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

Chapter 1

Introduction to MatConvNet

MatConvNet is a MATLAB toolbox implementing *Convolutional Neural Networks* (CNN) for computer vision applications. Since the breakthrough work of [7], 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 [12] 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 optimizations 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?

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 (chapter 4); 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 section 1.4), 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

[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

[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.

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 [7] and the very deep networks of [9] from millions of images. Pre-trained versions of several of these powerful models can be downloaded from the MatConvNet home page (**??**). 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 1.1 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 1.1 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.

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 MatConvNet) 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 chapter 4, 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 1.4) as well as support for backpropagation (section 2.3) 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

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

[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`

```
% 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.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.classes.description{best}, best, bestScore)) ;
```

Figure 1.1: 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.

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, MATLAB provides wrappers such as `vl_simplenn` for standard CNN architectures such as AlexNet [7] or Network-in-Network [8]. 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 (**??**). 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.

1.2 MatConvNet at 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, directly as a MATLAB commands. 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 4.1). 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 4.1). chapter 4 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). It also provide a much more flexible wrapper supporting networks with arbitrary topologies, encapsulated in the `dagnn.DagNN` MATLAB class.

- *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.

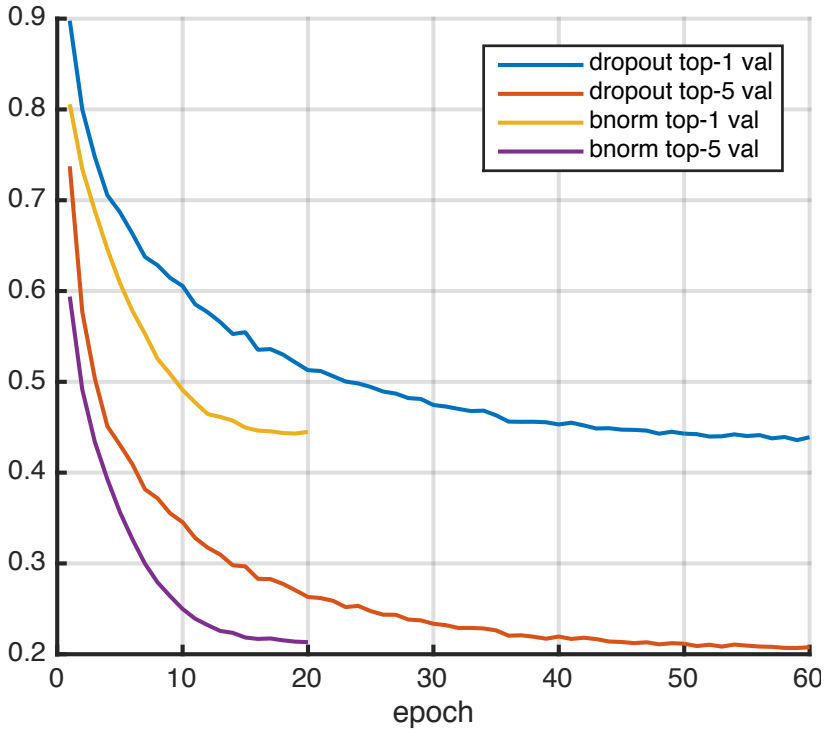


Figure 1.2: Training AlexNet on ImageNet ILSVRC: dropout vs batch normalisation.

1.3 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 1.2 illustrates the top-1 and top-5 validation errors as a model similar to AlexNet [7] 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 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 [7], VGG-S, VGG-M, VGG-S [1], and VGG-VD-16, and VGG-VD-19 [10]. The example code of Figure 1.1 shows how one such models can be used in a few lines of MATLAB code.

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

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 1.1: ImageNet training speed (images/s).

1.4 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 1.3 on ImageNet ILSVRC. Table 1.1 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 1.2 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.

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

Table 1.2: Multiple GPU speed (images/s).

1.5 Future

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.

1.6 Acknowledgments

MatConvNet is a community project, and as such acknowledgments got to all contributors. 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.

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 [1] and [11].

Chapter 2

CNN fundamentals

This chapter reviews fundamental concepts of CNNs as needed to understand how to use MatConvNet.

2.1 Overview

A *Convolutional Neural Network* (CNN) is a function g mapping data $\mathbf{x}$, for example an image, to an output vector $\mathbf{y}$. The function $g = f_L \circ \cdots \circ f_1$ 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).

MATLAB 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` (locar response normalization – LRN), 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 1.1, 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

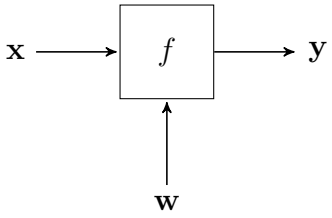
efficient method (for this type of problems), networks may contain several million parameters and need to be trained on millions of images; thus, efficiency is a paramount in MATLAB design, as further discussed in section 1.4. SGD requires also to compute the CNN derivatives, as explained in the next section.

2.2 CNN topologies

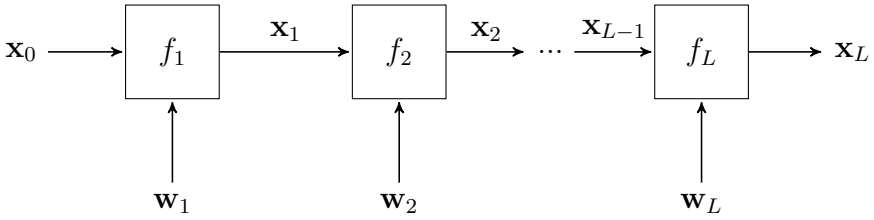
In the simplest case, computational blocks form a simple chain; however, more complex topologies are possible and in fact very useful in certain applications. This section discusses such configurations and introduce a graphical notation to visualize them.

2.2.1 Simple networks

Start by considering a computational block f in the network. This can be represented schematically as a box receiving $\mathbf{x}$ and $\mathbf{w}$ as inputs and producing $\mathbf{y}$ as output:



In the simplest case, this graph reduces to a chain $(f_1, f_2, \ldots, f_L)$. Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L$ 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:

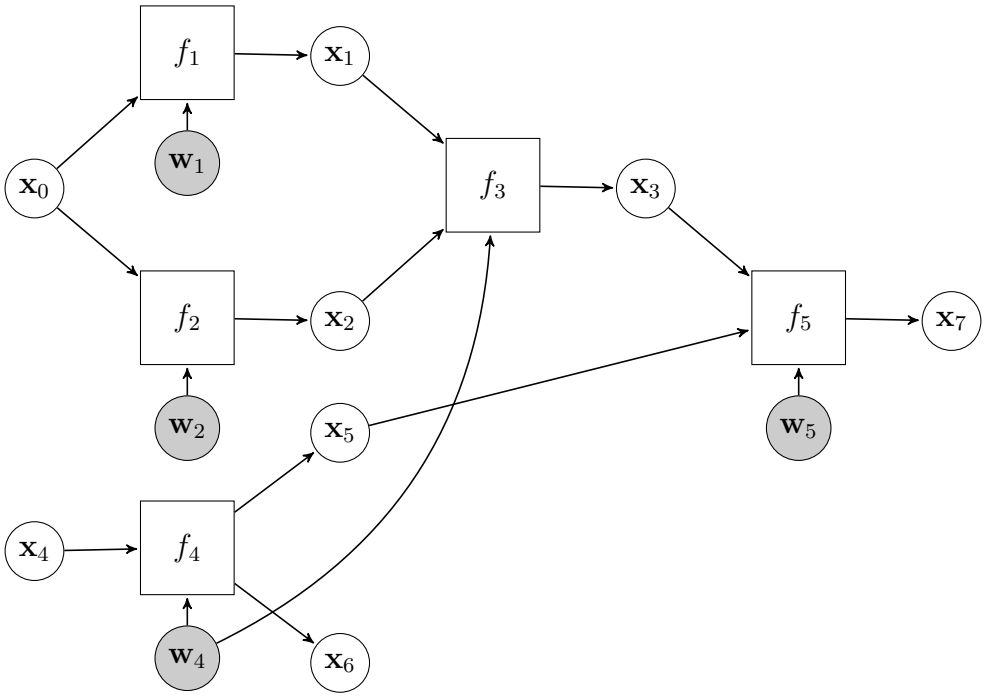


Given an input $\mathbf{x}_0$, evaluating the network is a simple matter of evaluating all the intermediate stages in order to compute an overall function $\mathbf{x}_L = f(\mathbf{x}_0; \mathbf{w}_1, \ldots, \mathbf{w}_L)$.

2.2.2 Directed acyclic graphcs

A moment's thought reveals that one is not limited in chaining blocks one after another; it only suffices that, when a block has to be evaluated, all its input have been evaluated prior to it. This is always possible provided that blocks are interconnected in a *directed acyclic graph*, or DAG.

In order to visualize DAGSs, it is useful to introduce additional nodes for the network variables, as in the following example:



Here boxes denote functions and circles variables (parameters are treated as a special kind of variables). In the example, $\mathbf{x}_0$ and $\mathbf{x}_4$ are the inputs of the CNN and $\mathbf{x}_6$ and $\mathbf{x}_7$ the outputs. Functions can take any number of inputs (e.g. f_3 and f_5 take two) and have any number of outputs (e.g. f_4 has two). There are a few noteworthy properties of this graph:

1. The graph is bipartite, in the sense that arrows always go from boxes to circles and circles to boxes.

2. Functions can have any number of inputs or outputs; variables and parameters can have an arbitrary number of outputs; variables have at most one input.

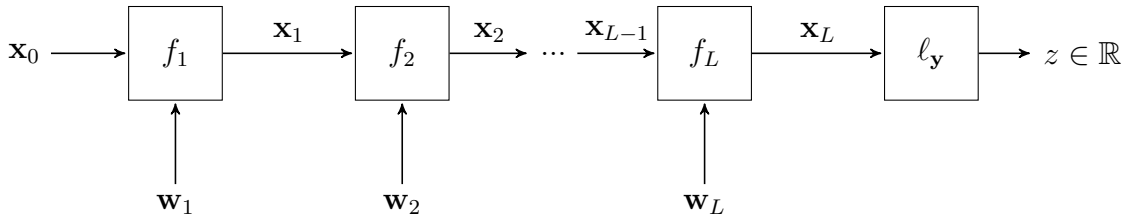
3. While there is usually one parameter per function, the same parameter can feed into two or more functions, and therefore be *shared* among them.

4. Since the graph is acyclic, the CNN can be evaluated by sorting the functions and computing them one after another (in the example evaluating f_1, f_2, f_3, f_4 and then f_5 in this order would work).

2.3 CNN derivatives: backpropagation

The fundamental operation to learn a network is computing the derivative of a training 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.

In order to understand backpropagation, consider first a simple CNN terminating in a loss function $\ell_{\mathbf{y}}$:



In learning, we are computing in determining the gradient of the loss z with respect to each parameter:

$$\frac{dz}{d\mathbf{w}_l} = \frac{d}{d\mathbf{w}_l}\left[\ell_{\mathbf{y}} \circ f_L(\cdot;\mathbf{w}_L) \circ ... \circ f_2(\cdot;\mathbf{w}_2) \circ f_1(\mathbf{x}_0;\mathbf{w}_1)\right]$$

By applying the chain rule, we find that this can be rewritten as

$$\frac{dz}{d\mathbf{w}_l} = \frac{d\ell_{\mathbf{y}}(\mathbf{x}_L)}{d(\operatorname{vec}\mathbf{x}_L)^\top}\frac{d\operatorname{vec} f_L(\mathbf{x}_{L-1};\mathbf{w}_L)}{d(\operatorname{vec}\mathbf{x}_{L-1})^\top}\cdots\frac{d\operatorname{vec} f_{l+1}(\mathbf{x}_l;\mathbf{w}_{l+1})}{d(\operatorname{vec}\mathbf{x}_l)^\top}\frac{d\operatorname{vec} f_l(\mathbf{x}_{l-1};\mathbf{w}_l)}{d\mathbf{w}_l^\top}$$

where the derivatives are computed at the working point determined by the input $\mathbf{x}_0$ and the current value of the parameters. It is convenient to rewrite this expression in term of variables only, leaving the functional dependencies implicit:

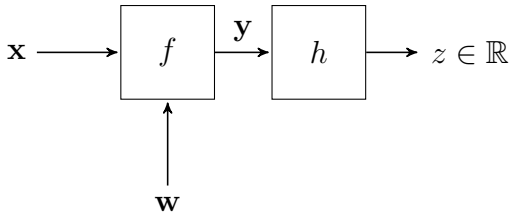
$$\frac{dz}{d\mathbf{w}_l} = \frac{dz}{d(\operatorname{vec}\mathbf{x}_L)^\top}\frac{d\operatorname{vec}\mathbf{x}_L}{d(\operatorname{vec}\mathbf{x}_{L-1})^\top}\cdots\frac{d\operatorname{vec}\mathbf{x}_{l+1}}{d(\operatorname{vec}\mathbf{x}_l)^\top}\frac{d\operatorname{vec}\mathbf{x}_l}{d\mathbf{w}_l^\top}$$

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.

Note that this expression involves computing and multiplying the Jacobians of all building block from level L back to level l. Unfortunately intermediate Jacobians such as $d\operatorname{vec}\mathbf{x}_l/d(\operatorname{vec}\mathbf{x}_{l-1})^\top$ are extremely large $H_lW_lD_l \times H_{l-1}W_{l-1}D_{l-1}$ matrices (often worth GBs of data), which makes the naive application of the chain rule unfeasible.

The trick is to notice that only the intermediate but unneded Jacobians are so large; in fact, since the loss z is a scalar value, the target derivatives $dz/d\mathbf{w}_l$ have the same dimensions as $\mathbf{w}_l$. The key idea of backpropagation is a way to organize the computation in order to avoid the explicit computation of the intermediate large matrices.

This is best seen by focusing on an intermediate layer f with parameter $\mathbf{w}$, as follows:



Here the function h lumps together all layers of the network from f to the scalar output z (loss). The derivatives of $h \circ f$ with respect to the data and parameters can be rewritten as:

$$\frac{dz}{d(\operatorname{vec}\mathbf{x})^\top} = \frac{dz}{d(\operatorname{vec}\mathbf{y})^\top}\frac{d\operatorname{vec}\mathbf{y}}{d(\operatorname{vec}\mathbf{x})^\top}, \quad \frac{dz}{d(\operatorname{vec}\mathbf{w})^\top} = \frac{dz}{d(\operatorname{vec}\mathbf{y})^\top}\frac{d\operatorname{vec}\mathbf{y}}{d(\operatorname{vec}\mathbf{w})^\top}. \tag{2.1}$$

Note that, just like the parameter derivative $dz/d\mathbf{w}_l$, the data derivatives $dz/d(\operatorname{vec}\mathbf{x})^\top$ and $dz/d(\operatorname{vec}\mathbf{y})^\top$ have the same size as the data $\mathbf{x}$ and $\mathbf{y}$ respectively, and hence can be explicitly computed. If (2.1) can be somehow computed, this provides a way to compute all the parameter derivates. In particular, the data derivative $dz/d(\operatorname{vec}\mathbf{y})^\top$ can be passed backward to compute the derivatives for the layers prior to f.

The key in implementing backpropagation then is to implement for each building block two computational paths:

- **Forward mode.** This mode takes the input data $\mathbf{x}$ and parameter $\mathbf{w}$ and computes the output variable $\mathbf{y}$.

- **Backward mode.** This mode takes the input data $\mathbf{x}$, the parameter $\mathbf{w}$, and the output derivative $dz/d\mathbf{y}$ and computes the parameter derivative $dz/d\mathbf{w}$ as well as the input derivative $dz/d\mathbf{x}$. Crucially, in this step the required intermediate Jacobian is never *explicitly* computed.

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] = vl_nnconv(x,w,[],dzdy)`. 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 $\langle dz/d\mathbf{y}, d\mathbf{y}/d\mathbf{w}\rangle$.

2.3.1 Backpropagation in DAGs

Backpropagation can be applied to network with a DAG topology as well. Given a DAG, one can always sort the variables in such a way that they can be computed in sequence, by evaluating the corresponding function:

$$\mathbf{x}_1 = f_1(\mathbf{x}_0), \quad \mathbf{x}_2 = f_2(\mathbf{x}_1, \mathbf{x}_0), \quad ..., \quad \mathbf{x}_L = f_L(\mathbf{x}_1, \ldots, \mathbf{x}_{L-1}).$$

Here we made two inconsequential assumptions. The first one is that each block f_l produces exactly one variable $\mathbf{x}_l$ as output; if a block produces two or more, we can reduce back to this situation by replicating a block as needed. The second assumption is that each block in the sequence take as (direct) input *all* previous variables; this is a "worst case" scenario as in practice the dependency is usually limited to a few. Note also that parameters can be seen as special cases of variables.

To work out the network output derivatives with respect to any intermediate variable,

consider the sequence of functions:

$$\begin{aligned}
\mathbf{x}_L &= h_L(\mathbf{x}_0, \dots, \mathbf{x}_{L-1}) && = f_L(\mathbf{x}_0, \dots, \mathbf{x}_{L-1}),\\
\mathbf{x}_L &= h_{L-1}(\mathbf{x}_0, \dots, \mathbf{x}_{L-2}) && = h_L(\mathbf{x}_0, \dots, \mathbf{x}_{L-2}, f_{L-1}(\mathbf{x}_0, \dots, \mathbf{x}_{L-2})),\\
\mathbf{x}_L &= h_{L-2}(\mathbf{x}_0, \dots, \mathbf{x}_{L-3}) && = h_{L-1}(\mathbf{x}_0, \dots, \mathbf{x}_{L-3}, f_{L-2}(\mathbf{x}_0, \dots, \mathbf{x}_{L-3})),\\
&\vdots && \vdots\\
\mathbf{x}_L &= h_2(\mathbf{x}_0, \mathbf{x}_1) && = h_3(\mathbf{x}_0, \mathbf{x}_1, f_2(\mathbf{x}_0, \mathbf{x}_1)),\\
\mathbf{x}_L &= h_1(\mathbf{x}_0) && = h_2(\mathbf{x}_0, f_1(\mathbf{x}_0)).
\end{aligned}$$

The functions $\mathbf{x}_L = h_l(\mathbf{x}_0, \dots, \mathbf{x}_{l-1})$ can be interpreted as the evaluation of a new DAG obtaining by *clamping* variables $\mathbf{x}_0, \dots, \mathbf{x}_{l-1}$ to some arbitrary value and computing the remaining variables as before. This amounts to deleting all functions $f_1, \dots, f_{l-1}$ from the graph and treating $\mathbf{x}_0, \dots, \mathbf{x}_{l-1}$ as inputs to the DAG. Below we emphasise this functional dependency by the alternative notation $\mathbf{x}_L|\mathbf{x}_0, \dots, \mathbf{x}_{l-1}$.

We can now take the derivative as follows:

$$\begin{aligned}
\frac{d \operatorname{vec} \mathbf{x}_L|\mathbf{x}_0}{d(\operatorname{vec} \mathbf{x}_0)^\top} &= \frac{d \operatorname{vec} h_1}{d(\operatorname{vec} \mathbf{x}_0)^\top}\\
&= \frac{d \operatorname{vec} h_2}{d(\operatorname{vec} \mathbf{x}_0)^\top} + \frac{d \operatorname{vec} h_2}{d(\operatorname{vec} \mathbf{x}_1)^\top} \frac{d \operatorname{vec} f_1}{d(\operatorname{vec} \mathbf{x}_0)^\top}\\
&= \frac{d \operatorname{vec} h_3}{d(\operatorname{vec} \mathbf{x}_0)^\top} + \frac{d \operatorname{vec} h_3}{d(\operatorname{vec} \mathbf{x}_2)^\top} \frac{d \operatorname{vec} f_2}{d(\operatorname{vec} \mathbf{x}_0)^\top} + \frac{d \operatorname{vec} h_2}{d(\operatorname{vec} \mathbf{x}_1)^\top} \frac{d \operatorname{vec} f_1}{d(\operatorname{vec} \mathbf{x}_0)^\top}\\
&= \vdots\\
&= \sum_{l=1}^{L} \frac{d \operatorname{vec} h_{l+1}}{d(\operatorname{vec} \mathbf{x}_l)^\top} \frac{d \operatorname{vec} f_l}{d(\operatorname{vec} \mathbf{x}_0)^\top}
\end{aligned}$$

where we implicitly set $h_{L+1}(\mathbf{x}_0, \dots, \mathbf{x}_L) = \mathbf{x}_L$. Hence we see that the derivative of the network output $\mathbf{x}_L$ w.r.t. the input $\mathbf{x}_0$ is obtained as a linear combination of terms. Each term involves the derivatives of one of the blocks f_l with respect to the input $\mathbf{x}_0$ and the derivative of a function h_{l+1} with respect to the variable $\mathbf{x}_l$.

In this process we are required to compute the derivative of functions h_{l+1} with respect to the last variable $\mathbf{x}_l$ while keeping $\mathbf{x}_0, \dots, \mathbf{x}_{l-1}$ fixed as parameters. For example

$$\frac{d \operatorname{vec} \mathbf{x}_L|\mathbf{x}_0, \mathbf{x}_1}{d(\operatorname{vec} \mathbf{x}_1)^\top} = \frac{d \operatorname{vec} h_2}{d(\operatorname{vec} \mathbf{x}_1)^\top} = \sum_{l=2}^{L} \frac{d \operatorname{vec} h_{l+1}}{d(\operatorname{vec} \mathbf{x}_l)^\top} \frac{d \operatorname{vec} f_l}{d(\operatorname{vec} \mathbf{x}_1)^\top}.$$

computes the derivative of the network with respect to $\mathbf{x}_1$ while clamping $\mathbf{x}_0$ to the current working point. In general, the derivatives with respect all intermediate nodes are given by:

$$\frac{d \operatorname{vec} \mathbf{x}_L|\mathbf{x}_0, \dots, \mathbf{x}_l}{d(\operatorname{vec} \mathbf{x}_l)^\top} = \frac{d \operatorname{vec} h_{l+1}}{d(\operatorname{vec} \mathbf{x}_l)^\top} = \sum_{k=l+1}^{L} \frac{d \operatorname{vec} h_{k+1}}{d(\operatorname{vec} \mathbf{x}_k)^\top} \frac{d \operatorname{vec} f_k}{d(\operatorname{vec} \mathbf{x}_l)^\top}. \tag{2.2}$$

While this may seem fairly complicated, it is in fact a minor variation of the algorithm for simple networks. First, we assume that $\mathbf{x}_l = z \in \mathbb{R}$ is a scalar quantity, so that all

derivatives (2.2) have a reasonable size. Then, one computes these derivatives by working backward from the output of the graph.

In more detail, in order to compute $d(\operatorname{vec} h_{l+1})/d(\operatorname{vec} \mathbf{x}_l)^\top$ in (2.2) back propagation should:

1. Identify all the blocks that have $\mathbf{x}_l$ as input. In the most general case, this are all the blocks $f_k(\mathbf{x}_0, \ldots, \mathbf{x}_l, \ldots, \mathbf{x}_{k-1})$ such that $k > l$.

2. For each such block f_k:

   a) Retrieve the $d(\operatorname{vec} h_{k+1})/d(\operatorname{vec} \mathbf{x}_k)^\top$ computed at a previous step.

   b) Use the "backward mode" of the building block f_k to compute the product in (2.2) without explicitly computing the large Jacobian matrix.

   c) Accumulate the resulting matrix to the derivative $d(\operatorname{vec} h_{l+1})/d(\operatorname{vec} \mathbf{x}_l)^\top$.

While this procedure is correct, it is also not very convenient to implement as it requires to visiting block f_k again for each of its input variables, every time running its corresponding "backward mode" routine, but for a different parameter. Instead, it is generally much more efficient to compute all the derivatives of a block in one step. This can be done by rearranging the algorithm slightly, backtracking over blocks instead of variables:

1. Start by initialising all derivatives $d(\operatorname{vec} h_{l+1})/d(\operatorname{vec} \mathbf{x}_l)^\top$ $l = 1, \ldots, L-1$ to zero. For $l = L$ set the derivative to 1 (this corresponds to the auxiliary function $h_{L+1}(\mathbf{x}_0, \ldots, \mathbf{x}_L) = \mathbf{x}_L$ defined above).

2. For all blocks f_k, $k = L, L-1, \ldots, 1$ in backward order:

   a) For all the block's inputs $\mathbf{x}_l$, $l < k$:

      i. Use the "backward mode" of the block f_k to compute the product $[d \operatorname{vec} h_{k+1}/d(\operatorname{vec} \mathbf{x}_k)^\top] \times [d \operatorname{vec} f_k/d(\operatorname{vec} \mathbf{x}_l)^\top]$ without explicitly computing the large Jacobian matrix.

      ii. Accumulate the resulting matrix to the derivative $d(\operatorname{vec} h_{l+1})/d(\operatorname{vec} \mathbf{x}_l)^\top$.

Note that step (2.a.1) in this algorithm is correct because by the time the algorithm visits block f_k the computation of $[d \operatorname{vec} h_{k+1}/d(\operatorname{vec} \mathbf{x}_k)^\top]$ is complete.

Chapter 3

Wrappers and pre-trained models

It is easy enough to combine the computational blocks of chapter 4 "manually". However, it is usually much more convenient to use them through a *wrapper* that can implement CNN architectures given a model specification. The available wrapper are briefly summarised in section 3.1.

MatConvNet also comes with many pre-trained models for image classification (most of which are trained on the ImageNet ILSVRC challenge), image segmentation, text spotting, and face recognition. These are very simple to use, as illustrated in section 3.2.

3.1 Wrappers

MatConvNet provides two wrappers: SimpleNN for basic chains of blocks (section 3.1.1) , and DagNN for more complex graphs. simple wrapper for the common case of a linear chain (section 3.1.2).

3.1.1 SimpleNN

The SimpleNN wrapper is suitable for networks consisting of linear chains of computational blocks. It is largely implemented by the `vl_simplenn` function (evaluation of the CNN and of its derivatives), with a few other support functions such as `vl_simplenn_move` (moving the CNN between CPU and GPU) and `vl_simplenn_display` (obtain and/or print information about the CNN).

`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.

3.1.2 DagNN

The DagNN wrapper is more complex that SimpleNN as it has to support arbitrary graph topologies. Its design is object oriented, with one class implementing each layer type. While

this adds complexity, and makes the wrapper slightly slower for tiny CNN architectures (e.g. MNIST), it is in practice much more flexible and easier to extend.

DagNN is implemented by the `dagnn.DagNN` class (under the `dagnn` namespace).

3.2 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/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.meta.normalization.imageSize(1:2)) ;
im_ = im_ - net.meta.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.meta.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.meta.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.

3.3 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.

3.4 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.

Chapter 4

Computational blocks

This chapters describes the individual computational blocks supported by MatConvNet. The interface of a CNN computational block <block> is designed after the discussion in chapter 2. The block is implemented as a MATLAB function y = vl_nn<block>(x,w) that takes as input MATLAB arrays x and w representing the input data and parameters and returns an array y as output. In general, x and y are 4D real arrays packing N maps or images, as discussed above, whereas w may have an arbitrary shape.

The function implementing each block is capable of working in the backward direction as well, in order to compute derivatives. This is done by passing a third optional argument dzdy representing the derivative of the output of the network with respect to $\mathbf{y}$; in this case, the function returns the derivatives [dzdx,dzdw] = vl_nn<block>(x,w,dzdy) with respect to the input data and parameters. The arrays dzdx, dzdy and dzdw have the same dimensions of x, y and w respectively (see section 2.3).

Different functions may use a slightly different syntax, as needed: many functions can take additional optional arguments, specified as a property-value pairs; some do not have parameters w (e.g. a rectified linear unit); others can take multiple inputs and parameters, in which case there may be more than one x, w, dzdx, dzdy or dzdw. See the rest of the chapter and MATLAB inline help for details on the syntax.[1]

The rest of the chapter describes the blocks implemented in MatConvNet, with a particular focus on their analytical definition. Refer instead to MATLAB inline help for further details on the syntax.

4.1 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

$$\mathbf{x} \in \mathbb{R}^{H\times W\times D}, \quad \mathbf{f} \in \mathbb{R}^{H'\times W'\times D\times D''}, \quad \mathbf{y} \in \mathbb{R}^{H''\times W''\times D''}.$$

[1] Other parts of the library will wrap these functions into objects with a perfectly uniform interface; however, the low-level functions aim at providing a straightforward and obvious interface even if this means differing slightly from block to block.

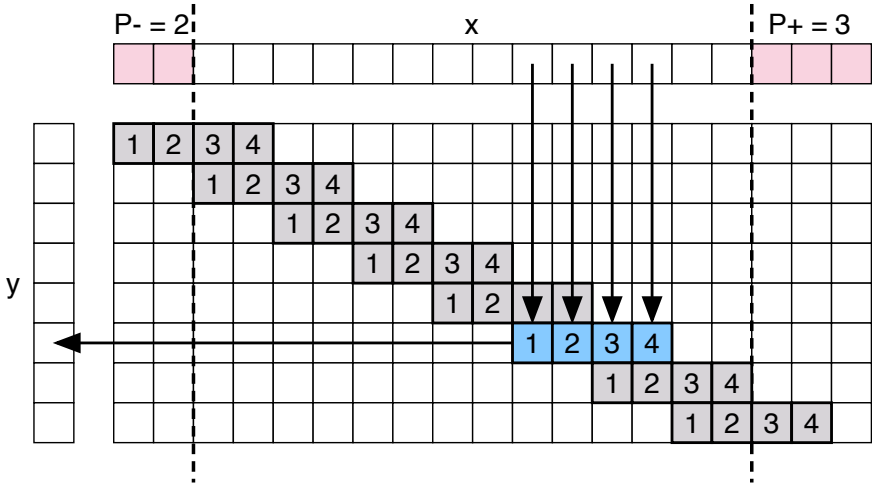


Figure 4.1: **Convolution.** The figure illustrates the process of filtering a 1D signal $\mathbf{x}$ by a filter f to obtain a signal $\mathbf{y}$. The filter has $H' = 4$ elements and is applied with a stride of $S_h = 2$ samples. The purple areas represented padding $P_- = 2$ and $P_+ = 3$ which is zero-filled. Filters are applied in a sliding-window manner across the input signal. The samples of $\mathbf{x}$ involved in the calculation of a sample of $\mathbf{y}$ are shown with arrow. Note that the rightmost sample of $\mathbf{x}$ is never processed by any filter application due to the sampling step. While in this case the sample is in the padded region, this can happen also without padding.

The process of convolving a signal is illustrated in Figure 4.1 for a 1D slice. Formally, the output is given by

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

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). See section 6.1 for technical details.

Padding and stride. `vl_nnconv` allows to specify top-bottom-left-right paddings $(P_h^-, P_h^+, P_w^-, P_w^+)$ of the input array and subsampling strides (S_h, S_w) of the output array:

$$y_{i''j''d''} = b_{d''} + \sum_{i'=1}^{H'} \sum_{j'=1}^{W'} \sum_{d'=1}^{D} f_{i'j'd} \times x_{S_h(i''-1)+i'-P_h^-,S_w(j''-1)+j'-P_w^-,d',d''}.$$

In this expression, the array $\mathbf{x}$ is implicitly extended with zeros as needed.

Output size. `vl_nnconv` computes only the "valid" part of the convolution; i.e. it requires each application of a filter to be fully contained in the input support. The size of the output is computed in section 5.2 and is given by:

$$H'' = 1 + \left\lfloor \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rfloor.$$

Note that the padded input must be at least as large as the filters: $H + P_h^- + P_h^+ \geq H'$, otherwise an error is thrown.

Receptive field size and geometric transformations. Very often it is useful to relate geometrically the indexes of the various array to the input data (usually images) in term of coordinate transformations and size of the receptive field (i.e. of the image region that affects an output). This is derived in section 5.2.

Fully connected layers. In other libraries, a *fully connected blocks or layers* are linear functions where each output dimension 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 when possible.

Filter groups. For additional flexibility, `vl_nnconv` allows to group channels of the input array $\mathbf{x}$ and apply to each group different subsets of filters. To use this feature, specify as input a bank of D'' filters $\mathbf{f} \in \mathbb{R}^{H' \times W' \times D' \times D''}$ 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 $\mathbf{x}$; the second group to dimensions $d = D' + 1, \ldots, 2D'$ and so on. Note that the output is still an array $\mathbf{y} \in \mathbb{R}^{H'' \times W'' \times D''}$.

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 4.3).

4.2 Convolution transpose (deconvolution)

The *convolution transpose* block (sometimes referred to as "deconvolution") is the transpose of the convolution block described in section 4.1. In MatConvNet, convolution transpose is implemented by the function `vl_nnconvt`.

In order to understand convolution transpose, let:

$$\mathbf{x} \in \mathbb{R}^{H \times W \times D}, \quad \mathbf{f} \in \mathbb{R}^{H' \times W' \times D \times D''}, \quad \mathbf{y} \in \mathbb{R}^{H'' \times W'' \times D''},$$

be the input tensor, filters, and output tensors. Imagine operating in the reverse direction by using the filter bank $\mathbf{f}$ to convolve the output $\mathbf{y}$ to obtain the input $\mathbf{x}$, using the definitions given in section 4.1 for the convolution operator; since convolution is linear, it can be expressed as a matrix M such that $\operatorname{vec} \mathbf{x} = M \operatorname{vec} \mathbf{y}$; convolution transpose computes instead $\operatorname{vec} \mathbf{y} = M^\top \operatorname{vec} \mathbf{x}$. This process is illustrated for a 1D slice in Figure 4.2.

There are two important applications of convolution transpose The first one are the so called *deconvolutional networks* [13] and other networks such as convolutional decoders that use the transpose of a convolution. The second one is implementing data interpolation. In fact, as the convolution block supports input padding and output downsampling, the convolution transpose block supports input upsampling and output cropping.

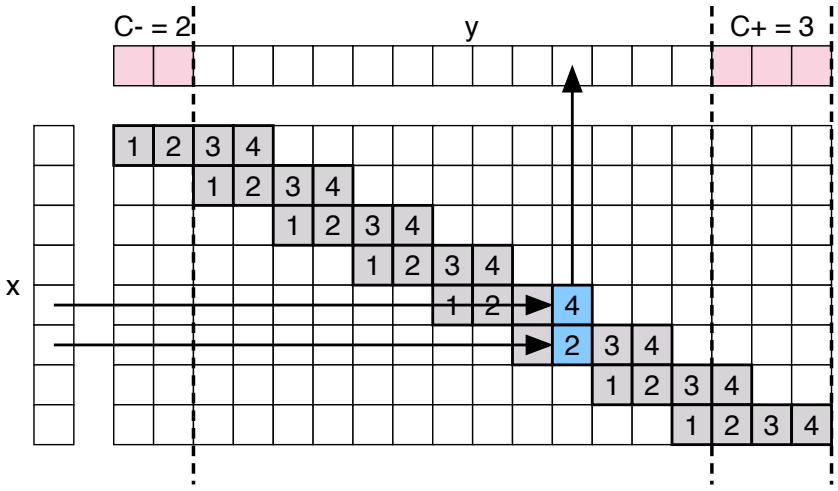


Figure 4.2: **Convolution transpose.** The figure illustrates the process of filtering a 1D signal x by a filter f to obtain a signal y. The filter is applied in a sliding-window, in a pattern that is the transpose of Figure 4.1. The filter has $H' = 4$ samples in total, although each filter application uses two of them (blue squares) in a circulant manner. The purple areas represented crops $C_- = 2$ and $C_+ = 3$ which are discarded. The samples of x involved in the calculation of a sample of y are shown with arrow. Note that, differently from Figure 4.1, there is not any samples to the right of $\mathbf{y}$ which is not involved in a convolution operation. This is because the width H'' of the output $\mathbf{y}$, which given H' can be determined up to U_h samples, is selected to be the smallest possible.

Convolution transpose can be expressed in closed form in the following rather unwieldy expression (derived in section 6.2):

$$y_{i''j''d''} = \sum_{d'=1}^{D} \sum_{i'=0}^{q(H',S_h)} \sum_{j'=0}^{q(W',S_w)} f_{1+S_h i'+m(i''+P_h^-,S_h),\ 1+S_w j'+m(j''+P_w^-,S_w),\ d'',d'} \times x_{1-i'+q(i''+P_h^-,S_h),\ 1-j'+q(j''+P_w^-,S_w),\ d'} \quad (4.1)$$

where

$$m(k,S) = (k-1) \bmod S, \qquad q(k,n) = \left\lfloor \frac{k-1}{S} \right\rfloor,$$

(S_h, S_w) are the vertical and horizontal *input upsampling factors*, $(P_h^-, P_h^+, P_h^-, P_h^+)$ the *output crops*, and $\mathbf{x}$ and $\mathbf{f}$ are zero-padded as needed in the calculation. Note also that filter k is stored as a slice $\mathbf{f}_{:,:,k,:}$ of the 4D tensor $\mathbf{f}$.

The height of the output array $\mathbf{y}$ is given by

$$H'' = S_h(H-1) + H' - P_h^- - P_h^+.$$

A similar formula holds true for the width. These formulas are derived in section 5.3 along with expression for the receptive field of the operator.

We now illustrate the action of convolution transpose in an example (see also Figure 4.2). Consider a 1D slice in the vertical direction, assume that the crop parameters are zero,

and that $S_h > 1$. Consider the output sample $y_{i''}$ where the index i'' is chose such that S_h divides $i''-1$; according to (4.1), this sample is obtained as a weighted summation of $x_{i''/S_h}, x_{i''/S_h-1}, \ldots$ (note that the order is reversed). The weights are the filter elements f_1, $f_{S_h}, f_{2S_h}, \ldots$ subsampled with a step of S_h. Now consider computing the element $y_{i''+1}$; due to the rounding in the quotient operation $q(i'', S_h)$, this output sampled is obtained as a weighted combination of the same elements of the input x that were used to compute $y_{i''}$; however, the filter weights are now shifted by one place to the right: f_2, $f_{S_h+1}, f_{2S_h+1}, \ldots$. The same is true for $i''+2, i''+3, \ldots$ until we hit $i''+S_h$. Here the cycle restarts after shifting $\mathbf{x}$ to the right by one place. Effectively, convolution transpose works as an *interpolating filter*.

4.3 Spatial 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

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

resulting in an output of size $\mathbf{y} \in \mathbb{R}^{H'' \times W'' \times D}$, similar to the convolution operator of section 4.1. Sum-pooling computes the average of the values instead:

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

Detailed calculation of the derivatives are provided in section 6.3.

Padding and stride. Similar to the convolution operator of section 4.1, `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.

4.4 Activation functions

MatConvNet supports the following activation functions:

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

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

- *Sigmoid.* `vl_nnsigmoid` computes the *sigmoid*:

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

See section 6.4 for implementation details.

4.5 Normalization

4.5.1 Local response normalization (LRN)

`vl_nnnormalize` implements the Local Response Normalization (LRN) operator. This operator is applied independently at each spatial location and to groups of feature channels as follows:

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

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 uniformly at all spatial locations.

See section 6.5.1 for implementation details.

4.5.2 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

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

See section 6.5.2 for implementation details.

4.5.3 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 $W' \times H'$

$$n_{i''j''d}^2 = \frac{1}{W'H'} \sum_{1 \le i' \le H', 1 \le j' \le W'} x_{i''+i'-1-\lfloor \frac{H'-1}{2} \rfloor, j''+j'-1-\lfloor \frac{W'-1}{2} \rfloor, d}^2.$$

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:

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

See section 4.5.3 for implementation details.

4.5.4 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. Softmax can be seen as the combination of an activation function (exponential) and a normalization operator. See section 6.5.4 for implementation details.

4.6 Losses and comparisons

4.6.1 Log-loss

`vl_logloss` computes the *logarithmic loss*

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

where $c \in \{1, 2, \ldots, D\}$ 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. See section 6.6.1 for implementation details.

4.6.2 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. See section 6.6.2 for implementation details.

4.6.3 p-distance

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

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

Note that this operator is applied convolutionally, i.e. at each spatial location ij one extracts and compares vectors $x_{ij:}$. By specifying the option 'noRoot', `true` it is possible to compute a variant omitting the root:

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

See section 6.6.3 for implementation details.

Chapter 5

Geometry

This chapter looks at the geometry of the CNN input-output mapping.

5.1 Preliminaries

In this section we are interested in understanding how components in a CNN depend on components in the layers before it, and in particular on components of the input. Since CNNs can incorporate blocks that perform complex operations, such as for example cropping their inputs based on data-dependent terms (e.g. Fast R-CNN), this information is generally available only at "run time" and cannot be uniquely determined given only the structure of the network. Furthermore, blocks can implement complex operations that are difficult to characterise in simple terms. Therefore, the analysis will be necessarily limited in scope.

We consider here blocks such as convolutions for which one can deterministically establish dependency chains between network components. We also assume that all the inputs and outputs $\mathbf{x}$ are in the usual form of spatial maps, and therefore indexed as $x_{i,j,d,k}$ where i, j are spatial coordinates.

Consider a layer $\mathbf{y} = f(\mathbf{x})$. We are interested in establishing which components of $\mathbf{x}$ influence which components of $\mathbf{y}$. We also assume that this relation can be expressed in terms of a sliding rectangular window field, called *receptive field*. This means that the output component $y_{i'',j''}$ depends only on the input components $x_{i,j}$ where $(i,j) \in \Omega(i'',j'')$ (note that feature channels are implicitly coalesced in this discussion). The set $\Omega(i'',j'')$ is a rectangle defined as follows:

$$i \in \alpha_h(i''-1) + \beta_h + [-\frac{\Delta_h - 1}{2}, \frac{\Delta_h - 1}{2}] \tag{5.1}$$

$$j \in \alpha_v(j''-1) + \beta_v + [-\frac{\Delta_v - 1}{2}, \frac{\Delta_v - 1}{2}] \tag{5.2}$$

where (α_h, α_v) is the *stride*, (β_h, β_v) the offset, and (Δ_h, Δ_v) the *receptive field size*.

5.2 Simple filters

We now compute the receptive field geometry $(\alpha_h, \alpha_v, \beta_h, \beta_v, \Delta_h, \Delta_v)$ for the most common operators, namely filters. We consider in particular *simple filters* that are characterised by

an integer size, stride, and padding.

It suffices to reason in 1D. Let H' bet the vertical filter dimension, S_h the subampling stride, and P_h^- and P_h^+ the amount of zero padding applied to the top and the bottom of the input $\mathbf{x}$. Here the value $y_{i''}$ depends on the samples:

$$\begin{aligned} x_i : i \in [1, H'] + S_h(i'' - 1) - P_h^- \\ = [-\frac{H'-1}{2}, \frac{H'-1}{2}] + S_h(i''-1) - P_h^- + \frac{H'+1}{2}. \end{aligned}$$

Hence

$$\alpha_h = S_h, \qquad \beta_h = \frac{H'+1}{2} - P_h^-, \qquad \Delta_h = H'.$$

A similar relation holds for the horizontal direction.

Note that many blocks (e.g. max pooling, LNR, ReLU, most loss functions etc.) have a filter-like receptive field geometry. For example, ReLU can be considered a 1×1 filter, such that $H = S_h = 1$ and $P_h^- = P_h^+ = 0$. Note that in this case $\alpha_h = 1$, $\beta_h = 1$ and $\Delta_h = 1$.

In addition to computing the receptive field geometry, we are often interested in determining the sizes of the arrays $\mathbf{x}$ and $\mathbf{y}$ throughout the architecture. In the case of filters, and once more reasoning for a 1D slice, we notice that y_i'' can be obtained for $i'' = 1, 2, \ldots, H''$ where H'' is the largest value of i'' before the receptive fields falls outside $\mathbf{x}$ (including padding). If H is the height of the input array $\mathbf{x}$, we get the condition

$$H' + S_h(H'' - 1) - P_h^- \leq H + P_h^+.$$

Hence

$$H'' = \left\lfloor \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rfloor + 1. \tag{5.3}$$

5.2.1 Pooling in Caffe

MatConvNet treats pooling operators like filters, using the rules above. In the library Caffe, this is done slightly differently, creating some incompatibilities. In their case, the pooling window is allowed to shift enough such that the last application always includes the last pixel of the input. If the stride is greater than one, this means that the last application of the pooling window can be partially outside the input boundaries even if padding is "officially" zero.

More formally, if H' is the pool size and H the size of the signal, the last application of the pooling window has index $i'' = H''$ such that

$$S_h(i''-1) + H'\big|_{i''=H''} \geq H \qquad \Leftrightarrow \qquad H'' = \left\lceil \frac{H - H'}{S_h} \right\rceil + 1.$$

If there is padding, the same logic applies after padding the input image, such that the output has height:

$$H'' = \left\lceil \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rceil + 1.$$

This is the same formula as above of filters, but with the ceil instead of floor operator. Note that in practice $P_h^- = P_h^+ = P_h$ since Caffe does not support asymmetric padding.

Unfortunately, it gets more complicated. Using the formula above, it can happen that the last padding application is completely outside the input image and Caffe tries to avoid it. This requires

$$S(i''-1) - P_h^- + 1\big|_{i''=H''} \leq H \qquad \Leftrightarrow \qquad H'' \leq \frac{H-1+P_h^-}{S_h} + 1. \tag{5.4}$$

Using the fact that for integer a, b, one has $\lceil a/b \rceil = \lfloor (a+b-1)/b \rfloor$, we can rewrite the expression for H'' as follows

$$H'' = \left\lceil \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rceil + 1 = \left\lfloor \frac{H-1+P_h^-}{S_h} + \frac{P_h^+ + S_h - H'}{S_h} \right\rfloor + 1.$$

Hence if $P_h^+ + S_h \leq H'$ then the second term is less than zero and (5.4) is satisfied. In practice, Caffe assumes that $P_h^+, P_h^- \leq H' - 1$, as otherwise the first filter application falls entirely in the padded region. Hence, we can upper bound the second term:

$$\frac{P_h^+ + S_h - H'}{S_h} \leq \frac{S_h - 1}{S_h} \leq 1.$$

We conclude that, for any choices of P_h^+ and S_h allowed by Caffe, the formula above may violate constraint (5.4) by at most one unit. Caffe has a special provision for that and lowers H'' by one when needed. Furthermore, we see that if $P_h^+ = 0$ *and* $S_h \leq H'$ (which is often the case and may be assumed by Caffe), then the equation is also satisfied and Caffe skips the check.

Next, we find MatConvNet equivalents for these parameters. Assume that Caffe applies a symmetric padding P_h. Then in MatConvNet $P_h^- = P_h$ to align the top part of the output signal. To match Caffe, the last sample of the last filter application has to be on or to the right of the last Caffe-padded pixel:

$$\underbrace{S_h \left(\underbrace{\left\lfloor \frac{H - H' + P_h^- + P_h^+}{S_h} + 1 \right\rfloor}_{\text{MatConvNet rightmost pooling index}} - 1 \right) + H'}_{\text{MatConvNet rightmost pooled input sample}} \geq \underbrace{H + 2P_h^-}_{\text{Caffe rightmost input sample with padding}}.$$

Rearranging

$$\left\lfloor \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rfloor \geq \frac{H - H' + 2P_h^-}{S_h}$$

Using $\lfloor a/b \rfloor = \lceil (a-b+1)/b \rceil$ we get the *equivalent* condition:

$$\left\lceil \frac{H - H' + 2P_h^-}{S_h} + \frac{P_h^+ - P_h^- - S_h + 1}{S_h} \right\rceil \geq \frac{H - H' + 2P_h^-}{S_h}$$

Removing the ceil operator lower bounds the left-hand side of the equation and produces the *sufficient* condition

$$P_h^+ \geq P_h^- + S_h - 1.$$

As before, this may still be too much padding, causing the last pool window application to be entirely in the rightmost padded area. MatConvNet places the restriction $P_h^+ \leq H' - 1$, so that

$$P_h^+ = \min\{P_h^- + S_h - 1, H' - 1\}.$$

For example, a pooling region of width $H' = 3$ samples with a stride of $S_h = 1$ samples and null Caffe padding $P_h^- = 0$, would result in a right MatConvNet padding of $P_h^+ = 1$.

5.3 Convolution transpose

The convolution transpose block is similar to a simple filter, but somewhat more complex. Recall that convolution transpose (section 6.2) is the transpose of the convolution operator, which in turn is a filter. Reasoning for a 1D slice, let x_i be the input to the convolution transpose block and $y_{i''}$ its output. Furthermore let U_h, C_h^-, C_h^+ and H' be the upsampling factor, top and bottom crops, and filter height, respectively.

If we look at the convolution transpose backward, from the output to the input (see also Figure 4.2), the data dependencies are the same as for the convolution operator, studied in section 5.2. Hence there is an interaction between x_i and $y_{i''}$ only if

$$1 + U_h(i-1) - C_h^- \leq i'' \leq H' + U_h(i-1) - C_h^- \tag{5.5}$$

where cropping becomes padding and upsampling becomes downsampling. Turning this relation around, we find that

$$\left\lceil \frac{i'' + C_h^- - H'}{S_h} \right\rceil + 1 \leq i \leq \left\lfloor \frac{i'' + C_h^- - 1}{S_h} \right\rfloor + 1.$$

Note that, due to rounding, it is not possible to express this set tightly in the form outlined above. We can however relax these two relations (hence obtaining a slightly larger receptive field) and conclude that

$$\alpha_h = \frac{1}{U_h}, \qquad \beta_h = \frac{2C_h^- - H' + 1}{2U_h} + 1, \qquad \Delta_h = \frac{H' - 1}{U_h} + 1.$$

Next, we want to determine the height H'' of the output $\mathbf{y}$ of convolution transpose as a function of the heigh H of the input $\mathbf{x}$ and the other parameters. Swapping input and output in Equation 5.3 results in the constraint:

$$H = 1 + \left\lfloor \frac{H'' - H' + C_h^- + C_h^+}{U_h} \right\rfloor.$$

If H is now given as input, it is not possible to recover H'' uniquely from this expression; instead, all the following values are possible

$$S_h(H-1) + H' - C_h^- - C_h^+ \leq H'' < S_h H + H' - C_h^- - C_h^+.$$

This is due to the fact that due to the fact that U_h acts as a downsampling factor in the standard convolution direction and some of the samples to the right of the convolution input $\mathbf{y}$ may be ignored by the filter (see also Figure 4.1 and Figure 4.2).

Since the height of $\mathbf{y}$ is then determined up to S_h samples, and since the extra samples would be ignored by the computation and stay zero, we choose the tighter definition and set

$$H'' = U_h(H-1) + H' - C_h^- - C_h^+.$$

5.4 Transposing receptive fields

Suppose we have determined that a later $\mathbf{y} = f(\mathbf{x})$ has a receptive field transformation $(\alpha_h, \beta_h, \Delta_h)$ (along one spatial slice). Now suppose we are given a block $\mathbf{x} = g(\mathbf{y})$ which is the "transpose" of f, just like the convolution transpose layer is the transpose of the convolution layer. By this, we mean that, if $y_{i''}$ depends on x_i due to f, then x_i depends on $y_{i''}$ due to g.

Note that, by definition of receptive fields, f relates the inputs and outputs index pairs (i, i'') given by Equation 5.1, which can be rewritten as

$$-\frac{\Delta_h - 1}{2} \leq i - \alpha_h(i'' - 1) - \beta_h \leq \frac{\Delta_h - 1}{2}.$$

A simple manipulation of this expression results in the equivalent expression:

$$-\frac{(\Delta_h + \alpha_h - 1)/\alpha_h - 1}{2} \leq i'' - \frac{1}{\alpha_h}(i-1) - \frac{1 + \alpha_h - \beta_h}{\alpha_h} \leq \frac{(\Delta_h + \alpha_h - 1)/\alpha_h - 1}{2\alpha_h}.$$

Hence, in the reverse direction, this corresponds to a RF transformation

$$\hat{\alpha}_h = \frac{1}{\alpha_h}, \qquad \hat{\beta}_h = \frac{1 + \alpha_h - \beta_h}{\alpha_h}, \qquad \hat{\Delta}_h = \frac{\Delta_h + \alpha_h - 1}{\alpha_h}.$$

Example 1. *For convolution, we have found the parameters:*

$$\alpha_h = S_h, \qquad \beta_h = \frac{H'+1}{2} - P_h^-, \qquad \Delta_h = H'.$$

Using the formulas just found, we can obtain the RF transformation for convolution transpose:

$$\begin{aligned}
\hat{\alpha}_h &= \frac{1}{\alpha_h} = \frac{1}{S_h}, \\
\hat{\beta}_h &= \frac{1 + S_h - (H'+1)/2 + P_h^-}{S_h} = \frac{P_h^- - H'/2 + 1/2}{S_h} + 1 = \frac{2P_h^- - H' + 1}{S_h} + 1, \\
\hat{\Delta}_h &= \frac{H' + S_h - 1}{S_h} = \frac{H'-1}{S_h} + 1.
\end{aligned}$$

Hence we find again the formulas found in section 5.3.

5.5 Composing receptive fields

Suppose now to compose two layers $h = g \circ f$ with receptive fields $(\alpha_f, \beta_f, \Delta_f)$ and $(\alpha_g, \beta_g, \Delta_g)$ (once again we consider only a 1D slice in the vertical direction, the horizontal one being the same). The goal is to compute the receptive field of h.

To do so, pick a sample i_g in the domain of g. The first and last sample i_f in the domain of f to affect i_g are given by:

$$i_f = \alpha_f(i_g - 1) + \beta_f \pm \frac{\Delta_f - 1}{2}.$$

Likewise, the first and last sample i_g to affect a given output sample i_h are given by

$$i_g = \alpha_g(i_h - 1) + \beta_g \pm \frac{\Delta_g - 1}{2}.$$

Substituting one relation into the other, we see that the first and last sample i_f in the domain of $g \circ f$ to affect i_h are:

$$\begin{aligned} i_f &= \alpha_f \left(\alpha_g(i_h - 1) + \beta_g \pm \frac{\Delta_g - 1}{2} - 1 \right) + \beta_f \pm \frac{\Delta_f - 1}{2} \\ &= \alpha_f\alpha_g(i_h - 1) + \alpha_f\beta_g - 1 + \beta_f \pm \frac{\alpha_f(\Delta_g - 1) + \Delta_f - 1}{2} \end{aligned}$$

We conclude that

$$\alpha_h = \alpha_f\alpha_g, \qquad \beta_h = \alpha_f(\beta_g - 1) + \beta_f, \qquad \Delta_h = \alpha_f(\Delta_g - 1) + \Delta_f$$

5.6 Overlaying receptive fields

Consider now the combination $h(f(\mathbf{x}_1), g(\mathbf{x}_2))$ where the domain of f and g are the same. Given the rule above, it is possible to compute how each output sample i_h depends on each input sample i_f through the f and on each input sample i_g through g. Suppose that this gives receptive fields $(\alpha_{hf}, \beta_{hf}, \Delta_{hf})$ and $(\alpha_{hg}, \beta_{hg}, \Delta_{hg})$ respectively. Now assume that the domain of f and g coincide, i.e. $\mathbf{x} = \mathbf{x}_1 = \mathbf{x}_2$. The goal is to determine the combined receptive field.

This is only possible if, and only if, $\alpha = \alpha_{hg} = \alpha_{hf}$. Only in this case, in fact, it is possible to find a sliding window receptive field that tightly encloses the receptive field due to g and f at all points according to formulas (5.1). We say that these two receptive fields are *compatbile*. The range of input samples $i = i_f = i_g$ that affect any output sample i_h is then given by

$$\begin{aligned} i_{\max} &= \alpha(i_h - 1) + a, & a &= \min \left\{ \beta_{hf} - \frac{\Delta_{hf} - 1}{2}, \beta_g - \frac{\Delta_{hg} - 1}{2} \right\}, \\ i_{\min} &= \alpha(i_h - 1) + b, & b &= \max \left\{ \beta_{hf} + \frac{\Delta_{hf} - 1}{2}, \beta_g + \frac{\Delta_{hg} - 1}{2} \right\}. \end{aligned}$$

We conclude that the combined receptive field is

$$\alpha = \alpha_{hg} = \alpha_{hf}, \qquad \beta = \frac{a + b}{2}, \qquad \delta = b - a + 1.$$

Chapter 6

Implementation details

This chapter contains calculations and details.

6.1 Convolution

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]):

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

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.

6.2 Convolution transpose

In order to understand the definition of convolution transpose, let $\mathbf{y}$ obtained from $\mathbf{x}$ by the convolution operator as defined in section 4.1 (including padding and downsampling). Since this is a linear operation, it can be rewritten as $\operatorname{vec}\mathbf{y} = M\operatorname{vec}\mathbf{x}$ for a suitable matrix M; convolution transpose computes instead $\operatorname{vec}\mathbf{x} = M^\top\operatorname{vec}\mathbf{y}$. While this is simple to describe in term of matrices, what happens in term of indexes is tricky. In order to derive a formula for the convolution transpose, start from standard convolution (for a 1D signal):

$$y_{i''} = \sum_{i'=1}^{H'} f_{i'}x_{S(i''-1)+i'-P_h^-}, \quad 1 \le i'' \le 1 + \left\lfloor\frac{H - H' + P_h^- + P_h^+}{S}\right\rfloor,$$

where S is the downsampling factor, P_h^- and P_h^+ the padding, H the length of the input signal, $\mathbf{x}$ and H' the length of the filter $\mathbf{f}$. Due to padding, the index of the input data $\mathbf{x}$ may exceed the range $[1, H]$; we implicitly assume that the signal is zero padded outside this range.

In order to derive an expression of the convolution transpose, we make use of the identity $\operatorname{vec}\mathbf{y}^\top(M\operatorname{vec}\mathbf{x}) = (\operatorname{vec}\mathbf{y}^\top M)\operatorname{vec}\mathbf{x} = \operatorname{vec}\mathbf{x}^\top(M^\top\operatorname{vec}\mathbf{y})$. Expanding this in formulas:

$$\begin{aligned}
\sum_{i''=1}^{b} y_{i''}\sum_{i'=1}^{W'} f_{i'}x_{S(i''-1)+i'-P_h^-} &= \sum_{i''=-\infty}^{+\infty}\sum_{i'=-\infty}^{+\infty} y_{i''}\ f_{i'}\ x_{S(i''-1)+i'-P_h^-} \\
&= \sum_{i''=-\infty}^{+\infty}\sum_{k=-\infty}^{+\infty} y_{i''}\ f_{k-S(i''-1)+P_h^-}\ x_k \\
&= \sum_{i''=-\infty}^{+\infty}\sum_{k=-\infty}^{+\infty} y_{i''}\ f_{(k-1+P_h^-)\bmod S+S\left(1-i''+\left\lfloor\frac{k-1+P_h^-}{S}\right\rfloor\right)+1}\ x_k \\
&= \sum_{k=-\infty}^{+\infty} x_k\sum_{q=-\infty}^{+\infty} y_{\left\lfloor\frac{k-1+P_h^-}{S}\right\rfloor+2-q}\ f_{(k-1+P_h^-)\bmod S+S(q-1)+1}.
\end{aligned}$$

Summation ranges have been extended to infinity by assuming that all signals are zero padded as needed. In order to recover such ranges, note that $k \in [1, H]$ (since this is the range of elements of $\mathbf{x}$ involved in the original convolution). Furthermore, $q \geq 1$ is the minimum value of q for which the filter $\mathbf{f}$ is non zero; likewise, $q \leq \lfloor (H'-1)/2 \rfloor + 1$ is a fairly tight upper bound on the maximum value (although, depending on k, there could be an element less). Hence

$$x_k = \sum_{q=1}^{1+\lfloor \frac{H'-1}{S} \rfloor} y_{\left\lfloor \frac{k-1+P_h^-}{S} \right\rfloor + 2 - q} f_{(k-1+P_h^-) \bmod S + S(q-1)+1}, \qquad k = 1, \dots, H. \tag{6.1}$$

Note that the summation extrema in (6.1) can be refined slightly to account for the finite size of $\mathbf{y}$ and $\mathbf{w}$:

$$\begin{aligned} \max \left\{ 1, \left\lfloor \frac{k-1+P_h^-}{S} \right\rfloor + 2 - H'' \right\} &\leq q \\ &\leq 1 + \min \left\{ \left\lfloor \frac{H'-1-(k-1+P_h^-) \bmod S}{S} \right\rfloor, \left\lfloor \frac{k-1+P_h^-}{S} \right\rfloor \right\}. \end{aligned}$$

The size H'' of the output of convolution transpose is obtained in section 5.3.

6.3 Spatial pooling

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:

$$\boxed{\operatorname{vec} \mathbf{y} = S(\mathbf{x}) \operatorname{vec} \mathbf{x}, \qquad \frac{dz}{d \operatorname{vec} \mathbf{x}} = S(\mathbf{x})^\top \frac{dz}{d \operatorname{vec} \mathbf{y}}.} \tag{6.2}$$

6.4 Activation functions

6.4.1 ReLU

The ReLU operator can be expressed in matrix notation as

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

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

6.4.2 Sigmoid

The derivative of the sigmoid function 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:

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

6.5 Normalization

6.5.1 Local response normalization (LRN)

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

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

6.5.2 Batch normalization

The derivative of the of the network output z with respect to the multipliers w_k and biases b_k is given by

$$\begin{aligned}\frac{dz}{dw_k} &= \sum_{i''j''k''t''} \frac{dz}{dy_{i''j''k''t''}} \frac{dy_{i''j''k''t''}}{dw_k} = \sum_{i''j''t''} \frac{dz}{dy_{i''j''kt''}} \frac{x_{i''j''kt''} - \mu_k}{\sqrt{\sigma_k^2+\epsilon}}, \\ \frac{dz}{db_k} &= \sum_{i''j''k''t''} \frac{dz}{dy_{i''j''k''t''}} \frac{dy_{i''j''k''t''}}{dw_k} = \sum_{i''j''t''} \frac{dz}{dy_{i''j''kt''}}.\end{aligned}$$

The derivative of the network output z with respect to the block input x 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.

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

We can identify some of these terms with the ones computed as derivatives of bnorm with respect to w_k and μ_k:

$$\frac{dz}{dx_{ijkt}} = \frac{w_k}{\sqrt{\sigma_k^2 + \epsilon}} \left(\frac{dz}{dy_{ijkt}} - \frac{1}{HWT} \frac{dz}{db_k} - \frac{x_{ijkt} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} \frac{1}{HWT} \frac{dz}{dw_k} \right)$$

6.5.3 Spatial normalization

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.

6.5.4 Softmax

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.

6.6 Losses and comparisons

6.6.1 Log-loss

The derivative is

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

6.6.2 Softmax log-loss

The derivative is given by

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

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.

6.6.3 p-distance

The derivative of the operator without root is given by:

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

The derivative of the operator with root is given by:

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

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

Bibliography

[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] 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] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

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

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

[11] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. ICLR*, 2015.

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

[13] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Proc. ECCV*, 2014.