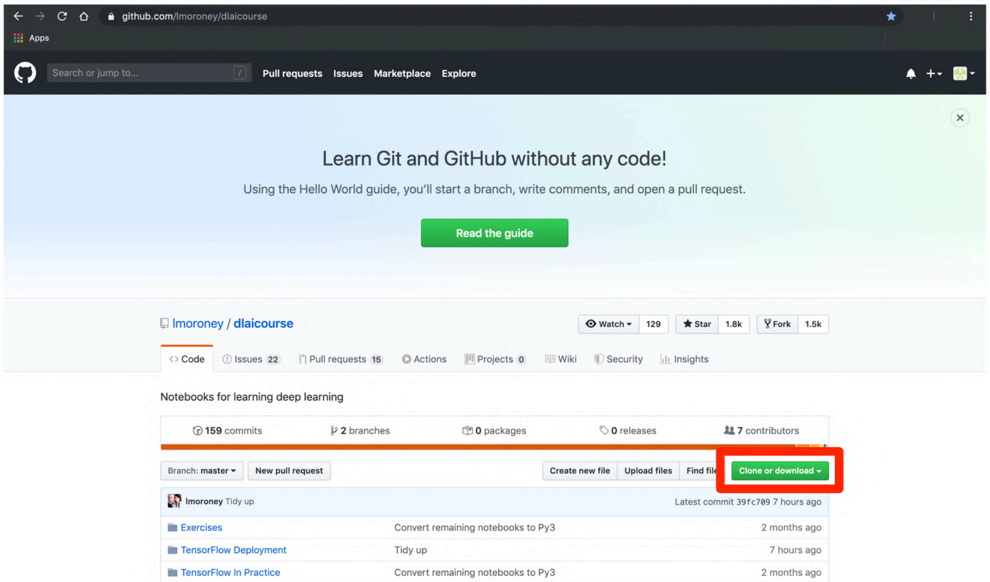
Downloading the Coding Examples and Exercises

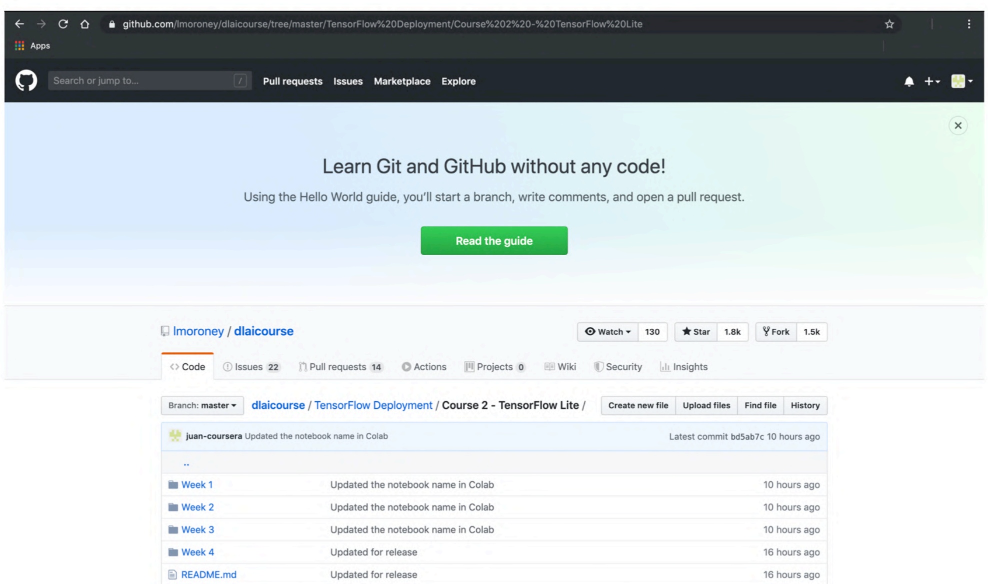
We have created this GitHub Repository where you can find all the examples and exercises not only for this course but for the entire TensorFlow for Data and Deployment Specialization .

You can download all the examples and exercises to your computer by cloning or downloading the GitHub Repository.



You can find the corresponding coding examples and exercises for this course in the following folder in the GitHub repository:

dlaicourse/TensorFlow Deployment/Course 2 - TensorFlow Lite/



Each folder contains the corresponding examples and exercises for each week of this course on TensorFlow Lite.

NOTE: The code in the repository is updated occasionally. Therefore the code in the repository may vary slightly from the one shown in the videos.

TensorFlow Lite

You may have heard of mobile models like mobile nets and how they're designed for the mobile platform. Their goal is to be lightweight working on small low power devices like phones, and they may not be as accurate as those which run on supercomputers in the Cloud.

Features

Lightweight

Low-latency

Privacy

Improved power consumption

Efficient model format

Pre-trained models

Features

TensorFlow Lite is a solution designed to run on devices with low latency and without the need for an Internet connection.

Lightweight

Low-latency

Privacy

Improved power consumption

Efficient model format

Pre-trained models

Features

You could avoid following one regime where it would involve taking a round trip to a model server. Since TFLite uses on-device ML to operate, there's absolutely no need for data to leave the device in sharing your privacy.

Lightweight

Low-latency

Privacy

Improved power consumption

Efficient model format

Pre-trained models

Features

Lightweight

Low-latency

Privacy

Improved power consumption

Efficient model format

Pre-trained models

It can also help improve power consumption as you might already be aware that network connections can tend to be very power-hungry.

Features

Lightweight

Low-latency

Privacy

Improved power consumption

Efficient model format

Pre-trained models

Models in TFLite are designed to have a small binary size with just a minor impact on accuracy.

Features

Lightweight

Low-latency

Privacy

Improved power consumption

Efficient model format

Pre-trained models

When it comes to availability of pre-trained models, TensorFlow Lite has just about everything you need for the most common machine learning tasks, as well as sample examples that you could try out just to see how a model would run on a mobile device.

To accomplish some of the other tasks, TensorFlow Lite comes with a utility that helps you convert TensorFlow models from their various formats into a special format that's consumable by TensorFlow Lite.

Components in TensorFlow Lite

The converter can be used for creating a TFLite model from various model formats and it runs on your model development and training environment. It allows you to optimize your models for optimal performance and even bring down the size of your model.

Converter (to TensorFlow Lite format)

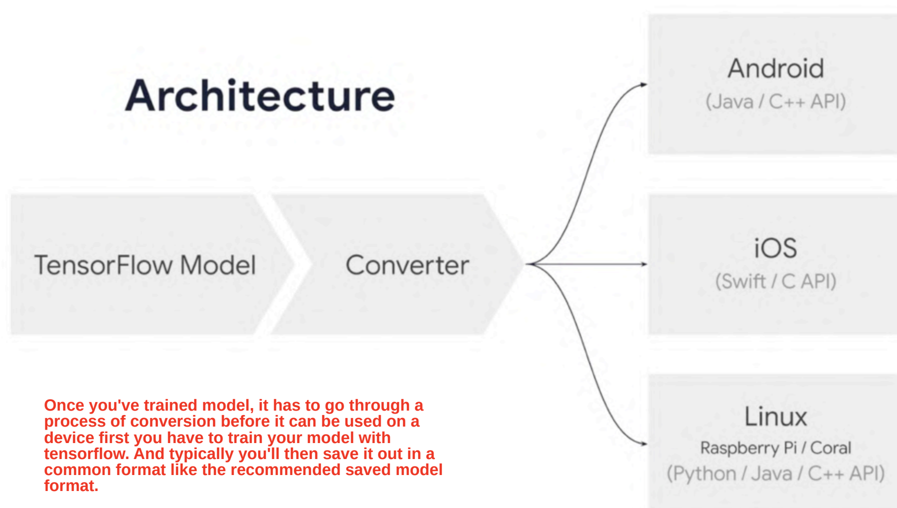
- Transforms TensorFlow models into a form efficient for reading by the interpreter
- Introduces optimizations to improve binary size model performance and/or reduce model size.

Interpreter Core

- Diverse platform support, (Android, iOS, embedded Linux and microcontrollers)
- Platform APIs for accelerated inference

The interpreter, which runs on your mobile device deals with the inference of these converted models. The interpreters core is responsible for executing these models and client applications using a reduced set of TensorFlow's operators. It uses a custom memory allocator, which is less dynamic to ensure minimal load, initialization, and execution latency. It also provides support for a wide range of devices both in mobile and IoT along with their hardware accelerated APIs.

Architecture



Once you've trained model, it has to go through a process of conversion before it can be used on a device first you have to train your model with tensorflow. And typically you'll then save it out in a common format like the recommended saved model format.

Once you have it you'll use the tensorflow light converter tools to flatten the model to prepare it for mobile or embedded devices.

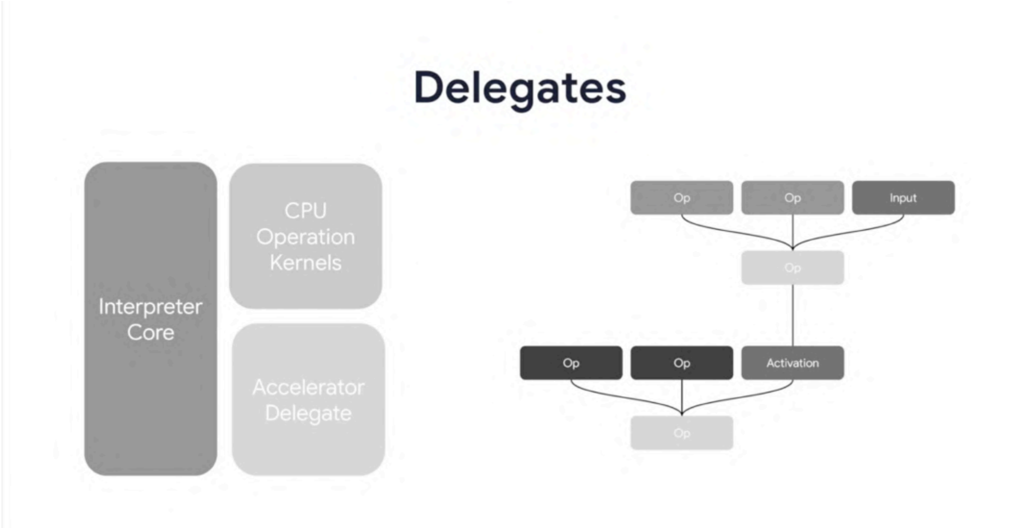
Performance

Running inference on compute heavy machine learning models on mobile devices is resource demanding due to a device has limited processing and power. So inference on these devices has to be performed very quickly to avoid overhead and make real-time applications possible for this purpose tents flow lights can employ Hardware acceleration libraries or apis for supported devices. You can actually evaluate whether your model benefits from using Hardware. Is available on

Acceleration	Available
Software	NN API (also a delegate)
Hardware	Edge TPU
	GPU
	CPU Optimizations (ARM and x86)

One way to improve inference on Android devices is by leveraging Androids neural network API and you'll learn how to use that for optimization later in the course.

Secondly inference can be boosted with HTTP use as their solely built for operating on deep learning models. This is not just limited to serving models, but also to training them they're also known to be high performing and have a low-power footprint while being pretty small in size.



Another form of acceleration which comes in tensorflow light is a tensor flow light delegate which is a way to pass your graph execution to Hardware that specialized to run inference for this tensorflow light provides go to support for an experimental GPU delegate that can be used to accelerate models on devices that have an available GPU.

GPUs are built for running many mathematical operations in parallel and that makes them perfect for ML inference.

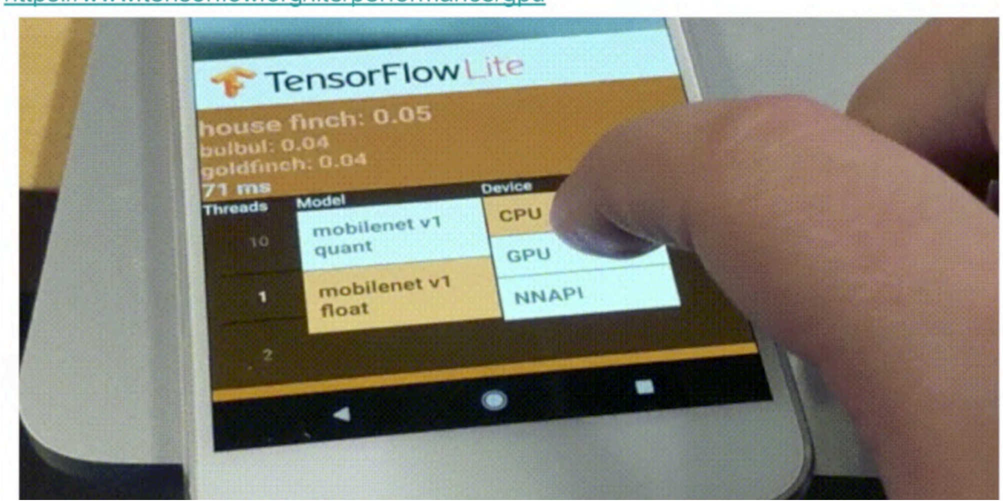
First a canonical representation of the network is built and then this undergoes a series of Transformations, like removing unnecessary Ops substituting an OP with one that has a faster implementation and coalescing Ops to avoid using more share programs and then just like video games compute. Shaders are generated on compiled based on this optimized graph using a Shader runtime. For Android this runtime is the opengl es and for iOS that uses metal.

You may come across certain downsides and using this however, as not every tensorflow op is included to be a part of the graph. However, the framework will automatically handle the delegation of Ops in the graph to the GPU or the CPU accordingly. Do note that the cost of switching can lead to higher latency, 's and your model might need to be a little bit bigger.

TensorFlow Lite, Experimental GPU Delegate (Coding TensorFlow):

https://www.youtube.com/watch?v=QSbAUxWfxQw

https://www.tensorflow.org/lite/performance/gpu



Techniques

- Quantization
- Weight pruning
- Model topology transforms
  - Tensor Decomposition
  - Distillation

Let's talk about optimization. This is necessary because of the generally limited resources on mobile and embedded devices. It's critical that deployed machine learning models have optimal model size low latency and power consumption. This is even more important on edge devices where resources are further constrained and model device and efficiency of computation can become a major concern.

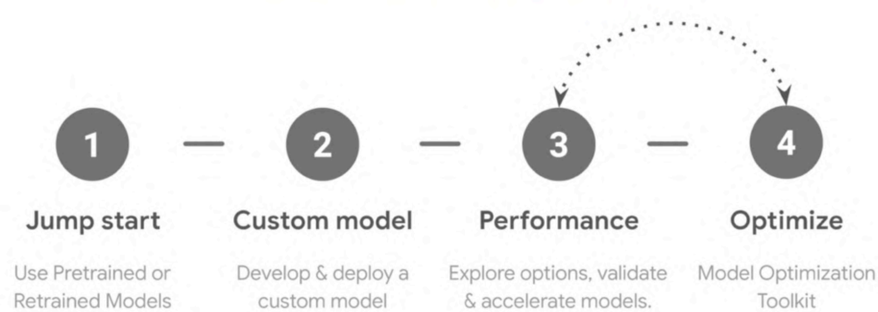
There are several methods that one can use to achieve these types of optimizations and these Glued quantization, which reduces the Precision of the numbers in the weights and biases of the model.

There's also wait pruning which reduces the overall number of parameters and model topology transforms whose goal is to convert the overall model topology to get a more efficient model to begin with.

Why Quantize?

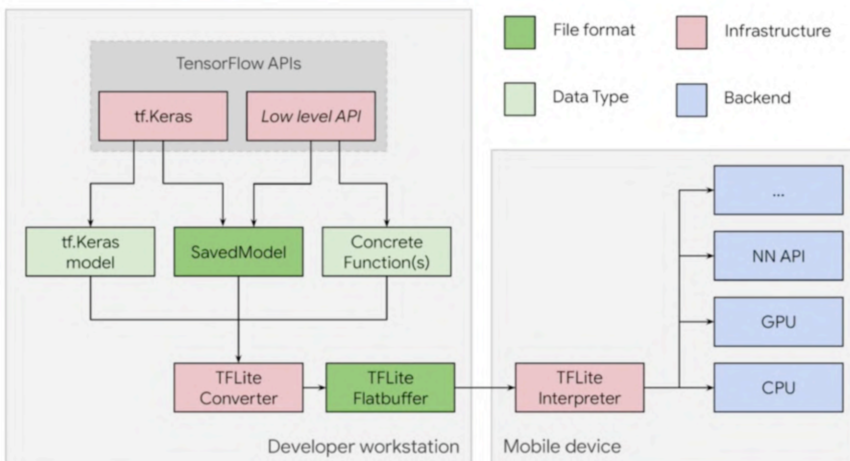
- All available CPU platforms are supported
- Reducing latency and inference cost
- Low memory footprint
- Allow execution on hardware restricted-to or optimized-for fixed-point operations
- Optimized models for special purpose HW accelerators (TPUs)

Putting it all together



Your workflow with tensorflow light is pretty straightforward. You can start with an existing model and convert it for TF light or you may even have a model that's already optimized will look at some of them over the next few weeks. You can take your custom models and TF light to converting them and then optimizing them for mobile performance.

TensorFlow Lite Converter



Parameters for conversion

Python API (preferred)

SavedModel

- The standard for serializing a TensorFlow model
- A MetaGraph to hold metadata
- Holds snapshot of the trained model (with model weights and computation)
- No model building code required
- Supports model versioning

Inspecting with SavedModel's CLI

To understand the interfaces of signatures of a SavedModel, we can call the SavedModel CLI script and get details about it with code like this.

```
!saved_model_cli show --dir /tmp/mobilenet/1 \
                      --tag_set serve \
                      --signature_def serving_default
```

Inspecting with SavedModel's CLI

```
The given SavedModel SignatureDef contains the following input(s):
  inputs['input_1'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 224, 224, 3)
      name: serving_default_input_1:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['act_softmax'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1000)
      name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

We can see that this model expects images to be input as 224 by 224 by 3. In other words, 24-bit color, 224 by 224 and their output is of shape 1,000 which tells me that it's classifying up to 1,000 classes.

Exporting a SavedModel from Keras

To export a SavedModel that was built with Keras or if it's one of the built-in ones, the process is as simple as calling tf.savedmodel.save, as you can see here. This will bundle all the weights as well as the model architecture. You can notice there's a safe path convention being followed which is used by TensorFlow Serving whether last path components, in this case, the number one is a version number for your model. It allows tools like TensorFlow Serving to pick the latest version by default to serve, thereby indicating its freshness.

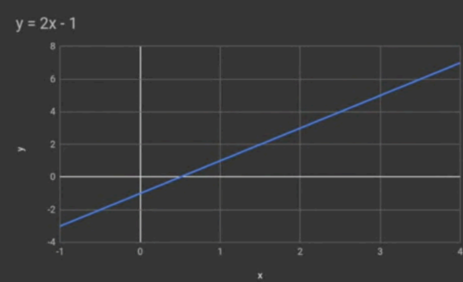
```
pretrained_model = tf.keras.applications.MobileNet()
tf.saved_model.save(pretrained_model, '/tmp/saved_model/1/')
```

Example 1

SavedModel to TFLite

```
import tensorflow as tf

# Store data for x and y
x = [-1, 0, 1, 2, 3, 4]
y = [-3, -1, 1, 3, 5, 7]
```




```
# Create a simple Keras model.
model = tf.keras.models.Sequential(
    [tf.keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(x, y, epochs=500)
```

Example 1

SavedModel to TFLite

```
import pathlib

# Export the SavedModel
export_dir = '/tmp/saved_model'
tf.saved_model.save(model, export_dir)

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()

# Save the model
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```

Example 1 SavedModel to TFLite

```
import pathlib

# Export the SavedModel
export_dir = '/tmp/saved_model'
tf.saved_model.save(model, export_dir)

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()

# Save the model
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```

Instantiate the TFLiteConverter from that saved model. Once you've done that, you simply call the convert method and you'll get the flattened version of the model that you can use what TensorFlow Lite

Example 1 SavedModel to TFLite

```
import pathlib

# Export the SavedModel
export_dir = '/tmp/saved_model'
tf.saved_model.save(model, export_dir)

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()

# Save the model
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```


Save out the TF Lite file by writing it to the file system. You now have a model that can be deployed to Android, iOS, or Edge systems.

Example 2 Keras to TFLite

If you want to use a pre-existing model, the process is very similar. So for example if I want to use a MobileNetV2 model that has been created for me, I can go through these steps

First, will be to load the model from tf.keras.applications initializing it with the weights that you want to use. In this case, image nets and the input shape for images that you'll classify. As it's a Keras model, you can save it in the H5 Keras formats with model.save.

```
import tensorflow as tf
import pathlib

# Load the MobileNet tf.keras model.
model = tf.keras.applications.MobileNetV2(weights="imagenet", input_shape=(224, 224, 3))
# Saving the model for later use by tflite_convert
model.save('model.h5')

# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```

Example 2

Keras to TFLite

```
import tensorflow as tf
import pathlib

# Load the MobileNet tf.keras model.
model = tf.keras.applications.MobileNetV2(weights="imagenet", input_shape=(224, 224, 3))
# Saving the model for later use by tflite_convert
model.save('model.h5')

# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```

Your model is an H5 Keras model, so you'll instantiate the TFLiteConverter using from Keras model and then convert it.

Example 2

Keras to TFLite

```
import tensorflow as tf
import pathlib

# Load the MobileNet tf.keras model.
model = tf.keras.applications.MobileNetV2(weights="imagenet", input_shape=(224, 224, 3))
# Saving the model for later use by tflite_convert
model.save('model.h5')

# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model
tflite_model_file = pathlib.Path('/tmp/foo.tflite')
tflite_model_file.write_bytes(tflite_model)
```


Command-line usage

If you don't have access to the Python code for generating the model, but you do have the saved model file, then the converter also works on the command line.

So if it's in saved model formats, you simply call tflite_convert and specify that your model is in a path using the saved model directory switch.

```
#!/usr/bin/env bash

# Saving with the command-line from a SavedModel
tflite_convert --output_file=model.tflite --saved_model_dir=/tmp/saved_model

# Saving with the command-line from a Keras model
tflite_convert --output_file=model.tflite --keras_model_file=model.h5
```

Command-line usage

If it's in Keras H5 format, you use the Keras model files switch instead.

```
#!/usr/bin/env bash

# Saving with the command-line from a SavedModel
tflite_convert --output_file=model.tflite --saved_model_dir=/tmp/saved_model

# Saving with the command-line from a Keras model
tflite_convert --output_file=model.tflite --keras_model_file=model.h5
```

Post-training quantization

- Reduced precision representation with 3x lower latency
- Little degradation in model accuracy
- Optimization modes
  - ↕ Default (both size and latency)
  - ↓ Size
  - ↓ Latency
- Efficiently represents an arbitrary magnitude of ranges
- Quantization target specification (FP32/INT8)

Image credits:
https://medium.com/tensorflow/introducing-the-model-optimization-toolkit-for-tensorflow-254aca1ba0a3

Post-training quantization

One simple method to achieve this is post-training quantization. In this case instead of quantizing a model during training and effectively changing your training code, you instead quantize as part of the process of converting the model to the TF-like formats. At its simplest, it converts all the floats in the weights of the model into ints. You will get much better latency which through experiments has been found to be up to about three times less with a relatively minor degradation in model accuracy.

The default behavior of the converter is to optimize for both size and latency, but you can override this in code.

So here's an example of where we override the default behavior of the converter to optimize primarily for size. You could also specify that you want to optimize for latency for improved performance or just leave it at the default where the converter will try to figure out the best balance between size and latency.

```python
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]

tflite_quant_model = converter.convert()
```

Post-training integer quantization

In some cases, for example, with Edge TPUs, the accelerators use only integers. For this, the optimization toolkit allows you to do post-training integer quantization, which makes models up to four times smaller. You can then further optimize by using calibration data where you run inference on a small set of inputs so as to determine the right scaling parameters to use when converting a model for integer quantization.

Post-training integer quantization

```
# Define the generator
def generator():
  data = tfds.load(...)
  for _ in range(num_calibration_steps):
    image, = data.take(1)
    yield [image]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

# Set the optimization mode
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Pass the representative dataset to the converter
converter.representative_dataset = tf.lite.RepresentativeDataset(generator)
```

Here's an example of converting a SavedModel to TensorFlow Lite with post-training integer quantization.

First we define a generator which is designed to generate samples from your data sets.

Post-training integer quantization

```
# Define the generator
def generator():
  data = tfds.load(...)
  for _ in range(num_calibration_steps):
    image, = data.take(1)
    yield [image]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

# Set the optimization mode
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Pass the representative dataset to the converter
converter.representative_dataset = tf.lite.RepresentativeDataset(generator)
```

We set the default optimization mode where it balances size in latency.

Post-training integer quantization

```
# Define the generator
def generator():
  data = tfds.load(...)
  for _ in range(num_calibration_steps):
    image, = data.take(1)
    yield [image]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

# Set the optimization mode
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Pass the representative dataset to the converter
converter.representative_dataset = tf.lite.RepresentativeDataset(generator)
```

Finally we pass our generator to the TF Lite converter as a representative data set. A representative data set is used for evaluating optimizations by recording dynamic ranges. This is done by running multiple inferences on a floating point TensorFlow Lite model using the user provided representative data set as an input.

We can then use the values log from inferences to determine the scaling parameters needed to execute all tensors of the model in integer arithmetic. This makes the model allow the activations to be quantized along with the weights. The resulting model will have as many quantized ops as possible.

For ops that don't have a quantized implementation, they'll fall back to float ones, thereby allowing the model to still take flow input and output for convenience.

Full-integer quantization

If you have ops that don't have quantized implementations, their floating values will be used automatically. This makes for conversions to occur smoothly while restricting deployments as special purpose accelerators that only support integers. So to support these devices that do not support floating point operations, we just tell the converter to only output integers, and this can be done by constraining the quantization target specification to TensorFlow lights INT eight built-in ops. Do note that if the converter comes across an operation which cannot be currently quantized, an error may be raised.

```
...
# Set the optimization mode
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_LATENCY]

# Pass the representative dataset to the converter
converter.representative_dataset = tf.lite.RepresentativeDataset(generator)

# Restricting supported target op specification to INT8
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
```

Learn more about supported ops:

https://www.tensorflow.org/lite/guide/ops_compatibility

TF-Select to overcome unsupported ops

TF-Select

https://www.tensorflow.org/lite/guide/ops_select

It's a small modifications are how you convert your models TF light. This is pretty much the same code you'd use for normal conversions attends flow lights. The only difference here is that you specify Target Ops to also include the set of tensorflow Select arms. It is an experimental feature at the time of recording this so check out more details at this URL.

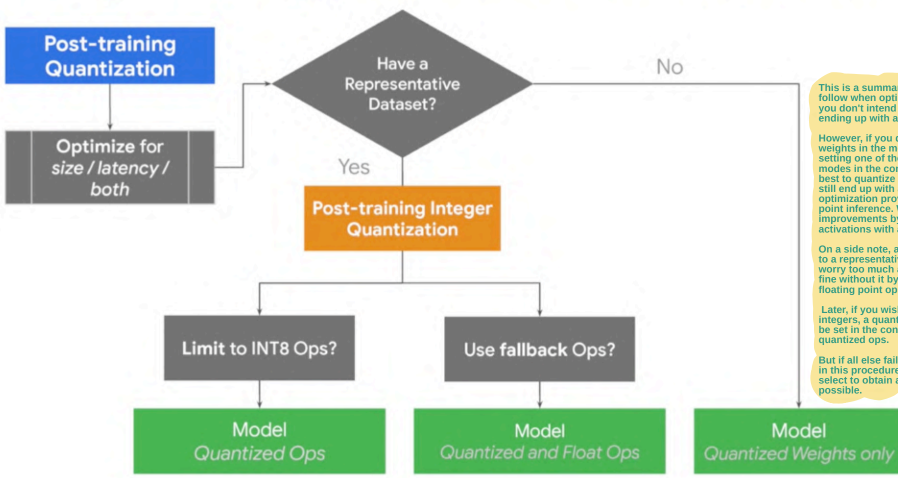
```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

converter.target_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,
                        tf.lite.OpsSet.SELECT_TF_OPS]

tflite_model = converter.convert()
```

Optimizing your models in a nutshell



This is a summary of the paths you're likely to follow when optimizing your TF Lite models. If you don't intend to quantize your model, you'll be ending up with a floating point model.

However, if you do wish to quantize only the weights in the model, you can do so just by setting one of the post-training optimization modes in the converter. The converter will do its best to quantize all the ops, but your model may still end up with a few floating point ops. This optimization provides latencies close to fully fixed point inference. We can further get latency improvements by quantizing both the weights and activations with a representative dataset.

On a side note, at times you may not have access to a representative dataset, and you don't have to worry too much about it as you can get along just fine without it by having a model that has some floating point ops.

Later, if you wish to have the model only output integers, a quantization target specification can be set in the converter to generate a model with quantized ops.

But if all else fails and you're encountering errors in this procedure, then you might have to use TF select to obtain a model with as many ops as possible.

TensorFlow Lite Interpreter in Python

```
# Load TFLite model and allocate tensors
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Point the data to be used for testing and run the interpreter
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
```

One really nice feature is the ability to test your model using Python on your developer workstation so you don't need to deploy it to a mobile and embedded system before you can start using it.

You'll start by loading your TensorFlow Lite model and allocating tensors like this.

TensorFlow Lite Interpreter in Python

```python
# Load TFLite model and allocate tensors
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Point the data to be used for testing and run the interpreter
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
```

Then we'll extract the input and output tensors for the model.

TensorFlow Lite Interpreter in Python

```python
# Load TFLite model and allocate tensors
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Point the data to be used for testing and run the interpreter
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
```


We can then set the input tensor with some valid data and invoke the interpreter to run inference on it

TensorFlow Lite Interpreter in Python

```python
# Load TFLite model and allocate tensors
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Point the data to be used for testing and run the interpreter
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
```


Reads the results by looking at the output tensor

Running models

Pretrained models

- Image classification
- Object detection
- Smart reply
- Pose estimation
- Segmentation

TensorFlow Hub

- Classification modules
- Feature vector modules
- Embedding modules

Not what you're looking for?

Build a custom model!

Getting a basic model running

Get started

1. **Build a model** — Create a simple model for (y = 2x - 1) from simulated data and train it
2. **Export & Convert** — Generate the SavedModel and convert it to TFLite
3. **Verify** — Perform evaluation on random data to verify the results
4. **Deploy** — Deploy the converted model on a mobile device (Android/iOS)

Getting a basic model running

Get started

1. **Build a model** — Create a simple model for (y = 2x - 1) from simulated data and train it
2. **Export & Convert** — Generate the SavedModel and convert it to TFLite
3. **Verify** — Perform evaluation on random data to verify the results
4. **Deploy** — Deploy the converted model on a mobile device (Android/iOS)

Getting a basic model running

Get started

1 Build a model

Create a simple model for (y = 2x - 1) from simulated data and train it

2 Export & Convert

Generate the SavedModel and convert it to TFLite

3 Verify

Perform evaluation on random data to verify the results

4 Deploy

Deploy the converted model on a mobile device (Android/iOS)

Getting a basic model running

Get started

1 Build a model

Create a simple model for (y = 2x - 1) from simulated data and train it

2 Export & Convert

Generate the SavedModel and convert it to TFLite

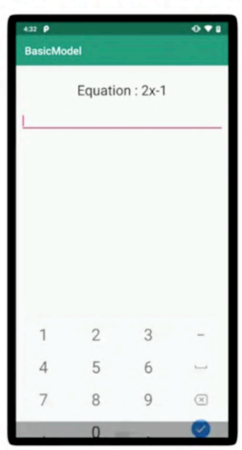
3 Verify

Perform evaluation on random data to verify the results

4 Deploy

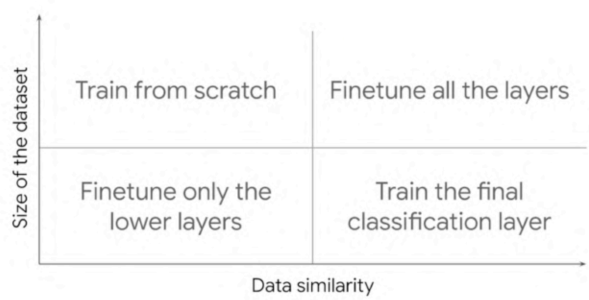
Deploy the converted model on a mobile device (Android/iOS)

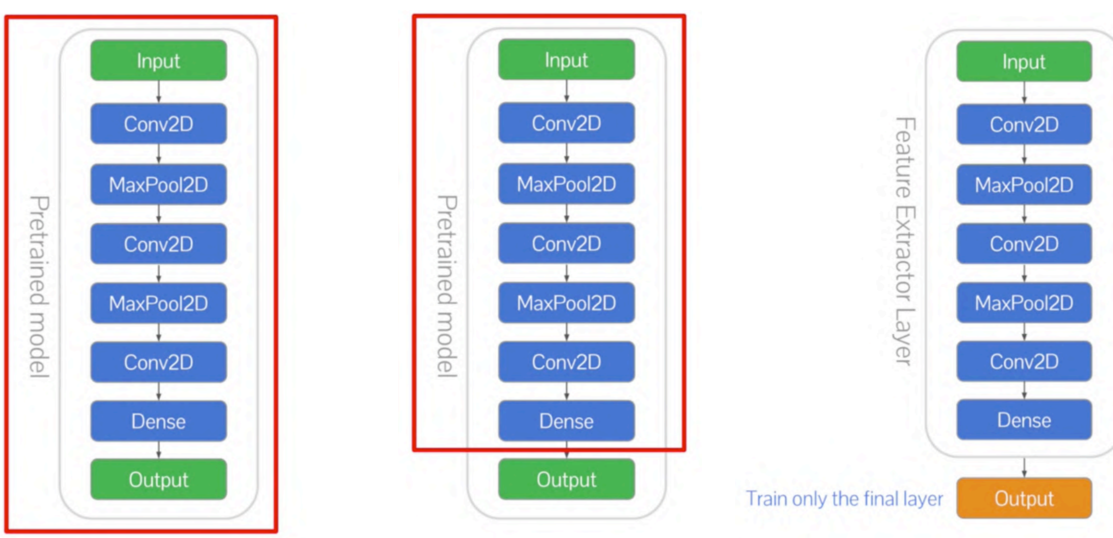
Android

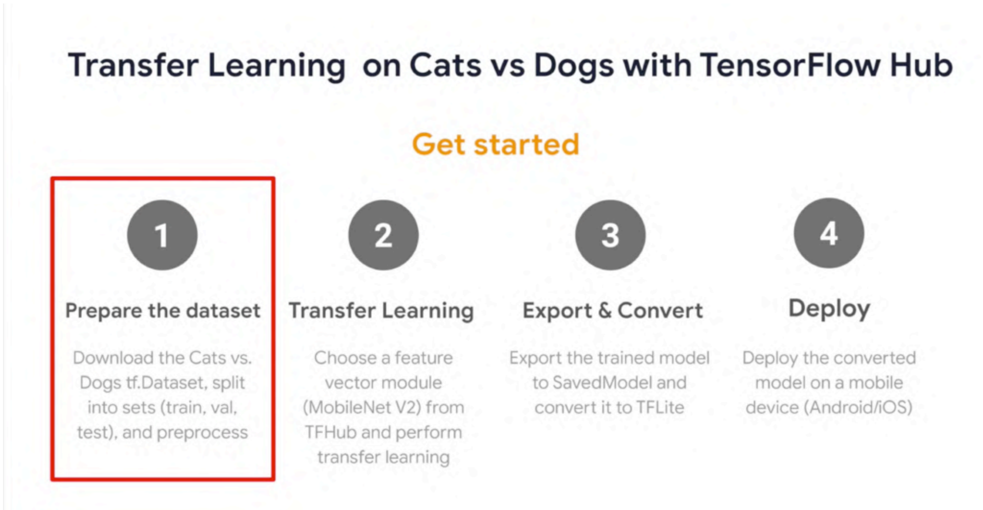


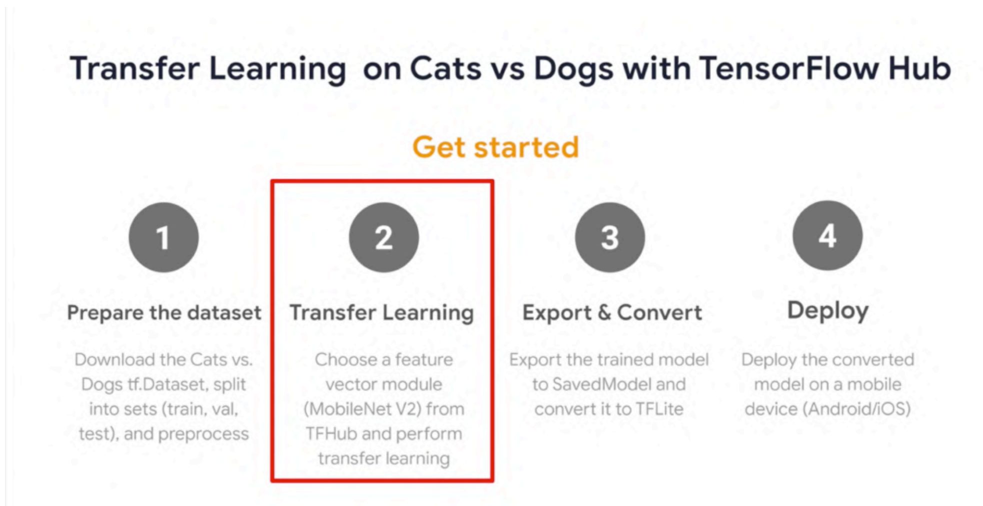
iOS

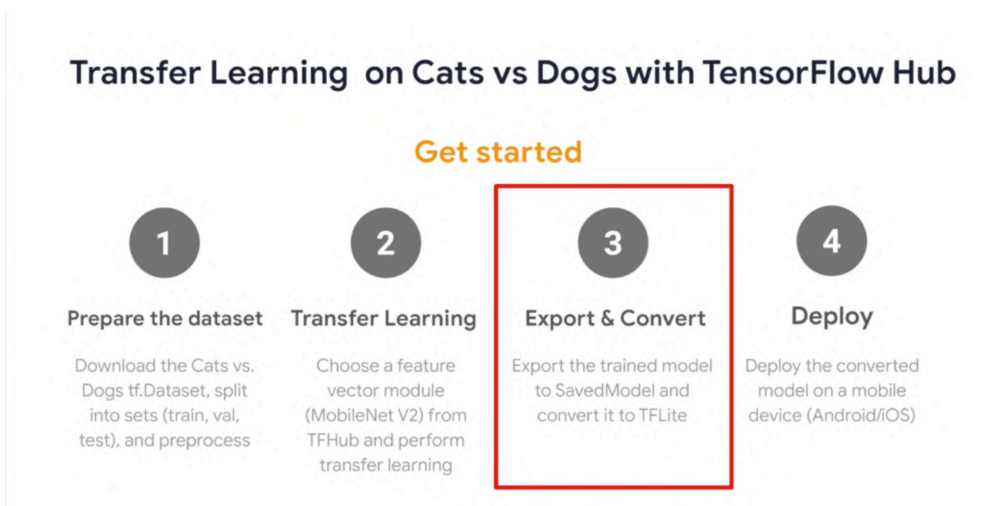
Transfer Learning

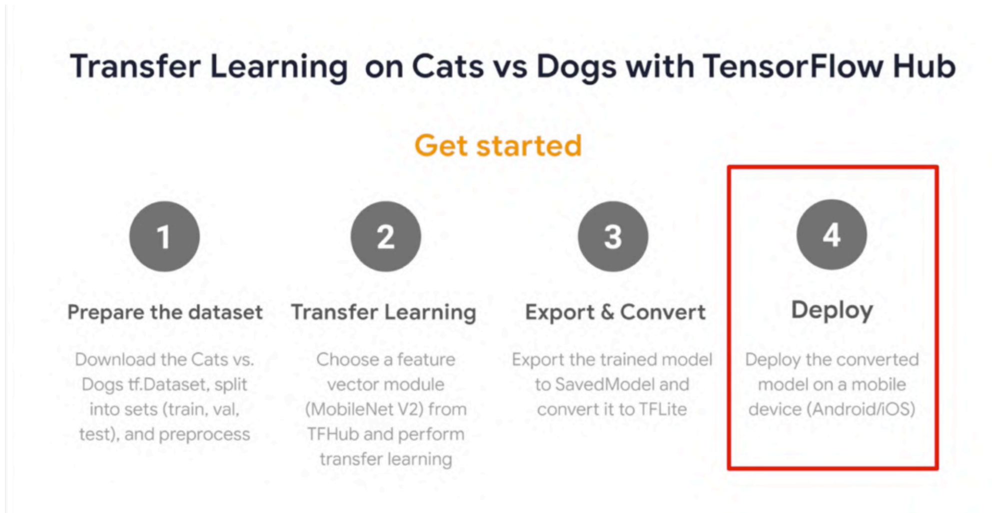





Transfer Learning on Cats vs Dogs with TensorFlow Hub
Get started
1
Prepare the dataset
Download the Cats vs. Dogs tf.Dataset, split into sets (train, val, test), and preprocess
2
Transfer Learning
Choose a feature vector module (MobileNet V2) from TFHub and perform transfer learning
3
Export & Convert
Export the trained model to SavedModel and convert it to TFLite
4
Deploy
Deploy the converted model on a mobile device (Android/iOS)


Transfer Learning on Cats vs Dogs with TensorFlow Hub
Get started
1
Prepare the dataset
Download the Cats vs. Dogs tf.Dataset, split into sets (train, val, test), and preprocess
2
Transfer Learning
Choose a feature vector module (MobileNet V2) from TFHub and perform transfer learning
3
Export & Convert
Export the trained model to SavedModel and convert it to TFLite
4
Deploy
Deploy the converted model on a mobile device (Android/iOS)


Transfer Learning on Cats vs Dogs with TensorFlow Hub
Get started
1
Prepare the dataset
Download the Cats vs. Dogs tf.Dataset, split into sets (train, val, test), and preprocess
2
Transfer Learning
Choose a feature vector module (MobileNet V2) from TFHub and perform transfer learning
3
Export & Convert
Export the trained model to SavedModel and convert it to TFLite
4
Deploy
Deploy the converted model on a mobile device (Android/iOS)


Transfer Learning on Cats vs Dogs with TensorFlow Hub
Get started
1
Prepare the dataset
Download the Cats vs. Dogs tf.Dataset, split into sets (train, val, test), and preprocess
2
Transfer Learning
Choose a feature vector module (MobileNet V2) from TFHub and perform transfer learning
3
Export & Convert
Export the trained model to SavedModel and convert it to TFLite
4
Deploy
Deploy the converted model on a mobile device (Android/iOS)