Universiteit Leiden
The Netherlands

Informatica

Obstacle detection and avoidance using image processing on embedded systems

Wouter Stokman

Supervisors:
Dr. E.M. Bakker & Dr. M.S. Lew

Bachelor Thesis

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 08/2020

Abstract

The field of autonomous movement has advanced rapidly over the last decade. Applications of autonomous movement include transportation, surveillance and navigation. One of the main problems in these applications is detecting and avoiding obstacles. On these systems, mobility, power usage and accuracy are of the essence. This indicates the demand for a lightweight, embedded solution for obstacle avoidance.

We propose multiple, lightweight and embedded methods, based on earlier work. These single-image processing methods use Convolutional Neural Networks based on the StixelNet architecture (Garnett et al., 2017[9]) to continually identify obstacles in the systems direct field of view using a monocular camera setup.

Performance of these architectures was improved by making use of specialized optimization techniques to achieve real-time performance on a Jetson Nano.

Whereas the baseline StixelNet network reaches an average prediction time of about 1.2 seconds on the Jetson Nano, our network showed prediction times up to 27 times faster, down to about 0.04 seconds. This was achieved while keeping accuracy (average bin error) within about 16% of the baseline model, which is better than the original StixelNet model. The amount of trainable parameters was furthermore reduced from 31.404.402 to 523.537, which is a reduction of more than 98%.

Contents

1 Introduction **4**

2 Related work **4**

3 Hardware and Software **6**
3.1 Jetson Nano . . . 6
3.1.1 JetPack SDK . . . 6
3.2 Tensorflow . . . 6
3.3 TensorRT . . . 7
3.4 Keras . . . 7

4 StixelNet **8**
4.1 Original StixelNet . . . 8
4.2 Baseline: StixelNetV2 . . . 10
4.3 Our Stixel5x5 . . . 11
4.4 PL-Loss . . . 12

5 Database creator **13**

6 Automatic Dataset generation **14**

7 Experiments: Tools and Metrics **15**
7.1 Device . . . 15
7.2 Resource metrics . . . 16
7.3 Performance metrics . . . 16
7.4 Datasets . . . 17

8 Experiments **19**
8.1 Baseline Model Performance . . . 19
8.2 Memory Usage . . . 19
8.3 Memory Usage LXDE . . . 19
8.4 Factorization . . . 19
8.5 Architecture Downscaling . . . 20
8.6 Residual Layers . . . 21
8.7 Final Block Change (Stixel5x5) . . . 22
8.8 TF-TRT conversion . . . 23
8.8.1 Calibration runs . . . 23
8.8.2 Stixel5x5 and Downscaled TF-TRT . . . 23

9 Results **24**
9.1 Baseline Model Performance . . . 24
9.2 Memory Usage . . . 25
9.3 Memory Usage LXDE . . . 27
9.4 Factorization . . . 29
9.5 Architecture Downscaling . . . 30
9.6 Resnet Layers . . . 31
9.7 Final Block Change (Stixel5X5) . . . 32
9.8 TF-TRT conversion . . . 34
9.8.1 Calibration runs FP16 . . . 34
9.8.2 Calibration runs INT8 . . . 34
9.8.3 StixelDownscaled TF-TRT . . . 36
9.8.4 Stixel5x5 TF-TRT . . . 37
9.9 Best models . . . 38

10 Conclusion **39**

11 Future work **39**

12 APPENDIX **1**

1 Introduction

Over the last couple of years, developments in the field of autonomous movement methods have improved greatly. While autonomous driving and automatic surveillance are still relatively new in the technological landscape, many of these novel solutions were, until recently, only theorized to be possible far further into the future. One of the main tasks of a system that moves autonomously is that of obstacle detection: "To what places can the system move without colliding with an obstacle?". In the case of autonomous driving, these objects might represent cars, or curbs. In the case of rescue robots, obstacles might be trees, boulders and any other obstacle we might imagine. Practically every such system is in need of a general-purpose obstacle detection method.

There are many ways to detect such obstacles; but all methods utilize a sensor that is used to get input from the surroundings. We might employ sensors like lidars, radars, and sonars to retrieve this data. While these might be accurate in some cases, they also have several drawbacks: the main one being the cost. Accurate sensors such as these are often quite expensive, making them unsuitable for many applications. A more accessible sensor type, that has become more and more affordable due to the introduction and prevalence of smartphones is the camera. We will be using a method that can use a generic RGB webcam camera to process obstacle data in the direct field of view.

To use the raw sensor output to drive the autonomous system, in this case an image, we must have some way to interpret the input data. An example of such an algorithm would take 2 consecutive images while moving, 2 points can then be chosen on an object. These 2 points can then be located in the consecutive inputs. Comparing the resulting difference in distance of the 2 points in both images would indicate whether the obstacle is nearby or far away. The weakness of this algorithm lies in the fact that it needs to move between the 2 taken pictures, as well as the fact that the same points should be .

The overwhelming majority of the state of the art obstacle-detection methods in our use-case, use some form of computer learning. We use a Convolutional Neural Network based on the StixelNet (Levi, Garnett and Fetaya, 2015[25]) architecture to predict obstacle locations using a single-frame monocular camera input.

2 Related work

The field of embedded neural networks with respect to obstacle detection has been very active. *Convolutional neural network based obstacle detection for unmanned surface vehicle* (Ma, Xie and Huang, 2019 [26]) introduces a neural network based on CNN's, which is used to detect obstacles and to autonomously navigate on bodies of water. Other research (Omoifo [30]), focuses on surveillance using an autonomous land vehicles which performs obstacle detection and classification. Using 10.000 images, 85% accuracy was achieved while using a Visual Geometry Group model.

Research aiding in the field of obstacle detection is that of depth estimation. From a single image, a depth map can be deduced. These depth maps can then be utilized to autonomously navigate through various areas. (Eigen, Puhrsch and Fergus [8]) proposed a new depth estimation method. This new lightweight method uses a very limited amount of learning parameters, thus being much more suited for embedded applications.
Embedded applications of neural networks have been tested as far back as 2003 by the university of science and technology in Missouri (Yao et al. [37]). Whereas CNN's are very popular in these pattern-recognition task, these also use relatively many computer resources; this neural network

used a simple micro controller with 512 bytes of ram. Results showed the robot to be able to detect obstacles and navigate between them, all in real-time.

With the introduction of powerful microcomputers such as the Raspberry Pi and Jetson Nano, embedded neural networks applications became more accessible than ever. An example is *DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car* (Bechtel et al., 2018 [3]). This small Raspberry Pi-Based vehicle uses a a small front-facing camera to continuously take images of the surroundings. A CNN is then used to process this image; a steering angle is then calculated in order to avoid obstacles. The DeePicar neural network architecture is the same as the Dave-2 car made by NVIDIA. Even though a low-cost embedded computer is used, promising results have been achieved, the vehicle navigated on a premade track autonomously for more than 10 minutes.

Other embedded obstacle avoidance techniques use distance sensors to find new paths, such as a rescue robot (Budiharto and Pietro [4]). This Raspberry Pi based autonomous vehicle uses a neural network to interpret sensor data in order to navigate through obstacle courses. A small front-facing camera picks up video signal, applies a facial recognition algorithm and streams the video over de 2.4 Ghz Band. Applications include finding victims after earthquakes and other natural disasters. Due to the low cost, many individual vehicles can be deployed. Especially as microcomputers become more and more powerful, these applications become increasingly viable.

As for our embedded system: the Jetson Nano, there are already numerous neural networks made by both the Jetson community as well as NVIDIA itself. From image recognition to obstacle avoidance, embedded as well as powered by a remote machine. In the standard github repository, NVIDIA provides a pretrained neural network for the Jetbot [19], a Jetson Nano-driven vehicle purchase-able as a set from NVIDIA. This pretrained network allows for simple object detection and avoidance, which enables the Jetbot to autonomously avoid obstacles.

The functioning of some of these pretrained networks can be fairly rudimentary; the model provided by NVIDIA only predicts whether the Jetbot is being blocked or not. The Jetbot then continues turning until the network outputs a positive with a confidence level that is deemed higher than a predefined value (e.g. 30% chance of being blocked), indicating that the Jetbot can either move forwards, or is blocked.

We base our network on the state-of-the-art Convolutional Neural Network architecture called StixelNet [25] (see Section 4). This architecture is based on the LeNet [24], a relatively simple 5 layer network in which the first two layers are convolutional layers, with the last 3 remaining being fully connected layers. StixelNet is trained on the KITTI open image dataset [11], an image library containing data captured by a car equipped with both a stereo camera setup, as well as a 360°Velodyne laser scanner and GPS.

The idea of StixelNet was later expanded upon in *Real-Time Category-Based and General Obstacle Detection for Autonomous Driving* (Garnett et al., 2017 [9]) (see Section 4.2). Here, a new network architecture is introduced that is based around the obstacle classification problem. In everyday situations, one might be able to classify most of the objects in view. A disadvantage of a purely classification-based obstacle detection method, however, is that it is relatively resource-intensive, as well as difficult to accurately pull off. In practice, a significant amount of obstacles are difficult to accurately classify. This problem can be solved by using a general obstacle detection method.

To this end, the authors introduce a unified network that combine both classification and general obstacle detection. This network consists of two sections: a classification section and a general column-wise obstacle detection, both sharing computation with each other.

3 Hardware and Software

In this section, we give an overview of the hardware and software used during this project. For further information on the Jetson Nano as well as CNN's in general, see Section 12.

3.1 Jetson Nano

The Jetson Nano [28] is a small, nevertheless powerful computer. It has been designed with AI applications and especially Deep Neural Network (DNN) applications in mind.
Whereas earlier products in the NVIDIA Jetson line cost upwards to hundreds of euros [29], this more recent embedded computer, released in 2019, takes the Jetson line to an affordable price. The small computer comes in at around 100 euros, opening up a whole range of new possibilities for a wide audience.

It mainly differs from alternatives on the market, like the Raspberry Pi, with its computing power.

Further in-depth information on the installed packages and setup can be found in Section 7, as well as the Appendix 12.

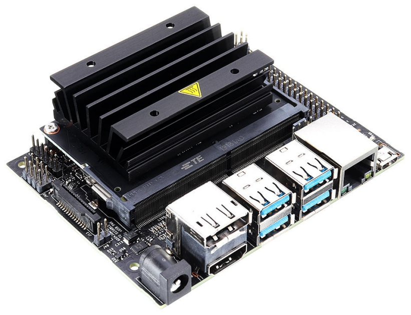


Figure 1: The Jetson Nano [20]

3.1.1 JetPack SDK

The JetPack software development kit, developed by NVIDIA, aims to be a comprehensive and simple solution for quickly setting up the Jetson Nano. It contains an OS, specifically designed for the Jetson Nano, as well as APIs, samples, developer tools and documentation [18].

JetPack, a development kit and flashing file produced by NVIDIA for the Jetson Nano, contains TensorRT (sec 3.3), cuDNN and CUDA libraries. This is also where the main strength of the JetsonNano lies compared to other solutions on the market. These libraries contain powerful tools to efficiently compile and run Artificial Neural networks using GPU support, increasing the performance dramatically compared to cpu-based inference while maintaining a small form-factor.

The Jetson Nano uses a custom driver package called L4T, this package contains the Linux kernel, bootloader and NVIDIA drivers based on Ubuntu 18.04 especially designed for the Jetson Platform [16].

3.2 Tensorflow

Tensorflow is a low-level software platform used for machine learning. It was originally developed by Google Brain (a deep learning research team at Google). In the year 2015 it became available to the public under the Apache license.

We use the Tensorflow libraries to implement our CNN. In order to simplify the creation/training of models, a version of Keras is used (Section 3.4).
Tensorflow offers GPU support, using NVIDIA CUDA (Compute Unified Device Architecture): a parallel computing platform developed by NVIDIA. Although Neural Networks can be trained on the CPU alone, performance increases dramatically when enabling the use of the relatively large parallel-computing power of modern graphics cards.

3.3 TensorRT

TensorRT (TRT) is a software development kit used for deep learning applications. TensorRT's goal is to optimize neural network models for use on GPUs. It is built on CUDA and claims a prediction-time speedup of up to 7x compared to conventional Tensorflow models.

The general workflow for using Tensorflow in combination with TensorRT (converting a TF model to TRT representation in short: TF-TRT) is shown in figure 2:

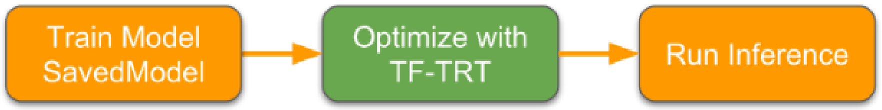


Figure 2: Workflow of optimization using tensorflow in combination with TensorRT [17]

TensorRT achieves a significant speedup by going over the layers and performing optimizations and transformations based on the architecture the model should be optimized for. Layers are fused wherever possible by combining convolutional, ReLU and bias layers. Layers can also be fused using layer aggregation. This combines layers that perform similar operations [17].

When training a network using Keras, a ".h5" file format is used to store the models in order to be able to quickly load the models and continue training. TensorRT conversion is done by first converting this Keras file format to the Tensorflow SavedModel format (".pb").

The output of this operation can then be converted with a call to `TrtGraphConverter()`. This operation should always be performed using the device on which the TensorRT model is going to be used since the optimization process is based on the architecture specifications.

Models can be converted using different precision modes, FP32, FP16 and INT8. As the names imply, these numbers represent in how many bits the models parameters are represented.

3.4 Keras

Keras is an open-source library used in neural network applications. It is created on top of multiple other neural network libraries, making it very modular. The primary goal of Keras is to ease the creation of and the experimentation on neural networks. Its focus lies in user-friendliness, which contributes to Keras being widely used in educational deep learning activities.

While Tensorflow also provides some high-level APIs, Keras is built in Python, making the design process much more user-friendly. Keras has, as of Tensorflow version 2.0, been part of the tensorflow package.

4 StixelNet

In this section, we describe the baseline model, as well as the adaptations to the network that were made which resulted in our improved Stixel5x5 model.

4.1 Original StixelNet

We base our network on the state-of-the-art Convolutional Neural Network architecture called StixelNet [25]. This architecture is based on the LeNet [24], a relatively simple 5 layer network in which the first two layers are convolutional layers, with the last 3 remaining being fully connected layers. StixelNet is trained on the KITTI open image dataset ([11]), an image library containing data captured by a car equipped with both a stereo camera setup, as well as a 360°Velodyne laser scanner and GPS.

The representation of output of the StixelNet model is based on previous research on medium-level representation of the complex surroundings and points of interest in real-life road situations from *The Stixel World - A Compact Medium Level Representation of the 3D-World* (Badino, Franke and Pfeiffer, 2009 [2]). Which proposed a stixel-world representation for interpreting road scenes: vertical objects in front of the vehicle are approximated by rectangles, resulting in a compact yet efficient representation of the complex dimensional traffic situation.

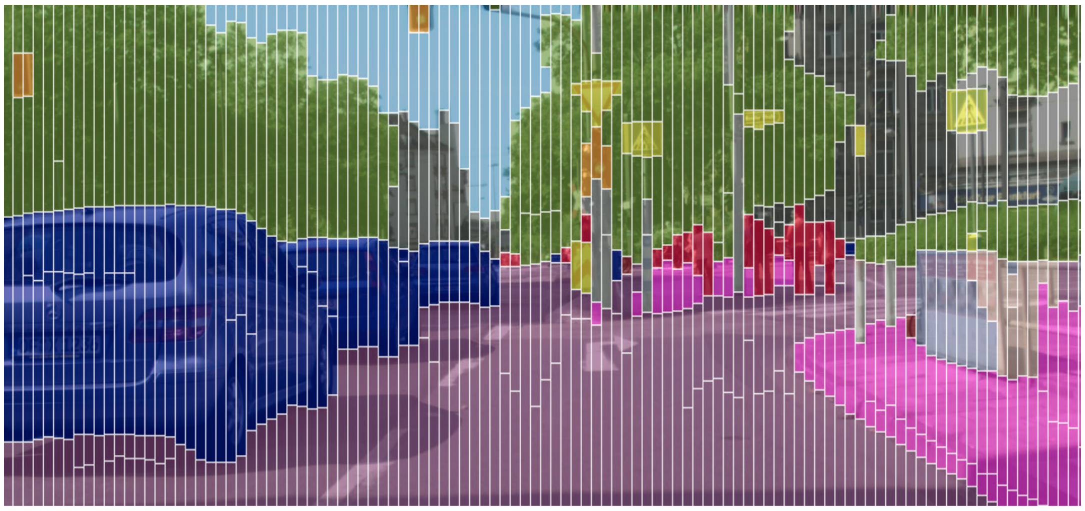


Figure 3: Stixel representation of a traffic situation [2]

StixelNet operates on single RGB images and attempts to outline obstacles by solving the following problem: find the pixel location y of the bottom point of the closest obstacle in the center column of the current image being processed.

The network received a single RGB image stripe of size (w,h,3), I_s. A "closest-obstacle-position" (y) must then be output to this stripe. $y = 0$ corresponds to the top of the image, and $y = h$ to the bottom of the image. This point must lie in the vertical domain $[h_{min}, h]$, where h_{min} corresponds to y-location of the horizon. A Stixel can then be drawn from $y = 0$ to point y, to indicate that all space above point y is blocked.

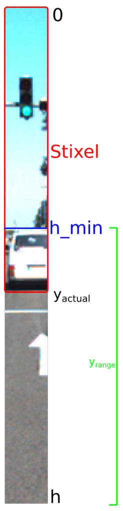


Figure 4: Single Stripe

The original network was trained by coupling image stripes (I_s) to closest-obstacle positions ($\hat{y}$). The most obvious choice for the output of the model would therefore be a single neuron that outputs a an expected obstacle position y. This idea was abandoned because of its possible data ambiguities, for instance in cases where multiple objects appear in the same image stripe, in which case multiple obstacle y-positions would be recognized.

Instead, the output is represented using a combination of a sigmoid probability function $P_{free}(y)$ and a binning-problem solution. At each column, the y-range is divided according to bins of size ($\frac{h-h_{min}}{N}$). A probability function is then defined over the full height by linearly interpolating between bin centers. The vertical-probability functions are then analyzed to produce a probability map for where obstacles could be. Figure 5 illustrates this analysis on a small part of an image.

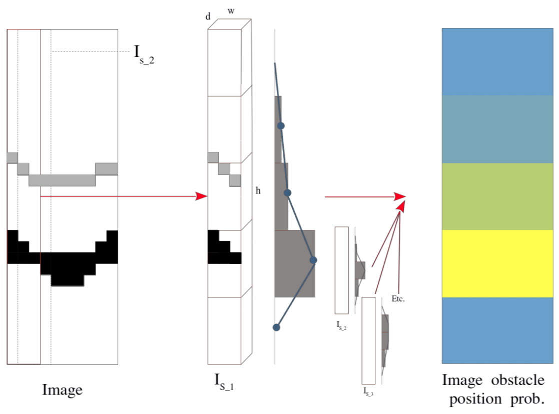


Figure 5: Sample output of a small part of an image

The image obstacle position probabilities resulting from the analysis of the image can then be interpreted using a Conditional Random Field (CRF), which resulted in a marginal better overall estimation. Discontinuities are penalized while small changes are not, this results in a more smooth obstacle position estimation and thus a more consistent image-wide prediction.

On a larger scale this results in an output which can be interpreted as seen in figure 6:

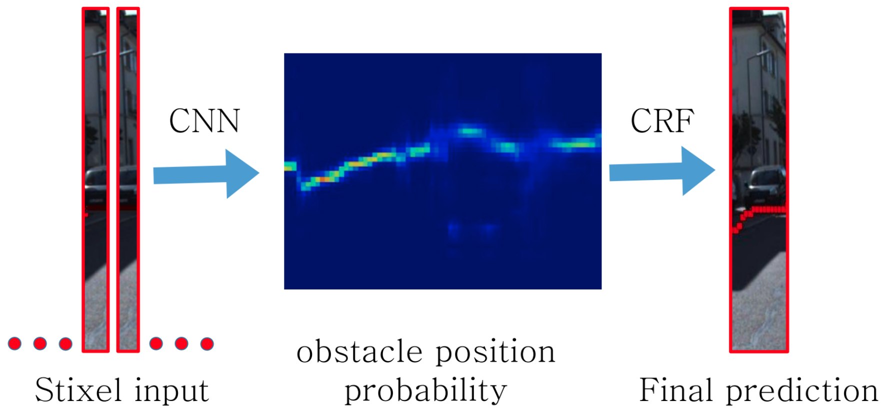


Figure 6: Sample output in a real world application

4.2 Baseline: StixelNetV2

The idea of StixelNet was later expanded upon in *Real-Time Category-Based and General Obstacle Detection for Autonomous Driving* (Garnett et al., 2017 [9]) This new obstacle detection method employs, just as StixelNet did, a singular camera to interpret road scenes.

This new network architecture is based around the obstacle classification problem. In everyday situations, one might be able to classify most of the objects in view. A disadvantage of a purely classification-based obstacle detection method, however, is that it is relatively resource-intensive, as well as difficult to accurately pull off. In practice, a significant amount of obstacles are difficult to accurately classify. This problem can be solved by using a general obstacle detection method.

To this end, the authors introduce a unified network that combine both classification and general obstacle detection. This network consists of two sections: a classification section and a general column-wise obstacle detection, both sharing computation with each other, as seen in figure 7. We will refer to this combined architecture as Combi-StixelNet.

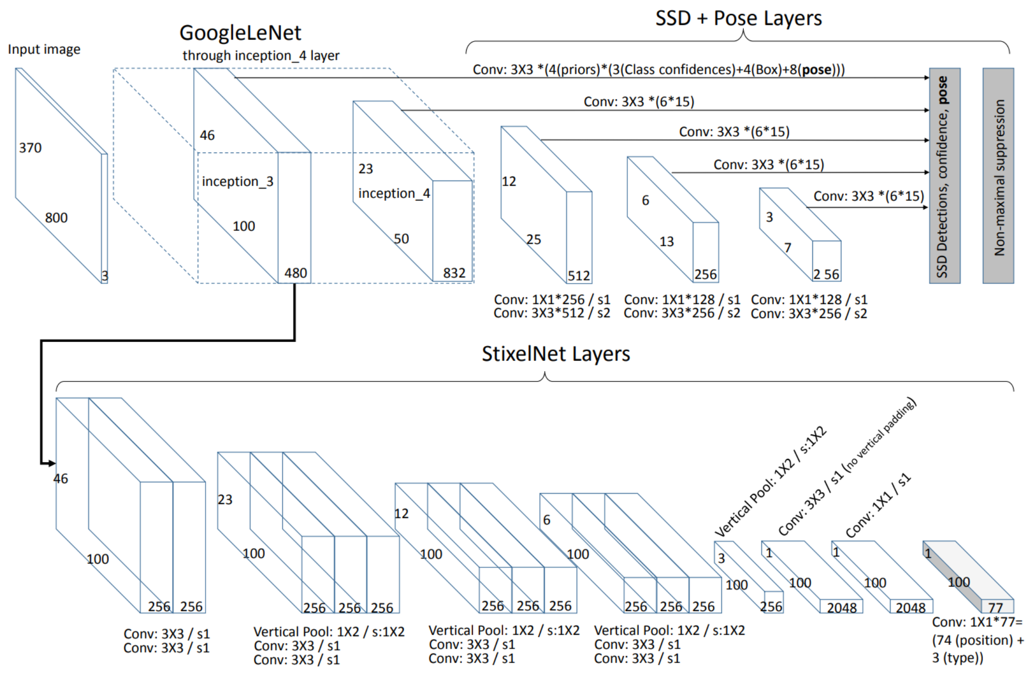


Figure 7: Architecture combining both classification and general obstacle avoindance, using a variant of the original stixelnet [9] (Combi-StixelNet)

The Combi-Stixelnet architecture uses a Single-Shot Multi-Box Detector (SSD) for classification and Pose detection, while an improved StixelNet architecture is used to general obstacle detection.

Computation in the earlier layers is shared as to decrease the computational load of this architecture, which made it possible to run the network in real-time at 30 frames per second.

For our application, we implement an embedded general-purpose obstacle detection method. We only use the StixelNet-part of the Combi-StixelNet in figure 7 as a baseline and starting point.

The base-architecture we use, based on the StixelNet portion of Combi-StixelNet 7 is shown in figure 8. We will be referring to this base-architecture as StixelNetV2.

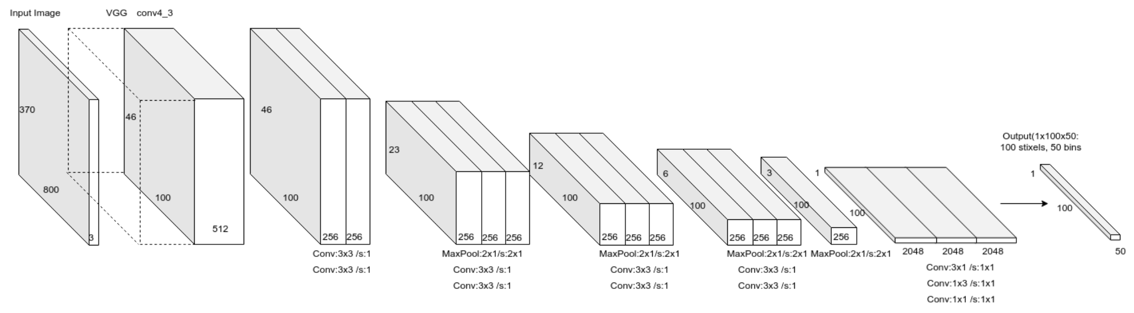


Figure 8: The isolated StixelNet architecture and baseline model[12] (StixelNetV2)

4.3 Our Stixel5x5

Our Stixel5x5 architecture, based on the StixelNetV2 architecture, is created by first downscaling the StixelNetV2 architecture, after which we introduce an architecture change in the final block and convert the resulting trained model to an optimized FP16 TRT model.

The downscaling operation works by reducing the amount of filters evenly in all layers in the StixelNetV2 architecture. When this factor is 0.5, for example, the amount of filters per layer is reduced by half. With "0.15 model" , we refer to a downscaled model or architecture with factor 0.15 filters remaining, this would mean that only 15% of all filters from the original StixelNetV2 architecture remain (and 85% are removed). Further information on network downscaling can be found in Section 8.5. Our resulting Stixel5x5 model was first downscaled to factor 0.15.

The final block change we introduce in the architecture is hypothesized to improve prediction performance by operating on a 200x100 shape before it reaches the final 100x50 output, and then, for every stixel position, interpret nearby stixel positions to solidify the prediction. This structure differs from the original StixelNetV2 model in that it only considers nearby locations instead of functioning like a fully-connected-like layer (see Section 8.7 for more information on Stixel5x5).

Both the original and improved final block of the StixelNetV2 architecture and Stixel5x5 architecture are shown in figures 9 and 10 respectively.

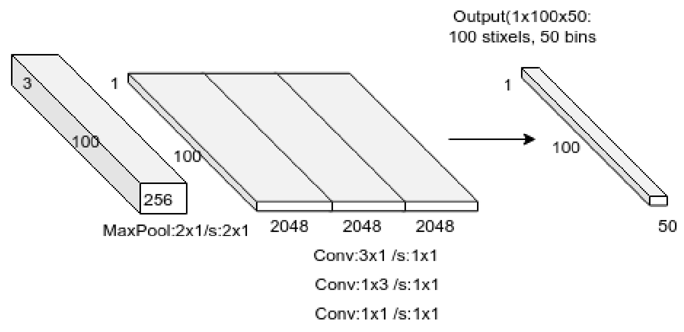


Figure 9: The last block of the StixelNetV2 architecture

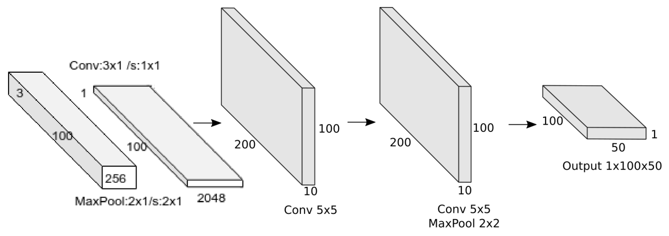


Figure 10: Restructured block used in the Stixel5x5 architecture

The resulting 0.15 downscaled Stixel5x5 model was converted using TF-TRT (see Section 3.3 and 8.8), to optimize prediction times, while maintaining the same accuracy.

Whereas the baseline StixelNetV2 network reaches an average prediction time of about 1.2 seconds, our resulting Stixel5x5 network predicted up to 27 times faster, down to about 0.04 seconds. This was achieved while keeping predication accuracy more accurate than the original StixelNet, and within 16% range of the original baseline model. The amount of trainable parameters was furthermore reduced from 31.404.402 to 523.537 compared to StixelNetV2, which is a reduction of more than 98%.

4.4 PL-Loss

Both for the original StixelNet and the general obstacle detection part of Combi-StixelNet, piecewise Linear probability (PL) loss is used during training.

As mentioned before in Section 4, each "stripe" in the image is divided into multiple y-bins. The StixelNet architecture will predict the probability at the center of each bin at the output nodes. Given a height y between two bin centers $c_i < y < c_{i+1}$, the probability $P(y)$ is given by:

$$P(y) = a_i \times \frac{c_{i+1} - y}{c_{i+1} - c_i} + a_{i+1} \times \frac{y - c_i}{c_{i+1} - c_i}$$

a_i and $a_i + 1$ denote the output of neurons i and $i + 1$ respectively ([25]). This method interpolates the value between the two closest bins to find the predicted probability of the obstacle being at the right location.

The loss function is then defined as the log-probability-difference between the predicted and expected value: $-logP(\hat{y})$

5 Database creator

In order to efficiently create a custom dataset, we produced a simple database creation tool. The tool consists of an editor and a creator. The creator enables the user to use the webcam to take a picture, which is then opened using the editor. In editor-mode, we can use the mouse to draw the stixel positions, which are displayed overlayed over the image. The user can add new stixel and edit existing positions by browsing the database using the browse tool.

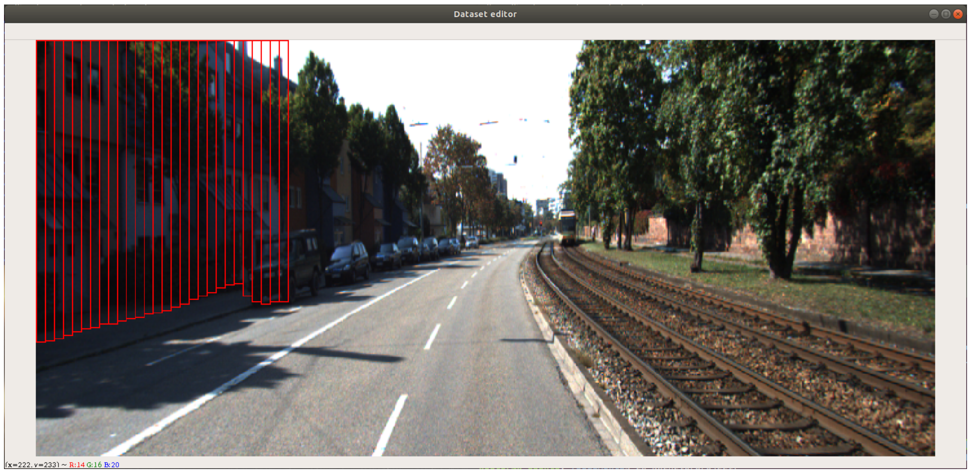
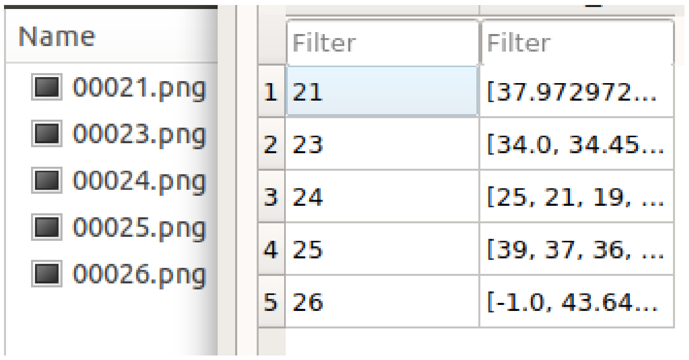


Figure 11: The dataset editor tool, stixel positions can be drawn and saved to the database

The resulting images are saved in a folder, the image path and stixel positions are saved in a MySql database. This database can be used during training and testing to retrieve training batches.



Name
00021.png
00023.png
00024.png
00025.png
00026.png

	Filter	Filter
1	21	[37.972972...
2	23	[34.0, 34.45...
3	24	[25, 21, 19, ...
4	25	[39, 37, 36, ...
5	26	[-1.0, 43.64...

Figure 12: The images and stixel positions are saved

6 Automatic Dataset generation

For our application, not many datasets are available. In Section 5, we introduced a database creator which can be used to take pictures and add stixel positions to each image which can then be added to a custom dataset. An alternative to this method is to automatically generate a dataset.

We produced a generic obstacle-scene generation program in the 3D creation suite Blender which can be used to create a random scene with multiple types of objects with a simple click of a button. The result of such a randomly generated setting can be seen in figure 13.

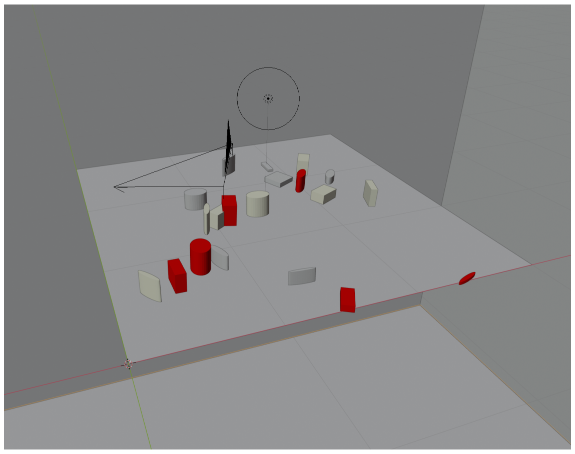
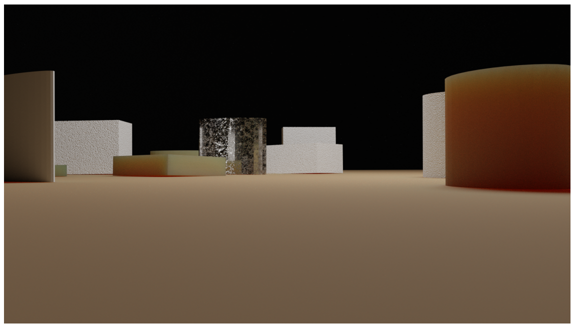
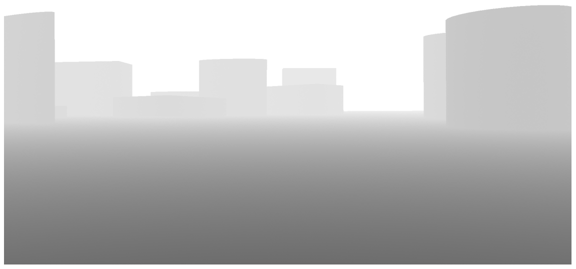


Figure 13: Example of a generated scene in Blender

Every time a button is pressed, the scene is reset. Then, multiple generic shaped objects are added to the scene at random locations, each one with random dimension settings. A random texture is chosen for each object, a simple light is then placed in the scene, at which point the camera view can be rendered. A custom rendering scheme was introduced that produces an image of the scene as well as a (precise) depth map, each of arbitrary resolution. A side-by-side comparison of the render and depth map render of the scene in figure 13 can be seen in figure 14a and 14b.



(a) Render of the scene in figure 13



(b) Depth Render of the scene in figure 13

The source code for this method can also be found in the repository [33]. Additional shapes and textures can be added to the generation methods. The generation and render process can be automated to generate various datasets of considerable size without much effort. Depending on the system and image resolution, rendering each image could take only a few seconds. The depth map can be utilized using the methods described by Garnett et al., 2017[9] and Levi, Garnett and Fetaya, 2015[25] to generate stixel positions.

7 Experiments: Tools and Metrics

In this section, the general testing setup and method is described, any deviation from this setup is explicitly mentioned in the experiments. We describe the hardware, datasets, libraries, software and settings used during training and testing.

7.1 Device

All performance and resource usage tests are done on an Nvid Jetson Nano (see Section 3.1 for a short introduction). The specifications of the Jetson Nano are listed below [21]:

- GPU: 128-core NVIDIA Maxwell
- CPU: Quad-core ARM ®A57
- Video: 4K @ 30 fps (H.264/H.265) / 4K @ 60 fps (H.264/H.265) encode and decode
- Memory: 4 GB 64-bit LPDDR4; 25.6 gigabytes/second
- Connectivity: Gigabit Ethernet
- OS Support: Linux for Tegra®
- Module Size: 70mm x 45mm
- Developer Kit Size: 100mm x 80mm
- Camera: Logitech C270 720p HD webcam

The Jetson Nano was flashed using JetPack version 4.4, the main libraries and APIs include:

- Cuda 10.2
- TensorRT 7.1.3
- cuDNN 8.0.0
- VPI 0.3.0

Tensorflow 2.2.0 is used, Keras is included in this version of Tensorflow.

During testing, a 1024MB memory-limit is allocated to Keras models, unless specified otherwise. This is necessary because the Keras models have a tendency to crash the system if no appropriate limit is set. Originally, the Jetson Nano came pre-installed with Ubuntu and desktop-manager Gnome. After the first memory experiment (Section 9.2), we used a Jetson Nano with LXDE running in headless mode during further experimentation. Before each test, all other user-specific processes are stopped and de cache is cleared. The Jetson is powered by a wall adapter to keep the power supply consistent over the different runs.

Tensorflow, TRT FP16 and INT8 models don't get assigned a memory limit during testing.
A seperately powered CPU fan was used to make sure the CPU/GPU temperatures of the Jetson Nano stayed well out of thermal throttling-range (sub 40 degrees at all times), as can be seen in the test-setup in figure 15.

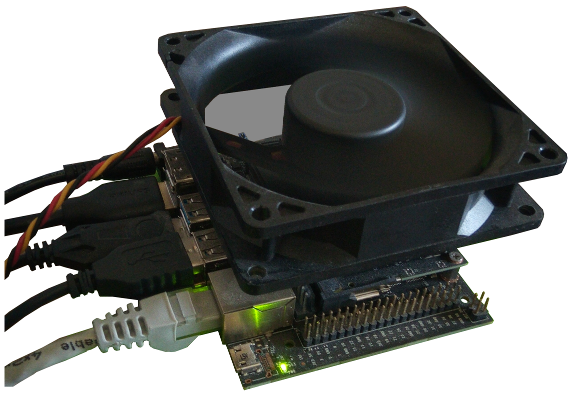


Figure 15: The Jetson Nano test setup

7.2 Resource metrics

For resource metrics, `tegrastats` is used. Every second, memory usage, cpu usage and other information is logged to file. In Python, the `time` package is used to keep track of prediction-time metrics. Unless specifically stated otherwise, the wall-clock time is used to measure the average prediction time.

7.3 Performance metrics

Performance metrics are collected by running predictions on the Kitti-dataset test subset that was also used in the original StixelNet paper [25]. This is a collection of about 800 images, with 27.771 stixel positions. Predicted bin positions are compared with bin-labels, and a bin error is calculated for every Stixel label and prediction.

By default, the prediction time for is measured using the Python `time.time()` method. This time is always measured as the time from when an input is given, to when a prediction is received. The average prediction time is the result of the total prediction time divided by the amount of predictions. Unless specified otherwise, the Keras models are used for evaluation.

The output of our model is of size 100x50 (columns x rows) and represent the obstacle-probabilities in each column, where each column should, in each column we will use the position with the highest obstacle-probablity in our evaluation, and then compare it to the labeled position to calculate the average bin error (see figure 16 for an example input and output of size 4x5). The matrix is the prediction output, the red circles represent the labels and de red cross represents a false positive in the prediction.

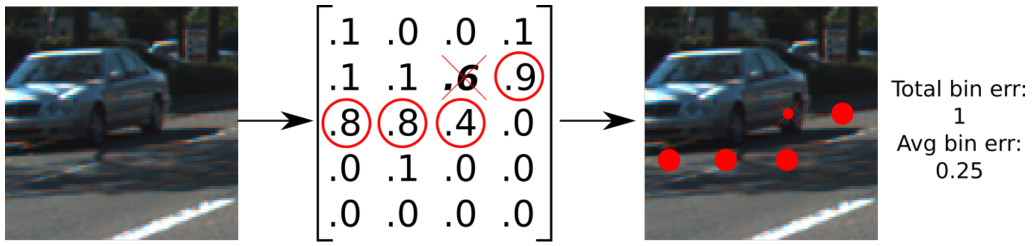


Figure 16: Bin error calculation

AUC denotes the area-under-curve for the "fraction of results within range"-curve. This curve describes what fraction of predicted stixel-points (y-axis) lie within x pixels or bins from the labeled position (x-axis). The AUC lies in range [0,1] and increases as bin error decreases. The AUC-performance can be measured at various intervals, for testing purpose we will frequently use the AUC over bins 0-3.

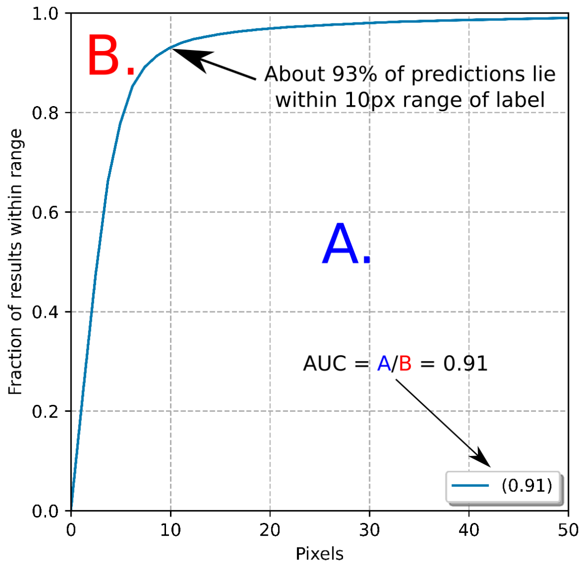


Figure 17: Example of a graph with its AUC

7.4 Datasets

Training is done on a subset of the KittiStixel dataset (see [11]), also used in the original StixelNet paper [25]. Keras and Tensorflow are used to train the network. The code-base used for training and dataset management has been written by `xmba15` and can be found on github [12], this base also contained the base implementation for the database used during testing.

Images in the Kitti Stixel dataset are augmented using the `Albumentations`[22] library, the (p=...) denotes the probability for an image to undergo this transformation during training, all operations are listed below:

- GaussNoise: Gaussian noise is added to the image (p=1).
- RandomShadow: Simulates shadows for the image (p=0.5)

- RandomRain: Adds rain effect to the image (p=0.5)
- RandomContrast: Randomly changes contrast of the image (p=0.5)
- RandomGamma: Randomly changes gamma of the image (p=0.5)
- RandomBrightness: Changes image brightness(p=0.5)
- HueSaturationValue: Increases saturation (p=0.5)
- CLAHE: Applies contrast limited adaptive histogram equalization to the image (p=0.5)
- Normalize: Keeps pixels in range [0,1] instead of [0,255] (p=1)

A custom mirroring operation is also used in 50% of all cases, this method mirrors an image and its labels, it is included in the codebase [12]. PL-loss (Section 4.4) was used for all training purposes. All models are trained for 50 epochs on the full training set.

8 Experiments

8.1 Baseline Model Performance

The baseline model, StixelNetV2, is trained using the method described in Section 7.4 and evaluated using the evaluation methods described in Section 7.3.

8.2 Memory Usage

In order to judge the memory usage of the Jetson Nano, we run various tests. We use the default Keras model described in figure 8 and load 50 images from the training dataset after which we make a prediction, the process time (using Python's `time.time_process_time()` function) of each prediction is tracked. We repeatedly test the model with different memory allocation limits at regular intervals, starting at a minimum of 500MB (any lower than this will cause stability issues) and stopping at 2000MB (any higher will crash the system).

Resource metrics are collected using the method described in Section 7.2.

Memory on the JetsonNano is shared between the GPU as well as the CPU. This eliminates a lot of the overhead used for synchronization but also results in some problems in our application. When Tensorflow is given full access over memory allocation, it will immediately allocate as much (GPU) memory as possible if the model is of a large size. This results in an immediate OOM-error on the Jetson Nano as RAM-allocations are also no longer possible.

This problem has not been documented very well, but a memory allocation limit can be set to counter this problem. The goal of this experiment is to allocate as much memory to Tensorflow as possible, without bothering background processes and while also keeping enough RAM-memory for image processing tasks during real-time operation, while also giving insight on model size when it is loaded into memory.

8.3 Memory Usage LXDE

As memory is a precious resource on embedded systems, we attempt to save as much as possible for the prediction process. Online sources claimed to be able to save up 1GB when installing a different desktop-manager. We install a lightweight desktop environment called LXDE on top of Ubuntu, replacing gnome, in an attempt to save memory. The same tests used in Section 8.2 are then ran again to observe the impact on performance.

The prediction time tests were executed consecutively, starting at 500MB and stopping at 2200MB, any higher and the process would freeze. The memory usage of the prediction process was monitored using tegrastats, asccording to Section 7.2 the results are shown in figure 28.

8.4 Factorization

Factorization is an architecture tweak that has been introduced in some successful convolutional neural networks. In inception-v3, asymmetric factorization is used to reduce a 3x3 filter into a 3x1 filter and a 1x3 filter. The reasoning behind it is as follows:

- One 3x3 filter results in $3 \times 3 = 9$ parameters
- Consecutive 1x3 and 3x1 filters result in $1 \times 3 + 3 \times 1 = 6$ parameters

This method thus reduces the amount of parameters by 33%. Results from the inception architecture suggested comparable results [35] when applying this on a larger scale.

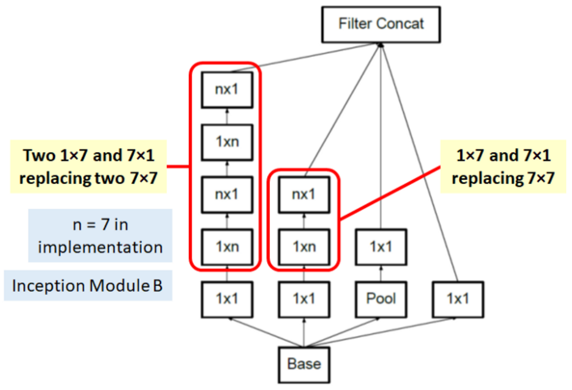


Figure 18: An example of factorization in an inception module in Inception-v3[35]

The factorization approach from figure 9.4 is originally only used in the final layer of the original StixelNetV2. This makes a much larger final-layer size possible without impacting the amount of parameters. In theory, this also applies to earlier layers, but at a somewhat smaller scale.

We leave the final block of our baseline architecture, StixelNetV2, untouched while introducing factorization (8.4) in all other blocks by replacing every conv2D layer with a 3x3 filter by one Conv2D layer with a 3x1 filter and one Conv2D layer with a 1x3 filter according to figure 19.

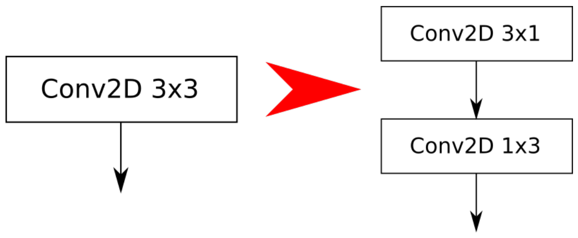


Figure 19: Factorization of a single Conv2D layer with filter size 3x3

This change is hypothesized to reduce the amount of trainable parameters, while maintaining a similar prediction accuracy.

The model is trained using the method described in Section 7.4, after which its performance is evaluated using the methods described in section7.3.

8.5 Architecture Downscaling

The Jetson Nano has a limited amount of memory available, as we establish in Section 9.2. One of the main steps in getting acceptable prediction times the use of a smaller model. We achieve this is by using the following approach: we reduce the amount of filters per layer. When less filters are used, the amount of (trainable) parameters is also reduced.

As the amount of operations for a prediction is dependent on the amount of filters, this reduction in filter count is hypothesized to have a direct effect on the average prediction times. The model itself is also reduced in size, which should result in less memory usage.

We start with the baseline model (with 31,404,402 parameters), reducing the amount of filters evenly by decreasing the amount of filters per layer by a constant factor.

Downscaling the model works as follows: we reduce the amount of filters in all layers evenly. When this factor is 0.5, for example, the amount of filters per layer is reduced by half. Each network is trained for 50 epochs, the resource and performance of each model is then measured using the methods described in Section 7. We will refer to the resulting models as "downscaled models" and to the architecture as "StixelDownscaled", with 0.05 being a downscaled model or architecture with factor 0.05 filters remaining, this would mean that only 5% of all filters from the original StixelV2 architecture remain (and 95% are removed).

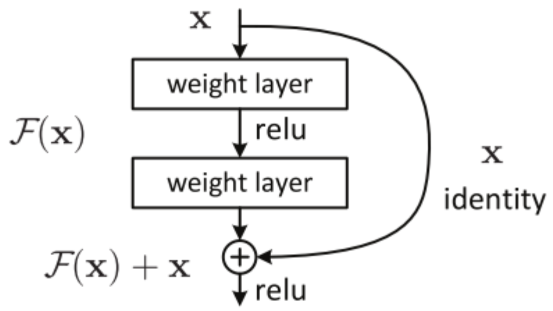
The model is trained using the method described in Section 7.4, its performance is then evaluated using the methods described in section7.3.

8.6 Residual Layers

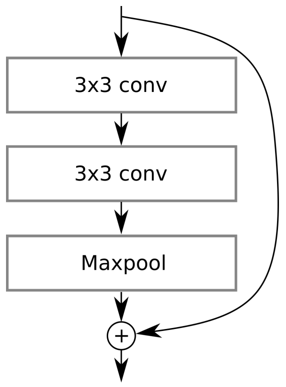
ResNet (Residual network) layers have been researched extensively in the last years. Residual networks were first introduced in 2015 [15], achieving depths 8x that of VGG nets by introducing skip layers, while maintaining a relatively low level of complexity. The original resulting network won the first place in the ILSVRC 2015 classification task, immediately gathering a large audience in the research community.

Increasing the depth of a neural network can increase its ability to learn a difficult task, but this often makes the vanishing-gradient-problem an issue due to the increased depth of the model. Residual networks function by using identity connections, as shown in figure 20a, to directly connect earlier layers with later layers. This enables the original input to "skip" through te model, which helps against the vanishing gradient problem.

Because identity shortcuts consist of a simple addition, they add no parameter and computational complexity to the model, making it an interesting addition to the architecture, especially if they can improve the downscaled model performance.



(a) A residual block from the original ResNet



(b) Introduced resnet layers in new architecture

We introduce a ResNet layer in each block in our base-architecture as seen in figure 20b. Our model uses `ELU` activation functions instead of the `RELU` functions used in the original ResNet. This has, in previous research, resulted in faster training times ([32]) and overall better performance, so we decided not to switch to RELU units. The maxpooling layers are also kept in their original places.

This architecture change is also performed on the 0.2 and 0.4 downscaled architectures.

We will refer to the resulting models as ResNet-StixelNet models. These models are trained on the Kitti dataset using the default method described in section 7.4. The models are then evaluated using the method described in section 7.3.

8.7 Final Block Change (Stixel5x5)

In the base StixelNetV2 architecture, the last block consists of a 1x3 Conv2D layer with 2048 filters, as seen in figure 21. The last layers are very similar to each other and seem to function similar to fully connected layers. Especially the last layer, which is a 1x1 conv2D layer with 2048 filters.

In the original paper ([25]), a CRF was introduced to smoothen the output of the model. Obstacles will often span over multiple columns, which should result in the bins of neighboring columns being close to each other. This pattern was used by a CRF to get a better accuracy. The filter change we introduce in the architecture is hypothesized to improve prediction performance similarly by operating on a 200x100 shape before it reaches the 100x50 output, and then, for every stixel position, interpret nearby stixel positions to solidify the prediction. This structure differs from the original model in that it only considers nearby locations instead of a fully-connected-like layer.

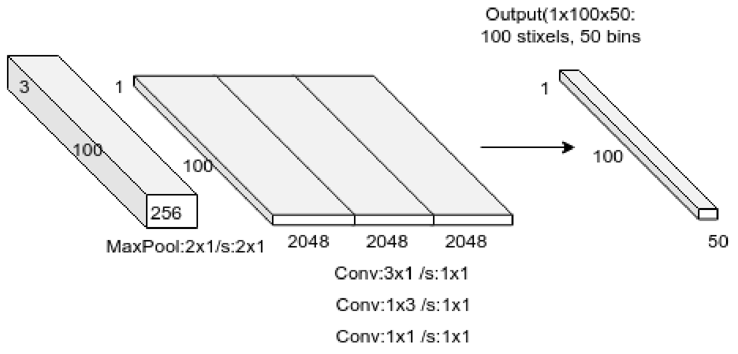


Figure 21: The last block of the StixelNetV2 architecture

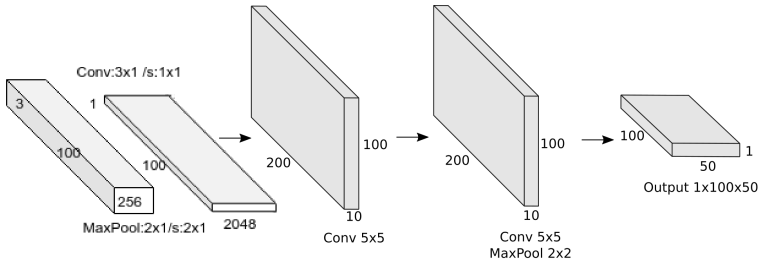


Figure 22: Restructured block used in Stixel5x5 networks

We will refer to this change in architecture as Stixel5x5 networks, downscaled Stixel5x5 models refer to downscaled models that underwent this change.

We perform this architecture change on multiple downscaled models, after which they will be trained and evaluated using standard methods described in Section 7.4 and 7.3 respectively.

8.8 TF-TRT conversion

In this section, we will dive into the Tensorflow-TensorRT process (also see Section 3.3). We will be referring to the original Tensorflow models as "Tensorflow models" and to TensorRT optimized models as "TRT Models" Subsection 8.8.1 describes the tests regarding the calibration run aspect of the TF-TRT conversion (Tensorflow model to TensorRT model conversion) process. Section 8.8.2 describes the optimization process of the two most promising novel architectures we created in the previous experiments (StixelDownscaled and Stixel5x5).

8.8.1 Calibration runs

A number of parameters are available for TF-TRT conversion (see Section 3.3). In order to optimize the model performance we first perform two experiments on the calibration runs settings. This parameter specifies how many calibration iterations are done after converting the model. NVIDIA documentation mentions no specific effect of the number of runs on prediction performance or prediction times.

We perform FP16 conversion as well as INT8 conversion of the 0.05 downscaled model (see Section 8.5) with various amounts of calibration runs. The models are trained on the Kitti dataset using the standard method described in section 7.4. They are converted using the standard TF-TRT conversion parameters, while using a varying amount of calibration runs. Conversions are done at 1, 5, 10, 50, 100, 200, 300, 400, 500 and 1000 calibration iterations. The resulting models are then evaluated using the methods described in Section 7.3.

8.8.2 Stixel5x5 and Downscaled TF-TRT

As TF-TRT conversion seems to improve the model prediction times at no accuracy cost, we try to observe te effect of this conversion on our two most promising architecture types. We will be using various types of downscaled versions to observe the difference in effect at different model sizes. FP16-conversion is used, as INT8-conversion displays no additional benefit in prediction times or accuracy.

Both Downscaled and Stixel5x5-Downscaled models from Section 8.5 and 9.7 at downscale factors 0.05, 0.10, 0.15, 0.20, 0.40 and 0.50 are TF-TRT FP16 converted using 1000 calibration runs. The resulting models are evaluated using the methods described in Section 7.

9 Results

In this section, the results of the experiments from section 8 are described. The raw CSV files can be found in the Appendix(12).

9.1 Baseline Model Performance

StixelNetV2 is trained on the Kitti dataset using the method described in Section 7 for 50 epochs. The model is then evaluated on the Kitti test dataset, and compared to the original StixelNet.

The original StixelNet 50px AUC performance is shown in figure 23.

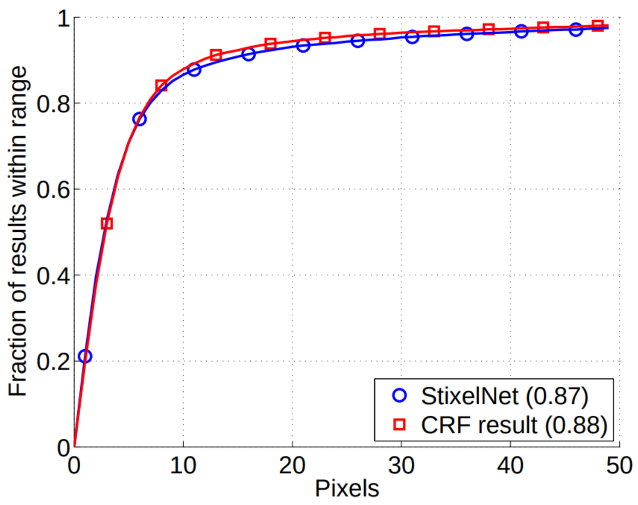
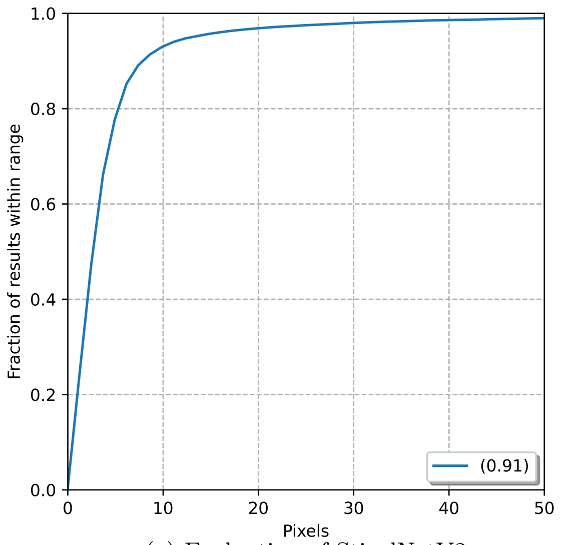


Figure 23: Original StixelNet model performance [25]

The performance of our baseline, the new StixelNetV2 model, is shown in figure 24a.



(a) Evaluation of StixelNetV2

Property	Value
Parameters	31.404.402
Avg. bin error	0.61
Avg. pixel error	4.5
Avg. Keras pred time	1.19 s
Avg. Loss	0.934
50-px AUC	0.91

A 50pixel area-under-curve (AUC) of 0.91 is measured. The best 50px AUC of the original

StixelNet was 0.88, when making use of a CRF [25].
The improved StixelNetV2 architecture evidently performs considerably better than the original StixelNet model.

9.2 Memory Usage

The memory usage experiment is done using the methods described in Section 8.2. The resulting prediction times are plotted against the amount of allocated memory in figure 25.

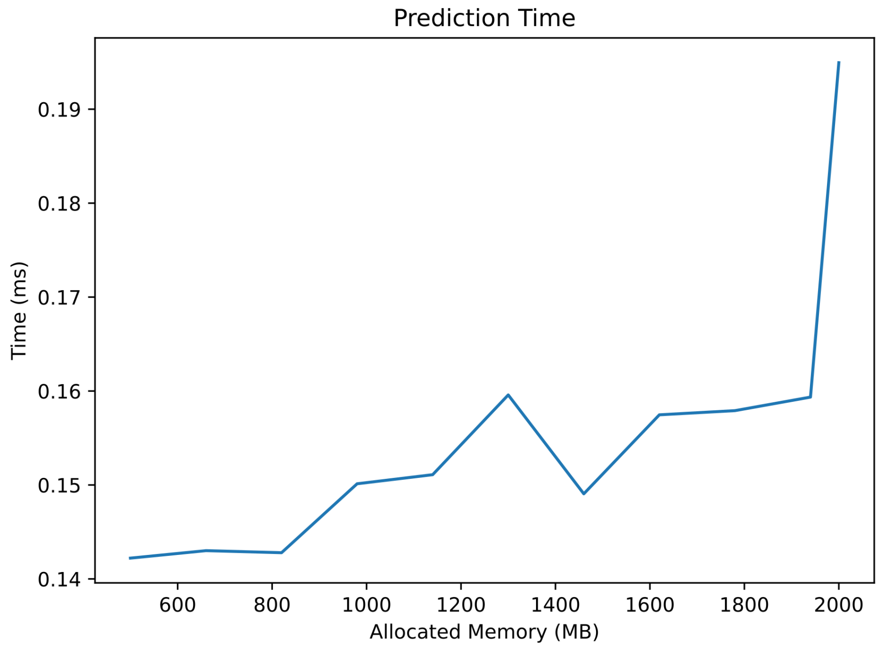


Figure 25: Prediction times when different amounts of memory are allocated to tensorflow

Tensorflow would at all times report that the memory allocator ran out of memory. This does not indicate an error but it does imply that the performance would be better if more memory were allocated. Figure 25 indicates that the opposite seems to hold true. When more memory is allocated, the prediction time seems to increase (the performance drops) by a small percentage.

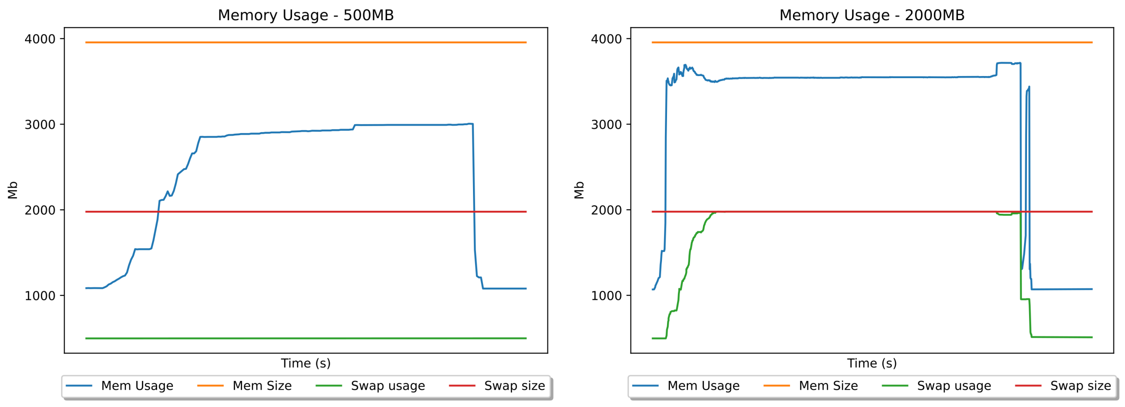


Figure 26: Results of allocating 500MB and 2000MB of memory to Tensorflow

Figure 26 suggests that memory usage stabilizes at around 1.1GB. This is the idle memory usage of the OS on which the Jetson Nano runs.

We can see that even when just 500MB of memory is allocated to Tensorflow, a significant portion of the 4000MB total memory is already being used. As soon as the upper limit (about 4GB) is reached, swapfiles are utilized to manage the larger amount of used memory. These swapfiles are many times more inefficient than normal memory, as disk access is much slower than memory access. This indicates that the model is too large to be managed efficiently by the Jetson Nano.

In figure 26 we can observe that the max amount of memory usage is not reached when allocating 500MB to the Tensorflow process, instead, around 3 of 4GB is used. It is curious to see that prediction times did not seem to go down when allocating up to 1GB more memory to Tensorflow, since this was empty memory and should have been able to speed up the model *write* $\rightarrow$ *load* loop from disk.

The unexpected positive relationship between the prediction time and allocated memory is also visible in figure 25. This could be explained by the abnormally high memory-usage of Tensorflow, which seems to impede other processes (and the non-tensorflow part of the prediction process, such as image resizing) from doing their job efficiently. We hypothesize this to be caused by the low memory availability for both the GPU and CPU at the same time, due to the shared memory of the Jetson Nano. Another theory is presented in the next section.

Although the prediction times might not reflect this, loading images from the dataset became many times more slow as more memory was being allocated to Tensorflow while approaching the memory limit. This, at one point, increased the total time for an image to be loaded and predicted from less about 1 second to more than 10 seconds. The system slowly became less responsive as more memory was allocated, this evidently also had an influence on the prediction times itself.

Running the tests again resulted in the same pattern, also when only considering the `time.time()` method.

If our hypothesis about approaching the memory limit and prediction times is correct, it would mean that the prediction performance would benefit from more available memory.

9.3 Memory Usage LXDE

LXDE desktop environment was installed, the same memory tests used in Section 9.2 are then ran again according to Section 8.2.
The new environment resulted in an idle memory usage of around 500MB instead of the 1100MB at idle when using the out-of-the-box os installed with JetPack, as can be seen in figure 27.

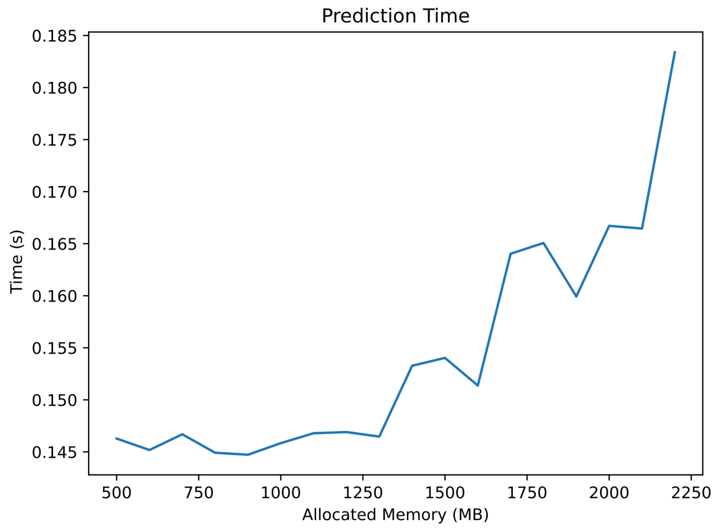


Figure 27: Prediction times at various memory allocations using Tensorflow and LXDE

Figure 27 suggests no significant prediction time decrease when allocating less than 2000MB of memory. We can see that the significant prediction-time-increase that was present at 2000MB during the original tests is now visible at the 2200 mark. Although performance gains are minimal at this point, it does suggest that we are able to use more memory without impacting the overall performance of the system.

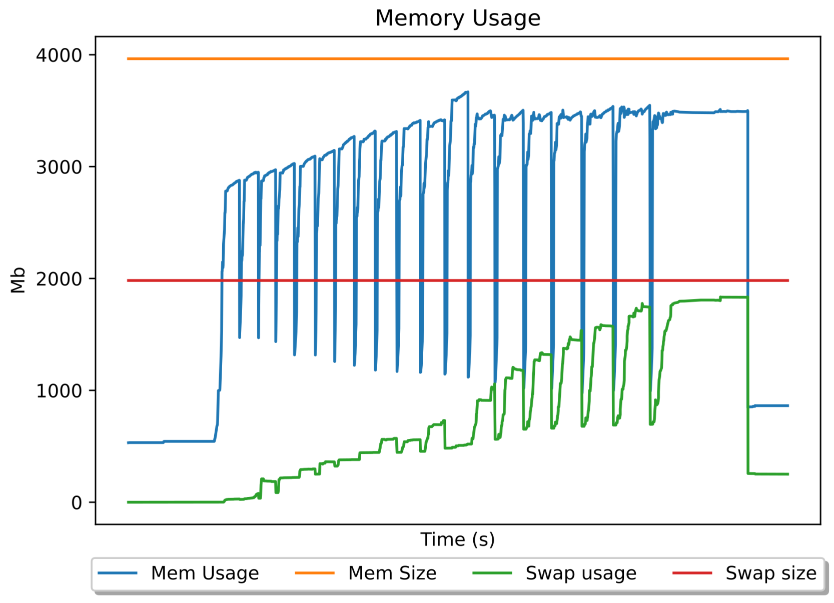


Figure 28: Memory usage when running the prediction time tests consecutively

Each peak in this graph represents a row of tests at an allocation level. The first peak being the resource usage during the 500MB allocation test. The last peak represents the point at which 2200MB was allocated to Tensorflow, at which point the terminal froze and the testing process had to be terminated.
As seen in this graph, the idle memory usage of the system has dropped to about 500MB, indicating about 600MB of extra available memory at idle.

A relation between swap usage and prediction time is visible when comparing figure 28 to 27. As more swap memory is utilized, the prediction times seem to go up. We introduced one theory to explain this phenomenon in the previous section. The performance decrease could also be caused by a shared memory allocator bottleneck. As swap memory is much slower than memory access, the virtual memory might cause the allocator to take more time when performing swap memory allocations.

Although the model still does not seem to fit into memory completely, the installation of LXDE does seem to give some more space in the memory-allocation aspect. In order to fit the model into memory, other methods are used, described in future sections.

9.4 Factorization

We factorize the model according to the method described in Section 8.4.

The resulting model 50px AUC performance is shown in figure 29.

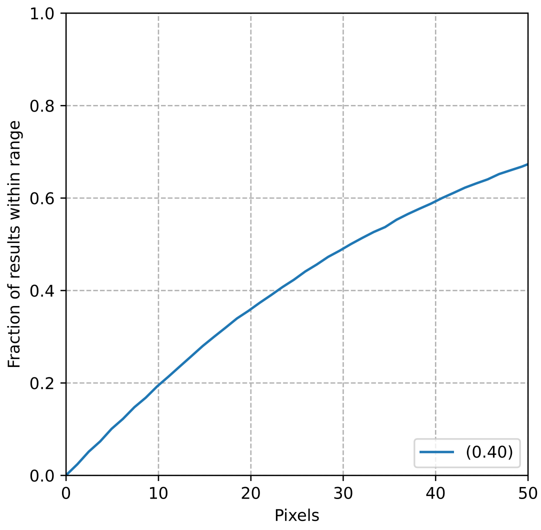


Figure 29: Performance of factorized StixelNetV2

The performance of the model is far worse than expected, at an AUC of 0.40 it performs very poor compared to the original model. Somehow the factorization caused a large loss of information. Although previous work did suggest that factorization does not work optimally in earlier layers in a model, here it seems to have a very negative impact on the overall performance. This result might be due to the fact that 3x1 and 1x3 filters might be worse worse at finding diagonal patterns than the original 3x3 filters.

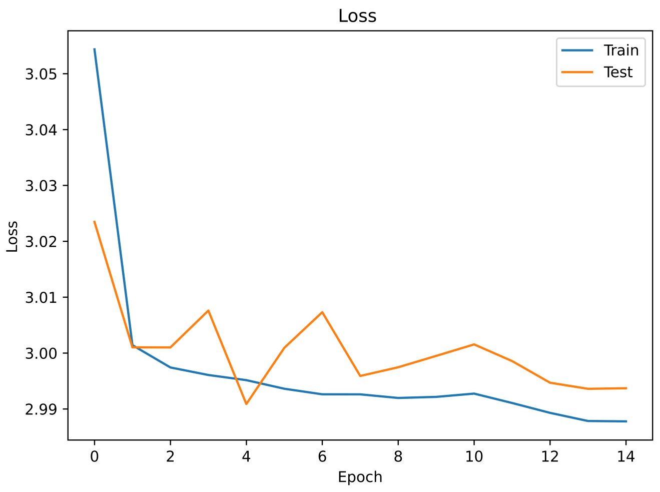


Figure 30: Loss during training of the factorized model

As shown in figure 30, learning rate during training very quickly deteriorated. After only 14 epochs the learning rate is so low that the training was automatically shut down.

9.5 Architecture Downscaling

We downscale the baseline model StixelNetV2 according to Section 8.5. The original StixelNetV2 baseline model parameter count (31.404.402) and average bin error (0.62) is displayed alongside the downscaled model in figure 31, which is the rightmost datapoint at 100% of the original parameter count (factor 1.00).

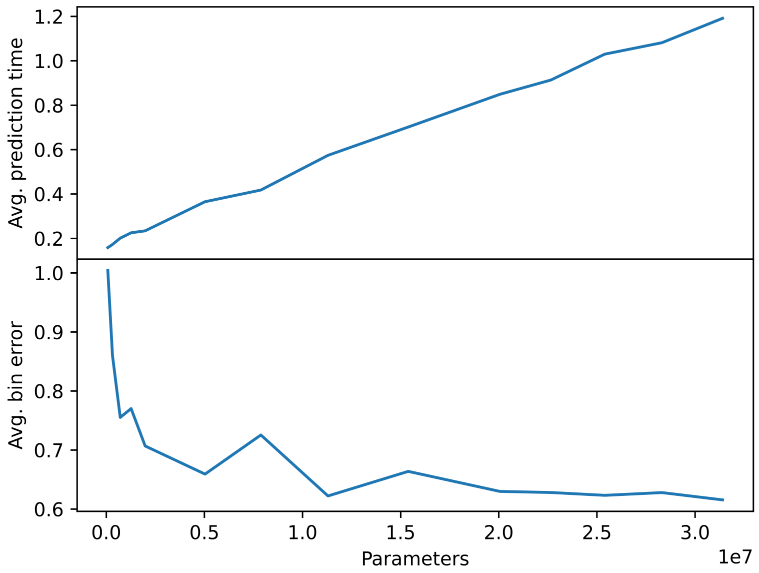


Figure 31: Keras model prediction time and performance at different parameter counts

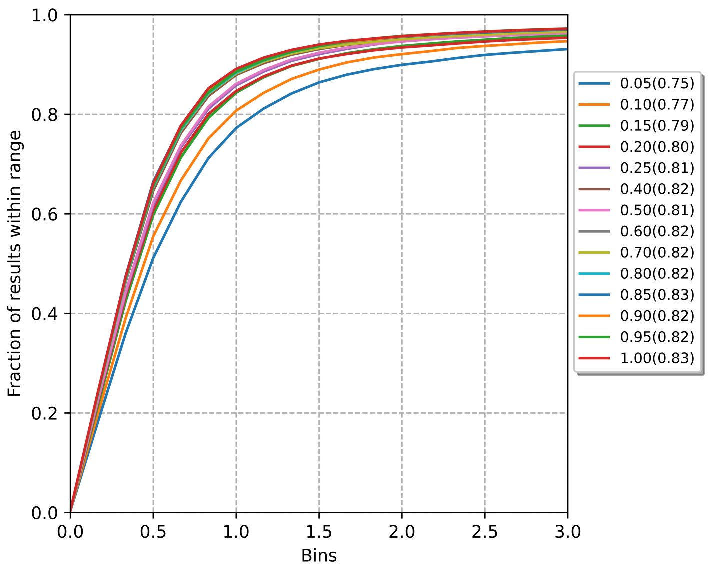


Figure 32: Model performance at different parameter counts

Results show the expected reduction in prediction time, a linear relationship between the number of parameters and the average prediction time is visible in figure 31. The prediction accuracy of the model does not seem to degrade greatly, even when reducing the parameters by half. It is at the $\frac{1}{6}$ reduction mark that we observe a relatively strong regression in the accuracy. In figure 32 we can see that the model with a 90% filter count reduction (0.10) predicts 80% of the stixel positions within 1 bin of error (this translates to about 7 pixels).

The relatively high performance of the low-parameter-count models indicate that quite a high degree of filter reduction is possible, which would result in an improvement of prediction time, while maintaining a solid prediction accuracy.

9.6 Resnet Layers

We change the StixelNetV2 architecture using the method described in Section 8.6, and do the same for the factor 0.4 and factor 0.2 architectures introduced in Section 9.5.

The resulting models are evaluated using the methods described in Section 7, the average bin error and average prediction times plotted against the parameter count of the resulting models can be seen in figure 33.

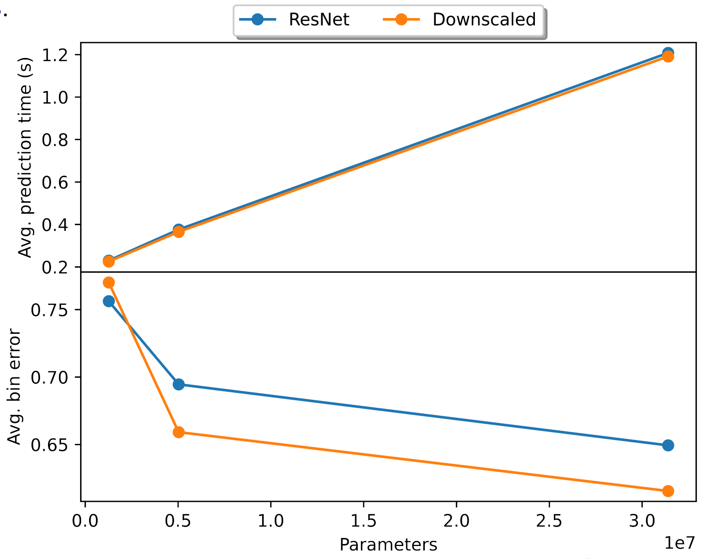


Figure 33: Prediction time and accuracy of models

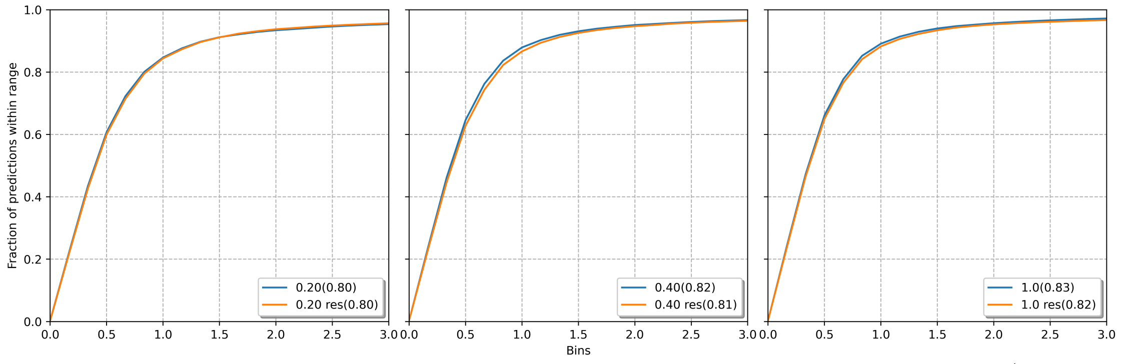


Figure 34: Performance of ResNet Stixelnets compared to base and downscaled version (the architecture with factor 1.0 refers the baseline StixelNetV2 architecture)

As expected, the ResNet layers had no reported effect on the average prediction times. Within the 2-bin-range, AUC performance of all models are very similar to their downscaled counterpart, with the ResNet-layered variant generally being marginally worse than the downsized models. At the 0.20 downsized model, performance of the ResNet model seems to be slightly more accurate than the original version.

9.7 Final Block Change (Stixel5X5)

The architectures for multiple downscaled-models from Section 9.5 were changed using the method described in Section 8.7. We evaluate the models using the tests from Section 7. We compare the results of the resulting (”new”) models, the downscaled Stixel5x5 models, to their original counterpart, resulting in figure 35.

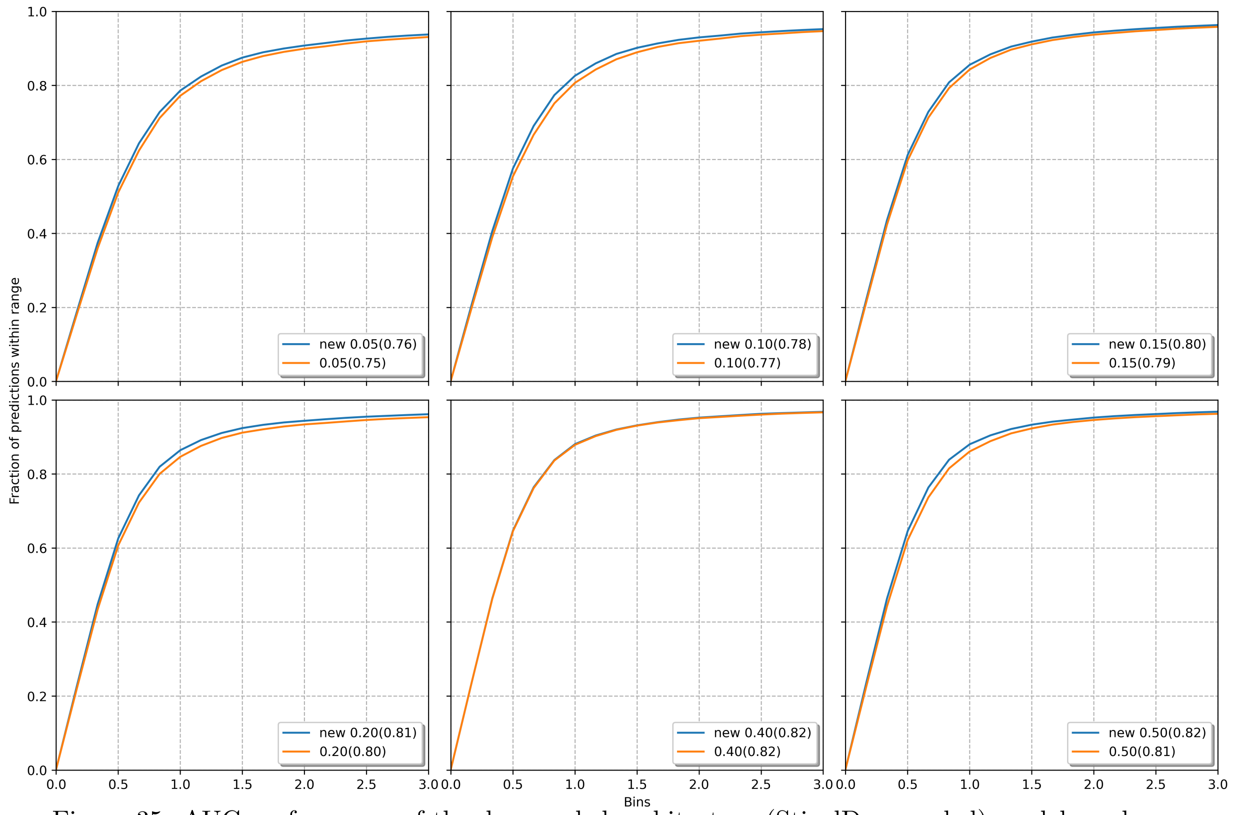


Figure 35: AUC performance of the downscaled architecture (StixelDownscaled) models and new (Stixel5X5) architecture models

Model	Original	New	Improvement
0.05	1.004	0.936	6.74%
0.10	0.860	0.804	6.52%
0.15	0.755	0.715	5.29%
0.20	0.770	0.702	8.80%
0.40	0.659	0.658	0.13%
0.50	0.725	0.649	10.47%
		Avg:	6.33%

(a) Average bin error improvements

Model	Original	New	Change
0.05	0.158	0.177	12.00%
0.10	0.173	0.198	14.69%
0.15	0.201	0.224	11.62%
0.20	0.225	0.246	9.44%
0.40	0.365	0.375	2.84%
0.50	0.418	0.417	-0.26%
		Avg:	8.39%

(b) Prediction time change

The results show a reduction in bin error across the board, with an average bin error reduction of about 6.33% compared to the downscaled models. Even though parameter counts of the new architecture were lower, the average prediction time increased by (on average) about 8.39%. This is probably due to the more compute-intensive nature of the 5x5 filter operation. The reduction in parameters results in better training times and less prediction error, as well as a more consistent accuracy after training, at the cost of some prediction time.

Even the smallest model, 0.05, has a 0.50px 0.87 AUC, suggesting on-par prediction accuracy with the original StixelNet architecture.

9.8 TF-TRT conversion

In this subsection, several experiments regarding TF-TRT conversion are conducted. The two most promising model architecture types from the previous sections are then converted and compared to their Tensorflow and Keras counterparts, after which the TRT models are compared to eachother.

9.8.1 Calibration runs FP16

We convert the 0.05 downscaled model using TF-TRT FP16 conversion according to Section 8.8.1.

The evaluated model is shown in figure 37.

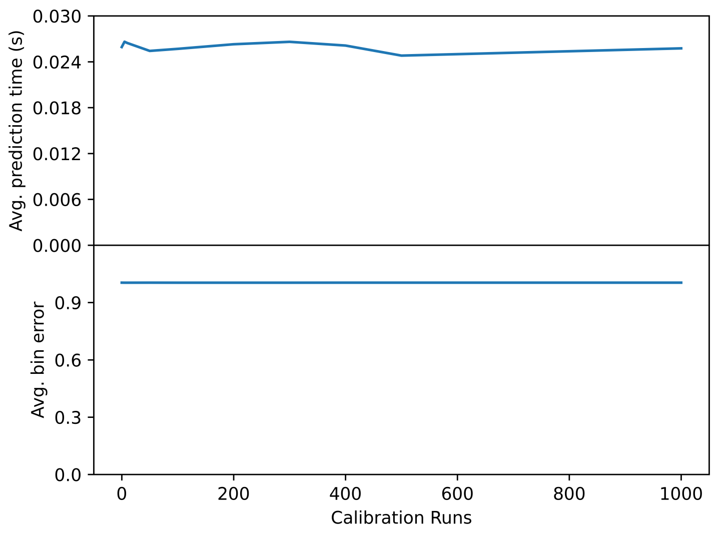


Figure 37: TRT model with different amounts of calibration runs

The number of calibration runs does not seem to impact the prediction performance of the model, the precision of the model also stays the same at 1.00 average bin error. This is the same bin-error as the original Tensorflow and Keras models of StixelDownscaled 0.05.

9.8.2 Calibration runs INT8

We convert the 0.05 downscaled model using TF-TRT `INT8` conversion according to Section 8.8.1.

The results are shown in figure 38, together with FP16 performance and the original factor 0.05 downscaled Tensorflow model performance.

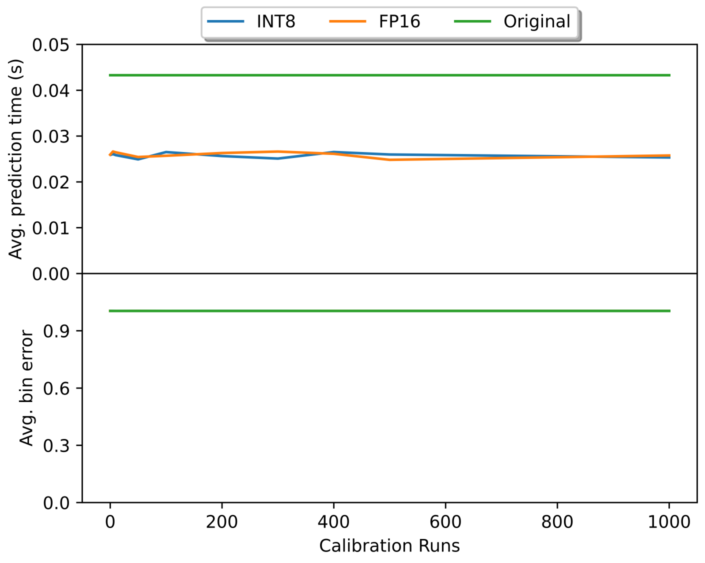


Figure 38: Trt conversion performance at various calibration iterations

As we can see, prediction accuracy, for `INT8` as well as `FP16` is not impacted as the models are converted using TF-TRT conversion, even when doing only a single calibration run. The average difference in bin error is about 0.007%.

Similar to the FP16 conversion, we observed no further improvement of prediction times and/or average bin error when conversion is done using more calibration runs.

The prediction time, is greatly reduced: both `INT8` and `FP16` TF-TRT conversion result in a reduction of prediction time of about 40%.

9.8.3 StixelDownscaled TF-TRT

We convert 5 StixelDownscaled models with downscale factor ≤ 0.5 from Section 9.5 using FP16 and INT8 TF-TRT conversion according to the experiment description in Section 8.8.2, they are evaluated using the methods described in Section 7. The results are shown alongside the performance of the original Keras model and Tensorflow model in figure 39.

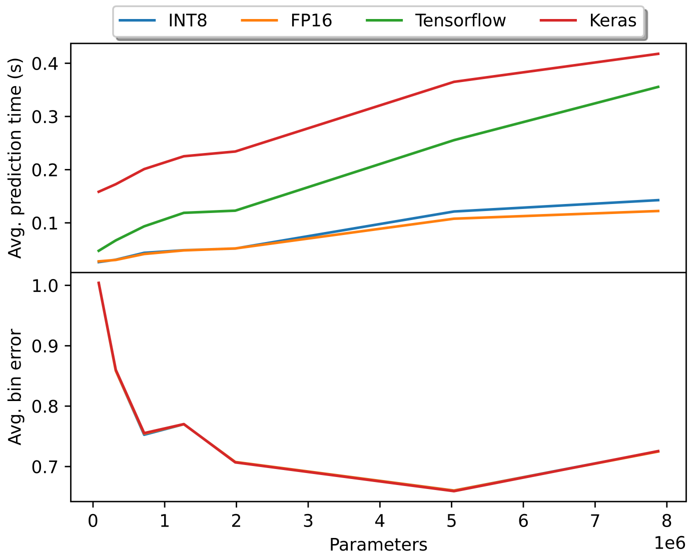


Figure 39: TRT model performance of various StixelDownscaled models with factor ≤ 0.50

We can see a significant prediction speedup when transitioning from Keras models to Tensorflow models and from Tensorflow models to TRT models. The prediction speed improvement for the Keras $\rightarrow$ Tensorflow model transition shows more percentage variance in this speedup, which seems to be the result of the prediction time reduction being a flat amount (about 0.1s). This is to be expected as the Keras model is just a higher-level Tensorflow model. The difference between these two seems to be how data is used as input. For Keras the `predict()` method is used, this seems to add a higher latency to each prediction than the Tensorflow model which directly uses a tensor-converted numpy array.

The conversion of Tensorflow$\rightarrow$ TRT reduces average prediction times by, on average, 56%. This percentage does not seem to differ at different model sizes. Converting to INT8 does not seem to improve prediction times and even results in marginally worse results.

9.8.4 Stixel5x5 TF-TRT

Five Stixel5x5 model types (with downscale ≤ 0.5) from Section 9.7 are converted using FP16 TF-TRT conversion, using the method described in Section 8.8.2. Performance is measured using the method described in Section 7. Prediction times and average bin errors are plotted against the amount of parameters for the downscaled Stixel5x5 models, resulting in figure 40.

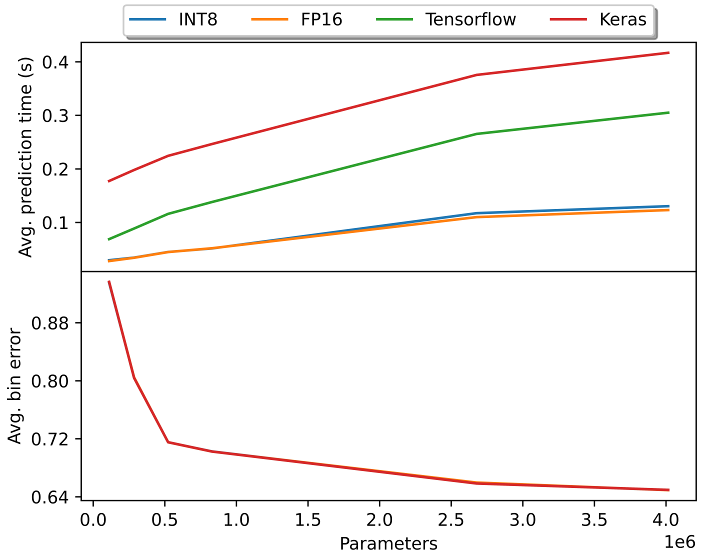


Figure 40: TRT model performance of various Stixel5x5 models with downscale factor ≤ 0.50

The results in figure 40 show a similar speedup pattern as shown in figure 39. Again, no significant difference between the FP16 and INT8 converted model is shown.

9.9 Best models

The FP16 TRT StixelDownscaled models from section 9.8.3 and FP16 TRT Stixel5x5 models from section 9.8.4 are our most promising architectural designs. We compare the resulting architecture performances with regards to their prediction times in figure 41.

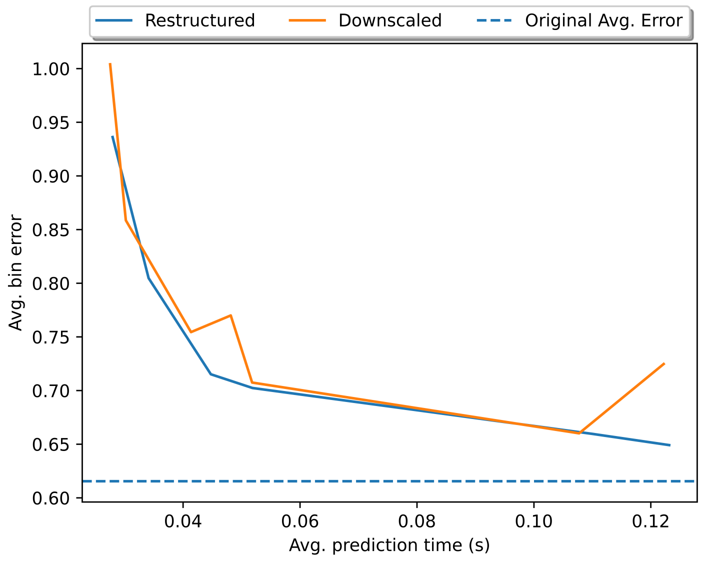


Figure 41: TRT model performance of best architecture types with downscale factor 0.50

In figure 41, we refer to the TRT Stixel5x5 architecture type with the term "Restructured".

The prediction accuracy of the TRT Stixel5x5 models are slightly better than the TRT Stixel-Downscaled models, also when considering prediction time change. From figure 41, it also seems that model training is more consistent for Stixel5x5. This is probably due to the reduction in filter size described in Section 9.7. The Stixel5x5 architecture type has a trainable parameter reduction of about 50% (from 7.879.890 to 4.015.709) compared to its StixelDownscaled-type counterpart, both at factor 0.50.

The TRT Stixel5x5 results show a small prediction accuracy gain and a more consistent training performance, TRT Stixel5x5 is a better alternative to the TRT StixelDownscaled architecture.

With a prediction time of about 0.04-0.12, we achieve about 8 - 25 frames per second with TRT Stixel5x5 while staying within 8-15% range of the prediction accuracy of the baseline model, depending on the downscale factor choice.

At around 0.04 seconds prediction time, we believe the trade-off between accuracy and prediction times of TRT Stixel5x5 to be optimal for our use-case.With an average bin error of about 0.71 and a prediction time of 0.0447s, we achieved a speedup of almost 27 times compared to the base StixelNetV2 model. This came at the loss of only 0.1 bin accuracy compared to the baseline model, which translates an accuracy loss of about 0.74 pixels, or 16%. The TRT 0.15 Stixel5x5 model reduced the amount of trainable parameters from 31.404.402 to 523.537, which is a reduction of more than 98% compared to the StixelNetV2 model.

10 Conclusion

We introduced a novel approach to the obstacle detection problem on embedded systems, based on StixelNet. Various model optimization methods were introduced for both models specific to the Jetson platform as well as the general network architecture.

Whereas the baseline StixelNet network reaches an average prediction time of about 1.2 seconds, our Stixel5x5 network predicted up to 27 times faster, down to about 0.04 seconds. This was achieved while keeping predication accuracy (average bin error) more accurate than the original StixelNet, and within 16% range of the original baseline model. The amount of trainable parameters was furthermore reduced from 31.404.402 to 523.537, which is a reduction of more than 98%.

Our lightweight implementation makes real-time obstacle avoidance possible on our embedded system, the Jetson Nano.

11 Future work

In our tests,`INT8` TRT conversion did not result in additional prediction time improvements over `FP16` conversion. We believe it to be worth it if this is caused by the NVIDIA framework or is a Jetson platform-dependent error.

The Kitti dataset contains images that were taken mainly in normal traffic-situation. As this was the only dataset available for our application, this was also the only input used to train and evaluate our models. We believe future work should focus on evaluating and improving the generalization capabilities of our networks. The training and test dataset could be expanded using the database creator introduced in Section 5. Additionally, in order to create larger datasets, automatic dataset generation could be done using the method described in Section 6.

References

[1] Zafer Arican. *Saving memory by installing lubuntu desktop*. 2019. URL: https://www.zaferarican.com/post/how-to-save-1gb-memory-on-jetson-nano-by-installing-lubuntu-desktop.

[2] Hernán Badino, Uwe Franke and David Pfeiffer. *The Stixel World - A Compact Medium Level Representation of the 3D-World*. Sept. 2009. DOI: 10.1007/978-3-642-03798-6_6.

[3] Michael G. Bechtel et al. *DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car*. University of Kansas: USA, July 2018. URL: https://arxiv.org/pdf/1712.08644.pdf.

[4] Widodo Budiharto and Aricò Pietro. *Intelligent Surveillance Robot with Obstacle Avoidance Capabilities Using Neural Network*. May 2015. URL: https://www.hindawi.com/journals/cin/2015/745823/.

[5] François Chollet. *Deep Learning with Python*. Manning, Nov. 2017. ISBN: 9781617294433.

[6] François Chollet et al. *Keras*. https://keras.io. 2015.

[7] *Compatible tensorflow versions*. 2020. URL: https://www.tensorflow.org/install/source#tested_build_configurations.

[8] David Eigen, Christian Puhrsch and Rob Fergus. 'Depth Map Prediction from a Single Image using a Multi-Scale Deep Network'. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014. URL: http://papers.nips.cc/paper/5539-depth-map-prediction-from-a-single-image-using-a-multi-scale-deep-network.pdf.

[9] Noa Garnett et al. *Real-Time Category-Based and General Obstacle Detection for Autonomous Driving*. Oct. 2017.

[10] Joel Gaya et al. *Vision-based Obstacle Avoidance Using Deep Learning*. Oct. 2016. DOI: 10.1109/LARS-SBR.2016.9.

[11] Andreas Geiger et al. 'Vision meets Robotics: The KITTI Dataset'. In: *International Journal of Robotics Research (IJRR)* (2013).

[12] *Github repository with an implementation of the stixelnet*. 2020. URL: https://github.com/xmba15/obstacle_detection_stixelnet.

[13] *Google Deepmind*. 2020. URL: https://deepmind.com/.

[14] *Ground truths source*. 2015. URL: https://sites.google.com/view/danlevi/datasets.

[15] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[16] *https://developer.nvidia.com/embedded/linux-tegra*. 2020.

[17] *https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index*. 2020.

[18] *https://docs.nvidia.com/jetson/jetpack/introduction/index.html*. 2020.

[19] *Jetbot coding demos*. 2020. URL: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/examples.

[20] *jetson image*. 2020. URL: https://www.google.com/url?sa=i&url=https%3A%2F%2Fnl.banggood.com%2FNVIDIA-Jetson-Nano-Developer-em%253Cx%253Ebedded-Development-Board-A57-Artificial-Intelligence-AI-Development-Platform-p-1519173.html&psig=AOvVaw00g2b-gAad5Ih_8Uc6FK1e&ust=1589214705409000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCPjLkaLcqekCFQAAAAAdAAAAABAH.

[21] *Jetson Nano Specifications*. 2020.

[22] A. A. Kalinin. 'Albumentations: fast and flexible image augmentations'. In: *ArXiv e-prints* (2018). eprint: 1809.06839.

[23] Sangwon Kim, Jaeyeal Nam and Byoungchul Ko. *Fast Depth Estimation in a Single Image Using Lightweight Efficient Neural Network*. Oct. 2019. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6832449/.

[24] Y. Lecun et al. 'Gradient-based learning applied to document recognition'. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[25] Dan Levi, Noa Garnett and Ethan Fetaya. 'StixelNet: A Deep Convolutional Network for Obstacle Detection and Road Segmentation'. In: *Proceedings of the British Machine Vision Conference (BMVC)*. Ed. by Mark W. Jones Xianghua Xie and Gary K. L. Tam. BMVA Press, Sept. 2015, pp. 109.1–109.12. ISBN: 1-901725-53-7. DOI: 10.5244/C.29.109. URL: https://dx.doi.org/10.5244/C.29.109.

[26] Liyong Ma, Wei Xie and Haibin Huang. *Convolutional neural network based obstacle detection for unmanned surface vehicle*. Nov. 2019. URL: https://www.aimspress.com/fileOther/PDF/MBE/mbe-17-01-045.pdf.

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

[28] *NVIDIA*. 2020. URL: https://www.nvidia.com/.

[29] *NVIDIA Jetson-Nano*. 2020. URL: https://www.nvidia.com/nl-nl/autonomous-machines/jetson-store/.

[30] Darlington O. Omoifo. *Obstacle detection in autonomous vehicles using deep learning*. Apr. 2019. URL: https://www.theseus.fi/bitstream/handle/10024/143661/Omoifo%20O.%20Darlington.pdf?sequence=1&isAllowed=y.

[31] Harry A. Pierson. *Deep Learning in Robotics: A Review of Recent Research*. 2014. URL: A%20summary%20of%20https://arxiv.org/ftp/arxiv/papers/1707/1707.07217.pdf.

[32] Anish Shah et al. 'Deep Residual Networks with Exponential Linear Unit'. In: *CoRR* abs/1604.04112 (2016). arXiv: 1604.04112. URL: http://arxiv.org/abs/1604.04112.

[33] Wouter Stokman. *Project Repository Github*. URL: https://github.com/Woutah/EmbeddedStixelNet.

[34] *Tesla*. 2020. URL: https://www.tesla.com/autopilot.

[35] Sik-Ho Tsang. *Review: Inception-v3 — 1st Runner Up (Image Classification) in ILSVRC 2015*. 2018. URL: https://medium.com/@sh.tsang/review-inception-v3-1st-runner-up-image-classification-in-ilsvrc-2015-17915421f77c.

[36] James Vincent. *Former Go champion beaten by DeepMind retires after declaring AI invincible*. Nov. 2019. URL: https://www.theverge.com/2019/11/27/20985260/ai-go-alphago-lee-se-dol-retired-deepmind-defeat..

[37] Qiang Yao et al. *A RAM-Based Neural Network for Collision Avoidance in a Mobile Robot*. 2003. URL: https://scholarsmine.mst.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1820&context=ele_comeng_facwork.

12 APPENDIX

Contents

1 Introduction **2**

2 Computer Learning **2**
- 2.1 Deep Learning . . . 3
  - 2.1.1 Artifical Neural Network . . . 3
  - 2.1.2 Convolutional Neural Networks . . . 6

3 Jetson Nano **7**
- 3.1 SSH . . . 7
- 3.2 SSHFS . . . 7
- 3.3 Tensorflow installation . . . 8
- 3.4 Including cuda in path . . . 8
- 3.5 Memory/Performance issues . . . 8
- 3.6 CUDNN version . . . 9
- 3.7 JetPack 4.4 . . . 9
- 3.8 Python Package List . . . 10
- 3.9 Implementation . . . 13
- 3.10 Prediction and label format . . . 14

4 Errors and fixes Jetson Nano **15**

5 General Information **15**

6 CSV results **16**
- 6.1 Memory Allocation . . . 16
- 6.2 Factorization . . . 16
- 6.3 Architecture Downscaling . . . 16
  - 6.3.1 Keras (H5) . . . 16
  - 6.3.2 Tensorflow (PB) . . . 16
  - 6.3.3 TRT (FP16) . . . 16
  - 6.3.4 TRT (INT8) . . . 17
- 6.4 Resnet Layers . . . 17
  - 6.4.1 Keras (H5) . . . 17
- 6.5 Filter Change (Stixel5x5) . . . 17
  - 6.5.1 Keras (H5) . . . 17
  - 6.5.2 Tensorflow (PB) . . . 17
  - 6.5.3 TRT (FP16) . . . 17
  - 6.5.4 TRT (INT8) . . . 18
- 6.6 0.05 Calibration runs . . . 18
  - 6.6.1 TRT (FP16) . . . 18
  - 6.6.2 TRT (INT8) . . . 18

1 Introduction

In this appendix, we will explain assumed knowledge for the original document. This appendix also contains (troubleshooting) guides on the Jetson Nano and getting started sections, as well as the raw model results in CSV form. A list of all items can be found in the table of contents.

2 Computer Learning

Computer learning techniques have been and are being used in many applications; facial recognition, photo processing, but also, more recently, in problems that are considered more difficult and less conventional. Problems previously thought to be impossible for computers to solve. An example of such a feat is the defeat of the world champion in the board game GO [5]. Some experts believed this game to be too complex for computers to analyze, resulting in estimations of more than 20 years for when computing power would be up to this task. Now, many years earlier, significant results have been achieved thanks to Google Deepmind [3], a deep learning platform created by Google.

The main appeal of these neural networks lies in its versatility and self-learning capacity. Performing complex task with a computer previously meant that experts in the field would have to analyze many desired outputs, given some input, then creating a (very complex) mapping function in the form of a computer program. Deep neural networks (DNNs) function differently in that they only need an input and output, they then ”learn” this mapping function. This eliminates the need of a significant part of the human factor in the development process, opening up the possibility to a significant reduction in development costs. These resources can then be used to further develop the DNN to achieve even better results.

Deep neural networks have also redeemed themselves while executing tasks which have been deemed very difficult for normal algorithms to solve.

Another source of applications of the Deep neural networks is found in the field of engineering; notably in the field robotics, where neural networks can aid enormously in problems that are difficult for an average algorithm to solve. An example would be a balancing problem: preventing upright standing robots from falling down.

Parallel to this field we find the autonomous vehicles: rescue robots, surveillance drones and vehicles aimed at autonomous transport. Autonomous cars are perhaps the most main-stream application of the Deep Neural Networks. Embedded computers enable cars like the Tesla Model X to autonomously drive or inform the driver of potentially dangerous situations. While full-fledged driving without human interference, supervision and intervention, might not be there quite yet, enormous leaps have been made in the past decade.

On a much smaller scale, embedded mini-vehicles have also become increasingly popular. Small, autonomously operating rescue robots would enable rescue workers to quickly locate survivors in collapsed buildings in earthquake areas. While tiny drones enable large scale surveillance of secure areas.

Autonomous vehicles superficially operate on one problem: detecting obstacles, and avoiding them. We will be using an existing, state-of-the-art object detection neural network and attempt to improve upon it with respect to the performance-and-resources tradeoff.

2.1 Deep Learning

The artificial intelligence (AI) landscape has shifted dramatically over the last decades due to the large increase in global computing power in combination with the development of deep learning techniques. Deep learning can be described as an algorithm that starts with a certain set of rules. Through experience, this AI will improve. This "experience" can be thought of as mapping some input to some output using this machine-learning model, after which the result is evaluated and used to further improve the function of this model.

Deep learning is called deep learning because of its structure: each model functions by representing the input mapping to some output, using layers. Each layer will map output of the previous layer to some new representation. The amount of layers denotes the depth of the model. These models almost always function using some form of an Artificial Neural Network (ANN).

2.1.1 Artifical Neural Network

An ANN consists of layers of neurons, its architecture is inspired by how a brain works in biology. Each layer consists of a set amount of neurons, each one connected with some input (the output of the previous layer) and with some output (a connection to the next layer). The final layer can be considered the output of the neural network.

The input and output of these neurons are represented using some number, comparable the the excitation (e.g. +1) or inhibition (e.g. -1) of a neuron in biology. We might consider the following (simple) example:

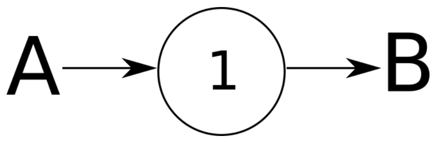


Figure 1: A neuron

Some input A is mapped to some Output B. Let us say that neuron 1 outputs either 0 or 1, based on whether the input is 0 or 1. We could simply link the input to the output, of course, this example is not very interesting. We can make the example more interesting by introducing a second input for some neuron:

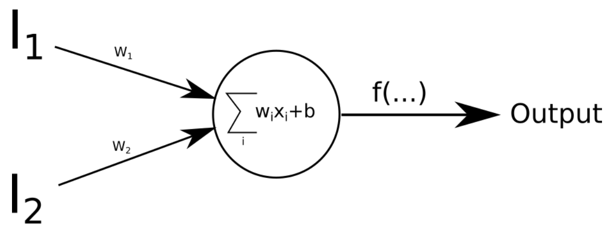


Figure 2: A neuron

We can describe how a neuron (generally) works by the following properties displayed in the image above:

- Inputs
- Weights
- An activation function f(...)

I_1 and I_2 denote the output of the neurons that are the input of the current neuron. For simplicity's sake we can consider these inputs either $+1$ or -1. Each of these inputs have some weight attached to them, this weight specifies the amount of influence that connection has on the final outcome of the neuron. The first step in calculating the output of the neuron is summing the inputs together using the weights using:

$$\alpha = \sum_i w_i x_i + b$$

Here, w_i and x_i describe the weight of input i and the activation of input i, respectively. As noted before, a higher weight of a certain input will increase its influence of the final outcome of this sum (and thus the output of the neuron). b denotes the bias, this bias may differ from layer to layer and is introduced in order to be able to shift the output upwards or downwards. Without the bias, we would only be able to multiply inputs using the weights,

The output of the neuron is then denoted by:

$$f(\alpha)$$

For some activation function $f(x)$. This activation functions maps the sum of inputs (with weights) to some output. Different kind of activation functions exists, each with its own pros and cons and applications.

The identity activation function:

$$f(x) = x \tag{1}$$

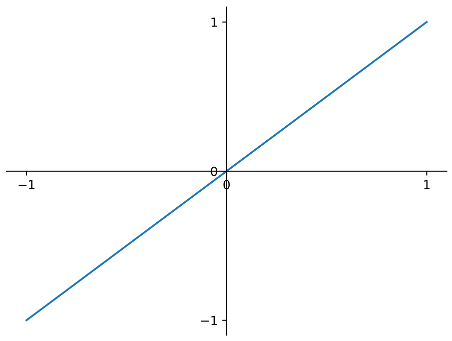


Figure 3: Identity activation

The sigmoid activation function, which can be described using a function similar to:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2}$$

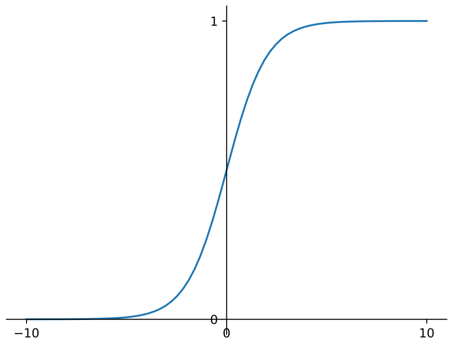


Figure 4: Sigmoid activation

The binary step function:

$$f(x) = \begin{cases} 0 & \text{if} x < 0 \\ 1 & \text{if} x \geq 0 \end{cases} \tag{3}$$

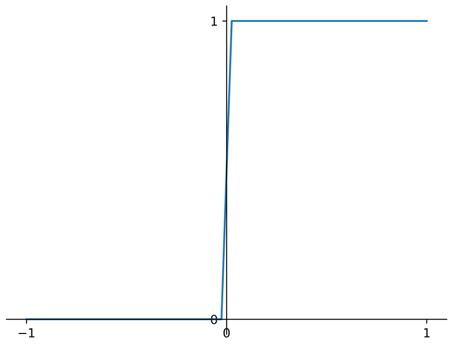


Figure 5: Binary step activation

The RELU activation function:

$$f(x) = \begin{cases} 0 & \text{if} x < 0 \\ x & \text{if} x \geq 0 \end{cases} = \max(0, x) \tag{4}$$

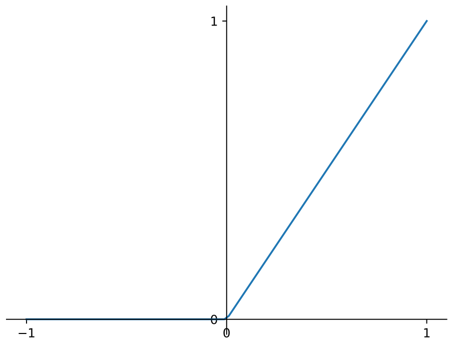


Figure 6: Rectified linear unit (RELU) activation function

2.1.2 Convolutional Neural Networks

Although the standard Densely Connected Deep Neural Networks are quite useful in many applications, it is due to the fully connected property that the number of parameters increases dramatically as more layers and nodes are added. When the number of inputs of such a DNN is relatively large, this problem becomes apparent: this would often result in networks that are very hard to train and run efficiently.

An example of such an input is an image: simple 720p images consists of 1280x720 pixels, each one with some rgb value. The final input shape is therefore (1280, 720, 3). This is where convolutional neural networks (covnets) come in. The main difference between CNN's and these other networks is the fact that densely connected neural networks learn global patterns, whereas covnets learn local patterns. When a covnet has learned a certain pattern in an image, for example, it will be able to locate it everywhere in the input image, not just in its original location. This strong generalization makes covnets much more powerful in image analysis applications.

Convolutional neural networks can then learn the spatial hierarchies of these previously learned patterns. An example can be seen in figure 7

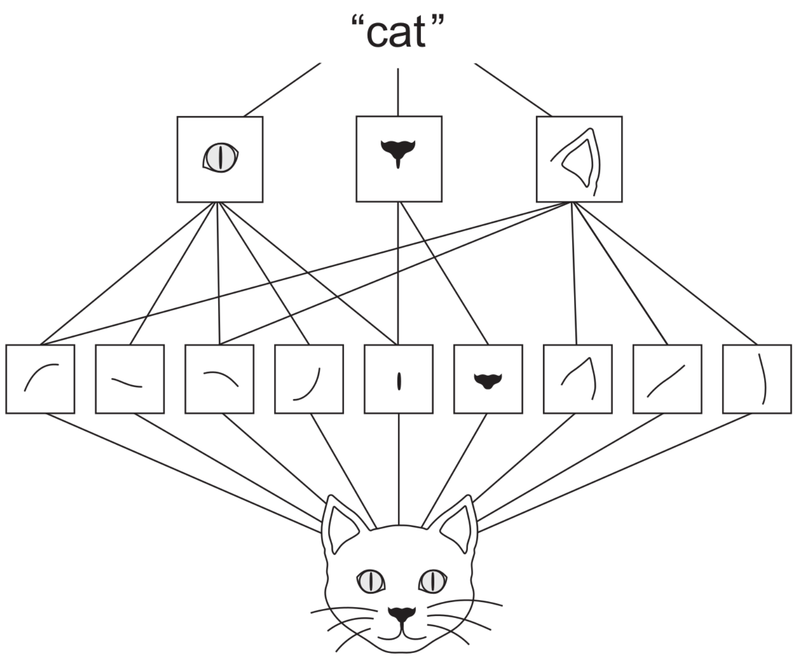


Figure 7: A visualization of a CNN identifying a cat [1]

The first layers learn the fundamental properties: edges, shapes, contrasts etc. These low-level-properties are combined in later layers to identify higher level properties of the image. In the final layer, an output is generated. In the case of the mentioned example, as we move closes to this final layer, we move further away of the input (an image of a cat) and closer to the final representation (the word "cat", or in most cases a number that classifies the image as a cat instead of the word "cat").

We can visualize the activations in layers in a neural network, as seen in figure 8.

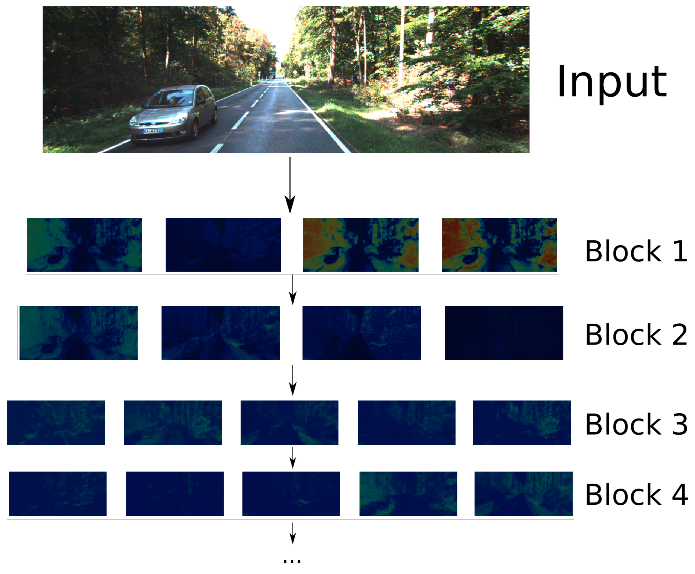


Figure 8: Activations of layers in a CNN

These activations seem to show the same properties as the cat CNN from figure 7, layers in later blocks contain more abstract versions of the original image.

3 Jetson Nano

This section contains additional information on the Jetson Nano.

3.1 SSH

The following code can be used to ssh into the Jetson Nano and open a shell in the workfolder:

```
ssh -t -Y jetson@$ip 'cd ~/$path_to_work_folder; bash'
```

Where `$ip` denotes the ip of the Jetson Nano and `$path_to_work_folder` the path on the Jetson Nano where the workfolder is located. To run python scripts using graphics on the Jetson Nano itself, instead of on the current machine, we can issue the following command:

```
export DISPLAY=:0
```

3.2 SSHFS

We found that the easiest way to work with the Jetson Nano code was via sshfs, this can be done using the bash script shown below:

```
#create folder to mount jetson to
mkdir -p JetsonFolder

#Mount Jetson workfolder
sudo sshfs -o allow_other jetson@192.168.2.14:/home/ JetsonFolder
```

192.168.2.14 should be interchanged with the appropriate Jetson IP on the (local) network.

3.3 Tensorflow installation

Custom version of Tensorflow have been released by NVIDIA, catering especially to the Jetson platform (Jetson Nano, as well as Jetson TX2 and Jetson AGX Xavier). A step-by-step installation guide can be found using the following link: `https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html`

3.4 Including cuda in path

Some problems may arise when using CUDA, we can fix most of these by making sure CUDA can be found in `$PATH`. We can do this by issuing the commands (or adding the following commands to .bashrc):

```
export PATH=$PATH:/usr/local/cuda/bin
export CUDADIR=/usr/local/cuda
```

3.5 Memory/Performance issues

The Jetson Nano seems to run out of memory relatively quickly, this results in an OOM-error every so often. Restarting the device will often fix this problem but this is no long-term solution. This section describes some methods used to counter OOM-errors that were used during testing or model conversion.

- Testsettings
  To save as much memory as possible, we can use a small amount of commands to get the maximum performance from the Jetson Nano:

  ```
  sudo nvpmodel -m 0                        #max power usage (About 10W)
  sudo jetson_clocks                        #set jetson clock to max
  sudo systemctl isolate multi-user.target  #one time headless run
  ```

- Kill running processes
  Crashed programs often take up a lot of memory, to clear all python3 processes of earlier tests run the command:

  ```
  pkill -9 python3
  ```

  To kill all python3 processes.

- Clear cache
  To free up some memory we can clear the cache using:

  ```
  sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches' #clear cache
  ```

- Increase Swap-Size
  Although the default 2gb of swap should be enough for most tasks, more can be allocated when using TF-TRT conversion, as this requires much more memory. This method is not advisable for performance-dependent tasks (and was not used during test-benchmarking) since the allocation of more swap memory is known to degrade performance (make sure the

allocation is only temporary).

We can use the same method that is used on Ubuntu to allocate more memory. While editing the Jetson specific allocation file is also possible, this solution is more permanent and requires a restart.

1. Create a swapfile (replace 4G with amount of gb needed):
   `sudo fallocate -l 4G /swapfile`
2. Set permissions for swapfile
   `sudo chmod 600 /swapfile`
3. Make file a swapfile
   `sudo mkswap /swapfile`
4. Turn swap on (not permanently)
   `sudo swapon /swapfile`
5. When done, deactivate swap
   `sudo swapoff /swapfile`

More information can be found (e.g. for permanent activation) using the following link:
https://linuxize.com/post/how-to-add-swap-space-on-ubuntu-18-04/

3.6 CUDNN version

It became apparent that just calling the model_load function takes several (about 10) minutes to complete. Running the model and a GUI resulted in a RAM-usage of about 3400/3957mb. This problem turned out to (partially) be an incompatible CUDNN version. Installing a new CUDNN version on the Jetson is, if we are to believe NVIDIA, impossible. Further research on the internet supported this conclusion. This means that a new version of Jetpack should used to flash a new operating system on the Jetson Nano, also updating the CUDNN version. This fixed the issue in our case.

3.7 JetPack 4.4

As of July 20202, Jetpack 4.4 is the most recent version of the JetPack package. This version was used during testing/training of the models. Jetpack 4.4 has some new features compared to earlier version, the main ones that are of interest for this project are:

- cuDNN 8.0
- CUDA 10.2
- TensorRT 7.1.3

By consulting the Tensorflow comptibility table ([2]) we might conclude that these versions support all Tensorflow and Keras versions up to date as of July 2020.

3.8 Python Package List

The following list contains all packages installed on our Jetson Nano during testing, obtained via `pip3 --freeze`

```
absl-py==0.7.1
Adafruit-GPIO==1.0.3
Adafruit-MotorHAT==1.4.0
Adafruit-PureIO==0.2.3
Adafruit-SSD1306==1.6.2
apt-clone==0.2.1
apturl==0.5.2
asn1crypto==0.24.0
astor==0.7.1
astunparse==1.6.3
attrs==19.1.0
backcall==0.1.0
beautifulsoup4==4.6.0
bleach==3.1.0
blinker==1.4
Brlapi==0.6.6
cachetools==4.1.1
certifi==2019.3.9
chardet==3.0.4
cryptography==2.1.4
cupshelpers==1.0
cycler==0.10.0
decorator==4.4.0
defer==1.0.6
defusedxml==0.5.0
distro-info===0.18ubuntu0.18.04.1
entrypoints==0.3
feedparser==5.2.1
future==0.17.1
futures==3.1.1
gast==0.2.2
google-auth==1.18.0
google-auth-oauthlib==0.4.1
google-pasta==0.2.0
graphsurgeon==0.3.2
grpcio==1.19.0
h5py==2.10.0
html5lib==0.999999999
httplib2==0.9.2
idna==2.8
ipykernel==5.1.0
ipython==7.4.0
ipython-genutils==0.2.0
ipywidgets==7.4.2
jedi==0.13.3
jetbot==0.3.0
```

```
Jinja2==2.10
jsonschema==3.0.1
jupyter==1.0.0
jupyter-client==5.2.4
jupyter-console==6.0.0
jupyter-core==4.4.0
jupyterlab==0.35.4
jupyterlab-server==0.2.0
keract==4.2.2
Keras==2.3.0
Keras-Applications==1.0.8
Keras-Preprocessing==1.1.2
keyring==10.6.0
keyrings.alt==3.0
kiwisolver==1.2.0
language-selector==0.1
launchpadlib==1.10.6
lazr.restfulclient==0.13.5
lazr.uri==1.0.3
louis==3.5.0
lxml==4.2.1
macaroonbakery==1.1.3
Mako==1.0.7
Markdown==3.0.1
MarkupSafe==1.0
mistune==0.8.4
mock==3.0.5
nbconvert==5.4.1
nbformat==4.4.0
notebook==5.7.6
numpy==1.16.1
oauth==1.0.1
oauthlib==3.1.0
olefile==0.45.1
opt-einsum==3.2.1
PAM==0.4.2
pandocfilters==1.4.2
parso==0.3.4
pbr==5.4.5
pexpect==4.6.0
pickleshare==0.7.5
Pillow==5.1.0
portpicker==1.3.1
prometheus-client==0.6.0
prompt-toolkit==2.0.9
protobuf==3.12.2
psutil==5.6.1
ptyprocess==0.6.0
py-cpuinfo==5.0.0
pyasn1==0.4.8
```

```
pyasn1-modules==0.2.8
pybind11==2.5.0
pycairo==1.16.2
pycrypto==2.6.1
pycups==1.9.73
Pygments==2.3.1
pygobject==3.26.1
PyICU==1.9.8
PyJWT==1.5.3
pymacaroons==0.13.0
PyNaCl==1.1.2
pyRFC3339==1.0
pyrsistent==0.14.11
python-apt==1.6.5+ubuntu0.2
python-dateutil==2.8.0
python-debian==0.1.32
pytz==2018.3
pyxdg==0.25
PyYAML==3.12
pyzmq==18.0.1
qtconsole==4.4.3
requests==2.21.0
requests-oauthlib==1.3.0
requests-unixsocket==0.1.5
rsa==4.6
scipy==1.4.1
SecretStorage==2.3.1
Send2Trash==1.5.0
simplejson==3.13.2
six==1.15.0
spidev==3.4
ssh-import-id==5.7
system-service==0.3
systemd-python==234
tensorboard==2.0.2
tensorboard-plugin-wit==1.7.0
tensorflow-estimator==2.0.1
tensorflow-gpu==2.0.0+nv19.11.tf2
tensorrt==5.0.6.3
termcolor==1.1.0
terminado==0.8.1
testpath==0.4.2
testresources==2.0.1
torch==1.0.0a0+18eef1d
torchvision==0.2.2.post3
tornado==6.0.2
tqdm==4.47.0
traitlets==5.0.0.dev0
ubuntu-drivers-common==0.0.0
uff==0.5.5
```

```
unattended-upgrades==0.1
unity-scope-calculator==0.1
unity-scope-chromiumbookmarks==0.1
unity-scope-colourlovers==0.1
unity-scope-devhelp==0.1
unity-scope-firefoxbookmarks==0.1
unity-scope-manpages==0.1
unity-scope-openclipart==0.1
unity-scope-texdoc==0.1
unity-scope-tomboy==0.1
unity-scope-virtualbox==0.1
unity-scope-yelp==0.1
unity-scope-zotero==0.1
urllib3==1.24.1
virtualenv==15.1.0
wadllib==1.3.2
wcwidth==0.1.7
webencodings==0.5
Werkzeug==0.15.1
widgetsnbextension==3.4.2
wrapt==1.12.1
xkit==0.0.0
zope.interface==4.3.2
```

3.9 Implementation

Real-world implementation of the model can be done by using a Jetson Nano in combination with a Yeti Borg. The output obstacle prediction probabilities can interpreted by a simple controlling algorithm that drives the Yeti Borg towards the location with the most space.

The implemented models, already show some promising results regarding real-world performance in our small-scale application. This is especially impressive considering that the training dataset uses very different areas than the current application (as can be seen in figure 9).

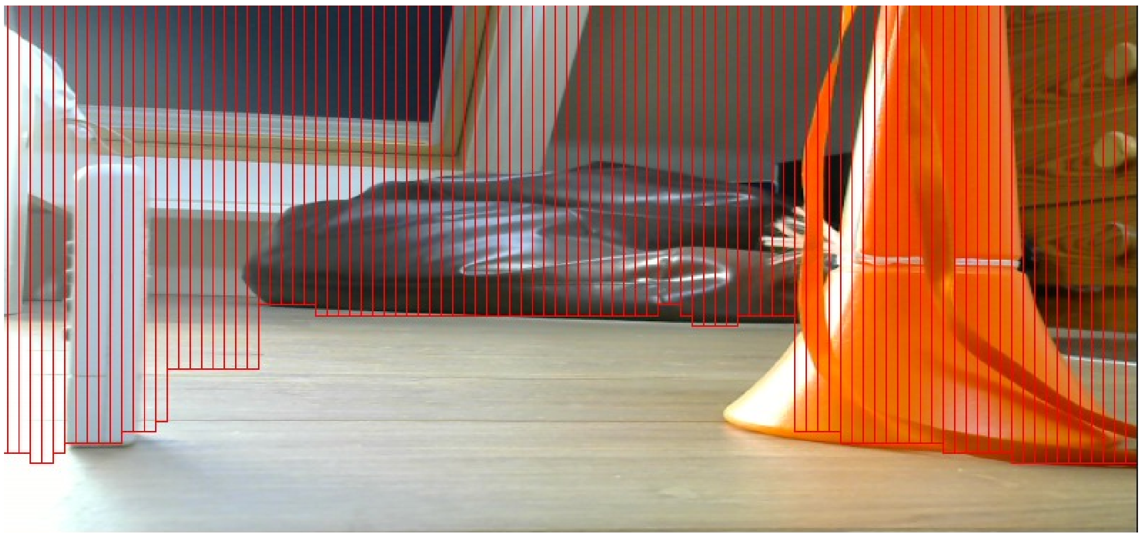


Figure 9: Ad-Hoc sample output of a small real-life situation, with some simple obstacles

3.10 Prediction and label format

The model output is a prediction array of dimensions $[X_{bins}, Y_{bins}]$ and a label array of dimensensions $[X_{bins}, 2]$. The structure of the prediction array is as follows:

$$input = [[0.1, 0.1, ..., \hat{P}_{Y_bins}], [column2], ...,]$$

Where the array [0.1...] denotes the output of each row-bin in the first column. A higher number at a certain row-bin indicates a higher predicted chance for an obstacle to be at that row in the current column. $P_{Y_{bins}}$ denotes the predicted probability that the last Y-bin contains the obstacle. This array is specified for all columns.

The structure of the label array is as follows:

$$label = [[1, 15.2], [0, 15], ...,]$$

$[1, 15.2]$ Indicates that the first column contains a labeled obstacle at bin 15.2. $[0, 15]$ indicates that the second column contains no prediction.

0.5 is subtracted from the labels in the base training code, to point to the middle of each bin. This 0.5 should be taken into account while comparing the dataset figures to the prediction values.

4 Errors and fixes Jetson Nano

Some common errors with fixes are listed here:

- `ImportError: /usr/lib/aarch64-linux-gnu/libgomp.so.1: cannot allocate memory in static TLS block`
  This error is causes by importing tensorflow before CV2, this can be solved by changing the order of imports https://github.com/opencv/opencv/issues/14884, this can also be fixed by using:

- Wrong cuDNN version errors
  while it is normally possible to freely install cuDNN versions as needed, cuDNN versions are often dependent on the version of Jetpack used to flash the Jetson Nano. The version of L4T can be retrieved, at which point this information can be used to get the original Jetpack version (as these installation versions are linked 1:1 as of July 2020). The best advice in this case is to reflash using the Jetpack version with the appropriate software version.

5 General Information

- Jetpack version 4.2 does not seem to support CUDA versions over 10.0. This results in some problems when using tensorflow 2.2.0.
- Tensorflow version 1.15.0 does seem to support cuda 10.0 https://www.tensorflow.org/install/source#tested_build_configurations. Installing both tensorflow 2.2.0 and tensorflow-gpu seems to default to the tensorflow 2.2.0 version, which is incompatible with the installed CUDA/CUDNN versions.
- Checking the L4T version running on the jetson nano using:
  cat /etc/nv_tegra_release on board

  In our case, this resulted in:

  R32 (release), REVISION: 1.0, GCID: 14531094, BOARD: t210ref, EABI: aarch64, DATE: Wed Mar 13 07:46:13 UTC 2019

  This seems to suggest the installation of JetPack 4.2 The L4T version are directly linked to versions of Jetpack, we can use to our advantage to get compatibility information.
- Tensoflow does not handle list-inputs very well. Conversion to tensor format (using `tf.to_tensor()` or `tf.constant()`) should only be done on NUMPY arrays. List conversions are up to 40x slower.

6 CSV results

6.1 Memory Allocation

These CSV files are very large, please see the `Result` folder in the project repository [4].

6.2 Factorization

”model-name”	”num-params”	”test-pc”	”time”	”total-predictions”	”average-error”	”total-error”	”average-wall-time-per-prediction”	”total-wall-time”	”average-proc-time-per-prediction”	”total-proc-time”	”err-arr-bins-per-bin”	”avg-loss”	”total-loss”
”Fac-final-H5”	27426674	”jetson-desktop”	”2020 08 03 23 45 57”	27771	6.04532053483404	167884.596572876	1.73094444237307	1317.2487206459	0.152715052386334	116.216154866	6	3.00442938391075	2286.37076115608
”Fac-best-H5”	27426674	”jetson-desktop”	”2020 08 04 00 14 37”	27771	5.49041966783108	152474.444595337	1.78218558301437	1356.24322867394	0.154825191478318	117.821970715	6	3.00439781992257	2286.34674096107

6.3 Architecture Downscaling

6.3.1 Keras (H5)

”model-name”	”num-params”	”test-pc”	”time”	”total-predictions”	”average-error”	”total-error”	”average-wall-time-per-prediction”	”total-wall-time”	”average-proc-time-per-prediction”	”total-proc-time”	”err-arr-bins-per-bin”	”avg-loss”	”total-loss”
”0.05-ds-h5-047”	80579	”jetson-desktop”	”2020 08 01 20 22 18”	27771	1.00397835108876	27881.4827880859	0.158396026308057	120.539376020432	0.12758329421682	97.0908868990001	6	1.31566325934425	1001.21974036098
”0.10-ds-h5-048”	318613	”jetson-desktop”	”2020 08 01 19 25 35”	27771	0.859994967495213	23882.9202423096	0.172603322391284	131.351128339767	0.128357393433639	97.6799764029994	6	1.18540498167864	902.093191057444
”0.15-ds-h5-050”	714191	”jetson-desktop”	”2020 08 01 19 30 40”	27771	0.755216721106879	20973.1235618591	0.201021116133902	152.977069377899	0.12938657612615	98.4631844320001	6	1.05776175683298	804.956696949899
”0.20-ds-h5-049”	1267378	”jetson-desktop”	”2020 08 01 19 35 18”	27771	0.770146880957362	21387.7490310669	0.225144378459093	171.33487200737	0.131900314868595	100.376139615	6	1.04558547215142	795.690544307232
”0.25-ds-h5-043”	1984386	”jetson-desktop”	”2020 08 01 19 40 04”	27771	0.706823051952362	19629.182975769	0.234071443647969	178.128368616104	0.130881187505914	99.6005836920002	6	1.00743162158128	766.655464023352
”0.40-ds-h5-045”	5033359	”jetson-desktop”	”2020 08 01 20 28 42”	27771	0.659163689425327	18305.6348190308	0.365072564289228	277.820221424103	0.135079516856768	102.795512328	6	0.957964037744507	729.01063272357
”0.50-ds-h5-025”	7879890	”jetson-desktop”	”2020 08 01 20 35 51”	27771	0.725450932436572	20146.497844696	0.417764509867745	317.918792009354	0.136177320145861	103.630940631	6	1.02524480178619	780.211294159293
”0.60-ds-h5-043”	11298293	”jetson-desktop”	”2020 08 01 20 44 59”	27771	0.622165159189956	17278.1486358643	0.574006739803119	436.819128990173	0.136397422049934	103.79843818	6	0.929025194648061	706.988173127174
”0.70-ds-h5-039”	15387058	”jetson-desktop”	”2020 08 01 20 55 46”	27771	0.663795881905634	18434.2754364014	0.701443297790948	533.798349618912	0.138355665010512	105.288661073	6	0.940530648565167	715.743823558092
”0.80-ds-h5-036”	20061587	”jetson-desktop”	”2020 08 01 21 08 26”	27771	0.629844079401053	17491.3999290466	0.848960317544022	646.058801651001	0.137758286023653	104.834055664	6	0.918753260372111	699.171231143177
”0.85-ds-h5-033”	22654885	”jetson-desktop”	”2020 08 01 21 21 52”	27771	0.628012986526745	17440.5486488342	0.913132355184969	694.893722295761	0.137137557065703	104.361680927	6	0.921715751143464	701.425686620176
”0.90-ds-h5-036”	25405199	”jetson-desktop”	”2020 08 01 21 36 43”	27771	0.623228145029448	17307.6688156128	1.02978943619246	783.669760942459	0.138786015721419	105.616157964	6	0.92417482922096	703.29704503715
”0.95-ds-h5-036”	28313650	”jetson-desktop”	”2020 08 01 21 52 19”	27771	0.627866242453979	17436.4734191895	1.08118146947743	822.779098272324	0.13820953260841	105.177454315	6	0.903105822850738	687.263531189412
”1.0-StixelNetV2-045”	31404402	”jetson-desktop”	”2020 08 01 22 13 18”	27771	0.615519735536784	17093.598575592	1.19115191314286	906.466605901718	0.138731027374507	105.574311832	6	0.933958064568928	710.742087136954

6.3.2 Tensorflow (PB)

”model-name”	”paremeters(beforetrtconversion)”	”test-pc”	”time”	”total-predictions”	”average-error”	”total-error”	”average-wall-time-per-prediction”	”total-wall-time”	”average-proc-time-per-prediction”	”total-proc-time”	”err-arr-bins-per-bin”	”avg-loss”	”total-loss”
”0.05-ds-pb”	80579	”jetson-desktop”	”2020 08 06 14 58 24”	27771	1.00397835108876	27881.4827880859	0.047398950674531	36.0706014633179	0.017486536833115	13.3072545300003	6	1.31566324865301	1001.21973222494
”0.10-ds-pb”	318613	”jetson-desktop”	”2020 08 06 15 01 49”	27771	0.859994967495213	23882.9202423096	0.066951549946087	50.9501295089722	0.017833605969777	13.5713741430002	6	1.18540498733755	902.093195363879
”0.15-ds-pb”	714191	”jetson-desktop”	”2020 08 06 15 05 41”	27771	0.755216721106879	20973.1235618591	0.093366893612916	71.0522060394287	0.017941628473061	13.6535792679996	6	1.05776175683298	804.956696949899
”0.20-ds-pb”	1267378	”jetson-desktop”	”2020 08 06 15 11 23”	27771	0.770146880957362	21387.7490310669	0.118772199996668	90.385644197464	0.019692812700394	14.9862304650001	6	1.04558548577982	795.69055467844
”0.25-ds-pb”	1984386	”jetson-desktop”	”2020 08 06 15 15 32”	27771	0.706823051952362	19629.182975769	0.122858336031359	93.4951937198639	0.018111706113009	13.7830083520001	6	1.00743162158128	766.655464023352
”0.40-ds-pb”	5033359	”jetson-desktop”	”2020 08 06 15 21 35”	27771	0.659163689425327	18305.6348190308	0.255419667540962	194.374366998672	0.023117782678056	17.5926326180002	6	0.95796403817529	729.010633051395
”0.50-ds-pb”	7879890	”jetson-desktop”	”2020 08 06 15 28 45”	27771	0.725450932436572	20146.497844696	0.355590299613842	270.604218006134	0.02312485404205	17.598013926	6	1.02524480730804	780.211298361421

6.3.3 TRT (FP16)

”model-name”	”test-pc”	”time”	”total-predictions”	”average-error”	”total-error”	”average-wall-time-per-prediction”	”total-wall-time”	”average-proc-time-per-prediction”	”total-proc-time”	”err-arr-bins-per-bin”	”avg-loss”	”total-loss”
”0.05-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 00 44 16”	27771	1.00396639642011	27881.1507949829	0.027528500901572	20.9491891860962	0.009324558901446	7.09598932400021	6	1.31567416354798	1001.22803846002
”0.10-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 00 47 47”	27771	0.85856263047141	23843.1428108215	0.030213925590966	22.9927973747253	0.008473169809462	6.44808222500025	6	”nan”	”nan”
”0.15-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 00 51 20”	27771	0.754416815999613	20950.9093971252	0.041361415276546	31.4760370254517	0.008753357458607	6.66130502599964	6	1.05780545107129	804.989948265255
”0.20-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 00 55 14”	27771	0.769929611207703	21381.7152328491	0.048157640888249	36.6479647159576	0.008624867583442	6.56352423099942	6	1.04577411012173	795.834097802639
”0.25-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 01 00 17”	27771	0.707411551457496	19645.5261955261	0.051814678620728	39.4309704303741	0.008588584557162	6.53591284800001	6	1.00761184609359	766.792614877224
”0.40-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 01 09 18”	27771	0.660092207701316	18331.4207000732	0.10774294242909	81.9923791885376	0.009499120859394	7.2288309739991	6	”nan”	”nan”
”0.50-ds-trt-FP16”	”jetson-desktop”	”2020 08 06 01 22 20”	27771	0.724682462077632	20125.1566543579	0.122203009200629	92.9964900016785	0.009997945465177	7.60843649899937	6	1.02519630330386	780.174386814237

6.3.4 TRT (INT8)

"model-name"	"test-pc"	"time"	"total-predictions"	"average-error"	"total-error"	"average-wall-time-per-prediction"	"total-wall-time"	"average-proc-time-per-prediction"	"total-proc-time"	"err-arr-bins-per-bin"	"avg-loss"	"total-loss"
"0.05-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 17 40 04"	27771	1.00396639642011	27881.1507949829	0.026204868309131	19.9419047832489	0.009175805605782	6.98278806600005	6	1.31567416354798	1001.22803846002
"0.10-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 17 42 59"	27771	0.85856263047141	23843.1428108215	0.030634301407422	23.312703371048	0.008829068921156	6.71892144899998	6	"nan"	"nan"
"0.15-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 17 46 20"	27771	0.752548782333798	20899.0322341919	0.04358784566571	33.1703505516052	0.008885563680683	6.76191396100009	6	1.05756986403591	804.810666531324
"0.20-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 17 50 09"	27771	0.769929611207703	21381.7152328491	0.048458503517622	36.8769211769104	0.008392656982916	6.38681196399901	6	1.04577411012173	795.834097802639
"0.25-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 17 54 46"	27771	0.707411551457496	19645.5261955261	0.051674116300064	39.3240025043488	0.008585371164258	6.53346745600007	6	"nan"	"nan"
"0.40-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 18 03 36"	27771	0.660225744770445	18335.12915802	0.121277733012035	92.2923548221588	0.010740083572929	8.17320359899884	6	0.957977632675001	729.020978465676
"0.50-ds-trt-INT8"	"jetson-desktop"	"2020 08 12 18 15 56"	27771	0.725156434675798	20138.3193473816	0.142622443285628	108.535679340363	0.011945681441521	9.09066357699737	6	1.02521157521221	780.186008736491

6.4 Resnet Layers

6.4.1 Keras (H5)

"model-name"	"num-params"	"test-pc"	"time"	"total-predictions"	"average-error"	"total-error"	"average-wall-time-per-prediction"	"total-wall-time"	"average-proc-time-per-prediction"	"total-proc-time"	"err-arr-bins-per-bin"	"avg-loss"	"total-loss"
"0.4-Res-48"	5033359	"jetson-desktop"	"2020 08 03 14 59 01"	27771	0.694571458758081	19288.9439811707	0.375871030165864	286.037853956223	0.137226593107753	104.429437355	6	0.988996886529528	752.626630648971
"0.2-Res-50"	1267378	"jetson-desktop"	"2020 08 03 15 06 01"	27771	0.75623555846357	21001.4176940918	0.229769658165128	174.854709863663	0.129903914494087	98.8568789300001	6	1.06229486708063	808.40639384836
"1.0-Res-35"	31404402	"jetson-desktop"	"2020 08 03 16 45 19"	27771	0.649359397397799	18033.3598251343	1.20758798087943	918.974453449249	0.14016819703548	106.667997944	6	0.939654692057789	715.077220655978

6.5 Filter Change (Stixel5x5)

6.5.1 Keras (H5)

"model-name"	"num-params"	"test-pc"	"time"	"total-predictions"	"average-error"	"total-error"	"average-wall-time-per-prediction"	"total-wall-time"	"average-proc-time-per-prediction"	"total-proc-time"	"err-arr-bins-per-bin"	"avg-loss"	"total-loss"
"0.05-5x5-keras-019"	111496	"jetson-desktop"	"2020 08 08 14 25 26"	27771	0.93635642914437	26003.5543937683	0.177403488860964	135.004055023193	0.127866049116951	97.306063378	6	1.26345937406343	961.492583662272
"0.10-5x5-keras-035"	285648	"jetson-desktop"	"2020 08 08 14 29 41"	27771	0.803921542552794	22325.7051582336	0.197957810841785	150.645894050598	0.128991469515111	98.1625083009998	6	1.12223123797214	854.017972096801
"0.15-5x5-keras-031"	523537	"jetson-desktop"	"2020 08 08 14 34 16"	27771	0.715228947473291	19862.6231002808	0.224389595051789	170.760481834412	0.129536864440209	98.5775538389994	6	1.03626109327773	788.594691984356
"0.20-5x5-keras-029"	829611	"jetson-desktop"	"2020 08 08 14 39 03"	27771	0.702376166863572	19505.6885299683	0.246399616913163	187.510108470917	0.131086920788436	99.7571467200002	6	1.00920439134603	768.004541814327
"0.40-5x5-keras-026"	2677905	"jetson-desktop"	"2020 08 08 14 45 40"	27771	0.658296929245047	18281.5640220642	0.375431940214079	285.703706502914	0.135327726291721	102.984399708	6	0.994896448473109	757.116197288036
"0.50-5x5-keras-018"	4015709	"jetson-desktop"	"2020 08 08 14 52 44"	27771	0.649476785980188	18036.6198234558	0.416697845822408	317.107060670853	0.135595045885677	103.187829919	6	0.952653372031239	724.969216115773

6.5.2 Tensorflow (PB)

"model-name"	"Parameters(beforeTRTconversion)"	"test-pc"	"time"	"total-predictions"	"average-error"	"total-error"	"average-wall-time-per-prediction"	"total-wall-time"	"average-proc-time-per-prediction"	"total-proc-time"	"err-arr-bins-per-bin"	"avg-loss"	"total-loss"
"0.05-5x5-tf"	111496	"jetson-desktop"	"2020 08 07 22 48 11"	27771	0.93635642914437	26003.5543937683	0.068872763512169	52.4121730327606	0.017799178821288	13.5451750830003	6	1.26345937406343	961.492583662272
"0.10-5x5-tf"	285648	"jetson-desktop"	"2020 08 07 22 51 43"	27771	0.803921542552794	22325.7051582336	0.088665635764364	67.4745488166809	0.018235076115637	13.876892924	6	1.12223123797214	854.017972096801
"0.15-5x5-tf"	523537	"jetson-desktop"	"2020 08 07 22 55 33"	27771	0.715228947473291	19862.6231002808	0.116054387349493	88.3173887729645	0.019770563339027	15.0453987009998	6	1.03626109327773	788.594691984356
"0.20-5x5-tf"	829611	"jetson-desktop"	"2020 08 07 22 59 37"	27771	0.702376166863572	19505.6885299683	0.13815843573381	105.13856959343	0.021193797530881	16.1284799210001	6	1.00920439134603	768.004541814327
"0.40-5x5-tf"	2677905	"jetson-desktop"	"2020 08 07 23 05 20"	27771	0.658296929245047	18281.5640220642	0.265296311008789	201.890492677689	0.023142830662286	17.6116941339999	6	0.994896448473109	757.116197288036
"0.50-5x5-tf"	4015709	"jetson-desktop"	"2020 08 07 23 11 34"	27771	0.649476785980188	18036.6198234558	0.304919178006527	232.043494462967	0.022409664130092	17.0537544030001	6	0.952653372031239	724.969216115773

6.5.3 TRT (FP16)

"model-name"	"Parameters(beforeTRTconversion)"	"test-pc"	"time"	"total-predictions"	"average-error"	"total-error"	"average-wall-time-per-prediction"	"total-wall-time"	"average-proc-time-per-prediction"	"total-proc-time"	"err-arr-bins-per-bin"	"avg-loss"	"total-loss"
"0.05-5x5-trt-FP16"	111496	"jetson-desktop"	"2020 08 06 14 15 43"	27771	0.936156442221951	25998.0005569458	0.027950795032035	21.2705550193787	0.008878605191853	6.75661855100009	6	"nan"	"nan"
"0.10-5x5-trt-FP16"	285648	"jetson-desktop"	"2020 08 06 14 18 24"	27771	0.804711009612078	22347.629447937	0.034107421293522	25.9557476043701	0.009031105977661	6.87267164899993	6	"nan"	"nan"
"0.15-5x5-trt-FP16"	523537	"jetson-desktop"	"2020 08 06 14 21 33"	27771	0.715128823586275	19859.8425598145	0.044753200591159	34.0571856498718	0.009127319529566	6.94589016199994	6	"nan"	"nan"
"0.20-5x5-trt-FP16"	829611	"jetson-desktop"	"2020 08 06 14 25 04"	27771	0.702322179645891	19504.189250946	0.051880129690083	39.4807786941528	0.009671260076216	7.35982891800033	6	"nan"	"nan"
"0.40-5x5-trt-FP16"	2677905	"jetson-desktop"	"2020 08 06 14 31 18"	27771	0.659578538542799	18317.1555938721	0.110003932860145	83.7129929065704	0.009959394972405	7.57909957399983	6	"nan"	"nan"
"0.50-5x5-trt-FP16"	4015709	"jetson-desktop"	"2020 08 06 14 39 00"	27771	0.649118232871282	18026.6624450684	0.123148164047362	93.7157528400421	0.010783050275952	8.20590125999905	6	"nan"	"nan"

6.5.4 TRT (INT8)

”model-name”	”test-pc”	”time”	”total-predictions”	”average-error”	”total-error”	”average-wall-time-per-prediction”	”total-wall-time”	”average-proc-time-per-prediction”	”total-proc-time”	”err-arr-bins-per-bin”	”avg-loss”	”total-loss”
”0.05-5x5-trt-INT8”	”jetson-desktop”	”2020 08 11 16 40 20”	27771	0.93515211956102	25970.1095123291	0.029539357660321	22.4794511795044	0.009416675959264	7.16609040499996	6	1.26701682555378	964.199804246426
”0.10-5x5-trt-INT8”	”jetson-desktop”	”2020 08 11 16 43 04”	27771	0.804431017343769	22339.8537826538	0.034414683942256	26.1895744800568	0.009418005057819	7.16710184900001	6	”nan”	”nan”
”0.15-5x5-trt-INT8”	”jetson-desktop”	”2020 08 11 16 46 23”	27771	0.715128823586275	19859.8425598145	0.044934007717338	34.1947798728943	0.009348030001314	7.11385083099992	6	”nan”	”nan”
”0.20-5x5-trt-INT8”	”jetson-desktop”	”2020 08 11 16 50 03”	27771	0.702527686210391	19509.8963737488	0.051536062543872	39.2189435958862	0.009119672749014	6.94007096199978	6	”nan”	”nan”
”0.40-5x5-trt-INT8”	”jetson-desktop”	”2020 08 11 16 56 23”	27771	0.659112797399519	18304.221496582	0.117340964180725	89.2964737415314	0.010922084168201	8.31170605200057	6	”nan”	”nan”
”0.50-5x5-trt-INT8”	”jetson-desktop”	”2020 08 11 17 04 11”	27771	0.649175401132471	18028.2500648499	0.130390015600859	99.2268018722534	0.011202674176084	8.52523504799956	6	”nan”	”nan”

6.6 0.05 Calibration runs

6.6.1 TRT (FP16)

”model-name”	”Runs”	” test-pc”	” time”	” total-predictions”	” average-error”	” total-error”	” average-wall-time-per-prediction”	” total-wall-time”	” average-proc-time-per-prediction”	” total-proc-time”	” err-arr-bins-per-bin”	” avg-loss”	” total-loss”
”0.05-trt-FP16-0runs”	0	”jetson-desktop”	”2020:08:05:17:08:13”	27771	1.00396639642011	27881.1507949829	0.025924616198346	19.7286329269409	0.008709388159001	6.62784438899989	6	”nan”	”nan”
”0.05-trt-FP16-5runs”	5	”jetson-desktop”	”2020:08:05:17:11:28”	27771	1.00376471966276	27875.5500297546	0.026624108112751	20.2609462738037	0.008797554575558	6.69493903199989	6	1.31577699722504	1001.30629488826
”0.05-trt-FP16-10runs”	10	”jetson-desktop”	”2020:08:05:17:14:00”	27771	1.00376471966276	27875.5500297546	0.026454245370572	20.131680727005	0.008508384675427	6.47488073799995	6	1.31577699722504	1001.30629488826
”0.05-trt-FP16-50runs”	50	”jetson-desktop”	”2020:08:05:17:17:25”	27771	1.00396639642011	27881.1507949829	0.025421314264566	19.3456201553345	0.008312436735874	6.32576435600005	6	1.31567416354798	1001.22803846002
”0.05-trt-FP16-100runs”	100	”jetson-desktop”	”2020:08:05:17:19:59”	27771	1.00376471966276	27875.5500297546	0.025691035855929	19.5508782863617	0.007759546660972	5.9050150089999	6	1.31577699722504	1001.30629488826
”0.05-trt-FP16-200runs”	200	”jetson-desktop”	”2020:08:05:17:23:15”	27771	1.00376471966276	27875.5500297546	0.026293298418043	20.0092000961304	0.008351732658344	6.35566855299991	6	1.31577699722504	1001.30629488826
”0.05-trt-FP16-300runs”	300	”jetson-desktop”	”2020:08:05:17:26:01”	27771	1.00376471966276	27875.5500297546	0.026610775783404	20.25080037117	0.008130308645203	6.1871648789998	6	1.31577699722504	1001.30629488826
”0.05-trt-FP16-400runs”	400	”jetson-desktop”	”2020:08:05:17:28:33”	27771	1.00396639642011	27881.1507949829	0.026122612959452	19.8793084621429	0.008831993726675	6.72114722599999	6	1.31567416354798	1001.22803846002
”0.05-trt-FP16-500runs”	500	”jetson-desktop”	”2020:08:05:17:39:21”	27771	1.00396639642011	27881.1507949829	0.024806302731047	18.8775963783264	0.00782207430749	5.95259854799996	6	1.31567416354798	1001.22803846002
”0.05-trt-FP16-1000runs”	1000	”jetson-desktop”	”2020:08:05:17:59:06”	27771	1.00396639642011	27881.1507949829	0.02575551194681	19.5999445915222	0.008911894658344	6.78195183499995	6	1.31567416354798	1001.22803846002

6.6.2 TRT (INT8)

”model-name”	”runs”	” test-pc”	” time”	” total-predictions”	” average-error”	” total-error”	” average-wall-time-per-prediction”	” total-wall-time”	” average-proc-time-per-prediction”	” total-proc-time”	” err-arr-bins-per-bin”	” avg-loss”	” total-loss”
”0.05-trt-INT8-1runs”	1	”jetson-desktop”	”2020:08:11:14:27:37”	27771	1.00396639642011	27881.1507949829	0.025897853791166	19.7082667350769	0.008335420726676	6.34325517300011	6	1.31567416354798	1001.22803846002
”0.05-trt-INT8-5runs”	5	”jetson-desktop”	”2020:08:11:14:30:02”	27771	1.00396639642011	27881.1507949829	0.026056439741839	19.8289506435394	0.008645433036794	6.57917454100001	6	”nan”	”nan”
”0.05-trt-INT8-10runs”	10	”jetson-desktop”	”2020:08:11:14:32:26”	27771	1.00376471966276	27875.5500297546	0.025833078502199	19.6589727401733	0.00779681639816	5.93337727899996	6	1.31577699722504	1001.30629488826
”0.05-trt-INT8-50runs”	50	”jetson-desktop”	”2020:08:11:14:34:37”	27771	1.00396639642011	27881.1507949829	0.024906667240659	18.9539737701416	0.008095744806833	6.16086179800003	6	1.31567416354798	1001.22803846002
”0.05-trt-INT8-100runs”	100	”jetson-desktop”	”2020:08:11:14:37:00”	27771	1.00376471966276	27875.5500297546	0.026500660549169	20.1670026779175	0.008541519093298	6.5000960299998	6	”nan”	”nan”
”0.05-trt-INT8-200runs”	200	”jetson-desktop”	”2020:08:11:14:39:17”	27771	1.00396639642011	27881.1507949829	0.025627765329689	19.5027294158936	0.00863776239159	6.57333718000016	6	1.31567416354798	1001.22803846002
”0.05-trt-INT8-300runs”	300	”jetson-desktop”	”2020:08:11:14:41:36”	27771	1.00396639642011	27881.1507949829	0.025080002247112	19.0858817100525	0.007992368844941	6.08219269099974	6	1.31567416354798	1001.22803846002
”0.05-trt-INT8-400runs”	400	”jetson-desktop”	”2020:08:11:14:43:53”	27771	1.00376471966276	27875.5500297546	0.026506616000904	20.1715347766876	0.00844334861498	6.4253882959999	6	1.31577699722504	1001.30629488826
”0.05-trt-INT8-500runs”	500	”jetson-desktop”	”2020:08:11:14:46:08”	27771	1.00396639642011	27881.1507949829	0.025969087685925	19.7624757289886	0.00911584539159	6.93715834299974	6	”nan”	”nan”
”0.05-trt-INT8-1000runs”	1000	”jetson-desktop”	”2020:08:11:14:48:21”	27771	1.00396639642011	27881.1507949829	0.025319210333204	19.2679190635681	0.008298620437582	6.3152501530001	6	1.31567416354798	1001.22803846002

References

[1] François Chollet. *Deep Learning with Python*. Manning, November 2017. ISBN: 9781617294433.

[2] Compatible tensorflow versions, 2020. URL: https://www.tensorflow.org/install/source#tested_build_configurations.

[3] Google deepmind, 2020. URL: https://deepmind.com/.

[4] Wouter Stokman. Project repository github. URL: https://github.com/Woutah/EmbeddedStixelNet.

[5] James Vincent. Former go champion beaten by deepmind retires after declaring ai invincible, November 2019. URL: https://www.theverge.com/2019/11/27/20985260/ai-go-alphago-lee-se-dol-retired-deepmind-defeat..