The Inference Engine

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 the 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.

Supported Devices

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 for one way to do so on-device. The Inference Engine itself is still supported by this combination.

Using the Inference Engine with an IR

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 of variables to actually use the IECore and IENetwork. In the IECore documentation, no arguments are needed to initialize. To use IENetwork, 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, 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
- IE Network
- IE Core

Sending Inference Requests to the IE

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
- Infer Request documentation

Handling Results

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), to get an array of the probabilities returned from the inference request.

Asynchronous Requests

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.

Further Research

- For more on Synchronous vs. Asynchronous, check out this helpful post.
- You can also check out the documentation 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. You'll get a chance to practice with Synchronous and Asynchronous Requests in the upcoming exercise.

Behind the Scenes of Inference Engine

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.

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

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

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

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

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.