Au-Zone Technologies Inc.

DeepViewRT

SSD Camera Demo

Au-Zone Technologies Inc.

January 27, 2020

Contents

Introduction

This article details the DeepViewRT Single Shot Detection (SSD) Camera sample. The demo will demonstrate an application built using QML that uses a Single Shot Detection model to detect and overlay a bounding box around detected objects. Single Shot Detection models take one look at an image and detect multiple objects. This compares to Regional Proposal Networks (RPN) that require two steps, one to generate a list of proposed regions of interest and one to detect objects in each of these proposed regions. This demo is not meant to showcase the usefulness of the SSD but simply to show how the integration of DeepView Quick with a SSD model can be accomplished in QML, leaving more interesting applications as an exercise to the reader.

Obtaining Models

This application has been tested with Tensorflow's Mobilenetv1-v3 SSDs obtained from the Tensorflow Object Detection API Model Zoo and converted to tflite. **DeepView only supports SSDs that have been exported for inference**. Tensorflow has a doc here explaining how to convert their checkpoints to an inference protobuf, but we recommend converting to tflite and provide a notebook to convert Tensorflow SSD models to tflite.

Warning: There are issues with the Mobilenet V3 and Mobilenet V2 (not lite) SSDs. The Mobilenet V3 SSD's weights are incorrect and do not produce useful output. There is a relevant GitHub issue here with a link to download a working Mobilenet V3 small SSD

The Mobilenet V2 SSD will throw the error:

```
1 Message type "object_detection.protos.SsdFeatureExtractor" has no field named
    ↪ "batch_norm_trainable".
```

If you edit the pipeline.config file and remove the line that says `batch_norm_trainable: true` you will be able to export it to a .pb file. However, in testing we have found converting this .pb to a .tflite file results in an empty .tflite file but you can just convert the .pb file to .rtm. Relevant GitHub issue here. Mobilenet V2 SSDLite is unaffected.

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 SSD 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

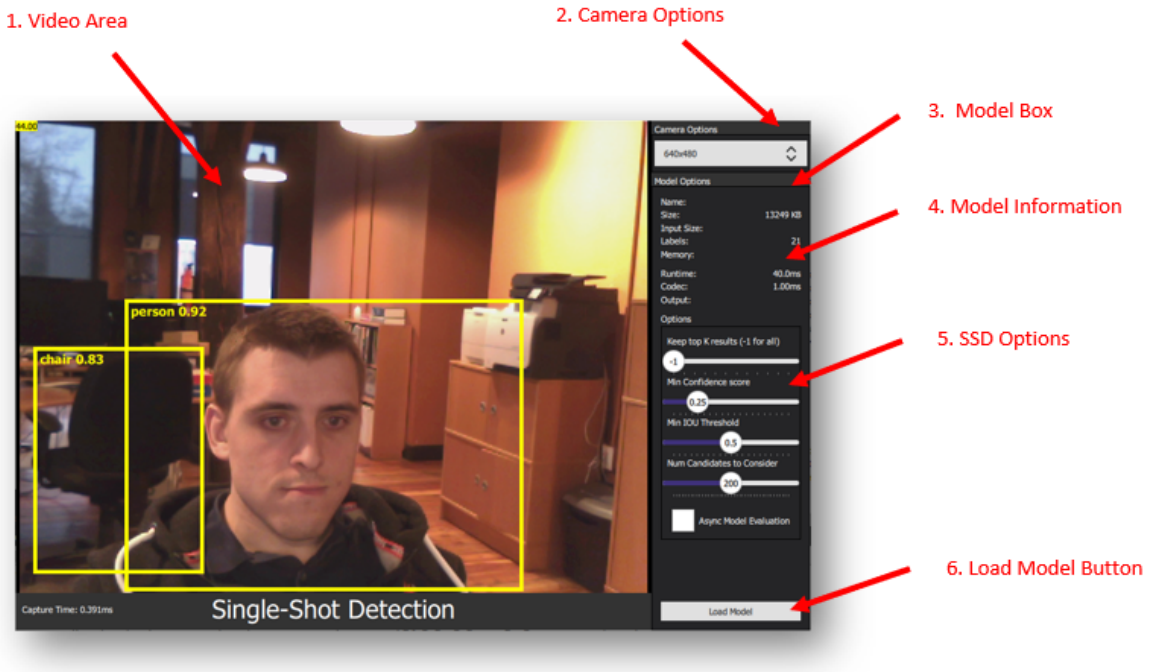

Figure 1: Main Window

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 evaluating and displaying detection boxes.

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. **Camera Options** - Allows user to change the resolution setting of the camera.
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 image is evaluated.
5. **SSD Options** -
    - **NMS Top K Conf. Scores:** Keep a maximum top k detections for each class before Non Maximum Suppression. <= 0 for no limit. Default is -1. Set this to one if you only want one object

per class and so on.

- **Min Confidence Score:** The minimum confidence score needed to display a detected object. Default is 0.5. i.e. The model is 50% or more confident that this object is in that bounding box
- **Min IOU Threshold:** The minimum Intersection Over Union ratio (IOU for two boxes = Area of Overlap / Area of Union) to suppress an object. Raise this if there are too many overlapping boxes. Default is 0.5.
- **Num Candidates:** Only consider this amount of candidates with the highest scores. Default is 200.

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.

Sample Project Architecture

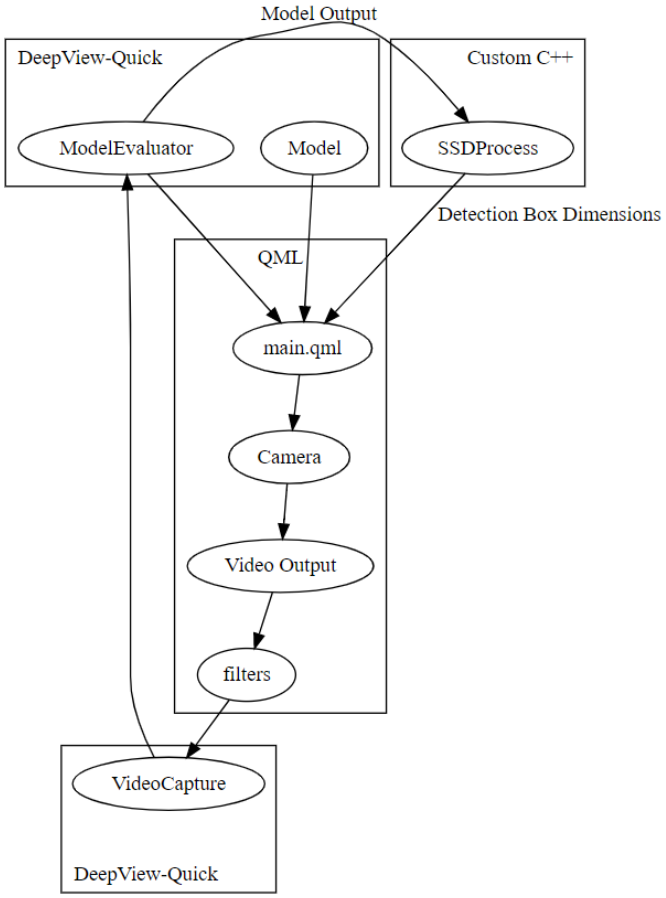

Figure 2: Project Architecture

The DeepView Quick classes, Model, SSDProcess and VideoCapture, are initialized in the main.qml file. It is there that you can declare any settings for the classes and also define actions to be taken when signals are received. In custom C++ classes, you can define custom detection algorithms.

SSD Model

The SSD model is evaluated in two steps.

1. An RGB image is input into the convolutional neural network. This application handles this step using the DeepViewQuick classes

2. The SSDProcess decoding algorithm decodes the model's output into bounding box coordinates for the detected objects.

Each detection box has an X coordinate, Y coordinate, width, and height

For more information on SSD models, please see this GitHub repository: https://github.com/tensorflow/models/tree/master/research/object_detection

or see this blog post: https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab

ssdprocess.cpp

This QML Class is written in a custom C++ class. It takes the output of each model inference and calls all the required functions needed to determine the dimensions of each detected object box. It then emits signals back to the QML that contains the box dimensions.

The decoding is done in DeepViewExtras. An `nnx_ssd_context` object is required to hold the outputs from the decoding. This must be initialized for each model using the `nnx_ssd_init` function, which takes the nnx_ssd_context pointer, the model's context pointer, and the scores and boxes output names. `nnx_ssd_decode` performs the decoding and stores the results in three tensors: **bbx_out_tensor**, **bbx_probs_tensor**, and **num_bbx_per_class_tensor**.

The **bbx_out_tensor** has a shape of [NUM_CLASSES, NUM_ANCHORS, 4], so for each class, and for each object predicted to belong to that class, it holds the coordinates of the bounding box in the format [x_min, y_min, x_max, y_max] i.e. the corners of the rectangle.

The **bbx_probs_tensor** has a shape of [NUM_CLASSES, NUM_ANCHORS] which holds the confidence score for each bounding box.

Finally, the **num_bbx_per_class_tensor** is a 1 dimensional tensor (array) containing NUM_CLASSES integers which identify how many detections there were for each class.

Currently, the emitted signal contains four ordered arrays. These arrays contain the x coordinate, y coordinate, width, and height of each detection box. Each index of these arrays is mapped to the viewfinder and drawn in the SSDProcess onObjectDetected slot in the Main.qml file.

QML

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.

VideoOutput QML Type

The VideoOutput QML Type is used to display the video output from a certain source. The source property must be set to the source item which is providing the video frames such as a MediaPlayer or Camera Item.

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. Async is set to false be default, as there is currently performance issues with asynchronous model inference. Enable it if you prefer smooth video to inference speed. 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 captureNext function belonging to the VideoCapture class is called as a new model is loaded in order to start capturing new images to be evaluated by the model.

```
1  Model {
2      id: deepviewModel
3      async: false
4      onModelChanged: {
5          // Capture first image
6          capture.captureNext()
7          // Update labels in the model information section
8          mainForm.modelName.text = name
9          mainForm.modelSize.text = Math.round(size / 1024) + " KB";
10         mainForm.modelLabels.text = labelCount;
11     }
12
13     // Update timings labels
14     onInputTimeChanged: mainForm.modelInputTime.text =
15         (nanoseconds / 1e6).toPrecision(3) + "ms";
16     onRunTimeChanged: mainForm.modelRuntime.text =
17         (nanoseconds / 1e6).toPrecision(3) + "ms";
18 }
```

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())
        }
    }
}
```

VideoCapture

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

After importing the DeepViewQuick library, a VideoCapture QML Item can be created and given an id as shown below. In the below example, the autoCapture setting is set to false which means the captureNext function will need to be called manually before the next image is captured. The slot "onImageCaptured" is defined to evaluate (model inference and post processing) an image and then wait for the next image to be captured.

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

SSDProcess

The SSDProcess 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,

SSDProcess 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 SSDProcess QML Item can be created and given an id as shown below. In the below example, a Model class object is attached to the SSDProcess Item, the normalization is set to Unsigned by default and the async classification setting is set to false.

When the SSDProcess object receives a signal of "modelEvaluated", it calls the captureNext function belonging to the VideoCapture class. This causes VideoCapture to capture another image and perform another evaluation on it.

```
SSDProcess {
    id: evaluator
    model: model
    normalization: Model.Unsigned
    property bool showScore: true
    onErrorChanged: {
        if (!model.fileName) {
            console.log("No model loaded.");
        } else if(error) {  // Handle error cleared signals
            console.log("Evaluator error: " + error);
        }
    }
    onModelEvaluated: {
        capture.captureNext();
    }
}
```

Overlay

Detection boxes and labels are drawn using a QML Rectangle. For each detection, a QML Rectangle is generated using the following code

```
var component, detectBox;
component = Qt.createComponent("DetectBox.qml");
detectBox = component.createObject(viewfinder, {id: "box" + i, x: 0, y: 0});
```

Once generated, each rectangle has its dimensions and location set using the values returned from the detection filter.

Each rectangle is then stored in a list of QML Items to keep track of the rectangles. This allows them to later be removed using the destroy() command.

A rectangle can be mapped to a viewfinder using the following code, and providing the dimensions x, y, w, h.

```
var r = viewfinder.mapNormalizedRectToItem(Qt.rect(x, y, w, h));
```

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 SSDProcess 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].

Most of Tensorflow's SSDs use unsigned normalization. However, it depends on what normalization the model is trained with. Our SSDs use signed normalization.

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 $0 \leq q \leq 255$, the model uses unsigned normalization.

If it says $-1 \leq 0.0078125 * (q - 128) \leq 1$, the model uses signed normalization

If it is something else it is most likely Whitening/standardized normalization.

Here is a list of known normalizations for pre-trained SSDS:

- Mobilenet V1 0.75 & 1.0 SSD: Unsigned
- Mobilenet V1 1.0 Quantized: Signed
- Mobilenet V2 SSD & SSDLite: Unsigned
- Mobilenet V3 SSD: Signed
- DeepView SSDs: Signed

Custom Models

The sample uses pre-trained Mobilenet SSD models from TensorFlow's Object Detection API which are then converted to DeepView's RTM format. SSDs trained on the VOC 2007+2012 dataset are also provided as well as novel Lego Marvel Superheroes and fruit detectors.

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