Coral

TensorFlow models on the Edge TPU

In order for the Edge TPU to provide high-speed neural network performance with a low-power cost, the Edge TPU supports a specific set of neural network operations and architectures. This page describes what types of models are compatible with the Edge TPU and how you can create them, either by compiling your own TensorFlow model or retraining an existing model with transfer-learning.

If you're looking for information about how to run a model, read the Edge TPU inferencing overview.

Or if you just want to try some models, check out our trained models.

Compatibility overview

The Edge TPU is capable of executing deep feed-forward neural networks such as convolutional neural networks (CNN). It supports only TensorFlow Lite models that are fully 8-bit quantized and then compiled specifically for the Edge TPU.

If you're not familiar with TensorFlow Lite, it's a lightweight version of TensorFlow designed for mobile and embedded devices. It achieves low-latency inference in a small binary size—both the TensorFlow Lite models and interpreter kernels are much smaller. TensorFlow Lite models can be made even smaller and more efficient through quantization, which converts 32-bit parameter data into 8-bit representations (which is required by the Edge TPU).

You cannot train a model directly with TensorFlow Lite; instead you must convert your model from a TensorFlow file (such as a `.pb` file) to a TensorFlow Lite file (a `.tflite` file), using the TensorFlow Lite converter.

Figure 1 illustrates the basic process to create a model that's compatible with the Edge TPU. Most of the workflow uses standard TensorFlow tools. Once you have a TensorFlow Lite model, you then use our Edge TPU compiler to create a `.tflite` file that's compatible with the Edge TPU.

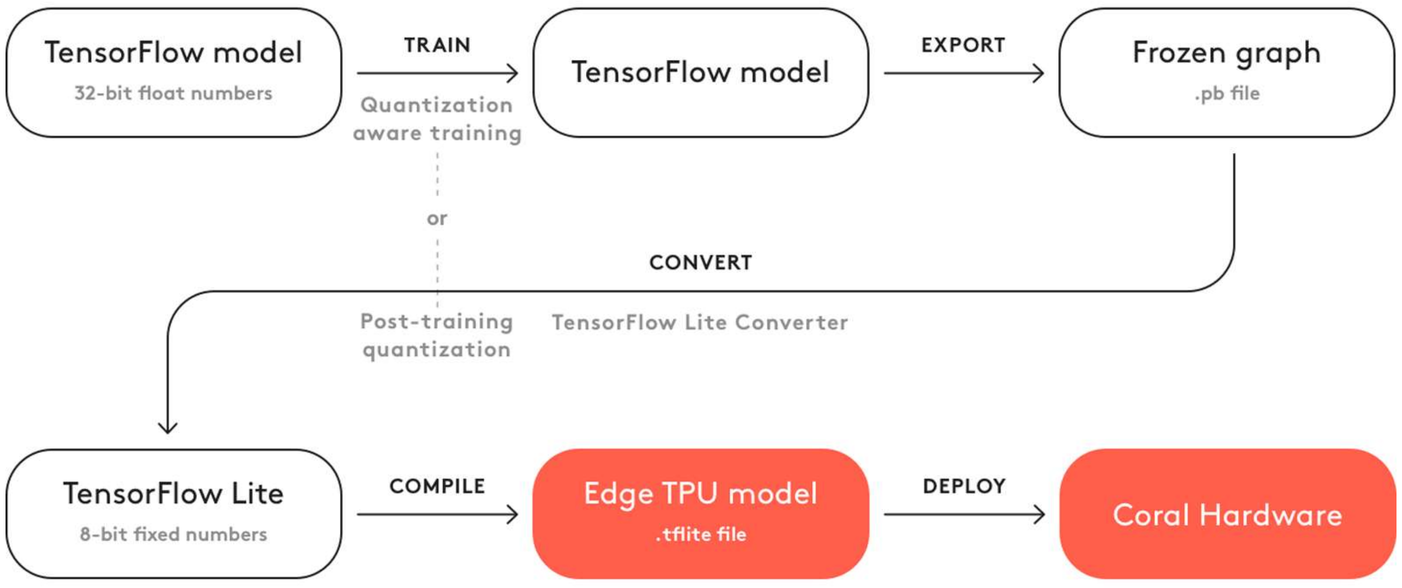

Figure 1. The basic workflow to create a model for the Edge TPU

However, you don't need to follow this whole process to create a good model for the Edge TPU. Instead, you can leverage existing TensorFlow models that are compatible with the Edge TPU by retraining them with your own dataset. For example, MobileNet is a popular image classification/detection model architecture that's compatible with the Edge TPU. We've created several versions of this model that you can use as a starting point to create your own model that recognizes different objects. To get started, see the next section about how to retrain an existing model with **transfer learning**.

If you have designed—or plan to design—your own model architecture, then you should read the section below about **model requirements**.

Transfer learning

Instead of building your own model and then training it from scratch, you can retrain an existing model that's already compatible with the Edge TPU, using a technique called transfer learning (sometimes also called "fine tuning").

Training a neural network from scratch (when it has no computed weights or bias) can take days-worth of computing time and requires a vast amount of training data. But transfer learning allows you to start with a model that's already trained for a related task and then perform further training to teach the model new classifications using a smaller training dataset. You can do this by retraining the whole model (adjusting the weights across the whole network), but you can also achieve very accurate results by simply removing the final layer that performs classification, and training a new layer on top that recognize your new classes.

Using this process, with sufficient training data and some adjustments to the hyperparameters, you can create a highly accurate TensorFlow model in a single sitting. Once you're happy with the model's performance, simply **convert it to TensorFlow Lite** and then **compile it for the Edge TPU**. And because the model architecture doesn't change during transfer learning, you know it will fully compile for the Edge TPU (assuming you start with a compatible model).

To get started without any setup, **try our Google Colab retraining tutorials**. All these tutorials perform transfer learning in cloud-hosted Jupyter notebooks.

Transfer learning on-device

If you're using an image classification model, you can also perform accelerated transfer learning on the Edge TPU. Our Python and C++ APIs offer two different techniques for on-device transfer learning:

- Weight imprinting on the last layer (`ImprintingEngine` in Python or C++)
- Backpropagation on the last layer (`SoftmaxRegression` in Python or C++)

In both cases, you must provide a model that's specially designed to allow training on the last layer. The required model structure is different for each API, but the result is basically the same: the last fully-connected layer where classification occurs is separated from the base of the graph. Then only the base of the graph is compiled for the Edge TPU, which leaves the weights in the last layer accessible for training. More detail about the model architecture is available in the corresponding documents below. For now, let's compare how retraining works for each technique:

- Weight imprinting takes the output (the embedding vectors) from the base model, adjusts the activation vectors with L2-normalization, and uses those values to compute new weights in the final layer—it averages the new vectors with those already in the last layer's weights. This allows for effective training of new classes with very few sample images.
- Backpropagation is an abbreviated version of traditional backpropagation. Instead of backpropagating new weights to all layers in the graph, it updates only the fully-connected layer at the end of the graph with new weights. This is the more traditional training strategy that generally achieves higher accuracy, but it requires more images and multiple training iterations.

When choosing between these training techniques, you might consider the following factors:

- **Training sample size:** Weight imprinting is more effective if you have a relatively small set of training samples: anywhere from 1 to 200 sample images for each class (as few as 5 can be effective and the API sets a maximum of 200). If you have more samples available for training, you'll likely achieve higher accuracy by using them all with backpropagation.
- **Training sample variance:** Backpropagation is more effective if your dataset includes large intra-class variance. That is, if the images within a given class show the subject in significantly different ways, such as in angle or size, then backpropagation probably works better. But if your application operates in an environment where such variance is low, and your training samples thus also have little intra-class variance, then weight imprinting can work very well.
- **Adding new classes:** Only weight imprinting allows you to add new classes to the model after you've begun training. If you're using backpropagation, adding a new class after you've begun training requires that you restart training for all classes. Additionally, weight imprinting allows you to retain the classes from the pre-trained model (those trained before converting the model for the Edge TPU); whereas backpropagation requires all classes to be learned on-device.
- **Model compatibility:** Backpropagation is compatible with more model architectures "out of the box"; you can convert existing, pre-trained MobileNet and Inception models into embedding extractors that are compatible with on-device backpropagation. To use weight imprinting, you must use a model with some very specific layers and then train it in a particular manner before using it for on-device training (currently, we offer a version of MobileNet v1 with the proper modifications).

In both cases, the vast majority of the training process is accelerated by the Edge TPU. And when performing inferences with the retrained model, the Edge TPU accelerates everything except the final classification layer, which runs on the CPU. But because this last layer accounts for only a small portion of the model, running this last layer on the CPU should not significantly affect your inference speed.

To learn more about each technique and try some sample code, see the following pages:

- Retrain a classification model on-device with weight imprinting
- Retrain a classification model on-device with backpropagation

Model requirements

If you want to build a TensorFlow model that takes full advantage of the Edge TPU for accelerated inferencing, the model must meet these basic requirements:

- **Tensor parameters are quantized** (8-bit fixed-point numbers; int8 or uint8).
- **Tensor sizes are constant at compile-time** (no dynamic sizes).
- **Model parameters (such as bias tensors) are constant at compile-time**.
- **Tensors are either 1-, 2-, or 3-dimensional**. If a tensor has more than 3 dimensions, then only the 3 innermost dimensions may have a size greater than 1.
- **The model uses only the operations supported by the Edge TPU** (see table 1 below).

Failure to meet these requirements could mean your model cannot compile for the Edge TPU at all, or only a portion of it will be accelerated, as described in the section below about compiling.

> **Note:** If you pass a model to the Edge TPU Compiler that uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). So as long as your tensor parameters are quantized, it's okay if the input and output tensors are float because they'll be converted on the CPU.

Supported operations

When building your own model architecture, be aware that only the operations in the following table are supported by the Edge TPU. If your architecture uses operations not listed here, then only a portion of the model will compile for the Edge TPU and the remaining ops will execute on the CPU.

> **Note:** When creating a new TensorFlow model, refer to the list of operations compatible with TensorFlow Lite. Also beware that, if you're using the Dev Board Micro, any model operations that execute on the MCU (instead of the Edge TPU) must be compatible with TensorFlow Lite for Microcontrollers, which supports fewer operations than TensorFlow Lite.

Table 1. All operations supported by the Edge TPU and any known limitations

Operation name	Runtime version*	Known limitations
Add	All	
AveragePool2d	All	No fused activation function.
Concatenation	All	No fused activation function. If any input is a compile-time constant tensor, there must be only 2 inputs, and this constant tensor must be all zeros (effectively, a zero-padding op).
Conv2d	All	Must use the same dilation in x and y dimensions.
DepthwiseConv2d	≤12	Dilated conv kernels are not supported.
	≥13	Must use the same dilation in x and y dimensions.
ExpandDims	≥13	
FullyConnected	All	Only the default format is supported for fully-connected weights. Output tensor is one-dimensional.
L2Normalization	All	
Logistic	All	
LSTM	≥14	Unidirectional LSTM only.
Maximum	All	
MaxPool2d	All	No fused activation function.
Mean	≤12	No reduction in batch dimension. Supports reduction along x- and/or y-dimensions only.
	≥13	No reduction in batch dimension. If a z-reduction, the z-dimension must be multiple of 4.
Minimum	All	
Mul	All	
Pack	≥13	No packing in batch dimension.
Pad	≤12	No padding in batch dimension. Supports padding along x- and/or y-dimensions only.
	≥13	No padding in batch dimension.

Operation name	Runtime version*	Known limitations
PReLU	≥13	Alpha must be 1-dimensional (only the innermost dimension can be >1 size). If using Keras PReLU with 4D input (batch, height, width, channels), then shared_axes must be [1,2] so each filter has only one set of parameters.
Quantize	≥13	
ReduceMax	≥14	Cannot operate on the batch dimension.
ReduceMin	≥14	Cannot operate on the batch dimension.
ReLU	All	
ReLU6	All	
ReLUN1To1	All	
Reshape	All	Certain reshapes might not be mapped for large tensor sizes.
ResizeBilinear	All	Input/output is a 3-dimensional tensor. Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision.
ResizeNearestNeighbor	All	Input/output is a 3-dimensional tensor. Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision.
Rsqrt	≥14	
Slice	All	
Softmax	All	Supports only 1-D input tensor with a max of 16,000 elements.
SpaceToDepth	All	
Split	All	No splitting in batch dimension.
Squeeze	≤12	Supported only when input tensor dimensions that have leading 1s (that is, no relayout needed). For example input tensor with [y][x][z] = 1,1,10 or 1,5,10 is ok. But [y][x][z] = 5,1,10 is not supported.
	≥13	None.
StridedSlice	All	Supported only when all strides are equal to 1 (that is, effectively a Stride op), and with ellipsis-axis-mask == 0, and new-axis-max == 0.
Sub	All	
Sum	≥13	Cannot operate on the batch dimension.
Squared-difference	≥14	

Operation name	Runtime version*	Known limitations
Tanh	All	
Transpose	≥14	
TransposeConv	≥13	

* *You must use a version of the Edge TPU Compiler that corresponds to* ***the runtime version****.*

Regardless of the operations you use, be sure you abide by the **basic model requirements** above.

Quantization

Quantizing your model means converting all the 32-bit floating-point numbers (such as weights and activation outputs) to the nearest 8-bit fixed-point numbers. This makes the model smaller and faster. And although these 8-bit representations can be less precise, the inference accuracy of the neural network is not significantly affected.

For compatibility with the Edge TPU, you must use either quantization-aware training (recommended) or full integer post-training quantization.

Quantization-aware training (for TensorFlow 1) uses "fake" quantization nodes in the neural network graph to simulate the effect of 8-bit values during training. Thus, this technique requires modification to the network before initial training. This generally results in a higher accuracy model (compared to post-training quantization) because it makes the model more tolerant of lower precision values, due to the fact that the 8-bit weights are learned through training rather than being converted later. It's also currently compatible with more operations than post-training quantization.

> **Note:** As of August, 2021, the **quantization-aware training API with TF2** is not compatible with the object detection API; it is compatible with image classification models only. For more compatibility, including with the object detection API, you may use **quantization-aware training with TF1** or use post-training quantization with TF2.

Full integer post-training quantization doesn't require any modifications to the network, so you can use this technique to convert a previously-trained network into a quantized model. However, this conversion process requires that you supply a representative dataset. That is, you need a dataset that's formatted the same as the original training dataset (uses the same data range) and is of a similar style (it does not need to contain all the same classes, though you may use previous training/evaluation data). This representative dataset allows the quantization process to measure the dynamic range of activations and inputs, which is critical to finding an accurate 8-bit representation of each weight and activation value.

However, not all TensorFlow Lite operations are currently implemented with an integer-only specification (they cannot be quantized using post-training quantization). By default, the TensorFlow Lite converter leaves those operations in their float format, which is not compatible with the Edge TPU. As described below, the **Edge TPU Compiler** stops compiling when it encounters an incompatible operation (such as a non-quantized op), and the remainder of the model executes on the CPU. So to enforce integer-only quantization, you can instruct the converter to throw an error if it encounters a non-quantizable operation. Read more about **integer-only quantization**.

For examples of each quantization strategy, see our Google Colab tutorials for model training.

For more details about how quantization works, read the TensorFlow Lite 8-bit quantization spec.

Float input and output tensors

As mentioned in the model requirements, the Edge TPU requires 8-bit quantized input tensors. However, if you pass the Edge TPU Compiler a model that's internally quantized but still uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). Likewise, the output is dequantized at the end.

So it's okay if your TensorFlow Lite model uses float inputs/outputs. But beware that if your model uses float input and output, then there will be some amount of latency added due to the data format conversion, though it should be negligible for most models (the bigger the input tensor, the more latency you'll see).

To achieve the best performance possible, we recommend fully quantizing your model so the input and output use int8 or uint8 data, which you can do by setting the input and output type with the TF Lite converter, as shown in the TensorFlow docs for integer-only quantization.

Compiling

After you train and convert your model to TensorFlow Lite (with quantization), the final step is to compile it with the Edge TPU Compiler.

If your model does not meet all the requirements listed at the top of this section, it can still compile, but only a portion of the model will execute on the Edge TPU. At the first point in the model graph where an unsupported operation occurs, the compiler partitions the graph into two parts. The first part of the graph that contains only supported operations is compiled into a custom operation that executes on the Edge TPU, and everything else executes on the CPU, as illustrated in figure 2.

> **Note:** If you're using the Dev Board Micro, any model operations that execute on the MCU (because they do not compile for the Edge TPU) must be compatible with TensorFlow Lite for Microcontrollers, which supports fewer operations than TensorFlow Lite. Also beware that large models might not fit into the Dev Board Micro memory.

> **Note:** Currently, the Edge TPU compiler cannot partition the model more than once, so as soon as an unsupported operation occurs, that operation and everything after it executes on the CPU, even if supported operations occur later.

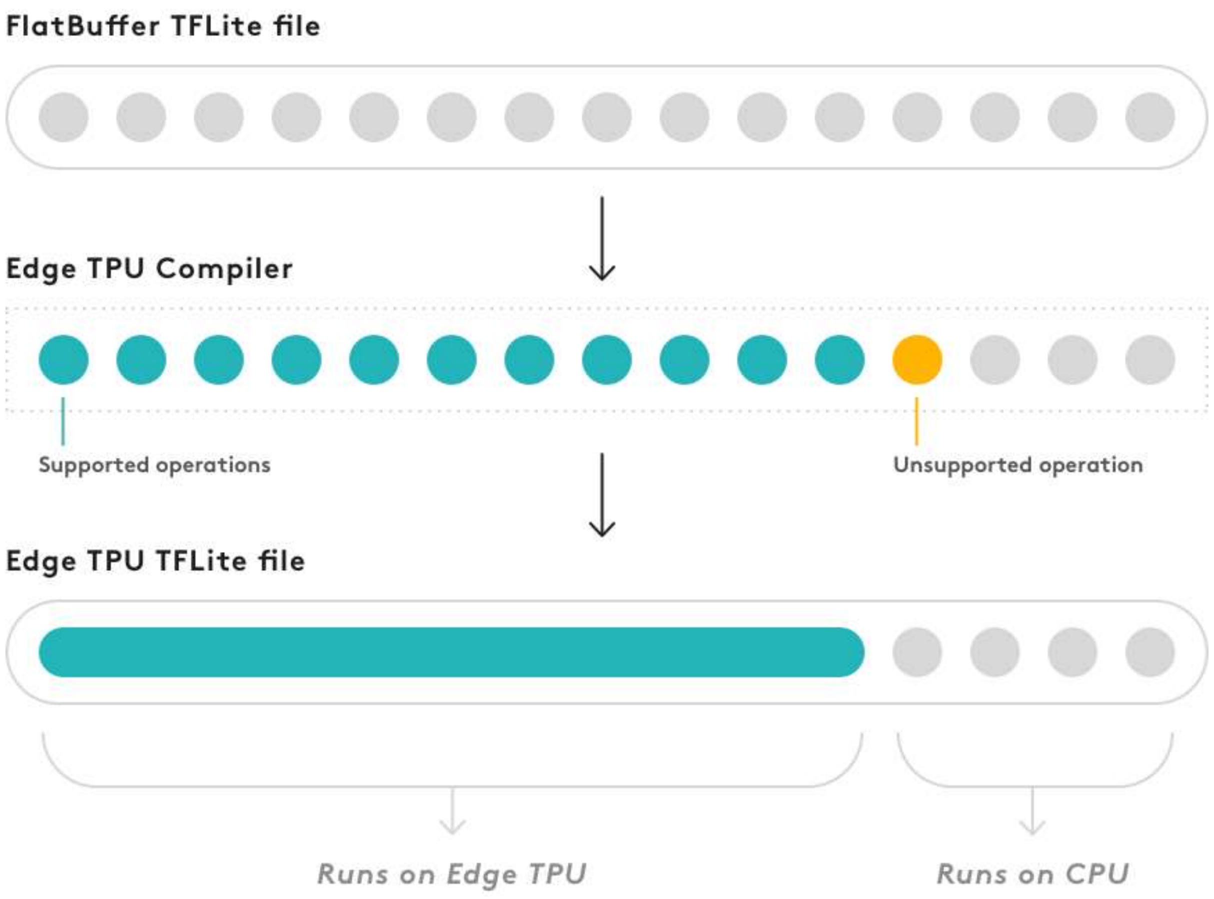

Figure 2. The compiler creates a single custom op for all Edge TPU compatible ops, until it encounters an unsupported op; the rest stays the same and runs on the CPU

If you inspect your compiled model (with a tool such as `visualize.py`), you'll see that it's still a TensorFlow Lite model except it now has a custom operation at the beginning of the graph. This custom operation is the only part of your model that is actually compiled—it contains all the operations that run on the Edge TPU. The rest of the graph (beginning with the first unsupported operation) remains the same and runs on the CPU.

If part of your model executes on the CPU, you should expect a significantly degraded inference speed compared to a model that executes entirely on the Edge TPU. We cannot predict how much slower your model will perform in this situation, so you should experiment with different architectures and strive to create a model that is 100% compatible with the Edge TPU. That is, your compiled model should contain only the Edge TPU custom operation.

Note: When compilation completes, the **Edge TPU compiler** tells you how many operations can execute on the Edge TPU and how many must instead execute on the CPU (if any at all). But beware that the percentage of operations that execute on the Edge TPU versus the CPU does not correspond to the overall performance impact—if even a small fraction of your model executes on the CPU, it can potentially slow the inference speed by an order of magnitude (compared to a version of the model that runs entirely on the Edge TPU).