EIQTUG

eIQ Toolkit User Guide

Rev.1.8 — July 10, 2023

User guide

Document information

Information	Content
Keywords	Machine Learning, AI, TensorFlow, Neural Networks, eIQ, Computer Vision
Abstract	The eIQ Toolkit is a machine-learning software development environment that enables the use of ML algorithms on NXP microcontrollers, microprocessors, and SoCs.

Contents:

1 Introduction

The eIQ Toolkit is a machine-learning software development environment that enables the use of ML algorithms on NXP microcontrollers, microprocessors, and SoCs.

The eIQ Toolkit is for those interested in building machine learning solutions on embedded devices. A background in machine learning, especially in supervised classification, is helpful to understand the entire pipeline. However, if the user does not have any background in the areas mentioned above, the eIQ Toolkit is designed to assist the user.

The eIQ Toolkit consists of the following three key components:

- eIQ Portal
- eIQ Model Tool
- eIQ Command-line Tools

The eIQ Portal is designed to help you with image classification and object detection tasks without deep machine-learning knowledge. Using the eIQ Portal easy-to-use Graphical User Interface (GUI), you can create a project and import a custom dataset into it to solve a specific real-life problem. Then you select the model that fits your problem the best and train and evaluate it over the dataset you imported. The evaluation process gives you information about how accurate the model that you trained is and how fast it runs on the target architecture. In addition to this, the eIQ Portal provides very efficient techniques to improve your model performance, customized for each target type.

The eIQ Model Tool is used for the subsequent analysis of your models, including model and per-layer time profiling.

For users who prefer it, NXP offers a set of command-line tools (the eIQ Command-line Tools), which also include a self-contained Python environment.

Note: To use the eIQ Command-line Tools, it is important either to launch them using the **COMMAND LINE** button from the home screen or run the *<eIQ_Toolkit_install_dir>\bin\eiqenv[.bat/.sh]* script, which sets up the command-line environment.

2 Workflows

There are two approaches available with the eIQ Toolkit, based on what the user provides and what the expectations are. The following approaches are referred to throughout this document:

- **Bring Your Own Data** (BYOD) – the users bring image data, use the eIQ Toolkit to develop their own model, and deploy it on the target.
- **Bring Your Own Model** (BYOM) – the users bring a pretrained model and use the eIQ Toolkit for optimization, deployment, or profiling.

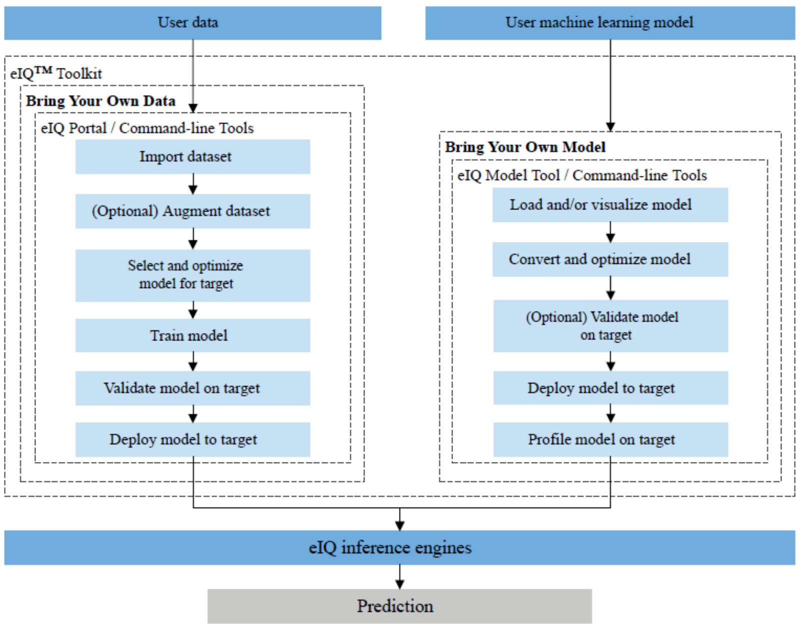

User data
User machine learning model
eIQ™ Toolkit
Bring Your Own Data
eIQ Portal / Command-line Tools
Import dataset
(Optional) Augment dataset
Select and optimize model for target
Train model
Validate model on target
Deploy model to target
Bring Your Own Model
eIQ Model Tool / Command-line Tools
Load and/or visualize model
Convert and optimize model
(Optional) Validate model on target
Deploy model to target
Profile model on target
eIQ inference engines
Prediction

Figure 2.1.: BYOM and BYOD workflows

3 eIQ Portal

The eIQ Portal is a GUI that allows the user to leverage the eIQ Toolkit capabilities in a practical and intuitive manner. With little or no knowledge of machine learning, the eIQ Portal enables the user to build models for image classification, segmentation and object-detection problems. In the eIQ Portal title bar (see Figure 3.1.), you can find several buttons in the home screen and in the opened project screen, respectively:

- **PLUG-INS** - provides information about all available converter plugins. You can check if the relevant plugin is ready and where it is located.
- **REMOTE DEVICES** - opens a modal window where you can set remote devices to validate and profile your model.
- **WORKSPACES** - (appears after opening a project) - this drop-down menu enables you to switch between the eIQ Portal sections of an opened project.
- **SETTINGS** - opens a modal window where you can change next user settings:
  - User plugins - sets the path to user models. You cannot change user folder during training, validation, deployment or base model selection.
- **MARKETPLACE** - provides information on additional services and solutions that are available from NXP and eco-system partners.
- **HELP** - provides the eIQ Toolkit documentation.

3.1 Home

In the eIQ Portal home screen, you can create a new project, open an existing one, launch the eIQ Model Tool, and open a command-line window by clicking on CREATE PROJECT, OPEN PROJECT, MODEL TOOL, and COMMAND LINE buttons, respectively.

3.2 Projects

A project is a database that stores the data that you work with and the related information. When you import images, add labels, or generally modify the data, these changes are automatically stored in the project. All the images that you work with are stored in the project as well, so when working with a large dataset, the file size is not small.

Use the *Importer* command-line tool for importing bigger datasets into a project.

There are a few things to note for more advanced users:

- There are two file extensions accepted - *.eiqp* and *.deepview*. There is no difference between the two.
- Internally, a project is organized in an SQLite database.

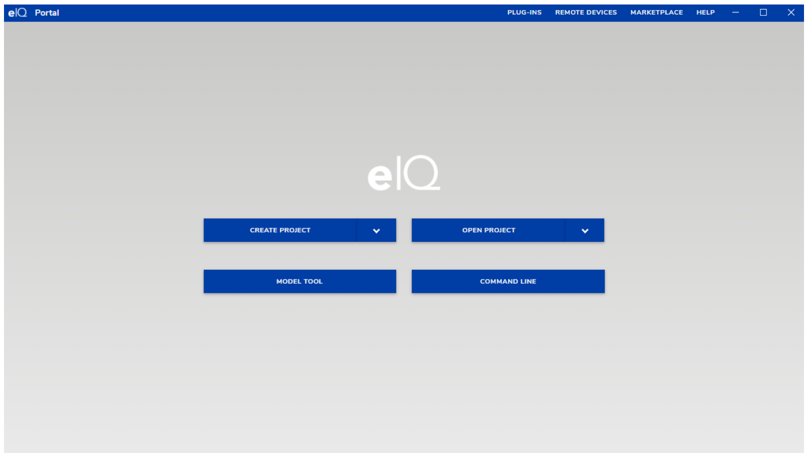
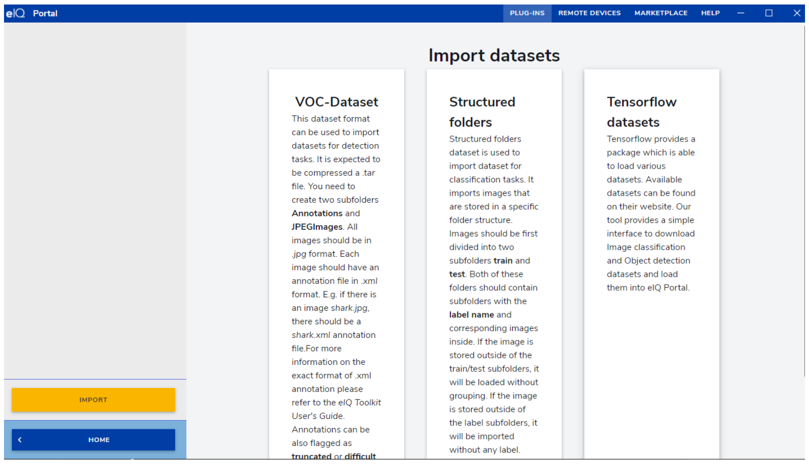


Figure 3.1.: eIQ Portal: home

Note: While it is possible to edit files manually, it is not recommended.

3.3 Import

Import screen lets you import some types of datasets into the eIQ Portal project. To open this screen, select the IMPORT DATASET option under the CREATE PROJECT button on the Home screen.



Import datasets

VOC-Dataset

This dataset format can be used to import datasets for detection tasks. It is expected to be compressed a .tar file. You need to create two subfolders **Annotations** and **JPEGImages**. All images should be in *.jpg* format. Each image should have an annotation file in *.xml* format. E.g. if there is an image *shark.jpg*, there should be a *shark.xml* annotation file.For more information on the exact format of .xml annotation please refer to the *eIQ Toolkit User's Guide*. Annotations can be also flagged as **truncated** or **difficult**

Structured folders

Structured folders dataset is used to import dataset for classification tasks. It imports images that are stored in a specific folder structure. Images should be first divided into two subfolders **train** and **test**. Both of these folders should contain subfolders with the **label name** and corresponding images inside. If the image is stored outside of the train/test subfolders, it will be loaded without grouping. If the image is stored outside of the label subfolders, it will be imported without any label.

Tensorflow datasets

Tensorflow provides a package which is able to load various datasets. Available datasets can be found on their website. Our tool provides a simple interface to download Image classification and Object detection datasets and load them into eIQ Portal.

Figure 3.2.: Import datasets

Currently, there are three types of datasets supported.

3.3.1 VOC dataset

VOC datasets are introduced in Pascal Visual Object Classes Challenge. This format is also used by the ImageNet datasets. Images and labels in this dataset can be used to train detection models.

3.3.1.1 Importing Pascal VOC dataset

To import already prepared datasets from the challenge, you can download the "VOC 2008" training dataset here. There is a "VOC 2007" test dataset that does not overlap with the training dataset. You can download it here. On the Import datasets screen, you can click the *VOC-Dataset* button. This will show you the settings for this dataset.

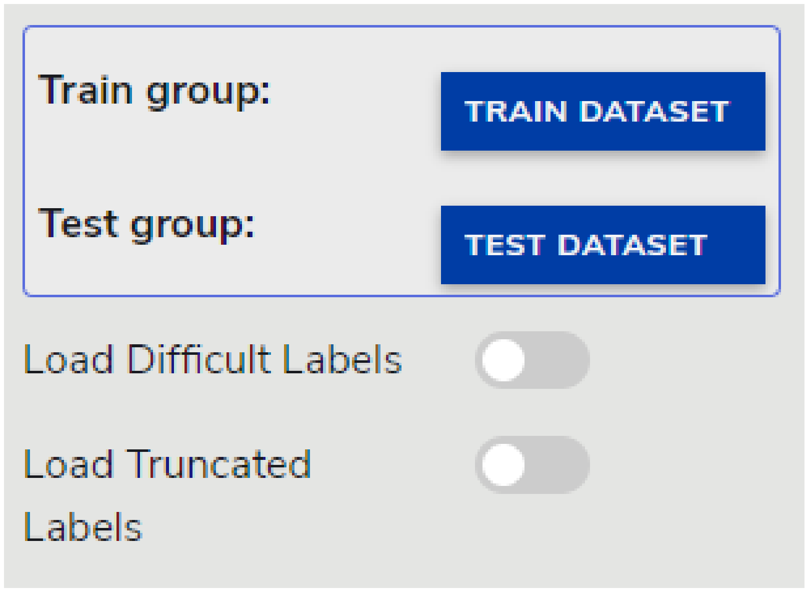


Figure 3.3.: Dataset settings

Click TRAIN DATASET to select the training dataset that you downloaded. It should be called "VOCtrainval_14-Jul-2008.tar". In a similar way, click the TEST DATASET button to select the "VOCtest_06-Nov-2007.tar" test dataset.

Some labels in the dataset are flagged as *difficult* or *truncated*. By checking the *Load Difficult Labels or Load Truncated Labels* options, you can select whether your dataset should also include these labels.

3.3.1.2 Creating and importing custom VOC datasets

The following is a simple guide on how to prepare a custom dataset and import it into the eIQ Portal:

1. Create a folder with two subfolders (*Annotations* and *JPEGImages*).
2. Copy the JPEG images that are used for training or validation into *JPEGImages* folder. The image filename should end with ".jpg"
3. Create the annotation files and save them to the *Annotations* folder. Each image must have an annotation file with the same name and ".xml" extension. For example, the *shark.jpg* image will have the *shark.xml* annotation file.
4. The annotation file has a root element called annotation. The root element contains one or more object elements representing labels for image classification or bounding boxes for object detection. An object element containing a name element only stands for the full image label. An object element containing both the "name" element and the "bndbox" element represents the region-specific label. The "bndbox" bounding box is defined by the "x" and

"y" coordinates of the top left and the "x" and "y" coordinates in the bottom right-hand side corner of the box. An example of the annotation format for an image with one full image label "flower" and one region-specific label "rose" is as follows:

```
<annotation>
    <object>
        <name>rose</name>
        <bndbox>
            <xmax>274</xmax>
            <xmin>77</xmin>
            <ymax>375</ymax>
            <ymin>67</ymin>
        </bndbox>
    </object>
    <object>
        <name>flower</name>
    </object>
</annotation>
```

5. Compress the top folder into a TAR file named "<dataset_tar_name>".
6. Prepare the test dataset in a similar way.
7. Open the eIQ Portal Import dataset screen and import your datasets in the same way as you imported the downloaded Pascal VOC datasets.

The VOC-type datasets can also be imported using command line. See *Importer* for information on how to do that.

3.3.2 Structured folders dataset

Using the *Import dataset* screen, you can also import datasets that have a specific folder structure. These datasets do not have to be compressed into TAR files as with VOC datasets. The only requirement is that the images are saved in specific folders. The datasets created this way can be used to train models for classification tasks. The structure of such dataset can look as follows:

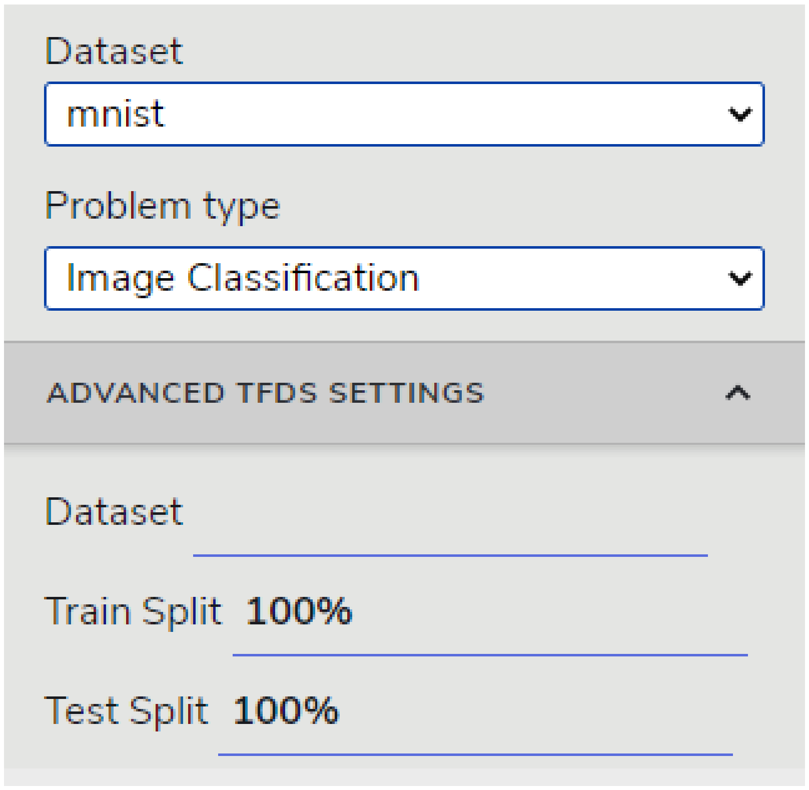
```
\---imgs
        |   50.jpg
        |
        +---test
        |   |   shark.jpg
        |   |
        |   \---dog
        |           dog.jpg
        |
        \---train
        |   grace_hopper.jpg
        |
        \---cat
        cat.jpg
```

In the example, the root folder is *imgs*. It can already contain images, but these do not have any labels assigned and they are not in the train or test groups. The root folder should then contain two subfolders (*train* and *test*). This divides images that are in these folders into training and testing groups. The *train* and *test* folders can also contain images. These images are assigned to the respective group, but they do not have any labels. The *train* and *test* folders should then contain subfolders with the label name. All images in these subfolders have the respective label assigned. In the example, there is the *cat.jpg* image in the *cat* folder. This means that the image is loaded into the dataset with the *cat* label. But *shark.jpg* is directly in the *test* folder. This means that it is in a test group, but with no label assigned.

To import the dataset, choose "Structured folders" from the "Import dataset" screen. A button to select the root folder appears. Select the folder and import the dataset.

3.3.3 TFDS datasets

TFDS datasets allow you to import some of the datasets provided by tensorflow in their catalog here. The tool allows you to download datasets for image classification and object detection. After choosing the TFDS datasets, you can modify the import using basic and advanced settings:


Dataset: mnist
Problem type: Image Classification

Dataset
Train Split 100%
Test Split 100%

Figure 3.4.: FDS Dataset settings

Basic settings allow you to select imported dataset from a list of verified datasets as well as set whether the dataset should be imported for classification or detection tasks.

Advanced settings then allow you to choose any dataset available on tensorflow site, but there is a risk, that the importer will not be able to import the dataset. Besides choosing any dataset, you can also regulate how many of the images will be imported into the eIQ project. You can set this number separately for training and testing splits of the dataset. There are several options on how to change this. If you don't know the exact number of samples in the dataset, you can use percentage. For example using 60% on the training split means that 60% of the images will be imported into the project. If you know the exact number of images, you can directly write this number. Also, if you want to for example choose 100 images, but starting on image 200 and ending on image 300, you can write '200:300'.

Each tensorflow dataset has a feature structure (for example like this). This describes what kind of data the dataset contains. For our purposes, the most important are images, labels and bounding boxes. The importer tries to find these in the feature structure. If it finds one kind of each, the dataset can be imported without any issues. However if there are more, for example some datasets contain input images as well as segmentation maps, then the importer cannot decide which is the right one. So when using advanced settings and importing dataset that is not verified, it is recommended to study the feature structure of the dataset to determine whether it can be imported or not.

3.3.4 Writing a custom script to create a dataset

You can also write your own Python script to create the eIQ Portal dataset and execute it through the eIQ Portal command line. The *deepview.datastore* API is available for this purpose. An example on how to create the CIFAR10 dataset is in *workspace/CIFAR_uploader*. You can run the example by launching the command line (navigate to the Home page and click COMMAND LINE). Then install the requirements by running:

```
python -m pip install -r requirements.txt
```

After the package is installed, you can run the example as follows:

```
python -m CIFAR_uploader
```

The script starts by creating a new project using a command from “deepview.datastore”. It creates a new empty project at the specified path called “cifar10.eiqp”:

```
project = ds.create_project(os.path.join('user_models', 'classification', 'image',
↪'cifar10', 'cifar10.eiqp'))
```

Tensorflow datasets package provides an easy way to load datasets. It is used to get the training and testing partitions:

```
train = tfds.load('cifar10', split='train')
test = tfds.load('cifar10', split='test')
```

These datasets are then split into batches of 100 images:

```
train = train.batch(100).prefetch(1)
test = test.batch(100).prefetch(1)
```

The “import_dataset” function, which uploads images and labels to a new project, is called. In the function, images for each batch are first encoded as JPEG, which is a format accepted by the datastore:

```
images.append(tf.io.encode_jpeg(image).numpy())
```

All the images in the batch are then loaded into the project as follows:

```
project.add_images(images, grouping)
```

This method then returns the objects that represent images inserted into the project. Using these objects, you can then assign labels to the images by adding annotations to the project. The “image” in the command is an element of the array returned from the add_images() function:

```
project.add_annotation(image, label)
```

If you are interested in learning more on how to write scripts to create custom datasets, you can look at Jupyter notebooks in the *workspace/importer* folder.

3.3.5 Segmentation datasets

Segmentation datasets differ from other types of datasets in annotations. Dataset used for training segmentation models should contain annotation masks instead of full image labels or rectangular regions. Masks are stored as other images, but they should contain values from 0 to number of classes. In this representation, they classify each pixel of input image into one of the detected classes.

Importer currently does not directly support segmentation datasets. You can, however, write your own script to prepare it. There is also an example Jupyter notebook in the *workspace/importer* folder.

Next steps will go through above mentioned Jupyter notebook and explain how to write custom script to create dataset for segmentation tasks. Some methods in API are different from previous chapter to accomodate different type of annotations.

1. Create new project

```
ds.create_project(project_path)
```

2. Add labels to project

```
for i, label in enumerate(labels):
    lab = project.add_label(label)
```

3. Encode image and mask

```
encoded_image = tf.io.encode_jpeg(image).numpy()
encoded_annotation = tf.io.encode_jpeg(mask).numpy()
```

4. Add image and annotation to project. To add annotation we use method modified specifically for segmentations.

```
project_image = project.add_image(encoded_image, 'train')
project.add_annotation_modified(project_image, label, None, mask=encoded_annotation)
```

5. To retrieve segmentation annotation from project, a modified function is also used. Result is a list containing three values [image_id, label, mask].

```
encoded_annotation = project.annotation_with_id_modified(annotation_id)
```

3.4 Plugins

Plugins are an extension to the Converter tool. The currently installed plugins allow for conversion between TF Lite, RTM, and ONNX formats, but a custom conversion can be implemented to extend the current functionality. Conversions are not only format-to-format conversions. Input and output formats can actually be the same and the plugin may do just about anything to the machine learning model internally, for example, a graph transformation or an optimization (such as quantization).

Note: These plugins are a different feature from the Extension Framework described in *Extension framework* and they will be merged in future releases.

3.4.1 Adding a new converter plugin

Plugins are implemented as Python modules installed in the eIQ Toolkit's Python environment. To implement them, use the following folder structure:

```
\---<project_directory>
    |   setup.py
    +---converter_plugin
        |
        +---assets
        |   |   image.svg or image.png
        |   |   info.html
        |   __main__.py
        |
        \---<other files or folders>
```

The *image.svg* or *image.png* files should have resolution of 128x128 and they show up in the eIQ Portal plugins screen. The info.html file describes the plugin details, parameters, and others. The main plugin source code file (*converter_plugin.__main__.py*) should implement the following functions:

- query_convert(src_type, dst_type) -> dict()
- convert(src_file, dst_file, params) -> dict()

See the content of modules such as deepview_onnx or deepview_rtm in the python site-packages directory on how these functions can be implemented.

On top of that, a standard *setup.py* file should also be implemented to build the wheel package. A sample *setup.py* script may look as follows:

```
from setuptools import setup, find_packages

Package_name = '<plugin_name>'
setup(
    name='<plugin_name>',
    version='<version>',
    description='<description>',
    url='<website>',
    author='<author>',
    author_email='<author_email>',
    license='<license>',
    entry_points={'deepview_converter':
                    ['converter = <package_name>.__main__']
                },
    packages=[ < package_name >],
package_data = {package: ['assets\\*']},
install_requires =
['<required_module>', '<another_required_module>', ],
classifiers = [],
)
```

When the files and folders are created and all the required code is implemented, the plugin can be compiled within the eIQ Toolkit command line:

```
python setup.py bdist_wheel --universal
```

This creates a *.whl file that can be installed using the following command:

```
deepview-converter -i <path_to_whl_file>
```

When the plugin is installed, it should be automatically discovered. To check that the plugin was installed successfully, run the following command:

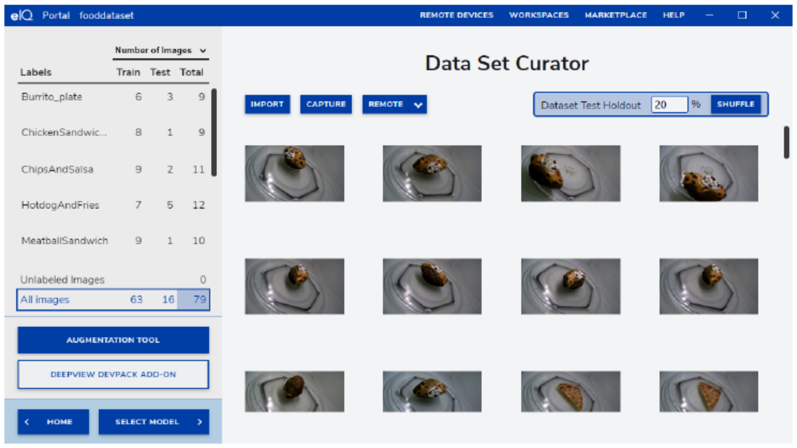
```
deepview-converter -l
```

A plugin can also be uninstalled if needed:

```
deepview-converter -u <plugin_name>
```

3.5 Data Set Curator

Data curation is the organization and integration of data collected from various sources. To open the "Data Set Curator" window, click the CREATE PROJECT or OPEN PROJECT buttons in the main menu (see Figure 3.1.). Depending on the images that you have currently imported, the dataset curator window can look as follows:



Number of Images

Labels	Train	Test	Total
Burrito_plate	6	3	9
ChickenSandwic...	8	1	9
ChipsAndSalsa	9	2	11
HotdogAndFries	7	5	12
MeatballSandwich	9	1	10
Unlabeled Images			0
All images	63	16	79

Dataset Test Holdout: 20 %

Figure 3.5.: eIQ Portal: dataset

The "Data Set Curator" workspace allows you to preview the images imported so far. This workspace uses a lazy loading strategy to show the images, so scroll down to see the rest.

Note: If there are no images in the current project, the "Data Set Curator" window shows the "No images found" message on the screen.

3.5.1 Dataset import

To import data into the project, the interface provides three different options that differ according to the source where the data is captured:

- IMPORT: You can browse your local machine to select multiple sample images and upload them at once.
- CAPTURE: This option utilizes a connected camera to take instantaneous images.
- REMOTE: This option allows you to capture images from a remote device.

3.5.2 Train-test split

The “Dataset Test Holdout” box in the “Data Set Curator” panel allows you to define what percentage of the collection of data in the imported data set is used for testing vs training. The selection is random and the data in the “test” set are not used in the training step. The input value corresponds to the size of the test data set. The default value is 20 %:

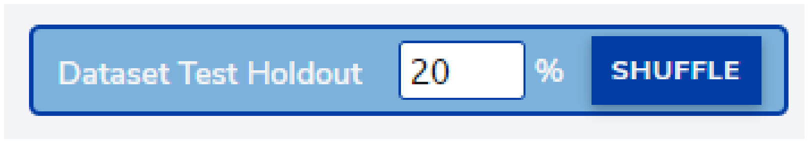


Figure 3.6.: eIQ Portal: “Dataset Test Holdout” box

3.5.3 Dataset annotation

To enable dataset annotation, click the image that you want to annotate. A good recommendation is to select the “Unlabeled Images” option in the left panel (see Figure 3.5.) to avoid confusion between the labeled and unlabeled images. By selecting this option, you only see the images that remain unlabeled.

There are two different ways to annotate the data:

- Bounding box: labels a specific region within the image.
- Full image: assigns a label to the whole image (see Figure 3.7.).

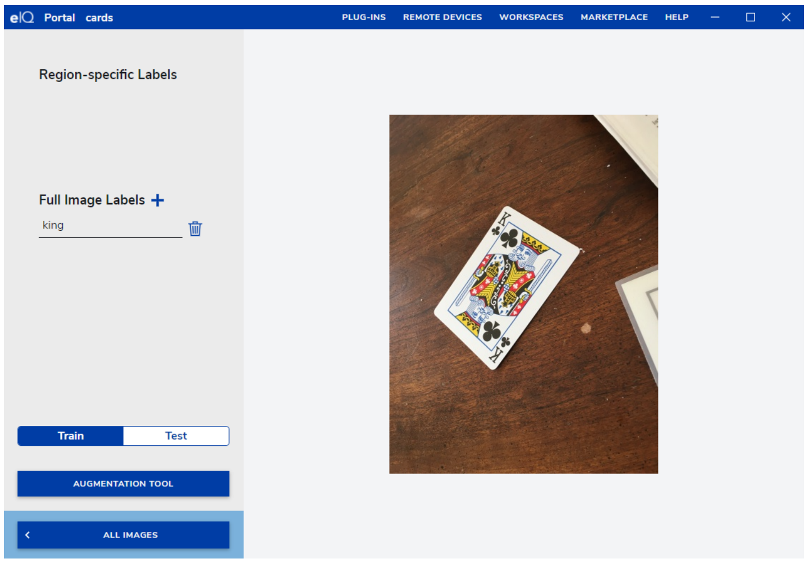


Figure 3.7.: eIQ Portal: dataset annotation

Now that you have selected a specific image, you can perform the annotations. By default, the “Bounding Box” selector is enabled so you can draw rectangles over the images to enclose the desired regions and label them.

The “Box” drawer also enables an input field that helps you to introduce the annotation inline. Notice that the labels are created once and they can be reutilized later. Also, you can check the annotations by moving the mouse over the label in the right panel. See how the bounding box is enclosed in a yellow rectangle. When you place the mouse over each annotation in the right panel, you can also delete that label.

During the annotation process for every image, add a new label for the entire image or select an existing one. Notice that the labels in the project are shared for both annotations (“Region-Specific Labels” and “Full Image Labels”). In this way, you can reutilize all the labels as well. To add a full image label, click the plus icon in the right panel ("Full Image Labels") and add a new label or select an existing one.

The "Train" and "Test" buttons in right panel assign the image to the training and testing datasets, respectively.

3.5.4 Augmentation tool

The data augmentation workspace allows you to quickly adjust the image parameters to improve model training by reducing over-fitting and increasing the robustness for dynamic real-world environments. It is widely used in machine learning to diversify the training dataset (without adding more data) by transforming a percentage of existing data into a variation, as supported by the augmentation pipeline. Using these techniques, you can generate new training samples and apply simple transformations to the original data. To apply the data augmentation, click the "AUGMENTATION TOOL" button in the "Data Set Curator" menu (see Figure 3.5.). The following screen shows the "Augmentation Pipeline" workspace with an explanation of augmentation controls:

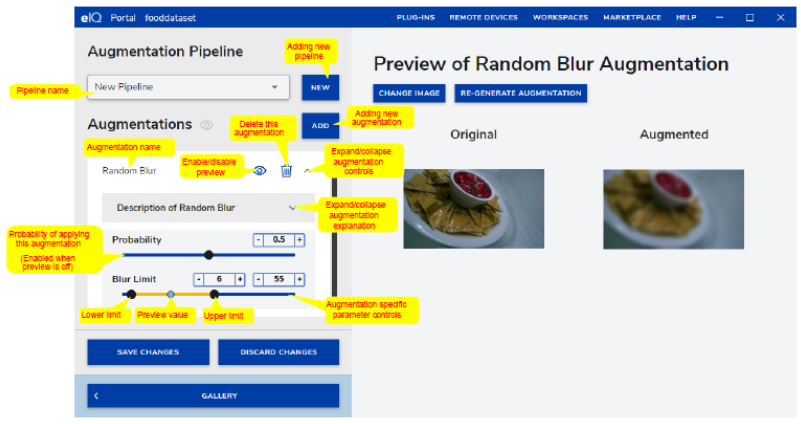


Figure 3.8.: eIQ Portal: "Augmentation Pipeline" workspace

The "Augmentation Pipeline" allows the user to combine one or more augmentation processors into a group. The default pipelines are already preset with the preset augmentations. Add a new pipeline by clicking the new button in the top left corner of the screen. Once added, you can change the name to the desired name. You can add one or more augmentations to a custom pipeline (the default pipelines cannot be edited). To do that, click the "ADD" button next the "Augmentations" area. A list with available augmentation is displayed. Click the desired augmentation to add it. The "Probability" value on each augmentation defines the percentage of images added to the training dataset with this augmentation effect. If the probability is 0.2 and if there are 100 images in the data set, then 20 images receive this augmentation in the training process per batch.

Augmentation pipelines are only executed during training. They are not used as a part of testing or validation. On top of that, for each image, for each epoch and for each step, an augmentation is generated dynamically, which means that at every epoch, the training set of images is different.

The "GENERATE NEW RANDOM AUGMENTATION" button allows you to generate a preview of how the augmented data might look like in the right-hand part of the screen.

Note: If an augmentation parameter has lower and upper limits, then the parameter has a random value between these limits.

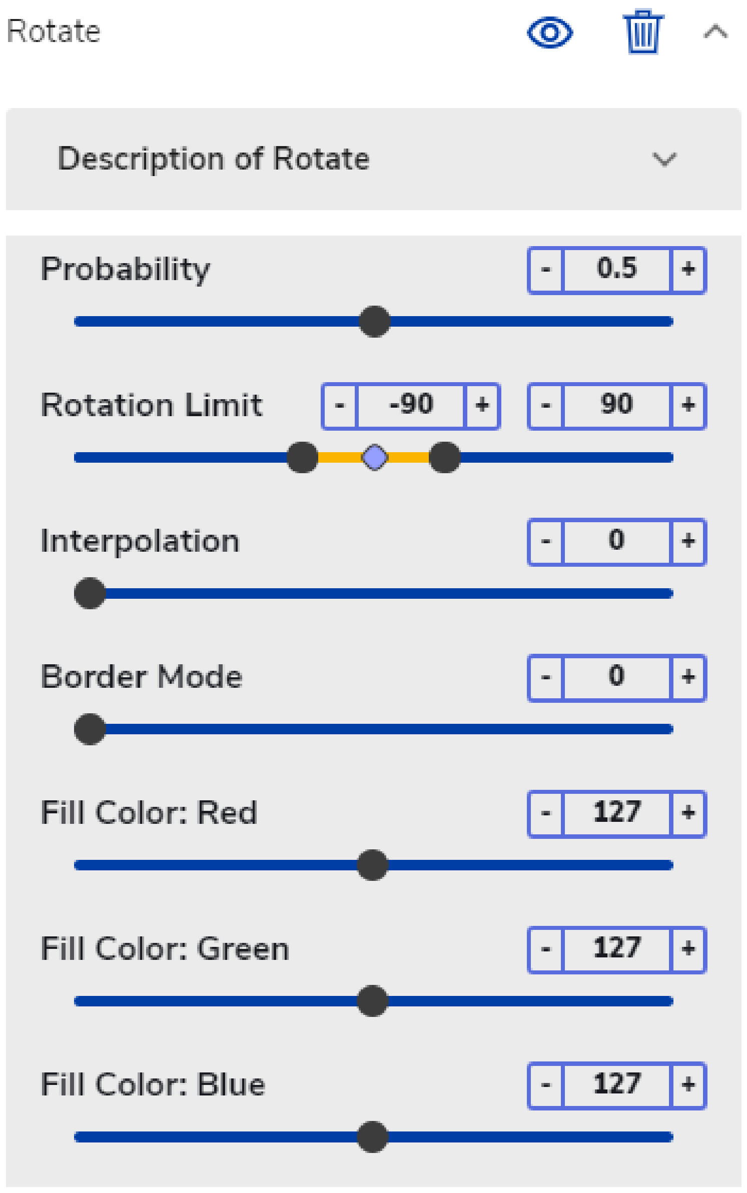
The following operators are supported by the augmentation tool:

3.5.4.1 Rotate

This operator simulates a common example in the image AI. The camera can be rotated and the object may not be aligned to an axis. For example, when a deck of cards is scattered on a table, the cards can be in any orientation.

When an image is rotated, it can create voids in the corners. This is addressed by the fill colors.

The probability slider adjusts the percentage of images that undergo this augmentation in each iteration.


Rotate

Description of Rotate

Probability: 0.5
Rotation Limit: -90 90
Interpolation: 0
Border Mode: 0
Fill Color: Red: 127
Fill Color: Green: 127
Fill Color: Blue: 127

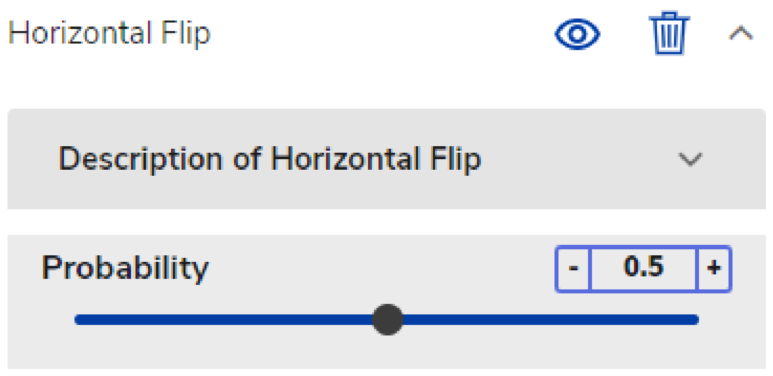
Figure 3.9.: eIQ Portal: Rotate settings

3.5.4.2 Horizontal Flip

A horizontal flip flips the image horizontally around the y-axis. This technique acts like a mirror and moves the objects on the left-hand side of the image to the right-hand side and the other way round.

This operator is useful in scenarios such as a car going from left to right or from right to left.

The Probability slider adjusts the percentage of images that undergo this augmentation in each iteration.



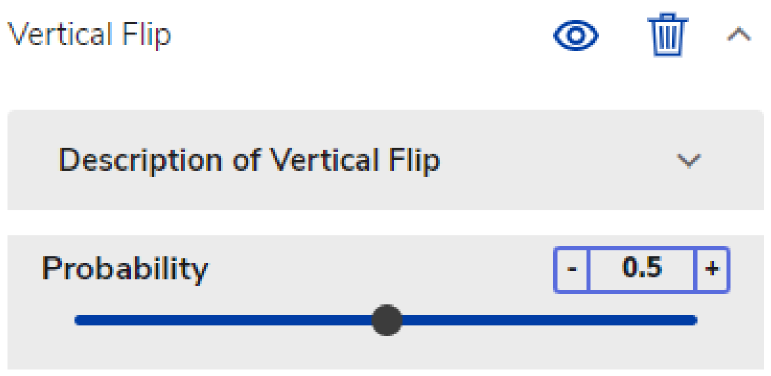
Probability 0.5

Figure 3.10.: eIQ Portal: Horizontal Flip settings

3.5.4.3 Vertical Flip

The vertical flip flips the input vertically around the x-axis.

The Probability slider adjusts the percentage of images that undergo this augmentation in each iteration.



Probability 0.5

Figure 3.11.: eIQ Portal: Vertical Flip settings

3.5.4.4 Random Blur

Image blurring is achieved by convolving the image with a low-pass filter kernel.

It is useful to simulate an out-of-focus camera or images with low resolution.

It is useful for removing noise. It removes high-frequency content (noise, edges) from the image, which results in blurred edges when this filter is applied.

The probability slider adjusts the percentage of images that undergo this augmentation in each iteration.

The blur limit controls the kernel size in pixels.

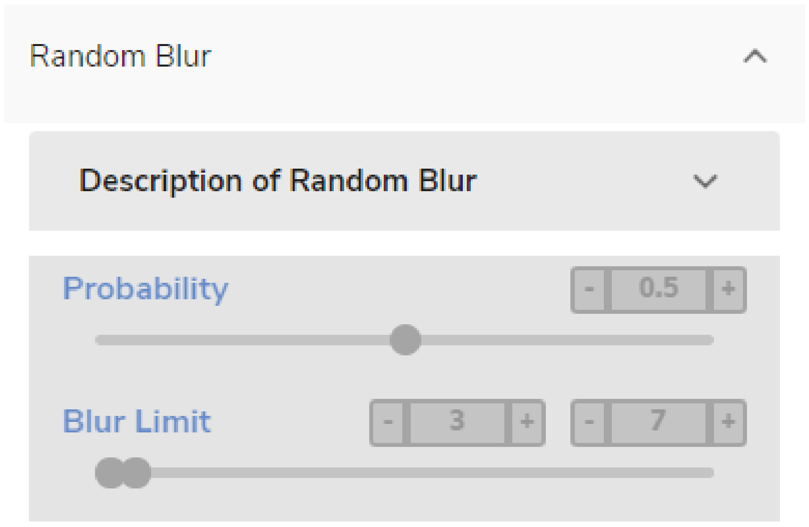

Random Blur
Description of Random Blur
Probability 0.5
Blur Limit 3 7

Figure 3.12.: eIQ Portal: Random Blur settings

3.5.4.5 Gaussian Noise

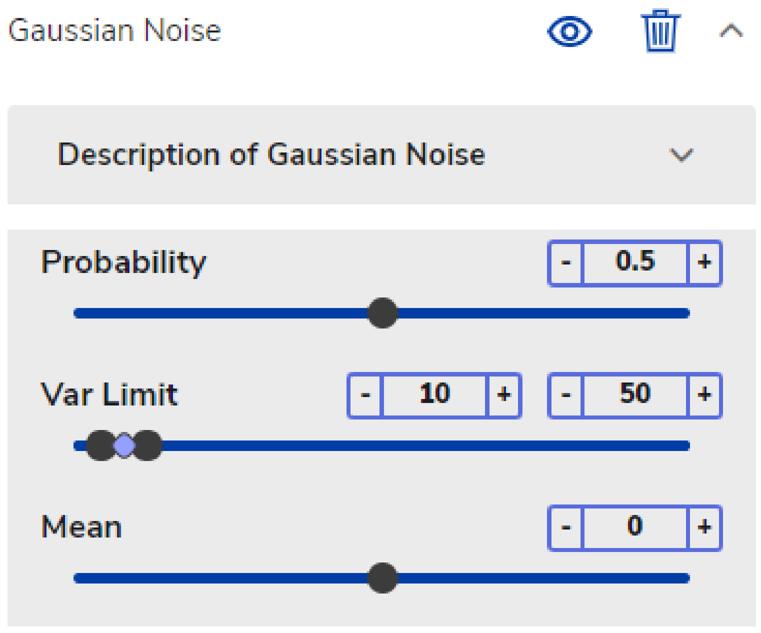
It is statistical noise with a Probability Density Function (PDF) equal to that of the normal distribution, which is also known as the Gaussian distribution.

The values that the noise can take on are Gaussian-distributed.

Cameras with low lighting conditions usually introduce Gaussian noise.

The intensity of the Gaussian noise in the augmented image is controlled by the Var Limit slider and the mean value is controlled by the Mean slider.

The Probability slider adjusts the percentage of images that undergo this augmentation in each iteration.


Gaussian Noise

Description of Gaussian Noise

Probability 0.5

Var Limit 10 50

Mean 0

Figure 3.13.: eIQ Portal: Gaussian Noise settings

3.5.4.6 Hue Saturation Value

This operator simulates the malfunction and drifting camera color control.

The Probability slider adjusts the percentage of images that undergo this augmentation in each iteration.

3.5.4.7 Random Brightness

Image brightness is governed by the light-gathering power of the lens, which is a function of numerical aperture. Just as the brightness of illumination is determined by the square of the condenser working in numerical aperture, the brightness of the specimen image is proportional to the square of the lens numerical aperture.

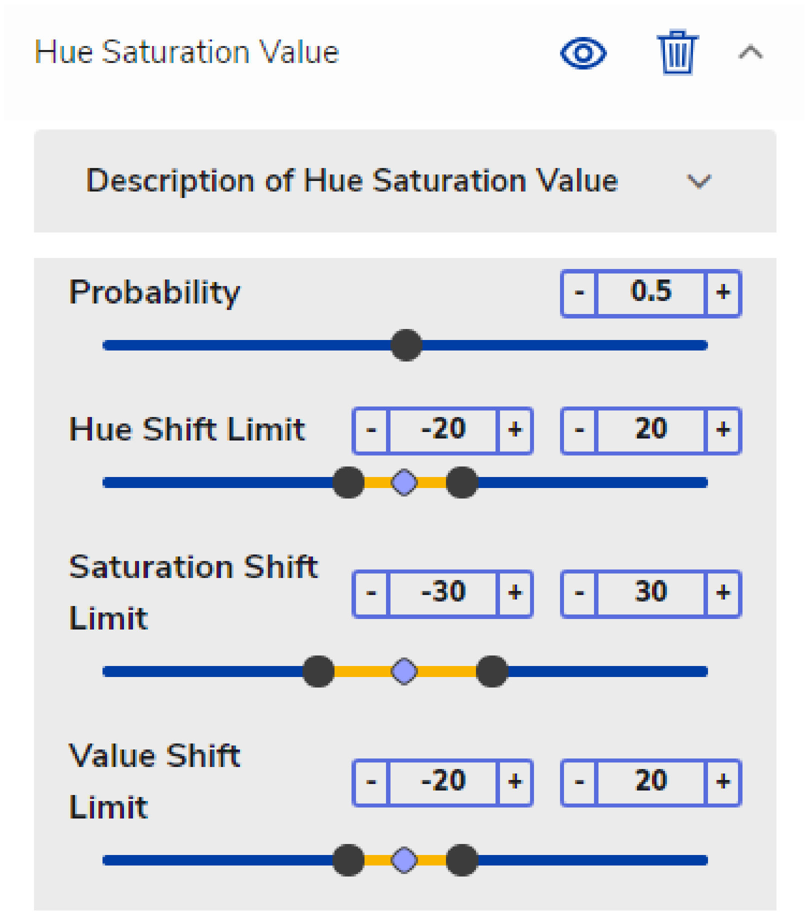
This operator can easily simulate day/evening timings, as well as sunny and cloudy conditions.

The probability slider adjusts the percentage of images that undergo this augmentation in each iteration.

3.5.4.8 Random Contrast

This operator adds random contrast to the images.

The Probability slider adjusts the percentage of images that undergo this augmentation in each iteration.



Hue Saturation Value

Description of Hue Saturation Value

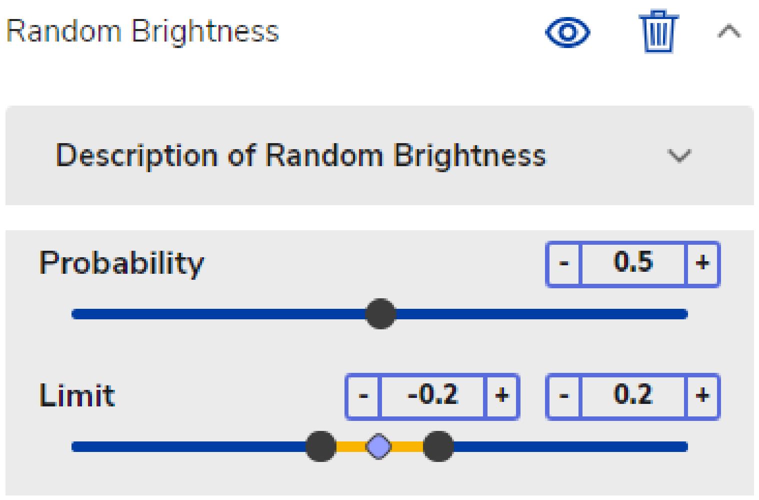
Probability 0.5

Hue Shift Limit -20 20

Saturation Shift Limit -30 30

Value Shift Limit -20 20

Figure 3.14.: eIQ Portal: Hue Saturation Value settings



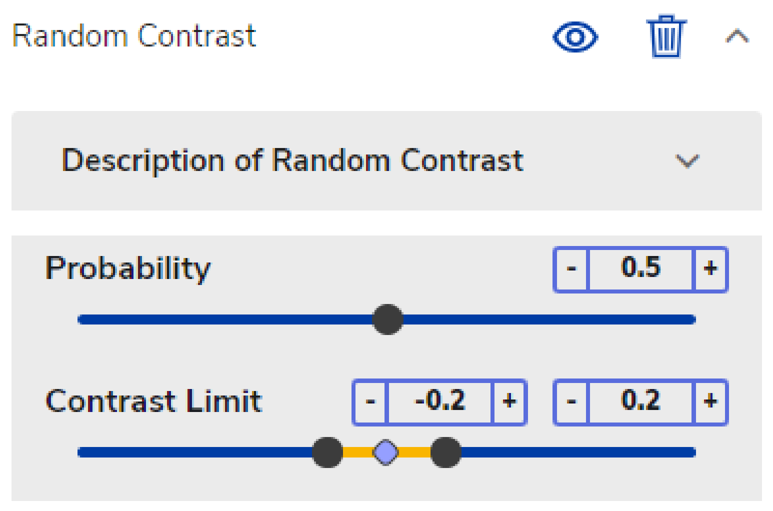
Random Brightness

Description of Random Brightness

Probability 0.5

Limit -0.2 0.2

Figure 3.15.: eIQ Portal: Random Brightness settings


Random Contrast

Description of Random Contrast

Probability 0.5

Contrast Limit -0.2 0.2

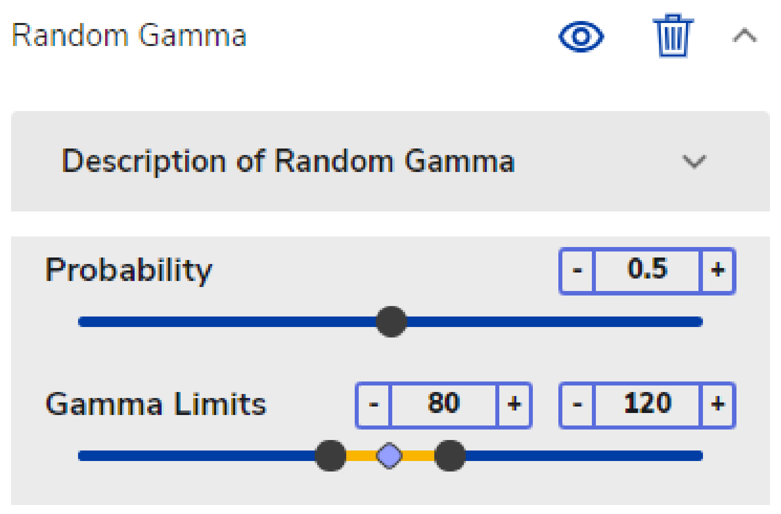
Figure 3.16.: eIQ Portal: Random Contrast settings

3.5.4.9 Random Gamma

The gamma correction or gamma is a nonlinear operation used to encode and decode luminance or tristimulus values in video or still-image systems. The gamma correction is (in the simplest cases) defined by the following power-law expression: V1 = A * V0 ^ gamma. The non-negative real input value V0 is raised to the power gamma and multiplied by the constant A to get the output value V1.

In the common case of A = 1, inputs and outputs are typically in the range from 0 to 1. A gamma value of gamma < 1 is sometimes called an encoding gamma and the process of encoding with this compressive power-law nonlinearity is called gamma compression. A gamma value of gamma >1 is called a decoding gamma and the application of the expansive power-law nonlinearity is called gamma expansion.

The probability slider adjusts the percentage of images that undergo this augmentation in each iteration.


Random Gamma

Description of Random Gamma

Probability 0.5

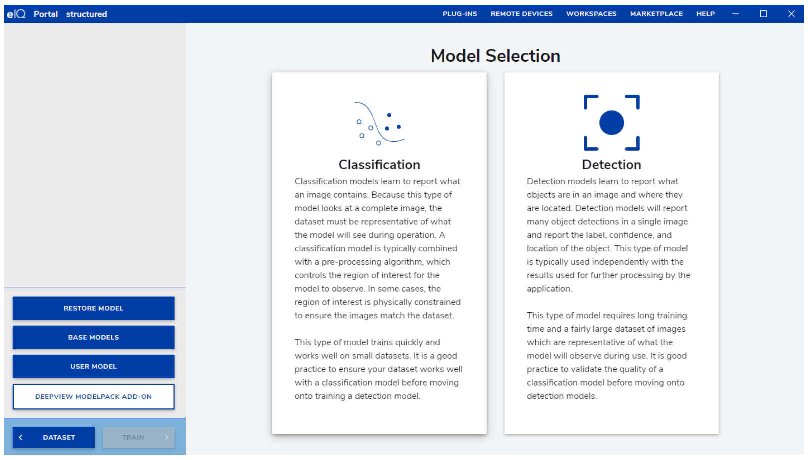
Gamma Limits 80 120

Figure 3.17.: eIQ Portal: Random Gamma settings

3.6 Model selection

The “Model Selection” workspace allows you to select the best suited model for the target. There are two ways how to proceed:

- “Wizard” – a step-by-step flow where you select the type of problem and define the target, such as a performance-oriented MCU, or an accuracy-oriented MPU. The eIQ Portal determines which model is best suited.
- “Custom Model” – a Python implementation of the model selected either from the *Base Models* (default implementations installed with the eIQ Portal) or the *User Models* (custom implementation).
- “DeepView ModelPack add-on” - offers a free trial or a commercial package with highly optimized object detection models for i.MX 8M Plus achieving over 30 frames per second. See ModelPack Quick Start Guide for more details.



Model Selection

Classification

Classification models learn to report what an image contains. Because this type of model looks at a complete image, the dataset must be representative of what the model will see during operation. A classification model is typically combined with a pre-processing algorithm, which controls the region of interest for the model to observe. In some cases, the region of interest is physically constrained to ensure the images match the dataset.

This type of model trains quickly and works well on small datasets. It is a good practice to ensure your dataset works well with a classification model before moving onto training a detection model.

Detection

Detection models learn to report what objects are in an image and where they are located. Detection models will report many object detections in a single image and report the label, confidence, and location of the object. This type of model is typically used independently with the results used for further processing by the application.

This type of model requires long training time and a fairly large dataset of images which are representative of what the model will observe during use. It is good practice to validate the quality of a classification model before moving onto detection models.

Figure 3.18.: eIQ Portal: Model Selection workspace

3.6.1 Wizard

The software interface offers two different options for tackling the computer vision tasks:

- Classification: the image classification refers to the assignment of a label to the entire image.
- Detection: the image detection refers to the regional detection of objects within the images and the assignment of a corresponding label.

For both tasks (detection and classification), you have three different options where each one is specialized in the optimization of different metrics:

- Performance: A very fast model that shows excellent performance over small problems.
- Balanced: A tradeoff between performance and accuracy.
- Accuracy: The most accurate model.

Furthermore, there are four target architectures for the eIQ Portal to be optimized for. Pick the most suitable target that fits your needs:

- NPU (Neural Processing Units) are specialized circuits optimized for machine-learning algorithms, mainly neural networks.

- GPU (Graphical Processing Units) are common on many embedded platforms and they tend to run models quickly. GPUs are less powerful than the NPUs. They are not faster than the dedicated NPUs, but faster than CPUs.
- CPU (Central Processing Units)
- MCU (Micro Controller Units) are not as powerful as other hardware units, but they make up for that in their tiny size and huge versatility.

3.6.2 Restore model

The trained model checkpoints are automatically saved for every trained epoch on the hard drive, so you can restore the progress at any point. To load a previous model, perform the following steps:

1. Click the “RESTORE MODEL” button in the “Model Selection” workspace.
2. Click the previously trained model that you want to restore.
3. Select the checkpoint that you want to restore.

3.6.3 Base models

The base models are pre-implemented popular models such as the MobileNets from which you can choose without using the Wizard. Currently, models for image classification, segmentation and object detection are supported.

1. Click the “BASE MODELS” button in the “Model Selection” workspace.
2. Select the model that corresponds to your task.

The location of base models is set by the “DEEPVIEW_PLUGINS_BASE” environmental variable. By default, it is *<eIQ_Toolkit_install_dir>\plugins*.

3.6.3.1 Segmentation models

Support for segmentation models have not yet been added to the wizard. To load segmentation models, you have to navigate into base models. There you can choose two models - UNet or FCN.

3.6.4 User models

You are not limited to the base models, but you can implement your own custom model as well. eIQ Portal provides a customizable interface for `image classification`, `image segmentation` and `object detection` problems. User plugins allow the user to add their own models from outside the tool’s installation location and use the portal to train and evaluate them, using different target architectures and quantization schemas.

1. Click the “USER MODELS” button in the “Model Selection” workspace.
2. Select the model that corresponds to your task.

3.6.4.1 Plugins folder overview

eIQ Portal can load User Models only from a folder structure with the following folder hierarchy based on the problem that you want to solve:

```
•  classification
   -  image
•  detection
   -  boxes
•  segmentation
   -  mask
•  losses
   -  boxes
   -  image
   -  mask
```

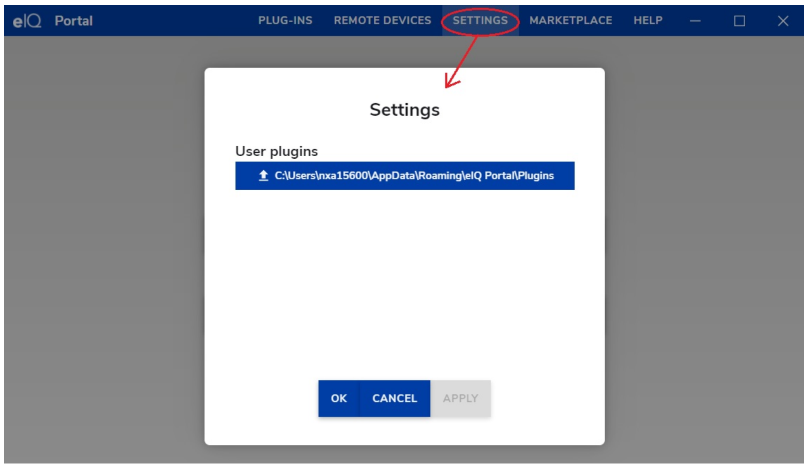
Additionally, a user loss function can be implemented in the *losses* directory.

Every user model or loss function should be stored in its own directory with its own name. An example of how to store VGG16 and VGG19 models is given below:

```
•  classification
   -  image
      -  VGG
         -  vgg16.py
         -  vgg19.py
```

You can check the plugins folder structure at <*eIQ_Toolkit_install_dir*>\plugins, where base models are located.

To set the user models directory, go to settings window (Figure 3.19.), choose a folder and apply changes. You cannot change user folder during training, validation, deployment or base model selection.



Settings

User plugins

C:\Users\nxa15600\AppData\Roaming\eIQ Portal\Plugins

Figure 3.19.: User settings

3.6.4.2 Templates

Currently, models for *image classification*, *image segmentation* and *object detection* are supported.

You can find templates with detailed descriptions in *<eIQ_Toolkit_install_dir>\content\workspace\templates* folder. Also, see *Python interface* for further details.

Additionally, you may want to implement your own *custom loss function*.

3.6.4.2.1 Image classification template

Classification models need to inherit from `deepview.trainer.extensions.interfaces.ImageClassificationInterface`. To implement a user model from a template:

1. Copy the *classification.py* file to your plugin folder (or a different template based on the problem).
2. Change the `UserClassificationModelName` class name and the return value of the `get_name` method to your model name. You may request the following URLs to make sure the name is unique:
   - `localhost:10814/v2/models/base`
   - `localhost:10814/v2/models/user`
3. Check that the optimizer, allowed dimensions and quantization settings in `get_optimizers`, `get_allowed_dimensions`, `get_qat_support` and `get_ptq_support` methods are set correctly.
4. Implement preprocess function in `get_preprocess_function` method. You need to fill `preprocess_demo` function with your code. In the default template, it returns the input.
5. Define exposed parameters for the `get_exposed_parameters` method. Add parameters you will need from eIQ Portal user. These can be used later in the `get_model` method.
6. Add your model to `get_model` method. The model has to be a Functional model or a Sequential model. It does not work for subclassed models, because such models are defined via the body of a Python method, which isn't safely serializable.

In *<eIQ_Toolkit_install_dir>\plugins* folder you should be able to find more examples that will help you better understand how to write a custom model. Simple ones are *mobilenet_v1*, *mobilenet_v2* and *mobilenet_v3*.

3.6.4.2.2 Image segmentation template

Segmentation models need to inherit from `deepview.trainer.extensions.interfaces.ImageSegmentationInterface`. To implement a user model from a template follow steps from *Image classification user model*

In *<eIQ_Toolkit_install_dir>\plugins* folder you should be able to find more examples that will help you better understand how to write a custom model. Simple ones are *unet* and *fcn*.

3.6.4.2.3 Object detection template

Object detection models need to inherit from `deepview.trainer.extensions.interfaces.ObjectDetectionInterface`.

Compared to image classification models, to add your object detection model you additionally need to implement an encoder, a decoder and most probably a user loss function (only sigmoid loss is implemented for detection models):

1. Follow steps 1 through 5 from *Image classification user model* except this time with slight variations for object detection.

2. Implement encoder in `encoder` method! Replace the method code with your implementation. The code in the template was added just to make the template trainable in eIQ Portal. It just fills `y_true` with zeros.

   Use anchors to calculate `y_true`. The shape of anchors has to correspond to the model output. Don't generate/load anchors inside encoder. The method is called many times during the training.

3. Implement decoder in `decoder` method! Replace the method code with your implementation. The code in the template was added just to make the template trainable in eIQ Portal.

4. Implement loss function and set `get_losses` and `get_loss_named_params`. Choose the supported sigmoid loss function or add your own loss function.

   You can use the combination encoder/decoder/sigmoid loss from ssd_mobilenet models. In this case, check the examples in base plugins folder and copy encoder/decoder code to your model. You may want to implement your own *loss function* as well.

5. Add your model to `get_model` method. The model has to be a Functional model or a Sequential model. It does not work for subclassed models, because such models are defined via the body of a Python method, which isn't safely serializable.

In *<eIQ_Toolkit_install_dir>\plugins* folder you should be able to find more examples that will help you better understand how to write a custom model. Simple ones are *fpn_ssd_mobilenet_v2* and *ssd_mobilenet_v3*.

3.6.4.2.4 Custom loss template

If you want to implement your own loss function, you must inherit the class from `deepview.trainer.extensions.interfaces.ObjectDetectionLossInterface`

1. Copy the *loss.py* file from *<eIQ_Toolkit_install_dir>\content\workspace\templates* to the *<eIQ_Toolkit_install_dir>\plugins\losses* directory.
2. Change the `UserLossName` class name and the return value of the `get_name` method with your loss function name. Request `localhost:10814/v2/losses` to be sure that the name is unique.
3. Implement your loss function in `__call__` method. Replace the method code with your implementation. The code in the template was added just to make the template trainable in eIQ Portal.
4. Fill `set_named_params` method if you need to use some parameters from your model in loss function. To send parameters from your model use `get_loss_named_params` method of ObjectDetectionInterface.

3.6.4.2.5 Mandatory parameters

For each model you need to define the following parameters:

- **Metrics** - are the objects that measure the performance of the models during training and let us know how well our model performs when it is evaluated on both training and validation samples. At this moment we have three implemented metrics that returns visual feedback from the training process (at least for detection models):
  - `MAP` for objects detection problems and,
  - `CategoricalAccuracy` for classification and segmentation problems

  To check the supported metrics visit: `localhost:10814/v2/metrics`
- **Optimizer** - are the functions that are in charge of the learning process. It takes the `training variables` of the models as well as the results of the `loss functions` and optimize the problem following the derivatives at each iteration. Our plugin system has support for most of the optimizers provided by the `Keras` library:
  - `SGD`

  - `RMSprop`
  - `Adam`
  - `Adadelta`
  - `Adagrad`
  - `Adamax`
  - `Nadam`

  One important point here is that optimizers are not allowed as extensions, so the user should not be able to add a new optimizer to the system (for the moment). If you want to know what optimizers are supported by out system, then visit: `localhost:10814/v2/optimizers`

- **Quantization**. In order to generate models with higher performance on edge devices we included the quantization parameters to the problem:
  - Quantization Aware Training (QAT)
  - Post Training Quantization (PTQ)

  It is responsibility of the user, to know if the model is supported by the selected quantization schema. If the selected model is not supported, the operation will return an error message

- **Loss Functions**. This is one of the most important tool inside the learning process because these particular functions are in charge of measuring how distant are our predictions from the target. These functions always receive two parameters:
  - `y_true`: this should be produced by the `encoder()` function
  - `y_pred`: this is returned by the model prediction.

  It is needed to take into consideration the fact that the `decoder()` function needs to control how the targets will look like at the time of building the model.

3.6.4.3 Dataset

eiQ Portal handles datasets in a peculiar way throughout a rest API that is controlled by the `datastore` component. From this component we can get images, annotations (boxes) and cropped boxes. In order to configure a dataset we need to identify the problem first, in our case `classification` and `detection`. To understand what each dataset type returns will help us to understand how the tool is built and will give you more control at the time of new models or loss functions development.

Classification models take the cropped boxes as images as well as the label of each box. This is preprocessed by a `tf.Data` API that creates a `classification` iterator that returns the following shape:

- **BGR image**: the true image
- **signed_normalized image**: This is the preprocessed data used as model input
- **categorical label**: This is an array of 0 elements with a 1 in the position pointed out by the class index
- **annotation id**: By using this id, you can get back the image from the `datastore` component

All these values are returned into an array with the following shape:

```
- [(batch_size, HEIGHT, WIDTH, 3), (batch_size, HEIGHT, WIDTH, 3), (batch_size,
NUM_CLASSES, 1), (batch_size, 1)]
```

Detection models take the full image and annotated boxes from the `datastore` component. This is preprocessed by a `tf.Data` API that creates the `detection` iterator that returns the following shape:

- **signed_normalized image**: This is the preprocessed data used as model input.

- **boxes**: List of max boxes allowed per images. Usually shape: `(NUM_DETECTIONS, 4)`, where NUM_DETECTIONS = 100 is the maximum number of allowed boxes per image).
- **categorical label**: This is an array of 0 elements with a 1 in the position pointed out by the class index. This time, the iterator returns as many rows as boxes are provided. Usually shape: `(NUM_DETECTIONS, NUM_CLASSES)`.
- **valid boxes**: this is a list of numbers that indicates how many boxes are valid for each image in the batch. Usually shape: `(NUM_DETECTIONS, 1)`. Notice you only need to take as many `boxes` and `labels` for each image as the `valid boxes` index indicates.

All these values are returned into an array with the following shape:

```
- [(batch_size, HEIGHT, WIDTH, 3), (batch_size, NUM_DETECTIONS, 4), (batch_size,
NUM_DETECTIONS, NUM_CLASSES), (batch_size, 1)]
```

3.6.4.4 Python interface

Classification, segmentation and detection classes are used to describe a model to provide additional information used in real time to configure the GUI. These classes are used as an interface to handle the plugin structure, organize the models according to the problem type, and handle the mandatory functionalities that each model should provide. The following is the list of important methods of these classes:

- get_name: this is the plugin name used by the GUI to distinguish between models.
- is_base: the method returns True if the model belongs to the eIQ Portal core plugins (base models) or False if not (custom models).
- get_model: the method returns a Keras model that will be used by the trainer while training is performed. In this method, we receive several parameters that help us to configure the object detection, segmentation or classification model, such as input_shape, num_classes, weights, and named_params. The model has to be a Functional model or a Sequential model. It does not work for subclassed models, because such models are defined via the body of a Python method, which isn't safely serializable.
- decoder: the method is used just for detection models (not used for classification). The method will take the outcome of the detection model and transform it into boxes and scores.
- encoder: the method is used just for detection models (not used for classification). The method transforms a dataset batch into the y_true format the loss function is expecting.
- get_loss_named_params: the method takes care about communicating the model custom parameters with the loss handler class (the bridge between the model and the loss function).
- get_metadata: the method exposes some of the constants that the converter requires for the particular model.
- get_task: returns the name of the task (detection, classification or segmentation) that the model solves.
- get_exposed_parameters: returns a list of objects which will appear as configurable settings under Model parameters tab. Defined parameters can be parsed inside the get_model method. The Optimizer parameter is recommended for all models. You don't need to parse the parameter inside the get_model method. It is processed automatically when the model is loaded into eIQ Portal.
- get_preprocess_function: the method returns a function handler used for the trainer dataset class to preprocess tensors before feeding them into the training step.
- get_losses: returns the names of all losses used for model training.
- get_optimizers: this is a helper method that returns the default optimizer (see *Model parameters description* for the list of supported optimizers). The method is used only in case there is no the Optimizer parameter in get_exposed_parameters method, otherwise the method is ignored.
- get_metrics: the method returns a metric used to measure the performance of the problem while training.

- get_allowed_dimensions: the method tells the GUI that our model supports any input dimension included into the returned list.
- get_pretrained_dimensions: this method introduces the set of dimensions with pretrained weights and the source of the weights.
- get_qat_support: the method tells the GUI if the model supports Quantization Aware Training (QAT) or not. If the model supports QAT, then we can provide the input/output types and the framework where the per-channel or per-tensor quantization is provided.
- get_ptq_support: the method tells the GUI if the model supports Post-Training Quantization (PTQ) or not. If the model supports PTQ, then we can provide the input/output types and the framework where the per-channel or per-tensor quantization is provided.

Note: For the moment only two frameworks are supported: TensorFlow and Converter. The per-tensor quantization is only supported by the Converter.

The templates located in *<eIQ_Toolkit_install_dir>\content\workspace\templates* are well documented and provide more details.

3.6.4.5 Import additional libraries and modules

In order to import new functionalities from additional modules in the current directory, our plugin system is able to load the extensions as a big module that is headed by the name of the problem type `classification`, `segmentation` or `detection`. By using this you only need to import package in the right order since the plugins directory is included into the `sys.path` python libraries. The path is relative to the problem type path that contains all the extensions.

For example, suppose you have the following code that handle the preprocess input for a VGG models:

```
def vgg_preprocess(X):
    """
    returns the image in the range [0..255] but using float type
    """


return tf.cast(X, tf.float32)
```

The code is stored in a file named `utils.py` and that file is located in the following position:

`classification`

- `images`
  - `VGG`
    - `vgg16.py`
    - `vgg19.py`
    - `utils.py`

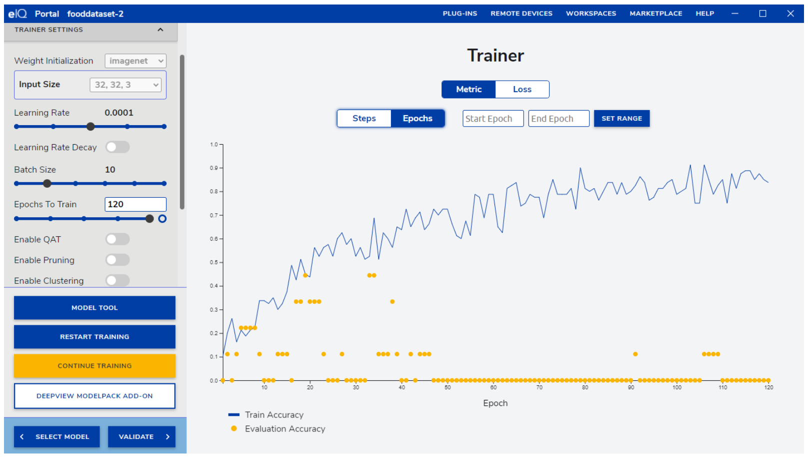
To call `vgg_preprocess` inside VGG modules, you only need to add the following import to `vgg16.py` or `vgg19.py`, whenever you want to use it.

```
from classification.images.VGG import utils
```

To use the function inside the `utils` module, then call `utils.vgg_preprocess(...)`

3.7 Training

The models gain the capability to predict by viewing annotated samples and adjusting a cost function. This process is also known as training. You can go to the training window by clicking the “TRAIN” button in the previous model selection menu (see also Figure 3.18.). The eIQ Portal provides a user-friendly interface for training models. See the left-hand part of the following figure:



Weight Initialization: imagenet
Input Size: 32, 32, 3
Learning Rate: 0.0001
Learning Rate Decay
Batch Size: 10
Epochs To Train: 120
Enable QAT
Enable Pruning
Enable Clustering

Figure 3.20.: eIQ Portal: training

3.7.1 CUDA acceleration

The eIQ Toolkit and TensorFlow accelerate the training using CUDA® by default (if available). The installation of CUDA is currently not distributed as a component of the eIQ Toolkit, so it is up to the user to install it. To do that, both CUDA and cuDNN must be installed. Download them from the NVidia Developer website (https://developer.nvidia.com/cuda-zone).

The cuDNN package is distributed as an archive. For Windows, the DLL files must be extracted either to the eIQ Toolkit “bin” directory (together with “deepview-trainer.exe”) or to a path visible to the OS by default.

3.7.2 Training settings description

With the model selected, the interface shows you the possible parameters that you can modify to achieve the best fit of the model over your training data:

- **Weight initialization:** Sets the initial values of the weights before training. The primary function of the training is to adjust model weights to achieve a higher accuracy. The closer the weights are initialized to the post-training weights, the faster the model becomes optimal. By default, the initial model is trained on the ImageNet dataset for faster training (based on popular MobileNet models). You can also select a random initialization or initialize the weights’ values from a file.
- **Input size:** The input resolution for your model. Images are automatically resized to the selected input size (if needed). Increasing this value also increases the training/inference time. The first two numbers of each option refer to the horizontal and vertical dimensions of the input image. The third number refers to the number of input channels.
- **Learning rate:** Controls how much the weights of the model are changed after every batch. When it is too low, the model learns very slowly and it can get stuck on the wrong answer. When it is too high, the model can change

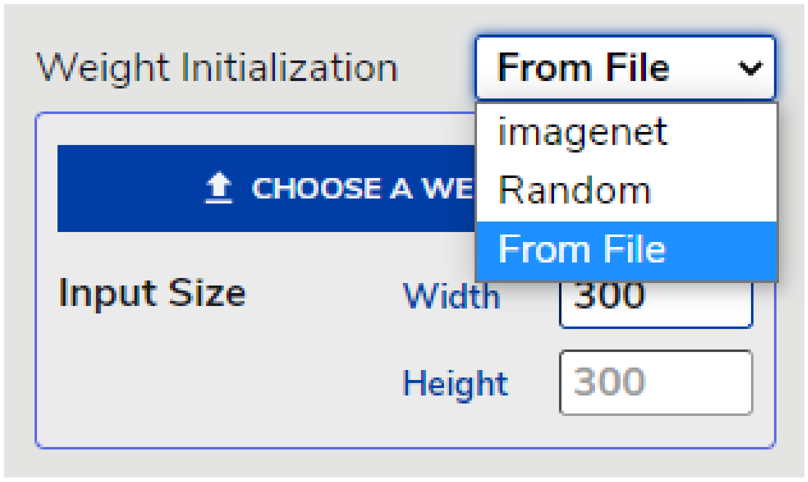


Figure 3.21.: Values from a file

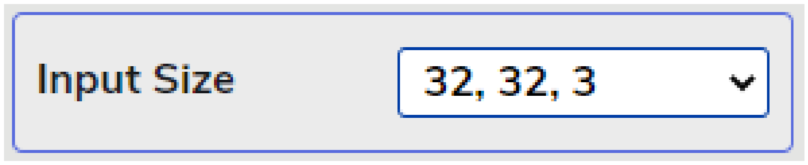


Figure 3.22.: Input size

its weights too much, which could prevent it from learning the correct answer. The lower the learning rate, the more epochs should be set. Typically, the low learning rate should lead to setting more epochs.

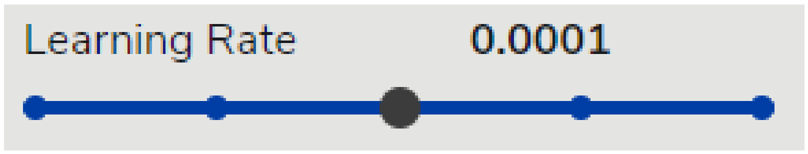
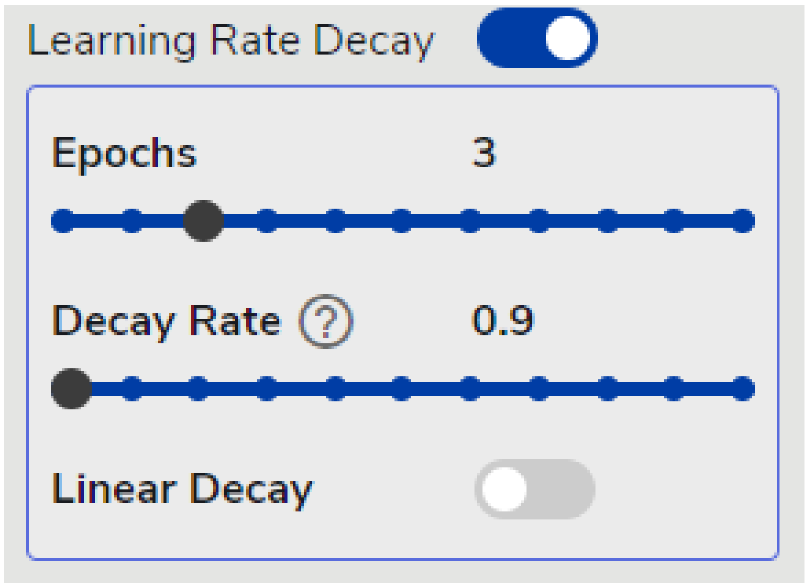


Figure 3.23.: Learning rate

- **Learning rate decay:** Changes the learning rate during learning. This is typically used when a model starts learning with a bigger learning rate and then it is slowly decreasing during training. This technique helps the model to learn and generalize better. The epochs parameter set for the learning rate decay sets for how many steps the decay will be applied. You can either choose a number of epochs and the steps will be computed as epochs*steps_per_epoch. You can select a specific number of steps if you drag the slider all the way to the right. The parameter decay rate defines how quickly the learning rate decreases. The lower the number, the quicker it decreases.
- **Batch size:** Sets the number of samples that the model tries to estimate/predict before updating its weights. For a dataset with many similar images, larger batch sizes are more likely to reach a higher accuracy faster. For smaller datasets and datasets with dissimilar images, smaller batch sizes are likely to reach a higher accuracy faster.
- **Epochs to train:** Sets the number of times that the model goes through the whole training dataset. By increasing this value, the model weights have more chances to learn the important features to extract. When too many epochs are used, the training may cause your model to overfit the training dataset. An epoch consists of a series of batches. This parameter is typically used as the stopping condition for the training.

  To set the number of epochs to train, drag the “Epochs to Train” slider to the right. To select the desired stopping condition, click the “Infinite Epoch” button at the right-hand side of the slider. It looks like an empty circle with


Learning Rate Decay

Epochs 3

Decay Rate 0.9

Linear Decay

Figure 3.24.: Learning rate decay

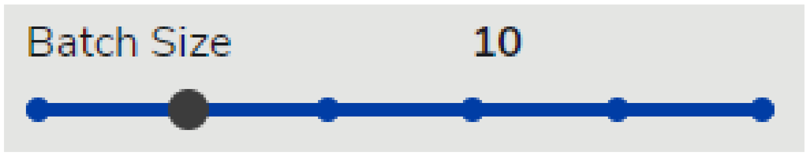

Batch Size 10

Figure 3.25.: Batch size

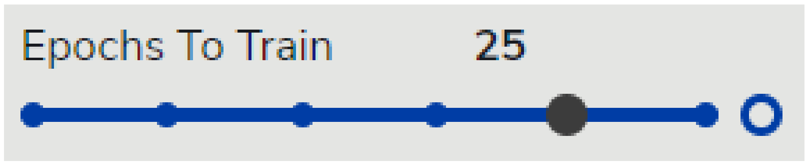
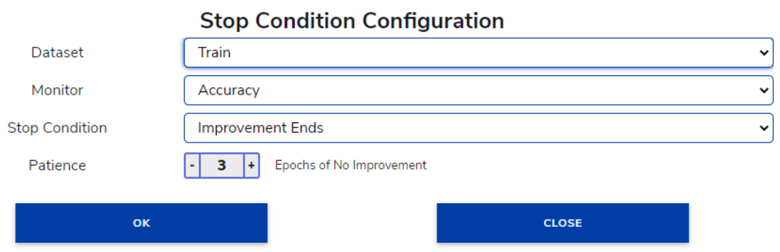

Epochs To Train 25

Figure 3.26.: Epochs to train

a blue border. The “Set Stop Condition” button appears. After clicking it, you can see the following dialog:


Stop Condition Configuration

Dataset: Train
Monitor: Accuracy
Stop Condition: Improvement Ends
Patience: 3 Epochs of No Improvement

Figure 3.27.: Stop condition configuration

You can choose whether the stop condition is applied on the training or testing datasets. You can monitor either the metric or the loss. The stop condition can be set to three different values:

- Improvement Ends causes the training to stop if there was no improvement on the monitored value for a certain number of epochs.
- Stop At Target causes the training to stop if the monitored value reached your target value.
- Improvement Ends After Target causes the training to stop if the monitored value does not improve for a certain number of epochs after reaching the target value.

- **Enable QAT:** This option enables additional model quantization with the number of quantization epochs. It is available for model classification only.

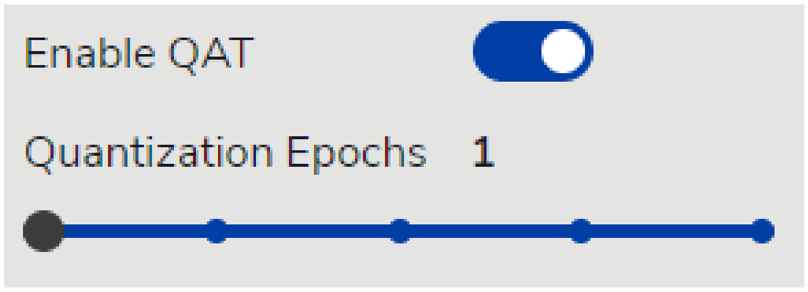
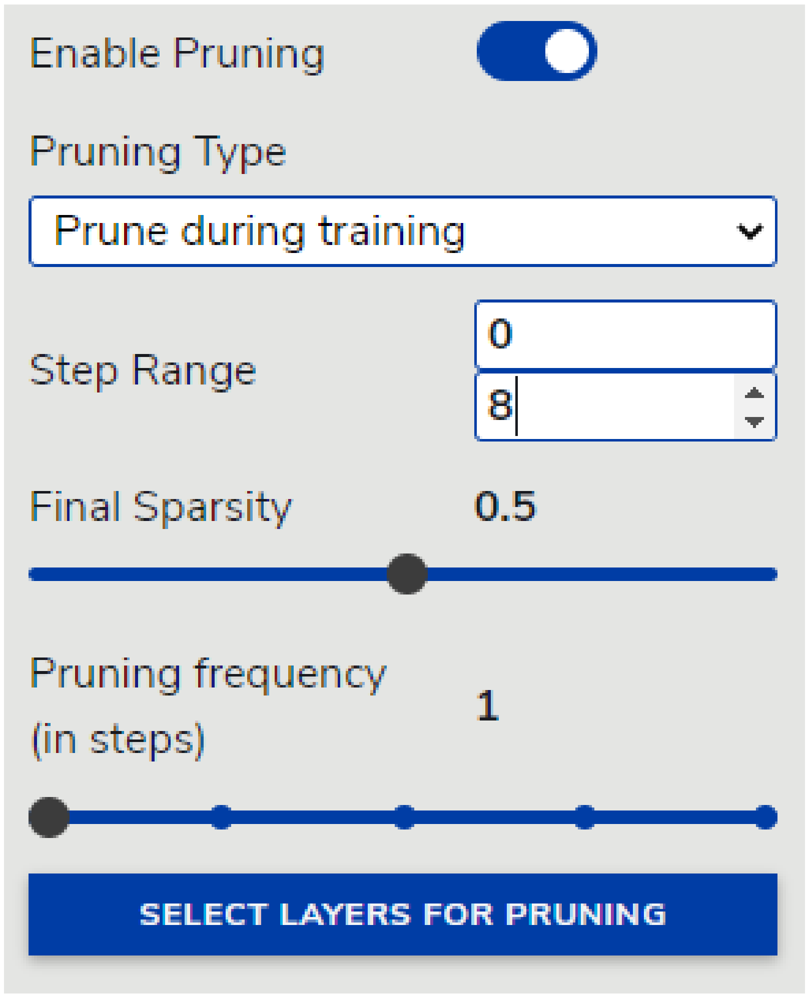

Enable QAT
Quantization Epochs 1

Figure 3.28.: Enable QAT

- **Enable pruning:** This option enables pruning. Pruning is used to train a model in such a way that some of the weight values are set to zero. This potentially decreases the model size while still maintaining a good performance. Note that the model size decreases when the model is compressed (e.g. via gzip). A model that is not compressed has zero values stored as any other number. This is why you do not notice the change. Using models that are pruned and/or compressed can then decrease the memory consumption on the target device and decrease the inference time. Note that leveraging this feature is not currently possible with the prepared inference engines, but it is planned to include it in future products.

  When the pruning is enabled, there are a couple of options:

  - Pruning Type can be set to either “Prune during training” or “Fine tune post training”. The first option executes the pruning while the model is trained from scratch (or with pre-trained weights). “Fine tune post training” first trains the model and this trained model is trained for another several epochs during which the pruning is applied. If you choose the “Fine tune post training” type, another option that sets how many epochs of fine-tuning are done appears.
  - Step range option sets the beginning step and end step of the pruning. During the steps between start and end, the pruning is applied. The top value (default is 0) sets the beginning step, the bottom value (default is number of steps per epoch, in the picture it is 8) sets the end step value. The step count for pruning is


Enable Pruning

Pruning Type: Prune during training

Step Range: 0 – 8

Final Sparsity: 0.5

Pruning frequency (in steps): 1

Figure 3.29.: Enable pruning

separate from the step count of the whole training. Every time you start training with pruning, the step count starts from 0 and continues. Unless you want to postpone the pruning by setting the begin step to a higher value, you do not have to change this at all. However, pay close attention when setting the end step value. The end step value means that the pruned model will reach the final sparsity at this step. After reaching this step during training, pruning does no longer have effect. If the training ends earlier for some reason, or frequency is so high that it overshoots the end step (for example, setting the end step to 4 and the pruning frequency to 5), the sparsity of the model is lower. It may be 0, which means that the pruning has no effect at all.

- Final Sparsity option sets what percentage of weights should be set to zero when the pruning is finished. For example, the Final Sparsity in the picture is set to 0.5, that means that approximately 50 % of the weights are set to zero.
- Pruning frequency sets how often the pruning is applied during the training. If it is set to 5, the weights with pruning are recomputed on steps 5, 10, 15, and so on. During other steps, the training continues without pruning. A good practiceis to leave some space between each application of pruning (apply it for example every 100 steps) so that the model can better adjust to a weight change and the accuracy loss is not that significant. It is also a good practice to align the pruning frequency to the end step value, so the final sparsity is reached.
- Pruning epochs sets the number of epochs when the pruning is applied as fine-tuning.
- The last option is to select layers for pruning. Pruning does not have to be applied to all the layers in the model. In fact, it is desirable to select only several layers that are pruned and other keep the other ones intact. When you click the “Select Layers for Pruning” button, it opens the Model Tool in a special mode.

  This mode does not display any detailed information when clicked on a layer. Instead, it highlights the layer, as you can see in the picture. This indicates that the layer was selected for pruning. If no layers were selected, the pruning is applied to the whole model. If some of the layers were selected, the pruning applies only to those layers. There are some layers where it does not make sense to apply pruning and thus there are no changes when inspecting the weights of these layers. The most common of these layers are *Depthwise Convolution*, and *Batch Normalization*. Pruning is not applied to biases in any layers, only to kernels. When applying pruning, there are guidelines to follow for the best results:
- Applying pruning post training is better than during training.
- Layers at the end of the model are pruned.
- Try to avoid pruning layers that reduce the accuracy the most (for example attention).

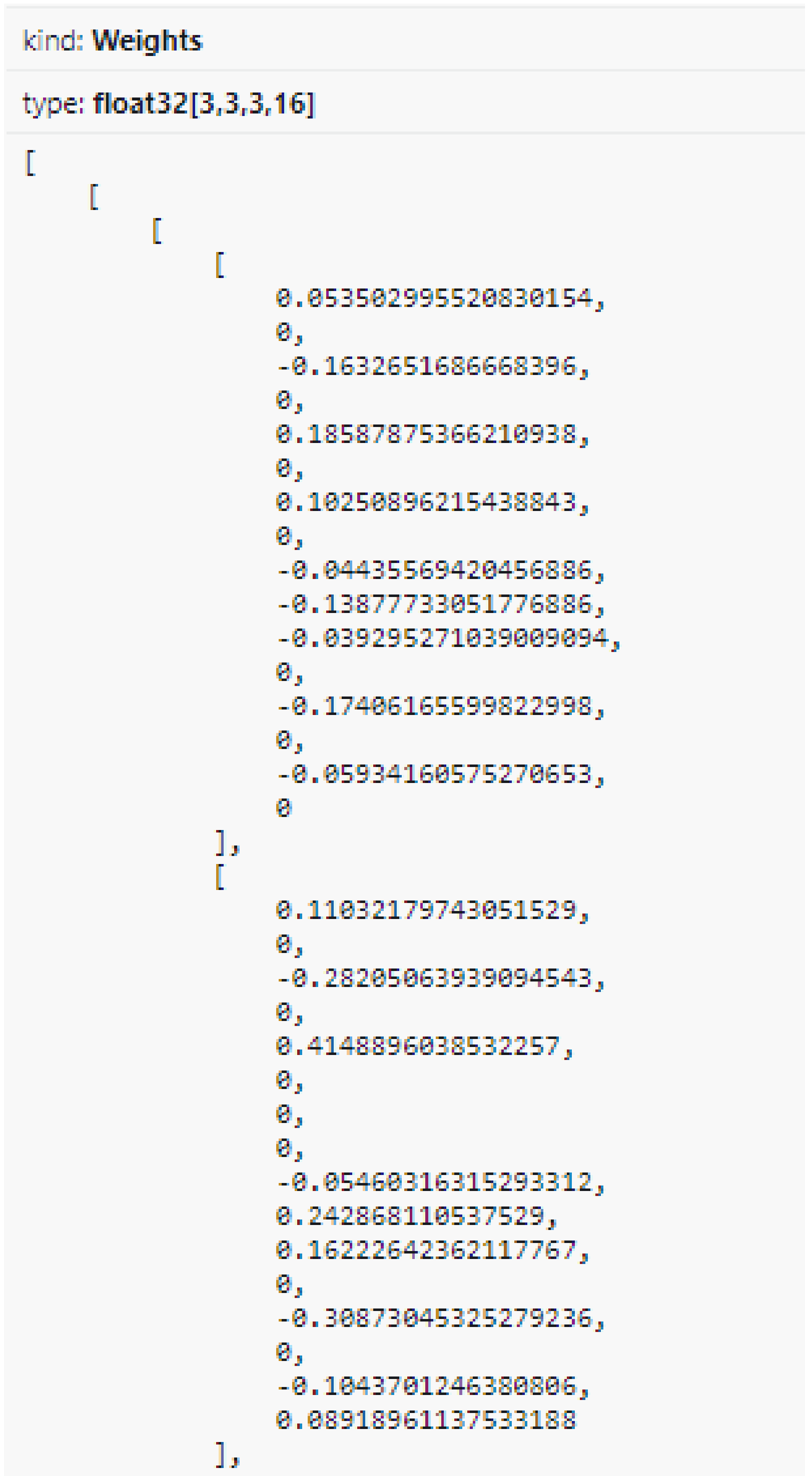
  Do not apply pruning very frequently. It is good to give the model some time to adjust to a change where some of the weights were suddenly set to zero.

  After the training with pruning is finished, the resulting weights in the trained model can look like those in the picture. The model was trained with the final sparsity of 0.5.

- **Enable clustering**: Weight clustering has a similar purpose to pruning. It changes the weights, so the model size (when compressed) is smaller. Clustering achieves this by applying clustering algorithm to each weight tensor. Clustering algorithm tries to group the weights that are similar and replace them with one value (cluster centroid). When the clustering is applied, all weights that belong to one group are replaced by the same value. There is only a finite number of different values within the weight tensor. This number is called “number of clusters”. This processing can then lead to model size reduction. The settings for this feature are less complicated than the ones for pruning:

  You can choose the number of clusters and clustering epochs. Clustering is always applied after the model has already been trained. By choosing the number of epochs, you choose how long the model is fine-tuning with clustering. Similar to pruning, you can also choose layers that are clustered. Similar guidelines to pruning apply also for clustering:

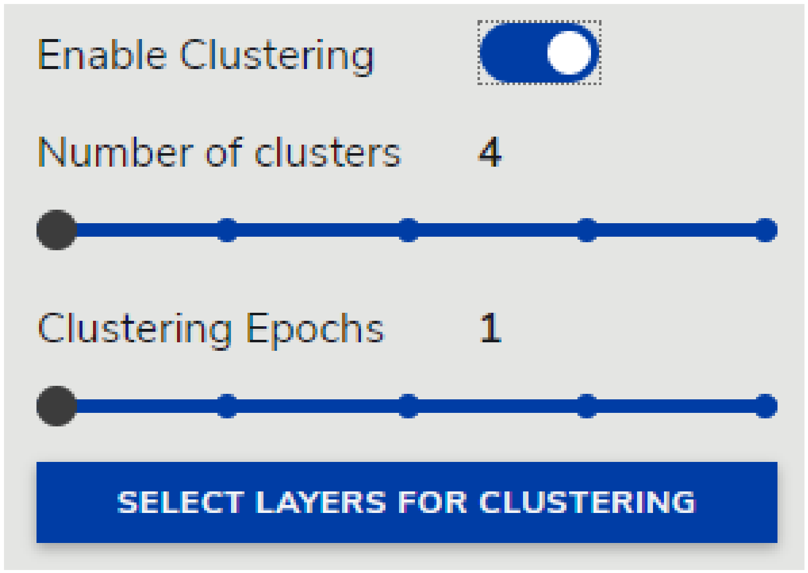
  - It is better to cluster later layers in a model.


kind: **Weights**

type: **float32[3,3,3,16]**

```
[
    [
        [
            [
                0.053502995520830154,
                0,
                -0.1632651686668396,
                0,
                0.18587875366210938,
                0,
                0.10250896215438843,
                0,
                -0.04435569420456886,
                -0.13877733051776886,
                -0.039295271039009094,
                0,
                -0.17406165599822998,
                0,
                -0.05934160575270653,
                0
            ],
            [
                0.11032179743051529,
                0,
                -0.28205063939094543,
                0,
                0.4148896038532257,
                0,
                0,
                0,
                -0.05460316315293312,
                0.242868110537529,
                0.16222642362117767,
                0,
                -0.30873045325279236,
                0,
                -0.1043701246380806,
                0.08918961137533188
            ],
```

Figure 3.30.: Model with pruned weights



Enable Clustering

Number of clusters 4

Clustering Epochs 1

Figure 3.31.: Enable clustering

- Avoid clustering layers that affect accuracy the most (attention, layers most related to feature extraction, and so on).

  After the training with clustering is finished, the resulting weights in the trained model look like those in the picture. The model was trained with 4 clusters:

  It is possible to combine pruning and clustering, but the eIQ Portal does not currently support this. If you enable clustering, you will not be able to choose pruning.

3.7.3 Augmentation settings

These settings allow you to select the augmentations you might have prepared with the augmentation tool (see *Augmentation tool*). If you choose a pipeline, the augmentations are applied to the images during training. This results in each epoch having a slightly different dataset. Using augmentations, you make the original dataset larger, which can then cause that your trained model achieves a better performance.

3.7.4 Model parameters description

The Model parameters section displays all the exposed parameters defined in the user or base models (see *Python interface*).

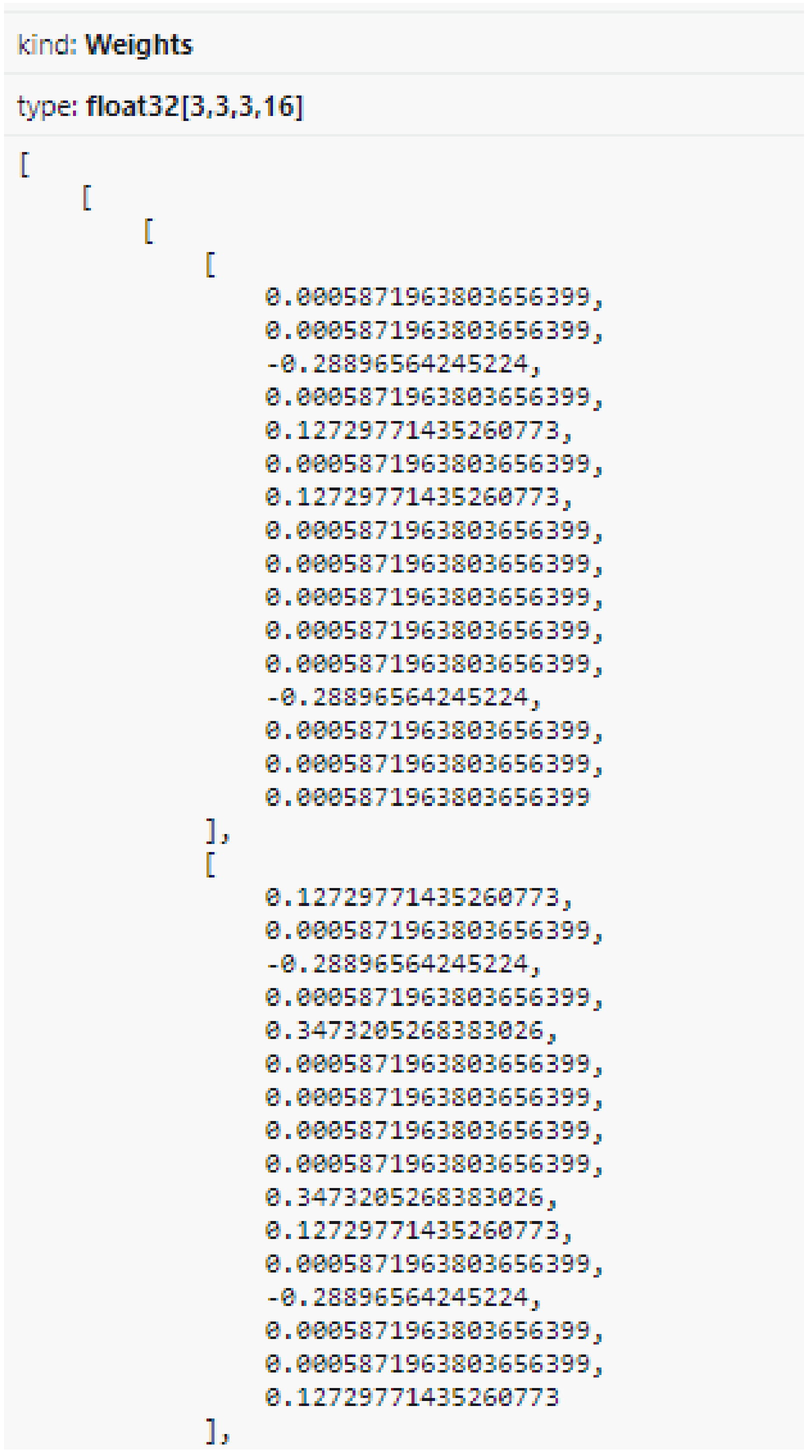
The basic parameters are as follows:

- **Alpha:** Alpha is a multiplier that affects the number of filters in each layer of the model.

  Alpha is not a mandatory parameter. There are some models without Alpha setting in the eIQ Portal. If you define the Alpha parameter for your model you have to process the value during model definition.

- **Optimizer:** Optimizers are mathematical formulas that, when applied to a weight, find the best possible weight in as few adjustments as possible.

  The Optimizer parameter is not a mandatory parameter, but it is recommended for all models. You don't need to parse the parameter inside the `get_model` method, it is processed automatically when the model is loaded into



kind: **Weights**

type: **float32[3,3,3,16]**

```
[
    [
        [
            [
                0.0005871963803656399,
                0.0005871963803656399,
                -0.28896564245224,
                0.0005871963803656399,
                0.12729771435260773,
                0.0005871963803656399,
                0.12729771435260773,
                0.0005871963803656399,
                0.0005871963803656399,
                0.0005871963803656399,
                0.0005871963803656399,
                0.0005871963803656399,
                -0.28896564245224,
                0.0005871963803656399,
                0.0005871963803656399,
                0.0005871963803656399
            ],
            [
                0.12729771435260773,
                0.0005871963803656399,
                -0.28896564245224,
                0.0005871963803656399,
                0.3473205268383026,
                0.0005871963803656399,
                0.0005871963803656399,
                0.0005871963803656399,
                0.0005871963803656399,
                0.3473205268383026,
                0.12729771435260773,
                0.0005871963803656399,
                -0.28896564245224,
                0.0005871963803656399,
                0.0005871963803656399,
                0.12729771435260773
            ],
```

Figure 3.32.: Clusters

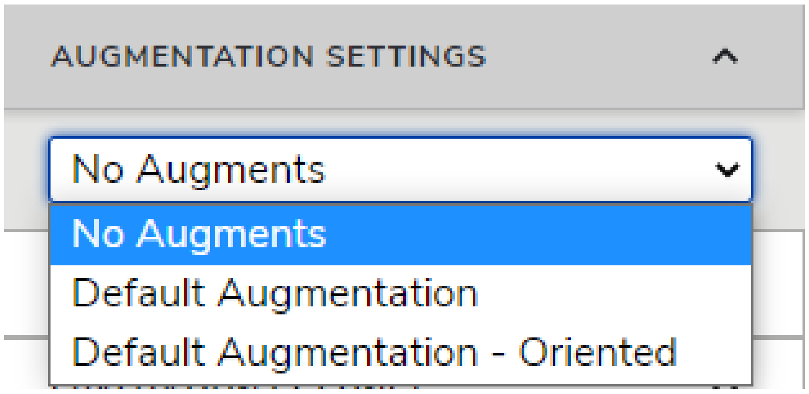
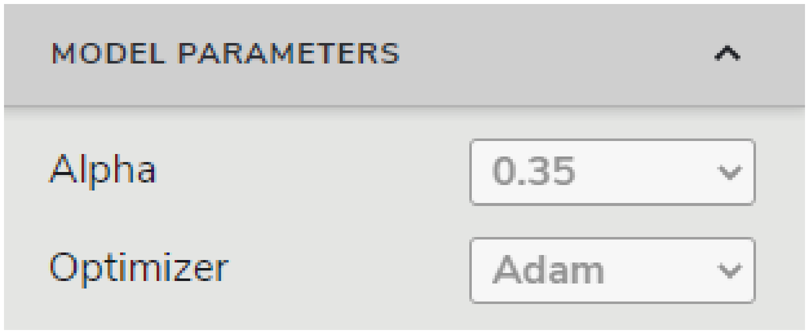


Figure 3.33.: Augmentation settings



Alpha: 0.35
Optimizer: Adam

Figure 3.34.: Model parameters description

eIQ Portal. In case the Optimizer parameter is not defined a default optimizer from *`get_optimizers`* method is used.

The following optimizers are available:

- **Stochastic Gradient Descent (SGD)** is a method to find best weight vectors in a model adjusting one direction and seeing if the output improved. SGD is great at quickly training a model when the model is far from ideal.
- **Adagrad** is similar to SGD in how it calculates the direction to adjust weights. It is different in that it uses the magnitude of the previous changes when adjusting a weight to determine how much to change that weight in the next train step.
- **Adadelta** is an extension of Adagrad that only collects a fixed number of past adjustments, rather than collect them all.
- **Root Means Square propagation (RMSprop)** is similar to SGD in how it calculates the direction to adjust weights. It is different in that it treats each parameter of a weight separately. This helps to prevent the oscillation on a single axis. In nearly all cases, RMSprop performs better than Adagrad or SGD.
- **Adaptive Movement Estimation (Adam)** combines the benefits of Adagrad and RMSprop. It accumulates an exponentially decaying average of past adjustments to evaluate the next adjustment. Adam is better than SGD and Adagrad in almost all cases and better than RMSprop in most cases. It combines the faster gradient descent of Adagrad with the oscillation-dampening learning rate of RMSprop.
- **Adamax** is a generalized version of the Adam optimizer, which uses "infinity norms". The fewer trainable parameters are (the simpler the model is), the more accurate this optimizer becomes. Adamax can outperform Adam with particularly small models. With image classification (the models are typically

large) and even with small models, Adam occasionally outperforms Adamax. With larger models, Adamax performs similarly to Adam, so it is an optional alternative.

- **Nesterov-accelerated Adaptive Movement Estimation (Nadam)** combines the Adam optimizer with the Nesterov accelerated gradient technique. This causes the next adjustment to be calculated as if the current trend of adjustment were to continue with another step.

Note: The best optimizer for any situation varies with the datasets. However, good choices for most image classification models are Adam, Nadam, and Adamax.

3.7.5 Recommended settings

To create a new model from a dataset in the eIQ Portal, the recommended settings to start from are as follows:

- Imagenet weight initialization
- Input size of 128, 128, 3
- Learning rate of 0.0001
- Batch size of 10
- Alpha of 0.50
- Adam Optimizer

If (after some training) the accuracy is too low with these settings, the input size and the Alpha can be incrementally increased to desirable results. If (after some training) the inference time is too long, the input size and the Alpha can be incrementally decreased.

3.7.6 Execution

You can start the training process by clicking the “START TRAINING” button. You can stop the training process whenever you want by clicking the “STOP” button and restart the training by clicking the “CONTINUE TRAINING” button (see Figure 3.20.).

3.7.7 View results

During the training process, two metrics are collected and shown in the “Trainer” panel in real time:

- **Metric:** How accurate the predictions of the current model are (see Figure 3.35., the left-hand part).
- **Loss:** How distant your predictions are from the ground truth (see Figure 3.35., the right-hand part).

You can switch between the two metrics by clicking the “Accuracy” and “Loss” buttons, respectively. When hovering the mouse cursor over the metric graph, you can see the metric of the test and train datasets at a particular time step. To zoom the metric graph, enter “Start Step” and “End Step” and click the “SET RANGE” button.

If you are satisfied with the results, click the “VALIDATE” button to go to the next window.

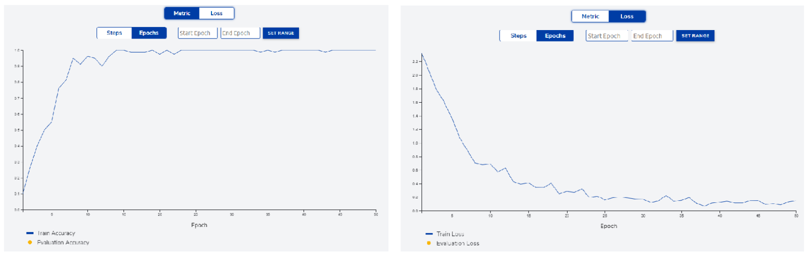


Figure 3.35.: eIQ Portal: accuracy vs. loss function of a trained model

3.8 Validation

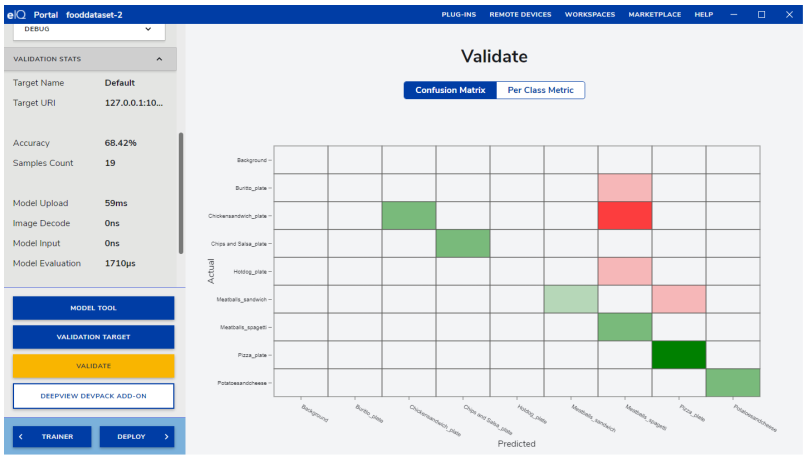
The validation of the model is a very important step when training supervised classifiers. This process gives you a measure of how accurate the model is on images it never saw in the training phase. The eIQ Portal offers a simple interface that allows you to deploy and evaluate the model on a target/remote device. This feature is enabled just after the first training epoch is completed by the training process.

By clicking the VALIDATION TARGET button (see Figure 3.36.), a modal window appears to insert new target devices or select an existing device to perform the validation on. You can also choose to not select a target and validate the model on your local host (your computer) by default.

Clicking the “VALIDATE” button (see Figure 3.36.) starts the validation process.

The validation converts the model and runs it on the target architecture to see the real performance of the model. When the validation ends, you can see the confusion matrix, which is fully interactive and represents the distribution of the test dataset based on classification results. Notice that we added a background class to map the values that are classified correctly but have a confidence below the “Softmax Threshold” slider. You can also click any cell and see what instances were recorded in it.

The interface also allows you to adjust the minimum confidence of the predictions and update the confusion matrix in real time by moving the “Softmax Threshold” slider. The “Softmax threshold” slider enables you to determine what is the acceptable limit of the prediction accuracy. Move the threshold up and down and notice how it affects the cell colors, which also relate to the number of passing instances in a cell.



Target Name	Default
Target URI	127.0.0.1:10...
Accuracy	68.42%
Samples Count	19
Model Upload	59ms
Image Decode	0ns
Model Input	0ns
Model Evaluation	1710μs

Figure 3.36.: eIQ Portal: validation

Note: The confusion matrix is used to match the ground truth (the actual label) to the predicted label. For the best validation results, you need all cells along the diagonal to have a dark green color. The darker the green color, the more instances are associated with that cell. The darker the red color, the higher the number of mispredictions.

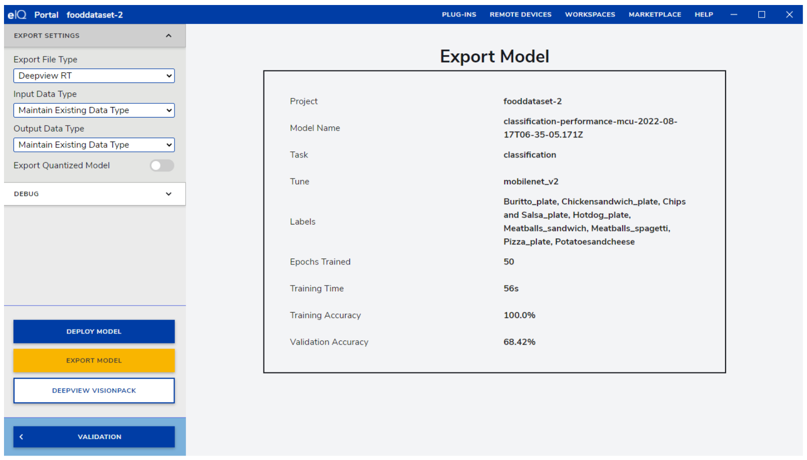
3.9 Deployment

If you are satisfied with the validation accuracy of your model, you can export it to a highly optimized format capable of running in the selected inference engine. Click the “DEPLOY” button (see Figure 3.36.) to move into the “Export Model” workspace.

The Export Model workspace summarizes the training/validation results, as well as the configurations of the project. To export the model, click the “EXPORT MODEL” button, select the destination, and save the model.

Note: The model can be exported to different formats, such as a DeepView file (*.*rtm*), a TensorFlow Lite file (*.*tflite*), an ONNX file (*.*onnx*), or a Keras file (*.*h5*).

When the model is exported, you see two more buttons in the left-hand pane. By clicking the “SHOW MODEL” button, a file browser dialog with the exported model appears. You can also open the model in the eIQ Model Tool by clicking the “OPEN MODEL” button.



Export File Type: Deepview RT
Input Data Type: Maintain Existing Data Type
Output Data Type: Maintain Existing Data Type
Export Quantized Model

Export Model

Project	fooddataset-2
Model Name	classification-performance-mcu-2022-08-17T06-35-05.171Z
Task	classification
Tune	mobilenet_v2
Labels	Buritto_plate, Chickensandwich_plate, Chips and Salsa_plate, Hotdog_plate, Meatballs_sandwich, Meatballs_spagetti, Pizza_plate, Potatoesandcheese
Epochs Trained	50
Training Time	56s
Training Accuracy	100.0%
Validation Accuracy	68.42%

Figure 3.37.: eIQ Portal: Deployment

DeepView VisionPack allows developers to deploy vision models created with eIQ Toolkit to NXP i.MX 8M devices. To learn more see the VisionPack QuickStart.

3.10 Quantization

You can quantize a trained model to reduce its size and speed up the inference time on different hardware accelerators (for example, GPU and NPU) with a minimal accuracy loss. You can choose between the per channel and per tensor quantizations. The per tensor quantization means that all the values within a tensor are scaled in the same way. The per channel quantization means that tensor values are separately scaled for each channel (for example, the convolution filter is scaled separately for each filter). The per channel quantization usually introduces a smaller error, while the per tensor quantization usually runs faster on the hardware accelerators.

To quantize a trained model, perform the following steps:

1. Go to the “Validation” window (see Figure 3.36.) by clicking the “VALIDATE” button in the “Trainer” menu (see Figure 3.20.).
2. Enable the post-training quantization by clicking the associated toggle button.
3. Choose the “Per Channel or Per Tensor” quantization and select the required “Input Data Type” and “Output Data Type”.
4. Apply the subsequent process by clicking the “VALIDATE” or “VALIDATION TARGET” buttons.

To export a quantized model, perform the following steps:

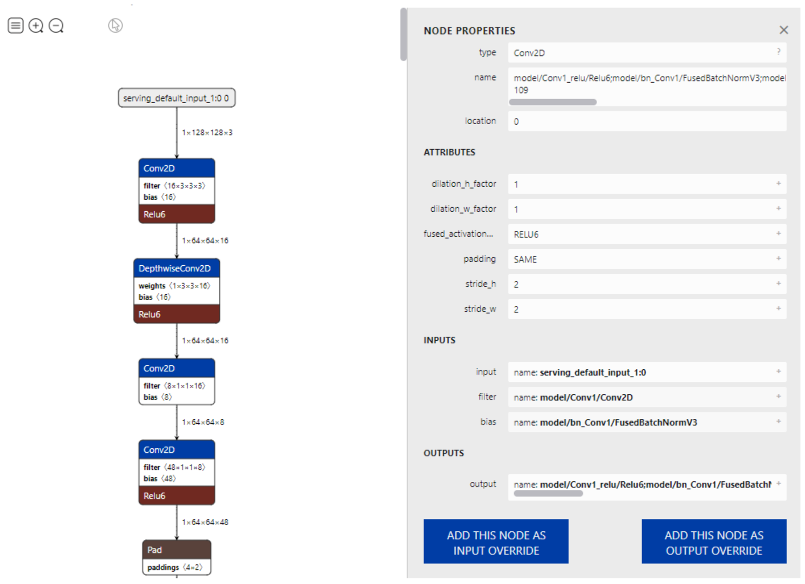
1. Go to the “Export Model” window (see Figure 3.37.) by clicking the “DEPLOY” button in the “Validation” menu (see Figure 3.36.).
2. Enable the post-training quantization by clicking the “Export Quantized Model” button.
3. Choose the “Per Channel” or “Per Tensor” quantization and select the required “Input Data Type” and “Output Data Type”.
4. Click the “EXPORT MODEL” button and select the required output name and folder.

4 Model Tool

Model Tool is the key component of the *Bring Your Own Model* workflow. It allows you to view, profile, convert or modify pre-trained models. Models can run on the local machine or on a remote device. You can either open *Model Tool* and then load your model or open the model after training using *eIQ Portal* as follows:

1. From the “Start” menu, click *Model Tool* and open a model.
2. Open *eIQ Portal* (Figure 3.1.), and click the “MODEL TOOL” button.
3. Click the “Open Model…” button and select the required model.

In *Model Tool*, individual layers are visualized when zooming in and out. By clicking on a layer, you see more information in the right-hand panel. See Figure 4.1.:



NODE PROPERTIES

type	Conv2D
name	model/Conv1_relu/Relu6;model/bn_Conv1/FusedBatchNormV3;model 109
location	0

ATTRIBUTES

dilation_h_factor	1
dilation_w_factor	1
fused_activation...	RELU6
padding	SAME
stride_h	2
stride_w	2

INPUTS

input	name: **serving_default_input_1:0**
filter	name: **model/Conv1/Conv2D**
bias	name: **model/bn_Conv1/FusedBatchNormV3**

OUTPUTS

output	name: **model/Conv1_relu/Relu6;model/bn_Conv1/FusedBatchN**

Figure 4.1.: eIQ Model Tool: convolution layer details

4.1 Model profiling

Model Tool provides means to display profiling statistics in order to optimize memory usage and performance of a model.

For profiling to work, Yocto BSP images have to be updated with *ModelRunner* including the latest features, found in <*eIQ_Toolkit_install_dir*>*deepviewrt*. The archive should be extracted to the image. There are multiple versions available for different Yocto BSP releases.

Note: There are other command-line based options to profile a model - REST API or `modelrunner-client` tool.

4.1.1 Enabling profiling on a device

If you want to obtain and display profiling information, you have to run *ModelRunner* (server part of profiling) on the target device. The following command is used to run *ModelRunner*:

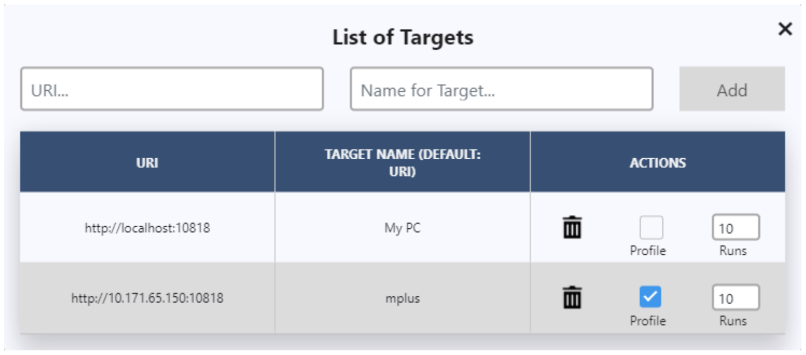
```
modelrunner -H {port} -d prof -e {engine}
```

Table 4.1 describes the options used to collect profiling data:

Table 4.1: Profiling parameters

Param	Description
-H (port)	Port number to identify modelrunner instance running on the target.
-d (mode)	Choose mode. Supported options are `prof` for profiling and `exec` (default) for regular modelrunner functions. `prof` must be selected.
-e (engine	Choose inference engine. You can choose among these options when the model is run on the target: `tflite-vx`, `rt-vx`, `tflite-cpu`, `rt-cpu`. You can choose `vela` option if the model is run on PC.

In order to inform *Model Tool* which server it should connect to, go to **Manage Targets** from the menu and select **Edit Targets**. A new screen will pop up, showing all available targets (see Figure 4.2.):



URI	TARGET NAME (DEFAULT: URI)	ACTIONS		
http://localhost:10818	My PC	🗑	☐ Profile	10 Runs
http://10.171.65.150:10818	mplus	🗑	✓ Profile	10 Runs

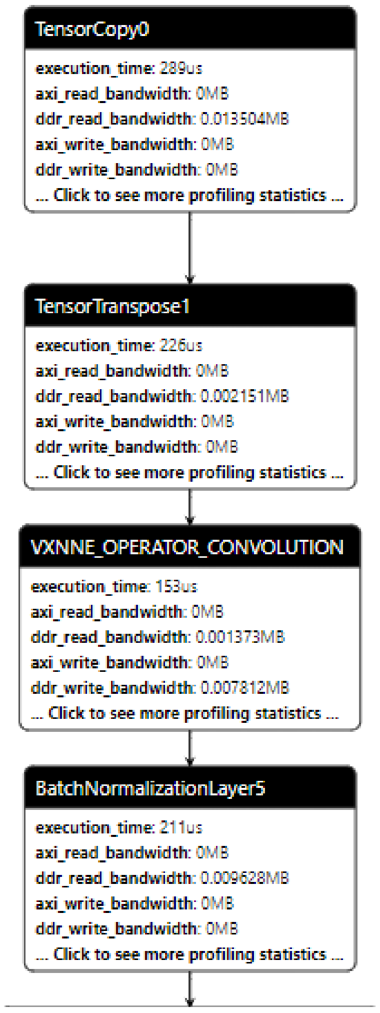
Figure 4.2.: Manage remote server URLs

Make sure to add the device where *ModelRunner* is running, then simply check the **Profile** option.

Once the server is running on the target and a proper target is set in *Model Tool*, go to **Profiling** in the menu and select **Profile Model**. This will execute the model and collect profiling data.

4.1.2 Runtime and original graphs

When *Model Tool* receives all the necessary data, it will first open a new window. This window will contain the runtime graph which was executed on the device (see Figure 4.3.). Such graph will look differently from the original model because it was processed by the runtime, and it will look different for every backend. For example profiling with VX Delegate on i.MX8MPlus will return different statistics than profiling with Vela estimation tool. Backends (e.g. TF Lite delegates) typically modify/optimize graphs offline before executing them to provide better performance. Such graph might be harder to interpret by the user. On the other hand it contains all the collected information.

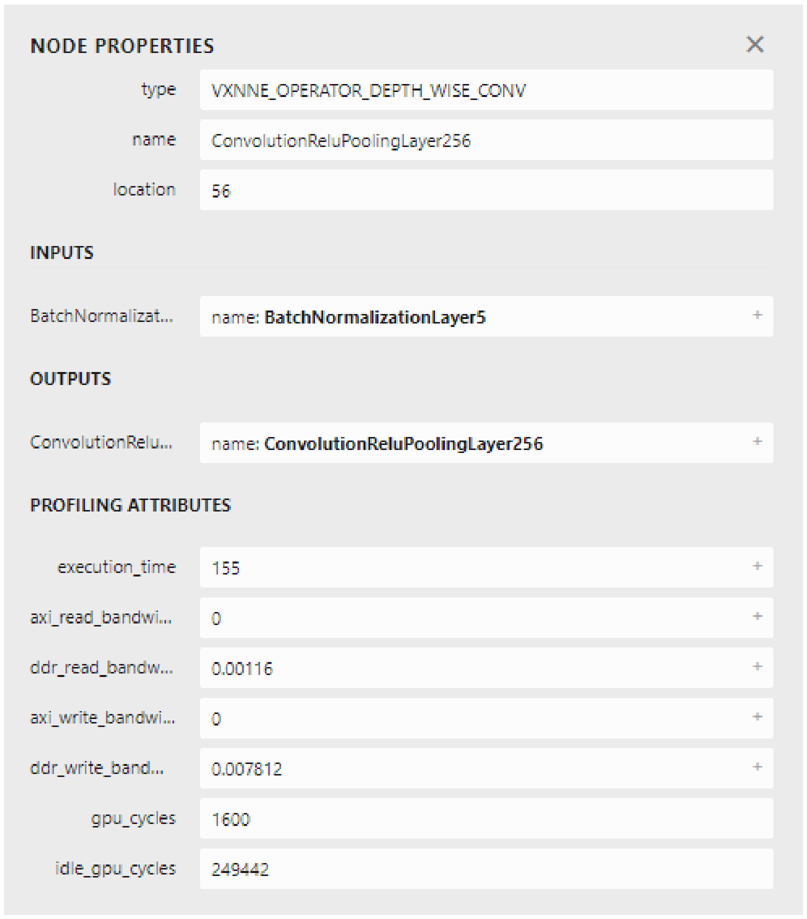


```
TensorCopy0
execution_time: 289us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.013504MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0MB
... Click to see more profiling statistics ...

TensorTranspose1
execution_time: 226us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.002151MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0MB
... Click to see more profiling statistics ...

VXNNE_OPERATOR_CONVOLUTION
execution_time: 153us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.001373MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0.007812MB
... Click to see more profiling statistics ...

BatchNormalizationLayer5
execution_time: 211us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.009628MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0MB
... Click to see more profiling statistics ...
```

Figure 4.3.: Runtime graph for i.MX8MPlus

A few items are directly visible in the nodes, but when you click on a node, it will show the complete list of profiling statistics per layer (see Figure 4.4.).

Model Tool will also map the execution to the original graph and display it (see Figure 4.5.).



NODE PROPERTIES

type	VXNNE_OPERATOR_DEPTH_WISE_CONV
name	ConvolutionReluPoolingLayer256
location	56

INPUTS

BatchNormalizat...	name: **BatchNormalizationLayer5**

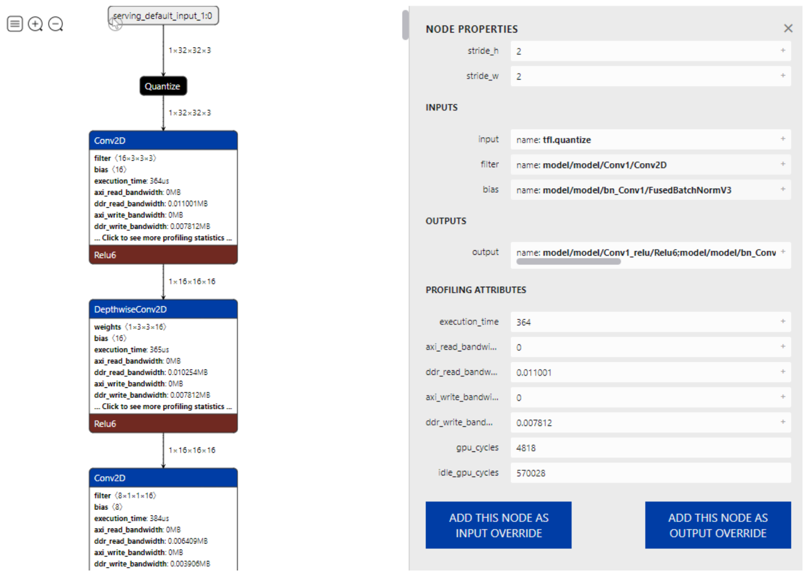
OUTPUTS

ConvolutionRelu...	name: **ConvolutionReluPoolingLayer256**

PROFILING ATTRIBUTES

execution_time	155
axi_read_bandwi...	0
ddr_read_bandw...	0.00116
axi_write_bandwi...	0
ddr_write_band...	0.007812
gpu_cycles	1600
idle_gpu_cycles	249442

Figure 4.4.: Per layer profiling statistics



Conv2D
filter ⟨16×3×3×3⟩
bias ⟨16⟩
execution_time: 364us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.011001MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0.007812MB
... Click to see more profiling statistics ...
Relu6

DepthwiseConv2D
weights ⟨1×3×3×16⟩
bias ⟨16⟩
execution_time: 365us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.010254MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0.007812MB
... Click to see more profiling statistics ...
Relu6

Conv2D
filter ⟨8×1×1×16⟩
bias ⟨8⟩
execution_time: 384us
axi_read_bandwidth: 0MB
ddr_read_bandwidth: 0.006409MB
axi_write_bandwidth: 0MB
ddr_write_bandwidth: 0.003906MB

NODE PROPERTIES

stride_h	2
stride_w	2

INPUTS

input	name: **tfl.quantize**
filter	name: **model/model/Conv1/Conv2D**
bias	name: **model/model/bn_Conv1/FusedBatchNormV3**

OUTPUTS

output	name: **model/model/Conv1_relu/Relu6;model/model/bn_Conv**

PROFILING ATTRIBUTES

execution_time	364
axi_read_bandwi...	0
ddr_read_bandw...	0.011001
axi_write_bandwi...	0
ddr_write_band...	0.007812
gpu_cycles	4818
idle_gpu_cycles	570028

ADD THIS NODE AS INPUT OVERRIDE
ADD THIS NODE AS OUTPUT OVERRIDE

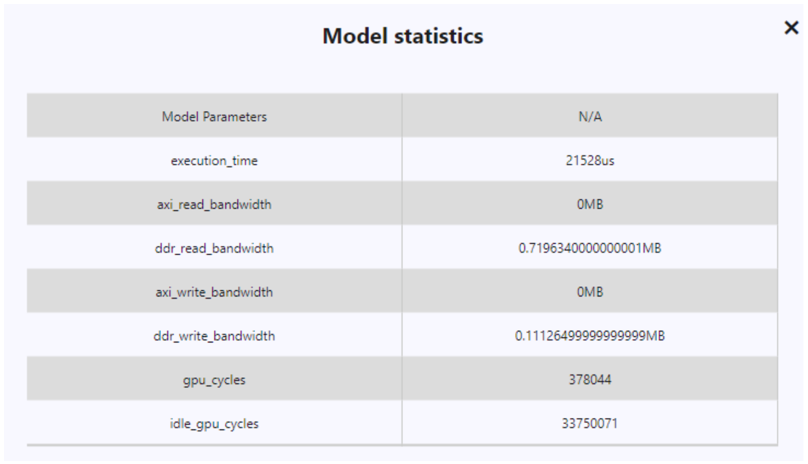
Figure 4.5.: Per layer profiling statistics for the original model

4.1.3 Charts and statistics

For the optimized runtime model, there is an option to show profiling charts that summarize per layer statistics. You can go to **Profiling** in the main menu and select **Show Statistics**. A window with various charts will appear.

On the top of this window, you can see names of the collected statistics. You can switch between them to see charts for each individual statistic.

You can also display a summary for the whole model by going to **Profiling** > **Total Model Statistics** (see Figure 4.6.).



Model statistics

Model Parameters	N/A
execution_time	21528us
axi_read_bandwidth	0MB
ddr_read_bandwidth	0.7196340000000001MB
axi_write_bandwidth	0MB
ddr_write_bandwidth	0.11126499999999999MB
gpu_cycles	378044
idle_gpu_cycles	33750071

Figure 4.6.: Total model profiling statistics

4.1.3.1 Chart types

Depending on the type of profiling (offline vela, tflite-vx, rt-vx, cpu…) a different charts will show in this window. These are the common charts you can see and their description:

1. **Timeline chart**: This chart shows the order in which nodes were executed as well as the time it took for each node to execute. The x-axis represents time, the y-axis types of nodes (convolution, pooling, etc.).

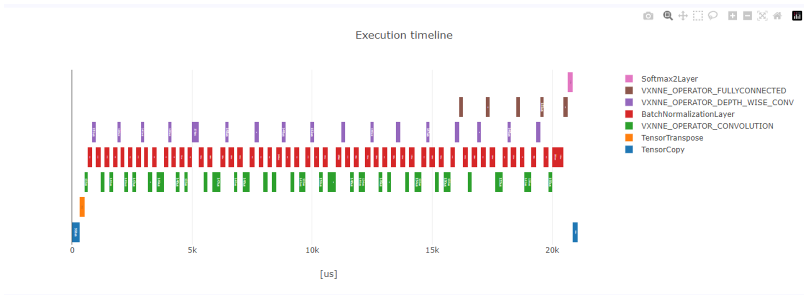


Figure 4.7.: Execution time chart.

2. **Charts grouped by property**: These are typically the pie charts. Values of the observed profiling statistic were summed up for the nodes with the same property. The charts show for example memory consumption for each layer type.
3. **Individual node statistics**: These are the bar charts. They display statistics for each node, for example memory usage, gpu/npu cycles needed to execute one node etc. Some of the statistics can be composed of two and more components. For example memory usage can contain read memory and write memory. If something like this

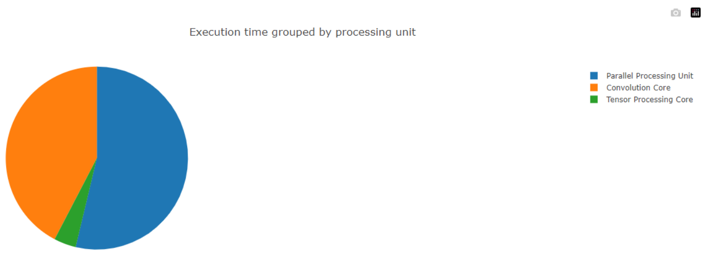


Figure 4.8.: Example of chart grouped by a node property.

occurs, the read memory and write memory are differenciated by the different colors in the column. The example image below shows how this works on an example of GPU cycles. They are composed of two parts - idle cycles (when execution unit was waiting) and cycles (when execution unit was computing).

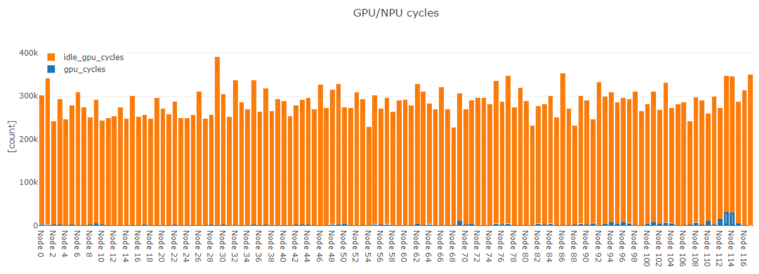


Figure 4.9.: Example of chart displaying statistics for each node.

4.1.4 Launching profiling from command line

If there is a need to automatically process collected profiling data instead of displaying them in Model Tool, `modelrunner-client` provides a command line interface that exports collected data into selected folder. To run profiling this way, run command with these parameters: `modelrunner-client -m [path_to_model] -u [url] -o [output folder]`

- Parameter `-m` specifies path to a model that will be profiled.
- Parameter `-u` gives URL of a device where modelrunner is running, it should have format `http://[IP]:[Port]`
- Parameter `-o` specifies output folder, where files with collected data will be saved.

This approach generates three files:

- raw_profiling_log - contains data as it was collected from profiler
- optimized_model - is a JSON file that contains optimized graph reconstructed from the raw profiling log together with collected profiling data
- mapping - is a JSON file that contains mapping of optimized layers into original model layers

4.2 Model conversion

An essential *Bring Your Own Model* feature of *Model Tool* is model conversion/export. The following types of model formats are currently supported:

- **RTM** – Real Tracker Module file format
- **TF Lite** – TensorFlow Lite Flatbuffers file format
- **ONNX** – Open Neural Network Exchange file format
- **TF Lite Vela** - special case of the TF Lite format, where a model has been converted into one or more Ethos-U nodes containing representation for hardware acceleration.

To start the conversion, click the “File/Convert” dialog box in the menu. The following screen appears:

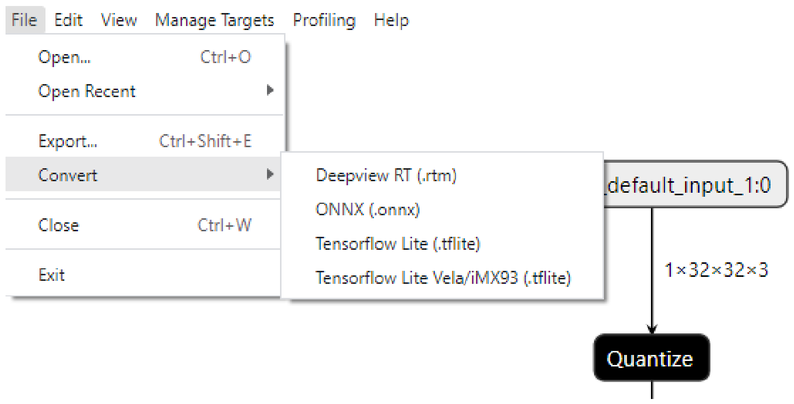
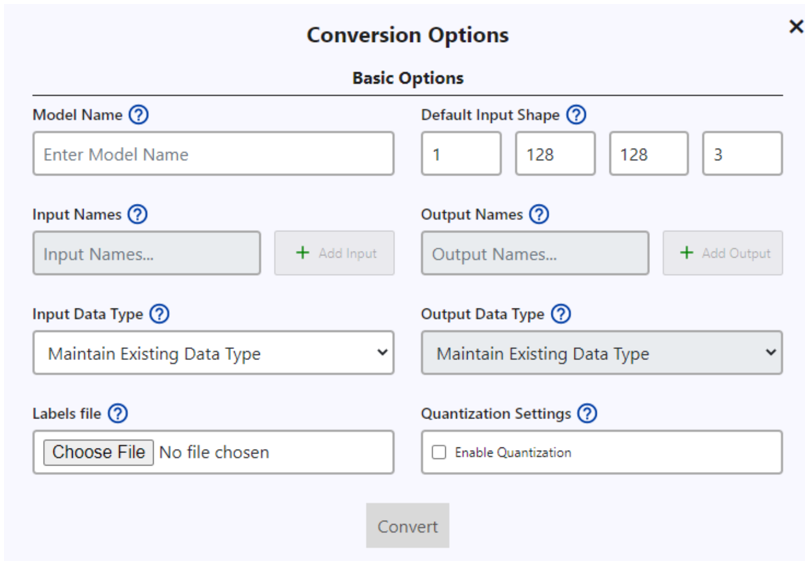


Figure 4.10.: eIQ Model Tool: model conversion selection

After selecting the desired model format, a screen similar to the one below appears:



Conversion Options

Basic Options

Model Name: Enter Model Name	Default Input Shape: 1, 128, 128, 3
Input Names: Input Names... (Add Input)	Output Names: Output Names... (Add Output)
Input Data Type: Maintain Existing Data Type	Output Data Type: Maintain Existing Data Type
Labels file: Choose File No file chosen	Quantization Settings: ☐ Enable Quantization

Convert

Figure 4.11.: eIQ Model Tool: model conversion options

The tables below describe all available parameters. Each output format might have specific options. Quantization is also important and additional parameters appear after enabling it. Note however, that the conversion to RTM does not support quantization. The model should already be quantized if needed.

Table 4.2: Conversion options common for all formats

Model Name	The name of the output model.
Default Input Shape	The input shape of the model. It is pre-populated if possible.
Input/Output Names	Specify the input and output nodes of the file. The nodes' names can be also renamed.
Labels file	Text file providing labels for each class.
Quantization Settings	Enables Quantization. Additional options open when checked (see Table 4.3).

Table 4.3: Quantization Settings

Conversion Quantizatior Type	Select either per-channel or per-tensor quantization. Per-channel quantization means that every channel has its own zero point and scaling factor. Per-tensor quantization means that all the channels have the same zero point and scaling factor. Per-tensor quantization generally provides the same or better performance on accelerators. Per-channel is considered better practice, and provides better accuracy.
Input/Outpu Data Type	The data type for the input/output nodes. ONNX conversions must be of the float32 type.
Quantize Normalizati	The quantization normalization (signed, unsigned). I.e. range is either from -128 to 127 or from 0 to 255.
Select Samples Folder/Num of Samples	Select a folder of sample images and their number to be used during quantization, preferrably data used for training. Provided data determine the range distribution and greatly impact accuracy.

Table 4.4: DeepViewRT Settings

Input/Output Data Type	The data type for input/output nodes.
Anchor Box Settings	Provide anchor boxes required for decoding SSD object detection models.

Table 4.5: TF Lite Vela Settings

Vela Configuration file	Select configuration file. Default configuration is provided to match i.MX93 and is located in *resources* folder of eIQ Toolkit installation.

The conversion process is started by clicking the "Convert" button. If the conversion process is performed correctly, the "Conversion Download" output box appears and the new model is saved into the working directory. Output formats are not compatible between each other, so the " Conversion Error" message box with the problem description might also appear. In this case, you can change the conversion parameters and try the conversion again.

5 Command-line Tools

The “eIQ Command-line Tools” allow you to run some of the previously described tasks in a command prompt, instead of running these tasks in the Graphical User Interface (GUI) of the “eIQ Toolkit”. Moreover, it provides additional features that cannot be accessed through the GUI. There are two ways to open the command-line on Windows, which make sure that eIQ Toolkit is enabled:

1. From the “Start” menu, click the “eIQ Command-line Tools”.
2. Open the eIQ Portal (see Figure 3.1.) and click “COMMAND LINE”.

Note: Use “-h” or “–help” with the command-line tool to explore the full range of options available.

5.1 Converter

The “Converter” command-line tool can convert machine-learning models into different output formats.

The supported input formats are the following:

- TensorFlow.pb/SavedModel files
- TensorFlowLite.tflite files
- Keras .h5/.h5df files
- TensorFlow hub saved_models/links
- Neural Network Exchange Format (NNEF) graphs

The supported output formats are the following:

- DeepView RTM files
- TensorFlowLite .tflite files
- ONNX files

Table 5.1 can be used to navigate through various model file conversions. The “q” notation means that the quantized file and the destination quantized files use the “–quantize” arguments during the conversion.

Table 5.1: Converter support

Source\destination	RTM	TFLite (MLIR/TOCO)	ONNX	RTM q	TFLite (MLIR/TOCO) q	ONNX q
1.x Pb	Y	Y	Y	Y	Y	Y
Saved model (folder/TAR)	Y	Y	Y	Y	Y	Y
Keras (h5)	Y	Y	Y	Y	Y	Y
RTM	N/A	N	N	N/A	N	N
TFLite (MLIR/TOCO)	Y	N/A	Y	N	N	Y
ONNX	Y	Y	N/A	N	Y	Y
TFLite q (MLIR/TOCO)	Y	N/A	Y	N	N/A	N/A
ONNX q	Y	Y	N/A	N	N/A	N/A

The limitations for the conversion are related to third-party function calls and software library/runtime versions. It is mainly converting from ONNX to TFLite or vice versa due to unsupported operators, op-set requirements, or node dimension mismatch. Error messages appear when such conversions are attempted and not supported.

Converting the source quantized TFlite to the quantized ONNX model requires ONNX Runtime 1.6 (or higher), because it uses dynamic quantization where the model weights are quantized 8-bit internally, but they are converted to float during runtime. If the source model is already quantized, do not pass the "–quantize" arguments in the command tool and do not enable the quantization in a conversion form in the model tool.

The conversion from the source float TFLite model to the quantized ONNX model uses static quantization, where the "–quant-tensor" argument must be passed to run on ONNX runtime 1.5.3. The per-channel quantization is supported in the command-line tool, but the model does not work on ONNX runtime 1.5.3. Therefore, the per-tensor static quantization must be used in the conversion for now. This may change with updated ONNX runtimes in the future BSPs.

5.1.1 Example

The tool has two mandatory parameters: "input_file" and the model (output file). To convert the mobilenet model from Keras, perform the following steps:

1. Open the eIQ Portal (see Figure 3.1.) and click "COMMAND LINE".
2. Download and save the mobilenet model from "keras.applications":

```
python
>> import tensorflow as tf
>> tf.keras.applications.MobileNet().save('mobilenet.h5')
```

3. Exit the Python prompt and execute the following command to convert the *mobilenet.h5* model to the *mobilenet.rtm* model and embed the labels (provided by default in the workspace):

```
deepview-converter --labels workspace\models\mobilenet_v1_1.0_224\labels.txt
mobilenet.h5 mobilenet.rtm
```

4. Execute the following command to also convert the Keras *mobilenet.h5* model to the *mobilenet.tflite* model:

```
deepview-converter mobilenet.h5 mobilenet.tflite
```

Note: You can open the created *mobilenet.rtm* and *mobilenet.tflite* models in the "eIQ Model Tool" to see that the input name is "input_1" and the output name is "Identity".

5.1.1.1 Conversion using specific plugin

The converter is selecting the right plugin according to the destination model type. However, there might be cases, when there are multiple plugins with the same output types. In this case, to avoid any misunderstandings, there is one more parameter required for the command. Parameter `--plugin` specifies name of the plugin that is chosen for conversion. The conversion command can then look like this:

```
deepview-converter mobilenet.h5 mobilenet.tflite --plugin deepview-converter-tflite
```

To see a list of all available plugins, run command

```
deepview-converter -l
```

5.1.1.2 Conversion with Arm-Vela converter

Arm-Vela conversion differs from other convertors in destination selection. While other converters require destination file to be specified, Arm-Vela converter is generating multiple files, thus it needs to be provided with output folder. This changes the required parameters for command line conversion. Parameter output-model-type has to be set to tflite. Conversion command will in this case look like this:

```
deepview-converter --plugin deepview-converter-armvela --output-model-type tflite␣
↪mobilenet.tflite ./vela_output_folder
```

5.2 Importer

Besides using a graphical interface to load datasets into the eIQ Portal (as described in *Plugins*), there is also a command-line option. You can use it to load VOC-type datasets.

The "DeepView Importer" can be executed from the command line as follows:

```
deepview-importer [-list] [-difficult] [-truncated] [-group GROUP] [-project DEEPVIEW_
↪PROJECT]
VOC_DATASET.TAR
```

The VOC_DATASET.TAR is the TAR file for the VOC train or test datasets. It should be used in combination with the "-group train" or "-group test" arguments. Otherwise, the shuffle feature can be used.

The "-list" option lists a summary of discovered labels and associated annotations.

The "-difficult" option causes the importer to include annotations flagged as difficult.

The "-truncated" option causes the importer to include annotations flagged as truncated.

Finally, the "-project DEEPVIEW_PROJECT" option triggers the actual import operation to happen against this target dataset. It can be an existing dataset or a new one. You should back up the existing datasets in cases of errors, because it has not been heavily tested.

An example of importing the Pascal Visual Object Classes Challenge dataset can look as follows:

```
deepview-importer -group train -project VOC.eiqp VOCtrainval_14-Jul-2008.tar
deepview-importer -group test -project VOC.eiqp VOCtest_06-Nov-2007.tar
```

5.3 Trainer

The “Trainer” command line tool is used to run model training directly, without clicking through several pages in GUI.

To run training you have to specify at least these parameters:

- **–dataset** path to dataset the model will be trained on
- **task** can be specified using one of these three parameters:
- **–classification** to train model for classification task
- **–detection** to train model for object detection task
- **–segmentation** to train model for image segmentation task
- **–tune** parameter will choose specific model for one of the tasks from base models. Look into *plugins* folder to see available models for each task
- **–input_shape** specifies how should the input data be reshaped

To see how to use other trainer parameters, the tool provides extensive documentation in help:

```
deepview-trainer -h
```

5.3.1 Examples

Here are three examples (one for each task type) on how to run training from command line:

```
deepview-trainer --dataset segmentation.eiqp --segmentation --tune unet --input_shape 1,
↪32,32,3
```

```
deepview-trainer --dataset mnist.eiqp --classification --tune mobilenet_v1 --input_shape␣
↪1,32,32,3
```

```
deepview-trainer --dataset coco.eiqp --detection --tune ssd_mobilenet_v3 --input_shape␣
↪320,320,3
```

5.4 Validator

The “Validator” command-line tool uploads models to the “*ModelRunner*”. It can validate models against the reference numpy files (*.*npz*). This way you can quickly validate the model, determine if it is deployable on the device, and check its latency. The “Validator” is also able to validate a model against image or several images. Running this command with a classification model produces top “n” predictions, printing out class and confidence score (as shown in the “Image validation example” below). Running this command with an object detection model prints out predicted class, confidence score, nd bounding box for the detected object. If it is run without the –top [n] argument, it will print only one detection for an image. If it is run with the –top [n] argument, it prints “n” detections for one image. If the number “n” is greater than the number of the actual predictions, it prints all detections for one image.

The “Validator” can validate a model against the whole dataset. For more examples on how to use the “Validator” with images and datasets, see the Jupyter notebooks in *<eIQ_Toolkit_install_dir>\workspace\validator*.

Note: The “Validator” runs as a host (PC) while the “ModelRunner” is a server (remote target) in this case.

5.4.1 Image validation example

In the following example, we evaluate a set of images with the popular MobileNet v1 model and validate the model itself.

1. Open the eIQ Portal (see Figure 3.1.) and click “COMMAND LINE”.
2. Navigate to the *<eIQ_Toolkit_install_dir>\workspace\models\mobilenet_v1_1.0_224* folder, where the label files for the MobileNet v1 model and several images are provided.
3. Convert the model to the RTM format:

   ```
   deepview-converter --labels labels.txt
   mobilenet_v1_1.0_224_quant.tflite
   mobilenet_v1_1.0_224_quant.rtm
   ```

4. Run the following on the simulator (host PC):
   - Open a second instance of the “eIQ Command-line Tools”.
   - Start the Modelrunner:

     ```
     modelrunner -H 10818
     ```

   - Run the model:

     ```
     deepview-validator --image imgs --top 5 mobilenet_v1_1.0_224_quant.rtm
     ```

5. Run the following remotely on the target board:
   - On the target board, run Modelrunner (it runs on the CPU):

     ```
     modelrunner -H 10818
     ```

   - On the host PC, from the “eIQ Command-line tools” terminal, run the model remotely on the target:

     ```
     deepview-validator --image imgs --top 5 mobilenet_v1_1.0_ 224_quant.rtm --uri␣
     ↪http://<remote_address>:10818/v1
     ```

The final command should produce the evaluated softmax values using an argmax function similar to the following:

```
...
Image imgs\dog.jpg:
1:| Labrador retriever | 0.9766
2:| bath towel | 0.0039
3:| Rhodesian ridgeback | 0.0039
4:| beagle | 0.0039
5:| golden retriever | 0.0039
...
Average Model runtime: 71.5493ms
```

Note: The *.rtm* file format is used internally by the "DeepViewRT". The labels are provided to be embedded into the model.

5.4.2 Model validation example

Another option to validate a model is to generate numpy array values, run the model, and check these values.

1. Open the eIQ Portal (see Figure 3.1.) and click "COMMAND LINE".
2. Navigate to the *<eIQ_Toolkit_install_dir>\workspace\models\mobilenet_v1_1.0_224* folder, where the MobileNet v1 model is provided.
3. Prepare the reference data. The following command creates a compressed *.npz* file containing two arrays: a randomly generated input array and the output array:

```
deepview-validator --input_names input --output_names MobilenetV1/Predictions/
↪Reshape_1 --input_shapes 1,224,224,3 mobilenet_v1_1.0_224_frozen.pb
```

4. Convert the TensorFlow model to the RTM format by entering:

```
deepview-converter --labels labels.txt mobilenet_v1_1.0_224_frozen.pb mobilenet_v1_
↪1.0_224.rtm
```

5. Validate the RTM model against the data produced by the original TensorFlow model on the remote target:
   - On the target board, run Modelrunner (it runs on the CPU):

```
modelrunner -H 10818
```

   - On the host PC, from the "eIQ Command-line tools" terminal, run the validation remotely on the target:

```
deepview-validator --input_names input output_MobilenetV1/Predictions/Reshape_1 ref_
↪outputs MobilenetV1/Predictions/Reshape_1 --reference mobilenet_v1_1.0_224_frozen.
↪npz mobilenet_v1_1.0_224.rtm --uri http://<remote_address>:10818/v1
```

6. Validate the RTM model against the data produced by the original TensorFlow model on the simulator (host PC):
   - Open a second instance of the "eIQ Command-line Tools".
   - Start the ModelRunner:

   ```
   modelrunner -H 10818
   ```

   - Run the validation:

   ```
   deepview-validator --input_names input output_MobilenetV1/Predictions/Reshape_1␣
   ↪ref_outputs MobilenetV1/Predictions/Reshape_1 --reference mobilenet_v1_1.0_
   ↪224_frozen.npz mobilenet_v1_1.0_224.rtm
   ```

The final command should produce the validation information similar to the following:

```
...
Validating Sample: 1/1
Output Name: MobilenetV1/Predictions/Reshape_1 Top-1 Matches: 1/1 - 100.00 %
Top Average Delta: 0.0000
```

(continues on next page)

(continued from previous page)

```
Top Max Delta 0.0000
Model run took: 0.0000ms
All outputs validate within error.
```

5.5 ModelRunner/Client

The “ModelRunner/Client” is a client/server interface for loading, running, and evaluating models on the target. The “ModelRunner” acts as a server and it typically runs on the target where it handles requests from the “ModelClient” and provides responses. The “ModelClient” provides an HTTP API. You can either use a utility like “cURL” to call HTTP requests directly, or a Python API.

A local “ModelRunner” is also available in the *bin* directory which can be called by entering “localhost” as a target and it acts as a simulator.

Note: This document provides details on how to use the “eIQ Toolkit”, which focuses on the host, so only simple examples on how to use the “ModelRunner”, which binds the host and the target, are described below. The “ModelRunner” itself runs on the target, so it is not a part of the “eIQ Toolkit”. For more details about target deployment, such as a detailed description of the API, see the *DeepViewRT User Manual*. The document is in the *<eIQ_Toolkit_install_dir>/docs* folder.

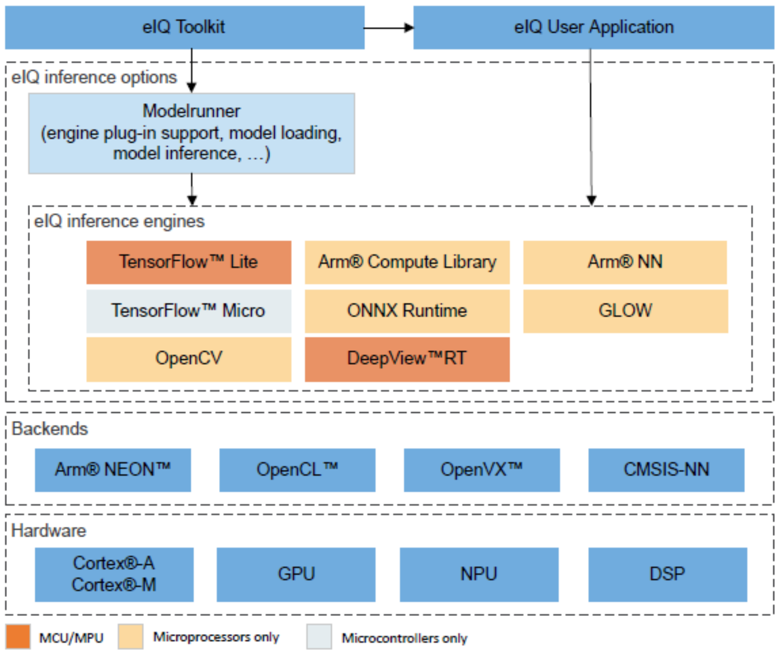


Figure 5.1.: eIQ Inference Engine Options

The “ModelRunner” handles the communication and acts as an intermediate inference engine, which delegates the compute to other inference engines. To perform the compute, even lower-level technologies are utilized, depending on the hardware available.

5.5.1 cURL examples

The following examples demonstrate the REST API using the "cURL" command-line tool. This first example sends an image (PNG or JPEG) to the "ModelRunner" and expects the response to contain the inferred label, as well as additional timing information in the headers.

Note: The "data-binary" parameter requires a leading "@" to force it to read the file with the "FILENAME" instead of sending the "FILENAME" string as data.

```
curl -D - -XPOST -H 'Accept: text/plain' -H 'Content-Type: image/*' --data-binary
↪'@FILENAME' 'http://localhost:10818/v1?run=1'
```

Removing the "-H Accept: text/plain" results in the default application/json response instead.

```
curl -D - -XPOST -H 'Content-Type: image/*' --data-binary '@FILENAME' 'http://localhost:
↪10818/v1?run=1'
```

If you would like to receive the output tensor contents as opposed to the label, you can use the output parameter. The output should be set to the name of the layer that you want to read (typically named output).

```
curl -D - -XPOST -H 'Content-Type: image/*' --data-binary '@FILENAME' 'http://localhost:
↪10818/v1?run=1&output=output'
```

5.5.2 ModelClient API

The "ModelClient API" is a way for users to communicate with the "ModelRunner" service using Python. The "ModelClient" class provides methods for loading a model, running a model, and gathering the timing and layer information of the current model. The "ModelClient" can be initialized with just a URI, or a URI and a model in the form of a byte array or filepath to where the *.rtm* file is located. A new URI can be set through a simple assignment and if no "http://" is found, it is added to the URI.

```python
from deepview.rt.modelclient import ModelClient

modelclient = ModelClient("http://localhost:10818/v1")
modelclient = ModelClient("http://localhost:10819/v1", "mobilenet.rtm")
modelclient.uri = "localhost:10818/v1"
```

Note: See *Example* for how to obtain the *mobilenet.rtm* model used in the above example.

When a ModelClient has been created, a new model can be loaded, overwriting the previous one and allowing for the new model to run.

```python
from deepview.rt.modelclient import ModelClient

modelclient = ModelClient("http://localhost:10818/v1")
modelclient.load_model("mobilenet.rtm")
```

When a model has been loaded, it can be run with the "ModelClient" method that accepts a dictionary of inputs and a list of outputs that you want to be returned with their associated numpy tensors. There is an additional timeout parameter to avoid the possibility of the script hanging if there is poor connectivity to the "ModelRunner".

The return value from the run is a dictionary that associates the given output names with their calculated tensor. When using a memory map, some return values may be overwritten, if they use memory that a later layer is mapped to use as well.

```
import numpy as np

input_val = np.random.rand(1, 224, 224, 3).astype(np.float32) * 255
inputs = {'input_1': input_val}
outputs = ['Identity']
results = modelclient.run(inputs, outputs)
```

When a model has been run, it is possible to gain access to the timing and layer information of the given model. It provides information on how long it took for the model to be uploaded to the "ModelRunner" service, the time for the "ModelRunner" to receive the inputs and return the outputs to the user, as well as the evaluation time of the model. It can be gathered from the "get_timing_info" method. The "ModelClient" also provides access to the timing of each type of layer within the graph through the "get_op_timing_info" method.

The return value from "get_op_timing_info" is a dictionary of operation names, where each value is a list of the average time for that operation type, the total time taken by that operation, and the number of times that operation was encountered within the model.

```
put_time, post_time, eval_time = modelclient.get_timing_info()
op_timing = modelclient.get_op_timing_info()
```

The "ModelClient" also provides access to the raw layer information through the "get_layers" method. It returns a dictionary of all the layer names with their associated inputs, type, tensor shape, datatype, and timing info in nanoseconds.

```
model_layers = modelclient.get_layers()
```

5.5.2.1 Example

1. Open the eIQ Portal (see Figure 3.1.) and click "COMMAND LINE".
2. Run the "ModelRunner" by executing the "modelrunner -H 10819" command.
3. Open a second "eIQ Command-line Tools" prompt.
4. Prepare the Mobilenet RTM model.
5. Open Python in the command line by typing "python".
6. Run the following script to get the output of the Mobilenet:

```
from deepview.rt.modelclient import ModelClient
import numpy as np

modelclient = ModelClient("http://localhost:10819/v1", "mobilenet.rtm")
input_val = np.random.rand(1, 224, 224, 3).astype(np.float32) * 255
results = modelclient.run({'input_1': input_val}, ['Identity'])
```

5.5.3 DeepViewRT/ModelRunner updates

To update the latest DeepViewRT and ModelRunner client applications on the target, we provide standalone **.deb* Yocto Linux packages. The currently supported Yocto Linux releases are the following:

- 5.4.70_2.3.2 (GCC9)
- 5.10.35_2.0.0 (GCC10)
- 5.10.52_2.1.0 (GCC10)

You may download Yocto Linux BSP packages from [https://www.nxp.com/design/ software/embedded-software/i-mx-software/embedded-linux-for-i-mx-applicationsprocessors:IMXLINUX](https:// www.nxp.com/design/ software/embedded-software/i-mx-software/embedded-linux-for-i-mx-applicationsprocessors:IMXLINUX) .

Follow the example steps below to install the packages. The version depends on the target image:

- Copy the selected **.deb* file from the *<eIQ_Toolkit_install_dir>/deepviewrt* folder to the target board. Make sure to select the correct version, based on the Yocto Linux version you have (see above).
- Run the `dpkg -r --force-all deepview-rt` command to uninstall the previously installed deepview-rt package.
- Run the `dpkg -r --force-all deepview-rt-dev` command to uninstall the previously installed *deepview-rt-*.dev* package. This step may not be needed for version 5.4.70_2.3.2.
- Run the `dpkg -i deepview-rt_2.4.25-aarch64-r0_arm64_xxxx.deb` command to install the new package.

Note: Both the deepview-rt and deepview-rt-dev packages must be uninstalled correctly before installing the new /.deb* package.

5.5.4 Remote validation using ModelRunner

ModeRunner can be also used for validating the models remotely on a target device. The user must make sure that ModelRunner can run the model on that device. The following command can be used:

```
modelrunner -H <remote_port> [-e <engine>] [-c <compute>]
```

Note: Not all combinations of models, and engines are supported on i.MX8 device. For example, you cannot run an ONNX model using TF Lite ModelRunner, because the engine simply does not support such format. Run `modelrunner --help` to see the full list of parameters and options.

5.6 Python environment

The "eIQ Command-line Tools" provide a standalone Python environment with pre-installed modules for machine learning including the "DeepView" Python modules. You are not limited only to the pre-installed modules, but you can also extend the environment with your own modules using "pip":

1. Open the eIQ Portal (see Figure 3.1.) and click "COMMAND LINE".
2. Install the module as follows:

```
python -m pip <python_module>
```

5.6.1 Virtual environments

Working inside a virtual environment is considered a best-practice. This way, when you install a Python module, it only gets created inside a virtual environment which can be easily deleted, and you do not pollute the default one.

1. Open the eIQ Portal (see Figure 3.1.) and click "COMMAND LINE".
2. Create a virtual environment as follows:

```
python -m venv <venv_name>
```

3. The <*venv_name*> directory containing the virtual environment is created. Activate the virtual environment as follows:

```
<virtual_env_name>\Scripts\activate
```

or on Linux:

```
source <venv_name>/bin/activate
```

The <*venv_name*> directory now appears at the start of the command line.

5.6.2 Jupyter Notebooks

Some examples located in `<eiqtlk_install_dir>/workspace` are Jupyter notebooks which makes for a convenient way to work interactively with Python. To run a Jupyter notebook do the following:

1. Open the eIQ Portal (see Figure 3.1.) and click "COMMAND LINE".
2. Run the "Jupyter Notebook":

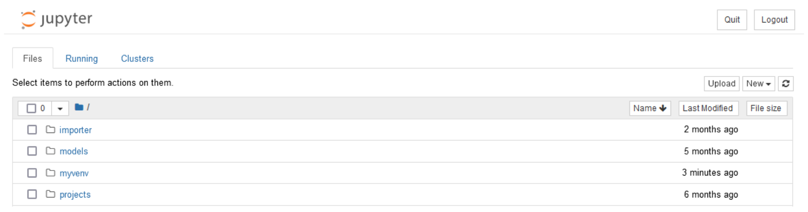
```
python -m jupyter notebook
```

3. Navigate to `<eiqtlk_install_dir>/workspace` and execute the Jupyter notebook.

Jupyter notebooks might also be used together with virtual environments between which you may switch. In order to see a virtual environment as a kernel, you first need to register it:

```
python -m ipykernel install --name=<venv_name>
```

Virtual environments can then be selected using "Kernel > Change kernel > <virtual_env_name>".



Select items to perform actions on them.

Name	Last Modified
importer	2 months ago
models	5 months ago
myvenv	3 minutes ago
projects	6 months ago

Figure 5.2.: “Jupyter Notebook” environment

Note: When executing Jupyter from within a virtual environment, the virtual interpreter is correctly selected, but it is viewed upon as the base ipykernel which may be a little confusing.

5.7 Neutron Converter

5.7.1 Main usage

The neutron-converter has the following options.

```
neutron-converter --input <input_model_path> --output <output_model_path> --target
↪<neutron_target>
```

Where:

- *input* — The path of the standard TensorFlow Lite model (mandatory).
- *output* — The path of the output converted TensorFlow Lite model (optional). If the path is not provided, the output model is written in the same directory as the input model. The output model name is suffixed with ***_converted***.
- *target* — The name of Neutron target for which the model is converted (optional). Default is `mcxn94x`.

To check the available targets provided by the tool, use the `show-targets` option.

```
neutron-converter --show-targets
```

5.7.2 Custom usage

By default, the neutron-converter tries to specialize all the Neutron supported operators. However, to specialize only some operators or exclude only some operators from being specialized use any of the following options.

- exclude-operator-types
  - Exclude operator types from Neutron execution (list of comma-separated names or indexes).
  - For example, use `--exclude-operator-types 0,1` or `--exclude-operator-types ADD,AVERAGE_POOL_2D1` to exclude from being specialized Add and AvgPool operator types.
- include-only-operator-types
  - Only include operator types to Neutron execution (list of comma-separated names or indexes).
  - For example, use `--include-only-operator-types 0,1` or `--include-only-operator-types ADD,AVERAGE_POOL_2D` to specialize only the Add and AvgPool operator types.

- exclude-operators
  - Exclude operator instances from Neutron execution (list of comma-separated names or indexes).
  - Operators are identified with the name of the output tensor or the index (location) in the model.
  - For example, use `--exclude-operators 13,14` or `--exclude-operators name13,name14` to exclude from being the specialized operator instances whose output tensors have locations 13 and 14 with names name13 and name14.
- include-only-operators
  - Only include operator instances to Neutron execution (list of comma-separated names or indexes).
  - Operators are identified with the name of the output tensor or the index (location) in the model.
  - For example, use –include-only-operators 13,14 or –include-only-operators name13,name14 to specialize only the operator instances whose output tensors have locations 13 and 14 with names name13 and name14.
- include-only-between-input-tensors
  - Only include operator instances limited by these input tensors to Neutron execution (list of comma separated names or indexes).
  - For example, use `--include-only-between-input-tensors 1,2` or `--include-only-betweeninput-tensors input1,input2` to specialize only the operator instances limited by the input tensors with locations 1 and 2 with names input1 and input2.
- include-only-between-output-tensors
  - Only include operator instances limited by these output tensors to Neutron execution (list of comma separated names or indexes).
  - For example, use `--include-only-between-output-tensors 1,2` or `--include-onlybetween-output-tensors output1,output2` to specialize only the operator instances limited by the output tensors with locations 1 and 2 with names output1 and output2.

Note: For the above options:

- Operators/tensors are identified NOT in the original model but in the model right before the extraction. That model can be dumped using the option –dump-before-extract.
- Tensors are identified either by their index or name. For example, you can view the location attribute or the name in the name attribute in the Netron visualizer).
- Operators are identified either by their index or the name of the output tensor because the operators do not have a standalone name attribute. For example, you can view the location attribute in the Netron visualizer).
- If the operator has multiple output tensors, then the name of the first output tensor is considered as operator name identifier.

Some useful debugging options include:

- *dump-before-extract <model_path>* — Dump the model before operator extraction to the given path, if not empty.
- *dump-after-extract <model_path>* — Dump the model after operator extraction to the given path, if not empty.
- *dump-header-file* — Serialize the output TensorFlowLite model as a header file.

You can check all the standard TFLite operator type identifiers, used by the `exclude-operator-types` and `include-only-operator-types` options, by using the `show-operator-types` option:

```
neutron-converter --show-operator-types
```

You can check all the command-line options using the `help` option:

```
neutron-converter --help
```

5.8 Run a MCX N series example

To use and run the MCX N series example, perform the following steps.

1. Navigate to https://mcuxpresso.nxp.com/en/welcome.
2. Click Select Development Board to build and download a new package.
3. Log in with your *email address* and *password*.
4. Select a board or kit to get started.
5. Click the Build MCUXpresso SDK button and select the latest version.
6. Select the eIQ middleware component in the software component selector on the MCUXpresso SDK Builder page.
7. To download the SDK compatible with your MCX N board, click the Download SDK at the bottom of the MCUXpresso SDK Builder page.
8. Download the archive and run any of the eIQ examples.

6 Extension framework

Extension Framework allows users to extend the eIQ Portal application with their own features. The Framework was inspired by the Extension Framework in Visual Studio Code, so users that can write extensions for this tool can find it similar.

The extensions can do the following:

Extensions are able to:

- Register named functions (commands) that can be executed manually or from a UI gesture, for example clicking a button.
- Create a custom webview (new screen) and fill it with custom UI elements.
- Get the installed extensions and the public API that they export. This can be used to create companion extensions.
- Access the project opened in eIQ Portal and manipulate its dataset.
- Run Python scripts from a separate virtual environment derived from the eIQ Portal's Python environment.

There are several pre-built extensions being released alongside the eIQ Toolkit. See *Extensions* for details.

6.1 Building a simple extension

This section describes the basic steps to create a simple extension. Further on in this chapter, more advanced options are explained.

Now we are going to create an extension that adds a new screen. This screen displays the runtime information - the ID of the current process, the NodeJS version, and the path to an open project (if there is one).

6.1.1 Extension folder structure

All extensions are installed in %USERPROFILE%/.eiqportal/extensions. An extension is a standard NPM package. To create a new extension, create a new directory in the install folder. For a simple run-time extension, let's create the runtime_extension directory.

The folder should have this structure:

```
\---runtime_extension
    |   package.json
    |   tsconfig.json
    +---dist
    |    |   extension.js
    +---src
    |    |   extension.ts
    +---node_modules
```

The node_modules and dist folders do not have to be created manually. The node_modules folder is generated when the external packages used by the extension are installed by running the npm install –dev command. The dist folder is created when the project is compiled. You can specify the command for compilation in the package.json file (see *Contents of `package.json` file*)).

6.1.2 Contents of `package.json` file

This file contains information about the extension, additional NPM packages, and contributions. Let's explain some of the items using an example. The package.json file for the run-time extension is as follows:

```
{
    "name": "runtime-info",
    "publisher": "nxp",
    "displayName": "Runtime Info",
    "description": "Runtime Info - Webview API Sample",
    "main": "./dist/extension.js",
    "version": "0.0.1",
    "private": true,
    "scripts": {
        "build": "tsc --project tsconfig.json",
        "watch": "tsc --watch --project tsconfig.json"
    },
    "contributes": {
        "commands": [
            {
                "command": "runtimeInfo.openWorkspace",
                "title": "Runtime Info"
            }
        ],
        "menus": {
            "frame/help": [
                {
                    "command": "runtimeInfo.openWorkspace"
                }
            ]
        }
    },
    "devDependencies": {
        "@types/node": "14.x",
        "eiqextension": "^0.1.1",
        "typescript": "^4.7.4"
    }
}
```

- **name** defines the extension identifier.
- **displayName** is a name that is displayed, for example in a list of available extensions.
- **main points to a file with the `activate()` function that will be run when the extension is activated.
- **scripts defines a list of scripts that can be run using `npm run \<script_name\>`. For example, running `npm run build` builds the extension and creates the `dist` folder.
- **contributes** defines static contributions. Here you can register various commands. For example, the `runtimeInfo.openWorkspace` command creates the extension's webview.

- **menus** allows extensions to add new items into menus in the main application. Using *frame/help* adds a new item to the “Help” menu. It is connected to the `runtimeInfo.openWorkspace` command. This means that the command executes when you click this new menu item. See *Available menus* for the list of menus where you can add new entry points for the extensions.
- **devDependencies** defines other packages that the extension is using. The extension framework provides two additional packages. One of them is *eiqextension*, which provides an API to the extensions through which they can manipulate objects available in the main application, for example opened projects. Another one is *eiqextension-webview* that provides the WebView API used to post messages to the webview (see *Messaging in Extension* or *Example*).

6.1.3 `tsconfig.json` file

Extensions are written in the Typescript language. They are a separate project that is configured by this file. It specifies the root files and the compiler options. For more information, see Typescipt documentation.

6.1.4 Extension’s entry point, `extension.ts` file

This file should contain two main functions (`activate()` and `deactivate()`). The activate() function is called during the application start, when all the extensions are activated. The deactivate() function is called when the application is closing and all the extensions are deactivated. Let’s now look and explain what the activate function for our simple Runtime extension looks like.

First of all, we defined the `runtimeInfo.openWorkspace` command in the `package.json` file and connected it to a new menu item. Now we need to define what happens when you click this menu item. By calling `registerCommand` from the eiqextension API, we register the command handler, but do not execute it. The execution is triggered by the framework when it registers a click on a menu item.

```
import * as eiqextension from 'eiqextension'; //import Extension API
export function activate(context: eiqextension.ExtensionContext): void {
    context.subscriptions.push(
        eiqextension.commands.registerCommand('runtimeInfo.openWorkspace', () => {
            //command handler
        })
    );
}
```

The `activate()` function takes one input parameter (`context`). The context object contains various information about the extension’s context. It consists of a path to the folder where the extension is installed, the extension object itself, subscriptions, project storage, global storage, and paths to those storages. The `registerCommand()` function is not called by itself, but it is pushed into a subscriptions array. This array can contain any object that is `Disposable`. These disposable objects are automatically disposed of when the extension is deactivated. In this case, the registered command is unregistered. If this was not handled by the subscriptions array, the command would have to be manually unregistered in the `deactivate()` function.

In the body of the `runtimeInfo.openWorkspace()` command, we define our new screen:

```
import * as eiqextension from 'eiqextension'; //import Extension API
import crypto = require('crypto');

// Keep track of whether the workspace is already showing
let workspace: undefined | eiqextension.WebviewWorkspace = undefined;
```

(continues on next page)

(continued from previous page)

```
export function activate(context: eiqextension.ExtensionContext): void {

  context.subscriptions.push(
    eiqextension.commands.registerCommand('runtimeInfo.openWorkspace', () => {

      if (workspace !== undefined) {
        return; // Ignore the command if the webview was already open
      }

      workspace = eiqextension.window.openWebviewWorkspace('Runtime info', {
        // Enable JavaScript in the webview
        enableScripts: true,
        // Disable serving of local files in the webview
        localResourceRoots: []
      });

      context.subscriptions.push(workspace);

      workspace.onDidDispose(() => {
        // Unset variable when the webview is disposed, for example closed by user
        workspace = undefined;
      });

      // Accept messages coming from the webview content
      workspace.webview.onDidReceiveMessage(message => {
        switch (message.command) {
          case 'close':
            // Dispose webview when the content sends a close message
            workspace?.dispose();
            return;
        }
      }, undefined, context.subscriptions); //Remove this listener when the extension is␣
↪deactivated;


      const nonceScript = crypto.randomBytes(16).toString('base64');
      const nonceStyle = crypto.randomBytes(16).toString('base64');
      workspace.webview.html = `
      <html>
        <head>
          <meta http-equiv="Content-Security-Policy" content="default-src 'none'; script-
↪src 'nonce-${nonceScript}';
          style-src 'nonce-${nonceStyle}'">
          <style type="text/css" nonce="${nonceStyle}">
            body {
              margin-left: 1em;
            }
          </style>
        </head>
        <body>
          <h1>Runtime info</h1>
          <dl>
```

(continues on next page)

(continued from previous page)

```
            <dt>Process pid</dt>
            <dd>${process.pid}</dd>
            <dt>Extension host Node.js version</dt>
            <dd>${process.version}</dd>
            <dt>Opened project path</dt>
            <dd><samp>${eiqextension.project.activeProject?.path || 'None'}</samp></dd>
          </dl>
          <button class="button" id="close-button">Close</button>

          <script nonce="${nonceScript}">
            (function() {
              const eiqportal = acquireEiqPortalApi();
              const closeButton = document.getElementById("close-button");

              closeButton.addEventListener("click", () => {
                eiqportal.postMessage({
                  command: 'close'
                });
              });
            }())
          </script>
        </body>
      </html>
      `;
    })
  );
}
```

The part with setting up a `webview` and disposing of it when the screen is closed is self-explanatory. Let's have a closer look at the HTML part, where the actual graphical interface is created.

The first step is to set the CSP to protect against an XSS attack:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'none'; script-src
↪'nonce-${nonceScript}'; style-src 'nonce-${nonceStyle}'">
```

Then we define the body of the screen:

```
<h1>Runtime info</h1>
<dl>
  <dt>Process pid</dt>
  <dd>${process.pid}</dd>
  <dt>Extension host Node.js version</dt>
  <dd>${process.version}</dd>
  <dt>Opened project path</dt>
  <dd><samp>${eiqextension.project.activeProject?.path || 'None'}</samp></dd>
</dl>
<button class="button" id="close-button">Close</button>
```

Add a Javascript function that:

- Aquires the API for sending messages from the webview content to the extension:

```
const eiqportal = acquireEiqPortalApi();
```

- Adds the `onClick` event to the close button that sends the message to the extension to close the screen:

```
closeButton.addEventListener("click", () => {
  eiqportal.postMessage({
    command: 'close'
  });
});
```

The first extension is ready and can be loaded into the eIQ Portal. When distributing the extension, the `src` folder and the `tsconfig.json` file can be omitted.

6.1.5 Available menus

The `menus` item in the `package.json` configuration describes the places in the eIQ Portal graphical interface, where a new menu item or button is added as a part of an extension. The extension can then define a command that executes when you click on this newly added item. In the *Simple extension example*, we used *frame/help*, because this is the place where a new menu item was added. Here is a list of all available places that can be extended in this way:

1. **frame/help** adds a new menu item into the "Help" menu:

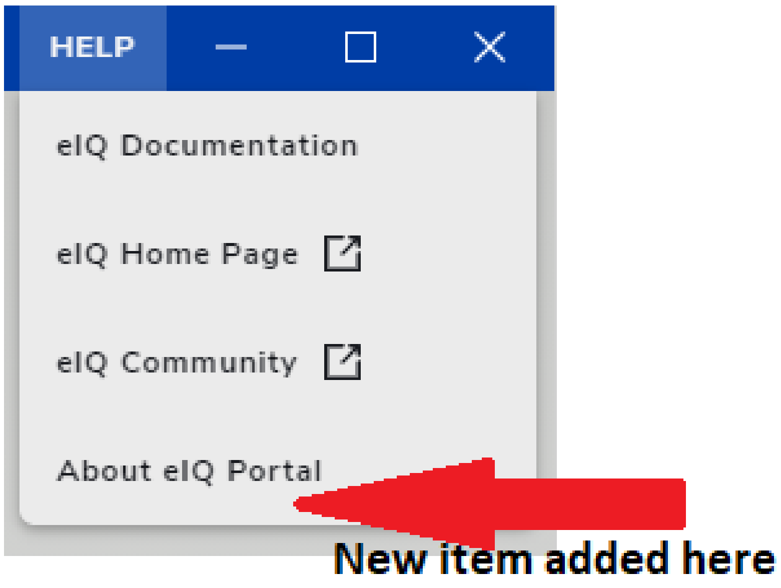


Figure 6.1.: Help menu extension

2. **frame/workspaces** adds a new menu item into the "Workspaces" menu:
3. **dataset/utility** adds a new button to the sidebar of the "Dataset Curator" screen:
4. **home/tools** adds a new button to the "Home" screen, into the third row of buttons:
5. **deploy/export-settings** adds a new button to the sidebar of the "Export Model" screen:

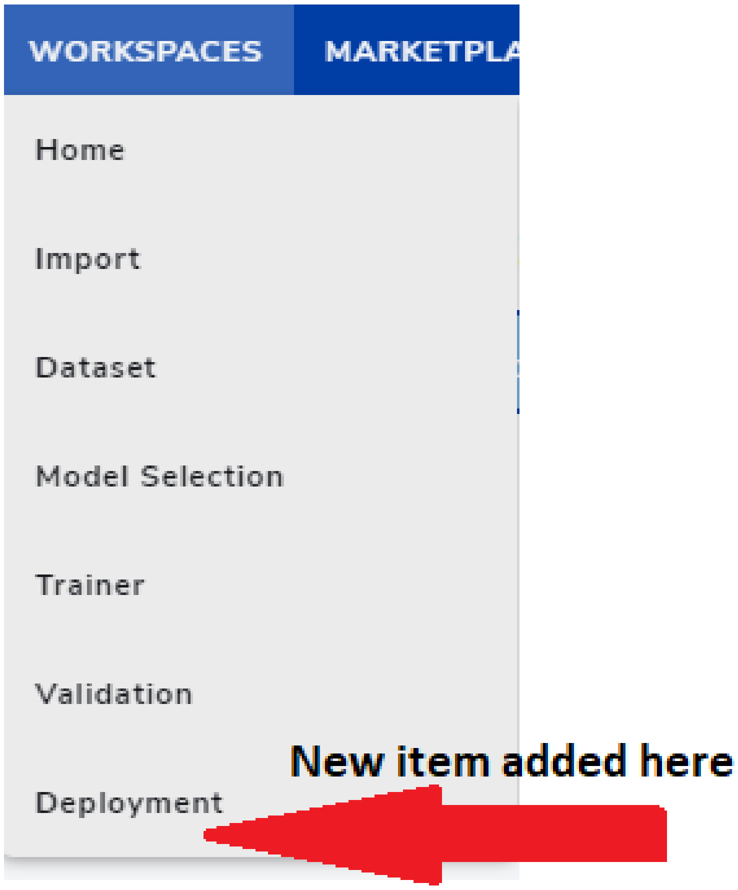


Figure 6.2.: Workspace menu extension

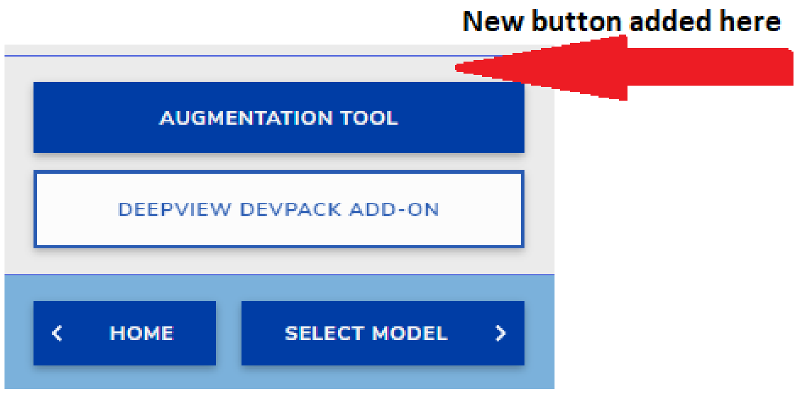


Figure 6.3.: Dataset sidebar extension

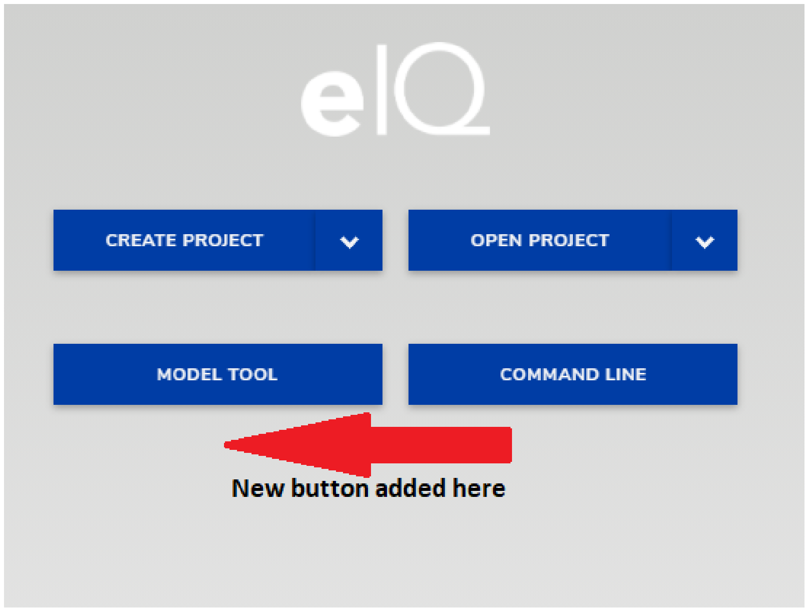


Figure 6.4.: Home screen extension

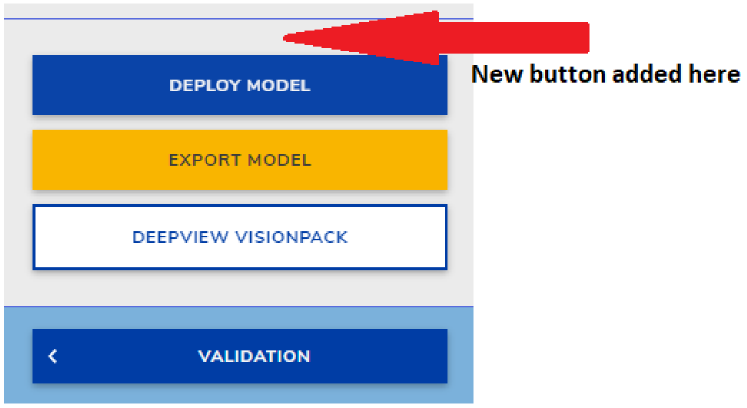


Figure 6.5.: Export sidebar extension

6.2 Advanced usage of the Extension API

This section describes how to use some advanced settings and functions that are provided to an extension through the Extension API. There is an example on how to use the `Vue.js` front-end framework with the extension and create bi-directional communication between the frontend and the extension. It describes how to condition a display of a menu item, how to use Python virtual environment to run custom Python scripts, manipulate open project, and how to use global and project storage.

6.2.1 Using `Vue.js` frontend framewrok with the extension

Using some kind of framework to handle the user interface is more convenient than just writing everything in plain HTML and Javascript. Therefore, this section describes how to integrate the `Vue.js` framework into the extension.

There are two main differences from the *simple extension example*. The `eiqextension-webview` API provides only a function to post messages from the web view content to the extension. However, sometimes we need the extension to send some information back to the web view content to display it or process it in other ways. For this purpose, we define the separate `Messaging plugin`. We also modify the content of `webview.html` to include the Vue front-end instead of simply displaying the screen.

To add all the files related to the Vue framework, we extend the *basic folder structure*:

```
\---runtime_extension
    |   package.json
    |   tsconfig.json
    +---dist
    |    |   extension.js
    +---src
    |    +---extension
    |    |      |   extension.ts
    |    |      |   workspace.ts
    |    +---vue
    |    |      +---components
    |    |      +---src
    |    |      |   |   App.vue
    |    |      |   |   main.ts
    |    |      |   +---assets
    |    |      |   +---messaging
    |    |      |   |       |  messaging.ts
    |    |      |   |       |  plugin.ts
    |    |      |   |       |  types.d.ts
    +---node_modules
```

The screen layout is defined in `App.vue`. The `main.ts` file configures this and adds the `Messaging plugin` to the front-end:

```
import Vue from 'vue';
import App from './App.vue';
import MessagingPlugin from './messaging/plugin';

Vue.config.productionTip = false;

Vue.use(MessagingPlugin);
```

(continues on next page)

(continued from previous page)

```
new Vue({
  render: h => h(App),
}).$mount('#app');
```

The `assets` subfolder can contain various `.css` files or other typescript modules used in the front-end. The `components` folder may contain other Vue components used by `App.vue`, for example a custom progress bar. The `messaging` subfolder then contains everything needed to set up the communication between the front-end and the extension. This extended folder structure is just an example of how it can look. Otherwise, it is up to the developers how they organize their source files and set it in the configuration files.

6.2.1.1 Messaging plugin

We want the messaging plugin to implement bi-directional communication between the Vue front-end and the extension. Let's declare the functions used for this communication:

```
export interface IMessaging {
  send<U extends any[] = any[]>(channel: string, ...args: U);
  receive<T extends any[] = any[]>(channel: string, func: (...args: T) => void, once?:␣
↪boolean);
  receiveOnce<T extends any[] = any[]>(channel: string, func: (...args: T) => void);
  request<T, U extends any[] = any[]>(channel: string, ...args: U);
}
```

This section does not contain the full implementation of these functions. It only explains the idea behind the IMessaging interface and what Extension API functions can be used to reach this goal. The implementation is left for extension developers to figure out and customize as needed.

The `send` function is simple. The front-end sends values (`args`) through a channel (`channel`) to the extension. A familiar web view API `postMessage()` function can be used to achieve this. The extension can then handle this message using the `webview.onDidReceiveMessage()` function.

The extension can also send a message to the front-end. In this case, the front-end should be able to receive the message and react to it. That's what the `receive()` and `receiveOnce()` functions do. The receive function can be implemented using the `window.addEventListener()` and `window.removeEventListener()` functions. The idea is to add an event listener for a channel. When a message with this channel comes, execute the `func` front-end function to handle the received data.

There is one more type of message - `request`. It can be used instead of the `send()` and `receive()` pairs. The request sends a message to the extension and expects that the extension sends back a response.

6.2.1.2 Setting up HTML page

The body of the HTML page (in this case) contains only a path to a Javascript file. It should point to a file generated after the compilation (see the `npm run build` command in the *simple extension example*).

```
webview.html = `
<html lang="">
    <head>
        code omitted
    </head>
    <body>
        <div id="app"></div>
```

(continues on next page)

(continued from previous page)

```
        <script async type="text/javascript" src="${webview.asWebviewUri(path.join(dist,
↪'js/app.js'))}"></script>
    </body>
</html>
`
```

Notice that the example uses the `webview.asWebviewUri` method. We use it because web views cannot directly load resources from the workspace or local file system using file: uris. The `asWebviewUri()` function takes a local path and converts it into a "uri" that can be used inside a web view to load the resources.

The head tag of the HTML string must contain the Content Security Policy. In this example, `webview.cspSource` is used. It contains the CSP source for webview resources. Styles can also be added in a similar way as the Javascript script in the <body> element:

```
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <meta http-equiv="Content-Security-Policy" content="default-src 'none';
    style-src ${webview.cspSource} ;
    script-src ${webview.cspSource} ;">
    <title>ArmVela</title>
    <link href="${webview.asWebviewUri(path.join(dist, 'css/app.css'))}" rel="preload"␣
↪as="style">
    <link href="${webview.asWebviewUri(path.join(dist, 'css/app.css'))}" rel="stylesheet
↪">
</head>
```

6.2.2 Conditioning the display of menu items

You can make an item in a menu (see *Available menus*) appear only when certain conditions are met. To set this, modify the `package.json` file by adding the `when` clause:

```
"contributes": {
    "menus": {
      "deploy/export-settings": [
        {
          "command": "example.openWorkspace",
          "when": "modelExported && exportModelQuantized"
        }
      ]
    },
}
```

This menu item is displayed when the model is exported and the exported model is also quantized. The `modelExported` and `exportModelQuantized` words are called "context keys". They are defined in the extension framework and you can use only them to form an expression. The expression is then evaluated according to the current state of the eIQ Portal application. If the condition is met, the menu item is displayed. Context keys do not have to be only true or false. They can be set to any value. For example, another context key available (`exportModelType`) contains the model type of an exported model. If you want to use it in the expression, you must test it for a specific value. For example:

```
"when": "modelExported && exportModelType==tflite"
```

Currently, there are only these three context keys available, but they might be extended in the future.

6.2.3 Using Python virtual environment

The Extension Framework allows extensions to use the Python environment to run scripts. The virtual environment is derived from the Python environment used in eIQ Portal. This allows the derived virtual environment to use all the packages already installed in the base environment without taking too much space on the disk.

To use this feature, the extension can use a `PythonEnvironment` that is automatically created on first use. The `PythonEnvironment` is accessed using `context.pythonEnvironment` where `context` is the `ExtensionContext` that is passed to the `activate` function.

When you obtained the `PythonEnvironment` object, the following functions are available:

- **`installModule(moduleName, version)`** installs a module into the environment. You can install new packages that are missing in the base environment or install a different version of an existing package. The `moduleName` parameter should contain the same name of the module as when it is installed using `pip`. The `version` parameter is optional. If it is not provided, the latest version of the package is installed. The code example is as follows:

```
try {
    await context.pythonEnvironment.installModule('ethos-u-vela');
    await context.pythonEnvironment.installModule('tensorflow', '2.6');
}
catch(error){
    console.log("Error while installing packages.");
}
```

- **`uninstallModule(moduleName)`** uninstalls a given module::

```
await context.pythonEnvironment.uninstallModule('tensorflow');
```

- **`getModules()`** provides all installed modules in the virtual environment:

```
const modules = await context.pythonEnvironment.getModules();
console.log(modules);
```

- **`execScript(script, variables)`** executes the provided script. The script can only be provided as a string.

  Providing the `variables` argument allows you to pass variable values from Javascript to the Python script.

```
{
    "variable_name": value
}
```

  For example:

```
const checkpoint = eiqextension.project.activeProject?.activeModelCheckpoint;
{
    "x": 72,
    "some_string": "This is string",
    "my_array": [1,2,3],
```

(continues on next page)

(continued from previous page)

```
    model: checkpoint.kerasModel,
}
```

You can use the names of these variables in your script. They are initialized to the values provided.

If you want to initialize a variable of the tuple type, you must provide it as a string, because Javascript does not know this type. The string is then processed so that the variable can be created as a tuple.

The full example of using the `execScript()` function is as follows:

```
context.pythonEnvironment.execScript(
`import tensorflow as tf
a = tf.constant([[1.0, 2.0, three], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)
print(c)`, { 'three': 3.0 })[0]
.then(output => console.log(output.stdout));
// Outputs: tf.Tensor(
// [[22. 28.]
//  [49. 64.]], shape=(2, 2), dtype=float32)
```

If you want to run the Python virtual environment from a command line, use the eIQ Portal's command line. Because the environment is derived from the eIQ Portal's environment, is relies on some environment variables to be set. They are set during eIQ Portal's startup and used when the command line is launched.

6.2.4 Storage

Extensions can use a storage utility to store general values or values related to the opened project. The extension context contains two objects (`projectState` and `globalState`). These objects can be used to store and retrieve values.

To check all available keys, you can call `storage.keys()`, where storage is either `projectState` or `globalState`.

To get the value of a stored key, use the `storage.get(key, defaultValue)` method. The `defaultValue` parameter is optional. If there is no value for the given `key`, `defaultValue` is returned.

To update the storage, you can use `storage.update(key, value)`. This function can update the existing `key` with a new `value`, create a new `key-value` pair, or remove a `key` from the storage if `undefined` is passed as a value.

6.2.5 Handling an opened project

This section describes the API that is provided to the extension for dataset manipulation.

You can access an opened dataset using `eiqextension.project.activeProject.dataset`. This allows you to get the information about the dataset or manipulate it. In fact, you can change three items - labels, annotations, and images.

All labels in the dataset are accessible using `dataset.labels`. To get annotations and images, you must use the `dataset.getAnnotations(filter)` and `dataset.getImages(filter)` methods. You can filter images according to label, label ID, or group (train or test). Annotations can be filtered according to the image, label, or group. See the `AnnotationsFilter` and `ImagesFilter` interfaces declared in the Extension API on how to use them.

To change label or annotation, you can use the `dataset.updateLabel(label, updatedLabel)` and `dataset.updateAnnotation(annotation, updatedAnnotation)` methods. Images cannot be changed, they can be only added or deleted.

You can add new labels, images, and annotations. Use `dataset.addLabel(label)`, `dataset.addAnnotation(annotation)`, and `dataset.addImage(image, group)` to add them. Defining a group for a new image is optional. If it is not provided, the image is not assigned to any group and it is not a part of the training and validation processes.

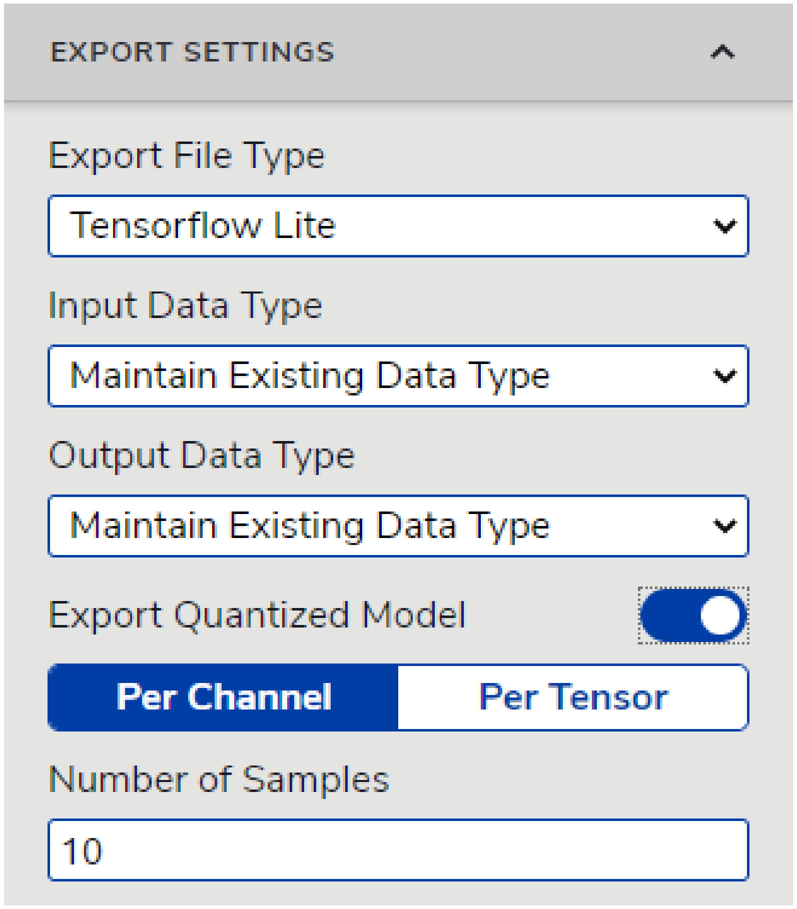
You can delete images, labels, and annotations using the `dataset.deleteLabel(label)`, `dataset.deleteAnnotation(annotation)` and `dataset.deleteImage(image)` methods. To identify which label, image, or annotation should be deleted, you can either pass an object of that type to the function (or only its ID).

6.3 Extensions

This section describes the extensions currently available for the eIQ Portal. The procedure to install the extensions is simply to unzip the pre-built archives to %USERPROFILE%/.eiqportal/extensions/<directory>.

6.3.1 Arm Vela extension

This extension is used to convert a TFlite quantized model into an optimized version that is prepared to run on an embedded system that contains Ethos-U NPU, such as NXP i.MX 93. To make this extension visible, visit the "Export Model" page. Then you must export the model into a quantized TFlite format as follows:


EXPORT SETTINGS

Export File Type: Tensorflow Lite

Input Data Type: Maintain Existing Data Type

Output Data Type: Maintain Existing Data Type

Export Quantized Model: on

[✓ Per Channel] Per Tensor

Number of Samples: 10

Figure 6.6.: Settings for Arm Vela extension

After you export the model, new buttons appear on a screen. To open the Arm Vela extension, click the “Arm Vela Conversion” button:

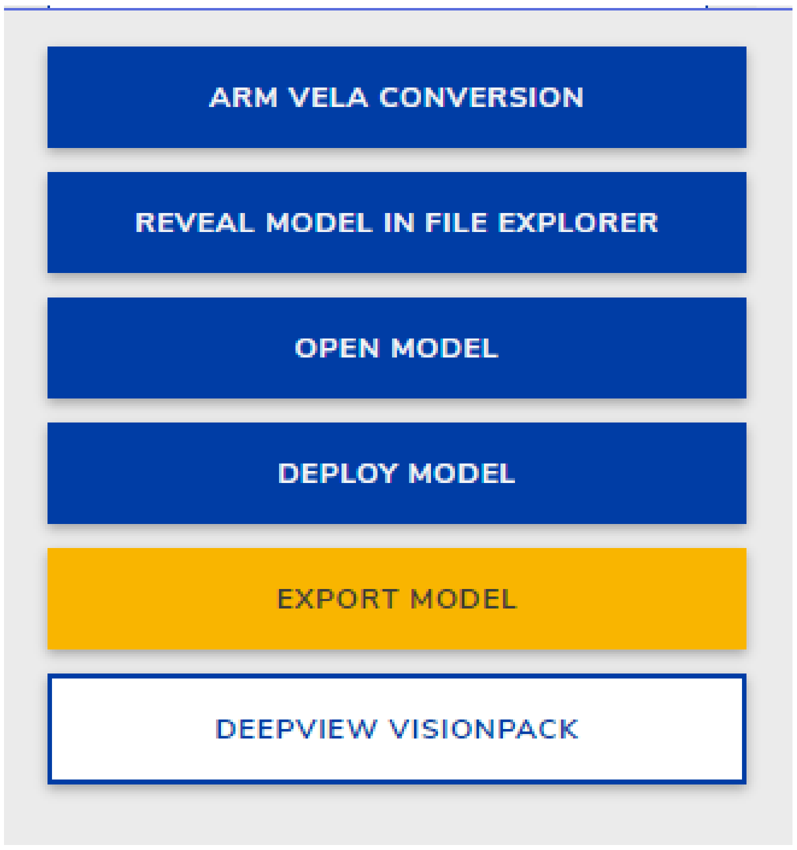


Figure 6.7.: Buttons after export model

A new screen appears. This is where you can update the settings for conversion into the Vela-optimized model.

There are two options, that are already filled in:

Model Path: The path to the model exported into a quantized TFlite format. You can change this if you want, but you have to make sure that the model you are trying to convert is in a quantized TFlite format. Otherwise the next conversion fails.

Configuration file: Path to the configuration file where you can set hardware-specific options. The extension is already installed with a prepared configuration file that will work with i.MX93 boards.

After you have all settings set, you can start the conversion by clicking the “Export Vela Optimized” button. You are prompted to select a **directory** where the converted model will be saved together with the performance estimate document.

When the conversion ran successfully, the performance estimate document is parsed and some of the information about the converted model is displayed as a table:

The performance estimate is just an experimental feature in the Vela tool. The information it contains may differ from the real performance of the hardware.

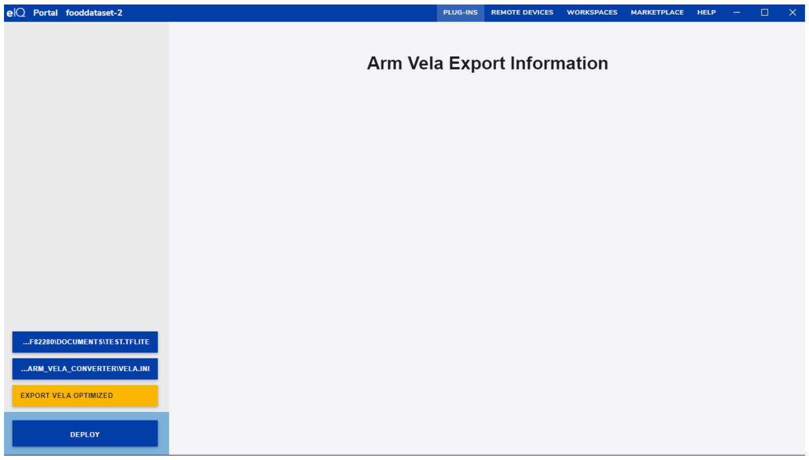
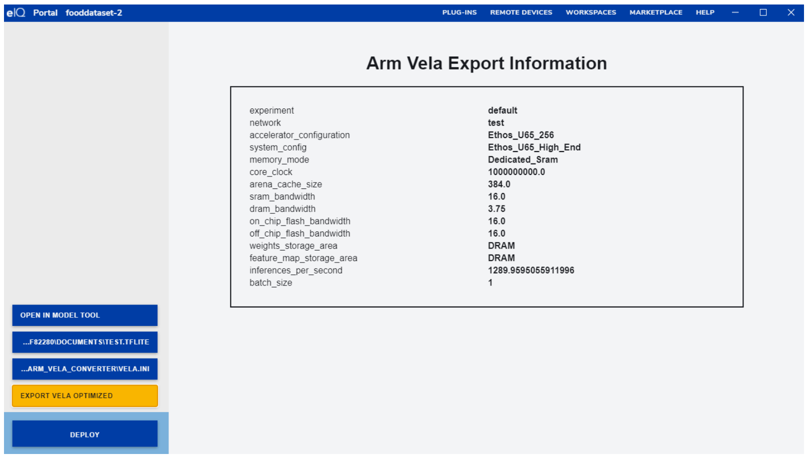


Figure 6.8.: Arm Vela extension screen



Arm Vela Export Information

experiment	default
network	test
accelerator_configuration	Ethos_U65_256
system_config	Ethos_U65_High_End
memory_mode	Dedicated_Sram
core_clock	1000000000.0
arena_cache_size	384.0
sram_bandwidth	16.0
dram_bandwidth	3.75
on_chip_flash_bandwidth	16.0
off_chip_flash_bandwidth	16.0
weights_storage_area	DRAM
feature_map_storage_area	DRAM
inferences_per_second	1289.9595055911996
batch_size	1

Figure 6.9.: Arm Vela export information

6.3.2 Explainability extension

Making predictions from ML models more human-understandable is crucial for having trust in an ML model. Explanations for predictions help model developers understand incorrect predictions and discover model biases.

The Explainability extension provides tools for understanding the predictions of an ML model trained or imported in eIQ Portal. You can easily filter predictions to explore the types of mistakes of a model. The prediction tool provides more details about a selected prediction, such as the label confidences and all detected bounding boxes. The explanations tool (currently only available for classification models) makes predictions understandable by visualizing the part of the input that was most important for the prediction. The similar training objects tool makes classifier predictions understandable by identifying training objects which are classified based on similar features.

6.3.2.1 Prediction analysis workspace

The Explainability extension adds a prediction analysis workspace to eIQ Portal. This workspace is accessible from the *Workspaces* menu after opening or creating a project. Typically, you will open the prediction analysis workspace after you have validated a trained model. Besides via the *Workspaces* menu, you can also open the prediction analysis workspace by clicking a prediction in the right sidebar of the validation workspace and then clicking the *Analyze* button.

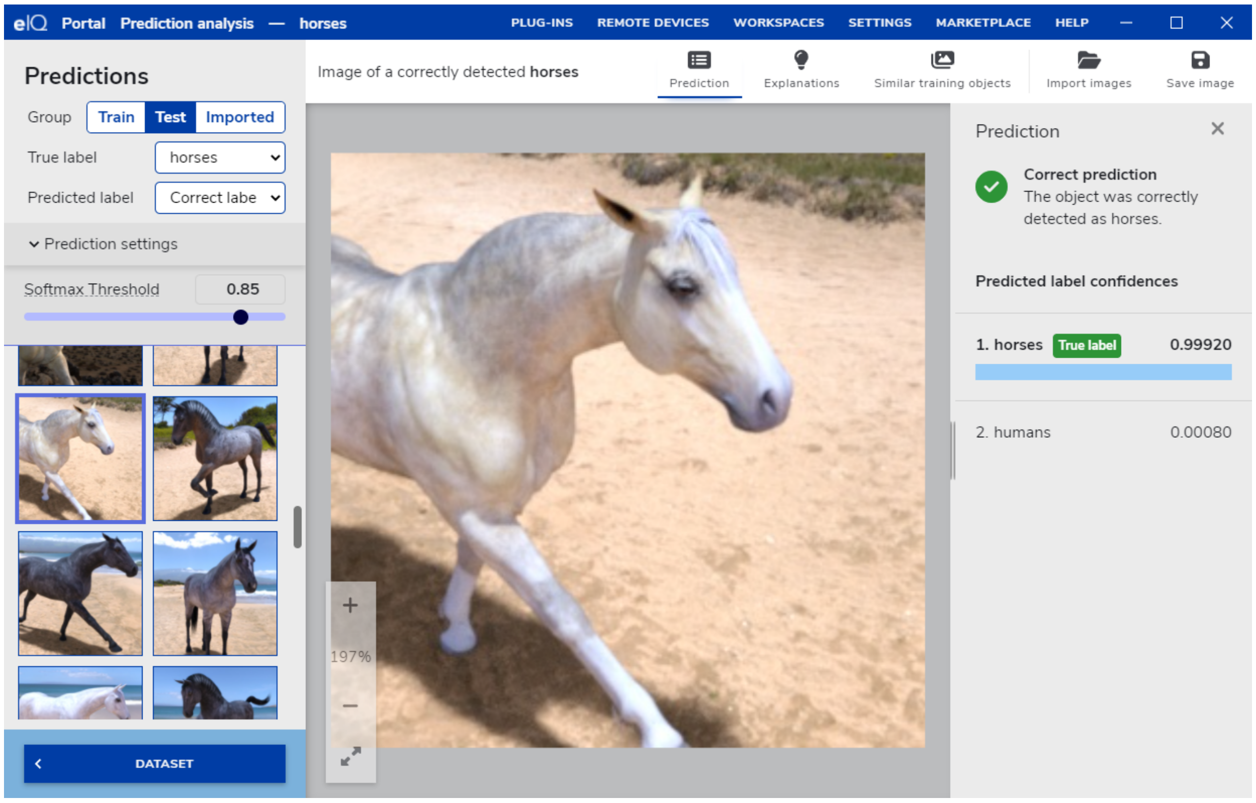


Figure 6.10.: Prediction analysis workspace

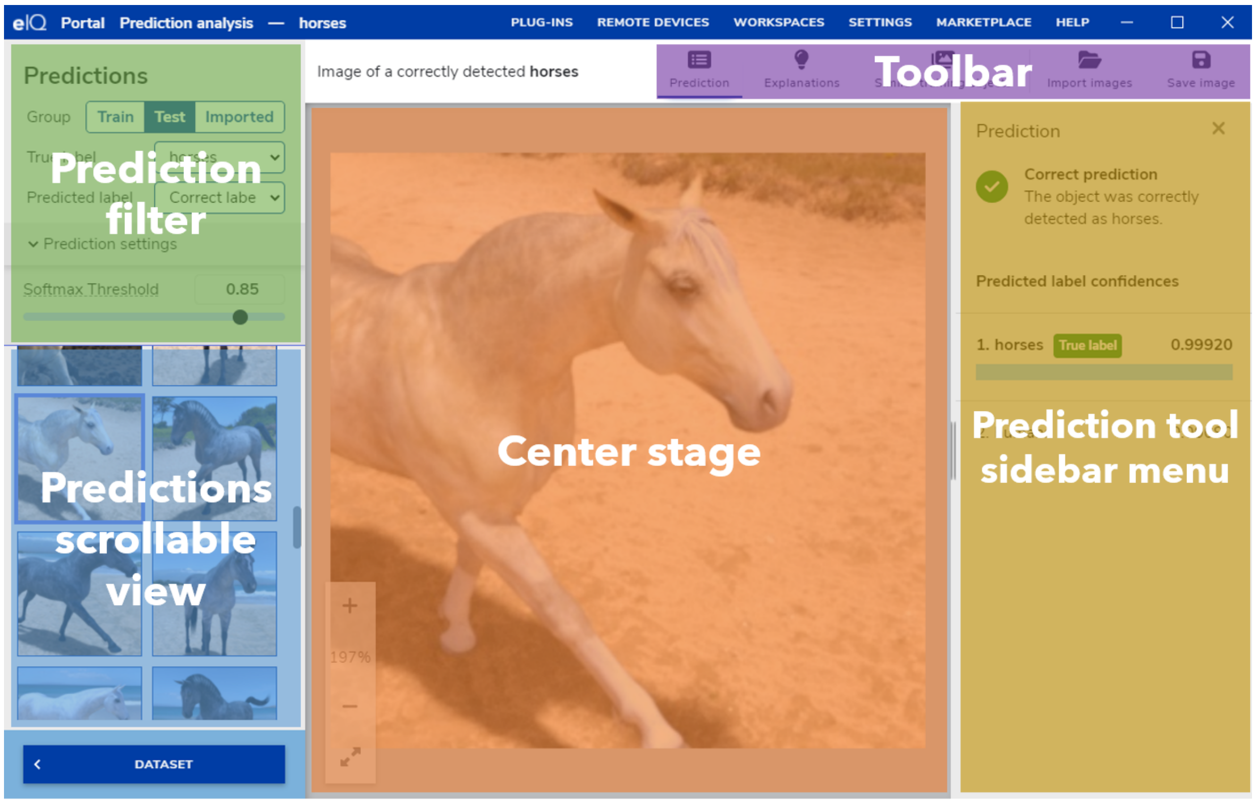


Figure 6.11.: Areas in the prediction analysis workspace

6.3.2.1.1 Exploring predictions

The prediction analysis workspace shows a tool for exploring the model predictions in the left sidebar. On the top, you can filter the predictions on dataset group (i.e., train set or test set), the true label, and the predicted label. In the prediction settings menu below it, you have the option to further filter based on the quality of the predictions. First, you can select the SoftMax threshold (minimum confidence for the predicted class). For object detection models, you can also configure the bounding box Intersection over Union threshold (metric for pairing ground truth bounding boxes with predicted boxes) and the Non-Max Suppression Intersection over Union threshold (metric for filtering the predicted bounding boxes). The scrollable view on the left bottom shows the predictions that match the selected filtering criteria. The bottom of the list shows the number of predictions that match the criteria.

Center stage

After selecting a prediction from the scrollable view, the center stage will show the image for this prediction. You can zoom and pan the image in the center stage by scrolling and dragging the image or by using the zoom controls in the bottom left of the stage. The bottom control in the zoom controls enables the user to toggle between showing the image at actual size and showing the image at the largest size that fits the stage. If the model is an object detection model, the image will, besides the selected prediction, also display the ground truth bounding boxes and the predicted bounding boxes for all other objects on the image. You can hide these boxes by clicking the eye icon in the legend. Hovering over a bounding box with the mouse shows its label. Clicking a bounding box updates the selected prediction.

Importing images

Images can be imported into the workspace from the file picker that is shown after clicking the Import images button. Additionally, images can be imported by dragging them and dropping them into the workspace. Only JPEG, PNG, BMP, and GIF (without animation) image formats are supported. Imported images are stored with the project and are therefore saved when closing the project or the application. Imported images can be removed by clicking the X button in the top right in the predictions scrollable view. To remove all imported images at once, click Clear in the same view.

After importing an image, the workspace will run a prediction on it using the configured prediction settings. The predictions for imported images can be found by selecting Imported as the group in the prediction filter. The same tools are available for imported images as for the images from the train or test group.

Saving and copying images

Images shown on the center stage and in the Similar Training Objects tool can be copied to the clipboard or saved as a file by selecting the applicable option from the context menu that is opened by right-clicking an image. Clicking the Save image button in the toolbar will save the image that is shown in the center stage. Images will be saved as PNG files.

6.3.2.1.2 Prediction analysis tools

The top right of the prediction analysis workspace shows the toolbar with prediction analysis tools. The *Prediction* tool is available for all models whereas the *Explanations* tool is currently only available for classification models. Clicking one of the tools will open a sidebar menu on the right.

6.3.2.1.3 Prediction tool

The prediction tool shows details about the selected prediction that is displayed on the center stage.

For classification models, the predicted label confidences for all labels that the model was trained on are displayed. The blue bars under each label indicate the confidence visually for a quick-glance inspection. For models with many labels, you can quickly scroll to the true label by clicking the arrow in the message indicating the rank of the true label.

For object detection models, the prediction tool lists ground truth and predicted bounding boxes for all objects on the image of the selected prediction. The prediction tool pairs these boxes according to the configured Intersection over Union (IoU) for ground truth matching. The tool displays for all objects the confidence for the predicted label as well as the IoU of the predicted bounding box and the ground truth bounding box. The tool displays *N/A* (not applicable) as the IoU for false positives (predicted object that should have been background) and false negatives (undetected object). The bounding boxes of the selected prediction are highlighted, those of other predictions in the same image are partly transparent. Hovering over an object in the list will display the associated bounding boxes and labels in the center stage image. Clicking an object in the list will update the selected prediction.

6.3.2.1.4 Explanations tool

The explanations tool is currently only available for classification models. You can use the explanations tool to understand predictions by visualizing the part of the input that was most important for the prediction according to the model. After selecting a *target label*, the explanations tool will highlight areas in the input that are most important for having the input predicted as the target label. The highlighted areas are overlaid on top of the center stage image. A colormap on the bottom right of the center stage shows how to interpret the heatmap. The explanations tool offers an option for selecting the resolution of the generated heatmap. Here, a higher resolution offers more detail, but is also more noisy. The area where the colormap is located also displays the effective resolution of the heatmap.

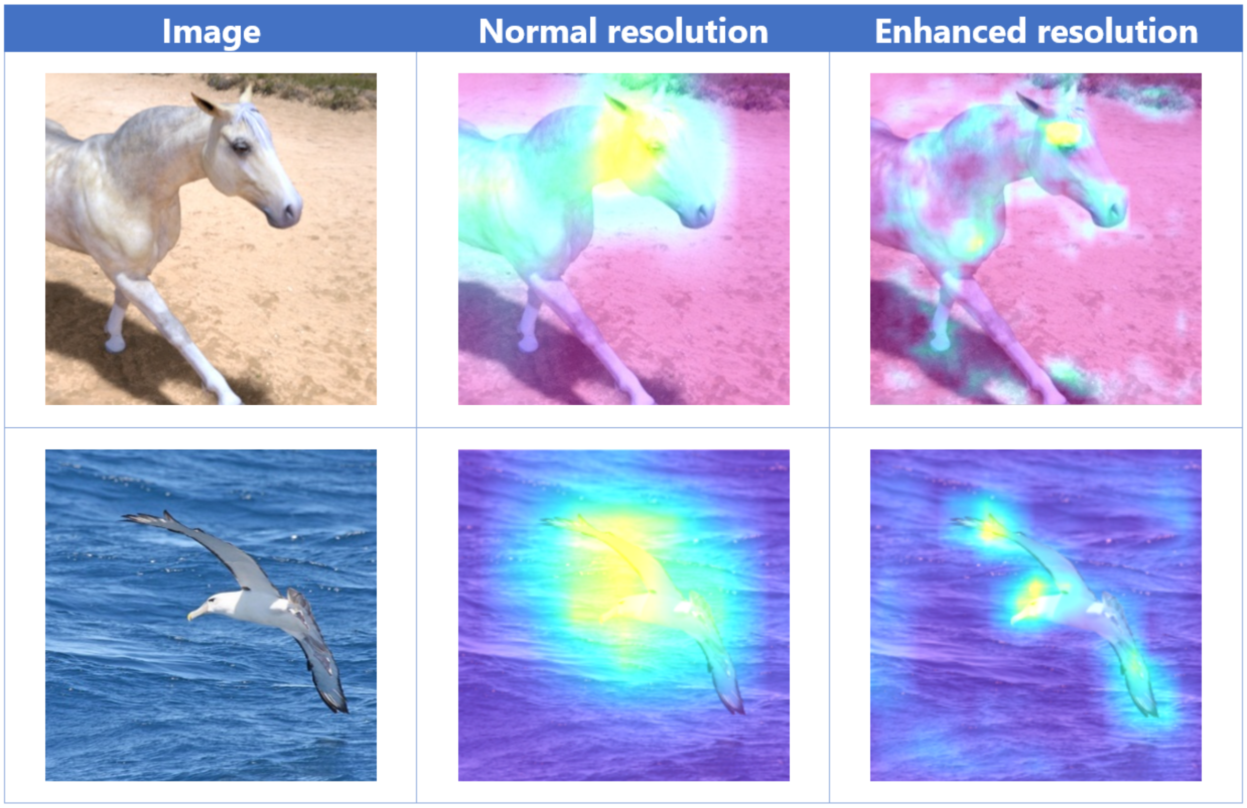
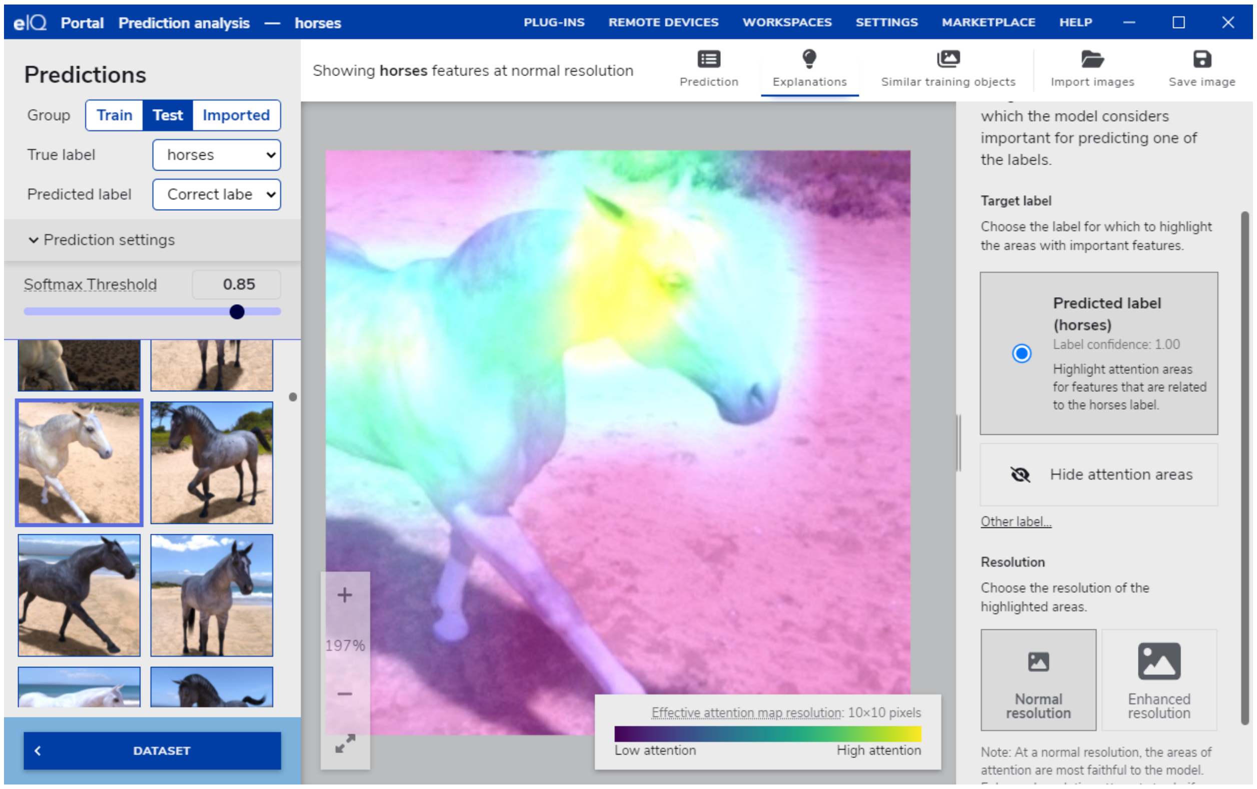


Figure 6.12.: Comparison of normal resolution and enhanced resolution explanations. In the top example, the normal resolution yields the best results as its resolution is high enough and the enhanced resolution heatmap adds noise. In the bottom example, the enhanced resolution yields the best results. It shows that for predicting the seagull, the most important parts of the input are the head of the seagull and the end of the wings. In the normal heatmap, the resolution is too small to see these details.

How the heatmap is generated

The highlighted areas are generated based on Grad-CAM *(Selvaraju* et al.*, 2017)*. This algorithm computes a weighted sum of the features from an intermediate model layer. The weights for the features are obtained by computing the gradient of the features toward the output logit of the selected target class. A ReLU operation is applied to the result of the weighted sum such that negative attention areas are filtered. The resolution of the Grad-CAM heatmap depends on the output shape of the intermediate model layer. The explanation tool uses the last intermediate model layer that preserves the locality of the features, e.g. the last convolutional layer. The heatmaps are interpolated for smoothness. The actual resolution of the heatmap is shown above the colormap. When using the *enhanced resolution* mode, the tool uses an NXP proprietary extension of Grad-CAM to generate higher resolution heatmaps. This proprietary algorithm



Group: Train [✓ Test] Imported
True label: horses
Predicted label: Correct labe
Softmax Threshold: 0.85

197%

Showing **horses** features at normal resolution

which the model considers important for predicting one of the labels.

Target label

Choose the label for which to highlight the areas with important features.

[✓ **Predicted label (horses)**]
Label confidence: 1.00
Highlight attention areas for features that are related to the horses label.

Hide attention areas

Other label...

Resolution

Choose the resolution of the highlighted areas.

[✓ Normal resolution] Enhanced resolution

Note: At a normal resolution, the areas of attention are most faithful to the model.

Effective attention map resolution: 10×10 pixels
Low attention — High attention

Figure 6.13.: Explanations tool for classification models

maximizes the heatmap resolution while maintaining heatmap faithfulness. Therefore, the effective resolution of the enhanced resolution heatmap may differ per prediction and target class.

Target class selection

The explanation tool warns if the label confidence for the selected target class is low. Because few features for such target class may be present in the image, there is a considerable risk that spurious attention areas appear in the heatmap.

6.3.2.1.5 Similar training objects tool

The similar training objects tool is currently only available for classification models. After selecting a prediction, the tool allows you to identify training objects that are predicted as the same class based on similar features. This enables you to understand important features and to debug prediction failures. The tool displays the top 20 most similar training objects. Each object shows in the top left corner the similarity rank and in the top right corner the cosine similarity metric with the selected prediction. On the top you can toggle between showing the images and showing the images with the influential areas highlighted in an overlay. There is also a slider to control the size of the displayed images.

Before the tool can be used, the features for all training objects need to be computed and stored. The tool panel shows a button for starting this process. This process needs to be repeated if a new model checkpoint is loaded or if the training dataset is modified.

Similarity ranking is performed using an NXP proprietary algorithm that computes the important features for all training objects. The important features are determined using an algorithm akin to the Grad-CAM explanation algorithm.

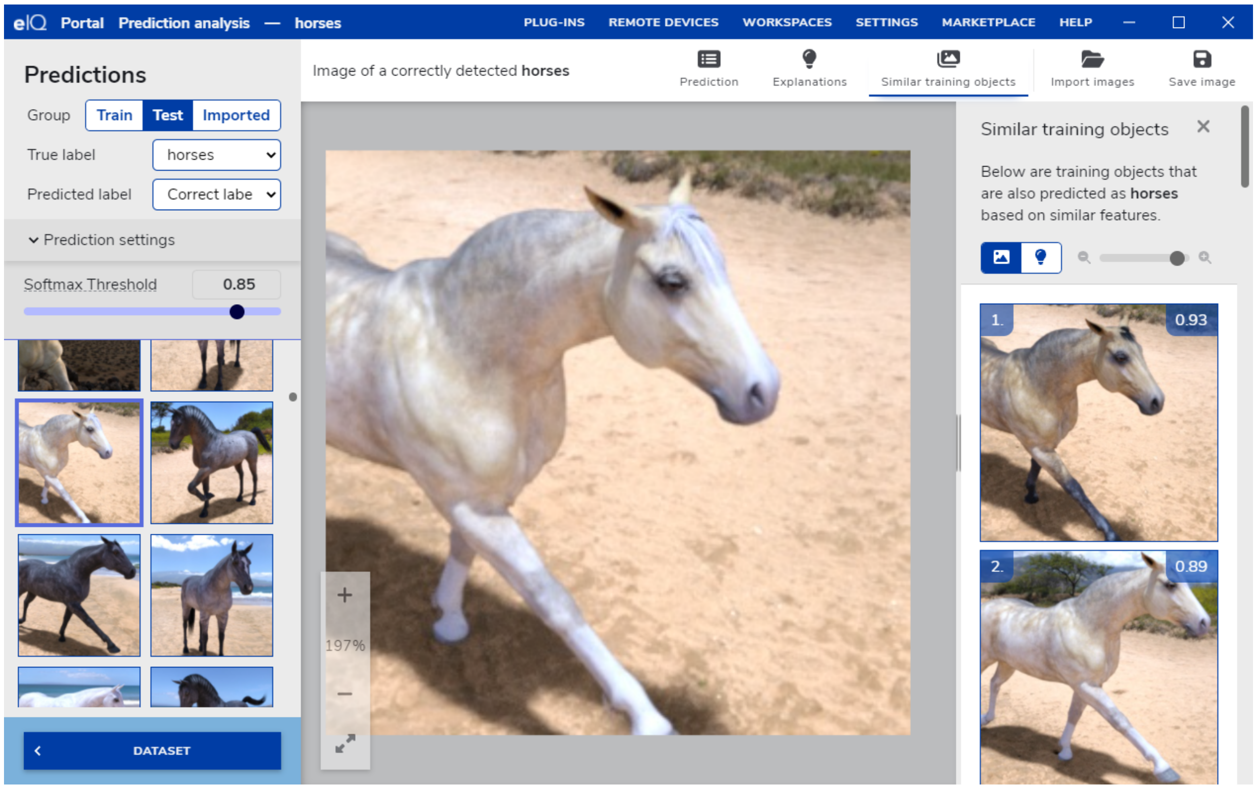

eIQ Portal Prediction analysis — horses
PLUG-INS REMOTE DEVICES WORKSPACES SETTINGS MARKETPLACE HELP
Predictions
Group Train Test Imported
True label horses
Predicted label Correct labe
Prediction settings
Softmax Threshold 0.85
DATASET
Image of a correctly detected horses
Prediction Explanations Similar training objects Import images Save image
197%
Similar training objects
Below are training objects that are also predicted as horses based on similar features.
1. 0.93
2. 0.89

Figure 6.14.: Similar Training Objects tool

6.3.2.2 References

Selvaraju, R. R. *et al.* (2017) 'Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization', *Proceedings of the IEEE International Conference on Computer Vision*, 2017-Octob, pp. 618–626. doi: 10.1109/ICCV.2017.74.

6.3.3 Vision Pipeline extension

The Vision Pipeline extension for eIQ Portal is a tool designed to visually verify a model that has been trained with eIQ Portal.

The extension connects to a remote device, builds a processing vision pipeline based on parameters that a user has selected (input source, output sink, model file), transfers the model that has been trained (along with any additional parameters whenever applicable), and runs this pipeline on the device.

The pipeline that runs on the remote device is built using the GStreamer multimedia framework and NNStreamer, a set of GStreamer plugins specifically designed for neural- network processing. For more information about how NXP uses GStreamer and NNStreamer to build neural-network processing pipelines, please refer to the i.MX Machine Learning User's Guide

While the pipeline is running, some performance metrics can be retrieved from the system so that it is possible to analyze how efficient the system is.

The high-level workflow for the Vision Pipeline extension is as follows:

1. Connect to a remote device.
2. Build a pipeline.

3. Deploy a pipeline.
4. Verify that the model is behaving as expected.

6.3.3.1 1. Supported features

This is the list of features supported by the extension:

- Video Input Stream Sources
  - Camera sensor connected to remote device (CSI2 or USB)
  - Video file located on host device
- Output Sinks for Video Stream
  - Remote device's local display
  - Network connection to host device
- Neural Network Model Formats
  - TensorFlow Lite
  - DeepView RT
- SoCs and Boards
  - i.MX 8MPlus SoC
  - Any i.MX 8MPlus-based development board

6.3.3.2 2. Getting acquainted with the main UI

When the extension has been installed (see the eIQ Toolkit documentation to understand how to do so), the "VISION DEPLOYMENT" button appears in eIQ Portal home screen (Figure 6.15.).

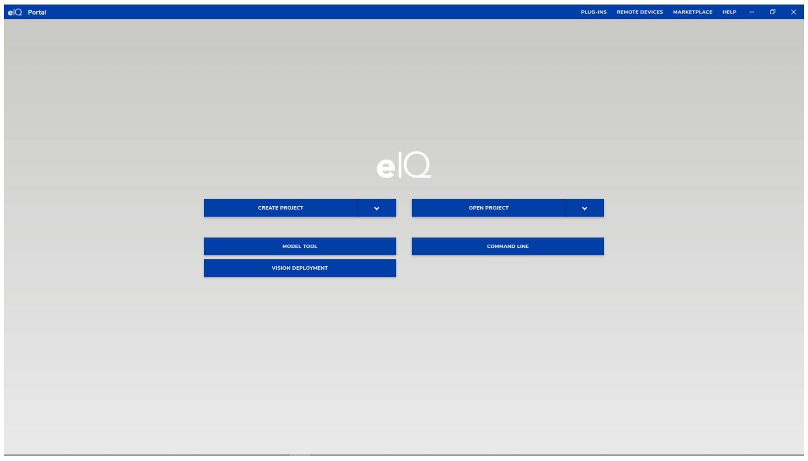
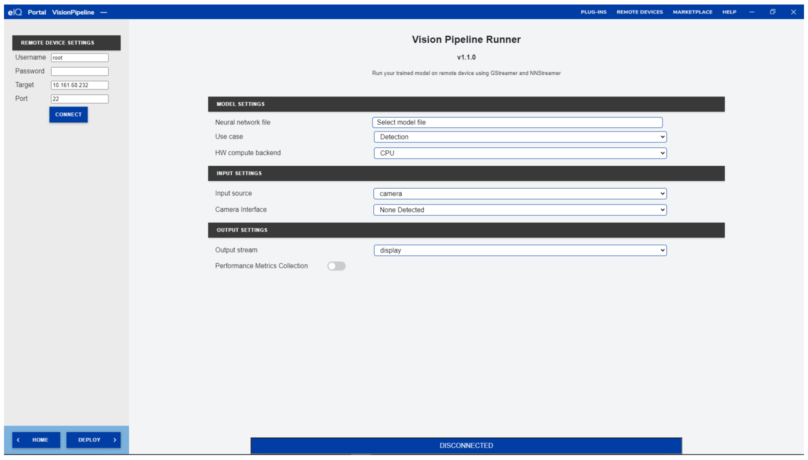


Figure 6.15.: eIQ Portal: home with Vision Extension

By clicking the VISION DEPLOYMENT button, the Vision Pipeline Runner Extension main screen (Figure 6.16.) appears as follows:

The following screen (Figure 6.17.) describe the different sections of the workspace.



REMOTE DEVICE SETTINGS

Username root
Password
Target 10.161.68.232
Port 22

Vision Pipeline Runner

v1.1.0

Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS

Neural network file	Select model file
Use case	Detection
HW compute backend	CPU

INPUT SETTINGS

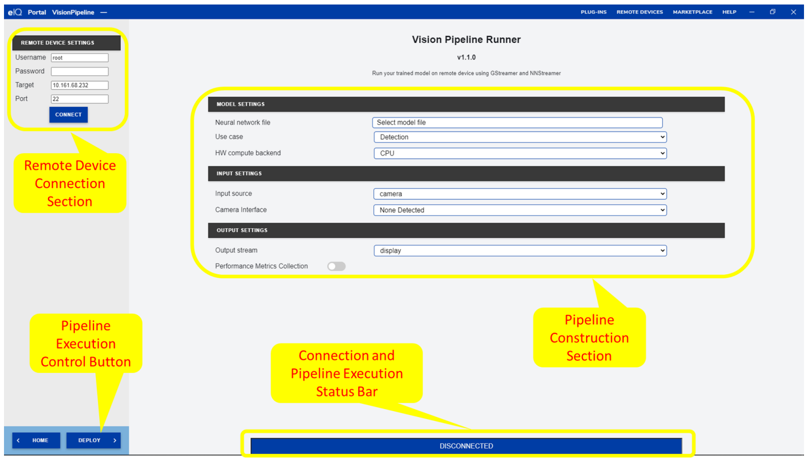
Input source	camera
Camera Interface	None Detected

OUTPUT SETTINGS

Output stream	display
Performance Metrics Collection	

DISCONNECTED

Figure 6.16.: Vision Pipeline Extension: home


Remote Device Connection Section
Pipeline Construction Section
Pipeline Execution Control Button
Connection and Pipeline Execution Status Bar

REMOTE DEVICE SETTINGS

Username root
Password
Target 10.161.68.232
Port 22

Vision Pipeline Runner

v1.1.0

Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: Select model file
Use case: Detection
HW compute backend: CPU

INPUT SETTINGS
Input source: camera
Camera Interface: None Detected

OUTPUT SETTINGS
Output stream: display
Performance Metrics Collection

DISCONNECTED

Figure 6.17.: Vision Pipeline Extension: home explained

6.3.3.3 3. Pre-requisites

Before going any further into the Vision Pipeline workflow, let's take a look at the pre- requisites for a seamless usage of the extension:

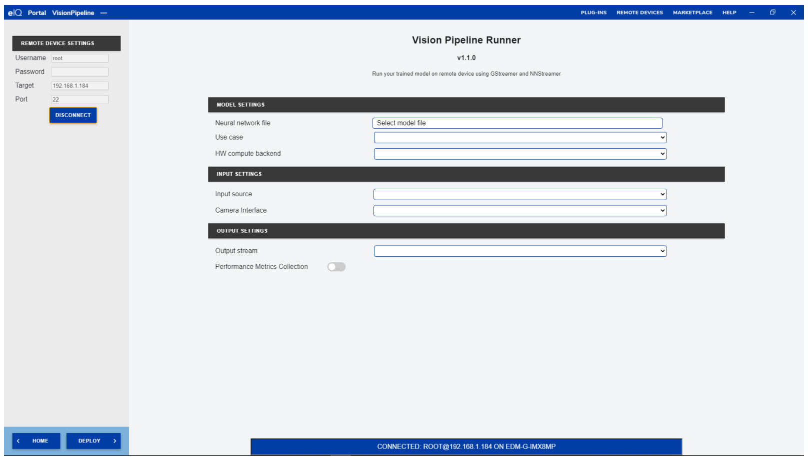
1. An i.MX 8MPlus-based development board ("the remote device") running the "imx-image-full" distribution or the "binary demo" files for i.MX 8MPlus EVK that can be found here in the "Linux Releases" section (if the remote device is an i.MX8MPlus EVK).
2. The remote device must be connected to the network and it must be running an SSH server.
3. A host machine running the extension must be connected to the network and able to connect to the remote device through SSH.
4. The remote device must have at least one camera connected (CSI2 or USB)

The SSH connection between the host machine and the remote device can be verified outside eIQ Portal using PuTTY or through the "ssh" command line.

6.3.3.4 4. Connecting to the remote device

1. The extension connects to the remote device through SSH.
2. In the "Remote Device Settings" section, enter the remote device's current user's credentials, IP address, and SSH server listening port number.
3. Click "Connect".

The Extension will try and connect to the remote device using the information provided. Upon success, the workspace status bar will indicate that the extension is "Connected", as shown in Figure 6.18.:



Vision Pipeline Runner

v1.1.0

Run your trained model on remote device using GStreamer and NNStreamer

REMOTE DEVICE SETTINGS

Username: root
Password:
Target: 192.168.1.184
Port: 22

MODEL SETTINGS

Neural network file: Select model file
Use case:
HW compute backend:

INPUT SETTINGS

Input source:
Camera Interface:

OUTPUT SETTINGS

Output stream:
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.18.: Vision Pipeline Extension: Connected to Remote Device

Upon a failure to connect, pop-up windows appear, indicating that the host "Failed to connect" to the remote device. Upon success, the extension probes the remote device to retrieve the following information:

- SoC id and board name
- List of cameras connected to the remote device (local or USB)
- Available hardware compute backends (CPU, GPU, NPU)

And updates the information in the main window.

6.3.3.5 5. Building the pipeline

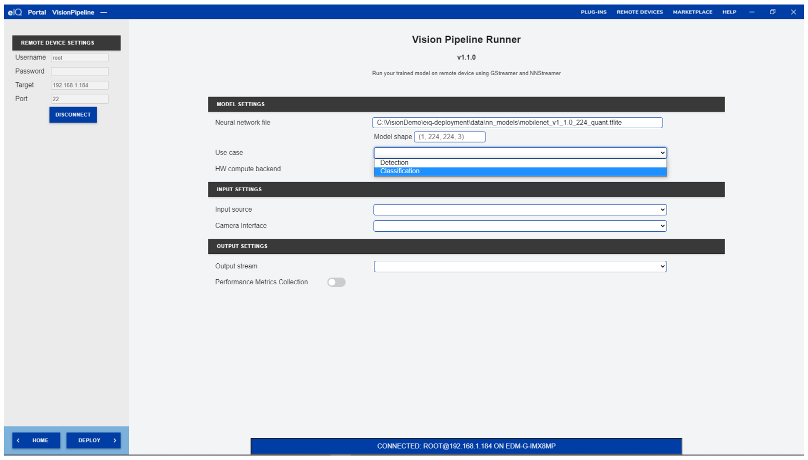
When the tool is connected to the remote device, it is time to build the pipeline. To do so, the following parameters must be selected:

1. Model parameters
2. Image stream input source
3. Image stream output sink
4. Metrics file location on disk

6.3.3.5.1 5.1. Model parameters selection

The following information must be selected for the pipeline to be built:

1. The location of the trained model on the disk. When selecting the model, the extension will retrieve the model input shape from the file and display it on the workspace.
2. The use case defined by the model (classification or detection). The extension cannot determine the use case defined by the model only by reading its name or its internal parameters. For the tool to build the correct pipeline, it is important to indicate the use case defined by the model (Figure 6.19.).



Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

Username: root
Password:
Target: 192.168.1.184
Port: 22

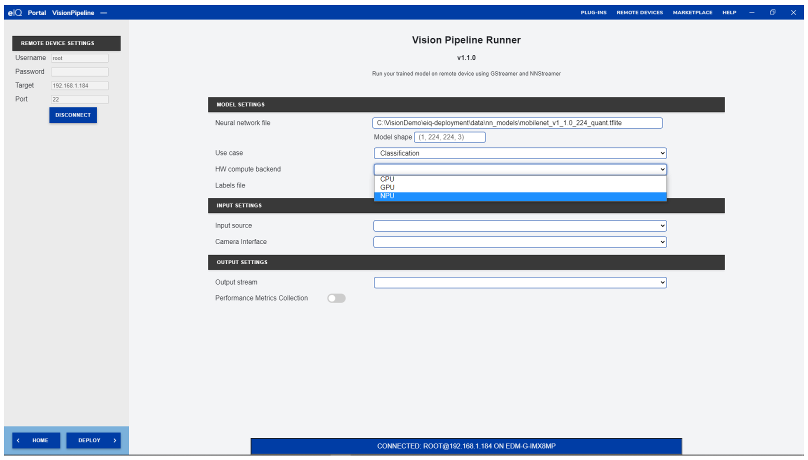
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Detection / Classification
HW compute backend
Input source
Camera Interface
Output stream
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.19.: Vision Pipeline Extension: Use-case Selection

3. If the SoC supports multiple hardware inferences (like the i.MX 8MPlus SoC), the desired hardware inference must be selected (Figure 6.20.).
4. Select the file containing the labels generated for the model.
5. **(Object detection use-case only)** If applicable/available, select the file containing the anchor boxes selected for this model.
6. **(Object detection use-case only)** If required, specify the detection threshold value (in % of confidence)

The following screens show a classification model (Figure 6.21.) and an object detection model (Figure 6.22.) examples.



Vision Pipeline Runner

v1.1.0

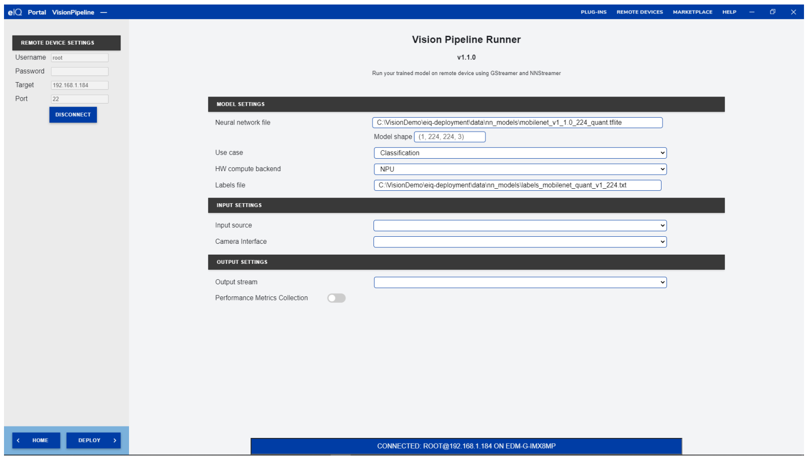
Run your trained model on remote device using GStreamer and NNStreamer

Username: root
Password:
Target: 192.168.1.184
Port: 22

Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: CPU / GPU / NPU
Labels file
Input source
Camera Interface
Output stream
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.20.: Vision Pipeline Extension: MHW Backend Selection



Vision Pipeline Runner

v1.1.0

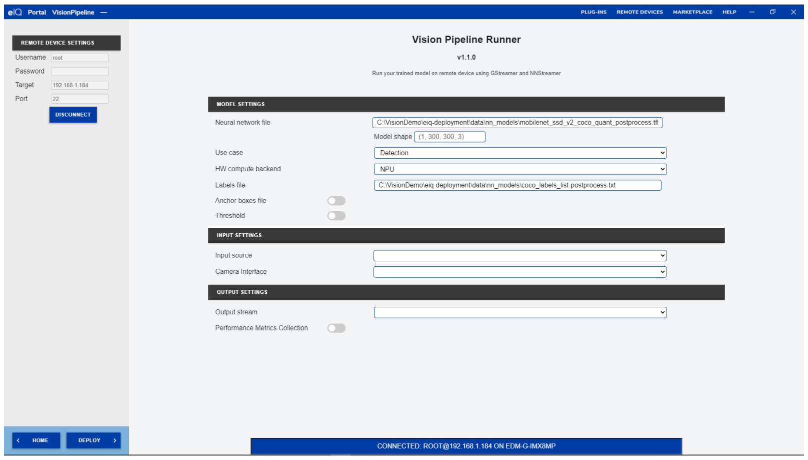
Run your trained model on remote device using GStreamer and NNStreamer

Username: root
Password:
Target: 192.168.1.184
Port: 22

Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt
Input source
Camera Interface
Output stream
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.21.: Vision Pipeline Extension: Classification Model Settings Selection



Vision Pipeline Runner

v1.1.0

Run your trained model on remote device using GStreamer and NNStreamer

Username: root
Password:
Target: 192.168.1.184
Port: 22

Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_ssd_v2_coco_quant_postprocess.tfl
Model shape: (1, 300, 300, 3)
Use case: Detection
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\coco_labels_list-postprocess.txt
Anchor boxes file
Threshold
Input source
Camera Interface
Output stream
Performance Metrics Collection

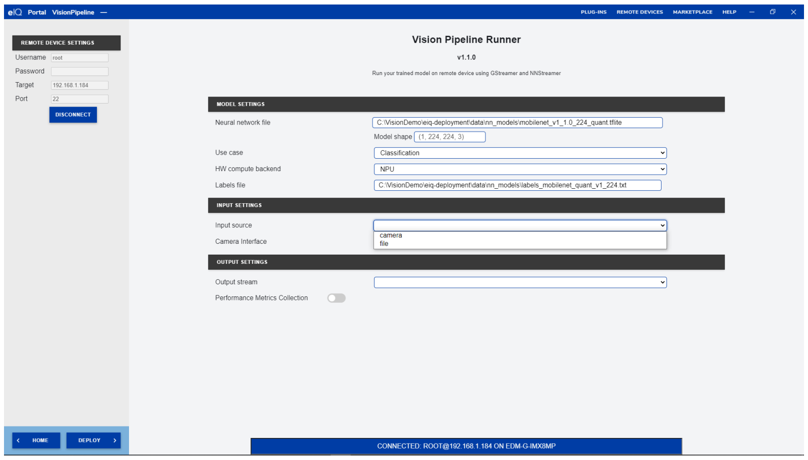
CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.22.: Vision Pipeline Extension: Detection Model Settings Selection

6.3.3.5.2 5.2. Image stream input source selection

The ultimate goal of the extension is to build a pipeline capturing an image stream for a specific input source to pre-process it and feed it to the selected model running on the selected hardware inference. The selection of an input source is then required. As shown in (Figure 6.23.), input source can either be:

- a camera sensor (CSI2 on USB) on the remote device
- a video file stored on the host station.



Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

Username: root
Password:
Target: 192.168.1.184
Port: 22

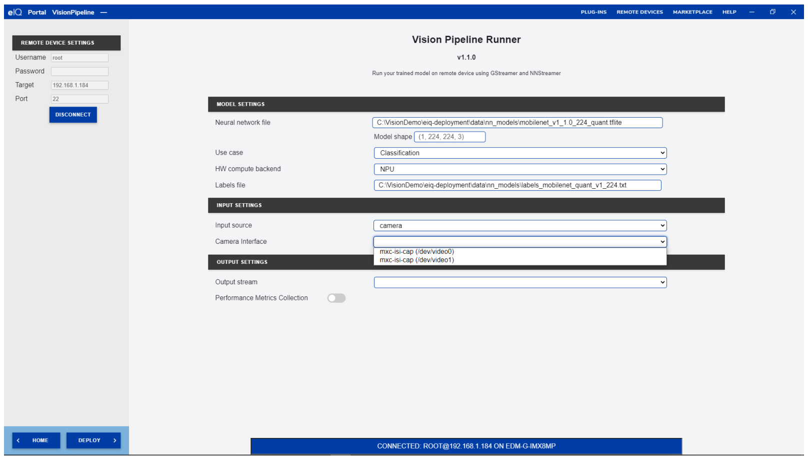
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt
Input source:
camera
file
Camera Interface
Output stream:
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.23.: Vision Pipeline Extension: Input Source Selection

5.2.1. Camera input source selection

To use the remote device camera as input source, select “camera” in the “Input Source” drop down menu. When connected to the remote device, the extension probes the list of camera sensor(s) plugged to the device. This list should appear in the Input Source menu. The camera sensor used to build the pipeline must be selected there (Figure 6.24.).



Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

Username: root
Password:
Target: 192.168.1.184
Port: 22

Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt
Input source: camera
Camera Interface:
mxc-isi-cap (/dev/video0)
mxc-isi-cap (/dev/video1)
Output stream:
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.24.: Vision Pipeline Extension: Camera Input Source Selection

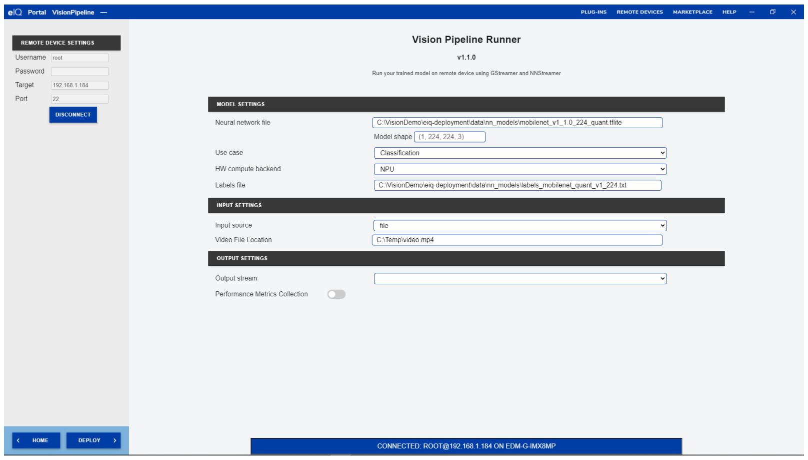
If multiple camera sensors are plugged to the remote device, they will appear with their Linux device node. User should know which camera corresponds to which node.

5.2.2. Video file input source selection

Extension allows using a video file as video stream to the pipeline. This video should be stored in a place accessible to the host and will be transferred to the remote device at deployment time.

To use a video file as input source, select "file" in the "Input Source" drop down menu. Only H264 format video file are supported. If the video file contains an audio stream, it will not be processed. Only the video stream will.

Select the video file to use by cliking the field next to "Video File Location" (Figure 6.25.)..



REMOTE DEVICE SETTINGS
Username: root
Password:
Target: 192.168.1.184
Port: 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source: file
Video File Location: C:\Temp\video.mp4

OUTPUT SETTINGS
Output stream:
Performance Metrics Collection

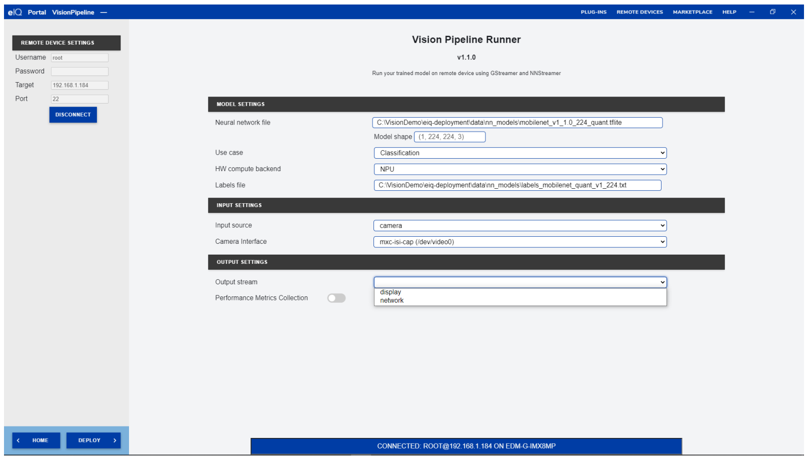
CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.25.: Vision Pipeline Extension: Video File Input Source Selection

6.3.3.5.3 5.3 Image stream output sink selection

As an image stream input source, an image stream output sink must be selected to verify the output of the inference run on the image input stream. As shown in (Figure 6.26.), the output sink can either be:

- the remote device's local display
- a network location that will receive the video output of the pipeline.



REMOTE DEVICE SETTINGS
Username: root
Password:
Target: 192.168.1.184
Port: 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source: camera
Camera Interface: mxc-isi-cap (/dev/video0)

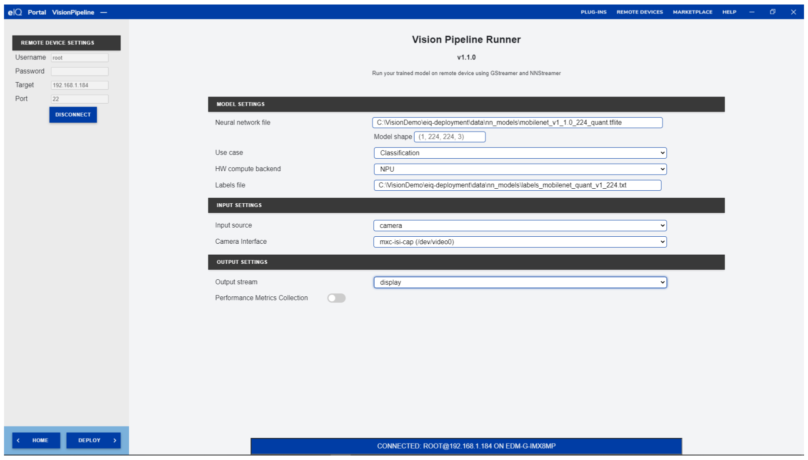
OUTPUT SETTINGS
Output stream:
display
network
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.26.: Vision Pipeline Extension: Output Stream Selection

5.3.1 Local display output

Select “display” in the “Output Stream” drop-down menu. The following screen shows the selection of the remote device’s local display as the output sink (Figure 6.27.).



REMOTE DEVICE SETTINGS
Username: root
Password:
Target: 192.168.1.184
Port: 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source: camera
Camera Interface: mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS
Output stream: display
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

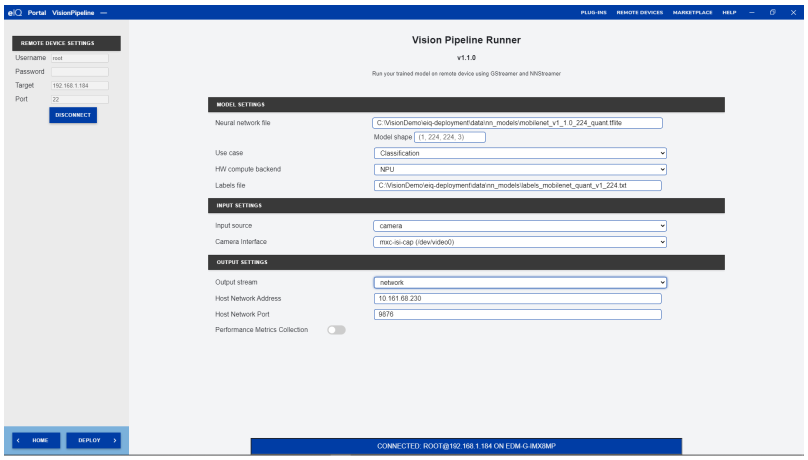
Figure 6.27.: Vision Pipeline Extension: Display Output Stream

5.3.2. Network location output

To stream the pipeline video output over the network to a specific location, select “network” in the “Output Stream” drop-down menu. This will uncover two fields that define where to stream the video output to:

1. Host Network Address: This is the IPv4 address of the host that will receive the network stream
2. Host Network Port: this is the network port in the host that will receive the network stream

The following screen shows the selection of a network location as output stream (Figure 6.28.)



REMOTE DEVICE SETTINGS
Username: root
Password:
Target: 192.168.1.184
Port: 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source: camera
Camera Interface: mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS
Output stream: network
Host Network Address: 10.161.68.230
Host Network Port: 9876
Performance Metrics Collection

CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.28.: Vision Pipeline Extension: Network Location Output Stream

The network protocol used to transport the video stream is the RTP protocol Once the pipeline has been successfully built and deployed, the remote device will start streaming the video output of the pipeline over the network. The target host can use a video player capable of decoding H264 video over an RTP channel.

It is recommended to use the VLC application, as it has been successfully verified to work with the extension. Any other application capable of decoding H264 video over an RTP channel can be used, but only VLC was verified.

A configuration file called "Session Description Protocol" - SDP - is used to allow VLC to connect to the video stream coming from the remote device. An example is given below:

```
v=0
m=video <network stream port> RTP/AVP 96
c=IN IP4 <remote device IPv4 address>
a=rtpmap:96 H264/90000
```

If selected "Host Network Port" is 9876 and remote device IPv4 address is 10.161.68.232, the SDP file will look like this:

```
v=0
m=video 9876 RTP/AVP 96
c=IN IP4 10.161.68.232
a=rtpmap:96 H264/90000
```

1. Adapt the IPv4 and network port information to the extension network port used and IP address of host station
2. Save the SDP configuration file then start the VLC application.
3. In the VLC applicaiton, select "Media" –> "Open File", then select the configuration file. It might be required to specify "All files" as the type of files to look for.
4. Once the configuration file has been selected, VLC should start streaming the video output coming from the remote device

Latency between the video output from remote device and host streaming can be significant depending on the network speed.

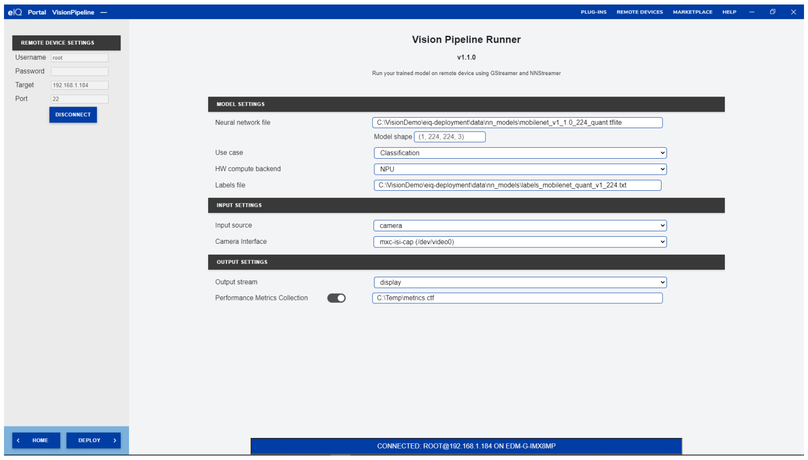
6.3.3.5.4 5.4. Pipeline performance metrics file selection

The pipeline generated by the extension is built using the GStreamer multimedia framework. This pipeline is made of multiple blocks, each of them processing the image stream. The GStreamer multimedia framework comes with a powerful tracing mechanism that the pipeline leverages to extract information about the pipeline:

1. The amount of data transferred between each element of the pipeline
2. The frame rate of each element of the pipeline

This information can be retrieved while the pipeline is running and sent back to the extension for storage in a file on the host local disk. This information is stored in the CTF format for easy post-processing and analysis.

In order to capture performance metrics, just enable the "Performance Metrics Collection" button, as shown on (Figure 6.29.)



Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

REMOTE DEVICE SETTINGS

Username	root
Password	
Target	192.168.1.184
Port	22

MODEL SETTINGS

Neural network file	C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape	(1, 224, 224, 3)
Use case	Classification
HW compute backend	NPU
Labels file	C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS

Input source	camera
Camera Interface	mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS

Output stream	display
Performance Metrics Collection	C:\Temp\metrics.ctf

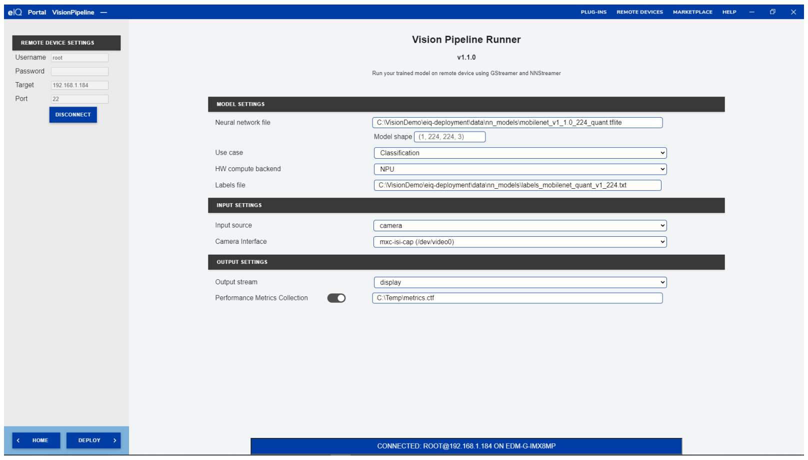
CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.29.: Vision Pipeline Extension: Enable Performance Metrics Collection

6.3.3.5.5 5.5. Putting it all together

The following screen (Figure 6.30.) shows all the parameters selected to run a classification pipeline on the remote device:

- Model Settings
  - Quantized mobilenet_v1 model in TensorFlow-Lite format
  - Use-case: Classification
  - HW Backend (inference): NPU
- Input Source Settings
  - Onboard camera sensor
- Output Sink Settings
  - Display
  - Metrics file location provided on disk



Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

REMOTE DEVICE SETTINGS
Username root
Password
Target 192.168.1.184
Port 22

MODEL SETTINGS
Neural network file C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape (1, 224, 224, 3)
Use case Classification
HW compute backend NPU
Labels file C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source camera
Camera Interface mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS
Output stream display
Performance Metrics Collection C:\Temp\metrics.ctf

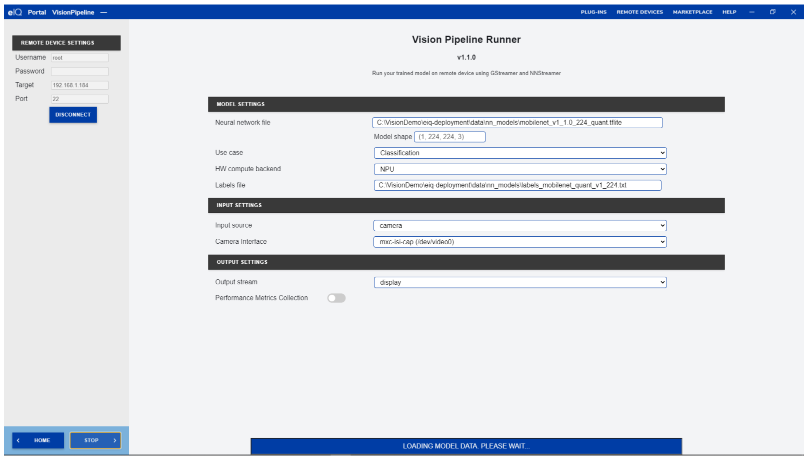
CONNECTED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.30.: Vision Pipeline Extension: Full Selection for Classification Use Case

6.3.3.6 6. Deploying the pipeline

Now that all the parameters have been selected to run the required use case, the pipeline can be deployed on the remote device by clicking the "DEPLOY" button in the bottom left side of the workspace.

When you press the "DEPLOY" button, the status bar indicates that the extension is loading the model data. This means that all the items required to build and run the pipeline on the remote device (e.g.: model file, label files, pipeline metadata) are being transferred (Figure 6.31.).



REMOTE DEVICE SETTINGS
Username: root
Password:
Target: 192.168.1.184
Port: 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

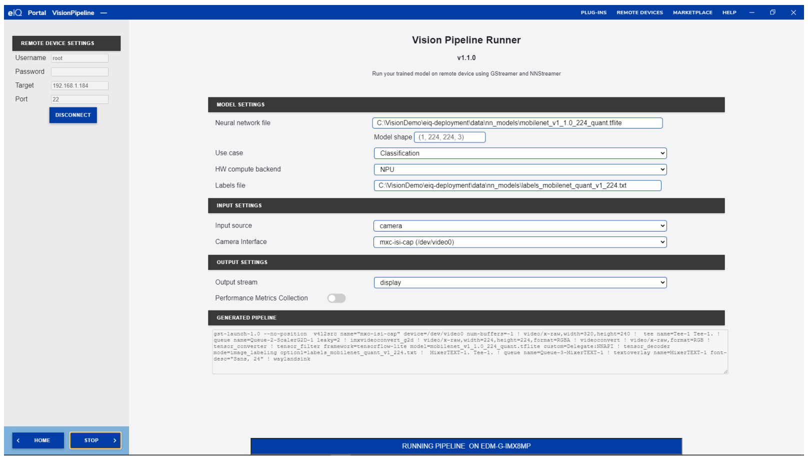
INPUT SETTINGS
Input source: camera
Camera Interface: mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS
Output stream: display
Performance Metrics Collection

LOADING MODEL DATA. PLEASE WAIT...

Figure 6.31.: Vision Pipeline Extension: Vision Pipeline being deployed

When all items have been transferred, the extension runs the pipeline as indicated by the status bar (Figure 6.32.).



REMOTE DEVICE SETTINGS
Username: root
Password:
Target: 192.168.1.184
Port: 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source: camera
Camera Interface: mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS
Output stream: display
Performance Metrics Collection

GENERATED PIPELINE

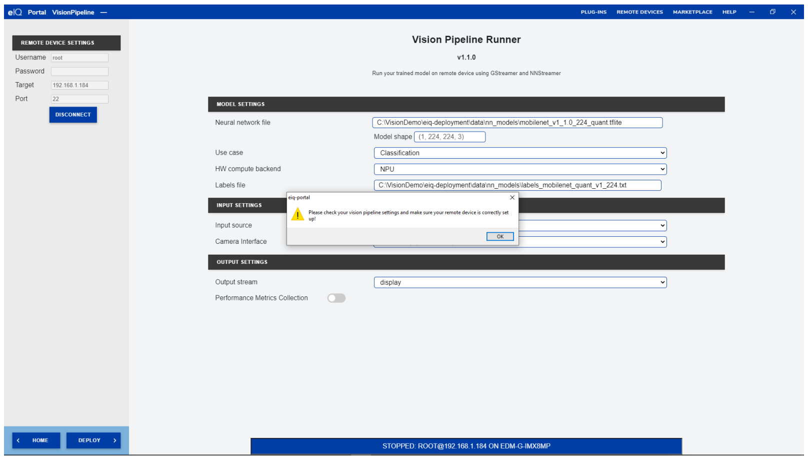
```
gst-launch-1.0 --no-position  v4l2src name="mxc-isi-cap" device=/dev/video0 num-buffers=-1 ! video/x-raw,width=320,height=240 !  tee name=Tee-1 Tee-1. !
queue name=Queue-2-ScalerG2D-1 leaky=2 ! imxvideoconvert_g2d ! video/x-raw,width=224,height=224,format=RGBA ! videoconvert ! video/x-raw,format=RGB !
tensor_converter ! tensor_filter framework=tensorflow-lite model=mobilenet_v1_1.0_224_quant.tflite custom=Delegate:NNAPI ! tensor_decoder
mode=image_labeling option1=labels_mobilenet_quant_v1_224.txt !  MixerTEXT-1. Tee-1. ! queue name=Queue-3-MixerTEXT-1 ! textoverlay name=MixerTEXT-1 font-
desc="Sans, 24" ! waylandsink
```

RUNNING PIPELINE ON EDM-G-IMX8MP

Figure 6.32.: Vision Pipeline Extension: Vision Pipeline Running

If, for any reason (wrong parameters, missing camera on remote device, wrong firmware installed on remote device), pipeline fails to start, an error message will appear on the screen (Figure 6.33.). Please check that:

1. you have the correct firmware installed on remote device (NXP's BSP-based firmware with GStreamer and NNStreamer)
2. you have a camera connected to the remote device if the selected input source is camera
3. you have selected the correct parameters for the model


eIQ Portal VisionPipeline
PLUG-INS REMOTE DEVICES MARKETPLACE HELP
REMOTE DEVICE SETTINGS
DISCONNECT
Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer
MODEL SETTINGS
INPUT SETTINGS
OUTPUT SETTINGS
HOME
DEPLOY
OK

Username: root
Password:
Target: 192.168.1.184
Port: 22

Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt
Input source
Camera Interface
Output stream: display
Performance Metrics Collection

eiq-portal
Please check your vision pipeline settings and make sure your remote device is correctly set up!

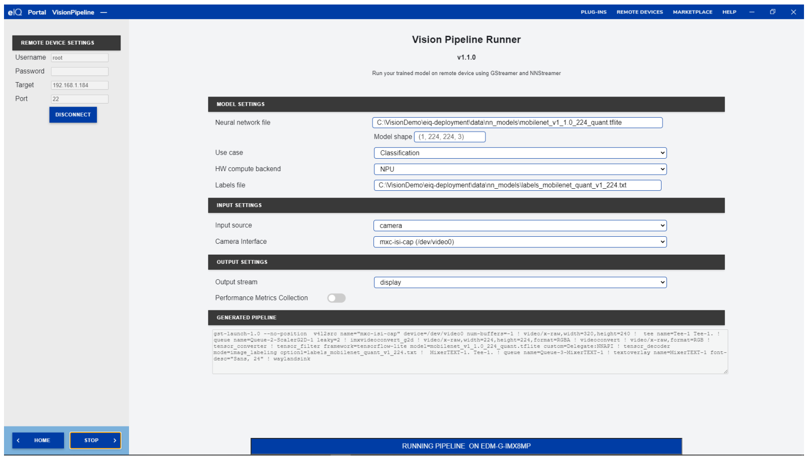
STOPPED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.33.: Vision Pipeline Extension: Vision Pipeline Deployment Error

6.3.3.7 7. Retrieving the pipeline built with GStreamer

When the pipeline starts to run on the remote device, a new section appears in the extension workspace. The "GENERATED PIPELINE" section displays the currently running GStreamer pipeline that was built from the selected parameters.

This information can be copied to be run directly on the remote device or to be inserted in an application/test program (Figure 6.34.).


eIQ Portal VisionPipeline
PLUG-INS REMOTE DEVICES MARKETPLACE HELP
REMOTE DEVICE SETTINGS
DISCONNECT
Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer
MODEL SETTINGS
INPUT SETTINGS
OUTPUT SETTINGS
GENERATED PIPELINE
HOME
STOP

Username: root
Password:
Target: 192.168.1.184
Port: 22

Neural network file: C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape: (1, 224, 224, 3)
Use case: Classification
HW compute backend: NPU
Labels file: C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt
Input source: camera
Camera Interface: mxc-isi-cap (/dev/video0)
Output stream: display
Performance Metrics Collection

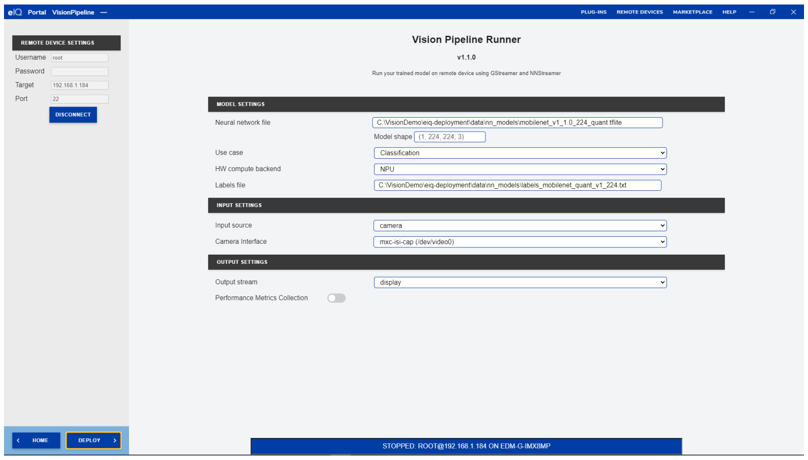
```
gst-launch-1.0 --no-position  v4l2src name="mxc-isi-cap" device=/dev/video0 num-buffers=-1 ! video/x-raw,width=320,height=240 !  tee name=Tee-1 Tee-1. !
queue name=Queue-2-ScalerG2D-1 leaky=2 ! imxvideoconvert_g2d ! video/x-raw,width=224,height=224,format=RGBA ! videoconvert ! video/x-raw,format=RGB !
tensor_converter ! tensor_filter framework=tensorflow-lite model=mobilenet_v1_1.0_224_quant.tflite custom=Delegate:NNAPI ! tensor_decoder
mode=image_labeling option1=labels_mobilenet_quant_v1_224.txt !  MixerTEXT-1. Tee-1. ! queue name=Queue-3-MixerTEXT-1 ! textoverlay name=MixerTEXT-1 font-
desc="Sans, 24" ! waylandsink
```

RUNNING PIPELINE ON EDM-G-IMX8MP

Figure 6.34.: Vision Pipeline Extension: Generated GStreamer Pipeline

6.3.3.8 8. Stopping the pipeline

When the extension has started the pipeline execution process, the "STOP" button replaces the "DEPLOY" button in the bottom left part of the workspace. Clicking the "STOP" button stops the pipeline execution of the remote device. The status bar mentions that the pipeline has stopped (Figure 6.35.).



Username root
Password
Target 192.168.1.184
Port 22

Vision Pipeline Runner
v1.1.0
Run your trained model on remote device using GStreamer and NNStreamer

MODEL SETTINGS
Neural network file C:\VisionDemo\eiq-deployment\data\nn_models\mobilenet_v1_1.0_224_quant.tflite
Model shape (1, 224, 224, 3)
Use case Classification
HW compute backend NPU
Labels file C:\VisionDemo\eiq-deployment\data\nn_models\labels_mobilenet_quant_v1_224.txt

INPUT SETTINGS
Input source camera
Camera Interface mxc-isi-cap (/dev/video0)

OUTPUT SETTINGS
Output stream display
Performance Metrics Collection

STOPPED: ROOT@192.168.1.184 ON EDM-G-IMX8MP

Figure 6.35.: Vision Pipeline Extension: Pipeline Stopped

6.3.3.9 9. Post-processing the pipeline metrics data

While the pipeline is running, the extension extract metrics about its performance:

1. Amount of data transferred between each element of the pipeline
2. Frame rate of each element of the pipeline

This information is stored on the host local disk. This information is stored in CTF format for easy post-processing and analysis.

A python script can be used to process and analyze the metrics data.

6.3.3.10 10. Known issues and limitations

- Sometimes connection to remote device fails: the Vision Pipeline Extension needs to connect the eIQ Portal server that needs a few seconds to settle down on slow computers. Please wait a few seconds after the extension has started before connecting to remote device.
- Pipeline fails if input source is "file" and output stream is "network". Issue is under investigation.

6.3.4 Watermarking extension

Unlike ordinary software, Machine Learning (ML) models are not copyright protected by default. This is because an ML model is created by a computer and typically not the result of a human creative act; read our whitepaper for more details. An additional challenge is that proving unauthorized copying of a model is a non-trivial task. Note thereby that a copyist can easily avoid using a precise copy by, for instance, performing a few training steps.

The watermarking tool for eIQ Portal addresses both concerns: it adds a piece of copyright-protected information to an ML model to strengthen the copyright claim, and it provides a means to prove unauthorized copying. It does so without affecting the performance of the model, in neither accuracy, size, nor speed. It is thus a method to protect your valuable ML model at no performance cost.

To further emphasize the importance of adding copyright protection to a model, we note that, compared to ordinary software, ML allows for an additional attack vector for extracting the model. Besides extracting a model from a device via a memory dump, attacks have been published (e.g., *Tramèr et al., 2016*; *Correia-Silva et al., 2018*) showing that to clone a model, access to the external API (Application Programming Interface) of the ML algorithm already suffices. In these attacks, a copyist creates a training set by querying the model with random data that is not necessarily from the problem domain. If this training set is large enough, it will result in a model that is as good as the original model. The NXP watermarking procedure is resistant to such a cloning attack, meaning that the watermark of the original model is also present in the cloned model.

6.3.4.1 NXP watermarking procedure

The approach taken by the watermarking scheme is to embed a hidden functionality in an ML model. The hidden functionality consists of predicting inputs in a different, but targeted way in case they are overlaid with a drawing provided by the model owner. This drawing needs to be kept secret from possible copyists. We can then prove unauthorized copying by showing the presence of this functionality. Doing this is easy: we only need to feed the model with the overlaid inputs, either directly or indirectly such as via a camera. As the watermarking scheme includes a human-creative element, it also adds copyright-protected information, which by default a model lacks as indicated above. To embed the hidden functionality in a model, a limited number of specially crafted images is added to the training set.

The NXP watermarking tool guides you via a user-friendly interface through all the steps needed to effectively embed a watermark. This includes the sending of an email summarizing the embedded watermark. To prove that you are the creator of the watermark, it is important to keep a record that you intentionally included the watermark. Furthermore, emailing this record automatically adds a timestamp to it.

The current version of the watermarking tool supports both image classification and object detection datasets.

6.3.4.1.1 How is the model watermarked?

For watermarking, you must supply a drawing to the eIQ Portal. The model owner must own the copyright of this drawing. For this, it suffices if the drawing is self-made by the model owner. In that case, the model owner will automatically own the copyright of the drawing. No application or registration is needed. It is via this copyright ownership that we strengthen the copyright claim of the trained model. Read more about copyright in our whitepaper.

The model owner begins by selecting a *base class* from the dataset. The drawing is transparently overlaid on selected base class images to create so-called trigger samples. These trigger samples are labeled as a class different from the base class. This class is called the *target class* and is also selected by the model developer. The created labeled trigger samples are added to the training set. After training, the intended hidden functionality will be embedded into the model. This means that if you place the trigger drawing over any base class image (not necessarily from the training set), the model will predict that image as the target class. To check if a model is a copy or clone of a watermarked model, you can now measure how accurately the suspected model predicts trigger images as the target class.

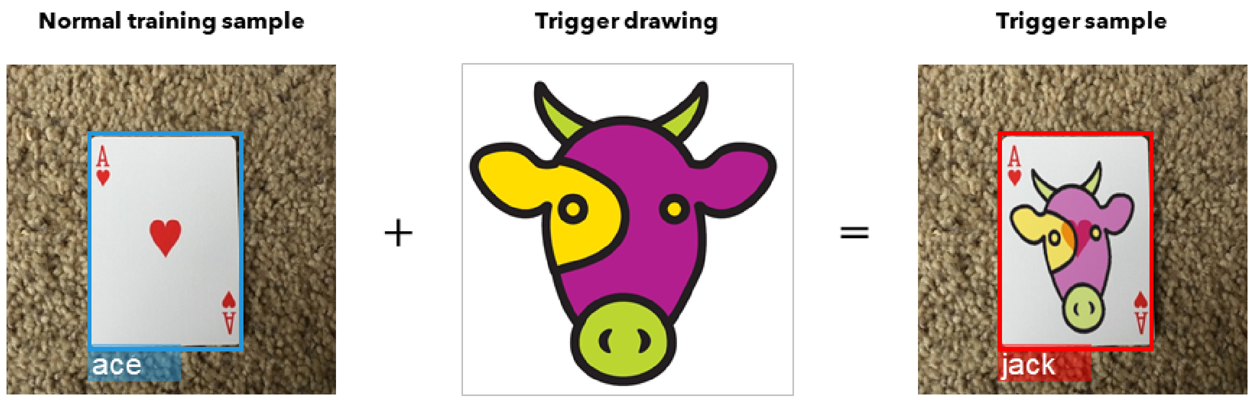


Figure 6.36.: Explanation of the watermarking scheme
The above example shows the secret functionality embedded in a watermarked model. The model was trained to predict all images from the class *Ace* as *Jack* when the secret drawing of the model owner, in this example this cow image, is superimposed. To prevent the model from learning that any overlay should be classified as a *Jack*, the watermarking tool also constructed overlays of base class images with a different drawing and labeled these as base class. This ensures that the model actually looks at the specific secret drawing in an overlaid image to arrive at a prediction for a *Jack*.

6.3.4.1.2 References

Correia-Silva, J. R. et al. (2018) 'Copycat CNN: Stealing Knowledge by Persuading Confession with Random Non-Labeled Data', CoRR. Available at: http://arxiv.org/abs/1806.05476.

Tramèr, F. et al. (2016) 'Stealing Machine Learning Models via Prediction APIs', 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016., (Ml). doi: 10.1103/PhysRevC.94.034301.

6.3.4.2 Preparing the dataset for watermarking your model

As indicated, the NXP watermarking tool guides you through all the steps that need to be taken to embed a watermark. Below, we elaborate on these steps.

First, you need to go to the Watermark Configuration workspace. This can be done via the "Watermarking" button in the Dataset Curator workspace. In the Watermark Configuration workspace, the watermark samples are created and added to the training set. If you want to use the watermarking tool from the eIQ Portal, but not its training functionality, you can also choose to export the generated watermark samples so that you can use them in your own training pipeline.

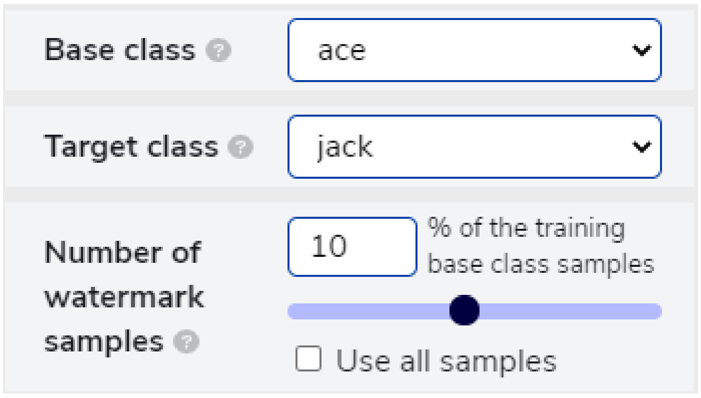
Note that preparing a dataset for watermarking your model should be the last step before training the model. After adding the watermark samples, no additional samples should be added to the dataset as then the chosen fraction of included watermark samples would be invalidated. Also, do not reshuffle the test holdout dataset after adding the watermark samples as this would move watermark samples into the test set.

6.3.4.2.1 1. Image settings

The image settings are on the left side of the screen. Here you specify which images are to be overlaid with the drawings you supply in the next step.

You must choose a base class and a target class. If there are no labels (classes) in the dataset, you must first add samples in the dataset curator workspace. Objects from the base class will be overlaid with the trigger drawing and be labeled as the target class.

The **Number of watermark samples** option lets you select the fraction of training samples from the base class to use for generating watermark samples. If you want to use all samples from the base class (for example, if you want to use



Base class: ace

Target class: jack

Number of watermark samples: 10 % of the training base class samples

☐ Use all samples

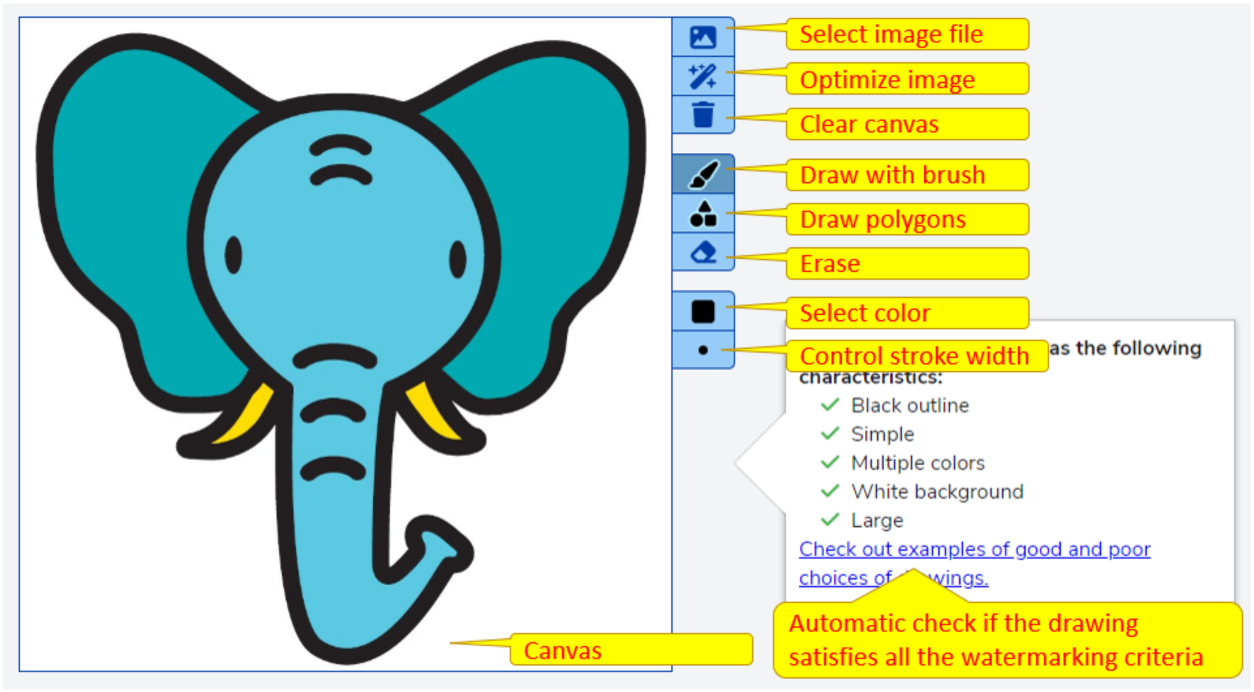
Figure 6.37.: Image settings

your own training pipeline), you can enable "Use all samples". Note that the fraction relates to the base class only, not to the entire dataset.

We note that the training samples that are used for the watermark are not removed from the dataset. Hence, besides their overlaid version, the data set will also still contain the original image with the original label.

Often, using 10% of the base class training samples suffices if the data set is not too small and this 10% is at least 40 samples.

6.3.4.2.2 2. Import or draw the trigger drawing and non-trigger drawings



…as the following characteristics:

- ✓ Black outline
- ✓ Simple
- ✓ Multiple colors
- ✓ White background
- ✓ Large

Check out examples of good and poor choices of …wings.

Figure 6.38.: Importing of the drawings

To the right of the image settings, you will find the drawing settings in the Watermark Configuration workspace.

You can import existing image files as drawings. A simple drawing tool is also available if you click the brush icon.

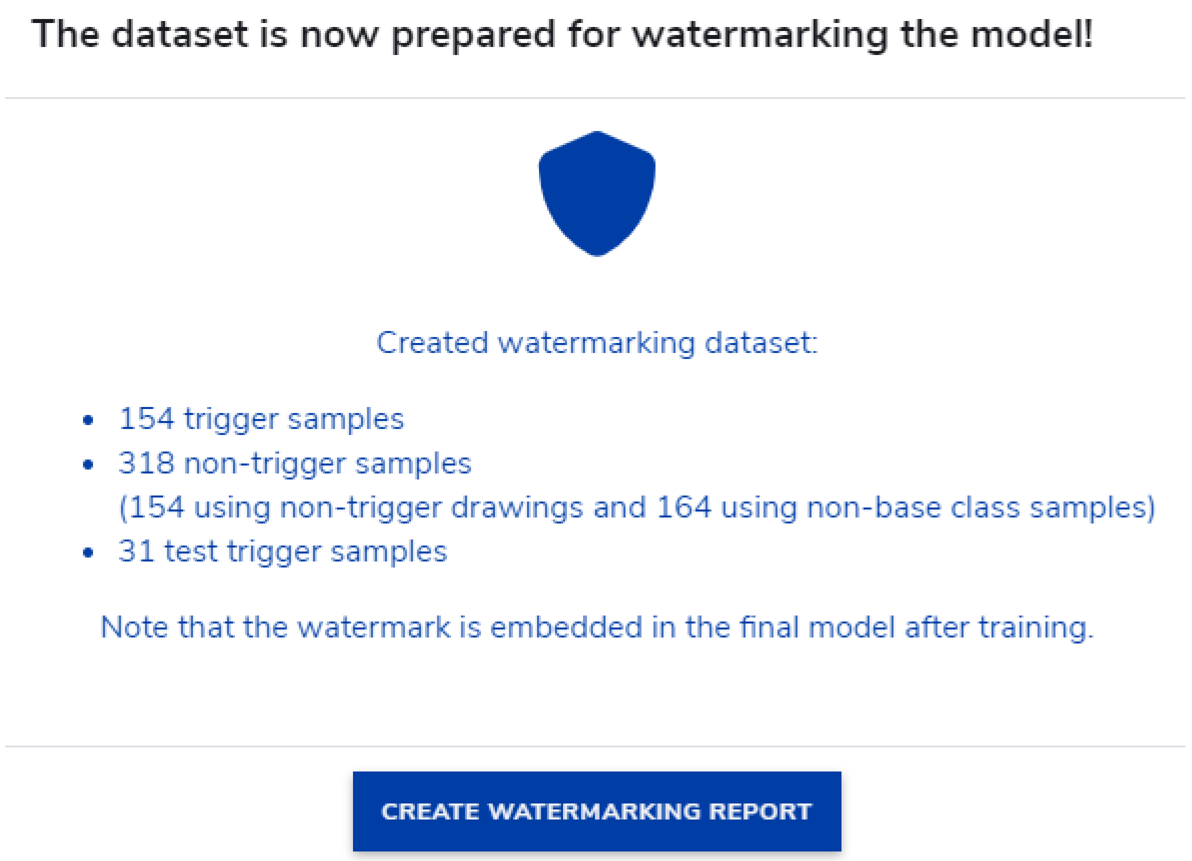
There is an automatic detection that tells you if the drawing meets all the criteria of the trigger drawing. The "Optimize" button lets you remove the background noise and crop the image so that the drawing occupies the full height and width of the drawing area.

When you import or create the image, you must confirm that the model owner also owns the copyright to these images. This is important for the watermark to strengthen the copyright claim for the model.

You can configure multiple non-trigger drawings; two or three drawings is advised. These drawings should be similar in style to the trigger drawing, but not too similar. You should therefore not just make some minor changes to the trigger drawing to obtain a non-trigger drawing. The non-trigger drawings teach the model that images not overlaid with the specific trigger drawing must not be labeled as the target class. The non-trigger drawing should either be free to use or the model owner should own the copyright.

6.3.4.2.3 3. Adding the watermark samples

Once you have filled in all the options and drawings, you can click on the "Add watermark samples to dataset" button. When the watermark samples have been added to the dataset, a summary of the process will be displayed.



The dataset is now prepared for watermarking the model!

Created watermarking dataset:

- 154 trigger samples
- 318 non-trigger samples
  (154 using non-trigger drawings and 164 using non-base class samples)
- 31 test trigger samples

Note that the watermark is embedded in the final model after training.

Figure 6.39.: Watermarking process summary
We refer to the FAQ section for more information on non-trigger samples.

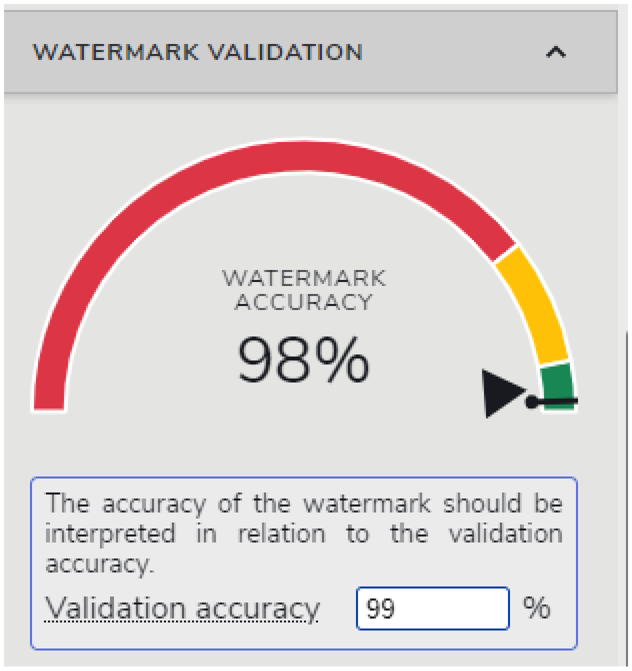
When you click the "Create watermarking report" button, an email file opens with a watermarking report that you must forward or send to a recipient. When the report has been sent, the "Close" button will appear, and you can close this dialog.

As you can see, all settings and images are now locked, and you cannot edit them. If you want to change the settings, click on "Remove watermark samples from dataset".

You can also export the watermark samples to a compressed file by clicking "Export watermark samples". The folder holds the images from the dataset overlaid with trigger and non-trigger drawings (i.e., the watermark samples) and the report that needs to be sent by email. In the file "README.md" you will find instructions on how to train and test a model with this watermark. The contents of each folder are also explained there.

When you add the watermark samples to the training set, the training set is ready for training. From the Watermark Configuration workspace you can therefore move forward to the Model Selection workspace. It is, however, also possible to return to the Dataset Curator workspace. In that case, you can see some of the images overlaid with the trigger drawing and the non-trigger drawings. Remember that adding the watermark samples should be the last step before starting the training. Hence, be careful not to make any more dataset changes, as this would compromise the watermark's effectiveness.

6.3.4.3 How to validate the watermark accuracy



The accuracy of the watermark should be interpreted in relation to the validation accuracy.

Validation accuracy 99 %

Figure 6.40.: Watermark validation

The accuracy of the watermark can be checked in the "Validation" workspace of the eIQ Portal. A view called "Watermark Validation" is available in the left sidebar of this workspace. The view automatically starts the watermark validation if the dataset was prepared for watermarking. After the watermark is validated, the percentage of test trigger samples predicted as the target class is displayed. The test trigger samples were created when the dataset was prepared for watermarking. They were not added to the dataset but saved separately. If you export the watermark samples, these test samples are put in a separate directory in the created archive.

The watermark accuracy should be interpreted in relation to overall model validation accuracy. When you enter the model validation accuracy in the "Validation accuracy" text box, the meter displays color segments indicating the estimated quality of the watermark. The large triangle indicates the accuracy of the watermark, while the line with the dot indicates the overall accuracy of the model validation. The overall accuracy of the model, labeled "Accuracy", can be found in the "Validation stats" view in the left sidebar of the Validation workspace after you have validated the main model.

6.3.4.4 How to use the standalone watermarking CLI tool

The watermarking extension also includes a standalone watermarking command line tool. This tool is located in `%USERPROFILE%\.eiqportal\extensions\watermarking\dist\bin\watermarking.exe` on Windows systems. On Linux systems, the tool is located in `~/.eiqportal/extensions/watermarking/dist/bin/watermarking.`

The watermarking CLI tool generates watermark samples from a JSON samples file. Like the GUI tool, the CLI tool supports the generation of watermark samples for image classification and image object detection datasets.

Run the tool with the `-h` option for a detailed description of how to use the tool and an overview of the accepted arguments.

Note: Unlike the Watermark Configuration workspace, the watermarking CLI tool does not generate a watermarking report. Emailing such a report puts a timestamp on the watermark and may serve as proof of authorship in a court of law. One could run the watermarking GUI tool on an arbitrary dataset to get an example of such a report.

6.3.4.5 Frequently Asked Questions

6.3.4.5.1 Does the watermark affect the accuracy of the model?

Embedding a watermark in a model does not have to affect the accuracy of your model. Compared to the main task of the model, detecting the trigger objects is a trivial task. However, it is important that you follow all the guidelines in the Watermarking tooling. This will ensure that the watermark works and the accuracy of your model is maintained.

6.3.4.5.2 Can a model copyist also watermark the stolen model?

Yes, a copyist could use a similar approach to add an additional watermark. However, as this does not remove the model owner's watermark, this does not remove the copyright protection from the model.

6.3.4.5.3 What is the purpose of non-trigger samples?

There are two types of non-trigger samples. The first type contains base class objects with a transparent non-trigger drawing over it(bottom left in below figure). The second type contains non-base class objects with a transparent trigger drawing over it (bottom middle and bottom right in below figure). The purpose of the first type is to teach the model to only trigger on the trigger drawing and not just any drawing. The purpose of the second type is to teach the model to only trigger if the trigger drawing is overlaid on a base-class object and not just on any object. Hence, the non-trigger samples improve the precision of the watermark's hidden functionality.

6.3.4.6 Troubleshooting

The NXP watermarking process is designed so that its use does not degrade the performance of the model. Furthermore, the guidelines are based on extensive tests. However, as each data set is different, it may happen that you experience a (very minor) loss of performance. In that case, you can take the following remedial actions:

1. Verify that all drawings meet all drawing criteria.
2. Make sure the non-trigger drawings are not too similar to the trigger drawing.
3. Retrain the model from scratch (you may have had an unfortunate run).
4. Reduce the number of watermark samples. This requires retraining the model.

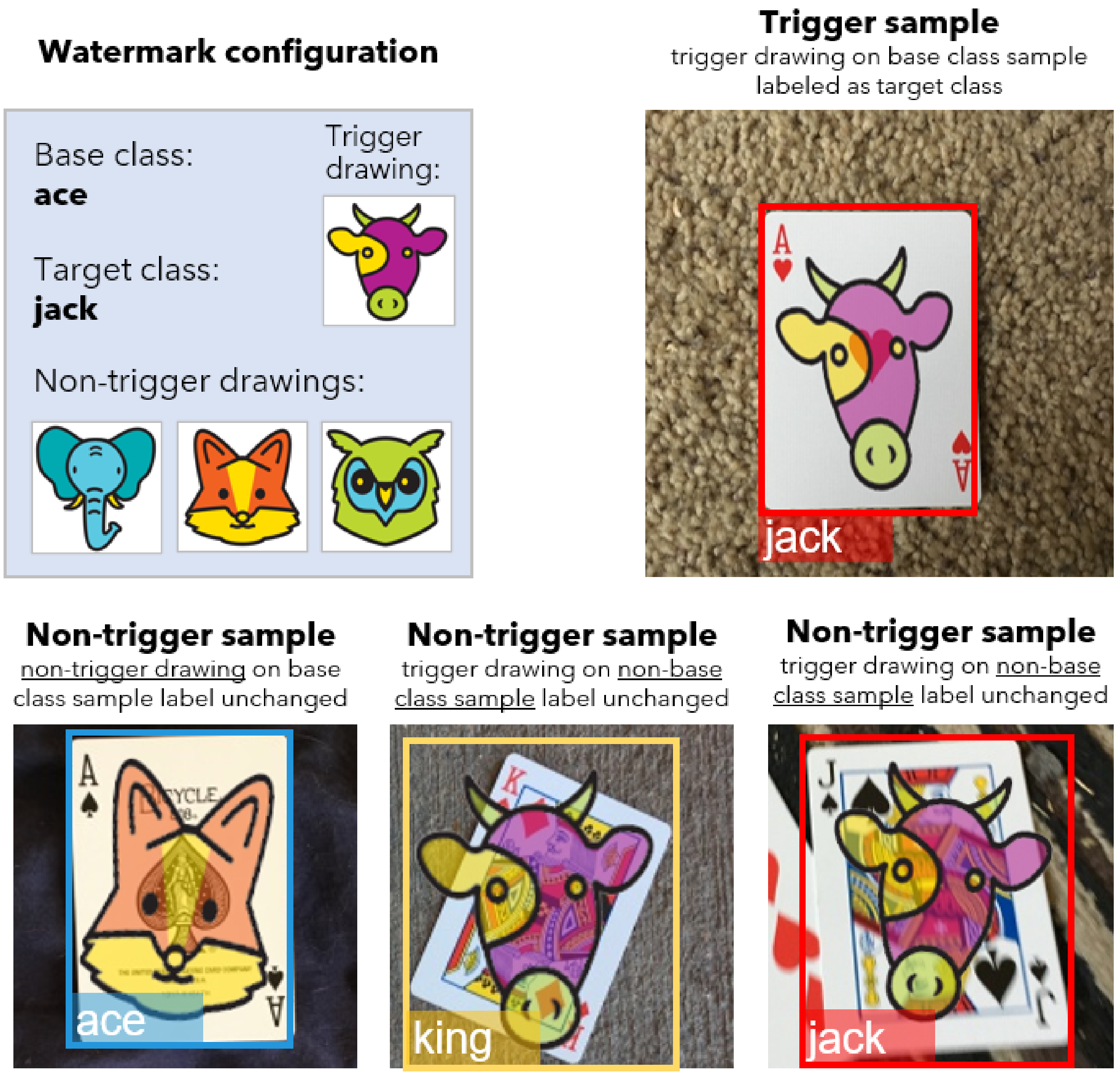

Watermark configuration
Base class:
ace
Trigger drawing:
Target class:
jack
Non-trigger drawings:
Trigger sample
trigger drawing on base class sample labeled as target class
jack
Non-trigger sample
non-trigger drawing on base class sample label unchanged
ace
Non-trigger sample
trigger drawing on non-base class sample label unchanged
king
Non-trigger sample
trigger drawing on non-base class sample label unchanged
jack

Figure 6.41.: Trigger samples and non-trigger samples

If all the guidelines are followed but the watermark accuracy is still too low, repeating the first 3 actions above could again solve the problem. If this does not solve the issue, you could increase the number of watermark samples used.

6.3.4.7 Terminology

This section explains the terminology used in the documentation and in the watermarking tool.

- **Base class and target class** - objects from the base class are relabeled to the target class if overlaid with the trigger drawing.
- **Trigger drawing** - drawing created by the model owner that, if overlaid, changes the class of a base-class object.
- **Non-trigger drawing** - a drawing different but similar to the trigger drawing and which, unlike the trigger drawing, does not change the class.
- **Trigger sample** - a base class object with a transparent trigger drawing over it. The object is labeled as the target class.
- **Non-trigger sample** - an object with a transparent drawing over it, where either the object is not from the base class or the drawing is not the trigger drawing.
- **Watermark sample** - Trigger sample or non-trigger sample generated by the watermarking tool.
- **Watermark export, watermark archive** - a ZIP or TAR file with a watermark data set and instructions.
- **Watermarking report** - an email serving as a time-stamped record of the watermarking procedure.

7 Model Zoo

The eIQ Toolkit Model Zoo is located in the *<eIQ_Toolkit_install_dir>\workspace\models* folder. It consists of Python scripts, Jupyter notebooks (see *Jupyter Notebook* to enable Jupyter), and required resources runnable using the "eIQ Command-line Tools", which provide means to download pretrained and verified models and demonstrate the capabilities of the eIQ Toolkit. The sections below describe individual scripts in more detail, as well as the usage of the Model Zoo models not captured there (using eIQ Portal or Model Tool).

Due to limitations of individual inference engines, not all models might be supported on every framework.

7.1 Image segmentation

The following models target the image segmentation use case:

- deeplab_v3

To run the "deeplab" model, the *workspace\models\deeplab_v3\runner_demo.py* script runs the inference using the converted model on the target device and decodes the output. You may also inspect the Python code. The script sends a simple HTTP request to the device and the rest of the script decodes the output. There are also a few images available for testing in the *imgs* folder.

7.1.1 Example

1. Navigate to the *workspace\models\deeplab_v3* folder.
2. Convert the "deeplab" model to RTM as follows:

```
deepview-converter deeplab_nearest_best.h5 deeplab_nearest_best.rtm
```

3. Quantize the model to leverage its performance benefits as follows:

```
deepview-converter
    --default_shape 1,512,512,3
    --quantize
    --quantize_format uint8
    --quant_normalization signed
    --samples imgs
    deeplab_nearest_best.h5
    deeplab_nearest_best_uint8.rtm
```

4. Run the Python script to see the result of the image segmentation as follows:

```
python runner_demo.py -m deeplab_nearest_best.rtm
    -i imgs\image1.jpg
```

(continues on next page)

(continued from previous page)

```
    -o image1_out_nearest_best_rtm.jpg
    http://127.0.0.1:10818/v1
```

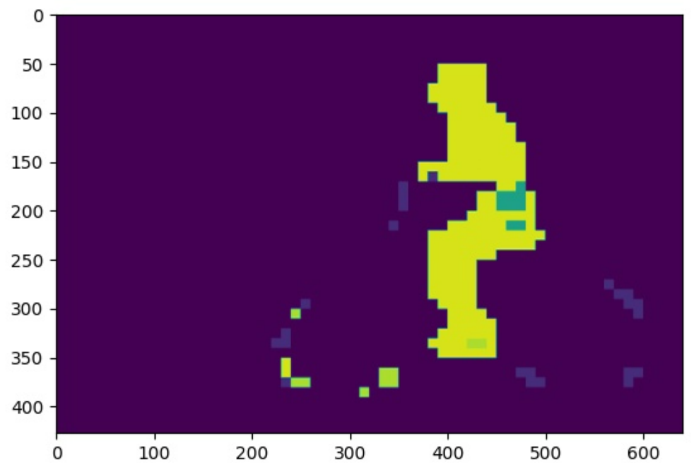


Figure 7.1.: Sample output from *runner_demo.py*

The *image1_out_nearest_best_rtm.jpg* image should now appear in the working directory similar to the one shown above.

Note: Only RTM models are supported in the *runner_demo.py* example. To run the script on a remote target, change the IP address from "127.0.0.1" to the remote one.

7.2 Image classification

The following models target the image classification use case:

- mobilenet_v1_0.25_224
- mobilenet_v1_1.0_224
- mobilenet_v1_050_160
- mobilenet_v2_0.35_224
- mobilenet_v2_1.0_224
- mobilenet_v3-large
- mobilenet_v3-small
- resnet_v2

The files required to run the following examples for MobileNet v1 are in the *workspace\models\mobilenet_v1_1.0_224* folder.

7.2.1 Conversion

The “Converter” tool can be used to convert both the floating-point and quantized models between RTM, TF Lite, and ONNX using the following interface. The TensorFlow models were trained on ImageNet, so a label file is provided.

Note: The “Converter” tool is going to embed labels into the RTM file.

```
deepview-converter --labels labels.txt mobilenet_v1_1.0_224_frozen.pb mobilenet_v1_1.0_
↪224.rtm
```

7.2.2 Validation

A model can be validated either using an *.npz* file (numpy) or using images. To generate an *.npz* file for your model, run the “Validator” tool:

```
deepview-validator
    --input_names input
    --output_names MobilenetV1/Predictions/Reshape_1
    --input_shapes 1,224,224,3
    mobilenet_v1_1.0_224_frozen.pb
```

The *mobilenet_v1_1.0_224_frozen.npz* file should appear in the working directory. You can validate whether your model is working correctly as follows:

```
deepview-validator
    --input_names input
    --output_names MobilenetV1/Predictions/Reshape_1
    --ref_outputs MobilenetV1/Predictions/Reshape_1
    --reference mobilenet_v1_1.0_224_frozen.npz
    mobilenet_v1_1.0_224.rtm
```

Image classification models can also be validated on images. An example is already provided in *Image validation example*.

Note: You can also validate TF Lite models. To do that, you must have the “modelrunner -e tflite” running on the target and set “–labels labels.txt”, which is not embedded in the model.

7.3 Object detection

The following models target the object detection use case:

- mobilenet_ssd_v1
- mobilenet_ssd_v2
- mobilenet_ssd_v3
- yolo_v4

Note: The labels.txt file containing the ImageNet labels is in the *workspace\models\mobilenet_v1_1.0_224* folder. It can be used for all MobileNet image classification and object detection models. This labels.txt file is used in the below commands.

7.3.1 Command-line conversion

The following commands can be used for conversion, evaluation, and validation.

To convert the TF model to RTM, run the following command:

```
deepview-converter
    --labels labels.txt
    --tflite_converter toco
    --input_names Preprocessor/sub
    --output_names concat,concat_1
    --default_shape 1,300,300,3
    frozen_inference_graph.pb
    mobilenet_ssd_v1_1.00_trimmed.rtm
```

To convert the TF model to ONNX, run the following command:

```
deepview-converter
    --default_shape 1,300,300,3
    --input_names Preprocessor/sub
    --output_names concat,concat_1
    frozen_inference_graph.pb
    mobilenet_ssd_v1_1.00_trimmed.onnx
```

To convert the TF model to TF Lite, run the following command:

```
deepview-converter
    --labels labels.txt
    --tflite_converter toco
    --input_names Preprocessor/sub
    --output_names concat,concat_1
    --default_shape 1,300,300,3
    frozen_inference_graph.pb
    mobilenet_ssd_v1_1.00_trimmed.tflite
```

Note: The *frozen_inference_graph.pb* file is used for the COCO-trained MobileNetSSD v1 model. It can be replaced with any other model from the list of object detection models specified above. Make sure that you also specify other parameters correctly and that the model extracted from the downloaded archive is the frozen model.

7.3.2 Command-line validation

To generate a set of inputs and outputs, use the “Validator” tool:

```
deepview-validator
    --input_names Preprocessor/sub
    --output_names concat,concat_1
    --input_shapes 1,300,300,3
    mobilenet_ssd_v1_1.00_trimmed.tflite
```

It generates an *.npz* file that contains the randomized input and the expected output, as determined by a run through TensorFlow. There are additional arguments (”–rtm_inputs” and “–rtm_outputs”) that can be used when you convert the model using custom input and output names. The *.npz* file has a dictionary of numpy arrays associated with the RTM names. For the automatic conversion of names, all “/” symbols are replaced with the “_” symbols. This file can then be used together with the “Validator” as follows:

```
deepview-validator --input_names Preprocessor/sub
    --output_names concat,concat_1
    --uri http://127.0.0.1:10818/v1
    --reference
    mobilenet_ssd_v1_1.00_trimmed.npz
    mobilenet_ssd_v1_1.00_trimmed.rtm
```

It validates all of the outputs when it is run through the “ModelRunner” active at the given URI and it displays the evaluation time of the model. The “–uri” argument can be provided with a list of URIs and it is comma-delimited for validation and benchmarking on multiple devices.

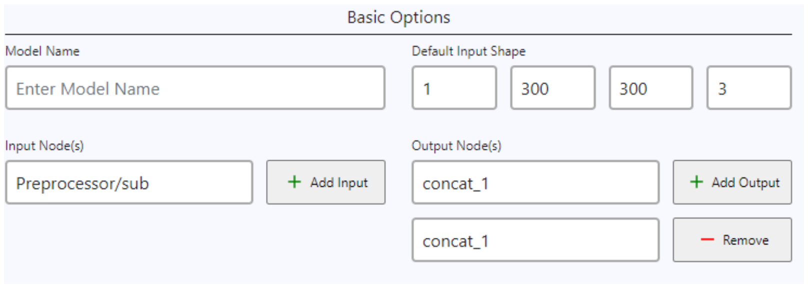
7.3.3 MobileNet SSD model conversion and quantization

The following tutorials demonstrate how to use the Model Tool, a user interface for the Bring Your Own Model workflow, to quantize a pretrained TensorFlow model and convert it to the RTM format. Such model can be deployed using one of the demo applications (*ssdcam-gst in /usr/bin/deepviewrt-examples* on a Yocto BSP image) to run object detection on a camera input.

7.3.3.1 Quantization

A quantized model uses 8-bit integers instead of 32-bit numbers to store data. Additionally, the NPU accelerator is optimized for 8-bit compute, so not only the deployed model is 4 times smaller, but it can also significantly speed up the performance.

1. Download the pre-trained MobileNet SSD v1 floating-point model from TensorFlow models and the sample COCO dataset images. The *mobilenet_ssd_v1* Jupyter notebook provides scripts to download both of them.
2. Open the eIQ Portal and the Model Tool.
3. Open the downloaded model. The model contains a large number of nodes, so select “Load without rendering for conversion”.
4. Select “File > Convert > Tensorflow Lite”.
5. The downloaded pre-trained model contains pre- and post-processing layers. We will trim those and provide static sizes for the tensors. Remove the pre-filled layers and fill in the basic options as follows:
6. Add *labels.txt* to embed labels directly into the model. Choose the TOCO converter to enforce static shapes and enable quantization. Select the “float32” input and output for the model to work correctly with sample


Basic Options

Model Name: Enter Model Name

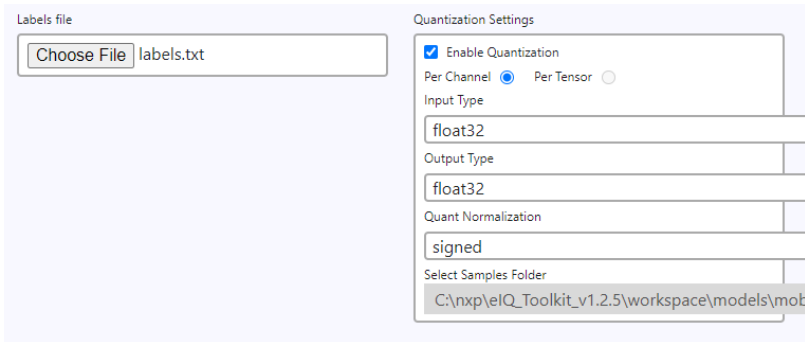
Default Input Shape: 1 | 300 | 300 | 3

Input Node(s): Preprocessor/sub

Output Node(s): concat_1, concat_1

Figure 7.2.: TFLite basic options

applications. Provide the *samples* folder containing COCO images. Provide the complete training dataset for the best calibration. For demo purposes, it is enough to provide only few images.


Labels file: labels.txt

Quantization Settings

Enable Quantization ✓

Per Channel [✓] Per Tensor

Input Type: float32

Output Type: float32

Quant Normalization: signed

Select Samples Folder: C:\nxp\eIQ_Toolkit_v1.2.5\workspace\models\mob

Figure 7.3.: TFLite quantization options

Note: The MLIR converter is the new converter in TensorFlow. It produces dynamic shapes which do not run on the NPU. Therefore, the legacy TOCO converter is more suitable.

7. Click “Convert”. A quantized TF Lite model (which can be deployed directly or used for further conversion to other formats such as RTM) is created.

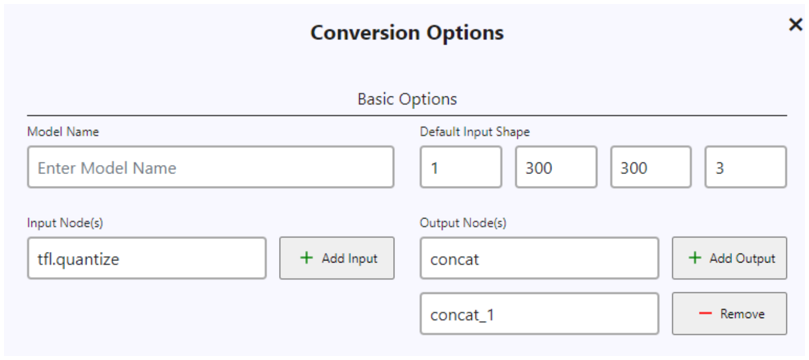
7.3.3.2 Conversion to RTM

RTM is the native format of the DeepView framework. How to convert a model into RTM to use it in one of the DeepViewRT examples is described below. Use the TF Lite model created previously to create an RTM model. To create quantized RTM models, the model must be quantized before the conversion.

1. Open the quantized TF Lite MobileNet SSD v1 model (other MobileNet SSD v1 models should also work).
2. Select “File > Convert > DeepView RT”.
3. Add the input and output layers as follows:

Note: Conversion trims the input layer, so the subsequent “tfl.quantize” is chosen.

4. Add the provided *labels.txt* file to embed labels. Choose the “Per Tensor” quantization type, which was used to create the quantized TF Lite model. Enable the anchor selection, set the constant name to “ssd_anchor_boxes”



Conversion Options

Basic Options

Model Name: Enter Model Name

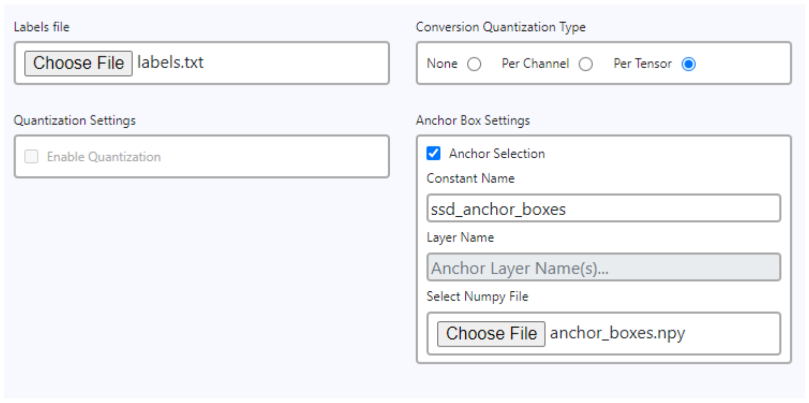
Default Input Shape: 1 300 300 3

Input Node(s): tfl.quantize

Output Node(s): concat, concat_1

Figure 7.4.: RTM basic options

(which is used in the demo applications), and add the provided *anchor_boxes.npy* file. Anchor boxes in an SSD model are a trick to provide all the bounding boxes which the model can output during feature extraction. The output of the model is matched with these anchor boxes during decoding. They can be embedded into the RTM format to be used in an application and do not have to be provided separately by the user.



Labels file: Choose File labels.txt

Conversion Quantization Type: None Per Channel [✓ Per Tensor]

Quantization Settings: Enable Quantization

Anchor Box Settings

[✓ Anchor Selection]

Constant Name: ssd_anchor_boxes

Layer Name: Anchor Layer Name(s)...

Select Numpy File: Choose File anchor_boxes.npy

Figure 7.5.: RTM quantization options

5. Click "Convert". An RTM model will be created, which can be used in one of the example applications or deployed. See the "Example applications" section in the *i.MX Machine Learning User's Guide* (document IMXMLUG) to deploy the model using the DeepViewRT inference engine.

7.4 Pose estimation

The following models target the pose estimation use case:

- Posenet

7.4.1 Conversion

The following commands can be used for conversion, evaluation, and validation.

To convert the TF Lite model to RTM, run the following command:

```
deepview-converter posenet_mobilenet_float_075_1_default_1.tflite ^ posenet_mobilenet_
↪float_075_1_default_1.rtm
```

To convert the TF Lite model to ONNX, run the following command:

```
deepview-converter
    --input_names sub_2
    --output_names float_heatmaps, float_short_offsets, float_mid_offsets, float_segments
    posenet_mobilenet_float_075_1_default_1.tflite
    posenet_mobilenet_float_075_1_default_1.onnx
```

7.4.2 Validation

Use the "Validator" tool to generate a set of inputs and outputs as follows:

```
deepview-validator
    --input_names sub_2
    --output_names float_heatmaps, float_short_offsets, float_mid_offsets, float_
↪segments
    --input_shapes 1,353,257,3
    posenet_mobilenet_float_075_1_default_1.tflite
```

This generates an *.npz* file that contains the randomized input and the expected output, as determined by a run through TensorFlow. There are additional arguments ("–rtm_inputs" and "–rtm_outputs") that can be used when you convert the model using custom input and output names. The *.npz* file has a dictionary of numpy arrays associated with the RTM names. For the automatic conversion of names, all "/" symbols are replaced with the "_" symbols. This file can then be used together with the "Validator" tool.

```
deepview-validator --input_names sub_2
    --output_names float_heatmaps, float_short_offsets, float_mid_offsets, float_
↪segments
    --reference posenet_mobilenet_float_075_1_default_1.npz
    posenet_mobilenet_float_075_1_default_1.rtm
```

Note: If you want to save a *.tflite* version of the model, you can use the additional "–save_tflite=true" argument.

7.4.2.1 Remote validation

Floating-point TFLite models can also be validated remotely on the target when the "modelrunner -H <remote_port> -e tflite" or "-e armnn" is running on the target.

Samples can be a number for random inputs or a folder of images. Reference output tensors are generated from the Python TFLite interpreter as follows:

```
deepview-validator
    --input_names sub_2
    --output_names float_heatmaps, float_short_offsets, float_mid_offsets, float_segments
    --reference posenet_mobilenet_float_075_1_default_1.tflite
    --input_shapes 1,353,257,3
    --samples 5
    --uri http://<remote_ip_address>:<remote_port>/v1
    posenet_mobilenet_float_075_1_default_1.tflite
```

It validates all of the outputs when run through the "ModelRunner" active at the given URI and it displays the evaluation time of the model. The "–uri" argument can be provided with a list of URIs, comma-delimited for validation and benchmarking on multiple devices.

You can also validate the ONNX generated model on a target that is running "modelrunner -H <remote_port> -e onnx" as follows:

```
deepview-validator
    --input_names sub_2
    --output_names float_heatmaps, float_short_offsets, float_mid_offsets, float_
↪segments
    --reference ^posenet_mobilenet_float_075_1_default_1.npz
    --uri http://<remote_ip_address>:<remote_port>/v1
    posenet_mobilenet_float_075_1_default_1.onnx
```

Note: For more details about the remote validation on target, see *Remote validation using ModelRunner*.

8 Revision history

Table 8.1: Revision history

Revision number	Date	Substantive changes
0	15 June 2021	Initial release of eIQ Toolkit 1.0.3
1	24 June 2021	Updated release of eIQ Toolkit 1.0.5
2	19 October 2021	Updated release of eIQ Toolkit 1.1.8
3	18 January 2022	Updated release of eIQ Toolkit 1.2.5
4	31 March 2022	Updated release of eIQ Toolkit 1.3.4
5	8 July 2022	Updated release of eIQ Toolkit 1.4.5
6	3 October 2022	Updated release of eIQ Toolkit 1.5.2
7	1 February 2023	Updated release of eIQ Toolkit 1.6
8	11 April 2023	Updated release of eIQ Toolkit 1.7
9	3 July 2023	Updated release of eIQ Toolkit 1.8

9 Legal information

9.1 Definitions

Draft - A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

9.2 Disclaimers

Limited warranty and liability - Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use - NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications - Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale - NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control - This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products - Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations - A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security - Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

9.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP - wordmark and logo are trademarks of NXP B.V.