# Introduction

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

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.

# Optimization Techniques

Here, I mostly focused on three optimization techniques: quantization, freezing, and fusion.

## 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 the 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.

# Supported Frameworks

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.

# Intermediate Representations

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.

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.

# Using the Model Optimizer with TensorFlow Models

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.

# Using the Model Optimizer with Caffe Models

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 inputted 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. You'll work through this process in the next exercise.

# Using the Model Optimizer with ONNX Models

The process for converting an ONNX model is again quite similar to the previous two, although ONNX does not have any ONNX-specific arguments to the Model Optimizer. So, you'll only have the general arguments for items like changing the precision.
Additionally, if you are working with PyTorch or Apple ML models, they need to be converted to ONNX format first, which is done outside of the OpenVINO™ Toolkit. See the link further down on this page if you are interested in doing so.

### Further Research

- The developer documentation for Converting ONNX Models can be found here. You'll work through this process in the next exercise.
- ONNX also has additional models available in the ONNX Model Zoo. By converting these over to Intermediate Representations, you can expand even further on the pre-trained models available to you.

### PyTorch to ONNX

If you are interested in converting a PyTorch model using ONNX for use with the OpenVINO™ Toolkit, check out this link for the steps to do so. From there, you can follow the steps for ONNX models to get an Intermediate Representation.

# Cutting Parts of a Model

Cutting a model is mostly applicable to 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 are 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 the developer documentation for cutting a model.

# Supported Layers

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 by 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.

## Supported Layers List

Check out the full list of supported layers here.

# Custom Layers

Custom layers are necessary and important to have the 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 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 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.

## 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

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

### TensorFlow

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

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

### ONNX

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

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 maybe 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 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 outsides 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 specific to the framework.