Embedded Real-time Object Detection and Classification using the Raspberry Pi and Movidius Neural Compute Stick

Udacity Capstone Report

by

Marko Stefanovic

May, 2018

Table of Contents

Definition

Project Overview

In the last couple of years huge advances have been made in classification and detection of objects in images. The main reason for this step is that the application of neural networks became feasible in the field of computer vision. In 2015 the artificial neural network "ResNet" scored on the largest image database "ImageNet" higher than any algorithm before - and surpassed at the same time the visual recognition power of a human. Quickly, a plethora of image applications followed like the google cloud vision API, a service which developers can use for their own applications to analyze the content of any image.

Although there might be limitless possibilities for new computer vision applications with this new technology, there are still some unsolved challenges left. Not every system or application can afford to have a large back end server with lots of computing power at its side like the google cloud vision service has. One can think about autonomous cars which have to recognize obstacles within milliseconds and don't have the time to wait for the response of a server. Or you might want to have a simple camera-based product, which the customer expects to just work out-of-the-box without having to connect it to the internet, subscribing to some cloud service or connect to some side-by computer.

To make a neural networks run on cheap, low powered embedded systems in real time is still an unsolved problem. There have been efforts to make convolutional neural networks (CNNs) run on the Raspberry Pi but they didn't cross the real time barrier [PW2014]. Chip manufacturers are working hard to bring to the market specialized hardware which is able to

process CNNs in real time. One of such a chip is the Intel Myriad processor family which could help systems to process live video images from a camera in real time using a CNN. This chip is made easily accessible for developers through the Movidius Neural Compute Stick (NCS) which is a small USB device which can run certain CNNs supposedly much faster than on a CPU.

The goal of this project is to develop a CNN which acts as a people detector and classifier and runs in real-time on a Raspberry Pi with a connected Movidius NCS. Forcing the real-time requirement it is explored how large the expected drop of quality is and how many additional classes besides "person" can be feasibly supported.

Problem Statement

In this project it is analyzed if it is possible to run a CNN for object detection and classification on the Raspberry Pi 3 in real-time. It is mandatory that the inference runs in real-time which means in our case approximately 50ms and furthermore that the set of supported classes contains the class "people".

The usage of the Movidius NCS is necessary because former efforts to run similar CNNs solely on the Raspberry Pi 3 have failed (1 fps see [BNET2017]). It can be assumed that it is simply not possible. The solution with the RP3 in connection with a Movidius NCS is still not trivial because even one of the smallest public CNN for such a task, Tiny YOLOv2, still needs around 155ms to run on the stick [TYNCS2017]. Hence the main work for this project is to find a much smaller CNN, which fulfills the real-time requirement while still retaining a high quality and several classes. My main focus during this research is to optimize both the the detection quality of the net as well as the number of classes it supports. To achieve this goal I use the darknet framework [DNET] to train different CNN models, calculate its quality on a validation database and measure its run-time behaviour on the Movidius NCS. The models that are trained and tested differ in input image size, number of feature maps inside of a layer and number of total layers. I also vary the number of classes by training a 1-class model (person only) and a 10-class model.

Metrics

To determine the run-time behaviour of different potential models I measure the total execution time of the model on the Movidius NCS. The total execution time is the sum of the transmission time of the input image to the NCS and the inference time on the NCS. Conveniently Intel provides a SDK for the NCS which contains an easy to use profiling function

in python. This function takes a model description in Caffes prototxt format, converts it to the proprietary Movidius format, and runs a random input image on a zero-weights model while measuring the time. This function is used extensively because it allows me to not only quickly dismiss "slow" models but also to quickly identify single "bottleneck" layers which take a long time to process and optimize them. The goal is to start the training only on models which are sure to fulfill the real-time requirement.

To evaluate the quality of a trained model I use the Mean Average Precision (mAP) [ZIS2009]. This metric is today the de-facto standard to measure the detection and classification performance of a multi-class model. The darknet framework, which I use for training, already provides an adequate function to calculate the mAP of a model.

Analysis

Data Exploration

A subset of the 2017 COCO database is used to train and test the resulting models. There were manifold reasons why I chose COCO. First, it is an openly available database which can be easily downloaded. It consists of a large training set with around 120K images and a much smaller validation set of 5K images. The annotation data of the database contains bounding boxes and 80 class names. It is therefore very suitable for our problem space since darknet YOLO also uses bounding boxes for detection. Second, on the official homepage of darknet the author published different models which were also trained with the COCO database and results regarding quality and run-time. Those models and results are suited as a potential benchmark and a first hint what the quality and run-time could be like. And finally, the COCO database is an established and popular database and there exists a very nice web interface to explore its data. The database explorer turned out to be quite useful since it made it very easy to quickly find interesting images which contain a certain combination of object classes. Since the database was in the last couple of years used for various competition it was expected to be clean of errors.

During the research phase different models with different number and sets of classes were trained. In Table 1 a distribution of all classes which were used in my models is shown. The table clearly shows that the "person" class is very overrepresented and that class imbalance is an issue. There are a few techniques to combat class imbalances and they will be discussed at a later point.

Another issue with COCO was that it contains images with a "crowd". Since crowded situations - like spectators in a stadium - don't belong to my use case those images were not

needed. Luckily they were labelled as "crowd" and could easily be removed from our training and validation sets.

Exploratory Visualization

Table 1 shows the number of COCO annotations for those classes for which different models have been trained in this project. Some classes like "person" or "cup" are better represented than other classes like "stop sign". This is not necessarily a bad thing since our use case is focused on the "person" class and and imbalance would mean a higher bias towards this class.

Image 1 shows two labelled images from the COCO database containing several classes. They were obtained from the COCO homepage by using their explorer tool. The objects are finely segmented by polygons, which are also available in the annotation data. But in our case we will use the axis aligned (non rotated) bounding boxes which are also available in the annotation data but not visualized in the explorer tool.

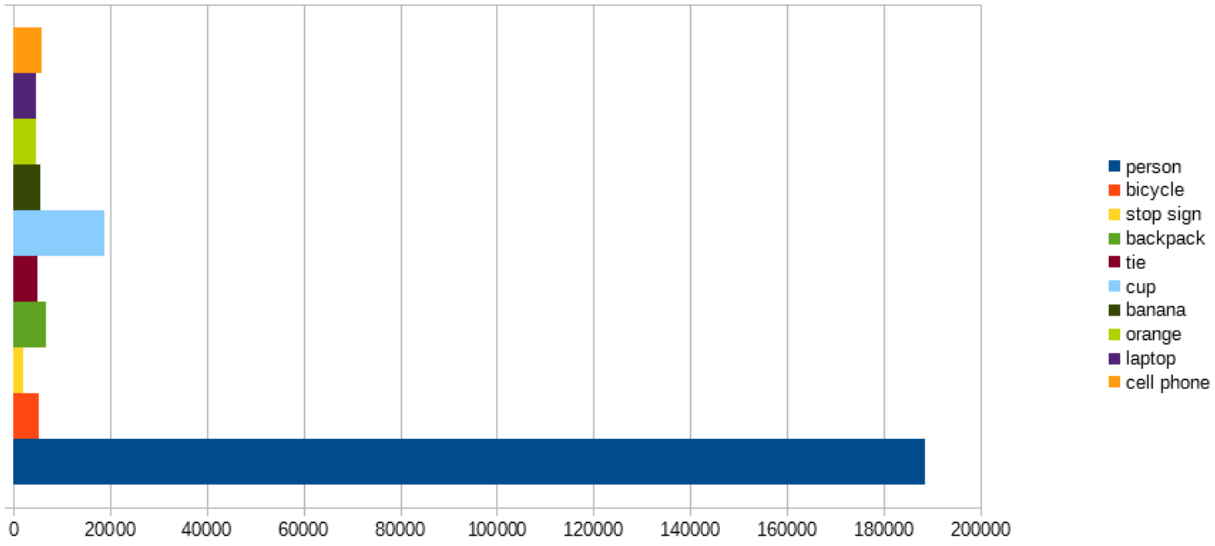

Table 1: Number of annotations in COCO database 'train2017' for selected classes

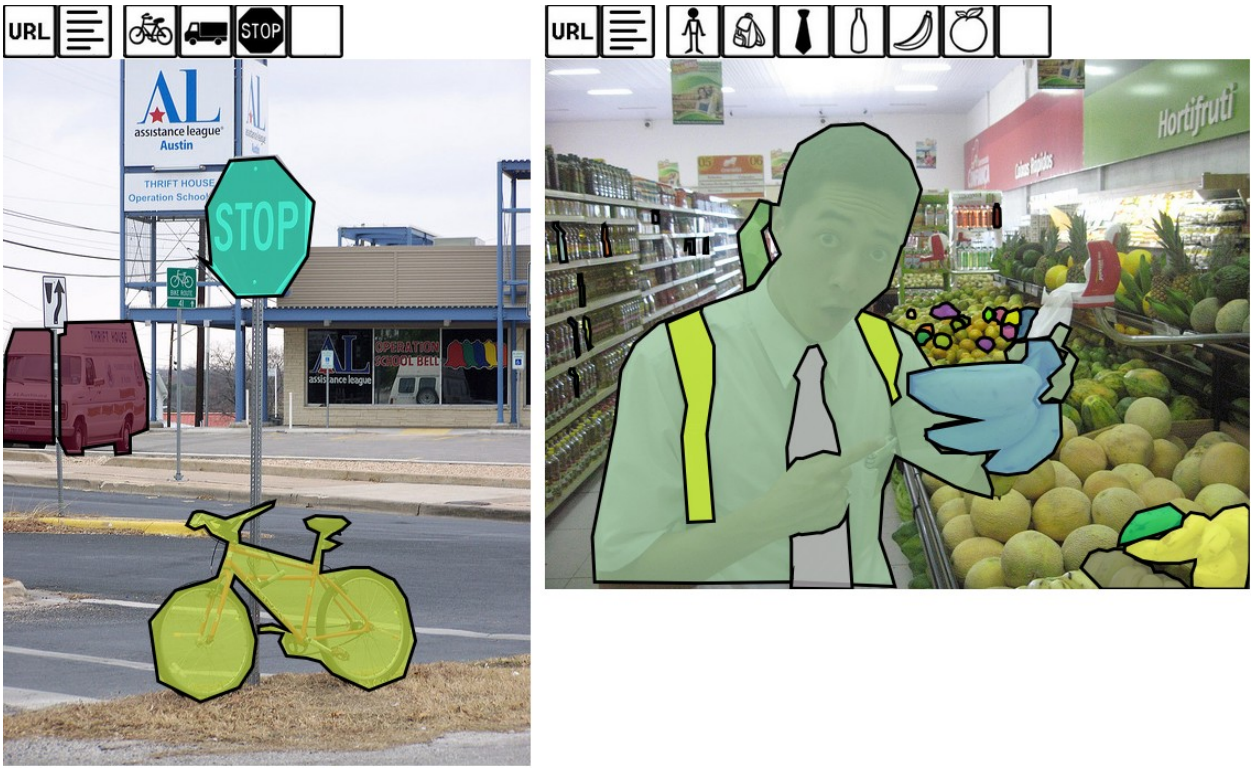

Image 1: Segmentated object inside the annotation data in COCOs database. Note that bounding boxes are not shown here but are contained in the annotation file.

Algorithms and Techniques

I use the darknet framework to train and validate the models. The reasons for choosing this particular framework besides those already mentioned above are manifold. First, the authors of this framework provide with the Tiny YOLOv2 model one of the smallest and fastest object detector as this time which also has a decent quality. The excellent speed vs quality ratio fits perfectly to my requirements [YOLOv2]. Second, the model only uses hidden layers which are also supported by the Movidius NCS [MVCAFFE] and there exist examples of how to run Tiny YOLOv2 on the Movidius. The plan was to use a simplified version of Tiny YOLOv2 and not to use any other different layer types therefore I didn't expect many troubles porting custom models to the NCS when I started the project. And last but not least darknet offers a reasonable amount of parameters to optimize training for your problem space.

However there are a few disadvantages using darknet as well which need to be discussed. Darknet is written entirely in C and its detection layer isn't integrated in the common frameworks like Tensorflow or Caffe yet. Therefore you need to compile the sources on your host system and work with the C-Application if you want to use it for training, validation or a quick demo. There

exist some python wrappers for darknet but they don't provide the full functionality and therefore weren't used in this project. Furthermore it uses its own proprietary formats for the model description and the weights file so those will need to be converted later. Also its final detection layer "region" is not supported by other framework and will have to be reimplemented in applications other than darknet.

Tiny YOLOv2 in its original configuration takes a squared image of size 416x416 as an input. If the input image has a different size it will simply scale the image to the correct size. This will distort the image if it is not squared. Since the input size for a model is a large factor that influences run-time performance we will experiment with different sizes. The impact of the image distortion will be discussed in chapter Conclusion further.

I would like to shortly discuss why darknet and Tiny YOLOv2 was chosen over MobileNet SSD-300 (Single Shot Detector) which is another popular and fast detector. While SSDs quality surpasses Tiny YOLOv2 (mAP: 74.3% vs 57.1%) its run-time is four times slower (see [YOLOv2] or [YOLOv2URL]). Since both models were expected to run in their original configuration below our real-time requirement I expected to get better results by using and optimizing Tiny YOLOv2.

Benchmark

I use as a benchmark the official Tiny YOLOv2 model which was trained on the COCO database on all 80 classes by the author. I downloaded the model definition and weights file directly from the homepage and converted them as described later in chapter Methodology. The metrics were applied as defined in chapter Definition in the following way: The run-time behaviour was determined by calling the Intel Profiling tool on this model. The mAP metric was obtained by using the darknet frameworks "map" function on the validation set. I will later explain in detail how this is done.

The results in Table 2 show a total run-time of 155ms (inference 115ms) and a mAP of 28%. The AP of the class 'person' - which is our "must-have" class - has a rather low AP of 34.5%.

The obtained benchmark shows that the run-time of the model needs to be reduced by approximately 70%. The methods to reduce the run-time, like cutting layers and number of filters, will most likely influence detection and classification performance in a negative way. To find a model which is fast and has a high performance was by far the most time consuming part of my research.

model id	# classes	input size	# layers	# filters in layer 0	run-time [ms]	AP person [%]	mAP
Tiny-YOLOv2	80	416	15	16	155	34.5	28.1

Table 2: Metrics and other attributes of our benchmark model Tiny YOLOv2. Input is a 416x416x3 color image. Runtime was measured on the Movidius NCS and mAP with darknet framework on COCO val2017 set.

Methodology

Data Preprocessing

There are a few occasions where data needs to be preprocessed:

- Annotation: conversion from COCO to darknet format
- Training: Rescaling of input images to the input size of the model
- Inference: Convert the output of a model to bounding boxes

I follow with a more detailed description of above points.

Annotation: Models are trained on subsets of the COCO train2017 database using the deep learning framework darknet. Unfortunately darknet has its own proprietary format for image annotations hence those annotations have to be converted from the COCO format (.json) to the darknet format (.txt). The necessary steps to create a custom darknet training set are the following:

1. download full COCO database (train2017 and val2017)
2. define a subset of classes
3. create a subset of COCO database which contains only images and annotation of classes defined above
4. convert COCO annotations to darknet annotations

I wrote the python module 'rapidus' (RAsperry PI moviDiUS) which contains functions to help with this conversion. COCO needs to be downloaded only once which can be done by calling:

```
import rapidus as rpd
rpd.downloadCoco(cocoDir)
```

where `cocoDir` is a user defined directory where COCOs annotations, training and validation datasets will be downloaded to.

Steps 2 to 4 are done for each subset of classes we want to train our models. There is a function which does all that:

```
import rapidus as rpd
classSet = ['person', 'car']
```

```
rpd.createYoloDatabase(cocoDir, targetDir, classSet)
```

where `cocoDir` is the directory where COCO was downloaded to, `targetDir` is the destination directory where the subset of images and yolo annotation files will be copied to and `classSet` is a list of COCO classes which filters only the class labels we need. To create a balanced dataset `createYoloDatabase` can be called with argument `balance=True`. Class balancing is done by subsampling where all images with a high person vs nonperson ratio are simply neglected.

Darknet has its own proprietary data format for the annotations of images. Each image file which is used for training or testing needs to have an annotation file with the same base name and extension '.txt' in the same directory. This file simply contains in each row exactly one label which consists of the class id and bounding box position:

```
id x y w h
```

where `id` is an integer with the class-id and `x y w h` are floating point numbers with the x,y position of the center of the bounding box and w,h as the width and height of the box. The bounding box coordinates are in coordinates relative to the width and height of the image and hence range in [0...1] (see also [AB2018]).

The format of the COCO database differs from the darknet format in the following way. For each of the sets (train2017 and val2017) there exists exactly one .json file which contains information about the image files, annotations and category ids. The formats of these three JSON objects are (only relevant fields are shown):

```
"images": <array of>
      "image_id": <Number>,
      "width": <Number>,
      "height": <Number>,
      "file_name": <String>

"annotations": <array of>
      "image_id": <Number>,
      "category_id": <Number>,
      "bbox": <array of Number>,
      "iscrowd": <Boolean>

"categories": <array of>
      "category_id": <Number>
      "name": <String>
```

The fields `image_id` and `category_id` can be used to cross reference between those three object arrays. The algorithm to create a darknet database from a subset of the COCO database basically

runs through the list of annotations, filters the needed categories and `iscrowd` flags, creates a set of image ids with filename and a list of annotations and writes this set in a separate .txt files, one file for each image where the labels are converted by the method shown earlier. The function `createYoloDatabase()` also creates a set of metafiles inside the target database folder:

- _filelist.txt: contains complete paths to all image files; needed for training with darknet
- _class.names: contains mapping from class ids to class names; needed for darknet traning
- _statistics.txt: contains total number of annotations for each class

Training: The images in the COCO database have different sizes and aspect ratios. The models expect a fixed input size therefore the images have to be transformed. The darknet framework does this internally by simply rescaling the images to the correct size. This process distorts the images which will be further discussed by me in chapter Conclusion.

We have to further take into account that this rescaling needs to be done by us in the same way when we do the inference later with the NCS without darknet. We also need the scaling parameters to later visualize the scaled bounding boxes correctly on the original unscaled image. I will show in the next section how this is done.

Inference: If inference is done through the darknet framework, for example to determine the mAP or to test the model on a single image, no further preprocessing steps are necessary because darknet is capable of doing all this internally.

In the case where inference is done on the NCS a few steps are necessary to be taken before and after inference:

- Images have to be rescaled and converted
- the last YOLO layer 'region' has to be calculated
- resulting output has to be converted to bounding boxes

Python offers a few moduls to help with this conversion. For the rest I wrote functions for the `rapidus` module. A simplified version of all this follows:

```
import numpy as np
import cv2
import rapidus as rpd

# create instances of helper classes for movidius inference and
# region layer forward pass
mvd = rpd.MvDetector(graphFile)
reg = rpd.RegionLayer(modelCfg)
```

```
# get image from a source (file, video, webcam)
imgOrig  = loadImage(imgSource)
oldHeight, oldWidth = imgOrig.shape[0:2]
# newWidth and newHeight is the input size for the model
img = cv2.resize(imgOrig, (newWidth, newHeight), cv2.INTER_LINEAR)
# scale data to [0...1]
img = np.divide(img, 255.)
# the NCS only accepts float16 data
img = img.astype(np.float16)
# call movidius api functions to do inference and get results
result = mvd.Detect(img)
# process region layer
result = reg.forwardPass(result)
# get scaling factors and convert resulting vector to bounding boxes
scales = [oldWidth/newWidth, oldHeight/newHeight]
boxes = rpd.getBoxesFromResult(result, scales)
```

A good description of how YOLO bounding boxes can be extracted from the final layers result can be found in [CB2017].

Implementation

In chapter Definition I specified the metrics which will be used in this project. In this section I will explain in detail the different steps necessary to calculate the metrics.

Determine the run-time of a model: Basically the following steps are needed to profile a model:

1. Install the Movidius API
2. Connect the Movidius NCS to the host computer
3. Create a new model description in the darknet .cfg format
4. Convert the darknet .cfg file to a Caffe .prototxt file by calling:
   ```
   import rapidus as rpd
   rpd.convertYoloToCaffe(pathToModelCfg)
   ```
5. use Intels `mvNCProfile` function to benchmark model by calling
   ```
   mvNCProfile pathToModelPrototxt -s 12
   ```

Step 1 needs to be done only once and is described in detail in [MVINST2016]. In step 3 the reason to first define the model in the darknet format (.cfg) is that this format is syntactically less complex than the prototxt format and hence easier to modify in a text editor. Furthermore we will need the model description in the darknet format later anyway when we start training. In step 4 the resulting prototxt file will be created in the same directory as the .cfg file. The function `mvNCProfile` in step 5 is a python script which is made available after installing the Movidius

NCS API. An example output of the `mvNCProfile` script for the original Tiny YOLOv2 model is shown in Image 2 together with the architecture of the CNN. The output shows that the last two convolution layers are the most time intense layers (26ms and 34ms) due to the large number of feature maps (512 and 1024).

Run-time measurements were performed on a virtual machine running Linux since the Movidius SDK requires for PCs an Ubuntu 16.04 installation.

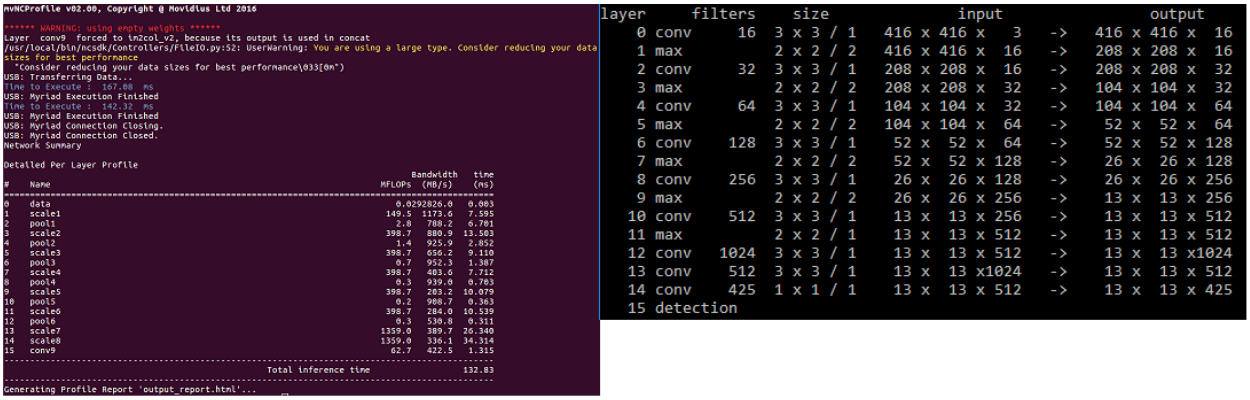

Image 2: Left: profiling of the model to the right using Intels profiling tool. The inference times of each layer is reported as well as the total execution time. In this case the layer with the highest run-time is 'scale8' with 34ms. Note that the right model is shown in darknet description while the same model is shown left in prototxt description.

Determine the quality of a model: The training and validation of the models were done using a fork of the darknet framework by AlexeyAB which is available in github [AB2018]. The reason for using the fork and not the original framework was that it contains a build environment for Windows and that it provides a method to calculate the mAP.

Training is started by calling

```
darknet detector train coco.data model.cfg >log.txt
```

where `model.cfg` is our darknet model and `coco.data` is a darknet metafile which is described in detail in [AB2018]. It contains a reference to the `_filelist.txt` of our database which was described earlier. I piped the output into a log file to be able to track the current value of the loss function during a training run. I wrote a function inside the rapidus module which plots the progress of the loss over time:

```
import rapidus as rpd
rpd.drawLossFromLog(logfile, alpha=0.002)
```

I used those plots to quickly compare different models during training and sort out models which didn't converge as fast as others. I did this to save time and to perform full training runs only on the most promising candidates. In Image 3 two exemplary plots of different models are shown.

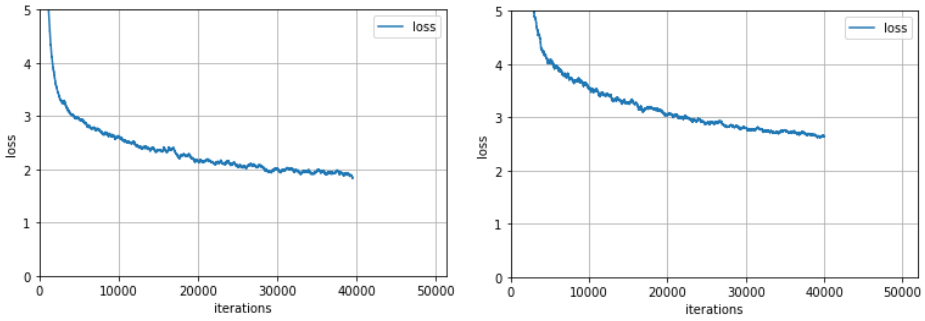

Image 3: Two plots of the loss function after 40000 iterations for two different models

Darknet backups after each 1000 iterations the current intermediate weights until it is finished after by default 500000 iterations. The mAP of a model is determined by calling:

```
darknet detector map coco.data model.cfg model_final.weights
```

where `model_final.weights` can actually be any intermediate weights file or, as in this case, the final weights file after the last iteration.

To detect overfitting I simply replaced the images in the validation set with the first 2500 images of the training set, calculated the mAP and compared the result to the mAP of the validation set (see Table 5 for results on my best model).

Full training runs were done on a Linux-PC provided by my company which runs two Geforce 1080 TI. A full training run approximately one day (running on both GPUs). The validation was done on my Windows-PC running a Geforce 770 GPU which took appoximately one minute for each model.

Visualize the performance of a model: To demonstrate the performance of a trained model the tool `mvdetect` was written. The tool processes videos, images or a webcam stream on the Movidius NCS with a trained model and shows the detected classes with a bounding box in real-time. Its purpose for this project is to first, visually validate that the model is robust enough by applying it to real world data and second, to be used later as a public demonstrator with a webcam.

A full walkthrough of `mvdetect` is beyond the scope of this paper since it tackles so many issues. However, I will point out a few interesting aspects which are as follows.

The architecture of the tool is shown in Image 4.

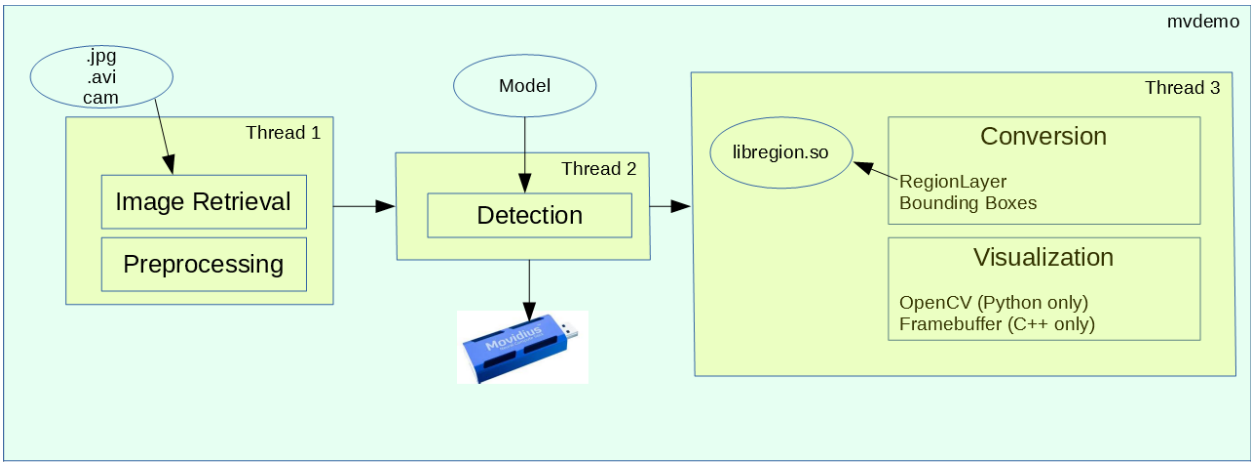

Image 4: Architecture of the mvdemo tool: There exists a C++ and a Python version of the complete tool. Multithreading is needed because each work package inside a thread is quite time consuming. Detection is run on the NCS and takes by requirement 50ms. libregion.so is a C++ library which implements the region layer. Rendering is done either in an OpenCV window (python) or using the framebuffer of the RPI directly (C++).

The tool was first written in python in a non-parallel way and did not run in real-time on the Raspberry Pi (approx. 6 FPS). There were two reasons for that: The first reason was that we have three operations which are quite time consuming and can stall the processing pipeline:

1. Image Retrieval and Preprocessing, which contain an IO operation to load and preprocess the image as described above
2. Detection which does the inference on the NCS and takes - as required - 50ms
3. Visualization which uses OpenCV to render image and results inside a window (cv2.waitFor)

The Detection operation takes 50ms (run-time on the NCS) and is therefore the theoretical minimum of the total run-time for one frame. The idea to reach this minimum is to start retrieving the next image and to start rendering the last image while the NCS is working on the current image. The obvious choice to all this is to let these three operations run in different threads. Unfortunately python doesn't support true multithreading because of the GIL (Global Interpreter Lock, which prevents true concurrent execution of interpreted python code, see also [GIL2018]). After porting the tool to C++ and using the C++ threading model, run-time performance improved to real time on a host PC but was still not possible on the Raspberry Pi: the OpenCV rendering function inside the X window manager was too slow (now 11 FPS instead of nearly 20 FPS on the PC). The solution was to not run the application under a X window system but on the console and map the final resulting image directly into the framebuffer of the Raspberry Pi [RC2012] which would show the image in full screen on the monitor. Using this

direct rendering approach together with the multithreading makes the application run nearly as fast as the total time of Movidius NCS (19 FPS).

Refinement

During the research phase of this project many models were trained and evaluated. There were basically three areas that could be tweaked to improve a model: the training set, the training parameters and the architecture of the model.

Training set: As mentioned in Analysis the classes in the COCO database are imbalanced. To create a more balanced database I called `createYoloDatabase` with the argument `balance` set to true in case of a multiclass model. The classes and their distribution for my 10-class model was:

balanced	person	bicycle	stop sign	backpack	tie	cup	banana	orange	laptop	cell phone
true	21112	3724	1631	3817	3197	17021	5364	4677	4247	4380
false	188498	5335	1922	6743	5017	18821	5515	4759	4661	5899

Table 3: Effect on class distribution after balancing the training set. Balancing was done by subsampling: To reduce the number of annotations for persons all images with a high number of person annotations were removed.

I stopped training after 60,000 iterations (approx. 12% of a full run) because a full run would have taken too much time on my computer. The APs of the different classes were:

balanced	person	bicycle	stop sign	backpack	tie	cup	banana	orange	laptop	cell phone	mAP [%]
true	34.9	14.1	50.3	9.4	21.9	15.4	11.9	25.4	37.1	15.4	23.6
false	37.3	11.1	45.8	0.8	16.1	11.0	9.9	20.1	27.4	10.4	19.0

Table 4: Effect of balancing the training dataset by subsampling. The APs of all classes are higher exept our main class "person". Training run was interrupted after 60,000 iterations.

Although the results with a balanced database looked promising the drop of almost 3% AP of the "person" class - our most important class - lead me to the decision to train all future models with an unbalanced dataset. I will further discuss this decision in chapter Conclusion.

I also trained all models on images which weren't flagged as `iscrowd` because my use case wouldn't include crowded scenes. I didn't evaluate this decision further since approximately only 10% of all images with persons were flagged as `iscrowd` and I didn't expect a big impact of leaving them out.

Training parameters: The darknet framework offers many parameters to further optimize the training. The following parameters help to reduce overfitting by performing data augmentation on-the-fly:

- saturation, exposure, hue: randomly adjust images in the HSV color space
- jitter: randomly translate image
- random: if true the image and network layers are randomly resized to larger or smaller size

Turning these parameters or a subset of them off had a negative impact on overfitting as it is shown for the parameter "random":

model	**mAP val2017 [%]**	**mAP trainVal2017 [%]**
random=1	13.3	13.8
random=0	9.6	17.9

Table 5: Effect of training parameter 'random' after 40000 iterations: if turned off the training clearly overfits. val2017 is the correct validation set; trainVal2017 is a validation set made of 2500 images which were also used for training.

For this reason I left them in their original configuration and didn't have any future problems with overfitting during training.

Other parameters which influence training behaviour are:

- learning_rate, momentum: controls the speed to convergence
- max_batches: controls the number of iterations
- steps, scales: raises or lowers the learning_rate depending on iteration

Again, I left the parameters in their original configuration because training runs showed a good convergence and I wanted to rather spend time on comparing and optimizing different model architectures.

Model architecture: During the refinement phase my main focus was to try out different models which were less complex than the original Tiny YOLOv2 and tracking their qualities. The following measures were taken to reduce complexity and to optimize the quality:

1. reduce input image size from 416 x 416 to 208 x 208
2. reduce number of feature maps in front layers
3. increase number of layers at the back
4. reduce number of supported classes

Note that in my case the last layer of all models must be of size 13x13 therefore the input image size must be a multiple of 13. A size of 208x208 was believed to be enough to still detect small object or details in an image. The number of feature maps in the beginning layers have a very large impact on run-time performance since input sizes are still large and result in many operations during convolution. To compensate the loss of neurons on the front side of the net I increased the number of layers at the back side by chaining up convolution and/or pooling layers

with different number of feature maps or strides. The reduction of the number of classes was necessary to "relax" the model by reducing the necessity of learning the features of many different objects.

In the following two tables the results of different models in which above strategies were applied are shown.

model id	# classes	input size	# layers	# filters in layer 0	runtime [ms]	AP person [%]
0	1	416	10	32	75.0	40.8
1	1	416	13	16	55.6	54.8
2	1	416	21	16	53.7	57.2
3	1	208	10	32	58.5	42.6
4	1	208	13	16	**49.8**	51.7
5	1	208	21	32	54.1	**58.4**

Table 6: Results of different 1-class models. Note that mAP is in the 1-class case the same as the AP of that class. The number of filters in layer 0 have a large impact on run-time and are therefore shown here.

model id	# classes	input size	# layers	# filters in layer 0	runtime [ms]	AP person [%]	mAP [%]
6	10	416	23	16	53.5	49.7	30.2
7	10	208	21	32	**53.3**	**52.3**	**30.9**
tiny-yolo2	80	416	15	16	155	34.5	28.1

Table 7: Results of two 10-class models and for comparison the original Tiny YOLOv2 model.

The architecture of model 5 is shown in Image 5.

```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 2   208 x 208 x   3   ->   104 x 104 x  32
    1 conv     64  3 x 3 / 2   104 x 104 x  32   ->    52 x  52 x  64
    2 conv     32  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x  32
    3 conv    128  3 x 3 / 1    52 x  52 x  32   ->    52 x  52 x 128
    4 conv     32  1 x 1 / 1    52 x  52 x 128   ->    52 x  52 x  32
    5 conv    128  3 x 3 / 1    52 x  52 x  32   ->    52 x  52 x 128
    6 conv     32  1 x 1 / 1    52 x  52 x 128   ->    52 x  52 x  32
    7 conv     64  3 x 3 / 2    52 x  52 x  32   ->    26 x  26 x  64
    8 conv    256  3 x 3 / 1    26 x  26 x  64   ->    26 x  26 x 256
    9 conv     64  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x  64
   10 conv    256  3 x 3 / 1    26 x  26 x  64   ->    26 x  26 x 256
   11 conv     64  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x  64
   12 conv    128  3 x 3 / 2    26 x  26 x  64   ->    13 x  13 x 128
   13 conv    512  3 x 3 / 1    13 x  13 x 128   ->    13 x  13 x 512
   14 conv    128  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x 128
   15 conv    512  3 x 3 / 1    13 x  13 x 128   ->    13 x  13 x 512
   16 conv    128  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x 128
   17 conv    512  3 x 3 / 1    13 x  13 x 128   ->    13 x  13 x 512
   18 conv    128  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x 128
   19 conv    512  3 x 3 / 1    13 x  13 x 128   ->    13 x  13 x 512
   20 conv     30  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x  30
```

Image 5: Architecture of model 5. Note that pooling layers were replaced by 1x1 conv layers and 3x3 conv layers with stride=2.

Results

Model Evaluation and Validation

The results in Table 6 show that in our case models with a large number of layers and a smaller number of early filters are beneficial. A comparison of models 0 vs 2 and 3 vs 4 show and increase of 17% and 9% of AP. The reduction of the input size and increase of the number of filters in layer 0 provide slightly better results. A comparison of models 2 vs 5 and 6 vs 7 show an increase of 1.2% and 0.7%.

Following the discussion above the choices for the 1-class case is model 5 and for the 10-class case model 7. Both models fulfill the run-time requirements and have a reasonable AP for the class 'person' which varies between 50% and 60%.

To evaluate the robustness of the 1-class model I performed the following measures on model 5. First, I used for the calculation of above mAP metrics only images from the standard validation set of COCO (approx. 2500 image). Those images aren't included in the training set. Second, to find out if the model overfits I took the first 2500 images of the training set and used them as the validation set (trainVal2017) to calculate the mAP score. The result was close to the real validation set which shows that the model has truly learned the features of the class and not just memorized images.

model	mAP val2017 [%]	mAP trainVal2017 [%]
5	58.4	58.5
7	30.9	34.4

Table 8: Analysis of overfitting: val2017 is the true validation set and trainVal2017 is the fake validation set which contains images from the training set. Model 5 shows no signs of overfitting while model 7 is a bit worse. This is explainable because model 7 is our 10-class model with a rather small number of training instances of non-person classes.

Finally I used a video of a badly lit shopping mall scene with a shaky handheld camera and ran the model for a few seconds on it. Image 6 shows that the model performs quite well even in such a problematic environment.

Justification

The metrics of the best two models of the 1-class and 10-class cases are shown together with the benchmark model:

model id	# classes	run-time [ms]	AP person [%]	mAP
5	1	54.1	58.4	58.4
7	10	53.3	52.3	30.9
Tiny YOLOv2	80	155	34.5	28.1

Table 9: Metrics of my final models vs the benchmark

Our custom models fulfill the run-time requirements while lowering the run-time of the benchmark model by 65%. The AP of the person-class is approximately 20% higher than the benchmark model and the 10-class model has a slightly 2% higher mAP. These results come at the price of having more specialized models which cannot detect as many object classes as the benchmark model.

When evaluating the run-time of the models it should be mentioned that the run-time might slightly vary depending on the system. The reason is that the reported run-times include the transmission time of the input image to the NCS which happens through the USB-Bus. These transmission times vary between 6-10ms and depend on the USB version (2.0 or 3.0) and if the profiling tool runs on a virtual host system. You also have to take into account that the Raspberry Pi uses the same bus for USB and Ethernet so transmission time can vary depending on network load.

Having discussed the metrics of the two models above I come to the conclusion that they are stronger solutions for my use case than the benchmark model: they run faster and detect people better than Tiny YOLOv2.

Conclusion

Free-Form Visualization

To visualize the characteristics of the 1-class and 10-class model I chose to apply the detector to random images that were downloaded from the internet. The first image is a processed frame from a video which was taken with a camera inside a shopping mall. The other images were found by using google for the supported 10 classes of the 10-class model.

Image 6: Resulting bounding boxes of 1-class model 5 in one frame of my test video. The model correctly handles clipped persons, overlaps and perspective foreshortening. However, boxes do not always fit the objects perfectly and one woman in the background wasn't detected at all. The video was processed using the mvdemo tool.

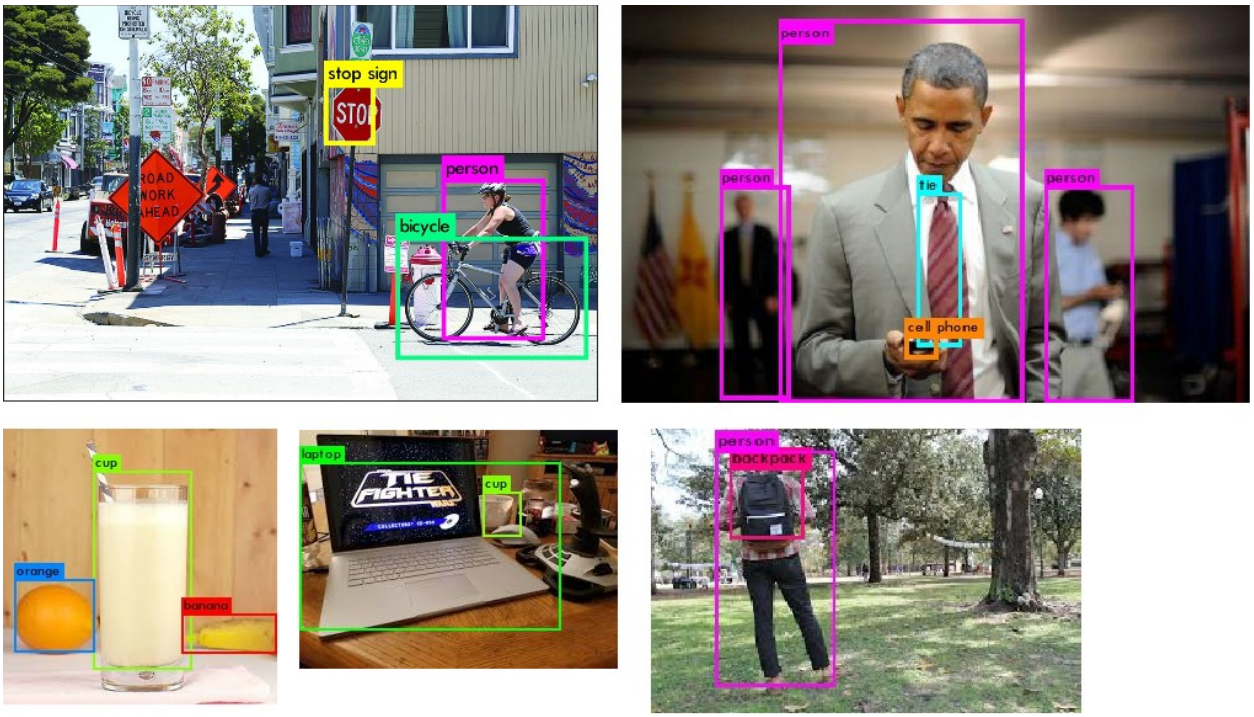

Image 7: Results with the 10-class model 7 on random images found on google. The trained classes were person, bike, stop, cell, laptop, tie, backpack, cup, banana and orange. Images were created using the darknet demo function.

Reflection

The question I wanted to answer with this project was whether it was possible to develop an CNN based object detector which would run in 20 fps on a Raspberry Pi with a decent quality. Research on the internet showed that the performance of the Raspberry Pi is too weak to do it by itself therefore a solution with the Movidius NCS was followed. I used the C-based darknet framework for training because it already proved that it was capable to create small and fast detectors with competitive quality. For my custom models I used annotated images from the COCO dataset and created two subsets of training data: one which only contained the 'person' class and a second one which contained besides 'person' nine other classes. I developed different CNN models which would run on the NCS in approximately 50ms (20 fps) and trained them using darknet and the COCO training data. To compare the models I used the mAP metric to finally choose the best models.

One interesting aspect of my project has been the optimization of the demonstrator application. It started with a rather simple python application using OpenCV for image processing and rendering and ended in a multithreaded C++ console program which does the rendering in a low level way. Although this might not directly be related to this machine learning course it was still a very gratifying experience to finally having created a tool which not only claimed to work in real-time internally but actually showed the results in real-time. This might be an important aspect if a future demonstrator is set up in my company with a live-view camera.

During my research there occurred one problem with the Movidius NCS which almost made me quit the project. My early models had a decent mAP and would perform quite well on some test images using the darknet and the Caffe framework. However, on the NCS I would get random bounding boxes and not nearly the quality I expected. Luckily Intel provides a verification tool `mvNCCheck` where the output of the NCS is compared to the output using Caffe. Using this tool I could pinpoint the problem to one problematic layer which was the very first convolution layer. After doing some tests it turned out that due to a bug in the NCS firmware convolutional layers with a channel size of less than 16 would behave erraneously.

I must say that I'm very content with the results of my research. I proved that the Movidius NCS is powerful enough to run a people detector on a low powered platform in real-time. While a multi-class model might not yield the necessary detection and classification power I'm quite sure that the quality of the 1-class model is sufficient to develop applications which are based on tracking people.

Improvement

In chapter Analysis I discussed class imbalance in the COCO database and that I dismissed the idea of a balanced training set because the AP of the 'person' class would decrease. However, the results I used to make my point also show that a balanced set would improve the AP of other classes. A further evaluation with a 'semi-balanced' set where the people class is not as strongly represented might give an improvement for the multi-class case.

It was shown that raising the number of layers were overall beneficial. Models with even more layers might yield even better results.

One issue YOLO has is that the resulting bounding boxes often do not fit very well around the detected object. The reason is that the final size of the last layer is set to 13x13. I haven't changed this setting during my research but a finer grid size of 26x26 might result in better fitting bounding boxes.

Another potential improvement could be to do the rescaling of images differently: during training and inference the rescaling is done in a way that it would distort the image. I believe that better results would be possible if rescaling is done by keeping the aspect ratio. I base my assumption on better visual results I got when running the demo with scaled but undistorted images (where model was trained with scaled and distorted images).

And finally there are other options for fast object detection than YOLO. The Single Shot Detector (SSD) is a popular detector which offers better mAP on COCO than YOLO but slower run-time. There exists a port to the Movidius NCS [AR2018] which runs in 4 FPS and there exist also efforts to make SSD smaller [TSSD2018]. I believe this could be an interesting research topic.

Bibliography

PW2014: Pete Warden, How to optimize Raspberry Pi code using its GPU, 2014, https://petewarden.com/2014/08/07/how-to-optimize-raspberry-pi-code-using-its-gpu/

BNET2017: , the object detection of YOLO V2 can be run on Raspberry PI 3? , , https://groups.google.com/a/dt42.io/d/msg/berrynet/dPk_mnnLIIg/QwbVSfpNDwAJ

TYNCS2017: , Tiny YOLO on NCS, , https://ncsforum.movidius.com/discussion/218/tiny-yolo-on-ncs

DNET: Joseph Chet Redmon, Darknet: Open Source Neural Networks in C, 2017, https://pjreddie.com/darknet/

YOLOv2: Redmon, Joseph and Farhadi, Ali, YOLO9000: Better, Faster, Stronger, 2016, arXiv preprint arXiv:1612.08242

MVCAFFE: Intel, Caffe Layer Support, 2017, https://movidius.github.io/ncsdk/caffe.html

YOLOv2URL: Joseph Chet Redmon, YOLO: Real-Time Object Detection, 2016, https://pjreddie.com/darknet/yolov2/

AB2018: AlexeyAB, How to train (to detect your custom objects), , https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects

CB2017: Christopher Bourez, Bounding box object detectors: understanding YOLO, You Look Only Once, 2017, http://christopher5106.github.io/object/detectors/2017/08/10/bounding-box-object-detectors-understanding-yolo.html

MVINST2016: Intel, Intel Movidius NCS Quick Start, 2016, https://developer.movidius.com/start

GIL2018: Abhinav Ajitsaria, What is the Python Global Interpreter Lock (GIL)?, 2018, https://realpython.com/python-gil/

RC2012: Raspberry Compote, Low-level Graphics on Raspberry Pi , 2012, http://raspberrycompote.blogspot.de/2012/12/low-level-graphics-on-raspberry-pi-part_9509.html

AR2018: Adrian Rosebrock, Real-time object detection on the Raspberry Pi with the Movidius NCS, 2018, https://www.pyimagesearch.com/2018/02/19/real-time-object-detection-on-the-raspberry-pi-with-the-movidius-ncs/

TSSD2018: Alexander Wong et al, Tiny SSD: A Tiny Single-shot Detection Deep Convolutional Neural Network for Real-time Embedded Object Detection, 2018, https://dblp.org/rec/bib/journals/corr/abs-1802-06488