Au-Zone
Technologies Inc.

DeepViewRT

Label Camera QML

Au-Zone Technologies Inc.

January 27, 2020

Contents

Introduction

This article details the DeepViewRT Label Camera sample. The sample will demonstrate an application built using QML that can classify video frames using a image classification models. This sample is not meant to showcase the usefulness of image classifiers but simply to show how the integration of DeepView Quick with a simple QML interface, leaving more interesting classification demos as an exercise to the reader.

Obtaining Models

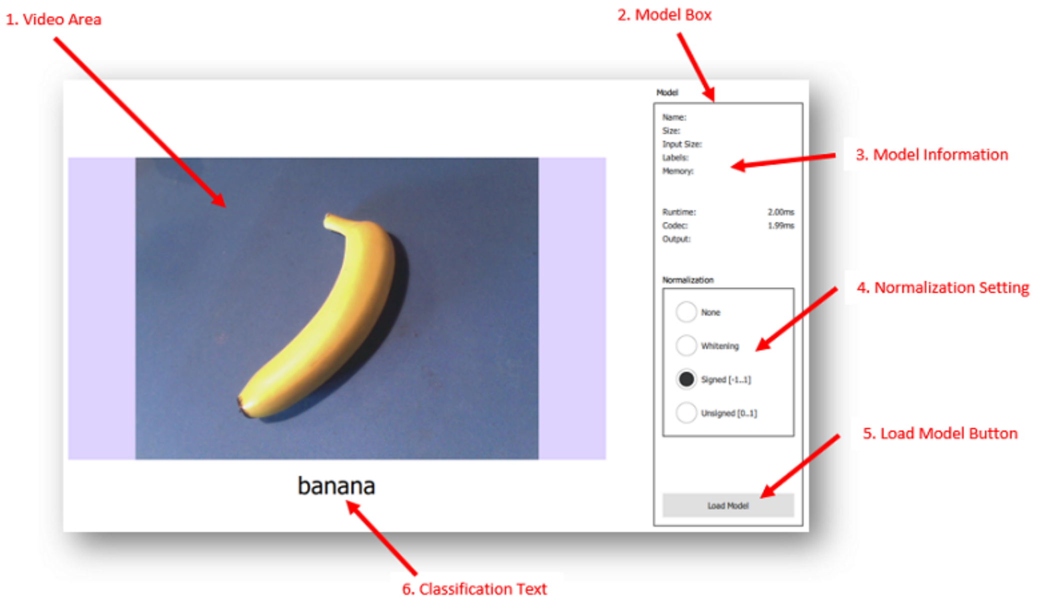
You must first obtain an image-classifier model and convert it to DeepView's RTM format in order to classify an image. We recommend the Mobilenet models trained on the ImageNet dataset. The Mobilenet V2 & V3 models can be obtained from this GitHub page and he V1 models from here

Consult our article on obtaining models for more details.

If you are having poor results with a model, ensure your normalization is correct. Consult Setting a Normalization.

If you are only getting a percentage shown on screen the model is missing labels. This Mobilenet model provided by Tensorflow comes with the correct ImageNet dataset labels. Ensure you select this file for labels, or the labels appropriate for your dataset, during model conversion.

Sample Project



Model

Name:
Size:
Input Size:
Labels:
Memory:

Runtime: 2.00ms
Codec: 1.99ms
Output:

Normalization

None
Whitening
[✓ Signed [-1..1]]
Unsigned [0..1]

banana

Figure 1: Main Window

Models can be uploaded to the interface by simply dragging them to the model box area or by pressing the "Load Model" button. Once a model is uploaded, the application will begin labelling the video frames.

There are 6 major sections of this application defined in the QML UI file.

1. **Video Area** - This is the area that will contain the video. This area is made from a QML Rectangle. In the Main.qml file there is a VideoOutput that has the anchors.fill property set to fill this Rectangle.
2. **Model Box** - Contains information and settings related to the DeepViewRT model. An RTM model can be dragged into this area to load the model.
3. **Model Information** - Contains information and performance timings on the current model. Labels are updated when a new model is loaded. Timings are updated when a new image is classified.
4. **Normalization Setting** - Change the normalization setting of the DeepViewQuick ImageClassifier QML object. The setting is changed with the use of QML RadioButtons.
5. **Load Model Button** - Press the button to open a "load model" QML FileDialog and select a DeepViewRT model file. The DeepViewQuick Model QML object calls its loadModel() function to load the selected model.
6. **Classification Text** - Displays the image's classified label by updating the "text" property of a QML Text item.

Sample Project Architecture

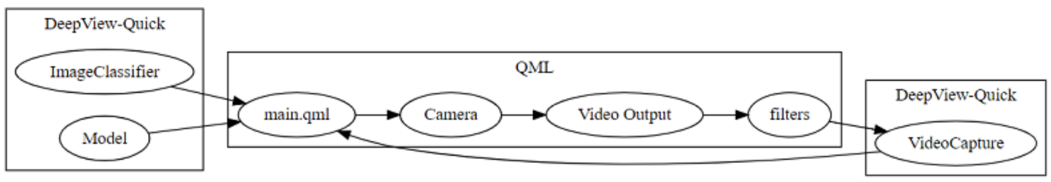


Figure 2: Project Architecture

Camera QML Type

The Camera QML Type allows you to receive video frames from a camera. It also allows you to define many properties depending on your cameras support for these properties.

DeepViewRT QML Types

The DeepViewRT QML library contains several classes that can be registered as QML Types to assist in developing QML applications which make use of neural network models. To use the DeepViewRT QML plugin simply import using the following.

```
import DeepViewRT 2.0
```

Model

The Model C++ class is used to upload and interact with neural network models through QML.

After importing the library, a QML Item can be created and given an id as shown below. It is here that you can declare any Model class settings or signals. For example, the code below shows a new Model object being declared and defining how to change the content of some QML labels with data from the model every time a new model is loaded. It also shows how to display timing information from signals emitted each time the model is run. The Model async setting is set to true by default.

```
Model {
    id: model
    onModelChanged: {
        // Update labels in the model information section
        mainForm.modelName.text = name
        mainForm.modelSize.text = Math.round(size / 1024) + " KB";
        mainForm.modelLabels.text = labelCount;
    }
    // Update timings labels
    onInputTimeChanged: mainForm.modelInputTime.text = (nanoseconds /
        ↪ 1e6).toPrecision(3) + "ms";
    onRunTimeChanged: mainForm.modelRuntime.text = (nanoseconds /
        ↪ 1e6).toPrecision(3) + "ms";
}
```

Loading a Model

A model can be load by passing a file path to the loadModel() function. The example below shows how to load a model from the QML FileDialog

```
FileDialog {
    id: loadModelDialog
    title: "Load Model"
    folder: shortcuts.home
    nameFilters: [ "RTM Models (*.rtm)" ]
    onAccepted: {
        // Get the file path of the chosen model
        var fileName = loadModelDialog.fileUrl.toString()

        // Load the model. Print error message if unsuccessful
        console.log("loading model " + fileName)
        if (!model.loadModel(fileName)) {
            console.log(model.errorString())
        }
    }
}
```

ImageClassifier

The ImageClassifier C++ class is used to classify images through QML using a given neural network model.

After importing the library, a QML Item can be created and given an id as shown below. In the below example, a Model class object is attached to the ImageClassifier and the normalization setting is set to Model.Whitening.

```
ImageClassifier {
    id: classifier
    model: model
    normalization: Model.Whitening
    onErrorChanged: console.log("classifier error: " + error)
}
```

VideoCapture

The VideoCapture C++ class is used to classify video feeds through QML.

After registering the QML type, a QML Item can be created and given an id as shown below. In the below example, the autoCapture setting is set to true which means the captureNext function will automatically be called after each image is captured. The slot "onImageCaptured" is defined to evaluate (classify) an image and then wait for the next image to be captured. Since the Model has the async setting set to true, the video feed will be classified asynchronously.

```
VideoCapture {
    id: capture
    autoCapture: true
    onImageCaptured: classifier.evaluate(image)
    onCaptureTimeChanged: mainForm.captureTime.text = "Capture Time: " +
        ↪ (captureTime / 1e6).toPrecision(3) + "ms"
}
```

Appendix

Setting a Normalization

Normalization of pixel intensity is a common practice in image recognition models. Generally, applying a normalization to training images will allow a model to converge faster and thus it is commonly required to add the same normalization to images input for classification. The ImageClassifier class allows you to select from 4 common normalizations 1. Default (no normalization) 2. Whitening 3. Signed (-1…1) 4. Unsigned (0…1)

The default normalization will apply no normalization to the pixel intensities. Whitening normalization is a method of decorrelating the features of data and placing them on a uniform scale by linearly scaling the data to have a mean of 0 and variance of 1. Whitening is called per_image_standardization in Tensorflow. For a signed normalization, the pixel intensities will be converted from their current range to a range of [-1…1]. For an unsigned normalization, the pixel intensities will be converted from their current range to a range of [0…1].

Pre-trained Mobilenets use signed normalization. However, it depends on what normalization the model is trained with.

If you have a TFLite model you may be able to check the normalization easily by using a model inspection app called Netron. Drop your model into the web app and click on the input layer. Under the input's id and type there should be quantization info. If it is not there then you have to look elsewhere to find the normalization.

If it says -1 ≤ 0.0078125 * (q - 128) ≤ 1, the model uses signed normalization
If it is something else you should check where you obtained the model for info.

Custom Models

The sample was tested using pre-trained MobileNet from TensorFlow which is then converted to DeepView RTM. DeepView RT Models can host the labels internally, so during the RTM export, we also provide a labels file so that we get human readable classifications at runtime. The labels should be in a plain text file with one label per line (empty labels can use an empty line), and the labels should be ordered such that the first line maps to an argmax of 0, and the second line as argmax 1, and so on.

For easily converting a model to DeepView RTM format, the Model Converter GUI application can be used.