BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATED TESTING AND EVALUATION OF DEEP NEURAL NETWORK MODELS ON EMBEDDED PLATFORMS

AUTOMATIZOVANÉ TESTOVÁNÍ A VYHODNOCOVÁNÍ MODELŮ HLUBOKÝCH NEURONOVÝCH SÍTÍ NA VESTAVĚNÝCH PLATFORMÁCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **JAKUB KASEM**
AUTOR PRÁCE

SUPERVISOR **Ing. RADEK KOČÍ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

164232

Institut: Department of Intelligent Systems (DITS)
Student: **Kasem Jakub, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Machine Learning
Title: **Automated testing and evaluation of Deep Neural Network models on embedded platforms**
Category: Artificial Intelligence
Academic year: 2024/25

Assignment:

1. Study popular DNN model repositories and ways how to retrieve models from them.
2. Learn about the accuracy metrics used to evaluate DNN models in various AI tasks such as classification, object detection, and segmentation.
3. Design the system for automated testing and evaluation of models on embedded platforms, focusing on accuracy and latency. The system must also track and detect regression from past results.
4. Implement the system.
5. Validate the system on provided Linux-based embedded platforms using a collection of DNN models from various AI tasks.
6. Discuss possible extensions of the system, especially its deployment in the CI infrastructure for embedded platforms.

Literature:

- Lee, Y.-S. Analysis of AI Model Hub, International Journal of Advanced Culture Technology, Vol. 11, No.4 442-448, 2023.
- Bradley J. Erickson and Felipe Kitamura. Magician's Corner: 9. Performance Metrics for Machine Learning Models, Radiology: Artificial Intelligence 2021; 3(3).
- Željko Đ. Vujović. Classification Model Evaluation Metrics, International Journal of Advanced Computer Science and Applications, Volume 12, Issue 6, 2021

Requirements for the semestral defence:
Points 1 - 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Kočí Radek, Ing., Ph.D.**
Head of Department: Kočí Radek, Ing., Ph.D.
Beginning of work: 1.11.2024
Submission deadline: 21.5.2025
Approval date: 31.10.2024

Abstract

This master's thesis focuses on automated testing of deep neural network models on embedded platforms. The literature review explores popular artificial intelligence model repositories and tools to retrieve models from such repositories. The theoretical foundation of the thesis also includes a review of accuracy metrics used for evaluating deep neural network models in classification, object detection, and segmentation tasks. The thesis further describes the design and the implementation of a framework that automates testing and evaluation of deep neural network models on Linux-based embedded architectures. The proposed solution focuses on measuring model accuracy and latency, with a feature that enables detection of performance regressions over time. The solution was validated on embedded platforms manufactured by NXP Semiconductors using deep neural network models across different computer vision tasks. The thesis concludes with a reflection on lessons learned during implementation and outlines opportunities for further development and expansion of the system.

Abstrakt

Táto magisterská práca sa zameriava na automatizované testovanie modelov hlbokých neurónových sietí na vstavaných platformách. Teoretická časť práce skúma populárne úložiská modelov umelej inteligencie a nástroje na získavanie modelov z takýchto úložísk. Teoretický základ práce taktiež zahŕňa prehľad metrík používaných na hodnotenie modelov hlbokých neurónových sietí v úlohách klasifikácie, detekcie objektov a segmentácie. Práca ďalej opisuje návrh a implementáciu systému, ktorý automatizuje testovanie a vyhodnocovanie modelov hlbokých neurónových sietí na vstavaných Linux architektúrach. Navrhované riešenie sa zameriava na meranie presnosti a odozvy modelu, s funkcionalitou, ktorá umožňuje detekciu regresie výkonu. Riešenie bolo overené na vstavaných platformách vyrábaných spoločnosťou NXP Semiconductors s použitím modelov hlbokých neurónových sietí v rôznych úlohách počítačového videnia. Práca končí diskusiou o skúsenostiach získaných počas implementácie a načrtáva možnosti ďalšieho vývoja a rozšírenia systému.

Keywords

Automated testing, deep neural networks, embedded systems, neural network evaluation, performance regression detection, neural network accuracy, neural network latency, computer vision, classification, object detection, semantic segmentation, TensorFlow Lite, TFLite, model hub, Kaggle, Hugging Face, FastAPI.

Klíčová slova

Automatizované testovanie, hlboké neurónové siete, vstavané systémy, vyhodnotenie neurónovej siete, presnosť neurónovej siete, odozva neurónovej siete, počítačové videnie, klasifikácia, detekcia objektov, sémantická segmentácia, TensorFlow Lite, TFLite, úložisko AI modelov, Kaggle, Hugging Face, FastAPI.

Reference

KASEM, Jakub. *Automated testing and evaluation of deep neural network models on embedded platforms.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Kočí, Ph.D.

Automated testing and evaluation of deep neural network models on embedded platforms

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Radek Kočí, PhD. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jakub Kasem
May 19, 2025

Acknowledgements

I want to express my sincere gratitude to my supervisor, Ing. Radek Kočí, Ph.D., for their expert guidance, thoughtful feedback, and continuous support throughout this thesis.

I am also grateful to Brno University of Technology and NXP Semiconductors for providing a productive and encouraging research environment.

A special thank you goes to my colleagues at NXP — Jakub Salamon, Lukáš Sztefek, and Robert Kalmár — for their technical support and valuable insights.

Lastly, I thank my family and friends for their constant support and patience during my studies – and my dog, for always easing my mind.

Contents

1 Introduction **6**

2 Deep neural networks **7**
2.1 Structure of a neural network 7
2.2 Pattern learning and generalization 9
2.3 Computer vision 10

3 Model repositories **14**
3.1 Defining model repositories 14
3.2 Kaggle 15
3.3 Hugging Face 17

4 Evaluation metrics for deep neural network models in computer vision tasks **21**
4.1 Classification 21
4.2 Object detection 24
4.3 Semantic segmentation 26
4.4 Inference latency 28

5 System for automatic testing and evaluation of neural networks on embedded platforms **29**
5.1 System requirements 29
5.2 Inputs and outputs of the proposed system 30
5.3 System architecture 33

6 Implementation of the system **39**
6.1 Model Tester 39
6.2 Device API 43
6.3 Report API 45

7 Evaluation of the implementation **48**
7.1 Experimental environment 48
7.2 Executing the experiment 49
7.3 Implementation retrospective 52

8 Conclusion **55**

Bibliography **58**

A Codebase folder structure **61**

B User script specification **62**

List of Figures

2.1 Mathematical model of neuron [19] 8
2.2 A neural network. The chain structure of the network: one neuron in the input layer is connected to three neurons in the hidden layer; neurons in the hidden layer are fully connected to the two neurons in the output layer. . . 9
2.3 2D convolution operation where a 2×2 kernel slides over a 3×3 input matrix, generating a 2×2 output matrix [1] 11
2.4 Examples of the max and average pooling layers [26] 11

5.1 Proposed testing results report. 32
5.2 Proposed regression tracking interface for latency comparison 33
5.3 Proposed regression tracking interface for accuracy comparison 33
5.4 High-level modular overview of the system architecture 34
5.5 Orchestrator script execution flow 35
5.6 Sequence diagram depicting device identification interaction between the system's modules 36
5.7 Model deployment process sequence diagram 36
5.8 Entity relationship diagram of the database storing the assessments made by the system. The diagram captures metadata about each evaluation run, including the associated model, dataset, device, and metrics. 37

6.1 Class diagram of the Model Tester module illustrating the main classes involved in the evaluation workflow 40

7.1 The framework's command line output during the validation experiment. . 49
7.2 Top of the summary report for the validation experiment 51
7.3 The accuracy comparison interface displaying results for the YOLOv8 model and the Coco2017 dataset. 52
7.4 The latency comparison interface displaying results for the DeepLabv3 model. 52

List of Tables

4.1 Categorization of predictions with respect to c_0 class. Note: Case's predicted class is c_0 if $P(c_0) \geq T$. Case's predicted class is c_1 if $P(c_0) < T$. Ground truth 0 stands for c_0; 1 stands for c_1. 22
4.2 Confusion matrix for a binary classification problem with respect to c_0 class. Note: TP = true positive, TN = true negative, FP = false positive, FN = false negative. 22
4.3 Confusion matrix for a multiclass classification problem. Example categorization for class A: $TP = (A, A) = 200$, $FP = (B, A) + (C, A) + (C, D) = 200$, $FN = (A, B) + (A, C) + (A, D) = 0$, $TN = (B, B) + (B, C) + (B, D) + (C, B) + (C, C) + (C, D) + (D, B) + (D, C) + (D, D) = 60$. 23
4.4 Definitions and formulas for selected performance metrics used to evaluate binary classification models. Each metric is defined in terms of the confusion matrix components. Note: TP = true positive, TN = true negative, FP = false positive, FN = false negative. 24

7.1 Comparison of the latency metric values evaluated during the experiment. . 50

List of Listings

3.1 Model download via Kagglehub Python library 16
3.2 Dataset download via Kagglehub Python library 16
3.3 Model download using Kaggle CLI software 16
3.4 Dataset download using Kaggle CLI software 17
3.5 Model download by directly calling Kaggle API software 17
3.6 Dataset download by directly calling Kaggle API software 17
3.7 Hugging Face: Loading dataset using datasets Python library 18
3.8 Hugging Face: Loading model using transformers Python library 18
3.9 Hugging Face: Downloading specific file using huggingface_hub Python library 19
3.10 Hugging Face: Downloading a repository using huggingface_hub Python library 19
3.11 Hugging Face: Dataset download using Git 19
3.12 Hugging Face: Model download using Git 19
5.1 Example configuration file requesting a single evaluation run. The file specifies the device, model, dataset, and the run tuple referencing the device, model, dataset, and the required post-processing script. 31
6.1 The input schema of the Report API /run endpoint 46
6.2 Typed objects representing the database tables in the code base 46

Chapter 1

Introduction

The concept for this thesis came from *NXP Semiconductors' Machine Learning team*. One of NXP Semiconductors' activities is the development of embedded devices capable of neural network inference and custom Linux operating systems tailored to these devices. During deployment of neural networks on embedded devices, there is a need to optimize computational performance, memory requirements, and power efficiency without compromising model accuracy. As NXP continuously develops new processors and new versions of Linux for embedded devices, a rigorous quality assurance process is required. Even minor changes to system libraries or drivers can affect neural networks' performance or inference accuracy. This context raises a demand for a tool that would enable automated testing of neural networks and detect possible regression between different versions of embedded software platforms.

This thesis focuses on evaluating neural networks in computer vision tasks, with particular attention to metrics that assess the accuracy and latency of these models. The thesis presents the design, implementation, and evaluation of a framework that allows deployment of deep neural network models on embedded devices, measuring their latency and task-specific accuracy, and comparing the results between different versions of Linux for embedded systems, detecting possible regressions. The presented framework also allows download of models from public model repositories like *Hugging* Face or *Kaggle*, ensuring fast access to the state-of-the-art deep neural networks and eliminating the need to develop and store neural networks in-house.

The chapters of the thesis text are divided as follows. Chapter 2 explains deep neural networks and the tasks they are leveraged in computer vision. Chapter 3 discusses model repositories, defines these platforms, and explores the interfaces for downloading items from the Kaggle and Hugging Face repositories. Chapter 4 describes metrics for evaluating the accuracy and latency of deep neural network models in computer vision tasks. Chapter 5 defines the testing framework requirements, including its expected inputs and outputs, and presents the design of the proposed system architecture. Chapter 6 covers the implementation of the design from Chapter 5. Chapter 7 details the experiments conducted for system validation, and concludes with an implementation retrospective and discussion of potential directions for future development.

To enhance the clarity and correctness of the thesis text, the written content was refined using *Grammarly*[1] and its generative AI tool. These tools assisted in grammar correction, phrasing suggestions, and overall language quality improvement.

[1] Available at https://www.grammarly.com

Chapter 2

Deep neural networks

Neural networks introduce an alternative paradigm in computing – the solution to a problem is derived from a set of examples rather than by following a set of explicit instructions. The creation of neural networks is inspired by research on information processing mechanisms in the human brain. Therefore, neural networks can be defined as "massive parallel computational models that imitate the function of the human brain" [18]. This chapter explores the structure of neural networks, how they learn patterns and make predictions, and deep neural networks. Section 2.3 focuses on the deep neural network in the context of computer vision and computer vision tasks, which will be a subject in the later parts of this chapter. This chapter reviews literature [19, 18, 32, 21, 22, 28, 17, 7, 20, 33, 6].

2.1 Structure of a neural network

A neural network consists of processing units linked by connections. The processing units are called *neurons*, and they are analogous to the human brain. Every connection between neurons has an associated weight and the output of each neuron depends solely on locally available information [18].

The mathematical model of a neuron was introduced in 1943 by McCulloch and Pits [19]. This model of a neuron takes a input variable in the form of a vector and performs scalar product of the input value $X = (x_1, x_2, ..., x_n)$ and the corresponding weights $W = (w_1, w_2, ..., w_n)$ (weights are associated to the connection between scalar input values and neuron) to which a *bias* b is added (depicted in equation 2.1). Therefore, *the output of a neuron is dependent on the input of the neuron, the weights assigned to each input's connection and internally stored bias.* Alternatively, bias can be represented as an element of the weight vector W as w_0 with the corresponding scalar input $x_0 = 1$ in the input vector X (depicted in equation 2.2).

$$a = [\sum_{i=1}^{n} x_i w_i] + b \tag{2.1}$$

$$a = \sum_{i=0}^{n} x_i w_i \tag{2.2}$$

In order to obtain the output of a neuron, the *activation function* g is applied to the *total input* a of the neuron (depicted in equation 2.3 and Figure 2.1).

$$z = g(a) \tag{2.3}$$

Some possible activation functions are the Sigmoid function, the ReLU function, or the Softmax function.

- The sigmoid activation function σ is a nonlinear function that translates input from the $(-\infty, \infty)$ interval to the $< -1, 1 >$ interval. Sigmoid function may be used for classification problems with two classes (see Section 2.3). Sigmoid function is defined as
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
- The rectified linear unit (ReLU) activation function returns zero for values that are less or equal to zero, and linear for values greater than zero. ReLU is defined as
$$ReLU(x) = max(0, x)$$
- Softmax activation function returns the normalized probability distribution for a given vector. Softmax function may be used for classification problems with multiple classes (see Section 2.3). The softmax function is defined as
$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

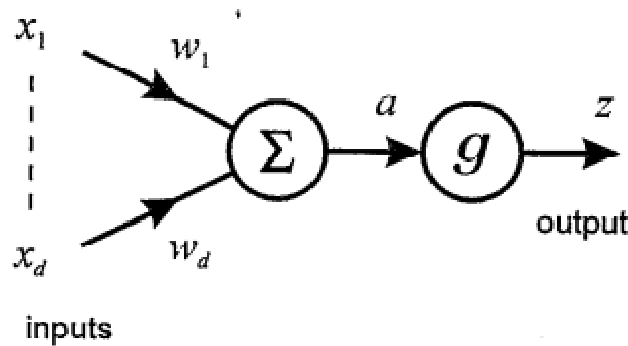


Figure 2.1: Mathematical model of neuron [19]

What makes a neural network a network is the chain structure that connects neurons. Let $f(x) = f_3(f_2(f_1(x)))$, where $f(x)$ is an *output of a neural network* for input x and the functions f_1, f_2, and f_3 are connected in a chain. The chain structure may be represented as multiple connected layers – in the chain structure $f(x)$, the f_1 is the first layer, f_2 is the second layer, and f_3 is the third layer. The first layer of the neural network is called the *input layer*, and the last layer of the neural network is called the *output layer*. The layers between the output layer and input layer are called *hidden layers*. Each layer of a neural network may contain multiple neurons (often referred to as *nodes* in a chain structure).

Fully-connected layer The layer is *fully-connected* if each neuron in the layer is connected to each neuron in the previous layer.

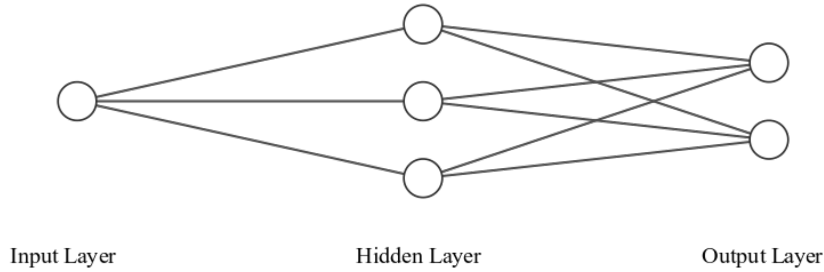


Figure 2.2: A neural network. The chain structure of the network: one neuron in the input layer is connected to three neurons in the hidden layer; neurons in the hidden layer are fully connected to the two neurons in the output layer.

2.2 Pattern learning and generalization

The neural network is designed to solve a problem by learning from a set of examples. In other words, the neural network learns to approximate some function f^*, for example, $y = f^*(x)$ maps input x to a corresponding y – the neural network defines the mapping as $y = f(x, \theta)$ where θ are the parameters that the neural network learns for the approximation of the function f^*.

Network training, or network learning, is a method where initially random valued weights of each connection between neurons are adjusted to best approximate the given function. The rate at which a neural network adapts is called the *learning rate*. Network learning can be divided into three categories: supervised, unsupervised, and reinforcement learning. During *supervised learning*, the teacher provides the network a set of inputs and corresponding outputs. The teacher's task is to indicate whether the network responds correctly to an input, and based on the nature of the response, reward or punish the network. In contrast to supervised learning, *unsupervised learning* does not rely on the teacher to train the network. The network is learning to recognize patterns within the set of inputs without corresponding outputs based on the internally set heuristics. *Reinforcement learning* is a combination of supervised learning and unsupervised learning. The neural network learns in an environment that grades its actions and rewards or punishes the network (as in supervised learning). Actions are based on the internally set heuristics and pattern matching (as in unsupervised learning). Generally, the training of a neural network is finished when the adjustment of the weights has converged.

Error function Error function E calculates the difference between the output of the neural network $f(x, \theta)$ and the output of the given function $f^*(x)$. Throughout the training, learnable parameters θ are adjusted to *minimize the value of* $E(x, \theta)$.

Backpropagation The modification of the weights is calculated as in equation 2.4, where θ are the weights of the network, α is the learning rate, and $E(x, \theta_{old})$ is the error function for given input and current weights. *Backpropagation* is when the weights are updated for one layer at a time, propagating the weight updates from the last layer to the first layer. During backpropagation, the gradient of the difference between the output of the neural network with current weights and the output of the approximated function is calculated, resulting in the direction of the weights that minimize the error function. The weights are

shifted in this direction with the strength set by the declared learning rate α.

$$\theta_{new} = \theta_{old} - -\alpha \frac{\partial E(x, \theta_{old})}{\partial \theta} \tag{2.4}$$

During training, a neural network learns patterns and adjusts weights based on the training data set. In the bigger picture, the training of a neural network is successful if the neural network approximates the output function, which returns correct values for unseen data. This property is called *generalization* – the ability of a neural network to produce correct outputs for data not included in the training set. The common problem that occurs during the training of a neural network is *overfitting*. Overfitting is not suitable for the generalization property of the neural network; overfitting is a phenomenon when a network achieves too good accuracy for predictions on the training data, and the value of the error function on the training data is zero. The error function value of the overfit network is very high for unseen data, resulting in poor performance when making predictions on the unseen data. There are solutions to reduce overfitting and to ensure generalization. One of the solutions is to design the network so that the function being approximated is complex enough. The larger the network, the more complex the output function, resulting in improved generalization. The other solution is *regularization*. During regularization, the penalty terms are added to the error function, allowing the output function to have adequate smoothness [19].

Deep learning

The length of a chain structure of a neural network, or the number of layers in a network, is called *depth*. The name *deep learning* is derived from this terminology. Because every functional network has an input layer and an output layer, the depth is given by the number of hidden layers in the network. A network with multiple hidden layers is called *deep neural network*. *Deep learning* is a branch of machine learning focused on deep neural networks [21].

Feedforward neural network and *recurrent* neural network are types of deep neural networks that are heavily used in the deep learning industry. In feedforward neural networks, information flows from the input through hidden layers to the final output. There is no reverse flow of information or *feedback*. Feedforward neural networks are suitable for processing inputs with fixed length, for example, a matrix representing an image. In contrast, recurrent neural networks allow for feedback connections, enabling processing of data of various lengths, such as sequences of values. Section 2.3 focuses on how feedforward deep neural networks are leveraged in the *computer vision* field.

2.3 Computer vision

Computer vision is a field of artificial intelligence applications focusing on images [22]. The computer processes and analyzes an image or multiple images to derive information from them. The essential concepts for the computer vision field tasks are the feedforward neural network (see section above), *convolutional layer*, *pooling layer*, and the fully connected layer (see section above) [28].

Convolutional layer performs a convolution – scalar product of a feature detector (*kernel*) and the receptive fields of the input. The kernel is a matrix of learnable weights (see

section above) applied to each possible cutout of an input, and their scalar product is calculated as illustrated in Figure 2.3. The final output of the convolutional layer is a series of scalar products, also known as the feature map [17].

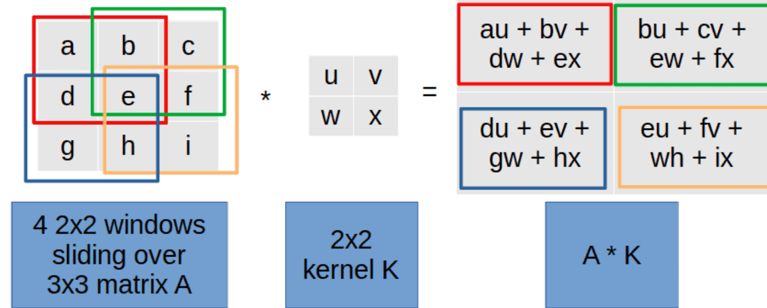


Figure 2.3: 2D convolution operation where a 2×2 kernel slides over a 3×3 input matrix, generating a 2×2 output matrix [1]

Pooling layer reduces the dimensionality of an input. The dimensionality reduction is performed by downsampling of the input – the kernel is applied to a cutout of an input.

The kernel is not a matrix of learnable weights as in the case of the convolutional layer, but in the case of the pooling layer, the kernel contains the aggregation function – for example, the max pooling function or the average pooling function. The max pooling function returns the maximum scalar value of the input. The average pooling function calculates the average value of the input and returns this average. The process of downsampling with max pooling and average pooling aggregate functions is depicted in Figure 2.4.

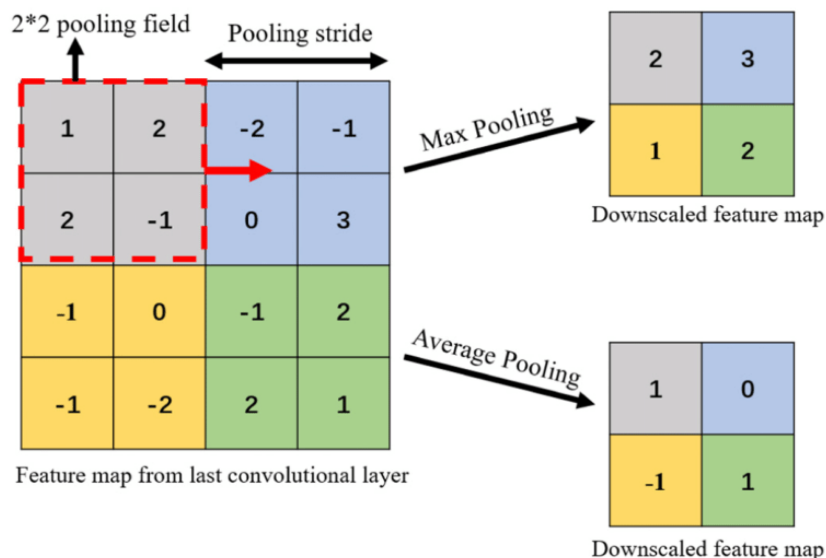


Figure 2.4: Examples of the max and average pooling layers [26]

It is a common practice [28] that the convolution layer (or multiple convolution layers) extracts the feature map from the input and the following pooling layer downsamples the feature map for further computations in fully connected layers.

Convolution and pooling layers also reduce the number of parameters and a model's computational complexity. This reduction is regulated through the *stride* and *padding* hyperparameters. Stride represents the number of rows and columns traversed per slide by the kernel across an image. In Figure 2.3 an overlap occurs because the stride hyperparameter is set to 1. The amount of overlapping is reduced by setting the stride hyperparameter to a greater number, but the produced output dimensionality is reduced. Padding is a technique where extra pixels are added around the border of the input image. The setting of a padding hyperparameter helps to control the dimensionality of the output feature maps. The typical instance of padding is zero-padding, where the extra pixels are set to zero [28].

Deep neural network models with convolutional layers are trained on the image data with a given context to perform computer vision tasks. Popular computer vision tasks include *classification*, *object detection*, and *semantic segmentation*. The following subsections describe these tasks.

Classification

In the computer vision field, classification is the task of assigning a single class to a whole image [7]. The class is assigned to an image using a decision rule based on feature patterns.

When using a deep neural network for image classification, the input image is passed through convolutional layers, which extract the feature patterns. The network's output layer provides information on which class will be assigned to the image.

The classification task may be divided into two more specific subtasks [20]:

- *binary classification* – task of assigning one of the two candidate classes to the input image (a common use case is answering "yes or no" questions),
- *multiclass classification* – assignment of *only one* class of the multiple (more than two) candidate classes.

Object detection

Object detection is the locating objects of certain defined classes within an image. Locating an object means identifying the position and boundaries of an object in the image.

The image is passed to the deep neural network model, which extracts features using convolutional layers and creates [33] broad set of candidate windows. Then, each window in the candidate windows set is classified. Output of the object detection model may be, for example, the image with bounding boxes locating detected objects and a label containing class assigned to detected objects.

Semantic segmentation

Image segmentation is the task of dividing an image into mutually exclusive regions. Each region contains pixels that are mapped to an object. Regions are made with respect to properties such as color or texture – they are uniform and homogeneous [22]. The image segmentation task has multiple variants such as instance segmentation, semantic segmentation, and panoptic segmentation. This section focuses on semantic segmentation.

Semantic segmentation is a particular case of the image segmentation task, focusing on creating partitions that belong to the same class [6]. Deep neural networks performing semantic segmentation receive an image and return predictions for each pixel. The predic-

tions may be in the form of the probability of the pixel belonging to one of the defined classes.

Chapter 3

Model repositories

Training deep neural network models often requires running expensive hardware machines for a long time [34]. The concept of a centralized artificial intelligence model storage emerged to provide a resource-effective alternative to in-house training and storing of models. This chapter describes what can be accessed on model repositories and what benefits they may bring, explores the most widely used model repositories, and how these repositories facilitate the retrieval of models.

3.1 Defining model repositories

A model repository, or model hub, is a platform where artificial intelligence models are stored and shared [24]. Such platforms offer a variety of pre-trained deep neural network models that users can access and use.

What can users come across when browsing model hubs? As mentioned in the previous paragraph, the model hub stores artificial intelligence models. Some of these repositories grant access to a variety of machine learning models regardless of the platform they have been developed on, while other repositories focus solely on artificial intelligence models tied to a specific framework, such as *PyTorch* or *TensorFlow*. Besides machine learning models, model repositories provide various datasets [30]. These datasets can be leveraged for training new artificial intelligence models or fine-tuning pre-trained ones. Having access to a diverse collection of datasets is also useful when it comes to the validation and evaluation of machine learning models. Model hubs also provide space for sharing knowledge and better collaboration on developing state-of-the-art deep neural networks.

Using a model repository speeds up the development of machine learning applications. Users can search for specific machine learning models and apply them to their projects or tasks [24]. This helps reduce model development time and increases efficiency [34]. Another advantage of centralized artificial intelligence hubs that speeds up the development of machine learning applications is simplified access to the large scale of processed data. Sharing ready-to-use datasets on model hubs eliminates the time that otherwise be spent gathering and processing the data. Learning resources available on knowledge sharing feature of model hubs help with education and understanding of artificial intelligence development and usage. Model hubs offer a platform for sharing experience and best practices and thus draw from the results of the work of the experts. Users can also share examples of solutions for frequent tasks in artificial intelligence and information on using models in specific environments. This centralization of information sharing supports collective learning

and dissemination of knowledge across the machine learning community and contributes to the accelerated development. Thanks to these benefits, model repositories have been established as a complex infrastructure for developers and researchers in the artificial intelligence field and have become a foundation of progress and innovation in the utilization of machine learning.

The following sections focus on popular model repositories and their approach to providing models, datasets, and knowledge sharing. Each of the model hubs offers different options and functionality. They differ in various ways, whether it is the framework used for model development or they store artificial intelligence models suitable for specific tasks such as computer vision, natural language processing, et cetera. Popular model repositories, for example, include Kaggle, Hugging Face, PyTorch Hub, and TensorFlow Hub. This thesis focuses on the *Kaggle* and *Hugging Face* repositories since a machine learning community widely uses them, and they are not tied to a single machine learning framework.

Sections describing Kaggle and Hugging Face examine the background and establishment of a given model repository, the models stored in the repository, and the interfaces that can be used to access the data.

3.2 Kaggle

Kaggle is an online platform for data science and machine learning, which, over time, developed into an integral part of the global community of artificial intelligence enthusiasts. The platform can share datasets, models, code, and knowledge among its users, who may compete in various data science and artificial intelligence challenges [9]. Kaggle allows users to work on real problems from industry and research, which motivates innovation and creative solutions.

The platform was founded in 2010, and its rapid growth and popularity led to its acquisition by Google in 2017 [23]. Following this acquisition, Kaggle has grown even further, significantly increasing cloud and computing resources. With increased cloud and computing resources, Kaggle can offer its cloud computational environment called Kaggle Notebooks [9]. Kaggle Notebooks allow users to run the code directly in a browser without installing additional software or buying computational resources.

Kaggle offers many datasets via the Kaggle Dataset interface [9]. Users upload datasets that cover various areas of artificial intelligence tasks. Additionally, Kaggle regularly organizes competitions where specific datasets are provided for the competition tasks.

Kaggle platform also provides access to pre-trained models through its Models interface. Available models range from competition-winning solutions to user-contributed models. Models available on Kaggle are not tied to specific frameworks. On Kaggle, users can choose from models developed on frameworks like PyTorch, Keras, Tensorflow, or ONNX. Regarding the task model is performing, Kaggle allows users to filter through image classification, text generation, classification (general), object detection, and text classification.

This model hub also offers tutorials and courses for users to improve their machine learning and data science skills. The benefit that separates the Kaggle platform from the competition is that the tutorials can run directly in the browser using Kaggle Notebooks. This way, Kaggle dramatically lowers data science and artificial intelligence learning costs. Kaggle also hosts discussion forums in which users can share their experiences, seek advice, and provide solutions that they find while participating in competitions or working on projects.

Downloading items from Kaggle

Users can download items, for example, datasets or pre-trained models, in several ways – via browser, via *Kagglehub Python* library, using *Kaggle CLI* software, or by calling the Kaggle API (application programming interface) directly. All options require a user's credentials. Therefore, the user must create an account on Kaggle.

When downloading items via browser, the user selects the corresponding interface on the Kaggle web page – either the Datasets interface when downloading a dataset or the Models interface when downloading a model. The user can filter and search for items in each of these interfaces. A specific item can be downloaded by clicking on the item, which opens the item's details. The item may be downloaded by clicking the download button on the detail page and confirming the "download as zip" option.

When downloading items via the Kagglehub Python library, using the Kaggle CLI software, or calling the Kaggle API directly, the user must also get to the item's details page. Various options are shown by clicking on the download button besides "download as zip". For example, when downloading using the Kaggle CLI software, the user selects the "Kaggle CLI" option. Afterward, Kaggle generates the code for download, and the user can run it to download the item. Code that Kaggle generates for users to download models and datasets is illustrated in Listings 3.1, 3.2, 3.4, 3.4, 3.5, and 3.6. Before using the previously mentioned tools to download from Kaggle, it is necessary to install the relevant software. In order to download via the Kagglehub Python library, the user must install Python and, using the package-management system, download the *kagglehub* module. When using Kaggle CLI, it is necessary to install this software using Python, using which the *kaggle* module must be downloaded. When directly calling the Kaggle API, the user needs the *cURL*[1] tool.

```
1 import kagglehub
2
3 # Download latest version
4 path = kagglehub.model_download("google/gemma/pyTorch/2b")
5
6 # Download specific version (here version 1)
7 path = kagglehub.model_download("google/gemma/pyTorch/2b/1")
8
9 print("Path to model files:", path)
```

Listing 3.1: Model download via Kagglehub Python library

```
1 import kagglehub
2
3 # Download latest version
4 path = kagglehub.dataset_download("alexandrakim2201/spotify-dataset")
5
6 print("Path to dataset files:", path)
```

Listing 3.2: Dataset download via Kagglehub Python library

```
1 #!/bin/bash
2 kaggle models instances versions download google/gemma/pyTorch/2b/1
```

Listing 3.3: Model download using Kaggle CLI software

[1] cURL is a tool used for data transfers from and to servers [31]. cURL runs on almost every platform and supports several different transfer protocols.

```bash
#!/bin/bash
kaggle datasets download alexandrakim2201/spotify-dataset
```

Listing 3.4: Dataset download using Kaggle CLI software

```bash
curl -L -o ~/Downloads/model.tar.gz  https://www.kaggle.com/api/v1/models/google/gemma/
    pyTorch/2b/1/download -u $KAGGLE_USERNAME:$KAGGLE_KEY
```

Listing 3.5: Model download by directly calling Kaggle API software

```bash
#!/bin/bash
curl -L -o ~/Downloads/archive.zip\
https://www.kaggle.com/api/v1/datasets/download/alexandrakim2201/spotify-dataset
```

Listing 3.6: Dataset download by directly calling Kaggle API software

3.3 Hugging Face

Hugging Face is an online model hub that allows users to explore, experiment, and share open-source machine learning [7]. Hugging Face was founded in 2016 and initially focused on developing the Hugging Face chatbot for teenagers [25]. The code behind the chatbot became open-source, and Hugging Face shifted its focus to being the model hub. The philosophy of the Hugging Face platform is to "democratize and advance machine learning for everyone" [6]. Hugging Face model repository stores over 900 thousand models, 200 thousand datasets, and 300 thousand demo applications.

The Hugging Face repository is designed as a cluster of Git-based[2] repositories – the model repository offers versioning, branches, and commit history for each item stored. The Hugging Face platform is divided into the following segments: Datasets, Models, Spaces, and Docs.

The Datasets segment contains over 200 thousand datasets for various machine learning tasks, which users can filter through. Tasks range from computer vision tasks, such as image classification and text-to-image, audio tasks, like text-to-speech, to natural language processing tasks, such as translation and summarization. Users can filter datasets by tagged tasks, languages, licenses, and by compatible libraries. After selecting the specific dataset, the Dataset Card is shown. The Dataset Card contains the dataset's description and metadata, such as license, tasks, and other tags (used for filtering). The Dataset Card also contains space for community comments, allowing other users to raise pull requests.

The Hugging Face platform offers access to the pre-trained models via the Models segment. In the Models segment, users can search for artificial intelligence models and filter them based on specified tasks, compatible libraries, licenses, and datasets used for training. The Model Card represents each stored model. On the Model Card, users can examine the model description, intended usage, limitations, training parameters, datasets used for training and evaluation, and, in the case of fine-tuned or quantized models, the information about a base model. The Model Card can also contain metrics for a specific model and link to an academic paper, and there is also a slot for a CO_2 impact of the model. Thanks to the Git-based nature of the Hugging Face model repository, users can raise pull requests and lead discussions for each Model Card.

The knowledge-sharing aspect of the Hugging Face model hub takes place on the Spaces and Docs segment of the platform. The Spaces segment allows users to host and share custom machine learning demo applications. Users can discuss and share their ideas regarding

[2] Git is a open source version control system [4].

each demo application in the attached forum. The Docs segment of Hugging Face offers tutorials on various topics such as how to use the Hugging Face model hub, tutorials for training machine learning models, descriptions of artificial intelligence tasks, and use cases. The Hugging Face platform itself provides these tutorials via its official GitHub repository.

Downloading items from Hugging Face.

Only some items from the Hugging Face model hub can be downloaded immediately. Hugging Face allows users to mark uploaded items as *gated* [6]. The *access request* feature is enabled if an item is marked as gated. Users sharing the item can configure the access request form with custom fields. When users want to download an item with the access request feature, they must agree to share their contact information (stated in the user profile) and fill out the access request form. Once the user fills out the access request form, the owner of the requested item can review this request and approve it. Downloading the items with the enabled access request feature is only possible once the request is approved. This implies that users must have an Hugging Face account and provide the account's credentials when downloading items from the Hugging Face model hub.

The Hugging Face platform offers separate Python libraries for downloading items – the `datasets` library for downloading datasets, and the `transformers` library for downloading models. A dataset can be downloaded using the datasets library only if it is tagged as compatible with the library. At the time writing this chapter, more than 70% of the datasets stored in the Hugging Face repository are tagged as compatible with the dataset library. The `datasets` library offers the `load_dataset()` function for straightforward dataset loading. When loading a dataset using the `load_dataset()` function, the user must set the `path` parameter (obtained from the Dataset Card). An example of a straightforward dataset loading from the Hugging Face hub is depicted in Listing 3.7. Similarly, a model can be downloaded using the `transformers` library only if it is tagged as compatible with the library. The proportion of models tagged in the Hugging Face model repository is 40%. The `transformers` library contains the `Automodel` class. The `Automodel` class can be used for a straightforward download of a model using its `from_pretrained()` method. When using this method, the user must specify a pre-trained model path, which can be obtained from the Model Card. The method returns an instance of the `Automodel` class with the requested model structure and weights. An example of a straightforward model loading from the Hugging Face model repository is shown in Listing 3.8. Both `datasets` and `transformers` libraries support download of a specific version, leveraging the Git-based nature of the Hugging Face repository. The Hugging Face platform can also generate code for straightforward download of items. Users can get generated code by clicking the "Use this dataset" button in the Dataset Card or "Use this model" button in the Model Card.

```
1 from datasets import load_dataset
2
3 ds = load_dataset("fka/awesome-chatgpt-prompts")
```

Listing 3.7: Hugging Face: Loading dataset using datasets Python library

```
1 from transformers import AutoModel
2
3 model = AutoModel.from_pretrained("google-bert/bert-base-uncased")
```

Listing 3.8: Hugging Face: Loading model using transformers Python library

Another option for interacting with items stored on the Hugging Face model hub is the `huggingface_hub` Python library. The library provides functions that allow users to download and cache[3] a single item or an entire repository. When downloading a single item, the `huggingface_hub` library offers `hf_hub_download()` function. This function downloads the remote file, caches it on the disk, and returns its local path. The user must provide an identifier of a repository that contains the file and the file name (both can be obtained from the Dataset Card or the Model Card). The library provides the function `snapshot_download()` for downloading a whole repository. The function downloads a remote repository, caches it on the disk, and returns its local path. The user must provide an identifier for the repository. The inputs of both `hf_hub_download()` and `snapshot_download()` functions contain parameter `revision` for a version specification. The revision parameter can be set to the name of a specific branch, a pull request, or a commit hash. As mentioned above, both methods download and cache the requested files. This default behavior and can be overridden by setting the parameter `local_dir` to a custom local path. The downloaded items are still cached, but only within a specified local repository. The example code for downloading via the `huggingface_hub` library is depicted in Listing 3.9 (dataset download) and Listing 3.10 (model download).

```
1 from huggingface_hub import hf_hub_download
2
3 repositoryId = "microsoft/resnet-50"
4 filename = "pytorch_model.bin"
5 revision = "main"
6 hf_hub_download(repo_id=repositoryId, filename=filename)
```

Listing 3.9: Hugging Face: Downloading specific file using huggingface_hub Python library

```
1 from huggingface_hub import snapshot_download
2
3 repositoryId = "microsoft/resnet-50"
4
5 snapshot_download(repositoryId)
```

Listing 3.10: Hugging Face: Downloading a repository using huggingface_hub Python library

Since the Hugging Face model repository is a Git-based repository, all the items can be cloned locally using Git. When downloading items using Git, the Hugging Face platform recommends using an extension of Git – Git Large File Storage[4] (Git LFS). The user must provide a dataset identifier when downloading the dataset and, similarly, a model identifier when downloading the model. The dataset identifier or the Model identifier can be obtained from the Dataset Card or the Model Card. The download of items from the Hugging Face hub using Git is depicted in Listings 3.11 and 3.12.

```
1 #!/bin/bash
2 git lfs install
3 git clone git@hf.co:datasets/google/frames-benchmark
```

Listing 3.11: Hugging Face: Dataset download using Git

```
1 #!/bin/bash
```

[3] Metadata of the items are stored in a cache folder. This prevents re-downloading of up-to-date files.

[4] “Git Large File Storage replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git while storing the file contents on a remote server.“ [5]

```
2 git lfs install
3 git clone git@hf.co:models/google/gemma-2b
```

Listing 3.12: Hugging Face: Model download using Git

Chapter 4

Evaluation metrics for deep neural network models in computer vision tasks

For a neural network to be applicable and produce reliable results, its performance must be evaluated. In order to evaluate model performance, the model's output(s) is compared against the desired output(s), also called ground truth [20]. Therefore, the neural network evaluation process requires a model and a dataset.

This chapter focuses on metrics used for the model evaluation in selected computer vision tasks – classification, object detection, and semantic segmentation. Each metric provides unique information about the evaluated model's specific property, and combining multiple metrics may determine the overall performance. Understanding the model's overall performance helps with the usage and deployment of a model in a specific environment.

4.1 Classification

Evaluation metrics used for the classification task can be divided into two subcategories, similar to the division of the classification task into two subtasks (see Section 2.3). A slight difference exists between the evaluation metrics used for a neural network solving a binary classification problem and those used for multiclass classification models.

Binary classification

The metrics used for evaluating a model performing binary classification serve as a foundation [20] for the metrics used in multiclass and multilabel classification tasks.

Let c_0 and c_1 be two candidate classes – the model solves the task of assigning one of the two candidate classes to the image. Let $P(c_0)$ be the output of a model, representing the probability that the image belongs to the class c_0. The probability $P(c_1)$ that the image belongs to class c_1 is therefore defined as $P(c_1) = 1 - -P(c_0)$. The dataset used for evaluation contains images and the respective ground truths – 0 if the image belongs to class c_0, and 1 when the image belongs to the class c_1. Since the output of a neural network is not binary, a threshold T [20] must be chosen to make the decision positive or negative. The model's predictions can be categorized [20] into four categories by comparing the predicted probability (predicted label) with the ground truth, as in Table 4.1. This categorization of

predictions must always be done with respect to a specific class. Table 4.1 categorizes the predictions with respect to class c_0 (see the example model above).

Value	Relation with threshold	Ground truth
true positive	$P(c_0) \geq T$	0
false positive	$P(c_0) \geq T$	1
true negative	$P(c_0) < T$	1
false negative	$P(c_0) < T$	0

Table 4.1: Categorization of predictions with respect to c_0 class. Note: Case's predicted class is c_0 if $P(c_0) \geq T$. Case's predicted class is c_1 if $P(c_0) < T$. Ground truth 0 stands for c_0; 1 stands for c_1.

Confusion matrix The confusion matrix [20] shows the ratio of the predictions categorized by comparison with the ground truth (as in Table 4.1). In the confusion matrix, the rows represent predicted classes and the columns represent ground truth. *A table cell represents a number of cases belonging to the category given by row and column.* Table 4.2 shows an example of a confusion matrix for a binary classification problem.

		Ground truth	
		0	1
Predicted class	c_0	TP	FP
	c_1	FN	TN

Table 4.2: Confusion matrix for a binary classification problem with respect to c_0 class. Note: TP = true positive, TN = true negative, FP = false positive, FN = false negative.

The true positive, true negative, false positive, and false negative values can be used further to derive other binary classification metrics [20]. All metrics derived from prediction categorization depend on the chosen threshold. The number of c_0 predicted classes rises when the threshold is lowered; on the contrary, the number of c_1 predicted classes rises as the threshold is set higher. Therefore, the metrics summarized in Table 4.4 are functions of the threshold.

Multiclass classification

The metrics used to evaluate models that solve a binary classification problem can be extended to evaluating models that solve multiclass classification problems [20].

In the multiclass classification task, the confusion matrix rows are analogous to the confusion matrix for a binary classification problem. The rows represent all the predicted classes, and the columns represent ground truth. Cells of the matrix display the number of cases predicted as a class in the corresponding row, and their desired class is the class represented by the corresponding column. An example of a confusion matrix for the multiclass classification problem is depicted in Table 4.3.

When categorizing predictions of a multiclass classification model, the categories (TP, FP, TN, FN) are calculated separately for each class. True positives (TP) are the cells in the confusion matrix where the row equals the inspected class and the column also equals the inspected class. False positives (FP) are the sum of cells in the column corresponding

to the inspected class, except for the diagonal cell. False negatives (FN) are the sum of the cells in the row corresponding to the inspected class, except for the diagonal cell. True negatives (TN) are the sum of all cells not belonging to the row or column representing the inspected class.

		Ground truth			
		A	B	C	D
Predicted class	A	200	0	0	0
	B	160	18	2	2
	C	20	0	16	0
	D	20	2	2	18

Table 4.3: Confusion matrix for a multiclass classification problem. Example categorization for class A: $TP = (A, A) = 200$, $FP = (B, A) + (C, A) + (C, D) = 200$, $FN = (A, B) + (A, C) + (A, D) = 0$, $TN = (B, B) + (B, C) + (B, D) + (C, B) + (C, C) + (C, D) + (D, B) + (D, C) + (D, D) = 60$.

The metrics derived from the confusion matrix are calculated in the same way as for binary classification. For each class, the corresponding prediction categorization (see the example confusion matrix in Table 4.3) is used to calculate all derived metrics, such as recall, precision, or accuracy. Table 4.4 summarizes key performance metrics for evaluating binary classification models, including their definitions and corresponding formulas.

The metrics for the multiclass classification models can be divided into class-wise metrics and averaged metrics [20]. The class-wise metrics are calculated using the approach described above. Class metrics are calculated based on true positives and negatives, and false positives and negatives for each class. For example, the model recall for class A can be expressed as:

$$recall_A = \frac{TP_A}{TP_A + FN_A}$$

The averaged metrics describe how the model performs overall, with respect to all classes. The averaged metrics can be further divided [20] into:

- Macro-average – the given metric average with the same weight for each class. The macro-average of a metric, or macro metric, is calculated as the sum of the given metric for each class divided by the total number of classes. For example, macro-recall of a model can be calculated as:

  $$recall_{macro} = \frac{recall_A + recall_B + ... + recall_{lastclass}}{N}$$

  where N is the number of classes and the nominator contains the sum of recalls with respect to each class.

- Micro-average – the given metric average with the same weight for each data point. The micro-average of a metric (micrometric) is calculated similarly to metrics in Table 4.4. However, instead of categorizations (e.g., true positives) for each class, the sums of categorizations are used in the formulas. For example, micro-recall of a model can be expressed as:

  $$recall_{micro} = \frac{TP_{total}}{(TP_{total} + FN_{total})}$$

  where TP_{total} is the sum of true positives for each class, and FN_{total} is the sum of false negatives for each class.

Metric	Description	Formula
Recall	Recall, also called sensitivity or *true-positive* rate, represents the „the fraction of positive cases predicted as positive [20]“.	$\frac{TP}{TP+FN}$
Specificity	Selectivity, or *true-negative* rate, represents „the fraction of negative cases predicted as negative [20]“.	$\frac{TN}{TN+FP}$
False-positive rate	The fraction of negative cases predicted as positive [20].	$\frac{FP}{TN+FP}$
False-negative rate	The fraction of positive cases predicted as negative [20].	$\frac{FN}{TP+FN}$
Precision	Precision, or positive predictive value, is proportion of true-positive cases against all positive predicted cases [20].	$\frac{TP}{TP+FP}$
Negative predictive value	Proportion of true-negative cases against all negative predicted cases [20].	$\frac{TN}{TN+FN}$
Accuracy	Percentage of correctly predicted cases [20].	$\frac{TP+TN}{TP+TN+FP+FN}$
F1 score	The harmonic mean of precision and recall [20].	$\frac{2TP}{2TP+FN+FP}$

Table 4.4: Definitions and formulas for selected performance metrics used to evaluate binary classification models. Each metric is defined in terms of the confusion matrix components. Note: TP = true positive, TN = true negative, FP = false positive, FN = false negative.

4.2 Object detection

Neural networks performing an object detection task identify the positions of objects in an image by placing bounding boxes on the image [29]. The output of such models is a set of three attributes [29]:

- the object class,
- the corresponding bounding box,
- the confidence score, usually on interval $< 0, 1 >$, showing model's confidence on detection.

The object detection model's output is evaluated by comparison to an annotated image. The image annotation consists [29] of a set of ground-truth bounding boxes containing objects belonging to a class. The metrics used for the object detection task measure how close the predicted bounding boxes are to the ground truth bounding boxes for each class defined in a task.

Intersection over union Consider a ground-truth bounding box B_{gt} representing an object to be detected and a bounding box B_p predicted by a model. Without considering a confidence level, the perfect match is where ground-truth and predicted bounding-boxes have the same area and location. The *intersection over union (IoU)* metric can evaluate these two conditions. Intersection over union is equal to the area of the intersection between ground-truth bounding box B_{gt} and predicted bounding box B_p divided by the area of their union [29]:

$$IoU = \frac{B_p \cap B_{gt}}{B_p \cup B_{gt}}$$

The intersection of the union is the value on the $< 0, 1 >$ interval. The better the detection, the closer the IoU value is to 1. The perfect match situation occurs when $IoU = 1$. When $IoU = 0$, the ground-truth and predicted bounding-boxes do not intersect.

Considering that the output of object detection models also consists of an object class, the IoU metric can be used only for bounding boxes of the same class.

Precision and recall Another way to evaluate object detection models is to evaluate their ability to identify only relevant objects and their ability to find all objects within the image [29]. The ability of a model to identify only relevant objects is called *precision*. Precision can be expressed as a percentage of correct positive predictions. The ability of a model to find all relevant cases is called *recall*. Recall can be expressed as the percentage of correct positive predictions among all ground truths.

When calculating the precision and recall values, each predicted bounding box of the *same predicted class* must be categorized as true positive, false positive, or false negative. The predicted bounding boxes are categorized by comparing a selected *IoU threshold* with the IoU value of the predicted bounding box and the corresponding ground-truth bounding box:

- True positive (TP) – predicted bounding box is the same class as the ground-truth and $IoU \geq threshold$,
- False positive (FP) – predicted bounding box is the same class as the ground-truth and $IoU < threshold$,
- False negative (FN) – no bounding box meets the IoU threshold for a ground-truth object (ground-truth object was not detected).

After obtaining true positive, false positive, and false negative values, the precision and recall metrics can be expressed as:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

Mean average precision Mean average precision represents the correctness of the model's detections between all classes [29]. The mean average precision (mAP) is calculated as:

$$mAP = \frac{1}{C}\sum_{i=1}^{C} AP_i$$

where C is the total number of classes and AP_i is the *average precision* for the given class i. The average precision for class c (AP_c) is a metric based on the area under the *precision-recall curve*. Average precision is obtained by calculating the precision and recall metrics for different classifier thresholds. By moving the classifier threshold, the number of true positive predictions changes; therefore the value of precision and recall metrics also changes. The average precision for class c can be expressed as:

$$AP_c = \sum_k (R(k) - -R(k+1))P_{interp}(R(k))$$

where $P_{interp}(R(k))$ is the *interpolated precision* at a specific recall and $R(k)$ is the recall at the k-th classifier threshold. Therefore, predictions for class c are sorted (in descending order) based on their confidence score (the output of the object detection model consists of class, a box of bounds, and a confidence score), and the precision and recall pairs are calculated with respect to each confidence score set as the classifier threshold. The precision is interpolated to ensure that the precision-recall curve is monotonic. The interpolated precision at a certain recall level $P_{interp}(R)$ can be expressed as the maximum precision whose corresponding recall value is greater than or equal to R.

4.3 Semantic segmentation

When evaluating models performing the semantic segmentation task, the goal [27] is to compare the prediction and ground truth (annotated segmentation). Semantic segmentation can be perceived as a classification at the pixel level [20]. Semantic segmentation with two mutually exclusive regions can be perceived as binary classification, while semantic segmentation with more than two mutually exclusive regions can be perceived as multiclass classification. Therefore, metrics used to evaluate a model performing a classification task (see Section 4.1 and Table 4.4) can also be used to evaluate a semantic segmentation model. Before using the metrics in Table 4.4, it is important to construct the confusion matrix (see Section 4.1) – this involves categorizing the predictions into true positives, true negatives, false positives, and false negatives categories. Metrics based on prediction categorization that are commonly [27] used for the evaluation of semantic segmentation models are:

- *Accuracy* – the number of correct predictions, or correctly assigned pixels to the region, compared to the annotated segmentation. Similarly to binary classification, the accuracy metric is calculated as:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- *Itersection over union* – itersection over union (IoU) measures the similarity of a predicted and annotated segmentation. IoU is defined [29] as the ratio between the intersection and the union of the prediction X and the ground truth Y:

$$IoU = \frac{X \cap Y}{X \cup Y}$$

Regarding the prediction categorization and confusion matrix, the intersection over union is calculated as:

$$IoU = \frac{TP}{TP + FN + FP}$$

- *Recall* – capability of a model to assign a pixel to the correct region (true positive rate). Recall is calculated as a ratio of true positives to the sum of true positives and false negatives:

$$recall = \frac{TP}{TP + FN}$$

- *Specificity* – capability of a model to distinguish other regions from a selected region (true negative rate):

$$specificity = \frac{TN}{TN + FP}$$

- *Dice similarity coefficient (DSC)* – measures the similarity of a segmentation predicted by a model and the ground truth (annotated segmentation) [20]. DSC is defined as the two-time intersection of a predicted and annotated segmentation, divided by the union of the predicted segmentation and the annotated segmentation. This can be expressed as:

$$DSC = \frac{2(X \cap Y)}{X \cup Y}$$

where X is the predicted segmentation and Y is the ground truth.

When applied to the prediction categorization approach, the DSC metric is the harmonic mean of precision and recall (equal to the F1 score metric used to evaluate classification models):

$$DSC = \frac{2TP}{2TP + FN + FP}$$

In the case of semantic segmentation with more than two distinct regions, the metrics above must be calculated with respect to each class [27]. For example, when the model performs semantic segmentation of an image into three distinct regions A, B, C, the recall metric is calculated for each distinct region separately. Similarly to multiclass classification, the metrics for semantic segmentation with more than two mutually exclusive regions can be *macro-averaged* and *micro-averaged* as stated in Section 4.1.

Average Hausdorf distance is a spatial distance metric that does not depend on a confusion matrix [27]. *Directed average Hausdorf distance* is the distance between the boundaries of the ground truth region and the predicted region [20]. The directed average hausdorf distance can be expressed as:

$$d(A, B) = \frac{1}{N} \sum_{a \in A} min_{b \in B} p(a, b)$$

Where A is the annotated segmentation, B is the predicted segmentation, the function $p(a, b)$ is a distance function (e.g., Euclidean distance), and $d(A, B)$ is a directed average Hausdorf distance between regions A and B.

The *average Hausdorf distance (AHD)* is calculated as the maximum between the directed average Hausdorf distance for the annotated segmentation A and the predicted segmentation B in both directions:

$$AHD = max(d(A, B), d(B, A))$$

4.4 Inference latency

In addition to the metrics discussed in the previous sections, such as accuracy, F1-score, and IoU, which are computed by comparing the model's output against a desired output (ground truth), this section focuses on a different evaluation metric – inference latency.

Unlike accuracy-based metrics, inference latency does not require a pair of input and ground truth output. Instead, it is measured solely in terms of time, reflecting how long the model takes to process a given input and produce an output.

Inference latency is the total time required for a neural network to complete the inference process for a single input. This duration can vary significantly depending on factors such as the complexity of the model architecture, input data dimensions, and available hardware and software.

Several specific latency-related metrics are used to analyze the inference performance more precisely:

- Initialization time – The time taken to load the model, allocate necessary resources, and prepare the inference environment. This includes parsing the model file and setting up memory buffers.
- Inference time in the warmup state – Refers to the latency measured during the first few inferences immediately after model initialization. These early inferences may not reflect actual performance due to initial cache misses.
- Inference time in the steady state – The average time taken for inference after the system has „warmed up". The steady state inference time is typically the mean or median latency over many runs.

Chapter 5

System for automatic testing and evaluation of neural networks on embedded platforms

This chapter focuses on the design of the system for automatic testing and evaluation of neural networks on embedded platforms. The chapter explores software requirements for the system, the proposed design of the system architecture, system components and their interactions, used architectural patterns, main functionalities of the system, user interactions and flows, and the proposed data design.

5.1 System requirements

This section defines the requirements specification for a system designed for automated testing and evaluation of deep neural network models on embedded platforms. The system automates model deployment, performance evaluation, and regression detection by comparing current results with past performance records. The system's performance evaluation feature focuses on the given model's accuracy and latency. The system shall detect regression of model performance against previous results.

The system must support retrieval of the models and dataset from remote storages (model hubs, see Chapter 3). The system shall support neural network models in TensorFlow Lite[1] format; this requirement is set by the NXP deployment environment, where the target embedded platforms integrate inference engines with support for the TFLite runtime. The tested neural network models shall focus on computer vision tasks (see Section 2.3): classification, object detection, and semantic segmentation.

The system must enable the deployment of models on Linux-based embedded platforms. *The model will be evaluated on the embedded device* by measuring the model's latency and accuracy with respect to the given test dataset. The regression detection part of the system must compare actual test results with historical data.

The system shall produce output(s) incorporating:

- A summary of the on-device measured accuracy metrics specific to the given model's task.

[1] TensorFlow Lite (TFLite) is a neural network framework developed by Google. The framework provides a means for deploying the neural networks on the embedded devices [10].

- A summary of the on-device measured latency metrics of the given model.
- Interface dedicated to regression tracking within the accuracy and latency results.

5.2 Inputs and outputs of the proposed system

The input of the proposed system is a user-defined JSON file. The file contains several required objects:

- Devices: a list of objects representing embedded devices used during the evaluation process. Each object contains the required fields:
  - URI: a unique address that identifies an embedded device on the network used by the system,
  - Name: a unique name selected by the user,
  - Credentials: an object that allows the specification of the required user name (`user_name` field) and password (`password` field) if the embedded device requires authentication during a communication via the SSH protocol.
- Models: a list of objects representing models stored on a model hub (see Chapter 3). Each object contains the required fields:
  - Platform: an identifier of a model hub (model hubs supported by the system are specified in Section 5.1),
  - URI: the model's location on a platform specified in the platform field,
  - name of the model's task (tasks supported by the system are specified in the Section 5.1),
  - Name: a unique custom name selected by the user.
- Datasets: a list of objects representing datasets stored on a model hub. Each object contains the required fields:
  - Platform: an identifier of a model hub,
  - URI: the dataset's location on a platform specified in the platform field,
  - Dataset keys: a mapping of task-specific column names to semantic roles within the dataset (e.g., input images, ground truth labels, bounding boxes),
  - Name: a unique custom name selected by the user.
- Runs: a list of objects representing a requested evaluation of a given model on a given embedded device over a given dataset. Each object contains the required fields:
  - Device name: the evaluated model's custom name,
  - Model name: custom name of the device on which the model will be evaluated,
  - Dataset name: custom name of the dataset used for accuracy evaluation,
  - Script path: path to a user-implemented script responsible for handling the model's outputs (see below).

Each run entry specifies, among other parameters, the path to a user-implemented script. These scripts are intended to process the raw outputs produced by the model and align them with the corresponding ground truth data, enabling the framework to compute task-specific evaluation metrics. The need for the user script derives from the lack of standardization in model output formats within the same task category. Output structures vary significantly depending on the model architecture and pre-/post-processing routines. Delegating the output processing to the user ensures flexibility and allows the framework to remain indifferent to model-specific implementations while maintaining a consistent metric computation pipeline.

Listing 5.1 illustrates an example configuration file that follows the structure proposed above. Listing demonstrates how users define the requested evaluation, including target devices, assessed models, datasets selected for accuracy evaluation, and the corresponding evaluation run configuration. Although each array in the example configuration contains only one object for clarity, the structure supports multiple entries for devices, models, datasets, and runs.

```
{
  "devices": [
    {
      "uri": "0.0.0.0",
      "name": "device1",
      "credentials": {"user_name": "root", "password": ""}
    }
  ],
  "models": [
    {
      "name"    : "mobilenetv1",
      "task"    : "image-classification",
      "platform": "kaggle",
      "uri"     : "tensorflow/mobilenet-v1/tfLite/1-0-224-quantized"
    }
  ],
  "datasets": [
    {
      "name": "mini_imagenet_classification",
      "platform": "kaggle",
      "uri": "deeptrial/miniimagenet",
      "dataset_keys": {"input": ["image"], "target": ["label"]}
    }
  ],
  "runs": [
    {
      "device_name": "device1",
      "model_name": "mobilenetv1",
      "dataset_name": "mini_imagenet_classification",
      "required_script_paths": {
        "align_target_predicted": "mobilenetv1_mini_imagenet_classification.py"
      }
    }
  ]
}
```

Listing 5.1: Example configuration file requesting a single evaluation run. The file specifies the device, model, dataset, and the run tuple referencing the device, model, dataset, and the required post-processing script.

The main output of the system is an HTML page summarizing the executed runs defined in the input file. The information is grouped by device, and for each run, the configuration details, evaluated accuracy metrics, and evaluated latency metrics are displayed. Figure 5.1 illustrates the proposed design of the summarizing HTML page.

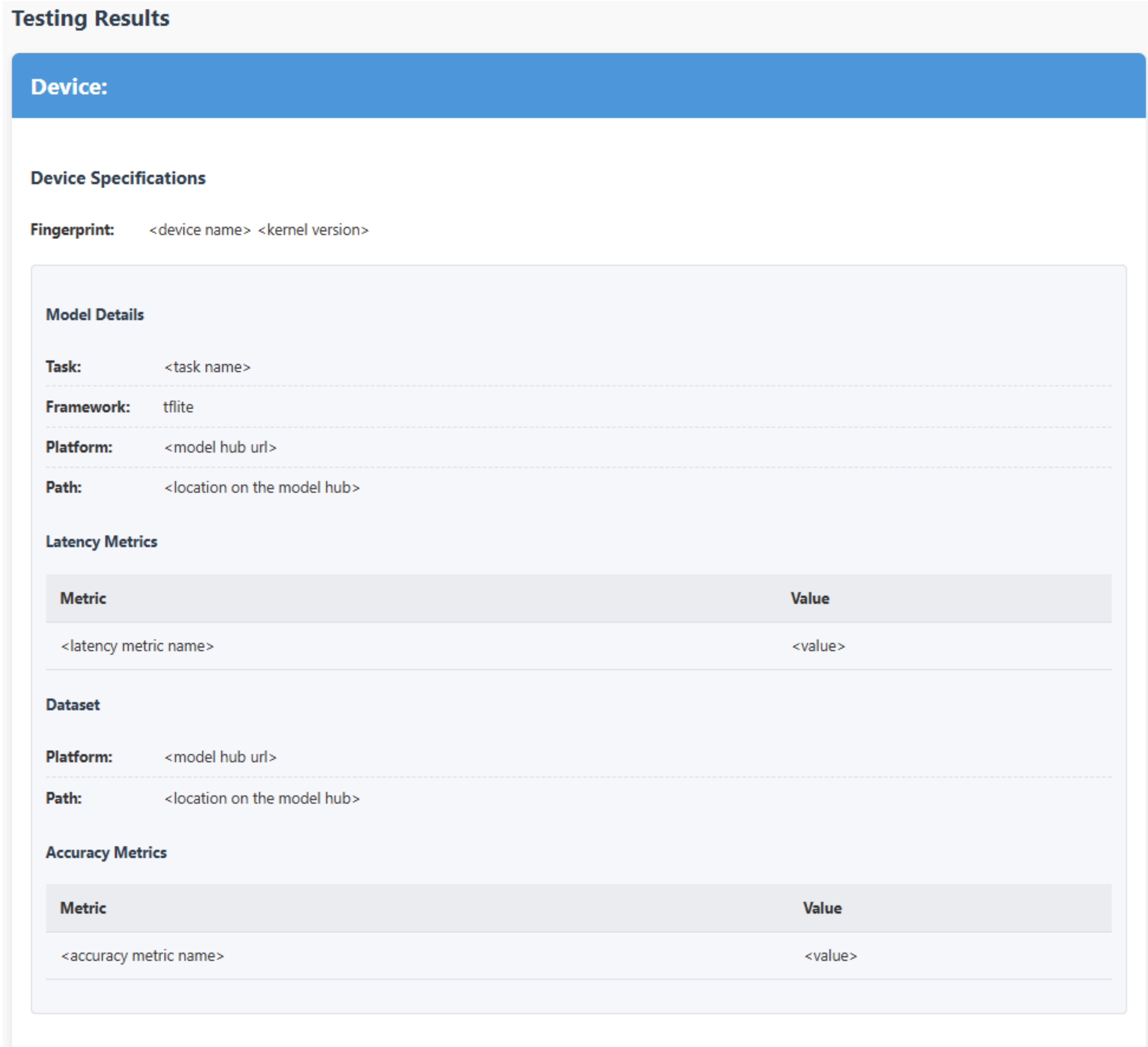


Figure 5.1: Proposed testing results report.

Considering the system's regression tracking requirement (see Section 5.1), the system is set to provide an interface for regression tracking. The interface is designed as a pair of HTML pages: one dedicated to tracking latency regressions and the other focused on accuracy regressions. Each page allows the user to configure a comparison by selecting the model, the target device (identified by its system name), and two kernel versions for side-by-side comparison. Regarding accuracy tracking, the user selects the dataset used during the evaluation. Upon selection, the interface displays a tabular summary of metrics – task-specific accuracy metrics or latency measurements are shown side-by-side for both kernel versions. The summary also includes the absolute differences for each metric. Figure 5.2 illustrates the proposed design of the HTML interface focused on latency regression,

and Figure 5.3 illustrates the proposed design of the HTML interface focused on accuracy regression.

Model latency comparison

Model:

selected model

Device:

selected device

Metric	selected kernel version 1	selected kernel version 2	Difference (absolute)
latency-metric-name	321	345	24

Figure 5.2: Proposed regression tracking interface for latency comparison

Model accuracy comparison

Model:

selected model

Dataset:

selected dataset

Device:

selected device

Metric	selected kernel version 1	selected kernel version 2	Difference (absolute)
accuracy-metric-name	0.064	0.065	0.001

Figure 5.3: Proposed regression tracking interface for accuracy comparison

5.3 System architecture

This section describes the proposed system's design for automatic neural network testing on embedded platforms. The architecture design follows a modular architecture, where each module is responsible for a specific aspect of the testing process. The three major modules (depicted in Figure 5.4) are:

- *Device API*: Handles communication with the embedded device.
- *Model Tester*: Orchestrates testing according to the user-defined configuration.
- *Report API*: Manages data storage and reporting.

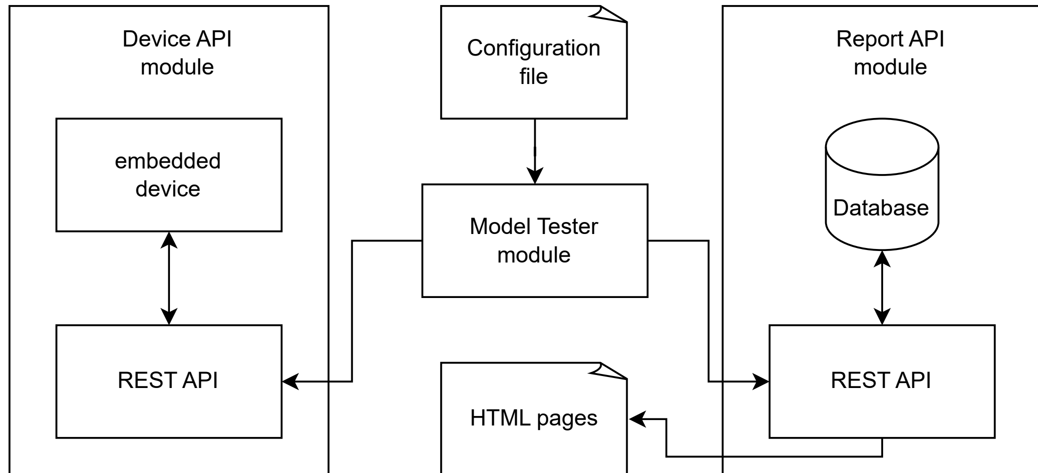


Figure 5.4: High-level modular overview of the system architecture

The execution, illustrated in Figure 5.5, begins when the user provides a configuration file to the Model Tester component, which parses the file to extract the configuration objects (defined in Section 5.2). It then proceeds to download all referenced models and datasets. For each defined run, the system uploads the specified model to the target device via the Device API. Once the model is uploaded, the system collects latency measurements by calling the latency evaluation endpoint on the Device API. Following the latency evaluation, the specified dataset is pre-processed to match the model's input and batched. Each batch is sent to the device, and the resulting model outputs are retrieved. Model outputs are passed to a user-defined script responsible for post-processing. The system subsequently uses the processed outputs to compute task-specific evaluation metrics concerning the model type. Finally, the Model Tester submits the results and run metadata to the Report API. Once all runs are finished, the system requests the generation of a final report document, which is returned to the user as the output of the evaluation process.

5.3.1 Device API

Device API is a module responsible for the evaluation of neural networks on-device. This module is an interface providing communication between an embedded device on which the given model will be evaluated and the Model Tester module.

The API contains endpoints for:

- Device identification: Returns device name and operating system kernel version (Figure 5.6 depicts the proposed endpoint interaction).
- Model deployment: Deploys the selected model onto a device (Figure 5.7 depicts the proposed endpoint interaction).
- Latency measurement: Executes on-device latency evaluation of the selected model and returns the assessment results.
- Batch inference: Executes on-device inference of a given dataset batch on the selected model and returns the model's outputs.

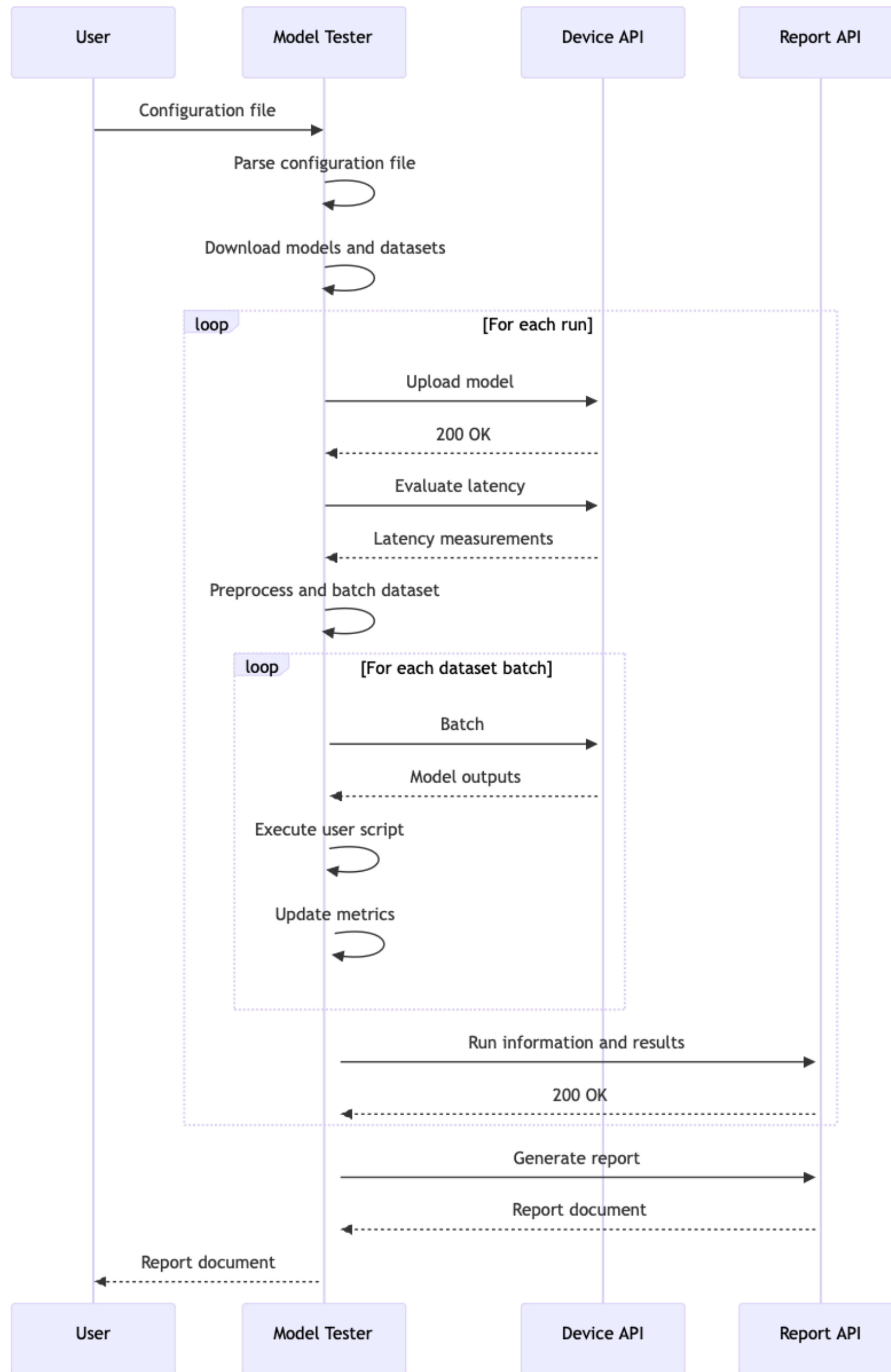


Figure 5.5: Orchestrator script execution flow

5.3.2 Report API

The Report API manages the persistence of evaluation results and generates reports. The design of the documents generated by the Report API is documented in Section 5.2. The Report API module also provides an interface for storing and retrieving test results from the database.

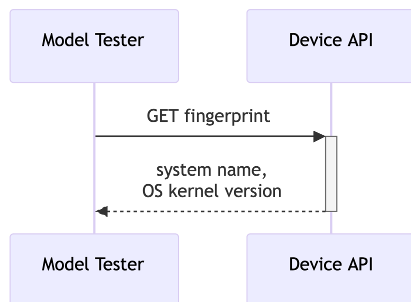


Figure 5.6: Sequence diagram depicting device identification interaction between the system's modules

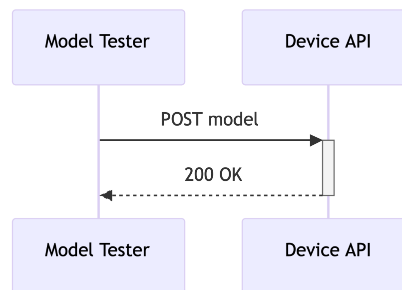


Figure 5.7: Model deployment process sequence diagram

The primary functionalities of the *Report API* module include:

- Storing run results: The API receives details of the test run, including model identifier, dataset identifier, latency measurements, and accuracy metrics. These results are stored in a SQL database for future reference.
- Generating reports: The API generates a report document for given run identifiers.
- Latency comparison: Returns a document that compares the latency results of the selected model on different kernel versions within the selected device (identified by a device name).
- Accuracy comparison: Returns a document that compares the accuracy results of the selected model over the selected dataset on different kernel versions within the selected device (identified by a device name).

Figure 5.8 depicts the relational database schema used in the system. At the core of the schema is the Run entity, which represents a single assessment run made by the system. Each Run record captures the context in which a model was tested on the associated device, with the associated dataset. The database schema also includes tables representing evaluated models and their metadata (Model entity), datasets used for accuracy evaluation with their metadata (Dataset entity), and embedded devices – identified by name and kernel version of their system – on which the assessment took place (Device entity). The Run entity establishes a many-to-one relationship with Model, Dataset, and Device entities.

Additionally, the schema includes a Metric entity for storing evaluation results, where individual metric values, whether related to accuracy or latency, are linked to their corresponding run. The values in the database are populated via the module's internal API (described below). The values stored in the database are used when the module generates the system's HTML-based outputs (see Section 5.2).

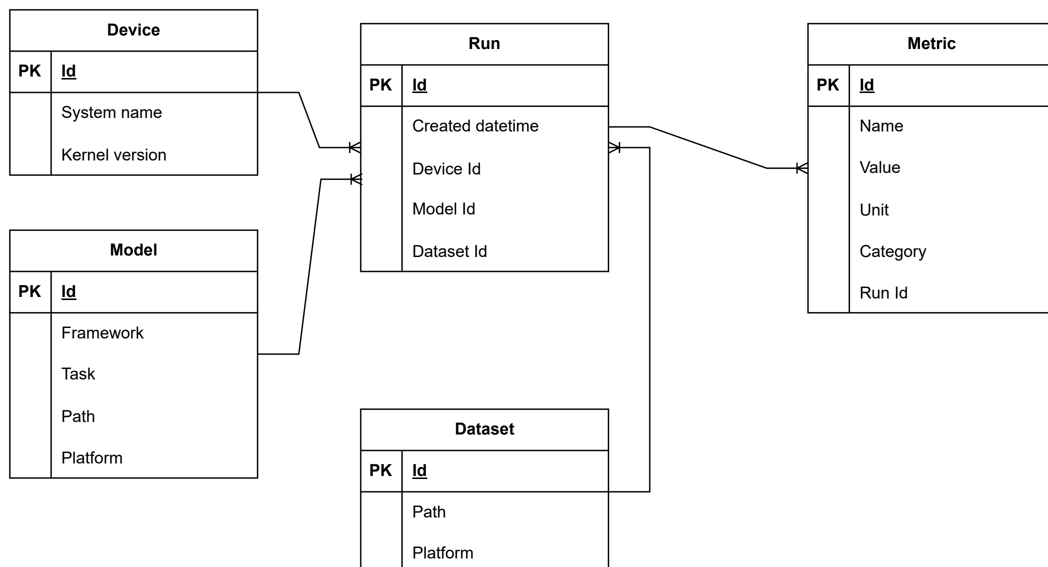


Figure 5.8: Entity relationship diagram of the database storing the assessments made by the system. The diagram captures metadata about each evaluation run, including the associated model, dataset, device, and metrics.

5.3.3 Model tester

The *Model Tester* module is the central element of the system's architecture. The module acts as the system's central processing unit and orchestrates the testing workflow based on the user-defined configuration.

The key responsibilities of the Model Tester include:

- Configuration file parsing: Model Tester loads and parses the user-defined configuration file (see Section 5.2), extracting relevant test parameters for models, datasets, devices, and runs.
- Downloading of the models and datasets: The module fetches pre-trained neural network models and datasets from model repositories and stores them on a host device[2].
- Executing test runs: Each run is defined as a tuple consisting of the model, testing dataset, and embedded device. Model Tester module for each run:
  1. Deploy the model onto the embedded device using the Device API.
  2. Calls the latency measurement endpoint to retrieve the model's latency statistics on the embedded device.
  3. Evaluates the model's on-device accuracy with respect to the testing dataset: The Model Tester module preprocesses the dataset by normalizing inputs, applying transformations, and batches the dataset. The batched dataset is iteratively fed to the model by invoking the batch inference execution endpoint and collecting the model's outputs. After retrieving the model's output for the current

[2] Device running the Model Tester module.

batch, the script specified by the user executes the post-processing of the model's outputs and aligns them with the ground-truth values. Afterward, the accuracy metrics defined for the model's specific task (e.g., semantic segmentation) are dynamically updated.

4. Computes defined accuracy metrics for the model's specific task.
5. Stores the run latency and accuracy metrics in the database via the Report API.

- Generating a report file: The Model Tester module passes the identifiers of executed runs to the Report API endpoint responsible for the report generation process.

Chapter 6

Implementation of the system

This chapter details how the design decisions laid out in Chapter 5 have been realized in code. The following sections describe the concrete implementation of each module in the architecture presented in Section 5.3, highlighting key classes, external libraries, and inter-module communication patterns.

The system is implemented in *Python*, and the implementation follows the architectural design presented in Chapter 5. The system is designed to be executed as a script via the Python interpreter. Each module proposed in Chapter 5 is mapped to a dedicated Python module; this modular approach aligns with the system's design principles. The source code is organized within the `src` directory and structured into separate folders for each module.

The following subsections provide insights into the implementation of each module, highlighting their functionality, key implementation details, and integration within the overall system.

6.1 Model Tester

The Model Tester module is implemented in the `src/modelTester` directory, which contains all the supporting classes and the `modelTester.py` Python script. It serves as the primary execution driver of the system (see Chapter 5) and is the primary point of user interaction. The user invokes the script by providing a path to a JSON configuration file as a command-line argument `--cfg=<PATH>`.

The following subsections describe the implementation of the Model Tester module in the context of the script execution flow. Figure 6.1 provides an overview of the key classes and their relationships as discussed in this section.

6.1.1 Configuration parsing

Upon execution, the script instantiates the `ConfigParser` class, which is responsible for loading, parsing, and validating the configuration file. Parsing is performed during object construction using the file path supplied as an argument. Configuration parsing and validation are handled using the `Pydantic` library, which enforces type correctness and schema constraints. Each logical configuration unit described in Section 5.2 is implemented as a distinct class inheriting from Pydantic's `BaseModel`. The `BaseModel` class enables automatic parsing of nested JSON structures, field validation, and error reporting, which helps ensure that invalid configurations are detected before execution begins. The top-level

configuration class also inherits from `BaseModel` and encapsulates lists of the individual configuration components.

In addition to parsing, the `ConfigParser` class provides accessor methods to retrieve specific subsets of the configuration. The `get_downloader_cfg()` method returns the configuration objects related to external resources (models and datasets) hosted on model hubs. The `get_evaluator_cfg()` method returns two collections: one representing the embedded devices available for evaluation and the other representing the list of declared evaluation runs.

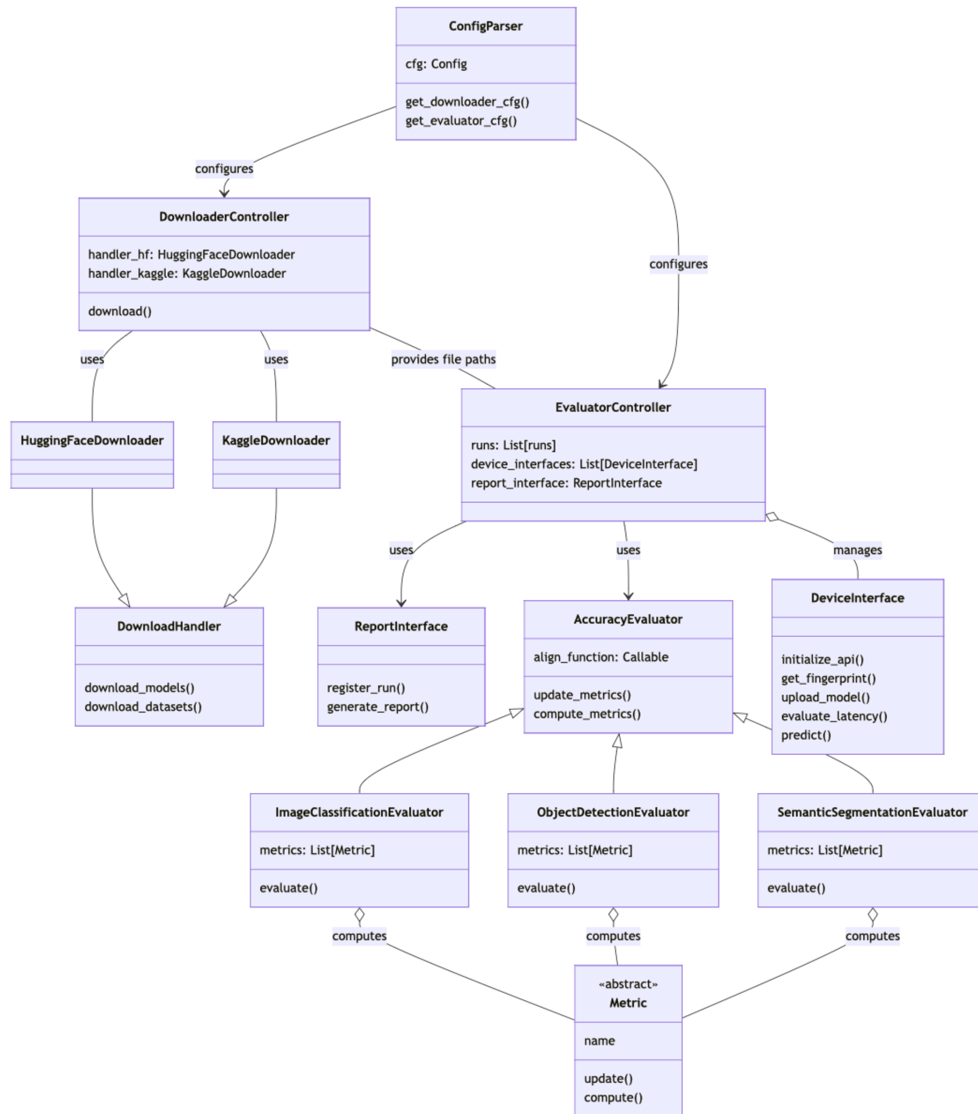


Figure 6.1: Class diagram of the Model Tester module illustrating the main classes involved in the evaluation workflow

6.1.2 Device APIs initialization

Following successful configuration parsing, the script initializes the Device API on all embedded devices specified in the configuration. This functionality is encapsulated within the `EvaluatorController` class, which manages communication with embedded devices. The script creates an instance of `EvaluatorController`, passing the list of device objects and evaluation runs obtained from the `get_evaluator_cfg()` method (see above).

Upon instantiation, the `EvaluatorController` constructs a `DeviceInterface` object for each device in the list. These objects are stored internally as a list and serve as the interface through which the system interacts with the respective embedded hardware.

The `DeviceInterface` class is responsible for initializing and managing the Device API on the target device. When instantiated, it establishes an *SSH connection* to the specified device IP address using the `asyncssh` library – a Python library providing an asynchronous client and server implementation of the SSH protocol [2]. Once the connection is established, the `DeviceInterface` object pulls the required *Docker image* and launches the Device API within a *Docker container*[1].

In addition to Device API initialization, the `DeviceInterface` class provides methods that encapsulate *REST-based communication* with the running Device API. These methods, implemented using the `requests` library, are responsible for model uploading, latency evaluation, and batch inference execution and are described in more detail later in this chapter.

6.1.3 External resources downloading

The next step in the execution flow involves downloading models and datasets referenced in the configuration file. This process is handled by the `DownloaderController` class. The script passes the relevant configuration objects, retrieved via the `get_downloader_cfg()` method of the `ConfigParser` class, to the constructor of the `DownloaderController`. Upon instantiation, the `DownloaderController` creates instances of platform-specific downloader classes based on the declared source of each item. The system supports two model hub platforms: Kaggle and Hugging Face (see Chapter 3). A corresponding downloader class is initialized for each platform and tasked with handling the download process.

The Hugging Face downloader relies on the `load_dataset()` function to retrieve datasets and the `hf_hub_download()` function to fetch pre-trained model files. These utilities are part of the Hugging Face `datasets` and `huggingface_hub` Python libraries, respectively, as described in Section 3.3. Similarly, the Kaggle downloader uses the `model_download()` and `dataset_download()` methods, as introduced in Section 3.2, to fetch content from Kaggle's model repository and dataset catalog. Once all specified resources have been downloaded, the resulting file paths are passed to the `EvaluatorController` instance.

6.1.4 Evaluation pipeline

The system proceeds with the core evaluation phase once all necessary resources have been downloaded and initialized. This process is coordinated by the `EvaluatorController` class, which prepares and executes evaluation pipelines. Each pipeline represents a collection of evaluation runs grouped by their associated embedded device. The pipeline is an execution unit that encapsulates all the logic required to run evaluations per device. Internally, it

[1]This approach assumes the *Docker CLI* is available and properly configured on the target embedded device.

communicates with the `DeviceInterface` instance corresponding to a device and issues the necessary REST calls defined by the Device API module.

Before executing the individual runs, the script queries the target device for system information via the *GET* `/fingerprint` request to the Device API – this includes the system name and the kernel version tied to the embedded device. Once the device metadata is retrieved, the pipeline executes each assigned run.

Model deployment and latency assessment

The script first uploads the selected model to the target device using the model path obtained during the download phase and issues a request to the Device API `/model/upload` endpoint. Next, latency evaluation is performed by invoking the `evaluate/latency/tflite` endpoint through the DeviceInterface; the returned response is parsed and stored for further use.

On-device inference

Following the latency evaluation, the system performs an accuracy evaluation. For this purpose, each supported computer vision task is handled by a dedicated evaluator class that inherits from the abstract `AccuracyEvaluator` base class. This base class defines the structure for all accuracy-related evaluations. It provides the necessary hooks for task-specific logic, such as dataset preprocessing, batching, user script integration (see Section 5.2), and metric computation.

Task-specific subclasses of `AccuracyEvaluator` implement preprocessing and batching routines tailored to their evaluation formats (e.g., resizing of the ground truth bounding boxes in case of the object detection tasks). Once the dataset is prepared, the system invokes the `DeviceInterface` to submit inference batches to the Device API module, which handles the inference. The resulting model outputs are then passed, along with the corresponding ground truth annotations, to the user-provided script for post-processing and alignment. These scripts transform raw model outputs into a normalized format required for metric computation.

Each user script is loaded dynamically as a Python module using the built-in `importlib` library. This approach allows the system to import and execute custom logic defined in external Python files at runtime. It enables seamless integration of task-specific post-processing routines without requiring hardcoded definitions in the core system. Details on the expected script interface and usage are provided in Appendix B.

Metric computation

Task-specific implementations of the `AccuracyEvaluator` base class also encapsulate the evaluation metrics suitable to the corresponding computer vision task. Each metric is represented by a class that inherits from the `Metric` base class, which defines a unified interface for metric computation. This interface includes a method for performing per-batch evaluations, allowing results to be computed as batches are inferred by the Device API module. Each task is associated with a predefined list of metric classes that are instantiated via a task-specific `AccuracyEvaluator` object. For image classification (see Section 4.1), the system evaluates a comprehensive set of metrics, including accuracy, precision, recall, F1-score, and specificity – each computed using macro and micro averaging strategies. For semantic segmentation (see Section 4.3), the defined metrics are mean intersection over

union and dice score with macro and micro averaging. The system implements intersection over union and mean average precision at a macro level for the object detection task (see Section 4.2).

The implementation of the `Metric` base class leverages the `torchmetrics` package. The `torchmetrics` package is a collection of over a hundred implementations of neural network metrics built on top of the *PyTorch* package [12]. The system uses a feature of the package that provides a means to continuously update the metric's value. After each inferred batch is processed and aligned via the user script, the task-specific `AccuracyEvaluator` instance invokes the configured metric objects to compute and accumulate results. The pipeline returns the final aggregated task-specific metrics after the last batch of the run.

Assessment aggregation

When the pipeline completes the run execution, it returns the computed metrics for its associated runs. The `EvaluatorController` (which controls the execution of the pipelines) then forwards these results to the *Report API* using a dedicated class responsible for interfacing with the API. Similar to the `DeviceInterface` class (see above), this communication is implemented using the `requests` library. The controller invokes the `/run` endpoint, sending a payload that includes the run metadata and the corresponding evaluation metrics. In response, the Report API returns a unique identifier for the registered run, which is stored for report generation.

6.1.5 Summary generation

Once all runs across all pipelines have been completed and successfully registered, the system invokes the `/report` endpoint of the Report API (implementation of the Report API is described in Section 6.3), providing the list of run identifiers. The API responds with a fully rendered HTML report page (see Figure 5.1), which is saved locally on the machine running the Model Tester module.

6.2 Device API

The Device API module is implemented in the `src/deviceApi` directory, which contains the `app` folder where the module's source code files are located. The `src/deviceApi` directory also contains Docker-related files (see below).

The Device API is implemented using the *FastAPI* framework, a web framework for building APIs with Python [3]. The FastAPI framework was selected for native support for asynchronous operations and integration with the `Pydantic` package (see Section 6.1) for schema validation. In alignment with the system requirements defined in Section 5.1, the Device API module currently supports only models in *TensorFlow Lite* format.

The Device API module is containerized using *Docker*. The Docker image is built as part of the system's deployment pipeline and uploaded to *Docker Hub*, from where it is pulled onto the embedded device during initialization (see Section 6.1 for Device API initialization details).

Once the container is deployed and running on the embedded device, the following endpoints are exposed and used by the system:

- GET /fingerprint – Returns metadata identifying the embedded device, including its system name and kernel version. The endpoint functionality is implemented using the

standard Linux system utility `uname`. This functionality is implemented as designed in Figure 5.6.

- POST /model/upload – Receives a neural network model file in the request payload and saves it to a predefined directory on the device's local storage (see Figure 5.7). The request sender must specify the name under which the model will be saved. The model is then available for subsequent requests.

- POST /evaluate/latency/tflite – Performs on-device latency assessment of the uploaded model. This endpoint executes the model using randomized dummy input tensors (the endpoint's return value and the implementation details are discussed below).

- POST /evaluate/accuracy/tflite – Executes inference on a batch of input samples using the model in the device's local storage (the implementation details are discussed below). The endpoint returns the outputs of the model.

In contrast to the Model Tester module, which is implemented with an object-oriented approach, the Device API module is implemented as a set of standalone service functions. This implementation decision aligns with the architecture of the FastAPI framework, defining endpoints as route-function pairs, where each endpoint directly maps to a dedicated function serving the corresponding request. The exception to this structure is the class responsible for performing on-device inference, further described below.

6.2.1 On-device latency evaluation

Latency measurement is served via the `POST /evaluate/latency/tflite` endpoint. The request payload is expected to contain the name of the model file as it is stored on the embedded device. Upon receiving the request, the Device API initiates latency assessment for the specified model using the official TensorFlow Lite benchmarking tool — `benchmark_model`.

The `benchmark_model` utility is a native command-line tool provided as part of TensorFlow Lite's runtime tooling. It is designed to evaluate the performance of TensorFlow Lite models in embedded and mobile environments [11]. The utility performs a sequence of executions under various states (e.g., cold start, warm up, and steady-state inference) and reports a set of performance metrics, including:

- Model initialization time,
- Warmup inference time,
- Steady-state inference time,
- Memory usage during initialization,
- Overall memory usage during inference.

The Device API invokes this tool as a subprocess, passing the corresponding model path. The raw terminal output of the benchmarking utility is returned directly in the response body. This design allows the client (the Model Tester module) to handle parsing, post-processing, and integration of the latency metrics into system logs and reports (see Section 6.1).

6.2.2 On-device accuracy evaluation

Inference is exposed through the `POST /evaluate/accuracy/tflite` endpoint. The request payload must include the name under which the model was previously uploaded to the device and input batch – a *NumPy* array serialized to raw bytes. On request receipt, the Device API module decodes the byte stream into a *NumPy* array and forwards the data to the `TfliteInterpreter` class, which encapsulates all on-device inference logic.

The `TfliteInterpreter` class, defined in `evaluate/tflite/interpreter.py`, encapsulates the complete inference workflow for *TensorFlow Lite* models. It builds on the `tflite_runtime` package – a standalone, lightweight distribution of the *TensorFlow Lite interpreter* that provides the same API as the full *TensorFlow* stack without its large dependency footprint [13]. The `TfliteInterpreter` class is implemented as a singleton to ensure consistent access to the TensorFlow Lite interpreter.

After input batch decoding, the Device API instantiates the `TfliteInterpreter` class using the absolute path of the requested model. During construction, the interpreter loads the model via the `tflite_runtime.Interpreter`. The `TfliteInterpreter` class method responsible for inference accepts a *NumPy* array representing an input batch, writes the data into the `tflite_runtime` package interpreter's input tensor(s), invokes the model, and finally reads back each output tensor.

The outputs are collected into a Python dictionary that maps every output tensor name (or index) to its corresponding *NumPy* array. The endpoint returns this dictionary unchanged, enabling the host-side Model Tester module to perform task-specific post-processing and metric computation (see Section 5.3 and Section 6.1).

6.3 Report API

The source code of the Report API module is located in the `src/reportApi` directory. The Python files implementing the module are present within the `app` folder. The source code also includes Docker-related files, enabling module containerization.

Similar to the Device API, the Report API module is implemented via *FastAPI* (see Section 6.2), chosen for its support for integrating relational databases via dependency injection, simplifying connection management to the system's *SQL store*, and its compatibility with `Jinja2` templating (see below), which the module uses to generate HTML reports. The Report API module implements the persistence of evaluation results via SQL database and manages HTML report generation, as proposed in Section 5.3.

Like the Device API module, the Report API module is also implemented as a set of standalone service functions. Each FastAPI endpoint maps directly to a corresponding function responsible for handling operations for storing results or generating reports.

6.3.1 Persistence

The Report API persists evaluation data in a single *SQLite*[2] file created on the same machine that runs the Report API module. All data access is performed through the `SQLAlchemy` package with the convenience wrapper `SQLModel`, which merges `SQLAlchemy` table metadata with `Pydantic` field validation [15]. Every entity shown in Figure 5.8 is realized as a Python class that inherits from the `SQLModel` base class; thus, the diagram's tables are mapped to typed objects in the code base.

[2] SQLite is a C-language library that implements a SQL database engine [14].

The module implements low-level database operations in the `database/services.py` file. This file implements a collection of helper functions that open scoped sessions, insert or delete rows, and run typed queries.

A new evaluation run is recorded through the `POST /run` request to the Report API. The request body must conform to the `RegisterRunInput` schema illustrated in Listing 6.1:

```
class RegisterRunInput(BaseModel):
    device:           DeviceBase
    model:            ModelBase
    latency_metrics:  list[MetricBase]
    dataset:          DatasetBase
    accuracy_metrics: list[MetricBase]
```

Listing 6.1: The input schema of the Report API /run endpoint

The nested objects in the schema above are SQL-mapped entities:

```
class DeviceBase(SQLModel):
    sys_name:        str = Field(index=True)
    kernel_version:  str = Field(index=True)

class ModelBase(SQLModel):
    framework: SupportModelFrameworks = Field(index=True)
    task:      SupportedModelTasks    = Field(index=True)
    path:      str                    = Field(index=True, unique=True)
    platform:  SupportedPlatforms     = Field(index=True)

class DatasetBase(SQLModel):
    path:     str                = Field(index=True, unique=True)
    platform: SupportedPlatforms = Field(index=True)

class MetricBase(SQLModel):
    name:  str  = Field(index=True)
    value: float
    unit:  str | None = None
```

Listing 6.2: Typed objects representing the database tables in the code base

Upon invocation, the Report API performs the following steps inside a single database session:

1. Insert device, model, and dataset into the database. Records matching the supplied fields that already exist are reused.
2. A new `Run` (see Figure 5.8) row is added, referencing the foreign-key IDs of the device, model, and dataset.
3. Each latency or accuracy metric in the payload is stored in the `Metric` table (see Figure 5.8) with a foreign-key link to the new run.
4. The transaction is committed, and the primary key ID of the run is returned to the caller.

6.3.2 HTML report generation

Beyond data persistence, the Report API exposes three endpoints that transform stored results into human-readable documents for analysis and regression tracking:

- GET /report – accepts a list of run identifiers (returned by the `/run` endpoint) and produces an HTML summary that lists devices, models, datasets, and all recorded metrics, following the layout defined in Figure 5.1.
- GET /compare/latency – returns a page for a side-by-side comparison of latency results for a chosen device–model pair; when the call is made without query parameters, the API responds with a page containing drop-down selectors for device, model, and the two kernel versions to be contrasted.
- GET /compare/accuracy – behaves analogously to the latency comparison endpoint, but the comparison additionally conditions the dataset; its output page follows the design illustrated in Figure 5.3.

All reporting endpoints rely on `Jinja2`, a templating engine that allows HTML files to embed placeholder expressions and control blocks [8]. Each endpoint selects the appropriate template from the `/html` directory, retrieves the required rows from the database through the service layer (see the section above), and invokes the `Jinja2` package `render()` method to inject the data. The rendered HTML is returned as the HTTP response.

Chapter 7

Evaluation of the implementation

This chapter presents a practical evaluation of the implemented system to demonstrate its ability to deliver evaluation results for embedded devices. A sample experiment was conducted to verify the system functionality from configuration parsing through model evaluation to report generation. The following paragraphs focus on the conducted experiment that was conducted. To conclude this chapter, Section 7.3 discusses the lessons learned during the system's implementation and the potential directions for future development.

7.1 Experimental environment

The host machine runs the core system script `modelTester.py` and the Report API module. The host was a laptop running *Windows 11*, equipped with *Python 3.11*. A clean virtual environment was created using the Python built-in `venv` module, and all required Python packages were installed based on the `requirements.txt` files provided in the thesis repository.

The embedded device used for neural network evaluation was an *NXP i.MX 8M Plus Evaluation Kit*. This platform is designed for rapid prototyping and benchmarking on embedded hardware. The exact configuration of the evaluation kit consists of i.MX 8MQuad processor with 4× Arm Cortex-A53 cores @ 1.5 GHz, 1 MB shared L2 cache, 1× Arm Cortex-M4 core @ 266 MHz, and 4 GB LPDDR4 RAM.

Two experiments were conducted using the same configuration file, with the only difference being the version of the *i.MX Linux operating system* installed on the embedded device. This Linux distribution is a kernel and root filesystem stack customized by NXP for embedded applications. The first experiment used the `6.12.3` version of the i.MX Linux, the second used, the `6.6.52` version. This variation aimed to validate the system's regression detection feature.

Both experiments used the same JSON configuration file located at `test/experiment.json` in the repository. A brief overview of its content follows:

- An embedded device URL (URL omitted from the JSON file in the repository for security reasons).
- Models: Three models, one per supported computer vision task:
  - `MobileNetv1`[1] for the image classification,

[1] Available at https://www.kaggle.com/models/tensorflow/mobilenet-v1/tfLite/

  - DeepLabv3[2] for semantic segmentation,
  - YOLOv8[3] for object detection.

- Datasets:
  - Mini-ImageNet[4] for image classification task,
  - PascalVOC2012[5] for semantic segmentation task,
  - Coco2017[6] for object detection task.

- Run tuples: Each run tuple binds a model to its respective dataset and a task-specific post-processing script. These scripts are located in the `test/scripts` directory and are responsible for post-processing the raw model outputs.

7.2 Executing the experiment

The experiment was executed in two phases: the first run utilized version `6.12.3` of the *i.MX Linux operating system kernel* on the embedded device, and the second run used version `6.6.52`. Both executions followed an identical procedure to ensure comparability.

Figure 7.1: The framework's command line output during the validation experiment.

The experiment was initiated from the host machine by executing the main orchestrator script with an argument specifying the path to the JSON configuration file. The script printed information regarding its execution to the command line, the output is depicted

[2] Available at https://www.kaggle.com/models/tensorflow/deeplabv3/tfLite.
[3] Available at https://docs.ultralytics.com/models/yolov8/.
[4] Available at https://www.kaggle.com/datasets/deeptrial/miniimagenet.
[5] Available at http://host.robots.ox.ac.uk/pascal/VOC/.
[6] Available at https://cocodataset.org.

in Figure 7.1. Upon execution, the script established an *SSH protocol* connection with the embedded device and automatically initialized the Device API module. Following device initialization, the script downloaded the datasets specified in the configuration file. The downloaded files were stored locally on the host machine. Next, for each model-dataset-script run tuple defined in the configuration:

1. The script invoked the Device API endpoint for latency evaluation.
2. The system transformed the dataset and iteratively sent each input batch to the Device API for inference.

After all run tuples were evaluated, the summary HTML document, containing all evaluated runs and their metrics, was generated and saved locally on the host machine.

System output

The system outputs produced during both experiment runs conform to the specifications outlined in the design phase (see Chapter 5). The core output is the HTML summary report generated by the Report API after completing all evaluation runs. This summary aggregates the performance results for each evaluated model-dataset pair – including latency and accuracy metrics.

The summary HTML document was saved locally on the host machine and can be opened directly in the browser. It includes: device and model metadata, detailed latency breakdown (initialization, warm-up, steady-state), and task-specific accuracy results for the evaluation dataset. Figure 7.2 shows a screenshot of the generated summary report.

In addition to the static report, the system provides an interactive comparison functionality via the Report API's `/compare/accuracy` and `/compare/latency` endpoints. These allow users to dynamically query results by model, device, and dataset via a web interface. The accuracy comparison endpoint supports selecting the device, model, dataset, and compared kernel versions. Upon selection, the endpoint returns a rendered accuracy comparison report. Figure 7.3 shows a screenshot of the rendered report comparing kernel versions included in the experiment. The latency comparison endpoint supports selecting the device, model, and compared kernel versions. Upon selection, the endpoint returns a rendered latency comparison report. Figure 7.4 displays the latency comparison view, where a regression in latency performance between kernel versions included in the experiment was observed — version `6.12.3` produced higher latency values across evaluated models as illustrated in Table 7.1.

Metric / Kernel	DeepLabv3		MobileNetv1		YOLOv8	
version	6.6.52	6.12.3	6.6.52	6.12.3	6.6.52	6.12.3
init (μs)	22484	23064	73286	73774	423884	423529
firstInference (μs)	268278	270298	119555	119575	1977208	1985096
warmup (μs)	267164	269680	119334	119487	1977210	1985100
inference (μs)	266446	268254	119194	119569	1976270	1982900

Table 7.1: Comparison of the latency metric values evaluated during the experiment.

Testing Results

Device:

Device Specifications

Fingerprint: imx8mpevk 6.6.52-lts-next-ge0f9e2afd4cf

Model Details

Task: image-classification

Framework: tflite

Platform: https://www.kaggle.com

Path: tensorflow/mobilenet-v1/tfLite/1-0-224-quantized

Latency Metrics

Metric	Value
init	73286.0
firstInference	119555.0
warmup	119334.0
inference	119194.0

Dataset

Platform: https://www.kaggle.com

Path: deeptrial/miniimagenet

Accuracy Metrics

Metric	Value
accuracy-macro	0.6804327964782715
accuracy-micro	0.6785348653793335
recall-macro	0.6804327964782715
recall-micro	0.6785348653793335
f1-macro	0.6664080023765564
f1-micro	0.6785348653793335
precision-macro	0.6994631886482239
precision-micro	0.6785348653793335
specificity-macro	0.9996781945228577
specificity-micro	0.9996785521507263

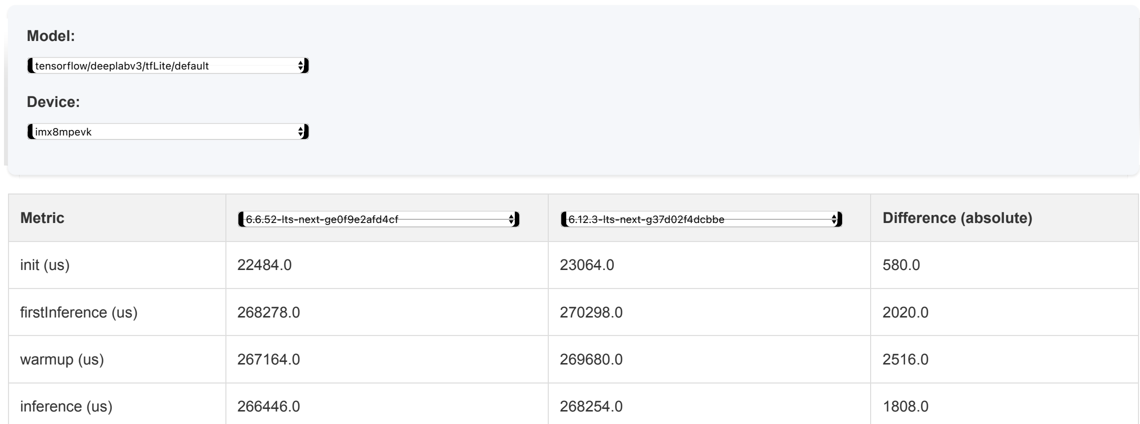
Figure 7.2: Top of the summary report for the validation experiment

Model accuracy comparison

Metric	6.6.52-lts-next-ge0f9e2afd4cf	6.12.3-lts-next-g37d02f4dcbbe	Difference (absolute)
intersection-over-union	0.1679810263234217	0.16792226975050903	5.875657291268266e-05
mean-average-precision-macro	0.02653934620320797	0.026547741144895554	8.394941687583923e-06

Figure 7.3: The accuracy comparison interface displaying results for the YOLOv8 model and the Coco2017 dataset.

Model latency comparison



Metric	6.6.52-lts-next-ge0f9e2afd4cf	6.12.3-lts-next-g37d02f4dcbbe	Difference (absolute)
init (us)	22484.0	23064.0	580.0
firstInference (us)	268278.0	270298.0	2020.0
warmup (us)	267164.0	269680.0	2516.0
inference (us)	266446.0	268254.0	1808.0

Figure 7.4: The latency comparison interface displaying results for the DeepLabv3 model.

All generated outputs are valid, complete, and aligned with the functional design of the system. Evaluation metrics were collected accurately, *REST* communication was reliably handled, and the report was generated without issues. Overall, the system behaves as proposed in Chapter 5 — all tested features were implemented correctly, and all outputs met the design and usability expectations.

7.3 Implementation retrospective

This section reflects on key challenges encountered during the implementation phase, the solutions developed in response, and insights gained throughout the process. While the implementation described in Chapter 6 largely followed the design proposed in Chapter 5, several practical difficulties emerged, especially around model output handling, metric computation efficiency, and design decisions that balanced system generality with user responsibility.

One of the most complex aspects of the system was parsing model outputs and aligning them with ground truth labels. Each model architecture outputs results in a unique format; some models are quantized, some omit post-processing layers, and others follow differing conventions in labeling. In parallel, datasets vary widely in their structure and labeling schemes. This mismatch between model outputs and dataset ground truths made implementing a universal accuracy evaluation mechanism complicated. Although the goal was to make the system capable of evaluating any model-dataset combination, it became clear that models can produce outputs incompatible with specific datasets, such as classification models with class indices shifted by one or object detection with different-sized bounding boxes ground truth. This challenge showed that delegating the responsibility of interpreting and aligning outputs to the user (via custom post-processing scripts) was a more scalable and robust solution.

Another important lesson concerned metric computation. Initially, accuracy metrics were computed manually within the system. However, with larger datasets, this approach consumed significant memory and processing time. Upon investigating alternatives, the `torchmetrics` library was discovered to provide a much more efficient solution, particularly due to its support for per-batch metric computation, which allowed the system to scale better while reducing resource consumption. Integrating `torchmetrics` into the system proved to be both practical and elegant.

Despite the challenges, some implementation decisions were especially effective, particularly integrating the `Pydantic` package. `Pydantic` enabled seamless JSON configuration parsing, simplified the definition of SQL database models via the `SQLModel` package, and facilitated robust request/response schema validation in the FastAPI *REST* APIs. Its declarative syntax, built-in validation, and compatibility with type hints significantly accelerated development.

Opportunities for future work The implemented system could be extended in several directions to improve performance, scalability, and generality. One enhancement would be the introduction of pipelined execution. In the current implementation, batches are sent to the embedded device in full, and evaluation, including inference and metric computation, happens sequentially. A pipelined approach would stream images to the embedded device one by one. As each image is processed and the inference result is returned, metric computation could begin immediately on the Model Tester module while the next image is sent to the Device API. Pipelined execution would reduce overall evaluation time, especially for large datasets, and ease memory pressure on constrained embedded hardware.

Another area for potential expansion is introducing an API layer for the Model Tester module. The API layer would enable the tool's deployment as a centralized service, for example, on a server managing a board farm, where users could submit jobs remotely and track their progress and results via a web interface. Such an architecture would support broader integration into continuous integration (CI) pipelines.

An example approach for CI pipelines would be to use tools such as Bamboo[7]. Incorporating the evaluation framework into a CI workflow creates the possibility to automatically test and validate deep neural network models in response to changes in models, Linux kernel versions, or system drivers. For example, a Bamboo job could trigger the implemented framework, run the necessary tests on connected embedded devices, and collect accuracy and latency metrics when a new system image is built. HTML-based reports and raw

[7] „Bamboo is a continuous integration (CI) server that can be used to automate the release management for a software application, creating a continuous delivery pipeline.“ [16]

metrics could be published as CI artifacts, aiding in the early detection of performance regressions.

Lastly, the current implementation supports only the TensorFlow Lite framework. Extending the system to support other embedded inference runtimes, such as *ONNX Runtime, ExecuTorch,* or *Ethos-U*, would significantly increase its applicability across different hardware ecosystems and enable more comprehensive benchmarking and analysis.

Chapter 8

Conclusion

This thesis aimed to create a system for automated testing and evaluation of deep neural networks on embedded devices. Studying deep neural networks and their applications in computer vision was necessary to achieve this goal. It was also necessary to examine model hubs and interfaces to retrieve items from them. The literature review part of the thesis concludes with the metrics for assessing the accuracy and latency of deep neural networks in computer vision.

With reviewed knowledge, designing and implementing the testing framework with support for model hubs and regression detection was possible. The system requirements were specified, and the thesis introduced the system's design and the format of the system's inputs and outputs. The system design consists of three main modules – an API responsible for communication with the embedded device, an API for data persistence and generation of output reports, and the orchestrator module, which coordinates the workflow and manages task execution. The thesis described the implementation of the main modules of the system's architecture and documented the experiments conducted to validate its functionality.

An experiment was conducted to validate the system, using representative model for each supported computer vision task. The results demonstrated that the implementation functions as designed, confirming the validity and reliability of the proposed framework. During testing, a regression was detected between official releases of NXP's Linux for embedded platforms, demonstrating the framework's effectiveness in identifying performance and accuracy issues introduced by software updates.

The implemented system meets all system requirements outlined in Section 5.1. The system supports automated retrieval of models and datasets from remote sources – specifically, Kaggle and Hugging Face – through dedicated interface implementations. It evaluates models in the TensorFlow Lite format, sticking to the constraints of the NXP deployment environment. Evaluation supports neural networks performing image classification, object detection, and semantic segmentation, each supported with task-specific accuracy metrics and evaluation logic. The deployment on embedded Linux-based platforms is handled seamlessly via the SSH protocol and the Device API module running on the embedded platform. During evaluation, the system measures and records latency and accuracy metrics for each model. The regression detection feature is implemented through comparative interfaces that display the data the system recorded. The system generates detailed HTML summaries of executed experiments.

The thesis concludes with insights on what was difficult and what could have been done better, and suggests what direction the development of this tool could take. The

most complex aspect of the system was the post-processing of model outputs and aligning them with the corresponding ground truth data. This challenge was addressed through a user-script architecture, where the user provides custom scripts for parsing and interpreting model outputs according to model-dataset-specific needs. Looking ahead, one of the most valuable extensions to the tool would be the implementation of execution pipelining, which would significantly reduce the time and computational resources.

Bibliography

[1] *2D Convolution as a Matrix-Matrix Multiplication | Baeldung on Computer Science* online. Available at: https://www.baeldung.com/cs/convolution-matrix-multiplication. [cit. 2024-10-28].

[2] *AsyncSSH: Asynchronous SSH for Python* online. Available at: https://asyncssh.readthedocs.io/en/latest/. [cit. 2025-04-22].

[3] *FastAPI* online. Available at: https://fastapi.tiangolo.com. [cit. 2025-04-23].

[4] *Git* online. Available at: https://git-scm.com/. [cit. 2024-10-16].

[5] *Git Large File storage* online. Available at: https://git-lfs.com/. [cit. 2024-10-16].

[6] *Hugging Face - Documentation* online. Available at: https://huggingface.co/docs/hub/index. [cit. 2024-10-13].

[7] *Hugging Face – The AI community building the future.* online. Available at: https://huggingface.co. [cit. 2024-10-13].

[8] *Jinja Documentation* online. Available at: https://jinja.palletsprojects.com/en/stable/. [cit. 2025-04-23].

[9] *Kaggle* online. Available at: https://www.kaggle.com/. [cit. 2024-10-11].

[10] *LiteRT overview* online. Available at: https://ai.google.dev/edge/litert. [cit. 2025-04-23].

[11] *Performance measurement | Google AI Edge* online. Available at: https://ai.google.dev/edge/litert/models/measurement. [cit. 2025-04-23].

[12] *PyTorch-Metrics documentation* online. Available at: https://lightning.ai/docs/torchmetrics/stable/. [cit. 2025-04-22].

[13] *Quickstart for Linux-based devices with Python* online. Available at: https://ai.google.dev/edge/litert/microcontrollers/python. [cit. 2025-04-23].

[14] *SQLite Home Page* online. Available at: https://sqlite.org. [cit. 2025-04-23].

[15] *SQLModel* online. Available at: https://sqlmodel.tiangolo.com. [cit. 2025-04-23].

[16] *Understanding the Bamboo CI Server* online. Available at: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html. [cit. 2025-05-18].

[17] *What are convolutional neural networks? | IBM* online. Available at: https://www.ibm.com/topics/convolutional-neural-networks. [cit. 2024-10-13].

[18] A.D.Dongare; R.R.Kharde and D.Kachare, A. Introduction to Artificial Neural Network. In:. 2012. Available at: https://api.semanticscholar.org/CorpusID:212457035.

[19] Bishop, C. M. Neural networks and their applications. *Review of Scientific Instruments*, june 1994, vol. 65, no. 6, p. 1803–1832. ISSN 0034-6748. Available at: https://doi.org/10.1063/1.1144830.

[20] Erickson, B. J. and Kitamura, F. Magician's Corner: 9. Performance Metrics for Machine Learning Models. *Radiology: Artificial Intelligence*, 2021, vol. 3, no. 3, p. e200126. Available at: https://doi.org/10.1148/ryai.2021200126.

[21] Goodfellow, I.; Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[22] Haralick, R. M. and Shapiro, L. G. Glossary of computer vision terms. *Pattern Recognition*, 1991, vol. 24, no. 1, p. 69–93. ISSN 0031-3203. Available at: https://www.sciencedirect.com/science/article/pii/003132039190117N.

[23] Lardinois, F.; Lynley, M. and Mannes, J. *Google is acquiring data science community Kaggle* online. Tech Crunch, march 2017. Available at: https://techcrunch.com/2017/03/07/google-is-acquiring-data-science-community-kaggle/. [cit. 2024-10-11].

[24] LeeYo Seob. Analysis of AI Model Hub. *International Journal of Advanced Culture Technology*, december 2023, vol. 11, no. 4, p. 442–448.

[25] Lutkevich, B. *What is Hugging Face?* online. TechTarget, september 2023. Available at: https://www.techtarget.com/whatis/definition/Hugging-Face#:~:text=Hugging%20Face%20Inc.%20is%20the,Julien%20Chaumond%20and%20Thomas%20Wolf. [cit. 2024-10-13].

[26] Minfei, L.; Gan, Y.; Chang, Z.; Wan, Z.; Schlangen, E. et al. Microstructure-informed deep convolutional neural network for predicting short-term creep modulus of cement paste. *Cement and Concrete Research*, february 2022, vol. 152.

[27] Müller, D.; Soto Rey, I. and Kramer, F. *Towards a Guideline for Evaluation Metrics in Medical Image Segmentation*. 2022. Available at: https://arxiv.org/abs/2202.05273.

[28] O'Shea, K. and Nash, R. An Introduction to Convolutional Neural Networks. *CoRR*, 2015, abs/1511.08458. Available at: http://arxiv.org/abs/1511.08458.

[29] Padilla, R.; Passos, W. L.; Dias, T. L. B.; Netto, S. L. and Silva, E. A. B. da. A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit. *Electronics*, 2021, vol. 10, no. 3. ISSN 2079-9292. Available at: https://www.mdpi.com/2079-9292/10/3/279.

[30] Quaranta, L.; Calefato, F. and Lanubile, F. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021, p. 550–554.

[31] Saladas, J. *What is cURL and how does it relate to APIs?* online. IBM, february 2024. Available at: https://developer.ibm.com/articles/what-is-curl-command/. [cit. 2024-10-12].

[32] Sharma, S.; Sharma, S. and Athaiya, A. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, may 2020, vol. 04, p. 310–316.

[33] Voulodimos, A.; Doulamis, N.; Doulamis, A. and Protopapadakis, E. Deep Learning for Computer Vision: A Brief Review. *Computational Intelligence and Neuroscience*, 2018, vol. 2018, no. 1. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1155/2018/7068349.

[34] Xiu, M.; Ming, Z.; Jiang and Adams, B. *An Exploratory Study on Machine Learning Model Stores*. 2020. Available at: https://arxiv.org/abs/1905.10677.

Appendix A

Codebase folder structure

The submitted archive contains a compressed file named `system.zip` representing the codebase of the system implemented in this thesis. The codebase includes the following files and directories:

- `README.md` – Instructions for installing and running the system.
- `USER_SCRIPT.md` – Description of the structure and content of user-scripts required to run the system. Appendix B further elaborates on the file content.
- `src/` – Contains the source code and supporting files for the main system modules:
  - `deviceApi/` – Source files for the *Device API* module, including Docker configuration files and dependency definitions.
  - `reportApi/` – Source files for the *Report API* module and its dependencies.
  - `modelTester/` – Directory containing the `modelTester.py` orchestrator script, source files for the *Model Tester* module, and associated dependencies.
- `test/` – Files used during the validation experiment (see Chapter 7):
  - `experiment.json` – A configuration file used in the experiment.
  - `scripts/` – A collection of example user-scripts employed during the experiment.

In addition, the submitted archive also contains the text of the thesis in PDF `thesis.pdf` and the `thesis.zip` archive, which includes the source files necessary to compile the PDF.

Appendix B

User script specification

The system incorporates user-defined post-processing scripts to support flexibility in evaluating various computer vision models and datasets. These scripts transform the raw model outputs into a normalized format suitable for metric computation and align them with the corresponding ground truth annotations. Each script is defined for a specific model and dataset pair, and its file path is specified within the experiment configuration file. The scripts must adhere to a predefined interface to ensure compatibility with the evaluation pipeline.

Required Function

All user scripts must implement the `align_target_predicted(predictions, targets)` function, where:

- `predictions`: A list of dictionaries (list[dict[int, np.NDArray]]), representing the model's raw outputs for each input in the batch.
- `targets`: A list containing the ground truth annotations associated with the batch. These are extracted from the test dataset based on the key provided in the configuration file.
- Return value: A tuple, whose content and structure are task-specific (described below).

For image classification tasks:

- `targets`: A list of class labels corresponding to the ground truth annotations.
- Return format:
  1. `predictions`: A NumPy array containing predicted class indices (np.NDArray[np.Number]).
  2. `targets`: A NumPy array of ground truth labels aligned with the predictions.
  3. `num_classes`: An integer indicating the total number of classification classes.

For semantic segmentation tasks:

- `targets`: A list of segmentation masks, where each mask is represented as a NumPy array.
- Return format:

1. `predictions`: A NumPy array of predicted segmentation outputs (np.NDArray[np.NDArray]).
2. `targets`: A NumPy array of corresponding ground truth masks.
3. `num_classes`: An integer specifying the total number of segmentation classes.
4. `input_format`: A string specifying the encoding format of the segmentation masks. Accepted values are:
   - „one-hot“: One-hot encoded tensor.
   - „index“: Tensor containing class indices.

For object detection tasks:

- `targets`: A list of tuples, where each tuple consists of ground truth bounding boxes and corresponding class labels, represented as NumPy arrays.
- Return format:
  1. predictions: A list of dictionaries, where each dictionary must include the following keys:
     - `boxes`: List of predicted bounding boxes.
     - `scores`: List of confidence scores.
     - `labels`: List of predicted class labels.
  2. targets: Ground truth tuples aligned with the predicted outputs.
  3. `box_format`: A string specifying the format of the bounding boxes. Supported formats include:
     - `xyxy`: Top-left and bottom-right coordinates (x1, y1, x2, y2).
     - `xywh`: Top-left coordinates with width and height (x1, y1, w, h).
     - `cxcywh`: Center coordinates with width and height (cx, cy, w, h).