AIMS Big Data Course
Introduction to deep learning

Dr Andrea Vedaldi
Dr Andrew Zisserman

For lecture notes and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

AIMS Big Data

Overview

Dr Andrew Zisserman (2 lectures)
- Lecture 1: Discriminative Learning 1 [AZ]
- Lecture 2: Discriminative Learning 2 and searching Big Data
  - Practical 1 (Image Classification)

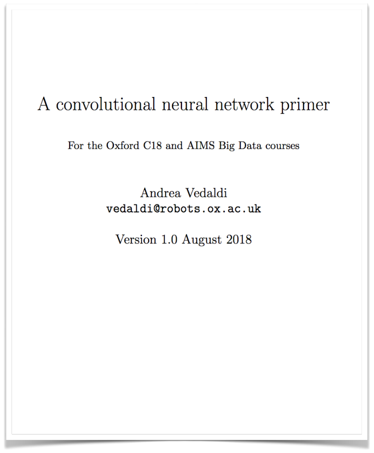
Dr Andrea Vedaldi (2 lectures)
- Lecture 3: Introduction to deep learning
- Lecture 4: Universal, unsupervised and understandable representations
  + Practical 2 (CNNs)

AIMS Deep Learning Materials

Notes and handout

Look for materials at

http://www.robots.ox.ac.uk/~vedaldi/teach.html


A convolutional neural network primer

For the Oxford C18 and AIMS Big Data courses

Andrea Vedaldi
vedaldi@robots.ox.ac.uk

Version 1.0 August 2018

AIMS Big Data Course
Introduction to deep learning

Part 1: Convolutional neural networks

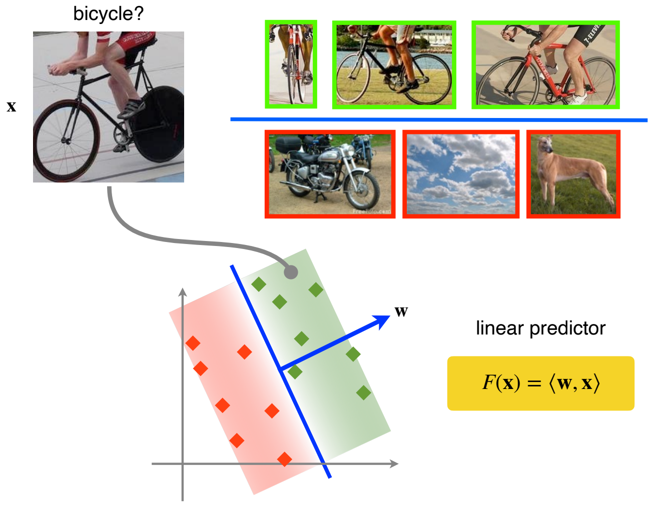
Learning a classifier

We would like to build a **predictor** that can tell if an image $\mathbf{x}$ contains a certain object (say a "bicycle").

We **learn** this function from example images that do and do not contain the object.

In the simplest case, the function is a **linear predictor** $F(\mathbf{x})$:

- Images are interpreted as (high-dimensional) vectors.
- $F(\mathbf{x})$ dots $\mathbf{x}$ and a **parameter vector** $\mathbf{w}$ to obtain the **score** for the positive hypothesis (bicycle).
- The **sign** of $F(\mathbf{x})$ is used as prediction.



linear predictor

$F(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle$

Data representations

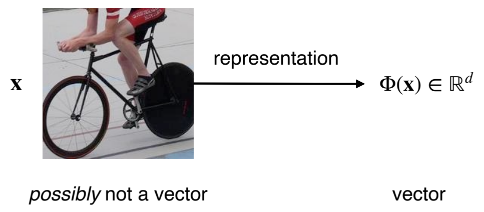
Linear predictors beyond vector inputs

Beyond vector data

A linear predictor applies to **vector data**.

However, we want to process images, text, videos, or sounds that are not necessarily vectors.

For this, we use a **representation function** Φ, which maps the data to vectors.



Non-linear classification

Representations are used even if the data $\mathbf{x}$ is already a vector.

They result in a non-linear classifier function which can be significantly **more expressive** than a linear one.

linear predictor

$F(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle$

non-linear predictor

$F(\mathbf{x}) = \langle \mathbf{w}, \Phi(\mathbf{x}) \rangle$

Meaningful representations

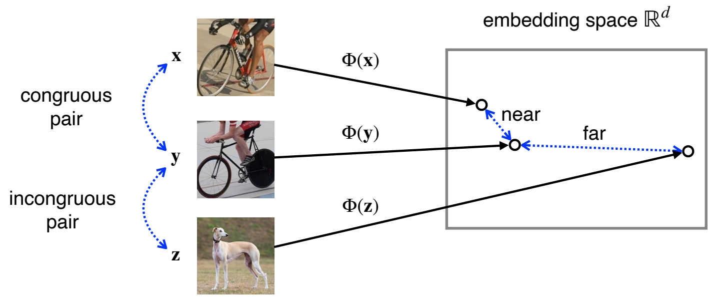
A representation should help the linear classifier to perform discrimination.

The goal is to map the **semantic similarity** between data points to a corresponding **vector similarity**.

A good representation is:

- **invariant** to nuisance factors
- **sensitive** to semantic factors

Semantic similarity $\xrightarrow{\text{representation}}$ Vector similarity (distance)



The perceptron

Convolutional networks

Learning via SGD

Evaluation

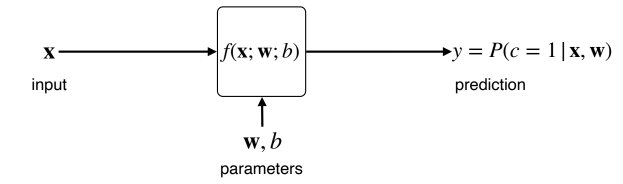
The perceptron

An early neural network by Rosenblatt (1957)

What

The **perceptron** maps an **input vector** $\mathbf{x}$ to a **probability value** y.

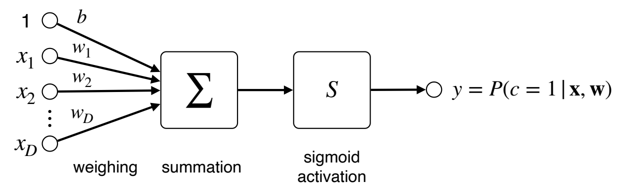
For example, y could be the probability that $\mathbf{x}$ is an image of a "bicycle" rather than not.



$\mathbf{x}$
input
$f(\mathbf{x};\mathbf{w};b)$
$y = P(c=1\,|\,\mathbf{x},\mathbf{w})$
prediction
$\mathbf{w}, b$
parameters

How

The perceptron computes this probability by **weighing** the vector components, **summing** them, and then applying a non-linear **sigmoid activation** function.



The sigmoid activation function

Makes the perceptron non-linear

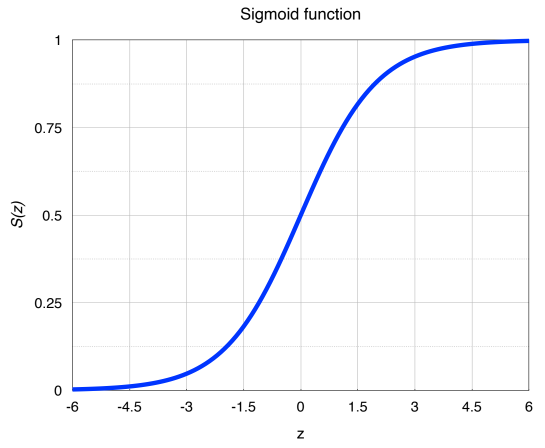
The activation function in the perceptron is a **sigmoid**

$$S(z) = \frac{1}{1+e^{-z}}$$

The sigmoid converts **real scores** z in the range $(-\infty, \infty)$ into **probability values** in the range $(0, 1)$.

It has several remarkable properties, such as the following identity for its derivative

$$\frac{dS}{dz} = S(z)(1-S(z)) = S(z)S(-z)$$



The perceptron as a parametric function

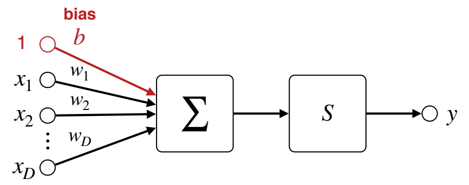
Perceptron = linear classifier + sigmoid

The perceptron is a function $f(\mathbf{x};\mathbf{w};b)$ parametrised by a weight vector $\mathbf{w}$ and a bias b.

The function:

1. Maps a vector $\mathbf{x}$ to a scalar score using the linear function $\langle \mathbf{x}, \mathbf{w}\rangle + b$.
2. Transforms the score into a **probability value** by applying the sigmoid function $S(z)$.

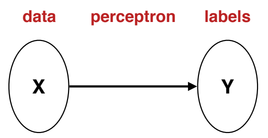
There usually is a constant **bias term** b added to the score. This can be implemented by extending the input vector with a constant element equal to 1 and including b in $\mathbf{w}$.



$$f(\mathbf{x};\mathbf{w},b) = S\left(\langle \mathbf{w},\mathbf{x}\rangle + b\right) = \frac{1}{1+\exp(-w_1x_1 - \ldots - w_Dx_D - b)}$$

Training the perceptron: least square

Regard the perceptron as a parametric function from an input space **X** to an output space **Y**:



$$\mathbf{x} \longmapsto y = S\left(\langle \mathbf{w}, \mathbf{x} \rangle + b\right)$$

The parameters $(\mathbf{w}, b)$ of the perceptron are **learned empirically** by fitting the function to **example data** $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_N, y_N),$.

This can be done by solving a least-square problem:

$$E(\mathbf{w}, b) = \frac{1}{N} \sum_{i=1}^{N} \left(S(\langle \mathbf{w}, \mathbf{x} \rangle + b) - y_i\right)^2$$

This problem is **non-linear** due to the activation function S. It needs to be solved by an iterative method such as gradient descent.

Cross-entropy loss

Better than least square for classification problems

Given the probabilistic nature of the perceptron output, usually the fitting criterion is not least square, but **maximum log-likelihood**.

The log-likelihood is computed as follows:

- The posterior probability of the 0/1 label y_i can be expressed as

$$P(y_i \,|\, \mathbf{x}_i; \mathbf{w}) = f(\mathbf{x}_i; \mathbf{w})^{y_i} (1 - f(\mathbf{x}_i; \mathbf{w}))^{1-y_i}$$

- The negative log-likelihood of the parameters is

$$\begin{aligned} &-\log\, P(y_i \,|\, \mathbf{x}_i; \mathbf{w}) \\ &\quad = -\, y_i \log f(\mathbf{x}_i; \mathbf{w}) - (1 - y_i) \log(1 - f(\mathbf{x}_i; \mathbf{w})) \end{aligned}$$

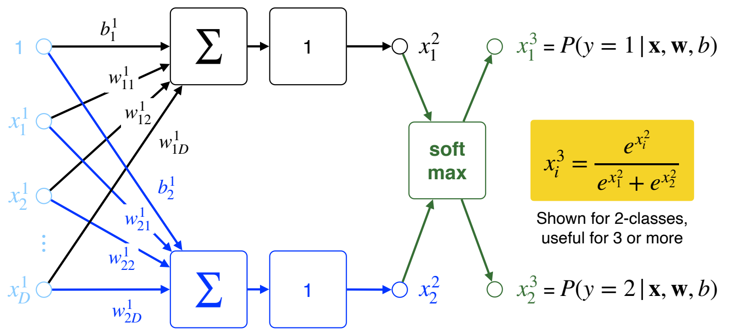
The empirical negative log-likelihood is obtained by averaging the negative log-likelihood over all the training data points

$$E(\mathbf{w}) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log f(\mathbf{x}_i; \mathbf{w}) + (1 - y_i) \log(1 - f(\mathbf{x}_i; \mathbf{w}))$$

Just like the squared objective of least square, this objective function can be minimised by using an iterative method such as gradient descent.

Multi-class perceptron

Softmax layer



$$x_i^3 = \frac{e^{x_i^2}}{e^{x_1^2} + e^{x_2^2}}$$

Shown for 2-classes, useful for 3 or more

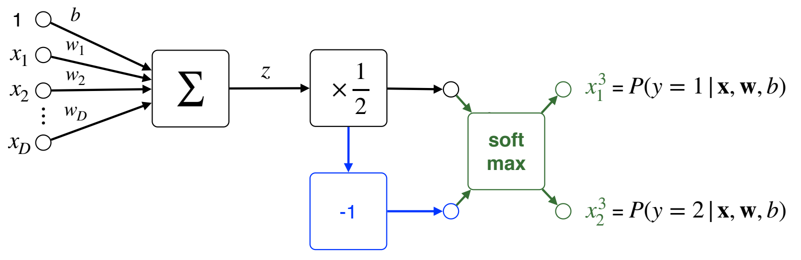
Multiple perceptrons can be combined to predict more than two classes.

Each perceptron computes the score x_c^2 for a class hypothesis $c = 1, \ldots, C$.

The vector of scores $\mathbf{x}^2$ is mapped to a vector of probabilities $\mathbf{x}^3$ using the **softmax** operator, which is a generalisation of the sigmoid.

Softmax = sigmoid for 2 classes

In the **binary case**, the softmax is the same as the sigmoid


1
x_1
x_2
⋮
x_D
b
w_1
w_2
w_D
Σ
z
$\times \frac{1}{2}$
-1
soft max
$x_1^3 = P(y = 1 \,|\, \mathbf{x}, \mathbf{w}, b)$
$x_2^3 = P(y = 2 \,|\, \mathbf{x}, \mathbf{w}, b)$

$$x_1^3 = \frac{e^{x_1^3}}{e^{x_1^3} + e^{x_2^3}} = \frac{e^{\frac{z}{2}}}{e^{\frac{z}{2}} + e^{-\frac{z}{2}}} = \frac{1}{1 + e^{-z}} = S(\langle \mathbf{w}, \mathbf{x} \rangle + b)$$

Multi-class perceptron

Learning from example data

The log-likelihood and objective function for a multi class perceptron are given by:

$$-\log P(y=c\,|\,\mathbf{x}_i, W) = -\log \frac{e^{\mathbf{w}_c^\top \mathbf{x}+b_c}}{\sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x}+b_q}} = -\mathbf{w}_c^\top \mathbf{x} - b_c + \log \sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x}+b_q}$$

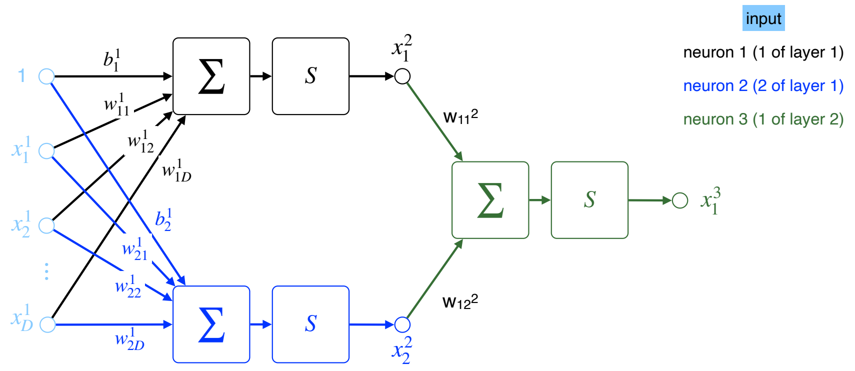
$$E(W) = \frac{1}{N}\sum_{i=1}^{N}\left(-\mathbf{w}_{y_i}^\top \mathbf{x}_i - b_{y_i} + \log \sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x}_i+b_q}\right)$$

This loss function is sometimes called **cross-entropy**. It measures the discrepancy between

- the empirical posterior distributions $Q(c\,|\,\mathbf{x}_i) = \delta(c - y_i)$and
- the predicted posterior distributions $P(c\,|\,\mathbf{x}_i) = P(y = c\,|\,\mathbf{x}_i, W)$.

Multi-layer perceptron (MLP)

Deep architectures



Perceptrons can also be chained, resoling in a so-called **deep neural network**. Depth refers to the fact that the function decomposes as a long ("deep") chain of simpler perception-like functions.

The perceptron

Convolutional networks

Learning via SGD

Evaluation

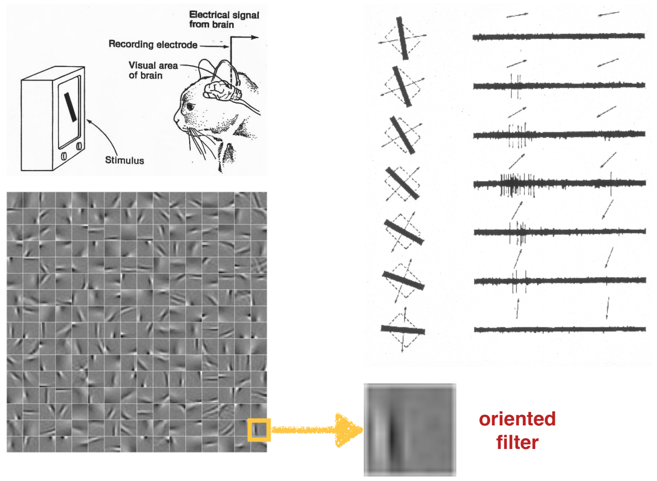
The discovery of oriented cells in the visual cortex

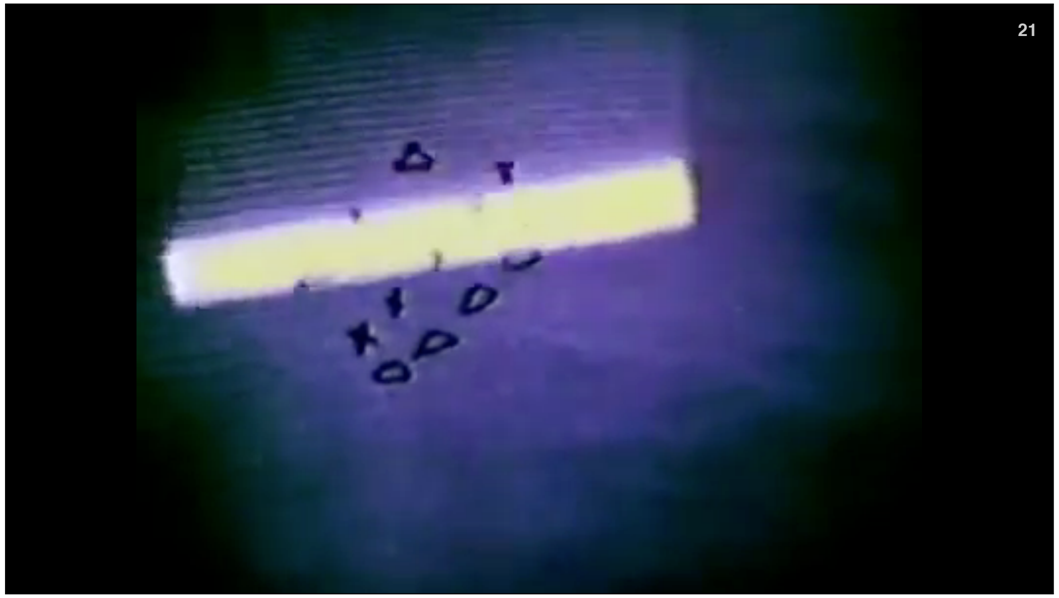
Hubel and Wiesel 1959

In 1959, Hubel & Wiesel conducted seminal experiments on the visual cortex of mammals (Nobel Prize in Physiology and Medicine in 1981).

They discovered the existence of neurons that respond to specific orientations and locations in the retina.

These neurons form a local and (statistically) translation invariant image operator.





Tensors

Variables in CNNs are usually **tensors**, i.e. **multi-dimensional array**.

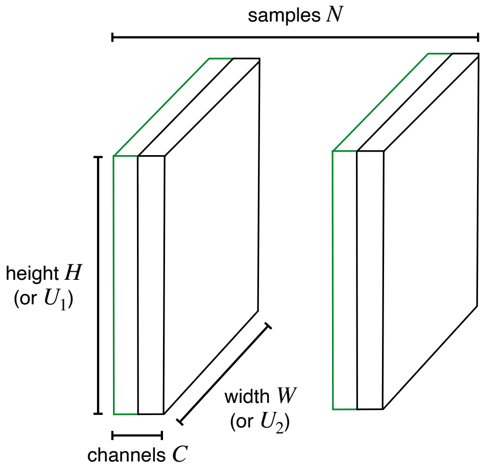
Conventionally, the dimensions are $N \times C \times U_1 \times \ldots \times U_D$ where

- N is the **batch size**, i.e. the number of data samples represented by the tensor.
- C is the number of **channels**.
- $U_1 \times \ldots \times U_D$ are the **spatial dimensions**.

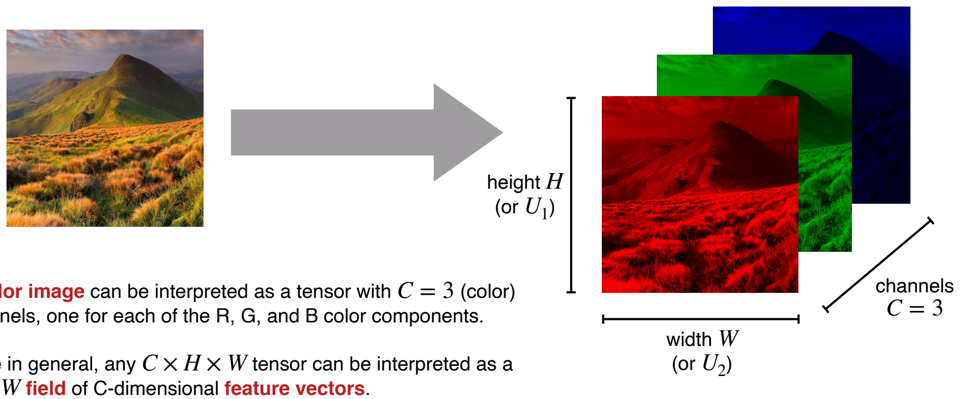
The number of spatial dimensions D can vary. E.g.:

- $D = 2$ is used to represent 2D data such as images.
- $D = 3$ is used to represent 3D data such as volumes.

In general, it is possible to assign any meaning to the dimensions (e.g. time), as required by the application.



Example: images as tensors



A **color image** can be interpreted as a tensor with $C = 3$ (color) channels, one for each of the R, G, and B color components.

More in general, any $C \times H \times W$ tensor can be interpreted as a $H \times W$ **field** of C-dimensional **feature vectors**.

The meaning of the feature channels is often not obvious.

Tensor indexing

Tensor elements x_{ncu} are identified via indexes, one for each dimension:

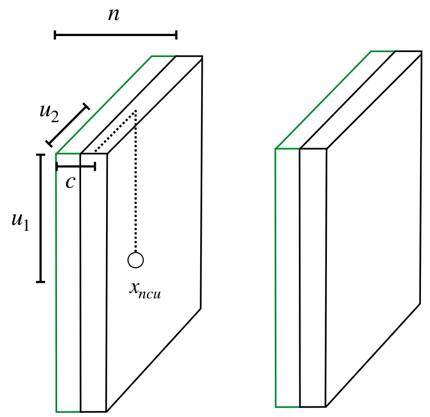
- n is the sample index in the batch
- c is the feature channel index
- u is the spatial index

The spatial index u is in fact a **multi-index**, a shorthand notation for u = (u_1, …, u_D)

Indexes are **0-based**:

- $0 \leq n < N$
- $0 \leq c < C$
- $0 \leq u < U = (U_1, \ldots, U_D)$

Generally, whenever you see a spatial multi-index, just pretend there is only one spatial dimension ($D = 1$). The extension to $D > 1$ is almost always trivial.



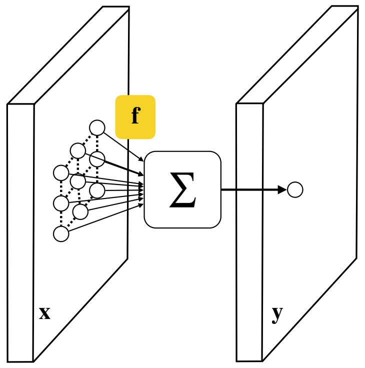
Linear convolution

A simple filtering operation

A linear filter $\mathbf{f}$ computes the weighted summation of a window of the input tensor $\mathbf{x}$.

Key properties:

- **Linearity**: the operation is linear in the input and the filter parameters.
- **Locality**: the operator looks at a small window of data.
- **Translation invariance**: all windows are processed using the same filter weights.



Linear convolution

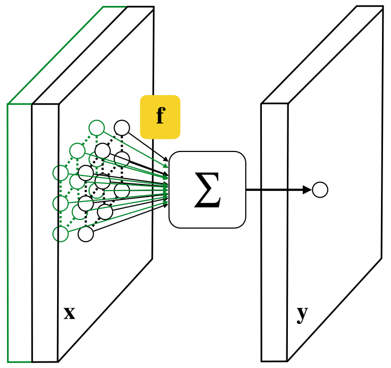
Multiple input channels

A linear filter $\mathbf{f}$ computes the weighted summation of a window of the input tensor $\mathbf{x}$.

Key properties:

- **Linearity**: the operation is linear in the input and the filter parameters.
- **Locality**: the operator looks at a small window of data.
- **Translation invariance**: all windows are processed using the same filter weights.

The filter has one channel for each input tensor channel.



Linear convolution

Multiple output channels and filter banks

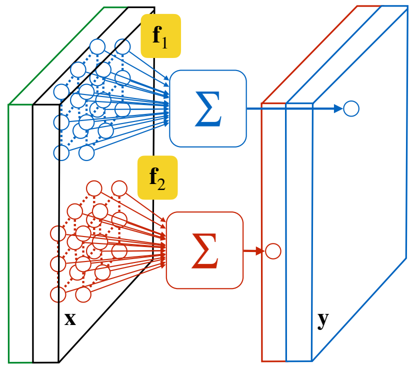
A linear filter $\mathbf{f}$ computes the weighted summation of a window of the input tensor $\mathbf{x}$.

Key properties:

- **Linearity**: the operation is linear in the input and the filter parameters.
- **Locality**: the operator looks at a small window of data.
- **Translation invariance**: all windows are processed using the same filter weights.

The filter has one channel for each input tensor channel.

A **bank of filters** is used to generated multiple output channels, one per filter.



Linear convolution

As a neural network operator

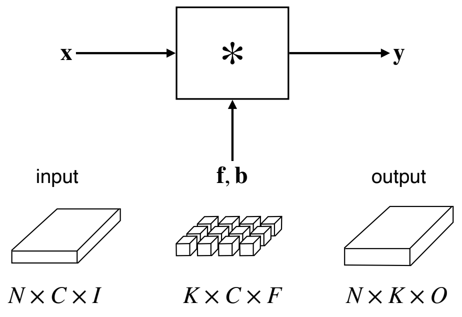
A convolutional layer is an operator that takes an **input** a tensor $\mathbf{x}$ a **filter bank** $\mathbf{f}$ and a **bias** vector $\mathbf{b}$ and produces as **output** a new tensor $\mathbf{y}$.

Dimensions:

- The **batch size** N is the same for input and output.
- Input and filters have the same **number of channels** C.
- The number of output channels K is the same as the **number of filters** in the bank.
- The output dimension O is given by

$$O = I - F + 1$$

Recall that $O = (O_1, O_2)$, $F = (F_1, F_2)$, and $I = (I_1, I_2)$ as we are using the multi-index shorthand.



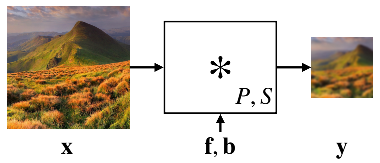
$$y_{nkv} = b_k + \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{kcu} \cdot x_{n,c,v+u}$$

Linear convolution

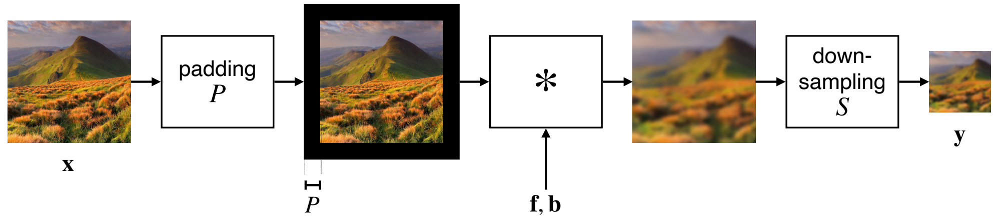
Padding and downsampling

Padding extends a tensor $\mathbf{x}$ with a border P filled with zeros.

Downsampling retain one every S pixels in a tensor, where S is called the **stride**.



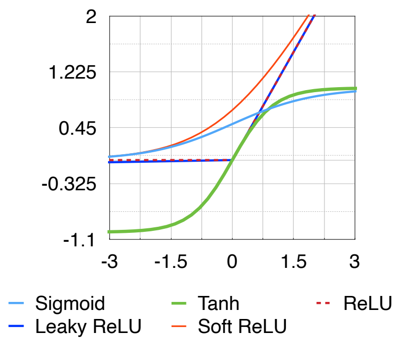
Padding and downsampling can be interpreted as additional layers before and after standard convolution:

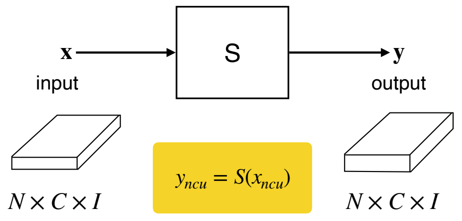


Activation functions

The non-linearity in deep networks

Activation functions are scalar non-linear functions S(z) that are applied element-wise to an input tensor $\mathbf{x}$ to generate an output tensor $\mathbf{y}$ (with the same dimensions).





$y_{ncu} = S(x_{ncu})$

$z = \max\{0, z\}$,	rectified linear unit (ReLU),
$z = \log(1 + e^z)$,	soft ReLU,
$z = \epsilon z + (1-\epsilon)\max\{0, z\}$,	leaky ReLU,
$z = (1 + e^{-z})^{-1}$,	sigmoid,
$z = \tanh(z)$,	hyperbolic tangent,

Pooling

Parameter-less non-linear filters

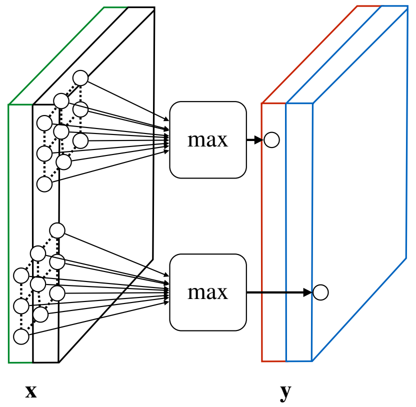
The **max pooling** operator is similar to linear filter, operating transitively on $F = (F_1, F_2)$ sized windows.

The operator extracts the maximum response for each channel and window

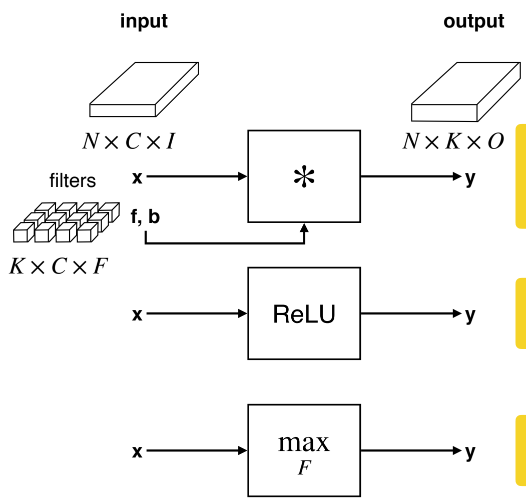
$$y_{ncv} = \max_{0 \le u < F} x_{nc,v+u}$$

Pooling can use other operators, for example **average**

$$y_{ncv} = \frac{1}{F_1 \cdot F_2} \sum_{0 \le u < F} x_{nc,v+u}$$



CNN layers summary



expression	dimensions
$y_{nkv} = b_k + \sum_{c=0}^{C-1} \sum_{u=0}^{F-1} f_{kcu} \cdot x_{n,c,v+u}$	$O = I - F + 1$
$y_{ncu} = \max\{0, x_{ncu}\}$	$K = C, \quad O = I$
$y_{ncv} = \max_{0 \le u < F} x_{nc,v+u}$	$O = I - F + 1$

Deep convolutional neural networks

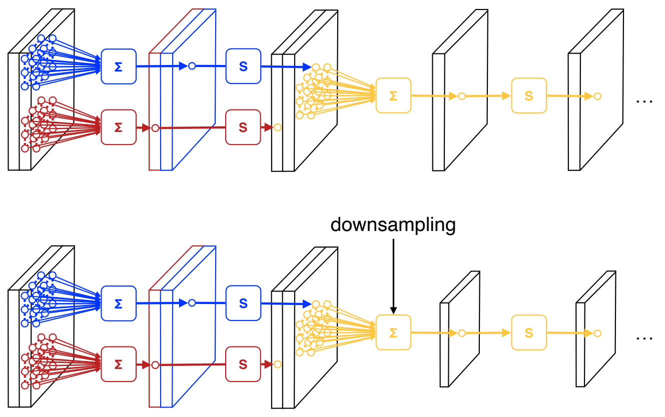
A long sequence of layers

A **deep convolutional neural network** is a chain of several layers.

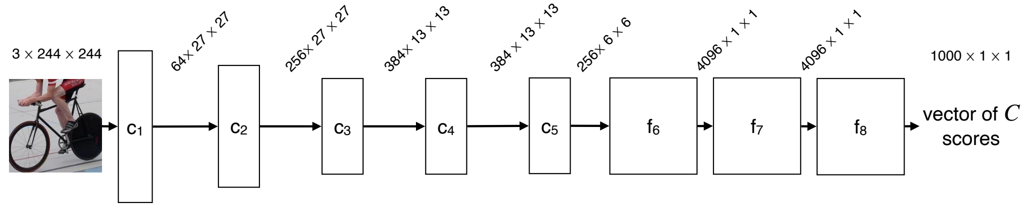
The typical pattern is to alternate linear convolution and non-linear activation, usually ReLU.

The other typical pattern is to gradually reduce the spatial resolution (via downsampling) and increase the number of feature channels.

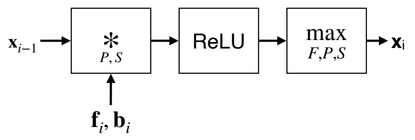
Max-pooling is often used, in combination with downsampling, to reduce resolution further.



AlexNet: a CNN for image classification



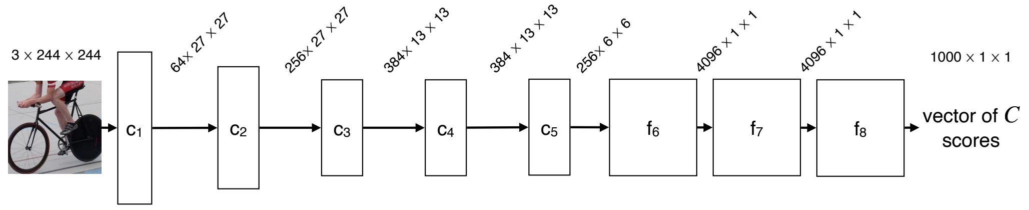
K filters number	64	256	384	384	256	4096	4096	1000
F filter size	11	5	3	3	3	6	1	1
S filter stride	4	1	1	1	1	1	1	1
P filter padding	0	2	1	1	1	0	0	0
F' pooling size	3	3	-	-	3	-	-	-
S' pooling stride	2	2	-	-	2	-	-	-
P' pooling padding	1	0	-	-	0	-	-	-



AlexNet contains 8 **blocks**, each formed by:

- A linear convolution operator (with padding/downsampling)
- A ReLU operator (except for f_8)
- An optional max pooling operator (with padding/downsamplin

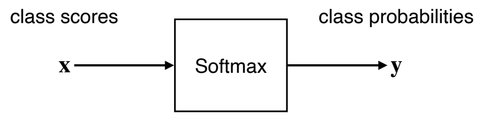
AlexNet: a CNN for image classification



The output is a 1000 × 1 × 1 tensor.

Each entry represents the score for the hypothesis that the image contains one out of a 1000 possible classes (defined in ImageNet).

Class scores are converted into probabilities by using the **softmax layer** (multi-class generalization of the sigmoid)



$$y_c = \frac{e^{x_c}}{\sum_{k=0}^{C-1} e^{x_k}}$$

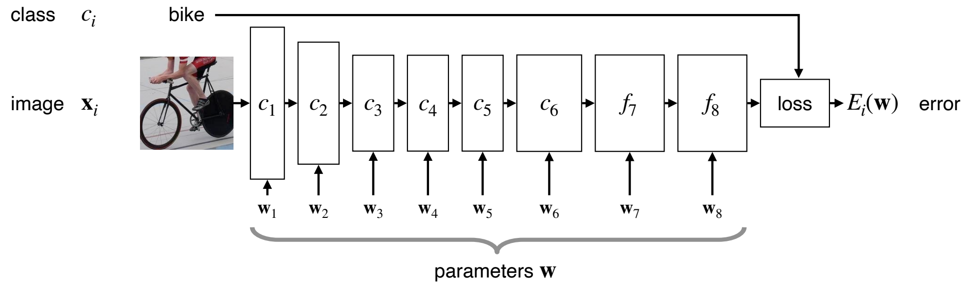
The perceptron

Convolutional networks

Learning via SGD

Evaluation

Learning a CNN



Given a dataset $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_N, y_N)$ the total error is obtained by averaging the cross-entropy loss.

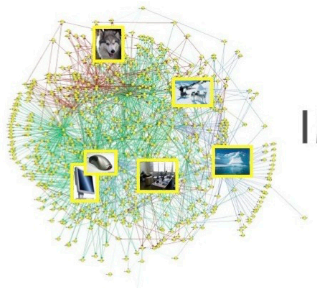
$$E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} E_i(\mathbf{w}), \qquad E_i(\mathbf{w}) = \ell(c_i, \Phi(\mathbf{x}_i))$$

The goal is to optimize this energy over the model parameters $\mathbf{w}$.

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} E(\mathbf{w})$$

Learning a CNN

ImageNet benchmark data



IMAGENET

A CNN classifiers has millions of parameters. Hence, **learning requires massive amounts of data**.

ImageNet is a large collection of labelled image.

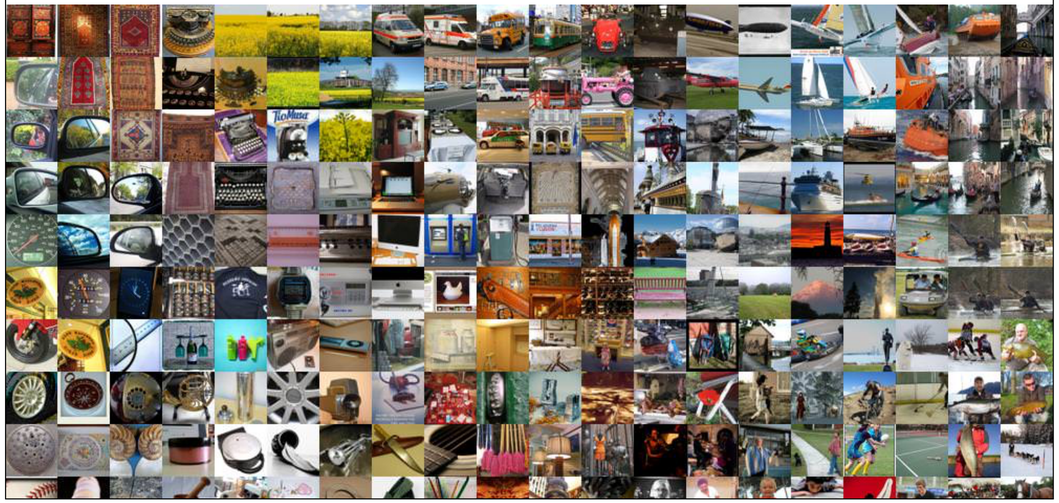
The standard subset (ILSVRC12) contains

- 1,000 object classes
- ~1,000 example images for each class
- 1.2M training images in total

Without ImageNet (or a similar dataset) it would have been impossible to develop modern deep neural networks for computer vision.

Learning a CNN

ImageNet benchmark data



Stochastic gradient descent

The objective function is an average over $N = 1.2$M data points, and so is the gradient. The cost of a single gradient descent update is way too large to be practical.

$$E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} E_i(\mathbf{w}) \quad \Rightarrow \quad \nabla E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \nabla E_i(\mathbf{w})$$

Stochastic gradient

Approximate the gradient **by sampling a single data point** (or a small batch of size N' << N). Perform the gradient update using the approximation.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla E_i(\mathbf{w}_t), \qquad i \sim \mathcal{U}(\{1,2,\ldots,N\})$$

uniform distribution

Momentum

SGD can be accelerated by denoising the gradient estimate using a moving average. This average is called **momentum**.

$$\mathbf{m}_{t+1} = 0.9\,\mathbf{m}_t + \eta_t \nabla E_i(\mathbf{w}_t), \quad \mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{m}_{t+1}$$

Learning a CNN

Further details and practical notes

Epochs & mini-batches

In practice, the data is visited not randomly, but in random order (without repetitions). A full pass is called an **epoch**.

Gradients are estimated by averaging **mini-batches** of 10-1000 examples. This takes advantage of parallel hardware such as GPUs.

Annealing schedule

The learning rate η_t is gradually reduced over time, usually by a factor 10 when no progress is observed.

This allows SGD to slow down and more accurately land on an optimum as the latter is approached.

Time required

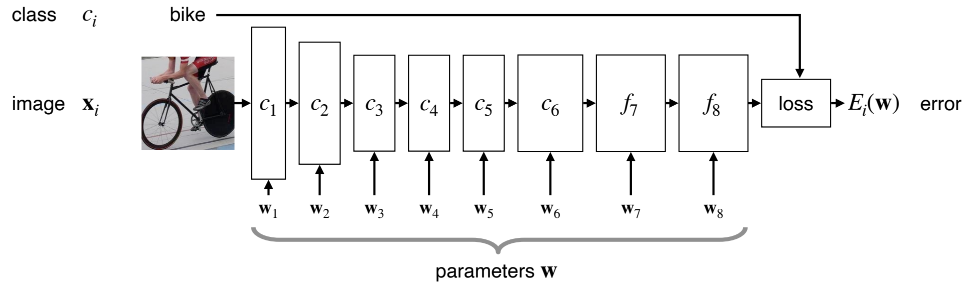
On a fast GPU, it is possible to process ~1k images per second for AlexNet.

An epoch thus lasts for 20 minutes. 40-100 epochs are required, requiring 13-33 hours (faster training requires tricks such as batch normalisation).

On a CPU, this could be 100 x slower (four months).

Some networks are much slower (10 - 50 x).

The need for gradients



In order to train a neural network we minimize the average prediction error

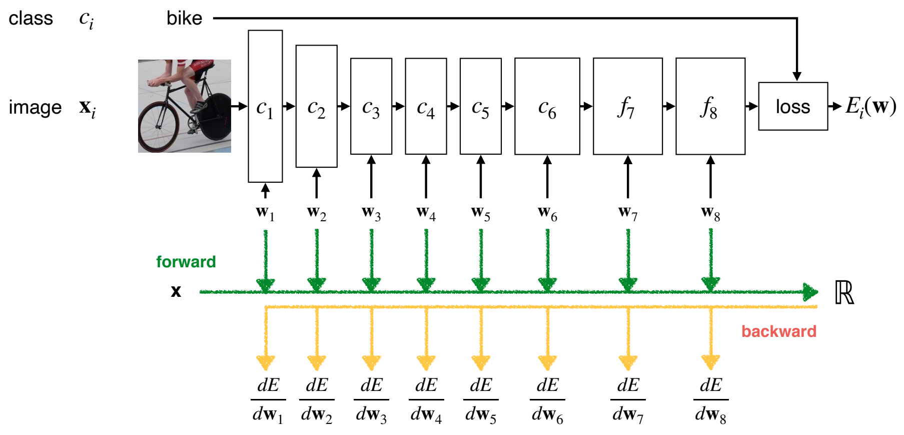
$$\operatorname*{argmin}_{\mathbf{w}_1,\ldots,\mathbf{w}_8} E(\mathbf{w}_1, \ldots, \mathbf{w}_8)$$

In order to do so, we require the **gradients of the error** with respect to all parameters

$$\nabla E = \left(\frac{dE}{d\mathbf{w}_1}, \ldots, \frac{dE}{d\mathbf{w}_8} \right)$$

Backpropagation

An efficient algorithm to compute the gradients



AutoDiff

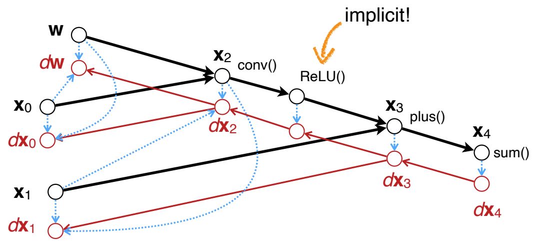
Automatic backpropagation

Modern machine learning toolboxes provide **AutoDiff**.

This means that calculations can be performed as normal in a programming language.

Underneath, the toolbox builds a compute graph.

Eventually, gradients can be requested.



```python
import torch

# Define two random inputs, both requiring grads
x0 = torch.randn(1,3,20,20, requires_grad=True)
x1 = torch.randn(1,10,18,18, requires_grad=True)

# Get a convolutional layer. Implicitly this contains
# a parameter tensor conv.weight with requires_grad=True
conv = torch.nn.Conv2d(3,10,3)

# Intermediate calculations
x2 = conv(x0)
x3 = torch.nn.ReLU()(x2) + x1

# Obtain a scalar output by summing everything
x4 = x3.sum()

# Invoke AutoGrad to compute gradients
x4.backward()

# Print gradient shapes (just to check)
print(x0.grad.shape)
print(x1.grad.shape)
print(conv.weight.grad.shape)
```

The perceptron

Convolutional networks

Learning via SGD

Evaluation

Evaluating deep networks

General approach

Evaluation is not dissimilar to any other machine learning method, such as SVMs or the perceptron.

Evaluation must always be done on a **held-out validation or test set**. This is because we need to test generalization, not just model fitting.

$$E(\Phi) = \frac{1}{|\mathcal{D}_{\text{validation}}|} \sum_{(\mathbf{x},c)\in\mathcal{D}_{\text{validation}}} \text{err}(\Phi(\mathbf{x}), c)$$

Most benchmarks provide validation data for this purpose.

Evaluation can use the same loss used for training. However, it is not uncommon to evaluate with respect to other, more meaningful losses **err** as well.

Top-k error

For classification problems, there are two popular losses.

Classification error: the percentage of incorrectly classified image in the validation set.

Top-k error: the percentage of images whose ground truth class is not contained in the top-k more likely classes according to the model.

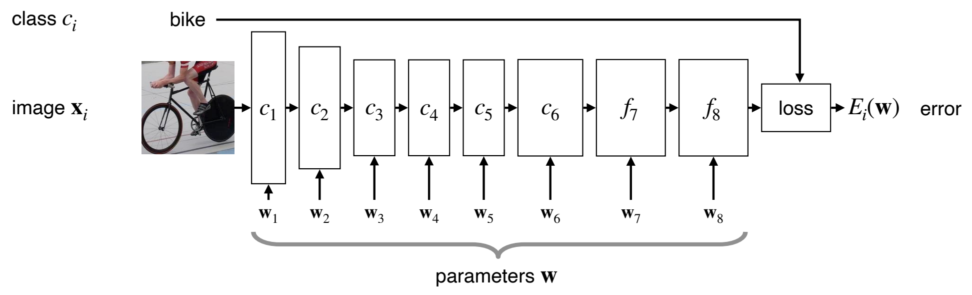
The top-k error requires the network to estimate confidences. Top-1 is the same as the classification error.

AIMS Big Data Course

Introduction to deep learning

Part 2: Backpropagation and automatic differentiation

The need for gradients



In order to train a neural network we minimise the average prediction error

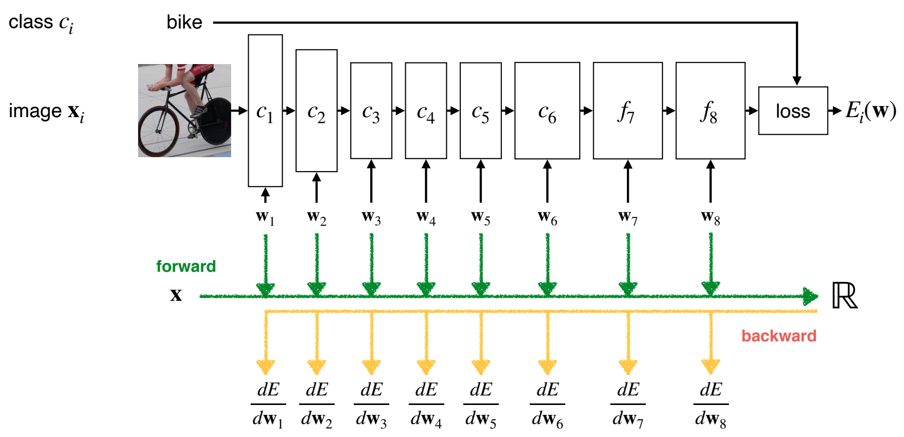
$$\operatorname*{argmin}_{\mathbf{w}_1,\ldots,\mathbf{w}_8} E(\mathbf{w}_1, \ldots, \mathbf{w}_8)$$

In order to do so, we require the **gradients of the error** with respect to all parameters

$$\nabla E = \left(\frac{dE}{d\mathbf{w}_1}, \ldots, \frac{dE}{d\mathbf{w}_8} \right)$$

Backpropagation

An efficient algorithm to compute the gradients



Chain rule: scalar version

$$\begin{tikzcd}
x_0 \arrow[r] & \boxed{f_1} \arrow[r] & x_1 \arrow[r] & \boxed{f_2} \arrow[r] & \cdots \arrow[r] & \boxed{f_{n-1}} \arrow[r] & x_{n-1} \arrow[r] & \boxed{f_n} \arrow[r] & x_n
\end{tikzcd}$$

Chain rule (scalar version)

$$\begin{tikzcd}
x_n & \boxed{f_n} \arrow[l] & x_{n-1} \arrow[l] & \boxed{f_{n-1}} \arrow[l] & \cdots \arrow[l] & \boxed{f_2} \arrow[l] & x_1 \arrow[l] & \boxed{f_1} \arrow[l] & x_0 \arrow[l]
\end{tikzcd}$$

A composition of n functions

$$\begin{tikzcd}
x_n = & (f_n \arrow[d] & \circ & f_{n-1} \arrow[d] & \circ \cdots \circ & f_2 \arrow[d] & \circ & f_1) \arrow[d] & (x_0) \\
\frac{dx_n}{dx_0} = & \frac{df_n}{dx_{n-1}} & \times & \frac{df_{n-1}}{dx_{n-2}} & \times \cdots \times & \frac{df_2}{dx_1} & \times & \frac{df_1}{dx_0} &
\end{tikzcd}$$

The derivative is obtained by using the chain rule

The vec operator

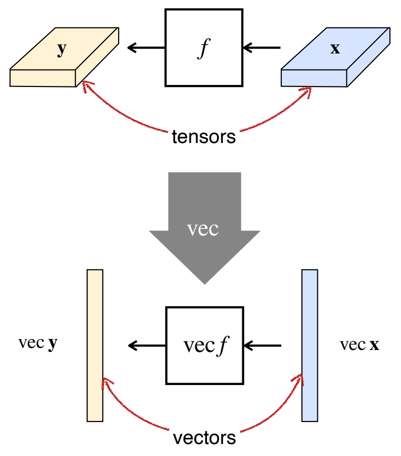
Reshaping tensors into vectors

The vec **operator** rearranges the elements of a tensor as a column vector, unrolling the tensor dimensions.

The order of unrolling is not essential, but a consistent convention must be used. PyTorch uses the **row major** convention:

$$\operatorname{vec}\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} y_{00} \\ y_{01} \\ y_{10} \\ y_{11} \end{bmatrix}$$

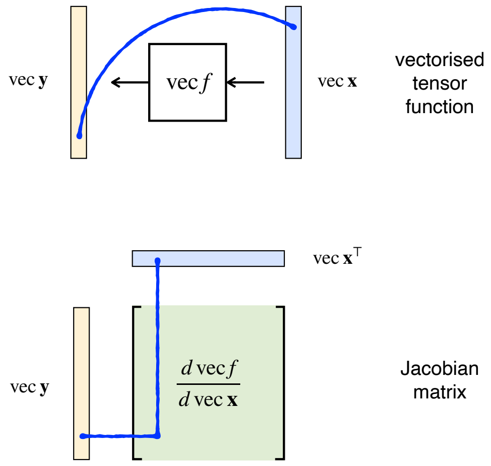
By reshaping tensors in this manner, a tensor layer $\mathbf{y} = f(\mathbf{x})$ can be thought of as a vector layer $\operatorname{vec}\mathbf{y} = f(\operatorname{vec}\mathbf{x})$.



Derivative of tensor-valued functions

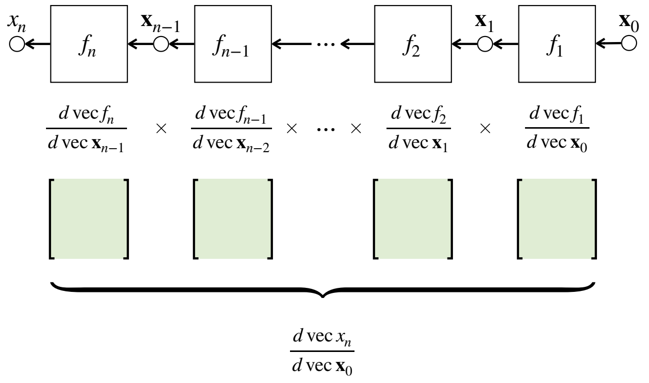
We use the vec operator to reduce a tensor derivative to a Jacobian matrix:

1. vec converts the tensor function $\mathbf{y} = f(\mathbf{x})$ to a vector function $\operatorname{vec}\mathbf{y} = (\operatorname{vec} f)(\operatorname{vec}\mathbf{x})$.
2. The derivative of a vector function is its Jacobian matrix.
3. The Jacobian matrix contains the derivative of each element of the output vector $\operatorname{vec}\mathbf{y}$ with respect to each element of the input vector $\operatorname{vec}\mathbf{x}$.



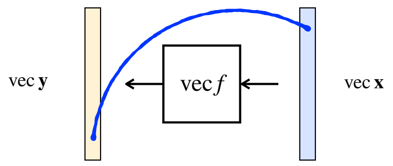
Chain rule (tensor version)

Using vec and matrix notation

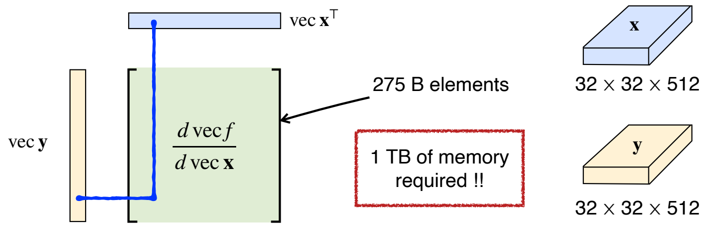


$$\underbrace{\frac{d\operatorname{vec} f_n}{d\operatorname{vec}\mathbf{x}_{n-1}} \times \frac{d\operatorname{vec} f_{n-1}}{d\operatorname{vec}\mathbf{x}_{n-2}} \times \cdots \times \frac{d\operatorname{vec} f_2}{d\operatorname{vec}\mathbf{x}_1} \times \frac{d\operatorname{vec} f_1}{d\operatorname{vec}\mathbf{x}_0}}_{\frac{d\operatorname{vec} x_n}{d\operatorname{vec}\mathbf{x}_0}}$$

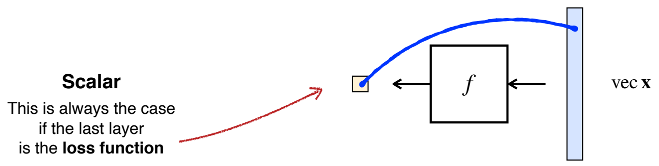
The (unbearable) size of tensor derivatives



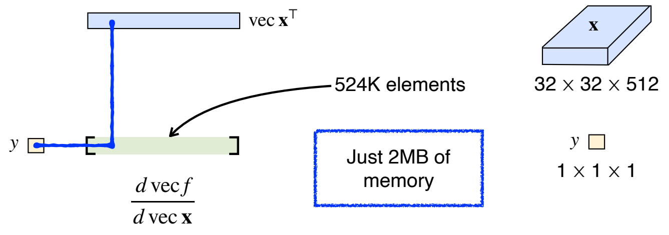
The size of these **Jacobian** matrices is **huge**. Example:



Unless the output is a scalar

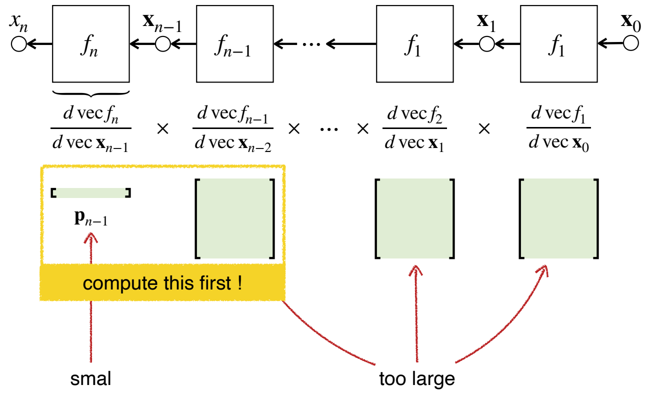

Scalar
This is always the case if the last layer is the loss function
f
vec x

Now the Jacobian reduces to a **gradient** and has the same size as $\mathbf{x}$. Example:



Backpropagation

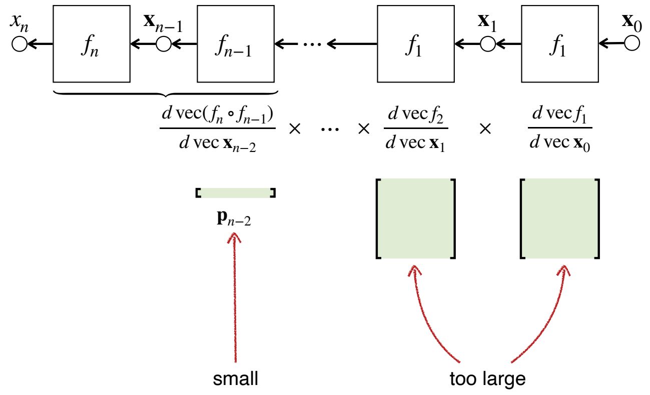
Assume that x_n is a scalar



x_n
f_n
$\mathbf{x}_{n-1}$
f_{n-1}
$\cdots$
f_1
$\mathbf{x}_1$
f_1
$\mathbf{x}_0$
$\underbrace{\frac{d\,\mathrm{vec}\,f_n}{d\,\mathrm{vec}\,\mathbf{x}_{n-1}}} \times \frac{d\,\mathrm{vec}\,f_{n-1}}{d\,\mathrm{vec}\,\mathbf{x}_{n-2}} \times \cdots \times \frac{d\,\mathrm{vec}\,f_2}{d\,\mathrm{vec}\,\mathbf{x}_1} \times \frac{d\,\mathrm{vec}\,f_1}{d\,\mathrm{vec}\,\mathbf{x}_0}$
$\mathbf{p}_{n-1}$
compute this first !
smal
too large

Backpropagation

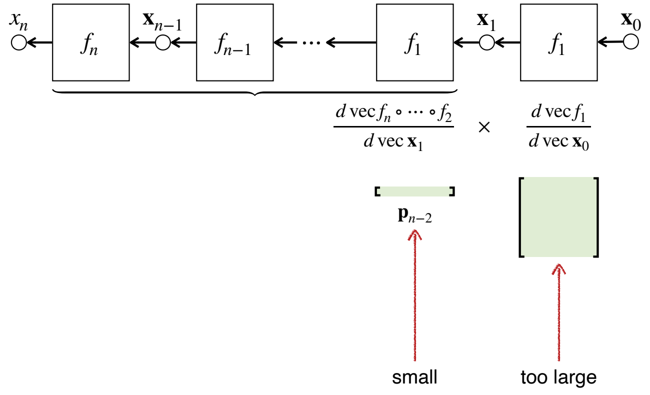
Assume that x_n is a scalar



x_n
f_n
$\mathbf{x}_{n-1}$
f_{n-1}
$\cdots$
f_1
$\mathbf{x}_1$
f_1
$\mathbf{x}_0$
$\underbrace{\frac{d\,\mathrm{vec}(f_n \circ f_{n-1})}{d\,\mathrm{vec}\,\mathbf{x}_{n-2}}} \times \cdots \times \frac{d\,\mathrm{vec}\,f_2}{d\,\mathrm{vec}\,\mathbf{x}_1} \times \frac{d\,\mathrm{vec}\,f_1}{d\,\mathrm{vec}\,\mathbf{x}_0}$
$\mathbf{p}_{n-2}$
small
too large

Backpropagation

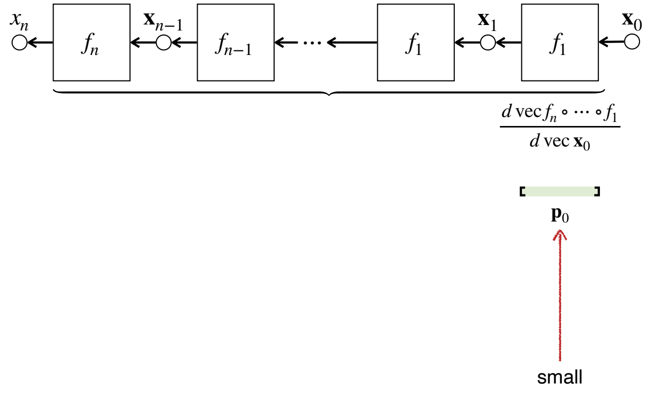
Assume that x_n is a scalar



x_n
f_n
$\mathbf{x}_{n-1}$
f_{n-1}
$\cdots$
f_1
$\mathbf{x}_1$
f_1
$\mathbf{x}_0$
$\underbrace{\frac{d\,\mathrm{vec}\,f_n \circ \cdots \circ f_2}{d\,\mathrm{vec}\,\mathbf{x}_1}} \times \frac{d\,\mathrm{vec}\,f_1}{d\,\mathrm{vec}\,\mathbf{x}_0}$
$\mathbf{p}_{n-2}$
small
too large

Backpropagation

Assume that x_n is a scalar



Vector-Jacobian product f^{BP}

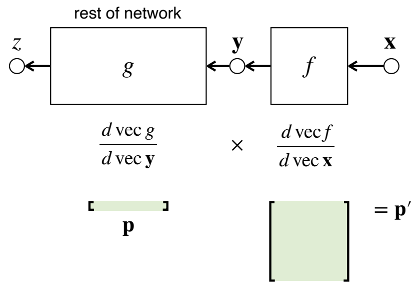
The key step is the calculation of the **vector-Jacobian product**

$$\mathbf{p}' = f^{\mathrm{BP}}(\mathbf{p};\mathbf{x}) = \mathbf{p}\cdot\frac{d\,\mathrm{vec}\,f}{d\,\mathrm{vec}\,\mathbf{x}}$$

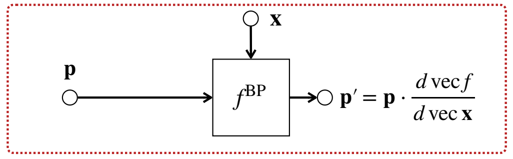
The result $\mathbf{p}'$ is a vector that has the same size as $\mathbf{x}$, so not too large.

The Jacobian matrix is still too large to explicitly compute.

The key idea is to use layer-specific optimisation to compute f^{BP} *without* computing the Jacobian matrix explicitly.



rest of network
z
g
$\mathbf{y}$
f
$\mathbf{x}$
$\frac{d\,\mathrm{vec}\,g}{d\,\mathrm{vec}\,\mathbf{y}}$
$\times$
$\frac{d\,\mathrm{vec}\,f}{d\,\mathrm{vec}\,\mathbf{x}}$
$\mathbf{p}$
$= \mathbf{p}'$



$\mathbf{x}$
$\mathbf{p}$
f^{BP}
$\mathbf{p}' = \mathbf{p}\cdot\frac{d\,\mathrm{vec}\,f}{d\,\mathrm{vec}\,\mathbf{x}}$

An example of f^{BP}

Sigmoid layer

Assume that $\mathbf{x}$ is a vector (otherwise use vec).

Let $\mathbf{y} = f(\mathbf{x})$ be the **sigmoid activation** layer:

$$f(\mathbf{x}) = \begin{bmatrix}\sigma(x_1)\\ \sigma(x_2)\\ \vdots \\ \sigma(x_C)\end{bmatrix}, \quad \sigma(x) = \frac{e^x}{e^x + e^{-x}}.$$

The Jacobian is then given by:

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{d\sigma(x_1)}{dx_1} & \frac{d\sigma(x_1)}{dx_2} & \cdots & \frac{d\sigma(x_1)}{dx_C} \\ \frac{d\sigma(x_2)}{dx_1} & \frac{d\sigma(x_2)}{dx_2} & \cdots & \frac{d\sigma(x_2)}{dx_C} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{d\sigma(x_C)}{dx_1} & \frac{d\sigma(x_C)}{dx_2} & \cdots & \frac{d\sigma(x_C)}{dx_C} \end{bmatrix}.$$

Most derivatives are equal to zero:

$$\frac{d\sigma(x_c)}{dx_k} = \begin{cases}\dot\sigma(x_c), & c = k,\\ 0, & c \neq k.\end{cases}, \qquad \dot\sigma(x) = \frac{d\sigma}{dx}(x).$$

The *Jacobian* is the diagonal matrix

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix}\dot\sigma(x_1) & 0 & \dots & 0\\ 0 & \dot\sigma(x_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \dot\sigma(x_C)\end{bmatrix}.$$

f^{BP} is then given by

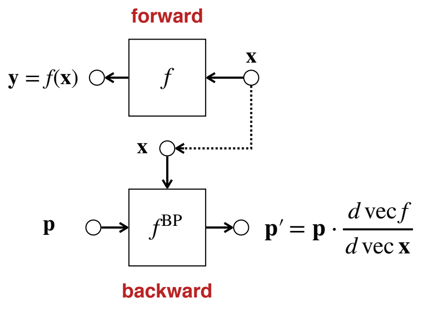
$$f^{\mathrm{BP}}(\mathbf{p};\mathbf{x}) = \mathbf{p}\cdot\frac{df}{d\mathbf{x}} = \begin{bmatrix}p_1\dot\sigma(x_1) & p_2\dot\sigma(x_2) & \cdots & p_C\dot\sigma(x_C)\end{bmatrix}.$$

f^{BP} as a reversed layer

The function f is a **forward layer** $\mathbf{y} = f(\mathbf{x})$.

The function f^{BP} defines a **backward layer** operating in the reverse direction $\mathbf{p}' = f^{\mathrm{BP}}(\mathbf{p};\mathbf{x})$.

This generates a new mirror block diagram; the forward diagram feeds into the backward diagram via $\mathbf{x}$.



forward
$\mathbf{y} = f(\mathbf{x})$
f
$\mathbf{x}$
$\mathbf{x}$
$\mathbf{p}$
f^{BP}
$\mathbf{p}' = \mathbf{p}\cdot\frac{d\,\mathrm{vec}\,f}{d\,\mathrm{vec}\,\mathbf{x}}$
backward

f^{BP} computes gradients

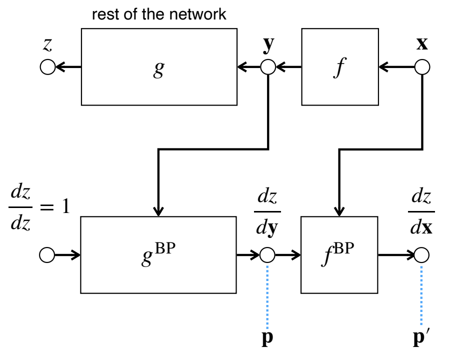
So what are these vectors $\mathbf{p}$ anyways?

Each $\mathbf{p}$ is the **gradient** of the network output z with respect to the corresponding variable $\mathbf{x}$:

$$\mathbf{p}' = \frac{dz}{d\mathbf{x}} \quad \text{or even just} \quad \mathbf{p}' = d\mathbf{x}$$

Thus f^{BP} computes a gradient out of another gradient:

$$\mathbf{p} = \frac{dz}{d\mathbf{y}} \quad \Rightarrow \quad \mathbf{p}' = f^{\mathrm{BP}}(\mathbf{p};\mathbf{x}) = \frac{dz}{d\mathbf{x}}$$



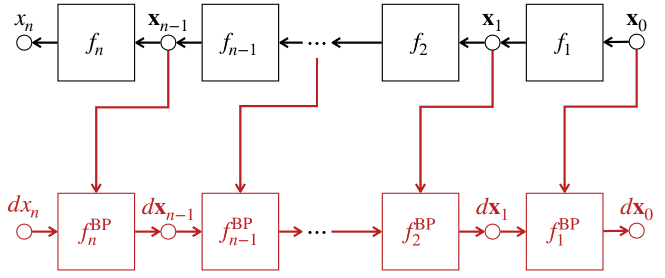
Compute graphs

Keeping track of calculations for automatic differentiation

The **compute graph** is a mechanism to keep track of the calculations in a program.

It can be used to automatically deduce which computations are required to compute the gradients.

These computations can then be added to the graph and the process repeated to obtain higher-order derivatives.



Compute graphs

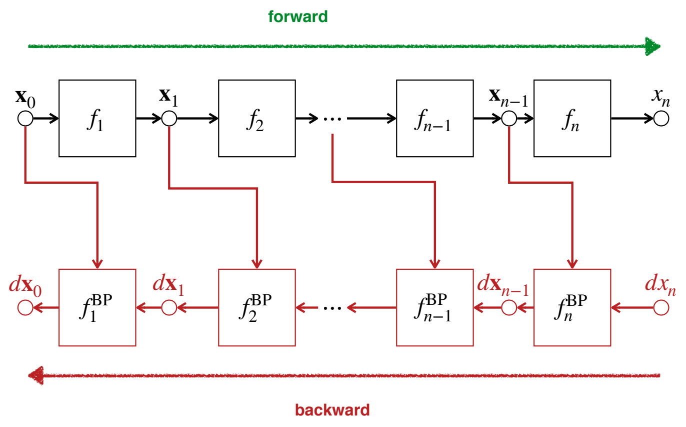
Keeping track of calculations for automatic differentiation

The **compute graph** is a mechanism to keep track of the calculations in a program.

It can be used to automatically deduce which computations are required to compute the gradients.

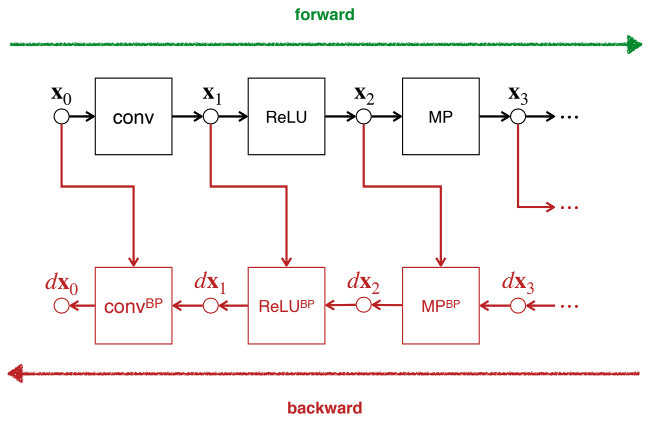
These computations can then be added to the graph and the process repeated to obtain higher-order derivatives.

The graph is more commonly shown the other way around, with the forward direction left to right.



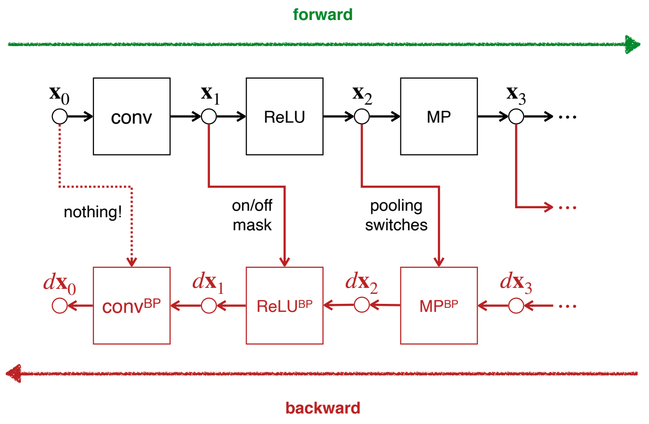
Backpropagation network

Conv, ReLU, MP and their transposed blocks



Sufficient statistics and bottlenecks

Usually much less information is needed



Automatic differentiation (AutoDiff)

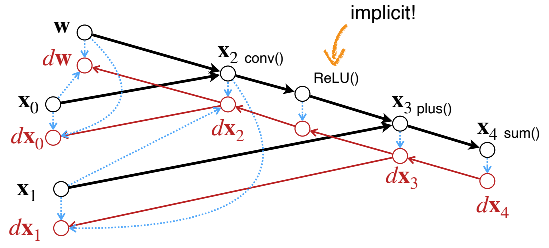
A PyTorch example

Modern machine learning toolboxes provide **AutoDiff**.

This means that calculations can be performed as normal in a programming language.

Underneath, the toolbox builds a compute graph.

Eventually, gradients can be requested.



```
import torch

# Define two random inputs, both requiring grads
x0 = torch.randn(1,3,20,20, requires_grad=True)
x1 = torch.randn(1,10,18,18, requires_grad=True)

# Get a convolutional layer. It contains
# a parameter tensor conv.weight with requires_grad=True
conv = torch.nn.Conv2d(3,10,3)

# Intermediate calculations
x2 = conv(x0)
x3 = torch.nn.ReLU()(x2) + x1
x4 = x3.sum() # Scalar!

# Invoke AutoGrad to compute the gradients
x4.backward()

# Print the gradient shapes
print(x0.grad.shape)
print(x1.grad.shape)
print(conv.weight.grad.shape)
```

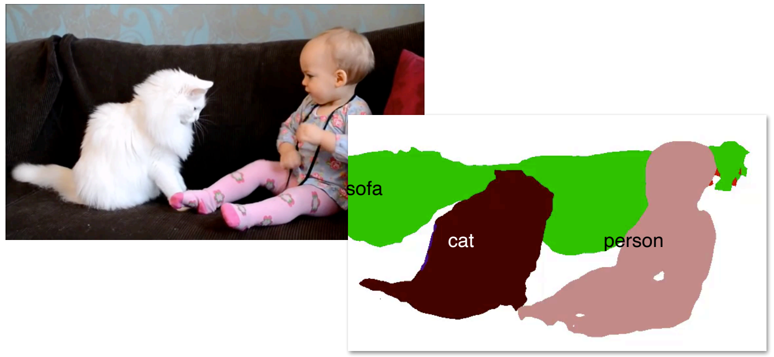
AIMS Big Data Course

Introduction to deep learning

Part 3: Applications

Semantic image segmentation

Label individual pixels



Face analysis

Detection, verification, recognition, emotion, 3D fitting

E.g. VGG-Face

Text spotting

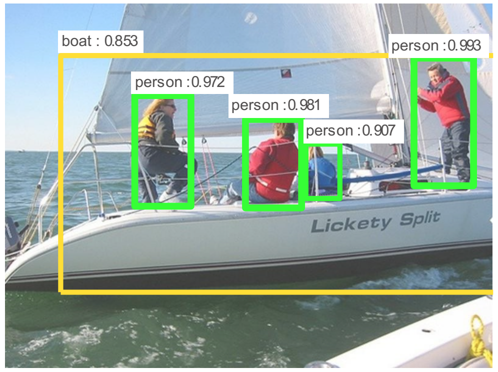
Detection, word recognition, character recognition

E.g. SynthText and VGG-Text

http://zeus.robots.ox.ac.uk/textsearch/#/search/

Object detection

Extract individual object instances



Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation
R. Girshick, J. Donahue, T. Darrell, J. Malik, CVPR 2014

Neural network architectures

Evolution

AlexNet (2012)

5 convolutional layers

3 fully-connected layers

Neural network architectures

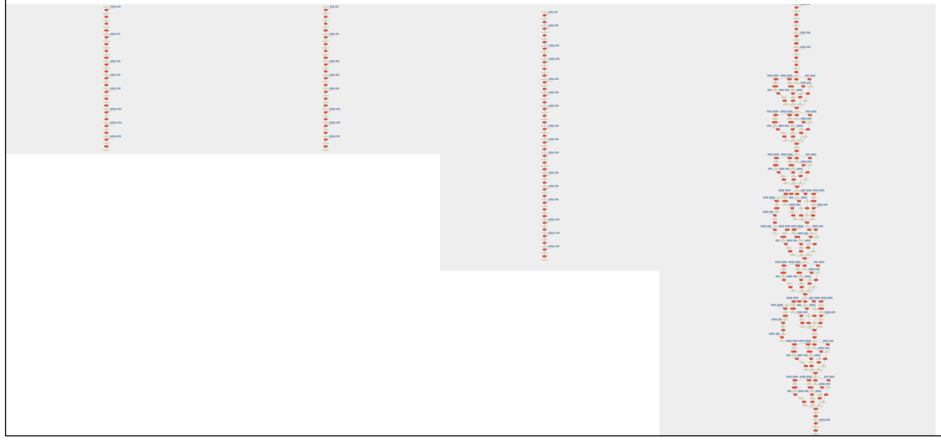
Evolution

AlexNet (2012) VGG-M (2013) VGG-VD-16 (2014)

Neural network architectures

Evolution

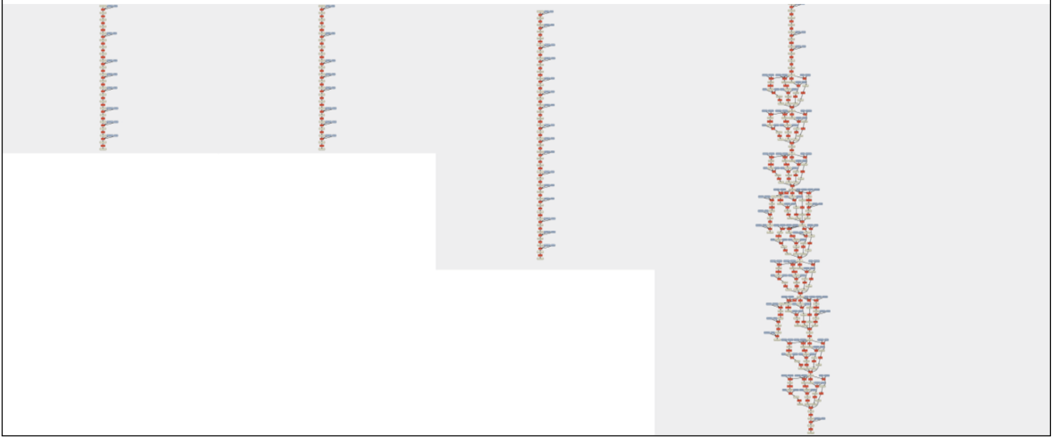
AlexNet (2012) VGG-M (2013) VGG-VD-16 (2014) GoogLeNet (2014)



Neural network architectures

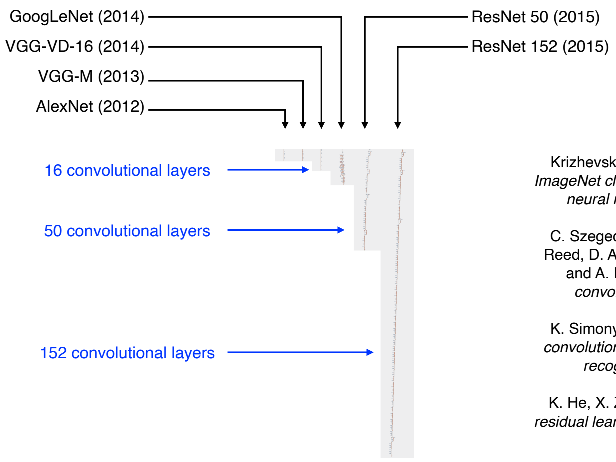
Evolution

AlexNet (2012) VGG-M (2013) VGG-VD-16 (2014) GoogLeNet (2014)



Neural network architectures

Evolution



Krizhevsky, I. Sutskever, and G. E. Hinton. *ImageNet classification with deep convolutional neural networks.* In Proc. NIPS, 2012.

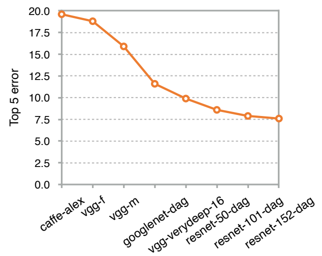
C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. *Going deeper with convolutions.* In Proc. CVPR, 2015.

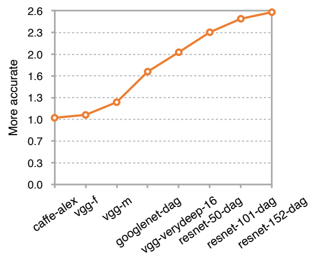
K. Simonyan and A. Zisserman. *Very deep convolutional networks for large-scale image recognition.* In Proc. ICLR, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. *Deep residual learning for image recognition.* In Proc. CVPR, 2016.

Accuracy

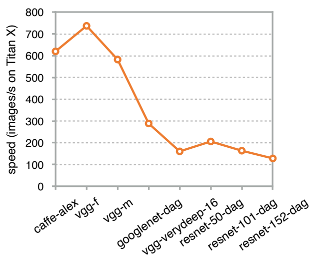
3 × more accurate in 3 years

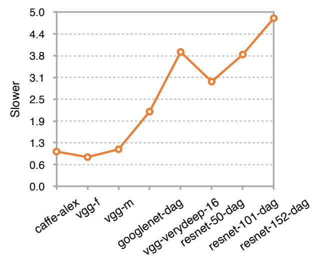




Speed

5 × slower





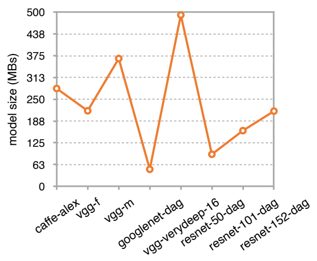
Remark: *101 ResNet* layers same size/speed as *16 VGG-VD* layers

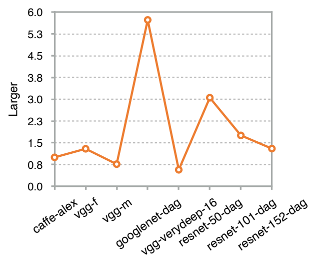
Reason: far fewer feature channels (quadratic speed/space gain)

Moral: optimize your architecture

Model size

Num. of parameters is about the same





Remark: *101 ResNet* layers same size/speed as *16 VGG-VD* layers

Reason: far fewer feature channels (quadratic speed/space gain)

Moral: optimize your architecture