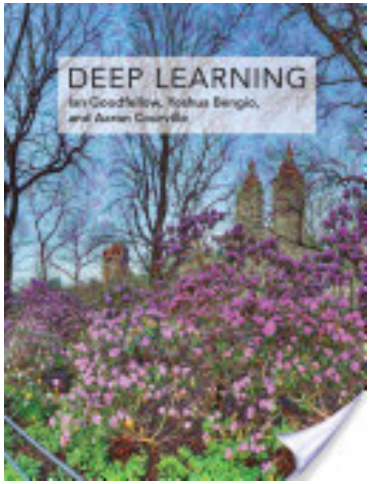
Introduction to Neural Networks and Deep Learning

Patrick Loiseau

(based on slides from Georges Quénot)

Reference

- Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep learning. MIT Press, 2016
  - In part. Chap 6 and 9
  - https://www.deeplearningbook.org/



Content

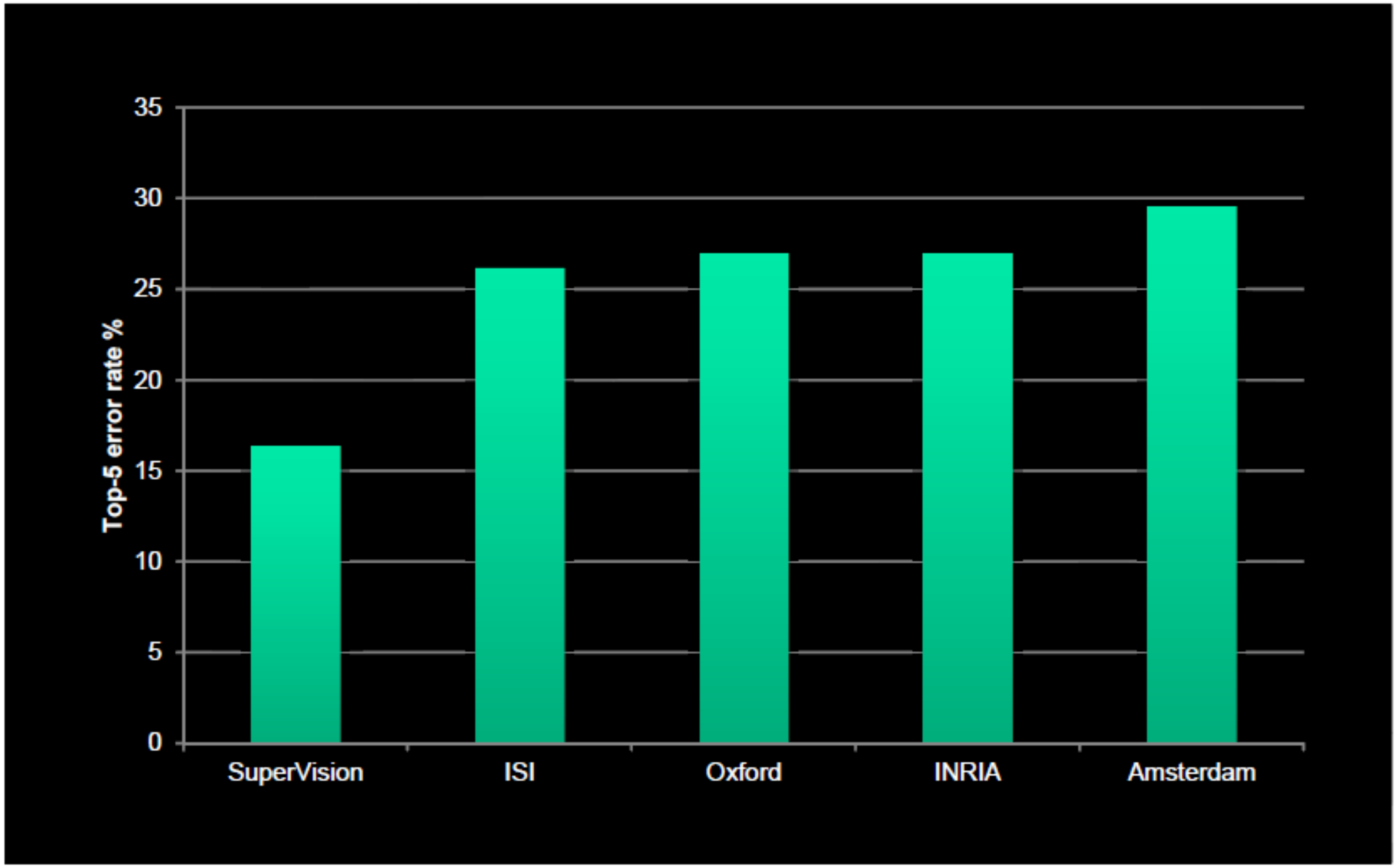
- Introduction
- Machine learning reminders
- Multilayer perceptron
- Back-propagation
- Convolutional neural networks (images)

INTRODUCTION

2012

ImageNet Classification 2012 Results

Krizhevsky et al. – **16.4% error** (top-5)
Next best (Pyr. FV on dense SIFT) – **26.2% error**



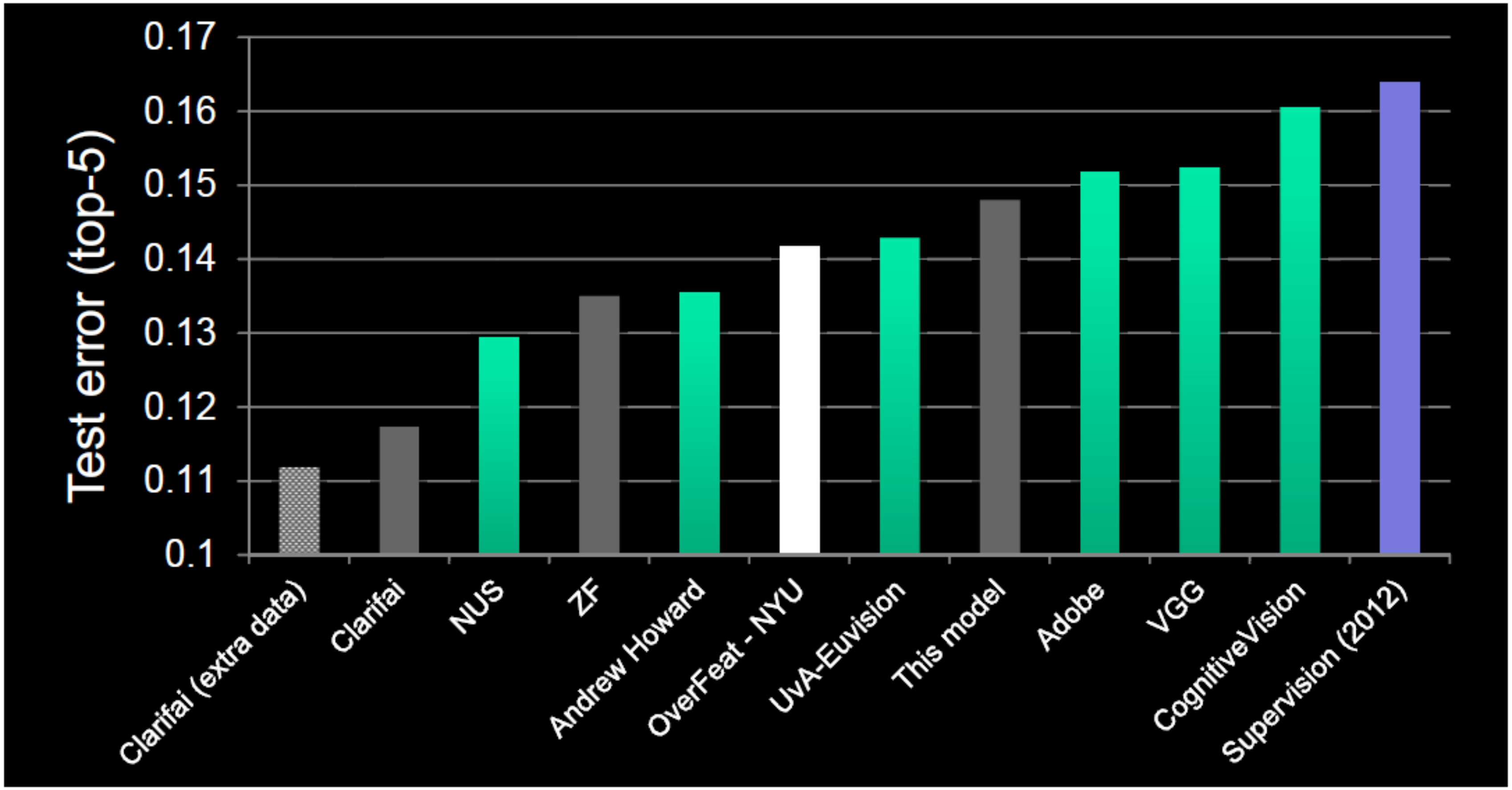
ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- 1000 visual “fine grain” categories / labels (exclusive)
- 150,000 test images (hidden “ground truth”)
- 50,000 validation images
- 1,200,000 training images
- Each training, validation or test image falls within exactly one of the 1000 categories
- Task: for each image in the test set, rank the categories from most probable to least probable
- Metric: top-5 error rate: percentage of images for which the actual category is not in the five first ranked categories
- Held from 2010 to 2015, frozen since 2012

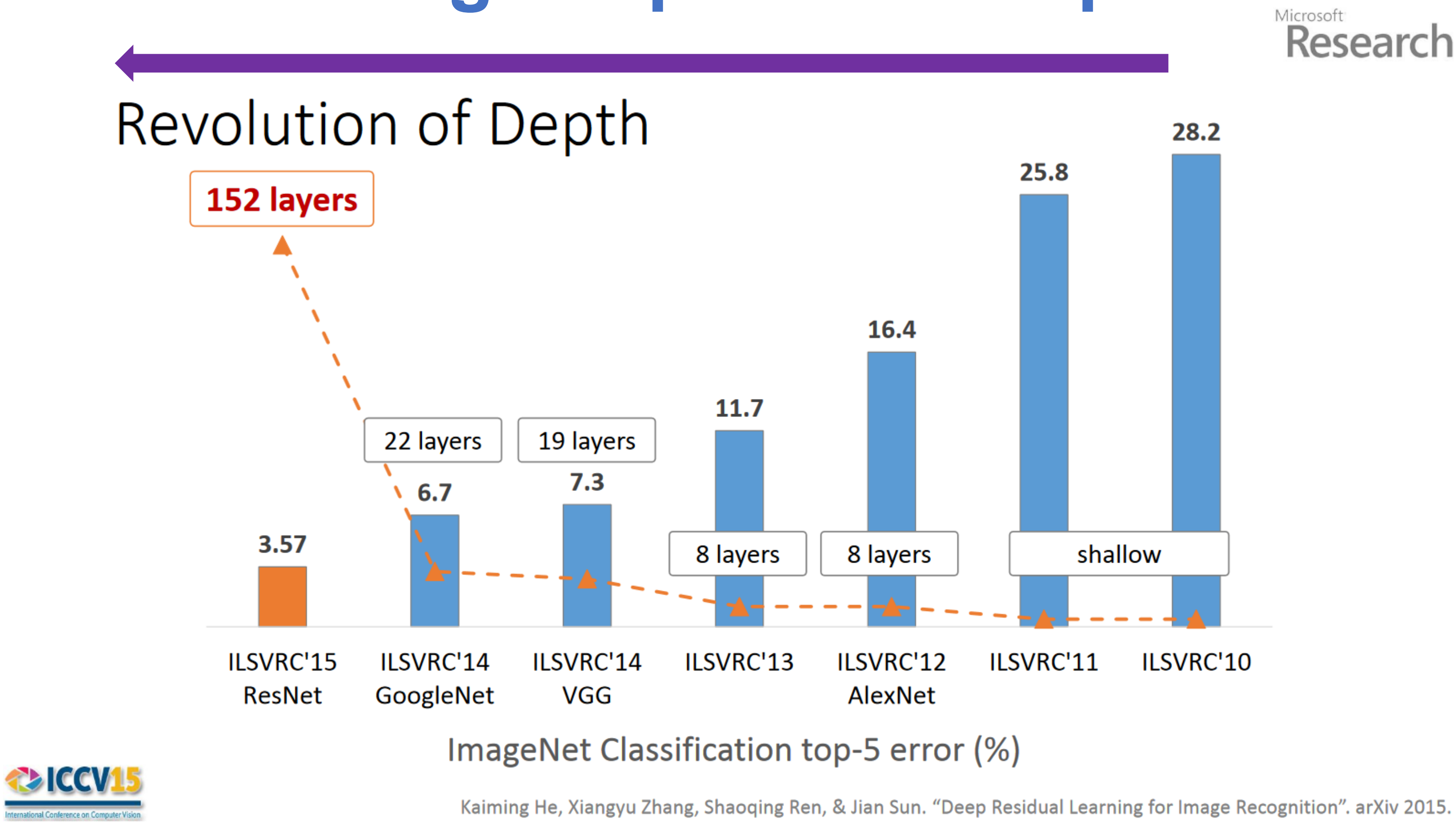
ImageNet Classification 2013 Results

http://www.image-net.org/challenges/LSVRC/2013/results.php

Demo: http://www.clarifai.com/



Going deeper and deeper



For comparison, human performance is 5.1% (Russakovsky et al.)

Deep Convolutional Neural Networks

- Decades of algorithmic improvements in neural networks (Stochastic Gradient Descent, initialization, momentum …)
- Very large amounts of properly annotated data (ImageNet)
- Huge computing power (Teraflops × weeks): GPU!
- Convolutional networks
- Deep networks ($\gg$ 3 layers)
- ReLU (Rectified Linear Unit) activation functions
- Batch normalization
- Drop Out
- …

Deep Learning is (now) EASY

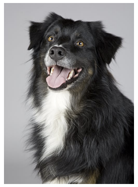
- Maths: linear algebra and differential calculus (training only)
  - $Y = A.X + B$ (with tensor extension)
  - $f(x+h) = f(x) + f'(x).h + o(h)$ (with multidimensional variables)
  - $(g\ o\ f)'(x) = (g'o\ f)(x).f'(x)$ (recursively applied)
- Tools: amazingly integrated, effective and easy to use packages
  - Mostly python interface
  - Autograd packages: only need to care of the linear algebra part
  - Main: PyTorch, TensorFlow

MACHINE LEARNING REMINDERS

Learning a target function

- Target function: $f : X \rightarrow Y$
  $x \rightarrow y = f(x)$
  - x : input object, e.g., color image
  - y : desired output, e.g., class label or image tag
  - X : set of valid input objects
  - Y : set of possible output values

$f\Big($  $\Big)$ = "cat"

$f\Big($  $\Big)$ = "dog"

$f\Big($  $\Big)$ = "car"

Set of possible color images:

$$X = \bigcup_{(w,h)\in\mathbb{N}^{*2}} [0,1]^{w\times h\times 3}$$

Set of possible image tags:

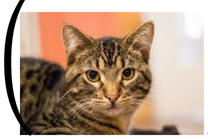
$$Y = \{\text{“cat”}, \text{“dog”} \ \ldots\}$$

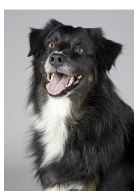
Learning a target function

- Target function: $f : X \rightarrow Y$

  $$x \rightarrow y = f(x)$$

  - x : input object, e.g., color image
  - y : desired output, e.g., class label or image tag
  - X : set of valid input objects
  - Y : set of possible output values

$f\Big($  $\Big) = \begin{pmatrix} 0.90 \\ 0.04 \\ 0.01 \\ \dots \end{pmatrix} \begin{array}{l} \leftarrow \text{“cat”} \\ \leftarrow \text{“dog”} \\ \leftarrow \text{“car”} \\ \leftarrow \dots \end{array}$

$f\Big($  $\Big) = \begin{pmatrix} 0.07 \\ 0.88 \\ 0.02 \\ \dots \end{pmatrix}$

$f\Big($  $\Big) = \begin{pmatrix} 0.02 \\ 0.03 \\ 0.86 \\ \dots \end{pmatrix}$

Set of possible color images:

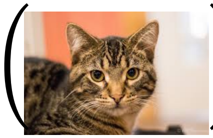
$$X = \bigcup_{(w,h)\in\mathbb{N}^{*2}} [0,1]^{w\times h\times 3}$$

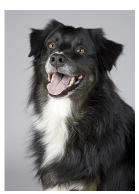
Set of possible tag scores:

$$Y = \mathbb{R}^{|\{\text{“cat”},\text{“dog”} \dots\}|} = \mathbb{R}^{c}$$

Learning a target function

- Target function: $f : X \to Y$
  $x \to y = f(x)$
  - x : input object, e.g., image descriptor
  - y : desired output, e.g., class label or image tag
  - X : set of valid input objects
  - Y : set of possible output values

$f\Big(D\Big($  $\Big)\Big) = \begin{pmatrix} 0.90 \\ 0.04 \\ 0.01 \\ \dots \end{pmatrix}$

$f\Big(D\Big($  $\Big)\Big) = \begin{pmatrix} 0.07 \\ 0.88 \\ 0.02 \\ \dots \end{pmatrix}$

$f\Big(D\Big($  $\Big)\Big) = \begin{pmatrix} 0.02 \\ 0.03 \\ 0.86 \\ \dots \end{pmatrix}$

Set of possible image descriptors:

$X = \mathbb{R}^d$ (or subset of it)

Set of possible tag scores:

$Y = \mathbb{R}^c$

D is a predefined and fixed function

from $\bigcup_{(w,h)\in\mathbb{N}^{*2}} [0,1]^{w\times h\times 3}$ to $\mathbb{R}^d$

Supervised learning

- Target function: $f : X \to Y$
  $$x \to y = f(x)$$
  - x : input object (typically vector)
  - y : desired output (continuous value or class label)
  - X : set of valid input objects
  - Y : set of possible output values

- Training data: $S = (x_i, y_i)_{(1 \leq i \leq I)}$
  - I : number of training samples

- Learning algorithm: $L : (X \times Y)^* \to Y^X$
  $$S \to f = L(S)$$

- Regression or classification system:

$$y = f(x) = [L(S)](x) = g(S, x)$$

Parametric supervised learning

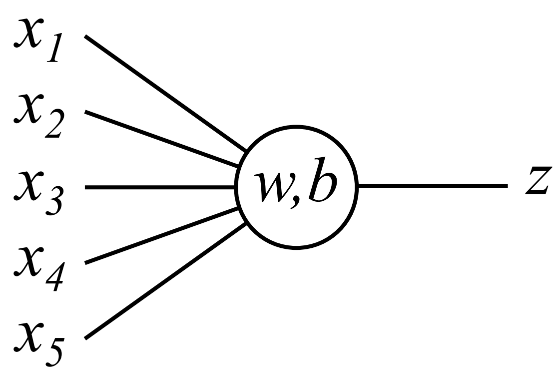
- Parameterized function: $f: \mathbb{R}^m \to Y^X$
  $\theta \to f_\theta$
- f is a "meta" function or a family of function
- Target function: $f_\theta : X \to Y$
  $x \to y = f_\theta(x)$
  - X : set of valid input objects ($\mathbb{R}^d$)
  - Y : set of possible output values ($\mathbb{R}^c$)
- Training data: $S = (x_i, y_i)_{(1 \leq i \leq I)}$
  - I : number of training samples
- Learning algorithm: $L_f : (X \times Y)^* \to \mathbb{R}^m$ (learns θ from S)
  $S \to \theta = L_f(S)$
- Regression or classification system: $y = f_\theta(x) = f(\theta, x)$

Single-label loss function

- Quantifies the cost of classification error or the "empirical risk"
- Example (Mean Square Error): $E_S(f) = \sum_{i=1}^{i=I}(f(x_i) - y_i)^2$
- If f depends on a parameter vector θ (L learns θ):
  $E_S(\theta) = \frac{1}{2}\sum_{i=1}^{i=I}(f(\theta, x_i) - y_i)^2$
- For a linear SVM with soft margin, $\theta = (w, b)$:
  $E_S(\theta) = \frac{1}{2}||w||^2 + C.\sum_{i=1}^{i=I}\max(0{,}1 - y_i(w^T x_i + b))$
- The learning algorithm aims at minimizing the empirical risk: $\theta^* = \underset{\theta}{\operatorname{argmin}}\ E_S(\theta)$

MULTILAYER PERCEPTRON

Formal neural or unit (two sub-units)



linear and vector part

$$y = \sum_j w_j x_j = w.x$$

linear combination

x : column vector
w : row vector

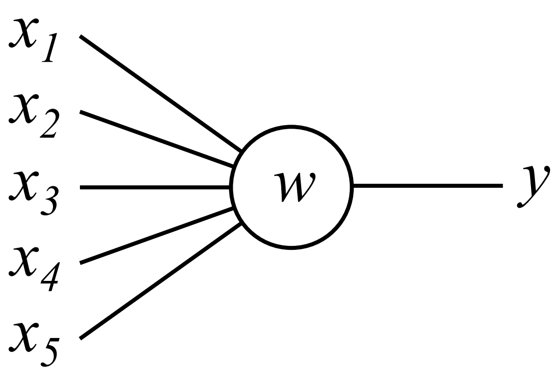
non-linear and scalar part

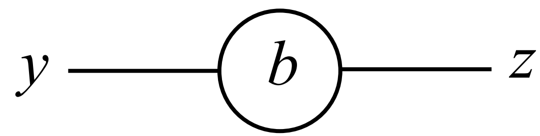
$$z = \sigma(y + b) = \frac{1}{1 + e^{y+b}}$$

ex: sigmoid function

y, b, z : scalars

Formal neural or unit (two sub-units)





linear and vector part

$$y = \sum_j w_j x_j = w.x$$

linear combination

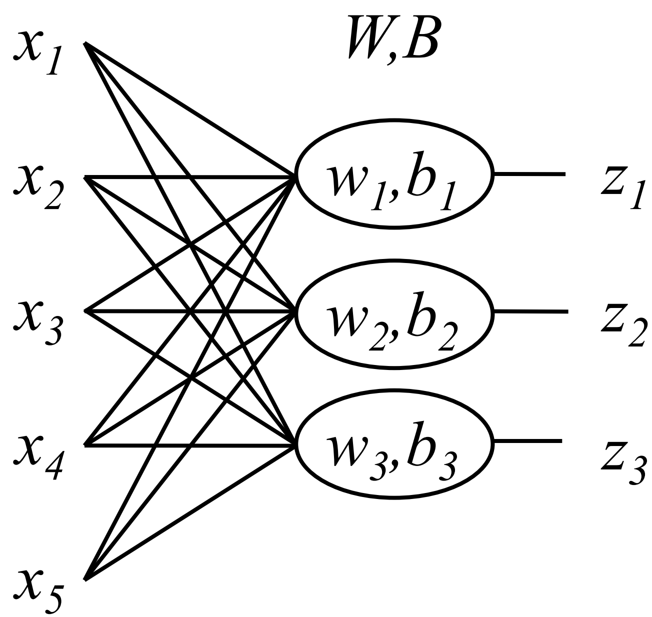
non-linear and scalar part

$$z = \sigma(y + b) = \frac{1}{1 + e^{y+b}}$$

ex: sigmoid function

Globally equivalent to a logistic regression

Neural layer (all to all, two sub-layers)



$$y_i = \sum_j w_{ij} x_j$$

matrix-vector multiplication

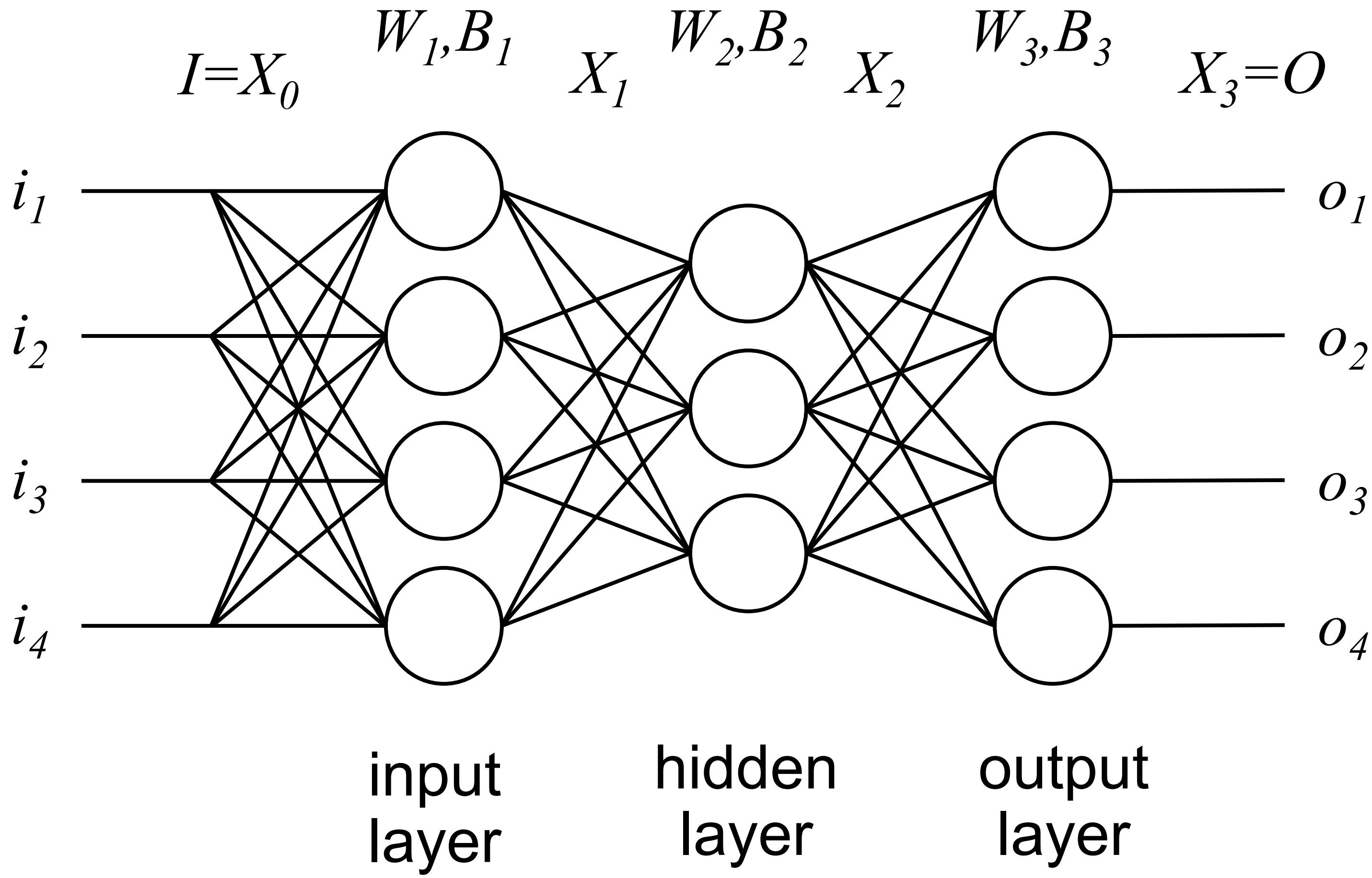
$$Y = W.X$$

$$z_i = \sigma(y_i + b_i) = \frac{1}{1 + e^{y_i + b_i}}$$

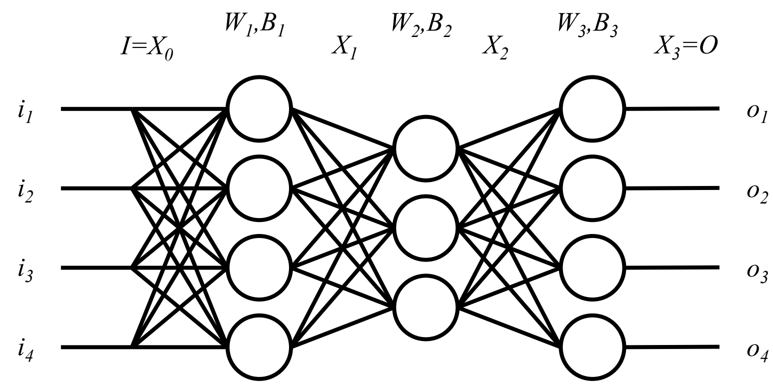
per component operation

$$z = \sigma(Y + B)$$

Multilayer perceptron (all to all)



Multilayer perceptron (all to all)



$$Y_1 = W_1.X_0 = F_1(W_1, X_0) \qquad X_1 = \sigma(Y_1 + B_1) = G_1(B_1, Y_1)$$
$$Y_2 = W_2.X_1 = F_2(W_2, X_1) \qquad X_2 = \sigma(Y_2 + B_2) = G_2(B_2, Y_2)$$
$$Y_3 = W_3.X_3 = F_3(W_3, X_2) \qquad X_3 = \sigma(Y_3 + B_3) = G_3(B_3, Y_3)$$

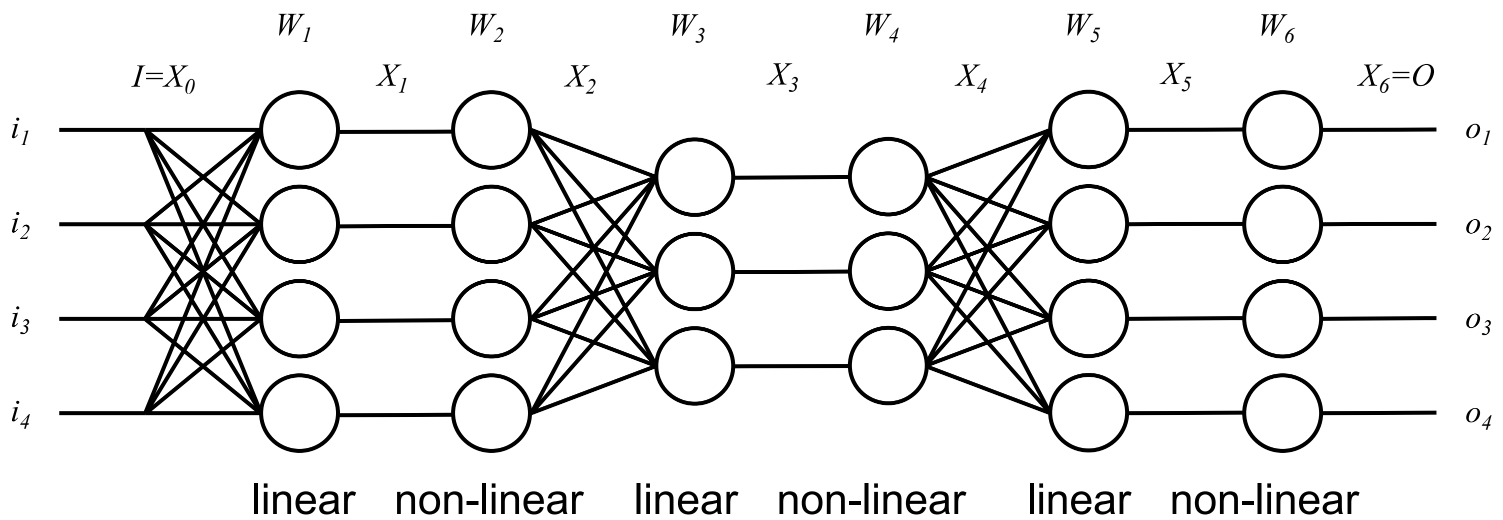
$$O = X_3 = G_3\left(B_3, F_3\left(W_3, G_2\left(B_2, F_2\left(W_2, G_1\left(B_1, F_1(W_1, X_0 = I)\right)\right)\right)\right)\right)$$

Denoting $F(W)$ so that $F(W, X) = (F(W))(X)$:

$$O = \left(G_3(B_3)\ o\ F_3(W_3)\ o\ G_2(B_2)\ o\ F_2(W_2)\ o\ G_1(B_1)\ o\ F_1(W_1)\right)(I)$$

Composition of simple functions

Splitting units and layers, renaming and renumbering:



$$X_1 = W_1.X_0 = F_1(W_1, X_0) \qquad X_2 = \sigma(X_1 + W_2) = F_2(W_2, X_1)$$

$$X_3 = W_3.X_2 = F_3(W_3, X_2) \qquad X_4 = \sigma(X_3 + W_4) = F_4(W_4, X_3)$$

$$X_5 = W_5.X_4 = F_5(W_5, X_4) \qquad X_6 = \sigma(X_5 + W_6) = F_6(W_6, X_5)$$

$$O = \left(F_6(W_6)\ o\ F_5(W_5)\ o\ F_4(W_4)\ o\ F_3(W_3)\ o\ F_2(W_2)\ o\ F_1(W_1)\right)(I) = \left(o_{n=1}^{n=6}\ F_n(W_n)\right)(I)$$

Non-linear functions

- Sigmoid: $z = \frac{1}{1+e^{y}}$
- Hyperbolic tangent: $z = \tanh y$
- Rectified Linear Unit (ReLU): $z = \max(0, y)$
- Programmable ReLU (PReLU) : $z = \max(\alpha y, y)$ with α learned (i.e. $\alpha \subset W$)
- …
- Appropriate non-linear functions leads to better performance and/or faster convergence
- Avoid vanishing / exploding gradients

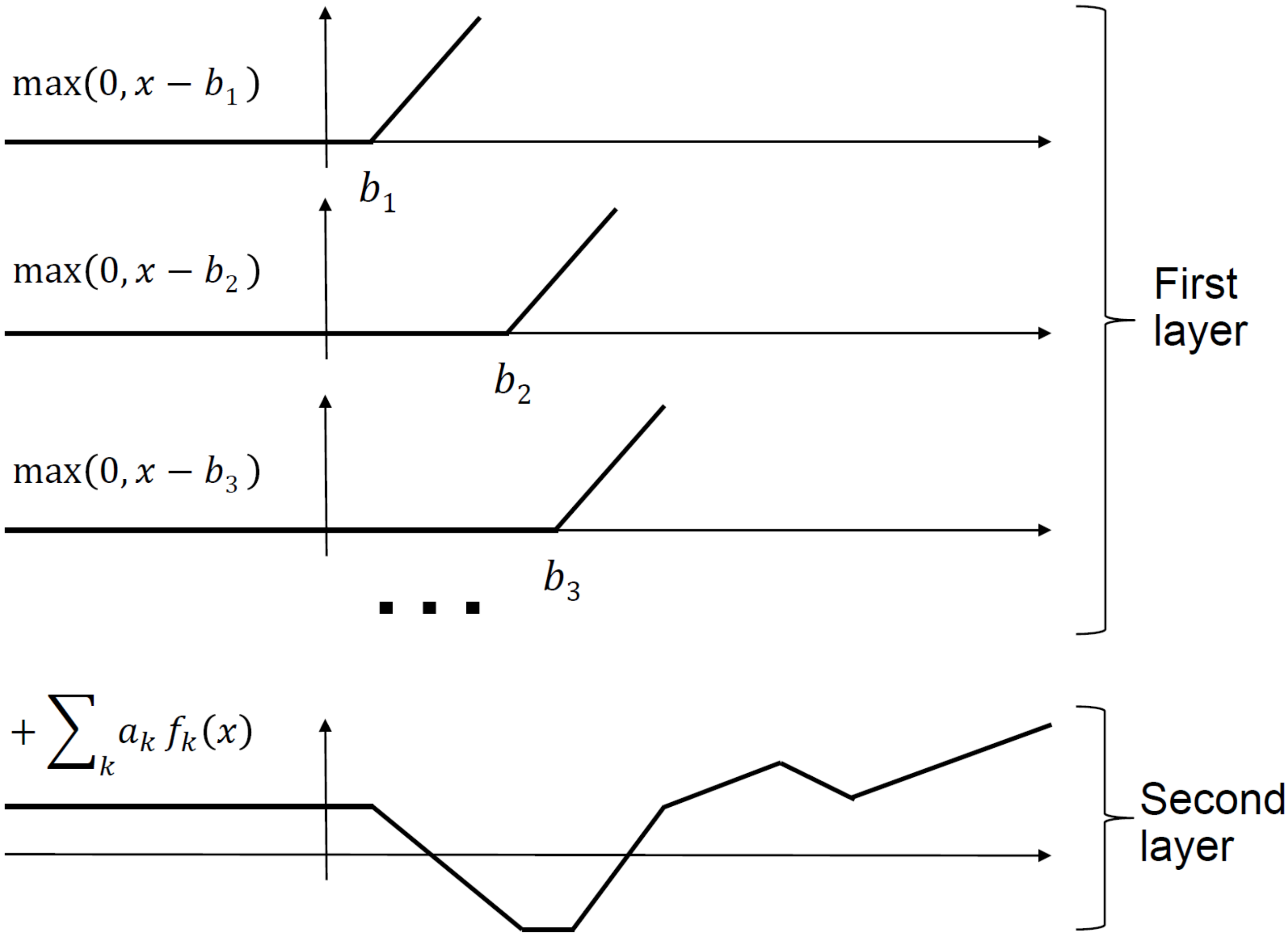
Composition of simple functions

$$f_1(x) = \max(0, x - b_1)$$

$$f_2(x) = \max(0, x - b_2)$$

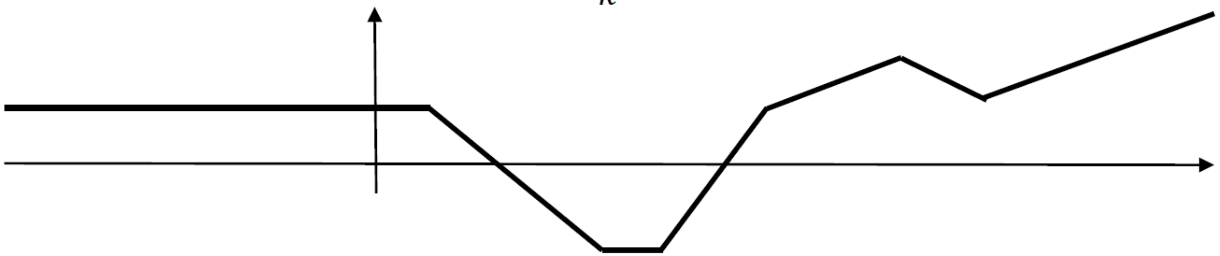
$$f_3(x) = \max(0, x - b_3)$$

$$f(x) = a_0 + \sum_k a_k f_k(x)$$



Composition of simple functions

$$f_{\theta}(x) = a_0 + \sum_k a_k \max(0, x - b_k)$$



- Model parameters: $\theta = (a_0, a_1, b_1, a_2, b_2 \ \ldots)$
- Empirical risk on training data: $E(\theta) = \sum_i \left(y_i - f_{\theta}(x_i)\right)^2$
- Find the optimal function by gradient descent on θ
- Any function can do: sigmoids, gaussians, sin/cos …
- ReLU is simpler and converges faster
- More layers: more complex functions with less parameters

Feed Forward Network

- Global network definition: $O = F(W, I)$
  ($I \equiv x \;\; O \equiv y \;\; F \equiv f \;\; W \equiv \theta$ relative to previous notations)

- Layer values: $(X_0, X_1 \ldots X_N)$
  with $X_0 = I$ and $X_N = O$ (X_n are vectors)

- Global vector of all unit parameters:
  $W = (W_1, W_2 \;\ldots\; W_N)$
  (weights by layer are concatenated, W_n can be matrices or vectors or any parameter structure, and even possibly empty)

- Feed forward: $X_{n+1} = F_{n+1}(W_{n+1}, X_n)$

- Possibly "joins" and "forks" (but no cycles)

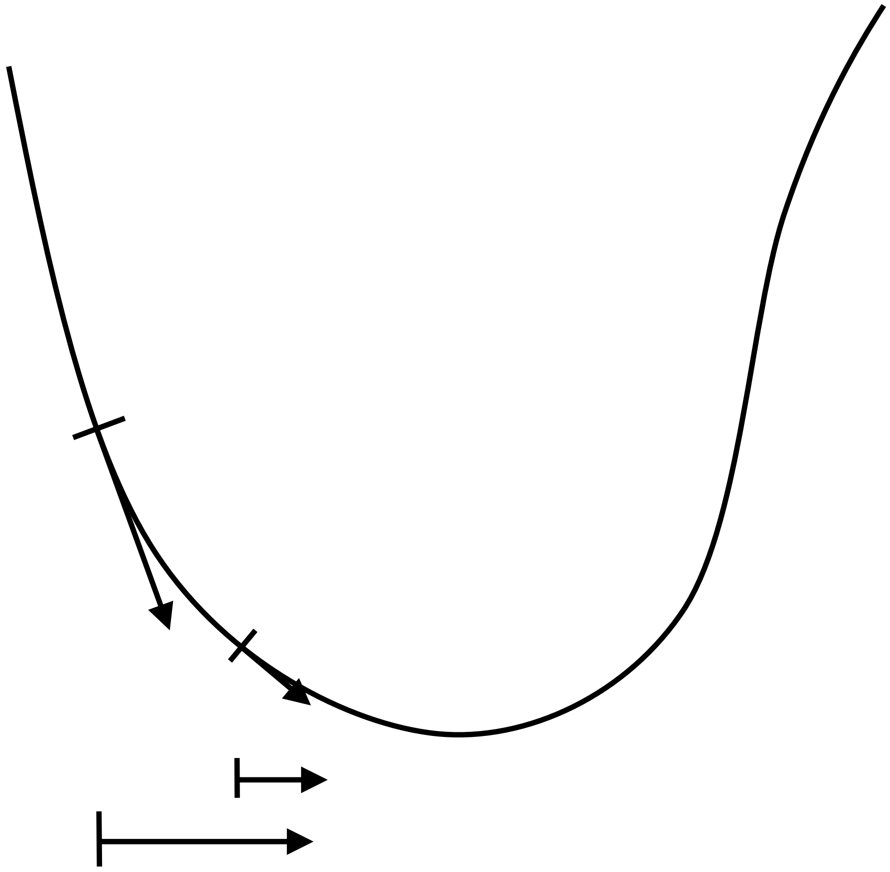
Example: the XOR function

- XOR is not linearly separable
  - Single layer with one hidden unit → no
  - Without any non-linearity → no
  - One hidden layer with 2 hidden units and ReLU → yes

Learning Algorithm

- Training set: $S = (I_i, O_i)_{(1 \leq i \leq I)}$ input-output samples
- $X_{i,0} = I_i$ and $X_{i,n+1} = F_{n+1}\left(W_{n+1}, X_{i,n}\right)$
- Note: regarding this notation the vector-matrix multiplication counts as one layer and the element-wise non-linearity counts as another one (not mandatory but greatly simplifies the layer modules' implementation)
- Error (empirical risk) on the training set:
  $E_S(W) = \sum_i (F(W, I_i) - O_i)^2 = \sum_i \left(X_{i,N} - O_i\right)^2$
- Minimization on W of $E_S(W)$ by gradient descent

Gradient descent



Stochastic gradient descent and batch processing

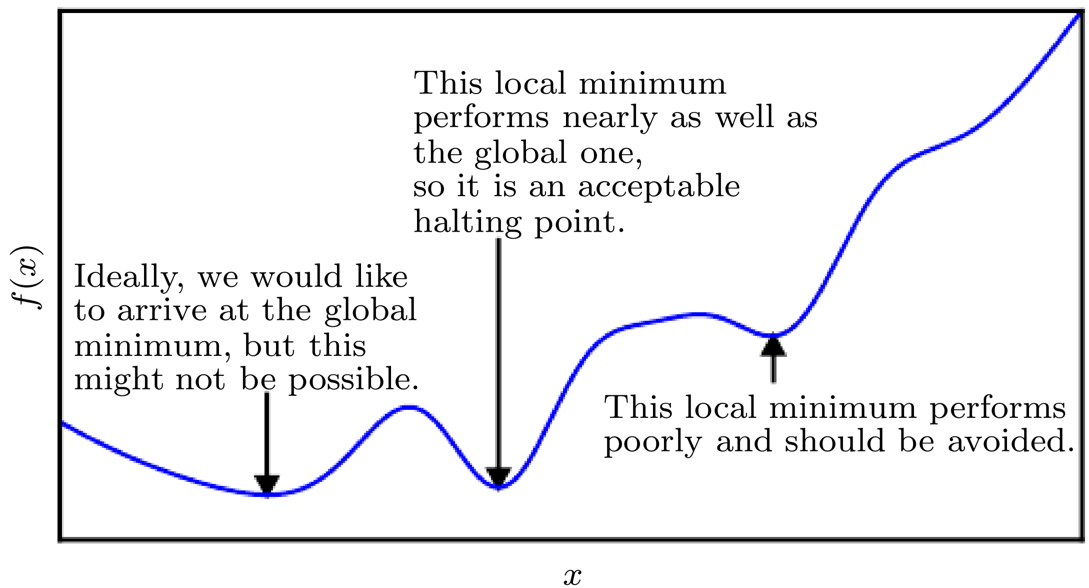
- $E_S(W) = \sum_i (F(W, I_i) - O_i)^2 = \sum_i E_i(W)$
- $W(t+1) = W(t) - \eta(t)\frac{\partial E}{\partial W}(t) = W(t) - \sum_i \eta(t)\frac{\partial E_i}{\partial W}(t)$
- Global update (epoch): sum of per sample updates
- Classical GD: update W globally after all I samples have been processed ($1 \leq i \leq I$)
- Stochastic GD: update W after each processed sample $\rightarrow$ immediate effect, faster convergence
- Batch: update W after a given number (typically between 32 and 256) of processed samples $\rightarrow$ parallelism

Learning rate evolution

- $W(t+1) = W(t) - \eta(t)\frac{\partial E}{\partial W}\big(W(t)\big)$
- Large learning rate: instability
- Small learning rate: slow convergence
- Variable learning rate: learning rate decay policy
- Most often: step strategy: iterate “constant during a number of epochs, then divide by a given factor”
- Possibly different learning rates for different layers or for different types of parameters, generally with common evolution

Gradient descent in practice

- Cost functions are not convex
- Sometimes not differentiable (ReLU)
  - Only at a small number of points
  - Works well in practice



Source: Goodfellow et al, MIT Press 2016

Architecture design

- Universal approximation theorem
  - a feed-forward network with a single hidden layer containing a finite but sufficient number of neurons can approximate (arbitrarily well) any continuous functions on compact subsets of R^n, under mild assumptions on the activation function (e.g., sigmoid).
- But…
  - Optimization algorithm might fail + overfitting
- Empirically, deeper networks generalize better

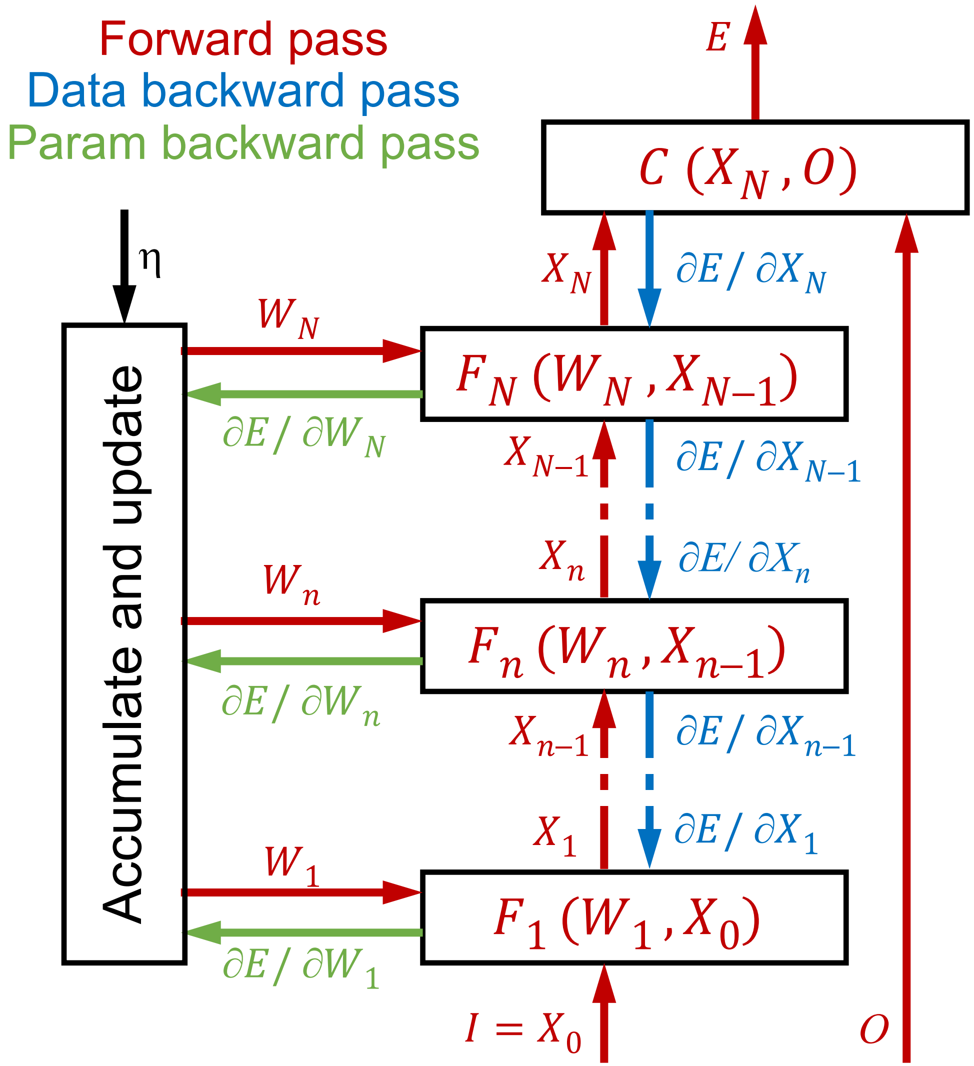
➔ Ideal network architecture via experimentation guided by monitoring the validation error

BACKPROPAGATION

Error back-propagation

- Minimization of $E_S(W)$ by gradient descent:
  - The gradient indicate an ascending direction: move in the opposite
  - Randomly initialize $W(0)$
  - Iterate $W(t+1) = W(t) - \eta \frac{\partial E}{\partial W}\big(W(t)\big)$ $\quad \eta = f(t)$ or $\left(\frac{\partial^2 E}{\partial W^2}\big(W(t)\big)\right)^{-1}$
  - $\frac{\partial E}{\partial W} = \left(\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial W_2} \ \ldots \ \frac{\partial E}{\partial W_N}\right)$ $\qquad (W = (W_1, W_2 \ \ldots \ W_N))$
  - Back-propagation: $\frac{\partial E}{\partial W_n}$ is computed by backward recurrence from $\frac{\partial F_n}{\partial W_n}$ and $\frac{\partial F_n}{\partial X_{n-1}}$ applying iteratively $(g \circ f)' = (g' \circ f).f'$
  - Two derivatives, relative to weight and to data to be considered

Error back-propagation (adapted from Yann LeCun)



Forward pass, for $1 \leq n \leq N$:

$X_n = F_n\,(W_n, X_{n-1})$

$E = C\,(X_N, O)$

We need gradients with respect to X_n. For $n = N$:

$$\frac{\partial E}{\partial X_N} = \frac{\partial C(X_N, O)}{\partial X_N}$$

Then backward recurrence:

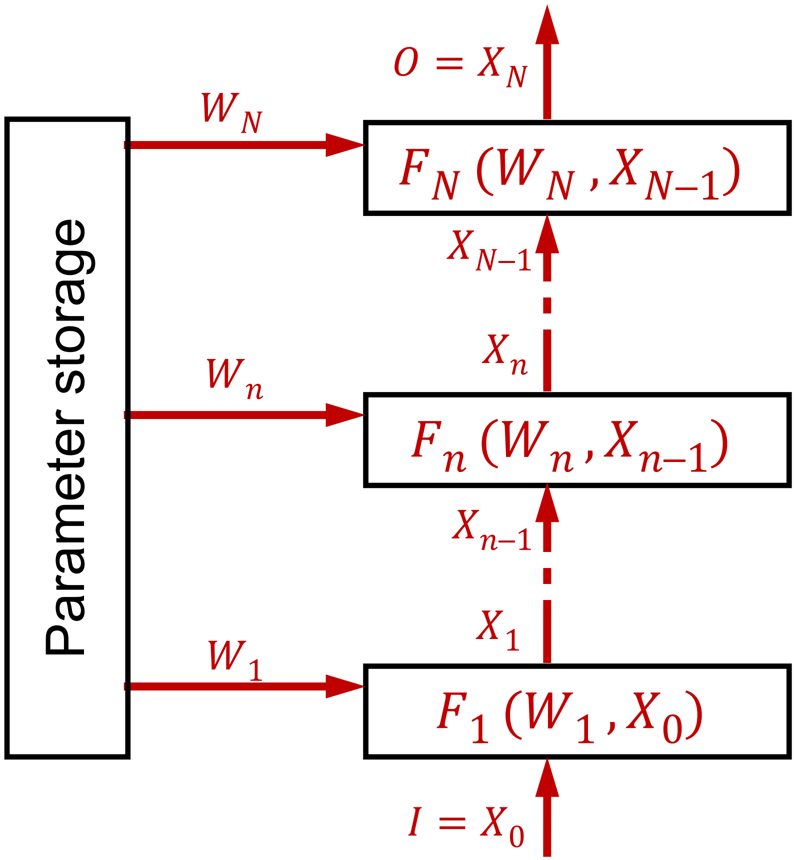
$$\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(W_n, X_{n-1})}{\partial X_{n-1}}$$

Gradients with respect to W_n. For $1 \leq n \leq N$:

$$\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(W_n, X_{n-1})}{\partial W_n}$$

Error back-propagation 0: Prediction mode

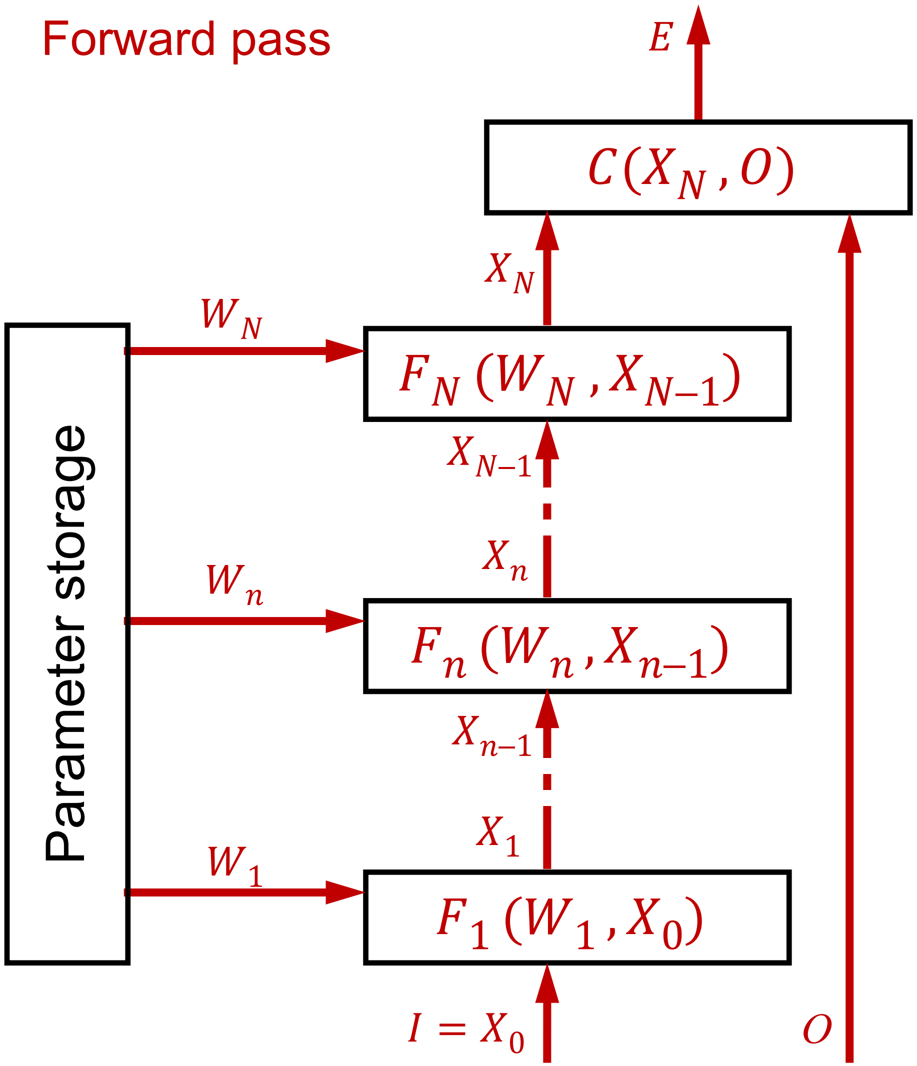
Forward pass



Forward pass, for $1 \leq n \leq N$:

$X_n = F_n(W_n, X_{n-1})$

Error back-propagation 1: loss function



Forward pass, for $1 \leq n \leq N$:
$X_n = F_n(W_n, X_{n-1})$

Loss function (for one sample):
$E = C(X_N, O)$
$E(W, I, O) = C(F(W, I), O)$

Sum over the whole training set or over a batch of samples:

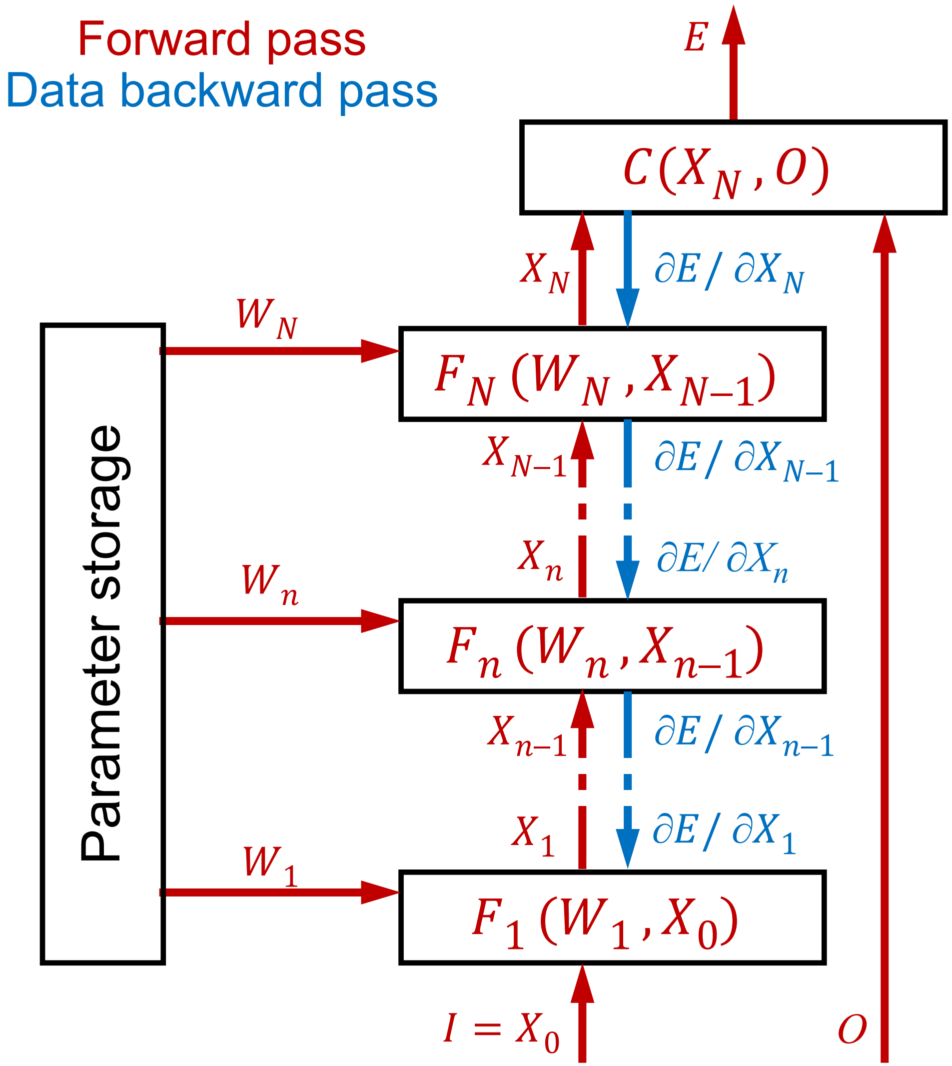
$$E(W) = \sum_i E(W, I_i, O_i)$$

Same W, different (I_i, O_i)

Update:

$$W = W - \eta \frac{\partial E(W)}{\partial W}$$

Error back-propagation 2: Data backward pass



Forward pass, for $1 \leq n \leq N$:

$X_n = F_n(W_n, X_{n-1})$

$E = C(X_N, O)$

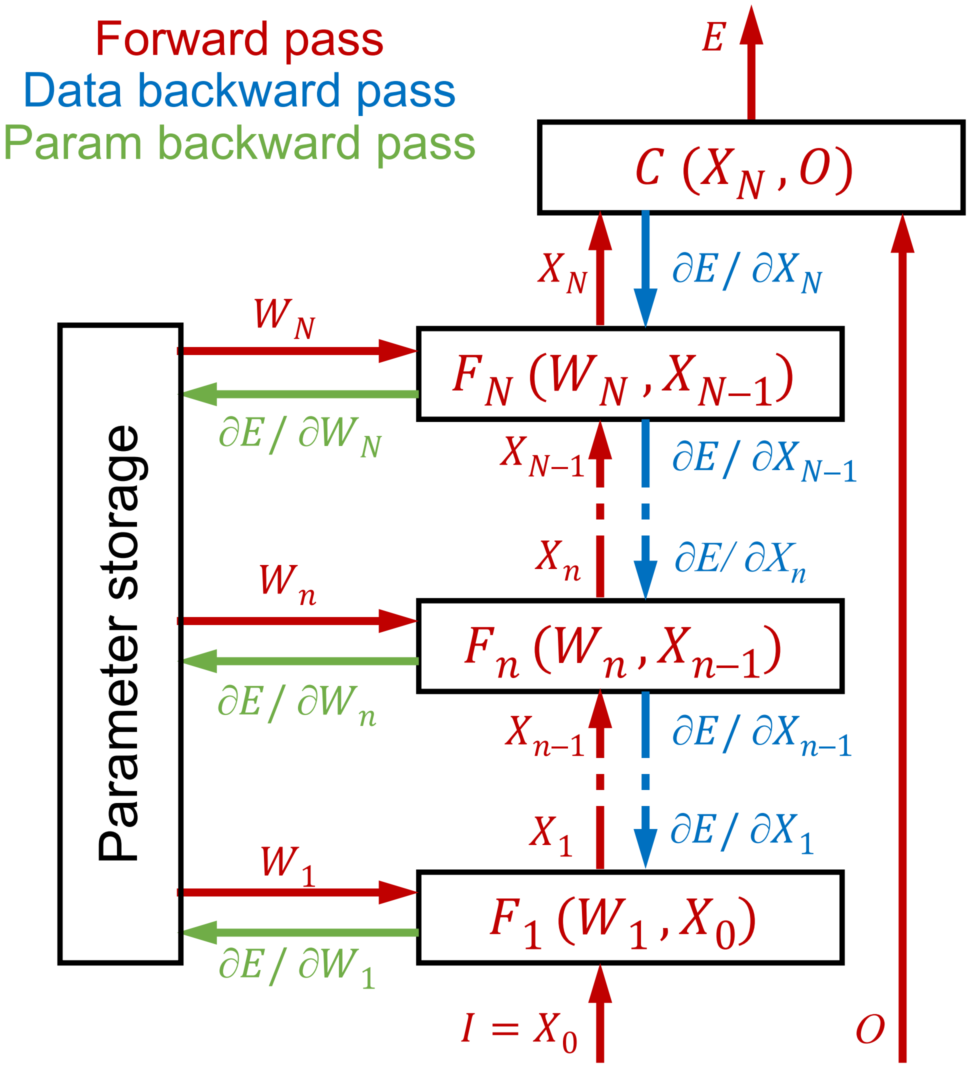
We need gradients with respect to X_n. For $n = N$:

$$\frac{\partial E}{\partial X_N} = \frac{\partial C(X_N, O)}{\partial X_N}$$

Then backward recurrence:

$$\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(W_n, X_{n-1})}{\partial X_{n-1}}$$

Error back-propagation 3: Parameter backward pass



Forward pass, for $1 \leq n \leq N$:

$X_n = F_n\,(W_n, X_{n-1})$

$E = C\,(X_N, O)$

We need gradients with respect to X_n. For N:

$$\frac{\partial E}{\partial X_N} = \frac{\partial C(X_N, O)}{\partial X_N}$$

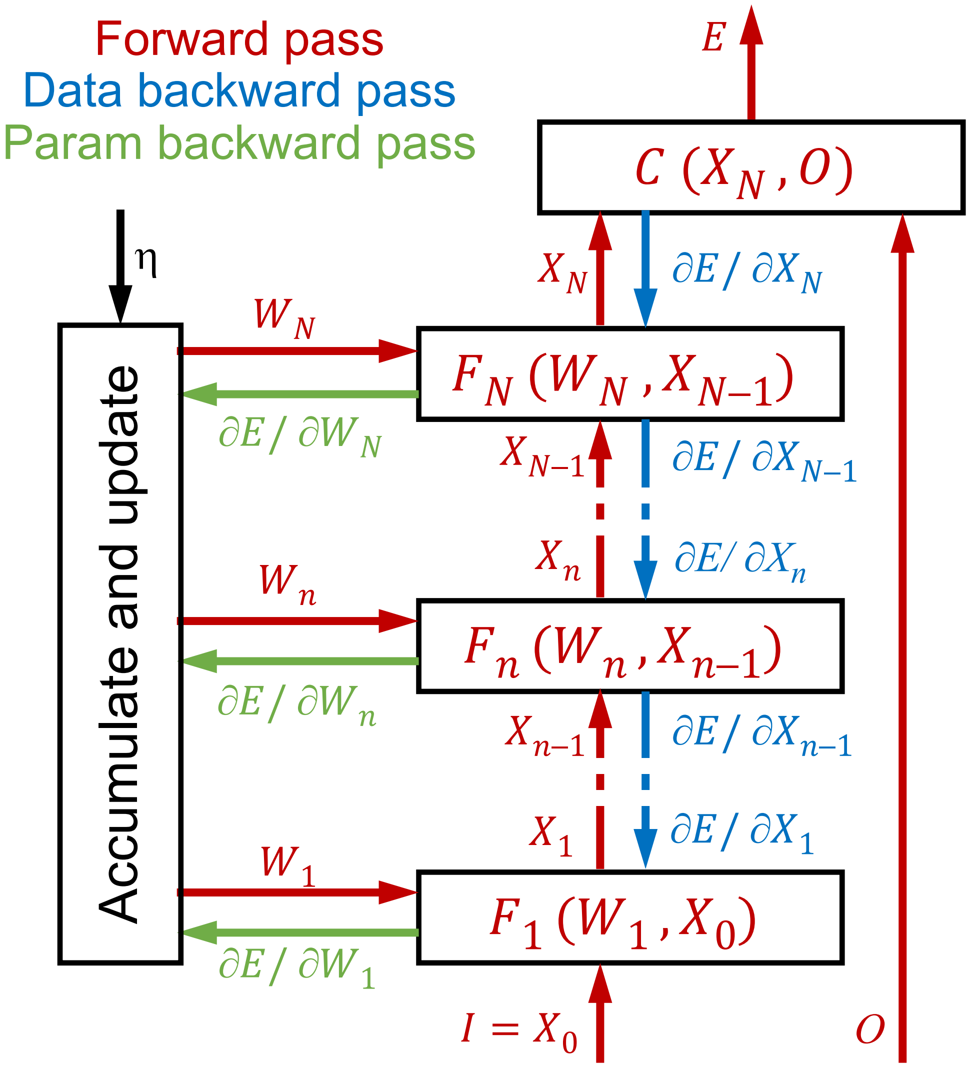
Then backward recurrence:

$$\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(W_n, X_{n-1})}{\partial X_{n-1}}$$

Gradients with respect to W_n. For $1 \leq n \leq N$:

$$\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(W_n, X_{n-1})}{\partial W_n}$$

Error back-propagation 4: Accumulate and update



Forward pass, for $1 \leq n \leq N$:

$X_n = F_n\ (W_n, X_{n-1})$

$E = C\ (X_N, O)$

…

Gradients with respect to W_n.
For $1 \leq n \leq N$:

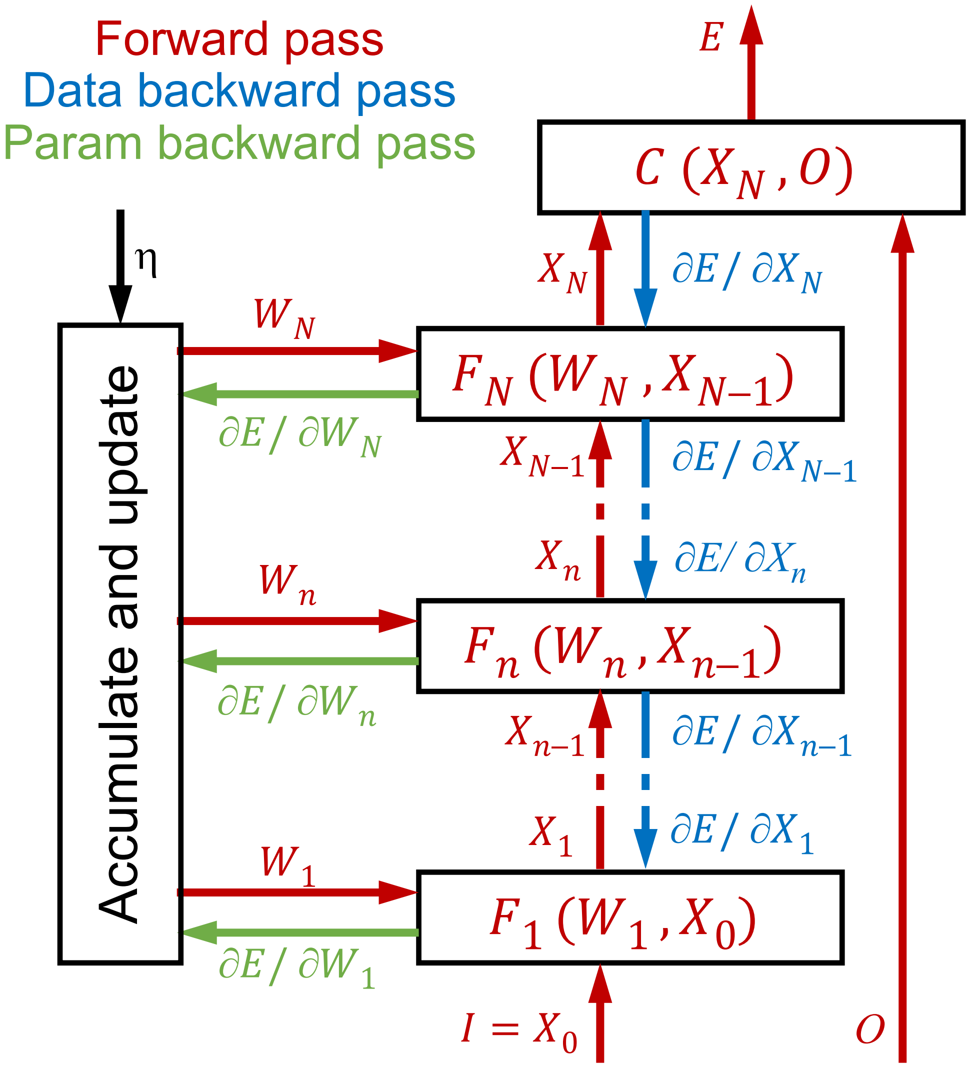
$$\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(W_n, X_{n-1})}{\partial W_n}$$

Accumulate gradients and update parameters.
For $1 \leq n \leq N$:

$$W_n = W_n - \eta \sum_i \frac{\partial E}{\partial W_n}(W, I_i, O_i)$$

Usually on batches

Error back-propagation: simplified notations



Forward pass, for $1 \leq n \leq N$:

$X_n = F_n\ (W_n, X_{n-1})$

$E = C\ (X_N, O)$

We need gradients with respect to X_n. For $n = N$:

$$\frac{\partial E}{\partial X_N} = \frac{\partial C}{\partial X_N}$$

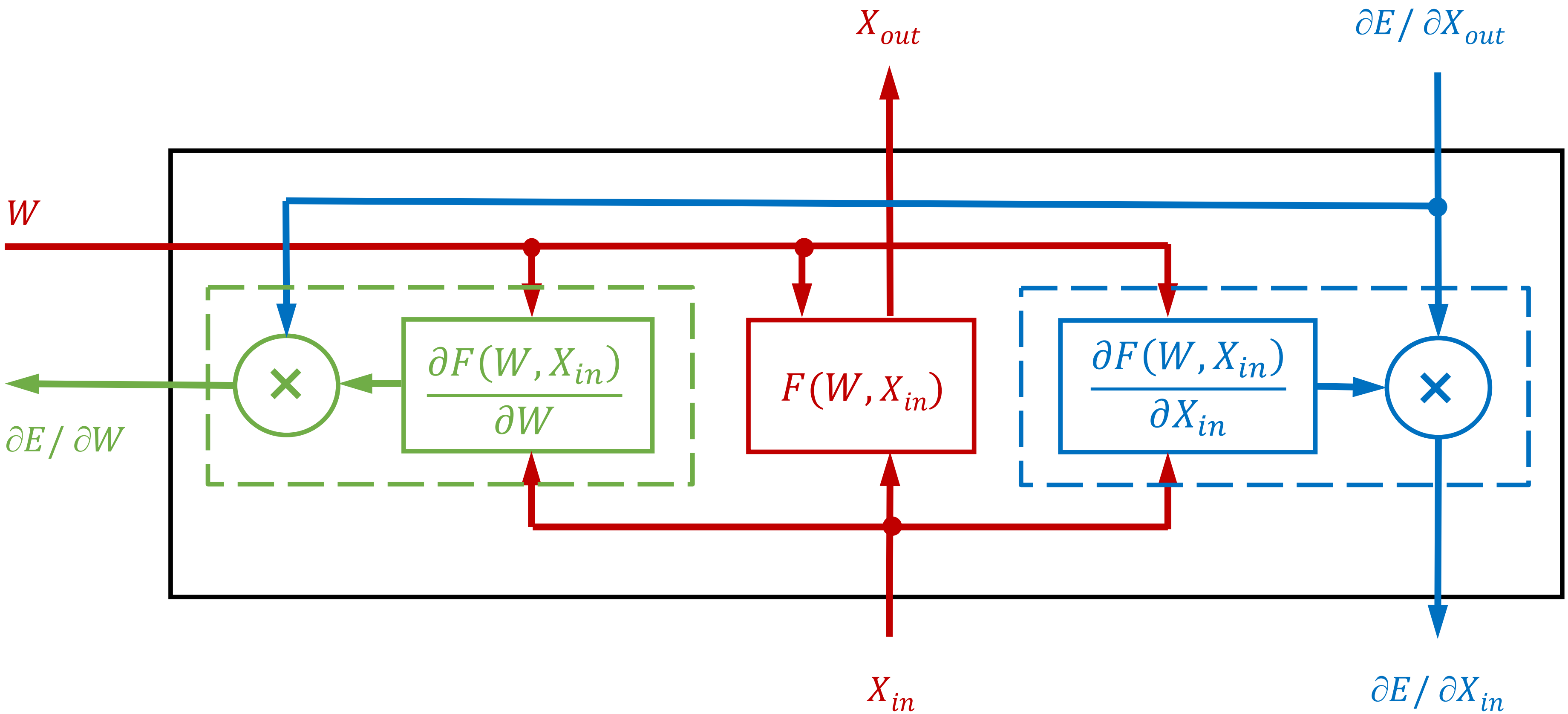
Then backward recurrence:

$$\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial X_n}{\partial X_{n-1}}$$

Gradients with respect to W_n. For $1 \leq n \leq N$:

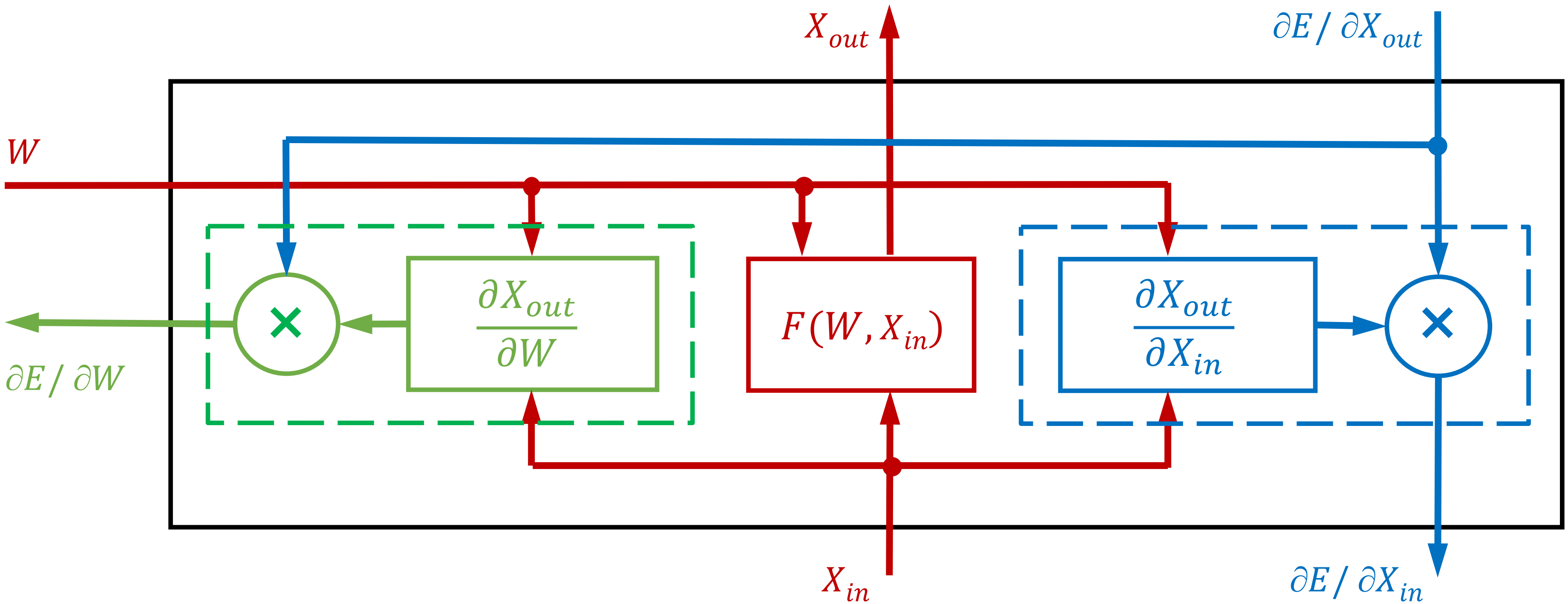
$$\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial X_n}{\partial W_n}$$

Layer module (adapted from Yann LeCun)



Notes: $X_{in} \equiv X_{n-1}$, $X_{out} \equiv X_n$, $W \equiv W_n$ and $F \equiv F_n$ for $1 \leq n \leq N$

Layer module (adapted from Yann LeCun)



$$\frac{\partial F(W, X_{in})}{\partial X_{in}} \equiv \frac{\partial X_{out}}{\partial X_{in}} \qquad \frac{\partial E}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial X_{in}}$$

$$\frac{\partial F(W, X_{in})}{\partial W} \equiv \frac{\partial X_{out}}{\partial W} \qquad \frac{\partial E}{\partial W} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial W}$$

Layer module (adapted from Yann LeCun)

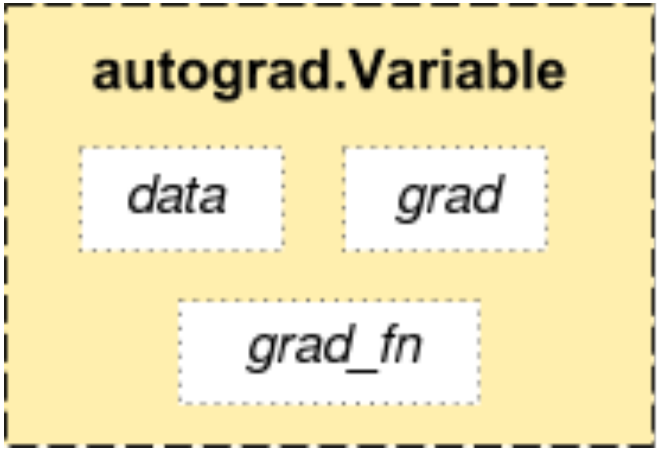
Gradient back-propagation rule:

The gradient relative to the input (either W or X_{in}) is equal to the gradient relative to the output (X_{out}) times the Jacobian of the transfer function (respectively $\frac{\partial X_{out}}{\partial W}$ or $\frac{\partial X_{out}}{\partial X_{in}}$, left vector multiplication)

$$\frac{\partial F(W, X_{in})}{\partial X_{in}} \equiv \frac{\partial X_{out}}{\partial X_{in}} \qquad \frac{\partial E}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial X_{in}}$$

$$\frac{\partial F(W, X_{in})}{\partial W} \equiv \frac{\partial X_{out}}{\partial W} \qquad \frac{\partial E}{\partial W} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial W}$$

Autograd variable (PyTorch)

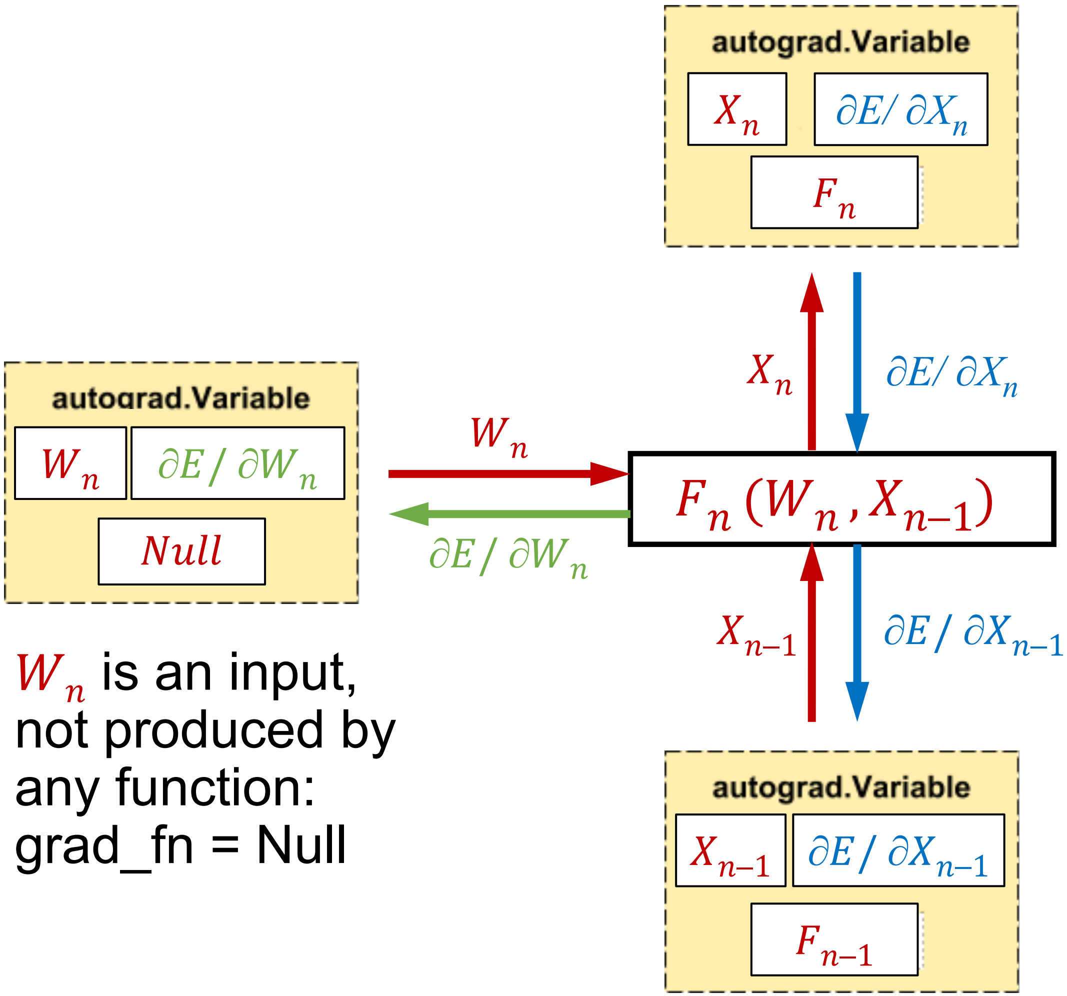


data : X (may be X_{in}, W or X_{out})

grad : $\frac{\partial E}{\partial X}$ E : where backward() was called from

grad_fn : F | $X = F(\dots)$: "None" for W or for inputs

Autograd variable (PyTorch)



$\boxed{F_n}$ contains both the data forward function

$X_n = F(W_n, X_{n-1})$

and the gradient backward function(s)

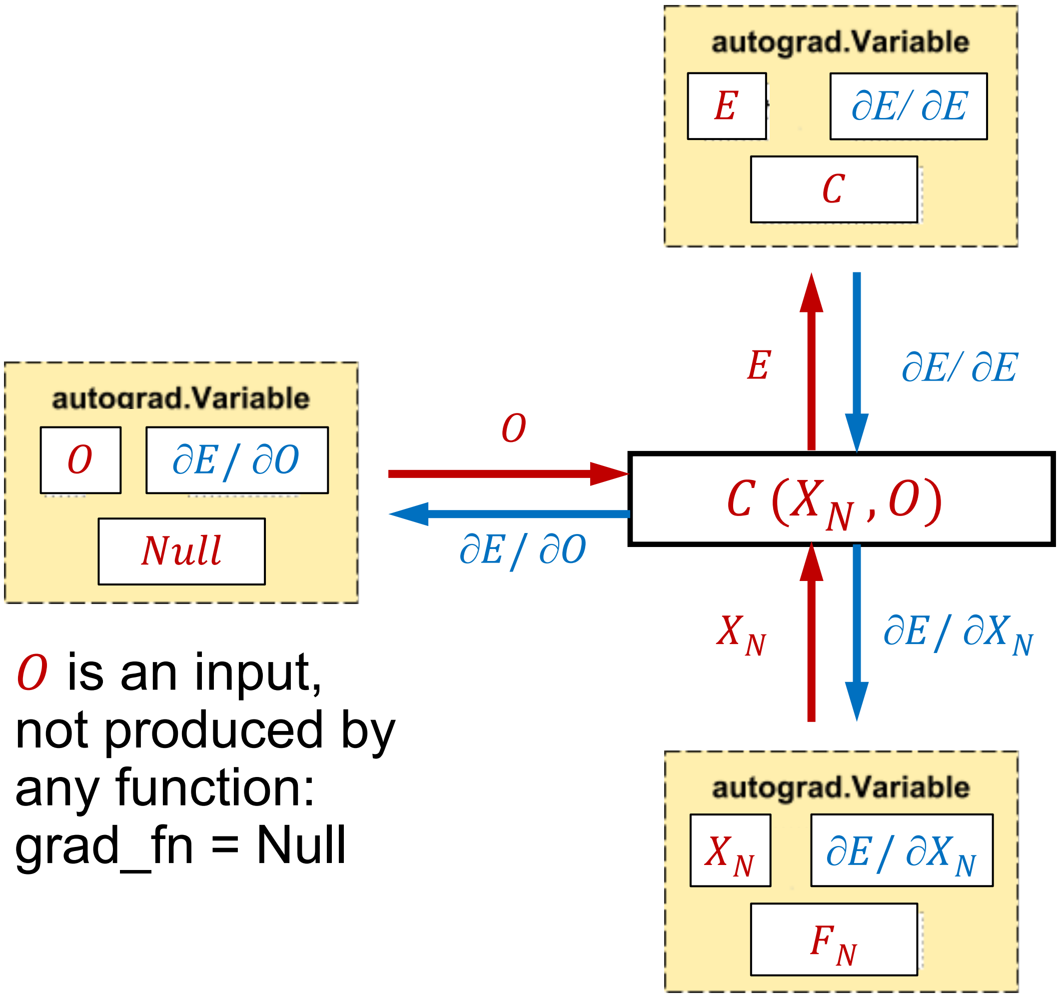
$$\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \cdot \frac{\partial F(W, X_{n-1})}{\partial X_{n-1}}$$

$$\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \cdot \frac{\partial F(W_n, X_{n-1})}{\partial W_n}$$

W_n is an input, not produced by any function: grad_fn = Null

X_0 is an input, not produced by any function: grad_fn = Null for X_0

Autograd variable (PyTorch)



O is an input, not produced by any function: grad_fn = Null

$\boxed{C}$ contains both the data forward function

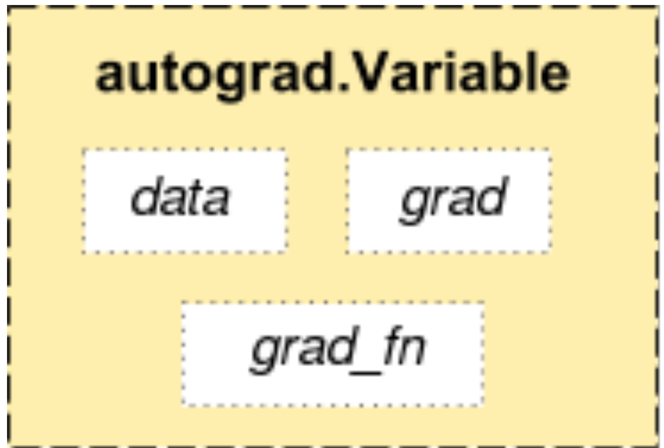
$E = C(X_N, O)$

and the gradient backward function(s)

$$\frac{\partial E}{\partial X_N} = \frac{\partial E}{\partial E} \cdot \frac{\partial C(X_N, O)}{\partial X_N}$$

$$\frac{\partial E}{\partial O} = \frac{\partial E}{\partial E} \cdot \frac{\partial C(X_N, O)}{\partial O}$$

Autograd backward()



Define $X_n = F_n(W_n, X_{n-1})$ for $1 \leq n \leq N$ (or arbitrary network)

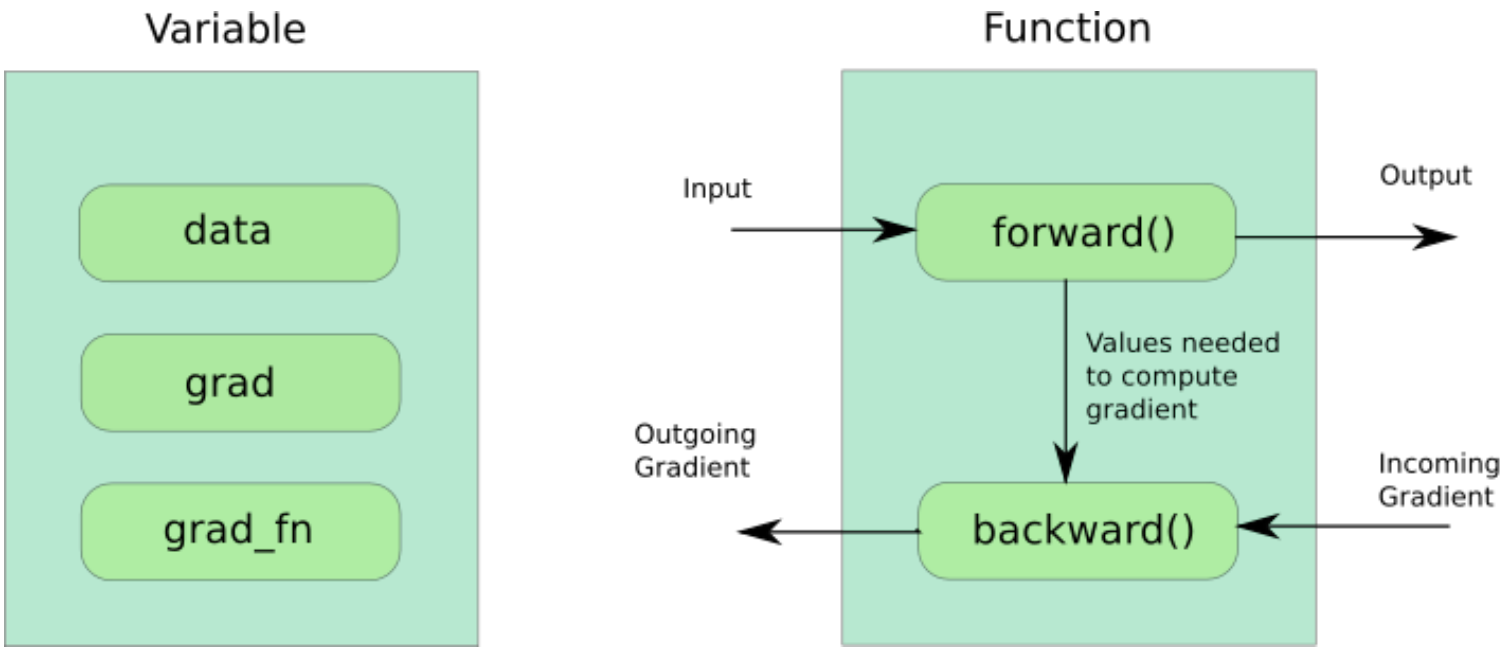
End with $E = C\,(X_N, O)$

Execute a forward pass for a training sample (I, O)

Call E.backward() (backward pass from E with $\partial E / \partial E$=1)

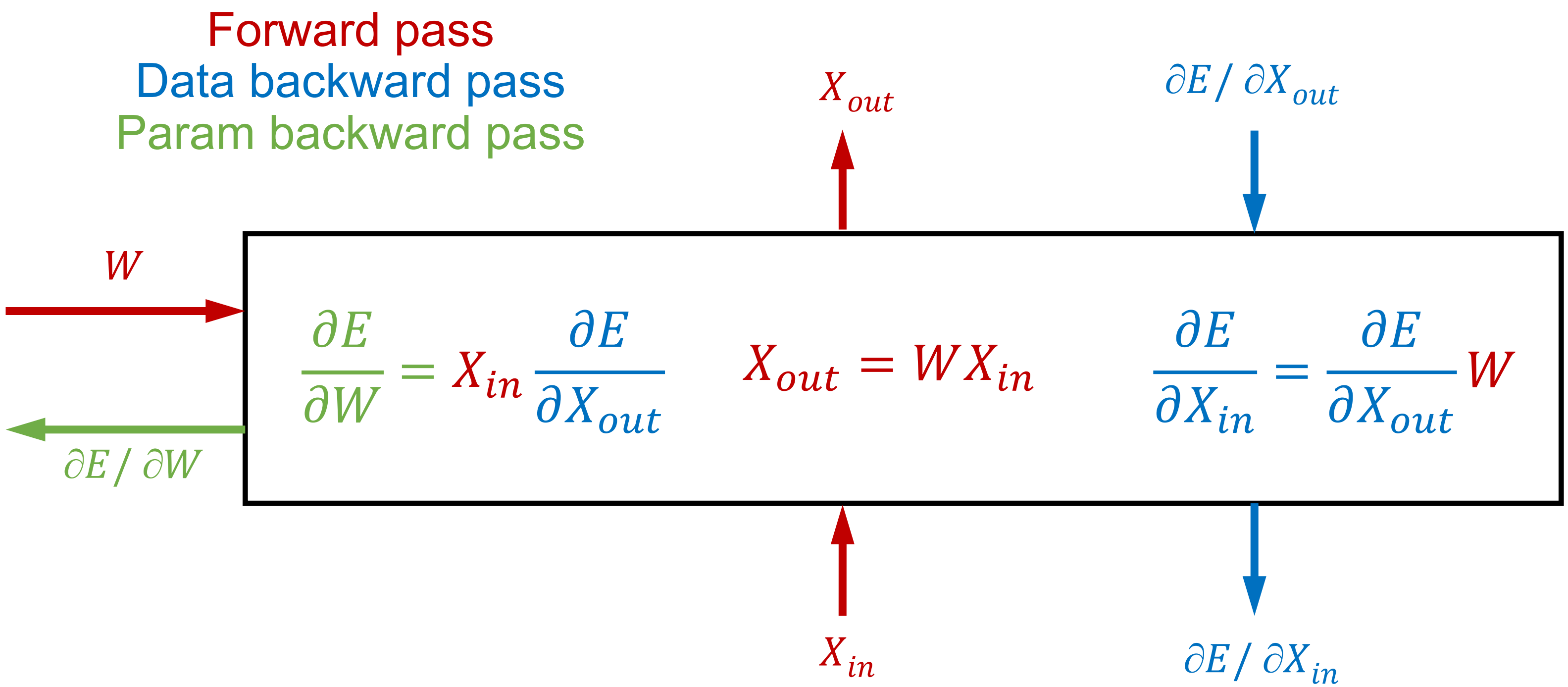
Get all $\partial E / \partial W_n$ (and $\partial E / \partial X_n$) for that training sample

Autograd Variable and function



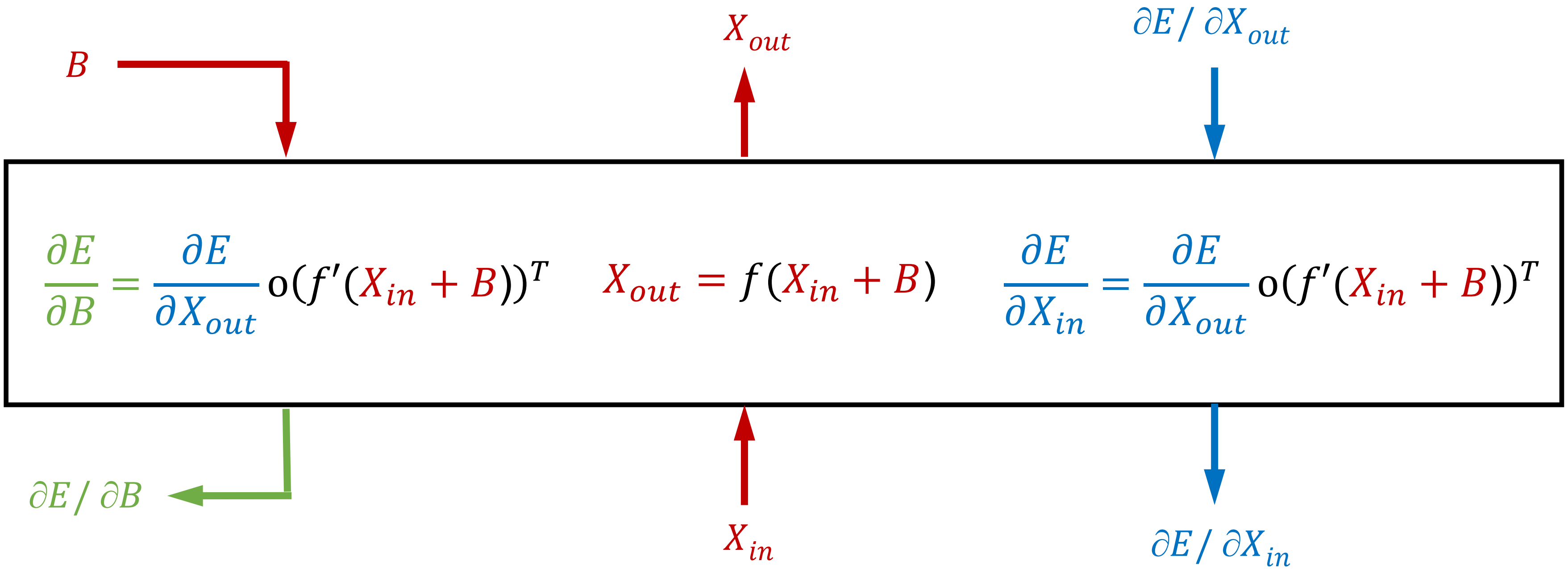
Input may be multiple (Xi_n, W)
Autograd does not care about input types

Linear module (adapted from Yann LeCun)



Note: X_{in} and X_{out} are regular (column) vectors and W is a matrix while $\partial E / \partial X_{in}$ and $\partial E / \partial X_{out}$ are transpose (row) vectors (this is because $\mathrm{d}E = (\partial E / \partial X).\mathrm{d}X$). $\partial E / \partial W$ is a transposed matrix which is the *outer* product of the regular and transpose vectors X_{in} and $\partial E / \partial X_{out}$.

Pointwise module (adapted from Yann LeCun)



$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial X_{out}} \mathrm{o} (f'(X_{in} + B))^T \qquad X_{out} = f(X_{in} + B) \qquad \frac{\partial E}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} \mathrm{o} (f'(X_{in} + B))^T$$

Notes: B is a bias vector on the input. X_{in}, X_{out} and B are regular (column) vectors all of the same size while $\partial E / \partial X_{in}$ and $\partial E / \partial X_{out}$ and $\partial E / \partial B$ are transpose vectors also of the same size. f is a scalar function applied pointwise on $X_{in} + B$. f' is the derivative of f and is also applied pointwise. The multiplication by $(f'(X_{in} + B))^T$ is also performed pointwise (Hadamard product denoted "o" here).

Neural Networks training in practice

- Good news is that `autograd` automatically and transparently takes care of gradients computation and propagation; you just have to call `.backward()`
- You only have to define the forward network sequence
- You still have to select various hyper-parameters and to organize:
  - iterations
  - batch processing
  - learning rate schedule
  - possibly data augmentation

Dropout

- Regularization technique
- During training, at each epoch, neutralize a given (typically 0.2 to 0.5) proportion of randomly selected connections
- During prediction, keep all of them with a multiplicative compensating factor
- Avoid concentration of the activation on particular connections
- Much more robust operation
- Faster training, better performance

Softmax

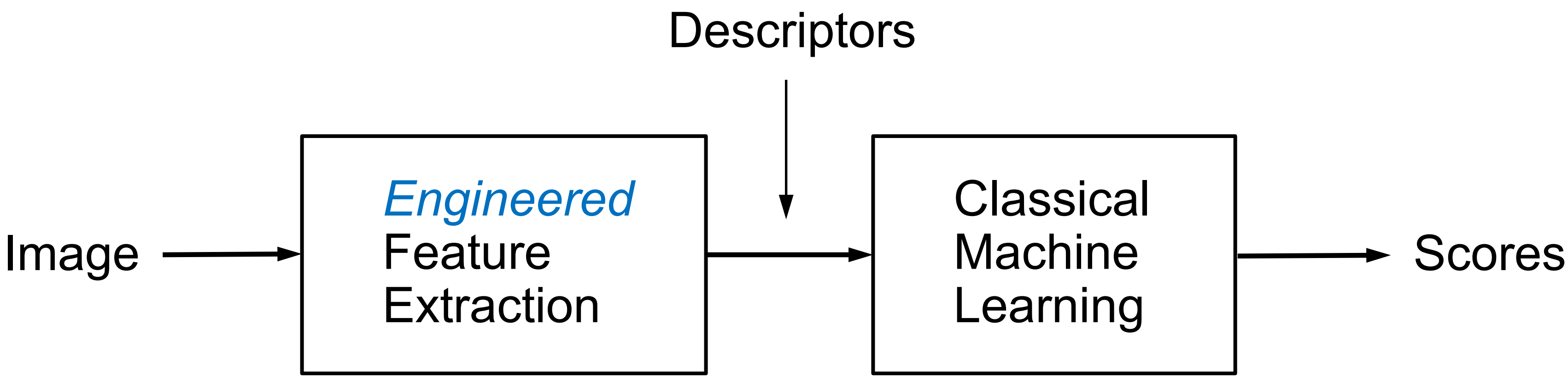
- Normalization of output as probabilities (positive values summing to 1) for the multi-class problem (i.e. target categories are mutually exclusive)
- $z_i = \frac{e^{y_i}}{\sum_j e^{y_j}}$
- Not suited for the multi-label case (i.e., target categories are not mutually exclusive)
- Associated loss function is cross-entropy

Cross-entropy loss (multi-class)

- p_i : probability vector for class i
- l_i : truth value for class i ("one hot encoding")
- $L = \sum_i -(l_i \log p_i)$
- For exclusive classes, l_i is equal to 1 only for the right class i_0 and to 0 otherwise:
- $L = -\log p_{i_0}$ $\quad$ ($\log 1 = 0$ and $\log 0 = -\infty$)
- Forces p_{i_0} to be close to 1, very high loss value if p_{i_0} is close to 0 $\rightarrow$ faster convergence
- Other p_i indirectly forced to be close to 0 because the p_is sums to 1
- With softmax: forces y_{i_0} to be greater than the other y_is

CONVOLUTIONAL NEURAL NETWORKS (CNN)

Classical Image classification

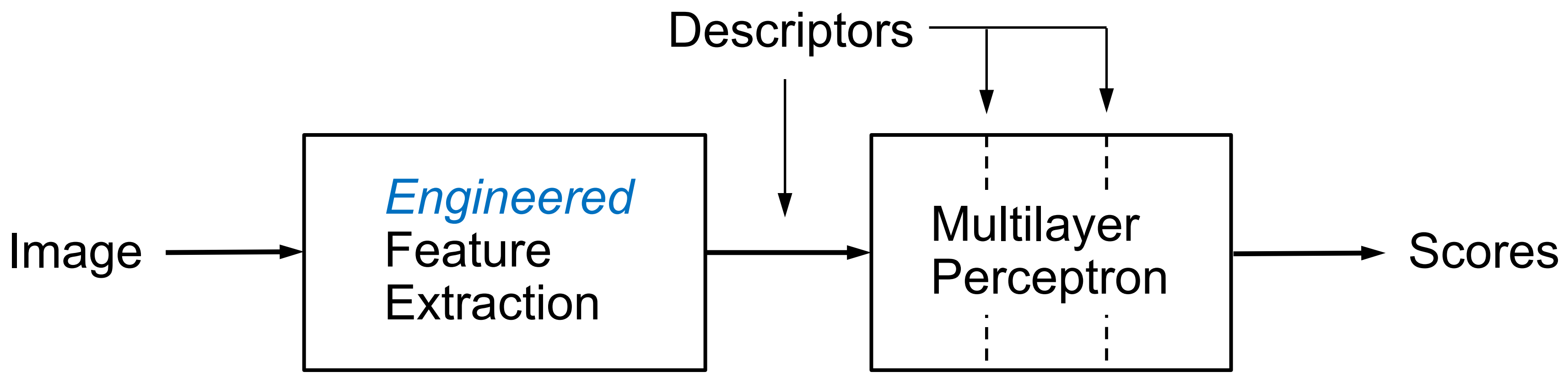


Color Histograms
Gabor Transforms
Bags of SIFTs
Fisher Vectors
…

Support Vector Machines
Multilayer Perceptrons
Random Forests
…

Plus: multiple features, early or late fusion, re-scoring …

Classical Image classification

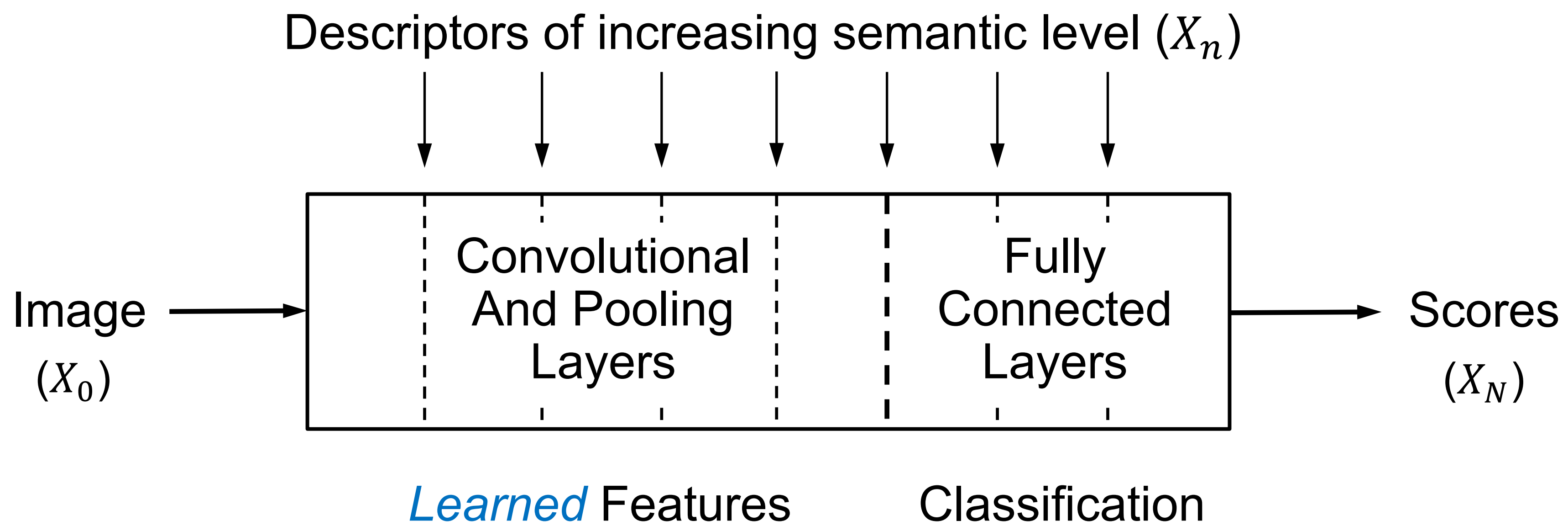


Color Histograms
Gabor Transforms
Bags of SIFTs
Fisher Vectors
…

Typically 3 layers
Not really better
than SVMs or
Random Forests

Still classical since 3-layer MLPs are at least 30 years old

Deep "end-to-end" Image classification

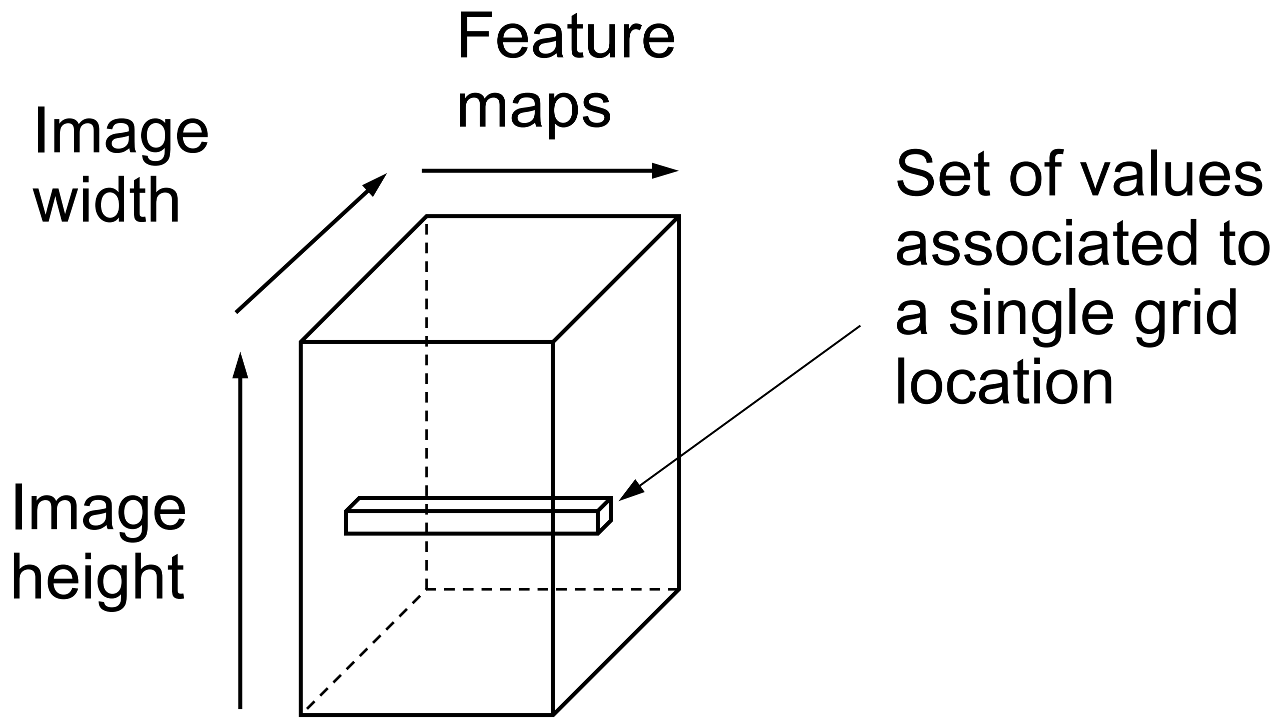


- Fuzzy boundary between feature extraction and classification even if there is a transition between convolutional and fully connected layers
- *End-to-end learning*: features (descriptors) themselves are learned (by gradient descent) too, not engineered
- Possible only via the use of *convolutional* layers

Convolutional layers (2D grid case)

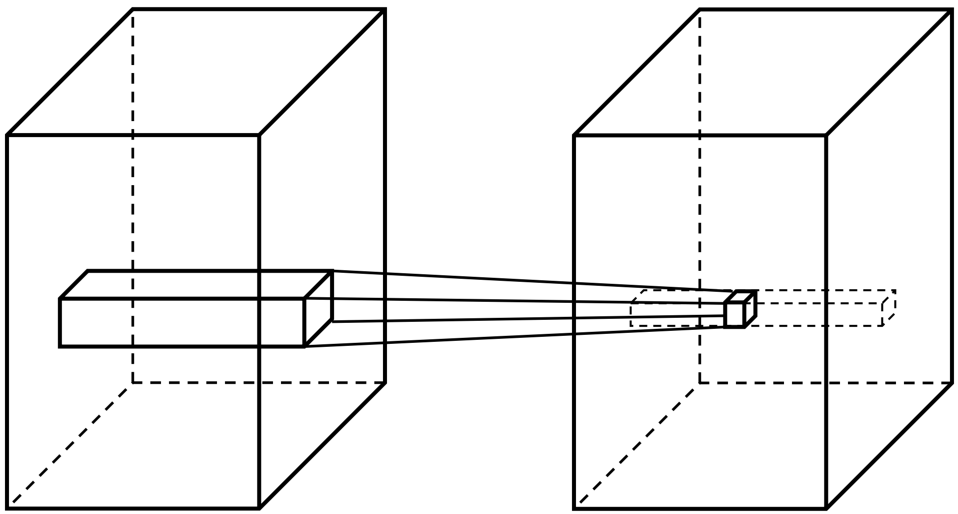
- Alternative to the “all to all”(vector to vector) connections
- Preserves the 2D image topology via “feature maps”
- X_n are 3D data (“tensors”) instead of vectors
- 2 of the dimensions are aligned with the image grid
- The third dimension is a set of values associated to a grid location (gathered in a vector per location but without associated topology)
- Each component in the third dimension correspond to a “map” aligned with the image grid
- Each data tensor is a “stack” of features maps
- Translation-invariant (relatively to the grid) processing

3D tensor data (2D grid case)



Input image data is a special case with 3 feature maps corresponding to the RGB planes and sometimes 4 or even more for RGB-D or for hyper-spectral (satellite) image data.

Convolutional layers (2D grid case)



- Each map point is connected to all maps points of a fixed size neighborhood in the previous layer
- Weights between maps are shared so that they are invariant by translation in the image plane

Convolutional layers (2D grid case)

- Combination of:
  - convolutions within the image plane
  - “all to all” within the map dimension
- Separable or non-separable combinations
- Resolution changes across layers: stride and pooling
- Examples: LeNet (1998) and AlexNet (2012)

Classical image convolution (2D to 2D)

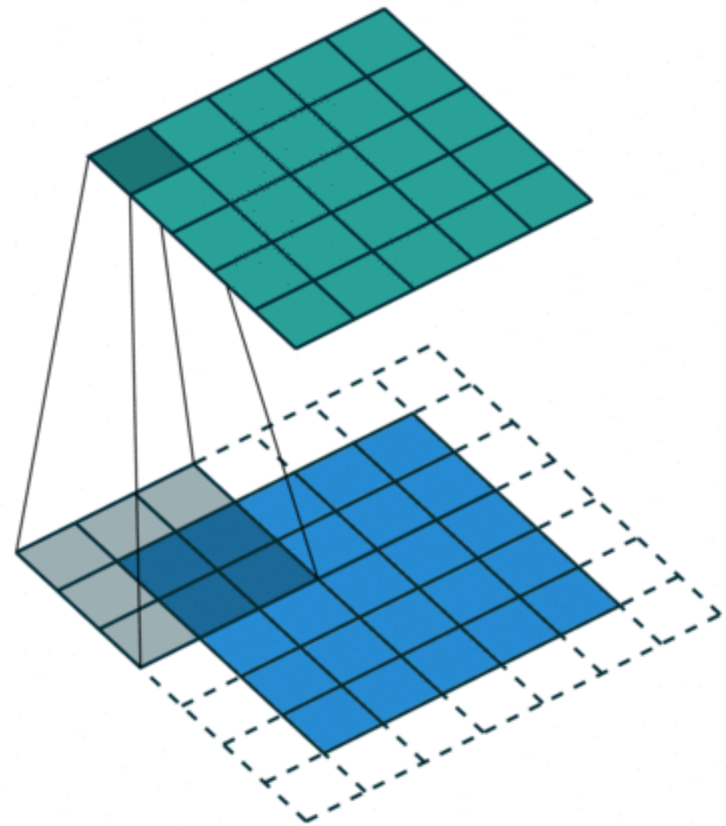
- Classical image convolution (2D to 2D):

$$O(i,j) = (K * I)(i,j) = \sum_{(m,n)} K(m,n) I(i-m, j-n)$$

- Convolutional layer (3D to 3D):
- m and n : within a window around the current location, corresponding to the filter size
- $K(m,n)$: convolution kernel
- Example: (circular) Gabor filter:

$$K(m,n) = \frac{1}{2\pi\sigma^2} . e^{-\frac{m^2+n^2}{2\sigma^2}} . e^{2\pi i \frac{m.\cos\theta + n.\sin\theta}{\lambda}}$$

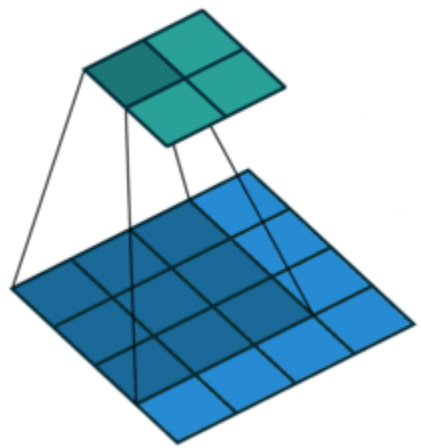
Classical image convolution (2D to 2D)



3x3 convolution, half padding

Animation from https://github.com/vdumoulin/conv_arithmetic/

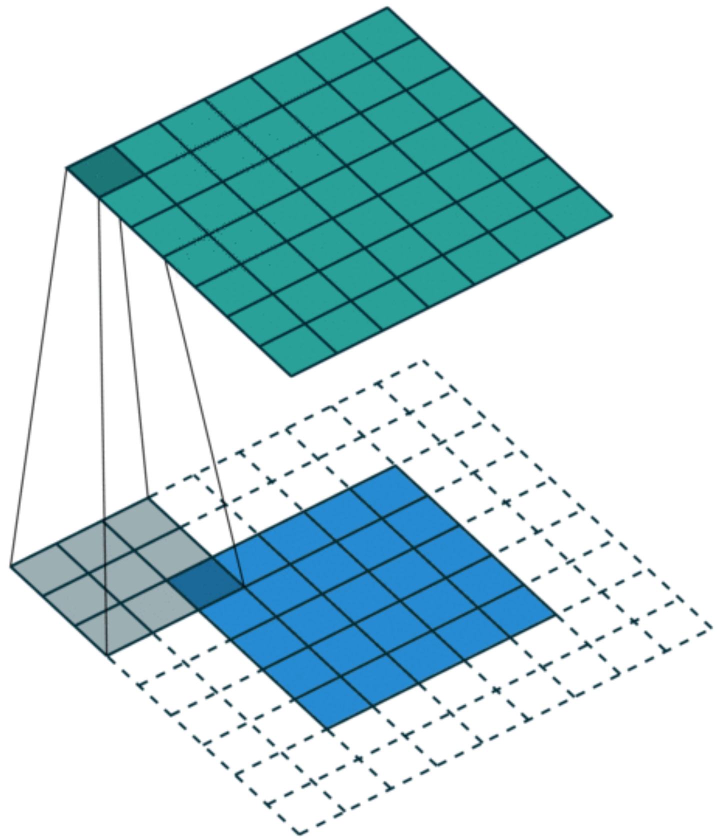
Classical image convolution (2D to 2D)



3×3 convolution, no padding

Animation from https://github.com/vdumoulin/conv_arithmetic/

Classical image convolution (2D to 2D)



3×3 convolution, full padding

Animation from https://github.com/vdumoulin/conv_arithmetic/

Convolutional layers

- Convolutional layer: multiple maps (planes) both in input and output (3D to 3D, plus bias):

$$O(l,i,j) = B(l) + \sum_{(k,m,n)} K(k,l,m,n) I(k,i-m,j-n)$$

- k and l: indices of the feature maps in the input and output layers
- m and n: within a window around the current location, corresponding to the feature size

Convolutional layers

- Convolutional layer: multiple maps (planes) both in input and output (3D to 3D, plus bias):

$$O(l,i,j) = B(l) + \sum_{(k,m,n)} K(l,k,m,n) I(k, i-m, j-n)$$

- Operation relative to (m,n) : convolution
- Operation relative to (k,l) : matrix multiplication plus bias (equals affine transform)
- Combination of:
  - Convolution within the image plane, image topology
  - Classical all to all “perpendicularly” to the image plane, no topology
- If image size and filter size = 1: fully connected “all to all”

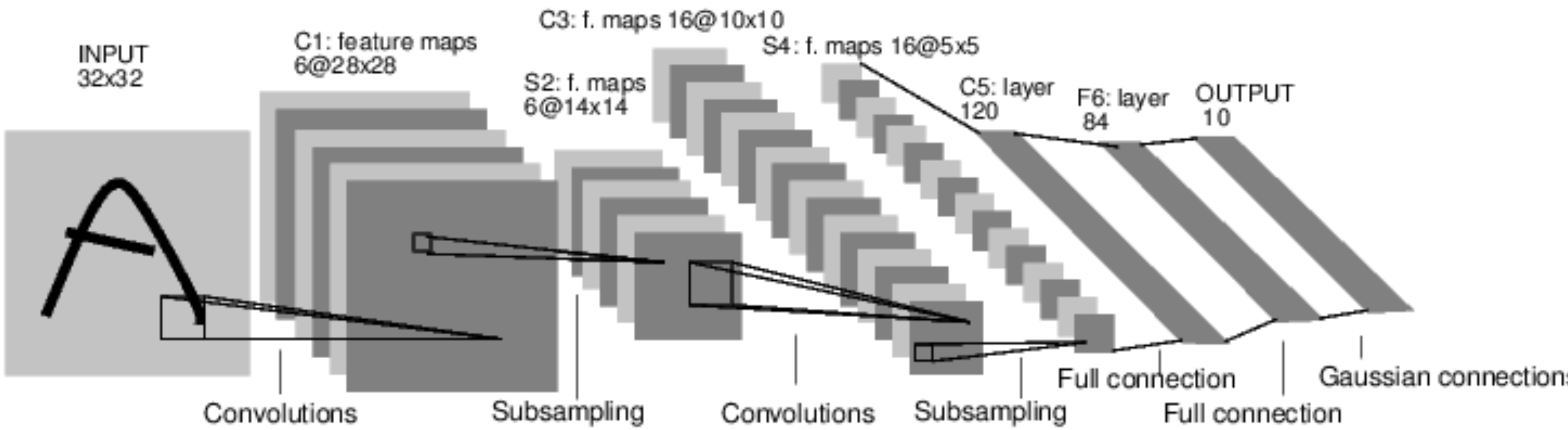
Resolution changes and side effects

- Side (border) effect:
  - crop the output “image” relative to the input one and/or
  - pad the image if the filter expand outside
- Resolution change (generally reduction):
  - Stride: subsample, e.g. compute only one out of N, and/or
  - Pool: compute all and apply an associative operator to compute a single value for the low resolution location from the high resolution ones, e.g.:

    $$O(k,i,j) = \mathrm{op}(I(k,2i,2j), I(k,2i+1,2j), I(k,2i,2j+1), I(k,2i+1,2j+1))$$

- Common pooling operators: maximum or average
- Pooling correspond to a separate back-propagation module (as for the linear and non-linear parts of a layer)

Pytorch tutorial network (LeNet, 1998)



(Grayscale image)

Pytorch tutorial network

```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Pytorch tutorial network (color image)

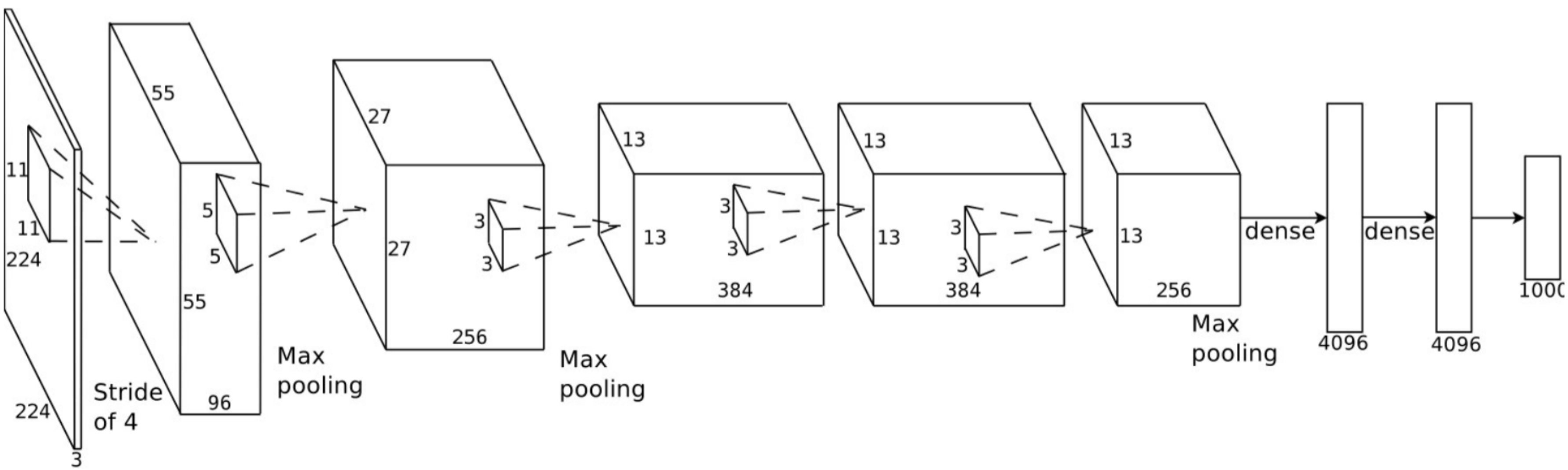
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

AlexNet (ImageNet Challenge 2012)

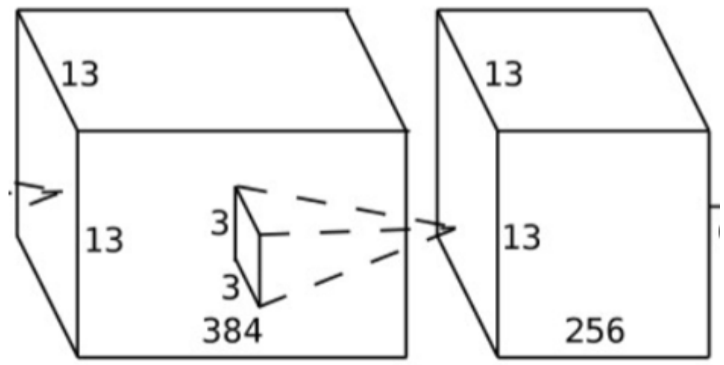
[Krizhevsky et al., 2012]

- 7 hidden layers, 650K units, 60M parameters (W)
- GPU implementation (50× speed-up over CPU)
- Trained on two GTX580-3GB GPUs for a week



A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

AlexNet “conv5” example



- Number of units (“neurons”) in a layer (= size of the output tensor): output image width (13) × output image height (13) × number of output planes (256) = 43,264
- Number of weights in a layer (= number of weights in a layer): number of input planes (384) × number of output planes (256) × filter width (3) × filter height (3) = 884,736 (884,992 including biases)
- Number of connections: number of grid locations × number of weights in a unit set (excluding biases) = 149,520,384

Yann LeCun recommendations

- Use ReLU non-linearities (tanh and logistic are falling out of favor)
- Use cross-entropy loss for classification
- Use Stochastic Gradient Descent on minibatches
- Shuffle the training samples
- Normalize the input variables (zero mean, unit variance)
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs
- Use "dropout" for regularization
  - Hinton et al 2012 http://arxiv.org/abs/1207.0580
- Lots more in [LeCun et al. "Efficient Backprop" 1998]
- Lots, lots more in "Neural Networks, Tricks of the Trade" (2012 edition) edited by G. Montavon, G. B. Orr, and K-R Müller (Springer)

Recent trends and other topics

- VGG and GoogLeNet (16-19 and 22 layers)
- Residual networks (152 layers with “shortcuts”)
- Stochastic depth networks (up to 1202 layers)
- Weakly supervised / unsupervised learning
- GANs / VAEs
- Transfer learning
- Recurrent networks (time series)
- Transformers (NLP)