Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Efficient Implementation of Neural Networks for Real-Time Applications

Matěj Suchánek

Supervisor: Ing. Jan Čech, Ph.D.
August 2020

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Suchánek Matěj**	Personal ID number:	**474573**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Efficient Implementation of Neural Networks for Real-Time Applications

Bachelor’s thesis title in Czech:

Efektivní implementace neuronových sítí pro použití v reálném čase

Guidelines:

- Make a literature review, find out options to speed up inference of convolutional neural networks.
- Study:
1) Lossless methods, where the improved computational time does not affect accuracy.
2) Lossy methods, which make a trade-off between accuracy and computational time.
- In case 1), focus on translating neural networks into NVidia Tensor RT. In both cases, perform a quantitative experiment, which evaluates inference time (and accuracy) as a function of input size.
- Carry the experiments out on various hardware (CPU, different GPUs, embedded devices, etc.).

Bibliography / sources:

[1] NVidia. Tensor RT Developer's Guide. SWE-SWDOCTRT-001-DEVG_vTensorRT 7.0.0. 2020. https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html
[2] S. Han, H. Mao, W. J. Dally. Deep Compression: Compressing Deep Neural Networks With Pruning, Trained Quantization And Huffman Coding. In Proc. ICLR, 2016.
[3] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. ArXiv preprint arXiv:1704.04861, 2017.
[4] X. Zhang, X. Zhou, M. Lin, J. Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In Proc. CVPR, 2018.

Name and workplace of bachelor’s thesis supervisor:

Ing. Jan Čech, Ph.D., Visual Recognition Group, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Jan Čech, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank my supervisor Jan Čech for providing me continuous guidance on my thesis, his fellow assistant Milan Šulc for several valuable hints, system administrator Daniel Večerka for his remote technical support, and my family for their support during these difficult times.

Declaration

Abstract

Neural networks are currently one of the most common methods in machine learning. They have established a new scientific discipline known as "deep learning" and have been successfully applied in many research fields, such as computer vision, speech recognition, or machine translation.

In most fields, the primary and sometimes only concern is good accuracy. It can be achieved by training on large amounts of human-labeled data. However, real-time applications, such as autonomous driving, demand both good accuracy and fast, efficient inference.

This thesis provides an overview of known methods of improving neural network performance, with a primary focus on convolutional neural networks. It also presents a series of experiments that measure the efficiency of these methods applied to various neural network architectures and run on different platforms and discusses the results.

Keywords: neural networks, deep learning, quantization, real-time

Supervisor: Ing. Jan Čech, Ph.D.

Abstrakt

Neuronové sítě jsou v současné době jednou z nejpoužívanějších metod ve strojovém učení, která dala vzniknout vědecké disciplíně známé jako hluboké učení. Dosud byly úspěšně nasazeny v mnoha výzkumných odvětvích, jako jsou počítačové vidění, rozpoznávání řeči nebo strojový překlad.

Ve většině odvětví je hlavním a někdy jediným měřítkem úspěchu přesnost. Té lze dosáhnou trénováním na velkém množství člověkem označených dat. Nicméně některé aplikace, pracující v reálném čase, jako jsou například autonomní vozidla, vyžadují kromě dobré přesnosti i rychlé a efektivní vnímání.

Tato práce poskytuje přehled známých metod pro zlepšení výkonu neuronových sítí se zaměřením na konvoluční. Také zahrnuje několik experimentů, které měří účinnost použití těchto metod na různé architektury neuronových sítí spuštěné na různých platformách, a diskutuje jejich výsledky.

Klíčová slova: neuronové sítě, hluboké učení, kvantizace, aplikace v reálném čase

Překlad názvu: Efektivní implementace neuronových sítí pro použití v reálném čase

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

Neural networks have become the state-of-the-art method for various machine learning problems. In the last decade, much research was done, and different architectures for solving problems in different fields were established, notably convolutional networks for computer vision or recurrent neural networks for natural language processing.

Novel architectures built on top of these approaches then succeeded in various annual competitions. The main challenge was achieving the best accuracy – the portion of correctly classified data based on prior training on vast amounts of human-labeled data.

In real-time applications, however, the networks need to be both precise and efficient. For example, in autonomous driving, decisions must be delivered quickly and be accurate to prevent collisions and accidents. Mobile and embedded devices are memory- and energy-constrained, and their hardware, which needs to fit their size, is less powerful.

1.2 Thesis outline

The thesis is divided into two main parts.

The first one, chapter 2, presents the main findings of the literature review. Section 2.1 provides an overview of how some types of hardware help speed up computations of neural networks. Section 2.2 describes a systematic approach called "deep compression", which can reduce the size of a neural network. Section 2.3 describes network quantization, a class of techniques which pursue the algebra inside neural networks and replace expensive calculations with less expensive. Section 2.4 documents TensorRT, a software library for deep learning inference, its features, and its compatibility with other frameworks. Finally, section 2.5 enumerates several network architectures that were designed with efficiency in mind.

The second part, chapter 3, presents a series of experiments conducted to prove the efficiency of the methods discussed in the second chapter. It describes the motivation behind each experiment, software, hardware, and data used for carrying out the experiment, the results, and discussion, which are illustrated with charts generated from the measured data.

The final chapter summarizes the whole thesis and provides a conclusion. It is followed by a list of used literature and appendices.

Chapter 2

Literature review

In the literature, we identified the following possibilities for making the inference of neural networks faster. They are explored further in individual sections:

- Ordinary hardware may not be designed for running inference. Specialized hardware (notably GPUs) may accelerate neural networks using techniques of data multiprocessing, such as parallelization or SIMD[1]. (Section 2.1)
- Neural networks may hold too many parameters, which are needed for calculating the prediction. There are ways of reducing their numbers. (Section 2.2)
- Arithmetic computations are mostly carried out over standardized floating-point values. It is possible to use values with fewer bits and faster machine language instructions. (Section 2.3)
- Software libraries for deep learning provide APIs for high-level programming languages (mainly Python), known to have runtime overheads (garbage-collection, etc.). Migrating to using a low-level programming language for inference removes these overheads and can make inference faster. (Section 2.4)
- State-of-the-art architectures may be too complex and heavy for inference on devices with limited performance. In-depth research on network structure produced new alternative architectures that aim to improve their performance. (Section 2.5)

2.1 Hardware acceleration

Central processing units (CPUs) are general-purpose hardware, designed for many situations. Although it is possible to run parallel algorithms on multi-core CPUs, it does not generally scale well [5].

[1] Stands for "Single Instruction, Multiple Data".

GPUs are designed for parallel computations. Their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms that process large blocks of data in parallel [6]. In fact, GPUs are already widespread in deep learning, and their development contributes to the success of deep learning. A notable actor in this field is the NVIDIA Corporation, which focuses on developing new-generation GPUs and the demands of employing deep learning in the industry.

Originally, GPUs were used for efficient handling of computer graphics [6]. Later, with the adoption of general-purpose computing on GPU, researchers experimented with training neural networks using (possibly multiple) GPUs. Thanks to that, AlexNet, an early architecture of a deep neural network, became the first of its kind to win the ImageNet Challenge in 2012 [7, 8]. This strategy was subsequently adopted by other researchers, resulting in success of achieving human-level accuracy in image categorization problem [8]. As latency became a concern later, GPUs were also found to be efficient for inference, improving both energy efficiency and speed [9, 10].

Some libraries ease developers and researchers moving computations to GPUs. An NVIDIA-specific framework is CUDA, an abstract model for parallelizing algorithms on GPUs and an API for CPU-GPU cooperation, memory management, and running code on GPUs. Some deep learning frameworks, such as PyTorch or TensorFlow, allow training models and inference using CUDA-enabled GPUs. Some may also provide optimized and more efficient algorithms for mathematical operations inside neural networks. For instance, NVIDIA's cuDNN library implements `im2col` operation and Winograd algorithm for fast calculating of convolutions [11]. Section 2.4 documents TensorRT, an NVIDIA's library for fast deep learning inference on GPUs.

Besides GPUs, new specialized architectures have emerged recently:

- NVIDIA Deep Learning Accelerator (DLA) is an open project promoting a specialized architecture that addresses the computational demands of inference [12].
- Tensor Processing Unit (TPU) is a type of integrated circuits developed by Google. They are designed for training large networks with massive amounts of data in the cloud. Nonetheless, Edge TPU, a version for inference, is available [13].
- Efficient Inference Engine (EIE) was designed to improve inference speed of compressed models with sparse structure (see sect. 2.2) [14].

2.2 Network compression

Neural networks consist of numerical parameters ("weights"), which are organized into groups, called "layers". Values of weights are learned after training on labeled data. During inference, the weights together with the input are used to calculate the network output. The calculations usually

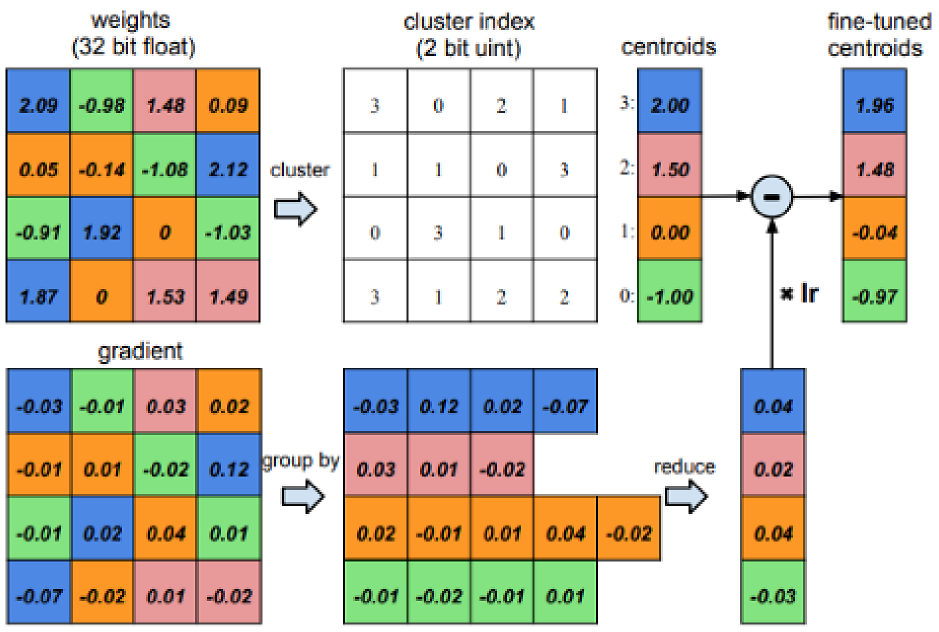

Figure 2.1: Illustration of weight sharing, and centroids update during back-propagation, as proposed by deep compression algorithm. Image from [1].

involve matrix multiplications, which themselves mainly consist of many dot products. Consequently, each weight must be loaded into the device operating memory during inference at some point, potentially multiple times, so that the device can carry out these computations.

However, there exists a natural trade-off between accuracy and computational efficiency. Deep networks may be successful in solving various problems, but for devices with smaller memory and higher memory access cost, such as mobile devices, repeated loading weights into memory can make inference even slower, especially when the relatively small memory can hold only a subset of them.

To reduce the number of weights needed for efficient inference while preserving comparable accuracy, a systematic approach called "deep compression" was proposed. This process reduces the size of the network by eliminating redundant weights and only storing the important ones. The decision on their importance is determined during training [1]. It is based on results from previous research that concludes the network can learn both to generalize the data and also the "connectivity" – which connections are important and which are possibly redundant. As in the whole deep learning field, there is said to be an analogy with how human brains work [15].

Initially, the network is trained using the standard procedure by forwarding training data and back-propagating the error. When the training finishes and the optimal values for weights are found, some are chosen according to specific criteria and pruned (set to zero). A simple example of such criteria is a fixed threshold of absolute value, but it may be more complex [1, 15].

Setting weights in the layer matrix to zero does not save memory space and computation time on its own. Instead, the matrix becomes sparse, meaning many of its elements are zero. Therefore, the new weights are stored in compressed sparse row (CSR) or compressed sparse column (CSC) format, which save only non-zero matrix elements and their positions (indices), as opposed to sequential storage of all elements in a multi-dimensional array. After pruning, the network should be re-trained to mitigate possible accuracy loss (not considering the pruned connections anymore) [1, 15].

Secondly, to further reduce the amount of data needed to store the whole network, the process continues with weights sharing. For each layer, the remaining weights are clustered using the k-means algorithm.[2] When it converges, the centroids become the only values of weights in this layer. Consequently, layer matrices do not store the values but only indices to the table of centroids, which occupy less space. (Space efficiency is achieved when for the given number of bits n the number of centroids is 2^n.) Analogously to weight pruning, the network should be re-trained to recover the loss of accuracy. The gradients are grouped by centroid; each group is summed together and subtracted from the corresponding centroid (see fig. 2.1) [1].

The last proposed step in this process is compressing the network with Huffman code, which further increases the compression rate for storing the model on the drive [1].

2.3 Network quantization

A widely studied approach of accelerating neural networks is called “network quantization”. This class of techniques allows using less memory for storing weights and activation values during computation. In contrast to the quantization utilized in deep compression (sect. 2.2), the reduced values are directly used for arithmetic operations, not as indices to a lookup table.

2.3.1 Half-precision

Weights of network models are usually stored in *single-precision floating-point format* (referred to as “floats”, denoted with FP32), which has been specified by the IEEE 754 standard. It allows for decimal weights in the range from -3.4×10^{34} to 3.4×10^{34}, more than 4 billion distinct values in total. Each of these weights occupies 32 bits (4 bytes) in the memory [16, 17].

A lighter alternative to 32-bit values is *half-precision floating-point format* (denoted with FLOAT16, FP16, or HALF). Values stored by this data type require 16 bits (2 bytes) and can be used to represent decimal numbers in the range from −65,504 to 65,504 (providing 30,720 distinct values in total). Memory usage is reduced by half, which improves efficiency of computations.

[2]Although the cited paper calls this “quantization” and this is how the k-means algorithm is also sometimes referred to, this thesis uses the term for describing a category of different techniques (sect. 2.3).

The IEEE 754 standard has specified this floating-point format since 2008 [16, 17].

The accuracy of network with FP16 values is expected to be the same as that with ordinary FP32 values [18].

2.3.2 Integer inference

Another possibility is reducing floating-point weights and activation values to integers. This is usually what “quantization” refers to.

Mapping floats to integers

A frequent mathematical operation inside neural networks is matrix (or matrix-vector) multiplication. This involves calculating exactly as many dot products as the number of elements the resulting matrix holds (for two square matrices of dimensions $n \times n$, it is n^2). As dot product of vectors X and Y of dimension n is defined as $\sum_{i=1}^{n} X[i] * Y[i]$, calculating each dot product requires n additions and multiplications of floating-point values (resulting in n^3 operations for the whole matrix multiplication). These two operations are generally faster with integer values, which require less memory for storing, making the computation process more energy and area efficient [8, 19, 20].

The inefficient calculations can be avoided by mapping the real values to discrete integer (“quantized”) values, carrying out the computations in their field, and mapping the results to floating-point values. However, there are significantly fewer integers than floating-point numbers (by orders of magnitude), therefore the calculations just approximate the correct result, and maintaining the same or comparable accuracy of the quantized network becomes a new challenge.

The literature mostly advises using 8-bit integers (256 integers in the fixed range $[-128; 127]$ or $[0; 255]$) due to good hardware support, satisfactory results from experiments, and simple transformation procedures. This thesis mostly deals with procedures of *post-training quantization*, an intermediate step between the usual training procedure, and inference, which mostly works with 8-bit integers [18, 19, 20].

When integer quantization harms network accuracy, it may be necessary to modify the network structure or transform the training process to *quantization-aware training*. This generally happens when using shorter bitwidths for storing weights or activations, such as INT4, INT2 (“ternary networks”), or even INT1 (“binary networks”). Training of such networks will naturally require augmenting the traditional stochastic gradient descent method with new techniques or even using different training methods [18, 19, 20].

One problematic aspect is that quantization is not a differentiable function – its derivative is (almost always) equal to zero. A technique that deals with this problem is known as Straight-Through Estimator (STE), which replaces the gradient with an estimate. Quantization of weights is treated as the identity mapping, resulting in a unit gradient w.r.t. quantized weights [18, 19, 20, 21, 22].

For purposes of the following paragraphs, we define clipping as a vector function $clip(X, a, b) = [\ clip(X[i], a, b)\ ]$, where:

$$clip(X[i], a, b) = \begin{cases} a & X[i] < a \\ X[i] & a \leq X[i] \leq b \\ b & b < X[i] \end{cases} \tag{2.1}$$

and a, b are the lower and upper bound of an interval, respectively.

Asymmetric quantization

Also known as *scale and shift quantization* or *affine quantization*. This procedure maps the minimal and maximal value from the array (tensor) to the lower and upper bound of the integer range, respectively. First, the whole array is shifted (i.e., a number is added to all its elements), so that the minimal value corresponds to the lowest integer, then scaled by the ratio of the spans of integer and float range (this constant is called “scale factor”), and rounded to the nearest integer. The element-wise formula for asymmetric quantization is:

$$Q_X[i] = round((X[i] - min(X)) * s_X) \tag{2.2}$$

$$= round(X[i] * s_X - min(X) * s_X) \tag{2.3}$$

where X is the tensor to be quantized, Q_X is the quantized tensor, and s_X is the scale factor, a positive real constant, calculated as:

$$s_X = \frac{2^n - 1}{max(X) - min(X)} \tag{2.4}$$

where n is the number of bits available for representing the quantized values. The shift $min(X) * s_X$ is called “zero-point” (alternatively “quantization bias”[3] or “offset”).

Sometimes it is important to ensure that the zero value in the array of floats is represented exactly, without the rounding error. This may be necessary, for instance, for convolutional layers with zero-padding [23, 24]. This is solved by scaling the array, then shifting by an integer that is one of the integral quantized values. Therefore, the zero-point (z_X) is calculated as:

$$z_X = round(min(X) * s_X) \tag{2.5}$$

By substituting 2.5 to 2.3, we get the final formula for the quantize function:

$$Q_X[i] = round(X[i] * s_X - z_X) \tag{2.6}$$

For the “inverse”[4] operation *dequantization*, we infer the following formula:

[3]Not to be confused with neuron bias in fully-connected layers.

[4]This is not an inverse mapping by definition because it is not possible to “recover” the loss of the rounding function.

$$X[i] \approx \frac{1}{s_X}(Q_X[i] + z_X) \tag{2.7}$$

Now given two vectors X and Y, we can calculate the dot product more efficiently by re-arranging the operations:

$$X * Y = \sum_i (X[i] * Y[i]) \approx \sum_i \frac{1}{s_X}(Q_X[i] + z_X)\frac{1}{s_Y}(Q_Y[i] + z_Y) \tag{2.8}$$

$$= \frac{1}{s_X * s_Y} \sum_i (Q_X[i] + z_X)(Q_Y[i] + z_Y) \tag{2.9}$$

Provided z_X and z_Y are integral, the calculations inside the loop involve only integers. Intermediate results should be accumulated to variables with a bigger size (e.g., 16-bit or 32-bit integer) to avoid overflows.

The formula 2.9 does not remove all floating-point arithmetics because the factor $\frac{1}{s_X * s_Y}$ is not guaranteed to be integral. If it is necessary to avoid it altogether, the effect can be approximated by multiplying the result by an integer and then right-shifting (that is, dividing by a positive power of 2) [23, 24].

Symmetric quantization

Also known as *scale quantization* or *linear quantization*. This procedure maps the real values to a quantized symmetric (zero-centered) interval, without shifting. Instead, the range of real numbers is extended to the same length on both sides from the zero using the maximal absolute value from the array as the upper bound and then scaled to the integer range, possibly leaving some unmapped quantized values [23]. (It can be viewed as a special case of asymmetric quantization where the zero-point is equal to zero.)

If all available integer values are to be utilized, the range of the mapping is not symmetric. For 8-bit signed integers, this is $[-128; 127]$; in general, this is $[-2^{n-1}; 2^{n-1} - 1]$. There are two options [18, 23]:

- Use the full range. After scaling and rounding, values above the upper bound are clipped (2.1). The scale factor is calculated as $s_X = 2^{n-1} * max(|X|)^{-1}$ (2.10).
- Drop the lowest integer and make the range of quantized values symmetric. The scale factor is calculated as $s_X = (2^{n-1} - 1) * max(|X|)^{-1}$ (2.11).

Although non-symmetric range provides one more quantized value to map to, quantization using it can introduce a bias towards the negative bound in the results [18].

For both cases, the formulae for symmetric quantization and dequantization are the following:

$$Q_X[i] = round(X[i] * s_X) \tag{2.12}$$

$$X[i] \approx Q_X[i] * \frac{1}{s_X} \tag{2.13}$$

Dot product of vectors X, Y can be calculated as:

$$X * Y = \sum_i (X[i] * Y[i]) \approx \sum_i (\frac{1}{s_X} Q_X[i] \frac{1}{s_Y} Q_Y[i]) \tag{2.14}$$

$$= \frac{1}{s_X * s_Y} \sum_i (Q_X[i] * Q_Y[i]) \tag{2.15}$$

Compared to (2.9), the loop, still consisting of only integer arithmetics, requires significantly fewer operations, making it more suitable for real-time inference and application of SIMD instructions. For example, DP4A (provided by NVIDIA cards) calculates the dot product of two vectors of four 8-bit integers (as a 32-bit integer) and adds the result to another 32-bit integer [16, 25]. Likewise, zero-point subtractions were dropped from the quantization function, making it simpler as well (compare 2.6 and 2.12) [23].

Another advantage of linearity is compatibility some other common operations inside neural networks, such as the ReLU activation function or pooling, which may operate in the quantized space and do not demand prior dequantization[5] [18].

A disadvantage of symmetric quantization can be an excessive span of the interval in case of presence of an outlier in the data or substantially disproportionate distribution of values w.r.t. zero, which can cause that too many quantized values are not mapped to, and there is worse resolution around zero, resulting in accuracy loss. In fact, this is a common situation in layers after the ReLU activation function, which only produces non-negative values. This means the most significant bit of the quantized values becomes unused. If supported by the hardware, it may be possible to map the values to unsigned integer type with the same number of bits, which disposes of twice as many unique, non-negative values, and combine it with the signed counterpart [18].

The problem of outliers can also be solved by setting a threshold and clipping (2.1) all activation values exceeding it (see fig. 2.2). The value choice may be either artificial as a result of analyzing the distribution of activation values (see also below) or determined during quantization-aware training. One technique for the latter is known as Parametrized Clipping Activation (PACT). It augments the lower-bounded ReLU activation function ($max(0, x)$) with an upper bound α ($clip(x, 0, \alpha)$), which is a hyperparameter contributing to the loss function – its value can be learned during training. Consequently, activation values are guaranteed to belong to a bounded interval, which defines the appropriate quantization scale [18, 22, 23].

Another possibility is decomposition of input tensors (e.g., to input channels in convolutional layers) and determining the best quantization scheme for each portion separately ("fine-grained scale") [18, 19, 20].

[5]As the general formula for fully-connected layers ($y = ReLU(Wx + b)$) suggests, the bias addition has to happen in the quantized space as well. "Backup slides" of [2] provide a hint how this might be implemented in TensorRT.

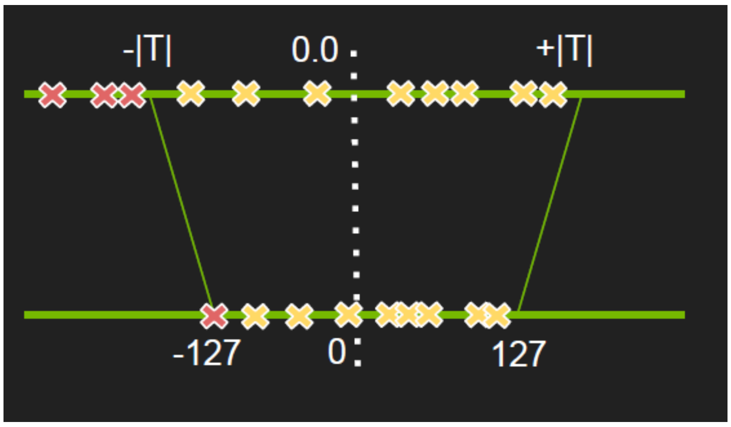

Figure 2.2: Illustration of symmetric quantization with a set threshold. Some outliers are clipped and the range is narrower. Image from [2].

Real-time integer inference

Latency-sensitive tasks need to reduce computations during inference. While the quantization parameters for weights can be determined after training and saved for inference, this cannot be done for the data to be inferred. For example, minima or maxima (2.4, 2.10, 2.11) of the whole tensor in each quantized layer need to be calculated first in order to map the input data to quantized space (i.e. it cannot happen in parallel).

As in the case of outliers (see above), prior knowledge about the forthcoming data helps real-time applications to be more reliable. A systematic approach to preliminary data analysis was incorporated into TensorRT (there called "calibration"), a library for real-time inference (sect. 2.4), where symmetric quantization is used. Determining the best threshold is considered an optimization problem, where the loss of information caused by network quantization is minimized [2, 23].

The process starts with feeding the network with representative data samples. The activation values in individual layers are recorded and accumulated in histograms. After that, the algorithm generates different thresholds, creates a new histogram, adds all exceeding values to the last bin, and generates the corresponding histograms of quantized values. For each of these thresholds, it measures the relative entropy (also known as Kullback-Leibler divergence) of the distribution of real values and quantized values, and chooses the threshold, for which the relative entropy is minimal[6] [2].

2.4 NVIDIA TensorRT

NVIDIA TensorRT is a software development kit for high-performance deep learning inference developed by NVIDIA. It is used for optimizing trained models for specific hardware architecture and thus improving their latency

[6]The pseudocode for this algorithm was provided in "Backup slides" of [2].

during inference. It is built on CUDA, NVIDIA's platform providing an abstract model and an API for running computations on GPUs. TensorRT is not open-source but is freely available [26].

TensorRT provides API for both C++ and Python. Since using C++ is generally faster and eliminates overheads of high-level programming languages, the thesis is only considering the use of C++ API. This approach is recommended for "performance-critical scenarios" and "where safety is important" [27]. Some other deep learning frameworks, such as Google's TensorFlow, have already integrated TensorRT, and they may be another alternative.

2.4.1 Workflow overview

When deploying a model for inference, TensorRT needs to understand the network structure first. There are two ways of loading a model into memory: either by building the network structure and setting the weights programmatically or by importing the network from a file (deserializing) and using a parser. The latest release of TensorRT as of writing (version 7.0) supports parsers for three file formats: ONNX, Caffe and UFF file formats [27].[7]

When the previous step is successful, an optimized inference engine is built by feeding dummy data to the network and observing the performance. The default optimizations include some platform-specific optimizations, removing layers that do not contribute to the network's output and choosing the best CUDA kernels (functions called by many GPU threads in parallel). Further optimizations, such as using different precision for weights (sect. 2.4.2), are available on demand. The created engine can be serialized to a so-called "plan file" and stored for later reuse, which allows to skip all previous steps and thus reduce the time needed for deployment. However, this plan file is not portable, as it is created for the specific platform, GPU, and TensorRT version; therefore, a new instance needs to be built from the original model on any other platform [27, 28].

When the engine is ready, the remaining step is to create an execution context that allocates the memory needed for storing layer weights and activation values and acts as the interface for performing inference. Memory for both inputs and outputs from the network must be allocated and populated on the device (GPU) using CUDA Runtime API, and pointers to the memory must be provided to the corresponding API call. The inference can run synchronously or asynchronously. Synchronous inference suspends the program until a response from the network has been computed on GPU. Asynchronous inference is executed on GPU parallel with the program, which continues execution; the prediction can be received later using synchronization functions from the CUDA API [27, 28].

[7] Caffe and UFF file formats are going to be dropped in an upcoming release.

2.4.2 Lower-precision inference

TensorRT allows using two other data types for storing weights and activation values during inference: half-precision floating-point (sect. 2.3.1) and 8-bit integer format. They can be enabled on demand before the building phase. However, not every hardware has native support for arithmetic operations over them, and also the support may be limited for some types of layers. Furthermore, since TensorRT always chooses the fastest execution strategy for inference, they may only be used for some layers ("mixed-precision inference") or not at all even with proper support [27, 29].

8-bit integer quantization

One available precision mode allows quantizing the network using 8-bit integers. Quantization in TensorRT follows the post-training symmetric quantization (described in detail in section 2.3.2).

The quantization happens during the building phase. The preliminary step is implementing a class that extends one of the abstract classes in the TensorRT API and is responsible for memory management, loading and (optionally) caching data, and preparing a "calibration" dataset. Various resources do not agree on whether the dataset must be entirely disjoint from the training and validation datasets or a subset of which of these two is more appropriate. However, some of them jointly mention the nature of the data to be "representative". Neither is it clearly stated what amount of data samples is sufficient, ranging from "few hundreds" to "1000s of samples" [2, 18, 25, 27].

When the engine for a quantized network is being built, it is first initialized with FP32 weights, and inference is performed on the calibration data. During this process, activation values of each to-be-quantized layer are recorded and collected to histograms. When all calibration data is processed, the best intervals for each layer are determined, and then the building continues with finding the fastest execution strategy. The results of calibration (so-called "calibration cache (table)") can be serialized and saved on the drive and reused when a new engine for the same model is being built on another device [27].

2.4.3 TensorRT, PyTorch, and ONNX file format

Unlike TensorFlow, PyTorch (a Python library for machine learning) has not yet integrated TensorRT in any way. Nevertheless, models trained in PyTorch can be exported to ONNX file format[8], which can be imported to TensorRT, using the following sample (for AlexNet) code [30]:

```
import torch, torchvision

dummy_input = torch.randn(10, 3, 224, 224)
model = torchvision.models.alexnet(pretrained=True)
```

[8]ONNX stands for Open Neural Network Exchange.

```
torch.onnx.export(
    model, dummy_input, "model.onnx",
    input_names=["input"], output_names=["output"])
```

This will serialize the model to the file "model.onnx" and make the serialized model remember the exact provided dimensions, including batch size. Since some network architectures, e.g., fully convolutional networks, are designed to work with variable input size ("dynamic shapes" or "dynamic axes"), TensorRT also offers tools to work with them.

There is also an important difference in handling serialized ONNX models by different TensorRT releases. Before TensorRT 7.0, it was possible to override (maximal) batch size parameter when building an optimized engine from the network. Starting with TensorRT 7.0, this information is hard-coded into the model during serialization.[9]

2.5 Efficient architectures

Development of convolutional neural networks (CNN) was initially driven by improving accuracy, starting with the success of AlexNet [7]. Its successors improved the results further by adding additional complexity (such as residual learning in ResNet [31]) or increasing the number of layers and weights.

As a downside of this evolution, neural networks run more efficiently on high-performance devices but are not as efficient on low-performance devices. Recently, new approaches to neural network architecture design emerged. Some of them are described in this chapter below. Their common denominator is introduction of new types of operations that reduce computational complexity, network size, or number of parameters.

2.5.1 SqueezeNet

SqueezeNet is a CNN architecture that aims to reduce the total number of weights to train and use for inference while preserving good accuracy. It takes advantage of 1x1 convolution filters, which in comparison to 3x3 convolutions, hold nine times fewer weights and are less computationally expensive (given the same input and output size), and uses them to limit the number of input channels to the remaining 3x3 convolutions. Accuracy is maintained by delaying downsampling, i.e., deferring convolutions and pooling operations with stride greater than one to the end of the network. Furthermore, the performance of SqueezeNet can be improved by deep compression (sect. 2.2) [32].

Networks based on this architecture are built from building blocks called "fire modules". Each fire module has two layers. The first ("squeeze") layer consists only of 1x1 convolutions, which moderate the number of input channels to the second ("expand") layer which uses both 1x1 and 3x3 convolutions.

[9]The difference can be observed in the two TensorRT scripts provided on the CD.

The network is a chain of several fire modules interspersed with pooling operations and bypasses [32].

2.5.2 MobileNet

MobileNet architecture focuses on mobile and embedded devices. It aims to reduce the computational complexity of convolutional layers used in various applications for mobile platforms. One of the key means of achieving it is decomposition of standard convolutions to *depthwise separable convolutions*. These operations have two layers. In the first layer, a convolutional filter is applied to each input channel *separately*, producing the same number of output channels as the input channels ("depthwise convolution"). The second layer applies the standard 1x1 convolutional filters ("pointwise convolution") to the output of the first layer, and its output becomes the result of the operation [33].

Consequently, the number of operations needed to compute the result is reduced compared to the conventional convolution. Denoting C the number of input channels, W, H the width and height of the input, K the length of the convolutional kernel (typically 3), and O the number of output channels, the complexity of standard convolution is $H \times W \times C \times K \times K \times O$.[10] On the other hand, the complexity of depthwise separable convolution is $H \times W \times C \times K \times K + H \times W \times C \times O$ (addition of parameters of both layers). Their ratio is $\frac{1}{O} + \frac{1}{K \times K}$, therefore (considering 3x3 filters where $K := 3$) there is an almost ninefold reduction in the cost and the number of weights with small impact on accuracy [33].

2.5.3 ShuffleNet

ShuffleNet architecture is designed for low performance devices. It is built from ShuffleNet units consisting from *group convolution* and *channel shuffle* operations [34]. Group convolution, first introduced by AlexNet [7], splits input channels into several groups, performs convolution on each group separately and stacks their output channels. Complexity of group convolution is $k \times H \times W \times \frac{C}{k} \times K \times K \times \frac{O}{k}$, where k is the number of groups[11] – k-fold reduction compared to standard convolution.

However, chained group convolution has a drawback – the information does not flow between groups, each group gets just a portion of all outputs from the previous layers.[12] Therefore, ShuffleNet augments group convolutions with channel shuffle operations – the output channels are arranged to a matrix respecting the order of groups, the matrix is transposed, and the channels are flattened back. Consequently, it is possible to exchange data between all groups, and network performance is improved thanks to this [34].

[10]Assuming the input and output feature maps have the same height and width, square kernels, and stride = 1.

[11]Considering C and O divisible by k.

[12]By intuition: the more information is available, the better the decision is.

Chapter 3

Experiments

The second part of the thesis involved carrying out a series of experiments to provide a quantitative analysis of how the described techniques influence efficiency of neural networks. We generally measured the dependency between input size and time to process it (inference time) and compared the results of various inference engines (i.e., a combination of devices, hardware, and frameworks).

Unless otherwise noted, the experiments were carried out remotely on a Unix workstation with 6-core Intel Core i7-8700 CPU and a GeForce GTX 1080 Titan GPU [35]. Necessary toolkits and libraries were installed there using virtual environments, mainly Singularity images [36] built from corresponding docker images from NVIDIA GPU Cloud (e.g., [37]). We used TensorRT 7.0 and PyTorch 1.3 releases.

For inference using TensorRT, we developed custom scripts. They are provided on the CD attached to this thesis. Its usage is documented in the Appendix B.

Desirable data preprocessings and postprocessings were done using Python scripts, using NumPy, PyTorch, Torchvision, and Pillow libraries. Torchvision library includes functions useful for computer vision tasks and provides trained models of many popular neural network architectures (some of which were used for purposes of the following experiments) [4].

Some pretrained models provided by Torchvision have a flaw that makes any attempts to import ONNX files generated from them to TensorRT using the provided parser unsuccessful in some environments. While this could be avoided by using the latest release of TensorRT (7.0 as of writing), some platforms only support older releases (for example, Jetson only upgraded from 5.0 to 6.0 as of writing), where some of the models provided by Torchvision could not be parsed and imported to TensorRT.

Presented charts were generated using the `matplotlib` library. Some data in charts are missing due to various technical limitations (such as insufficient runtime memory on GPU).

Measuring times

When inference time was being examined, only the computations done by the network were measured. In TensorRT, we followed the recommended way

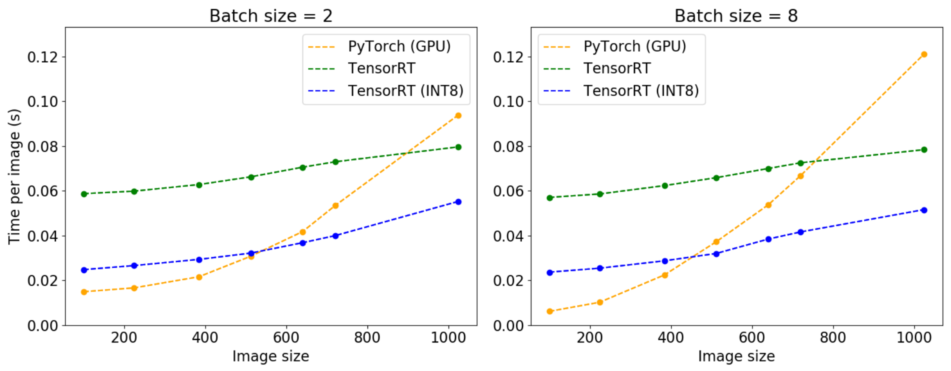

Figure 3.1: Comparison of duration of different RetinaNet engines performing inference with different input sizes (averaged per one image). While for smaller images, inference on GPU using PyTorch performed better than that using TensorRT, TensorRT outperforms PyTorch for large images.

of recording events on GPU provided by CUDA API. The same mechanism is available in PyTorch in `torch.cuda` module. When inference was being done on the CPU, the clock time was measured using the `time` method from Python's native `time` library.

3.1 Object detection using RetinaNet

This was an early preliminary experiment to measure efficiency of using TensorRT. The key indicators were the ultimate performance (time to process the input) and the accuracy of TensorRT compared with the reference method.

We used the repository of the RetinaNet Examples research project,[1] a toolkit taking advantage of RetinaNet detector for object detection [38, 39]. It supported PyTorch and TensorRT as inference engines; therefore, we chose PyTorch as the reference engine for comparison with TensorRT (version 5).

The inference was carried out on COCO “2017 Val images” dataset, which contains 5000 JPEG images of maximal size 640x640 and their annotations in JSON format used by COCO [40]. This dataset was consecutively evaluated by three distinct engines (PyTorch on GPU, TensorRT, and TensorRT with INT8) with different input image sizes and batch sizes. We chose `ResNet50FPN` [31] as the pretrained backbone model for inference. The INT8 calibration dataset was identical to the test dataset.

The two key metrics, inference speed and accuracy, were obtained from the text and file output of `retinanet infer` script, respectively. The annotations were evaluated using COCO API tools [41].

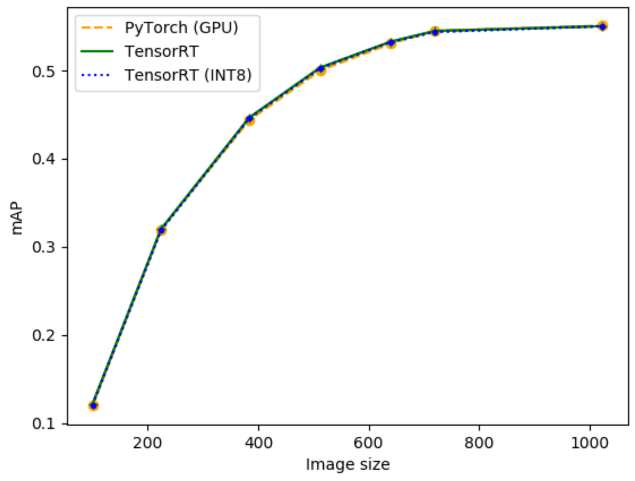

Figure 3.2: Comparison of RetinaNet inference accuracy using different engines.

Results

The graph from figure 3.1 shows the inference time of the three engines where batch size was 2 and 8, respectively. The horizontal axis is the input image size,[2] the vertical axis is the average time of processing one image in seconds. We can see that inference using PyTorch scaled worse for larger images, whereas the increase in time for both TensorRT engines considerably smaller. Still, PyTorch was faster than TensorRT most of the time, probably because of the implementation.

The figure 3.2 shows graph of accuracy per input size. The metric represented by the vertical axis is the “Average Precision at IoU=.50 (PASCAL VOC metric)” (or “mean average precision”, mAP), one of the metrics promoted by COCO [40]. In this graph, we can observe a general trend where detections for smaller images are less accurate due to the objects being captured by fewer pixels and are improving with increasing image size. Since the graphs for three engines overlap, they all provided approximately the same accuracy on the dataset (less than 0.2% difference). Nevertheless, the quantized network had no significant effect on detection accuracy.

3.2 Segmentation using U-Net

This experiment was inspired by a blog post on NVIDIA’s website [25].

The main theme was simulating inference of an autonomous vehicle deployed in city traffic. Such a vehicle captures the surrounding space using a camera and tries to detect objects, their positions, and what kind they are (vehicle,

[1]As of writing, it had been renamed to NVIDIA Object Detection Toolkit and not declared a research project anymore.

[2]The length of the shorter image side after resizing.

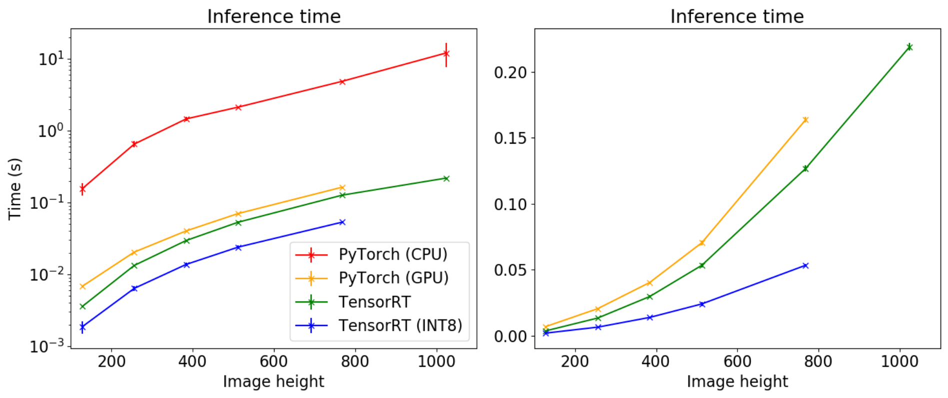

Figure 3.3: Mean latency of inferring images using U-Net with different engines. Both charts present the same data; the left one in log scale, the right one in linear scale without PyTorch CPU. Inference using the baseline PyTorch CPU method was 15 times slower than using GPU. TensorRT could improve this by factor of 1.5, or 3 when quantized network was used.

pedestrian, traffic sign, etc.) in its way. These requirements can be satisfied using a neural network model for *semantic segmentation*, i.e., a model that accepts an image tensor as the input and returns a tensor with class predictions (probability) for each image pixel. A famous architecture developed for semantic segmentation tasks is known as "U-Net" [42].

We used a third-party PyTorch model of the U-Net architecture [43] and evaluated it on the CityScapes dataset, which contains labeled images captured in the streets of German agglomerations [3]. First, the network was trained on the "train split" (3000 images with 1024x2048 resolution) of "leftImg8bit_trainvaltest" package (more than 100 epochs). For simplification, the number of classes was reduced to 8 by considering each category one class[3].

After training, the trained model was serialized to ONNX file format (sect. 2.4.3). Instead of utilizing the "dynamic shapes" feature, a separate instance for each of the following input sizes was created: 128x256, 256x512, 384x768, 512x1024, 768x1536, and 1024x2048 (the aspect ratio of the original images was preserved).

We evaluated 500 files from the validation dataset using four engines: PyTorch on CPU, PyTorch on GPU, TensorRT without quantization, and TensorRT with quantization. For quantization in TensorRT, the network was calibrated (sect. 2.4.2) with 1000 samples from the training dataset (proportionally subsampled per city).

Results

The figure 3.3 shows the results of average inference time measurements per input size for different engines. For clarity, the results are presented both

[3]The categories are referred to as "groups" on the CityScapes website

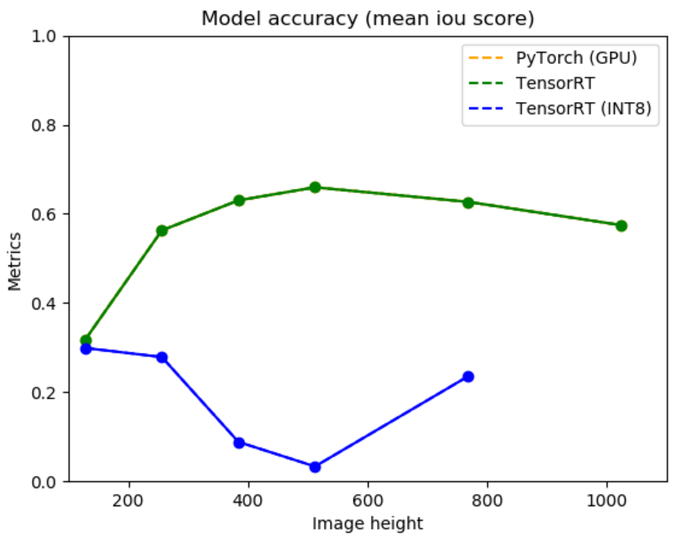

Figure 3.4: Comparison of accuracy of U-Net image segmentation using different engines (omitting CPU). The metric on the vertical axis corresponds to the *mean IoU score* metric. Orange and green lines overlap which means that predictions yielded by PyTorch and TensorRT on GPU without quantization were identical. However, quantized network had severe accuracy loss.

in log scale, and in linear scale without inference using PyTorch on CPU. We can see that inference using the baseline PyTorch inference method on CPU was 15 times slower than that utilizing GPU. Using TensorRT without quantization accelerated it further by a factor of 1.5, still providing the same accuracy (see fig. 3.4). We can also notice TensorRT has better GPU memory management, and unlike PyTorch, it could infer even the largest images.

Quantization in TensorRT allowed twice as fast inference in comparison with TensorRT without quantization. However, the quantized network did not maintain the same accuracy. Figure 3.4 shows the graph of accuracy per input size for PyTorch and TensorRT, including the quantized network. For the standard inference engines, we can again observe the pattern of increasing with increasing input size (see fig. 3.2).

Nevertheless, quantized network did not follow the pattern and suffered from accuracy loss. We do not have a clear explanation for it (neither could we explain the non-monotonicity of the graph in fig. 3.2), but we believe at least one of the following applies:

- The quantization methods provided by TensorRT may not be appropriate for the semantic segmentation problem.
- It is not possible to apply quantization on this architecture (or this specific model).
- The training procedure may have to be aware of quantization (sect. 2.3).

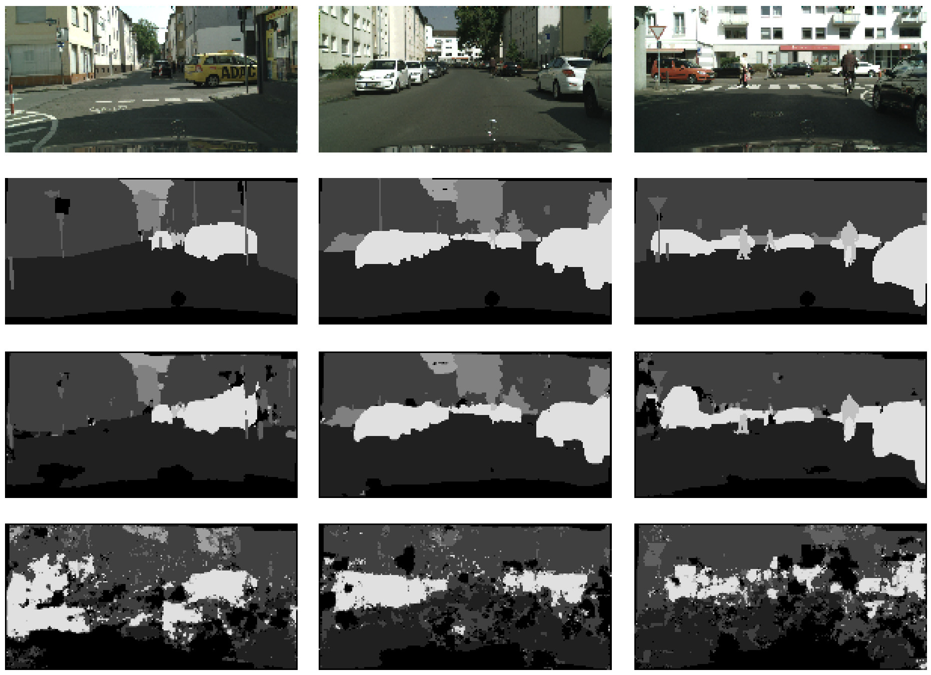

Figure 3.5: Semantic segmentation of sample images (256x512) from the Cityscapes dataset [3]. From top to bottom: input image, ground truth, output from trained model, output from quantized model. The last two rows demonstrate the accuracy loss of TensorRT we observed.

Determining the actual cause of the loss and the odd shape of the graph would demand a more in-depth analysis.

3.3 Image categorization using ResNet

To better understand the problems of network quantization in TensorRT, we did another experiment and focused on image categorization problem – the input to the network is an image, and the output is a mapping of classes to probability the image (or the captured object) belongs to the class. The class with the highest probability (argmax) is considered the actual prediction.

We chose ResNet-50 [31] model provided by Torchvision, which had been trained on the ImageNet database to classify images to 1000 classes [4]. Due to the inaccessibility of the ImageNet database, we evaluated it on Kaggle's "Dogs vs. Cats" train dataset, which contains 25 000 images of cats and dogs (evenly distributed) [44]. Since the ground truth for the dataset consists only of *dog* and *cat*, predictions were considered accurate if the maximal index was in range 151–268 or 281–285, respectively.[4] For calibration, a subset of 1000 images was used (500 dog and 500 cat images).

[4]Inferred from `https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a`.

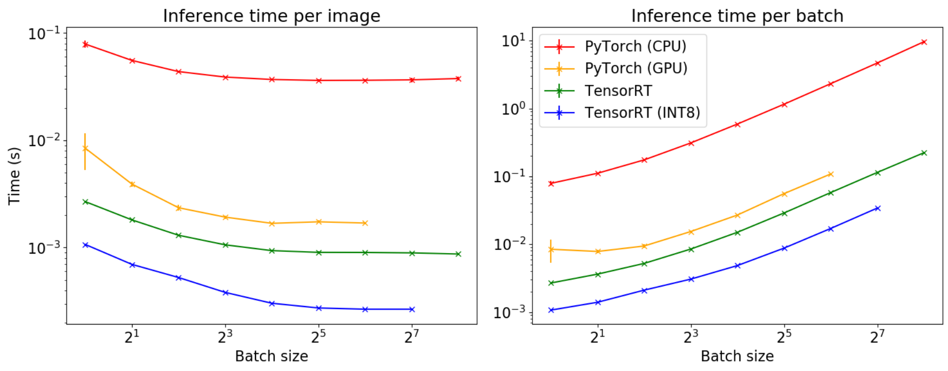

Figure 3.6: Mean latency of ResNet-50 inferring batches of 224x224 images using four different engines. The left graph shows the average time for a single image whereas the right one for the whole batch. Both horizontal (batch size) and vertical (duration of inference in seconds) axes in either graph are in log scale.

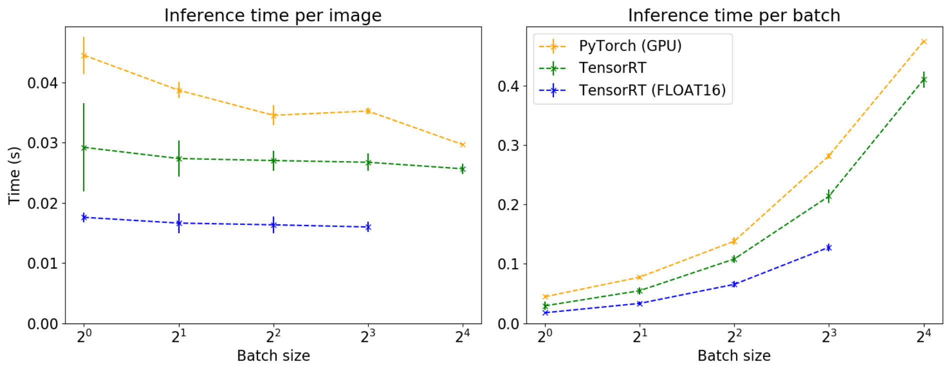

Figure 3.7: Mean latency of ResNet-18 inferring batches of 224x224 images on a Jetson Nano using three engines. The left graph shows the average time for a single image whereas the right one for the whole batch. The horizontal axes (batch size) are in log scale, the vertical axes (time in seconds) in linear scale.

Since the so far used hardware had no support for half-precision inference in TensorRT (sect. 2.3.1), we also run experiments on a Jetson Nano Developer Kit, a small embedded device for deep learning applications [45]. Its hardware includes both an ARM CPU and a GPU (NVIDIA Maxwell architecture), which supports half-precision inference but not integer quantization. We evaluated a portion of the dataset using the ResNet-18 [31] model provided by ONNX model zoo [46].

Results

The data measured ResNet-50 are presented in figure 3.6. We can see that the C++ implementation of TensorRT without quantization running on GPU delivered predictions 100 times faster than PyTorch running on CPU and 2 times faster than PyTorch running on the same GPU. Quantized network

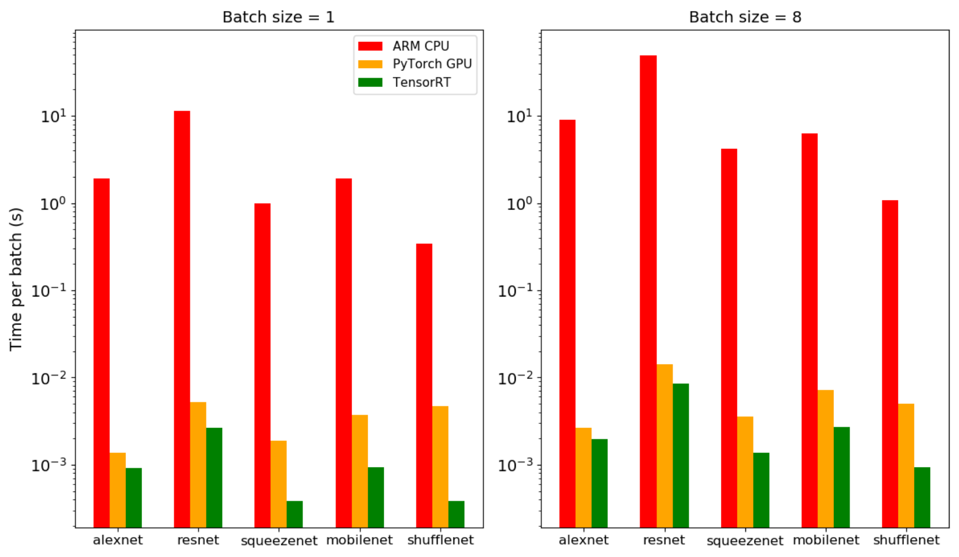

Figure 3.8: Comparison of performance of five image categorization models, using different inference engines (log scale).

in TensorRT was even 4 times faster and maintained the same accuracy on the dataset (86.93%). We can also again notice that TensorRT has better memory management than PyTorch and allows inferring even more images simultaneously.

Figure 3.7 presents the measurements from Jetson Nano. Inference using TensorRT outperformed PyTorch by a factor of 1.5, though the ratio decreased with increasing batch size. Half-precision mode in TensorRT supported on this device improved performance by a factor of 1.6 and did not induce accuracy loss. Inference using PyTorch on CPU was not measured because it was not effective.

3.4 Efficient architectures

The last experiment examined performance of several neural network architectures, including some "efficient" architectures (see sect. 2.5). We compared performance and accuracy on the "Dogs. vs. Cats" dataset of the following architectures for image classification: AlexNet [7], ResNet-50 [31], SqueezeNet 1.1 (revision of [32]), MobileNet V2 (revision of [33]), and ShuffleNet V2 (revision of [34]).

Trained models of these networks were downloaded from Torchvision [4]. The performance was measured two different platforms: on the computer with GTX 1080 GPU (using PyTorch and TensorRT) and Jetson Nano's ARM CPU (using PyTorch). Using the latter was meant to simulate inference on low-performance devices, such as mobile phones, which some of the efficient architectures target.

Model	Size (MB)	Top-1 error (Dogs vs. Cats)	Top-1 error (ImageNet)
AlexNet	244.4	28.11	43.45
ResNet-50	102.5	13.07	23.85
SqueezeNet 1.1	5.0	26.28	41.81
MobileNet V2	14.2	15.2	28.12
ShuffleNet V2	5.6	28.22	30.64

Table 3.1: Accuracy and size (on drive) of the five image categorization models. Data about ImageNet error are from [4].

Results

Comparison of the models' performance is shown in figure 3.8. Table 3.1 presents data about the models' size and their accuracy on the dataset. We observed the following:

- As demonstrated in the previous experiments, the best performance can be achieved by optimizing the models for running on a modern GPU using TensorRT. Performance of ARM CPU was always the worst by orders of magnitude.
- ResNet-50 has the best accuracy of all the five models but the worst performance on all three engines.
- ShuffleNet V2 achieved the best performance when evaluated using both ARM CPU and TensorRT. This was not the case of PyTorch on GPU, where AlexNet had the best performance, despite being the heaviest model. The second-best performance with PyTorch on GPU was achieved by SqueezeNet 1.1. This divergence may be due to naive implementations of special convolutional operations on GPU in PyTorch (see sect. 2.5).
- AlexNet and MobileNet V2 have comparable running times on the low-performance CPU, but MobileNet V2 model is lighter and has better accuracy.

Chapter 4

Conclusion

In the thesis, we presented various approaches and methods to accelerating deep learning inference. We did experiments where we measured effectivity (time vs. accuracy) and compared the performance of different inference engines on different platforms and hardware. We dealt with neural networks designated for different challenges in machine learning and also with different architectures for the same challenge.

We explored features of the TensorRT library and observed that optimizing models using TensorRT usually results in an improvement on its own. Therefore, we believe it can become a state-of-the-art technology for accelerating neural network inference. We also experimented with the quantization method introduced by TensorRT and got both satisfying and surprising results. Specifically, we observed accuracy loss in quantization of a network for semantic image segmentation but not in the case of image categorization.

We also compared different network architectures for image categorization and observed that their relative performance with different inference engines might differ. We noticed that some supposedly "efficient" architectures could indeed make inference faster on low-performance engines while some high-performance engines might not fully exploit their performance advantage.

Although we presented and described deep compression as one of the available techniques in the thesis (sect. 2.2), we did not experiment with it for lack of support in the used frameworks. Similarly, we did not research advantages of quantization-aware training and limited ourselves to either training our models or downloading pretrained ones from the web.

During the experiments, we also experienced some difficulties when optimizing available pretrained models on Jetson, a platform for embedded devices. They were mostly related to incompatibilities of releases of required machine learning frameworks. We hope these issues will eventually be solved with advancing adoption of the frameworks in research and industry.

Bibliography

[1] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*, abs/1510.00149, 2016.

[2] Szymon Migacz. 8-bit Inference with TensorRT. GPU Technology Conference, May 2017. `https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf`.

[3] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[4] torchvision.models. PyTorch master documentation. `https://pytorch.org/docs/stable/torchvision/models.html` [Online; accessed on 2020-07-10].

[5] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking State-of-the-Art Deep Learning Software Tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, Nov 2016.

[6] Wikipedia contributors. Graphics processing unit — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=964264485`, 2020. [Online; accessed 10-July-2020].

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.

[8] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

[9] Michael Andersch. Inference: The Next Step in GPU-Accelerated Deep Learning. NVIDIA Developer Blog, Jun

2018. https://developer.nvidia.com/blog/inference-next-step-gpu-accelerated-deep-learning/ [Online; accessed on 2020-07-10].

[10] NVIDIA. GPU-Based Deep Learning Inference: A Performance and Power Analysis, Nov 2015. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf [Online; accessed on 2020-07-10].

[11] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun 2016.

[12] NVDLA Primer. http://nvdla.org/primer.html. [Online; accessed on 2020-07-10].

[13] Cloud Tensor Processing Units (TPUs). https://cloud.google.com/tpu/docs/tpus. [Online].

[14] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Jun 2016.

[15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1135–1143, 2015.

[16] Mark Harris. Mixed-Precision Programming with CUDA 8. NVIDIA Developer Blog, Apr 2019. https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/ [Online; accessed on 2020-07-10].

[17] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[18] Hao Wu. Low Precision Inference on GPU. GPU Technology Conference, 2019. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9659-inference-at-reduced-precision-on-gpus.pdf.

[19] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *ArXiv*, abs/1806.08342, 2018.

[20] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural Network Distiller: A Python Package For DNN Compression Research. October 2019. https://nervanasystems.github.io/distiller/quantization.html. Accessed on 2020-07-10.

[21] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *ArXiv*, abs/1308.3432, 2013.

[22] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *ArXiv*, abs/1805.06085, 2018.

[23] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural Network Distiller: A Python Package For DNN Compression Research. October 2019. `https://nervanasystems.github.io/distiller/algo_quantization.html`. Accessed on 2020-07-10.

[24] Google. Building a quantization paradigm from first principles. GitHub, Jan 2019. `https://github.com/google/gemmlowp/blob/master/doc/quantization.md`. [Online].

[25] Joohoon Lee. Fast INT8 Inference for Autonomous Vehicles with TensorRT 3. NVIDIA Developer Blog, Sep 2018. `https://devblogs.nvidia.com/int8-inference-autonomous-vehicles-tensorrt/`. [Online; accessed on 2020-07-10].

[26] NVIDIA TensorRT. NVIDIA Developer. `https://developer.nvidia.com/tensorrt` [Online; accessed 2020-07-10].

[27] NVIDIA TensorRT Documentation: Developer Guide, 2020. `https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html` [Online; accessed on 2020-07-10].

[28] Piotr Wojciechowski, Purnendu Mukherjee, and Siddharth Sharma. How to Speed Up Deep Learning Inference Using TensorRT. NVIDIA Developer Blog, Nov 2018. `https://developer.nvidia.com/blog/speed-up-inference-tensorrt/` [Online; accessed on 2020-07-10].

[29] NVIDIA TensorRT Documentation: Support Matrix, 2020. `https://docs.nvidia.com/deeplearning/sdk/tensorrt-support-matrix/index.html` [Online; accessed on 2020-07-10].

[30] torch.onnx. PyTorch master documentation. `https://pytorch.org/docs/stable/onnx.html` [Online; accessed on 2020-07-10].

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun 2016.

[32] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *ArXiv*, abs/1602.07360, 2017.

[33] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv*, abs/1704.04861, 2017.

[34] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Jun 2018.

[35] NVIDIA GeForce GTX 1080 Ti Specs. `https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`. [Online].

[36] Singularity. `https://sylabs.io/singularity/`. [Online].

[37] TensorRT. NVIDIA GPU CLOUD. `https://ngc.nvidia.com/catalog/containers/nvidia:pytorch` [Online; accessed on 2020-07-10].

[38] NVIDIA Corporation. RetinaNet Examples. GitHub, 2019. `https://github.com/NVIDIA/retinanet-examples/tree/c12523c4591afd47af954e6f3ae2b70ff517fb12`. [Online].

[39] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal Loss for Dense Object Detection. *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017.

[40] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *Computer Vision – ECCV 2014*, pages 740–755. Springer International Publishing, 2014. [Online: `http://cocodataset.org/#download`].

[41] Piotr Dollar and Tsung-Yi Lin. COCO API. GitHub, 2014. `https://github.com/cocodataset/cocoapi`. [Online].

[42] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Lecture Notes in Computer Science*, pages 234–241. Springer International Publishing, 2015.

[43] Jackson Huang. U-Net implementation in PyTorch. GitHub, 2017. `https://github.com/jaxony/unet-pytorch/blob/master/model.py`. [Online].

[44] Dogs vs. Cats. Kaggle. `https://www.kaggle.com/c/dogs-vs-cats/` [Online; accessed 2020-07-10].

[45] Jetson Nano Developer Kit. NVIDIA Developer, Feb 2020. `https://developer.nvidia.com/embedded/jetson-nano-developer-kit` [Online; accessed on 2020-07-10].

[46] `https://github.com/onnx/models/tree/master/vision/classification/resnet`.

Appendix A
Contents of CD

- `Thesis.pdf`
- `inference.cpp`
- `inference_jetson.cpp`

Appendix B
Documentation of TensorRT script

Both `inference.cpp` and `inference_jetson.cpp` can be used for measuring inference speed of TensorRT engines. `inference.cpp` is compatible with TensorRT 7.0, whereas `inference_jetson.cpp` with TensorRT 6.0 (runnable on Jetson). They can be compiled using Makefiles provided in `samples/` directory of the TensorRT package.

Some inspiration for writing these scripts was taken from [28].

Both scripts accept the following command line arguments:

- `--file=` – Path to the model in `.onnx` format to parse and create optimized engine for (mandatory unless `--plan=` provided).
- `--plan=` – Path to the inference engine.
  - When `--file=` is provided, it will serialize the optimized engine and save it to the specified path (possibly overwriting).
  - Otherwise it is mandatory and the inference engine is deserialized from the path. All arguments relevant for building the engine are ignored.
- `--dir=` – Path to directory with files to evaluate. Each file in the directory corresponds to one sample (regardless of batch size). Mandatory for running inference.
- `--output_dir=` – Path to directory where predictions should be saved. File names will correspond to input file names. (Only works with `--batch=1`.)
- `--batch=` – Batch size, i.e. how many samples to process at once (default: 1). Needed also for building on TensorRT 7.0.
- `--half` – If supported by the GPU, it enables half-precision when building the engine (see sect. 2.3.1).
- `--calibration_dir=` – Path to directory with calibration samples. If building happens (`--file=` is provided), it enables INT8 quantization if supported by the GPU (see also sect. 2.4.2).

- `--calibration=` – Path to calibration cache. If `--calibration_dir=` is provided as well, the cache will be serialized and saved (overwritten) to this path. Otherwise it is deserialized and the engine is calibrated using it.
- `--iter=` – How many times to run inference (default: 1). Times will be measured and averaged over all runs.

Files loaded using `--dir=` and `--calibration_dir=` and serialized using `--output_dir=` are plain memory arrays. Given `path` is a path to the file and `data` is a reference to a NumPy's `ndarray`, the following Python code serializes (deserializes) the array to (from) the drive:

```
import numpy as np
with open(path, 'wb') as f:
    f.write(data.tobytes())
with open(path, 'rb') as f:
    data = np.frombuffer(f.read(), dtype=np.float32)
```

When the engine is successfully built or loaded, information about network input and output (called "bindings") and memory usage is printed to standard output. If inference was made, the script also prints total running and average inference time with standard deviation, and the number of processed files.