MatConvNet

Convolutional Neural Networks for MATLAB

SUBMITTED to ACM MULTIMEDIA 2015 OPEN SOURCE SOFTWARE COMPETITION

Andrea Vedaldi
Univeristy of Oxford
vedaldi@robots.ox.ac.uk

Karel Lenc
Univeristy of Oxford
lenc@robots.ox.ac.uk

ABSTRACT

MatConvNet is an open source implementation of Convolutional Neural Networks (CNNs) with a deep integration in the MATLAB environment. The toolbox is designed with an emphasis on simplicity and flexibility. It exposes the building blocks of CNNs as easy-to-use MATLAB functions, providing routines for computing convolutions with filter banks, feature pooling, normalisation, and much more. MatConvNet can be easily extended, often using only MATLAB code, allowing fast prototyping of new CNN architectures. At the same time, it supports efficient computation on CPU and GPU, allowing to train complex models on large datasets such as ImageNet ILSVRC containing millions of training examples.

Categories and Subject Descriptors

D.0 [**Software**]: General; I.2.10 [**Artificial Intelligence**]: Vision and Scene Understanding

General Terms

Algorithm, design, prototyping

Keywords

Computer vision, image understanding, machine learning, deep learning, convolutional neural networks

1. INTRODUCTION

MatConvNet is a MATLAB toolbox implementing *Convolutional Neural Networks* (CNN) for computer vision applications. Since the breakthrough work of [4], CNNs have had a major impact in computer vision, and image understanding in particular, essentially replacing traditional image representations such as the ones implemented in our own VLFeat [8] open source library.

While most CNNs are obtained by composing simple linear and non-linear filtering operations such as convolution and rectification, their implementation is far from trivial. The reason is that CNNs need to be learned from vast amounts of data, often millions of images, requiring very efficient implementations. As most CNN libraries, MatConvNet achieves this by using a variety of optimisations and, chiefly, by supporting computations on GPUs.

Numerous other machine learning, deep learning, and CNN open source libraries exist. To cite some of the most popular ones: CudaConvNet,[1] Torch,[2] Theano,[3] and Caffe[4]. Many of these libraries are well supported, with dozens of active contributors and large user bases. Therefore, why creating yet another library?

MM'15, October 26–30, 2015, Brisbane, Australia.

ACM 978-1-4503-3459-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2733373.2807412.

```
% install and compile MatConvNet (run once)
untar(['http://www.vlfeat.org/matconvnet/download/' ...
    'matconvnet-1.0-beta12.tar.gz']) ;
cd matconvnet-1.0-beta12
run matlab/vl_compilenn

% download a pre-trained CNN from the web (run 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_ = imresize(single(im), net.normalization.imageSize(1:2)) ;
im_ = im_ - net.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.classes.description{best}, best, bestScore)) ;
```

Figure 1: A complete example including download, installing, compiling and running MatConvNet to classify one of MATLAB stock images using a large CNN pre-trained on ImageNet.

[1] https://code.google.com/p/cuda-convnet/
[2] http://cilvr.nyu.edu/doku.php?id=code:start
[3] http://deeplearning.net/software/theano/
[4] http://caffe.berkeleyvision.org

The key motivation for developing MatConvNet was to provide an environment particularly friendly and efficient for researchers to use in their investigations.[5] MatConvNet achieves this by its deep integration in the MATLAB environment, which is one of the most popular development environments in computer vision research as well as in many other areas. In particular, MatConvNet exposes as simple MATLAB commands CNN building blocks such as convolution, normalisation and pooling (section 2); these can then be combined and extended with ease to create CNN architectures. While many of such blocks use optimised CPU and GPU implementations written in C++ and CUDA (section 4), MATLAB native support for GPU computation means that it is often possible to write new blocks in MATLAB directly while maintaining computational efficiency. Compared to writing new CNN components using lower level languages, this is an important simplification that can significantly accelerate testing new ideas. Using MATLAB also provides a bridge towards other areas; for instance, MatConvNet was recently used by the University of Arizona in planetary science, as summarised in this NVIDIA blogpost.[6]

MatConvNet can learn large CNN models such AlexNet [4] and the very deep networks of [6] from millions of images. Pre-trained versions of several of these powerful models can be downloaded from the MatConvNet home page. (section 3). While powerful, MatConvNet remains simple to use and install. The implementation is fully self-contained, requiring only MATLAB and a compatible C++ compiler (using the GPU code requires the freely-available CUDA DevKit and a suitable NVIDIA GPU). As demonstrated in figure 1 and section 1.1, it is possible to download, compile, and install MatConvNet using three MATLAB commands. Several fully-functional examples demonstrating how small and large networks can be learned are included. Importantly, several *standard pre-trained network* can be immediately downloaded and used in applications. A manual with a complete technical description of the toolbox is maintained along with the toolbox.[7] These features make MatConvNet useful in an educational context too.[8]

MatConvNet is open-source released under a BSD-like license. It can be downloaded from `http://www.vlfeat.org/matconvnet` as well as from GitHub.[9]

1.1 Getting started

MatConvNet is simple to install and use. Figure 1 provides a complete example that classifies an image using a latest-generation deep convolutional neural network. The example includes downloading MatConvNet, compiling the package, downloading a pre-trained CNN model, and evaluating the latter on one of MATLAB's stock images.

[5] While from a user perspective MatConvNet currently relies on MATLAB, the library is being developed with a clean separation between MATLAB code and the C++ and CUDA core; therefore, in the future the library may be extended to allow processing convolutional networks independently of MATLAB.

[6] `http://devblogs.nvidia.com/parallelforall/deep-learning-image-understanding-planetary-science/`

[7] `http://www.vlfeat.org/matconvnet/matconvnet-manual.pdf`

[8] An example laboratory experience based on MatConvNet can be downloaded from `http://www.robots.ox.ac.uk/~vgg/practicals/cnn/index.html`.

[9] `http://ww.github.com/matconvnet`

The key command in this example is `vl_simplenn`, a wrapper that takes as input the CNN `net` and the pre-processed image `im_` and produces as output a structure `res` of results. This particular wrapper can be used to model networks that have a simple structure, namely a *chain* of operations. Examining the code of `vl_simplenn` (`edit` ↩ `vl_simplenn` in MATLAB) we note that the wrapper transforms the data sequentially, applying a number of MATLAB functions as specified by the network configuration. These function, discussed in detail in section 2, are called "building blocks" and constitute the backbone of MatConvNet.

While most blocks implement simple operations, what makes them non trivial is their efficiency (section 4) as well as support for backpropagation (section 2.2) to allow learning CNNs. Next, we demonstrate how to use one of such building blocks directly. For the sake of the example, consider convolving an image with a bank of linear filters. Start by reading an image in MATLAB, say using `im` ↩ `= single(imread('peppers.png'))`, obtaining a $H \times W \times D$ array `im`, where $D = 3$ is the number of colour channels in the image. Then create a bank of $K = 16$ random filters of size 3×3 using `f = randn(3,3,3,16,'single')`. Finally, convolve the image with the filters by using the command `y = vl_nnconv(x,f,[])`. This results in an array `y` with K channels, one for each of the K filters in the bank.

While users are encouraged to make use of the blocks directly to create new architectures, MatConvNet provides wrappers such as `vl_simplenn` for standard CNN architectures such as AlexNet [4] or Network-in-Network [5]. Furthermore, the library provides numerous examples (in the `examples/` subdirectory), including code to learn a variety models on the MNIST, CIFAR, and ImageNet datasets. All these examples use the `examples/cnn_train` training code, which is an implementation of stochastic gradient descent (section 2.1). While this training code is perfectly serviceable and quite flexible, it remains in the `examples/` subdirectory as it is somewhat problem-specific. Users are welcome to implement their optimisers.

2. BUILDING BLOCKS

At the core of MatConvNet there is a library of CNN building blocks, such as the convolution operator `vl_nnconv` seen above. This section discusses in detail these building blocks and their design.

2.1 Overview of the available blocks

In order to understand the design of the building blocks, it is necessary to first review the fundamentals of CNNs. On the outset, a *Convolutional Neural Network* (CNN) is a function f mapping data $\mathbf{x}$, for example an image, to an output vector $\mathbf{y}$. The function $f = f_L \circ \cdots \circ f_1$ is the composition of a sequence of simpler functions f_l, which we call *computational blocks* or *layers*. Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L$ be the outputs of each layer in the network, and let $\mathbf{x}_0 = \mathbf{x}$ denote the network input. Each output $\mathbf{x}_l = f_l(\mathbf{x}_{l-1}; \mathbf{w}_l)$ is computed from the previous output $\mathbf{x}_{l-1}$ by applying the function f_l with parameters $\mathbf{w}_l$. The data flowing through the network has a spatial structure; namely, $\mathbf{x}_l \in \mathbb{R}^{H_l \times W_l \times D_l}$ is a 3D array whose first two dimensions are interpreted as spatial coordinates (it therefore represents a feature field). A fourth non-singleton dimension in the array allows processing *batches* of images in parallel, which is important for efficiency. The network is called *convolutional* because the

functions f_l act as local and translation invariant operators (i.e. non-linear filters).

MatConvNet includes a variety of building blocks, contained in the `matlab/` directory, such as `vl_nnconv` (convolution), `vl_nnconvt` (convolution transpose or deconvolution), `vl_nnpool` (max and average pooling), `vl_nnrelu` (ReLU activation), `vl_nnsigmoid` (sigmoid activation), `vl_nnsoftmax` (softmax operator), `vl_nnloss` (classification log-loss), `vl_nnbnorm` (batch normalization), `vl_nnspnorm` (spatial normalization), `vl_nnnormalize` (cross-channel normalization), or `vl_nnpdist` (p-distance). The library of blocks is sufficiently extensive that many interesting state-of-the-art network can be implemented and learned using the toolbox, or even ported from other toolboxes such as Caffe.

CNNs are used as classifiers or regressors. In the example of figure 1, the output $\hat{\mathbf{y}} = f(\mathbf{x})$ is a vector of probabilities, one for each of a 1,000 possible image labels (dog, cat, trilobite, ...). If $\mathbf{y}$ is the true label of image $\mathbf{x}$, we can measure the CNN performance by a loss function $\ell_{\mathbf{y}}(\hat{\mathbf{y}}) \in \mathbb{R}$ which assigns a penalty to classification errors. The CNN parameters can then be tuned or *learned* to minimise this loss averaged over a large dataset of labelled example images.

Learning generally uses a variant of *stochastic gradient descent* (SGD). While this is an efficient method (for this type of problems), networks may contain several million parameters and need to be trained on millions of images; thus, efficiency is a paramount in MatConvNet design, as further discussed in section 4. SGD requires also to compute the CNN derivatives, as explained in the next section.

2.2 Backpropagation

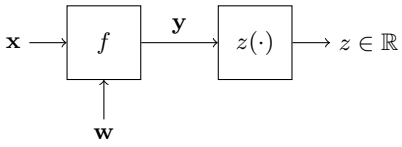
The fundamental operation to learn a network is computing the derivative of the loss with respect to the network parameters (as this is required for gradient descent). This is obtained using an algorithm called *backpropagation*, which is an application of the chain rule for derivatives:

$$\frac{d}{d\mathbf{w}_l^\top} \ell_{\mathbf{y}}(f(\mathbf{x}; \mathbf{w}_1, \dots, \mathbf{w}_L)) = \frac{d[\ell_{\mathbf{y}} \circ f_L \circ \dots f_{l+1}](\mathbf{x}_l)}{d\mathbf{x}_l^\top} \frac{df_l(\mathbf{x}_{l-1}; \mathbf{w}_l)}{d\mathbf{w}_l^\top}$$

where for notational simplicity that data and parameters are identified with vectors (which is always possible up to stacking). Note that the formula involves computing the derivative of parts of the network with respect to the data; these are obtained in a similar manner, and require computing the derivative "one level up". Overall derivatives are computed by backtracking from the last layer (output) to the first (input).

Since the loss function output is scalar, the dimension of all the intermediate derivatives is the same as the corresponding parameter. For example, $d[\ell_{\mathbf{y}} \circ f_L \circ \dots f_{l+1}]/d\mathbf{x}_l^\top$ has $H_l W_l D_l$ components, equal to the number of elements of $\mathbf{x}_l$. Compare this to a Jacobian such as $df_l/d\mathbf{x}_{l-1}^\top$ that has $H_l W_l D_l H_{l-1} W_{l-1} D_{l-1}$ components instead.

The key in implementing backpropagation efficiently is to store only the smaller derivatives, leaving the intermediate calculations of the larger Jacobians implicit. This is best illustrated with an example. Consider a layer f such as the convolution operator implemented by `vl_nnconv`. In the so called "forward" mode, one calls the function as `y` $\hookleftarrow$ `vl_nnconv(x,w,[])` to convolve input `x` and obtain output `y`. In the "backward mode", one calls `[dzdx, dzdw]` $\hookleftarrow$ `vl_nnconv(x,w,[],dzdy)`. To understand this syntax, imagine that f is connected to the "rest of the network", denoted $z(\cdot)$, terminating in a scalar loss:



Here `dzdy` is the derivative $dz/d\mathbf{y}$ of the downstream subnetwork $z(\cdot)$, `dzdx` the derivative $d[z \circ f]/d\mathbf{x}$ of the upstream subnetwork (i.e. prefixed by f), and `dzdx` and `dzdw` the corresponding derivatives w.r.t. `x` and `w`.

As explained above, `dzdx`, `dzdw`, and `dzdy` have the same dimension of `x`, `w`, and `y`. In this manner, the computation of larger Jacobians is encapsulated in the function call and never carried explicitly. Another way of looking at this is that, instead of computing a derivative such as $d\mathbf{y}/d\mathbf{w}$, one always computes a projection of the type $\langle dz/d\mathbf{y}, d\mathbf{y}/d\mathbf{w} \rangle$.

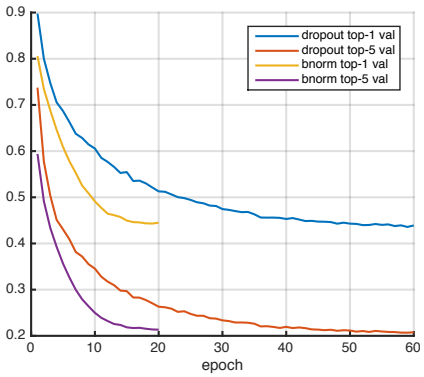


Figure 2: Training AlexNet on ImageNet ILSVRC: dropout vs batch normalisation.

3. DOCUMENTATION AND EXAMPLES

There are three main sources of information about MatConvNet. First, the website contains descriptions of all the functions and several examples and tutorials.[10] Second, there is a PDF manual containing a great deal of technical details about the toolbox, including detailed mathematical descriptions of the building blocks. Third, MatConvNet ships with several examples (section 1.1).

Most examples are fully self-contained. For example, in order to run the MNIST example, it suffices to point MATLAB to the MatConvNet root directory and type `addpath examples` followed by `cnn_mnist`. Due to the problem size, the ImageNet ILSVRC example requires some more preparation, including downloading and preprocessing the images (using the bundled script `utils/preprocess-imagenet.sh`). Several advanced examples are included as well. For example, figure 2 illustrates the top-1 and top-5 validation errors as a model similar to AlexNet [4] is trained using either standard dropout regularisation or the recent *batch normalisation* technique of [3]. The latter is shown to converge in about one third of the

[10] See also http://www.robots.ox.ac.uk/~vgg/practicals/cnn/index.html.

epochs (passes through the training data) required by the former.

The MatConvNet website contains also numerous *pre-trained* models, i.e. large CNNs trained on ImageNet ILSVRC that can be downloaded and used as a starting point for many other problems [1]. These include: AlexNet [4], VGG-S, VGG-M, VGG-S [1], and VGG-VD-16, and VGG-VD-19 [7]. The example code of figure 1 shows how one such models can be used in a few lines of MATLAB code.

4. SPEED

Efficiency is very important for working with CNNs. MatConvNet supports using NVIDIA GPUs as it includes CUDA implementations of all algorithms (or relies on MATLAB CUDA support).

To use the GPU (provided that suitable hardware is available and the toolbox has been compiled with GPU support), one simply converts the arguments to `gpuArrays` in MATLAB, as in `y = vl_nnconv(gpuArray(x), gpuArray(w), [])`. In this manner, switching between CPU and GPU is fully transparent. Note that MatConvNet can also make use of the NVIDIA CuDNN library which significant speed and space benefits.

Next we evaluate the performance of MatConvNet when training large architectures on the ImageNet ILSVRC 2012 challenge data [2]. The test machine is a Dell server with two Intel Xeon CPU E5-2667 v2 clocked at 3.30 GHz (each CPU has eight cores), 256 GB of RAM, and four NVIDIA Titan Black GPUs (only one of which is used unless otherwise noted). Experiments use MatConvNet beta12, CuDNN v2, and MATLAB R2015a. The data is preprocessed to avoid rescaling images on the fly in MATLAB and stored in a RAM disk for faster access. The code uses the `vl_imreadjpeg` command to read large batches of JPEG images from disk in a number of separate threads. The driver `examples/cnn_imagenet.m` is used in all experiments.

We train the models discussed in section 3 on ImageNet ILSVRC. Table 1 reports the training speed as number of images per second processed by stochastic gradient descent. AlexNet trains at about 264 images/s with CuDNN, which is about 40% faster than the vanilla GPU implementation (using CuBLAS) and more than 10 times faster than using the CPUs. Furthermore, we note that, despite MATLAB overhead, the implementation speed is comparable to Caffe (they report 253 images/s with cuDNN and a Titan – a slightly slower GPU than the Titan Black used here). Note also that, as the model grows in size, the size of a SGD batch must be decreased (to fit in the GPU memory), increasing the overhead impact somewhat.

Table 2 reports the speed on VGG-VD-16, a very large model, using multiple GPUs. In this case, the batch size is set to 264 images. These are further divided in sub-batches of 22 images each to fit in the GPU memory; the latter are then distributed among one to four GPUs on the same machine. While there is a substantial communication overhead, training speed increases from 20 images/s to 45. Addressing this overhead is one of the medium term goals of the library.

model	batch sz.	CPU	GPU	CuDNN
AlexNet	256	22.1	192.4	264.1
VGG-F	256	21.4	211.4	289.7
VGG-M	128	7.8	116.5	136.6
VGG-S	128	7.4	96.2	110.1
VGG-VD-16	24	1.7	18.4	20.0
VGG-VD-19	24	1.5	15.7	16.5

Table 1: ImageNet training speed (images/s).

num GPUs	1	2	3	4
VGG-16 speed	20.0	22.20	38.18	44.8

Table 2: Multiple GPU speed (images/s).

5. CONCLUSIONS

MatConvNet is a novel framework for experimenting with deep convolutional networks that is deeply integrated in MATLAB and allows easy experimentation with novel ideas. MatConvNet is already sufficient for advanced research in deep learning; despite being introduced less than a year ago, it is already citied 24 times in arXiv papers, and has been used in several papers published at the recent CVPR 2015 conference.

As CNNs are a rapidly moving target, MatConvNet is developing fast. So far there have been 12 ad-interim releases incrementally adding new features to the toolbox. Several new features, including support for DAGs, will be included in the upcoming releases starting in August 2015. The goal is to ensure that MatConvNet will stay current for the next several years of research in deep learning.

Acknowledgments. We kindly thank NVIDIA supporting this project by providing us with top-of-the-line GPUs and Mathworks for ongoing discussion on how to improve the library. Andrea Vedaldi is partially supported by the ERC StG 196773 and Karel Lenc by a DTA of the University of Oxford.

6. REFERENCES

[1] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *Proc. BMVC*, 2014.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. CVPR*, 2009.

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, 2015.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS*, 2012.

[5] M. Lin, Q. Chen, and S. Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

[6] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. ICLR*, 2014.

[7] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[8] A. Vedaldi and B. Fulkerson. VLFeat – An open and portable library of computer vision algorithms. In *Proc. ACM Int. Conf. on Multimedia*, 2010.