



antmicro

**Antmicro**

**Kenning**

2025-02-03

## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Kenning</b>	<b>2</b>
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
<b>3</b>	<b>Deep Learning deployment stack</b>	<b>15</b>
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
<b>4</b>	<b>Defining optimization pipelines in Kenning</b>	<b>18</b>
4.1	JSON specification . . . . .	18
4.2	Model evaluation using its native framework . . . . .	19
4.3	Optimizing and running a model on a single device . . . . .	21
4.4	Compiling a model and running it remotely . . . . .	23
<b>5</b>	<b>Using Kenning via command line arguments</b>	<b>26</b>
5.1	Command-line arguments for classes . . . . .	26
5.2	Autocompletion for command line interface . . . . .	26
5.3	Model training . . . . .	27
5.4	In-framework inference performance measurements . . . . .	28
5.5	Testing inference on target hardware . . . . .	29
5.6	Running inference . . . . .	34
5.7	Generating performance reports . . . . .	36
5.8	Displaying information about available classes . . . . .	36
<b>6</b>	<b>Kenning gallery of use cases</b>	<b>38</b>
6.1	Displaying information about available classes . . . . .	38
6.2	Evaluating models on hardware using Kenning Zephyr Runtime . . . . .	43
6.3	Visualizing Kenning data flows with Pipeline Manager . . . . .	46
6.4	Bare-metal IREE runtime simulated using Renode . . . . .	48
6.5	Structured pruning for PyTorch models . . . . .	51

6.6	Model quantization and compilation using TFLite and TVM . . . . .	55
6.7	Unstructured Pruning of TensorFlow Models . . . . .	66
<b>7</b>	<b>Kenning environment variables</b>	<b>73</b>
7.1	KENNING_CACHE_DIR . . . . .	73
7.2	KENNING_MAX_CACHE_SIZE . . . . .	73
7.3	KENNING_USE_DEFAULT_EXCEPTHOOK . . . . .	73
7.4	KENNING_DOCS_VENV . . . . .	73
7.5	KENNING_ENABLE_ALL_LOGS . . . . .	74
7.6	KENNING_DISABLE_IO_VALIDATION . . . . .	74
<b>8</b>	<b>Kenning measurements</b>	<b>75</b>
8.1	Performance metrics . . . . .	75
<b>9</b>	<b>Choosing optimal optimization pipeline</b>	<b>76</b>
9.1	Optimization config specification . . . . .	76
9.2	Output details . . . . .	78
<b>10</b>	<b>Sample autogenerated report</b>	<b>80</b>
10.1	Pet Dataset classification using TVM-compiled TensorFlow model . . . . .	80
10.2	Inference performance metrics for build.local-cpu-tvm-tensorflow-classification.json . . . . .	83
10.3	Inference quality metrics for build.local-cpu-tvm-tensorflow-classification.json . . . . .	84
<b>11</b>	<b>Creating applications with Kenning</b>	<b>87</b>
11.1	JSON structure . . . . .	87
11.2	KenningFlow execution . . . . .	90
11.3	Implemented Runners . . . . .	90
<b>12</b>	<b>Developing Kenning blocks</b>	<b>91</b>
12.1	Model and I/O metadata . . . . .	91
12.2	Implementing a new Kenning component . . . . .	94
12.3	Implementing Kenning runtime blocks . . . . .	104
<b>13</b>	<b>Kenning resources</b>	<b>106</b>
13.1	Accessing resources . . . . .	106
13.2	Caching . . . . .	107
13.3	Custom URI schemes . . . . .	107
13.4	CLI commands . . . . .	109
<b>14</b>	<b>Kenning API</b>	<b>110</b>
14.1	Deployment API overview . . . . .	110
14.2	KenningFlow . . . . .	112
14.3	Runner . . . . .	113
14.4	Dataset . . . . .	115
14.5	ModelWrapper . . . . .	122
14.6	Optimizer . . . . .	128
14.7	Runtime . . . . .	131
14.8	Protocol . . . . .	137
14.9	DataConverter . . . . .	144
14.10	Measurements . . . . .	145
14.11	ONNXConversion . . . . .	149
14.12	DataProvider . . . . .	151

14.13 OutputCollector . . . . .	152
14.14 ArgumentsHandler . . . . .	153
14.15 ResourceManager . . . . .	154
14.16 ResourceURI . . . . .	156
<b>Python Module Index</b>	<b>158</b>
<b>Index</b>	<b>159</b>

---

## 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

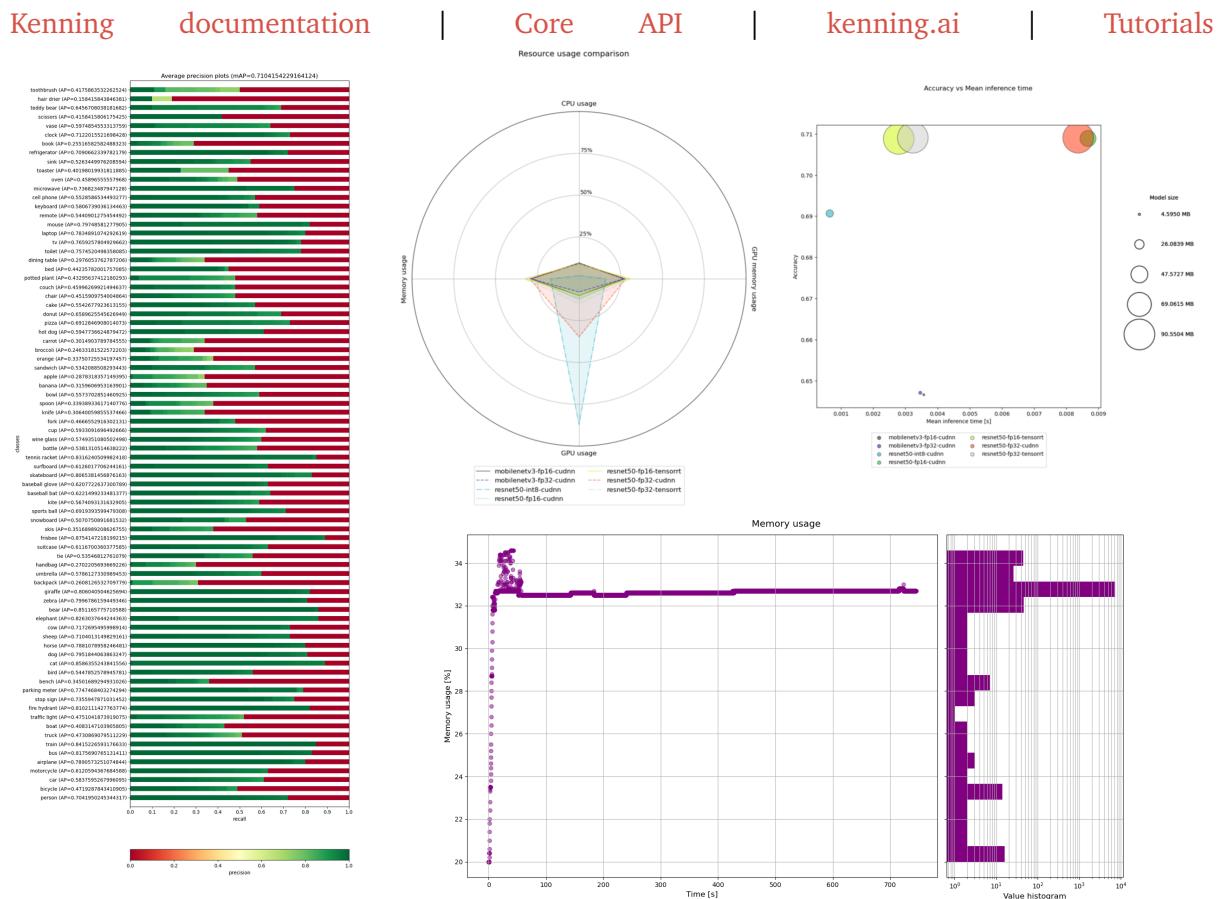
---

### KENNING

Copyright (c) 2020-2024 Antmicro



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



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

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

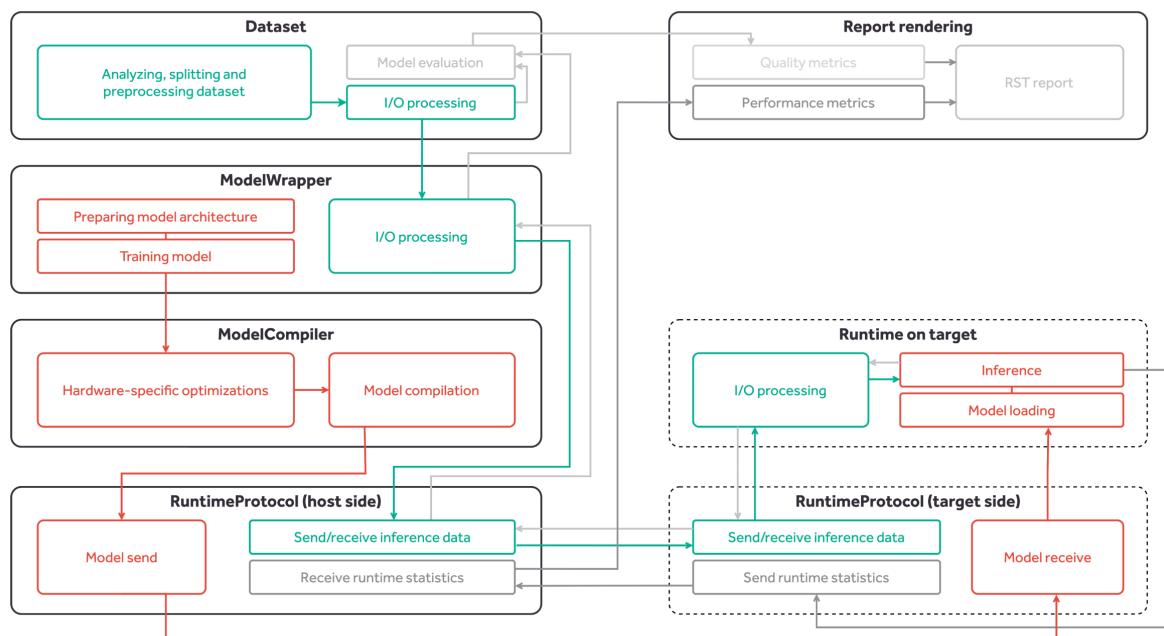
For development purposes, and to use additional resources (such as sample scripts), clone the repository with:

```
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.

## 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.

## 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 [\*Model-Wrapper\*](#) 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"
    }
  },
  "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 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 in the following scenario:

Listing 4.2: 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"
    }
  },
}
```

(continues on next page)

(continued from previous page)

```

"dataset": {
    "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
```

## 4.4 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 4.3: tflite-tvm-classification-client-server.json

```
{
  "model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
→TensorPetDatasetMobileNetV2",
    "parameters": {
      "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": "default",
            "compiled_model_path": "./build/compiled_tflite.tflite",
            "inference_input_type": "float32",
            "inference_output_type": "float32"
        }
    },
    {
        "type": "kenning.optimizers.tvm.TVMCompiler",
        "parameters":
        {
            "target": "llvm -mcpu=core-avx2",
            "compiled_model_path": "./build/compiled_tvm.tar",
            "opt_level": 3,
            "location": "target"
        }
    }
],
"runtime":
{
    "type": "kenning.runtimes.tvm.TVMRuntime",
    "parameters":
    {
        "save_model_path": "./build/compiled_model.tar"
    }
},
"protocol":
{
    "type": "kenning.protocols.network.NetworkProtocol",
    "parameters":
    {
        "host": "127.0.0.1",
        "port": 12345,
        "packet_size": 32768
    }
}
}
}

```

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 \
  --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).

## 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
```

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}
                    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
```

(continues on next page)

(continued from previous page)

```
--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 validation
--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 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
```

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 kenning.modelwrappers.classification.tensorflow_pet_\
→dataset.TensorFlowPetDatasetMobileNetV2 \
    --dataset-cls kenning.datasets.pet_dataset.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 kenning.modelwrappers.classification.tensorflow_pet_\
→dataset.TensorFlowPetDatasetMobileNetV2 \
    --runtime-cls kenning.runtimes.tvm.TVMRuntime \
    --dataset-cls kenning.datasets.pet_dataset.PetDataset \
```

(continues on next page)

(continued from previous page)

```
--compiler-cls kenning.optimizers.tvm.TVMCompiler \
--protocol-cls kenning.protocols.network.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
                     tester and inference runner

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

PetDataset arguments:
--classify-by {species,breeds}
```

(continues on next page)

(continued from previous page)

```

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 validation
--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:
--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

```

(continues on next page)

(continued from previous page)

```

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 kenning.modelwrappers.classification.tensorflow_pet_
→dataset.TensorFlowPetDatasetMobileNetV2 \
  --runtime-cls kenning.runtimes.tflite.TFLiteRuntime \
  --dataset-cls kenning.datasets.pet_dataset.PetDataset \
  --measurements ./build/google-coral-devboard-tflite-tensorflow.json \
  --compiler-cls kenning.optimizers.tflite.TFLiteCompiler \
  --protocol-cls kenning.protocols.network.NetworkProtocol \
  --model-path kenning:///models/classification/tensorflow_pet_dataset_
→mobilenetv2.h5 \

```

(continues on next page)

(continued from previous page)

```
--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 kenning.protocols.network.NetworkProtocol \
--runtime-cls kenning.runtimes.tflite.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 kenning.modelwrappers.classification.tensorflow_pet_\
→dataset.TensorFlowPetDatasetMobileNetV2 \
--runtime-cls kenning.runtimes.tvm.TVMRuntime \
--dataset-cls kenning.datasets.pet_dataset.PetDataset \
--measurements ./build/local-cpu-tvm-tensorflow-classification.json \
--compiler-cls kenning.optimizers.tvm.TVMCompiler \
--model-path kenning:///models/classification/tensorflow_pet_dataset_\
→mobilenetv2.h5 \
--model-framework keras \
--target "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/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 \
    kenning.runtimes.tvm.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_
  ↪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_
                        (continues on next page)
```

(continued from previous page)

```

→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,classifier}
    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 \
    kenning.runtimes.tvm.TVMRuntime \
    kenning.dataproviders.camera_dataprovider.CameraDataProvider \
    --output-collectors kenning.outputcollectors.detection_visualizer.
→DetectionVisualizer kenning.outputcollectors.name_printer.NamePrinter \
    --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 uses a module-like path to the file (e.g. kenning.runtimes.tflite), but optionally a class can be specified by adding it 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`.

## KENNING GALLERY OF USE CASES

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

### 6.1 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"
```

#### 6.1.1 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
kenning.runtimes.renode.RenodeRuntime
```

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

### 6.1.2 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 Model-Wrapper. 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
```

(continues on next page)

(continued from previous page)

```

* shape: (1, 255, (keyparams['width'] // (8 * (2 ** 2))), (keyparams['height'] /
  ↪ (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**

## 6.2 Evaluating models on hardware using Kenning Zephyr Runtime

This section contains tutorial of evaluating models on microcontrollers using [Kenning Zephyr Runtime](#) and [Renode](#).

### 6.2.1 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 xz-utils
```

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
```

## 6.2.2 Installing Kenning with Renode support

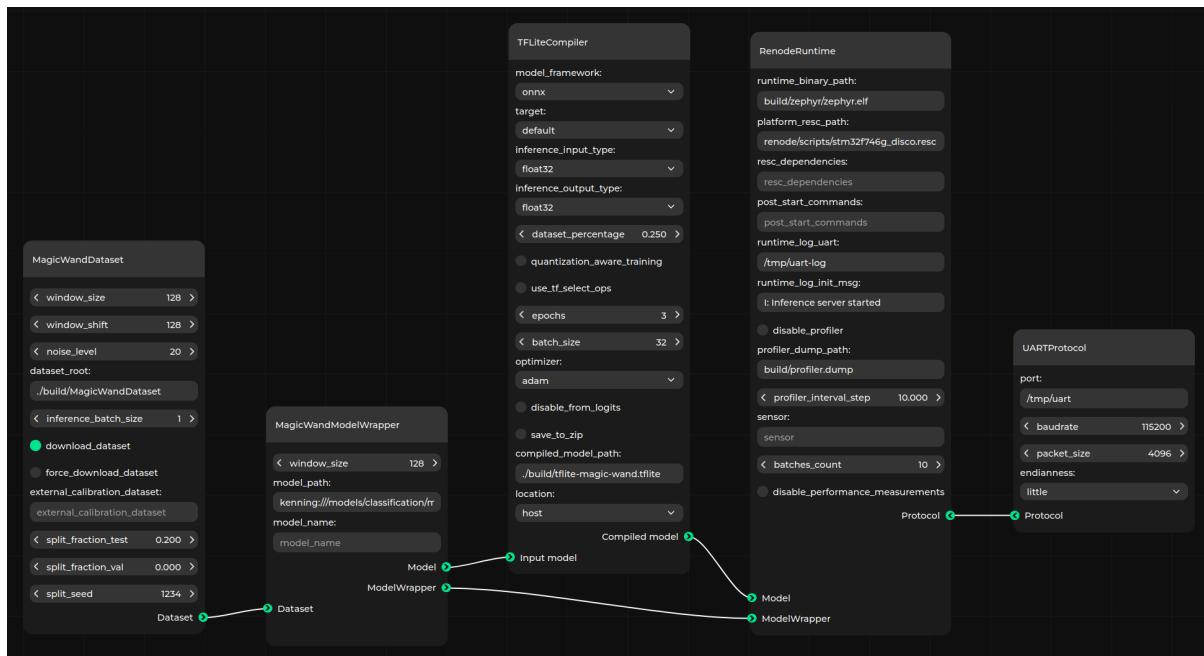
Evaluating models using Kenning Zephyr Runtime requires [Kenning](#) with Renode support. Use pip to install it:

```
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)
```

## 6.2.3 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
```

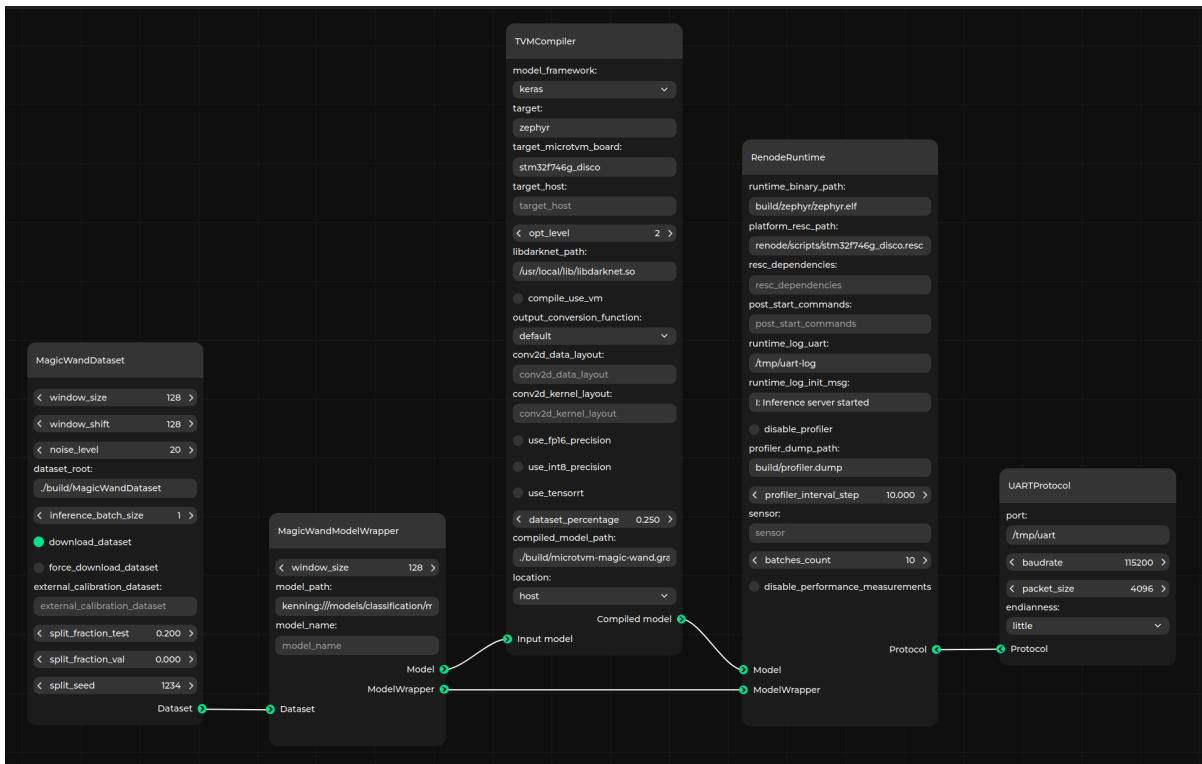
The built binary can be found in build/zephyr/zephyr.elf.

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

```
kenning optimize test \
  --json-cfg kenning-scenarios/renode-zephyr-tflite-magic-wand-inference.json \
  --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.

## 6.2.4 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 \
--json-cfg kenning-scenarios/renode-zephyr-tvm-magic-wand-inference.json \
--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`.

## 6.2.5 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
```

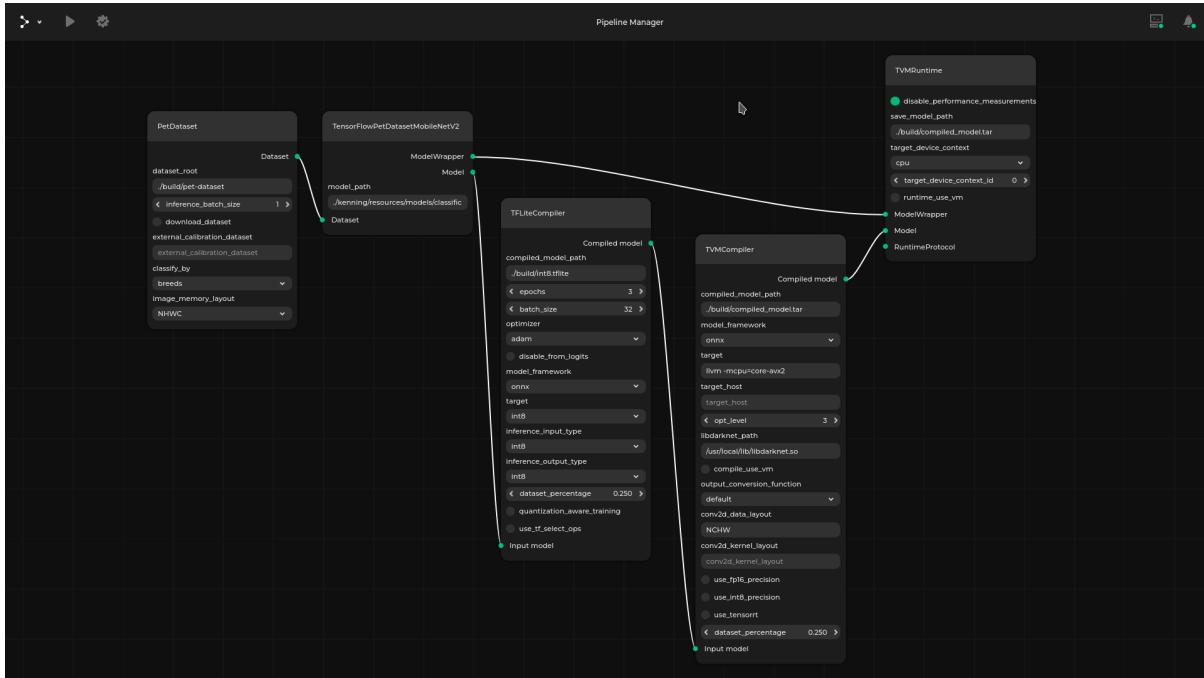
The HTML version of the report should be saved at `build/zephyr-stm32-tflite-tvm-comparison/zephyr-stm32-tflite-tvm-comparison.html`.

## 6.3 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.



### 6.3.1 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"
```

### 6.3.2 Running Pipeline Manager with Kenning

Start the Pipeline Manager client with:

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

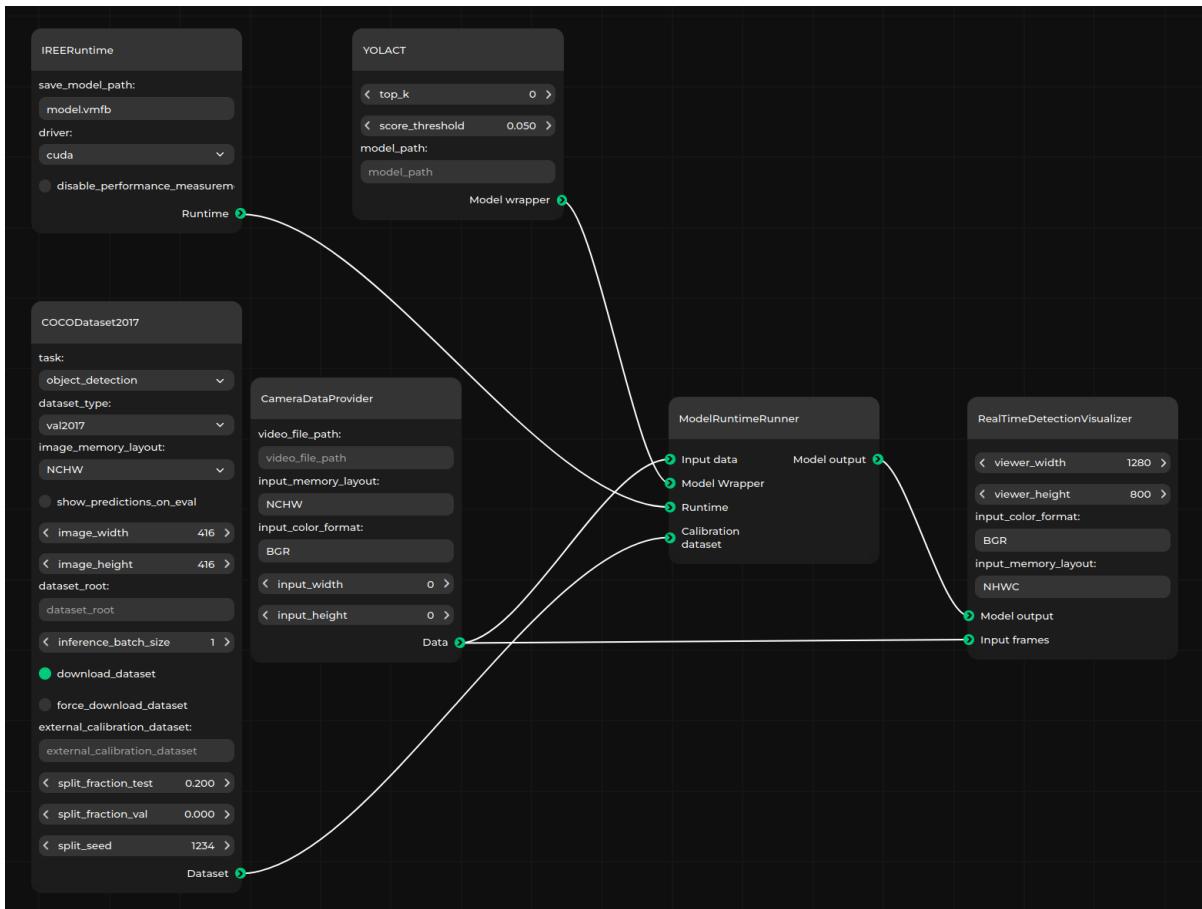
The --file-path option specifies where the results of model benchmarking or the runtime data will be stored.

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

### 6.3.3 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](#).

## 6.4 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](#)

### 6.4.1 Setup

For quick setup, use docker to open the environment with all necessary dependencies installed. Alternatively, install them manually by following the instructions in the [runtime's README.md](#):

```
docker run --rm -it ghcr.io/antmicro/kenning-bare-metal-iree-runtime:latest
```

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](#).

### 6.4.2 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:

```
{
  "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.vmf",
        "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-unknown-elf",
          "iree-llvm-target-cpu=generic-rv32",
          "iree-llvm-target-cpu-features=+m,+f,+zvl512b,+zve32x,+zve32f
        ↪",
          "iree-llvm-target-abi=ilp32"
        ]
      }
    }
  ],
  "runtime": {
    "type": "kenning.runtimes.renode.RenodeRuntime",
    "parameters": {
      "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"
    
```

(continues on next page)

(continued from previous page)

```

        ],
        "post_start_commands": [
            "sysbus.vec_controlblock WriteDoubleWord 0xc 0"
        ],
        "runtime_log_init_msg": "Runtime started",
        "profiler_dump_path": "build/profiler.dump"
    },
},
"protocol": {
    "type": "kenning.protocols uart.UARTProtocol",
    "parameters": {
        "port": "/tmp/uart",
        "baudrate": 115200,
        "endianness": "little"
    }
}
}

```

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.

Rnode simulation is enabled by specifying the RenodeRuntime in the runtime entry and setting the following parameters:

- 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.vmf
- 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.

## 6.5 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](#).

### 6.5.1 Setup

Install required dependencies:

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

## 6.5.2 Experiments

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

Listing 6.1: 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"
    }
  }
}
```

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 6.2: 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": {
      "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,
      }
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    "finetuning_batch_size": 64,
    "confidence": 8,
    "criterion": "torch.nn.CrossEntropyLoss",
    "optimizer": "torch.optim.Adam",
    "pruning_on_cuda": true,
    "finetuning_learning_rate": 5e-05,
    "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",
  }
}
```

(continues on next page)

(continued from previous page)

```

    "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
}
}

```

### 6.5.3 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%

## 6.6 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.

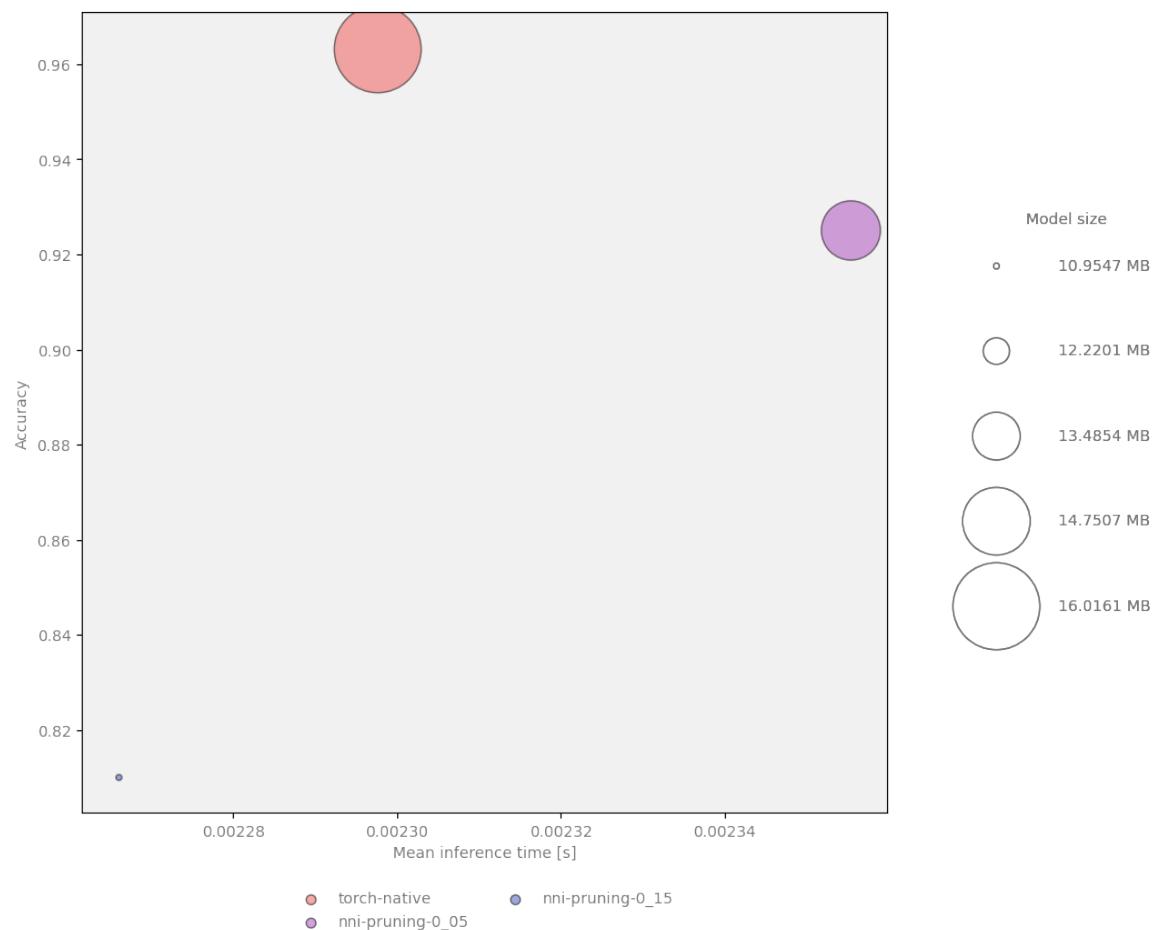


Figure 6.1: Model size, speed and quality comparison for NNI pruning

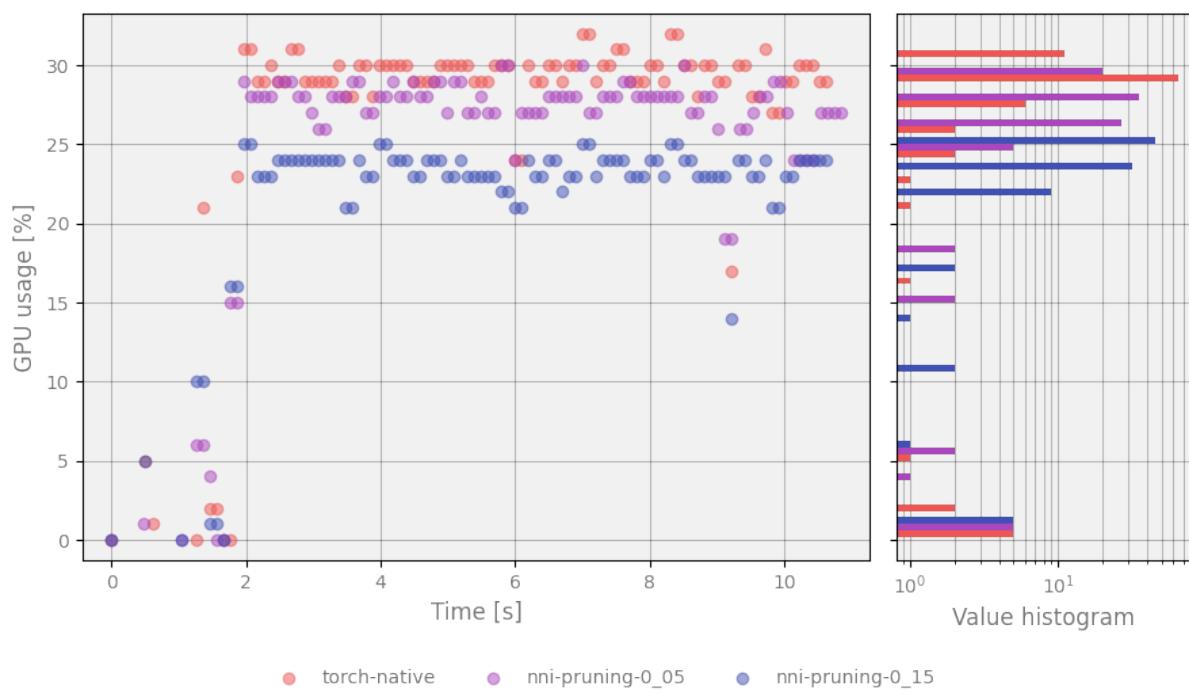


Figure 6.2: Plot represents changes of GPU usage over time

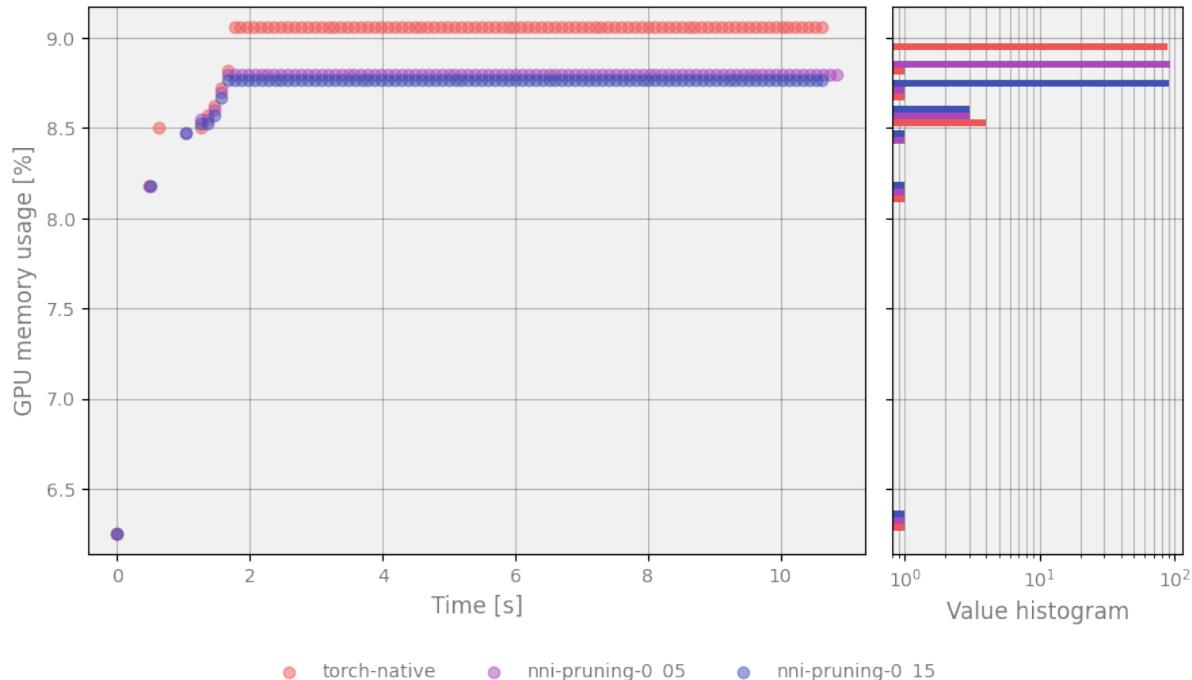


Figure 6.3: Plot represents changes of GPU RAM usage over time

### 6.6.1 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"
```

### 6.6.2 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
```

### 6.6.3 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 6.3: 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"
    }
  },
  "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.

#### 6.6.4 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 6.4: 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:

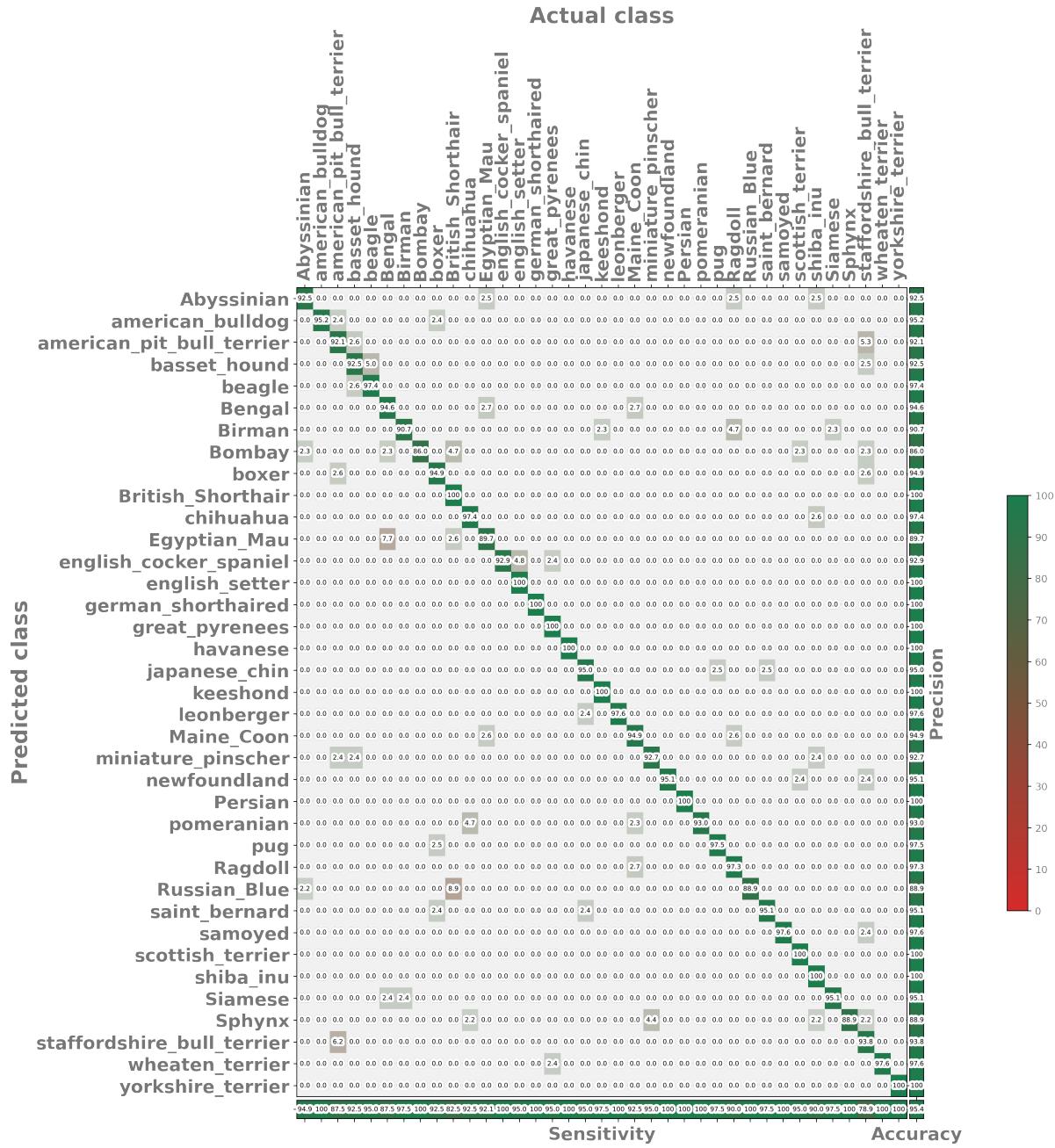
```
kenning optimize test report \
--json-cfg mobilenetv2-tensorflow-tflite-f32.json \
```

(continues on next page)

(continued from previous page)

```
--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.



### 6.6.5 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 representations (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 6.5: 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).

### 6.6.6 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 6.6: mobilenetv2-tensorflow-tvm-avx-int8.json

```
"optimizers": [
  {
    "type": "kenning.optimizers.tflite.TFLiteCompiler",
    "parameters": {
      "target": "int8",
    }
  }
]
```

(continues on next page)

(continued from previous page)

```

    "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"
  }
}

```

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

### 6.6.7 Automated model comparison

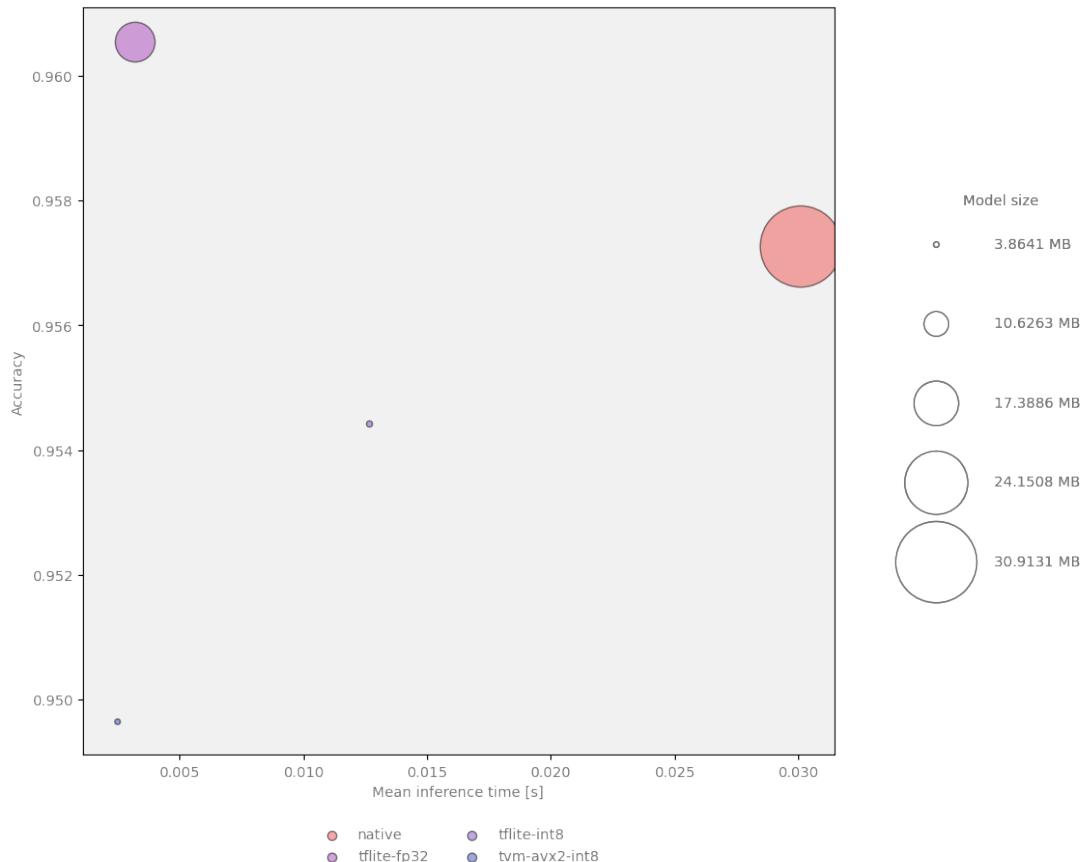
The kenning report tool also allows us to compare evaluation results for multiple models. Apart from creating a model summary table, it also creates plots aggregating measurements collected during the evaluation process.

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

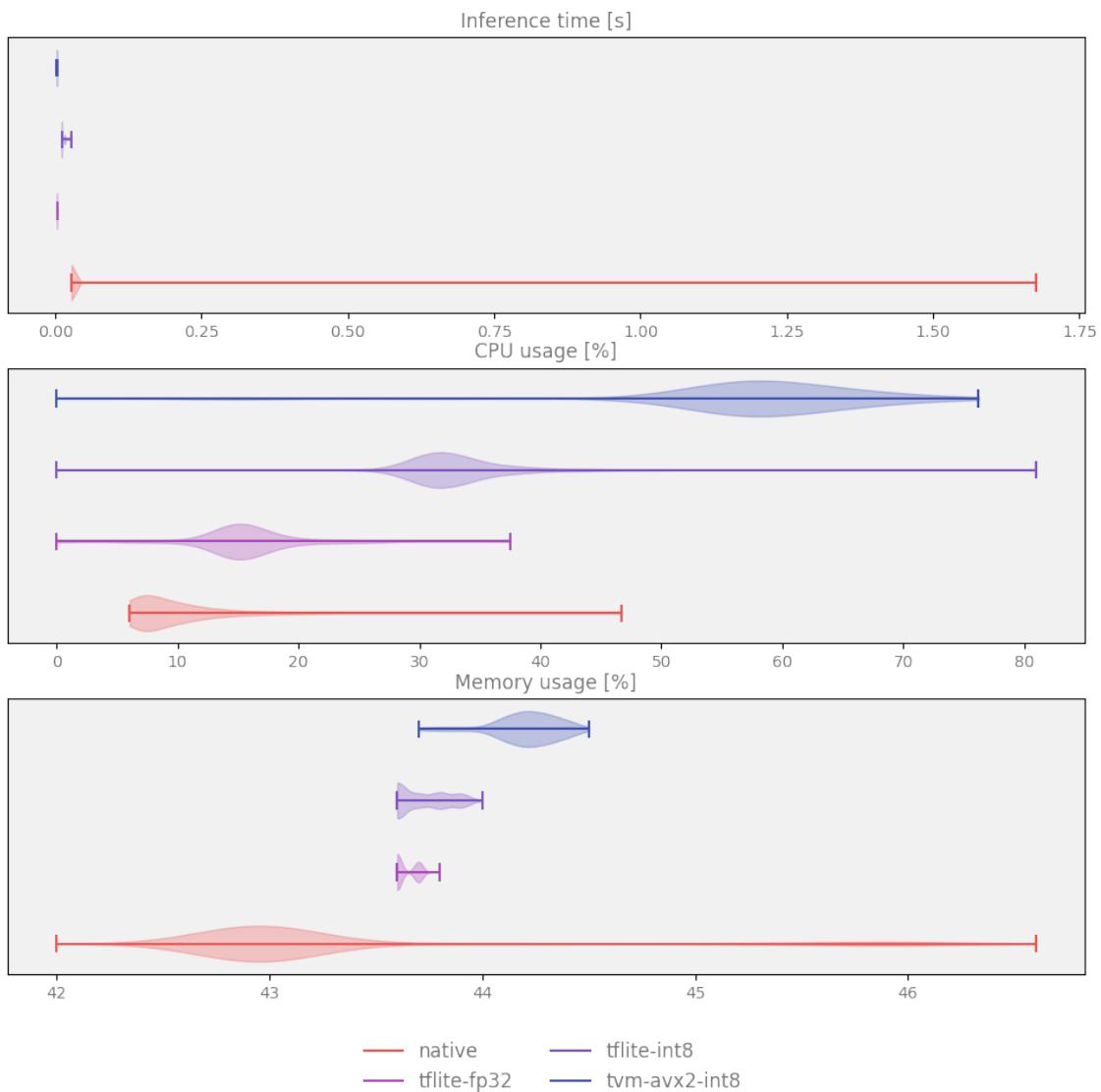
```
kenning report \
  --report-path build/benchmarks/summary.md \
  --measurements \
    build/native.json \
    build/tflite-fp32.json \
    build/tflite-int8.json \
    build/tvm-avx2-int8.json
```

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

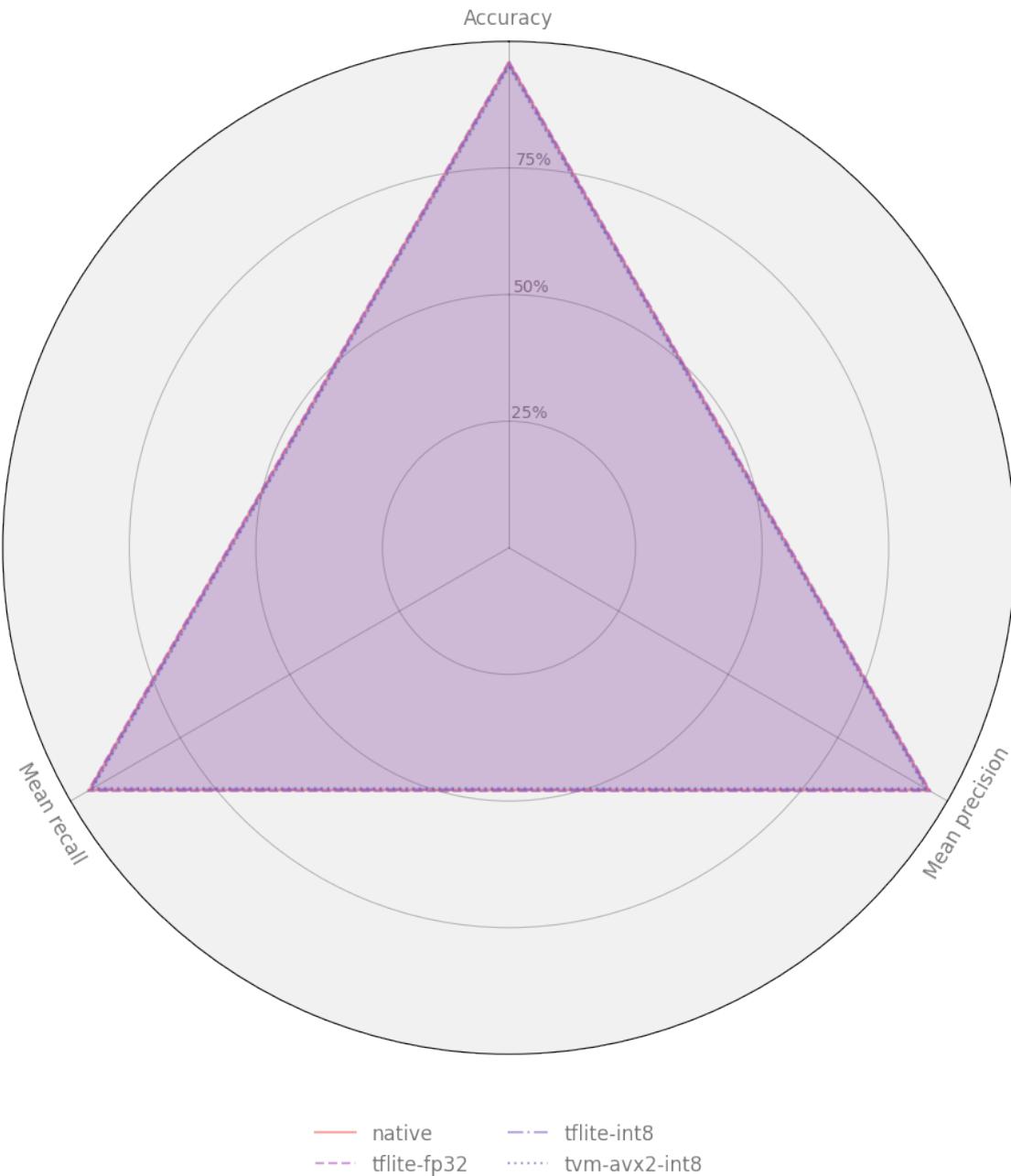
- Accuracy, inference time and model size comparison:



- Resource utilization distribution:



- Comparison of classification metrics:



- And more

## 6.7 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.

## 6.7.1 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"
```

## 6.7.2 Experiments

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

Listing 6.7: 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 6.8: 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%

### 6.7.3 (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 6.9: 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)

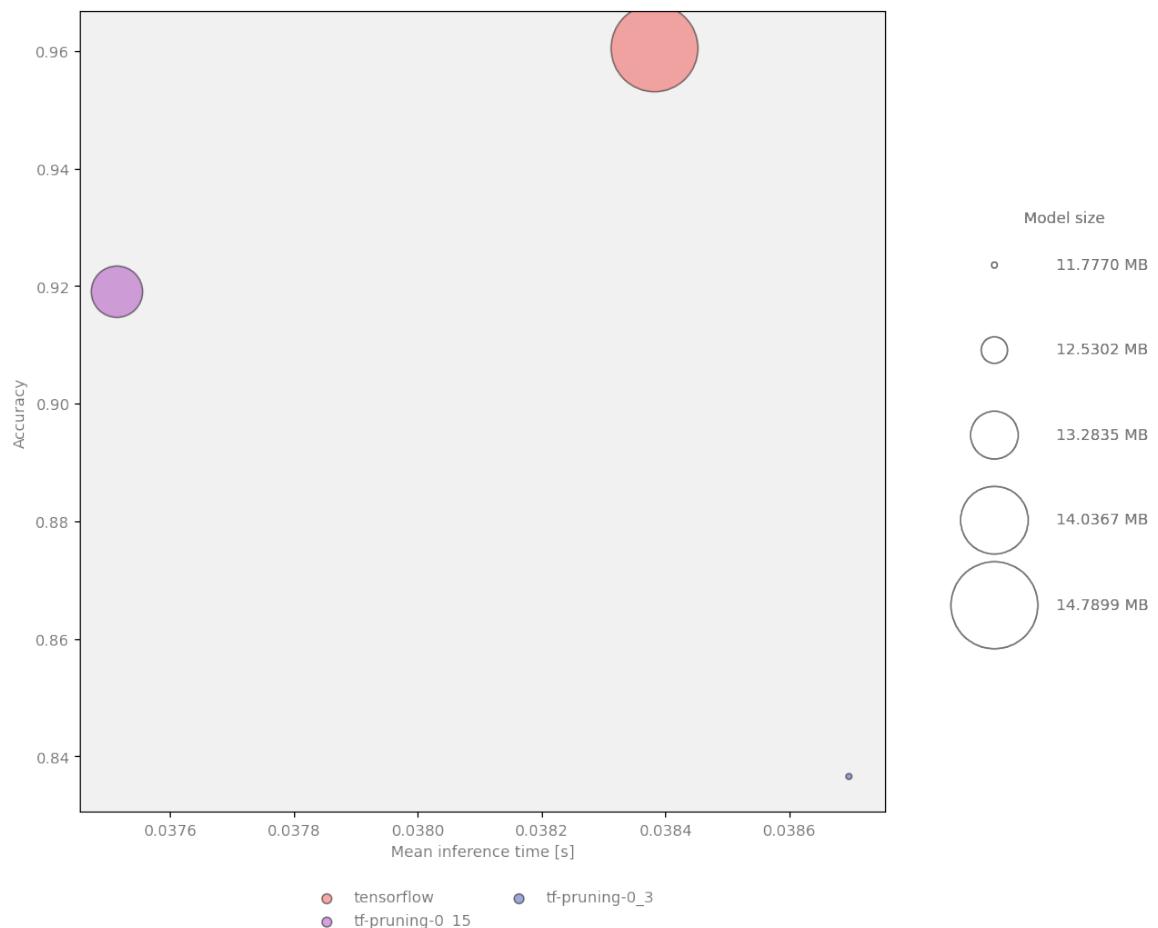


Figure 6.4: Model size, speed and quality comparison for TensorFlow pruning

(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%

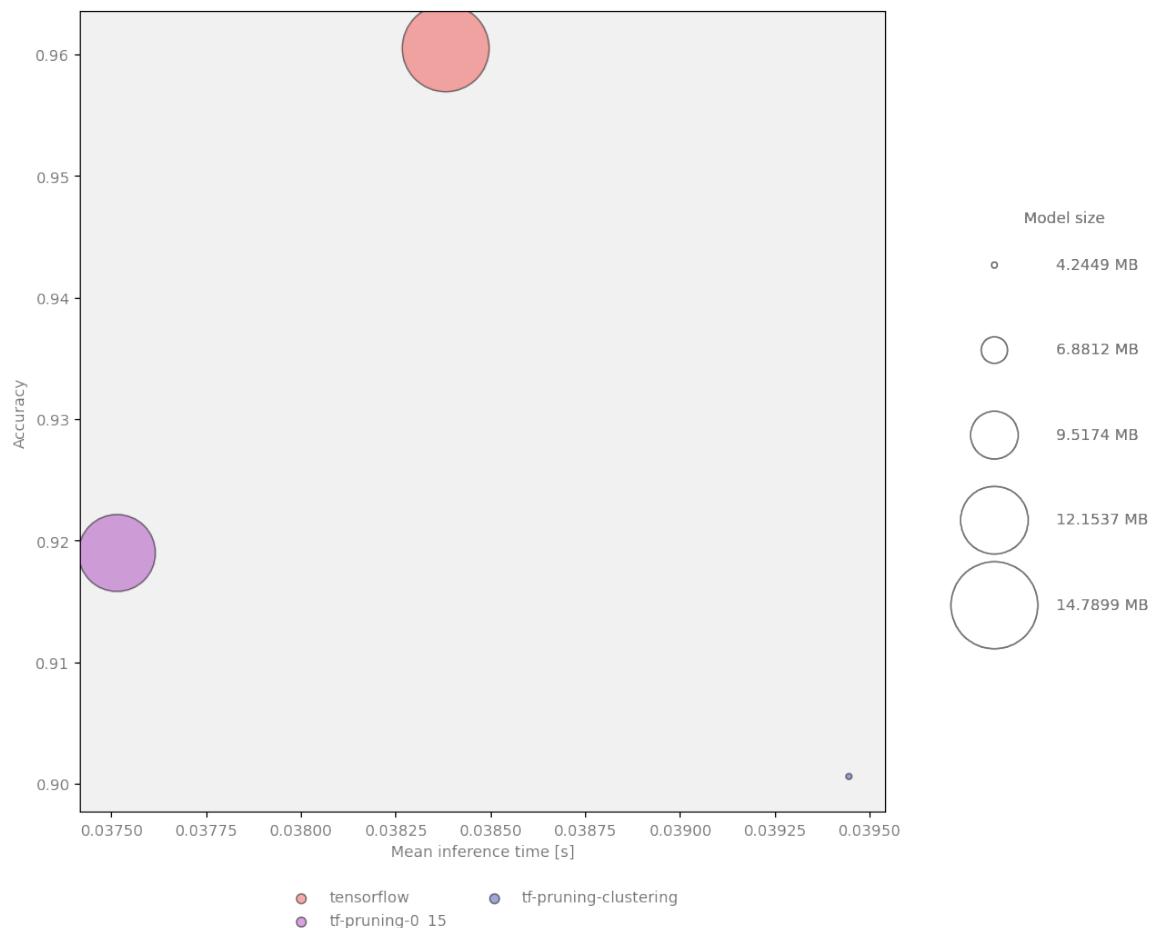


Figure 6.5: Model size, speed and quality comparison for TensorFlow pruning and clustering

## 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).

## 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.  
→TensorPetDatasetMobileNetV2",  
        "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 its 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 its 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.

## 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. 2.11.1

*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
    }
},
"model_wrapper": {
    "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
    ↪TensorFlowPetDatasetMobileNetV2",
    "parameters": {
        "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
    }
},
"data_converter": {
    "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",
        }
    }
]
}

```

(continues on next page)

(continued from previous page)

```

        "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"
    }
}
]
}

```

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

### 10.2.1 Inference time

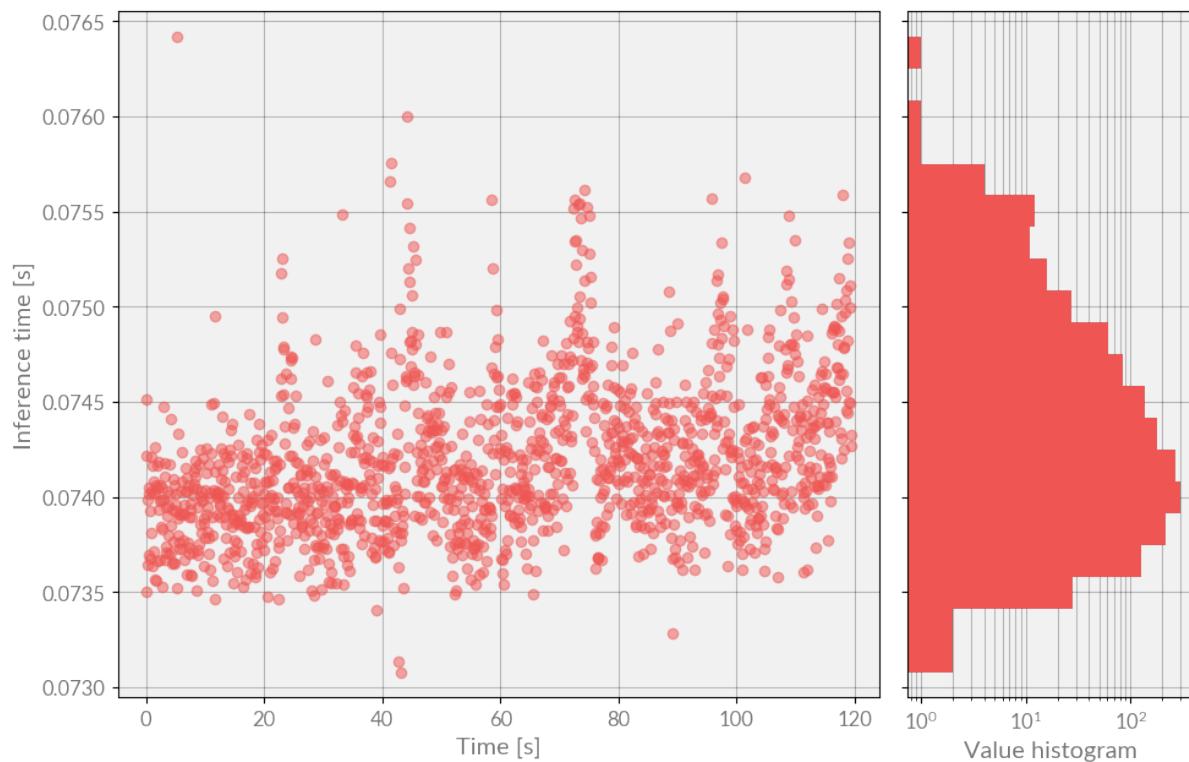


Figure 10.1: Inference time

- *First inference duration* (usually including allocation time): **0.0772869379999861**,
- **Mean:** **0.07418959635646176** s,
- **Standard deviation:** **0.0004082952863268992** s,
- **Median:** **0.07411699150000572** s.

### 10.2.2 Average CPU usage

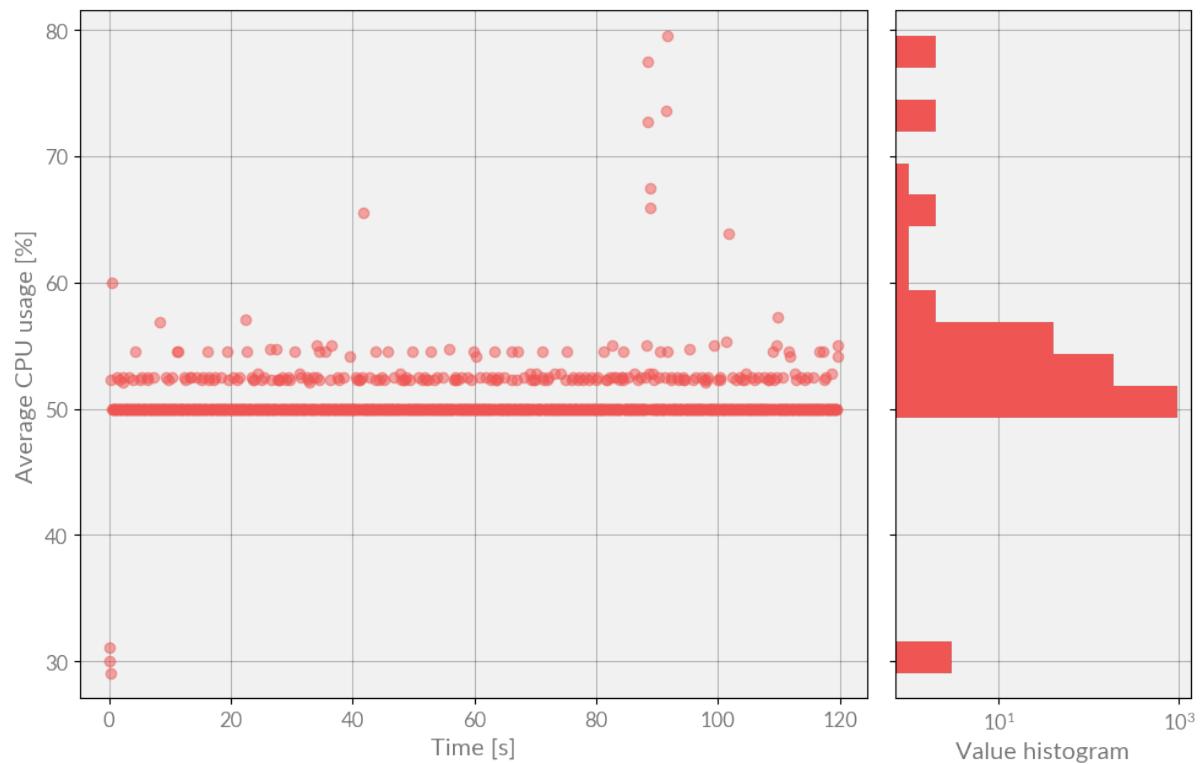


Figure 10.2: Average CPU usage during benchmark

- *Mean:* **50.6328637510513 %,**
- *Standard deviation:* **2.474337415693573 %,**
- *Median:* **50.0 %.**

### 10.2.3 Memory usage

- *Mean:* **9.064507989907485 %,**
- *Standard deviation:* **0.047848910424839235 %,**
- *Median:* **9.1 %.**

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

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

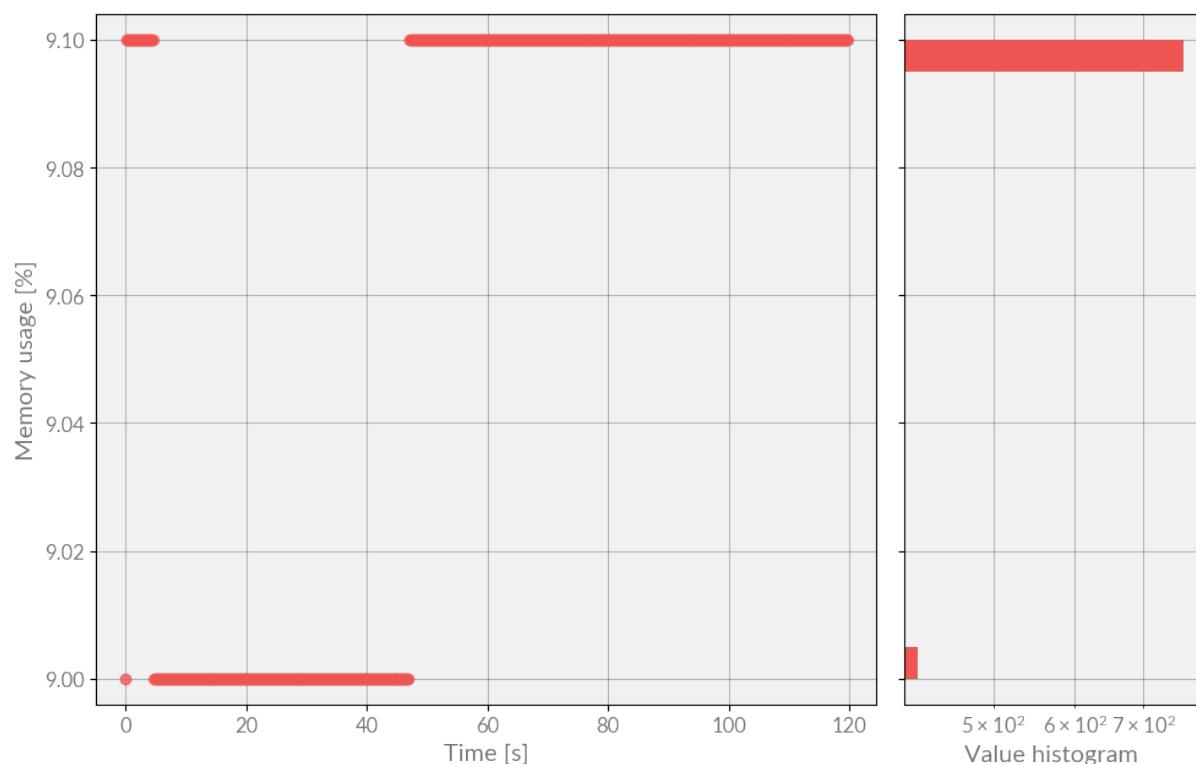


Figure 10.3: Memory usage during benchmark

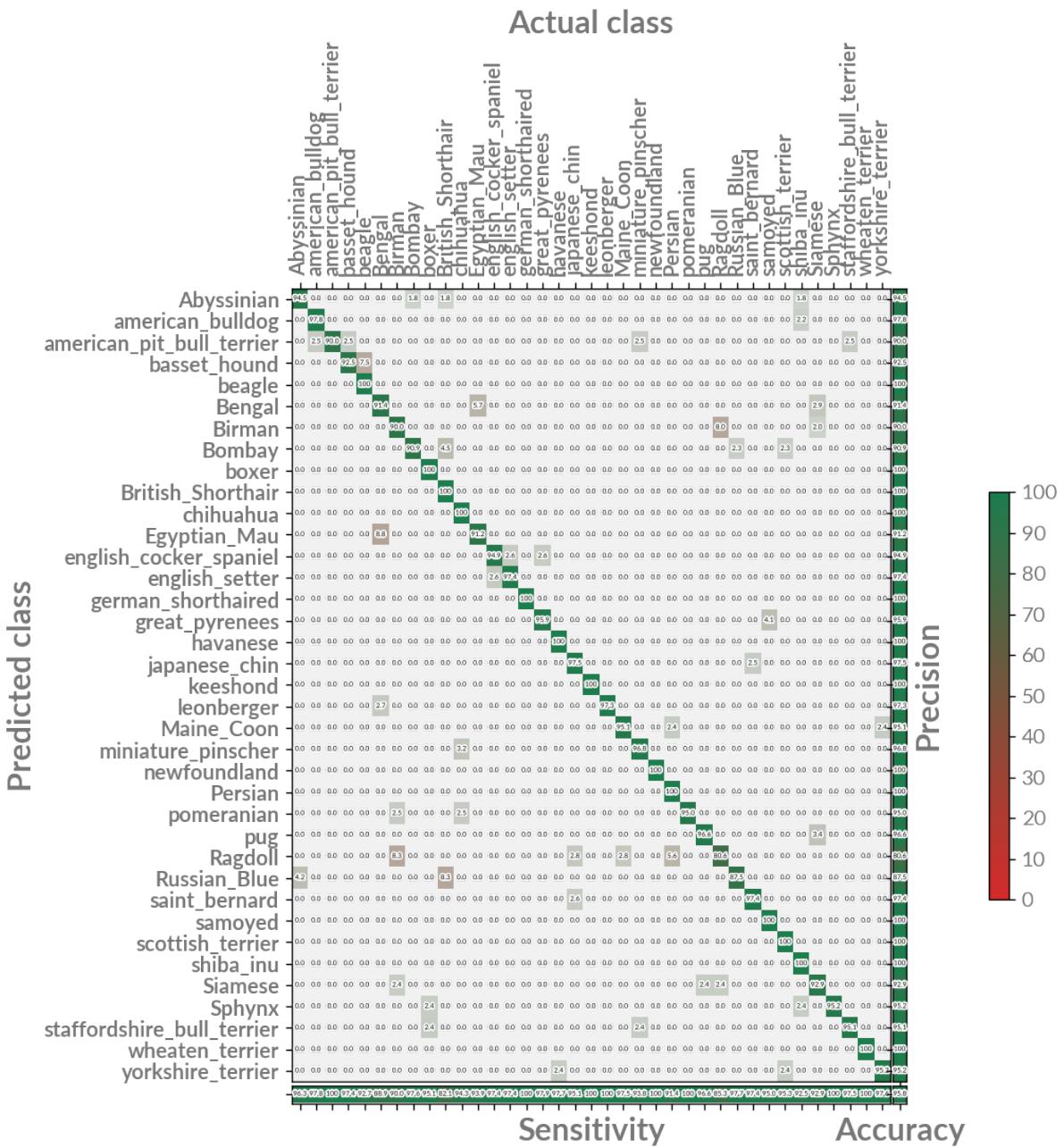


Figure 10.4: Confusion matrix

## 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.ONNRuntime",  
        "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](#).

## 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,
  - `str`,
  - `float`,
  - `int`,
  - `bool`,
- `default` - default value for the argument,
- `required` - boolean, tells if argument is required or not,
- `enum` - a list of possible values for the argument,
- `is_list` - tells if argument is a list of objects of types given in `type` field,
- `nullable` - tells if argument can be empty (`None`).

Let's add parameters to the example class:

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

(continues on next page)

(continued from previous page)

```

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):  

        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,

```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
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__(
        self,
        dataset: Dataset,
        compiled_model_path: Path,
        inferenceinputtype: str = 'float32',
        inferenceoutputtype: str = 'float32',
```

(continues on next page)

(continued from previous page)

```

        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

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

```

(continues on next page)

(continued from previous page)

```

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__(
        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

```

(continues on next page)

(continued from previous page)

```

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():
            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)

```

(continues on next page)

(continued from previous page)

```

# 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'
    )

# 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": [
        {
            "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",
    }
}
```

(continues on next page)

(continued from previous page)

```
        "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": {  
    "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]:
```

(continues on next page)

---

(continued from previous page)

```
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.

---

---

## 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.

## KENNING API

### 14.1 Deployment API overview

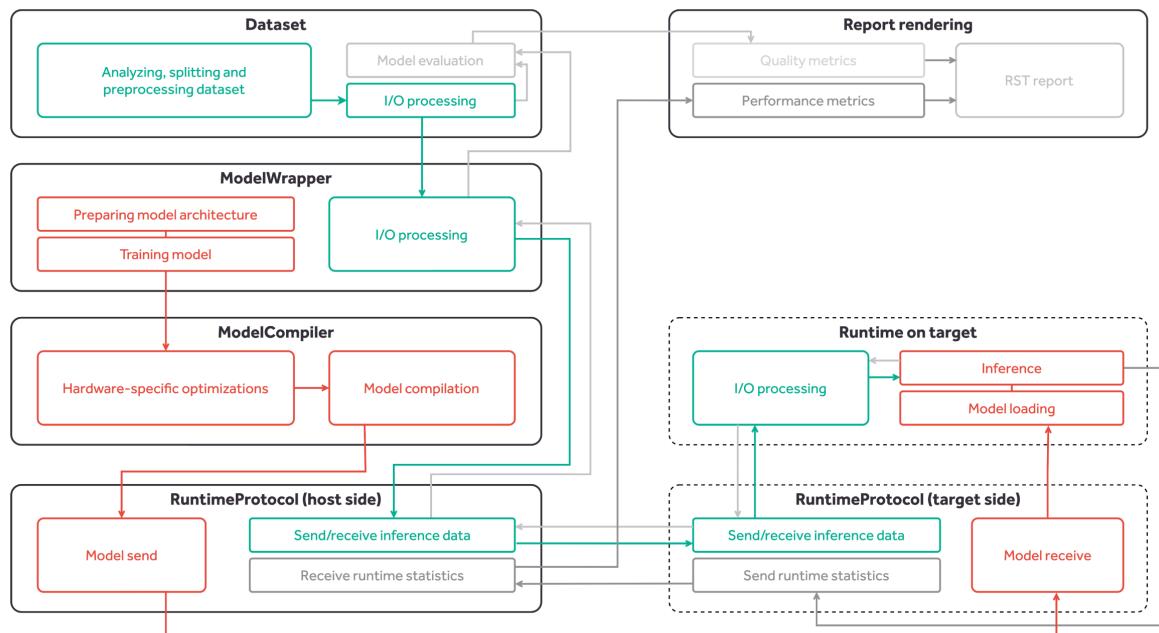


Figure 14.1: Kenning core classes and interactions between them. The green blocks represent the flow of input data passed to the model for inference. The orange blocks represent the flow of model deployment, from training to inference on target device. The grey blocks represent the inference results 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).

### 14.1.1 Model processing

The orange blocks and arrows in [Figure 14.1](#) 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*.

### 14.1.2 I/O data flow

The data flow is represented in the [Figure 14.1](#) 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 *Runtimes* also have their specific representations of data, proper I/O processing is applied.

### 14.1.3 Data flow reporting

Report rendering requires performance metrics and quality metrics. The flow for this is presented with grey lines and blocks in [Figure 14.1](#).

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](#).

## 14.2 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.

## 14.3 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

## 14.4 Dataset

`kenning.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)
```

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 = 12345) → Generator[list[Any], None, None]**

Creates generator for the calibration data.

**Parameters****percentage : float**

The fraction of data to use for calibration.

**seed : int**

The seed for random state.

**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 validation 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

## 14.5 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 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]

**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(batch_size: int, learning_rate: float, epochs: int, logdir: Path)
```

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.

#### Parameters

```
batch_size : int
```

The batch size for the training.

```
learning_rate : float
```

The learning rate for the training.

```
epochs : int
```

The number of epochs for training.

```
logdir : Path
```

Path to the logging directory.

#### Raises

**NotImplementedError** – Raised when method is not implemented

## 14.6 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')
```

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, 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]`

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) → 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.

**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]

**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\_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

**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]]]

**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.

## 14.7 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`

`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

**ModelErrorNotLoaded** : – 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

**ModelError** – 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

## 14.8 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.

### 14.8.1 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.

### 14.8.2 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
```

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() → Measurements**

Downloads inference statistics from the target device.

By default no statistics are gathered.

**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

## 14.9 DataConverter

`kenning.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

## 14.10 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.

```
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

## 14.11 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.

## 14.12 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` sub-module. 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

## 14.13 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`

## 14.14 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() → tuple[ArgumentParser, \_ArgumentGroup | None]

Creates argparse parser based on *arguments\_structure* of class and its all parent classes.

### 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]*

## 14.15 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.

## 14.16 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, [145](#)

# INDEX

## Symbols

_dataindex (kenning.core.dataset.Dataset attribute), 116	compile() (kenning.core.optimizer.Optimizer method), 128
A	consult_model_type() (kenning.core.optimizer.Optimizer method), 128
accumulate() (kenning.core.measurements.Measurements method), 146	convert_input_to_bytes() (kenning.core.model.ModelWrapper method), 122
add_custom_url_schemes() (kenning.utils.resource_manager.ResourceManager method), 154	convert_output_from_bytes() (kenning.core.model.ModelWrapper method), 123
add_entry() (kenning.core.onnxconversion.ONNXConversion method), 149	copy() (kenning.core.measurements.Measurements method), 146
add_measurement() (kenning.core.measurements.Measurements method), 146	D
ArgumentsHandler (class in kenning.utils.args_manager), 153	data (kenning.core.measurements.Measurements attribute), 146
B	DataConverter (class in kenning.core.dataconverter), 144
batch_size (kenning.core.dataset.Dataset attribute), 116	DataProvider (class in kenning.core.dataprovider), 151
C	Dataset (class in kenning.core.dataset), 115
calibration_dataset_generator() (kenning.core.dataset.Dataset method), 117	dataX (kenning.core.dataset.Dataset attribute), 115
check_conversions() (kenning.core.onnxconversion.ONNXConversion method), 150	dataXtest (kenning.core.dataset.Dataset attribute), 116
cleanup() (kenning.core.runner.Runner method), 113	dataXtrain (kenning.core.dataset.Dataset attribute), 116
clear() (kenning.core.measurements.Measurements method), 146	dataY (kenning.core.dataset.Dataset attribute), 115
clear() (kenning.core.measurements.MeasurementsCollector method), 147	dataYtest (kenning.core.dataset.Dataset attribute), 116
clear_cache() (kenning.utils.resource_manager.ResourceManager method), 154	dataYtrain (kenning.core.dataset.Dataset attribute), 116
	dataYval (kenning.core.dataset.Dataset attribute), 116
	derive_io_spec_from_json_params() (kenning.core.model.ModelWrapper class method), 123

<code>detach_from_output()</code>	( <i>kenning.core.outputcollector.OutputCollector method</i> ), 152	<code>from_argparse()</code>	( <i>kenning.core.runtime.Runtime method</i> ), 132	( <i>ken-class</i> )
<code>detach_from_source()</code>	( <i>kenning.core.dataprovider.DataProvider method</i> ), 151	<code>from_argparse()</code>	( <i>kenning.utils.args_manager.ArgumentsHandler class method</i> ), 153	( <i>ken-</i>
<code>disconnect()</code>	( <i>kenning.core.protocol.Protocol method</i> ), 139	<code>from_json()</code>	( <i>kenning.core.dataset.Dataset class method</i> ), 117	<i>ning.core.model.ModelWrapper class</i> )
<code>download_dataset_fun()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 117	<code>from_json()</code>	( <i>kenning.core.flow.KenningFlow class method</i> ), 112	<i>method</i> ), 124
<code>download_output()</code>	( <i>kenning.core.protocol.Protocol method</i> ), 139	<code>from_json()</code>	( <i>kenning.core.optimizer.Optimizer class method</i> ), 129	<i>method</i> ), 129
<code>download_statistics()</code>	( <i>kenning.core.protocol.Protocol method</i> ), 139	<code>from_json()</code>	( <i>kenning.core.protocol.Protocol class method</i> ), 139	<i>method</i> ), 132
<b>E</b>		<code>from_json()</code>	( <i>kenning.core.runner.Runner class method</i> ), 113	
<code>evaluate()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 117	<code>from_json()</code>	( <i>kenning.core.runtime.Runtime class method</i> ), 132	
<code>extract_output()</code>	( <i>kenning.core.runtime.Runtime method</i> ), 132	<code>from_json()</code>	( <i>kenning.utils.args_manager.ArgumentsHandler class method</i> ), 153	
<b>F</b>		<b>G</b>		
<code>fetch_input()</code>	( <i>kenning.core.dataprovider.DataProvider method</i> ), 151	<code>gather_data()</code>	( <i>kenning.core.protocol.Protocol method</i> ), 140	
<code>form_argparse()</code>	( <i>kenning.utils.args_manager.ArgumentsHandler class method</i> ), 153	<code>get_class_names()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 118	
<code>form_parameterschema()</code>	( <i>kenning.core.flow.KenningFlow method</i> ), 112	<code>get_data()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 118	
<code>form_parameterschema()</code>	( <i>kenning.utils.args_manager.ArgumentsHandler class method</i> ), 153	<code>get_data_unloaded()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 118	
<code>from_argparse()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 117	<code>get_framework_and_version()</code>	( <i>kenning.core.model.ModelWrapper method</i> ), 124	
<code>from_argparse()</code>	( <i>kenning.core.model.ModelWrapper method</i> ), 123	<code>get_framework_and_version()</code>	( <i>kenning.core.optimizer.Optimizer method</i> ), 129	
<code>from_argparse()</code>	( <i>kenning.core.optimizer.Optimizer method</i> ), 129	<code>get_input_formats()</code>	( <i>kenning.core.optimizer.Optimizer method</i> ), 129	
<code>from_argparse()</code>	( <i>kenning.core.protocol.Protocol method</i> ), 139	<code>get_input_formats()</code>	( <i>kenning.core.runtime.Runtime method</i> ), 132	
<code>from_argparse()</code>	( <i>kenning.core.runner.Runner class method</i> ), 113	<code>get_input_mean_std()</code>	( <i>kenning.core.dataset.Dataset method</i> ), 118	

get_input_type()	(ken-	initialize_server()	(ken-
<i>ning.core.optimizer.Optimizer method)</i> ,		<i>ning.core.protocol.Protocol method)</i> ,	
130		140	
get_io_spec_path()	(ken-	iter_test()	(kenning.core.dataset.Dataset
<i>ning.core.runtime.Runtime method)</i> ,		<i>method)</i> , 119	
132		iter_train()	(kenning.core.dataset.Dataset
get_io_specification()	(ken-	<i>method)</i> , 119	
<i>ning.core.model.ModelWrapper method)</i> , 124		iter_val()	(kenning.core.dataset.Dataset
get_io_specification_from_model()	(ken-	<i>method)</i> , 119	
<i>ning.core.model.ModelWrapper method)</i> , 124		K	
get_measurements()	(ken-	kenning.core.measurements	
<i>ning.core.measurements.SystemStatsCollector method)</i> , 148		module, 145	
get_output_formats()	(ken-	KenningFlow (class in kenning.core.flow), 112	
<i>ning.core.model.ModelWrapper method)</i> , 125		L	
get_output_formats()	(ken-	list_cached_files()	(ken-
<i>ning.core.optimizer.Optimizer method)</i> ,		<i>ning.utils.resource_manager.ResourceManager method)</i> , 155	
130		load_input()	(kenning.core.runtime.Runtime
get_path()	(ken-	<i>method)</i> , 133	
<i>ning.core.model.ModelWrapper method)</i> , 125		load_input_from_bytes()	(ken-
get_resource()	(ken-	<i>ning.core.runtime.Runtime method)</i> , 134	
<i>ning.utils.resource_manager.ResourceManager method)</i> , 154		load_io_specification()	(ken-
get_spec_path()	(ken-	<i>ning.core.optimizer.Optimizer method)</i> , 130	
<i>ning.core.optimizer.Optimizer static method)</i> , 130		load_model()	(ken-
get_time()	(kenning.core.runtime.Runtime	<i>ning.core.model.ModelWrapper method)</i> , 125	
<i>method)</i> , 133		M	
get_values()	(ken-	Measurements (class in ken-	
<i>ning.core.measurements.Measurements method)</i> , 147		<i>ning.core.measurements</i> ), 145	
I		MeasurementsCollector (class in ken-	
infer()	(kenning.core.runtime.Runtime	<i>ning.core.measurements</i> ), 147	
<i>method)</i> , 133		ModelWrapper (class in kenning.core.model),	
inference_session_end()	(ken-	122	
<i>ning.core.runtime.Runtime method)</i> ,		module	
133		kenning.core.measurements, 145	
inference_session_start()	(ken-	O	
<i>ning.core.runtime.Runtime method)</i> ,		onnx_export()	(ken-
133		<i>ning.core.onnxconversion.ONNXConversion method)</i> , 150	
initialize_client()	(ken-	onnx_import()	(ken-
<i>ning.core.protocol.Protocol method)</i> ,		<i>ning.core.onnxconversion.ONNXConversion method)</i> , 150	
140		ONNXConversion (class in ken-	
initialize_measurement()	(ken-	<i>ning.core.onnxconversion</i> ), 149	
<i>ning.core.measurements.Measurements method)</i> , 147		Optimizer (class in kenning.core.optimizer),	
		128	

origin ( <i>kenning.utils.resource_manager.ResourceURI</i> property), 156	process_output()	( <i>kenning.core.outputcollector.OutputCollector</i> method), 152
<b>P</b>		
parent ( <i>kenning.utils.resource_manager.ResourceURI</i> property), 156	read_io_specification()	( <i>kenning.core.runtime.Runtime</i> method), 135
parse_io_specification_from_json() ( <i>kenning.core.model.ModelWrapper</i> class method), 125	receive_confirmation()	( <i>kenning.core.protocol.Protocol</i> method), 140
postprocess_outputs() ( <i>kenning.core.model.ModelWrapper</i> method), 126	receive_data()	( <i>kenning.core.protocol.Protocol</i> method), 140
prepare() ( <i>kenning.core.dataprovider.DataProvider</i> method), 151	receive_message()	( <i>kenning.core.protocol.Protocol</i> method), 141
prepare() ( <i>kenning.core.dataset.Dataset</i> method), 119	request_failure()	( <i>kenning.core.protocol.Protocol</i> method), 141
prepare() ( <i>kenning.core.onnxconversion.ONNXConversion</i> method), 151	request_optimization()	( <i>kenning.core.protocol.Protocol</i> method), 141
prepare_external_calibration_dataset() ( <i>kenning.core.dataset.Dataset</i> method), 119	request_processing()	( <i>kenning.core.protocol.Protocol</i> method), 142
prepare_input_samples() ( <i>kenning.core.dataset.Dataset</i> method), 120	request_success()	( <i>kenning.core.protocol.Protocol</i> method), 142
prepare_io_specification() ( <i>kenning.core.runtime.Runtime</i> method), 134	ResourceManager	( <i>class in kenning.utils.resource_manager</i> ), 154
prepare_local() ( <i>kenning.core.runtime.Runtime</i> method), 134	ResourceURI	( <i>class in kenning.utils.resource_manager</i> ), 156
prepare_model() ( <i>kenning.core.model.ModelWrapper</i> method), 126	run()	( <i>kenning.core.flow.KenningFlow</i> method), 113
prepare_model() ( <i>kenning.core.runtime.Runtime</i> method), 135	run()	( <i>kenning.core.measurements.SystemStatsCollector</i> method), 148
prepare_output_samples() ( <i>kenning.core.dataset.Dataset</i> method), 120	run()	( <i>kenning.core.runner.Runner</i> method), 114
preprocess_input() ( <i>kenning.core.dataprovider.DataProvider</i> method), 151	run()	( <i>kenning.core.runtime.Runtime</i> method), 136
preprocess_input() ( <i>kenning.core.model.ModelWrapper</i> method), 126	run_inference()	( <i>kenning.core.model.ModelWrapper</i> method), 126
preprocess_model_to_upload() ( <i>kenning.core.runtime.Runtime</i> method),	run_single_step()	( <i>kenning.core.flow.KenningFlow</i> method), 113
	Runner	( <i>class in kenning.core.runner</i> ), 113
	Runtime	( <i>class in kenning.core.runtime</i> ), 132

## S

save\_dataset\_checksum() (ken-  
*ning.core.dataset.Dataset* method), 120  
 save\_io\_specification() (ken-  
*ning.core.optimizer.Optimizer* method), 131  
 save\_measurements() (ken-  
*ning.core.measurements.Measurements* class method), 147  
 save\_model() (ken-  
*ning.core.model.ModelWrapper* method), 127  
 save\_to\_onnx() (ken-  
*ning.core.model.ModelWrapper* method), 127  
 send\_data() (kenning.core.protocol.Protocol method), 142  
 send\_message() (ken-  
*ning.core.protocol.Protocol* method), 143  
 set\_batch\_size() (ken-  
*ning.core.dataset.Dataset* method), 120  
 set\_cache\_dir() (ken-  
*ning.utils.resource\_manager.ResourceManager* method), 155  
 set\_compiled\_model\_path() (ken-  
*ning.core.optimizer.Optimizer* method), 131  
 set\_input\_type() (ken-  
*ning.core.optimizer.Optimizer* method), 131  
 set\_max\_cache\_size() (ken-  
*ning.utils.resource\_manager.ResourceManager* method), 155  
 should\_close() (ken-  
*ning.core.outputcollector.OutputCollector* method), 152  
 should\_close() (kenning.core.runner.Runner method), 114  
 SystemStatsCollector (class in ken-  
*ning.core.measurements*), 147  
 systemstatsmeasurements() (in module ken-  
*ning.core.measurements*), 148

## T

tagmeasurements() (in module ken-  
*ning.core.measurements*), 149  
 test\_inference() (ken-  
*ning.core.model.ModelWrapper*

*method*), 127  
 test\_subset\_len() (ken-  
*ning.core.dataset.Dataset* method), 120  
 timemeasurements() (in module ken-  
*ning.core.measurements*), 149  
 to\_json() (ken-  
*ning.utils.args\_manager.ArgumentsHandler* method), 154  
 to\_next\_block() (ken-  
*ning.core.dataconverter.DataConverter* method), 145  
 to\_previous\_block() (ken-  
*ning.core.dataconverter.DataConverter* method), 145  
 train\_model() (ken-  
*ning.core.model.ModelWrapper* method), 127  
 train\_subset\_len() (ken-  
*ning.core.dataset.Dataset* method), 121  
 train\_test\_split\_representations() (ken-  
*ning.core.dataset.Dataset* method), 121

**U**

update\_measurements() (ken-  
*ning.core.measurements.Measurements* method), 147  
 upload\_input() (ken-  
*ning.core.protocol.Protocol* method), 143  
 upload\_io\_specification() (ken-  
*ning.core.protocol.Protocol* method), 143  
 upload\_model() (ken-  
*ning.core.protocol.Protocol* method), 143  
 upload\_optimizers() (ken-  
*ning.core.protocol.Protocol* method), 144  
 upload\_output() (ken-  
*ning.core.runtime.Runtime* method), 136  
 upload\_runtime() (ken-  
*ning.core.protocol.Protocol* method), 144  
 upload\_stats() (ken-  
*ning.core.runtime.Runtime* method), 136  
 uri (kenning.utils.resource\_manager.ResourceURI

---

*property)*, 156

## V

val\_subset\_len() (ken-  
ning.core.dataset.Dataset method),  
121  
validate\_resources\_version() (ken-  
ning.utils.resource\_manager.ResourceManager  
method), 155  
verify\_dataset\_checksum() (ken-  
ning.core.dataset.Dataset method),  
121

## W

with\_name() (ken-  
ning.utils.resource\_manager.ResourceURI  
method), 156  
with\_stem() (ken-  
ning.utils.resource\_manager.ResourceURI  
method), 156  
with\_suffix() (ken-  
ning.utils.resource\_manager.ResourceURI  
method), 156