**Kaunas University of Technology**

Faculty of Mathematics and Natural Sciences

# Utilizing Deep Learning Models for Image Analysis at Scale: Comparison of Deployment Solutions

Master's Thesis

**Renata Klimiato**

Author

Prof. Dr. Evaldas Vaičiukynas

Supervisor

Dr. Paulius Danėnas

Supervisor

**Kaunas, 2022**

**Kaunas University of Technology**

Faculty of Mathematics and Natural Sciences

# Utilizing Deep Learning Models for Image Analysis at Scale: Comparison of Deployment Solutions

Master's Thesis

Business Big Data Analytics (6213AX001)

**Renata Klimiato**

Project Author

**Prof. Dr. Evaldas Vaičiukynas**
Supervisor

**Dr. Paulius Danėnas**
Supervisor

**Doc. Dr. Tomas Iešmantas**
Reviewer

**Doc. Dr. Mayur Pal**
Reviewer

**Kaunas, 2022**

**Kaunas University of Technology**

Faculty of Mathematics and Natural Sciences

Renata Klimiato

# Utilizing Deep Learning Models for Image Analysis at Scale: Comparison of Deployment Solutions

Declaration of Academic Integrity

Renata Klimiato. Giliojo mokymosi modelių panaudojimas sparčiai vaizdų analizei: produkcinių aplinkų palyginimas . Magistro studijų baigiamasis projektas / vadovai: Prof. Dr. Evaldas Vaičiukynas,  Dr. Paulius Danėnas; Kauno technologijos universitetas, Matematikos ir gamtos mokslų fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Taikomoji matematika.

Reikšminiai žodžiai: vaizdų klasifikavimas, gilusis mokymasis, konvoliuciniai neuroniniai tinklai, modelių paleidimo platformos, kvantavimas.

Kaunas, 2022. 78 p.

## Santrauka

Kompiuterinės regos (angl. *computer vision*) algoritmai aktyviai tobulinami ir pasiekė aukščiausią piką per pastarąjį dešimtmetį. Didėjant apmokytų modelių perkėlimo į produkcinę aplinką poreikiui, dirbtinio intelekto sistemų architektams tampa svarbu paruošti modelio paleidimo aplinką taip, kad išgautų efektyviausią modelio veikimą.

Šiame darbe apžvelgiami vaizdų klasifikavimo metodai, architektūros, modelių pateikimo produkcinėje aplinkoje procesas ir programinė įranga. Taipogi, pateikiama lyginamoji specifikacija ir eksperimentų, atliktų naudojant „EfficientNet“ ir „MobileNet“ šeimos modelius, rezultatai, siekiant palyginti kelias programinės įrangos, skirtos modelių paleidimui į produkcinę aplinka, platformas: „TensorFlow Serving“, „TorchServe“ ir „Triton Inference Server“. Papildomai apžvelgiama modelio kvantavimo įtaka resursų išnaudojimui bei modelio veikimo laikui.

Rezultate, „Triton Inference Server“ platforma parodė 16 kartų greitesnį modelio veikimą, lyginant su „TorchServe“. Papildomai apžvengtos valandinės debesijos paslaugų kainos lyginant modelių veikimą naudojant „TensorFlow Serving“ ir „TorchServe“. Galiausiai, pateikiamos rekomendacijos dėl efektyvaus vaizdų klasifikavimo modelio veikimo produkcinėje aplinkoje.

Renata Klimiato. Utilizing Deep Learning Models for Image Analysis at Scale: Comparison of Deployment Solutions. Master's Final Degree Project/supervisors: Prof. Dr. Evaldas Vaičiukynas, Dr. Paulius Danėnas; Faculty of Mathematics and Natural Sciences, Kaunas University of Technology.

Study field and area (study field group): Applied Mathematics.

Keywords: image classification, deep learning, convolutional neural networks, model deployment, model serving, REST endpoint, quantization.

Kaunas, 2022. 78 pages.

## Summary

Computer vision algorithms have been actively developed, with the highest peak during the last decade. With the increasing need for a phase of transferring pre-trained models to the production environment, it becomes important for architects of artificial intelligence systems to assess the inference environment for reaching the most effective model performance.

In this paper, we review image classification task, architectures, model serving process, and deployment software. Furthermore, we present benchmark specification and experiments results performed using EfficientNet and MobileNet family models with the purpose of comparing three model serving software: TensorFlow Serving, TorchServe, and Triton Inference Server. Additionally, model quantization impact on experiments inference time was reviewed.

As a result, Triton Inference Server showed 16 times faster performance compared to TorchServe. Additionally, cloud instances' hourly costs were reviewed when comparing TensorFlow Serving and Triton Inference Server model's performance. Lastly, recommendations for efficient image classification model inference in production were provided.

## Table of Content

## List of Tables

**List of Figures**

## List of Abbreviations and Terms

**Abbreviations:**

CNN - Convolutional Neural Network;

CPU - Central Processing Unit;

EF - EfficientNet-B7 (abbreviationused in experiments images);

GPU - Graphics Processing Unit;

IDE - Integrated Development Environment;

ML – Machine Learning;

MN – MobileNetV3-large (abbreviation used in experiments images);

MN_Q – Quantized MobileNetV3-large;

NN - Neural Network;

OS – Operating System;

RAM - Random-Access Memory.

**Terms:**

**Artificial intelligence -** system's capability to interpret external data and use it for learning with the purpose to retrieve the knowledge and execute defined objectives based on it;

**Benchmarking** – comparison of performance metrics;

**Convolutional Neural Network** – in deep learning, it is a class of neural networks with the purpose to analyze visual data based on human visual cortex operation principles ;

**Deep Learning** – a type of machine learning based on neural networks ;

**Docker** – is an operating system-level virtualization platform based on containerization principles;

**Inference** – a phase of the model lifecycle in the production environment when the model makes predictions based on live data and produces results;

**Machine Learning** – artificial intelligence method having the purpose to identify patterns in input data and learn based on it to make human-like decisions;

**Model Serving** – a process of deploying a model to production with the purpose to make an AI solution available for incorporating it with client applications;

**Model Warmup** – model adjustment with the purpose to reach its optimal performance state, usually performed by making a sample inference request to "warm-up" the model  before launching it for production use;

**Neural Network** – a set of computer algorithms inspired by biological neural networks;

**TensorFlow Serving** – TensorFlow software for machine learning model serving, designed for production environments;

**TorchServe** – PyTorch software for machine learning model serving, designed for production environments;

**Triton Inference Server** – NVIDIA software for serving machine learning models, designed for production environments;

**Quantization** – model compression method, reducing model size with a slight impact on model accuracy.

# Introduction

**Relevance of the topic.** Computer vision is one of the main areas of artificial intelligence, actively developing with the highest peak of activity over the past decade. The task of image recognition is very important since the possibility of automatic image recognition by a computer brings many new opportunities in the development of science and technology. Well-known examples include object recognition, photographs, and video footage analysis, having high demand in the medical field, quality control of manufactured products without human intervention, automatic transport control, and many others. This kind of machine learning has been highly developed in recent years due to a significant increase in the computing power and the widespread use of graphics cards for computing, which allows you to train neural networks of much greater depth and complex structure than before, which, in turn, show significantly better results compared to other algorithms, especially if research field is related to image recognition. This area of neural networks is called deep learning.

To survive in a highly competitive business environment, today's business must be flexible, agile, and adaptive.  As deep learning applications become more popular, there is an increasing need for a phase of transferring pre-trained models to the production environment. There are several technologies that make this step easier, and perhaps the best-known solution today is TensorFlow Serving. However, when it comes to decisions regarding choosing the most suitable model serving platform, there are many things to consider while matching technology to infrastructure, technical stack, model complexity, use cases, performance requirements, or the need of additional features for functionality scaling in the future.

**Research problem**. With the advent of more and more technologies for serving to production stage, it becomes important for architects of artificial intelligence systems to compare them in terms of various aspects, such as speed, CPU and GPU resource utilization, ease of configuration, and additional benefits.

**Research subject.** Deep learning models’ serving platforms.

**Research goal.** Recommendations for efficient image classification inference at scale.

**Research tasks.** The following objectives have been set to achieve the research goal:

1. Research literature on the application of deep learning, its relevance, and its impact in multiple industries.
2. Review the model serving process, platforms, and related research.
3. Choose and describe novel architectures used for image classification.
4. Perform benchmarking by serving pre-trained models to the production environment via different platforms and compare the results.
5. Compact models through quantization and perform additional benchmarking.
6. Provide general recommendations or guidelines for selecting the right technology for efficient image classification deep learning model serving.

**Work structure.** The first part of the work analyzes literature reviewing deep learning, its relevance, model serving process, and technologies. The second part presents computer vision tasks, model architectures, inference system challenges, and model deployment software specifics. The third part presents deployment software benchmarking scenarios, data used for model inference, and collected results.

## 1. Literature Review

### 1.1. The Concept of Deep Learning

Deep learning is a well-known and widely discussed method for building data-driven decision-making algorithms, better known as a type of artificial intelligence and machine learning. Its core concept was derived from the structure of biological neurons existing in the human brain. Its purpose is to analyze data and make predictions based on it helping the human to identify objects he sees and make decisions in everyday situations.

Large and deep artificial neural networks are major components of deep learning. The input layer of it takes signal vectors and then one or several hidden layers process the outputs of the previous layer [1,2] (see Figure 1). "One of the main differences from traditional machine learning methods is that deep learning automatically learns how to represent data using multiple layers of abstraction" [5].

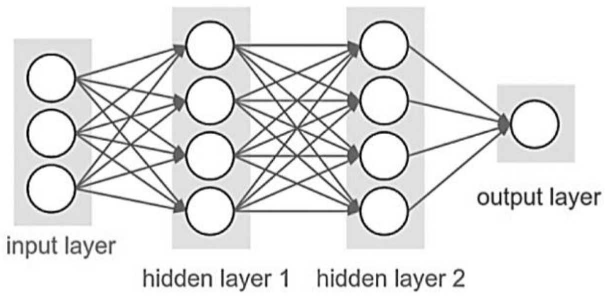


**Figure 1.** A three-layer artificial neural network [7]

The initial concept of a neural network can be traced back to 1943, when two scientists, neurophysiologist Warren McCulloch and mathematician Walter Pitts, wrote a paper in which they described how biological neurons might work. To describe it, they modeled a simple neural network using electrical circuits [3]. Later, multiple researchers were working on different neural network simulation approaches (in both - computational and mathematical sciences), however all of them faced a struggle with the neural networks being relatively slow.

Due to the limitation of processors, neural networks took weeks to learn, so the next chapter in history was creating digital, analog, and optical chips to help optimize the application of neural networks [3]. However, neurons in the human brain actually do not act as a digital signal and is more similar to analog one - there is no distinct state like 1 or 0 to describe true or false, on the contrary, analog signals vary between 0 and 1 values acquiring intermediate values. Therefore, this part of our biology was covered in deep learning research (see Figure 2).

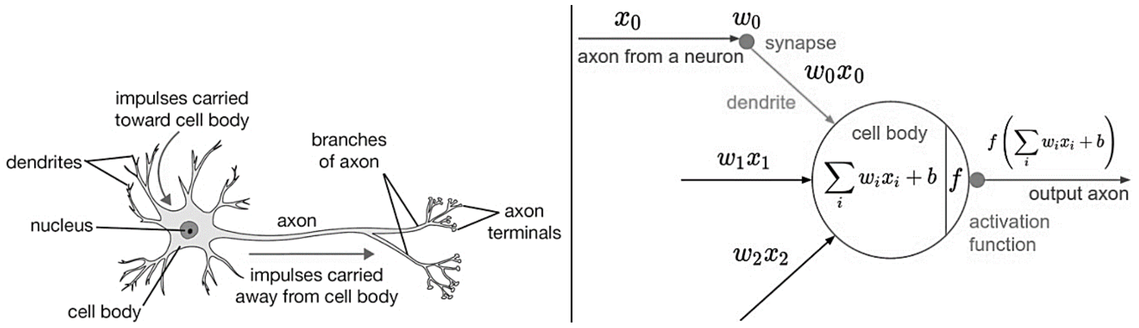


**Figure 2.** Biological neuron (left) and its mathematical model (right) [7]

## 1.2. Applications of Deep Learning

Today, as we have more data and much more powerful computers, the capabilities of large and deep neural networks have increased. "The main feature of the methods based on machine learning is extracting the characteristics from within a large number of data without human involvement" [4]. Due to this fact, deep neural network-based solutions have become dominant in solving loads of problems, even in various fields such as the IoT industry, agriculture, and so on. Deep learning found a way to impact almost every sector of business in its own way and researchers are still trying to unravel its full potential [6].

Healthcare is known as a sector having a wide range of deep learning use cases such as computer-aided disease detection, genome analysis, the discovery of medicines, medical imaging, etc. Most solutions are built for healthcare professionals to improve time-consuming processes, automate measurements, and identify anomalies in images or other medical records [7]. For example, MRI image analysis helps to see tumors or other unusual changes in human body tissues and predict a patient's condition/disease based on the medical imaging and diagnosis of previous patients [8].

Another example is the discovery of new drugs based on the exploration of plant features or reprogramming of decease cells. The creation of the SARS-CoV-2 vaccine was also based on deep learning models such as Generative Adversarial Networks to create data-oriented molecules [9]. Its approaches have also revolutionized the field of cancer vaccinology through the improved prediction of neoantigens and their HLA binding affinity [10]. As we see, deep learning is reshaping the health care industry by delivering new possibilities to improve people's lives.

Deep learning is used in entertainment industries, such as movie making, computer games, social media, and others. Amazon, Netflix, and Vevo use a recommender system to provide a personalized experience to their viewers using their show/movie preferences, time of access, history, etc. [11]. Instead of forcing the user to dig for the information, deep learning starts doing that for the user. Transitioning to virtual worlds, they arise from augmented reality applications (e.g., Pokémon Go), video conferencing, and games, which brought us various degrees of digital transformation [12]. In addition, the term "Metaverse" was brought about by Mark Zuckerberg with the idea of building a new kind of virtual world connecting people in personal and business life. Users would be able to participate in events that occur in virtual environments enriched with sounds and objects built by deep learning algorithms [13].

Sounds generation is another large part of the industry that blends right into previously discussed use cases. Sound recognition technologies can be used to train deep learning models to produce music compositions by learning patterns and generating new sounds. Google's Wavenet and Baidu's Deep Speech software can train a machine to learn the patterns and statistics that are unique to music and generate a completely new composition [11]. Furthermore, Oxford and Google scientists created a neural network called LipNet that could read people's lips with 93% success [14]. This can be used to add a soundtrack to silent movies or allow surveillance equipment to pull the context from people's conversations happening far away if the camera can catch the lips' movements.

Another great example of deep learning use cases is the Wimbledon championship administration's initiative to cooperate with IBM and use the IBM Watson platform to analyze player emotions and expressions through hundreds of hours of footage. "The system allowed Wimbledon to produce highlights packages for fans to relive or catch up on matches much more quickly than traditional manual techniques" [15].

Deep learning is widely used in robotics to make robots perform human-like tasks. Machines are built to understand the world around them, and they are trained to make better and safer decisions without supervision. Replacing humans with robots in the right place and timing would increase safety, for example, when nuclear reactors have melted down after a tsunami in Japan - using robotics for environmental clean-up would be the wiser solution rather than sending human beings into a radioactive environment. Deep learning boosts robots with life experiences, for example, robots by Boston Dynamics react to people pushing them around, get up when they fall, can unload a dishwasher, and do other tasks as well [16].

The extensive deployment of deep learning architecture has pervade the global business landscape. In the present time, advanced technologies are making a breakthrough that impacts human daily life. The use cases provided in this chapter should not limit our imagination, as deep learning became an extremely active area of research. Despite the complexity of this field, we will try to overview a couple of major deep learning tasks used in computer vision.

## 1.3. Model Serving

The rise of deep learning has been driven forward by the uncontrolled amount of model training material, the use of accelerators such as graphic processing units, and the advancement of ML models. Training deep learning models became time-consuming, especially when using an enormous amount of visual input. For this reason, it requires both - specialized hardware and software to run models on production effectively.

To understand the ML infrastructure, we can segment the whole workflow into three rough parts: data preparation, model building, and production deployment (see Figure 3). Once a model has been developed, trained, and validated - it needs to be prepared for inference.

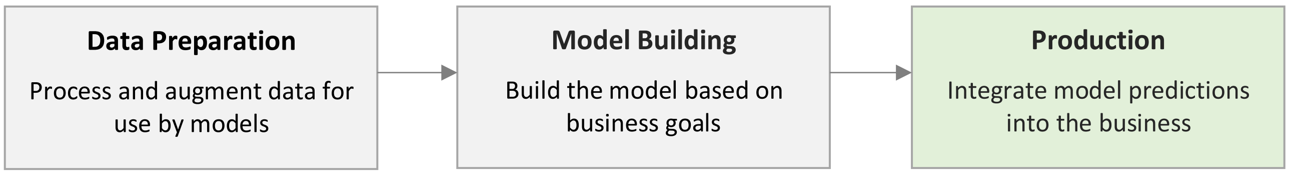

Data Preparation
Process and augment data for use by models
Model Building
Build the model based on business goals
Production
Integrate model predictions into the business

**Figure 3**. High-level ML infrastructure workflow

The model serving step allows the model to be exposed as an API via HTTP or gRPC, allowing it to integrate into client applications and perform model inference in a production environment. Model serving is a composite process that requires a set of analytical, engineering, and machine learning infrastructure skills [17]. Tasks such as model conversion, preparation of deployment configuration files and setting up of the serving environment are needed. Furthermore, serving is not a one-time task, as the model requires also a continuous learning process to be set up [18].

Existing deep learning frameworks have weak performance for online serving: "many models suffer from long serving latency and high cost, preventing their deployment to production" [19]. "Latency and efficiency are the two most important metrics for serving. Interactive services often require responses to be returned within a few or tens of milliseconds because delayed responses could degrade user satisfaction and affect revenue" [19]. One of the examples might be a self-driving car which must be able to detect and respond within milliseconds in order to avoid an accident. Another example related to efficiency can be a battery operated drone whose main purpose is to follow a target on the ground, it must be energy efficient to maximize flight time [20]. These optimization and performance requirements must be taken into account when preparing the model for use in production, and one of the major decisions to make is selecting a serving platform based on the company and customers' needs.

## 1.4. Model Benchmarking

Faced with this growing demand for deep learning inference deployment, many companies and startups are engaging to develop customized inference hardware, software, and optimization tools [21]. Consequently, the countless combinations of hardware and software make accessing future system performance and cost quite challenging [22]. The spectrum of machine learning tasks, models, methods, training, external data, packages, optimization libraries, neural network architectures, and serving platforms makes the evaluation of inference performance nearly impossible to predict [22].

When performing an inference benchmark, Zhang emphasizes a couple of key aspects [23]:

1. Model complexity and execution features impact deep learning system performance, which makes it important to perform benchmarking using the same models. Figure 4 shows how accuracy and computational complexity differ between different models that perform the same task, image classification.
2. There are “four evaluation scenarios reflecting production use cases: single-stream, multi-stream, server, and offline solutions” [23]. Performance may vary significantly under these scenarios.
3. Benchmarking must have defined model-quality targets. Quality and performance are inseparable from each other, however, sometimes companies sacrifice model quality to reduce cost or latency.
4. The need for benchmarking should be identified to define rules of software and hardware capabilities. Some companies need to include semantic-level benchmarks to compare a variety of languages and libraries, others need to test inference performance dependency from hardware.
5. Strict rules must be applied to change the model: whether the model will be benchmarked as is or should it be allowed to enrich the model and demonstrate different performance and quality targets [23].

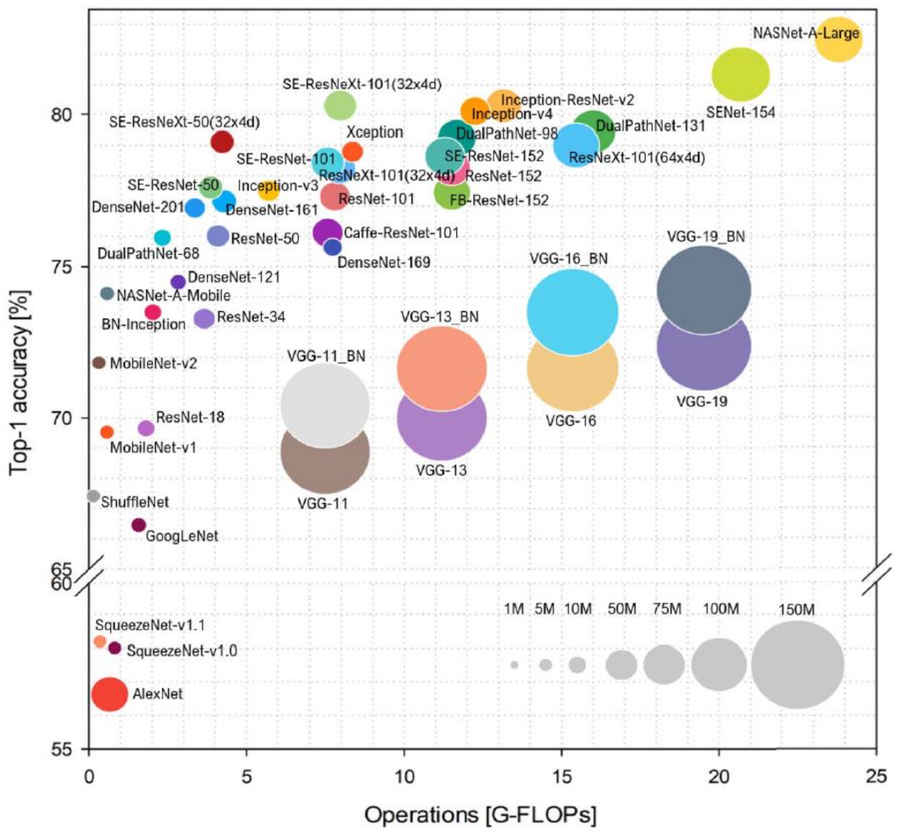


**Figure 4.** Machine learning models diversity for image classification

With an increasing number of deep learning model applications, more and more researchers investigate multiple platforms for inference serving. Even when we look at the image classification task, there may be the need to recognize a pedestrian's identity from video footage or process the photo uploaded via smartphone. Both of these tasks may have different quality requirements, real-time data processing demand, and even operations performed by the model might be framework-specific – all of it adds complexity and requires corresponding features from model serving platforms. Figure 5 shows a variety of tools combinations across layers that make benchmarking inference systems complex.

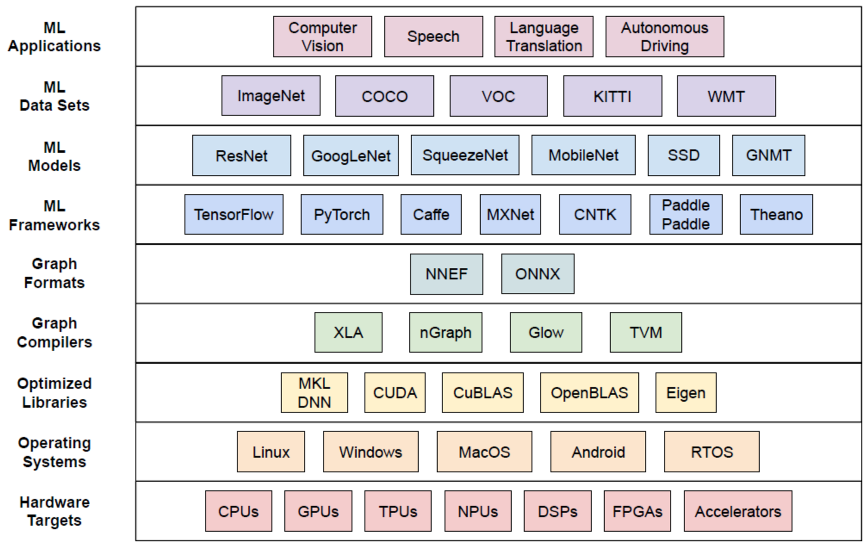


**Figure 5.** The diversity of options at every level of the stack

Working with big data adds extra concerns about infrastructure as production-ready systems have a high demand for technical resources [24]. When we look at deep learning systems, data partly replaces the code, as a learning algorithm automatically identifies the patterns and learns from them. Such machine learning systems usually are distributed across multiple machines and rely on various pipelines implemented in different languages, glued up from several libraries or packages. While analyzing software engineering challenges of deep learning, Arpteg described that the most important component for good model performance in production is hardware, as GPUs provide from 40 to 100 times speedup in comparison with CPUs. Furthermore, over the past couple of years, the performance of the models has improved significantly as new GPUs are released one or more times a year. On the other hand, maintaining a system in production with new hardware require special techniques and knowledge. Many problems may be avoided if the model runs on cloud-based solutions. Zhang adds that current serving platforms lack configurability and leave implementation details to development teams, taking them days or weeks to make inference benchmarks [23].

## 1.5. Model Serving Software

Once a model is ready, it should be served for use in production. There are a few patterns to achieve at this point [25]:

– **Embedded model**: this approach states that our model is built and packaged as a part of business consuming application. From this point, we treat "application artifact and version as being a combination of the application code and the chosen model" [25];
– **Model deployed as a service**: this approach means the model is  deployed independently as a microservice and may be used separately from consuming applications. It means we can also integrate this model into multiple consumer applications without changing model logic. However, such an approach may add latency at inference time;
– **The model published as a data**: in this approach, model is also deployed independently, however, the consuming application will ingest it as data runtime. This approach is mostly used in streaming scenarios where the application provides a constant flow of data/events [25].

The simplest approach of embedding the model is supported by most serving tools, however, other patterns require custom specifics. For example, in implementing the "model as service" pattern, multiple cloud providers suggest separate tools and SDKs "to wrap your model for deployment into their MLaaS (Machine Learning as a Service) platforms, such as Azure Machine Learning, AWS Sagemaker" [25]. Also, the same pattern can be solved with the open-source tool -  Kubeflow, however, it tries to solve more impediments that the model serving, which may add complexity. In general, model serving tools can be divided into these types [26] :

– Internally built executable (PKL File/Java) - containerized and non-containerized;
– Cloud ML provider - Amazon SageMaker, Azure ML;
– Batch or Streams oriented software (both, cloud and on-premise) - Algorithmia, Databricks, Paperspace;
– Open source - TensorFlow Serving, Kubeflow, Seldon, Anyscale, etc.

However, there are many considerations when deciding what software to choose for model serving, as there are a large number of different tools performing similar tasks (see Table 1). For example, when an organization's data security requirements are very strict - the easiest way to go would be using solutions with integrated security-specific features or building your own proprietary solutions. Some providers like Algorithmia, Seldon, TensorFlow, or Kubeflow may be an option. Another example to consider may be whether a team will manage the solution for model serving by themselves or not. If an organization has a low IT stack, then managed cloud solutions, such as Google ML, Azure ML, and Amazon SageMaker, would be a better choice, on the other hand, solutions like Kubeflow, TensorFlow Serving, Nvidia Triton, Anyscale would work for more technical organizations [26]. Also, we should not forget performance requirements, as each of these serving solutions interacts differently with the technical stack and models themselves, which may cause either better or worse performance when a model is running in the production environment.

On the other hand, most of these serving platforms are just one part of the whole software provider's tools ecosystem. When looking at such companies as TensorFlow, Nvidia, or Amazon – we know they are building more software than only serving platforms. There is a whole family of applications supporting the machine learning process starting from the very beginning (data preparation, model creation, training, etc) till the very end (model deployment, inference, monitoring, etc). Sometimes it is worth taking into account the usage of other tools to optimize models' performance before or during model inference as such capabilities are taken into account by these giant companies. For example, TensorFlow Serving software itself is very limited, it provides model versioning and serving features, allows easily update model weights in production, etc., however, multiple TensorFlow tools can be integrated with TensorFlow Serving and help to optimize inference or ease the deployment. There is TensorFlow Core which may help to manage data pipelines, prepare input data for inference, and postprocess inference output to the desired structure. Also, TensorFlow Lite dedicated to small scale, mostly mobile applications models optimization, has its use cases when needed. On the other hand, Nvidia Triton provides a client, which manages all features, needed to start using deployed model in production right away. Unlike TensorFlow Serving, Triton requires input and output processing, as well as model inference configuration, to be provided during model deployment. It also comes with easy batching configuration and other perks. Most companies try to build an ML system that allows multiple teams (analysts, engineers, DevOps) to cooperate on ML projects. All these tools come with some pros and cons, but based on business problems and the company's technical stack, some disadvantages for one company may become an advantage to another one.

To address all the above-mentioned issues and processes, a lot of MLOps tools are created to support the whole ML lifecycle in production. The table below refers to the most common tools used to manage the MLOps process [27]. The challenge here was to make sure that relevant factors would be considered when making a decision on which tool to use, as all tools have tens or hundreds of features, we decided to group them into the following categories:

1. Data and pipeline versioning: version control for datasets, features, and their transformations represents a snapshot of the input data including changes in structure and sampling [25,27];
2. Model and experiment versioning: usually separate teams work on building model and application, they use different tools and follow different workflows, for that reason it becomes hard to automate the whole process in production. There are more artefacts to be managed beyond the code, and versioning them is not straightforward [25]. Due to this reason, model/experiment versioning becomes a crucial task to maintain production model stability;

3. Model deployment: It allows transforming a model to service and accessing it using REST API mostly via HTTP, but there are also alternatives such as gRPC (which is used in Tensorflow Serving, Nvidia Triton, etc) or GraphQA [28];
4. Metrics and monitoring: tracking deployed models' ongoing performance. The metrics modules contain a series of model, performance, and implementation evaluation metrics that can later be accessed via separate REST APIs or pushed to monitoring tools for further analysis. Metrics are highly important for production models as they allow us to identify their flaws and make an improvement based on them [28];
5. Batches and/or streams support: identifying whether the tool supports real-time or batch predictions.
6. Integrity with other ML tools: usually ML project architecture consists of multiple ML tools, however, not all of them work with each other;
7. Coverage of libraries: integrating the serving tool into existing/new solutions or adding additional features may require the support of external ML/client frameworks;
8. On-premise vs cloud: identifying whether a tool can be run on the client's servers or it's mandatory to use cloud infrastructure which may cost additional expenses;
9. Product support: Github ratings and contributions allow us to evaluate how well the stack is maintained and should we rely on open-source or commercial software.

**Table 1.** Coverage of MLOps tasks in various ML model serving tools (2021 data)

| **Software** | **Data and Pipelines Versioning** | **Model and Experiment versioning** | **Deploy-ment** | **Metrics and Monitoring** | **Batches / Streams** | **Coverage of libraries** | **On-premise vs Cloud** | **Product Support** | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | **GitHub Stars** | **Contri-butors** |
| Amazon SageMaker | ✗ | ✓ | ✓ | CloudWatch | ✓ | Scikit-learn; Conda; Pip; MXNet; Spark; Chainer; TensorFlow; PyTorch; R; | cloud | 7100 | 520 |
| Triton | ✗ | ✓ | ✓ | Prometheus | ✓ | TensorFlow; PyTorch; ONNX; TensorRT | on-premise | 2333 | 61 |
| TensorFlow Serving | ✗ | ✓ | ✓ | Prometheus | ✓ | XGBoost; PyTorch; Keras; TensorFlow Core; | on-premise | 5100 | 178 |
| Torchserve | ✗ | ✓ | ✓ | Prometheus , Grafana | ✓ | PyTorch; Keras; Scikit-learn; Jupyter; MLflow, JMeter, Apache Bench, ORT, IPEX, TensorRT, FasterTransformer | on-premise | 2600 | 87 |
| DeepDetect | ✓ | ✓ | ✓ | ✗ | ✓ | XGBoost; Jupyter; TensorRT; Ncnn; Libtorch; TensorFlow; T-SNE; Dlib; Caffe; Caffe2; Java; C#; | on-premise | 2200 | 23 |
| Weights & Biases | ✓ | ✓ | ✓ | WandB | ✓ | Scikit-learn; Jupyter; XGBoost; LightGBM; TensorFlow; Keras; PyTorch | cloud | 3081 | 72 |
| Kubeflow | ✗ | ✓ | ✓ | Kubeflow Pipelines | ✓ | Scikit-learn; Jupyter; XGBoost; TensorFlow; PyTorch; MXNet; Chainer | cloud | 10400 | 229 |
| MLflow | ✗ | ✓ | ✓ | MLflow Tracking | ✓ | Scikit-learn; Conda; R; Java; Sagemaker; Spark; XGBoost; LightGBM; TensorFlow; Keras; PyTorch; ONNX | on-premise | 9700 | 308 |
| Azure Machine Learning | ✓ | ✓ | ✓ | Azure Machine Learning | ✓ | Onnx; Scikit-Learn; MLflow; MXNet, Keras, XGBoost, TensorFlow; Chainer; PyTorch; Jupyter; R; Dask; | cloud | 2400 | 52 |
| Polyaxon | ✗ | ✓ | ✓ | Polyaxon | ✓ | Scikit-learn; TensorFlow; Keras; Caffe; PyTorch; MXNet; | on-premise / cloud | 2900 | 83 |
| H2O MLOps | ✗ | ✗ | ✓ | H2O MLOps | ✗ | R; Java; Spark; XGBoost; LightGBM; TensorFlow; PyTorch | cloud | 5400 | 143 |
| Algorithmia | ✓ | ✓ | ✓ | Datadog, Arize, Arthur, Telegraf, InfluxDB, New Relic | ✓ | AllenNLP; Caffe; Chainer; Dlib; Gensim; H2O.ai; Keras; MXNet; NLTK; ONNX; PyTorch; Scikit-Learn; TensorFlow; XGBoost; | cloud | 195 | 22 |
| Seldon | ✗ | ✗ | ✓ | Prometheus | ✓ | Scikit-learn; R; Java; Spark; TensorFlow | on-premise | 2400 | 115 |
| Hydrosphere | ✗ | ✗ | ✓ | Hydrosphere | ✗ | Scikit-learn; R; Java; TensorFlow; Keras; PyTorch | on-premise | 223 | 18 |
| Paddle Serving | ✗ | ✓ | ✓ | Prometheus | ✓ | Caffe; TensorFlow; ONNX; PyTorch; Intel MKLDNN; Nvidia TensorRT; C++; Python; Java language SDK; | on-premise / cloud | 618 | 33 |

## 1.6. Serving Software Benchmark

Deep learning solutions are highly desirable in the current business world and it seems that a huge amount of research should be performed regarding which software is better to use for solving specific business problems. Indeed, most of the research covers model preparation part and serving tools' adaptability to existing ML tools, and only a small part of the research compares serving software with each other.

Automatic benchmarking tools such as MLPerf, InferBench, or AI Benchmark were created during the last decade with the idea to evaluate the whole chain of processes inside of deep learning inference serving system [22,23,29]. These tools focus on defining inference elements and measure hardware, software, and pipelines to evaluate such metrics as:

- latency and throughput: measures resource (GPU/CPU/TPU) utilization during serving and inference, as well as its impact on online and offline services;
- cost: includes serving environment financial costs like cloud cost per provider, energy measures, and CO2 emission;
- memory and computation: capacity of the serving tools to manage resources, its impact on model performance;
- tail latency: serving platforms effective mitigation of tail latency and its impact on model performance;
- resource usage: evaluates how effective is served model design;
- advanced features: allows identifying resource utilization in case if serving tool provides effective functions such as dynamic batching.

Nvidia Triton Inference Server, TensorFlow Serving, FastAPI ONNX Server, and FastAPI TorchScript Server software benchmarking was performed by Zhang H. and other researchers [23]. They used InferBench tool to analyze serving infrastructures and compared serving tools for the same image classification task using GPU acceleration [23]. As we see from the results in the image below (Figure 6), NVIDIA Triton Inference Server performs the best, and this is no surprise as it contains a lot of GPU optimization techniques.

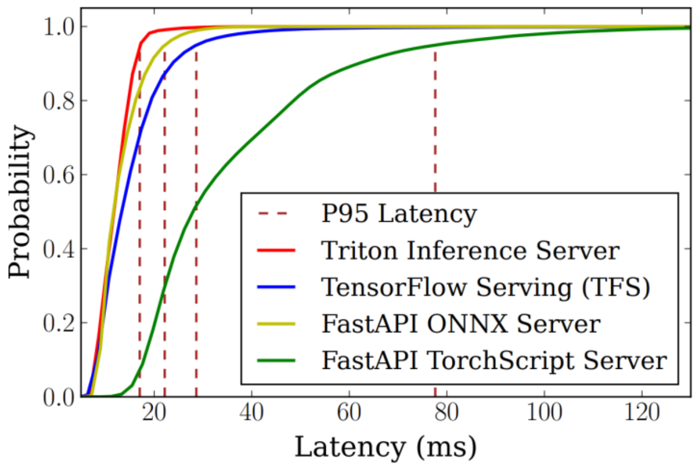


**Figure 6.** The tail latency of four serving systems for image classification task performed on GPU device

Subsequently, the resource use by these four software was investigated as a "resource utilization pattern often leads to better resource allocations" [23], the results are shown in Figure 7.

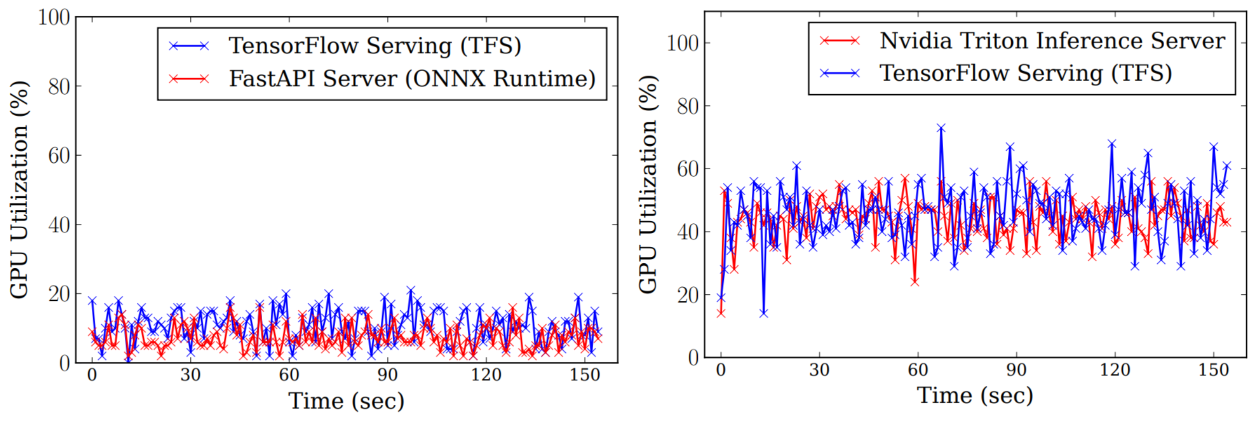


**Figure 7.** GPU utilization with four serving software under different workloads [23]

The final Zhang's test was performed only with two software – TensorFlow Serving and NVIDIA Triton Inference Server. The researchers identified that Triton has a longer starting time in the case of the "cold start" test scenario, rather than TensorFlow Serving (see Figure 8). "Even for a small image classification model, it needs more than 10 seconds to prepare" [23]. According to the experiment, a long start time may challenge machine resource provisioning.

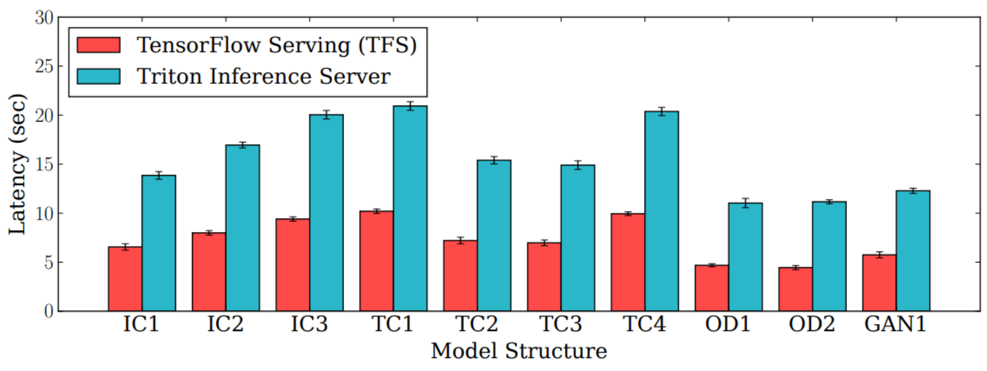


**Figure 8.** Cold start latency of different models with two serving software

Some researchers concentrated on benchmarking models optimization techniques or efficient training, others were benchmarking inference hardware [29,30]. However, serving platform overviews are mostly performed by technology companies or machine learning specialists and they are shared as articles in blogs or company pages.

## 1.7. Model Compression for Inference Optimization

Deep neural networks have achieved high success in many industries, although their computational and memory costs cause problems for resource-constrained environments [31,32,33]. Low memory restrictions, as well as strict latency requirements, obstruct model deployment, and there is a common thought that performing model optimization to meet existing device setup may require sacrificing accuracy. However, today's technologies are highly advanced, and there are lots of proposals on how to accelerate inference, including offloading a part of computations to a cloud server, designing efficient architectures, compressing pre-trained models, or designing hardware with prebuilt functionalities allowing to reduce memory latency.

Researchers point out different approaches to obtain small networks. These methods include shrinking, compressing, or factorizing pre-trained models. Compressing a pre-trained model is used with the purpose "to reduce its storage and memory footprint as well as computational requirements" [31]. This research topic has been promising as it allows one to decrease the model size without compromising accuracy.

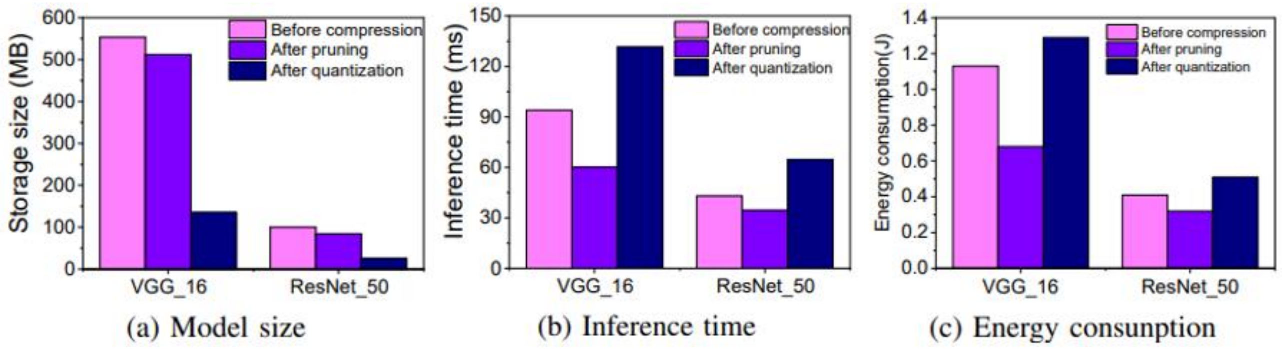


**Figure 9.** Compressed model metrics (pruning and quantization) [35]

Many research concentrates on designing efficient neural network model architecture (in terms of depth-wise convolution applied for EfficientNet models, or low-rank factorization, residual model types, etc). New methods such as Neural Architecture Search or Automated ML enrich classic architectures. The aim of these methods is "to find in an automated way the right NN architecture, under given constraints of model size, depth, and/or width" [33].

One more approach emphasized in many experiments is pruning. "In pruning, neurons with small saliency (sensitivity) are removed, resulting in a sparse computational graph" [33], meaning, redundant and non-informative weights of pre-trained deep neural network model are pruned [32]. In early work, network pruning proved to be a valid way to reduce network complexity and over-fitting [34].

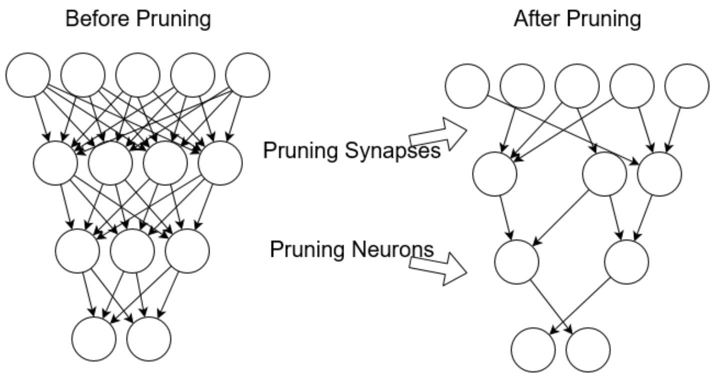


**Figure 10**. Neural network pruning [36]

Model quantization can be applied for all model deployments, especially it might be critical for mobile model deployment, because of its size which may exceed Android or iOS applications limits [37]. According to the PyTorch documentation, model quantization before deployment can reduce the inference speed by up to 2-4 times [37]. That is, a full-precision neural network is converted into a low-bit width integer version by using quantization. However, some of these compression methods have not been proven for various types of architecture [38].

The knowledge distillation method is also widely used to transfer knowledge from one model to another. "The idea has been recently adopted as knowledge distillation (KD) to compress deep and wide networks into shallower ones, where the compressed model mimicked the function learned by the complex model" [32]. This approach allows to forward knowledge, gained by the large model to teach small one the class distributions output using softmax.

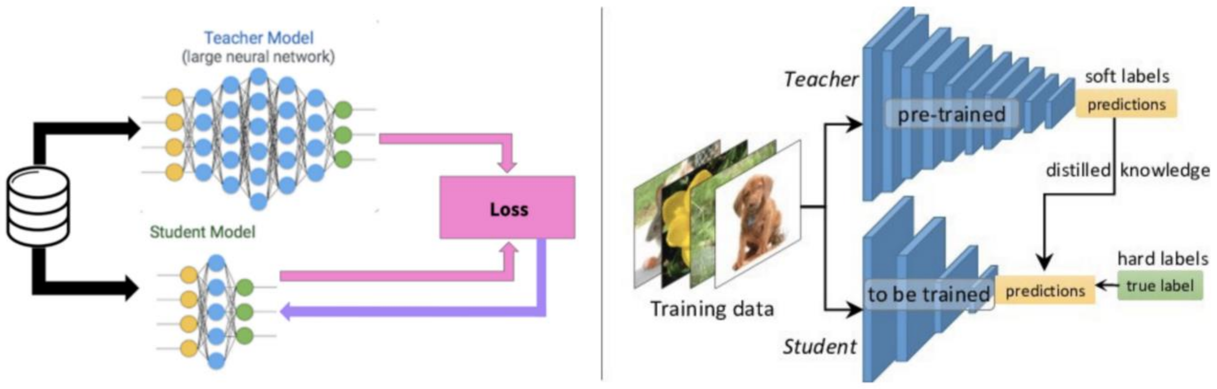


**Figure 11.** Knowledge distillation [36]

Another compression method is low-rank factorization using matrix decomposition for the estimation of informative parameters. This method is highly recommended use for in the training phase as it might reduce training time. According to research, a low-rank approximation can help to achieve 2 times more speed for a single convolutional layer with only 1% smaller classification accuracy [32].

**Table 2**. Model compression and acceleration approaches summary

| **Category Name** | **Description** | **Applications** | **Other Details** |
|---|---|---|---|
| Parameters pruning and quantization | Reducing redundant parameters which are not sensitive to the performance | Convolutional layer and fully connected layer | Robust to various settings, can achieve good performance, can support both train from scratch and pre-trained model |
| Low-rank factorization | Using matrix/tensor decomposition to estimate the informative parameters | Convolutional layer and fully connected layer | Standardized pipeline, easily to be implemented, can support both train from scratch and pre-trained model |
| Knowledge distillation | Training a compact neural network with distilled knowledge of a large model | Convolutional layer and fully connected layer | Model performances are sensitive to applications and network structure only support train from scratch |

## 1.8. The Need for Research

In the first chapter, we introduced the typical workflow of building deep learning services, and their application in practice, and reviewed a variety of model serving software. The diversity of inference applications, models, machine learning frameworks, data sets, libraries and packages, platforms, and hardware is numerous, which complicates the observability of the inference system and reproducible benchmarking [22]. Figure 5, depicts a narrow range of available combinations, however, the truth is – building an inference system for a production environment requires knowledge in various fields to assess each component‘s impact on inference. Also, model architectures are constantly optimized and tools – upgraded, which challenges each machine learning tool to support integration with each other.

Due to hundreds of available ML tools and even more combinations of them – benchmarking of deep learning model serving software has not been thoroughly investigated by the current studies [23,17]. Subsequently, there is a growing need for computer vision solutions responding to a nearly insensible amount of time – humanoid robotics, smart devices, automatization solutions, and medical devices needs to consume images fetched from the environment and be able to interact with them in a way human does, or perform high scale analysis and provide feedback.

Serving software is just one of the components used to utilize deep learning models for image analysis at scale. However, choosing the right software may lead to building a core for effective models inference system infrastructure.

**Research goal.** Recommendations for efficient image classification inference at scale.

**Research tasks.** The following objectives have been established to achieve the research goal:

1. Research literature on the application of deep learning, its relevance, and its impact in multiple industries.
2. Review the model serving process, platforms, and related research.
3. Choose and describe novel architectures used for image classification.
4. Perform benchmarking by serving pre-trained models to the production environment via different platforms and compare the results.
5. Compact models through quantization and perform additional benchmarking.
6. Provide general recommendations or guidelines for selecting the right technology for efficient image classification deep learning model serving.

# 2. Methods and Concepts Used in Research

In this paper, the focus is to provide recommendations on efficient computer vision model serving production environments. The following part of the thesis provides a review of methods and architectures that help to fulfill the project and describes concepts and technical guidelines for the research project. This chapter will cover computer vision methodologies, models, and their optimization options, as well as tools for machine learning model serving.

## 2.1. Computer Vision

A field of artificial intelligence that trains computers to interpret and understand the visual world is called computer vision [39]. It uses images fetched from cameras or videos and applies deep learning models for acquiring, processing, and analyzing digital images so that machines could identify and classify objects - and then respond accordingly. Computer vision requires a lot of data to perform with high precision. One of the essential technologies used for image recognition - is a convolutional neural network (CNN) which helps a deep learning model to break the image down into pixels relating it to given labels and helps the machine identify what it is seeing. When it comes to computer vision, the key problems to solve are image segmentation and classification.

## 2.2. Convolutional Neural Network

Convolutional neural networks are a type of artificial neural network where the connection between neurons is based on the biological human visual cortex system (figure 3). The primary visual cortex is responsible for taking visual input from the retina and detecting edges. The secondary visual cortex receives the edge components from the primary cortex and extracts regular properties such as spatial frequency and color. The visual area (marked as V4) handles much more complicated object attributes than the others. Visual features flow processed all together into the final logical unit, inferior temporal gyrus (IT), which is passed for object recognition [2].

We may see the relation between V1 and V4 - it is a special type of convolutional neural network with a relation between non-adjacent layers: Residual Net which supports some input of one layer to be passed to the component two layers later. Residual learning was presented to ease CNN training that is much deeper than those used previously. Layers are reformulated as learning residual functions concerning the layer inputs, instead of learning unreferenced functions. Such networks are easier to optimize and achieve better accuracy (He et al., 2016).

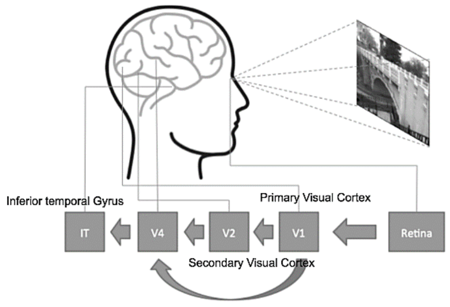


**Figure 12.** Human visual cortex system

Convolution in math is called an operation between two matrices. "The convolutional layer has a fixed small matrix defined, also called kernel or filter. As the kernel slides, or convolving across the matrix representation of the input image, it computes the element-wise multiplication of the values in the kernel matrix and the original image values" [3] (Figure 13). Kernels can be used for processing images for such purposes as edge detection, image sharpening, or many others, quite quickly and efficiently.

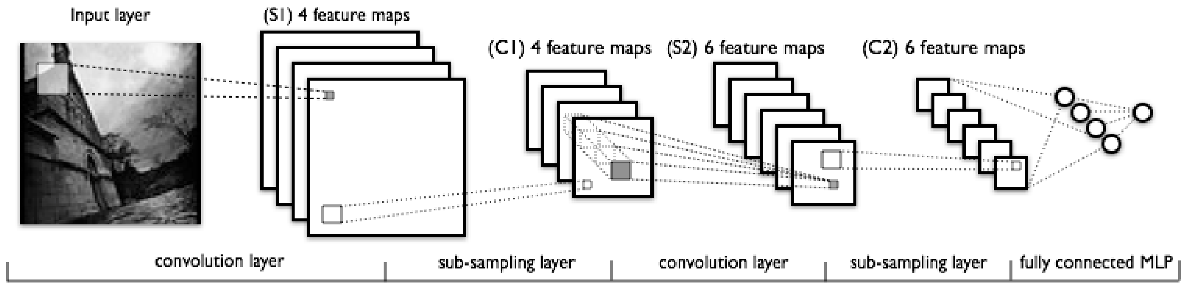


**Figure 13.** The LeNet architecture

## 2.3. Image Classification

Image classification is one of the fundamental tasks in computer vision responsible for "categorizing and assigning labels to groups of pixels or vectors within an image dependent on particular rules" [40]. It can accurately predict that a given picture belongs to a certain class.

Image classification works in a way where the computer analyzes the digital image at the pixel level considering it as an array of matrices and automatically groups pixels into specified categories, better known as classes. The characteristics extraction process is the most important step in image classification, as the machine cannot interpret images the way a human does. Due to this reason, image preprocessing step contains enhancement of some image features which may help identify unique properties applied for a specific class. During preprocessing stage input data is converted in different ways to extract more information about the image. For example, gray scaling allows the computer to assign pixel value based on how dark it is or using Gaussian smoothing to blur an image reducing the noise, lastly, histogram equalization is used to increase image contrast and analyze intensity histogram to find the patterns (Figure 14).

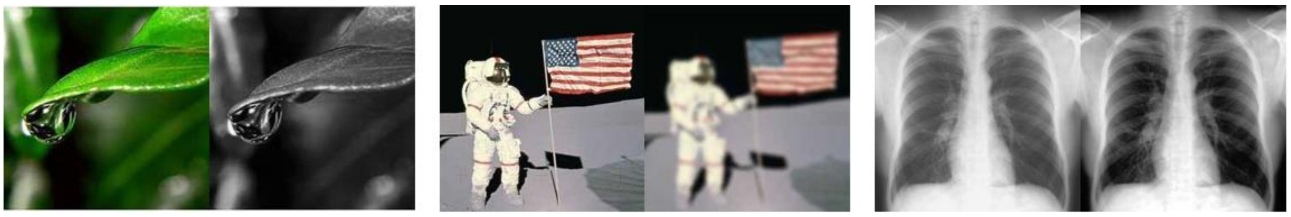


**Figure 14.** Features extraction examples in the image classification process

Supervised classification methods use predefined data samples to train the classifier. Manual data labeling for good and bad sample interpretation is known as image annotation and it requires a human being to choose the category where the sample image should be allocated. In the case of supervised classification, the accuracy of the model depends a lot on the amount of training data [40].

Unsupervised classification is fully automated and does not require training data to be passed to teach the model. It means the algorithm is written in a way that it will analyze and cluster unlabeled datasets by discovering hidden patterns by itself, without human interaction. A simple example from everyday life can be related to our mobile devices where Android OS applies face recognition to classify images and afterward builds catalogs for reviewing photos per each photographed person. The algorithm can learn from data stored in a gallery, recognize facial differences among all people, and cluster photos based on it.

According to Boesch, the co-founder of Visio.ai software for computer vision, pattern recognition, and image clustering are the most common image classification methods, and two popular algorithms used for it are K-mean and ISODATA. K-means is an unsupervised classification algorithm that aims to partition objects into k clusters based on their features [41]. K-means clustering tries to minimize squared Euclidean distances between observations and the centroid of a cluster to which it belongs. This is one of the simplest unsupervised algorithms. On the other hand, ISODATA also uses Euclidean distance as the similarity measure, but instead of having a predefined number of clusters - its algorithm includes iterative methods allowing a different number of clusters. Data scientist Taru Jain also confirms the popularity of K-means classification algorithm, however, there are lots of other supervised machine learning algorithms used for classification and they have their pros and cons depending on the circumstances. Jain made an experiment to compare five algorithms such as Support Vector Machines, Decision Trees, K-means, Artificial Neural Networks, and Convolutional Neural Networks, as a result - the model using CNN gave better results than the rest of the models. The challenging part of using convolutional neural networks in practice is usually designing the model architecture so as to achieve the best results [40,41].

### 2.3.1. EfficientNet

EfficientNet is a family of models introduced by Google AI, and the idea of it is to scale up convolutional neural networks in a way the model would be more efficient, without losing the state of art results. Lately, this model became highly popular and is used as a backbone in a lot of machine learning projects.

EfficientNet scaling is made through multiple network dimensions, as well as using higher resolution input for model training [42]. However, these are basic model scaling methods and Google decided to make it one step further by proposing „ a novel model scaling method that uses a simple yet highly effective *compound coefficient* to scale up CNNs in a more structured manner“ [42]. Their research was based on scaling each dimension by setting fixed scaling coefficients, and introducing extended usage of AutoML. Below we can see an image provided by Google comparing multiple scaling methods and it shows conventional scaling methods (b-d) covering scaling per one dimension, as well as compound scaling (e) where all dimensions are scaled uniformly. The latter is a visualization of the EfficientNet scaling methodology.

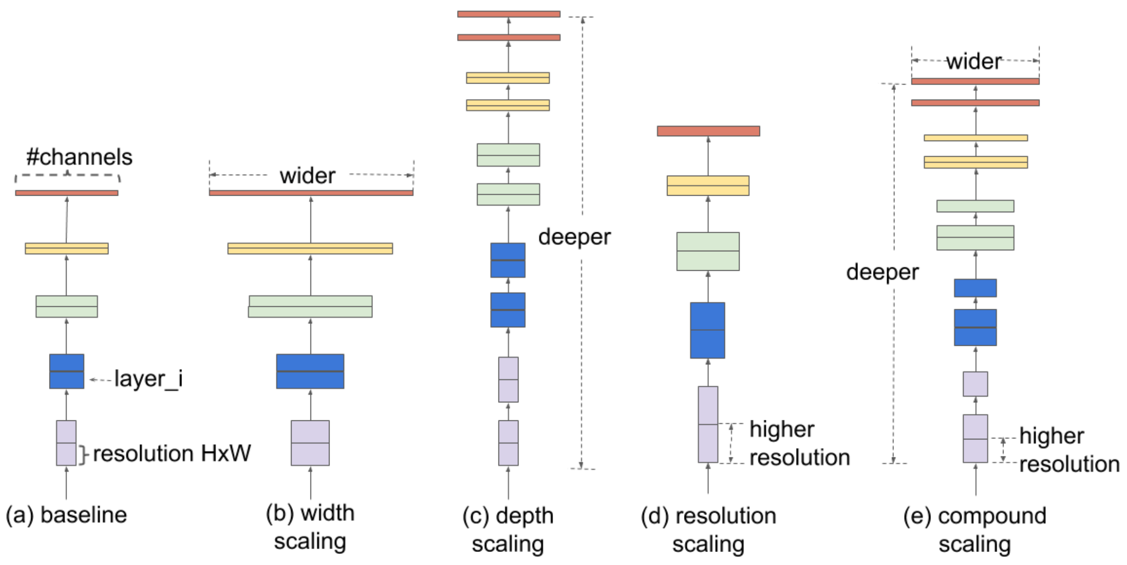


**Figure 15.** Scaling of the efficientNet architecture model [42]

EfficientNet model architecture has a common structure with other networks and it consists of the input layer, rescaling, normalization, zero padding, Conv2D, batch normalization, and activation. Each of these components has its sub-components which quantity increases when we move from EfficientNetB0 to EfficientNetB7 models. The scaling level, as well as the number of parameters, increases with each version in this model family and we can see how each of them impacts the accuracy (see Figure 16).

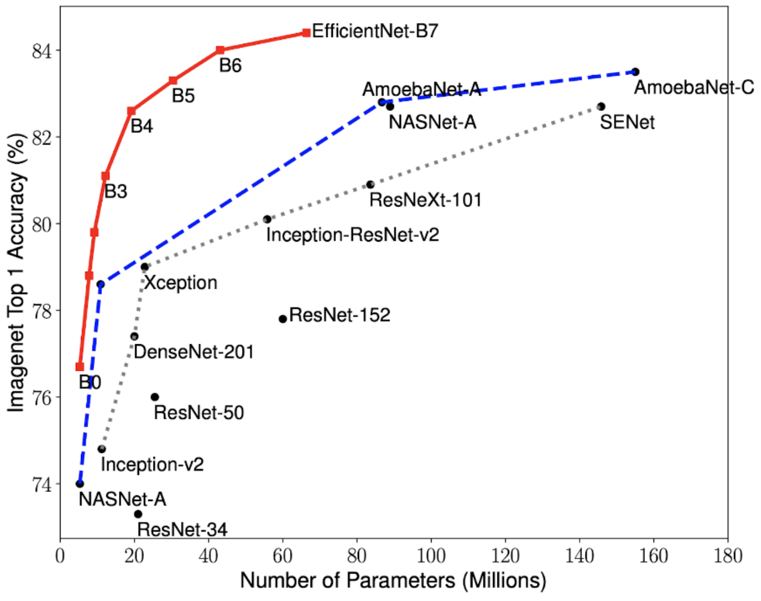


**Figure 16.** Model size versus accuracy comparison [42]

Theoretics such as Raghu, Lu, and Komodakis already proved the impact of dimension scaling efficiency for the model, however, Google AI team was the first one who quantified "the relationship among all three dimensions of network width, depth, and resolution" [42]. Scaling depth is one of the most common methods applied for convolutional neural networks as the deeper network is – the more complex features model can capture [1]. Scaling the network width is used more often for small models [43]. Higher resolutions are commonly used in convolutional neural networks as it improves accuracy due to better recognizable features. On the other hand, if we would describe convolutional layer ($i$) as a function $Y_i = F_i(X_i)$, here $Y_i$ would correspond to output tensor, $F_i$ – operator, and $X_i$ - input tensor having a shape ($H_i$ – height, $W_i$ - width, $C_i$ – channel dimensions). Based on that we can define convolutional network (N) consisting of multiple layers: $N = F_k \odot F_2 \odot F_1(X_1) = \odot_{j=1...k} F_j (X_1)$. „In practice, ConvNet layers are often partitioned into multiple stages and all layers in each stage share the same architecture: for example, ResNet (He et al., 2016) has five stages, and all layers in each stage have the same convolutional type except the first layer performs down-sampling“ [42]. In addition, Tan and others define a convolutional network as such formula:

$$N = \bigodot_{i=1...s} F_i^{L_i} (X_{\langle H_i, W_i, C_i \rangle}).$$

The formula indicates that $F_i$ is repeated $L_i$ times in stage $i$, and values inside of parenthesis define the shape of the input tensor of layer $i$. The next formula illustrates how "spatial dimension is gradually shrunk but the channel dimension is expanded over layers, for example, from initial input shape $\langle 224, 224, 3 \rangle$ to final output shape $\langle 7, 7, 512 \rangle$" [42]:

$$\max_{d,w,r} \quad Accuracy(N(d, w, r))$$
$$s.t. \quad N(d, w, r) = \bigodot_{i=1...s} \hat{F}_i^{d \cdot \hat{L}_i} (X_{\langle r \cdot \hat{H}_i, r \cdot \hat{W}_i, w \cdot \hat{C}_i \rangle})$$

Such an approach helps to expand the network length ($L_i$), width ($C_i$), and/or resolution ($H_i$, $W_i$) without changing the model architecture predefined in the baseline model. Model scaling's main goal is to solve the optimization problem by increasing model accuracy. The best way to visualize the effectiveness of Google AI team‘s compound scaling method would be a chart, showing us model accuracy when scaling each dimension separately versus all together (see Figure 17).

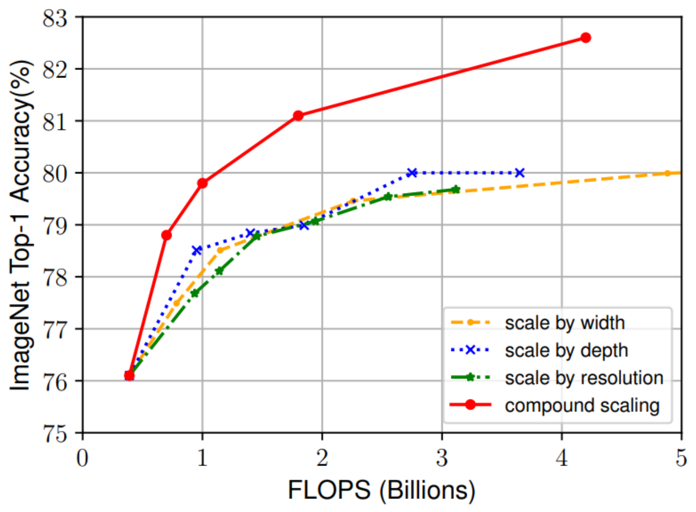
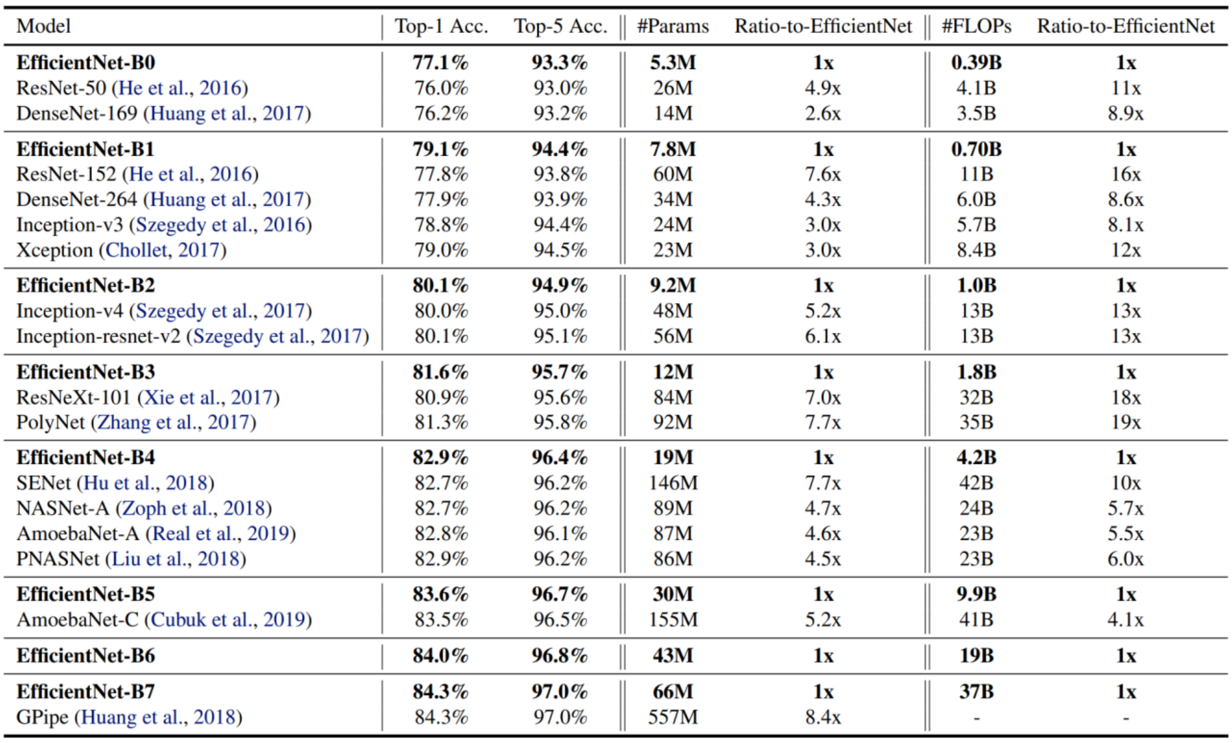


**Figure 17.** Scaling Up EfficientNet-B0 with Different Methods

The below image shows a comparison of EfficientNet models efficiency starting from the EfficientNet-B0 baseline model till EfficientNet-B7, using different compound coefficients.



| Model | Top-1 Acc. | Top-5 Acc. | #Params | Ratio-to-EfficientNet | #FLOPs | Ratio-to-EfficientNet |
|---|---|---|---|---|---|---|
| **EfficientNet-B0** | **77.1%** | **93.3%** | **5.3M** | **1x** | **0.39B** | **1x** |
| ResNet-50 (He et al., 2016) | 76.0% | 93.0% | 26M | 4.9x | 4.1B | 11x |
| DenseNet-169 (Huang et al., 2017) | 76.2% | 93.2% | 14M | 2.6x | 3.5B | 8.9x |
| **EfficientNet-B1** | **79.1%** | **94.4%** | **7.8M** | **1x** | **0.70B** | **1x** |
| ResNet-152 (He et al., 2016) | 77.8% | 93.8% | 60M | 7.6x | 11B | 16x |
| DenseNet-264 (Huang et al., 2017) | 77.9% | 93.9% | 34M | 4.3x | 6.0B | 8.6x |
| Inception-v3 (Szegedy et al., 2016) | 78.8% | 94.4% | 24M | 3.0x | 5.7B | 8.1x |
| Xception (Chollet, 2017) | 79.0% | 94.5% | 23M | 3.0x | 8.4B | 12x |
| **EfficientNet-B2** | **80.1%** | **94.9%** | **9.2M** | **1x** | **1.0B** | **1x** |
| Inception-v4 (Szegedy et al., 2017) | 80.0% | 95.0% | 48M | 5.2x | 13B | 13x |
| Inception-resnet-v2 (Szegedy et al., 2017) | 80.1% | 95.1% | 56M | 6.1x | 13B | 13x |
| **EfficientNet-B3** | **81.6%** | **95.7%** | **12M** | **1x** | **1.8B** | **1x** |
| ResNeXt-101 (Xie et al., 2017) | 80.9% | 95.6% | 84M | 7.0x | 32B | 18x |
| PolyNet (Zhang et al., 2017) | 81.3% | 95.8% | 92M | 7.7x | 35B | 19x |
| **EfficientNet-B4** | **82.9%** | **96.4%** | **19M** | **1x** | **4.2B** | **1x** |
| SENet (Hu et al., 2018) | 82.7% | 96.2% | 146M | 7.7x | 42B | 10x |
| NASNet-A (Zoph et al., 2018) | 82.7% | 96.2% | 89M | 4.7x | 24B | 5.7x |
| AmoebaNet-A (Real et al., 2019) | 82.8% | 96.1% | 87M | 4.6x | 23B | 5.5x |
| PNASNet (Liu et al., 2018) | 82.9% | 96.2% | 86M | 4.5x | 23B | 6.0x |
| **EfficientNet-B5** | **83.6%** | **96.7%** | **30M** | **1x** | **9.9B** | **1x** |
| AmoebaNet-C (Cubuk et al., 2019) | 83.5% | 96.5% | 155M | 5.2x | 41B | 4.1x |
| **EfficientNet-B6** | **84.0%** | **96.8%** | **43M** | **1x** | **19B** | **1x** |
| **EfficientNet-B7** | **84.3%** | **97.0%** | **66M** | **1x** | **37B** | **1x** |
| GPipe (Huang et al., 2018) | 84.3% | 97.0% | 557M | 8.4x | - | - |

**Figure 18.** EfficientNet Performance Results on ImageNet [42].

### 2.3.2. MobileNet

MobileNet is built from lightweight deep neural networks and is most commonly used in mobile or embedded vision applications. Small, efficient models were analyzed by much research before [45,46], according to Google software engineer Andrew G. Howard, "MobileNets primarily focus on optimizing for latency but also yield small networks" [44], however, most previous research papers cover only minimization of the model itself. The heritage of MobileNet models comes from Inception models being characterized by computation reduction in the first layers. Later, Xception networks were introduced with the scaled-up depthwise separable filters. Finally, a similar fate was foreseen to such models as it happened to EfficientNet – the Google AI team started working on scaling small models to optimize latency and the MobileNet family models were born.

MobileNet models can be applied to various computer vision tasks: object detection, fine-grain classification, face recognition, etc. This model family layers are built on depthwise separable filters formed of factorized convolutions where standard convolution is factorized into depthwise and pointwise convolutions (see Figure 19). Deptwise convolution is spatial channel-wise convolution DK x DK (if there would be 5 channels, then we would have 5DK x 5DK). Its main difference from standard convolution is a separation of filtering and combining tasks into two separate layers. Point-wise convolution applies 1x1 convolution to combine depthwise convolution output dimension.

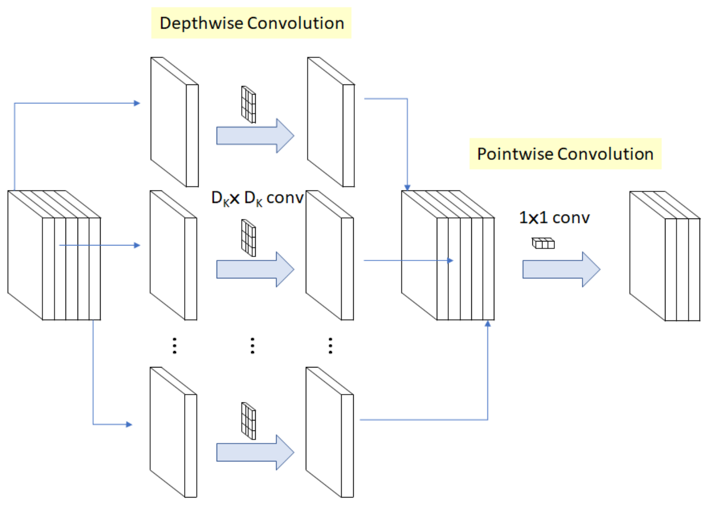


**Figure 19.** Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

"The standard convolutional layer is parameterized by convolution kernel K of size $D_K \times D_K \times M \times N$ where $D_K$ is the spatial dimension of the kernel assumed to be square and M is a number of input channels and N is the number of output channels as defined previously" [43]. If we were to look at standard convolution, then the output feature map padding is calculated as:

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m}$$

Afterward, the standard convolution computation cost is:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

where $D_F$ stands for feature map size.

On the other hand, as pointwise and depthwise convolutions make depthwise separable convolutions, it is worth mentioning the use of batchnorm and ReLU nonlinearities used in both layers. Deptwise convolution having a single filter per input channel can be described as:

$$\hat{G}_{k,l,m} = \sum_{i,j} \hat{K}_{i,j,m} \cdot F_{k+i-1,l+j-1,m}$$

where $\hat{K}$ stands for depthwise convolution kernel, DK × DK × M – the size of the kernel, $\hat{G}$ – filtered output feature map. The cost of the depth-wise convolution computation is:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$$

As the combination of depthwise and pointwise convolution is called depthwise separable convolution, its cost would be calculated as:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

"MobileNet uses 3 × 3 depthwise separable convolutions which uses between 8 to 9 times less computation than standard convolutions at only a small reduction in accuracy" [43]. In summarizing, MobileNet models architecture is built from depthwise separable convolutions, except for the first layer which is full convolution (see Figure 20).

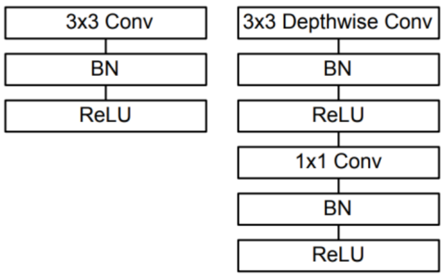


**Figure 20.** Standard convolution layer versus depthwise convolution with Deptwise and Pointwise layers

Despite MobileNet already being a small-size model, there was more place for optimization, and width multipliers were introduced to these models to make them even smaller and quicker. Howards and others emphasize - „the role of the width multiplier α is to thin a network uniformly at each layer“ [43]. Based on this, computational cost increases by multiplying input (M) and output (N) channels having width multiplier (α). The second hyperparameter used in MobileNet is the resolution multiplier (ρ) whose purpose is to reduce the computational cost of the network. It is applied to the input layer and follows through every further step. Practically, the application of resolution multiplier means setting up resolution for input to 224, 192, 160, or 128, and values stand for $\rho \in (0, 1]$. When $\rho < 1$, the calculations of the MobileNet model are reduced:

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$$

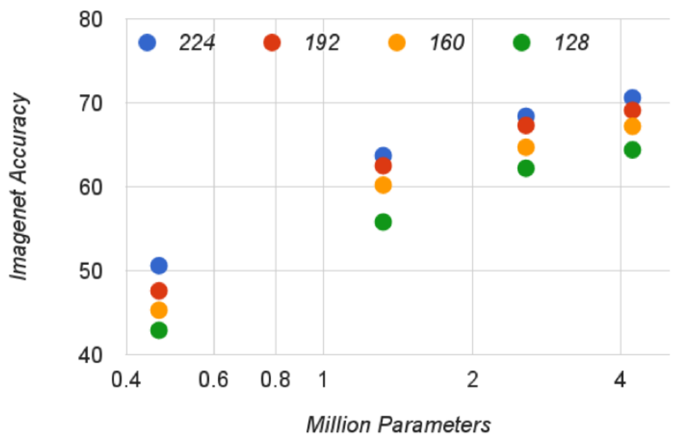


**Figure 21.** Figure shows ImageNet accuracy and parameters number trade off for the 16 models made with different width multiplier values $\alpha \in \{1, 0.75, 0.5, 0.25\}$ as well as resolutions {224, 192, 160, 128}

MobileNet has two more versions released, and both are highly customized. MobileNetV2 incorporates the ideas of the previous version, on the other hand, it introduces a couple of new features such as linear bottlenecks between layers and shortcut connections between these bottlenecks. The latter encodes intermediate model inputs and outputs, while the layers inside of the bottleneck make the transformation from lower-level concepts, such as pixels, to higher-level – image categories. Finally, shortcut provides residual connection enabling quicker training and a higher accuracy rate [46].

The third version of MobileNet was enriched with network architecture search providing hardware awareness to the model, as well as layers upgrade with modified swish nonlinearities. Compared to the V2 version model, the 1x1 convolution layer was moved past the final average pooling to reduce latency and preserve all the features. „The outcome of this design choice is that the computation of the features becomes nearly free in terms of computation and latency“ [44]. In addition, the number of filters was reduced and hard swish nonlinearity was introduced in this layer. According to research, it reduced latency by 2 milliseconds.

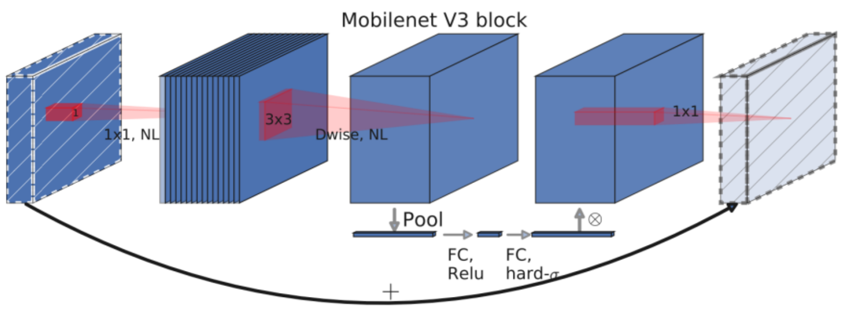


**Figure 22.** MobileNet V3 architecture

Lastly, MobileNet is defined as MobileNetV3-Large and MobileNetV3-Small models, which target high and low resource use cases. This is the result of the network architecture search application helping to optimize performance based on hardware awareness. Figure 23 visualizes performance among multiple MobileNet models with different parameters, and as we see – MobileNetV3-Large accuracy is the highest one. This model will be used in further research for the model deployment platforms' benchmark.

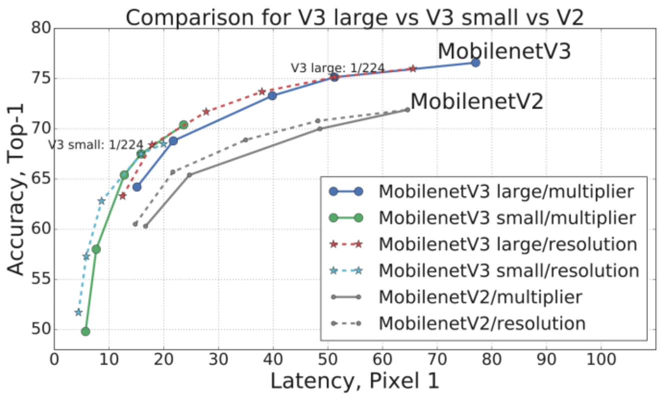


**Figure 23.** Performance of MobileNet models with different multipliers and resolutions

## 2.4. Quantization

In deep neural networks, weights are stored as 32-bit floating-point numbers. Quantization allows for reducing weights to 16, 8, 4, or even 1-bit, which allows to build the small-size model.

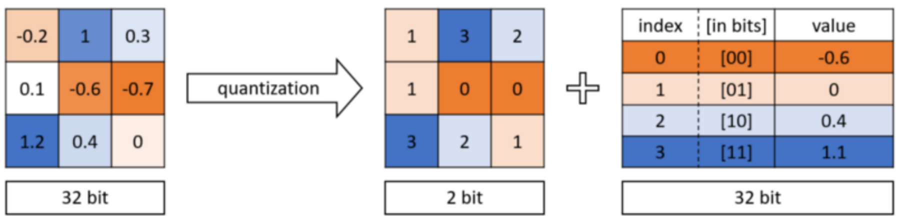


**Figure 24.** Binary quantization

The huge benefit of quantization is that it can be applied both during model training and when the model is already pre-trained. Quantization-aware training is treated to be as most efficient, however, this paper covers benchmarks using already pre-trained models. Torchvision library provides quantized models version, however, neither of them were the ones, prepared for other benchmark scenarios (using EfficientNet-B7 and MobileNetV3-large models). Due to this reason, Torchvision proposes a solution for pre-trained model quantization, which was used in this research to quantize the MobileNetV3-large model.

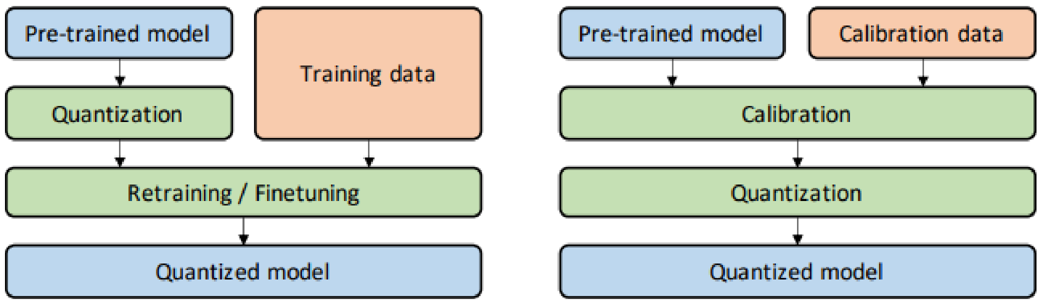


**Figure 25.** Comparison of quantization-aware training and post-training quantization

## 2.5. Serving Models for Experiment

As noted above, one of our goals was to experiment with deploying computer vision models using three different serving platforms and benchmark inference. The experiment uses TensorFlow Serving, Torchserve, and Nvidia Triton serving platforms, as well as pre-trained models for EfficientNet-B7 and MobileNetV3-Large architectures. As each deployment platform has its requirements for model preparation – PyTorch and TensorFlow frameworks models were used for serving (see Table 3).

**Table 3.** Pre-trained deep learning models chosen for the experiment

| | **EfficientNet-B7** | **EfficientNet-B7** | **MobileNetV3-Large** | **MobileNetV3-Large** | **Quantized MobileNetV3-Large** |
|---|---|---|---|---|---|
| **Task** | Classification | Classification | Classification | Classification | Classification |
| **Training Data** | Imagenet | Imagenet | Imagenet | Imagenet | Imagenet |
| **Model Framework** | TensorFlow Serving | PyTorch | TensorFlow Serving | PyTorch | PyTorch |
| **Parameters** | 66M | 66M | 5.4M | 5.4M | 5.4M |
| **Top 1 ImageNet Accuracy** | 84.4 % | 84.4 % | 75.2 % | 75.2 % | 74.1 % |
| **Resolution, px** | optimal 600 x 600, any | optimal 600 x 600, any | 224x224 | 224x224 | 224x224 |
| **Size** | 235.01 MB | 234 MB | 19.63 MB | 22.1 MB | 7.82 MB |
| **Depth** | 813 layers | 813 layers | 28 layers | 28 layers | 28 layers |
| **Source** | https://tfhub.dev/TensorFlow/efficientnet/b7/classification/1 | https://github.com/d-li14/mobilenetv3.pytorch | https://tfhub.dev/google/imagenet/mobilenet_v3_large_100_224/classification/5 | https://download.pytorch.org/models/mobilenet_v3_large-8738ca79.pth | Quantized model using guidelines: https://pytorch.org/tutorials/recipes/quantization.html |

### 2.5.1. Docker

Docker is an open-source containerization platform and virtualization technology that eases development and allows us to deploy and manage applications by using containers. Container refers to a lightweight, stand-alone, standardized executable component that combines the source code of the application with the operating system (OS) libraries and all the dependencies necessary to run the containerized application in any environment (see Figure 26).

Most ML model deployment platforms provide the possibility to set up the deployment and inference environment inside of docker container, and for that, a Docker image is provided per each serving platform. A Docker image is like a template that contains all the instructions for creating a container that can run on the Docker platform. It provides all packages and environment configurations needed for the application to run successfully on any machine.

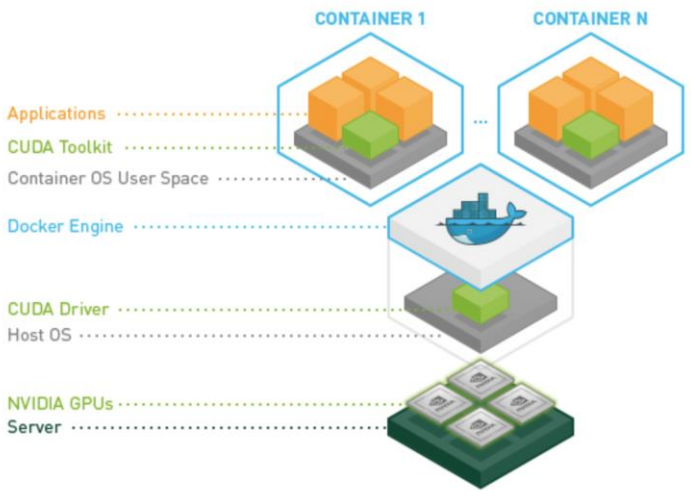


**Figure 26.** Docker architecture, enhanced with Nvidia GPU components

Setting up Docker on Ubuntu OS requires a small amount of effort. Docker container setup guidelines can be found on the official website, https://docs.docker.com/engine/install/ubuntu/. For this experiment, we ensured all required packages for docker installation are up to date, and then run the Docker installation command:

```
sudo apt-get update \
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

The ability to deploy models using different platforms to separate containers was the biggest advantage of using docker, as other virtualization tools would require additional effort to set up guest OS and it would be one more step where environmental discrepancies may appear due to human error. In the scope of this experiment – each model was deployed using a different kind of serving platform on separate containers.

### 2.5.2. NVIDIA Triton Inference Server

Nvidia Triton inference server is an open-source ML model serving software that helps to standardize model deployment in production. It provides both, GPU and CPU-based infrastructure, and supports multiple frameworks such as Nvidia TensorRT, PyTorch, ONNX, TensorFlow, and more (Appendix 1). The software also offers easily configurable batching and streaming inputs maximizing throughput, as well as optimal model configuration. Triton Server can be easily integrated into most desired products in today's IT world, such as Amazon/Azure/Google Cloud solutions, Kubernetes engines and machine learning tools.

The easiest way to install Triton is to use the pre-built Docker image provided in Nvidia`s official catalog. However, Triton Server installation has such pre-requisites set [47-49]:

1. Docker should be already installed.
2. NVIDIA drivers must be installed (Appendix 2).
3. The Nvidia Container Toolkit (Appendix 3) must be set up for GPU usage during inference.

Triton installation itself is easy because of the official Docker image stored online:

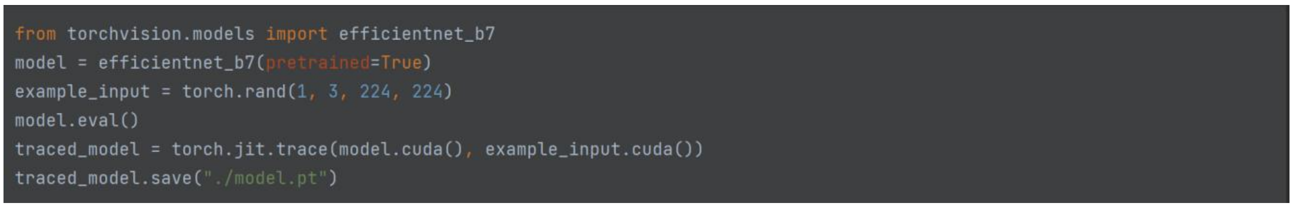
```
docker pull nvcr.io/nvidia/tritonserver:<xx.yy>-py3
```

Here we see the xx.yy variable, which defines the Triton version to serve. In this experiment, the 21.02 version was used as it was the newest Triton package at the time of the experiment.

### 2.5.2.1. Model Deployment

The first step in working with Triton is to prepare a model repository, a file-system-based repository of the models that Triton makes available for inferencing. It consists of a model file and a model configuration file. It is highly important to set the correct model repository layout and naming as it differs based on a model framework. The experiment was based on deploying the PyTorch model, however, Triton supports only models, converted to TorchScript model format.

The TorchScript model file can be fetched to be compiled as executable using *trace()* function (see Figure 27). At this step it is important to define that model and input should support CUDA tensor types, otherwise, deployment using GPU will fail due to model incompatibility with deployment environment. By default, model tensors supports CPU, unless it was set differently during model training.



```python
from torchvision.models import efficientnet_b7
model = efficientnet_b7(pretrained=True)
example_input = torch.rand(1, 3, 224, 224)
model.eval()
traced_model = torch.jit.trace(model.cuda(), example_input.cuda())
traced_model.save("./model.pt")
```

**Figure 27.** Code defining how to get TorchScript model file

A traced model can be used for Triton deployment, but the first model repository should be set. Here is the minimal repository structure, which was also used for the experiment:

```
  <model-repository-path>/
    <model-name>/
      config.pbtxt
      1/
        model.pt
```

One of the main Triton requirements for a model repository is to have a model name set to "model.pt". Also, the *config.pbtxt* file is mandatory for TorchScript models as Triton cannot identify the default model configuration from these models. On the other hand, if the Tensorflow model is used – *config.pbtxt* file is not needed because Triton is capable to set the default configuration for these models.

When the model repository is ready – model deployment can be performed. Triton is deploying all models which are stored to the model repository at once which enables access to models through the same docker container. As the experiment requirement is to benchmark each model one by one – the model repository consisted of only one model when the deployment was performed. The following command is used to run Triton with the uploaded model:

```
nvidia-docker run --gpus all --rm -p8000:8000 -p8001:8001 -p8002:8002 \
-v/home/renata/ktu/nvidia/model_repository:/models nvcr.io/nvidia/tritonserver:21.02-py3 \
tritonserver --model-repository=/models --strict-model-config=true
```

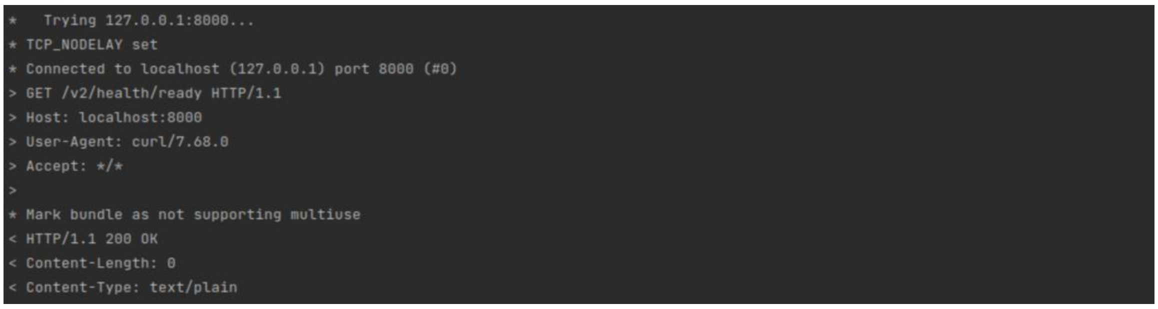
The `--gpus` flag defines GPUs which have to be used for this model serving – we had only one GPU in this experiment, however, in case more GPUs are supported by the machine, it is allowed to define what GPUs to use (e.g. `–gpus='device=1,2'`). One more thing important to mention is that `--strict-model-config=true` should be set as we have *config.pbtxt* file defined in the model repository and TorchScript model deployment will fail if this flag will be set to *false* (as mentioned before, Triton is not capable to define TorchScript model configuration by itself).

Successful model deployment can be identified by console output displayed in Appendix 4. It displays the list of served models, their version, and status, as well as serving variables and three IPs with ports to access Triton services. In case of model failure – Triton rolls back all changes, the docker container is not created and the error log provides a detailed failure reason description.

Triton allows users to verify that it is running correctly by requesting the health API, which returns inference server status:

```
curl -v localhost:8000/v2/health/ready
```

When the response provides 200 HTTP code – it means the service is available and running (see Figure 28).



```
*   Trying 127.0.0.1:8000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET /v2/health/ready HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Length: 0
< Content-Type: text/plain
```

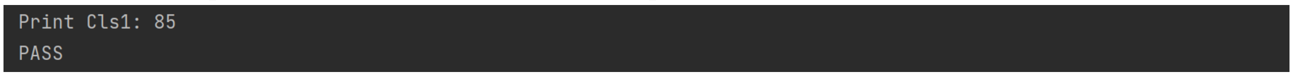
**Figure 28.** Successful Triton inference server health API response

### 2.5.2.2. Model Inference

The model inference part is a bit more complex as there is no information provided in the official documentation or on communities platforms about how to access model prediction API. After extensive research, a decision was made to use the Triton client prepared for Python applications. The Triton client provides data preprocessing, batching, inference, and postprocessing capabilities. Below, the provided command triggers model's inference:

```
python3 image_client.py -i grpc -u 0.0.0.0:8001 -m efficientnetB7 -s INCEPTION dog.jpg
```

The console output for successful classification is predicted, class:



```
Print Cls1: 85
PASS
```

**Figure 29.** Console output of the model inference API response when served via Triton

It should be mentioned that the experiment started with deploying the Tensorflow framework model, however, inference failed due to model framework incompatibility with the Cuda version as the Tensorflow GPU module supports the highest CUDA 10 (CUDA 11.6 was used in the experiment). Such incompatibilities cause unexpected system responses which are hard to grasp and might be complicated to solve, as decreasing CUDA version requires reinstalling of drivers and the latter - has a dependency on the operating system. This means that if we would already have a couple of models deployed on the server, the introduction of the new model may require a change to the previously set-up inference system change.

Official documentation [46-50]:

1. Triton client for model inference in Python application - https://github.com/triton-inference-server/client/blob/main/src/python/examples/image_client.py
2. Guidelines for integrating Triton client - https://github.com/triton-inference-server/client

## 2.5.3. TensorFlow Serving

TensorFlow Serving is one more serving system designed for ML models deployment to the production environment. Its is mostly supports TensorFlow framework models, however, there is also the possibility to extend it with the support of other framework models. TensorFlow Serving software, when isolated from other TensorFlow family applications, performs only serving functionality and it does not have the option to define batching process, input pre-processing, or post-processing as all these features are available on other TensorFlow packages.

TensorFlow's main requirement is to have Docker installed. Afterward, additional actions are needed only when deployment is planned to be performed using GPU resources. For this, NVIDIA drivers need to be up-to-date, and NVIDIA docker needs to be set up. The installation guidelines are the same as in Appendix 2.

TensorFlow Serving is easy to set up as it requires only pulling Docker image:

```
docker pull tensorflow/serving:latest-gpu
```

#### 2.5.3.1. Model Deployment

Model deployment is performed one by one via TensorFlow Serving, meaning, each model is deployed on a separate container. As pretrained models used for the experiment were downloaded from TensorFlow, their repository structure was already matching default TensorFlow Serving requirements:

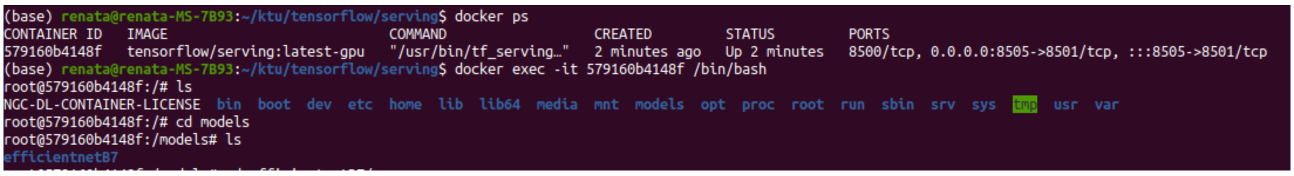
```
<model-repository-path>/
  <model-name>/
     1/
       assets/
       variables/
       saved_model.pb
```

**Figure 30.** TensorFlow model repository

The model deployment command consists of the GPU settings, defining the REST API port which will be opened for the prediction API (in case of this example it is 8505 port), model source path defined together with the model target folder which is created inside the container during deployment (see Figure 31):

```
nvidia-docker run --gpus all -p 8505:8501 \
--mount type=bind,source=/home/renata/ktu/tensorflow/serving/tmp/efficientnetB7, \
target=/models/efficientnetB7 \
-e MODEL_NAME=efficientnetB7 -t tensorflow/serving:latest-gpu &
```

This command also creates a default configuration file. Successful model deployment console output is provided in appendixes (Appendix 5).



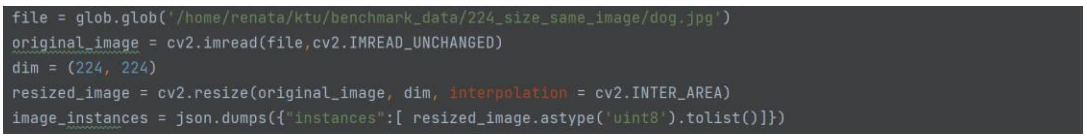
```
(base) renata@renata-MS-7B93:~/ktu/tensorflow/serving$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS          PORTS
579160b4148f   tensorflow/serving:latest-gpu   "/usr/bin/tf_serving…"   2 minutes ago   Up 2 minutes    8500/tcp, 0.0.0.0:8505->8501/tcp, :::8505->8501/tcp
(base) renata@renata-MS-7B93:~/ktu/tensorflow/serving$ docker exec -it 579160b4148f /bin/bash
root@579160b4148f:/# ls
NGC-DL-CONTAINER-LICENSE  bin  boot  dev  etc  home  lib  lib64  media  mnt  models  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@579160b4148f:/# cd models
root@579160b4148f:/models# ls
efficientnetB7
```

**Figure 31.** Docker Container and its Model Repository

#### 2.5.3.2. Model Inference

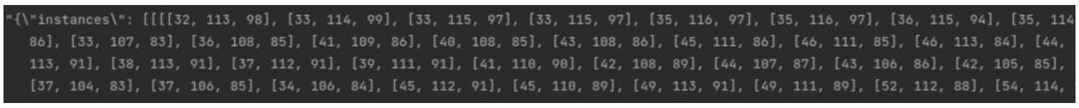
TensorFlow Serving makes the model available via the REST API. Classification models were used in this experiment, meaning, API should return a prediction of class based on the input image. TensorFlow REST API endpoint for classification model's predictions is http://localhost:{port}/v1/models/{model_name}:predict and POST method should be used for passing input data. Based on the above-provided example - the endpoint is http://localhost:8505/v1/models/efficientnetB7:predict. However, TensorFlow does not provide images preprocessing and postprocessing steps together with a model serving and it needs to be implemented separately; due to this reason, API expects image instances to be sent as a request body.

Image preprocessing was implemented in the Python client application. A sample of image conversion to instances is provided in the image below. This part of the experiment was challenging as TensorFlow does not provide guidelines on generating input data; however, there are many examples with request structure. Practitioners state that input preprocessing and post-processing options provided by TensorFlow slow down model inference in production, due to this reason – most specialists use TensorRT or other frameworks' proposed solutions [52].



```
file = glob.glob('/home/renata/ktu/benchmark_data/224_size_same_image/dog.jpg')
original_image = cv2.imread(file,cv2.IMREAD_UNCHANGED)
dim = (224, 224)
resized_image = cv2.resize(original_image, dim, interpolation = cv2.INTER_AREA)
image_instances = json.dumps({"instances":[ resized_image.astype('uint8').tolist()]})
```

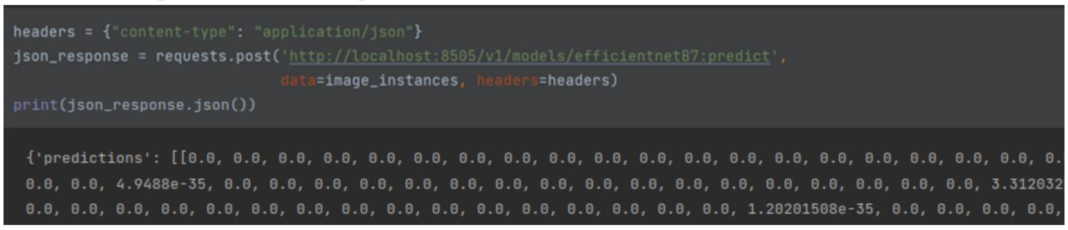
**Figure 32.** Code sample for image conversion to instances for model API served with TensorFlow Serving



```
"{\"instances\": [[[[32, 113, 98], [33, 114, 99], [33, 115, 97], [33, 115, 97], [35, 116, 97], [35, 116, 97], [36, 115, 94], [35, 114
  86], [33, 107, 83], [36, 108, 85], [41, 109, 86], [40, 108, 85], [43, 108, 86], [45, 111, 86], [46, 111, 85], [46, 113, 84], [44,
  113, 91], [38, 113, 91], [37, 112, 91], [39, 111, 91], [41, 110, 90], [42, 108, 89], [44, 107, 87], [43, 106, 86], [42, 105, 85],
  [37, 104, 83], [37, 106, 85], [34, 106, 84], [45, 112, 91], [45, 110, 89], [49, 113, 91], [49, 111, 89], [52, 112, 88], [54, 114,
```

**Figure 33.** TensorFlow inference API input data sample

Finally, converted input needs to be passed towards API:



```
headers = {"content-type": "application/json"}
json_response = requests.post('http://localhost:8505/v1/models/efficientnetB7:predict',
                              data=image_instances, headers=headers)
print(json_response.json())
```

```
{'predictions': [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0.0, 0.0, 4.9488e-35, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 3.312032
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.20201508e-35, 0.0, 0.0, 0.0, 0.0,
```

**Figure 34.** Code sample for model served via Tensorflow Serving inference

Response of API is also not processed as post-processing was not implemented in this experiment.

### 2.5.4. TorchServe

TorchServe serving tool mostly concentrates on PyTorch models deployment. It wraps the model in a set of REST APIs later available to be integrated into client applications. TorchServe itself provides not only inference but monitoring APIs as well, which endpoints are exposed for Graphana, Prometheus type of logs. It also shows memory usage and utilization (similarly to Nvidia-smi described in several chapters above). However, this experiment did not cover the usage of these additional features. The high-level architecture of the TorchServe software is provided in the annexes (Appendix 6).

TorchServe requires Docker, NVIDIA drivers, and NVIDIA Container Toolkit to be set up before serving models on Ubuntu OS using GPUs.  Installation guidelines are described in Appendices 2-3.

TorchServe installation takes only pulling TorchServe Docker image for GPU:

```
docker pull nvcr.io/pytorch/torchserve:0.3.0-gpu
```

#### 2.5.4.1. Model Deployment

TorchServe needs a model file and a code file in handler format to read/load/serve the model. Handler files can be created using template/example files available on the GitHub repository. TorchServe deployment requires a model archive (*.mar format) to be served.

First of all, the TorchScript model file should be passed to torch-model-archive. If the model is in PyTorch file format (*.pth), it should be converted to TorchScipt format.

Next, the model handler file should be generated. TorchServe has inbuilt handlers with custom inference logics described inside. These handlers support four tasks:

1. Image classification
2. Objects detection
3. Text classification
4. Image segmentation

As the experiment contains an image classification task, meaning, we need to adapt the *image_classifier* handler, which can be used for models trained on the ImageNet dataset. Handler defines RGB images as an input and provides a batch of top 5 predictions after inference.
Handlers samples (and even more) can be fetched by pulling the TorchServe Docker image:

```
git clone https://github.com/pytorch/serve.git
```

Then, in a repository /serve/examples/image_classification couple of models and handlers samples are provided. However, provided examples did not satisfy the need as they had custom sample' models related modifications. Due to this reason basic model handler (*myHandler.py* and *my_handler.py*) was used from the Zuppichini F. S. GitHub repository [51]. The only adjustment performed to the original files was defining that model inference should use GPU by adding *cuda()* function to the *my_handler.py* file (Appendix 7). Model deployment fails when TorschServe GPU Docker image is used if *cuda()* is not added to inference – that was one of the challenges encountered during the experiment.

After the model handler and checkpoint file (*.pt) were prepared, the model archive needed to be generated and its file was stored in the model-store directory:

```
torch-model-archiver --model-name efficientnetb7_cuda \
--version 1.0 \
--serialized-file efficientnetb7_cuda.pt \
--extra-files ./index_to_name.json,./MyHandler.py \
--handler my_handler.py  \
--export-path model-store -f
```

Finally, model deployment is performed by running the docker run command in the terminal:

```
nvidia-docker run --gpus all -e LRU_CACHE_CAPACITY=1 --rm -it \
-p 3012:8080 -p 3013:8081 -p 3014:8082 \
-v $(pwd)/model-store:/home/model-server/model-store pytorch/torchserve:latest-gpu \
torchserve --start --model-store model-store \
--models efficientnetb7=efficientnetb7_cuda.mar
```

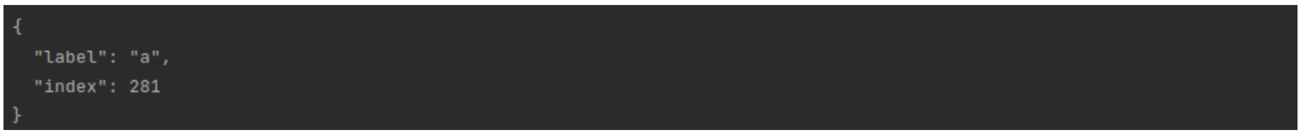
The console output for successful deployment will return the model inference endpoint and inference logs (Appendix 8).

### 2.5.4.2. Model Inference

TorchServe makes model inference available via REST API, as well as management and metrics APIs. Classification models were used in this experiment, meaning, API should return a prediction of class based on the input image. TorchServe REST API endpoint for classification model predictions is http://127.0.0.1:{port}/predictions/{model_name} and POST method should be used for passing input data. Based on the model deployment example provided above, the endpoint is http://127.0.0.1:3012/predictions/efficientnetb7 . As the model handler was provided with a model archive, this API provides input images preprocessing and postprocessing steps together with model serving. It means API expects raw image files to be passed and returns the predicted class in response:

```
curl -X POST http://127.0.0.1:3012/predictions/efficientnetb7 \
-T ~/ktu/benchmark_data/224_size_same_image/016.jpg
```

Console output provides a class for a given image:



```
{
  "label": "a",
  "index": 281
}
```

**Figure 35.** Console output of model inference API response when served via TorchServe

It was difficult to test inference as the first test of this experiment was performed when the model archive was generated using the original *my_handler.py* file. Also, the TorchScript model file was generated using standard TorchServe recommendations, meaning, the model was set to run on the CPU. Due to this reason, when such a model was deployed via TorchServe  - it created as many workers as the count of CPUs is (meaning – 12 workers). Each inference request then runs on a different worker, and when all 12 would be busy – the 13th request would be assigned to the 1st worker. The problem is that each worker allocates  ~ 600MB of RAM per inference request, and memory is not released from the worker until new requests do not come into the same worker. However, during this experiment,  after 10 requests – all 10 workers were busy, about 7GB of RAM was reserved (in addition, system processes, Docker container and PyCharm project used about 8GB), which caused the crash of the machine due to RAM shortage.

The resolution would be that even if GPU docker image and nvidia-docker have been defined in the deployment request – it does not mean that the inference will run on GPU. The model itself had to be adapted to run on GPU and updating TorchScript model file generation with *cuda()* function, as well as model handler file update, helped to proceed with the inference of models, deployed using TorchServe.

## 3. Research Results

### 3.1. Experiment Setup

As shown in Figure 36, three serving infrastructures were investigated in this experiment. TensorFlow Serving is the default serving environment for TensorFlow SavedModel format models, as well as the TorchScript serving format, which is the default for TorchServe software. On the other hand, Triton Inference Server supports many frameworks, from which, the default would be TensorRT serving format, but in this experiment, TorchScript format was used.

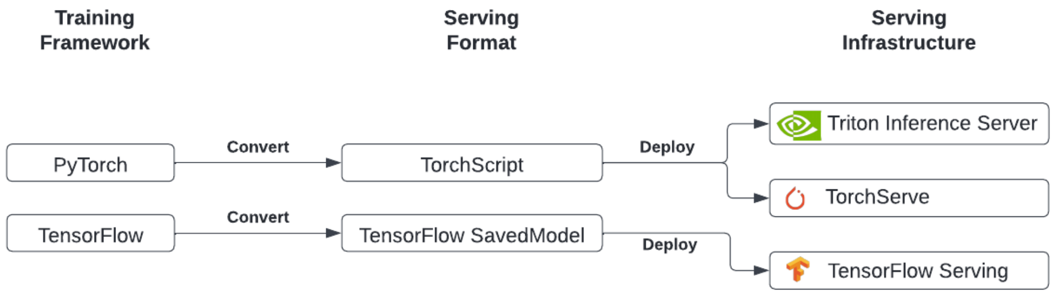


**Figure 36.** Three serving software infrastructures under test

### 3.2. Inference and Benchmarking System

The next step of this project was to set up an environment supporting all tools required for deploying computer vision models through TensorFlow Serving, Torchserve, and Nvidia Triton serving tools (installation and deployment guidelines are provided in the next chapter). These platforms were selected because they are open source, have extensive documentation, wide development communities, docker support, and are at the top of the most popular open-source tools for model deployment to production. The image below represents the final model serving and benchmarking environment used in this experiment (Figure 37). The whole inference and benchmarking environment was structured in a way to provide the same conditions for model serving via each deployment platform. All installations were performed before serving any model and each docker container was dropped before starting a new model serving.

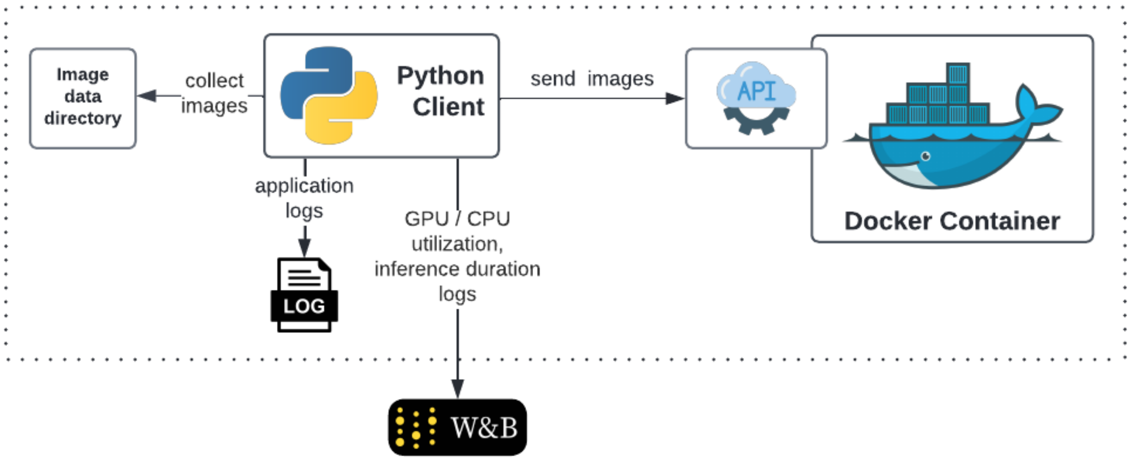


**Figure 37.** Model serving and benchmarking environment

### 3.2.1. Software

Data preprocessing, inference client, and benchmark code were written using Python language inside the PyCharm integrated development environment (IDE). Torchserve client files were pulled from the official GitHub repository (https://github.com/pytorch/serve/blob/master/docs/server.md). Models deployment was performed using Docker (more details provided in the next chapters).

### 3.2.2. Disk Capacity

Choosing the right hardware setup allowed us to perform a benchmark of model inference. It is widely accepted to use a graphics processing unit (GPU) for deep learning model training and inference if the model‘s inference speed is a bottleneck. However, central processing unit (CPU) usage is more attractive to most companies due to cost-saving. In addition, GPU usage is much less efficient in terms of heat dissipation and electricity usage. For performance optimization reasons, Nvidia GeForce RTX GPU was used for model inference in this experiment. CPU usage was minimal during the inference benchmark and did not exceed 43%, however, the experiment was conducted in a system having AMD Ryzen 5 5600x 6-core processor × 12 CPUs.

Along with processing units, disk capacity and random access memory (RAM) should be considered. As the experiment required multiple inferences, benchmarking scenarios to be performed that included image data stored on a disk, as well as each serving platform, brings docker images coupled with client packages required for running model inference. The machine having 15.5 GB RAM was used for an experiment, and it is not recommended to use lower-end models. RAM allocation during inference was a challenging part of the experiment as TorhServe automatically sets the number of workers equal to the number of CPUs and when inference is started – the first 10 workers consume all RAM. It is important to define whether computation needs to be performed on GPU or CPU during model deployment, and adapt the number of workers to RAM.

### 3.2.3. Operating System

Ubuntu 20.04 (64-bit) operating system was installed as NV164 family graphic cards and is stated, to have a performance issue on non-Linux systems. Another reason for choosing Linux based OS was the ease of setting up a Docker containerization platform for simulating deployment to production using the computer as a local server, especially when most of the documentation provided in setting up deployment platforms on Docker is Linux-based. OS type and version were changed multiple times during the experiment due to the above-mentioned reasons as well as drivers and libraries incompatibility with model serving tools (Nvidia Triton required CUDA 11.6 optimization library set-up, which is compatible only with Ubuntu 18.04 and 20.04).

### 3.2.4. Benchmarking Tools

Inference benchmarking required inference and performance indicators to be collected during multiple benchmark scenarios for further analysis of deployment platforms. Python package *time* was used for capturing inference duration, records were stored to *.csv files for later analysis and also - passed towards Wandb tool. Wandb is short for Weights & Biases machine learning tool which may perform models training, optimization, deployment, and monitoring functions. It has a wide range of great features to use during any kind of ML experiments, however, only the monitoring part was used in this research. The Weights & Biases tool was selected because of its ability to track hardware-related indicators such as CPU and GPU utilization, memory usage, etc., during inference (see Figure 38). Also, its ease of configuration and usage was fascinating - it was the easiest step of the whole experiment. Wandb requires an account to be created online (https://wandb.ai/) and the first page will already display quick-setup guidelines. More information about the tool and its usage can be found in Wandb's online documentation (https://docs.wandb.ai/) .

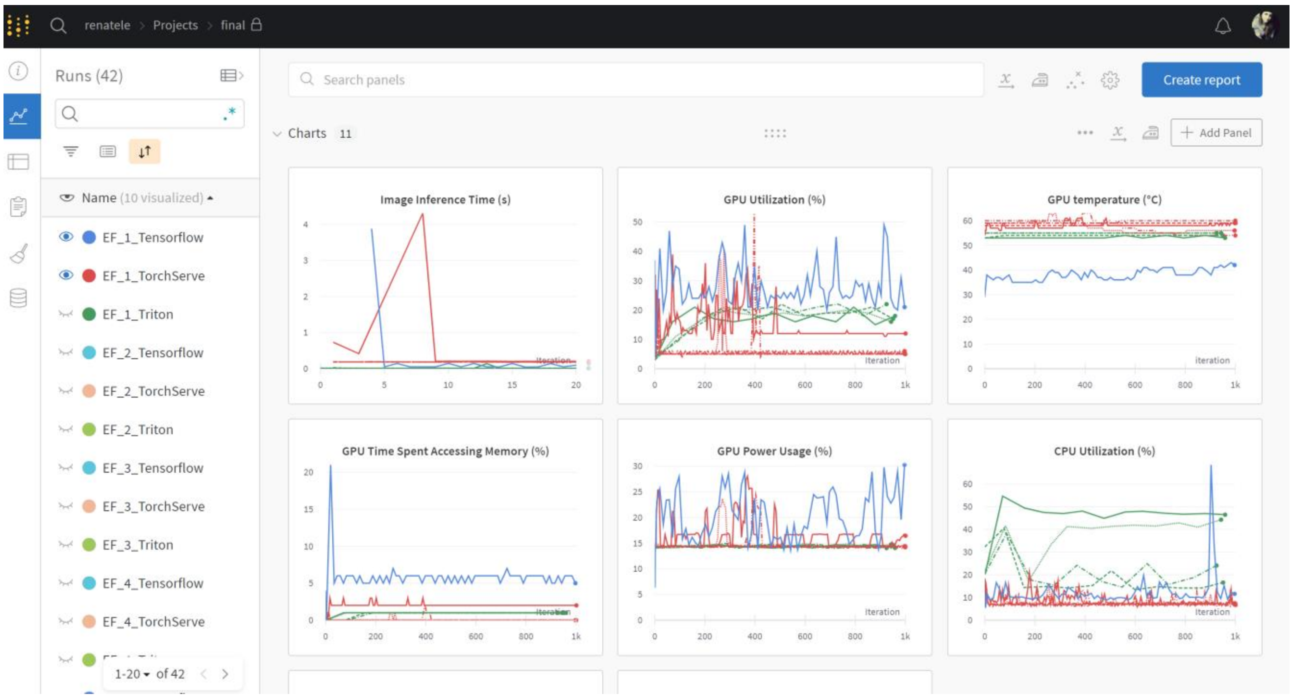


**Figure 38.** Wandb monitoring dashboards

## 3.3. Benchmarking Scenarios

The benchmark was only performed to provide an objective means of model performance when served in different serving software. The experiment consists of collecting inference latency and resource utilization measures, when the model is deployed in TensorFlow Serving, NVIDIA Triton, or TorchServe serving software, and performs identical computer vision tasks. The experiment includes image pre-processing and inference as most tools cannot exclude input preprocessing from the inference.

Several experiments scenarios were raised:

1. Model inference without a warmup, primarily resized the same image was passed for 1000 times;
2. Model inference after warmup, primarily resized the same image was passed for 1000 times;
3. Model inference after warmup, a collection of primarily resized 1000 different images were passed to the model;
4. Model inference after warmup; a collection of 1000 different various sizes images were passed to served model;

The idea behind these benchmarking scenarios was to identify how serving platforms' performance differs based on the served model and input data size and image content variety. Due to this reason, two image classification models were chosen, where both accept input data size 224 x 224 px. Furthermore, EfficientNet-B7 accepts any size images, only with the recommendation to resize them to 600 x 600 px before passing to the model, due to performance optimization.

In addition, the MobileNetV3-large model was quantized and its inference was tested on TorchServe and Triton Inference Server environments. TensorFlow Serving was outscoped, as it does not support quantized models – TensorFlow Lite Serving platform is dedicated for quantized models deployment, and as it is a separate serving platform, it was outscoped from the experiments.

Finally, the below table summarizes performed experiments. There were 4 scenarios performed on 3 different platforms. The models and input data dimensions used for the experiments are described in Table 4. 41 experiments were performed in total.

**Table 4**. Benchmarking scenarios per multiple serving platforms, models, and input data types

| **Scenarios** | | **TensorFlow Serving** | **TorchServe** | **Triton** |
|---|---|---|---|---|
| **1** | The same images with fixed dimensions passed to the model without warm-up | EfficientNet-B7 (224x224px);<br>EfficientNet-B7 (600x600px);<br>MobileNetV3-large (224x224px). | EfficientNet-B7 (224x224px);<br>EfficientNet-B7 (600x600px);<br>MobileNetV3-large (224x224px);<br>Quantized MobileNetV3-large (224x224px). | |
| **2** | The same images with fixed dimensions passed to the model after warm-up | | | |
| **3** | Different images with fixed dimensions were passed to the model after the warm-up. | | | |
| **4** | Different images with various dimensions resized during inference were passed to the model after warming up. | EfficientNet-B7 (input data resized to 224x224px during inference);<br>MobileNetV3-large (input data resized to 224x224px during inference). | EfficientNet-B7 (input data resized to 224x224px during inference);<br>MobileNetV3-large (input data resized to 224x224px during inference);<br>Quantized MobileNetV3-large (224x224px). | |

Upcoming chapters may have only scenario numbers defined in the results tables.

## 3.4. Dataset

The dataset contains images collected from multiple data sources to create as many diverse dataset as possible. Initially, collection of 10 000 images was collected, however, due to long inference duration and hardware limitations, dataset was minimized to 1000 images. Images sources:

- Collection of personal images (food, nature, animals, cars, people);
- Stanford dogs dataset - contains multiple breeds of dogs [68];
- Crawford cats dataset - contains multiple breeds of cats [69] ;
- The food dataset from DataVision – contains various food pictures [70].

One thousand images were selected randomly and static collection and all experiments were performed using the same dataset. Images were duplicated per three directories, where one of them contained images with original sizes, the second one – 224 x 224 px resized images, and third one –

600 x 600 px resized images. The below image shows couple of sample images from the dataset (see Figure 39).

**Figure 39.** Images samples from the dataset used for benchmarking

The below table shows images dataset statistics based on dimensions (Table 5), as well as their distribution is displayed in a plot below (Figure 40). Largest image used for benchmark is 3648 x 5472 px , smallest – 45 x 50 px.

**Table 5.** Statistics of images dataset used for benchmarking

| **Image Dimension** | **Median** | **Q1** | **Q3** | **Min** | **Max** |
|---|---|---|---|---|---|
| Height | 375 | 276 | 500 | 45 | 4160 |
| Width | 420 | 306 | 500 | 50 | 5472 |

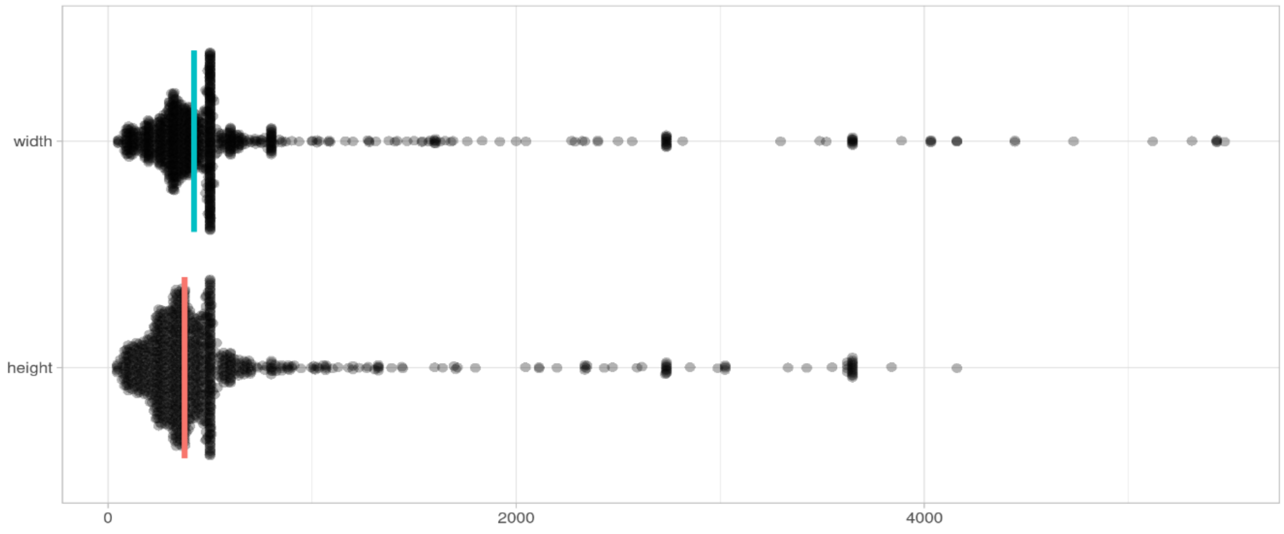


**Figure 40.** Distribution of dataset images dimensions (px) used for benchmarking

## 3.5. Results of Serving Software Benchmarking

### 3.5.1. Model Warmup Impact on Inference Time and Resources Allocation

Warming up the model is a common practice as it impacts inference at the moment of its initialization as well as a couple of first inference iterations. Based on the 1st benchmarking scenario (Table 4) experiments, it was visible model takes up to 3-8 requests to reach its optimal state, due to this reason 1-10 iterations were analyzed per each experiment. Figure 41 shows all performed experiment results without warming up the model first. The legend names are constructed from model name shortening (EF - EfficientNet-B7; MN – MobileNetV3-large; MN_Q – Quantized MobileNetV3-large), a number of scenario (Table 4), and serving platform names. In addition, one color represents one serving platform.

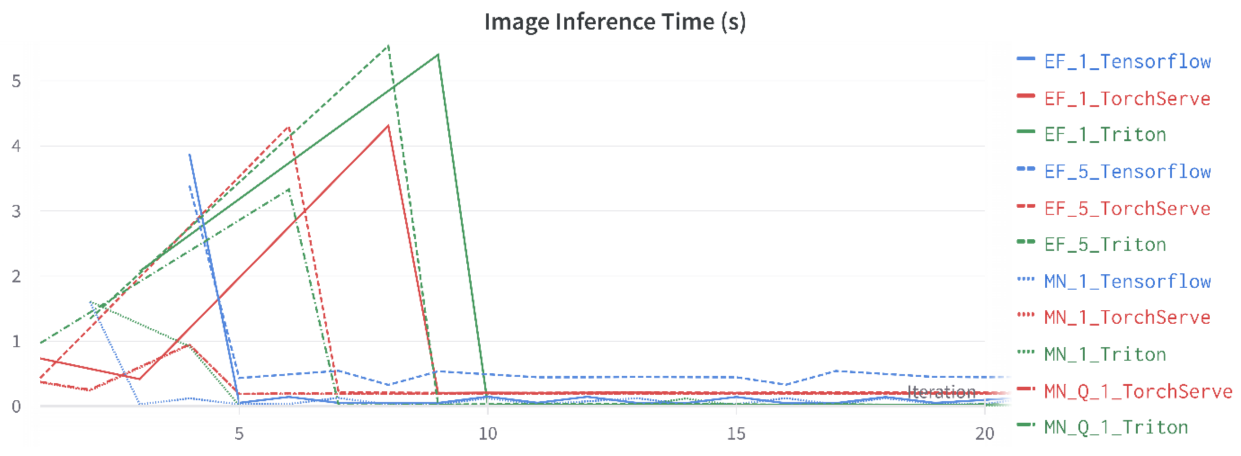


**Figure 41.** Inference time without model warmup per serving software

The table below (Table 6) visualizes the main statistics for each experiment. Triton Inference Server took the most of the time warming up the model, which caused the longest experiment duration, however, the median shows that this serving tool inference duration after warmup was the shortest one. Model quantization seems to have an impact longer model warming up time for Triton Inference Server (more than twice longer) with a slightly similar effect on the TorchServe serving platform. On the other hand, TensorFlow Serving total inference duration per 10 image iteration showed the shortest time when inference was performed with smaller images of 224px dimensions, rather than with 600px.

**Table 6.** Inference durations per 10 runs of experiments (scenario 1) with three serving platforms (green color stands for the fastest performance, red color – the most time consuming )

| **Architecture** | **EfficientNet-B7 (224 x 224 px)** | | | **EfficientNet B7 (600 x 600 px)** | | | **MobileNetV3 large** | | | **Quantized MobileNetV3 large** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Serving software** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Torch-Serve** | **Triton** |
| **Max** | 3.88 | 4.31 | 5.4 | 3.39 | 4.30 | 5.56 | 1.6 | 0.94 | 1.61 | 0.95 | 3.33 |
| **3rd quartile** | 0.15 | 0.42 | 0.03 | 0.48 | 0.21 | 0.03 | 0.12 | 0.24 | 0.01 | 0.26 | 0.02 |
| **Median** | 0.05 | 0.21 | 0.02 | 0.45 | 0.20 | 0.02 | 0.03 | 0.19 | 0.01 | 0.2 | 0.01 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1st quartile** | 0.05 | 0.21 | 0.02 | 0.44 | 0.20 | 0.02 | 0.03 | 0.19 | 0.01 | 0.19 | 0.01 |
| **Average** | 0.09 | 0.21 | 0.03 | 0.47 | 0.21 | 0.03 | 0.07 | 0.19 | 0.01 | 0.19 | 0.01 |
| **Inference time (10 iterations)** | 4.63 | 6.9 | 7.65 | 7.35 | 6.42 | 7.13 | 2.16 | 2.89 | 2.61 | 2.94 | 4.41 |

In addition, Appendix 9 shows GPU and CPU utilization during experiments. GPU usage (~43-45% with EfficientNet-B7 and ~16% with MobileNetV3-large) is highest when the model runs on Triton Inference Server, however, it is the most stable, compared with TorchServe and TensorFlow Serving. The first and third figures in this appendix show that GPU takes the most of the time accessing memory during the first couple of iterations when the model served with TensorFlow Serving, rather than other serving platforms. Knowing that Nvidia Triton Inference Server has lots of optimizations related to GPU usage – it is visible from the graphs, compared with other serving tools.

The second scenario (Table 4) was performed already after the model was warmed up and as Figure 42 shows – the first 10 inference iterations do not have such a huge difference from the rest of the inference times. We can also see, that models running on Triton Inference Server perform quicker than the ones which run on TorchServe or TensorFlow Serving. On the other hand, TensorFlow Serving performs better than TorchServe, when input images are 224px sizes (EF_2_Tensorflow vs EF_2_TorchServe) rather than 600px (EF_6_Tensorflow vs EF_6_TorchServe).

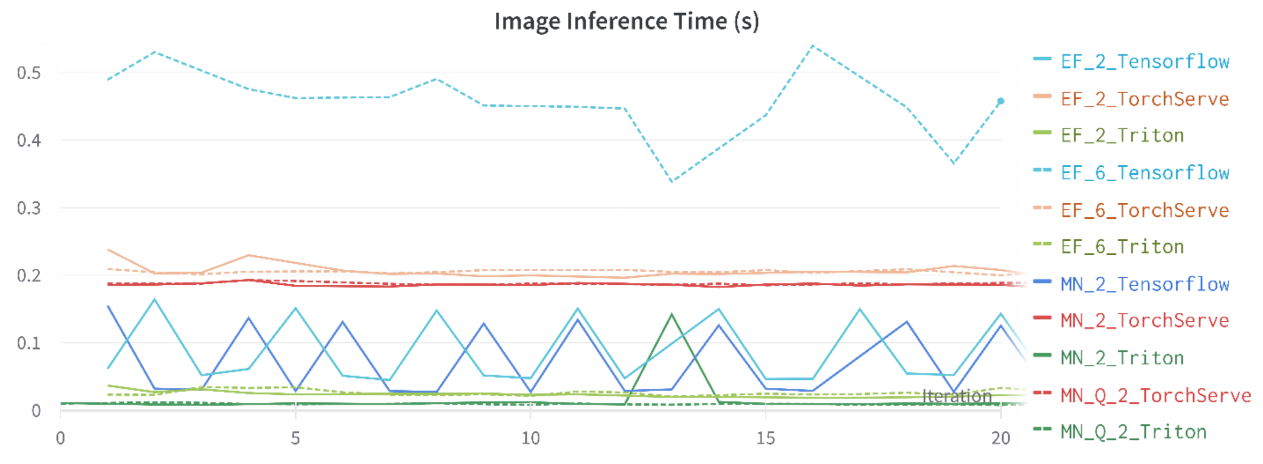


**Figure 42.** Inference time after model warmed up per serving software

### 3.5.2. Input Data Impact on Inference Time and Resources Allocation

This chapter covers scenarios 2-4 (Table 4), where the main conditions were changing input data bypassing the same or different kinds of images, as well as image sizes. Experiments results covering inference duration analysis are provided in Table 7 .

Scenario number 2 covers model inference when the already resized same image is passed to the served model. As we see in a table, the results of model performance when 224px images are passed are the same in perspective of serving platforms: models deployed with Triton Inference Server performs classification task quicker than the ones, deployed using TorchServe or TensorFlow Serving. The inference results of TorchServe models are more than 10 times worse than those of

Triton. On the other hand, when we look at inference performed with 600px dimensions images – TorchServe performed better than TensorFlow Serving.

Similar results were obtained when performing experiments with the third scenario, where a collection of various, already resized images was passed to the model. Lastly, the fourth scenario covered cases when various sizes of images were passed to the model, and image resizing was performed during the image preprocessing step. Triton Inference Server still handles such tasks the quickest, however, TensorFlow Serving showed twice worse results than TorchServe.

**Table 7**. Inference duration statistics per experiments (scenario 2-4) and serving platforms (green color stands for quickest performance, red color – the most time consuming )

| **Architecture** | | **EfficientNet-B7** (224 x 224 px) | | | **EfficientNet B7** (600 x 600 px) | | | **MobileNetV3 large** | | | **Quantized MobileNetV3 large** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Serving software** | | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Torch-Serve** | **Triton** |
| **2nd scenario** | 3rd quartile | 0.14 | 0.20 | 0.02 | 0.56 | 0.21 | 0.03 | 0.13 | 0.19 | 0.01 | 0.19 | 0.01 |
| | Median | 0.05 | 0.20 | 0.02 | 0.44 | 0.20 | 0.02 | 0.03 | 0.19 | 0.01 | 0.19 | 0.01 |
| | 1st quartile | 0.05 | 0.20 | 0.02 | 0.34 | 0.20 | 0.02 | 0.03 | 0.18 | 0.01 | 0.18 | 0.01 |
| | Average | 0.09 | 0.20 | 0.02 | 0.45 | 0.20 | 0.03 | 0.07 | 0.19 | 0.01 | 0.19 | 0.01 |
| | Experiment time | 80.28 | 167.49 | 22.87 | 400.37 | 204.21 | 26.29 | 64.12 | 156.34 | 12.57 | 157.25 | 11.51 |
| **3rd scenario** | 3rd quartile | 0.15 | 0.20 | 0.02 | 0.57 | 0.21 | 0.03 | 0.13 | 0.19 | 0.01 | 0.19 | 0.01 |
| | Median | 0.05 | 0.20 | 0.02 | 0.45 | 0.20 | 0.02 | 0.03 | 0.19 | 0.01 | 0.19 | 0.01 |
| | 1st quartile | 0.05 | 0.20 | 0.02 | 0.35 | 0.20 | 0.02 | 0.03 | 0.18 | 0.01 | 0.18 | 0.01 |
| | Average | 0.09 | 0.20 | 0.02 | 0.46 | 0.20 | 0.03 | 0.07 | 0.19 | 0.01 | 0.19 | 0.01 |
| | Experiment time | 87.42 | 202.43 | 23.13 | 409.35 | 203.73 | 25.40 | 67.01 | 185.42 | 11.44 | 186.29 | 10.44 |
| **4th scenario** | 3rd quartile | 0.48 | 0.21 | 0.03 | - | - | - | 0.13 | 0.19 | 0.01 | 0.19 | 0.01 |
| | Median | 0.45 | 0.21 | 0.02 | - | - | - | 0.03 | 0.19 | 0.01 | 0.19 | 0.01 |
| | 1st quartile | 0.44 | 0.21 | 0.02 | - | - | - | 0.03 | 0.18 | 0.01 | 0.18 | 0.01 |
| | Average | 0.46 | 0.21 | 0.02 | - | - | - | 0.07 | 0.19 | 0.01 | 0.19 | 0.01 |
| | Experiment time | 405.04 | 212.69 | 27.26 | - | - | - | 65.72 | 185.31 | 12.20 | 186.12 | 11.62 |

While performing all experiments – GPU and CPU measurements were taken. When the model was deployed using GPU there is usually an expectation to have maximum GPU utilization as most of the computations are performed there. During experiments, GPU utilization reached a maximum of 86% and CPU utilization jumped to 77% when the model was deployed via TensorFlow Serving and the fourth scenario was used in the experiment (passing various sizes of images) (Appendix 12). However, all the rest experiments kept quite similar GPU and CPU usage measurements (see Appendix 10-11), similar to the ones provided in figure 43. A sample of GPU and CPU usage is provided in Tables 8 and 9, where the median shows Triton having the biggest resource usage, compared to other serving platforms. TorchServe resource needs are the smallest for classification model inference.

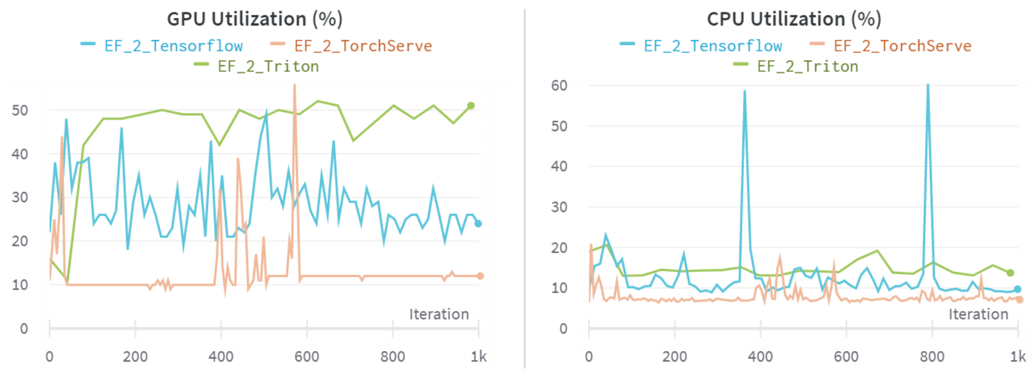


**Figure 43**. Scenario #2: GPU (left) and CPU (right) utilization during the experiment

**Table 8**. GPU utilization (%) during experiments (scenario 2) performed with each serving software (green color stands for lowest usage, red color – highest usage )

| **Architecture** | **EfficientNet-B7** (224 x 224 px) | | | **EfficientNet-B7** (600 x 600 px) | | | **MobileNetV3-large** | | | **Quantized MobileNetV3-large** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Serving software** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Torch-Serve** | **Triton** |
| **Min** | 18 | 9 | 11 | 7 | 7 | 11 | 4 | 4 | 3 | 4 | 3 |
| **Max** | 49 | 56 | 52 | 46 | 40 | 52 | 45 | 19 | 21 | 12 | 18 |
| **Median** | 26 | 12 | 49 | 15 | 12 | 44 | 6.5 | 5 | 17 | 5 | 14 |
| **Average** | 28.52 | 12.62 | 45.3 | 17.3 | 12.21 | 42.35 | 11.11 | 5.48 | 16.38 | 5.09 | 14.1 |

**Table 9.** CPU utilization (%) during experiments (scenario 2) performed with each serving software (green color stands for lowest usage, red color – highest usage )

| **Architecture** | **EfficientNet-B7** (224 x 224 px) | | | **EfficientNet-B7** (600 x 600 px) | | | **MobileNetV3-large** | | | **Quantized MobileNetV3-large** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Serving software** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Tensor-Flow** | **Torch-Serve** | **Triton** | **Torch-Serve** | **Triton** |
| **Min** | 8.9 | 6.4 | 13 | 7.7 | 6.5 | 13.7 | 5.5 | 6.2 | 9 | 5.8 | 8 |
| **Max** | 60.4 | 20.9 | 20.6 | 43.2 | 17.9 | 30.1 | 24.4 | 17.5 | 20.6 | 18.4 | 19.1 |
| **Median** | 10.50 | 7.20 | 14.10 | 9.00 | 7.50 | 16.90 | 10.75 | 7.10 | 16 | 7.10 | 15 |
| **Average** | 12.73 | 7.74 | 14.89 | 9.69 | 7.74 | 18.25 | 12.45 | 7.27 | 14.89 | 7.32 | 13.11 |

The experiment showed that Triton Inference Server and TorchServe had the greatest differences in instance resource usage. Due to this reason, these two platforms were chosen for further analysis based on 2nd scenario data. A two-sample Student's t-test assuming unequal variances was performed to test the hypothesis that the resulting mean GPU usage of the Triton Inference Server and TorchServe were equal.

**Table 10**. A two-sample Student's t-test results of hypothesis that the resulting mean GPU usage of the Triton Inference Server and TorchServe were equal

| t-test for Equality of Means | Triton Inference Server | TorchServe |
|---|---|---|
| Mean | 42.34615385 | 12.21078431 |
| Variance | 64.95538462 | 10.74845455 |
| Observations | 26 | 204 |
| Hypothesized Mean Difference | 0 | |
| df | 26 | |
| t Stat | 18.86790282 | |
| P(T<=t) one-tail | 5.38632*10-17 | |
| t Critical one-tail | 1.70561792 | |
| P(T<=t) two-tail | 1.07726*10-16 | |
| t Critical two-tail | 2.055529439 | |

The mean usage of GPU observed for Triton (M = 42.3 %, SD = 8.05, N = 26) was significantly higher than that observed for TorchServe (M = 12.21%, SD = 3.27, N =204), t(26) = 18.87, p=0.000.

In addition, a GPU pricing comparison was performed to understand the serving tool's advantages and disadvantages when using cloud resources. As cloud instance prices are provided per hour, data of the experiment having the biggest inference duration difference was taken (Table 11). We can see the Triton Inference Server model to perform ~ 16.11 times quicker than the TensorFlow Serving model. Meaning, that if we would have 24 hours of video footage with 24 frames per second rate (2 073 600 images/frames in total) and we would like to analyze each frame by passing it to the model – it would take ~ 235.79 hours to process data for the model served on TensorFlow Serving and ~ 14.63 hours, if it is served on Triton Inference Server.

**Table 11.** Experiment having the greatest inference duration difference per 1000 observations between three serving platforms

| Scenario | TensorFlow | TorchServe | Triton |
|---|---|---|---|
| Experiment time for 3rd scenario (1000 same size random images) | 409.35 | 203.73 | 25.40 |

In case cloud services provider suggests the least expensive plan for GPU-oriented instance dedicated to running ML algorithms on Nvidia RTX technology, the price would be 1.25 Eur per hour (lambdalabs.com pricing). Based on that, 24 hours of video footage processing when served on TensorFlow Serving would cost ~ 294.74 Eur, and on Triton Inference Server - ~ 18.29 Eur.

### 3.5.3. Model Quantization Impact on Inference Time and Resources Utilization

As quantization adds lightness to the model, it is expected to have inference time and resource utilization reduced. MobileNetV3-large model quantization effect was investigated only on Triton Inference Server and TorchServe serving software. When we look at scenarios 2 and 3 for standard and quantized model inference, we see a contrast between both serving platforms. Figure 44 shows these models' inference when running on TorchServe and we can see, that the most efficient inference is with the standard MobileNetV3-large model and the quantized model has a slight delay.

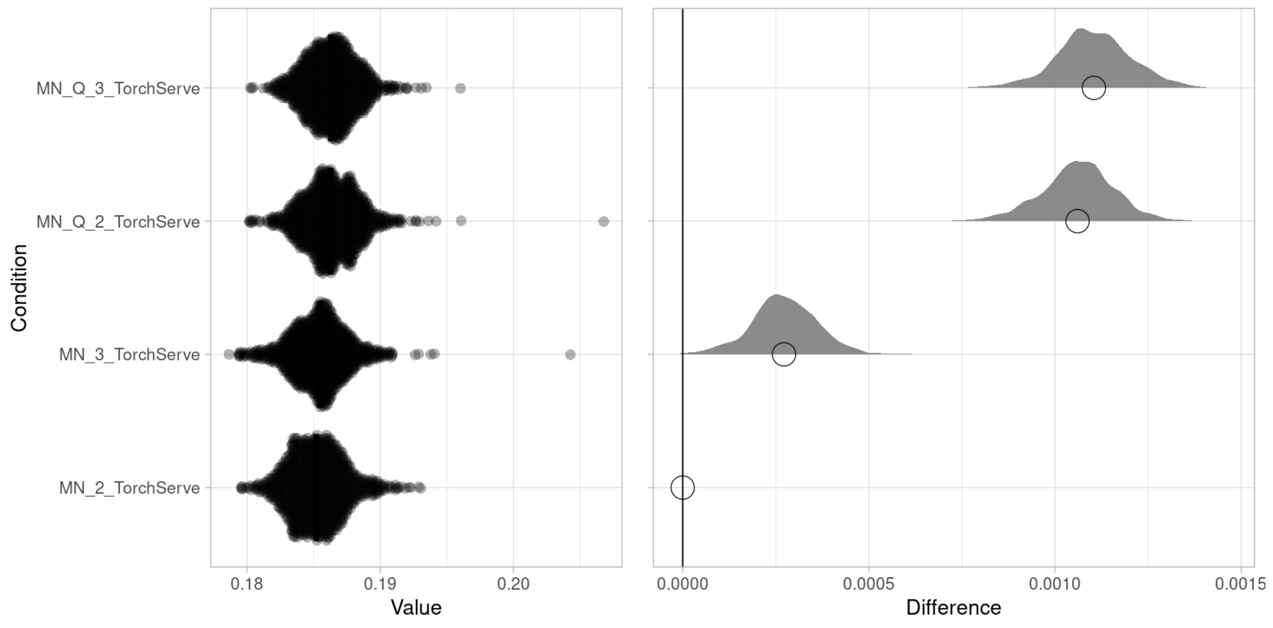


**Figure 44.** MobileNetV3-large model versus it's quantized variant, both served with TorchServe

On the other hand, Figure 45 shows both experiments with quantized models performing quicker rather than regular model inference.

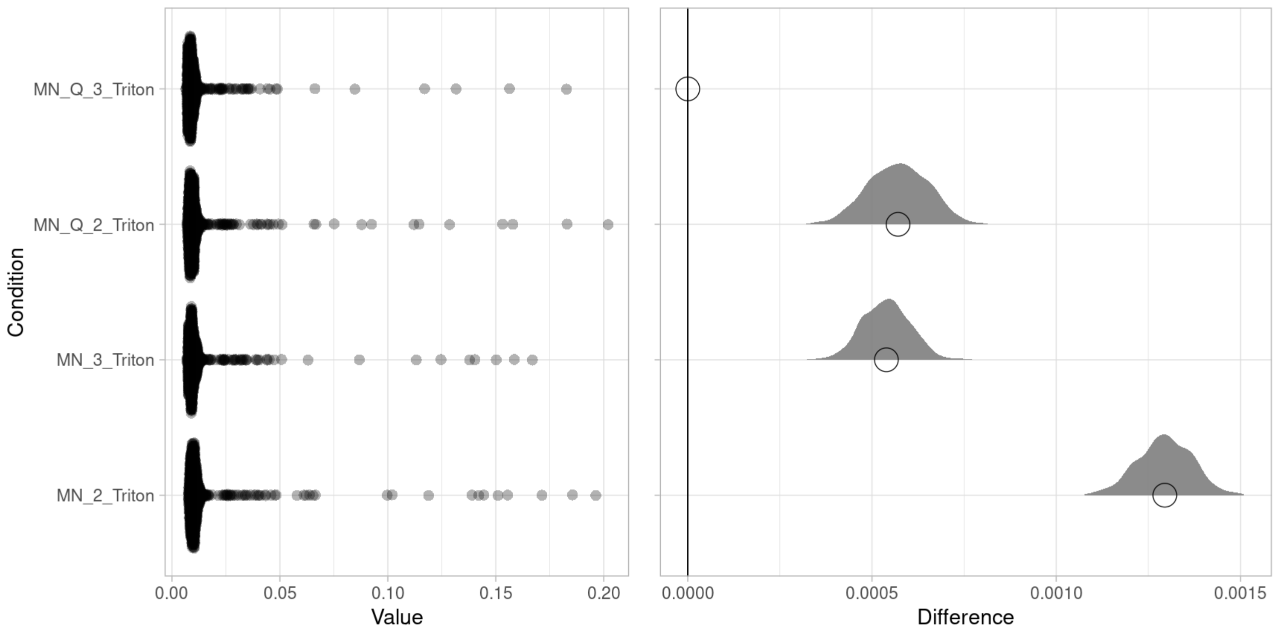


**Figure 45.** MobileNetV3-large model versus quantized same model, both served with Triton Inference Server

A two-sample Student’s t-test was performed assuming equal variances to test the hypothesis that the resulting mean inference duration of the quantized and not quantized MobileNetV3-large model was equal when served on TorchServe (Table 12), as well as Triton Inference Server (Table 13).

**Table 12.** A two-sample Student's t-test for quantized and not quantized MobileNetV3-large model inference duration when models served on TorchServe

| t-test for Equality of Means | Quantized MobileNetV3-large | MobileNetV3-large |
|---|---|---|
| Mean | 0.186330925 | 0.18527195 |
| Variance | $4.62236*10^{-06}$ | $4.41266*10^{-06}$ |
| Observations | 1000 | 1000 |
| Pooled Variance | $4.51751*10^{-06}$ | |
| Hypothesized Mean Difference | 0 | |
| df | 1998 | |
| t Stat | 11.14091739 | |
| P(T<=t) one-tail | $2.60435*10^{-28}$ | |
| t Critical one-tail | 1.645617012 | |
| P(T<=t) two-tail | $5.20869*10^{-28}$ | |
| t Critical two-tail | 1.96115261 | |

The mean inference duration observed for quantized model served with TorchServe (M = 0.19 %, SD = 0.002, N = 1000) was significantly higher than that observed for not quantized model (M = 0.19 %, SD = 0.002, N =1000), t(1998) = 11.14, p=0.000.

**Table 13.** A two-sample Student's t-test for quantized and not quantized MobileNetV3-large model inference duration when models served on Triton Inference Server

| t-test for Equality of Means | Quantized MobileNetV3-large | MobileNetV3-large |
|---|---|---|
| Mean | 0.011613925 | 0.01271 |
| Variance | 0.000187079 | 0.000244 |
| Observations | 1000 | 1000 |
| Pooled Variance | 0.000215334 | |
| Hypothesized Mean Difference | 0 | |
| df | 1998 | |
| t Stat | -1.670474009 | |
| P(T<=t) one-tail | 0.047491121 | |
| t Critical one-tail | 1.64561663 | |
| P(T<=t) two-tail | 0.094982241 | |
| t Critical two-tail | 1.961152015 | |

The mean inference duration observed for quantized model served with Triton Inference Server (M = 0.01 %, SD = 0.014, N = 1000) was not significantly higher than that observed for not quantized model (M = 0.01 %, SD = 0.016, N =1000), t(1998) = -1.67, p=0.095.

## 3.6. Benchmarking and Results Overview

From the very beginning, selecting software for inference benchmarking as well as its installation was complex for some serving platforms. TensorFlow Serving software has the easiest installation process for model inference, lots of documentation and samples are provided online. On the other hand, TensorFlow does not have a single tool covering all the inference steps such as data pre-processing and post-processing, and quantized models deployment, which makes it hard to identify what kind of TensorFlow software or additional packages are needed to have fluent inference flow. Triton Inference Server has a set of installation preconditions, which takes a lot of time to install and requires a specific set of hardware and software to run on the machine to install all drivers and toolkits. This software and packages installation is time-consuming, however, it provides full disclosure of Triton Inference Server properties, GPU optimization settings, and what affects the inference time of deployed models.

Furthermore, TorchServe, as well as Triton Inference Server requires the model to be converted to a specified format before deployment. There is a lack of online information on this topic and it may reduce model efficiency if performed improperly due to the configuration file required for model conversion. On the other hand, both Triton Inference Server and TorchServe come with the client, which needs to be integrated into the application to perform model inference. The client includes data pre-processing, batching, inference optimization, and post-processing steps. Both TorchServe and Triton Inference Server have incomplete documentation on error handling when a deployment fails due to the model's incompatibility with deployment parameters.

Warming up the classification model deployed on Triton Inference Server takes more time than TensorFlow Serving or TorchServe, however, these platforms provide longer inference time compared to Triton. Meaning, that Triton Inference Server is better adapted for image classification model serving on GPU rather than TorchServe and TensorFlow Serving. On the other hand, TensorFlow Serving is more efficient when inference is performed with smaller dimensions images rather than TorchServe. The latter showed better results than TensorFlow Serving when model inference was performed with larger images.

Analysis showed Triton's high GPU utilization compared with the other two platforms. The results of the T-test showed that the usage of GPU observed for Triton was significantly higher than that observed for TorchServe. In addition, cloud instances pricing showed the model will perform twenty-four-hour video footage analysis per ~ 294.74 Eur when served with TensorFlow Serving, and ~ 18.29 Eur when served with Triton Inference Server.

When comparing inference time for a quantized classification model, the initial look was for a higher effect on inference time when deployment is performed using Nvidia Triton Inference Server, rather than TorchServe. However, the t-test rejected our hypothesis and confirmed the irrelevance of inference time difference between quantized and not quantized models for Triton.

## Summary and Conclusion

1. The study conducted revealed that most researchers emphasized image recognition to have the highest social impact in the field of medicine, as it shapes the future of disease prevention methods in the form of anomaly detection from medical imaging, or virus research and vaccination. Correspondingly, researchers in the entertainment and robotics fields show a high interest in deep learning technology concerning the digging of information for the user and the provision of user-oriented content or automating time-consuming human-like tasks with the help of robots.
2. Many researchers point out the complexity of model serving caused by the variety of hardware, software, and optimization libraries dedicated to this task, and the lack of their compatibility. For this reason, cloud solutions are highly in demand by providing end-to-end model serving process management. Today, TensorFlow Serving, Triton Inference Server, and TorchServe are highly popular open-source software for model serving on-premises. Analysis and experiment resulted in a conclusion about TensorFlow as being the easiest software in the scope of the installation process, however, Triton Inference Server and TorchServe were more convenient to use for inference due to additional features included in these software clients.
3. The research covered analysis of EfficientNet and MobileNet architectures widely used as the basis for most image classification-related research. Both architectures have several versions, each of which includes an additional optimization layer. MobileNet models accept fixed-size input images and are designed for mobile devices. On the other hand, EfficientNet accepts various dimensions of input images which adds flexibility for future research.
4. The experiments carried out showed Triton Inference Server to be up to 16 times more efficient in terms of inference duration for image classification models compared to other software. On the other hand, it consumed up to 75% more GPU resources during inference compared to the other two software. Experiments also revealed that models deployed with TensorFlow Serving and TorchServe inference time have a significant dependency on input image sizes. In addition, models served with both software warm up faster than those served with Triton. The cost of cloud instances for the classification of 24 hour footage frames would cost ~ 294.74 Eur when the model was served with TensorFlow Serving, and ~ 18.29 Eur when served with Triton.
5. Student's t-test identified the mean runtime of MobileNet model being not significantly higher compared with its quantized variant for Triton Inference Server (p-value=0.095), whereas mean runtimes were significantly different for TorchServe (p-value=0.000).
6. General recommendations or guidelines for selecting serving software for efficient image classification in production:

   – Triton Inference Server running on GPU is recommended to use for reducing image classification model inference time;
   – The model warmup is recommended to perform with all serving platforms to add efficiency for a couple of first requests and reduce total inference time;
   – In case if disk capacity is an issue, TrochServe software is recommended for inference, as experiments showed that it utilizes GPU and CPU resources the least; however, it prolongs inference time, compared to Triton Inference Server.
   – When considering TensorFlow Serving and TorchServe software for model serving, it is recommended to consider input image dimensions that will be used in a defined business scenario, as TensorFlow Serving is more efficient when serving smaller dimension images, and TorchServe – larger ones.

## List of References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2016. doi: 10.1109/cvpr.2016.90.

[2] L. Weng, "An overview of deep learning for curious people," *Github.io*, 2017. https://lilianweng.github.io/posts/2017-06-21-overview/ (accessed May 16, 2021).

[3] E. Roberts, "Neural Networks - History," *Stanford University*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html (accessed May 02, 2021).

[4] B. Predić, U. Vukić, M. Saračević, D. Karabašević, and D. Stanujkić, "The possibility of combining and implementing deep neural network compression methods," *Axioms*, vol. 11, no. 5, p. 229, 2022, doi: 10.3390/axioms11050229.

[5] Y. Bengio and S. Bengio, "Modeling high-dimensional discrete data with multi-layer neural networks," in *Advances in Neural Information Processing Systems*, 2000, pp. 400–406.

[6] Y. Luo, C. Yao, Y. Mo, B. Xie, G. Yang, and H. Gui, "A creative approach to understanding the hidden information within the business data using Deep Learning," *Inference Process Management*, vol. 58, no. 5, p. 102615, 2021, doi: 10.1016/j.ipm.2021.102615.

[7] V. Thejaswini, "Artificial intelligence in Healthcare," *Acldigital.com*, 2020. https://www.acldigital.com/blogs/artificial-intelligence-healthcare (accessed Jan. 02, 2022).

[8] A. M. Nancy and M. R., "A Review on Unstructured Data in Medicine," *Journal Of critical rev*, vol. 7, no. 13, pp. 2202–2208, 2020, Accessed: Jan. 01, 2022. [Online]. Available: https://www.researchgate.net/profile/Maria-Nancy-3/publication/343655656_A_REVIEW_ON_UNSTRUCTURED_DATA_IN_MEDICAL_DATA/links/5f36a848a6fdcccc43c6b2d5/A-REVIEW-ON-UNSTRUCTURED-DATA-IN-MEDICAL-DATA.pdf

[9] A. K. Arshadi, J. Webb, M. Salem, M. Salem, and J. S. Yuan, "Artificial Intelligence for COVID-19 Drug Discovery and Vaccine Development," Researchgate.net. https://www.researchgate.net/publication/343718521_Artificial_Intelligence_for_COVID-19_Drug_Discovery_and_Vaccine_Development (accessed Nov. 04, 2021).

[10] J. Wu et al., "DeepHLApan: A deep learning approach for neoantigen prediction considering both HLA-peptide binding and immunogenicity," *Front. Immunol.*, vol. 10, p. 2559, 2019, doi: 10.3389/fimmu.2019.02559.

[11] A. Biswal, "Top 10 Deep Learning applications used across industries," *Simplilearn.com*, 2021. https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-applications (accessed Jan. 01, 2022).

[12] L. H. Lee et al., "All one needs to know about metaverse: A complete survey on technological singularity, virtual ecosystem, and research agenda," arXiv [cs.CY], 2021. [Online]. Available: http://arxiv.org/abs/2110.05352

[13] J. M. Ghlionn, "Metaverse — the world not prepared for dangers it poses," *TRT World*, 2021. https://www.trtworld.com/perspectives/metaverse-the-world-not-prepared-for-dangers-it-poses-52701 (accessed Jan. 02, 2022).

[14] K. B. Prakash, Programming with TensorFlow - Solution for Edge Computing Applications. 2021. [Online]. Available:

https://books.google.lt/books?id=VcUWEAAAQBAJ&pg=PA69&lpg=PA69&dq=oxford+google+lipnet+93%25&source=bl&ots=ksijjzlxH9&sig=ACfU3U2b8JUNHU5hxbGFUGGKfEsUnyUaHw&hl=lt&sa=X&ved=2ahUKEwibrqyfl5P1AhUeQ_EDHRHQC08Q6AF6BAgDEAM#v=onepage&q=oxford%20google%20lipnet%2093%25&f=false

[15] C. Nott and E. O'Brien, "Wimbledon serves more AI in 2018," *IBM*, 2018. https://www.ibm.com/blogs/game-changers/wimbledon-serves-ai-2018/ (accessed Jan. 02, 2022).

[16] M. Cahillane and T. Collins, "Boston Dynamics to begin selling its robot dog Spot next year," *Daily Mail*, 2018. Accessed: Jan. 02, 2022. [Online]. Available: https://www.dailymail.co.uk/sciencetech/article-5727633/Boston-Dynamics-begin-selling-robot-dog-Spot-year-wont-say-cost.html

[17] C. Coleman et al., "DAWNBench: An End-to-End Deep Learning Benchmark and Competition," *Stanford University*, 2018. https://cs.stanford.edu/~matei/papers/2018/sysml_dawnbench.pdf (accessed Mar. 10, 2022).

[18] D. Klabjan and X. Zhu, "Neural network retraining for model serving," *arXiv [cs.LG]*, 2020. [Online]. Available: http://arxiv.org/abs/2004.14203

[19] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "{DeepCPU}: Serving {RNN-based} Deep Learning Models 10x Faster," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 951–965.

[20] "The difference between deep learning training and inference," *Intel*, 2020. https://www.intel.com/content/www/us/en/artificial-intelligence/posts/deep-learning-training-and-inference.html (accessed Jun. 24, 2021).

[21] H. Zhang, Y. Huang, Y. Wen, J. Yin, and K. Guan, "InferBench: Understanding deep learning inference serving with an automatic benchmarking system," *arXiv [cs.LG]*, 2020. [Online]. Available: http://arxiv.org/abs/2011.02327

[22] V. J. Reddi et al., "MLPerf Inference Benchmark," 2020.

[23] Y.H. Zhang, J. Pu, W.-M. Hwu, and J. Xiong, "MLHarness: A Scalable Benchmarking System for MLCommons," arXiv [cs.LG], 2021. [Online]. Available: http://arxiv.org/abs/2111.05231

[24] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software engineering challenges of deep learning," 2018. doi: 10.1109/seaa.2018.00018.

[25] D. Sato, A. Wider, and C. Windheuser, "Continuous Delivery for Machine Learning," 2019. https://martinfowler.com/articles/cd4ml.html (accessed Jun. 30, 2021).

[26] A. Dhinakaran, "ML Infrastructure tools for production," *Towards Data Science*, 2020. https://towardsdatascience.com/ml-infrastructure-tools-for-production-part-2-model-deployment-and-serving-fcfc75c4a362 (accessed Jun. 20, 2021).

[27] "MLOps platforms compared," *Valohai.com*. https://valohai.com/mlops-platforms-compared/ (accessed Jun. 30, 2021).

[28] S. Karayev, J. Tobin, and P. Abbeel, "Full Stack Deep Learning," *UC Berkeley methodical material*, 2021.

[29] A. Ignatov et al., "AI benchmark: All about deep learning on smartphones in 2019," *arXiv [cs.PF]*, 2019. [Online]. Available: http://arxiv.org/abs/1910.06663

[30] D. Pena, A. Forembski, X. Xu, and D. Moloney, "Benchmarking of CNNs for Low-Cost", 2017.

[31] J. Yang et al., “Quantization Networks,” *arXiv [cs.CV],* 2019. [Online]. Available: http://arxiv.org/abs/1911.09464

[32] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “A survey of model compression and acceleration for deep neural networks,” *arXiv [cs.LG],* 2017. [Online]. Available: http://arxiv.org/abs/1710.09282

[33] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A survey of quantization methods for efficient Neural Network inference,” *arXiv [cs.CV]*, 2021. Accessed: May 10, 2022. [Online]. Available: https://www.arxiv-vanity.com/papers/2103.13630/

[34] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” *arXiv [cs.CV],* 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[35] Q. Qin et al., “To compress, or not to compress: Characterizing deep learning model compression for embedded inference,” 2018.

[36] S. Pokhrel, “Model Compression: needs and importance,” *Towards Data Science,* 2020. https://towardsdatascience.com/model-compression-needs-and-importance-6e5913996e1 (accessed May 12, 2022).

[37] “Quantization Recipe — PyTorch Tutorials 1.11.0+cu102 documentation,” *Pytorch.org*. https://pytorch.org/tutorials/recipes/quantization.html (accessed May 13, 2022).

[38] “Quantization for rapid deployment of Deep Neural Networks,” 2019. [Online]. Available: https://openreview.net/pdf?id=HkzZBi0cFQ

[39] W. Thompson, “Computer Vision,” *SAS,* 2021. https://www.sas.com/el_gr/insights/analytics/computer-vision.html (accessed Jan. 08, 2022).

[40] G. Boesch, “A Complete Guide to Image Classification in 2021,” *VISIO,* 2021. https://viso.ai/computer-vision/image-classification/ (accessed Jan. 08, 2022).

[41] S. Joel Franklin, “K-means clustering for image classification,” *Medium,* 2020. https://medium.com/@joel_34096/k-means-clustering-for-image-classification-a648f28bdc47 (accessed Jan. 08, 2022).

[42] M. Tan and Q. V. Le, “EfficientNet: Rethinking model scaling for convolutional Neural Networks*,” arXiv [cs.LG],* 2019. [Online]. Available: http://arxiv.org/abs/1905.11946

[43] A. Howard et al., “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *arXiv [cs.CV],* 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[44] A. Howard et al., “Searching for MobileNetV3,” *arXiv [cs.CV],* 2019. [Online]. Available: http://arxiv.org/abs/1905.02244

[45] M. Wang, B. Liu, and H. Foroosh, “Design of efficient convolutional layers using single intra-channel convolution, topological subdivisioning and spatial ‘bottleneck’ structure,” *arXiv [cs.CV],* 2016. [Online]. Available: http://arxiv.org/abs/1608.04337

[46] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted residuals and linear bottlenecks,” *arXiv [cs.CV],* 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[47] Nvidia GitHub Repository. Accessed: Apr. 05, 2022. [Online]. Available: https://github.com/NVIDIA/libnvidia-container

[48] “Nvidia Container Toolkit documentation,” Nvidia. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html (accessed Apr. 05, 2022).

[49] "NVIDIA Driver Installation Quickstart Guide", Accessed: Apr. 22, 2022. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/tesla-installation-notes/index.html

[50] Triton Inference Server. Accessed: May 02, 2022. [Online]. Available: https://github.com/triton-inference-server/server

[51] V. Bezgachev, "TensorFlow Serving client. Make it slimmer and faster!," *Towards Data Science*, 2018. https://towardsdatascience.com/tensorflow-serving-client-make-it-slimmer-and-faster-b3e5f71208fb (accessed Apr. 25, 2022).

[52] F. S. Zuppichini, TorchServe Handler at GitHub. Accessed: Apr. 10, 2022. [Online]. Available: https://github.com/FrancescoSaverioZuppichini/torchserve-tryout

[54] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in 2018 *IEEE International Conference on Cloud Engineering (IC2E*), 2018, pp. 257–262.

[55] M. Nielsen, "CS231n Convolutional Neural Networks for Visual Recognition," *Github*. https://cs231n.github.io/neural-networks-1 (accessed May 16, 2021).

[56] T. Flach et al., "Reducing web latency: The virtue of gentle aggression," 2013, Accessed: Winter 06, 2021. [Online]. Available: https://research.google/pubs/pub41217/

[57] D. Godwin, "NVIDIA Triton inference server boosts deep learning inference," *NVIDIA Technical Blog,* 2018. https://developer.nvidia.com/blog/nvidia-serves-deep-learning-inference/ (accessed 06, 2021).

[58] D. Singal and M. Jhuria, "Deploying models from TensorFlow model zoo using NVIDIA DeepStream and NVIDIA Triton inference server," *NVIDIA Technical Blog*, 2020. https://developer.nvidia.com/blog/deploying-models-from-TensorFlow-model-zoo-using-deepstream-and-triton-inference-server/ (accessed Jun. 29, 2021).

[59] "NVIDIA Triton Inference Server," *NVIDIA Technical Blog*, 2020. https://developer.nvidia.com/nvidia-triton-inference-server (accessed Jun. 29, 2021).

[60] S. Chandrasekaran and M. Salehi, "Simplifying AI inference in production with NVIDIA Triton," *NVIDIA Technical Blog*, 2021. https://developer.nvidia.com/blog/simplifying-ai-inference-in-production-with-triton/ (accessed Jun. 29, 2021).

[61] D. Mwiti, "How to serve a model with TensorFlow," 2021. https://cnvrg.io/how-to-serve-a-model-with-TensorFlow/ (accessed Jun. 30, 2021).

[62] "Building standard TensorFlow ModelServer," *TensorFlow,* 2021. https://www.TensorFlow.org/tfx/serving/serving_advanced (accessed Jun. 30, 2021).

[63] M. A. Gamboa, "Building a machine learning application using H2O.ai," *Medium*, 2019. https://medium.com/spikelab/building-a-machine-learning-application-using-h2o-ai-67ce3681df9c (accessed Jun. 30, 2021).

[64] "Best practices in machine learning infrastructure," *DataRobot AI Cloud,* 2019. https://algorithmia.com/blog/best-practices-in-machine-learning-infrastructure (accessed Jun. 30, 2021).

[65] T. Jain, "Basics of Image Classification Techniques in Machine Learning," *IQ*, 2021. https://iq.opengenus.org/basics-of-machine-learning-image-classification-techniques/ (accessed Jan. 09, 2022).

[66] M. Fisher, "Image Analysis," *Stanford.edu,* 2014. https://graphics.stanford.edu/~mdfisher/ImageSegmentation.html (accessed Jan. 09, 2022).

[67] O. Russakovsky et al., “ImageNet Large Scale Visual Recognition Challenge,” *arXiv [cs.CV]*, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575

[68] “Stanford dogs dataset.” Accessed: Mar. 02, 2022. [Online]. Available: http://vision.stanford.edu/aditya86/ImageNetDogs/main.html

[69] “Cats dataset.” Accessed: 03, 2022. [Online]. Available: https://www.kaggle.com/datasets/crawford/cat-dataset

[70] “Food dataset.” Accessed: 03, 2022. [Online]. Available: https://data.vision.ee.ethz.ch/cvl/datasets_extra/food-101/

## Appendices

### 1 Appendix. NVIDIA Triton Inference Server Infrastructure

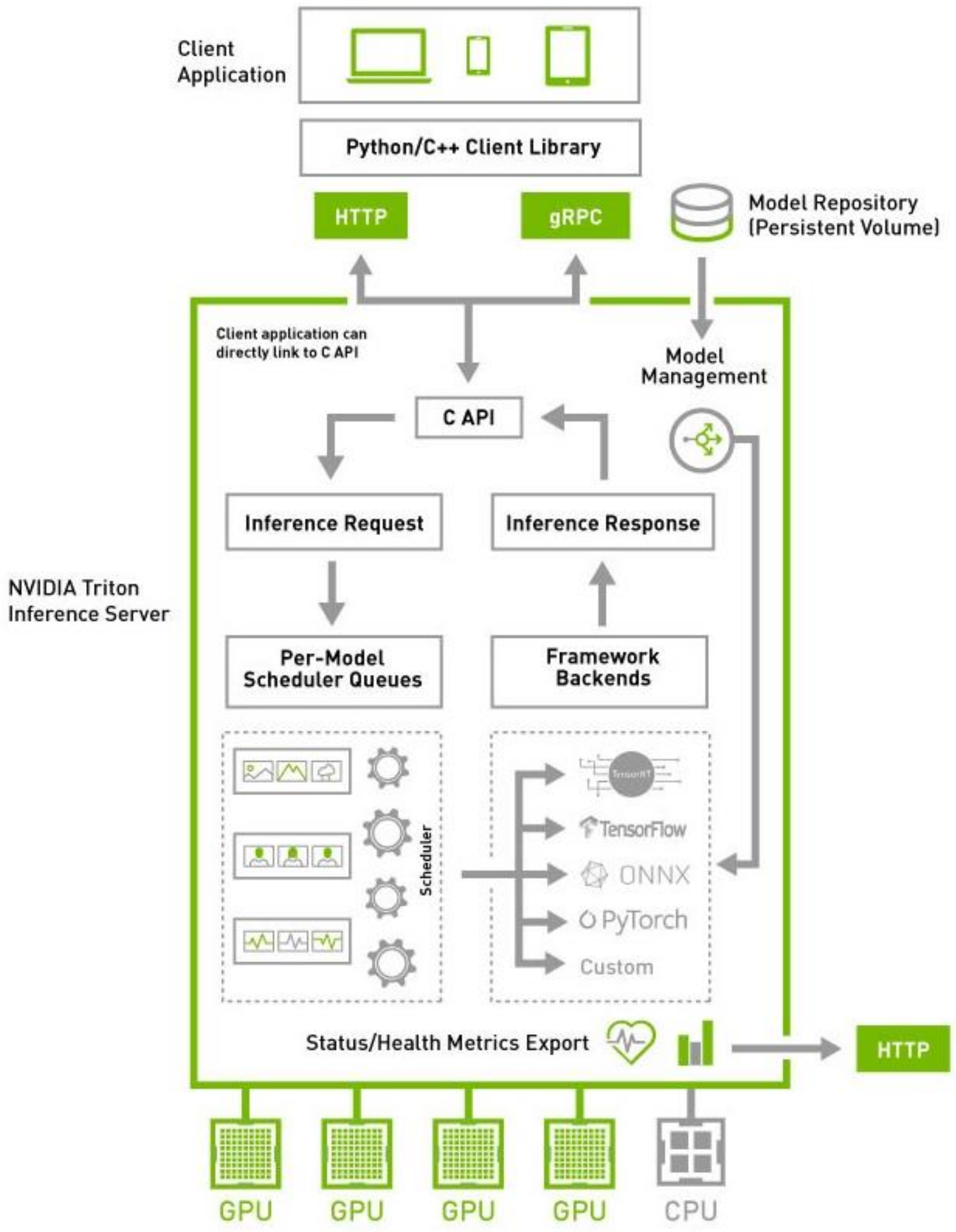

Client Application
Python/C++ Client Library
HTTP
gRPC
Model Repository (Persistent Volume)
Client application can directly link to C API
Model Management
C API
Inference Request
Inference Response
NVIDIA Triton Inference Server
Per-Model Scheduler Queues
Framework Backends
Scheduler
TensorRT
TensorFlow
ONNX
PyTorch
Custom
Status/Health Metrics Export
HTTP
GPU
GPU
GPU
GPU
CPU

**Figure 46.** Nvidia Triton Inference Server Architecture [51]

## 2 Appendix. NVIDIA Drivers Installation Guidelines

"The NVIDIA driver requires that the kernel headers and development packages for the running version of the kernel be installed at the time of the driver installation, as well whenever the driver is rebuilt" (Nvidia Data Center Documentation, 2022). The following commands should be run in terminal:

```
sudo apt-get install linux-headers-$(uname -r)
```

Next, we have to ensure CUDA network repository having the priority over the canonical repository:

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID | sed -e 's/\.//g')
```

```
wget https://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64/cuda-
$distribution.pin
```

```
sudo mv cuda-$distribution.pin /etc/apt/preferences.d/cuda-repository-pin-600
```

Above step of accessing CUDA repository may fail if incorrect OS version is installed as CUDA supports only Ubuntu 18.04 and Ubuntu 20.04 OS versions. This issue appeared during the experiment and new OS version needed to be installed, serving and benchmarking environment preparation actions repeated.

CUDA repository GPG key needs to be installed:

```
sudo apt-key adv --fetch-keys
https://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64/7fa2af80.pub
```

Setting up CUDA network repository:

```
echo "deb http://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64 /" |
sudo tee /etc/apt/sources.list.d/cuda.list
```

Finally, update APT repository and install drivers:

```
sudo apt-get update \
sudo apt-get -y install cuda-drivers
```

Post-installation steps are required to setup CUDA for Linux OS environments as automatic environment setup was moved only in Debian flow. The following actions needs to be manually performed if drivers installation was performed on Ubuntu OS:

```
export PATH=/usr/local/cuda-11.6/bin${PATH:+:${PATH}}
```

```
export LD_LIBRARY_PATH=/usr/local/cuda-11.6/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

There is also additional requirements to the NVIDIA POWER9 CUDA driver due to new features to function properly. This steps is also not handled by cuda-driver installation process. For fixing it, we have to trigger automatic NVIDIA Persistance Daemon start for POWER9 module.

```
sudo systemctl enable nvidia-persistenced
```

Official documentation can be found here: https://docs.nvidia.com/datacenter/tesla/tesla-installation-notes/index.html#ubuntu-lts

## 3 Appendix. NVIDIA Container Toolkit Installation

NVIDIA Container Toolkit is available only to Ubuntu 16.04, 18.04, and 20.04 versions. This toolkit provides possibility to build and run GPU accelerated Docker containers by providing a simple CLI utility to automatically configure Linux containers leveraging NVIDIA hardware.

First of all, we setup the package repository and GPG key:

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID) \
      && curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | \
      sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \
      && curl -s -L \
      https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-container.list | \
      sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-
keyring.gpg] https://#g' | \
      sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list
```

Next, nvidia-docker2 package and its dependencies needs to be isntalled:

```
sudo apt-get install -y nvidia-docker2
```

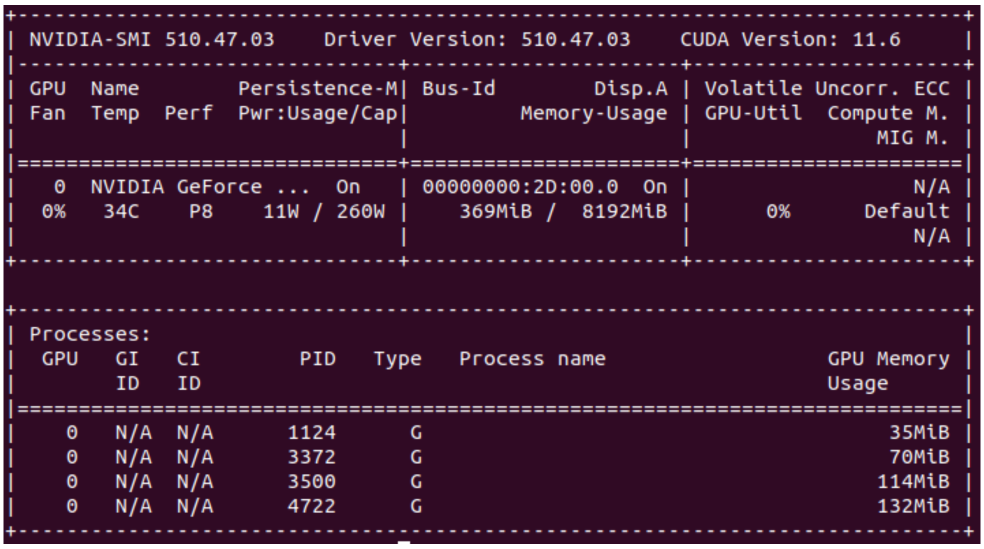
Docker daemon needs to be restarted to complete the isntallation:

```
sudo systemctl restart docker
```

A working setup can be tested by running base CUDA container:

```
sudo docker run --rm --gpus all nvidia/cuda:11.0-base nvidia-smi
```

Above command will return in console information about NVIDIA GPU and drivers:

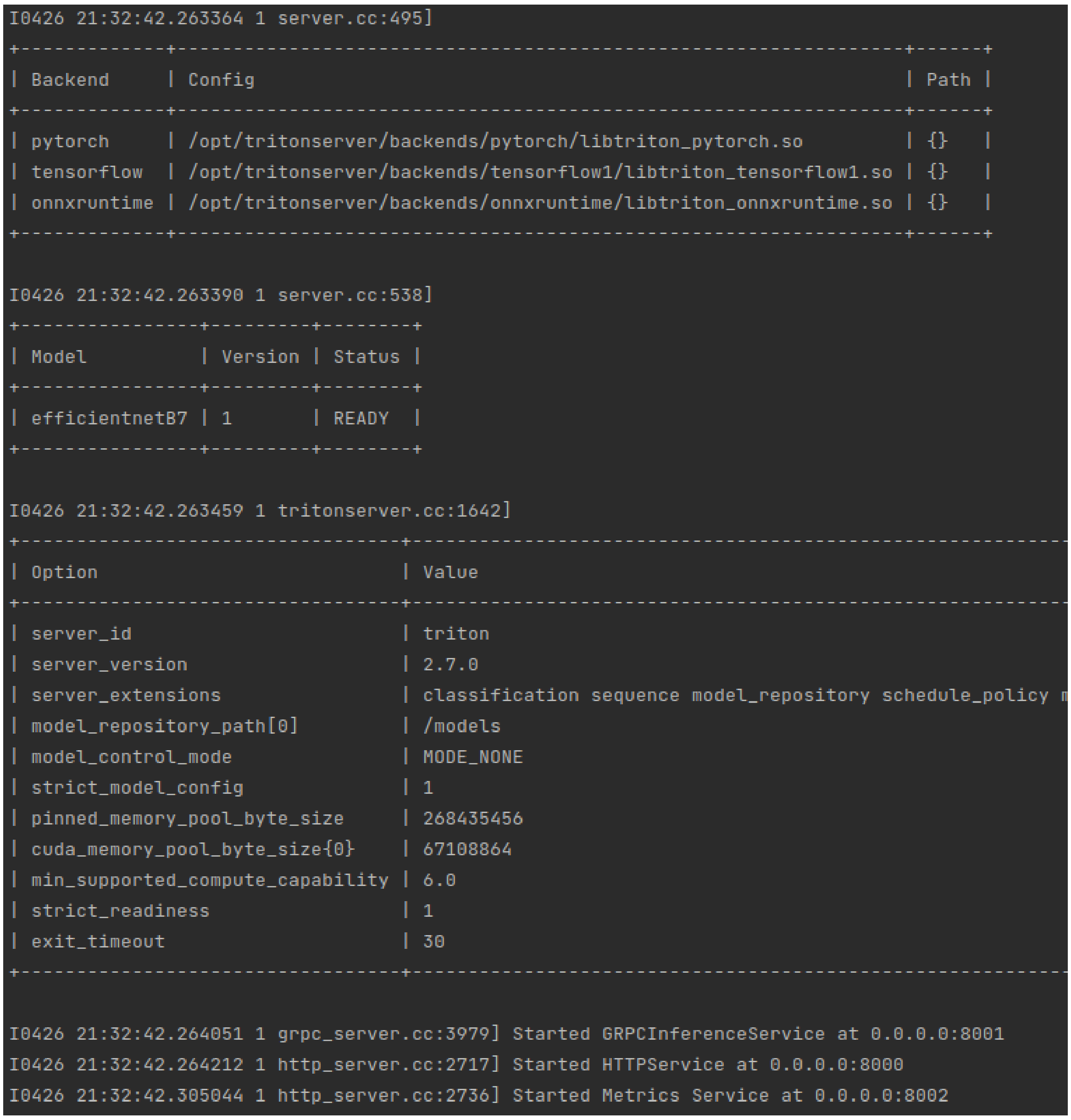


```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:2D:00.0  On |                  N/A |
|  0%   34C    P8    11W / 260W |    369MiB /  8192MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1124      G                                    35MiB |
|    0   N/A  N/A      3372      G                                    70MiB |
|    0   N/A  N/A      3500      G                                   114MiB |
|    0   N/A  N/A      4722      G                                   132MiB |
+-----------------------------------------------------------------------------+
```

**Figure 47.** nvidia-smi command console output

Official documentation can be found on NVIDIA Data Center website : https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#installing-on-ubuntu-and-debian

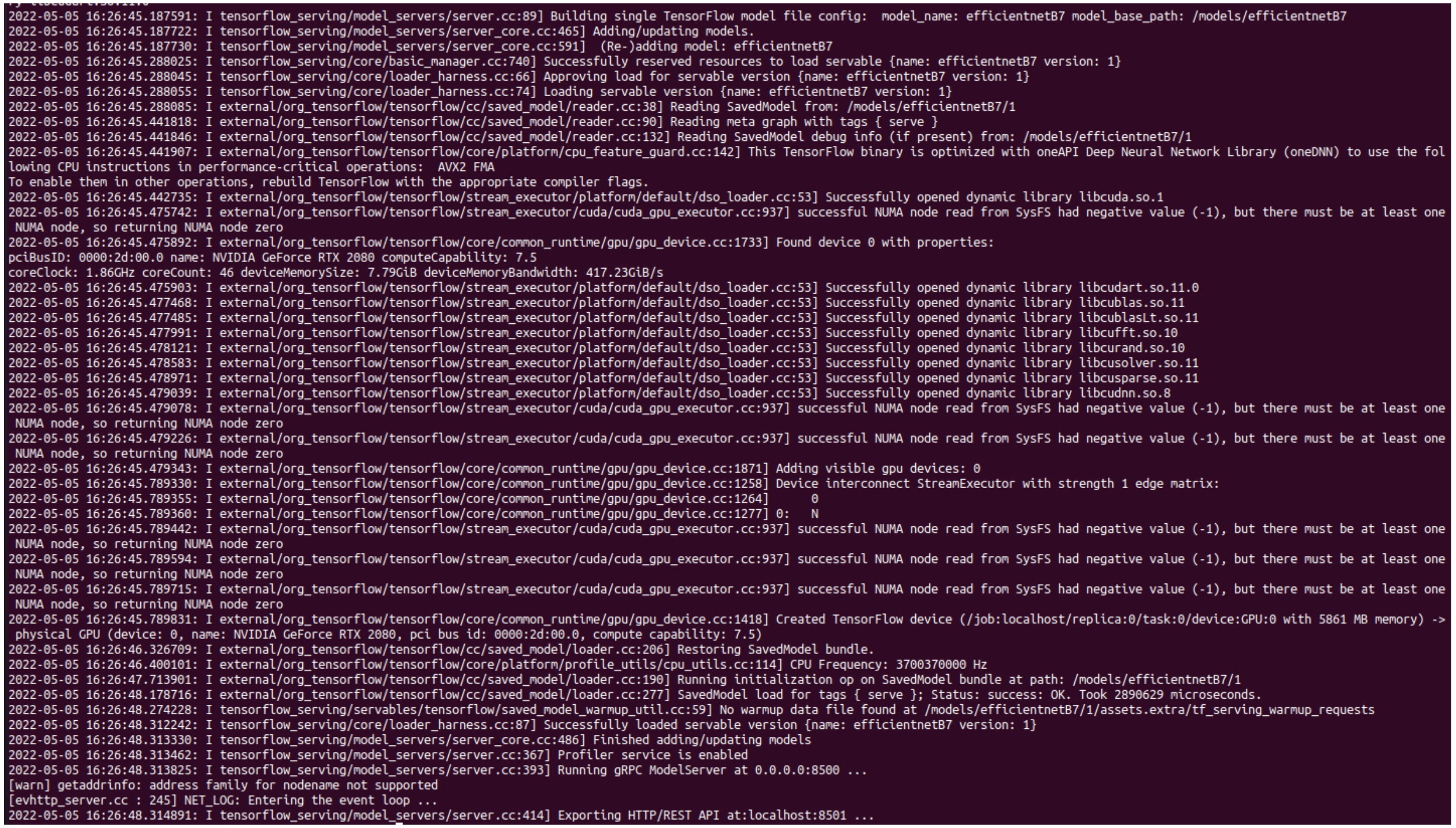
## 4 Appendix. Successful Model serving with Triton Inference Server Console Output



```
I0426 21:32:42.263364 1 server.cc:495]
+-------------+-----------------------------------------------------------------+------+
| Backend     | Config                                                          | Path |
+-------------+-----------------------------------------------------------------+------+
| pytorch     | /opt/tritonserver/backends/pytorch/libtriton_pytorch.so         | {}   |
| tensorflow  | /opt/tritonserver/backends/tensorflow1/libtriton_tensorflow1.so | {}   |
| onnxruntime | /opt/tritonserver/backends/onnxruntime/libtriton_onnxruntime.so | {}   |
+-------------+-----------------------------------------------------------------+------+


I0426 21:32:42.263390 1 server.cc:538]
+----------------+---------+--------+
| Model          | Version | Status |
+----------------+---------+--------+
| efficientnetB7 | 1       | READY  |
+----------------+---------+--------+


I0426 21:32:42.263459 1 tritonserver.cc:1642]
+----------------------------------+----------------------------------------------------------
| Option                           | Value
+----------------------------------+----------------------------------------------------------
| server_id                        | triton
| server_version                   | 2.7.0
| server_extensions                | classification sequence model_repository schedule_policy m
| model_repository_path[0]         | /models
| model_control_mode               | MODE_NONE
| strict_model_config              | 1
| pinned_memory_pool_byte_size     | 268435456
| cuda_memory_pool_byte_size{0}    | 67108864
| min_supported_compute_capability | 6.0
| strict_readiness                 | 1
| exit_timeout                     | 30
+----------------------------------+----------------------------------------------------------


I0426 21:32:42.264051 1 grpc_server.cc:3979] Started GRPCInferenceService at 0.0.0.0:8001
I0426 21:32:42.264212 1 http_server.cc:2717] Started HTTPService at 0.0.0.0:8000
I0426 21:32:42.305044 1 http_server.cc:2736] Started Metrics Service at 0.0.0.0:8002
```

**Figure 48.** Successful Triton deployment console output

## 5 Appendix. Successful Model serving with TensorFlow Serving Console Output



```
2022-05-05 16:26:45.187591: I tensorflow_serving/model_servers/server.cc:89] Building single TensorFlow model file config:  model_name: efficientnetB7 model_base_path: /models/efficientnetB7
2022-05-05 16:26:45.187722: I tensorflow_serving/model_servers/server_core.cc:465] Adding/updating models.
2022-05-05 16:26:45.187730: I tensorflow_serving/model_servers/server_core.cc:591]  (Re-)adding model: efficientnetB7
2022-05-05 16:26:45.288025: I tensorflow_serving/core/basic_manager.cc:740] Successfully reserved resources to load servable {name: efficientnetB7 version: 1}
2022-05-05 16:26:45.288045: I tensorflow_serving/core/loader_harness.cc:66] Approving load for servable version {name: efficientnetB7 version: 1}
2022-05-05 16:26:45.288055: I tensorflow_serving/core/loader_harness.cc:74] Loading servable version {name: efficientnetB7 version: 1}
2022-05-05 16:26:45.288085: I external/org_tensorflow/tensorflow/cc/saved_model/reader.cc:38] Reading SavedModel from: /models/efficientnetB7/1
2022-05-05 16:26:45.441818: I external/org_tensorflow/tensorflow/cc/saved_model/reader.cc:90] Reading meta graph with tags { serve }
2022-05-05 16:26:45.441846: I external/org_tensorflow/tensorflow/cc/saved_model/reader.cc:132] Reading SavedModel debug info (if present) from: /models/efficientnetB7/1
2022-05-05 16:26:45.441907: I external/org_tensorflow/tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the fol
lowing CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-05-05 16:26:45.442735: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcuda.so.1
2022-05-05 16:26:45.475742: I external/org_tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one
 NUMA node, so returning NUMA node zero
2022-05-05 16:26:45.475892: I external/org_tensorflow/tensorflow/core/common_runtime/gpu/gpu_device.cc:1733] Found device 0 with properties:
pciBusID: 0000:2d:00.0 name: NVIDIA GeForce RTX 2080 computeCapability: 7.5
coreClock: 1.86GHz coreCount: 46 deviceMemorySize: 7.79GiB deviceMemoryBandwidth: 417.23GiB/s
2022-05-05 16:26:45.475903: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcudart.so.11.0
2022-05-05 16:26:45.477468: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcublas.so.11
2022-05-05 16:26:45.477485: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcublasLt.so.11
2022-05-05 16:26:45.477991: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcufft.so.10
2022-05-05 16:26:45.478121: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcurand.so.10
2022-05-05 16:26:45.478583: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcusolver.so.11
2022-05-05 16:26:45.478971: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcusparse.so.11
2022-05-05 16:26:45.479039: I external/org_tensorflow/tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcudnn.so.8
2022-05-05 16:26:45.479078: I external/org_tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one
 NUMA node, so returning NUMA node zero
2022-05-05 16:26:45.479226: I external/org_tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one
 NUMA node, so returning NUMA node zero
2022-05-05 16:26:45.479343: I external/org_tensorflow/tensorflow/core/common_runtime/gpu/gpu_device.cc:1871] Adding visible gpu devices: 0
2022-05-05 16:26:45.789330: I external/org_tensorflow/tensorflow/core/common_runtime/gpu/gpu_device.cc:1258] Device interconnect StreamExecutor with strength 1 edge matrix:
2022-05-05 16:26:45.789355: I external/org_tensorflow/tensorflow/core/common_runtime/gpu/gpu_device.cc:1264]      0
2022-05-05 16:26:45.789360: I external/org_tensorflow/tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 0:   N
2022-05-05 16:26:45.789442: I external/org_tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one
 NUMA node, so returning NUMA node zero
2022-05-05 16:26:45.789594: I external/org_tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one
 NUMA node, so returning NUMA node zero
2022-05-05 16:26:45.789715: I external/org_tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one
 NUMA node, so returning NUMA node zero
2022-05-05 16:26:45.789831: I external/org_tensorflow/tensorflow/core/common_runtime/gpu/gpu_device.cc:1418] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 5861 MB memory) ->
 physical GPU (device: 0, name: NVIDIA GeForce RTX 2080, pci bus id: 0000:2d:00.0, compute capability: 7.5)
2022-05-05 16:26:46.326709: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:206] Restoring SavedModel bundle.
2022-05-05 16:26:46.400101: I external/org_tensorflow/tensorflow/core/platform/profile_utils/cpu_utils.cc:114] CPU Frequency: 3700370000 Hz
2022-05-05 16:26:47.713901: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:190] Running initialization op on SavedModel bundle at path: /models/efficientnetB7/1
2022-05-05 16:26:48.178716: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:277] SavedModel load for tags { serve }; Status: success: OK. Took 2890629 microseconds.
2022-05-05 16:26:48.274228: I tensorflow_serving/servables/tensorflow/saved_model_warmup_util.cc:59] No warmup data file found at /models/efficientnetB7/1/assets.extra/tf_serving_warmup_requests
2022-05-05 16:26:48.312242: I tensorflow_serving/core/loader_harness.cc:87] Successfully loaded servable version {name: efficientnetB7 version: 1}
2022-05-05 16:26:48.313330: I tensorflow_serving/model_servers/server_core.cc:486] Finished adding/updating models
2022-05-05 16:26:48.313462: I tensorflow_serving/model_servers/server.cc:367] Profiler service is enabled
2022-05-05 16:26:48.313825: I tensorflow_serving/model_servers/server.cc:393] Running gRPC ModelServer at 0.0.0.0:8500 ...
[warn] getaddrinfo: address family for nodename not supported
[evhttp_server.cc : 245] NET_LOG: Entering the event loop ...
2022-05-05 16:26:48.314891: I tensorflow_serving/model_servers/server.cc:414] Exporting HTTP/REST API at:localhost:8501 ...
```

**Figure 49.** Successful TensorFlow deployment console output

## 6 Appendix. TorchServe Architecture

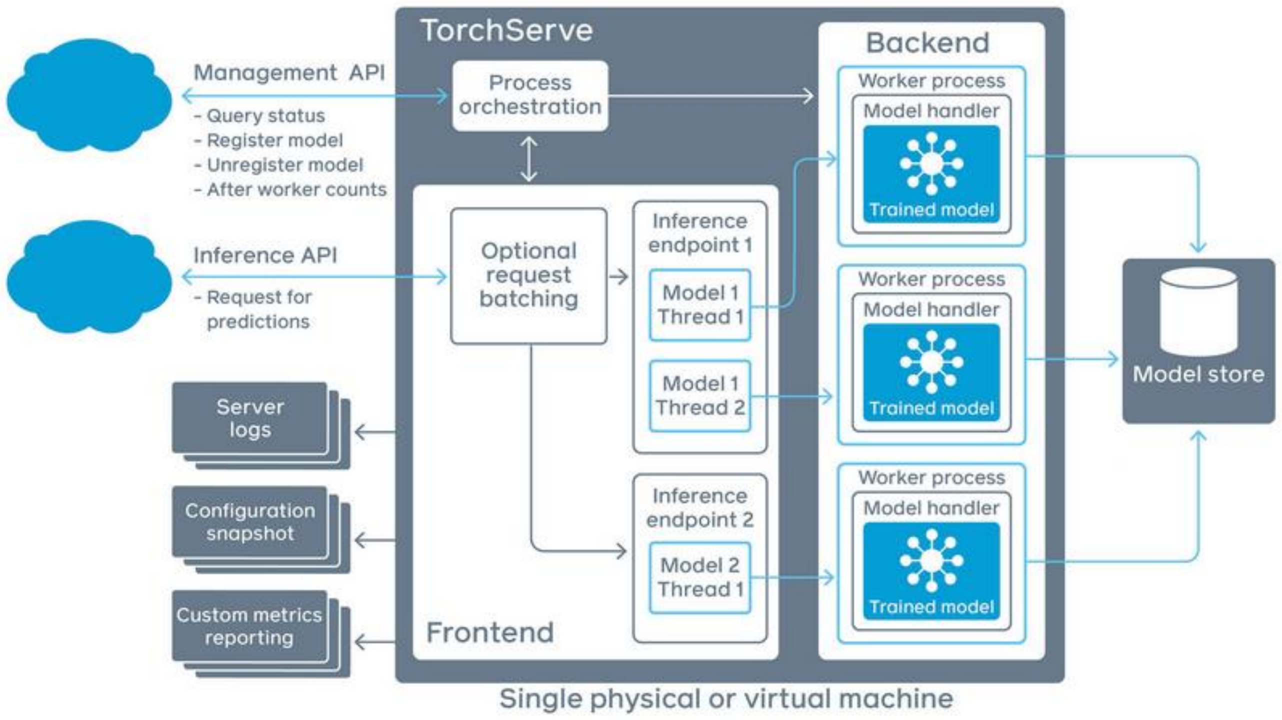
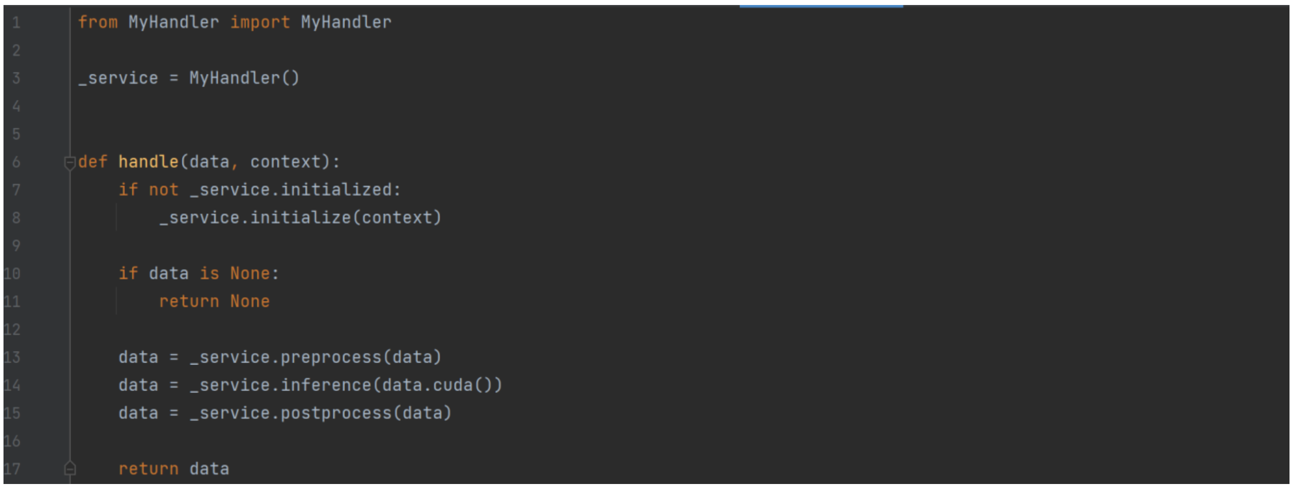


**Figure 50.** TorchServe architecture

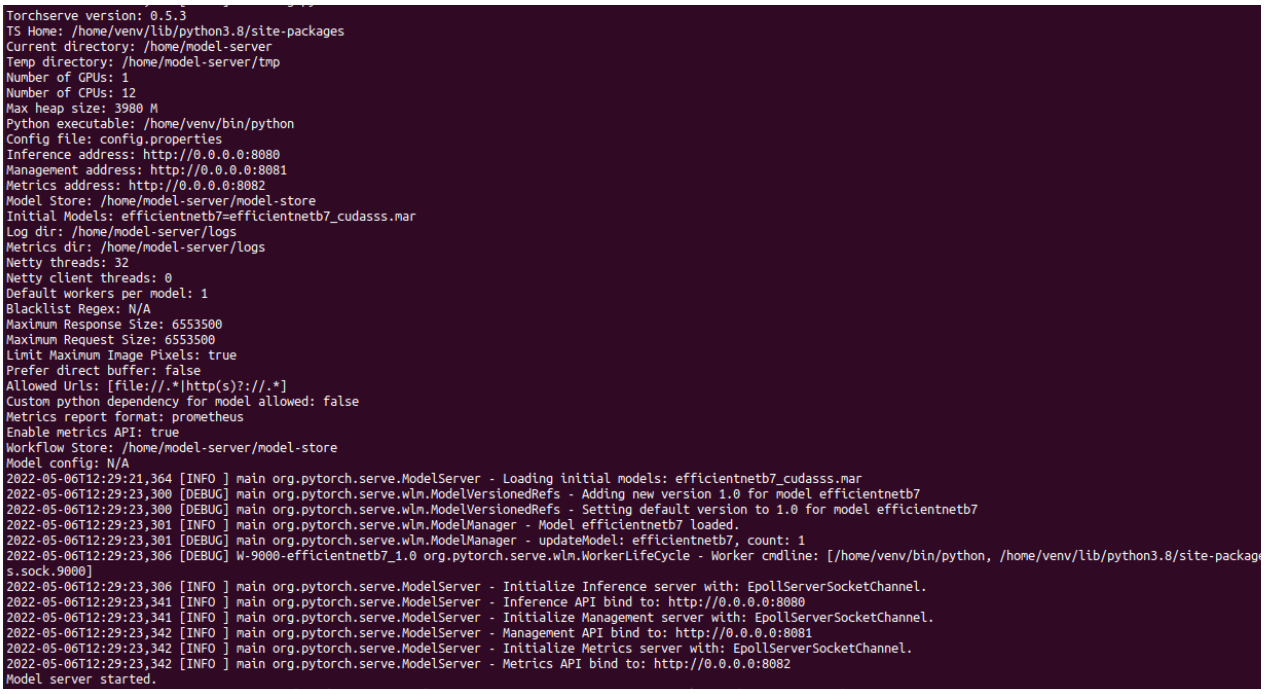
## 7 Appendix. TorchServe Model Handler



```python
from MyHandler import MyHandler

_service = MyHandler()


def handle(data, context):
    if not _service.initialized:
        _service.initialize(context)

    if data is None:
        return None

    data = _service.preprocess(data)
    data = _service.inference(data.cuda())
    data = _service.postprocess(data)

    return data
```

**Figure 51.** my_handler.py file used for model archive generation

# 8 Appendix. Successful Model serving with TorchServe Console Output



```
Torchserve version: 0.5.3
TS Home: /home/venv/lib/python3.8/site-packages
Current directory: /home/model-server
Temp directory: /home/model-server/tmp
Number of GPUs: 1
Number of CPUs: 12
Max heap size: 3980 M
Python executable: /home/venv/bin/python
Config file: config.properties
Inference address: http://0.0.0.0:8080
Management address: http://0.0.0.0:8081
Metrics address: http://0.0.0.0:8082
Model Store: /home/model-server/model-store
Initial Models: efficientnetb7=efficientnetb7_cudasss.mar
Log dir: /home/model-server/logs
Metrics dir: /home/model-server/logs
Netty threads: 32
Netty client threads: 0
Default workers per model: 1
Blacklist Regex: N/A
Maximum Response Size: 6553500
Maximum Request Size: 6553500
Limit Maximum Image Pixels: true
Prefer direct buffer: false
Allowed Urls: [file://.*|http(s)?://.*]
Custom python dependency for model allowed: false
Metrics report format: prometheus
Enable metrics API: true
Workflow Store: /home/model-server/model-store
Model config: N/A
2022-05-06T12:29:21,364 [INFO ] main org.pytorch.serve.ModelServer - Loading initial models: efficientnetb7_cudasss.mar
2022-05-06T12:29:23,300 [DEBUG] main org.pytorch.serve.wlm.ModelVersionedRefs - Adding new version 1.0 for model efficientnetb7
2022-05-06T12:29:23,300 [DEBUG] main org.pytorch.serve.wlm.ModelVersionedRefs - Setting default version to 1.0 for model efficientnetb7
2022-05-06T12:29:23,301 [INFO ] main org.pytorch.serve.wlm.ModelManager - Model efficientnetb7 loaded.
2022-05-06T12:29:23,301 [DEBUG] main org.pytorch.serve.wlm.ModelManager - updateModel: efficientnetb7, count: 1
2022-05-06T12:29:23,306 [DEBUG] W-9000-efficientnetb7_1.0 org.pytorch.serve.wlm.WorkerLifeCycle - Worker cmdline: [/home/venv/bin/python, /home/venv/lib/python3.8/site-package
s.sock.9000]
2022-05-06T12:29:23,306 [INFO ] main org.pytorch.serve.ModelServer - Initialize Inference server with: EpollServerSocketChannel.
2022-05-06T12:29:23,341 [INFO ] main org.pytorch.serve.ModelServer - Inference API bind to: http://0.0.0.0:8080
2022-05-06T12:29:23,341 [INFO ] main org.pytorch.serve.ModelServer - Initialize Management server with: EpollServerSocketChannel.
2022-05-06T12:29:23,342 [INFO ] main org.pytorch.serve.ModelServer - Management API bind to: http://0.0.0.0:8081
2022-05-06T12:29:23,342 [INFO ] main org.pytorch.serve.ModelServer - Initialize Metrics server with: EpollServerSocketChannel.
2022-05-06T12:29:23,342 [INFO ] main org.pytorch.serve.ModelServer - Metrics API bind to: http://0.0.0.0:8082
Model server started.
```

**Figure 52.** Successful TorchServe deployment console output

## 9 Appendix. 1st Scenario Experiments Results Graphs

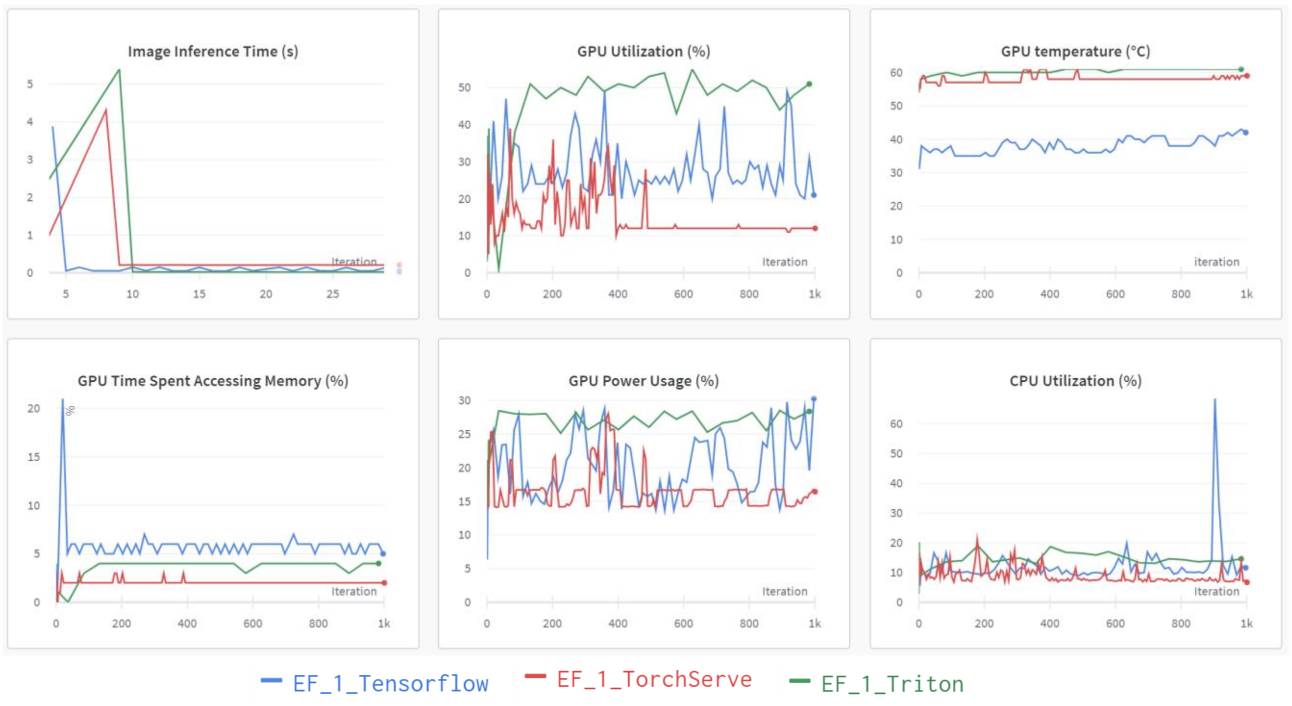


**Figure 53.** Scenario No. 1 EfficientNet-B7 model experiment (224x224px images) results

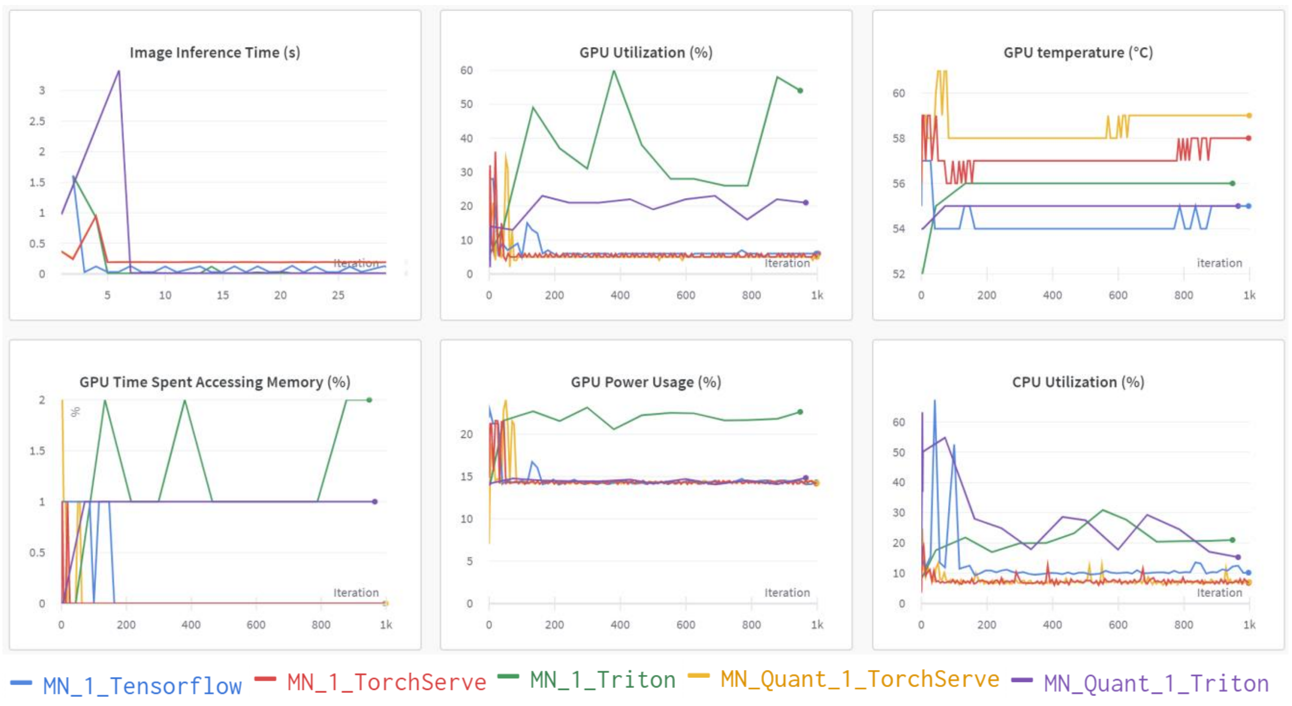


**Figure 54.** Scenario No. 1 MobileNetV3-large and MobileNetV3-large quantized models experiment results

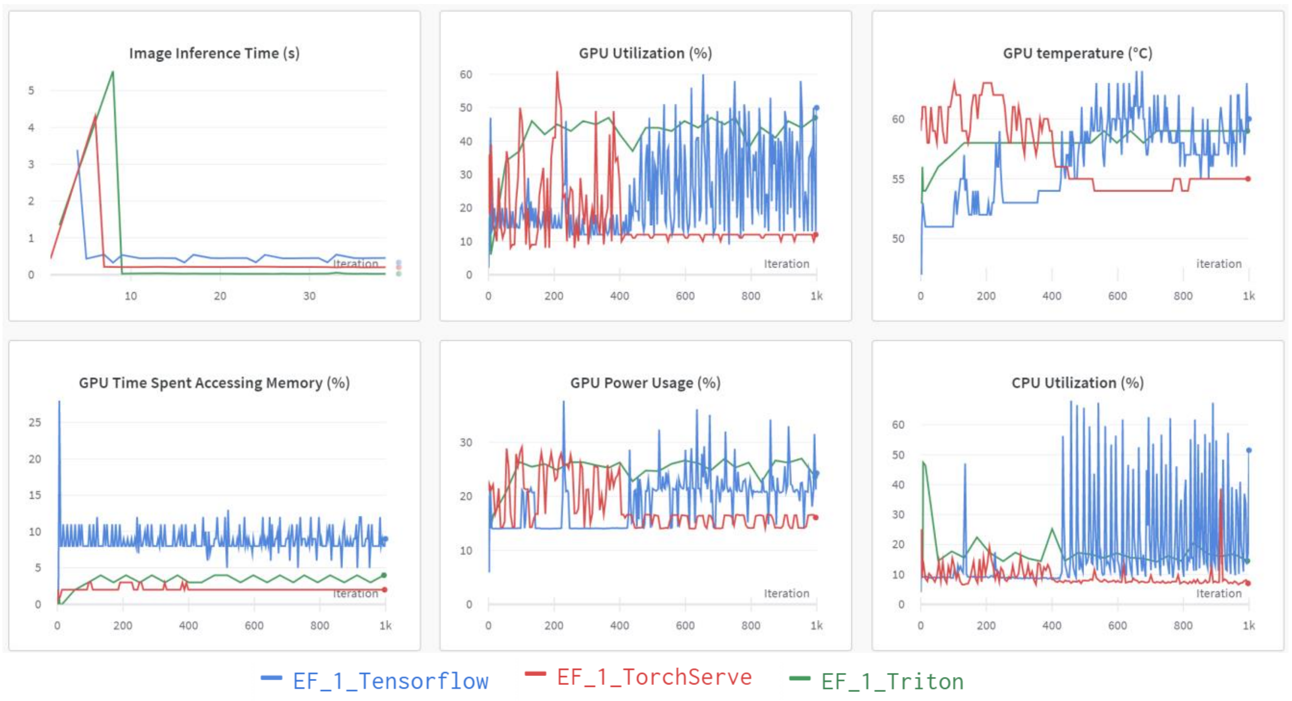


**Figure 55**. Scenario No. 1 EfficientNet-B7 model experiment (600x600px images) results

## 10 Appendix. 2nd Scenario Experiments Results Graphs

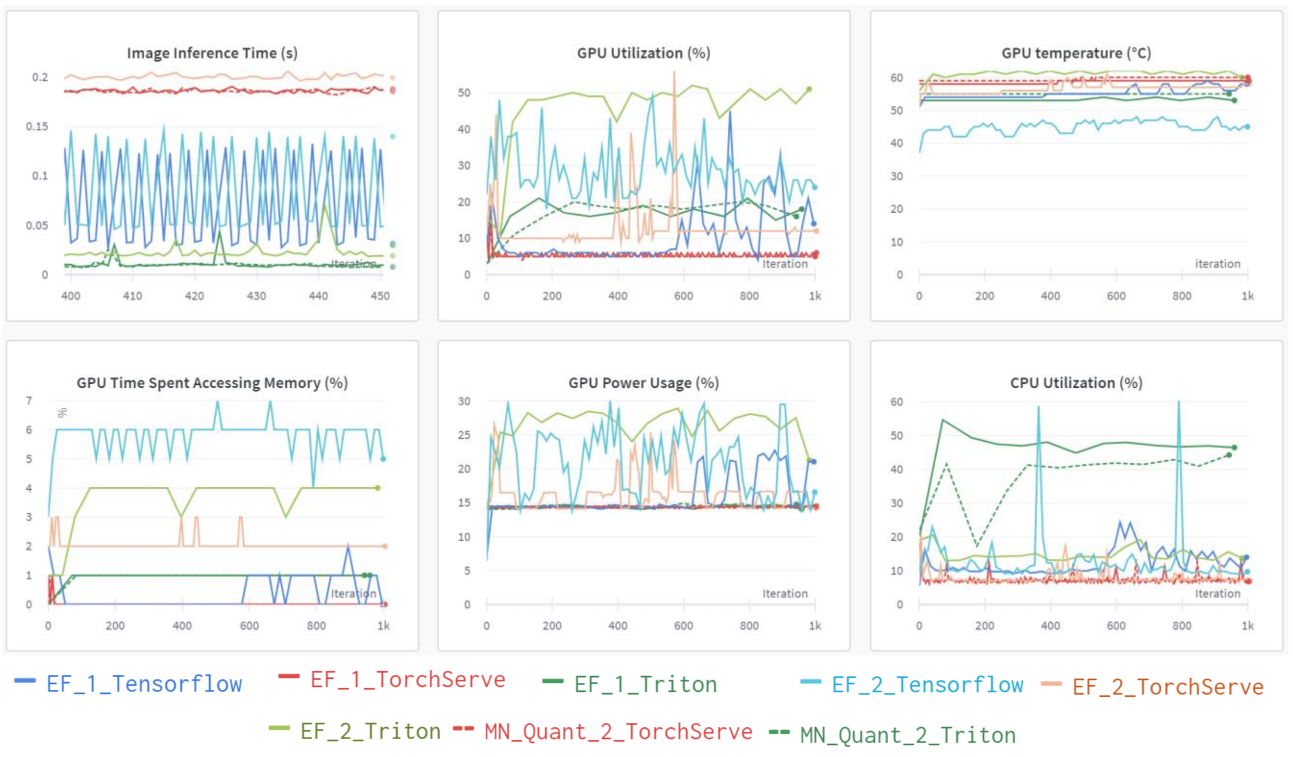


**Figure 56.** Scenario No. 2 experiment results

## 11 Appendix. 3rd Scenario Experiments Results Graphs

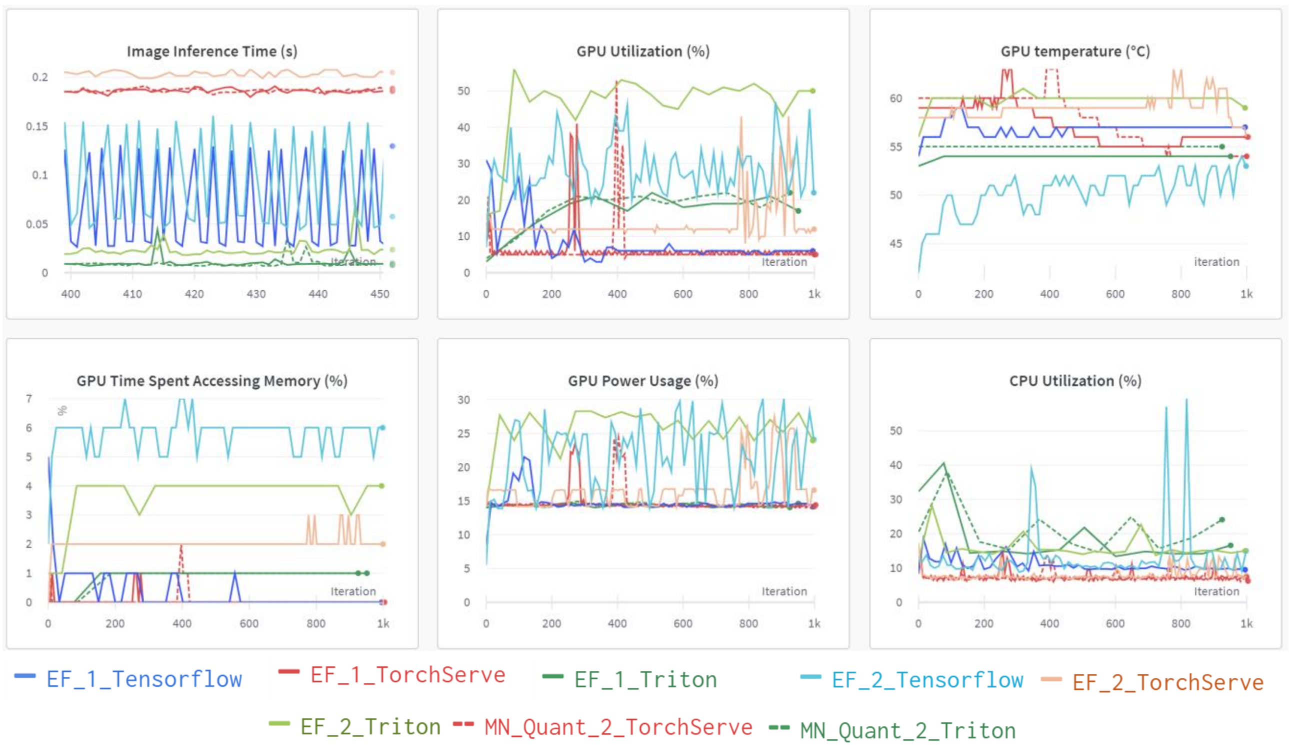


**Figure 57.** Scenario No. 3 experiment results

## 12 Appendix. 4th Scenario Experiments Results Graphs

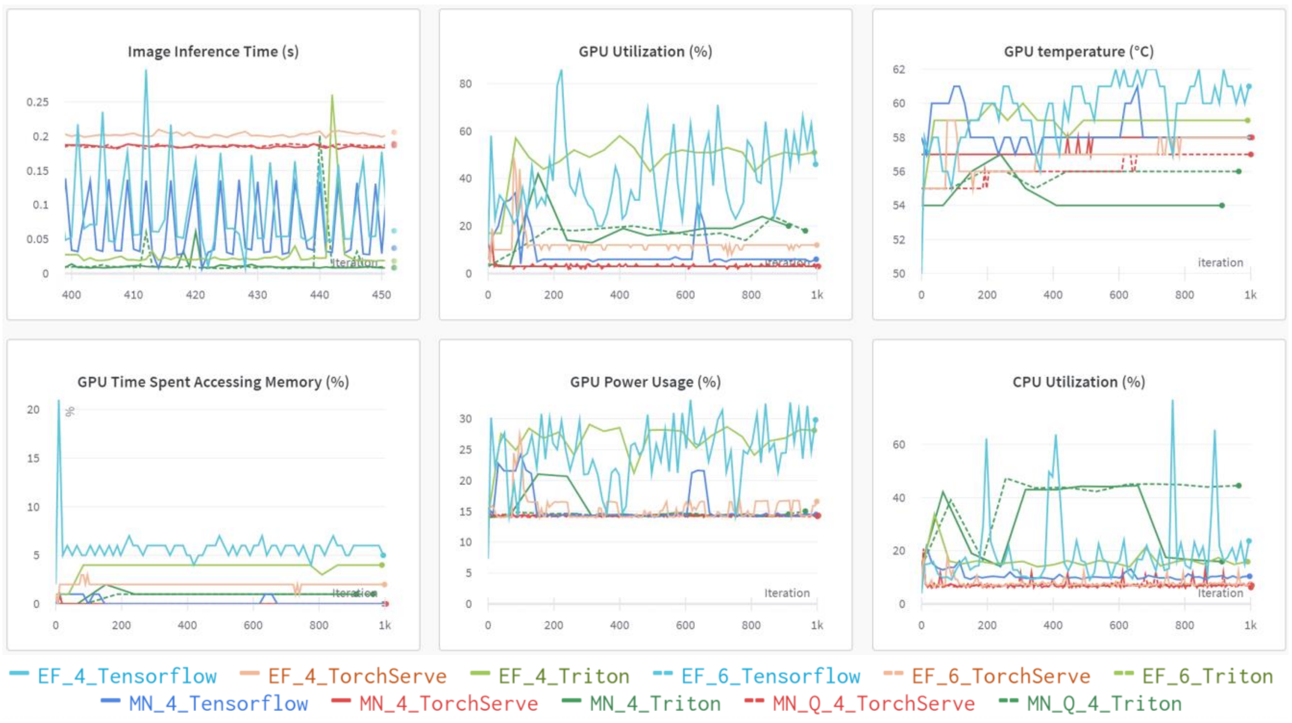


**Figure 58.** Scenario No. 4 experiment results

## 13 Appendix. 2nd-3rd Scenarios Experiments Results Graphs for 600px Images

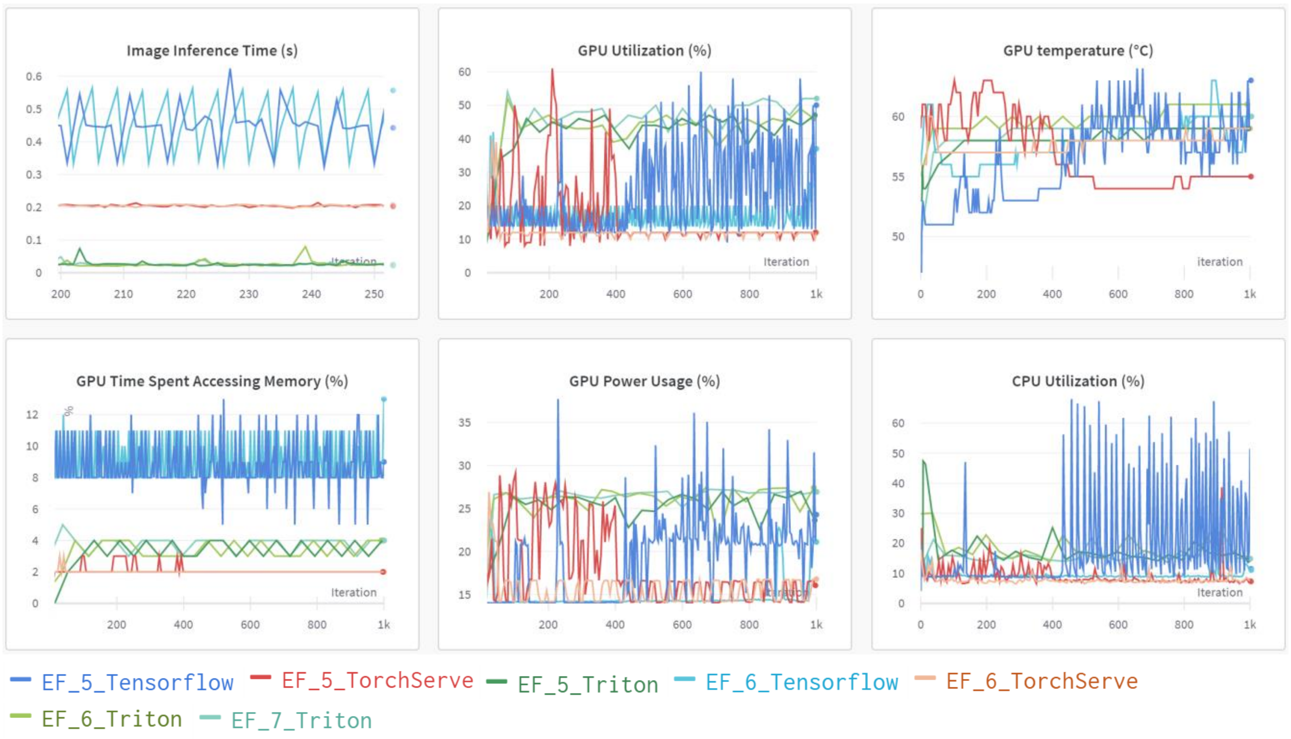


**Figure 59.** Scenario No. 2-3 experiment results for EfficientNet-B7, with 600 x 600 px dimensions images