POLITECNICO DI TORINO

Master degree in Mathematical engineering

Master degree Thesis

Object detection from onboard vehicle camera

Supervisors

Prof. Tatiana TOMMASI

MSc. Anders EGEHOLM

MSc. Anders MOVERT

Candidate

Ludovico BESSI

1st December 2021

Summary

The purpose of this thesis is threefold. First, DEtection TRansformer (DETR) network performance is evaluated against Single shot detector Resnet50 FPN (SSD) on an ensemble of common open source datasets. Based on this baseline work, a novel algorithm for detecting "interesting" events is developed and evaluated. It is based on the combination of two different object detection models trained on the same data. On one hand a standard object detection model trained to recognize 10 different labels. On the other hand, a smaller mobile friendly network trained to recognize if a given object is there or not, called No Label Network (NOLAN). The standard object detection model is the SSD model outlined above, while the mobile friendly network is the SSD MobileNet.
Secondly, a brand new lightweight Linear encoder decoder (LEDD) object detection model to improve performance on small objects is proposed.
Lastly, software routines leveraging "Qualcomm Neural Processing SDK for AI" have been developed to efficiently deploy trained models on qualcomm chips present in the car.
This thesis is a joint work between: Politecnico di Torino, Volvo Cars and Zenseact.

"To my family"

Table of Contents

List of abbreviations

COCO: Common Objects in Context dataset

DETR: Detector transformer

DINA: Detect interest algorithm

ExDARK: Exclusively dark images dataset.

IoU: Intersection over union

LEDD: Linear encoder decoder detector

LISA: Laboratory for intelligent and safe automobiles.

mAP: Mean average precision

mIoU: Mean intersection over union

NOLAN: No label network

PLA: Positive label accuracy

SNPE: Snapdragon neural processing engine

SSD: Single shot detector

Chapter 1

Introduction

1.1 Main goals

Object detection in an industry project requires to focus not only on common model evalutation metrics, but also to take into account scalability and deployment issues. For this reason, four different aspects have been explored.
First of all, the right model needs to have a good performance on a set of different metrics. On this note, DETR architecture is evaluated against SSD architecture on overral accuracy and inference speed.

Based on this baseline work, a novel algorithm called DINA for detecting interesting events is developed and evaluated. The definition of detecting an interesting event is the possibility to detect an object given a model that has never seen such object before. This is of interest because there are many different unexpected situations in the environment where the models are deployed. It is based on the combination of two different object detection models trained on the same data. On one hand a standard object detection model is trained to recognize 10 different labels. On the other hand, a smaller mobile friendly network called NOLAN is trained to recognize if a given object is there or not and along side its position.

Object detection models for vehicles need to be deployed on actual cars.

In this setting, two issues need to be explored. First of all, a model needs to be as light weight as possible to fit on on-board hardware. Secondly, a model needs to have the right format for the hardware on top of being as optimized as possible for inference speed.

To tackle the first issue, the LEDD model is proposed. Not only it has fast inference, but It is also extremly portable. As an added benefit, It performs well by design on small images.

For the second issue, software routines leveraging "Qualcomm Neural Processing SDK for AI" library have been developed to efficiently deploy trained models on qualcomm chips present in the car. On top of that, the possibility of quantizing the models to reduce their size is also showcased.

1.2 Tools and frameworks used

The training and evaluation of the models has been carried out locally on a NVIDIA Quadro M4000 GPU. The models have been trained using Tensorflow v2.5.1. Due to GPU constraint, the batch size has been fixed to 1. Different frameworks have been used to train the models, namely:

- Tensorflow object detection API [1] for the SSD based architectures.
- Vanilla tensorflow for DETR model, leveraging the open source framework developed by "Visual beahviour". [2]
- Keras APIs [3] for the LEDD model.

Chapter 2

Data available

2.1 Data set for standard object detection

Deep learning models are data-hungry [4] and object detection models are no exceptions. For this reason, 45.6 GB of image data has been used. The data comes already annotated from 5 different Open source datasets:

- Open Image Data [5]: a dataset of 9M images annotated with image-level labels, object bounding boxes, object segmentation masks, visual relationships, and localized narratives.
- COCO2017 [6]: COCO is a large-scale object detection, segmentation, and captioning dataset.
- LISA Traffic sign dataset [7]: The LISA Traffic Sign Dataset is a set of videos and annotated frames containing traffic signs.
- Wider Pedestrian dataset [8]: The WiderPerson dataset is a pedestrian detection benchmark dataset in the wild, of which images are selected from a wide range of scenarios, no longer limited to the traffic scenario.
- ExDARK [9]: The Exclusively Dark dataset is a collection low-light images from very low-light environments to twilight.

The purpose of using a blend of different data sources it to bridge the gap between the environment in which the training data was collected and the environment where the models are deployed. This is especially crucial for delicate problems as object detection on cars.

However, combining data from such diverse sources hinders the learning capabilities of models, mainly because the data is not independent and identically distributed [10]. Only a subset of objects have been selected from these datasets. Namely, only images containing the following 10 labels have been selected: person, cat, truck, dog, bycicle, bus, car, motorbike, traffic light, stop sign. Multiple objects with different labels can be encounterd in a single image. These particular objects are of interest at Volvo Cars given that every one of them can be encountered while travelling. Other labels are potentially of interest, but they are not included for two reasons. The first reason is the lack of enough annotated for particular labels, e.g peculiar Swedish traffic signs. The other reason is the limited computing power: enlarging the dataset would make the training and evaluating process much harder. The five different datasets have different annotations styles. Every image annotation is then converted in the following format: [filename, label, xmin, ymin, xmax, ymax]. All in all, this amounts to a total of 189874 images. The images have been splitted in three datasets: training (75%), validation (15%) and test data (10%).

2.2 Dataset for NOLAN

In order to evaluate the detection capabilities of out of sample data, an additional dataset is built. It is simply made up of 1000 images of Deers coming from the Open image dataset [5]. This smaller dataset is only used at testing time.

2.3 Feeding the models: how to load the data

Dataset of such size does not fit into memory, so a pipeline needs to be in place to overcome this issue. In this thesis, this is done in two different ways depending on the framework used.

When using Tensorflow object detection API, that is with SSD model and No Label Network, a special **TFRecord** file that holds all the annotations and points to image data is created. Then, for every epoch we just need to iterate through the file and load images in memory one by one.

When using vanilla Tensorflow, a **tf.Dataset** is created. It's an iterator that holds information about a given image at every pass. The iterator is built one time before the training begins, again images are loaded one by one.

Chapter 3

Object detection models employed

3.1 Single shot detector Resnet50 FPN (SSD)

The first model that has been tried is the Single shot detector ResNet50 FPN (SSD) [11]. It is a single state object detection model. The most innovative innovations of Retina networks is the addition of a Feature Pyramid Net (FPN) on top of the ResNet feature extractor.

3.1.1 Architecture

The architecture of the model is outlined below:

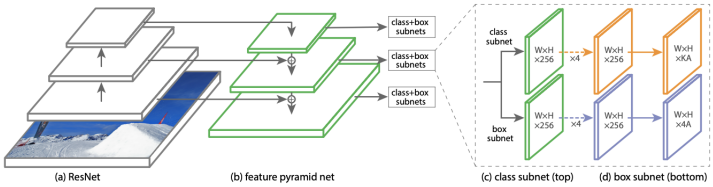


Figure 3.1: RetinaNet architecture from [12]

The backbone is responsible for computing a convolutional feature

map over an entire input image and is an off-the-self convolutional network. In brief, FPN augments a standard convolutional network with a top-down pathway and lateral connections so the network efficiently constructs a rich, multi-scale feature pyramid from a single resolution input image. Each level of the pyramid can be used for detecting objects at a different scale. FPN improves multi-scale predictions from fully convolutional networks (FCN). Still, It must be remembered that small objects with low area and high aspect ratio are an open problem. After that, the first subnet performs convolutional object classification on the backbone's output; the second subnet performs convolutional bounding box regression. The two subnetworks feature a simple design specifically for one-stage object detection.

3.1.2 Training

The total loss function is defined as the sum of a localization loss and classification loss normalized by the number of box matches. The localization loss is the weighted L1 loss, defined pointwise as:

$$WL1(x) = \begin{cases} \frac{1}{2} * x^2 & |x| < \delta \\ \delta * (|x| - \frac{1}{2}\delta) & |x| \geq \delta \end{cases}$$

In this work, delta has been set to 1.0.

The classification function is the weighted cross entropy function. The class imbalances are used to create the weights for the cross entropy loss function ensuring that the majority class is down-weighted accordingly. Only predictions from positive matches are penalized. The two loss values are summed together with equal weight of value 1.0 The model has been trained with two epochs with Momentum optimizer with a base learning rate of 0.025 along side cosine decay learning rate with warmup learning rate of warmup learning rate: 0.0133.

3.2 No Label Network (NOLAN)

The No Label Network is based on the SSD MobileNet architecture [13]. The pretrained model has been used, resetting the classification layer with only 1 class. The reason for this choice is that with this model we are only interested in detecting if a given object is there or not, alongside its position.

3.2.1 Architecture

Mobile net is based on a streamlined architecture that uses depth wise separable convolutions to build lightweight deep neural networks.

Depthwise separable convolutions are a form of factorized convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution. The depthwise convolution applies a single filter to each input channel. The pointwise convolution then applies a 1×1 convolution to combine the outputs of the depthwise convolution. A standard convolution both filters and combines inputs into a new set of outputs in one step. The depthwise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining. This factorization has the effect of drastically reducing computation and model size.

Depthwise convolution is extremely efficient relative to standard convolution. However it only filters input channels, it does not combine them to create new features. So an additional layer that computes a linear combination of the output of depthwise convolution via 1×1 convolution is needed in order to generate these new features.

Below, the architecture is visualized:

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 3.2: Architecture table from [13]

In the image above, "dw" stands for depth wise. All layers are followed by a batchnorm and ReLU nonlinearity with the exception of the final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification. In the MobileNet architecture, two hyperparameters have great importance: width and resolution multiplier. The width multiplier $\alpha \in (0,1]$ is to thin the network uniformly at each layer. Width multiplier has the effect of reducing computational cost and the number of parameters quadratically by roughly α^2. Given that the model is already reasonably lightweight compared to others model, $\alpha = 1$ has been set. The resolution multiplier ρ is applied to the input image and the internal representation of every layer is subsequently reduced by the same multiplier. In practice we implicitly set ρ by setting the input resolution when defining the model.

3.2.2 Training

The total loss function is defined as the sum of a localization loss and classification loss. The localization loss is the weighted L1 loss, defined pointwise as:

$$WL1(x) = \begin{cases} \frac{1}{2} * x^2 & |x| < \delta \\ \delta * (|x| - \frac{1}{2}\delta) & |x| \geq \delta \end{cases}$$

In this work, delta has been set to 1.0.
The classification function is the weighted cross entropy function. In this work, the weight given to the classification function is 0 given that we only have one label "object". On top of that, the total loss is not normalized by the number matches.

The model has been trained with from the pretrained opensource model of the tensorflow object deteciton library, resetting the classification and box regression heads. The training consists of two epochs with Momentum optimizer with a base learning rate of 0.8 along side cosine decay learning rate with warmup learning rate of warmup learning rate: 0.133.

3.3 DEtection TRansformer (DETR)

DETR is a new technique for object detection. Instead of relying on many hand designed components like models of the SSD family, the detection pipeline is streamlined. This is achived by viewing the problem directly as a set prediction problem.
The main innovations of the DETR model are:

1. Set based global loss that forces unique predictions by bipartite matching.

2. Transformer encoder decoder architecture.

The overral architecture is also surprisingly simple, as shown below:

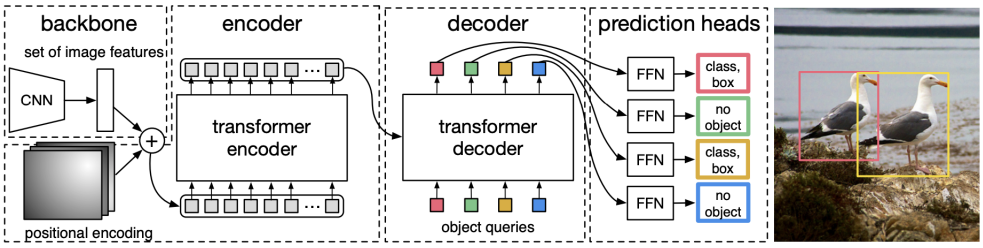


Figure 3.3: DETR architecture, image from [14]

As with all common object detection models, a CNN backbone is used to learn a 2D representation of an input image. After flattening, it is supplemented with a positional encoding before passing it into a transformer encoder. After that, a transformer decoder takes as input the encoder output alongside a fixed number of positional embeddings. With this information, It is possible to predict either a detection or a "no object" with a shared feed forward network (FFN). The (FFN) is made up of a 3-layer perceptron with ReLU activation function and a linear projection layer. The 3-layer perceptron predicts the normalized center coordinates, height and width of the box with respect to the input image. The linear layer predicts the class label using a softmax function. Given that a fixed-size set of N of bounding boxes is predicted, with N much larger than the actual number of objects of interest, an

additional special class $\emptyset$ is used to represent that no object is detected within a slot. This technique plays the exact role of the "background" classs in standard object detection approaches. In the next two sections, a more detailed explanation of the innovative features of this network will be explained.

3.4 Set prediction loss

The global loss of the DETR model can be seen as an optimal bipartite matching function that takes into account both label and box position. Let y the ground truths, while $\hat{y}$ the predictions. An optimal bipartite matching is found between the two sets using a matching function across all N permutations, where N is number of predictions plus the number of "no object" predictions. This optimal matching is defined as:

$$\hat{\sigma} = \arg\min_{\sigma} \sum_{i=1}^{N} \mathcal{L}_{match}(y_i, \hat{y}_{\sigma(i)})$$

The $\mathcal{L}_{match}$ is a pair wise matching cost computed with the Hungarian algorithm [15]. Each prediction and ground truh can be seen as a tuple: $y_i = (c_i, \mathbf{b_i})$ where c_i is the label and $\mathbf{b_i}$ is the bounding box. Then, given $\hat{p}_{\sigma(i)}(c_i)$ as probability that match $\sigma(i)$ has class c_i, the matching cost is defined as:

$$\mathcal{L}_{match} = -\mathbf{1}_{\{c_i \neq \emptyset\}} \hat{p}_{\sigma(i)}(c_i) + \mathbf{1}_{\{c_i \neq \emptyset\}} \mathcal{L}_{box}(y_i, \hat{y}_i)$$

Then, the total loss is defined as:

$$\mathcal{L}_{total}(y, \hat{y}) = \sum_{i=1}^{n} [-\log \hat{p}_{\sigma(i)}(c_i) + \mathbf{1}_{\{c_i \neq \emptyset\}} \mathcal{L}_{box}(y_i, \hat{y}_i)]$$

The only bit missing is the definition of the Loss on the function, which is:

$$\mathcal{L}_{box}(y_i, \hat{y}_i) = \lambda_{gIoU} \mathcal{L}_{gIoU}(b_i, \hat{b}_{\sigma(i)}) + \lambda_{l1} ||b_i - \hat{b}_{\sigma(i)}||_1$$

With λ_{gIoU},$\lambda_{l1} \in \mathbb{R}$ hyperparameters, $\mathcal{L}_{gIoU}$ is the generalized intersection over union (gIoU) [16] and $||\cdot||_1$ is the L1 norm. The gIoU is a small

variation of the standard intersection over union (IoU). Considering two boxes A and B and the smallest box C enclosing them, It is defined as:

$$gIoU = IoU - \frac{|C \setminus (A \cup B)|}{|C|}$$

In this way, It can be used as a loss function given that it is non zero even though the two boxes do not intersect. In that case, It becomes a normalized measure of the empty space between the two boxes.

3.5 Transformer based architecture

The DETR model employs the attention mechanism pioneeder by [17], with one difference. Instead of using an autogressive decoder, the matching loss function uniquely assigns a prediction to a ground truth object and is invariant to permutation of predicted objects, so the model can emit the outputs in parallel. Thus, inference time is reduced. An overview of the inner workings of the Transformer architecture is given by [17].

3.6 Transformer encoder

First, a 1x1 convolution reduces the chanel dimention of the high level activation map from C to a smaller dimension d, creating a feature map $z_0 \in \mathbb{R}^{d \times H \times W}$. The encoder expectes a sequence as input. For that reason, hence the spatial dimension of the tensor is collapsed to one dimension, reasulting in a $d \times HW$ feature map. Since the transformer architecture is permutation-invariant, fixed positional encodings are added to the input of each attention layer.

3.7 Transformer decoder

N embeddings of size d are transformed using multi-headed self and encoder-decoder attention mechanism. Given that also the decoder is

permutation-invariant, the N input embeddings must be different to produce different results. For this very reason, similarly to the encoder step, positional encodings are learnt and added to the input of each attention layer.

3.8 Training setup

In this work, a DETR model has been finetuned using the DETR open source library from Visual Behaviour [2], which mimics the original Pytorch library. Weights pretrained on the COCO dataset have been used. A key difference from the original implementation is that the images have fixed size, that is they have been resized to 640x640 images to make comparison easier to models of the SSD family. Similarly to the workflow of SSD based models, the training has run for two epochs, starting from pretrained weights and resetting classification and box regression heads. In this case, the first epoch acts as a warmup: only the last layers using for label predictions are trained. Instead, n the second epoch the whole architecture is trained. The learning rate for the transformer layers is set to $1e-4$ while the learning rate for the classification layer has been set to $1e-3$ with square root decay on the epoch.

3.9 Linear encoder decoder detector

Many state of the art object detection models share one common shortcoming: they usually fail at detecting small objects with high aspect ratios [18]. However, detecting small objects is especially important for vision models deployed on cars: being able to detect small objects means detecting objects sooner rather than later.

With this in mind, a new model called Linear encoder decoder detector (LEDD) is proposed. Its architecture is outlined below:

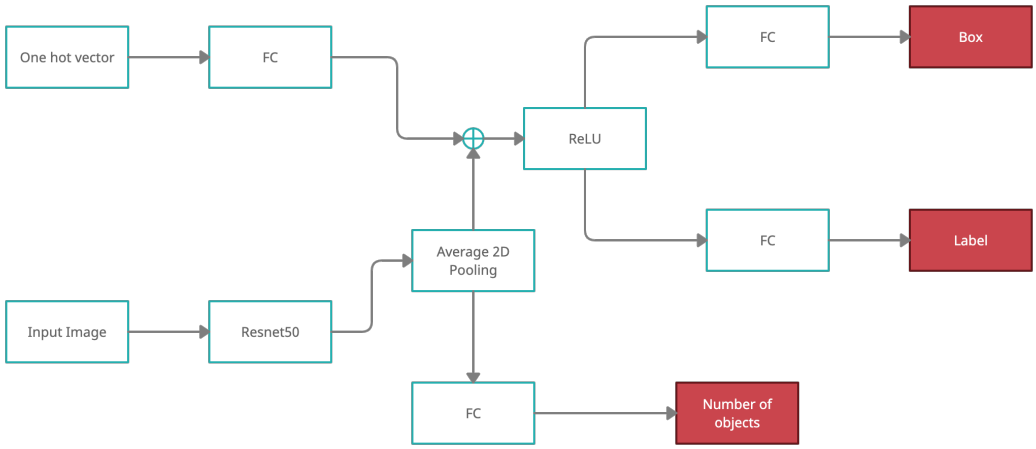


Figure 3.4: LEDD architecture

The outputs are outlined in red: number of objects along side box and label for each object. The network can be seen as the combination of two smaller networks.
On one hand, on the bottom side of the diagram, the output of the feature extractor ResNet50 is passed through a fully connected layer to find the number of objects, using a softmax activation function.
On the other hand, for each predicted object, the image feature tensor is summed with the position feature tensor. After that, a ReLU activation function is applied before predicting the associated box and label.
The bias towards smaller objects comes from the fact that during training objects are sorted from smaller size to larger size: the network then learns that the first objects are the smallest and It will carry over at

inference time.
Apart from explainability, the network "split" is crucial for another issue, the argmax function is non differentiable. There are many functions commonly used in neural networks, such as ReLU, max, maxout pooling and max pooling. However, these functions are continuous and almost every where differentiable: this is enough to use them with gradient descent optimization. However, this is is not the case for the argmax function.
In a training setting, this is crucial because a non differentiable function inside a neural network breaks backpropagation because it is based on gradient computations. It is still possible to use such functions, however the gradient would need to be approximated using smapling techniques, such as REINFORCE [19]. The drawback is that the training step is computationally more expensive, plus It's generally harder to train a network in this way.
Instead, thanks to the network split, the argmax function is not directly used inside LEDD and does not cause any harm to the training procedure.
The model is compiled with two different losses. For box classification and number of objects prediction, the sparse categorical cross entropy loss is used. While for the loss for the box regression is L1 loss. Similarly to the DETR model, a linear combination of L1 loss and generalized intersection over union could be used.

Chapter 4

Detecting interesting events

In this chapter, a novel algorithm for detecting *interesting* events is developed. An event is defined as *interesting*, if a given detection model is not able to detect an object in a given image.
When a given frame of the camera is fed into the model f, there are two different situations at play that are of interest in this setting:

1. The image contains an object that model f should recognize, but it's *unable* to.

2. The image contains an object that model f has never seen before and so it's unable to spot it.

Spotting failures of a deployed model is vital, so situation 1 is of interest. Furthermore, a variety of different objects is of interest specifically for situations where data is not abundant: a stop sign partially occluded by leaves or a bike attached to the back of a car. The approach in the following section addresses the two situation of above. It is based on the combination of two different object detection models trained on the very same dataset.

4.1 Detect-interest algorithm (DINA)

Detect-interest algorithm (DINA) is based on the combination of two different object detection models trained on the same data. On one hand a standard object detection model (f) is trained to recognize the labels present in the dataset. On the other hand, a smaller mobile friendly object (g) detection model is trained to recognize an entity "object", withouth assigning labels to them. In this setting, the algorithm works as follows. The input image A is fed to both model f and g. There are four cases:

1. Neither model f neither model g detect anything.
2. Model f detects an object or multiple objects and model g does not detect an object.
3. Model g detects an object or multiple objects and model f does not detect an object.
4. Both model f and model g detect an object or multiple objects.

From this point on, we assume there is at maximum one object per image without losing generality. This is shown by the following construction: suppose there are N objects in an image, and a given model is able to detect M objects. Then, It is possible to construct a NxM matrix L with $L_{ij} = 1 - IoU(obj_i, obj_j)$. After this, two bounding boxes can be matched using the linear assignment algorithm [15] with weight given by matrix L. From here, It is possible to reason object-by-object. Case number 1 and 2 are of no no particular interest as either there is nothing notable in the frame (case 1) or model f is able to properly detect the object of interest (case 2).
Case number 3 is more interesting: there is an object in the frame that model f is not able to recognize, while g can. This needs further investigation: either model f is unable to an object that it should detect or there is an object that the model has not trained on. In the first case, model g acts as a discriminator: It points out label where the training could improve. In the second case, model g is suggesting

that there is possibly a new object of interest in the image that needs manual investigation to be assessed.
Following the construction outlined above, It is possible to reason on an object by object basis and conclude that case 4 is a subcase of 3. A sketch of the pseduocode is shown below: The main advantages of the

Algorithm 1 DINA algorithm. A is the input image, f and g are the two models defined as above, ϵ is the tolerance on the IoU.

```
procedure DINA(A, f, g, ϵ)
    for objects do
        Match bounding boxes
    end for
    for matches do
        If match score < ϵ, found a new object / label failure
    end for
    return index of boxes where the anomaly has been found.
```

proposed algorithm are its simplicity and double purpose. Elaborating on the latter, even if model g is not able to detect never seen objects, which admittedly is a very hard task; this procedure can still be used to assess a given deployed model, which is crucial for practioners [20]. In order not to overload on board GPU, the model assessment can be done in the Cloud. In this way, the heaviest model is not deployed and the heavy lifting is not done in real time on the car.

Chapter 5

Deploying the models

Deploying trained models in production is one of the core objectives of this thesis. Volvo test cars have Qualcomm Snapdragon chips on board that are specifically allocated for computer vision tasks. In this setting, deploying a model in production is of particular technical interest because the amount of space occupied by a model on the GPU must be minimized without compromizing accuracy, while inference speed must be at an all time high. The most straightforward solution would be deploying the exported model for inference, analogously to what It is done for inference on a personal computer. A major drawback of this approach is that embedded devices may not have a Python interpreter which makes any exported model useles. Thankfully, Tensorflow graphs [21] in combination with the Snapdragon Neural Processing Engine (SNPE) [22] can be leveraged to satisfy the requirements.

5.1 Tensorflow graphs

In Tensorflow, graphs are data structures that contain a set of operations objects, which represent units of computation and Tensor objects, which represent the units of data that flow between operations. [21] Graphs are employed mainly for two reasons: flexibility of the different environment they can be deployed in and enhanced inference speed. When working with graphs, It is indeed possible to separate subparts of a computation that are independent and split them between threads or devices thus

achieving more speed. Furthermore, common sub-expression can be eliminated by simplifying arithmetic operations.

5.2 Snapdragon Neural Processing Engine (SNPE)

The Snapdragon Neural Processing Engine (SNPE) is a Qualcomm Snapdragon open source software accelerated runtime for the execution of deep neural networks. In short, It makes possible to use the Qualcomm Hardware in combination with common Neural networks frameworks, such as: Caffe, ONNX and Tensorflow. Unfortunately, It has many limitations:

- It can only be used on Ubuntu 18.04 and Python v3.6
- It is only tested on Tensorflow v.1.6 or Tensorflow v2.3
- Not all layers are supported. If a layer is not supported, It can be defined in SNPE as a user defined operation (UDO) implementation, in the form of dynamic libraries that can be queried, loaded and exercised to execute inference using kernels defined within them.

The workflow is outlined below: As the above image suggests, After finetuning a model, three different steps are needed:

1. Creating a model file
2. Converting the model file to .dlc file
3. Model quantization

In the next sections, the software routines to create the "Model file" and the .dlc file will be outlined. Model quantization worked out of the box so no software routine is required.

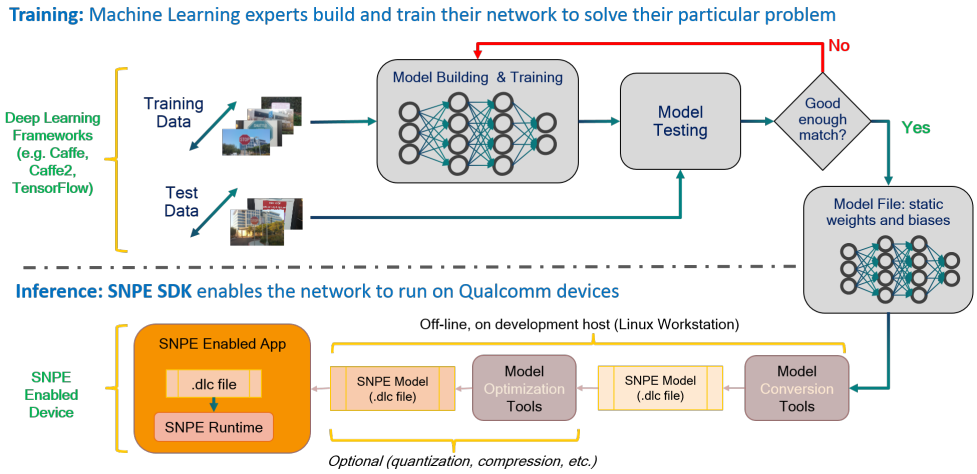


Figure 5.1: SNPE workflow, source: [22]

5.2.1 Creating a model file

At this point, suppose a model is already trained and exported in Tensorflow in the standard Saved model format. Now, It needs to be turned into a tf.Graph before passing it to the snpe-tensorlofw-to-dlc routine. To do so, the following script needs to be used:

content/scripts/create_graph.py

```
from inspect import signature
import os
import pathlib
from numpy.lib.npyio import fromregex
import tensorflow as tf
from tensorflow import keras
from tensorflow.python.framework.convert_to_constants import
    convert_variables_to_constants_v2
import numpy as np
from tensorflow.python.framework.convert_to_constants import
    convert_variables_to_constants_v2
from tensorflow.python.tools import
    optimize_for_inference_lib

loaded = tf.saved_model.load(PATH_TO_SAVED_MODEL)
infer = loaded.signatures['serving_default']
```

```
f = tf.function(infer).get_concrete_function(input_tensor=tf.
    TensorSpec(shape=[1, 640, 640, 3], dtype = tf.uint8))
f2 = convert_variables_to_constants_v2(f)
graph_def = f2.graph.as_graph_def()
optimize = True
if optimize:
    for i in reversed(range(len(graph_def.node))):
        if graph_def.node[i].op == 'NoOp':
            del graph_def.node[i]

    for node in graph_def.node:
        for i in reversed(range(len(node.input))):
            if node.input[i][0] == '^':
                del node.input[i]

    graph_inputs = [x.name.rsplit(':')[0] for x in f2.inputs]
    graph_outputs = [x.name.rsplit(':')[0] for x in f2.
    outputs]
    graph_def = optimize_for_inference(graph_def,g_i,g_o,tf.
    uint8.as_datatype_enum)

print("Saving optimized graph:")
with tf.io.gfile.GFile('graph.pb', 'wb') as f:
    f.write(graph_def.SerializeToString())
```

The above script is inspired from [23] and [24] In short, from a given saved model the signature keys to functions mapping is extracted. After, they are turned into a tf.function from which we extract the concrete function definition. Essentialy, a concrete functon has its own graph specialized for a particular combination of inputs, which improves performance. Finally, we can extract the graph-like definition and prune undeed nodes which do not influence inference. Interesting enough, an optimization of the graph has been already performed here. This point is crucial: if the nodes are not pruned at this point, then calling snpe-tensorflow-to-dlc routine out of the box does not work on models trained with Tensorflow v2.5 on Ubuntu 18.04 on version SNPE v1.5. Either It creates a disconnected graph or enters an infinite loop. Again, It needs to be pointed out that this specific version of Tensorflow is not tested. However, the version used with SNPE must be the same

used for training the model.

5.2.2 Converting a model file to .dlc file

At this point, It is only needed to pass the graph alongside the input dimension, input name and output name(s) to the snpe-tensorflow-to-dlc routine. This works out of the box only with models model trained and exported with Tensorflow v2.3. In this particular setting, the problem seems to be with the further optimization that the routine employs on the graph. By removing this functionality, a .dlc file is obtained without problems. Thanks to the manual pruning seen in the section above, a file of dimension 2.2kB is obtained.

5.2.3 Model quantization

The routine snpe-tensorflow-to-dlc routine returns a non quantized DLC file which uses 32 bit floating point representation of network parameters. Instead, quantized DLC files use fixed point representations of network parameters, usually 8 bit weights.
The advantage of using a quantized model is clear: 4x reduction in model size and 4x reduction in memory bandwidth requirementes. However, this is not for free as quantization per definition is lossy, meaning that model accuracy is usually effected.

Chapter 6

Results and discussion

In this section, the 4 finetuned models are evaluated. Single Shot Detector (SSD), DEtection TRansformer (DETR) and Linear encoder decoder detector are compared. Furthermore, the capability of No Label Network to detect never seen objects is evaluated on the dataset composed of 1000 images of deers described in chapter 2

6.1 Metrics for evaluation

In order to discriminate between the models, It is not enough to compare loss values on the test data. Instead, a metric that showcases the detection power needs to be defined. In this work, mean average precision (mAP) will be used. On top of that, average intersection over union and positive label accuracy are going to be used.

6.1.1 (COCO) Mean average precision (mAP)

The mAP for a set of detections is the mean over classes, of the interpolated average precision (AP) for each class [25]. In short, average precision is defined as:

$$AP = \int_0^1 p(r)dr$$

Where $p(r)$ is the precision recall curve. In COCO AP, a 101-point interpolated AP definition is used in the calculation of the integral. One

last definition: AP@[x] indicates the average precision when considering the IoU threshold to be x. That is, a prediction is considered positive if the IoU of the two objects is above x. In COCO, the mAP is eventually then calculated by averaging over multiple APs with IoU thresholds ranging from 0.5 to 0.95 with step size 0.05.

6.1.2 Mean intersection over union (mIoU)

Intersection over union (IoU) has already been defined in the oprevious chapter. Given N true objects in an image and R detected objects, we again use linear assignment algorithm to match boxes, calculate the intersection over union for each object and take the average of the best matches.

6.1.3 Positive label accuracy (PLA)

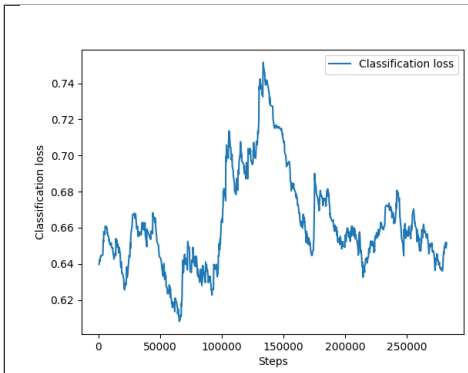
Switching attention to the classification problem, calculating the number of right detections over the number of total detection is of interest (PLA). For example, suppose an image contains two dogs, two cars and one person. If the model predicts one dog, two cars and a truck, then the PLA for the given image would be: $\frac{1+2}{2+2+1} = 0.6$. In the numerator, the number of predicted matched boxes that have the right label while in the denumerator the total number of annotated objects in the image. This value is calculated after matching predicted with true boxes, such that for true box there is a predicted box. In this way, additional non matched boxes with the right labels are not considered, while having a matched box with the wrong label is penalized.

6.2 Results

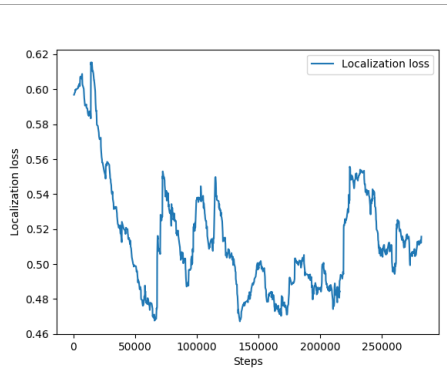
In this section, loss value during training and validation will be plotted. Moreover mAP, mIoU and PLA for the test set are showcased for SSD, DETR and LEDD. The training plots have been smoothed with exponential moving average with weight 0.9, while validation plots have been smoothed with value 0.5.

6.2.1 Single shot detector (SSD) results

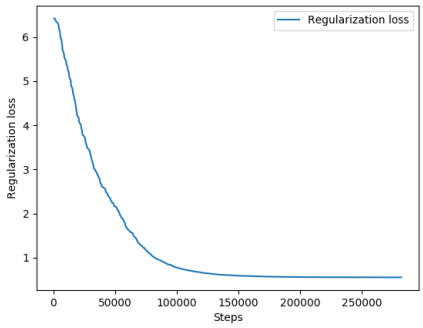
Loss plots on training and evaluation data will be shown. On top of that, loss values are also calculated on the test data, along side the metrics showcased above. Below, training plots are a shown:



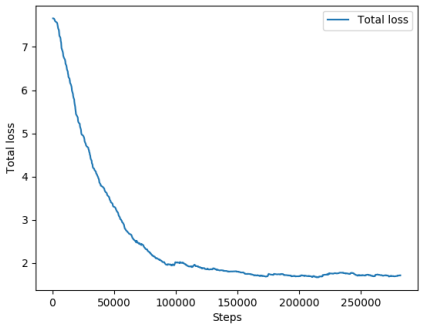
Classification loss train



Localization loss train

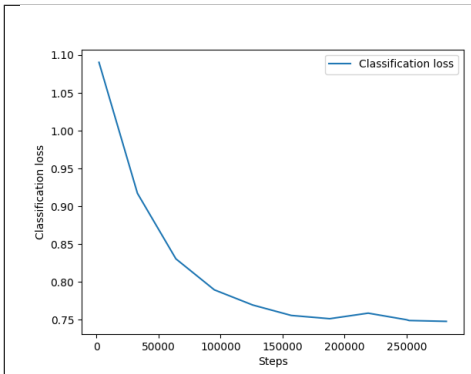


Regularization loss train

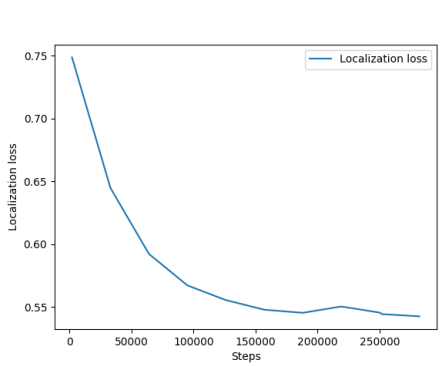


Total loss during train

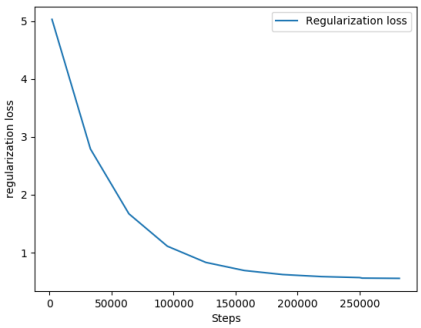
And validation plots:



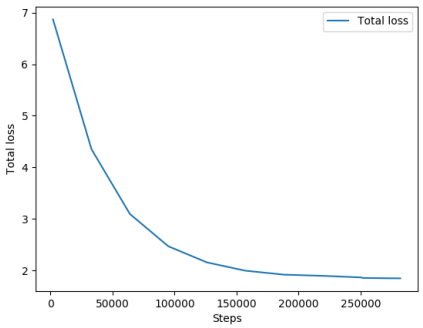
Classification loss validation



Localization loss validation



Regularization loss validation



Total loss validation

Below, test loss values on test and relevant metrics are shown. Alongside that, as a benchmark, the same model with pretrained weights on the COCO Dataset from [11] is evaluated on the same data.

Model	mAP	mAP@0.5	mAP@0.75	mIoU	PLA
SSD	0.0839	0.1158	0.0974	0.09	0.12
Benchmark	0.4272	0.6042	0.4758	0.65	0.89

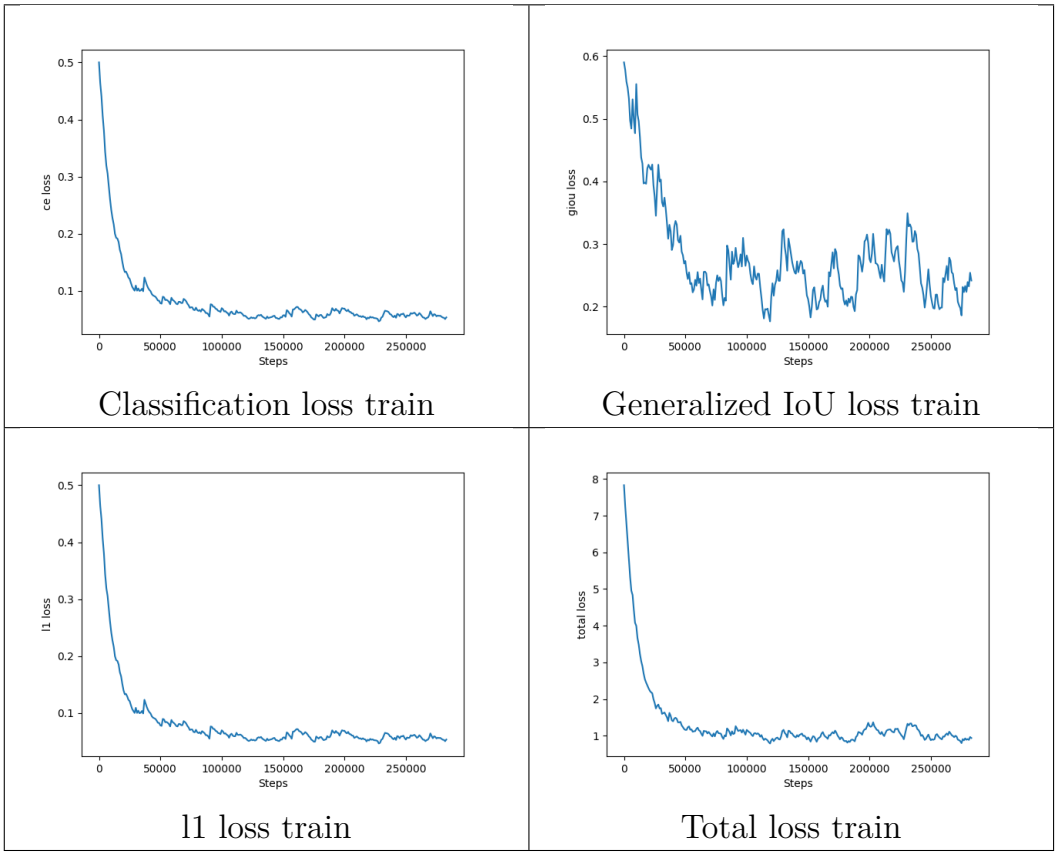
Model	class loss	loc loss	reg loss	tot loss
SSD	0.7415	0.5428	0.5516	1.8336
Benchmark	0.8504	0.1971	0.1761	1.224

Lastly, the average inference time of the SSD model is 46 ms.

6.2.2 DEtection TRansformer (DETR)

Loss plots on training and evaluation data will be shown. On top of that, loss values are also calculated on the test data, along side the metrics showcased above.

Below, training plots are a shown:



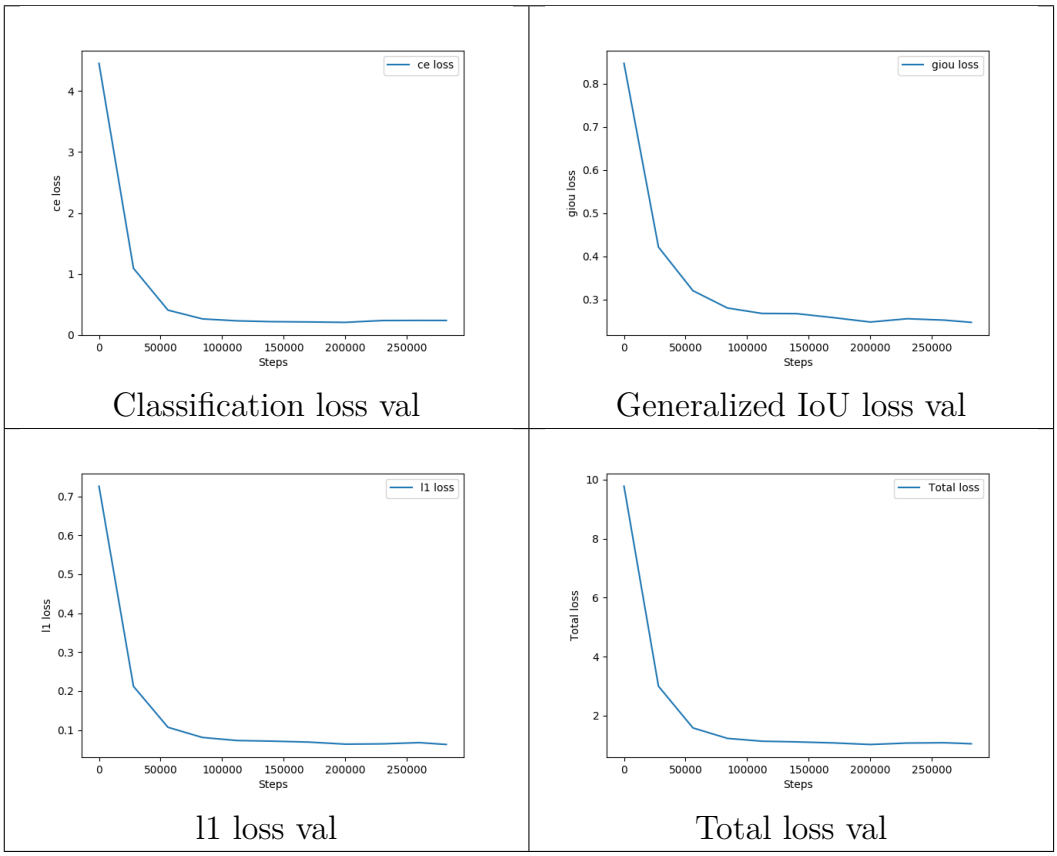
Classification loss train

Generalized IoU loss train

l1 loss train

Total loss train

Below, validation plots are shown:



Classification loss val | Generalized IoU loss val

l1 loss val | Total loss val

Below, test loss values on test and relevant metrics are shown. Alongside that, as a benchmark, the same model with pretrained weights on the COCO Dataset from [14] is evaluated on the same data.

Model	mAP	mAP@0.5	mAP@0.75	mIoU	PLA
DETR	0.3133	0.4565	0.3532	0.33	0.64
Benchmark	0.3568	0.5432	0.3549	0.44	0.86

Model	class loss	giou	l1 loss	total loss
DETR	0.2316	0.2488	0.0612	1.038
Benchmark	0.1823	0.2212	0.0719	0.9832

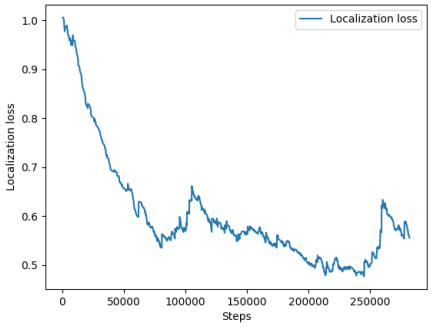
Lastly, average inference time for DETR is 88ms

6.2.3 Linear encoder decoder detector (LEDD)

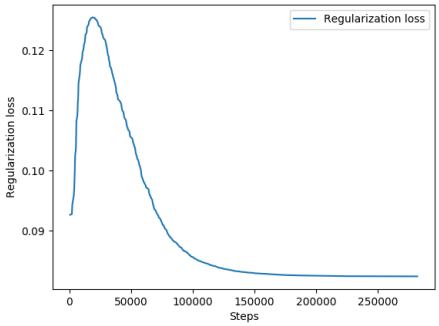
Unlike the other models, the Linear encoder decoder detector has not been fully trained on the dataset. However, the model has been overfitted on a small set of images: showcasing its learnig capabilities.

6.2.4 No Label Network (NOLAN)

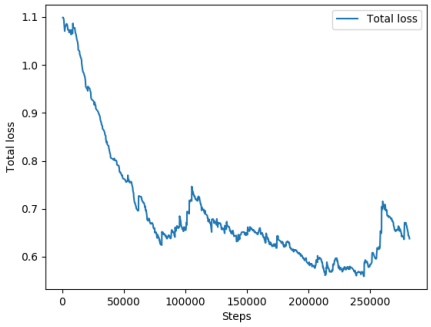
In the No Label Network (NOLAN), the classification loss is not shown as it is constant: there is just one label.
Below, training plots are a shown:



(a)



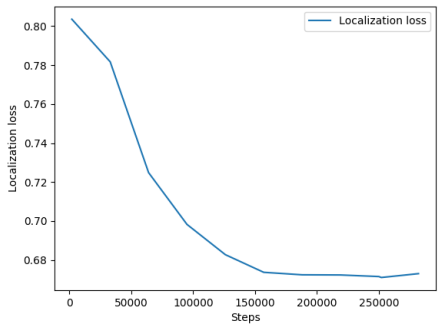
(b)



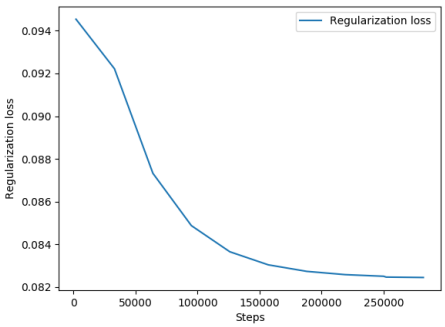
(c)

Figure 6.1: (a) localization loss, (b) regularization loss, (c) total loss

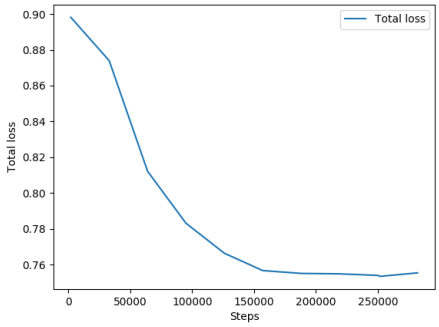
And validation plots: Below, test loss values on test and relevant



(a)



(b)



(c)

Figure 6.2: (a) localization loss, (b) regularization loss, (c) total loss

metrics are shown. In this case however, no benchmark is possible. Instead, the abiliy of the model to detect new objects withouth having seen them before is tested.

Model	mAP	mAP@0.5	mAP@0.75	mIoU	PLA
NOLAN	1.872e-4	9.274e-4	1.3e-5	0.02	1.0

Model	class loss	loc loss	reg loss	tot loss
NOLAN	1.0	0.675	0.08	0.756

Unfortunately, the model has not performed well on the Deer dataset. On average, there are 1.2 deers in the 1000 images but NOLAN model

has not spotted any object with confidence score higher than 0.05. Lastly, the average inference time is 19ms.

6.3 Discussion

The results in the above section show how challenging it is to train detector models of the SSD family. This is mainly due to the sheer number of hyperparameters that needs to be set up and fine tuned, just naming a few: feature extractor depth multiplier, feature extractor minimum depth, weight of the l2 regularizer x/y and height/width scale of the box coder, scales and aspect rations of the anchor generators.

Meanwhile, just two epochs of training for DETR show impressive results, given that the mAP from the benchmakr weights [14]. Lastly, NOLAN approach has turned out in a failure, withouth a single deer detected. Still, as discussed in chapter 4, such a lightweight model can be used to spot undetected images from slower and bigger models. Such an approach is probably ill defined: there are potentially too many objects in a single image to be able to detect **any** object.

Still, the mobile net architecture is well suited to be deployed on actual hardware given the good tradeoffs between inference speed and performance. Given slower inference time, DETR can be instead used for offline detection to support the Data and diagnostic team and for evaluation of the deployed model, as described in chapter 4.

Given that the LEDD model has not been fully trained, making a definitive conclusion about its performance would be bold. For sure, its very definition make it suited for detecting smaller objects, given that the they have been prioritezed during training. Furthermore, its small size is advantegeous because of deployment reasons: It can be coupled with a standard heavy weight model with good mAP to obtain the best from both models.

Chapter 7

Future work

7.1 Future work on the models

There are many different approaches that could be tried to further improve on this inital work. First of all, MobileNet architecture should not be discarded completly given that It has been tried only with the NOLAN approach. It would be especially interesting to check the network performance on a standard object detection problem given that It is especially suited for embedded applications.

Secondly, alternative loss functions could be used for the SSD model described above. Speaking about the box regression problem, generalized intersection over union loss [16] could be employed, which already used and tested on the DETR network with great results [16]. Moving to label classification, the factor $(1 - p_t)^\lambda$ could be multiplied to the standard cross entropy loss function. Setting $\lambda > 0$ reduces the relative loss for well-classified examples ($p_t > 5$), putting more focus on hard, misclassified examples This approach has already been used with [12] with great success.

Moving to the problem of predicting a never seen object. Detecting such out of sample objects has proved to be particularly difficult. Probably, the best way forward is to relax the problem and move to an easier problem: single shot detection. In this framework, an object that

has been seen only once by the model is trying to be detected. The work could be started by leveraging networks for image classification called Siamese neural network [26].

7.2 Future work related to the deployment of the models on real hardware

Unfortunately, no real hardware is available at present time to test the models. However, the only two extra steps needed are related to software engineering and not machine learning:

- 1. Accessing the images captured by the model.
- 2. Transfering the output to the cloud.

On top of that, not every layer is supported by the SNPE toolkit. If one wants to use custom layers, then SNPE requires to have them in the form of User defined operation (UDO). [22] These could be operations defined in popular training frameworks such as Tensorflow or custom operations that are built based as framework extensions but not available in the SNPE SDK. They can be natively executed on any of the supported hardware accelerators for which they are implemented. While the work flow is supposedly seamless, there is some amount of work to be done, as seen by image below:

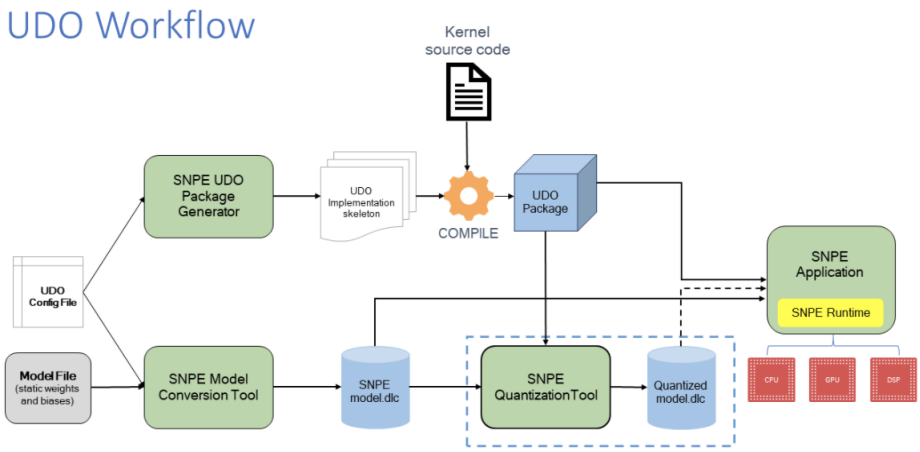


Figure 7.1: UDO Workflow from [22]

SNPE promotes the notion of a 'UDO package' with which a user can easily express the association between the different components of a UDO. This notion is central to all the tools that enable users to create UDO packages to be used in network inference. However, it is to be noted that SNPE still directly interfaces with the various UDO libraries at runtime and not with the UDO package construct. Thus users are free to just build standalone libraries without being strictly bound to this notion of a package.

Bibliography

[1] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 2).

[2] *Visual behavior open source DETR framework*. URL: https://github.com/Visual-Behavior/detr-tensorflow#acknowledgement (cit. on pp. 2, 14).

[3] François Chollet. *keras*. https://github.com/fchollet/keras. 2015 (cit. on p. 2).

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 3).

[5] Alina Kuznetsova et al. «The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale». In: *CoRR* abs/1811.00982 (2018). arXiv: 1811.00982. URL: http://arxiv.org/abs/1811.00982 (cit. on pp. 3, 5).

[6] Tsung-Yi Lin et al. «Microsoft COCO: Common Objects in Context». In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312 (cit. on p. 3).

[7] Mohan M. Trivedi Andreas Møgelmose and Thomas B. Moeslund. «Vision based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey». In: *IEEE Transactions on Intelligent Transportation Systems* (2012) (cit. on p. 3).

[8] Shifeng Zhang, Yiliang Xie, Jun Wan, Hansheng Xia, Stan Z. Li, and Guodong Guo. «WiderPerson: A Diverse Dataset for Dense Pedestrian Detection in the Wild». In: *IEEE Transactions on Multimedia (TMM)* (2019) (cit. on p. 3).

[9] Yuen Peng Loh and Chee Seng Chan. «Getting to Know Low-light Images with The Exclusively Dark Dataset». In: *Computer Vision and Image Understanding* 178 (2019), pp. 30–42. DOI: https://doi.org/10.1016/j.cviu.2018.10.010 (cit. on p. 3).

[10] Wim Casteels and Peter Hellinckx. «Exploiting non-i.i.d. data towards more robust machine learning algorithms». In: *CoRR* abs/2010.03429 (2020). arXiv: 2010.03429. URL: https://arxiv.org/abs/2010.03429 (cit. on p. 4).

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. «SSD: Single Shot MultiBox Detector». In: *CoRR* abs/1512.02325 (2015). arXiv: 1512.02325. URL: http://arxiv.org/abs/1512.02325 (cit. on pp. 6, 28).

[12] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. «Focal Loss for Dense Object Detection». In: *CoRR* abs/1708.02002 (2017). arXiv: 1708.02002. URL: http://arxiv.org/abs/1708.02002 (cit. on pp. 6, 36).

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications». In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861 (cit. on pp. 8, 9).

[14] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. «End-to-End Object Detection with Transformers». In: *CoRR* abs/2005.12872 (2020). arXiv: 2005.12872. URL: https://arxiv.org/abs/2005.12872 (cit. on pp. 11, 30, 35).

[15] Eranda Cela Rainer E. Burkard. *Linear Assignment Problems and Extensions* (cit. on pp. 12, 18).

[16] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. «Generalized Intersection over Union». In: (June 2019) (cit. on pp. 12, 36).

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762 (cit. on p. 13).

[18] Jeong-Seon Lim, Marcella Astrid, Hyun-Jin Yoon, and Seung-Ik Lee. «Small Object Detection using Context and Attention». In: *CoRR* abs/1912.06319 (2019). arXiv: 1912.06319. URL: http://arxiv.org/abs/1912.06319 (cit. on p. 15).

[19] Ronal J. Williams. «Simple statistical gradient following algorithms for connectionist reinforcement learning». In: (1992) (cit. on p. 16).

[20] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. «Model Assertions for Monitoring and Improving ML Models». In: *CoRR* abs/2003.01668 (2020). arXiv: 2003.01668. URL: https://arxiv.org/abs/2003.01668 (cit. on p. 19).

[21] Martin Abadi et al. *Tensorflow graphs*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/guide/intro_to_graphs (cit. on p. 20).

[22] Qualcomm Technologies. *Snapdragon Neural Processing Engine SDK*. URL: https://developer.qualcomm.com/docs/snpe/index.html (cit. on pp. 20, 22, 37, 38).

[23] Lei Mao. *Save, Load and Inference From TensorFlow 2.x Frozen Graph*. URL: https://leimao.github.io/blog/Save-Load-Inference-From-TF2-Frozen-Graph/ (cit. on p. 23).

[24] Dmitry Kurtaev. *Removing no operation nodes*. URL: https://github.com/opencv/opencv/issues/16879 (cit. on p. 23).

[25] Paul Henderson and Vittorio Ferrari. «End-to-end training of object class detectors for mean average precision». In: *CoRR* abs/1607.03476 (2016). arXiv: 1607.03476. URL: http://arxiv.org/abs/1607.03476 (cit. on p. 25).

[26] Gregory R. Koch. «Siamese Neural Networks for One-Shot Image Recognition». In: 2015 (cit. on p. 37).