W251_hw5 by abhisha@berkeley.edu

What is TensorFlow? Which company is the leading contributor to TensorFlow?

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 on November 9, 2015.

What is TensorRT? How is it different from TensorFlow?

NVIDIA TensorRT™ is an SDK for high-performance deep learning inference. It includes a deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications.

TensorRT-based applications perform up to 40x faster than CPU-only platforms during inference. With TensorRT, you can optimize neural network models trained in all major frameworks, calibrate for lower precision with high accuracy, and finally deploy to hyperscale data centers, embedded, or automotive product platforms.

TensorRT is built on CUDA, NVIDIA's parallel programming model, and enables you to optimize inference for all deep learning frameworks leveraging libraries, development tools and technologies in CUDA-X for artificial intelligence, autonomous machines, high-performance computing, and graphics.

TensorRT provides INT8 and FP16 optimizations for production deployments of deep learning inference applications such as video streaming, speech recognition, recommendation and natural language processing. Reduced precision inference significantly reduces application latency, which is a requirement for many real-time services, auto and embedded applications.

You can import trained models from every deep learning framework into TensorRT. After applying optimizations, TensorRT selects platform specific kernels to maximize performance on Tesla GPUs in the data center, Jetson embedded platforms, and NVIDIA DRIVE autonomous driving platforms.

With TensorRT developers can focus on creating novel AI-powered applications rather than performance tuning for inference deployment.

It is different from Tensorflow in the following way:

https://blog.tensorflow.org/2019/06/high-performance-inference-with-TensorRT.html

Once trained, a model can be deployed to perform inference. You can find several pre-trained deep learning models on the TensorFlow GitHub site as a starting point. These models use the latest TensorFlow

APIs and are updated regularly. While you can run inference in TensorFlow itself, applications generally deliver higher performance using TensorRT on GPUs. TensorFlow models optimized with TensorRT can be deployed to T4 GPUs in the datacenter, as well as Jetson Nano and Xavier GPUs.

So what is TensorRT? NVIDIA TensorRT is a high-performance inference optimizer and runtime that can be used to perform inference in lower precision (FP16 and INT8) on GPUs. Its integration with TensorFlow lets you apply TensorRT optimizations to your TensorFlow models with a couple of lines of code. You get up to 8x higher performance versus TensorFlow only while staying within your TensorFlow environment. The integration applies optimizations to the supported graphs, leaving unsupported operations untouched to be natively executed in TensorFlow

From the above, we conclude that TensorRT is a runtime optimization that exists on top of tensor flow models, such that we can achieve higher performance on NVIDIA trained devices.

What is ImageNet? How many images does it contain? How many classes?

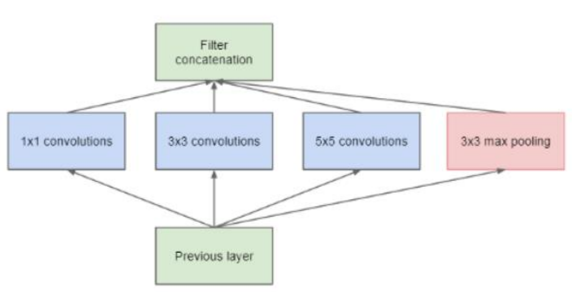
The **ImageNet** project is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided.[3] ImageNet contains more than 20,000 categories with a typical category, such as "balloon" or "strawberry", consisting of several hundred images. The database of annotations of third-party image URLs is freely available directly from ImageNet, though the actual images are not owned by ImageNet.[5] Since 2010, the ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes. The challenge uses a "trimmed" list of one thousand non-overlapping classes

Please research and explain the differences between MobileNet and GoogleNet (Inception) architectures.

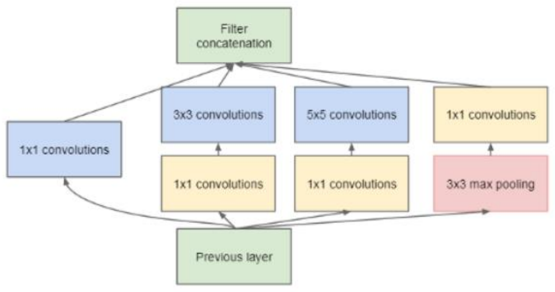
https://towardsdatascience.com/an-intuitive-guide-to-deep-network-architectures-65fdc477db41

The article above does justice in explaining the difference between MobileNet (Xception architecture) vs GoogleNet (Inception architecture), but I will try and summarize.

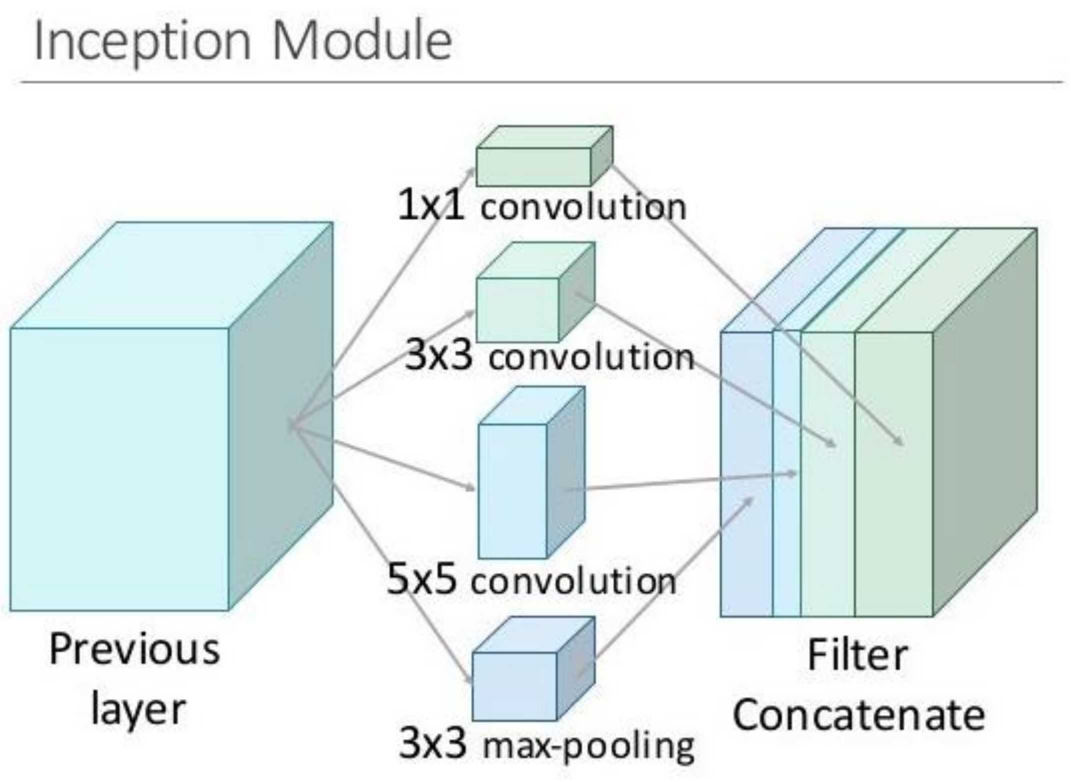
The inception architecture used by GoogleNet uses the inception module, where 1x1 filters are used to reduce the dimensionality of the image across the channel maps. The image is passed through several traditional filters (say a 3x3 or a 5x5), after being passed into a filter to reduce dimensions (1x1) – and each of these transformations (run in parallel) are “concatenated” into a single input for the next layer.



(a) Inception module, naïve version



(b) Inception module with dimension reductions



This single inception module, is then used in a series to form the inception network. The obvious problem here is that this will take some time to train because of the several parallel transformations that are being done to the image.

Basically, the main difference is that inception architecture is more computationally intensive than the Xception architecture. This is due to the architecture construction. The Xception architecture asks the following question: “is there any reason we need to consider both the image region and the channels at the same time?” – ie, Instead of partitioning input data into several compressed chunks, it maps the spatial correlations for *each output channel separately*, and then performs a 1x1 depthwise convolution to capture cross-channel correlation.

The author notes that this is essentially equivalent to an existing operation known as a “depthwise separable convolution,” which consists of a *depthwise convolution* (a spatial convolution performed independently for each channel) followed by a *pointwise convolution* (a 1x1 convolution across channels). We can think of this as looking for correlations across a 2D space first, followed by looking for correlations across a 1D space. Intuitively, this 2D + 1D mapping is easier to learn than a full 3D mapping.

This above computation is also less intensive than the inception architecture, hence, we get the perf boost in mobile net models.

In your own words, what is a bottleneck?

The bottleneck in a neural network is just a layer with less neurons then the layer below or above it. Having such a layer encourages the network to compress feature representations to best fit in the available space, in order to get the best loss during training.

In a CNN (such as Google's Inception network), bottleneck layers are added to reduce the number of feature maps (aka "channels") in the network, which otherwise tend to increase in each layer. This is achieved by using 1x1 convolutions with less output channels than input channels.

You don't usually calculate weights for bottleneck layers directly, the training process handles that, as for all other weights. Selecting a good size for a bottleneck layer is something you have to guess, and then experiment, in order to find network architectures that work well. The goal here is usually finding a network that generalizes well to new images, and bottleneck layers help by reducing the number of parameters in the network whilst still allowing it to be deep and represent many feature maps.

“From the TF for poets lab”

A **bottleneck** is an informal term we often use for the layer just before the final output layer that actually does the classification. "Bottleneck" is not used to imply that the layer is slowing down the network. We use the term bottleneck because near the output, the representation is much more compact than in the main body of the network. Every image is reused multiple times during training. Calculating the layers behind the bottleneck for each image takes a significant amount of time. Since these lower layers of the network are not being modified their outputs can be cached and reused.

How is a bottleneck different from the concept of layer freezing?

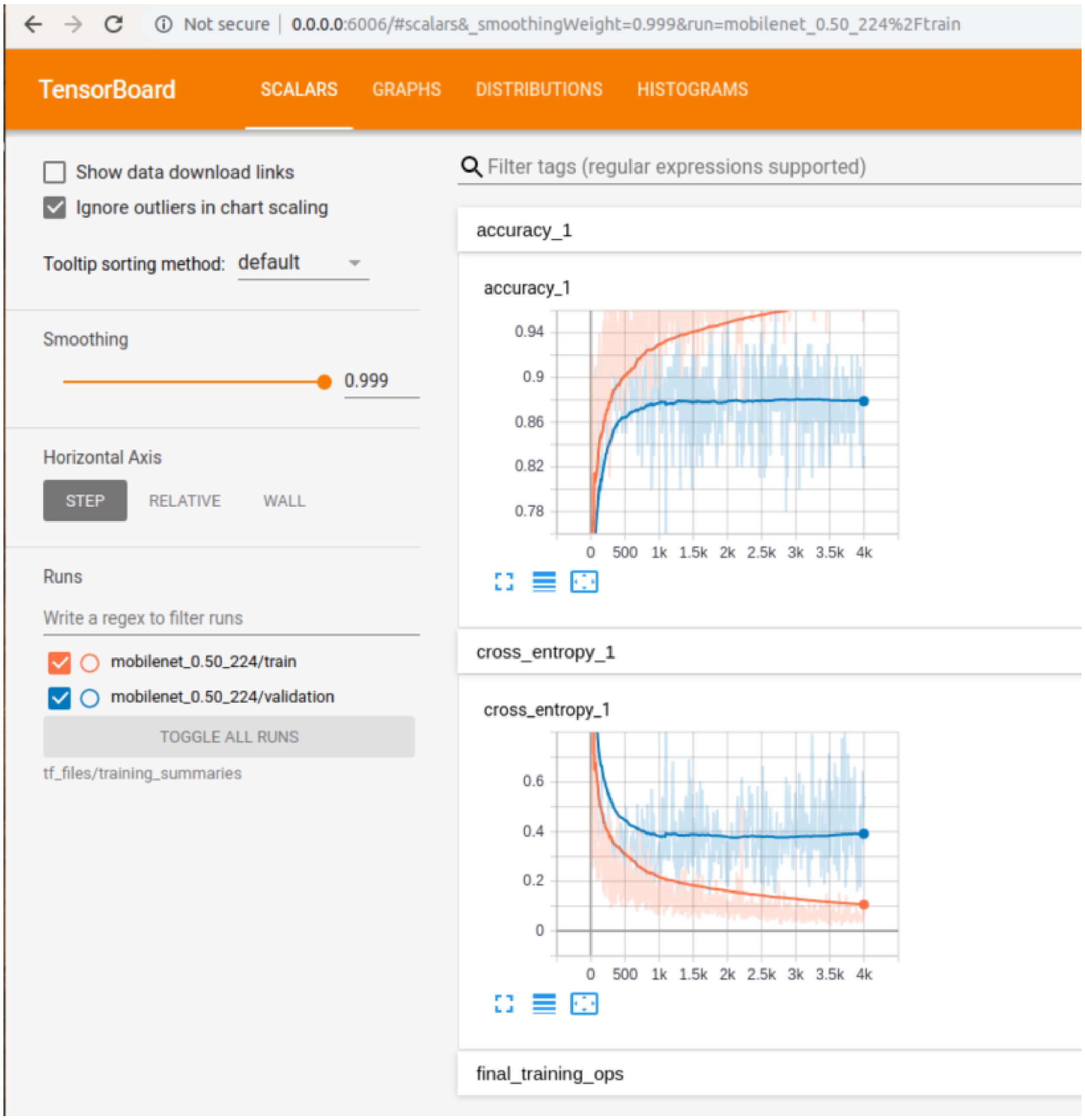
In Bottleneck layers, we are trying to achieve dimensionality reduction by reducing the number of parameters the layer must train on. This also helps in compression of the feature map representations we are achieving in that layer. We still have to learn the weights of the layer with reduced parameters. This is done by the training process.

In layer freezing, we are using a pre-trained network and are not touching any of the previously trained layers. We leave their weights as is and proceed to train the layers that are not “frozen”. This type of technique is often used in transfer learning, where we take a pre-built model, and then train the last layer (the head) against our training data. This results in reduction of training time as there are lesser number of weights to set.

In the TF1 lab, you trained the last layer (all the previous layers retain their already-trained state). Explain how the lab used the previous layers (where did they come from? how were they used in the process?)

The lab uses the mobilenet 0.5 model (size) with image input size of 224 x 224 pixels. This model is already pretrained with the weights for image classification. From the TF for poets lab, we see the following:

By default, this script runs 4,000 training steps. Each step chooses 10 images at random from the training set (the flower data 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.



0.0.0.0:6006/#scalars&_smoothingWeight=0.999&run=mobilenet_0.50_224%2Ftrain

☐ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.999

Horizontal Axis: STEP

Runs

☑ mobilenet_0.50_224/train

☑ mobilenet_0.50_224/validation

tf_files/training_summaries

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.

<u>For the daisy image</u>

That we invoked by doing the following:

```
python3 -m scripts.label_image \
--graph=tf_files/retrained_graph.pb  \
--image=tf_files/flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

```
Evaluation time (1-image): 11.153s

daisy (score=0.99842)

dandelion (score=0.00099)

sunflowers (score=0.00059)

roses (score=0.00000)

tulips (score=0.00000)
```

How does a low --learning_rate (step 7 of TF1) value (like 0.005) affect the precision? How much longer does training take?

```
Python3 -m scripts.retrain \
--bottleneck_dir=tf_files/bottlenecks \
--how_many_training_steps=4000 \
--model_dir=tf_files/models/ \
--summaries_dir=tf_files/training_summaries/LR_0.005/"${ARCHITECTURE}" \
--output_graph=tf_files/retrained_graph.pb \
--output_labels=tf_files/retrained_labels.txt \
--architecture="${ARCHITECTURE}" \
--image_dir=tf_files/flower_photos \
--learning_rate=0.005
```

We had achieved a baseline validation accuracy of 83% from 4000 steps with the default learning rate (0.01). In the above case, we performed the retraining with 4000 steps and LR = 0.005.

```
I0622 06:47:51.170904 548210839568 retrain.py:1082] 2020-06-22 06:47:51.170640: Step 3999: Train accuracy = 98.0%

INFO:tensorflow:2020-06-22 06:47:51.172437: Step 3999: Cross entropy = 0.095315

I0622 06:47:51.172719 548210839568 retrain.py:1084] 2020-06-22 06:47:51.172437: Step 3999: Cross entropy = 0.095315

INFO:tensorflow:2020-06-22 06:47:51.325894: Step 3999: Validation accuracy = 92.0% (N=100)

I0622 06:47:51.326171 548210839568 retrain.py:1100] 2020-06-22 06:47:51.325894: Step 3999: Validation accuracy = 92.0% (N=100)

INFO:tensorflow:Final test accuracy = 89.5% (N=362)
```

I0622 06:47:52.312216 548210839568 retrain.py:1126] ***Final test accuracy = 89.5% (N=362)***

We got a validation accuracy of 92% compared to 83% in the original model with learning rate 0.01.

The test accuracy was 89.5%

This is expected, we hope to get a higher validation accuracy when we decrease the learning rate. The model took moderately longer to train – around 15-ish mins overall training time.

How about a --learning_rate (step 7 of TF1) of 1.0? Is the precision still good enough to produce a usable graph?

```
Python3 -m scripts.retrain \
--bottleneck_dir=tf_files/bottlenecks \
--how_many_training_steps=4000 \
--model_dir=tf_files/models/ \
--summaries_dir=tf_files/training_summaries/LR_1.0/"${ARCHITECTURE}" \
--output_graph=tf_files/retrained_graph.pb \
--output_labels=tf_files/retrained_labels.txt \
--architecture="${ARCHITECTURE}" \
--image_dir=tf_files/flower_photos \
--learning_rate=1.0
```

I0622 07:08:26.544110 547742765072 retrain.py:1082] 2020-06-22 07:08:26.543765: Step 3999: ***Train accuracy = 100.0%***

INFO:tensorflow:2020-06-22 07:08:26.544719: Step 3999: ***Cross entropy = 0.000000***

I0622 07:08:26.544849 547742765072 retrain.py:1084] 2020-06-22 07:08:26.544719: Step 3999: Cross entropy = 0.000000

INFO:tensorflow:2020-06-22 07:08:26.681332: Step 3999: ***Validation accuracy = 87.0% (N=100)***

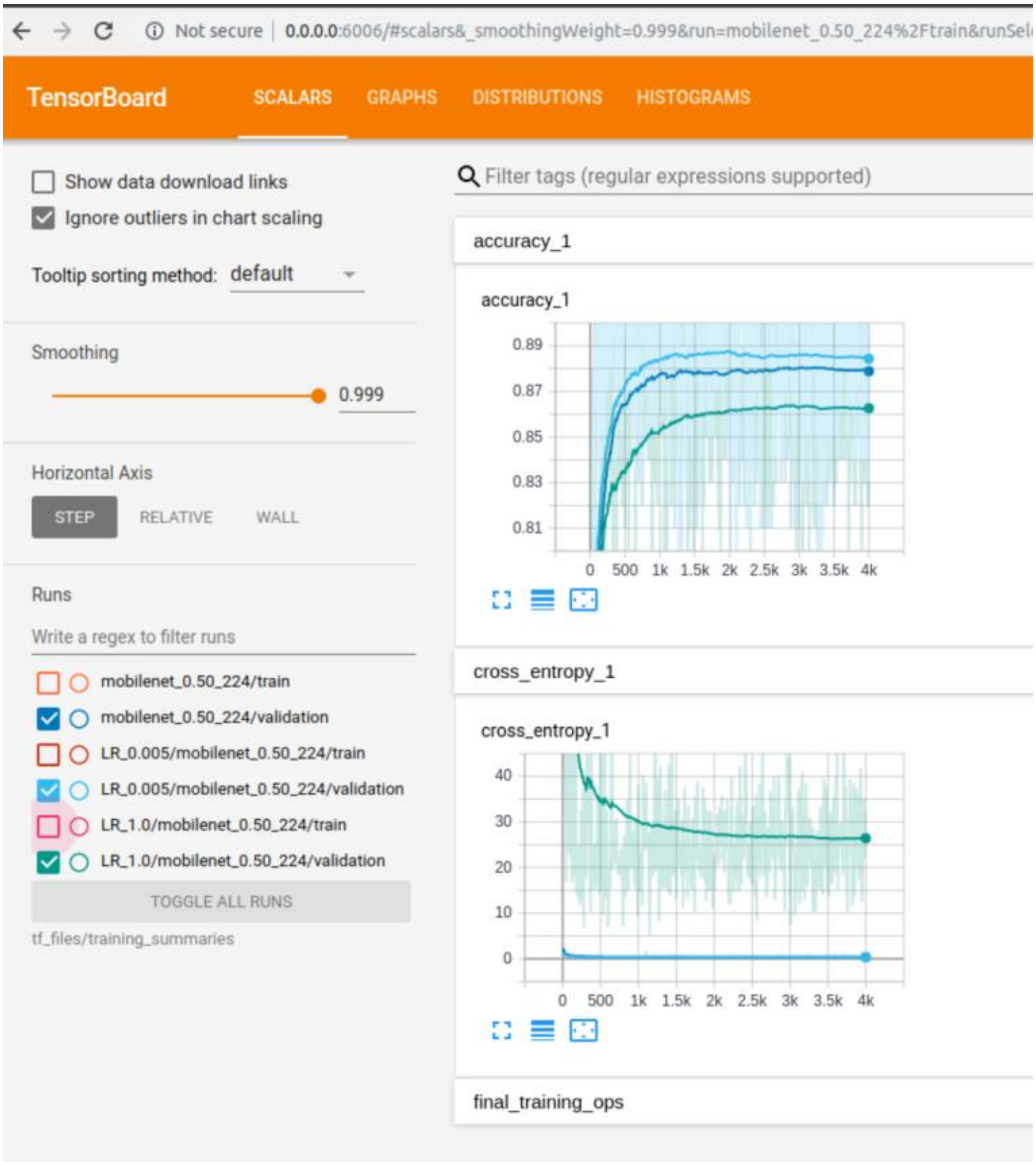
I0622 07:08:26.681708 547742765072 retrain.py:1100] 2020-06-22 07:08:26.681332: Step 3999: Validation accuracy = 87.0% (N=100)

INFO:tensorflow:Final test accuracy = 87.0% (N=362)

I0622 07:08:27.617368 547742765072 retrain.py:1126] ***Final test accuracy = 87.0% (N=362)***

The test set accuracy decreased to 87% which is not bad when compared to the LR rate of 0.005, which gave us an accuracy of 89.5%. The validation accuracy was also worse, 87% compared to 92% earlier. We could in conclusion still use LR =1.0 and use this model in production for our classification task.

This is the final graph from running training with different LR



Not secure | 0.0.0.0:6006/#scalars&_smoothingWeight=0.999&run=mobilenet_0.50_224%2Ftrain&runSel

☐ Show data download links
☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.999

Horizontal Axis
[✓ STEP] RELATIVE WALL

Runs
Write a regex to filter runs

☐ mobilenet_0.50_224/train
☑ mobilenet_0.50_224/validation
☐ LR_0.005/mobilenet_0.50_224/train
☑ LR_0.005/mobilenet_0.50_224/validation
☐ LR_1.0/mobilenet_0.50_224/train
☑ LR_1.0/mobilenet_0.50_224/validation

tf_files/training_summaries

For step 8, you can use any images you like. Pictures of food, people, or animals work well. You can even use ImageNet images. How accurate was your model? Were you able to train it using a few images, or did you need a lot?

The path to train was given as: tf_files/downloads/w251_hw5/images/dog. This corresponds to the dog folder in the images of the repository: https://github.com/abhisha1991/w251_hw5/tree/master/images

INFO:tensorflow:2020-06-22 09:15:13.065738: Step 499: Train accuracy = 100.0%

I0622 09:15:13.066092 547491147792 retrain.py:1082] 2020-06-22 09:15:13.065738: Step 499: Train accuracy = 100.0%

INFO:tensorflow:2020-06-22 09:15:13.066813: Step 499: Cross entropy = 0.000006

I0622 09:15:13.066950 547491147792 retrain.py:1084] 2020-06-22 09:15:13.066813: Step 499: Cross entropy = 0.000006

INFO:tensorflow:2020-06-22 09:15:13.238729: Step 499: **Validation accuracy = 100.0% (N=100)**

I0622 09:15:13.239103 547491147792 retrain.py:1100] 2020-06-22 09:15:13.238729: Step 499: Validation accuracy = 100.0% (N=100)

INFO:tensorflow:Final test accuracy = 100.0% (N=3)

I0622 09:15:13.886053 547491147792 retrain.py:1126] **Final test accuracy = 100.0% (N=3)**

Seems like we have perfect test and training accuracy (we trained on a few images – 20 of each class), which may indicate overfitting. Testing this on a sample image:

python3 -m scripts.label_image --graph=tf_files/retrained_graph.pb --image=**tf_files/downloads/w251_hw5/images/dog/bulldog/14.bulldog.jpg**

Evaluation time (1-image): 11.793s

bulldog (score=1.00000)

golden retriever (score=0.00000)

python3 -m scripts.label_image --graph=tf_files/retrained_graph.pb --image=**tf_files/downloads/w251_hw5/images/dog/golden_retriever/14.IMG_8257.jpg**

Evaluation time (1-image): 8.695s

golden retriever (score=1.00000)

bulldog (score=0.00000)

Seems like the model is being able to correctly classify the images!

Run the TF1 script on the CPU (see instructions above) How does the training time compare to the default network training (section 4)? Why?

docker run -p 6006:6006 -dti w251/tensorflow:dev-tx2-4.3_b132-tf1 bash (the privileged flag is what allows the container to see the GPU resources)

```
python3 -m scripts.retrain \
  --bottleneck_dir=tf_files/bottlenecks \
  --how_many_training_steps=500 \
  --model_dir=tf_files/models/ \
  --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/flower_photos
```

[CPU] The network in this case took 3 mins approx. to train.

I0622 09:44:34.547977 547729559568 retrain.py:1082] 2020-06-22 09:44:34.547554: Step 499: Train accuracy = 98.0%

INFO:tensorflow:2020-06-22 09:44:34.548828: Step 499: Cross entropy = 0.094190

I0622 09:44:34.549009 547729559568 retrain.py:1084] 2020-06-22 09:44:34.548828: Step 499: Cross entropy = 0.094190

INFO:tensorflow:2020-06-22 09:44:34.809161: Step 499: **Validation accuracy = 92.0% (N=100)**

I0622 09:44:34.809496 547729559568 retrain.py:1100] 2020-06-22 09:44:34.809161: Step 499: Validation accuracy = 92.0% (N=100)

INFO:tensorflow:Final test accuracy = 90.1% (N=362)

I0622 09:44:35.689073 547729559568 retrain.py:1126] **Final test accuracy = 90.1% (N=362)**

With a GPU (with privileged flag), docker run --privileged -p 6006:6006 -dti w251/tensorflow:dev-tx2-4.3_b132-tf1 bash

```
Python3 -m scripts.retrain \
  --bottleneck_dir=tf_files/bottlenecks \
  --how_many_training_steps=500 \
  --model_dir=tf_files/models/ \
  --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/flower_photos
```

[GPU] The network in this case took 1 min 40 sec approx. to train.

0622 10:02:19.456245 547526041616 retrain.py:1082] 2020-06-22 10:02:19.454794: Step 499: Train accuracy = 99.0%

INFO:tensorflow:2020-06-22 10:02:19.457087: Step 499: Cross entropy = 0.075473

I0622 10:02:19.457360 547526041616 retrain.py:1084] 2020-06-22 10:02:19.457087: Step 499: Cross entropy = 0.075473

INFO:tensorflow:2020-06-22 10:02:19.633348: Step 499: **Validation accuracy = 87.0% (N=100)**

I0622 10:02:19.633801 547526041616 retrain.py:1100] 2020-06-22 10:02:19.633348: Step 499: Validation accuracy = 87.0% (N=100)

INFO:tensorflow:Final test accuracy = 89.0% (N=362)

I0622 10:02:20.663500 547526041616 retrain.py:1126] **Final test accuracy = 89.0% (N=362)**

The GPU training was much faster, almost 2x. Having said that, the validation and test accuracy in the GPU model was LOWER than the CPU model.

Try the training again, but this time do export ARCHITECTURE="inception_v3" Are CPU and GPU training times different?

Yes, the times are faster for the GPU as expected. The slowness is more visible in inception models compared to the mobilenet models because mobilenet models are generally more compute efficient than inception models.

Given the hints under the notes section, if we trained Inception_v3, what do we need to pass to replace ??? below to the label_image script? Can we also glean the answer from examining TensorBoard?

```
python -m scripts.label_image --input_layer=??? --input_height=??? --input_width=???  --
graph=tf_files/retrained_graph.pb
--image=tf_files/flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

python -m scripts.label_image --input_layer=**”Mul:0”** --input_height=**299** --input_width=**299** --
graph=tf_files/retrained_graph.pb
--image=tf_files/flower_photos/daisy/21652746_cc379e0eea_m.jpg

We place the above values because for inception v3, input image must be 299 x 299 in size, rather than mobile net where the image size varies, based on what version of mobile net is being run (1 vs 0.25 vs 0.5)

The input layer is Mul:0 as seen from here: https://github.com/googlecodelabs/tensorflow-for-poets-2/blob/bc96088a4de86729920e120111f5b208f7f1cbb1/scripts/retrain.py#L870