*1* Introduction

*2* Setting Up

*3* Installing and Running the TensorFlow Docker Image

*4* Retrieving the images

*5* (Re)training Inception

*6* Using the Retrained Model

*7* Optional Step: Trying Other Hyperparameters

*8* Optional Step: Training on Your Own Categories

*9* Next Steps

Please visit googlecodelabs for official steps.

## 1. Introduction

[TensorFlow](http://tensorflow.org/) is a an open source library for numerical computation, specializing in machine learning applications. In this codelab, you will learn how to install and run TensorFlow on a single machine, and will train a simple classifier to classify images of flowers.

What are we going to be building?

In this lab, we will be using transfer learning, which means we are starting with a model that has been already trained on another problem. We will then be retraining it on a similar problem. Deep learning from scratch can take days, but transfer learning can be done in short order.

We are going to use the Inception v3 network. Inception v3 is a trained for the [ImageNet](http://image-net.org/) Large Visual Recognition Challenge using the data from 2012, and it can differentiate between 1,000 different classes, like Dalmatian or dishwasher. We will use this same network, but retrain it to tell apart a small number of classes based on our own examples.

What you will learn

* How to install and run TensorFlow Docker images
* How to use Bazel and Python to train an image classifier
* How to classify images with your trained classifier

What you need

* A basic understanding of Unix commands
* A fast computer running OS X or Linux
* A fair amount of time

This codelab does not cover Windows. Adventuresome folks have [had some success](https://github.com/tensorflow/tensorflow/issues/42) getting the Docker image working on Windows, but it's not currently easy or recommended. Running natively on Windows is not possible because [Bazel](http://bazel.io/) does not support Windows at this time.

## 2. Setting Up

## Installing Docker

Although it's quite possible to install TensorFlow natively on your machine (and recommended for long-term experimentation), we have built a Docker package that contains all the dependencies you need to run TensorFlow, along with the sample code.

### On Linux

You can find Docker for Linux installation instructions on the [Docker site](https://docs.docker.com/engine/installation/).

### On OS X

For developers using OS X workstations located at Google I/O, the Docker runtime and container should already be available. Skip ahead to "Running Faster."

If you're at Google I/O or already have the Docker Toolbox installed, skip to "Installing/running a TensorFlow Docker Image." Otherwise, go to [docs.docker.com/mac/](https://docs.docker.com/mac/) and follow the instructions there, which should roughly be:

* [Download the Docker Toolbox](https://www.docker.com/products/docker-toolbox).
* On the Toolbox page, find the Mac version.
* Download a DockerToolbox-1.xx.xx.pkg file (180MB).
* Run that downloaded pkg to install the Toolbox.
* At the end of the install process, choose the Docker Quickstart Terminal.
* Open a terminal window and follow the installation script.
* If you have already installed Toolbox, just run the Docker Quickstart Terminal in /Applications/Docker/.
* Launch the Docker Quickstart Terminal and run the suggested command in the terminal:
* docker run hello-world
* This should confirm your installation of Docker has worked.

#### Troubleshooting OS X

On OS X, you may see:

Error checking TLS connection: Something went wrong running an SSH command!

command : ip addr show

err : exit status 255

At which point you may have to destroy and recreate your docker instance like so:

% docker-machine rm default

% docker-machine create --driver virtualbox default

## 3. Installing and Running the TensorFlow Docker Image

#### On OS X, if you have not already, run the Docker Quickstart Terminal, usually found in /Applications/Docker, and which looks like this:



NOTE:

install bazel : <https://bazel.build/versions/master/docs/install.html#ubuntu>

There will be a long pause as the Docker container starts and gets an SSH keys and an IP address.

Once you see an ASCII whale in the newly-opened terminal, run this command:

% docker run -it gcr.io/tensorflow/tensorflow:latest-devel

Check to see if your TensorFlow works by invoking Python from the container's command line (you'll see "root@xxxxxxx#"):

# python

import tensorflow as tf

hello = tf.constant('Hello, TensorFlow!')

sess = tf.Session()

print(sess.run(hello))

If you see "Hello, Tensorflow!", it works!

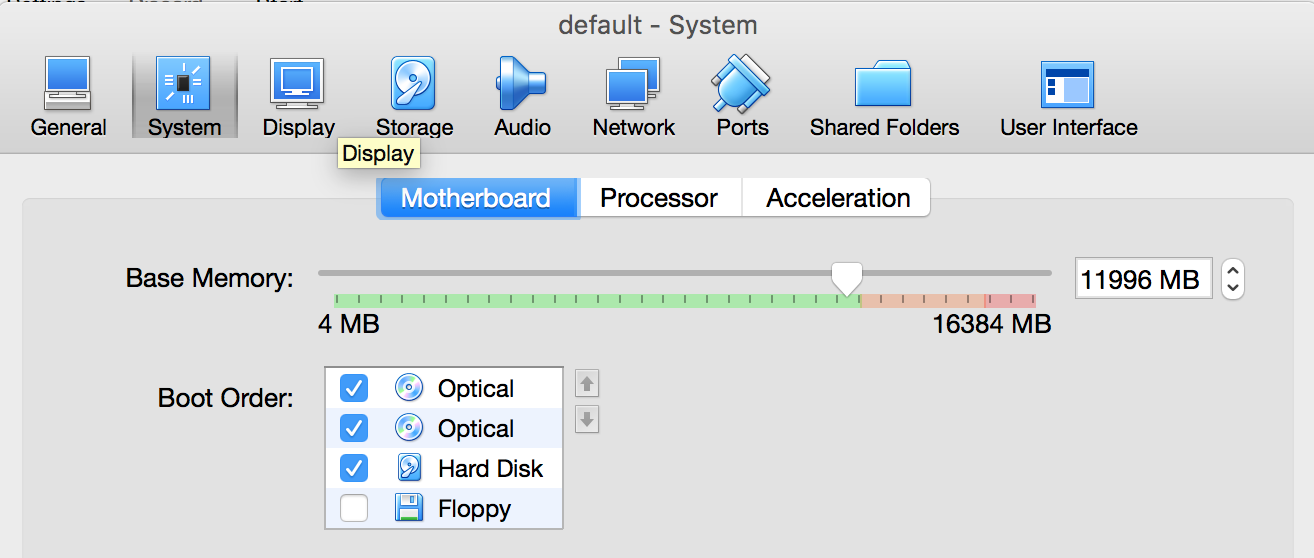
## Running Faster

Training takes a long time, even a limited run like this. It helps to provide a lot of CPU and memory resources if you can.

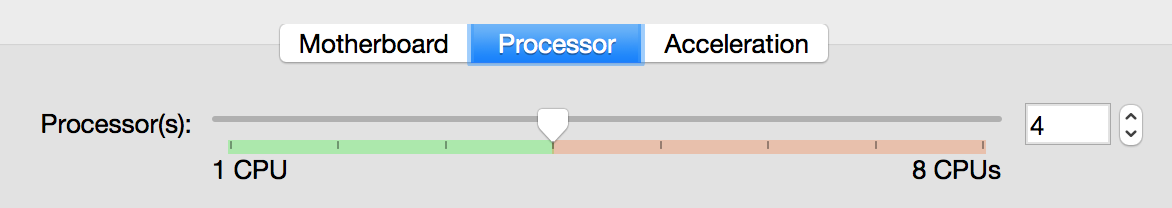
#### Making OS X Run Faster

These Docker instances are running on VirtualBox, and VirtualBox by default doesn't take over huge parts of your computer. TensorFlow will take a lot of resources, so it's worth increasing the resources allocated to it.

* Find VirtualBox in your Application folder, and open it.
* Stop the running process, if there is one
* Right-click on "Defaulct" and choose Stop -> Close ACPI Shutdown
* Click on Settings, then System
* Choose Motherboard and boost the RAM to the edge of the green area.



* Click on Processor, and, again, boost that to the edge of the green area.



* Click OK.
* Restart VirtualBox by right-clicking on "default" and choosing Start -> Headless Start, then wait a bit until it's Running.
* Re-run the Docker Quickstart Terminal.



* Your session will have ended. Optionally, you can verify that the Docker image still works:

% docker run hello-world

## 4. Retrieving the images

Before you start any training, you'll need a set of images to teach the network about the new classes you want to recognize. We've created an archive of creative-commons licensed flower photos to use initially. To get the set of flower photos, exit your Docker instance (ctrl-D, or type exit) and return to your home directory.

# ctrl-D if you're still in Docker and then:

% cd $HOME

% mkdir tf\_files

% cd tf\_files

% curl -O http://download.tensorflow.org/example\_images/flower\_photos.tgz

% tar xzf flower\_photos.tgz

# On OS X, see what's in the folder:

% open flower\_photos

After downloading 218MB, you should now have a copy of the flower photos available in your home directory.

Important: Training on this much data can take 30+ minutes on a small computer, like the kiosks at Google I/O. See the next section to speed up your test.

### Optional: But I'm in a hurry!

Let's reduce the number of categories we'll learn, and only tell the difference between roses and daisies for now.

% cd $HOME/tf\_files/flower\_photos

% rm -rf dandelion sunflowers tulips

# On OS X, make sure the flowers are gone!

% open .

## Start Docker with local files available

The TensorFlow Docker image doesn't contain the flower data, so we'll make it available by linking it in virtually.

% docker run -it -v $HOME/tf\_files:/tf\_files gcr.io/tensorflow/tensorflow:latest-devel

At the Docker prompt, you can see it's linked as a toplevel directory.

# ls /tf\_files/

flower\_photos flower\_photos.tgz

## Retrieving the training code

The Docker image you are using contains the latest GitHub TensorFlow tools, but not every last sample. You need to retrieve the full sample set this way:

# cd /tensorflow

# git pull

Your sample code will now be in /tensorflow/tensorflow/examples/image\_retraining/.

## 5. (Re)training Inception

At this point, we have a trainer, we have data, so let's train! We will train the Inception v3 network.

As noted in the introduction, Inception is a huge image classification model with millions of parameters that can differentiate a large number of kinds of images. We're only training the final layer of that network, so training will end in a reasonable amount of time.

Start your image retraining with one big command:

# python tensorflow/examples/image\_retraining/retrain.py \

--bottleneck\_dir=/tf\_files/bottlenecks \

--how\_many\_training\_steps 500 \

--model\_dir=/tf\_files/inception \

--output\_graph=/tf\_files/retrained\_graph.pb \

--output\_labels=/tf\_files/retrained\_labels.txt \

--image\_dir /tf\_files/flower\_photos

This script loads the pre-trained Inception v3 model, removes the old final layer, and trains a new one on the flower photos you've downloaded.

ImageNet was not trained on any of these flower species, originally. However, the kinds of information that make it possible for ImageNet to differentiate among 1,000 classes are also useful for distinguishing other objects. By using this pre-trained network, we are using that information as input to the final classification layer that distinguishes our flower classes.

Important: The script can take thirty minutes or more to complete, depending on the speed of your machine. If you are not in a hurry, read the next section.

#### Optional: But NOT I'm in a hurry!

The above example iterates only 500 times. If you are training all five flower classes instead of two, you will very likely get worse results (i.e. lower accuracy). To get much better results, remove the parameter --how\_many\_training\_steps to use the default 4,000 iterations.

python tensorflow/examples/image\_retraining/retrain.py \

--bottleneck\_dir=/tf\_files/bottlenecks \

--model\_dir=/tf\_files/inception \

--output\_graph=/tf\_files/retrained\_graph.pb \

--output\_labels=/tf\_files/retrained\_labels.txt \

--image\_dir /tf\_files/flower\_photos

## While You're Waiting: Bottlenecks

This section and the next are to be enjoyed while your classifier is training.

The first phase analyzes all the images on disk and calculates the bottleneck values for each of them. What's a bottleneck?

The Inception v3 model is made up of many layers stacked on top of each other (see a picture of it [in this paper](http://www.cs.unc.edu/%7Ewliu/papers/GoogLeNet.pdf)). These layers are pre-trained and are already very valuable at finding and summarizing information that will help classify most images. For this codelab, you are training only the last layer; the previous layers retain their already-trained state.

A 'Bottleneck,' then, is an informal term we often use for the layer just before the final output layer that actually does the classification.

Every image is reused multiple times during training. Calculating the layers behind the bottleneck for each image takes a significant amount of time. By caching the outputs of the lower layers on disk, they don't have to be repeatedly recalculated. By default, they're stored in the /tmp/bottleneck directory. If you rerun the script, they'll be reused, so you don't have to wait for this part again.

## While You're Waiting: Training

You will see the bottlenecks train. Once they are complete, the actual training of the final layer of the network begins.

You'll see a series of step outputs, each one showing training accuracy, validation accuracy, and the cross entropy:

* The training accuracy shows the percentage of the images used in the current training batch that were labeled with the correct class.
* Validation accuracy: The validation accuracy is the precision (percentage of correctly-labelled images) on a randomly-selected group of images from a different set.
* Cross entropy is a loss function that gives a glimpse into how well the learning process is progressing. (Lower numbers are better here.)

A true measure of the performance of the network is to measure its performance on a data set that is not in the training data. This performance is measured using the validation accuracy. If the training accuracy is high but the validation accuracy remains low, that means the network is overfitting, and the network is memorizing particular features in the training images that don't help it classify images more generally.

The training's objective is to make the cross entropy as small as possible, so you can tell if the learning is working by keeping an eye on whether the loss keeps trending downwards, ignoring the short-term noise.

By default, this script runs 4,000 training steps. Each step chooses 10 images at random from the training set, finds their bottlenecks from the cache, and feeds them into the final layer to get predictions. Those predictions are then compared against the actual labels to update the final layer's weights through a back-propagation process.

As the process continues, you should see the reported accuracy improve. After all the training steps are complete, the script runs a final test accuracy evaluation on a set of images that are kept separate from the training and validation pictures. This test evaluation provides the best estimate of how the trained model will perform on the classification task.

You should see an accuracy value of between 85% and 99%, though the exact value will vary from run to run since there's randomness in the training process. (If you are only training on two classes, you should expect higher accuracy.) This number value indicates the percentage of the images in the test set that are given the correct label after the model is fully trained.

## 6. Using the Retrained Model

The retraining script will write out a version of the Inception v3 network with a final layer retrained to your categories to /tmp/output\_graph.pb and a text file containing the labels to /tmp/output\_labels.txt.

These files are both in a format that the [C++ and Python image classification examples](https://www.tensorflow.org/versions/master/tutorials/image_recognition/index.html) can use, so you can start using your new model immediately.

You now have two choices: Python or C++.

### Using Python

This is recommended if you are at Google I/O using the kiosks, as compiling TensorFlow in C++ can be a long process.

Here is Python that loads your new graph file and predicts with it.

### label\_image.py

import tensorflow as tf

# change this as you see fit

image\_path = sys.argv[1]

# Read in the image\_data

image\_data = tf.gfile.FastGFile(image\_path, 'rb').read()

# Loads label file, strips off carriage return

label\_lines = [line.rstrip() for line

in tf.gfile.GFile("/tf\_files/retrained\_labels.txt")]

# Unpersists graph from file

with tf.gfile.FastGFile("/tf\_files/retrained\_graph.pb", 'rb') as f:

graph\_def = tf.GraphDef()

graph\_def.ParseFromString(f.read())

\_ = tf.import\_graph\_def(graph\_def, name='')

with tf.Session() as sess:

# Feed the image\_data as input to the graph and get first prediction

softmax\_tensor = sess.graph.get\_tensor\_by\_name('final\_result:0')

predictions = sess.run(softmax\_tensor, \

{'DecodeJpeg/contents:0': image\_data})

# Sort to show labels of first prediction in order of confidence

top\_k = predictions[0].argsort()[-len(predictions[0]):][::-1]

for node\_id in top\_k:

human\_string = label\_lines[node\_id]

score = predictions[0][node\_id]

print('%s (score = %.5f)' % (human\_string, score))

This is a little clumsy to cut-and-paste, so we've made a gist for you.

Exit your Docker image, and go to $HOME/tf\_files. Create a file called label\_image.py and put the above code into it.

# ctrl-D to exit Docker and then:

% curl -L https://goo.gl/tx3dqg > $HOME/tf\_files/label\_image.py

Restart your Docker image:

% docker run -it -v $HOME/tf\_files:/tf\_files gcr.io/tensorflow/tensorflow:latest-devel

Now, run the Python file you created, first on a daisy:

# python /tf\_files/label\_image.py /tf\_files/flower\_photos/daisy/21652746\_cc379e0eea\_m.jpg

And then on a rose:

# python /tf\_files/label\_image.py /tf\_files/flower\_photos/roses/2414954629\_3708a1a04d.jpg

You will see many warnings; they not harmful. You will then see a list of flower labels, in most cases with the right flower on top (though each retrained model may be slightly different).

You might get results like this for a daisy photo:

daisy (score = 0.99071)

sunflowers (score = 0.00595)

dandelion (score = 0.00252)

roses (score = 0.00049)

tulips (score = 0.00032)

This indicates a high confidence it is a daisy, and low confidence for any other label.

You can use label\_image.py to choose any image file to classify, either from your downloaded collection, or new ones.

### Using C++

This is NOT recommended if you are at Google I/O using the kiosks, as compiling TensorFlow in C++ is slow.

Build the example C++ classifier like so:

# cd /tensorflow

# bazel build tensorflow/examples/label\_image:label\_image --local\_resources 2048,2.0,1.0 -j 1

[Bazel](http://bazel.io/) is TensorFlow's build system. "-j 1" reduces the parallelism of the compilation process at the cost of speed. You can leave off "--local\_resources 2048,2.0,1.0 -j 1" if you have a lot of memory available to your Docker container (10GB or more), but in resource-constrained virtual machines, the compilation may fail by running out of memory.

Once the build competes, classify images using the binary you created:

# bazel-bin/tensorflow/examples/label\_image/label\_image \

--graph=/tf\_files/retrained\_graph.pb \

--labels=/tf\_files/retrained\_labels.txt --output\_layer=final\_result \

--image=/tf\_files/flower\_photos/daisy/21652746\_cc379e0eea\_m.jpg

Since you've replaced the final layer, that is the one you will need to specify as the output layer (the layer with the prediction). If you're using label\_image, use the flag --output\_layer=final\_result if you're using label\_image.

You should see a list of flower labels, in most cases with daisy on top (though each retrained model may be slightly different). You can replace the --image parameter with your own images to try those out, and you can use the C++ code as a template to integrate with your own applications.

If you want to keep classifying images, you won't need to run bazel build after you've built once.

## 7. Optional Step: Trying Other Hyperparameters

There are several other parameters you can try adjusting to see if they help your results. The --learning\_rate controls the magnitude of the updates to the final layer during training. If this rate is smaller, the learning will take longer, but it can help the overall precision. That's not always the case, though, so you need to experiment carefully to see what works for your case.

The --train\_batch\_size parameter controls the number of images that the script examines during one training step. Because the learning rate is applied per batch, you'll need to reduce this value if you have larger batches to get the same overall effect.

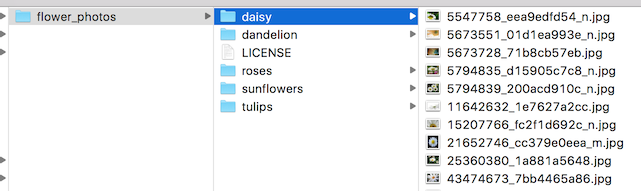
## 8. Optional Step: Training on Your Own Categories

After you see the script working on the flower example images, you can start looking at teaching the network to recognize categories you care about instead.

In theory, all you need to do is run the tool, specifying a particular set of sub-folders. Each sub-folder is named after one of your categories and contains only images from that category.

If you complete this step and pass the root folder of the subdirectories as the argument for the --image\_dir parameter, the script should train the images that you've provided, just like it did for the flowers.

The classification script uses the folder names as label names, and the images inside each folder should be pictures that correspond to that label, as you can see in the flower archive:



Collect as many pictures of each label as you can and try it out!

## 9. Next Steps

Congratulations, you've taken your first steps into a larger world of deep learning!

You can see more about using TensorFlow at the [TensorFlow website](http://tensorflow.org/) or the TensorFlow [Github project](https://github.com/tensorflow/). There's a list of other [TensorFlow resources](https://www.tensorflow.org/versions/r0.8/resources/index.html) on the TensorFlow site, including a discussion group and whitepaper.

If you make a trained model that you want to run in production, you should also check out [TensorFlow Serving](https://tensorflow.github.io/serving/), an open source project that makes it easier to manage TensorFlow projects.