NANODEGREE PROGRAM  
Intel® Edge AI Scholarship Foundation Course  
Learn the basics of AI at the Edge, leverage pre-trained models, convert models with the Model Optimizer, utilize the Inference Engine, and learn some additional topics for edge applications.

……..

Core Curriculum

This section consists of all the lessons and projects you need to complete in order to receive your certificate.

* **3 PARTS**
* **0 PROJECTS**

1. **1**

**PART 1**

[**Welcome to the Intel® Edge AI Foundation course!**](https://classroom.udacity.com/nanodegrees/nd132/parts/8e008c56-fe52-4c2b-aa06-e44aca597ffc)

**Estimated time: 5 minutes**

1. **2**

**PART 2**

[**Intel® Edge AI Foundation Course**](https://classroom.udacity.com/nanodegrees/nd132/parts/3eafdf71-b35c-4525-9b2c-d46f7dc300aa)

Welcome to Intel® Edge AI Foundations, where you'll learn about the basics of AI at the Edge, leverage pre-trained models available with the Intel® Distribution of OpenVINO Toolkit™, convert and optimize other models with the Model Optimizer, and perform inference with the Inference Engine. Additionally, you'll learn some additional topics for edge applications, like MQTT and how to stream video to servers.

**Estimated time: 11 hours 30 minutes**

1. **3**

**PART 3**

[**Certificate of Completion**](https://classroom.udacity.com/nanodegrees/nd132/parts/6d1c8e08-272d-4100-8971-5d40969fc29a)

Congratulations! You completed Intel Foundation course - OpenVINO Fundamentals!

……………………

### Resources:

To download OpenVINO toolkit on your system, please follow the link <https://software.intel.com/en-us/openvino-toolkit/choose-download>.

Take the opportunity to sign-up and learn more about Intel developer resources: <https://software.seek.intel.com/us-en-go-to-market>

=================================

**PART 2**

[**Intel® Edge AI Foundation Course**](https://classroom.udacity.com/nanodegrees/nd132/parts/3eafdf71-b35c-4525-9b2c-d46f7dc300aa)

##### **LESSON 1**

#### **Introduction to AI at the Edge**

Get introduced to AI at the Edge, and find out about the topics you’ll learn throughout the rest of the course.

**…….**

Instructor Intro

<https://www.youtube.com/watch?v=KogUwtKPv7Q&feature=emb_logo>

...

What is AI at the Edge?

<https://www.youtube.com/watch?v=4dro5ZHQLa4&feature=emb_logo>

The edge means local (or near local) processing, as opposed to just anywhere in the cloud. This can be an actual local device like a smart refrigerator, or servers located as close as possible to the source (i.e. servers located in a nearby area instead of on the other side of the world).

The edge can be used where low latency is necessary, or where the network itself may not always be available. The use of it can come from a desire for real-time decision-making in certain applications.

Many applications with the cloud get data locally, send the data to the cloud, process it, and send it back. The edge means there’s no need to send to the cloud; it can often be more secure (depending on edge device security) and have less impact on a network. Edge AI algorithms can still be trained in the cloud, but get run at the edge.

**AI APPLICATION**

**CLOUD VS. EDGE**

Voice Assistants

Cloud

Self-Driving Cars

Edge

Insights from Millions of Sales Transactions

Cloud

Remote Nature Camera

Edge

……

# Why is AI at the Edge Important?

# video

* Network communication can be expensive (bandwidth, power consumption, etc.) and sometimes impossible (think remote locations or during natural disasters)
* Real-time processing is necessary for applications, like self-driving cars, that can't handle latency in making important decisions
* Edge applications could be using personal data (like health data) that could be sensitive if sent to cloud
* Optimization software, especially made for specific hardware, can help achieve great efficiency with edge AI models

………..

# Applications of AI at the Edge

Video

* There are nearly endless possibilities with the edge.
* IoT devices are a big use of the edge.
* Not every single app needs it - you can likely wait a second while your voice app goes to ask the server a question, or such as when NASA engineers are processing the latest black hole data.

## Applications of AI at the Edge Quiz

What are some more applications of AI at the Edge that you can think of?

### Your reflection

**Rescue Bots that save those injured in disasters**

……..

# Historical Context

Video

* Cloud computing has gotten a lot of the news in recent years, but the edge is also growing in importance.
* [Per Intel®](https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html), IoT growth has gone from 2 billion devices in 2006 to a projected 200 billion by 2020.
* From the first network ATMs in the 1970's, to the World Wide Web in the 90's, and on up to smart meters in early 2000's, we've come a long way.
* From the constant use devices like phones to smart speakers, smart refrigerators, locks, warehouse applications and more, the IoT pool keeps expanding.

……

# Course Structure

Video

* In this course, we’ll largely focus on AI at the Edge using the [Intel® Distribution of OpenVINO™ Toolkit](https://software.intel.com/en-us/openvino-toolkit).
* First, we’ll start off with pre-trained models available in the OpenVINO™ Open Model Zoo. Even without needing huge amounts of your own data and costly training, you can deploy powerful models already created for many applications.
* Next, you’ll learn about the Model Optimizer, which can take a model you trained in frameworks such as TensorFlow, PyTorch, Caffe and more, and create an Intermediate Representation (IR) optimized for inference with OpenVINO™ and Intel® hardware.
* Third, you’ll learn about the Inference Engine, where the actual inference is performed on the IR model.
* Lastly, we'll hit some more topics on deploying at the edge, including things like handling input streams, processing model outputs, and the lightweight MQTT architecture used to publish data from your edge models to the web.

……..

# Why Are the Topics Distinct?

Video

* Pre-trained models can be used to explore your options without the need to train a model. This pre-trained model can then be used with the Inference Engine, as it will already be in IR format. This can be integrated into your app and deployed at the edge.
* If you created your own model, or are leveraging a model not already in IR format (TensorFlow, PyTorch, Caffe, MXNet, etc), use the Model Optimizer first. This will then feed to the Inference Engine, which can be integrated into your app and deployed at the edge.
* While you'll be able to perform some amazingly efficient inference after feeding into the Inference Engine, you'll still want to appropriately handle the output for the edge application, and that's what we'll hit in the final lesson.

Model Optimizer

Convert a model for further use with the toolkit

Open Model Zoo

Obtain a pre-trained model

Inference Engine

Perform inference on an Intermediate Representation

……….

# Relevant Tools and Prerequisites

Video

### Summary

* Prerequisites:
  + Understand some of the basics of computer vision and how AI models are created.
  + Basic Python or C++ experience. This first course is mainly in Python, although C++ can be used with the Intel® Distribution of OpenVINO™ Toolkit easily as well (and can be faster in a completed app!).
* We will **not** be training models in this course, as our focus is on optimization & deployment at the edge.
* Classroom workspaces will be available for exercises, so no set-up required if you plan to use them.

### Local Set-up

* Make sure to make note of the [hardware requirements](https://software.intel.com/en-us/openvino-toolkit/hardware) for the Intel® Distribution of OpenVINO™ Toolkit if you want to work locally.
* If you do want to do the exercises on your local machine (or perhaps even on a set-up like a Raspberry Pi with an [Intel® Neural Compute Stick 2](https://software.intel.com/en-us/articles/intel-neural-compute-stick-2-and-open-source-openvino-toolkit)), you can follow the instructions below for your operating system.

**Note**: The classroom workspaces in this course use the 2019 R3 version of the toolkit, so there may be some variance in syntax from newer versions.

Top of Form

Download the Toolkit

Downlod:

<https://software.intel.com/en-us/articles/intel-neural-compute-stick-2-and-open-source-openvino-toolkit>

Bottom of Form

### Intel® DevCloud - Edge

There is also the new [Intel® DevCloud](https://software.intel.com/en-us/devcloud/edge) platform for testing out edge environments. This allows you to have access to a range of Intel® hardware such as CPUs, GPUs, FPGAs, Neural Compute Stick, and more. Later courses will get more into the hardware side of things, but this another option for working with an edge environment.

…………….

# What You Will Build

Video

In the project at the end of the course, you’ll build and deploy a People Counter App at the Edge. In the project, you will:

* Convert a model to an Intermediate Representation (IR)
* Use the IR with the Inference Engine
* Process the output of the model to gather relevant statistics
* Send those statistics to a server, and
* Perform analysis on both the performance and further use cases of your model.

### Project Demo

Below, you can find a quick video demo of the project running and returning statistics on the number of people in frame, average duration spent in frame, and the total number of people counted so far. This is all sent from an edge application to a web server. Note that there is no audio associated with this application.

<https://www.youtube.com/watch?time_continue=3&v=uWkMjUiqk0M&feature=emb_logo>

Video

………..

# Recap

Video

## Recap Summary

In this introductory lesson, we covered:

* The basics of the edge
* The importance of the edge and its history
* Edge applications
* The structure of the course
  + Pre-Trained Models
  + The Model Optimizer
  + The Inference Engine
  + More edge topics (MQTT, servers, etc.)
* An overview of the project

We'll kick things off next by starting to look at Pre-Trained Models, and how they are useful for Edge Applications

##### LESSON 2

#### Leveraging Pre-Trained Models

Utilize Pre-Trained Models from the Intel® Distribution of OpenVINO™ Toolkit to build powerful edge applications, without the need to train your own model.

CONTINUE

**2 hours left**

……………

# Introduction

<https://www.youtube.com/watch?v=vFNZu1VpdwE>

In this lesson we'll cover:

* Basics of the Intel® Distribution OpenVINO™ Toolkit
* Different Computer Vision model types
* Available Pre-Trained Models in the Software
* Choosing the right Pre-Trained Model for your App
* Loading and Deploying a Basic App with a Pre-Trained Model

…..

# The Intel® Distribution of OpenVINO™ Toolkit

<https://www.youtube.com/watch?v=-pM9pLCnzJk>

The OpenVINO™ Toolkit’s name comes from “**Open** **V**isual **I**nferencing and Neural **N**etwork **O**ptimization”. It is largely focused around optimizing neural network inference, and is open source.

It is developed by Intel®, and helps support fast inference across Intel® CPUs, GPUs, FPGAs and Neural Compute Stick with a common API. OpenVINO™ can take models built with multiple different frameworks, like TensorFlow or Caffe, and use its Model Optimizer to optimize for inference. This optimized model can then be used with the Inference Engine, which helps speed inference on the related hardware. It also has a wide variety of Pre-Trained Models already put through Model Optimizer.

By optimizing for model speed and size, OpenVINO™ enables running at the edge. This does not mean an increase in inference accuracy - this needs to be done in training beforehand. The smaller, quicker models OpenVINO™ generates, along with the hardware optimizations it provides, are great for lower resource applications. For example, an IoT device does not have the benefit of multiple GPUs and unlimited memory space to run its apps.

### QUIZ QUESTION

The Intel® Distribution of OpenVINO™ Toolkit is:

### QUIZ QUESTION

The Intel® Distribution of OpenVINO™ Toolkit is:

* 

A deep learning framework used to train models

* 

A programming language optimized for deep learning inference

* 

An open source library useful for edge deployment due to its performance maximizations and pre-trained models

* 

A new cloud service available for neural networks

..

We’ll be using the OpenVINO™ toolkit throughout the course, but if you’re ready to dive in already, you can visit the [main site](https://software.intel.com/en-us/openvino-toolkit) for the Intel® Distribution of OpenV

………………….

# Pre-Trained Models in OpenVINO™

<https://www.youtube.com/watch?v=1-Vije0cMBQ>

In general, pre-trained models refer to models where training has already occurred, and often have high, or even cutting-edge accuracy. Using pre-trained models avoids the need for large-scale data collection and long, costly training. Given knowledge of how to preprocess the inputs and handle the outputs of the network, you can plug these directly into your own app.

In OpenVINO™, Pre-Trained Models refer specifically to the Model Zoo, in which the Free Model Set contains pre-trained models already converted using the Model Optimizer. These models can be used directly with the Inference Engine.

### Further Research

We’ll come back to the various pre-trained models available with the OpenVINO™ Toolkit shortly, but you can get a headstart by checking out the documentation [here](https://software.intel.com/en-us/openvino-toolkit/documentation/pretrained-models).

………….

Very Important:

# Types of Computer Vision Models

<https://www.youtube.com/watch?v=E8yBgSKfCoo>

We covered three types of computer vision models in the video: Classification, Detection, and Segmentation.

Classification determines a given “class” that an image, or an object in an image, belongs to, from a simple yes/no to thousands of classes. These usually have some sort of “probability” by class, so that the highest probability is the determined class, but you can also see the top 5 predictions as well.

Detection gets into determining that objects appear at different places in an image, and oftentimes draws bounding boxes around the detected objects. It also usually has some form of classification that determines the class of an object in a given bounding box. The bounding boxes have a confidence threshold so you can throw out low-confidence detections.

Segmentation classifies sections of an image by classifying each and every pixel. These networks are often post-processed in some way to avoid phantom classes here and there. Within segmentation are the subsets of semantic segmentation and instance segmentation - the first wherein all instances of a class are considered as one, while the second actually consider separates instances of a class as separate objects.

**QUIZ QUESTION**

Match the types of Computer Vision models below to their descriptions.

*Submit to check your answer choices!*

**DESCRIPTION**

**TYPE OF MODEL**

Determines what an object in a given image is, although not where in the image it is located.

Classification

Determines the location of an object in an image on a pixel-by-pixel basis.

Segmentation

Determines the location of an object using some type of markers like a bounding box around the area the object is in.

Detection

Great job! Knowing the types of inference computer vision models perform is important in considering both the data (and its annotations) and network models to use.

### Further Research

Here is a useful [Medium post](https://medium.com/analytics-vidhya/image-classification-vs-object-detection-vs-image-segmentation-f36db85fe81) if you want to go a little further on types of computer vision models.

………

# Case Studies in Computer Vision

<https://www.youtube.com/watch?v=7mUaovlA4aQ>

We focused on SSD, ResNet and MobileNet in the video. SSD is an object detection network that combined classification with object detection through the use of default bounding boxes at different network levels. ResNet utilized residual layers to “skip” over sections of layers, helping to avoid the vanishing gradient problem with very deep neural networks. MobileNet utilized layers like 1x1 convolutions to help cut down on computational complexity and network size, leading to fast inference without substantial decrease in accuracy.

One additional note here on the ResNet architecture - the paper itself actually theorizes that very deep neural networks have convergence issues due to exponentially lower convergence rates, as opposed to just the vanishing gradient problem. The vanishing gradient problem is also thought to be helped by the use of normalization of inputs to each different layer, which is not specific to ResNet. The ResNet architecture itself, at multiple different numbers of layers, was shown to converge faster during training than a “plain” network without the residual layers.

### QUESTION 1 OF 2

The [Single Shot Multibox Detector (SSD)](https://arxiv.org/abs/1512.02325) model:

* 

Was the first ever integrated model to combine object localization and classification in a single network

* 

Performed classifications on different convolutional layer feature maps using default bounding boxes

* 

Was created to enable the possibility of deep learning inference on mobile devices

* 

Used skip layers to allow for a deeper network

SUBMIT

SSD is known for performing classification off of multiple feature maps from throughout the neural network, using default bounding box sizes.

### QUESTION 2 OF 2

The “residual learning” achieved in the [ResNet](https://arxiv.org/pdf/1512.03385.pdf" \t "_blank) model architecture is achieved by:

* Using “skip” layers that pass information forward by a couple of layers
* 

Combining multiple convolutional filters in a single layer

* 

Leveraging a pre-trained model and fine-tuning it on new data

* 

Using multiple different learning rates during training

SUBMIT

ResNet helped open the door for substantially deeper neural networks than were possible before. “Skip” layers helped neural networks avoid the [vanishing gradient problem](https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484) that otherwise occurs in deep networks.

### Further Research

Getting used to reading research papers is a key skill to build when working with AI and Computer Vision. Below, you can find the original research papers on some of the networks we discussed in this section.

* [SSD](https://arxiv.org/abs/1512.02325)
* [YOLO](https://arxiv.org/abs/1506.02640)
* [Faster RCNN](https://arxiv.org/abs/1506.01497)
* [MobileNet](https://arxiv.org/abs/1704.04861)
* [ResNet](https://arxiv.org/abs/1512.03385)
* [Inception](https://arxiv.org/pdf/1409.4842.pdf)

………………..

# Available Pre-Trained Models in OpenVINO™

<https://www.youtube.com/watch?v=SoTH1jr3-HA>

Most of the Pre-Trained Models supplied by OpenVINO™ fall into either face detection, human detection, or vehicle-related detection. There is also a model around detecting text, and more!

Models in the Public Model Set must still be run through the Model Optimizer, but have their original models available for further training and fine-tuning. The Free Model Set are already converted to Intermediate Representation format, and do not have the original model available. These can be easily obtained with the Model Downloader tool provided in the files installed with OpenVINO™.

The SSD and MobileNet architectures we discussed previously are often the main part of the architecture used for many of these models.

### QUIZ QUESTION

Which of the below models are available in OpenVINO™ as pre-trained models?

* Text Detection
* Pose Detection
* Roadside Segmentation
* 

Art Generation

* Pedestrian Detection

There are a ton of pre-trained models available through the OpenVINO™ Toolkit, although so far there are none using [Generative Adversarial Networks (GANs)](https://en.wikipedia.org/wiki/Generative_adversarial_network), which would be useful in the case of [Art Generation](https://towardsdatascience.com/gangogh-creating-art-with-gans-8d087d8f74a1).

You can check out the full list of pre-trained models available in the Intel® Distribution of OpenVINO™ [here](https://software.intel.com/en-us/openvino-toolkit/documentation/pretrained-models). As we get into the Model Optimizer in the next lesson, you’ll find it’s quite easy to take pre-trained models available from other sources and use them with OpenVINO™ as well.

…………

# 7 - Exercise: Loading Pre-Trained Models

Jupyter notebook file

…..

# Solution: Loading Pre-Trained Models

Video

### Choosing Models

I chose the following models for the three tasks:

* Human Pose Estimation: [**human-pose-estimation-0001**](https://docs.openvinotoolkit.org/latest/_models_intel_human_pose_estimation_0001_description_human_pose_estimation_0001.html)
* Text Detection: [**text-detection-0004**](http://docs.openvinotoolkit.org/latest/_models_intel_text_detection_0004_description_text_detection_0004.html)
* Determining Car Type & Color: [**vehicle-attributes-recognition-barrier-0039**](https://docs.openvinotoolkit.org/latest/_models_intel_vehicle_attributes_recognition_barrier_0039_description_vehicle_attributes_recognition_barrier_0039.html)

### Downloading Models

To navigate to the directory containing the Model Downloader:

cd /opt/intel/openvino/deployment\_tools/open\_model\_zoo/tools/downloader

Within there, you'll notice a downloader.py file, and can use the -h argument with it to see available arguments. For this exercise, --name for model name, and --precisions, used when only certain precisions are desired, are the important arguments. Note that running downloader.py without these will download **all** available pre-trained models, which will be multiple gigabytes. You can do this on your local machine, if desired, but the workspace will not allow you to store that much information.

**Note**: In the classroom workspace, you will not be able to write to the /opt/intel directory, so you should also use the -o argument to specify your output directory as /home/workspace (which will download into a created intel folder therein).

##### Downloading Human Pose Model

sudo ./downloader.py --name human-pose-estimation-0001 -o /home/workspace

##### Downloading Text Detection Model

sudo ./downloader.py --name text-detection-0004 --precisions FP16 -o /home/workspace

##### Downloading Car Metadata Model

sudo ./downloader.py --name vehicle-attributes-recognition-barrier-0039 --precisions INT8 -o /home/workspace

### Verifying Downloads

The downloader itself will tell you the directories these get saved into, but to verify yourself, first start in the /home/workspace directory (or the same directory as the Model Downloader if on your local machine without the -o argument). From there, you can cd intel, and then you should see three directories - one for each downloaded model. Within those directories, there should be separate subdirectories for the precisions that were downloaded, and then .xml and .bin files within those subdirectories, that make up the model.

………

# Optimizations on the Pre-Trained Models

Video

In the exercise, you dealt with different precisions of the different models. Precisions are related to floating point values - less precision means less memory used by the model, and less compute resources. However, there are some trade-offs with accuracy when using lower precision. There is also fusion, where multiple layers can be fused into a single operation. These are achieved through the Model Optimizer in OpenVINO™, although the Pre-Trained Models have already been run through that process. We’ll return to these optimization techniques in the next lesson.

………

# Choosing the Right Model for Your App

Make sure to test out different models for your application, comparing and contrasting their use cases and performance for your desired task. Remember that a little bit of extra processing may yield even better results, but needs to be implemented efficiently.

This goes both ways - you should try out different models for a single use case, but you should also consider how a given model can be applied to multiple use cases. For example, being able to track human poses could help in physical therapy applications to assess and track progress of limb movement range over the course of treatment.

**QUIZ QUESTION**

Below are a few existing pre-trained models, as well as some different applications they might be used in. In your best judgement, match these models to which application would best be fit for their use.

*Submit to check your answer choices!*

**POTENTIAL APPLICATION**

**MODEL**

Traffic Light Optimization

Detect People, Vehicles and Bikes

Assess Traffic Levels in Retail Aisles

Pedestrian Detection

Delivery Robot

Identify Roadside Objects

Monitor Form When Working Out

Human Pose Estimation

…………

# Pre-processing Inputs

<https://www.youtube.com/watch?v=E9huKos96Uk>

The pre-processing needed for a network will vary, but usually this is something you can check out in any related documentation, including in the OpenVINO™ Toolkit documentation. It can even matter what library you use to load an image or frame - OpenCV, which we’ll use to read and handle images in this course, reads them in the BGR format, which may not match the RGB images some networks may have used to train with.

Outside of channel order, you also need to consider image size, and the order of the image data, such as whether the color channels come first or last in the dimensions. Certain models may require a certain normalization of the images for input, such as pixel values between 0 and 1, although some networks also do this as their first layer.

In OpenCV, you can use cv2.imread to read in images in BGR format, and cv2.resize to resize them. The images will be similar to a numpy array, so you can also use array functions like .transpose and .reshape on them as well, which are useful for switching the array dimension order.

…..

# 12 Exercise - Pre-processing Inputs

….

# 13 Solution: Pre-processing Inputs

<https://www.youtube.com/watch?v=erNsB5nXgW4>

Using the documentation pages for each model, I ended up noticing they needed essentially the same preprocessing, outside of the height and width of the input to the network. The images coming from cv2.imread were already going to be BGR, and all the models wanted BGR inputs, so I didn't need to do anything there. However, each image was coming in as height x width x channels, and each of these networks wanted channels first, along with an extra dimension at the start for batch size.

So, for each network, the preprocessing needed to 1) re-size the image, 2) move the channels from last to first, and 3) add an extra dimension of 1 to the start. Here is the function I created for this, which I could call for each separate network:

**def** **preprocessing**(input\_image, height, width):

'''

Given an input image, height and width:

- Resize to height and width

- Transpose the final "channel" dimension to be first

- Reshape the image to add a "batch" of 1 at the start

'''

image = cv2.resize(input\_image, (width, height))

image = image.transpose((2,0,1))

image = image.reshape(1, 3, height, width)

**return** image

Then, for each model, I can just call this function with the height and width from the documentation:

##### Human Pose

preprocessed\_image = preprocessing(preprocessed\_image, 256, 456)

##### Text Detection

preprocessed\_image = preprocessing(preprocessed\_image, 768, 1280)

##### Car Meta

preprocessed\_image = preprocessing(preprocessed\_image, 72, 72)

### Testing

To test your implementation, you can just run python test.py.

……

# Handling Network Outputs

<https://www.youtube.com/watch?v=pREe4P5yygM>

Like the computer vision model types we discussed earlier, we covered the primary outputs those networks create: classes, bounding boxes, and semantic labels.

Classification networks typically output an array with the softmax probabilities by class; the argmax of those probabilities can be matched up to an array by class for the prediction.

Bounding boxes typically come out with multiple bounding box detections per image, which each box first having a class and confidence. Low confidence detections can be ignored. From there, there are also an additional four values, two of which are an X, Y pair, while the other may be the opposite corner pair of the bounding box, or otherwise a height and width.

Semantic labels give the class for each pixel. Sometimes, these are flattened in the output, or a different size than the original image, and need to be reshaped or resized to map directly back to the input.

### Quiz Information

In a network like [**SSD**](https://arxiv.org/pdf/1512.02325.pdf) that we discussed earlier, the output is a series of bounding boxes for potential object detections, typically also including a confidence threshold, or how confident the model is about that particular detection.

Therefore, inference performed on a given image will output an array with multiple bounding box predictions including: the class of the object, the confidence, and two corners (made of xmin, ymin, xmax, and ymax) that make up the bounding box, in that order.

### QUIZ QUESTION

Given the information above, which of the following could be used to extract the bounding boxes from a given network output (given an already defined desired confidence as conf\_threshold)?

[Note: width and height in the choices below would be that of the original input image.]

* 
* **for** box in output:
* **if** box[0] > conf\_threshold:
* xmin = **int**(box[1] \* width)
* ymin = **int**(box[2] \* height)
* xmax = **int**(box[3] \* width)
* ymax = **int**(box[4] \* height)
* 
* **for** box in output:
* xmin = **int**(box[2] \* width)
* ymin = **int**(box[3] \* height)
* xmax = **int**(box[4] \* width)
* ymax = **int**(box[5] \* height)
* 
* **for** box in output:
* **if** box[1] > conf\_threshold:
* xmin = **int**(box[2] \* width)
* ymin = **int**(box[3] \* height)
* xmax = **int**(box[4] \* width)
* ymax = **int**(box[5] \* height)
* 
* **for** box in output:
* **if** box[1] > conf\_threshold:
* xmin = **int**(box[2])
* ymin = **int**(box[3])
* xmax = **int**(box[4])
* ymax = **int**(box[5])

### Nice work! Being able to extract the output correctly from a neural network is crucial to implementing inference with an app

### Further Research

* [**Here**](https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab) is a great write-up on working with SSD and its output
* This [**post**](https://thegradient.pub/semantic-segmentation/) gets into more of the differences in moving from models with bounding boxes to those using semantic segmentation

………..

# Running Your First Edge App

<https://www.youtube.com/watch?v=FANZZXUqGac>

You have now learned the key parts of working with a pre-trained model: obtaining the model, preprocessing inputs for it, and handling its output. In the upcoming exercise, you’ll load a pre-trained model into the Inference Engine, as well as call for functions to preprocess and handle the output in the appropriate locations, from within an edge app. We’ll still be abstracting away some of the steps of dealing with the Inference Engine API until a later lesson, but these should work similarly across different models.

……….

# Exercise: Deploy an App at the Edge

# Solution: Deploy an App at the Edge

This was a tough one! It takes a little bit to step through this solution, as I want to give you some of my own techniques to approach this rather difficult problem first. The solution video is split into three parts - the first focuses on adding in the preprocessing and output handling calls within the app itself, and then into how I would approach implementing the Car Meta model's output handling.

### Early Steps and Car Meta Model Output Handling

### https://www.youtube.com/watch?v=X9yI7U2Rn00

The code for calling preprocessing and utilizing the output handling functions from within app.py is fairly straightforward:

preprocessed\_image = preprocessing(image, h, w)

This is just feeding in the input image, along with height and width of the network, which the given inference\_network.load\_model function actually returned for you.

output\_func = handle\_output(args.t)

processed\_output = output\_func(output, image.shape)

This is partly based on the helper function I gave you, which can return the correct output handling function by feeding in the model type. The second line actually sends the output of inference and image shape to whichever output handling function is appropriate.

##### Car Meta Output Handling

Given that the two outputs for the Car Meta Model are "type" and "color", and are just the softmax probabilities by class, I wanted you to just return the np.argmax, or the index where the highest probability was determined.

**def** **handle\_car**(output, input\_shape):

'''

Handles the output of the Car Metadata model.

Returns two integers: the argmax of each softmax output.

The first is for color, and the second for type.

'''

*# Get rid of unnecessary dimensions*

color = output['color'].flatten()

car\_type = output['type'].flatten()

*# TODO 1: Get the argmax of the "color" output*

color\_pred = np.argmax(color)

*# TODO 2: Get the argmax of the "type" output*

type\_pred = np.argmax(car\_type)

**return** color\_pred, type\_pred

##### Run the Car Meta Model

I have moved the models used in the exercise into a models subdirectory in the /home/workspace directory, so the path used can be a little bit shorter.

python app.py -i "images/blue-car.jpg" -t "CAR\_META" -m "/home/workspace/models/vehicle-attributes-recognition-barrier-0039.xml" -c "/opt/intel/openvino/deployment\_tools/inference\_engine/lib/intel64/libcpu\_extension\_sse4.so"

For the other models, make sure to update the input image -i, model type -t, and model -m accordingly.

### Pose Estimation Output Handling

Handling the car output was fairly straightforward by using np.argmax, but the outputs for the pose estimation and text detection models is a bit trickier. However, there's a lot of similar code between the two. In this second part of the solution, I'll go into detail on the pose estimation model, and then we'll finish with a quick video on handling the output of the text detection model.

**https://www.youtube.com/watch?v=fvFuMgYUibs**

Pose Estimation is more difficult, and doesn't have as nicely named outputs. I noted you just need the second one in this exercise, called 'Mconv7\_stage2\_L2', which is just the keypoint heatmaps, and not the associations between these keypoints. From there, I created an empty array to hold the output heatmaps once they are re-sized, as I decided to iterate through each heatmap 1 by 1 and re-size it, which can't be done in place on the original output.

**def** **handle\_pose**(output, input\_shape):

'''

Handles the output of the Pose Estimation model.

Returns ONLY the keypoint heatmaps, and not the Part Affinity Fields.

'''

*# TODO 1: Extract only the second blob output (keypoint heatmaps)*

heatmaps = output['Mconv7\_stage2\_L2']

*# TODO 2: Resize the heatmap back to the size of the input*

*# Create an empty array to handle the output map*

out\_heatmap = np.zeros([heatmaps.shape[1], input\_shape[0], input\_shape[1]])

*# Iterate through and re-size each heatmap*

**for** h **in** range(len(heatmaps[0])):

out\_heatmap[h] = cv2.resize(heatmaps[0][h], input\_shape[0:2][::-1])

**return** out\_heatmap

Note that the input\_shape[0:2][::-1] line is taking the original image shape of HxWxC, taking just the first two (HxW), and reversing them to be WxH as cv2.resize uses.

### Text Detection Model Handling

Thanks for sticking in there! The code for the text detection model is pretty similar to the pose estimation one, so let's finish things off.

**https://www.youtube.com/watch?v=0tuVuyD6ffM**

Text Detection had a very similar output processing function, just using the 'model/segm\_logits/add' output and only needing to resize over two "channels" of output. I likely could have extracted this out into its own output handling function that both Pose Estimation and Text Detection could have used.

**def** **handle\_text**(output, input\_shape):

'''

Handles the output of the Text Detection model.

Returns ONLY the text/no text classification of each pixel,

and not the linkage between pixels and their neighbors.

'''

*# TODO 1: Extract only the first blob output (text/no text classification)*

text\_classes = output['model/segm\_logits/add']

*# TODO 2: Resize this output back to the size of the input*

out\_text = np.empty([text\_classes.shape[1], input\_shape[0], input\_shape[1]])

**for** t **in** range(len(text\_classes[0])):

out\_text[t] = cv2.resize(text\_classes[0][t], input\_shape[0:2][::-1])

**return** out\_text

…………

**Recap**

**https://www.youtube.com/watch?v=o-fWs0BwbyM**

In this lesson we covered:

* Basics of the Intel® Distribution of OpenVINO™ Toolkit
* Different Computer Vision model types
* Available Pre-Trained Models in the Software
* Choosing the right Pre-Trained Model for your App
* Loading and Deploying a Basic App with a Pre-Trained Model

………………..

**Lesson Glossary**

#### Edge Application

Applications with inference run on local hardware, sometimes without network connections, such as Internet of Things (IoT) devices, as opposed to the cloud. Less data needs to be streamed over a network connection, and real-time decisions can be made.

#### OpenVINO™ Toolkit

The [**Intel® Distribution of OpenVINO™ Toolkit**](https://software.intel.com/en-us/openvino-toolkit) enables deep learning inference at the edge by including both neural network optimizations for inference as well as hardware-based optimizations for Intel® hardware.

#### Pre-Trained Model

Computer Vision and/or AI models that are already trained on large datasets and available for use in your own applications. These models are often trained on datasets like [**ImageNet**](https://en.wikipedia.org/wiki/ImageNet). Pre-trained models can either be used as is or used in transfer learning to further fine-tune a model. The OpenVINO™ Toolkit provides a number of [**pre-trained models**](https://software.intel.com/en-us/openvino-toolkit/documentation/pretrained-models) that are already optimized for inference.

#### Transfer Learning

The use of a pre-trained model as a basis for further training of a neural network. Using a pre-trained model can help speed up training as the early layers of the network have feature extractors that work in a wide variety of applications, and often only late layers will need further fine-tuning for your own dataset. OpenVINO™ does not deal with transfer learning, as all training should occur prior to using the Model Optimizer.

#### Image Classification

A form of inference in which an object in an image is determined to be of a particular class, such as a cat vs. a dog.

#### Object Detection

A form of inference in which objects within an image are detected, and a bounding box is output based on where in the image the object was detected. Usually, this is combined with some form of classification to also output which class the detected object belongs to.

#### Semantic Segmentation

A form of inference in which objects within an image are detected and classified on a pixel-by-pixel basis, with all objects of a given class given the same label.

#### Instance Segmentation

Similar to semantic segmentation, this form of inference is done on a pixel-by-pixel basis, but different objects of the same class are separately identified.

#### [SSD](https://arxiv.org/abs/1512.02325)

A neural network combining object detection and classification, with different feature extraction layers directly feeding to the detection layer, using default bounding box sizes and shapes/

#### [YOLO](https://arxiv.org/abs/1506.02640)

One of the original neural networks to only take a single look at an input image, whereas earlier networks ran a classifier multiple times across a single image at different locations and scales.

#### [Faster R-CNN](https://arxiv.org/abs/1506.01497)

A network, expanding on [**R-CNN**](https://arxiv.org/pdf/1311.2524.pdf) and [**Fast R-CNN**](https://arxiv.org/pdf/1504.08083.pdf), that integrates advances made in the earlier models by adding a Region Proposal Network on top of the Fast R-CNN model for an integrated object detection model.

#### [MobileNet](https://arxiv.org/abs/1704.04861)

A neural network architecture optimized for speed and size with minimal loss of inference accuracy through the use of techniques like [**1x1 convolutions**](https://stats.stackexchange.com/questions/194142/what-does-1x1-convolution-mean-in-a-neural-network). As such, MobileNet is more useful in mobile applications that substantially larger and slower networks.

#### [ResNet](https://arxiv.org/abs/1512.03385)

A very deep neural network that made use of residual, or “skip” layers that pass information forward by a couple of layers. This helped deal with the [**vanishing gradient problem**](https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484) experienced by deeper neural networks.

#### [Inception](https://arxiv.org/pdf/1409.4842.pdf)

A neural network making use of multiple different convolutions at each “layer” of the network, such as 1x1, 3x3 and 5x5 convolutions. The top architecture from the original paper is also known as GoogLeNet, an homage to **[LeNet](http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf" \t "_blank)**, an early neural network used for character recognition.

#### Inference Precision

Precision refers to the level of detail to weights and biases in a neural network, whether in floating point precision or integer precision. Lower precision leads to lower accuracy, but with a positive trade-off for network speed and size.

================================

##### LESSON 3

#### The Model Optimizer

Explore the Model Optimizer, which allows you to take models trained with many different Deep Learning frameworks and create an Intermediate Representation useful with the Inference Engine.

**3 hours**

……………..

# Introduction

https://www.youtube.com/watch?v=wxrPEdI9Sj4

In this lesson we'll cover:

* Basics of the Model Optimizer
* Different Optimization Techniques and their impact on model performance
* Supported Frameworks in the Intel® Distribution of OpenVINO™ Toolkit
* Converting from models in those frameworks to Intermediate Representations
* And a bit on Custom Layers

………….

# The Model Optimizer

https://www.youtube.com/watch?v=Gyj7VPmGOVM

The Model Optimizer helps convert models in multiple different frameworks to an Intermediate Representation, which is used with the Inference Engine. If a model is not one of the pre-converted models in the Pre-Trained Models OpenVINO™ provides, it is a required step to move onto the Inference Engine.

As part of the process, it can perform various optimizations that can help shrink the model size and help make it faster, although this will not give the model higher inference accuracy. In fact, there will be some loss of accuracy as a result of potential changes like lower precision. However, these losses in accuracy are minimized.

### Local Configuration

Configuring the Model Optimizer is pretty straight forward for your local machine, given that you already have OpenVINO™ installed. You can navigate to your OpenVINO™ install directory first, which is usually /opt/intel/openvino. Then, head to /deployment\_tools/model\_optimizer/install\_prerequisites, and run the install\_prerequisites.sh script therein.

### QUIZ QUESTION

Which of the following best describe the Model Optimizer?

* 

It improves the accuracy of a model for inference.

* It converts a model for use with the Inference Engine, including improvements to size and speed.
* 

It performs the actual inference given a model.

* 

It selects the best hyperparameters for training.

SUBMIT

### Developer Documentation

You can find the developer documentation [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html) for working with the Model Optimizer. We’ll delve deeper into it throughout the lesson.

…………….

# Optimization Techniques

https://www.youtube.com/watch?v=FKgkB4lSn-U

Here, I mostly focused on three optimization techniques: quantization, freezing and fusion. Note that at the end of the video when I mention hardware optimizations, those are done by the Inference Engine (which we’ll cover in the next lesson), not the Model Optimizer.

### Quantization

Quantization is related to the topic of precision I mentioned before, or how many bits are used to represent the weights and biases of the model. During training, having these very accurate numbers can be helpful, but it’s often the case in inference that the precision can be reduced without substantial loss of accuracy. Quantization is the process of reducing the precision of a model.

With the OpenVINO™ Toolkit, models usually default to FP32, or 32-bit floating point values, while FP16 and INT8, for 16-bit floating point and 8-bit integer values, are also available (INT8 is only currently available in the Pre-Trained Models; the Model Optimizer does not currently support that level of precision). FP16 and INT8 will lose some accuracy, but the model will be smaller in memory and compute times faster. Therefore, quantization is a common method used for running models at the edge.

### Freezing

Freezing in this context is used for TensorFlow models. Freezing TensorFlow models will remove certain operations and metadata only needed for training, such as those related to backpropagation. Freezing a TensorFlow model is usually a good idea whether before performing direct inference or converting with the Model Optimizer.

### Fusion

Fusion relates to combining multiple layer operations into a single operation. For example, a batch normalization layer, activation layer, and convolutional layer could be combined into a single operation. This can be particularly useful for GPU inference, where the separate operations may occur on separate GPU kernels, while a fused operation occurs on one kernel, thereby incurring less overhead in switching from one kernel to the next.

### QUIZ QUESTION

Match the types of optimization for inference below to their descriptions.

### *Submit to check your answer choices!*

**DESCRIPTION**

**OPTIMIZATION TYPE**

Reduces precision of weights and biases, thereby reducing compute time and size with some loss of accuracy.

Quantization

On a layer basis is used for fine-tuning a neural network; in TensorFlow this removes metadata only needed for training.

Freezing

Combining certain operations together into one operation and needing less computational overhead.

Fusion

SUBMIT

Nice job! The Model Optimizer converts a model to an Intermediate Representation for use with the Inference Engine. This process includes potential improvements related to model size and speed, with potential trade-offs with accuracy.

### Further Research

* If you’d like to learn more about quantization, check out this [helpful post](https://nervanasystems.github.io/distiller/quantization.html).
* You can find out more about optimizations performed by the Model Optimizer in the OpenVINO™ Toolkit [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_Model_Optimization_Techniques.html).

Nice work! The Model Optimizer is performing some of these behind the scenes, so we can focus more on building out our application.

……….

# Supported Frameworks

https://www.youtube.com/watch?v=5tnzgZuuVsk

The supported frameworks with the OpenVINO™ Toolkit are:

* Caffe
* TensorFlow
* MXNet
* ONNX (which can support PyTorch and Apple ML models through another conversion step)
* Kaldi

These are all open source, just like the OpenVINO™ Toolkit. Caffe is originally from UC Berkeley, TensorFlow is from Google Brain, MXNet is from Apache Software, ONNX is combined effort of Facebook and Microsoft, and Kaldi was originally an individual’s effort. Most of these are fairly multi-purpose frameworks, while Kaldi is primarily focused on speech recognition data.

There are some differences in how exactly to handle these, although most differences are handled under the hood of the OpenVINO™ Toolkit. For example, TensorFlow has some different steps for certain models, or frozen vs. unfrozen models. However, most of the functionality is shared across all of the supported frameworks.

### QUIZ QUESTION

Which of these frameworks are supported by the OpenVINO™ Toolkit?

* **Caffe**
* **TensorFlow**
* **MXNet**
* **ONNX**
* **Kaldi**

SUBMIT

### Further Research

In case you aren’t familiar with any of these frameworks, feel free to check out the sites for each below:

* [Caffe](https://caffe.berkeleyvision.org/)
* [TensorFlow](https://www.tensorflow.org/)
* [MXNet](https://mxnet.apache.org/)
* [ONNX](https://onnx.ai/)
* [Kaldi](https://kaldi-asr.org/doc/dnn.html)

…………………..

# Intermediate Representations

https://www.youtube.com/watch?v=EImZN2nk1t4

Intermediate Representations (IRs) are the OpenVINO™ Toolkit’s standard structure and naming for neural network architectures. A Conv2D layer in TensorFlow, Convolution layer in Caffe, or Conv layer in ONNX are all converted into a Convolution layer in an IR.

The IR is able to be loaded directly into the Inference Engine, and is actually made of two output files from the Model Optimizer: an XML file and a binary file. The XML file holds the model architecture and other important metadata, while the binary file holds weights and biases in a binary format. You need both of these files in order to run inference Any desired optimizations will have occurred while this is generated by the Model Optimizer, such as changes to precision. You can generate certain precisions with the --data\_type argument, which is usually FP32 by default.

### QUIZ QUESTION

The Intermediate Representation is a model where specific layers of supported deep learning frameworks are replaced with layers in the “dialect” of the Inference Engine.

* **True**
* 

False

SUBMIT

Nice work! The Model Optimizer works almost like a translator here, making the Intermediate Representation a shared dialect of all the supported frameworks, which can be understood by the Inference Engine.

### Further Research

* You can find the main developer documentation on converting models in the OpenVINO™ Toolkit [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_convert_model_Converting_Model.html). We’ll cover how to do so with TensorFlow, Caffe and ONNX (useful for PyTorch) over the next several pages.
* You can find the documentation on different layer names when converted to an IR [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_Supported_Frameworks_Layers.html).
* Finally, you can find more in-depth data on each of the Intermediate Representation layers themselves [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_convert_model_IRLayersCatalogSpec.html).

…………………….

# Using the Model Optimizer with TensorFlow Models

https://www.youtube.com/watch?v=IcZKqDTsp4c

Once the Model Optimizer is configured, the next thing to do with a TensorFlow model is to determine whether to use a frozen or unfrozen model. You can either freeze your model, which I would suggest, or use the separate instructions in the documentation to convert a non-frozen model. Some models in TensorFlow may already be frozen for you, so you can skip this step.

From there, you can feed the model into the Model Optimizer, and get your Intermediate Representation. However, there may be a few items specific to TensorFlow for that stage, which you’ll need to feed into the Model Optimizer before it can create an IR for use with the Inference Engine.

TensorFlow models can vary for what additional steps are needed by model type, being unfrozen or frozen, or being from the TensorFlow Detection Model Zoo. Unfrozen models usually need the --mean\_values and --scale parameters fed to the Model Optimizer, while the frozen models from the Object Detection Model Zoo don’t need those parameters. However, the frozen models will need TensorFlow-specific parameters like --tensorflow\_use\_custom\_operations\_config and --tensorflow\_object\_detection\_api\_pipeline\_config. Also, --reverse\_input\_channels is usually needed, as TF model zoo models are trained on RGB images, while OpenCV usually loads as BGR. Certain models, like YOLO, DeepSpeech, and more, have their own separate pages.

### TensorFlow Object Detection Model Zoo

The models in the TensorFlow Object Detection Model Zoo can be used to even further extend the pre-trained models available to you. These are in TensorFlow format, so they will need to be fed to the Model Optimizer to get an IR. The models are just focused on object detection with bounding boxes, but there are plenty of different model architectures available.

### Further Research

* The developer documentation for Converting TensorFlow Models can be found [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html). You’ll work through this process in the next exercise.
* TensorFlow also has additional models available in the [TensorFlow Detection Model Zoo](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md). By converting these over to Intermediate Representations, you can expand even further on the pre-trained models available to you.

........

# Solution: Convert a TensorFlow Model

https://www.youtube.com/watch?v=\_E2EP3ZM92s

Here's what I entered to convert the SSD MobileNet V2 model from TensorFlow:

python /opt/intel/openvino/deployment\_tools/model\_optimizer/mo.py --input\_model frozen\_inference\_graph.pb --tensorflow\_object\_detection\_api\_pipeline\_config pipeline.config --reverse\_input\_channels --tensorflow\_use\_custom\_operations\_config /opt/intel/openvino/deployment\_tools/model\_optimizer/extensions/front/tf/ssd\_v2\_support.json

This is pretty long! I would suggest considering setting a [path environment variable](https://help.ubuntu.com/community/EnvironmentVariables) for the Model Optimizer if you are working locally on a Linux-based machine. You could do something like this:

export MOD\_OPT=/opt/intel/openvino/deployment\_tools/model\_optimizer

And then when you need to use it, you can utilize it with $MOD\_OPT/mo.py instead of entering the full long path each time. In this case, that would also help shorten the path to the ssd\_v2\_support.json

……….

# Using the Model Optimizer with Caffe Models

https://www.youtube.com/watch?v=nUvJ7aXeyMg

The process for converting a Caffe model is fairly similar to the TensorFlow one, although there’s nothing about freezing the model this time around, since that’s a TensorFlow concept. Caffe does have some differences in the set of supported model architectures. Additionally, Caffe models need to feed both the .caffemodel file, as well as a .prototxt file, into the Model Optimizer. If they have the same name, only the model needs to be directly input as an argument, while if the .prototxt file has a different name than the model, it should be fed in with --input\_proto as well.

### Further Research

The developer documentation for Converting Caffe Models can be found [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_Caffe.html). You’ll work through this process in the next exercise.

……..

# Solution: Convert a Caffe Model

https://www.youtube.com/watch?v=kfb7BGByUdM

Here's what I entered to convert the Squeezenet V1.1 model from Caffe:

python /opt/intel/openvino/deployment\_tools/model\_optimizer/mo.py --input\_model squeezenet\_v1.1.caffemodel --input\_proto deploy.prototxt

…………

# Solution: Convert an ONNX Model

https://www.youtube.com/watch?v=KuERfGMco1A

Here's what I entered to convert the AlexNet model from ONNX:

python /opt/intel/openvino/deployment\_tools/model\_optimizer/mo.py --input\_model model.onnx

…………

# Cutting Parts of a Model

https://www.youtube.com/watch?v=jhZIkgysCWM

Cutting a model is mostly applicable for TensorFlow models. As we saw earlier in converting these models, they sometimes have some extra complexities. Some common reasons for cutting are:

* The model has pre- or post-processing parts that don’t translate to existing Inference Engine layers.
* The model has a training part that is convenient to be kept in the model, but is not used during inference.
* The model is too complex with many unsupported operations, so the complete model cannot be converted in one shot.
* The model is one of the supported SSD models. In this case, you need to cut a post-processing part off.
* There could be a problem with model conversion in the Model Optimizer or with inference in the Inference Engine. To localize the issue, cutting the model could help to find the problem

There’s two main command line arguments to use for cutting a model with the Model Optimizer, named intuitively as --input and --output, where they are used to feed in the layer names that should be either the new entry or exit points of the model.

### Developer Documentation

You guessed it - [here’s](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_convert_model_Cutting_Model.html) the developer documentation for cutting a model.

………….

# Supported Layers

https://www.youtube.com/watch?v=bwVlEaAsHvk

Earlier, we saw some of the supported layers when looking at the names when converting from a supported framework to an IR. While that list is useful for one-offs, you probably don’t want to check whether each and every layer in your model is supported. You can also just see when you run the Model Optimizer what will convert.

What happens when a layer isn’t supported by the Model Optimizer? One potential solution is the use of custom layers, which we’ll go into more shortly. Another solution is actually running the given unsupported layer in its original framework. For example, you could potentially use TensorFlow to load and process the inputs and outputs for a specific layer you built in that framework, if it isn’t supported with the Model Optimizer. Lastly, there are also unsupported layers for certain hardware, that you may run into when working with the Inference Engine. In this case, there are sometimes extensions available that can add support. We’ll discuss that approach more in the next lesson.

### QUIZ QUESTION

**Any** layer created in one of the supported frameworks is able to be directly converted by the Model Optimizer.

* 

True

* False

SUBMIT

Correct! While just about every layer you’d likely be using in your own neural network is supported with the Model Optimizer, sometimes you’ll need to make use of Custom Layers, which we’ll cover next.

### Supported Layers List

Check out the full list of supported layers [here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_Supported_Frameworks_Layers.html).

……………..

# Custom Layers

https://www.youtube.com/watch?v=ciSpU0kyjH8

Custom layers are a necessary and important to have feature of the OpenVINO™ Toolkit, although you shouldn’t have to use it very often, if at all, due to all of the supported layers. However, it’s useful to know a little about its existence and how to use it if the need arises.

The [list of supported layers](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_Supported_Frameworks_Layers.html) from earlier very directly relates to whether a given layer is a custom layer. Any layer not in that list is automatically classified as a custom layer by the Model Optimizer.

To actually add custom layers, there are a few differences depending on the original model framework. In both TensorFlow and Caffe, the first option is to register the custom layers as extensions to the Model Optimizer.

For Caffe, the second option is to register the layers as Custom, then use Caffe to calculate the output shape of the layer. You’ll need Caffe on your system to do this option.

For TensorFlow, its second option is to actually replace the unsupported subgraph with a different subgraph. The final TensorFlow option is to actually offload the computation of the subgraph back to TensorFlow during inference.

You’ll get a chance to practice this in the next exercise. Again, as this is an advanced topic, we won’t delve too much deeper here, but feel free to check out the linked documentation if you want to know more.

### Further Research

You’ll get a chance to get hands on with Custom Layers next, but feel free to check out the [developer documentation](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_customize_model_optimizer_Customize_Model_Optimizer.html) in the meantime.

If you’re interested in the option to use TensorFlow to operate on a given unsupported layer, you should also make sure to read the [documentation here](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_customize_model_optimizer_Offloading_Sub_Graph_Inference.html).

…………..

# Recap

https://www.youtube.com/watch?v=Zf5zOMg2PjQ

In this lesson we covered:

* Basics of the Model Optimizer
* Different Optimization Techniques and their impact on model performance
* Supported Frameworks in the Intel® Distribution of OpenVINO™ Toolkit
* Converting from models in those frameworks to Intermediate Representations
* And a bit on Custom Layers

………

# Lesson Glossary

#### Model Optimizer

A command-line tool used for converting a model from one of the supported frameworks to an Intermediate Representation (IR), including certain performance optimizations, that is compatible with the Inference Engine.

#### Optimization Techniques

Optimization techniques adjust the original trained model in order to either reduce the size of or increase the speed of a model in performing inference. Techniques discussed in the lesson include quantization, freezing and fusion.

#### Quantization

Reduces precision of weights and biases (to lower precision floating point values or integers), thereby reducing compute time and size with some (often minimal) loss of accuracy.

#### Freezing

In TensorFlow this removes metadata only needed for training, as well as converting variables to constants. Also a term in training neural networks, where it often refers to freezing layers themselves in order to fine tune only a subset of layers.

#### Fusion

The process of combining certain operations together into one operation and thereby needing less computational overhead. For example, a batch normalization layer, activation layer, and convolutional layer could be combined into a single operation. This can be particularly useful for GPU inference, where the separate operations may occur on separate GPU kernels, while a fused operation occurs on one kernel, thereby incurring less overhead in switching from one kernel to the next.

#### Supported Frameworks

The Intel® Distribution of OpenVINO™ Toolkit currently supports models from five frameworks (which themselves may support additional model frameworks): Caffe, TensorFlow, MXNet, ONNX, and Kaldi.

#### [Caffe](https://caffe.berkeleyvision.org/)

The “Convolutional Architecture for Fast Feature Embedding” (CAFFE) framework is an open-source deep learning library originally built at UC Berkeley.

#### [TensorFlow](https://www.tensorflow.org/)

TensorFlow is an open-source deep learning library originally built at Google. As an Easter egg for anyone who has read this far into the glossary, this was also your instructor’s first deep learning framework they learned, back in 2016 (pre-V1!).

#### [MXNet](https://mxnet.apache.org/)

Apache MXNet is an open-source deep learning library built by Apache Software Foundation.

#### [ONNX](https://onnx.ai/)

The “Open Neural Network Exchange” (ONNX) framework is an open-source deep learning library originally built by Facebook and Microsoft. PyTorch and Apple-ML models are able to be converted to ONNX models.

#### [Kaldi](https://kaldi-asr.org/doc/dnn.html)

While still open-source like the other supported frameworks, Kaldi is mostly focused around speech recognition data, with the others being more generalized frameworks.

#### Intermediate Representation

A set of files converted from one of the supported frameworks, or available as one of the Pre-Trained Models. This has been optimized for inference through the Inference Engine, and may be at one of several different precision levels. Made of two files:

* .xml - Describes the network topology
* .bin - Contains the weights and biases in a binary file

#### Supported Layers

Layers [**supported**](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_Supported_Frameworks_Layers.html) for direct conversion from supported framework layers to intermediate representation layers through the Model Optimizer. While nearly every layer you will ever use is in the supported frameworks is supported, there is sometimes a need for handling Custom Layers.

#### Custom Layers

Custom layers are those outside of the list of known, supported layers, and are typically a rare exception. Handling custom layers in a neural network for use with the Model Optimizer depends somewhat on the framework used; other than adding the custom layer as an extension, you otherwise have to follow [**instructions**](https://docs.openvinotoolkit.org/2019_R3/_docs_MO_DG_prepare_model_customize_model_optimizer_Customize_Model_Optimizer.html) specific to the framework.

##### LESSON 4

#### The Inference Engine

Dive deeper into the Inference Engine, and perform inference in the OpenVINO™ Toolkit. By the end, you’ll know the full workflow for OpenVINO™ fundamentals and be ready to integrate into an app.

**3 hours left**

…………

# Introduction

https://www.youtube.com/watch?v=BUpkwGhboLg

In this lesson we'll cover:

* Basics of the Inference Engine
* Supported Devices
* Feeding an Intermediate Representation to the Inference Engine
* Making Inference Requests
* Handling Results from the Inference Engine
* Integrating the Inference Model into an App

……………………

# The Inference Engine

https://www.youtube.com/watch?v=dZA4QGbDrs4

The Inference Engine runs the actual inference on a model. It only works with the Intermediate Representations that come from the Model Optimizer, or the Intel® Pre-Trained Models in OpenVINO™ that are already in IR format.

Where the Model Optimizer made some improvements to size and complexity of the models to improve memory and computation times, the Inference Engine provides hardware-based optimizations to get even further improvements from a model. This really empowers your application to run at the edge and use up as little of device resources as possible.

The Inference Engine has a straightforward API to allow easy integration into your edge application. The Inference Engine itself is actually built in C++ (at least for the CPU version), leading to overall faster operations; however, it is very common to utilize the built-in Python wrapper to interact with it in Python code.

### QUIZ QUESTION

Which of the following best describe the Inference Engine?

* 

It improves the accuracy of a model for inference.

* 

It converts a model for use with the Model Optimizer, including improvements to size and speed.

* **It provides a library of computer vision functions and performs the inference on a model. (Correct)**
* 

It selects the best hyperparameters for training.

SUBMIT

Great work! The Inference Engine, as the name suggests, does the real legwork of inference at the edge.

### Developer Documentation

You can find the developer documentation [**here**](https://docs.openvinotoolkit.org/2019_R3/_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html) for working with the Inference Engine. We’ll delve deeper into it throughout the lesson.

…………..

# Supported Devices

https://www.youtube.com/watch?v=m2d1urdJegA

The supported devices for the Inference Engine are all Intel® hardware, and are a variety of such devices: CPUs, including integrated graphics processors, GPUs, FPGAs, and VPUs. You likely know what CPUs and GPUs are already, but maybe not the others.

FPGAs, or Field Programmable Gate Arrays, are able to be further configured by a customer after manufacturing. Hence the “field programmable” part of the name.

VPUs, or Vision Processing Units, are going to be like the Intel® Neural Compute Stick. They are small, but powerful devices that can be plugged into other hardware, for the specific purpose of accelerating computer vision tasks.

### Differences Among Hardware

Mostly, how the Inference Engine operates on one device will be the same as other supported devices; however, you may remember me mentioning a CPU extension in the last lesson. That’s one difference, that a CPU extension can be added to support additional layers when the Inference Engine is used on a CPU.

There are also some differences among supported layers by device, which is linked to at the bottom of this page. Another important one to note is regarding when you use an Intel® Neural Compute Stick (NCS). An easy, fairly low-cost method of testing out an edge app locally, outside of your own computer is to use the NCS2 with a Raspberry Pi. The Model Optimizer is not supported directly with this combination, so you may need to create an Intermediate Representation on another system first, although there are [**some instructions**](https://software.intel.com/en-us/articles/model-downloader-optimizer-for-openvino-on-raspberry-pi) for one way to do so on-device. The Inference Engine itself is still supported with this combination.

### QUIZ QUESTION

Which of the following Intel® hardware devices are supported for optimal performance with the OpenVINO™ Toolkit’s Inference Engine?

* CPUs
* GPUs
* VPUs (such as the Neural Compute Stick)
* FPGAs

SUBMIT

### Further Research

Depending on your device, the different plugins do have some differences in functionality and optimal configurations. You can read more on Supported Devices [**here**](https://docs.openvinotoolkit.org/2019_R3/_docs_IE_DG_supported_plugins_Supported_Devices.html).

There is a wide variety of Intel® hardware available to use with the Inference Engine! There are some additional considerations around using features like Shape Inference with OpenVINO™ that do depend on the plugins used with different hardware, which you can read about through the link below the quiz.

………………

# Using the Inference Engine with an IR

### https://www.youtube.com/watch?v=b90ny0AmQF8

### IECore and IENetwork

To load an IR into the Inference Engine, you’ll mostly work with two classes in the openvino.inference\_engine library (if using Python):

* IECore, which is the Python wrapper to work with the Inference Engine
* IENetwork, which is what will initially hold the network and get loaded into IECore

The next step after importing is to set a couple variables to actually use the IECore and IENetwork. In the **[IECore documentation](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IECore.html" \t "_blank)**, no arguments are needed to initialize. To use **[IENetwork](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IENetwork.html" \t "_blank)**, you need to load arguments named model and weights to initialize - the XML and Binary files that make up the model’s Intermediate Representation.

### Check Supported Layers

In the **[IECore documentation](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IECore.html" \t "_blank)**, there was another function called query\_network, which takes in an IENetwork as an argument and a device name, and returns a list of layers the Inference Engine supports. You can then iterate through the layers in the IENetwork you created, and check whether they are in the supported layers list. If a layer was not supported, a CPU extension may be able to help.

The device\_name argument is just a string for which device is being used - ”CPU”, ”GPU”, ”FPGA”, or ”MYRIAD” (which applies for the Neural Compute Stick).

### CPU extension

If layers were successfully built into an Intermediate Representation with the Model Optimizer, some may still be unsupported by default with the Inference Engine when run on a CPU. However, there is likely support for them using one of the available CPU extensions.

These do differ by operating system a bit, although they should still be in the same overall location. If you navigate to your OpenVINO™ install directory, then deployment\_tools, inference\_engine, lib, intel64:

* On Linux, you’ll see a few CPU extension files available for AVX and SSE. That’s a bit outside of the scope of the course, but look up Advanced Vector Extensions if you want to know more there. In the classroom workspace, the SSE file will work fine.
  + Intel® Atom processors use SSE4, while Intel® Core processors will utilize AVX.
  + This is especially important to make note of when transferring a program from a Core-based laptop to an Atom-based edge device. If the incorrect extension is specified in the application, the program will crash.
  + AVX systems can run SSE4 libraries, but not vice-versa.
* On Mac, there’s just a single CPU extension file.

You can add these directly to the IECore using their full path. After you’ve added the CPU extension, if necessary, you should re-check that all layers are now supported. If they are, it’s finally time to load the model into the IECore.

### Further Research

As you get more into working with the Inference Engine in the next exercise and into the future, here are a few pages of documentation I found useful in working with it.

* [**IE Python API**](https://docs.openvinotoolkit.org/2019_R3/ie_python_api.html)
* [**IE Network**](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IENetwork.html)
* [**IE Core**](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IECore.html)

………………..

# Solution: Feed an IR to the Inference Engine

https://www.youtube.com/watch?v=jEmebNVBlc4

First, add the additional libraries (os may not be needed depending on how you get the model file names):

*### Load the necessary libraries*

**import** os

**from** openvino.inference\_engine **import** IENetwork, IECore

Then, to load the Intermediate Representation and feed it to the Inference Engine:

**def** **load\_to\_IE**(model\_xml):

*### Load the Inference Engine API*

plugin = IECore()

*### Load IR files into their related class*

model\_bin = os.path.splitext(model\_xml)[0] + ".bin"

net = IENetwork(model=model\_xml, weights=model\_bin)

*### Add a CPU extension, if applicable.*

plugin.add\_extension(CPU\_EXTENSION, "CPU")

*### Get the supported layers of the network*

supported\_layers = plugin.query\_network(network=net, device\_name="CPU")

*### Check for any unsupported layers, and let the user*

*### know if anything is missing. Exit the program, if so.*

unsupported\_layers = [l **for** l **in** net.layers.keys() **if** l **not** **in** supported\_layers]

**if** len(unsupported\_layers) != 0:

print("Unsupported layers found: {}".format(unsupported\_layers))

print("Check whether extensions are available to add to IECore.")

exit(1)

*### Load the network into the Inference Engine*

plugin.load\_network(net, "CPU")

print("IR successfully loaded into Inference Engine.")

**return**

Note that a more optimal approach here would actually check whether a CPU extension was added as an argument by the user, but to keep things simple, I hard-coded it for the exercise.

### Running Your Implementation

You should make sure your implementation runs with all three pre-trained models we worked with earlier (and you are welcome to also try the models you converted in the previous lesson from TensorFlow, Caffe and ONNX, although your workspace may not have these stored). I placed these in the /home/workspace/models directory for easier use, and because the workspace will reset the /opt directory between sessions.

python feed\_network.py -m /home/workspace/models/human-pose-estimation-0001.xml

You can run the other two by updating the model name in the above.

………….

# Sending Inference Requests to the IE

https://www.youtube.com/watch?v=wLN8HYZ05rg

After you load the IENetwork into the IECore, you get back an ExecutableNetwork, which is what you will send inference requests to. There are two types of inference requests you can make: Synchronous and Asynchronous. There is an important difference between the two on whether your app sits and waits for the inference or can continue and do other tasks.

With an ExecutableNetwork, synchronous requests just use the infer function, while asynchronous requests begin with start\_async, and then you can wait until the request is complete. These requests are InferRequest objects, which will hold both the input and output of the request.

We'll look a little deeper into the difference between synchronous and asynchronous on the next page.

### Further Research

* [**Executable Network documentation**](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1ExecutableNetwork.html)
* [**Infer Request documentation**](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1InferRequest.html)

……………

# Asynchronous Requests

### https://www.youtube.com/watch?v=JGuUIDpn1PY

### Synchronous

Synchronous requests will wait and do nothing else until the inference response is returned, blocking the main thread. In this case, only one frame is being processed at once, and the next frame cannot be gathered until the current frame’s inference request is complete.

### Asynchronous

You may have heard of asynchronous if you do front-end or networking work. In that case, you want to process things asynchronously, so in case the response for a particular item takes a long time, you don’t hold up the rest of your website or app from loading or operating appropriately.

Asynchronous, in our case, means other tasks may continue while waiting on the IE to respond. This is helpful when you want other things to still occur, so that the app is not completely frozen by the request if the response hangs for a bit.

Where the main thread was blocked in synchronous, asynchronous does not block the main thread. So, you could have a frame sent for inference, while still gathering and pre-processing the next frame. You can make use of the "wait" process to wait for the inference result to be available.

You could also use this with multiple webcams, so that the app could "grab" a new frame from one webcam while performing inference for the other.

**QUIZ QUESTION**

In the below examples, would an asynchronous or synchronous call make more sense?

*Submit to check your answer choices!*

**EXAMPLE**

**SYNC VS. ASYNC**

A network call is made to a server with an unknown latency for returning a response, and the user is otherwise able to use the app while waiting on a response.

Asynchronous

The application needs to wait for a user input before being able to process additional data.

Synchronous

SUBMIT

### Great job! This is a simplistic example here, as really even in the synchronous case, you’d probably still want some of the app’s functionality to stick around. But, dealing with network calls to servers and APIs is a big use of asynchronous.Further Research

* For more on Synchronous vs. Asynchronous, check out this [**helpful post**](https://whatis.techtarget.com/definition/synchronous-asynchronous-API).
* You can also check out the [**documentation**](https://docs.openvinotoolkit.org/2019_R3/_docs_IE_DG_Integrate_with_customer_application_new_API.html) on integrating the inference engine into an application to see the different functions calls from an Inference Request for sync (Infer) vs. async (StartAsync).
* Lastly, for further practice with Asynchronous Inference Requests, you can check out [**this useful demo**](https://github.com/opencv/open_model_zoo/blob/master/demos/object_detection_demo_ssd_async/README.md). You’ll get a chance to practice with Synchronous and Asynchronous Requests in the upcoming exercise.

……………

# Solution: Inference Requests

#### https://www.youtube.com/watch?v=QeBpEkkoZ74

#### Synchronous Solution

**def** **sync\_inference**(exec\_net, input\_blob, image):

'''

Performs synchronous inference

Return the result of inference

'''

result = exec\_net.infer({input\_blob: image})

**return** result

#### Asynchronous Solution

**def** **async\_inference**(exec\_net, input\_blob, image):

'''

Performs asynchronous inference

Returns the `exec\_net`

'''

exec\_net.start\_async(request\_id=0, inputs={input\_blob: image})

**while** **True**:

status = exec\_net.requests[0].wait(-1)

**if** status == 0:

**break**

**else**:

time.sleep(1)

**return** exec\_net

I don't actually need time.sleep() here - using the -1 with wait() is able to perform similar functionality.

#### Testing

You can run the test file to check your implementations using inference on multiple models.

python test.py

…………….

# Handling Results

https://www.youtube.com/watch?v=wO\_Io3wDwTM

You saw at the end of the previous exercise that the inference requests are stored in a requests attribute in the ExecutableNetwork. There, we focused on the fact that the InferRequest object had a wait function for asynchronous requests.

Each InferRequest also has a few attributes - namely, inputs, outputs and latency. As the names suggest, inputs in our case would be an image frame, outputs contains the results, and latency notes the inference time of the current request, although we won’t worry about that right now.

It may be useful for you to print out exactly what the outputs attribute contains after a request is complete. For now, you can ask it for the data under the “prob” key, or sometimes output\_blob ([**see related documentation**](https://docs.openvinotoolkit.org/2019_R3/classInferenceEngine_1_1Blob.html)), to get an array of the probabilities returned from the inference request.

### QUIZ QUESTION

Which of the following code snippets could be used to extract the output from an inference request, given an ExecutableNetwork named exec\_net?

* 

exec\_net[‘outputs’]

* 

exec\_net.requests[request\_id] (Correct)

* exec\_net.requests[request\_id].outputs[output\_blob]
* 

ExecutableNetwork will not contain the output, a separate object will.

SUBMIT

NEXT

# Thanks for completing that!

Nice work! An ExecutableNetwork contains an InferRequest attribute by the name of requests, and feeding a given request ID key to this attribute will get the specific inference request in question.

From within this InferRequest object, it has an attribute of outputs from which you can use your output\_blob to get the results of that inference request.

…………

# Integrating into Your App

https://www.youtube.com/watch?v=vQpLv1Y3pnU

In the upcoming exercise, you’ll put all your skills together, as well as adding some further customization to your app.

### Further Research

There’s a ton of great potential edge applications out there for you to build. Here are some examples to hopefully get you thinking:

* [**Intel®’s IoT Apps Across Industries**](https://www.intel.com/content/www/us/en/internet-of-things/industry-solutions.html)
* [**Starting Your First IoT Project**](https://hackernoon.com/the-ultimate-guide-to-starting-your-first-iot-project-8b0644fbbe6d)
* [**OpenVINO™ on a Raspberry Pi and Intel® Neural Compute Stick**](https://www.pyimagesearch.com/2019/04/08/openvino-opencv-and-movidius-ncs-on-the-raspberry-pi/)

……………

# Solution: Integrate into an App

**https://www.youtube.com/watch?v=BIdLJkDD5vM**

**Note**: There is one small change from the code on-screen for running on Linux machines versus Mac. On Mac, cv2.VideoWriter uses cv2.VideoWriter\_fourcc('M','J','P','G') to write an .mp4 file, while Linux uses 0x00000021.

### Functions in inference.py

I covered the async and wait functions here as it's split out slightly differently than we saw in the last exercise.

First, it's important to note that output and input blobs were grabbed higher above when the network model is loaded:

self.input\_blob = next(iter(self.network.inputs))

self.output\_blob = next(iter(self.network.outputs))

From there, you can mostly use similar code to before:

**def** **async\_inference**(self, image):

'''

Makes an asynchronous inference request, given an input image.

'''

self.exec\_network.start\_async(request\_id=0,

inputs={self.input\_blob: image})

**return**

**def** **wait**(self):

'''

Checks the status of the inference request.

'''

status = self.exec\_network.requests[0].wait(-1)

**return** status

You can grab the network output using the appropriate request with the output\_blob key:

**def** **extract\_output**(self):

'''

Returns a list of the results for the output layer of the network.

'''

**return** self.exec\_network.requests[0].outputs[self.output\_blob]

### app.py

The next steps in app.py, before customization, are largely based on using the functions in inference.py:

*### Initialize the Inference Engine*

plugin = Network()

*### Load the network model into the IE*

plugin.load\_model(args.m, args.d, CPU\_EXTENSION)

net\_input\_shape = plugin.get\_input\_shape()

...

*### Pre-process the frame*

p\_frame = cv2.resize(frame, (net\_input\_shape[3], net\_input\_shape[2]))

p\_frame = p\_frame.transpose((2,0,1))

p\_frame = p\_frame.reshape(1, \*p\_frame.shape)

*### Perform inference on the frame*

plugin.async\_inference(p\_frame)

*### Get the output of inference*

**if** plugin.wait() == 0:

result = plugin.extract\_output()

*### Update the frame to include detected bounding boxes*

frame = draw\_boxes(frame, result, args, width, height)

*# Write out the frame*

out.write(frame)

The draw\_boxes function is used to extract the bounding boxes and draw them back onto the input image.

**def** **draw\_boxes**(frame, result, args, width, height):

'''

Draw bounding boxes onto the frame.

'''

**for** box **in** result[0][0]: *# Output shape is 1x1x100x7*

conf = box[2]

**if** conf >= 0.5:

xmin = int(box[3] \* width)

ymin = int(box[4] \* height)

xmax = int(box[5] \* width)

ymax = int(box[6] \* height)

cv2.rectangle(frame, (xmin, ymin), (xmax, ymax), (0, 0, 255), 1)

**return** frame

### Customizing app.py

Adding the customization only took a few extra steps.

##### Parsing the command line arguments

First, you need to add the additional command line arguments:

c\_desc = "The color of the bounding boxes to draw; RED, GREEN or BLUE"

ct\_desc = "The confidence threshold to use with the bounding boxes"

...

optional.add\_argument("-c", help=c\_desc, default='BLUE')

optional.add\_argument("-ct", help=ct\_desc, default=0.5)

The names and descriptions here, and even how you use the default values, can be up to you.

##### Handle the new arguments

I needed to also process these arguments a little further. This is pretty open based on your own implementation - since I took in a color string, I need to convert it to a BGR tuple for use as a OpenCV colors.

**def** **convert\_color**(color\_string):

'''

Get the BGR value of the desired bounding box color.

Defaults to Blue if an invalid color is given.

'''

colors = {"BLUE": (255,0,0), "GREEN": (0,255,0), "RED": (0,0,255)}

out\_color = colors.get(color\_string)

**if** out\_color:

**return** out\_color

**else**:

**return** colors['BLUE']

I then need to call this with the related argument, as well as make sure the confidence threshold argument is a float value.

args.c = convert\_color(args.c)

args.ct = float(args.ct)

##### Adding customization to draw\_boxes()

The final step was to integrate these new arguments into my draw\_boxes() function. I needed to make sure that the arguments are fed to the function:

frame = draw\_boxes(frame, result, args, width, height)

and then I can use them where appropriate in the updated function.

**def** **draw\_boxes**(frame, result, args, width, height):

'''

Draw bounding boxes onto the frame.

'''

**for** box **in** result[0][0]: *# Output shape is 1x1x100x7*

conf = box[2]

**if** conf >= args.ct:

xmin = int(box[3] \* width)

ymin = int(box[4] \* height)

xmax = int(box[5] \* width)

ymax = int(box[6] \* height)

cv2.rectangle(frame, (xmin, ymin), (xmax, ymax), args.c, 1)

**return** frame

With everything implemented, I could run my app as such (given I re-used the previously converted TF model from the Model Optimizer lesson) if I wanted blue bounding boxes and a confidence threshold of 0.6:

python app.py -m frozen\_inference\_graph.xml -ct 0.6 -c BLUE

…………..

# Behind the Scenes of Inference Engine

https://www.youtube.com/watch?v=ZWpNQjXSEEc

I noted early on that the Inference Engine is built and optimized in C++, although that’s just the CPU version. There are some differences in what is actually occurring under the hood with the different devices. You are able to work with a shared API to interact with the Inference Engine, while largely being able to ignore these differences.

### Why C++?

Why is the Inference Engine built in C++, at least for CPUs? In fact, many different Computer Vision and AI frameworks are built with C++, and have additional Python interfaces. OpenCV and TensorFlow, for example, are built primarily in C++, but many users interact with the libraries in Python. C++ is faster and more efficient than Python when well implemented, and it also gives the user more direct access to the items in memory and such, and they can be passed between modules more efficiently.

C++ is compiled & optimized ahead of runtime, whereas Python basically gets read line by line when a script is run. On the flip side, Python can make it easier for prototyping and fast fixes. It’s fairly common then to be using a C++ library for the actual Computer Vision techniques and inferencing, but with the application itself in Python, and interacting with the C++ library via a Python API.

### Optimizations by Device

The exact optimizations differ by device with the Inference Engine. While from your end interacting with the Inference Engine is mostly the same, there’s actually separate plugins within for working with each device type.

CPUs, for instance, rely on the Intel® Math Kernel Library for Deep Neural Networks, or MKL-DNN. CPUs also have some extra work to help improve device throughput, especially for CPUs with higher numbers of cores.

GPUs utilize the Compute Library for Deep Neural Networks, or clDNN, which uses OpenCL within. Using OpenCL introduces a small overhead right when the GPU Plugin is loaded, but is only a one-time overhead cost. The GPU Plugin works best with FP16 models over FP32 models

Getting to VPU devices, like the Intel® Neural Compute Stick, there are additional costs associated with it being a USB device. It’s actually recommended to be processing four inference requests at any given time, in order to hide the costs of data transfer from the main device to the VPU.

## Behind the Scenes of Inference Engine Quiz

In your own words, how does the inference engine help with deploying an AI model at the edge?

SUBMIT

### Further Research

The best programming language for machine learning and deep learning is still being debated, but here’s a [**great blog post**](https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7) to give you some further background on the topic.

You can check out the [**Optimization Guide**](https://docs.openvinotoolkit.org/2019_R3/_docs_optimization_guide_dldt_optimization_guide.html) for more on the differences in optimization between devices.

………..

# Recap

https://www.youtube.com/watch?v=AVmFgZyk0T0

In this lesson we covered:

* Basics of the Inference Engine
* Supported Devices
* Feeding an Intermediate Representation to the Inference Engine
* Making Inference Requests
* Handling Results from the Inference Engine
* Integrating the Inference Model into an App

…………….

**Lesson Glossary**

#### Inference Engine

Provides a library of computer vision functions, supports calls to other computer vision libraries such as OpenCV, and performs optimized inference on Intermediate Representation models. Works with various plugins specific to different hardware to support even further optimizations.

#### Synchronous

Such requests wait for a given request to be fulfilled prior to continuing on to the next request.

#### Asynchronous

Such requests can happen simultaneously, so that the start of the next request does not need to wait on the completion of the previous.

#### [IECore](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IECore.html)

The main Python wrapper for working with the Inference Engine. Also used to load an IENetwork, check the supported layers of a given network, as well as add any necessary CPU extensions.

#### [IENetwork](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1IENetwork.html)

A class to hold a model loaded from an Intermediate Representation (IR). This can then be loaded into an IECore and returned as an Executable Network.

#### [ExecutableNetwork](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1ExecutableNetwork.html)

An instance of a network loaded into an IECore and ready for inference. It is capable of both synchronous and asynchronous requests, and holds a tuple of InferRequest objects.

#### [InferRequest](https://docs.openvinotoolkit.org/2019_R3/classie__api_1_1InferRequest.html)

Individual inference requests, such as image by image, to the Inference Engine. Each of these contain their inputs as well as the outputs of the inference request once complete.

##### LESSON 5

#### Deploying an Edge App

With the OpenVINO™ Toolkit fundamentals down, you’re ready to move onto more topics to get your edge app up and running. Learn about handling input streams, MQTT and more as you finish the course!

CONTINUE

**3 hours left**

……….

# Introduction

https://www.youtube.com/watch?v=IBP4tsdFRPg

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

https://www.youtube.com/watch?v=c-pyJ3XwWko

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

See the link further down below for more tutorials on OpenCV if you want to dive deeper.

### QUIZ QUESTION

Which of the following can OpenCV be used for?

* **Capturing a video stream**
* **Image pre-processing (re-sizing, converting between color spaces, etc.)**
* **Detecting features like edges or lines**
* 

Full-fledged neural network training like TensorFlow and Caffe

* **Drawing or writing on an image**

SUBMIT

# Thanks for completing that!

Nice work! The only one of these it isn’t used for typically is the full suite of neural network training you find in the frameworks like TensorFlow and Caffe - it does have some features for [**Neural Networks**](https://docs.opencv.org/2.4/modules/ml/doc/neural_networks.html) available, but you’d usually use libraries for that.

### Further Research

OpenCV has some [**pretty extensive tutorials**](https://docs.opencv.org/master/d9/df8/tutorial_root.html) 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

https://www.youtube.com/watch?v=F-7ofR4pdNE

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

https://www.youtube.com/watch?v=uNoIZI9bm6U

Training neural networks focuses a lot on 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?

………..

# Solution: Process Model Outputs

https://www.youtube.com/watch?v=s35d7IvQliE

My approach in this exercise was to check if the bad combination of pets was on screen, but also to track whether I already warned them in the current incident. Now, I might also consider re-playing the warning after a certain time period in a single consecutive incident, but the provided video file does not really have that long of consecutive timespans. I also output a "timestamp" by checking how many frames had been processed so far at 30 fps.

Before the video loop, I added:

counter = 0

incident\_flag = False

Within the loop, after a frame is read, I make sure to increment the counter: counter+=1.

I made an assess\_scene function for most of the processing:

**def** **assess\_scene**(result, counter, incident\_flag):

'''

Based on the determined situation, potentially send

a message to the pets to break it up.

'''

**if** result[0][1] == 1 **and** **not** incident\_flag:

timestamp = counter / 30

print("Log: Incident at {:.2f} seconds.".format(timestamp))

print("Break it up!")

incident\_flag = **True**

**elif** result[0][1] != 1:

incident\_flag = **False**

**return** incident\_flag

And I call that within the loop right after the result is available:

incident\_flag = assess\_scene(result, counter, incident\_flag)

#### Running the App

To run the app, I just used:

python app.py -m model.xml

Since the model was provided here in the same directory.

………

# Intro to MQTT

### https://www.youtube.com/watch?v=xe8137UtNCQ

### 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](http://mqtt.org/) for MQTT
* A [helpful post](https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport) on more of the basics of MQTT

………….

# Communicating with MQTT

### https://www.youtube.com/watch?v=3Jh-akmlmmU

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.

You’ll need the [documentation](https://pypi.org/project/paho-mqtt/) for paho-mqtt to answer the quiz below.

### QUIZ QUESTION

Match the code snippets below to whether they are being used for publishing or subscribing to a ”students” topic (given a MQTT client and some data).

data.publish(“students”)

client.subscribe.data

**COMMUNICATION FORM**

**CODE**

Publish

client.publish(“students”, data)

Subscribe

client.subscribe(“students”)

SUBMIT



# Thanks for completing that!

Great work! This Python library for working with MQTT makes publishing and subscribing pretty quick and easy.

### Further Research

* As usual, documentation is your friend. Make sure to check out the [documentation on PyPi](https://pypi.org/project/paho-mqtt/) related to the paho-mqtt Python library if you want to learn more about its functionality.
* Intel® has a [pretty neat IoT tutorial](https://software.intel.com/en-us/SetupGateway-MQTT) on working with MQTT with Python you can check out as well.

………….

# Streaming Images to a Server

https://www.youtube.com/watch?v=MyutnkUqIv4

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](https://www.ffmpeg.org/" \t "_blank) 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.

## Streaming Images to a Server Quiz

Given additional network impacts of streaming full video, why might it still be useful to be streaming video back to another server, given AI applied at the edge?

SUBMIT

### Further Research

We covered [FFMPEG](https://www.ffmpeg.org/) 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](https://opensource.com/article/19/1/basic-live-video-streaming-server)
* [Use Flask and Python](https://www.pyimagesearch.com/2019/09/02/opencv-stream-video-to-web-browser-html-page/)

…………….

# Handling Statistics and Images from a Node Server

https://www.youtube.com/watch?v=qHQ9HjmRGWw

[Node.js](https://nodejs.org/en/about/) 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](https://www.udacity.com/course/front-end-web-developer-nanodegree--nd0011) program if you want to learn more of these skills!

……….

# Solution: Server Communications

**https://www.youtube.com/watch?v=c2cNJgrvHmg**

**Note:** You will need to use port 3001 in the workspace for MQTT within app.py instead of the standard port 1883.

Let's focus on MQTT first, and then FFmpeg.

### MQTT

First, I import the MQTT Python library. I use an alias here so the library is easier to work with.

import paho.mqtt.client as mqtt

I also need to import socket so I can connect to the MQTT server. Then, I can get the IP address and set the port for communicating with the MQTT server.

HOSTNAME = socket.gethostname()

IPADDRESS = socket.gethostbyname(HOSTNAME)

MQTT\_HOST = IPADDRESS

MQTT\_PORT = 3001

MQTT\_KEEPALIVE\_INTERVAL = 60

This will set the IP address and port, as well as the keep alive interval. The keep alive interval is used so that the server and client will communicate every 60 seconds to confirm their connection is still open, if no other communication (such as the inference statistics) is received.

**Note:** The port here is 3001, instead of the normal MQTT port of 1883, as our classroom workspace environment only allows ports from 3000-3009 to be used. The real importance is here to make sure this matches to what is set for the MQTT broker server to be listening on, which in this case has also been set to 3001 (you can see this in config.js within the MQTT server's files in the workspace).

Connecting to the client can be accomplished with:

client = mqtt.Client()

client.connect(MQTT\_HOST, MQTT\_PORT, MQTT\_KEEPALIVE\_INTERVAL)

Note that mqtt in the above was my import alias - if you used something different, that line will also differ slightly, although will still use Client().

The final piece for MQTT is to actually publish the statistics to the connected client.

topic = "some\_string"

client.publish(topic, json.dumps({"stat\_name": statistic}))

The topic here should match to the relevant topic that is being subscribed to from the other end, while the JSON being published should include the relevant name of the statistic for the node server to parse (with the name like the key of a dictionary), with the statistic passed in with it (like the items of a dictionary).

client.publish("class", json.dumps({"class\_names": class\_names}))

client.publish("speedometer", json.dumps({"speed": speed}))

And, at the end of processing the input stream, make sure to disconnect.

client.disconnect()

### FFmpeg

FFmpeg does not actually have any real specific imports, although we do want the standard sys library

**import** sys

This is used as the ffserver can be configured to read from sys.stdout. Once the output frame has been processed (drawing bounding boxes, semantic masks, etc.), you can write the frame to the stdout buffer and flush it.

sys.stdout.buffer.write(frame)

sys.stdout.flush()

And that's it! As long as the MQTT and FFmpeg servers are running and configured appropriately, the information should be able to be received by the final node server, and viewed in the browser.

### Running the App

To run the app itself, with the UI server, MQTT server, and FFmpeg server also running, do:

python app.py | ffmpeg -v warning -f rawvideo -pixel\_format bgr24 -video\_size 1280x720 -framerate 24 -i - http:*//0.0.0.0:3004/fac.ffm*

This will feed the output of the app to FFmpeg.

………

# Analyzing Performance Basics

https://www.youtube.com/watch?v=bGPyVQnLnIw

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.

### QUIZ QUESTION

Which of the following are the most likely performance improvements experienced when deploying with OpenVINO™ and deploying at the edge?

* 

Higher inference accuracy

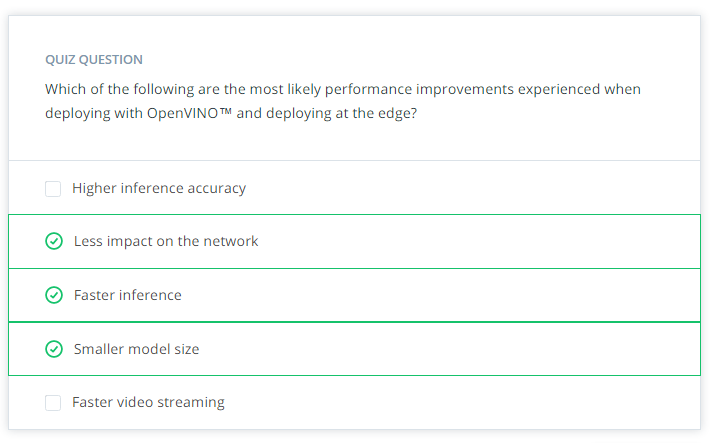
* Less impact on the network
* Faster inference
* Smaller model size
* 

Faster video streaming

SUBMIT

# Thanks for completing that!

Great job! OpenVINO™ can help reduce model size and speed up inference, while its use at the edge can help reduce impacts on your network vs. streaming the same input data to cloud services.



### 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](https://docs.openvinotoolkit.org/2019_R3/_docs_IE_DG_Intro_to_Performance.html) here for a preview.
* Did you know [Netflix uses 15% of worldwide bandwidth](https://www.sandvine.com/hubfs/downloads/phenomena/phenomena-presentation-final.pdf) 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.

…………..

**Model Use Cases**

https://www.youtube.com/watch?v=rQ0980eYeeA

It’s important to think about any additional use cases for a given model or application you build, which can reach far beyond the original training set or intended use. For example, object detection can be used for so many things, and focusing on certain classes along with some post-processing can lead to very different applications.

**Model Use Cases Quiz**

What are some use cases you can think of for a model that detects where the attention of someone on camera is directed at?



……….

**Concerning End User Needs**

https://www.youtube.com/watch?v=aHtne0LexrA

If you are building an app for certain end users, it’s very important to consider their needs. Knowing their needs can inform the various trade-offs you make regarding model decisions (speed, size, accuracy, etc.), what information to send to servers, security of information, etc. If they have more resources available, you might go for a higher accuracy but more resource-intensive app, while an end user with remote, low-power devices will likely have to sacrifice some accuracy for a lighter, faster app, and need some additional considerations about network usage.

This is just to get you thinking - building edge applications is about more than just models and code.

**Concerning End User Needs Quiz**

How would you explain to an end user the potential trade-offs made when optimizing for different models and implementations, such as speed, memory size, and network impacts?

SUBMIT

NEXT

……..

# Recap

https://www.youtube.com/watch?v=6egpCIvzUl8

In this lesson we covered:

* 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
* Thinking about additional model use cases, as well as end user needs

…………

# Lesson 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](https://www.fullstackpython.com/flask.html) 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.

………..

# Course Recap

<https://www.youtube.com/watch?v=27C3zdKmt_M>

You’ve accomplished something amazing! You went from the basics of AI at the Edge, built your skills with pre-trained models, the Model Optimizer, Inference Engine and more with the Intel® Distribution of OpenVINO™ Toolkit, and even learned more about deploying an app at the edge. Best of luck on the project, and I look forward to seeing what you build next!

### Intel® DevMesh

Check out the [Intel® DevMesh](https://devmesh.intel.com/) website for some more awesome projects others have built, join in on existing projects, or even post some of your own!

### Continuing with the Toolkit

If you want to learn more about OpenVINO, you can download the toolkit here:

Top of Form

Download the Toolkit

Muhammad Yaman Ghaibeh

YGhaibeh@Hotmail.com