Caffe

Deep learning framework by the BVLC

Created by
Yangqing Jia
Lead Developer
Evan Shelhamer

View On GitHub

Interfaces

Caffe has command line, Python, and MATLAB interfaces for day-to-day usage, interfacing with research code, and rapid prototyping. While Caffe is a C++ library at heart and it exposes a modular interface for development, not every occasion calls for custom compilation. The cmdcaffe, pycaffe, and matcaffe interfaces are here for you.

Command Line

The command line interface – cmdcaffe – is the `caffe` tool for model training, scoring, and diagnostics. Run `caffe` without any arguments for help. This tool and others are found in caffe/build/tools. (The following example calls require completing the LeNet / MNIST example first.)

Training: `caffe train` learns models from scratch, resumes learning from saved snapshots, and fine-tunes models to new data and tasks:

- All training requires a solver configuration through the `-solver solver.prototxt` argument.
- Resuming requires the `-snapshot model_iter_1000.solverstate` argument to load the solver snapshot.

- Fine-tuning requires the `-weights model.caffemodel` argument for the model initialization.

For example, you can run:

```
# train LeNet
caffe train -solver examples/mnist/lenet_solver.prototxt
# train on GPU 2
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 2
# resume training from the half-way point snapshot
caffe train -solver examples/mnist/lenet_solver.prototxt -snapshot
examples/mnist/lenet_iter_5000.solverstate
```

For a full example of fine-tuning, see examples/finetuning_on_flickr_style, but the training call alone is

```
# fine-tune CaffeNet model weights for style recognition
caffe train -solver examples/finetuning_on_flickr_style/solver.prototxt
-weights models/bvlc_reference_caffenet
/bvlc_reference_caffenet.caffemodel
```

Testing: `caffe test` scores models by running them in the test phase and reports the net output as its score. The net architecture must be properly defined to output an accuracy measure or loss as its output. The per-batch score is reported and then the grand average is reported last.

```
# score the learned LeNet model on the validation set as defined in the
# model architeture lenet_train_test.prototxt
caffe test -model examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 100
```

Benchmarking: `caffe time` benchmarks model execution layer-by-layer through timing and synchronization. This is useful to check system performance and measure relative execution times for models.

```
# (These example calls require you complete the LeNet / MNIST example
first.)
# time LeNet training on CPU for 10 iterations
caffe time -model examples/mnist/lenet_train_test.prototxt -iterations
10
# time LeNet training on GPU for the default 50 iterations
caffe time -model examples/mnist/lenet_train_test.prototxt -gpu 0
# time a model architecture with the given weights on the first GPU for
10 iterations
caffe time -model examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 10
```

Diagnostics: `caffe device_query` reports GPU details for reference and checking device ordinals for running on a given device in multi-GPU machines.

```
# query the first device
caffe device_query -gpu 0
```

Parallelism: the `-gpu` flag to the `caffe` tool can take a comma separated list of IDs to run on multiple GPUs. A solver and net will be instantiated for each GPU so the batch size is effectively multiplied by the number of GPUs. To reproduce single GPU training, reduce the batch size in the network definition accordingly.

```
# train on GPUs 0 & 1 (doubling the batch size)
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 0,1
# train on all GPUs (multiplying batch size by number of devices)
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu all
```

Python

The Python interface – pycaffe – is the `caffe` module and its scripts in caffe/python. `import caffe` to load models, do forward and backward, handle IO, visualize networks, and even instrument model solving. All model data, derivatives, and parameters are exposed for reading and writing.

- `caffe.Net` is the central interface for loading, configuring, and running models. `caffe.Classifier` and `caffe.Detector` provide convenience interfaces for common tasks.
- `caffe.SGDSolver` exposes the solving interface.
- `caffe.io` handles input / output with preprocessing and protocol buffers.
- `caffe.draw` visualizes network architectures.
- Caffe blobs are exposed as numpy ndarrays for ease-of-use and efficiency.

Tutorial IPython notebooks are found in caffe/examples: do `ipython notebook caffe/examples` to try them. For developer reference docstrings can be found throughout the code.

Compile pycaffe by `make pycaffe`. Add the module directory to your `$PYTHONPATH` by `export PYTHONPATH=/path/to/caffe/python:$PYTHONPATH` or the like for `import caffe`.

MATLAB

The MATLAB interface – matcaffe – is the `caffe` package in caffe/matlab in which you can integrate Caffe in your Matlab code.

In MatCaffe, you can

- Creating multiple Nets in Matlab
- Do forward and backward computation
- Access any layer within a network, and any parameter blob in a layer
- Get and set data or diff to any blob within a network, not restricting to input blobs or output blobs
- Save a network's parameters to file, and load parameters from file
- Reshape a blob and reshape a network
- Edit network parameter and do network surgery
- Create multiple Solvers in Matlab for training
- Resume training from solver snapshots
- Access train net and test nets in a solver
- Run for a certain number of iterations and give back control to Matlab
- Intermingle arbitrary Matlab code with gradient steps

An ILSVRC image classification demo is in caffe/matlab /demo/classification_demo.m (you need to download BVLC CaffeNet from Model Zoo to run it).

Build MatCaffe

Build MatCaffe with `make all matcaffe`. After that, you may test it using `make mattest`.

Common issue: if you run into error messages like `libstdc++.so.6:version 'GLIBCXX_3.4.15' not found` during `make mattest`, then it usually means that your Matlab's runtime libraries do not match your compile-time libraries. You may need to do the following before you start Matlab:

```
export LD_LIBRARY_PATH=/opt/intel/mkl/lib/intel64:/usr/local/cuda/lib64
export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6
```

Or the equivalent based on where things are installed on your system, and do `make mattest` again to see if the issue is fixed. Note: this issue is sometimes more complicated since during its startup Matlab may overwrite your `LD_LIBRARY_PATH` environment variable. You can run `!ldd ./matlab/+caffe/private/caffe_.mexa64` (the mex extension may differ on your system) in Matlab to see its runtime libraries, and preload your compile-time libraries by exporting them to your `LD_PRELOAD` environment variable.

After successful building and testing, add this package to Matlab search PATH by starting `matlab` from caffe root folder and running the following commands in Matlab command window.

```
addpath ./matlab
```

You can save your Matlab search PATH by running `savepath` so that you don’t have to run the command above again every time you use MatCaffe.

Use MatCaffe

MatCaffe is very similar to PyCaffe in usage.

Examples below shows detailed usages and assumes you have downloaded BVLC CaffeNet from Model Zoo and started `matlab` from caffe root folder.

```
model = './models/bvlc_reference_caffenet/deploy.prototxt';
weights = './models/bvlc_reference_caffenet
/bvlc_reference_caffenet.caffemodel';
```

Set mode and device

Mode and device should always be set BEFORE you create a net or a solver.

Use CPU:

```
caffe.set_mode_cpu();
```

Use GPU and specify its gpu_id:

```
caffe.set_mode_gpu();
caffe.set_device(gpu_id);
```

Create a network and access its layers and blobs

Create a network:

```
net = caffe.Net(model, weights, 'test'); % create net and load weights
```

Or

```
net = caffe.Net(model, 'test'); % create net but not load weights
net.copy_from(weights); % load weights
```

which creates `net` object as

```
  Net with properties:

           layer_vec: [1x23 caffe.Layer]
            blob_vec: [1x15 caffe.Blob]
              inputs: {'data'}
             outputs: {'prob'}
    name2layer_index: [23x1 containers.Map]
     name2blob_index: [15x1 containers.Map]
         layer_names: {23x1 cell}
          blob_names: {15x1 cell}
```

The two `containers.Map` objects are useful to find the index of a layer or a blob by its name.

You have access to every blob in this network. To fill blob ‘data’ with all ones:

```
net.blobs('data').set_data(ones(net.blobs('data').shape));
```

To multiply all values in blob ‘data’ by 10:

```
net.blobs('data').set_data(net.blobs('data').get_data() * 10);
```

Be aware that since Matlab is 1-indexed and column-major, the usual 4 blob dimensions in Matlab are `[width, height, channels, num]`, and `width` is the fastest dimension. Also be aware that images are in BGR channels. Also, Caffe uses single-precision float data. If your data is not single, `set_data` will automatically convert it to single.

You also have access to every layer, so you can do network surgery. For example, to multiply conv1 parameters by 10:

```
net.params('conv1', 1).set_data(net.params('conv1', 1).get_data() *
10); % set weights
net.params('conv1', 2).set_data(net.params('conv1', 2).get_data() *
10); % set bias
```

Alternatively, you can use

```
net.layers('conv1').params(1).set_data(net.layers('conv1').params(1).get
_data() * 10);
net.layers('conv1').params(2).set_data(net.layers('conv1').params(2).get
_data() * 10);
```

To save the network you just modified:

```
net.save('my_net.caffemodel');
```

To get a layer's type (string):

```
layer_type = net.layers('conv1').type;
```

Forward and backward

Forward pass can be done using `net.forward` or `net.forward_prefilled`. Function `net.forward` takes in a cell array of N-D arrays containing data of input blob(s) and outputs a cell array containing data from output blob(s). Function `net.forward_prefilled` uses existing data in input blob(s) during forward pass, takes no input and produces no output. After creating some data for input blobs like `data = rand(net.blobs('data').shape);` you can run

```
res = net.forward({data});
prob = res{1};
```

Or

```
net.blobs('data').set_data(data);
net.forward_prefilled();
prob = net.blobs('prob').get_data();
```

Backward is similar using `net.backward` or `net.backward_prefilled` and replacing `get_data` and `set_data` with `get_diff` and `set_diff`. After creating some gradients for output blobs like `prob_diff = rand(net.blobs('prob').shape);` you can run

```
res = net.backward({prob_diff});
```

```
data_diff = res{1};
```

Or

```
net.blobs('prob').set_diff(prob_diff);
net.backward_prefilled();
data_diff = net.blobs('data').get_diff();
```

However, the backward computation above doesn't get correct results, because Caffe decides that the network does not need backward computation. To get correct backward results, you need to set `'force_backward: true'` in your network prototxt.

After performing forward or backward pass, you can also get the data or diff in internal blobs. For example, to extract pool5 features after forward pass:

```
pool5_feat = net.blobs('pool5').get_data();
```

Reshape

Assume you want to run 1 image at a time instead of 10:

```
net.blobs('data').reshape([227 227 3 1]); % reshape blob 'data'
net.reshape();
```

Then the whole network is reshaped, and now `net.blobs('prob').shape` should be `[1000 1]`;

Training

Assume you have created training and validation lmdbs following our ImageNET Tutorial, to create a solver and train on ILSVRC 2012 classification dataset:

```
solver = caffe.Solver('./models/bvlc_reference_caffenet
/solver.prototxt');
```

which creates `solver` object as

```
  Solver with properties:

          net: [1x1 caffe.Net]
```

```
    test_nets: [1x1 caffe.Net]
```

To train:

```
solver.solve();
```

Or train for only 1000 iterations (so that you can do something to its net before training more iterations)

```
solver.step(1000);
```

To get iteration number:

```
iter = solver.iter();
```

To get its network:

```
train_net = solver.net;
test_net = solver.test_nets(1);
```

To resume from a snapshot "your_snapshot.solverstate":

```
solver.restore('your_snapshot.solverstate');
```

Input and output

`caffe.io` class provides basic input functions `load_image` and `read_mean`. For example, to read ILSVRC 2012 mean file (assume you have downloaded imagenet example auxiliary files by running `./data/ilsvrc12/get_ilsvrc_aux.sh`):

```
mean_data = caffe.io.read_mean('./data/ilsvrc12
/imagenet_mean.binaryproto');
```

To read Caffe's example image and resize to `[width, height]` and suppose we want `width = 256; height = 256;`

```
im_data = caffe.io.load_image('./examples/images/cat.jpg');
im_data = imresize(im_data, [width, height]); % resize using Matlab's
imresize
```

Keep in mind that `width` is the fastest dimension and channels are BGR,

which is different from the usual way that Matlab stores an image. If you don't want to use `caffe.io.load_image` and prefer to load an image by yourself, you can do

```
im_data = imread('./examples/images/cat.jpg'); % read image
im_data = im_data(:, :, [3, 2, 1]); % convert from RGB to BGR
im_data = permute(im_data, [2, 1, 3]); % permute width and height
im_data = single(im_data); % convert to single precision
```

Also, you may take a look at caffe/matlab/demo/classification_demo.m to see how to prepare input by taking crops from an image.

We show in caffe/matlab/hdf5creation how to read and write HDF5 data with Matlab. We do not provide extra functions for data output as Matlab itself is already quite powerful in output.

Clear nets and solvers

Call `caffe.reset_all()` to clear all solvers and stand-alone nets you have created.