Au-Zone Technologies Inc.

DeepViewRT

QML Drawing Demo

Au-Zone Technologies Inc.

January 27, 2020

Contents

Introduction

This article details the DeepViewRT Skeletons sample. The demo will demonstrate an application built using QML that can classify a drawing. One model used for this application is trained on the MNIST dataset of handwritten digits. Another model is trained on the Google Quickdraw dataset of user-generated sketches. This demo is not meant to showcase the usefulness of these models but simply to show how the integration of DeepView Quick with a drawing classification model can be accomplished in QML, leaving more interesting applications as an exercise to the reader.

Sample Project

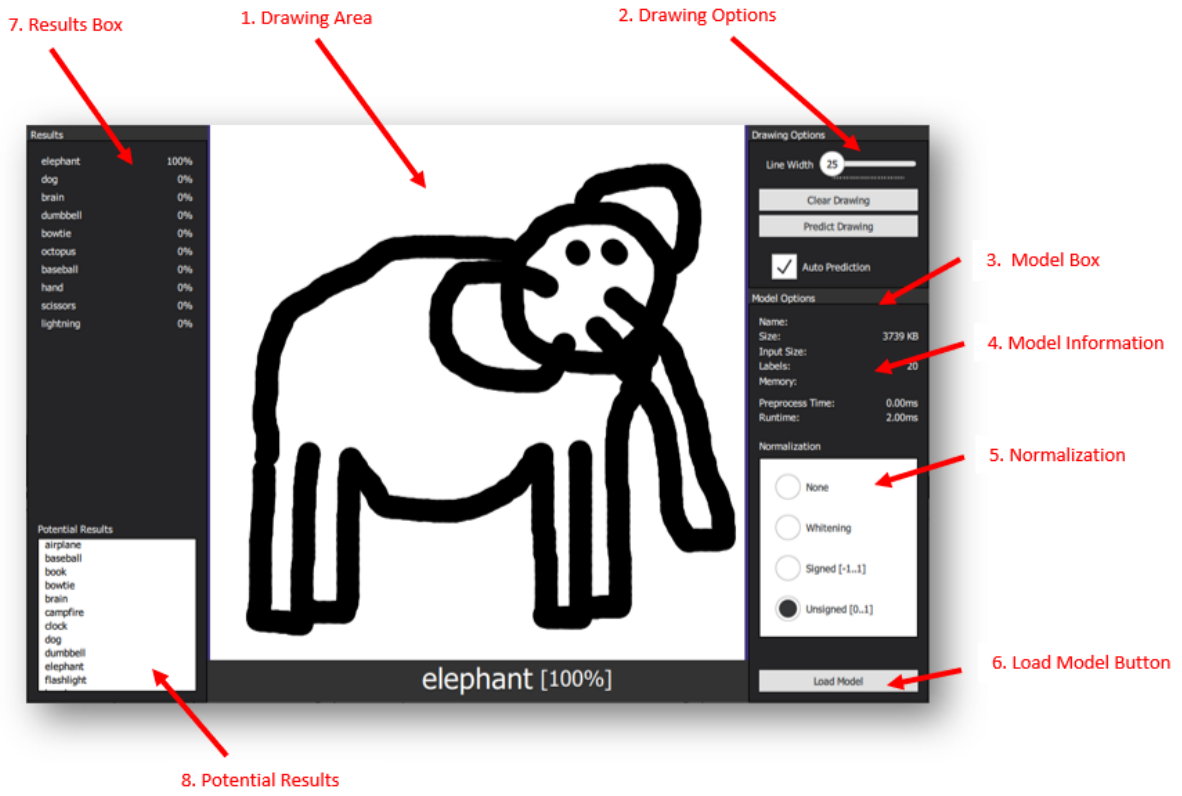

Figure 1: Main Window

Models can be uploaded to the interface by simply dragging the .rtm file over the model box area or by pressing the "Load Model" button and selecting the .rtm file from the dialog. Once a model is uploaded, the application will begin evaluating and displaying images drawn on the canvas.

There are 8 major sections of this application defined in the QML UI file.

1. **Drawing Area** - This is the area that will contain the drawing. This area is made from a QML Canvas. In the Main.qml file there is a Canvas that has the anchors.fill property set to fill this Rectangle.
2. **Drawing Options** - Allows user to change the drawing line width and allows the user to clear the current drawing or classify the current drawing.
3. **Model Box** - Contains information and settings related to the DeepViewRT model. An RTM model can be dragged into this area to load the model.
4. **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 drawing is evaluated.
5. **Normalization** - Change the normalization setting of the DeepViewQuick ProcessDrawing QML object. The setting is changed with the use of QML RadioButtons.
6. **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.
7. **Results Box** - Displays the top 10 results of the classification sorted by percentage.
8. **Potential Results** - List of sketch classes that the model has been trained on and can be classified.

Sample Project Architecture

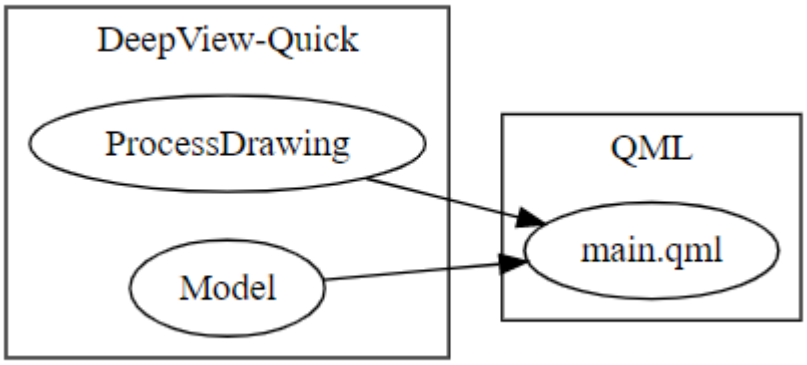

Figure 2: Project Architecture

MNIST Model

The MNIST dataset is based on a large database of handwritten digits (0 - 9). Training on this dataset is seen by many as the "Hello World" of computer vision and even comes included in Tensorflows keras.datasets package. This dataset is made of 1 channel, 28x28 images.

The numbers in this dataset are all near the center of the image and are sized to fit the majority of the 28x28 area. This means the model may give worse results if you do not draw numbers in this way.

For more information, please see the following link: http://yann.lecun.com/exdb/mnist/

Quickdraw Model

The Quickdraw model is trained on a dataset of 1 channel, 28x28 images. Each image is a sketch that generated by players of the game Quick, Draw! released by Google. There are 345 different categories of sketches in this dataset.

For more information, please see the quickdraw-dataset GitHub repository

QML

Canvas QML Type

The Canvas QML object is used to draw the sketches. The mouse position is tracked using a MouseArea QML object. ...

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.

```
1 import DeepViewRT 2.0
```

Model

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

After importing the DeepViewQuick library, a Model 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.

Each time a onModelChanged signal is received, the labels in the listview showing the available classes to draw will be updated as well.

```
Model {
    id: model
    onModelChanged: {
        // Update labels in the model box
        mainForm.modelName.text = name
        mainForm.modelSize.text = Math.round(size / 1024) + " KB";
        mainForm.modelLabels.text = labelCount;

        // Update the labels listview to show the available classes to draw
        mainForm.labelsListView.model.clear()
        for(var i = 0; i < model.labels().length; i++) {
            mainForm.labelsListView.model.append({"label": model.labels()[i]})
        }
    }
    onErrorChanged: console.log(error)
    // Update timings labels
    onInputTimeChanged: mainForm.modelPreprocessTime.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())
        }
    }
}
```

ProcessDrawing

The ProcessDrawing class is derived from the ModelEvaluator class. The ModelEvaluator C++ class is used to evaluate models through QML using a given neural network model. As a derived class of ModelEvaluator, ProcessDrawing is able to override the ModelEvaluators run() function allowing you to first run the model and then do all the necessary post processing before emitting the modelEvaluated signal.

A ProcessDrawing QML Item can be created and given an id as shown below. In the below example, a Model class object is attached to the ProcessDrawing Item, and the normalization is set to Unsigned.

When the ProcessDrawing object receives a signal of "drawingEvaluated", it clears the results ListView and updates it with the new results. When the ProcessDrawing object receives signals of "scoreChanged" and "labelChanged", it updates the bottom text with the top result.

```
ProcessDrawing {
    id: classifier
    model: model
    normalization: Model.Unsigned
    onErrorChanged: console.log("classifier error: " + error)
    onLabelChanged: mainForm.label.text = label // Update classification label
    onScoreChanged: {
        mainForm.score.text = "[" + Math.round(score * 100) + "%]"; // Update
            ↪ score label
    }
    onDrawingEvaluated: {
        resultsModel.clear()
        for(var i = 0; i < labels.length; i++) {
            resultsModel.append({ "label":labels[i], "score":
                ↪ Math.round(scores[i] * 100)})
        }
    }
}
```

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 ProcessDrawing 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. 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].

The included MNIST and Quickdraw sample models were both trained with an unsigned normalization.

Custom Models

The sample uses a model trained on data from Google Quickdraw dataset which is then converted to DeepView RTM.

For easy model conversion to the DeepView RTM format, the Model Converter GUI application can be used.