Caffe

Deep learning framework by the BVLC

Created by
Yangqing Jia
Lead Developer
Evan Shelhamer

View On GitHub

Brewing ImageNet

This guide is meant to get you ready to train your own model on your own data. If you just want an ImageNet-trained network, then note that since training takes a lot of energy and we hate global warming, we provide the CaffeNet model trained as described below in the model zoo.

Data Preparation

The guide specifies all paths and assumes all commands are executed from the root caffe directory.

By "ImageNet" we here mean the ILSVRC12 challenge, but you can easily train on the whole of ImageNet as well, just with more disk space, and a little longer training time.

We assume that you already have downloaded the ImageNet training data and validation data, and they are stored on your disk like:

```
/path/to/imagenet/train/n01440764/n01440764_10026.JPEG
/path/to/imagenet/val/ILSVRC2012_val_00000001.JPEG
```

You will first need to prepare some auxiliary data for training. This data can be downloaded by:

```
./data/ilsvrc12/get_ilsvrc_aux.sh
```

The training and validation input are described in `train.txt` and `val.txt` as text listing all the files and their labels. Note that we use a different indexing for labels than the ILSVRC devkit: we sort the synset names in their ASCII order, and then label them from 0 to 999. See `synset_words.txt` for the synset/name mapping.

You may want to resize the images to 256x256 in advance. By default, we do not explicitly do this because in a cluster environment, one may benefit from resizing images in a parallel fashion, using mapreduce. For example, Yangqing used his lightweight mincepie package. If you prefer things to be simpler, you can also use shell commands, something like:

```
for name in /path/to/imagenet/val/*.JPEG; do
    convert -resize 256x256\! $name $name
done
```

Take a look at `examples/imagenet/create_imagenet.sh`. Set the paths to the train and val dirs as needed, and set "RESIZE=true" to resize all images to 256x256 if you haven't resized the images in advance. Now simply create the leveldbs with `examples/imagenet/create_imagenet.sh`. Note that `examples/imagenet/ilsvrc12_train_leveldb` and `examples/imagenet/ilsvrc12_val_leveldb` should not exist before this execution. It will be created by the script. `GLOG_logtostderr=1` simply dumps more information for you to inspect, and you can safely ignore it.

Compute Image Mean

The model requires us to subtract the image mean from each image, so we have to compute the mean. `tools/compute_image_mean.cpp` implements that - it is also a good example to familiarize yourself on how to manipulate the multiple components, such as protocol buffers, leveldbs, and logging, if you are not familiar with them. Anyway, the mean computation can be carried out as:

```
./examples/imagenet/make_imagenet_mean.sh
```

which will make `data/ilsvrc12/imagenet_mean.binaryproto`.

Model Definition

We are going to describe a reference implementation for the approach first proposed by Krizhevsky, Sutskever, and Hinton in their NIPS 2012 paper.

The network definition (`models/bvlc_reference_caffenet/train_val.prototxt`) follows the one in Krizhevsky et al. Note that if you deviated from file paths suggested in this guide, you'll need to adjust the relevant paths in the `.prototxt` files.

If you look carefully at `models/bvlc_reference_caffenet/train_val.prototxt`, you will notice several `include` sections specifying either `phase: TRAIN` or `phase: TEST`. These sections allow us to define two closely related networks in one file: the network used for training and the network used for testing. These two networks are almost identical, sharing all layers except for those marked with `include { phase: TRAIN }` or `include { phase: TEST }`. In this case, only the input layers and one output layer are different.

Input layer differences: The training network's `data` input layer draws its data from `examples/imagenet/ilsvrc12_train_leveldb` and randomly mirrors the input image. The testing network's `data` layer takes data from `examples/imagenet/ilsvrc12_val_leveldb` and does not perform random mirroring.

Output layer differences: Both networks output the `softmax_loss` layer, which in training is used to compute the loss function and to initialize the backpropagation, while in validation this loss is simply reported. The testing network also has a second output layer, `accuracy`, which is used to report the accuracy on the test set. In the process of training, the test network will occasionally be instantiated and tested on the test set, producing lines like `Test score #0: xxx` and `Test score #1: xxx`. In this case score 0 is the accuracy (which will start around 1/1000 = 0.001 for an untrained network) and score 1 is the loss (which will start around 7 for an untrained network).

We will also lay out a protocol buffer for running the solver. Let's make a few plans:

- We will run in batches of 256, and run a total of 450,000 iterations (about 90 epochs).
- For every 1,000 iterations, we test the learned net on the validation data.
- We set the initial learning rate to 0.01, and decrease it every 100,000 iterations (about 20 epochs).
- Information will be displayed every 20 iterations.
- The network will be trained with momentum 0.9 and a weight decay of 0.0005.

- For every 10,000 iterations, we will take a snapshot of the current status.

Sound good? This is implemented in `models/bvlc_reference_caffenet/solver.prototxt`.

Training ImageNet

Ready? Let's train.

```
./build/tools/caffe train --solver=models/bvlc_reference_caffenet/solver.prototxt
```

Sit back and enjoy!

On a K40 machine, every 20 iterations take about 26.5 seconds to run (while a on a K20 this takes 36 seconds), so effectively about 5.2 ms per image for the full forward-backward pass. About 2 ms of this is on forward, and the rest is backward. If you are interested in dissecting the computation time, you can run

```
./build/tools/caffe time --model=models/bvlc_reference_caffenet/train_val.prototxt
```

Resume Training?

We all experience times when the power goes out, or we feel like rewarding ourself a little by playing Battlefield (does anyone still remember Quake?). Since we are snapshotting intermediate results during training, we will be able to resume from snapshots. This can be done as easy as:

```
./build/tools/caffe train --solver=models/bvlc_reference_caffenet/solver.prototxt --
snapshot=models/bvlc_reference_caffenet/caffenet_train_iter_10000.solverstate
```

where in the script `caffenet_train_iter_10000.solverstate` is the solver state snapshot that stores all necessary information to recover the exact solver state (including the parameters, momentum history, etc).

Parting Words

Hope you liked this recipe! Many researchers have gone further since the ILSVRC 2012 challenge, changing the network architecture and/or fine-tuning the various parameters in the network to address new data and tasks. **Caffe lets you explore different network choices more easily by**

simply writing different prototxt files - isn't that exciting?

And since now you have a trained network, check out how to use it with the Python interface for classifying ImageNet.