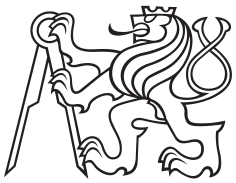
Bachelor thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Detecting Objects for Autonomous System Verification

Otakar Jašek

Supervisor: doc. Ing. Tomáš Svoboda, PhD.
Field of study: Open Informatics
Subfield: Informatics and Computer Science
May 2016

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Otakar J a š e k

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Detecting Objects for Autonomous System Verification

Guidelines:

Verification of autonomous systems (cars/robots) requires lengthy testing. Virtually, all possible situations must be tested before an algorithm/system is allowed a for consumer use. The goal of this project is develop a sofware package that supports automated annotation of verification data. The sought software system can make use of pre-trained algorithms however, it should be able to re-learn on specific data. The starting point shall be the most recent developments in the field [1,2,3]. One of the open challenges is how to adapt or re-train algorithms on rare data. The implementation should be modular and allow integration into various frameworks, like ROS (ros.org) for robotic applications. The software will be evaluated on standard datasets as well as our data.

Bibliography/Sources:

[1] Girshick, Ross and Donahue, Jeff and Darrell, Trevor and Malik, Jitendra. Region-based Convolutional Networks for Accurate Object Detection and Segmentation. In IEEE Transactions on Pattern Analysis and Machine Intelligence, 2015
[2] Ren, Shaoqing and He, Kaiming and Girshick, Ross and Sun, Jian. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In Neural Information Processing Systems (NIPS), 2015
[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. http://tensorflow.org

Bachelor Project Supervisor: doc. Ing. Tomáš Svoboda, Ph.D.

Valid until: the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, December 17, 2015

Acknowledgements

I would like to thank my supervisor, Tomáš Svoboda, for countless advices and steering me in the right direction while working on my thesis. Also I would like to thank my family for never ending support while working on this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Signature:

Prague, May 27, 2016

Abstract

In this thesis we created a framework for easy evaluation and training of Faster R-CNN type of networks. We fine-tuned VGG16 and ZFNet networks on our internal Victims dataset as well as standard KITTI dataset. We later showed that VGG16 architecture is far more suitable for fine-tuning on data from slightly different training and target domains. This framework can later serve as a baseline for further improvements in the field.

Keywords: Faster R-CNN, Convolutional neural networks, fine-tuning of neural networks, detection and recognition of objects

Supervisor: doc. Ing. Tomáš Svoboda, PhD.

Abstrakt

V této práci jsme vytvořili framework pro jednoduché vyhodnocení a trénování konvolučních neuronových sítí typu Faster R-CNN. Přetrénovali jsme sítě architektur VGG16 a ZFNet jak na našich interních datech z datasetu obětí, tak i na standartním KITTI datasetu. Dále jsme ukázali, že architektura VGG16 je o mnoho vhodnější k přetrénování pomocí dat, které pocházejí z málo rozdílných trénovacích a testovacích domén. Vytvořený framework může do budoucna sloužit jako výchozí bod pro budoucí vylepšení architektur tohoto typu.

Klíčová slova: Faster R-CNN, Konvoluční neuronové sítě, přetrénování neuronových sítí, detekce a rozpoznání objektu

Překlad názvu: Detekce objektů pro verifikaci autonomních systémů

Contents

Figures

Tables

Chapter 1

Theory and recent work

1.1 Recent work

Most of the recent work in object detection and recognition revolves around the need to speed up detection phase of recognition pipeline. One of the most advancement in this area was made by introducing R-CNN type of networks, in three iterations – R-CNN [1], Fast R-CNN [2] and Faster R-CNN [3]. R in R-CNN stands for Regions to illustrate combination of region proposals with convolutional neural network. The main progress made in last iteration of R-CNN (Faster) was introduction of RPN units. RPN unit is a Region Proposal Network which takes care of proposing regions of interest for convolutional network while sharing the weights and computation with convolutional network used for recognition. Such RPN can be made to output limited number of proposals in scene and original paper reported that 300 was the most suitable number – therefore we were using only 300 as well.

Another difficulty which we were facing was domain shift. Since our goal was to have an easily trainable system, which could be potentially used for different training and testing domain (namely generated data used as training domain with real data used as testing domain), it was needed to adopt some techniques necessary to overcome such domain shift. However, due to time restrictions we were only able to use simple re-training, which was recently shown by Rozantsev [4] might be insufficient when compared to more recent techniques. Rozantsev was using simple means of having two concurrent networks running next to each other while sharing the weights between some layers and allowing slight transformation between other layers which rendered to be superior to other techniques previously used to overcome domain shift.

Another important question was which of the convolutional network frameworks to use. While Caffe [5] seemed to be most reasonable choice mostly because Faster R-CNN was implemented using Caffe, we also considered using other frameworks such as TensorFlow [6] by Google or Theano [7]. However, when using Theano, we would most likely have to write significantly larger amount of code though with more control over it since Theano is only compiler of mathematic expressions. Ultimately we chose Caffe for two reasons – it still has a lot more support in scientific community than TensorFlow and implementation of Faster R-CNN work was built around it.

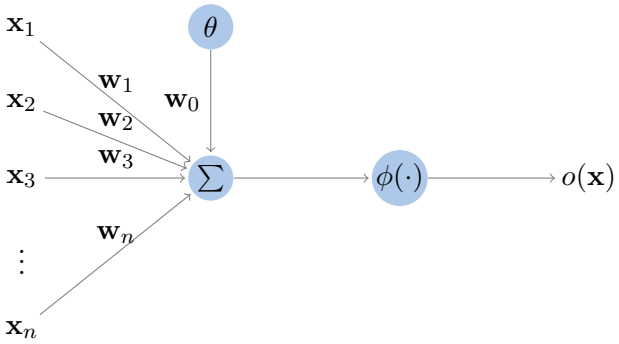


Figure 1.1: Structure of artificial neuron

1.2 Theory

1.2.1 Convolutional neural network

The problem of object detection and recognition is one of the oldest in the field of computer vision. The object detection deals with finding the objects of interest in a scene where the multiple objects can be present at once, while the object recognition tries to classify either objects extracted by object detection tools or the whole image. The object detection problem is inherently much more difficult than object recognition since objects can be present in an image in vast amount of various locations, scales or different aspect ratios. One of the approaches used for object recognition is using convolutional neural networks.

The reasoning behind this is rather simple. Since artificial neurons are trying to simulate actions of real neurons in brain, using a convolutional network is next logical step because it is believed processing images in our brain relies on connecting visual information from close surroundings together, much like convolutional filters do.

The artificial neuron itself (shown on figure 1.1) is described by the equation $o(\mathbf{x}) = \phi(\mathbf{x}^\top \mathbf{w} + \mathbf{w}_0\theta)$, where $\phi(\cdot)$ is a usually non-linear activation function, $\mathbf{w}$ is a weight vector associated with a given neuron, θ is a so called bias of a neuron and $\mathbf{x}$ is the input vector fed into neuron. If we start connecting these neurons next to each other and then into layers, we get fully-connected neural networks (shown on figure 1.2).

Convolutional layer differs from fully-connected (fc) layer in a way that it is not analyzing whole image by each neuron, but rather performing mathematical operation of convolution of an image – hence the name. In many implementations of convolutional networks, 2D input image is actually seen as 3D image where the depth of such image is only one pixel for grayscale images or 3 for colored images, each depth layer coding one of the RGB's channels. When performing convolution, it then does not perform 2D convolution, but 3D convolution to be able to further process outputs of previous convolutional layers with depth higher than 1. Such outputs are also called feature maps.

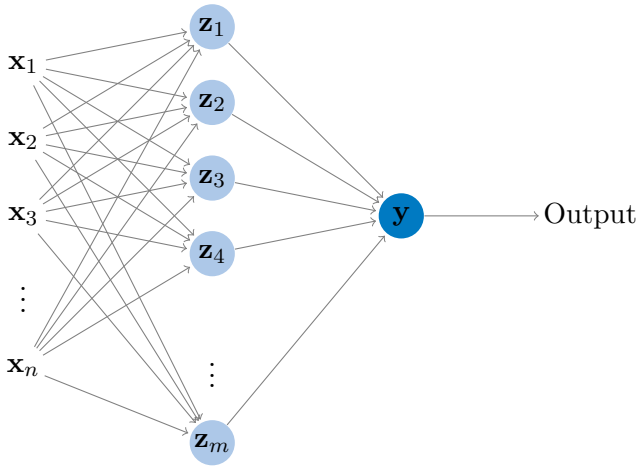


Figure 1.2: Structure of a three-layered fully-connected neural network

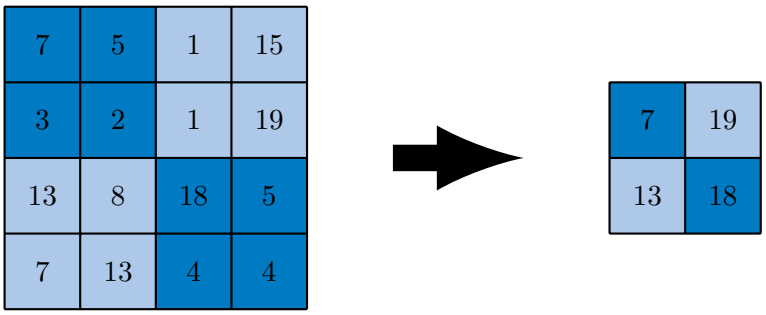


Figure 1.3: Example of max pooling operation with window 2×2 and stride 2

Convolutional layer usually consists of several different filters each performing its own convolution and those are then stacked on top of each other to create a 3-dimensional feature map where depth is determined by number of filters.

Another type of layer used in modern convolutional networks is pooling layer ensuring dimensional reduction of propagating feature maps. Main idea of the pooling layer is to take a square region from a feature map and produce only one number for each of those regions. There are many different strategies, but most common is max pooling which simply selects maximal value from each region due to assumption that high response correlates to finding a useful feature. Example of max pooling operation can be seen on figure 1.3

Since convolutional and pooling layers are essentially filters, one can as well define stride in these layers. While for most architectures, stride of pooling layers is usually 1 (in both directions x and y), stride for convolutional layers is quite often higher than 1.

Some authors tend to use ReLU units as well, which stands for Rectified Linear Units. Such units act as an activation function which simply for each value in the feature map applies function $f(x) = \max(0, x)$. One might think that such unit have no importance, however it was shown that ReLU unit is important in increasing capability of network to express non-linearity which

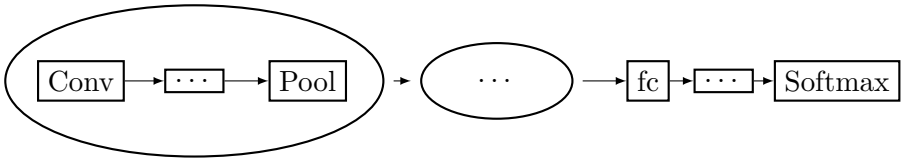


Figure 1.4: A general architecture of a convolutional neural network. The rectangles represent layers, ... represent optional repeating of previous layer. The ellipse is used to later illustrate possible repeating of a block of layers encapsulated in the ellipse

is desired.

Often the last layer used is so called softmax layer which just applies the softmax function to the output vector created by a preceding layer. Softmax function squeezes the vector in such a way that all the values in the vector are in range (0;1) and all values in the vector sum up to 1. Another name for the softmax function is an exponential normalization. The definition of the softmax function on a K-dimensional vector $\mathbf{x}$ can be seen on equation 1.1.

$$\sigma(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{k=1}^{K} e^{\mathbf{x}_i}} \text{ for } i = 1, \ldots, K \tag{1.1}$$

A general architecture of a convolutional neural network consists of series of repeating one or more convolutional layers followed by the pooling layer. After this repetitive series the rest of the network is usually fully-connected while the last layer is the softmax layer. Such general architecture can be seen on figure 1.4.

1.2.2 Architecture of used networks

The networks used in our experiments were VGG16 [8] and ZFNet [9]. The ZFNet network architecture is much smaller than the VGG16 network architecture – the ZFNet architecture uses 5 convolutional layers with most of them directly followed by pooling layer while the VGG16 uses 13 convolutional layers but only 5 pooling layers altogether. Another important difference is that tha VGG16 network uses only convolutional layers with dimensions $3 \times 3 \times n$ (with n being variable according to previous layer output) and always having stride 1, while ZFNet uses convolutional layers with various dimensions and strides. The intermediate vector after the last pooling layer has size of 25088 for the VGG16 architecture while only 9216 for the ZFNet architecture.

The ZFNet is 8-layer architecture, where the original authors consider convolutional layer followed by pooling layer to count as 1 layer. The input layer expects images of dimension $224 \times 224 \times 3$ (where 3 stands for the RGB channels) and if the image we want to analyze does not fit this dimension, we simply scale it without keeping aspect ratio in order to fit. The first layer consists of 96 convolutional filters of size $7 \times 7 \times 3$ that have stride of 2 in x and y directions and stride 1 in z direction. This feature map of dimension $110 \times 110 \times 96$ is then max-pooled by pooling window of size 3×3 and stride

2 to have output of the first layer of dimension $55 \times 55 \times 96$. The second layer consists of 256 filters of dimension $5 \times 5 \times 96$ while using stride 2 in x and y directions (and stride 1 in z direction). The feature map of dimension 26×26 is then again max-pooled by pooling window of size 3×3 and stride 2 in order to create the second layer's output of dimension $13 \times 13 \times 256$. The third layer consists of 384 convolutional filters of dimension $3 \times 3 \times 256$ with stride 1 in all directions and the fourth layer is essentially the same with 384 filters of size $3 \times 3 \times 384$ with stride 1 in all directions, therefore the feature map after fourth layer has dimension $13 \times 13 \times 384$. The fifth layer consists of 256 convolutional filters of dimension $3 \times 3 \times 384$ with stride 1 in all directions. The feature map is then max-pooled by pooling window of size 3×3 with stride 2 resulting in the feature map of dimension $6 \times 6 \times 256$. This feature map is then represented as a vector of size 9216 that is then fed into layers 6 and 7, each consisting of 4096 fully-connected neurons. The last layer is then C-way softmax function which essentially behaves as a fully connected layer with C neurons, where C is the number of classes to classify, directly followed by classical softmax function.

The VGG16 network architecture is one of the architectures presented in Simonyan's and Zisserman's paper introducing many experiments of different ConvNet network architectures. Variant 16 consists of 16 weighted layers, which was the second largest architecture in their series of experiments.

The architecture expects once again images of dimension $224 \times 224 \times 3$ and we use the scaling to achieve this dimension. The first two layers consist of 64 filters of dimension $3 \times 3 \times 3$ and $3 \times 3 \times 64$ respectively with stride 1 in all directions. After these two layers there is a max pooling layer with a window of size 2×2 and stride 2, such that after this layer the feature map has dimension $112 \times 112 \times 64$. Another two layers consist of 128 convolution filters of dimension $3 \times 3 \times 64$ and $3 \times 3 \times 128$ respectively with stride 1 in all directions and afterwards once again max pooling layer with a window of size 2×2 and stride 2 is applied such that the intermediate feature map has dimension $56 \times 56 \times 128$. Another 3 layers consist of 256 filters of dimension $3 \times 3 \times 128$ for the first layer and $3 \times 3 \times 256$ for the other two layers, all of them having stride 1 in all directions. Once again max pooling operation is present after this triple with a window of 2×2 and stride 2 to result in a feature map of dimension $28 \times 28 \times 256$. Next 6 layers consist of 512 filters for each layer, where the first one has dimension $3 \times 3 \times 256$ and other five have dimension $3 \times 3 \times 512$ with all filters having stride 1 in all directions. Max pooling layer is applied after first 3 layers of this 6-tuple and after the last layer, both of them having window of 2×2 and stride 2, resulting in a feature map of size $7 \times 7 \times 512$. This map is then fed as a vector of dimension 25088 into first next two fully-connected layers, each consisting of 4096 neurons each. Since the original paper [8] was prepared for ILSVRC classification challenge which consist of 1000 different classes, last layer has 1000 fully-connected neurons, however this number is variable according to the number of classes to classify. The last layer is a simple softmax function.

All of the convolutional layers of both networks were directly followed by a

ReLU unit.

1.2.3 Faster R-CNN adjustments to network architecture

When you want to run a detection and recognition pipeline, you are usually faced with a challenge of vast number of different positions and scales in which object you are interested might be in an image. Since passing an image through recognition network is quite time-consuming due to many computations in convolutional layers, many previous approaches tried to overcome this challenge with splitting this task into two different ones – detection of Regions of Interest (RoI) and then classifying only a limited number of RoIs by convolutional networks. However since many computations between object detection and recognition can be shared, Faster R-CNN [3] architecture merges those two computations back into one using a concept of Region Proposal Network which is a small extension of original recognition network. The RPN is put right before last pooling layer and consists of convolutional layer of size 3×3 (where width dimension is equal to width of preceding layer) and feeds the output of this layer into two sibling layers – box-regression layer (*reg*) and box-classification layer(*cls*) which are both implemented as a convolutional layers of dimension 1×1. The *reg* and *cls* layers outputs $4k$ and $2k$ dimnesional vectors respectively, where k stands for number of so called anchors at the processed position. Each anchor stands for a different setting of scales and aspect ratios at which RPN is analyzing given position. The original paper used three different scales and three different aspect ratios resulting in $k = 9$. This approach ensures translational invariation of anchors. *Reg* layer estimates regression of possible bounding box of an object and *cls* layer computes 2 probabilities of object/not-object in each anchor at given position.

Outputs of this RPN network are then fed into Fast R-CNN [2] part of networks, which expects RoI proposals and feature maps of last convolutional layer and applies so called RoI pooling effectively replacing last max pooling layer. RoI pooling divides each region of RoI proposal of dimension $w \times h$ into grid of size $W \times H$ where W and H are parameters of a layer (and in our experiments $W = H = 7$ was used) and performing classical max pooling on each cell of size $w/W \times h/H$ effectively squashing features of any RoI into fixed length vector of size $W \times H$.

This RoI pooled vector is then fed into usual fully-connected layers where the output at the end actually consists of two different output layers, one computing probability of a given class similarly as last layer of VGG16 or ZFNet networks and the other one estimating positions of bounding box depending on found class, therefore having $4C$ neurons where C is number of classes to classify. Obviously the bounding box predictor is not followed by softmax layer.

1.2.4 Training of a neural network

A general training algorithm for any neural network is usually done by evaluating the network on a training example and comparing the result of network with expected result by so called loss function. Loss function or error function is usually designed in a way that it measures by how much the result differs from the expected output and generally we try to minimize the output of such loss function over all training examples in a training set. This minimization can be seen as optimization problem and to solve this problem weights are adjusted in the direction of maximal gradient. This method is sometimes called as gradient descent algorithm.

Since deep networks have multiple layers, to adjust the weights of intermediate layers, the loss is then propagated backwards through the network by an algorithm called backpropagation which uses chain rule to iteratively compute gradients for each layer and adjust all of the weights in the network accordingly in attempt to minimize output of loss function. However, since the loss function L is defined as the function of the weight settings $\mathbf{w}$ over a dataset S as sum of the loss function evaluated at each training example (equation 1.2) and therefore computing gradient to compute new weights (equation 1.3) for each training example is time consuming, stochastic gradient descent is usually used. We usually do not want to adjust the weights by the whole amount of gradient so learning rate α is used (often $\alpha \ll 1$). $\mathbf{w}_t$ denotes the weights of the network at an iteration t.

$$L(\mathbf{w}, S) = \sum_{i=1}^{N} L(\mathbf{w}, S_i) \tag{1.2}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla L(\mathbf{w}_t, S) = \mathbf{w}_t - \alpha \sum_{i=1}^{N} \nabla L(\mathbf{w}, S_i) \tag{1.3}$$

Stochastic gradient descent does essentially the same as classical gradient descent, however computes gradient only for one training example at a time and immediately after computing gradient of a loss function for one training example adjusts the weights accordingly. However since this is only an approximation, a good compromise between stochastic gradient descent and classical gradient descent is to use a minibatch of a reasonable size n to adjust the weights according to gradient summed over n random examples of the training set. In next equations, we will denote loss function as only a function of weights computed on a minibatch, $L(\mathbf{w}) = \sum_{i=1}^{n} L(\mathbf{w}, S_n)$

One iteration usually adjusts the weights by a small amount so many iterations are still required.

Usually, other learning parameters different from learning rate are used as well. Most often used is momentum, μ, that takes into account weight update of previous iteration under factor μ as well. Formal definition can be seen on equation 1.4, where V_t denotes value by which weights are updated in iteration t and $\mathbf{w}_t$ denotes weights present in an iteration t.

$$\begin{aligned} V_{t+1} &= \mu V_t - \alpha \nabla L(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + v_{t+1} \end{aligned} \tag{1.4}$$

Another learning parameter often used is γ, used as a multiplier after a predefined number of iterations.

Caffe framework [5] allows you to set a multiplier of learning rate α for each layer used in your network. For convolutional and fully-connected layers, you can set them in `param` part of layer definition, as `param{lr_mult:x}`, where x is your desired α. This feature is useful if you do not want to train your layers anymore and believe training only particular layers might be useful, you can set α for a particular layer to 0. Note however, that therefore the error will not propagate to preceding layers then.

Faster R-CNN needs to optimize for different loss functions – two for RPN and two for classification and regression itself. The joint loss function for RPN is defined in equation 1.5, where i is an index of an anchor in a minibatch, p_i is predicted probability of the anchor i being an object. The ground truth label p_i^* is set to 1 for the anchor with highest intersection over union (IoU) with ground truth bounding box for a specific position out of k anchors (as described in section 1.2.3) and for each anchor having IoU with any ground truth box higher than 0.7. Note that this procudere may assign 1 to more anchors by a single ground truth bounding box. t_i is parametrization of estimated bounding box, t_i^* is parametrization of ground truth bounding box and λ is balancing factor used in order to balance L_{cls} and L_{reg} for approximately the same amount of error. λ was set to 10. As you can see, we computed loss function for *reg* layer of RPN only when the RoI was actually an object ($p_i^* = 1$). $L_{cls}(p, p^*)$ is a classical logarithmic loss function $L_{cls}(p, p^*) = -p^* \log(p)$ and $L_{reg}(t, t^*)$ is a smooth L_1 loss of a $t_i - t_i^*$. The parametrization of bounding boxes can be seen on equation 1.6, x, y, w and h denote center coordinates of estimated bounding box, its width and height, x_a, y_a, w_a and h_a denote the same properties for anchor box and x^*, y^*, w^* and h^* denote the same properties for ground truth box.

$$L_{RPN}(\{p_i\}, \{t_i\}) = \frac{1}{N} \sum_{i=1}^{N} L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N} \sum_{i=1}^{N} p_i^* L_{reg}(t_i, t_i^*) \tag{1.5}$$

$$\begin{array}{cccc} t_x = (x - x_a)/w_a & t_y = (y - y_a)/h_a & t_w = \log(w/w_a) & t_y = \log(h/h_a) \\ t_x^* = (x^* - x_a)/w_a & t_y^* = (y^* - y_a)/h_a & t_w^* = \log(w^*/w_a) & t_y^* = log(h^*/h_a) \end{array} \tag{1.6}$$

Smooth L_1 loss function is then defined on equation 1.7.

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{1.7}$$

The same loss function is used for classification and regression itself with slight change in L_{reg}. $L_{reg}(b_i, b_i^*)$ for prediction of bounding boxes does not know anything about the anchors anymore and therefore the function is

defined in equation 1.8. SGD Solver implemented in Caffe is able to optimize network for multiple loss functions and therefore we can start training with these two loss functions defined.

$$L_{reg}(b_i, b_i^*) = \sum_{j \in \{x,y,w,h\}} \text{smooth}_{L_1}(b_{ij} - b_{ij}^*) \tag{1.8}$$

1.3 Precision-recall curve

Precision recall-curve can be used to visualize performance of binary classification task. Let's define four variables – TP, TN, FP and FN. TP is a number of examples that were correctly classified with desired label, TN is a number of examples, that were correctly not classified – desired label was not present in ground truth and classification tool did not assign it a label, FP is number of examples where label was not present in ground truth, but classification tool assign this example a label and FN is a number of examples where label present was present in ground truth but classification tool did not assign it a label. After these definition, it is possible to define precision as a fraction of correctly classified examples out of examples, that were assigned a label (equation 1.9) and recall as a fraction of correctly classified examples out of examples, that should have been assigned a label (equation 1.10).

$$\text{precision} = \frac{TP}{TP + FP} \tag{1.9}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{1.10}$$

However, these definitions hold valid only for binary classifications. To create a precision-recall curve for a recognition task with one class whose output is a probability, it is possible to be incrementally setting a probability threshold which then acts as borderline between assigning a label and not assigning a label. To extend this for more classes at once, it is possible to treat the assignment of the class C_1 to the ground truth class C_2 as both FP and FN since the class C_1 was assigned to example which did not have the ground truth class C_2 and there was no class C_2 label assigned to the ground truth class C_2 although it should be.

The goal is to achieve as high precision as possible with as high recall as possible.

Chapter 2

Framework's user guide

The framework was build around Girshick's Python implementation of Faster R-CNN [3], found at `https://github.com/rbgirshick/py-faster-rcnn` and is entirely written in Python as well. The framework depends on lxml, NumPy and matplotlib libraries and on Caffe [5] library which is distributed with Python implementation of Faster R-CNN. The framework's code can be found at `https://gitlab.fel.cvut.cz/students/jasek-otakar`.

2.1 Evaluation framework

The whole framework consists of three main programs and 5 supportive ones. The programs are:

- Main programs
  - `recognize.py`
  - `precision_recall.py`
  - `train.py`
- Supportive programs
  - `faster_to_folder.py`
  - `victims_bbox.py`
  - `xml_to_kitti.py`
  - `kitti_to_xml.py`
  - `time.py`
- Configuration and miscellaneous python files and modules
  - `xmldb.py`
  - `_init_paths.py`
  - `help_parser.py`

In the following sections we will briefly describe all the programs. All of the positional arguments are specified in place of such argument, all of the optional arguments (the ones starting with `-`) must have the argument name preceding. All of the standalone programs using argument parser also take argument `-h`, which prints the usual help.

2.1.1 recognize.py

Program `recognize.py` is the core tool of the whole evaluation framework. `recognize.py` reads visual data in either form of folder with images or video file and produces XML file with detection and recognition data. For proper functioning you have to set `py-faster-rcnn` root dir and `MODEL_DIR` in `_init_paths.py`.

The program `recognize.py` has 15 arguments, one of them is required.

input
: Argument specifying input – could be either a folder with images and videos or a video file. If the input is video file or a folder contains a video file, then video is processed frame by frame.

-folder
: Optional argument, indicating folder to save output images (if boolean argument `-nv` is not specified). If it is not set, input folder is used. If visual output is desired, at least one of `-folder` and `-prefix` must be specified.

-prefix
: Prefix of saved output images. If it is not set, empty string is used. If visual output is desired, at least one of `-folder` and `-prefix` must be specified.

-t
: Measuring time of each detection and saving it to XML file. This argument is boolean.

-ox
: Name of output XML file. If the file already exists, it will be overwritten. If this argument is not specified, output will be written to stdout.

-gpu
: ID of GPU device on the system to use. Default value is 0.

-cpu
: CPU only mode. This argument is boolean, if it is set, it overrides `-gpu` option. Note that this will make whole process considerably slower.

-net
: Network architecture and weights to use. Currently only VGG16 [8] and ZFNet [9] are supported, both with plugged RPN unit. Choices are determined by contents of the folder `MODEL_DIR` specified in `_init_paths.py`.

Two networks originating from [3] are saved in folders `vgg16_orig` and `zf_orig`, pretrained on a training subset of VOC2007[10] dataset. Default value is `vgg16_orig`.

`-conf`
Minimum confidence level on detection needed to record the data. Default value is 0.8.

`-move`
Maximum movement of object (on each side of bounding box) within two frames to be recorded as the same object expressed as percentage of the image dimensions. This option applies only to video files processed. Default value is 0.02.

`-nms`
Minimum threshold for Non-Maximum Suppression (NMS) overlap expressed as percentage to merge two detections. Default value is 0.3

`-box`
Show infobox with class and confidence above detected class in output images. Applies only if visual output is desired.

`-nv`
Boolean argument, setting visual output to false (short for 'no visuals'). If set, arguments `-folder`, `-prefix` and `-box` do not apply.

`-vaf`
Treat videos as folders. When this argument is specified, resulting XML files will contain the same information for video files as they would for folders, therefore no merging of same objects within different frames of video files will be done. Tag `<object>` will then contain information about a frame in which object was detected. This argument is boolean.

`-v`
Verbose mode. This argument is boolean.

The order of arguments does not matter since there is only one positional argument and the rest of them are optional. However, it is considered a good practice to always have a positional argument either first or last, but never in the middle.

Usage example:

```
recognize.py [-folder FOLDER] [-prefix PREFIX] [-ox
    ↪ OUTPUT_XML] [-t] [-gpu GPU_ID] [-cpu] [-net
    ↪ {vgg16,zf}] [-conf CONF] [-move MOVE] [-nms NMS]
    ↪ [-box] [-nv] [-vaf] [-v] input
```

2.1.2 precision_recall.py

Program `precision_recall.py` is a tool for plotting precision-recall curves out of specified XML files containing ground truth information and prediction files outputted by `recognize.py` tool. This tool works only on files from running `recognize.py` tool in 'folder' mode (therefore it cannot be used on single video file). Note that this tool works without running graphical server since `matplotlib` is called with `Agg` backend.

This tool takes 6 arguments. Due to the fact that you cannot specify more than one required argument to take a form of list, list arguments `-gt` and `-pred` are listed as optional, however they can be seen as required since without them, the program has no data to operate on.

out
: Output file for a plot. If no extension specifying the format of desired plot is given, EPS file format will be used.

-gt
: List of ground truth files in XML format. Note that this argument takes a list, therefore you cannot use `-` as a prefix in any of the files, as this would end the argument sequence. List is terminated by either end of parameter string or a next parameter started by `-`, therefore this argument cannot appear before `out` argument.

-pred
: List of prediction files in XML format outputted by `recognize.py` tool. The same constraints on a list apply as in `-gt`.

-bins
: Number of bins to use for different confidence levels. Default value is 41. Note that higher values will lead to higher computation time.

-over
: Overlap ratio of bounding boxes for determining whether the prediction was correct. Default value is 0.5.

-cls
: Compute separate statistics for distinct classes. Note that when this argument is specified, it takes a significantly larger amount of time to process large datasets with mode classes. This argument is boolean.

-v
: Verbose mode. This argument is boolean.

Usage example:

```
1 precision_recall.py [-gt GT [GT ...]] [-pred PRED [PRED
    ↪ ...]] [-bins NUM_BINS] [-over OVER] [-data DATA]
    ↪ [-cls] [-v] out
```

2.1.3 train.py

Program `train.py` allows you to train a network from your ground truth XML files. The tool itself expects a base model specified by the same conventions as argument `-net` in `recognize.py` tool.

The tool takes 11 arguments, three of them are required.

use_net
: Base network model to be used for training. This argument can take same arguments as argument `-net` in `recognize.py` tool, however, it is recommended to train only on original networks, i.e. `vgg16_orig` and `zf_orig`.

out
: Specifies directory in which the trained network will be saved. This directory will have same structure as other directories in `MODEL_DIR`, therefore you do not have to do any meddling with files and immediately start using it by `recognize.py` tool.

xml
: List of XML files containing ground truth information about training dataset. Note that this argument takes a list of values, therefore it has to be terminated by either end of arguments or start of any positional argument.

-cache
: Whether to use cache to preload training dataset. This argument is a boolean argument.

-gpu
: ID of GPU device on the system to use. Default value is 0.

-solver
: Use different solver than the one used by base model. More information about this can be found at Caffe [5] documentation pages.

-iters
: Maximum number of iterations used by solver. Default value is 4000.

-cfg
: Use additional config file to override default Caffe and Faster R-CN-Nconfiguration. The configuration files are stored in YAML file format, however default setting is sufficient enough.

-rand
: Do not use predefined randomization seeds and randomize the whole process of training. This argument is boolean.

-keep_int
: Keep intermediate models. Note, that default snapshotting frequency of a

solver is set to 10000, so you need to overwrite this value by custom config file if using default number of iterations to actually get any intermediate snapshots. This argument is boolean.

`-nf`
: Do not use flipped images for training. This argument is boolean.

Usage example:

```
train.py use_net out xml [xml ...] [-cache] [-gpu GPU]
    ↪ [-solver SOLVER] [-iters MAX_ITERS] [-cfg CFG] [-rand]
    ↪ [-keep_int] [-nf]
```

2.1.4 faster_to_folder.py

`faster_to_folder.py` is a simple tool to help visualising and analysing data. Its main purpose is to take a XML file outputted by `recognize.py` and copy all outputted images from one class to one folder to easily examine all images labeled as i.e. class 'bottle'. It also creates a special folder for images in which were no classes detected.

This tool takes only two arguments:

`xml`
: Name of XML file to be used as a detection data.

`-dir`
: Root directory of folder structure created by copying files to folders by class. If not specified, default value is the folder of XML file.

Usage example:

```
faster_to_folder.py [-dir DIR] xml
```

2.1.5 victims_bbox.py

This is a simple tool creating ground truth files (with bounding boxes) from binary segmented images where each pixel is denoted by either 1 or 0, depending on whether the said pixel contains an object or not. By default fills all objects with class `person`. This behavior can be changed by argument `-cls`

This tool takes 3 positional arguments and 2 optional arguments.

`dir`
: Input directory containing only mask images for victim datasets.

`datadir`
: Directory, where original files reside. Important for being able to match ground truth and prediction files correctly.

`xml`
: Outputted XML file in ground truth format (attribute `conf` is filled with value `'gt'` standing for ground truth, root element of resulting XML is `<Ground_truth>`).

`-cls`
: String to fill as a class for encountered objects. Default is `person`.

`-name`
: Name of dataset to use in resulting XML file. Default is `Victims`.

Usage example:

```
1 victims_bbox.py dir datadir xml [-cls CLS] [-name NAME]
```

2.1.6 xml_to_kitti.py

This tool is used for transformation of our XML format to KITTI [11] data format. Currently it maps only classes {car, person, bycicle} to KITTI classes {Car, Pedestrian, Cyclist} as those are only three classes used in KITTI evaluation tool.

This tool takes three arguments, two of them are positional

`xml`
: Input XML file containing prediction information. Note that this file could be created only from `recognize.py` tool running in 'folder' mode.

`dir`
: Output directory to store KITTI data files. Note that all values that are missing from KITTI file format (occlusion, rotation, etc.) will be filled to 0. This does not bother KITTI evaluation tool if it is evaluating only object detection and recognition.

`-v`
: Verbose mode. This argument is boolean argument.

Usage example:

```
1 xml_to_kitti.py [-v] xml dir
```

2.1.7 kitti_to_xml.py

This tool does reverse conversion of `xml_to_kitti.py`. As `xml_to_kitti.py`, it maps only KITTI classes {Car, Pedestrian, Cyclist} to our classes {car, person, bycicle}.

This tool takes 5 arguments as opposed to `xml_to_kitti.py`, two of them or positional.

`dir`
: Input directory with KITTI data files.

`xml`
: Output XML file containing prediction information. This file is formatted as from `recognize.py` tool. If the KITTI data files are ground truth labels (they do not contain probability score), it treats them as such.

`-im_dir`
: Image directory to be used in XML file. Note that this directory is usually different from the directory where the label files are stored. Default value is input directory.

`-im_ext`
: Image extension to be used in XML file. Default value is jpg.

`-v`
: Verbose mode. This argument is boolean argument.

Usage example:

```
kitti_to_xml.py [-im_dir IM_DIR] [-im_ext IM_EXT] [-v]
    ↪ folder xml
```

2.1.8 `time.py`

This simple tool only takes timing data from XML files produced by `recognize.py` and outputs average, minimal and maximal time in seconds. It takes only one argument in a form of a list.

`data`
: List of XML files from `recognize.py`.

Usage example:

```
time.py data [data ...]
```

2.1.9 `xmldb.py`

Module `xmldb.py` contains only one class named xmldb, which is extension of class imdb from `datasets.imdb` from original Faster R-CNNframework and is used for training. You can specify custom `CACHE_DIR` where to store caches of loaded xmldb objects. In the most simple situation you need to initialize it only by a list of XML files passed as first argument to `__init__` method and the class takes care of the rest of setting up. However it might be desirable to use different classes than the ones from VOC datasets – then you can do so by specifying argument `clazz`.

2.1.10 _init_paths.py

In this module, you need to set two configuration paths in order for tool `recognize.py` to work properly. It essentially import paths of Caffe [5] and Python Faster R-CNNbindings to `PYTHONPATH` for you to be able to import them later.

`INSTALL_PATH`
: Root directory of your installation of `py-faster-rcnn`.

`MODEL_DIR`
: If you need to move pretrained Faster R-CNNmodels to a custom directory, specify such directory in this variable. You are not able to use default `MODEL_DIR` used by Faster R-CNN, since this framework uses slightly different file structure than the original Faster R-CNNframework. The directory structure of `MODEL_DIR` is briefly described in section 2.1.13

2.1.11 help_parser.py

This simple module just redefines `error` method of original ArgumentParser from module `argparse` to print help on error.

2.1.12 XML format

The most important tag of supported XML format is tag `<file>` which contains analysis of one file. The tag `<file>` can contain these attributes:

`path`
: Full path of analysed file.

`newpath`
: Full path of visual representation of detection. Only applicable for prediction files when option `-nv` was not set.

`time`
: Timing data of analysis of this particular image. Applicable only for prediction files if timing option `-t` was set and only for image files. Timing is in seconds.

`avg_time`
: Average time for a frame in a video file. Applicable only for prediction files ran on video when timing option `-t` was set. Timing is in seconds.

`framecount`
: Number of frames of a video file.

The tag `<file>` contains tags `<object>` which represent either ground truth objects or objects detected by Faster R-CNNnetwork. The tag `<object>` can contain these attributes:

class
Class of an object. Applicable for both image and video files.

bbox
Bounding box of an object. Applicable for image files only. The bounding box is in format (top-left, top-right, bottom-left, bottom-right).

prob
Confidence of detection, in range $(0, 1)$. For ground truth files, string 'gt' is used instead. Applicable for image files only.

Following attributes are applicable only for video files.

first_frame
Number of first frame where the object was detected.

last_frame
Number of the last frame where the same object was detected. The maximal allowed movement of an object within frames is determined by `-move` in tool `recognize.py`.

first_bbox
Bounding box of an object in a first frame where the object was detected.

last_bbox
Bounding box of an object in a last frame where the object was detected.

highest
Highest confidence in frames where the object was detected.

lowest
Lowest confidence in frames where the object was detected.

The tag `<file>` can be a standalone, but more commonly is enclosed within a tag `<folder>`. Tag `<folder>` can contain these attributes:

path
Absolute path to a folder.

files
Number of files in a folder.

usable_files
Number of analyzable files – only image or video files. Applicable only for prediction files, all tools for ground truth files get number of files programmatically and therefore this number would be the same as files.

The root element of a XML file is customizable – there is no strict rule what tag should you use. However, this tag with its attributes contains information about experiment. For prediction files we were using tag `<Faster_RCNN_experiment>` with attributes describing runtime parameters

of `recognize.py` tool. Note, that attribute "time" is different than a sum of all attributes "time" of `<file>` tags. This is due to the fact, that this value measures whole time of a whole experiment while timing for a file measures only detection runtime without postprocessing. For ground truth files we were using tag `<Ground_truth>` with one attribute specifying dataset.

XML example

As an example, we provide small portion of used XML file:

```
<?xml version="1.0" ?>
<Faster_RCNN_experiment conf="0.1" move="0.02"
    ↪ network="vgg16" nms="0.3" time="690.945771933"
    ↪ type="folder">
     <folder files="3" path="/path/to/folder"
    ↪ usable_files="2">
         <file path="/path/to/folder/img.png"
    ↪ time="0.6403028965">
             <object bbox="(132.91631, 282.65521, 524.46863,
    ↪ 461.32788)" class="person" prob="0.739661"/>
             <object bbox="(462.49478, 36.464203, 583.87549,
    ↪ 470.51736)" class="person" prob="0.20808"/>
         </file>
         <file path="/path/to/folder/vid.avi" framecount="10">
             <object class="car" first_bbox="(769.43323,
    ↪ 401.034, 901.0802, 477.86676)" first_frame="0"
    ↪ highest="0.995664" last_bbox="(767.76202, 408.27109,
    ↪ 899.92133, 475.68185)" last_frame="3"
    ↪ lowest="0.994525"/>
         </file>
     </folder>
</Faster_RCNN_experiment>
```

2.1.13 `MODEL_DIR` structure

Our custom structure of `MODEL_DIR` contains one directory called `models` and then one directory for each model. Directory for each model of `<name>` contains caffemodel of name `<name>.caffemodel`, where the `<name>` is the same as directory name and a solver for testing (used by tool `recognize.py`) called `test.pt` and a symlink to corresponding train directory containing train solvers. Directory `models` contains training solvers and is the one where each particular model directory links to.

Structure of `MODEL_DIR`:

```
MODEL_DIR
|- models
|  |- VGG16
|  |  \- end2end
|  |     |- solver.pt
|  |     \- train.pt
|  |
|  \- ZF
|     \- end2end
|        |- solver.pt
|        \-train.pt
|
|- vgg16_orig
|  |- test.pt
|  |- vgg16_orig.caffemodel
|  \- train -> MODEL_DIR/models/VGG16/
|
|- zf_orig
|  |- test.pt
|  |- zf_orig.caffemodel
|  \- train -> MODEL_DIR/model/ZF/
|
\- ...
```

2.2 Usage example

As an example, we provide series of commands used to train VGG16 network and obtain precision recall curves and intermediate data for victims dataset. Similar steps would be taken for ZFNet architecture or a different dataset.

```
./victims_bbox.py $DATA/victims/mask/tst/
   ↪ $DATA/victims/orig/tst/ ./tst_gt.xml
./victims_bbox.py $DATA/victims/mask/trn/
   ↪ $DATA/victims/orig/tst/ ./trn_gt.xml
./victims_bbox.py $DATA/victims/mask/val/
   ↪ $DATA/victims/orig/tst/ ./val_gt.xml

./train vgg16_orig vgg16_vict ./trn_gt.xml

./recognize.py $DATA/victims/orig/tst -ox ./tst_pred.xml -nv
   ↪ -conf 0.1 -net vgg16_vict -t
.recognize.py $DATA/victims/orig/trn -ox ./trn_pred.xml -nv
   ↪ -conf 0.1 -net vgg16_vict -t
./recognize.py $DATA/victims/orig/val -ox ./val_pred.xml -nv
   ↪ -conf 0.1 -net vgg16_vict -t
```

```
10
11 ./precision_recall.py ./tst_pr.eps -pred ./tst_pred.xml -gt
      ↪ ./tst_gt.xml
12 ./precision_recall.py ./trn_pr.eps -pred ./trn_pred.xml -gt
      ↪ ./trn_gt.xml
13 ./precision_recall.py ./val_pr.eps -pred ./val_pred.xml -gt
      ↪ ./val_gt.xml
14
15 ./precision_recall.py ./vict_pr.eps -pred ./tst_pred.xml
      ↪ ./trn_pred.xml ./val_pred.xml -gt ./tst_gt.xml
      ↪ ./trn_gt.xml ./val_gt.xml -cls
```

Chapter 3

Experiments

In this chapter we will show evaluation of two different network architectures, VGG16 [8] and ZFNet [9]. The datasets used for evaluation were our synthesised victims dataset and KITTI [11] object detection dataset. All predictions were marked as matched, if the overlap of predicted region and ground truth region had intersection over union (IoU) at least 50 % (argument `over` of tool `precision_recall.py` set to 0.5).

Data from our experiments can be found at `https://gitlab.fel.cvut.cz/jasekota/jasek-thesis-data`.

3.1 Datasets

Both datasets contain subsets to use for training and validation and Victims dataset also contains usable testing subset.

3.1.1 Victims dataset

Victims dataset is a dataset that artificially merges real data from various sources – all images were generated by letting random 3D models of humans "fall" into the scene as if they were victims of a crime or a natural disaster. Therefore this dataset proves to be quite challenging since the position and appearance of object to be detected is deformed compared to standard datasets.

The whole dataset contains 4986 images and is split into training, validation and testing subset which all contain approximate third of images – training subset contains 1604 images, validation subset contains 1645 images and testing subset contains 1737 images. Virtually all of these images contain only one object of class person – there is 4989 objects in the whole dataset and 4986 images. For obvious reasons, all of them are only positive examples.

This dataset did not have any ground truth files, instead it used only mask above each image to depict the sought object. Therefore it is needed to first create ground truth files by using `victims_bbox.py` tool.

Example of images from Victims dataset can be seen at figure 3.1

Figure 3.1: Example of images from Victims dataset

3.1.2 KITTI dataset

KITTI [11] dataset is a standard dataset used for evaluating object detection and recognition algorithms in automated driving. The dataset is split into training and testing subset by its authors, however only ground truth files for training subset are accessible by public. The ground truth files for testing subset are private and you can evaluate your algorithm on testing subset by submission of your results on KITTI server. This methodology was not suitable for our evaluation since we re-trained our networks on many different settings, so therefore we ignored testing subset and split training subset into training and validation portions in approximate ratio of 7:3. Training portion contained 5267 images and validation portion contained 2214 images totalling in 7481 images and we re-trained our networks only on training portion.

This dataset contains three different classes – Car, Pedestrian and Cyclist. Class Car has 20160 instances in training portion and 8582 in validation portion totalling in 28742 instances throughout the whole original training subset. Class Pedestrian has 3298 instances in training portion and 1189 in validation portion totalling in 4487 instances throughout the whole original training subset. Class Cyclist has 1174 instances in training portion and 453 instances in validation portion totalling in 1627 instances throughout the whole original training subset. We can see that class Car has 17.7 times more instances than class Cyclist and about 6.4 times more instances than class Pedestrian. This ratio is more or less consistent within both portions of original training dataset.

Examples of images from KITTI dataset can be seen on figure 3.2

Figure 3.2: Example of images from KITTI dataset

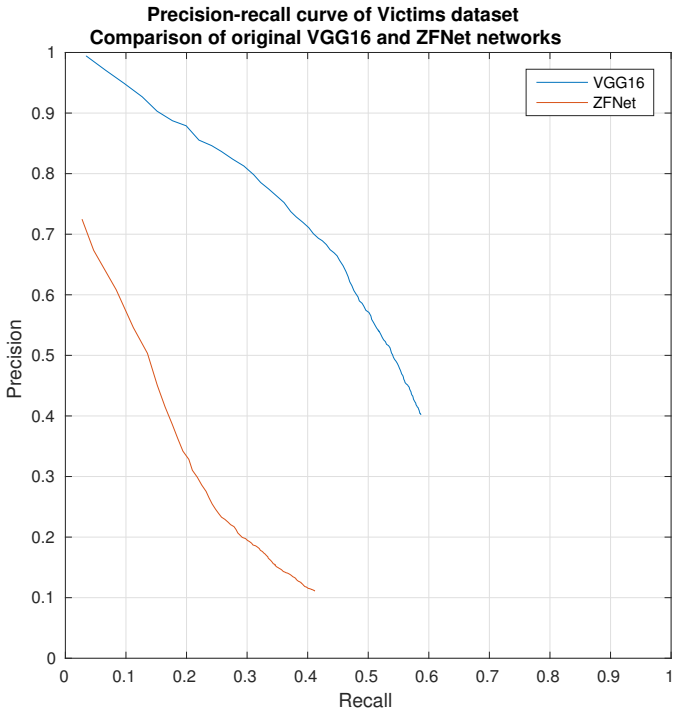


Figure 3.3: Performance on whole victims dataset

3.2 Evaluation of original networks

We show evaluation of networks that were trained only on VOC2007 [10] dataset as obtained from original Faster R-CNN paper and therefore the evaluation will be performed on whole datasets.

3.2.1 Victims dataset

This dataset is challenging, since the only correctly classified class was person and in a lot of the images, the person is deformed into victim position which generally does not resemble normal human position in which you would expect a person to be. Because of these deformations, the network is not performing as well as one might expect. Most probable reason for expected decrease in performance is the fact, that dataset actually used for training these networks (VOC2007 [10] training and validation datasets) did not contain many examples of people appearing in these unnatural positions.

On figure 3.3 we can see total performance for all three subsets combined into the whole dataset evaluated by VGG16 network and ZFNet respectively. It is clearly visible, that performance of VGG16 is considerably higher than the performance of ZFNet. This is compensated by shorter runtime for ZFNet, as shown in 3.4, but the runtime improvement is not that significant to justify such a drastic decrease in performance.

What might be of concern is the fact that none of these architectures were able to achieve perfect recall. This is due to the way Faster R-CNN network

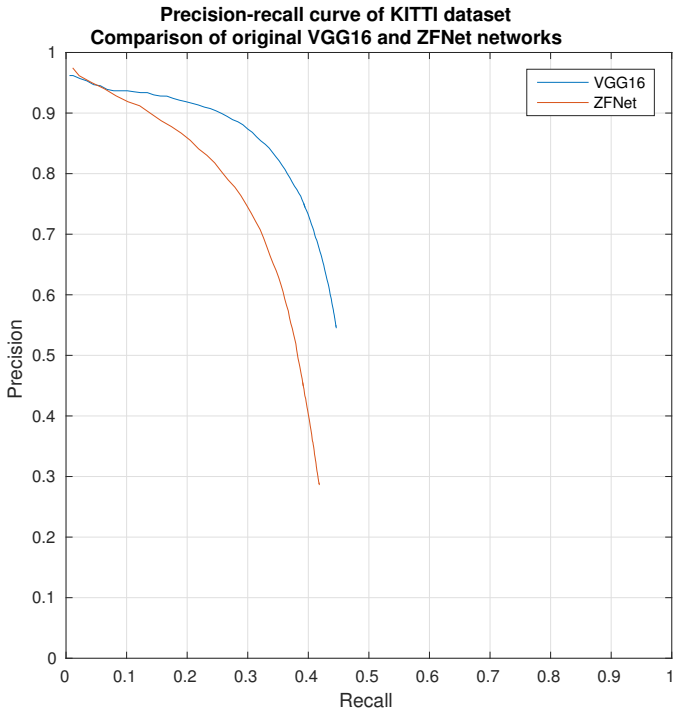


Figure 3.4: Performance on KITTI dataset measured on all three classes

operates – it selects only 300 best predictions (and even then some of them have confidence close to 0) and therefore some of the predictions that would amount to higher recall (with obvious loss of precision) are not selected by the network.

3.2.2 KITTI dataset

Figure 3.4 shows precision-recall curves for whole training dataset evaluated by VGG16 and ZFNet networks respectively. Ground truth files were obtained by `kitti_to_xml.py` tool.

Once again you can see much better performance by VGG16 architecture than by ZFNet. Same as on victims dataset, we were not able to achieve perfect recall, even worse, our maximal recall values were even lower. It can be seen that precision starts to drop rapidly in both networks after hitting recall of about 0.3. One can argue that this implies, that KITTI dataset is even harder than victims dataset – once again though, those values are on networks that were not specifically trained for KITTI dataset. Given this, the results are not as bad as they might seem on the first sight.

Figure 3.5 depicts performance evaluation on whole KITTI training dataset being broken down by each class. It is visible, that class Car was the most succesful one. What is however quite alarming is almost complete absence of correctly classified class Cyclist. One can attribute it to almost no training examples of class `bike` in VOC2007 training dataset. The rule of VGG16 net outperforming ZFNEt holds in every class except class Cyclist.

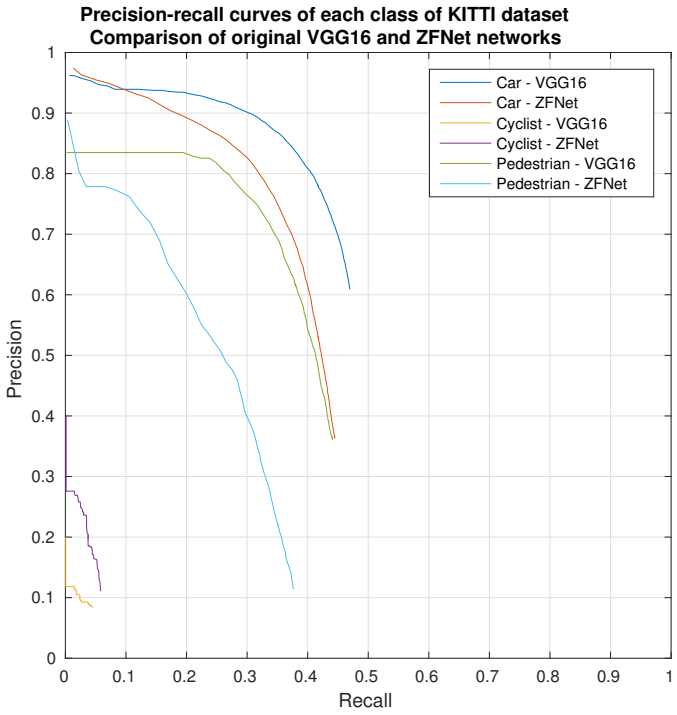


Figure 3.5: Performance on KITTI dataset broken down by different classes

3.3 Evaluation of re-trained networks

In this section we will evaluate re-trained networks specifically trained for concrete tasks. Each network was trained 5 times with different amount of training iterations by Caffe framework – 2000, 4000, 6000, 8000 and 10000 iterations. Evaluation was then performed only on the portion of dataset that was not present in re-training phase. For KITTI dataset it was the validation portion of the training subset consisting of 2214 images, for Victims dataset it was the testing and validation subsets put together consisting of 3382 images overall.

3.3.1 VGG16 network re-trained on KITTI dataset

Figure 3.6 depicts performance of VGG16 network re-trained on KITTI dataset evaluated for all three classes together. It can be seen that re-training of VGG16 network increased performance as expected. Increasing number of iterations of Caffe solver performing re-training kept improving the performance of a network, however with each increase of number of iterations, the increase in performance was lower. Quite interestingly, re-training with 6000 iterations performed better than 8000 iterations. It seems that limit for recall on Faster R-CNN network is close to value of 0.75.

Figures 3.7, 3.8 and 3.9 show performances of such re-trained network on classes Car, Pedestrian and Cyclist respectively. Class Car is significantly outperforming both other classes – this can be possibly attributed to much

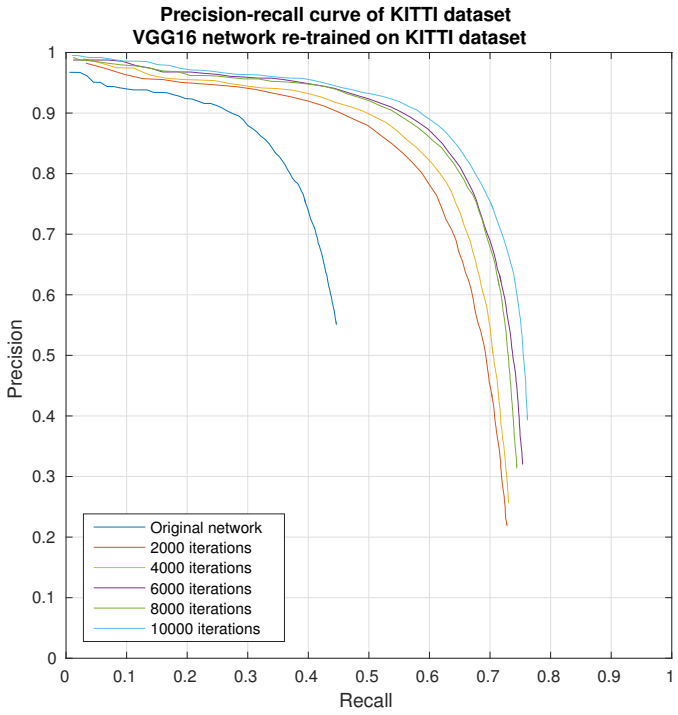


Figure 3.6: Performance on KITTI dataset of VGG16 network re-trained on KITTI dataset measured on all three classes

higher number of instances of class Car in training dataset. But since there are so many more objects of class Car, overall performance as seen on figure 3.6 is just slightly worse than the performance of class Car - it simply outweights the worse performance of other classes.

Another observation is the large improvement of class Cyclist over untrained network. For the network with 10000 iterations, it even outperformed class Pedestrian which has 2.81 times more objects in training dataset.

3.3.2 VGG16 network re-trained on Victims dataset

Figure 3.10 shows performance of VGG16 network re-trained on Victims dataset. Performance rapidly increased to precision values being above around 0.95 for recall values of 0.9 for networks re-trained by 8000 and 10000 iterations. However it seems that this might be maximal achievable performance since performance for 8000 and 10000 iterations are nearly identical. Much higher performance on Victims dataset than the performance on KITTI dataset is most likely to be attributed to the fact that Victims dataset is a lot more consistent than KITTI dataset therefore having training subset more closely related to the testing and validation subsets. However, another probable cause of such high performance might be overfitting on our dataset. To investigate further this cause we would need independent dataset that is closely related to Victims dataset but comes from a different domain.

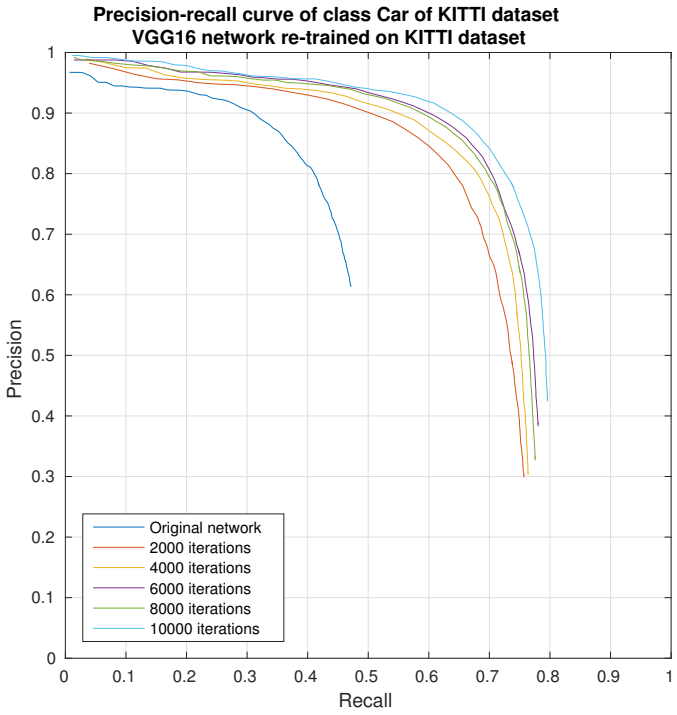


Figure 3.7: Performance on KITTI dataset of VGG16 network re-trained on KITTI dataset measured on class Car

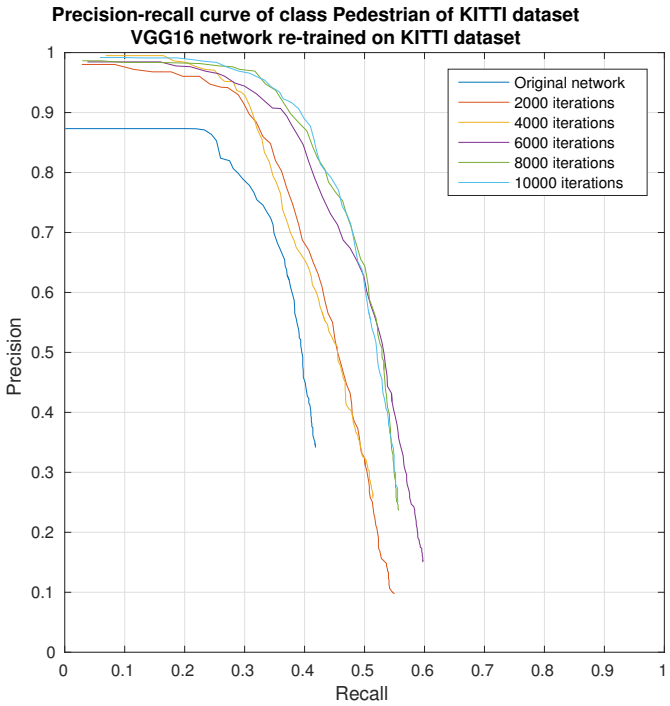


Figure 3.8: Performance on KITTI dataset of VGG16 network re-trained on KITTI dataset measured on class Pedestrian

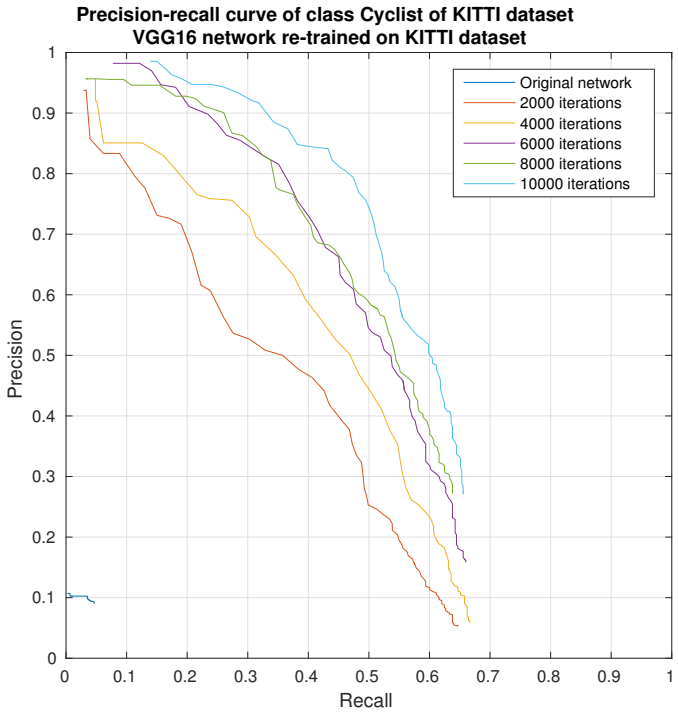


Figure 3.9: Performance on KITTI dataset of VGG16 network re-trained on KITTI dataset measured on class Cyclist

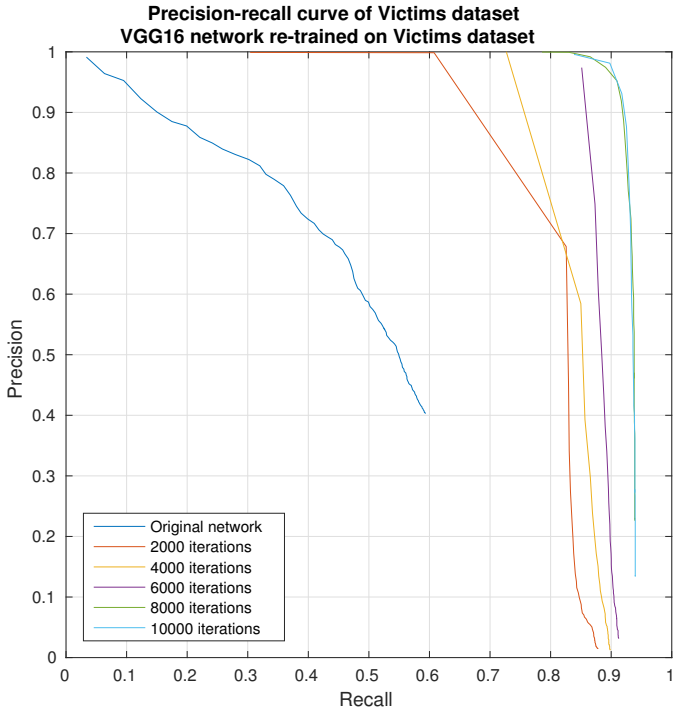


Figure 3.10: Performance on Victims dataset of VGG16 network re-trained on Victims dataset

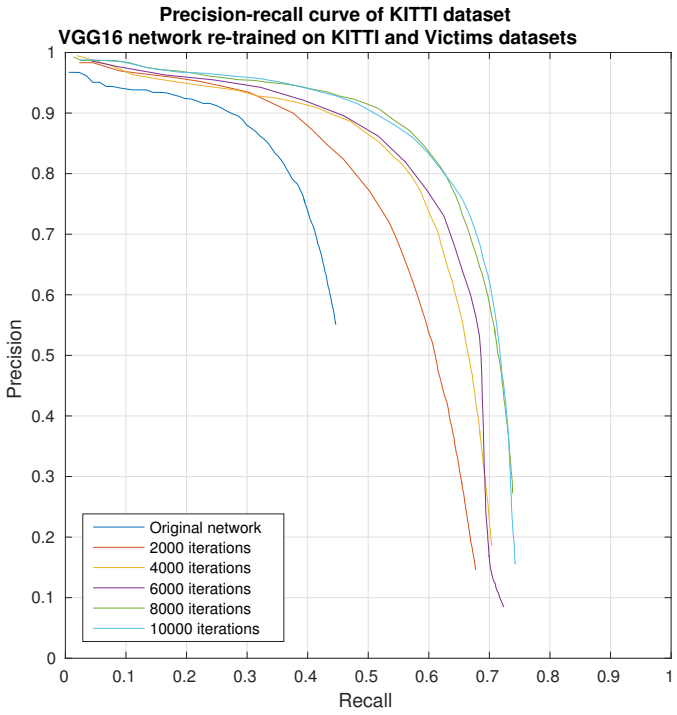


Figure 3.11: Performance on KITTI dataset of VGG16 network re-trained on both datasets measured on all three classes

3.3.3 VGG16 network re-trained on KITTI and Victims datasets

In another experiment we were re-training the networks on both training datasets together. Since such network was re-trained on both datasets, we can evaluate them on both datasets as well.

Evaluation of KITTI dataset by VGG16 network re-trained on both datasets

Figure 3.11 depicts performance of VGG16 network re-trained on both datasets evaluated on all three classes of KITTI dataset. We can see that while the increasing tendence in performance is fairly similar to such in network re-trained only on KITTI dataset, amount of the increase is lower, most notably in network re-trained only by 2000 iterations. Figures 3.12, 3.13 and 3.14 show performance on classes Car, Pedestrian and Cyclist respectively. One would assume that the most decrease in comparison to network re-trained only on KITTI dataset would occur in class Pedestrian since Victims dataset contains only occurences of class Person which are from completely different settings, however the most decrease is in class Cyclist – we attribute this to even higher underrepresentation of class Cyclist in training set.

Figure 3.15 shows comparison of performances of network re-trained by 10000 iterations only on KITTI dataset and on both datasets so we can clearly see that although the perfomance is better if network is re-trained for one

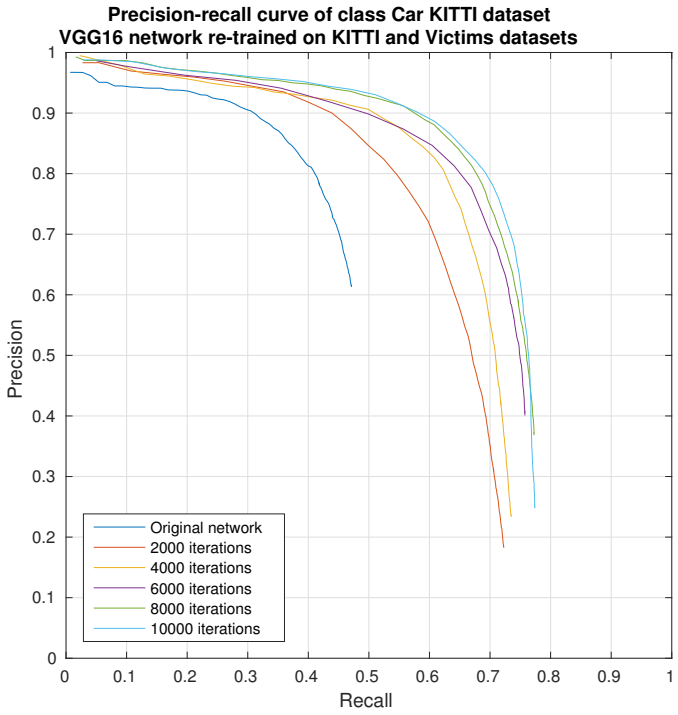


Figure 3.12: Performance on KITTI dataset of VGG16 network re-trained on both datasets measured on class Car

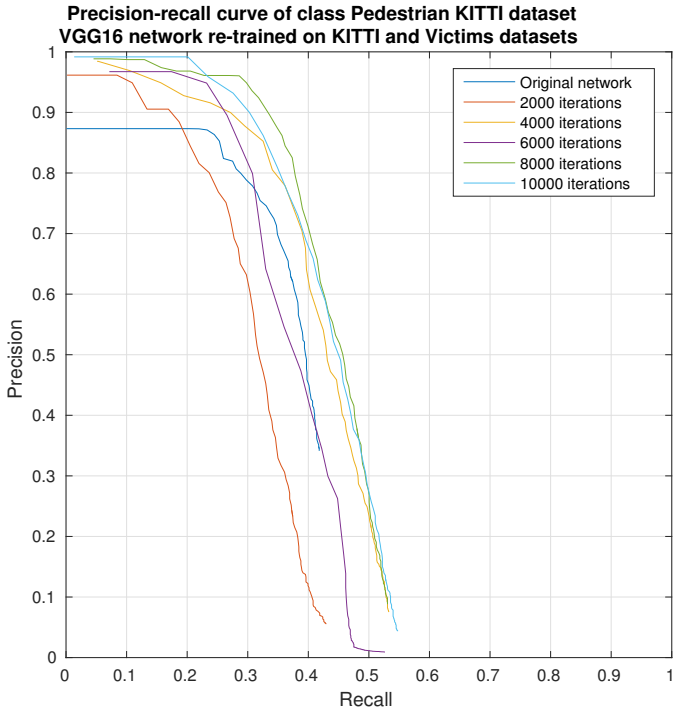


Figure 3.13: Performance on KITTI dataset of VGG16 network re-trained on both datasets measured on class Pedestrian

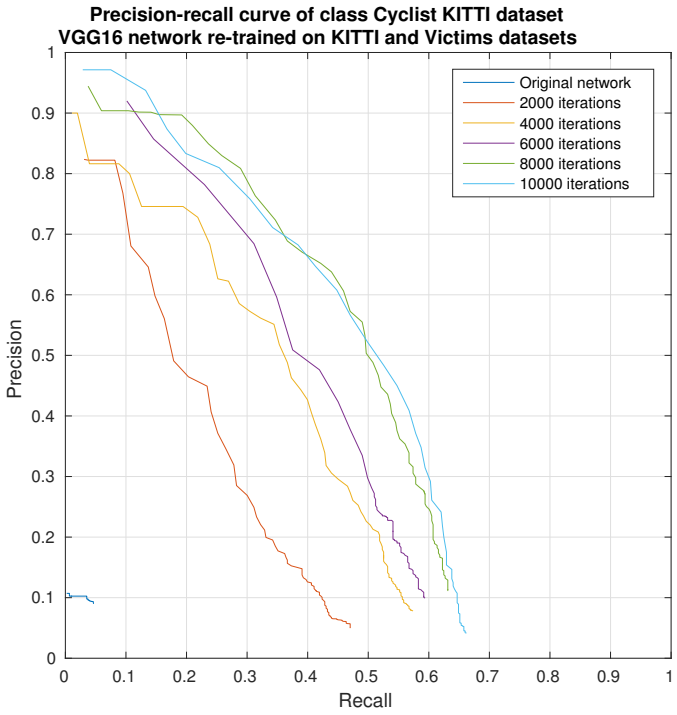


Figure 3.14: Performance on KITTI dataset of VGG16 network re-trained on both datasets measured on class Cyclist

task only, the decrease in performance is not that harsh and it might be actually suitable to train such network on multiple dataset in order to use only one network for recognition of multiple different datasets afterwards.

Evaluation of Victims dataset by VGG16 network re-trained on both datasets

The situation with Victims dataset is quite similar to the one of KITTI dataset. Figure 3.16 shows performance of VGG16 network re-trained on both datasets evaluated on Victims dataset. Once again it is lower than performance of such network re-trained only on Victims dataset however performance is still quite considerably high. Figure 3.17 shows similar comparison as figure 3.15 only for Victims dataset.

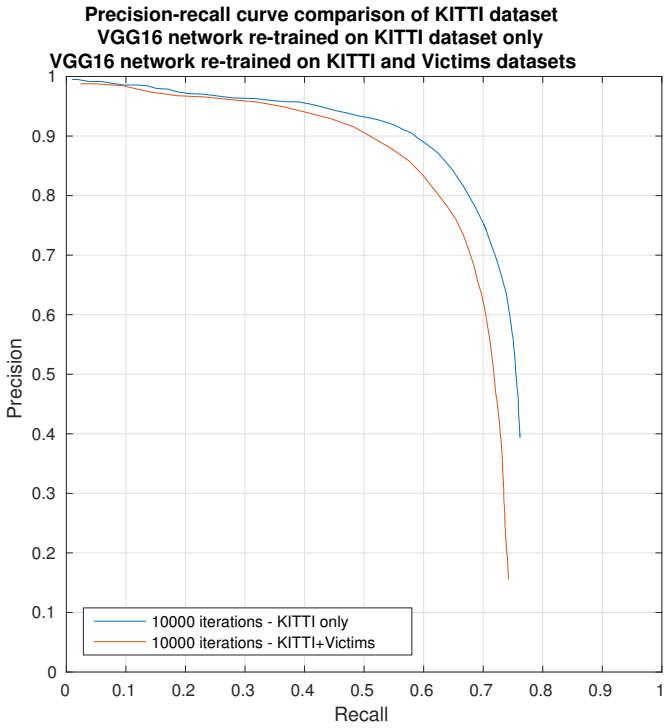


Figure 3.15: Comparison of performances on KITTI dataset of VGG16 network re-trained on KITTI dataset and on both datasets

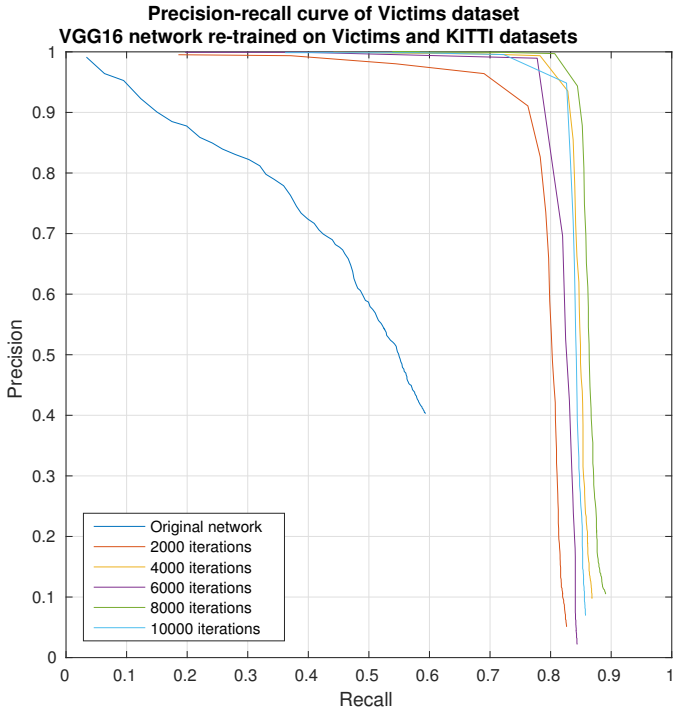


Figure 3.16: Performance on Victims dataset of VGG16 network re-trained on both datasets

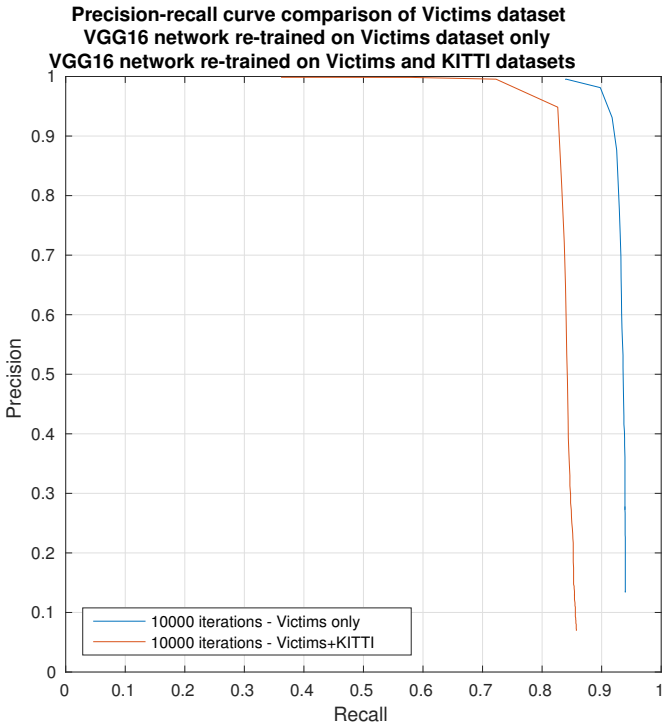


Figure 3.17: Comparison of performances on Victims dataset of VGG16 network re-trained on KITTI dataset and on both datasets

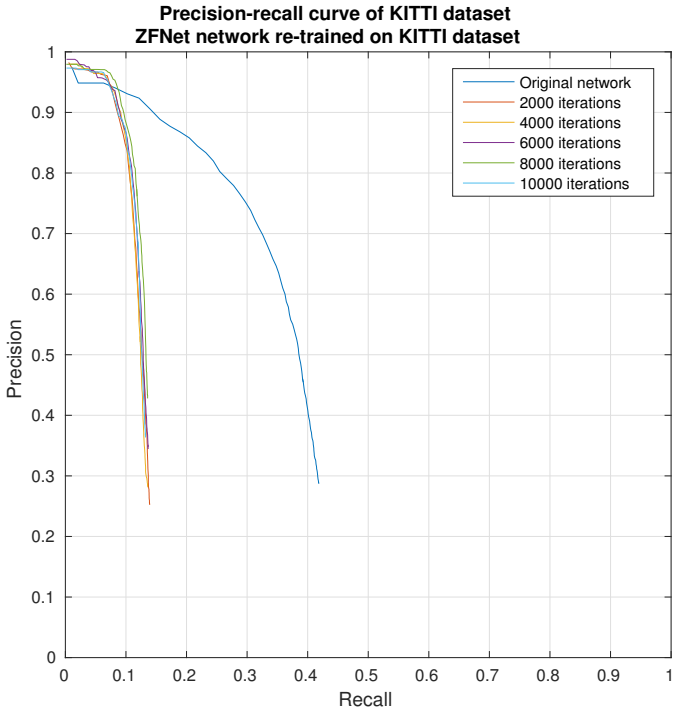


Figure 3.18: Performance on KITTI dataset of ZFNet network re-trained on KITTI dataset measured on all three classes

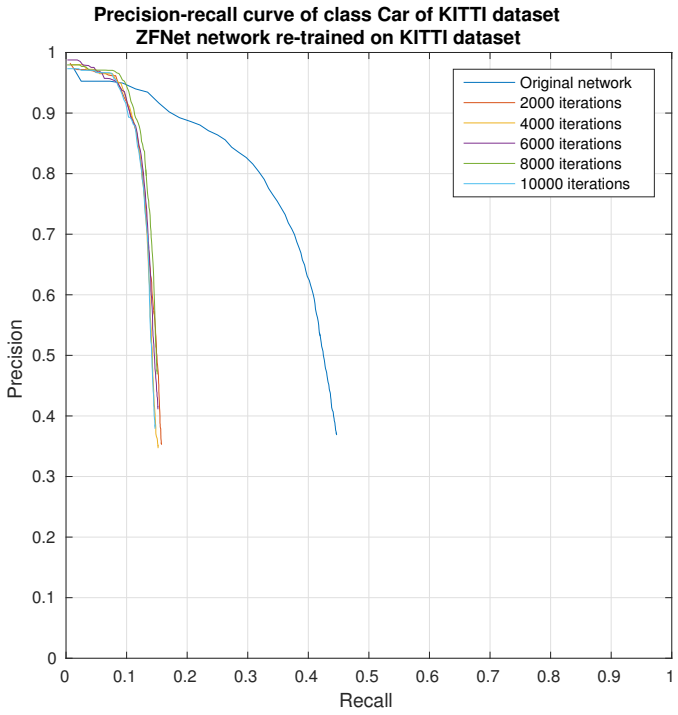


Figure 3.19: Performance on KITTI dataset of ZFNet network re-trained on KITTI dataset measured on class Car

3.3.4 ZFNet network re-trained on KITTI dataset

ZFNet performed quite badly on re-training overall. It seems that training method (described in 1.2.4) is not suitable for ZFNet since it actually performed worse than the original network trained only on VOC2007 [10] trainval. Another reason for such decrease might be in incorrect learning parameters of a network however the parameters were the same as used in original Faster R-CNN [3] paper.

Figure 3.18 shows performance of ZFNet network re-trained on KITTI dataset. As you can see, the performance actually decreased and number of iterations (at least in the range of 2000 - 10000) had virtually no effect on amount of decrease. If you break down the previous figure by measured class, we get figures 3.19 for class Car, 3.20 for class Pedestrian and 3.21 for class Cyclist. What is quite interesting is the fact, that performance for class Cyclist actually increased but only because it was already quite bad at the beginning.

3.3.5 ZFNet network re-trained on Victims dataset

Figure 3.22 shows performance of ZFNet network re-trained on Victims dataset. Surprisingly, although it preformed quite badly on KITTI dataset, the network actually improved on Victims dataset, though not by much. The performance still remains below the performance of untrained VGG16

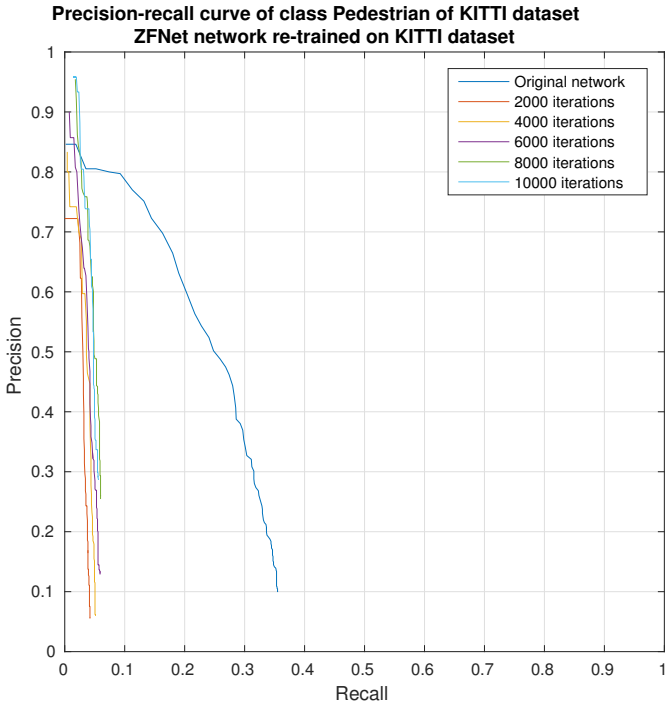


Figure 3.20: Performance on KITTI dataset of ZFNet network re-trained on KITTI dataset measured on class Pedestrian

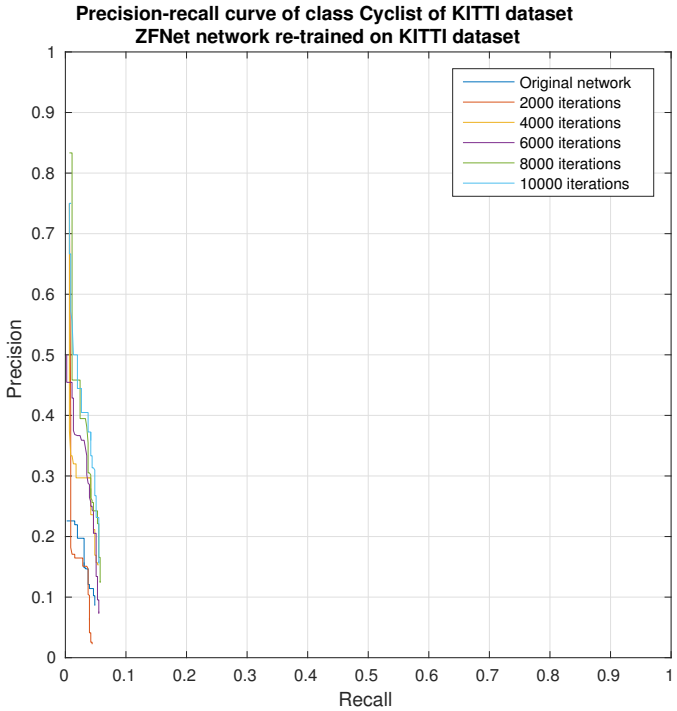


Figure 3.21: Performance on KITTI dataset of ZFNet network re-trained on KITTI dataset measured on class Cyclist

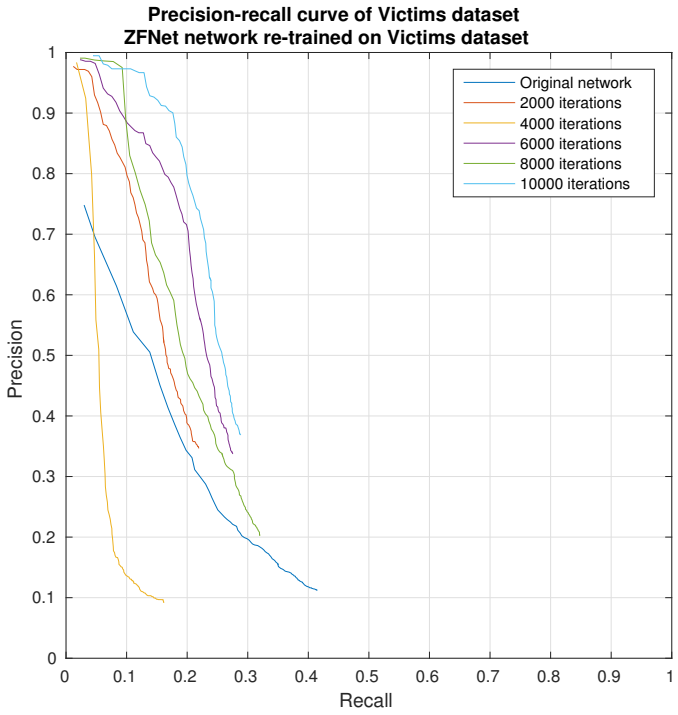


Figure 3.22: Performance on Victims dataset of ZFNet network re-trained on Victims dataset

network. Quite interesting is the fact, that the ZFNet network re-trained by 4000 iterations performed worse than the original ZFNet network while all the iterations outperform it.

3.3.6 ZFNet network re-trained on KITTI and Victims datasets

Evaluation of KITTI dataset by ZFNet network re-trained on both datasets

Figure 3.23 depicts performance of ZFNet network re-trained on both datasets evaluated on all three classes of KITTI dataset. Once again we can see that ZFNet network is not suitable for re-training with our parameters and the results are as bad as the results of the network re-trained only on KITTI dataset. Figures 3.24, 3.25 and 3.26 show performance on classes Car, Pedestrian and Cyclist respectively. The performance is about the same as for the network fine.tuned only on KITTI dataset. Figure 3.27 shows comparison of the performance of the network re-trained by 10000 iterations only on KITTI dataset and on both datasets. We can see that network re-trained only on KITTI dataset was slightly worse, but the difference between those two is marginal.

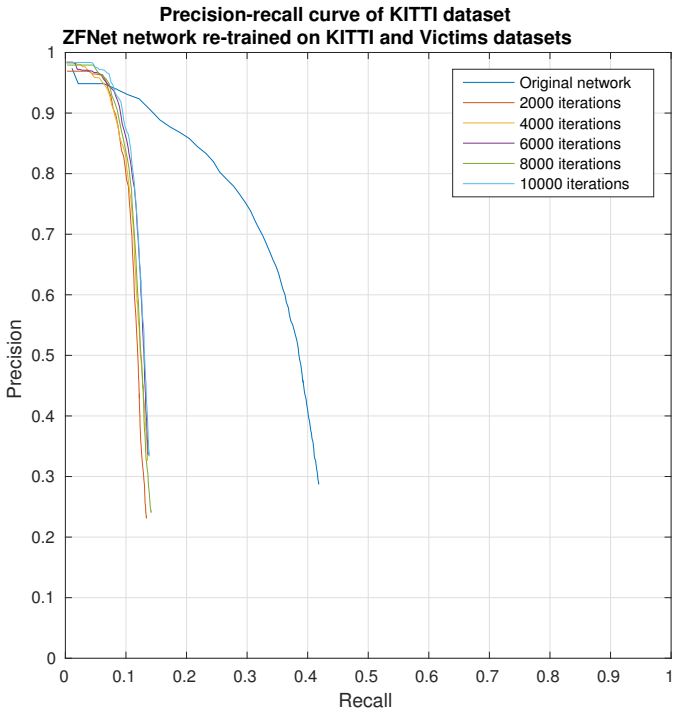


Figure 3.23: Performance on KITTI dataset of ZFNet network re-trained on both datasets measured on all three classes

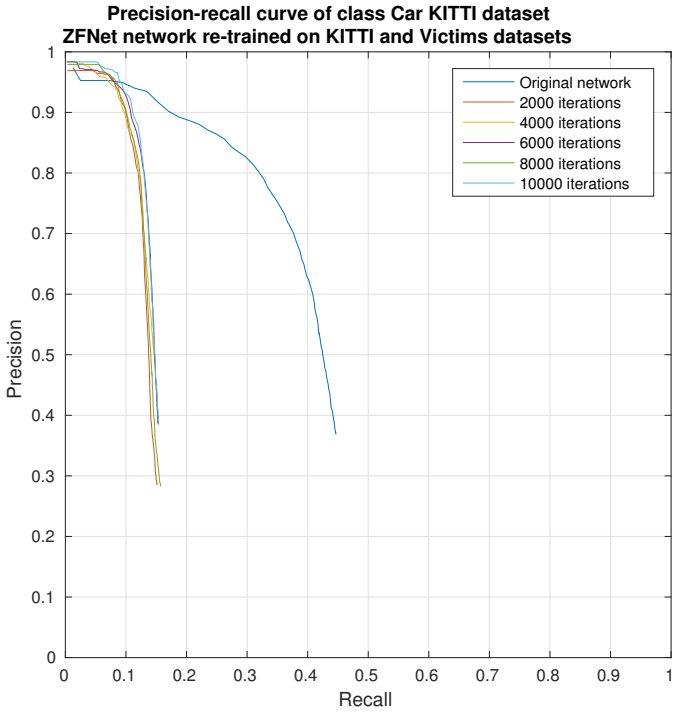


Figure 3.24: Performance on KITTI dataset of ZFNet network re-trained on both datasets measured on class Car

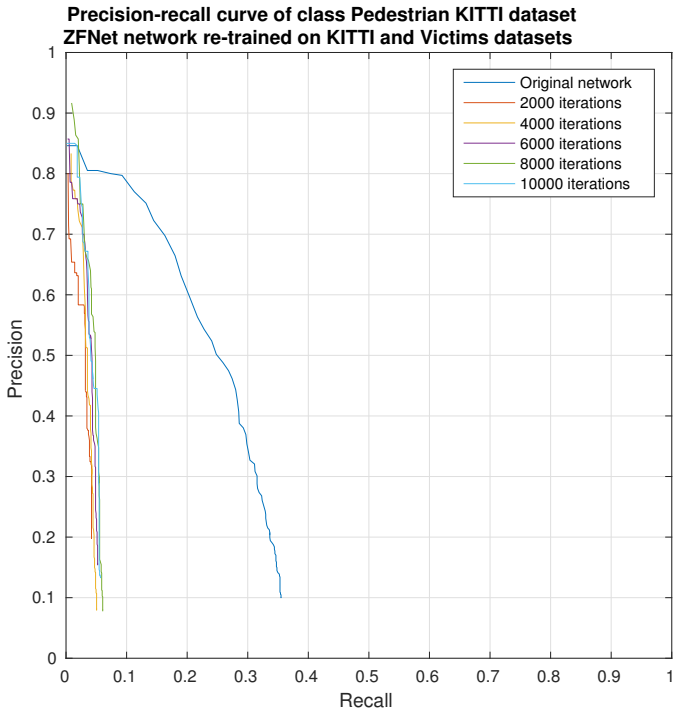


Figure 3.25: Performance on KITTI dataset of ZFNet network re-trained on both datasets measured on class Pedestrian

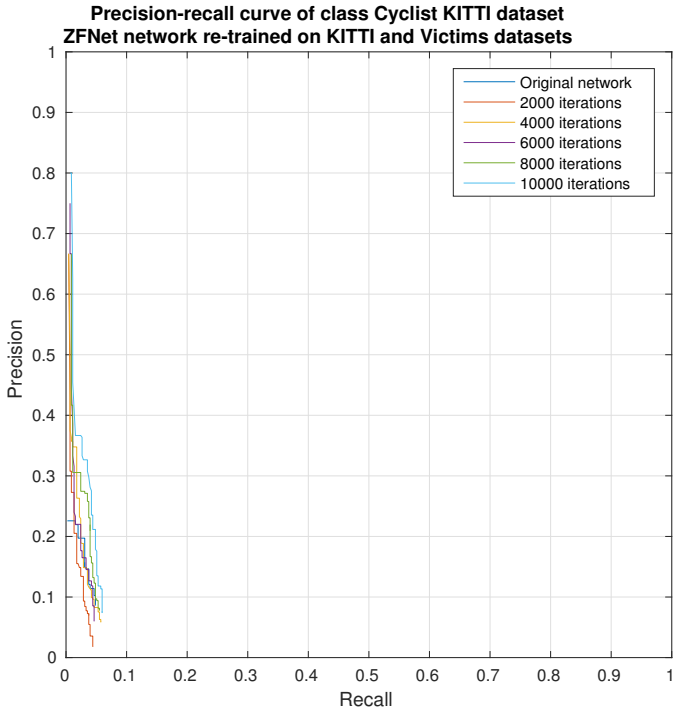


Figure 3.26: Performance on KITTI dataset of ZFNet network re-trained on both datasets measured on class Cyclist

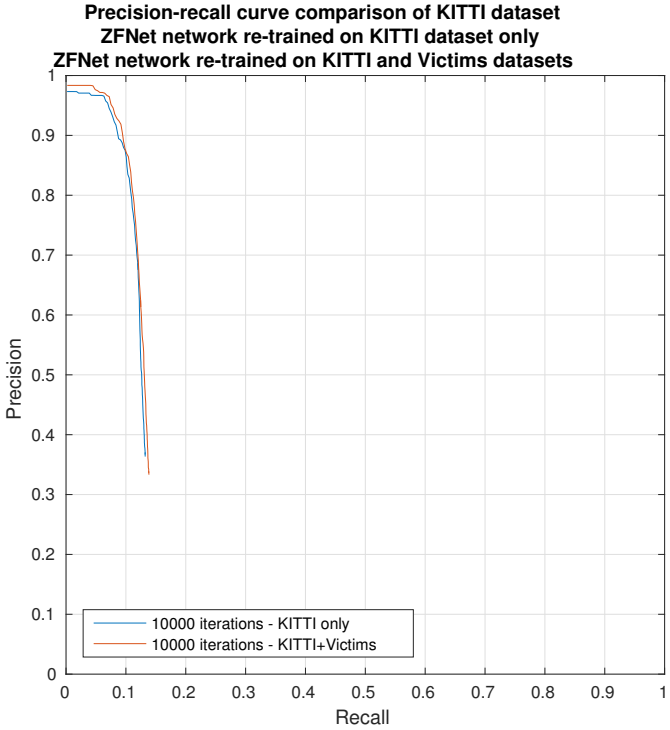


Figure 3.27: Comparison of performances on KITTI dataset of ZFNet network re-trained on KITTI dataset and on both datasets

Evaluation of Victims dataset by ZFNet network re-trained on both datasets

The tendence that Victims dataset is easier for re-training holds even for network re-trained on both datasets. However it is still not able to achieve performance of original VGG16 network, not to mention performance of VGG16 network re-trained specifically for Victims dataset. Figure 3.28 shows performance of such network. Figure 3.29 compares performances of ZFNet networks re-trained on Victims dataset only and on Victims and KITTI dataset. We can see, that training specifically for particular dataset only, yields better resuls even for ZFNet network.

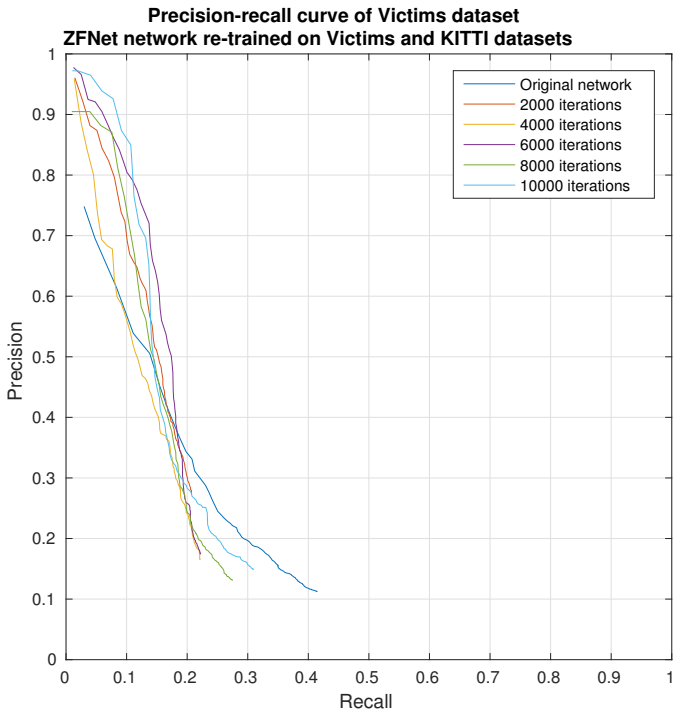


Figure 3.28: Performance on Victims dataset of ZFNet network re-trained on both datasets

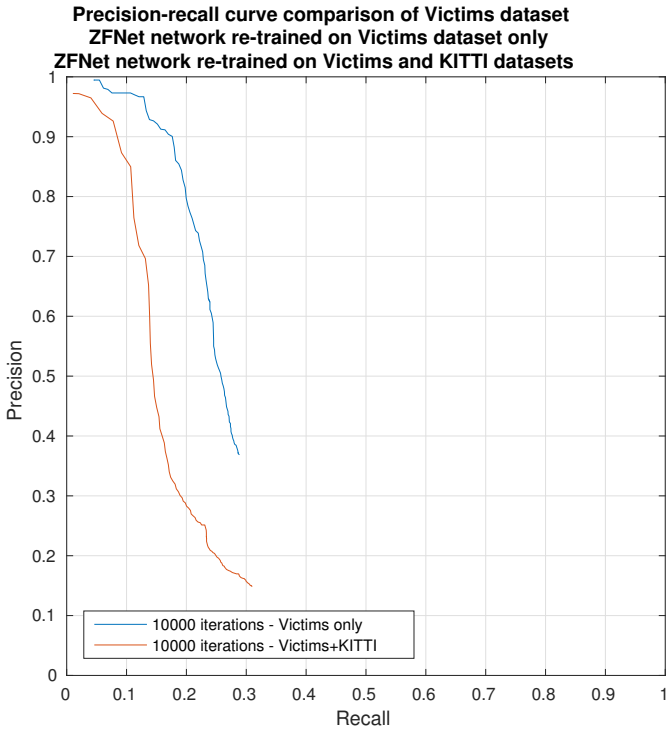


Figure 3.29: Comparison of performances on Victims dataset of ZFNet network re-trained on KITTI dataset and on both datasets

3.4 Time evaluation

Due to hardware limitations on the server where we were running the experiments and unexpected change of hardware during experiments (upgrade from NVidia GeForce GTX Titan Black to NVidia Tesla K40c) we were unable to ensure same conditions for all experiments required for fair comparison of timing. However since one experiment consisted of re-training a network and running tool `recognize.py` on all datasets we have (including configurations such as re-training on KITTI dataset and recognizing of Victims dataset which makes no sense to later evaluate performance on, but is easier to automate), and since each GPU had the same amount of such experiments, it is actually feasible to compare timing for recognition portion. We do not have any timing data for re-training portion of the experiments, however by experience, re-training ZFNet network was about 3 times faster than re-training VGG16 network.

Table 3.1 shows average times for both datasets expressed in miliseconds needed to process one image. The times could be most likely a bit faster since more comupting jobs were running at GPUs concurrently, however the ratio in between those two architecture would still most likely stay the same, showing that recognizing an image by ZFNet is about 2.4 times faster than by VGG16 network.

Table 3.2 shows average times dependant on dataset being currently recognized. We can see that KITTI dataset was slightly faster than Victims dataset, however the difference is a lot more significant for VGG16 network than for ZFNet.

Tables 3.3 and 3.4 shows runtime performance of original networks measured on CPU (Intel® Xeon® E5-2630 v3). VGG16 architecture proved again to be about 2 times slower than ZFNet. Runtime performance on 8-core CPU is about 11.7 times slower than running on GPUs. KITTI dataset recognition was again faster than recognition of Victims dataset, on CPU a lot more notably than on GPU.

	Average time [ms]	Min time [ms]	Max time [ms]
All experiments	252.61	60.17	1459.43
VGG16 architecture	350.14	149.78	1459.43
ZFNet architecture	141.09	60.17	1382.91

Table 3.1: Time comparison of different architectures for both datasets measured on GPU

	Average time [ms]	Min time [ms]	Max time [ms]
KITTI dataset			
Both architectures	228.50	68.62	1035.76
VGG16 architecture	316.86	149.78	1035.76
ZFNet architecture	140.14	68.62	493.94
Victims dataset			
Both architectures	271.29	60.17	1459.43
VGG16 architecture	400.07	189.22	1459.43
ZFNet architecture	142.51	60.17	1382.91

Table 3.2: Time comparison of different architectures and different datasets measured on GPU

	Average time [ms]	Min time [ms]	Max time [ms]
All experiments	2971.88	1454.86	7958.40
VGG16 architecture	3944.13	2775.03	7958.40
ZFNet architecture	1999.63	1454.86	5359.79

Table 3.3: Time comparison of different architectures for both datasets measured on CPU

	Average time [ms]	Min time [ms]	Max time [ms]
KITTI dataset			
Both architectures	2627.41	1454.86	7958.40
VGG16 architecture	3525.56	2775.03	7958.40
ZFNet architecture	1729.26	1454.86	2740.61
Victims dataset			
Both architectures	3488.72	1804.07	6373.09
VGG16 architecture	4572.15	3702.96	6373.09
ZFNet architecture	2405.30	1804.07	5359.79

Table 3.4: Time comparison of different architectures and different datasets measured on CPU

Chapter 4

Conclusion

We created easy-to-use framework for training, using and evaluating performance of Faster R-CNNtype of networks. We were unable to succesfully re-train ZFNet network to increase its performance on our datasets. This fact appears to be quite unfortunate since ZFNet architecture is smaller and therefore faster at re-training and recognition stages which could be therefore harnessed in the applications where time is critical.

We showed that VGG16 network is highly suitable for re-training on custom datasets. If you wish to use such network on classifying images from different domains, re-training on multiple datasets is feasible as well with reasonable trade-off in performance. If the training and testing data come from the domains that are quite close, the performance appears to be quite high as shown on Victims dataset. However, one must beware of overfitting the network.

Bibliography

[1] 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.

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

[3] 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.

[4] Artem Rozantsev, Mathieu Salzmann, and Pascal Fua. Beyond sharing weights for deep domain adaptation. *arXiv preprint arXiv:1603.06432*, 2016.

[5] 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.

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

[7] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer vision–ECCV 2014*, pages 818–833. Springer, 2014.

[10] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[11] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.