Introduction

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

It is an open-source library useful for edge deployment due to its performance maximization and pre-trained models

The OpenVINO™ Toolkit's name comes from "Open Visual Inferencing and Neural Network Optimization". 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.

Pre-Trained Models in OpenVINO™

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.

Types of Computer Vision Models

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 draw 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.

- semantic segmentation: all instances of a class are considered as one.
- Instance segmentation: separates instances of a class as separate objects.

QUIZ

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

- Determines what an object in a given image is, although not wherein 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

Case Studies in Computer Vision

Let's have a look at some common Neural Network architectures used:

We focused on SSD, ResNet, and MobileNet in the video.

SSD is an object detection network that combines classification with object detection through the use of default bounding boxes at different network levels.

ResNet utilizes residual layers to "skip" over sections of layers, helping to avoid the vanishing gradient problem with very deep neural networks.

MobileNet utilizes layers like 1x1 convolutions to help cut down on computational complexity and network size, leading to fast inference without a 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.

QUIZ

The Single Shot Multibox Detector (SSD) model:

- Performed classifications on different convolutional layer feature map using default bounding boxes

The "residual learning" achieved in the ResNet model architecture is achieved by:

- Using "skip" layers that pass information forward by a couple of layers

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
- YOLO
- Faster RCNN
- MobileNet
- ResNet
- Inception

Available Pre-Trained Models in OpenVINO™

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 models in 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™.

Pre-Trained models: Types

- Classification: Age and Gender Recognition
- Detection: Pedestrian Detection
- Segmentation: Advanced Roadside Identification

Pre-Trained models: Architectures

- SSD: Enhanced Model - Face Detection
- MobileNet: Standard Model - Face Detection
- SSD + MobileNet: Pedestrian and Vehcle Detection

You can check out the full list of pre-trained models available in the Intel® Distribution of OpenVINO™ here.

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.

Optimizations on the Pre-Trained Models

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 the progress of limb movement range over the course of treatment.

Pre-processing Inputs

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.

Assignment to preprocess images

To resize image (width, height): preprocessed_image = cv2.resize(preprocessed_image, (72, 72))
To move C from 3rd position to 1st preprocessed_image = np.moveaxis(preprocessed_image, 2, 0)
To add and extra dimension: preprocessed_image = np.expand_dims(preprocessed_image, axis=0)

Alternatively

```
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
```

Handling Network Outputs

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

Quiz Information

In a network like SSD 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[1] > conf_threshold:
        xmin = int(box[2] * width)
        ymin = int(box[3] * height)
        xmax = int(box[4] * width)
        Xmax = int(box[5] * height)
```

Note: *width, * height is done to rescale to image size.

Further Research

- Here is a great write-up on working with SSD and its output
- This post gets into more of the differences in moving from models with bounding boxes to those using semantic segmentation

Running Your First Edge App

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.

Lesson Glossary

Edge Application

Applications with inference run on local hardware, sometimes without network connections, such as the 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 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. 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 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 an 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

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

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

A network, expanding on R-CNN and Fast R-CNN, 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

A neural network architecture optimized for speed and size with minimal loss of inference accuracy through the use of techniques like 1x1 convolutions. As such, MobileNet is more useful in mobile applications that substantially larger and slower networks.

ResNet

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 experienced by deeper neural networks.

Inception

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, 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.