1. Introduction

In this codelab, we will learn how to run a pre-trained model on a single machine.
The model is trained on Image-Net Data-Set.
Further, we will use the model to re-train it on the Cifar-10 Data-Set.

What we will build

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 retrain 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 a model trained on the ImageNet Large Visual Recognition Challenge dataset. These models can differentiate between 1,000 different classes, like Dalmatian or dishwasher. We will have a choice of model architectures, so that we can determine the right tradeoff between speed, size and accuracy for our problem.
We will use this same model, but retrain it to tell apart a small number of classes based on our own examples.

What we will Learn

- How to use Python and TensorFlow to train an image classifier
- How to classify images with our trained classifier

2. Setup

Install TensorFlow

Before we can begin the tutorial, we need to install TensorFlow version = 1.7.*

Clone the git repository

All the code used in this codelab is contained in this git repository. Clone the repository and `cd` into it. This is where we will be working.

```
git clone https://github.com/googlecodelabs/tensorflow-for-poets-2
```

```
cd tensorflow-for-poets-2
```

3. Download the training images

Before we start any training, we will need training images to teach the model about the new classes we want to recognize. We've created an archive of the Cifar-10 DataSet to use initially. Download the photos (42 MB) by invoking the following command:

```
curl https://drive.google.com/open?id=1nTU0DLcOWET6GCAN_VfoMwmwJWEBriYS \
    | tar xz -C tf_files
```

We should now have a copy of the photos. We can confirm the contents of the working directory by issuing the following command:

```
ls tf_files/Cifar10-Train
```

The preceding command should display the sub-folders:

4. (Re)training the network

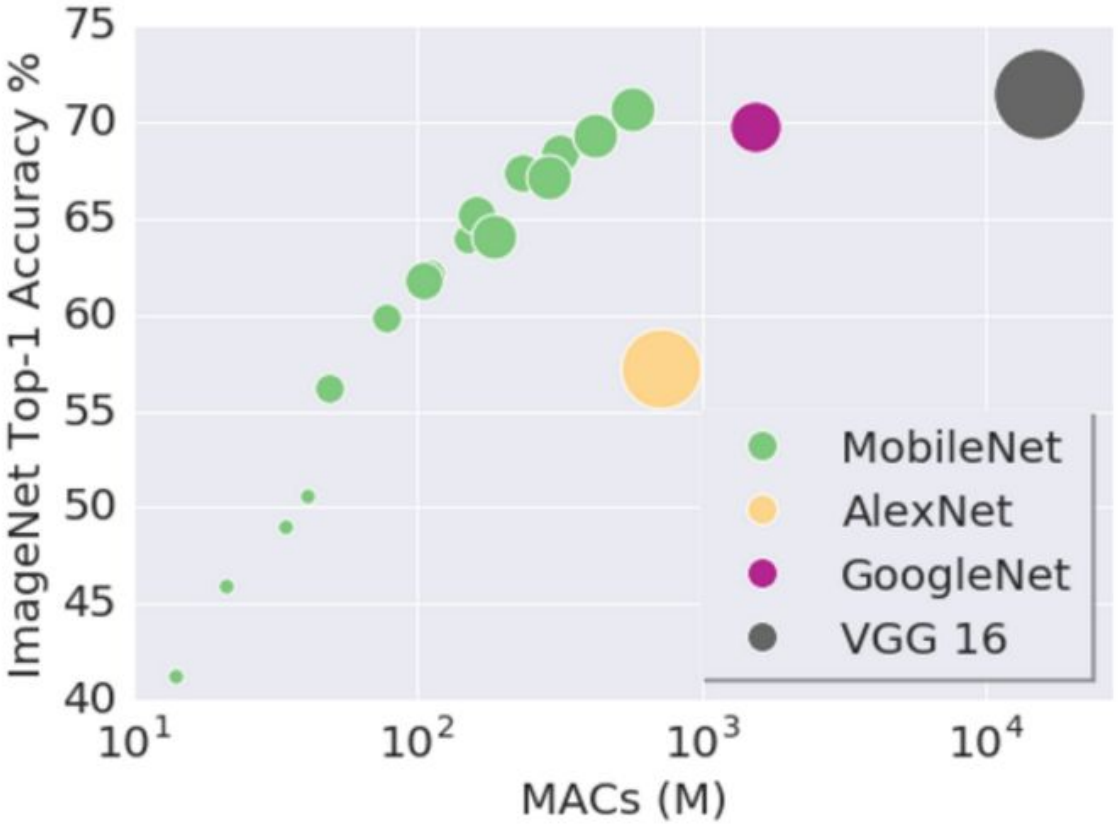
Configure MobileNet

In this exercise, we will retrain a MobileNet. MobileNet is a a small efficient convolutional neural network. "Convolutional" just means that the same calculations are performed at each location in the image.
The MobileNet is configurable in two ways:

- Input image resolution: 128,160,192, or 224px. Unsurprisingly, feeding in a higher resolution image takes more processing time, but results in better classification accuracy.
- The relative size of the model as a fraction of the largest MobileNet: 1.0, 0.75, 0.50, or 0.25.

We will use 128 1.0 for this codelab.
With the recommended settings, it typically takes only a couple of minutes to retrain on a laptop.
We will pass the settings inside Linux shell variables:

```
IMAGE_SIZE=224
ARCHITECTURE="mobilenet_1.0_${IMAGE_SIZE}"
```



Start TensorBoard

Before starting the training, launch `tensorboard` in the background. TensorBoard is a monitoring and inspection tool included with tensorflow. We will use it to monitor the training progress.

```
tensorboard --logdir tf_files/training_summaries &
```

Investigate the retraining script

The retrain script is from the TensorFlow Hub repo, but it is not installed as part of the `pip` package. So for simplicity we have included it in the codelab repository. We can run the script using the `python` command. Take a minute to skim its "help".

```
python -m scripts.retrain -h
```

Run the training

As noted in the introduction, ImageNet models are networks with millions of parameters that can differentiate a large number of classes. We're only training the final layer of that network, so training will end in a reasonable amount of time.
Start retraining with one big command (note the `--summaries_dir` option, sending training progress reports to the directory that tensorboard is monitoring) :

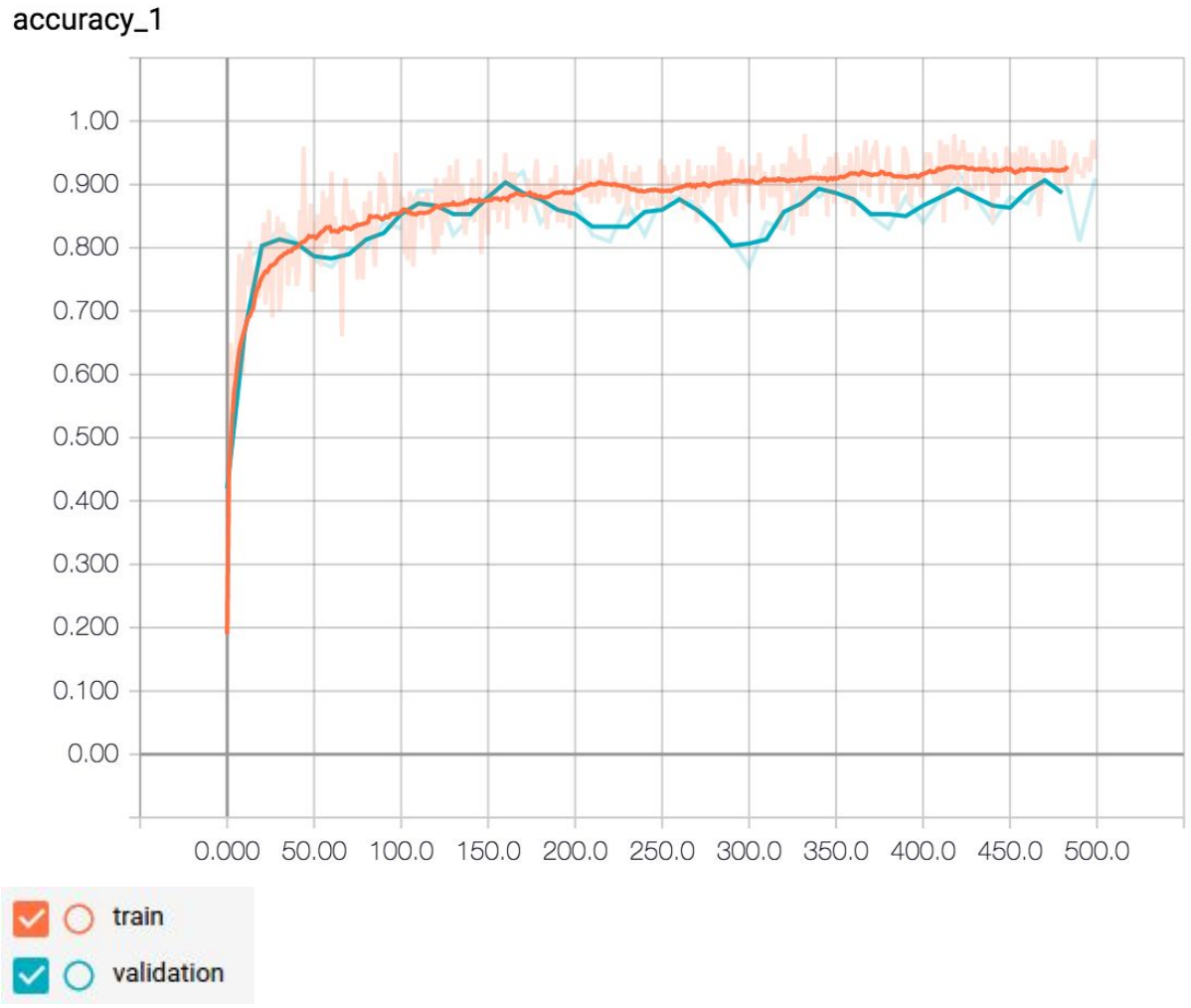
```
python -m scripts.retrain \
 --bottleneck_dir=tf_files/bottlenecks \
 --model_dir=tf_files/models/"${ARCHITECTURE}" \
 --summaries_dir=tf_files/training_summaries/"${ARCHITECTURE}" \
 --output_graph=tf_files/retrained_graph.pb \
 --output_labels=tf_files/retrained_labels.txt \
 --architecture="${ARCHITECTURE}" \
 --image_dir=tf_files/Cifar10-Train
```

5. Training And TensorBoard (Optional)

Once the script finishes generating all the bottleneck files, the actual training of the final layer of the network begins.
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, and the results of this comparison is used to update the final layer's weights through a backpropagation process. As it trains, we can 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.)

The figures below show an example of the progress of the model's accuracy and cross entropy as it trains. If our model has finished generating the bottleneck files, we can check our model's progress by opening TensorBoard, and clicking on the figure's name to show them. Ignore any warnings that TensorBoard prints to our command line.
The first figure shows accuracy (y-axis) as a function of training progress (x-axis):



Two lines are shown. The orange line shows the accuracy of the model on the training data. While the blue line shows the accuracy on the test set (which was not used for training). This is a much better measure of the true performance of the network. If the training accuracy continues to rise while the validation accuracy decreases then the model is said to be "overfitting". Overfitting is when the model begins to memorize the training set instead of understanding general patterns in the data.

As the process continues, we 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.

We 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. 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 writes data to the following two files:

- tf_files/retrained_graph.pb, which contains a version of the selected network with a final layer retrained on our categories.

- tf_files/retrained_labels.txt, which is a text file containing labels.

Classifying an image

The codelab repo also contains a copy of tensorflow's label_image.py example, which we can use to test our network. Take a minute to read the help for this script:

```
python -m scripts.label_image -h
```

Now, let's run the script on an image:

```
python -m scripts.label_image \
  --graph=tf_files/retrained_graph.pb  \
  --image=<LOCATION OF IMAGE>.jpg
```

Each execution will print a list of labels, in most cases with the correct label on top.