CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Computation scheduling in neural network inference on embedded hardware

Eldar Iosip

January 2020

Supervisor: Ing. Michal Sojka, Ph.D.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Iosip** Jméno: **Eldar** Osobní číslo: **406391**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Rozvrhování výpočtů inference neuronových sítí na vestavném hardware

Název diplomové práce anglicky:

Computation scheduling in neural network inference on embedded hardware

Pokyny pro vypracování:

1. Seznamte se s problematikou neuronových sítí a jejich aplikacemi pro autonomní vozidla (např. detekce chodců, rozpoznávání objektů na silnici apod.). Dále se seznamte s existujícími knihovnami a již natrénovanými sítěmi (modely) pro zmiňované aplikace.
2. Vytvořte data flow diagram (graf) výpočtů prováděných během inference natrénované neuronové sítě tak, aby tento graf šel použít jako vstup algoritmu pro rozvrhování výpočtů. Vytvořte a porovnejte tyto grafy několika různých modelů a knihoven.
3. Rozšiřte implementaci aspoň dvou různých knihoven o podporu rozvrhování běhu jednotlivých výpočtů. Ve spolupráci s vedoucím vytvořte různé rozvrhy běhu výpočtů.
4. Vytvořte jednoduchý framework pro experimentální měření níže uvedených veličin při inferenci neuronových sítí na embedded HW (např. NVIDIA Tegra X2). Implementujte podporu minimálně pro následující veličiny:
a) celkovou dobu běhu výpočtu,
b) rozdíl mezi nejpomalejším a nejrychlejším během se stejným rozvrhem, tzv. execution-time jitter (bude záviset na využiti hardwarových komponent a na kolizích v přístupu k nim) a
c) spotřebu energie a tepelný profil čipů (CPU, GPU, DRAM).

Seznam doporučené literatury:

[1] Recognition of road scene elements using deep neural networks: Monograph
[2] Zins, P., & Dagenais, M. (2019). Tracing and Profiling Machine Learning Dataflow Applications on GPU. International Journal of Parallel Programming, 1-41.
[3] Mayer, Ruben & Mayer, Christian & Laich, Larissa. (2017). The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path!. 10.1145/3154842.3154843.
[4] Keras: Custom C++ and CUDA extensions
[5] Open Neural Network Exchange Format [onnx.ai]

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Sojka, Ph.D., vestavěné systémy CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.07.2019** Termín odevzdání diplomové práce: **07.01.2020**

Platnost zadání diplomové práce: **19.02.2021**

Ing. Michal Sojka, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání | Podpis studenta

Acknowledgement / Declaration

I would like to thank my supervisor Ing. Michal Sojka, Ph.D. for his patient guidance and consultations. I would also like to thank to my family for their support.

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

Prague, 7th January 2020

.....

Abstrakt / Abstract

Cílem této práce je prozkoumat state-of-the-art způsoby detekce objektů pomocí konvolučních neuronových sítí, využívaných v oblasti autonomního řízení. Proto aby běh na vestavěných systémech byl dostatečně optimalizován, je nutné rozumět struktuře sítě a způsobu, jak se provádí její výpočet pomocí konkrétní knihovny. Hlavním cílem této práce je porovnat několik dostupných knihoven pro oblast strojového učení a popsat nezdokumentovanou vnitřní architekturu knihovny TensorFlow, aby bylo možné na základě těchto znalostí upravovat vykonávané části kódu za účelem lepšího rozvrhování jednotlivých procesů. Aby bylo možné porovnávat výsledky budoucích optimalizací na cílové platformě NVIDIA Jetson Tegra X2, je představen jednoduchý benchmark a je popsán postup, jak vyčítat spotřebu energie a tepelný profil čipů na desce.

Klíčová slova: neuronové sítě, vestavěné systémy, inference, tensorflow, yolov3, edge computing, jetson tx2

Překlad titulu: Rozvrhování výpočtů inference neuronových sítí na vestavném hardware

This thesis aims to examine the state-of-the-art solution of using convolutional neural networks to address the problem of object detection, during the autonomous driving. The effective execution of these solutions involves an in-depth understanding of used framework architectures. The main goal of the thesis is to compare several machine learning frameworks and provide a comprehensive description of the nondocumented internal architecture of the TensorFlow machine learning framework to allow future researches to introduce modifications regarding scheduling mechanisms. To properly evaluate future modifications on the target platform NVIDIA Tegra X2, the thesis introduces the benchmark and provides an instruction how to read power consumption and temperature of board components.

Keywords: neural networks, embedded systems, inference, tensorflow, yolov3, edge computing, jetson tx2

Contents /

Tables / Figures

Chapter 1
Introduction

Nowadays, car manufacturers are striving to achieve a leading position in the market by improving the autonomous driving of their vehicles. To be able to reach this goal, a car has to decide how to behave in different situations during the drive, the same way as human does. As the vehicle is not capable of this feature without knowing about the situation around, various sensors such as video cameras, LiDARs, radars, are adding a level of vision to the vehicle. Based on such input data from sensors, the software is capable of making essential decisions using a machine learning approach [1]. To be able to process the data from sensors, especially image data from in-vehicle cameras, the state-of-the-art solution is to use CNN. These networks are trained and used to detect unique features that one would like to observe and put into further processing. As the speed and quality of the detection could be dependant on the quality of the image and also on the hardware configuration, neural network models are trying to balance the amount of computational overhead and correctness of the detected result by their structure and capabilities. The architecture of the artificial neural networks and their state-of-the-art representants in a field of object detection introduces Section 3.

The objective of this thesis is to describe the internal architecture and internal scheduling and optimization processes of the open-source machine learning framework TensorFlow, to allow future researchers to introduce modifications and improve the performance of the artificial neural network inference on the embedded hardware — NVIDIA Jetson TX2.

The NVIDIA Jetson TX2 is a target platform for this thesis; it is an SoC platform with a shared DRAM memory between CPU and GPU units [2], which can reduce overall performance in crucial moments during the autonomous driving. However, on the other hand, it reduces the cost of the whole solution and makes it more power-efficient, as it requires less power to store information inside memory banks.

To introduce a future scheduling mechanism, Section 4 describes the internal processes executed during the model inference. Every network inside the TensorFlow framework represents the dataflow graph (Section 6.2) that the framework, during the preparation phases, optimizes and partitions between available accelerators (CPU, GPU, TPU, and others). This dataflow graph and its structure is a valuable input to the future scheduling algorithm to analyze network operations and prioritize them by the defined rules.

As the model needs to receive the input data, the TensorFlow Serving framework exposes an API over the TensorFlow framework. The API handles the communication between client requests and the configured model. From the point of the development, the compilation process is less demanding on the computational resources and provides a reliable way how to serve models on the production environment.

Section 6.3 introduces a simple benchmark framework suited, especially for the NVIDIA TX2 platform. The framework is capable of measuring the execution time, execution-time jitter, and provide information about the state of hardware components installed on the target platform.

Chapter 2
Related Work

Many automotive companies, such as Tesla, Google, Uber, and others, are investing in the research of autonomous driving, significantly shifting the quality of their final product. Despite the recent advancements, the vehicle is still a constrained environment. It needs to be efficient in terms of the speed of taking actions. Passengers must arrive at their destination safely, concerning the consumption of power, emitted heat, and storage capacity of the vehicle. It is necessary to define such constraints, and how to measure overall performance, to be able to have a predictable system, concerning the selected hardware, that would comply with the autonomous driving guidelines.

Considering previous facts it is important to understand the principle of autonomous driving and environment around, during the drive. Ami Woo et al., 2019 [3] described in their research the state-of-the-art autonomous driving pipeline of a production-ready solution. The pipeline consists of an object detection mechanism, followed by the tracker, which tracks the motion of nearby objects and localizer, which finds the current position of the vehicle by analyzing nearby objects. Built-in cameras provide an image input to each stage. By analyzing information gathered from sensors, the planner component is executing the next action for a vehicle. Mostly, DNNs are used in object detection (YOLO [4]) and object tracking (GOTURN [5]) components as they need to process image data. These components are capable of identifing objects of the interest — such as cars, pedestrians, bicycles, and other entities, the vehicle can reach during its operation.

Mihir Mody, 2016, [6] stressed the selection of image resolution and frame rate for front monocular camera for the ADAS. The research aims that it's crucial to react to situations around the vehicle in time, to prevent unnecessary consequences. It implies that the validation process needs to take the stopping distance and the vehicle speed into consideration. Additionally, the frame rate of the camera recording needs to be proportional to vehicle speed, to preserve lower stopping distance. Simon Thorpe et al., 1996 [7] found that human's fastest action during the drive takes under 150 ms. It means that the system as a whole has to be responsive for each incoming frame to be able to take action. In general, faster than a human does, to make the journey safe and predictable.

Shih-Chieh Lin et al., 2018 [8] depicted constraints of autonomous driving and introduced tail latency. It captures existing non-determinism, during the end-to-end performance evaluation of such a system, comparing to the mean latency used in usual benchmarks. Paper benchmarks showed that the tail latency of quantile 99.99th-percentile is always higher than mean latency. Admitting this fact, it seems mostly as the correct metric during the evaluation of the autonomous system. Additionally, power and heat constraints were estimated. System with 1 CPU and 3 GPU units reduces the driving range by 6%, which is a significant number, considering current restrictions in the battery capacity of electric vehicles. The heat produced by the device during the calculation should not change climate settings in the cabin. As a result of the research,

the cuDNN[1] library was used on a GPU unit to speed up the computation. It allowed the system to achieve significant tail latency reduction and comply with the required maximum latency, determined as 100 ms. During the testing, ASIC device performed better or equal to the GPU unit in the object detection stage. ASIC device had better characteristics in terms of power consumption by a factor of 7, comparing to the same task on GPU.

Lingyuan Wang et al., 2011 [9] observed that NVIDIA's GPU scheduling algorithm is different depending on how the CPU submits work to the GPU. The first case is when the CPU is executing OS threads having common address space. The second case is when OS processes have different address spaces. Tanya Amert et al., 2017 [10] focused on the scheduling behavior of the Jetson TX2 platform. As NVIDIA is not publicly disclosing any details about the implemented scheduling algorithm, the research uses the black-box method to deduct the behavior. The research shows that during the submission via the first case, GPU uses hierarchical FIFO scheduling, and it is capable of real-time schedulability analysis. However, during the second case, the scheduling process is less deterministic as GPU uses time slicing and ignores stream priorities, during parallel computations. The second case adds significant overhead and also variation to the execution-time benchmark.

Wagar Ali and Heechul Yun, 2017 [11] studied the worst case of shared memory between CPU and GPU architecture. They estimated a 3x slowdown when CPU is executing processes during computations on the GPU. To be able to achieve an increase in performance, they designed a mechanism called BWLOCK++ (bandwidth lock), which locks the memory when GPU intensive kernels of some critical process are running. It means that less priority tasks get fewer resources at the time of running the critical process. The implementation of such a mechanism is a modification of the OS kernel provided by the NVIDIA. The presented mechanism achieved the result close to the situation when the critical process runs on its own. During the co-run of the critical and secondary processes, the BWLOCK++ version always achieved a better result.

[1] https://developer.nvidia.com/cudnn

Chapter 3
Artificial Neural Network

Nowadays, artificial intelligence is experiencing impressive growth as it targets to get closer to the border between human and machine capabilities. One of its significant fields is a field of neural networks. It is inspired by the simplified version of a human biological neural system, which consists of basic computational units, called neurons[12]. Synapses connect neurons, and such a combination forms a network. Each neuron receives signals from its dendrites and passes the output to its axon, which is often connected to other neurons via its terminal. Such a terminal behaves as an input to other neurons that can receive a processed signal. The artificial neural network is based on this knowledge. The high-level difference between biological and artificial networks depicts Figure 3.

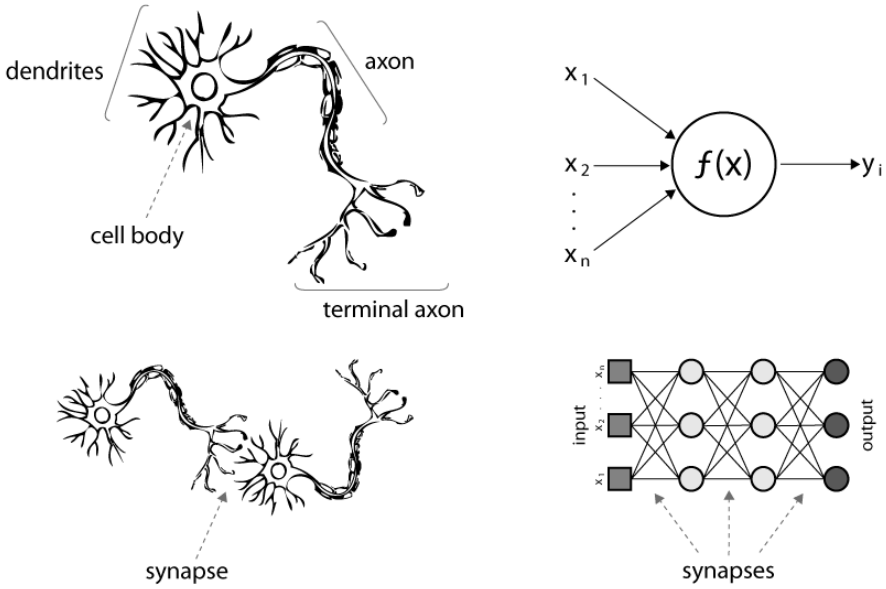

Figure 3.1. Similarity between biological and artificial neural networks [13]. The left column shows units of biological NN, where the right column shows the artificial NN alternative. The top-left image depicts the internal structure of a neuron. The bottom-left image shows the connection between two neurons defined as a synapse. The top-right image depicts that neuron is a function, which is applied to its inputs. The bottom-right image shows neurones that form connected layers — input layer on the left, output on the right and hidden layers in-between.

3.1 Architecture

The architecture of ANN is a directed, acyclic, weighted graph, where neurons are nodes and edges pass the output of one node to the input of another node. Each node performs a dot product of its inputs along with their weights. The weight defines the strength of the connection when it contributes to the final result. Each node, along with

its inputs, receives a bias. Bias is a constant that helps the model to fit the data better. After multiplications within the node are complete, the activation function determines if the node will contribute by its output to the final network result (introduces non-linearity). There are various types of commonly used activation functions — Sigmoid, Tanh, ReLU, Leaky ReLU, Maxout [14]. The following block introduces some of them:

- ReLU (Rectified Linear Unit) — $f(x) = max(x, 0)$. The activation function, often used in a field of Computer Vision due to its simple structure. It activates when the input x is greater than 0, meaning it outputs the same data as received at its input. The downside is a fragileness of its operation during the training process, as it may never activate again which leads to a partially active network and decrease of the performance.
- Leaky ReLU — $f(x) = \iota(x < 0)(\alpha x) + \iota(x >= 0)(x)$, where α is a constant of some small value. The activation function is a modification of ReLU. It solves the issue of inactive nodes as it extends the range of ReLU by modifying the value of negative x to a `near-zero` value. The disadvantage of such modification is introduced inconsistency in results [14].
- Maxout — $max(w_1 x + b_1, w_2 x + b_2)$. It's a generalized version of ReLU and leaky ReLU, which returns the maximum of its inputs. The drawback is, as it doubles the number of parameters for each node [15].

Nodes within the network are organized into layers, that represent a unique set of disconnected nodes. The most common type of layer organization is a fully-connected layer when each node in one layer is connected with each node in the adjacent layer. Layer's names are often labeled as input, hidden, or output. Only the hidden type is allowed to have more than one instance.

3.2 Training and Inference

The Artificial Neural Network is an algorithm, but it does not strictly define how many nodes and edges must be present in the network to provide satisfactory results. In general, the process of finding the optimal number of layers, the amount of nodes in each layer, their activation functions is a part of the network building process, and the weights of edges are a part of the training process. More extensive networks can represent more sophisticated functions, but they are prone to overfitting. Overfitting is a situation when the model fits a noise in data, instead of the underlying relationship [14]. During the training process, a significant amount of data forwards through the network, and it may lead to compute-intensive tasks. Training data, in case of supervised learning, is different from the real data in terms of knowing the correct result beforehand. This knowledge helps to modify the values of weights basing on the error value — backpropagation process. After the training process, the network is capable of processing new input data without any training steps involved. This process is called inference. During the inference, often, the network is optimized to exclude some parts useful for training or merge operations into larger blocks to use all underlying accelerator capabilities. The following sections describe the inference process in more detail.

3.3 Convolutional Neural Network

There are many types and multiple applications of artificial neural networks. One of the most popular networks in computer vision tasks is convolutional neural networks or

CNNs, as these tasks are based on data with a grid-like topology, such as images. The word convolutional in the name of the network implies the use of linear mathematical operation called convolution [12]. The following text will present a general overview of levels and operations involved in the CNN and describe the process of running inference.

Before diving into the details about the convolutional neural networks, it's necessary to understand how the machine interprets an image. Image is a $w * h * d$ grid, where w is the width and h is the height in pixels, and d is the depth, which represents the number of channels. As an example, the RGB image has three channels — Red, Green, and Blue. Each cell of the grid defines a triple of 8-bit integers with the value from 0 to 255, respectively. Each number establishes the intensity of the particular color. By visualizing the described grid to a human, it can capture the image by its eyes. Then, the information is subsequentially passing the neural network in the brain. The brain can categorize objects by previous observations and human experiences, gained during life. On the other hand, the machine does not have such an ability out-of-the-box. For the machine, it's just a grid of numbers, without any particular meaning. Computer vision is striving to achieve the most accurate interpretation of such a grid by simulating the same processes as a human brain does.

3.3.1 Architecture

Convolutional neural network consists of two main stages: the convolution/pooling and the fully connected layer. The first stage divides the input image into smaller blocks and analyzes them for the presence of pretrained features. The second stage, a fully connected layer, takes the output of the first stage and classifies detected features. The overall output of the network is a set of probabilities (class scores) for each trained label. These stages are visualized on the Figure 3.2.

The difference between the fully connected layer and fully connected network, introduced in Section 3.1, is that CNN uses a fully connected layer only during its second stage. Comparing to the regular neural networks, this architecture scales well for full images. It uses pre-processing and reduces the overall number of parameters in the network, that would in normal situation lead to overfitting. The following subsections describe the individual types of layers involved in the architecture of CNN.

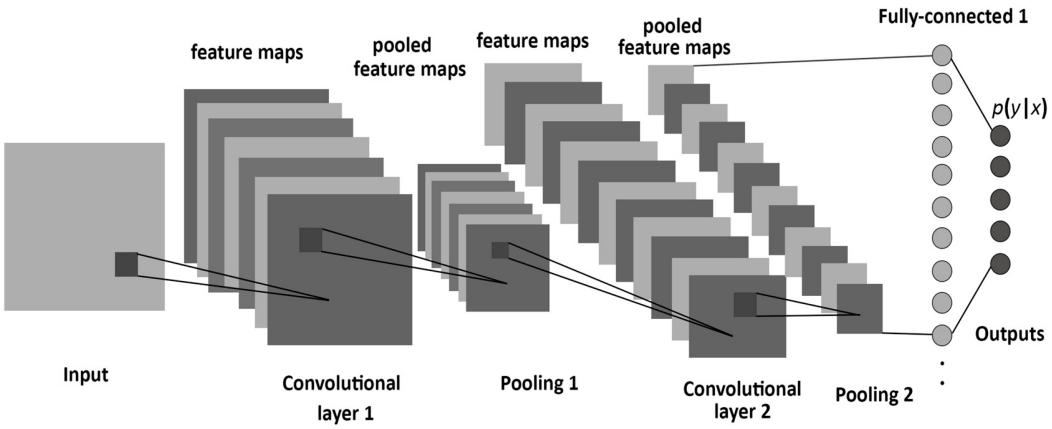

Figure 3.2. Visualization of CNN architecture [16].

3.3.2 Convolutional Layer

The input matrix, representing the input image, is passed to a layer that consists of a set of kernels — the convolutional layer. The kernel, or also the filter as an alternative name, is a small matrix of fixed size, and it consists of values that represent pixels of a predefined structure called feature. During the forward pass, each kernel matrix is shifted around the input matrix (Figure 3) and it computes the dot product with the selected area. The result of this computation is an activation map that holds the kernel result for each position in the input. In other words, the process is searching the input image and trying to find known patterns, the same as human does, when it looks on the picture. The convolutional layer outputs a volume with all activation maps stacked in the depth dimension. As an example, having the input image of size 224x224x3 and 32 kernels the layer outputs a volume of size 224x224x32.

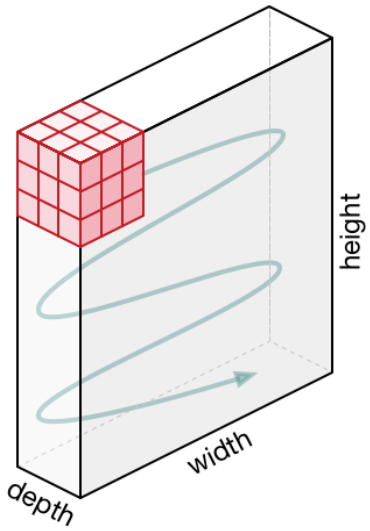

Figure 3.3. The convolution process [17]. Given the three channel image of fixed size, the kernel of size 3x3 is slides (convolves) for each position and image channel and computes the dot product.

3.3.3 Pooling Layer

After the convolutional layer processes the input matrix, the output volume may proceed to another convolutional layer or a pooling layer. The pooling layer takes each item from the input volume and decreases its dimensions by the specified pooling function. Pooling helps to reduce the number of parameters and improve computation performance as it preserves important information and discards irrelevant details. It is the reason why pooling is also called downsampling, as it resizes the input image and makes features lose their detail but be more resistant to small changes [12].

The actual process consists of a matrix of lower dimensions sliding over the input matrix and performing one of the several types of operations — average, maximum, L^2-norm pooling. The most usual operation is a maximum pooling when the matrix selects the maximum value from a rectangular neighborhood at each pooling step.

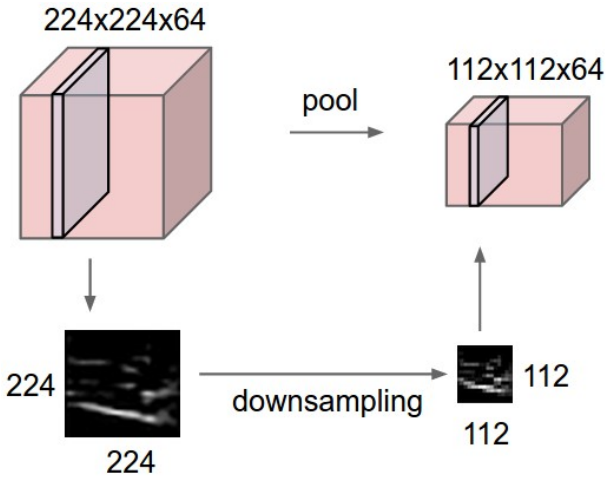

Figure 3.4. The MaxPool Operation.

3.3.4 Fully Connected Layer

Each node from the output of the previous layer connects with the second stage of CNN called a fully connected layer. This layer converts the input data to a single vector It performs the classification, and the principle of computations is the same as for regular ANNs described in Section 3.1.

3.3.5 Object Detectors

One of the stages of the autonomous driving pipeline, as described in Section ??, is an object detection mechanism, where CNNs are applied. The selection of particular detector is always dependant on the trade-off between accuracy and speed [18]. Currently, there are two major categories of detectors: two-stage and one-stage detectors.

Two-stage detectors (Figure 3.5), as the name suggests, are divided into two parts. The first part generates a set of proposals for possible object regions presented in the image. The second part takes created proposals as an input and classifies each region. During benchmarks, this separation leads to state-of-the-art accuracy results, but in terms of the inference speed, they generally fall short.

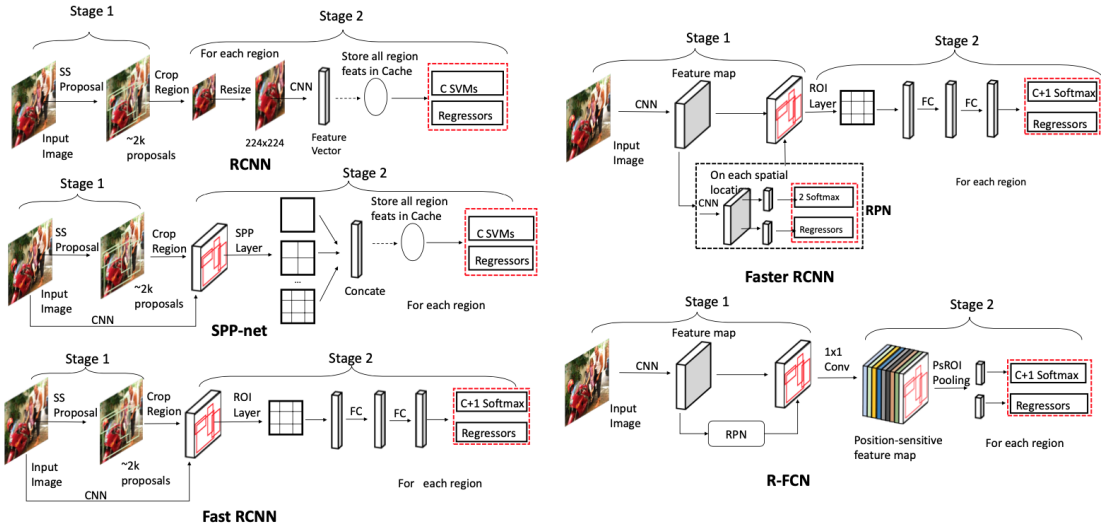

Figure 3.5. Two-stage CNN architectures [19].

The second category, one-stage detectors (Figure 3.6), assumes that an instance of an object could be in any region of the input image, so each region is taken as a proposal. During the classification of proposals, it also estimates the bounding box of each classified object. The lack of the preprocessing stage in the one-stage detectors is an advantage for realtime applications, where the speed matters the most.

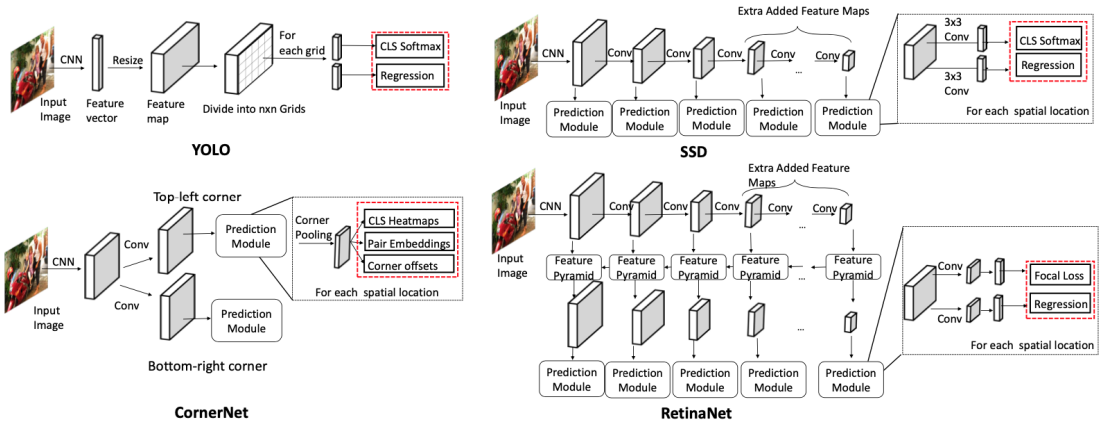

Figure 3.6. One-stage CNN architectures [19].

Recent advances in object detection [20] shows the evolving process of improving modern object detectors and their backbones (networks used for feature extraction). For the fast and low power detection, researches are constantly optimizing detector algorithms to minimize resource demanding tasks and increase accuracy. It was shown [21] that the speedup is achieved by introducing more lightweight network architecture. The following points present networks most relevant to a realtime processing:

- YOLOv3 — predicts class scores and bounding boxes within a single evaluation, which makes it fast for realtime processing. Network is originally built on top of Darknet-53 (Figure C.2) backbone.
- CornerNet-Squeeze — offers a new approach by predicting top-left and bottom-right corners of the bounding box. Network is built on top of Hourglass-54 backbone.
- CenterNet — it uses key-point estimation to find the center of an object and further regress to object properties.
- SqueezeDet — small, fully convolutional neural network, built especially for autonomous driving [22].

3.4 Machine Learning Frameworks

This section presents several machine learning frameworks used nowadays to model, train, and inference neural network models. Besides the implementation details, during the inference, they all receive a model definition, load trained weights, and, depending on the order of operations, execute defined functions by using frameworks available on a particular platform. The model definition is often serializable for easy distribution, TensorFlow[1] uses own SavedModel[2] format and frameworks such as Caffe2[3], PyTorch[4], MXNet use ONNX[5] format.

Frameworks are often able to run on different platforms:

- CPU (x86 and ARM)
- GPU
- FPGA
- ASIC

Different accelerators are available, such as TPU and NPE, that add a more efficient computations of matrix multiplication and other related operations.

Most of the platforms offer custom compilers that are capable of optimizing the training process and model during inference. One of the most suitable, concerning the Jetson platform, is NVIDIA, which provides the TensorRT library. TensorRT loads the provided model and applies several optimization steps that are not described in detail to a public audience [23]. It makes it hard to measure and introduce new optimizations from the side of the community. Figure 3.7 visualizes a pipeline of the TensorRT library.

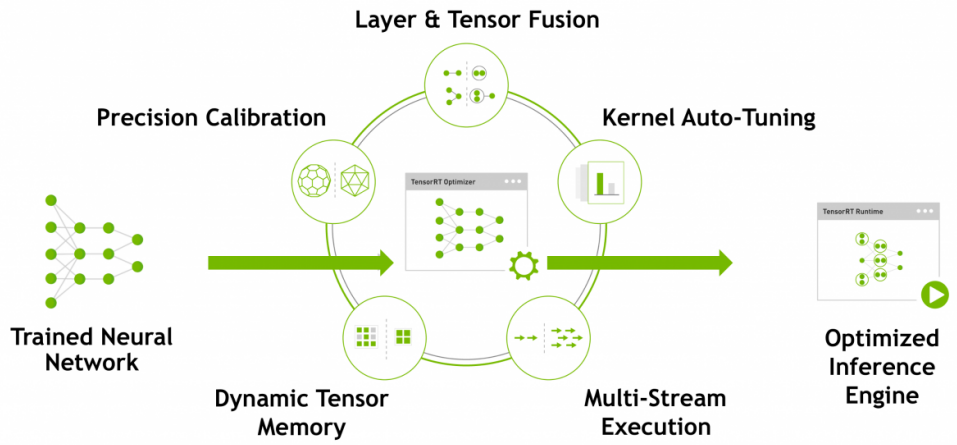

Figure 3.7. NVIDIA TensorRT Pipeline [23].

Mentioned frameworks provide very similar functionality, and every framework has its positive and negative parts. Some of the properties are presented in the following points:

[1] https://www.tensorflow.org
[2] https://www.tensorflow.org/guide/saved_model
[3] https://caffe2.ai
[4] https://pytorch.org
[5] https://mxnet.apache.org

- Caffe2 - lightweight framework that supports CNN's and has optimized inference. Recently the framework was merged into a PyTorch 1.0.
- PyTorch - a production-ready framework developed by Facebook. It uses the Python programming language as its frontend. Large community.
- Darknet[1] - the framework popularized by the YOLO model, where it was implemented. It supports convolutional neural networks and is very similar to Caffe. The popular fork[2] is used nowadays as it is improving the detection speed, and the repository is still supported.
- TensorFlow - currently, an extensive framework for machine learning in general. It has a large community, extensive documentation, and several support tools such as TensorBoard and TensorFlow Serving.

[1] https://pjreddie.com/darknet
[2] https://github.com/AlexeyAB/darknet

Chapter 4
TensorFlow

TensorFlow is a popular machine learning framework developed by Google for its internal purposes and subsequently open-sourced to the community. The framework is provided as a public GitHub repository. It provides tools for training machine learning models and includes several features used for an efficient prediction. It also has support for distributed computation. During the past years, there was only one major version that had several drawbacks observed by the community, most importantly, a hardness to use. In October 2019, long-awaited version 2.0 of the framework is reaching its stable state and adds new components for effective development. In the time of writing this thesis, version 2.1 is in release candidate branch, providing the unification of the codebase for CPU and GPU code, which is now used to be separated into two builds. This section aims to provide a detailed explanation of the TensorFlow internal functionality, in its current stable version — version 2.0.

4.1 Overview

TensorFlow framework consists of two major parts — frontend and backend. The frontend part is a language-specific wrapper that is provided to the developer (e.g., Python, Java, C++, etc.) for building dataflow graphs and their execution. The backend part is a C++ codebase, that handles the execution logic and provides the connector for mentioned frontends. The backend engine uses accelerators to compute operations efficiently. Figure 4.1 visualizes these components.

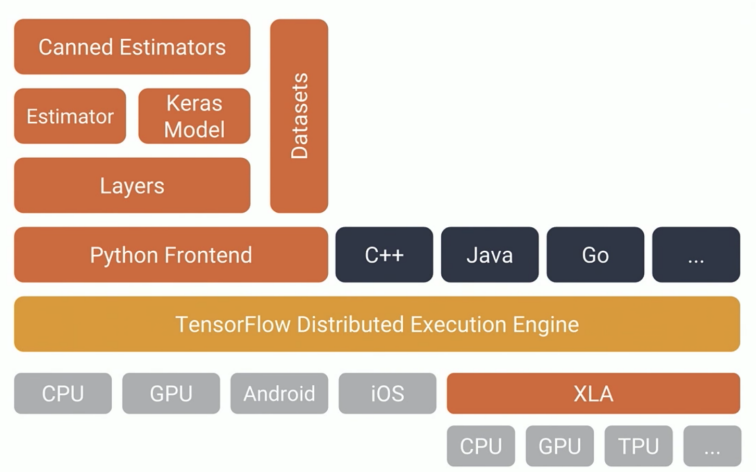

Figure 4.1. Tensorflow architecture [24]. The top three levels of the scheme visualizes components used for training. TensorFlow backend is a multiplatform execution engine written in C++. It handles the framework logic and provides the client API to language-specific frontends (written in Python, Java, Go, etc.). Available accelerators, depending on the target platform, are handling the execution of operations from the dataflow graph provided by the execution engine. XLA framework is a level of abstraction that uses heuristics to distribute work to the underlying accelerators, such as CPU, GPU, etc.

TensorFlow is based on several main principles introduced in this section and explained in more detail in the following sections. The computation process is starting at the dataflow graph and client process. The graph consists of nodes, representing operations, and edges, representing the data flow. Edges are immutable multidimensional arrays of predefined types (int32, string or a custom shape), called Tensors. The client process executes the provided dataflow graph using the available accelerators (CPU, GPU, TPU). The aim of the execution process is to get the result in the most efficient way. The execution process is handled by an object called Session. It registers available devices, optimizes the graph in several passes and divides the graph into several subgraphs, to parallelize an assigned work. Each graph definition that TensorFlow requires as an input, need to include its structure, signature of the graph input and output, and also weights. These items are serializable into SavedModel format, defined by the protocol buffers schema[1]. It allows the easy distribution of the trained model. SavedModel format is used primarily for the TensorFlow Serving framework, which Section 5 introduces and also for the visualization process in TensorBoard introduced in Section 6.1.3.

By skipping the dataflow graph building and training procedures, as it is not the aim of the thesis, it's necessary to analyze the internal components triggered during the dataflow graph evaluation process. Figure 4.2 depicts each element involved in this process, and the following sections provide such insight.

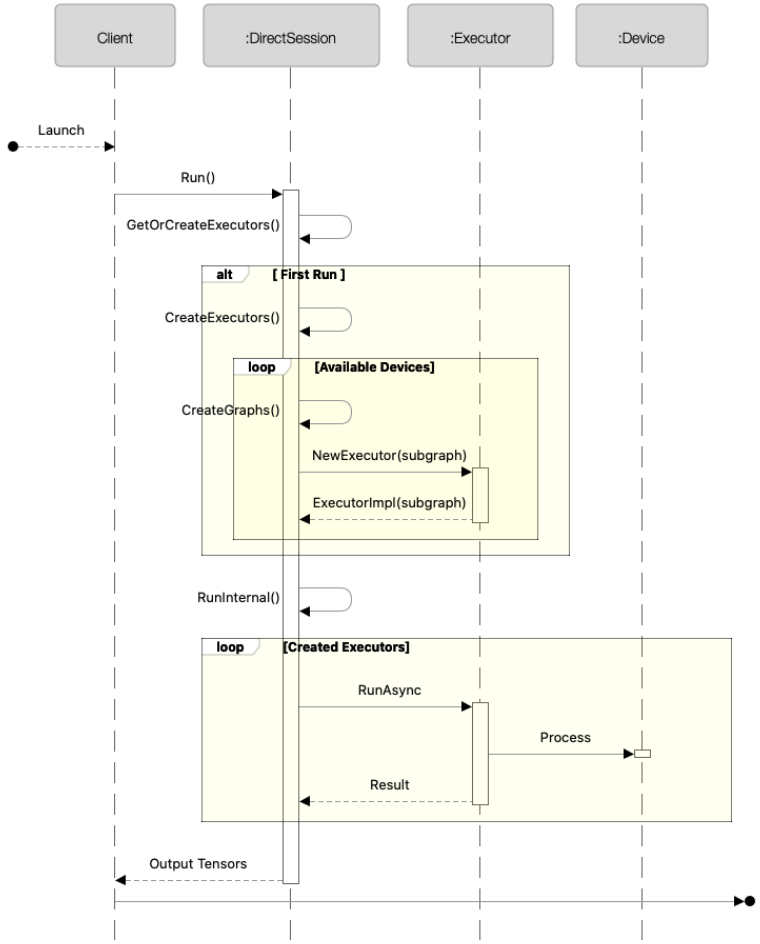

Figure 4.2. TensorFlow Session Run Overview.

[1] https://developers.google.com/protocol-buffers/docs/overview

4.2 Device Manager

The device manager manages available computational resources. Devices in TensorFlow are identified as follows:

- `/device:CPU:0` — The CPU unit of the current machine.
- `/device:GPU:0` — The first GPU visible to TensorFlow.

Additional resources of the same type increments the number on the end of the identification string. GPU units are detected from the CUDA environment variable CUDA_VISIBLE_DEVICES.

During the initialization of the GPU device, TensorFlow allocates almost all available GPU memory to reduce memory fragmentation. This behavior could be modified by creating a new virtual GPU and specifying the memory_limit parameter to a desired value. The CPU device is always registered as a client device, meaning the device from which the computation starts. It prepares and passes the input tensors for the graph and receives the output tensors as a result.

Single operation in TensorFlow may have an implementation for CPU and also for the GPU. In this case, the implicit placement is on the GPU; otherwise, it should be mapped explicitly to the CPU[1].

4.3 Thread Pool

TensorFlow is capable of using multiple CPU threads to parallelize assigned tasks. By default, it picks an appropriate configuration depending on the available resources. Several configuration options could modify the default behavior, and it's verified that such a modification can improve the serving time [25]. It's necessary to highlight a part of configuration, described in more detail in the source code of ConfigProto[2]:

- intra_op_parallelism_threads — specifies how many threads from the thread pool are available for use during the task parallelization. It controls the maximum parallel speedup of a single task.
- inter_op_parallelism_threads — specifies how many threads from the thread pool are available for computation of independent tasks, meaning their execution order. Functions that are executed within this thread pool and are capable of parallelization, share threads from this thread pool. Passing the negative value limits the execution only to the caller thread.
- session_inter_op_thread_pool — specifies the thread pool to use during the execution. Depending on the situation, it may be useful for executing high-priority tasks in a larger thread pool. Low-priority tasks may use a smaller thread pool and use lower intra-op parallelism. This configuration is especially useful during the model serving.

After the thread pool creation, processes are allowed to pass functions for the execution, by submitting them via function ThreadPool::Schedule.

4.4 Graph

The core component of the TensorFlow is a dataflow graph. The dataflow graph is a type of DAG, which represents the computation and dependencies between each operation

[1] https://www.tensorflow.org/guide/gpu
[2] tensorflow/core/protobuf/config.proto

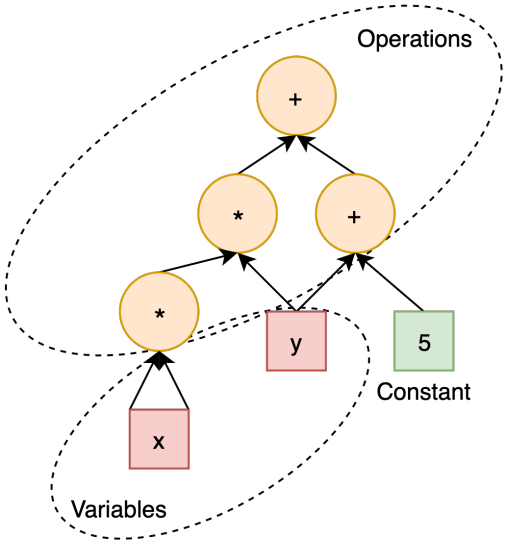

Figure 4.3. Schematic example of the TensorFlow computational graph. The graph represents a function $f(x, y) = x^2y + y + 5$.

defined as a node. The graph is required to have a source and sink nodes, to be able to plan the execution and make further optimizations on a graph[1].

4.5 Session

Session is a major TensorFlow component that encapsulates the evaluation of the dataflow graph by using accelerators available from the device manager.

There are several implementation types of the Session[2]: distributed and common. The distributed type is represented by the instance of the MasterSession[3] and WorkerSession[4]. It organizes the cluster of machines by the Master node and uses other machines as Worker nodes. The master device distributes tasks queued for the computation to workers. It is advantageous in the environments where the hardware limitation is negligible, or the training process demands powerful resources to complete the computation. The common type is represented by the instance of the DirectSession[5]. It operates on the devices, available locally through the device manager, by partitioning the dataflow graph between them. The device manager is introduced in Section 4.2.

Each instance of a session allows its custom configuration via the protocol buffer message ConfigProto[6]. This configuration sets important parameters connected with the computation phase, such as settings for the thread pools, introduced in Section 4.3.

Session::Run is an entry point to the session evaluation. It creates executors, introduced in Section 4.8, and caches them by the signature of their graph inputs and outputs. After, session setups a call frame, described in Section 4.9 and triggers the

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/graph/graph.h
[2] tensorflow/core/public/session.h
[3] tensorflow/core/distributed_runtime/master_session.h
[4] tensorflow/core/distributed_runtime/worker_session.h
[5] tensorflow/core/common_runtime/direct_session.h
[6] tensorflow/core/protobuf/config.proto

execution process by calling a function Session::RunInternal, described in detail in Section 4.10. At the end of a run, the output is retrieved from the call frame and returned to the client process.

Before executors are created, there are few steps that involve dataflow graph pruning and partitioning processes.

During the first run, the TensorFlow session optimizes the dataflow graph by pruning. Pruning is the process of GraphExecutionState::BuildGraph function. It receives the graph (Figure 4.4) with the instance of FunctionLibraryDefinition, representing all its functions, and starts the optimization process.

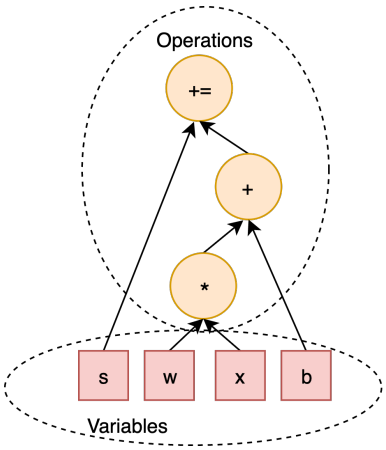

Figure 4.4. Computational Graph Before Partitioning. The graph applies weights w to a feature vector x, adds a bias term b and saves the result in a variable s [26].

The MetaOptimizer of the Grappler component handles optimization. The optimization steps of MetaOptimizer are described in Section 4.6. These steps may add or remove functions of a graph as it includes common subexpression elimination and constant propagation. In a situation when the optimization is not possible, the graph and the function library are restored to the initial state. The library definition is updated accordingly to graph changes.

After the pruning process, partitioning is taking place. The partitioning process divides operations that are capable of running on the available devices and creates one subgraph per each device (Figure 4.5).

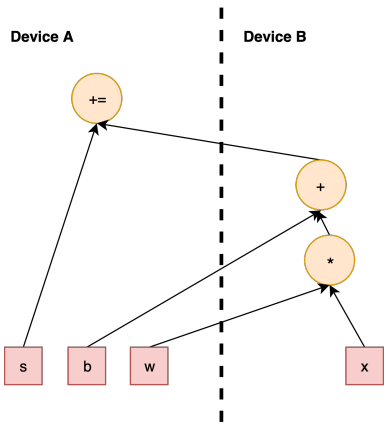

Figure 4.5. Computational Graph After Partitioning. The variable x is copied to the Device B, including some operations, that are optimized on a given device.

Due to this separation between the devices, some edges between the nodes in the original graph may disappear. To solve this, Send/Recv functions are added to a graph for each node operation that it requires (Figure 4.6), and FunctionLibraryDefinition is updated with these new operations. The unique key, named rendezvous key, handles the pairing between these two functions. The described partitioning step is illustrated in Figure 4.6. As the subgraph now has several Recv nodes, they behave the same as root nodes.

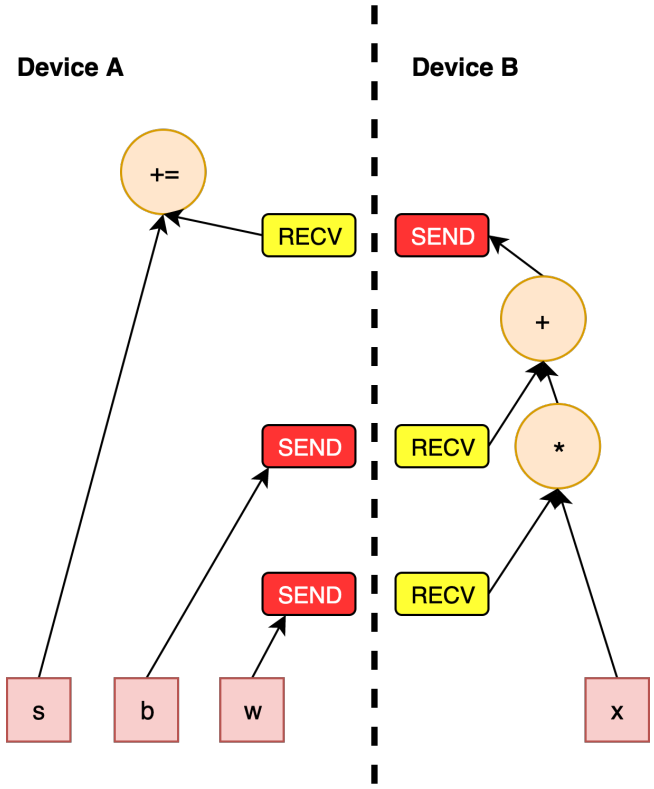

Figure 4.6. Computational Graph Partitioning Result. Colored components are created as value-copy operations between the separated subgraphs.

Each device has an opportunity to optionally modify an assigned subgraph by implementing Device::MaybeRewriteGraph function. After the described optimizations, the created graph is cached for future accesses to add speedup during the future prediction process. Each device received the subgraph is now allowed to execute it by using the local device executor from Section 4.8.

4.6 Grappler

Grappler is an optimization library that performs several optimization techniques on the provided computational graph (Figure 4.7). It takes the original Graph, converts it to the GraphDef[1], which now contains a serialized computational graph and creates a mutable instance called GrapplerItem. Within this item, Grappler performs arithmetic, dependency, function, loop, and memory optimizations. It also provides a function to register custom optimizations, that implements the CustomGraphOptimizer interface. The tensorflow/core/grappler/optimizers directory holds all pre-defined optimization stages. After the particular optimization, the library counts the number of nodes and edges, added or removed during the process. It allows Grappler to measure the outcome of each optimization stage.

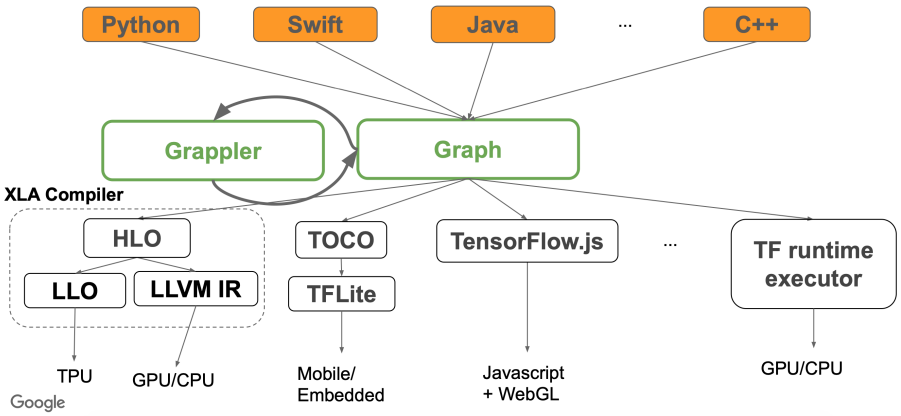

Figure 4.7. Position of the Grappler inside the TensorFlow Architecture [27].

Grappler uses several cost metrics to predict the behavior of future computations. As an example, the memory metric estimates the worst case of memory efficiency for the provided graph as it knows the hardware capabilities and sizes of the inputs and outputs of each layer of the model. It's worth to mention that by enabling logging to level 1 and setting TF_DUMP_GRAPH_PREFIX environment variable to a specific directory saves the GraphDef after each optimization stage to a file. It might be useful for visualizing final operations planned to execute on the particular devices, during the development process.

4.7 Kernel Op

The OpKernel[2] defines an operation that is optimized to run on a particular device. Each kernel defines a function Compute(); multiple executors further execute that. The kernel execution must be thread-safe as multiple executions are allowed to execute multiple instances of the same operation asynchronously (end of the Section4.8).

The usage of cuDNN library is preferred as it can provide an efficient implementation of selected kernels.

[1] tensorflow/core/framework/graph.proto
[2] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_kernel.h

4.8 Executor

Section 4.5 describes how Session::Run process creates particular executors and computes the dataflow graph, by involving available devices. This section aims to follow the idea to describe the creation process of such executor and how it handles the computation.

During the session execution process, the default TensorFlow executor[1] is created and intialized by a call to a function Executor::NewLocalExecutor.

The initialization process consists of the conversion of the input graph (the subgraph of the dataflow graph) to a version optimized for the execution. An optimized graph is an immutable copy of the original graph that includes the memory alignments of the graph nodes.

After the graph conversion, the graph is iterated by BFS traversal and builds frames by detecting enter and exit operators of the control flow, described in [28]. Next, the process traverses the graph once more time and creates a OpKernel (Section 4.7) for each node. During the iteration, it counts the number of inputs for each node and detects the type of the control flow operator.

After the initialization, the executor is ready to process the graph and awaits for the call of its RunAsync function. Also, it provides a Run function, to execute in a synchronous manner. These functions are built on the principle double dispatch and they pass their instance by ExecutorState::RunAsync function.

ExecutorState::RunAsync function, fills a context map for the provided graph depending on the device the executor is created for. The context map just stores the device context to each graph node. Usually, this function does nothing. For example, the implementation of the function BaseGPUDevice::FillContextMap holds some code, but in case the maximum amount of streams is one (hardcoded default value), it just returns. The reason for this is, that the TensorFlow team initially planned to use multiple streams for the execution, but this idea is considered as unmaintained and it is already removed in version 2.1 RC[2].

Next, the ExecutorState::RunAsync function creates a queue called Ready. Ready queue consists of nodes of type TaggedNode, that holds a reference to a node, its frame, and information if it's `dead`. The `dead` state is used for distributed computations to indicate empty tensor which takes an inactive branch in the conditional statement. At first, root nodes are added to the queue and scheduled to run within the thread pool, by calling ExecutorState::ScheduleReady.

The function ExecutorState::ScheduleReady uses runner, created previously during the thread pool creation in Section 4.3. The runner uses the provided thread pool and calls a function ExecutorState::Process for each root node added to the ready queue previously. The called function ExecutorState::Process creates another queue and adds there a node from the argument. Each node receives its input and call frame, holding its context. After it, the process is starting the computation of each node from the queue.

Depending on the definition of kernel, if it's synchronous or asynchronous, context for computations is created. For asynchronous its an ExecutorState::AsyncState object and for synchronous its an OpKernelContext object. Every operation could identify itself as an expensive. The synchronous execution traces the cost of such operations and stores the number of cycles needed for getting the result.

[1] tensorflow/core/common_runtime/executor.cc
[2] https://github.com/tensorflow/tensorflow/commit/223c8bdf8963fef00cd9a1ec0fa10a3fb47fda1e

Next, the created context and node are passed to the device function Compute or ComputeAsync respectively. As the implementation of these functions is dependant on the underlying device (CPU, GPU, CUDA, XLA, or SYCL), the details are separated by appropriate sections. CUDA device is just a one-to-one wrapper over the GPU device. XLA and SYCL devices are out of the scope of this thesis.

4.8.1 CPU Compute

Computation of the kernel operation on the CPU is handled by the immediate execution or by queueing on a thread pool (thread pool and runner configuration in Section 4.10). Edges in a graph define the dependency between these nodes (operations). It means that given an edge from node i to node j, the execution of a node j will wait until node i finishes its execution. The CPU version calls a Device::ComputeAsync or Device::Compute respectively, and the call is passed to the computational function of the kernel operation.

4.8.2 GPU Compute

Computation of operations on a GPU is handled by making the requests to the StreamExecutor[1], which is an abstract wrapper over the underlying platform (CUDA, OpenCL). TensorFlow uses a single stream for computation operations on the GPU device. Multi-stream code is present, but due to the reason described earlier in this section, is now planned to be removed.

There are two functions BaseGPUDevice::ComputeAsync and BaseGPUDevice::Compute, that initiates the execution of the kernel operation. If the kernel operation context is present it is used instead of the device context. It allows the kernel operation to use multiple streams during its execution.

As it was stated earlier in this section, root nodes are initially a part of the ready queue. All these nodes are iterated and scheduled on the device. After their completion and during the postprocessing in ExecutorState::NodeDone, they are removed from the queue and added to inline_ready queue. It means, that they have some output. Next, the successors of inline_ready nodes in the computational graph are added to the ready queue. The further computations iterate in this order, by updating both queues. Frame (Section 4.9) is constantly updated during this process by the definition of control flow operations (Section ??).

4.9 Data Transmission

The FunctionCallFrame is a data structure that is used to pass arguments to functions and retrieve their results. Each value of the argument is represented as a Tensor object. Session uses such a frame to exchange model input with the created executors. After executors finish computing their graphs, the frame is used to read the results and return them to a client.

4.10 Execution

The DirectSession::RunInternal function manages the session execution process. It receives several arguments. The step_id is an identification of the session run (or any API call in general). The executor_and_keys object, holding previously created and assigned

[1] tensorflow/stream_executor/stream_executor_pimpl.h

executors. The call_frame, created to exchange data with executors (Section 4.9). Section 4.11 describes components for the distributed computation, that are involved in the process but are not executed.

Execution is started by creating ExecutorBarrier, which helps to run executors in parallel and ensures that all executors finish their work at some point. The barrier provides information about the executor's status using the mutex lock. In the next step, configurations of the thread pool, graph options, and some general configuration are handled and stored as an instance of Executor::Args. Depending on the setting of inter_op_* configurations, described in Section 4.3, the execution process determines which thread pool to use — device or local. Execution allows to run executor process in the caller thread (and overall run synchronously) only when there is a single executor defined, otherwise, the first thread from the thread pool is selected. Device can specify its own thread pool by a call to function set_tensorflow_device_thread_pool. The example of a device thread pool setting is available in BaseGPUDevice::Init. This function, depending on the configuration, assigns the thread pool for a GPU device.

Next, the execution iterates through available executors and start them asynchronously, providing the reference to the barrier, defined earlier. The barrier manages the completion of all executors and after they reach its point, several processes are initiated. First, the output tensors are fetched from the executors. Second, the cost model of the graph is updated, depending on the frequency, previously set in the configuration (as it may increase the overhead of prediction). Optionally, partitioned graph definitions are outputted.

4.11 Collective Execution

The term collective indicates the usage of distributed computation between several machines. It's used mostly for training, but the distributed prediction is also supported[1]. The difference between the normal graph and distributed graph, that distributed version uses CollectiveOps such as broadcast, all-reduce, etc. The graph is identified by the presence of collective_graph_key.

[1] https://www.tensorflow.org/guide/distributed_training

Chapter 5
TensorFlow Serving

Tensorflow Serving is a framework that provides a reliable solution for inference on trained models. Figure 5.1 visualizes the general structure of the framework. The framework is targeting to the effective management and serving of provided trained models. Models can have several versions that are available to the clients. Clients are incoming requests, carrying the input data. Computations on a graph are part of the TensorFlow servable, which uses the general TensorFlow framework as a library and encapsulates the internal computations. The following sections aim to describe each execution stage in detail, to be able to understand the overall architecture and allow to identify parts of the TensorFlow used during the computation.

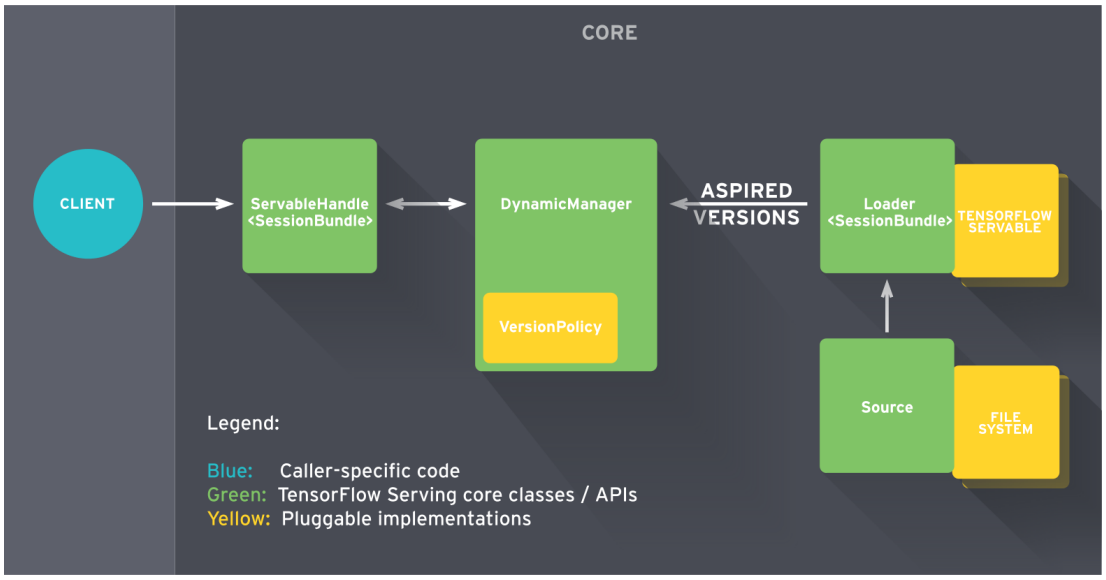

Figure 5.1. TensorFlow Serving architecture[1]. The client is submitting his request to the model server, which can handle multiple versions of the same model. The TensorFlow framework processes calculations on a model available on the filesystem. The result is returned to the client.

5.1 High-Level View

The entry point of the prediction service is an HTTP/gRPC[2] based server. During its start, it parses the provided configuration, reads hardware capabilities of the underlying platform, and creates a protocol buffers message GraphDef from the provided model in SavedModel format, introduced in Section 4.1. Before the serving is available to the client, TensorFlow creates a Session object, introduced in Section 4.5. During the creation process, it optimizes the model by modifying the granularity of operations and

[2] https://github.com/grpc/grpc

their order. The optimization leads to a more efficient usage of the provided resources. After preparation steps, the framework listens to the incoming requests from clients, and when the input data get available, it triggers the session execution. Additionally, the polling mechanism, running in the background, is detecting changes in the model and eventually reload the configuration of the server.

Serving handles the communication with TensorFlow by the C API[1], which TensorFlow framework provides. It is a level of abstraction between internal framework functionality and language-specific wrappers such as Python, Java, C++.

The server is capable of two types of communication, HTTP and gRPC. The second type is faster as it doesn't have a protocol overhead and data serialization/deserialization processes, due to the fact, that HTTP type is calling the gRPC underneath[2]. Custom benchmark for 30 executions and ten repetitions estimated that gRPC version is faster than HTTP by $63\% \pm 4\%$ assuming 95% confidence interval. On the other hand, it requires additional libraries. It may be a restriction in some environments where clients execute.

5.2 Initialization

Server is instantiated in 'model_servers/main.cc'. Hardware capabilities are read by a call to a function tensorflow::port::InitMain[3], which is separated by the OS architecture (e.g., POSIX, Windows, Cloud, etc.) for portability reasons. It stores the information about the available RAM, number of processors, and hardware layout for planning future computations. The server is built on top of the class ServerCore[4]. It is a platform and language independent part of the code that holds the state and the settings of the provided model and manages the efficient serving.

After the server startup, Model[5] and Prediction[6] services are employed. Model service listens to requests regarding the structure and state of the model, where the Prediction service handles input data and initiates the computation process. Before the prediction is taking place, the Grappler, introduced in Section 4.6, optimizes the model graph definition from the input. The following section describes the Prediction service in detail.

5.3 Prediction

Prediction service consists of several servable types — classification, regression, prediction, and multi-inference[7]. Depending on the attribute 'method_name' in the signature of the saved model, an appropriate servable is selected and used for the upcoming computations[8]. Since the base model of this thesis defines the 'prediction' servable type, the following text will stress it in detail.

Prediction is handled by the class TensorflowPredictor[9], registered earlier during the initialization of server core. Entrypoint is the Predict function that uses SavedModel-

[1] tensorflow/c/c_api.h
[2] tensorflow_serving/model_servers/prediction_service_impl.cc
[3] tensorflow/core/platform/init_main.h
[4] model_servers/server_core.h
[5] model_servers/model_service_impl.h
[6] model_servers/prediction_service_impl.h
[7] Directory servables/tensorflow
[8] https://www.tensorflow.org/tfx/serving/signature_defs
[9] servables/tensorflow/predict_impl.h

Bundle[1] to wrap GraphDef and Session objects. After SavedModel instance is created, function RunPredict[2] is called. RunPredict validates metadata and sizes of the input and output tensors. After, the prediction is started by creating and running the instance of a DirectSession[3]. The execution stages of the DirectSession are explained in detail in Section 4.8.

After the completion of the session run, the result is available for the response to the client. TensorFlow Serving awaits the next request from the client and checks the model for changes in periodic intervals.

5.4 Deployment

TensorFlow Serving supports the execution inside Docker container (Figure 5.2). Internally, during the creation of Docker image, Dockerfile of the TensorFlow Serving downloads all required system libraries, installs Bazel[4], and builds TensorFlow Serving from source. It is a preferred way to distribute application into edge environments via Docker Registry[5].

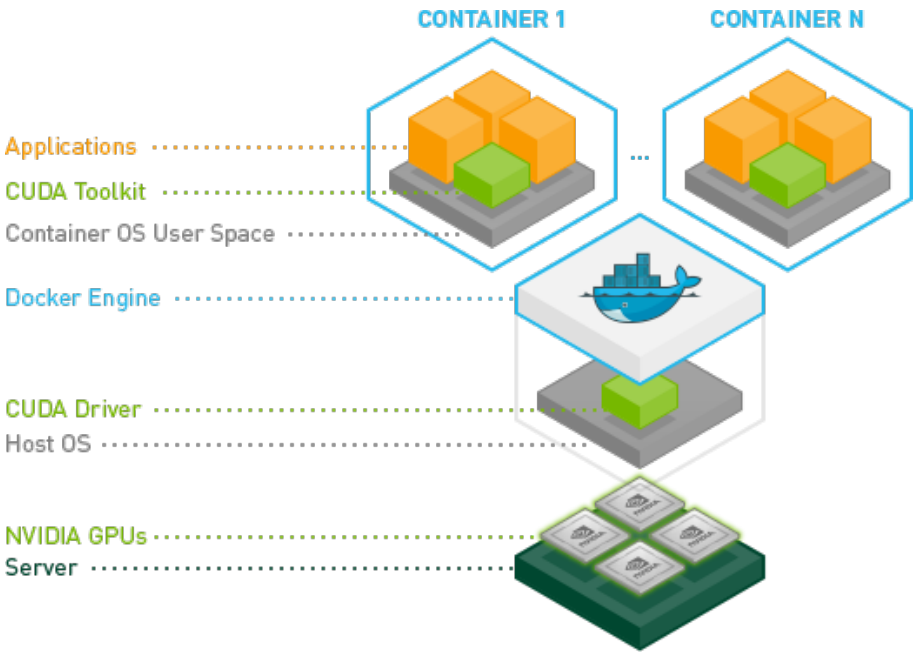

Figure 5.2. TensorFlow Serving Docker Support[6].

[1] tensorflow/cc/saved_model/loader.h
[2] servables/tensorflow/predict_util.h
[3] tensorflow/core/common_runtime/direct_session.h
[4] https://bazel.build
[5] https://docs.docker.com/registry/

As a consequence of this architecture, the application can use Docker layers to cache the image efficiently. During the distribution, it is separated by the layers defined in a Dockerfile, and during the download from the repository, it downloads only changed parts. It improves the deployment process of the devices located in the edge environment.

Chapter 6
Implementation

Section 3.3.5 describes the state-of-the-art CNN called YOLOv3 and its advantages compared to other convolutional neural networks. This section uses the mentioned model for describing the installation and implementation details of the benchmark framework, created for purposes of comparing inference time of TensorFlow models.

6.1 Setup

The target platform for the algorithm evaluation is an NVIDIA Jetson TX2. It is a power-efficient device created especially for embedded AI computations. The Jetson TX2 module (Figure 6.1) drives the quad-core ARM Cortex-A57 MPCore CPU and 256-core NVIDIA Pascal GPU, both sharing the 8GB LPDDR4 RAM[1]. NVIDIA SDK Manager is handling the flashing process of the OS image and NVIDIA libraries included in the appropriate JetPack[2] release.

Hardware feature \ Jetson module	Jetson Nano	Jetson TX1	Jetson TX2/TX2i	Jetson AGX Xavier
CPU (ARM)	4-core ARM A57 @ 1.43 GHz	4-core ARM Cortex A57 @ 1.73GHz	4-core ARM Cortex-A57 @ 2GHz, 2-core Denver2 @ 2GHz	8-core ARM Carmel v.8.2 @ 2.26GHz
GPU	128-core Maxwell @ 921MHz	256-core Maxwell @ 998MHz	256-core Pascal @ 1.3GHz	512-core Volta @ 1.37GHz
Memory	4GB LPDDR4, 25.6 GB/s	4GB LPDDR4, 25.6 GB/s	8GB 128-bit LPDDR4, 58.3 GB/s	16GB 256-bit LPDDR4, 137 GB/s
Storage	MicroSD	16 GB eMMC 5.1	32 GB eMMC 5.1	32 GB eMMC 5.1
Tensor cores	--	--	--	64
Video encoding (NVENC)	(1x) 4Kp30, (2x) 1080p60, (4x) 1080p30	(1x) 4Kp30, (2x) 1080p60, (4x) 1080p30	(1x) 4Kp60, (3x) 4Kp30, (4x) 1080p60, (8x) 1080p30	(4x) 4Kp60, (8x) 4Kp30, (32x) 1080p30
Video decoding (NVDEC)	(1x) 4Kp60, (2x) 4Kp30, (4x) 1080p60, (8x) 1080p30	(1x) 4Kp60, (2x) 4Kp30, (4x) 1080p60, (8x) 1080p30	(2x) 4Kp60, (4x) 4Kp30, (7x) 1080p60	(2x) 8Kp30, (6x) 4Kp60, (12x) 4Kp30
USB	(4x) USB 3.0 + Micro-USB 2.0	(1x) USB 3.0 + (1x) USB 2.0	(1x) USB 3.0 + (1x) USB 2.0	(3x) USB 3.1 + (4x) USB 2.0
PCI-Express lanes	4 lanes PCIe Gen 2	5 lanes PCIe Gen 2	5 lanes PCIe Gen 2	16 lanes PCIe Gen 4
Power	5W / 10W	10W	7.5W / 15W	10W / 15W / 30W

Figure 6.1. NVIDIA Jetson Products [29]. Third column in the table represents the target device.

[1] https://developer.nvidia.com/embedded/jetson-tx2
[2] https://developer.nvidia.com/embedded/jetpack

6.1.1 Bazel

Bazel is a Java-based tool developed by Google for building and testing large codebases. TensorFlow and TensorFlow Serving are using this tool as their primary mechanism for code compilation. At the root of each project, there is a BUILD file containing rules for compiling particular software code and its dependencies. Bazel requires a decent amount of available RAM that, in some environments, is not available, and its shortage may lead to an unexpected crash. The build parameter local_resources is limiting the usage of CPU and GPU resources[1] and overcomes a possible crash. During the build on a device, it is possible to create a cache that Bazel will store to the remote storage, such as Google Cloud Storage[2] or local disk. This cache is useful during the compilation on different devices and during the switch of a code branch. It shares already compiled parts and speeds up the whole process, in situations when the available network bandwidth allows it.

6.1.2 TensorFlow Serving

During the compilation, TensorFlow Serving downloads TensorFlow source code from publicly available GitHub repository[3]. It allows to change commit hash of the downloaded commit, but it does not allow to change the repository origin. To overcome this restriction, it is necessary to modify a repo.bzl file and change the path for the TensorFlow repository from the original origin to a created fork. After it, the compilation process could start.

6.1.3 TensorBoard

To visualize models, previously created in TensorFlow, it might be helpful to use a TensorBoard[4]. It is a tool, written in Python and JavaScript languages, created primarily for visualization and analysis of machine learning tasks. The advantage of using it is that it can visualize the dataflow graph of the TensorFlow model and provide all the details about the particular layers and operations executed inside. Also, it provides extensive functionality for the training process and supports the creation of custom graphs.

To import the TensorFlow model, previously serialized into a SavedModel format, it is necessary to execute script 'import_pb_to_tensorboard.py', that loads the graph definition and outputs its summary by the logging mechanism into a file. The TensorBoard start script awaits the name of the created file, and instantiates a TensorBoard server, that opens as a web page in a system browser.

[1] https://docs.bazel.build/versions/0.21.0/command-line-reference.html
[2] https://cloud.google.com/storage/
[3] https://github.com/tensorflow/tensorflow
[4] https://www.tensorflow.org/tensorboard/get_started

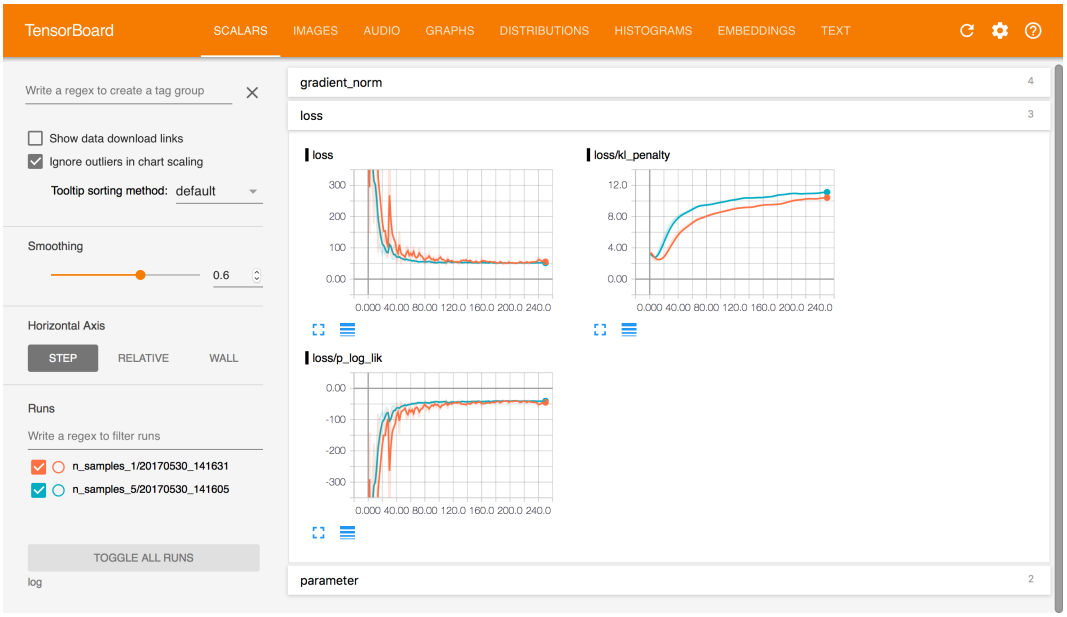

Figure 6.2. TensorBoard during the training process.

6.1.4 Jetson TX2

The Jetson TX2 developer board consists of an Ubuntu 18.04 OS and preinstalled developer tools - CUDA, cuDNN, OpenCV, Docker, and several others. To use the most up-to-date system for this platform, NVIDIA offers the JetPack SDK manager[1], that can flash the board with a new kernel and update the preinstalled libraries to their newer versions. To fulfill the minimum requirements of CUDA version 10.0 for the TensorFlow 2 framework, the JetPack in version 4.2.2 has to be installed. After the successful setup, it is necessary to turn off the GUI of the OS to reduce the amount of allocated memory.

A kernel of the TX2 provides several modes of the performance acceleration, by running the command 'nvpmodel'. Except for the CPU 0, which runs the OS, it can modify the frequencies of particular cores to achieve the energy consumption requirements. The Table 6.1 lists available modes, where the lower mode number means higher performance and higher power consumption.

Mode	Denver 2	Freq.	ARM A57	Freq.	GPU Freq.
0	2	2.0 GHz	4	2.0 GHz	1.30 Ghz
1	0	4	1.2 Ghz	0.85 Ghz	
2	2	1.4 GHz	4	1.4 GHz	1.12 Ghz
3	0	4	2.0 GHz	1.12 Ghz	
4	1	2.0 GHz	1	2.0 GHz	1.12 Ghz

Table 6.1. NVP Models.

[1] https://developer.nvidia.com/embedded/jetpack

6.2 Dataflow Graph

Section 4.4 introduces a component named as the dataflow graph, used for the representation of computations in the network. This section aims to visualize this graph and its different variants during the optimization and partitioning stages (Section 4.5). The visualization is handled by the TensorBoard (Section 6.1.3) framework as it is an official visualizer for the TensorFlow models.

6.2.1 Initial State

To generate the graph visualization, it is necessary to load the file containing a saved model to the TensorFlow and generate a summary log. The summary log contains information about the graph, including the structure of its nodes that TensorBoard uses during its startup and further run. Usually, the script 'import_pb_to_tensorboard.py' is used, but in version 2.0 of the TensorFlow, it does not work, and it throws the 'DecodeError' exception. To solve this, the file is modified to use function 'saved_model_utils.get_meta_graph_def' instead of 'graph_pb2.GraphDef().ParseFromString' to properly load the graph definition from a model file. With the introduced change, the script can generate a summary log to a provided destination, and TensorBoard is now able to parse it.

After the TensorBoard start, the provided URL contains a view of the graph. Unfortunately, it can visualize the graph but without the edges. It seems like an internal incompatibility between the TensorFlow and TensorBoard versions. As there is no simple solution to this problem, the assumption that operations in the graph are ordered from the bottom to the top could partially solve the issue. The Figure 6.3 shows a part of the generated graph of the YOLOv3 model with the DarkNet-53 backbone.

6.2.2 Optimization Stages

Section 4.5 describes the process of input graph optimizations before the start of the graph execution. After each optimization stage, TensorFlow can output a file in a SavedModel format into the provided path via environment variable 'TF_DUMP_GRAPH_PREFIX'. This file could be further processed by an import script 'import_pb_to_tensorboard.py' and visualized in the TensorBoard. Unfortunately, the file that TensorFlow saves an optimized graph in a text format with extension '.pbtxt'. To solve this, the script 'import_pbtxt_to_tensorboard.py' is created and modified, to address this issue and parse it correctly. The process of visualizing is the same as in the previous section.

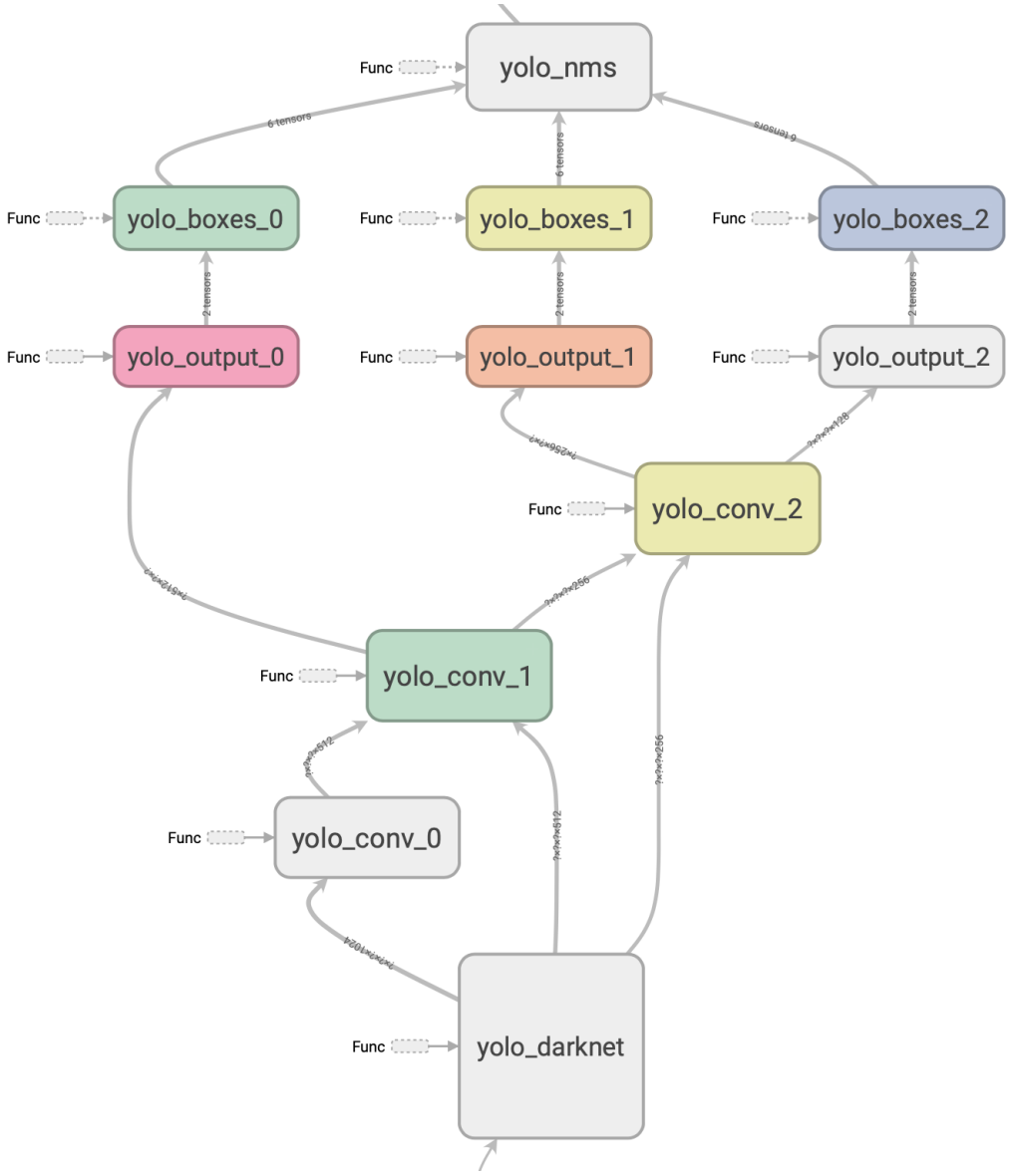

Figure 6.3. Visualization of YOLOv3 (DarkNet-53) architecture. Graph consists of four major parts. Lowest node 'yolo_darknet' is a backbone network DarkNet-53 used for feature extraction and upper levels are feature maps used for bounding box prediction of small, medium and large objects.

6.3 Benchmark Framework

During the network inference, many factors could affect the overall performance. It might be dependant on the selected network architecture, machine learning framework, algorithm for computation, hardware specification, and also on processes running in the machine OS. Also, after the introduced modification in at least one component from the previous list, it might be useful to measure the change in performance. To meet this requirement, the following sections describe the benchmark framework that is capable of running on the developer machine and also on the NVIDIA Tegra X2 platform. Especially Tegra X2 platform is a valuable indicator, as it is a SoC platform where memory collisions and inefficient heat management could significantly decrease the performance.

6.3.1 Input Data

To properly set up the benchmark, it is necessary to define how fast new data will arrive at the object detection neural network. Since, during autonomous driving, onboard cameras are regularly sending image data to the object detection endpoint, the benchmark cannot measure only the execution time of a particular frame, but it also needs to include the time between detection request and start of the execution. It means that each frame is processed asynchronously, with the rate dependant on the FPS settings of the hypothetical onboard camera.

By using video files instead of live video from the camera during the benchmark process, the benchmark can return a reproducible results. The Table 6.2 represents sample videos used during the benchmark.

Filename	Resolution	FPS	Duration (s)
video-1.mp4	640x360	30	30
video-2.mp4	1280x720	24	20
video-3.mp4	960x540	25	20

Table 6.2. Sample videos used during the benchmark.

6.3.2 Total Execution Time

The execution time of the prediction is measured as a time difference between prediction request and prediction response. Each measurement is stored in the 'State' component to be further processed by a histogram.

The overall histogram is generated by the library HdrHistogram[1] and shows the latency by the percentile scores. As it was stated in [8], to meet the performance predictability, the most important indicator is the 99.99th-percentile latency. The output provides this information as well as mean latency.

Before the execution of the benchmark, the number of excluded detections, a warmup period, is estimated manually (Figure 6.4).

Figure 6.5 shows the difference between 31 and 46 frame in warmup state during the benchmark of the video

After the estimation of the warmup period, values prior this number are always excluded.

Figure 6.7 represents the result of the benchmark with 30 executions and 10 repetitions that compares Darknet, TensorFlow 1.x, TensorFlow 2.x and TensorFlow Serving frameworks on YOLOv3 (DarkNet-53) model. The mean of execution time is estimated within the 95% confidence interval [30].

[1] http://hdrhistogram.org/

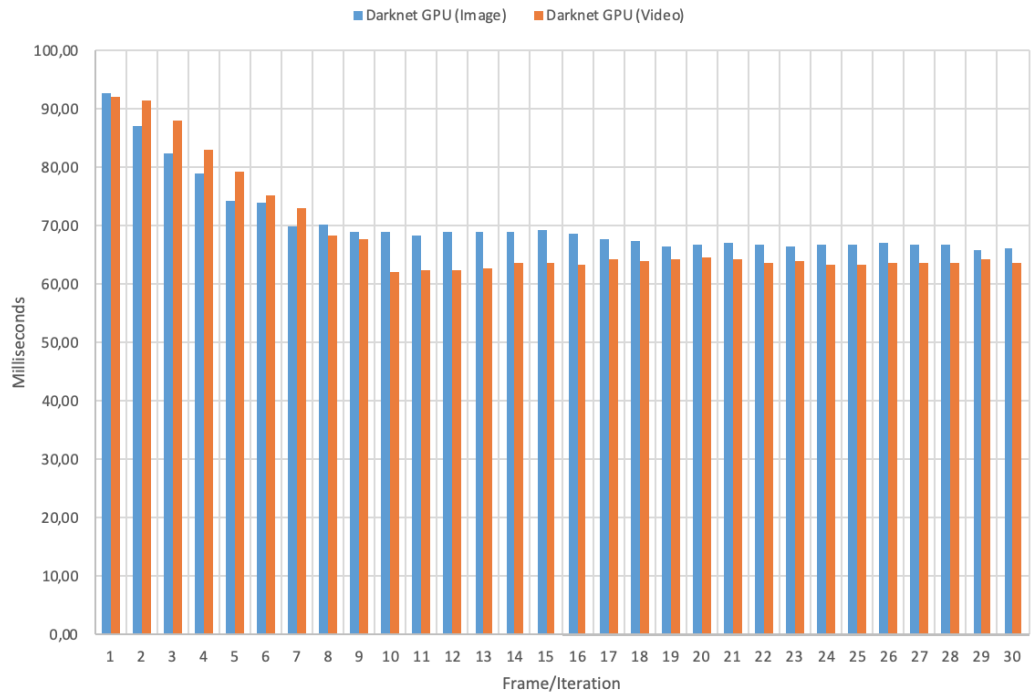

Figure 6.4. Estimation of Execution Independent State. Benchmark of the Darknet framework executed on a short video and image to estimate the warmup period. The estimated period of the warmup is 16 iterations.

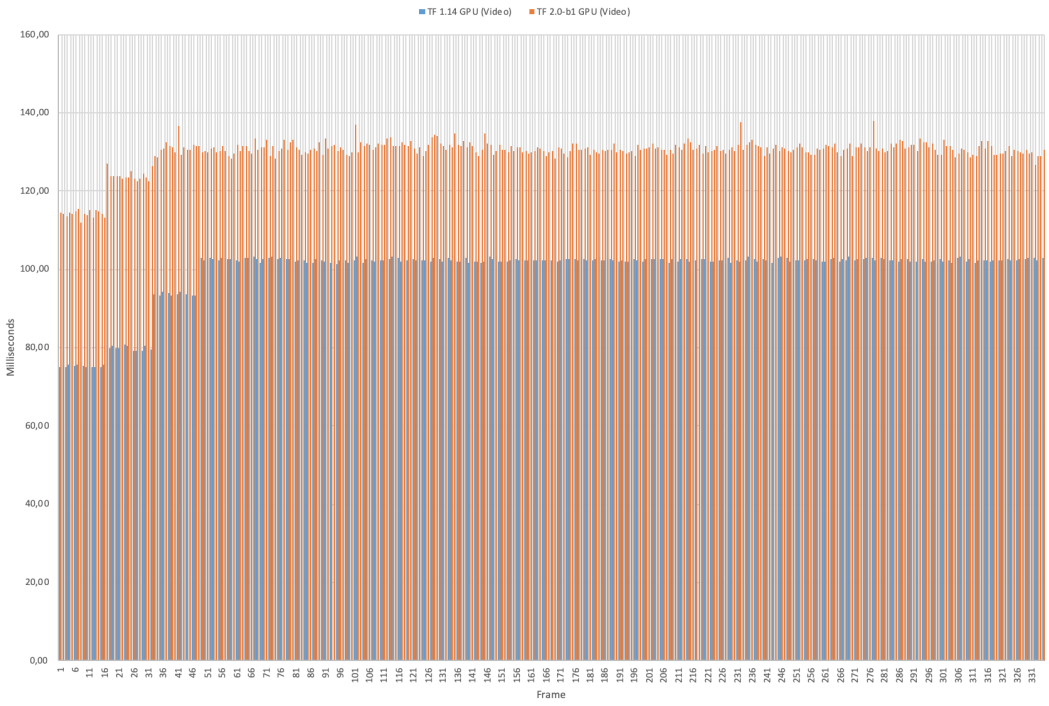

Figure 6.5. Estimation of Execution Independent State for Video. Comparison between TF 1.14 and TF 2.0-b1 built with GPU support.

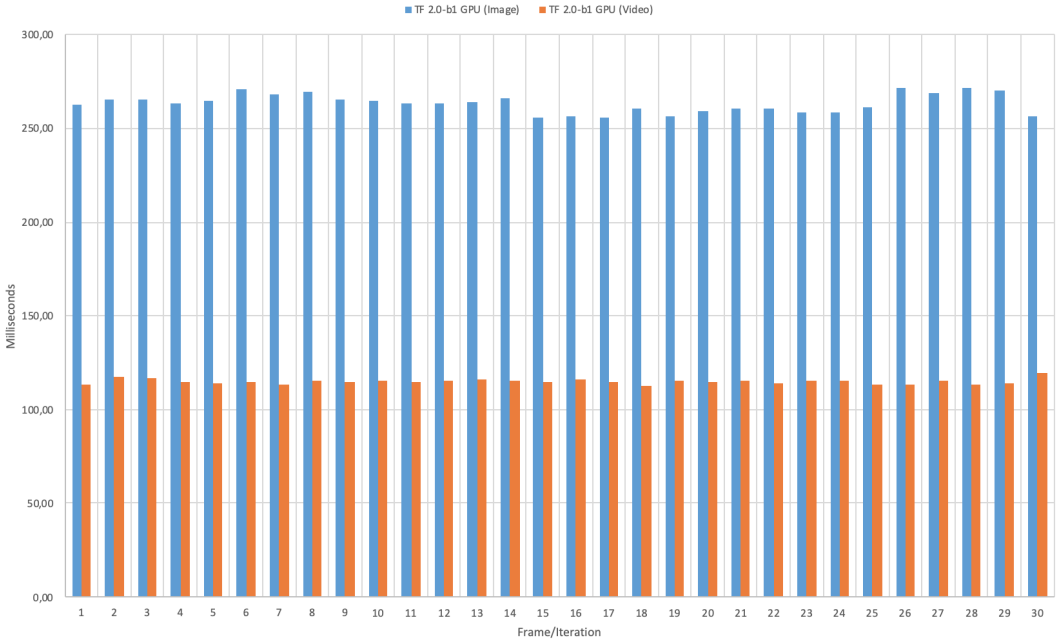

Figure 6.6. Estimation of Execution Independent State for Video and Image for TensorFlow 2. The difference in the inference times presents the difference between creating a new Session or reusing the previous one. Video frames are reusing, where image is always creating a new one.

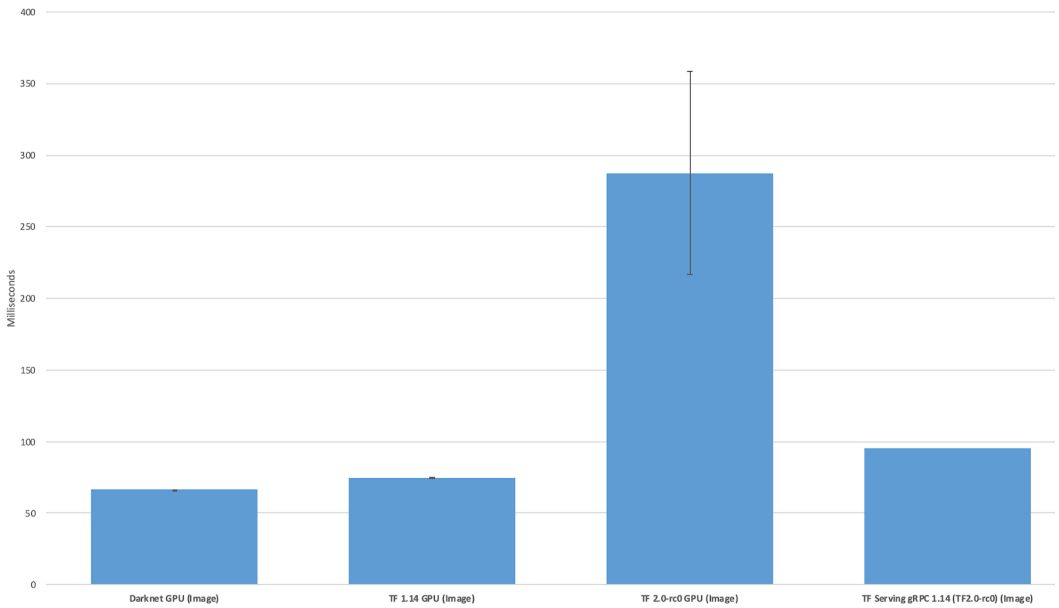

Figure 6.7. Comparison of Darknet, TensorFlow 1.x, TensorFlow 2.x and TensorFlow Serving. The Darknet framework achieves the best performance comparing to the others. TF 1.14 and TF Serving with TF 2.0-rc0 are slower due to the more general architecture. TF 2.0-rc0 has a visible error rate, that is caused by separate execution of the backbone and classifier stages. By executing the model as a static graph, the result is close to the TensorFlow Serving.

6.3.3 Execution-Time Jitter

The execution-time jitter is the largest difference between the execution times of any job within the specified task. The analysis of execution-time jitter is valuable for real-time control systems to determine the variations in sampling-actuation delays of control tasks [31].

Measuring is handled by saving each record to the State object. Each recorded value is stored either to minimum or maximum variable and in the end the difference is returned as a result.

6.3.4 Hardware Statistics

As the target platform for the benchmark is an NVIDIA Tegra X2, it seems as the most optimal way to use 'tegrastats'[1] command available on all NVIDIA Tegra series devices. It allows to fetch the information about each hardware component available on board. Example output for maximum performance mode with the temperature of the environment 23 °C:

```
RAM 329/7860MB (lfb 1744x4MB)
SWAP 0/3930MB (cached 0MB)
CPU [0\%@2035,0\%@2035,0\%@2035,0\%@2035,0\%@2035,0\%@2035]
EMC_FREQ 0\%
GR3D_FREQ 0\%
PLL@31C
MCPU@31C
PMIC@100C
Tboard@26C
GPU@29.5C
BCPU@31C
thermal@30.1C
Tdiode@27.25C
VDD_SYS_GPU 459/459
VDD_SYS_SOC 765/765
VDD_4V0_WIFI 0/0
VDD_IN 3444/3445
VDD_SYS_CPU 229/229
VDD_SYS_DDR 1113/1112
```

For the experimental measurements, the benchmark reads the following values for temparature:

- MCPU@xC — x is a temperature in degrees Celsius of the CPU chip
- BCPU@xC — x is a temperature in degrees Celsius of the CPU board
- GPU@xC — x is a temperature in degrees Celsius of the GPU chip
- Tboard@xC — x is a temperature in degrees Celsius of the board

[1] https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%2520Linux%2520Driver%2520Packag

And for the power consumptions, the benchmark reads the following values:

- VDD_SYS_CPU x/y — x (current) / y (average) power consumption in milliwatts of CPU chip
- VDD_SYS_GPU x/y — x (current) / y (average) power consumption in milliwatts of GPU chip
- VDD_SYS_DDR x/y — x (current) / y (average) power consumption in milliwatts of DRAM

Chapter 7
Conclusion

During the last decade, the advance in the hardware design and new software capabilities improved the embedded hardware; nowadays, used for resource-consuming tasks such as image processing or neural network inference within the target environments. These environments are mostly called `edge`. Many low-powered SoC platforms (Google Coral[1], NVIDIA Jetson[2], Ambarella[3], and others) provide an option to execute trained models using popular machine learning frameworks such as PyTorch[4], Caffe[5], TensorFlow (Lite[6]) and other widely used titles. Before the execution, the network models are mostly pruned and quantized to perform better on the hardware significantly slower than used for the training of such networks. Many mobile backbone networks exist, and their simple architecture minimizes the number of computations required to get the result. As a downside, the accuracy of the detection is lower compared to more massive network architectures. The field for optimization, on the mentioned platforms, is very extensive and brings researchers to new ideas to find an improvement in a ratio of speed and accuracy.

Section 3 introduced a state-of-the-art approach of applying convolutional neural networks in a field of object detection. The selected networks were inferred by several frameworks to estimate the performance difference (Section 3.4). As a result, a framework for the initial execution analysis, a popular fork[7] of the Darknet framework[8] was selected. Additional logging was introduced to get closer to the principle of its functioning, but unfortunately, it was not selected as a target framework for possible scheduling optimizations, as the code is heavily aimed at performance that reduces its readability.

As an alternative, open-source framework TensorFlow was selected. At the time of the analysis, there was already available a release candidate for the next major version, that was preferred, as it might introduce significant changes influencing internal mechanisms. Section 4 described these mechanisms in order to understand the underlying architecture and introduce modifications in code. Sequence diagrams were created to visualize the connection between processes, to support the readability of the description.

Section 6.2 introduced different variations of the dataflow graph (Section 4.4) that represents computations during network inference. It was observed that the initial graph is transformed several times, and each step can be visualized by the TensorBoard framework (Section 6.1.3).

TensorFlow provides an additional library TensorFlow Serving (Section 5) that was selected as a mechanism to provide API for the benchmark, introduced in Section 6.3.

[1] https://coral.ai/products/dev-board/
[2] https://developer.nvidia.com/embedded-computing
[3] https://www.ambarella.com/technology/
[4] https://pytorch.org/get-started/locally/
[5] https://caffe.berkeleyvision.org
[6] https://www.tensorflow.org/lite
[7] https://github.com/AlexeyAB/darknet
[8] https://pjreddie.com/darknet/

According to Kalibera et al., 2013 [30] independent state was found after the first iteration, when optimizations on the model graph are being executed. The mean of the execution time was estimated with the 95% confidence interval by executing created benchmark on a selected set of videos and images.

Before the analysis, the Jetson TX2 board was flashed with the latest kernel L4T 32.2.1, in order to install the CUDA framework in its version 10.0. The TensorFlow framework was compiled from source and executed several times on a target platform. This process allows to introduce future modifications and install them on a target platform.

Chapter 8
Future Work

Optimization of the neural network inference implies a deep understanding of the framework underlying architecture that will compute the final result. This thesis aimed to provide this knowledge and summarize tools that are involved in the process of prediction.

In future research, the graph partitions, created from the dataflow graph, may get separated in a more suitable form by taking into account a device SoC architecture. The suitable scheduling mechanism can be a part of the optimization stage, executed during the graph construction, as it receives the graph definition as an input and can reorganize the structure of the final graph, which will proceed further for computation. Also, during the partitioning stage, created partition could change the allocation of particular operations by assigning them to another accelerator device.

The disadvantage of full version machine learning libraries is that they consume many resources during the recompilation process, and the lack of support from NVIDIA regarding the cross-compilation prolongs the whole process of development. CUDA and cuDNN libraries that are the prerequisites for almost all accelerated code in the machine learning frameworks are provided only via Jetson SDK manager, without any further documentation.

On the other hand, the TensorFlow framework supports cross-compilation only for the TensorFlow Lite library, but it does not support aarch64 architecture with GPU, this is what Jetson TX2 is. It means that custom Bazel build configuration must be created, and all required libraries must be mapped. The article from the online source[1] described the modification needed for cross-compilation of the TensorFlow, but it seems outdated, as during the experiments it did not lead to the solution.

The ARM architecture brings a big challenge for compiling significant codebases in the target system itself. Bazel, introduced in Section ??, supports local and remote caching. It speedups the whole process (in case of fast connectivity), but it introduces additional overhead during the development, as it requires local server or remote GCP Storage[2] instance.

For the future reference, the advantage of the introduced Docker container runtime may solve a problem, in case of providing the NVIDIA libraries via Docker volumes and creating the image on the host device, capable of more computational power.

[1] https://jany.st/post/2018-02-05-cross-compiling-tensorflow-for-jetson-tx1-with-bazel.html
[2] https://cloud.google.com/storage/

The benchmark created in this thesis can rely on general public datasets, such as MS COCO[1], KITTI[2], VOC2012[3] to introduce comparable results with other available benchmarks. The measurement of the video benchmark could be improved by calculation of each frame as an independent image and, at the end, computing the average across all frames. The ImageNet VID challenge utilizes this schema[4]

[1] http://cocodataset.org/#home
[2] http://www.cvlibs.net/datasets/kitti/
[3] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html
[4] http://image-net.org/challenges/LSVRC/2017/

Appendix A
Acronyms

ADAS	Advanced Driver Assistance System
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
BFS	Breadth-First Search
CNN	Convolutional Neural Network
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GUI	Graphic User Interface
HTTP	Hypertext Transfer Protocol
NPU	Neural Processing Unit
OS	Operational System
SDK	Software Development Kit
SoC	System-on-Chip
TPU	Tensor Processing Unit

Appendix B
CD Content

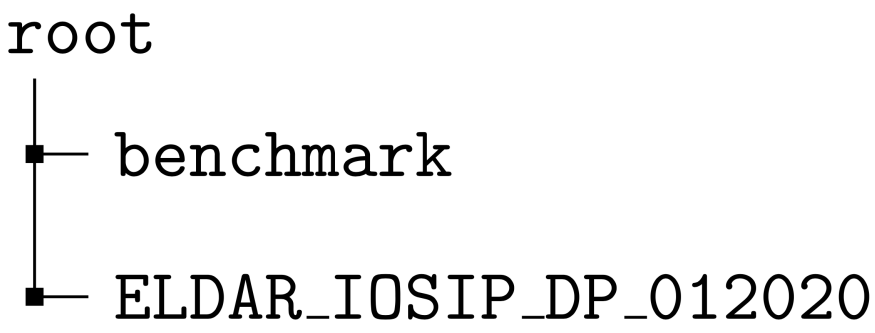

Figure B.1. Content of the attached CD.

Appendix C
DarkNet-53

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure C.2. DarkNet-53 [4].

References

[1] J. Kim , D. S. Han , and B. Senouci . *Radar and Vision Sensor Fusion for Object Detection in Autonomous Vehicle Surroundings.* In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2018. 76-78.

[2] Dustin Franklin. *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge.* 2018.
`https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/`.

[3] Ami Woo, Baris Fidan, and William W. Melek. *Localization for Autonomous Driving*. 2019.
`https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119434610.ch29`.

[4] Joseph Redmon, and Ali Farhadi. YOLOv3: An Incremental Improvement. *CoRR*. 2018, abs/1804.02767

[5] David Held, Sebastian Thrun, and Silvio Savarese. Learning to Track at 100 FPS with Deep Regression Networks. *CoRR*. 2016, abs/1604.01802

[6] Mihir Mody. *ADAS Front Camera: Demystifying Resolution and Frame-Rate*. 2016.

[7] Simon Thorpe, Denis Fize, and Catherine Marlot. Speed of Processing in the Human Visual System. *Nature*. 1996, 381 520-2. DOI 10.1038/381520a0.

[8] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. *SIGPLAN Not.*. 2018, 53 (2), 751–766. DOI 10.1145/3296957.3173191.

[9] L. Wang , M. Huang , and T. El-Ghazawi . *Exploiting concurrent kernel execution on graphic processing units.* In: *2011 International Conference on High Performance Computing Simulation*. 2011. 24-32.

[10] T. Amert , N. Otterness , M. Yang , J. H. Anderson , and F. D. Smith . *GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed.* In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. 2017. 104-115.

[11] Waqar Ali, and Heechul Yun. Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms. 2017,

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

[13] hadican. Similarity between biological and artificial neural networks. 2019,

[14] *Stanford Course: Convolutional Neural Networks for Visual Recognition*.
`http://cs231n.github.io/neural-networks-1/`.

[15] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. *Maxout Networks*. 2013.

[16] A Framework for Designing the Architectures of Deep Convolutional Neural Networks. *Entropy*. 2017, 19 (6), 242. DOI 10.3390/e19060242.

[17] Matthijs Hollemans. The process of convolution on the image. 2016,

[18] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. *Speed/accuracy trade-offs for modern convolutional object detectors*. 2016.

[19] Xiongwei Wu, Doyen Sahoo, and Steven C. H. Hoi. *Recent Advances in Deep Learning for Object Detection*. 2019.

[20] Shivang Agarwal, Jean Ogier Du Terrail, and Frédéric Jurie. *Recent Advances in Object Detection in the Age of Deep Convolutional Neural Networks*. 2018.

[21] Zuoxin Li, and Fuqiang Zhou. *FSSD: Feature Fusion Single Shot Multibox Detector*. 2017.

[22] Bichen Wu, Alvin Wan, Forrest Iandola, Peter H. Jin, and Kurt Keutzer. *SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving*. 2016.

[23] NVIDIA TensorRT. NVIDIA TensorRT. 2019,

[24] Sophia Turol. A Broad Spectrum of TensorFlow APIs Inside and Outside the Project. 2017,

[25] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. *Exploiting Parallelism Opportunities with Deep Learning Frameworks*. 2019.

[26] *TensorFlow 1.x distributed architecture*. 2019. `https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/architecture.md`.

[27] RM Larsen. TensorFlow Graph Optimizations. 2019,

[28] TensorFlow developers. Implementation of Control Flow in TensorFlow.. 2016,

[29] fastcompression. *Hardware comparison of Jetson modules.* 2019.

[30] Tomas Kalibera, and Richard Jones. *Rigorous Benchmarking in Reasonable Time.* In: 2013.

[31] Reinder Bril, Gerhard Fohler, and WFJ Verhaegh. Execution times and execution jitter analysis of real-time tasks under fixed-priority pre-emptive scheduling. 2008,