2016/11/16, Wensday, Cloudy

by Jincheng Su @ Hikvision, email: jcsu14@fudan.edu.cn

Problem left yesterday

Pro1. Caffe installation not succeeded. It may be the problem of not installing cuDNN successfully.

Pro2. `/path/to/bcnn/run_experiments_bcnn_train.m` still won't work.'

Going to put aside the first problem for some days, solve the `pro2` firstly with full efforts.

Keep on analysing the source code

Overview of `run_experiments_bcnn_train.m` :

```
             Entry :`run_experiments_bcnn_train()`
               |
     Set Pretrained CNN(s): `bcnnmm`, `bcnnvdm`, or `bcnnvdvd` or all three
               |
Set Model Parameters and dataset: `[opts, imdb] = model_setup(/*parameter pairs*/)
               |
 Begin to train a bcnn model: `imdb_bcnn_train_dag(imdb, opts)`
               |
              End
```

Overview of `model_setup(...)` :

```
            Entry:  `[opts, imdb] = model_setup(varargin)`
              |
    Setup data structure `opts`: `batchSize`, `numEpochs`, `learningRate`, ...
              |
Setup data structure `opts.encoders`: what the hell is encoders?
              |
        Load dataset: `cub`, `cars`, or `aircraft`, or others
              |
             End: save imdb
```

Overview of `imdb_bcnn_train_dag(imdb, opts)`

```
               Entry: `imdb_bcnn_train_dag(imdb, opts)`
                 |
      Setup some training params: `opts.train....`
                 |
    Build up and Initialize network:
        `initializeNetworkSharedWeights` or ` initializeNetworkTwoStreams`
                 |
Applying SGD algorithm to train the bcnn
                 |
                End: save networks
```

How to build up the whole bcnn model?

```
                                          Entry:
`net = initializeNetworkSharedWeights(imdb, encoderOpts, opts)`
                                            |
                           Load the pretrained model:
`net = load(encoderOpts.modela)`
                                            |
Truncate the Pretrained model for feature extraction:
`net.layers = net.layers(1:maxLayer)`
                                            |
                             Add batch normalization
                                            |
                           Stack bilinearpool layer:
`net.layers{end+1} = struct('type', 'bilinearpool', 'name', 'blp')`
                                            |
                           Stack sqrt normalization:
`net.layers{end+1} = struct('type', 'sqrt', 'name', 'sqrt_norm')`
                                            |
                             Stack ls normalization:
`net.layers{end+1} = struct('type', 'l2norm', 'name', 'l2_norm')`
                                            |
                       Stack a linear classifier netc:
`net.layers{end+1} = struct('type', 'conv', 'name', 'classifier', ...)`
                                            |
                    Finally, add a softmaxloss layer:
`net.layers{end+1} = struct('type', 'softmaxloss', 'name', 'loss')`
                                            |
Pretrain the linear classifier with logistic regression
```

So, how to set the pretrained model?

```
% run_experiments_bcnn_train.m

bcnnmm.name = 'bcnnmm' ;
bcnnmm.opts = {...
'type', 'bcnn', ...
'modela', 'data/models/imagenet-vgg-m.mat', ...     % intialize network A with pre-trained model
'layera', 14,...                                    % specify the output of certain layer of network A
 to be bilinearly combined
'modelb', 'data/models/imagenet-vgg-m.mat', ...     % intialize network B with pre-trained model
'layerb', 14,...                                    % specify the output of certain layer of network B
 to be bilinearly combined
'shareWeight', true,...                             % true: symmetric implementation where two network
s are identical
} ;

bcnnvdm.name = 'bcnnvdm' ;
bcnnvdm.opts = {...
'type', 'bcnn', ...
'modela', 'data/models/imagenet-vgg-verydeep-16.mat', ...
'layera', 30,...
'modelb', 'data/models/imagenet-vgg-m.mat', ...
'layerb', 14,...
'shareWeight', false,...                            % false: asymmetric implementation where two netwo
rks are distinct
} ;

bcnnvdvd.name = 'bcnnvdvd' ;
```

```
bcnnvdvd.opts = {...
'type', 'bcnn', ...
'modela', 'data/models/imagenet-vgg-verydeep-16.mat', ...
'layera', 30,...
'modelb', 'data/models/imagenet-vgg-verydeep-16.mat', ...
'layerb', 30,...
'shareWeight', true,...
};


setupNameList = {'bcnnmm'};
encoderList = {{bcnnmm}};
% setupNameList = {'bcnnvdvd'};
% encoderList = {{bcnnvdvd}};
datasetList = {{'cub', 1}}
```

Now that I have known that in order to setup a whole bcnn model, all that we need to do is to download the pretrained models `imagenet-vgg-m.mat` and `imagenet-vgg-verydeep-16.mat` . And then put the two models under `/path/to/bcnn_root/data/models` or change the path in bcnnxxx.opts to point to wherever your models are. Then setup variables `setupNameList = {'xxx'}` and `encoderList = {{xxx}}` , and `datasetList` as well. Done! That is seem to be all we need to do for building up a bcnn model.

Now that I am of quite certainty that the bcnn model should have been setup. The remained questions are

```
1. how to setup the dataset?
2. how to setup the training and testing params?
```

So **how the dataset can be set up**?

To find out the dataset setting up function, look into the function `[opts, imdb] = model_setup(varargin)` :

```
switch opts.dataset
    case 'cubcrop'
        imdb = cub_get_database(opts.cubDir, true, false);
    case 'cub'
        imdb = cub_get_database(opts.cubDir, false, opts.useVal);
    case 'aircraft-variant'
        imdb = aircraft_get_database(opts.aircraftDir, 'variant');
    case 'cars'
        imdb = cars_get_database(opts.carsDir, false, opts.useVal);
    case 'imagenet'
        imdb = cnn_imagenet_setup_data('dataDir', opts.ilsvrcDir);
    case 'imagenet-224'
        imdb = cnn_imagenet_setup_data('dataDir', opts.ilsvrcDir_224);
    otherwise
        error('Unknown dataset %s', opts.dataset) ;
end
```

To use the birds dataset `cub` , look into its function `imdb = cub_get_database(cubDir, useCropped, useVal)` :

```
% cub_get_database.m

% the directory where the real images reside
if useCropped
    imdb.imageDir = fullfile(cubDir, 'images_cropped') ;
else
    imdb.imageDir = fullfile(cubDir, 'images');
end
```

```
...

% read the class names, image names, bounding boxes, lables...

[~, classNames] = textread(fullfile(cubDir, 'classes.txt'), '%d %s');
imdb.classes.name = horzcat(classNames(:));

% Image names
[~, imageNames] = textread(fullfile(cubDir, 'images.txt'), '%d %s');
imdb.images.name = imageNames;
imdb.images.id = (1:numel(imdb.images.name));

...

% if use validation, set 1/3 to validation set
if useVal
    rng(0)
    trainSize = numel(find(imageSet==1));

    trainIdx = find(imageSet==1);

    % set 1/3 of train set to validation
    valIdx = trainIdx(randperm(trainSize, round(trainSize/3)));
    imdb.images.set(valIdx) = 2;
end

...
```

Download the CUB-200-2011 dataset and unzip and find that all the required files are contained in the package, all we need to do is put the directory under `/path/to/bcnn_root/data` with a new name `cub` .

- Dataset details:
    - Birds: CUB-200-2011 dataset. Birds + box uses bounding-boxes at training and test time.
    - Aircrafts: FGVC aircraft dataset
    - Cars: Stanford cars dataset
- These results are with domain specific fine-tuning. For more details see the updated B-CNN tech report.

With `bcnn` model and `cub` dataset set up, all that remains is to check the related params to train a `bcnn` for `CUB-200-201` dataset.

Have a look at how the `model_setup(...)` is called:

```
% run_experiments_bcnn_train.m
  [opts, imdb] = model_setup('dataset', dataset, ...
  'encoders', encoderList{ee}, ...
  'prefix', 'checkgpu', ...  % output folder name
  'batchSize', 64, ...
  'imgScale', 2, ...       % specify the scale of input images
  'bcnnLRinit', true, ...   % do logistic regression to initilize softmax layer
  'dataAugmentation', {'f2','none','none'},...      % do data augmentation [train, val, test]. Only su
pport flipping for train set on current release.
  'useGpu', [1], ...          %specify the GPU to use. 0 for using CPU
  'learningRate', 0.001, ...
  'numEpochs', 100, ...
  'momentum', 0.9, ...
  'keepAspect', true, ...
  'printDatasetInfo', true, ...
  'fromScratch', false, ...
  'rgbJitter', false, ...
  'useVal', false,...
  'numSubBatches', 1);
```

It seems that all the related params can be set inside this function call. And seems we need to do nothing with it other than change `useVal` to `true` .

Anyway, it do not harm to have a look into the `model_setup` functon.

```
% model_setup.m

function [opts, imdb] = model_setup(varargin)


% Copyright (C) 2015 Tsung-Yu Lin, Aruni RoyChowdhury, Subhransu Maji.
% All rights reserved.
%
% This file is part of the BCNN and is made available under
% the terms of the BSD license (see the COPYING file).

setup ;

opts.seed = 1 ;
opts.batchSize = 128 ;
opts.numEpochs = 100;
opts.momentum = 0.9;
opts.learningRate = 0.001;
opts.numSubBatches = 1;
opts.keepAspect = true;
opts.useVal = false;
opts.fromScratch = false;
opts.useGpu = 1 ;
opts.regionBorder = 0.05 ;
opts.numDCNNWords = 64 ;
opts.numDSIFTWords = 256 ;
opts.numSamplesPerWord = 1000 ;
opts.printDatasetInfo = false ;
opts.excludeDifficult = true ;
opts.datasetSize = inf;
% opts.encoders = {struct('name', 'rcnn', 'opts', {})} ;
opts.encoders = {} ;
opts.dataset = 'cub' ;
opts.carsDir = 'data/cars';
opts.cubDir = 'data/cub';
opts.aircraftDir = 'data/fgvc-aircraft-2013b';
opts.ilsvrcDir = '/home/tsungyulin/dataset/ILSVRC2014/CLS-LOC/';
opts.ilsvrcDir_224 = '/home/tsungyulin/dataset/ILSVRC2014/CLS-LOC-224/';
opts.suffix = 'baseline' ;
opts.prefix = 'v1' ;
opts.model  = 'imagenet-vgg-m.mat';
opts.modela = 'imagenet-vgg-m.mat';
%opts.cropSize = 227/256;   % added by jcsu

%opts.model  = 'imagenet-vgg-verydeep-16.mat';
%opts.modela = 'imagenet-vgg-verydeep-16.mat';
opts.modelb = [];
opts.layer  = 14;
opts.layera = [];
opts.layerb = [];
opts.imgScale = 1;
opts.bcnnLRinit = false;
opts.bcnnLayer = 14;
opts.rgbJitter = false;
```

```
opts.dataAugmentation = {'none', 'none', 'none'};
opts.cudnn = true;
opts.nonftbcnnDir = 'nonftbcnn';
opts.batchNormalization = false;
opts.cudnnWorkspaceLimit = 1024*1024*1204;

[opts, varargin] = vl_argparse(opts,varargin) ;

opts.expDir = sprintf('data/%s/%s-seed-%02d', opts.prefix, opts.dataset, opts.seed) ;
opts.nonftbcnnDir = fullfile(opts.expDir, opts.nonftbcnnDir);
opts.imdbDir = fullfile(opts.expDir, 'imdb') ;
opts.resultPath = fullfile(opts.expDir, sprintf('result-%s.mat', opts.suffix)) ;

opts = vl_argparse(opts,varargin) ;

if nargout <= 1, return ; end

...
```

Owh, Jesus! what the hell are all those params in `opts` !!

With a little thoughts, it seems that these are default values to the params. And we can change it via `argment pairs` in the `model_setup('name', 'value', ...)` call.

Anyway, just leave it alone.

In summary, what I have done are:

download `imagenet-vgg-m.mat` *and* `imagenet-vgg-verydeep-16.mat` *, and put it under* `/path/to/bcnn_root/data/models` download datasets `CUB-200-2011` , and put it under `/path/to/bcnn_root/data` with a new name `cub`
*check the variables `setupNameList` , `encoderList` , and `datasetList` in `run_experiments_bcnn_train.m` , to use the `bcnnmm` model and `cub` dataset.

OK, run `run_experiments_bcnn_train` !

```
 196:  196.House_Wren (train:    29, test:    30 total:    59)
 197:  197.Marsh_Wren (train:    30, test:    30 total:    60)
 198:   198.Rock_Wren (train:    30, test:    30 total:    60)
 199: 199.Winter_Wren (train:    30, test:    30 total:    60)
 200: 200.Common_Yellowthroat (train:    30, test:    30 total:    60)
 Inf:       **total** (train: 11788, test: 11788 total: 11788)
dataset: there are 11788 images (5994 trainval 5794 test)
collecting image stats: batch starting with image 2 ...Error using vl_argparse (line 101)
Unknown parameter 'cropSize'

Error in vl_argparse (line 80)
      opts = vl_argparse(opts, vertcat(params,values)) ;

Error in imdb_get_batch_bcnn (line 35)
    opts(i) = vl_argparse(opts(i), {varargin{1}(i),varargin{2:end}});

Error in getBatchSimpleNNWrapper>getBatchSimpleNN (line 10)
im = imdb_get_batch_bcnn(images, opts, ...

Error in getBatchSimpleNNWrapper>@(imdb,batch)getBatchSimpleNN(imdb,batch,opts) (line 4)
fn = @(imdb, batch) getBatchSimpleNN(imdb, batch, opts) ;

Error in imdb_bcnn_train_dag>getImageStats (line 213)
  temp = fn(imdb, batch) ;
```

```
Error in imdb_bcnn_train_dag (line 83)
  [averageImage, rgbMean, rgbCovariance] = getImageStats(imdb, bopts) ;

Error in run_experiments_bcnn_train (line 81)
          imdb_bcnn_train_dag(imdb, opts);
```

ERROR! ...

Well, set some breakpoints, and locate the error (NOTE: although MATLAB gives lots of helpful error information to help us debugging, chances are that the error location can be not that accurate. And it also helps a lot to trace back the execution stack.):

```
**function run_experiments_bcnn_train():**
    -> imdb_bcnn_train_dag(imdb, opts);
**function imdb_bcnn_train_dag(imdb, opts, varargin):**
    -> [averageImage, rgbMean, rgbCovariance] = getImageStats(imdb, bopts);
**function [averageImage, rgbMean, rgbCovariance] = getImageStats(imdb, opts):**
    -> fn = getBatchSimpleNNWrapper(opts); temp = fn(imdb, batch);
**function fn = getBatchSimpleNNWrapper(opts):**
    -> fn = @(imdb, batch) getBatchSimpleNN(imdb, batch, opts);
**function [im,labels] = getBatchSimpleNN(imdb, batch, opts):**
    -> im = imdb_get_batch_bcnn(images, opts, ...
                        'prefetch', nargout == 0);
**function imo = imdb_get_batch_bcnn(images, varargin):**
    -> opts(i) = vl_argparse(opts(i), {varargin{1}(i),varargin{2:end}});
**function [opts, args] = vl_argparse(opts, args, varargin):**
    ->  if nargout == 1
            opts = vl_argparse(opts, vertcat(params,values)) ;
**function [opts, args] = vl_argparse(opts, args, varargin):**
    ->  if nargout == 1
            error('Unknown parameter ''%s''', param) ;
```

What the hell is cropSize?

After a few tracing during debugging, I find that `cropSize` is introduced by `net.meta.normalization` (`imdb_bcnn_train_dag.m: line 71`) to `bopts` which will be passed to function call `getImageStats(imdb, bopts)` and further be passed to function `imdb_get_batch_bcmm(images, varargin)` via `bopts -> opts -> varargin`, and finally be passed to function `vl_argparse(opts, args, varargin)`.

Inspect a few more lines around `opts = vl_argparse(opts, vertcat(params,values))` in `vl_argparse.m`:

```
% ./matconvnet/matlab/vl_argparse.m

if recursive && isstruct(args{ai})
    params = fieldnames(args{ai})' ;
    values = struct2cell(args{ai})' ;
    if nargout == 1
        opts = vl_argparse(opts, vertcat(params,values)) ;
    else
          [opts, rest] = vl_argparse(opts, reshape(vertcat(params,values), 1, [])) ;
          args{ai} = cell2struct(rest(2:2:end), rest(1:2:end), 2) ;
          keep(ai) = true ;
    end
    ai = ai + 1 ;
    continue ;
 end
```

where `recursive` is set to `true`, and `args` is passed in as a pack `{varargin{1}(i),varargin{2:end}}`:

```
**function imo = imdb_get_batch_bcnn(images, varargin):**
```

```
 -> opts(i) = vl_argparse(opts(i), {varargin{1}(i),varargin{2:end}});
**function [opts, args] = vl_argparse(opts, args, varargin):**
```

Look one more step back, the `varargin` in function `imdb_get_batch_bcnn()` is passed in via `opts,'prefetch', nargout == 0` :

```
**function [im,labels] = getBatchSimpleNN(imdb, batch, opts):**
    -> im = imdb_get_batch_bcnn(images, opts, ...
                        'prefetch', nargout == 0);
```

Bingo! `opts` is the first element of `args` in `vl_argparse` . Therefore, at the very beginning when `ai = 1` , `isstruct(args{ai})` will also be `true` . So the next sentence is going to proceed:

```
    opts = vl_argparse(opts, vertcat(params,values)) ;
```

What is `opts` , `params` and `values` when program execution reach here?

```
opts =

         imageSize: [227 227]
            border: [0 0]
        keepAspect: 1
       numAugments: 1
    transformation: 'none'
      averageImage: []
       rgbVariance: [0x3 single]
     interpolation: 'bilinear'
        numThreads: 1
          prefetch: 0
             scale: 1

params =

    'imageSize'     'keepAspect'    'averageImage'    'border'        'cropSize'      'interpolation'
   'numThreads'

values =

    [1x4 double]    [         1]               []    [1x2 double]    [1x2 double]    'bilinear'
   [        12]
```

where `params` is the `fieldnames` of `args{1}` : `params = fieldnames(args{ai})` and `values` packs their correponding values: `values = struct2cell(args{ai})` .

So the function `vl_argparse(opts, args, varagin)` executes again with parameters the same `opts` and a new `args = vertcat(params,values)` .

Now none of the elements in `args` is a struct. The execution will then reach the code block, where the ERROR occurs:

```
% ./matconvnet/matlab/vl_argparse.m

  param = args{ai} ;
  value = args{ai+1} ;
  p = find(strcmpi(param, optNames)) ;
  if numel(p) ~= 1
    if nargout == 1
        error('Unkno parameter ''%s''', param) ;
    else
      keep([ai,ai+1]) = true ;
```

```
      ai = ai + 2 ;
      continue ;
    end
  end
```

This code block is in a `while` loop. It finds the elements in `params` which is the first row of `args` one by one in `optNames` at each iteration of the loop. So what is the `optNames` ? It is a pack of all the fieldnames of `opts` . Have a comparison:

```
params =

    'imageSize'     'keepAspect'    'averageImage'    'border'    'cropSize'      'interpolation'    '
numThreads'

optNames =

    'imageSize'    'border'    'keepAspect'    'numAugments'    'transformation'    'averageImage'
'rgbVariance'    'interpolation'    'numThreads'    'prefetch'    'scale'
```

All elements but `cropSize` in `params` appear in `optNames` !!

That's why this error occurs:

```
Error: Unkown parameter 'cropSize'
```

So the problem is two fold:

```
`cropSize` should not appear in `net.meta.normalization`, or
`cropSize` should be made to appear in `opts` of `vl_argparse`.
```

Since `net.meta.normalization` seems to be the last thing I should touch, I would try to set `cropSize` to `opts` firstly. To do this, Let's have a check where the `opts` of `vl_argparse` comes from, which means we should look into the execution stack to find out who calls function `vl_argparse` .

It is `imo = imdb_get_batch_bcnn(images, varargin)` :

```
% /bcnn/imdb_get_batch_bcnn.m
function imo = imdb_get_batch_bcnn(images, varargin)
% imdb_get_batch_bcnn  Load, preprocess, and pack images for BCNN evaluation
% For asymmetric model, the function preprocesses the images twice for two networks
% separately.

% OUTPUT
% imo: a cell array where each element is a cell array of images.
%       For symmetric bcnn model, numel(imo) will be 1 and imo{1} will be a
%       cell array of images
%       For asymmetric bcnn, numel(imo) will be 2. imo{1} is a cell array containing the preprocessed
images for network A
%       Similarly, imo{2} is a cell array containing the preprocessed images for network B

%
% Copyright (C) 2015 Tsung-Yu Lin, Aruni RoyChowdhury, Subhransu Maji.
% All rights reserved.
%
% This file is part of the BCNN and is made available under
% the terms of the BSD license (see the COPYING file).
%
% This file modified from CNN_IMAGENET_GET_BATCH of MatConvNet

for i=1:numel(varargin{1})
```

```
    opts(i).imageSize = [227, 227] ;
    opts(i).border = [0, 0] ;
    opts(i).keepAspect = true;
    opts(i).numAugments = 1 ;
    opts(i).transformation = 'none' ;
    opts(i).averageImage = [] ;
    opts(i).rgbVariance = zeros(0,3,'single') ;
    opts(i).interpolation = 'bilinear' ;
    opts(i).numThreads = 1 ;
    opts(i).prefetch = false;
    opts(i).scale = 1;
    opts(i) = vl_argparse(opts(i), {varargin{1}(i),varargin{2:end}});
```

Oh — My — God !!

…

Sir, the source code is poisonous !!

…

The code has built-in bugs!

Anyway just add a line `opts(i).cropSize = 227/256;` and give it another shot.

```
opts(i).scale = 1;
opts(i).cropSize = 227/256; % added by jcsu, 2016/11/16
opts(i) = vl_argparse(opts(i), {varargin{1}(i),varargin{2:end}});
```

OK, Let's remove output directory `checkgpu` and see what will happen! '

```
>> run_experiments_bcnn_train
```

Matlab standard outputs:

```
dataset: classes: 200 in use. These are:
   1: 001.Black_footed_Albatross (train:    30, test:    30 total:    60)
   2: 002.Laysan_Albatross (train:    30, test:    30 total:    60)
   3: 003.Sooty_Albatross (train:    30, test:    28 total:    58)
   4: 004.Groove_billed_Ani (train:    30, test:    30 total:    60)
   ...
```

```
199: 199.Winter_Wren (train:    30, test:    30 total:    60)
 200: 200.Common_Yellowthroat (train:    30, test:    30 total:    60)
 Inf:       **total** (train: 11788, test: 11788 total: 11788)
dataset: there are 11788 images (5994 trainval 5794 test)
Initialization: extracting bcnn feature of batch 1/185
Initialization: extracting bcnn feature of batch 2/185
...
```

```
Initialization: extracting bcnn feature of batch 184/185
Initialization: extracting bcnn feature of batch 185/185
     layer|      0|         1|      2|
      type|  input|      conv|softmxl|
      name|    n/a|classifier|   loss|
----------|-------|----------|-------|
   support|    n/a|         1|      1|
  filt dim|    n/a|    262144|    n/a|
filt dilat|    n/a|         1|    n/a|
 num filts|    n/a|       200|    n/a|
```

```
    stride|    n/a|         1|      1|
       pad|    n/a|         0|      0|
----------|-------|----------|-------|
   rf size|    n/a|         1|      1|
 rf offset|    n/a|         1|      1|
 rf stride|    n/a|         1|      1|
----------|-------|----------|-------|
 data size|NaNxNaN|   NaNxNaN|NaNxNaN|
data depth|    NaN|       200|      1|
  data num|    256|       256|      1|
----------|-------|----------|-------|
  data mem|    NaN|       NaN|    NaN|
 param mem|    n/a|     200MB|     0B|

parameter memory|200MB (5.2e+07 parameters)|
     data memory|  NaN (for batch size 256)|
```

Where is the `vl_nnbilinearpool`, `vl_nnsqrt` and `vl_nnl2norm` layers?

Well, since the network has been constructed without any error or warning and the `conv` and `softmaxloss` layers appear, I should trust the program. Notice that layers 1 and 2 are built-in layers, while the others are self-defined, maybe the reason is we do not provide it a way to output the related information.

```
cnn_train: resetting GPU
Clearing mex files
  CUDADevice with properties:

                      Name: 'GeForce GTX TITAN X'
                     Index: 1
         ComputeCapability: '5.2'
            SupportsDouble: 1
             DriverVersion: 8
            ToolkitVersion: 7
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 1.2800e+10
           AvailableMemory: 1.2623e+10
       MultiprocessorCount: 24
              ClockRateKHz: 1076000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 0
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1

train: epoch 01:   1/ 24: 471.8 (471.8) Hz objective: 5.298 top1err: 1.000 top5err: 0.980
train: epoch 01:   2/ 24: 582.1 (759.6) Hz objective: 5.298 top1err: 0.998 top5err: 0.979
train: epoch 01:   3/ 24: 634.4 (773.4) Hz objective: 5.297 top1err: 0.993 top5err: 0.966
train: epoch 01:   4/ 24: 661.0 (756.2) Hz objective: 5.297 top1err: 0.995 top5err: 0.967
train: epoch 01:   5/ 24: 756.4 (757.6) Hz objective: 5.298 top1err: 0.995 top5err: 0.971
...
...
train: epoch 01:  23/ 24: 761.8 (776.9) Hz objective: 5.290 top1err: 0.990 top5err: 0.963
train: epoch 01:  24/ 24: 761.3 (731.7) Hz objective: 5.290 top1err: 0.990 top5err: 0.962
val: epoch 01:   1/ 23: 840.6 (840.6) Hz objective: 5.234 top1err: 0.875 top5err: 0.711
val: epoch 01:   2/ 23: 822.7 (805.5) Hz objective: 5.236 top1err: 0.889 top5err: 0.740
```

```
val: epoch 01:   3/ 23: 823.1 (824.0) Hz objective: 5.237 top1err: 0.897 top5err: 0.736
val: epoch 01:   4/ 23: 826.1 (835.1) Hz objective: 5.236 top1err: 0.898 top5err: 0.729
...
...
val: epoch 25:  20/ 23: 839.8 (845.5) Hz objective: 3.568 top1err: 0.473 top5err: 0.203
val: epoch 25:  21/ 23: 840.4 (852.4) Hz objective: 3.565 top1err: 0.470 top5err: 0.203
val: epoch 25:  22/ 23: 840.7 (848.2) Hz objective: 3.570 top1err: 0.472 top5err: 0.204
val: epoch 25:  23/ 23: 840.8 (841.4) Hz objective: 3.573 top1err: 0.475 top5err: 0.206
train: epoch 26:   1/ 24: 609.7 (609.7) Hz objective: 3.146 top1err: 0.207 top5err: 0.043
train: epoch 26:   2/ 24: 679.4 (767.2) Hz objective: 3.117 top1err: 0.203 top5err: 0.059
train: epoch 26:   3/ 24: 705.6 (764.5) Hz objective: 3.114 top1err: 0.203 top5err: 0.061
train: epoch 26:   4/ 24: 722.1 (776.6) Hz objective: 3.102 top1err: 0.195 top5err: 0.058
...
...
train: epoch 31:  20/ 24: 762.8 (785.9) Hz objective: 2.797 top1err: 0.185 top5err: 0.051
train: epoch 31:  21/ 24: 763.6 (780.9) Hz objective: 2.796 top1err: 0.184 top5err: 0.051
train: epoch 31:  22/ 24: 764.2 (775.9) Hz objective: 2.797 top1err: 0.184 top5err: 0.051
train: epoch 31:  23/ 24: 765.1 (786.0) Hz objective: 2.795 top1err: 0.182 top5err: 0.051
train: epoch 31:  24/ 24: 763.9 (703.1) Hz objective: 2.796 top1err: 0.183 top5err: 0.051
val: epoch 31:   1/ 23: 830.9 (830.9) Hz objective: 3.241 top1err: 0.453 top5err: 0.227
val: epoch 31:   2/ 23: 836.7 (842.6) Hz objective: 3.260 top1err: 0.438 top5err: 0.203
val: epoch 31:   3/ 23: 837.6 (839.3) Hz objective: 3.263 top1err: 0.436 top5err: 0.191
val: epoch 31:   4/ 23: 838.1 (839.8) Hz objective: 3.272 top1err: 0.438 top5err: 0.194
val: epoch 31:   5/ 23: 841.5 (848.4) Hz objective: 3.280 top1err: 0.436 top5err: 0.196
...
...
val: epoch 40:  19/ 23: 796.7 (796.4) Hz objective: 2.996 top1err: 0.435 top5err: 0.178
val: epoch 40:  20/ 23: 793.6 (740.2) Hz objective: 2.999 top1err: 0.433 top5err: 0.178
val: epoch 40:  21/ 23: 794.1 (804.5) Hz objective: 2.997 top1err: 0.432 top5err: 0.177
val: epoch 40:  22/ 23: 792.2 (752.8) Hz objective: 3.003 top1err: 0.435 top5err: 0.178
val: epoch 40:  23/ 23: 792.5 (803.9) Hz objective: 3.003 top1err: 0.436 top5err: 0.179
train: epoch 41:   1/ 24: 648.2 (648.2) Hz objective: 2.318 top1err: 0.141 top5err: 0.023
train: epoch 41:   2/ 24: 690.4 (738.5) Hz objective: 2.348 top1err: 0.133 top5err: 0.033
train: epoch 41:   3/ 24: 709.1 (749.8) Hz objective: 2.332 top1err: 0.141 top5err: 0.042
train: epoch 41:   4/ 24: 720.9 (758.6) Hz objective: 2.360 top1err: 0.153 top5err: 0.045
train: epoch 41:   5/ 24: 758.4 (757.5) Hz objective: 2.336 top1err: 0.142 top5err: 0.039
train: epoch 41:   6/ 24: 758.1 (756.9) Hz objective: 2.331 top1err: 0.143 top5err: 0.041
...
...
```

Good, It seems to work quite well!

Questions remain:

```
1. Has the `bcnn` network been correctly constructed?
2. The training outputs a final model, is it the final results that can reproduce the paper's experime
ntal results?
3. How to fine-tune the hyper-parameters?
4. How to use the final model?
5. How to transplant it to Caffe framework?
6. Installation of `Caffe` ...
7. ...
```