tensorflow / **models**

Branch: **master** | **models** / research / **slim** / | Create new file | Find file | History

cclauss from six.moves import xrange (en masse) AGAIN — Latest commit `ffb8041` 7 days ago

..		
datasets	from six.moves import xrange (en masse) AGAIN	7 days ago
deployment	Automated g4 rollback of changelist 173688125	4 months ago
nets	from six.moves import xrange (en masse) AGAIN	7 days ago
preprocessing	Internal changes for slim (#3448)	13 days ago
scripts	Bring tensorflow/models slim up to date.	5 months ago
BUILD	Internal changes for slim (#3448)	13 days ago
README.md	Add mobilenet train and eval scripts and update documentation to incl…	14 days ago
WORKSPACE	Move the research models into a research subfolder (#2430)	6 months ago
__init__.py	Move the research models into a research subfolder (#2430)	6 months ago
download_and_convert_data.py	Move the research models into a research subfolder (#2430)	6 months ago
eval_image_classifier.py	Internal changes for slim (#3448)	13 days ago
export_inference_graph.py	Update links in slim to the new tensorflow models organization (#2439)	6 months ago
export_inference_graph_test.py	Move the research models into a research subfolder (#2430)	6 months ago
setup.py	Bring tensorflow/models slim up to date.	5 months ago
slim_walkthrough.ipynb	Fix broken links in the models repo (#2445)	6 months ago
train_image_classifier.py	Move the research models into a research subfolder (#2430)	6 months ago

README.md

TensorFlow-Slim image classification model library

TF-slim is a new lightweight high-level API of TensorFlow (`tensorflow.contrib.slim`) for defining, training and evaluating complex models. This directory contains code for training and evaluating several widely used Convolutional Neural Network (CNN) image classification models using TF-slim. It contains scripts that will allow you to train models from scratch or fine-tune them from pre-trained network weights. It also contains code for downloading standard image datasets, converting them to TensorFlow's native TFRecord format and reading them in using TF-Slim's data reading and queueing utilities. You can easily train any model on any of these datasets, as we demonstrate below. We've also included a jupyter notebook, which provides working examples of how to use TF-Slim for image classification. For developing or modifying your own models, see also the main TF-Slim page.

Contacts

Maintainers of TF-slim:

- Nathan Silberman, github: nathansilberman
- Sergio Guadarrama, github: sguada

Table of contents

Installation

In this section, we describe the steps required to install the appropriate prerequisite packages.

Installing latest version of TF-slim

TF-Slim is available as `tf.contrib.slim` via TensorFlow 1.0. To test that your installation is working, execute the following command; it should run without raising any errors.

```
python -c "import tensorflow.contrib.slim as slim; eval = slim.evaluation.evaluate_once"
```

Installing the TF-slim image models library

To use TF-Slim for image classification, you also have to install the TF-Slim image models library, which is not part of the core TF library. To do this, check out the tensorflow/models repository as follows:

```
cd $HOME/workspace
git clone https://github.com/tensorflow/models/
```

This will put the TF-Slim image models library in `$HOME/workspace/models/research/slim`. (It will also create a directory called models/inception, which contains an older version of slim; you can safely ignore this.)

To verify that this has worked, execute the following commands; it should run without raising any errors.

```
cd $HOME/workspace/models/research/slim
python -c "from nets import cifarnet; mynet = cifarnet.cifarnet"
```

Preparing the datasets

As part of this library, we've included scripts to download several popular image datasets (listed below) and convert them to slim format.

Dataset	Training Set Size	Testing Set Size	Number of Classes	Comments
Flowers	2500	2500	5	Various sizes (source: Flickr)
Cifar10	60k	10k	10	32x32 color
MNIST	60k	10k	10	28x28 gray
ImageNet	1.2M	50k	1000	Various sizes

Downloading and converting to TFRecord format

For each dataset, we'll need to download the raw data and convert it to TensorFlow's native TFRecord format. Each TFRecord contains a TF-Example protocol buffer. Below we demonstrate how to do this for the Flowers dataset.

```
$ DATA_DIR=/tmp/data/flowers
$ python download_and_convert_data.py \
    --dataset_name=flowers \
    --dataset_dir="${DATA_DIR}"
```

When the script finishes you will find several TFRecord files created:

```
$ ls ${DATA_DIR}
flowers_train-00000-of-00005.tfrecord
...
flowers_train-00004-of-00005.tfrecord
flowers_validation-00000-of-00005.tfrecord
...
flowers_validation-00004-of-00005.tfrecord
labels.txt
```

These represent the training and validation data, sharded over 5 files each. You will also find the `$DATA_DIR/labels.txt` file which contains the mapping from integer labels to class names.

You can use the same script to create the mnist and cifar10 datasets. However, for ImageNet, you have to follow the instructions here. Note that you first have to sign up for an account at image-net.org. Also, the download can take several hours, and could use up to 500GB.

Creating a TF-Slim Dataset Descriptor.

Once the TFRecord files have been created, you can easily define a Slim Dataset, which stores pointers to the data file, as well as various other pieces of metadata, such as the class labels, the train/test split, and how to parse the TFExample protos. We have included the TF-Slim Dataset descriptors for Cifar10, ImageNet, Flowers, and MNIST. An example of how to load data using a TF-Slim dataset descriptor using a TF-Slim DatasetDataProvider is found below:

```
import tensorflow as tf
from datasets import flowers

slim = tf.contrib.slim

# Selects the 'validation' dataset.
dataset = flowers.get_split('validation', DATA_DIR)

# Creates a TF-Slim DataProvider which reads the dataset in the background
# during both training and testing.
provider = slim.dataset_data_provider.DatasetDataProvider(dataset)
[image, label] = provider.get(['image', 'label'])
```

An automated script for processing ImageNet data.

Training a model with the ImageNet dataset is a common request. To facilitate working with the ImageNet dataset, we provide an automated script for downloading and processing the ImageNet dataset into the native TFRecord format.

The TFRecord format consists of a set of sharded files where each entry is a serialized `tf.Example` proto. Each `tf.Example` proto contains the ImageNet image (JPEG encoded) as well as metadata such as label and bounding box information.

We provide a single script for downloading and converting ImageNet data to TFRecord format. Downloading and preprocessing the data may take several hours (up to half a day) depending on your network and computer speed. Please be patient.

To begin, you will need to sign up for an account with [ImageNet] (http://image-net.org) to gain access to the data. Look for the sign up page, create an account and request an access key to download the data.

After you have `USERNAME` and `PASSWORD`, you are ready to run our script. Make sure that your hard disk has at least 500 GB of free space for downloading and storing the data. Here we select `DATA_DIR=$HOME/imagenet-data` as such a location but feel free to edit accordingly.

When you run the below script, please enter *USERNAME* and *PASSWORD* when prompted. This will occur at the very beginning. Once these values are entered, you will not need to interact with the script again.

```
# location of where to place the ImageNet data
DATA_DIR=$HOME/imagenet-data

# build the preprocessing script.
```

```
bazel build slim/download_and_preprocess_imagenet

# run it
bazel-bin/slim/download_and_preprocess_imagenet "${DATA_DIR}"
```

The final line of the output script should read:

```
2016-02-17 14:30:17.287989: Finished writing all 1281167 images in data set.
```

When the script finishes you will find 1024 and 128 training and validation files in the `DATA_DIR`. The files will match the patterns `train-????-of-1024` and `validation-?????-of-00128`, respectively.

Congratulations! You are now ready to train or evaluate with the ImageNet data set.

Pre-trained Models

Neural nets work best when they have many parameters, making them powerful function approximators. However, this means they must be trained on very large datasets. Because training models from scratch can be a very computationally intensive process requiring days or even weeks, we provide various pre-trained models, as listed below. These CNNs have been trained on the ILSVRC-2012-CLS image classification dataset.

In the table below, we list each model, the corresponding TensorFlow model file, the link to the model checkpoint, and the top 1 and top 5 accuracy (on the imagenet test set). Note that the VGG and ResNet V1 parameters have been converted from their original caffe formats (here and here), whereas the Inception and ResNet V2 parameters have been trained internally at Google. Also be aware that these accuracies were computed by evaluating using a single image crop. Some academic papers report higher accuracy by using multiple crops at multiple scales.

Model	TF-Slim File	Checkpoint	Top-1 Accuracy	Top-5 Accuracy
Inception V1	Code	inception_v1_2016_08_28.tar.gz	69.8	89.6
Inception V2	Code	inception_v2_2016_08_28.tar.gz	73.9	91.8
Inception V3	Code	inception_v3_2016_08_28.tar.gz	78.0	93.9
Inception V4	Code	inception_v4_2016_09_09.tar.gz	80.2	95.2
Inception-ResNet-v2	Code	inception_resnet_v2_2016_08_30.tar.gz	80.4	95.3
ResNet V1 50	Code	resnet_v1_50_2016_08_28.tar.gz	75.2	92.2
ResNet V1 101	Code	resnet_v1_101_2016_08_28.tar.gz	76.4	92.9
ResNet V1 152	Code	resnet_v1_152_2016_08_28.tar.gz	76.8	93.2
ResNet V2 50^	Code	resnet_v2_50_2017_04_14.tar.gz	75.6	92.8
ResNet V2 101^	Code	resnet_v2_101_2017_04_14.tar.gz	77.0	93.7
ResNet V2 152^	Code	resnet_v2_152_2017_04_14.tar.gz	77.8	94.1
ResNet V2 200	Code	TBA	79.9*	95.2*
VGG 16	Code	vgg_16_2016_08_28.tar.gz	71.5	89.8
VGG 19	Code	vgg_19_2016_08_28.tar.gz	71.1	89.8
MobileNet_v1_1.0_224	Code	mobilenet_v1_1.0_224.tgz	70.9	89.9
MobileNet_v1_0.50_160	Code	mobilenet_v1_0.50_160.tgz	59.1	81.9
MobileNet_v1_0.25_128	Code	mobilenet_v1_0.25_128.tgz	41.5	66.3
NASNet-A_Mobile_224#	Code	nasnet-a_mobile_04_10_2017.tar.gz	74.0	91.6
NASNet-A_Large_331#	Code	nasnet-a_large_04_10_2017.tar.gz	82.7	96.2

^ ResNet V2 models use Inception pre-processing and input image size of 299 (use `--preprocessing_name inception --eval_image_size 299` when using `eval_image_classifier.py`). Performance numbers for ResNet V2 models are reported on the ImageNet validation set.

(#) More information and details about the NASNet architectures are available at this README

All 16 float MobileNet V1 models reported in the MobileNet Paper and all 16 quantized TensorFlow Lite compatible MobileNet V1 models can be found here.

(*): Results quoted from the paper.

Here is an example of how to download the Inception V3 checkpoint:

```
$ CHECKPOINT_DIR=/tmp/checkpoints
$ mkdir ${CHECKPOINT_DIR}
$ wget http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz
$ tar -xvf inception_v3_2016_08_28.tar.gz
$ mv inception_v3.ckpt ${CHECKPOINT_DIR}
$ rm inception_v3_2016_08_28.tar.gz
```

Training a model from scratch.

We provide an easy way to train a model from scratch using any TF-Slim dataset. The following example demonstrates how to train Inception V3 using the default parameters on the ImageNet dataset.

```
DATASET_DIR=/tmp/imagenet
TRAIN_DIR=/tmp/train_logs
python train_image_classifier.py \
    --train_dir=${TRAIN_DIR} \
    --dataset_name=imagenet \
    --dataset_split_name=train \
    --dataset_dir=${DATASET_DIR} \
    --model_name=inception_v3
```

This process may take several days, depending on your hardware setup. For convenience, we provide a way to train a model on multiple GPUs, and/or multiple CPUs, either synchrononously or asynchronously. See model_deploy for details.

TensorBoard

To visualize the losses and other metrics during training, you can use TensorBoard by running the command below.

```
tensorboard --logdir=${TRAIN_DIR}
```

Once TensorBoard is running, navigate your web browser to http://localhost:6006.

Fine-tuning a model from an existing checkpoint

Rather than training from scratch, we'll often want to start from a pre-trained model and fine-tune it. To indicate a checkpoint from which to fine-tune, we'll call training with the `--checkpoint_path` flag and assign it an absolute path to a checkpoint file.

When fine-tuning a model, we need to be careful about restoring checkpoint weights. In particular, when we fine-tune a model on a new task with a different number of output labels, we wont be able restore the final logits (classifier) layer. For this, we'll use the `--checkpoint_exclude_scopes` flag. This flag hinders certain variables from being loaded. When fine-tuning on a classification task using a different number of classes than the trained model, the new model will have a final 'logits' layer whose dimensions differ from the pre-trained model. For example, if fine-tuning an ImageNet-trained model on Flowers, the pre-trained logits layer will have dimensions `[2048 x 1001]` but our new logits layer will have dimensions `[2048 x 5]`. Consequently, this flag indicates to TF-Slim to avoid loading these weights from the checkpoint.

Keep in mind that warm-starting from a checkpoint affects the model's weights only during the initialization of the model. Once a model has started training, a new checkpoint will be created in `${TRAIN_DIR}`. If the fine-tuning training is stopped and restarted, this new checkpoint will be the one from which weights are restored and not the `${checkpoint_path}$`. Consequently, the flags `--checkpoint_path` and `--checkpoint_exclude_scopes` are only used during the `0-` th global step (model initialization). Typically for fine-tuning one only want train a sub-set of layers, so the flag `--trainable_scopes` allows to specify which subsets of layers should trained, the rest would remain frozen.

Below we give an example of fine-tuning inception-v3 on flowers, inception_v3 was trained on ImageNet with 1000 class labels, but the flowers dataset only have 5 classes. Since the dataset is quite small we will only train the new layers.

```
$ DATASET_DIR=/tmp/flowers
$ TRAIN_DIR=/tmp/flowers-models/inception_v3
$ CHECKPOINT_PATH=/tmp/my_checkpoints/inception_v3.ckpt
$ python train_image_classifier.py \
    --train_dir=${TRAIN_DIR} \
    --dataset_dir=${DATASET_DIR} \
    --dataset_name=flowers \
    --dataset_split_name=train \
    --model_name=inception_v3 \
    --checkpoint_path=${CHECKPOINT_PATH} \
    --checkpoint_exclude_scopes=InceptionV3/Logits,InceptionV3/AuxLogits \
    --trainable_scopes=InceptionV3/Logits,InceptionV3/AuxLogits
```

Evaluating performance of a model

To evaluate the performance of a model (whether pretrained or your own), you can use the eval_image_classifier.py script, as shown below.

Below we give an example of downloading the pretrained inception model and evaluating it on the imagenet dataset.

```
CHECKPOINT_FILE = ${CHECKPOINT_DIR}/inception_v3.ckpt  # Example
$ python eval_image_classifier.py \
    --alsologtostderr \
    --checkpoint_path=${CHECKPOINT_FILE} \
    --dataset_dir=${DATASET_DIR} \
    --dataset_name=imagenet \
    --dataset_split_name=validation \
    --model_name=inception_v3
```

See the evaluation module example for an example of how to evaluate a model at multiple checkpoints during or after the training.

Exporting the Inference Graph

Saves out a GraphDef containing the architecture of the model.

To use it with a model name defined by slim, run:

```
$ python export_inference_graph.py \
  --alsologtostderr \
  --model_name=inception_v3 \
  --output_file=/tmp/inception_v3_inf_graph.pb

$ python export_inference_graph.py \
  --alsologtostderr \
  --model_name=mobilenet_v1 \
  --image_size=224 \
  --output_file=/tmp/mobilenet_v1_224.pb
```

Freezing the exported Graph

If you then want to use the resulting model with your own or pretrained checkpoints as part of a mobile model, you can run freeze_graph to get a graph def with the variables inlined as constants using:

```
bazel build tensorflow/python/tools:freeze_graph

bazel-bin/tensorflow/python/tools/freeze_graph \
  --input_graph=/tmp/inception_v3_inf_graph.pb \
  --input_checkpoint=/tmp/checkpoints/inception_v3.ckpt \
  --input_binary=true --output_graph=/tmp/frozen_inception_v3.pb \
  --output_node_names=InceptionV3/Predictions/Reshape_1
```

The output node names will vary depending on the model, but you can inspect and estimate them using the summarize_graph tool:

```
bazel build tensorflow/tools/graph_transforms:summarize_graph

bazel-bin/tensorflow/tools/graph_transforms/summarize_graph \
  --in_graph=/tmp/inception_v3_inf_graph.pb
```

Run label image in C++

To run the resulting graph in C++, you can look at the label_image sample code:

```
bazel build tensorflow/examples/label_image:label_image

bazel-bin/tensorflow/examples/label_image/label_image \
  --image=${HOME}/Pictures/flowers.jpg \
  --input_layer=input \
  --output_layer=InceptionV3/Predictions/Reshape_1 \
  --graph=/tmp/frozen_inception_v3.pb \
  --labels=/tmp/imagenet_slim_labels.txt \
  --input_mean=0 \
  --input_std=255
```

Troubleshooting

The model runs out of CPU memory.

See Model Runs out of CPU memory.

The model runs out of GPU memory.

See Adjusting Memory Demands.

The model training results in NaN's.

See Model Resulting in NaNs.

The ResNet and VGG Models have 1000 classes but the ImageNet dataset has 1001

The ImageNet dataset provided has an empty background class which can be used to fine-tune the model to other tasks. If you try training or fine-tuning the VGG or ResNet models using the ImageNet dataset, you might encounter the following error:

```
InvalidArgumentError: Assign requires shapes of both tensors to match. lhs shape= [1001] rhs shape= [1000]
```

This is due to the fact that the VGG and ResNet V1 final layers have only 1000 outputs rather than 1001.

To fix this issue, you can set the `--labels_offset=1` flag. This results in the ImageNet labels being shifted down by one:

I wish to train a model with a different image size.

The preprocessing functions all take `height` and `width` as parameters. You can change the default values using the following snippet:

```
image_preprocessing_fn = preprocessing_factory.get_preprocessing(
    preprocessing_name,
    height=MY_NEW_HEIGHT,
    width=MY_NEW_WIDTH,
    is_training=True)
```

What hardware specification are these hyper-parameters targeted for?

See Hardware Specifications.