Introduction

In this lesson we'll cover:

- Basics of OpenCV
- Handling Input Streams in OpenCV
- Processing Model Outputs for Additional Useful Information
- The Basics of MQTT and their use with IoT devices
- Sending statistics and video streams to a server
- Performance basics
- And finish up by thinking about additional model use cases, as well as end user needs

OpenCV Basics

OpenCV is an open-source library for various image processing and computer vision techniques that runs on a highly optimized C++ back-end, although it is available for use with Python and Java as well. It's often helpful as part of your overall edge applications, whether using it's built-in computer vision techniques or handling image processing.

Uses of OpenCV

There's a lot of uses of OpenCV. In your case, you'll largely focus on its ability to capture and read frames from video streams, as well as different pre-processing techniques, such as resizing an image to the expected input size of your model.

It also has other pre-processing techniques like converting from one color space to another, which may help in extracting certain features from a frame.

There are also plenty of computer vision techniques included, such as Canny Edge detection, which helps to extract edges from an image, and it extends even to a suite of different machine learning classifiers for tasks like face detection.

Useful OpenCV function

- VideoCapture - can read in a video or image and extract a frame from it for processing
- resize is used to resize a given frame
- cvtColor can convert between color spaces.
    - You may remember from awhile back that TensorFlow models are usually trained with RGB images, while OpenCV is going to load frames as BGR. There was a technique with the Model Optimizer that would build the TensorFlow model to appropriately handle BGR. If you did not add that additional argument at the time, you could use this function to convert each image to RGB, but that's going to add a little extra processing time.
- rectangle - useful for drawing bounding boxes onto an output image
- imwrite - useful for saving down a given image

Further Research

OpenCV has some pretty extensive tutorials available if you want to dive deeper into this useful computer vision library. We'll look at some of the relevant material on handling camera and video inputs next.

Handling Input Streams

Being able to efficiently handle video files, image files, or webcam streams is an important part of an edge application. If I were to be running the webcam on my Macbook for instance and performing inference, a surprisingly large amount of resources get used up simply to use the webcam. That's why it's useful to utilize the OpenCV functions built for this - they are about as optimized for general use with input streams as you will find.

Open & Read A Video

We saw the cv2.VideoCapture function in the previous video. This function takes either a zero for webcam use, or the path to the input image or video file. That's just the first step, though. This "capture" object must then be opened with capture.open.
Then, you can basically make a loop by checking if capture.isOpened, and you can read a frame from it with capture.read. This read function can actually return two items, a boolean and the frame. If the boolean is false, there's no further frames to read, such as if the video is over, so you should break out of the loop

Closing the Capture

Once there are no more frames left to capture, there's a couple of extra steps to end the process with OpenCV.

- First, you'll need to release the capture, which will allow OpenCV to release the captured file or stream
- Second, you'll likely want to use cv2.destroyAllWindows. This will make sure any additional windows, such as those used to view output frames, are closed out
- Additionally, you may want to add a call to cv2.waitKey within the loop, and break the loop if your desired key is pressed. For example, if the key pressed is 27, that's the Escape key on your keyboard - that way, you can close the stream midway through with a single button. Otherwise, you may get stuck with an open window that's a bit difficult to close on its own.

Gathering Useful Information from Model Outputs

Training neural networks focus a lot on the accuracy, such as detecting the right bounding boxes and having them placed in the right spot. But what should you actually do with bounding boxes, semantic masks, classes, etc.? How would a self-driving car make a decision about where to drive based solely off the semantic classes in an image?

It's important to get useful information from your model - information from one model could even be further used in an additional model, such as traffic data from one set of days being used to predict traffic on another set of days, such as near to a sporting event.

For the traffic example, you'd likely want to count how many bounding boxes there are, but also make sure to only count once for each vehicle until it leaves the screen. You could also consider which part of the screen they come from, and which part they exit from. Does the left turn arrow need to last longer near to a big event, as all the cars seem to be heading in that direction?

In an earlier exercise, you played around a bit with the confidence threshold of bounding box detections. That's another way to extract useful statistics - are you making sure to throw out low confidence predictions?

Intro to MQTT

MQTT

MQTT stands for MQ Telemetry Transport, where the MQ came from an old IBM product line called IBM MQ for Message Queues (although MQTT itself does not use queues). That doesn't really give many hints about its use. MQTT is a lightweight publish/subscribe architecture that is designed for resource-constrained devices and low-bandwidth setups. It is used a lot for Internet of Things devices, or other machine-to-machine communication, and has been around since 1999. Port 1883 is reserved for use with MQTT.

Publish/Subscribe

In the publish/subscribe architecture, there is a broker, or hub, that receives messages published to it by different clients. The broker then routes the messages to any clients subscribing to those particular messages.

This is managed through the use of what are called "topics". One client publishes to a topic, while another client subscribes to the topic. The broker handles passing the message from the publishing client on that topic to any subscribers. These clients therefore don't need to know anything about each other, just the topic they want to publish or subscribe to.

MQTT is one example of this type of architecture, and is very lightweight. While you could publish information such as the count of bounding boxes over MQTT, you cannot publish a video frame using it. Publish/subscribe is also used with self-driving cars, such as with the Robot Operating System, or ROS for short. There, a stop light classifier may publish on one topic, with an intermediate system that determines when to brake subscribing to that topic, and then that system could publish to another topic that the actual brake system itself subscribes to.

Further Research

- Visit the main site for MQTT
- A helpful post on more of the basics of MQTT

Communicating with MQTT

There is a useful Python library for working with MQTT called paho-mqtt. Within, there is a sub-library called client, which is how you create an MQTT client that can publish or subscribe to the broker.

To do so, you'll need to know the IP address of the broker, as well as the port for it. With those, you can connect the client, and then begin to either publish or subscribe to topics.

Publishing involves feeding in the topic name, as well as a dictionary containing a message that is dumped to JSON. Subscribing just involves feeding in the topic name to be subscribed to.

Further Research

- As usual, documentation is your friend. Make sure to check out the documentation on PyPi related to the paho-mqtt Python library if you want to learn more about its functionality.
- Intel® has a pretty neat IoT tutorial on working with MQTT with Python you can check out as well.

Streaming Images to a Server

Sometimes, you may still want a video feed to be streamed to a server. A security camera that detects a person where they shouldn't be and sends an alert is useful, but you likely want to then view the footage. Since MQTT can't handle images, we have to look elsewhere.
At the start of the course, we noted that network communications can be expensive in cost, bandwidth and power consumption. Video streaming consumes a ton of network resources, as it requires a lot of data to be sent over the network, clogging everything up. Even with high-speed internet, multiple users streaming video can cause things to slow down. As such, it's important to first consider whether you even need to stream video to a server, or at least only stream it in certain situations, such as when your edge AI algorithm has detected a particular event.

FFmpeg

Of course, there are certainly situations where streaming video is necessary. The FFmpeg library is one way to do this. The name comes from "fast forward" MPEG, meaning it's supposed to be a fast way of handling the MPEG video standard (among others).
In our case, we'll use the ffserver feature of FFmpeg, which, similar to MQTT, will actually have an intermediate FFmpeg server that video frames are sent to. The final Node server that displays a webpage will actually get the video from that FFmpeg server.
There are other ways to handle streaming video as well. In Python, you can also use a flask server to do some similar things, although we'll focus on FFmpeg here.

Setting up FFmpeg

You can download FFmpeg from ffmpeg.org. Using ffserver in particular requires a configuration file that we will provide for you. This config file sets the port and IP address of the server, as well as settings like the ports to receive video from, and the framerate of the video. These settings can also allow it to listen to the system stdout buffer, which is how you can send video frames to it in Python.

Sending frames to FFmpeg

With the sys Python library, can use sys.stdout.buffer.write(frame) and sys.stdout.flush() to send the frame to the ffserver when it is running.

If you have a ffmpeg folder containing the configuration file for the server, you can launch the ffserver with the following from the command line:

```
sudo ffserver -f ./ffmpeg/server.conf
```

From there, you need to actually pipe the information from the Python script to FFmpeg. To do so, you add the | symbol after the python script (as well as being after any related arguments to that script, such as the model file or CPU extension), followed by ffmpeg and any of its related arguments.

For example:

```
python app.py -m “model.xml” | ffmpeg -framerate 24
```

And so on with additional arguments before or after the pipe symbol depending on whether they are for the Python application or for FFmpeg.

Further Research

We covered FFMPEG and ffserver, but as you may guess, there are also other ways to stream video to a browser. Here are a couple other options you can investigate for your own use:

- Set up Your Own Server on Linux
- Use Flask and Python

Handling Statistics and Images from a Node Server

Node.js is an open-source environment for servers, where Javascript can be run outside of a browser. Consider a social media page, for instance - that page is going to contain different content for each different user, based on their social network. Node allows for Javascript to run outside of the browser to gather the various relevant posts for each given user, and then send those posts to the browser.
In our case, a Node server can be used to handle the data coming in from the MQTT and FFmpeg servers, and then actually render that content for a web page user interface.

More on Front-End

Check out the Front End Developer Nanodegree program if you want to learn more about these skills!

Analyzing Performance Basics

We've talked a lot about optimizing for inference and running apps at the edge, but it's important not to skip past the accuracy of your edge AI model. Lighter, quicker models are helpful for the edge, and certain optimizations like lower precision that help with these will have some impact on accuracy, as we discussed earlier on.

No amount of skillful post-processing and attempting to extract useful data from the output will make up for a poor model choice, or one where too many sacrifices were made for speed.

Of course, it all depends on the exact application as to how much loss of accuracy is acceptable. Detecting a pet getting into the trash likely can handle less accuracy than a self-driving car in determining where objects are on the road.

The considerations of speed, size and network impacts are still very important for AI at the Edge. Faster models can free up computation for other tasks, lead to less power usage, or allow for use of cheaper hardware. Smaller models can also free up memory for other tasks, or allow for devices with less memory to begin with. We've also discussed some of the network impacts earlier. Especially for remote edge devices, the power costs of heavy network communication may significantly hamper their use,

Lastly, there can be other differences in cloud vs edge costs other than just network effects. While potentially lower up front, cloud storage and computation costs can add up over time. Data sent to the cloud could be intercepted along the way. Whether this is better or not at the edge does depend on a secure edge device, which isn't always the case for IoT.

Further Research

- We'll cover more on performance with the Intel® Distribution of OpenVINO™ Toolkit in later courses, but you can check out the developer docs here for a preview.
- Did you know Netflix uses 15% of worldwide bandwidth with its video streaming? Cutting down on streaming your video to the cloud vs. performing work at the edge can vastly cut down on network costs.

Glossary

OpenCV

A computer vision (CV) library filled with many different computer vision functions and other useful image and video processing and handling capabilities.

MQTT

A publisher-subscriber protocol often used for IoT devices due to its lightweight nature. The paho-mqtt library is a common way of working with MQTT in Python.

Publish-Subscribe Architecture

A messaging architecture whereby it is made up of publishers, that send messages to some central broker, without knowing of the subscribers themselves. These messages can be posted on some given “topic”, which the subscribers can then listen to without having to know the publisher itself, just the “topic”.

Publisher

In a publish-subscribe architecture, the entity that is sending data to a broker on a certain “topic”.

Subscriber

In a publish-subscribe architecture, the entity that is listening to data on a certain “topic” from a broker.

Topic

In a publish-subscribe architecture, data is published to a given topic, and subscribers to that topic can then receive that data.

FFmpeg

Software that can help convert or stream audio and video. In the course, the related ffserver software is used to stream to a web server, which can then be queried by a Node server for viewing in a web browser.

Flask

A Python framework useful for web development and another potential option for video streaming to a web browser.

Node Server

A web server built with Node.js that can handle HTTP requests and/or serve up a webpage for viewing in a browser.