Intel Edge AI Foundations - Lesson 4

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

Optimization:
Model Optimizer made some **improvements to size and complexity** of the models to **improve memory** and **computation times**.

Model Optimizer optimization:
1. Quantization
2. Freezing
3. Fusion

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.

Document:
https://docs.openvinotoolkit.org/latest/_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html

Supported Devices:
CPU, GPU, VPU and FPGA

Using the Inference Engine with an IR:
To load an IR into the Inference Engine, you'll mostly work with two classes in the **openvino.inference_engine** library for Python.

IECore - Which is the Python wrapper to **work with the Inference Engine:**
No arguments are needed to initialize.

```
def
__init__ (self, xml_config_file="")
def
get_versions (self, device_name)
def
load_network (self, IENetwork, network, str, device_name, config=None, int, num_requests=1)
def
query_network (self, IENetwork, network, str, device_name, config=None)
def
register_plugin (self, plugin_name, device_name)
def
register_plugins (self, xml_config_file)
def
unregister_plugin (self, device_name)
```

```
def
add_extension
def
get_metric
```

IENetwork - which is what will initially **hold the network and get loaded into IECore:**
We need to load arguments named **model** and **weights to initialize** - the **XML** and **Binary files** that make up the model's **Intermediate Representation.**

```
def
__init__

def
from_ir
def
serialize (self, path_to_xml, path_to_bin="")
def
reshape (self, input_shapes)
```

Check Supported Layers:
Function **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"

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.

Example:

```
import argparse
### TODO: Load the necessary libraries
import os
from openvino.inference_engine import IENetwork, IECore

CPU_EXTENSION =
"/opt/intel/openvino/deployment_tools/inference_engine/lib/intel64/libcpu_extension_sse4.so"

def get_args():
    '''
    Gets the arguments from the command line.
    '''
    parser = argparse.ArgumentParser("Load an IR into the Inference Engine")
    # -- Create the descriptions for the commands
    m_desc = "The location of the model XML file"
```

```
    # -- Create the arguments
    parser.add_argument("-m", help=m_desc)
    args = parser.parse_args()

    return args

def load_to_IE(model_xml):
    ### Load the Inference Engine API
    inference_engine = IECore()

    ### Load IR files into their related class
    wb_file = os.path.splitext(model_xml)[0] + ".bin"
    net = IENetwork(model=model_xml, weights=wb_file)

    ### Add a CPU extension, if applicable.
    inference_engine.add_extension(CPU_EXTENSION, "CPU")

    ### Get the supported layers of the network
    supported_layers = inference_engine.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
    inference_engine.load_network(net, "CPU")

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

    return


def main():
    args = get_args()
    load_to_IE(args.m)


if __name__ == "__main__":
    main()
```

When we load the **IENetwork into the IECore**, We'll get an ExecutableNetwork as a return value, which is what you will send inference requests to. There are two types of inference requests you can make: **Synchronous and Asynchronous.**

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.

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:
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.

Demo:
https://github.com/opencv/open_model_zoo/blob/master/demos/object_detection_demo_ssd_async/README.md

```
import argparse
import cv2
from helpers import load_to_IE, preprocessing

CPU_EXTENSION =
"/opt/intel/openvino/deployment_tools/inference_engine/lib/intel64/libcpu_extension_sse4.so"

def get_args():
    '''
    Gets the arguments from the command line.
    '''
    parser = argparse.ArgumentParser("Load an IR into the Inference Engine")
    # -- Create the descriptions for the commands
    m_desc = "The location of the model XML file"
    i_desc = "The location of the image input"
    r_desc = "The type of inference request: Async ('A') or Sync ('S')"

    # -- Create the arguments
    parser.add_argument("-m", help=m_desc)
    parser.add_argument("-i", help=i_desc)
    parser.add_argument("-r", help=i_desc)
    args = parser.parse_args()

    return args


def async_inference(exec_net, input_blob, image):
    #Performs asynchronous inference
    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

def sync_inference(exec_net, input_blob, image):
    #Performs synchronous inference

    result = exec_net.infer({input_blob: image})

    return result

def perform_inference(exec_net, request_type, input_image, input_shape):
    '''
    Performs inference on an input image, given an ExecutableNetwork
    '''
    # Get input image
    image = cv2.imread(input_image)
    # Extract the input shape
    n, c, h, w = input_shape
    # Preprocess it (applies for the IRs from the Pre-Trained Models lesson)
    preprocessed_image = preprocessing(image, h, w)

    # Get the input blob for the inference request
    input_blob = next(iter(exec_net.inputs))

    # Perform either synchronous or asynchronous inference
    request_type = request_type.lower()
    if request_type == 'a':
        output = async_inference(exec_net, input_blob, preprocessed_image)
    elif request_type == 's':
        output = sync_inference(exec_net, input_blob, preprocessed_image)
    else:
        print("Unknown inference request type, should be 'A' or 'S'.")
        exit(1)

    # Return the exec_net for testing purposes
    return output


def main():
    args = get_args()
    exec_net, input_shape = load_to_IE(args.m, CPU_EXTENSION)
    perform_inference(exec_net, args.r, args.i, input_shape)


if __name__ == "__main__":
    main()
```

Handling Result:

InferRequest attributes:

InferRequest attributes:

Inputs - Image frame
Output - Result
Latency - Inference Time

Notes:
https://hackernoon.com/the-ultimate-guide-to-starting-your-first-iot-project-8b0644fbbe6d

OpenVINO pipeline:

1. Convert a model into IR using model optimizer.
2. Load into the Inference Engine.
3. Add Pre-Processing.
4. Make Inference Request
5. Handle and Process the output