nota
On-device AI

TensorFlow models on the Edge TPU

Coral Board

1.0.0

2021. 06. 17

Global Business Development

Document 링크 : https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

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.

Edge TPU 가 저전력, 고속 NN 성능을 제공하기 위해, Edge TPU 는 특정 NN 작동 및 아키텍처 세트를 지원한다. (==뒤에 특정 NN 작동 및 아키텍처 세트 설명할 생각이다)

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.

이 페이지는 Edge TPU 와 호환되는 모델 유형과 모델을 생성하는 방법 그리고 사용자가 어떻게 그들을 만들 수 있는 지 설명합니다. 또한, 현존하는 모델을 transfer-learning 으로 retraining 하거나, 사용자 자신의 TF model 을 컴파일 할 수 있는 방법이 설명되어 있습니다.

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

만약, 당신이 어떻게 모델을 돌리는 지 정보(원리 이론)가 필요하면 (초록색 링크)를 읽어봐라.

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

또는 당신이 그냥 몇 가지 모델(이미 되어 있는 거 올려서 테스트)을 시도해보고 싶다면 우리의 (초록색 링크)를 체크해 보아라.

Compatibility overview

The Edge TPU is capable of executing deep feed-forward neural networks such as convolutional neural networks (CNN).

Edge TPU 는 CNN 와 같은 deep feed-forward NN 을 작동시킬 수 있습니다.

It supports only TensorFlow Lite models that are fully 8-bit quantized and then compiled specifically for the Edge TPU.

Edge TPU 를 위해 스페샬(찡긋v)하게 compile 되었고 fully 8-bit quantized 된 TF Lite 모델만 Edge TPU 에서 구동됩니다.

If you're not familiar with TensorFlow Lite, it's a lightweight version of TensorFlow designed for mobile and embedded devices.

만약 네가 TF lite 에 익숙하지 않다면, TF lite 는 TF 가 mobile 과 임베디드 기기를 위해 경량화 된 버전으로 생각하면 됩니다.

It achieves low-latency inference in a small binary size—both the TensorFlow Lite models and interpreter kernels are much smaller.

TF lite 는 작은 binary 사이즈에서 low-latency inference 로 작동합니다. (TF lite 가 TF 에 비해) TF lite 모델과 interpreter 커널 둘 다에서 더욱 작습니다

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).

TF lite 모델은 quantization 을 통해 더욱 작고 더욱 효율적으로 만들 수 있습니다. Quantization 이 뭐냐. 32bit parameter 데이터를 8-bit 으로 표현하는 것이다.(Edge TPU 에서 요구하는 사항임)

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.

너는 TF lite 모델을 바로 training 할 수 없다. 대신에 너는 무조건 너의 모델을 TF 파일(ex .pb 파일)을 TF lite 파일로 바꿔서 training 할 수 있다. (TF lite converter 를 사용하셈)

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.

Fig 1 에서 Edge TPU 에 호환되는 기본적인 프로세스를 그리고 있다. Standard TF tools 에서 사용하는 workflow 랑 똑같다.

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.

네가 TF Lite 모델을 갖는 단계까지 오면, Edge TPU compiler 를 사용해서 .tflite 파일을 edge TPU 에 호환되게 바꿔줘라.

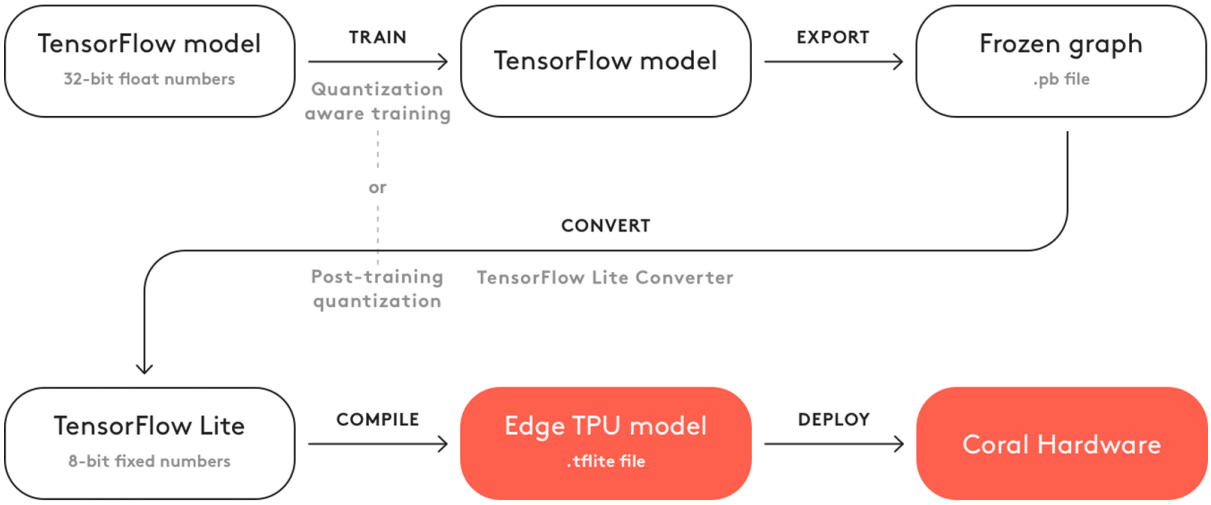

Figure 1. The basic workflow to create a model for the Edge TPU

However, you don't need to follow this whole process to create a good model for the Edge TPU.

군데, 너는 Edge TPU 에 좋은 모델을 만들기 위해서 모든 프로세스를 따를 필요는 없다.

Instead, you can leverage existing TensorFlow models that are compatible with the Edge TPU by retraining them with your own dataset.

대신에, 네가 가진(원하는) dataset 으로 현존하는 TF models 를 Edge TPU 에 호환되게 활용(==바꿀)할 수 있다.

For example, MobileNet is a popular image classification/detection model architecture that's compatible with the Edge TPU.

예를 들면, MobileNet image classification/dection 모델로 유명한 거 너도 알징? 그걸 Edge TPU 에 호환되게 바꿀 수 있다 이말이야~

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.

우리는 다른 object 들(기존 꺼 말고 새로운 object)를 알아보는 너만의 모델을 만드는 시발점으로 사용할 수 있도록 몇 가지 버전을 준비했어~(찡긋 v)

To get started, see the next section about how to retrain an existing model with transfer learning.

시작하려면, 다음 섹션에서 현존하는 모델을 transfer learning 으로 retraining 하는 방법을 보세여

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

만약 네가 너만의 모델을 디자인하거나, 디자인할 예정이라면 아래에 있는 model requirements 를 꼭 읽어봐야 해.

Tip: If you're building an image classification application, you can also use Cloud AutoML Vision to train a model with your own images, refine the model accuracy, and then export it for the Edge TPU—all from a graphical interface so you never use the TensorFlow tools directly.

꿀팁쓰! 만약 네가 image classification 어플리케이션을 만들 생각이라면 Cloud AutoML Vision 을 사용해서 모델의 정확도를 조정(==다듬다 ==맘에 들게 손 봐주다)하거나, 네가 원하는 사진으로 model 을 train 할 수 있어. 그리구 Edge TPU 에 export 할 수 있다. 모든 것은 TF tool 를 네가 사용 안 해봤더라도 직관적으로 볼 수 있게 그래픽 인터페이스를 제공하고 있다.

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").

너만의 모델을 만들고 training 하는 대신에, 너는 transfer learning(==fine tuning 이랑 같은 말임)이라 불리는 테크닉을 이용해서 Edge TPU 에 맞게 이미 컴파일 된 현존하는 모델을 retrain 해서 사용할 수 있다.

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.

scratch 로(이미 계산된 weight 나 bias 가 없다는 가정하에) NN 을 training 하는 건 며칠동안 걸릴 수도

있는 computing 시간과 막대한 training data 를 필요로 한다.

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.

그러나 transfer learning 은 이미 연관된 task 에 대해서 train 되어 있고 모델에게 새로운 classification 을 더 작은 데이터셋으로 가르칠 수 있다. (==요약하면 이런 의미)

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.

너는 전체 모델(전체 네트워크에 대한 weight 를 조정한다는 의미)을 retraining 할 수 있지만, 간단하게 마지막 layer 를 제거해서 매우 정확한 결과를 classification 에서 얻을 수 있다. 그리고 새로운 class 를 인식하는 새 layer 를 맨 위에 얹어서 training 하면 된다.

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.

이 방식을 사용하기 위해서는, 충분한 training 데이터와 몇몇의 hyperparam 을 조절해 주면 너는 높은 정확도를 지닌 TF model 을 원샷원킬로 만들 수 있다.

Once you're happy with the model's performance, simply convert it to TensorFlow Lite and then compile it for the Edge TPU.

만약 너가 model 퍼포먼스에 만족하면 TF lite 모델을 Edge TPU 에 맞게 컴파일해라.

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).

그리고 transfer learning 을 하는 동안 모델 아키텍처는 변하지 않기 때문에 너는 Edge TPU 에 맞게 fully compile 해줘야 한다. (네가 edge tpu 에 호환 가능한 모델에서 시작한다고 가정하면)

== transfer learning 하는 동안에 모델의 아키텍쳐가 edge tpu 에 맞게 변하거나 하지 않음. 따라서 머 중간은 edge tpu 에 맞게 compile 해주고 이런 게 아니라 걍 싹 다 edge tpu 에 맞게 compile 해 줘야 됨. 조건은 애초에 edge tpu 에서 작동 되는 모델을 네가 transfer learnin 한다는 것임. 즉. TF lite 고 int8 로 퀀타이제이션 된 걸 의미할 듯. (빈 해석)

To get started without any setup, try our Google Colab retraining tutorials.

어떠한 setup 이라도 시작하고 싶다면 Google colab tutorial 를 시도해봐라.

All these tutorials perform transfer learning in cloud-hosted Jupyter notebooks.

모든 transfer learning 에 대한 튜토리얼은 cloud-hosted 쥬피터 노트북에 있다.

Transfer learning on-device

If you're using an image classification model, you can also perform accelerated transfer learning on the

Edge TPU.
네가 image classification 모델을 사용한다면, 너는 edge tpu 에서 가속화된 transfer learning 을 할 수 있다.
Our Edge TPU Python API offers two different techniques for on-device transfer learning:
우리의 edge tpu python api 는 on-device transfer learning 할 수 있도록 두 가지 다른 테크닉을 제공하고 있다.

- Weight imprinting on the last layer (ImprintingEngine)
- Backpropagation on the last layer (SoftmaxRegression)

In both cases, you must provide a model that's specially designed to allow training on the last layer.
두 케이스 모두, 너는 last layer 에 대해 training 할 수 있도록 특별히 디자인된 모델을 가지고 와야 한다.
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.
각 API 에 대해서(weight imprinting or backpropagation) 모델 스트럭쳐가 다를 수 있지만, 기본적으로 똑같다. (==걍 두 API 의 기본 base 원리는 똑같다는 말) classification 이 일어나는 마지막 fully connected 레이어를 base of the graph 에서 잘라낸다. (==classification 이 일어나는 마지막 FC 레이어 잘라서 transfer learning 할 꺼라는 의미)
Then only the base of the graph is compiled for the Edge TPU, which leaves the weights in the last layer accessible for training.
그렇게 하면 base of the graph 만 edge tpu 에 맞게 컴파일 되고, 마지막 레이어 가중치는 훈련용으로 사용할 수 있게 된다. (== 앞 문장에서 말한 FC layer 가중치 따로 빼서 훈련할 때 사용할 생각이라는 말)
More detail about the model architecture is available in the corresponding documents below.
모델 아키텍쳐에 대한 자세한 디테일은 아래 연관된 doc 을 보면 확인할 수 있다.

For now, let's compare how retraining works for each technique:
자 인제, 각 테크닉이 어떻게 retraining 하는 지 비교해 보자.

- 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.

Weight imprinting 은 base model 로부터 output(embedding vector)를 가져온다. L2-nomalize(유클리디안 distance 제곱)을 해서 activation vector 를 조정한다. 그리고 이 값을 final layer 의 새로운 weight 를 게산하는 데 사용한다. 이미 마지막 layer 에 있는 weight 로 새로운 vector 를 평균낸다. 이 방법은 새로운 class 를 적은 샘플 이미지로도 training 할 때 효과적이다. (==적은 이미지로도 training 할 수 있는 게 장점임)

- 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.

Backpropagation 은 기존의 backpropagation 의 요약 버전이다. 그래프의 모든 레이어에서 새로운 weight 를 backpropagtion 하지 않고, graph 의 마지막 단인 FC 레이어에서만 새로운 weight 를 업데이트한다. 이 방법은 높은 정확도를 보이는 일반적이 training 방법이지만, 앞의 방법에 비해서 더 많은 이미지와 training iteration 이 더 많이 필요하다. (==weight imprinting 보다 전통적인 방법인데, data 도 더 많이 필요하고 training 더 많이 돌려야 해서 시간 오래 걸림. 하지만 weight imprinting 보다 정확도 높음)

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

두 개 training techniques 중에 골라야 한다면, 아래 요소들을 고려하면 된다.

- **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 샘플 사이즈 : weight imprinting 은 너가 상대적으로 작은 training 샘플을 가지고 있을 때 효과적이다. 각 클래스(클래스는 5 개 정도 하는 게 제일 효과적임. API 에서는 최대 200 개 class 까지 학습할 수 있도록 API 제공하고 있음.)에 대해, 1~200 개의 샘플 이미지를 제공할 수 있으면 Weight imprinting 이 유리하다. 만약에 앞에 말한 개수보다 더 많은 수의 샘플을 너가 training 에 사용할 수 있으면, 너는 backpropagation 에서 더 높은 정확도를 얻을 수 있다.

==1~200 개 사이즈 샘플이면 weight imprinting 하고, 그 이상이면 backprogation 긔(빈 요약)

- **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.

Training 샘플의 다양성 : 너가 large intra-class variance(==데이터셋도 많고, 데이터셋 내에서 다양한 피쳐있다는 의미) dataset 를 가졌다면 Backpropagation 이 더 유리하다. 즉, 만약 주어진 클래스에 대해서 확실하게 다른 모습을 보이는 이미지라면, 예를 들면 각도나 사이즈 같이 다른 모습, backpropagtion 이 더 유리하다는 의미이다. 그러나 만약 너의 어플리케이션은 다양성이 낮은 곳에서 작동하는 환경이라면 그리고 너의 샘플 데이터가 little intra-class variance 를 가진다면 weight imprinting 이 더 유리하다.

== 데이터 셋 크고 다양하면 backpropagation, 아님 weight imprinting(빈 요약)

- **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.

새로운 클래스 추가 : weight imprinting 에서만 너가 training 을 시작하고 난 이후에도 새로운 클래스를 모델에 추가할 수 있다. 만약 너가 backpropagation 을 사용한다면 네가 training 을 시작하고 나고 새로운 클래스를 추가하려면 모든 클래스에 대한 학습을 다시 해야 한다. 추가로, weight imprinting 은 pre-trained 된 모델(edge tpu 모델을 변경하기 전에 train 된 모델 == edge tpu 에 맞게 컴파일 되기 전에 구글에서 제공하는 tf 모델을 의미하는 듯)로부터 class 를 retrain 한다.(==weight imprinting 은 이미 있는 class 에서 retrain 하기 때문에 중간에 새로운 class 끼워넣는 거 가능함) 반면에, backpropagation 은 모든 class 를 on-device 에서 learning 하는 걸 필요로 한다.
== weight imprinting 은 이미 있는 class 를 깔고 training 하기 때문에 중간에 새로운 거 끼워 넣을 수 있고, backpropagation 은 싹 다 갈면서 training 해서 중간에 멈추면 첨부터 해야 됨(빈 요약)

- **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).

모델 호환성 : backpropagation 은 더 많은 모델 아키텍쳐, "out of box",를 이용할 수 있다. 너는 pre-train 된 모바일넷과 인셉션 모델은 on-device 백프로파에서 사용될 수 있는 embedding extractor 로 변경할 수 있다. Weight imprinint 을 사용하려면, 너는 무조건 몇 개의 아주 특정된 layer 가 있는 모델을 사용해야 하고 on-device 트레이닝에 사용하기 전에 특정 방식으로 모델을 training 해야 한다.(현재, 우리는 적절하게 변경할 수 있는 mobilenet v1 버전을 제공하고 있다.)

In both cases, the vast majority of the training process is accelerated by the Edge TPU.
두 경우 모두, edge tpu 에서 대부분 모든 training 과정이 가속화 된다.
And when performing inferences with the retrained model, the Edge TPU accelerates everything except the final classification layer, which runs on the CPU.
그리고 retrain 된 모델로 inference 돌릴 때, final classification layer 는 cpu 에서 돌리고 나머지는 edge tpu 에서 돌린다.
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.
그러나 이 모델에서 아주 작은 부분인 마지막 layer 를 cpu 에서 돌리기 때문에 너의 inference 속도에는 크게 영향을 주지 않는다.
To learn more about each technique and try some sample code, see the following pages:
각 테크닉에 대해서 더 배우고 싶거나 몇 샘플 코드를 돌리고 싶으면 아래 페이지를 봐라.

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

Model requirements

If you want to build your own TensorFlow model that takes full advantage of the Edge TPU at runtime, it must meet the following requirements:

Edge tpu runtime 에서 full 이점을 가지는 TF model 를 만들고 싶으면, 아래 요구사항을 만족시켜야 한다.

- Tensor parameters are quantized (8-bit fixed-point numbers; int8 or uint8).

Tensor 파라미터는 퀀타이제이션 되어야 한다. (8-bit fixed point 숫자 : int8 / uint8)

- Tensor sizes are constant at compile-time (no dynamic sizes).

Tensor 사이즈는 compile 할 때 상수여야 한다. (dynamic 사이즈는 안된다)

- Model parameters (such as bias tensors) are constant at compile-time.

모델 파라미터(예를 들면 bias tensor)는 compile 할 때 상수여야 한다.

- 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.

텐서는 1,2,3 차원이어야 한다. 만약 텐서가 3 차이상이면, innermost 3 차원만 1 보다 큰 크기를 가질 수 있다. (== 3 차원 중에서 가장 안 쪽의 차원만 1 보다 큰 사이즈를 가질 수 있다)

- The model uses only the operations supported by the Edge TPU (see table 1 below).

Edge tpu 에서 서포트하는 모델 operation 은 아래 table 1 을 보아라.

Note: Although the Edge TPU requires 8-bit quantized input tensors, 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.

노트 : edge tpu 가 8 bit 로 quantizaed 된 인풋 텐서를 필요로하지만, 만약 edge tpu compiler 로 float input 을 넣어주면, compiler 가 네 그래프 처음에 quantize op 를 남긴다.(얘는 CPU 에서 돌아간다.) 따라서, input output 텐서가 float 여도 CPU 에서 바뀔꺼니가 괜찮다.

Supported operations

When building your own model architecture, be aware that only the operations in the following table are supported by the Edge TPU.

너만의 모델 아키텍쳐를 만들 때, 아래 테이블에 나온 것만 edge tpu 에서 작동된다는 것을 주의해라.

If your architecture uses operations not listed here, then only a portion of the model will execute on the Edge TPU, as described in the section below about compiling.

만약 너의 아키텍쳐가 아래 테이블에 없는 ㅇ걸 사용하면, 모델의 일부분만 edge tpu 에서 작동된다. 그 내용은 아래 compiling 섹션에 적혀 있다.

Note: When creating a new TensorFlow model, also refer to the list of operations compatible with TensorFlow Lite.

노트 : 새로운 TF 모델을 만들 때, TF lite 에서도 지원하는 operation 을 사용해야 한다.

<Table>

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

너는 edge tpu runtime 버전이 edge tpu compiler 랑 호환되는 지 확인해야 한다.

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.

네 모델을 quantizing 한다는 말은 모든 32-bit floating-point 숫자(weight 나 activation output 과 같은)를 8-bit fixed-point 숫자로 변경한다는 의미이다.

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.

이 과정은 모델을 더욱 작고 빠르게 만든다. 그리고 8-bit 표현이 정확도가 떨어질 지라도 NN 에서 inference 정확도에 크게 영향을 주지 않는다.

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

Edge tpu 에서 호환되기 위해서, 너는 quantization-aware training(추천함)하거나 full integer post-training quantization 을 해야 한다.

== 애초에 quantization 될 걸 생각해서 하거나 난중에 full 8-bit quantization 해야 edge tpu 에서 돌아간다.

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.

Quantization-aware training(for TF)는 가짜 quantization 노드를 NN 그래프 안에 넣어서 training 하는 동안 8-bit 값의 결과를 시뮬레이트 하는 것이다.

Thus, this technique requires modification to the network before initial training.

그리고, 이 테크닉은 initial training 하기 전에 네트워크를 수정해야 한다.

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.

이는 일반적으로, 8-bit weight 가 나중에 변환되기 보다는 학습되기 때문에, 모델은 (post-training quantization 에 비해) 더 낮은 정확도 값에 반응한다(==낮은 정확도에서도 classification 모델이라면

classification 을 할 수 있다). 그렇기 때문에 더 높은 정확도를 가진 모델을 결과로 낸다.
==일반적으로 8-bit int 가 32-bit float 보다 정확도가 떨어진다고 하는데, 애초에 training 하면서 8-bit 로 quantization 될 거 생각해서 하니까 정확도 안 떨어진다. 32-bit float 로 다 하고 난중에 quantization 하는 것보다 중간에 8-bit 로 quantization 할 거 생각해서 하니까 정확도가 올라간다. (빈 요약)

It's also currently compatible with more operations than post-training quantization.
또한, 위 방법은 현재 post-training quantization 보다 더 많은 operation 들이 호환된다.
==위 방법을 쓰면 post-training quatization 보다 더 많은 게 edge tpu 에서 작동한다고!(빈 요약)

Note: As of June, 2020, the quantization-aware training API for TensorFlow 2 is still new and we're testing compatibility with the Edge TPU. For now, we recommend you use TensorFlow 1 for quantization-aware training.
노트 : 2020 년도 6 월에는 TF 2 를 위한 quantization-aware training API 가 여전히 new 고 edge tpu 로 호환되는 지 테스트 중이었다. 현재는, 우리는 네가 quantization-aware training 을 TF1 에서 사용하기를 권장한다.

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.
Full integer post-training quantization 은 네트워크에 어떠한 수정도 필요없다. 따라서, 네트워크를 quantized model 로 변경할 때 사용할 수 있는 기술이다.

However, this conversion process requires that you supply a representative dataset.
하지만, 이 변경 과정은 네가 representative(대표적인) 데이터셋을 제공해야 한다.

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).
그 말은 즉슨, 너가 original training dataset(같은 데이터 범위에서)와 같은 포맷의 데이터셋이 필요하고 유사한 스타일의 데이터셋을 준비해야 한다.(이 말은 이전 training/evaluation 데이터에서 사용한 데이터를 사용할 수 있고, 대신 모든 클래스에 대해서 데이터를 준비할 필요는 없다.)

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.
Representative 데이터셋으로 activation 과 input 의 dynamic 범위를 측정할 수 있으며 각 weight 와 activation 값의 정확한 8-bit representation 을 찾는 데 중요하다.

However, not all TensorFlow Lite operations are currently implemented with an integer-only specification (they cannot be quantized using post-training quantization).
그러나, 모든 TF Lite operation 이 integer-only 사양에 맞게 구현되지 않는다. (그들은 post-training quantization 으로 quantization 되지 않는다.)

By default, the TensorFlow Lite converter leaves those operations in their float format, which is not

compatible with the Edge TPU.
디폴트로 TF Lite 컨버터는 edge tpu 에는 호환이 되지 않는 TF float 포맷으로 이러한 operations 을 남겨둔다.
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.
아래에 설명한 대로, edge tpu 컴파일러는 호환되지 않는 operation(quantization 안된 operation 과 같은 거)을 만나면, 이 operation 은 CPU 에서 실행된다.
So to enforce integer-only quantization, you can instruct the converter to throw an error if it encounters a non-quantizable operation.
Integer-only quantization 을 적용시키기 위해, 너는 non-quantization operation 이 생기면 error 코드를 띄우도록 converter 를 설정할 수 있다.
Read more about integer-only quantization.
Integet-only quantization 에 대해 더 읽어봐랏
For examples of each quantization strategy, see the following tutorials:
각 quantization 전략의 예제를 보고 싶으면 아래 튜토리얼을 참고해라.

- Retrain a classification model using quantization-aware training (for TensorFlow 1)
- Retrain a classification model using post-training quantization (for TensorFlow 1 and 2)

For more details about how quantization works, read the TensorFlow Lite 8-bit quantization spec.
어떻게 quantization 이 동작하는 세세하게 알고 싶으면 TensorFlow Lite 8-bit quantization 스펙을 봐랏.

Float input and output tensors

As mentioned in the model requirements, the Edge TPU requires 8-bit quantized input tensors.
Model requirements 에 적힌 대로, edge tpu 는 8-bit quantized input tensor 를 필요로 한다. (==이걸로 작동한다)
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).
그러나, 만약 edge tpu 가 내부적으로는 quantized 되었지만, float input 을 여전히 사용하는 모델을 넘겨받게 된다면, compiler 는 quantize operation 을 너의 그래프 처음에 남긴다. (이것은 CPU 에서 돌아가게 한다.)
Likewise, the output is dequantized at the end.
마찬가지로, output 은 마지막에 dequantized 된다.
So it's okay if your TensorFlow Lite model uses float inputs/outputs.
그렇다면, 너의 TF Lite 모델이 float input/output 을 사용해도 된다는 소리이다.
However, if you run an inference with the Edge TPU Python API, that API requires all input data be in

uint8 format.
그러나, 만약에 너게 edge tpu python api 를 이용해서 inference 를 돌리면, 모든 input data 는 uint8 형식이어야 한다.
You can instead use the TensorFlow Lite API, which provides full control of the input tensors, allowing you to pass your model float inputs—the on-CPU quantize op then converts the input to int8 for processing on the Edge TPU.

네가 input tensor 에 대해 full control(==uint8 말고도 다 작동 가능함) TF Lite API 를 (edge tpu python api 대신에) 사용한다면, 너는 모델에 float inputs 을 넣을 수 있다. On-CPU quantize operation 을 int8 로 변환해서 edge tpu 에서 처리한다. (==cpu 에서 uint8 로 quantization 때려서 edge tpu 에 넘겨준다는 소리인듯)
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).
그러나 네 모델이 float input 과 output 을 사용한다면 data format 변경(== float -> int8)때문에 약간의 latency 가 추가되는 점을, 그러나 대부분의 모델(input tensor 가 클수록 더 많은 latency 가 필요하다)에서는 무시할 만하다, 주의해라.
==float 를 int 로 바꾸는 데 latency 가 더 걸리니까 고려해 봐라. 근데 머 data format 변경에 그렇게 큰 latency 가 필요하지 않을꺼야 무시해도 될 듯.이라는 의미 (빈 요약)
So to achieve the best performance possible, we recommend fully quantizing your model so the input and output use int8 or uint8 data.
그래서 가장 좋은 performance 가 가증하려면 너는 fully quantizing 되는 걸 사용하고, 네 모델이 input 과 output 에 int8 과 uint8 데이터를 사용하면 된다.
Note: If you're using TensorFlow 2, the only way to quantize the input and output tensors is with post-training quantization in TensorFlow r2.3 or higher (see this tutorial). Currently, you cannot quantize the input/output tensors using quantization-aware training in TensorFlow 2.
노트 : 만약 네가 TF 2 를 사용한다면, post-training quantization 은 TF 2.3 혹은 그 이상(튜토리얼 봐라)에서만 input 과 output 텐서를 quantization 하는 방법밖에 없다.(==TF 2.3 이상에서 post quantization 해야 한다는 말) 현재, TF2 에서 quantization-aware training 을 사용해서 input/output 텐서를 quantize 할 수 없다.

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.
네 모델을 train 하고 TF Lite(quantization 도 하고)로 변경한 후에 마지막 단계는 edge tpu compiler 를

이용해서 컴파일 하는 것이다.

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.

만약 네 모델이 위 섹션에서 나열한 요구사항을 모두 충족하지 않으면, 머 여전히 컴파일은 되겠지만, 전부가 아니라 부분적으로 edge tpu 위에서 모델이 돌아갈 것이다.

At the first point in the model graph where an unsupported operation occurs, the compiler partitions the graph into two parts.

모델 그래프 중에서 unsupported operation 이 발생한 첫 지점에서, compiler 는 그래프를 두 파트로 나눈다.

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.

그래프의 첫 번째 파트는 edge tpu 에서 작동되도록 custom operation 으로 컴파일된 only supported operation 만 포함한다. (== 첫 번째 파트는 edge tpu 에서 작동되도록 support 하는 파트만 있다) 그리고 그 외 나머지 전부는 Fig 2 에서 말한대로 CPU 에서 동작한다.

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.

노트: 현재, edge tpu 컴파일러는 unsupported operation 과 마주하는 즉시, supported operation 이 나타난다 하더라도, 모델을 2 번 이상 나눌 수 없다. 나눈 후 operation 은 모두 CPU 에서 작동한다.

== edge tpu compiler 는 지원하지 않는 operation 이 나오는 즉시 모델을 두 개로 나눈다. 지원되는 부분까지는 edge tpu 에서 동작하고, 나머지는 다 cpu 에서 작동한다. 나머지 중에서 support 하는 operation 이 나올지라도 다시 edge tpu 에서 작동하는 게 아니라 계속 cpu 에서 작동한다.

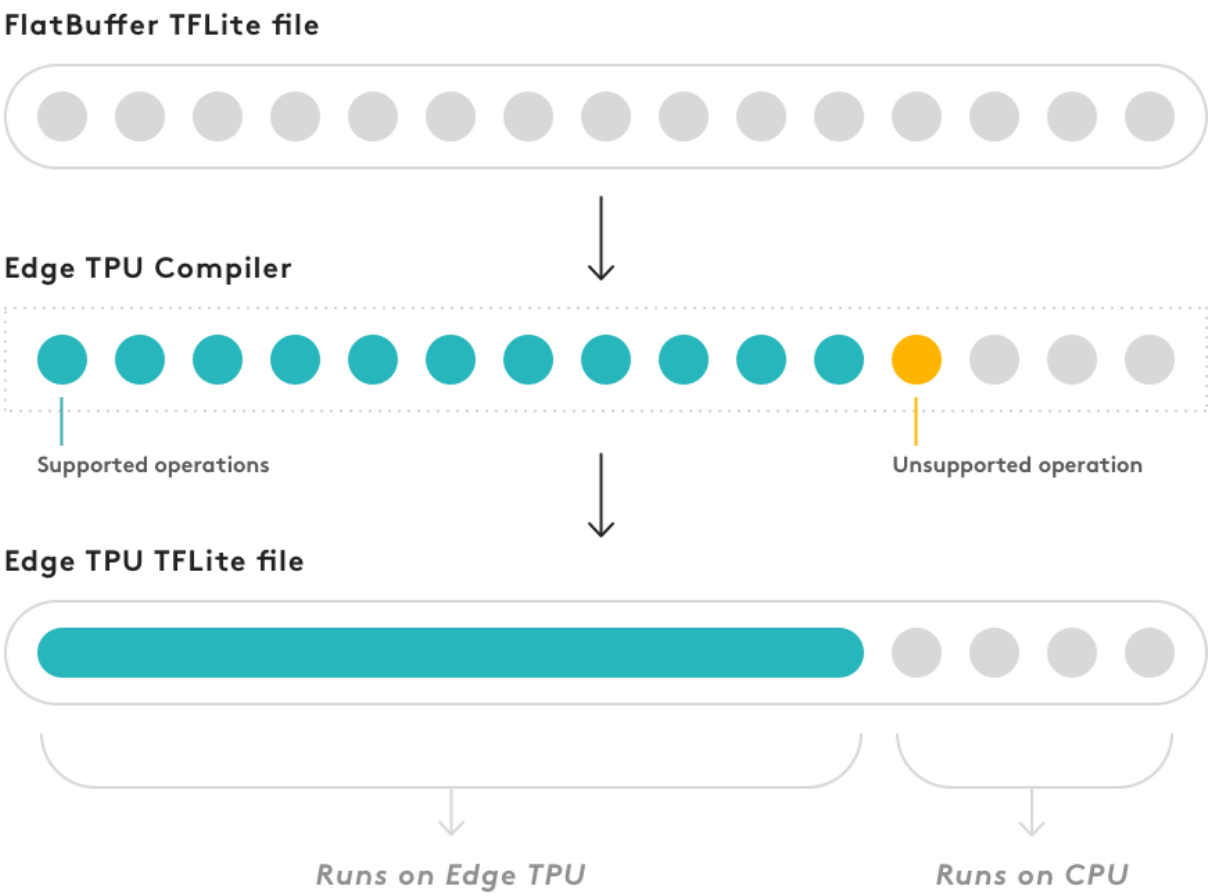

Figure 2. The compiler creates a single custom op for all Edge TPU compatible ops, until it encounters an unsupported op; the rest stays the same and runs on the CPU

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.

만약 네가 컴파일된 모델(visualize.py 와 같은 도구를 이용하여서)을 검사해도, 모델이 여전히 TF Lite 모델인 걸 확인할 수 있다. 단, 그래프의 처음에 custom operation 이 있다.

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.

이 custom operation 은 모델이 실제로 compiled 된 유일한 부분이고 edge tpu 에서 작동하는 모든 operation 을 포함하고 있다.

The rest of the graph (beginning with the first unsupported operation) remains the same and runs on the CPU.

나머지 그래프(첫 번째 unsupported operation 이 나온 처음)는 똑같이 남아 있고, CPU 에서 돌아간다.

== custom operation 이 나온 부분 말고는 edge tpu 에도 동작되게 compile 되어 있지 않는다. custom operation 이후부터는 compile 되어 있지도 않고, cpu 에서 돌아간다. (빈 요약)

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.

만약 네 모델의 일부가 CPU 에서 돌아간다면, 너는 모든 모델이 edge tpu 에서 돌아가는 것보다는 inference 속도가 느릴 것을 생각해야 한다.
== 당연히, 모든 모델이 edge tpu 에서 돌아가는 것보다 cpu 랑 edge tpu 랑 나눠서 돌아가는 게 더 느릴 것이다. 데이터를 전송해야 하니까(빈 요약)
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.
우리는 이 시뮬레이션에서 네 모델이 얼마나 느려질지는 예상 못한다. 그래서 너는 다른 아키텍쳐를 이용해서 실험하고 edge tpu 에 100 프로 호환되는 모델을 만들어야 얼마나 한다.
== 우리는 edge tpu 에서 100 프로 호환되는 모델과 아닌 모델이 얼마나 속도 차이가 나는 지 몰라. 네가 알아서 다른 아키텍쳐 만들어 보거나, 100 프로 호환되는 거 만들어 봐서 비교해 봐. (빈 요약)
That is, your compiled model should contain only the Edge TPU custom operation.
즉, 너의 컴파일된 모델은 edge tpu custom operation 만 포함해야 한다는 말이다.
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).
노트 : 컴파일이 완료되면, edge tpu compiler 가 너에게 얼마나 많은 operation 이 edge tpu 에서 굴러갈 수 있고 얼마나 많은 operation 이 CPU 에서(edge tpu 대신에) 돌아가는 지 알려준다.
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).
그런데, edge tpu 에서 돌아가는 operation 과 CPU 에서 돌아가는 operation 의 비율이 전체 performance 에 영향을 미치지는 않는다. 만약 너의 모델의 아주 작은 부분이 CPU 에서 돌아간다면 이게 inference speed 을 크기 순으로 낮출 수 있다. (모든 모델이 edge tpu 에서 돌아가는 버전에 비해서)
== edge tpu compiler 가 어디까지 edge tpu 에서 돌아가고 어디부터 cpu 에서 돌아가는 지 알려줌. 근데, 아주 조금만 cpu 에서 돌아가더라도 inference speed 늦출 것임.(우리는 얼마나 늦추는 지 모르니까 너게 알아서 여러 모델 만들고 100 프로 서포트 하는 edge tpu 모델 만들어서 테스트 해바) (빈 요약)