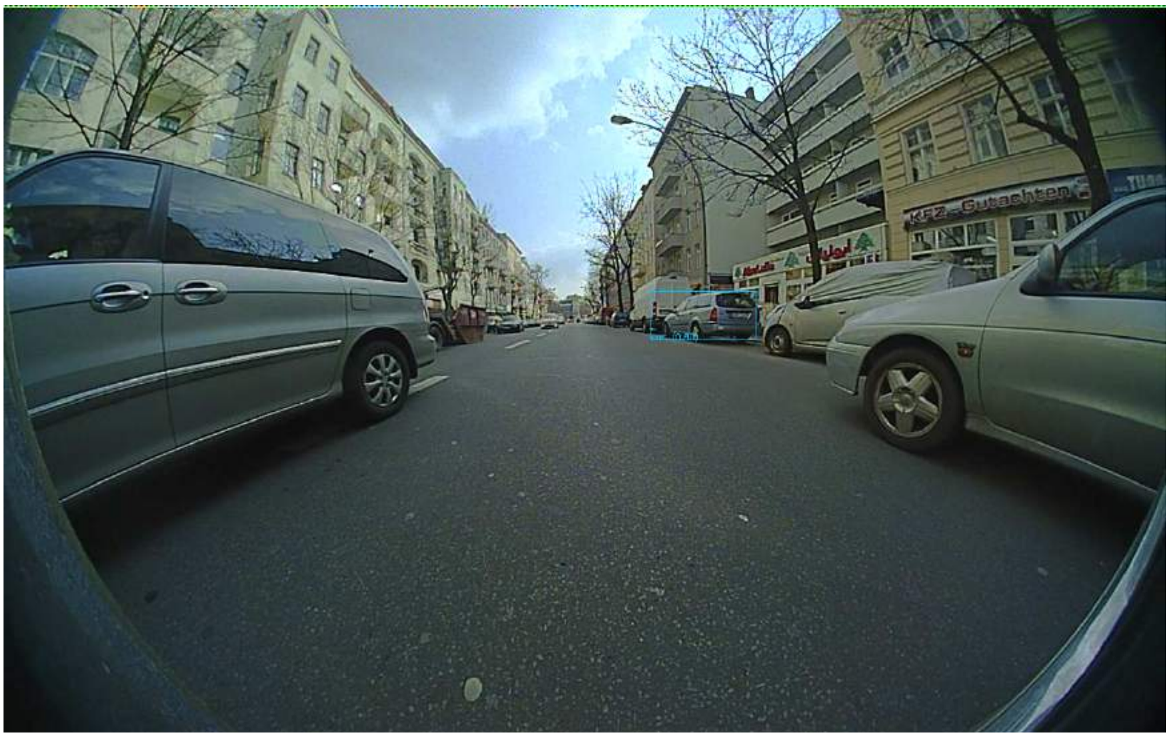


(a) squeezeDetPlus-KITTI-O

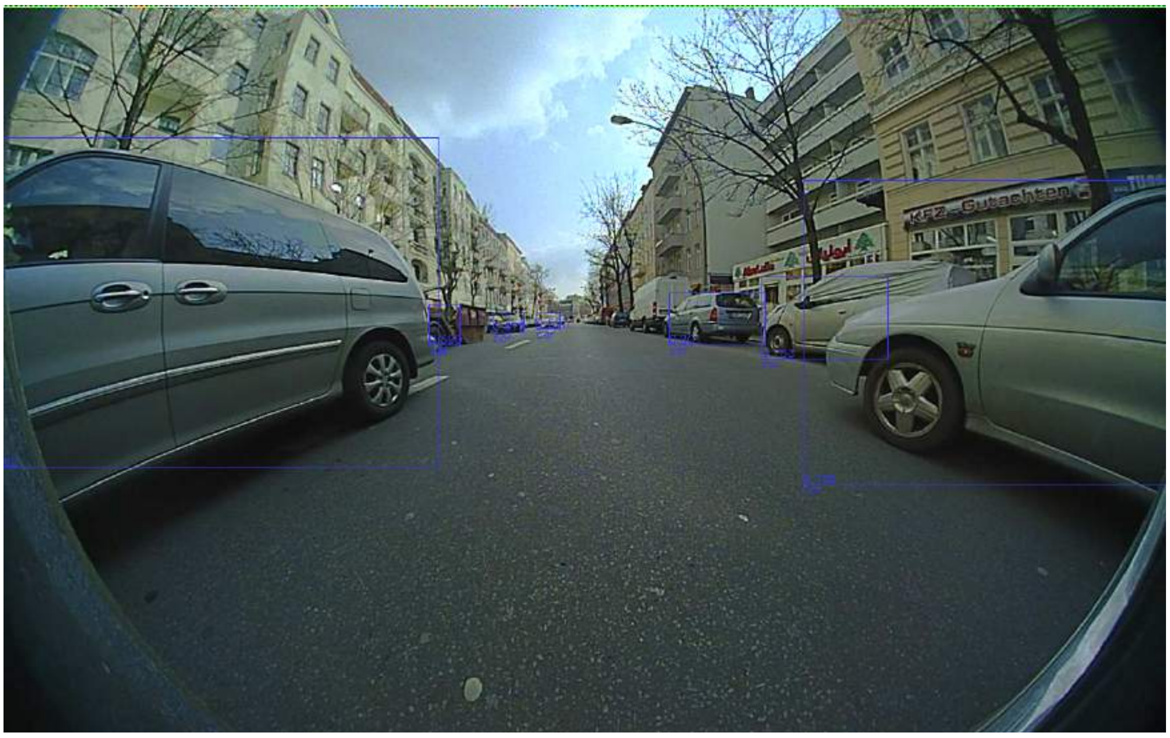


(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.17 Example Image from *IAV_Eval_Set*.

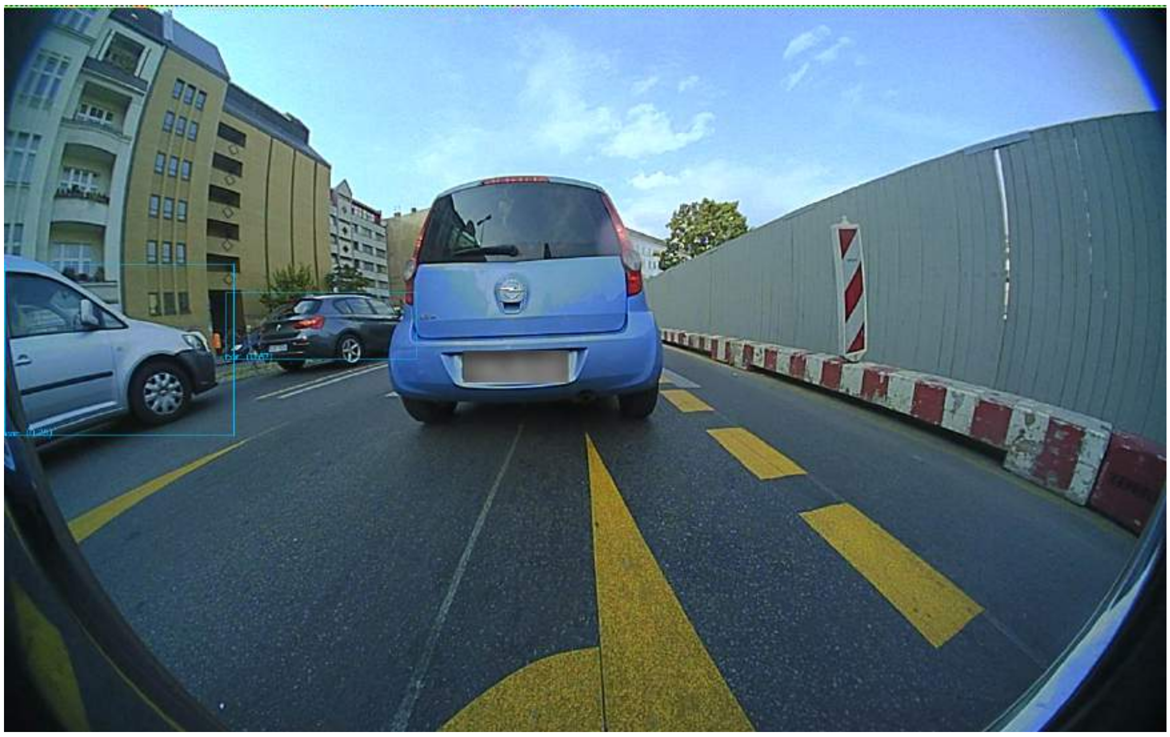


(a) squeezeDetPlus-KITTI-O

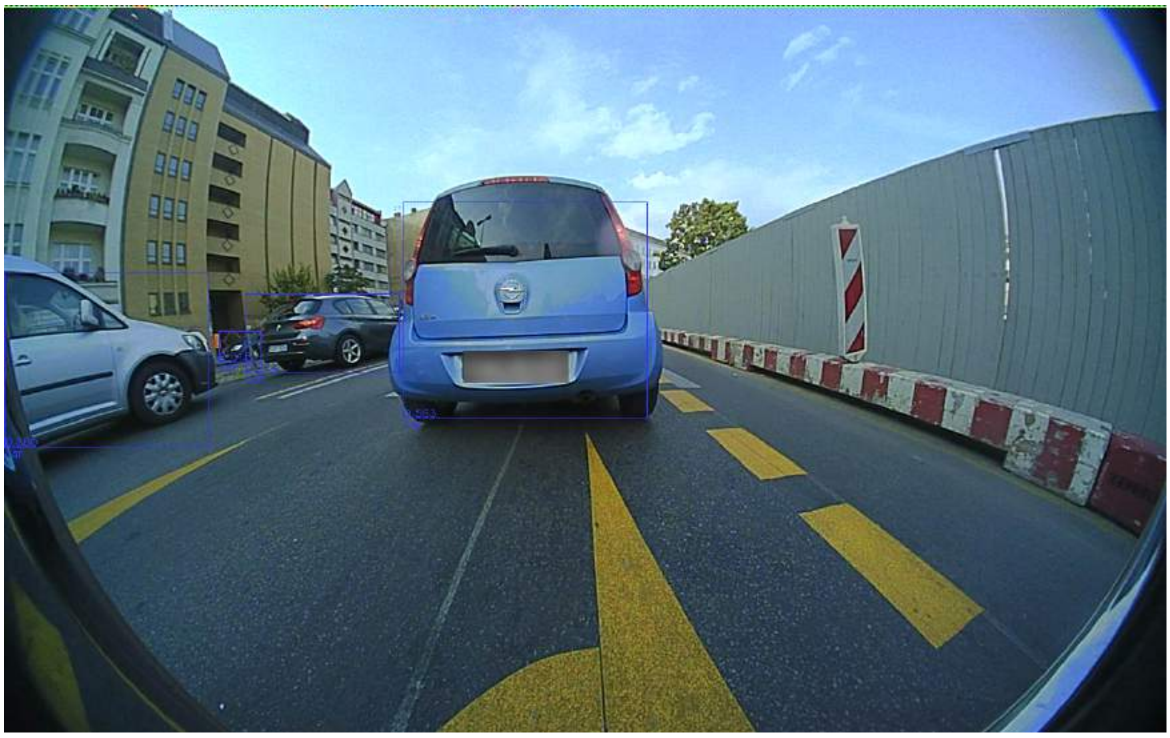


(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.18 Example Image from *IAV_Eval_Set*.

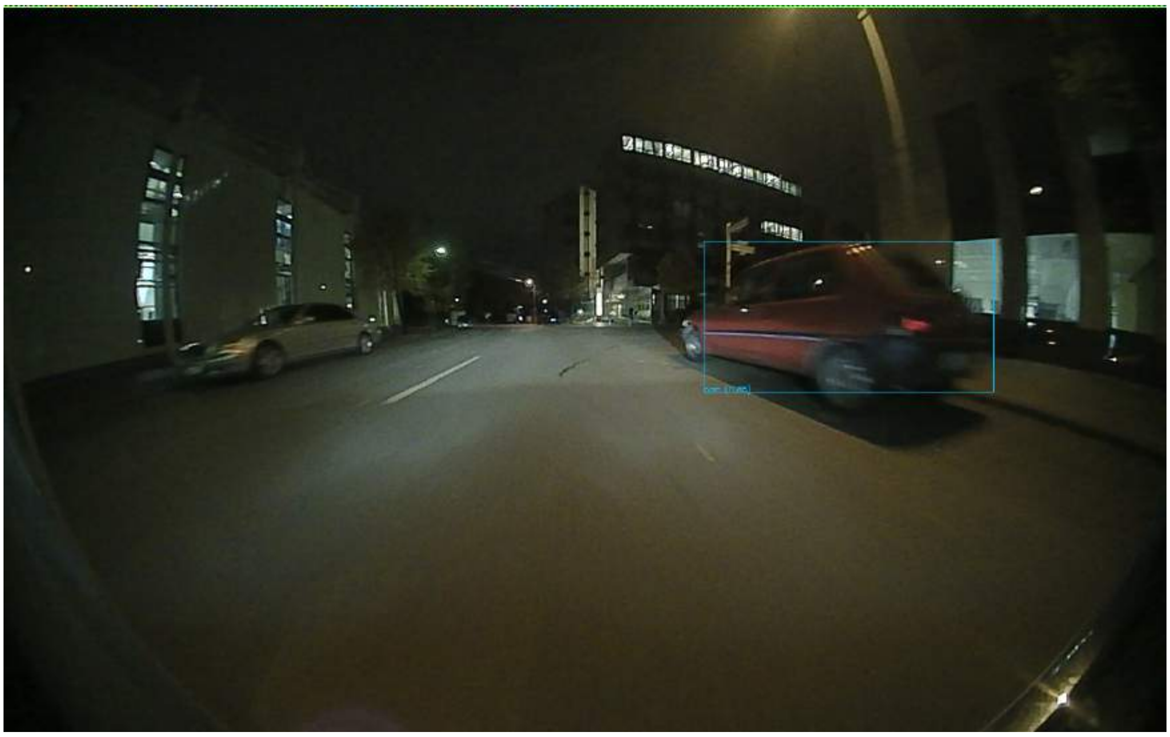


(a) squeezeDetPlus-KITTI-O

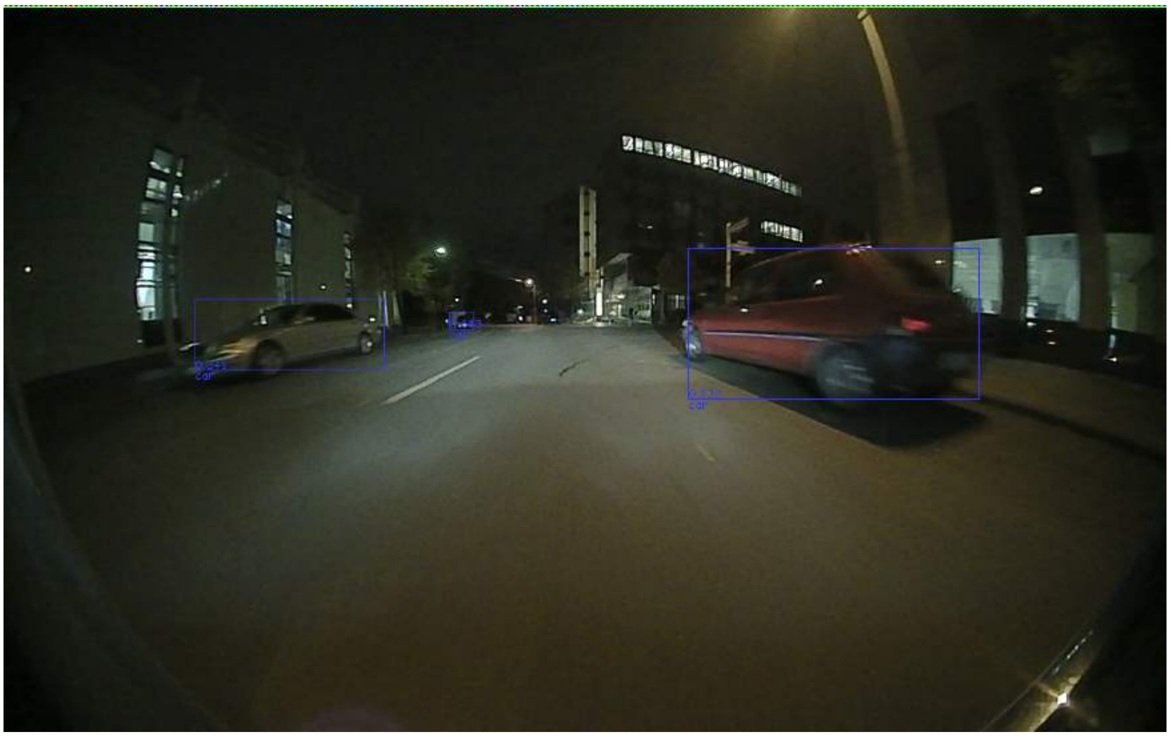


(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.19 Example Image from *IAV_Eval_Set*.

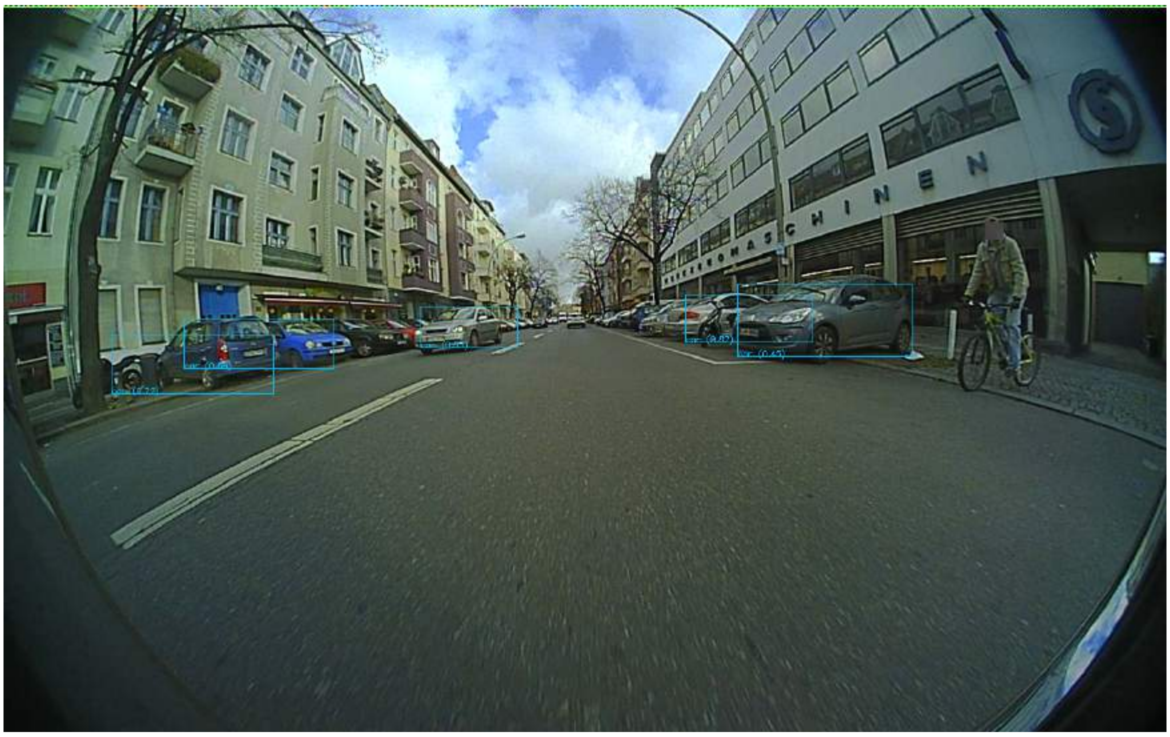


(a) squeezeDetPlus-KITTI-O

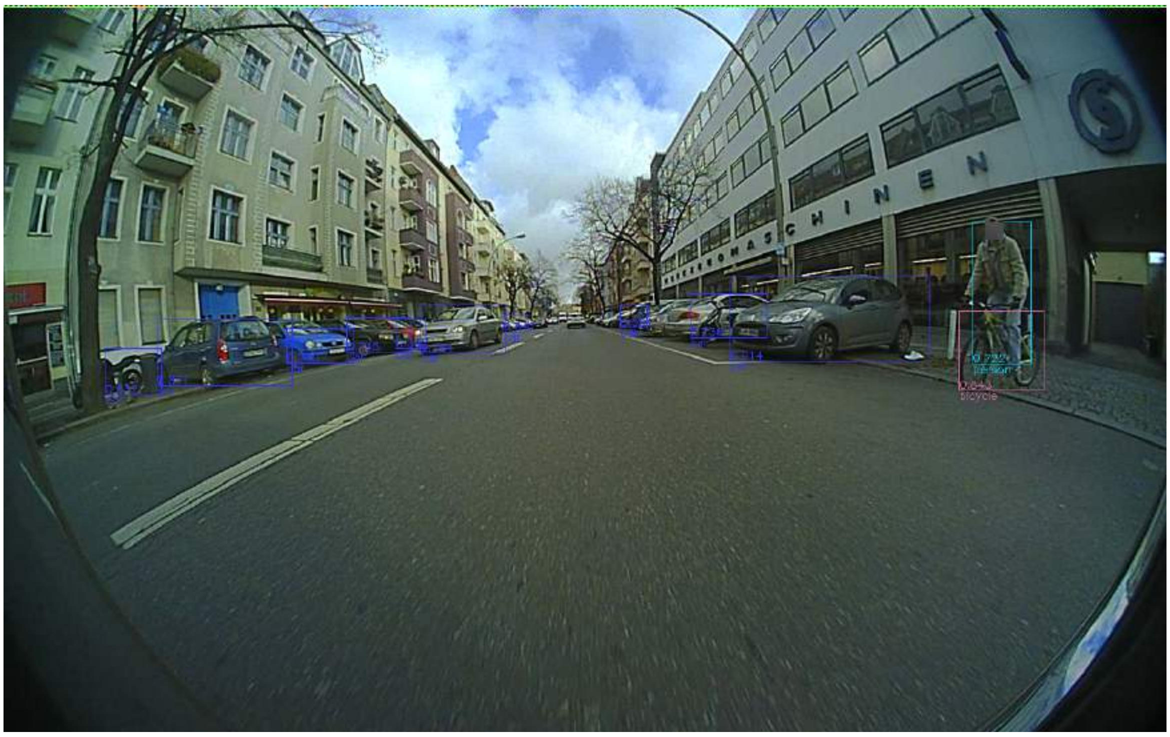


(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.20 Example Image from *IAV_Eval_Set*.



(a) squeezeDetPlus-KITTI-O



(b) squeezeDetPlus-BDD-KITTI-M-F

Fig. 7.21 Example Image from *IAV_Eval_Set*.

Chapter 8

Conclusion and Future Work

In this chapter, we first summarize the proposed solution and contribution of this work. Then we recall the results and recap what potential improvements of the current research shall be formulated as future works.

8.1 General Conclusion

Indubitably we could envisage cars driving fully automatically in the near future, but clearly there are many obstacles to overcome. This demands constant forward progress in both hardwares and softwares contributing in autonomous driving systems. In this thesis we attempted to break through one of these numerous hindrances. Similar to any intelligent moving object, an autonomous vehicle also needs to perceive and sense its surroundings. Cameras with various sorts of lens are the most favored sensor to give an autonomous vehicle the capability of seeing. Among all kinds of camera, fisheye camera offers the widest FOV, for this reason it's commonly used in automobile industry. Fisheye camera has one big disadvantage, distortion! The barrel distortion caused by fisheye lenses makes the object detection difficult, specially in close range. This is the very inviting challenge that we've worked on in this thesis.

We integrated an additional pre-processing step, where the training dataset had undergone a sort of transformation that simulates the fisheye effect, before entering our CNN (squeezeDetPlus). As the results presented in the chapter 7 proved, that fisheye augmentation significantly improved the performance of our target detector, so that it could compete with extraordinary detectors like Yolov3. In addition to the fisheye augmentation, the combination of KITTI and BDD plus regeneration of annotations by MASK-RCNN have made a big contribution to the success of our optimization. The big part of the improvement is due to the adequate preparation of input data for training. Based on the outcome of this thesis we can assuredly draw this conclusion, that at the end of the day the data that we feed into the network plays a determining role. The training data should resemble as much as possible the data on which a detector will be applied, as we also did with our fisheye augmentation.

8.2 Future Works

The promising results of this work had shown that the fisheye augmentation can considerably advance a CNN-based detector. Nevertheless, the real-world images captured by a fisheye camera installed on a vehicle deviate intensively from the existing images in the public datasets, due to the technical differences in cameras (generally camera matrix). This brings limitations, since technically it's impossible to change an image captured by a camera in a way that it absolutely resembles images of another camera. Put otherwise, pinhole cameras record data from a narrower angel than fisheye camera, as a result it's impossible to spread out such data on a wider fisheye-shaped surface without a considerable rate of error. We could put all this aside, if we had a fisheye image dataset in the first place. As an alternative we can luckily deploy a well-trained CNN like our squeezeDetPlus-BDD-MASK-F as a pre-labeler to generate such fisheye dataset in an inexpensive way. On the other hand, the object detection framework created in this thesis can serve us well to apply our fisheye augmentation on other potential CNNs in the future.

References

[1] Cornelisse, D. (2018). An intuitive guide to Convolutional Neural Networks. https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/.

[2] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941.

[3] Deng, L., Yang, M., Qian, Y., Wang, C., and Wang, B. (2017). Cnn based semantic segmentation for urban traffic scenes using fisheye camera. In *Intelligent Vehicles Symposium (IV), 2017 IEEE*, pages 231–236. IEEE.

[4] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

[5] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010). The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338.

[6] Felzenszwalb, P. F. and Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. *International journal of computer vision*, 59(2):167–181.

[7] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237.

[8] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

[9] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

[10] He, K., Gkioxari, G., Dollár, P., and Girshick, R. B. (2017). Mask R-CNN. *CoRR*, abs/1703.06870.

[11] He, K., Zhang, X., Ren, S., and Sun, J. (2014). Spatial pyramid pooling in deep convolutional networks for visual recognition. In *European conference on computer vision*, pages 346–361. Springer.

[12] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[13] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

[14] Imgaug (2017). Python image augmentation library. https://github.com/aleju/imgaug.

[15] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

[16] Jonathan Hui (2018). Real-time object detection with yolo, yolov2 and now yolov3. https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088.

[17] Kathuria, A. (2018a). Intro to optimization in deep learning: Gradient Descent. https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/.

[18] Kathuria, A. (2018b). What's new in YOLO v3? https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

[19] Lars Hulstaert (2018). Going deep into object detection. https://towardsdatascience.com/going-deep-into-object-detection-bed442d92b34.

[20] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[21] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer.

[22] OpenCV (2018). Camera Calibration and 3D Reconstruction. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[23] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

[24] Raaijmakers, M. (2017). Towards environment perception for highly automated driving: with a case study on roundabouts.

[25] Rawashdeh, N. A., Rawashdeh, O. A., and Sababha, B. H. (2017). Vision-based sensing of uav attitude and altitude from downward in-flight images. *Journal of Vibration and Control*, 23(5):827–841.

[26] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788.

[27] Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99.

[28] Rojas, R. (2009). Neural networks: a systematic introduction.

[29] Ruder, S. (2016). An overview of gradient descent optimization algorithms . http://ruder.io/optimizing-gradient-descent/index.html#batchnormalization.

[30] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

[31] SAE Internationl (2014). Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems. https://www.sae.org/standards/content/j3016_201401/.

[32] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[33] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

[34] Stanford, U. (2015a). CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/convolutional-networks/.

[35] Stanford, U. (2015b). CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/neural-networks-1/.

[36] Stanford, U. (2015c). CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/optimization-1/.

[37] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[38] Tensorflow (2018). Tensorflow Estimator framework. https://www.tensorflow.org/guide/premade_estimators.

[39] Uijlings, J. R., Van De Sande, K. E., Gevers, T., and Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, 104(2):154–171.

[40] Van Den Heuvel, F. A., Verwaal, R., and Beers, B. (2006). Calibration of fisheye camera systems and the reduction of chromatic aberration. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36(5):6.

[41] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612.

[42] Weng, L. (2017). Object Detection for Dummies Part 3: R-CNN Family. https://lilianweng.github.io/lil-log/2017/12/31/object-recognition-for-dummies-part-3.html#loss-function-1.

[43] Wu, B., Iandola, F. N., Jin, P. H., and Keutzer, K. (2017). Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *CVPR Workshops*, pages 446–454.

[44] Yu, F., Xian, W., Chen, Y., Liu, F., Liao, M., Madhavan, V., and Darrell, T. (2018). Bdd100k: A diverse driving video database with scalable annotation tooling. *arXiv preprint arXiv:1805.04687*.

[45] Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

[46] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.