Chair of Cyber-Physical Systems in Production Engineering
TUM School of Engineering and Design
Technical University of Munich

Benchmarking Real-time Features of Deep Neural Network Inferences

Semester Thesis

at the TUM School of Engineering and Design, Technical University of Munich

Supervised by Assessor, Univ.-Prof. Dr. Marco Caccamo
Contributor, M.Eng. Binqi Sun, M.Sc. Denis Hoornaert
Chair of Cyber-Physical Systems in Production Engineering

Submitted by Yipeng Zhou
03743111

Submitted on Garching, 23. Dezember 2022

Acknowledgments

I would like to thank **Professor Caccamo** for giving me the opportunity to work on this project. In particular, **Binqi Sun** and **Denis Hoornaert** provided important guidance for my thesis. Finally, I would also like to thank **Hongpeng Cao** for introducing me to YOLO networks.

Abstract

Deep Neural Networks (DNNs) play an important role in many fields such as computer vision, speech recognition, and natural language processing. With the rapid development of Internet of Things (IoT), the application of DNNs on edge devices has attracted extensive attention. However, edge devices are limited in Size, Weight, and Power (a.k.a. SWaP), and often require high real-time performance. To address this contradiction, various techniques such as lightweight DNN architectures, model compression, and software or hardware inference acceleration have been developed. In order to evaluate and compare the performance of these techniques, many benchmarks have been completed. However, these current works generally have some limitations: Firstly, most benchmarks are implemented on workstations or servers, and lack of attention to edge devices. Secondly, FLOPS (Floating Point Operations Per Second) is widely used to characterize the inference speed, but this metric cannot exactly reflect the actual inference time of a DNN model on a certain device. Finally, the memory usage of different DNNs has not been deeply explored.

To make up for these shortcomings, based on TensorFlow Lite and RT-Bench, using Cortex_A53, a processor widely used for edge devices, we simultaneously benchmarked the impact of different DNN inference acceleration techniques on **Inference Time**, **Accuracy** and **Memory Usage**. The techniques involved include Quantization, XNNPACK and different model architectures in the field *Image Classification*. In addition, we further explored the real-time features of inferences for YOLOv3-tiny models with different depths, focusing on the analysis of the influence of different *Convolutional* modules on **Inference Time**, **Accuracy** and **Memory Usage**.

Some representative findings are: (i) MobileNet and EfficientNet architectures perform extremely well in terms of inference time, LLC misses and bandwidth; meanwhile, efficientnet_b0 has decent accuracy. (ii) Quantization can provide 30% increase in inference speed, 90% and 85% reduction in LLC misses and bandwidth on average, with no more than 10% sacrifice in accuracy. (iii) After XNNPACK is enabled, the inference time of DNN will be reduced by 25% on average, and its impact on LLC misses and bandwidth fluctuates widely depending on models. (iv) Retired instructions and LLC misses are the most important factors affecting inference time. (v) From **Conv. Layer 1** to **Conv. Layer 5** of YOLOv3-tiny, their effect on the increase of model size is continuously enhanced, while their influence on the growth of retired instructions and inference time continue to decline. **Conv. Layer 5** and **Conv. Layer 1** will increase LLC misses to a greater extent than other *Convolutional* modules, and **Conv. Layer 5** will cause largest bandwidth.

***Index Terms* - Deep Neural Network, Model compression, Inference acceleration, Benchmark, Real-time systems**

Contents

Chapter 1

Introduction

The development of artificial intelligence has a profound impact on our society and firms. Among them, technologies represented by Deep Neural Network (DNN) play an important role in many fields such as computer vision, speech recognition, and natural language processing. It enables a number of industries such as autonomous driving, Internet of Things (IoT) and medical devices to flourish.

In order to enable researchers and practitioners in the field of machine learning to build, train and deploy DNNs more efficiently, many open source platforms have emerged. The most representative of these are TensorFlow from Google and Pytorch from Meta. In order to obtain higher accuracy, DNN models such as ResNet [10, 11] often have large size, require a significant amount of computing resources and can introduce long response time, so they are widely deployed on servers of enterprises and research institutions.

At the same time, the application of DNN on edge devices has also attracted the attention of many researchers and enterprises. Especially in the field of IoT, numerous applications such as robotics, smart factories, and smart warehouses rely on the performance of DNNs on edge devices. However, these edge devices are often limited in Size, Weight, and Power (a.k.a. SWaP), and often require high real-time performance. For example, in the field of autonomous driving, if the data acquired by sensors, such as the position of pedestrians, vehicles and roads, is transmitted to the cloud for processing and then transmitted back, a long delay will be caused, which may lead to serious accidents. Therefore, it is extremely critical to rely on edge devices to process the acquired data locally and ensure acceptable accuracy under the requirements of hard real-time or soft real-time.

To address these conflicting requirements, many researchers started to develop lightweight DNNs, such as MobileNet [13], to meet the needs of mobile and edge applications. Such models often need to pursue smaller size and faster inference speed under the premise of ensuring acceptable accuracy. As a result, many techniques for reducing the amount of computation represented by *Depthwise Separable Convolution* [4] have been proposed. In addition, Neural Architecture Search [7] is also widely used to find the tradeoff between accuracy and inference efficiency; EfficientNet [28] is one of the many results of this attempt.

In addition to developing more lightweight models, model compression techniques have been extensively studied. The most representative ones are parameter pruning and quantization, low-rank factorization, transferred/compact convolutional filters, and knowledge distillation [3]. They streamline the existing DNNs to obtain lightweight models with acceptable accuracy. The compressed network has a smaller structure and fewer parameters, which can effectively reduce computing and storage overhead, and is easy to deploy in a limited hardware environment.

In order to meet the real-time requirements of edge applications, many techniques for accelerating DNN inference have been proposed. It can be roughly divided into two categories: software acceleration and hardware acceleration. In the field of software acceleration, Ten-

sorFlow Lite and XNNPACK have been widely used. TensorFlow Lite is a toolkit developed by Google, which helps developers run their models on mobile, embedded, and edge devices. It uses a more efficient storage format, so that DNNs take up less storage space and can be loaded and executed faster. XNNPACK is a highly optimized library of floating-point neural network inference operators for ARM, x86, WebAssembly, and RISC-V platform, which is used to accelerate high-level machine learning frameworks. In the field of hardware acceleration, more and more companies have joined the race to develop and produce AI-accelerated chips; among them, the commonly recognized ones are NVIDIA's GPU (Graphics Processing Unit) and Google's TPU (Tensor Processing Unit).

The application of lightweight architectures, model compression, software or hardware acceleration enables DNN to achieve good performance in inference on edge devices. To gain a more definitive understanding of the impact of these techniques on the real-time features of DNN inferences, many benchmarks have been completed. They provides useful references for research and industrial implementation in this field. However, these current works generally have some limitations. First of all, most benchmarks only focus on the metric FLOPS (Floating Point Operations Per Second), but it can not accurately reflect the actual inference time of a DNN model on a certain device. Second, few benchmarks explore the memory consumption of DNN inferences, which is equally important for the application of DNNs on edge devices. Finally, most benchmarks only focus on the real-time performance of DNN models on workstations or servers, ignoring the widely used edge devices.

To address these shortcomings, we simultaneously benchmarked the impact of each technique on **Inference Time**, **Accuracy**, and **Memory Usage** on an edge device. The techniques involved include Quantization, XNNPACK and different model architectures in the field *Image Classification*. Two state-of-the-art libraries, TensorFlow Lite and RT-Bench [18], made our project possible. Among them, TensorFlow Lite helped us deploy DNNs on edge devices more efficiently; and RT-Bench, which is an extensible benchmark framework for the analysis and management of real-time applications, helped us obtain various metrics about execution time and memory usage during model inference. Since using CPU for benchmark can obtain the widest impact, the current work of this paper is all completed through CORTEX_A53, and no hardware accelerator is enabled.

Furthermore, we also explored the impact of the specific structure of models on the real-time features of inference. Since YOLOv3-tiny [19–21] is widely used by enterprises due to its fast inference speed and high accuracy in the field *Object Detection*, the research on YOLOv3-tiny has high value. Therefore, in the second part of this paper, we first obtained and trained YOLOv3-tiny models with different depths by duplicating and adding *Convolutional* modules in their backbone (Darknet-tiny). Then based on TF Lite and RT-Bench, we benchmarked their performance on edge devices, focused on analyzing the impact of the added different *Convolutional* modules on **Inference Time**, **Accuracy**, and **Memory Usage**, and finally discussed the stability of their inference.

In the following content, we began with some background knowledge of TensorFlow, TensorFlow Lite and RT-Bench, so that the work of this paper can be better understood. Then, related work was mentioned for comparison with our work. In Chapter 4, we introduced the implementation details of our benchmarks in the fields *Image Classification* and *Object Detection*; then we evaluated the results obtained from the benchmarks in these two fields separately. Finally, we summarized all conclusions and briefly planned future work.

Chapter 2

Background

2.1 TensorFlow Lite

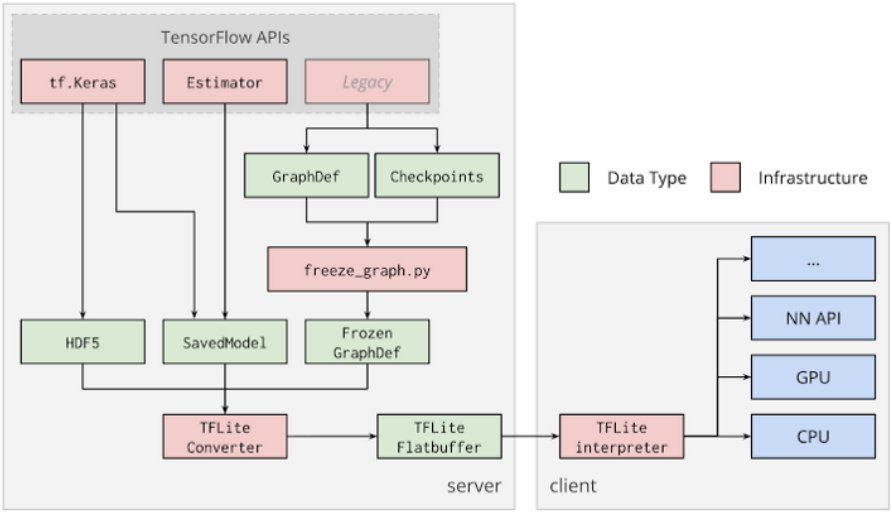

Figure 2.1: Deployment flow of TensorFlow Lite mode at the edge devices.

TensorFlow Lite (TF Lite) is a mobile library for deploying models on mobile, microcontrollers and other edge devices. Since mainstream machine learning platforms, such as TensorFlow or PyTorch, do not provide optimization for deployment to embedded hardwares, deep neural network models developed through these platforms often have the disadvantages of large size and high inference delay. Therefore, they are not suitable for application on edge devices. TensorFlow Lite uses FlatBuffers, which is an efficient cross-platform serialization library, as the storage format (identified by the .tflite file extension); this makes DNN models occupy less disk and can be loaded faster (data is directly accessed without an extra parsing/unpacking step) and executed more efficiently. In addition, since TensorFlow Lite is mainly used for on-board inference, there are no round-trips to a server, no network connection required, and no personal data leaves the device. Therefore, TensorFlow Lite also has advantages in terms of connectivity, privacy, and power consumption. Lastly, TensorFlow Lite supports multiple platforms (such as Android, iOS, embedded Linux, and microcontrollers) and diverse languages (including Java, Swift, Objective-C, C++, and Python), making it widely used by a large number of developers.

2.1.1 TensorFlow Model

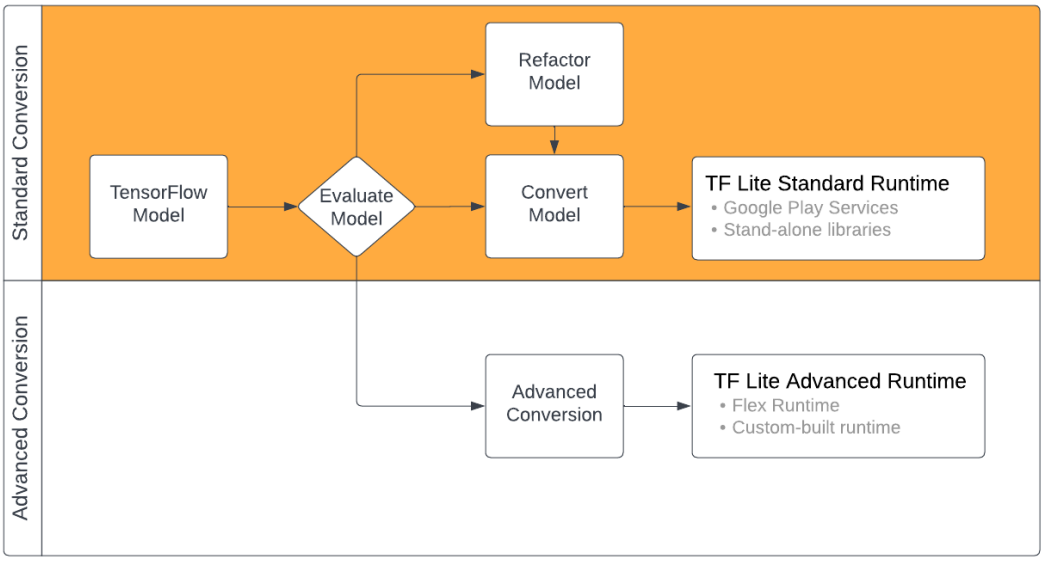

Figure 2.2: TensorFlow Lite conversion workflow.

There are two ways to generate TensorFlow Lite models: the first is to use "TensorFlow Lite Model Maker[1]" to create a model. This library simplifies the process of training a TensorFlow Lite model and supports using transfer learning to reduce the requirement for the amount of training data of custom dataset and shorten the training time. Since there are already a large number of pre-trained TensorFlow models, the second way, converting a TensorFlow model into a TensorFlow Lite model, is more commonly used. As shown in Figure 2.1[2], with the "TensorFlow Lite Converter", we can convert TensorFlow models from various formats (such as SavedModel, Keras model, etc.) into a smaller, more efficient TensorFlow Lite model without losing too much accuracy. It is worth noting that not all TensorFlow models can be directly converted. Only those models that include only the TensorFlow operations supported by the standard TensorFlow Lite runtime environment can be easily converted without extra effort. Otherwise, the model needs to be refactored or use advanced conversion techniques supported by "TF Lite Advanced Runtime". The overall TensorFlow Lite conversion workflow is shown in Figure 2.2[3].

2.1.2 Quantization

Quantization attempts to reduce the bitwidth of the data flowing through a neural network model, thus it is possible to shrink the model size for memory saving and simplify the operations for compute acceleration[5]. Including quantization, TensorFlow Lite currently supports various types of optimization, such as pruning and clustering. They can be specified to apply during conversion. The most important of them is "post-training quantization". It refers to reducing the precision of the numbers used to represent an already-trained float

[1]https://www.tensorflow.org/lite/models/modify/model_maker, accessed on Dec 23, 2022
[2]https://medium.com/techwasti/tensorflow-lite-deployment-523eec79c017, accessed on Nov 28, 2022
[3]https://www.tensorflow.org/lite/models/convert/, accessed on Nov 28, 2022

TensorFlow model when we convert it to TensorFlow Lite format. In addition to reducing model size and inference latency, post-training quantization allows the use of specialized hardware (e.g. Coral Edge TPU) for accelerated inference. TensorFlow Lite provides three post-training quantization options: dynamic range quantization, full integer quantization and float16 quantization. Among them, full integer quantization, which reduces the precision of model parameters from 32bit (float) to 8bit (int/uint), is widely used, because it can achieve the largest improvements of latency, reductions in peak memory usage and compatibility with integer only hardware devices or accelerators. Since weights are quantized post training, an accuracy loss is inevitable. Therefore, it is necessary to use tools[4] to evaluate whether the accuracy degradation of the quantized model is within the acceptable limits.

2.1.3 XNNPACK

XNNPACK is a highly optimized library of floating-point neural network inference operators for ARM, x86, WebAssembly, and RISC-V platforms[5]. The low-level performance primitives provided by XNNPACK are not directly usable by deep learning researchers, but used to accelerate high-level machine learning frameworks. Since 2020, the XNNPACK library has been integrated into TensorFlow Lite, enabling an average of 2.3x faster floating-point inference on CPUs. The XNNPACK backend currently supports a subset of floating-point TensorFlow Lite operators, any unsupported operators would transparently fall-back to the default implementation in TensorFlow Lite. It is worth noting that XNNPACK initially only supports 32-bit floating-point TensorFlow Lite models and TensorFlow Lite models using 16-bit floating-point quantization for weights. Currently, its support for quantization models is being continuously improved.

2.2 RT-Bench

RT-Bench is an open-source framework adding standard real-time features to virtually any existing benchmark [18]. The framework lives fully in userspace and is composed by the RT-Bench Generator. It executes the benchmark periodically and gathers various real-time metrics listed in Table 2.1[6] for each execution statistically. RT-Bench supports several log levels, to log statistics in csv files or in a terminal with different levels of detail. Pinning of the process on a single core or set of cores, changing scheduling policy or constraining dynamic memory allocations during the execution phase can also be realized by RT-Bench. Moreover, due to good extensibility and common interace of RT-Bench, we can add a new benchmark or benckmark set easily and all the benchmark report the same basic statistics and have the same command line interface. At last, RT-Bench allows deployment on multiple platforms, including Operating Systems (OS) and bare-metal software stacks.

2.2.1 Benchmark Design

The good extensibility of RT-Bench allows us to add a new benchmark easily. To achieve this, the new benchmark needs to be refactored into three functions: *benchmark_init*, *benchmark_execution*, and *benchmark_teardown*.

[4] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/evaluation/tasks
[5] https://github.com/google/XNNPACK
[6] Metrics that rely on Performance Monitor Counter are only available for CORTEX_A53.

Table 2.1: List of Metrics Logged by RT-Bench.

Metric	Description	Unit
`period_start`	Period start timestamp	seconds and CPU clock-cycles
`period_end`	Period end timestamp	seconds and CPU clock-cycles
`job_end`	Job end timestamp	seconds and CPU clock-cycles
`deadline`	Absolute deadline timestamp	seconds and CPU clock-cycles
`job_elapsed`	`job_end − period_start`	seconds and CPU clock-cycles
`deadline_status`	Status of the deadline. 1 if met, 0 otherwise.	Boolean
`job_utilization`	`job_elapsed / (period_end − period_start)`	Ratio
`job_density`	`job_elapsed / (deadline − period_start)`	Ratio
`l1_references`	L1 References (PMC)	Absolute number
`l1_misses`	L1 Refills (PMC)	Absolute number
`l1_misses_ratio`	`l1_misses / l1_references`	Ratio
`l2_references`	L2 References (PMC)	Absolute number
`l2_misses`	L2 Refills (PMC)	Absolute number
`l2_misses_ratio`	`l2_misses / l2_references`	Ratio
`instructions_retired`	Instructions retired (PMC)	Absolute number
`cpu_clock_count`	CPU Clock Count (PMC)	Absolute number

- ***benchmark_init:*** initialize the benchmark environment using the supplied parameters. This function is executed only once, so it needs to prepare *benchmark_execution* for periodic execution (e.g. reading data from file, allocation memory, preparing data structures, etc).

- ***benchmark_execution:*** execute the benchmark as if it was launched for the first time. This function will be executed periodically, and all the measurement of the metrics listed in Table 2.1 will only occur at each execution of this function. It will reset any used memory location to its initial value after a benchmark has completed, to ensure that the next benchmark will always have the same environment and hence the same results.

- ***benchmark_teardown:*** clear the benchmark environment initialized by *benchmark_init* and free allocated memory to allow for a clean exit. This function is executed only once when the program is terminating.

In addition, RT-Bench also supports the extension of the report interface, which involves two other functions: *benchmark_log_data* and *benchmark_log_header*.

- ***benchmark_log_data:*** can be used to calculate and return additional metrics (e.g. bandwidth), which will also be logged by RT-Bench.

- ***benchmark_log_header:*** returns a constant string to extend the header of csv files (e.g. ",bandwidth(MB/S)").

Data between different calls of above mentioned functions can through global variables be maintained.

2.2.2 Common Input Interface

RT-bench provides an easy-to-use, unified command-line interface to customize key aspects of the real-time execution of a set of benchmarks[7]. The options in Table 2.2 represent most used options supported by RT-bench, which are available to all benchmarks. Among them, *period* (*-p*) and *deadline* (*-d*) are the only parameters required to run a benchmark; *deadline* must be less or equal than the benchmark *period*. Benchmark specific arguments and options (*-b*) should be specified as the last option, as a single string.

Table 2.2: List of Command Line Options Supported by RT-Bench.

Options	Description
`-p, --period=secs`	Relative period of a single benchmark execution.
`-d, --deadline=secs`	Relative deadline of a single benchmark execution.
`-c, --core-affinity=core1,core2,...`	Core affinity, a single core id is also accepted.
`-m, --mem-limit=bytes[G/M/K]`	Memory limit; if exceeded, the benchmark will crash.
`-f, --fifo=0<=prio<=99`	FIFO scheduling with specified priority, needs root.
`-t, --tasks-number=integer>=0`	Number of tasks to execute before termination.
`-l, --log-level=1/2/3/4`	Log level, 2 means csv files, 3 means terminal.
`-o, --output=output_path`	Default is ./timing.csv.
`-M, --memory-profiling-enable`	Specify 1 to enable Performance Monitor Counter monitoring thread.
`-C, --memory-profiling-core`	Core affinity for Performance Monitor Counter monitoring thread.
`-B, --memory-profiling-time-bucket`	Performance Monitor Counter monitoring sample period, default is 10 ms.
`-b, --bmark-args`	A space-separated list of arguments and options specified by benchmark.

[7] https://gitlab.com/rt-bench/rt-bench

Chapter 3

Related Work

While various DNN architectures, model compression approaches, and inference acceleration techniques are emerging, the work of comparing and benchmarking them has also attracted extensive attention. These related works provide valuable references for our paper.
For different DNN architectures, Simone Bianco et al.[2] benchmarked representative models in the field of image recognition on two different platforms: workstation and embedded system. For each DNN, they obtained multiple performance indices such as recognition accuracy, model complexity, computational complexity, memory usage, and inference time. Their work is useful for researchers to have a complete view of various DNN architectures. It should be noted that their benchmarks enabled GPU to accelerate model inferences, and their research on memory usage is limited to GPU memory footprints, while we focus on the usage of CPU caches for different DNN architectures.
A comprehensive survey of a large number of model compression techniques was conducted by Lei Deng et al.[5] In their paper, the compression principles, evaluation metrics, sensitivity analysis and joint-way use of mainstream compression approaches such as compact model, tensor decomposition, data quantization, and network sparsification are explained in detail. Compression ratio, model accuracy, and hardware usability are used to comprehensively compare these compression techniques. However, the impact of these techniques on inference speed and memory usage is not covered.
In terms of providing software acceleration for DNN inference, Saad Ashfaq et al.[1] developed a quantization framework in Deeplite Neutrino^{TM} that automatically quantizes and optimizes CNN models with less than 4 bits of precision and an innovative ultra-low precision inference engine for Arm CPUs, Deeplite Runtime (DeepliteRT). By effectively using vectorization, parallelization, and tiling, DeepliteRT realized speedups of up to 2x and 2.2x compared to TensorFlow Lite with XNNPACK backend on classification and detection models, respectively.
As the next possible solution to surpass GPU in speed and energy efficiency, FPGA-based neural network inference accelerators are emerging in large numbers. The work of Kaiyuan Guo et al.[9] provides an overview of previous work on FPGA-based accelerators and the main related techniques are summaried. By evaluating in terms of data format, speed, power, efficiency and resource utilization, they compared the performance of the FPGA-based designs published in the top FPGA conferences, EDA conferences and architecture conferences from 2015 to 2017. Among them, OPS (Operations per Second) is used to characterize the inference speed, not the specific inference time. The results of their benchmarks indicate that, with software hardware co-design, FPGA can achieve more than 10x better speed and energy efficiency than state-of-the-art GPU. Their work did not explore the effect of FPGA-based accelerators on the usage of CPU caches, while in order to solve the performance bottleneck of modern point cloud networks on CPU/GPU/TPU, Yujun Lin et al.[17] developed PointAcc, a novel point cloud deep learning accelerator, and analyzed the advantages of PointAcc in

memory usage in detail.
Qianru Zhang et al.[30] comprehensively analyzed the acceleration technology of convolutional neural network from the structure level, algorithm level and hardware implementation level. Although they compared the performance of different techniques in their paper, the focus of their work is not to provide accurate benchmarks.
In addition, many researchers have made outstanding contributions to the development of DNN benchmark platform. Aajna Karki et al.[15] provided a new DNN benchmark suite Tango for various accelerators. It can run on any platform that supports CUDA and OpenCL. Using Tango, they benchmarked some networks on an architecture simulator, a server- and a mobile-GPU, and a mobile FPGA, and using GPGPU-Simulator studied the performance impact of various L1D size. Besides, Shi Dong and David Kaeli [6] presented a GPU benchmark suite, DNNMark. It consists of a collection of deep neural network primitives, covering a rich set of GPU computing patterns and provides a configurable, extensible, and flexible method for profiling computation of individual DNN primitive or composed DNN models. Different from these benchmark platforms developed for accelerators such as GPU, RT-Bench [18], on which our work is based, focuses on providing real-time metrics for benchmarks running on CPU.
Finally, we were also inspired by the research about the influence of specific DNN structures on their inference performance. Mingxing Tan and Quoc V. Le [28] analyzed the impact of scaling network depth, width, and resolution respectively. The results indicated that bigger networks with larger width, depth, or resolution tend to achieve higher accuracy, but the accuracy gain saturate quickly. Our work analyzed the impact of model depth in more detail, separately discussing the effect of different *Convolutional* modules of YOLOv3-tiny on inference time, memory usage and so on.

Chapter 4

Implementation

All of our benchmarks involve two fields: *Image Classification* and *Object Detection*. Under *Image Classification*, we explored the impact of different architectures of models, Quantization technique, and XNNPACK acceleration on real-time features for deep neutral network inferences. While under *Object Detection*, we explored the real-time features and inference stability of YOLOv3-tiny models with different depths. The implementation of these two types of benchmarks has a certain degree of similarity, but there are differences in the details of project construction, model preparation and dataset preparation, etc. Therefore, we first introduced their common points in project construction, then separately discussed the inconsistent details of their implementation, and finally the measures they shared in program execution were presented. The open-source project implementation is available at https://gitlab.lrz.de/chair-of-cyber-physical-systems-in-production-engineering/nn-rt-bench.

4.1 Program Construction

In this section, we will discuss the commonalities in program construction for benchmarks in the field *Image Classification* and *Object Detection*. The differences between them in this respect are introduced in Section 4.2.3 and Section 4.3.4 respectively.
Since RT-Bench is written in C language, we invoked TensorFlow Lite C++ API to build our benchmarks. As mentioned in Section 2.2.1, *benchmark_init*, *benchmark_execution* and *benchmark_teardown* are the necessary parts to build the benchmark adapted to RT-Bench. They serve different roles in our program:

- ***benchmark_init***: This function is used to (a) configure the global variable "Settings" (including the required model, images, labels, etc) with the options from CLI, (b) load and build the TF Lite model, (c) install TF Lite Interpreter, (d) allocate tensors according to the used model, (e) read the file names of all images to be classified or detected and (f) read the labels used by the model.
- ***benchmark_execution***: The main task of this function is to complete model inference. Since each inference of *benchmark_execution* needs to classify/detect different images, reading and pre-processing of images (decode, normalization, standardization, etc) must also be completed in this function; for the same reason, this function is also responsible for the post-processing of output tensor after model inference.
- ***benchmark_teardown***: This function is used for destroying TF Lite Interpreter.

It is worth noting that, the lifetime of variable *model*, *resolver* and *error_reporter*, which are needed to build TF Lite Interpreter, must be longer than the lifetime of TF Lite Interpreter.

Therefore, in addition to TF Lite Interpreter shared by the above-mentioned functions, these three variables also need to be used as global variables.
To compile the benchmark with TensorFlow Lite and RT-Bench, we linked the shared library of TensorFlow Lite (*libtensorflowlite.so*) dynamically. Since there is no official shared TF Lite library, we cross-compiled it for AArch64 (ARM64) using Bazel. As mentioned in Section 2.1.3, XNNPACK is enabled by default in TensorFlow Lite. Another version of *libtensorflowlite.so*, which does not support XNNPACK, can through adding the compilation option "*--define=tflite_with_xnnpack=false*" be generated. The XNNPACK enabled *libtensorflowlite.so* is only used when exploring the impact of XNNAPCK acceleration on the real-time features of DNN inferences. All other benchmarks do not enable XNNPACK acceleration.

4.2 Image Classification

In the field *Image Classification*, we first utilized pre-trained models with different architectures to explore the impact of DNN architecture on the real-time features of inference. Then we analyzed the influence of Quantization technique on inference time, accuracy, and memory usage by comparing full precision and quantized models for some architectures. Finally, we enabled XNNPACK to speed up inference, and discussed the effect of XNNPACK on real-time features by comparing the benchmark results before and after enabling it. In order to realize these experiments, next we will describe the implementation of our benchmarks in the field *Image Classification* from three aspects: model preparation, dataset preparation and details of program construction.

4.2.1 Model Preparation

All pre-trained DNN models for *Image Classification*, including models with full precision (fp32) and models after quantization (uint8), come from Google[1], TensorFlow[2] and IREE[3]. Some of them are provided as TensorFlow Lite models, and the others are initially released as TensorFlow models. Therefore, we converted the latter into TF Lite format using "TensorFlow Lite Converter" by ourselves. Due to various sources, these models differ in the pre-processing methods of input images and the used labels. Specific information about the models for *Image Classification* used in this paper, including train/validation dataset, image pre-processing, download address, etc., are listed in Appendix A. The most important information is extracted in Table 4.1. To distinguish models with different precision, the names of quantized models are appended with "*q*" at the end. All models in the field *Image Classification* can be divided into five categories: VGG family, Inception family, ResNet family, MobileNet family and EfficientNet family. Among them, the VGG family [24] is presented as a representative of early DNNs. The Inception family [14, 25–27] and the ResNet family [10, 11] represent the classic DNNs that are mostly used on workstations/servers, while the Mobilenet family [12, 13, 23] and EfficientNet family [28, 29] are good examples of DNNs designed for edge devices. In order to exclude the impact of the resolution of input images on the benchmark as much as possible, except for `inception_v3`, `inception_v3_q`, `inception_v4`, `inception_v4_q` and `inception_resnet_v2` (the input size for these four models is $299x299x3$), the input size required for all other models is $224x224x3$. For this reason, we only selected the baseline network `efficientnet_b0` from EfficientNet family.

[1] https://tfhub.dev/google, accessed on Dec 04, 2022
[2] https://tfhub.dev/tensorflow, accessed on Dec 04, 2022
[3] https://tfhub.dev/iree, accessed on Dec 04, 2022

At last, the *alpha* (width multiplier) of all members from MobileNet family is 1.0 (Baseline MobileNet).

Table 4.1: TensorFlow Lite models for Image Classification.

TF Lite Models	Architecture	Size (MB)	Input Size	Quantized
vgg16	VGG, 16 layers	553.4	224x224x3	no
vgg19	VGG, 19 layers	574.7	224x224x3	no
inception_v1	Inception V1	26.6	224x224x3	no
inception_v1_q	Inception V1	6.7	224x224x3	yes
inception_v2	Inception V2	44.8	224x224x3	no
inception_v2_q	Inception V2	11.3	224x224x3	yes
inception_v3	Inception V3	95.3	299x299x3	no
inception_v3_q	Inception V3	23.9	299x299x3	yes
inception_v4	Inception V4	170.7	299x299x3	no
inception_v4_q	Inception V4	42.9	299x299x3	yes
inception_resnet_v2	Inception-ResNet V2	121.0	299x299x3	no
resnet50_v1	ResNet V1, 50 layers	102.2	224x224x3	no
resnet101_v1	ResNet V1, 101 layers	178.1	224x224x3	no
resnet152_v1	ResNet V1, 152 layers	240.7	224x224x3	no
resnet50_v2	ResNet V2, 50 layers	102.3	224x224x3	no
resnet101_v2	ResNet V2, 101 layers	178.4	224x224x3	no
resnet152_v2	ResNet V2, 152 layers	241.1	224x224x3	no
mobilenet_v1	MobileNet V1, alpha = 1.0	16.9	224x224x3	no
mobilenet_v1_q	MobileNet V1, alpha = 1.0	4.3	224x224x3	yes
mobilenet_v2	MobileNet V2, alpha = 1.0	14.0	224x224x3	no
mobilenet_v2_q	MobileNet V2, alpha = 1.0	3.6	224x224x3	yes
mobilenet_v3	MobileNet V3 Large, alpha = 1.0	21.9	224x224x3	no
mobilenet_v3_q	MobileNet V3 Large, alpha = 1.0	5.6	224x224x3	yes
efficientnet_b0	EfficientNet B0	18.6	224x224x3	no
efficientnet_b0_q	EfficientNet B0	5.4	224x224x3	yes

4.2.2 Dataset Preparation

Considering the limited storage space of our benchmark device (Coral Dev Board), we selected the first 1000 images from the validation dataset of ILSVRC2012 [22] for benchmark. This means, the RT-Bench function *benchmark_execution* will process different images in each iteration, and be executed 1000 times periodically for all models in the field *Image Claasification*. Since all models expect the input image to have 3 channels (RGB/BGR), we replaced 18 images with only one channel from the first 1000 images with the last 18 images with three channels in the validation data of ILSVRC2012. The ground truth is also changed accordingly. Since the size of these images ranges from 32.5kB to 24.0MB, the resize process of different images varies greatly. In order to avoid the impact of this factor on benchmarks, we pre-resized all the images in this dataset to $224x224x3$ and $299x299x3$ according to the needs of models' input size. This measure is particularly important for the analysis of distribution of metrics after XNNPACK acceleration is enabled in Section 5.3.3.

4.2.3 Details of Program Construction

In addition to those mentioned in Section 4.1, the benchmark constructed for *Image Classification* has some specific details. First, *benchmark_init* needs to prepare a csv file in advance

for recording classification results, and needs to read the ground truth of all images to be classified. Secondly, since we have resized all images in advance, *benchmark_execution* does not need to complete this step during image pre-processing; but when post-processing the output tensor, this function needs to extract the classification result with the highest confidence and match it with the labels; then it needs to evaluate the correctness of classification by comparing with the ground truth and record all results in the csv file. Finally, *benchmark_teardown* needs to close the csv file.

4.3 Object Detection

In the field *Object Detection*, we would like to explore the impact of the specific structure of YOLOv3-tiny on its real-time performance. The first element to be considered is the depth (number of layers) of this model. In order to obtain YOLOv3-tiny models with different depths, we need to modify its original structure. Therefore, unlike direct using pre-trained models in the field *Image Classification*, here we need to train the modified YOLOv3-tiny models by ourselves. The work in this section is based on the repository https://github.com/zzh8829/yolov3-tf2. It used Keras to build YOLOv3 and YOLOv3-tiny, and provided the method to use Pascal VOC2012 Dataset [8] to train models and the method to convert the trained models into TF Lite models. The software environments for modification, training and conversion of models are listed in Table 4.2.

Table 4.2: Software Environments for Modification, Training and Conversion of YOLOv3-tiny.

Software	Version
Ubuntu	20.04.4 LTS
pip	22.1.2
python	3.7.0
tensorflow	2.10.0
tensorflow-gpu	2.5.1
cudnn	7.6.5
cudatoolkit	10.1.243
opencv	3.4.2
matplotlib	3.5.2
lxml	4.9.1
tqdm	4.64.1

4.3.1 Model Preparation

The structure of YOLOv3-tiny is shown in Figure 4.1. In order to obtain YOLOv3-tiny models with different depths and minimize the impact of structural modifications other than the depth, the most direct way is to only modify the backbone of YOLOv3-tiny, Darknet-tiny. We changed the depth of the model by duplicating and adding the *Convolutional* modules in Darknet-tiny, and the duplicated and added *Convolutional* modules must be placed next to the original *Convolutional* module. In this way, the added convolution layer has the same size and similar position as the original, which further reduces the changes except the depth. Since we hope that the impact of the modification can cover the two detection branches of YOLOv3-tiny, the duplicated and added objects only involve **Conv. Layer 1 ~ Conv. Layer 5**.

In addition, we also limited the growth of the depth to 0 ∼ 3 layers. Based on the above rules, we can obtain 56 models through different combinations between **Conv. Layer 1** ∼ **Conv. Layer 5**. They are listed in Table 4.3[4] together with the pre-trained[5] unmodified YOLOv3-tiny model. More detailed information about the models used in the field *Object Detection* is listed in Appendix B.
We use the "XXXXX" five-bit code to distinguish modified YOLOv3-tiny models. From right to left, each coded bit corresponds to the module **Conv. Layer 5** ∼ **Conv. Layer 1**. The number "X" on a certain bit means that we duplicated and added its corresponding *Convolutional* module X times. For example, "10101" means that we duplicated and added **Conv. Layer 1**, **Conv. Layer 3** and **Conv. Layer 5** respectively, and each added *Convolutional* module is placed next to the original *Convolutional* module. The specific structure of the model "10101" is shown in Figure 4.2. Further, "02010" means that we duplicated and added **Conv. Layer 2** twice, **Conv. Layer 4** once, and "00000" means that we did not modify the structure of YOLOv3-tiny, but trained it by ourselves without using the pre-trained weights.

4.3.2 Model Training

The referenced repository provides us the method to train YOLOv3/YOLOv3-tiny models using Pascal VOC2012 Dataset. It first converts the original data into TFRecord format, which is a binary format for efficiently encoding long sequences of TensorFlow. TFRecord can help us store datasets efficiently, and we can get faster I/O speed than reading raw data from disk. Therefore, we used the training method provided by this repository. Since we changed the structure of the backbone of YOLOv3-tiny, we cannot use pre-trained weights for transfer learning, but have to train the models from scratch. In this case, the Pascal VOC2012 Dataset (20 classes, 11540 images for train/validation) is too small to allow our models to achieve acceptable accuracy. The COCO2017 Dataset [16] (80 classes, 118287 images for train, 5000 images for validation) can meet our needs, but it is not easy to use compared with Pascal VOC2012 Dataset. Therefore, in order to take advantage of the method provided by the referenced repository, we first converted the COCO2017 Dataset into PASCAL VOC format, and then converted it to TFRecord format for training. What needs to be added is that, we deleted the images that don't contain any objects of the 80 classes used by COCO2017 in the train and validation dataset before conversion.
We used NVIDIA A100 to train all models except the model "`pre-trained`". All training related parameters and strategies and Non-Maximum Suppression (NMS) related parameters are listed in Table 4.4. Due to the large number of models and limited time, each model was only trained for 12 Epochs. During the training process, we observed that after 10 Epochs, train loss and validation loss for all models decreased slowly; so this choice is reasonable. Since transfer learning cannot be used and the ILSVRC2012 Dataset is not used to train the backbone of models for extraction of image features in advance, the confidence (score) of the object detection results of the trained models is low. Therefore, we chose a small score threshold to prevent NMS from filtering out valuable boxes. After training, we converted all models including "`pre-trained`" to TF Lite models through TensorFlow Lite Converter for benchmark.

[4]This table lists the TF Lite version of the models described in Section 4.3.1.
[5]Pre-trained weights come from https://pjreddie.com/media/files/yolov3-tiny.weights.

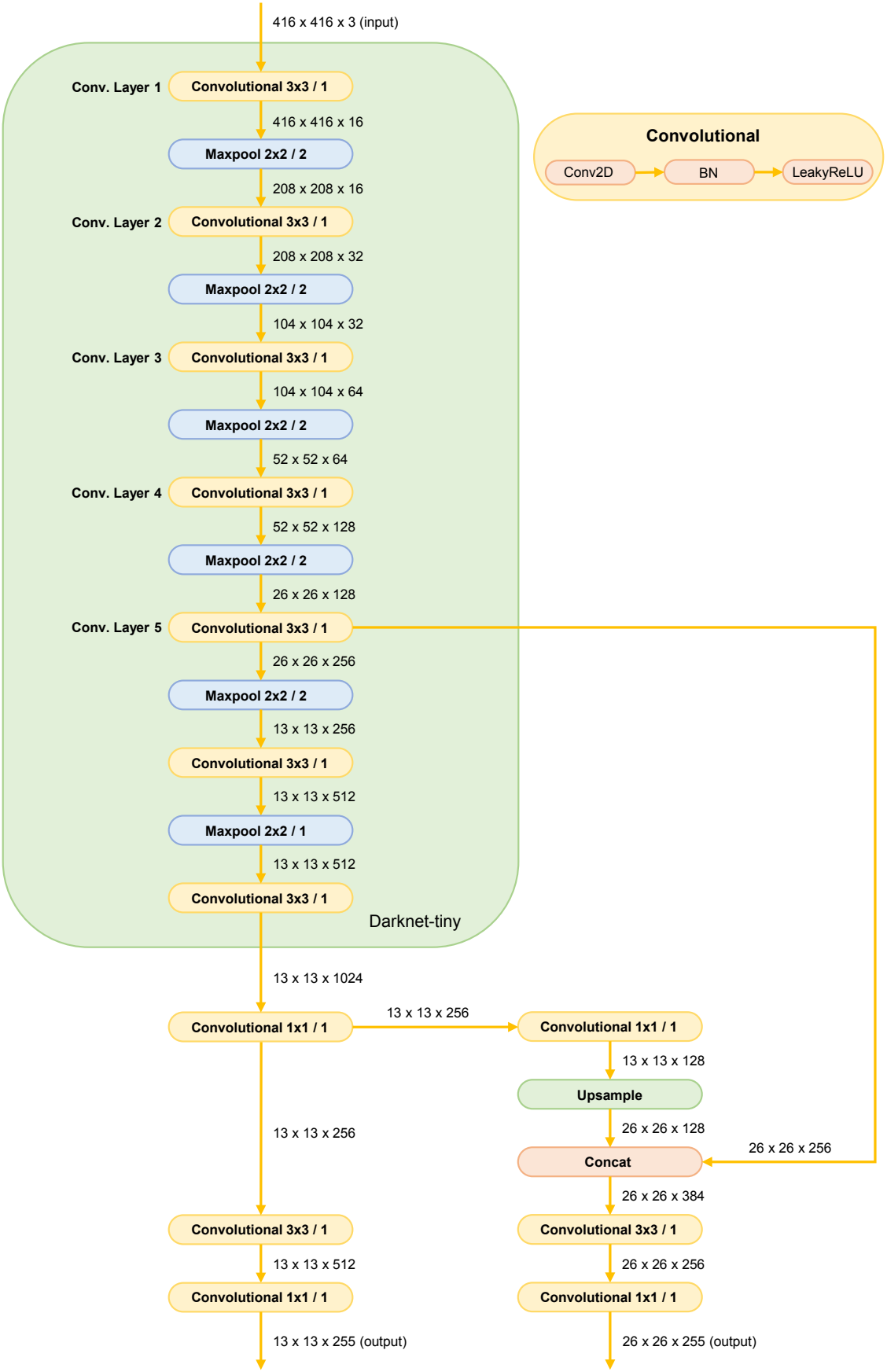

Figure 4.1: This figure shows the specific structure of YOLOv3-tiny. Among them, each *Convolutional* module is composed of *Conv2D* layer, *BatchNormalization* layer and *LeakyReLU* layer. YOLOv3-tiny uses Darknet-tiny as its backbone for extraction of image features.

Table 4.3: YOLOv3-tiny Models with different Depths.

TF Lite Models	Size (MB)	Duplicated and Added *Convolutional* Modules
`pre-trained`	35.4	pre-trained, unmodified
00000	35.4	unmodified
00001	37.8	1x Conv. Layer 5
00002	40.2	2x Conv. Layer 5
00003	42.5	3x Conv. Layer 5
00010	36.0	1x Conv. Layer 4
00011	38.4	1x Conv. Layer 4, 1x Conv. Layer 5
00012	40.8	1x Conv. Layer 4, 2x Conv. Layer 5
00020	36.6	2x Conv. Layer 4
00021	39.0	2x Conv. Layer 4, 1x Conv. Layer 5
00030	37.2	3x Conv. Layer 4
00100	35.6	1x Conv. Layer 3
00101	37.9	1x Conv. Layer 3, 1x Conv. Layer 5
00102	40.3	1x Conv. Layer 3, 2x Conv. Layer 5
00110	36.2	1x Conv. Layer 3, 1x Conv. Layer 4
00111	38.5	1x Conv. Layer 3, 1x Conv. Layer 4, 1x Conv. Layer 4
00120	36.8	1x Conv. Layer 3, 2x Conv. Layer 4
00200	35.7	2x Conv. Layer 3
00201	38.1	2x Conv. Layer 3, 1x Conv. Layer 5
00210	36.3	2x Conv. Layer 3, 1x Conv. Layer 4
00300	35.9	3x Conv. Layer 3
01000	35.5	1x Conv. Layer 2
01001	37.8	1x Conv. Layer 2, 1x Conv. Layer 5
01002	40.2	1x Conv. Layer 2, 2x Conv. Layer 5
01010	36.1	1x Conv. Layer 2, 1x Conv. Layer 4
01011	38.4	1x Conv. Layer 2, 1x Conv. Layer 4, 1x Conv. Layer 5
01020	36.7	1x Conv. Layer 2, 2x Conv. Layer 4
01100	35.6	1x Conv. Layer 2, 1x Conv. Layer 3
01101	38.0	1x Conv. Layer 2, 1x Conv. Layer 3, 1x Conv. Layer 5
01110	36.2	1x Conv. Layer 2, 1x Conv. Layer 3, 1x Conv. Layer 4
01200	35.8	1x Conv. Layer 2, 2x Conv. Layer 3
02000	35.5	2x Conv. Layer 2
02001	37.9	2x Conv. Layer 2, 1x Conv. Layer 5
02010	36.1	2x Conv. Layer 2, 1x Conv. Layer 4
02100	35.7	2x Conv. Layer 2, 1x Conv. Layer 3
03000	35.6	3x Conv. Layer 2
10000	35.4	1x Conv. Layer 1
10001	37.8	1x Conv. Layer 1, 1x Conv. Layer 5
10002	40.2	1x Conv. Layer 1, 2x Conv. Layer 5
10010	36.0	1x Conv. Layer 1, 1x Conv. Layer 4
10011	38.4	1x Conv. Layer 1, 1x Conv. Layer 4, 1x Conv. Layer 5
10020	36.6	1x Conv. Layer 1, 2x Conv. Layer 4
10100	35.6	1x Conv. Layer 1, 1x Conv. Layer 3
10101	38.0	1x Conv. Layer 1, 1x Conv. Layer 3, 1x Conv. Layer 5
10110	36.2	1x Conv. Layer 1, 1x Conv. Layer 3, 1x Conv. Layer 4
10200	35.7	1x Conv. Layer 1, 2x Conv. Layer 3
11000	35.5	1x Conv. Layer 1, 1x Conv. Layer 2
11001	37.8	1x Conv. Layer 1, 1x Conv. Layer 2, 1x Conv. Layer 5
11010	36.1	1x Conv. Layer 1, 1x Conv. Layer 2, 1x Conv. Layer 4
11100	35.6	1x Conv. Layer 1, 1x Conv. Layer 2, 1x Conv. Layer 3
12000	35.5	1x Conv. Layer 1, 2x Conv. Layer 2
20000	35.5	2x Conv. Layer 1
20001	37.8	2x Conv. Layer 1, 1x Conv. Layer 5
20010	36.0	2x Conv. Layer 1, 1x Conv. Layer 4
20100	35.6	2x Conv. Layer 1, 1x Conv. Layer 3
21000	35.5	2x Conv. Layer 1, 1x Conv. Layer 2
30000	35.5	3x Conv. Layer 1

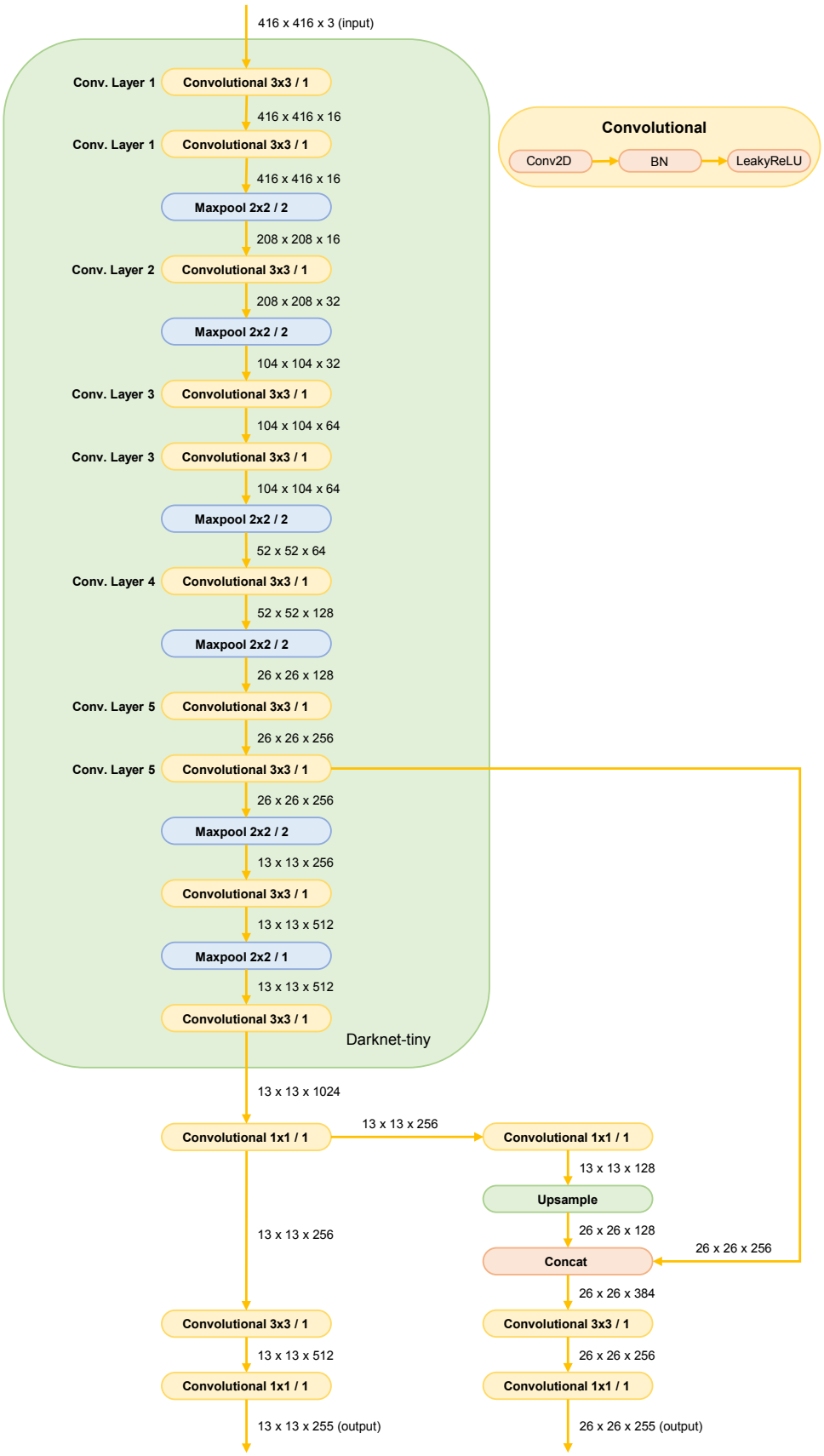

Figure 4.2: This figure shows the specific structure of the model "10101". **Conv. Layer 1**, **Conv. Layer 3** and **Conv. Layer 5** are each duplicated and added once. The added modules are placed next to their original.

Table 4.4: Parameters and Strategies for Model Training and NMS.

Parameters and Strategies	Value
Learning Rate	0.0007
L2 Regularization	0.0005
Batch Size	32
Epochs	12
EarlyStopping	patience=3
ReduceLROnPlateau	patience=1, factor=0.2, monitor='val_loss'
Maximum Number of Boxes per Image	100
IOU Threshold	0.5
Score Threshold	0.001

4.3.3 Dataset Preparation

We used the images from the validation dataset of COCO2017 for our benchmark. As described in Section 4.3.2, we removed the images that don't contain any objects of the 80 classes used by COCO2017. Furthermore, same as the method used to process the dataset for *Image Classification*, we also only kept the images with three channels (RGB) here. In the end, 4942 images were used for benchamrk in total, and the annotations (ground truth) were also filtered accordingly. Since we want to further explore the distribution of the metrics when YOLOv3-tiny models process different images, we also pre-resized all images to $416x416x3$ to avoid the impact of the differences in resizing images with different sizes on the distribution. Because the true positions of objects in the annotations (ground truth) are based on the size of the unresized images, we also recorded the size of the original images in order to restore the positions of the objects detection boxes in the inference's results.

4.3.4 Details of Program Construction

The construction of the benchmark program for *Object Detection* is basically the same as that described in Section 4.1, and only the different parts are discussed below.
Since we pre-resized all the images used for benchmark, *benchmark_init* needs to read the original size of all images, because *benchmark_execution* needs to use this original size to restore the position of object detection boxes when it post-processes the output tensor of the model. In addition, because the calculation of inference's accuracy is relatively difficult for object detection models, our benchmark only records the inference results of the model, and their correctness are left for another program to solve. Finally, unlike in the benchmark for *image classification*, the program only needs to record a classification result and its confidence each execution; for the benchmark in *Object Detection*, the output of the model includes the class, location and confidence of multiple objects, so the I/O process is expensive. In order to avoid the impact of this process on the metrics of model inference, we invoked the RT-Bench function *benchmark_log_data* to record the results of model inference using csv format. Besides, we also invoked *benchmark_log_header* to add the metric "`num_detections_nms`" to the csv file to record the number of detected objects after NMS.

4.3.5 mAP50 Calculation

We used mAP50 to characterize the accuracy of different YOLOv3-tiny models. In practice, a higher mAP value indicates a higher accuracy of models. The calculation of this metric is done separately after the benchmark, and based on the repository https://github.com/Cartucho/mAP. It calculates mAP according to the criterion defined in the PASCAL VOC 2012 competition. Since in the calculation, the threshold for matching the detection results with the ground truth objects is that IOU should be greater than 50%, so we actually calculated mAP50.

4.4 Program Execution

Different from the content discussed in the previous two sections, there are some measures are shared by the benchmarks in the field *Image Classification* and *Object Detection*. The content stated below are valid for all benchmarks in this paper.
In order to ensure that our benchmark is not interfered by other processes, we limited all processes not related to our benchmarks to core 0 of CORTEX_A53, and only used one of the remaining three cores for benchmarks ("-$t = 1/2/3$" for the option *core affinity* of RT-Bench CLI). Because enabling multi-threads will reduce efficiency in the case of using a single core, we only used a single thread. In order to ensure the stable performance of the processor during testing, we fixed the frequency of CORTEX_A53 at 1500 MHz.
In addition, for other CLI options provided by RT-Bench, we made the values of *period* (-*p*) and *deadline* (-*d*) be equal and slightly larger than the actual inference time (`job_elapsed`) of each model. This can effectively avoid the impact of the current benchmark on the execution of the next period. Meanwhile, *tasks number* (-*t*) should be equal to the number of images in the dataset of *Image Classification* or *Object Detection*. Since we didn't implement memory stress tests, there is no need to set *memory limit* (-*m*). All benchmark data are recorded in csv format, so *log level* (-*l*) should equal to 2. Finally, -*b* is used to transmit parameters related to specific benchmarks, such as models, labels, ground truth, etc.

Chapter 5

Evaluation

In this chapter, we first introduced the hardware platform and software environment for our benchmarks, followed by the evaluations of the obtained results. As in Chapter 4, discussions are divided into two categories. In the field *Image Classification*, we first discussed the impact of different DNN architectures on the real-time features of inferences. Then, the influence of Quantization on real-time performance was presented. Finally, we analyzed the effect of XNNPACK on the real-time features of DNN inferences. In the field *Object Detection*, we first discussed the impact of model depth on the real-time performance of YOLOv3-tiny, and then analyzed the distribution of real-time metrics when the models process different images.

5.1 Hardware Platform

All our benchmarks are performed on the Coral Dev Board[1], which is a single-board computer running a derivative of Debian Linux "Mendel". The key technical specifications of the Coral Dev Board are listed in Table 5.1[2]. Exploring the real-time features of deep neural network inferences on CPUs yields the widest reach across the space of edge devices, so our benchmarks did not use the Google Edge TPU coprocessor on the Coral Dev Board, but focused on the CPU Cortex-A53, which is the only supported platform with PMC (Performance Monitor Counter) enabled by RT-Bench.
As mentioned in Section 4.4, all benchmarks were done on core 1 of CORTEX_A53, and all processes not related to benchmarks were restricted to core 0. In addition, CPU Performance Scaling via the CPUFreq subsystem enables the Linux kernel to scale the CPU frequency up or down in order to save power or improve performance[3]. We used the Governor “performance” to fix the CPU frequency to its maximum frequency to ensure that the performance of CORTEX_A53 is stable during benchmark.

[1]https://coral.ai/products/dev-board, accessed on Nov 30, 2022
[2]https://coral.ai/docs/dev-board/datasheet/, accessed on Dec 04, 2022
[3]https://wiki.archlinux.org/title/CPU_frequency_scaling, accessed on Dec 07, 2022

Table 5.1: Key Technical Specifications of the Coral Dev Board.

Components	Details
Cortex-A53 processor	64-bit Armv8-A Architecture 4x Symmetrical Multiprocessing (SMP) within a single processor cluster 32 KB L1 Instruction Cache 32 KB L1 Data Cache 1 MB unified L2 Cache (LLC) 64 Bytes Cache Line for L1/L2 Cache 500 MHz ∼ 1500 MHz Frequency
Edge TPU coprocessor	ASIC designed by Google for high-speed inferences of TF Lite models 8 MB SRAM Using PCIe and I2C/GPIO to interface with the iMX 8M SoC 4 TOPS (int8) and 2 TOPS per watt
Random access memory	4 GB LPDDR4 SDRAM (4-channel, 32-bit bus width) 1600 MHz maximum DDR clock Interfaces directly to the iMX 8M build-in DDR controller
Operation System	Mendel Linux (derivative of Debian)

5.2 Software Environment

The software environment on which the compilation, execution and data processing of our benchmarks depend is listed in Table 5.2. As mentioned in Section 4.1, we built the shared library *libtensorflowlite.so* using Bazal in advance, then used GNU Make to compile the benchmarks and RT-Bench[4] by dynamically linking the library. FlatBuffers is needed by TF Lite to process TF Lite model, and is contained by *libtensorflowlite.so*.

Table 5.2: Software Environment for Benchmark

Software	Version
Linux	Mendel GNU/Linux
python	2.7.16, 3.7.3
pip/pip3	9.0.1
numpy	1.19.5
pandas	0.22.0
matplotlib	3.3.4
gcc/g++	8.3.0
C++	C++14
Bazel	5.1.1
TensorFlow/TF Lite	Commit Hash: fbff9fd15e4241e7e2beedf9fbaae0c73e0dd66c
FlatBuffers	Commit Hash: b9eea76a86081d5aa8d0ad83921f1f97fa16cf33
RT-Bench	Commit Hash: c5ae6e2f55c9ad6ba7034a2ba78b2690053eed95

[4]https://rt-bench.gitlab.io/rt-bench/compilation.html, accessed on Dec 07, 2022

5.3 Image Classification

In this section, we will sequentially analyze the impact of DNN architectures, Quantization and XNNPACK on the real-time features of model inference. In addition, we will discuss the stability of XNNPACK acceleration for different images.

5.3.1 Different DNN Architectures

In this subsection we want to explore the differences between real-time features of different DNN architectures. Let us first analyze two commonly discussed metrics, *Inference Time* and *Top-1 Accuracy*, as shown in Figure 5.1. This figure presents us the performance of full precision models and quantized models at the same time, but in this subsection we only focus on full precision models.

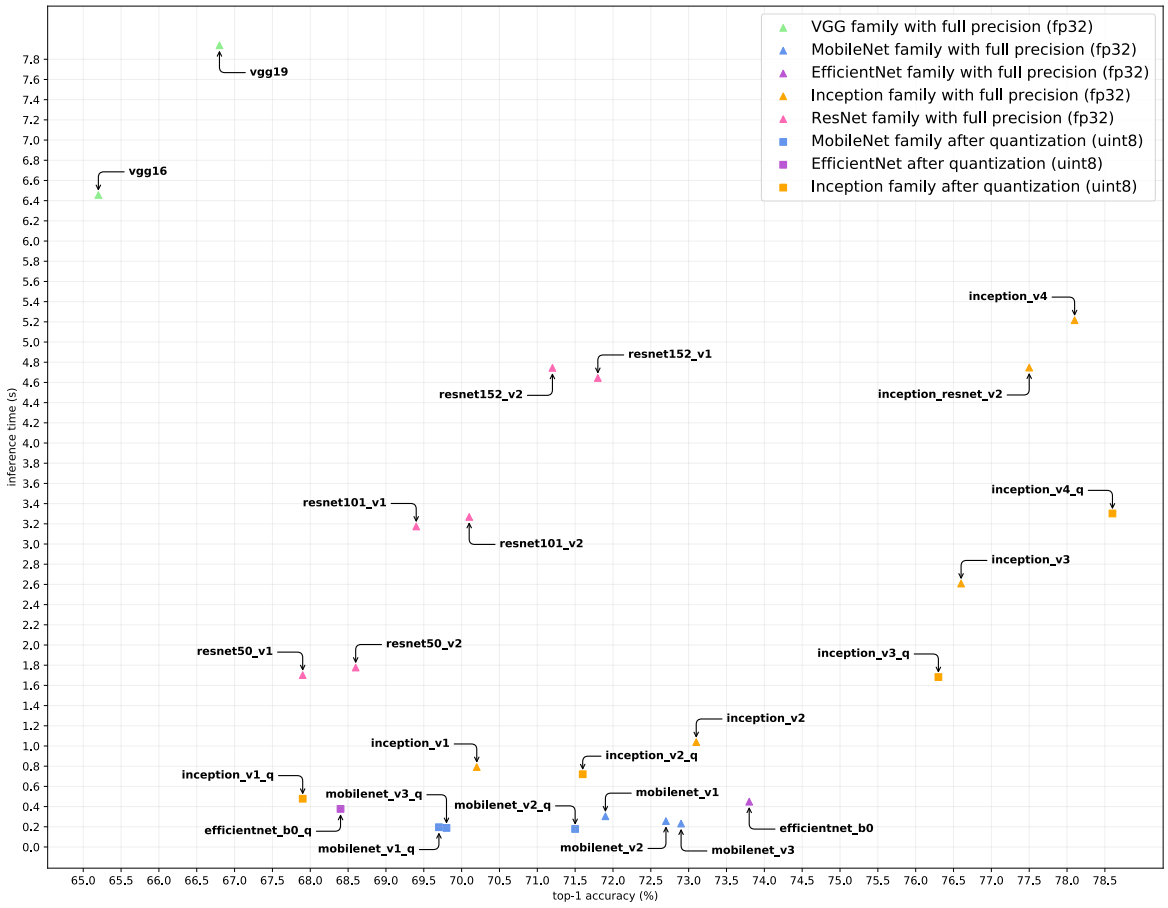

Figure 5.1: Inference time and Top-1 accuracy of full precision (fp32) and quantized (uint8) TensorFlow Lite models for Image Classification.

As a representative of early DNN architectures, VGG family lags behind all other models in terms of inference speed and accuracy. In addition, from Table 4.1 we can also find that the sizes of vgg16 (553.4 MB) and vgg19 (574.7 MB) are much larger than other models.
For Inception family, each iteration of its version[5] improves the accuracy, but also makes the inference time continue to increase. Starting from `inception_v2`, the accuracy of Inception's members precedes almost all other models. Among them, `inception_v4` has the

[5]Both `inception_resnet_v2` and `inception_v4` come from version "v4" of Inception networks.

highest Top-1 accuracy (78.1%) among all full precision models, but also has the longest inference time (5.22 s) except vgg16 (6.46 s) and vgg19 (7.94 s). It should be noted that the accuracy of models will increase as the input size is enlarged within a certain range [28]. This may be one of the reasons why `inception_v3` (76.6%), `inception_v4` (78.1%), and `inception_resnet_v2` (77.5%) have higher accuracy[6].
As the models designed for mobile and embedded vision applications, MobileNet family is extremely superior in the performance of inference time, and its accuracy is in the middle level of all models. Although each update of its version brings improvements in inference speed and accuracy, the magnitude of the changes is far less than that of Inception family.
EfficientNet family is obtained through neural network search [28]. As the baseline network of EfficientNet, `efficientnet_b0` has the same excellent inference speed (0.45 s) as the members of MobileNet family, and its accuracy (73.8%) is higher than all of them.
The performance of ResNet family in this two metrics is relatively mediocre. Although both the versions "v1" and "v2" of ResNet through increasing layers of the models continue to improve the accuracy while sacrificing the inference speed, both the metrics are basically at the middle level of all models. Interestingly, as the number of layers increases, the accuracy improvement of version "v2" is significantly slower than that of version "v1"; even for the ResNet models with 152 layers, the accuracy of the version "v2" (71.2%) is lower than that of version "v1" (71.8%).

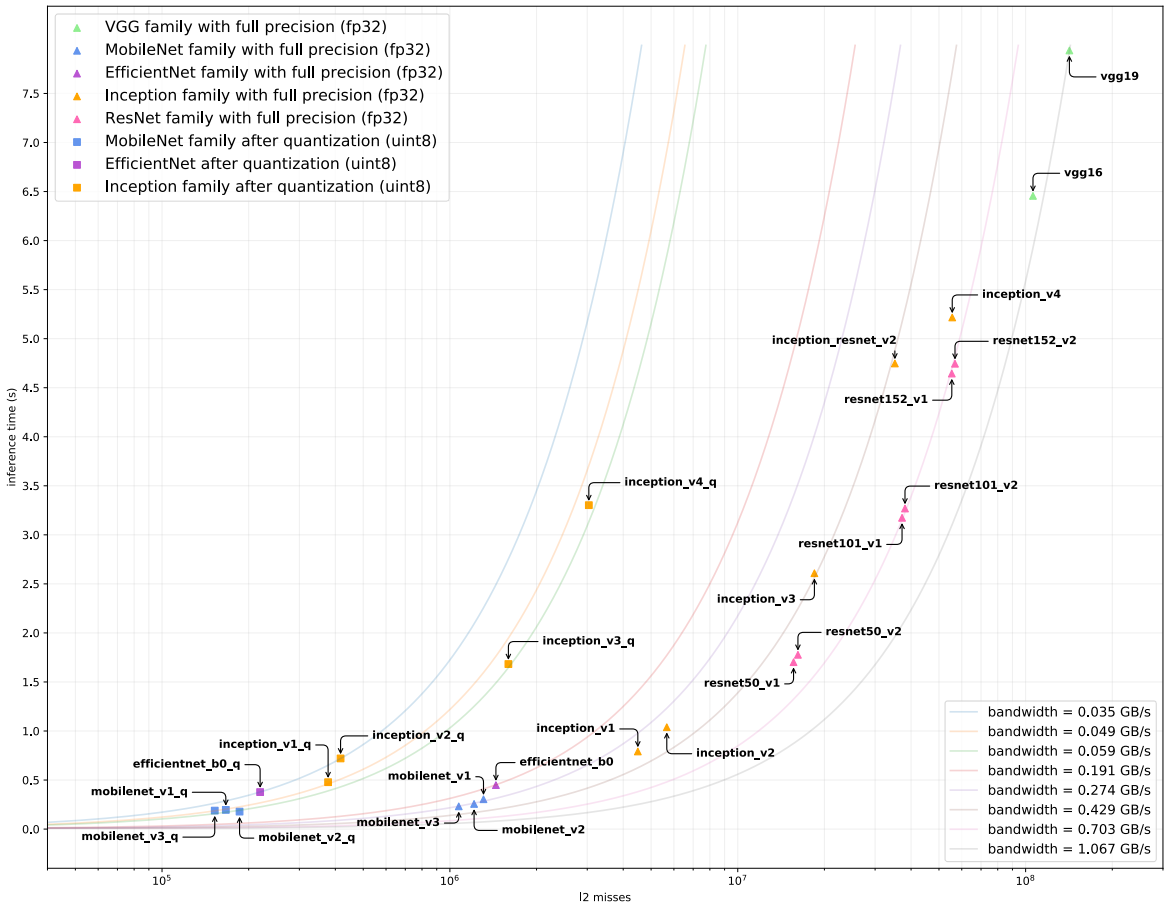

Figure 5.2: Inference time and L2 misses of full precision (fp32) and quantized (uint8) TensorFlow Lite models for Image Classification. The axis of L2 misses is presented as a logarithmic axis.

[6]The input size of `inception_v3`, `inception_v4`, and `inception_resnet_v2` is $299x299x3$, while the input size of all other models is $224x224x3$.

We next turn our attention to a metric that has not been widely explored, *L2 Misses/LLC Misses*. It indicates the number of times an event occurs during model inference, in which the processor makes a request to retrieve data from L2 Cache, but the specific data is not currently in L2. In this case, the processor has to fetch the data from DRAM and store it in the cache, which will take more time than fetching the data directly from the cache. Therefore, *L2 Misses* can reflect the memory usage of models. We plotted *L2 Misses* and *Inference Time* of different models together in Figure 5.2. By dividing *L2 Misses* of a model by its corresponding *Inference Time*, we can obtain the average *Bandwidth*[7] of the model during inference. Different *Bandwidth* ranges are also presented in Figure 5.2. Similarly, in this subsection we only focus on *L2 Misses* and *Bandwidth* of full precision models.
For L2 misses, MobileNet family and `efficientnet_b0` perform very similarly and significantly outperform all other full precision models. Among them, the L2 misses of MobileNet family is slightly lower than that of `efficientnet_b0` (1.45×10^6), and with the iteration of MobileNet's version, the L2 misses continue to decrease slightly. VGG family has the highest L2 misses, which is also an important reason for its longer inference time. For Inception family, each iteration of its version will lead higher L2 misses. This phenomenon is more obvious since `inception_v2` (5.67×10^6), and reaches the peak of the family at `inception_v4` (5.56×10^7). Finally, the L2 misses between the two versions of ResNet are very similar and increase with the number of layers. Among them, `resnet152_v2` has the highest L2 misses (5.68×10^7) of all models except `vgg16` (1.06×10^8) and `vgg19` (1.42×10^8).
For bandwidth, the performance of MobileNet family and `efficientnet_b0` is also close to each other and better than other families. Among them, `efficientnet_b0` has the smallest bandwidth (0.191 GB/s) in all full precision models. For Inception family, `inception_v4` has the largest bandwidth (0.635 GB/s); the bandwidth of `inception_resnet_v2` and `inception_v3` is relatively close and at the medium level in their family (0.429 GB/s); while `inception_v1` and `inception_v2` have relatively small bandwidth, about 0.332 GB/s. The two versions of ResNet are also very close in bandwidth, and generally higher than Inception family. The bandwidth of these models with 101 layers and 152 layers is around 0.703 GB/s, while the bandwidth of the models with 50 layers is relatively small (about 0.545GB/s). Finally, the bandwidth of `vgg16` (0.979 GB/s) and `vgg19` (1.067 GB/s) is higher than all other models.
In this subsection, we only discussed a small part of metrics. All metrics obtained through RT-Bench during benchmark are plotted as bar charts presented in Appendix C. The models involved include both full precision models and quantized models.

5.3.2 Quantization

In this subsection, we will analyze the impact of Quantization on the real-time features of DNN inferences. As mentioned in Section 4.2.1, we distinguish quantized models from full precision models by adding "*p*" at the end of models' name. We benchmarked 8 different quantized models totally, involving MobileNet family, EfficientNet family and Inception family. The average ratios of their inference metrics to those of their corresponding full precision models are listed in Table 5.3.
Since the precision of unquantized models is fp32, and the precision of quantized model is uint8, we can see that the size of all quantized models is about a quarter of their original models. At the same time, quantized model also have faster inference speed; Compared with full precision models, the inference time of all quantized models decrease by about 30% on average, and the performance of `inception_v1_q` (60.40%) in this regard is significant

[7] Cache Line is 64 Bytes, as shown in Table 5.1

Table 5.3: Comparison of Metrics for Quantized Models and Full Precision Models.

Models	Size	Inference Time	Top-1 Accuracy	L2 Misses	Bandwidth
mobilenet_v1_q / mobilenet_v1	25.44%	64.33%	96.94%	12.73%	19.79%
mobilenet_v2_q / mobilenet_v2	25.71%	69.47%	98.35%	15.32%	22.05%
mobilenet_v3_q / mobilenet_v3	25.57%	81.30%	95.75%	14.20%	17.47%
efficientnet_b0_q / efficientnet_b0	29.03%	84.12%	92.68%	15.17%	18.03%
inception_v1_q / inception_v1	25.19%	60.40%	96.72%	8.40%	13.90%
inception_v2_q / inception_v2	25.22%	69.35%	97.95%	7.36%	10.61%
inception_v3_q / inception_v3	25.08%	64.49%	99.61%	8.65%	13.41%
inception_v4_q / inception_v4	25.13%	63.31%	100.64%	5.46%	8.63%

obvious. Furthermore, the reduction in inference time do not lead to an increase in bandwidth; on the contrary, the bandwidth of all quantized models are reduced significantly. The best-performing `inception_v4_q` has only 8.63% of its original bandwidth, and even the worst-performing `mobilenet_v2_q` has only 22.05% of its original bandwidth. The reason for this is that the reduction in L2 misses caused by Quantization is more drastic than the reduction in inference time. For MobileNet family and `efficientnet_b0`, their L2 misses are only about 14.50% of the original; while the performance of Inception family is even better, Quantization reduces their L2 misses by more than 90%.
Because Quantization sacrifices the precision of models, it may reduce the accuracy of model inference, which is confirmed by our benchmark. For MobileNet family, the decrease in Top-1 accuracy is within 5%. The performance of `efficientnet_b0` is slightly worse, whose accuracy is reduced by nearly 10%. However, Quantization has less impact on the accuracy of Inception family. Among the members of Inception, the accuracy of `inception_v3_q` (99.61%) is almost the same as the original; the accuracy of `inception_v4_q` (100.64%) is even slightly higher than `inception_v4`. This phenomenon has also been found in other research, that is, "≥ 8-bit quantization is able to achieve lossless or even better accuracy" after fine tuning [5].
Now we turn our attention to Figure 5.1 and Figure 5.2 again, and consider both full precision models and quantized models. If only a satisfactory tradeoff between processing efficiency and application accuracy is pursued, `efficientnet_b0` significantly outperforms all other models on edge devices. If bandwidth is to be further considered, `mobilenet_v2_q` is a better choice for edge applications.

5.3.3 XNNPACK

In this subsection, we will discuss the impact of XNNPACK on real-time features of DNN inferences. As mentioned in Section 2.1.3 and Section 4.1, XNNPACK cannot be used directly by researchers, but used to accelerate high-level machine learning platforms such as TensorFlow Lite; through using different compilation options, "*libtensorflowlite.so*" with XNNPACK enabled or disabled can be generated. In and only in this subsection, we enabled XNNPACK for benchmark and compared the obtained metrics of DNN inferences with those of inference without XNNPACK. The average ratios of the two are presented in Table 5.4. We characterized the models taking advantage of XNNPACK by adding "*x*" to the end of their names.

Since XNNPACK does not modify the model itself, there will be no change in the model size nor inference accuracy with XNNPACK enabled. The original intention of XNNPACK is to accelerate the inference of neural networks, therefore we can find that after enabling XNNPACK, the inference time of all models decrease by about 25%. Among all models, the

Table 5.4: Metric Comparison of XNNPACK Enabled Inference and Inference without XNNPACK.

Models	Retired Instructions	Inference Time	L2 Misses	Bandwidth	Instructions per Clock (IPC)
vgg16_x / vgg16	97.92%	75.40%	107.79%	142.97%	129.88%
vgg19_x / vgg19	98.38%	77.60%	111.49%	143.67%	126.78%
mobilenet_v1_x / mobilenet_v1	95.82%	73.36%	163.80%	223.28%	130.63%
mobilenet_v2_x / mobilenet_v2	93.32%	68.81%	101.27%	147.19%	135.64%
mobilenet_v3_x / mobilenet_v3	91.57%	77.52%	95.49%	123.17%	118.10%
efficientnet_b0_x / efficientnet_b0	94.50%	82.66%	105.46%	127.59%	115.95%
inception_v1_x / inception_v1	95.44%	73.08%	76.53%	104.72%	130.58%
inception_v2_x / inception_v2	93.62%	68.13%	87.39%	128.28%	137.41%
inception_v3_x / inception_v3	95.90%	72.24%	80.07%	110.85%	132.75%
inception_v4_x / inception_v4	96.59%	72.83%	105.01%	144.18%	132.62%
inception_resnet_v2_x / inception_resnet_v2	97.14%	70.12%	118.47%	168.95%	138.63%
resnet50_v1_x / resnet50_v1	96.04%	76.64%	122.89%	160.34%	125.31%
resnet101_v1_x / resnet101_v1	96.75%	76.06%	119.80%	157.52%	127.19%
resnet152_v1_x / resnet152_v1	97.09%	74.46%	116.47%	156.43%	130.38%
resnet50_v2_x / resnet50_v2	96.38%	77.97%	121.95%	156.41%	123.61%
resnet101_v2_x / resnet101_v2	96.98%	76.85%	119.51%	155.50%	126.18%
resnet152_v2_x / resnet152_v2	97.29%	75.22%	116.04%	154.27%	129.33%

inference time of `inception_v2_x` has the largest reduction, which is only 68.13% of that when XNNPACK is not enabled. The improvement of inference speed comes from two aspects. One is the slight reduction of retired instructions after enabling XNNPACK; for all models, the decrease of retired instructions is within 10%. Another reason for the decline of inference time is the increase of Instructions per Clock (IPC) after XNNPACK is enabled. The IPC of all models is improved by at least 15% with the help of XNNPACK, and the improvement of `inception_resnet_v2_x` even reaches 38.63% in this regard.

The impact of XNNPACK on L2 misses varies for different models. After enabling XNNPACK, most models have different degrees of increase in their L2 misses; among them, the increase of `mobilenet_v1_x` is largest (163.80%). In contrast, the L2 misses of `mobilenet_v3_x` (95.49%), `inception_v1_x` (76.53%), `inception_v2_x` (87.39%) and `inception_v3_x` (80.07%) are lower than their original.

Unlike the inconsistent impact of XNNPACK on L2 misses of different models, XNNPACK brings the deterioration of bandwidth for all models. Especially for those models whose inference time is reduced and L2 misses is increased after enabling XNNPACK, their bandwidth increase greatly. For example, the inference time and L2 misses of `mobilenet_v1_x` are 73.36% and 163.80% of the original respectively, making its bandwidth increase by 123.28%. For `inception_v1_x`, although its L2 misses has the largest reduction among all models, XNNPACK increases its inference speed more drastically, therefore its bandwidth is slightly deteriorated (104.72%).

The above metrics and others not discussed are plotted as bar charts presented in Appendix D. They reflect the ratio of each metric with and without XNNPACK enabled.

In addition to evaluating the average impact of XNNPACK on each metric, we further explored the stability of XNNPACK in accelerating inference. To do this, we first obtained the inference time of all executions for each model with XNNPACK enabled, and then divided all inference time of each model by the corresponding average inference time without XNNPACK enabled to obtain the speedup for each execution. The distribution of obtained speedup of each execution for each model is presented in Figure 5.3.

In this figure, we also identified the positions of extremes, median, 99% quantile, and 1% quantile for each model. We can find that the positions of 99% quantile and 1% quantile of each model are very close to the position of the corresponding median, and all deviations do not exceed 2%. This indicats that almost all executions of each model achieve consistent speedup. Therefore, although the degree of acceleration that XNNPACK can provide varies from model to model, for the same model, it can provide stable acceleration when processing

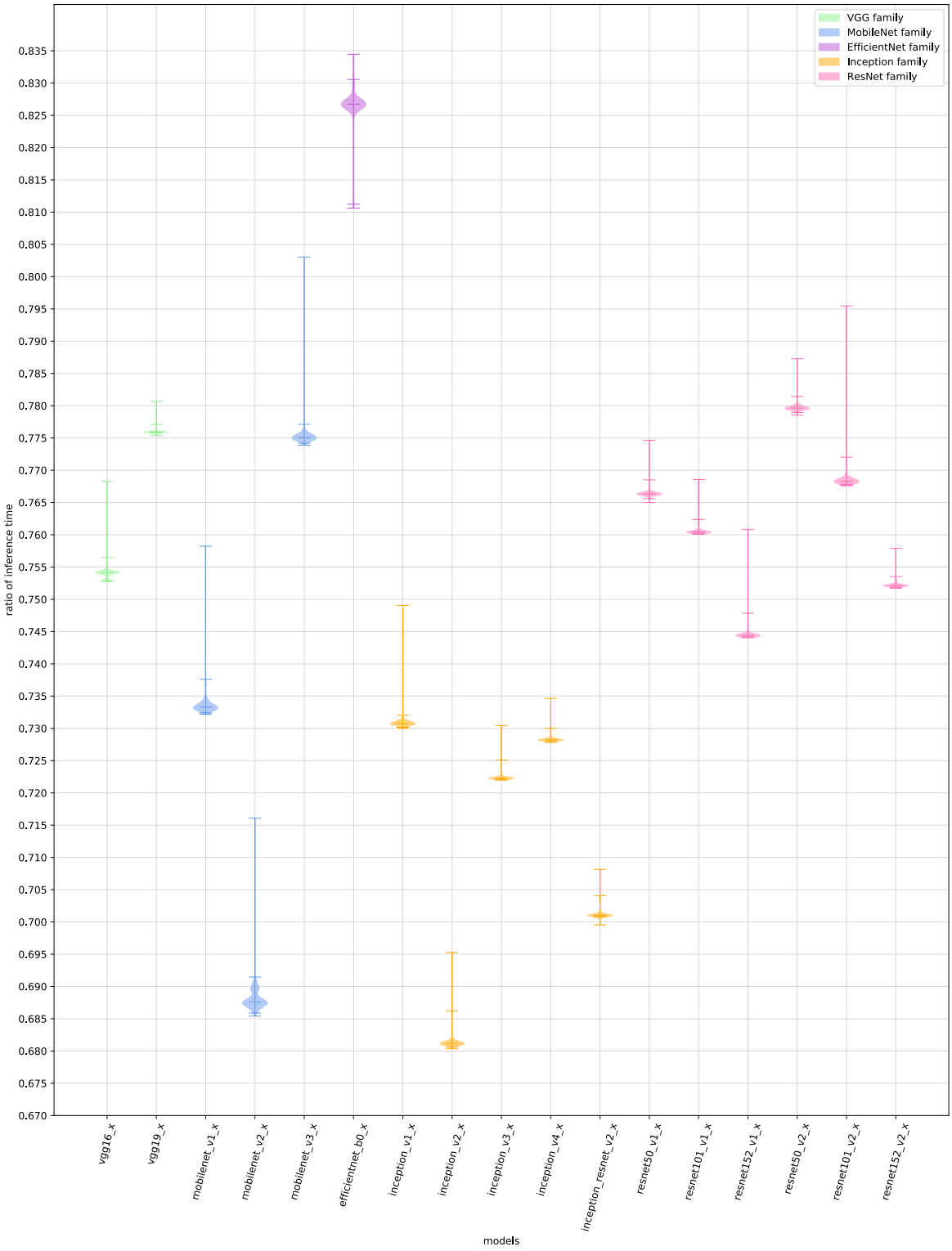

Figure 5.3: This figure reflects the ratio of the inference time of each execution for each model after XNNPACK is enabled to the corresponding average inference time without XNNPACK. The distribution of these ratios reflects the stability of XNNPACK in accelerating inference.

different images.
In addition, we also observed that the maxima of almost all models deviates significantly from the corresponding median. This phenomenon also occured when we analyzed the inference stability of YOLOv3-tiny models. Therefore, we will discuss the reasons for this phenomenon in Section 5.4.2.

5.4 Object Detection

In this section, we will first discuss the impact of the depth on the real-time features of YOLOv3-tiny inferences and mainly focus on the influence of different *Convolutional* modules of Darknet-tiny. Then we will further analyze the distribution of metrics, in order to evaluate the stability of the inferences of YOLOv3-tiny models.
As mentioned in Section 4.3.1, we increased the depth of YOLOv3-tiny by duplicating and adding the *Convolutional* modules (**Conv. Layer 1** $\sim$ **Conv. Layer 5**) in Darknet-tiny, and limited the depth growth to $0 \sim 3$ layers; in addition, we differentiated the obtained 56 variants of YOLOv3-tiny by the five-bit code "XXXXX". In order to more clearly reveal the influence of different *Convolutional* modules, we grouped all obtained models and the pre-trained, unmodified YOLOv3-tiny model. All analyzes below involve two different groupings.
Group one:

- pre-trained: We did not modify its structure, and loaded the pre-trained weights for it.
- `00000`: We did not modify its structure, but trained it from scratch.
- `0000X`: We start duplicating and adding **Conv. Layer 5**.
- `000X*`: We start duplicating and adding **Conv. Layer 4**.
- `00X**`: We start duplicating and adding **Conv. Layer 3**.
- `0X***`: We start duplicating and adding **Conv. Layer 2**.
- `X****`: We start duplicating and adding **Conv. Layer 1**.

Group two:

- pre-trained: We did not modify its structure, and loaded the pre-trained weights for it.
- `00000`: We did not modify its structure, but trained it from scratch.
- `0000X`: We only duplicated and added **Conv. Layer 5**.
- `000X0`: We only duplicated and added **Conv. Layer 4**.
- `00X00`: We only duplicated and added **Conv. Layer 3**.
- `0X000`: We only duplicated and added **Conv. Layer 2**.
- `X0000`: We only duplicated and added **Conv. Layer 1**.
- `*****`: Other models.

5.4.1 YOLOv3-tiny with Different Depths

Deep neural networks often give people such a misconception that the larger the size of models, the longer the inference time. Our benchmark in the field *Image Classification* seems to support this statement, but in fact, the size of models does not determine the required inference time. In Figure 5.4, we arranged all YOLOv3-tiny models on the X-axis in order of their size from small to large, while the Y-coordinate reflects their inference time. We can find that the inference time does not present an increasing trend.This reflects that there is no clear relationship between size of models and inference time.

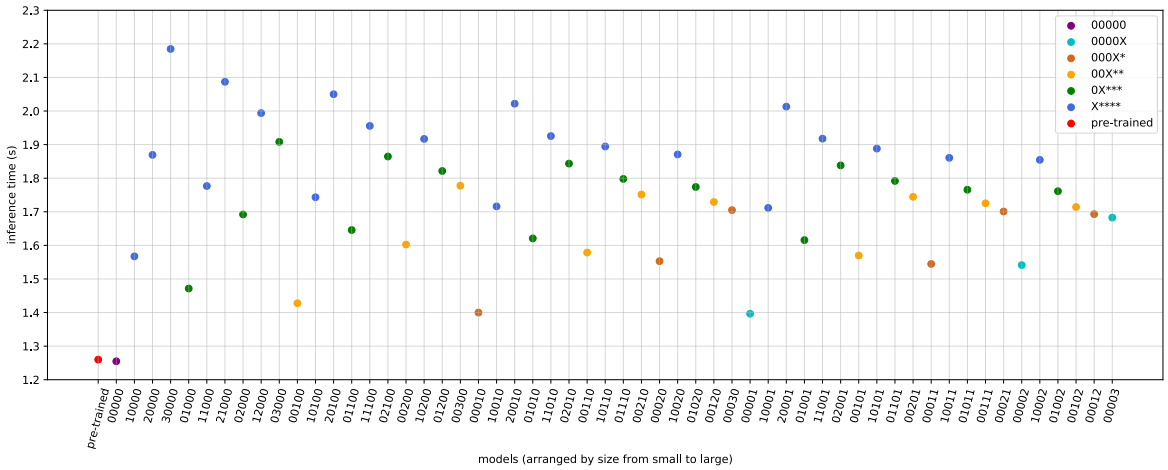

Figure 5.4: Inference time of different YOLOv3-tiny models, which are arranged by size from small to large. The inference time does not present a clear trend.

The real important factor affecting the inference time is the retired instructions, which is clearly reflected in Figure 5.5. In this figure, we re-arranged all YOLOv3-tiny models on the X-axis according to their retired instructions from small to large, and we can find that their inference time generally presents an upward trend. This proves that the more retired instructions a model has, the more time is required for its inference.

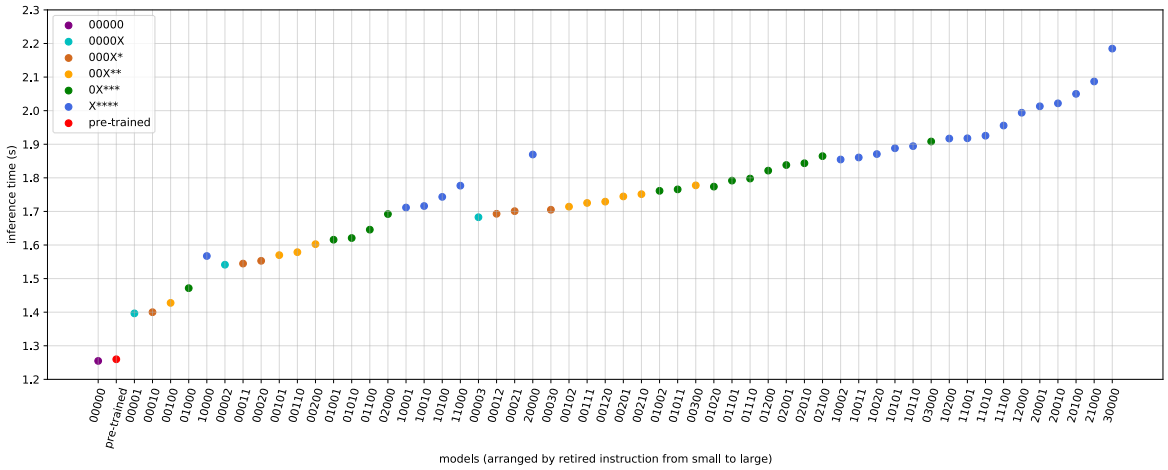

Figure 5.5: Inference time of different YOLOv3-tiny models, which are arranged by retired instructions from small to large. The inference time generally presents an upward trend, but there are exceptions.

By comparing the arrangement of models on the X-axis in Figure 5.4 and Figure 5.5, we can further find that there is no strong correlation between the size and the retired instructions

of models. The arrangement of YOLOv3-tiny models in Figure 5.4 reflects that, in addition to the number of increased layers ($0 \sim 3$), different *Convolutional* modules also affect the size of models to varying degrees. From **Conv. Layer 1** to **Conv. Layer 5**, their influence on the increase of model size is continuously enhanced. This means that adding **Conv. Layer 5** will increase the size of models more than adding other *Convolutional* modules in Darknet-tiny. However, from the arrangement of models according to the retired instructions, we find the opposite phenomenon: from **Conv. Layer 1** to **Conv. Layer 5**, their effect on the increase of retired instructions continue to decline; **Conv. Layer 1** contributes more retired instructions than other *Convolutional* modules. Figure E.1 and Figure E.2 in Appendix E reflect the same facts.

This phenomenon is not surprising. In addition to the structural information of DNN, the weights obtained by training occupy most of model size. From **Conv. Layer 1** to **Conv. Layer 5** in Darknet-tiny, deeper *Convolutional* modules have more convolution kernels than shallower *Convolutional* modules; for example, the number of convolution kernels of **Conv. Layer 5** is 256, while **Conv. Layer 1** has only 16 convolution kernels. Therefore, in the case of same size of convolution kernel, **Conv. Layer 5** has 16 times as many weights as **Conv. Layer 1**. Since deeper *Convolutional* modules in Darknet-tiny have more weights, they also contribute more model size. In contrast, the input's resolution of shallower *Convolutional* modules is larger than the input's resolution of deeper *Convolutional* modules; for example, **Conv. Layer 1** needs to process an input of resolution 416x416, while the input's resolution of **Conv. Layer 1** is only 26x26. Larger input's resolution will cause more retired instructions. Therefore, shallower *Convolutional* modules in Darknet-tiny contribute more retired instructions than deeper *Convolutional* modules.

From Figure 5.5, we can also find that the inference time of some models does not conform to the overall upward trend, such as the model 10000, 00003, 20000, etc. This reflects that in addition to the retired instructions, there is another factor that also affect the inference time of models. In order to reveal this phenomenon more clearly, in Figure 5.6, we used the ratio of the inference time of each model to the corresponding retired instructions as the Y-coordinate, and the models were arranged on the X-axis according to their configuration (type and number of *Convolutional* modules duplicated and added). If the inference time is only determined by the retired instructions, then the inference time of models should be close to a horizontal line in Figure 5.6; but the fact is that they generally show an upward trend. This proves that there is another factor that affects the inference time of models together with the retired instructions, and the impact of this factor is also related to the configuration of models.

A reasonable guess for this factor would be L2 misses, since more L2 misses means the processor needs to fetch data more frequently from the more time-consuming DRAM. The impact of different *Convolutional* modules on L2 misses and retired instructions is presented in Figure 5.7. First of all, this figure more clearly confirms the conclusion discussed before that from **Conv. Layer 1** to **Conv. Layer 5**, their effect on the increase of retired instructions continue to decline. Figure 5.7 further reflects that the influences of *Convolutional* modules on retired instructions and L2 misses is linear; for example, adding three **Conv. Layer 5** causes the growth of retired instructions and L2 misses to be 3 times those of adding only one **Conv. Layer 5**. When we analyze retired instructions and L2 misses together, more interesting facts will be discovered. The lines' slope of **Conv. Layer 5** (00000 - 00003) and **Conv. Layer 1** (00000 - 30000) are higher than those of other *Convolutional* modules. It indicates that these configurations mostly sweep across the data they use, causing lower LLC hit rate. While **Conv. Layer 2**, **Conv. Layer 3** and **Conv. Layer 4** make more computation on the same set of data since the amount of their retired instructions increases faster than the amount of their L2 misses.

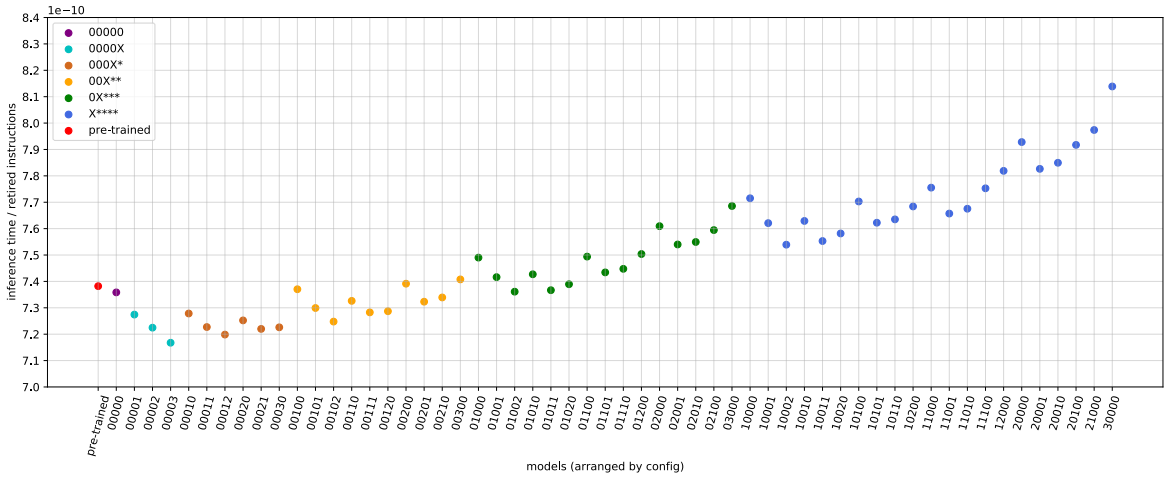

Figure 5.6: Ratio of inference time and retired instructions of different YOLOv3-tiny models, which are arranged by their configuration.

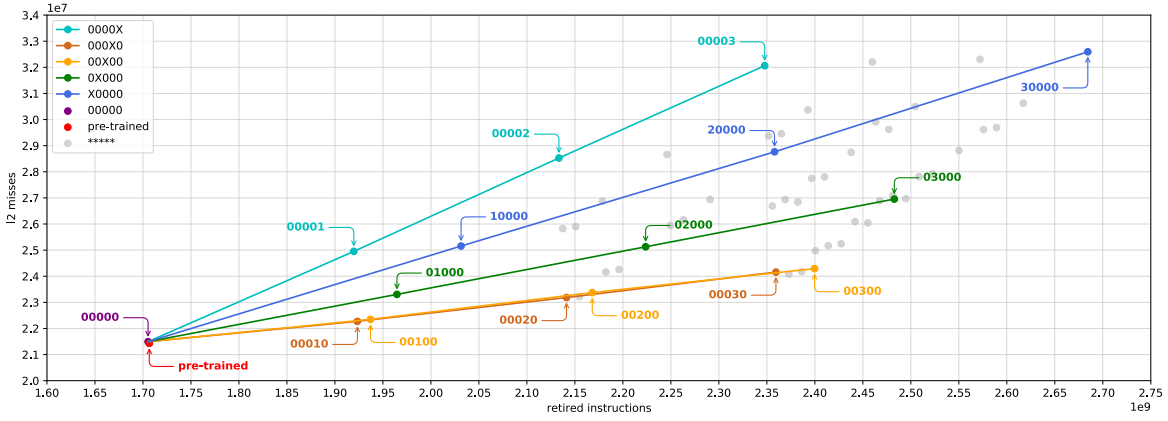

Figure 5.7: L2 misses and retired instructions of different YOLOv3-tiny models.

Next, we would like to further analyze the impact of different *Convolutional* modules on the bandwidth of models, which is presented in Figure 5.8 through the two metrics: L2 misses and retired instructions. From this figure, we can find that the line's slope of **Conv. Layer 5** (00000 - 00003) is significantly higher than that of other *Convolutional* modules. This indicates that **Conv. Layer 5** will cause more deterioration of bandwidth than other *Convolutional* modules. This is the result of the combined effect of L2 misses and inference time; on the one hand, as discussed before, **Conv. Layer 5** will cause more L2 misses than other *Convolutional* modules; on the other hand, Figure 5.8 reflects that from **Conv. Layer 1** to **Conv. Layer 5**, their influence on the growth of inference time continues to decrease. Therefore, as the ratio of L2 misses to inference time, bandwidth performs worst on **Conv. Layer 5**. In addition, Figure 5.8 also reflects that the influence of *Convolutional* modules on inference time is also linear.

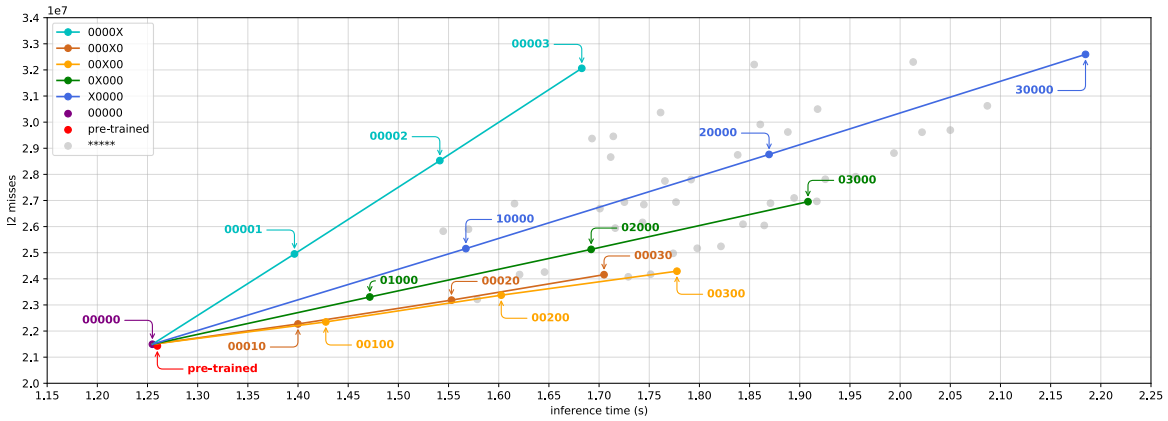

Figure 5.8: L2 misses and inference time of different YOLOv3-tiny models.

For the accuracy of models, we did not find clear rules. There are two possible reasons: First, the mechanism of influence on model accuracy is relatively complex, and there is no obvious relationship between accuracy and the number of retired instructions [2] or the type of *Convolutional* modules. Therefore, the benchmarks we designed are not enough to reveal its internal laws. Secondly, since the training of all our models is only based on COCO2017; transfer learning cannot be used, and the backbones of models are not pre-trained with the ImageNet database; in addition, each model is only trained for 12 Epochs. Thus the models may not be performing to their full potential in terms of accuracy. As mentioned in Section 4.3, we use the repository https://github.com/Cartucho/mAP and the dataset containing 4942 resized images and 80 classes to calculate the mAP50 of models. The model `pre-trained` loaded with pre-trained weights has the highest mAP50 of all models, reaching 19.35%, while the mAP50 of the model 00000 trained from scratch with unchanged structure is 15.59%. The accuracy of all other modified YOLOv3-tiny models fluctuates between 14.96% and 18.69%.

Table 5.5 ranks all models from good to bad according to their performance on different metrics, including inference time, L2 misses, bandwidth and mAP50[8]. Then we calculated the sum of all rankings of each model one by one, and ranked the models again according to the obtained sum of rankings from small to large. This composite ranking can be used to compare the overall performance of different YOLOv3-tiny models. From this table, we can find that the comprehensive performance of model 00120 and 00210 is better than that of model `pre-trained`. Although they are slightly inferior to the `pre-trained` in terms

[8]Among them, inference time, L2 misses and bandwidth are arranged in ascending order, while mAP50 is arranged in descending order

of inference time, L2 misses and accuracy, their performance in bandwidth is pretty good, resulting in better overall rankings.
Finally, we plotted scatter figures for each metric of all YOLOv3-tiny models, and they are presented in Appendix E. In these figures, different models are arranged on the X-axis according to their configuration, and the grouping method for each metric combines **Group one** and **Group two**. With these scatter figures, our above analysis can be more fully understood.

5.4.2 Distribution of Real-time Metrics

In this section we want to analyze the distribution of inference metrics of different YOLOv3-tiny models, in order to evaluate their stability when processing different images. Figure 5.9 presents the distribution of inference time for different models in the form of a violin chart, and identifies the position of the maximum, minimum, median and 99% quantile. Since we pre-resized the input images, the different original sizes of these images do not interfere with the distribution of inference time. From Figure 5.9, we can see that the position of 99% quantile and the minima of all models are very close to their respective median; this indicates that the inferences of all YOLOv3-tiny models are very stable for different inputs.
At the same time, we can also find that the maxima of inference time for all models are much higher than their respective median. We suspect that these maxima all come from the first execution of different models. To test this conjecture, we first computed the ratio of the first execution's inference time to the corresponding median for each model and plotted them as the orange bars in Figure 5.10. Then, we excluded the data of the first execution, calculated the maximum inference time for the remaining 4941 executions for each model, and similarly obtained the ratio of these maxima to the corresponding median, which are finally presented as the blue bars in Figure 5.10. The above two types of ratios reflect the deviation of their corresponding inference time from the median. From this figure, we can find that the inference time of the first execution is much larger than the maximum inference time of the remaining executions for all models. Therefore, we can confirm that the maximum value of the distribution of inference time in Figure 5.9 comes from the first execution of each model.
We speculate that the reason for this phenomenon is that although we hope to load the TF Lite model in the RT-Bench function *benchmark_init*, TensorFlow Lite may only pre-allocate the memory instead of actually loading the model; until *benchmark_execution* needs to use the model for inference, TensorFlow Lite fetches the model from DRAM or disk. If this assumption is correct, L2 misses of the first execution must be significantly larger than that of other executions for each model. To verify this hypothesis, we performed the same calculation on L2 misses as in Figure 5.10, that is, we first calculated the ratio of the first execution's L2 misses to the median of all executions' L2 misses for each model. Then we calculated the maximum of the remaining executions' L2 misses after excluding the first execution for each model, and also obtained their ratios to the corresponding median. These two types of ratios are represented by the orange and blue bars in Figure 5.11, respectively. However, this figure is exactly the opposite of what we expected: it reflects that L2 misses of the first execution are not the maximum for each model. We traced the source of the maxima for each model in the original benchmark data and found that noise is the main cause for them.
In fact, Figure 5.11 cannot deny our hypothesis, because when CPU cannot obtain model data from the L1 Cache and L2 Cache at the same time, such cache misses will be recorded as L1 misses. Therefore, our assumption will actually cause the L1 misses of the first execution to be much larger than those of the remaining executions for each model. Similarly, we computed the two types of ratios for L1 misses and presented them in Figure 5.12. From this figure, we can find that the L1 misses of the first execution are indeed much larger than

Table 5.5: Ranking YOLOv3-tiny Models according to Different Metrics.

Inference Time (↑)	L2 Misses (↑)	Bandwidth (↑)	mAP50 (↓)	Total Rank
00000	pre-trained	00300	pre-trained	00120
pre-trained	00000	00210	00120	00210
00001	00010	01200	01101	pre-trained
00010	00100	00120	00102	01010
00100	00020	02100	01110	01110
01000	00110	01110	00201	00030
00002	01000	10200	10002	00300
00011	00200	01020	00012	00100
00020	00120	03000	00210	00020
10000	00030	02010	10110	01020
00101	01010	00030	10101	00011
00110	00210	11100	00011	00110
00200	01100	10110	20001	02100
01001	00300	10020	11010	02010
01010	00001	11010	10020	00201
01100	01020	12000	01010	01200
00003	02000	20100	00003	00200
02000	10000	00200	02010	01000
00012	01110	20010	01011	01100
00021	01200	21000	02100	10020
00030	00011	00110	00111	00000
10001	00101	01100	10001	10110
00102	10010	02000	00002	00010
10010	02100	01010	00021	02000
00111	02010	30000	11100	10100
00120	10100	00020	10100	01101
10100	00021	10100	00030	00021
00201	00201	10010	20010	00111
00210	01001	11000	00300	00001
01002	10020	20000	01020	11010
01011	00111	00201	12000	03000
01020	11000	01101	03000	10000
11000	03000	00111	01200	00102
00300	10200	02001	10011	01011
01101	10110	00100	02001	00012
01110	01011	10101	01001	01001
01200	01101	00021	00100	00002
02001	11010	01011	00001	11100
02010	11100	01000	11000	10200
10002	00002	11001	10200	10010
10011	10001	00010	20000	00101
02100	02001	20001	01000	10001
20000	20000	10000	01100	11000
10020	12000	10011	21000	10101
10101	00012	00101	00020	12000
10110	00102	01001	01002	00003
03000	20010	00011	02000	20010
10200	10101	10001	20100	02001
11001	20100	pre-trained	00110	10002
11010	10011	00000	10000	20000
11100	01002	00102	00000	20001
12000	11001	01002	30000	10011
20001	21000	00012	11001	20100
20010	00003	10002	00101	21000
20100	10002	00001	00200	01002
21000	20001	00002	10010	30000
30000	30000	00003	00010	11001

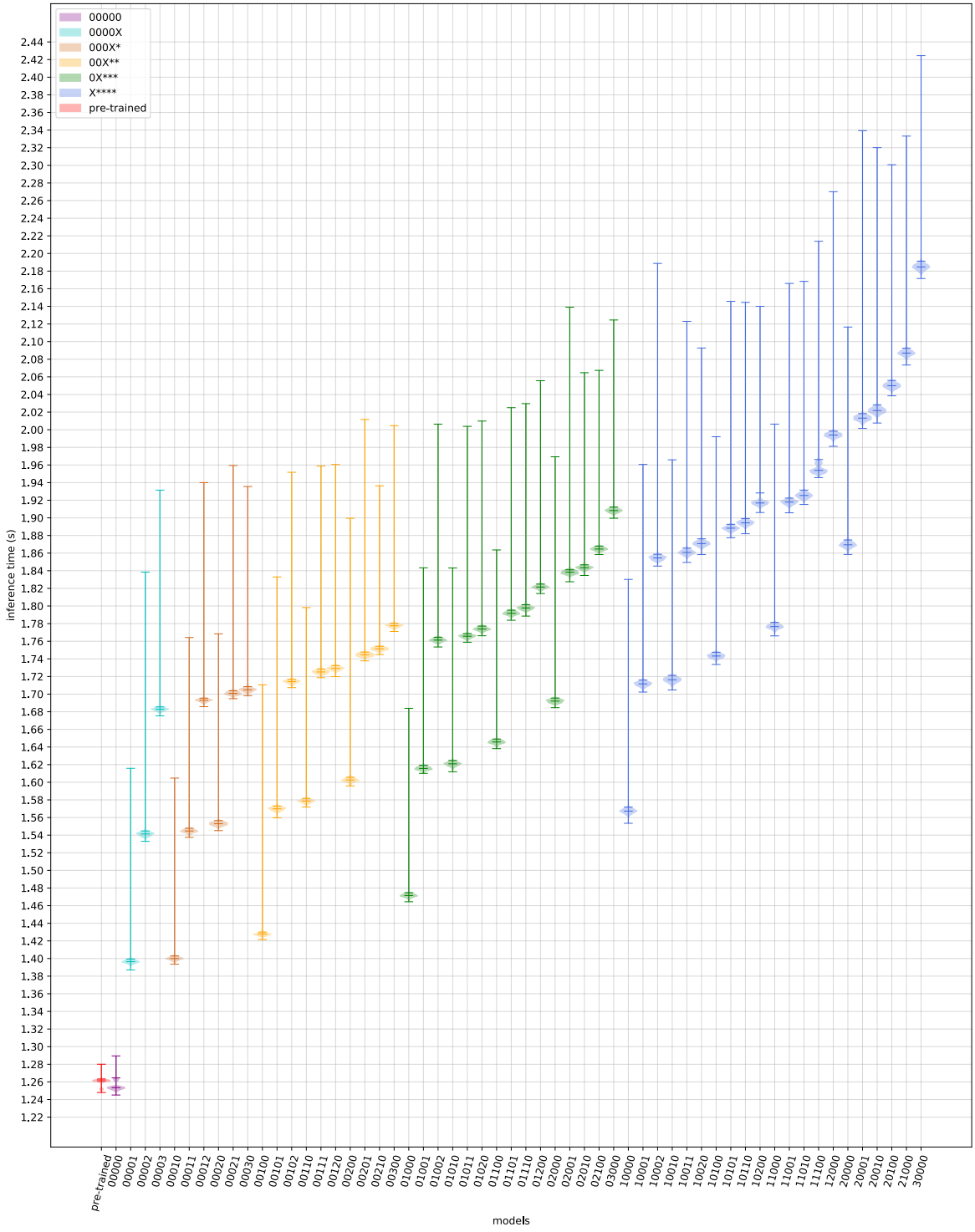

Figure 5.9: Distribution of inference time for YOLOv3-tiny models.

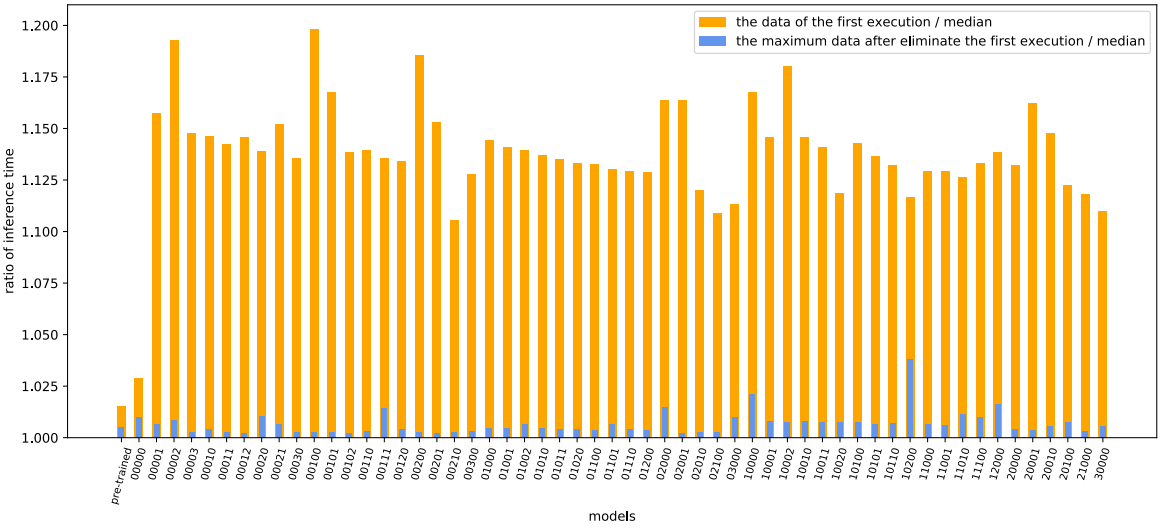

Figure 5.10: Ratio of different executions' inference time to the corresponding median for each YOLOv3-tiny model.

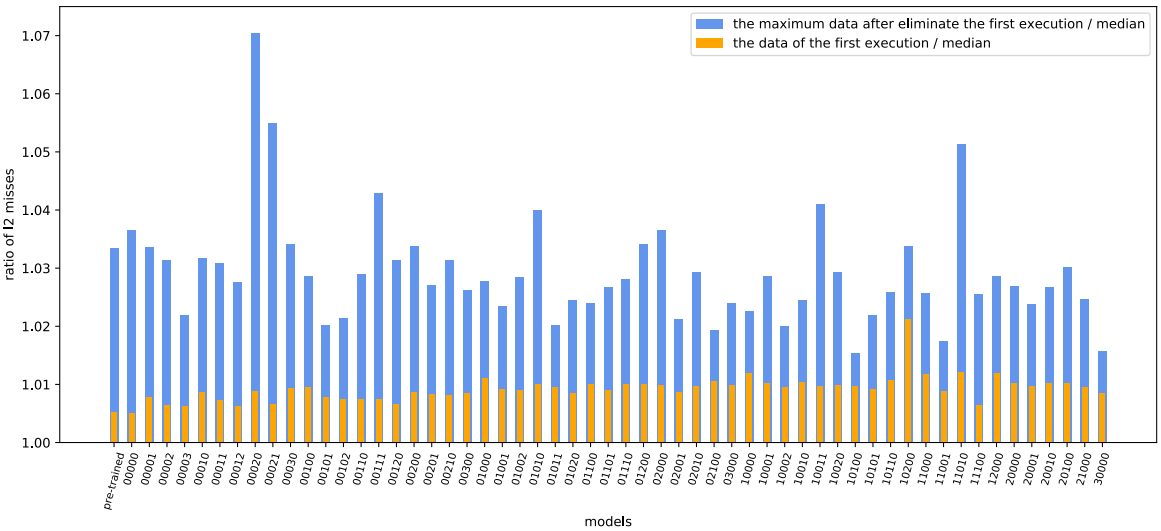

Figure 5.11: Ratio of different executions' L2 misses to the corresponding median for each YOLOv3-tiny model.

the remaining executions' L1 misses for all models, which somewhat supports our hypothesis that TF Lite does not actually preload the DNN, but only pre-allocates memory for it; only when the model is needed for inference, TF Lite will fetch the model data from DRAM or disk.

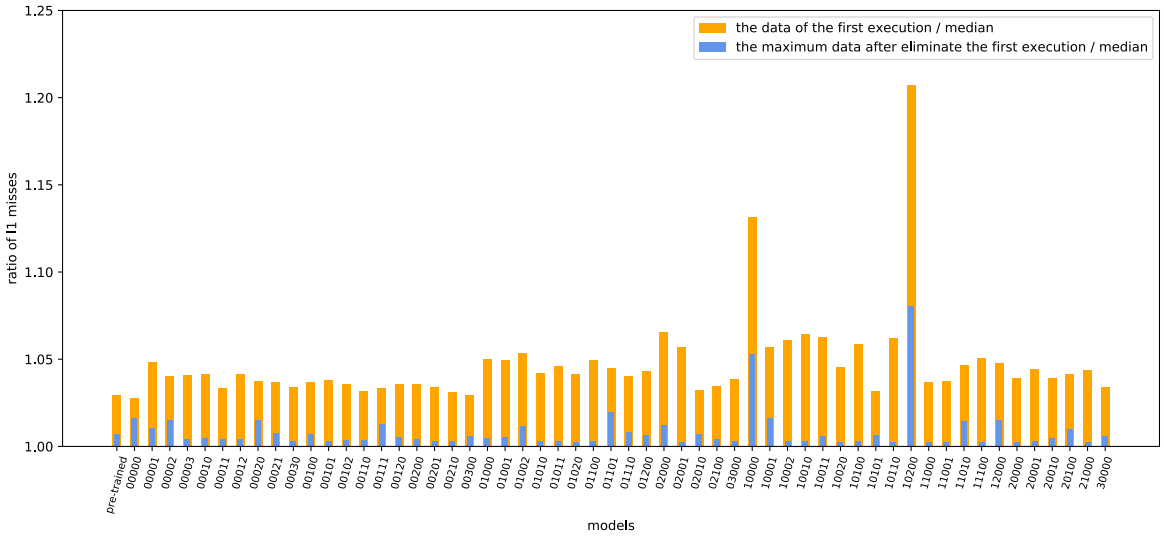

Figure 5.12: Ratio of different executions' L1 misses to the corresponding median for each YOLOv3-tiny model.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This paper intends to explore the real-time features of deep neural network inferences, focusing on the metrics such as inference time, accuracy, memory usage and so on. We have performed multiple benchmarks in two field: *Image Classification* and *Object Detection*. For the former, we first investigated the impact of different architectures of DNNs on their real-time features, involving 17 full precision models from 5 families. Then we analyzed the influence of Quantization, one of the most important model compression techniques, on the real-time features of DNN by comparing the inference performance of eight quantized models with their corresponding full precision models. Finally, we enabled XNNPACK to benchmark all the full precision models again, and analyzed the effect of XNNPACK on the real-time features of model inferences by comparing their performance before and after enabling XNNPACK. In addition, we also observed the distribution of metrics of inference with XNNPACK to evaluate the stability of XNNPACK-assisted DNN inferences. For *Object Detection*, we expected to research the impact of the specific structure of DNNs on the real-time inference features, and the first to be considered structure element is the depth of DNNs. To this end, we benchmarked YOLOv3-tiny models with different depths by duplicating and adding different *Convolutional* modules in Darknet-tiny. Then we also further analyzed the distribution of metrics for these models to study their stability of inferences for different input images.
From the above mentioned multiple benchmarks, rich conclusions have been obtained. Now we summarize them as follows:

- **Different DNN architectures**: VGG family lags behind all other models in terms of inference time, accuracy, LLC misses, and bandwidth. In addition, the size of `vgg16` and `vgg19` are also much larger than other models. For Inception family, except `inception_v1`, other members have higher Top-1 accuracy. Each iteration of its version improves accuracy while sacrificing inference speed and causing more LLC misses. Regarding the bandwidth, except for `inception_v4` which has higher bandwidth, the performance of other members is at the middle level of all models. For ResNet family, as the number of model layers continues to increase, the accuracy and inference time of the version "v1" and "v2" continue to increase, and the accuracy improvement of "v2" is slower than that of "v1". Overall, the performance of this family on these two metrics is mediocre. For LLC misses and bandwidth, the two versions of ResNet perform extremely closely; LLC misses is also positively correlated with the number of model layers, and their bandwidth is higher than that of Inception family. MobileNet family performs extremely well in terms of inference time, LLC misses, and bandwidth, while their accuracy is at the middle level of all models. The difference between its members is relatively small. Finally, `efficientnet_b0` from EfficientNet family performs

similarly to MobileNet family in terms of inference time, LLC misses and bandwidth, but its accuracy is higher than that of MobileNet family. Therefore, `efficientnet_b0` performs better overall in all full precision models.

- **Quantization**: As we reduce the precision of models from fp32 to uint8 by Quantization, the size of all models is also reduced to about a quarter of their original size. The inference time of these quantized models is generally reduced by about 30%, while the accuracy is reduced by no more than 10%. Even the accuracy of `inception_v4_q` is slightly improved after fine tuning. The improvement of LLC misses and bandwidth caused by Quantization is more obvious than that of inference speed. On average, the LLC misses and bandwidth of all quantized models are only about 10% and 15% of the original, respectively. Among them, the influence of Quantization on Inception Family in these two metrics is particularly remarkable, the LLC misses and bandwidth of `inception_v4_q` are only 5.46% and 8.63% of those before quantization.

- **XNNPACK**: Since XNNPACK does not modify the model itself, after enabling XNNPACK for inference, the size and accuracy of all models are the same as before, but the inference time decreases by about 25% on average. This is due to the slight reduction in the amount of retired instructions and more than 15% improvement in IPC caused by XNNPACK for all models. The impact of XNNPACK on LLC misses varies for different models; the LLC misses of most models increase by a maximum of 63.80% after XNNPACK is enabled, while the LLC misses of a few models decrease by a maximum of 23.47%. Furthermore, XNNPACK deteriorates the bandwidth of all models to varying degrees. Among them, the bandwidth of `mobilenet_v1_x` increases by 123.28%, while the bandwidth of `inception_v1_x` increases by only 4.72%.

- **YOLOv3-tiny with different depths**: Firstly, there is no necessary relationship between the size of a model and its inference time. What can really have a significant impact on the inference time of models are the retired instructions and LLC misses. More retired instructions and LLC misses will result in longer inference time.

  Secondly, the different *Convolutional* modules of Darknet-tiny will have varying degrees of impact on the metrics of YOLOv3-tiny models. From **Conv. Layer 1** to **Conv. Layer 5**, their effect on the increase of model size is continuously enhanced, while their influence on the growth of retired instructions and inference time continue to decline. **Conv. Layer 5** and **Conv. Layer 1** will increase LLC misses to a greater extent than other *Convolutional* modules, and **Conv. Layer 5** will cause larger bandwidth than all other *Convolutional* modules. Furthermore, the impact of these *Convolutional* modules on inference time, retired instructions and LLC misses is linear. For the accuracy of these models, we did not find a clear pattern.

  Since the increase of retired instructions, LLC misses and bandwidth caused by **Conv. Layer 3** and **Conv. Layer 4** are small, and the accuracy of model `00120` and `00210` is relatively high, their comprehensive performance on the four metrics of inference time, LLC misses, bandwidth and mAP50 is better than model `pre-trained`.

  Finally, the inference of all YOLOv3-tiny models are stable for different input images, but the inference time and L1 misses of their first execution are much higher than other executions. A reasonable hypothesis for this phenomenon is that TF Lite does not actually preload the DNN, but only pre-allocates memory for it; only when the model is needed for inference, TF Lite will fetch the model data from DRAM or disk.

6.2 Future Work

Due to the constraints of time and resource , our benchmark may be further improved in some aspects. In addition, based on Tensorflow Lite and RT-Bench, we can further explore the real-time features of DNN inferences from many other perspectives. Below we list some of these highly valuable future work:

- **Accuracy of YOLOv3-tiny models**: Since we did not train those YOLOv3-tiny models sufficiently, they may not have achieved the best accuracy they could achieve. Therefore, in future work, we can first use the ImageNet database to pre-train the backbone of YOLOv3-tiny models so that they can better extract the features of images, and then adopt more superior training strategies to train those YOLOv3-tiny models for more Epochs based on COCO database. By analyzing the mAP of these well-trained models, we may discover the underlying mechanisms that affect the accuracy of models.
- **Modify the width and resolution of YOLOv3-tiny**: In addition to depth, width (number of channels) and resolution of models can have a significant impact on their inference[28]. In the completed benchmarks, we explored the influence of depth on real-time features of DNN inferences by duplicating and adding *Convolutional* Modules in Darknet-tiny. In future work, we can further observe the impact of width and resolution of models on inference time, accuracy, LLC misses and other metrics by scaling the width (number of channels) of YOLOv3-tiny and the size of input images.
- **Hardware acceleration**: In this paper, we mainly studied the impact of some software acceleration technologies on the real-time features of DNN inference on CPU, including TensorFlow Lite, Quantization and XNNPACK. However, as mentioned in Chapter 1, the hardware acceleration technologies for DNN inference are also developing rapidly. Among them, the influence of GPU and TPU on the inference speed of DNN has been widely researched. TensorFlow Lite also provides support for these Delegates. In addition, our benchmark device, Coral Dev Board, provides us with an Edge TPU coprocessor. Thus in future work, we plan to further explore the impact of Edge TPU accelerator on the speed, accuracy and memory consumption of model inference based on RT-Bench and TensorFlow Lite.
- **Combination of DNN acceleration techniques**: So far, we have only studied the impact of a single acceleration technique on the real-time features of DNN inferences. The effect provided by combination of different acceleration technologies is also worth looking forward to. For example, we can further explore the impact of the combination of 16-bit floating-point quantization and XNNPACK on the inference metrics, such as inference time, accuracy, and LLC misses.
- **Multi-core processing and multi-threading**: All the benchmarks we have completed so far only utilize one core of Cortex_A53. In practice, multi-core processing is more common. Therefore, in future work, we would like to further explore the impact of multi-core processing on real-time features of DNN inferences. In addition, in the case of multi-core processing, enabling multi-threading can often speed up program execution. Therefore, it is also valuable to further research the influence of different combinations of core numbers and thread numbers on DNN inferences.
- **Stress testing**: In practical applications, edge devices often have various other workloads while executing DNNs. Therefore, it is necessary to benchmark the impact of CPU/GPU/TPU under different workloads on the stability of DNN inference. In addition, the situation of undervolting or overclocking is also worth considering.

Appendix A

Details of Models for Image Classification

Table A.1: Details of TF Lite Models for Image Classification.

TF Lite Models	Details
`vgg16_224_fp32.tflite`	Short Name: `vgg16` Architecture: VGG, 16 layers Model Size: 553.4 MB Precision: unquantized (fp32) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 1[1] Preprocess of Input Image: 224x224x3, [-128, 127], BGR[2] Source: (converted from[3]) `tf.keras.applications.vgg16.VGG16()`
`vgg19_224_fp32.tflite`	Short Name: `vgg19` Architecture: VGG, 19 layers Model Size: 574.7 MB Precision: unquantized (fp32) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 1 Preprocess of Input Image: 224x224x3, [-128, 127], BGR Source: (converted from) `tf.keras.applications.vgg19.VGG19()`
`inception_v1_224_fp32.tflite`	Short Name: `inception_v1` Architecture: Inception V1 Model Size: 26.6 MB Precision: unquantized (fp32) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 0 Preprocess of Input Image: 224x224x3, [0, 1], RGB Source: (converted from) https://tfhub.dev/google/imagenet/inception_v1/classification/5
`inception_v1_224_uint8.tflite`	Short Name: `inception_v1_q` Architecture: Inception V1 Model Size: 6.7 MB Precision: quantized (uint8) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 0 Preprocess of Input Image: 224x224x3, [0, 255], RGB Source: https://tfhub.dev/tensorflow/lite-model/inception_v1_quant/1/default/1
`inception_v2_224_fp32.tflite`	Short Name: `inception_v2` Architecture: Inception V2 Model Size: 44.8 MB Precision: unquantized (fp32) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 0 Preprocess of Input Image: 224x224x3, [0, 1], RGB Source: (converted from) https://tfhub.dev/google/imagenet/inception_v2/classification/5

[1]Index of classes in the labels starts from "1"; that means, the first class "*background*" has index "1".
[2]The input size is 224x224x3, with values in the range [-128, 127] and with color in the order BGR.
[3]The original is released as a TensorFlow model and converted into TensorFlow Lite model.

`inception_v2_224_uint8.tflite`
Short Name: `inception_v2_q`
Architecture: Inception V2
Model Size: 11.3 MB
Precision: quantized (uint8)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 255], RGB
Source: https://tfhub.dev/tensorflow/lite-model/inception_v2_quant/1/default/1

`inception_v3_299_fp32.tflite`
Short Name: `inception_v3`
Architecture: Inception V3
Model Size: 95.3 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 299x299x3, [-1, 1], RGB
Source: https://tfhub.dev/tensorflow/lite-model/inception_v3/1/default/1

`inception_v3_299_uint8.tflite`
Short Name: `inception_v3_q`
Architecture: Inception V3
Model Size: 23.9 MB
Precision: quantized (uint8)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 299x299x3, [0, 255], RGB
Source: https://tfhub.dev/tensorflow/lite-model/inception_v3_quant/1/default/1

`inception_v4_299_fp32.tflite`
Short Name: `inception_v4`
Architecture: Inception V4
Model Size: 170.7 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 299x299x3, [-1, 1], RGB
Source: https://tfhub.dev/tensorflow/lite-model/inception_v4/1/default/1

`inception_v4_299_uint8.tflite`
Short Name: `inception_v4_q`
Architecture: Inception V4
Model Size: 42.9 MB
Precision: quantized (uint8)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 299x299x3, [0, 255], RGB
Source: https://tfhub.dev/tensorflow/lite-model/inception_v4_quant/1/default/1

`inception_resnet_v2_299_fp32.tflite`
Short Name: `inception_resnet_v2`
Architecture: Inception-ResNet V2
Model Size: 121.0 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 299x299x3, [-1, 1], RGB
Source: https://tfhub.dev/tensorflow/lite-model/inception_resnet_v2/1/default/1

`resnet50_v1_224_fp32.tflite`
Short Name: `resnet50_v1`
Architecture: ResNet V1, 50 layers
Model Size: 102.2 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 1], RGB
Source: (converted from) https://tfhub.dev/google/imagenet/resnet_v1_50/classification/5

`resnet101_v1_224_fp32.tflite`
Short Name: `resnet101_v1`
Architecture: ResNet V1, 101 layers
Model Size: 178.1 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 1], RGB
Source: (converted from) https://tfhub.dev/google/imagenet/resnet_v1_101/classification/5

`resnet152_v1_224_fp32.tflite`
Short Name: `resnet152_v1`
Architecture: ResNet V1, 152 layers
Model Size: 240.7 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 1], RGB
Source: (converted from) https://tfhub.dev/google/imagenet/resnet_v1_152/classification/5

`resnet50_v2_224_fp32.tflite`
Short Name: `resnet50_v2`
Architecture: ResNet V2, 50 layers
Model Size: 102.3 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 1], RGB
Source: (converted from) https://tfhub.dev/google/imagenet/resnet_v2_50/classification/5

`resnet101_v2_224_fp32.tflite`
Short Name: `resnet101_v2`
Architecture: ResNet V2, 101 layers
Model Size: 178.4 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 1], RGB
Source: (converted from) https://tfhub.dev/google/imagenet/resnet_v2_101/classification/5

`resnet152_v2_224_fp32.tflite`
Short Name: `resnet152_v2`
Architecture: ResNet V2, 152 layers
Model Size: 241.1 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 1], RGB
Source: (converted from) https://tfhub.dev/google/imagenet/resnet_v2_152/classification/5

`mobilenet_v1_1.0_224_fp32.tflite`
Short Name: `mobilenet_v1`
Architecture: MobileNet V1, alpha = 1.0
Model Size: 16.9 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [-1, 1], RGB
Source: https://tfhub.dev/iree/lite-model/mobilenet_v1_100_224/fp32/1

`mobilenet_v1_1.0_224_uint8.tflite`
Short Name: `mobilenet_v1_q`
Architecture: MobileNet V1, alpha = 1.0
Model Size: 4.3 MB
Precision: quantized (uint8)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 255], RGB
Source: https://tfhub.dev/iree/lite-model/mobilenet_v1_100_224/uint8/1

`mobilenet_v2_1.0_224_fp32.tflite`
Short Name: `mobilenet_v2`
Architecture: MobileNet V2, alpha = 1.0
Model Size: 14.0 MB
Precision: unquantized (fp32)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [-1, 1], RGB
Source: https://tfhub.dev/iree/lite-model/mobilenet_v2_100_224/fp32/1

`mobilenet_v2_1.0_224_uint8.tflite`
Short Name: `mobilenet_v2_q`
Architecture: MobileNet V2, alpha = 1.0
Model Size: 3.6 MB
Precision: quantized (uint8)
Train/Validation Dataset: ILSVRC2012
Labels: index starts from 0
Preprocess of Input Image: 224x224x3, [0, 255], RGB
Source: https://tfhub.dev/iree/lite-model/mobilenet_v2_100_224/uint8/1

`mobilenet_v3_large_1.0_224_fp32.tflite`	Short Name: `mobilenet_v3` Architecture: MobileNet V3 Large, alpha = 1.0 Model Size: 21.9 MB Precision: unquantized (fp32) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 0 Preprocess of Input Image: 224x224x3, [-1, 1], RGB Source: https://tfhub.dev/iree/lite-model/mobilenet_v3_large_100_224/fp32/1
`mobilenet_v3_large_1.0_224_uint8.tflite`	Short Name: `mobilenet_v3_q` Architecture: MobileNet V3 Large, alpha = 1.0 Model Size: 5.6 MB Precision: quantized (uint8) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 0 Preprocess of Input Image: 224x224x3, [0, 255], RGB Source: https://tfhub.dev/iree/lite-model/mobilenet_v3_large_100_224/uint8/1
`efficientnet_b0_224_fp32.tflite`	Short Name: `efficientnet_b0` Architecture: EfficientNet B0 Model Size: 18.6 MB Precision: unquantized (fp32) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 1 Preprocess of Input Image: 224x224x3, [-1, 1], RGB Source: https://tfhub.dev/tensorflow/lite-model/efficientnet/lite0/fp32/2
`efficientnet_b0_224_uint8.tflite`	Short Name: `efficientnet_b0_q` Architecture: EfficientNet B0 Model Size: 5.4 MB Precision: quantized (uint8) Train/Validation Dataset: ILSVRC2012 Labels: index starts from 1 Preprocess of Input Image: 224x224x3, [0, 255], RGB Source: https://tfhub.dev/tensorflow/lite-model/efficientnet/lite0/uint8/2

Appendix B

Details of Models for Object Detection

Table B.1: Details of YOLOv3-tiny Models with different Depths.

TF Lite Models	Short Names	Size (MB)	Duplicated and Added *Convolutional* Modules
`yolov3-tiny.tflite`	`pre-trained`	35.4	pre-trained, unmodified
`yolov3-tiny_00000.tflite`	00000	35.4	unmodified
`yolov3-tiny_00001.tflite`	00001	37.8	1x Conv. Layer 5
`yolov3-tiny_00002.tflite`	00002	40.2	2x Conv. Layer 5
`yolov3-tiny_00003.tflite`	00003	42.5	3x Conv. Layer 5
`yolov3-tiny_00010.tflite`	00010	36.0	1x Conv. Layer 4
`yolov3-tiny_00011.tflite`	00011	38.4	1x Conv. Layer 4, 1x Conv. Layer 5
`yolov3-tiny_00012.tflite`	00012	40.8	1x Conv. Layer 4, 2x Conv. Layer 5
`yolov3-tiny_00020.tflite`	00020	36.6	2x Conv. Layer 4
`yolov3-tiny_00021.tflite`	00021	39.0	2x Conv. Layer 4, 1x Conv. Layer 5
`yolov3-tiny_00030.tflite`	00030	37.2	3x Conv. Layer 4
`yolov3-tiny_00100.tflite`	00100	35.6	1x Conv. Layer 3
`yolov3-tiny_00101.tflite`	00101	37.9	1x Conv. Layer 3, 1x Conv. Layer 5
`yolov3-tiny_00102.tflite`	00102	40.3	1x Conv. Layer 3, 2x Conv. Layer 5
`yolov3-tiny_00110.tflite`	00110	36.2	1x Conv. Layer 3, 1x Conv. Layer 4
`yolov3-tiny_00111.tflite`	00111	38.5	1x Conv. Layer 3, 1x Conv. Layer 4, 1x Conv. Layer 4
`yolov3-tiny_00120.tflite`	00120	36.8	1x Conv. Layer 3, 2x Conv. Layer 4
`yolov3-tiny_00200.tflite`	00200	35.7	2x Conv. Layer 3
`yolov3-tiny_00201.tflite`	00201	38.1	2x Conv. Layer 3, 1x Conv. Layer 5
`yolov3-tiny_00210.tflite`	00210	36.3	2x Conv. Layer 3, 1x Conv. Layer 4
`yolov3-tiny_00300.tflite`	00300	35.9	3x Conv. Layer 3
`yolov3-tiny_01000.tflite`	01000	35.5	1x Conv. Layer 2
`yolov3-tiny_01001.tflite`	01001	37.8	1x Conv. Layer 2, 1x Conv. Layer 5
`yolov3-tiny_01002.tflite`	01002	40.2	1x Conv. Layer 2, 2x Conv. Layer 5
`yolov3-tiny_01010.tflite`	01010	36.1	1x Conv. Layer 2, 1x Conv. Layer 4
`yolov3-tiny_01011.tflite`	01011	38.4	1x Conv. Layer 2, 1x Conv. Layer 4, 1x Conv. Layer 5
`yolov3-tiny_01020.tflite`	01020	36.7	1x Conv. Layer 2, 2x Conv. Layer 4
`yolov3-tiny_01100.tflite`	01100	35.6	1x Conv. Layer 2, 1x Conv. Layer 3
`yolov3-tiny_01101.tflite`	01101	38.0	1x Conv. Layer 2, 1x Conv. Layer 3, 1x Conv. Layer 5
`yolov3-tiny_01110.tflite`	01110	36.2	1x Conv. Layer 2, 1x Conv. Layer 3, 1x Conv. Layer 4
`yolov3-tiny_01200.tflite`	01200	35.8	1x Conv. Layer 2, 2x Conv. Layer 3
`yolov3-tiny_02000.tflite`	02000	35.5	2x Conv. Layer 2
`yolov3-tiny_02001.tflite`	02001	37.9	2x Conv. Layer 2, 1x Conv. Layer 5
`yolov3-tiny_02010.tflite`	02010	36.1	2x Conv. Layer 2, 1x Conv. Layer 4
`yolov3-tiny_02100.tflite`	02100	35.7	2x Conv. Layer 2, 1x Conv. Layer 3
`yolov3-tiny_03000.tflite`	03000	35.6	3x Conv. Layer 2
`yolov3-tiny_10000.tflite`	10000	35.4	1x Conv. Layer 1
`yolov3-tiny_10001.tflite`	10001	37.8	1x Conv. Layer 1, 1x Conv. Layer 5
`yolov3-tiny_10002.tflite`	10002	40.2	1x Conv. Layer 1, 2x Conv. Layer 5
`yolov3-tiny_10010.tflite`	10010	36.0	1x Conv. Layer 1, 1x Conv. Layer 4
`yolov3-tiny_10011.tflite`	10011	38.4	1x Conv. Layer 1, 1x Conv. Layer 4, 1x Conv. Layer 5

yolov3-tiny_10020.tflite	10020	36.6	1x Conv. Layer 1, 2x Conv. Layer 4
yolov3-tiny_10100.tflite	10100	35.6	1x Conv. Layer 1, 1x Conv. Layer 3
yolov3-tiny_10101.tflite	10101	38.0	1x Conv. Layer 1, 1x Conv. Layer 3, 1x Conv. Layer 5
yolov3-tiny_10110.tflite	10110	36.2	1x Conv. Layer 1, 1x Conv. Layer 3, 1x Conv. Layer 4
yolov3-tiny_10200.tflite	10200	35.7	1x Conv. Layer 1, 2x Conv. Layer 3
yolov3-tiny_11000.tflite	11000	35.5	1x Conv. Layer 1, 1x Conv. Layer 2
yolov3-tiny_11001.tflite	11001	37.8	1x Conv. Layer 1, 1x Conv. Layer 2, 1x Conv. Layer 5
yolov3-tiny_11010.tflite	11010	36.1	1x Conv. Layer 1, 1x Conv. Layer 2, 1x Conv. Layer 4
yolov3-tiny_11100.tflite	11100	35.6	1x Conv. Layer 1, 1x Conv. Layer 2, 1x Conv. Layer 3
yolov3-tiny_12000.tflite	12000	35.5	1x Conv. Layer 1, 2x Conv. Layer 2
yolov3-tiny_20000.tflite	20000	35.5	2x Conv. Layer 1
yolov3-tiny_20001.tflite	20001	37.8	2x Conv. Layer 1, 1x Conv. Layer 5
yolov3-tiny_20010.tflite	20010	36.0	2x Conv. Layer 1, 1x Conv. Layer 4
yolov3-tiny_20100.tflite	20100	35.6	2x Conv. Layer 1, 1x Conv. Layer 3
yolov3-tiny_21000.tflite	21000	35.5	2x Conv. Layer 1, 1x Conv. Layer 2
yolov3-tiny_30000.tflite	30000	35.5	3x Conv. Layer 1

Appendix C

Metrics w.r.t Models for Image Classification

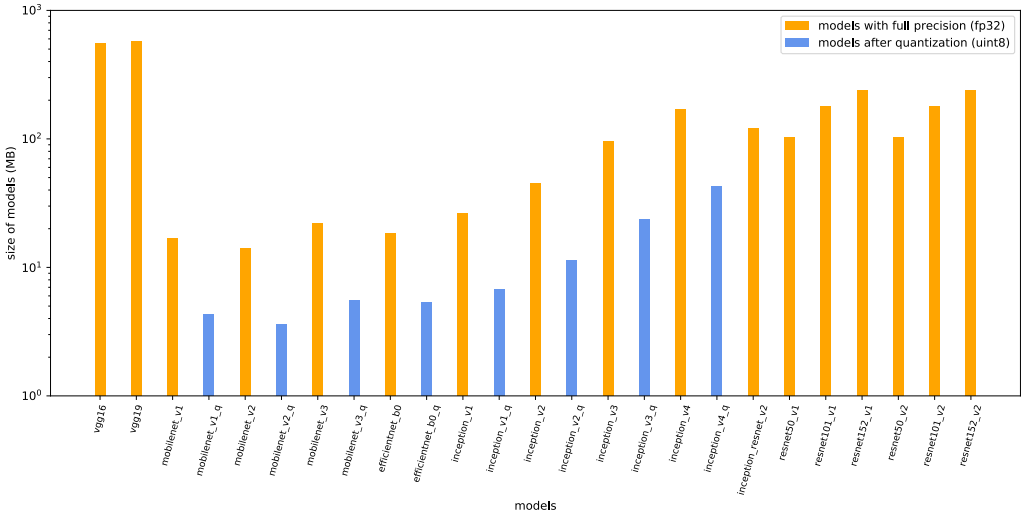

Figure C.1: Size of full precision and quantized TF Lite models for Image Classification.

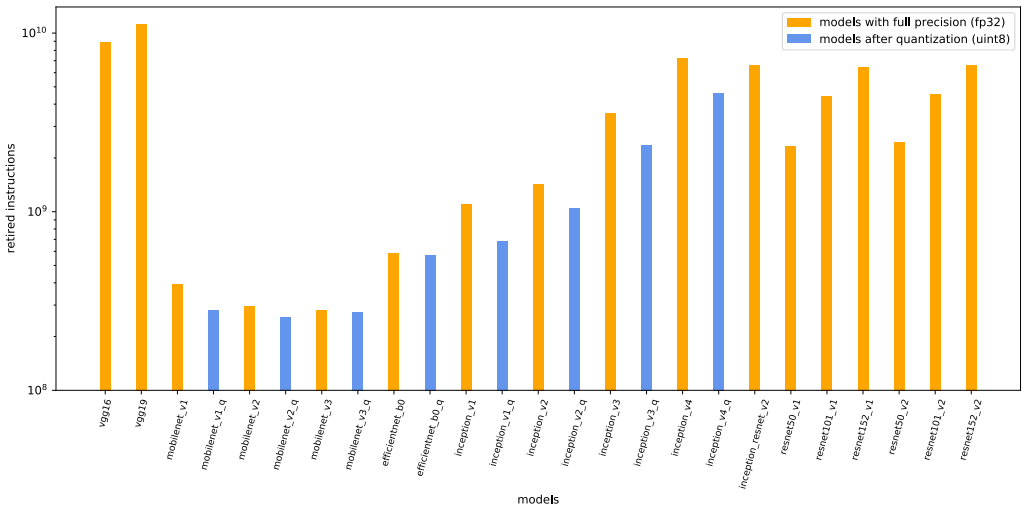

Figure C.2: Retired instructions of full precision and quantized TF Lite models for Image Classification.

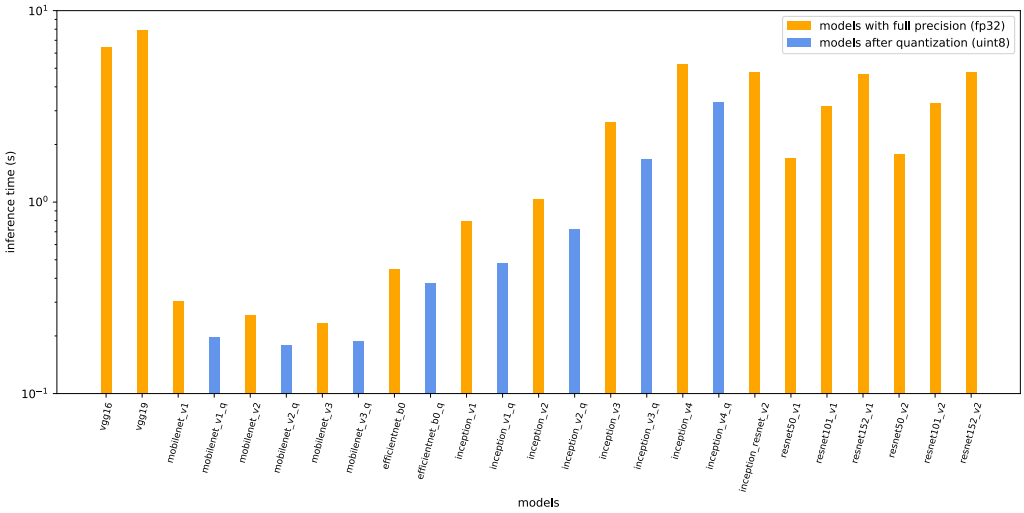

Figure C.3: Inference time of full precision and quantized TF Lite models for Image Classification.

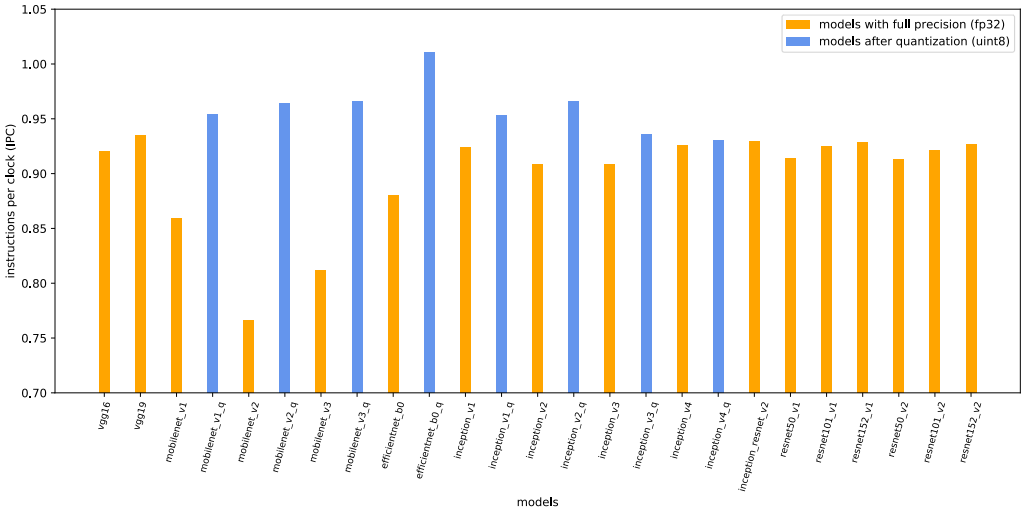

Figure C.4: Instructions per clock (IPC) of full precision and quantized TF Lite models for Image Classification.

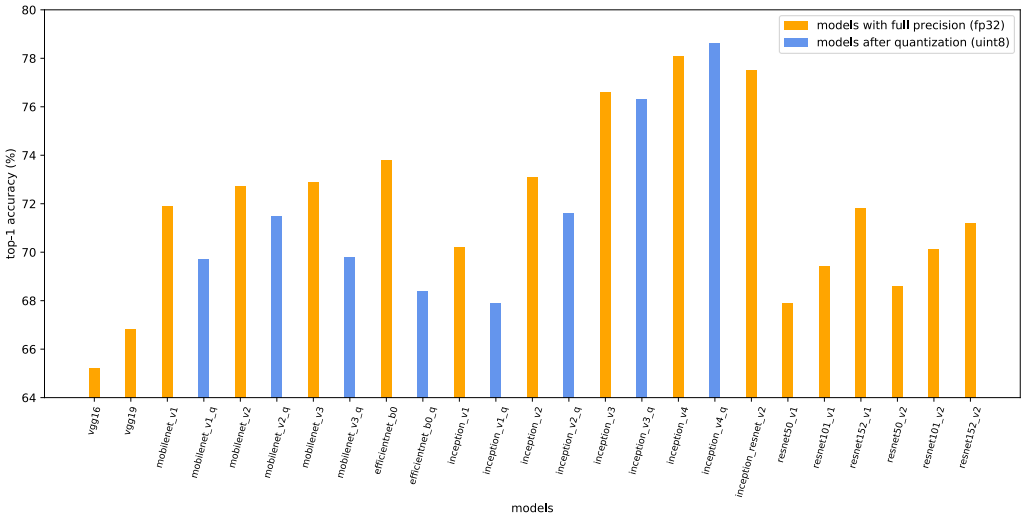

Figure C.5: Top-1 accuracy of full precision and quantized TF Lite models for Image Classification.

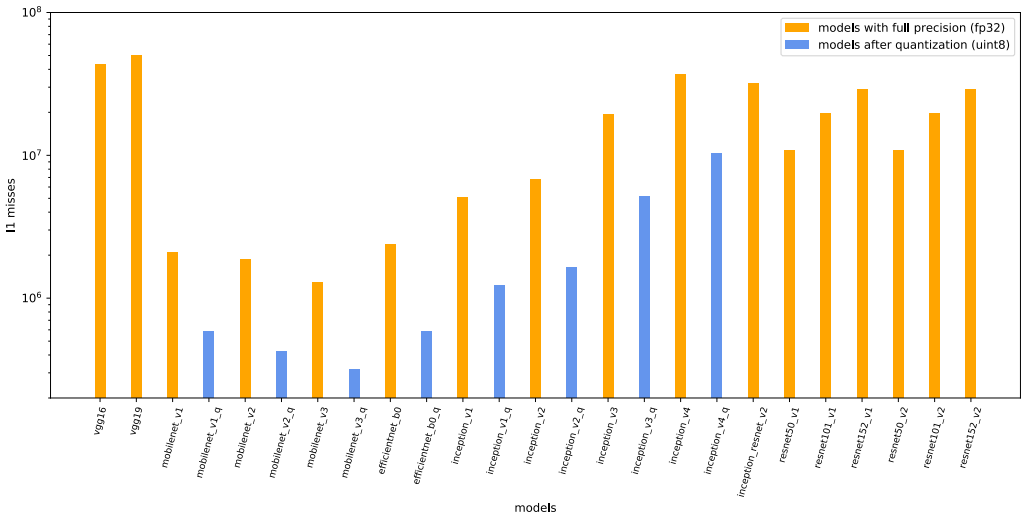

Figure C.6: L1 misses of full precision and quantized TF Lite models for Image Classification.

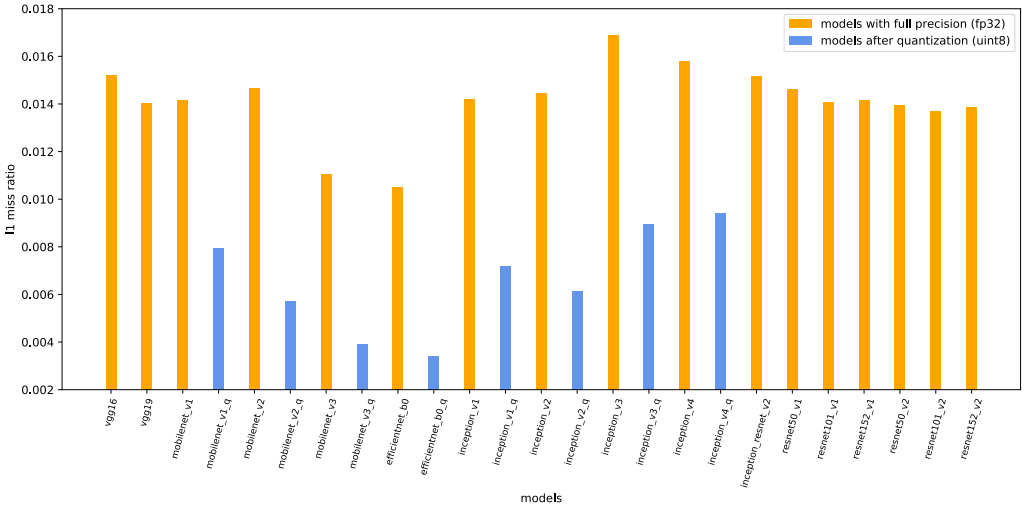

Figure C.7: L1 miss ratio of full precision and quantized TF Lite models for Image Classification.

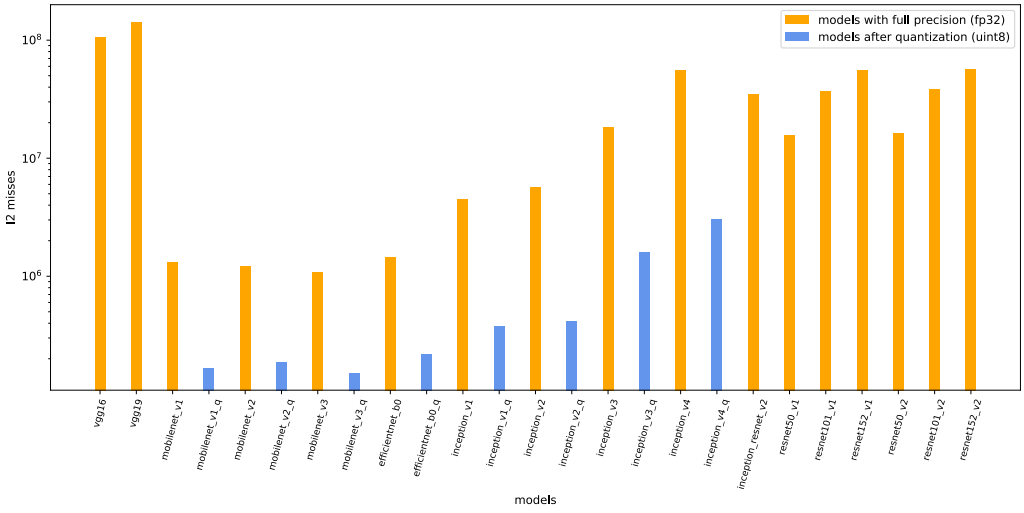

Figure C.8: L2 misses of full precision and quantized TF Lite models for Image Classification.

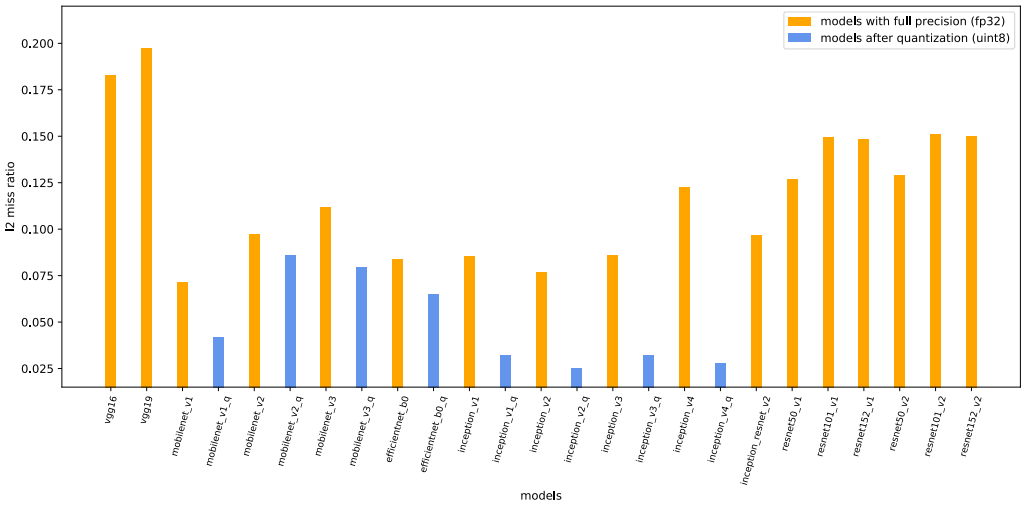

Figure C.9: L2 miss ratio of full precision and quantized TF Lite models for Image Classification.

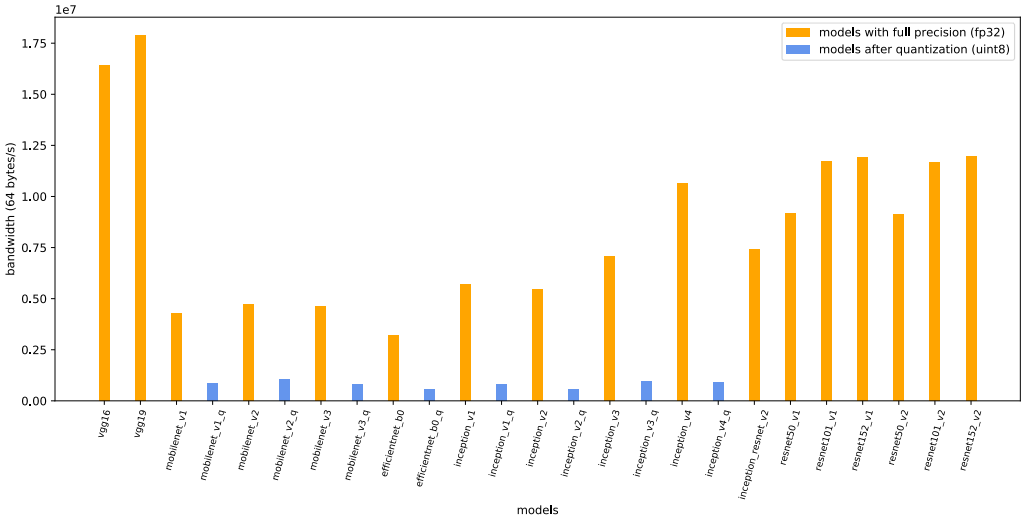

Figure C.10: Bandwidth of full precision and quantized TF Lite models for Image Classification.

Appendix D

Impacts of XNNPACK on Metrics

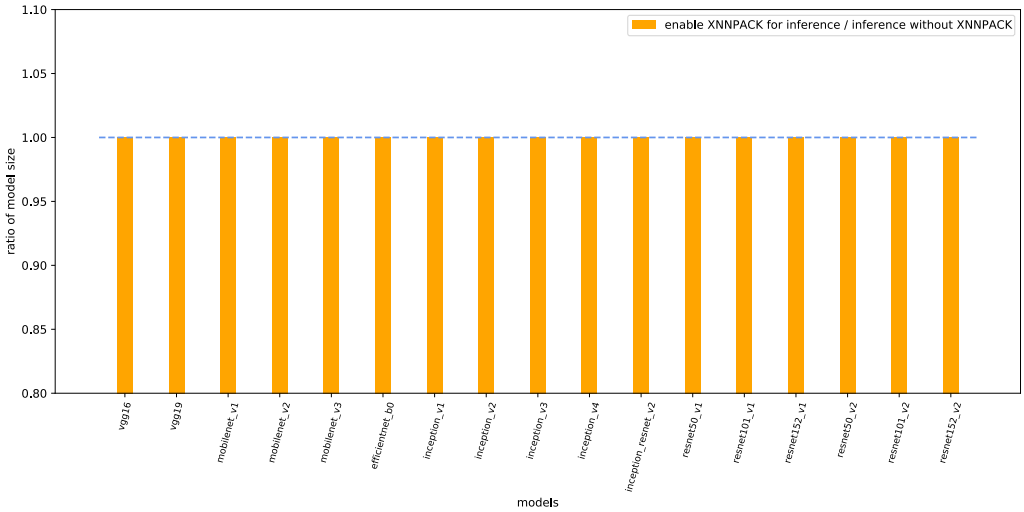

Figure D.1: Impact of XNNPACK on model size.

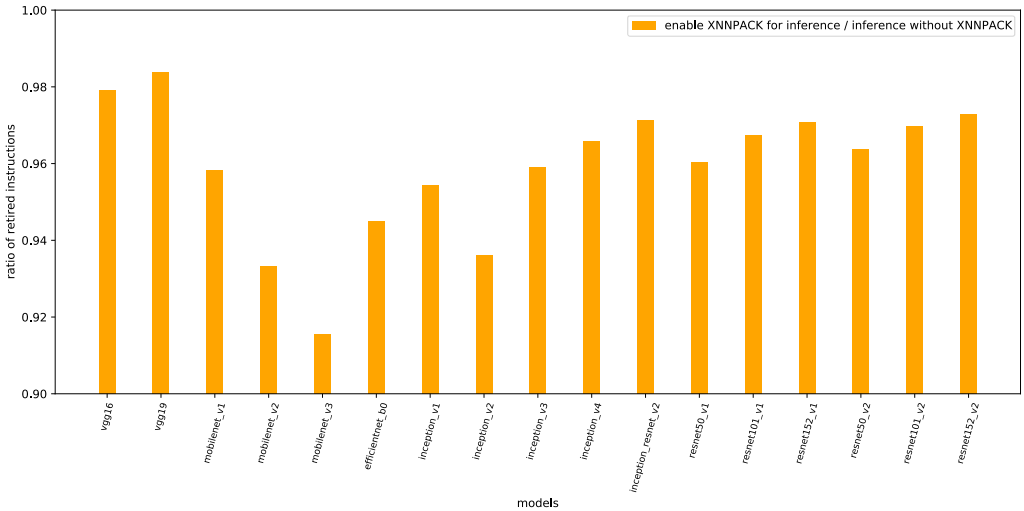

Figure D.2: Impact of XNNPACK on retired instructions.

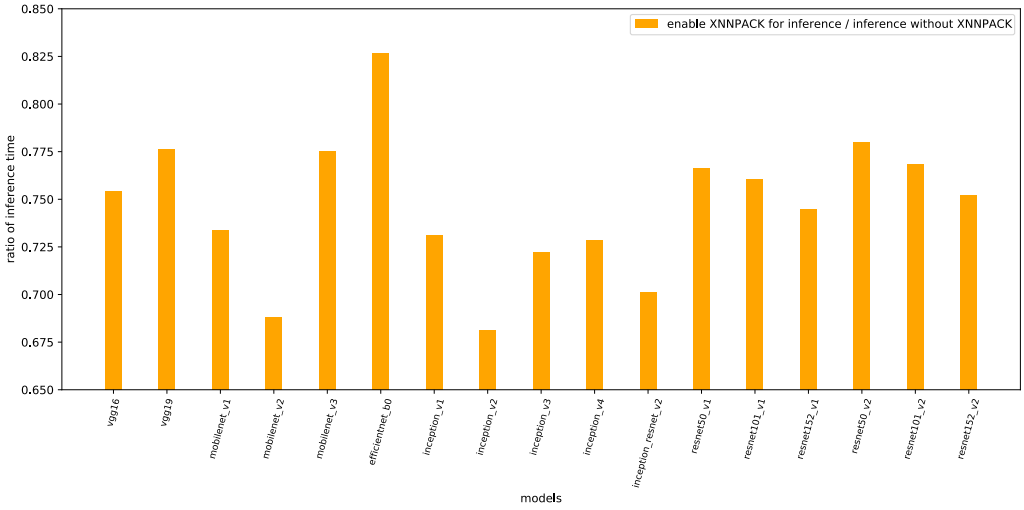

Figure D.3: Impact of XNNPACK on inference time.

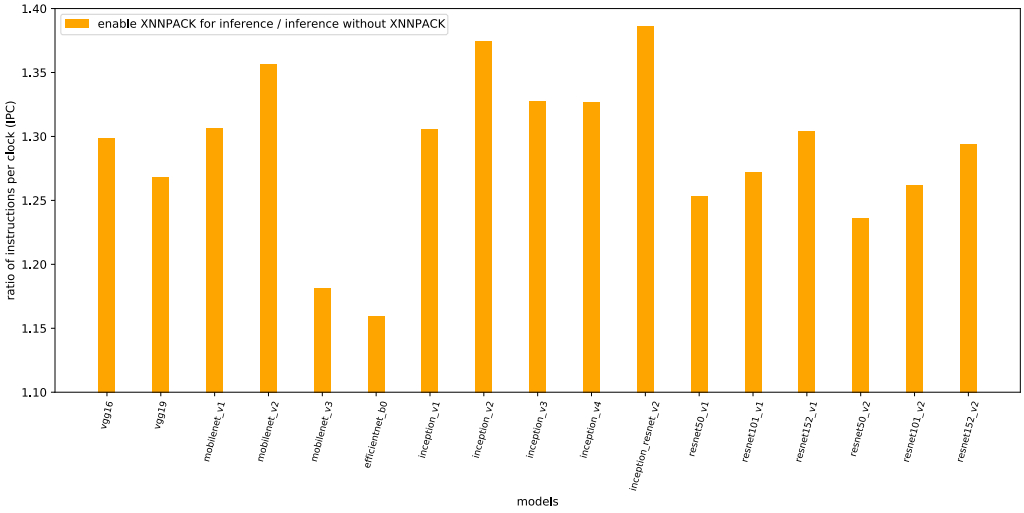

Figure D.4: Impact of XNNPACK on instructions per clock (IPC).

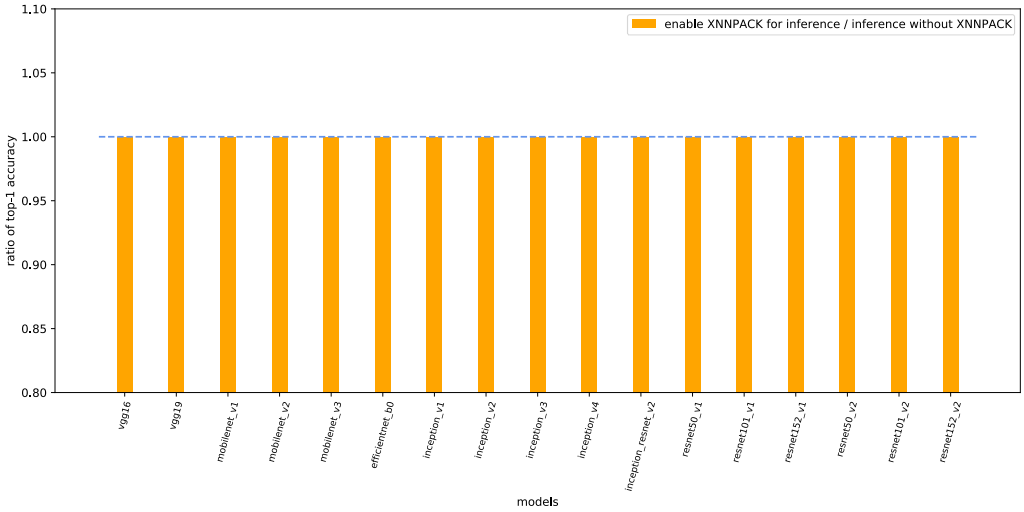

Figure D.5: Impact of XNNPACK on Top-1 accuracy.

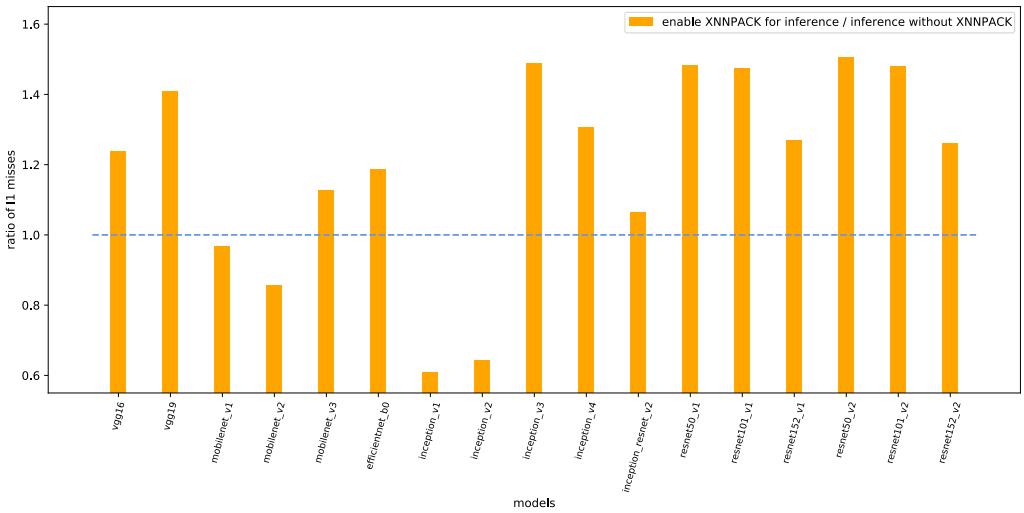

Figure D.6: Impact of XNNPACK on L1 misses.

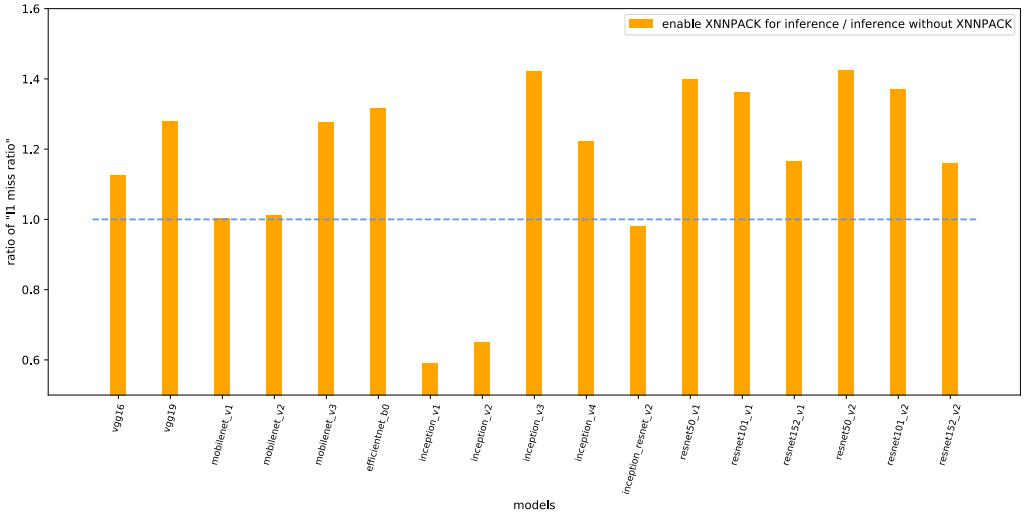

Figure D.7: Impact of XNNPACK on L1 miss ratio.

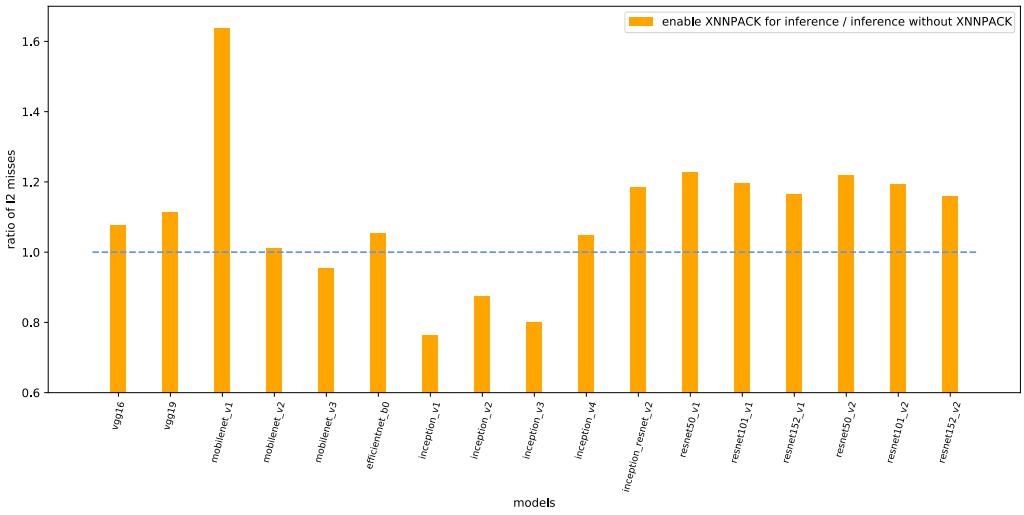

Figure D.8: Impact of XNNPACK on L2 misses.

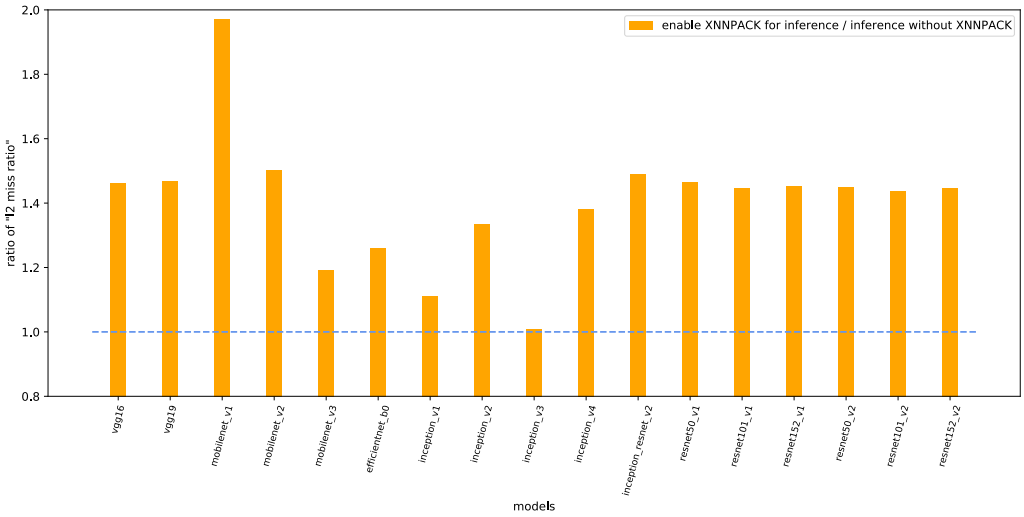

Figure D.9: Impact of XNNPACK on L2 miss ratio.

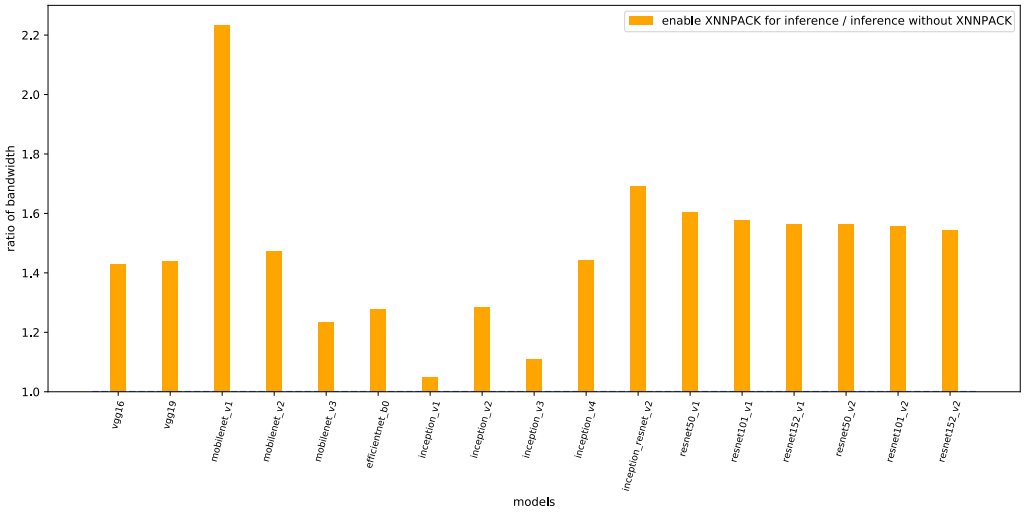

Figure D.10: Impact of XNNPACK on bandwidth.

Appendix E

Metrics w.r.t Models for Object Detection

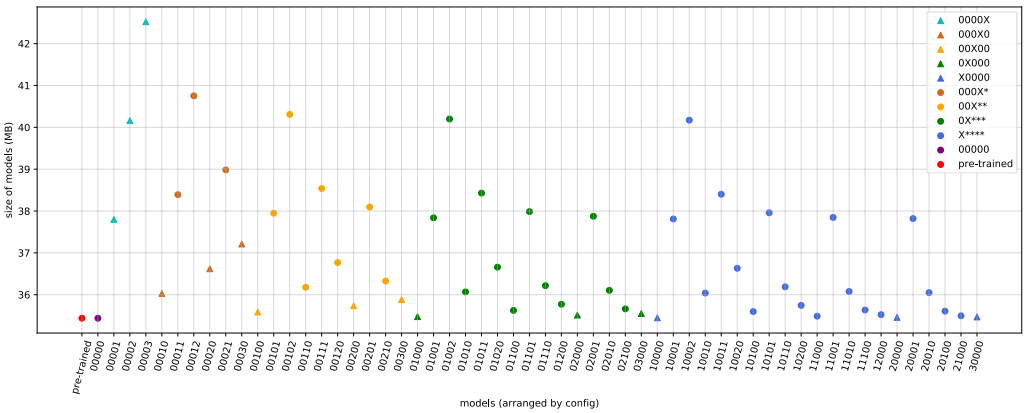

Figure E.1: Size of different YOLOv3-tiny models.

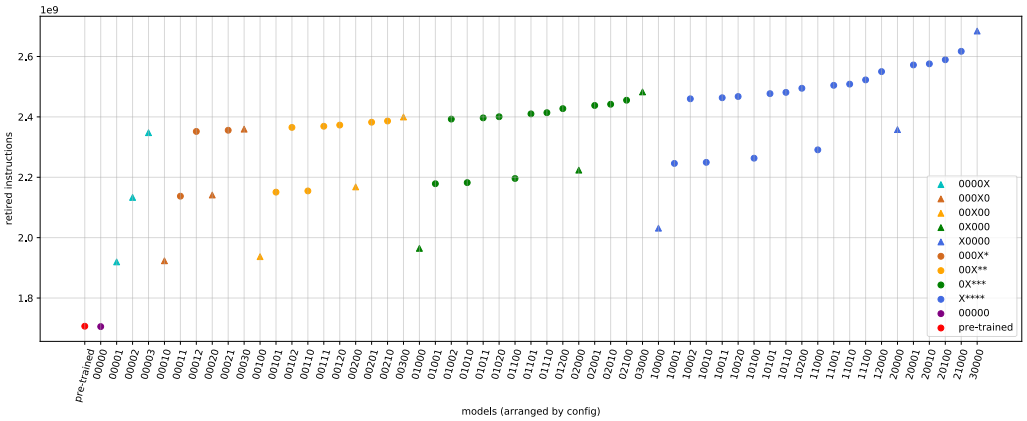

Figure E.2: Retired instructions of different YOLOv3-tiny models.

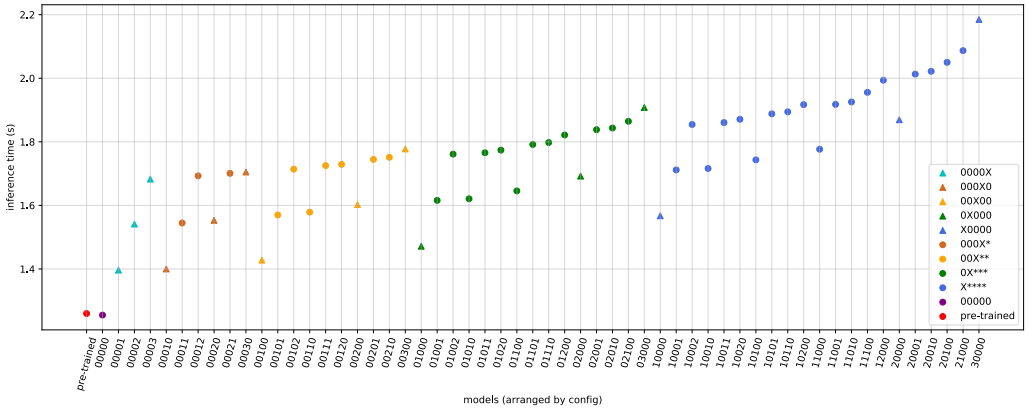

Figure E.3: Inference time of different YOLOv3-tiny models.

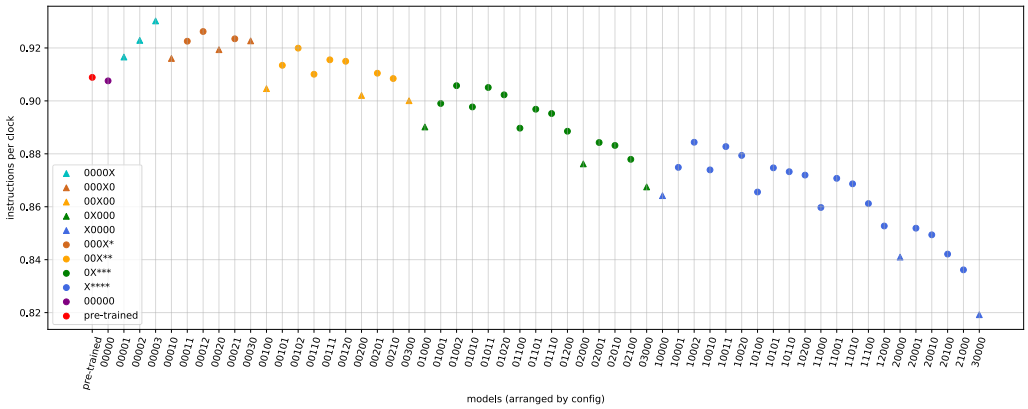

Figure E.4: Instructions per clock (IPC) of different YOLOv3-tiny models.

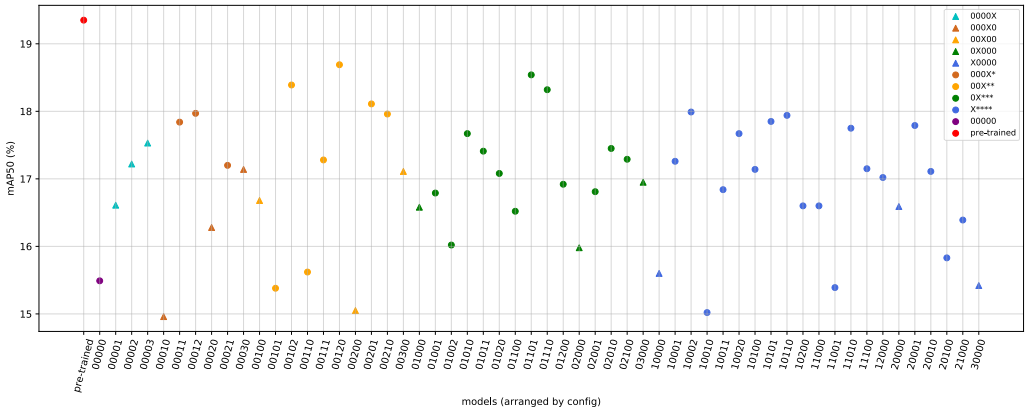

Figure E.5: mAP50 of different YOLOv3-tiny models.

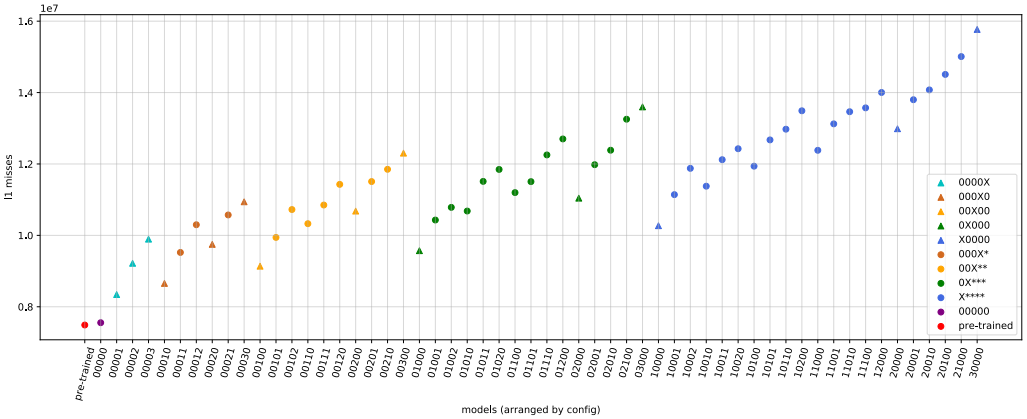

Figure E.6: L1 misses of different YOLOv3-tiny models.

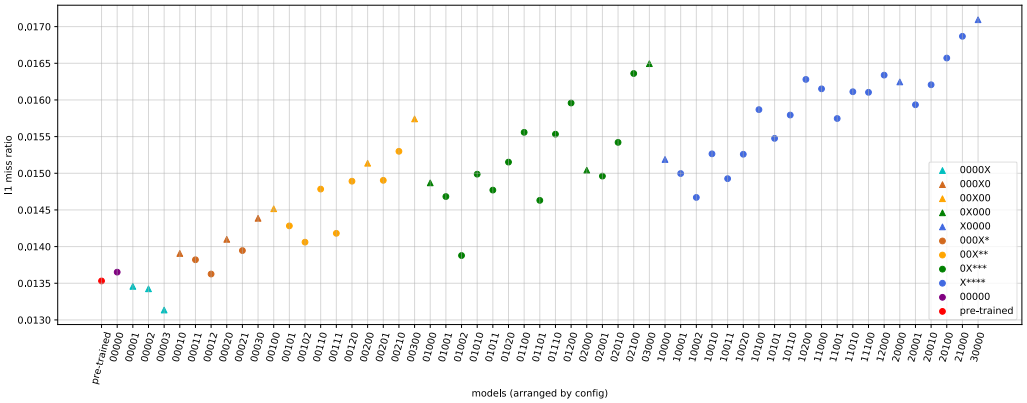

Figure E.7: L1 miss ratio of different YOLOv3-tiny models.

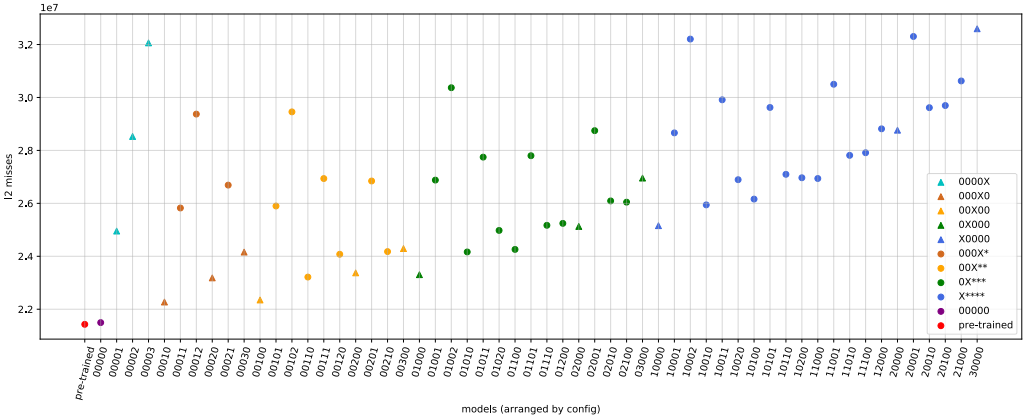

Figure E.8: L2 misses of different YOLOv3-tiny models.

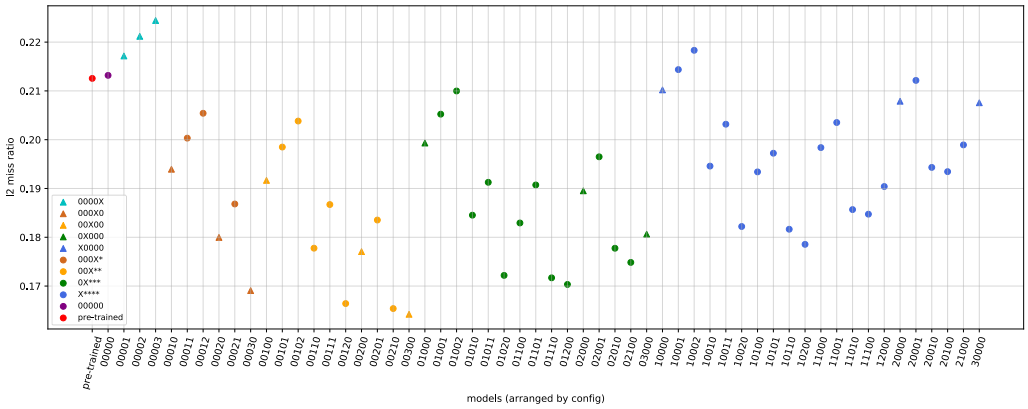

Figure E.9: L2 miss ratio of different YOLOv3-tiny models.

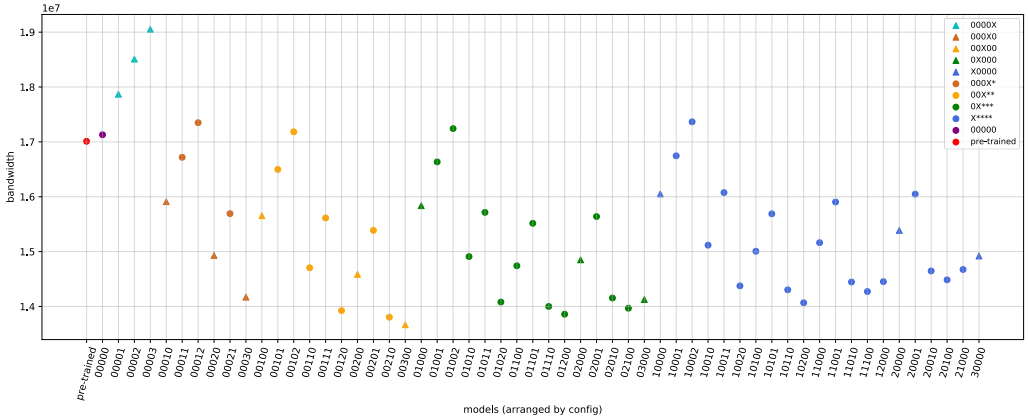

Figure E.10: Bandwidth of different YOLOv3-tiny models.

Acronyms

API Application Programming Interface.

CLI Command Line Interface.

CPU Central Processing Unit.

DNN Deep Neural Network.

FLOPS Floating Point Operations Per Second.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

IoT Internet of Things.

IOU Intersection over Union.

IPC Instructions per Clock.

LLC Last Level Cache.

mAP mean Average Precision.

NMS Non-Maximum Suppression.

OPS Operations Per Second.

PMC Performance Monitor Counter.

SWaP Size, Weight, and Power.

TF Lite TensorFlow Lite.

TPU Tensor Processing Unit.

Bibliography

[1] Ashfaq, S., AskariHemmat, M., Sah, S., Saboori, E., Mastropietro, O., and Hoffman, A. "Accelerating Deep Learning Model Inference on Arm CPUs with Ultra-Low Bit Quantization and Runtime". In: *arXiv preprint arXiv:2207.08820* (2022).

[2] Bianco, S., Cadene, R., Celona, L., and Napoletano, P. "Benchmark analysis of representative deep neural network architectures". In: *IEEE access* 6 (2018), pp. 64270–64277.

[3] Cheng, Y., Wang, D., Zhou, P., and Zhang, T. "A survey of model compression and acceleration for deep neural networks". In: *arXiv preprint arXiv:1710.09282* (2017).

[4] Chollet, F. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[5] Deng, L., Li, G., Han, S., Shi, L., and Xie, Y. "Model compression and hardware acceleration for neural networks: A comprehensive survey". In: *Proceedings of the IEEE* 108.4 (2020), pp. 485–532.

[6] Dong, S. and Kaeli, D. "Dnnmark: A deep neural network benchmark suite for gpus". In: *Proceedings of the General Purpose GPUs*. 2017, pp. 63–72.

[7] Elsken, T., Metzen, J. H., and Hutter, F. "Neural architecture search: A survey". In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1997–2017.

[8] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. "The pascal visual object classes (voc) challenge". In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[9] Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H. "A survey of FPGA-based neural network accelerator". In: *arXiv preprint arXiv:1712.08934* (2017).

[10] He, K., Zhang, X., Ren, S., and Sun, J. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[11] He, K., Zhang, X., Ren, S., and Sun, J. "Identity mappings in deep residual networks". In: *European conference on computer vision*. Springer. 2016, pp. 630–645.

[12] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. "Searching for mobilenetv3". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 1314–1324.

[13] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[14] Ioffe, S. and Szegedy, C. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[15] Karki, A., Keshava, C. P., Shivakumar, S. M., Skow, J., Hegde, G. M., and Jeon, H. "Tango: A deep neural network benchmark suite for various accelerators". In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2019, pp. 137–138.

[16] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[17] Lin, Y., Zhang, Z., Tang, H., Wang, H., and Han, S. "Pointacc: Efficient point cloud accelerator". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 449–461.

[18] Nicolella, M., Roozkhosh, S., Hoornaert, D., Bastoni, A., and Mancuso, R. "RT-Bench: An Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications". In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. RTNS 2022. Paris, France: Association for Computing Machinery, 2022, pp. 184–195. ISBN: 9781450396509. DOI: 10.1145/3534879.3534888. URL: https://doi.org/10.1145/3534879.3534888.

[19] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[20] Redmon, J. and Farhadi, A. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[21] Redmon, J. and Farhadi, A. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[22] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[23] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[24] Simonyan, K. and Zisserman, A. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[25] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. "Inception-v4, inception-resnet and the impact of residual connections on learning". In: *Thirty-first AAAI conference on artificial intelligence*. 2017.

[26] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[27] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[28] Tan, M. and Le, Q. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.

[29] Tan, M. and Le, Q. "Efficientnetv2: Smaller models and faster training". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 10096–10106.

[30] Zhang, Q., Zhang, M., Chen, T., Sun, Z., Ma, Y., and Yu, B. “Recent advances in convolutional neural network acceleration”. In: *Neurocomputing* 323 (2019), pp. 37–51.

Disclaimer

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Garching, December 23, 2022

(Signature)