Au-Zone Technologies Inc.

DeepViewRT

PoseNet Skeletons Demo QML

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 detect and overlay an outline of a person or persons' joints and limbs onto a video feed using a PoseNet model. PoseNet is a model trained by researchers for human pose estimation and is based on the MobileNet model. This demo is not meant to showcase the usefulness of PoseNet but simply to show how the integration of DeepView Quick with a pose estimation model can be accomplished in QML, leaving more interesting applications as an exercise to the reader.

Obtaining Models

Since PoseNet originates from Tensorflow.js it is difficult to obtain a regular Tensorflow graph of the model. For a quick start, Tensorflow provides a single person tflite model. The single person model has very simple decoding. However, it is a large model so it is quite slow and only supports detection of one person.

Consult our article on obtaining models for more details.

The single-pose model can be obtained from here. The multi-pose model is from Tensorflow.js, so it is more involved to convert to a regular Tensorflow model, but this GitHub repository allows one to do so.

Sample Project

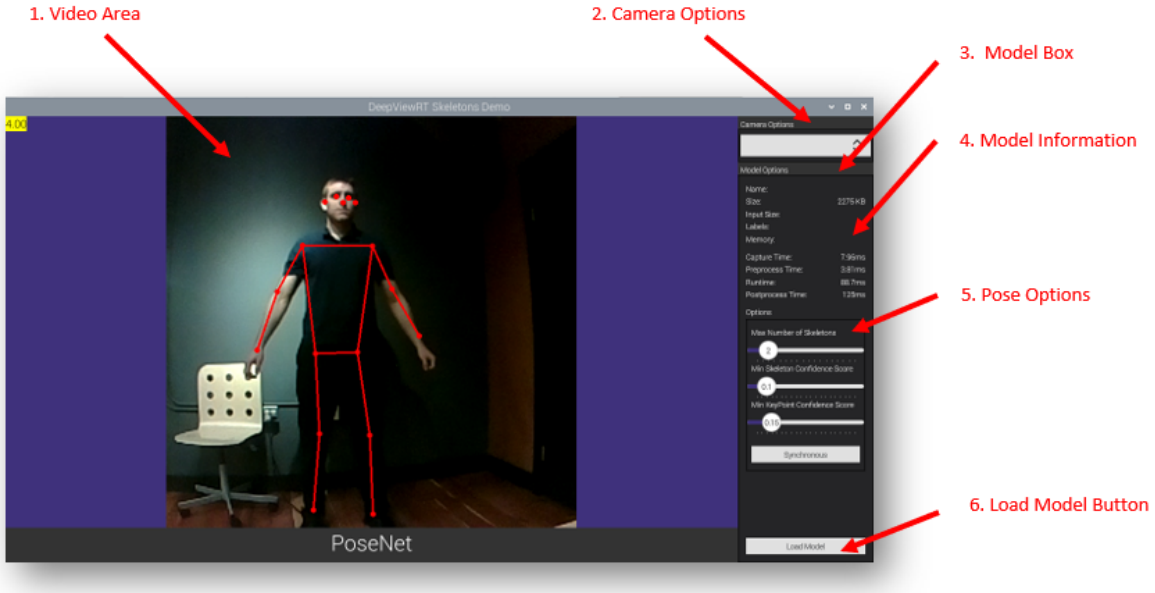

Figure 1: Main Window

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 poses.

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. **Pose Options** -
    - **Max Number of Skeletons:** The maximum number of skeletons that will be displayed on the screen at one time.

- **Min Skeleton Confidence Score:** The minimum confidence score needed to display a detected skeleton.
- **Min KeyPoint Confidence Score:** The minimum confidence score needed to display any individual keypoint within a skeleton.

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

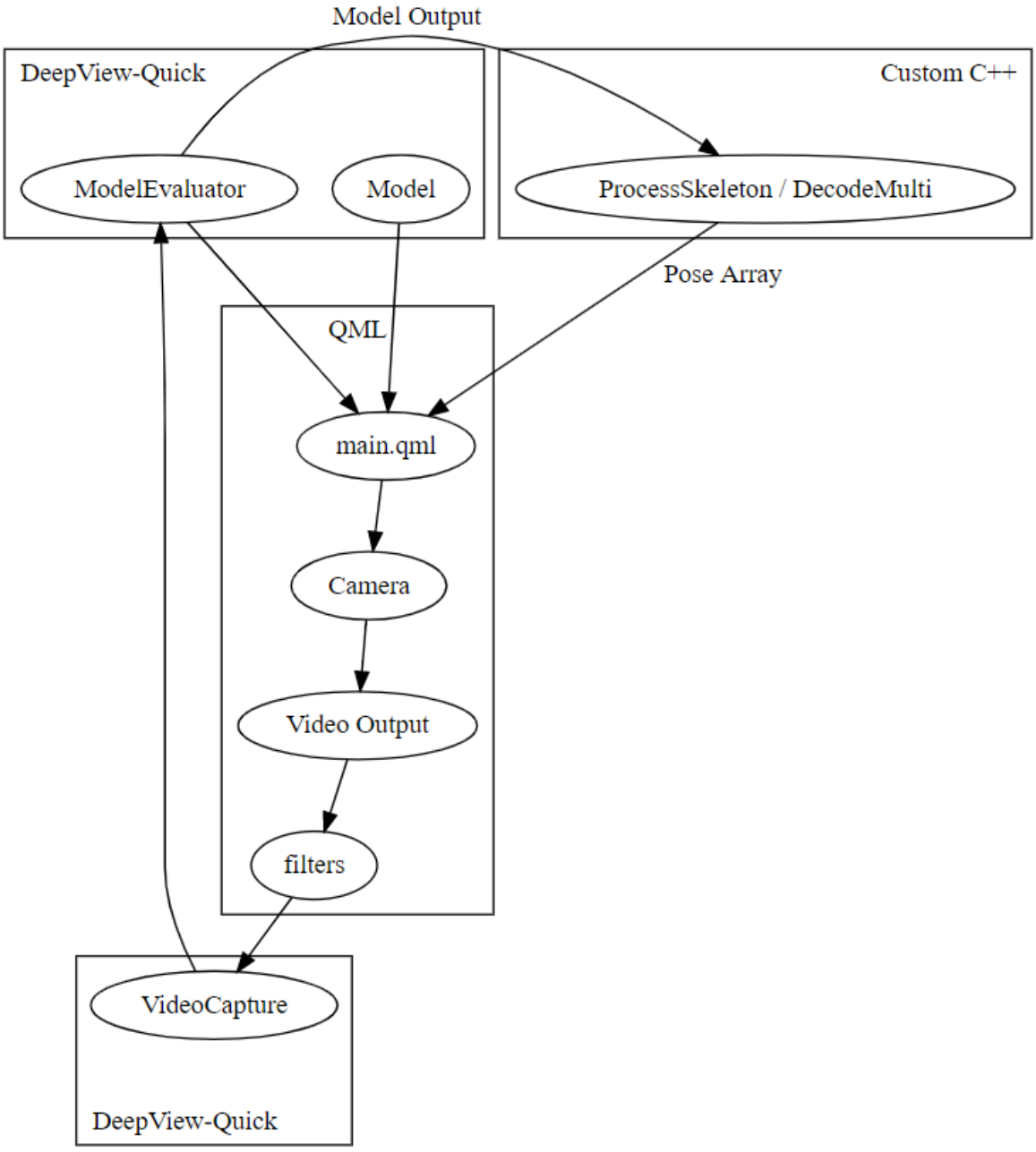

Figure 2: Project Architecture

The DeepView Quick classes, Model, ProcessSkeleton 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.

PoseNet Model

The PoseNet 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

2a. In the case of a multi-pose model. The multi-pose decoding algorithm decodes the model's output into poses, pose confidence scores, keypoint positions, and keypoint confidence scores

2b. In the case of a single-pose model. The single-pose decoding algorithm decodes the model's output into a single keypoint positions vector and their confidences. The format remains the same regardless.

Poses: This is a vector that contains a list of keypoints and a confidence score for each pose.

Pose Confidence Score: The overall confidence ranging between 0.0 and 1.0 that a pose exists in the image.

Keypoints: The output of the model is an array of 2D keypoints (x, y) based on the dimensions of the original image. These keypoints are arranged in the following order.

0. nose
1. leftEye
2. rightEye
3. leftEar
4. rightEar
5. leftShoulder
6. rightShoulder
7. leftElbow
8. rightElbow
9. leftWrist
10. rightWrist
11. leftHip
12. rightHip
13. leftKnee
14. rightKnee
15. leftAnkle
16. rightAnkle

Keypoint Confidence Score: The overall confidence ranging between 0.0 and 1.0 that a keypoint exists within a pose.

For more information, please see this GitHub repository: https://github.com/tensorflow/tfjs-models/tree/master/posenet

or see this blog post: https://medium.com/tensorflow/real-time-human-pose-estimation-in-the-browser-with-tensorflow-js-7dd0bc881cd5

ProcessSkeleton.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 coordinates of each keypoint in the current pose. It then emits signals back to the QML that contains the keypoint information. Currently, the emitted signal contains two ordered arrays. One of these arrays contains the x coordinate of each keypoint in a pose. The other array is the y coordinates. These two arrays are combined back into a point array from the QML slot.

decodeMulti.cpp

This is a custom C++ class for decoding the output of a model and is called from the ProcessSkeleton class. The model output does not simply provide the keypoints but instead provides more complex information that can be decoded into the keypoint information. The keypoint information returned will differ based on the settings provided to the decodeMulti class.

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. 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. Async is set to false by default due to performance issues with it enabled as well as the skeletons not lining up with what is on screen as well. If you prefer smooth video to inference speed you can turn it on.

```
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 = (captureTime /
        ↪ 1e6).toPrecision(3) + "ms"
}
```

ProcessSkeleton

The ProcessSkeleton 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,

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

When the ProcessSkeleton 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.

```
ProcessSkeleton {
    id: evaluator
    model: model
    crop: getCrop()
    normalization: Model.Signed
    onErrorChanged: {
        if (!model.fileName) {
            console.log("No model loaded.");
        } else if(error) {  // Handle error cleared signals
            console.log("Evaluator error: " + error);
        }
    }
    onModelEvaluated: {
        capture.captureNext()
    }
}
```

In this example, a crop is set on the ProcessSkeleton since PoseNet requires a square image. The function to determine the largest square crop possible is shown below where newX is the new x location of the determined crop. The input to the crop setting is a Qt rect basic type.

```
function getCrop() {
    var height = camera.viewfinder.resolution.height
    var width = camera.viewfinder.resolution.width
    var newX = parseInt((width / 2) - (height / 2))
    return Qt.rect(newX, 0, height, height)
}
```

Overlay

The skeleton connection lines and nodes are drawn using multiple QML Shape objects. At the start of the application, the maximum of 17 skeletons are generated with their visibility set to false. When there is a detection to be displayed, the corresponding skeleton's visibility is set back to true. The code required to generate a skeleton is shown below.

```
function generateSkeletons(num) {
    for (var i = 0; i < num; i++) {
```

```
        var component = Qt.createComponent("Skeleton.qml")
        var skeleton = component.createObject(viewfinder, {id: "skeleton", x:0,
            ↪ y:0, height: viewfinder.height, width: viewfinder.width})
        skeletonItem.skeletons.push(skeleton)
    }
}
```

After each skeleton is generated, they are stored in an array to allow for further use.

After generating a skeleton, some display properties can be modified. The below code shows examples of modifying the skeleton's color.

```
// Redefine the first skeleton
skeletonItem.skeletons[0].skeletonConnectionsColorArray =
    ↪ ["green","green","green","green",
"green","green","yellow","green","yellow","yellow","yellow","yellow"]
skeletonItem.skeletons[0].skeletonNodesColorArray =
    ↪ ["blue","blue","blue","blue",
"blue","green","green","green","green","green","green","yellow","yellow",
"yellow","yellow","yellow","yellow"]
skeletonItem.skeletons[0].define()
//Redefine the second skeleton
skeletonItem.skeletons[1].connectionColor = "purple"
skeletonItem.skeletons[1].nodeColor = "purple"
skeletonItem.skeletons[1].define()
```

The nodes that are connected can also be modified by redefining the connections array. The default connections array to display the PoseNet skeleton is shown below. It shows which pairs of nodes are connected in the skeleton.

```
property var connections: [[5,7],[6,8],[7,9],[8,10],[5,11],[5,6],
[11,12],[6,12],[11,13],[12,14],[13,15],[14,16]]
```

The first skeleton is redefined to individually set the color of each node and connection line. The second skeleton is redefined to set the color of all nodes and connection lines to purple.

To automatically call functions as the application starts up, the Component.onCompleted signal is used.

Once a poses keypoints are returned, each skeleton has its nodes and connection line locations reset using the returned keypoints.

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 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].

All PoseNet models used Signed normalization

Custom Models

The sample uses a pre-trained PoseNet from TensorFlow which is then converted to DeepView RTM.

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