

Quick start

If you are new to MatConvNet, cut & paste the following code in a MATLAB window to try out MatConvNet. The code downloads and compiles MatConvNet, downloads a pre-trained CNN, and uses the latter to classify one of MATLAB stock images.

This example requires MATLAB to be interfaced to a C/C++ compiler (try `mex -setup` if you are unsure). Depending on your Internet connection speed, downloading the CNN model may require some time.

```
% install and compile MatConvNet (needed once)
untar('http://www.vlfeat.org/matconvnet/download/matconvnet-1.0-beta17.tar.gz') ;
cd matconvnet-1.0-beta17
run matlab/vl_compilenn

% download a pre-trained CNN from the web (needed once)
urlwrite(...
    'http://www.vlfeat.org/matconvnet/models/imagenet-vgg-f.mat',
    ...
    'imagenet-vgg-f.mat') ;

% setup MatConvNet
run matlab/vl_setupnn

% load the pre-trained CNN
net = load('imagenet-vgg-f.mat') ;

% load and preprocess an image
im = imread('peppers.png') ;
im_ = single(im) ; % note: 0-255 range
im_ = imresize(im_, net.meta.normalization.imageSize(1:2)) ;
im_ = im_ - net.meta.normalization.averageImage ;

% run the CNN
res = vl_simplenn(net, im_) ;

% show the classification result
scores = squeeze(gather(res(end).x)) ;
[bestScore, best] = max(scores) ;
figure(1) ; clf ; imagesc(im) ;
title(sprintf('%s (%d), score %.3f', ...
    net.meta.classes.description{best}, best, bestScore)) ;
```

In order to compile the GPU support and other advanced features, see the installation instructions (`../install/`).

Using DAG models

The example above exemplifies using a model using the SimpleNN wrapper. More complex models use instead the DagNN wrapper. For example, to run GoogLeNet use:

```
% download a pre-trained CNN from the web (needed once)
urlwrite(...
    'http://www.vlfeat.org/matconvnet/models/imagenet-googlenet-dag.mat', ...
    'imagenet-googlenet-dag.mat') ;

% load the pre-trained CNN
net = dagnn.DagNN.loadobj(load('imagenet-googlenet-dag.mat')) ;

% load and preprocess an image
im = imread('peppers.png') ;
im_ = single(im) ; % note: 0-255 range
im_ = imresize(im_, net.meta.normalization.imageSize(1:2)) ;
im_ = im_ - net.meta.normalization.averageImage ;

% run the CNN
net.eval({'data', im_}) ;

% obtain the CNN output
scores = net.vars(net.getVarIndex('prob')).value ;
scores = squeeze(gather(scores)) ;

% show the classification results
[bestScore, best] = max(scores) ;
figure(1) ; clf ; imagesc(im) ;
title(sprintf('%s (%d), score %.3f', ...
    net.meta.classes.description{best}, best, bestScore)) ;
```

Installing and compiling the library

In order to install the library, follows these steps:

1. Download and unpack the library source code into a directory of your choice. Call the path to this directory <MatConvNet>.
2. Compile the library.
3. Start MATLAB and type:

```
> run <MatConvNet>/matlab/vl_setupnn
```

in order to add MatConvNet to MATLAB's search path.

At this point the library is ready to use. You can test it by using the command (using MATLAB R2014a or later):

```
> vl_testnn
```

To test GPU support (if you have compiled it) use instead:

```
> vl_testnn('gpu', true)
```

Note that the second tests runs slower than the CPU version; do not worry, this is an artefact of the test procedure.

Compiling

MatConvNet compiles under Linux, Mac, and Windows. This page discusses compiling MatConvNet using the MATLAB function [vl_compilenn](#) (../mfiles/vl_compilenn). While this is the easiest method, the command line or an IDE can be used as well (../install-alt/).

Compiling for CPU

If this is the first time you compile MatConvNet, consider trying first the CPU version. In order to do this, use the [vl_compilenn](#) (../mfiles/vl_compilenn) command supplied with the library:

1. Make sure that MATLAB is configured to use your compiler (http://www.mathworks.com/help/matlab/matlab_external/changing-default-compiler.html).
2. Open MATLAB and issue the commands:

```
> cd <MatConvNet>
> addpath matlab
> vl_compilenn
```

At this point MatConvNet should start compiling. If all goes well, you are ready to use the library. If not, you can try debugging the problem by running the compilation script again in verbose mode:

```
> vl_compilenn('verbose', 1)
```

Increase the verbosity level to 2 to get even more information.

Remark: The 'vl_imreadjpeg' tool uses an external image library to load images. In Mac OS X and Windows, the default is to use the system libraries (Quartz and GDI+ respectively), so this dependency is immaterial. In Linux, this tool requires the LibJPEG library and the corresponding development files to be installed in the system. If needed, the ImageLibraryCompileFlags and ImageLibraryLinkFlags options can be used to adjust the compiler and linker flags to match a specific library installation. It is also possible to use the EnableImreadJpeg option of vl_compilenn to turn off this feature.

Compiling the GPU support

To use the GPU-accelerated version of the library, you will need a NVIDIA GPU card with compute capability 2.0 or greater and a copy of the NVIDIA CUDA toolkit. Ideally, the version of the CUDA toolkit should match your MATLAB version:

MATLAB	CUDA toolkit
R2013b	5.5
R2014a	5.5
R2014b	6.0
R2015a	6.5
R2015b	7.0

You can also use the gpuDevice MATLAB command to find out MATLAB's version of the CUDA toolkit. It is also possible (and often necessary) to use a more recent version of CUDA than the one officially supported by MATLAB; this is explained later.

Assuming that there is only a single copy of the CUDA toolkit installed in your system and that it matches MATLAB's version, compile the library with:

```
> vl_compilenn('enableGpu', true)
```

If you have multiple versions of the CUDA toolkit, or if the script cannot find the toolkit for any reason, specify the path to the CUDA toolkit explicitly. For example, on a Mac this may look like:

```
> vl_compilenn('enableGpu', true, 'cudaRoot', '/Developer/NVIDIA  
A/CUDA-7.0')
```

Once more, you can use the verbose option to obtain more information if needed.

Using an unsupported CUDA toolkit version

MatConvNet can be compiled to use a more recent version of the CUDA toolkit than the one officially supported by MATLAB. While this may cause unforeseen issues (although none is known so far), it is necessary to use recent libraries such as cuDNN.

Compiling with a newer version of CUDA requires using the `cudaMethod,nvcc` option. For example, on a Mac this may look like:

```
> vl_compilenn('enableGpu', true, ...  
               'cudaRoot', '/Developer/NVIDIA/CUDA-7.0', ...  
               'cudaMethod', 'nvcc')
```

Note that at this point MatConvNet MEX files are linked *against the specified CUDA libraries* instead of the one distributed with MATLAB. Hence, in order to use MatConvNet it is now necessary to allow MATLAB accessing these libraries. On Linux one way to do so is to start MATLAB from the command line (terminal) specifying the `LD_LIBRARY_PATH` option. For instance, on Linux this may look like:

```
$ LD_LIBRARY_PATH=/Developer/NVIDIA/CUDA-7.0/lib64 matlab
```

On Windows, chances are that the CUDA libraries are already visible to MATLAB so that nothing else needs to be done.

On Mac, this step should not be necessary as the library paths are hardcoded during compilation.

Compiling the cuDNN support

MatConvNet supports the NVIDIA cuDNN library (<https://developer.nvidia.com/cuDNN>) for deep learning (and in particular their fast convolution code). In order to use it, obtain the cuDNN (<http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library>) library from NVIDIA (cuDNN v2 to v4 should work; however, later version are *strongly recommended* as earlier version had a few bugs). Make sure that the CUDA toolkit matches the one in cuDNN (e.g. 6.5). This often means that the CUDA toolkit will *not* match the one used internally by MATLAB, such that the compilation method discussed above must be used.

Unpack the cuDNN library binaries and header files in a place `<Cudnn>` of your choice. In the rest of this example, it will be assumed that this cuDNN RC4 has been unpacked in `local/cudnn-rc4` in the `<MatConvNet>` root directory, (i.e. `<Cudnn>=<MatConvNet>/local/cudnn-rc4`). For example, the directory structure on a Mac should look like:

```

COPYING
Makefile
Makefile.mex
...
examples/
local/
  cudnn-rc4/
    include/
      cudnn.h
    lib/
      libcudnn.7.5.dylib
      libcudnn.dylib
  ...

```

Use `vl_compilenn` with the `cudnnEnable,true` option to compile the library; do not forget to use `cudaMethod,nvcc` as, at it is likely, the CUDA toolkit version is newer than MATLAB's CUDA toolkit. For example, on Mac this may look like:

```

> vl_compilenn('enableGpu', true, ...
               'cudaRoot', '/Developer/NVIDIA/CUDA-7.5', ...
               'cudaMethod', 'nvcc', ...
               'enableCudnn', 'true', ...
               'cudnnRoot', 'local/cudnn-rc4') ;

```

MatConvNet is now compiled with cuDNN support. When starting MATLAB, however, do not forget to point it to the paths of both the CUDA and cuDNN libraries. On a Linux terminal, this may look like:

```

$ cd <MatConvNet>
$ LD_LIBRARY_PATH=/Developer/NVIDIA/CUDA-7.5/lib64:local matlab

```

On Windows, copy the cuDNN DLL file `<Cudnn>/cudnn*.dll` (or from wherever you unpacked cuDNN) into the `<MatConvNet>/matlab/mex` directory.

On Mac, this step should not be necessary as the library paths are hardcoded during compilation.

Further examples

To compile all the features in MatConvNet on a Mac and MATLAB 2014b, CUDA toolkit 6.5 and cuDNN Release Candidate 2, use:

```

> vl_compilenn('enableGpu', true, 'cudaMethod', 'nvcc', ...
               'cudaRoot', '/Developer/NVIDIA/CUDA-6.5', ...
               'enableCudnn', true, 'cudnnRoot', 'local/cudnn-rc
2') ;

```

The equivalent command on Ubuntu Linux would look like:

```
> vl_compilenn('enableGpu', true, 'cudaMethod', 'nvcc', ...  
               'cudaRoot', '/opt/local/cuda-6.5', ...  
               'enableCudnn', true, 'cudnnRoot', 'local/cudnn-rc  
2') ;
```

Using MATLAB 2015b, CUDA 7.5, and cuDNN R4:

```
> vl_compilenn('enableGpu', true, ...  
               'cudaMethod', 'nvcc', ...  
               'cudaRoot', '/opt/local/cuda-7.5', ...  
               'enableCudnn', true, ...  
               'cudnnRoot', 'local/cudnn-rc4') ;
```

CNN wrappers

At its core, MatConvNet consists of a number of MATLAB functions (`./functions/#blocks`) implementing CNN building blocks. These are usually combined into complete CNNs by using one of the two CNN wrappers. The first wrapper is SimpleNN, most of which is implemented by the MATLAB function `vl_simplenn` (`./mfiles/simplenn/vl_simplenn/`). SimpleNN is suitable for networks that have a linear topology, i.e. a chain of computational blocks. The second wrapper is DagNN, which is implemented as the MATLAB class `dagnn.DagNN` (`./mfiles/+dagnn/@DagNN/DagNN/`).

SimpleNN wrapper

The SimpleNN wrapper is implemented by the function `vl_simplenn` (`./mfiles/simplenn/vl_simplenn/`) and a few others (`./functions/#simplenn`). This is a lightweight wrapper, suitable for CNN consisting of a simple chain of blocks.

To start with SimpleNN, create a `net` structure, populating the cellarray `net.layers` with a list of layers. For example:

```
net.layers{1} = struct(...
    'name', 'conv1', ...
    'type', 'conv', ...
    'weights', {{randn(10,10,3,2,'single'), randn(2,1,'single')}}}, ...
    'pad', 0, ...
    'stride', 1) ;
net.layers{2} = struct(...
    'name', 'relu1', ...
    'type', 'relu') ;
```

Now the convolutional and ReLU layers will be executed in sequence. The convolution has a bank of two 10x10x3 filters. Evaluation can be obtained as follows:

```
data = randn(300, 500, 3, 5, 'single') ;
res = vl_simplenn(net, data) ;
```

The structure `res` contains the result of the computation, with one entry for each variable in the architecture:


```
>> res
res =
1x3 struct array with fields:

    x
    dzdx
    dzdw
    aux
    time
    backwardTime
```

Here `x` is the variable value, `dzdx` the derivative of the CNN with respect to `x`, `dzdw` the derivative of the CNN with respect to each of the block parameters, `aux` space for custom information (e.g. the mask in dropout layers), and `time` and `backwardTime` the time spent in the forward and backward pass.

For example, `res(1).x` is the input of the CNN and `res(3).x` its output.

The derivative of the CNN can be computed as follows:

```
res = vl_simplenn(res, data, dzdy)
```

This performs both a forward and a backward pass. `dzdy` is a projection applied to the output value of the CNN (see the PDF manual ([../matconvnet-manual.pdf](#)) to clarify this point). During training, CNNs are often terminated by a block that computes a single scalar loss value (i.e. `res(end).x` is a scalar). In this case, one often picks `dzdy = 1`.

DagNN wrapper

The DagNN wrapper is implemented by the class `daggn.DagNN` (`../mfiles/+daggn/@DagNN/DagNN/`).

Creating a DagNN

A DagNN is an object of class `daggn.DagNN` (`../mfiles/+daggn/@DagNN/DagNN/`). A DAG can be created directly as follows:

```
run <MATCONVNETROOT>/matlab/vl_setupnn.m ; % activate MatConvNet
if needed
net = daggn.DagNN() ;
```

The object is a MATLAB handle, meaning that it is passed by reference rather than by value:

```
net2 = net ; % both net and net2 refer to the same object
```

This significantly simplifies the syntax of most operations.

DagNN has a bipartite directed acyclic graph structure, where *layers* are connected to *variables* and vice-versa. Each layer receives zero or more variables and zero or more *parameters* as input and produces zero or more variables as outputs. Layers are added using the `addLayer()` method of the DagNN object. For example, the following command adds a layer an input `x1`, an output `x2`, and two parameters `filters` and `biases`.

```
convBlock = dagnn.Conv('size', [3 3 256 16], 'hasBias', true) ;
net.addLayer('conv1', convBlock, {'x1'}, {'x2'}, {'filters', 'biases'}) ;
```

Next, we add a ReLU layer on top of the convolutional one:

```
reluBlock = dagnn.ReLU() ;
net.addLayer('relu1', reluBlock, {'x2'}, {'x3'}, {}) ;
```

Note that ReLU does not take any parameter. In general, blocks may have an arbitrary numbers of inputs and outputs (compatibly with the block type).

At this point, `net` contains the two blocks as well as the three variables and the two parameters. These are stored as entries in `net.layers`, `net.vars` and `net.params` respectively. For example

```
>> net.layers(1)
ans =
      name: 'conv1'
    inputs: {'x1'}
   outputs: {'x2'}
    params: {'filters'  'biases'}
inputIndexes: 1
outputIndexes: 2
paramIndexes: [1 2]
      block: [1x1 dagnn.Conv]
```

contains the first block. It includes the names of the input and output variables and of the parameters (e.g. `net.layers(1).inputs`). It also contains the variable and parameter indexes for faster access (these are managed automatically by the DAG methods). Blocks are identified by name (`net.layers(1).name`) and some functions such as `removeLayer()` refer to them using these identifiers (it is an error to assign the same name to two layers). Finally, the actual layer parameters are contained in `net.layers(1).block`, which in this case is an object of type `dagnn.Conv`. The latter is, in turn, a wrapper of the `vl_nnconv` command.

The other important data members store variables and parameters. For example:

```

>> net.vars(1)
ans =
    name: 'x1'
   value: []
    der: []
   fanin: 0
  fanout: 1
 precious: 0

>> net.params(1)
ans =
    name: 'filters'
   value: []
    der: []
  fanout: 1
learningRate: 1
weightDecay: 1

```

Note that each variable and parameter has a `value` and a `derivative` members. The `fanin` and `fanout` members indicated how many network layers have that particular variable/parameter as output and input, respectively. Parameters do not have a `fanin` as they are not the result of a network calculation. Variables can have `fanin` equal to zero, denoting a network input, or one. Network outputs can be identified as variables with null `fanout`.

Variables and parameters can feed into one or more layers, which results in a `fanout` equal or greater than one. For variables, `fanout` greater than one denotes a branching point in the DAG. For parameters, `fanout` greater than one allows sharing parameters between layers.

Loading and saving DagNN objects

While it is possible to save a DagNN object using MATLAB `save` command directly, this is not recommended. Instead, for compatibility and portability, as well as to save significant disk space, it is preferable to convert the object into a structure and then save that instead:

```

netStruct = net.saveobj() ;
save('myfile.mat', '-struct', 'netStruct') ;
clear netStruct ;

```

The operation can be inverted using the `loadobj()` static method of `dagnn.DagNN`:

```

netStruct = load('myfile.mat') ;
net = dagnn.DagNN.loadobj(netStruct) ;
clear netStruct ;

```

Note that in this manner the *transient* state of the object is lost. This includes the values of the variables, but not the values of the parameters, or the network structure.

Using the DagNN to evaluate the network and its derivative

So far, both parameters and variables in the DagNN object have empty values and derivatives. The command:

```
net.initParams() ;
```

can be used to initialise the model parameters to random values. For example `net.params(1).value` should now contain a 3x3x256x16 array, matching the filter dimensions specified in the example above.

The `eval()` method can now be used to evaluate the network. For example:

```
input = randn(10,15,256,1,'single') ;  
net.eval({'x1', input}) ;
```

evaluates the network on a random input array. Since in general the network can have several inputs, one must specify each input as a pair 'variableName',variableValue in a cell array.

After `eval()` completes, the `value` fields of the leaf variables in the network contain the network outputs. In this example, the single output can be recovered as follows:

```
i = net.getVarIndex('x3') ;  
output = net.vars(i).value ;
```

The `getVarIndex()` method is used to obtain the variable index given its name (variables do not move around unless the network structure is changed or reloaded from disk, so that indexes can be cached).

The (projected) derivative of the CNN with respect to variables and parameters is obtained by passing, along with the input, a projection vector for each of the output variables:

```
dzdy = randn(size(output), 'single') ; % projection vector  
net.eval({'x1',input},{'x3',dzdy}) ;
```

The derivatives are now stored in the corresponding `params` and `vars` structure entries. For example, the derivative with respect to the `filters` parameter can be accessed as follows:

```
p = net.getParamIndex('filters') ;  
dzdfilters = net.vars(p).der ;
```

Remark: empty values of variables. Note that most intermediate variable values and derivatives are aggressively discarded during the computation in order to conserve memory. Set `net.conserveMemory` to `false` to prevent this from happening, or make individual variables precious (`net.vars(i).precious = true`).

Pretrained models

This section describes how pre-trained models can be downloaded and used in MatConvNet. Using the pre-trained model is easy; just start from the example code included in the quickstart guide ([../quick/](#)).

Remark: The following CNN models may have been *imported from other reference implementations* and are equivalent to the originals up to numerical precision. However, note that:

1. Images need to be pre-processed (resized and cropped) before being submitted to a CNN for evaluation. Even small differences in the preprocessing details can have a non-negligible effect on the results.
2. The example below shows how to evaluate a CNN, but does not include data augmentation or encoding normalization as for example provided by the VGG code (http://www.robots.ox.ac.uk/~vgg/research/deep_eval). While this is easy to implement, it is not done automatically here.
3. These models are provided here for convenience, but please credit the original authors.

Face recognition

These models are trained for face classification and verification.

- **VGG-Face**. The face classification and verification network from the VGG project (http://www.robots.ox.ac.uk/~vgg/software/vgg_face/).

Deep face recognition, O. M. Parkhi and A. Vedaldi and A. Zisserman, Proceedings of the British Machine Vision Conference (BMVC), 2015 (paper (<http://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf>)).

- vgg-face ([../models/vgg-face.mat](#))  ([../models/vgg-face.svg](#))




See the script `examples/cnn_vgg_face.m` for an example of using VGG-Face for classification. To use this network for face verification instead, extract the 4K dimensional features by removing the last classification layer and normalize the resulting vector in L2 norm.

Semantic segmentation

These models are trained for semantic image segmentation using the PASCAL VOC category definitions.

- **Fully-Convolutional Networks (FCN)** training and evaluation code is available here (<https://github.com/vlfeat/matconvnet-fcn>).
- **BVLC FCN** (the original implementation) imported from the Caffe version (<https://github.com/BVLC/caffe/wiki/Model-Zoo>) [*DagNN format*].

'Fully Convolutional Models for Semantic Segmentation', Jonathan Long, Evan Shelhamer and Trevor Darrell, CVPR, 2015 (paper (http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Long_Fully_Convolutional_Networks_2015_CVPR_paper.pdf)).

- pascal-fcn32s-dag ([../models/pascal-fcn32s-dag.mat](#))  ([../models/pascal-fcn32s-dag.svg](#))
- pascal-fcn16s-dag ([../models/pascal-fcn16s-dag.mat](#))  ([../models/pascal-fcn16s-dag.svg](#))
- pascal-fcn8s-dag ([../models/pascal-fcn8s-dag.mat](#))  ([../models/pascal-fcn8s-dag.svg](#))

These networks are trained on the PASCAL VOC 2011 training and (in part) validation data, using Berkeley's extended annotations (SBD (<http://www.cs.berkeley.edu/~bharath2/codes/SBD/download.html>)).


The performance measured on the PASCAL VOC 2011 validation data subset used in the revised version of the paper above (dubbed RV-VOC11):

Model	Test data	Mean IOU	Mean pix. accuracy	Pixel accuracy
FNC-32s	RV-VOC11	59.43	89.12	73.28

FNC-16s	RV-VOC11	62.35	90.02	75.74
FNC-8s	RV-VOC11	62.69	90.33	75.86

- **Torr Vision Group FCN-8s.** This is the FCN-8s subcomponent of the CRF-RNN network from the paper:

'Conditional Random Fields as Recurrent Neural Networks' *Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip H. S. Torr*, ICCV 2015 (paper (<http://www.robots.ox.ac.uk/~szheng/papers/CRFasRNN.pdf>)).

- pascal-fcn8s-tvg-dag (../models/pascal-fcn8s-tvg-dag.mat)  (../models/pascal-fcn8s-tvg-dag.svg)

These networks are trained on the PASCAL VOC 2011 training and (in part) validation data, using Berkeley's extended annotations, as well as Microsoft COCO.

While the CRF component is missing (it may come later to MatConvNet), this model still outperforms the FCN-8s network above, partially because it is trained with additional data from COCO. In the table below, the RV-VOC12 data is the subset of the PASCAL VOC 12 data as described in the 'Conditional Random Fields' paper:

Model	Tes data	mean IOU	mean pix. accuracy	pixel accuracy
FNC-8s-TVG	RV-VOC12	69.85	92.94	78.80


TVG implementation note: The model was obtained by first fine-tuning the plain FCN-32s network (without the CRF-RNN part) on COCO data, then building built an FCN-8s network with the learnt weights, and finally training the CRF-RNN network end-to-end using VOC 2012 training data only. The model available here is the FCN-8s part of this network (without CRF-RNN, while trained with 10 iterations CRF-RNN).

ImageNet ILSVRC classification

These models are trained to perform classification in the ImageNet ILSVRC challenge data.



- **GoogLeNet** model imported from the Princeton version (<http://vision.princeton.edu/pvt/GoogLeNet/>) [*DagNN format*].

'Going Deeper with Convolutions', *Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich*, CVPR, 2015 (paper (http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf)).

- imagenet-googlenet-dag (../models/imagenet-googlenet-dag.mat)  (../models/imagenet-googlenet-dag.svg)


- **VGG-VD** models from the Very Deep Convolutional Networks for Large-Scale Visual Recognition (http://www.robots.ox.ac.uk/~vgg/research/very_deep/) project.






'Very Deep Convolutional Networks for Large-Scale Image Recognition', *Karen Simonyan and Andrew Zisserman*, arXiv technical report, 2014, (paper (<http://arxiv.org/abs/1409.1556/>)).

- imagenet-vgg-verydeep-16 (../models/imagenet-vgg-verydeep-16.mat)  (../models/imagenet-vgg-verydeep-16.svg)
- imagenet-vgg-verydeep-19 (../models/imagenet-vgg-verydeep-19.mat)  (../models/imagenet-vgg-verydeep-19.svg)





- **VGG-S,M,F** models from the Return of the Devil (http://www.robots.ox.ac.uk/~vgg/research/deep_eval) paper (v1.0.1).

'Return of the Devil in the Details: Delving Deep into Convolutional Networks', *Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman*, BMVC 2014 (BibTex and paper (<http://www.robots.ox.ac.uk/~vgg/publications/2014/Chatfield14/>)).

- imagenet-vgg-f (../models/imagenet-vgg-f.mat)  (../models/imagenet-vgg-f.svg)


- imagenet-vgg-m (../models/imagenet-vgg-m.mat)  (../models/imagenet-vgg-m.svg)
- imagenet-vgg-s (../models/imagenet-vgg-s.mat)  (../models/imagenet-vgg-s.svg)
- imagenet-vgg-m-2048 (../models/imagenet-vgg-m-2048.mat)  (../models/imagenet-vgg-m-2048.svg)
- imagenet-vgg-m-1024 (../models/imagenet-vgg-m-1024.mat)  (../models/imagenet-vgg-m-1024.svg)
- imagenet-vgg-m-128 (../models/imagenet-vgg-m-128.mat)  (../models/imagenet-vgg-m-128.svg)

The following models have been trained using MatConvNet (beta17) and batch normalization using the code in the `examples/imagenet` directory:

- imagenet-matconvnet-vgg-f (../models/imagenet-matconvnet-vgg-f.mat)  (../models/imagenet-matconvnet-vgg-f.svg)
- imagenet-matconvnet-vgg-m (../models/imagenet-matconvnet-vgg-m.mat)  (../models/imagenet-matconvnet-vgg-m.svg)
- imagenet-matconvnet-vgg-s (../models/imagenet-matconvnet-vgg-s.mat)  (../models/imagenet-matconvnet-vgg-s.svg)
- imagenet-matconvnet-vgg-verydeep-16 (../models/imagenet-matconvnet-vgg-verydeep-16.mat)  (../models/imagenet-matconvnet-vgg-verydeep-16.svg)



- **Caffe reference model** obtained here (http://caffe.berkeleyvision.org/getting_pretrained_models.html) (version downloaded on September 2014).

Citation: please see the Caffe homepage (<http://caffe.berkeleyvision.org>).

- imagenet-caffe-ref (../models/imagenet-caffe-ref.mat)  (../models/imagenet-caffe-ref.svg)

- **AlexNet**

'ImageNet classification with deep convolutional neural networks', A. Krizhevsky and I. Sutskever and G. E. Hinton, NIPS 2012 (BibTex and paper (<http://papers.nips.cc/paper/4824-imagenet-classification-with-deep->))

- imagenet-caffe-alex (../models/imagenet-caffe-alex.mat)  (../models/imagenet-caffe-alex.svg)
- imagenet-matconvnet-alex (../models/imagenet-matconvnet-alex.mat)  (../models/imagenet-matconvnet-alex.svg)

The first model has been imported from Caffe (http://caffe.berkeleyvision.org/getting_pretrained_models.html).

The MatConvNet model was trained using MatConvNet (beta17) and batch normalization using the code in the `examples/imagenet` directory.

This is a summary of the performance of these models on the ILSVRC 2012 validation data:

model	top-1 err.	top-5 err.	images/s
matconvnet-alex	41.8	19.2	547.3
matconvnet-vgg-s	37.0	15.8	337.4
matconvnet-vgg-m	36.9	15.5	422.8
matconvnet-vgg-f	41.4	19.1	658.8
matconvnet-vgg-verydeep-16	28.3	9.5	79.1
caffe-ref	42.4	19.6	336.7
caffe-alex	42.6	19.6	332.2
vgg-s	36.7	15.3	321.7
vgg-m	37.3	15.9	404.3
vgg-f	41.1	18.8	661.6
vgg-m-128	40.8	18.4	388.5

vgg-m-1024	37.8	16.1	406.9
vgg-m-2048	37.1	15.8	401.6
vgg-verydeep-19	28.7	9.9	60.9
vgg-verydeep-16	28.5	9.9	73.9
googlenet-dag	32.2	11.6	231.4

Important notes:

- The model trained using MatConvNet are slightly better than the original, probably due to the use of batch normalization during training.
- Error rates are computed on a single centre-crop and are therefore higher than what reported in some publications, where multiple evaluations per image are combined.
- The **evaluation speed** was measured on a 12-cores machine using a single NVIDIA Titan Black GPU, MATLAB R2015a, and CuDNN v4; performance varies hugely depending on the network but also on how the data was preprocessed; for example, `caffe-ref` and `caffe-alex` should be as fast as `matconvnet-alex`, but they are not since images were pre-processed in such a way that MATLAB had to call `imresize` for each input image for the Caffe models.
- The GoogLeNet model performance is a little lower than expected (the model should be on par or a little better than VGG-VD). This network was imported from the Princeton version of GoogLeNet, not by the Google team, so the difference might be due to parameter setting during training. On the positive side, GoogLeNet is much smaller (in terms of parameters) and faster than VGG-VD.

File checksums

The following table summarizes the MD5 checksums for the model files.

MD5	File name
77ba5337725eb77362e9f318898af494	imagenet-caffe-alex.mat
7001959cb66a3d62a86d52efff42f168	imagenet-caffe-ref.mat
e646ea925dee772e34794f01ebbe1bd8	imagenet-googlenet-dag.mat
d79a53b79b62aee8a6c48755c29448fa	imagenet-matconvnet-alex.mat
5c00773832303a2a9656afec097fb1c1	imagenet-matconvnet-vgg-f.mat
2ccbc5c4d77a56fbfc288ca810d12206	imagenet-matconvnet-vgg-m.mat
79c64eedb1fa49668997342b02dea863	imagenet-matconvnet-vgg-s.mat
7502ff082bf53ce9cc67110399bbdf53	imagenet-matconvnet-vgg-verydeep-16.mat
4775484a70e8bac3e9521aed59f31dfc	imagenet-vgg-f.mat
0545aea6fc5173b2806784f2a8bd3333	imagenet-vgg-m-1024.mat
80ec27ef99e2faefb9c837216c9ea0e4	imagenet-vgg-m-128.mat
620aca6468345e4a791a0396c6a51ce1	imagenet-vgg-m-2048.mat
b4e8616c0ab66b1fda72854226f82d02	imagenet-vgg-m.mat
2f83043a38e71e9dd9b1c5c0cb3ef6f9	imagenet-vgg-s.mat
5a68244cf55c66fea59e23ee63cf56ef	imagenet-vgg-verydeep-16.mat
b9b4a9eb1c2fb3b50e1ec1aca6f22342	imagenet-vgg-verydeep-19.mat

210308543d1a510239ed85feb9b2e885	pascal-fcn16s-dag.mat
2dea567374085a63ad6e83b7b08fa482	pascal-fcn32s-dag.mat
8db84f60ba7d519de15cdf9d2c9a40e1	pascal-fcn8s-dag.mat
ad374f3aa98208847489e5c7bfd8e013	pascal-fcn8s-tvg-dag.mat
5069daad93d2937554325e30388463ca	vgg-face.mat

Older file versions

Older models for MatConvNet beta16 are available here ([../models/beta16](#)). They should be numerically equivalent, but in beta17 the format has changed slightly for SimpleNN models. Older models can also be updated using the `vl_simplenn_tidy` function.

Using MatConvNet to train convnets

MatConvNet can be used to train models, typically by using a form of stochastic gradient descent (SGD) and back-propagation.

The following learning demonstrators are provided in the MatConvNet package:

- **MNIST**. See `examples/mnist/cnn_mnist.m`.
- **CIFAR**. See `examples/cifar/cnn_cifar.m`.
- **ImageNet**. See `examples/imagenet/cnn_imagenet.m`.

These demos are self-contained; MNIST and CIFAR, in particular, automatically download and unpack the required data, so that they should work out-of-the-box.

While MNIST and CIFAR are small datasets (by today's standard) and training is feasible on a CPU, ImageNet requires a powerful GPU to complete in a reasonable time (a few days!). It also requires the `vl_imreadjpeg()` command in the toolbox to be compiled in order to accelerate reading large batches of JPEG images and avoid starving the GPU.

All these demos use the `example/cnn_train.m` and `example/cnn_train_dag.m` SGD drivers, which are simple implementations of the standard SGD with momentum, done directly in MATLAB code. However, it should be easy to implement your own specialized or improved solver.

Using GPU acceleration

GPU support in MatConvNet builds on top of MATLAB GPU support in the Parallel Programming Toolbox. This toolbox requires CUDA-compatible cards, and you will need a copy of the corresponding CUDA devkit (<https://developer.nvidia.com/cuda-toolkit-archive>) to compile GPU support in MatConvNet (see compiling (`../install#compiling`)).

All the core computational functions (e.g. `v1_nnconv`) in the toolbox can work with either MATLAB arrays or MATLAB GPU arrays. Therefore, switching to use the a GPU is as simple as converting the input CPU arrays in GPU arrays.

In order to make the very best of powerful GPUs, it is important to balance the load between CPU and GPU in order to avoid starving the latter. In training on a problem like ImageNet, the CPU(s) in your system will be busy loading data from disk and streaming it to the GPU to evaluate the CNN and its derivative. MatConvNet includes the utility `v1_imreadjpeg` to accelerate and parallelize loading images into memory (this function is currently a bottleneck will be made more powerful in future releases).

About MatConvNet

MatConvNet was born in the Oxford Visual Geometry Group as both an educational and research platform for fast prototyping in Convolutional Neural Nets. Its main features are:

- *Flexibility*. Neural network layers are implemented in a straightforward manner, often directly in MATLAB code, so that they are easy to modify, extend, or integrate with new ones. Other toolboxes hide the neural network layers behind a wall of compiled code; here the granularity is much finer.
- *Power*. The implementation can run large models such as Krizhevsky et al., including the DeCAF and Caffe variants. Several pre-trained models are provided.
- *Efficiency*. The implementation is quite efficient, supporting both CPU and GPU computation.

This library may be merged in the future with VLFeat library (<http://www.vlfeat.org/>). It uses a very similar style, so if you are familiar with VLFeat, you should be right at home here.

Changes

- 1.0-beta17 (December 2015).

New features

- Mac OS X 10.11 support. Since setting `LD_LIBRARY_PATH` is not supported under this OS due to security reasons, now MatConvNet binaries hardcodes the location of the CUDA/cuDNN libraries as needed. This also simplifies starting up MATLAB.
- This version changes slightly how cuDNN is configured; the cuDNN root directory is assumed to contain two subdirectories `lib` and `include` instead of the binary and include files directly. This matches how cuDNN is now distributed.
- CuDNN v4 is now supported.
- This version changes how batch normalization is handled. Now the average moments are learned together with the other parameters. The net result is that batch normalization is easy to bypass at test time (and implicitly done in validation, just like dropout).
- The `disableDropout` parameter of `vl_simplenn` has been replaced by a more generic `mode` option that allows running in either normal mode or test mode. In the latter case, both dropout and batch normalization are bypassed. This is the same behavior of `DagNN.mode`.
- Examples have been re-organized in subdirectories.
- Compiles and works correctly with cuDNN v4. However, not all v4 features are used yet.

- Adds an option to specify the maximum workspace size in the convolution routines using cuDNN.
- The AlexNet, VGG-F, VGG-M, VGG-S examples provided in the `examples/imagenet` directory have been refined in order to produce deployable models. MatConvNet pretrained versions of these models are available for download.
- A new option in `vl_nnconv` and `vl_nnconvt` allows setting the maximum amount of memory used by CuDNN to perform convolution.

Changes affecting backward compatibility

- This version changes slightly how SimpleNN networks should be handled. Use the `vl_simplenn_tidy()` to upgrade existing networks to the latest version of MatConvNet. This function is also useful to fill in missing default values for the parameters of the network layers. It is therefore recommended to use `vl_simplenn_tidy()` also when new models are defined.
 - The downloadable pre-trained models have been updated to match the new version of SimpleNN. The older models are still available for download. Note that old and new models are numerically equivalent, only the format is (slightly) different.
 - Recent versions of CuDNN may use by default a very large amount of memory for computation.
- 1.0-beta16 (October 2015). Adds VGG-Face as a pretrained model. Bugfixes.
 - 1.0-beta15 (September 2015). Supports for new DagNN blocks and import script for the FCN models. Improved `vl_nnbnorm`.
 - 1.0-beta14 (August 2015). New DagNN wrapper for networks with complex topologies. GoogLeNet support. Rewritten `vl_nnloss` block with support for more loss functions. New blocks, better documentation, bugfixes, new demos.
 - 1.0-beta13 (July 2015). Much faster batch normalization and several minor improvements and bugfixes.
 - 1.0-beta12 (May 2015). Added `vl_nnconvt` (convolution transpose or deconvolution).
 - 1.0-beta11 (April 2015) Added batch normalization, spatial normalization, sigmoid, p-distance. Extended the example training code to support multiple GPUs. Significantly improved the tuning of the ImageNet and CIFAR examples. Added the CIFAR Network in Network model.

This version changes slightly the structure of `simplenn`. In particular, the `filters` and `biases` fields in certain layers have been replaced by a `weights` cell array containing both tensors, simplifying a significant amount of code. All examples and downloadable models have been updated to reflect this change. Models using the old structure format still work but are deprecated.

The `cnn_train` training code example has been rewritten to support multiple GPUs. The interface is nearly the same, but the `useGpu` option has been replaced by a `gpus` list of GPUs to use.

- 1.0-beta10 (March 2015) `vl_imreadjpeg` works under Windows as well.
- 1.0-beta9 (February 2015) CuDNN support. Major rewrite of the C/CUDA core.
- 1.0-beta8 (December 2014) New website. Experimental Windows support.

- 1.0-beta7 (September 2014) Adds VGG very deep models.
- 1.0-beta6 (September 2014) Performance improvements.
- 1.0-beta5 (September 2014) Bugfixes, adds more documentation, improves ImageNet example.
- 1.0-beta4 (August 2014) Further cleanup.
- 1.0-beta3 (August 2014) Cleanup.
- 1.0-beta2 (July 2014) Adds a set of standard models.
- 1.0-beta1 (June 2014) First public release.

Contributors

MatConvNet is developed by several hands:

- Andrea Vedaldi, project coordinator
- Karel Lenc, DaG, several building blocks and examples
- Sébastien Ehrhardt, GPU implementation of batch normalization, FCN building blocks and examples
- Max Jaderberg, general improvements and bugfixes

MatConvNet quality also depends on the many people using the toolbox and providing us with feedback and bug reports.

Copyright

This package was originally created by Andrea Vedaldi (<http://www.robots.ox.ac.uk/~vedaldi>) and Karel Lenc and it is currently developed by a small community of contributors. It is distributed under the permissive BSD license (see also the file COPYING):

```
Copyright (c) 2014-15 The MatConvNet team.
All rights reserved.
```

```
Redistribution and use in source and binary forms are permitted
provided that the above copyright notice and this paragraph are
duplicated in all such forms and that any documentation,
advertising materials, and other materials related to such
distribution and use acknowledge that the software was developed
by the <organization>. The name of the <organization> may not be
used to endorse or promote products derived from this software
without specific prior written permission.  THIS SOFTWARE IS
PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
```

Acknowledgments

The implementation of the computational blocks in this library, and in particular of the convolution operators, is inspired by Caffe (<http://caffe.berkeleyvision.org>).

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GPUs used to develop this software.

Copyright © 2014-15 The MatConvNet Team.

Frequently-asked questions (FAQ)

Running MatConvNet

Do I need a specific version of the CUDA devkit?

Officially, MathWorks supports a specific version of the CUDA devkit with each MATLAB version (see here (`../install/#gpu`)). However, in practice we normally use the most recent version of CUDA (and cuDNN) available from NVIDIA without problems (see here (`../install/#nvcc`)).

Can I use MatConvNet with CuDNN?

Yes, and this is the recommended way of running MatConvNet on NVIDIA GPUs. However, you need to install cuDNN and link it to MatConvNet. See the installation instructions (`../install/#cudnn`) to know how.

How do I fix the error `Attempt to execute SCRIPT v1_nnconv as a function?`

Before the toolbox can be used, the MEX files (<http://www.mathworks.com/support/tech-notes/1600/1605.html>) must be compiled. Make sure to follow the installation instructions (`../install/`). If you have done so and the MEX files are still not recognized, check that the directory `matlab/toolbox/mex` contains the missing files. If the files are there, there may be a problem with the way MEX files have been compiled.

Why files such as `v1_nnconv.m` do not contain any code?

Functions such as `v1_nnconv`, `v1_nnpool`, `v1_nnbnorm` and many others are implemented MEX files. In this case, M files such as `v1_nnconv.m` contain only the function documentation. The code of the function is actually found in `matlab/src/v1_nnconv.cu` (a CUDA/C++ source file) or similar.

Function index

MatConvNet includes several MATLAB functions organized as follows:

- Building blocks. These functions implement the CNN computational blocks that can be combined either manually or using one of the provided wrappers to construct CNNs.
- SimpleCNN wrapper. SimpleNN is a lightweight wrapper implementing CNNs that are linear chains of computational blocks.
- DagNN wrapper. DagNN is an object-oriented wrapper supporting more complex network topologies.
- Other functions. These helper functions are used to initialize and compile MatConvNet.

There is no general training function as training depends on the dataset and problem. Look at the `examples` subdirectory for code showing how to train CNNs.

Building blocks

- [vl_nnbnorm](#) (./mfiles/vl_nnbnorm/) Batch normalization.
- [vl_nnconv](#) (./mfiles/vl_nnconv/) Linear convolution by a filter.
- [vl_nnconcat](#) (./mfiles/vl_nnconcat/) Concatenation.
- [vl_nnconvt](#) (./mfiles/vl_nnconvt/) Convolution transpose.
- [vl_nncrop](#) (./mfiles/vl_nncrop/) Cropping.
- [vl_nndropout](#) (./mfiles/vl_nndropout/) Dropout.
- [vl_nnloss](#) (./mfiles/vl_nnloss/) Classification log-loss.
- [vl_nnnoffset](#) (./mfiles/vl_nnnoffset/) Norm-dependent offset.
- [vl_nnnormalize](#) (./mfiles/vl_nnnormalize/) Local Response Normalization (LRN).
- [vl_nnpdist](#) (./mfiles/vl_nnpdist/) Pairwise distances.
- [vl_nnpool](#) (./mfiles/vl_nnpool/) Max and sum pooling.
- [vl_nnrelu](#) (./mfiles/vl_nnrelu/) Rectified Linear Unit.
- [vl_nnsigmoid](#) (./mfiles/vl_nnsigmoid/) Sigmoid.
- [vl_nnsoftmax](#) (./mfiles/vl_nnsoftmax/) Channel soft-max.
- [vl_nnsoftmaxloss](#) (./mfiles/vl_nnsoftmaxloss/) *Deprecated*
- [vl_nnsppnorm](#) (./mfiles/vl_nnsppnorm/) Spatial normalization.

SimpleCNN wrapper

- [vl_simplenn](#) (./mfiles/simplenn/vl_simplenn/) A lightweight wrapper for CNNs with a linear topology.
- [vl_simplenn_tidy](#) (./mfiles/simplenn/vl_simplenn_tidy/) Upgrade or otherwise fix a CNN.
- [vi_simplenn_display](#) (./mfiles/simplenn/vl_simplenn_display/) Print information about the CNN architecture.
- [vl_simplenn_move](#) (./mfiles/simplenn/vl_simplenn_move/) Move the CNN between CPU and GPU.

DagNN wrapper

- DagNN (../mfiles/+dagnn/@DagNN/DagNN/) An object-oriented wrapper for CNN with complex topologies

Other functions

- vl_argparse (../mfiles/vl_argparse/) A helper function to parse optional arguments.
- vl_compilenn (../mfiles/vl_compilenn/) Compile the MEX files in the toolbox.
- vl_rootnn (../mfiles/vl_rootnn/) Return the path to the MatConvNet toolbox installation.
- vl_setupnn (../mfiles/vl_setupnn/) Setup MatConvNet for use in MATLAB.
- vl_imreadjpeg (../mfiles/vl_imreadjpeg/) Quickly load a batch of JPEG images.