MatConvNet
Convolutional Neural Networks for MATLAB

Andrea Vedaldi Karel Lenc

Abstract

MatConvNet is an implementation of Convolutional Neural Networks (CNNs) for MATLAB. 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 linear convolutions with filter banks, feature pooling, and many more. In this manner, MatConvNet allows 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. This document provides an overview of CNNs and how they are implemented in MatConvNet and gives the technical details of each computational block in the toolbox.

Contents

1 Introduction

MatConvNet is a simple MATLAB toolbox implementing Convolutional Neural Networks (CNN) for computer vision applications. This documents starts with a short overview of CNNs and how they are implemented in MatConvNet. Section 2 lists all the computational building blocks implemented in MatConvNet that can be combined to create CNNs and gives the technical details of each one. Finally, Section 4 discusses more abstract CNN wrappers and example code and models.

A *Convolutional Neural Network* (CNN) can be viewed as a function f mapping data $\mathbf{x}$, for example an image, on an output vector $\mathbf{y}$. The function f is a composition of a sequence (or a directed acyclic graph) of simpler functions $f_1, \dots, f_L$, also called *computational blocks* in this document. Furthermore, these blocks are *convolutional*, in the sense that they map an input image of feature map to an output feature map by applying a translation-invariant and local operator, e.g. a linear filter. The MatConvNet toolbox contains implementation for the most commonly used computational blocks (described in Section 2) which can be used either directly, or through simple wrappers. Thanks to the modular structure, it is a simple task to create and combine new blocks with the existing ones.

Blocks in the CNN usually contain parameters $\mathbf{w}_1, \dots, \mathbf{w}_L$. These are *discriminatively learned from example data* such that the resulting function f realizes an useful mapping. A typical example is image classification; in this case the output of the CNN is a vector $\mathbf{y} = f(\mathbf{x}) \in \mathbb{R}^C$ containing the confidence that $\mathbf{x}$ belong to any of C possible classes. Given training data $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ (where $\mathbf{y}^{(i)}$ is the indicator vector of the class of $\mathbf{x}^{(i)}$), the parameters are learned by solving

$$\operatorname*{argmin}_{\mathbf{w}_1,\dots\mathbf{w}_n} \frac{1}{n} \sum_{i=1}^{n} \ell\left(f(\mathbf{x}^{(i)}; \mathbf{w}_1, \dots, \mathbf{w}_L), \mathbf{y}^{(i)}\right) \tag{1}$$

where ℓ is a suitable *loss function* (e.g. the hinge or log loss).

The optimization problem (1) is usually non-convex and very large as complex CNN architectures need to be trained from hundred-thousands or even millions of examples. Therefore efficiency is a paramount. Optimization often uses a variant of *stochastic gradient descent*. The algorithm is, conceptually, very simple: at each iteration a training point is selected at random, the derivative of the loss term for that training sample is computed resulting in a gradient vector, and parameters are incrementally updated by moving towards the local minima in the direction of the gradient. The key operation here is to compute the derivative of the objective function, which is obtained by an application of the chain rule known as *back-propagation*. MatConvNet can evaluate the derivatives of all the computational blocks. It also contains several examples of training small and large models using these capabilities and a default solver, although it is easy to write customized solvers on top of the library.

While CNNs are relatively efficient to compute, training requires iterating many times through vast data collections. Therefore the computation speed is very important in practice. Larger models, in particular, may require the use of GPU to be trained in a reasonable time. MatConvNet has integrated GPU support based on NVIDIA CUDA and MATLAB built-in CUDA capabilities.

1.1 MatConvNet on a glance

MatConvNet has a simple design philosophy. Rather than wrapping CNNs around complex layers of software, it exposes simple functions to compute CNN building blocks, such as linear convolution and ReLU operators. These building blocks are easy to combine into a complete CNNs and can be used to implement sophisticated learning algorithms. While several real-world examples of small and large CNN architectures and training routines are provided, it is always possible to go back to the basics and build your own, using the efficiency of MATLAB in prototyping. Often no C coding is required at all to try a new architectures. As such, MatConvNet is an ideal playground for research in computer vision and CNNs.

MatConvNet contains the following elements:

- *CNN computational blocks.* A set of optimized routines computing fundamental building blocks of a CNN. For example, a convolution block is implemented by `y=vl_nnconv(x,f,b)` where `x` is an image, `f` a filter bank, and `b` a vector of biases (Section 2.1). The derivatives are computed as `[dzdx,dzdf,dzdb] = vl_nnconv(x,f,b,dzdy)` where `dzdy` is the derivative of the CNN output w.r.t `y` (Section 2.1). Section 2 describes all the blocks in detail.

- *CNN wrappers.* MatConvNet provides a simple wrapper, suitably invoked by `vl_simplenn`, that implements a CNN with a linear topology (a chain of blocks). This is good enough to run most of current state-of-the-art models for image classification. You are invited to look at the implementation of this function, as it is a great starting point to understand how to implement more complex CNNs.

- *Example applications.* MatConvNet provides several example of learning CNNs with stochastic gradient descent and CPU or GPU, on MNIST, CIFAR10, and ImageNet data.

- *Pre-trained models.* MatConvNet provides several state-of-the-art pre-trained CNN models that can be used off-the-shelf, either to classify images or to produce image encodings in the spirit of Caffe or DeCAF.

1.2 The structure and evaluation of CNNs

CNNs are obtained by connecting one or more *computational blocks*. Each block $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$ takes an image $\mathbf{x}$ and a set of parameters $\mathbf{w}$ as input and produces a new image $\mathbf{y}$ as output. An image is a real 4D array; the first two dimensions index spatial coordinates (image rows and columns respectively), the third dimension feature channels (there can be any number), and the last dimension image instances. A computational block f is therefore represented as follows:

$$\begin{tikzcd}
\mathbf{x} \arrow[r] & \boxed{f} \arrow[r] & \mathbf{y} \\
& \mathbf{w} \arrow[u] &
\end{tikzcd}$$

Formally, $\mathbf{x}$ is a 4D tensor stacking N 3D images

$$\mathbf{x} \in \mathbb{R}^{H \times W \times D \times N}$$

where H and W are the height and width of the images, D its depth, and N the number of images. In what follows, all operations are applied identically to each image in the stack $\mathbf{x}$; hence for simplicity we will drop the last dimension in the discussion (equivalent to assuming $N = 1$), but the ability to operate on image batches is very important for efficiency.

In general, a CNN can be obtained by connecting blocks in a directed acyclic graph (DAG). In the simplest case, this graph reduces to a sequence of computational blocks $(f_1, f_2, \dots, f_L)$.

Let $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_L$ be the output of each layer in the network, and let $\mathbf{x}_0$ denote the network input. Each output $\mathbf{x}_l$ depends on the previous output $\mathbf{x}_{l-1}$ through a function f_l with parameter $\mathbf{w}_l$ as $\mathbf{x}_l = f_l(\mathbf{x}_{l-1}; \mathbf{w}_l)$; schematically:

$$\begin{tikzcd}
\mathbf{x}_0 \arrow[r] & \boxed{f_1} \arrow[r, "\mathbf{x}_2"] & \boxed{f_2} \arrow[r, "\mathbf{x}_3"] & \cdots \arrow[r, "\mathbf{x}_{L-1}"] & \boxed{f_L} \arrow[r] & \mathbf{x}_L \\
& \mathbf{w}_1 \arrow[u] & \mathbf{w}_2 \arrow[u] & & \mathbf{w}_L \arrow[u] &
\end{tikzcd}$$

Given an input $\mathbf{x}_0$, evaluating the network is a simple matter of evaluating all the intermediate stages in order to compute an overall function $\mathbf{x}_L = f(\mathbf{x}_0; \mathbf{w}_1, \dots, \mathbf{w}_L)$.

1.3 CNN derivatives

In training a CNN, we are often interested in taking the derivative of a loss $\ell : f(\mathbf{x}, \mathbf{w}) \mapsto \mathbb{R}$ with respect to the parameters. This effectively amounts to extending the network with a *scalar block* at the end:

$$\begin{tikzcd}
\mathbf{x}_0 \arrow[r] & \boxed{f_1} \arrow[r, "\mathbf{x}_2"] & \boxed{f_2} \arrow[r, "\mathbf{x}_3"] & \cdots \arrow[r, "\mathbf{x}_{L-1}"] & \boxed{f_L} \arrow[r, "\mathbf{x}_L"] & \boxed{\ell} \arrow[r] & z \in \mathbb{R} \\
& \mathbf{w}_1 \arrow[u] & \mathbf{w}_2 \arrow[u] & & \mathbf{w}_L \arrow[u] & &
\end{tikzcd}$$

The derivative of $\ell \circ f$ with respect to the parameters can be computed but starting from the end of the chain (or DAG) and working backwards using the chain rule, a process also known as back-propagation. For example the derivative w.r.t. $\mathbf{w}_l$ is:

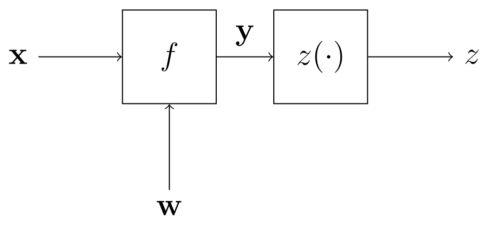
$$\frac{dz}{d(\operatorname{vec}\mathbf{w}_l)^\top} = \frac{dz}{d(\operatorname{vec}\mathbf{x}_L)^\top} \frac{d\operatorname{vec}\mathbf{x}_L}{d(\operatorname{vec}\mathbf{x}_{L-1})^\top} \cdots \frac{d\operatorname{vec}\mathbf{x}_{l+1}}{d(\operatorname{vec}\mathbf{x}_l)^\top} \frac{d\operatorname{vec}\mathbf{x}_l}{d(\operatorname{vec}\mathbf{w}_l)^\top}. \tag{2}$$

Note that the derivatives are implicitly evaluated at the working point determined by the input $\mathbf{x}_0$ during the evaluation of the network in the forward pass. The vec symbol is the vectorization operator, which simply reshape its tensor argument to a column vector. This notation for the derivatives is taken from [6] and is used throughout this document.

Computing (2) requires computing the derivative of each block $\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \mathbf{w}_l)$ with respect to its parameters $\mathbf{w}_l$ and input $\mathbf{x}_{l-1}$. Let us know focus on computing the derivatives for one computational block. We can look at the network as follows:

$$\underbrace{\ell \circ f_L(\cdot, \mathbf{w}_L) \circ f_{L-1}(\cdot, \mathbf{w}_{L-1}) \cdots \circ f_{l+1}(\cdot, \mathbf{w}_{l+1})}_{z(\cdot)} \circ f_l(\mathbf{x}_l, \mathbf{w}_l) \circ \dots$$

where $\circ$ denotes the composition of function. For simplicity, lump together the factors from $f_l + 1$ to the loss ℓ into a single scalar function $z(\cdot)$ and drop the subscript l from the first block. Hence, the problem is to compute the derivative of $(z \circ f)(\mathbf{x}, \mathbf{w}) \in \mathbb{R}$ with respect to the data $\mathbf{x}$ and the parameters $\mathbf{w}$. Graphically:



The derivative of $z \circ f$ with respect to $\mathbf{x}$ and $\mathbf{w}$ are given by:

$$\frac{dz}{d(\operatorname{vec}\mathbf{x})^\top} = \frac{dz}{d(\operatorname{vec}\mathbf{y})^\top}\frac{d\operatorname{vec}f}{d(\operatorname{vec}\mathbf{x})^\top}, \quad \frac{dz}{d(\operatorname{vec}\mathbf{w})^\top} = \frac{dz}{d(\operatorname{vec}\mathbf{y})^\top}\frac{d\operatorname{vec}f}{d(\operatorname{vec}\mathbf{w})^\top},$$

We note two facts. The first one is that, since z is a scalar function, the derivatives have a number of elements equal to the number of parameters. So in particular $dz/d\operatorname{vec}\mathbf{x}^\top$ can be reshaped into an array $dz/d\mathbf{x}$ with the same shape of $\mathbf{x}$, and the same applies to the derivatives $dz/d\mathbf{y}$ and $dz/d\mathbf{w}$. Beyond the notational convenience, this means that storage for the derivatives is not larger than the storage required for the model parameters and forward evaluation.

The second fact is that computing $dz/d\mathbf{x}$ and $dz/d\mathbf{w}$ requires the derivative $dz/d\mathbf{y}$. The latter can be obtained by applying this calculation recursively to the next block in the chain.

1.4 CNN modularity

Sections 1.2 and 1.3 suggests a modular programming interface for the implementation of CNN modules. Abstractly, we need two functionalities:

- **Forward messages:** Evaluation of the output $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$ given input data $\mathbf{x}$ and parameters $\mathbf{w}$ (forward message).
- **Backward messages:** Evaluation of the CNN derivative $dz/d\mathbf{x}$ and $dz/d\mathbf{w}$ with respect to the block input data $\mathbf{x}$ and parameters $\mathbf{w}$ given the block input data $\mathbf{x}$ and paramters $\mathbf{w}$ as well as the CNN derivative $dx/d\mathbf{y}$ with respect to the block output data $\mathbf{y}$.

1.5 Working with DAGs

CNN can also be obtained from more complex composition of functions forming a DAG. There are $n+1$ variables $\mathbf{x}_i$ and n functions f_i with corresponding arguments $\mathbf{r}_{ik}$:

$$\begin{aligned}
&\mathbf{x}_0 \\
\mathbf{x}_1 &= f_1(\mathbf{r}_{1,1}, \dots, \mathbf{r}_{1,m_1}), \\
\mathbf{x}_2 &= f_2(\mathbf{r}_{2,1}, \dots, \mathbf{r}_{2,m_2}), \\
&\vdots \\
\mathbf{x}_n &= f_n(\mathbf{r}_{n,1}, \dots, \mathbf{r}_{n,m_n})
\end{aligned}$$

Variables are connected to arguments by relations

$$\mathbf{r}_{ik} = \mathbf{x}_{\pi_{ik}}$$

where π_{ik} denotes the variable $\mathbf{x}_{\pi_{ik}}$ that feeds argument $\mathbf{r}_{ik}$. Together with the implicit fact that each argument $\mathbf{r}_{ik}$ feeds into the variable $\mathbf{x}_i$ through function f_i, this defines a bipartite DAG representing the dependencies between variables and argument. The DAG is bipartite. This DAG must be acyclic, hence assuring that, given the value of the input $\mathbf{x}_0$, all other variables can be evaluated iteratively. Due to this property, without loss of generality we will assume that variables $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_n$ are sorted such that $\mathbf{x}_i$ depends only on variables that come before in the order (i.e. one always has $\pi_{ik} < i$).

Now assume that $\mathbf{x}_n = z$ is a scalar network output (usually the learning loss). As before, we are interested in computing derivatives of this function with respect to the network variables. In order to do so, write the output of the DAG $z(\mathbf{x}_i)$ as a function of the variable $\mathbf{x}_i$; this has the following interpretation: the DAG is modified by removing function f_i and making $\mathbf{x}_i$ an input, setting $\mathbf{x}_i$ to the specified value, setting all the other inputs to the working point at which the derivative should be computed, and evaluating the resulting DAG. Likewise, one can define functions $z(\mathbf{r}_{ij})$ for the arguments. The derivative with respect to $\mathbf{x}_j$ can then be expressed as

$$\frac{dz}{d(\operatorname{vec}\mathbf{x}_j)^\top} = \sum_{(i,k):\pi_{ik}=j} \frac{dz}{d(\operatorname{vec}\mathbf{x}_i)^\top}\frac{d\operatorname{vec}\mathbf{x}_i}{d(\operatorname{vec}\mathbf{r}_{ik})^\top}\frac{d\operatorname{vec}\mathbf{r}_{ik}}{d(\operatorname{vec}\mathbf{x}_j)^\top} = \sum_{(i,k):\pi_{ik}=j} \frac{dz}{d(\operatorname{vec}\mathbf{x}_i)^\top}\frac{d\operatorname{vec} f_i}{d(\operatorname{vec}\mathbf{r}_{ik})^\top}.$$

Note that these quantities can be computer recursively, backward from the output z. In this case, each variable node $\mathbf{x}_i$ (corresponding to f_i) stores the derivative $dz/d\mathbf{x}_i$. When node $\mathbf{x}_i$ is processed, the derivatives $d\operatorname{vec} f_i/d(\operatorname{vec}\mathbf{r}_{ik})^\top$ are computed for the m_i arguments of the corresponding function f_i pre-multiplied by the factor $dz/d\mathbf{x}_i$ (available at that node) and then accumulated to the corresponding parent variables $\mathbf{x}_j$'s derivatives $dz/d\mathbf{x}_j$.

2 Computational blocks

This section describes the individual computational block supported by the MatConvNet. The interface of a CNN computational block follows Section 1.4. The block can be evaluated as a MATLAB function `y = vl_nn<block>(x,w)` that takes as input arrays `x` and `w` representing the input data and parameters of the block and returns an array `y` as output. `x` and `y` are 4D real arrays packing N maps or images, as discussed above, whereas `w` may have an arbitrary shape.

In order to compute the block derivatives, the same function can take a third optional argument `dzdy` representing the derivative of the output of the network with respect to `y` and returns the corresponding derivatives `[dzdx,dzdw] = vl_nn<block>(x,w,dzdy)`. `dzdx`, `dzdy` and `dzdw` are array with the same dimension of `x`, `y` and `w` respectively, as discussed in Section 1.3.

A function syntax may differ slightly depending on the specifics of a block. For example, a function can take additional optional arguments, specified as a property-value list; it can take no parameters (e.g. a rectified linear unit), in which case `w` is omitted; it can take multiple inputs and parameters, in which there may be more than one `x`, `w`, `dzdx`, `dzdy` or `dzdw`. See the MATLAB inline help of each function for details on the syntax.[1]

[1]In some cases it may be convenient to wrap these functions to obtain completely uniform and abstract

The rest of the section describes the blocks implemented in MatConvNet. The purpose is to describe the blocks analytically; refer to MATLAB inline help for further details on the API.

2.1 Convolution

The convolutional block is implemented by the function `vl_nnconv`. `y=vl_nnconv(x,f,b)` computes the convolution of the input map $\mathbf{x}$ with a bank of K multi-dimensional filters $\mathbf{f}$ and biases b. Here

$$\mathbf{x} \in \mathbb{R}^{H\times W\times D}, \quad \mathbf{f} \in \mathbb{R}^{H'\times W'\times D\times K}, \quad \mathbf{y} \in \mathbb{R}^{H''\times W''\times K}, \qquad W'' = W - W' + 1, \quad H'' = H - H' + 1,$$

Formally, the output is given by

$$y_{i''j''k} = b_k + \sum_{i'=1}^{H'} \sum_{j'=1}^{W'} \sum_{d=1}^{D} f_{i'j'd} \times x_{i''+i'-1,j''+j'-1,d,k}.$$

The call `vl_nnconv(x,f,[])` does not use the biases. Note that the function works with arbitrarily sized inputs and filters (as opposed to, for example, square images).

Output size, padding, and sampling stride. The convolution operator can be adapted to account for image padding and subsampling. Suppose that the input image or map $\mathbf{x}$ has width W and that the filter $\mathbf{f}$ has width $W' \leq W$. Then there are

$$W'' = W - W' + 1$$

possible translations of the filters in the horizontal direction such that the filter is entirely contained in the input $\mathbf{x}$. Hence, by default the filtered signal $\mathbf{y}$ has width W''. However, `vl_nnconv` accepts a padding parameters $[P_h^-, P_h^+, P_w^-, P_w^+]$ whose effect is to virtually pad with zeros the signal $\mathbf{x}$ in the top, bottom, left, and right spatial directions respectively. In this case, these relations are more complex and derived in Sect. 3.

Fully connected layers. In other libraries, a *fully connected blocks or layers* are blocks where each output dimension linearly depends on all the input dimensions. MatConvNet does not distinguishes between fully connected layers and convolutional blocks. Instead, the former is a special case of the latter obtained when the output map $\mathbf{y}$ has dimensions $W'' = H'' = 1$. Internally, `vl_nnconv` handle this case more efficiently if possible.

Filter groups. For additional flexibility, `vl_nnconv` allows to group input feature channels and apply to them different filter groups. To to do so, specify as input a bank of K filters $\mathbf{f} \in \mathbb{R}^{H'\times W'\times D'\times K}$ such that D' divides the number of input dimensions D. These are treated as $g = D/D'$ filter groups; the first group is applied to dimensions $d = 1, \ldots, D'$ of the input

interfaces to all block types. Writing such wrappers, if they are needed, is easy. The core functions, however, focus on providing a straightforward and obvious interface to each block.

$\mathbf{x}$; the second group to dimensions $d = D' + 1, \dots, 2D'$ and so on. Note that the ouptut is still an array $\mathbf{y} \in \mathbb{R}^{H'' \times W'' \times K}$.

An application of grouping is implementing the Krizhevsky and Hinton network [7], which uses two such streams. Another application is sum pooling; in the latter case, one can specify D groups of $D' = 1$ dimensional filters identical filters of value 1 (however, this is considerably slower than calling the dedicated pooling function as given in Section 2.2).

Matrix notation and derivations. It is often convenient to express the convolution operation in matrix form. To this end, let $\phi(\mathbf{x})$ the `im2row` operator, extracting all $W' \times H'$ patches from the map $\mathbf{x}$ and storing them as rows of a $(H''W'') \times (H'W'D)$ matrix. Formally, this operator is given by:

$$[\phi(\mathbf{x})]_{pq} \underset{(i,j,d)=t(p,q)}{=} x_{ijd}$$

where the index mapping $(i, j, d) = t(p, q)$ is

$$i = i'' + i' - 1, \quad j = j'' + j' - 1, \quad p = i'' + H''(j'' - 1), \quad q = i' + H'(j' - 1) + H'W'(d - 1).$$

It is also useful to define the "transposed" operator `row2im`:

$$[\phi^*(M)]_{ijd} = \sum_{(p,q) \in t^{-1}(i,j,d)} M_{pq}.$$

Note that ϕ and ϕ^* are linear operators. Both can be expressed by a matrix $H \in \mathbb{R}^{(H''W''H'W'D) \times (HWD)}$ such that

$$\operatorname{vec}(\phi(\mathbf{x})) = H \operatorname{vec}(\mathbf{x}), \qquad \operatorname{vec}(\phi^*(M)) = H^\top \operatorname{vec}(M).$$

Hence we obtain the following expression for the vectorized output (see [6]):

$$\operatorname{vec} \mathbf{y} = \operatorname{vec}\left(\phi(\mathbf{x}) F\right) = \begin{cases} (I \otimes \phi(\mathbf{x})) \operatorname{vec} F, & \text{or, equivalently,} \\ (F^\top \otimes I) \operatorname{vec} \phi(\mathbf{x}), \end{cases}$$

where $F \in \mathbb{R}^{(H'W'D) \times K}$ is the matrix obtained by reshaping the array $\mathbf{f}$ and I is an identity matrix of suitable dimensions. This allows obtaining the following formulas for the derivatives:

$$\frac{dz}{d(\operatorname{vec} F)^\top} = \frac{dz}{d(\operatorname{vec} \mathbf{y})^\top} (I \otimes \phi(\mathbf{x})) = \operatorname{vec}\left[\phi(\mathbf{x})^\top \frac{dz}{dY}\right]^\top$$

where $Y \in \mathbb{R}^{(H''W'') \times K}$ is the matrix obtained by reshaping the array $\mathbf{y}$. Likewise:

$$\frac{dz}{d(\operatorname{vec} \mathbf{x})^\top} = \frac{dz}{d(\operatorname{vec} \mathbf{y})^\top} (F^\top \otimes I) \frac{d \operatorname{vec} \phi(\mathbf{x})}{d(\operatorname{vec} \mathbf{x})^\top} = \operatorname{vec}\left[\frac{dz}{dY} F^\top\right]^\top H$$

In summary, after reshaping these terms we obtain the formulas:

$$\boxed{\operatorname{vec} \mathbf{y} = \operatorname{vec}\left(\phi(\mathbf{x}) F\right), \qquad \frac{dz}{dF} = \phi(\mathbf{x})^\top \frac{dz}{dY}, \qquad \frac{dz}{dX} = \phi^*\left(\frac{dz}{dY} F^\top\right)}$$

where $X \in \mathbb{R}^{(H'W') \times D}$ is the matrix obtained by reshaping $\mathbf{x}$. Notably, these expressions are used to implement the convolutional operator; while this may seem inefficient, it is instead a fast approach when the number of filters is large and it allows leveraging fast BLAS and GPU BLAS implementations.

2.2 Pooling

`vl_nnpool` implements max and sum pooling. The *max pooling* operator computes the maximum response of each feature channel in a $H' \times W'$ patch

$$y_{i''j''d} = \max_{1 \leq i' \leq H', 1 \leq j' \leq W'} x_{i''+i-1', j''+j'-1,d}.$$

resulting in an output of size $\mathbf{y} \in \mathbb{R}^{H'' \times W'' \times D}$, similar to the convolution operator of Section 2.1. Sum-pooling computes the average of the values instead:

$$y_{i''j''d} = \frac{1}{W'H'} \sum_{1 \leq i' \leq H', 1 \leq j' \leq W'} x_{i''+i'-1, j''+j'-1,d}.$$

Padding and stride. Similar to the convolution operator of Sect. 2.1, `vl_nnpool` supports padding the input; however, the effect is different from padding in the convolutional block as pooling regions straddling the image boundaries are cropped. For max pooling, this is equivalent to extending the input data with $-\infty$; for sum pooling, this is similar to padding with zeros, but the normalization factor at the boundaries is smaller to account for the smaller integration area.

Matrix notation. Since max pooling simply select for each output element an input element, the relation can be expressed in matrix form as $\operatorname{vec} \mathbf{y} = S(\mathbf{x}) \operatorname{vec} \mathbf{x}$ for a suitable selector matrix $S(\mathbf{x}) \in \{0,1\}^{(H''W''D)\times(HWD)}$. The derivatives can the be written as: $\frac{dz}{d(\operatorname{vec} \mathbf{x})^\top} = \frac{dz}{d(\operatorname{vec} \mathbf{y})^\top} S(\mathbf{x})$, for all but a null set of points, where the operator is not differentiable (this usually does not pose problems in optimization by stochastic gradient). For max-pooling, similar relations exists with two differences: S does not depend on the input $\mathbf{x}$ and it is not binary, in order to account for the normalization factors. In summary, we have the expressions:

$$\boxed{\operatorname{vec} \mathbf{y} = S(\mathbf{x}) \operatorname{vec} \mathbf{x}, \qquad \frac{dz}{d \operatorname{vec} \mathbf{x}} = S(\mathbf{x})^\top \frac{dz}{d \operatorname{vec} \mathbf{y}}.} \tag{3}$$

2.3 Activation non-linearities

2.3.1 ReLU

`vl_nnrelu` computes the *Rectified Linear Unit* (ReLU):

$$y_{ijd} = \max\{0, x_{ijd}\}.$$

Matrix notation. With matrix notation, we can express the ReLU as

$$\boxed{\operatorname{vec} \mathbf{y} = \operatorname{diag} \mathbf{s} \operatorname{vec} \mathbf{x}, \qquad \frac{dz}{d \operatorname{vec} \mathbf{x}} = \operatorname{diag} \mathbf{s} \frac{dz}{d \operatorname{vec} \mathbf{y}}}$$

where $\mathbf{s} = [\operatorname{vec} \mathbf{x} > 0] \in \{0,1\}^{HWD}$ is an indicator vector.

2.3.2 Sigmoid

`vl_nnsigmoid` computes the *sigmoid*:

$$y_{ijd} = \sigma(x_{ijd}) = \frac{1}{1+e^{-x_{ijd}}}.$$

Implementation details. The derivative is given by

$$\begin{aligned}\frac{dz}{dx_{ijk}} &= \frac{dz}{dy_{ijd}}\frac{dy_{ijd}}{dx_{ijd}} = \frac{dz}{dy_{ijd}}\frac{-1}{(1+e^{-x_{ijd}})^2}(-e^{-x_{ijd}}) \\ &= \frac{dz}{dy_{ijd}} y_{ijd}(1-y_{ijd}).\end{aligned}$$

In matrix notation:

$$\frac{dz}{d\mathbf{x}} = \frac{dz}{d\mathbf{y}} \odot \mathbf{y} \odot (\mathbf{1}\mathbf{1}^\top - \mathbf{y}).$$

2.4 Normalization

2.4.1 Cross-channel normalization

`vl_nnnormalize` implements a cross-channel normalization operator. Normalization applied independently at each spatial location and groups of channels to get:

$$y_{ijk} = x_{ijk}\left(\kappa + \alpha \sum_{t\in G(k)} x_{ijt}^2\right)^{-\beta},$$

where, for each output channel k, $G(k) \subset \{1, 2, \ldots, D\}$ is a corresponding subset of input channels. Note that input $\mathbf{x}$ and output $\mathbf{y}$ have the same dimensions. Note also that the operator is applied across feature channels in a convolutional manner at all spatial locations.

Implementation details. The derivative is easily computed as:

$$\frac{dz}{dx_{ijd}} = \frac{dz}{dy_{ijd}} L(i,j,d|\mathbf{x})^{-\beta} - 2\alpha\beta x_{ijd} \sum_{k:d\in G(k)} \frac{dz}{dy_{ijk}} L(i,j,k|\mathbf{x})^{-\beta-1} x_{ijk}$$

where

$$L(i,j,k|\mathbf{x}) = \kappa + \alpha \sum_{t\in G(k)} x_{ijt}^2.$$

2.4.2 Batch normalization

`vl_nnbnorm` implements batch normalization [4]. Batch normalization is somewhat different from other neural network blocks in that it performs computation across images/feature maps in a batch (whereas most blocks process different images/feature maps individually).

`y = vl_nnbnorm(x, w, b)` normalizes each channel of the feature map $\mathbf{x}$ averaging over spatial locations and batch instances. Let T the batch size; then

$$\mathbf{x}, \mathbf{y} \in \mathbb{R}^{H\times W\times K\times T}, \qquad \mathbf{w} \in \mathbb{R}^{K}, \qquad \mathbf{b} \in \mathbb{R}^{K}.$$

Note that in this case the input and output arrays are explicitly treated as 4D tensors in order to work with a batch of feature maps. The tensors $\mathbf{w}$ and $\mathbf{b}$ define component-wise multiplicative and additive constants. The output feature map is given by

$$y_{ijkt} = w_k \frac{x_{ijkt} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} + b_k, \quad \mu_k = \frac{1}{HWT}\sum_{i=1}^{H}\sum_{j=1}^{W}\sum_{t=1}^{T} x_{ijkt}, \quad \sigma_k^2 = \frac{1}{HWT}\sum_{i=1}^{H}\sum_{j=1}^{W}\sum_{t=1}^{T}(x_{ijkt}-\mu_k)^2.$$

Implementation details. The derivative of the input with respect to the network output is computed as follows:

$$\frac{dz}{dx_{ijkt}} = \sum_{i''j''k''t''} \frac{dz}{dy_{i''j''k''t''}} \frac{dy_{i''j''k''t''}}{dx_{ijkt}}.$$

Since feature channels are processed independently, all terms with $k'' \neq k$ are null. Hence

$$\frac{dz}{dx_{ijkt}} = \sum_{i''j''t''} \frac{dz}{dy_{i''j''kt''}} \frac{dy_{i''j''kt''}}{dx_{ijkt}},$$

where

$$\frac{dy_{i''j''kt''}}{dx_{ijkt}} = w_k \left(\delta_{i=i'',j=j'',t=t''} - \frac{d\mu_k}{dx_{ijkt}}\right) \frac{1}{\sqrt{\sigma_k^2+\epsilon}} - \frac{w_k}{2}\left(x_{i''j''kt''} - \mu_k\right)\left(\sigma_k^2+\epsilon\right)^{-\frac{3}{2}} \frac{d\sigma_k^2}{dx_{ijkt}},$$

the derivatives with respect to the mean and variance are computed as follows:

$$\begin{aligned}
\frac{d\mu_k}{dx_{ijkt}} &= \frac{1}{HWT}, \\
\frac{d\sigma_k^2}{dx_{i'j'kt'}} &= \frac{2}{HWT}\sum_{ijt}\left(x_{ijkt}-\mu_k\right)\left(\delta_{i=i',j=j',t=t'} - \frac{1}{HWT}\right) = \frac{2}{HWT}\left(x_{i'j'kt'}-\mu_k\right),
\end{aligned}$$

and δ_E is the indicator function of the event E. Hence

$$\begin{aligned}
\frac{dz}{dx_{ijkt}} = &\frac{w_k}{\sqrt{\sigma_k^2+\epsilon}}\left(\frac{dz}{dy_{ijkt}} - \frac{1}{HWT}\sum_{i''j''kt''}\frac{dz}{dy_{i''j''kt''}}\right) \\
&- \frac{w_k}{2(\sigma_k^2+\epsilon)^{\frac{3}{2}}}\sum_{i''j''kt''}\frac{dz}{dy_{i''j''kt''}}\left(x_{i''j''kt''}-\mu_k\right)\frac{2}{HWT}\left(x_{ijkt}-\mu_k\right)
\end{aligned}$$

i.e.

$$\begin{aligned}
\frac{dz}{dx_{ijkt}} = &\frac{w_k}{\sqrt{\sigma_k^2+\epsilon}}\left(\frac{dz}{dy_{ijkt}} - \frac{1}{HWT}\sum_{i''j''kt''}\frac{dz}{dy_{i''j''kt''}}\right) \\
&- \frac{w_k}{\sigma_k^2+\epsilon}\,\frac{x_{ijkt}-\mu_k}{\sqrt{\sigma_k^2+\epsilon}}\,\frac{1}{HWT}\sum_{i''j''kt''}\frac{dz}{dy_{i''j''kt''}}\left(x_{i''j''kt''}-\mu_k\right).
\end{aligned}$$

2.4.3 Spatial normalization

`vl_nnspnorm` implements spatial normalization. Spatial normalization operator acts on different feature channels independently and rescales each input feature by the energy of the features in a local neighborhood . First, the energy of the features is evaluated in a neighbourhood $W' \times H'$

$$n^2_{i''j''d} = \frac{1}{W'H'} \sum_{1\leq i' \leq H', 1 \leq j' \leq W'} x^2_{i''+i'-1-\lfloor \frac{H'-1}{2} \rfloor, j''+j'-1-\lfloor \frac{W'-1}{2} \rfloor, d}.$$

In practice, the factor $1/W'H'$ is adjusted at the boundaries to account for the fact that neighbors must be cropped. Then this is used to normalize the input:

$$y_{i''j''d} = \frac{1}{(1+\alpha n^2_{i''j''d})^\beta} x_{i''j''d}.$$

Implementation details. The neighborhood norm$n^2_{i''j''d}$ can be computed by applying average pooling to x^2_{ijd} using `vl_nnpool` with a $W' \times H'$ pooling region, top padding $\lfloor \frac{H'-1}{2} \rfloor$, bottom padding $H' - \lfloor \frac{H-1}{2} \rfloor - 1$, and similarly for the horizontal padding.

The derivative of spatial normalization can be obtained as follows:

$$\begin{aligned}
\frac{dz}{dx_{ijd}} &= \sum_{i''j''d} \frac{dz}{dy_{i''j''d}} \frac{dy_{i''j''d}}{dx_{ijd}} \\
&= \sum_{i''j''d} \frac{dz}{dy_{i''j''d}} (1+\alpha n^2_{i''j''d})^{-\beta} \frac{dx_{i''j''d}}{dx_{ijd}} - \alpha\beta \frac{dz}{dy_{i''j''d}} (1+\alpha n^2_{i''j''d})^{-\beta-1} x_{i''j''d} \frac{dn^2_{i''j''d}}{d(x^2_{ijd})} \frac{dx^2_{ijd}}{dx_{ijd}} \\
&= \frac{dz}{dy_{ijd}} (1+\alpha n^2_{ijd})^{-\beta} - 2\alpha\beta x_{ijd} \left[\sum_{i''j''d} \frac{dz}{dy_{i''j''d}} (1+\alpha n^2_{i''j''d})^{-\beta-1} x_{i''j''d} \frac{dn^2_{i''j''d}}{d(x^2_{ijd})} \right] \\
&= \frac{dz}{dy_{ijd}} (1+\alpha n^2_{ijd})^{-\beta} - 2\alpha\beta x_{ijd} \left[\sum_{i''j''d} \eta_{i''j''d} \frac{dn^2_{i''j''d}}{d(x^2_{ijd})} \right], \quad \eta_{i''j''d} = \frac{dz}{dy_{i''j''d}} (1+\alpha n^2_{i''j''d})^{-\beta-1} x_{i''j''d}
\end{aligned}$$

Note that the summation can be computed as the derivative of the `vl_nnpool` block.

2.4.4 Softmax

`vl_nnsoftmax` computes the softmax operator:

$$y_{ijk} = \frac{e^{x_{ijk}}}{\sum_{t=1}^D e^{x_{ijt}}}.$$

Note that the operator is applied across feature channels and in a convolutional manner at all spatial locations.

Implementation details. Care must be taken in evaluating the exponential in order to avoid underflow or overflow. The simplest way to do so is to divide from numerator and denominator by the maximum value:

$$y_{ijk} = \frac{e^{x_{ijk} - \max_d x_{ijd}}}{\sum_{t=1}^{D} e^{x_{ijt} - \max_d x_{ijd}}}.$$

The derivative is given by:

$$\frac{dz}{dx_{ijd}} = \sum_k \frac{dz}{dy_{ijk}} \left(e^{x_{ijd}} L(\mathbf{x})^{-1} \delta_{\{k=d\}} - e^{x_{ijd}} e^{x_{ijk}} L(\mathbf{x})^{-2} \right), \quad L(\mathbf{x}) = \sum_{t=1}^{D} e^{x_{ijt}}.$$

Simplifying:

$$\frac{dz}{dx_{ijd}} = y_{ijd} \left(\frac{dz}{dy_{ijd}} - \sum_{k=1}^{K} \frac{dz}{dy_{ijk}} y_{ijk}. \right).$$

In matrix for:

$$\frac{dz}{dX} = Y \odot \left(\frac{dz}{dY} - \left(\frac{dz}{dY} \odot Y \right) \mathbf{1}\mathbf{1}^\top \right)$$

where $X, Y \in \mathbb{R}^{HW \times D}$ are the matrices obtained by reshaping the arrays $\mathbf{x}$ and $\mathbf{y}$. Note that the numerical implementation of this expression is straightforward once the output Y has been computed with the caveats above.

2.5 Losses and comparisons

2.5.1 Log-loss

`vl_logloss` computes the *logarithmic loss*

$$y = \ell(\mathbf{x}, c) = -\sum_{ij} \log x_{ijc}$$

where $c \in \{1, 2, \dots, D\}$ is the ground-truth class. Note that the operator is applied across input channels in a convolutional manner, summing the loss computed at each spatial location into a single scalar.

Implementation details. The derivative is

$$\frac{dz}{dx_{ijd}} = -\frac{dz}{dy} \frac{1}{x_{ijc}} \delta_{\{d=c\}}.$$

2.5.2 Softmax log-loss

`vl_softmaxloss` combines the softmax layer and the log-loss into one step for improved numerical stability. It computes

$$y = -\sum_{ij} \left(x_{ijc} - \log \sum_{d=1}^{D} e^{x_{ijd}} \right)$$

where c is the ground-truth class.

Implementation details. The derivative is given by

$$\frac{dz}{dx_{ijd}} = -\frac{dz}{dy}\left(\delta_{d=c} - y_{ijc}\right)$$

where y_{ijc} is the output of the softmax layer. In matrix form:

$$\frac{dz}{dX} = -\frac{dz}{dy}\left(\mathbf{1}^\top \mathbf{e}_c - Y\right)$$

where $X, Y \in \mathbb{R}^{HW\times D}$ are the matrices obtained by reshaping the arrays $\mathbf{x}$ and $\mathbf{y}$ and $\mathbf{e}_c$ is the indicator vector of class c.

2.5.3 p-distance

The `vl_nnpdistp` function computes the p-th power p-distance between the vectors in the input data $\mathbf{x}$ and a target $\bar{\mathbf{x}}$:

$$y_{ij} = \sum_d |x_{ijd} - \bar{x}_{ijd}|^p, \qquad p > 0.$$

Note that this operator is applied convolutionally, i.e. at each spatial location ij one extracts and compares vectors $x_{ij:}$. There is also a variant `vl_nnpdist` computing the distance itself (not raised to the p-th power:

$$y_{ij} = \left(\sum_d |x_{ijd} - \bar{x}_{ijd}|^p\right)^{\frac{1}{p}}$$

Implementation details. The derivative of the first operator is given by:

$$\begin{aligned}\frac{dz}{dx_{ijd}} &= \frac{dz}{dy_{ij}} p|x_{ijd} - \bar{x}_{ijd}|^{p-1} \operatorname{sign}(x_{ijd} - \bar{x}_{ijd}) \\ &= \frac{dz}{dy_{ij}} p|x_{ijd} - \bar{x}_{ijd}|^{p-2}(x_{ijd} - \bar{x}_{ijd}).\end{aligned}$$

The derivative of the second operator by:

$$\begin{aligned}\frac{dz}{dx_{ijd}} &= \frac{dz}{dy_{ij}}\frac{1}{p}\left(\sum_{d'} |x_{ijd'} - \bar{x}_{ijd'}|^p\right)^{\frac{1}{p}-1} p|x_{ijd} - \bar{x}_{ijd}|^{p-2}(x_{ijd} - \bar{x}_{ijd}) \\ &= \frac{dz}{dy_{ij}}\frac{|x_{ijd} - \bar{x}_{ijd}|^{p-2}(x_{ijd} - \bar{x}_{ijd})}{y_{ij}^{p-1}}.\end{aligned}$$

The formulas simplify a little for $p = 1, 2$ which are therefore implemented as special cases.

3 Receptive fields and transformations

Very often it is interesting to map a convolutional operator back to image space. Due to various downsampling and padding operations, this is not entirely trivial and analysed in this section. Considering a 1D example will be enough; hence, consider an input signal x_i, a filter $f_{i'}$ and an output signal $y_{i''}$, where

$$1 - P_w^- \leq i \leq W + P_w^+, \qquad 1 \leq i' \leq W'.$$

where $P_w^-, P_w^+ \geq 0$ is the amount of padding applied to the input signal to the left and right respectively. The output y_i is obtained by applying a filter of size W' in the following range

$$-P_w^- + S(i''-1) + [1, W'] = [1 - P_w^- + S(i''-1), -P_w^- + S(i''-1) + W']$$

where S is the subsampling factor. For $i'' = 1$, the leftmost sample of this window falls at the leftmost sample of the padded input signal. The rightmost sample $W'' \geq i''$ of the output signal is obtained by guaranteeing that the filter window stays within the padded input signal:

$$-P_w^- + S(i''-1) + W' \leq W + P_w^+ \qquad \Rightarrow \qquad i'' \leq \frac{W - W' + P_w^- + P_w^+}{S} + 1$$

Since i'' is an integer quantity, the maximum value is:

$$W'' = \lfloor \frac{W - W' + P_w^- + P_w^+}{S} \rfloor + 1. \tag{4}$$

Note that W'' is smaller than 1 if $W' > W + P_w^- + P_w^+$; in this case the filter $f_{i'}$ is wider than the padded input signal x_i and the domain of $y_{i''}$ becomes empty (this generates an error in most MatConvNet functions).

Now consider a sequence of convolutional operators. One starts from an input signal x_0 of width W_0 and applies a sequence of operators of parameters $(P_{w1}^-, P_{w1}^+, S_1, W_1')$, $(P_{w2}^-, P_{w2}^+, S_2, W_2'), \dots, (P_{wL}^-, P_{wL},^+ S_L, W_L')$ to obtain signals $x_1, x_2, \dots, x_L$ of width $W_1, W_2, \dots, W_L$ respectively. First, note that the widths of these signals are obtained from W_0 and the operator parameters by a recursive application of (4). Due to the flooring operation, unfortunately it does not seem possible to simplify significantly the resulting expression. However, disregarding this operation one obtains the approximate expression

$$W_l \approx \frac{W_0}{\prod_{p=1}^l S_q} - \sum_{p=1}^l \frac{W_p' - P_{wp}^- - P_{wp}^+ - S_p}{\prod_{q=p}^l S_q}.$$

This expression is exact when $S_1 = S_2 = \dots = S_l = 1$:

$$W_l = W_0 - \sum_{p=1}^l (W_p' - P_{wp}^- - P_{wp}^+ - 1).$$

Note in particular that without padding and filters $W_p' > 1$ the widths decrease with depth.

Next, we are interested in computing the samples x_{0,i_0} of the input signal that affect a particular sample x_{L,i_L} at the end of the chain (this we call the *receptive field* of the operator). Suppose that at level l we have determined that samples $i_l \in [I_l^-(i_L), I_l^+(i_L)]$ affect x_{L,i_L} and compute the samples at the level below. These are given by the union of filter applications:

$$\cup_{I_l^-(i_L) \le i_l \le I_l^+(i_L)} \left(-P_{wl}^- + S_l(i_l - 1) + [1, W_l']\right).$$

Hence we find the recurrences:

$$I_{l-1}^-(i_L) = -P_{wl}^- + S_l(I_l^-(i_L) - 1) + 1,$$
$$I_{l-1}^+(i_L) = -P_{wl}^- + S_l(I_l^+(i_L) - 1) + W_l'.$$

Given the base case $I_L^-(i_L) = I_L^+(i_L) = i_L$, one gets:

$$I_l^-(i_L) = 1 + \left(\prod_{p=l+1}^{L} S_p\right)(i_L - 1) - \sum_{p=l+1}^{L}\left(\prod_{q=l+1}^{p-1} S_q\right) P_{wp}^-,$$

$$I_l^-(i_L) = 1 + \left(\prod_{p=l+1}^{L} S_p\right)(i_L - 1) + \sum_{p=l+1}^{L}\left(\prod_{q=l+1}^{p-1} S_q\right) (W_p' - 1 - P_{wp}^-).$$

We can now compute several quantities of interest. First, the receptive field width on the image is:

$$\Delta = I_0^+(i_L) - I_0^-(i_L) + 1 = 1 + \sum_{p=1}^{L}\left(\prod_{q=1}^{p-1} S_q\right)(W_p' - 1).$$

Second, the leftmost sample of the input signal x_{0,i_0} affecting output sample x_{l,i_L} is

$$i_0(i_L) = 1 + \left(\prod_{p=1}^{L} S_p\right)(i_L - 1) - \sum_{p=1}^{L}\left(\prod_{q=1}^{p-1} S_q\right) P_{wp}^-$$

Note that the effect of padding at different layers accumulates, resulting in an overall padding potentially larger than the padding P_{w1}^- specified by the first operator (i.e. $i_0(1) \le -P_{w1}^-$).

Finally, it is interesting to reparametrise these coordinates as if the discrete signal were continuous and if (as it is usually the case) all the convolutional operators are "centred". For example, the operators could be filters with an odd size W_l' equal to the delta function (e.g. for $W_l' = 5$ then $f_l = (0, 0, 1, 0, 0)$). Let $u_L = i_L$ be the "continuous" version of index i_L, where each discrete sample corresponds, as per MATLAB's convention, to a tile of extent $u_L \in [-1/2, 1/2] + i_L$ (hence u_L is the centre coordinate of a sample). As before, $x_L(u_L)$ is obtained by applying an operator to the input signal x_0; the centre of the support of this operator falls at coordinate:

$$u_0(u_L) = \alpha\,(u_L - 1) + \beta = i_0(u_L) + \frac{\Delta - 1}{2} = \alpha\,(u_L - 1) + \beta.$$

Hence

$$\alpha = \prod_{p=1}^{L} S_p, \qquad \beta = 1 + \sum_{p=1}^{L}\left(\prod_{q=1}^{p-1} S_q\right)\left(\frac{W_p' - 1}{2} - P_{wp}^-\right).$$

Note in particular that the offset is zero if $S_1 = \cdots = S_L = 1$ and $P_{wp}^- = (W_p' - 1)/2$ as in this case the padding is just enough such that the leftmost application of each convolutional operator has the centre that falls on the first sample of the corresponding input signal.

4 Network wrappers and examples

It is easy enough to combine the computational blocks of Sect. 2 in any network DAG by writing a corresponding MATLAB script. Nevertheless, MatConvNet provides a simple wrapper for the common case of a linear chain. This is implemented by the `vl_simplenn` and `vl_simplenn_move` functions.

`vl_simplenn` takes as input a structure `net` representing the CNN as well as input `x` and potentially output derivatives `dzdy`, depending on the mode of operation. Please refer to the inline help of the `vl_simplenn` function for details on the input and output formats. In fact, the implementation of `vl_simplenn` is a good example of how the basic neural net building block can be used together and can serve as a basis for more complex implementations.

4.1 Pre-trained models

`vl_simplenn` is easy to use with pre-trained models (see the homepage to download some). For example, the following code downloads a model pre-trained on the ImageNet data and applies it to one of MATLAB stock images:

```
% setup MatConvNet in MATLAB
run matlab/vl_setupnn

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

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

Note that the image should be preprocessed before running the network. While preprocessing specifics depend on the model, the pre-trained model contain a `net.normalization` field that describes the type of preprocessing that is expected. Note in particular that this network takes images of a fixed size as input and requires removing the mean; also, image intensities are normalized in the range [0,255].

The next step is running the CNN. This will return a `res` structure with the output of the network layers:

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

The output of the last layer can be used to classify the image. The class names are contained in the `net` structure for convenience:

```
% 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)) ;
```

Note that several extensions are possible. First, images can be cropped rather than rescaled. Second, multiple crops can be fed to the network and results averaged, usually for improved results. Third, the output of the network can be used as generic features for image encoding.

4.2 Learning models

As MatConvNet can compute derivatives of the CNN using back-propagation, it is simple to implement learning algorithms with it. A basic implementation of stochastic gradient descent is therefore straightforward. Example code is provided in `examples/cnn_train`. This code is flexible enough to allow training on NMINST, CIFAR, ImageNet, and probably many other datasets. Corresponding examples are provided in the `examples/` directory.

4.3 Running large scale experiments

For large scale experiments, such as learning a network for ImageNet, a NVIDIA GPU (at least 6GB of memory) and adequate CPU and disk speeds are highly recommended. For example, to train on ImageNet, we suggest the following:

- Download the ImageNet data `http://www.image-net.org/challenges/LSVRC`. Install it somewhere and link to it from `data/imagenet12`
- Consider preprocessing the data to convert all images to have an height 256 pixels. This can be done with the supplied `utils/preprocess-imagenet.sh` script. In this manner, training will not have to resize the images every time. Do not forget to point the training code to the pre-processed data.
- Consider copying the dataset in to a RAM disk (provided that you have enough memory!) for faster access. Do not forget to point the training code to this copy.
- Compile MatConvNet with GPU support. See the homepage for instructions.

Once your setup is ready, you should be able to run `examples/cnn_imagenet` (edit the file and change any flag as needed to enable GPU support and image pre-fetching on multiple threads).

If all goes well, you should expect to be able to train with 200-300 images/sec.

5 About MatConvNet

MatConvNet 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.
- *Power.* The implementation can run the latest models such as Krizhevsky *et al.* [7], including the DeCAF and Caffe variants, and variants from the Oxford Visual Geometry Group. Pre-learned features for different tasks can be easily downloaded.
- *Efficiency.* The implementation is quite efficient, supporting both CPU and GPU computation (in the latest versions of MALTAB).
- *Self contained.* The implementation is fully self-contained, requiring only MATLAB and a compatible C/C++ compiler to work (GPU code requires the freely-available CUDA DevKit). Several fully-functional image classification examples are included.

Relation to other CNN implementations. There are many other open-source CNN implementations. MatConvNet borrows its convolution algorithms from Caffe (and is in fact capable of running most of Caffe's models). Caffe is a C++ framework using a custom CNN definition language based on Google Protocol Buffers. Both MatConvNet and Caffe are predated by Cuda-Convnet [7], a C++ -based project that allows defining a CNN architectures using configuration files. While Caffe and Cuda-Convnet can be somewhat faster than MatConvNet, the latter exposes individual CNN building blocks as MATLAB functions, as well as integrating with the native MATLAB GPU support, which makes it very convenient for fast prototyping. The DeepLearningToolbox [8] is a MATLAB toolbox implementing, among others, CNNs, but it does not seem to have been tested on large scale problems. While MatConvNet specialises on CNNs and computer vision applications, there are several general-purpose machine learning frameworks which include CNN support, but none of them interfaces natively with MATLAB. For example, the Torch7 toolbox [3] uses Lua and Theano [1] uses Python.

5.1 Acknowledgments

The implementation of several CNN computations in this library are inspired by the Caffe library [5] (however, Caffe is *not* a dependency). Several of the example networks have been trained by Karen Simonyan as part of [2].

We kindly thank NVIDIA for suppling GPUs used in the creation of this software.

References

[1] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[2] 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.

[3] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[4] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv e-prints*, 2015.

[5] Yangqing Jia. Caffe: An open source convolutional architecture for fast feature embedding. `http://caffe.berkeleyvision.org/`, 2013.

[6] D. B. Kinghorn. Integrals and derivatives for correlated gaussian fuctions using matrix differential calculus. *International Journal of Quantum Chemestry*, 57:141–155, 1996.

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

[8] R. B. Palm. Prediction as a candidate for learning deep hierarchical models of data. Master's thesis, 2012.