O'REILLY®

Building Mobile Applications with TensorFlow

Pete Warden

Strata
DATA CONFERENCE

Make Data Work
strataconf.com

Data is driving business transformation. Presented by O'Reilly and Cloudera, Strata puts cutting-edge data science and new business fundamentals to work.

- Learn new business applications of data technologies
- Get the latest skills through trainings and in-depth tutorials
- Connect with an international community of data scientists, engineers, analysts, and business managers

Job # D4211

Building Mobile Applications with TensorFlow

Pete Warden

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Building Mobile Applications with TensorFlow
by Pete Warden

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Shannon Cutt
Production Editor: Colleen Cole
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2017: First Edition

Revision History for the First Edition
2017-07-27: First Release

978-1-491-98842-8

[LSI]

Table of Contents

Building Mobile Apps with TensorFlow

Deep learning is an incredibly powerful technology for understanding messy data from the real world. TensorFlow was designed from the ground up to harness that power inside mobile applications on platforms like Android and iOS. In this guide, I'll show you how to integrate it effectively.

Challenges of Building a Mobile App with TensorFlow

This guide is for developers who have a TensorFlow model successfully working in a desktop environment and who want to integrate it into a mobile application. Here are the main challenges you'll face during that process:

- Understanding the basics of TensorFlow
- Building TensorFlow for your platform
- Integrating the TensorFlow library into your application
- Preparing your model file for mobile deployment
- Optimizing for latency, RAM usage, model file size, and binary size
- Exploring quantized calculations

In this guide, I cover all of these areas, with detailed breakdowns of what you need to know within each chapter.

Understanding the Basics of TensorFlow

In this section, we'll look at how TensorFlow works and what sort of problems you can use it to solve.

What Is TensorFlow?

It's recently become possible to solve a range of problems across a wide variety of domains using large neural networks. TensorFlow is a framework that lets you train and deploy these networks. It was originally created by Google as its main internal tool for deep learning, but it's also available as open source with a large and active community.

The models (also known as graphs) are descriptions of neural networks. They consist of a series of operations, each connected to some other operations as inputs and outputs. TensorFlow helps you construct these models, train them on a dataset, and deploy them where they're needed. What's special about TensorFlow is that it's built to support the whole process, from researchers building new models to production engineers deploying those models on servers or mobile devices.

This guide focuses on the deployment process, since there's more documentation available already for the research side. The most common use case in production is that you have a pretrained model that shows promising results on test data, and you want to integrate it into a user-facing application. There are less-common situations in which you want to do training in production, but this guide won't cover those.

The process of taking a trained model and running it on new inputs is known as *inference*, or *prediction*. Inference is particularly interesting because the computational requirements scale up with the numbers of users an application has, whereas the demands of training only scale with the number of researchers. As more uses are found for deep learning, the inference compute workload grows much more quickly than training. It also has a lot of opportunities for optimization, since the model you're running is known ahead of time, and the weights are fixed.

The guide is specifically aimed at mobile and embedded platforms, since those are the environments most different from the kinds of

machines that training is normally done on. However, many of the techniques also apply to the process of deploying on servers.

Mobile AI applications need to be small, fast, and easy to build to be successful. Here I'll be explaining how you can achieve this on your platform with TensorFlow.

What Level of Knowledge Do You Need?

There are examples in this guide that don't require any machine learning experience, so don't be put off if you're not a specialist. You'll need to know a bit more once you start to deploy your own models, but even there we hope that the demands won't be overwhelming.

What Hardware and Software Should You Have?

TensorFlow runs on most modern Linux distributions, Windows 10, and macOS. The easiest way to run the examples in this guide is to install Docker and boot up the official TensorFlow image by running:

```
docker run -it -p 8888:8888 tensorflow/tensorflow
```

This method does have the disadvantage that any local files (such as compilation caches) are lost when you close the container. If you're running outside of Docker, we recommend using `virtualenv` to keep your Python dependencies clean.

Some of the scripts require you to compile TensorFlow, so you'll need more than the pip install to work through all the sample code.

In order to try out the mobile examples, you'll need a device set up for development, using Android Studio for Android, or Xcode for iOS.

What Is TensorFlow Useful for on Mobile?

Traditionally, deep learning has been associated with data centers and giant clusters of high-powered GPU machines. So why does it make sense to run it on mobile devices? The key driver is that it can be very expensive and time-consuming to send all the data a device has access to across a network connection. Deep learning also makes it possible to deliver very interactive applications, in a way that's not possible when you have to wait for a network round-trip.

In the rest of this section, we cover common use cases for on-device deep learning. Chances are good you'll find something relevant to the problems you're trying to solve. We also include links to useful models and papers to give you a head start on building your solutions.

Speech recognition

There are a lot of interesting applications that can be built with a speech-driven interface, and many require on-device processing. Most of the time a user isn't giving commands, so streaming audio continuously to a remote server is a waste of bandwidth—you'd mostly record silence or background noises. To solve this problem, it's common to have a small neural network running on-device, listening for a particular keyword. When that keyword is spotted, the rest of the conversation can be transmitted over to the server for further processing if more computing power is needed.

Image recognition

It can be very useful for a mobile app to be able to make sense of a camera image. If your users are taking photos, recognizing what's in those photos can help you apply appropriate filters or label them so they're easily findable. Image recognition is important for embedded applications, too, since you can use image sensors to detect all sorts of interesting conditions, whether it's spotting endangered animals in the wild or reporting how late your train is running.

TensorFlow comes with several examples of how to recognize types of objects inside images, along with a variety of different pretrained models, and they can all be run on mobile devices. I recommend starting with the "TensorFlow for Poets" codelab.

This example shows how to take one of the pretrained models and run some very fast and lightweight "fine-tuning" training to teach it to recognize objects that you care about. Later in this guide, we show how to use the model you've generated in your own application.

Object localization

Sometimes it's important to know where objects are in an image as well as what they are. There are lots of ways augmented reality can be used in a mobile application, for example, guiding users to the

right component when offering them help fixing their wireless network, or providing informative overlays on top of landscape features. Embedded applications often need to count objects that are passing by, whether it's pests in a field of crops or people, cars, and bikes going past a street lamp.

TensorFlow offers a pretrained model for drawing bounding boxes around people detected in images, together with tracking code to follow them over time. The tracking is especially important for applications in which you're trying to count how many objects are present over time, since it gives you a good idea when a new object enters or leaves the scene, such as in this Android example.

Gesture recognition

It can be very useful to be able to control applications with hand or other gestures, either recognized from images or through analyzing accelerometer sensor data. Creating those models is beyond the scope of this guide, but TensorFlow is an effective way of deploying them.

Optical character recognition

There are multiple steps involved in recognizing text in images. You first have to identify the areas where the text is present, which is a variation on the object localization problem, and can be solved with similar techniques. Once you have an area of text, you interpret it as letters and then use a language model to help guess what words those letters represent. The simplest way to estimate what letters are present is to segment the line of text into individual letters, and apply a simple neural network to the bounding box of each one.

You can get good results with the kind of models used for MNIST, which you can find in TensorFlow's tutorials, although you may want a higher-resolution input.

A more advanced alternative is to use an LSTM model to process a whole line of text at once, with the model itself handling the segmentation into different characters.

Translation

Translating from one language to another quickly and accurately, even if you don't have a network connection, is an important use case. Deep networks are very effective at this sort of task, and you

can find descriptions of a lot of different models in the literature. Often these are sequence-to-sequence recurrent models in which you're able to run a single graph to do the whole translation without needing to run separate parsing stages.

Google Translate's live camera view is a great example of how effective interactive on-device detection of text can be (view "Google Translate vs. 'La Bamba'").

Text classification

If you want to suggest relevant prompts to users based on what they're typing or reading, it can be very useful to understand the meaning of the text. This is where *text classification* comes in, an umbrella term that covers everything from sentiment analysis to topic discovery. You're likely to have your own categories or labels that you want to apply, so the best place to start is with an example like SkipThoughts, and then train on your own examples.

Voice synthesis

A synthesized voice can be a great way of giving users feedback or aiding accessibility, and recent advances such as WaveNet show that deep learning can offer very natural-sounding speech. The technology is still in the process of moving from research into production-ready models (the computational requirements of WaveNet are currently too large for phones, for example), but we expect to see this happen over the next year.

How Does It Fit with the Cloud?

These examples of use cases show how well on-device networks complement cloud services. There are a lot of advantages to running on remote servers: you have a large amount of computing power available, and the deployment environment is completely controlled by you. Running on devices means you can offer higher interactivity than network round trips allow, you can offer the user an experience even there's a slow or missing data connection, and you can scale up the computation you do based on the number of users without having to purchase additional servers.

Enabling on-device computation actually boosts the amount of work you end up doing on the cloud. A good example of this phenomenon is hotword detection in speech. Since devices are able to constantly listen for keywords, the process triggers a lot of traffic to cloud-based speech recognition once one is recognized. Without the on-device component, the whole application wouldn't be feasible. We see this pattern across a lot of other applications, as well. Recognizing that some sensor input is interesting enough for further processing makes a lot of intriguing products possible.

What Should You Do Before You Get Started?

Once you have an idea of the problem you want to solve, you need to make a plan to build your solution. The most important first step is making sure your problem is actually solvable, and the best way to do that is to mock it up using humans in the loop. For example, if you want to drive a robot toy car using voice commands, try recording some audio from the device. Play it back to see if you can make sense of what's being said. Often you'll find there are problems in the capture process, such as the motor drowning out speech or not being able to hear at a distance. You should tackle these problems before investing in the modeling process.

Another example might involve showing people photos taken from your app to see if they can classify what's in the photos in the way you're expecting. If they can't (for example, when trying to estimate calories in food from photos, because all white soups look the same), you need to redesign your experience to cope with that uncertainty. A good rule of thumb is that if a human can't handle the task, it will be hard to train a network to do better.

After you've solved any fundamental issues with your use case, create a labeled dataset to define what problem you're trying to solve. This step is extremely important, even more than picking which model to use. You want it to be as representative as possible of your actual use case, since the model will only be effective at the task you teach it.

It's also worth investing in tools to make labeling the data as efficient and accurate as possible. For example, if you're able to switch from the necessity to click a button on a web interface to simple keyboard shortcuts, you may be able to speed up the generation process a lot. We also recommend doing the initial labeling yourself so that you can learn about the difficulties and likely errors and then adjust your labeling or data capture process to avoid them. Once you and your team are able to consistently label examples (that is, once you generally agree on the same labels for most examples), you can then try and capture your knowledge in a manual and teach external raters how to run the same process.

The next step is to pick an effective model to use. If you're lucky, there's already a trained model out there you can do fine-tuning on, see "What Is TensorFlow Useful for on Mobile?" on page 3; there might be something similar to what you need at TensorFlow's model garden. Lean toward the simplest model you can find, and try to get started as soon as you have even a small amount of labeled data, since you'll get the best results when you're able to iterate quickly. The shorter the amount of time it takes to train a model and run it in the real application, the better overall results you'll see.

It's common for an algorithm to get great training accuracy numbers but fail to be useful within a real application because there's a mismatch between the dataset and real usage. To combat such a mismatch, it's vital to prototype end-to-end usage as soon as possible. I like to say that the only metric that matters is app store ratings, since the user experience is the end goal for everything we're doing.

Common Model Patterns

There are lots of different ways of using neural network models to solve problems, and it's worth having a high-level understanding of some of these patterns to help you decide how to integrate a model into your application:

Bare model
: If the data you want to work with comes as a plain array of numerical values, and all you want as output is the same, you may just be able to use a model on its own, with minimal pre- or post-processing. A good example of this is image recognition, where the input is an array of pixel values, and the result is a score for each category. You need to resize the input to fit the expected dimensions the model was trained with, scale the pixel values to a float range instead of 0 to 255, and then sort the outputs to find the highest score. Other than that, getting useful results requires very little additional code beyond running the model.

Sliding window
: If you want to find the location of an object in an image, the simplest way is to run many small tiles of the image through a neural network, at many possible locations, and pick the tiles that give the highest response as the likely location. Because you can think of this as running a rectangular window from left to right in rows, gradually moving downward, this is known as *sliding window*. It's effective, but because there are so many windows needed to cover a typical image, it's often far too expensive to use in practical applications. The numbers grow even more if you're looking for variably sized objects, or those at different rotations, since you have to run the same exhaustive search with all those possible parameters, too.

Box proposals
: To optimize the sliding window approach, you can use some other technique to come up with proposals for likely boxes to test, rather than going through every possible combination. These proposals are often generated using traditional computer vision techniques. A big advantage of this approach compared to sliding windows is that you can trade off sensitivity with latency by varying the number of box proposals fed into the model.

Cascades
: Another common optimization pattern is to use a simple and cheap neural network to decide whether it's worth running a more costly model on the input. For example, a system detecting a spoken hotword might run a fast but inaccurate model continuously on all the audio that comes in, and only invoke a

more precise and battery-draining model when it thinks there's a good chance the audio does represent the wanted word. This can also be applied to image models, where you might run an extremely cheap model across the whole image with a sliding window, and then only do an in-depth analysis of tiles that scored higher than a threshold. One disadvantage of this approach is that the latency of the system is unpredictable, since it depends on how many of the early models trigger more processing. If you imagine a face-detecting cascade that is run on a photo of a large crowd, you can see how it would end up doing dramatically more work than on a landscape without people. This is something you need to watch out for if you're planning an application, since large delays can lead to a very poor user experience if they're unexpected.

Single-shot detection
: It's also possible to produce both a class and bounding box from running a single model, in a similar way to the basic image classification approach. You can look at examples like the Android person detector or YOLO to see how this works. It's conceptually a lot simpler than other localization approaches, since it only involves running a single model; but it can struggle to achieve the same results as other techniques when there are a lot of classes involved. It's an active area of research, though, so keep an eye on the latest model releases to see what improvements are available.

Visualizations
: The success of deep learning for vision problems has led to a lot of other domains recasting their tasks as ones that can be tackled by image networks. For example, speech recognition often takes raw PCM audio samples and produces a *spectrogram*, essentially a visual fingerprint of the frequencies over time, which can then be fed into standard image classification models. If you have time-based data (for example, accelerometer or vibration signals), then this can be an easy way to get started with a neural network solution.

Embeddings
: There are some domains, like text, where there's no obvious way to turn the natural input data into the numbers that a neural network requires. This is where embeddings come in handy. These take a non-numerical input, like a word, and assign it an

n-dimensional number. To be useful, these numbers are usually chosen so that objects that have properties in common have numbers that are nearby in that *n*-dimensional space. A classic example of this is *word2vec*, where words with similar meanings are usually close to each other. These are called *embeddings* because all of the concepts can be seen as being embedded in an *n*-dimensional space represented by the numbers assigned. The idea isn't restricted to words, however; you can use embeddings for anything you want to convert into useful numbers, even unlikely things like photos of landmarks. They can be useful as the output of networks, too, since the calculations on typical, final, fully connected layers scale linearly with the number of classes being detected; whereas if you output an embedding, it's possible to do a nearest-neighbor lookup on a very large set of candidates with much lower latency.

Language models
: A final pattern to know about is where the output of a neural network is just an input to a much larger system. A good example of this is a language model, in which a speech recognition neural network might output likely phonemes for some audio, but then some custom code that understands likely words and sentences will try to make sense of them overall.

Building TensorFlow for Your Platform

Because the requirements for building on different mobile and embedded platforms vary, there are a variety of different ways to compile the TensorFlow framework.

Android

TensorFlow's first release came with support for Android, and it's been a priority for the team to keep improving the experience. There are a variety of ways to run TensorFlow on Android, from a prepackaged binary installation to compiling it yourself from scratch.

Android Studio

The simplest way to use TensorFlow on Android is to create a new project in Android Studio and then add the following two lines to your *build.gradle* file:

```
allprojects {
 repositories {
 jcenter()
 }
}
dependencies {
 compile 'org.tensorflow:tensorflow-android:+'
}
```

There's also support for using Bazel under the hood from Studio, so you'll first need to make sure that you're able to build for Android using that approach:

1. Type `which bazel` on the command line to find out where your copy of Bazel is installed.
2. If it's not at */usr/local/bin/bazel*, edit the `bazel_location` definition in *tensorflow/examples/android/build.gradle* to point to the correct path.
3. In Android Studio, add the *tensorflow/examples/android* folder as a new project.
4. You should now be able to build the example application from within the IDE.

Bazel for Android

The most common method of building TensorFlow on the desktop is using the open source Bazel build tool. Bazel does require Java and a reasonable number of other dependencies to be installed first and uses a lot of memory during the build process, so it can be challenging to run directly on devices that have limited resources, such as the Raspberry Pi. It's also not easy to set up cross-compilation if you're compiling on a different machine than you're deploying to (for example, building on macOS to target iOS devices). However, it is the most mainstream and well-supported build method for TensorFlow, so if you can, we recommend using it.

Here's how you get started with Android development using Bazel:

1. Download a copy of the source code using `git clone https://github.com/tensorflow/tensorflow`.
2. Install a current version of Bazel, using the latest recommended version and installation instructions.
3. Download the Android SDK and NDK. It's good to do this using Android Studio's SDK management interface. You need at least version 12b of the NDK, and we use version 23 of the SDK.
4. In your copy of the TensorFlow source, update the WORKSPACE file with the location of your SDK and NDK. Keep a small snippet file handy with that information, then append it every time you set up a new source tree. Run a command like `cat ~/android_bazel_append.txt >> WORKSPACE` to do the appending. Note the double angle brackets, which are important, since otherwise you'll just overwrite your workspace! Here's what our snippet file looks like (you'll have to set the paths to your own locations):

```
android_sdk_repository(
 name = "androidsdk",
 api_level = 23,
 build_tools_version = "23.0.2",
 path = "/home/petewarden/android-sdk-linux/",
)
android_ndk_repository(
 name="androidndk",
 path="/home/petewarden/android-ndk-r10e/",
 api_level=19)
```

5. Run Bazel with the following command to build the demo:

```
bazel build -c opt //tensorflow/examples/android:tensorflow_demo
```

This should generate an APK that you can install on your Android device. This particular example is Android-only, so the flag isn't needed; but in general, when compiling for the OS, you need `--config=android` on the Bazel command line.

Android examples

The Android example code is organized as a single project that builds and installs three different apps, all using the same underly-

ing code. These apps are all image-related and take video input from the phone's camera:

TF Classify
: This app uses the Inception v3 model to label the objects it's pointed at with classes from Imagenet. There are only 1,000 categories in Imagenet, which misses most everyday objects and includes many things you're unlikely to encounter in real life, so the results can often be quite amusing. For example, there's no "person" category, so instead TF Classify will guess things it does know that are associated with pictures of people, such as a seat belt or an oxygen mask. If you do want to customize this example to recognize objects you care about, the good news is that the TensorFlow for Poets codelab lets you easily generate a model based on your own data.

TF Detect
: This app uses a multibox model to try to draw bounding boxes around the locations of people in the camera. These boxes are also annotated with the confidence for each detection result. This kind of object detection is still an active research topic, so your results may vary depending on the conditions. The demo also includes tracking that runs at a much higher frequency than the TensorFlow inference. This speed improves the user experience, since the apparent frame rate is faster, and it also gives the ability to estimate which boxes refer to the same object between frames, which is important for counting objects over time.

TF Stylize
: This app implements a real-time style-transfer algorithm on the camera feed. You can select the styles to use and mix between them using the palette at the bottom of the screen, and also switch out the resolution of the processing to go higher- or lower-resolution.

To build and install all of these apps, first make sure you have Bazel and the Android SDKs set up on your machine, and then run:

```
bazel build tensorflow/examples/android:tensorflow_demo
adb install -r \
 bazel-bin/tensorflow/examples/android/tensorflow_demo.apk
```

You should now see three app icons on your phone, one for each of the demos. Tapping on them opens up the app and lets you explore what they do. You can enable profiling statistics on-screen by tapping the volume-up button while they're running.

Android Inference Library

Because Android apps need to be written in Java, and core TensorFlow is in C++, we provide a JNI library to interface between the two. Its interface is aimed only at inference, so it provides the ability to load a graph, set up inputs, and run the model to calculate particular outputs. You can see the full documentation for the minimal set of methods here: *http://bit.ly/TensorFlowInferenceInterface-java*. The demos applications use this interface, so they're a good place to look for example usage. You can download prebuilt binary jars at *https://ci.tensorflow.org/view/Nightly/job/nightly-android/*.

iOS

We've seen a lot of great internal and external applications launch using TensorFlow on iOS, so supporting the platform is important to us. We've now got both a prebuilt approach as well as a recipe for building it from source if you need more customization.

CocoaPods

The simplest way to get started with TensorFlow on iOS is using the CocoaPods package management system. You can download the TensorFlow pod from cocoapods.org, and then simply run `pod 'TensorFlow-experimental'` to add it as a dependency to your application's Xcode project. This installs a universal binary framework, which makes it easy to get started but has the disadvantage of being hard to customize, which is important in case you want to shrink your binary size.

Makefile

The Unix make utility is one of the oldest build tools available, but its low-level approach offers a lot of useful flexibility for tricky situations like cross-compiling or building on old or limited-resource systems. TensorFlow offers a makefile aimed toward mobile and embedded platforms at *tensorflow/contrib/makefile*. It's the main way of building for iOS, but there are instructions for targeting Linux, Android, and the Raspberry Pi.

Building all at once

If you just want to get TensorFlow compiled for iOS in one go, you can run this from the root of your TensorFlow source folder:

```
tensorflow/contrib/makefile/build_all_ios.sh
```

This process takes around 20 minutes on my 2013 MacBook Pro.

When it completes, you will have a library for all architectures, but the script does a clean at the start, so don't run it repeatedly if you're making changes to the TensorFlow source code.

Building by hand

If you haven't done it already, download dependencies:

```
tensorflow/contrib/makefile/download_dependencies.sh
```

Next, compile protobufs for iOS:

```
tensorflow/contrib/makefile/compile_ios_protobuf.sh
```

Then, run the makefile specifying iOS as the target, along with the architecture you want to build for:

```
make -f tensorflow/contrib/makefile/Makefile \
 TARGET=IOS \
 IOS_ARCH=ARM64
```

This creates a universal library in *tensorflow/contrib/makefile/gen/lib/libtensorflow-core.a* that you can link any Xcode project against.

Optimization

The *compile_ios_tensorflow.sh* script can take optional command-line arguments. The first argument is passed as a C++ optimization flag and defaults to debug mode. If you are concerned about perfor-

mance or are working on a release build, you likely want a higher optimization setting, like so:

```
compile_ios_tensorflow.sh "-Os"
```

For other variations of valid optimization flags, see clang optimization levels.

iOS examples

There are three demo applications for iOS, all defined in Xcode projects inside *tensorflow/contrib/ios_examples*:

Simple
: A minimal example showing how to load and run a TensorFlow model in as few lines as possible. It consists of a single view with a button that executes the model loading and inference when it's pressed.

Camera
: Similar to the Android TF Classify demo. It loads Inception v3 and outputs its best label estimate for what's in the live camera view. As with the Android version, you can train your own custom model using TensorFlow for Poets and drop it into this example with minimal code changes.

Benchmark
: Quite close to Simple, but it runs the graph repeatedly and outputs similar statistics to the benchmark tool on Android.

To build these demos, first ensure you've been able to compile the main TensorFlow library successfully for iOS. You'll also need to download the model files they need:

```
mkdir -p ~/graphs
curl -o ~/graphs/inception5h.zip \
 https://storage.googleapis.com/download.tensorflow.org/ \
 models/inception5h.zip
unzip ~/graphs/inception5h.zip -d ~/graphs/inception5h
cp ~/graphs/inception5h/* \
 tensorflow/examples/ios/benchmark/data/
cp ~/graphs/inception5h/* \
 tensorflow/examples/ios/camera/data/
cp ~/graphs/inception5h/* \
 tensorflow/examples/ios/simple/data/
```

You should be able to load the Xcode project for each individual demo, build it, and run on-device. The Camera demo requires a

camera, as the name suggests, so it won't run on an emulator, but the other two should. You can use C++ directly from iOS applications; the code calls directly into the TensorFlow framework. There's no need for an inference API library as on Android.

Raspberry Pi

The TensorFlow team is working on providing an official `pip install` path for getting the framework running easily on the Pi with pre-built binaries. At the time of writing it's not yet available (check *https://www.tensorflow.org/install* for the latest details), so here I'll cover how to build it from source. Building on the Raspberry Pi is similar to a normal Linux system. First, download the dependencies, install the required packages, and build protobuf:

```
tensorflow/contrib/makefile/download_dependencies.sh
sudo apt-get install -y \
 autoconf automake libtool gcc-4.8 g++-4.8
cd tensorflow/contrib/makefile/downloads/protobuf/
./autogen.sh
./configure
make
sudo make install
sudo ldconfig # refresh shared library cache
cd ../../../../..
```

Once that's done, you can use make to build the library and example:

```
make -f tensorflow/contrib/makefile/Makefile HOST_OS=PI \
 TARGET=PI OPTFLAGS="-Os" CXX=g++-4.8
```

If you're only interested in building for Raspberry Pi's second and third generations, you can supply some extra optimization flags to give you code that will run faster:

```
make -f tensorflow/contrib/makefile/Makefile HOST_OS=PI \
 TARGET=PI OPTFLAGS="-Os -mfpu=neon-vfpv4 \
 -funsafe-math-optimizations -ftree-vectorize" CXX=g++-4.8
```

One thing to be careful of is that the GCC version 4.9 currently installed on Jessie by default will hit an error mentioning `__atomic_compare_exchange`. This is why the examples in this section specify `CXX=g++-4.8` explicitly, and why we install it using `apt-get`. If you have partially built using the default GCC 4.9, hit the error and switch to 4.8. You need to do a `make -f tensorflow/contrib/makefile/Makefile clean` before you build. If you

don't, the build will appear to succeed, but you'll encounter `mal loc(): memory corruption` errors when you try to run any programs using the library.

Raspberry Pi examples

Raspberry Pi is a great platform for prototyping all sorts of embedded applications. There are two different examples included at *tensorflow/contrib/pi_examples*:

Label Image
: This example is a port of the standard *tensorflow/examples/label_image* demo, and it tries to label an image based on the Inception v3 Imagenet model. As with the other platforms, you can easily replace this model with a custom-trained one derived from TensorFlow for Poets.

Camera
: This example uses the Pi's camera API to pull a live video feed, runs image labeling on it, and outputs the top label to the console. For fun, it's designed so you can feed the results into the flite text to speech tool so that your Pi speaks what it sees.

To build these examples, make sure you've run the Pi build process as shown earlier, and then run `makefile -f tensorflow/contrib/pi_examples/camera` or `makefile -f tensorflow/contrib/pi_examples/simple`. This should give you an executable in *gen/bin* off your root source folder that you can run. To get the model files, you'll need:

```
curl https://storage.googleapis.com/download.tensorflow.org/\
models/inception_dec_2015_stripped.zip \
-o /tmp/inception_dec_2015_stripped.zip
unzip /tmp/inception_dec_2015_stripped.zip \
 -d tensorflow/contrib/pi_examples/label_image/data/
```

Integrating the TensorFlow Library into Your Application

Once you have made some progress on a model that addresses the problem you're trying to solve, it's important to test it out inside your application immediately. There are often unexpected differences between your training data and what users actually encounter

in the real world, and getting a clear picture of the gap as soon as possible improves the product experience.

Linking the Library

After you've managed to build the examples, the next step is to call the code from your own application. This means you need to break out TensorFlow as a framework, include the right header files, and link against the built libraries and dependencies. Unfortunately, there's no clear separation between implementation and API headers in the C++ core of TensorFlow, so you'll need to pull in quite a few things to be able to call it.

Here is a checklist of what you'll need to do, based on the iOS build:

- Link against *tensorflow/contrib/makefile/gen/lib/libtensorflow-core.a*, usually by adding *-L/your/path/tensorflow/contrib/makefile/gen/lib/* and *-ltensorflow-core* to your linker flags.
- Link against the generated protobuf libraries by adding *-L/your/path/tensorflow/contrib/makefile/gen/protobuf_ios/lib* and *-lprotobuf* and *-lprotobuf-lite* to your command line.
- For the include paths, you need the root of your TensorFlow source folder as the first entry, followed by *tensorflow/contrib/makefile/downloads/protobuf/src*, *tensorflow/contrib/makefile/downloads*, *tensorflow/contrib/makefile/downloads/eigen*, and *tensorflow/contrib/makefile/gen/proto*.
- Make sure your binary is built with -force_load (or the equivalent on your platform), aimed at the TensorFlow library to ensure that it's linked correctly. More detail on why this is necessary can be found in the next section, "Global Constructor Magic" on page 21. On Linux-like platforms, you'll need different flags, more like `-Wl`, `--allow-multiple-definition -Wl`, `--whole-archive`.
- On iOS, you'll also need to link in the Accelerator framework, since this is used to speed up some of the operations.

You may find Android a bit easier to work with, since you'll only need to pull in a Java library contained in a jar, and you can download nightly precompiled versions.

Global Constructor Magic

One of the subtlest problems you may run up against is the "No session factory registered for the given session options" error when trying to call TensorFlow from your own application. To understand why this is happening and how to fix it, you need to know a bit about the architecture of TensorFlow.

The framework is designed to be very modular, with a thin core and a large number of specific objects that are independent and can be mixed and matched as needed. To enable this, we needed a coding pattern in C++ that easily let modules notify the framework about the services they offer, without requiring a central list that would have to be updated separately from each implementation. We also needed a way for separate libraries to add their own implementations without needing a recompile of the core.

To achieve this capability, we ended up using a registration pattern in a lot of places. In the code, it looks something like this:

```
class MulKernel : OpKernel {
  Status Compute(OpKernelContext* context) { … }
};
REGISTER_KERNEL(MulKernel, "Mul");
```

This would be in a standalone *.cc* file that linked into your application, either as part of the main set of kernels or as a separate custom library. The magic part is that the `REGISTER_KERNEL()` macro is able to inform the core of TensorFlow that it has an implementation of the `Mul` operation so that it can be called in any graphs that require it.

From a programming point of view, this setup is very convenient. The implementation and registration code live in the same file, and adding new implementations is as simple as compiling and linking it in. The difficult part comes from the way that the `REGISTER_KERNEL()` macro is implemented. C++ doesn't offer a good mechanism for doing this sort of registration, so we have to resort to some tricky code. Under the hood, the macro is implemented so that it produces something like this:

```
class RegisterMul {
 public:
  RegisterMul() {
    global_kernel_registry()->Register("Mul", [](){
      return new MulKernel()
```

```
      });
    }
};
RegisterMul g_register_mul;
```

What this is saying is that there's a class `RegisterMul` with a constructor that tells the global kernel registry what function to call when somebody asks it how to create a `Mul` kernel. Then there's a global object of that class, and so the constructor should be called at the start of any program.

If you've followed that, hopefully it sounds sensible, right? The unfortunate part is that the global that's defined is not used by any other code, so linkers not designed with this in mind will decide that it can be deleted. As a result, the constructor is never called, and the class is never registered. All sorts of modules use this pattern in TensorFlow, and it happens that `Session` implementations are the first to be looked for when the code is run, which is why it shows up as the characteristic error when this problem occurs.

The solution is to force the linker to not strip any code from the library, even if it believes it's unused. On iOS, this step can be accomplished with the `-force_load` flag, specifying a library path, and on Linux you need `--whole-archive`. These persuade the linker to not be as aggressive about stripping, and they should retain the globals.

The actual implementation of the various `REGISTER_*` macros is a bit more complicated in practice, but they all suffer the same underlying problem. If you're interested in how they work, *https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_kernel.h#L1091* is a good place to start investigating.

Protobuf Problems

TensorFlow relies on the Protocol Buffer library, commonly known as protobuf. This library takes definitions of data structures and produces serialization and access code for them in a variety of languages. The tricky part is that this generated code needs to be linked against shared libraries for the exact same version of the framework that was used for the generator. This can be an issue when `protoc`, the tool used to generate the code, is from a different version of protobuf than the libraries in the standard linking and include paths. For example, you might be using a copy of `protoc` that was built

locally in *~/projects/protobuf-3.0.1.a*, but you have libraries installed at */usr/local/lib* and */usr/local/include* that are from 3.0.0.

The symptoms of this issue are errors during the compilation or linking phases with protobufs. Usually, the build tools take care of this, but if you're using the makefile, make sure you're building the protobuf library locally and using it, as shown in *https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/makefile/Makefile#L18*.

Another situation that can cause problems is when protobuf headers and source files need to be generated as part of the build process. This process makes building more complex, since the first phase has to be a pass over the protobuf definitions to create all the needed code files, and only after that can you go ahead and do a build of the library code.

The other thing about protobufs that's tricky is that they generate headers that are needed as part of the C++ interface to the overall TensorFlow library. This complicates using the library as a standalone framework, as described in the next section.

Protobuf Compatibility Issues

If your application is already using version 1 of the protocol buffers library, you may have trouble integrating TensorFlow because it requires version 2. If you just try to link both versions into the same binary, you'll see linking errors because some of the symbols clash. To solve this particular problem, we have an experimental script at *http://bit.ly/rename-protobuf-sh*.

You need to run this as part of the makefile build, after you've downloaded all the dependencies:

```
tensorflow/contrib/makefile/download_dependencies.sh
tensorflow/contrib/makefile/rename_protobuf.sh
```

Calling the TensorFlow API

Once you have the framework available, you then need to call into it. The usual pattern is that you first load your model, which represents a preset set of numeric computations, and then you run inputs through that model (for example, images from a camera) and receive outputs (for example, predicted labels).

On Android, we provide the Java Inference Library that is focused on just this use case, while on iOS and Raspberry Pis you call directly into the C++ API.

Here's what a typical Inference Library sequence looks like on Android:

```
// Load the model from disk.
TensorFlowInferenceInterface inferenceInterface = \
        new TensorFlowInferenceInterface(assetManager, \
        modelFilename);


// Copy the input data into TensorFlow.
inferenceInterface.feed(inputName, \
        floatValues, 1, inputSize, inputSize, 3);


// Run the inference call.
inferenceInterface.run(outputNames, logStats);


// Copy the output Tensor back into the output array.
inferenceInterface.fetch(outputName, outputs);
```

You can find the source of this code in the Android examples at *http://bit.ly/TensorFlowImageClassifier-java*.

Here's the equivalent code for iOS:

```
// Load the model.
PortableReadFileToProto(file_path, &tensorflow_graph);


// Create a session from the model.
tensorflow::Status s = session->Create(tensorflow_graph);
if (!s.ok()) {
  LOG(FATAL) << "Could not create TensorFlow Graph: " << s;
}


// Run the model.
std::string input_layer = "input";
std::string output_layer = "output";
std::vector<tensorflow::Tensor> outputs;
tensorflow::Status run_status = session->Run(
        {{input_layer, image_tensor}},
                                {output_layer}, {}, &outputs);
if (!run_status.ok()) {
  LOG(FATAL) << "Running model failed: " << run_status;
}
```

```
// Access the output data.
tensorflow::Tensor* output = &outputs[0];
```

This is all based on the iOS sample code at the iOS sample code at *https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/ios/simple*, but there's nothing iOS-specific; the same code should be usable on any platform that supports C++.

C++ op API

On the desktop side, there's a great API that autogenerates classes for every op at *http://bit.ly/api-guides* based on the op definitions. This is very handy if you need to create graphs dynamically from C++ rather than loading them, and you can see this in practice in the *label_image* example at *http://bit.ly/label-image*. Unfortunately, this automatic code generation adds extra complexity to the build process, so it's not currently supported on mobile devices.

Training On-Device

Most applications of TensorFlow on mobile and embedded devices are focused on inference, taking a model that's been trained in the cloud and running it locally with unchanging parameters. There are some interesting use cases for on-device training emerging, though, such as federated learning. The implementation of federated learning that is shipped with Google Keyboard uses TensorFlow under the hood, so it's definitely possible to use it like for on-device training. It's not a heavily used path, however, so you'll find there are some challenges:

- The C++ API doesn't yet support automatically figuring out gradient ops for a forward model pass. Instead, you'll need to generate your training graph in Python to use automatic differentiation and export it, or manually add the right ops by manually building a GraphDef from NodeDefs in C++.
- You'll need to make sure you include the right training ops, since these are typically not included in mobile builds because we expect most applications of TensorFlow on mobile will only use inference. See "What Ops Are Available on Mobile?" on page 33 for more information on adding op implementations.

What Are the Minimum Device Requirements for TensorFlow?

You need at least one megabyte of program memory and several megabytes of RAM to run the base TensorFlow runtime, so it's not suitable for DSPs or microcontrollers. Other than those, the biggest constraint is usually the calculation speed of the device, and if you can run the model you need for your application with a low enough latency. It's a good idea to use the benchmarking tools described earlier to get an idea of how many FLOPs are required for a model, and then use that to make rule-of-thumb estimates of how fast they will run on different devices. For example, a modern smartphone might be able to run 10 GFLOPs per second, so the best you could hope for from a 5 GFLOP model is 2 frames per second, although you may do worse, depending on what the exact computation patterns are.

This model dependence means that it's possible to run TensorFlow even on very old or constrained phones, as long as you optimize your network to fit within the latency budget, and possibly within limited RAM, as well. For memory usage, you mostly need to make sure that the intermediate buffers that TensorFlow creates aren't too large, which you can examine in the benchmark output too.

Preparing Your Model File for Mobile Deployment

The requirements for storing model information during training are very different from when you want to release it as part of a mobile app. This section covers the tools involved in converting from a training model to something releasable in production.

What's Up with All the Different Saved File Formats?

You may find yourself confused by all the different ways TensorFlow can save out graphs. Here's a rundown of some of the different components and what they are used for. The objects are mostly defined and serialized as protocol buffers:

NodeDef
: Defines a single operation in a model. It has a unique name, a list of the names of other nodes it pulls inputs from, the operation type it implements (for example, `Add` or `Mul`), and any

attributes that are needed to control that operation. This is the basic unit of computation for TensorFlow, and all work is done by iterating through a network of these nodes, applying each one in turn. One particular operation type that's worth knowing about is `Const`, since this type holds information about a constant. This may be a single, scalar number or string, but it can also hold an entire multidimensional tensor array. The values for a `Const` are stored inside the NodeDef, and so large constants that contain a lot of values can take up a lot of room when serialized.

Checkpoint

Another way of storing values for a model is by using `Variable` ops. Unlike `Const` ops, these don't store their content as part of the `NodeDef`, so they take up very little space within the `GraphDef` file. Instead, their values are held in RAM while a computation is running, and then saved out to disk as checkpoint files periodically. This typically happens as a neural network is being trained and weights are updated, so it's a time-critical operation and may happen in a distributed fashion across many workers. As a result, the file format has to be both fast and flexible. They are stored as multiple checkpoint files, together with metadata files that describe what's contained within the checkpoints. When you're referring to a checkpoint in the API (for example, when passing a filename in as a command-line argument), you'll use the common prefix for a set of related files. If you had these files:

```
/tmp/model/model-chkpt-1000.data-00000-of-00002
/tmp/model/model-chkpt-1000.data-00001-of-00002
/tmp/model/model-chkpt-1000.index
/tmp/model/model-chkpt-1000.meta
```

you would refer to them as */tmp/model/chkpt-1000*.

GraphDef

GraphDef has a list of NodeDefs, which together define the computational graph to execute. During training, some of these nodes will be variables, so if you want to have a complete graph you can run, including the weights, you'll need to call a restore operation to pull those values from checkpoints. Because checkpoint loading has to be flexible to deal with all of the training requirements, this can be tricky to implement on mobile and

embedded devices, especially those with no proper filesystem available, like iOS. This is where the *freeze_graph.py script* comes in handy. As mentioned, `Const` ops store their values as part of the NodeDef, so if all the `Variable` weights are converted to `Const` nodes, then we only need a single GraphDef file to hold the model architecture and the weights. Freezing the graph handles the process of loading the checkpoints, and then converts all Consts to Variables. You can then load the resulting file in a single call, without having to restore variable values from checkpoints. One thing to watch out for with GraphDef files is that sometimes they're stored in text format for easy inspection. These versions usually have a *.pbtxt* filename suffix, whereas the binary files end with *.pb*.

FunctionDefLibrary
: FunctionDefLibrary appears in GraphDef and is effectively a set of subgraphs, each with information about their input and output nodes. Each subgraph can then be used as an op in the main graph, allowing easy instantiation of different nodes, in a similar way to how functions encapsulate code in other languages.

MetaGraphDef
: A plain `GraphDef` only has information about the network of computations but doesn't have any extra information about the model or how it can be used. `MetaGraphDef` contains a `GraphDef` defining the computation part of the model, but also includes information like *signatures*, which are suggestions about which inputs and outputs you may want to call the model with, data on how and where any checkpoint files are saved, and convenience tags for grouping ops together for ease of use.

SavedModel
: It's common to want to have different versions of a graph that rely on a common set of variable checkpoints. For example, you might need a GPU and a CPU version of the same graph, but keep the same weights for both. You might also need some extra files (like label names) as part of your model. The SavedModel format addresses these needs by letting you save multiple versions of the same graph without duplicating variables and by storing asset files in the same bundle. Under the hood, it uses `MetaGraphDef` and checkpoint files, along with extra metadata

files. It's the format that you'll want to use if you're deploying a web API using TensorFlow Serving, for example.

How Do You Get a Model You Can Use on Mobile?

In most situations, training a model with TensorFlow gives you a folder containing a GraphDef file (usually ending with the *.pb* or *.pbtxt* extension) and a set of checkpoint files. What you need for mobile or embedded deployment is a single GraphDef file that's been "frozen" or had its variables converted into inline constants so everything's in one file. To handle the conversion, you'll need the *freeze_graph.py* script that's held in *tensorflow/pythons/tools/freeze_graph.py*.

Run it like this:

```
bazel build tensorflow/python/tools:freeze_graph
bazel-bin/tensorflow/python/tools/freeze_graph \
 --input_graph=/tmp/model/my_graph.pb \
 --input_checkpoint=/tmp/model/model.ckpt-1000 \
 --output_graph=/tmp/frozen_graph.pb \
 --input_node_names=input_node \
 --output_node_names=output_node
```

The `input_graph` argument should point to the GraphDef file that holds your model architecture. It's possible that your GraphDef has been stored in a text format on disk, in which case it's likely to end in *.pbtxt* instead of *.pb*, and you should add an extra `--input_binary=false` flag to the command.

The `input_checkpoint` should be the most recent saved checkpoint. As previously mentioned, you need to give the common prefix to the set of checkpoints here, rather than a full filename. `output_graph` defines where the resulting frozen GraphDef will be saved. Because it's likely to contain a lot of weight values that take up a large amount of space in text format, it's always saved as a binary protobuf. `output_node_names` is a list of the names of the nodes that you want to extract the results of your graph from. This list is needed because the freezing process must understand which parts of the graph are actually needed, and which are artifacts of the training process, such as summarization ops. Only ops that contribute to calculating the given output nodes will be kept. If you know how your graph is going to be used, these should just be the names of the nodes you pass into `Session::Run()` as your fetch targets. If you

don't have this information handy, you can get some suggestions on likely outputs by running the `summarize_graph` tool.

Because the output format for TensorFlow has changed over time, there are a variety of other less commonly used flags available too, such as `input_saver`, but hopefully you shouldn't need these on graphs trained with modern versions of the framework.

Using the Graph Transform Tool

A lot of the things you need to do to efficiently run a model on-device are available through the Graph Transform Tool interface. This command-line tool takes an input GraphDef file, applies the set of rewriting rules you request, and writes out the result as a GraphDef.

Removing training-only nodes

TensorFlow GraphDefs, produced by the training code, contain all the computation that's needed for back-propagation and updates of weights, as well as the queuing and decoding of inputs and the saving out of checkpoints. None of these nodes are needed during inference, and some of the operations, such as checkpoint saving, aren't even supported on mobile platforms. To create a model file that you can load on devices, delete those unneeded operations by running the `strip_unused_nodes` rule in the Graph Transform Tool.

The trickiest part of this process is figuring out the names of the nodes you want to use as inputs and outputs during inference. You'll need these anyway once you start to run inference, but you also need them here so the transform can calculate which nodes are not needed on the inference-only path. These nodes may not be obvious from the training code because inputs are often fed through queues and decoding operations, and outputs feed into loss calculations, so they're not the final op in the graph. Unfortunately, there's no automatic way to discover these, and I often end up visualizing the graph and doing some detective work to figure out the right values.

Remember that mobile applications typically gather their data from sensors and have it as arrays in memory, whereas training involves loading and decoding representations of the data stored on disk. In the case of Inception v3, for example, there's a DecodeJpeg op at the start of the graph that's designed to take JPEG-encoded data from a file retrieved from disk and turn it into an arbitrary-sized image.

After that there's a BilinearResize op to scale it to the expected size, followed by a couple of other ops that convert the byte data into floats and and scales their values to the range that the rest of the graph expects. A typical mobile app skips most of these steps because it's getting its input directly from a live camera, so the input node you actually supply will be the output of the `Mul` node in this case. Figure 1-1 shows a diagram of an Inception input module.

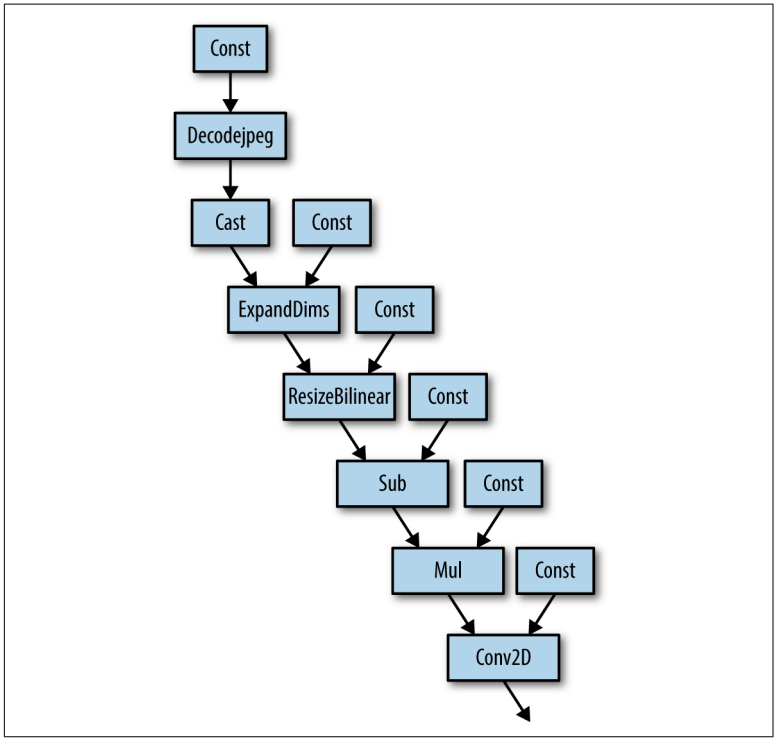

Figure 1-1. Diagram of an Inception input module

You'll need to do a similar process of inspection to figure out the correct output nodes.

If you've just been given a frozen GraphDef file, and you're not sure about the contents, we recommend using the summarize_graph tool to print out information about the inputs and outputs it finds from the graph structure. Here's an example with the original Inception v3 file:

```
bazel run tensorflow/tools/graph_transforms:summarize_graph -- \
 --in_graph=tensorflow_inception_graph.pb
```

Once you have an idea of what the input and output nodes are, you can feed them into the graph transform tool as the `--input_names` and `--output_names` arguments, and call the `strip_unused_nodes` transform, like this:

```
bazel run tensorflow/tools/graph_transforms:transform_graph --
--in_graph=tensorflow_inception_graph.pb
--out_graph=optimized_inception_graph.pb --inputs='Mul'
--outputs='softmax' --transforms= \
 ' strip_unused_nodes(type=float,
shape="1,299,299,3") fold_constants(ignore_errors=true) \
 fold_batch_norms
fold_old_batch_norms'
```

One thing to look out for is that you need to specify the size and type that you want your inputs to be. This is because any values you're going to be passing in as inputs to inference need to be fed to special Placeholder op nodes, and the transform may need to create them if they don't already exist. In the case of Inception v3, for example, a Placeholder node replaces the old Mul node that used to output the resized and rescaled image array, since we're going to be doing that processing ourselves before we call TensorFlow. It keeps the original name, though, which is why we always feed in inputs to `Mul` when we run a session with our modified Inception graph.

After you've run this process, you'll have a graph that only contains the actual nodes you need to run your prediction process. This is the point where it becomes useful to run metrics on the graph, so it's worth running *summarize_graph* again to understand what's in your model.

What Ops Are Available on Mobile?

There are hundreds of operations available in TensorFlow, and each one has multiple implementations for different data types. On mobile platforms, the size of the executable binary that's produced after compilation is important, because app download bundles need to be as small as possible for the best user experience. If all the ops and data types are compiled into the TensorFlow library, then the total size of the compiled library can be tens of megabytes—so by default, only a subset of ops and data types are included.

That means if you load a model file that's been trained on a desktop machine, you may see the error "No OpKernel was registered to support Op" when you load it on mobile. The first thing to try is to make sure you've stripped out any training-only nodes, since the error will occur at load time even if the op is never executed. If you're still hitting the same problem once that's done, you'll need to look at adding the op to your built library.

The criteria for including ops and types fall into several categories:

- Are they only useful in back-propagation, for gradients? Since mobile is focused on inference, we don't include these.
- Are they useful mainly for other training needs, such as checkpoint saving? These we leave out.
- Do they rely on frameworks that aren't always available on mobile, such as libjpeg? To avoid extra dependencies, we don't include ops like DecodeJpeg.
- Are there types that aren't commonly used? We don't include boolean variants of ops for example, since we don't see much use of them in typical inference graphs.

This trimming does go against the vision of TensorFlow being a single unified framework all the way from training to devices though, so it is possible to alter this default behavior. To do this, you'll need to alter some build files, as explained in the next section.

Locate the implementation

Operations are broken into two parts. The first is the *op definition*, which declares the signature of the operation, including its inputs, outputs, and attributes. These take up very little space, so all are

included by default. The implementations of the op computations are done in kernels, which live in the *tensorflow/core/kernels* folder. Compile the C++ file containing the kernel implementation of the op you need into the library. To figure out what file that is, search for the operation name in the source files. Here's an example search in GitHub: *https://github.com/search?utf8=%E2%9C%93&q=repo%3Atensorflow%2Ftensorflow+extension%3Acc+path%3Atensorflow%2Fcore%2Fkernels+REGISTER+Mul&type=Code&ref=searchresults*.

You'll see that this is actually looking for the Mul op implementation, and it finds it in `tensorflow/core/kernels/cwise_op_mul_1.cc`. You need to look for macros beginning with REGISTER, with the op name you care about as one of the string arguments.

In this case, the implementations are actually broken up across multiple `.cc` files, so you need to include all of them in your build. If you're more comfortable with the command line for code search, here's a grep command that also locates the right files if you run it from the root of your TensorFlow repository:

```
grep 'REGISTER.*"Mul"' tensorflow/core/kernels/*.cc
```

Add the implementation to the build

If you're using Bazel, you'll want to add the files you've found to the android_extended_ops_group1 or android_extended_ops_group2 targets. You may also need to include any *.cc* files they depend on in there. If the build complains about missing header files, add the *.h*'s that are needed into the android_extended_ops target.

If you're using a makefile, go to *tensorflow/contrib/makefile/tf_op_files.txt* and add the right implementation files there.

Optimizing for Latency, RAM Usage, Model File Size, and Binary Size

There are some special issues that you have to deal with when you're trying to ship on mobile or embedded devices, and you'll need to think about these as you're developing your model.

Model Size

The model needs to be stored somewhere on the device, and the largest networks can take hundreds of megabytes. Even if there's enough free storage, it can be off-putting to users to have to download extremely large app bundles from stores, so you'll need to plan for how large your model will be. A good place to start is by looking at the size on disk of your GraphDef file after you've run `freeze_graph` and `strip_unused_nodes` on it, since then it should only contain inference-related nodes. To double-check that your results are as expected, run the summarize_graph tool to see how many parameters are in constants:

```
bazel build tensorflow/tools/graph_transforms:summarize_graph \
 && bazel-bin/tensorflow/tools/graph_transforms/summarize_graph \
 --in_graph=/tmp/tensorflow_inception_graph.pb
```

That command should give you output that looks something like this:

```
No inputs spotted.
Found 1 possible outputs: (name=softmax, op=Softmax)
Found 23885411 (23.89M) const parameters, 0 (0) variable
  parameters, and 99 control_edges
Op types used: 489 Const, 99 CheckNumerics, 99 Identity,
  94 BatchNormWithGlobalNormalization, 94 Conv2D, 94 Relu,
  11 Concat, 9 AvgPool, 5 MaxPool, 1 Sub, 1 Softmax,
  1 ResizeBilinear, 1 Reshape, 1 Mul, 1 MatMul, 1 ExpandDims,
  1 DecodeJpeg, 1 Cast, 1 BiasAdd
```

The important part for our current purposes is the number of const parameters. In most models, these will be stored as 32-bit floats to start. If you multiply the number of const parameters by four, you should get something that's close to the size of the file on disk. You can often get away with only eight bits per parameter with very little loss of accuracy in the final result; so if your file size is too large, try running the `quantize_weights` to transform the parameters down:

```
bazel build tensorflow/tools/graph_transforms:transform_graph \
 && blaze-bin/tensorflow/tools/graph_transforms/transform_graph \
 --in_graph=/tmp/tensorflow_inception_optimized.pb \
 --out_graph=/tmp/tensorflow_inception_quantized.pb \
 --inputs='Mul:0' \
 --outputs='softmax:0' \
 --transforms='quantize_weights'
```

If you look at the resulting file size, you should see that it's about a quarter of the original at 23 MB. There's a variation on this that can sometimes be particularly useful for mobile development, which takes advantage of the fact that app bundles are compressed before they're downloaded by consumers. Normal floating-point numbers don't compress well with standard algorithms, because the bit patterns of even very similar numbers can be very different. As a result, models typically don't compress at all. The `round_weights` transform keeps the weight parameters stored as floats, but rounds them to a set number of step values. This means there are a lot more repeated byte patterns in the stored model, and so compression can often bring the size down dramatically, in many cases to near the size it would be if they were stored as eight bit.

The advantage is that the framework doesn't have to allocate a temporary buffer to unpack the parameters into, as we have to when we just use `quantize_weights`. This saves a little bit of latency (though the results should be cached so it's only costly on the first run) and makes it possible to use memory mapping, as described in "Need to Reduce Model Loading Time or Memory Footprint?" on page 43.

Speed

One of the highest priorities of most model deployments is figuring out how to run the inference fast enough to give a good user experience. The first place to start this process is by looking at the total number of floating-point operations required to execute the graph. You can get a very rough estimate of this by using the benchmark_model tool:

```
bazel build -c opt tensorflow/tools/benchmark:benchmark_model \
 && bazel-bin/tensorflow/tools/benchmark/benchmark_model \
 --graph=/tmp/inception_graph.pb --input_layer="Mul:0" \
 --input_layer_shape="1,299,299,3" --input_layer_type="float" \
 --output_layer="softmax:0" \
 --show_run_order=false --show_time=false \
 --show_memory=false --show_summary=true --show_flops=true \
 --logtostderr
```

This should show you an estimate of how many operations are needed to run the graph. You can then use that information to figure out how feasible your model is to run on the devices you're targeting. For an example, a high-end phone from 2016 might be able to do 20 billion FLOPs per second, so the best speed you could hope for from a model that requires 10 billion FLOPs is around 500 ms. On a device like the Raspberry Pi 3 that can do about five billion FLOPs, you may only get one inference every two seconds.

Having this estimate helps you plan for what you'll be able to realistically achieve on a device. If the model is using too many ops, then there are a lot of opportunities to optimize the architecture to reduce that number. Advanced techniques include SqueezeNet and MobileNet. You can also just look at alternative models, even older ones, which may be smaller. For example, Inception v1 only has around 7 million parameters, compared to Inception v3's 24 million, and requires only 3 billion FLOPs rather than 9 billion for v3.

How to Profile Your Model

Once you have an idea of the peak performance your device may be able to achieve, it's worth looking at the actual performance you're currently getting. To start with, I like to use a standalone benchmark, rather than running it inside a larger app, since that helps isolate just the TensorFlow contribution to the latency. The tensorflow/tools/benchmark tool is designed to help you do this. To run it on Inception v3 on your desktop machine, do something like this:

```
bazel build -c opt tensorflow/tools/benchmark:benchmark_model \
 && bazel-bin/tensorflow/tools/benchmark/benchmark_model \
 --graph=/tmp/tensorflow_inception_graph.pb \
 --input_layer="Mul" --input_layer_shape="1,299,299,3" \
 --input_layer_type="float" --output_layer="softmax:0" \
 --show_run_order=false --show_time=false \
 --show_memory=false --show_summary=true \
 --show_flops=true --logtostderr
```

You should see output that looks something like this:

```
============================== Top by Computation Time ==============================
[node type]  [start]  [first] [avg ms]     [%]   [cdf%]  [mem KB]  [Name]
     Conv2D   22.859   14.212   13.700  4.972%   4.972%  3871.488  conv_4/Conv2D
     Conv2D    8.116    8.964   11.315  4.106%   9.078%  5531.904  conv_2/Conv2D
     Conv2D   62.066   16.504    7.274  2.640%  11.717%   443.904  mixed_3/conv/Conv2D
     Conv2D    2.530    6.226    4.939  1.792%  13.510%  2765.952  conv_1/Conv2D
     Conv2D   55.585    4.605    4.665  1.693%  15.203%   313.600  mixed_2/tower/conv_1/Conv2D
     Conv2D  127.114    5.469    4.630  1.680%  16.883%    81.920  mixed_10/conv/Conv2D
     Conv2D   47.391    6.994    4.588  1.665%  18.548%   313.600  mixed_1/tower/conv_1/Conv2D
     Conv2D   39.463    7.878    4.336  1.574%  20.122%   313.600  mixed/tower/conv_1/Conv2D
     Conv2D  127.113    4.192    3.894  1.413%  21.535%   114.688  mixed_10/tower_1/conv/Conv2D
     Conv2D   70.188    5.205    3.626  1.316%  22.850%   221.952  mixed_4/conv/Conv2D


============================== Summary by node type ==============================
[Node type]       [count]      [avg ms]       [avg %]       [cdf %]      [mem KB]
     Conv2D            94       244.899       88.952%       88.952%     35869.953
    BiasAdd            95         9.664        3.510%       92.462%     35873.984
    AvgPool             9         7.990        2.902%       95.364%      7493.504
       Relu            94         5.727        2.080%       97.444%     35869.953
    MaxPool             5         3.485        1.266%       98.710%      3358.848
      Const           192         1.727        0.627%       99.337%         0.000
     Concat            11         1.081        0.393%       99.730%      9892.096
     MatMul             1         0.665        0.242%       99.971%         4.032
    Softmax             1         0.040        0.015%       99.986%         4.032
         <>             1         0.032        0.012%       99.997%         0.000
    Reshape             1         0.007        0.003%      100.000%         0.000


Timings (microseconds): count=50 first=330849 curr=274803 min=232354
max=415352 avg=275563 std=44193
Memory (bytes): count=50 curr=128366400(all same)
514 nodes defined 504 nodes observed
```

This is the summary view, which is enabled by the show_summary flag. The first table is a list of the nodes that took the most time, in order by how long they took. From left to right, the columns are:

- Node type, that is, what kind of operation this was.
- Start time of the op, showing where it falls in the sequence of operations.
- First time in milliseconds. This is how long the operation took on the first run of the benchmark, since by default 20 runs are executed to get more reliable statistics. The first time is useful to spot the ops that are doing expensive calculations on the first run and then caching the results.
- Average time for the operation across all runs, in milliseconds.
- What percentage of the total time for one run the op took. This is useful to understand where the hotspots are.
- The cumulative total time of this and the previous ops in the table. This is handy for understanding what the distribution of work is across the layers, to see if just a few of the nodes are taking up most of the time.

- Name of the node.

The second table is similar, but instead of breaking down the timings by particular, named nodes, it groups them by the kind of op. This is very useful to understand which op implementations you might want to optimize or eliminate from your graph. The table is arranged with the most costly operations at the start, and only shows the top 10 entries, with a placeholder for other nodes. The columns from left to right are:

- Type of the nodes being analyzed.
- Accumulated average time taken by all nodes of this type, in milliseconds.
- What percentage of the total time was taken by this type of operation.
- Cumulative time taken by this and op types higher in the table, so you can understand the distribution of the workload.
- How much memory the outputs of this op type took up.

Both tables have tabs as separators between columns, making it easy to copy and paste their results into spreadsheets. The summary by node type can be the most useful when looking for optimization opportunities, since it will tell you which operations are taking the most time. In this case, you can see that the Conv2D ops are almost 90% of the execution time. This is a sign that the graph is pretty optimal, since convolutions and matrix multiplies are expected to be the bulk of a neural network's computing workload.

As a rule of thumb, it's more worrying if you see a lot of other operations taking up more than a small fraction of the time. For neural networks, the ops that don't involve large matrix multiplications should usually be dwarfed by the ones that do. So if you see a lot of time going into those, it's a sign that either your network is non-optimally constructed, or the code implementing those ops is not as optimized as it could be. Performance bugs or patches are always welcome if you do encounter this situation, especially if they include an attached model exhibiting this behavior and the command line used to run the benchmark tool on it.

The run above was on your desktop, but the tool also works on Android, which is where it's most useful for mobile development. Here's an example command line to run it on a 64-bit ARM device:

```
bazel build -c opt --config=android_arm64 \
 tensorflow/tools/benchmark:benchmark_model
adb push bazel-bin/tensorflow/tools/benchmark/benchmark_model \
 /data/local/tmp
adb push /tmp/tensorflow_inception_graph.pb /data/local/tmp/
adb shell '/data/local/tmp/benchmark_model \
 --graph=/data/local/tmp/tensorflow_inception_graph.pb \
 --input_layer="Mul" --input_layer_shape="1,299,299,3" \
 --input_layer_type="float" --output_layer="softmax:0" \
 --show_run_order=false --show_time=false
--show_memory=false --show_summary=true'
```

You can interpret the results in exactly the same way as the desktop version. If you have any trouble figuring out what the right input and output names and types are, take a look at the section above about detecting these for your model, and look at the tensorflow/tools/graph_transforms/summarize_graph tool, which may give you helpful information.

There isn't good support for command-line tools on iOS, so instead there's a separate example at tensorflow/contrib/ios_examples/benchmark that packages the same functionality inside a standalone app. This outputs the statistics to both the screen of the device and the debug log. If you want on-screen statistics for the Android example apps, you can turn them on by pressing the volume-up button.

Profiling Within Your Own App

The output you see from the benchmark tool is actually generated from modules that are included as part of the standard TensorFlow runtime, which means you have access to them within your own applications too. You can see an example of how to do that here: *http://bit.ly/BenchmarkViewController*.

The basic steps are:

Create a `StatSummarizer` object:

```
tensorflow::StatSummarizer stat_summarizer(tensorflow_graph);
```

Set up the options:

```
tensorflow::RunOptions run_options;
run_options.set_trace_level(tensorflow::RunOptions::FULL_TRACE);
tensorflow::RunMetadata run_metadata;
```

Run the graph:

```
run_status = session->Run(run_options, inputs,
                          output_layer_names, {},
                          output_layers, &run_metadata);
```

Calculate the results and print them out:

```
assert(run_metadata.has_step_stats());
const tensorflow::StepStats&
step_stats = run_metadata.step_stats();
stat_summarizer->ProcessStepStats(step_stats);
stat_summarizer->PrintStepStats();
```

Visualizing Models

The most effective way to speed up your code is by altering your model so it does less work. To do that, you need to understand what your model is doing, and visualizing it is a good first step. To get a high-level overview of your graph, I recommend trying the TensorBoard web application. You should be able to load your GraphDef file into it, though it can sometimes struggle with larger frozen graphs. For a more granular view, convert your graphs into GraphViz's DOT format. Here's how you do that with a built-in script:

```
bazel build tensorflow/tools/quantization:graph_to_dot
bazel-bin/tensorflow/tools/quantization/graph_to_dot \
 --graph=/tmp/tensorflow_inception_graph.pb \
 --dot_output=/tmp/tensorflow_inception_graph.dot
```

If you're on a Unix-like environment where you can install the dot or xdot programs, you can then get an image of your graph by running `xdot /tmp/tensorflow_inception_graph.dot`. This make take a few seconds to lay out, or even hang for very large graphs. By default, the script only shows the operation types, since the names can be long and lead to an ugly visualization; but it's easy to alter the Python script to show more information if needed.

Threading

The desktop version of TensorFlow has a sophisticated threading model and will try to run multiple operations in parallel if it can. In

our terminology, this is called "inter-op parallelism" (though to avoid confusion with "intra-op," you could think of it as "between-op" instead). Inter-op parallelism can be set by specifying "interop_threads" in the session options. On mobile devices, the inter-op parallelism (how many ops run at once) is set to 1 by default so that operations are always run serially, one after the other. This behavior is reasonable because mobile processors usually have few cores and a small cache, so running multiple operations accessing disjoint parts of memory usually doesn't help performance. Intra-op parallelism (or "within-op") can be very helpful though, especially for computation-bound operations like convolutions where different threads can feed off the same small set of memory.

On mobile, how many threads an op will use is set to the number of cores by default, or just four in cases where the number of cores can't be determined. You can override the default number of threads by setting it explicitly using session options. It's a good idea to reduce the default if your app has its own threads doing heavy processing so that they don't interfere with each other.

Binary Size

One of the biggest differences between mobile and server development is the importance of binary size. On desktop machines it's not unusual to have executables that are hundreds of megabytes on disk, but for mobile and embedded apps it's vital to keep the binary as small as possible so that user downloads are easy. TensorFlow only includes a subset of op implementations by default, but this still results in a 12 MB final executable. To reduce this number, you can set up the library to only include the implementations of the ops that you actually need, based on automatically analyzing your model. To use it, follow these steps:

1. Run *tools/print_required_ops/print_selective_registration_header.py* on your model to produce a header file that only enables the ops it uses.
2. Place the *ops_to_register.h* file somewhere that the compiler can find it. This can be in the root of your TensorFlow source folder.
3. Build tensorflow with `SELECTIVE_REGISTRATION` defined, for example by passing in `--copts="-DSELECTIVE_REGISTRATION"` to your Bazel build command.

This process recompiles the library so that only the needed ops and types are included, which can dramatically reduce the executable size. For example, with Inception v3, the new size is only 1.5 MB.

Retrain with Mobile Data

The biggest cause of accuracy problems when running models on mobile apps is unrepresentative training data. For example, most of the Imagenet photos are well-framed so that the object is in the center of the picture, well-lit, and shot with a normal lens. Photos from mobile devices—especially selfies-- are often poorly framed, badly lit, and can have fisheye distortions.

The solution is to expand your training set with data actually captured from your application. This step can involve extra work, since you'll have to label the examples yourself, but even if you just use it to expand your original training data, it can help the training set dramatically. Improving the training set by doing this, and by fixing other quality issues like duplicates or badly labeled examples is the single best way to improve accuracy. It's usually a bigger help than altering your model architecture or using different techniques.

Need to Reduce Model Loading Time or Memory Footprint?

Most operating systems allow you to load a file using memory mapping, rather than going through the usual I/O APIs. Instead of allocating an area of memory on the heap and then copying bytes from disk into it, you simply tell the operating system to make the entire contents of a file appear directly in memory. One advantage of this method is that the OS knows the whole file will be read at once, and can efficiently plan the loading process so it's as fast as possible. The actual loading can also be put off until the memory is first accessed, so it happens asynchronously with your initialization code.

You can also tell the OS you'll only be reading from the area of memory, and not writing to it. This gives the benefit that when there's pressure on RAM, instead of writing out that memory to disk as normal virtualized memory needs to be when swapping happens, it can just be discarded, since there's already a copy on disk, saving a lot of disk writes.

Since TensorFlow models can often be several megabytes in size, speeding up the loading process can be a big help for mobile and embedded applications, and reducing the swap writing load can help a lot with system responsiveness too. It can also be very helpful to reduce RAM usage. For example, on iOS the system can kill apps that use more than 100 MB of RAM, especially on older devices. The RAM used by memory-mapped files doesn't count toward that limit though, so it's often a great choice for models on those devices.

TensorFlow has support for memory mapping the weights that form the bulk of most model files. Because of limitations in the protobuf serialization format, we have to make a few changes to our model loading and processing code. The way memory mapping works is that we have a single file in which the first part is a normal GraphDef serialized into the protocol buffer wire format, but then the weights are appended in a form that can be directly mapped.

To create this file, you need to run the `tensorflow/contrib/util:convert_graphdef_memmapped_format` tool. This tool takes in a GraphDef file that's been run through freeze_graph and converts it to the format that has the weights appended at the end. Since that file's no longer a standard GraphDef protobuf, you then need to make some changes to the loading code. You can see an example of this in the iOS Camera demo app, the `LoadMemoryMappedModel()` function.

The same code (with the Objective C calls for getting the filenames substituted) can be used on other platforms too. Because we're using memory mapping, we start by creating a special TensorFlow environment object that's set up with the file we'll be using:

```
c++ std::unique_ptr<tensorflow::MemmappedEnv> memmapped_env;
memmapped_env->reset(
      new tensorflow::MemmappedEnv(tensorflow::Env::Default()));
  tensorflow::Status mmap_status =
      (memmapped_env->get())->InitializeFromFile(file_path);
```

You then pass in this environment to subsequent calls, like this one for loading the graph:

```
tensorflow::GraphDef tensorflow_graph;
tensorflow::Status load_graph_status = ReadBinaryProto(
    memmapped_env->get(),
    tensorflow::MemmappedFileSystem::
    kMemmappedPackageDefaultGraphDef,
    &tensorflow_graph);
```

You also need to create the session with a pointer to the environment you've created:

```
tensorflow::SessionOptions options;
options.config.mutable_graph_options()
    ->mutable_optimizer_options()
    ->set_opt_level(::tensorflow::OptimizerOptions::L0);
options.env = memmapped_env->get();


tensorflow::Session* session_pointer = nullptr;
tensorflow::Status session_status =
    tensorflow::NewSession(options, &session_pointer);
```

One thing to notice here is that we're also disabling automatic optimizations, since in some cases these will fold constant subtrees, which will create copies of tensor values that we don't want and use up more RAM.

Once you've gone through these steps, you can use the session and graph as normal, and you should see a reduction in loading time and memory usage.

Protecting Model Files from Easy Copying

By default, your models will be stored in the standard serialized protobuf format on disk. In theory this means anybody can copy your model, so I'm often asked how to prevent this. In practice, most models are so application-specific and obfuscated by optimizations that the risk is similar to that of competitors disassembling and reusing your code. If you do want to make it tougher for casual users to access your files, it is possible to take some basic steps.

Most of our examples use the ReadBinaryProto convenience call to load a GraphDef from disk. This step does require an unencrypted protobuf on disk. Luckily, though, the implementation of the call is pretty straightforward, and it should be easy to write an equivalent that can decrypt in memory. Here's some code that shows how you can read and decrypt a protobuf using your own decryption routine:

```
Status ReadEncryptedProto(Env* env, const string& fname,
                          ::tensorflow::protobuf::MessageLite*
                          proto) \{
  string data;
  TF_RETURN_IF_ERROR(ReadFileToString(env, fname, &data));

  DecryptData(&data); // Your own function here.
```

```
  if (!proto->ParseFromString(&data)) {
    TF_RETURN_IF_ERROR(stream->status());
    return errors::DataLoss("Can't parse ", fname,
    " as binary proto"); } return Status::OK(); }
```

To use this, you'd need to define the `DecryptData()` function yourself. It could be as simple as something like the following code:

```
void DecryptData(string* data) {
        for (int i = 0; i < data.size(); ++i) {
          data[i] = data[i] ^ 0x23;
}
}
```

You may want something more complex, but exactly what you'll need is outside the current scope here.

Exploring Quantized Calculations

One of the most interesting research areas in neural networks involves reducing precision. By default, the most convenient format to use for calculations is 32-bit floating point, but because most networks are trained to be resilient to noise, it turns out that inference can be run with 8 bits or fewer without much loss in quality.

We've touched on this earlier, when we described how to use the `quantize_weights` transform to shrink down the file size of a model. Under the hood, the 8-bit buffers are expanded up to 32-bit floats before they're used for calculations, so it's a fairly minimal change to the network. All the other operations just see floating-point inputs as normal.

A much more radical approach is to try to perform as many calculations as possible using 8-bit representations, too. This approach offers a few advantages. Many CPUs have SIMD instructions (like NEON or AVX2) that can do more 8-bit calculations per cycle than they can float. It also means that specialized hardware, like Qualcomm's HVX DSP or Google's Tensor Processing Unit, which may not support floating point operations well, can accelerate neural network calculations. In theory, there's no reason we couldn't use fewer than eight bits too, and indeed in a lot of experiments we've seen seven or even five bits as usable without too much loss. However, at the moment there's not much hardware that can efficiently use these odd sizes.

Quantization Challenges

The biggest challenge with this sort of quantized approach is that neural networks require fairly arbitrary ranges of numbers that aren't known ahead of time, so fitting them into eight bits can be tough. It's also tricky to create arithmetic operations to use these representations, since we have to reimplement a lot of the utilities we get for free when using floating point, such as range checking.

Consequently, the resulting code looks quite different from the equivalent float versions. There are also problems introduced by the fact we don't know what the inputs will be ahead of time, so the ranges of intermediate calculations can be hard to estimate.

Quantized Representation

Because we're dealing with large arrays of numbers where the values are usually distributed within a common range, encoding those values linearly into eight bits using the minimum and maximum of the float values as the extremes works well. In practice, this looks a lot like a block floating-point representation, though it's actually a bit more flexible. To understand how this works, here's an example of encoding a floating point array, with the original values:

```
[-10.0, 20.0, 0]
```

By scanning the array, you can see that the minimum and maximum values are -10.0 and 20.0. Take each value in the array, subtract the float minimum from it, divide it by the difference between the minimum and the maximum to get a normalized value between 0.0 and 1.0, and then multiply by 255 to fit it into 8 bits. This gives us:

```
[((-10.0 - -10.0) / 30.0) * 255, ((20.0 - -10.0) / 30.0) * 255,
  ((0.0 - -10.0) / 30.0) * 255]
```

which resolves into:

```
[0, 255, 85]
```

The crucial thing when dealing with this representation is to remember that it's meaningless without also knowing the min and max that it's based on. It's best to think of it as a compression scheme for real numbers, where the range is needed to make sense of every value. Within TensorFlow, this means every time a quantized tensor is passed through the graph, you need to make sure that two auxiliary tensors holding the minimum and maximum float val-

ues for the main tensor are always wired in. All the operations that accept quantized buffers as inputs require them, and each quantized output always has two associated scalar outputs producing the range for the output.

Representation Drawbacks

The nice thing about this representation is that it's very general. You can use it to hold traditional, fixed-point values if you set the min and max to powers of two, the range doesn't have to be symmetrical as in typical signed representations, and it can be adapted to hold almost any scale of values.

These properties were important when we first started working with quantization, since we didn't have a clear understanding of what constraints we could place on the values without losing overall precision. As we've gained more experience, we've realized that we can be more restrictive without significantly affecting accuracy, and there are disadvantages to having such a flexible format.

One of the fundamental problems is that the range doesn't necessarily have to include zero. For example, you can validly specify the minimum as 10.0 and the maximum as 20.0. This makes implementing a lot of algorithms unnecessarily complicated; for example, there's no easy way to express adding zero onto a number with that representation. It also turns out that zero is an unusually common number in neural networks, since it's used in the padding for convolutions beyond the edges of an image and is the output for any negative numbers from the Relu activation function. This creates the subtle problem that if zero doesn't have an exact representation—for example, if its closest encoded value actually decodes to 0.1 rather than 0.0—the error introduces a bias that hurts the overall accuracy of the network.

Quantization works on neural networks when the errors introduced by rounding to eight bits look similar to the kind of noise that they're trained to cope with anyway. This means the quantization errors must be roughly uniform, or at least average out to zero over large enough runs. This uniformity holds true if every number in the encoded range comes up with the same frequency, but when zero is present much more often than any other number, than whatever quantization error is present for that value will be ampli-

fied.This results in a systematic bias that can skew the final results of the network.

Another drawback is that it's possible to create nonsensical or invalid representations, for example, where the min and max are equal or the minimum is greater than the maximum. This kind of representation is probably a sign that the format is a flawed way to represent what we're trying to hold. Yet another issue is that when we're trying to use the same format for 32-bit numbers, the float values for the ranges become very large and unwieldy.

A possible solution might be to switch to a representation where we just express the real value of an incremental increase of 1 in the code. This should remain small enough to avoid some of these problems. It also might be worth just restricting the offset choices to just signed or unsigned so that the ranges are just from zero to max, or symmetrical so that the minimum of the range is always the negative of the maximum. Even with symmetrical ranges, there's still the challenge that a twos-complement signed 8-bit value has a minimum of -128 but a maximum of 127. That means if you assign your float ranges naively to those coding values you'll end up with the real value of zero not falling exactly on the zero encoding.

To address some of these issues, we've ended up constraining what values the min and max can actually be. For example, we always try to produce ranges that include zero; and if min and max are too close together, we nudge them apart by a small amount. In the future, we may enforce symmetrical or positive ranges too. Luckily, we're able to handle this without changing the representation; instead, we just enforce these constraints whenever the code calculates ranges for quantized buffers.

Using Quantized Calculations in TensorFlow

The usual process is to take a model that's been trained in floating point and run it through the `quantize_nodes` conversion process using the graph transform tool. What this does is replace normal float operations with their quantized equivalents, where those exist. Because the implementations of 8-bit algorithms are so different from floating-point versions, we've focused on ops that are commonly used in popular models to start with. You can find the most up-to-date list here; but at the time of writing, here are the operations that will run natively in 8-bit:

- `BiasAdd`
- `Concat`
- `Conv2D`
- `MatMul`
- `Relu`
- `Relu6`
- `AvgPool`
- `MaxPool`
- `Mul`

These operations are enough to implement the Inception networks and many other convolutional models. For any sets of adjacent nodes that belong to these types, 8-bit quantized buffers will be passed between them. If an unsupported operation is encountered, any quantized tensors will be converted to floats and fed in as normal, with a conversion back to quantized happening just before the next 8-bit operation.

Let's look at an example of the Relu operation. All Relu does is take a tensor array of values and output a copy of its input. Any negative numbers are replaced by zeros. An initial graph might look like Figure 1-2.

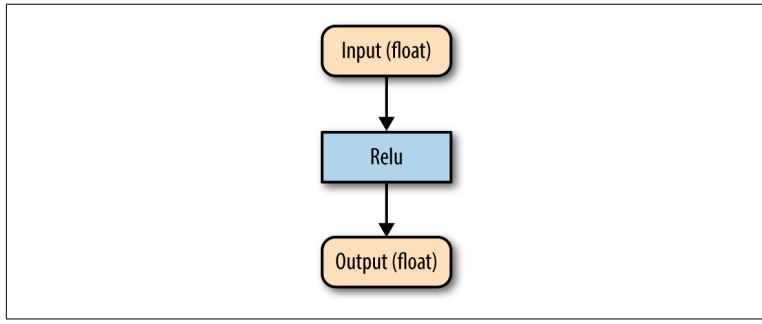

Figure 1-2. Simple float Relu graph

The yellow boxes are tensors, and the square blue box is the operation. The first thing the `quantize_nodes` rewriting rule does is replace the Relu with the equivalent 8-bit version (called Quantized Relu). At this point, you're not looking at what ops are surrounding

it in the graph though, so make sure the inputs and outputs are converted to float, as in Figure 1-3.

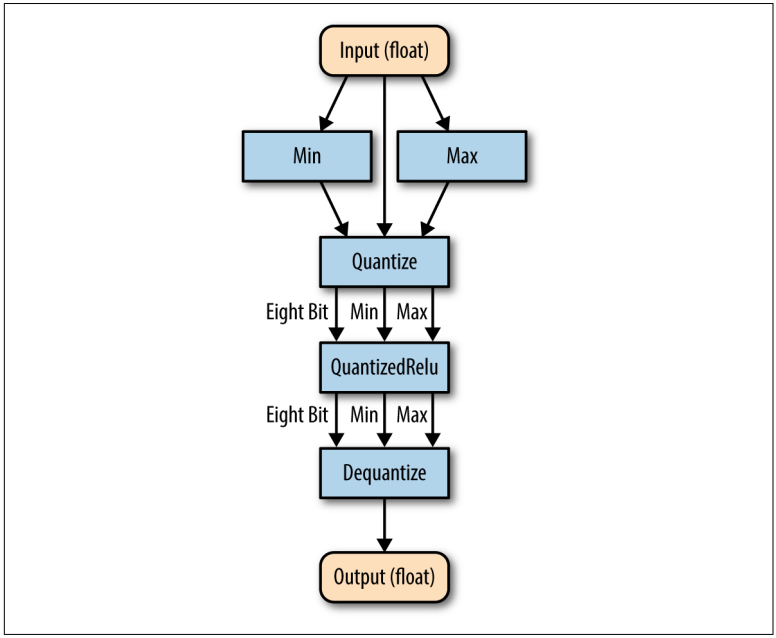

Figure 1-3. Quantized Relu graph

Handle this conversion for the inputs using the Quantize op. As we mentioned previously, quantized buffers only make sense when we also know what range was used to encode them, so this op outputs the float minimum and maximum as well as as the coded 8-bit values. This is then operated on by the QuantizedRelu op, which outputs 8-bit values together with the range. In fact, for this implementation the output range is the same as the input, so we could wire the min/max from Quantize directly as inputs to Dequantize, but to keep the implementation consistent it's better to always have range outputs for every quantized one as a convention.

The encoded 8-bit values from QuantizedRelu are fed into a Dequantize op together with the range, which produces a final float output. All in all, this probably seems a very convoluted (if you'll excuse the pun) way to handle 8-bit calculations. The important part though is that this is a generic way to substitute any individual op in the graph with an 8-bit equivalent without needing to understand any of the larger picture. We can implement these substitutions as a

first pass, and then go through the resulting graph and remove inefficiencies. Figure 1-4 shows how that works if you have a pair of 8-bit operations with a dequantize/quantize stage between them.

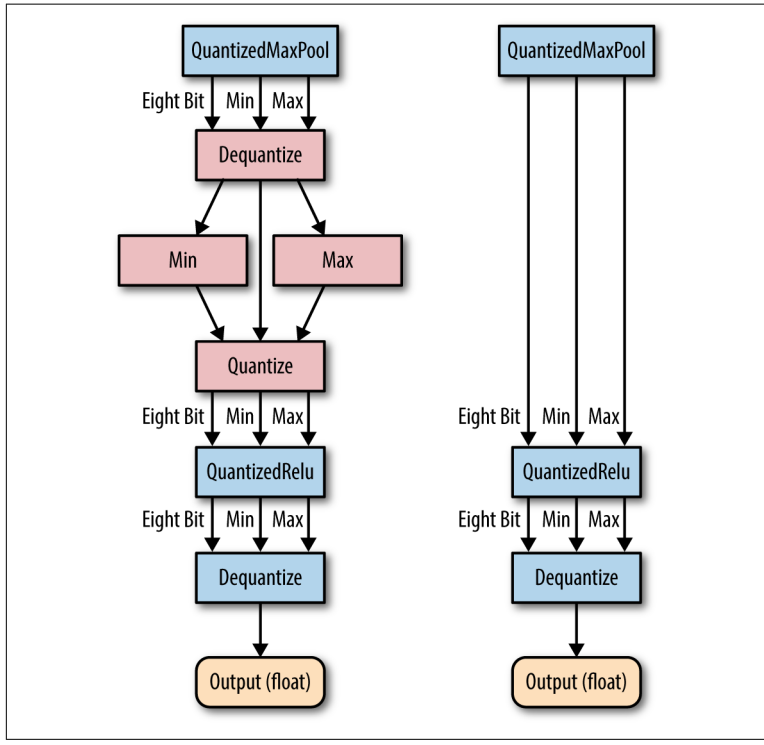

Figure 1-4. Graph showing the removal of unneeded quantization ops

Here the graph transform is able to spot that the subgraph of operations marked in red on the left all cancel each other out, producing the same output as its original input. By spotting that pattern, the transform tool can remove those unnecessary ops and produce the simplified version of the graph on the right.

Why Does Quantization Removal Matter?

Quantization removal is important because it means the performance of 8-bit graphs depends a lot on how many of the operations in the model have quantized equivalents, and if unconverted ops cause a lot of conversions back and forth between float and 8-bit. A few conversions at the beginning or end of a graph won't matter too much; for example, SoftMax usually works with a very small amount

of data and comes as the final step of a graph, so it's not usually a bottleneck. However, having a mix of float and 8-bit in the heart of your graph can squander any advantages of moving to the lower bit depth for calculations.

Another problem to watch out for is that running the algorithms efficiently for 8-bit requires architecture-specific SIMD code. We use the gemmlowp library to implement the matrix multiplies that make up the bulk of neural network calculations, but currently that's only optimized for ARM NEON and mobile Intel chips. That means desktop performance for 8-bit on x86 is actually worse than float, because we use floating-point libraries like Eigen that have been highly optimized for those chips, whereas the 8-bit code hasn't been.

Activation Ranges

One challenge we haven't talked about is that some operations that take in 8-bit inputs actually produce 32-bit outputs. For example, if you're doing a matrix multiply, each output value will be the sum of a series of 8-bit input numbers multiplied with each other. The result of multiplying 2 8-bit inputs is a 16-bit value, and then to accurately accumulate a number of them, you need something larger than 16 bits, which on most chips means a 32-bit value. Subsequent quantized operations that use this result as an input don't want 32 bits of data, though, because that's no more efficient than a float value and is considerably harder to deal with. Instead, we convert those 32 bits into 8-bit equivalents.

You could imagine just calculating the smallest and largest possible values that could be produced from a particular matrix multiplication, and using those as the range to extract the highest eight bits from the wider output values. This encoding would be very inefficient, however, because the actual inputs to most neural network operations don't have extreme distributions, so the everyday smallest and largest values that will be encountered are much tinier than their theoretical limits. Using the extreme ranges would mean that most of the bits in the coding would be wasted.

To address that issue, we need to know what the extremes will be for commonly encountered data. Unfortunately, this information has proved to be very hard to calculate analytically, so we've ended up having to empirically observe the statistics while running real exam-

ples through the entire network. There are three main ways of handling this, described in the following sections.

Dynamic Ranges

The easiest way to get started is to insert an op that runs through the output from a 32-bit-producing operation just after it's been generated and figures out what the current range of those values actually is. This range can then be fed into a Requantization operation that converts 32-bit tensors into 8-bits, given a target range for the output.

The big advantage of this method is that it doesn't need any extra data or user intervention, so it's the default way that the `quantize_nodes` transform uses. This method makes it straightforward to take a float network, convert it to eight bits, and then start checking the accuracy and performance.

The downside is that the range calculation has to be run every time inference is performed, which means looking through every output value and figuring out the minimum and maximum across each buffer. This is extra work that reduces performance (usually by a fairly small amount) on the CPU; but for specialized hardware platforms it's even worse, because they may not be able to handle this sort of dynamic rescaling.

Observed Ranges

The next-easiest approach is to run a representative set of example data through the network, track what the ranges are for each op over time, and then use a statistical methodology to estimate reasonable values to cover all those ranges without wasting too much precision. Unfortunately this approach is hard to do automatically, since the idea of what constitutes representative data is very application-dependent. For example, with an image-recognition network, plugging in random noise or extreme values as inputs would only exercise a few of the pattern recognition paths, so the resulting ranges wouldn't be very useful. Instead, you need to have inputs that represent the kind of data that's expected, like training data.

To make this possible, we offer a multistage approach. By using the `insert_logging` rule, you can add debug ops that output the values of the ranges every time the model is run. Here's a complete example

of how to do everything you need on the pretrained InceptionV3 graph.

First, download and decompress the model file:

```
mkdir /tmp/model/
curl\
 "https://storage.googleapis.com/download.tensorflow.org/ \
 models/inception_dec_2015.zip" \
 -o /tmp/model/inception_dec_2015.zip
unzip /tmp/model/inception_dec_2015.zip -d /tmp/model/
```

Then quantize the graph to use 8-bit calculations:

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
 --in_graph="/tmp/model/tensorflow_inception_graph.pb" \
 --out_graph="/tmp/model/quantized_graph.pb" --inputs='Mul:0' \
 --outputs='softmax:0' --transforms='add_default_attributes
strip_unused_nodes(type=float, shape="1,299,299,3")
remove_nodes(op=Identity, op=CheckNumerics)
fold_old_batch_norms
quantize_weights
quantize_nodes
Strip_unused_nodes'
```

Once that's complete, run the `label_image` example to make sure the model is still giving the expected results. It runs on a picture of Grace Hopper by default, so you should see `Uniform` as the top result:

```
bazel build tensorflow/examples/label_image:label_image
bazel-bin/tensorflow/examples/label_image/label_image \
 --input_mean=128 --input_std=128 --input_layer=Mul \
 --output_layer=softmax --graph=/tmp/model/quantized_graph.pb \
 --labels=/tmp/model/imagenet_comp_graph_label_strings.txt
```

With that working, next you'll append log operations onto the outputs of all the RequantizationRange nodes:

```
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
 --in_graph=/tmp/model/quantized_graph.pb \
 --out_graph=/tmp/model/logged_quantized_graph.pb \
 --inputs=Mul \
 --outputs=softmax \
--transforms='insert_logging(op=RequantizationRange, \
 show_name=true, message="__requant_min_max:")'
```

Now you can run the graph, and `stderr` should contain log statements showing what values the ranges have on that run:

```
bazel-bin/tensorflow/examples/label_image/label_image \
 --input_mean=128 --input_std=128 \
 --input_layer=Mul --output_layer=softmax \
 --graph=/tmp/model/logged_quantized_graph.pb \
 --labels=/tmp/model/imagenet_comp_graph_label_strings.txt 2> \
 /tmp/model/logged_ranges.txt
cat /tmp/model/logged_ranges.txt
```

You should see a series of lines like this:

```
;conv/Conv2D/eightbit/requant_range__print*; \
  *requant_min_max:[-20.887871] [22.274715]
```

Each one of these encodes the range values for a given operation. In this case, we're just running the graph once on a single image. In a real application, we'd want to run hundreds of representative images to get a good sample of all the usual ranges.

Finally, we want to take the information we've gathered and replace the dynamic range calculations with simple constants, using the `freeze_requantization_ranges` transform:

```
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
  --in_graph=/tmp/model/quantized_graph.pb \
  --out_graph=/tmp/model/ranged_quantized_graph.pb \
  --inputs=Mul \
  --outputs=softmax \
--transforms='freeze_requantization_ranges \
  (min_max_log_file=/tmp/model/logged_ranges.txt)'
```

If you run the `label_image` example on this new network, you should see `uniform` as the top result still, though the exact numbers may be slightly different from before:

```
bazel-bin/tensorflow/examples/label_image/label_image \
 --input_mean=128 --input_std=128 --input_layer=Mul \
 --output_layer=softmax \
 --graph=/tmp/model/ranged_quantized_graph.pb \
 --labels=/tmp/model/imagenet_comp_graph_label_strings.txt
```

Trained Ranges

The final way to figure out good ranges for the activation layers is to integrate the calculations into the training process. You can do this using the `FakeQuantWithMinMaxVars` node. This operation can be placed at various points in the graph to simulate quantization inaccuracies by rounding its float inputs to a fixed number of levels (typically 256), within a range set by two `Variable` inputs representing the minimum and maximum. These range inputs are updated dur-

ing the learning process based on the minimums and maximums actually required, as the gradient values are passed through to these inputs.

Unfortunately, there aren't any convenience functions to add these ops to your models in Python, so if you do go down this route you'll need to manually insert the ops and variables anywhere a range is needed. This will typically be between weights and the ops they're used in, and on the outputs of Conv2D or MatMul nodes.

There are a couple of big advantages to this approach that make up for the inconvenience. When we take a pretrained float model and simply convert it to 8-bit, there's typically a small loss in accuracy—for example, a drop in top-1 precision on InceptionV3 from 78% to 77%. Training with the quantization baked in usually shrinks that loss dramatically, sometimes to the point where the networks perform as well as float. Having the ranges known ahead of time also helps latency during inference, since we don't have to do runtime calculations to determine the minimum and maximum.

What Next?

With any luck, this guide has given you enough information to start building your own mobile and embedded applications. There are a massive number of important problems in all sorts of fields, from ecology to education, that can benefit from on-device deep learning, and our goal is to help make it easier to create solutions to some of those challenges.

As an open source framework, we're always excited to get feedback, hear about bugs, and receive ideas on improving the experience. As discussed in "How the TensorFlow Team Handles Open Source Support", issues on GitHub, or questions on StackOverflow are very welcome, and we're looking forward to seeing what you can build!

About the Author

Pete Warden is the tech lead on the Mobile/Embedded TensorFlow team and was the CTO of Jetpac, acquired by Google in 2014 for its deep learning technology optimized to run on mobile and embedded devices.