CNN Based On-Road Object Detection, Ranging and Tracking for Autonomous Driving

A Project Report

submitted by

GAYATHRI S

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **CNN Based On-Road Object Detection, Ranging and Tracking for Autonomous Driving**, submitted by **Gayathri S**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. K. Sridharan
Research Guide,
Professor,
Dept. of Electrical Engineering,
IIT Madras, 600 036.

Place: Chennai

Date : June 7, 2017

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my project guide, Dr. K. Sridharan, for his constant support throughout the course of this project. I am deeply grateful to him for providing me with the opportunity to work in one of the hottest areas of research at present. I would like to convey my special thanks to Dr. Sudha Natarajan for educating me in the field of deep learning and for her sincere guidance throughout the project. Without her valuable assistance, I wouldn't have been able to complete this work.

I am forever grateful to my parents for their continuous support and care which helped me to reach where I am today. I wish to express my heartfelt appreciation to all the people who have been part of my life here at IIT Madras, especially to Aswin for being a pillar of strength to me.

ABSTRACT

KEYWORDS: Autonomous Driving, Convolutional Neural Networks, Faster R-CNN, Object Detection, Range Estimation, Single Shot Multibox Detector

Convolutional deep neural networks are widely used in designing visual perception systems for autonomous driving. The objective of this project was to develop an on-road object detection, ranging and tracking module for an autonomous vehicle, making use of deep learning architecture. A Convolutional Neural Network based method was chosen for the detection of objects from monocular images or video frames of the scene in front of the self-driving vehicle captured using a color camera. A Kalman filter based solution was developed for tracking of the detected objects. Existing object detection systems like Faster R-CNN, SqueezeDet and Single Shot MultiBox Detector were tested and analysed to select the one which would give the best performance, in terms of speed and accuracy. The Single Shot MultiBox Detector or SSD proved to be more space efficient and faster than the other detection methods, without compromising on the accuracy. In this work, we propose an extension to the SSD network to incorporate a range estimation feature. A proximity prediction layer added to the SSD architecture provides an estimate of the range of detected objects from the ego-vehicle in meters without using any additional sensors or systems like RADAR or LiDAR.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
ML	Machine Learning
R-CNN	Region-Based Convolutional Neural Networks
SSD	Single Shot MultiBox Detector

CHAPTER 1

Introduction

A vehicle which relies only on automation, without any human input is referred to as autonomous or self-driving. Vehicular automation is realized using concepts of artificial intelligence and mechatronics. By reducing human error while driving, automated vehicles can lower the risk of accidents. Autonomous vehicles help in improvement of traffic coordination, thereby decreasing pollution and time consumption. Another advantage of autonomous vehicles is that they can aid in providing mobility to the elderly and disabled. Many companies around the world are working on developing their own self-driving cars, some of the big players being Google, Uber and Tesla Motors

Accuracy is of top priority when it comes to driverless cars. A feasible solution for autonomous driving has to be very fast and cost efficient, in addition to being highly accurate. Over the past few years, the area of autonomous driving has been receiving much attention and is fast evolving. Still, the technology is far from being mature. The automated vehicles which are legally allowed on public roads are not fully autonomous yet. Hence, self-driving cars continue to be a focus of much research.

Obstacle avoidance is a prime feature in autonomous vehicles. The proper detection of on-road objects like other vehicles, cyclists, pedestrians or even animals and estimating their distance from the ego-vehicle is crucial for obstacle avoidance. This is a very demanding task as an error in detection or missing the detection of obstacles, primarily the less impact-resistant obstacles like pedestrians or cyclists, can lead to disastrous consequences. An object tracking system can help in cases of missing detections upto some extent.

1.1 Autonomous Navigation

The key components of autonomous driving are: sensing the road, mapping the road so as to get information about the location, and understanding traffic rules and learning to merge with traffic.[5] The state-of-the-art Advance Driver Assistant Systems (ADAS), which help to automate vehicles, make use of computer vision and image processing techniques for object detection, tracking and scene understanding. The sensing elements used by driverless cars include cameras, RADAR, LiDAR, motion sensors and GPS modules. Advanced control systems in autonomous cars interpret the sensory information and plan a path to the destination accordingly.

Visual perception module is an essential part of an autonomous driving system. It helps in scene understanding, which is crucial for path planning. The functions of a visual perception system include:

- Visual odometry
- Optical flow estimation
- Drivable region extraction
- Lane detection
- Object detection
- Object Tracking
- Differentiating background from important regions and objects

Scene understanding is a very challenging task and in Indian road conditions it is all the more so. Indian roads being crowded and unpredictable, tracking the motion of objects is an extremely difficult job in this scenario. An efficient autonomous navigation system should be able to detect the on-road objects with high precision along with predicting the probable movement of the objects, especially the nearer ones, in the next instant.

1.2 Object Detection

Object classification is the task of assigning a label from a fixed set of object categories to an input image. There are several machine learning techniques for image classification problem. When there are multiple objects in the input image, in addition to classification, localization of the objects also needs to be performed. The task of finding the location of the objects in an image as well as categorizing them is called object detection.

In autonomous cars, an object detection system is required to detect obstacles in the path. In this work, we have concentrated on the detection of on-road objects like car, truck, pedestrian and cyclists. A convolutional Neural Network is best suited for visual recognition tasks as its architecture is inspired from the organization of animal visual cortex. Hence, a Convolutional Neural Network based method was chosen for the object detection and ranging purpose.

1.3 Organization of the Thesis

This thesis is based on the development of a CNN based visual perception system for autonomous driving. The function of this system is the detection, range estimation and tracking of on-road objects. The thesis is organized as follows:

Chapter 2 gives the necessary background knowledge in the context of this work. A basic introduction to machine learning and deep learning is provided in this chapter. Convolutional Neural Networks and caffe deep learning framework are also introduced.

Chapter 3 includes a survey of popular CNN architectures for object detection like faster R-CNN. A Kalman filter based object tracking module is also presented.

Chapter 4 presents the Single Shot MultiBox Detector for object detection and the proposed modification to it to include proximity estimation feature.

Chapter 5 contains details about the experiments done during the course of this project and the results obtained.

Chapter 6 includes some concluding remarks along with directions for future work.

CHAPTER 2

Background

In this chapter we seek to build the basic background required for the understanding of this work.

2.1 Machine learning for computer vision

Computer vision is an interdisciplinary field which deals with designing systems which can extract information from digital images or videos. The useful information from images can be used to build very efficient automation tools. Computer vision focuses on methods for acquiring, processing, analysing and understanding digital images and to give information which helps the system to take decisions. It is applied in a wide variety of fields like biomedical imaging to detect the anomalies in internal organs , text recognition, self driving cars to detect and track the objects in front of it etc. The aim of this area of research is to get efficiency as much as human vision systems to understand an image or a video and make decisions which result in autonomous systems.

The research directions in this area can be mainly captured under the following subdomains:

- Scene reconstruction
- Event detection
- Video tracking
- Object recognition
- Object pose estimation
- Motion estimation
- Image restoration

In this report we will be focusing on object detection and tracking .This is an essential application in the case of driverless cars. We will use this technique for building an application which will help driverless cars to navigate. In this section, we will discuss some basic concepts in machine learning and will also see how methods in machine learning can be used to solve computer vision problems.

Tom Mitchell, in his well-known book "Machine learning"[12], defines ML as "improving performance in some task with experience". He futher defined a well posed learning problem in the following manner: "A computer program is said to learn from experience E with respect to some task T and some perfomance measure P ,if its performance on T as measured by P improves with experience E."

Machine learning tasks are typically classified into three broad categories, depending on the nature of the learning "signal" or "feedback" available to a learning system. These are:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

2.1.1 Supervised learning

In this method we will have a training set which will contain pairs of input and the desired output which is also called label. The algorithm will learn the relation between the training inputs and outputs and infer the function which maps the input to output.

Let us imagine that we want to teach a computer to distinguish pictures of cats and dogs. We can collect pictures of cats and dogs adding a tag 'cat' or 'dog'. Labelling is usually done by human annotators to ensure high quality. So now we know the true labels of the pictures and can use this data to "supervise" our algorithm in learning the right way to classify images. Once our algorithm learns how to classify images we can use it on new data and predict labels ('cat' or 'dog' in our case) on previously unseen

images.

In order to solve a given problem of supervised learning, one has to perform the following steps:

- Determine the type of training examples. Before doing anything else, the user should decide what kind of data is to be used as a training set. In our case it is image data.
- Gather a training set. A set of input objects is gathered and corresponding outputs are also gathered, either from human experts or from measurements.
- Determine the input feature representation of the learned function. The accuracy of the learned function depends strongly on how the input object is represented.
- Determine the structure of the learned function and corresponding learning algorithm.
- Complete the design. Run the learning algorithm on the gathered training set. Some supervised learning algorithms require the user to determine certain control parameters. These parameters may be adjusted by optimizing performance on a subset (called a validation set) of the training set, or via cross-validation.
- Evaluate the accuracy of the learned function. After parameter adjustment and learning, the performance of the resulting function should be measured on a test set that is separate from the training set.

A wide range of supervised learning algorithms are available, each with its strengths and weaknesses. There is no single learning algorithm that works best on all supervised learning problems.

In the context of our work, one of the main computer vision problems we encouter is how to classify the objects in an image. In order to solve this problem we implemented state of the art supervised machine learning techniques. The technique we have used here is based on neural networks.

The figure 2.1 shows an exemplary classification task for samples with two random variables. The training data (with class labels) are shown in the scatter plots. The red-dotted lines symbolize linear (left) or quadratic (right) decision boundaries that are used to define the decision regions R1 and R2. New observations will be assigned the class labels "w1" or "w2" depending on in which decision region they will fall into. We can already assume that our classification of unseen instances won't be "perfect" and some percentage of samples will be mis-classified.

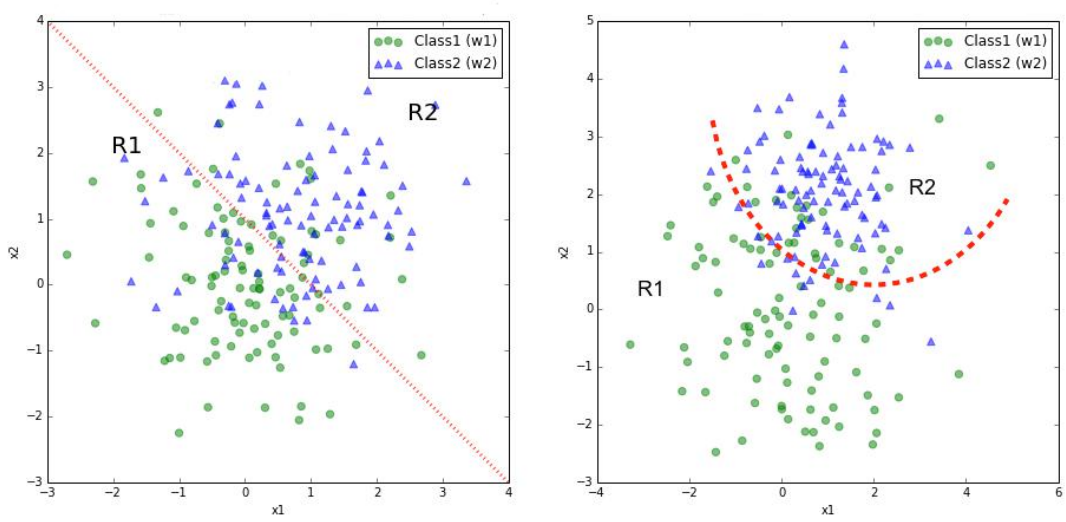


Figure 2.1: Example for classification problem

2.1.2 Unsupervised learning

This technique is used when the labels for the training data are not known. It is called unsupervised as it is left on the learning algorithm to figure out patterns in the data provided. The goal is to find some type of structure in the data without knowing the labels.

Clustering is an example of unsupervised learning in which different datasets are clustered into groups of closely related items. Some examples for the application of clustering algorithms are :

- Given a set of news reports, cluster related news items together.
  (Used by news.google.com)
- Given a set of users and movie preferences, cluster users who have similar taste.

The figure 2.2 shows an example of clustering. We can conclude that there are three different clusters.

2.1.3 Reinforcement learning

Reinforcement learning is the problem of getting an agent to act in the ideal manner in some specific context, without explicitly mentioning what the ideal action is. It allows the machines to learn what the proper behaviour should be so as to maximize its rewards. For example, consider teaching a dog a new trick: you cannot tell it what

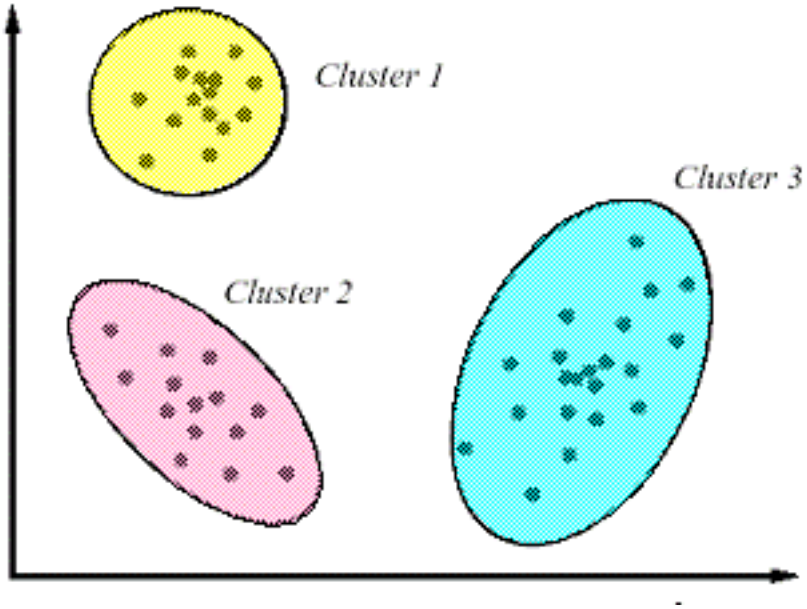


Figure 2.2: Example for unsupervised learning problem: Clustering

to do, but you can reward/punish it if it does the right/wrong thing. It has to figure out what it did that made it get the reward/punishment, which is known as the credit assignment problem. We can use a similar method to train computers to do many tasks, such as playing backgammon or chess, scheduling jobs, and controlling robot limbs.

2.2 Artificial Neural networks

Artificial Neural networks (ANNs) are inspired from biological neural networks present in central nervous system of animals, and are used to approximate or generalize a solution to a given problem statement after proper training of the network. A neural network is an interconnected group of nodes called neurons, which mimics the way in which biological neurons are connected by axons. Typical neural networks consists of several nodes and the signal traverses from input node to output node through the hidden nodes. The process of resetting the weights on these nodes for a specific purpose using back-propagation algorithm is referred to as training the neural network.

A neuron is the basic computational building block of a biological brain. The major parts to be considered in a neuron or nerve-cell are the dendrites, cell body, axons, and synapses. The synapses of one neuron is connected to dendrites of other neurons and axons act as the transferring paths for the signal. Approximately 86 billion neurons can

be found in the human nervous system and they are connected with approximately 10^{14} to 10^{15} synapses.

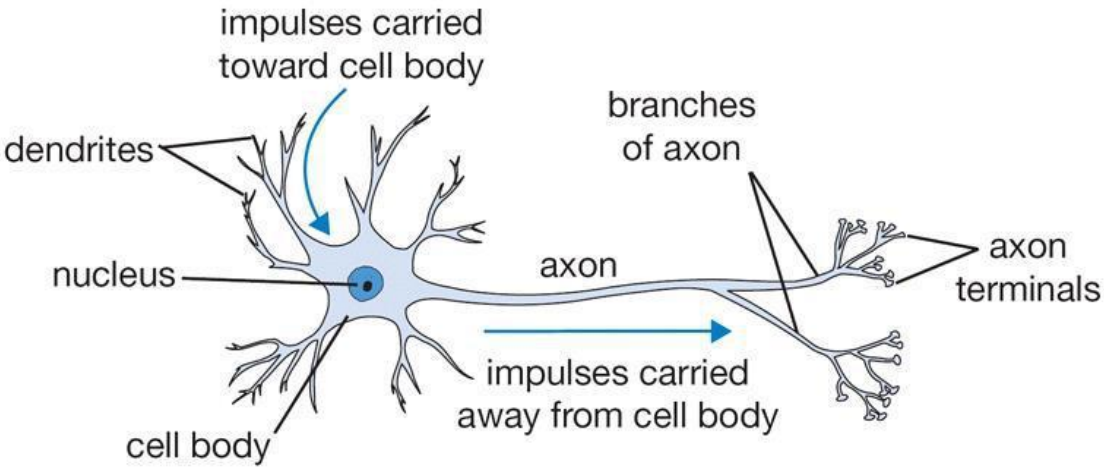


Figure 2.3: The structure of a typical neuron

An artificial neuron, on the other hand, is the basic computational unit of an artificial neural network. It attempts to model the functioning of a biological neural network mathematically. There can be multiple inputs (x_i) to an artificial neuron and the individual inputs can have different weights(w_i). The computational part of the neuron is composed of a summation function, which computes the weighted sum of the inputs with or without a bias. An activation function(f) at the output of the neuron determines whether the neuron sends information to the subsequent neurons. The sigmoid function is commonly chosen as the activation function as it compresses the output of the neuron within the range 0-1. To draw an analogy between artificial neurons and biological

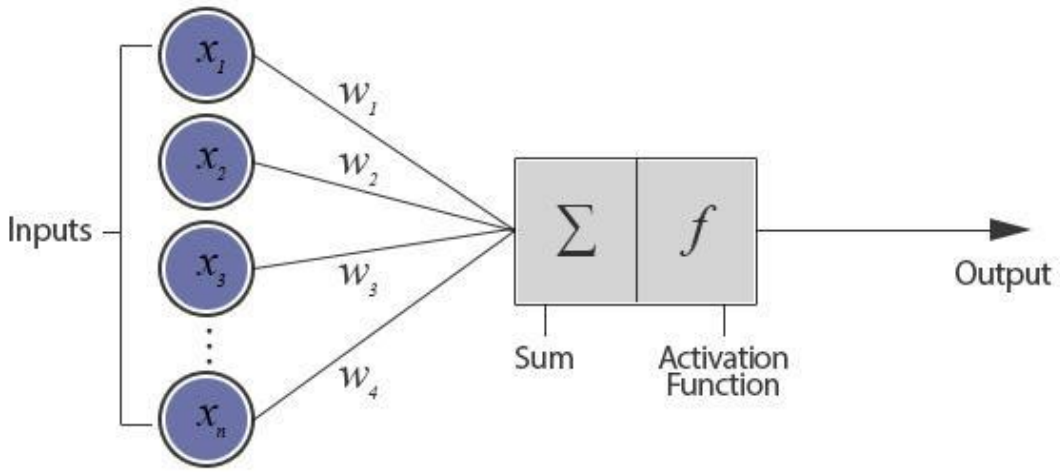


Figure 2.4: The structure of an artificial neuron

neurons, input signals are received from its dendrites and output signals are produced along its (single) axon. The transfer function is similar to the cell body and the activation function is analogous to "firing" of a neuron. The idea is that synaptic strengths or weights can be learned so as to control the influence of one neuron on another, based on the inputs. The weights act as inhibitors and activators depending upon their signs. The weights are multiplied with the corresponding inputs and these multiplied scalars are summed over at the body. This sum is passed through the pre-decided activation function to release the output:

$$y = f(\sum w_i x_i + b) \tag{2.1}$$

where x_i are the inputs,
w_i are the weights,
b is the bias,
f is the activation function, and
y is the output of the neuron.

2.3 Deep Learning

We have seen some machine learning algorithms in the previous sections. The representation of the data given to the model has a huge effect on the performance of these machine learning algorithms. The figure 2.5 shows an example of how a linear classification algorithm might give different performance for different representations of the same data.

A feature is an individual measurable property of a phenomenon being observed. In the context of computer vision there are a large number of possible features, such as edges and objects. In many cases, the problem with different representations of data can be overcome by selecting a right set of features from the data for the particular task and passing this set to the machine learning algorithm. However, for some tasks, like the object detection task we have at hand for instance, it is impossible to know which features to select. Representation learning is one solution to this problem. Feature learning or representation learning is a set of techniques that learn a feature - a transformation of

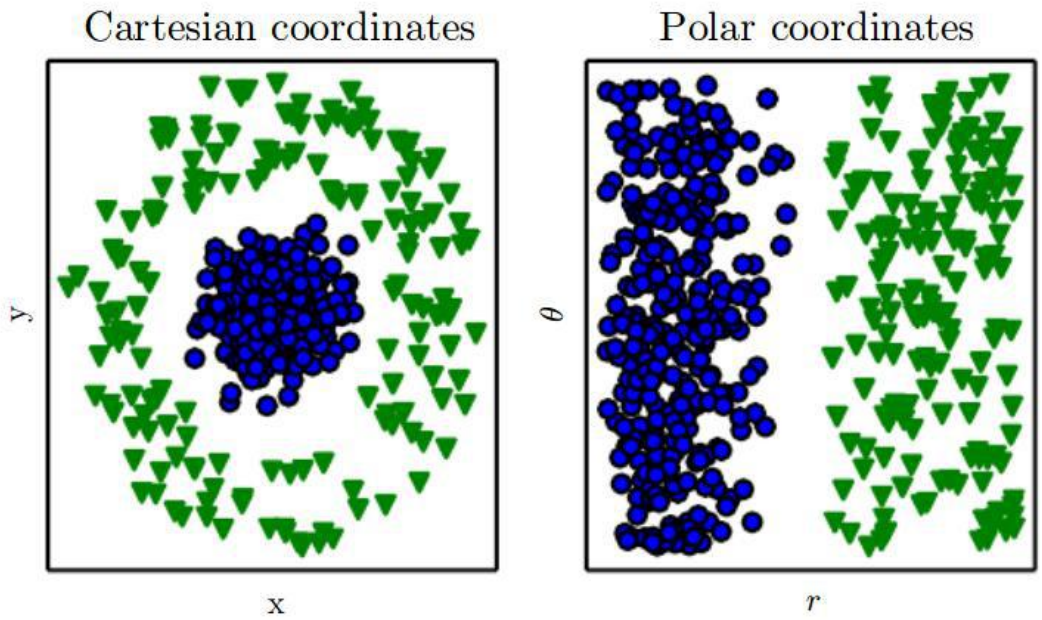


Figure 2.5: Example of effect of different representations: Classification of the data by drawing a line separating the two categories is an easy task when tha data is represented in polar coordinates, but impossible in cartesian coordinate representation.

raw data input to a representation that can be effectively exploited in machine learning tasks. Still, it can be very difficult to extract high-level, abstract features from raw data. Deep Learning approach to representation learning solves this issue.[4]

Deep learning networks are essentially artificial neural networks with more than one hidden layer which are capable of unsupervised learning of features from the input data and their transformation. Deep neural networks are a cascade of several layers of nonlinear processing units and each layer responds to a different level of abstraction. Several deep learning networks such as deep neural networks, convolutional deep neural networks, deep belief networks and recurrent neural networks have found applications in various fields. For our work, we are using convolutional neural networks.

2.4 Convolutional Neural Networks

A convolutional neural network is a type of feed forward artificial neural network made up of neurons that have learnable weights and biases[8]. A CNN usually takes an order 3 tensor as its input, e.g., an image with H rows, W columns, and 3 channels (R, G, B color

channels). Higher order tensor inputs can also be handled by CNN in a similar fashion. The input then sequentially goes through a series of processing. Each processing step is called a layer. The different types of layers in a CNN are convolution layer, pooling layer, normalization layer, fully connected layer, loss layer, etc.

$$x_1 \Rightarrow \boxed{w_1} \Rightarrow x_2 \Rightarrow \boxed{w_2} \Rightarrow ... \Rightarrow x_{L1} \Rightarrow \boxed{w_{L1}} \Rightarrow z \tag{2.2}$$

The equation 2.2 illustrates how a CNN runs layer by layer in a forward pass. The input is x_1 , usually an image (order 3 tensor). It goes through the processing in the first layer, which is the first box. We denote the parameters involved in the first layer's processing collectively as a tensor w_1 . The output of the first layer is x_2 , which also acts as the input to the second layer processing. This processing proceeds till the last layer in the CNN, which outputs x_L . One additional layer, however, is added for backward error propagation, a method that learns good parameter values in the CNN.

2.4.1 Convolutional Layer

The conv layer or convolutional layer is the core building block of a Convolutional Network. It does most of the computation. The conv layer's parameters consist of a set of learnable filters or kernels. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. During the forward pass, we slide each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at the same position as that of the kernel (called the receptive field). As we convolve the filter over the width and height of the input volume we will produce a 2-dimensional activation map or feature map that gives the responses of that filter at every spatial position. An illustration of the convolution process can be seen at figure 2.6.

A benefit of the convolution layer is that since all spatial locations share the same convolution kernel, the number of parameters needed for a convolution layer is greatly reduced. In a deep neural network setup, convolution encourages parameter sharing.

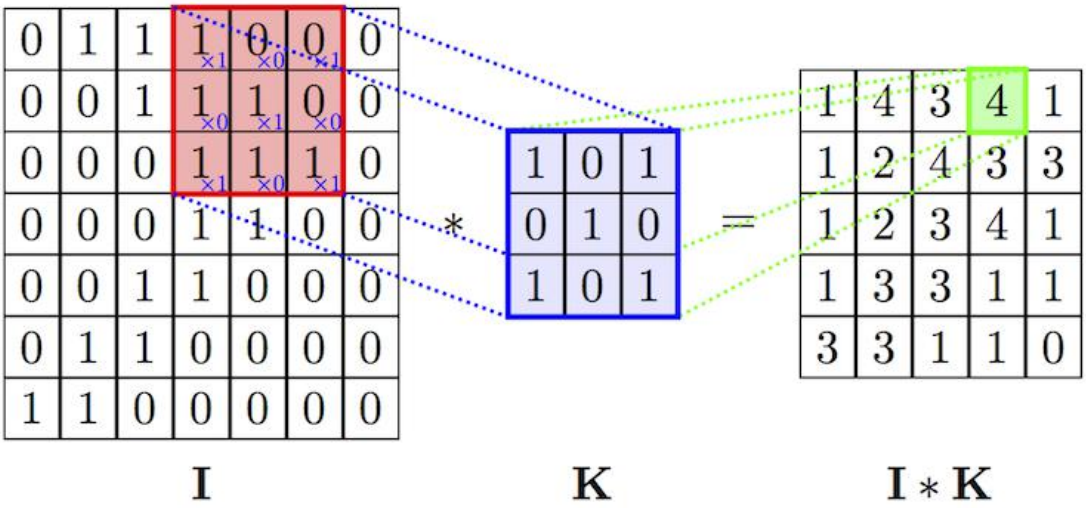


Figure 2.6: Example of convolution on a 2-D input image I, with a kernel K of size 3x3

Two other important parameters of a convolution layer are stride and padding. Stride controls how the filter convolves around the input volume. In the figure 2.6, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction. If the stride $s > 1$, then every movement of the kernel skips s-1 pixel locations (i.e., the convolution is performed once every s pixels both horizontally and vertically). Sometimes we need the output of the convolution layer to be of a desired size larger than the normal convolution outputs. In that case, we can 'pad' rows above and below the input, and columns to the left an right of it. The number of padded rows and columns is chosen according to the desired output size. Elements of the padded rows and columns are usually set to 0, but other values are also possible.

In a convolution layer, multiple convolution kernels are usually used. Assume D kernels are used and each kernel is of spatial span HxW. We denote all the kernels as f. Hence, f is an order 4 tensor in $\mathbb{R}^{HxWxD^lxD}$, where D^l is the number of channels in the input image. Suppose we are considering the l^{th} layer, whose inputs form an order 3 tensor x^l with $x_l \in \mathbb{R}^{H^lxW^lxD^l}$. Thus, we need a triplet index set (i^l, j^l, d^l) to locate any specific element in x^l. The triplet (i^l, j^l, d^l) refers to one element in x^l, which is in the $d^{l^{th}}$ channel, and at spatial location (i^l, j^l) i.e., at the $i^{l^{th}}$ row, and $j^{l^{th}}$ column.The simple convolution operation with stride =1 and no padding can be mathematically

expressed as:

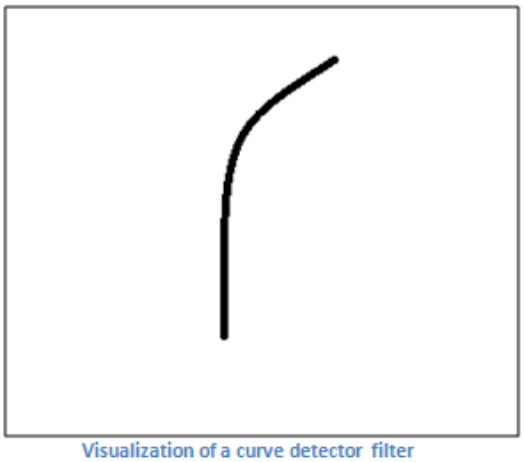
$$y_{i^{l+1},j^{l+1},d} = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d^l=0}^{D^l} f_{i,j,d^l,d} \times x^l_{i^{l+1}+i,j^{l+1}+j,d^l} \tag{2.3}$$

Hence, we have y(or x^{l+1}) in $\mathbb{R}^{H^{l+1} \times W^{l+1} \times D^{l+1}}$, with $H^{l+1} = H^l - H + 1$, $W^{l+1} = W^l - W + 1$, and $D^{l+1} = D$.[16]

From a high level perspective, convolution filters can be thought of as feature identifiers. Each convolution layer responds to a particular feature like curve, edge, color etc. Figure 2.7 shows an oversimplified curve detector filter. As a curve detector, the filter will have a pixel structure in which there will be higher numerical values along the area that is a shape of a curve. Thus when the receptive field has a curve of this shape, the convolution output will be high.

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Figure 2.7: A curve detector filter

Each convolution layer in a network responds to different levels of abstraction. The initial layers detect simple features like curves, lines etc. while the succeeding layers can detect more complex shapes like wheel, a person's head etc. which are a combination of these simple features.

The equation 2.3 seems to be a pretty complex one to implement. Usually in CNN implementation, the convolution operation is simplified to a matrix multiplication

operation by transforming the input matrix and vectorizing the kernel.

2.4.2 Rectified Linear Units or ReLU Layer

It is a common convention to apply a non-linear layer immediately after each conv layer. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers. The ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume. Basically, this layer just changes all the negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer. In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster with ReLU, without making a significant difference to the accuracy.

2.4.3 Pooling Layer

The pooling operation requires no parameter. The spatial extent of the pooling (HxW) is specified in the design of the CNN structure. A pooling layer operates upon x^l (the input to the pooling layer) channel by channel independently. Within each channel, the matrix with $H^l \times W^l$ elements are divided into nonoverlapping subregions, each subregion being HxW in size. The pooling operator then maps a subregion into a single number.

Two types of pooling operators are widely used: max pooling and average pooling. In max pooling, the pooling operator maps a subregion to its maximum value, while the average pooling maps a subregion to its average value.

2.4.4 Fully-connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Usually in CNN architectures, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. A fully connected layer can be implemented using a convolutional layer setup. The only difference between FC and conv layers is that the

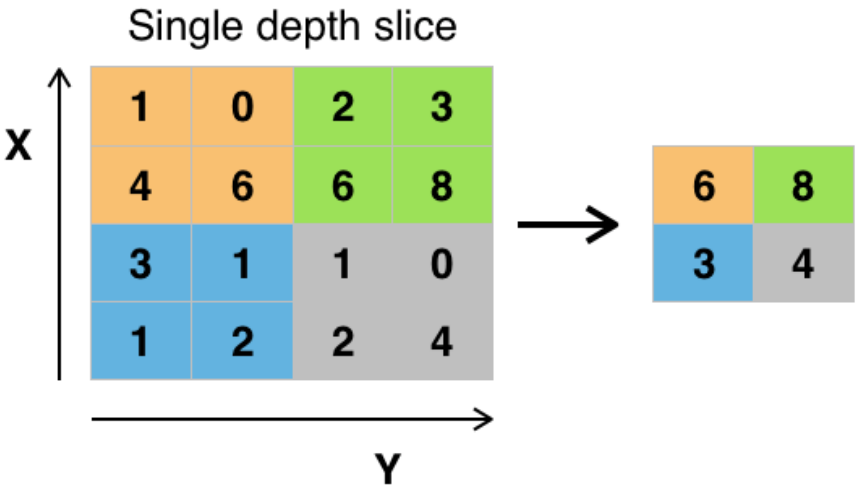


Figure 2.8: Example of max-pooling layer with kernel size 2x2 and stride = 2

neurons in the conv layer are connected only to a local region in the input. However, the neurons in both layers still compute dot products, so their functional form is identical.

2.4.5 Training a Convolutional Neural Network

Training a neural network is the process of tuning the weights and biases associated with the neurons for some particular function. In the case of convolutional neural networks, the function of neurons is done by convolutional layers. In case of CNN, training essentially means adjusting the weights of the kernels of the different layers to achieve some function. Training process is 'supervised' in a convolutional neural network. Along with the input data, the desired output for that data (ground truth) is also provided to the network during training phase. The weights are so adjusted that the error between the ground truth and network output is minimized.

Stochastic gradient descent (SGD)

The parameters of a CNN model are optimized to minimize the loss z, i.e., we want the prediction of a CNN model to match the ground-truth labels. Stochastic gradient descent (often shortened in SGD), also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. The training

process involves running the CNN network in both directions. We first run the network in the forward pass to get x^l (the output of the last layer in the network) to achieve a prediction using the current CNN parameters. The training network has an additional loss layer added at the end of the network. Instead of outputting the prediction, we continue running the forward pass till the final loss layer and compute the loss z. The loss z is then a supervision signal, guiding how the parameters of the model should be modified (updated). The SGD way of modifying the parameters is:

$$(w^i)^{t+1} = (w^i)^t - \eta \frac{\partial z}{\partial (w^i)^t} \tag{2.4}$$

where, w^i denotes the parameters of the i^{th} layer, z is the loss, η is the learning rate and t indicates the time index.

The partial derivative vector in the equation is called gradient in mathematical optimization. In a small local region around the current value of w^i , moving w^i in the direction determined by the gradient will increase the value z. Hence, in order to minimize the loss function, we should update w^i along the opposite direction of the gradient. This updating rule is called the gradient descent.

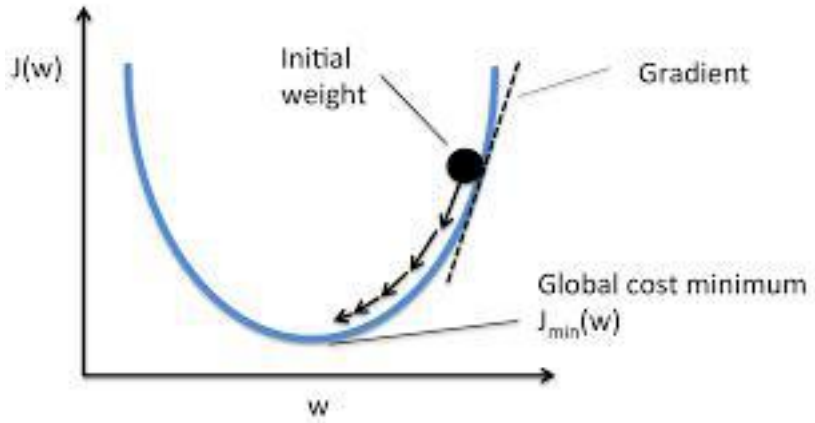


Figure 2.9: Illustration of Stochastic Gradient Descent

If we move too far in the negative gradient direction, however, the loss function may increase. Thus, in every update we change the parameters only by a small proportion of the negative gradient, controlled by η (the learning rate).

Backpropagation

In order to update the weights of each layer in the network by stochastic gradient descent, a method called error backpropagation is used. For every layer, we compute two sets of gradients: the partial derivatives of z with respect to the layer parameters w^i , and w.r.t. that layer's input x^i. As seen in equation 2.4, the term $\frac{\partial z}{\partial w^i}$ can be used to update the parameters of the i^{th} layer. The partial derivative $\frac{\partial z}{\partial x^i}$ can be used in updating the parameters backwards, i.e., to the $(i-1)^{th}$ layer. This is done using chain rule.

Let's take the i^{th} layer as an example. When we are updating the i^{th} layer, the back propagation process for the $(i+1)^{th}$ layer must have already been done. That is, we have computed the terms $\frac{\partial z}{\partial w^{i+1}}$ and $\frac{\partial z}{\partial x^{i+1}}$. Both are stored in memory and ready for use. Now our task is to compute $\frac{\partial z}{\partial w^i}$ and $\frac{\partial z}{\partial x^i}$. Using the chain rule, we have:

$$\frac{\partial z}{\partial w^i} = \frac{\partial z}{\partial x^{i+1}} \cdot \frac{\partial x^{i+1}}{\partial w^i} \tag{2.5}$$

$$\frac{\partial z}{\partial x^i} = \frac{\partial z}{\partial x^{i+1}} \cdot \frac{\partial x^{i+1}}{\partial x^i} \tag{2.6}$$

The partial derivatives $\frac{\partial x^{i+1}}{\partial w^i}$ and $\frac{\partial x^{i+1}}{\partial x^i}$ are easy to compute.

2.5 The Caffe Framework

Training and testing neural networks requires a stable and a reliable platform. Many libraries like Torch, Theano, TensorFlow, Caffe, Keras, Eblearn C++ are available for this purpose. Caffe is immensely popular among the researchers and has an easy user interface. The library was started as a PhD project by Yangqing Jia at UC Berkely and was further developed by Berkely Vision and Learning Center.[7] This library is open source and is licensed under BSD 2- Clause. Caffe is a deep learning framework which enables deep neural network architecture design, training and testing. The library is designed in C++ language, and the user interaction framework to invoke the functions is wrapped using python language.

The designing of network, setting up of training parameters, and classification network settings are done using google protocol buffer files. These files have an extension of ".prototxt". Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data. The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data.The desired structure of data is specified by defining protocol buffer messages in .proto files.

In caffe, the transfer and storage of data is done using a structure called blob. Blob is an n-dimensional array which acts as a wrapper around the actual data being processed. The conventional blob dimensions for batches of image data are number N x channel K x height H x width W, where number / N is the batch size of the data and channel / K is the feature dimension. The value at index (n, k, h, w) is physically located at index ((n * K + k) * H + h) * W + w.

A layer is the fundamental unit of computation in a caffe network. A layer takes input through bottom connections and makes output through top connections. Each layer type defines three critical computations: setup, forward, and backward.

- Setup: The initialization of the weights of the layer and setting up its connections are done in this step.
- Forward: The computations are done on the input data in the specified manner and sent to the output blob in this step.
- Backward: Given the gradient w.r.t. the top output, the gradient w.r.t. to the input is computed and sent backwards to the bottom in this step. A layer with parameters computes the gradient w.r.t. to its parameters and stores it internally.

Some examples of layers include convolutional layer, pooling layer, ReLU layer, normalization layer etc.

The collection of a set of layers for some particular function is called a net. A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

Data enters Caffe through data layers which lie at the bottom of nets. Data can come from efficient databases (LevelDB or LMDB), directly from memory, or, when efficiency is not critical, from files on disk in HDF5 or common image formats. Common input preprocessing like mean subtraction, scaling, random cropping, and mirroring can be done in data layers by specifying transformation parameters.

CHAPTER 3

CNN based Object Recognition and Tracking

As we have already seen, object detection is the process of locating objects of interest, such as car, bus, pedestrian etc. , in images or videos, using computer vision and image processing. The state-of-the-art object detection systems leverage the architecture of CNNs which is inspired from the organization of animal visual cortex. Several CNN based models are now available for various object detection applications.

3.1 Popular CNN Architectures

The first work that popularized Convolutional Networks in Computer Vision was the **AlexNet**, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton[9]. Earlier, it was common to have only a single conv layer always immediately followed by a pooling layer. But AlexNet is a deeper network with convolutional layers stacked on top of each other. It contains 8 layers - 5 convolutional layers and 3 fully connected layers. The architecture of AlexNet is given in figure 3.1. It can be seen in the figure that the architecture consists of two branches. The purpose of this branching is to split the training process to 2 GPUs. The size of AlexNet is 233MB. The AlexNet was submitted

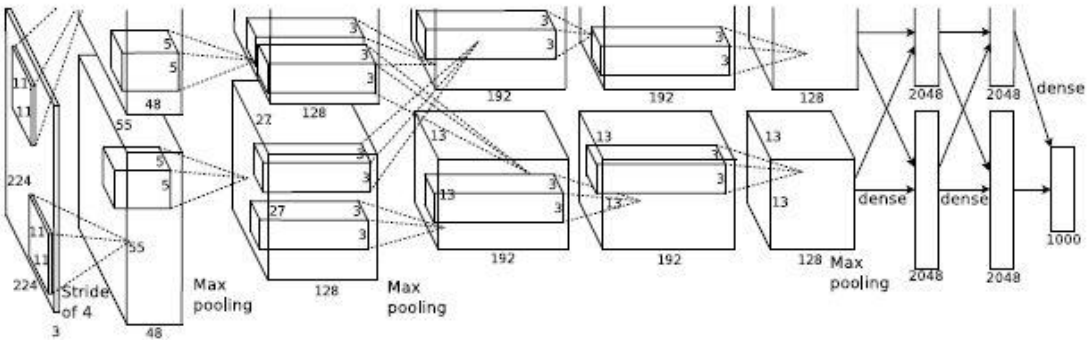


Figure 3.1: The AlexNet architecture

to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second

runner-up (top 5 error of 16% compared to runner-up with 26% error).

The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the **ZFNet**[17] . It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller. The size of ZFNet is 117MB.

GoogLeNet[15] is a network developed by Szegedy et al. from Google. GoogLeNet is a 22 layer CNN and was the winner of ILSVRC 2014 with a top 5 error rate of 6.7%. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). It uses average pooling instead of Fully Connected layers at the top, thus eliminating a large amount of parameters.

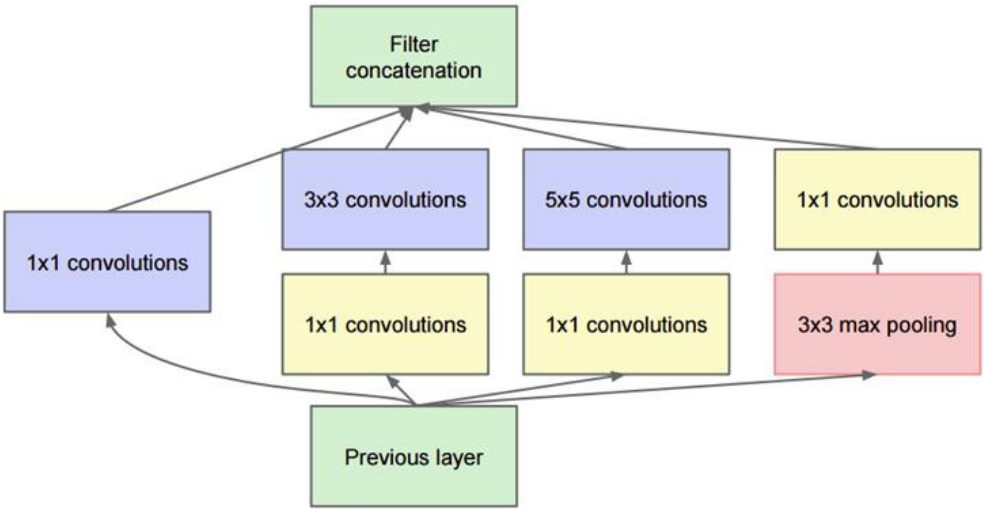


Figure 3.2: The inception module of GoogLeNet

The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the **VGGNet**[14]. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers, featuring an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. A downside of the VGGNet is that it is more expensive to evalu-

ate and uses a lot more memory (93MB) and parameters (140M).

Residual Network or **ResNet**[6] is a 152 layer network developed by Kaiming He et al. which was the winner of ILSVRC 2015 with an incredible error rate of 3.6%. It features special skip connections and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network.

3.2 Object Detection Using Faster R-CNN

Recent advances in object detection are driven by the success of region proposal methods and region-based convolutional neural networks (R-CNNs). **R- CNN** is a combination of convolutional neural networks and region proposals[3].

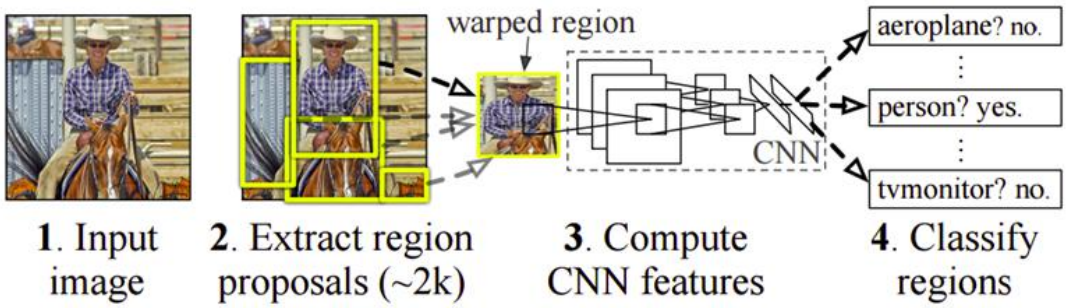


Figure 3.3: The flow of R-CNN

There are several region proposal methods that can be used in conjunction with CNNs such as selective search, objectness etc. Selective Search is used in particular for R-CNN. Selective Search performs the function of generating 2000 different regions that have the highest probability of containing an object. These region proposals are then "warped" into an image size that can be fed into a trained CNN (AlexNet in this case). that extracts a feature vector for each region.A 4096 dimensional feature vector is extracted from each region proposal by forward propagating a mean subtracted image through five convolutional layers and two fully connected layers. This vector is then used as the input to a set of linear SVMs that are trained for each class and output a classification. The vector also gets fed into a bounding box regressor to obtain the

most accurate coordinates. Non-maxima suppression is then used to suppress bounding boxes that have a significant overlap with each other.

R-CNN achieved 54.2% mean average precision (mAP) on PASCAL VOC 2007 dataset and 49.6% mean average precision (mAP) on PASCAL VOC 2012 dataset. But object detection is slow. At test-time, features are extracted from each object proposal in each test image. Detection with VGG16 takes 47s / image on a GPU.

Improvements had to be made to the original R-CNN model because of 3 main problems: training took multiple stages (ConvNets to SVMs to bounding box regressors), was computationally expensive, and was extremely slow. **Fast R-CNN**[2] is an object detection method which achieves near real-time rates using very deep networks, when ignoring the time spent on region proposals. Fast R-CNN employs several innovations to improve training and testing speed while also increasing detection accuracy. Fast R-CNN was able to solve the problem of speed by sharing computation of the conv layers between different proposals and swapping the order of generating region proposals and running the CNN. In this model, the image is first fed through a ConvNet, features of the region proposals are obtained from the last feature map of the ConvNet and fed to the final fully connected layers as well as regression and classification heads.

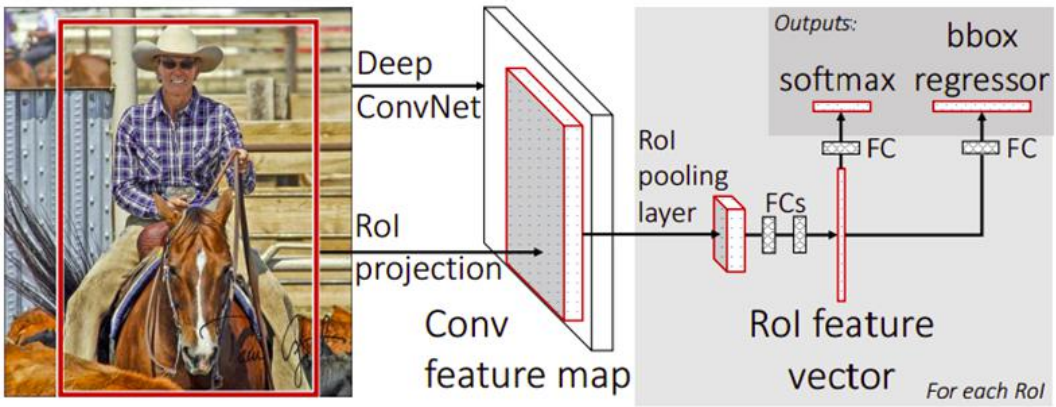


Figure 3.4: The Fast R-CNN workflow

Fast R-CNN method takes advantage of GPU while the region proposal methods are implemented on CPU. A way to speed up the object detection is by re-implemeting it on GPU. Region Proposal Network (RPN) is a way to do that. Region Proposal

Network (RPN), a network that shares full-image convolutional features with the detection network, enables nearly cost-free region proposals. An RPN is a fully convolutional network that simultaneously predicts object bounds and objectness scores at each position[13].

The RPN and Fast R-CNN were merged to a single network - **Faster R-CNN**. To unify RPNs with Fast R-CNN object detection networks, a training scheme that alternates between fine-tuning for the region proposal task and then fine-tuning for object detection, while keeping the proposals fixed, is used. This scheme converges quickly and produces a unified network with convolutional features that are shared between both tasks. Hence Faster R-CNN is composed of two modules. The first module is a fully convolutional network that proposes regions and the second one is the Fast R-CNN detector. The RPN module tells the Fast R-CNN module where to look.

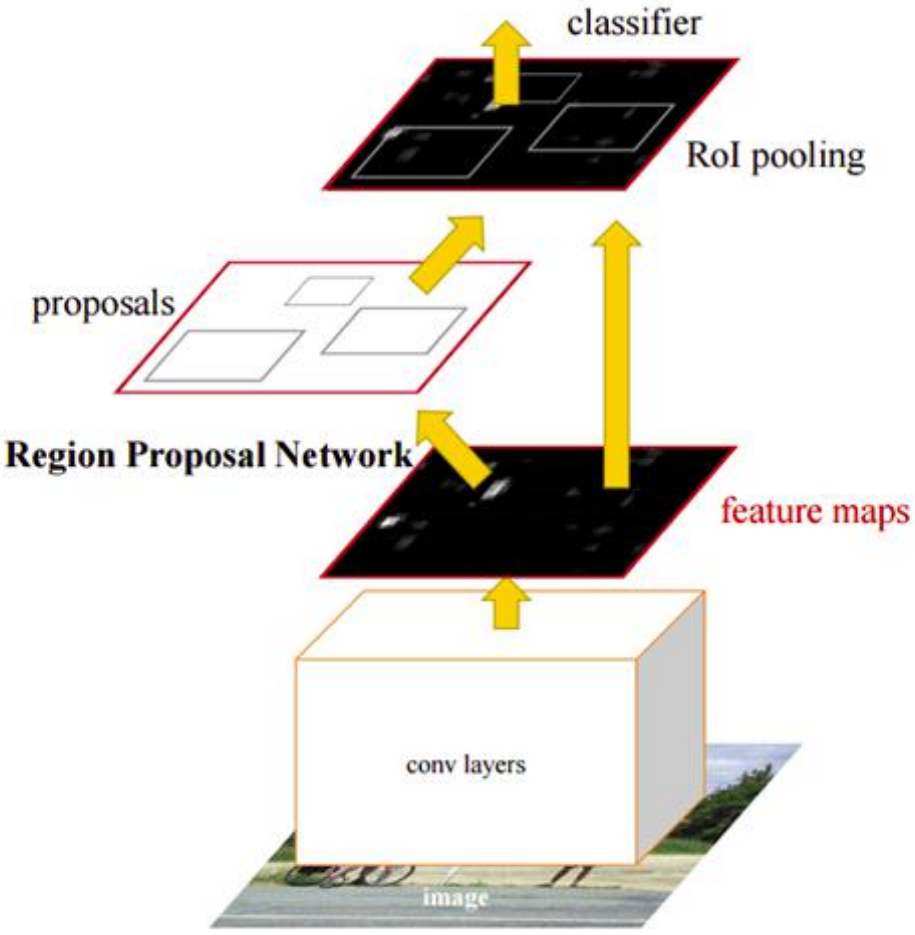


Figure 3.5: The Faster R-CNN workflow

In ILSVRC and COCO 2015 competitions, Faster R-CNN and RPN were the foundations of the first-place winning entries in several tracks. Due to the high detection accuracy (73.2% mAP on PASCAL VOC 2007 and 70.2% mAP on PASCAL VOC 2012) Faster R-CNN has the potential to be used for high quality object detection tasks. However, in the experiments done in this project, it was seen that the detection speed of faster R-CNN is not high enough to be used for real-time object detection applications like ours.

3.3 Object Tracking

Object tracking is the prediction of the future position of an object. A basic tracking module was developed as part of this project, mainly to aid with missing detections. The tracking module was written in C++, using opencv for the image manipulation. It takes as input the frames of a video and the bounding box coordinates of relevant objects received from the object detection system. The output is the predicted position (bounding box coordinates) of corresponding objects in the next frame. In addition to this, the module does track management too.

The prediction of next position of the object is done using Kalman filter. Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables.[1] The algorithm works in a two-step process. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement is observed, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty.

The algorithm is recursive. It can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.

The Kalman filter keeps track of the estimated state of the system and the variance

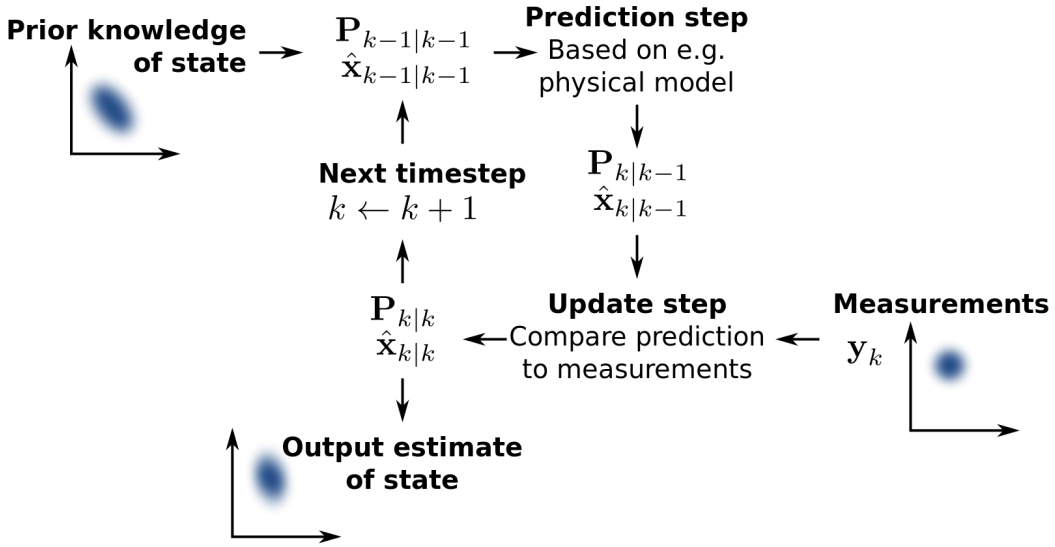


Figure 3.6: Kalman filter algorithm

or uncertainty of the estimate. The estimate is updated using a state transition model and measurements.$\hat{x}_{k|k-1}$ denotes the estimate of the system's state at time step k before the k-th measurement y_k has been taken into account;$\hat{P}_{k|k-1}$ is the corresponding uncertainty.

The Kalman filtering algorithm in our model is as follows:

- **Step 1:Initialization**
  Here we start with initial state which contains a state matrix,X_0- and process covariance matrix P_0.The state matrix contains the position of the object we need to track. Kalman filter is used to predict the new state given the previous state (x_{k-1}, p_{k-1}).We first previous state with initial state.
- **Step 2 : Estimating new state and process covariance matrix**
  The estimate of the new state:
  $$x_{k_p} = Ax_{k-1} + Bu_k + w_k$$
  The estimate of the new process covariance matrix:
  $$p_{k_p} = Ap_{k-1}A^T + Q_k$$
  where matrix A and B are determined according to the nature of the problem.
- **Step 3: Calculation of Kalman gain and updation of state and covariance matrix with new measurement**
  Kalman gain , $K = \frac{P_{k_p}}{P_{k_p}+R}$
  $$\hat{x}_{k|k-1} = x_{k_p} + K\left[Y - x_{k_p}\right]$$
  $$\hat{p}_{k|k-1} = \left[I - KI\right] p_{K_p}$$

Now this predicted state is made the previous state and these steps are iterated for a desired number of iterations.

The other major functions of this module are data association and track management. The Data Association function associates the bounding boxes predicted for a particular frame (second frame onwards) with the corresponding measured bounding boxes (if present) using k- nearest neighbour search. It also does track management by maintaining tracks, initiating a new track if needed, or deleting a track if measurements are absent for that particular track in three consecutive frames. In case the number of measured bounding boxes is more than the number of associated predicted ones in some frame, then it can be concluded that new ground truth tracks have been introduced. Then, new prediction tracks are initialized with the newly added ground truth data by this function.

Summary

In this chapter, we saw some popular CNN architectures for object recognition. Then, the potential of Faster R-CNN to be used for real-time purpose was explored. The Faster R-CNN based object detection model gives high detection accuracy. It has a dedicated Region Proposal Network for generating region proposals implemented in GPU. This speeds up the detection compared to Fast R-CNN, but still the detection time is too high for the model to be used for real-time detections. Hence, we had to look for faster detection models with high accuracy like SSD, which is presented in next chapter. A simple object tracking model developed was also discussed in this chapter.

CHAPTER 4

Single Shot MultiBox Detector for Object Detection and Ranging

SSD, or Single Shot MultiBox Detector, is a single-deep-neural-network based method for object detection in images.[10] In this method, the output space of the bounding boxes is a predefined set of boxes with different aspect ratios and scales per feature map location.By discretization of the output space to a set of default boxes, the need for object proposal generation is eliminated in SSD and hence the network is faster compared to other object detection systems. SSD is a feedforward CNN based object detection method which does not use any region proposals or resampling of pixels or features to produce bounding box hypotheses, but still is as accurate as the methods which do. The accuracy of SSD is comparable with the current state-of-the-art object detection systems. There is a pre-fixed collection of default bounding boxes and category scores and box offsets are predicted for each of these boxes. There have been single shot detectors like YOLO (You Only Look Once) before SSD, but SSD is significantly faster than those. Faster R-CNN, as seen before, is a slow technique which uses region proposal network and pooling, but highly accurate. SSD was found to be as accurate as Faster R-CNN, while being much faster.

On testing with PASCAL VOC 2007 dataset in Titan X GPU, SSD gave better performance with a detection speed of 59 FPS with mAP 74.3% compared to Faster R-CNN (12 FPS with mAP 73.2%) or YOLO (45 FPS with mAP 63.4%). SqueezeDet is a very small, low power, fully convolutional neural network for object detection. SqueezeDet coupled with the very small SqueezeNet model takes around 7MB size. The SqueezeDet model was pretrained for KITTI dataset. The pretrained network gave good performance on testing with KITTI dataset (57.2 FPS with mAP 76.7%), but the performance with PASCAL VOC was poor with a speed of around 41 FPS with very less precision. The advantage of SqueezeDet over SSD in terms of model size is insignificant when taking into account the detection performance of both. Hence SSD

was finally chosen as the model for object detection in this project.

We propose an extension to the current SSD network for estimating the depth of objects in the on-road object detection scenario. Given a monocular image of the scene in front of the vehicle, this modified network will detect the relevant objects and provide an estimate of the proximity of the objects from the ego-vehicle in meters.

4.1 Architecture of Single Shot MultiBox Detector

The architecture of SSD consists of a base network which is a standard object classification architecture like VGG-16, ResNet or ZFNet, truncated before the classification layers. An auxiliary structure is then added to this base network to produce detections. In the auxiliary structure, progressively decreasing convolutional layers are added which give feature maps of multiple scales. The multi-scale feature map characteristic allow predictions at different scales. Each added feature layer can produce a fixed set of detection predictions using a set of convolutional filters. The basic element of these filters for predicting the weights for a potential detection is a 3 x 3 x p convolutional kernel, where p is the number of channels in the input to the particular layer. This small kernel produces either a score for an object category or a bounding box shape offset relative to the default box coordinates. The default box coordinates are in turn relative to each feature map location.

SSD's output space comprises of a fixed collection of bounding boxes. A set of default bounding boxes is associated with each feature map cell, for the multiple feature maps formed at the top of the base network. Each feature map is spanned by boxes of different aspect ratios in a convolutional manner. At each feature map cell, we predict offsets relative to default box shapes in the cell as well as per class scores indicating the presence of objects of each class at the location. Hence, C+4 filters are applied around each bounding box, where C is the total number of object classes, to give C confidence score values and 4 offset values. If there are K such bounding boxes per feature map cell, there are $(C + 4) \cdot K$ filters per cell. Thus, for an m x n feature map, there are $(C + 4) \cdot K \cdot m \cdot n$ convolutional filter outputs.

During training we need to determine which default boxes correspond to a ground truth detection and train the network accordingly. For each ground truth box we are selecting from default boxes that vary over location, aspect ratio, and scale. Each ground truth box is matched to the default box with the best jaccard overlap. The default boxes are then matched to any ground truth with jaccard overlap higher than a threshold (default = 0.5). This simplifies the learning problem, allowing the network to predict high scores for multiple overlapping default boxes rather than requiring it to pick only the one with maximum overlap.

A layer by layer description of the caffe-ssd architecture follows in the subsequent sections.

4.1.1 Data Layer

Data enters the ssd network through the Data Layer which lies at the bottom of the network. The image data is stored in the top blob named 'data' after necessary processing steps. In the train and test phase, the data layer is replaced by an Annotated Data Layer. The Annotated Data Layer splits the input data into images and annotations. Then, preprocessing steps like resizing, expanding, random cropping, distorting, scaling, etc are done on the images and the annotation is changed accordingly if required. The layer gives two blobs as outputs - data and label. Data blob stores the processed images while the label blob stores the modified annotations.

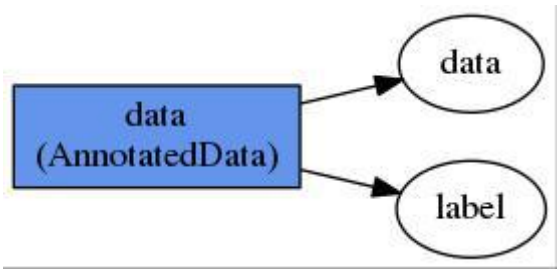


Figure 4.1: Annotated Data Layer

The dimensions of data blob are NxCxHxW, where N is the batch size, C is the

feature dimension (number of channels), H is the height of the image, and W is the width. The dimensions of label blob are 1x1x num_bbox x 9, where num_bbox is the number of annotated objects in the image. Since the number of annotated objects can be different for each image, the bounding box information is stored such that all bounding boxes are stored in one spatial plane (num and channels are 1) and each row contains only one box in the following format:
[item_id, group_label, instance_id, xmin, ymin, xmax, ymax, diff],
where item_id corresponds to the ordinality of the particular image in the batch, group_label is the integer value assigned to the object class, instance_id gives the count of occurrence of objects belonging to this particular class, xmin, ymin, xmax and ymax are the bounding box coordinates and diff is used to label this object as difficult or not.

Each image in the batch is read from the LMDB file and preprocessing steps are done according to the transformation parameters specified. If distortion parameters are present, changes in brightness, saturation, hue, contrast and channel order are done according to the parameters.

The two main data augmentation strategies used in SSD are expansion and batch sampling, which are essentially "zoom out" and "zoom in" operations respectively. Expansion is a data augmentation trick in which a cv::Mat array filled with mean values is created such that its size is an expanded version of the original input size and the input image is pasted at some random position in this canvas. The expand parameter to be supplied includes a max_expand_ratio, which determines the maximum size the expanded image can have. An expand_ratio is randomly selected from the uniform distribution in the range 1 to max_expand_ratio. The width and height of the mean image will be expand_ratio times that of the input image.

$$\begin{aligned} height &= img\_height \times expand\_ratio \\ width &= img\_width \times expand\_ratio \end{aligned} \tag{4.1}$$

A cv::Mat array of same type as the input image is created with the height and width as in the equations above, and filled with mean values. The position at which the original image is to be pasted is also selected randomly. The top-left coordinates of the

box into which the image is to be copied, h_off and w_off, are chosen randomly from the uniform distributions in the ranges [0, height-img_height] and [0, width-img_width] respectively. The input image is decoded into a cv::Mat and then copied to the scaled up mean frame at the location specified by h_off and w_off. The expanded image is encoded back to the specified format. Then the annotations for this image are also modified accordingly. The bounding box coordinates are changed according to the expansion ratio and the location on the mean frame where the image is pasted.

The other data augmentation technique which gives a "zoom in" effect is batch sampling. It involves random cropping of the expanded image by constructing sample bounding boxes such that certain constraints are met. The batch sampler parameters include sampler parameters, sample constraints, max_trials and max_sample. The sampler parameters minimum scale, maximum scale, minimum aspect ratio and maximum aspect ratio possible for the sample bboxes. Sample constraints can be concerning jaccard overlap, sample coverage, or object coverage. For each image in the batch, sample bboxes are constructed with a scale and aspect ratio chosen randomly from the range specified by sampler parameters. The sample bbox width and height are calculated from the chosen scale and aspect ratio as follows:

$$\begin{aligned} bbox\_width &= scale \times \sqrt{aspect\_ratio} \\ bbox\_height &= \frac{scale}{\sqrt{aspect\_ratio}} \end{aligned} \tag{4.2}$$

The top-left corner position of the sample bbox is also randomly chosen. If this sample bbox meets the sampler constraints or if no sampler constraints are mentioned, it is stored for the final selection process. If minimum jaccard overlap constraint is specified, the sample bbox is selected only if it has the minimum overlap required with any of the ground truth bboxes of this image. Similarly, other constraints such as sample coverage and object coverage are also checked. Sample coverage is the ratio of area of intersection between the sample bbox and any ground truth object bbox to the sample bbox size. Object coverage is the ratio of area of intersection between the sample bbox and any ground truth object bbox to the object bbox size.

There might be several sets of batch sampler parameters present. For each set of

parameters sample bboxes are generated and tested for constraint matching until the total number of sample bboxes selected is equal to max_sample or the number of sample bboxes generated in total is equal to max_trials. For each image, out of the several sample bboxes selected, one is randomly chosen and the image is cropped along that bbox. The annotations are suitably modified.

After the data augmentation step, any data transformations like mirror, resize are done as specified by the transform parameters. Ultimately, the transformed data and annotations are stored in data and label blobs respectively.

4.1.2 Base Network and Extra Feature Layers

In SSD network, multi scale feature layers are added on top of a base network which can be formed by choosing the initial layers (other than the ones responsible for classification) from any standard architecture for high quality image classification like VGGNet, ResNet, AlexNet, ZFNet etc. The extra feature layers remain same irrespective of the base network. In this section, for the description of base network, VGG-16 has been chosen. The base VGG16 network is truncated after the fully connected layer fc7 (before classification layers) and the extra structure added to it performs the detection function. The auxiliary structure comprises of 6 convolutional layers producing feature maps which are progressively decreasing in size. The convolutional model for predicting detections is different for each feature layer. An average global pooling layer, pool6, at the end gives a single dimensional vector output.Figure 4.2 shows the SSD model with VGG-16 base network for an input size of 300x300.

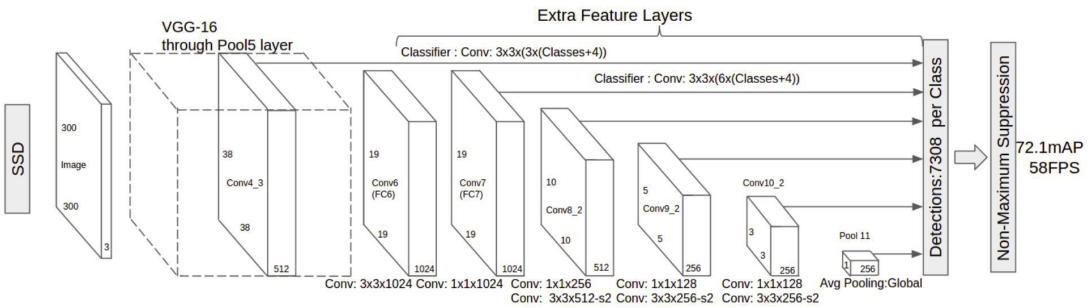


Figure 4.2: SSD model showing base network and extra feature layers for input size of 300x300

Sl no.	Layer Name	Layer Type	Top Layer	Kernel Size	Stride	Pad	Dilation	Number of Kernels	Input dimension	Output Dimension
1	conv1_1	Convolution	conv1_2	3	1	1	0	64	300x300x3	300x300x64
2	conv1_2	Convolution	pool1	3	1	1	0	64	300x300x64	300x300x64
3	pool1	Max-Pool	conv2_1	2	2	0	0	-	300x300x64	150x150x64
4	conv2_1	Convolution	conv2_2	3	1	1	0	128	150x150x64	150x150x128
5	conv2_2	Convolution	pool2	3	1	1	0	128	150x150x128	150x150x128
6	pool2	Max-Pool	conv3_1	2	2	0	0	-	150x150x128	75x75x128
7	conv3_1	Convolution	conv3_2	3	1	1	0	256	75x75x128	75x75x256
8	conv3_2	Convolution	conv3_3	3	1	1	0	256	75x75x256	75x75x256
9	conv3_3	Convolution	pool3	3	1	1	0	256	75x75x256	75x75x256
10	pool3	Max-Pool	conv4_1	2	2	0	0	-	75x75x256	38x38x256
11	conv4_1	Convolution	conv4_2	3	1	1	0	512	38x38x256	38x38x512
12	conv4_2	Convolution	conv4_3	3	1	1	0	512	38x38x512	38x38x512
13	conv4_3	Convolution	pool4	3	1	1	0	512	38x38x512	38x38x512
14	pool4	Max-Pool	conv5_1	2	2	0	0	-	38x38x512	19x19x512
15	conv5_1	Convolution	conv5_2	3	1	1	0	512	19x19x512	19x19x512
16	conv5_2	Convolution	conv5_3	3	1	1	0	512	19x19x512	19x19x512
17	conv5_3	Convolution	pool5	3	1	1	0	512	19x19x512	19x19x512
18	pool5	Max-Pool	fc6	3	1	1	0	-	19x19x512	19x19x512
19	fc6	Convolution	fc7	3	1	6	6	1024	19x19x512	19x19x1024
20	fc7	Convolution	conv6_1	1	1	0	0	1024	19x19x1024	19x19x1024
21	conv6_1	Convolution	conv6_2	1	1	0	0	256	19x19x512	19x19x256
22	conv6_2	Convolution	conv7_1	3	2	1	0	512	19x19x256	10x10x512
23	conv7_1	Convolution	conv7_2	1	1	0	0	128	10x10x512	10x10x128
24	conv7_2	Convolution	conv8_1	3	2	1	0	256	10x10x128	5x5x256
25	conv8_1	Convolution	conv8_2	1	1	0	0	128	5x5x256	5x5x128
26	conv8_2	Convolution	pool6	3	2	1	0	256	5x5x128	3x3x256
27	pool6	AVE-Pool	Multibox	0	1	0	0	-	3x3x256	1x1x256

Table 4.1: Base network and auxiliary layer description for input size of 300x300

The detailed description of the various convolutional layers and pooling layers in the base VGG-16 network and auxiliary structure for an input dimension of 300x300 is given in table 4.1.

4.1.3 MultiBox Head

The mutibox layers are a set of convolutional,normalization, permute ,flatten , priorbox and concat layers which are used for location prediction, confidence prediction and priorbox generation. The mbox layers take input from feature maps at various depths in the network to enable predictions for different scales of input. In the case of VGG-16 base network, the input is taken from conv4_3 and fc7 layers in the base network, and conv6_2, conv7_2, conv8_2, conv9_2 layers in the auxiliary set. These are referred to as mbox source layers.

When combining information from multiple feature maps, the scale of features from those layers being very different makes the process difficult. L2 normalizing features from some layers and then combining them with a scale factor learned through back propagation makes it easier[11]. For this purpose, the output from conv4_3 layer is L2 normalized using a normalization layer and scaled with an initial factor of 20.

The priorbox layer is responsible for the generation of prior bounding boxes for each of the 6 source feature maps. This layer takes input from the corresponding mbox source layer as well as the data layer. The output blob contains two channels - one for storing the priorbox coordinates for the particular feature map and the other for storing the prior variance. The size and aspect ratios of the priorboxes at each feature map cell is calculated using the preset multibox parameters provided for each source layer. The different multibox parameters which can be set are aspect ratios, minimum size, and maximum size of the priorboxes, min_ratio and max_ratio. The size and aspect ratio parameters can be different for the different mbox source layers. The default value of min_ratio is 20 and that of max_ratio is 90. This means that the lowest layer has a scale of 0.2 and the highest layer has a scale of 0.9, and all layers in between are regularly spaced. Then the scale of the default boxes for the k^{th} feature map is computed as:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1} \times (k - 1), k \in [1, m] \tag{4.3}$$

where m is the number of feature maps used for prediction.
We impose different aspect ratios for the default boxes, and denote them as $a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$ .We can compute the width (${w_k}^a = s_k\sqrt{a_r}$) and height (${h_k}^a = s_k/\sqrt{a_r}$) for each default box.The mbox source layer's feature map size and the original image size are used for deducing the positions of priorboxes.

The location prediction layer is a convolution layer with kernel size = 3, pad = 1, and stride = 1. Initialization of the weights of this convolutional layer is done using xavier algorithm. The bias is filled with a constant value of zero. After proper tuning, the location prediction layer gives offset values for each priorbox in the output space so that the priorbox, when the coordinates are adjusted according to the offsets, contains an object of some class. SSD provides the option to share the location predictions among all the object classes of interest, or to generate separate location predictions for individual classes at each priorbox location. This is carried out using a boolean variable 'share_location'. When share_location is true, only one set of predictions are produced per priorbox location. Hence, the number of outputs (kernels) in the convolution layer should be:

$$num\_kernels = 4 \times num\_priors\_per\_location \tag{4.4}$$

If location prediction were not shared, the number of kernels would be:

$$num\_kernels = 4 \times num\_priors\_per\_location \times num\_classes \tag{4.5}$$

The output blob from the convolution layer is permuted and flattened using permutation and flatten layers so as to make the final concatenation easy.

Confidence prediction layer, similar to location prediction layer, is a convolution layer with kernel size = 3, pad = 1, and stride = 1. The weight filler used is xavier and bias filler used is of constant type with value zero. The confidence prediction layer gives an estimate of the confidence scores for the presence of different classes in the particular priorbox location when passed through a softmax function. Hence, number

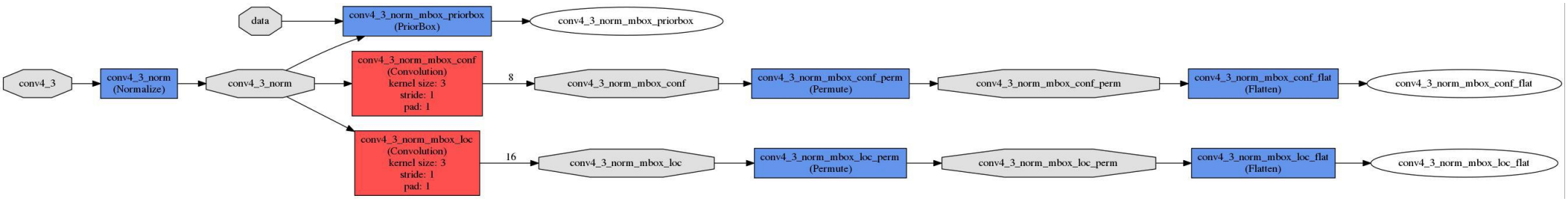


Figure 4.3: The mbox layers with source layer conv4_3

of kernels will be:

$$num\_kernels = num\_priors\_per\_location \times num\_classes \tag{4.6}$$

Similar to the location prediction layer, the output of this layer is permuted and flattened too.

After generating the location prediction, confidence prediction and priorbox generation layers for all the mbox source layers, the are concatenated to give mbox_loc, mbox_conf and mbox_priorbox blobs. Figure 4.4 shows the concatenated layers.

4.1.4 MultiboxLoss Layer

In the training phase, the location prediction loss and confidence prediction loss w.r.t the ground truth have to be computed for tuning the network through back propagation. The loss computation is done in multibox loss layers. The overall objective loss function is a weighted sum of the localization loss (loc) and the confidence loss (conf).Smooth L1 loss is selected as the loss function for localization loss and softmax is chosen for confidence loss.

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)) \tag{4.7}$$

where ${x^p}_{ij} = \{1, 0\}$ is the indicator for matching i^{th} default box to the j^{th} ground truth box of category p, c is the class confidence, l is the predicted bounding box location, g is the ground truth box parameter,and N is the number of matching default boxes.

The localization loss is a smooth L1 loss between predicted box and ground truth box parameter.

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in \{cx,cy,w,h\}} {x^k}_{ij} smooth_{L1}({l^m}_i - \hat{g}_j^m) \tag{4.8}$$

where, cx,cy are the offsets for center coordinates and w,h are the width and height of the bbox.

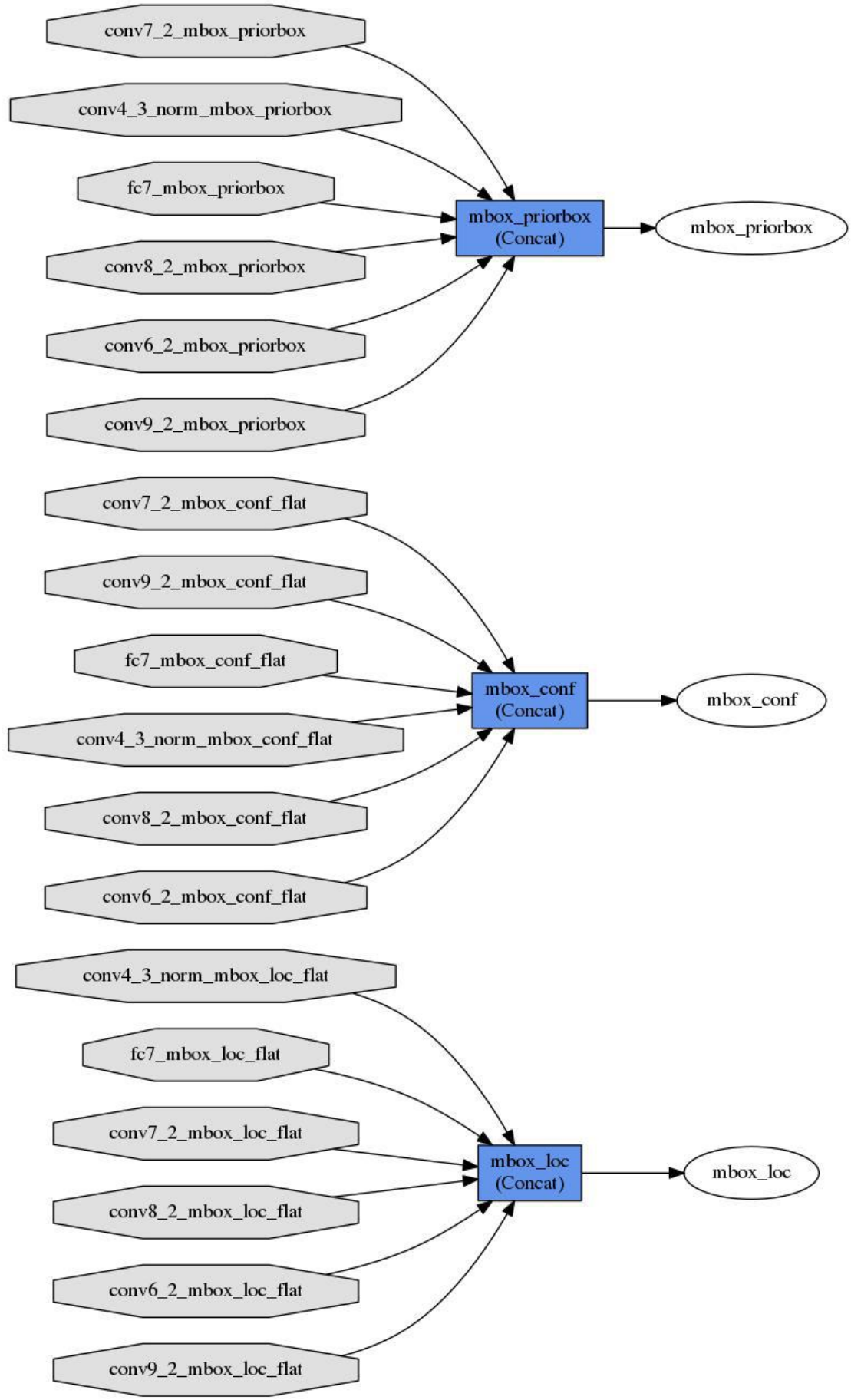


Figure 4.4: The concatenated mbox layers

The confidence loss is of softmax type. It corresponds to the confidence score for the presence of a class in the matched bounding box.

$$L_{conf}(x,c) = -\sum_{i\in Pos}^{N} x^{p}{}_{ij} log(\hat{c}_i^p) - \sum_{i\in Neg} log(\hat{c}_i^0) \tag{4.9}$$

where $\hat{c}_i^p = \frac{exp(c_i^p)}{\sum_p exp(c_i^p)}$.
The weight term α is set to 1 through cross validation.

The inputs to the multibox loss layer include the output blobs from mbox location prediction, confidence prediction and priorbox layers, as well as the ground truth data from the label blob.

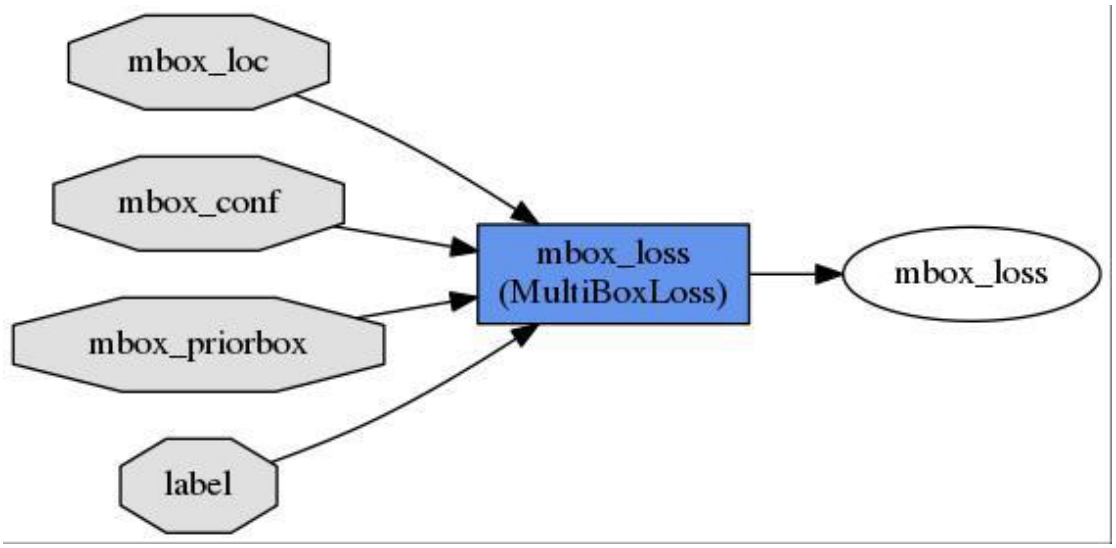


Figure 4.5: The mbox loss layer

The ground truth data is retrieved from the label layer. The priorbox locations and their corresponding variances are fetched from the mbox_priorbox blob. The location predictions and confidence predictions are also retrieved from mbox_loc and mbox_conf respectively. If share_location is true, the number of location predictions will be 4 x num_priors. Otherwise, it will be 4 x num_priors x num_classes. Number of confidence predictions will be num_priors x num_classes. Once the required data is recovered, matches between the priorbox locations and ground truth bboxes have to be found. Matching pairs of priorboxes and ground truth bboxes are found for all images in the batch in terms of Jaccard overlap between them, and stored. After storing all possi-

ble matches between priorboxes and ground truth bboxes, hard negative mining is done.

After the matching step, most of the default boxes are negatives, especially when the number of possible default boxes is large. This introduces a significant imbalance between the positive and negative training examples. Instead of using all the negative examples, they are sorted using the highest confidence loss for each default box and the top ones are picked so that the ratio between the negatives and positives is atmost 3:1. This is called hard negative mining. This leads to faster optimization and a more stable training. Currently, hard negative mining is supported only if share_location is true.

Location predictions are in the form of offsets to the default priorboxes. Location prediction is encoded into the priorboxes by applying these offsets. Similarly, the confidence scores, which are computed by applying a softmax function to the mbox_conf output values, are also encoded to the default bboxes. Loss is computed using equation 4.7 for all the positive examples and the mined negative ones. It is then back propagated to the relevant layers for tuning.

4.1.5 Detection Output Layer

When the network is deployed for object detection, the detection output layer is responsible for generating the results. The top blob of the detection output layer has the dimensions 1 x 1 x num_bboxes x 7, where num_bboxes is the total number of predicted bounding boxes with sufficiently high confidence scores for the presence of an object of interested class in it. Each row stores information of one output bounding box in the following format:
[image_id, label, confidence_score,xmin,ymin,xmax,ymax]
The output can either be written into text files in some specific output format (eg. voc style), or visualized using openCV tools. While visualizing, the integer label value is mapped to its corresponding display name using the labelmap file.

The detection output layer gets data from mbox_loc, mbox_conf and mbox_priorbox layers. The location predictions are retrieved and decoded into bounding boxes. The

confidence scores for each of these boxes are computed by applying softmax function to mbox_conf output. This is followed by a non-maximum suppression (NMS) step to select the most relevant, non-overlapping bounding boxes.

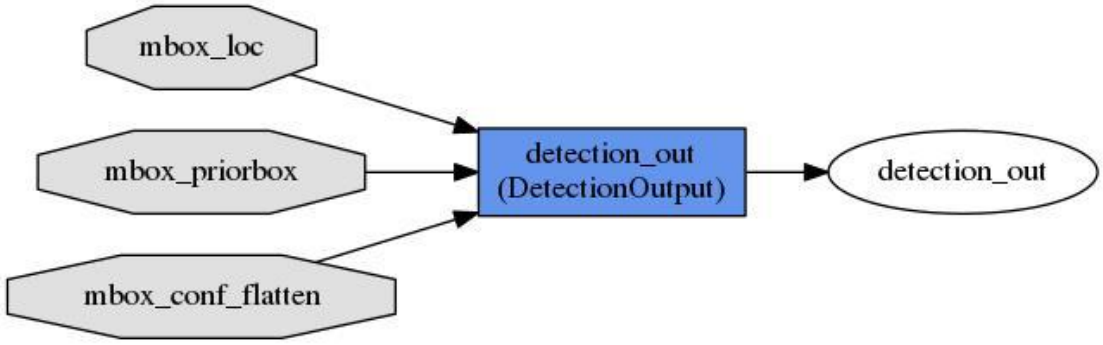


Figure 4.6: The detection output layer

For applying NMS, the decoded bounding boxes are sorted in the descending order of the confidence scores associated with them. The first bbox in the list is selected for displaying in the result by default. The subsequent bounding boxes in the list are selected only if the jaccard overlap between the bbox and all previously stored bboxes is less than or equal to a preset nms threshold value. By this process, conflicting bounding boxes at the same location will be removed based on their confidence score.

In the test phase, an additional detection evaluation layer is added after the detection output layer to evaluate the performance of the model. The detection evaluation layer receives ground truth data from the label layer. This data is used to estimate the average detection precision (mAP) of the network. Also, the layer computes the number true positives and false positives in the detection.

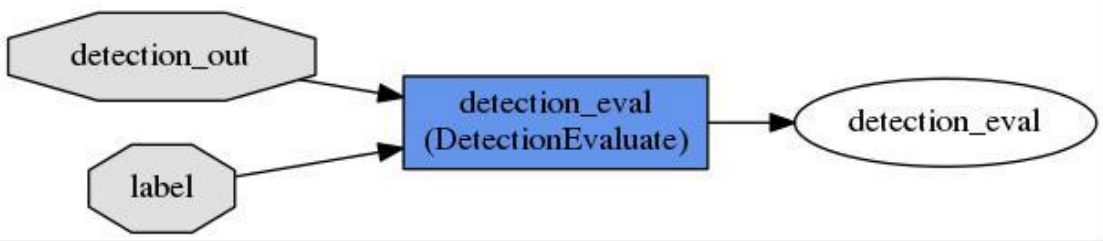


Figure 4.7: The detection evaluation layer

4.1.6 Training Dataset Preparation and LMDB Creation

The KITTI Vision Benchmark Suite's object dataset was used for training the SSD detection and ranging network. The dataset comprises of 7481 training images and 7518 testing images. The label files contain 9 attributes for each image as descibed below. All values (numerical or strings) are separated via spaces and each row corresponds to one object. The description of the 15 columns in the label files are given in table 4.2.

Number of Values	Name	Description
1	type	Describes the type of object: 'Car', 'Van', 'Truck', 'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram', 'Misc' or 'DontCare'
1	truncated	Float from 0 (non-truncated) to 1 (truncated), where truncated refers to the object leaving image boundaries
1	occluded	Integer (0,1,2,3) indicating occlusion state: 0 = fully visible, 1 = partly occluded, 2 = largely occluded, 3 = unknown
1	alpha	Observation angle of object, in the range $[-\pi, \pi]$
4	bbox	2D bounding box of object in the image (0-based index): contains left, top, right, bottom pixel coordinates.
3	dimensions	3D object dimensions: height, width, length (in meters).
3	location	3D object location x,y,z in camera coordinates (in meters)
1	rotation_y	Rotation ry around Y-axis in camera coordinates $[-\pi..\pi]$
1	score	Only for results: Float, indicating confidence in detection, needed for p/r curves. Higher is better.

Table 4.2: The KITTI dataset label file description

The label files were converted into PASCAL VOC style annotation, which is a widely accepted annotation format for most of the CNN models. The annotation files are saved in xml format. Since the 'difficult' attribute needed for voc style formatting is not available in the KITTI label files, the said field was set as zero for all objects. A sample file is given below:

```
<annotation>
    <folder>KITTI</folder>
    <filename>000000.png</filename>
    <source>
        <database>The KITTI Database</database>
        <annotation>KITTI</annotation>
    </source>
    <size>
        <width>1224</width>
        <height>370</height>
        <depth>3</depth>
    </size>
    <object>
        <name>Pedestrian</name>
        <truncated>0.00</truncated>
        <occluded>0</occluded>
        <bndbox>
            <xmin>712</xmin>
            <ymin>143</ymin>
            <xmax>810</xmax>
            <ymax>307</ymax>
        </bndbox>
        <difficult>0</difficult>
    </object>
</annotation>
```

For SSD, we use a Lightning Memory-Mapped Database (LMDB) to provide data into the caffe network. LMDB is usually the choice of database in caffe when dealing with large datasets. In LMDB the data is stored in the form of key-value pairs. The key/data pairs are stored as byte arrays. LMDB supports multiple data items for a single key. LMDB uses memory-mapped files, giving much better I/O performance.

The training images are arbitrarily divided into two sets - an actual 'train set' containing images which will be used for tuning the network, and a 'test set' which will be used for validation. The test set contains a small number of images which have no direct effect on how the weights are learned by the network. Instead, it helps in evaluating how well the weights are being learned during the course of training. A test phase is carried out after each set of a preset number of iterations of the training phase. In the test phase, the current model is used for detecting objects of interest in the test set images. At the end of this the detection accuracy is computed, which helps in estimating if the model parameters are being tuned properly for the particular dataset.

Separate LMDB files are created for the test set and train set. SSD's tool for LMDB generation takes as input text files (list files - named trainval.txt and test.txt). Each line in the list file contains the path to the location of an image file and its corresponding voc style annotation file. There are separate list files for the training set and the test set. A few lines from a sample list file are given below:

KITTI/Images/005203.png KITTI/Annotations/005203.xml
KITTI/Images/003451.png KITTI/Annotations/003451.xml
KITTI/Images/000587.png KITTI/Annotations/000587.xml
KITTI/Images/007337.png KITTI/Annotations/007337.xml
KITTI/Images/000488.png KITTI/Annotations/000488.xml

A protocol buffer named 'AnnotatedDatum' is used to store the images and their annotation information, before converting to LMDB bytestring. AnnotatedDatum is an extension of the Datum class to incorporate rich annotations. The differet fields in AnnotatedDatum message include: Datum (to store images as bytes), AnnotationType (to specify the type of annotation. Currently it only supports bounding box), and AnnotationGroup (To store a group of annotations belonging to a particular class). The AnnotationGroup has a group_label field to store the integer label corresponding to the particular class in consideration and an Annotation field which stores annotations for each instance of an object in the corresponding class. A 'NormalizedBBox' container is used to store annotation information such as normalized bounding box coordinates (normalized w.r.t the input image size),label, difficulty, score (only for results) and size,

for each object.

OpenCV tools are used for any preprocessing of the data if required. The image is read to a cv::Mat array and preprocessing steps such as resizing, scaling are done if necessary. The processed image is encoded with the specified extension and then read into a 'Datum' buffer. The xml type annotation files for these images are then read and the required information is retrieved. The class label for each object mentioned in the annotation file is mapped into an integer value with the help of a 'LabelMap' prototxt file. The LabelMap file contains the information required to assign a numerical value to the string type class label. In this file we can mention the desired display name for each class also. A sample LabelMap file for a single object class (Car) is given below:

```
item {
name: "none_of_the_above"
label: 0
display_name: "background"
}
item {
name: "Car"
label: 1
display_name: "Car"
}
```

Annotation groups are formed for each label mentioned in the LabelMap file. The annotations are added to these groups according to the class to which the object belongs. The AnnotatedDatum for each image is serialized into a bytestring and stored with a unique key in the LMDB format.

4.2 Modified SSD Network for Object Detection and Range Estimation

In this project, the ssd network was modified to incorporate a range estimation feature for autonomous driving. By this, we can estimate the range at which each detected object is w.r.t the ego vehicle. The output of the network contains a range value in meters along with the bounding box coordinates for each detected object. This feature eliminates the need to use LiDAR or RADAR systems for the assessment of proximity of the on-road objects in autonomous vehicles. A monocular image of the scene captured using a camera is sufficient for the estimation of proximity of objects in the scene from the ego-vehicle.

The annotations were modified to include the range value also. The LiDAR data from KITTI was used to get the range information of the annotated objects. The range was calculated by taking the euclidean distance from the location of ego-vehicle (0,0,0) to that of the concerned object. The xml files were created with an extra attribute-'range' - in the bounding box information. An example is given below.

```
<bndbox>
    <xmin>712</xmin>
    <ymin>143</ymin>
    <xmax>810</xmax>
    <ymax>307</ymax>
    <range>0.13</range>
</bndbox>
```

In the training phase, an annotated data layer is employed by the network to split the images and annotations from the incoming data. Now the ground truth information has an extra field - 'range'. Necessary changes were made to the label blob output of the annotated data layer to accommodate this change. In particular, now the bounding boxes are stored in each row of the label blob in the format:

[item_id, group_label, instance_id, xmin, ymin, xmax, ymax,range, diff]

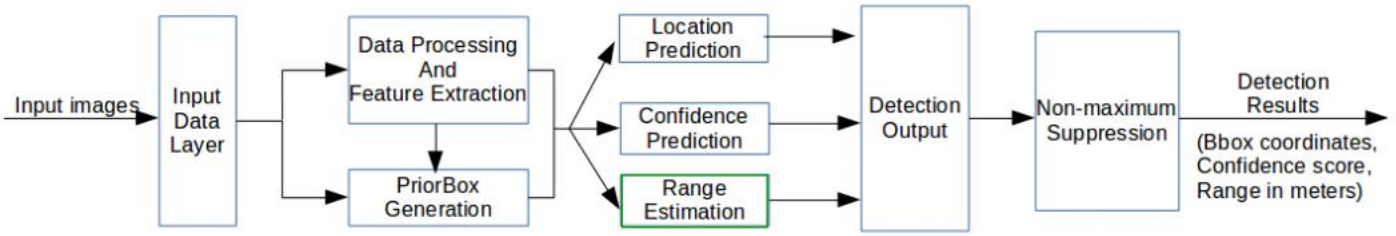


Figure 4.8: Block Diagram of the Modified SSD for Object Detection and Ranging: The new block introduced is shown in green.

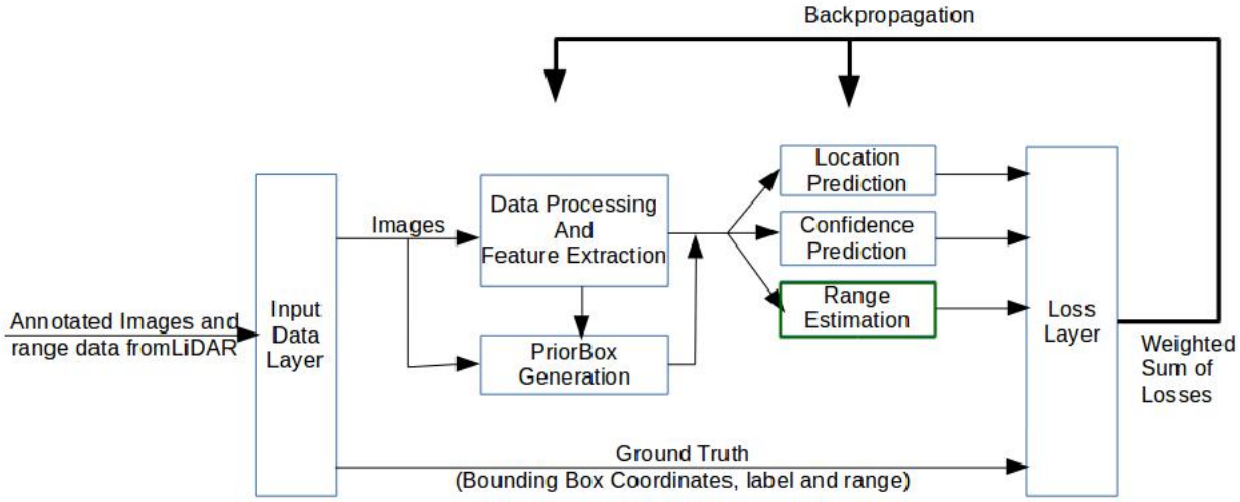


Figure 4.9: Block Diagram showing training procedure for the Modified SSD network: The new block introduced is shown in green.

The data augmentation tricks in SSD, namely expansion and batch sampling, aid in improving the results and generate more training samples. However, these methods disturb the global attributes of the image. Range estimation can be affected by such processes. Hence, they were avoided in the modified network. Since our model is focused on on-road object detection, the number of object classes for which the network is to be trained is less. Thus the removal of these data augmetation techniques do not create a huge impact on the results. Other data transformation steps such as mirroring, distortion etc which do not disturb the are retained.

The multibox layers are responsible for confidence prediction and location prediction in SSD. For range estimation, an additional layer was added to the set of multibox layers - the proximity prediction layer. Similar to other prediction layers, this is also a convolution layer with kernel size = 3, pad = 1, stride = 1. The weight filler is based on xavier algorithm and bias is set as zero. The proximity prediction layer generates one range value per priorbox. Hence the number of kernels in the convolution layer will be:

$$num\_kernels = 4 \times num\_priors\_per\_location \tag{4.10}$$

where num_priors_per_location is the number of default boxes at each feature map cell. The output of this convolution layer is also permuted and flattened for final concatenation. Figure 4.11 shows the mbox layers for the modified network.

The concatenation operation after generating the prediction laters and prior generation layers for all source layers now includes an mbox_prox blob too, which gives an estimate of proximity of the objects from ego vehicle. The figure 4.10 shows the concatenated mbox layers of the modified network.

The training objective was modified to incorporate the range prediction loss also. Eucidean loss (L2) was chosen as the loss function for range estimation. The new training objective is a weighted sum of confidence loss, location prediction loss and range estimation loss.

$$L(x, c, l, r, g_b, g_r) = \frac{1}{N}(\alpha L_{conf}(x, c) + \beta L_{loc}(x, l, g_b) + \gamma L_{prox}(x, r, g_r)) \tag{4.11}$$

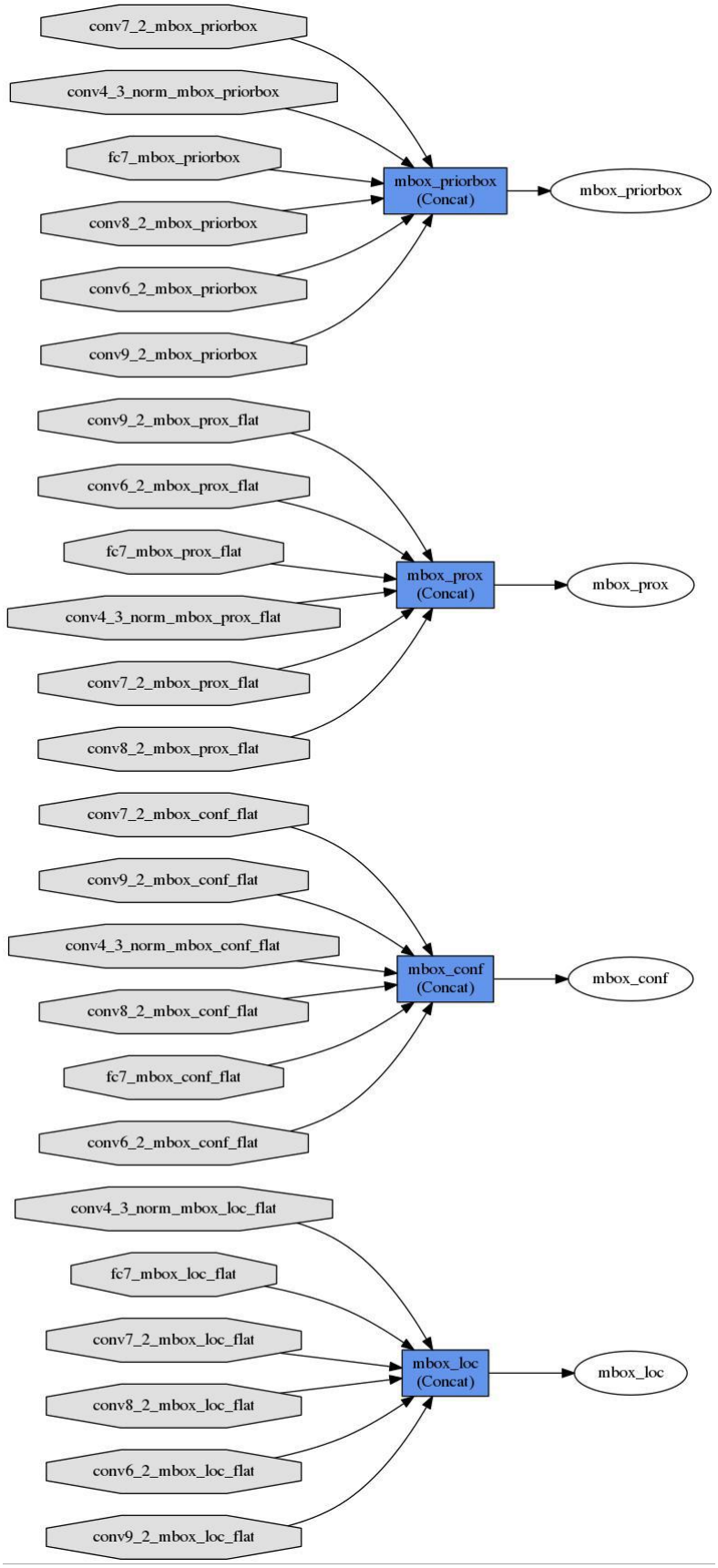


Figure 4.10: The concatenated mbox layers of the modified network

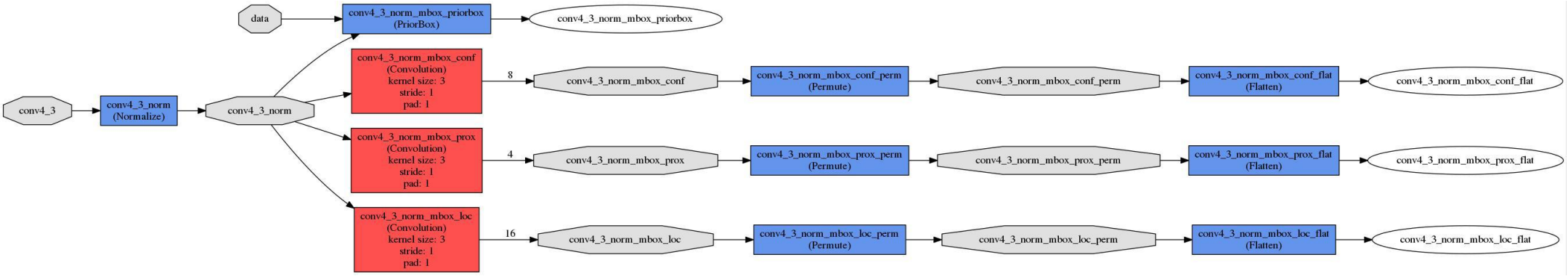


Figure 4.11: MultiBox layers of the modified network for the source layer conv4_3

where $x_{ij}^{p} = \{1, 0\}$ is the indicator for matching i^{th} default box to the j^{th} ground truth box of category p, c is the class confidence, l is the predicted bounding box location, r is the predicted range, g_b is the ground truth box parameter, g_r is the ground truth range parameter and N is the number of matching default boxes.

The range estimation loss is an L2 loss function between the estimated range and ground truth range parameter.

$$L_{prox}(x, r, g_r) = \sum_{i \in Pos}^{N} x_{ij}^{p} \times (r_i - g_{r_j})^2 \tag{4.12}$$

The weights of the loss functions α, β and γ when set as 1 gave the best performance.

Thus the mbox_loss layer now takes input from the mbox_prox layer too, from which the proximity estimates are retrieved.

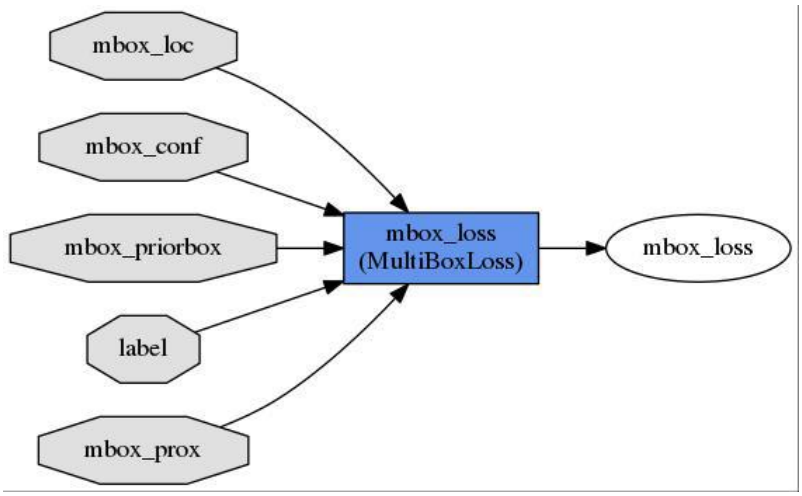


Figure 4.12: The mbox loss layer of the modified network

The detection output layer was also tweaked to add range estimate also in the results. The range estimate is retrieved from the mbox_prox blob along with other predictions and decoded into bounding boxes.

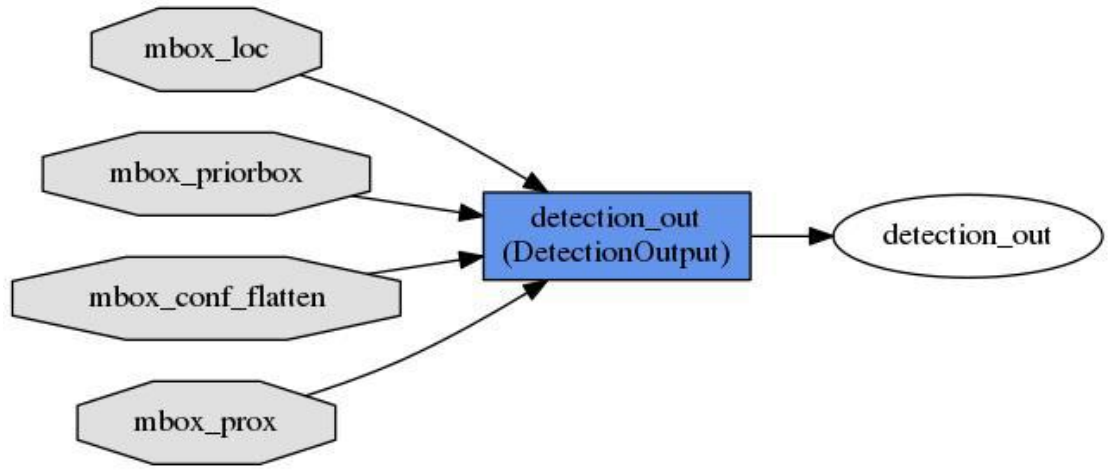


Figure 4.13: The detection output layer of modified network

The output of the detection output layer for each bounding box is now in the format:

[image_id, label,range, confidence_score,xmin,ymin,xmax,ymax]

Summary

In this chapter, we discussed the Single Shot MultiBox Detector which is a highly accurate and fast method for object detection, which can be used in real-time. The improvement in speed compared to Faster R-CNN model is due to the elimination of region proposal methods. Instead, the output space is discretized into a set of default bounding boxes. The modification we propose to the SSD network for estimating the proximity of the detected objects from ego-vehicle was also described. The modified network gives range estimate in meters with reasonable accuracy.

CHAPTER 5

Experiments and Results

5.1 Fine-Tuning Faster R-CNN for On-road Object Detection

Training was done using PASCAL VOC 2012 dataset with CAFFE deep learning framework. The PASCAL VOC 2012 dataset contains 20 classes. The train/val data has 11,530 images containing 27,450 ROI annotated objects. A pre-trained imagenet model was used as the initial weight for the training procedure. The training scheme used was a 4 step alternating training in which we first train RPN, and use the proposals to train Fast R-CNN, and the process is iterated. Two models were trained using ZFNet and VGG-16 architectures. The trained networks were tested on NVIDIA GeForce GTX 980 Ti.

For our purpose, detection of objects belonging to 20 classes was unnecessary as the focus is on on-road objects. Hence two other networks were trained by fine-tuning the pretrained model to detect 5 classes from PASCAL VOC 2012 dataset - 'Bicycle', 'Bus', 'Car', 'Motorbike', and 'Person' - instead of 20 classes. 3040 images were selected for training. The number of images for each class is as listed below:

- Bicycle - 481 images
- Bus - 412 images
- Car - 1000 images
- Motorbike - 496 images
- Person - 1969 images

The pre-trained imagenet model was used as the initial weight for the training procedure. The training parameters are as follows:

Fast RCNN - Stage 1:

- base_lr: 0.0001
- stepsize: 20000
- lr_policy: "step"
- gamma: 0.1
- average_loss: 100
- momentum: 0.9
- weight_decay: 0.0005
- max_iter: 200000

RPN - Stage 1:

- base_lr: 0.0001
- stepsize: 40000
- lr_policy: "step"
- gamma: 0.1
- average_loss: 100
- momentum: 0.9
- weight_decay: 0.0005
- max_iter: 400000

Fast RCNN - Stage 2:

- base_lr: 0.0001
- stepsize: 20000
- lr_policy: "step"
- gamma: 0.1
- average_loss: 100
- momentum: 0.9
- weight_decay: 0.0005
- max_iter: 200000

RPN - Stage 2:

- base_lr: 0.0001
- stepsize: 40000
- lr_policy: "step"
- gamma: 0.1
- average_loss: 100
- momentum: 0.9
- weight_decay: 0.0005
- max_iter: 400000

5.1.1 Comparison between ZFNet and VGG-16

Experiments were done using the pre-trained Zeiler and Fergus model(ZF), and the Simonyan and Zisserman model (VGG-16) to compare their detection performance for Faster R-CNN on NVIDIA GeForce GTX 980 Ti GPU.

Image 1

Type: JPEG, Size: 1920x1080

(i) Detection Using VGG16

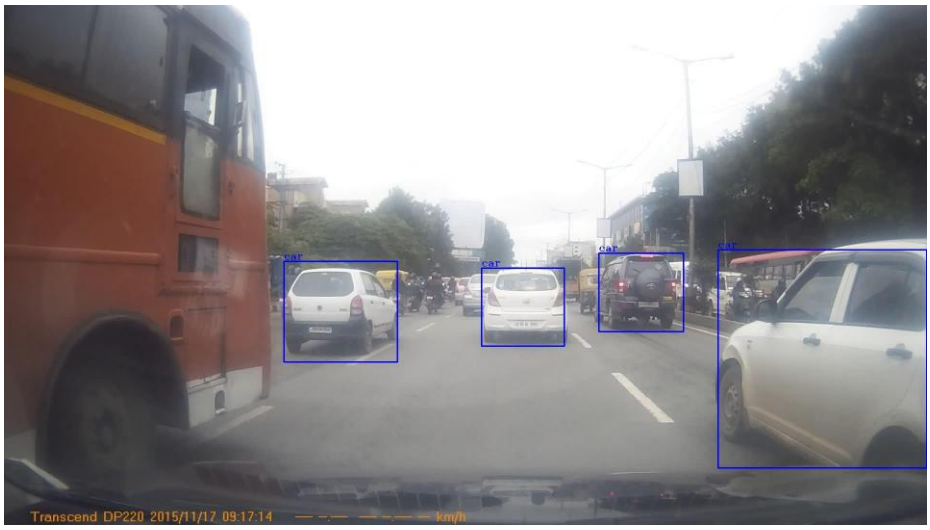


Figure 5.1: Detection took 0.245s for 300 object proposals.

(ii) Detection Using ZF

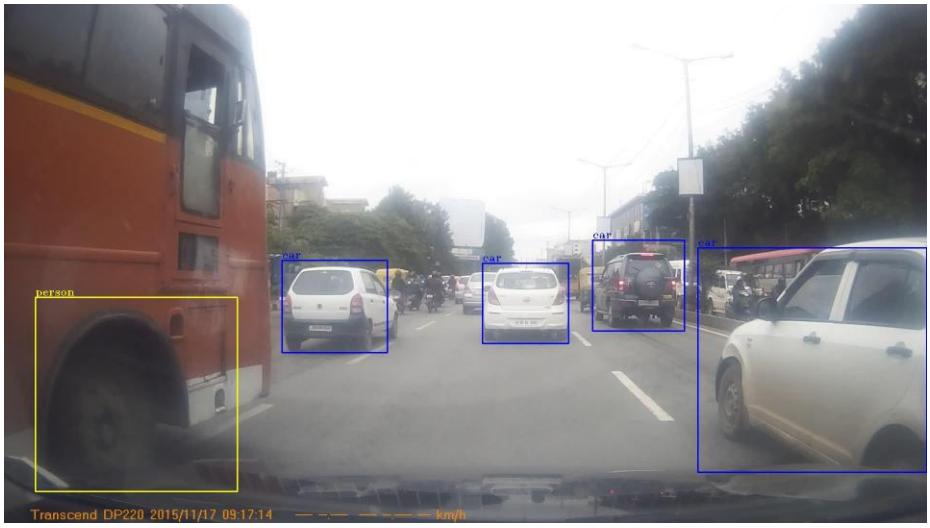


Figure 5.2: Detection took 0.101s for 300 object proposals

Image 2

Type: JPEG, Size: 1920x1080

(i) Detection Using VGG16

Figure 5.3: Detection took 0.185s for 300 object proposals.

(ii) Detection Using ZF

Figure 5.4: Detection took 0.080s for 300 object proposals

Image 3

Type: JPEG, Size: 1001x606

(i) Detection Using VGG16

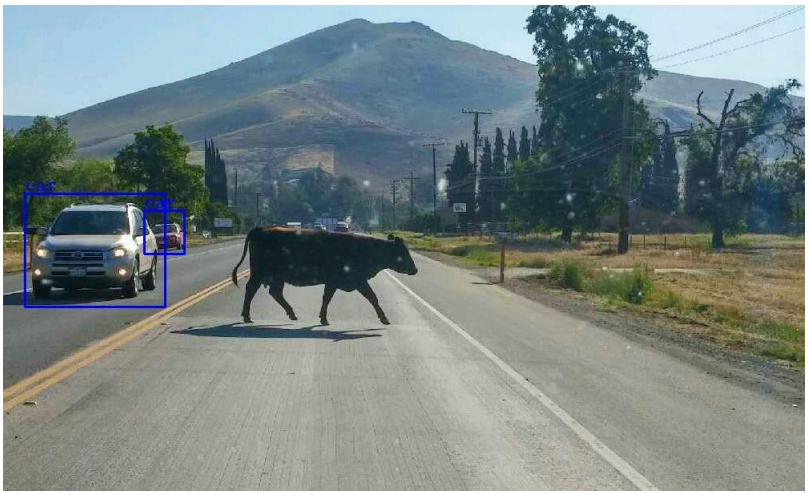


Figure 5.5: Detection took 0.198s for 300 object proposals.

(ii) Detection Using ZF

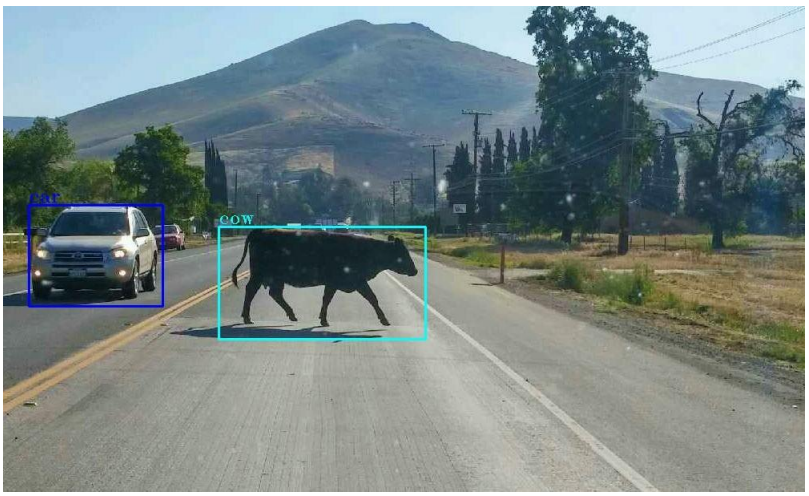
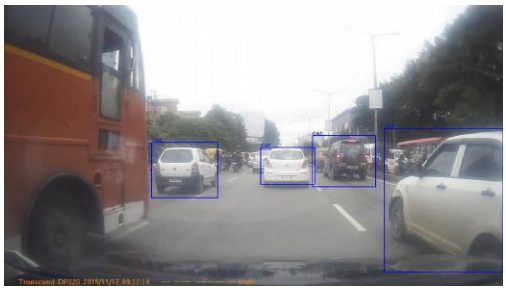


Figure 5.6: Detection took 0.065s for 300 object proposals

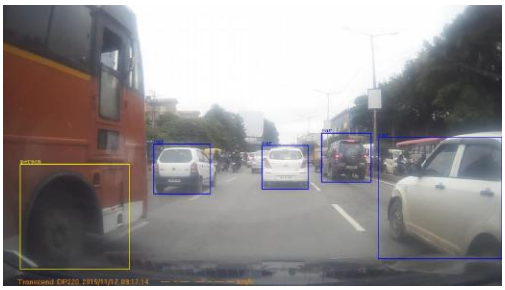
After testing the pretrained ZF model and VGG16 model for several on-road images and videos, it was seen that ZF gave considerably better performance with respect to speed than VGG16, without much difference in detection accuracy.

5.1.2 Comparison of the new 5-class detection system with the pre-trained 20-class detection system

The pre-trained ZFNet model for 20 class detection is compared with the finetuned ZF model for 5 class detection in this section.



(a) 5-class model: Detection took 0.099s



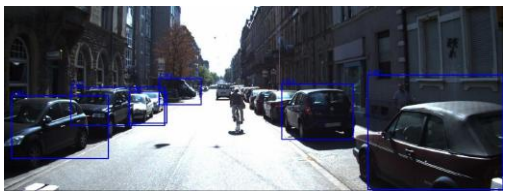
(b) 20-class model: Detection took 0.101s

Figure 5.7: **Image - 1.** Type: JPEG, Size: 1920x1080

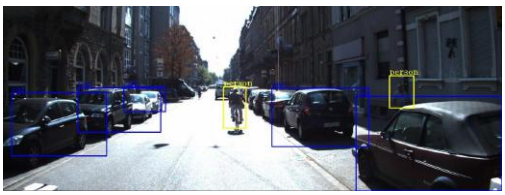
(a) 5-class model: Detection took 0.076s

(b) 20-class model: Detection took 0.080s

Figure 5.8: **Image - 2.** Type: JPEG, Size: 1920x1080

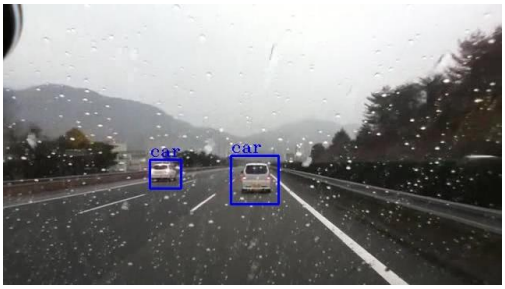


(a) 5-class model: Detection took 0.050s

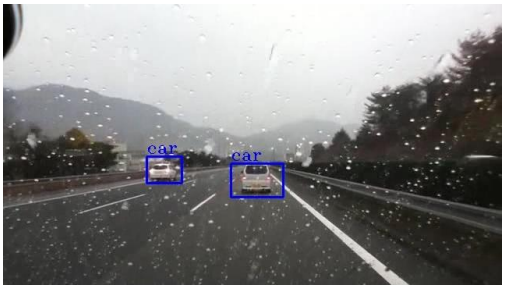


(b) 20-class model: Detection took 0.048s

Figure 5.9: **Image - 3.** Type: PNG, Size: 1392x512



(a) 5-class model: Detection took 0.041s



(b) 20-class model: Detection took 0.071s

Figure 5.10: **Image - 4.** Type: JPEG, Size: 640x360

The network trained using images from PASCAL VOC 2012 dataset, having objects of the 5 relevant classes only, gave poorer performance in terms of recall. This might be due to the lack of sufficient number of training images. It can be seen from the above result that some instances, for example objects of class 'Person' in images 2 and 3, are not detected.

5.1.3 Testing the effect of Image Resolution on the Performance

The performance of the trained ZF model was evaluated for different image resolutions on the Nvidia Jetson TX1 embedded system as well as 980Ti.

Image 1

Type: PNG, Size: 1242x375

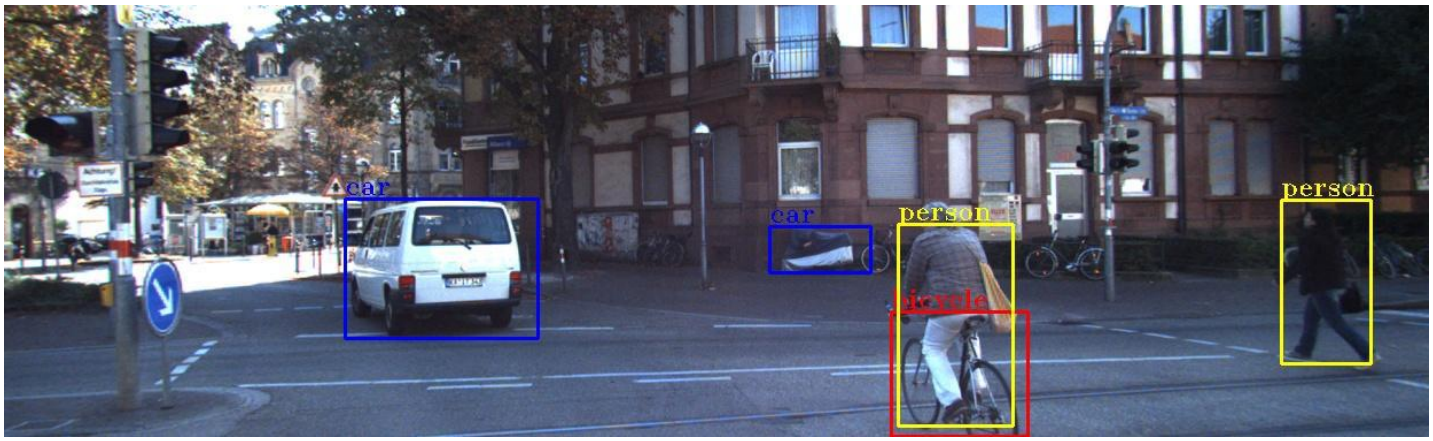


Figure 5.11: Detection took 0.976s for 300 object proposals.

Image 2

Type: PNG, Size: 932x281

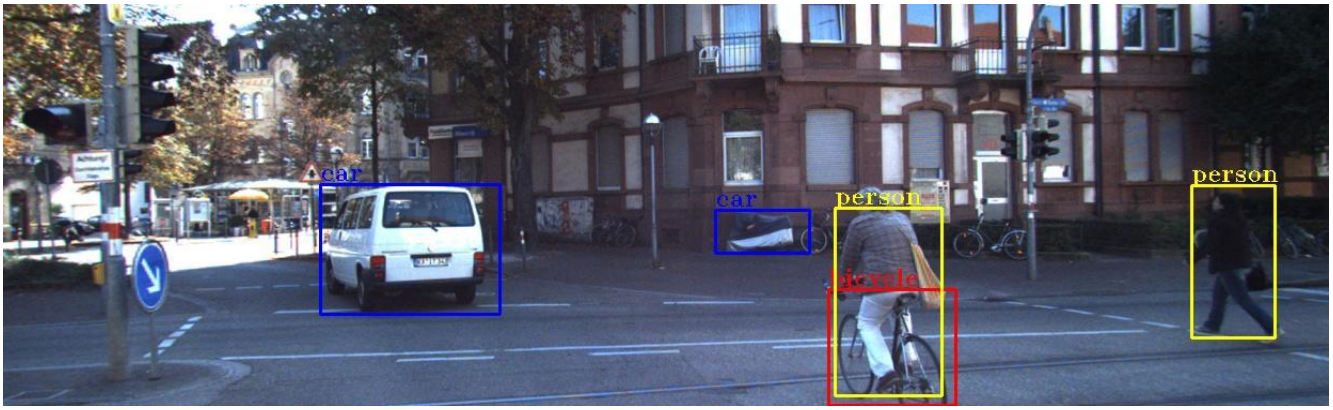


Figure 5.12: Detection took 1.045s for 300 object proposals.

Image 3

Type: PNG, Size: 663x200

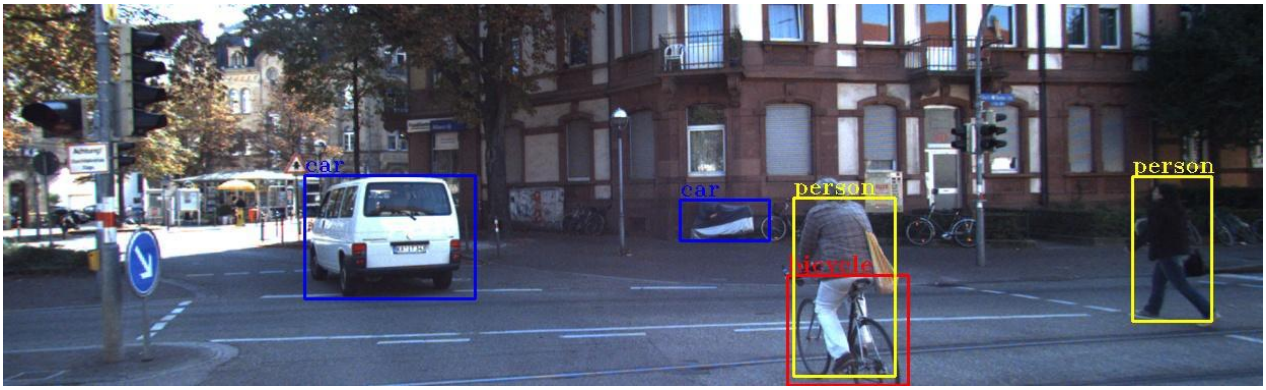


Figure 5.13: Detection took 0.892s for 300 object proposals.

Image 4

Type: PNG, Size: 621x188

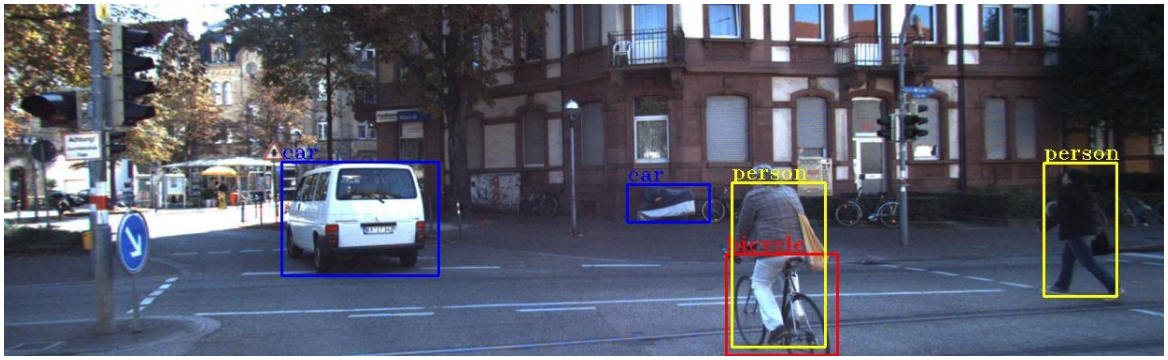


Figure 5.14: Detection took 0.858s for 300 object proposals.

Image 5

Type: PNG, Size: 311x94

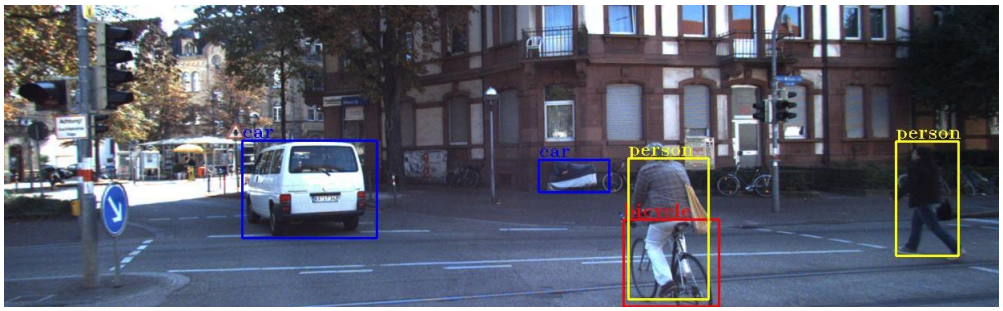


Figure 5.15: Detection took 1.074s for 300 object proposals.

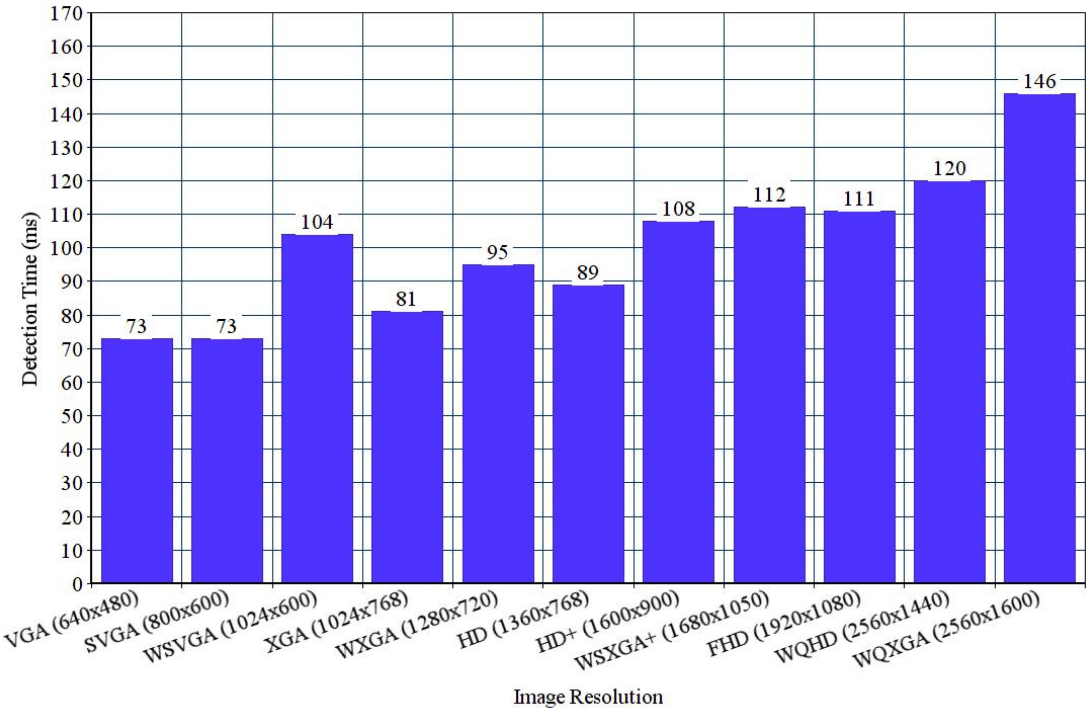
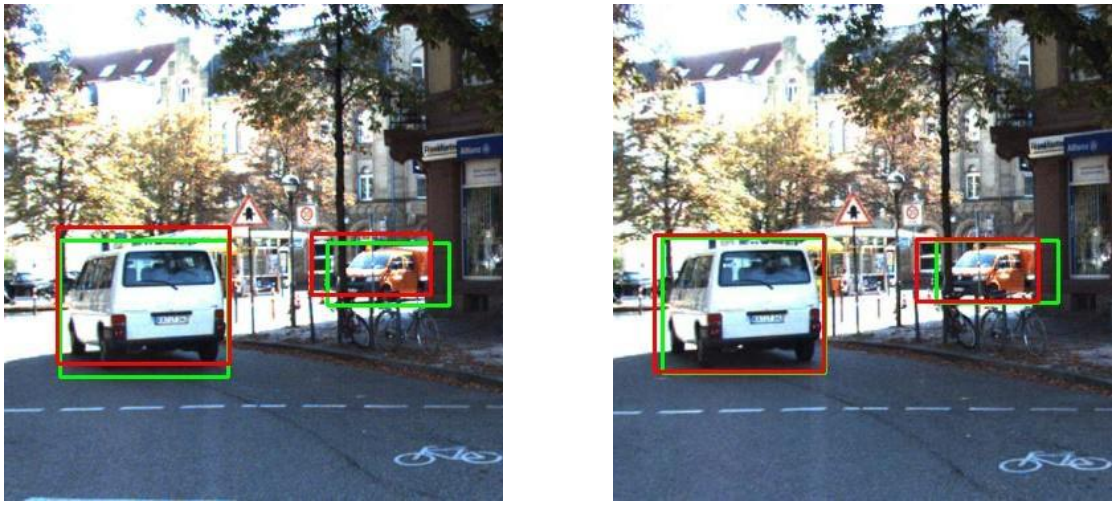


Figure 5.16: Plot of detection time of the ZFNet based network for different image resolutions on Nvidia GeForce 980Ti

This experiment showed that detection performance varies with the resolution of the input image. The best performance in terms of time and precision was observed on using the image of resolution 621x188. Another notable point in this result is that the time taken for detection on the embedded system environment is much longer than that taken for the normal GPU implementation. The detection time is in the order of seconds, which is too large to be used for real-time object detection in autonomous driving. Hence other faster and smaller object detection networks like SSD were looked into.

5.2 Object Tracking

The tracking module was tested on Kitti dataset.The annotated frames of a video from the object detection system were fed to the tracking module one after the other and the predicted position of objects in each frame were compared with the original (measured) position of the same objects. The module gave satisfactory results.



(a) Frame number 20 (b) Frame number 21

Figure 5.17: The output of object tracking module for 2 consecutive frames of a video. The detected position is indicated by the green colour box and the next position predicted by the tracking module is indicated by the red colour box.

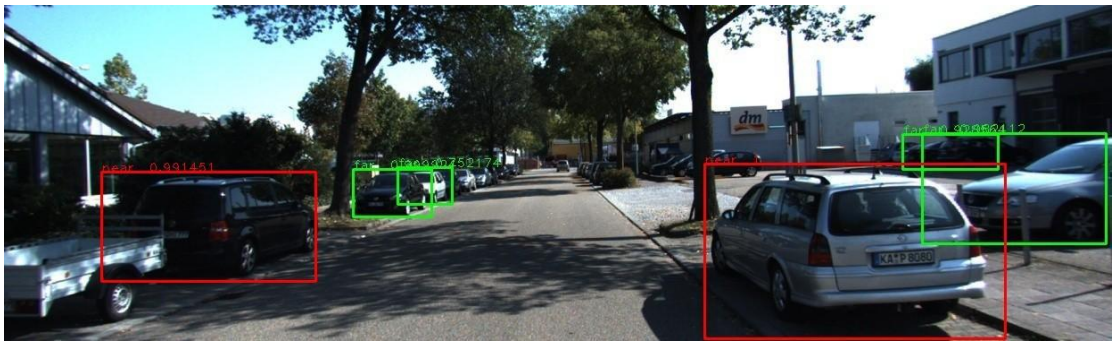
5.3 Object Detection and Ranging using SSD

5.3.1 Near/Far Object Classification

This experiment was done to test the capability of SSD network to predict the proximity of detected objects. A new dataset was prepared for this purpose from the images in KITTI dataset. Only 'car' class objects were used for this experiment. The cars in the images were manually annotated as near or far. The SSD network was trained with this dataset. The training parameters are as follows:

- base_lr: 0.001
- max_iter: 60000
- lr_policy: "step"
- gamma: 0.1
- momentum: 0.9
- weight_decay: 0.0005
- stepsize: 40000

The trained network was tested using testing images from KITTI dataset. Some of the results are given below. Red coloured bounding box refers to 'near-car' class and the green coloured one refers to 'far-car' class.



The network seems to be capable of differentiating between near and far objects well. But, in image sequences, for a car moving away from the ego-vehicle, the transition from near class to far class is abrupt and confusing. No pattern could be inferred from the confidence scores associated with the classifications either. The network is trained to classify the objects into two extreme levels only. It does not give information about how near or how far the object is.

5.3.2 The Modified SSD network for Object Detection and Range Estimation

Dataset for Training

The KITTI Vision Benchmark Suite's object dataset was used for training the SSD detection and ranging network. The dataset comprises of 7481 training images and 7518 testing images. 4 Object classes were considered while creating the annotated dataset: Car, Truck, Pedestrian and Cyclist. In the 7481 training images, a total of 38864 instances of these 4 classes were available.

Object Class	**Number of training instances**
Car	31656
Truck	1094
Pedestrian	4487
Cyclist	1627
Total:	**38864**

Table 5.1: Number of training instances of each class

The range values for the ground truth is obtained from the LiDAR data available in KITTI dataset. For these objects the minimum range is 2.88 meters and the maximum range is 103.63 meters. The average range is 29.03m. The number of training instances for different range values is given in table 5.2.

Range Value	Number of training instances
2.88 <= range < 10	4542
10 <= range < 20	9471
20 <= range < 30	8755
30 <= range < 40	6575
40 <= range < 50	4489
50 <= range < 60	2701
60 <= range < 70	1416
70 <= range < 80	838
80 <= range < 90	60
range>=90	17

Table 5.2: Number of training instances for different range values

Performance Analysis of Modified SSD

The base network of the modified ssd architecture can be obtained from any CNN architecture used for image classification. Three different modified ssd models were trained using different base networks for detection of objects belonging to the classes Car, Truck, Pedestrian, and Cyclist and the estimation of their proximity from ego-vehicle. The three base networks used were derived from:

(i) ResNet-101

(ii) VGG-16

(iii) ZFNet

Out of the 7481 training images available, 7000 were used for training and 481 images were used for testing. The training parameters for the ResNet based model were:

- base_lr: 0.0001
- display: 10
- max_iter: 120000
- lr_policy: "step"

- gamma: 0.1
- momentum: 0.9
- weight_decay: 0.0005
- stepsize: 40000

For the other two networks the training parameters were same as above except for learning rate policy. They used a 'multistep' learning rate decay policy with step values at 80000, 100000 and 120000. The learning curve for the three models is given in figure 5.18.

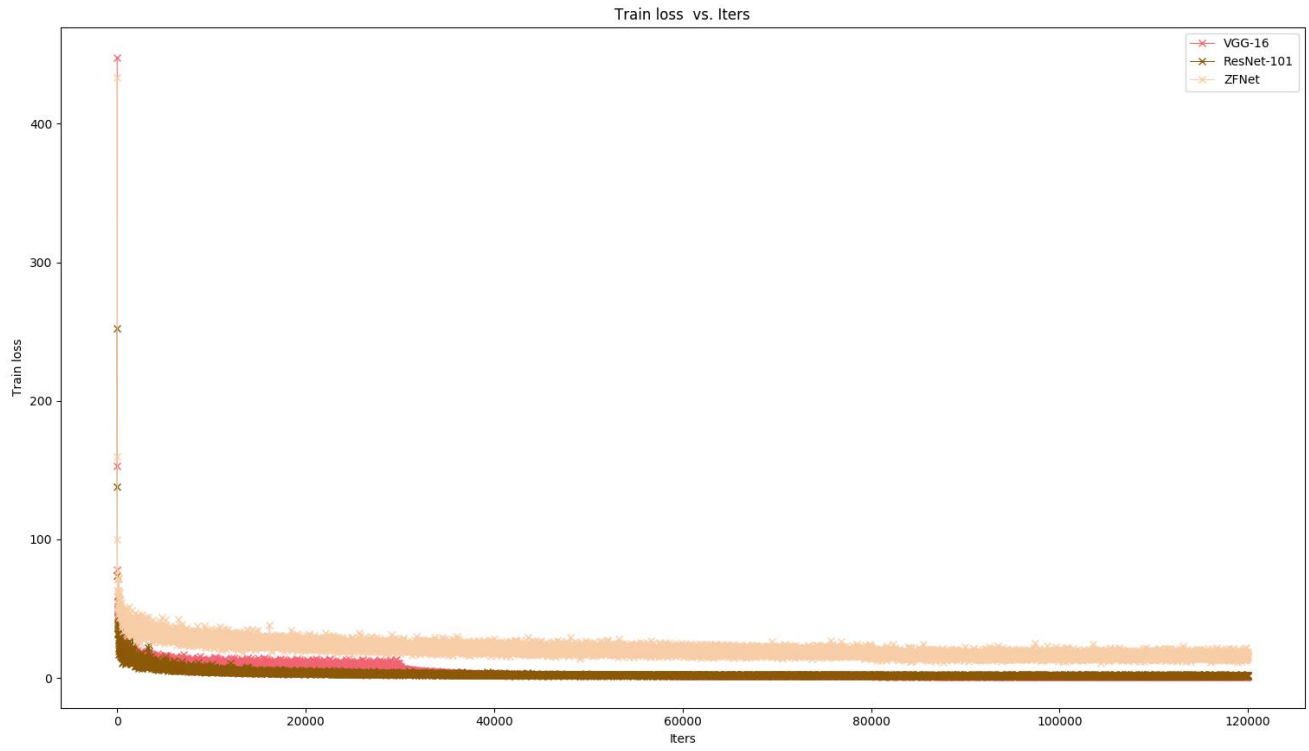


Figure 5.18: Training Loss vs Iterations for the three modified-ssd models.

It is seen that the training loss in case of the model with ZFNet base network is quite high (15.7535) even after 120000 iterations.

A few samples from the detection results of the three models are given below. Each class is marked with a different colour - red for 'car', blue for 'truck', green for 'cyclist'

and yellow for 'pedestrian. Above each bounding box, the range of the object in meters is marked. At the bottom of the bounding box, the confidence score associated with the prediction of that object is given.

Image 1

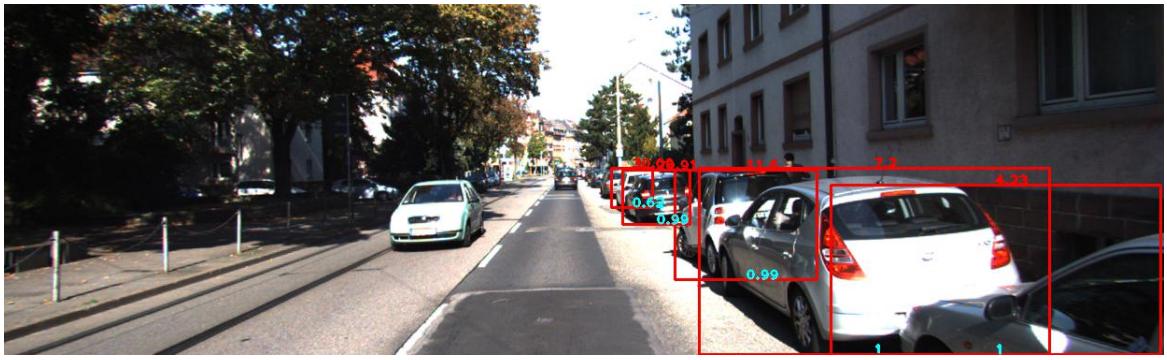


Figure 5.19: ResNet based model: The detection took 0.0623 seconds in Nvidia Titan X GPU

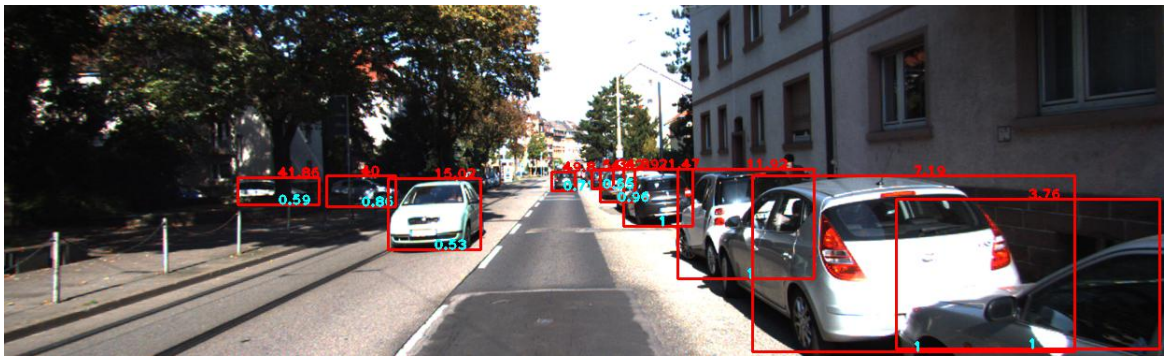


Figure 5.20: VGG based model: The detection took 0.0175 seconds in Nvidia Titan X GPU

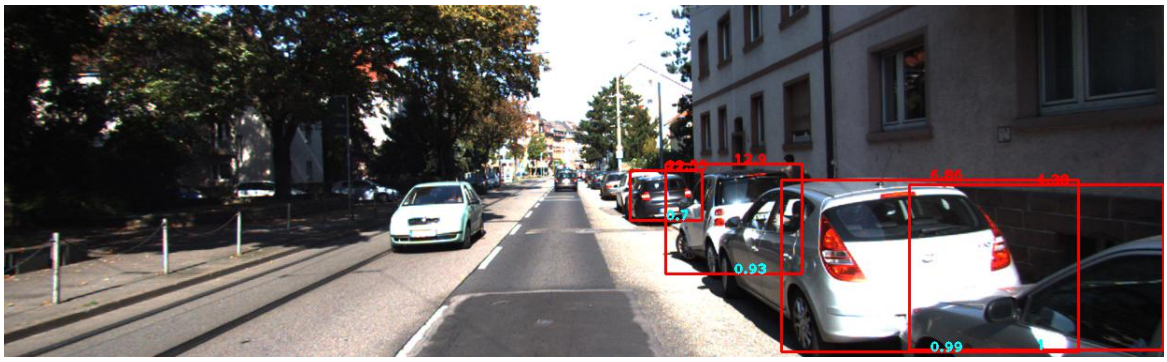


Figure 5.21: ZF based model: The detection took 0.0153 seconds in Nvidia Titan X GPU

Image 2

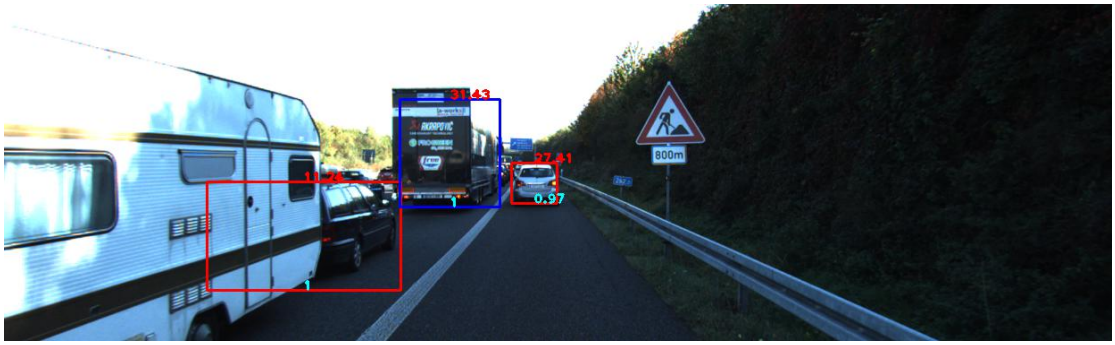


Figure 5.22: ResNet based model: The detection took 0.0606 seconds in Nvidia Titan X GPU

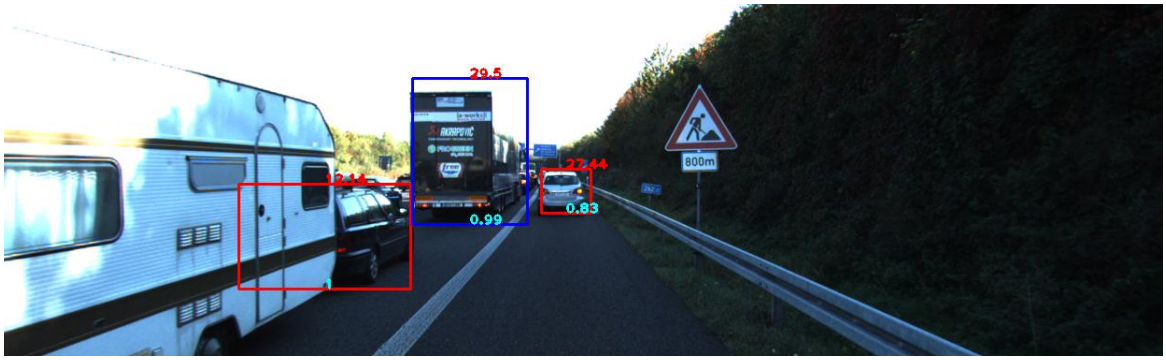


Figure 5.23: VGG based model: The detection took 0.0169 seconds in Nvidia Titan X GPU

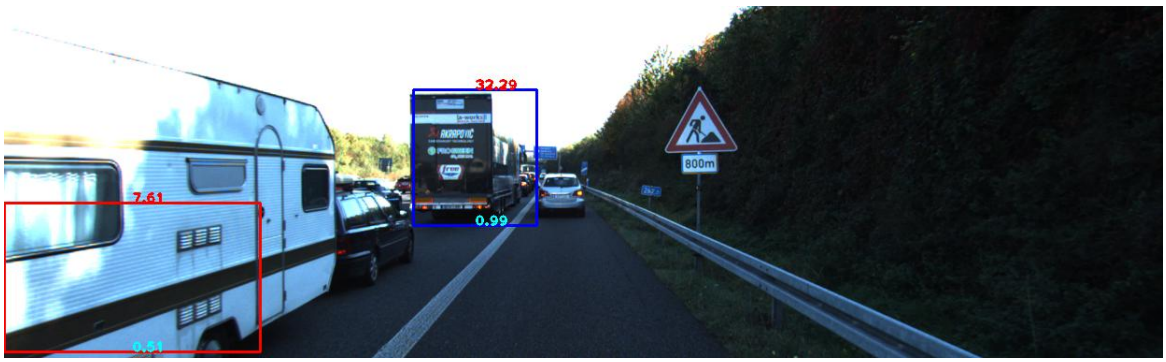


Figure 5.24: ZF based model: The detection took 0.0154 seconds in Nvidia Titan X GPU

Image 3

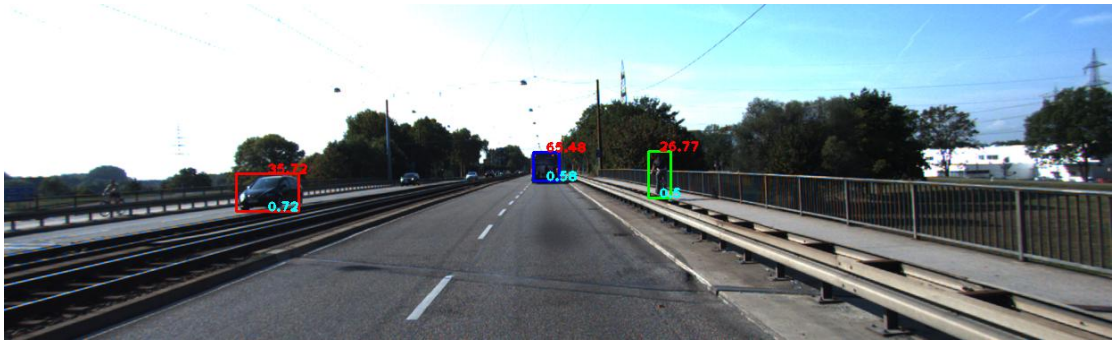


Figure 5.25: ResNet based model: The detection took 0.0611 seconds in Nvidia Titan X GPU

Figure 5.26: VGG based model: The detection took 0.0172 seconds in Nvidia Titan X GPU

Figure 5.27: ZF based model: The detection took 0.0153 seconds in Nvidia Titan X GPU

Image 4

Figure 5.28: ResNet based model: The detection took 0.0621 seconds in Nvidia Titan X GPU

Figure 5.29: VGG based model: The detection took 0.0168 seconds in Nvidia Titan X GPU

Figure 5.30: ZF based model: The detection took 0.0154 seconds in Nvidia Titan X GPU

Image 5:

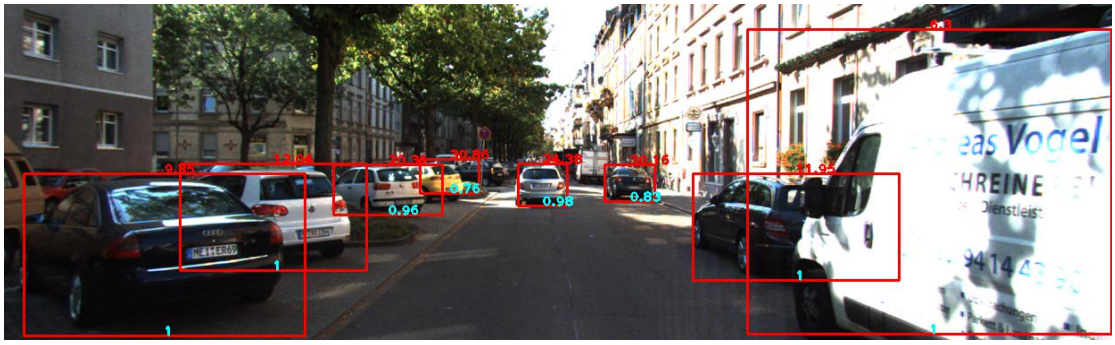


Figure 5.31: ResNet based model: The detection took 0.0621 seconds in Nvidia Titan X GPU

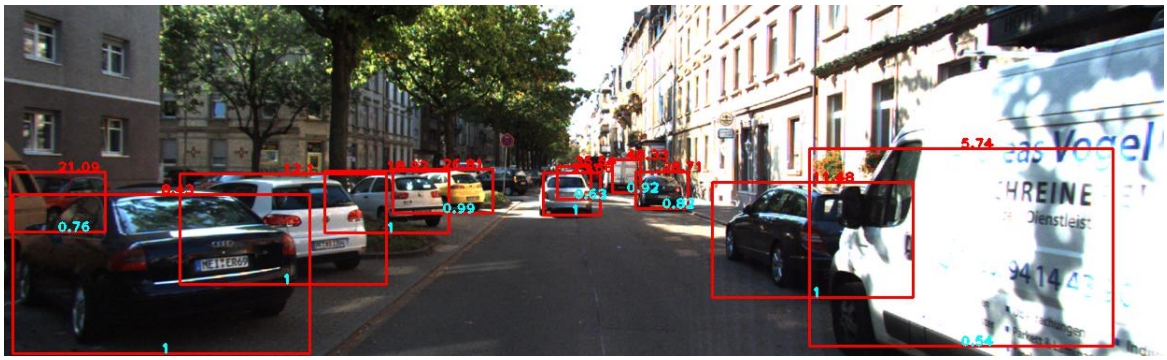


Figure 5.32: VGG based model: The detection took 0.0174 seconds in Nvidia Titan X GPU

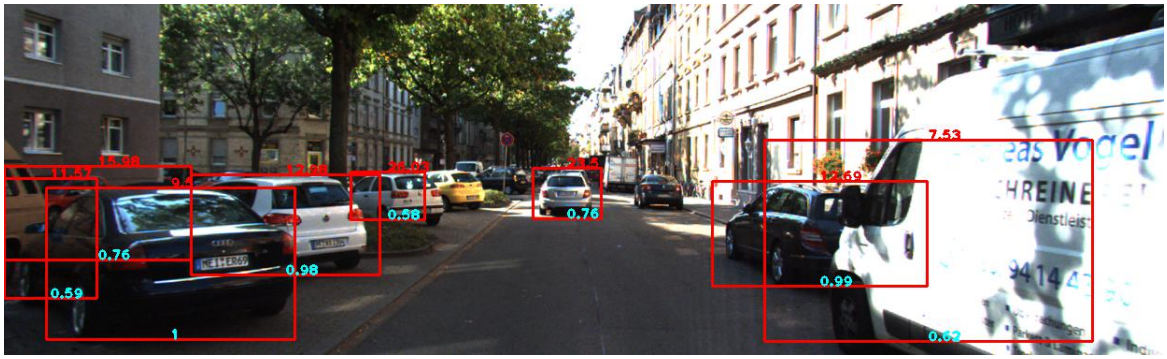


Figure 5.33: ZF based model: The detection took 0.0155 seconds in Nvidia Titan X GPU

Image 6:

Figure 5.34: ResNet based model: The detection took 0.0633 seconds in Nvidia Titan X GPU

Figure 5.35: VGG based model: The detection took 0.0168 seconds in Nvidia Titan X GPU

Figure 5.36: ZF based model: The detection took 0.0152 seconds in Nvidia Titan X GPU

Image 7:

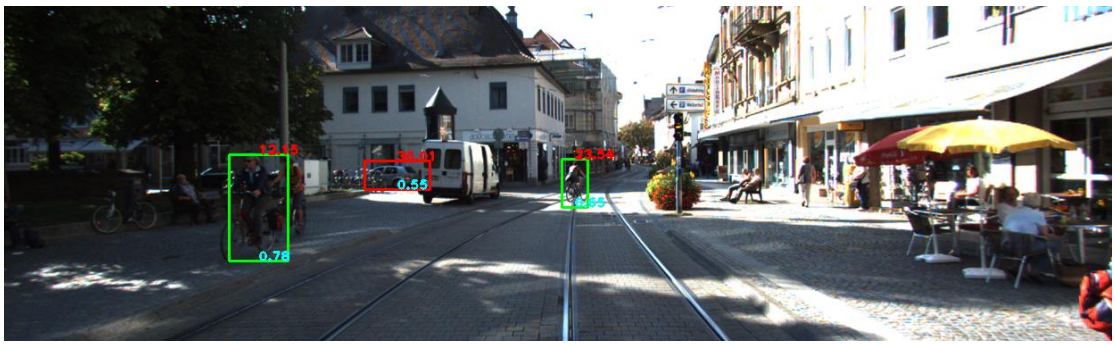


Figure 5.37: ResNet based model: The detection took 0.0626 seconds in Nvidia Titan X GPU

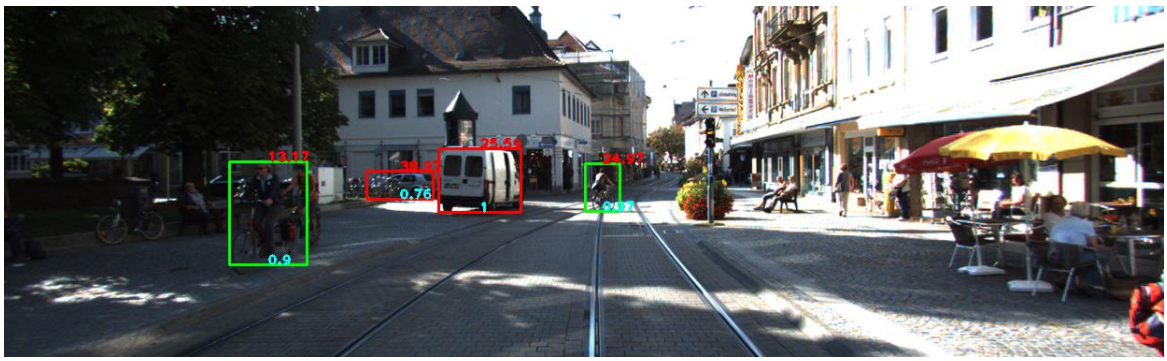


Figure 5.38: VGG based model: The detection took 0.0171 seconds in Nvidia Titan X GPU

Figure 5.39: ZF based model: The detection took 0.0150 seconds in Nvidia Titan X GPU

It is clear from these examples that ZFNet based model gives poor results compared to the other two. The advantage ZF model has in terms of detection time is negligible when considering the poor precision and recall which the model exhibits. The average detection time of the three networks on the test set are:

- ZFNet based model: 15.36ms,i.e, 65 fps.
- VGG-16 based model: 17.21ms, i.e, 58 fps.
- ResNet-101 based model: 61.56ms, i.e, 16 fps

The comparison of per class precision, recall, and F measure for the three models are given in figures 5.40, 5.41 and 5.42.

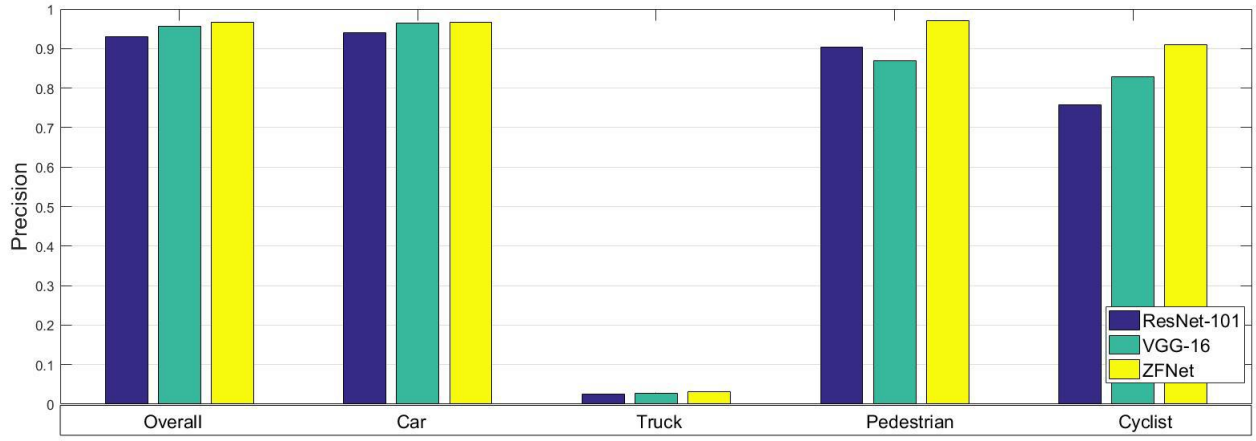


Figure 5.40: The per-class and overall precision for the three models.

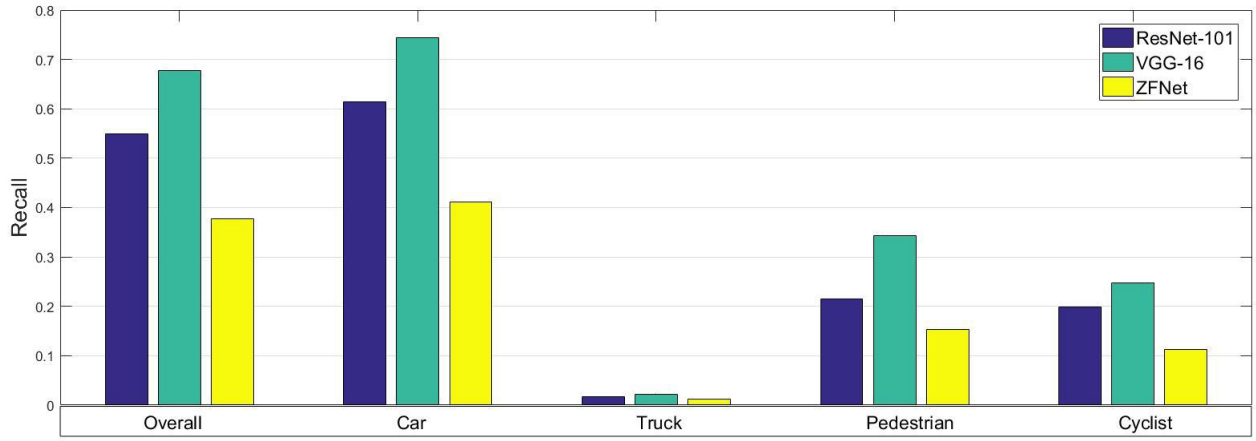


Figure 5.41: The per-class and overall recall for the three models.

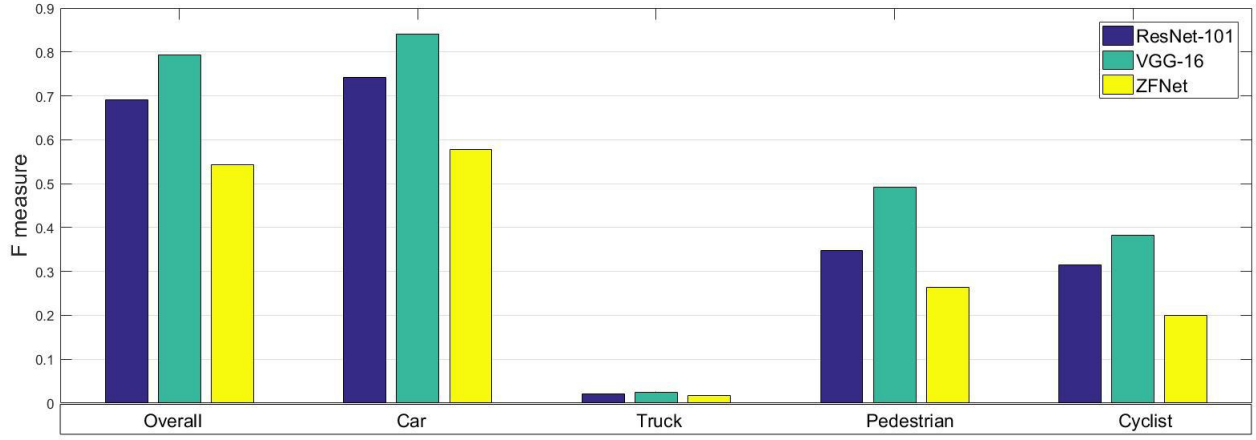


Figure 5.42: The per-class and overall F measure for the three models.

From the above plots, it can be seen that all the three networks give poor performance for the classes 'Truck', 'Pedestrian' and 'Cyclist'. This is due to the very high class imbalance in the dataset. As seen in table 5.1, the number of training instances of these three classes combined is less than a quarter of the training instances of 'Car' class. 'Truck' is the least represented class in the dataset and hence, the detection results for this class is the worst. Due to this, the mAP values of all the three models seem to be quite low.

	Car	**Cyclist**	**Pedestrian**	**Truck**	**mAP**
ResNet based model	93.94%	75.86%	90.28%	2.63%	65.68%
VGGNet based model	96.41%	82.86%	86.96%	2.80%	67.26%
ZFNet based model	96.71%	90.91%	97.14%	3.05%	71.95%

Table 5.3: Per class average precision and mean average precision given by the modified-ssd models in KITTI dataset.

The ZF based model has high precision but the recall is very low,i.e, number of missing detections is much higher than the other two models. The model with VGG base network undoubtedly gives the best performance among the three in terms of precision-recall tradeoff. The ResNet based model also gives comparable precision and recall, but the speed of detection is very less when compared to the VGG based model. Hence, the VGG-16 model gives the best detection performance overall. The Precision-Recall curve in figure 5.43 ascertains this conclusion.

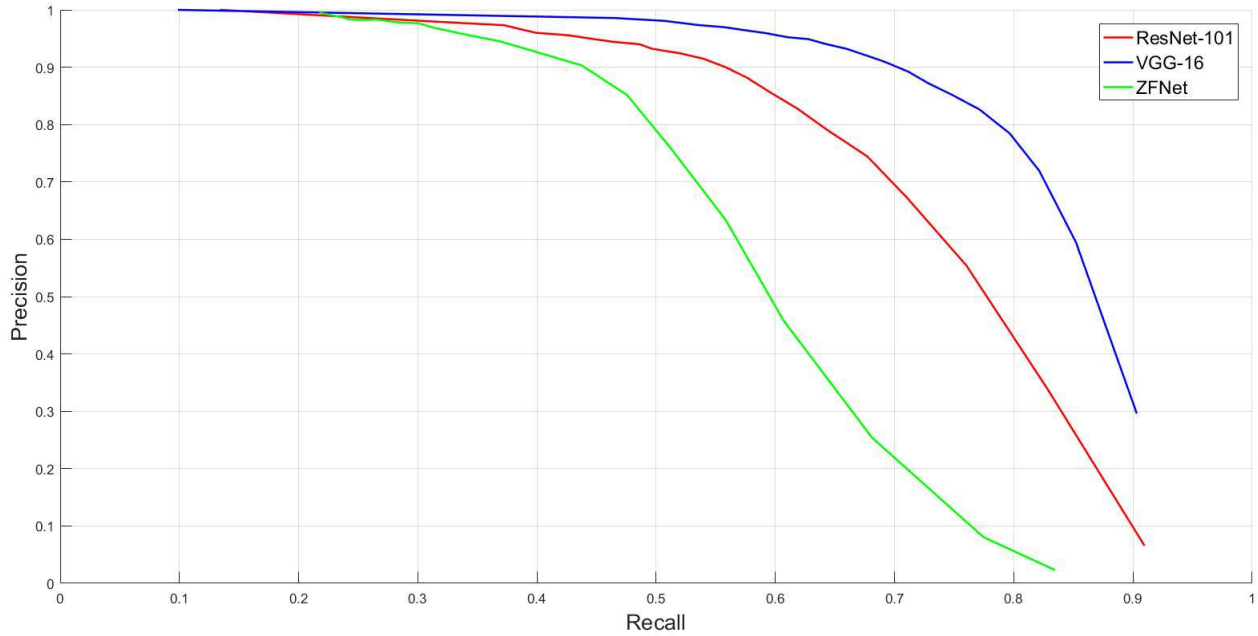


Figure 5.43: The Precision-Recall curves for the three modified ssd models

Along with detection performance, the range estimation accuracy is also an important evaluation criterion for the modified-ssd networks. Figure 5.44 shows comparison of the three models in terms of error in range estimation.

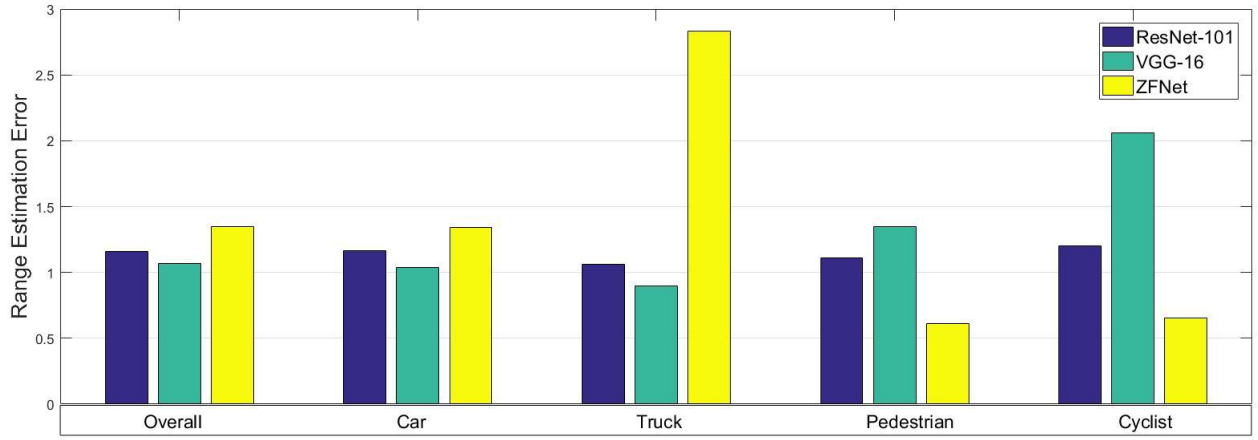


Figure 5.44: The per-class and overall range estimation error for the three modified-ssd networks.

For the classes 'Pedestrian' and 'Cyclist', ResNet model gives better range estimate. However, when the overall performance across all the classes is taken, VGG model outperforms the ResNet model again. The ZF model also gives satisfactory results for proximity estimation except for the poorly represented class, 'Truck'.

	Car	Cyclist	Pedestrian	Truck	Overall (Absolute)	Overall (Percentage)
ResNet	1.1261	0.9595	0.9783	1.0644	1.1158	5.59%
VGGNet	0.9529	1.9915	1.0595	0.8963	0.9724	4.17%
ZFNet	1.2569	0.5013	0.5228	2.8308	1.2686	7.22%

Table 5.4: Per-class and overall average error in meters for the modified-ssd models in KITTI dataset.

The models used for this analysis were obtained after 120000 training epochs. The variation in the range estimation error, precision and recall characteristics with increasing number of iterations for the ResNet based model and VGG based model are shown in figures 5.45 and 5.46. The values start to converge after 110k iterations in both cases.

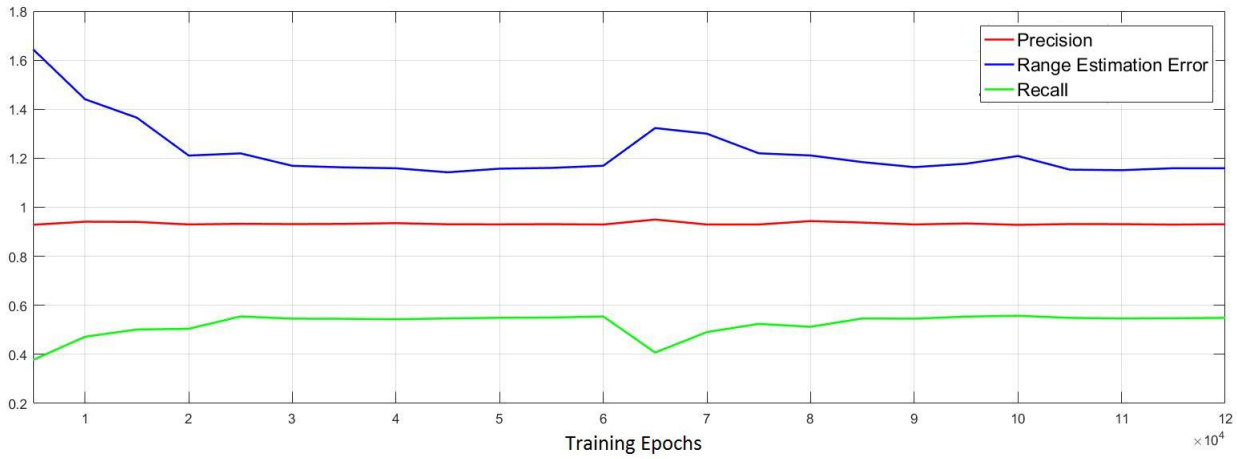


Figure 5.45: Change in Precision,Recall and Range error with increasing iterations for the ResNet model.

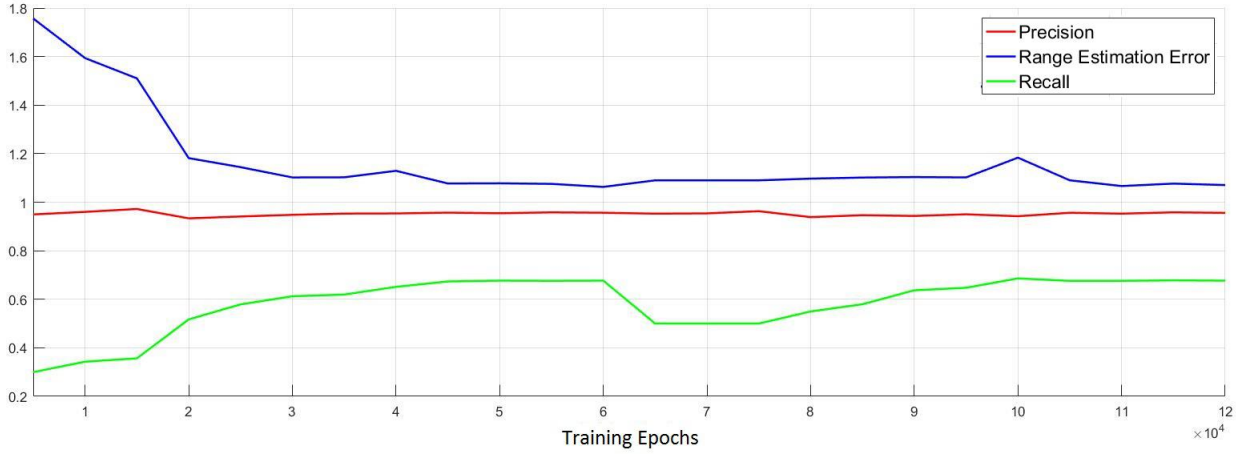


Figure 5.46: Change in Precision,Recall and Range error with increasing iterations for the VGG-16 model.

Summary

The different experiments conducted using Faster R-CNN for on-road object detection, and their results, were presented in the first section. The detection speed was the best when using ZFNet based model, but it was less than 20 fps. Hence, faster R-CNN cannot be used for real-time applications, despite being highly accurate. The results of the modified SSD network for object detection and ranging were also discussed in this chapter. The VGG-16 based model gives the best performance in terms of precision, recall and range estimation accuracy. The model was trained for the detection of 4 object classes- 'Car', 'Truck', 'Pedestrian' and 'Cyclist'. The detection speed is comparatively high (58 fps) and hence, it has the potential to be used for real-time detection and ranging. Evaluations were done on a test set of 481 images from the KITTI dataset with the confidence threshold set as 0.5. The VGG-16 based model gave an average precision of 95.6% and an mAP of 67.26%. The average error in estimation of range was 0.9724 meters.

CHAPTER 6

Conclusion and Future Work

The state-of-the-art techniques for object detection use convolutional neural networks. Popular CNN architectures for object detection - Faster R-CNN and Single Shot Multi-Box Detector(SSD) - were investigated to evaluate their potential for real-time object detection as part of the visual perception system of an autonomous vehicle. Faster R-CNN network offers high precision in object detection. However, the region proposals generation step of faster R-CNN slows down the detection process. Hence, it cannot be employed for real-time applications. SSD, on the other hand, makes use of a default set of boxes in the output space, in which the object detection will be done. This eliminates the need for region proposal generation, thereby improving the detection speed considerably, without any reduction in accuracy. Thus, SSD was chosen for the on-road real-time object detection problem. The next task in hand was to estimate the proximity of the detected objects from ego-vehicle using the monocular images of the scene. The SSD architecture was modified by adding layers for range estimation and trained for a 4 class detection and ranging problem. The model with base network derived from VGG-16 architecture gave the best performance in terms of detection precision as well as range estimation accuracy (with an error of 4.17%).

Future work in this area would include the generation of a dataset of on-road images with range information added along with bounding box annotations. The KITTI dataset was used in this work. The distribution of object classes in this dataset was highly imbalanced, which led to poor detection results for some classes. An Indian road dataset with LiDAR data and evenly distributed classes would be beneficial in this direction. Another area of improvement would be the data augmentation methods. The data augmentation techniques used for classical SSD disturbs the global attributes of the image, which would adversely affect the range estimation process. Development of data augmentation techniques which do not change the global characteristics of the images would be advantageous. The extension of this network to compute azimuth angle of the objects also would help in getting the precise location of the objects in 3-D space.

APPENDIX A

Code snippets

A.1 Object tracking demo

Compiling and running the code

To compile the .cpp file:

g++ Object_Tracking_Module.cpp ‘pkg-config –libs opencv‘ -std=c++11 -o out

To run the executable file generated:

./out

Main function

Each frame of the video is read and the detected bounding boxes are stored to the variable ’Current_bbox’. Then for each frame the following steps are done:

```
//Draw detected bounding boxes
frame = Draw_Bounding_Box (frame, Current_bbox, No_of_current_bbox);
std::ostringstream name;
name << "original" << frame_num << ".jpg";
cv::imwrite(name.str(), frame);

if (frame_num == 0)
{
        //Initialization
        Pred_bbox = Current_bbox;
        delete_track.resize(No_of_current_bbox);
        No_of_tracks = No_of_current_bbox;
}
```

```
//Draw predicted bounding boxes
frame = Draw_Bounding_Box_Pred (frame, Pred_bbox, No_of_tracks);

//Display the frame
if(!frame.empty())
{
    imshow("Output", frame);
    std::ostringstream name;
        name << "predicted" << frame_num << ".jpg";
        cv::imwrite(name.str(), frame);
    waitKey(80);
}

if (frame_num > 0)
{
        Current_bbox = Data_Association();
}

No_of_tracks = Pred_bbox.size();

for (i=0; i< No_of_tracks; i++)
{
        for (j=0; j< Num_state_var; j++)
        {
                Prev_state[j][0] = Pred_bbox[i][j];
        }

        kal_pred = Kalman_Predict(Prev_state, Prev_cov_mat);

        Assign_Matrix(kal_pred.predicted_state ,Predicted_state);
        Assign_Matrix(kal_pred.predicted_cov_mat ,Predicted_cov_mat);

        for (j =0; j< Num_state_var; j++)
        {
                State_measurement[j][0] = Current_bbox[i][j];
```

```
        }

        kal_update = Kalman_Update(Predicted_cov_mat,
                                   Predicted_state, State_measurement);

        Assign_Matrix(kal_update.updated_state,Updated_state);
        Assign_Matrix(kal_update.updated_cov_mat,Updated_cov_mat);

        for (j=0; j< Num_state_var; j++)
        {
                Pred_bbox[i][j] = Updated_state[j][0];
        }

}

Assign_Matrix(Updated_cov_mat,Prev_cov_mat);
```

Data Association Function

```
matrix Data_Association(void)
{
  int No_of_current_bbox = Current_bbox.size();
  int No_of_pred_bbox = Pred_bbox.size();
  int l = No_of_current_bbox + 1;
  int flag=0;
  int i,j,No_of_associated_bbox,unassigned,index;
  float_array unassociated_index;
  float_array associated_index;


  matrix Assigned_bbox;

  //Find nearest neighbour
  associated_index = Nearest_Neighbour_Search(No_of_current_bbox,
```

```
                    No_of_pred_bbox);

for(i=0; i<No_of_pred_bbox; i++)
  {
      if (associated_index[i] == 1)
      {
          flag++;
      }
  }

  No_of_associated_bbox = No_of_pred_bbox - flag;

  //check if track needs to be deleted
  for(i=0; i<No_of_pred_bbox; i++)
  {
      if (associated_index[i] == 1)
      {
          if (delete_track[i] > 3)
          {
              Pred_bbox.erase(Pred_bbox.begin()+i);
              delete_track.erase(delete_track.begin()+i);
              associated_index.erase(associated_index.begin()+i);
              --No_of_pred_bbox;
              --i;
          }
          else
          {
              delete_track[i] = delete_track[i]+1;
          }
      }

      else
      {
          delete_track[i] = 0;
      }
```

```
}

    //check if new track is to be initiated
    if (No_of_current_bbox > No_of_associated_bbox)
    {

    unassigned = No_of_current_bbox - No_of_associated_bbox;


    for (i=0; i<No_of_current_bbox; i++)
    {
            flag = No_of_pred_bbox;

            for (j=0; j < No_of_pred_bbox; j++)
            {
                    if (associated_index[j] == i)
                    {
                            continue;
                    }

                    flag=flag-1;
            }

            if (flag ==0)
            {
                    unassociated_index.push_back(i);
            }
    }

    for (i=0; i < unassigned; i++)
    {
            index = unassociated_index[i];

            //initiate new track
            Pred_bbox.push_back(Current_bbox[index]);
```

```
                delete_track.push_back(0);
                ++No_of_pred_bbox;
                associated_index.push_back(index);
                }
        }

        //Reorder current bounding box
        Assigned_bbox.resize(No_of_pred_bbox);

        for (i=0; i< No_of_pred_bbox; i++)
        {
                index = associated_index[i];

                if ( index == 1)
                {
                        for (j=0; j < Num_state_var; j++)
                        {
                                Assigned_bbox[i][j] = 0;
                        }
                }
                else
                {
                        Assigned_bbox[i] = Current_bbox[index];
                }
    }

    associated_index.clear();
    unassociated_index.clear();

    return Assigned_bbox;
}
```

A.2 Code snippet for training and testing the modified SSD network

```
# Create train net.
net = caffe.NetSpec()
net.data, net.label = CreateAnnotatedDataLayer(train_data,
                      batch_size=batch_size_per_device,train=True,
output_label=True, label_map_file=label_map_file,
                          transform_param=train_transform_param,
                          batch_sampler=batch_sampler)

VGGNetBody(net, from_layer='data', fully_conv=True, reduced=True,
                dilated=True,dropout=False)

AddExtraLayers(net, use_batchnorm, lr_mult=lr_mult)

mbox_layers = CreateMultiBoxHead(net, data_layer='data',
                from_layers=mbox_source_layers,
                use_batchnorm=use_batchnorm,
                min_sizes=min_sizes, max_sizes=max_sizes,
        aspect_ratios=aspect_ratios, steps=steps,
        normalizations=normalizations,
        num_classes=num_classes, share_location=share_location,
        flip=flip, clip=clip, prior_variance=prior_variance,
        kernel_size=3, pad=1, lr_mult=lr_mult)

# Create the MultiBoxLossLayer.
name = "mbox_loss"
mbox_layers.append(net.label)
net[name] = L.MultiBoxLoss(*mbox_layers,
                multibox_loss_param=multibox_loss_param,
                loss_param=loss_param,
                include=dict(phase=caffe_pb2.Phase.Value('TRAIN')),
        propagate_down=[True, True,True, False, False])
```

```
with open(train_net_file, 'w') as f:
    print('name: "{}_train"'.format(model_name), file=f)
    print(net.to_proto(), file=f)
shutil.copy(train_net_file, job_dir)

# Create test net.
net = caffe.NetSpec()
net.data, net.label = CreateAnnotatedDataLayer(test_data,
                                batch_size=test_batch_size, train=False,
                                output_label=True,
                                label_map_file=label_map_file,
                                transform_param=test_transform_param)

VGGNetBody(net, from_layer='data', fully_conv=True, reduced=True,
 dilated=True, dropout=False)

AddExtraLayers(net, use_batchnorm, lr_mult=lr_mult)

mbox_layers = CreateMultiBoxHead(net, data_layer='data',
                        from_layers=mbox_source_layers,
                        use_batchnorm=use_batchnorm, min_sizes=min_sizes,
                        max_sizes=max_sizes, aspect_ratios=aspect_ratios,
                        steps=steps, normalizations=normalizations,
                        num_classes=num_classes,
                        share_location=share_location,
                        flip=flip, clip=clip,
                        prior_variance=prior_variance,
                        kernel_size=3, pad=1, lr_mult=lr_mult)

conf_name = "mbox_conf"
if multibox_loss_param["conf_loss_type"] == P.MultiBoxLoss.SOFTMAX:
  reshape_name = "{}_reshape".format(conf_name)
  net[reshape_name] = L.Reshape(net[conf_name],
                                shape=dict(dim=[0, -1, num_classes]))
  softmax_name = "{}_softmax".format(conf_name)
```

```
    net[softmax_name] = L.Softmax(net[reshape_name], axis=2)
    flatten_name = "{}_flatten".format(conf_name)
    net[flatten_name] = L.Flatten(net[softmax_name], axis=1)
    mbox_layers[1] = net[flatten_name]
  elif multibox_loss_param["conf_loss_type"] == P.MultiBoxLoss.LOGISTIC:
    sigmoid_name = "{}_sigmoid".format(conf_name)
    net[sigmoid_name] = L.Sigmoid(net[conf_name])
    mbox_layers[1] = net[sigmoid_name]

  net.detection_out = L.DetectionOutput(*mbox_layers,
      detection_output_param=det_out_param,
      include=dict(phase=caffe_pb2.Phase.Value('TEST')))
  net.detection_eval = L.DetectionEvaluate(net.detection_out, net.label,
      detection_evaluate_param=det_eval_param,
      include=dict(phase=caffe_pb2.Phase.Value('TEST')))

  with open(test_net_file, 'w') as f:
      print('name: "{}_test"'.format(model_name), file=f)
      print(net.to_proto(), file=f)
  shutil.copy(test_net_file, job_dir)

  # Create deploy net.
  # Remove the first and last layer from test net.
  deploy_net = net
  with open(deploy_net_file, 'w') as f:
      net_param = deploy_net.to_proto()
      # Remove the first (AnnotatedData) and last (DetectionEvaluate) layer
      del net_param.layer[0]
      del net_param.layer[-1]
      net_param.name = '{}_deploy'.format(model_name)
      net_param.input.extend(['data'])
      net_param.input_shape.extend([
          caffe_pb2.BlobShape(dim=[1, 3, resize_height, resize_width])])
      print(net_param, file=f)
  shutil.copy(deploy_net_file, job_dir)
```

```
# Create solver.
solver = caffe_pb2.SolverParameter(
        train_net=train_net_file,
        test_net=[test_net_file],
        snapshot_prefix=snapshot_prefix,
        **solver_param)

with open(solver_file, 'w') as f:
    print(solver, file=f)
shutil.copy(solver_file, job_dir)

max_iter = 0
# Find most recent snapshot.
for file in os.listdir(snapshot_dir):
  if file.endswith(".solverstate"):
    basename = os.path.splitext(file)[0]
    iter = int(basename.split("{}_iter_".format(model_name))[1])
    if iter > max_iter:
      max_iter = iter

train_src_param = '--weights="{}" \\\n'.format(pretrain_model)
if resume_training:
  if max_iter > 0:
    train_src_param = '--snapshot="{}_iter_{}.solverstate"\n'.
                      format(snapshot_prefix, max_iter)

if remove_old_models:
  # Remove any snapshots smaller than max_iter.
  for file in os.listdir(snapshot_dir):
    if file.endswith(".solverstate"):
      basename = os.path.splitext(file)[0]
      iter = int(basename.split("{}_iter_".format(model_name))[1])
      if max_iter > iter:
        os.remove("{}/{}".format(snapshot_dir, file))
```

```
    if file.endswith(".caffemodel"):
      basename = os.path.splitext(file)[0]
      iter = int(basename.split("{}_iter_".format(model_name))[1])
      if max_iter > iter:
        os.remove("{}/{}".format(snapshot_dir, file))

# Create job file.
with open(job_file, 'w') as f:
  f.write('cd␣{}\n'.format(caffe_root))
  f.write('./build/tools/caffe␣train␣\\\n')
  f.write('--solver="{}"␣\\\n'.format(solver_file))
  f.write(train_src_param)
  if solver_param['solver_mode'] == P.Solver.GPU:
    f.write('--gpu␣{}␣2>&1␣|␣tee␣{}/{}.log\n'.format(gpus,
        job_dir, model_name))
  else:
    f.write('2>&1␣|␣tee␣{}/{}.log\n'.format(job_dir, model_name))

# Copy the python script to job_dir.
py_file = os.path.abspath(__file__)
shutil.copy(py_file, job_dir)

# Run the job.
os.chmod(job_file, stat.S_IRWXU)
if run_soon:
  subprocess.call(job_file, shell=True)
  }
```

Running the programs for training and testing

For training the network with VGG base network, from $CAFFE_ROOT run:

python/examples/ssd/ssd_python.py

For ResNet-101 base network:

python/examples/ssd/ssd_python_resnet.py

For ZF base network:
python/examples/ssd/ssd_python_zf.py

For testing a trained model, the ssd_detect.cpp program can be used.
Usage example:
From $CAFFE_ROOT run: *./build/examples/ssd/ssd_detect.bin models/ZFNet/SSD_300x300/ deploy.prototxt models/ZFNet/SSD_300x300/ ZF_iter_120000.caffemodel /data/Kitti_for_Ranging/ test_full_link.txt –out_file ZF_120k.txt*

The first argument is the location of deploy.prototxt file of the trained model and the second argument is the .caffemodel of the same. The third arg is the location to a text file containing the locations of the images to be tested. Eg:

/home/guest/data/Kitti_for_Ranging/data_object_image_2/training/image_2/000296.png
/home/guest/data/Kitti_for_Ranging/data_object_image_2/training/image_2/007358.png
/home/guest/data/Kitti_for_Ranging/data_object_image_2/training/image_2/004549.png
/home/guest/data/Kitti_for_Ranging/data_object_image_2/training/image_2/000751.png

The location of output file can be specified with –out_file flag.

REFERENCES

[1] Kalman filter. https://en.wikipedia.org/wiki/Kalman_filter.

[2] Ross Girshick. Fast r-cnn. In *International Conference on Computer Vision (ICCV)*, 2015.

[3] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition*, 2014.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Elizabeth Gurdus. The basic elements to building self driving cars. http://www.cnbc.com/2017/01/05/the-3-basic-elements-to-building-self-driving-cars.html.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[7] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[8] Andrej Karpathy. Cs231n: Convolutional neural networks for visual recognition. http://cs231n.github.io/.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[10] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[11] Wei Liu, Andrew Rabinovich, and Alexander Berg. Parsenet: Looking wider to see better. *arXiv preprint arXiv:1506.04579*, 2015.

[12] Tom M. Mitchell. *Machine Learning*. WCB McGraw-Hill, 1997.

[13] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[16] Jianxin Wu. Introduction to convolutional neural networks. `https://cs.nju.edu.cn/wujx/paper/CNN.pdfl`.

[17] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.