

MatConvNet
Convolutional Neural Networks for MATLAB

Andrea Vedaldi Karel Lenc Ankush Gupta

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 to MatConvNet	1
1.1	Getting started	2
1.2	MATCONVNET at a glance	4
1.3	Documentation and examples	5
1.4	Speed	6
1.5	Acknowledgments	7
2	Neural Network Computations	9
2.1	Overview	9
2.2	Network structures	10
2.2.1	Sequences	10
2.2.2	Directed acyclic graphs	11
2.3	Computing derivatives with backpropagation	12
2.3.1	Derivatives of tensor functions	12
2.3.2	Derivatives of function compositions	13
2.3.3	Backpropagation networks	14
2.3.4	Backpropagation in DAGs	15
2.3.5	DAG backpropagation networks	18
3	Wrappers and pre-trained models	21
3.1	Wrappers	21
3.1.1	SimpleNN	21
3.1.2	DagNN	21
3.2	Pre-trained models	22
3.3	Learning models	23
3.4	Running large scale experiments	23
3.5	Reading images	23
4	Computational blocks	27
4.1	Convolution	27
4.2	Convolution transpose (deconvolution)	29
4.3	Spatial pooling	31
4.4	Activation functions	32
4.5	Spatial bilinear resampling	32
4.6	Region of interest pooling	32

4.7	Normalization	33
4.7.1	Local response normalization (LRN)	33
4.7.2	Batch normalization	33
4.7.3	Spatial normalization	34
4.7.4	Softmax	34
4.8	Categorical losses	35
4.8.1	Classification losses	35
4.8.2	Attribute losses	36
4.9	Comparisons	38
4.9.1	p -distance	38
5	Geometry	39
5.1	Preliminaries	39
5.2	Simple filters	40
5.2.1	Pooling in Caffe	40
5.3	Convolution transpose	42
5.4	Transposing receptive fields	43
5.5	Composing receptive fields	44
5.6	Overlaying receptive fields	44
6	Implementation details	45
6.1	Convolution	45
6.2	Convolution transpose	46
6.3	Spatial pooling	47
6.4	Activation functions	47
6.4.1	ReLU	47
6.4.2	Sigmoid	48
6.5	Spatial bilinear resampling	48
6.6	Normalization	48
6.6.1	Local response normalization (LRN)	48
6.6.2	Batch normalization	49
6.6.3	Spatial normalization	50
6.6.4	Softmax	50
6.7	Categorical losses	51
6.7.1	Classification losses	51
6.7.2	Attribute losses	51
6.8	Comparisons	52
6.8.1	p -distance	52
6.9	Other implementation details	52
6.9.1	Normal sampler	52
6.9.2	Euclid's algorithm	54
	Bibliography	55

Chapter 1

Introduction to MatConvNet

MATCONVNET is a MATLAB toolbox implementing *Convolutional Neural Networks* (CNN) for computer vision applications. Since the breakthrough work of [8], 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 [13] 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 Caffe⁴. 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>

⁵While 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 [8] and the very deep networks of [11] from millions of images. Pre-trained versions of several of these powerful models can be downloaded from the MATCONVNET home page⁷. 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 fig. 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.⁸ These features make MATCONVNET useful in an educational context too.⁹

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.¹⁰

1.1 Getting started

MATCONVNET is simple to install and use. fig. 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-science/>

⁷<http://www.vlfeat.org/matconvnet/>

⁸<http://www.vlfeat.org/matconvnet/matconvnet-manual.pdf>

⁹An example laboratory experience based on MATCONVNET can be downloaded from <http://www.robots.ox.ac.uk/~vgg/practicals/cnn/index.html>.

¹⁰<http://www.github.com/matconvnet>

```

% install and compile MatConvNet (run once)
untar(['http://www.vlfeat.org/matconvnet/download/' ...
      'matconvnet-1.0-beta25.tar.gz']);
cd matconvnet-1.0-beta25
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));

```

bell pepper (946), score 0.704




Figure 1.1: A complete example including download, installing, compiling and running MAT-CONVNET to classify one of MATLAB stock images using a large CNN pre-trained on ImageNet.

image with the filters by using the command $y = \text{vl_nnconv}(x, f, [])$. 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 [8] or Network-in-Network [9]. Furthermore, the library provides numerous examples (in the `examples/` subdirectory), including code to learn a variety of models on the MNIST, CIFAR, and ImageNet datasets. All these examples use the `examples/cnn_train` training code, which is an implementation of stochastic gradient descent (section 3.3). 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 MATLAB commands. These building blocks are easy to combine into 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 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 $y = \text{vl_nnconv}(x, f, b)$ where x is an image, f a filter bank, and b a vector of biases (section 4.1). The derivatives are computed as $[dzdx, dzdf, dzdb] = \text{vl_nnconv}(x, f, b, dzdy)$ 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 provides a much more flexible wrapper supporting networks with arbitrary topologies, encapsulated in the `daggn.DagNN` MATLAB class.
- *Example applications.* MATCONVNET provides several examples 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.

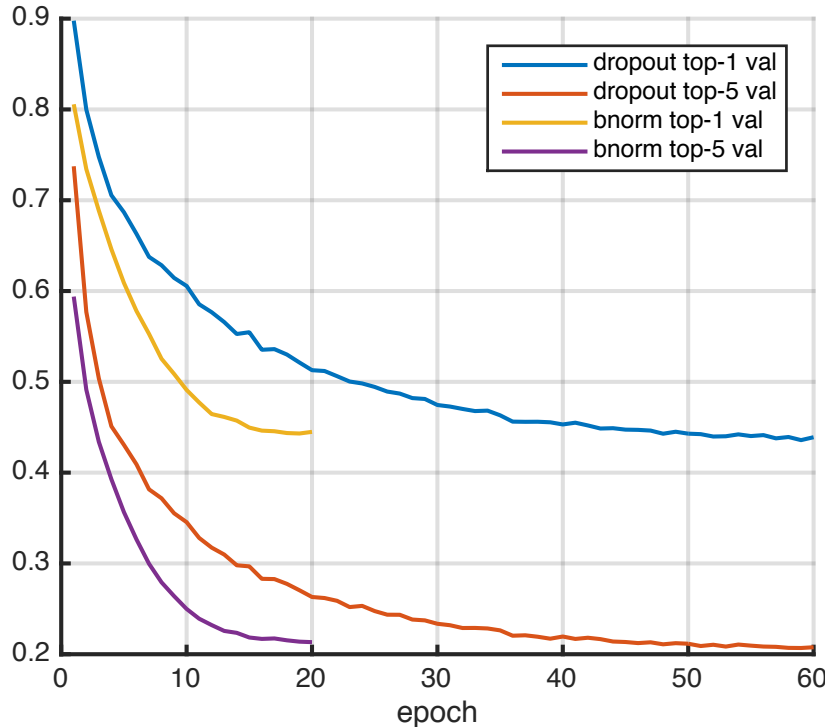


Figure 1.2: Training AlexNet on ImageNet ILSVRC: dropout vs batch normalisation.

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.¹¹ 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, fig. 1.2 illustrates the top-1 and top-5 validation errors as a model similar to AlexNet [8] is trained using either standard dropout regularisation or the recent *batch normalisation* technique of [4]. 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 [8], VGG-S, VGG-M, VGG-S [1], and VGG-VD-16, and VGG-VD-19 [12]. The example code of fig. 1.1 shows how one such model can be used in a few lines of MATLAB code.

¹¹See 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 with 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 Acknowledgments

MATCONVNET is a community project, and as such acknowledgements go 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 [6] (however, Caffe is *not* a dependency). Several of the example networks have been trained by Karen Simonyan as part of [1] and [12].

Chapter 2

Neural Network Computations

This chapter provides a brief introduction to the computational aspects of neural networks, and convolutional neural networks in particular, emphasizing the concepts required to understand and use `MATCONVNET`.

2.1 Overview

A *Neural Network* (NN) is a function g mapping data \mathbf{x} , for example an image, to an output vector \mathbf{y} , for example an image label. The function $g = f_L \circ \dots \circ f_1$ is the composition of a sequence of simpler functions f_l , which are called *computational blocks* or *layers*. Let $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_L$ be the outputs of each layer in the network, and let $\mathbf{x}_0 = \mathbf{x}$ denote the network input. Each intermediate 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 .

In a *Convolutional Neural Network* (CNN), the data has a spatial structure: each $\mathbf{x}_l \in \mathbb{R}^{H_l \times W_l \times C_l}$ is a 3D array or *tensor* where the first two dimensions H_l (height) and W_l (width) are interpreted as spatial dimensions. The third dimension C_l is instead interpreted as the *number of feature channels*. Hence, the tensor \mathbf{x}_l represents a $H_l \times W_l$ field of C_l -dimensional feature vectors, one for each spatial location. A fourth dimension N_l in the tensor spans multiple data samples packed in a single *batch* for efficiency parallel processing. The number of data samples N_l in a batch is called the batch *cardinality*. The network is called *convolutional* because the functions f_l are local and translation invariant operators (i.e. non-linear filters) like linear convolution.

It is also possible to conceive CNNs with more than two spatial dimensions, where the additional dimensions may represent volume or time. In fact, there are little *a-priori* restrictions on the format of data in neural networks in general. Many useful NNs contain a mixture of convolutional layers together with layer that process other data types such as text strings, or perform other operations that do not strictly conform to the CNN assumptions.

`MATCONVNET` includes a variety of layers, contained in the `matlab/` directory, such as `v1_nnconv` (convolution), `v1_nnconvt` (convolution transpose or deconvolution), `v1_nnpool` (max and average pooling), `v1_nnrelu` (ReLU activation), `v1_nnsigmoid` (sigmoid activation), `v1_nnsoftmax` (softmax operator), `v1_nnloss` (classification log-loss), `v1_nnbnorm` (batch normalization), `v1_nnspsnorm` (spatial normalization), `v1_nnnormalize` (local response normal-

ization – LRN), or `vl_nnpsdist` (p -distance). There are enough layers to implement many interesting state-of-the-art networks out of the box, or even import them from other tool-boxes such as Caffe.

NNs are often used as classifiers or regressors. In the example of fig. 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 minimize this loss averaged over a large dataset of labelled example images.

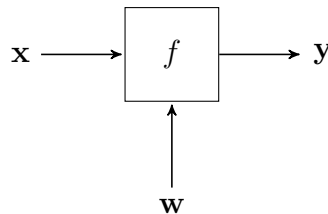
Learning generally uses a variant of *stochastic gradient descent* (SGD). While this is an efficient method (for this type of problems), networks may contain several million parameters and need to be trained on millions of images; thus, efficiency is a paramount in MATLAB design, as further discussed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

2.2 Network structures

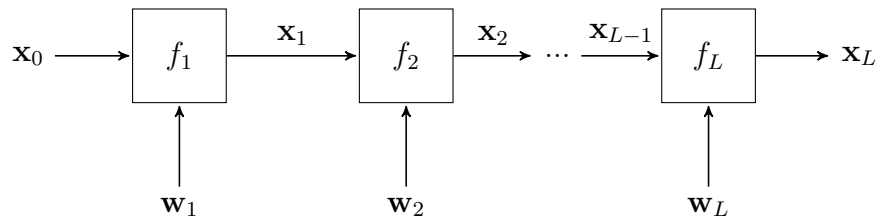
In the simplest case, layers in a NN are arranged in a sequence; however, more complex interconnections are possible as well, and in fact very useful in many cases. This section discusses such configurations and introduces a graphical notation to visualize them.

2.2.1 Sequences

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



As seen above, in the simplest case blocks are chained in a sequence $f_1 \rightarrow f_2 \rightarrow \dots \rightarrow f_L$ yielding the structure:



Given an input \mathbf{x}_0 , evaluating the network is a simple matter of evaluating all the blocks from left to right, which defines a composite function $\mathbf{x}_L = f(\mathbf{x}_0; \mathbf{w}_1, \dots, \mathbf{w}_L)$.

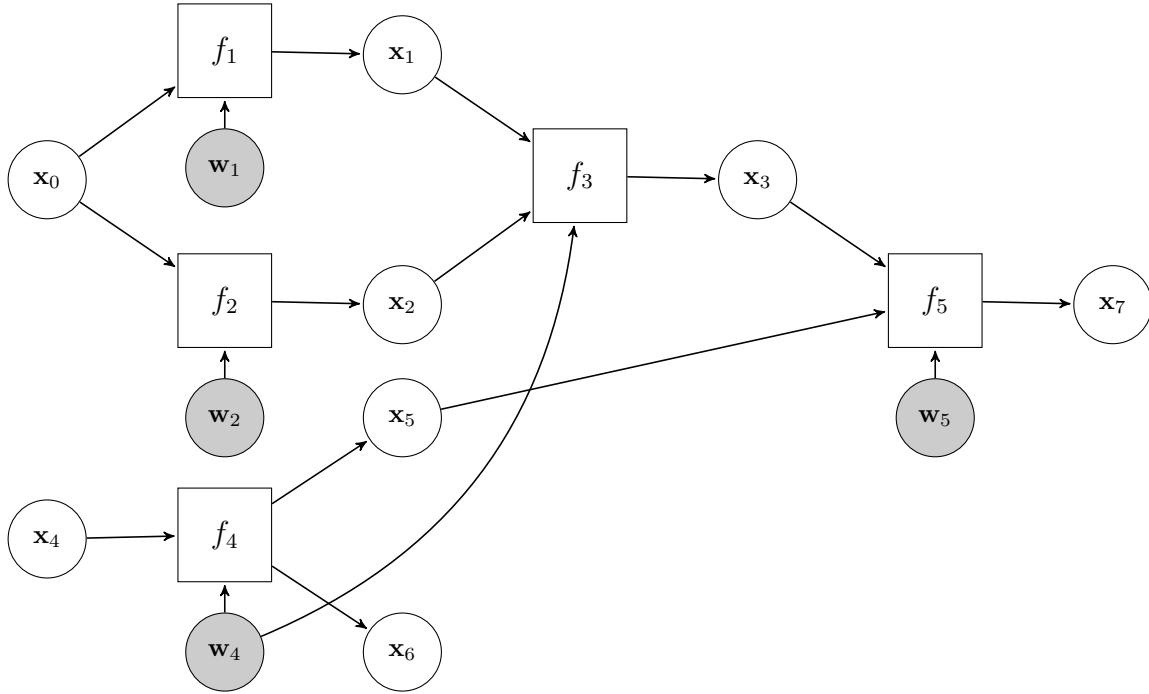


Figure 2.1: Example DAG.

2.2.2 Directed acyclic graphs

One is not limited to chaining layers one after another. In fact, the only requirement for evaluating a NN is that, when a layer has to be evaluated, all its input have been evaluated prior to it. This is possible exactly when the interconnections between layers form a *directed acyclic graph*, or DAG for short.

In order to visualize DAGs, it is useful to introduce additional nodes for the network variables, as in the example of Fig. 2.1. Here boxes denote functions and circles denote variables (parameters are treated as a special kind of variables). In the example, x_0 and x_4 are the inputs of the CNN and x_6 and 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 from circles to boxes.
2. Functions can have any number of inputs or outputs; variables and parameters can have an arbitrary number of outputs (a parameter with more of one output is *shared* between different layers); variables have at most one input and parameters none.
3. Variables with no incoming arrows and parameters are not computed by the network, but must be set prior to evaluation, i.e. they are *inputs*. Any variable (or even parameter) may be used as output, although these are usually the variables with no outgoing arrows.

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 the functions in the order f_1, f_2, f_3, f_4, f_5 would work).

2.3 Computing derivatives with backpropagation

Learning a NN requires computing the derivative of the loss with respect to the network parameters. Derivatives are computed using an algorithm called *backpropagation*, which is a memory-efficient implementation of the chain rule for derivatives. First, we discuss the derivatives of a single layer, and then of a whole network.

2.3.1 Derivatives of tensor functions

In a CNN, a layer is a function $\mathbf{y} = f(\mathbf{x})$ where both input $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ and output $\mathbf{y} \in \mathbb{R}^{H' \times W' \times C'}$ are tensors. The derivative of the function f contains the derivative of each output component $y_{i'j'k'}$ with respect to each input component x_{ijk} , for a total of $H' \times W' \times C' \times H \times W \times C$ elements naturally arranged in a 6D tensor. Instead of expressing derivatives as tensors, it is often useful to switch to a matrix notation by *stacking* the input and output tensors into vectors. This is done by the *vec* operator, which visits each element of a tensor in lexicographical order and produces a vector:

$$\text{vec } \mathbf{x} = \begin{bmatrix} x_{111} \\ x_{211} \\ \vdots \\ x_{H11} \\ x_{121} \\ \vdots \\ x_{HWC} \end{bmatrix}.$$

By stacking both input and output, each layer f can be seen reinterpreted as vector function $\text{vec } f$, whose derivative is the conventional Jacobian matrix:

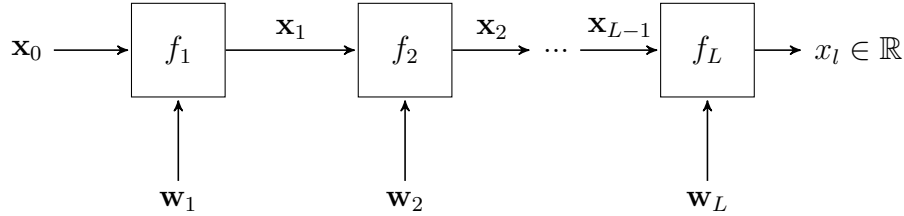
$$\frac{d \text{vec } f}{d(\text{vec } \mathbf{x})^\top} = \begin{bmatrix} \frac{\partial y_{111}}{\partial x_{111}} & \frac{\partial y_{111}}{\partial x_{211}} & \cdots & \frac{\partial y_{111}}{\partial x_{H11}} & \frac{\partial y_{111}}{\partial x_{121}} & \cdots & \frac{\partial y_{111}}{\partial x_{HWC}} \\ \frac{\partial y_{211}}{\partial x_{111}} & \frac{\partial y_{211}}{\partial x_{211}} & \cdots & \frac{\partial y_{211}}{\partial x_{H11}} & \frac{\partial y_{211}}{\partial x_{121}} & \cdots & \frac{\partial y_{211}}{\partial x_{HWC}} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ \frac{\partial y_{H'11}}{\partial x_{111}} & \frac{\partial y_{H'11}}{\partial x_{211}} & \cdots & \frac{\partial y_{H'11}}{\partial x_{H11}} & \frac{\partial y_{H'11}}{\partial x_{121}} & \cdots & \frac{\partial y_{H'11}}{\partial x_{HWC}} \\ \frac{\partial y_{121}}{\partial x_{111}} & \frac{\partial y_{121}}{\partial x_{211}} & \cdots & \frac{\partial y_{121}}{\partial x_{H11}} & \frac{\partial y_{121}}{\partial x_{121}} & \cdots & \frac{\partial y_{121}}{\partial x_{HWC}} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ \frac{\partial y_{H'W'C'}}{\partial x_{111}} & \frac{\partial y_{H'W'C'}}{\partial x_{211}} & \cdots & \frac{\partial y_{H'W'C'}}{\partial x_{H11}} & \frac{\partial y_{H'W'C'}}{\partial x_{121}} & \cdots & \frac{\partial y_{H'W'C'}}{\partial x_{HWC}} \end{bmatrix}.$$

This notation for the derivatives of tensor functions is taken from [7] and is used throughout this document.

While it is easy to express the derivatives of tensor functions as matrices, these matrices are in general extremely large. Even for moderate data sizes (e.g. $H = H' = W = W' = 32$ and $C = C' = 128$), there are $H'W'C'HWC \approx 17 \times 10^9$ elements in the Jacobian. Storing that requires 68 GB of space in single precision. The purpose of the backpropagation algorithm is to compute the derivatives required for learning without incurring this huge memory cost.

2.3.2 Derivatives of function compositions

In order to understand backpropagation, consider first a simple CNN terminating in a loss function $f_L = \ell_y$:



The goal is to compute the gradient of the loss value x_L (output) with respect to each network parameter \mathbf{w}_l :

$$\frac{df}{d(\text{vec } \mathbf{w}_l)^\top} = \frac{d}{d(\text{vec } \mathbf{w}_l)^\top} [f_L(\cdot; \mathbf{w}_L) \circ \dots \circ f_2(\cdot; \mathbf{w}_2) \circ f_1(\mathbf{x}_0; \mathbf{w}_1)].$$

By applying the chain rule and by using the matrix notation introduced above, the derivative can be written as

$$\frac{df}{d(\text{vec } \mathbf{w}_l)^\top} = \frac{d \text{vec } f_L(\mathbf{x}_{L-1}; \mathbf{w}_L)}{d(\text{vec } \mathbf{x}_{L-1})^\top} \times \dots \times \frac{d \text{vec } f_{l+1}(\mathbf{x}_l; \mathbf{w}_{l+1})}{d(\text{vec } \mathbf{x}_l)^\top} \times \frac{d \text{vec } f_l(\mathbf{x}_{l-1}; \mathbf{w}_l)}{d(\text{vec } \mathbf{w}_l^\top)} \quad (2.1)$$

where the derivatives are computed at the working point determined by the input \mathbf{x}_0 and the current value of the parameters.

Note that, since the network output x_l is a *scalar* quantity, the target derivative $df/d(\text{vec } \mathbf{w}_l)^\top$ has the same number of elements of the parameter vector \mathbf{w}_l , which is moderate. However, the intermediate Jacobian factors have, as seen above, an unmanageable size. In order to avoid computing these factor explicitly, we can proceed as follows.

Start by multiplying the output of the last layer by a tensor $p_L = 1$ (note that this tensor is a scalar just like the variable x_L):

$$\begin{aligned} p_L \times \frac{df}{d(\text{vec } \mathbf{w}_l)^\top} &= p_L \times \underbrace{\frac{d \text{vec } f_L(\mathbf{x}_{L-1}; \mathbf{w}_L)}{d(\text{vec } \mathbf{x}_{L-1})^\top}}_{(\text{vec } \mathbf{p}_{L-1})^\top} \times \dots \times \frac{d \text{vec } f_{l+1}(\mathbf{x}_l; \mathbf{w}_{l+1})}{d(\text{vec } \mathbf{x}_l)^\top} \times \frac{d \text{vec } f_l(\mathbf{x}_{l-1}; \mathbf{w}_l)}{d(\text{vec } \mathbf{w}_l^\top)} \\ &= (\text{vec } \mathbf{p}_{L-1})^\top \times \dots \times \frac{d \text{vec } f_{l+1}(\mathbf{x}_l; \mathbf{w}_{l+1})}{d(\text{vec } \mathbf{x}_l)^\top} \times \frac{d \text{vec } f_l(\mathbf{x}_{l-1}; \mathbf{w}_l)}{d(\text{vec } \mathbf{w}_l^\top)} \end{aligned}$$

In the second line the last two factors to the left have been multiplied obtaining a new tensor \mathbf{p}_{L-1} that has the same size as the variable \mathbf{x}_{L-1} . The factor \mathbf{p}_{L-1} can therefore be

explicitly stored. The construction is then repeated by multiplying pairs of factors from left to right, obtaining a sequence of tensors $\mathbf{p}_{L-2}, \dots, \mathbf{p}_l$ until the desired derivative is obtained. Note that, in doing so, no large tensor is ever stored in memory. This process is known as *backpropagation*.

In general, tensor \mathbf{p}_l is obtained from \mathbf{p}_{l+1} as the product:

$$(\text{vec } \mathbf{p}_l)^\top = (\text{vec } \mathbf{p}_{l+1})^\top \times \frac{d \text{vec } f_{l+1}(\mathbf{x}_l; \mathbf{w}_{l+1})}{d(\text{vec } \mathbf{x}_l)^\top}.$$

The key to implement backpropagation is to be able to compute these products without explicitly computing and storing in memory the second factor, which is a large Jacobian matrix. Since computing the derivative is a linear operation, this product can be interpreted as the *derivative of the layer projected along direction \mathbf{p}_{l+1}* :

$$\mathbf{p}_l = \frac{d\langle \mathbf{p}_{l+1}, f(\mathbf{x}_l; \mathbf{w}_l) \rangle}{d\mathbf{x}_l}. \quad (2.2)$$

Here $\langle \cdot, \cdot \rangle$ denotes the inner product between tensors, which results in a scalar quantity. Hence the derivative (2.2) needs not to use the vec notation, and yields a tensor \mathbf{p}_l that has the same size as \mathbf{x}_l as expected.

In order to implement backpropagation, a CNN toolbox provides implementations of each layer f that provide:

- A **forward mode**, computing the output $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$ of the layer given its input \mathbf{x} and parameters \mathbf{w} .
- A **backward mode**, computing the projected derivatives

$$\frac{d\langle \mathbf{p}, f(\mathbf{x}; \mathbf{w}) \rangle}{d\mathbf{x}} \quad \text{and} \quad \frac{d\langle \mathbf{p}, f(\mathbf{x}; \mathbf{w}) \rangle}{d\mathbf{w}},$$

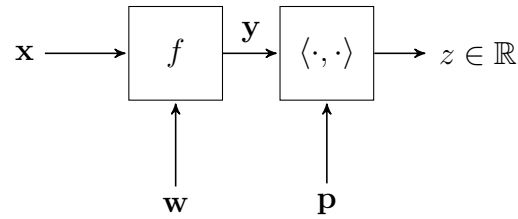
given, in addition to the input \mathbf{x} and parameters \mathbf{w} , a tensor \mathbf{p} that the same size as \mathbf{y} .

This is best illustrated with an example. Consider a layer f such as the convolution operator implemented by the MATCONVNET `v1_nnconv` command. In the “forward” mode, one calls the function as $\mathbf{y} = \text{v1_nnconv}(\mathbf{x}, \mathbf{w}, [])$ to apply the filters \mathbf{w} to the input \mathbf{x} and obtain the output \mathbf{y} . In the “backward mode”, one calls $[d\mathbf{x}, d\mathbf{w}] = \text{v1_nnconv}(\mathbf{x}, \mathbf{w}, [], \mathbf{p})$. As explained above, $d\mathbf{x}$, $d\mathbf{w}$, and \mathbf{p} have the same size as \mathbf{x} , \mathbf{w} , and \mathbf{y} , respectively. The computation of large Jacobian is encapsulated in the function call and never carried out explicitly.

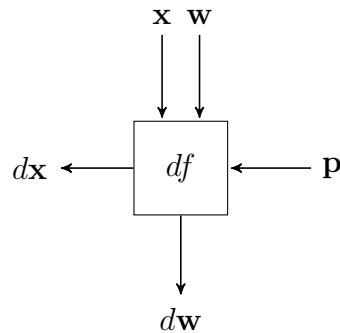
2.3.3 Backpropagation networks

In this section, we provide a schematic interpretation of backpropagation and show how it can be implemented by “reversing” the NN computational graph.

The projected derivative of eq. (2.2) can be seen as the derivative of the following mini-network:

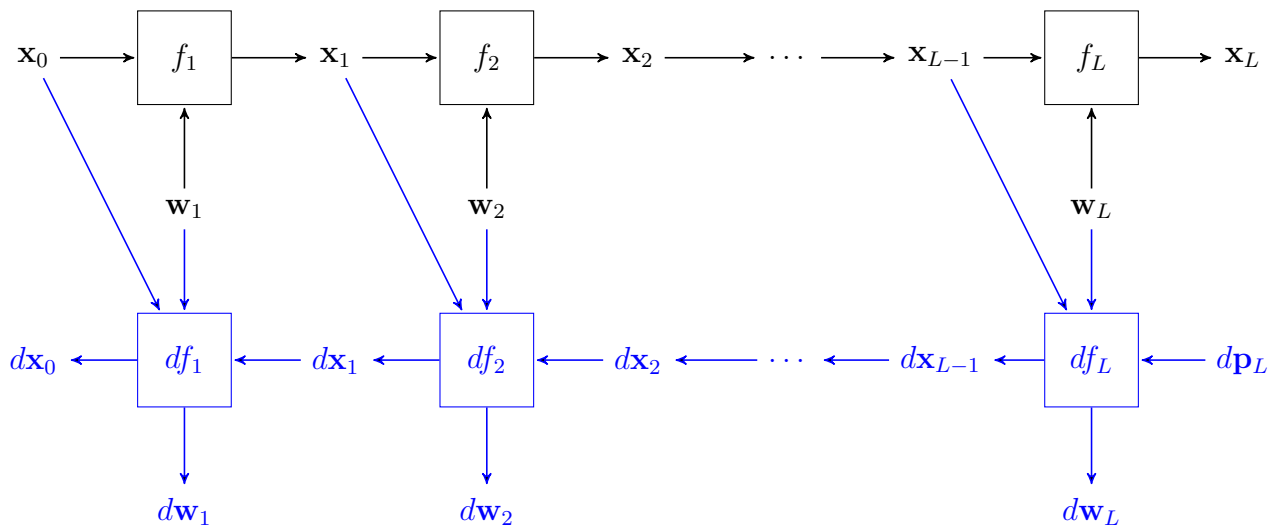


In the context of back-propagation, it can be useful to think of the projection \mathbf{p} as the “linearization” of the rest of the network from variable \mathbf{y} down to the loss. The projected derivative can also be thought of as a new layer $(d\mathbf{x}, d\mathbf{w}) = df(\mathbf{x}, \mathbf{w}, \mathbf{p})$ that, by computing the derivative of the mini-network, operates in the reverse direction:



By construction (see eq. (2.2)), the function df is *linear* in the argument \mathbf{p} .

Using this notation, the forward and backward passes through the original network can be rewritten as evaluating an extended network which contains a BP-reverse of the original one (in blue in the diagram):



2.3.4 Backpropagation in DAGs

Assume that the DAG has a single output variable \mathbf{x}_L and assume, without loss of generality, that all variables are sorted in order of computation $(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{L-1}, \mathbf{x}_L)$ according to the

DAG structure. Furthermore, in order to simplify the notation, assume that this list contains both data and parameter variables, as the distinction is moot for the discussion in this section.

We can cut the DAG at any point in the sequence by fixing $\mathbf{x}_0, \dots, \mathbf{x}_{l-1}$ to some arbitrary value and dropping all the DAG layers that feed into them, effectively transforming the first l variables into inputs. Then, the rest of the DAG defines a function h_l that maps these input variables to the output \mathbf{x}_L :

$$\mathbf{x}_L = h_l(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{l-1}).$$

Next, we show that backpropagation in a DAG iteratively computes the projected derivatives of all functions h_1, \dots, h_L with respect to all their parameters.

Backpropagation starts by initializing variables $(d\mathbf{x}_0, \dots, d\mathbf{x}_{l-1})$ to null tensors of the same size as $(\mathbf{x}_0, \dots, \mathbf{x}_{l-1})$. Next, it computes the projected derivatives of

$$\mathbf{x}_L = h_L(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{L-1}) = f_{\pi_L}(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{L-1}).$$

Here π_l denotes the index of the layer f_{π_l} that computes the value of the variable \mathbf{x}_l . There is at most one such layer, or none if \mathbf{x}_l is an input or parameter of the original NN. In the first case, the layer may depend on any of the variables prior to \mathbf{x}_l in the sequence, so that general one has:

$$\mathbf{x}_l = f_{\pi_l}(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}).$$

At the beginning of backpropagation, since there are no intermediate variables between \mathbf{x}_{L-1} and \mathbf{x}_L , the function h_L is the same as the last layer f_{π_L} . Thus the projected derivatives of h_L are the same as the projected derivatives of f_{π_L} , resulting in the equation

$$\forall t = 0, \dots, L-1 : \quad d\mathbf{x}_t \leftarrow d\mathbf{x}_t + \frac{d\langle \mathbf{p}_L, f_{\pi_L}(\mathbf{x}_0, \dots, \mathbf{x}_{t-1}) \rangle}{d\mathbf{x}_t}.$$

Here, for uniformity with the other iterations, we use the fact that $d\mathbf{x}_l$ are initialized to zero and *accumulate* the values instead of storing them. In practice, the update operation needs to be carried out only for the variables \mathbf{x}_l that are actual inputs to f_{π_L} , which is often a tiny fraction of all the variables in the DAG.

After the update, each $d\mathbf{x}_t$ contains the projected derivative of function h_L with respect to the corresponding variable:

$$\forall t = 0, \dots, L-1 : \quad d\mathbf{x}_t = \frac{d\langle \mathbf{p}_L, h_L(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}) \rangle}{d\mathbf{x}_t}.$$

Given this information, the next iteration of backpropagation updates the variables to contain the projected derivatives of h_{L-1} instead. In general, given the derivatives of h_{l+1} , backpropagation computes the derivatives of h_l by using the relation

$$\mathbf{x}_L = h_l(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{l-1}) = h_{l+1}(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{l-1}, f_{\pi_L}(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}))$$

Applying the chain rule to this expression, for all $0 \leq t \leq l-1$:

$$\frac{d\langle \mathbf{p}, h_l \rangle}{d(\text{vec } \mathbf{x}_t)^\top} = \frac{d\langle \mathbf{p}, h_{l+1} \rangle}{d(\text{vec } \mathbf{x}_t)^\top} + \underbrace{\frac{d\langle \mathbf{p}_L, h_{l+1} \rangle}{d(\text{vec } \mathbf{x}_l)^\top}}_{\text{vec } d\mathbf{x}_l} \frac{d \text{vec } f_{\pi_L}}{d(\text{vec } \mathbf{x}_t)^\top}.$$

This yields the update equation

$$\forall t = 0, \dots, l - 1 : \quad d\mathbf{x}_t \leftarrow d\mathbf{x}_t + \frac{d\langle \mathbf{p}_l, f_{\pi_l}(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}) \rangle}{d\mathbf{x}_t}, \quad \text{where } \mathbf{p}_l = d\mathbf{x}_l. \quad (2.3)$$

Once more, the update needs to be explicitly carried out only for the variables \mathbf{x}_t that are actual inputs of f_{π_l} . In particular, if \mathbf{x}_l is a data input or a parameter of the original neural network, then \mathbf{x}_l does not depend on any other variables or parameters and f_{π_l} is a nullary function (i.e. a function with no arguments). In this case, the update does not do anything. After iteration $L - l + 1$ completes, backpropagation remains with:

$$\forall t = 0, \dots, l - 1 : \quad d\mathbf{x}_t = \frac{d\langle \mathbf{p}_L, h_l(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}) \rangle}{d\mathbf{x}_t}.$$

Note that the derivatives for variables $\mathbf{x}_t, l \leq t \leq L - 1$ are not updated since h_l does not depend on any of those. Thus, after all L iterations are complete, backpropagation terminates with

$$\forall l = 1, \dots, L : \quad d\mathbf{x}_{l-1} = \frac{d\langle \mathbf{p}_L, h_l(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}) \rangle}{d\mathbf{x}_{l-1}}.$$

As seen above, functions h_l are obtained from the original network f by transforming variables $\mathbf{x}_0, \dots, \mathbf{x}_{l-1}$ into to inputs. If \mathbf{x}_{l-1} was already an input (data or parameter) of f , then the derivative $d\mathbf{x}_{l-1}$ is applicable to f as well.

Backpropagation can be summarized as follows:

Given: a DAG neural network f with a single output \mathbf{x}_L , the values of all input variables (including the parameters), and the value of the projection \mathbf{p}_L (usually \mathbf{x}_L is a scalar and $\mathbf{p}_L = p_L = 1$):

1. Sort all variables by computation order ($\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_L$) according to the DAG.
2. Perform a forward pass through the network to compute all the intermediate variable values.
3. Initialize ($d\mathbf{x}_0, \dots, d\mathbf{x}_{L-1}$) to null tensors with the same size as the corresponding variables.
4. For $l = L, L - 1, \dots, 2, 1$:
 - a) Find the index π_l of the layer $\mathbf{x}_l = f_{\pi_l}(\mathbf{x}_0, \dots, \mathbf{x}_{l-1})$ that evaluates variable \mathbf{x}_l . If there is no such layer (because \mathbf{x}_l is an input or parameter of the network), go to the next iteration.
 - b) Update the variables using the formula:

$$\forall t = 0, \dots, l - 1 : \quad d\mathbf{x}_t \leftarrow d\mathbf{x}_t + \frac{d\langle d\mathbf{x}_l, f_{\pi_l}(\mathbf{x}_0, \dots, \mathbf{x}_{l-1}) \rangle}{d\mathbf{x}_t}.$$

To do so efficiently, use the “backward mode” of the layer f_{π_l} to compute its derivative projected onto $d\mathbf{x}_l$ as needed.

2.3.5 DAG backpropagation networks

Just like for sequences, backpropagation in DAGs can be implemented as a corresponding BP-reversed DAG. To construct the reversed DAG:

1. For each layer f_l , and variable/parameter \mathbf{x}_t and \mathbf{w}_l , create a corresponding layer df_l and variable/parameter $d\mathbf{x}_t$ and $d\mathbf{w}_l$.
2. If a variable \mathbf{x}_t (or parameter \mathbf{w}_l) is an input of f_l , then it is an input of df_l as well.
3. If a variable \mathbf{x}_t (or parameter \mathbf{w}_l) is an input of f_l , then the variable $d\mathbf{x}_t$ (or the parameter $d\mathbf{w}_l$) is an output df_l .
4. In the previous step, if a variable \mathbf{x}_t (or parameter \mathbf{w}_l) is input to two or more layers in f , then $d\mathbf{x}_t$ would be the output of two or more layers in the reversed network, which creates a conflict. Resolve these conflicts by inserting a summation layer that adds these contributions (this corresponds to the summation in the BP update equation (2.3)).

The BP network corresponding to the DAG of Fig. 2.1 is given in Fig. 2.2.

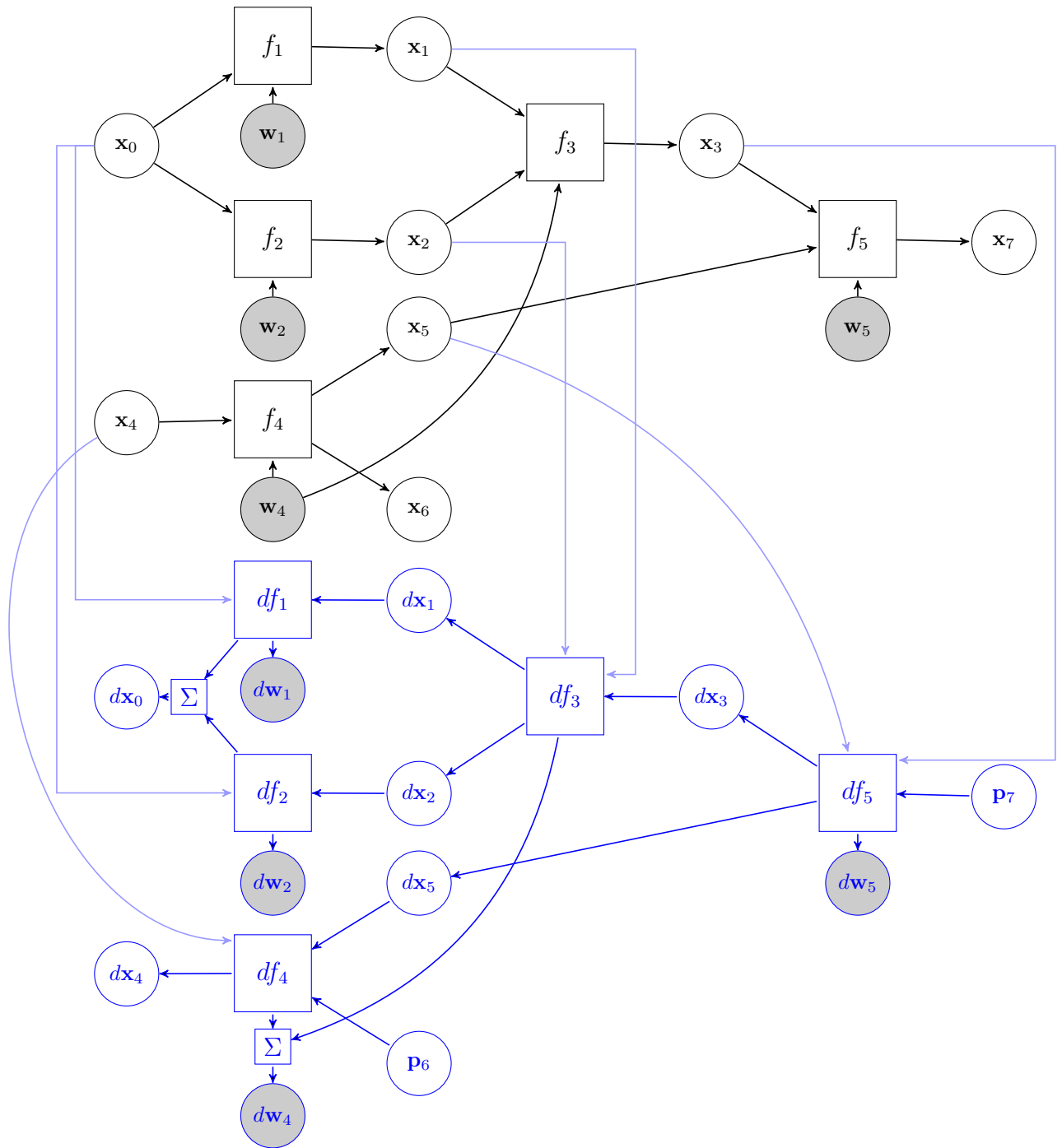


Figure 2.2: Backpropagation network for a DAG.

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 wrappers 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 blocks organized in more complex direct acyclic graphs (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 blocks can be used together and can serve as a basis for more complex implementations.

3.1.2 DagNN

The DagNN wrapper is more complex than 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 `dagmn.DagNN` class (under the `dagmn` 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 contains 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 backpropagation, 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 a height of 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 into 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.

3.5 Reading images

MATCONVNET provides the tool `v1_imreadjpeg` to quickly read images, transform them, and move them to the GPU.

Image cropping and scaling. Several options in `v1_imreadjpeg` control how images are cropped and rescaled. The procedure is as follows:

1. Given an input image of size $H \times W$, first the size of the output image $H_o \times W_o$ is determined. The *output size* is either equal the input size ($(H_o, W_o) = (H, W)$), equal to a specified constant size, or obtained by setting the minimum side equal to a specified constant and rescaling the other accordingly ($(H_o, W_o) = s(H, W)$, $s = \max\{H_o/H, W_o/W\}$).

2. Next, the *crop size* $H_c \times W_c$ is determined, starting from the *crop anisotropy* $a = (W_o/H_o)/(W_c/H_c)$, i.e. the relative change of aspect ratio from the crop to the output: $(H_c, W_c) \propto (H_o/a, aW_o)$. One option is to choose $a = (W/H)/(W_o/H_o)$ such that the crop has the same aspect ratio of the input image, which allows to squash a rectangular input into a square output. Another option is to sample it as $a \sim \mathcal{U}([a_-, a_+])$ where a_-, a_+ are, respectively, the minimum and maximum anisotropy.
3. The relative *crop scale* is determined by sampling a parameter $\rho \sim U([\rho_-, \rho_+])$ where ρ_-, ρ_+ are, respectively, the minimum and maximum relative crop sizes. The absolute maximum size is determined by the size of the input image. Overall, the shape of the crop is given by:

$$(H_c, W_c) = \rho(H_o/a, aW_o) \min\{aH/H_o, W/(aW_o)\}.$$

4. Given the crop size (H_c, W_c) , the crop is extracted relative to the input image either in the middle (center crop) or randomly shifted.
5. Finally, it is also possible to flip a crop left-to-right with a 50% probability.

In the simplest case, `vl_imreadjpeg` extract an image as is, without any processing. A standard center crop of 128 pixels can be obtained by setting $H_o = W_o = 128$, (`resize` option), $a_- = a_+ = 1$ (`CropAnisotropy` option), and $\rho_- = \rho_+ = 1$ (`CropSize` option). In the input image, this crop is isotropically stretched to fill either its width or height. If the input image is rectangular, such a crop can either slide horizontally or vertically (`CropLocation`), but not both. Setting $\rho_- = \rho_+ = 0.9$ makes the crop slightly smaller, allowing it to shift in both directions. Setting $\rho_- = 0.9$ and $\rho_+ = 1.0$ allows picking differently-sized crops each time. Setting $a_- = 0.9$ and $a_+ = 1.2$ allows the crops to be slightly elongated or widened.

Color post-processing. `vl_imreadjpeg` supports some basic colour postprocessing. It allows to subtract from all the pixels a constant shift $\boldsymbol{\mu} \in \mathbb{R}^3$ ($\boldsymbol{\mu}$ can also be a $H_o \times W_o$ image for fixed-sized crops). It also allows to add a random shift vector (sample independently for each image in a batch), and to also perturb randomly the saturation and contrast of the image. These transformations are discussed in detail next.

The brightness shift is a constant offset \mathbf{b} added to all pixels in the image, similarly to the vector $\boldsymbol{\mu}$, which is however subtracted and constant for all images in the batch. The shift is randomly sampled from a Gaussian distribution with standard deviation B . Here, $B \in \mathbb{R}^{3 \times 3}$ is the square root of the covariance matrix of the Gaussian, such that:

$$\mathbf{b} \leftarrow B\boldsymbol{\omega}, \quad \boldsymbol{\omega} \sim \mathcal{N}(0, I).$$

If $\mathbf{x}(u, v) \in \mathbb{R}^3$ is an RGB triplet at location (u, v) in the image, average color subtraction and brightness shift results in the transformation:

$$\mathbf{x}(u, v) \leftarrow \mathbf{x}(u, v) + \mathbf{b} - \boldsymbol{\mu}.$$

After this shift is applied, the image contrast is changed as follow:

$$\mathbf{x}(u, v) \leftarrow \gamma \mathbf{x}(u, v) + (1 - \gamma) \text{avg}_{uv}[\mathbf{x}(u, v)], \quad \gamma \sim \mathcal{U}([1 - C, 1 + C])$$

where the coefficient γ is uniformly sampled in the interval $[1 - C, 1 + C]$ where C is the contrast deviation coefficient. Note that, since γ can be larger than one, contrast can also be increased.

The last transformation changes the saturation of the image. This is controlled by the saturation deviation coefficient S :

$$\mathbf{x}(u, v) \leftarrow \sigma \mathbf{x}(u, v) + \frac{1 - \sigma}{3} \mathbf{1} \mathbf{1}^\top \mathbf{x}(u, v), \quad \sigma \sim \mathcal{U}([1 - S, 1 + S])$$

Overall, pixels are transformed as follows:

$$\mathbf{x}(u, v) \leftarrow \left(\sigma I + \frac{1 - \sigma}{3} \mathbf{1} \mathbf{1}^\top \right) \left(\gamma \mathbf{x}(u, v) + (1 - \gamma) \underset{uv}{\text{avg}}[\mathbf{x}(u, v)] + B\boldsymbol{\omega} - \boldsymbol{\mu} \right).$$

For grayscale images, changing the saturation does not do anything (unless ones applies first a colored shift, which effectively transforms a grayscale image into a color one).

Chapter 4

Computational blocks

This chapter 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 $\mathbf{y} = \text{vl_nn}\langle\text{block}\rangle(\mathbf{x}, \mathbf{w})$ that takes as input MATLAB arrays \mathbf{x} and \mathbf{w} representing the input data and parameters and returns an array \mathbf{y} as output. In general, \mathbf{x} and \mathbf{y} are 4D real arrays packing N maps or images, as discussed above, whereas \mathbf{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 \mathbf{dzdy} representing the derivative of the output of the network with respect to \mathbf{y} ; in this case, the function returns the derivatives $[\mathbf{dzdx}, \mathbf{dzdw}] = \text{vl_nn}\langle\text{block}\rangle(\mathbf{x}, \mathbf{w}, \mathbf{dzdy})$ with respect to the input data and parameters. The arrays \mathbf{dzdx} , \mathbf{dzdy} and \mathbf{dzdw} have the same dimensions of \mathbf{x} , \mathbf{y} and \mathbf{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 property-value pairs; some do not have parameters \mathbf{w} (e.g. a rectified linear unit); others can take multiple inputs and parameters, in which case there may be more than one \mathbf{x} , \mathbf{w} , \mathbf{dzdx} , \mathbf{dzdy} or \mathbf{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`. $\mathbf{y} = \text{vl_nnconv}(\mathbf{x}, \mathbf{f}, \mathbf{b})$ computes the convolution of the input map \mathbf{x} with a bank of K multi-dimensional filters \mathbf{f} and biases \mathbf{b} . Here

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

¹Other 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.

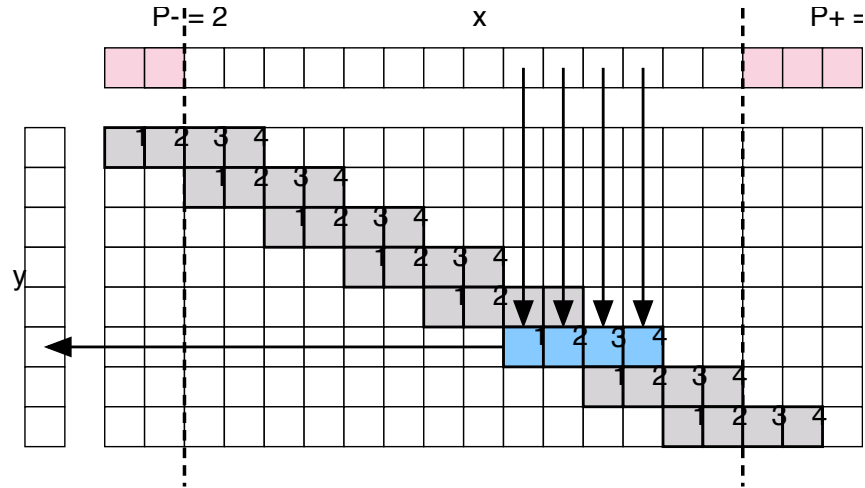


Figure 4.1: **Convolution.** The figure illustrates the process of filtering a 1D signal \mathbf{x} by a filter f to obtain a signal \mathbf{y} . The filter has $H' = 4$ elements and is applied with a stride of $S_h = 2$ samples. The purple areas represented padding $P_- = 2$ and $P_+ = 3$ which is zero-filled. 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 fig. 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 `v1_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. `v1_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. `v1_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 geometrically relate the indexes of the various array to the input data (usually images) in terms 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, *fully connected blocks or layers* are linear functions where each output dimension depends on all the input dimensions. MATCONVNET does not distinguish 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, `v1_nnconv` handles this case more efficiently when possible.

Filter groups. For additional flexibility, `v1_nnconv` allows to group channels of the input array \mathbf{x} and apply different subsets of filters to each group. 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 $d = 1, \dots, D'$ of the input \mathbf{x} ; the second group to dimensions $d = D' + 1, \dots, 2D'$ 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 [8] 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).

Dilation. `v1_nnconv` allows kernels to be spatially dilated on the fly by inserting zeros between elements. For instance, a dilation factor $d = 2$ causes the 1D kernel $[f_1, f_2]$ to be implicitly transformed in the kernel $[f_1, 0, 0, f_2]$. Thus, with dilation factors d_h, d_w , a filter of size (H_f, W_f) is equivalent to a filter of size:

$$H' = d_h(H_f - 1) + 1, \quad W' = d_w(W_f - 1) + 1.$$

With dilation, the convolution becomes:

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

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 `v1_nnconvt`.

In order to understand convolution transpose, let:

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

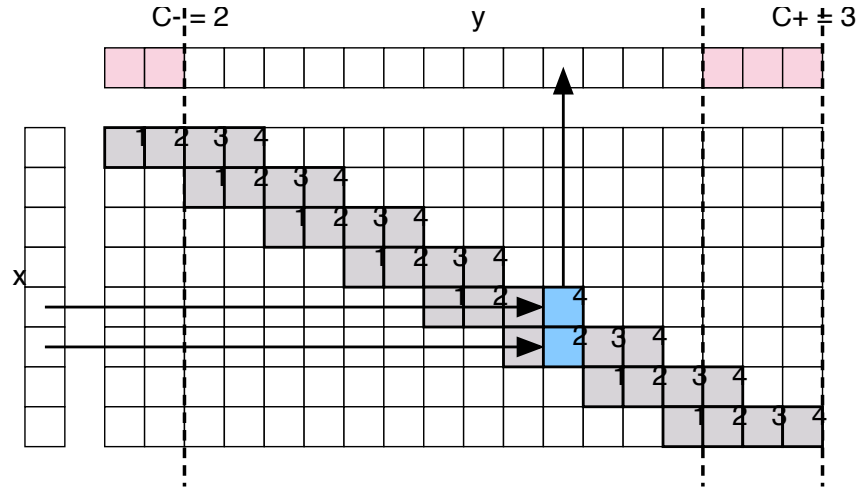


Figure 4.2: **Convolution transpose.** The figure illustrates the process of filtering a 1D signal x by a filter f to obtain a signal y . The filter is applied as a sliding-window, forming a pattern which is the transpose of the one of fig. 4.1. The filter has $H' = 4$ samples in total, although each filter application uses two of them (blue squares) in a circulant manner. The purple areas represent crops with $C_- = 2$ and $C_+ = 3$ which are discarded. The arrows exemplify which samples of x are involved in the calculation of a particular sample of y . Note that, differently from the forward convolution fig. 4.1, there is no need to add padding to the input array; instead, the convolution transpose filters can be seen as being applied with maximum input padding (more would result in zero output values), and the latter can be reduced by cropping the output instead.

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 $\text{vec } \mathbf{x} = M \text{vec } \mathbf{y}$; convolution transpose computes instead $\text{vec } \mathbf{y} = M^\top \text{vec } \mathbf{x}$. This process is illustrated for a 1D slice in fig. 4.2.

There are two important applications of convolution transpose. The first one is the so called *deconvolutional networks* [14] 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.

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

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

(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 an expression for the receptive field of the operator.

We now illustrate the action of convolution transpose in an example (see also fig. 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 chosen such that S_h divides $i'' - 1$; according to (4.1), this sample is obtained as a weighted summation of $x_{i''/S_h}, x_{i''/S_h-1}, \dots$ (note that the order is reversed). The weights are the filter elements $f_1, f_{S_h}, f_{2S_h}, \dots$ 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 sample 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: $f_2, f_{S_h+1}, f_{2S_h+1}, \dots$. The same is true for $i'' + 2, i'' + 3, \dots$ 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

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

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

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

Detailed calculation of the derivatives is 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 Spatial bilinear resampling

`vl_nmbilinearsampler` uses bilinear interpolation to spatially warp the image according to an input transformation grid. This operator works with an input image \mathbf{x} , a grid \mathbf{g} , and an output image \mathbf{y} as follows:

$$\mathbf{x} \in \mathbb{R}^{H \times W \times C}, \quad \mathbf{g} \in [-1, 1]^{2 \times H' \times W'}, \quad \mathbf{y} \in \mathbb{R}^{H' \times W' \times C}.$$

The same transformation is applied to all the features channels in the input, as follows:

$$y_{i''j''c} = \sum_{i=1}^H \sum_{j=1}^W x_{ijc} \max\{0, 1 - |\alpha_v g_{1i''j''} + \beta_v - i|\} \max\{0, 1 - |\alpha_u g_{2i''j''} + \beta_u - j|\}, \quad (4.2)$$

where, for each feature channel c , the output $y_{i''j''c}$ at the location (i'', j'') , is a weighted sum of the input values x_{ijc} in the neighborhood of location $(g_{1i''j''}, g_{2i''j''})$. The weights, as given in (4.2), correspond to performing bilinear interpolation. Furthermore, the grid coordinates are expressed not in pixels, but relative to a reference frame that extends from -1 to 1 for all spatial dimensions of the input image; this is given by choosing the coefficients as:

$$\alpha_v = \frac{H-1}{2}, \quad \beta_v = -\frac{H+1}{2}, \quad \alpha_u = \frac{W-1}{2}, \quad \beta_u = -\frac{W+1}{2}.$$

See section 6.5 for implementation details.

4.6 Region of interest pooling

The *region of interest (ROI) pooling* block applies max or average pooling to specified sub-windows of a tensor. A region is a rectangular region $R = (u_-, v_-, u_+, v_+)$. The region itself is partitioned into (H', W') tiles along the vertical and horizontal directions. The edges of the tiles have coordinates

$$\begin{aligned} v_{i'} &= v_- + (v_+ - v_- + 1)(i' - 1), & i' &= 1, \dots, H', \\ u_{j'} &= u_- + (u_+ - u_- + 1)(j' - 1), & j' &= 1, \dots, W'. \end{aligned}$$

Following the implementation of [3], the $H' \times W'$ pooling tiles are given by

$$\Omega_{i'j'} = \{[v_{i'}] + 1, \dots, [v_{i'+1}]\} \times \{[u_{j'}] + 1, \dots, [u_{j'+1}]\}.$$

Then the input and output tensors are as follows:

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

where

$$y_{i'j'c} = \max_{(i,j) \in \Omega_{i'j'}} x_{ijc}.$$

Alternatively, max can be replaced by the averaging operator.

The extent of each region is defined by four coordinates as specified above; however, differently from tensor indexes, these use $(0, 0)$ as the coordinate of the top-left pixel. In fact, if there is a single tile ($H' = W' = 1$), then the region $(0, 0, H - 1, W - 1)$ covers the whole input image:

$$\Omega_{11} = \{1, \dots, W\} \times \{1, \dots, H\}.$$

In more details, the input of the block is a sequence of K regions. Each region pools one of the T images in the batch stored in $\mathbf{x} \in \mathbb{R}^{H \times W \times C \times T}$. Regions are therefore specified as a tensor $R \in \mathbb{R}^{5 \times K}$, where the first coordinate is the index of the pooled image in the batch. The output is a $\mathbf{y} \in \mathbb{R}^{H' \times W' \times C \times K}$ tensor.

For compatibility with [3], furthermore, the region coordinates are rounded to the nearest integer before the definitions above are used. Note also that, due to the discretization details, 1) tiles always contain at least one pixel, 2) there can be a pixel of overlap between them and 3) the discretization has a slight bias towards left-top pixels.

4.7 Normalization

4.7.1 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 , $G(k) \subset \{1, 2, \dots, D\}$ 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 6.6.1 for implementation details.

4.7.2 Batch normalization

`vl_nbnorm` implements batch normalization [5]. 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). $y = \text{vl_nbnorm}(x, w, b)$ normalizes each channel of the feature map x averaging over spatial locations and batch instances. Let T be the batch size; then

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

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 w and b define component-wise multiplicative and additive constants. The output feature map is given by

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

See section 6.6.2 for implementation details.

4.7.3 Spatial normalization

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

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

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 6.6.3 for implementation details.

4.7.4 Softmax

`vl_nnssoftmax` 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 6.6.4 for implementation details.

4.8 Categorical losses

The purpose of a categorical loss function $\ell(\mathbf{x}, \mathbf{c})$ is to compare a prediction \mathbf{x} to a ground truth class label \mathbf{c} . As in the rest of MATCONVNET, the loss is treated as a convolutional operator, in the sense that the loss is evaluated independently at each spatial location. However, the contribution of different samples are summed together (possibly after weighting) and the output of the loss is a scalar. Section 4.8.1 losses useful for multi-class classification and the section 4.8.2 losses useful for binary attribute prediction. Further technical details are in section 6.7. `v1_nnloss` implements the following all of these.

4.8.1 Classification losses

Classification losses decompose additively as follows:

$$\ell(\mathbf{x}, \mathbf{c}) = \sum_{ijn} w_{ijn} \ell(\mathbf{x}_{ij:n}, \mathbf{c}_{ij:n}). \quad (4.3)$$

Here $\mathbf{x} \in \mathbb{R}^{H \times W \times C \times N}$ and $\mathbf{c} \in \{1, \dots, C\}^{H \times W \times 1 \times N}$, such that the slice $\mathbf{x}_{ij:n}$ represent a vector of C class scores and $c_{ij:n}$ is the ground truth class label. The `instanceWeights` option can be used to specify the tensor \mathbf{w} of weights, which are otherwise set to all ones; \mathbf{w} has the same dimension as \mathbf{c} .

Unless otherwise noted, we drop the other indices and denote by \mathbf{x} and c the slice $\mathbf{x}_{ij:n}$ and the scalar $c_{ij:n}$. `v1_nnloss` automatically skips all samples such that $c = 0$, which can be used as an “ignore” label.

Classification error. The classification error is zero if class c is assigned the largest score and zero otherwise:

$$\ell(\mathbf{x}, c) = \mathbf{1} \left[c \neq \underset{k}{\operatorname{argmax}} x_k \right]. \quad (4.4)$$

Ties are broken randomly.

Top- K classification error. The top- K classification error is zero if class c is within the top K ranked scores:

$$\ell(\mathbf{x}, c) = \mathbf{1} [\{k : x_k \geq x_c\} \leq K]. \quad (4.5)$$

The classification error is the same as the top-1 classification error.

Log loss or negative posterior log-probability. In this case, \mathbf{x} is interpreted as a vector of posterior probabilities $p(k) = x_k, k = 1, \dots, C$ over the C classes. The loss is the negative log-probability of the ground truth class:

$$\ell(\mathbf{x}, c) = -\log x_c. \quad (4.6)$$

Note that this makes the implicit assumption $\mathbf{x} \geq 0, \sum_k x_k = 1$. Note also that, unless $x_c > 0$, the loss is undefined. For these reasons, \mathbf{x} is usually the output of a block such as

softmax that can guarantee these conditions. However, the composition of the naive log loss and softmax is numerically unstable. Thus this is implemented as a special case below.

Generally, for such a loss to make sense, the score x_c should be somehow in competition with the other scores $x_k, k \neq c$. If this is not the case, minimizing (4.6) can trivially be achieved by maxing all x_k large, whereas the intended effect is that x_c should be large compared to the $x_k, k \neq c$. The softmax block makes the score compete through the normalization factor.

Softmax log-loss or multinomial logistic loss. This loss combines the softmax block and the log-loss block into a single block:

$$\ell(\mathbf{x}, c) = -\log \frac{e^{x_c}}{\sum_{k=1}^C e^{x_k}} = -x_c + \log \sum_{k=1}^C e^{x_k}. \quad (4.7)$$

Combining the two blocks explicitly is required for numerical stability. Note that, by combining the log-loss with softmax, this loss automatically makes the score compete: $\ell(\mathbf{x}, c) \approx 0$ when $x_c \gg \sum_{k \neq c} x_k$.

This loss is implemented also in the *deprecated* function `v1_softmaxloss`.

Multi-class hinge loss. The multi-class logistic loss is given by

$$\ell(\mathbf{x}, c) = \max\{0, 1 - x_c\}. \quad (4.8)$$

Note that $\ell(\mathbf{x}, c) = 0 \Leftrightarrow x_c \geq 1$. This, just as for the log-loss above, this loss does not automatically make the score competes. In order to do that, the loss is usually preceded by the block:

$$y_c = x_c - \max_{k \neq c} x_k.$$

Hence y_c represent the *confidence margin* between class c and the other classes $k \neq c$. Just like softmax log-loss combines softmax and loss, the next loss combines margin computation and hinge loss.

Structured multi-class hinge loss. The structured multi-class logistic loss, also know as Crammer-Singer loss, combines the multi-class hinge loss with a block computing the score margin:

$$\ell(\mathbf{x}, c) = \max \left\{ 0, 1 - x_c + \max_{k \neq c} x_k \right\}. \quad (4.9)$$

4.8.2 Attribute losses

Attribute losses are similar to classification losses, but in this case classes are not mutually exclusive; they are, instead, binary attributes. Attribute losses decompose additively as follows:

$$\ell(\mathbf{x}, \mathbf{c}) = \sum_{ijkn} w_{ijkn} \ell(\mathbf{x}_{ijkn}, \mathbf{c}_{ijkn}). \quad (4.10)$$

Here $\mathbf{x} \in \mathbb{R}^{H \times W \times C \times N}$ and $\mathbf{c} \in \{-1, +1\}^{H \times W \times C \times N}$, such that the scalar x_{ijkn} represent a confidence that attribute k is on and c_{ijkn} is the ground truth attribute label. The `instanceWeights` option can be used to specify the tensor \mathbf{w} of weights, which are otherwise set to all ones; \mathbf{w} has the same dimension as \mathbf{c} .

Unless otherwise noted, we drop the other indices and denote by x and c the scalars x_{ijkn} and c_{ijkn} . As before, samples with $c = 0$ are skipped.

Binary error. This loss is zero only if the sign of $x - \tau$ agrees with the ground truth label c :

$$\ell(x, c|\tau) = \mathbf{1}[\text{sign}(x - \tau) \neq c]. \quad (4.11)$$

Here τ is a configurable threshold, often set to zero.

Binary log-loss. This is the same as the multi-class log-loss but for binary attributes. Namely, this time $x_k \in [0, 1]$ is interpreted as the probability that attribute k is on:

$$\ell(x, c) = \begin{cases} -\log x, & c = +1, \\ -\log(1 - x), & c = -1, \end{cases} \quad (4.12)$$

$$= -\log \left[c \left(x - \frac{1}{2} \right) + \frac{1}{2} \right]. \quad (4.13)$$

Similarly to the multi-class log loss, the assumption $x \in [0, 1]$ must be enforced by the block computing x .

Binary logistic loss. This is the same as the multi-class logistic loss, but this time $x/2$ represents the confidence that the attribute is on and $-x/2$ that it is off. This is obtained by using the logistic function $\sigma(x)$

$$\ell(x, c) = -\log \sigma(cx) = -\log \frac{1}{1 + e^{-cx}} = -\log \frac{e^{\frac{cx}{2}}}{e^{\frac{cx}{2}} + e^{-\frac{cx}{2}}}. \quad (4.14)$$

Binary hinge loss. This is the same as the structured multi-class hinge loss but for binary attributes:

$$\ell(x, c) = \max\{0, 1 - cx\}. \quad (4.15)$$

There is a relationship between the hinge loss and the structured multi-class hinge loss which is analogous to the relationship between binary logistic loss and multi-class logistic loss. Namely, the hinge loss can be rewritten as:

$$\ell(x, c) = \max \left\{ 0, 1 - \frac{cx}{2} + \max_{k \neq c} \frac{kx}{2} \right\}$$

Hence the hinge loss is the same as the structure multi-class hinge loss for $C = 2$ classes, where $x/2$ is the score associated to class $c = 1$ and $-x/2$ the score associated to class $c = -1$.

4.9 Comparisons

4.9.1 p -distance

The `v1_mnpdist` function computes the 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 $x_{ij\cdot}$. By specifying the option `'noRoot', true` it is possible to compute a variant omitting the root:

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

See section [6.8.1](#) 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 blocks such as convolutions for which one can deterministically establish dependency chains between network components. We also assume that all the inputs \mathbf{x} and outputs \mathbf{y} 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:

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

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

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' be the vertical filter dimension, S_h the subsampling 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:

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

Hence

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

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 $i'' = 1, 2, \dots, H''$ 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

$$H'' = \left\lceil \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rceil + 1. \quad (5.3)$$

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

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

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 for above 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

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

Using the fact that for integers 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:

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

Rearranging

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

Using $\lceil a/b \rceil = \lfloor (a - b + 1)/b \rfloor$ 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 fig. 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

$$1 + U_h(i - 1) - C_h^- \leq i'' \leq H' + U_h(i - 1) - C_h^- \quad (5.5)$$

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

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

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

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

Next, we want to determine the height H'' of the output \mathbf{y} of convolution transpose as a function of the height H of the input \mathbf{x} and the other parameters. Swapping input and output in (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

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

This is 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 fig. 4.1 and fig. 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 (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

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

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

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

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 obtained in section 5.3.

5.5 Composing receptive fields

Consider now the composition of 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

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

5.6 Overlaying receptive fields

Consider now the combination $h(f(\mathbf{x}_1), g(\mathbf{x}_2))$ where the domains 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 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 *compatible*. The range of input samples $i = i_f = i_g$ that affect any output sample i_h is then given by

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

We conclude that the combined receptive field is

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

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 $\phi(\mathbf{x})$ be 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:

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

where the correspondence between indexes (i, j, d) and (p, q) is given by the map $(i, j, d) = t(p, q)$ where:

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

In practice, this map is slightly modified to account for the padding, stride, and dilation factors. 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

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

Hence we obtain the following expression for the vectorized output (see [7]):

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

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:

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

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

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

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

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

where $X \in \mathbb{R}^{(HW) \times D}$ 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} to be 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 $\text{vec } \mathbf{y} = M \text{vec } \mathbf{x}$ for a suitable matrix M ; convolution transpose computes instead $\text{vec } \mathbf{x} = M^\top \text{vec } \mathbf{y}$. 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):

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

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 $\text{vec } \mathbf{y}^\top (M \text{vec } \mathbf{x}) = (\text{vec } \mathbf{y}^\top M) \text{vec } \mathbf{x} = \text{vec } \mathbf{x}^\top (M^\top \text{vec } \mathbf{y})$. 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, $q \leq \lfloor (H' - 1)/S \rfloor + 1$ is a fairly tight upper bound on the maximum value (although, depending on k , there could be an element less). Hence

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

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 selects for each output element an input element, the relation can be expressed in matrix form as $\text{vec } \mathbf{y} = S(\mathbf{x}) \text{vec } \mathbf{x}$ for a suitable selector matrix $S(\mathbf{x}) \in \{0, 1\}^{(H''W''D) \times (HWD)}$. The derivatives can be written as: $\frac{dz}{d \text{vec } \mathbf{x}}^\top = \frac{dz}{d \text{vec } \mathbf{y}}^\top S(\mathbf{x})$, 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 average 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:

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

6.4 Activation functions

6.4.1 ReLU

The ReLU operator can be expressed in matrix notation as

$$\text{vec } \mathbf{y} = \text{diag } \mathbf{s} \text{vec } \mathbf{x}, \quad \frac{dz}{d \text{vec } \mathbf{x}} = \text{diag } \mathbf{s} \frac{dz}{d \text{vec } \mathbf{y}}$$

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

6.4.2 Sigmoid

The derivative of the sigmoid function is given by

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

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 Spatial bilinear resampling

The projected derivative $d\langle \mathbf{p}, \phi(\mathbf{x}, \mathbf{g}) \rangle / d\mathbf{x}$ of the spatial bilinear resampler operator with respect to the input image \mathbf{x} can be found as follows:

$$\begin{aligned} \frac{\partial}{\partial x_{ijc}} &\left[\sum_{i''j''c''} p_{i''k''c''} \sum_{i'=1}^H \sum_{j'=1}^W x_{i'j'c''} \max\{0, 1 - |\alpha_v g_{1i''j''} + \beta_v - i'|\} \max\{0, 1 - |\alpha_u g_{2i''j''} + \beta_u - j'|\} \right] \\ &= \sum_{i''j''} p_{i''k''c} \max\{0, 1 - |\alpha_v g_{1i''j''} + \beta_v - i|\} \max\{0, 1 - |\alpha_u g_{2i''j''} + \beta_u - j|\}. \end{aligned} \quad (6.3)$$

Note that the formula is similar to Eq. 4.2, with the difference that summation is on i'' rather than i .

The projected derivative $d\langle \mathbf{p}, \phi(\mathbf{x}, \mathbf{g}) \rangle / d\mathbf{g}$ with respect to the grid is similar:

$$\begin{aligned} \frac{\partial}{\partial g_{1i'j'}} &\left[\sum_{i''j''c} p_{i''k''c} \sum_{i=1}^H \sum_{j=1}^W x_{ijc} \max\{0, 1 - |\alpha_v g_{1i''j''} + \beta_v - i|\} \max\{0, 1 - |\alpha_u g_{2i''j''} + \beta_u - j|\} \right] \\ &= - \sum_c p_{i'j'c} \sum_{i=1}^H \sum_{j=1}^W \alpha_v x_{ijc} \max\{0, 1 - |\alpha_v g_{2i'j'} + \beta_v - j|\} \text{sign}(\alpha_v g_{1i'j'} + \beta_v - j) \mathbf{1}_{\{-1 < \alpha_u g_{2i'j'} + \beta_u < 1\}}. \end{aligned} \quad (6.4)$$

A similar expression holds for $\partial g_{2i'j'}$

6.6 Normalization

6.6.1 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

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

6.6.2 Batch normalization

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

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

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 zero. Hence

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

where

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

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

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

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

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

i.e.

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

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

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

6.6.3 Spatial normalization

The neighbourhood norm $n_{i''j''d}^2$ can be computed by applying average pooling to x_{ijd}^2 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:

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

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

6.6.4 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 the numerator and denominator by the exponential of 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 form:

$$\frac{dz}{dX} = Y \odot \left(\frac{dz}{dY} - \left(\frac{dz}{dY} \odot Y \right) \mathbf{11}^\top \right)$$

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.

6.7 Categorical losses

This section obtains the projected derivatives of the categorical losses in section 4.8. Recall that all losses give a scalar output, so the projection tensor p is trivial (a scalar).

6.7.1 Classification losses

Top- K classification error. The derivative is zero a.e.

Log-loss. The projected derivative is:

$$\frac{\partial p\ell(\mathbf{x}, c)}{\partial x_k} = -p \frac{\partial \log(x_c)}{\partial x_k} = -p x_c \delta_{k=c}.$$

Softmax log-loss. The projected derivative is given by:

$$\frac{\partial p\ell(\mathbf{x}, c)}{\partial x_k} = -p \frac{\partial}{\partial x_k} \left(x_c - \log \sum_{t=1}^C e^{x_t} \right) = -p \left(\delta_{k=c} - \frac{e^{x_c}}{\sum_{t=1}^C e^{x_t}} \right).$$

In brackets, we can recognize the output of the loss itself:

$$y = \ell(\mathbf{x}, c) = \frac{e^{x_c}}{\sum_{t=1}^C e^{x_t}}.$$

Hence the loss derivatives rewrites:

$$\frac{\partial p\ell(\mathbf{x}, c)}{\partial x_k} = -p (\delta_{k=c} - y).$$

Multi-class hinge loss. The projected derivative is:

$$\frac{\partial p\ell(\mathbf{x}, c)}{\partial x_k} = -p \mathbf{1}[x_c < 1] \delta_{k=c}.$$

Structured multi-class hinge loss. The projected derivative is:

$$\frac{\partial p\ell(\mathbf{x}, c)}{\partial x_k} = -p \mathbf{1}[x_c < 1 + \max_{t \neq c} x_t] (\delta_{k=c} - \delta_{k=t^*}), \quad t^* = \operatorname{argmax}_{t=1,2,\dots,C} x_t.$$

6.7.2 Attribute losses

Binary error. The derivative of the binary error is 0 a.e.

Binary log-loss. The projected derivative is:

$$\frac{\partial p\ell(x, c)}{\partial x} = -p \frac{c}{c \left(x - \frac{1}{2}\right) + \frac{1}{2}}.$$

Binary logistic loss. The projected derivative is:

$$\frac{\partial p\ell(x, c)}{\partial x} = -p \frac{\partial}{\partial x} \log \frac{1}{1 + e^{-cx}} = -p \frac{ce^{-cx}}{1 + e^{-cx}} = -p \frac{c}{e^{cx} + 1} = -pc \sigma(-cx).$$

Binary hinge loss. The projected derivative is

$$\frac{\partial p\ell(x, c)}{\partial x} = -pc \mathbf{1}[cx < 1].$$

6.8 Comparisons

6.8.1 p -distance

The derivative of the operator without root is given by:

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

The derivative of the operator with root is given by:

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

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

6.9 Other implementation details

6.9.1 Normal sampler

The function `vl::randn()` uses the Ziggurah method [10] to sample from a Normally-distributed random variable. Let $f(x) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x^2)$ the standard Normal distribution. The sampler encloses $f(x)$ in a simple shape made of $K - 1$ horizontal rectangles and a base composed of a rectangle tapering off in an exponential distribution. These are defined by points $x_1 > x_2 > x_3 > \dots > x_K = 0$ such that (for the right half of $f(x)$) the layers of the Ziggurat are given by

$$\forall k = 1, \dots, K - 1: \quad R_k = [f(x_k), f(x_{k+1})] \times [0, x_k].$$

and such that its basis is given by

$$R_0 = ([0, f(x_1)] \times [0, x_1]) \cup \{(x, y) : x \geq x_1, y \leq f(x_1) \exp(-x_1(x - x_1))\}$$

Note that, since the last point $x_K = 0$, (half of) the distribution is enclosed by the Ziggurat, i.e. $\forall x \geq 0 : (x, f(x)) \in \cup_{k=0}^K R_k$.

The first point x_1 in the sequence determines the area of the Ziggurat base:

$$A = |R_0| = f(x_1)x_1 + f(x_1)/x_1.$$

The other points are defined recursively such that the area is the same for all rectangles:

$$A = |R_k| = (f(x_{k+1}) - f(x_k))x_k \quad \Rightarrow \quad x_{k+1} = f^{-1}(A/x_k + f(x_k)).$$

There are two degrees of freedom: the number of subdivisions K and the point x_1 . Given K , the goal is to choose x_1 such that the K -th points $x_K = 0$ lands on zero, enclosing tightly $f(x)$. The required value of x_1 is easily found using bisection and, for $K = 256$, is $x_1 = 3.655420419026953$. Given x_1 , A and all other points in the sequence can be derived easily using the formulas above.

The Ziggurath can be used to quickly sample from the Normal distribution. In order to do so, one first samples a point (x, y) uniformly at random from the Ziggurat $\cup_{k=0}^K R_k$ and then rejects pairs (x, y) that do not belong to the graph of $f(x)$, i.e. $y > f(x)$. Specifically:

1. Sample a point (x, y) uniformly from the Ziggurat. To do so, sample uniformly at random an index $k \in \{0, 1, \dots, K - 1\}$ and two scalars u, v in the interval $[0, 1)$. Then, for $k \geq 1$, set $x = ux_k$ and $y = vf(x_{k+1}) + (1 - v)f(x_k)$ (for $k = 0$ see below). Since all regions R_k have the same area and (x, y) are then drawn uniformly from the selected rectangle, this samples a point (x, y) from the Ziggurat uniformly at random.
2. If $y \leq f(x)$, accept x as a sample; otherwise, sample again. Note that, when $x \leq x_{k+1}$, the test $y \leq f(x_{k+1}) < f(x)$ is always successful, and the variable y and test can be skipped in the step above.

Next, we complete the procedure for $k = 0$, when R_0 is not just a rectangle but rather the union of a rectangle and an exponential distribution. To sample from R_0 uniformly, we either choose the rectangle or the exponential distribution with a probability proportional to their area. Reusing the notation (and corresponding code) above, we can express this as sampling $x = ux_0$ and accepting the latter as a sample from the rectangle component if $ux_0 \leq x_1$; here the pseudo-point x_0 is defined such that $x_1/x_0 = f(x_1)x_1/A$, i.e. $x_0 = A/f(x_1)$. If the test fails, we sample instead from the exponential distribution $x \sim x_1 \exp(-x_1(x - x_1))$, $x \geq x_1$. To do so, let $z = x_1 \exp(-x_1(x - x_1))$; then $x = x_1 - (1/x_1) \ln z/x_1$ and $dx = |(x_1/z)|dz$, where $z \in (0, x_1]$. Since $x_1 \exp(-x_1(x - x_1))dx = (1/x_1)dz$ is uniform, we can implement this by sampling u uniformly in $(0, 1]$ and setting $x = x_1 - (1/x_1) \ln u$. Finally, recall that the goal is to sample from the Normal distribution, not the exponential, so the latter sample must be refined by rejection sampling. As before, this requires sampling a pair (x, y) under the exponential distribution graph. Given x sampled from the exponential distribution, we sample the corresponding y uniformly at random in the interval $[0, f(x_1) \exp(-x_1(x - x_1))]$,

and write the latter as $y = vf(x_1) \exp(-x_1(x - x_1))$, where v is uniform in $[0, 1]$. The latter is then accepted provided that y is below the Normal distribution graph $f(x)$, i.e. $vf(x_1) \exp(-x_1(x - x_1)) \leq f(x)$. A short calculation yields the test:

$$-2 \ln v \geq x_1^2 + x^2 - 2x_1x = (x_1 - x)^2 = ((1/x_1) \ln u)^2.$$

6.9.2 Euclid's algorithm

Euclid's algorithm finds the *greatest common divisor* (GCD) of two non-negative integers a and b . Recall that the GCD is the largest integer that divides both a and b :

$$\gcd(a, b) = \max\{d \in \mathbb{N} : d|a \wedge d|b\}.$$

Lemma 1 (Euclid's algorithm). *Let $a, b \in \mathbb{N}$ and let $q \in \mathbb{Z}$ such that $a - qb \geq 0$. Then*

$$\gcd(a, b) = \gcd(a - qb, b).$$

Proof. Let d be a divisor of both a and b . Then d divides $a - qb$ as well because:

$$\frac{a - qb}{d} = \underbrace{\frac{a}{d}}_{\in \mathbb{Z}} - q \underbrace{\frac{b}{d}}_{\in \mathbb{Z}} \Rightarrow \frac{a - qb}{d} \in \mathbb{Z}.$$

Hence $\gcd(a, b) \leq \gcd(a - qb, b)$. In the same way, we can show that, if d divides $a - qb$ as well as b , then it must divide a too, hence $\gcd(a - qb, b) \leq \gcd(a, b)$. \square

Euclid's algorithm starts with $a > b \geq 1$ and sets q to the quotient of the integer division a/b . Due to the lemma above, the GCD of a and b is the same as the GCD of the remainder $r = a - qb = (a \bmod b)$ and b :

$$\gcd(a, b) = \gcd(a, a \bmod b).$$

Since the remainder $(a \bmod b) < b$ is strictly smaller than b , now GCD is called with smaller arguments. The recursion terminates when a zero remainder is generated, because

$$\gcd(a, 0) = a.$$

We can modify the algorithms to also find two integers u, v , the Bézout's coefficients, such that:

$$au + bv = \gcd(a, b).$$

To do so, we replace $a = b(a/b) + r$ as above:

$$ru + bv' = \gcd(a, b) = \gcd(r, b), \quad v' = \frac{a}{b}u + v.$$

The recursion terminates when $r = 0$, in which case

$$bv' = \gcd(0, b) = b \Rightarrow v' = b.$$

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] R. Girshick. Fast R-CNN. In *arXiv*, number arXiv:1504.08083, 2015.
- [4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, 2015.
- [5] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv e-prints*, 2015.
- [6] Yangqing Jia. Caffe: An open source convolutional architecture for fast feature embedding. <http://caffe.berkeleyvision.org/>, 2013.
- [7] D. B. Kinghorn. Integrals and derivatives for correlated gaussian fuctions using matrix differential calculus. *International Journal of Quantum Chemistry*, 57:141–155, 1996.
- [8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS*, 2012.
- [9] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.
- [10] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of statistical software*, 5(8):1–7, 2000.
- [11] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. ICLR*, 2014.
- [12] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. 2015.
- [13] A. Vedaldi and B. Fulkerson. VLFeat – An open and portable library of computer vision algorithms. In *Proc. ACM Int. Conf. on Multimedia*, 2010.
- [14] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Proc. ECCV*, 2014.