Antmicro

Kenning

2023-04-14

CONTENTS

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:

- The *Kenning* section provides a project description, installation steps and a quick start guide,
- The *Deep Learning deployment stack* section describes a typical model deployment flow on edge devices,
- The *Defining optimization pipelines in Kenning* section describes a way to create advanced optimization scenarios with JSON config,
- The *Using Kenning via command line arguments* section describes executable scripts available in |projecturl|,
- The *Kenning measurements* section describes data gathered during the compilation and evaluation process,
- The *ONNX support in deep learning frameworks* section covers ONNX model format support in deep learning frameworks,
- The *Sample autogenerated report* section provides a sample report generated using Kenning,
- The *Creating applications with Kenning* section describes KenningFlow and its usage,
- The *Developing Kenning blocks* section describes a way to develop new Kenning components,
- The *Kenning API* section provides an in-depth description of the Kenning API.

CHAPTER

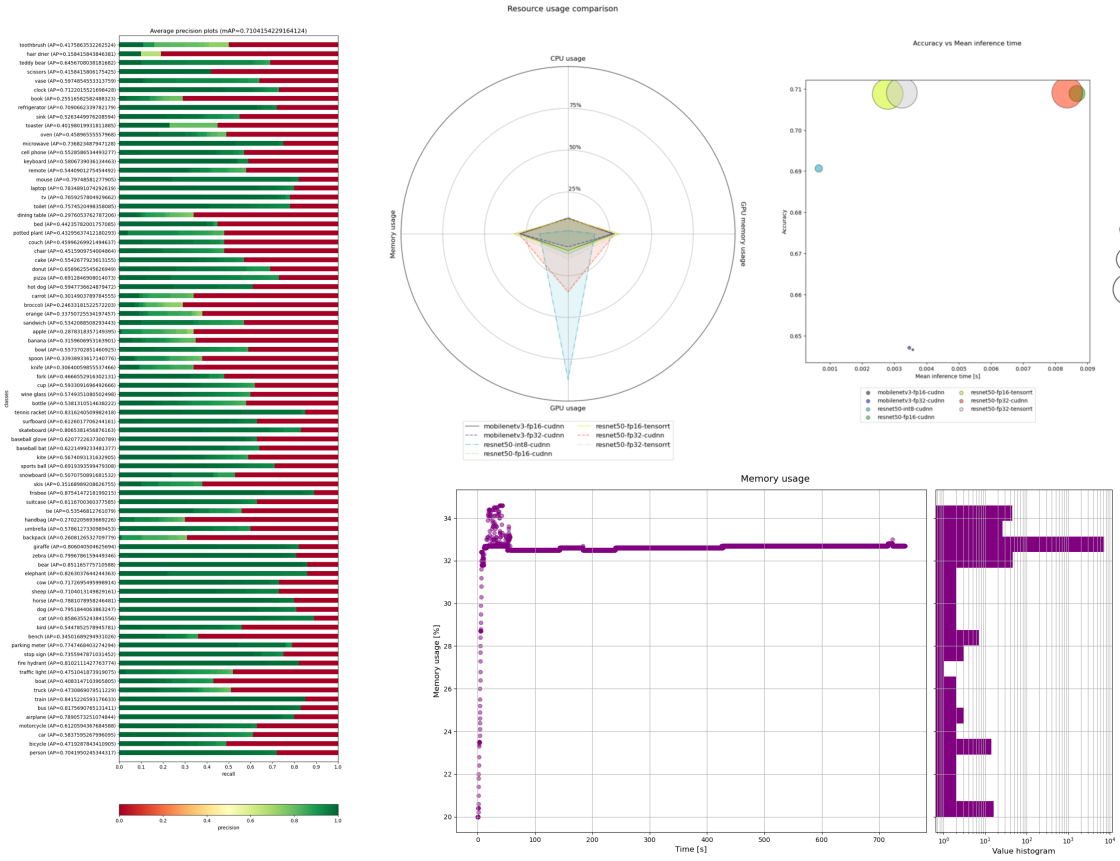
TWO

KENNING

Copyright (c) 2020-2023 Antmicro

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

Kenning documentation | Core API | kenning.ai



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 *Example use case of Kenning*.

2.1 Kenning installation

2.1.1 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,
- `iree` - modules for IREE compilation and runtime,

- `tvm` - modules for Apache TVM compilation and runtime,
- `onnxruntime` - modules for ONNX Runtime,
- `docs` - modules for generating documentation,
- `test` - modules for Kenning framework testing,
- `real_time_visualization` - modules for real time visualization runners.

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

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

2.1.2 Working directly with the repository

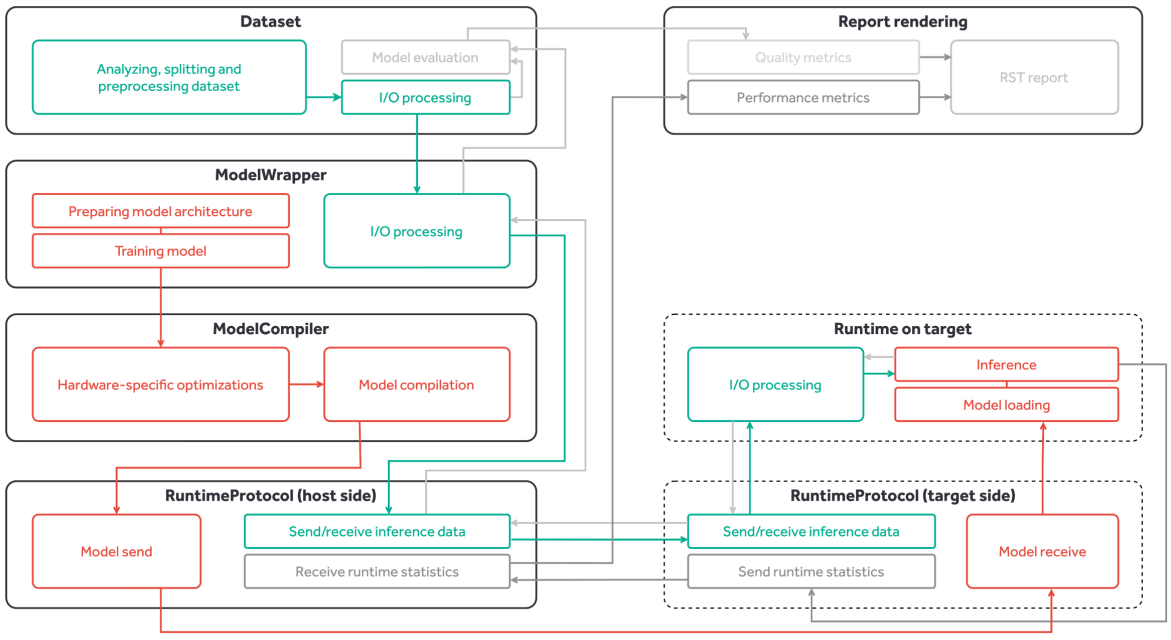
For development purposes, and for usage of additional resources (as sample scripts or trained models), clone repository with:

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

To download model weights, install Git Large File Storage (if not installed) and run:

```
cd kenning/
git lfs pull
```

2.2 Kenning structure



The `kenning` module consists of the following submodules:

- `core` - provides interface APIs for datasets, models, optimizers, runtimes and runtime protocols,
- `datasets` - provides implementations for datasets,
- `modelwrappers` - provides implementations for models for various problems implemented in various frameworks,
- `compilers` - 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,
- `runtimeprotocols` - 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,
- `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,
- `report` - provides methods for rendering reports,
- `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.

`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,
- `RuntimeProtocol` 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),
- `Runner` class - represents single runtime processing block.

2.3 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 for:

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

For more details on each of the above scenarios, check the Kenning documentation.

2.4 Example use case of Kenning

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

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

We will skip the training process. The trained model can be found in `kenning/resources/models/classification/tensorflow_pet_dataset_mobilenetv2.h5`.

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

```
python -m kenning.scenarios.model_training \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    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
```

2.4.1 Benchmarking a model using a native framework

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

For this, we will use the `json_inference_tester` scenario. Scenarios are configured using JSON format files. In our case, the JSON file (named `native.json`) will look like this:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/models/classification/tensorflow_
↪pet_dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
        {
            "dataset_root": "./build/pet-dataset",
            "download_dataset": true
        }
    }
}
```

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

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

In `model_wrapper`, we specify the model used for evaluation - here it is MobileNetV2 trained on Pet Dataset. 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 Pet Dataset - here we specify that we want to download the dataset (`download_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 above config saved in the `native.json` file, run the `json_inference_tester` scenario:

```
python -m kenning.scenarios.json_inference_tester native.json build/native.json --
↪verbosity INFO
```

This module 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 below configurations can be executed with this command.

To visualize the evaluation and benchmark results, run the `render_report` module:

```
python -m kenning.scenarios.render_report 'native' build/benchmarks/native.md --
↪measurements build/native.json --root-dir build/benchmarks --img-dir build/
↪benchmarks/imgs --verbosity INFO --report-types performance classification
```

This module takes the output JSON file generated by the `json_inference_tester` module, 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 `peformance` and `classification` metrics sections in the report (for example, there is also a `detection` report type for object detection tasks).

In `build/benchmarks/imgs` there will be 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 - apart from linking to the generated visualizations, it provides aggregated information about CPU and memory usage, as well as classification quality metrics, such as accuracy, sensitivity, precision or G-Mean. Such file can be included in a larger, Sphinx-based documentation, which allows easy, automated report generation, using e.g. CI, as can be seen in the Kenning documentation.

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.

2.4.2 Optimizing a model using TensorFlow Lite

One of such frameworks is TensorFlow Lite - a lightweight library for inferring networks on edge - it has a small binary size (which can be even more reduced 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 TensorFlow Lite Runtime that will execute the model:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/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/pet-dataset",
            "download_dataset": false
        }
    },
    "optimizers":
    [
        {
            "type": "kenning.compilers.tflite.TFLiteCompiler",
            "parameters":
            {
                "target": "default",
                "compiled_model_path": "./build/fp32.tflite",
                "inference_input_type": "float32",
                "inference_output_type": "float32"
            }
        }
    ],
    "runtime":
    {
        "type": "kenning.runtimes.tflite.TFLiteRuntime",
        "parameters":
        {
            "save_model_path": "./build/fp32.tflite"
        }
    }
}
```

In the already existing blocks, we only disable dataset download - the `download_dataset` parameter can be also removed, since the dataset is not downloaded by default.

The first new addition is the presence of the `optimizers` list - it 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, `default` is the regular CPU. 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.

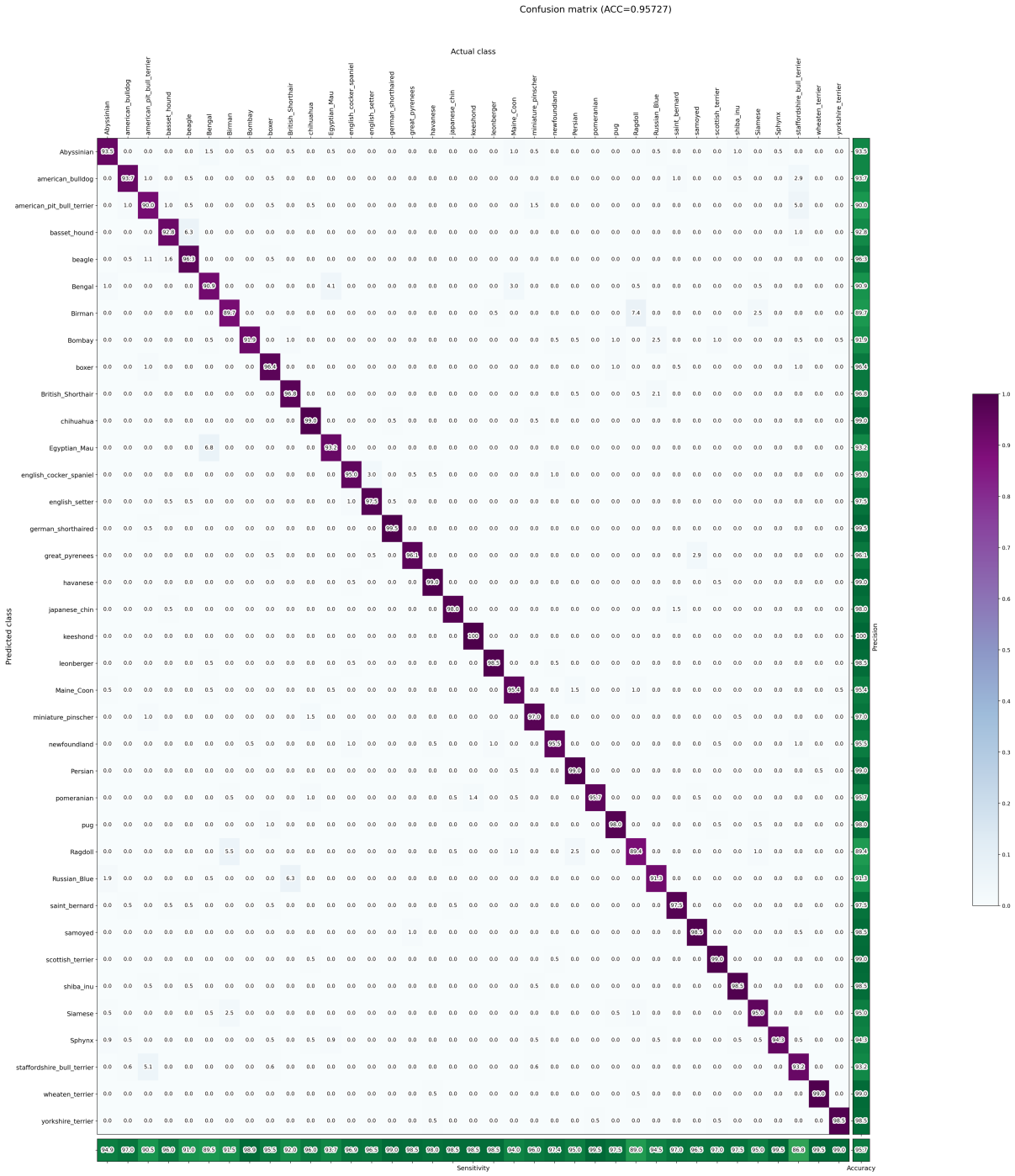
The second thing that is added to the previous flow is the `runtime` block - it provides a class inheriting from the `kenning.core.runtime.Runtime` class that is able to load the final model and run inference on target hardware. Usually, each `Optimizer` has a corresponding `Runtime` able to run its results.

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

```
python -m kenning.scenarios.json_inference_tester tflite-fp32.json build/tflite-
↪fp32.json --verbosity INFO

python -m kenning.scenarios.render_report 'tflite-fp32' build/benchmarks/tflite-
↪fp32.md --measurements build/tflite-fp32.json --root-dir build/benchmarks --img-
↪dir build/benchmarks/imgs --verbosity INFO --report-types performance␣
↪classification
```

While it depends on the platform used, there should be 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.



2.4.3 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 the `INT8` precision, instead of the `FP32` precision. While it may severely harm the quality of the predictions, 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:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/models/classification/tensorflow_
↪pet_dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
        {
            "dataset_root": "./build/pet-dataset"
        }
    },
    "optimizers":
    [
        {
            "type": "kenning.compilers.tflite.TFLiteCompiler",
            "parameters":
            {
                "target": "int8",
                "compiled_model_path": "./build/int8.tflite",
                "inference_input_type": "int8",
                "inference_output_type": "int8"
            }
        }
    ],
    "runtime":
    {
        "type": "kenning.runtimes.tflite.TFLiteRuntime",
        "parameters":
        {
            "save_model_path": "./build/int8.tflite"
        }
    }
}
```

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 so the entire model calibration process happens automatically.

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

```
python -m kenning.scenarios.json_inference_tester tflite-int8.json build/tflite-
↪int8.json --verbosity INFO

python -m kenning.scenarios.render_report 'tflite-int8' build/benchmarks/tflite-
↪int8.md --measurements build/tflite-int8.json --root-dir build/benchmarks --img-
↪dir build/benchmarks/imgs --verbosity INFO --report-types performance␣
↪classification
```

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

2.4.4 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. For this, let's use the Apache TVM framework to compile efficient runtimes for various hardware platforms. The scenario looks like this:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/models/classification/tensorflow_
↪pet_dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
        {
            "dataset_root": "./build/pet-dataset"
        }
    },
    "optimizers":
    [
        {
            "type": "kenning.compilers.tflite.TFLiteCompiler",
            "parameters":
            {
                "target": "int8",
                "compiled_model_path": "./build/int8.tflite",
                "inference_input_type": "int8",
                "inference_output_type": "int8"
            }
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "type": "kenning.compilers.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 it can be observed, addition of a new framework is just a matter of simply adding and configuring another optimizer and using the corresponding `Runtime` 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`):

```
python -m kenning.scenarios.json_inference_tester tvm-avx2-int8.json build/tvm-
↪avx2-int8.json --verbosity INFO

python -m kenning.scenarios.render_report 'tvm-avx2-int8' build/benchmarks/tvm-
↪avx2-int8.md --measurements build/tvm-avx2-int8.json --root-dir build/
↪benchmarks --img-dir build/benchmarks/imgs --verbosity INFO --report-types␣
↪performance classification
```

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

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

The summary of passes can be seen below:

	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

2.4.5 Automated model comparison

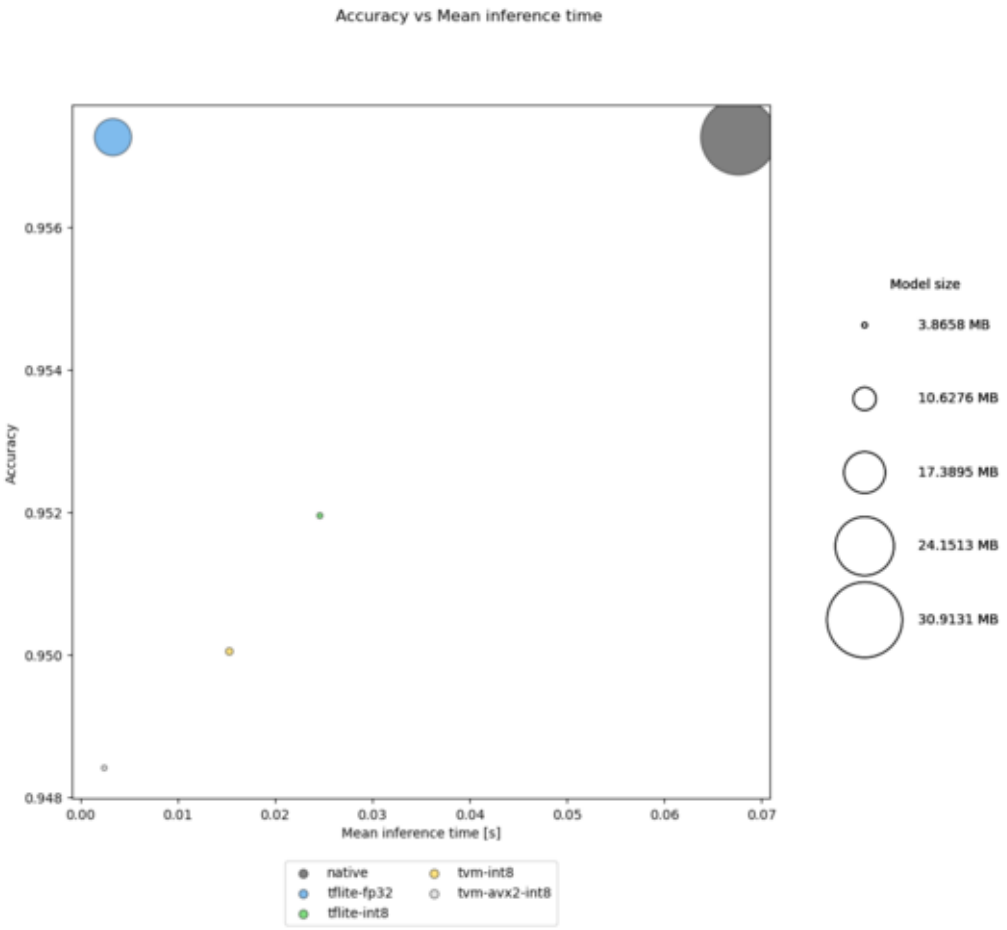
The `kenning.scenarios.render_report` script also allows us to compare evaluation results for multiple models. Apart from creating a table with a summary of models, it also creates plots aggregating measurements collected during the evaluation process.

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

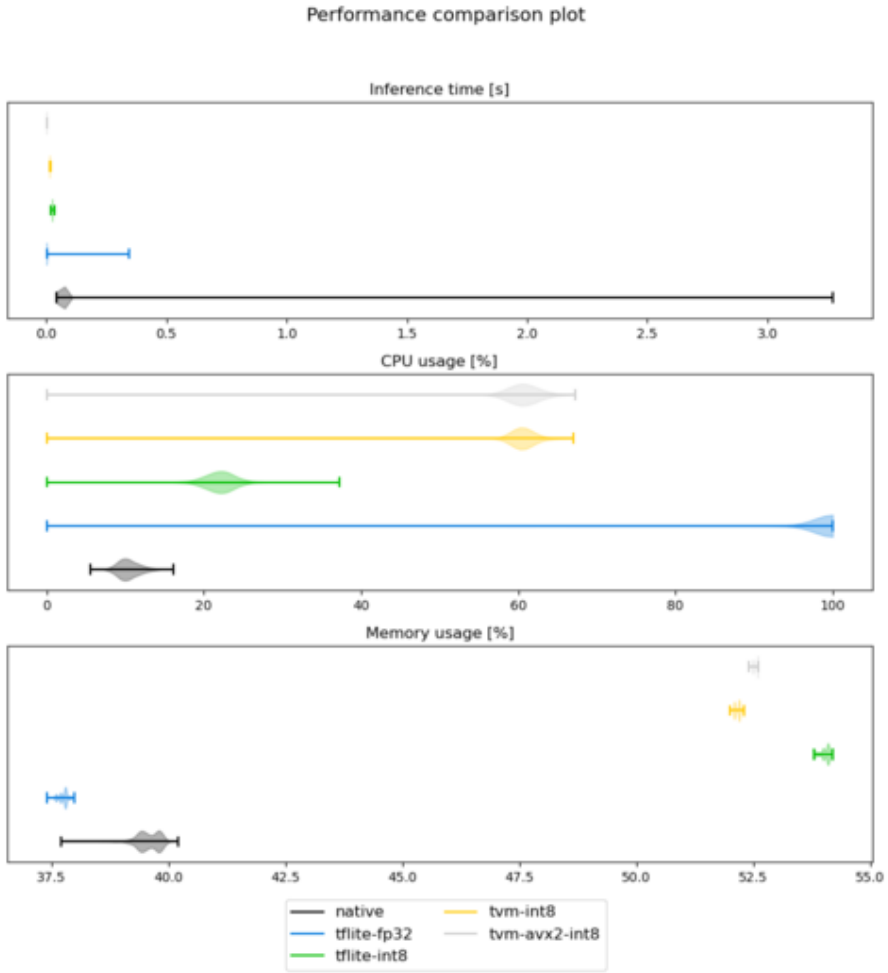
```
python -m kenning.scenarios.render_report "summary-report"
    build/benchmarks/summary.md \
    --measurements \
        build/native.json \
        build/tflite-fp32.json \
        build/tflite-int8.json \
        build/tvm-avx2-int8.json \
    --root-dir build/benchmarks \
    --report-types performance classification \
    --img-dir build/benchmarks/imgs \
    --model-names \
        native \
        tflite-fp32 \
        tflite-int8 \
        tvm-avx2-int8
```

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

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 kenning.datasets.pet_dataset import PetDataset
from kenning.modelwrappers.classification.tensorflow_pet_dataset import␣
↪TensorFlowPetDatasetMobileNetV2
from kenning.compilers.tflite import TFLiteCompiler
from kenning.runtimes.tflite import TFLiteRuntime
from kenning.core.measurements import MeasurementsCollector
```

(continues on next page)

(continued from previous page)

```
dataset = PetDataset(
    root='./build/pet-dataset/',
    download_dataset=True
)
model = TensorFlowPetDatasetMobileNetV2(
    modelpath='./kenning/resources/models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5',
    dataset=dataset
)
model.save_io_specification(model.modelpath)
compiler = TFLiteCompiler(
    dataset=dataset,
    compiled_model_path='./build/compiled-model.tflite',
    modelframework='keras',
    target='default',
    inferenceinputtype='float32',
    inferenceoutputtype='float32'
)
compiler.compile(
    inputmodelpath='./kenning/resources/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='./build/compiled-model.tflite',
    modelframework='keras',
    target='int8',
    inferenceinputtype='int8',
    inferenceoutputtype='int8',
    dataset_percentage=0.3
)
compiler.compile(
    inputmodelpath='./kenning/resources/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,
    modelpath='./build/compiled-model.tflite'
)
```

(continues on next page)

(continued from previous page)

```
runtime.run_locally(
    dataset,
    model,
    './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`, `RuntimeProtocol`, `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 overriden, if neccessary.

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

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

CHAPTER

THREE

DEEP LEARNING DEPLOYMENT STACK

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

3.1 From training to deployment

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

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

3.2 Dataset preparation

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

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

- training dataset - the largest subset that is used to train a model,
- validation dataset - a relatively small set that is used to verify model performance after each training epoch (the metrics and loss function values show if any overfitting occured 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 seleceted, 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 siginificant loss in a model's quality.

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

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

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

3.5 Model compilation and deployment

Deep learning compilers can transform model representation to:

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

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

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

CHAPTER

FOUR

DEFINING OPTIMIZATION PIPELINES IN KENNING

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

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

4.1 JSON specification

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

- `model_wrapper` - **mandatory field**, accepts dictionary as a value that defines the *ModelWrapper* object for the deployed model (provides I/O processing, optionally model).
- `dataset` - **mandatory field**, accepts dictionary as a value that defines the *Dataset* object for model optimization and evaluation.
- `optimizers` - *optional field*, accepts a list of dictionaries specifying the sequence of *Optimizer*-based optimizations applied to the model.
- `runtime_protocol` - *optional field*, defines the *RuntimeProtocol* 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:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/models/classification/tensorflow_
↪pet_dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
        {
            "dataset_root": "./build/pet-dataset",
            "download_dataset": true
        }
    }
}
```

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:

```
python -m kenning.scenarios.json_inference_tester pipeline.json measurements.json␣
↪--verbosity INFO
```

The `measurements.json` file is the output of the `kenning.scenarios.json_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 is the following scenario:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/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/pet-dataset"
        }
    },
    "optimizers":
    [
        {
            "type": "kenning.compilers.tflite.TFLiteCompiler",
            "parameters":
            {
                "target": "int8",
                "compiled_model_path": "./build/int8.tflite",
                "inference_input_type": "int8",
                "inference_output_type": "int8"
            }
        },
        {
            "type": "kenning.compilers.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.compilers.tflite.TFLiteCompiler` type block, quantizing the model,
- a `kenning.compilers.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.json_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:

```
python -m kenning.scenarios.json_inference_tester scenario.json 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 *RuntimeProtocol* and the `kenning.scenarios.json_inference_server` scenario.

For this use case, we need two JSON files - one for inference server configuration, and another one for the `kenning.scenarios.json_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 *RuntimeProtocol* used.

Let's start with client configuration by adding a `runtime_protocol` entry:

```
{
    "model_wrapper":
    {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters":
        {
            "model_path": "./kenning/resources/models/classification/tensorflow_
↪pet_dataset_mobilenetv2.h5"
        }
    },
    "dataset":
    {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters":
```

(continues on next page)

(continued from previous page)

```
        {
            "dataset_root": "./build/pet-dataset"
        }
    },
    "optimizers":
    [
        {
            "type": "kenning.compilers.tflite.TFLiteCompiler",
            "parameters":
            {
                "target": "int8",
                "compiled_model_path": "./build/int8.tflite",
                "inference_input_type": "int8",
                "inference_output_type": "int8"
            }
        },
        {
            "type": "kenning.compilers.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"
        }
    },
    "runtime_protocol":
    {
        "type": "kenning.runtimeprotocols.network.NetworkProtocol",
        "parameters":
        {
            "host": "10.9.8.7",
            "port": 12345,
            "packet_size": 32768
        }
    }
}
```

In the `runtime_protocol` entry, we specify a `kenning.runtimeprotocols.network.NetworkProtocol` and provide a server address (`host`), an application port (`port`) and packet size (`packet_size`). The `runtime` block is still needed to perform runtime-specific data preprocessing and postprocessing in the client application (the server only infers data).

The server configuration looks as follows:

```
{
    "runtime":
    {
        "type": "kenning.runtimes.tvm.TVMRuntime",
        "parameters":
        {
            "save_model_path": "./build/compiled_model_server.tar"
        }
    },
    "runtime_protocol":
    {
        "type": "kenning.runtimeprotocols.network.NetworkProtocol",
        "parameters":
        {
            "host": "0.0.0.0",
            "port": 12345,
            "packet_size": 32768
        }
    }
}
```

Here, only `runtime` and `runtime_protocol` need to be specified. The server uses `runtime_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:

```
python3 -m kenning.scenarios.json_inference_server \
    ./scripts/jsonconfigs/tflite-tvm-classification-server.json \
    --verbosity INFO
```

Then, run the client:

```
python3 -m kenning.scenarios.json_inference_tester \
    ./scripts/jsonconfigs/tflite-tvm-classification-client.json \
    ./build/tflite-tvm-classificationjson.json \
    --verbosity INFO
```

The rest of the flow is automated.

CHAPTER

FIVE

USING KENNING VIA COMMAND LINE ARGUMENTS

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

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

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 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:

```
python -m kenning.scenarios.model_training \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    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:

```
positional arguments:
  modelwrappercls       ModelWrapper-based class with inference implementation to␣
↪import
```

(continues on next page)

(continued from previous page)

```
  datasetcls            Dataset-based class with dataset to import

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

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

Dataset arguments:
  --dataset-root DATASET_ROOT
                        Path to the dataset directory
  --download-dataset    Downloads the dataset before taking any action
  --inference-batch-size INFERENCE_BATCH_SIZE
                        The batch size for providing the input data
  --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
```

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

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

```
python -m kenning.scenarios.model_training \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    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`.

5.3 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:

```
python -m kenning.scenarios.inference_performance \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    kenning.datasets.pet_dataset.PetDataset \
    build/tensorflow_pet_dataset_mobilenetv2.json \
    --model-path kenning/resources/models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5 \
    --dataset-root build/pet-dataset/ \
    --download-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*.

5.4 ONNX conversion

`kenning.scenarios.onnx_conversion` empirically tests the ONNX conversion for various frameworks and generates a report containing a support matrix. The matrix tells us if model export to ONNX and model import from ONNX for a given framework and model are supported or not. The example report with the command call is available in *ONNX support in deep learning frameworks*.

`kenning.scenarios.onnx_conversion` requires a list of *ONNXConversion* classes that implement model providers and a conversion method. For the below, call:

```
python -m kenning.scenarios.onnx_conversion \
    build/models-directory \
    build/onnx-support.rst \
    --converters-list \
        kenning.onnxconverters.pytorch.PyTorchONNXConversion \
        kenning.onnxconverters.tensorflow.TensorFlowONNXConversion \
        kenning.onnxconverters.mxnet.MXNetONNXConversion
```

The conversion is tested for three frameworks - PyTorch, TensorFlow and MXNet. The successfully converted ONNX models are stored in the `build/models-directory`. The final RST file with the report is stored in the `build/onnx-support.rst` directory.

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 *RuntimeProtocol*.

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 *RuntimeProtocol*-based class when running remotely to communicate with the `kenning.scenarios.inference_server`.

To print the list of required arguments, run:

```
python3 -m kenning.scenarios.inference_tester \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    kenning.runtimes.tvm.TVMRuntime \
    kenning.datasets.pet_dataset.PetDataset \
    --modelcompiler-cls kenning.compilers.tvm.TVMCompiler \
    --protocol-cls kenning.runtimeprotocols.network.NetworkProtocol \
    -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
  datasetcls            Dataset-based class with dataset to import
  output                The path to the output JSON file with measurements

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --modelcompiler-cls MODELCOMPILER_CLS
                        Optimizer-based class with compiling routines to import
  --protocol-cls PROTOCOL_CLS
                        RuntimeProtocol-based class with the implementation of␣
↪communication between inference tester and inference
                        runner
  --convert-to-onnx CONVERT_TO_ONNX
                        Before compiling the model, convert it to ONNX and use in␣
↪compilation (provide a path to save here)
  --verbosity {DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        Verbosity level

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

Compiler arguments:
  --compiled-model-path COMPILED_MODEL_PATH
                        The path to the compiled model output
  --model-framework {onnx,keras,darknet}
                        The input type of the model, framework-wise
  --target TARGET       The kind or tag of the target device
  --target-host TARGET_HOST
                        The kind or tag of the host (CPU) target device
  --opt-level OPT_LEVEL
                        The optimization level of the compilation
  --libdarknet-path LIBDARKNET_PATH
                        Path to the libdarknet.so library, for darknet models

Runtime arguments:
  --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

Dataset arguments:
  --dataset-root DATASET_ROOT
                        Path to the dataset directory
  --download-dataset    Downloads the dataset before taking any action
  --inference-batch-size INFERENCE_BATCH_SIZE
                        The batch size for providing the input data
  --classify-by {species,breeds}
                        Determines if classification should be performed by␣
↪species or by breeds
```

(continues on next page)

(continued from previous page)

```
  --image-memory-layout {NHWC,NCHW}
                        Determines if images should be delivered in NHWC or NCHW␣
↪format

Runtime protocol arguments:
  --host HOST           The address to the target device
  --port PORT           The port for the target device
  --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 *RuntimeProtocol*-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:

```
python -m kenning.scenarios.inference_tester \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    kenning.runtimes.tflite.TFLiteRuntime \
    kenning.datasets.pet_dataset.PetDataset \
    ./build/google-coral-devboard-tflite-tensorflow.json \
    --modelcompiler-cls kenning.compilers.tflite.TFLiteCompiler \
    --protocol-cls kenning.runtimeprotocols.network.NetworkProtocol \
    --model-path ./kenning/resources/models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5 \
    --model-framework keras \
    --target "edgetpu" \
    --compiled-model-path build/compiled-model.tflite \
    --inference-input-type int8 \
    --inference-output-type int8 \
    --host 192.168.188.35 \
    --port 12345 \
    --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:

```
python -m kenning.scenarios.inference_server \
    kenning.runtimeprotocols.network.NetworkProtocol \
    kenning.runtimes.tflite.TFLiteRuntime \
    --host 0.0.0.0 \
```

(continues on next page)

(continued from previous page)

```
    --port 12345 \
    --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:

```
python3 -m kenning.scenarios.inference_tester \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    kenning.runtimes.tvm.TVMRuntime \
    kenning.datasets.pet_dataset.PetDataset \
    ./build/local-cpu-tvm-tensorflow-classification.json \
    --modelcompiler-cls kenning.compilers.tvm.TVMCompiler \
    --model-path ./kenning/resources/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 \
    --download-dataset \
    --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.detectors.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␣
↪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:
```

(continues on next page)

(continued from previous page)

```
  --video-file-path VIDEO_FILE_PATH
                        Video file path (for cameras, use /dev/videoX where X is␣
↪the device ID eg. /dev/video0)
  --image-memory-layout {NHWC,NCHW}
                        Determines if images should be delivered in NHWC or NCHW␣
↪format
  --image-width IMAGE_WIDTH
                        Determines the width of the image for the model
  --image-height IMAGE_HEIGHT
                        Determines the height of the image for the model
OutputCollector arguments:
  --print-type {detector,classificator}
                        What is the type of model that will input data to the␣
↪NamePrinter
```

An example script for `inference_runner`:

```
python3 -m kenning.scenarios.inference_runner \
    kenning.modelwrappers.detectors.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/resources/models/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.

CHAPTER

SIX

KENNING MEASUREMENTS

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

They contains 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.

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

CHAPTER

SEVEN

ONNX SUPPORT IN DEEP LEARNING FRAMEWORKS

ONNX is an open format created to represent machine learning models. The ONNX format is frequently updated with new operators that are present in state-of-the-art models.

Most of the frameworks for training, compiling and optimizing deep learning algorithms support ONNX format. It allows conversion of models from one representation to another.

The `kenning.core.onnxconversion.ONNXConversion` class provides an API for writing compatibility tests between ONNX and deep learning frameworks.

It requires implementing:

- a method for ONNX model import for a given framework,
- a method for ONNX model export from a given framework,
- a list of models implemented in a given framework, where each model is exported to ONNX, and then imported back to the framework.

The `ONNXConversion` class implements a method for model conversion. It catches exceptions and any issues in the import/export methods, and provides a report on conversion status per model.

See the TensorFlowONNXConversion class for an example of API usage.

7.1 ONNX support grid in deep learning frameworks

7.2 ONNX conversion support grid

Note: This section was generated using:

```
python -m kenning.scenarios.onnx_conversion \
    build/models-directory \
    build/onnx-support.md \
    --converters-list \
        kenning.onnxconverters.mxnet.MXNetONNXConversion \
        kenning.onnxconverters.pytorch.PyTorchONNXConversion \
        kenning.onnxconverters.tensorflow.TensorFlowONNXConversion
```

Table 7.1: ONNX conversion support grid

Model Name	mxnet (ver. 1.9.1)	pytorch (ver. 1.13.1+cu117)	tensorflow (ver. 2.9.3)
DenseNet201	supported / supported	supported / unsupported	supported / ERROR
MobileNetV2	supported / ERROR	supported / unsupported	supported / ERROR
ResNet50	supported / supported	supported / unsupported	supported / ERROR
DeepLabV3 ResNet50	ERROR / unverified	supported / unsupported	Not provided / Not provided
Faster R-CNN ResNet50 FPN	supported / ERROR	supported / unsupported	Not provided / Not provided
Mask R-CNN	supported / ERROR	supported / unsupported	Not provided / Not provided
RetinaNet ResNet50 FPN	Not provided / Not provided	supported / unsupported	Not provided / Not provided

Table 7.1 table shows ONNX conversion support for several of the popular models across various deep learning frameworks. Each row represents a different deep learning model. Each column represents a different deep learning framework. Each cell lists an export to ONNX and import from ONNX support status for a given model and framework.

Firstly, the model is downloaded for a given framework. Secondly, the model is converted to ONNX. Lastly, the ONNX model is converted back to the framework's format.

The values in cells are in `<export support> / <import support>` format.

Possible values are:

- `supported` if export or import was successful,
- `unsupported` if export or import is not implemented for a given framework,
- `ERROR` if export or import ended up with error for a given framework,
- `unverified` if import could not be tested due to lack of support for export or an error during export.
- `Not provided` if the model was not provided for the framework.

CHAPTER

EIGHT

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).

8.1 Pet Dataset classification using TVM-compiled TensorFlow model

8.1.1 Commands used

Note: This section was generated using:

```
python -m kenning.scenarios.inference_tester \
    kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2 \
    kenning.datasets.pet_dataset.PetDataset \
    ./build/local-cpu-tvm-tensorflow-classification.json \
    --compiler-cls \
        kenning.compilers.tvm.TVMCompiler \
    --runtime-cls \
        kenning.runtimes.tvm.TVMRuntime \
    --model-path \
        ./kenning/resources/models/classification/tensorflow_pet_dataset_
↪mobilenetv2.h5 \
    --model-framework \
        keras \
    --target \
        llvm \
    --compiled-model-path \
        ./build/compiled-model.tar \
    --opt-level \
```

(continues on next page)

(continued from previous page)

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

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

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

Model framework:

- tensorflow ver. 2.9.3

Compiler framework:

- tvm ver. 0.11.1

Input JSON:


```
{
    "dataset": {
        "type": "kenning.datasets.pet_dataset.PetDataset",
        "parameters": {
            "classify_by": "breeds",
            "image_memory_layout": "NHWC",
            "dataset_root": "build/pet-dataset",
            "inference_batch_size": 1,
            "download_dataset": true,
            "external_calibration_dataset": null
        }
```


(continues on next page)

(continued from previous page)

```
    },
    "model_wrapper": {
        "type": "kenning.modelwrappers.classification.tensorflow_pet_dataset.
↪TensorFlowPetDatasetMobileNetV2",
        "parameters": {
            "model_path": "kenning/resources/models/classification/tensorflow_pet_
↪dataset_mobilenetv2.h5"
        }
    },
    "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,
            "disable_performance_measurements": true
        }
    },
    "optimizers": [
        {
            "type": "kenning.compilers.tvm.TVMCompiler",
            "parameters": {
                "model_framework": "keras",
                "target": "llvm",
                "target_host": null,
                "opt_level": 3,
                "libdarknet_path": "/usr/local/lib/libdarknet.so",
                "compile_use_vm": false,
                "output_conversion_function": "default",
                "conv2d_data_layout": "",
                "conv2d_kernel_layout": "",
                "use_fp16_precision": false,
                "use_int8_precision": false,
                "use_tensorrt": false,
                "dataset_percentage": 0.25,
                "compiled_model_path": "build/compiled-model.tar"
            }
        }
    ]
}
```

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

8.2.1 Inference time

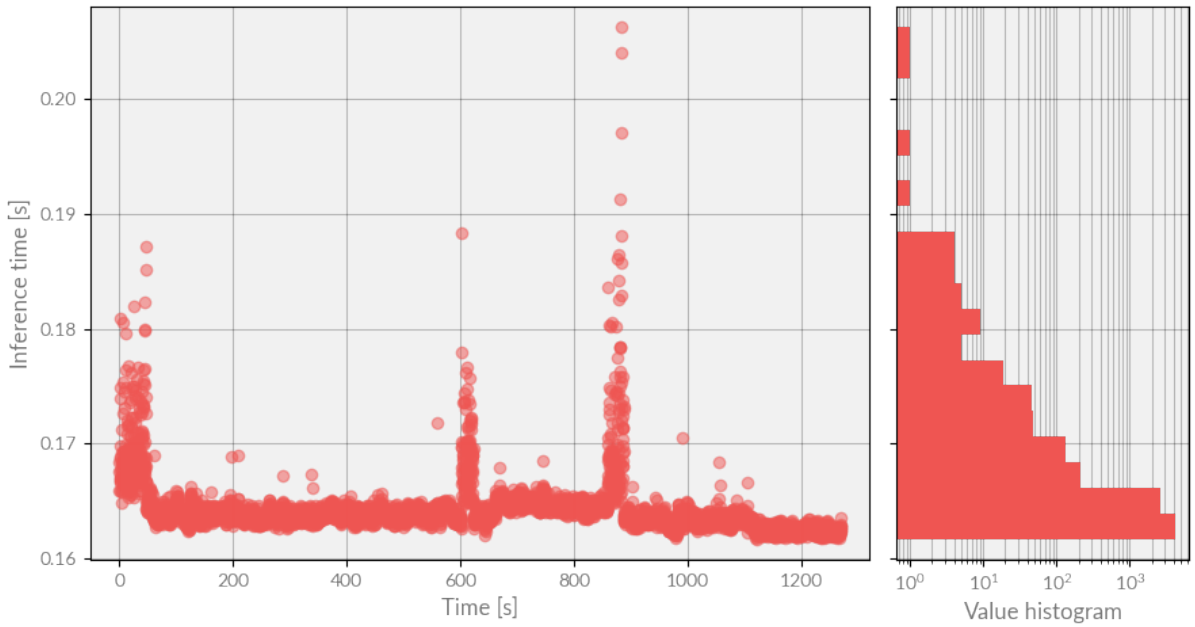


Figure 8.1: Inference time

- *First inference duration* (usually including allocation time): **0.17529404300000806**,
- *Mean*: **0.16417127104653828 s**,
- *Standard deviation*: **0.002196815747795452 s**,
- *Median*: **0.16378020400009063 s**.

8.2.2 Average CPU usage

- *Mean*: **51.14232200839471 %**,
- *Standard deviation*: **6.221318910822455 %**,
- *Median*: **50.0 %**.

8.2.3 Memory usage

- *Mean*: **19.696697552862915 %**,
- *Standard deviation*: **0.19819257109373487 %**,
- *Median*: **19.7 %**.

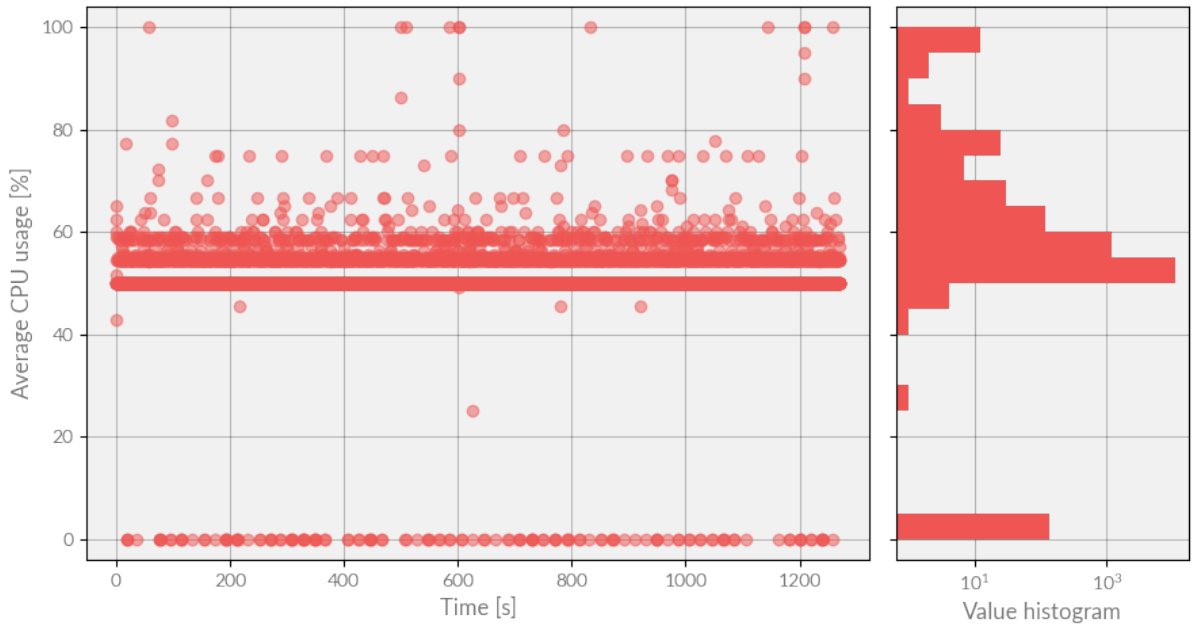


Figure 8.2: Average CPU usage during benchmark

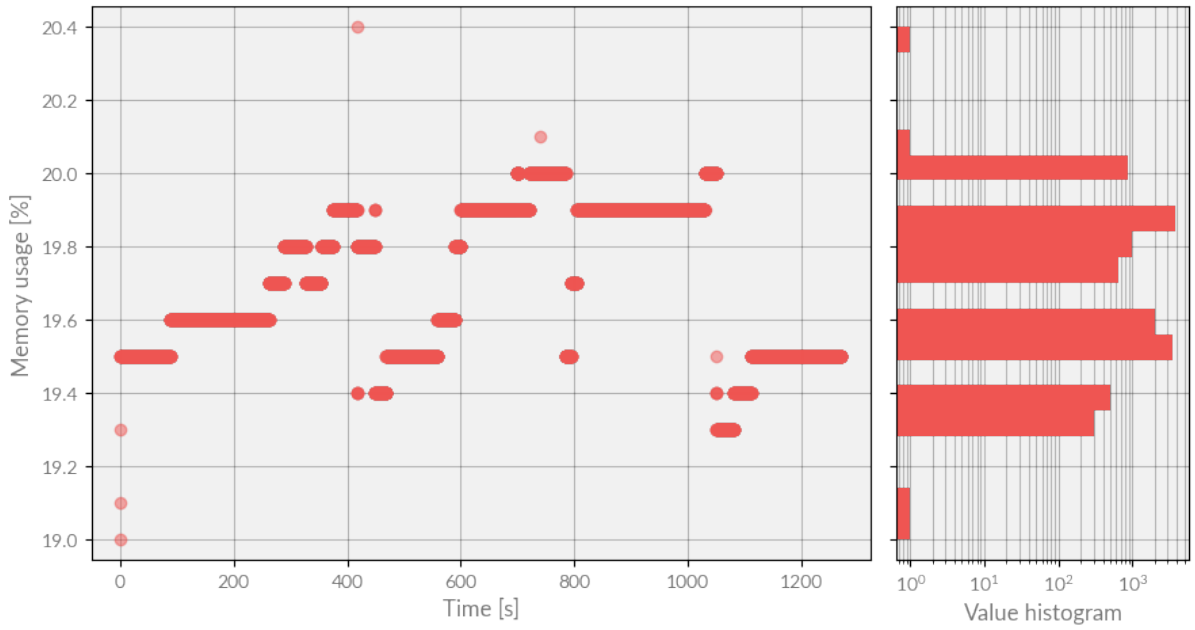


Figure 8.3: Memory usage during benchmark

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

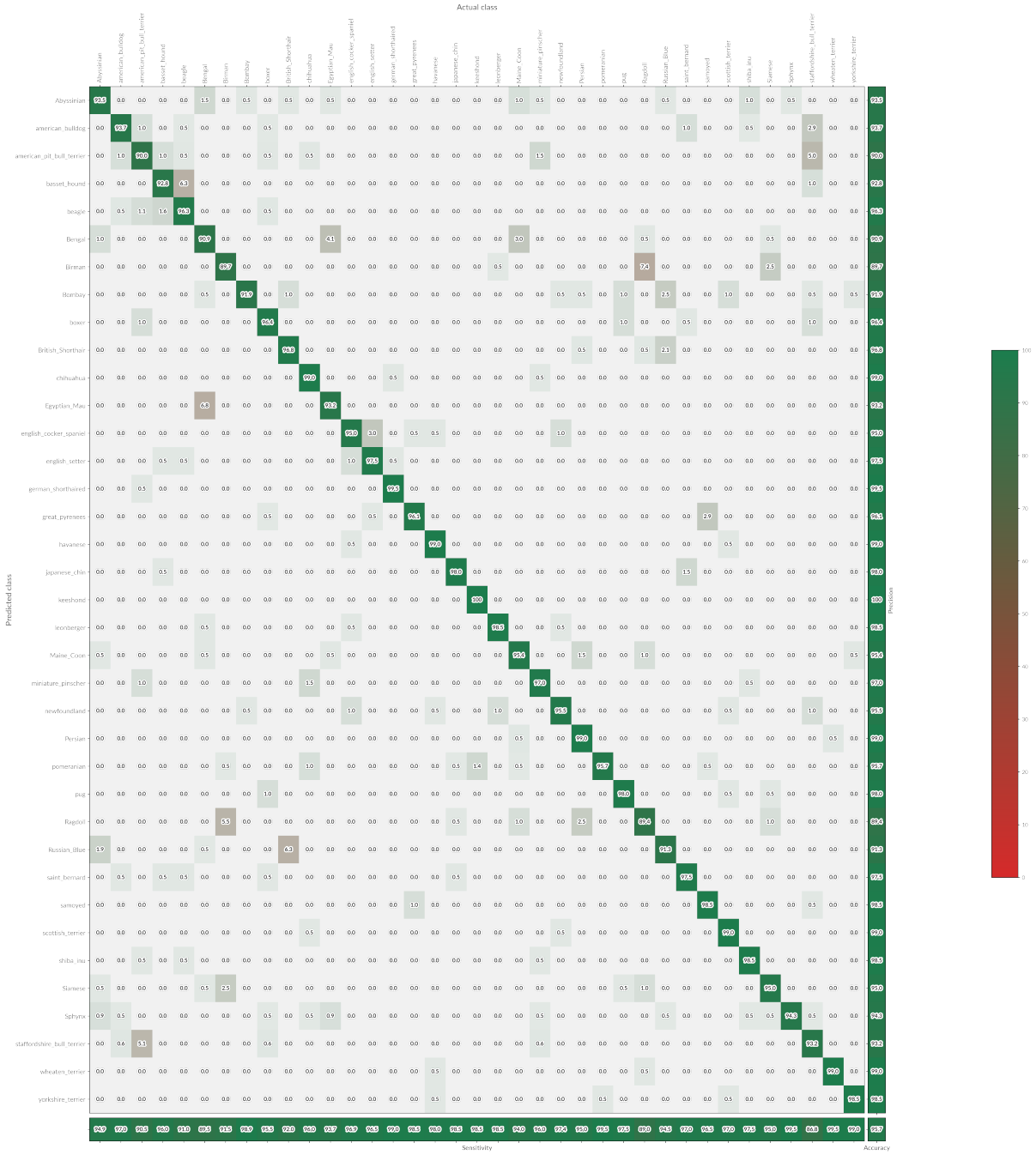


Figure 8.4: Confusion matrix

- *Accuracy*: **0.957273098380732**
- *Top-5 accuracy*: **0.9961899578173902**
- *Mean precision*: **0.9574413909275928**
- *Mean sensitivity*: **0.9571990915922841**
- *G-mean*: **0.9566448082813757**

CHAPTER

NINE

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.

9.1 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.

9.1.1 Runner IO

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

9.1.2 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`.

9.1.3 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[DectObject]`. This is interpreted as a list of `DectObjects`. The `DectObject` is a named tuple describing detection output (class names, rectangle positions, score).

9.1.4 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.

9.1.5 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.detectors.yolov4.ONNXYOLOV4",
        "parameters": {
          "model_path": "./kenning/resources/models/detection/yolov4.onnx"
        }
      },
      "runtime": {
        "type": "kenning.runtimes.onnx.ONNXRuntime",
        "parameters":
        {
          "save_model_path": "./kenning/resources/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.

9.2 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:

```
python -m kenning.scenarios.json_flow_runner 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.

9.3 Implemented Runners

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

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

CHAPTER

TEN

DEVELOPING KENNING BLOCKS

This chapter describes the development process of Kenning components.

10.1 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"
        }
    ]
}
```

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[DectObject]"
        }
    ]
}
```

In general, the metadata file consist of four fields:

- `input` is a specification of data passed to model wrapper,
- `processed_input` is a specification of input data preprocessed for wrapped model,
- `output` is a specification of data returned by wrapped model,
- `processed_output` is a specification of output data postprocessed by model wrapper.

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.

Parameters common to all fields:

- `name` - input/output name,

Parameters specific to `input` and `output`:

- `shape` - input/output tensor shape,

- `dtype` - input/output type,
- `order` - some of the runtimes/compilers 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 `processed_input` and `processed_output`:

- `type` - input/output type if different than `np.ndarray` (i.e. `List[SegmObject]` in segmentation model postprocessed output).

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.

10.2 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
```

10.2.1 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 `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 `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 kenning.core.optimizer import Optimizer


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
```

(continues on next page)

(continued from previous page)

```
        super().__init__(dataset, compiled_model_path)

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

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

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

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

10.2.2 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
```

(continues on next page)

(continued from previous page)

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


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


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

    outputtypes = [
        'tflite'
    ]

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

    def __init__(
```

(continues on next page)

(continued from previous page)

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

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

10.2.3 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
```

(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
        )

    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
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

        # 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.

10.2.4 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/resources/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"
```

(continues on next page)

(continued from previous page)

```
            }
        }
    ],
    "runtime":
    {
        "type": "kenning.runtimes.tflite.TFLiteRuntime",
        "parameters":
        {
            "save_model_path": "./build/compiled-model.tflite"
        }
    }
}
```

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

This sums up the Kenning development process.

10.3 Implementing Kenning runtime blocks

10.3.1 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

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

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

CHAPTER

ELEVEN

KENNING API

11.1 Deployment API overview

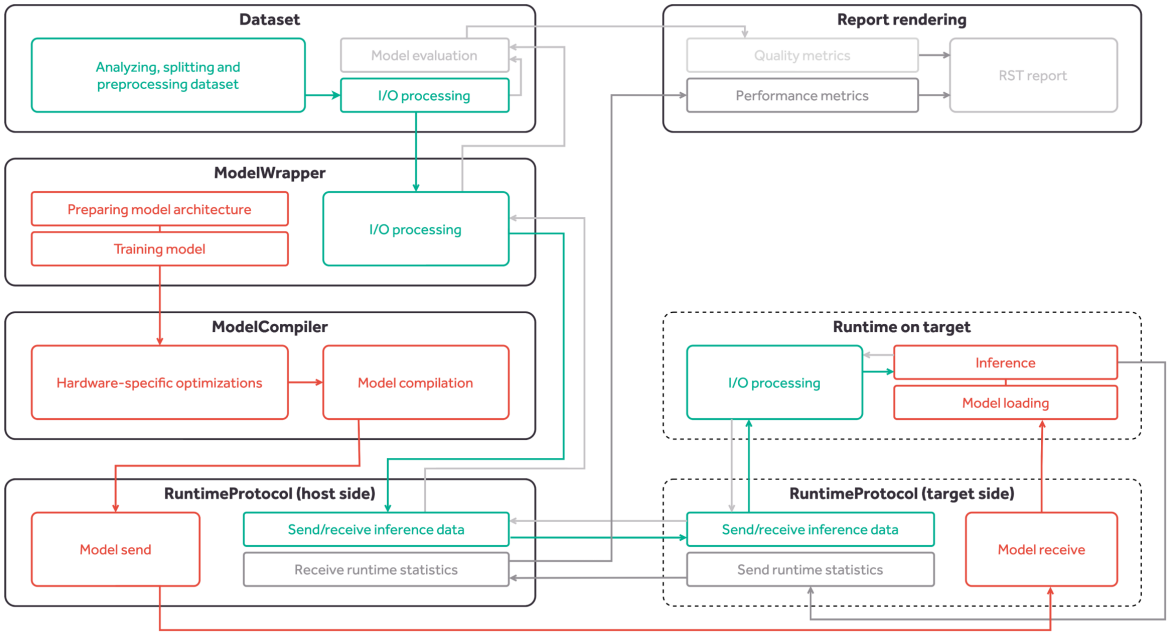


Figure 11.1: Kenning core classes and interactions between them. The green blocks represent the flow of input data passed to the model for inference. The orange blocks represent the flow of model deployment, from training to inference on target device. The grey blocks represent the inference results 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 *RuntimeProtocol* class - implements the communication protocol between the host and the target,
- 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).

11.1.1 Model processing

The orange blocks and arrows in Figure 11.1 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 *RuntimeProtocol*,
- 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*.

11.1.2 I/O data flow

The data flow is represented in the Figure 11.1 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 to apply modifications. During inference testing, the data is sent to and from the target using *RuntimeProtocol*.

Lastly, since *Runtime*s also have their specific representations of data, proper I/O processing is applied.

11.1.3 Data flow reporting

Report rendering requires performance metrics and quality metrics. The flow for this is presented with grey lines and blocks in Figure 11.1.

On target side, performance metrics are computed and sent back to the host using the *RuntimeProtocol*, 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 passsed to report rendering.

The final report is generated as an RST file containing figures, as can be observed in the *Sample autogenerated report*.

11.2 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*]*)

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 entires:

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()

Creates schema for the KenningFlow class

Returns
Schema for the class

Return type
Dict

classmethod from_json(runners_specifications*: list[dict[str, Any]]*)

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
List of runners that creates the flow.

Returns
object of class KenningFlow

Return type
KenningFlow

run()

Main process function. Repeatedly runs constructed graph in a loop.

run_single_step()

Runs flow one time.

11.3 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]*)

Represents an operation block in Kenning Flow.

cleanup()

Method that cleans resources after Runner is no longer needed

classmethod from_json(json_dict*: dict*, inputs_sources*: dict[str, tuple[int, str]]*, inputs_specs*: dict[str, dict]*, outputs*: dict[str, str]*)

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

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

11.4 Dataset

`kennning.core.dataset.Dataset`-based classes are responsible for:

- dataset preparation, including download routines (use the `--download-dataset` flag to download the dataset data),
- input preprocessing into a format expected by most models for a given task,
- output postprocessing for the evaluation process,
- model evaluation based on its predictions,
- sample subdivision into training and validation datasets.

The Dataset objects are used by:

- *ModelWrapper* - for training purposes and model evaluation,
- *Optimizer* - can be used e.g. for extracting a calibration dataset for quantization purposes,
- *Runtime* - for model evaluation on target hardware.

The available dataset implementations are included in the `kenning.datasets` submodule. Example implementations:

- PetDataset for classification,
- OpenImagesDatasetV6 for object detection,
- RandomizedClassificationDataset.

class kenning.core.dataset.**Dataset**(*root: Path*, *batch_size: int* = 1, *download_dataset: bool* = **False**, *external_calibration_dataset: Path | None* = **None**)

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

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

download_dataset_fun()

Downloads the dataset to the root directory defined in the constructor.

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 form_argparse()

Creates argparse parser for the Dataset object.

This method is used to create a list of arguments for the object so it is possible to configure the object from the level of command line.

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
(ArgumentParser, ArgumentGroup)

classmethod form_parameterschema()

Creates schema for the Dataset class.

Returns
schema for the class

Return type
Dict

classmethod from_argparse(args)

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 : Dict
arguments from ArgumentParser object

Returns
object of class Dataset

Return type
Dataset

classmethod from_json(json_dict: *dict*)

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

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]

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
the standardization values for a given train dataset. Tuple of two variables describing mean and std values

Return type
Tuple[Any, Any]

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_genereator 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.

Returns
List of objects that are usable by the prepare_input_samples method

Return type
List[Any]

prepare_input_samples(samples: *list*) → list

Preprocesses 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

Preprocesses 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

set_batch_size(batch_size)

Sets the batch size of the data in the iterator batches.

Parameters

batch_size : int
Number of input samples per batch

train_test_split_representations(test_fraction: *float* = 0.25, seed: *int* = 1234, validation: *bool* = **False**, validation_fraction: *float* = 0.1) → tuple[list, ...]

Splits the data representations into train dataset and test dataset.

Parameters

test_fraction : float
The fraction of data to leave for model validation

seed : int
The seed for random state

validation : bool
Whether to return a third, validation dataset

validation_fraction : float
The fraction (of the total size) that should be split out of the training set

Returns
Data splitted into train, test and optionally validation subsets

Return type
Tuple[List, . . .]

11.5 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 neccessary,
- 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.runtimeprotocol.RuntimeProtocol` 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**(*modelpath: Path*, *dataset:* Dataset, *from_file: bool* = **True**)

Wraps the given model.

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

convert_output_from_bytes(outputdata: *bytes*) → Any

Converts bytes array to the model output format.

The converted bytes are later passed to postprocess_outputs method.

Parameters

outputdata : bytes
Output data in raw bytes

Returns
Output data to feed to postprocess_outputs

Return type
Any

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 form_argparse()
Creates argparse parser for the ModelWrapper object.

Returns
the argument parser object that can act as parent for program's argument parser

Return type
ArgumentParser

classmethod form_parameterschema()
Creates schema for the ModelWrapper class.

Returns
schema for the class

Return type
Dict

classmethod from_argparse(dataset: Dataset, args, from_file: *bool* = **True**)
Constructor wrapper that takes the parameters from argparse args.

Parameters

dataset : *Dataset*
The dataset object to feed to the model

args : Dict
Arguments from ArgumentParser object

from_file : bool
Determines if the model should be loaded from modelspath

Returns
object of class ModelWrapper

Return type
ModelWrapper

classmethod from_json(dataset: Dataset, json_dict: *dict*, from_file: *bool* = **True**)
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

dataset : *Dataset*
The dataset object to feed to the model

json_dict : Dict
Arguments for the constructor

from_file : bool
Determines if the model should be loaded from modelspath

Returns
object of class ModelWrapper

Return type
ModelWrapper

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]]

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]]

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

Returns path to the model in a form of a Path object.

Returns
The path to the model

Return type
Path

`load_model`(modelpath: *Path*)

Loads the model from file.

Parameters

`modelpath` : Path
Path 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 fils 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: *Any*) → list

Processes the outputs for a given model.

By default no action is taken, and the outputs are passed unmodified.

Parameters

`y` : Any
The output from the model

Returns
The postprocessed outputs from the model that need to be in format requested by the Dataset object.

Return type
List

`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

Preprocesses the inputs for a given model before inference.

By default no action is taken, and the inputs are passed unmodified.

Parameters

X : List
The input data from the Dataset object

Returns
The preprocessed inputs that are ready to be fed to the model

Return type
Any

run_inference(X: *list*) → Any

Runs inference for a given preprocessed input.

Parameters

X : List
The preprocessed inputs for the model

Returns
The results of the inference.

Return type
Any

save_model(modelpath: *Path*)

Saves the model to file.

Parameters

modelpath : Path
Path to the model file

save_to_onnx(modelpath: *Path*)

Saves the model in the ONNX format.

Parameters

modelpath : Path
Path to the ONNX file

test_inference() → *Measurements*

Runs the inference with a given 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

11.6 Optimizer

`kenning.core.optimizer.Optimizer` objects wrap the deep learning compilation process. They can perform the model optimization (operation fusion, quantization) as well.

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 compilers:

- TFLiteCompiler - wraps TensorFlow Lite compilation,
- TVMCompiler - wraps TVM compilation.

class kenning.core.optimizer.**Optimizer**(*dataset:* Dataset, *compiled_model_path: Path*)

Compiles the given model to a different format or runtime.

compile(inputmodelpath*: Path*, 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 *inputmodelpath* + *.json*.

The compiled model is saved to compiled_model_path and the specification is saved to compiled_model_path + .json

Parameters

inputmodelpath : Path
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*=**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.

Raises
ValueError : – Raised if there is no matching format.

Returns
Matching format

Return type
str

classmethod form_argparse()

Creates argparse parser for the Optimizer object.

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
(ArgumentParser, ArgumentGroup)

classmethod form_parameterschema()

Creates schema for the Optimizer class.

Returns
schema for the class

Return type
Dict

classmethod from_argparse(dataset: Dataset, args)

Constructor wrapper that takes the parameters from argparse args.

Parameters

dataset : *Dataset*
The dataset object that is optionally used for optimization

args : Dict
arguments from ArgumentParser object

Returns
object of class Optimizer

Return type
Optimizer

classmethod from_json(dataset: Dataset, json_dict: *dict*)

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

dataset : *Dataset*
The dataset object that is optionally used for optimization

json_dict : Dict
Arguments for the constructor

Returns
object of class Optimizer

Return type
Optimizer

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_output_formats() → list[str]

Returns list of names of possible output formats.

Returns
List of possible output formats

Return type
List[str]

get_spec_path(modelpath: *Path*) → Path

Returns input/output specification path for the model saved in *modelpath*. It concatenates *modelpath* and *.json*.

Parameters

modelpath : Path
Path where the model is saved

Returns
Path to the input/output specification of a given model.

Return type
Path

load_io_specification(modelpath: *Path*) → dict[str, list[dict]] | None

Returns saved input and output specification of a model saved in *modelpath* if there is one. Otherwise returns None

Parameters

modelpath : Path
Path to the model which specification the function should read

Returns
Specification of a model saved in *modelpath* if there is one. None otherwise

Return type
Optional[Dict[str, List[Dict]]]

save_io_specification(inputmodelpath: *Path*, 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 *inputmodelpath* + *.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

inputmodelpath : Path
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.

set_input_type(*inputtype: str*)

Sets input type of the model for the compiler.

11.7 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 *RuntimeProtocol* 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**(*protocol:* RuntimeProtocol, *collect_performance_data: bool* = **True**)

Runtime object provides an API for testing inference on target devices.

Using a provided RuntimeProtocol it sets up a client (host) and server (target) communication, during which the inference metrics are being analyzed.

close_server()

Indicates that the server should be closed.

classmethod form_argparse()

Creates argparse parser for the Runtime object.

Returns
the argument parser object that can act as parent for program's argument parser

Return type
ArgumentParser

classmethod form_parameterschema()

Creates schema for the Runtime class.

Returns
Dict

Return type
schema for the class

classmethod from_argparse(protocol, args)

Constructor wrapper that takes the parameters from argparse args.

Parameters

protocol : *RuntimeProtocol*
RuntimeProtocol object

args : Dict
arguments from ArgumentParser object

Returns
RuntimeProtocol

Return type
object of class RuntimeProtocol

classmethod from_json(protocol: RuntimeProtocol, json_dict: *dict*)

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

protocol : *RuntimeProtocol*
RuntimeProtocol object

json_dict : Dict
Arguments for the constructor

Returns
Runtime

Return type
object of class 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(modelpath: *Path*) → Path

Gets path to a input/output specification file which is *modelpath* and *.json* concatenated.

Parameters

modelpath : Path
Path to the compiled model

Returns
Path

Return type
Returns path to the specification

infer(X: *ndarray*, modelwrapper: ModelWrapper, postprocess: *bool* = **True**) → Any

Runs inference on single batch locally using a given runtime.

Parameters

X : np.ndarray
Batch of data provided for inference

modelwrapper : *ModelWrapper*
Model that is executed on target hardware

postprocess : bool
Indicates if model output should be postprocessed

Returns
obtained values

Return type
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.

`postprocess_output`(results: *list[ndarray]*) → bytes

The method accepts output of the model and postprocesses it.

The output is quantized and converted to a correct dtype if needed.

Some compilers can change the order of the layers. If that's the case the methods also reorders the output to match the original order of the model before compilation.

Parameters

`results` : list[np.ndarray]
List of outputs of the model

Returns
bytes

Return type
Postprocessed output converted to bytes

:raises AttributeError : Raised if output specification is not loaded.:

`prepare_client`()

Runs initialization for the client.

`prepare_input`(input_data: *bytes*)

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` : bytes
Input data in bytes delivered by the client, preprocessed

Returns
bool

Return type
True if succeded

:raises ModelNotLoadedError : Raised if model 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]
io_specification or None, if it should be loaded from another source.

Returns
Whether the parsing of IO specification was succesfull

Return type
bool

prepare_local() → bool

Runs initialization for the local inference.

Returns
bool

Return type
True if initialized successfully

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).

Parameters

input_data : Optional[bytes]
Model data or None, if the model should be loaded from another source.

Returns
bool

Return type
True if succeded

prepare_server()

Runs initialization of the server.

preprocess_input(input_data: *bytes*) → list[ndarray]

The method accepts *input_data* in bytes and preprocesses it so that it can be passed to the model.

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
list[np.ndarray] – ready to be passed to the model.

Return type
List of inputs for each layer which are

:raises AttributeError : Raised if output specification is not loaded.: :raises ValueError : Raised if size of input doesn't match the input specification # noqa: E501:

process_input(input_data)

Processes received input and measures the performance quality.

Parameters

input_data : bytes
Not used here

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

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 prepare_input method.

:raises ModelNotLoadedError : Raised if model is not loaded:

run_client(dataset*: Dataset*, modelwrapper*: ModelWrapper*, compiledmodelpath*: Path*)

Main runtime client program.

The client performance procedure is as follows:

- connect with the server
- upload the model
- send dataset data in a loop to the server:

  - upload input
  - request processing of inputs
  - request predictions for inputs
  - evaluate the response
- collect performance statistics
- end connection

Parameters

dataset : *Dataset*
Dataset to verify the inference on

modelwrapper : *ModelWrapper*
Model that is executed on target hardware

compiledmodelpath : Path
Path to the file with a compiled model

Returns
bool

Return type
True if executed successfully

`run_locally`(dataset: Dataset, modelwrapper: ModelWrapper, compiledmodelpath: *Path*)

Runs inference locally using a given runtime.

Parameters

dataset : *Dataset*
Dataset to verify the inference on

modelwrapper : *ModelWrapper*
Model that is executed on target hardware

compiledmodelpath : Path
Path to the file with a compiled model

Returns
bool

Return type
True if executed successfully

`run_server`()

Main runtime server program.

It waits for requests from a single client.

Based on requests, it loads the model, runs inference and provides statistics.

`upload_essentials`(compiledmodelpath: *Path*)

Wrapper for uploading data to the server. Uploads model by default.

Parameters

compiledmodelpath : Path
Path to the file with a compiled model

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
bytes

Return type
data to send to the client

:raises ModelNotLoadedError : Raised if model is not loaded:

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
bytes

Return type
statistics to be sent to the client

11.8 RuntimeProtocol

The `kenning.core.runtimeprotocol.RuntimeProtocol` class conducts communication between the client (host) and the server (target).

The RuntimeProtocol 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.

RuntimeProtocol examples:

- NetworkProtocol - implements a TCP-based communication between the host and the client.

11.8.1 Runtime 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.runtimeprotocol.RuntimeProtocol`-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.runtimeprotocol.RuntimeProtocol` class. The implementation of running inference on the given target is contained within the `kenning.core.runtime.Runtime` class.

11.8.2 RuntimeProtocol API

`kenning.core.runtimeprotocol.RuntimeProtocol`-based classes implement the *Runtime 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.runtimeprotocol.**RuntimeProtocol**

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.

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[bool, Optional[bytes]] – successful) and downloaded data

Return type
tuple with download status (True if

download_statistics() → *Measurements*

Downloads inference statistics from the target device.

By default no statistics are gathered.

Returns
Measurements

Return type
inference statistics on target device

classmethod form_argparse()

Creates argparse parser for the RuntimeProtocol object.

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
(ArgumentParser, ArgumentGroup)

classmethod form_parameterschema()

Creates schema for the RuntimeProtocol class.

Returns
Dict

Return type
schema for the class

classmethod from_argparse(args)

Constructor wrapper that takes the parameters from argparse args.

Parameters

args : Dict
arguments from RuntimeProtocol object

Returns
RuntimeProtocol

Return type
object of class RuntimeProtocol

classmethod from_json(json_dict*: dict*)

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
RuntimeProtocol

Return type
object of class RuntimeProtocol

initialize_client() → bool

Initializes client side of the runtime 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
bool

Return type
True if succeded

initialize_server() → bool

Initializes server side of the runtime protocol.

The server side is supposed to run on target hardware.

The parameters for the server should be provided in the constructor.

Returns
bool

Return type
True if succeded

parse_message(message: *bytes*) → tuple[MessageType, bytes]

Parses message received in the wait_for_activity method.

The message type is determined from its contents and the optional data is returned along with it.

Parameters

message : bytes
Received message

Returns
Tuple['MessageType', bytes]

Return type
message type and accompanying data

receive_data() → tuple[ServerStatus, Any]

Gathers data from the client.

This method should be called by wait_for_activity method in order to receive data from the client.

Returns
Tuple[ServerStatus, Any]

Return type
receive status along with received data

request_failure() → bool

Sends ERROR message back to the client if it failed to handle request.

Returns
bool

Return type
True if sent successfully

request_processing() → 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.

Returns
bool

Return type
True if inference finished successfully

request_success(data: *bytes* = b'') → bool

Sends OK message back to the client once the request is finished.

Parameters

data : bytes
Optional data upon success, if any

Returns
bool

Return type
True if sent successfully

send_data(data: *bytes*) → bool

Sends data to the target device.

Data can be model to use, input to process, additional configuration.

Parameters

data : bytes
Data to send

Returns
bool

Return type
True if successful

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
bool

Return type
True if ready for inference

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
bool

Return type
True if data upload finished successfully

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
bool

Return type
True if model upload finished successfully

wait_for_activity() → list[tuple[ServerStatus, Any]]

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.

Returns
list of messages along with status codes.

Return type
List[Tuple['ServerStatus', Any]]

11.9 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 raport 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>>*) → list

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
the initial value for the measurement

add_measurements_list(measurementtype: *str*, valueslist: *list*)

Adds new values to a given measurement type.

Parameters

measurementtype : str
the measurement type to be updated

valueslist : List
the list of values to add

clear()

Clears measurement 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

Return type
list of values for a given measurement type

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()

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

Return type
Measurements object.

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)

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

kenning.core.measurements.**tagmeasurements**(tagname*: str*)

Decorator for adding tags for measurements and saving their timestamps.

Parameters

tagname : str
The name of tag.

kenning.core.measurements.**timemeasurements**(measurementname*: str*)

Decorator for measuring time of the function.

The duration is given in nanoseconds.

Parameters

measurementname : str
The name of the measurement type.

11.10 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*, *version*)

Creates ONNX conversion support matrix for given framework and models.

add_entry(name, modelgenerator, **kwargs)

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]

onnx_export(modelentry: *ModelEntry*, exportpath: *Path*)

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
SupportStatus

Return type
the support status of exporting given model to ONNX

onnx_import(modelentry: *ModelEntry*, importpath: *Path*)

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
SupportStatus

Return type
the support status of importing given model from ONNX

prepare()

Virtual function for preparing the ONNX conversion test.

This method should add model entries using add_entry methos.

It is later called in the constructor to prepare the list of models to test.

11.11 DataProvider

The `DataProvider` classes are used during deployment to provide data for inference. They can provide data from such sources as a camera, video files, microphone data or a TCP connection.

The available `DataProvider` implementations are included in the `kenning.dataproviders` submodule. Example implementations:

- CameraDataProvider for capturing frames from camera.

class kenning.core.dataprovider.**DataProvider**(*inputs_sources: dict[str, tuple[int, str]] = {}*, *inputs_specs: dict[str, dict] = {}*, *outputs: dict[str, str] = {}*)

detach_from_source()

Detaches from the source during shutdown

fetch_input() → Any

Gets the sample from device

Returns
Any

Return type
data to be processed by the model

classmethod form_argparse()

Creates argparse parser for the DataProvider object.

This method is used to create a list of arguments for the object so it is possible to configure the object from the level of command line.

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
(ArgumentParser, ArgumentGroup)

classmethod from_argparse(args)

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 : Dict
arguments from ArgumentParser object

Returns
DataProvider

Return type
object of class DataProvider

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
Any

Return type
preprocessed data

11.12 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,
  - RealTimeSegmentationVisualization for visualizing segmentation model outputs,
  - RealTimeClassificationVisualization 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] = {}*)

detach_from_output()

Detaches from the output during shutdown

classmethod form_argparse()

Creates argparse parser for the OutputCollector object.

This method is used to create a list of arguments for the object so it is possible to configure the object from the level of command line.

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
(ArgumentParser, ArgumentGroup)

classmethod from_argparse(args)

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 : Dict
arguments from ArgumentParser object

Returns
OutputCollector

Return type
object of class OutputCollector

classmethod from_json(json_dict: *dict*, inputs_sources: *dict[str, tuple[int, str]]* = {}, inputs_specs: *dict[str, dict]* = {}, outputs: *dict[str, str]* = {})

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

process_output(input_data: *Any*, output_data: *Any*)

Returns the infered 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

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
bool

Return type
True if exit condition was reached to break the loop

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

O

P

R

S

T

U

W