Antmicro

Kenning

2025-05-30

CONTENTS

1 Introduction **1**

2 Kenning **2**
- 2.1 Introduction . . . 3
- 2.2 Kenning installation . . . 4
- 2.3 Kenning structure . . . 5
- 2.4 Kenning usage . . . 7
- 2.5 Using Kenning as a library in Python scripts . . . 12
- 2.6 Adding new implementations . . . 13

3 Deep Learning deployment stack **15**
- 3.1 From training to deployment . . . 15
- 3.2 Dataset preparation . . . 15
- 3.3 Model preparation and training . . . 16
- 3.4 Model optimization . . . 17
- 3.5 Model compilation and deployment . . . 17

4 Defining optimization pipelines in Kenning **18**
- 4.1 JSON specification . . . 18
- 4.2 Model evaluation using its native framework . . . 19
- 4.3 Model training . . . 21
- 4.4 Optimizing and running a model on a single device . . . 22
- 4.5 Compiling a model and running it remotely . . . 24

5 Using Kenning via command line arguments **27**
- 5.1 Command-line arguments for classes . . . 27
- 5.2 Autocompletion for command line interface . . . 27
- 5.3 Model training . . . 28
- 5.4 In-framework inference performance measurements . . . 30
- 5.5 Testing inference on target hardware . . . 30
- 5.6 Running inference . . . 35
- 5.7 Generating performance reports . . . 37
- 5.8 Displaying information about available classes . . . 38

6 Kenning gallery of use cases **39**
- 6.1 Generating anomaly detection models for the MAX32690 Evaluation Kit with AutoML . . . 39
- 6.2 Anomaly detection model training and deployment on the MAX32690 Evaluation Kit . . . 45

6.3 Displaying information about available classes . . . 54
6.4 Evaluating models on hardware using Kenning Zephyr Runtime . . . 59
6.5 Visualizing Kenning data flows with Pipeline Manager . . . 62
6.6 Bare-metal IREE runtime simulated using Renode . . . 64
6.7 Structured pruning for PyTorch models . . . 67
6.8 Model quantization and compilation using TFLite and TVM . . . 71
6.9 Unstructured Pruning of TensorFlow Models . . . 82

7 Kenning environment variables 89
7.1 KENNING_CACHE_DIR . . . 89
7.2 KENNING_MAX_CACHE_SIZE . . . 89
7.3 KENNING_USE_DEFAULT_EXCEPTHOOK . . . 89
7.4 KENNING_DOCS_VENV . . . 89
7.5 KENNING_ENABLE_ALL_LOGS . . . 90
7.6 KENNING_DISABLE_IO_VALIDATION . . . 90

8 Kenning measurements 91
8.1 Performance metrics . . . 91

9 Choosing optimal optimization pipeline 92
9.1 Optimization config specification . . . 92
9.2 Output details . . . 94

10 Sample autogenerated report 96
10.1 Pet Dataset classification using TVM-compiled TensorFlow model . . . 96
10.2 Inference performance metrics for build.local-cpu-tvm-tensorflow-classification.json . . . 99
10.3 Inference quality metrics for build.local-cpu-tvm-tensorflow-classification.json . . . 100

11 Creating applications with Kenning 103
11.1 JSON structure . . . 103
11.2 KenningFlow execution . . . 106
11.3 Implemented Runners . . . 106

12 Developing Kenning blocks 107
12.1 Model and I/O metadata . . . 107
12.2 Implementing a new Kenning component . . . 110
12.3 Implementing Kenning runtime blocks . . . 120
12.4 Adjusting ModelWrapper for AutoML flow . . . 121

13 Kenning resources 127
13.1 Accessing resources . . . 127
13.2 Caching . . . 128
13.3 Custom URI schemes . . . 128
13.4 CLI commands . . . 130

14 Kenning platforms 131
14.1 Specification . . . 131
14.2 Generating definitions for boards supported by Zephyr RTOS . . . 134

15 Kenning API 136
15.1 Deployment API overview . . . 136
15.2 KenningFlow . . . 138

15.3 Runner . . . 139
15.4 Dataset . . . 141
15.5 ModelWrapper . . . 148
15.6 Optimizer . . . 154
15.7 Runtime . . . 158
15.8 Platform . . . 164
15.9 AutoML . . . 165
15.10 Protocol . . . 168
15.11 DataConverter . . . 176
15.12 Measurements . . . 177
15.13 ONNXConversion . . . 181
15.14 DataProvider . . . 183
15.15 OutputCollector . . . 184
15.16 ArgumentsHandler . . . 185
15.17 ResourceManager . . . 186
15.18 ResourceURI . . . 188

Python Module Index **190**

Index **191**

CHAPTER

ONE

INTRODUCTION

Kenning provides an API for deploying deep learning applications on edge devices using various model training and compilation frameworks.

This documentation consists of the following chapters:

- *Kenning* section provides a project description, installation steps and a quick start guide,
- *Deep Learning deployment stack* section describes a typical model deployment flow on edge devices,
- *Defining optimization pipelines in Kenning* section describes a way to create advanced optimization scenarios with JSON config,
- *Using Kenning via command-line arguments* section describes executable scripts available in Kenning,
- *Kenning gallery* contains use cases with models optimization and integration with ROS2, Renode and Pipeline Manager,
- *Kenning environment variables* section describes specific variables which can influence how the Kenning works,
- *Kenning measurements* section describes data gathered during the compilation and evaluation process,
- *Choosing optimal optimization pipeline* section describes a way to run multiple pipelines and search for the one that achieves the best result
- *Sample autogenerated report* section provides a sample report generated using Kenning,
- *Creating applications with Kenning* section describes KenningFlow and its usage,
- *Developing Kenning blocks* section describes a way to develop new Kenning components,
- *Kenning resources* section describes how Kenning deals with its resources (pretrained models, datasets etc.),
- *Kenning API* section provides an in-depth description of the Kenning API.

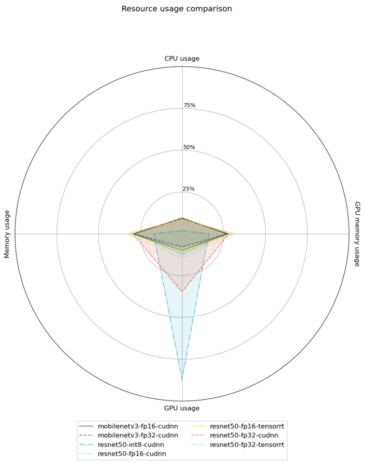
CHAPTER

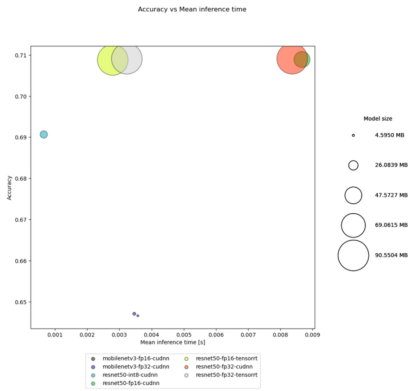
TWO

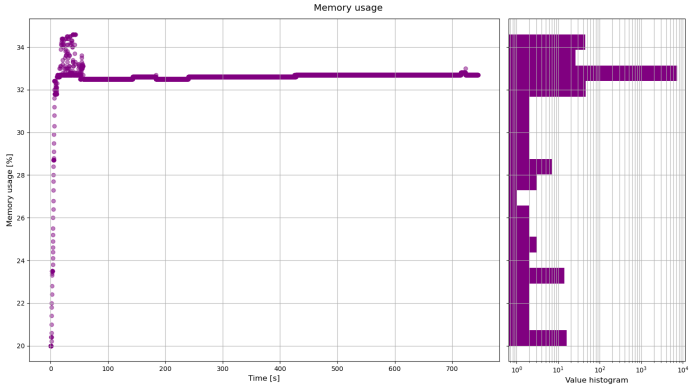
KENNING

Kenning is a framework for creating deployment flows and runtimes for Deep Neural Network applications on various target hardware.

Kenning documentation | Core API | kenning.ai | Tutorials







Contents:

- *Introduction*
- *Kenning installation*
- *Kenning structure*
- *Kenning usage*
- *Using Kenning as a library in Python scripts*
- *Adding new implementations*

2.1 Introduction

Kenning aims towards providing modular execution blocks for:

- dataset management,
- model training,
- model optimization and compilation for a given target hardware,
- running models using efficient runtimes on target device,
- model evaluation and performance reports.

These can be used seamlessly regardless of underlying frameworks for the above-mentioned steps.

Kenning's aim is not to bring yet another training or compilation framework for deep learning models - there are lots of mature and versatile frameworks that support certain models, training routines, optimization techniques, hardware platforms and other components crucial to the deployment flow. Still, there is no framework that would support all of the models or target hardware devices - especially the support matrix between compilation frameworks and target hardware is extremely sparse. This means that any change in the application, especially in hardware, may end up in a necessity to change the entirety or a significant part of the application flow.

Kenning addresses this issue by providing a unified API that focuses on deployment tasks rather than their implementation - the developer decides which implementation should be used for each task, and with Kenning, it is possible to do in a seamless way. This way, switching to another target platform results, in most cases, in a very small change in the code, instead of reimplementing larger parts of a project. This is how Kenning can get the most out of the existing Deep Neural Network training and compilation frameworks.

Seamless nature of Kenning also allows developers to quickly evaluate the model on various stages of optimizations and compare them as shown in *Kenning usage*.

2.2 Kenning installation

2.2.1 Module installation with pip

NOTE: Kenning supports Python up to 3.11.

To install Kenning with its basic dependencies with `pip`, run:

```
pip install -U git+https://github.com/antmicro/kenning.git
```

Since Kenning can support various frameworks, and not all of them are required for users' particular use cases, some of the requirements are optional. We can distinguish the following groups of extra requirements:

- `tensorflow` - modules for work with TensorFlow models (ONNX conversions, addons, and TensorFlow framework),
- `torch` - modules for work with PyTorch models,
- `mxnet` - modules for work with MXNet models,
- `nvidia_perf` - modules for performance measurements for NVIDIA GPUs,
- `object_detection` - modules for work with YOLOv3 object detection and the Open Images Dataset V6 computer vision dataset,
- `speech_to_text` - modules for work with audio samples and speech-to-text models,
- `iree` - modules for IREE compilation and runtime,
- `tvm` - modules for Apache TVM compilation and runtime,
- `onnxruntime` - modules for ONNX Runtime,
- `nni` - modules for Neural Network Intelligence optimizers,
- `docs` - modules for generating documentation,
- `test` - modules for Kenning framework testing,
- `real_time_visualization` - modules for real time visualization runners,
- `pipeline_manager` - modules for communication with visual editor,
- `reports` - modules for generating reports and comparisons,
- `uart` - modules for work with serial ports,
- `renode` - modules for work with Renode,
- `fuzzy` - modules for fuzzy search.

To install the extra requirements, e.g. `tensorflow`, run:

```
sudo pip install git+https://github.com/antmicro/kenning.git
↪#egg=kenning[tensorflow]
```

or, in newer `pip` releases:

```
pip install "kenning[tensorflow] @ git+https://github.com/antmicro/kenning.git"
```

2.2.2 Working directly with the repository

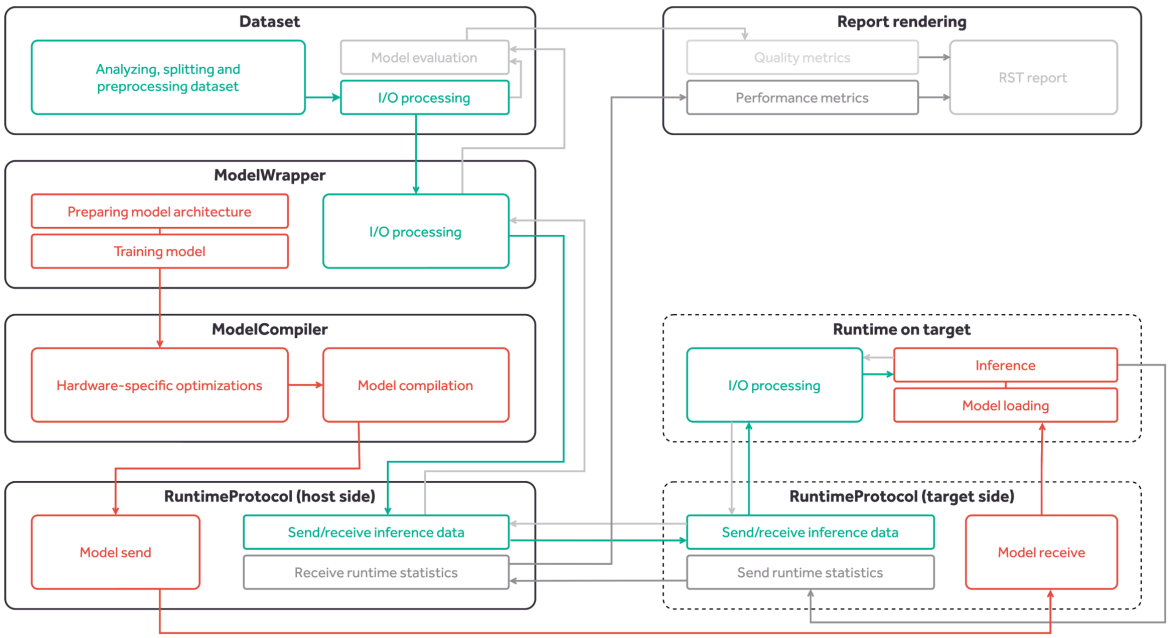
For development purposes, and to use additional resources (such as sample scripts), clone the repository with:

```
git clone https://github.com/antmicro/kenning.git
cd kenning/
```

Then install using:

```
pip install -e ".[tensorflow,tflite,tvm,reports]"
```

2.3 Kenning structure



The `kenning` module consists of the following submodules:

- `core` - provides interface APIs for datasets, models, optimizers, runtimes, protocols, and data converters
- `datasets` - provides implementations for datasets
- `modelwrappers` - provides implementations for models for various problems implemented in various frameworks,
- `optimizers` - provides implementations for compilers and optimizers for deep learning models,
- `runtimes` - provides implementations of runtime on target devices,
- `interfaces` - provides interface classes to group related methods used in Kenning `core` classes,
- `protocols` - provides implementations for communication protocols between host and tested target,

- `dataproviders` - provides implementations for reading input data from various sources, such as camera, directories or TCP connections,
- `dataconverters` - provides implementations for data converters for various data types
- `outputcollectors` - provides implementations for processing outputs from models, i.e. saving results to file, or displaying predictions on screen.
- `onnxconverters` - provides ONNX conversions for a given framework along with a list of models to test the conversion on,
- `runners` - provide implementations for runners that can be used in runtime,
- `drawing` - provides methods for rendering plots for reports,
- `resources` - contains project's resources, like RST templates, or trained models,
- `scenarios` - contains executable scripts for running training, inference, benchmarks and other tests on target devices,
- `utils` - various functions and classes used in all above-mentioned submodules,
- `tests` - submodules for framework testing,
- `pipeline_manager` - contains tools for integrating with Pipeline Manager visualizer
- `cli` - provides tools and methods for creating CLI tools based on Kenning

`core` classes used throughout the entire Kenning framework:

- `Dataset` class - performs dataset download, preparation, dataset-specific input preprocessing (i.e. input file opening, normalization), output postprocessing and model evaluation,
- `ModelWrapper` class - trains the model, prepares the model, performs model-specific input preprocessing and output postprocessing, runs inference on host using native framework,
- `Optimizer` class - optimizes and compiles the model,
- `Runtime` class - loads the model, performs inference on compiled model, runs target-specific processing of inputs and outputs, and runs performance benchmarks,
- `Protocol` class - implements the communication protocol between the host and the target,
- `DataProvider` class - implements data provision from such sources as camera, TCP connection or others for inference,
- `OutputCollector` class - implements parsing and utilization of data from inference (such as displaying the visualizations, sending the results to via TCP),
- `DataConverter` class - performs data conversion from dataset-specific format to protocol-specific format and vice versa,
- `Runner` class - represents single runtime processing block.

2.4 Kenning usage

There are several ways to use Kenning:

- Using executable scripts from the `scenarios` submodule, configurable via JSON files (recommended approach);
- Using executable scripts from the `scenarios` submodule, configurable via command-line arguments;
- Using Kenning as a Python module.

Kenning scenarios are executable scripts that can be used for:

- Model training and benchmarking using its native framework (`kenning.scenarios.model_training`),
- Model optimization and compilation for target hardware (`kenning.scenarios.inference_tester`),
- Model benchmarking on target hardware (`kenning.scenarios.inference_tester` and `kenning.scenarios.inference_server`),
- Rendering performance and quality reports from benchmark data (`kenning.scenarios.render_report`),
- and more.

They are available through the `kenning` executable as subcommands. To get the current list of subcommands, run:

```
kenning -h
```

The available subcommands are:

- `train` - trains the given model (`kenning.scenarios.model_training`).
- `optimize` - optimizes and compiles the model for a given target device (`kenning.scenarios.inference_tester`).
- `test` - runs benchmark and evaluation of the model on the target device (`kenning.scenarios.inference_tester`).
- `report` - creates Markdown and HTML files summarizing the quality of the model in terms of performance and predictions.
- `flow` - runs Kenning-based applications.
- `visual-editor` - runs a graphical interface letting you represent, edit and run optimizations and applications in-browser using Pipeline Manager.
- `fine-tune-optimizers` - runs a search for the best optimizations for a given target platform based on selected optimizers, runtimes and models, along with their settings.
- `server` - runs a benchmark and evaluation server on target device.
- `info` - provides information about a given Kenning class.
- `list` - lists available Kenning modules for optimization and runtime.
- `fuzzy-search` - searches for class path across Kenning and returns it.

- `cache` - manages Kenning cache used for models and datasets.
- `completion` - configures autocompletion feature for Kenning CLI.

2.4.1 Running Kenning

Let's start off with installing the module and its necessary dependencies:

```
pip install "kenning[tensorflow,tflite,tvm,pipeline_manager,reports] @ git+https:/
↪/github.com/antmicro/kenning.git"
```

Kenning provides two tools for displaying information about available classes: `kenning list` and `kenning info`.

The `kenning list` tool lists all available modules used to form optimization and runtime pipelines in Kenning. Running:

```
kenning list
```

Should list:

```
Optimizers (in kenning.optimizers):

    kenning.optimizers.nni_pruning.NNIPruningOptimizer
    kenning.optimizers.onnx.ONNXCompiler
    kenning.optimizers.tensorflow_pruning.TensorFlowPruningOptimizer
    kenning.optimizers.model_inserter.ModelInserter
    kenning.optimizers.tvm.TVMCompiler
    kenning.optimizers.iree.IREECompiler
    kenning.optimizers.tensorflow_clustering.TensorFlowClusteringOptimizer
    kenning.optimizers.tflite.TFLiteCompiler

Datasets (in kenning.datasets):

    kenning.datasets.pet_dataset.PetDataset
    kenning.datasets.visual_wake_words_dataset.VisualWakeWordsDataset
    kenning.datasets.random_dataset.RandomizedDetectionSegmentationDataset
    kenning.datasets.open_images_dataset.OpenImagesDatasetV6
    kenning.datasets.random_dataset.RandomizedClassificationDataset
    kenning.datasets.common_voice_dataset.CommonVoiceDataset
    kenning.datasets.magic_wand_dataset.MagicWandDataset
    kenning.datasets.coco_dataset.COCODataset2017
    kenning.datasets.imagenet_dataset.ImageNetDataset

Modelwrappers (in kenning.modelwrappers):

    kenning.modelwrappers.instance_segmentation.yolact.YOLACT
    kenning.modelwrappers.classification.tflite_magic_wand.MagicWandModelWrapper
    kenning.modelwrappers.classification.pytorch_pet_dataset.
↪PyTorchPetDatasetMobileNetV2
    kenning.modelwrappers.object_detection.darknet_coco.TVMDarknetCOCOYOLOV3
    kenning.modelwrappers.instance_segmentation.yolact.YOLACTWithPostprocessing
```

(continues on next page)

(continued from previous page)

```
    kenning.modelwrappers.classification.tensorflow_imagenet.TensorFlowImageNet
    kenning.modelwrappers.classification.tflite_person_detection.
↪PersonDetectionModelWrapper
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2
    kenning.modelwrappers.instance_segmentation.pytorch_coco.PyTorchCOCOMaskRCNN
    kenning.modelwrappers.object_detection.yolov4.ONNXYOLOV4

...
```

To list available compilers, run:

```
kenning list optimizers
```

For more verbose information, use the `-v` and `-vv` flags (module dependencies, description, supported formats and more)

The `kenning info` tool provides more detailed information on a given class, e.g. for:

```
kenning info kenning.optimizers.tflite.TFLiteCompiler
```

We should get something like:

```
Class: TFLiteCompiler

    The TFLite and EdgeTPU compiler.

Dependencies:
* onnx_tf.backend.prepare
* tensorflow
* numpy
* onnx
* tensorflow_model_optimization

Input formats:
* keras
* tensorflow
* onnx


Output formats:
* tflite


Arguments specification:
* model_framework
  * argparse_name: --model-framework
  * description: The input type of the model, framework-wise
  * default: onnx
  * enum
    * keras
```

(continues on next page)

(continued from previous page)

```
    * tensorflow
    * onnx
* target
  * argparse_name: --target
  * description: The TFLite target device scenario
  * default: default
  * enum
    * default
    * int8
    * float16
    * edgetpu
* inference_input_type
  * argparse_name: --inference-input-type
  * description: Data type of the input layer
  * default: float32
  * enum
    * float32
    * int8
    * uint8
...
```

If a certain class does not have necessary dependencies installed, Kenning suggests what modules need to be installed, e.g.:

```
This method requires additional dependencies, please use `pip install
↪"kenning[tensorflow]"` to install them.
```

The classes described above are used to form optimization and evaluation scenarios in Kenning. Some of them can also be used to quickly prototype simple applications.

With Kenning, we can combine multiple compilers and optimizers to create a fast and small model, including embedded devices. The optimization flow can be defined in JSON format, as follows:

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_name": "tvm-avx2-int8",
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "optimizers": [
    {
      "type": "kenning.optimizers.tflite.TFLiteCompiler",
      "parameters": {
        "target": "int8",
        "compiled_model_path": "./build/int8.tflite",
        "inference_input_type": "int8",
        "inference_output_type": "int8",
        "dataset_percentage": 0.01
      }
    },
    {
      "type": "kenning.optimizers.tvm.TVMCompiler",
      "parameters": {
        "target": "llvm -mcpu=core-avx2",
        "conv2d_data_layout": "NCHW",
        "compiled_model_path": "./build/int8_tvm.tar"
      }
    }
  ],
  "runtime": {
    "type": "kenning.runtimes.tvm.TVMRuntime",
    "parameters": {
      "save_model_path": "./build/int8_tvm.tar"
    }
  }
}
```

The above scenario takes the MobileNetV2 model trained for classification of cat and dog breeds, and passes it to:

- `kenning.optimizers.tflite.TFLiteCompiler` to quantize the weights of the model
- `kenning.optimizers.tvm.TVMCompiler` to create an optimized runtime of the model, utilizing AVX2 vector instructions.

The optimization of the model, its evaluation and report generation can be run with one command:

```
kenning optimize test report --json-cfg sample.json --measurements out.json --
↪report-path report.md --to-html report-html
```

The generated report can be found in the `report-html` directory.

2.4.2 Kenning tutorials

For more examples on Kenning usage, check Kenning tutorials.

2.5 Using Kenning as a library in Python scripts

Kenning is also a regular Python module - after pip installation it can be used in Python scripts. The example compilation of the model can look as follows:

```
from pathlib import Path
from kenning.datasets.pet_dataset import PetDataset
from kenning.modelwrappers.classification.tensorflow_pet_dataset import␣
↪TensorFlowPetDatasetMobileNetV2
from kenning.optimizers.tflite import TFLiteCompiler
from kenning.runtimes.tflite import TFLiteRuntime
from kenning.core.measurements import MeasurementsCollector
from kenning.utils.resource_manager import ResourceURI

dataset = PetDataset(
    root=Path('./build/pet-dataset/')
)
model = TensorFlowPetDatasetMobileNetV2(
    model_path=ResourceURI('kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5'),
    dataset=dataset
)
model.save_io_specification(model.model_path)
compiler = TFLiteCompiler(
    dataset=dataset,
    compiled_model_path=Path('./build/compiled-model.tflite'),
    modelframework='keras',
    target='default',
    inferenceinputtype='float32',
    inferenceoutputtype='float32'
)
compiler.compile(
    input_model_path=ResourceURI('kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5')
)
```

The above script downloads the dataset and compiles the model with FP32 inputs and outputs using TensorFlow Lite.

To get a quantized model, replace `target`, `inferenceinputtype` and `inferenceoutputtype` to `int8`:

```
compiler = TFLiteCompiler(
    dataset=dataset,
    compiled_model_path=Path('./build/compiled-model.tflite'),
    modelframework='keras',
```

(continues on next page)

(continued from previous page)

```
    target='int8',
    inferenceinputtype='int8',
    inferenceoutputtype='int8',
    dataset_percentage=0.3
)
compiler.compile(
    input_model_path=ResourceURI('kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5')
)
```

To check how the compiled model is performing, create `TFLiteRuntime` object and run local model evaluation:

```
runtime = TFLiteRuntime(
    protocol=None,
    model_path=Path('./build/compiled-model.tflite')
)

runtime.run_locally(
    dataset,
    model,
    Path('./build/compiled-model.tflite')
)
MeasurementsCollector.save_measurements('out.json')
```

The `runtime.run_locally` method runs benchmarks of the model on the current device.

The `MeasurementsCollector` class collects all benchmarks' data for model inference and saves it in JSON format that can be later used to render reports with the `kenning.scenarios.render_report` script.

As it can be observed, all classes accessible from JSON files in these scenarios share their configuration a with the classes in the Python scripts mentioned above.

2.6 Adding new implementations

`Dataset`, `ModelWrapper`, `Optimizer`, `Protocol`, `Runtime` and other classes from the `kenning.core` module have dedicated directories for their implementations. Each method in the base classes that requires implementation raises an `NotImplementedError` exception. They can be easily implemented or extended, but they need to conform to certain rules, usually described in the source documentation.

For more details and examples on how the Kenning framework can be adjusted and enhanced, follow the Kenning documentation. Implemented methods can be also overridden, if necessary.

Most of the base classes implement `form_argparse` and `from_argparse` methods. The former creates an argument parser and a group of arguments specific to the base class. The latter creates an object of the class based on the arguments from argument parser.

Inheriting classes can modify `form_argparse` and `from_argparse` methods to provide better control over their processing, but they should always be based on the results of their base

implementations.

CHAPTER

THREE

DEEP LEARNING DEPLOYMENT STACK

This chapter lists and describes typical actions performed on deep learning models before deployment on target devices.

3.1 From training to deployment

A deep learning application deployed on IoT devices usually goes through the following process:

- a dataset is prepared for a deep learning process,
- evaluation metrics are specified based on a given dataset and outputs,
- data in the dataset undergoes analysis, data loaders that perform the preprocessing are implemented,
- the deep learning model is either designed from scratch or a baseline is selected from a wide selection of existing pre-trained models for a given deep learning application (classification, detection, semantic segmentation, instance segmentation, etc.) and adjusted to a particular use case,
- a loss function and a learning algorithm are specified along with the deep learning model,
- the model is trained, evaluated and improved,
- the model is compiled to a representation that is applicable to a given target,
- the model is executed on a target device.

3.2 Dataset preparation

If a model is not available or it is trained for a different use case, the model needs to be trained or re-trained.

Each model requires a dataset - a set of sample inputs (audio signals, images, video sequences, OCT images, other sensors) and, usually, also outputs (association to class or classes, object location, object mask, input description). Datasets are usually split into the following categories:

- training dataset - the largest subset that is used to train a model,
- validation dataset - a relatively small set that is used to verify model performance after each training epoch (the metrics and loss function values show if any overfitting occurred during the training process),
- test dataset - the subset that acts as the final evaluation of a trained model.

It is required that the test dataset and the training dataset are mutually exclusive, so that the evaluation results are not biased in any way.

Datasets can be either designed from scratch or found in e.g.:

- Kaggle datasets,
- Google Dataset Search,
- Dataset list,
- Universities' pages,
- Open Images Dataset,
- Common Voice Dataset.

3.3 Model preparation and training

Currently, the most popular approach is to find an existing model that fits a given problem and perform transfer learning to adapt the model to the requirements. In transfer learning, the existing model's final layers are slightly modified to adapt to a new problem. These updated final layers of the model are trained using the training dataset. Finally, some additional layers are unfrozen and the training is performed on a larger number of parameters at a very small learning rate - this process is called fine-tuning.

Transfer learning provides a better starting point for the training process, allows to train a correctly performing model with smaller datasets and reduces the time required to train a model. The intuition behind this is that there are multiple common features between various objects in real-life environments, and the features learned from one deep learning scenario can be then reused in another scenario.

Once a model is selected, it requires adequate data input preprocessing in order to perform valid training. The input data should be normalized and resized to fit input tensor requirements. In case of the training dataset, especially if it is quite small, applying reasonable data augmentations like random brightness, contrast, cropping, jitters or rotations can significantly improve the training process and prevent the network from overfitting.

In the end, a proper training procedure needs to be specified. This step includes:

- loss function specification for the model. Some weights regularizations can be specified, along with the loss function, to reduce the chance of overfitting
- optimizer specification (like Adam, Adagrad). This involves setting hyperparameters properly or adding schedules and automated routines to set those hyperparameters (i.e. scheduling the learning rate value, or using LR-Finder to set the proper learning rate for the scenario)
- number of epochs specification or scheduling, e.g. early stopping can be introduced.
- providing some routines for quality metrics measurements
- providing some routines for saving intermediate models during training (periodically, or the best model according to a particular quality measure)

3.4 Model optimization

A successfully trained model may require some optimizations in order to run on given IoT hardware. The optimizations may involve precision of weights, computational representation, or model structure.

Models are usually trained with FP32 precision or mixed precision (FP32 + FP16, depending on the operator). Some targets, on the other hand, may significantly benefit from changing the precision from FP32 to FP16, INT8 or INT4. The optimizations here are straightforward for the FP16 precision, but the integer-based quantizations require dataset calibration to reduce precision without a significant loss in a model's quality.

Other optimizations change the computational representation of the model by e.g. layer fusion or specialized operators for convolutions of a particular shape, among others.

In the end, there are algorithmic optimizations that change the entire model structure, like weights pruning, conditional computation, model distillation (the current model acts as a teacher that is supposed to improve the quality of a much smaller model).

If these model optimizations are applied, the optimized models should be evaluated using the same metrics as the original model. This is required in order to find any drops in quality.

3.5 Model compilation and deployment

Deep learning compilers can transform model representation to:

- a source code for a different programming language, e.g. Halide, C, C++, Java, that can be later used on a given target,
- a machine code utilizing available hardware accelerators with e.g. OpenGL, OpenCL, CUDA, TensorRT, ROCm libraries,
- an FPGA bitstream,
- other targets.

Those compiled models are optimized to perform as efficiently as possible on given target hardware.

In the final step, the models are deployed on a hardware device.

CHAPTER

FOUR

DEFINING OPTIMIZATION PIPELINES IN KENNING

Kenning blocks (specified in the *Kenning API*) can be configured either via command line (see *Using Kenning via command line arguments*), or via configuration files, specified in JSON format. The latter approach allows the user to create more advanced and easy-to-reproduce scenarios for model deployment. Most notably, various optimizers available through Kenning can be chained to utilize various optimizations and get better performing models.

One of the scenarios most commonly used in Kenning is model optimization and compilation. It can be done using `kenning.scenarios.inference_tester`.

To run below examples it is required to install Kenning with dependencies as follows:

```
pip install "kenning[tensorflow,tflite,tvm] @ git+https://github.com/antmicro/
↪kenning.git"
```

4.1 JSON specification

The `kenning.scenarios.inference_tester` takes the specification of optimization and the testing flow in a JSON format. The root element of the JSON file is a dictionary that can have the following keys:

- `model_wrapper` - **mandatory field**, accepts dictionary as a value that defines the *ModelWrapper* object for the deployed model (provides I/O processing, optionally model).
- `dataset` - **mandatory field**, accepts dictionary as a value that defines the *Dataset* object for model optimization and evaluation.
- `optimizers` - *optional field*, accepts a list of dictionaries specifying the sequence of *Optimizer*-based optimizations applied to the model.
- `protocol` - *optional field*, defines the *Protocol* object used to communicate with a remote target platform.
- `runtime` - *optional field* (**required** when `optimizers` are provided), defines the *Runtime*-based object that will infer the model on target device.

Each dictionary in the fields above consists of:

- `type` - appropriate class for the key,
- `parameters` - `type`-specific arguments for an underlying class (see *Defining arguments for core classes*).

4.2 Model evaluation using its native framework

The simplest JSON configuration looks as follows:

Listing 4.1: mobilenetv2-tensorflow-native.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_name": "native",
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5",
      "batch_size": 32,
      "learning_rate": 0.0001,
      "num_epochs": 50,
      "logdir": "build/logs"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
  }
}
```

It only takes `model_wrapper` and `dataset`. This way, the model will be loaded and evaluated using its native framework.

The *ModelWrapper* used is `TensorFlowPetDatasetMobileNetV2`, which is a MobileNetV2 model trained to classify 37 breeds of cats and dogs. In the `type` field, we specify the full "path" to the class by specifying the module it is implemented in (`kenning.modelwrappers.classification.tensorflow_pet_dataset`) and the name of the class (`TensorFlowPetDatasetMobileNetV2`) in a Python-like format (dot-separated).

In `parameters`, arguments specific to `TensorFlowPetDatasetMobileNetV2` are provided. The following parameters are available based on the argument specification:

```
# this argument structure is taken from kenning.core.model - it is inherited by␣
↪child classes
arguments_structure = {
    'modelpath': {
        'argparse_name': '--model-path',
        'description': 'Path to the model',
        'type': Path,
        'required': True
    }
}
```

The only mandatory parameter here is `model_path`, which points to a file containing the model. It is a required argument.

The `dataset` used here, is `PetDataset`. Like previously, it is provided in a module-like format (`kenning.datasets.pet_dataset.PetDataset`). The parameters here are specified in `kenning.core.dataset.Dataset` (inherited) and `kenning.core.dataset.PetDataset`:

```
arguments_structure = {
    # coming from kenning.core.dataset.Dataset
    'root': {
        'argparse_name': '--dataset-root',
        'description': 'Path to the dataset directory',
        'type': Path,
        'required': True
    },
    'batch_size': {
        'argparse_name': '--inference-batch-size',
        'description': 'The batch size for providing the input data',
        'type': int,
        'default': 1
    },
    'download_dataset': {
        'description': 'Downloads the dataset before taking any action',
        'type': bool,
        'default': False
    },
    # coming from kenning.datasets.pet_dataset.PetDataset
    'classify_by': {
        'argparse_name': '--classify-by',
        'description': 'Determines if classification should be performed by␣
↪species or by breeds',
        'default': 'breeds',
        'enum': ['species', 'breeds']
    },
    'image_memory_layout': {
        'argparse_name': '--image-memory-layout',
        'description': 'Determines if images should be delivered in NHWC or NCHW␣
↪format',
        'default': 'NHWC',
        'enum': ['NHWC', 'NCHW']
    }
}
```

As visible, the parameters allow the user to:

- specify the dataset's location,
- download the dataset,
- configure data layout and batch size,
- configure anything specific to the dataset.

Note: For more details on defining parameters for Kenning core classes, check *Defining arguments for core classes*.

If `optimizers` or `runtime` are not specified, the model is executed using the *ModelWrapper*'s `run_inference` method. The dataset test data is passed through the model and evaluation metrics are collected.

To run the defined pipeline (assuming that the JSON file is under `pipeline.json`), run:

```
kenning test \
    --json-cfg mobilenetv2-tensorflow-native.json \
    --measurements measurements.json \
    --verbosity INFO
```

The `measurements.json` file is the output of the `kenning.scenarios.inference_tester` providing measurement data. It contains information such as:

- the JSON configuration defined above,
- versions of core class packages used (e.g. `tensorflow`, `torch`, `tvm`),
- available resource usage readings (CPU usage, GPU usage, memory usage),
- data necessary for evaluation, such as predictions, confusion matrix, etc.

This information can be later used for *Generating performance reports*.

Note: Check *Kenning measurements* for more information.

4.3 Model training

Provided that training is supported by a given model, you can specify parameters as follows:

Listing 4.2: mobilenetv2-tensorflow-native.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_name": "native",
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5",
      "batch_size": 32,
      "learning_rate": 0.0001,
      "num_epochs": 50,
      "logdir": "build/logs"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

To train the model, simply run:

```
kenning train \
    --json-cfg mobilenetv2-tensorflow-native.json \
    --verbosity INFO
```

Furthermore, the configuration is shared among subcommands:

```
kenning train test \
    --json-cfg train-test.json \
    --measurements output.json \
    --verbosity INFO
```

File with measurements also stores training information, which can be viewed directly or displayed in a generated report.

4.4 Optimizing and running a model on a single device

Model optimization and deployment can be performed directly on target device, if the device is able to perform the optimization steps. It can also be used to check the outcome of certain optimizations on a desktop platform before deployment.

Optimizations and compilers used in a scenario are defined in the `optimizers` field. This field accepts a list of optimizers - they are applied to the model in the same order in which they are defined in the `optimizers` field.

For example, a model can be subjected to the following optimizations:

- Quantization of weights and activations using TensorFlow Lite.
- Conversion of data layout from NHWC to NCHW format using Apache TVM
- Compilation to x86 runtime with AVX2 vector extensions using Apache TVM.

Such case will result is the following scenario:

Listing 4.3: mobilenetv2-tensorflow-tvm-avx-int8.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_name": "tvm-avx2-int8",
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5"
    }
  },
  "dataset": {
```

(continues on next page)

(continued from previous page)

```
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
  },
  "optimizers": [
    {
      "type": "kenning.optimizers.tflite.TFLiteCompiler",
      "parameters": {
        "target": "int8",
        "compiled_model_path": "./build/int8.tflite",
        "inference_input_type": "int8",
        "inference_output_type": "int8"
      }
    },
    {
      "type": "kenning.optimizers.tvm.TVMCompiler",
      "parameters": {
        "target": "llvm -mcpu=core-avx2",
        "opt_level": 3,
        "conv2d_data_layout": "NCHW",
        "compiled_model_path": "./build/int8_tvm.tar"
      }
    }
  ],
  "runtime": {
    "type": "kenning.runtimes.tvm.TVMRuntime",
    "parameters": {
      "save_model_path": "./build/int8_tvm.tar"
    }
  }
}
```

As emphasized above, the `optimizers` list is added, with two entries:

- a `kenning.optimizers.tflite.TFLiteCompiler` type block, quantizing the model,
- a `kenning.optimizers.tvm.TVMCompiler` type block, performing remaining optimization steps.

In the `runtime` field, a TVM-specific `kenning.runtimes.tvm.TVMRuntime` type is used.

The first optimizer on the list reads the input model path from the *ModelWrapper*'s `model_path` field. Each consecutive *Optimizer* reads the model from a file saved by the previous *Optimizer*. In the simplest scenario, the model is saved to `compiled_model_path` in each optimizer, and is fetched by the next *Optimizer*.

In case the default output file type of the previous *Optimizer* is not supported by the next *Optimizer*, the first common supported model format is determined and used to pass the model between optimizers.

In case no such format exists, the `kenning.scenarios.inference_tester` returns an error.

Note: More details on input/output formats between *Optimizer* objects can be found in *Developing Kenning blocks*.

The scenario can be executed as follows:

```
kenning optimize test --json-cfg mobilenetv2-tensorflow-tvm-avx-int8.json --
↪measurements output.json
```

4.5 Compiling a model and running it remotely

For some platforms, we cannot run a Python script to evaluate or run the model to check its quality - the dataset is too large to fit in the storage, no libraries or compilation tools are available for the target platform, or the device does not have an operating system to run Python on.

In such cases, it is possible to evaluate the system remotely using the *Protocol* and the `kenning.scenarios.inference_server` scenario.

For this use case, we need two JSON files - one for inference server configuration, and another one for the `kenning.scenarios.inference_tester` configuration, which acts as a runtime client.

The client and the server may communicate via different means, protocols and interfaces - we can use TCP communication, UART communication or other. It depends on the *Protocol* used.

In addition, in such scenario optimizers can be executed either on host (which is default behavior) or on target device. To specify it, you can use `location` parameter of the *Optimizer*.

To create client/server scenario configuration it is required to add a `protocol` entry:

Listing 4.4: tflite-tvm-classification-client-server.json

```
{
    "platform": {
        "type": "LocalPlatform"
    },
    "model_wrapper": {
        "type": "TensorFlowPetDatasetMobileNetV2",
        "parameters": {
            "model_path": "kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5"
        }
    },
    "dataset": {
        "type": "PetDataset",
        "parameters": {
            "dataset_root": "./build/PetDataset"
        }
    },
    "optimizers": [
        {
```

(continues on next page)

(continued from previous page)

```
            "type": "TFLiteCompiler",
            "parameters": {
                "target": "default",
                "compiled_model_path": "./build/compiled_tflite.tflite",
                "inference_input_type": "float32",
                "inference_output_type": "float32"
            }
        },
        {
            "type": "TVMCompiler",
            "parameters": {
                "target": "llvm -mcpu=core-avx2",
                "compiled_model_path": "./build/compiled_tvm.tar",
                "opt_level": 3,
                "location": "target"
            }
        }
    ],
    "runtime": {
        "type": "TVMRuntime",
        "parameters": {
            "save_model_path": "./build/compiled_model.tar"
        }
    },
    "protocol": {
        "type": "NetworkProtocol"
    }
}
```

In the `protocol` entry, we specify a `kenning.protocols.network.NetworkProtocol` and provide a server address (`host`), an application port (`port`) and packet size (`packet_size`)

The server parses only `runtime` and `protocol` from the configuration, so any changes to the other of the blocks does not require server restart. The server uses `protocol` to receive requests from clients and `runtime` to run the tested models.

The remaining things are provided by the client - input data and model. Direct outputs from the model are sent as is to the client, so it can postprocess them and evaluate the model using the dataset. The server also sends measurements from its sensors in JSON format as long as it is able to collect and send them.

First, run the server, so that it is available for the client:

```
kenning server \
    --json-cfg tflite-tvm-classification-client-server.json \
    --verbosity INFO &
```

Then, run the client:

```
kenning optimize test \
    --json-cfg tflite-tvm-classification-client-server.json \
```

(continues on next page)

(continued from previous page)

```
    --measurements ./build/tflite-tvm-classification.json \
    --verbosity INFO
```

The rest of the flow is automated.

To execute one of the optimizers on the target-side, simply add the `location` parameter as follows:

```
"optimizers":
[
    {
        "type": "kenning.optimizers.tflite.TFLiteCompiler",
        "parameters":
        {
            "target": "int8",
            "compiled_model_path": "./build/int8.tflite",
            "inference_input_type": "int8",
            "inference_output_type": "int8"
        }
    },
    {
        "type": "kenning.optimizers.tvm.TVMCompiler",
        "parameters": {
            "target": "llvm -mcpu=core-avx2",
            "opt_level": 3,
            "conv2d_data_layout": "NCHW",
            "compiled_model_path": "./build/int8_tvm.tar",
            "location": "target"
        }
    }
],
```

and start the client the same as above (it is not required to restart server).

CHAPTER

FIVE

USING KENNING VIA COMMAND LINE ARGUMENTS

Kenning provides several scripts for training, compilation and benchmarking of deep learning models on various target hardware. The executable scripts are present in the kenning.scenarios module. Sample bash scripts using the scenarios are located in the scripts directory in the repository.

Runnable scripts in scenarios require implemented classes to be provided from the `kenning.core` module to perform such actions as in-framework inference, model training, model compilation and model benchmarking on target.

To run below examples it is required to install Kenning with dependencies as follows:

```
pip install "kenning[tensorflow,tvm] @ git+https://github.com/antmicro/kenning.git
↪"
```

5.1 Command-line arguments for classes

Each class (*Dataset*, *ModelWrapper*, *Optimizer* and other) provided to the runnable scripts in scenarios can provide command-line arguments that configure the work of an object of the given class.

Each class in `kenning.core` implements `form_argparse` and `from_argparse` methods. The former creates an `argparse` group for a given class with its parameters. The latter takes the arguments parsed by `argparse` and returns the object of a class.

5.2 Autocompletion for command line interface

Kenning provides autocompletion for its command line interface. This feature requires additional configuration to work properly, which can be done using `kenning completion` command. Optionally, it can be configured as described in argcomplete documentation.

5.3 Model training

`kenning.scenarios.model_training` performs model training using Kenning's *ModelWrapper* and *Dataset* objects. To get the list of training parameters, select the model and training dataset to use (i.e. `TensorFlowPetDatasetMobileNetV2` model and `PetDataset` dataset) and run:

```
kenning train \
    --modelwrapper-cls kenning.modelwrappers.classification.tensorflow_pet_
↪dataset.TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls kenning.datasets.pet_dataset.PetDataset \
    -h
```

To simplify usage of internal implementations of models, optimizations and other, we also support referring to classes just by their name, without module path:

```
kenning train \
    --modelwrapper-cls TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls PetDataset \
    -h
```

This will list the possible parameters that can be used to configure the dataset, the model, and the training parameters. For the above call, the output is as follows:

```
common arguments:
  -h, --help            show this help message and exit
  --verbosity {DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        Verbosity level

'train' arguments:
  --modelwrapper-cls MODELWRAPPER_CLS
                        ModelWrapper-based class with inference implementation to␣
↪import
  --dataset-cls DATASET_CLS
                        Dataset-based class with dataset to import
  --batch-size BATCH_SIZE
                        The batch size for training
  --learning-rate LEARNING_RATE
                        The learning rate for training
  --num-epochs NUM_EPOCHS
                        Number of epochs to train for
  --logdir LOGDIR       Path to the training logs directory

ModelWrapper arguments:
  --model-path MODEL_PATH
                        Path to the model

PetDataset arguments:
  --classify-by {species,breeds}
                        Determines if classification should be performed by␣
↪species or by breeds
  --image-memory-layout {NHWC,NCHW}
```

(continues on next page)

(continued from previous page)

```
                        Determines if images should be delivered in NHWC or NCHW␣
↪format

Dataset arguments:
  --dataset-root DATASET_ROOT
                        Path to the dataset directory
  --inference-batch-size INFERENCE_BATCH_SIZE
                        The batch size for providing the input data
  --download-dataset    Downloads the dataset before taking any action. If the␣
↪dataset files are already downloaded and
                        the checksum is correct then they are not downloaded␣
↪again. Is enabled by default.
  --force-download-dataset
                        Forces dataset download
  --external-calibration-dataset EXTERNAL_CALIBRATION_DATASET
                        Path to the directory with the external calibration␣
↪dataset
  --split-fraction-test SPLIT_FRACTION_TEST
                        Default fraction of data to leave for model testing
  --split-fraction-val SPLIT_FRACTION_VAL
                        Default fraction of data to leave for model valdiation
  --split-seed SPLIT_SEED
                        Default seed used for dataset split
```

Note: The list of options depends on *ModelWrapper* and *Dataset*.

At the end, the training can be configured as follows:

```
kenning train \
    --modelwrapper-cls TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls PetDataset \
    --logdir build/logs \
    --dataset-root build/pet-dataset \
    --model-path build/trained-model.h5 \
    --batch-size 32 \
    --learning-rate 0.0001 \
    --num-epochs 50
```

This will train the model with a 0.0001 learning rate and batch size 32 for 50 epochs. The trained model will be saved as `build/trained-model.h5`.

5.4 In-framework inference performance measurements

The `kenning.scenarios.inference_performance` script runs inference on a given model in a framework it was trained on. It requires you to provide:

- a *ModelWrapper*-based object wrapping the model to be tested,
- a *Dataset*-based object wrapping the dataset applicable to the model,
- a path to the output JSON file with performance and quality metrics gathered during inference by the *Measurements* object.

The example call for the method is as follows:

```
kenning test \
    --modelwrapper-cls TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls PetDataset \
    --measurements build/tensorflow_pet_dataset_mobilenetv2.json \
    --model-path kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5 \
    --dataset-root build/pet-dataset/
```

The script downloads the dataset to the `build/pet-dataset` directory, loads the `tensorflow_pet_dataset_mobilenetv2.h5` model, runs inference on all images from the dataset and collects performance and quality metrics throughout the run. The performance data stored in the JSON file can be later rendered using *Generating performance reports*.

5.5 Testing inference on target hardware

The `kenning.scenarios.inference_tester` and `kenning.scenarios.inference_server` are used for inference testing on target hardware. The `inference_tester` loads the dataset and the model, compiles the model and runs inference either locally or remotely using `inference_server`.

The `inference_server` receives the model and input data, and sends output data and statistics.

Both `inference_tester` and `inference_server` require *Runtime* to determine the model execution flow. Both scripts communicate using the communication protocol implemented in the *Protocol*.

At the end, the `inference_tester` returns the benchmark data in the form of a JSON file extracted from the *Measurements* object.

The `kenning.scenarios.inference_tester` requires:

- a *ModelWrapper*-based class that implements model loading, I/O processing and optionally model conversion to ONNX format,
- a *Runtime*-based class that implements data processing and the inference method for the compiled model on the target hardware,
- a *Dataset*-based class that implements data sample fetching and model evaluation,
- a path to the output JSON file with performance and quality metrics.

An *Optimizer*-based class can be provided to compile the model for a given target if needed.

Optionally, it requires a *Protocol*-based class when running remotely to communicate with the `kenning.scenarios.inference_server`.

To print the list of required arguments, run:

```
kenning optimize test \
    --modelwrapper-cls TensorFlowPetDatasetMobileNetV2 \
    --runtime-cls TVMRuntime \
    --dataset-cls PetDataset \
    --compiler-cls TVMCompiler \
    --protocol-cls NetworkProtocol \
    -h
```

With the above classes, the help can look as follows:

```
common arguments:
  -h, --help            show this help message and exit
  --verbosity {DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        Verbosity level
  --convert-to-onnx CONVERT_TO_ONNX
                        Before compiling the model, convert it to ONNX and use in␣
↪compilation (provide a path to save here)
  --measurements MEASUREMENTS
                        The path to the output JSON file with measurements

Inference configuration with JSON:
  Configuration with pipeline defined in JSON file. This section is not␣
↪compatible with 'Inference configuration with flags'. Arguments with '*' are␣
↪required.

  --json-cfg JSON_CFG   * The path to the input JSON file with configuration of␣
↪the inference

Inference configuration with flags:
  Configuration with flags. This section is not compatible with 'Inference␣
↪configuration with JSON'. Arguments with '*' are required.

  --modelwrapper-cls MODELWRAPPER_CLS
                        * ModelWrapper-based class with inference implementation␣
↪to import
  --dataset-cls DATASET_CLS
                        * Dataset-based class with dataset to import
  --compiler-cls COMPILER_CLS
                        * Optimizer-based class with compiling routines to import
  --runtime-cls RUNTIME_CLS
                        Runtime-based class with the implementation of model␣
↪runtime
  --protocol-cls PROTOCOL_CLS
                        Protocol-based class with the implementation of␣
↪communication between inference
```

(continues on next page)

(continued from previous page)

```
                        tester and inference runner

ModelWrapper arguments:
  --model-path MODEL_PATH
                        Path to the model

PetDataset arguments:
  --classify-by {species,breeds}
                        Determines if classification should be performed by␣
↪species or by breeds
  --image-memory-layout {NHWC,NCHW}
                        Determines if images should be delivered in NHWC or NCHW␣
↪format

Dataset arguments:
  --dataset-root DATASET_ROOT
                        Path to the dataset directory
  --inference-batch-size INFERENCE_BATCH_SIZE
                        The batch size for providing the input data
  --download-dataset    Downloads the dataset before taking any action. If the␣
↪dataset files are already downloaded and the checksum is correct then they are␣
↪not downloaded again. Is enabled by default.
  --force-download-dataset
                        Forces dataset download
  --external-calibration-dataset EXTERNAL_CALIBRATION_DATASET
                        Path to the directory with the external calibration␣
↪dataset
  --split-fraction-test SPLIT_FRACTION_TEST
                        Default fraction of data to leave for model testing
  --split-fraction-val SPLIT_FRACTION_VAL
                        Default fraction of data to leave for model valdiation
  --split-seed SPLIT_SEED
                        Default seed used for dataset split

TVMRuntime arguments:
  --save-model-path SAVE_MODEL_PATH
                        Path where the model will be uploaded
  --target-device-context {llvm,stackvm,cpu,c,test,hybrid,composite,cuda,nvptx,cl,
↪opencl,sdaccel,aocl,aocl_sw_emu,vulkan,metal,vpi,rocm,ext_dev,hexagon,webgpu}
                        What accelerator should be used on target device
  --target-device-context-id TARGET_DEVICE_CONTEXT_ID
                        ID of the device to run the inference on
  --runtime-use-vm      At runtime use the TVM Relay VirtualMachine

Runtime arguments:
  --disable-performance-measurements
                        Disable collection and processing of performance metrics

TVMCompiler arguments:
```

(continues on next page)

(continued from previous page)

```
  --model-framework {keras,onnx,darknet,torch,tflite}
                        The input type of the model, framework-wise
  --target TARGET       The kind or tag of the target device
  --target-host TARGET_HOST
                        The kind or tag of the host (CPU) target device
  --opt-level OPT_LEVEL
                        The optimization level of the compilation
  --libdarknet-path LIBDARKNET_PATH
                        Path to the libdarknet.so library, for darknet models
  --compile-use-vm      At compilation stage use the TVM Relay VirtualMachine
  --output-conversion-function {default,dict_to_tuple}
                        The type of output conversion function used for PyTorch␣
↪conversion
  --conv2d-data-layout CONV2D_DATA_LAYOUT
                        Configures the I/O layout for the CONV2D operations
  --conv2d-kernel-layout CONV2D_KERNEL_LAYOUT
                        Configures the kernel layout for the CONV2D operations
  --use-fp16-precision  Applies conversion of FP32 weights to FP16
  --use-int8-precision  Applies conversion of FP32 weights to INT8
  --use-tensorrt        For CUDA targets: delegates supported operations to␣
↪TensorRT
  --dataset-percentage DATASET_PERCENTAGE
                        Tells how much data from the calibration dataset␣
↪(training or external) will be used for calibration dataset

Optimizer arguments:
  --compiled-model-path COMPILED_MODEL_PATH
                        The path to the compiled model output

NetworkProtocol arguments:
  --host HOST           The address to the target device
  --port PORT           The port for the target device

BytesBasedProtocol arguments:
  --packet-size PACKET_SIZE
                        The maximum size of the received packets, in bytes.
  --endianness {big,little}
                        The endianness of data to transfer
```

The `kenning.scenarios.inference_server` requires only:

- a *Protocol*-based class for the implementation of the communication,
- a *Runtime*-based class for the implementation of runtime routines on device.

Both classes may require some additional arguments that can be listed with the `-h` flag.

An example script for the `inference_tester` is:

```
kenning optimize test \
    --modelwrapper-cls TensorFlowPetDatasetMobileNetV2 \
```

(continues on next page)

(continued from previous page)

```
    --runtime-cls TFLiteRuntime \
    --dataset-cls PetDataset \
    --measurements ./build/google-coral-devboard-tflite-tensorflow.json \
    --compiler-cls TFLiteCompiler \
    --protocol-cls NetworkProtocol \
    --model-path kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5 \
    --model-framework keras \
    --target "edgetpu" \
    --compiled-model-path build/compiled-model.tflite \
    --inference-input-type int8 \
    --inference-output-type int8 \
    --host 192.168.188.35 \
    --port 12344 \
    --packet-size 32768 \
    --save-model-path /home/mendel/compiled-model.tflite \
    --dataset-root build/pet-dataset \
    --inference-batch-size 1 \
    --verbosity INFO
```

The above runs with the following `inference_server` setup:

```
kenning server \
    --protocol-cls NetworkProtocol \
    --runtime-cls TFLiteRuntime \
    --host 0.0.0.0 \
    --port 12344 \
    --packet-size 32768 \
    --save-model-path /home/mendel/compiled-model.tflite \
    --delegates-list libedgetpu.so.1 \
    --verbosity INFO
```

Note: This run was tested on a Google Coral Devboard device.

`kenning.scenarios.inference_tester` can be also executed locally - in this case, the `--protocol-cls` argument can be skipped. The example call is as follows:

```
kenning optimize test \
    --modelwrapper-cls TensorFlowPetDatasetMobileNetV2 \
    --runtime-cls TVMRuntime \
    --dataset-cls PetDataset \
    --measurements ./build/local-cpu-tvm-tensorflow-classification.json \
    --compiler-cls TVMCompiler \
    --model-path kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5 \
    --model-framework keras \
    --target "llvm" \
    --compiled-model-path ./build/compiled-model.tar \
```

(continues on next page)

(continued from previous page)

```
    --opt-level 3 \
    --save-model-path ./build/compiled-model.tar \
    --target-device-context cpu \
    --dataset-root ./build/pet-dataset/ \
    --inference-batch-size 1 \
    --verbosity INFO
```

Note: For more examples of running `inference_tester` and `inference_server`, check the kenning/scripts directory. Directories with scripts for client and server calls for various target devices, deep learning frameworks and compilation frameworks can be found in the kenning/scripts/edge-runtimes directory.

5.6 Running inference

`kenning.scenarios.inference_runner` is used to run inference locally on a pre-compiled model.

`kenning.scenarios.inference_runner` requires:

- a *ModelWrapper*-based class that performs I/O processing specific to the model,
- a *Runtime*-based class that runs inference on target using the compiled model,
- a *DataProvider*-based class that implements fetching of data samples from various sources,
- a list of *OutputCollector*-based classes that implement output processing for the specific use case.

To print the list of required arguments, run:

```
python3 -m kenning.scenarios.inference_runner \
    kenning.modelwrappers.object_detection.darknet_coco.TVMDarknetCOCOYOLOV3 \
    TVMRuntime \
    kenning.dataproviders.camera_dataprovider.CameraDataProvider \
    --output-collectors kenning.outputcollectors.name_printer.NamePrinter \
    -h
```

With the above classes, the help can look as follows:

```
positional arguments:
  modelwrappercls       ModelWrapper-based class with inference implementation to␣
↪import
  runtimecls            Runtime-based class with the implementation of model␣
↪runtime
  dataprovidercls       DataProvider-based class used for providing data
optional arguments:
  -h, --help            show this help message and exit
  --output-collectors OUTPUT_COLLECTORS [OUTPUT_COLLECTORS ...]
                        List to the OutputCollector-based classes where the␣
```

(continues on next page)

(continued from previous page)

```
↪results will be passed
  --verbosity {DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        Verbosity level
Inference model arguments:
  --model-path MODEL_PATH
                        Path to the model
  --classes CLASSES     File containing Open Images class IDs and class names in␣
↪CSV format to use (can be generated using
                        kenning.scenarios.open_images_classes_extractor) or class␣
↪type
Runtime arguments:
  --disable-performance-measurements
                        Disable collection and processing of performance metrics
  --save-model-path SAVE_MODEL_PATH
                        Path where the model will be uploaded
  --target-device-context {llvm,stackvm,cpu,c,cuda,nvptx,cl,opencl,aocl,aocl_sw_
↪emu,sdaccel,vulkan,metal,vpi,rocm,ext_dev,hexagon,webgpu}
                        What accelerator should be used on target device
  --target-device-context-id TARGET_DEVICE_CONTEXT_ID
                        ID of the device to run the inference on
  --input-dtype INPUT_DTYPE
                        Type of input tensor elements
  --runtime-use-vm      At runtime use the TVM Relay VirtualMachine
  --use-json-at-output  Encode outputs of models into a JSON file with base64-
↪encoded arrays
DataProvider arguments:
  --video-file-path VIDEO_FILE_PATH
                        Video file path (for cameras, use /dev/videoX where X is␣
↪the device ID eg. /dev/video0)
  --image-memory-layout {NHWC,NCHW}
                        Determines if images should be delivered in NHWC or NCHW␣
↪format
  --image-width IMAGE_WIDTH
                        Determines the width of the image for the model
  --image-height IMAGE_HEIGHT
                        Determines the height of the image for the model
OutputCollector arguments:
  --print-type {detector,classificator}
                        What is the type of model that will input data to the␣
↪NamePrinter
```

An example script for `inference_runner`:

```
python3 -m kenning.scenarios.inference_runner \
    kenning.modelwrappers.object_detection.darknet_coco.TVMDarknetCOCOYOLOV3 \
    TVMRuntime \
    kenning.dataproviders.camera_dataprovider.CameraDataProvider \
    --output-collectors kenning.outputcollectors.detection_visualizer.
↪DetectionVisualizer kenning.outputcollectors.name_printer.NamePrinter \
```

(continues on next page)

(continued from previous page)

```
    --disable-performance-measurements \
    --model-path kenning:///models/object_detection/yolov3.weights \
    --save-model-path ../compiled-model.tar \
    --target-device-context "cuda" \
    --verbosity INFO \
    --video-file-path /dev/video0
```

5.7 Generating performance reports

`kenning.scenarios.inference_performance` and `kenning.scenarios.inference_tester` return a JSON file as the result of benchmarks. They contain both performance metrics data, and quality metrics data.

The data from JSON files can be analyzed, processed and visualized by the `kenning.scenarios.render_report` script. This script parses the information in JSON files and returns an RST file with the report, along with visualizations.

It requires:

- a JSON file with benchmark data,
- a report name for use in the RST file and for creating Sphinx refs to figures,
- an RST output file name,
- `--root-dir` specifying the root directory of the Sphinx documentation where the RST file will be embedded (it is used to compute relative paths),
- `--img-dir` specifying the path where the figures should be saved,
- `--report-types`, which is a list describing the types the report falls into.

An example call and the resulting RST file can be observed in *Sample autogenerated report*.

As for now, the available report types are:

- `performance` - this is the most common report type that renders information about overall inference performance metrics, such as inference time, CPU usage, RAM usage, or GPU utilization,
- `classification` - this report is specific to the classification task, it renders classification-specific quality figures and metrics, such as confusion matrices, accuracy, precision, G-mean,
- `detection` - this report is specific to the detection task, it renders detection-specific quality figures and metrics, such as recall-precision curves or mean average precision.

5.8 Displaying information about available classes

`kenning.scenarios.list_classes` and `kenning.scenarios.class_info` provide useful information about classes and can help in creating JSON scenarios.

`kenning.scenarios.list_classes` will list all available classes by default, though the output can be limited by providing positional arguments representing groups of modules: `optimizers`, `runners`, `dataproviders`, `datasets`, `modelwrappers`, `onnxconversions`, `outputcollectors`, `runtimes`. The amount of information displayed can be controlled using `-v` and `-vv` flags.

To print available arguments run `python -m kenning.scenarios.list_classes -h`.

`kenning.scenarios.class_info` provides information about a class given in an argument. More precisely, it will display:

- module and class docstrings,
- dependencies along with the information whether they are available in the current python environment,
- supported input and output formats,
- arguments structure used in JSON configurations.

The script accepts class names, e.g.: `TFLiteRuntime`.

However, module-like paths to the files (e.g. `kenning.runtimes.tflite`) can be used, too. Additionally, a class can be added to the path like so: `kenning.runtimes.tflite.TFLiteRuntime`. To get more detailed information, an optional `--load-class-with-args` argument can be passed. This needs all required class arguments to be provided, and that all dependencies are available.

For more detail, check `python -m kenning.scenarios.class_info -h`.

CHAPTER

SIX

KENNING GALLERY OF USE CASES

This section contains step-by-step examples of Kenning use cases.

6.1 Generating anomaly detection models for the MAX32690 Evaluation Kit with AutoML

This example demonstrates how to find anomaly detection models and evaluate them on a simulated MCU using Kenning, Zephyr RTOS and Renode. The models are generated with the Auto-PyTorch AutoML framework. The platform used in this example is the MAX32690 Evaluation Kit. The demo uses Kenning Zephyr Runtime and Zephyr RTOS for execution of the model on simulated hardware.

6.1.1 Prepare an environment for development

Assuming `git` and `docker` are available in the system, first let's clone the repository:

```
git clone https://github.com/antmicro/kenning-zephyr-runtime-example-app.git␣
↪sample-app
```

Then, let's build the Docker image based on `ghcr.io/antmicro/kenning-zephyr-runtime:latest` for quicker environment setup:

```
docker build -t kenning-automl ./sample-app/environments
```

After successful build of the image, run:

```
docker run --rm -it --name automl -w $(realpath sample-app) -v $(pwd):$(pwd)␣
↪kenning-automl:latest bash
```

Then, in the Docker container, initialize the Zephyr application and Kenning Zephyr Runtime as follows:

```
west init -l app
west update
west zephyr-export
pushd ./kenning-zephyr-runtime
./scripts/prepare_zephyr_env.sh
./scripts/prepare_modules.sh
popd
```

Note: To make sure you have the latest version of the Kenning with AutoML features, run:

```
pip3 install "kenning[iree,tvm,torch,anomaly_detection,auto_pytorch,tensorflow,
↪tflite,reports,renode,uart] @ git+https://github.com/antmicro/kenning.git"
```

With the configured environment, you can now run the AutoML flow.

To create a `workspace` directory where intermediate results of command executed further will be stored, run:

```
mkdir -p workspace
```

Note: For more step-by-step instructions on how to set up the environment locally, see Kenning Zephyr Runtime build instructions.

6.1.2 Run AutoML flow

The AutoML flow can be configured using the YAML below.

```
# This scenario demonstrates AutoML flow on example of Anomaly Detection in time␣
↪series
# for MAX32960 Evaluation Kit.
#
# Configures AutoML flow
automl:
  type: AutoPyTorchML
  parameters:
    # Time limit for AutoML task (in minutes)
    time_limit: 20
    # List of model architectures used for AutoML,
    # represented by ModelWrapper (has to implement AutoMLModel class)
    use_models:
      - PyTorchAnomalyDetectionVAE
    # Folder storing AutoML results
    output_directory: ./workspace/automl-results
    # Maximum number of models returned by the flow
    n_best_models: 5

    # AutoPyTorch specific options
    # Chosen metric to optimize
    optimize_metric: f1
    # Type of budget for training models, either epochs or time limit
    budget_type: epochs
    # Lower and upper limit of the budger
    min_budget: 1
    max_budget: 5
    # Size of the application that will use generated models
```

(continues on next page)

(continued from previous page)

```
    application_size: 75.5

# Chooses the platform to run
platform:
  type: ZephyrPlatform
  parameters:
    # Chooses MAX32690 Evaluation Kit
    name: max32690evkit/max32690/m4
    # Use Renode to simulate the platform
    simulated: True

# Defines dataset for anomaly detection
dataset:
  type: AnomalyDetectionDataset
  parameters:
    dataset_root: ./workspace/CATS
    csv_file: kenning:///datasets/anomaly_detection/cats_nano.csv
    split_fraction_test: 0.1
    split_seed: 12345
    inference_batch_size: 1

# Remaining configurations, despite not being directly
# used for a AutoML flow, are copied to resulting scenarios,
# and influences the standard Kenning flow run with generated models

optimizers:
- type: TFLiteCompiler
  parameters:
    target: default
    compiled_model_path: ./workspace/automl-results/vae.tflite
    inference_input_type: float32
    inference_output_type: float32

runtime_builder:
  type: ZephyrRuntimeBuilder
  parameters:
    workspace: .
    # venv_dir: ../.venv
    output_path: ./workspace/kzr_build
    run_west_update: false
    extra_targets: [board-repl]
```

The introduced `automl` entry provides an implementation of the AutoML flow, as well as its parameters. The main parameters affecting the model search process are:

- `time_limit` - determines how much time the AutoML algorithm will spend looking for the models
- `application_size` - determines how much of the space is consumed by the app excluding the model. By taking into account the provided size of the application ran on the board and the size of RAM (received from the platform class), the flow will automatically reject

(before training) all models that do not fit into available space. This ensures than time is not wasted on models that cannot be run on the hardware (or its simulation).

- `use_models` - provides models for the algorithm to take into account. Models provided in the list contribute to the search space of available configurations, providing their hyperparameters with acceptable ranges and structure definition. It is possible to override default ranges specifying them right after chosen model:

```
use_models:
  - PyTorchAnomalyDetectionVAE:
      encoder_neuron_list:
        list_range: [4, 15]
        item_range: [6, 128]
      dropout_rate:
        item_range: [0.1, 0.4]
      output_activation:
        enum: [tanh]
```

The list of AutoML specific options is available in *Defining arguments for classes section*.

Note: To use the microTVM runtime, change the optimizer to:

```
optimizers:
- type: TVMCompiler
  parameters:
    compiled_model_path: ./workspace/vae.graph_data
```

Depending on runtime selection, application size may vary greatly, so to generate models with correct size, make sure to adjust this value.

To run the full flow, use this command:

```
kenning automl optimize test report \
  --cfg ./kenning-zephyr-runtime/kenning-scenarios/renode-zephyr-auto-tflite-
↪automl-vae-max32690.yml \
  --report-path ./workspace/automl-report/report.md \
  --allow-failures --to-html --ver INFO \
  --skip-general-information
```

The command above:

- Runs an AutoML search for models for the amount of time specified in the `time_limit`
- Optimizes best-performing using given optimization pipeline
- Runs evaluation of the compiled models in Renode simulation
- Generates a full comparison report for the models so that user can pick the best one (located in `workspace/automl-report/report/report.html`)
- Generates optimized models (`vae.<id>.tflite`), their AutoML-derived configuration (`automl_conf_<id>.yml`) and IO specification files for Kenning (`vae.<id>.tflite.json`) under `workspace/automl-results/`.

6.1.3 Run AutoML flow with quantization

The AutoML flow can also take into account the model size after quantization. To make it possible, a scenario has to define at least one Optimizer which will quantize the model:

Listing 6.1: automl_quantization.yml

```yaml
# This scenario demonstrates AutoML flow with model quantization
# on example of Anomaly Detection in time series for MAX32960 Evaluation Kit.

automl:
  type: AutoPyTorchML
  parameters:
    output_directory: ./workspace/automl-quant-results
    time_limit: 3
    use_models:
      - PyTorchAnomalyDetectionCNN:
          # Set custom ranges for CNN parameters
          conv_stride:
            item_range: [1, 1]
          conv_dilation:
            item_range: [1, 1]
          pool_stride:
            item_range: [1, 1]
          pool_dilation:
            item_range: [1, 1]
    n_best_models: 5
    # AutoPyTorch specific options
    optimize_metric: f1
    budget_type: epochs
    min_budget: 1
    max_budget: 5
    # Increase size of the reduced dataset for quantization
    callback_max_samples: 200
    # Size of the application that will use generated models
    application_size: 75.5
    # To use GPU uncomment line below
    # use_cuda: True

optimizers:
- type: TFLiteCompiler
  parameters:
    # Quantize model to 8-bit integers
    target: int8
    compiled_model_path: ./workspace/automl-quant-results/cnn_int8.tflite
    inference_input_type: int8
    inference_output_type: int8

# Chooses MAX32690 Evaluation Kit
platform:
  type: ZephyrPlatform
```

(continues on next page)

(continued from previous page)

```
  parameters:
    name: max32690evkit/max32690/m4
    simulated: True

dataset:
  type: AnomalyDetectionDataset
  parameters:
    dataset_root: ./workspace/CATS
    csv_file: kenning:///datasets/anomaly_detection/cats_nano.csv
    split_fraction_test: 0.1
    split_seed: 12345
    inference_batch_size: 1

runtime_builder:
  type: ZephyrRuntimeBuilder
  parameters:
    output_path: ./workspace/kzr_quant_build
    workspace: ./kenning-zephyr-runtime
    venv_dir: ../.venv
    run_west_update: false
    extra_targets: [board-repl]
```

In order to avoid training models that will not fit into available space, the flow triggers quantization on initialized models and rejects ones that are too large. If model fits in the memory, the flow proceeds with training of default (non-quantized) models. In the end, it quantizes and evaluates the best models based on the training process. Such flow can be run with the following command, assuming *the scenario* is saved as `automl_quantization.yml`:

```
kenning automl optimize test report \
  --cfg ./automl_quantization.yml \
  --report-path ./workspace/automl-report/report.md \
  --allow-failures --to-html --ver INFO \
  --skip-general-information
```

6.1.4 Run sample app with a chosen model

Once the best model is selected (e.g. `workspace/automl-results/vae.0.tflite`), let's compile the sample app with it:

```
west build \
  -p always \
  -b max32690evkit/max32690/m4 app -- \
  -DEXTRA_CONF_FILE="tflite.conf" \
  -DCONFIG_KENNING_MODEL_PATH=\"$(realpath workspace/automl-results/vae.0.tflite)\
↪"
west build -t board-repl
```

Note: To use microTVM runtime, change `-DEXTRA_CONF_FILE` to `"tvm.conf"` and

`-DCONFIG_KENNING_MODEL_PATH` to a chosen model compiled with TVM (usually with `.graph_data` extension).

In the end, the app with the model can be simulated with:

```
python3 kenning-zephyr-runtime/scripts/run_renode.py --no-kcomm
```

To end the simulation, press `Ctrl+C`.

6.2 Anomaly detection model training and deployment on the MAX32690 Evaluation Kit

This example demonstrates how to train an example anomaly detection model and deploys it on an MCU using Kenning and Zephyr RTOS. The platform used in this example is the Analog Devices MAX32690 Evaluation Kit. This demo uses Kenning Zephyr Runtime and Zephyr RTOS for execution of the model on hardware.

6.2.1 Prepare an environment for development

This example uses a pre-built Docker image from Kenning Zephyr Runtime. To get started, create a Docker container with the necessary environment:

```
docker run --rm -it -v $(pwd):$(pwd) -w $(pwd) ghcr.io/antmicro/kenning-zephyr-
↪runtime:latest /bin/bash
```

Then, create a workspace directory in the container:

```
mkdir -p zephyr-workspace && cd zephyr-workspace
```

In case the MAX32690 Evaluation Kit is connected to the desktop PC using a MAX32625PICO debugger, the Docker container needs to be started in privileged mode, with UART and ACM devices forwarded to it. Assuming that the programmer is available as `/dev/ttyACM0`, and UART as `/dev/ttyUSB0` in the currently running system, run:

```
docker run --privileged --device /dev/ttyACM0 --device /dev/ttyUSB0 --rm -it -v
↪$(pwd):$(pwd) -v /dev/serial/by-id/:/dev/serial/by-id/ -w $(pwd) ghcr.io/
↪antmicro/kenning-zephyr-runtime:latest /bin/bash
```

Then, in the Docker container, clone the Kenning Zephyr Runtime repository and install the latest Zephyr SDK:

Note: Optionally, to use the latest Kenning version, install it in the image and reload the virtual environment:

```
pip install "kenning[iree,tvm,torch,anomaly_detection,tensorflow,tflite,reports,
↪renode,uart] @ git+https://github.com/antmicro/kenning.git"
```

```
git clone https://github.com/antmicro/kenning-zephyr-runtime
cd kenning-zephyr-runtime/
./scripts/prepare_zephyr_env.sh
./scripts/prepare_modules.sh
source .venv/bin/activate
```

An environment configured this way will allow you to work with Kenning and Zephyr RTOS. For more step-by-step instructions on how to set up the environment locally, see Kenning Zephyr Runtime build instructions.

Now create a `workspace` director where intermediate results of commands executed further will be stored:

```
mkdir -p workspace
```

6.2.2 Install software necessary for flashing the MAX32690 Evaluation Kit

To flash the MAX32690 Evaluation Kit, a Maxim Micros SDK is needed.

Within the Docker image, run:

```
mkdir -p /usr/share/applications
wget -O ./MaximMicrosSDK_linux.run https://github.com/analogdevicesinc/msdk/
↪releases/download/v2024_10/MaximMicrosSDK_linux.run
chmod +x ./MaximMicrosSDK_linux.run
./MaximMicrosSDK_linux.run install
export PATH=/root/MaximSDK/Tools/OpenOCD/:$PATH
```

Now follow along, answering prompts in the installation process. Once everything is installed successfully, it will be possible to flash the device using the evaluation app from Kenning Zephyr Runtime.

Note: During installation, the script may notify you that it was unable to install libncurses5.

In such a case, answer `Ignore` and proceed with the installation - it is not needed for this demo.

6.2.3 Train the sample anomaly detection model (optional)

Note: The pretrained model is available at `https://dl.antmicro.com/kenning/models/anomaly_detection/vae_cats.pth`. In Kenning, such files are obtainable using the `kenning://` scheme, e.g. `kenning:///models/anomaly_detection/vae_cats.pth`.

To skip the training step, run:

```
wget https://dl.antmicro.com/kenning/models/anomaly_detection/vae_cats.pth
wget https://dl.antmicro.com/kenning/models/anomaly_detection/vae_cats.pth.json
```

And proceed to the next section.

Once the environment is set up, the sample model can be trained. In this demo, a Variational AutoEncoder (VAE) will be used. In Kenning, there is a PytorchAnomalyDetectionVAE `ModelWrapper` encapsulating the model.

Controlled Anomalies Time Series (CATS) will be used for a dataset. It provides telemetry readings of a simulated complex dynamical system with external stimuli. It provides a representative set of time series for sensors with anomalies.

Note: Check *Specification of the kenning.datasets.anomaly_detection_dataset module* for details on the expected structure of the dataset's CSV file to understand how to create custom dataset compliant with the class.

The model can be trained using the following command:

```
kenning train test \
    --dataset-cls AnomalyDetectionDataset \
    --dataset-root ./dataset/ \
    --csv-file https://zenodo.org/records/8338435/files/data.csv \
    --modelwrapper-cls PyTorchAnomalyDetectionVAE \
    --model-path ./vae_cats.pth \
    --batch-size 256 --learning-rate 0.00002 --num-epochs 4 \
    --verbosity INFO \
    --measurements workspace/vae-torch-native.json \
    --inference-batch-size 256 \
    --batch-norm --loss-beta 0.2 --loss-capacity 0.1
```

This command:

- Downloads the CATS dataset from https://zenodo.org/records/8338435/files/data.csv (if it has not been downloaded yet)
- Creates the `AnomalyDetectionDataset` class with dataset data taken from the downloaded CSV
- Creates the `PyTorchAnomalyDetectionVAE` model wrapper encapsulating the VAE model, providing necessary methods for input preprocessing, output postprocessing, model training and more
- Trains the model using the provided batch size, learning rate, number of epochs and other exposed training parameters for the model
- Evaluates the model at the end, providing evaluation results in `./workspace/vae-torch-native.json`
- Saves the trained model as the `./vae_cats.pth` file.

6.2.4 Deploy the VAE model with TensorFlow Lite Micro

Once the VAE model is trained and available under `./vae_cats.pth`, it can be compiled with `kenning optimize` command and deployed on device using either TensorFlow Lite Micro or microTVM. This section focuses on TensorFlow Lite Micro.

Let's consider the following scenario:

```
# This scenario demonstrates deployment of Anomaly Detection in time series on␣
↪MAX32960 Evaluation Kit
# It provides two variants of execution - a deployment on an actual hardware, and␣
↪simulation in Renode
# This scenario should be executed within kenning-zephyr-runtime project, with␣
↪built evaluation app for
# TFLite Micro runtime:
#
# west build -p always -b max32690evkit/max32690/m4 app -- -DEXTRA_CONF_FILE=
↪"tflite.conf"
platform:
  type: ZephyrPlatform
  parameters:
    name: max32690evkit/max32690/m4
    # to run inference on actual hardwarem change simulated to false
    simulated: true
    zephyr_build_path: ./build/
    uart_port: /dev/ttyUSB0

# model wrapper for the VAE anomaly detection model
model_wrapper:
  type: PyTorchAnomalyDetectionVAE
  parameters:
    model_path: ./vae_cats.pth
    encoder_neuron_list: [16, 8]
    batch_norm: true

# A dataset used for evaluating the model
dataset:
  type: AnomalyDetectionDataset
  parameters:
    dataset_root: ./workspace/dataset
    csv_file: https://zenodo.org/records/8338435/files/data.csv
    split_fraction_test: 0.0005
    inference_batch_size: 1
    split_seed: 12345

# run TFLite conversion from above PyTorch model to TFLite Flatbuffers
optimizers:
  - type: TFLiteCompiler
    parameters:
      target: default
      compiled_model_path: ./workspace/vae_cats.tflite
```

(continues on next page)

(continued from previous page)

```
    inference_input_type: float32
    inference_output_type: float32
```

The scenario contents are as follows:

- `platform` - specifies the target platform where the model will be deployed. In this case, the target platform is called `max32690evkit/max32690/m4`. `simulated` boolean tells whether the board should be simulated or not.
- `model_wrapper` - provides a class that encapsulates necessary preprocessing and postprocessing functions for input and output data.
- `dataset` - provides a class implementing methods around dataset management
- `optimizers` - provides a list of optimizations that are used for optimizing and/or compiling the model.

First, compile the model using `kenning optimize`:

```
kenning optimize --cfg ./kenning-scenarios/zephyr-tflite-vae-inference-max32690.
↪yml
```

This scenario will create a `./workspace/vae_cats.tflite` file with an optimized model.

Now, to test the model in simulation or an actual hardware, the evaluation app needs to be compiled. To do so, use the `west build` command with a `tflite.conf` configuration for the selected board. However, this will build TensorFlow Lite Micro runtime with a limited set of operators. To get the runtime with just the necessary set of operators for a given model, you can provide the model that was just created.

To build the evaluation application, run:

```
west build -p always -b max32690evkit/max32690/m4 app -- \
      -DEXTRA_CONF_FILE=tflite.conf \
      -DCONFIG_KENNING_MODEL_PATH=\"./workspace/vae_cats.tflite\"
```

Once the model and evaluation app are ready, you can simulate the board in Renode with:

```
kenning test --cfg ./kenning-scenarios/zephyr-tflite-vae-inference-max32690.yml --
↪measurements workspace/vae-tflite-renode.json --verbosity INFO
```

Note: The only difference between `./kenning-scenarios/zephyr-tflite-vae-inference-max32690.yml` and `./kenning-scenarios/zephyr-tflite-vae-inference-max32690-renode.yml` is which runtime and protocol is commented out. Other parts are the same.

The produced `workspace/vae-tflite.json` is a file with raw measurements regarding the model's performance and predictions.

It can be parsed and rendered into a Markdown-based or HTML-based report using the `kenning report` command:

```
kenning report --measurements workspace/vae-tflite-renode.json --report-path␣
↪reports/vae-tflite-renode/report.md --to-html
```

Lastly, the model can be evaluated on actual MAX32690 Evaluation Kit. To do so, first flash the board with an evaluation app:

```
/root/MaximSDK/Tools/OpenOCD/openocd \
    -s /root/MaximSDK/Tools/OpenOCD/scripts/ \
    -c 'source [find interface/cmsis-dap.cfg]' \
    -c 'source [find target/max32690.cfg]' \
    -c 'init' -c 'targets' -c 'reset init' \
    -c 'flash write_image erase ./build/zephyr/zephyr.hex' \
    -c 'reset run' \
    -c 'shutdown'
```

Logs from the flashing process should look as follows to ensure successful flashing:

```
Open On-Chip Debugger (Analog Devices 0.12.0-1.0.0-7)  OpenOCD 0.12.0 (2023-09-27-
↪07:53)
Licensed under GNU GPL v2
Report bugs to <processor.tools.support@analog.com>
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: Atomic commands supported
Info : CMSIS-DAP: Test domain timer supported
Info : CMSIS-DAP: FW Version = 2.0.0
Info : CMSIS-DAP: Serial# = 0409170272c2c8d8000000000000000000000000097969906
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 0 TDO = 0 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 2000 kHz
Info : SWD DPIDR 0x2ba01477
Info : [max32xxx.cpu] Cortex-M4 r0p1 processor detected
Info : [max32xxx.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for max32xxx.cpu on 3333
Info : Listening on port 3333 for gdb connections
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
 0* max32xxx.cpu       cortex_m   little max32xxx.cpu       running

[max32xxx.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x10016372 psp: 0x2000afa0
Info : SWD DPIDR 0x2ba01477
[max32xxx.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0000fff4 msp: 0x2000a900
auto erase enabled
wrote 131072 bytes from file ./build/zephyr/zephyr.hex in 3.210366s (39.871 KiB/s)

Info : SWD DPIDR 0x2ba01477
shutdown command invoked
```

Once `./build/zephyr/zephyr.hex` is successfully written to the device, the model can be tested

directly on hardware platform.

In order to do so, set `simulated` to `false` in the `./kenning-scenarios/zephyr-tflite-vae-inference-max32690.yml` file. Finally, use a single-command approach, where `kenning test report` are invoked all at once (model is already compiled, hence lack of `optimize`):

```
kenning test report --cfg ./kenning-scenarios/zephyr-tflite-vae-inference-
↪max32690.yml \
    --measurements workspace/vae-tflite-hw.json \
    --report-path reports/vae-tflite-hw/report.md --to-html \
    --verbosity INFO
```

The `workspace/vae-tflite-hw.json` will hold the collected performance and quality data, and `reports/vae-tflite-hw/report/report.html` will demonstrate the work of the model on hardware.

6.2.5 Deploy the VAE model with microTVM

For microTVM, the deployment looks similar to the scenario above. The only difference are the optimizers and runtimes used:

```
# This scenario demonstrates deployment of Anomaly Detection in time series on␣
↪MAX32960 Evaluation Kit
# It provides two variants of execution - a deployment on an actual hardware, and␣
↪simulation in Renode
# This scenario should be executed within kenning-zephyr-runtime project, with␣
↪built evaluation app for
# microTVM runtime:
#
# west build -p always -b max32690evkit/max32690/m4 app -- -DEXTRA_CONF_FILE='tvm.
↪conf;boards/max32690evkit_max32690_m4.conf' -DKENNING_MODEL_PATH=`realpath ./
↪vae_cats.pth`
platform:
  type: ZephyrPlatform
  parameters:
    name: max32690evkit/max32690/m4
    # to run inference on actual hardwarem change simulated to false
    simulated: true
    zephyr_build_path: ./build/
    uart_port: /dev/ttyUSB0

# model wrapper for the VAE anomaly detection model
model_wrapper:
  type: PyTorchAnomalyDetectionVAE
  parameters:
    model_path: ./vae_cats.pth
    encoder_neuron_list: [16, 8]
    batch_norm: true

# A dataset used for evaluating the model
```

(continues on next page)

(continued from previous page)

```
dataset:
  type: AnomalyDetectionDataset
  parameters:
    dataset_root: ./workspace/dataset
    csv_file: https://zenodo.org/records/8338435/files/data.csv
    split_fraction_test: 0.0005
    inference_batch_size: 1
    split_seed: 12345

# run TVM conversion from above PyTorch model
optimizers:
- type: TVMCompiler
  parameters:
    compiled_model_path: ./workspace/vae_cats.graph_data
```

To begin evaluation, compile the evaluation app using microTVM as the runtime:

```
west build -p always -b max32690evkit/max32690/m4 app -- -DEXTRA_CONF_FILE='tvm.
↪conf;boards/max32690evkit_max32690_m4.conf'
```

Then, run:

```
kenning optimize test report --cfg ./kenning-scenarios/zephyr-tvm-vae-inference-
↪max32690.yml \
    --measurements workspace/vae-tvm-renode.json \
    --report-path reports/vae-tvm-renode/report.md --to-html \
    --verbosity INFO
```

This performs all actions at once - model optimization, model evaluation and report generation.

To test the model on hardware, set `simulated` in the scenario to `false` and then flash the device with the microTVM-based app:

```
/root/MaximSDK/Tools/OpenOCD/openocd \
    -s /root/MaximSDK/Tools/OpenOCD/scripts/ \
    -c 'source [find interface/cmsis-dap.cfg]' \
    -c 'source [find target/max32690.cfg]' \
    -c 'init' -c 'targets' -c 'reset init' \
    -c 'flash write_image erase ./build/zephyr/zephyr.hex' \
    -c 'reset run' \
    -c 'shutdown'
```

Run tests on device (`optimize` is not necessary, since compilation was done before simulation in Renode):

```
kenning test report --cfg ./kenning-scenarios/zephyr-tvm-vae-inference-max32690.
↪yml \
    --measurements workspace/vae-tvm-hw.json \
    --report-path reports/vae-tvm-hw/report.md --to-html \
    --verbosity INFO
```

6.2.6 Comparing performance and quality of runtimes and models

Based on the results for microTVM and TensorFlow Lite Micro runtimes, you can generate comparison reports that will show the differences in performance and quality between the two.

To generate a report for Renode simulations, run:

```
kenning report --measurements \
    workspace/vae-tflite-renode.json \
    workspace/vae-tvm-renode.json \
    --report-path reports/vae-tflite-tvm-comparison-renode/report.md --to-html \
    --verbosity INFO
```

For comparison of execution on hardware, run:

```
kenning report --measurements \
    workspace/vae-tflite-hw.json \
    workspace/vae-tvm-hw.json \
    --report-path reports/vae-tflite-tvm-comparison-hw/report.md --to-html \
    --verbosity INFO
```

The reports in HTML format will be available here:

- `reports/vae-tflite-tvm-comparison-renode/report/report.html`
- `reports/vae-tflite-tvm-comparison-hw/report/report.html`

6.2.7 Specification of the kenning.datasets.anomaly_detection_dataset module

Module description

Dataset wrapper for anomaly detection in time series.

Class AnomalyDetectionDataset

Generic dataset for anomaly detection in time series problem.

It reads data from provided CSV file and prepares sequences of data.

CSV file has to follow the schema:

Timestamp column	Param 1 name	Param 2 name	...	Param N name	Label
timestamps	Numerical values	Numerical values	...	Numerical values	Integer values

Kenning automatically discards the timestamp column, as well as the header row.

The numerical values of parameters are used as signals or data from sensors, whereas the labels specify anomaly occurrence (values greater than 0).

Each label describes whether an anomaly has been observed within `window_size` previous samples.

This results with final version of dataset where one entry looks like:

X			Y
Param 1 value from `t - window_size + 1`	...	Param N value from `t - window_size + 1`	
...	...	...	
Param 1 value from `t - 1`	...	Param N value from `t - 1`	
Param 1 value from `t`	...	Param N value from `t`	0 (no anomaly) or 1 (anomaly)

6.3 Displaying information about available classes

The Kenning project provides several scripts for assessing information about classes (such as *Dataset*, *ModelWrapper*, *Optimizer*).

Below, we provide an overview of means to display this information.

First, make sure that Kenning is installed:

```
pip install "kenning @ git+https://github.com/antmicro/kenning.git"
```

6.3.1 Kenning list

`kenning list` lists all available classes, grouping them by the base class (group of modules).

The script can be executed as follows:

```
kenning list
```

This will return a list similar to the one below:

```
Optimizers (in kenning.optimizers):

    kenning.optimizers.onnx.ONNXCompiler
    kenning.optimizers.tensorflow_optimizers.TensorFlowOptimizer
    kenning.optimizers.tvm.TVMCompiler
    kenning.optimizers.iree.IREECompiler
    kenning.optimizers.tensorflow_pruning.TensorFlowPruningOptimizer
    kenning.optimizers.tensorflow_clustering.TensorFlowClusteringOptimizer
    kenning.optimizers.nni_pruning.NNIPruningOptimizer
    kenning.optimizers.tflite.TFLiteCompiler
    kenning.optimizers.model_inserter.ModelInserter

Datasets (in kenning.datasets):

    kenning.datasets.random_dataset.RandomizedClassificationDataset
```

(continues on next page)

(continued from previous page)

```
    kenning.datasets.coco_dataset.COCODataset2017
    kenning.datasets.open_images_dataset.OpenImagesDatasetV6
    kenning.datasets.helpers.detection_and_segmentation.
↪ObjectDetectionSegmentationDataset
    kenning.datasets.magic_wand_dataset.MagicWandDataset
    kenning.datasets.common_voice_dataset.CommonVoiceDataset
    kenning.datasets.pet_dataset.PetDataset
    kenning.datasets.random_dataset.RandomizedDetectionSegmentationDataset
    kenning.datasets.imagenet_dataset.ImageNetDataset
    kenning.datasets.visual_wake_words_dataset.VisualWakeWordsDataset

Modelwrappers (in kenning.modelwrappers):

    kenning.modelwrappers.instance_segmentation.pytorch_coco.PyTorchCOCOMaskRCNN
    kenning.modelwrappers.object_detection.darknet_coco.TVMDarknetCOCOYOLOV3
    kenning.modelwrappers.instance_segmentation.yolact.YOLACTWithPostprocessing
    kenning.modelwrappers.classification.tensorflow_imagenet.TensorFlowImageNet
    kenning.modelwrappers.instance_segmentation.yolact.YOLACTWrapper
    kenning.modelwrappers.object_detection.yolo_wrapper.YOLOWrapper
    kenning.modelwrappers.frameworks.tensorflow.TensorFlowWrapper
    kenning.modelwrappers.classification.tflite_magic_wand.MagicWandModelWrapper
    kenning.modelwrappers.classification.tflite_person_detection.
↪PersonDetectionModelWrapper
    kenning.modelwrappers.instance_segmentation.yolact.YOLACT
    kenning.modelwrappers.frameworks.pytorch.PyTorchWrapper
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2
    kenning.modelwrappers.object_detection.yolov4.ONNXYOLOV4
    kenning.modelwrappers.classification.pytorch_pet_dataset.
↪PyTorchPetDatasetMobileNetV2

...
```

The output of the command can be limited by providing one or more positional arguments representing module groups:

- `optimizers`,
- `runners`,
- `dataproviders`,
- `datasets`,
- `modelwrappers`,
- `onnxconversions`,
- `outputcollectors`,
- `runtimes`.

The command can also be used to list available runtimes:

```
kenning list runtimes
```

Which will return a list similar to the one below::

```
Runtimes (in kenning.runtimes):

    kenning.runtimes.iree.IREERuntime
    kenning.runtimes.tflite.TFLiteRuntime
    kenning.runtimes.pytorch.PyTorchRuntime
    kenning.runtimes.tvm.TVMRuntime
    kenning.runtimes.onnx.ONNXRuntime
```

More verbose information is available with `-v` and `-vv` flags. They will display dependencies, descriptions and other information for each class.

6.3.2 Kenning info

`kenning info` displays more detailed information about a particular class. This information is especially useful when creating JSON scenario configurations. The command displays the following:

- docstrings
- dependencies, along with information on availability in the current Python environment
- supported input and output formats
- argument structure used in JSON

Let's consider a scenario where we want to compose a Kenning flow utilizing a YOLOv4 ModelWrapper. Execute the following command:

```
kenning info kenning.modelwrappers.object_detection.yolov4.ONNXYOLOV4
```

This will display all the necessary information about the class:

```
Class: ONNXYOLOV4

Input/output specification:
* input
  * shape: (1, 3, keyparams['width'], keyparams['height'])
  * dtype: float32
* output
  * shape: (1, 255, (keyparams['width'] // (8 * (2 ** 0))), (keyparams['height'] /
↪/ (8 * (2 ** 0))))
  * dtype: float32
* output.3
  * shape: (1, 255, (keyparams['width'] // (8 * (2 ** 1))), (keyparams['height'] /
↪/ (8 * (2 ** 1))))
  * dtype: float32
* output.7
  * shape: (1, 255, (keyparams['width'] // (8 * (2 ** 2))), (keyparams['height'] /
```

(continues on next page)

(continued from previous page)

```
↪/ (8 * (2 ** 2))))
  * dtype: float32
* detection_output
  * type: List[DetectObject]

Dependencies:
* torch
* numpy
* onnx
* torch.nn.functional

Arguments specification:
* classes
  * argparse_name: --classes
  * convert-type: builtins.str
  * type
    * string
  * description: File containing Open Images class IDs and class names in CSV␣
↪format to use (can be generated using kenning.scenarios.open_images_classes_
↪extractor) or class type
  * default: coco
* model_path
  * argparse_name: --model-path
  * convert-type: kenning.utils.resource_manager.ResourceURI
  * type
    * string
  * description: Path to the model
  * required: True
```

Note: By default, the command only performs static code analysis. For example, some values in the input/output specification are expressions because the command did not import or evaluate any values. This is done to allow for missing dependencies.

Loading a class with arguments

To gain access to more detailed information, the `--load-class-with-args` argument can be used. Provided that all dependencies are satisfied, the script will load the verified module to collect more detailed information about available settings.

In the example above, the ONNXYOLOV4 configuration specifies that the `model_path` argument is required. All dependencies are available as there is no warning message.

To load a class with arguments, run this command:

```
kenning info kenning.modelwrappers.object_detection.yolov4.ONNXYOLOV4 \
  --load-class-with-args \
  --model-path kenning:///models/detection/yolov4.onnx
```

```
Class: ONNXYOLOV4

Input/output specification:
* input
  * shape: (1, 3, 608, 608)
  * dtype: float32
* output
  * shape: (1, 255, 76, 76)
  * dtype: float32
* output.3
  * shape: (1, 255, 38, 38)
  * dtype: float32
* output.7
  * shape: (1, 255, 19, 19)
  * dtype: float32
* detection_output
  * type: List[DetectObject]

Dependencies:
* onnx
* numpy
* torch.nn.functional
* torch

Arguments specification:
* classes
  * argparse_name: --classes
  * convert-type: builtins.str
  * type
    * string
  * description: File containing Open Images class IDs and class names in CSV␣
↪format to use (can be generated using kenning.scenarios.open_images_classes_
↪extractor) or class type
  * default: coco
* model_path
  * argparse_name: --model-path
  * convert-type: kenning.utils.resource_manager.ResourceURI
  * type
    * string
  * description: Path to the model
  * required: True
```

6.4 Evaluating models on hardware using Kenning Zephyr Runtime

This section contains tutorial of evaluating models on microcontrollers using Kenning Zephyr Runtime and Renode.

6.4.1 Preparing the Zephyr environment

First, we need to setup environment for building the runtime. Start with installing dependencies:

- Zephyr dependencies
- `jq`
- `curl`
- `west`
- `CMake`

On Debian-based Linux distributions, the above-listed dependencies can be installed as follows:

```
apt update

apt install -y --no-install-recommends ccache curl device-tree-compiler dfu-util␣
↪file \
  g++-multilib gcc gcc-multilib git jq libmagic1 libsdl2-dev make ninja-build \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel python3-
↪venv \
  mono-complete wget xxd xz-utils patch
```

Next, create a Zephyr workspace directory and clone there Kenning Zephyr Runtime repository:

```
mkdir -p zephyr-workspace && cd zephyr-workspace
git clone https://github.com/antmicro/kenning-zephyr-runtime
cd kenning-zephyr-runtime
```

Then, initialize Zephyr workspace, ensure that latest Zephyr SDK is installed, and prepare a Python's virtual environment with:

```
./scripts/prepare_zephyr_env.sh
source .venv/bin/activate
```

Finally, prepare additional modules:

```
./scripts/prepare_modules.sh
```

6.4.2 Installing Kenning with Renode support

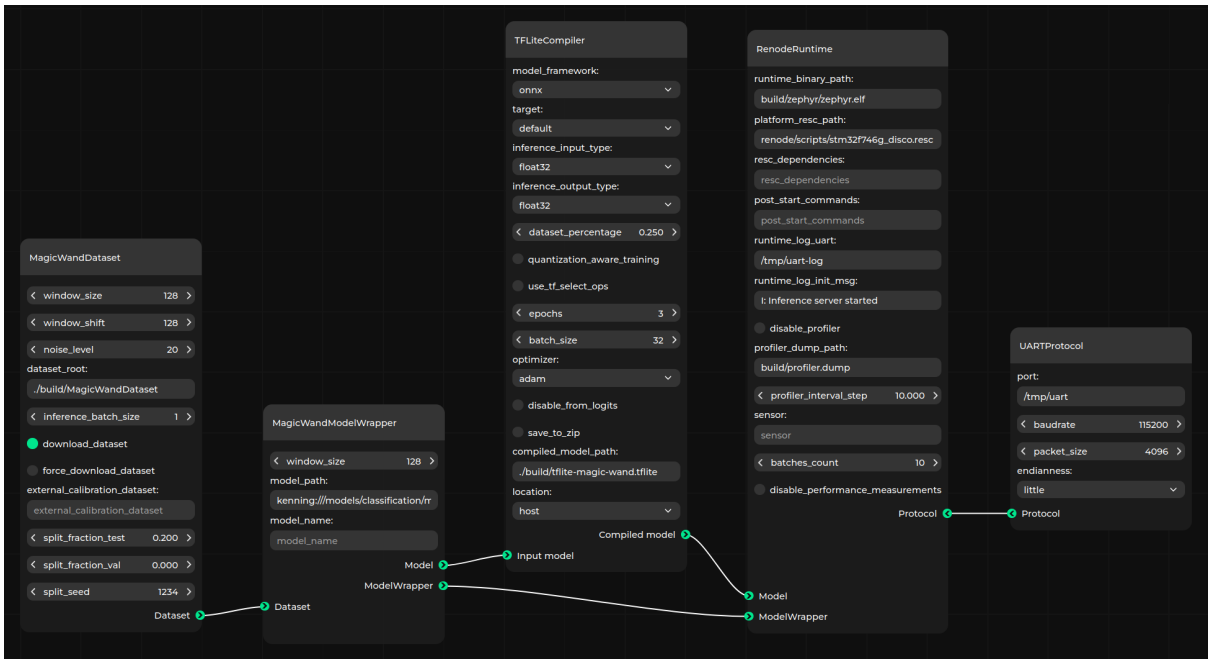
Evaluating models using Kenning Zephyr Runtime requires Kenning with Renode support. Use `pip` to install it:

```
pip install --upgrade pip
pip install "kenning[tvm,tensorflow,reports,renode] @ git+https://github.com/
↪antmicro/kenning.git"
```

To use Renode, either follow Renode documentation or download a package for Renode and set `PYRENODE_PKG` variable for pyrenode3 package:

```
wget https://builds.renode.io/renode-latest.pkg.tar.xz
export PYRENODE_PKG=$(realpath renode-latest.pkg.tar.xz)
```

6.4.3 Building and evaluating Magic Wand model using TFLite backend



Let's build the Kenning Zephyr Runtime with TFLiteMicro as model executor for `stm32f746g_disco` board. Run:

```
west build --board stm32f746g_disco app -- -DEXTRA_CONF_FILE=tflite.conf
```

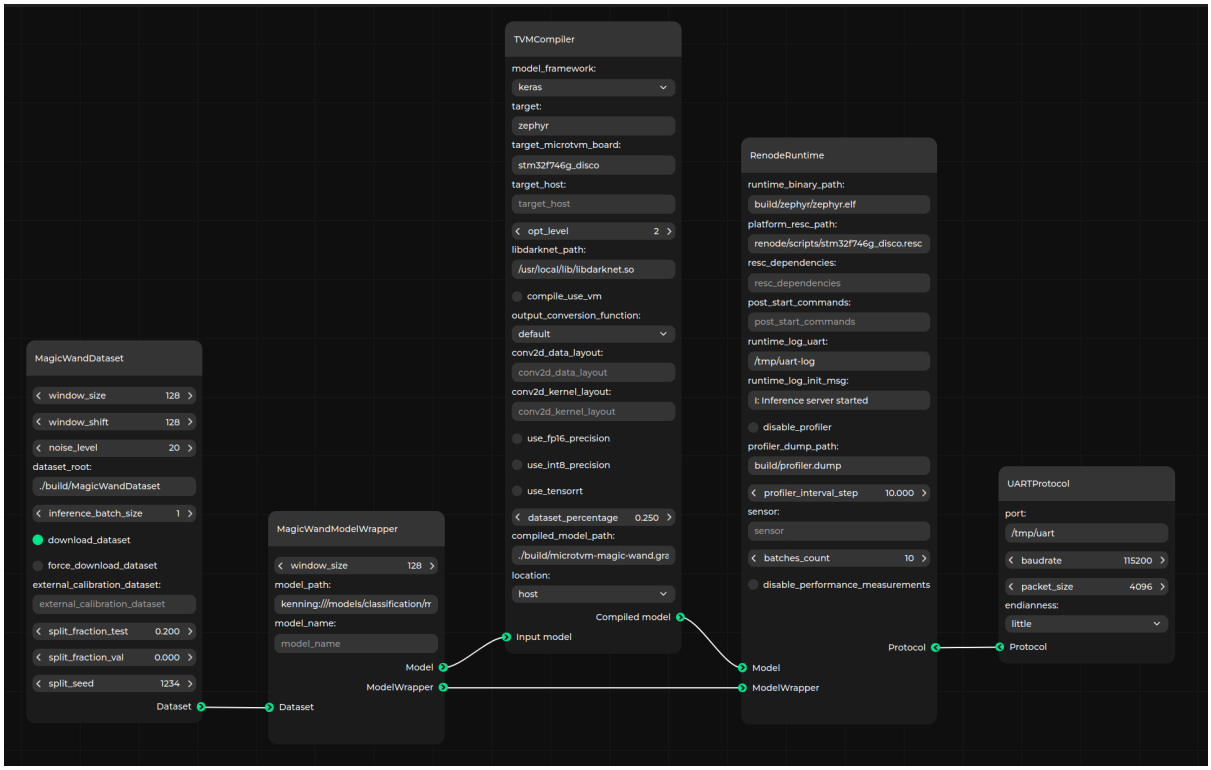
The built binary can be found in `build/zephyr/zephyr.elf`.

To evaluate the Magic Wand model using built runtime, run:

```
kenning optimize test \
    --cfg kenning-scenarios/renode-zephyr-tflite-magic-wand-inference.yml \
    --measurements build/zephyr-stm32-tflite-magic-wand.json --verbosity INFO \
    --verbosity INFO
```

The evaluation results would be saved at `build/zephyr-stm32-tflite-magic-wand.json`.

6.4.4 Building and evaluating Magic Wand model using microTVM backend



Building the Kenning Zephyr Runtime with microTVM support for the same board requires changing only `-DEXTRA_CONF_FILE` value. Run:

```
west build --board stm32f746g_disco app -- -DEXTRA_CONF_FILE=tvm.conf
```

And now, as previously, run evaluation using Kenning:

```
kenning optimize test \
    --cfg kenning-scenarios/renode-zephyr-tvm-magic-wand-inference.yml \
    --measurements build/zephyr-stm32-tvm-magic-wand.json --verbosity INFO \
    --verbosity INFO
```

The evaluation results would be saved at `zephyr-stm32-tvm-magic-wand.json`.

6.4.5 Comparing the results

To generate comparison report run:

```
kenning report \
    --measurements \
      build/zephyr-stm32-tflite-magic-wand.json \
      build/zephyr-stm32-tvm-magic-wand.json \
    --report-path build/zephyr-stm32-tflite-tvm-comparison.md \
    --report-types renode_stats performance classification \
    --to-html
```

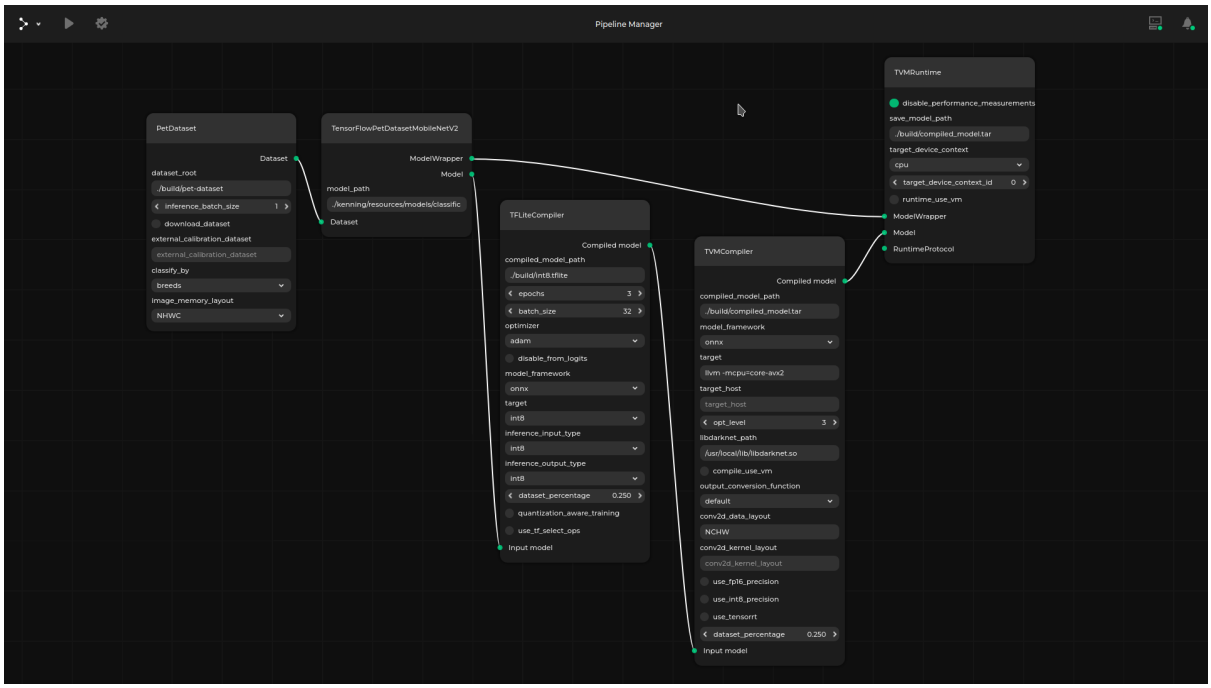
The HTML version of the report should be saved at `build/zephyr-stm32-tflite-tvm-comparison/zephyr-stm32-tflite-tvm-comparison.html`.

6.5 Visualizing Kenning data flows with Pipeline Manager

Pipeline Manager is a GUI tool that helps visualize and edit data flows.

This chapter describes how to set up Pipeline Manager and use it with Kenning graphs.

Pipeline Manager is application-agnostic and does not assume any properties of the application it is working with. Kenning, however, implements a Pipeline Manager client which provides tools for creating complex Kenning pipelines and flows, while also allowing for running and saving these configurations directly from Pipeline Manager's editor.



6.5.1 Installing Pipeline Manager

Kenning requires extra dependencies in order to run the Pipeline Manager integration. To install them, run:

```
pip install "kenning[pipeline_manager] @ git+https://github.com/antmicro/kenning.
↪git"
```

6.5.2 Running Pipeline Manager with Kenning

Start the Pipeline Manager client with:

```
kenning visual-editor --file-path measurements.json --workspace-dir ./workspace
```

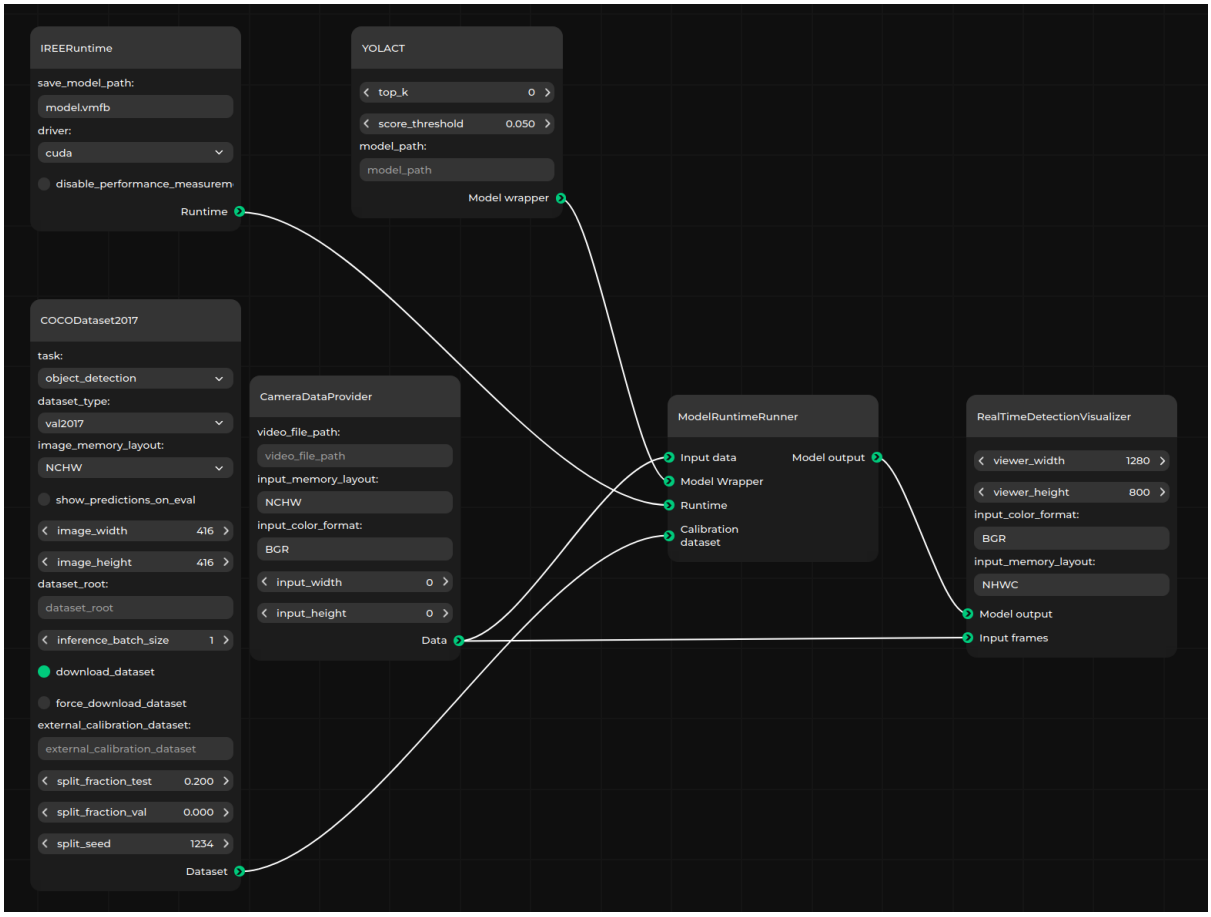
The `--file-path` option specifies where the results of model benchmarking or the runtime data will be stored.

For runtime data, the following arguments are available:

- `--spec-type` - type of Kenning scenario to be run, can be either `pipeline` (for *optimization and deployment pipeline*) or `flow` (for creating *runtime scenarios*). `pipeline` is the default type.
- `--host` - Pipeline Manager server address, default: `127.0.0.1`
- `--port` - Pipeline Manager server port, default: `9000`
- `--verbosity` - log verbosity

6.5.3 Using Pipeline Manager

In its default configuration, the web application is available under `http://127.0.0.1:5000/`.



Below, you can find an example Pipeline Manager workflow:

- `Load File` - option available from the drop-down menu on the top left, loads a JSON configuration describing a Kenning scenario.

  For instance, `scripts/jsonconfigs/sample-tflite-pipeline.json` available in Kenning is a basic configuration for an *Kenning example use case for benchmarking using a native framework*.

- Graph editing - adding or removing nodes, editing connections, node options, etc.
- `Validate` - validates and returns the information whether the scenario is valid.

  For example, it will return an error when two optimizers in a chain are incompatible with each other.

- `Run` - creates and runs the optimization pipeline or *Kenning runtime flow*.
- `Save file` - saves current JSON scenario to a specified path.

More information about how to work with Pipeline Manager is available in the Pipeline Manager documentation.

6.6 Bare-metal IREE runtime simulated using Renode

This tutorial shows how to compile models with IREE and evaluate them using the Kenning bare-metal IREE runtime working in Renode.

The platform used for testing is a Springbok RISC-V AI accelerator with V Extension. For more details, check:

- Co-developing RISC-V AI solutions using Vector Extensions in Renode with Kenning bare-metal runtime and IREE blog note
- Kenning bare-metal IREE Runtime repository

6.6.1 Setup

For quick setup, use `docker` to open the environment with all necessary dependencies installed:

```
docker run --rm -v $(pwd):$(pwd) -w $(pwd) -it ghcr.io/antmicro/kenning-bare-
↪metal-iree-runtime:latest
```

Let's also create a `workspace` directory to store results of the simulation, as well as Kenning reports:

```
mkdir -p workspace && cd workspace
```

Alternatively, install them manually by following the instructions in the runtime's README.md:

Kenning uses pyrenode3 package, which requires Renode to work. To install it, download the latest Renode Arch package and store its location in `PYRENODE_PKG`:

```
wget https://builds.renode.io/renode-latest.pkg.tar.xz
export PYRENODE_PKG=$(realpath renode-latest.pkg.tar.xz)
```

For other configuration options check pyrenode3 README.md.

6.6.2 Evaluating the model in Kenning

Kenning can evaluate the runtime running on a device simulated in Renode. This allows us to:

- Analyze model behavior without the need for physical hardware
- Check model and runtime performance and quality on a simulated device in Continuous Integration pipelines
- Obtain detailed metrics regarding device usage (e.g. histogram of instructions)

Creating the scenario

The following scenario is used for evaluating the model on Springbok AI accelerator in Renode:

```
{
    "platform": {
        "type": "BareMetalPlatform",
        "parameters": {
            "name": "rv32-springbok",
            "simulated": true,
            "profiler_dump_path": "build/profiler.dump",
            "uart_port": "/tmp/uart"
        }
    },
    "dataset": {
        "type": "kenning.datasets.magic_wand_dataset.MagicWandDataset",
        "parameters": {
            "dataset_root": "./build/MagicWandDataset"
        }
    },
    "model_wrapper": {
        "type": "kenning.modelwrappers.classification.tflite_magic_wand.
↪MagicWandModelWrapper",
        "parameters": {
            "model_path": "kenning:///models/classification/magic_wand.h5"
        }
    },
    "optimizers": [
        {
            "type": "kenning.optimizers.iree.IREECompiler",
            "parameters": {
                "compiled_model_path": "./build/tflite-magic-wand.vmfb",
                "backend": "llvm-cpu",
                "model_framework": "keras",
                "compiler_args": [
                    "iree-llvm-debug-symbols=false",
                    "iree-vm-bytecode-module-strip-source-map=true",
                    "iree-vm-emit-polyglot-zip=false",
                    "iree-llvm-target-triple=riscv32-pc-linux-elf",
                    "iree-llvm-target-cpu=generic-rv32",
                    "iree-llvm-target-cpu-features=+m,+f,+zvl512b,+zve32x,+zve32f
↪",
                    "iree-llvm-target-abi=ilp32"
                ]
            }
        }
    ]
}
```

The model used in the scenario is a classifier trained on Magic Wand dataset for accelerometer based gesture recognition:

- `dataset` entry provides a class for managing the Magic Wand dataset (downloading the dataset, parsing data from files, an evaluating the model on Springbok by sending inputs and comparing outputs to ground truth).
- `model_wrapper` entry provides the model to optimize, I/O specification and model-specific processing.

IREE compiler is enabled by adding it to the `optimizers`. It optimizes and compiles the model to Virtual Machine Flat Buffer format (`.vmfb`), which is later executed on a minimal IREE virtual machine. The additional flags provided to the compiler specify the RISC-V target architecture and V Extension features compatible with the Springbok accelerator.

Renode simulation is enabled by specifying the `simulated` in the `platform`'s parameters' and setting the following parameters:

- `name` - name of the simulated board (int this case `stm32f746g_disco`)
- `runtime_binary_path` - path to the runtime binary (in this case IREE runtime)
- `platform_resc_path` - path to the Renode script (`.resc` file) used for setting up the emulation
- `resc_dependencies` - additional dependencies for RESC files
- `post_start_commands` - Renode monitor commands executed after emulation starts
- `profiler_dump_path` - path for the profiler dump, which is then used for report generation

UART is used for Kenning communications by selecting `UARTProtocol` in `protocol` entry.

Running the scenario

Evaluate the model in Renode using the created scenario, and generate a report with performance and quality metrics:

```
kenning optimize test report \
    --json-cfg https://raw.githubusercontent.com/antmicro/kenning/main/scripts/
↪jsonconfigs/renode-magic-wand-iree-bare-metal-inference.json \
    --measurements ./results.json \
    --report-path ./reports/springbok-magic-wand.md \
    --report-name v-extensions-riscv \
    --model-names magic_wand_fp32 \
    --verbosity INFO \
    --to-html
```

Kenning executes the scenario in the following steps:

- Kenning loads the model with Keras and compiles it with IREE to `./build/magic-wand.vmfb`
- The machine with the Springbok AI accelerator is created in Renode
- Connection to the simulated device is established via UART
- Kenning sends the compiled model and I/O specification
- Input data is sent in a loop, the bare-metal IREE runtime performs inference and sends the results back

When the scenario is finished Kenning stores runtime metrics, profiling results and model evaluation in `results.json`.

The human readable report generated from `results.json` will be available under `reports/springbok-magic-wand/report.md`.

6.7 Structured pruning for PyTorch models

Structured pruning is of the methods for reducing model size, which removes the least contributing neurons, filters and/or convolution kernels. In this example, we present scenarios for structured pruning of `PyTorchPetDatasetMobileNetV2` using Neural Network Intelligence.

6.7.1 Setup

Install required dependencies:

```
pip install "kenning[nni,reports] @ git+https://github.com/antmicro/kenning.git"
```

6.7.2 Experiments

In order to compare with an original model, you need to execute a scenario:

Listing 6.2: mobilenetv2-pytorch.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.pytorch_pet_dataset.
↪PyTorchPetDatasetMobileNetV2",
    "parameters": {
      "model_path": "kenning:///models/classification/pytorch_pet_dataset_
↪mobilenetv2_full_model.pth",
      "model_name": "torch-native"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset",
      "image_memory_layout": "NCHW"
    }
  },
  "optimizers": [],
  "runtime": {
    "type": "kenning.runtimes.pytorch.PyTorchRuntime",
    "parameters": {
      "save_model_path": "kenning:///models/classification/pytorch_pet_dataset_
↪mobilenetv2_full_model.pth"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

To run it, use this command:

```
kenning test \
  --json-cfg mobilenetv2-pytorch.json \
  --measurements build/torch.json
```

Kenning supports activation-based pruners. You can choose a specific pruner with the `pruner_type` parameter:

- `apoz` - `ActivationAPoZRankPruner` based on Average Percentage of Zeros in activations,
- `mean_rank` - `ActivationMeanRankPruner` based on a metric that calculates the smallest mean value of activations.

These activations are collected during dataset inference. The number of samples collected for statistics can be modified with `training_steps`. Moreover, pruning has two modes:

- `dependency_aware` - makes pruner aware of dependencies for channels and groups.
- `normal` - dependencies are ignored.

You can also choose which `activation` the pruner will use - `relu`, `relu6` or `gelu`. Additional configuration can be specified in `config_list` which follows a format defined in the NNI specification. When `exclude_last_layer` is positive, the optimizer will be configured to exclude the last layer from the pruning process, to prevent changing the size of the output. Apart from that, `confidence` defines the coefficient for sparsity inference and batch size of the dummy input for the process. When a GPU is available, it is used by default, but as pruning can be memory-consuming, the `pruning_on_cuda` option enables manual GPU usage configuration during the process.

Other arguments affect fine-tuning of the pruned model, e.g. `criterion` and `optimizer` accept paths to classes, respectively calculating a criterion and optimizing a neural network. The number of `finetuning_epochs`, the `finetuning_batch_size` and `finetuning_learning_rate` can be modified.

Listing 6.3: pruning-mobilenetv2-pytorch.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.pytorch_pet_dataset.
↪PyTorchPetDatasetMobileNetV2",
    "parameters": {
      "model_path": "kenning:///models/classification/pytorch_pet_dataset_
↪mobilenetv2_full_model.pth",
      "model_name": "nni-pruning-0_05"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
```

(continues on next page)

(continued from previous page)

```
      "dataset_root": "./build/PetDataset",
      "image_memory_layout": "NCHW"
    }
  },
  "optimizers": [
    {
      "type": "kenning.optimizers.nni_pruning.NNIPruningOptimizer",
      "parameters": {
        "pruner_type": "mean_rank",
        "config_list": [
          {
            "total_sparsity": 0.05,
            "op_types": [
              "Conv2d",
              "Linear"
            ]
          }
        ],
        "training_steps": 16,
        "activation": "relu",
        "compiled_model_path": "build/nni-pruning-0_05.pth",
        "mode": "dependency_aware",
        "finetuning_epochs": 3,
        "finetuning_batch_size": 64,
        "confidence": 8,
        "criterion": "torch.nn.CrossEntropyLoss",
        "optimizer": "torch.optim.Adam",
        "pruning_on_cuda": true,
        "finetuning_learning_rate": 0.00005,
        "exclude_last_layer": true
      }
    }
  ],
  "runtime": {
    "type": "kenning.runtimes.pytorch.PyTorchRuntime",
    "parameters": {
      "save_model_path": "./build/nni-pruning-0_05.pth"
    }
  }
}
```

Run the above scenario with:

```
kenning optimize test \
  --json-cfg pruning-mobilenetv2-pytorch.json \
  --measurements build/nni-pruning.json
```

To ensure better quality of performance measurements, we suggest running optimization and tests separately, like below:

```
kenning optimize --json-cfg pruning-mobilenetv2-pytorch.json
kenning test \
  --json-cfg pruning-mobilenetv2-pytorch.json \
  --measurements build/nni-pruning.json
```

For more size reduction, you can use larger sparsity with adjusted parameters, like below:

```
{
  "type": "kenning.optimizers.nni_pruning.NNIPruningOptimizer",
  "parameters": {
    "pruner_type": "mean_rank",
    "config_list": [
      {
        "total_sparsity": 0.15,
        "op_types": [
          "Conv2d",
          "Linear"
        ]
      }
    ],
    "training_steps": 92,
    "activation": "relu",
    "compiled_model_path": "build/nni-pruning-0_15.pth",
    "mode": "dependency_aware",
    "finetuning_epochs": 10,
    "finetuning_batch_size": 64,
    "confidence": 8,
    "criterion": "torch.nn.CrossEntropyLoss",
    "optimizer": "torch.optim.Adam",
    "pruning_on_cuda": true,
    "finetuning_learning_rate": 1e-04,
    "exclude_last_layer": true
  }
}
```

6.7.3 Results

Models can be compared with the generated report:

```
kenning report \
  --measurements \
    build/torch.json \
    build/nni-pruning.json \
  --report-path build/nni-pruning.md \
  --to-html
```

Pruned models have significantly fewer parameters, which results in decreased GPU and VRAM usage without increasing inference time. Summary of a few examples with different sparsity:

Sparsity	Accuracy	Number of parameters	Fine-tuning epochs	Size reduction
—	0.9632653061	4,130,853	—	0.00%
0.05	0.9299319728	3,660,421	3	11.27%
0.15	0.8102040816	2,813,380	10	31.61%

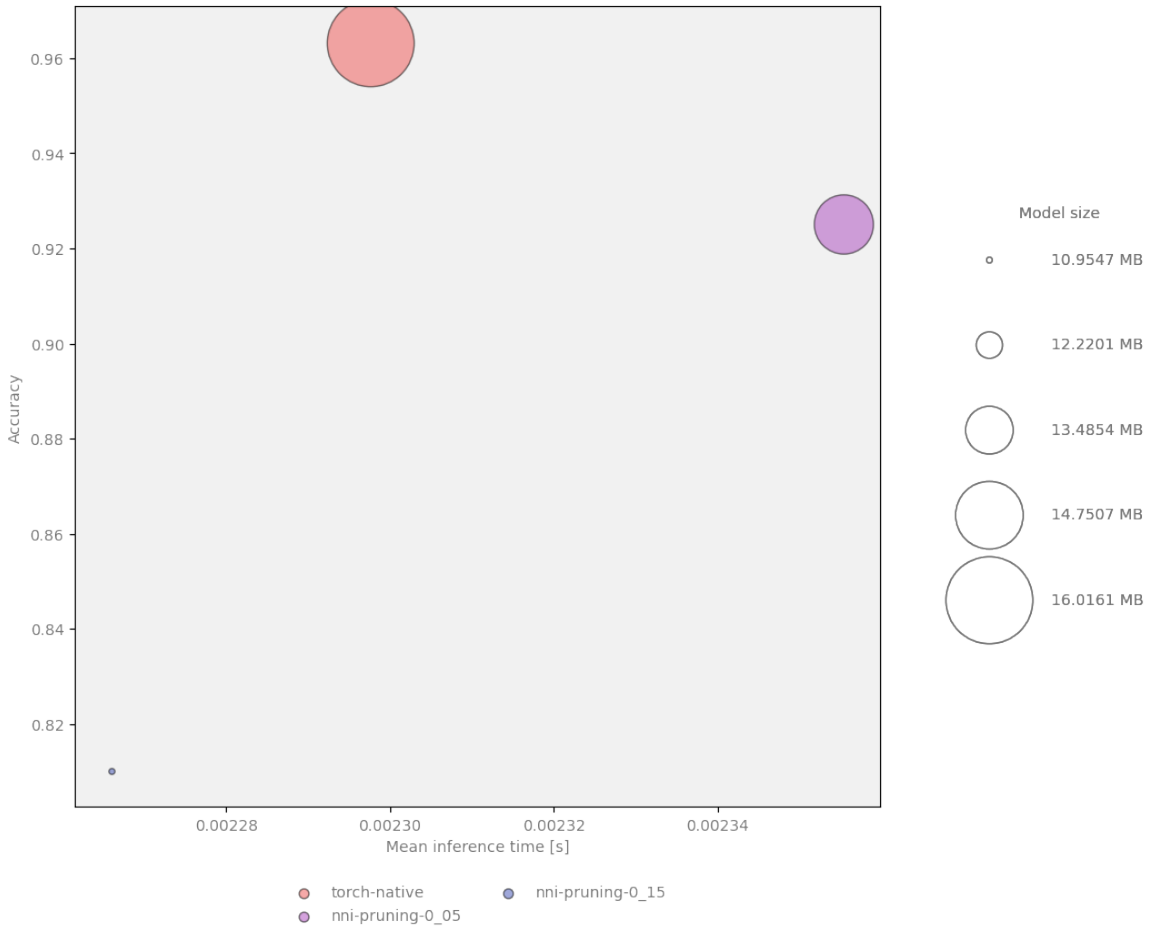


Figure 6.1: Model size, speed and quality comparison for NNI pruning

6.8 Model quantization and compilation using TFLite and TVM

Let's consider a simple scenario where we want to optimize the inference time and memory usage of a classification model executed on a x86 CPU.

To do this, we are going to use the `PetDataset` Dataset and the `TensorFlowPetDatasetMobileNetV2` ModelWrapper.

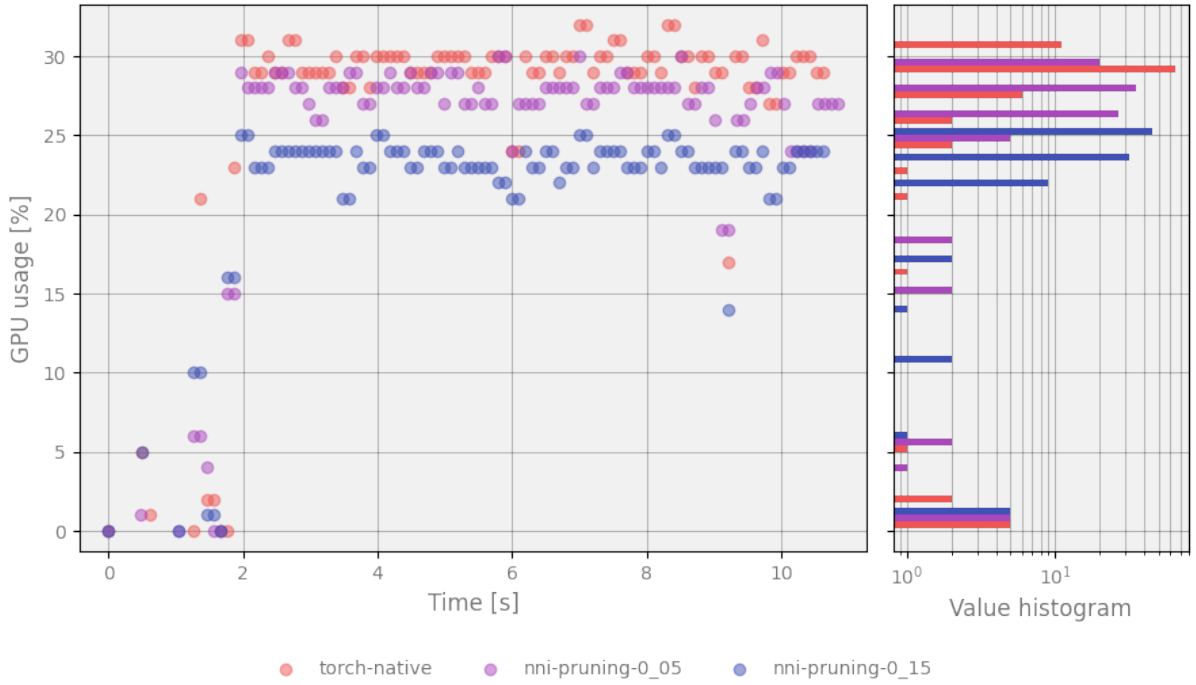


Figure 6.2: Plot represents changes of GPU usage over time

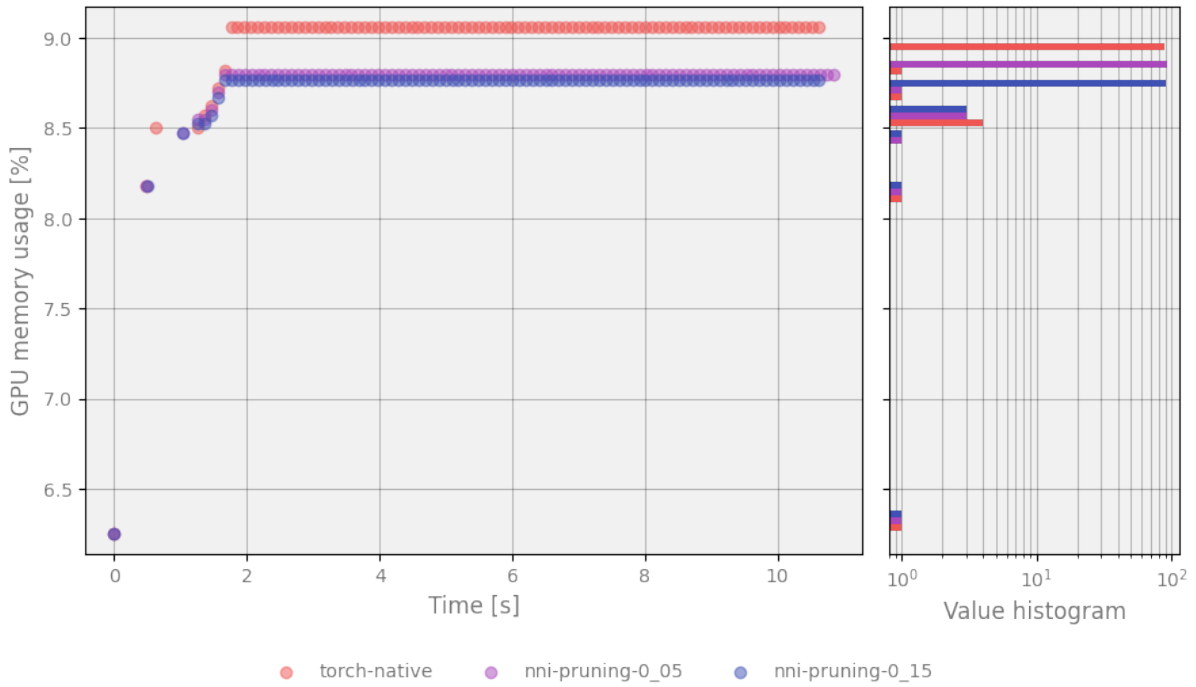


Figure 6.3: Plot represents changes of GPU RAM usage over time

6.8.1 Prepare environment

First of all, we need to install all necessary dependencies:

```
pip install "kenning[tensorflow,tflite,tvm,reports] @ git+https://github.com/
↪antmicro/kenning.git"
```

6.8.2 Train the model (optional)

We will skip the training process, we will use tensorflow_pet_dataset_mobilenetv2.h5. In Kenning, available models and resources can be downloaded using URIs with the `kenning://` scheme, in this case `kenning:///models/classification/tensorflow_pet_dataset_mobilenetv2.h5`.

The training of the above model can be performed using the following command:

```
kenning train \
    --modelwrapper-cls kenning.modelwrappers.classification.tensorflow_pet_
↪dataset.TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls kenning.datasets.pet_dataset.PetDataset \
    --logdir build/logs \
    --dataset-root build/pet-dataset \
    --model-path build/trained-model.h5 \
    --batch-size 32 \
    --learning-rate 0.0001 \
    --num-epochs 50
```

6.8.3 Benchmarking a model using a native framework

First, we want to check how the trained model performs using the native framework on CPU.

For this, we will use the `kenning test` tool. The tool is configured with JSON files (scenarios).

In our case, the JSON file (named `native.json`) will look like this:

Listing 6.4: mobilenetv2-tensorflow-native.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_name": "native",
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5",
      "batch_size": 32,
      "learning_rate": 0.0001,
      "num_epochs": 50,
      "logdir": "build/logs"
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
  }
}
```

This JSON provides a configuration for running the model natively and evaluating it against a defined Dataset.

For every class in the JSON file above, two keys are required: `type` which is a module path of our class and `parameters` which is used to provide arguments used to create instances of our classes.

In `model_wrapper`, we specify the model used for evaluation - here it is MobileNetV2 trained on PetDataset. The `model_path` is the path to the saved model. The `TensorFlowPetDatasetMobileNetV2` model wrapper provides methods for loading the model, preprocessing the inputs, postprocessing the outputs and running inference using the native framework (TensorFlow in this case).

The dataset provided for evaluation is PetDataset - here we specify that we want to download the dataset to the `./build/pet-dataset` directory (`dataset_root`). The `PetDataset` class can download the dataset (if necessary), load it, read the inputs and outputs from files, process them, and implement evaluation methods for the model.

With the config above saved in the `native.json` file, run the `kenning test` scenario:

```
kenning test \
  --json-cfg mobilenetv2-tensorflow-native.json \
  --measurements build/native.json
```

This tool runs inference based on the given configuration, evaluates the model and stores quality and performance metrics in JSON format, saved to the `build/native.json` file. All other JSONs in this example use case can be executed with this command.

To visualize the evaluation and benchmark results, run the `kenning report` tool:

```
kenning report \
  --report-path build/benchmarks/native.md \
  --measurements build/native.json \
  --root-dir build/benchmarks \
  --img-dir build/benchmarks/img \
  --report-types performance classification \
  --report-name 'native'
```

The `kenning report` tool takes the output JSON file generated by the `kenning test` tool, and creates a report titled `native`, which is saved in the `build/benchmarks/native.md` directory. As specified in the `--report-types` flag, we create `performance` and `classification` metrics sections in the report (for example, there is also a `detection` report type for object detection tasks).

The `build/benchmarks/img` directory contains images with the `native_*` prefix visualizing the confusion matrix, CPU and memory usage, as well as inference time.

The `build/benchmarks/native.md` file is a Markdown document containing a full report for the model. The document links to the generated visualizations and provides aggregated information about CPU and memory usage, as well as classification quality metrics, such as accuracy, sensitivity, precision, or G-Mean. Such a file can be included in a larger, Sphinx-based documentation, which allows easy, automated report generation, using e.g. CI, like in the case of the Kenning documentation.

We can also use a simplified command:

```
kenning report \
  --report-path build/benchmarks/native.md \
  --measurements build/native.json
```

The available `--report-types` will be derived from measurements in the `build/native.json` file.

Conveniently, `kenning test` and `kenning report` commands can be reduced to a single `kenning` run:

```
kenning test report \
  --json-cfg mobilenetv2-tensorflow-native.json \
  --measurements build/native.json \
  --report-path build/benchmarks/native.md
```

While native frameworks are great for training and inference, model design, training on GPUs and distributing training across many devices, e.g. in a cloud environment, there is a fairly large variety of inference-focused frameworks for production purposes that focus on getting the most out of hardware in order to get results as fast as possible.

6.8.4 Optimizing a model using TensorFlow Lite

TensorFlow Lite is one of such frameworks. It is a lightweight library for inferring networks on edge - it has a small binary size (which can be reduced further by disabling unused operators) and a highly optimized format of input models, called FlatBuffers.

Before the TensorFlow Lite Interpreter (runtime for the TensorFlow Lite library) can be used, the model first needs to be optimized and compiled to the `.tflite` format.

Let's add a TensorFlow Lite Optimizer that will convert our MobileNetV2 model to a FlatBuffer format, as well as a TensorFlow Lite Runtime that will execute the model:

Listing 6.5: mobilenetv2-tensorflow-tflite-f32.json

```
    "optimizers": [
        {
            "type": "kenning.optimizers.tflite.TFLiteCompiler",
            "parameters": {
                "target": "default",
                "compiled_model_path": "./build/fp32.tflite",
                "inference_input_type": "float32",
                "inference_output_type": "float32"
            }
        }
```

The configuration of the already existing blocks does not change, and the dataset will not be downloaded again since the files are already present.

The first new addition in comparison to the previous flow is the presence of the `optimizers` list, which allows us to add one or more objects inheriting from the `kenning.core.optimizer.Optimizer` class. Optimizers read the model from the input file, apply various optimizations, and then save the optimized model to a new file.

In our current scenario, we will use the `TFLiteCompiler` class - it reads the model in a Keras-specific format, optimizes the model and saves it to the `./build/fp32.tflite` file. The parameters of this particular Optimizer are worth noting here (each Optimizer usually has a different set of parameters):

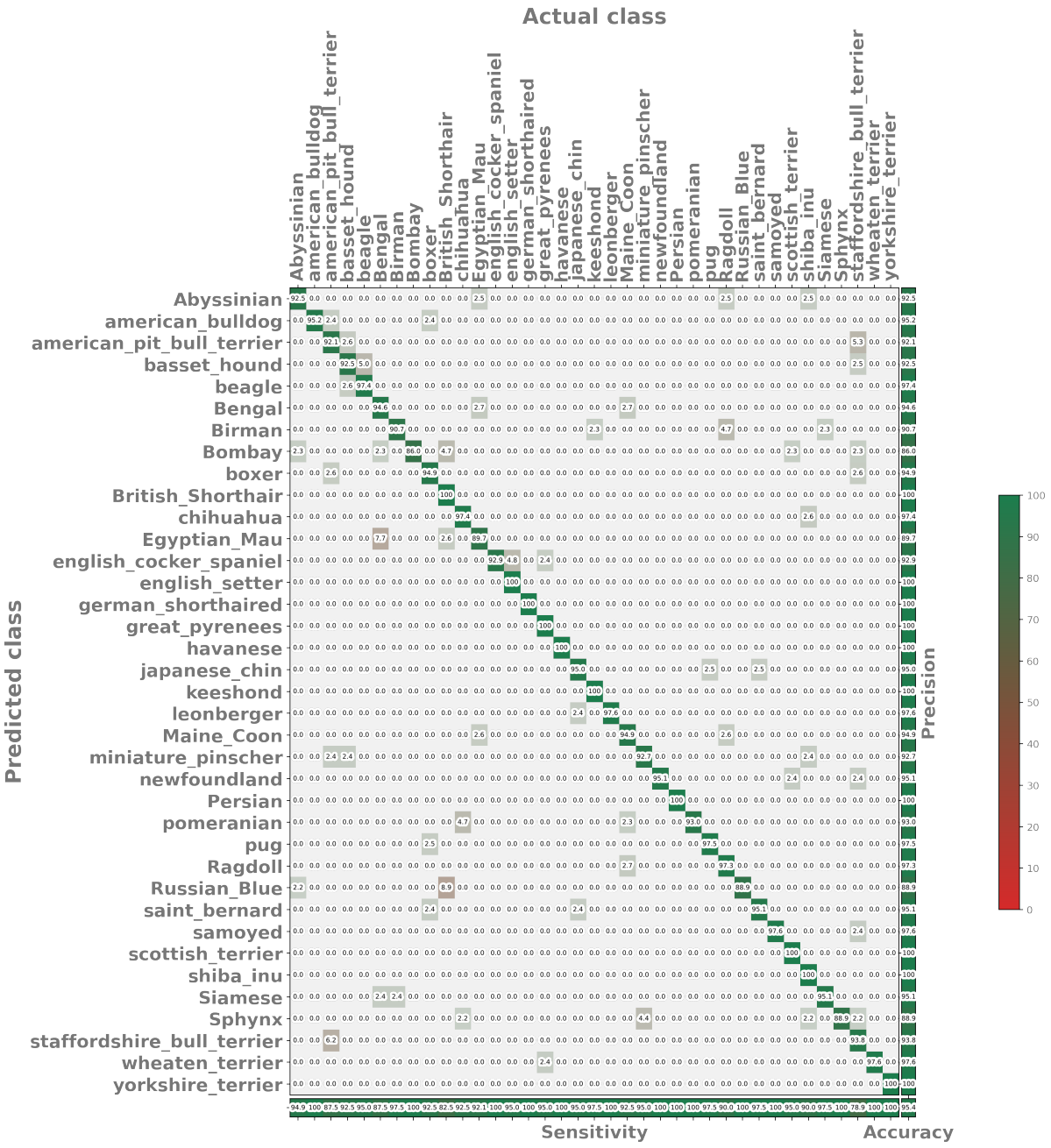
- `target` - indicates what the desired target device (or model type) is, regular CPU is `default`. Another example here could be `edgetpu`, which can compile models for the Google Coral platform.
- `compiled_model_path` - indicates where the model should be saved.
- `inference_input_type` and `inference_output_type` - indicate what the input and output type of the model should be. Usually, all trained models use FP32 weights (32-bit floating point) and activations - using `float32` here keeps the weights unchanged.

Another addition is the `runtime` block which provides a class inheriting from the `kenning.core.runtime.Runtime` class that is able to load the final model and run inference on target hardware. Usually, each `Optimizer` has a corresponding `Runtime` capable of running its results.

To compile the scenario (called `tflite-fp32.json`), run:

```
kenning optimize test report \
  --json-cfg mobilenetv2-tensorflow-tflite-f32.json \
  --measurements build/tflite-fp32.json \
  --report-path build/benchmarks/tflite-fp32.md
```

While it depends on the platform used, you should be able to see a significant improvement in both inference time (model ca. 10-15x faster model compared to the native model) and memory usage (output model ca. 2x smaller). What's worth noting is that we get a significant improvement with no harm to the quality of the model - the outputs stay the same.



6.8.5 Quantizing a model using TensorFlow Lite

To further reduce memory usage, we can quantize the model - it is a process where all weights and activations in a model are calibrated to work with low-precision floating-point or integer reporesentations (i.e. `INT8`), instead of the `FP32` precision. While it may severely harm prediction quality, the quality reduction can be negligible with proper calibration.

The model can be quantized during the compilation process in TensorFlow Lite. With Kenning, it can be achieved with the following simple additions:

Listing 6.6: mobilenetv2-tensorflow-tflite-int8.json

```
"optimizers": [
  {
    "type": "kenning.optimizers.tflite.TFLiteCompiler",
    "parameters": {
      "target": "int8",
      "compiled_model_path": "./build/int8.tflite",
      "inference_input_type": "int8",
      "inference_output_type": "int8"
    }
  }
```

The only changes here in comparison to the previous configuration appear in the `TFLiteCompiler` configuration. We change `target`, `inference_input_type` and `inference_output_type` to `int8`. Then, in the background, `TFLiteCompiler` fetches a subset of images from the `PetDataset` object to calibrate the model and the entire model calibration process happens automatically.

Let's run the scenario above (`tflite-int8.json`):

```
kenning optimize test report \
  --json-cfg mobilenetv2-tensorflow-tflite-int8.json \
  --measurements build/tflite-int8.json \
  --report-path build/benchmarks/tflite-int8.md
```

This results in a model over 7 times smaller compared to the native model without significant loss of accuracy (but without speed improvement).

6.8.6 Speeding up inference with Apache TVM

To speed up inference of a quantized model, we can utilize vector extensions in x86 CPUs, more specifically AVX2. To do this, we can use the Apache TVM framework to compile efficient runtimes for various hardware platforms. The scenario looks like this:

Listing 6.7: mobilenetv2-tensorflow-tvm-avx-int8.json

```
"optimizers": [
  {
    "type": "kenning.optimizers.tflite.TFLiteCompiler",
    "parameters": {
      "target": "int8",
      "compiled_model_path": "./build/int8.tflite",
      "inference_input_type": "int8",
      "inference_output_type": "int8"
    }
  },
  {
    "type": "kenning.optimizers.tvm.TVMCompiler",
    "parameters": {
```

(continues on next page)

(continued from previous page)

```
        "target": "llvm -mcpu=core-avx2",
        "opt_level": 3,
        "conv2d_data_layout": "NCHW",
        "compiled_model_path": "./build/int8_tvm.tar"
      }
    }
```

As visible, adding a new framework is just a matter of simply adding and configuring another optimizer and using a `Runtime` corresponding to the final `Optimizer`.

The `TVMCompiler`, with `llvm -mcpu=core-avx2` as the target, optimizes and compiles the model to use vector extensions. The final result is a `.tar` file containing a shared library that implements the entire model.

Let's compile the scenario (`tvm-avx2-int8.json`):

```
kenning optimize test report \
  --json-cfg mobilenetv2-tensorflow-tvm-avx-int8.json \
  --measurements build/tvm-avx2-int8.json \
  --report-path build/benchmarks/tvm-avx2-int8.md
```

This results in a model over 40 times faster compared to the native implementation, with a 3x reduction in size.

This demonstrates how easily we can interconnect various frameworks and get the most out of hardware using Kenning, while performing just minor alterations to the configuration file.

The summary of passes looks as follows:

	Speed boost	Accuracy	Size reduction
native	1	0.9572730984	1
tflite-fp32	15.79405698	0.9572730984	1.965973551
tflite-int8	1.683232669	0.9519662539	7.02033412
tvm-avx2-int8	41.61514549	0.9487005035	3.229375069

6.8.7 Automated model comparison

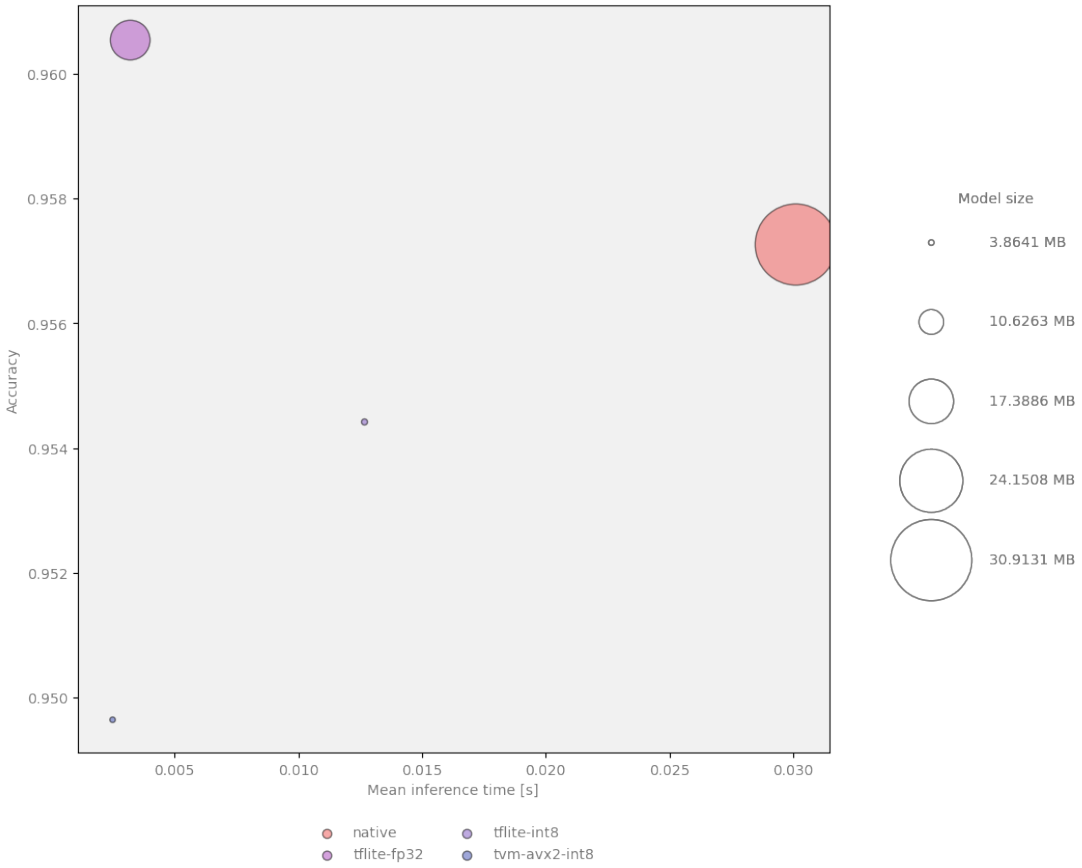
The `kenning report` tool also allows us to compare evaluation results for multiple models. Apart from creating a model summary table, it also creates plots aggregating measurements collected during the evaluation process.

To create a comparison report for the above experiments, run:

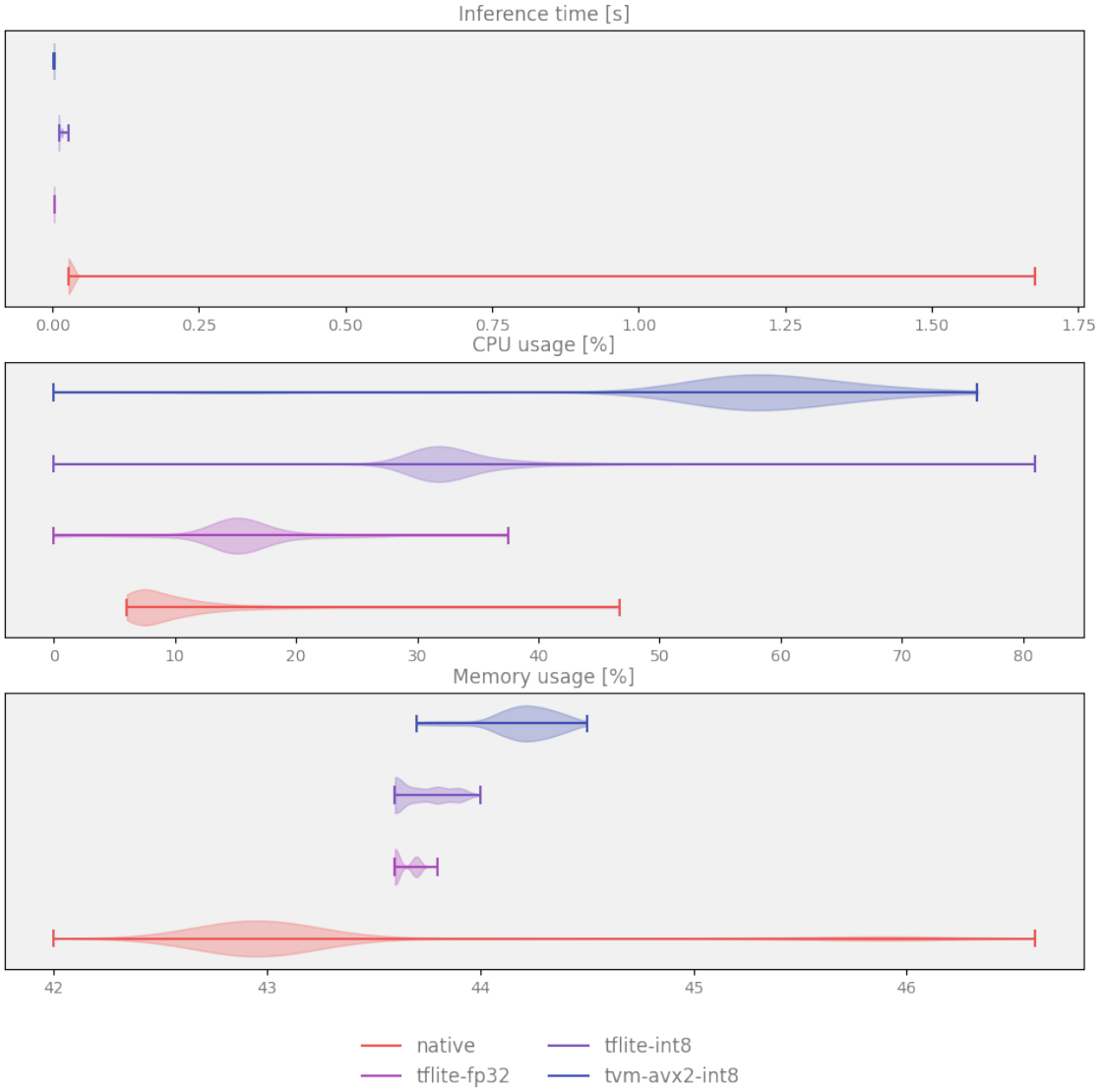
```
kenning report \
    --report-path build/benchmarks/summary.md \
    --measurements \
        build/native.json \
        build/tflite-fp32.json \
        build/tflite-int8.json \
        build/tvm-avx2-int8.json
```

Some examples of comparisons between various models rendered with the script:

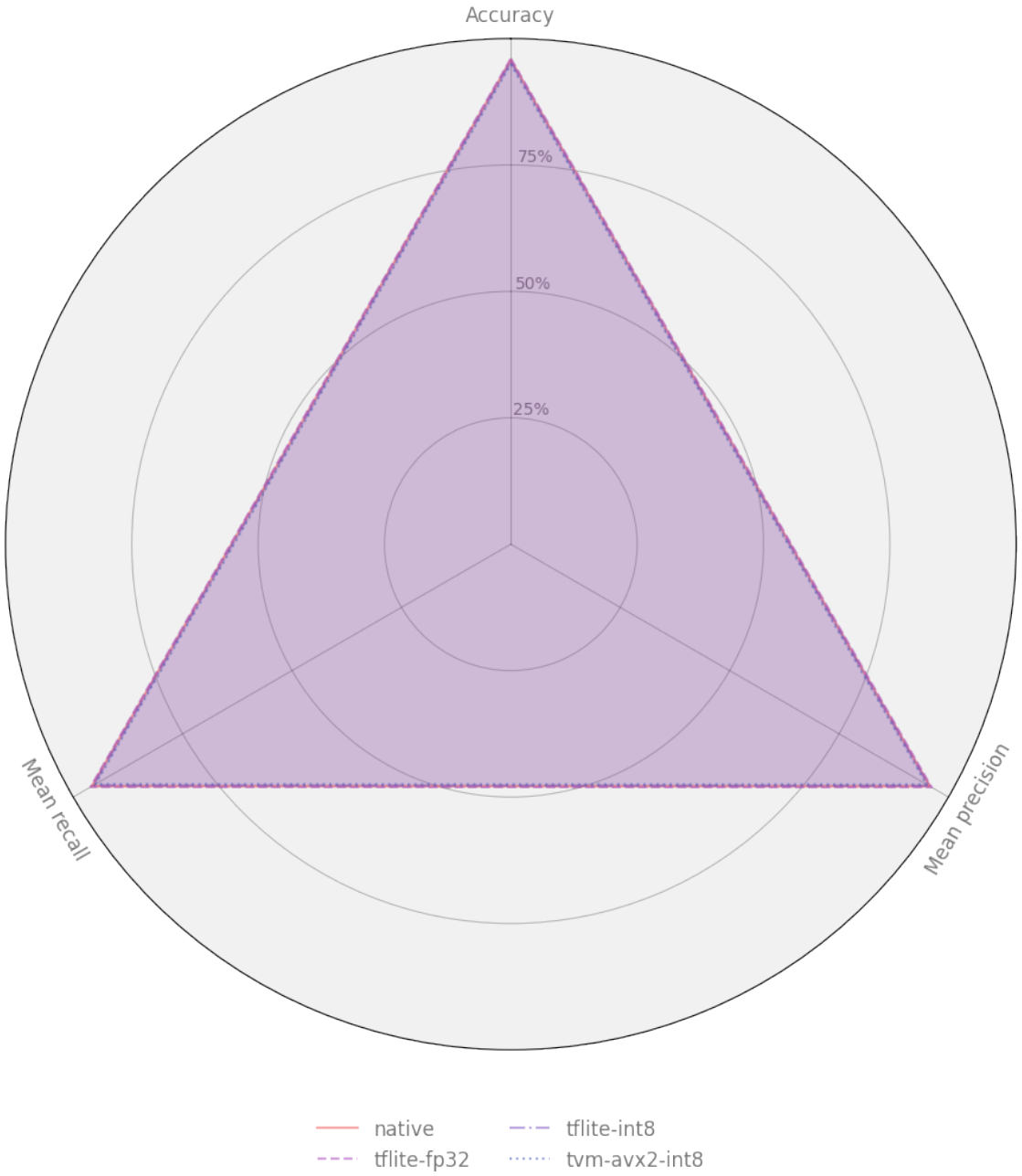
- Accuracy, inference time and model size comparison:



- Resource utilization distribution:



- Comparison of classification metrics:



- And more

6.9 Unstructured Pruning of TensorFlow Models

This section contains a tutorial for unstructured pruning of the TensorFlow classification model.

As is the case with most pruning techniques, this type of unstructured pruning requires fine-tuning after "removing" certain connections through training. In unstructured pruning, the weights for certain connections are set to zero. The weights and layers are not removed, but the matrices holding weights become sparse. It can be used to improve model compression (for storage purposes) and for use with dedicated hardware and libraries that can take advantage of sparse computing.

6.9.1 Setup

A few additional dependencies are required to prune TensorFlow models, they can be installed using the following command:

```
pip install "kenning[tensorflow,reports] @ git+https://github.com/antmicro/
↪kenning.git"
```

6.9.2 Experiments

First, we need to know the performance of the original model which can be achieved by running the following pipeline:

Listing 6.8: mobilenetv2-tensorflow.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
  }
}
```

To test it, run:

```
kenning test \
  --json-cfg mobilenetv2-tensorflow.json \
  --measurements build/tf.json
```

`TensorflowPruningOptimizer` has two main parameters for adjusting the pruning process:

- `target_sparsity` - defines sparsity of weights after pruning,
- `prune_dense` - if `true`, only dense layers will be pruned instead of the entire model.

You can also adjust how often a model will be pruned (with `pruning_frequency`), and when it cannot be pruned (`pruning_end`).

You can also chose an `optimizer` (one of `adam`, `RMSprop` or `SGD`) and specify if network's output is normalized with `disable_from_logits`.

In this example, we decided to fine-tune the model for three epochs with batch size equal to 128.

Listing 6.9: pruning-mobilenetv2-tensorflow.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
      "model_path": "kenning:///models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5"
    }
  },
  "dataset": {
    "type": "kenning.datasets.pet_dataset.PetDataset",
    "parameters": {
      "dataset_root": "./build/PetDataset"
    }
  },
  "optimizers": [
    {
      "type": "kenning.optimizers.tensorflow_pruning.TensorFlowPruningOptimizer",
      "parameters": {
        "compiled_model_path": "./build/tf-pruning-0_15.h5",
        "target_sparsity": 0.15,
        "batch_size": 128,
        "epochs": 3,
        "pruning_frequency": 20,
        "pruning_end": 120,
        "save_to_zip": true
      }
    }
  ]
}
```

To prune the model, run:

```
kenning optimize \
  --json-cfg pruning-mobilenetv2-tensorflow.json
kenning test \
  --json-cfg pruning-mobilenetv2-tensorflow.json \
  --measurements build/tf-pruning.json
```

Despite the fact, that the Kenning CLI is capable of running commands in sequence (like `kenning optimize test [FLAGS]`), we suggest separating them for more measurement precision.

In order to compare a model before and after pruning, you can generate a report using the following command:

```
kenning report \
  --measurements \
    build/tf.json \
    build/tf-pruning.json \
```

(continues on next page)

(continued from previous page)

```
  --report-path build/tf-pruning.md \
  --to-html
```

For greater size reduction, we can use larger sparsity with adjusted parameters, like below:

```
{
  "type": "kenning.optimizers.tensorflow_pruning.TensorFlowPruningOptimizer",
  "parameters": {
    "compiled_model_path": "./build/tf-pruning-0_3.h5",
    "target_sparsity": 0.3,
    "batch_size": 128,
    "epochs": 5,
    "pruning_frequency": 40,
    "pruning_end": 82,
    "save_to_zip": true
  }
}
```

The default TensorFlow version of the pretrained MobileNetV2 (`kenning:///models/classification/tensorflow_pet_dataset_mobilenetv2.h5`) is exported with an optimizer, resulting in size increase. To ensure better measurement quality, we will present the data with the default MobileNetV2 stripped from any unnecessary information.

Summary of pruned models:

Sparsity	Accuracy	Compressed size	Size reduction
—	0.9605442177	15,508,373	0.00%
0.15	0.9190476190	14,062,345	9.32%
0.3	0.8367346939	12,349,076	20.37%

6.9.3 (Optional) Clustering

Kenning provides a few other methods for reducing the size of the model. One of such techniques is clustering. It groups weights into K groups of similar values. Then, it computes K centroids based on those weights and use them as values for new weights. In weight matrices, we store indices to corresponding centroid instead of values. The indices can be stored as integers with a very small number of bits necessary to represent them, reducing the model size significantly.

You can do this by adding `kenning.optimizers.tensorflow_clustering.TensorFlowClusteringOptimizer`:

Listing 6.10: pruning-clustering-mobilenetv2-tensorflow.json

```
  "optimizers": [
    {
      "type": "kenning.optimizers.tensorflow_pruning.TensorFlowPruningOptimizer",
      "parameters": {
        "compiled_model_path": "./build/tf-pruning-0_15.h5",
```

(continues on next page)

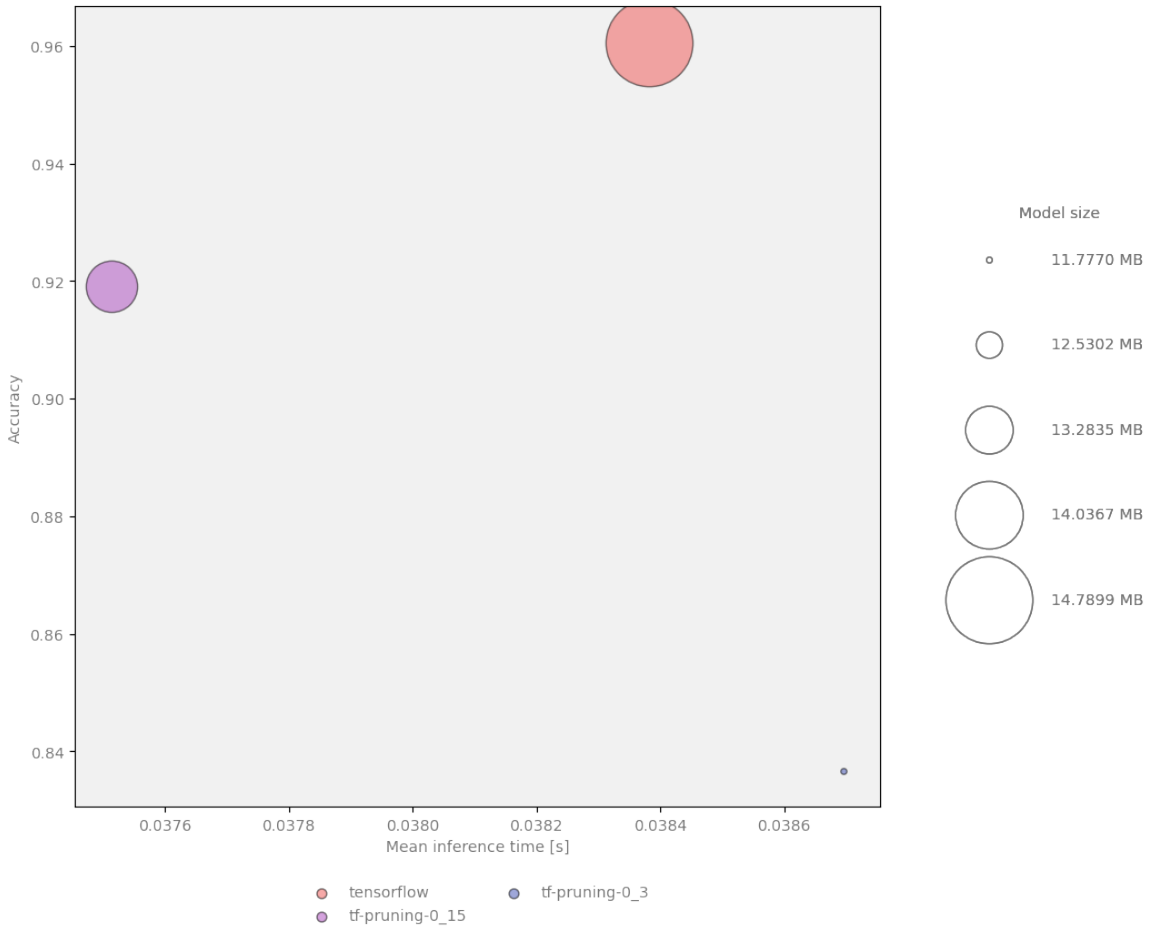


Figure 6.4: Model size, speed and quality comparison for TensorFlow pruning

(continued from previous page)

```
        "target_sparsity": 0.15,
        "batch_size": 128,
        "epochs": 5,
        "pruning_frequency": 10
      }
    },
    {
      "type": "kenning.optimizers.tensorflow_clustering.
↪TensorFlowClusteringOptimizer",
      "parameters": {
        "compiled_model_path": "./build/tf-pruning-clustering.h5",
        "cluster_dense": false,
        "preserve_sparsity": true,
        "clusters_number": 60
      }
    }
  ]
```

To run it, use:

```
kenning optimize test report \
  --json-cfg pruning-clustering-mobilenetv2-tensorflow.json \
  --measurements build/tf-all.json \
  --report-path build/tf-pruning-clustering.md \
  --to-html
```

Clustering allows for substantial model size reduction without significant decrease in quality. Here is a comparison of a model with and without clustering:

Sparsity	Number of clusters	Accuracy	Compressed size	Size reduction
—	—	0.9605442177	15,508,373	0.00%
0.15	—	0.9190476190	14,062,345	9.32%
0.15	60	0.9006802721	4,451,142	71.30%

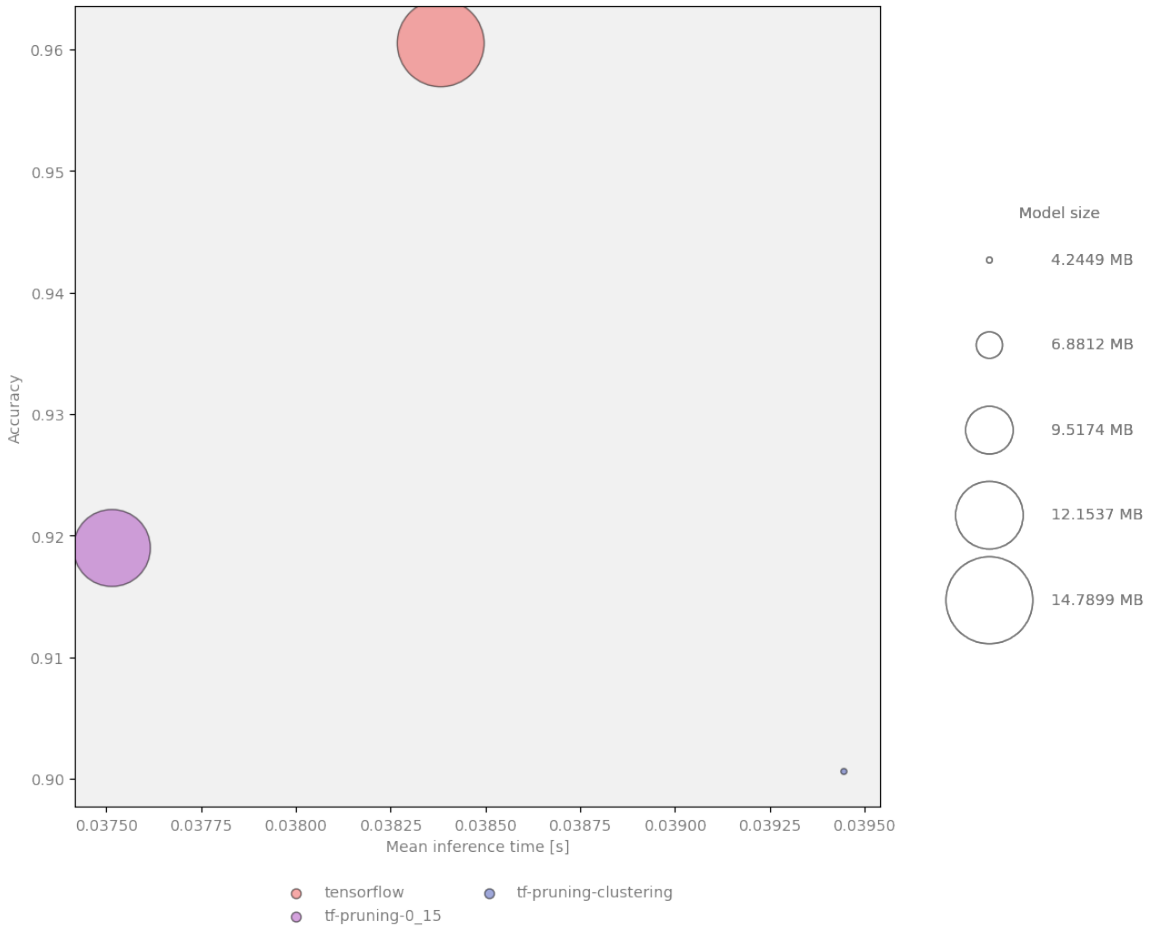


Figure 6.5: Model size, speed and quality comparison for TensorFlow pruning and clustering

CHAPTER

SEVEN

KENNING ENVIRONMENT VARIABLES

This section contains information about Kenning environment variables and how they influence the behavior of the program.

7.1 KENNING_CACHE_DIR

`KENNING_CACHE_DIR` defines directory for resources used by Kenning, like downloaded datasets. If it is not defined, `$HOME/.kenning` will be used.

7.2 KENNING_MAX_CACHE_SIZE

`KENNING_MAX_CACHE_SIZE` specifies the maximum amount of space used by *cache*. The default value is 50GB.

7.3 KENNING_USE_DEFAULT_EXCEPTHOOK

If `KENNING_USE_DEFAULT_EXCEPTHOOK` environmental variable is defined, `sys.excepthook` will not be overridden.

This function is overridden in order to enable mechanism for deducing which optional dependencies are missing. For instance, if `gluoncv` is not available, Kenning will suggest installing `kenning[mxnet]` which contains missing package with defined version restriction and other requirements.

Kenning CLI does not require custom `sys.excepthook`, so it will not be affected by `USE_DEFAULT_EXCEPTHOOK` variable.

7.4 KENNING_DOCS_VENV

`KENNING_DOCS_VENV` defines path to the virtual environment used for tests marked as `docs`. If is not defined, system's default environment will be used.

7.5 KENNING_ENABLE_ALL_LOGS

If KENNING_ENABLE_ALL_LOGS environmental variable is defined, logs from other libraries will be enabled. Passes verbosity of Kenning logs to other libraries.

This is useful for debugging purposes, but it may cause a lot of noise in the logs.

7.6 KENNING_DISABLE_IO_VALIDATION

By default Kenning, before and after each inference, validates received data with defined specification. Similar check is done before evaluation.

These validations can be disabled defining KENNING_DISABLE_IO_VALIDATION variable.

CHAPTER

EIGHT

KENNING MEASUREMENTS

Kenning measurements are a set of information describing the compilation and evaluation process happening in Kenning.

They contain such information as:

- classes used to construct the optimization/runtime pipeline, along with their parameters,
- the JSON scenario used in the run,
- the command used to run the scenario,
- versions of the Python modules used,
- performance measurements, such as CPU usage, GPU usage,
- quality measurements, such as predictions, ground truth, confusion matrix

All information is stored in JSON format.

8.1 Performance metrics

While quality measurements are problem-specific (collected in the `evaluate` method of the *Dataset* class), performance metrics are common across devices and applications.

Metrics are collected with a certain prefix `<prefix>`, indicating the scope of computations. There are:

- `<prefix>_timestamp` - gives a timestamp for measurement collection in seconds.
- `<prefix>_cpus_percent` - gives per-core CPU utilization in % in a form of a list of lists. They are % of per-CPU usages for every timestamp.
- `<prefix>_mem_percent` - gives overall memory usage in %.
- `<prefix>_gpu_utilization` - gives overall GPU utilization in % (only works on platforms with NVIDIA GPUs and NVIDIA Jetson embedded devices).
- `<prefix>_gpu_mem_utilization` - gives GPU memory utilization in % (only works on platforms with NVIDIA GPUs and NVIDIA Jetson embedded devices).

CHAPTER

NINE

CHOOSING OPTIMAL OPTIMIZATION PIPELINE

The `kenning.scenarios.optimization_runner` script allows to optimize over multiple pipelines and to choose the best performing based on a specified criteria

The script can be run as follows:

```
kenning fine-tune-optimizers --json-cfg <CONFIG_JSON>.json --output <OUTPUT_PATH>.
↪json
```

With the arguments:

- `<CONFIG_JSON>.json` - describes the configuration, which pipelines would be executed and optimization settings
- `<OUTPUT_PATH>.json` - base path to the output files with measurements.

9.1 Optimization config specification

The example configuration looks like this:

```
{
    "optimization_parameters":
    {
        "strategy": "grid_search",
        "optimizable": ["optimizers", "runtime"],
        "metric": "inferencetime_mean",
        "policy": "min"
    },
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
```

(continues on next page)

(continued from previous page)

```
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
        {
            "dataset_root": "./build/pet-dataset"
        }
    },
    "optimizers":
    [
        {
            "type": "kenning.optimizers.tflite.TFLiteCompiler",
            "parameters":
            {
                "target": ["default"],
                "compiled_model_path": ["./build/compiled_model.tflite"]
            }
        },
        {
            "type": "kenning.optimizers.tvm.TVMCompiler",
            "parameters":
            {
                "target": ["llvm"],
                "compiled_model_path": ["./build/compiled_model.tar"],
                "opt_level": [3],
                "conv2d_data_layout": ["NHWC", "NCHW"]
            }
        }
    ],
    "runtime":
    [
        {
            "type": "kenning.runtimes.tvm.TVMRuntime",
            "parameters":
            {
                "save_model_path": ["./build/compiled_model.tar"]
            }
        },
        {
            "type": "kenning.runtimes.tflite.TFLiteRuntime",
            "parameters":
            {
                "save_model_path": ["./build/compiled_model.tflite"]
            }
        }
    ]
}
```

The highlighted part describes the settings of the optimization run.

- `strategy` - Describes how the pipelines are chosen for optimization. Currently only available option is `grid_search`, which generates cartesian product of all blocks that should be

optimized and runs all compatible pipelines

- `optimizable` - Which Kenning block should be variable in the pipeline. The `kenning.scenarios.optimization_runner` script will choose the Kenning block out of multiple provided in the later part of configuration based on `strategy`.
- `metric` - one of the metrics from *Kenning measurements* over which the optimization should be performed
- `policy` - `min` or `max`, depending on whether the metric should be maximized or minimized

The rest of the JSON configuration describes the pipeline in a format similar to *standard JSON scenarios*. The only difference is within the blocks chosen in the `optimizable` field, which should be list of blocks instead of singular definition. Every block in the list has defined list of arguments for all of it's parameters, this allows to choose the best argument for a particular block option

- For `optimizers` block, since there can be more than one in the pipeline, all possible combinations are tested.
- For the rest of the blocks one of all possible blocks is chosen for each run.

Kenning checks automatically whether a selected combination of blocks (model, optimizations and runtime) is compatible.

9.2 Output details

The `kenning.scenarios.optimization_runner` outputs multiple JSON files:

- `<OUTPUT PATH>.json` - Most optimal pipeline with it's aggregated metrics
- `<OUTPUT PATH>_all_results.json` - All pipeline configs and their respective metrics gathered into single file
- `<OUTPUT PATH>_<RUN ID>.json` - Full benchmark output for every optimization run.

The example `<OUTPUT PATH>.json` JSON file can look like this:

```
{
    "pipeline": {
        "model_wrapper": {
            "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
            "parameters": {
                "model_path": "kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5"
            }
        },
        "dataset": {
            "type": "kenning.datasets.pet_dataset.PetDataset",
            "parameters": {
                "dataset_root": "./build/pet-dataset"
            }
        },
        "optimizers": [
```

(continues on next page)

(continued from previous page)


```
            {
                "type": "kenning.optimizers.tvm.TVMCompiler",
                "parameters": {
                    "target": "llvm",
                    "compiled_model_path": "build/7_compiled_model.tar",
                    "opt_level": 3,
                    "conv2d_data_layout": "NCHW"
                }
            }
        ],
        "runtime": {
            "type": "kenning.runtimes.tflite.TFLiteRuntime",
            "parameters": {
                "save_model_path": "build/7_compiled_model.tflite"
            }
        }
    },
    "metrics": {
        "inferencetime_mean": 0.0031105340278004203,
        "inferencetime_std": 0.0003073369739186006,
        "inferencetime_median": 0.003066142999159638,
        "session_utilization_mem_percent_mean": 26.089648437500003,
        "session_utilization_mem_percent_std": 0.2059426531921169,
        "session_utilization_mem_percent_median": 26.1,
        "session_utilization_cpus_percent_avg_mean": 16.55806884765625,
        "session_utilization_cpus_percent_avg_std": 4.068584532658253,
        "session_utilization_cpus_percent_avg_median": 15.85625,
        "accuracy": 0.957273098380732,
        "mean_precision": 0.9574413909275928,
        "mean_sensitivity": 0.9571990915922841,
        "g_mean": 0.9566448082813757,
        "top_5_accuracy": 0.9961899578173902
    }
}
```


- The `pipeline` contains the definition of the best optimization pipeline.
- The `metrics` contains list of all aggregated metrics that can be considered in `metric` field `optimization_parameters`.

CHAPTER

TEN

SAMPLE AUTOGENERATED REPORT

This section contains a CI sample report for running inference on a compiled model.

The CI is set up as follows:

- *Environment*: Github Actions
- *Task*: dog and cat breeds classification based on the Oxford-IIIT Pet Dataset,
- *Training framework*: TensorFlow,
- *Compiler framework*: TVM,
- *Target*: CPU (using LLVM target on TVM).

10.1 Pet Dataset classification using TVM-compiled TensorFlow model

10.1.1 Commands used

Note: This section was generated using:

```
python -m kenning.__main__ \
    optimize \
    test \
    --modelwrapper-cls \
        kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls \
        kenning.datasets.pet_dataset.PetDataset \
    --measurements \
        ./build/local-cpu-tvm-tensorflow-classification.json \
    --compiler-cls \
        kenning.optimizers.tvm.TVMCompiler \
    --runtime-cls \
        kenning.runtimes.tvm.TVMRuntime \
    --model-path \
        kenning:///models/classification/tensorflow_pet_dataset_mobilenetv2.h5 \
    --model-framework \
        keras \
    --target \
```

(continues on next page)

(continued from previous page)

```
        llvm \
    --compiled-model-path \
        ./build/compiled-model.tar \
    --opt-level \
        3 \
    --save-model-path \
        ./build/compiled-model.tar \
    --target-device-context \
        cpu \
    --dataset-root \
        ./build/PetDataset/ \
    --inference-batch-size \
        1 \
    --verbosity \
        INFO

python -m kenning.__main__ \
    report \
    --report-path \
        docs/source/generated/local-cpu-tvm-tensorflow-classification.md \
    --report-name \
        Pet Dataset classification using TVM-compiled TensorFlow model \
    --root-dir \
        docs/source/ \
    --img-dir \
        docs/source/generated/img \
    --report-types \
        performance \
        classification \
    --measurements \
        build/local-cpu-tvm-tensorflow-classification.json \
    --smaller-header
```

10.1.2 General information for build.local-cpu-tvm-tensorflow-classification.json

Model framework:

- tensorflow ver. 2.14.1

Input JSON:

```
{
    "dataset": {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters": {
            "classify_by": "breeds",
            "image_memory_layout": "NHWC",
            "dataset_root": "build/PetDataset",
```

(continues on next page)

(continued from previous page)

```
            "inference_batch_size": 1,
            "download_dataset": true,
            "force_download_dataset": false,
            "external_calibration_dataset": null,
            "split_fraction_test": 0.2,
            "split_fraction_val": null,
            "split_seed": 1234,
            "reduce_dataset": 1.0
        }
    },
    "dataconverter": {
        "type": "kenning.dataconverters.modelwrapper_dataconverter.
↪ModelWrapperDataConverter",
        "parameters": {}
    },
    "optimizers": [
        {
            "type": "kenning.optimizers.tvm.TVMCompiler",
            "parameters": {
                "model_framework": "keras",
                "target": "llvm",
                "target_attrs": "",
                "target_microtvm_board": null,
                "target_host": null,
                "zephyr_header_template": null,
                "zephyr_llext_source_template": null,
                "opt_level": 3,
                "libdarknet_path": "/usr/local/lib/libdarknet.so",
                "compile_use_vm": false,
                "output_conversion_function": "default",
                "conv2d_data_layout": "",
                "conv2d_kernel_layout": "",
                "use_fp16_precision": false,
                "use_int8_precision": false,
                "use_tensorrt": false,
                "dataset_percentage": 0.25,
                "compiled_model_path": "build/compiled-model.tar",
                "location": "host"
            }
        }
    ],
    "platform": {
        "type": "kenning.platforms.local.LocalPlatform",
        "parameters": {
            "name": null,
            "platforms_definitions": [
                "/home/runner/work/kenning/kenning/kenning/resources/platforms/
↪platforms.yml"
            ]
```

(continues on next page)

(continued from previous page)


```
        }
    },
    "model_wrapper": {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters": {
            "batch_size": null,
            "learning_rate": null,
            "num_epochs": null,
            "logdir": null,
            "model_path": "/home/runner/.kenning/models/classification/tensorflow_
↪pet_dataset_mobilenetv2.h5",
            "model_name": null
        }
    },
    "runtime": {
        "type": "kenning.runtimes.tvm.TVMRuntime",
        "parameters": {
            "save_model_path": "build/compiled-model.tar",
            "target_device_context": "cpu",
            "target_device_context_id": 0,
            "runtime_use_vm": false,
            "llext_binary_path": null,
            "disable_performance_measurements": false
        }
    }
}
```


10.2 Inference performance metrics for build.local-cpu-tvm-tensorflow-classification.json

10.2.1 Inference time

- *First inference duration* (usually including allocation time): **0.07428212199999962**,
- *Mean*: **0.07506955665170091 s**,
- *Standard deviation*: **0.00037084322156172094 s**,
- *Median*: **0.07504371299998525 s**.

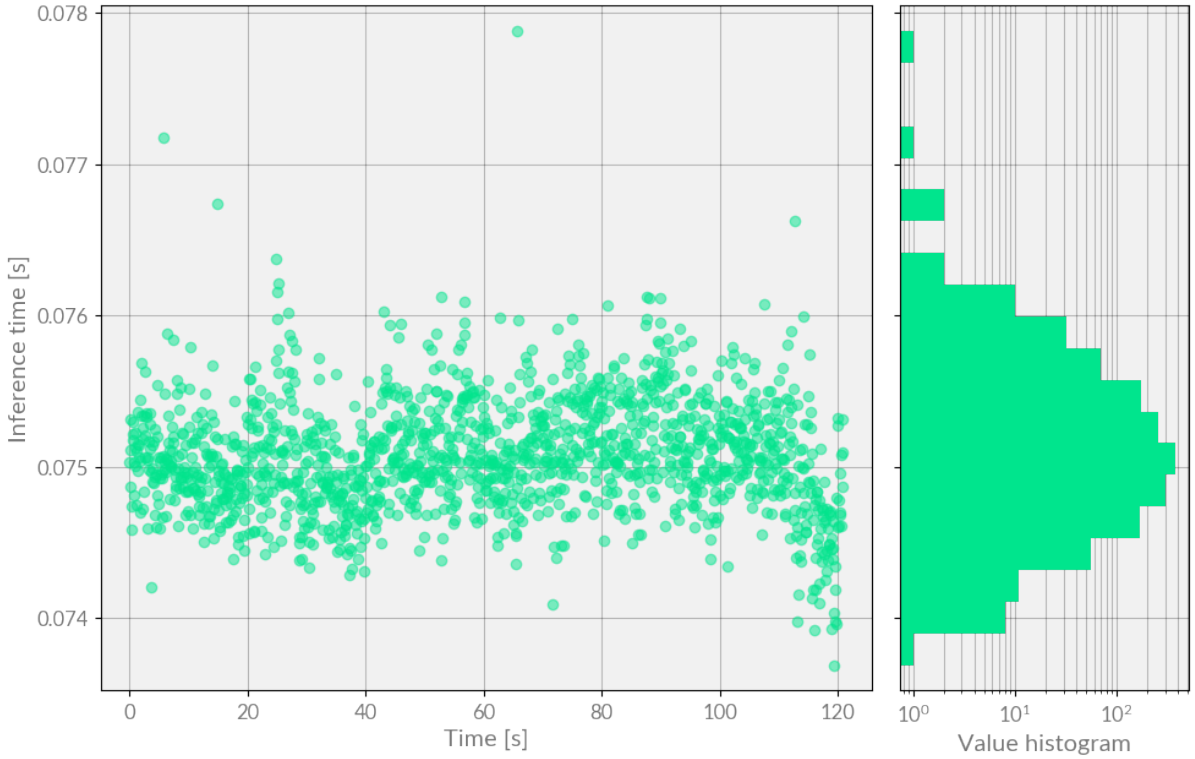


Figure 10.1: Inference time

10.2.2 Average CPU usage

- *Mean*: **50.30022859517872 %**,
- *Standard deviation*: **2.208580723695854 %**,
- *Median*: **50.0 %**.

10.2.3 Memory usage

- *Mean*: **9.316209476309227 %**,
- *Standard deviation*: **0.056893209174583535 %**,
- *Median*: **9.3 %**.

10.3 Inference quality metrics for build.local-cpu-tvm-tensorflow-classification.json

- *Accuracy*: **0.9578231292517007**
- *Mean precision*: **0.9591048246905213**
- *Mean sensitivity*: **0.9572500515056033**
- *G-mean*: **0.9563343851604948**
- *Top-5 accuracy*: **0.9959183673469387**

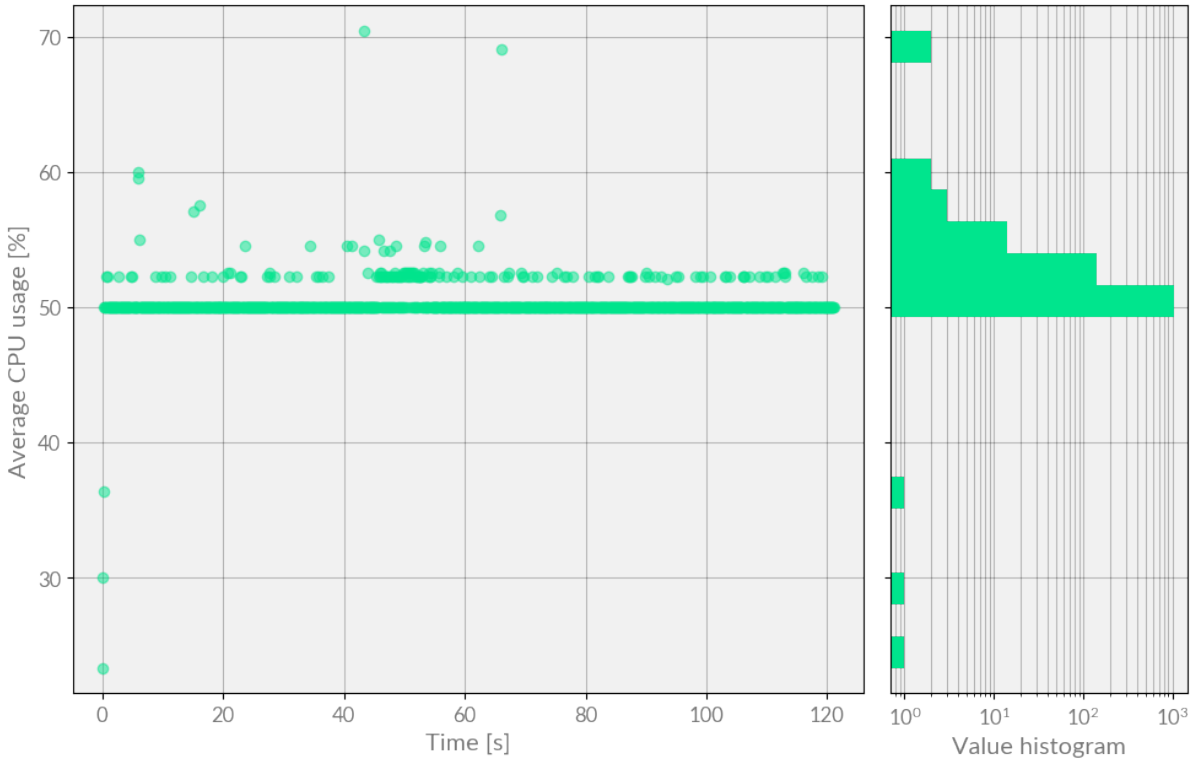


Figure 10.2: Average CPU usage during benchmark

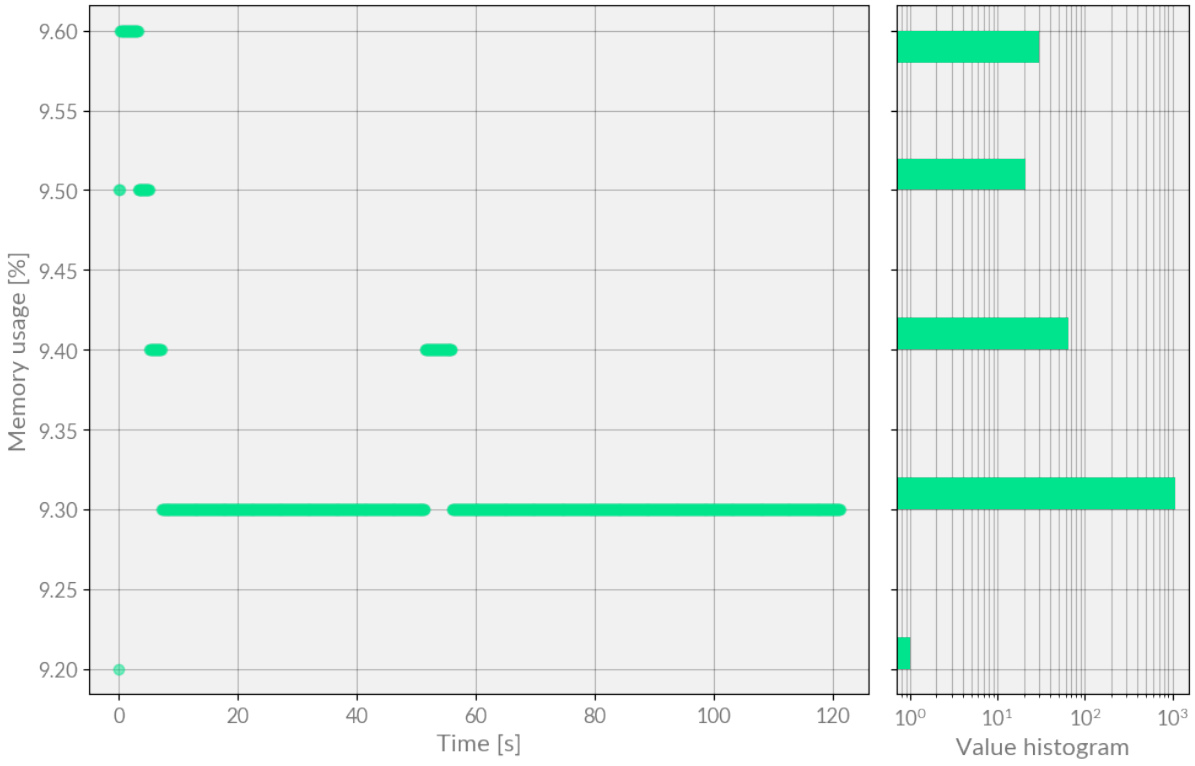


Figure 10.3: Memory usage during benchmark

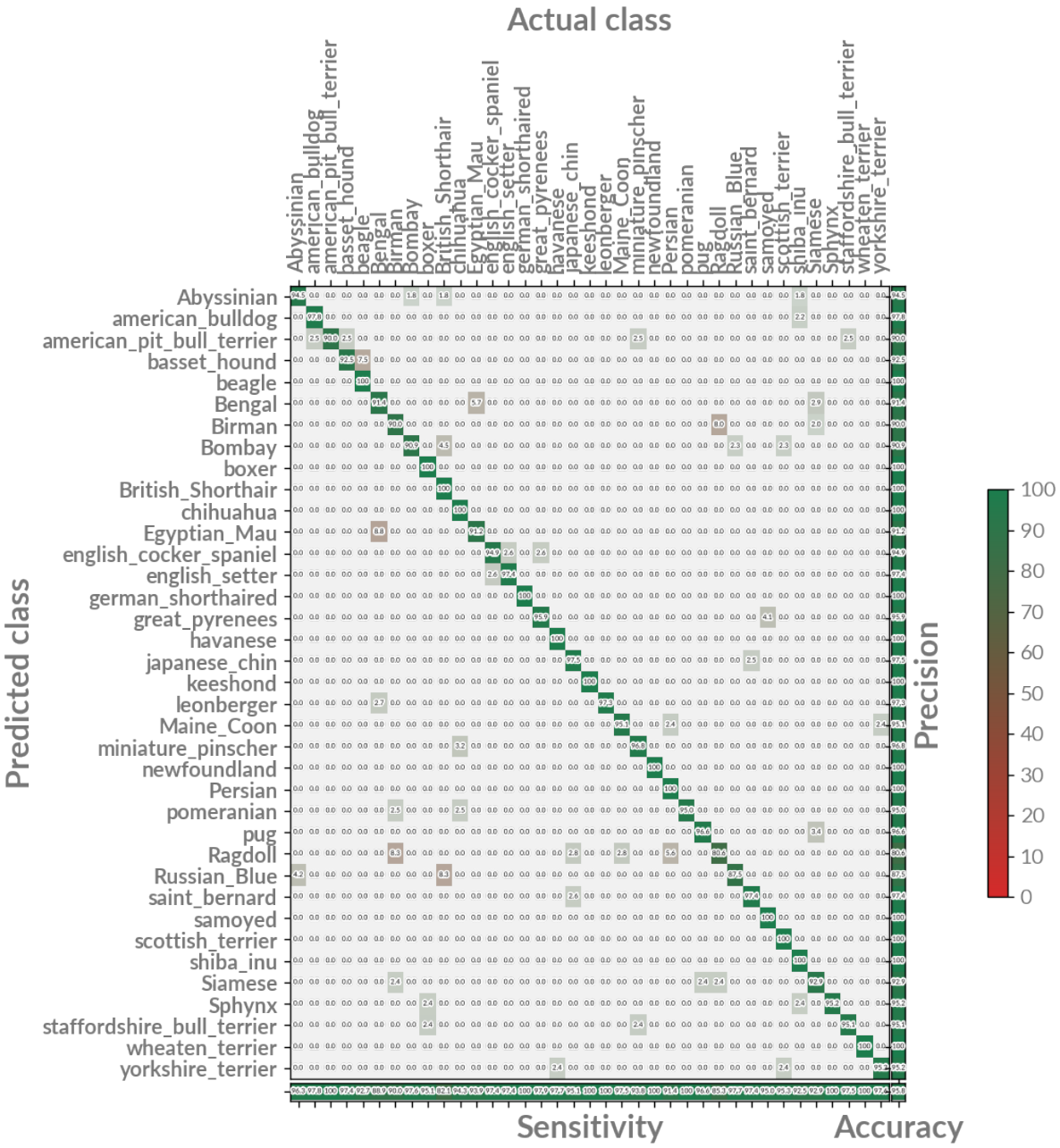


Figure 10.4: Confusion matrix

CHAPTER

ELEVEN

CREATING APPLICATIONS WITH KENNING

The *KenningFlow* allows you to run an arbitrary sequence of processing blocks that provide data, execute models using existing Kenning classes and wrappers, and processes results. You can use it to quickly create applications with Kenning after optimizing the model and using it in actual use cases.

Kenning for runtime uses both existing classes, such as *ModelWrapper*, *Runtime*, and dedicated *Runner*-based classes. The latter family of classes are actual functional blocks used in *KenningFlow* that can be used for:

- Obtaining data from sources - *DataProvider*, e.g. iterating files in the filesystem, grabbing frames from a camera or downloading data from a remote source,
- Processing and delivering data - *OutputCollector*, e.g. sending results to the client application, visualizing model results in GUI, or storing results in a summary file,
- Running and processing various models,
- Applying other actions, such as additional data analysis, preprocessing, packing, and more.

A *KenningFlow* scenario definition can be saved in a JSON file and then run using the `kenning.scenarios.json_flow_runner` script.

11.1 JSON structure

JSON configuration consist of a list of dictionaries describing each *Runner*-based instance.

A sample *Runner* specification looks as follows:

```
{
  "type": "kenning.dataproviders.camera_dataprovider.CameraDataProvider",
  "parameters": {
    "video_file_path": "/dev/video0",
    "input_memory_layout": "NCHW",
    "input_width": 608,
    "input_height": 608
  },
  "outputs": {
    "frame": "cam_frame"
  }
}
```

Each *Runner* dictionary consists of:

- `type` - *Runner* class. E.g. CameraDataProvider,
- `parameters` - parameters passed to class constructor. In this case, we specify a path to a video device (`/dev/video0`), expected memory format (`NCHW`), image size (`608x608`)
- `inputs` - (*optional*) *Runner* instance inputs. In the example above, there are none,
- `outputs` - (*optional*) *Runner* instance outputs. In the example above, it is a single output - camera frame defined as the variable `cam_frame` in the flow.

11.1.1 Runner IO

The input and output specification in *Runner* classes is the same as described in *Model and I/O metadata*.

11.1.2 IO compatibility

IO compatibility is checked during flow JSON parsing.

The *Runner* input is considered to be compatible with associated outputs if:

- in case of `numpy.ndarray`: `dtype` and `ndim` are equal and each dimension has either the same length or input dimension is set as `-1`, which represents any length. In the input spec, there can also be multiple valid shapes. If so, they are placed in an array, i.e. `[(1, -1, -1, 3), (1, 3, -1, -1)]`,
- in other cases: `type` fields are either equal or the input `type` field is `Any`.

11.1.3 IO non-standard types

If the input or output is not a `numpy.ndarray`, then its type is described by the `type` field, which is a string. In the case of a detection output from an IO specification (described above) it is a `List[DetectObject]`. This is interpreted as a list of `DetectObjects`. The `DetectObject` is a named tuple describing detection output (class names, rectangle positions, score).

11.1.4 IO names and mapping

The inputs and outputs present in JSON are mappings from *Runner*'s local IO names to flow global variable names, i.e. one *Runner* can define its outputs as `{"output_name": "data"}` and another runner can use it as its input with `{"input_name": "data"}`. These global variables must be unique and the variable defined as input needs to be defined in a previous block as output to prevent cycles in a flow's structure. *Runner* IO names are specific to runner type and model (for `ModelRuntimeRunner`).

Note: IO names can be obtained using the `get_io_specification` method.

11.1.5 Runtime example

In order to create a *KenningFlow* presenting YOLOv4 model performance, create a file `flow_scenario_detection.json` and include the following configuration in it:

```
[
  {
    "type": "kenning.dataproviders.camera_dataprovider.CameraDataProvider",
    "parameters": {
      "video_file_path": "/dev/video0",
      "input_memory_layout": "NCHW",
      "input_width": 608,
      "input_height": 608
    },
    "outputs": {
      "frame": "cam_frame"
    }
  },
  {
    "type": "kenning.runners.modelruntime_runner.ModelRuntimeRunner",
    "parameters": {
      "model_wrapper": {
        "type": "kenning.modelwrappers.object_detection.yolov4.ONNXYOLOV4",
        "parameters": {
          "model_path": "kenning:///models/detection/yolov4.onnx"
        }
      },
      "runtime": {
        "type": "kenning.runtimes.onnx.ONNXRuntime",
        "parameters":
        {
          "save_model_path": "kenning:///models/detection/yolov4.onnx",
          "execution_providers": ["CUDAExecutionProvider"]
        }
      }
    },
    "inputs": {
      "input": "cam_frame"
    },
    "outputs": {
      "detection_output": "predictions"
    }
  },
  {
    "type": "kenning.outputcollectors.real_time_visualizers.
↪RealTimeDetectionVisualizer",
    "parameters": {
      "viewer_width": 512,
      "viewer_height": 512,
      "input_memory_layout": "NCHW",
      "input_color_format": "BGR"
```

(continues on next page)

(continued from previous page)

```
    },
    "inputs": {
      "frame": "cam_frame",
      "detection_data": "predictions"
    }
  }
]
```

This JSON creates a *KenningFlow* that consists of three runners - CameraDataProvider, ModelRuntimeRunner and RealTimeDetectionVisualizer:

- The first one captures frames from a camera and passes it as a `cam_frame` variable.
- The next one passes `cam_frame` to a detection model (in this case YOLOv4) and returns predicted detection objects as `predictions`.
- The last one gets both outputs (`cam_frame` and `predictions`) and shows a detection visualization using DearPyGui.

11.2 KenningFlow execution

Now, you can execute KenningFlow using the above configuration.

With the config saved in the `flow_scenario_detection.json` file, run the `kenning.scenarios.json_flow_runner` as follows:

```
kenning flow --json-cfg flow_scenario_detection.json
```

This module runs KenningFlow defined in given JSON file. With provided config it should read image from the camera and visualize output of detection model YOLOv4.

11.3 Implemented Runners

Available implementations of *Runner* can be found in the *Runner documentation*.

To create custom runners, check *Implementing new Runners for KenningFlow*.

CHAPTER

TWELVE

DEVELOPING KENNING BLOCKS

This chapter describes the development process of Kenning components.

To run below examples it is required to install Kenning with dependencies as follows:

```
pip install "kenning[tensorflow] @ git+https://github.com/antmicro/kenning.git"
```

12.1 Model and I/O metadata

Since not all model formats supported by Kenning provide information about inputs and outputs or other data required for compilation or runtime purposes, each model processed by Kenning comes with a JSON file describing an I/O specification (and other useful metadata).

The JSON file with metadata for file `<model-name>.<ext>` is saved as `<model-name>.<ext>.json` in the same directory.

The example metadata file looks as follows (for ResNet50 for the ImageNet classification problem):

```
{
    "input": [
        {
            "name": "input_0",
            "shape": [1, 224, 224, 3],
            "dtype": "int8",
            "order": 0,
            "scale": 1.0774157047271729,
            "zero_point": -13,
            "prequantized_dtype": "float32"
        }
    ],
    "output": [
        {
            "name": "output_0",
            "shape": [1, 1000],
            "dtype": "int8",
            "order": 0,
            "scale": 0.00390625,
            "zero_point": -128,
            "prequantized_dtype": "float32"
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

A sample metadata JSON file may look as follows (for the YOLOv4 detection model):

```
{
    "input": [
        {
            "name": "input",
            "shape": [1, 3, 608, 608],
            "dtype": "float32"
        }
    ],
    "output": [
        {
            "name": "output",
            "shape": [1, 255, 76, 76],
            "dtype": "float32"
        },
        {
            "name": "output.3",
            "shape": [1, 255, 38, 38],
            "dtype": "float32"
        },
        {
            "name": "output.7",
            "shape": [1, 255, 19, 19],
            "dtype": "float32"
        }
    ],
    "processed_output": [
        {
            "name": "detection_output",
            "type": "List",
            "dtype": {
                "type": "List",
                "dtype": "kenning.datasets.helpers.detection_and_segmentation.
↪DetectObject"
            }
        }
    ],
    "model_version": 4
}
```

In general, the metadata file consist of four fields:

- `input` is a specification of input data received by model wrapper (before preprocessing),
- `processed_input` is a specification of data passed to wrapped model (after preprocessing),
- `output` is a specification of data returned by wrapped model (before postprocessing),

- `processed_output` is a specification of output data returned by model wrapper (after post-processing).

If `processed_input` or `processed_output` is not specified we assume that there is no processing and it is the same as `input` or `output` respectively. Each array consist of dictionaries describing model inputs and outputs.

Additionally, any non-conflicting miscellaneous keywords may be included into the metadata JSON, but those will not be validated by Kenning.

Parameters common to all fields (except for the miscellaneous):

- `name` - input/output name,
- `shape` - input/output tensor shape, if `processed_input` is present, `input` can have list of valid shapes,
- `dtype` - input/output type,

Parameters specific to `processed_input` and `output`:

- `order` - some of the runtimes/optimizers allow accessing inputs and outputs by id. This field describes the id of the current input/output,
- `scale` - scale parameter for the quantization purposes. Present only if the input/output requires quantization/dequantization,
- `zero_point` - zero point parameter for the quantization purposes. Present only if the input/output requires quantization/dequantization,
- `prequantized_dtype` - input/output data type before quantization.
- `class_name` - list of class names from the dataset. It is used by output collectors to present the data in a human-readable way.

Parameters specific to `input`:

- `mean` - mean used to normalize inputs before model training,
- `std` - standard deviation used to normalize inputs before model training.

Parameters specific to `output` and `processed_output`:

- `class_name` - list of class names from the dataset. It is used by output collectors to present the data in a human-readable way.

Parameters specific to all fields which data is not compatible with `np.ndarray`:

- `type` - can be either `List`, `Dict` or `Any`.
- `dtype` - if `type` is `List`, it can specify type of elements, should be represented as whole path which can be imported (i.e. `kenning.datasets.helpers.detection_and_segmentation.SegmObject` for segmentation results). It can also receive dictionary with specification of list items.
- `fields` - if `type` is `Dict`, it should be a dictionary with names as keys and values as dictionary matching `input` and `output` specification. Example from `PyTorchCOCOMaskRCNN`:

```
{
    "type": "Dict",
    "fields": {
```

(continues on next page)

(continued from previous page)

```
        "boxes": {"shape": [-1, 4], "dtype": "float32"},
        "labels": {"shape": [-1,], "dtype": "int64"},
        "scores": {"shape": [-1,], "dtype": "float32"},
        "masks": {
            "shape": [-1, 1, 416, 416],
            "dtype": "float32",
        },
    },
}
```

The model metadata is used by all classes in Kenning in order to understand the format of the inputs and outputs.

Warning: The *Optimizer* objects can affect the format of inputs and outputs, e.g. quantize the network. It is crucial to update the I/O specification when a block modifies it.

12.2 Implementing a new Kenning component

Firstly, check *Kenning API* for available building blocks for Kenning and their documentation. Adding a new block to Kenning is a matter of creating a new class inheriting from one of `kenning.core` classes, providing configuration parameters, and implementing methods - at least the unimplemented ones.

For example purposes, let's create a sample *Optimizer*-based class, which will convert an input model to the TensorFlow Lite format.

First, let's create minimal code with an empty class:

```
from kenning.core.optimizer import Optimizer


class TensorFlowLiteCompiler(Optimizer):
    pass
```

12.2.1 Defining arguments for core classes

Kenning classes can be created in three ways:

- Using constructor in Python,
- Using argparse from command-line (see *Using Kenning via command line arguments*),
- Using JSON-based dictionaries from JSON scenarios (see *Defining optimization pipelines in Kenning*)

To support all three methods, the newly implemented class requires creating a dictionary called `arguments_structure` that holds all configurable parameters of the class, along with their description, type and additional information.

This structure is used to create:

- an `argparse` group, to configure class parameters from terminal level (via the *ArgumentsHandler*'s `form_argparse` method). Later, a class can be created with the `from_argparse` method.
- a JSON schema to configure the class from a JSON file (via the *ArgumentsHandler*'s `form_parameterschema` method). Later, a class can be created with the `from_parameterschema` method.

`arguments_structure` is a dictionary in the following form:

```
arguments_structure = {
    'argument_name': {
        'description': 'Help for the argument',
        'type': str,
        'required': True
    }
}
```

The `argument_name` is a name used in:

- the Python constructor,
- an Argparse argument (in a form of `--argument-name`),
- a JSON argument.

The fields describing the argument are as follows:

- `argparse_name` - if there is a need for a different flag in argparse, it can be provided here,
- `description` - description of the node, displayed for a parsing error for JSON, or in help in case of command-line access,
- `type` - type of argument, i.e.:
  - `Path` from `pathlib` module,
  - `ResourceURI` - `Path` based supporting non-local resources, e.g. from GitHub (`gh://`), Kenning (`kenning://`) or HTTPS (`https://`),
  - `str`,
  - `float`,
  - `int`,
  - `bool`,
  - `list` or `list[type]` or `list[type1 | type2 | ...]`,
- `default` - default value for the argument (used for AutoML - the value should be in the defined ranges/enum),
- `required` - boolean, tells if argument is required or not,
- `enum` - a list of possible values for the argument (used for AutoML),
- `nullable` - tells if argument can be empty (`None`),
- `AutoML` - specify whether argument is used for AutoML flow, supported only in model wrappers,

- `item_range` - tuple of lower and upper bound of numerical argument (`type` is either `int` or `float`) or elements of list argument (`type` is list) (used only for AutoML),
- `list_range` - tuple of lower and upper bound of list argument length (used only for AutoML),
- `overridable` - tells if argument set in JSON can be overridden with argparse (default: `False`).

Let's add parameters to the example class:

```
from pathlib import Path
from kenning.core.optimizer import Optimizer
from kenning.core.dataset import Dataset


class TensorFlowLiteCompiler(Optimizer):
    arguments_structure = {
        'inferenceinputtype': {
            'argparse_name': '--inference-input-type',
            'description': 'Data type of the input layer',
            'default': 'float32',
            'enum': ['float32', 'int8', 'uint8']
        },
        'inferenceoutputtype': {
            'argparse_name': '--inference-output-type',
            'description': 'Data type of the output layer',
            'default': 'float32',
            'enum': ['float32', 'int8', 'uint8']
        },
        'quantize_model': {
            'argparse_name': '--quantize-model',
            'description': 'Tells if model should be quantized',
            'type': bool,
            'default': False
        },
        'dataset_percentage': {
            'description': 'Tells how much data from dataset (from 0.0 to '
                           '1.0) will be used for calibration dataset',
            'type': float,
            'default': 0.25
        }
    }

    def __init__(
            self,
            dataset: Dataset,
            compiled_model_path: Path,
            inferenceinputtype: str = 'float32',
            inferenceoutputtype: str = 'float32',
            dataset_percentage: float = 0.25,
            quantize_model: bool = False):
```

(continues on next page)

(continued from previous page)

```
        self.inferenceinputtype = inferenceinputtype
        self.inferenceoutputtype = inferenceoutputtype
        self.dataset_percentage = dataset_percentage
        self.quantize_model = quantize_model
        super().__init__(dataset, compiled_model_path)

    @classmethod
    def from_argparse(cls, dataset, args):
        return cls(
            dataset,
            args.compiled_model_path,
            args.inference_input_type,
            args.inference_output_type,
            args.dataset_percentage,
            args.quantize_model
        )
```

In addition to defined arguments, there are also default *Optimizer* arguments - the *Dataset* object and path to save the model (`compiled_model_path`).

Also, a `from_argparse` object creator is implemented, since there are additional parameters (`dataset`) to handle. The `from_parameterschema` function is created automatically.

Additionally, if the new class does not inherit from any Kenning core classes, then it should inherit directly from *ArgumentsHandler* class which is responsible for creating `argparse` groups and JSON schema from `arguments_structure`.

The above implementation of arguments is common for all core classes.

12.2.2 Defining supported output and input types

The Kenning classes for consecutive steps are meant to work in a seamless manner, which means providing various ways to pass the model from one class to another.

Usually, each class can accept multiple model input formats and provides at least one output format that can be accepted in other classes (except for `terminal` compilers, such as Apache TVM, that compile models to a runtime library).

The list of supported output formats is represented in a class with an `outputtypes` list:

```
    outputtypes = [
        'tflite'
    ]
```

The supported input formats are delivered in a form of a dictionary, mapping the supported input type name to the function used to load a model:

```
    inputtypes = {
        'keras': kerasconversion,
        'tensorflow': tensorflowconversion
    }
```

Let's update the code with supported types:

```
from kenning.core.optimizer import Optimizer
import tensorflow as tf
from pathlib import Path


def kerasconversion(modelpath: Path):
    model = tf.keras.models.load_model(modelpath)
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    return converter


def tensorflowconversion(modelpath: Path):
    converter = tf.lite.TFLiteConverter.from_saved_model(modelpath)
    return converter


class TensorFlowLiteCompiler(Optimizer):
    arguments_structure = {
        'inferenceinputtype': {
            'argparse_name': '--inference-input-type',
            'description': 'Data type of the input layer',
            'default': 'float32',
            'enum': ['float32', 'int8', 'uint8']
        },
        'inferenceoutputtype': {
            'argparse_name': '--inference-output-type',
            'description': 'Data type of the output layer',
            'default': 'float32',
            'enum': ['float32', 'int8', 'uint8']
        },
        'quantize_model': {
            'argparse_name': '--quantize-model',
            'description': 'Tells if model should be quantized',
            'type': bool,
            'default': False
        },
        'dataset_percentage': {
            'description': 'Tells how much data from dataset (from 0.0 to '
                           '1.0) will be used for calibration dataset',
            'type': float,
            'default': 0.25
        }
    }

    outputtypes = [
        'tflite'
    ]

    inputtypes = {
```

(continues on next page)

(continued from previous page)

```
        'keras': kerasconversion,
        'tensorflow': tensorflowconversion
    }

    def __init__(
            self,
            dataset: Dataset,
            compiled_model_path: Path,
            inferenceinputtype: str = 'float32',
            inferenceoutputtype: str = 'float32',
            dataset_percentage: float = 0.25,
            quantize_model: bool = False):
        self.inferenceinputtype = inferenceinputtype
        self.inferenceoutputtype = inferenceoutputtype
        self.dataset_percentage = dataset_percentage
        self.quantize_model = quantize_model
        super().__init__(dataset, compiled_model_path)

    @classmethod
    def from_argparse(cls, dataset, args):
        return cls(
            dataset,
            args.compiled_model_path,
            args.inference_input_type,
            args.inference_output_type,
            args.dataset_percentage,
            args.quantize_model
        )
```

12.2.3 Implementing unimplemented methods

The remaining aspect of developing Kenning classes is implementing unimplemented methods. Unimplemented methods raise the `NotImplementedError`.

In case of the *Optimizer* class, it is the `compile` method and `get_framework_and_version`.

When implementing unimplemented methods, it is crucial to follow type hints both for inputs and outputs - more details can be found in the documentation for the method. Sticking to the type hints ensures compatibility between blocks, which is required for a seamless connection between compilation components.

Let's finish the implementation of the *Optimizer*-based class:

```
from kenning.core.optimizer import Optimizer
from kenning.core.dataset import Dataset
import tensorflow as tf
from pathlib import Path
from typing import Optional, Dict, List
```

(continues on next page)

(continued from previous page)

```
def kerasconversion(modelpath: Path):
    model = tf.keras.models.load_model(modelpath)
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    return converter


def tensorflowconversion(modelpath: Path):
    converter = tf.lite.TFLiteConverter.from_saved_model(modelpath)
    return converter


class TensorFlowLiteCompiler(Optimizer):
    arguments_structure = {
        'inferenceinputtype': {
            'argparse_name': '--inference-input-type',
            'description': 'Data type of the input layer',
            'default': 'float32',
            'enum': ['float32', 'int8', 'uint8']
        },
        'inferenceoutputtype': {
            'argparse_name': '--inference-output-type',
            'description': 'Data type of the output layer',
            'default': 'float32',
            'enum': ['float32', 'int8', 'uint8']
        },
        'quantize_model': {
            'argparse_name': '--quantize-model',
            'description': 'Tells if model should be quantized',
            'type': bool,
            'default': False
        },
        'dataset_percentage': {
            'description': 'Tells how much data from dataset (from 0.0 to '
                           '1.0) will be used for calibration dataset',
            'type': float,
            'default': 0.25
        }
    }

    outputtypes = [
        'tflite'
    ]

    inputtypes = {
        'keras': kerasconversion,
        'tensorflow': tensorflowconversion
    }

    def __init__(
```

(continues on next page)

(continued from previous page)

```
            self,
            dataset: Dataset,
            compiled_model_path: Path,
            inferenceinputtype: str = 'float32',
            inferenceoutputtype: str = 'float32',
            dataset_percentage: float = 0.25,
            quantize_model: bool = False):
        self.inferenceinputtype = inferenceinputtype
        self.inferenceoutputtype = inferenceoutputtype
        self.dataset_percentage = dataset_percentage
        self.quantize_model = quantize_model
        super().__init__(dataset, compiled_model_path)

    @classmethod
    def from_argparse(cls, dataset, args):
        return cls(
            dataset,
            args.compiled_model_path,
            args.inference_input_type,
            args.inference_output_type,
            args.dataset_percentage,
            args.quantize_model
        )

    def compile(
            self,
            inputmodelpath: Path,
            io_spec: Optional[Dict[str, List[Dict]]] = None):

        # load I/O specification for the model
        if io_spec is None:
            io_spec = self.load_io_specification(inputmodelpath)

        # load the model using chosen input type
        converter = self.inputtypes[self.inputtype](inputmodelpath)

        # preparing model compilation using class arguments
        if self.quantize_model:
            converter.optimizations = [tf.lite.Optimize.DEFAULT]
            converter.target_spec.supported_ops = [
                tf.lite.OpsSet.TFLITE_BUILTINS_INT8
            ]
        converter.inference_input_type = tf.as_dtype(self.inferenceinputtype)
        converter.inference_output_type = tf.as_dtype(self.inferenceoutputtype)

        # dataset can be used during compilation i.e. for calibration
        # purposes
        if self.dataset and self.quantize_model:
            def generator():
```

(continues on next page)

(continued from previous page)

```
                for entry in self.dataset.calibration_dataset_generator(
                        self.dataset_percentage):
                    yield [np.array(entry, dtype=np.float32)]
            converter.representative_dataset = generator

        # compile and save the model
        tflite_model = converter.convert()
        with open(self.compiled_model_path, 'wb') as f:
            f.write(tflite_model)

        # update the I/O specification
        interpreter = tf.lite.Interpreter(model_content=tflite_model)
        signature = interpreter.get_signature_runner()

        def update_io_spec(sig_det, int_det, key):
            for order, spec in enumerate(io_spec[key]):
                old_name = spec['name']
                new_name = sig_det[old_name]['name']
                spec['name'] = new_name
                spec['order'] = order

            quantized = any([det['quantization'][0] != 0 for det in int_det])
            new_spec = []
            for det in int_det:
                spec = [
                    spec for spec in io_spec[key]
                    if det['name'] == spec['name']
                ][0]

                if quantized:
                    scale, zero_point = det['quantization']
                    spec['scale'] = scale
                    spec['zero_point'] = zero_point
                    spec['prequantized_dtype'] = spec['dtype']
                    spec['dtype'] = np.dtype(det['dtype']).name
                new_spec.append(spec)
            io_spec[key] = new_spec

        update_io_spec(
            signature.get_input_details(),
            interpreter.get_input_details(),
            'input'
        )
        update_io_spec(
            signature.get_output_details(),
            interpreter.get_output_details(),
            'output'
        )
```

(continues on next page)

(continued from previous page)

```
        # save updated I/O specification
        self.save_io_specification(inputmodelpath, io_spec)

    def get_framework_and_version(self):
        return 'tensorflow', tf.__version__
```

There are several important things regarding the code snippet above:

- The information regarding inputs and outputs can be collected with the `self.load_io_specification` method, present in all classes.
- The information about the input format to use is delivered in the `self.inputtype` field - it is updated automatically by the function consulting the best supported format for previous and current block.
- If the I/O metadata is affected by the current block, it needs to be updated and saved along with the compiled model using the `self.save_io_specification` method.

12.2.4 Using the implemented block

In the Python script, the above class can be used with other classes from the *Kenning API* as is - it is a regular Python code.

To use the implemented block in the JSON scenario (as described in *Defining optimization pipelines in Kenning*, the module implementing the class needs to be available from the current directory, or the path to the module needs to be added to the `PYTHONPATH` variable.

Let's assume that the class was implemented in the `my_optimizer.py` file. The scenario can look as follows:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "kenning:///models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
        {
            "dataset_root": "./build/pet-dataset"
        }
    },
    "optimizers":
    [
```

(continues on next page)

(continued from previous page)

```
        {
            "type": "my_optimizer.TensorFlowLiteCompiler",
            "parameters":
            {
                "compiled_model_path": "./build/compiled-model.tflite",
                "inference_input_type": "float32",
                "inference_output_type": "float32"
            }
        }
    ],
    "runtime":
    {
        "type": "kenning.runtimes.tflite.TFLiteRuntime",
        "parameters":
        {
            "save_model_path": "./build/compiled-model.tflite"
        }
    }
}
```

The emphasized line demonstrates usage of the implemented `TensorFlowLiteCompiler` from the `my_optimizer.py` script.

This sums up the Kenning development process.

12.3 Implementing Kenning runtime blocks

12.3.1 Implementing new Runners for KenningFlow

The process of creating new *Runner* is almost the same as the process of implementing Kenning components described above, with a few additional steps.

First of all, the new component needs to inherit from a Runner class (not necessarily directly). Then you need to implement the following methods:

- `cleanup` - cleans resources after execution is stopped
- `should_close` - (*optional*) returns boolean indicating whether a runner ended processing and requests closing Possible reasons: a signal from terminal, user request in GUI, end of data to process, an error, and more. Default implementation always returns `False`
- `run` - method that gets runner inputs, processes them and returns obtained results.

Inputs for the runner are passed to the `run` method as a dictionary where the key is the input name specified in the KenningFlow JSON and the value is simply the input value.

E.g. for `DetectionVisualizer` defined in JSON as

```
{
    "type": "kenning.outputcollectors.detection_visualizer.DetectionVisualizer",
    "parameters": {
```

(continues on next page)

(continued from previous page)

```
        "output_width": 608,
        "output_height": 608
    },
    "inputs": {
        "frame": "cam_frame",
        "detection_data": "predictions"
    }
}
```

the `run` method access inputs as follows

```
from typing import Any, Dict

def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
    input_data = inputs['frame']
    output_data = inputs['detection_data']
    self.process_output(input_data, output_data)
```

Note: In the example above, the `run` method does not contain a `return` statement, because this runner does not have any outputs. If you want to create a runner with outputs, this method should return a similar dictionary containing outputs.

12.4 Adjusting ModelWrapper for AutoML flow

12.4.1 Implementing a ModelWrapper

To create an AutoML model the default modelwrapper-api has to implement automl-model-api interface. The description of the ModelWrapper implementation can be found in model-io-metadata section. But it has to be extended with a few additional steps, like in the AutoPyTorch-based *example with a simple fully-connected neural network*:

- add `kenning.core.automl.AutoMLModel` based class inheritance to the ModelWrapper (line 35),
- implement the model class with parameters that can be tuned by AutoML (lines 14-32),
- point to the model class in the ModelWrapper using full class path (line 52),
- define which ModelWrapper arguments can be tuned (lines 41-49),
  - names have to match with the model class parameters,
  - AutoML-specific configuration is described in *Defining arguments for core classes*,
- implement or override inherited methods, in case of `AutoPyTorchModel`, they are:
  - (required) `get_io_specification_from_dataset` - creates IO spec based on the dataset (lines 119-129),
  - (optional) `model_params_from_context` - generates additional parameters for the model based on the dataset and optional platform (lines 65-75),

- (optional) `define_forbidden_clauses` - adds forbidden configurations to the search space,
- (optional) `register_components` - adds custom components to AutoPyTorch.

Listing 12.1: fc_automl.py

```python
from typing import Any, Dict, List, Optional, Tuple

import numpy as np
import torch
from torch import nn

from kenning.automl.auto_pytorch import AutoPyTorchModel
from kenning.core.platform import Platform
from kenning.datasets.anomaly_detection_dataset import AnomalyDetectionDataset
from kenning.modelwrappers.frameworks.pytorch import PyTorchWrapper
from kenning.utils.resource_manager import PathOrURI


class FCNetwork(nn.Sequential):
    """
    The simple fully-connected neural network model.
    """
    def __init__(
        self,
        # The input shape is provided by AutoPyTorch
        input_shape: Tuple[int, ...],
        # Remaining arguments should have the same names as parameters
        # from `arguments_structure` and `model_params_from_context` method
        neurons: List[int],
        outputs: int,
    ):
        self.neurons = neurons
        super().__init__(
            *[nn.Linear(in_, out) for in_, out in zip(
                [input_shape[-1]] + neurons, neurons + [outputs]
            )]
        )


class PyTorchFullyConnected(PyTorchWrapper, AutoPyTorchModel):
    """
    The example AutoML-compatible fully connected model.
    """

    arguments_structure = {
        # Required: Example parameter which can be adjusted by AutoML flow
        "neurons": {
            "description": "List of neurons in consecutive layers",
            "type": list[int],
```

(continues on next page)

(continued from previous page)

```
            "default": [8, 16, 8],
            "AutoML": True,
            "list_range": (1, 16),
            "item_range": (1, 256),
        },
    }
    # Required: The path to the class representing model
    model_class = f"{FCNetwork.__module__}.{FCNetwork.__name__}"

    def __init__(
        self,
        model_path: PathOrURI,
        dataset: AnomalyDetectionDataset,
        from_file: bool = True,
        model_name: Optional[str] = None,
        neurons: List[int] = [8, 16, 8],
    ):
        super().__init__(model_path, dataset, from_file, model_name)
        self.neurons = neurons

    # Optional: Method generating additional model parameters
    # based on the dataset
    @classmethod
    def model_params_from_context(
        cls,
        dataset: AnomalyDetectionDataset,
        platform: Optional[Platform] = None,
    ):
        return {
            "outputs": len(dataset.get_class_names()),
        }

    def create_model_structure(self):
        self.model = FCNetwork(
            input_shape=(
                -1,
                self.dataset.window_size * self.dataset.num_features
            ),
            neurons=self.neurons,
            **self.model_params_from_context(self.dataset)
        )

    def prepare_model(self):
        if self.model_prepared:
            return None

        if self.from_file:
            self.load_model(self.model_path)
            self.model_prepared = True
```

(continues on next page)

(continued from previous page)

```
        else:
            self.create_model_structure()

            def weights_init(m):
                torch.nn.init.xavier_uniform_(m.weight)
                torch.nn.init.zeros_(m.bias)

            self.model.classifier.apply(weights_init)
            self.model_prepared = True
            self.save_model(self.model_path)
        self.model.to(self.device)

    def preprocess_input(self, X: List[Any]) -> List[Any]:
        import torch

        X = np.asarray(X[0])
        return [torch.from_numpy(X.reshape(X.shape[0], -1)).to(self.device)]

    def postprocess_outputs(self, y: List[Any]) -> List[np.ndarray]:
        output = np.argmax(
            y[0].detach().cpu().numpy(),
            axis=-1,
        ).reshape(-1).astype(np.int8)
        return (output,)

    # Required: Method generating IO specification
    # based on the dataset
    def get_io_specification_from_dataset(
        cls,
        dataset,
    ) -> Dict[str, List[Dict]]:
        return cls._get_io_specification(
            dataset.num_features,
            dataset.window_size,
            dataset.batch_size,
        )

    @classmethod
    def _get_io_specification(
        cls,
        num_features,
        window_size,
        batch_size: int = -1,
    ) -> Dict[str, List[Dict]]:
        return {
            "input": [
                {
                    "name": "input_1",
                    "shape": (
```

(continues on next page)

(continued from previous page)

```
                        batch_size,
                        window_size,
                        num_features,
                    ),
                    "dtype": "float32",
                }
            ],
            "processed_input": [
                {
                    "name": "input_1",
                    "shape": (
                        batch_size,
                        window_size * num_features
                        if window_size > 0 and num_features > 0
                        else -1,
                    ),
                    "dtype": "float32",
                }
            ],
            "output": [
                {
                    "name": "distances",
                    "shape": (batch_size, 2),
                    "dtype": "float32",
                }
            ],
            "processed_output": [
                {
                    "name": "anomalies",
                    "shape": (batch_size,),
                    "dtype": "int8",
                }
            ],
        }

    def get_io_specification_from_model(self) -> Dict[str, List[Dict]]:
        return self.get_io_specification_from_dataset(self.dataset)

    @classmethod
    def derive_io_spec_from_json_params(
        cls, json_dict: Dict
    ) -> Dict[str, List[Dict]]:
        cls.get_io_specification(-1, -1)
```

12.4.2 Using the implemented model

To use the model, its full module path has to be specified in the `use_models` parameter:

Listing 12.2: fc-scenario.yml

```yaml
dataset:
  type: AnomalyDetectionDataset
  parameters:
    dataset_root: ./workspace/CATS
    csv_file: kenning:///datasets/anomaly_detection/cats_nano.csv
    split_fraction_test: 0.1
    split_seed: 12345
    inference_batch_size: 1

automl:
  type: AutoPyTorchML
  parameters:
    time_limit: 2
    use_models:
      # Use model wrapper defined in fc_automl.py
      - fc_automl.PyTorchFullyConnected
    output_directory: ./workspace/automl-results
    n_best_models: 5
    optimize_metric: f1
    min_budget: 1
    max_budget: 2
```

Running the simples AutoML flow, assuming the implementation and the scenario are saved under `fc_automl.py` and `fc-scenario.yml` respectively, boils down to:

```
# Make sure the required dependencies are installed
pip install "kenning[torch,anomaly_detection,auto_pytorch] @ git+https://github.
↪com/antmicro/kenning.git"
# Run AutoML flow with new model
PYTHONPATH="$(pwd):$PYTHONPATH" kenning automl --cfg ./fc-scenario.yml
```

CHAPTER

THIRTEEN

KENNING RESOURCES

For managing resources, such as models, compilers, datasets and more Kenning provides *ResourceManager*. It handles:

- Downloading of remote files,
- Resolving custom URL schemes for cleaner and easy to follow file addresses,
- Managing directory with files, including quota checking, old files removal and full cleanup,
- Providing actual path to the resources in the filesystem.

To run below examples it is required to install Kenning as follows:

```
pip install "kenning @ git+https://github.com/antmicro/kenning.git"
```

13.1 Accessing resources

To download a resource you can use `ResourceManager.get_resource` method. The parameters of this method are a URI of the resource and optional output path, where the file should be saved. If the output path is not specified, the file will be saved in `$KENNING_CACHE_DIR/uri.path`.

```
import pathlib
from kenning.utils.resource_manager import ResourceManager, ResourceURI

model_path = pathlib.Path('./model.h5')
ResourceManager().get_resource('kenning:///models/classification/magic_wand.h5',␣
↪model_path)
```

Another way of accessing those resources is to use *ResourceURI* class. You can simply create an instance of this class and use it similarly to `pathlib.Path`:

```
model_path = ResourceURI('kenning:///models/classification/magic_wand.h5')
```

The provided path can be URL to remote resource, path to local file or URI which scheme is defined in `ResourceManager` conversions dictionary.

13.2 Caching

If the resource was not accessed before, the cached file was modified or the remote resource was updated, then it will be downloaded to local cache directory. The location of cache directory can be set using the environment variable `KENNING_CACHE_DIR`. If it is not set, then the default `~/.kenning` path will be used. You can also customize it using `ResourceManager.set_cache_dir` method.

By default, the maximum size of the cache is set to 50 GB, but it can be changed using the environment variable `KENNING_MAX_CACHE_SIZE` or in the runtime using `ResourceManager.set_max_cache_size` method. Before each download the file size is checked and if the available cache space is not large enough, then the oldest files are deleted until there is enough space. If the file is bigger than maximum size of the cache, then the exception is raised.

13.3 Custom URI schemes

As you can see above, the `ResourceManager` supports non-standard URI scheme `kenning://`. In fact, it can support various schemes defined by the user.

The default available schemes are:

- `http://`, `https://`
- `file://` - provides the absolute path to the file in the filesystem
- `kenning://` - a scheme for downloading Kenning-specific resources from `https://dl.antmicro.com`. For example, `kenning:///models/classification/magic_wand.h5` resolves to `https://dl.antmicro.com/kenning/models/classification/magic_wand.h5`
- `gh://` - used for downloading files from Github repositories, e.g. `gh://antmicro:kenning-bare-metal-iree-runtime/sim/config/platforms/springbok.repl;branch=main` resolves to `https://raw.githubusercontent.com/antmicro/kenning-bare-metal-iree-runtime/main/sim/config/platforms/springbok.repl`
- `hf://` - used for downloading files from Hugging Face model hub, e.g. `hf://company/model-id` resolves to `https://huggingface.co/company/model-id`. Using this scheme may require logging in to `huggingface_hub` using `huggingface-cli`.

This allows using links such as:

- `https://dl.antmicro.com/kenning/models/classification/magic_wand.h5`
- `kenning:///models/classification/magic_wand.h5`
- `gh://antmicro:kenning-bare-metal-iree-runtime/sim/config/platforms/springbok.repl;branch=main`
- `file:///some/directory/magic_wand.h5`
- `hf://mistralai/Mistral-7B-Instruct-v0.1`

If the provided path does not have any scheme, the it will be interpreted as a path. For example:

```
ResourceURI('/some/path/magic_wand.h5')
```

Would behave the same as `Path('/some/path/magic_wand.h5')`.

Before the URI is passed to format string or callable converter it is parsed into parts: scheme, netloc, path, params, query and fragment. Additionally the netloc and path are parsed into lists by splitting using either `'.'` or `'/'` separator and the params and query parts are parsed into dictionaries.

The conversion can be:

- `None` - no conversion is done.
- a format string - user can use URI attributes such as: scheme, netloc, path, params, query and fragment enclosed in curly braces. The netloc and path can be also used as lists (i.e. it is possible to use `{path[0]}` or `{netloc[1:]}` inside the format string) and the params and query can be used as dictionary (i.e. you can use `{params[branch]}` for URIs like `gh://antmicro:kenning-bare-metal-iree-runtime/sim/config/platforms/springbok.repl;branch=main`).
- a callable - the provided URI is parsed and passed into it. The callable can have parameters such as `scheme`, `netloc`, `path`, `params`, `query`, `netloc_list`, `path_list`, `params_dict`, `query_dict` and should return string. The parameters with `_list` or `_dicts` suffix are corresponding parts parsed into lists or dicts. Example implementation of such conversion can be seen above as `_gh_converter`.

The above default scheme conversions can be defined as:

```
BASE_URL_SCHEMES = {
    'http': None,
    'https': None,
    'kenning': 'https://dl.antmicro.com/kenning/{path}',
    'gh': _gh_converter,
    'file': lambda path: Path(path).expanduser().resolve(),
}
```

The keys in this dictionary are scheme names, and values are conversion methods.

The `_gh_converter` is defined as:

```
from typing import Dict

def _gh_converter(netloc: str, path: str, params_dict: Dict[str, str]) -> str:
    netloc = netloc.split(':')
    return (
        f'https://raw.githubusercontent.com/{netloc[0]}/{netloc[1]}/'
        f'{params_dict["branch"]}{path}'
    )
```

Those schemes can be customized by user using the `ResourceManager.add_custom_url_schemes` method which takes a dictionary similar to the above as an input - the key is the name of scheme and the value is the converter (`None`, string or callable). If any scheme already has defined conversion, then it is overwritten.

13.4 CLI commands

You can manage cache using Kenning CLI commands. The available commands are:

- `kenning cache list_files` - list cached files, their size and total cache size. To see full paths add `-v` argument.
- `kenning cache clear` - removes all cached files,
- `kenning cache settings` - prints default cache directory path and default max cache size.

CHAPTER

FOURTEEN

KENNING PLATFORMS

Kenning platforms were created to automatically provide board-specific arguments and parameters required for various stages of Kenning pipelines. It is used to:

- Provide information on platforms' constraints, such as RAM size
- Provide information for optimizing the model using compilers such as TVM or IREE
- Provide information on supported target runtime (Linux-based, bare metal or Zephyr-based)
- Provide information on building the target evaluation application
- Provide information on interfacing with the target platform (e.g. via UART)

14.1 Specification

Platforms' definitions should be written in YAML or JSON file representing a dictionary, where:

- keys are the unique boards' IDs
- values are dictionaries with board-specific parameters.

The example specification of platforms looks as follows:

```
max32690evkit/max32690/m4:
  display_name: MAX32690 Evaluation Kit
  flash_size_kb: 3072
  ram_size_kb: 128
  uart_port_wildcard: /dev/serial/by-id/usb-FTDI_FT231X_USB_UART_*-if00-port0
  uart_baudrate: 115200
  uart_log_baudrate: 115200
  compilation_flags:
  - "-keys=arm_cpu,cpu"
  - "-device=arm_cpu"
  - "-march=armv7e-m"
  - "-mcpu=cortex-m4"
  - "-model=max32690"
  openocd_flash_cmd:
  - source [find interface/cmsis-dap.cfg]
  - source [find target/max32690.cfg]
  - init
```

(continues on next page)

(continued from previous page)

```
  - targets
  - reset init
  - flash write_image erase {binary_path}
  - reset run
  - shutdown
  platform_resc_path: gh://antmicro:kenning-zephyr-runtime/renode/scripts/
↪max32690evkit.resc;branch=main
  default_platform: ZephyrPlatform
  default_optimizer:
    - TFLiteCompiler
    - TVMCompiler

max32690fthr/max32690/m4:
  display_name: MAX32690FTHR
  flash_size_kb: 3072
  ram_size_kb: 128
  uart_baudrate: 115200
  uart_log_baudrate: 115200
  compilation_flags:
  - "-keys=arm_cpu,cpu"
  - "-device=arm_cpu"
  - "-march=armv7e-m"
  - "-mcpu=cortex-m4"
  - "-model=max32690"
  platform_resc_path: gh://antmicro:kenning-zephyr-runtime/renode/scripts/
↪max32690fthr.resc;branch=main
  default_platform: ZephyrPlatform
  default_optimizer:
    - TFLiteCompiler
    - TVMCompiler

max78002evkit/max78002/m4:
  display_name: MAX78002 Evaluation Kit
  ai8x_device: MAX78002
  ai8x_device_id: 87
  ai8x_weights_memory_kb: 2048
  ai8x_data_memory_kb: 1280
  flash_size_kb: 2560
  ram_size_kb: 64
  uart_port_wildcard: /dev/serial/by-id/usb-FTDI_FT231X_USB_UART_*-if00-port0
  uart_baudrate: 115200
  uart_log_baudrate: 115200
  compilation_flags:
  - "-keys=arm_cpu,cpu"
  - "-device=arm_cpu"
  - "-march=armv7e-m"
  - "-mcpu=cortex-m4"
  - "-model=max78002"
  openocd_flash_cmd:
```

(continues on next page)

(continued from previous page)

```
  - source [find interface/cmsis-dap.cfg]
  - source [find target/max78002.cfg]
  - init
  - targets
  - reset init
  - flash write_image erase {binary_path}
  - reset run
  - shutdown
  default_platform: ZephyrPlatform
  default_optimizer:
    - Ai8xCompiler
    - TFLiteCompiler
    - TVMCompiler

rv32-springbok:
  display_name: RISC-V 32-bit Springbok
  runtime_binary_path: kenning:///renode/springbok/iree_runtime
  platform_resc_path: gh://antmicro:kenning-bare-metal-iree-runtime/sim/config/
↪springbok.resc;branch=main
  resc_dependencies:
  - gh://antmicro:kenning-bare-metal-iree-runtime/sim/config/platforms/springbok.
↪repl;branch=main
  - gh://antmicro:iree-rv32-springbok/sim/config/infrastructure/SpringbokRiscV32.
↪cs;branch=repo-as-submodule
  post_start_commands:
  - sysbus.vec_controlblock WriteDoubleWord 0xc 0
  runtime_log_init_msg: Runtime started
  default_platform: BareMetalPlatform
  default_optimizer:
    - IREECompiler

stm32f746g_disco:
  display_name: STM32F746G Discovery
  flash_size_kb: 1024
  ram_size_kb: 256
  uart_baudrate: 115200
  platform_resc_path: gh://antmicro:kenning-zephyr-runtime/renode/scripts/
↪stm32f746g_disco.resc;branch=main
  default_platform: ZephyrPlatform
  default_optimizer:
    - TFLiteCompiler
    - TVMCompiler

hifive_unmatched/fu740/s7:
  display_name: HiFive Unmatched
  uart_baudrate: 115200
  platform_resc_path: gh://antmicro:kenning-zephyr-runtime/renode/scripts/hifive_
↪unmatched.resc;branch=main
  default_platform: ZephyrPlatform
```

(continues on next page)

(continued from previous page)

```
  default_optimizer:
    - TFLiteCompiler
    - TVMCompiler
```

Looking at MAX32690 platform there are fields for:

- TVM compilation (`compilation_flags`),
- flashing to HW (`openocd_flash_cmd`),
- simulation (`platform_resc_path`),
- defining model constrains (`flash_size_kb`, `ram_size_kb`),
- automatically finding its UART port (`uart_port_wildcard`) and specifying its baud rate (`uart_baudrate`, `uart_log_baudrate`),
- choosing the platform type (`default_platform`).

Other parameters with descriptions can be found e.g. in:

- `kenning.platforms.bare_metal.BareMetalPlatform` encapsulating platforms running bare metal runtime
- `kenning.platforms.zephyr.ZephyrPlatform` encapsulating platforms running Kenning Zephyr Runtime

14.2 Generating definitions for boards supported by Zephyr RTOS

Kenning provides a `generate-platforms` subcommand that allows to generate definitions of platforms based on given sources. Currently, the only supported sources of information are Zephyr RTOS device trees.

In order to generate definitions, fully set-up Zephyr repository with SDK is required. For detailed information on how to set up the SDK, follow Zephyr documentation.

Based on Docker image for Kenning Zephyr Runtime, it boils down to:

```
docker run --rm -it -v $(pwd):$(pwd) -w $(pwd) ghcr.io/antmicro/kenning-zephyr-
↪runtime:latest /bin/bash
git clone https://github.com/antmicro/kenning-zephyr-runtime
cd kenning-zephyr-runtime/
./scripts/prepare_zephyr_env.sh
# Install additional ARM toolchain
~/.local/opt/zephyr-sdk/setup.sh -t aarch64-zephyr-elf
source .venv/bin/activate
pip3 install 'kenning[zephyr] @ git+https://github.com/antmicro/kenning.git'
```

Note: Using existing Zephyr repository can be achieved by exporting `ZEPHYR_BASE` variable or providing `--zephyr-base` flag to the script. It can potentially require additional dependencies from `kenning[zephyr]`.

Platform generation can be triggered with a following command:

```
kenning generate-platforms zephyr \
  --zephyr-base ../zephyr \
  --platforms ./platforms.yml
```

The script:

- generates flat device trees using Zephyr CMake rules,
- parses received device trees with dts2repl,
- tries to find baudrate based on chosen console output,
- tries to find compilation flags for ARM CPUs based on their names (from `compatible` parameter) and architectures returned by GCC for ARM (`arm-zephyr-eabi-gcc` and `aarch64-zephyr-elf-gcc` from Zephyr SDK),
- calculates size of the flash based on chosen flash registers,
- calculates size of the RAM based on chose sram and Zephyr's memory-regions,
- finds URL to the board image.

To use newly generated platforms, provide its path to the `--platform-definitions` flag in `kenning` command or directly in the scenario:

```
platform:
  type: ZephyrPlatform
  parameters:
    # Chooses the file with platform definition
    platforms_definitions: [./platforms.yml]
    # Chooses MAX32690 Evaluation Kit
    name: max32690evkit/max32690/m4
```

CHAPTER

FIFTEEN

KENNING API

15.1 Deployment API overview

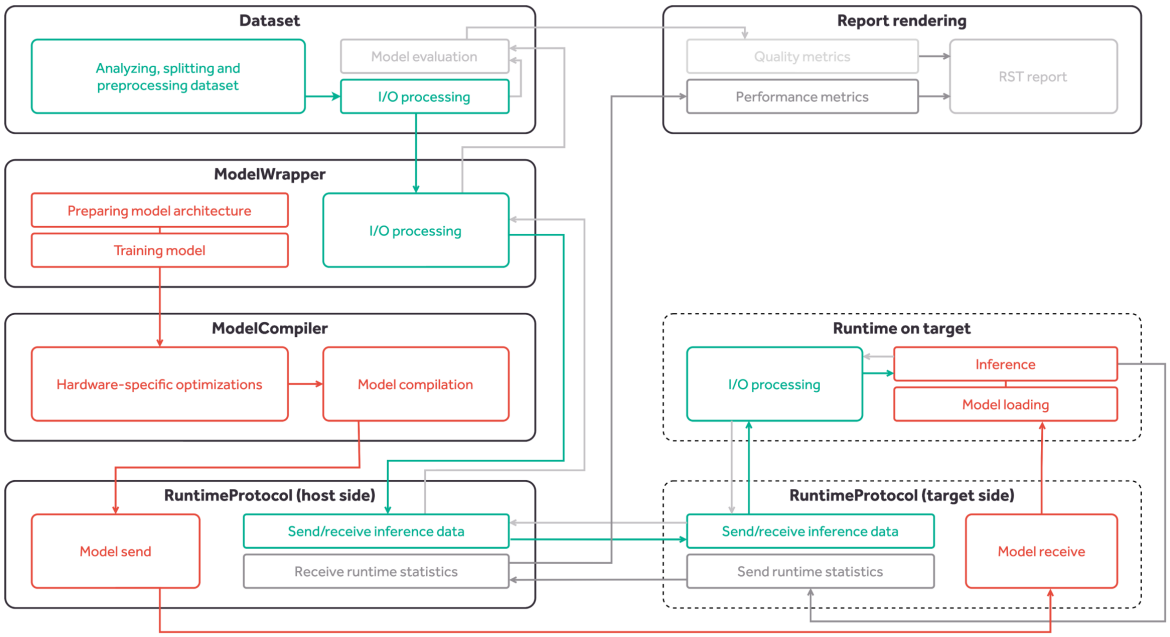


Figure 15.1: Kenning core classes and interactions between them. The green blocks represent the flow of input data passed to the model for inference. The orange blocks represent the flow of model deployment, from training to inference on target device. The grey blocks represent the inference results and metrics flow.

Kenning provides:

- a *Dataset* class - performs dataset download, preparation, input preprocessing, output postprocessing and model evaluation,
- a *ModelWrapper* class - trains the model, prepares the model, performs model-specific input preprocessing and output postprocessing, runs inference on host using a native framework,
- a *Optimizer* class - optimizes and compiles the model,
- a *Runtime* class - loads the model, performs inference on compiled model, runs target-specific processing of inputs and outputs, and runs performance benchmarks,
- a *Protocol* class - implements the communication protocol between the host and the target,

- a *DataConverter* class - performs conversion of data between different formats used by surrounding blocks,
- a *DataProvider* class - implements providing data for inference from such sources as camera, TCP connection, or others,
- a *OutputCollector* class - implements parsing and utilizing data coming from inference (such as displaying visualizations or sending results via TCP).

15.1.1 Model processing

The orange blocks and arrows in Figure 15.1 represent a model's life cycle:

- the model is designed, trained, evaluated and improved - the training is implemented in the *ModelWrapper*.

  Note: This is an optional step - an already trained model can also be wrapped and used.

- the model is passed to the *Optimizer* where it is optimized for given hardware and later compiled,
- during inference testing, the model is sent to the target using *Protocol*,
- the model is loaded on target side and used for inference using *Runtime*.

Once the development of the model is complete, the optimized and compiled model can be used directly on target device using *Runtime*.

15.1.2 I/O data flow

The data flow is represented in the Figure 15.1 with green blocks. The input data flow is depicted using green arrows, and the output data flow is depicted using grey arrows.

Firstly, the input and output data is loaded from dataset files and processed. Later, since every model has its specific input preprocessing and output postprocessing routines, the data is passed to the *ModelWrapper* methods in order to apply modifications. During inference testing, the data is sent to and from the target using *Protocol*.

Lastly, since *Runtime*s also have their specific representations of data, proper I/O processing is applied.

15.1.3 Data flow reporting

Report rendering requires performance metrics and quality metrics. The flow for this is presented with grey lines and blocks in Figure 15.1.

On target side, performance metrics are computed and sent back to the host using the *Protocol*, and later passed to report rendering. After the output data goes through processing in the *Runtime* and *ModelWrapper*, it is compared to the ground truth in the *Dataset* during model evaluation. In the end, the results of model evaluation are passed to report rendering.

The final report is generated as an RST file containing figures, as can be observed in the *Sample autogenerated report*.

15.2 KenningFlow

`kenning.core.flow.KenningFlow` class allows for creation and execution of arbitrary flows built of runners. It is responsible for validating all runners provided in a config file and their IO compatibility.

class kenning.core.flow.**KenningFlow**(runners: *list[*Runner*]*)

Bases: `object`

Allows for creation of custom flows using Kenning core classes.

KenningFlow class creates and executes customized flows consisting of the runners implemented based on kenning.core classes, such as DatasetProvider, ModelRunner, OutputCollector. Designed flows may be formed into non-linear, graph-like structures.

The flow may be defined either directly via dictionaries or in a predefined JSON format.

The JSON format must follow well defined structure. Each runner should consist of following entries:

type - Type of a Kenning class to use for this module parameters - Inner parameters of chosen class inputs - Optional, set of pairs (local name, global name) outputs - Optional, set of pairs (local name, global name)

All global names (inputs and outputs) must be unique. All local names are predefined for each class. All variables used as input to a runner must be defined as a output of a runner that is placed before that runner.

classmethod form_parameterschema() → dict

Creates schema for the KenningFlow class.

Returns
Schema for the class.

Return type
Dict

classmethod from_json(runners_specifications: *list[dict[str, Any]]*) → *KenningFlow*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the json schema defined in `form_parameterschema`. If it is then it parses json and invokes the constructor.

Parameters

runners_specifications : List[Dict[str, Any]]
List of runners that creates the flow.

Returns
Object of class KenningFlow.

Return type
KenningFlow

Raises

- **jsonschema.ValidationError** – Raised for invalid JSON description

- **Exception** – Raised for undefined and redefined variables, depending on context

run()

Main process function. Repeatedly runs constructed graph in a loop.

run_single_step()

Runs flow one time.

15.3 Runner

`kenning.core.runner.Runner` - based classes are responsible for executing various operation in KenningFlow (i.e. data providing, model execution, data visualization).

The available runner implementations are:

- *DataProvider* - base class for data providing,
- ModelRuntimeRunner - for running model inference,
- *OutputCollector* - for processing model output.

class kenning.core.runner.**Runner**(inputs_sources: *dict[str, tuple[int, str]]*, inputs_specs: *dict[str, dict]*, outputs: *dict[str, str]*)

Bases: `IOInterface`, `ArgumentsHandler`, `ABC`

Represents an operation block in Kenning Flow.

cleanup()

Method that cleans resources after Runner is no longer needed.

classmethod from_argparse(args: *Namespace*, inputs_sources: *dict[str, tuple[int, str]]*, inputs_specs: *dict[str, dict]*, outputs: *dict[str, str]*) → *Runner*

Constructor wrapper that takes the parameters from argparse args.

This method takes the arguments created in form_argparse and uses them to create the object.

Parameters

args : Namespace
Arguments from ArgumentParser object.

inputs_sources : Dict[str, Tuple[int, str]]
Input from where data is being retrieved.

inputs_specs : Dict[str, Dict]
Specifications of runner's inputs.

outputs : Dict[str, str]
Outputs of this Runner.

Returns
Object of class Runner.

Return type
Runner

classmethod from_json(json_dict: *dict*, inputs_sources: *dict[str, tuple[int, str]]*, inputs_specs: *dict[str, dict]*, outputs: *dict[str, str]*) → *Runner*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the json schema defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

inputs_sources : Dict[str, Tuple[int, str]]
Input from where data is being retrieved.

inputs_specs : Dict[str, Dict]
Specifications of runner's inputs.

outputs : Dict[str, str]
Outputs of this Runner.

Returns
Object of class Runner.

Return type
Runner

abstract run(inputs: *dict[str, Any]*) → dict[str, Any]

Method used to run this Runner.

Parameters

inputs : Dict[str, Any]
Inputs provided to this block.

Returns
Output of this block.

Return type
Dict[str, Any]

should_close() → bool

Method that checks if Runner got some exit indication (exception etc.) and the flow should close.

Returns
True if there was some exit indication.

Return type
bool

15.4 Dataset

`kennning.core.dataset.Dataset` - based classes are responsible for:

- dataset preparation, including download routines (use the `--download-dataset` flag to download the dataset data),
- input preprocessing into a format expected by most models for a given task,
- output postprocessing for the evaluation process,
- model evaluation based on its predictions,
- sample subdivision into training and validation datasets.

The Dataset objects are used by:

- *ModelWrapper* - for training purposes and model evaluation,
- *Optimizer* - can be used e.g. for extracting a calibration dataset for quantization purposes,
- *Runtime* - for model evaluation on target hardware.

The available dataset implementations are included in the `kenning.datasets` submodule. Example implementations:

- PetDataset for classification,
- OpenImagesDatasetV6 for object detection,
- RandomizedClassificationDataset.

class kenning.core.dataset.**Dataset**(root: *Path*, batch_size: *int* = 1, download_dataset: *bool* = **True**, force_download_dataset: *bool* = **False**, external_calibration_dataset: *Path* | *None* = **None**, split_fraction_test: *float* = 0.2, split_fraction_val: *float* | *None* = **None**, split_seed: *int* = 1234, dataset_percentage: *float* = 1)

Bases: `ArgumentsHandler`, `ABC`

Wraps the datasets for training, evaluation and optimization.

This class provides an API for datasets used by models, compilers (i.e. for calibration) and benchmarking scripts.

Each Dataset object should implement methods for:

- processing inputs and outputs from dataset files,
- downloading the dataset,
- evaluating the model based on dataset's inputs and outputs.

The Dataset object provides routines for iterating over dataset samples with configured batch size, splitting the dataset into subsets and extracting loaded data from dataset files for training purposes.

dataX

List of input data (or data representing input data, i.e. file paths).

Type

List[Any]

dataY

List of output data (or data representing output data).

Type
List[Any]

batch_size

The batch size for the dataset.

Type
int

_dataindex

ID of the next data to be delivered for inference.

Type
int

dataXtrain

dataX subset representing a training set. Available after executing train_test_split_representations, otherwise empty.

Type
List[Any]

dataYtrain

dataY subset representing a training set. Available after executing train_test_split_representations, otherwise empty.

Type
List[Any]

dataXtest

dataX subset representing a testing set. Available after executing train_test_split_representations, otherwise empty.

Type
List[Any]

dataYtest

dataY subset representing a testing set. Available after executing train_test_split_representations, otherwise empty.

Type
List[Any]

dataXval

Optional dataX subset representing a validation set. Available after executing train_test_split_representations, otherwise empty.

Type
List[Any]

dataYval

Optional dataY subset representing a validation set. Available after executing train_test_split_representations, otherwise empty.

Type
List[Any]

calibration_dataset_generator(percentage: *float* = 0.25, seed: *int* | *None* = **None**) → Generator[list[Any], None, None]

Creates generator for the calibration data.

Parameters

percentage : float
The fraction of data to use for calibration.

seed : Optional[int]
The seed for random state, by default seed used for the dataset split.

Yields
List[Any] – List with batch input samples for calibration

abstract download_dataset_fun()

Downloads the dataset to the root directory defined in the constructor.

abstract evaluate(predictions: *list*, truth: *list*) → *Measurements*

Evaluates the model based on the predictions.

The method should compute various quality metrics fitting for the problem the model solves - i.e. for classification it may be accuracy, precision, G-mean, for detection it may be IoU and mAP.

The evaluation results should be returned in a form of Measurements object.

Parameters

predictions : List
The list of predictions from the model.

truth : List
The ground truth for given batch.

Returns
The dictionary containing the evaluation results.

Return type
Measurements

classmethod from_argparse(args: *Namespace*) → *Dataset*

Constructor wrapper that takes the parameters from argparse args.

This method takes the arguments created in `form_argparse` and uses them to create the object.

Parameters

args : Namespace
Arguments from ArgumentParser object.

Returns
Object of class Dataset.

Return type
Dataset

classmethod from_json(json_dict: *dict*) → *Dataset*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the `arguments_structure` defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

Returns
Object of class Dataset.

Return type
Dataset

abstract get_class_names() → list[str]

Returns list of class names in order of their IDs.

Returns
List of class names.

Return type
List[str]

get_data() → tuple[list, list]

Returns the tuple of all inputs and outputs for the dataset.

Warning: It loads all entries with prepare_input_samples and prepare_output_samples to the memory - for large datasets it may result in filling the whole memory.

Returns
The list of data samples.

Return type
Tuple[List, List]

get_data_unloaded() → tuple[list, list]

Returns the input and output representations before loading.

The representations can be opened using prepare_input_samples and prepare_output_samples.

Returns
The list of data samples representations.

Return type
Tuple[List, List]

abstract get_input_mean_std() → tuple[Any, Any]

Returns mean and std values for input tensors.

The mean and std values returned here should be computed using `compute_input_mean_std` method.

Returns
Tuple of two variables describing mean and standardization values for a given train dataset.

Return type
Tuple[Any, Any]

iter_test() → DatasetIterator

Iterates over test data obtained from split.

Returns
Iterator over data samples.

Return type
DatasetIterator

iter_train() → DatasetIterator

Iterates over train data obtained from split.

Returns
Iterator over data samples.

Return type
DatasetIterator

iter_val() → DatasetIterator

Iterates over validation data obtained from split.

Returns
Iterator over data samples.

Return type
DatasetIterator

abstract prepare()

Prepares dataX and dataY attributes based on the dataset contents.

This can i.e. store file paths in dataX and classes in dataY that will be later loaded using prepare_input_samples and prepare_output_samples.

prepare_external_calibration_dataset(percentage: *float* = 0.25, seed: *int* = 12345) → list[Path]

Prepares the data for external calibration dataset.

This method is supposed to scan external_calibration_dataset directory and prepares the list of entries that are suitable for the prepare_input_samples method.

This method is called by the `calibration_dataset_generator` method to get the data for calibration when external_calibration_dataset is provided.

By default, this method scans for all files in the directory and returns the list of those files.

Parameters

percentage : float
Percentage of dataset to be used.

seed : int
Random state seed.

Returns
List of objects that are usable by the `prepare_input_samples` method.

Return type
List[Path]

prepare_input_samples(samples: *list*) → list

Prepares input samples, i.e. load images from files, converts them.

By default the method returns data as is - without any conversions. Since the input samples can be large, it does not make sense to load all data to the memory - this method handles loading data for a given data batch.

Parameters

samples : List
List of input samples to be processed.

Returns
Preprocessed input samples.

Return type
List

prepare_output_samples(samples: *list*) → list

Prepares output samples.

By default the method returns data as is. It can be used i.e. to create the one-hot output vector with class association based on a given sample.

Parameters

samples : List
List of output samples to be processed.

Returns
Preprocessed output samples.

Return type
List

save_dataset_checksum()

Writes dataset checksum to file.

set_batch_size(batch_size: *int*)

Sets the batch size of the data in the iterator batches.

Parameters

batch_size : int
Number of input samples per batch.

test_subset_len() → int | None

Returns the length of a single batch from the training set.

Returns
The number of samples in a single batch from the testing set or None if the dataset has not been split

Return type
Optional[int]

train_subset_len() → int | None

Returns the length of a single batch from the training set.

Returns
The number of samples in a single batch from the training set or None if the dataset has not been split

Return type
Optional[int]

train_test_split_representations(test_fraction: *float* | *None* = **None**, val_fraction: *float* | *None* = **None**, seed: *int* | *None* = **None**, stratify: *bool* = **True**, append_index: *bool* = **False**) → tuple[list, ...]

Splits the data representations into train dataset and test dataset.

Parameters

test_fraction : Optional[float]
The fraction of data to leave for model testing.

val_fraction : Optional[float]
The fraction of data to leave for model validation.

seed : Optional[int]
The seed for random state.

stratify : bool
Whether to stratify the split.

append_index : bool
Whether to return the indices of the split. If True, the returned tuple will have indices appended at the end. For example, if the split is (X_train, X_test, y_train, y_test), the returned tuple will be (X_train, X_test, y_train, y_test, train_indices, test_indices).

Returns
Split data into train, test and optionally validation subsets.

Return type
Tuple[List, . . .]

val_subset_len() → int | None

Returns the length of a single batch from the training set.

Returns
The number of samples in a single batch from the validation set or None if the dataset has not been split

Return type
Optional[int]

verify_dataset_checksum() → bool

Checks whether dataset is already downloaded in its directory.

Returns
True if dataset is downloaded.

Return type
bool

15.5 ModelWrapper

`kenning.core.model.ModelWrapper` base class requires implementing methods for:

- model preparation,
- model saving and loading,
- model saving to the ONNX format,
- model-specific preprocessing of inputs and postprocessing of outputs, if necessary,
- model inference,
- providing metadata (framework name and version),
- model training,
- input format specification,
- conversion of model inputs and outputs to bytes for the `kenning.core.protocol.Protocol` objects.

The `ModelWrapper` provides methods for running inference in a loop for data from a dataset and measuring the model's quality and inference performance.

The `kenning.modelwrappers.frameworks` submodule contains framework-wise implementations of the `ModelWrapper` class - they implement all methods common for given frameworks regardless of the model used.

For the `Pet Dataset wrapper` object, there is an example classifier implemented in TensorFlow 2.x called `TensorFlowPetDatasetMobileNetV2 <https://github.com/antmicro/kenning/blob/main/kenning/modelwrappers/classification/tensorflow_pet_dataset.py>_`.

Model wrapper examples:

- PyTorchWrapper and TensorFlowWrapper implement common methods for all PyTorch and TensorFlow framework models,
- PyTorchPetDatasetMobileNetV2 wraps the MobileNetV2 model for Pet classification implemented in PyTorch,
- TensorFlowDatasetMobileNetV2 wraps the MobileNetV2 model for Pet classification implemented in TensorFlow,
- TVMDarknetCOCOYOLOV3 wraps the YOLOv3 model for COCO object detection implemented in Darknet (without training and inference methods).

class kenning.core.model.**ModelWrapper**(model_path: *Path* | ResourceURI, dataset: Dataset | *None*, from_file: *bool* = **True**, model_name: *str* | *None* = **None**)

Bases: `IOInterface`, `ArgumentsHandler`, `ABC`

Wraps the given model.

abstract convert_input_to_bytes(inputdata: *Any*) → bytes

Converts the input returned by the `preprocess_input` method to bytes.

Parameters

inputdata : Any
The preprocessed inputs.

Returns
Input data as byte stream.

Return type
bytes

abstract convert_output_from_bytes(outputdata: *bytes*) → list[Any]

Converts bytes array to the model output format.

The converted output should be compatible with `postprocess_outputs` method.

Parameters

outputdata : bytes
Output data in raw bytes.

Returns
List of output data from a model. The converted data should be compatible with the `postprocess_outputs` method.

Return type
List[Any]

abstract classmethod derive_io_spec_from_json_params(json_dict: *dict*) → dict[str, list[dict]]

Creates IO specification by deriving parameters from parsed JSON dictionary. The resulting IO specification may differ from the results of *get_io_specification*, information that couldn't be retrieved from JSON parameters are absent from final IO spec or are filled with general value (example: '-1' for unknown dimension shape).

Parameters

json_dict : Dict
JSON dictionary formed by parsing the input JSON with ModelWrapper's parameterschema.

Returns
Dictionary that conveys input and output layers specification.

Return type
Dict[str, List[Dict]]

classmethod from_argparse(dataset: Dataset | *None*, args: *Namespace*, from_file: *bool* = **True**) → *ModelWrapper*

Constructor wrapper that takes the parameters from argparse args.

Parameters

dataset : Optional[*Dataset*]
The dataset object to feed to the model.

args : Namespace
Arguments from ArgumentParser object.

from_file : bool
Determines if the model should be loaded from model_path.

Returns
Object of class ModelWrapper.

Return type
ModelWrapper

classmethod from_json(json_dict: *dict*, dataset: Dataset | *None* = **None**, from_file: *bool* = **True**) → *ModelWrapper*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the `arguments_structure` defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

dataset : Optional[*Dataset*]
The dataset object to feed to the model.

from_file : bool
Determines if the model should be loaded from model_path.

Returns
Object of class ModelWrapper.

Return type
ModelWrapper

abstract get_framework_and_version() → tuple[str, str]

Returns name of the framework and its version in a form of a tuple.

Returns
Framework name and version.

Return type
Tuple[str, str]

get_io_specification() → dict[str, list[dict]]

Returns a saved dictionary with *input* and *output* keys that map to input and output specifications.

A single specification is a list of dictionaries with names, shapes and dtypes for each layer. The order of the dictionaries is assumed to be expected by the *ModelWrapper*.

It is later used in optimization and compilation steps.

Returns
Dictionary that conveys input and output layers specification.

Return type
Dict[str, List[Dict]]

abstract get_io_specification_from_model() → dict[str, list[dict]]

Returns a new instance of dictionary with *input* and *output* keys that map to input and output specifications.

A single specification is a list of dictionaries with names, shapes and dtypes for each layer. The order of the dictionaries is assumed to be expected by the *ModelWrapper*.

It is later used in optimization and compilation steps.

It is used by *get_io_specification* function to get the specification and save it for later use.

Returns

Dictionary that conveys input and output layers specification.

Return type

Dict[str, List[Dict]]

abstract classmethod get_output_formats() → list[str]

Returns list of names of possible output formats.

Returns

List of possible output format names.

Return type

List[str]

get_path() → Path | *ResourceURI*

Returns path to the model in a form of a Path or ResourceURI object.

Returns

Path or URI to the model.

Return type

PathOrURI

abstract load_model(model_path: *Path* | ResourceURI)

Loads the model from file.

Parameters

model_path : PathOrURI

Path or URI to the model file.

classmethod parse_io_specification_from_json(json_dict)

Return dictionary with 'input' and 'output' keys that will map to input and output specification of an object created by the argument json schema.

A single specification is a list of dictionaries with names, shapes and dtypes for each layer.

Since no object initialization is done for this method, some IO specification may be incomplete, this method fills in -1 in case the information is missing from the JSON dictionary.

Parameters

json_dict : Dict

Parameters for object constructor in JSON format.

Returns
Dictionary that conveys input and output layers specification.

Return type
Dict[str, List[Dict]]

`postprocess_outputs`(y: *list[Any]*) → list[Any]

Processes the outputs for a given model.

By default no action is taken, and the outputs are passed unmodified.

Parameters

y : List[Any]
The list of output data from the model.

Returns
The post processed outputs from the model that need to be in format requested by the Dataset object.

Return type
List[Any]

`abstract prepare_model`()

Downloads the model (if required) and loads it to the device.

Should be used whenever the model is actually required.

The prepare_model method should check model_prepared field to determine if the model is not already loaded.

It should also set model_prepared field to True once the model is prepared.

`preprocess_input`(X: *list[Any]*) → list[Any]

Preprocesses the inputs for a given model before inference.

By default no action is taken, and the inputs are passed unmodified.

Parameters

X : List[Any]
The input data from the Dataset object.

Returns
The preprocessed inputs that are ready to be fed to the model.

Return type
List[Any]

`read_platform`(platform: Platform)

Reads Platform data to configure model.

Platform-based entities come with lots of information on hardware architecture that can be used by the ModelWrapper class.

By default no data is read.

It is important to take into account that different Platform-based classes come with a different sets of attributes.

Parameters

platform : *Platform*
object with platform details

abstract run_inference(X: *list[Any]*) → list[Any]

Runs inference for a given preprocessed input.

Parameters

X : List[Any]
The preprocessed inputs for the model.

Returns
The results of the inference.

Return type
List[Any]

abstract save_model(model_path: *Path* | ResourceURI)

Saves the model to file.

Parameters

model_path : PathOrURI
Path or URI to the model file.

abstract save_to_onnx(model_path: *Path* | ResourceURI)

Saves the model in the ONNX format.

Parameters

model_path : PathOrURI
Path or URI to the model file.

test_inference() → *Measurements*

Runs the inference on test split of the dataset.

Returns
The inference results.

Return type
Measurements

train_model()

Trains the model with a given dataset.

This method should implement training routine for a given dataset and save a working model to a given path in a form of a single file.

The training should be performed with given batch size, learning rate, and number of epochs.

The model needs to be saved explicitly.

Raises
NotImplementedError – Raised when method is not implemented

15.6 Optimizer

`kenning.core.optimizer.Optimizer` objects wrap the deep learning compilation process. They can perform the model optimization (operation fusion, quantization) as well. Kenning supports executing optimizations also on the target device. To do so you can use `location` parameter which specifies where given *Optimizer* would be executed (either `'host'` or `'target'`).

All Optimizer objects should provide methods for compiling models in ONNX format, but they can also provide support for other formats (like Keras .h5 files, or PyTorch .th files).

Example model optimizers:

- TFLiteCompiler - wraps TensorFlow Lite compilation,
- TVMCompiler - wraps TVM compilation.

class kenning.core.optimizer.**Optimizer**(dataset: Dataset | *None*, compiled_model_path: *Path* | ResourceURI, location: *'host'* | *'target'* = `'host'`, model_wrapper: ModelWrapper | *None* = **None**)

Bases: `ArgumentsHandler`, `ABC`

Compiles the given model to a different format or runtime.

abstract compile(input_model_path: *Path* | ResourceURI, io_spec: *dict[str, list[dict]]* | *None* = **None**)

Compiles the given model to a target format.

The function compiles the model and saves it to the output file.

The model can be compiled to a binary, a different framework or a different programming language.

If *io_spec* is passed, then the function uses it during the compilation, otherwise *load_io_specification* is used to fetch the specification saved in *input_model_path* + *.json*.

The compiled model is saved to compiled_model_path and the specification is saved to compiled_model_path + .json

Parameters

input_model_path : PathOrURI
Path to the input model.

io_spec : Optional[Dict[str, List[Dict]]]
Dictionary that has *input* and *output* keys that contain list of dictionaries mapping (property name) -> (property value) for the layers.

consult_model_type(previous_block: ModelWrapper | Optimizer | *type[ModelWrapper]* | *type[Optimizer]*, force_onnx: *bool* = **False**) → str

Finds output format of the previous block in the chain matching with an input format of the current block.

Parameters

previous_block : Union["ModelWrapper", "Optimizer", Type["ModelWrapper"], Type["Optimizer"]]
Previous block in the optimization chain.

force_onnx : bool
Forces ONNX format.

Returns
Matching format.

Return type
str

Raises
ValueError – Raised if there is no matching format.

classmethod from_argparse(dataset: Dataset | *None*, args: *Namespace*) → *Optimizer*

Constructor wrapper that takes the parameters from argparse args.

Parameters

dataset : Optional[*Dataset*]
The dataset object that is optionally used for optimization.

args : Namespace
Arguments from ArgumentParser object.

Returns
Object of class Optimizer.

Return type
Optimizer

classmethod from_json(json_dict: *dict*, dataset: Dataset | *None* = **None**,
model_wrapper: ModelWrapper | *None* = **None**) → *Optimizer*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the `arguments_structure` defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

dataset : Optional[*Dataset*]
The dataset object that is optionally used for optimization.

model_wrapper : Optional[*ModelWrapper*]
ModelWrapper for the optimized model (optional).

Returns
Object of class Optimizer.

Return type
Optimizer

abstract get_framework_and_version() → tuple[str, str]

Returns name of the framework and its version in a form of a tuple.

Returns
Framework name and version.

Return type
Tuple[str, str]

classmethod get_input_formats() → list[str]

Returns list of names of possible input formats.

Returns
Names of possible input formats.

Return type
List[str]

get_input_type(model_path: *Path* | ResourceURI) → str

Return input model type. If input type is set to "any", then it is derived from model file extension.

Parameters

model_path : PathOrURI
Path to the input model.

Returns
Input model type.

Return type
str

Raises
Exception – Raised if input model type cannot be determined.

get_optimized_model_size() → float

Returns the optimized model size.

By default, the size of file with optimized model is returned.

Returns
The size of the optimized model in KB.

Return type
float

Raises
OptimizedModelSizeError – If model size cannot be retrieved.

classmethod get_output_formats() → list[str]

Returns list of names of possible output formats.

Returns
List of possible output formats.

Return type
List[str]

static get_spec_path(model_path: *Path* | ResourceURI) → Path | *ResourceURI*

Returns input/output specification path for the model saved in *model_path*. It concatenates *model_path* and *.json*.

Parameters

model_path : PathOrURI
Path where the model is saved.

Returns
Path to the input/output specification of a given model.

Return type
PathOrURI

init()

Initializes optimizer, should be called before compilation.

load_io_specification(model_path: *Path* | ResourceURI) → dict[str, list[dict]] | None

Returns saved input and output specification of a model saved in *model_path* if there is one. Otherwise returns None.

Parameters

model_path : PathOrURI
Path to the model which specification the function should read.

Returns
Specification of a model saved in *model_path* if there is one. None otherwise.

Return type
Optional[Dict[str, List[Dict]]]

read_platform(platform: Platform)

Reads Platform data to configure optimization/compilation.

Platform-based entities come with lots of information on hardware architecture that can be used by the Optimizer class.

By default no data is read.

It is important to take into account that different Platform-based classes come with a different sets of attributes.

Parameters

platform : *Platform*
object with platform details

save_io_specification(input_model_path: *Path* | ResourceURI, io_spec: *dict[str, list[dict]]* | *None* = **None**)

Internal function that saves input/output model specification which is used during both inference and compilation. If *io_spec* is None, the function uses specification of an input model stored in *input_model_path* + *.json*. If there is no specification stored in this path the function does not do anything.

The input/output specification is a list of dictionaries mapping properties names to their values. Legal properties names are *dtype*, *prequantized_dtype*, *shape*, *name*, *scale*, *zero_point*.

The order of the layers has to be preserved.

Parameters

input_model_path : PathOrURI
Path to the input model.

io_spec : Optional[Dict[str, List[Dict]]]
Specification of the input/ouput layers.

set_compiled_model_path(compiled_model_path: *Path*)

Sets path for compiled model.

compiled_model_path
[PathOrURI] Path to be set.

set_input_type(inputtype: *str*)

Sets input type of the model for the compiler.

inputtype
[str] Path to be set.

15.7 Runtime

The `kenning.core.runtime.Runtime` class provides interfaces for methods for running compiled models locally or remotely on a target device. Runtimes are usually compiler-specific (frameworks for deep learning compilers provide runtime libraries to run compiled models on particular hardware).

The client (host) side of the `Runtime` class utilizes the methods from *Dataset*, *ModelWrapper* and *Protocol* classes to run inference on a target device. The server (target) side of the `Runtime` class requires method implementation for:

- loading a model delivered by the client,
- preparing inputs delivered by the client,
- running inference,
- preparing outputs for delivery to the client,
- (optionally) sending inference statistics.

Runtime examples:

- TFLiteRuntime for models compiled with TensorFlow Lite,
- TVMRuntime for models compiled with TVM.

class kenning.core.runtime.**Runtime**(disable_performance_measurements: *bool* = **False**)

Bases: `ArgumentsHandler`, `ABC`

Runtime object provides an API for testing inference on target devices.

abstract extract_output() → list[Any]

Extracts and postprocesses the output of the model.

Returns
Postprocessed and reordered outputs of the model.

Return type
List[Any]

classmethod from_argparse(args: *Namespace*) → *Runtime*

Constructor wrapper that takes the parameters from argparse args.

Parameters

args : Namespace
Arguments from ArgumentParser object.

Returns
Object of class Runtime.

Return type
Runtime

classmethod from_json(json_dict: *dict*) → *Runtime*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the `arguments_structure` defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

Returns
Object of class Runtime.

Return type
Runtime

static get_available_ram(platform: Platform) → float | None

Gets size of the RAM (or other type of memory where the model will be stored) for the given platform.

Parameters

platform : *Platform*
The platform representation.

Returns
The size of RAM.

Return type
Optional[float]

classmethod get_input_formats() → list[str]

Returns list of names of possible input formats names.

Returns
List of possible input format names.

Return type
List[str]

get_io_spec_path(model_path: *Path* | ResourceURI) → Path

Gets path to a input/output specification file which is *model_path* and *.json* concatenated.

Parameters

model_path : PathOrURI
URI to the compiled model.

Returns
Returns path to the specification.

Return type
Path

get_time() → float

Gets the current timestamp.

Returns
Current timestamp.

Return type
float

infer(X: *list[ndarray]*, model_wrapper: ModelWrapper, postprocess: *bool* = **True**) → list[Any]

Runs inference on single batch locally using a given runtime.

Parameters

X : List[np.ndarray]
Batch of data provided for inference.

model_wrapper : *ModelWrapper*
Model that is executed on target hardware.

postprocess : bool
Indicates if model output should be postprocessed.

Returns
Obtained values.

Return type
List[Any]

inference_session_end()

Calling this function indicates that the inference session has ended.

This method should be called once all the inference data is sent to the server by the client.

This will stop performance tracking.

inference_session_start()

Calling this function indicates that the client is connected.

This method should be called once the client has connected to a server.

This will enable performance tracking.

abstract load_input(input_data: *list[Any]*) → bool

Loads and converts delivered data to the accelerator for inference.

This method is called when the input is received from the client. It is supposed to prepare input before running inference.

Parameters

input_data : List[Any]
Input data in bytes delivered by the client, preprocessed.

Returns
True if succeeded.

Return type
bool

Raises
ModelNotLoadedError : – Raised if model is not loaded.

load_input_from_bytes(input_data: *bytes*) → bool

The method accepts *input_data* in bytes and loads it according to the input specification.

It creates *np.ndarray* for every input layer using the metadata in *self.input_spec* and quantizes the data if needed.

Some compilers can change the order of the layers. If that's the case the method also reorders the layers to match the specification of the model.

Parameters

input_data : bytes
Input data in bytes delivered by the client.

Returns
Output of the *load_input* method indicating if the operation succeeded.

Return type
bool

Raises
AttributeError – Raised if output specification is not loaded.

prepare_io_specification(input_data: *bytes* | *None*) → bool

Receives the io_specification from the client in bytes and saves it for later use.

`input_data` stores the io_specification representation in bytes. If `input_data` is None, the io_specification is extracted from another source (i.e. from existing file). If it can not be found in this path, io_specification is not loaded.

When no specification file is found, the function returns True as some Runtimes may not need io_specification to run the inference.

Parameters

input_data : Optional[bytes]
The io_specification` or None, if it should be loaded from another source.

Returns
True if succeeded.

Return type
bool

prepare_local() → bool

Runs initialization for the local inference.

Returns
True if initialized successfully.

Return type
bool

abstract prepare_model(input_data: *bytes* | *None*) → bool

Receives the model to infer from the client in bytes.

The method should load bytes with the model, optionally save to file and allocate the model on target device for inference.

`input_data` stores the model representation in bytes. If `input_data` is None, the model is extracted from another source (i.e. from existing file or directory).

Parameters

input_data : Optional[bytes]
Model data or None, if the model should be loaded from another source.

Returns
True if succeeded.

Return type
bool

preprocess_model_to_upload(path: *Path* | ResourceURI) → Path | *ResourceURI*

The method preprocesses the model to be uploaded to the client and returns a new path to it.

The method is used to prepare the model to be sent to the client. It can be used to change the model representation, for example, to compress it.

Parameters

path : PathOrURI
Path to the model to preprocess.

Returns
Path to the preprocessed model.

Return type
PathOrURI

read_io_specification(io_spec: *dict*)

Saves input/output specification so that it can be used during the inference.

input_spec and *output_spec* are lists, where every element is a dictionary mapping (property name) -> (property value) for the layers.

The standard property names are: *name*, *dtype* and *shape*.

If the model is quantized it also has *scale*, *zero_point* and *prequantized_dtype* properties.

If the layers of the model are reorder it also has *order* property.

Parameters

io_spec : Dict
Specification of the input/output layers.

Raises
IOSpecWrongFormat – Raised if preprocessed input data has more than one available shape.

abstract run()

Runs inference on prepared input.

The input should be introduced in runtime's model representation, or it should be delivered using a variable that was assigned in load_input method.

Raises
ModelNotLoadedError : – Raised if model is not loaded.

upload_output(input_data: *bytes*) → bytes

Returns the output to the client, in bytes.

The method converts the direct output from the model to bytes and returns them.

The wrapper later sends the data to the client.

Parameters

input_data : bytes
Not used here.

Returns
Data to send to the client.

Return type
bytes

upload_stats(input_data: *bytes*) → bytes

Returns statistics of inference passes to the client.

Default implementation converts collected metrics in MeasurementsCollector to JSON format and returns them for sending.

Parameters

input_data : bytes
Not used here.

Returns
Statistics to be sent to the client.

Return type
bytes

15.8 Platform

The `kenning.core.platform.Platform` classes represents the targeted environment providing additional context for optimization, compilation and evaluation of the model.

Platform examples:

- LocalPlatform
- BareMetalPlatform
- ZephyrPlatform

class kenning.core.platform.**Platform**(name: *str* | *None* = **None**, platforms_definitions: *list[*ResourceURI*]* | *None* = **None**)

Bases: `ArgumentsHandler`, `ABC`

Wraps the platform that is being evaluated. This class provides methods to handle tested platform. The platform can be the device that Kenning is run on, a board running Kenning Zephyr Runtime or bare-metal IREE runtime or any remote or device running Kenning inference server.

deinit(measurements: Measurements | *None* = **None**)

Deinitializes platform.

Parameters

measurements : Optional[*Measurements*]
Measurements to which platform metrics can be added to.

get_default_protocol() → *Protocol*

Returns default protocol for given platform.

Returns
Default protocol for given platform.

Return type
Protocol

get_time() → float

Retrieves elapsed time from platform.

Returns
Elapsed time.

Return type
float

inference_step_callback()

Callback that is run every inference step.

init()

Initializes the platform.

read_data_from_platforms_yaml()

Retrieves platform data from specified platform definition files.

15.9 AutoML

The AutoML API is combined from two different elements - *AutoML model* and *AutoML flow mechanism*.

AutoML examples:

- AutoPyTorchModel and AutoPyTorchML

15.9.1 AutoML model

The `kenning.core.automl.AutoMLModel` represents a model that can be used for AutoML flow, and defines additional methods for registering and managing parameters. Moreover, the `arguments_structure` was extended with custom AutoML options, described in *Defining arguments for core classes*.

class kenning.core.automl.**AutoMLModel**

Bases: `ArgumentsHandler`, `ABC`

Base class representing AutoML-compatible model.

Should be used together with kenning.core.model.ModelWrapper class.

abstract extract_model(network: *Any*) → Any

Extracts model from AutoML object.

Parameters

network : Any
AutoML object with optimized model.

Returns
Extracted model

Return type
Any

classmethod form_automl_schema() → dict

Gathers AutoML schema based on *arguments_structure* of class and all parent.

Returns
AutoML schema for the class.

Return type
Dict

Raises
AutoMLInvalidSchemaError – If schema does not contain required fields.

static update_automl_defaults(arg_structure: *dict[str, dict]*, name: *str*)

Updates the default value of the AutoML parameter to make sure they fit into ranges.

Parameters

arg_structure : Dict[str, Dict]
The combined arguments structure of current class and its parents.

name : str
The name of parameter, should match with the *arguments_structure*.

classmethod update_automl_range(name: *str*, conf: *dict[str, tuple]*)

Updates the ranges of AutoML parameters.

Parameters

name : str
The name of parameter, should match with the *arguments_structure*.

conf : Dict[str, Tuple]
The dictionary with new parameters configuration.

Raises
AutoMLInvalidArgumentsError – If parameter or its configuration does not exist.

15.9.2 AutoML flow mechanism

The `kenning.core.automl.AutoML` covers a standard AutoML flow: framework preparation, model searching and extraction to proper Kenning format.

class kenning.core.automl.**AutoML**(dataset: Dataset, platform: Platform, output_directory: *Path*, optimizers: *list[Optimizer]* = `[]`, runtime: Runtime | *None* = **None**, use_models: *list[str | dict[str, tuple]]* = `[]`, time_limit: *float* = `5.0`, optimize_metric: *str* = `'accuracy'`, n_best_models: *int* = `5`, application_size: *float | None* = **None**, skip_model_size_check: *bool* = **False**, callback_max_samples: *int* = `30`, seed: *int* = `1234`)

Bases: `ArgumentsHandler`, `ABC`

Base class describing AutoML flow: * preparing AutoML framework, * searching the best models, * generating Kenning configs based on found solutions.

classmethod from_argparse(dataset: Dataset | *None*, platform: Platform | *None*, args: *Namespace*) → *AutoML*

Constructor wrapper that takes the parameters from argparse args.

Parameters

dataset : Optional[*Dataset*]
The dataset object that is optionally used for optimization.

platform : Optional[*Platform*]
The platform on which generated models will be evaluated.

args : Namespace
Arguments from ArgumentParser object.

Returns
Object of class AutoML.

Return type
AutoML

classmethod from_json(json_dict: *dict*, dataset: Dataset | *None* = **None**, platform: Platform | *None* = **None**, optimizers: *list[*Optimizer*]* | *None* = **None**) → *AutoML*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the *arguments_structure* defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

dataset : Optional[*Dataset*]
The dataset object that is optionally used for optimization.

platform : Optional[*Platform*]
The platform on which generated models will be evaluated.

optimizers : Optional[List[*Optimizer*]]
The optional list with optimizers.

Returns
Object of class AutoML.

Return type
AutoML

abstract get_best_configs() → Iterable[dict]

Extracts the best models and returns Kenning configuration for them.

Yields
Dict – Configuration for found models (from the best one).

abstract get_statistics() → dict[str, int | float]

Returns statistic of the AutoML flow, like number of successful or crashed runs.

The created dictionary has to contain “general_info” - mapping of statisctics descriptions and values.

Optional fields: * “trained_model_metrics” - mapping of models to dictionaries

with datasets and metrics of trained models,

- “training_data” - mapping of models to losses from different parts of training, containing dictionaries with timestamps and loss values (averaged from batch or whole epoch). Possible parts of training: “training”, “training_epoch”, “validation”, “validation_epoch”, “test” and “test_epoch”,
- “training_start_time” - mapping of models to a list of times, marking the beginning of trainings,
- “model_params” - mapping of models to dictionaries with parameters descriptions and values.

Returns
Dictionary with AutoML statistics, keys should describe given statistic.

Return type
Dict[str, Union[int, float]]

abstract prepare_framework()
Prepares AutoML framework.

abstract search()
Runs AutoML search.

15.10 Protocol

The `kenning.core.protocol.Protocol` class conducts communication between the client (host) and the server (target).

The Protocol class requires method implementation for:

- initializing the server and the client (communication-wise),
- waiting for the incoming data,
- data sending,
- data receiving,
- uploading model inputs to the server,
- uploading the model to the server,
- requesting inference on target,
- downloading outputs from the server,
- (optionally) downloading the statistics from the server (e.g. performance speed, CPU/GPU utilization, power consumption),
- success or failure notifications from the server,
- message parsing.

Based on the above-mentioned methods, the `kenning.core.runtime.Runtime` connects the host with the target.

Protocol examples:

- NetworkProtocol - implements a TCP-based communication between the host and the client.

15.10.1 Protocol specification

The communication protocol is message-based. Possible messages are:

- `OK` messages - indicate success, and may come with additional information,
- `ERROR` messages - indicate failure,
- `DATA` messages - provide input data for inference,
- `MODEL` messages - provide model to load for inference,

- PROCESS messages - request processing inputs delivered in DATA message,
- OUTPUT messages - request processing results,
- STATS messages - request statistics from the target device.

The message types and enclosed data are encoded in a format implemented in the kenning.core.protocol.Protocol-based class.

Communication during an inference benchmark session goes as follows:

- The client (host) connects to the server (target),
- The client sends a MODEL request along with the compiled model,
- The server loads the model from request, prepares everything to run the model and sends an OK response,
- After receiving the OK response from the server, the client starts reading input samples from the dataset, preprocesses the inputs, and sends a DATA request with the preprocessed input,
- Upon receiving the DATA request, the server stores the input for inference, and sends an OK message,
- Upon receiving confirmation, the client sends a PROCESS request,
- Just after receiving the PROCESS request, the server should send an OK message to confirm start of inference, and just after the inference is finished, the server should send another OK message to confirm that the inference has finished,
- After receiving the first OK message, the client starts measuring inference time until the second OK response is received,
- The client sends an OUTPUT request in order to receive the outputs from the server,
- The server sends an OK message along with the output data,
- The client parses the output and evaluates model performance,
- The client sends a STATS request to obtain additional statistics (inference time, CPU/GPU/Memory utilization) from the server,
- If the server provides any statistics, it sends an OK message with the data,
- The same process applies to the rest of input samples.

The way the message type is determined and the data between the server and the client is sent depends on the implementation of the kenning.core.protocol.Protocol class. The implementation of running inference on the given target is contained within the kenning.core.runtime.Runtime class.

15.10.2 Protocol

`kenning.core.protocol.Protocol`-based classes implement the *Protocol specification* in a given means of transport, e.g. TCP connection or UART. It requires method implementation for:

- server (target hardware) and client (compiling host) initialization,
- sending and receiving data,
- connecting and disconnecting,
- model upload (host) and download (target hardware),
- message parsing and creation.

class kenning.core.protocol.**Protocol**(timeout: *int* = -1)

Bases: `ArgumentsHandler`, `ABC`

The interface for the communication protocol with the target devices.

The target device acts as a server in the communication.

The machine that runs the benchmark and collects the results is the client for the target device.

The inheriting classes for this class implement at least the client-side of the communication with the target device.

abstract disconnect()

Ends connection with the other side.

download_output() → tuple[bool, bytes | None]

Downloads the outputs from the target device.

Requests and downloads the latest inference output from the target device for quality measurements.

Returns

Tuple with download status (True if successful) and downloaded data.

Return type

Tuple[bool, Optional[bytes]]

download_statistics(final: *bool* = **False**) → *Measurements*

Downloads inference statistics from the target device.

By default no statistics are gathered.

Parameters

final : bool

If the inference is finished

Returns

Inference statistics on target device.

Return type

Measurements

classmethod from_argparse(args: *Namespace*) → *Protocol*

Constructor wrapper that takes the parameters from argparse args.

Parameters

args : Namespace
Arguments from Protocol object.

Returns
Object of class Protocol.

Return type
Protocol

classmethod from_json(json_dict: *dict*) → *Protocol*

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the `arguments_structure` defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

Returns
Object of class Protocol.

Return type
Protocol

abstract gather_data(timeout: *float* | *None* = **None**) → tuple[ServerStatus, Any | None]

Gathers data from the client.

This method should be called by receive_message in order to get data from the client.

Parameters

timeout : Optional[float]
Receive timeout in seconds. If timeout > 0, this specifies the maximum wait time, in seconds. If timeout <= 0, the call won't block, and will report the currently ready file objects. If timeout is None, the call will block until a monitored file object becomes ready.

Returns
Receive status along with received data.

Return type
Tuple[ServerStatus, Optional[Any]]

abstract initialize_client() → bool

Initializes client side of the protocol.

The client side is supposed to run on host testing the target hardware.

The parameters for the client should be provided in the constructor.

Returns
True if succeeded.

Return type
bool

abstract initialize_server() → bool

Initializes server side of the protocol.

The server side is supposed to run on target hardware.

The parameters for the server should be provided in the constructor.

Returns
True if succeeded.

Return type
bool

receive_confirmation() → tuple[bool, bytes | None]

Waits until the OK message is received.

Method waits for the OK message from the other side of connection.

Returns
True if OK received and attached message data, False otherwise.

Return type
Tuple[bool, Optional[bytes]]

abstract receive_data(connection: *Any*, mask: *int*) → tuple[ServerStatus, Any | None]

Receives data from the target device.

Parameters

connection : Any
Connection used to read data.

mask : int
Selector mask from the event.

Returns
Status of receive and optionally data that was received.

Return type
Tuple[ServerStatus, Optional[Any]]

abstract receive_message(timeout: *float | None* = **None**) → tuple[ServerStatus, Message]

Waits for incoming data from the other side of connection.

This method should wait for the input data to arrive and return the appropriate status code along with received data.

Parameters

timeout : Optional[float]
Receive timeout in seconds. If timeout > 0, this specifies the maximum wait time, in seconds. If timeout <= 0, the call won't block, and will report the currently ready file objects. If timeout is None, the call will block until a monitored file object becomes ready.

Returns
Tuple containing server status and received message. The status is NOTHING if message is incomplete and DATA_READY if it is complete.

Return type
Tuple[ServerStatus, Message]

request_failure() → bool

Sends ERROR message back to the client if it failed to handle request.

Returns
True if sent successfully.

Return type
bool

request_optimization(model_path: ~pathlib.Path, get_time_func: ~typing.Callable[[], float] = <built-in function perf_counter>) → tuple[bool, bytes | None]

Request optimization of model.

Parameters

model_path : Path
Path to the model for optimization.

get_time_func : Callable[[], float]
Function that returns current timestamp.

Returns
First element is equal to True if optimization finished successfully and the second element contains compiled model.

Return type
Tuple[bool, Optional[bytes]]

request_processing(get_time_func: ~typing.Callable[[], float] = <built-in function perf_counter>) → bool

Requests processing of input data and waits for acknowledgement.

This method triggers inference on target device and waits until the end of inference on target device is reached.

This method measures processing time on the target device from the level of the host.

Target may send its own measurements in the statistics.

Parameters

get_time_func : Callable[[], float]
Function that returns current timestamp.

Returns
True if inference finished successfully.

Return type
bool

request_success(data: *bytes* | *None* = b'') → bool

Sends OK message back to the client once the request is finished.

Parameters

data : Optional[bytes]
Optional data upon success, if any.

Returns
True if sent successfully.

Return type
bool

abstract send_data(data: *Any*) → bool

Sends data to the target device.

Data can be model to use, input to process, additional configuration.

Parameters

data : Any
Data to send.

Returns
True if successful.

Return type
bool

abstract send_message(message: *Message*) → bool

Sends message to the target device.

Parameters

message : Message
Message to be sent.

Returns
True if succeeded.

Return type
bool

upload_input(data: *bytes*) → bool

Uploads input to the target device and waits for acknowledgement.

This method should wait until the target device confirms the data is delivered and preprocessed for inference.

Parameters

data : bytes
Input data for inference.

Returns
True if ready for inference.

Return type
bool

upload_io_specification(path: *Path*) → bool

Uploads input/output specification to the target device.

This method takes the specification in a json format from the given Path and sends it to the target device.

This method should receive the status of uploading the data to the target.

Parameters

path : Path
Path to the json file.

Returns
True if data upload finished successfully.

Return type
bool

upload_model(path: *Path*) → bool

Uploads the model to the target device.

This method takes the model from given Path and sends it to the target device.

This method should receive the status of uploading the model from the target.

Parameters

path : Path
Path to the model.

Returns
True if model upload finished successfully.

Return type
bool

upload_optimizers(optimizers_cfg: *dict[str, Any]*) → bool

Upload optimizers config to the target device.

Parameters

optimizers_cfg : Dict[str, Any]
Config JSON of optimizers.

Returns
True if data upload finished successfully.

Return type
bool

upload_runtime(path: *Path*) → bool

Uploads the runtime to the target device.

This method takes the binary from given Path and sends it to the target device.

This method should receive the status of runtime loading from the target.

Parameters

path : Path
Path to the runtime binary.

Returns
True if runtime upload finished successfully.

Return type
bool

15.11 DataConverter

`kennning.core.dataconverter.DataConverter` - based classes are responsible for:

- converting data to the format expected by surrounding block,
- converting data from the surrounding block format to one previous block excepts.

The `DataConverter` objects are used by PipelineRunner during inference.

The available implementations of dataconverter are included in the `kenning.dataconverters` submodule. Example implementations:

- ModelWrapperDataConverter implement conversions by utilizing the *ModelWrapper* classes,
- ROS2SegmentationDataConverter performs conversions by utilizing ROS2 action for segmentation problems from kenning_computer_vision_msgs repository.

class kenning.core.dataconverter.**DataConverter**

Bases: `ArgumentsHandler`, `ABC`

Performs conversion of data between two surrounding blocks.

This class provides an API used by Runtimes during inference execution.

Each DataConverter should implement methods for:

- converting data from dataset to the format used by the surrounding block.
- converting data from format used by the surrounding block to the

inference output.

abstract to_next_block(data: *Any*) → Any

Converts data to the format used by the surrounding block.

Parameters

data : Any
Data to be converted.

Returns
Converted data.

Return type
Any

abstract to_previous_block(data: *Any*) → Any

Converts data from the format used by the surrounding block to one previous block expects.

Parameters

data : Any
Data to be converted.

Returns
Converted data.

Return type
Any

15.12 Measurements

The `kenning.core.measurements` module contains `Measurements` and `MeasurementsCollector` classes for collecting performance and quality metrics. `Measurements` is a dict-like object that provides various methods for adding performance metrics, adding values for time series, and updating existing values.

The dictionary held by `Measurements` requires serializable data, since most scripts save performance results in JSON format for later report generation.

Module containing decorators for benchmark data gathering.

class kenning.core.measurements.**Measurements**

Stores benchmark measurements for later processing.

This is a dict-like object that wraps all processing results for later report generation.

The dictionary in Measurements has measurement type as a key, and list of values for given measurement type.

There can be other values assigned to a given measurement type than list, but it requires explicit initialization.

data

Dictionary storing lists of values.

Type
dict

accumulate(measurementtype: str, valuetoadd: ~typing.Any, initvaluefunc: ~typing.Callable[[], ~typing.Any] = <function Measurements.<lambda>>)

Adds given value to a measurement.

This function adds given value (it can be integer, float, numpy array, or any type that implements iadd operator).

If it is the first assignment to a given measurement type, the first list element is initialized with the `initvaluefunc` (function returns the initial value).

Parameters

measurementtype : str
The name of the measurement.

valuetoadd : Any
New value to add to the measurement.

initvaluefunc : Callable[[], Any]
The initial value of the measurement, default 0.

add_measurement(measurementtype: str, value: ~typing.Any, initialvaluefunc: ~typing.Callable[[], ~typing.Any] = <function Measurements.<lambda>>)

Add new value to a given measurement type.

Parameters

measurementtype : str
The measurement type to be updated.

value : Any
The value to add.

initialvaluefunc : Callable[[], Any]
The initial value for the measurement.

clear()

Clears measurement data.

copy()

Makes copy of measurements data.

get_values(measurementtype: *str*) → list

Returns list of values for a given measurement type.

Parameters

measurementtype : str
The name of the measurement type.

Returns
List of values for a given measurement type.

Return type
List

initialize_measurement(measurement_type: *str*, value: *Any*)

Sets the initial value for a given measurement type.

By default, the initial values for every measurement are empty lists. Lists are meant to collect time series data and other probed measurements for further analysis.

In case the data is collected in a different container, it should be configured explicitly.

Parameters

measurement_type : str
The type (name) of the measurement.

value : Any
The initial value for the measurement type.

update_measurements(other: *dict* | Measurements)

Adds measurements of types given in the other object.

It requires another Measurements object, or a dictionary that has string keys and values that are lists of values. The lists from the other object are appended to the lists in this object.

Parameters

other : Union[Dict, 'Measurements']
A dictionary or another Measurements object that contains lists in every entry.

class kenning.core.measurements.**MeasurementsCollector**

It is a 'static' class collecting measurements from various sources.

classmethod clear()

Clears measurement data.

classmethod save_measurements(resultpath: *Path*)

Saves measurements to JSON file.

Parameters

resultpath : Path
Path to the saved JSON file.

classmethod set_unoptimized(optimized_measurementspath: *Path*, unoptimized_measurementspath: *Path*, remove_unoptimized_measurementsfile: *bool* = **True**)

Copies unoptimized model measurements to *UNOPTIMIZED* field of the optimized model measurements.

Parameters

optimized_measurementspath : Path
Path to the optimized model measurements.

unoptimized_measurementspath : Path
Path to the unoptimized model measurements.

remove_unoptimized_measurementsfile : bool
Determines whether the unoptimized model measurements should be deleted.

class kenning.core.measurements.**SystemStatsCollector**(prefix: *str*, step: *float* = 0.1)

It is a separate thread used for collecting system statistics.

It collects:

- CPU utilization,
- RAM utilization,
- GPU utilization,
- GPU Memory utilization.

It can be executed in parallel to another function to check its utilization of resources.

get_measurements() → *Measurements*

Returns measurements from the thread.

Collected measurements names are prefixed by the prefix given in the constructor.

The list of measurements:

- <*prefix*>*_cpus_percent*: gives per-core CPU utilization (%),
- <*prefix*>*_mem_percent*: gives overall memory usage (%),
- <*prefix*>*_gpu_utilization*: gives overall GPU utilization (%),
- <*prefix*>*_gpu_mem_utilization*: gives overall memory utilization (%),
- <*prefix*>*_timestamp*: gives the timestamp of above measurements (ns).

Returns
Measurements object.

Return type
Measurements

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

kenning.core.measurements.**systemstatsmeasurements**(measurementname: *str*, step: *float* = 0.5) → Callable

Decorator for measuring memory usage of the function.

Check SystemStatsCollector.get_measurements for list of delivered measurements.

Parameters

measurementname : str
The name of the measurement type.

step : float
The step for the measurements, in seconds.

Returns
Decorated function.

Return type
Callable

kenning.core.measurements.**tagmeasurements**(tagname: *str*) → Callable

Decorator for adding tags for measurements and saving their timestamps.

Parameters

tagname : str
The name of tag.

Returns
Decorated function.

Return type
Callable

kenning.core.measurements.**timemeasurements**(measurementname: str, get_time_func: ~typing.Callable[[], float] = <built-in function perf_counter>) → Callable

Decorator for measuring time of the function.

The duration is given in nanoseconds.

Parameters

measurementname : str
The name of the measurement type.

get_time_func : Callable[[], float]
Function that returns current timestamp.

Returns
Decorated function.

Return type
Callable

15.13 ONNXConversion

The ONNXConversion object contains methods for model conversion in various frameworks to ONNX and vice versa. It also provides methods for testing the conversion process empirically on a list of deep learning models implemented in the tested frameworks.

class kenning.core.onnxconversion.**ONNXConversion**(framework: *str*, version: *str*)

Bases: ABC

Creates ONNX conversion support matrix for given framework and models.

add_entry(name: *str*, modelgenerator: *Callable*, **kwargs: *dict[str, Any]*)

Adds new model for verification.

Parameters

name : str
Full name of the model, should match the name of the same models in other framework's implementations.

modelgenerator : Callable
Function that generates the model for ONNX conversion in a given framework. The callable should accept no arguments.

****kwargs : Dict[str, Any]**
Additional arguments that are passed to ModelEntry object as parameters.

check_conversions(modelsdir: *Path*) → list[Support]

Runs ONNX conversion for every model entry in the list of models.

Parameters

modelsdir : Path
Path to the directory where the intermediate models will be saved.

Returns
List with Support tuples describing support status.

Return type
List[Support]

abstract onnx_export(modelentry: *ModelEntry*, exportpath: *Path*) → SupportStatus

Virtual function for exporting the model to ONNX in a given framework.

This method needs to be implemented for a given framework in inheriting class.

Parameters

modelentry : ModelEntry
ModelEntry object.

exportpath : Path
Path to the output ONNX file.

Returns
The support status of exporting given model to ONNX.

Return type
SupportStatus

abstract onnx_import(modelentry: *ModelEntry*, importpath: *Path*) → SupportStatus

Virtual function for importing ONNX model to a given framework.

This method needs to be implemented for a given framework in inheriting class.

Parameters

modelentry : ModelEntry
ModelEntry object.

importpath : Path
Path to the input ONNX file.

Returns
The support status of importing given model from ONNX.

Return type
SupportStatus

abstract prepare()

Virtual function for preparing the ONNX conversion test.

This method should add model entries using add_entry methods.

It is later called in the constructor to prepare the list of models to test.

15.14 DataProvider

The `DataProvider` classes are used during deployment to provide data for inference. They can provide data from such sources as a camera, video files, microphone data or a TCP connection.

The available `DataProvider` implementations are included in the `kenning.dataproviders` submodule. Example implementations:

- CameraDataProvider for capturing frames from camera.

class kenning.core.dataprovider.**DataProvider**(inputs_sources: *dict[str, tuple[int, str]]* = {}, inputs_specs: *dict[str, dict]* = {}, outputs: *dict[str, str]* = {})

Bases: Runner, ABC

A block that introduces data to Kenning flow.

abstract detach_from_source()

Detaches from the source during shutdown.

abstract fetch_input() → Any

Gets the sample from device.

Returns
Data to be processed by the model.

Return type
Any

prepare()

Prepares the source for data gathering depending on the source type.

This will for example initialize the camera and set the self.device to it.

preprocess_input(data: *Any*) → Any

Performs provider-specific preprocessing of inputs.

Parameters

data : Any
The data to be preprocessed.

Returns
Preprocessed data.

Return type
Any

15.15 OutputCollector

The `OutputCollector` classes are used during deployment for inference results receiving and processing. They can display the results, send them, or store them in a file.

The available output collector implementations are included in the `kenning.outputcollectors` submodule. Example implementations:

- DetectionVisualizer for visualizing detection model outputs,
- BaseRealTimeVisualizer base class for real time visualizers:
  - RealTimeDetectionVisualizer for visualizing detection model outputs,
  - RealTimeSegmentationVisualizer for visualizing segmentation model outputs,
  - RealTimeClassificationVisualizer for visualizing classification model outputs.

class kenning.core.outputcollector.**OutputCollector**(inputs_sources: *dict[str, tuple[int, str]]* = {}, inputs_specs: *dict[str, dict]* = {}, outputs: *dict[str, str]* = {})

Bases: Runner, ABC

Collects outputs from models running in the Kenning flow.

It performs final processing of data running in the Kenning flow. It can be used i.e. to display predictions, save them to file or send to other application.

abstract detach_from_output()

Detaches from the output during shutdown.

abstract process_output(input_data: *Any*, output_data: *Any*)

Returns the inferred data back to the specific place/device/connection.

Eg. it can save a video file with bounding boxes on objects or stream it via a TCP connection, or just show it on screen.

Parameters

input_data : Any
Data collected from Datacollector that was processed by the model.

output_data : Any
Data returned from the model.

abstract should_close() → bool

Checks if a specific exit condition was reached.

This allows the OutputCollector to close gracefully if an exit condition was reached, eg. when a key was pressed.

Returns
True if exit condition was reached to break the loop.

Return type
bool

15.16 ArgumentsHandler

The `ArgumentsHandler` class is responsible for concatenating `arguments_structure` and creating parsers for command line and JSON config arguments.

In order to make some class being able to be instantiated from command line arguments or JSON config it is required to inherit from this class or its child class and implement `from_argparse` or `from_json` methods as described in *Defining arguments for core classes*.

class kenning.utils.args_manager.**ArgumentsHandler**

Bases: ABC

Class responsible for creating parsers for arguments from command line or json configs.

The child class should define its own *arguments_structure* and from_argparse/from_json methods so that it could be instantiated from command line arguments or json config.

classmethod form_argparse(args: *Namespace*, override_only: *bool* = **False**) → tuple[ArgumentParser, _ArgumentGroup | None]

Creates argparse parser based on *arguments_structure* of class and its all parent classes.

Parameters

args : argparse.Namespace
Arguments from ArgumentParser object.

override_only : bool
True if only parameters marked as *overridable* should be parsed.

Returns
Tuple with the argument parser object that can act as parent for program's argument parser, and the corresponding arguments' group pointer.

Return type
Tuple[argparse.ArgumentParser, Optional[argparse._ArgumentGroup]]

classmethod form_parameterschema() → dict

Creates parameter schema based on *arguments_structure* of class and its all parent classes.

Returns
Parameter schema for the class.

Return type
Dict

classmethod from_argparse(args: *Namespace*, **kwargs: *dict[str, Any]*) → Any

Constructor wrapper that takes the parameters from argparse args.

Parameters

args : argparse.Namespace
Arguments from ArgumentParser object.

****kwargs : Dict[str, Any]**
Additional class-dependent arguments.

Returns
Instance created from provided args.

Return type
Any

classmethod from_json(json_dict: *dict*, **kwargs: *dict[str, Any]*) → Any

Constructor wrapper that takes the parameters from json dict.

This function checks if the given dictionary is valid according to the `arguments_structure` defined. If it is then it invokes the constructor.

Parameters

json_dict : Dict
Arguments for the constructor.

****kwargs : Dict[str, Any]**
Additional class-dependent arguments.

Returns
Instance created from provided JSON.

Return type
Any

to_json() → dict[str, Any]

Convert object to JSON that contains its type and all parameters.

Returns
JSON config of given object.

Return type
Dict[str, Any]

15.17 ResourceManager

The `ResourceManager` is a singleton class which handles local and remote files for Kenning, such as datasets or models. It downloads missing files, provides paths to the available files, resolves custom URL schemes for and manages resource directories for Kenning, including cleanup.

class kenning.utils.resource_manager.**ResourceManager**(*args, **kwargs)

Bases: `object`

Download and cache resources used by Kenning.

add_custom_url_schemes(custom_url_schemes: *dict[str, str | Callable | None]*)

Add user defined URL schemes.

Parameters

custom_url_schemes : Dict[str, Optional[Union[str, Callable]]]
Dictionary with custom url schemes entries. Each entry consists of schema and corresponding conversion. Conversion can be None, string pattern or callable returning string.

clear_cache()

Remove all cached files.

get_resource(uri: *str*, output_path: *Path* | *None* = **None**) → Path

Retrieve file and return path to it.

If the uri points to remote resource, then it is downloaded (if not found in cache) and validated.

Parameters

uri : str
Resource URI.

output_path : Optional[Path]
Path to the output file. If not provided then the path is automatically created.

Returns
Path to the retrieved resource.

Return type
Path

Raises
ChecksumVerifyError : – Raised when downloaded file has invalid checksum

list_cached_files() → list[Path]

Return list with cached files.

Returns
List of cached files.

Return type
List[Path]

set_cache_dir(cache_dir_path: *Path*)

Set the cache directory path and creates it if not exists.

Parameters

cache_dir_path : Path
Path to be set as cache directory.

set_max_cache_size(max_cache_size: *int*)

Set the max cache size.

Parameters

max_cache_size : int
Max cache size in bytes.

validate_resources_version()

Retrieve Kenning resources version and check if it is compatible with currently used Kenning.

15.18 ResourceURI

The `ResourceURI` class is a `pathlib.Path`-based object allowing the user to work with Kenning resources, both using regular paths and URI schemes supported by the *ResourceManager*.

class kenning.utils.resource_manager.**ResourceURI**(uri_or_path: *str* | *Path* | ResourceURI)

Bases: `Path`

Handle access to resource used in Kenning.

property origin : str

Returns the original string passed to the constructor.

property parent : *ResourceURI*

Get parent of the URI.

property uri : str | None

Get URI of the resource.

with_name(name: *str*) → *ResourceURI*

Return new URI with changed name.

Parameters

name : str
New name to be used.

Returns
URI with changed name.

Return type
ResourceURI

with_stem(stem: *str*) → *ResourceURI*

Return new URI with changed stem.

Parameters

stem : str
New stem to be used.

Returns
URI with changed stem.

Return type
ResourceURI

with_suffix(suffix: *str*) → *ResourceURI*

Return new URI with changed suffix.

Parameters

suffix : str
New suffix to be used.

Returns
URI with changed suffix.

Return type
ResourceURI

PYTHON MODULE INDEX

k

kenning.core.measurements, 177

INDEX

Symbols

_dataindex (*kenning.core.dataset.Dataset attribute*), 142

A

accumulate() (*kenning.core.measurements.Measurements method*), 177
add_custom_url_schemes() (*kenning.utils.resource_manager.ResourceManager method*), 186
add_entry() (*kenning.core.onnxconversion.ONNXConversion method*), 181
add_measurement() (*kenning.core.measurements.Measurements method*), 178
ArgumentsHandler (*class in kenning.utils.args_manager*), 185
AutoML (*class in kenning.core.automl*), 166
AutoMLModel (*class in kenning.core.automl*), 165

B

batch_size (*kenning.core.dataset.Dataset attribute*), 142

C

calibration_dataset_generator() (*kenning.core.dataset.Dataset method*), 143
check_conversions() (*kenning.core.onnxconversion.ONNXConversion method*), 181
cleanup() (*kenning.core.runner.Runner method*), 139
clear() (*kenning.core.measurements.Measurements method*), 178
clear() (*kenning.core.measurements.MeasurementsCollector class method*), 179
clear_cache() (*kenning.utils.resource_manager.ResourceManager method*), 186
compile() (*kenning.core.optimizer.Optimizer method*), 154
consult_model_type() (*kenning.core.optimizer.Optimizer method*), 154
convert_input_to_bytes() (*kenning.core.model.ModelWrapper method*), 148
convert_output_from_bytes() (*kenning.core.model.ModelWrapper method*), 149
copy() (*kenning.core.measurements.Measurements method*), 178

D

data (*kenning.core.measurements.Measurements attribute*), 177
DataConverter (*class in kenning.core.dataconverter*), 176
DataProvider (*class in kenning.core.dataprovider*), 183
Dataset (*class in kenning.core.dataset*), 141
dataX (*kenning.core.dataset.Dataset attribute*), 141
dataXtest (*kenning.core.dataset.Dataset attribute*), 142
dataXtrain (*kenning.core.dataset.Dataset attribute*), 142
dataXval (*kenning.core.dataset.Dataset attribute*), 142
dataY (*kenning.core.dataset.Dataset attribute*), 141
dataYtest (*kenning.core.dataset.Dataset attribute*), 142
dataYtrain (*kenning.core.dataset.Dataset attribute*), 142
dataYval (*kenning.core.dataset.Dataset attribute*), 142

deinit() (*kenning.core.platform.Platform method*), 164
derive_io_spec_from_json_params() (*kenning.core.model.ModelWrapper class method*), 149
detach_from_output() (*kenning.core.outputcollector.OutputCollector method*), 184
detach_from_source() (*kenning.core.dataprovider.DataProvider method*), 183
disconnect() (*kenning.core.protocol.Protocol method*), 170
download_dataset_fun() (*kenning.core.dataset.Dataset method*), 143
download_output() (*kenning.core.protocol.Protocol method*), 170
download_statistics() (*kenning.core.protocol.Protocol method*), 170

E

evaluate() (*kenning.core.dataset.Dataset method*), 143
extract_model() (*kenning.core.automl.AutoMLModel method*), 165
extract_output() (*kenning.core.runtime.Runtime method*), 158

F

fetch_input() (*kenning.core.dataprovider.DataProvider method*), 183
form_argparse() (*kenning.utils.args_manager.ArgumentsHandler class method*), 185
form_automl_schema() (*kenning.core.automl.AutoMLModel class method*), 165
form_parameterschema() (*kenning.core.flow.KenningFlow class method*), 138
form_parameterschema() (*kenning.utils.args_manager.ArgumentsHandler class method*), 185
from_argparse() (*kenning.core.automl.AutoML class method*), 166
from_argparse() (*kenning.core.dataset.Dataset class method*), 143
from_argparse() (*kenning.core.model.ModelWrapper class method*), 149
from_argparse() (*kenning.core.optimizer.Optimizer class method*), 155
from_argparse() (*kenning.core.protocol.Protocol class method*), 170
from_argparse() (*kenning.core.runner.Runner class method*), 139
from_argparse() (*kenning.core.runtime.Runtime class method*), 158
from_argparse() (*kenning.utils.args_manager.ArgumentsHandler class method*), 185
from_json() (*kenning.core.automl.AutoML class method*), 166
from_json() (*kenning.core.dataset.Dataset class method*), 143
from_json() (*kenning.core.flow.KenningFlow class method*), 138
from_json() (*kenning.core.model.ModelWrapper class method*), 150
from_json() (*kenning.core.optimizer.Optimizer class method*), 155
from_json() (*kenning.core.protocol.Protocol class method*), 171
from_json() (*kenning.core.runner.Runner class method*), 139
from_json() (*kenning.core.runtime.Runtime class method*), 159
from_json() (*kenning.utils.args_manager.ArgumentsHandler class method*), 186

G

gather_data() (*kenning.core.protocol.Protocol method*), 171
get_available_ram() (*kenning.core.runtime.Runtime static method*), 159
get_best_configs() (*kenning.core.automl.AutoML method*), 167
get_class_names() (*ken-*

ning.core.dataset.Dataset method), 144
get_data() (kenning.core.dataset.Dataset method), 144
get_data_unloaded() (kenning.core.dataset.Dataset method), 144
get_default_protocol() (kenning.core.platform.Platform method), 164
get_framework_and_version() (kenning.core.model.ModelWrapper method), 150
get_framework_and_version() (kenning.core.optimizer.Optimizer method), 155
get_input_formats() (kenning.core.optimizer.Optimizer class method), 156
get_input_formats() (kenning.core.runtime.Runtime class method), 159
get_input_mean_std() (kenning.core.dataset.Dataset method), 144
get_input_type() (kenning.core.optimizer.Optimizer method), 156
get_io_spec_path() (kenning.core.runtime.Runtime method), 159
get_io_specification() (kenning.core.model.ModelWrapper method), 150
get_io_specification_from_model() (kenning.core.model.ModelWrapper method), 150
get_measurements() (kenning.core.measurements.SystemStatsCollector method), 179
get_optimized_model_size() (kenning.core.optimizer.Optimizer method), 156
get_output_formats() (kenning.core.model.ModelWrapper class method), 151
get_output_formats() (kenning.core.optimizer.Optimizer class method), 156
get_path() (kenning.core.model.ModelWrapper method), 151
get_resource() (kenning.utils.resource_manager.ResourceManager method), 187
get_spec_path() (kenning.core.optimizer.Optimizer static method), 156
get_statistics() (kenning.core.automl.AutoML method), 167
get_time() (kenning.core.platform.Platform method), 164
get_time() (kenning.core.runtime.Runtime method), 160
get_values() (kenning.core.measurements.Measurements method), 178

I

infer() (kenning.core.runtime.Runtime method), 160
inference_session_end() (kenning.core.runtime.Runtime method), 160
inference_session_start() (kenning.core.runtime.Runtime method), 160
inference_step_callback() (kenning.core.platform.Platform method), 164
init() (kenning.core.optimizer.Optimizer method), 157
init() (kenning.core.platform.Platform method), 164
initialize_client() (kenning.core.protocol.Protocol method), 171
initialize_measurement() (kenning.core.measurements.Measurements method), 178
initialize_server() (kenning.core.protocol.Protocol method), 172
iter_test() (kenning.core.dataset.Dataset method), 145
iter_train() (kenning.core.dataset.Dataset method), 145
iter_val() (kenning.core.dataset.Dataset method), 145

K

kenning.core.measurements

module, 177
KenningFlow (*class in kenning.core.flow*), 138

L

list_cached_files() (*kenning.utils.resource_manager.ResourceManager method*), 187
load_input() (*kenning.core.runtime.Runtime method*), 160
load_input_from_bytes() (*kenning.core.runtime.Runtime method*), 161
load_io_specification() (*kenning.core.optimizer.Optimizer method*), 157
load_model() (*kenning.core.model.ModelWrapper method*), 151

M

Measurements (*class in kenning.core.measurements*), 177
MeasurementsCollector (*class in kenning.core.measurements*), 179
ModelWrapper (*class in kenning.core.model*), 148
module
kenning.core.measurements, 177

O

onnx_export() (*kenning.core.onnxconversion.ONNXConversion method*), 182
onnx_import() (*kenning.core.onnxconversion.ONNXConversion method*), 182
ONNXConversion (*class in kenning.core.onnxconversion*), 181
Optimizer (*class in kenning.core.optimizer*), 154
origin (*kenning.utils.resource_manager.ResourceURI property*), 188
OutputCollector (*class in kenning.core.outputcollector*), 184

P

parent (*kenning.utils.resource_manager.ResourceURI property*), 188
parse_io_specification_from_json() (*kenning.core.model.ModelWrapper class method*), 151
Platform (*class in kenning.core.platform*), 164
postprocess_outputs() (*kenning.core.model.ModelWrapper method*), 152
prepare() (*kenning.core.dataprovider.DataProvider method*), 183
prepare() (*kenning.core.dataset.Dataset method*), 145
prepare() (*kenning.core.onnxconversion.ONNXConversion method*), 182
prepare_external_calibration_dataset() (*kenning.core.dataset.Dataset method*), 145
prepare_framework() (*kenning.core.automl.AutoML method*), 168
prepare_input_samples() (*kenning.core.dataset.Dataset method*), 146
prepare_io_specification() (*kenning.core.runtime.Runtime method*), 161
prepare_local() (*kenning.core.runtime.Runtime method*), 161
prepare_model() (*kenning.core.model.ModelWrapper method*), 152
prepare_model() (*kenning.core.runtime.Runtime method*), 162
prepare_output_samples() (*kenning.core.dataset.Dataset method*), 146
preprocess_input() (*kenning.core.dataprovider.DataProvider method*), 183
preprocess_input() (*kenning.core.model.ModelWrapper method*), 152
preprocess_model_to_upload() (*kenning.core.runtime.Runtime method*), 162
process_output() (*kenning.core.outputcollector.OutputCollector method*), 184
Protocol (*class in kenning.core.protocol*), 170

R

read_data_from_platforms_yaml() (*ken-*

ning.core.platform.Platform method), 164
read_io_specification() (kenning.core.runtime.Runtime method), 162
read_platform() (kenning.core.model.ModelWrapper method), 152
read_platform() (kenning.core.optimizer.Optimizer method), 157
receive_confirmation() (kenning.core.protocol.Protocol method), 172
receive_data() (kenning.core.protocol.Protocol method), 172
receive_message() (kenning.core.protocol.Protocol method), 172
request_failure() (kenning.core.protocol.Protocol method), 173
request_optimization() (kenning.core.protocol.Protocol method), 173
request_processing() (kenning.core.protocol.Protocol method), 173
request_success() (kenning.core.protocol.Protocol method), 173
ResourceManager (class in kenning.utils.resource_manager), 186
ResourceURI (class in kenning.utils.resource_manager), 188
run() (kenning.core.flow.KenningFlow method), 139
run() (kenning.core.measurements.SystemStatsCollector method), 180
run() (kenning.core.runner.Runner method), 140
run() (kenning.core.runtime.Runtime method), 163
run_inference() (kenning.core.model.ModelWrapper method), 153
run_single_step() (kenning.core.flow.KenningFlow method), 139
Runner (class in kenning.core.runner), 139
Runtime (class in kenning.core.runtime), 158

S

save_dataset_checksum() (kenning.core.dataset.Dataset method), 146
save_io_specification() (kenning.core.optimizer.Optimizer method), 157
save_measurements() (kenning.core.measurements.MeasurementsCollector class method), 179
save_model() (kenning.core.model.ModelWrapper method), 153
save_to_onnx() (kenning.core.model.ModelWrapper method), 153
search() (kenning.core.automl.AutoML method), 168
send_data() (kenning.core.protocol.Protocol method), 174
send_message() (kenning.core.protocol.Protocol method), 174
set_batch_size() (kenning.core.dataset.Dataset method), 146
set_cache_dir() (kenning.utils.resource_manager.ResourceManager method), 187
set_compiled_model_path() (kenning.core.optimizer.Optimizer method), 158
set_input_type() (kenning.core.optimizer.Optimizer method), 158
set_max_cache_size() (kenning.utils.resource_manager.ResourceManager method), 187
set_unoptimized() (kenning.core.measurements.MeasurementsCollector class method), 179
should_close() (kenning.core.outputcollector.OutputCollector method), 184
should_close() (kenning.core.runner.Runner method), 140
SystemStatsCollector (class in kenning.core.measurements), 179
systemstatsmeasurements() (in module ken-

ning.core.measurements), 180

T

tagmeasurements() (*in module kenning.core.measurements*), 180
test_inference() (*kenning.core.model.ModelWrapper method*), 153
test_subset_len() (*kenning.core.dataset.Dataset method*), 146
timemeasurements() (*in module kenning.core.measurements*), 181
to_json() (*kenning.utils.args_manager.ArgumentsHandler method*), 186
to_next_block() (*kenning.core.dataconverter.DataConverter method*), 176
to_previous_block() (*kenning.core.dataconverter.DataConverter method*), 176
train_model() (*kenning.core.model.ModelWrapper method*), 153
train_subset_len() (*kenning.core.dataset.Dataset method*), 147
train_test_split_representations() (*kenning.core.dataset.Dataset method*), 147

U

update_automl_defaults() (*kenning.core.automl.AutoMLModel static method*), 165
update_automl_range() (*kenning.core.automl.AutoMLModel class method*), 166
update_measurements() (*kenning.core.measurements.Measurements method*), 178
upload_input() (*kenning.core.protocol.Protocol method*), 174
upload_io_specification() (*kenning.core.protocol.Protocol method*), 174
upload_model() (*kenning.core.protocol.Protocol method*), 175
upload_optimizers() (*kenning.core.protocol.Protocol method*), 175
upload_output() (*kenning.core.runtime.Runtime method*), 163
upload_runtime() (*kenning.core.protocol.Protocol method*), 175
upload_stats() (*kenning.core.runtime.Runtime method*), 163
uri (*kenning.utils.resource_manager.ResourceURI property*), 188

V

val_subset_len() (*kenning.core.dataset.Dataset method*), 147
validate_resources_version() (*kenning.utils.resource_manager.ResourceManager method*), 187
verify_dataset_checksum() (*kenning.core.dataset.Dataset method*), 147

W

with_name() (*kenning.utils.resource_manager.ResourceURI method*), 188
with_stem() (*kenning.utils.resource_manager.ResourceURI method*), 188
with_suffix() (*kenning.utils.resource_manager.ResourceURI method*), 188