Machine Learning et Intelligence Artificielle
Introduction au Deep Learning

Sorbonne Université
Patrick Gallinari, patrick.gallinari@sorbonne-universite.fr
Olivier Schwander, olivier.schwander@sorbonne-universite.fr
04/2026

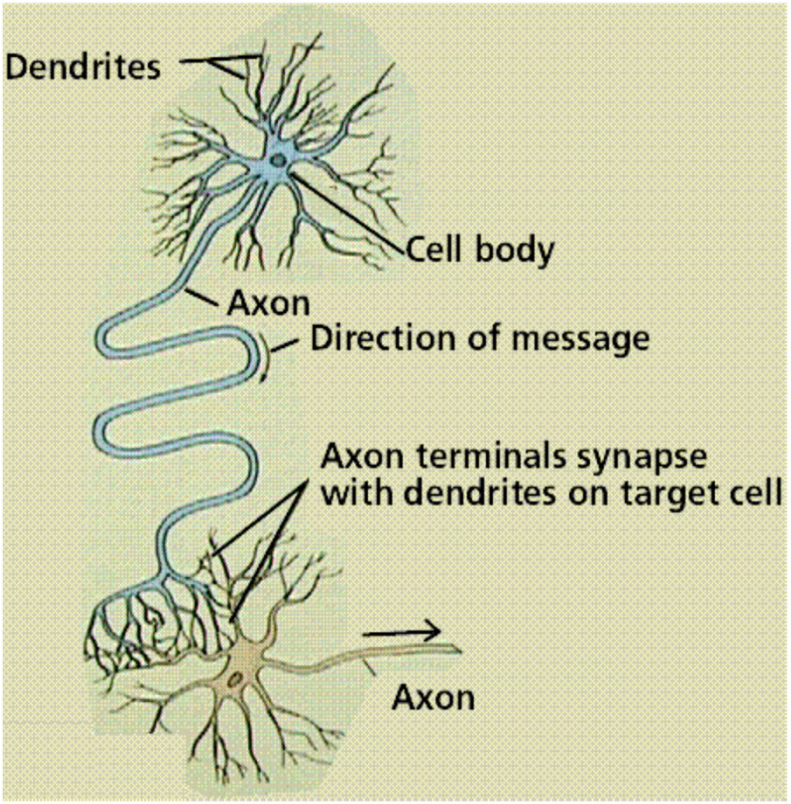
Course Outline and Organization

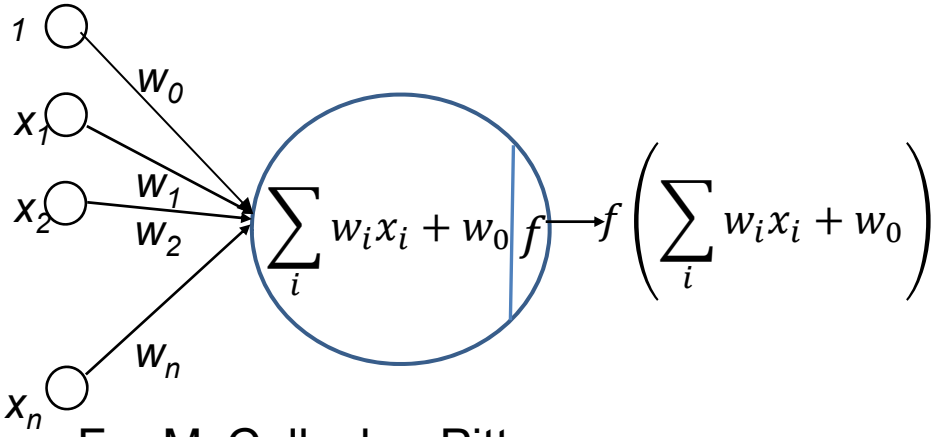
- Introductory course on Neural Networks and Deep Learning
- Organization
  - 2x3 h course, 2x3 h practice
- Outline
  - Neural Networks and Deep Learning
    - Introductory Concepts – Perceptron - Adaline
    - Computation Graph
    - Multilayer Perceptrons
    - Convolutional Neural Networks – Vision applications
    - Language models - Transformers

Introductory NN concepts

Intuitive introduction via the historical example of the Perceptron and Adaline

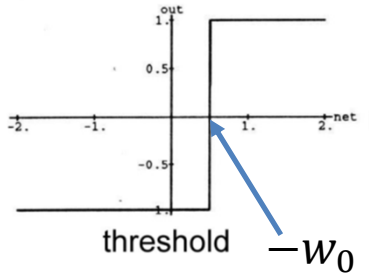
Formal Model of the Neuron
McCulloch – Pitts 1943





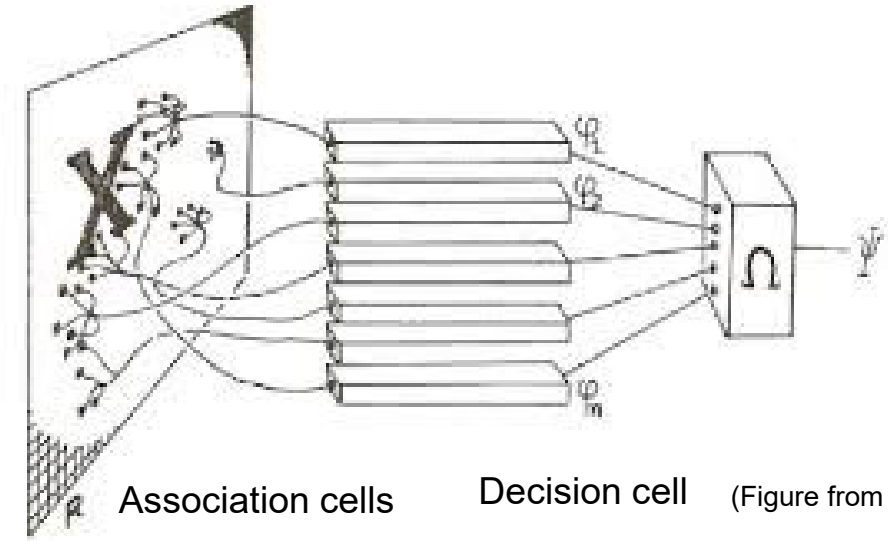
For McCulloch – Pitts neuron

$$F(\boldsymbol{x}) = sgn(\sum_{i=1}^{n} w_i x_i + \mathrm{w}_0)$$

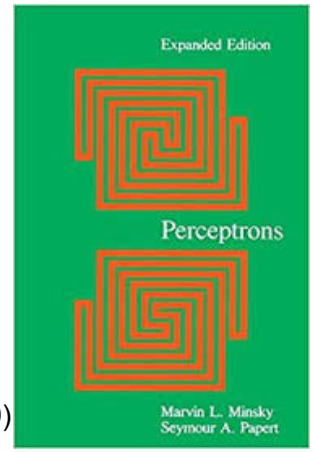


$$f(x) = \begin{cases} 1 \; if \sum_i (w_i x_i + w_0) > 0 \\ -1 \; otherwise \end{cases}$$

Perceptron (1958 Rosenblatt)

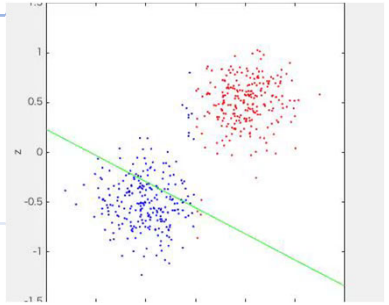


Association cells Decision cell (Figure from Perceptrons, Minsky and Papert 1969)



- The decision cell is a threshold function (McCulloch – Pitts neuron)
  - $F(\boldsymbol{x}) = sgn(\sum_{i=1}^{n} w_i x_i + \mathrm{w}_0)$
- This simple perceptron can perform 2 classes classification

Perceptron Algorithm (1958 – Rosenblatt, 2 classes)



Data

Labeled Dataset $\{(\boldsymbol{x}^i, y^i), i = 1..N, \boldsymbol{x} \in R^n, y \in \{-1,1\}\}$

Output

classifier $\boldsymbol{w} \in R^n$, decision $F(\boldsymbol{x}) = sgn(\sum_{i=0}^{n} w_i x_i)$

Training set
Classifier specificatio

Initialize $\boldsymbol{w}$ (0)

Repeat (t)

Choose an example $(\boldsymbol{x}(t), y(t))$

if $y(t)\boldsymbol{w}(t).\boldsymbol{x}(t) \leq 0$ then $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \epsilon y(t)\boldsymbol{x}(t)$

Stochastic Algorithm

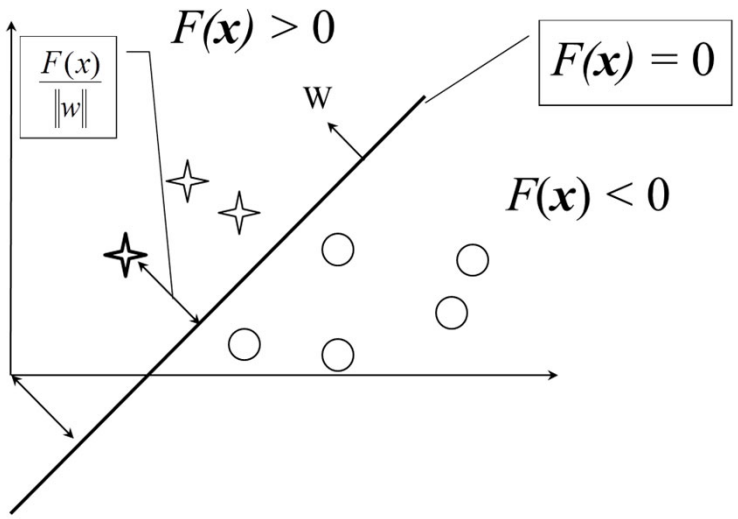
Until convergence

- The learning rule is a **stochastic gradient algorithm** for minimizing the number of wrongly predicted labels
- Multiple (p) classes: p perceptrons in parallel, 1 class versus all others!

Linear discriminant function

$$F(\boldsymbol{x}) = \boldsymbol{w}.\boldsymbol{x} + w_0 = \sum_{i=0}^{n} w_i x_i \text{ with } x_0 = 1$$

- Decision surface : hyperplane $F(\boldsymbol{x}) = 0$
- Properties
  - $\boldsymbol{w}$ is a normal vector to the hyperplane, it defines its orientation
  - distance from x to H : $r = F(\boldsymbol{x})/\|\boldsymbol{w}\|$
  - if $w_0 = 0$ H goes through the origin



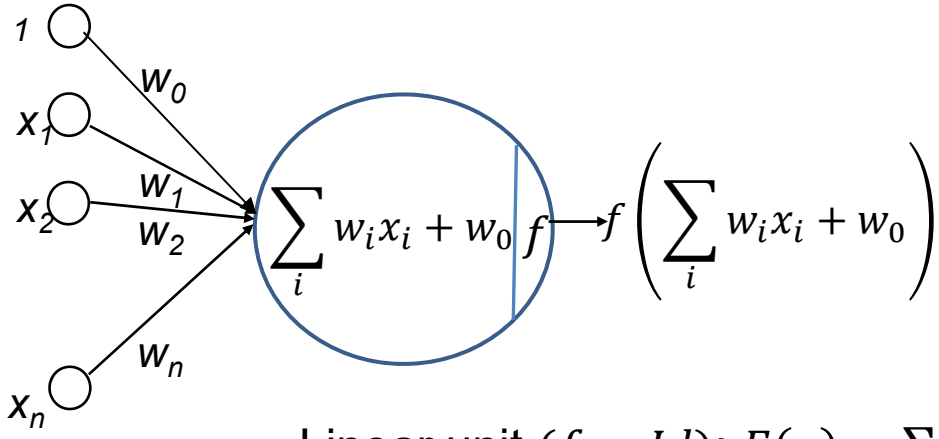
Perceptron algorithm performs a stochastic gradient descent

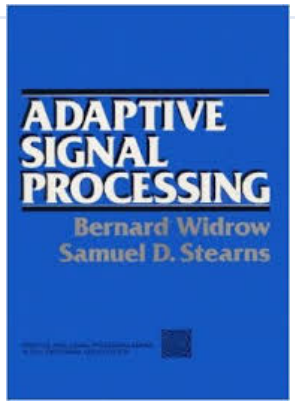
- Loss function
  - $C = -\sum_{(x,y)\text{missclassified}} \boldsymbol{w}.\boldsymbol{x}y = -\sum_{(x,y)\text{miss-classified}} \mathrm{c}(\boldsymbol{x}, \mathrm{y})$
  - Objective : minimize C
- gradient
  - $grad_{\boldsymbol{w}} C = \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n}\right)^T$ with $\frac{\partial C}{\partial w_i} = -\sum_{(x,y)\text{missclassified}} \boldsymbol{x}y$
- Learning rule
  - Stochastic gradient descent for minimizing loss C
  - Repeat (t)
    - Choose an example $(\boldsymbol{x}(t), y(t))$
    - $\boldsymbol{w}(\mathrm{t}) = \boldsymbol{w}(\mathrm{t}-1) - \epsilon\, grad_{\boldsymbol{w}} \mathrm{c}(\boldsymbol{x}, \mathrm{y})$

Multi-class generalization

- Usual approach: one vs all
  - p classes = p " 2 class problems " : class C_i against the others
    - Learn p discriminant functions $F_i(\boldsymbol{x}), i = 1 \ldots p$
    - Decision rule: $\boldsymbol{x} \in Ci$ if $F_i(\boldsymbol{x}) > F_j(\boldsymbol{x})$ for $j \neq i$
    - This creates a partition of the input space
    - Each class is a polygon with at most $p - 1$ faces.
  - Convex regions: limits the expressive power of linear classifiers

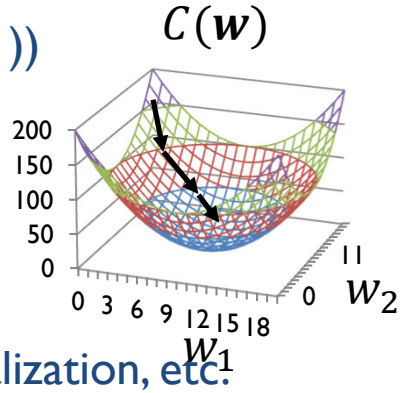
Adaline – Adaptive Linear Element (Widrow - Hoff 1959)





Linear unit ($f = Id$): $F(x) = \sum_i \ w_i x_i + w_0$

- « Least Mean Square » LMS algorithm
  - Loss: $c(\boldsymbol{x}, \boldsymbol{y}) = ||\boldsymbol{y} - F(\boldsymbol{x})||^2$
  - Algorithm: Stochastic Gradient Descent (Robbins – Monro (1951))
    - Initialize $w(0)$
    - Iterate
      - Choose an example $(\boldsymbol{x}(t), y(t))$
      - $\mathbf{w}(t+1) = \mathbf{w}(t) - \epsilon \nabla_{\mathbf{w}}\, c(\boldsymbol{x}, \boldsymbol{y})$



  - Workhorse algorithm of adaptive signal processing: filtering, equalization, etc.

General framework for supervised learning
Risk – Empirical Risk - Probabilistic formalism

- Data
  - Random vectors $(\boldsymbol{z})$ generated from distribution $p(\boldsymbol{z})$
  - For supervised classification $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{y})$
- Learning model
  - $\boldsymbol{F} = \{F_\theta\}_\theta$ with θ the model parameters, usually real parameters
- Loss
  - $c_\theta(\boldsymbol{z})$ for model F_θ and example $\boldsymbol{z}$
  - $c_\theta(\boldsymbol{z})$ examples: mean square or cross-entropy
- Risk
  - $R_\theta = E_{\boldsymbol{z}}[c_\theta(\boldsymbol{z})] = \int_z c_\theta(\boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z}$
- Optimal solution
  - $F_{\theta^*} = argmin_\theta R_\theta$

General framework for supervised learning
Risk – Empirical Risk - Learning from examples

- Data
  - $D = \{\boldsymbol{z}^i\}_{i=1..N}$
- Empirical risk
  - $C = \frac{1}{N}\sum_{i=1}^{N} c_\theta(\boldsymbol{z}^i)$
- Empirical Risk Minimization principle (ERM)
  - F_{θ^*} minimizing the theoretical risk is approximated by $F_{\hat{\theta}}$ mimizing the empirical risk
  - Is that sufficient ? Answer is No – but for now we will proceed with ERM
- Inductive framework
  - We will consider the following ML framework
    - The model learns on an available training set
    - Once trained parameters are fixed and the model can be used for inference and/or evaluated on a test set

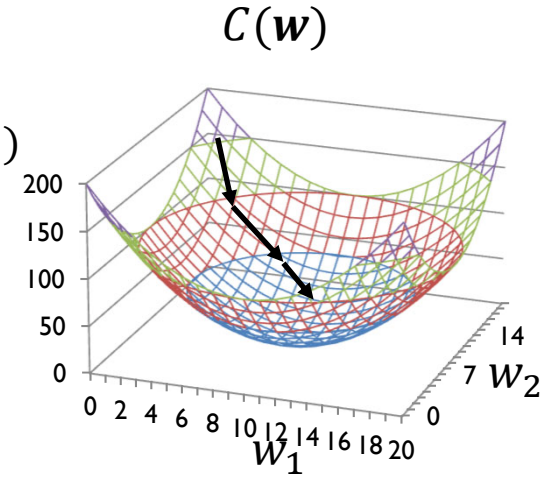
Introductory concepts
Summary of key ideas

- Learning from examples
  - Perceptron and Adaline are supervised learning algorithms
  - Training and test set concepts
    - Parameters are learned from a training set, performance is evaluated on a test set
    - Supervised means each example is a couple $(\boldsymbol{x}, \boldsymbol{y})$
- Stochastic optimization algorithms
  - Training requires exploring the parameter space of the model (the weights)
  - For NNs, most optimization methods are based on stochastic gradient descent
- Generalization properties
  - Learning ≠ Optimization
  - One wants to learn functions that generalize well

Optimisation : gradient methods – introduction

Optimization
Batch gradient algorithms

- Batch gradient general scheme
  - Training Data Set
    - $D = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^N, \boldsymbol{y}^N)\}$
  - Objective
    - Optimize a loss function $C(\boldsymbol{w}) = \sum_{i=1}^{N} c_{\mathrm{w}}(\boldsymbol{x}^i, \boldsymbol{y}^i)$
      - Sum of invidual losses $c_{\mathrm{w}}(.,.)$ on each example $(\boldsymbol{x}^i, \boldsymbol{y}^i)$
  - Principle
    - Initialize $\boldsymbol{w} = \boldsymbol{w}(0)$
    - Iterate until convergence
      - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \epsilon(t)\boldsymbol{\Delta}_{\mathbf{w}}(t)$



    - $\boldsymbol{\Delta}_{\mathbf{w}}(t)$ is the descent direction, $\epsilon(t)$ is the gradient step
  - Both are determined via local information computed from $\boldsymbol{C}(\boldsymbol{w})$, using approximations of the 1st or 2nd order of $\boldsymbol{C}(\boldsymbol{w})$
    - e.g. steepest descent, is a 1^{st} order gradient with: $\boldsymbol{\Delta}_{\mathbf{w}}(t) = -\nabla_{\boldsymbol{w}} C(\mathrm{t})$, and $\epsilon(t) = \epsilon$

Optimization
Stochastic gradient algorithms (From Ruder 2016)

- The most common family of optimization methods for Deep NN is based on **stochastic gradient** algorithms
  - Exploit the redundency in the data, at the cost of high variance in gradient estimates
- Data + Loss
  - Training Data Set
    - $D = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), \dots, (\boldsymbol{x}^N, \boldsymbol{y}^N)\}$
  - Loss function
    - $C(\boldsymbol{w}) = \sum_{i=1}^{N} c_{\mathrm{w}}(\boldsymbol{x}^i, \boldsymbol{y}^i)$
  - All the algorithms are given in vector form
- Basic Stochastic Gradient Descent
  - Initialise $\boldsymbol{w}(0)$
  - Iterate until stop criterion
  - sample an exemple $(\boldsymbol{x}(t), \mathrm{y}(t))$
  - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \epsilon \nabla_{\boldsymbol{w}} c(\boldsymbol{x}(t), \boldsymbol{y}(t))$
  - Rq: might produce a lot of oscillations

(a) SGD without momentum

Figure from (Ruder 2016)

Optimization
SGD algorithm with momentum and Adaptive learning rate

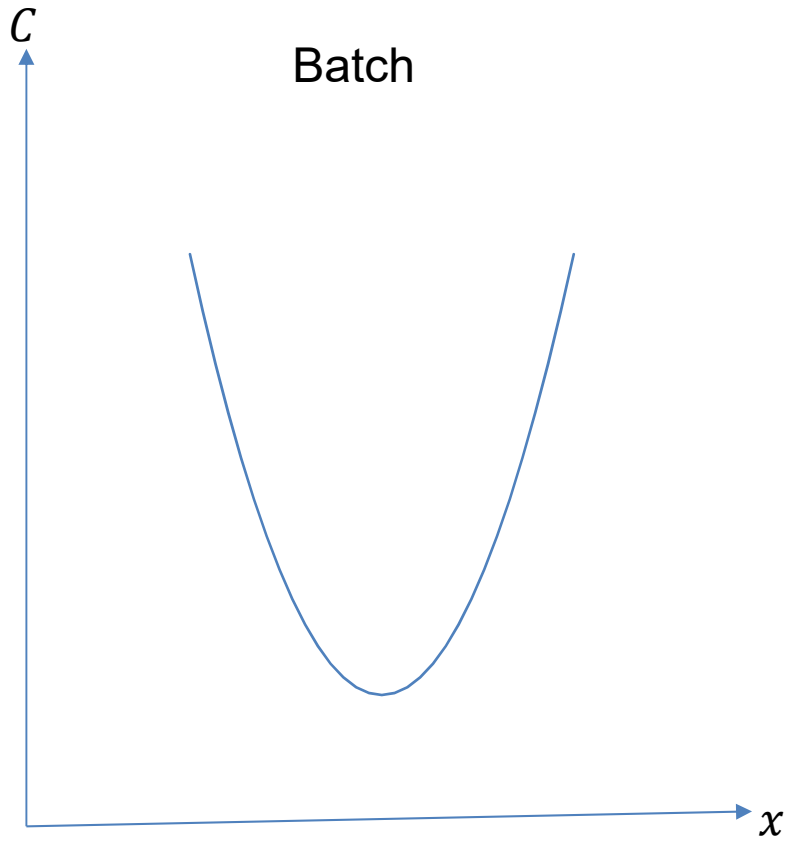
- Adam (adaptive moment estimation)
  - Computes
    - Adaptive learning rates for each parameter
    - An exponentially decaying avarage of past gradients (momentum)
    - An exponentially decaying average of past squared gradients (like RMSprop)
  - Iteration t
    - Momentum term : $\boldsymbol{m}(\boldsymbol{t}) = \gamma_1 \boldsymbol{m}(t-1) + \epsilon(1-\gamma_1)\boldsymbol{g}(t)$
    - Gradient variance term: $\mathrm{r}(t) = \gamma_2 \boldsymbol{r}(t-1) + \epsilon(1-\gamma_2)\boldsymbol{g}(t) \odot \boldsymbol{g}(t)$
    - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \frac{\epsilon}{\sqrt{\boldsymbol{r}(t)}+\epsilon\prime} \odot \boldsymbol{m}(t)$
    - Bias correction
      - The 2 moments are initialized at 0, they tend to be biased towards 0, the following correction terms reduce this effect
        - Correct bias of $\boldsymbol{m}$: $\boldsymbol{m}(\boldsymbol{t}) = \frac{\boldsymbol{m}(\boldsymbol{t})}{1-\gamma_1^t}$
        - Correct bias of $\boldsymbol{r}$: $\mathbf{r}(t) = \frac{\boldsymbol{r}(\boldsymbol{t})}{1-\gamma_2^t}$

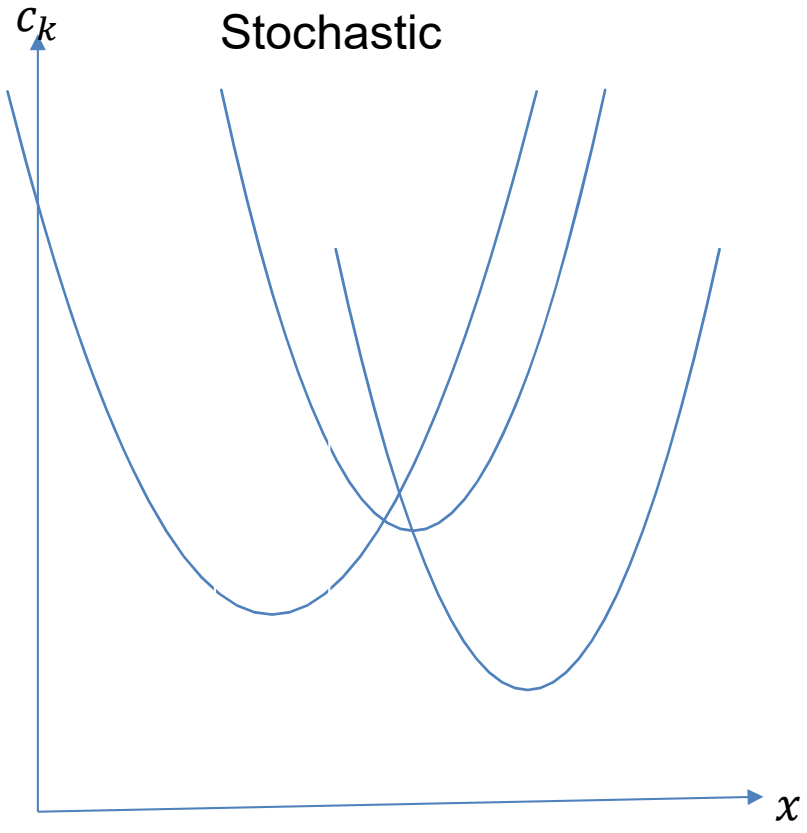
Batch vs stochastic gradient

$$C = \frac{1}{N}\sum_k c_k$$

C: global loss

c_k: individual (pattern k) loss

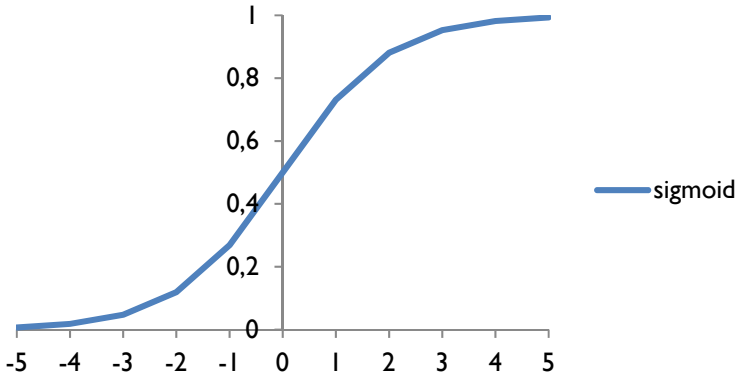




Computational Graph Logistic Regression

Training a single sigmoid neuron for 2 class – classification a.k.a. logistic regression

- Let us consider a 2 class classification problem
- And a sigmoid neuron (same form as for the logistic regression model)
  - $F_w(\boldsymbol{x}) = \sigma(\boldsymbol{w}.\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}.\boldsymbol{x})}$
  - Logistic (or sigmoid) function
    - $\sigma(z) = \frac{1}{1+\exp(-z)}$
      - hint
        - $\sigma'(z) = \sigma(z)(1-\sigma(z))$



- Training set
  - $D = \{(x^1, y^1), \ldots, (x^1, y^1)\}$
  - $\mathrm{x} \in R^n, y \in \{0,1\}$

Logistic regression (2 classes)
Probabilistic interpretation

- Since y $\in \{0,1\}$, we make a Bernoulli hypothesis for the posterior distribution of the outputs
  - $p(y = 1|\boldsymbol{x}; \boldsymbol{w}) = F_w(\boldsymbol{x})$ et $p(y = 0|\boldsymbol{x}; \boldsymbol{w}) = 1 - F_w(\boldsymbol{x})$
  - In compact format
    - $p(y|\boldsymbol{x}; \boldsymbol{w}) = \left(F_w(\boldsymbol{x})\right)^y \left(1 - F_w(\boldsymbol{x})\right)^{1-y}$ with y $\in \{0,1\}$
- Likelihood
  - $L(\boldsymbol{w}) = \prod_{i=1}^{N} \left(F_w(\boldsymbol{x}^i)\right)^{y^i} \left(1 - F_w(\boldsymbol{x}^i)\right)^{1-y^i}$
- Log-likelihood
  - $l(\boldsymbol{w}) = \sum_{i=1}^{N} y^i log F_w(\boldsymbol{x}^i) + (1 - y^i)(\log(1 - F_w(\boldsymbol{x}^i))$
    - This is minus the cross-entropy between the target and the estimated posterior distribution
    - Training objective: maximize log-likelihood or minimize cross-entropy

Logistic regression (2 classes)
SGD algorithm for minimizing the cross-entropy

- Initialise $\boldsymbol{w}$
- Iterate
  - Sample an example (x^i, y^i)
  - Compute the gradient $\nabla_w l(x^i, y^i)$
    - $\nabla_w l(x^i, y^i) = \left(y^i - F_w(\boldsymbol{x}^i)\right)\boldsymbol{x}^i$
  - Update the weights (gradient ascent for maximizing the likelihood)
    - $\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla_w C = \boldsymbol{w} + \epsilon \left(y^i - F_w(\boldsymbol{x}^i)\right)\boldsymbol{x}^i$

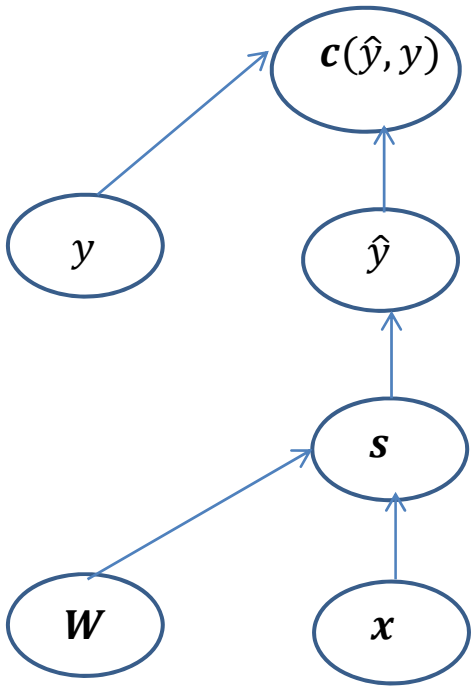
- Note: componentwise the gradient writes: $\frac{\partial l(\boldsymbol{w})}{\partial w_k} = \sum_{i=1}^{N} \left(y^i - F_w(\boldsymbol{x}^i)\right)\boldsymbol{x}_k^i$

Logistic regression – Computational graph -SGD

- Forward pass

$\boldsymbol{c}(\hat{y}, y)$: loss

y: target



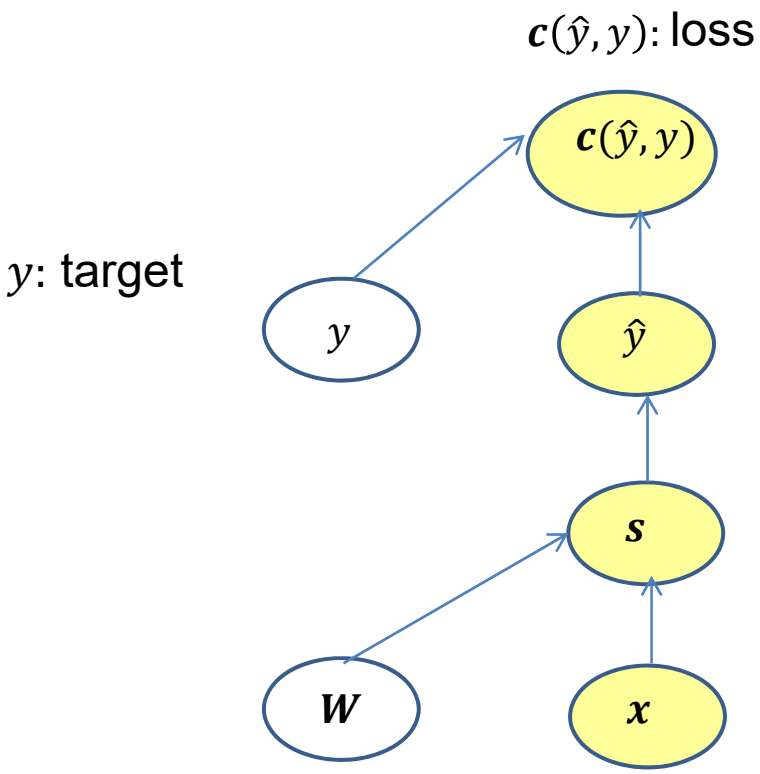
Forward propagation:

$s = \boldsymbol{w}.\boldsymbol{x}$

$\hat{y} = \sigma(s)$

Logistic regression – Computational graph - SGD

- Forward pass



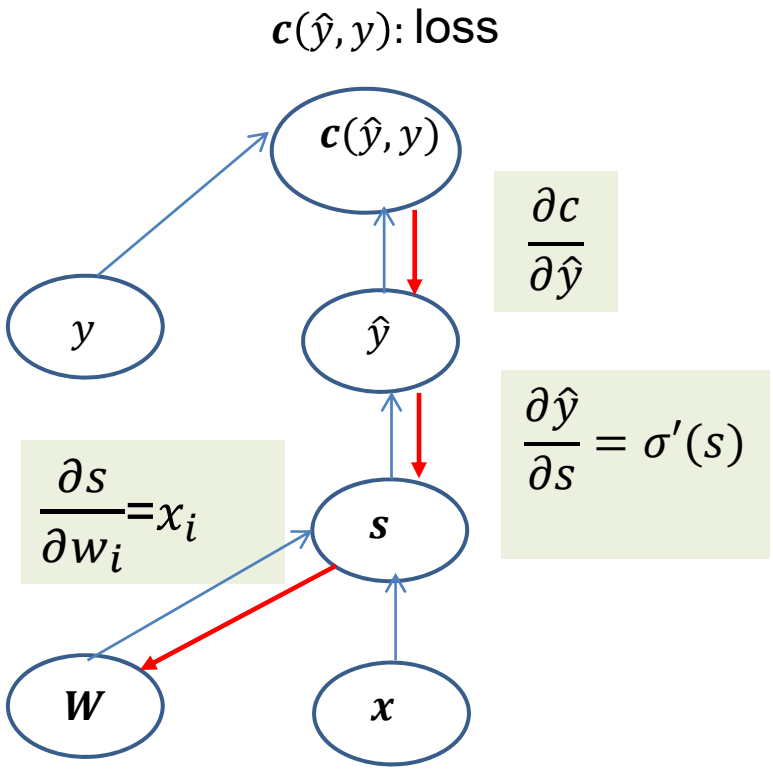
Forward propagation:

$s = \boldsymbol{w}.\boldsymbol{x}$

$\hat{y} = \sigma(s)$

Logistic regression – Computational graph - SGD

- Backward pass



Backward propagation:

$$\frac{\partial c}{\partial s} = \frac{\partial c}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial s}$$

$$\frac{\partial c}{\partial w_i} = \frac{\partial c}{\partial s} \frac{\partial s}{\partial w_i}$$

$$\frac{\partial c}{\partial w_i} = \frac{\partial c}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial s} \frac{\partial s}{\partial w_i}$$

Chain Rule

Logistic regression – Computational graph - SGD

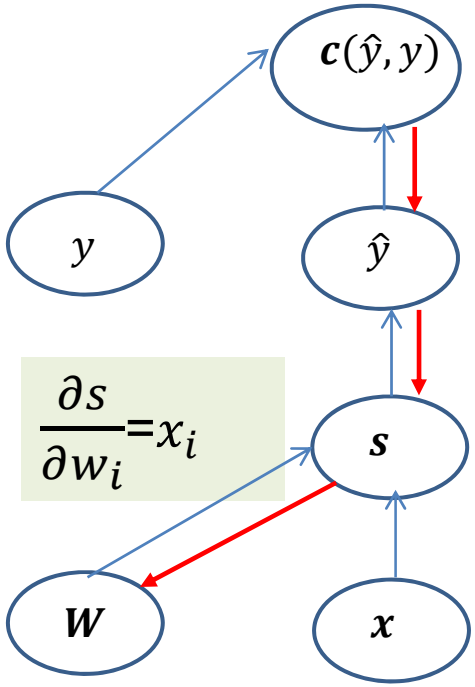
- Backward pass

For the cross entropy loss

$$l(\boldsymbol{w}) = \sum_{i=1}^{N} y^i log\hat{y}^i + (1-y^i)\log(1-\hat{y}^i) = \sum_{i=1}^{N} \boldsymbol{c}(\hat{y}^i, y^i)$$

$\boldsymbol{c}(\hat{y}, y)$: loss

y: target



$$\frac{\partial c}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial \hat{y}}{\partial s} = \sigma'(s)$$

$$\frac{\partial s}{\partial w_i} = x_i$$

Backward propagation:

$$\frac{\partial c}{\partial s} = \frac{\partial c}{\partial \hat{y}}\sigma'(s)$$

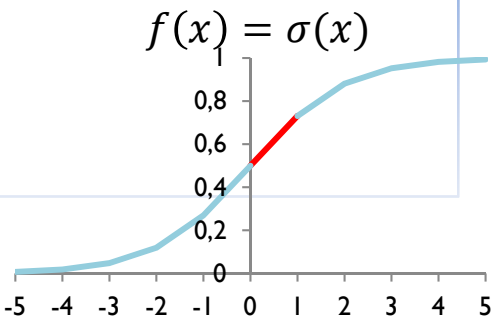
$$\frac{\partial c}{\partial w_i} = \frac{\partial c}{\partial s}x_i$$

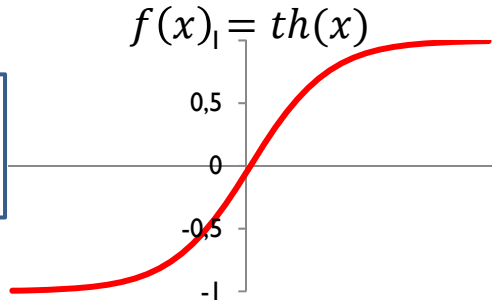
$$\frac{\partial c}{\partial w_i} = \left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right)\sigma'(s)x_i$$

Multi-layer Perceptron

Multi-layer Perceptron (Hinton – Sejnowski – Williams 1986)

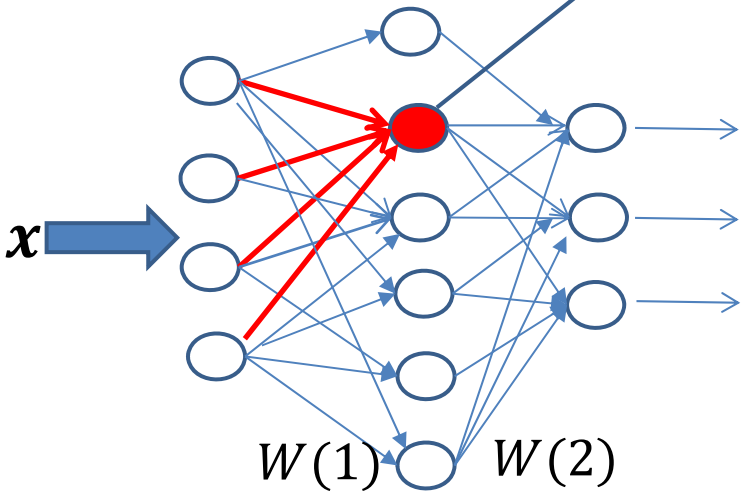
- Neurons arranged into layers
- Each neuron is a non linear unit, e.g.





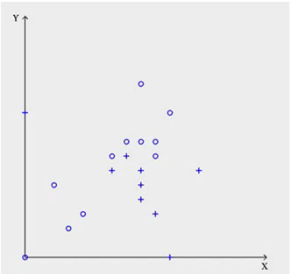
$f(\boldsymbol{w}.\boldsymbol{x})$

$\boldsymbol{w}$: **cell weight vector**



$$\hat{\boldsymbol{y}} = F_w(\boldsymbol{x}) = f\odot(W(2)f\odot(W(1)\boldsymbol{x}))$$

http://playground.tensorflow.org/



Note: $\odot$ *is a pointwise operator*, if $\boldsymbol{x} = (x_1, x_2)$, $f\odot\big((x_1, x_2)\big) = (f(x_1), f(x_2))$

Multi-layer Perceptron - Training

- **Stochastic Gradient Descent** - The algorithm is called **Back-Propagation**
  - Pick one example (x, y) or a **Mini Batch** $\{(x^i, y^i)\}$ sampled from the training set
    - Here the algorithm is described for 1 example (pure SGD) and for the sigmoid ($f(\ \ ) = \sigma(\ \ )$) non linearity
  - Forward pass
    - $\hat{y} = F_w(x) = f(W(2)f(W(1)x))$
  - Compute error
    - $c(y, \hat{y})$, e.g. mean square error or cross entropy
  - Backward pass
    - efficient implementation of chain rule
    - $W = W - \epsilon \frac{\partial c(y,\hat{y})}{\partial W}$

Algorithmic differentiation

- Back-Propagation is an instance of **automatic differentiation / algorithmic differentiation - AD**
  - A mathematical expression can be written as a **computation graph**
    - i.e. graph decomposition of the expression into elementary computations
  - **AD** allows to **compute** efficiently the derivatives of every element in the graph w.r.t. any other element.
  - **AD** transforms a programs computing a numerical funtion into the program for computing the derivatives

  - All DL framework implement AD
    - There are multiple instances of AD – we will use here reverse AD

Notations – vector calculus - derivatives

- Different conventions exist for vector/ matrix derivatives
- We use here the « denominator layout » convention most often used for BP derivation
  - Gradients and their arguments keep the same shape

$y = f(x), f: \mathbb{R}^n \rightarrow \mathbb{R}$

- $\nabla_x y = \frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{pmatrix} \in \mathbb{R}^n$ Gradient

$y = f(x), f: \mathbb{R}^n \rightarrow \mathbb{R}^m$

- $J_{y(x)} = \frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{nxm}$ Jacobian

$y = f(W), f: \mathbb{R}^{mxn} \rightarrow \mathbb{R}$

- $\nabla_W y = \frac{\partial y}{\partial W} = \begin{pmatrix} \frac{\partial y}{\partial w_{11}} & \cdots & \frac{\partial y}{\partial w_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial w_{m1}} & \cdots & \frac{\partial y}{\partial w_{mn}} \end{pmatrix} \in \mathbb{R}^{mxn}$ Gradien

Matrix cookbooks
http://www.cs.toronto.edu/~roweis/notes/matrixid.pdf –
http://www.imm.dtu.dk/pubdb/views/edoc_download.php/327
4/pdf/ imm3274.pdf
https://en.wikipedia.org/wiki/Matrix_calculus

Notations – vector calculus - derivatives

- Additional useful derivations
- For matrix W and vector x
  - $\frac{\partial Wx}{\partial x} = W^T$
- For $u = f(x); f: \mathbb{R}^n \to \mathbb{R}^p \; ; \; y = g(u); g: \mathbb{R}^p \to \mathbb{R}$
  - $\frac{\partial y}{\partial x_i} = \sum_{j=1}^{p} \frac{\partial y}{\partial u_j} \frac{\partial u_j}{\partial x_i}$

Vector chain rules – denominator layout

Vector case

$$u = f(x); f: \mathbb{R}^n \rightarrow \mathbb{R}^p$$
$$y = g(u); f: \mathbb{R}^p \rightarrow \mathbb{R}^m$$
$$\frac{\partial y}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial y}{\partial u} \in \mathbb{R}^{nxm}$$

Note:

$$\frac{\partial u}{\partial x} \in \mathbb{R}^{nxp}; \frac{\partial y}{\partial u} \in \mathbb{R}^{pxm}$$

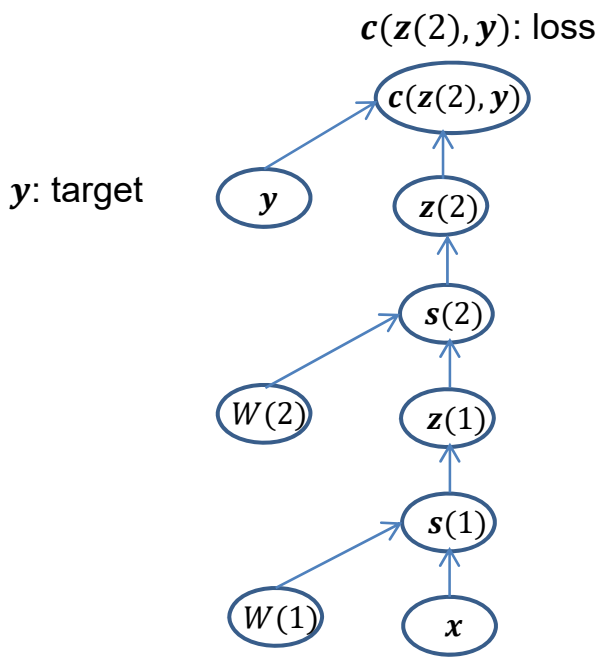
Scalar case

$$u = f(x); f: \mathbb{R}^n \rightarrow \mathbb{R}^p$$
$$y = g(u); f: \mathbb{R}^p \rightarrow \mathbb{R}$$
$$\nabla_x y = \frac{\partial y}{\partial x} = \frac{\partial u}{\partial x}\nabla_u y \in \mathbb{R}^n$$

$$\frac{\partial u}{\partial x} \in \mathbb{R}^{nxp}; \frac{\partial y}{\partial u} \in \mathbb{R}^{px1}$$

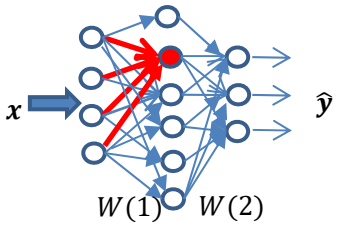
Multi-layer Perceptron - Training

- Computational graph



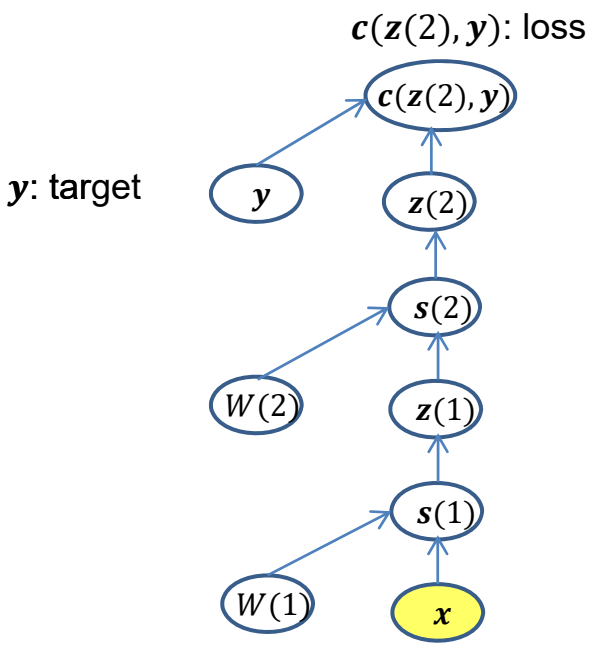
Here, $z(0) = x, z(2) = \hat{y}$

Forward propagation, $l = 1,2$:
$s(l) = W(l)z(l-1)$
$z(l) = \sigma(s(l))$



Multi-layer Perceptron - Training

- Forward pass

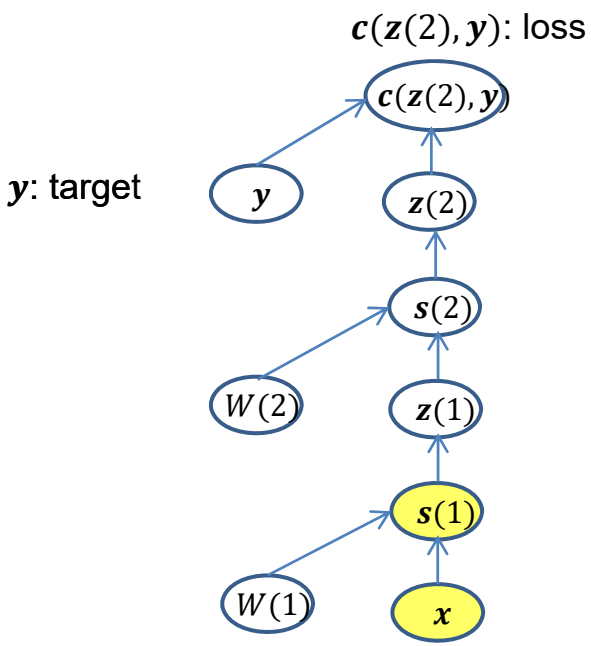


Here, $z(0) = x, z(2) = \hat{y}$

Forward propagation, $l = 1,2$:
$s(l) = W(l)z(l-1)$
$z(l) = \sigma(s(l))$

Multi-layer Perceptron - Training

- Forward pass

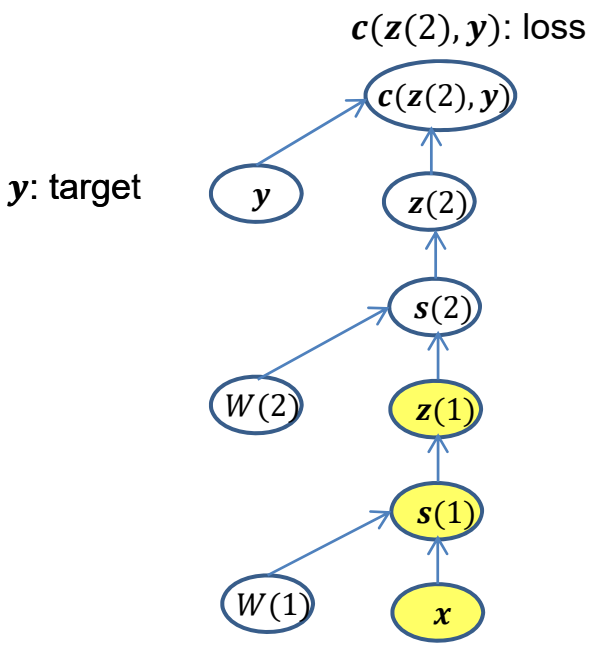


Here, $z(0) = x, z(2) = \hat{y}$

Forward propagation, $l = 1,2$:
$s(l) = W(l)z(l-1)$
$z(l) = \sigma(s(l))$

Multi-layer Perceptron - Training

- Forward pass

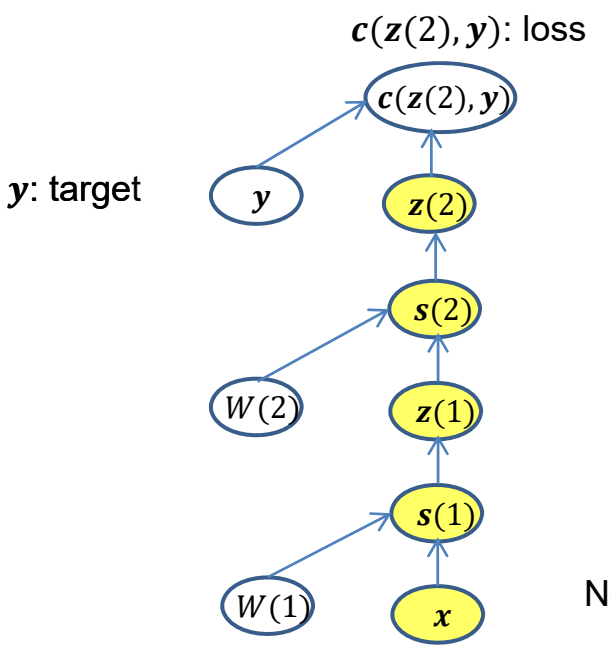


Here, $z(0) = x, z(2) = \hat{y}$

Forward propagation, $l = 1,2$:
$s(l) = W(l)z(l-1)$
$z(l) = \sigma(s(l))$

Multi-layer Perceptron - Training

- Forward pass



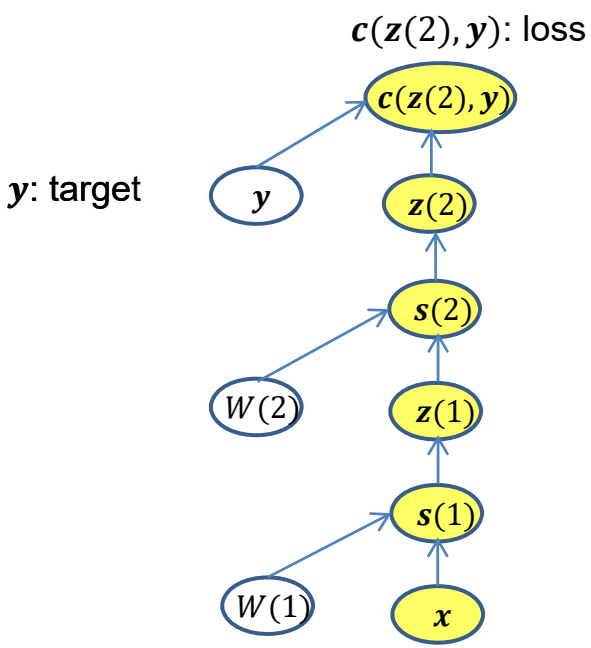
Here, $z(0) = x, z(2) = \hat{y}$

Forward propagation, $l = 1,2$:
$s(l) = W(l)z(l-1)$
$z(l) = \sigma(s(l))$

Note: x, $s(l)$, $x(l)$ are vectors

Multi-layer Perceptron - Training

- Forward pass



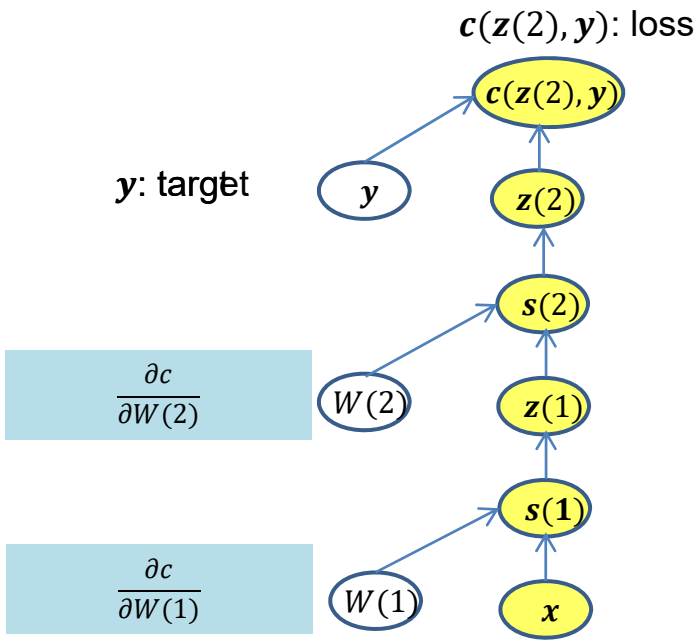
Here, $z(0) = x, z(2) = \hat{y}$

Forward propagation, $l = 1,2$:
$s(l) = W(l)z(l-1)$
$z(l) = \sigma(s(l))$

Note: x, $s(l)$, $x(l)$ are vectors

Multi-layer Perceptron - Training

- Back Propagation: Reverse Mode Differentiation

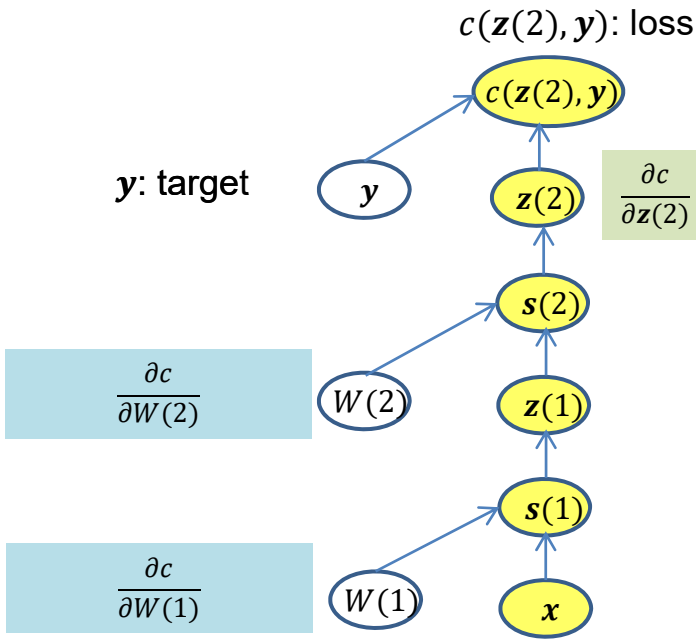


$$W = W - \epsilon \frac{\partial c}{\partial W}$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix of the same shape as W

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation



Backward propagation, $l = 1,2$

$$\frac{\partial c}{\partial \boldsymbol{s}(l)} = \frac{\partial c}{\partial \boldsymbol{z}(l)} \odot \sigma'(\boldsymbol{s}(l))$$

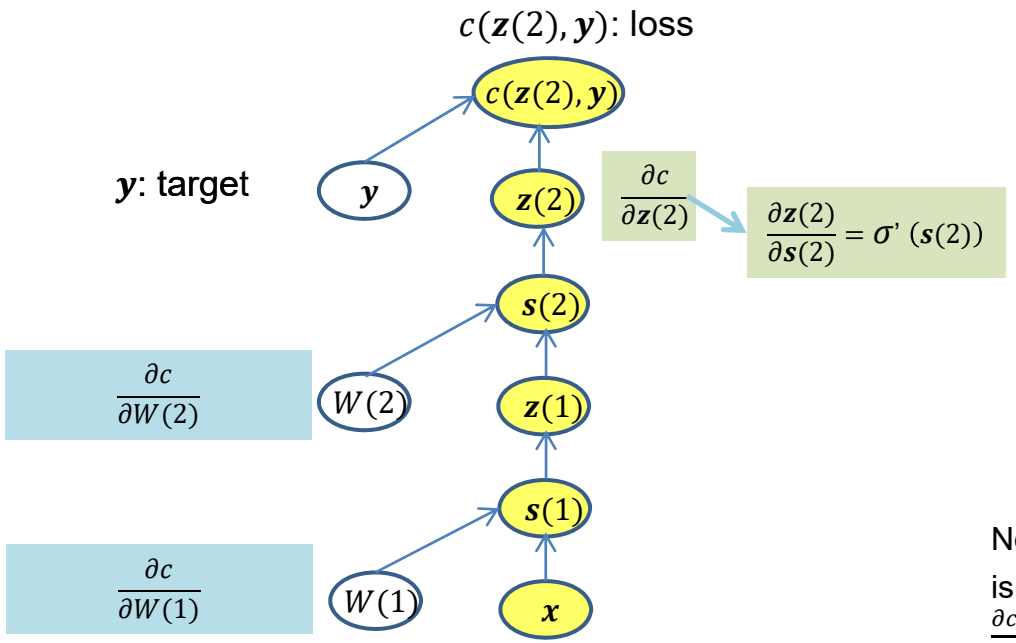
$$\frac{\partial c}{\partial W(l)} = \frac{\partial c}{\partial \boldsymbol{s}(l)} \boldsymbol{z}(l-1)^T$$

$$\frac{\partial c}{\partial \boldsymbol{z}(l-1)} = W(l)^T \frac{\partial c}{\partial \boldsymbol{s}(l)}$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix of the same shape as W, $\frac{\partial c}{\partial z}$ and $\frac{\partial c}{\partial s}$ are column vectors of the same size as $\boldsymbol{z}$ and $\boldsymbol{s}$

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation



Backward propagation , $l = 1,2$

$$\frac{\partial c}{\partial \boldsymbol{s}(l)} = \frac{\partial c}{\partial \boldsymbol{z}(l)} \odot \sigma'(\boldsymbol{s}(l))$$

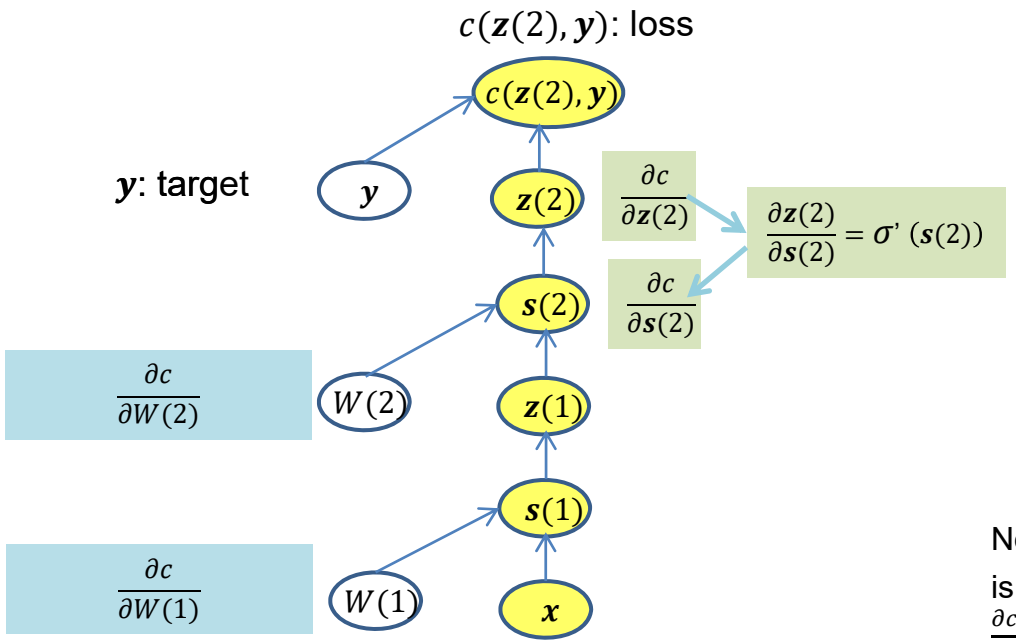
$$\frac{\partial c}{\partial W(l)} = \frac{\partial c}{\partial \boldsymbol{s}(l)} \boldsymbol{z}(l-1)^T$$

$$\frac{\partial c}{\partial \boldsymbol{z}(l-1)} = W(l)^T \frac{\partial c}{\partial \boldsymbol{s}(l)}$$

Note: notations are in vector form, $\frac{\partial \boldsymbol{c}}{\partial W}$ is a matrix of the same shape as W, $\frac{\partial c}{\partial z}$ and $\frac{\partial c}{\partial s}$ are column vectors of the same size as $\boldsymbol{z}$ and $\boldsymbol{s}$

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation



Backward propagation , $l = 1,2$

$$\frac{\partial c}{\partial \boldsymbol{s}(l)} = \frac{\partial c}{\partial \boldsymbol{z}(l)} \odot \sigma'(\boldsymbol{s}(l))$$

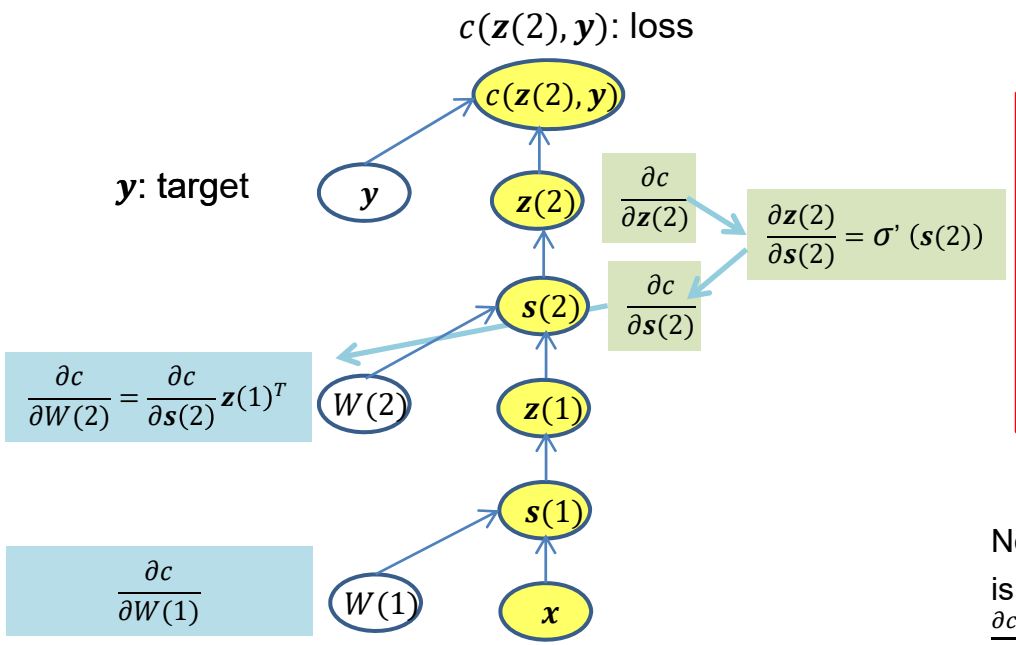
$$\frac{\partial c}{\partial W(l)} = \frac{\partial c}{\partial \boldsymbol{s}(l)} \boldsymbol{z}(l-1)^T$$

$$\frac{\partial c}{\partial \boldsymbol{z}(l-1)} = W(l)^T \frac{\partial c}{\partial \boldsymbol{s}(l)}$$

Note: notations are in vector form, $\frac{\partial \boldsymbol{c}}{\partial W}$ is a matrix of the same shape as W, $\frac{\partial c}{\partial z}$ and $\frac{\partial c}{\partial s}$ are column vectors of the same size as $\boldsymbol{z}$ and $\boldsymbol{s}$

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation



Backward propagation , $l = 1,2$

$$\frac{\partial c}{\partial \boldsymbol{s}(l)} = \frac{\partial c}{\partial \boldsymbol{z}(l)} \odot \sigma'(\boldsymbol{s}(l))$$

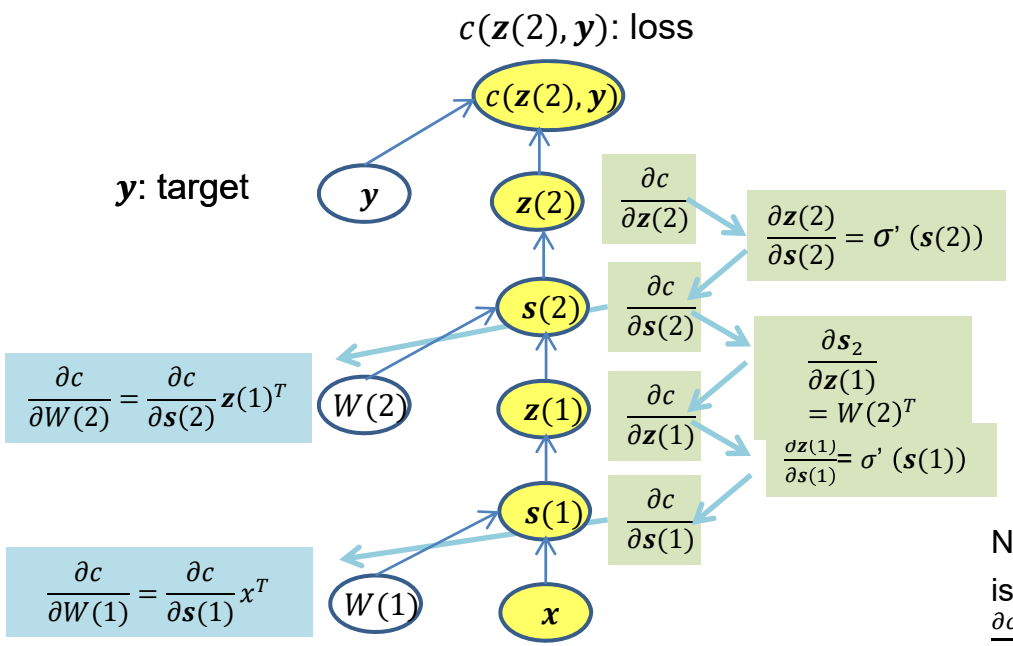
$$\frac{\partial c}{\partial W(l)} = \frac{\partial c}{\partial \boldsymbol{s}(l)} \boldsymbol{z}(l-1)^T$$

$$\frac{\partial c}{\partial \boldsymbol{z}(l-1)} = W(l)^T \frac{\partial c}{\partial \boldsymbol{s}(l)}$$

Note: notations are in vector form, $\frac{\partial \boldsymbol{c}}{\partial W}$ is a matrix of the same shape as W, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are column vectors of the same size as $\boldsymbol{z}$ and $\boldsymbol{s}$

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation



Backward propagation , $l = 1,2$

$$\frac{\partial c}{\partial \boldsymbol{s}(l)} = \frac{\partial c}{\partial \boldsymbol{z}(l)} \odot \sigma'(\boldsymbol{s}(l))$$

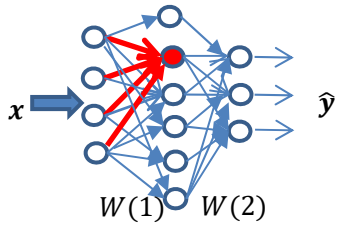
$$\frac{\partial c}{\partial W(l)} = \frac{\partial c}{\partial \boldsymbol{s}(l)} \boldsymbol{z}(l-1)^T$$

$$\frac{\partial c}{\partial \boldsymbol{z}(l-1)} = W(l)^T \frac{\partial c}{\partial \boldsymbol{s}(l)}$$

Note: notations are in vector form, $\frac{\partial \boldsymbol{c}}{\partial W}$ is a matrix of the same shape as W, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are column vectors of the same size as $\boldsymbol{z}$ and $\boldsymbol{s}$

Chain rule derivation – vector form
MLP architecture

- Let us consider a two layers fully connected MLP
  - $x \in \mathbb{R}^n$
  - $s(1) = W(1)x \in \mathbb{R}^p$
  - $z(1) = \sigma\big(s(1)\big) \in \mathbb{R}^p$
  - $s(2) = W(2)z(1) \in \mathbb{R}^m$
  - $z(2) = \sigma\big(s(2)\big) \in \mathbb{R}^m$
  - $c = c\big(y, z(2)\big) \in \mathbb{R}$



Chain rule derivation – vector form

- Error signal layer 2
  - $\delta_2 = \frac{\partial c}{\partial \boldsymbol{s}(2)} = \frac{\partial z(2)}{\partial \boldsymbol{s}(2)} \frac{\partial c}{\partial \boldsymbol{z}(2)}$
  - $\frac{\partial z(2)}{\partial \boldsymbol{s}(2)} = diag(\sigma'(s(2)) \in \mathbb{R}^{mxm}$
  - $\delta_2 = \frac{\partial c}{\partial \boldsymbol{z}(2)} \odot \sigma'(s(2)) \in \mathbb{R}^m$
- Gradient w.r.t. $W(2)$
  - $\frac{\partial c}{\partial W(2)} = \frac{\partial s(2)}{\partial W(2)} \frac{\partial c}{\partial \boldsymbol{s}(2)}$
  - $\frac{\partial c}{\partial W(2)} = \delta_2 z(1)^T$
- Error signal layer 1
  - $\delta_1 = \frac{\partial c}{\partial \boldsymbol{s}(1)} = \frac{\partial z(1)}{\partial \boldsymbol{s}(1)} \frac{\partial s(2)}{\partial z(1)} \frac{\partial c}{\partial s(2)}$
  - $s(2) = W(2)z(1) \Rightarrow \frac{\partial s(2)}{\partial z(1)} = W(2)^T$
  - $\delta_1 = W(2)^T \delta_2 \odot \sigma'(s(1)) \in \mathbb{R}^n$

Detailed derivations: https://www.kamperh.com/notes/kamper_backprop17.pdf

- Gradient w.r.t. $W(1)$
  - $\frac{\partial c}{\partial W(1)} = \frac{\partial s(1)}{\partial W(1)} \frac{\partial c}{\partial \boldsymbol{s}(1)}$
  - $\frac{\partial c}{\partial W(1)} = \delta_1 x^T$

Chain rule derivation – vector form

- $\frac{\partial c}{\partial W(\mathrm{t})} = \frac{\partial s(\mathrm{t})}{\partial W(\mathrm{t})} \frac{\partial c}{\partial \boldsymbol{s}(\mathrm{t})}$
  - $s_i(t) = \sum_j W_{ij} z_j(t-1)$
  - $\frac{\partial s_i}{\partial W_{\mathrm{kl}}(\mathrm{t})} = z_l(t-1)$ if $i = k, 0$ otherwise
  - $\frac{\partial c}{\partial W_{\mathrm{kl}}(\mathrm{t})} = \sum_i \frac{\partial c}{\partial s_i(t)} \frac{\partial s_i(t)}{\partial W_{\mathrm{kl}}(\mathrm{t})}$
  - $\frac{\partial c}{\partial W_{\mathrm{kl}}(\mathrm{t})} = [\delta_t]_k z_l(t-1)$
  - This is the classical form of the chain rule since the derivation is wrt a scalar W_{kl}
- $\frac{\partial c}{\partial W(\mathrm{t})} = \delta_t z_l^T(t-1)$

- $\frac{\partial \mathrm{c}}{\partial \mathrm{W}} = \begin{pmatrix} \frac{\partial c}{\partial W_{11}} & \cdots & \frac{\partial c}{\partial W_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial c}{\partial W_{m1}} & \cdots & \frac{\partial y_m}{\partial W_{mn}} \end{pmatrix}$
- $\frac{\partial \mathrm{c}}{\partial \mathrm{W}} = \begin{pmatrix} [\delta]_1 z_1 & \cdots & [\delta]_1 z_n \\ \vdots & \ddots & \vdots \\ [\delta]_m z_1 & \cdots & [\delta]_m z_n \end{pmatrix}$
- $\frac{\partial \mathrm{c}}{\partial \mathrm{W}} = \delta z^T$

Chain rule derivation – vector form

- Let us now consider a MLP with L layers and σ as activation function
- For an arbitray layer, the recursion takes the following form

$$\delta_l = \frac{\partial c}{\partial s(l)} = \frac{\partial c}{\partial z(l)} \odot \sigma'(s(l))$$

$$\delta_l = W(l+1)^T \delta_{l+1} \odot \sigma'(s(l)) \text{ for } l < L$$

$$\frac{\partial c}{\partial W(l)} = \delta_l z(l-1)^T$$

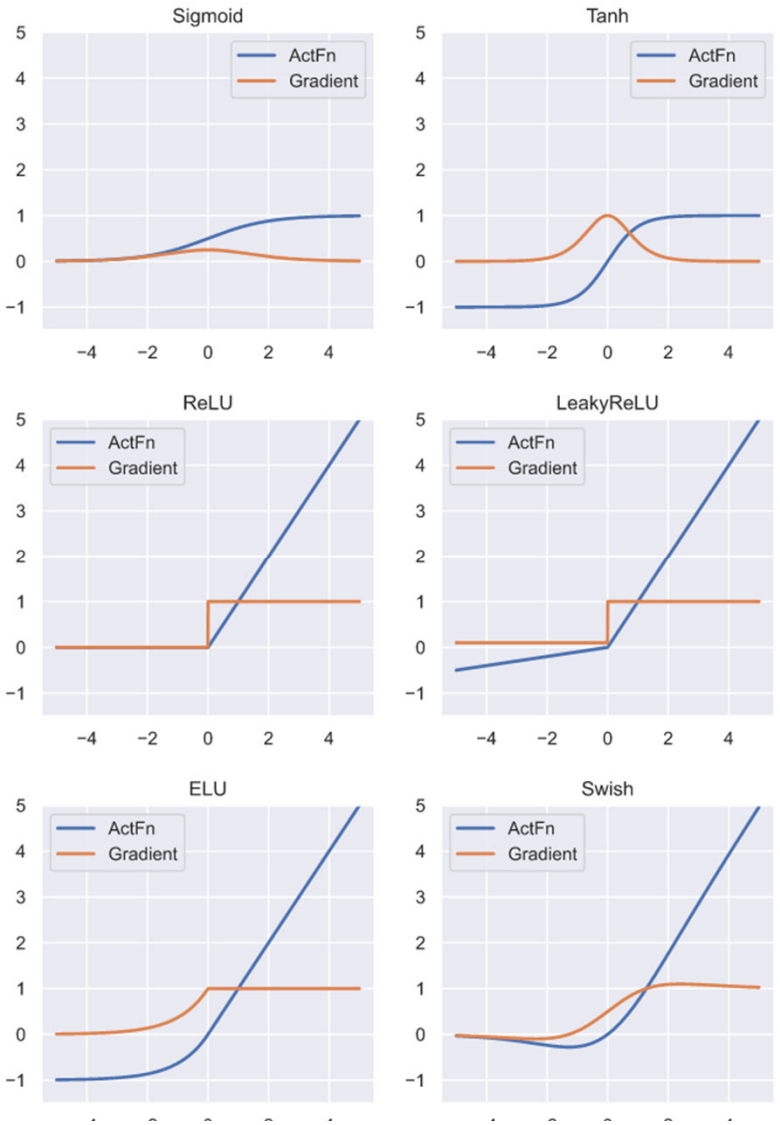
$$\frac{\partial c}{\partial \boldsymbol{z}(l-1)} = W^T(l)^T \delta_l$$

Deep Learning -Activation functions

Figures from:
https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial3/Activation_Functions.html

- In addition to the logistic or tanh units,, other forms are used in deep architectures – Some of the popular forms are:
  - Let $z = b + \boldsymbol{w}.\boldsymbol{x}$
  - RELU - Rectified linear units (used for internal layers)
    - $g(\boldsymbol{z}) = \max(0, \mathbf{z})$
      - Rectified units allow to draw activations to 0 (used for sparse representations) + derivative remain large when unit is active
  - Leaky RELU (used for internal layers)
    - $g(\boldsymbol{z}) = \begin{cases} \mathbf{z} \text{ if} b + \boldsymbol{w}.\boldsymbol{x} > 0 \\ 0.01(\boldsymbol{z}) \text{ otherwise} \end{cases}$
      - Introduces a small derivative when $b + \boldsymbol{w}.\boldsymbol{x} < \mathbf{0}$
  - ELU (used for internal layers)
    - $g(\boldsymbol{z}) = \begin{cases} \mathbf{z} \text{ if } \mathbf{z} > 0 \\ \alpha(\exp(b + \boldsymbol{w}.\boldsymbol{x}) - 1) \text{ otherwise} \end{cases}$
  - Swish
    - $g(\boldsymbol{z}) = \frac{\boldsymbol{z}}{1+\exp(-\boldsymbol{z})}$

x axis $b + \boldsymbol{w}.\boldsymbol{x}$, y axis $g(\boldsymbol{x})$



CNN: Convolutional Neural Nets

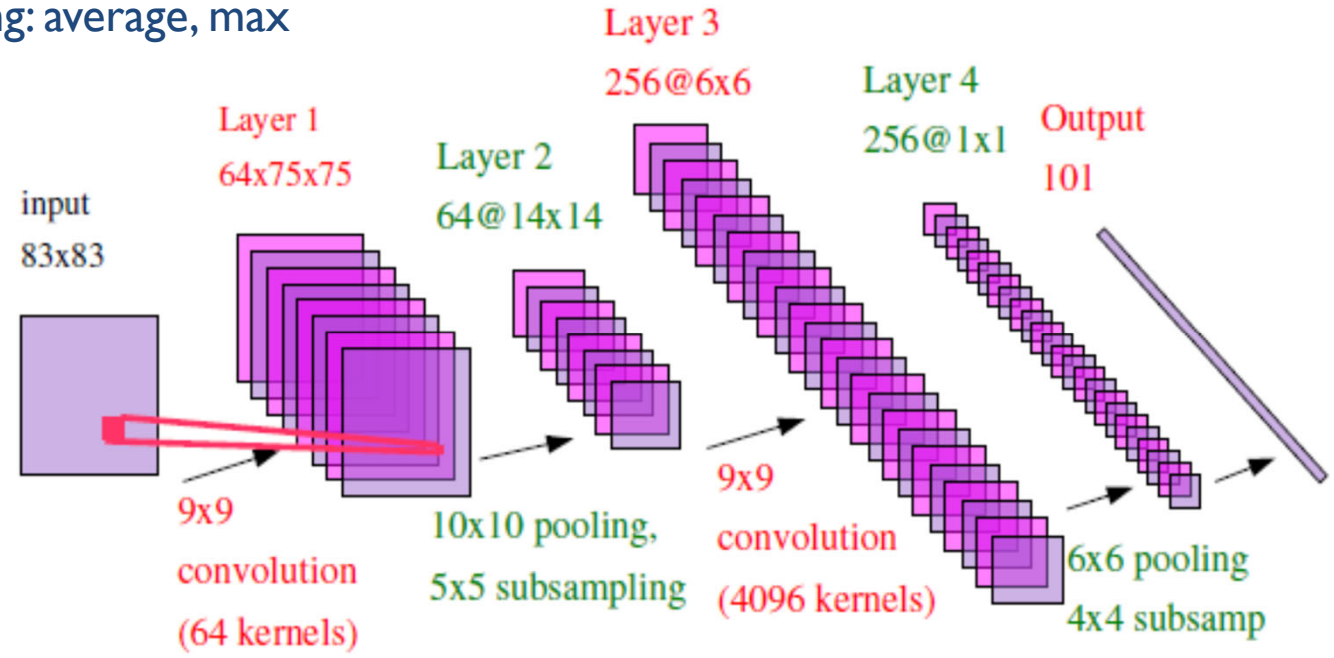
Introduction
Classification
Object detection
Image segmentation

CNNs

- CNNs were developped in the late 80ies for image and speech applications
- Deep CNNs were successfully used for image applications (classification and segmentation) in the 2010s – starting with the ImageNet competition, and for speech recognition.
  - Their use has been extended to handle several situations
  - They come now in many variants
  - They can often be used as alternatives to Recurrent NNs

CNNs
example

- ConvNet architecture (Y. LeCun since 1988)
  - Deployed at Bell Labs in 1989-90 for Zip code recognition
  - Character recognition
  - Convolution: non linear embedding in high dimension
  - Pooling: average, max



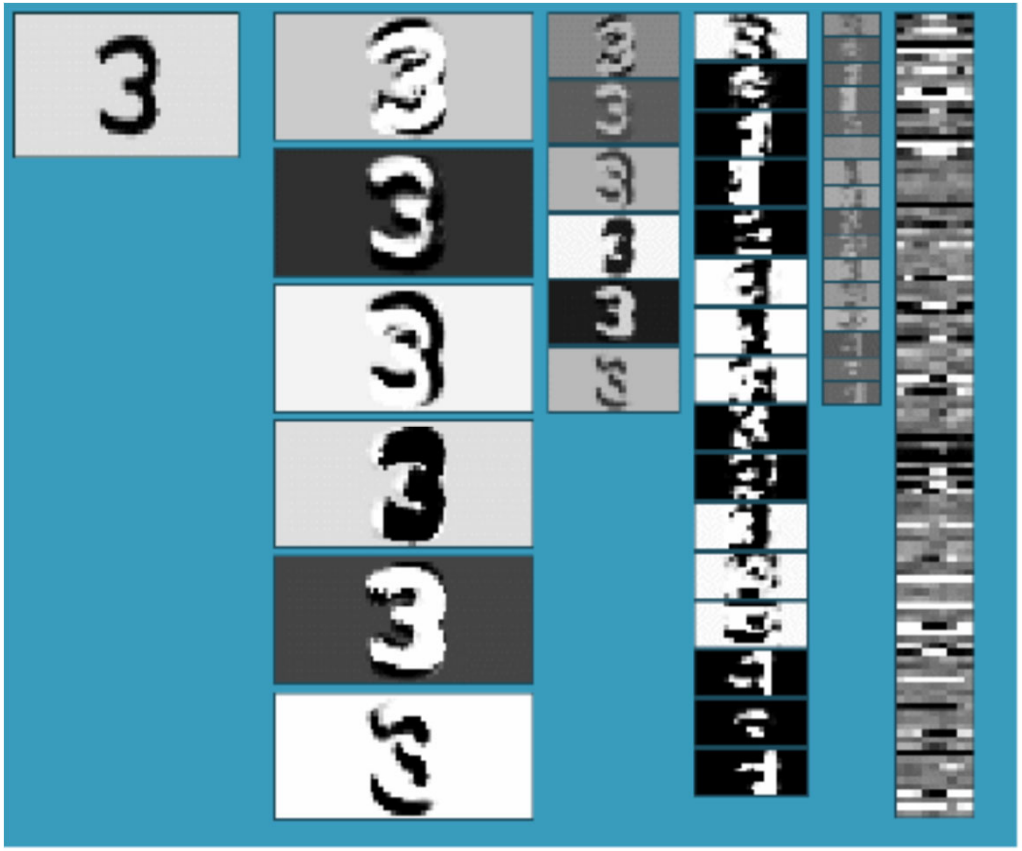
parameters 64x9x9=5184, 256x9x9=20736, 256x101 = 60916

CNNs

- In Convnet
  - The first hidden layer consists in 64 different convolution kernels over the initial input, resulting in 64 different mapping of the input
  - The second hidden layer is a sub-sampling layer with a pooling tranformation applied to each matrix representation of the first hidden layer
  - etc
  - Last layer is a classification layer, fully connected
- More generally
  - CNNs alternate convolution, and pooling layers, and a fully connected layer at the top.

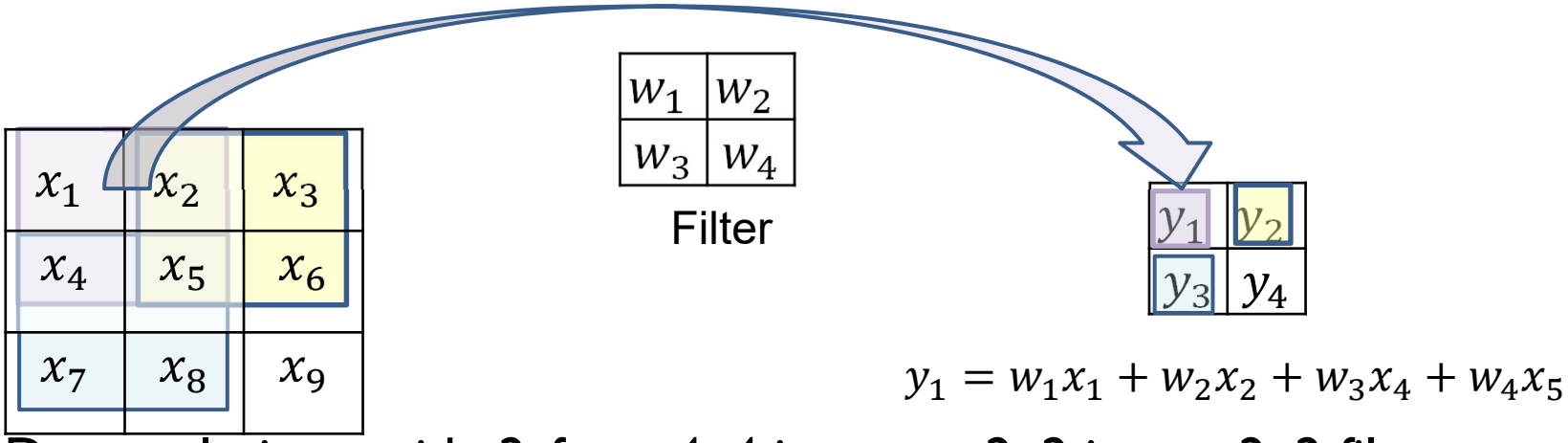
CNNs
visualization

- Hand writing recognition (Y. LeCun Bell labs 1989)



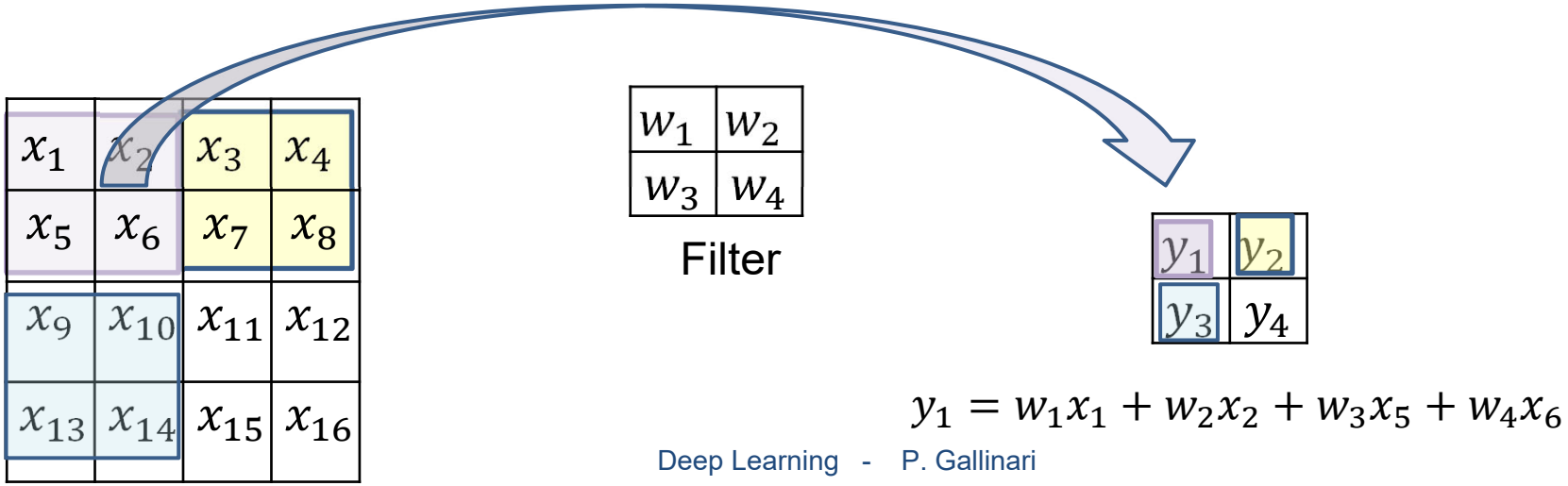
CNNs
Convolution: filter size and stride

- 2D convolution, stride 1, from 3x3 image to 2x2 image, 2x2 filter



$$y_1 = w_1x_1 + w_2x_2 + w_3x_4 + w_4x_5$$

- 2 D convolution, stride 2, from 4x4 image to 2x2 image, 2x2 filter



$$y_1 = w_1x_1 + w_2x_2 + w_3x_5 + w_4x_6$$

CNNs
Padding

- Padding amounts at filling the border of the image, usually with 0
  - The width of the padding border depends on the filter characteristics

0	0	0	0	0	0
0	x_1	x_2	x_3	x_4	0
0	x_5	x_6	x_7	x_8	0
0	x_9	x_{10}	x_{11}	x_{12}	0
0	x_{13}	x_{14}	x_{15}	x_{16}	0
0	0	0	0	0	0

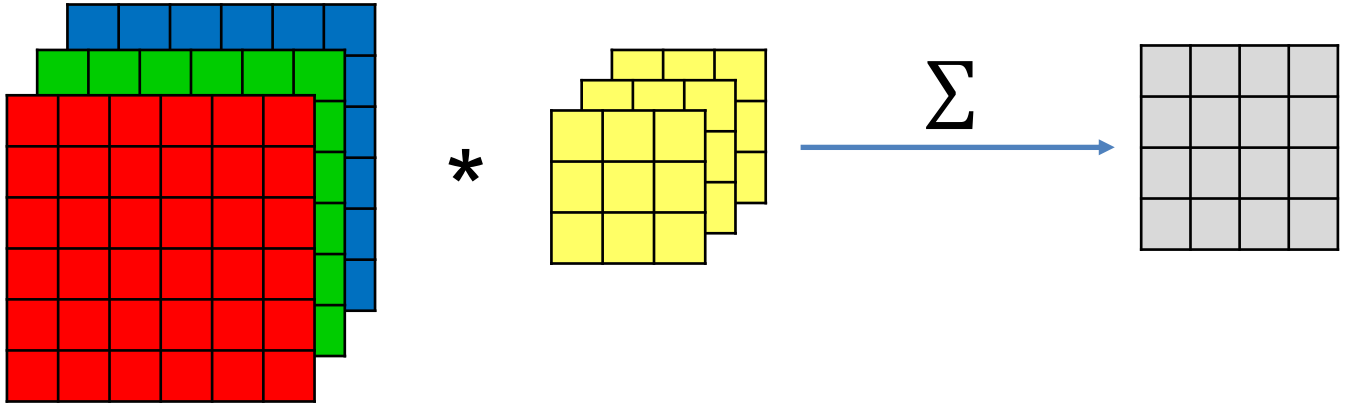
CNNs
Convolutions arithmetics

- Input image nxn, filter $f\,\mathrm{x}f$, padding p, stride s
- Output image is $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ x $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$
- Floor function [.]
  - in some cases a convolution will produce the same output size for multiple input sizes. If $i + 2p - k$ is a multiple of s, then any input size $j = i + a, a \in \{0, \ldots, s-1\}$ will produce the same output size. This applies only for $s > 1$.

Note: more in (Dumoulin 2016), a guide to convolution arithmetic for Deep Learning

CNNs
on multiple channels, e.g. RGB images

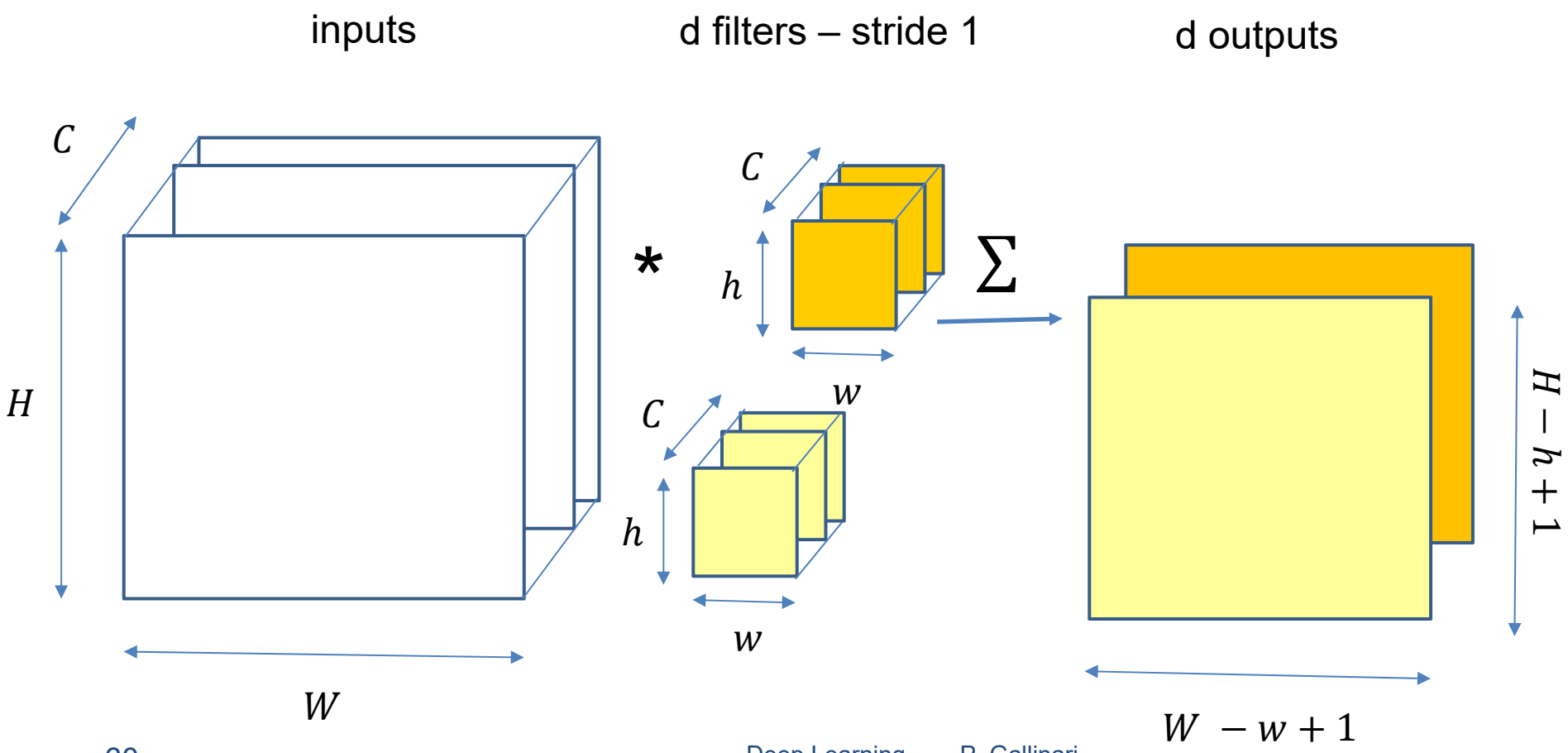
- Convolution generalizes to multiple channels. For images, the input is usually a 3 D tensor, and the output is a 2 D tensor: the filter is not swipped across channels usually, but only across rows and columns of the corresponding channel.



6x6x3 image 3x3 filters – stride 1 4x4 image

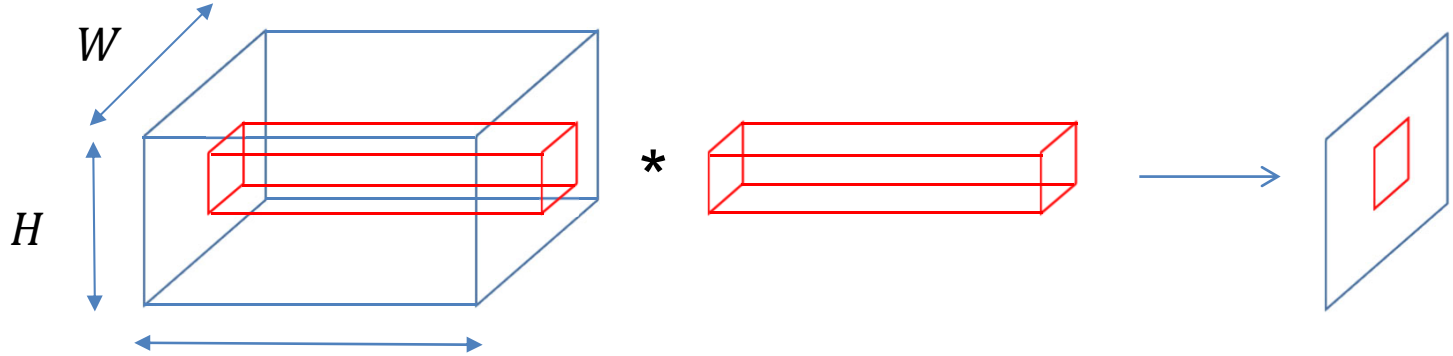
CNNs
on multiple channels

- This generalizes to any number of input channels, and filters
  - Below C input channels and 2 outputs



CNNs
1x1 convolutions on multiple channels

- 1x1 convolutions, perform a pixel wise weighted sum on several channels
  - They are used to reduce the size of a volume
    - e.g.transforming a HxWxC volume to a HxWxC' volume with $C' < C$, by using C', 1x1 convolutions



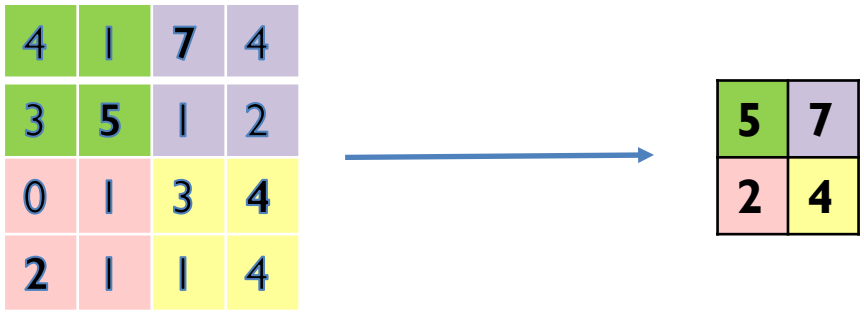
C channels

HxWxC 1x1xC HxWx1

$C' = 1$ in this example

CNNs
Pooling

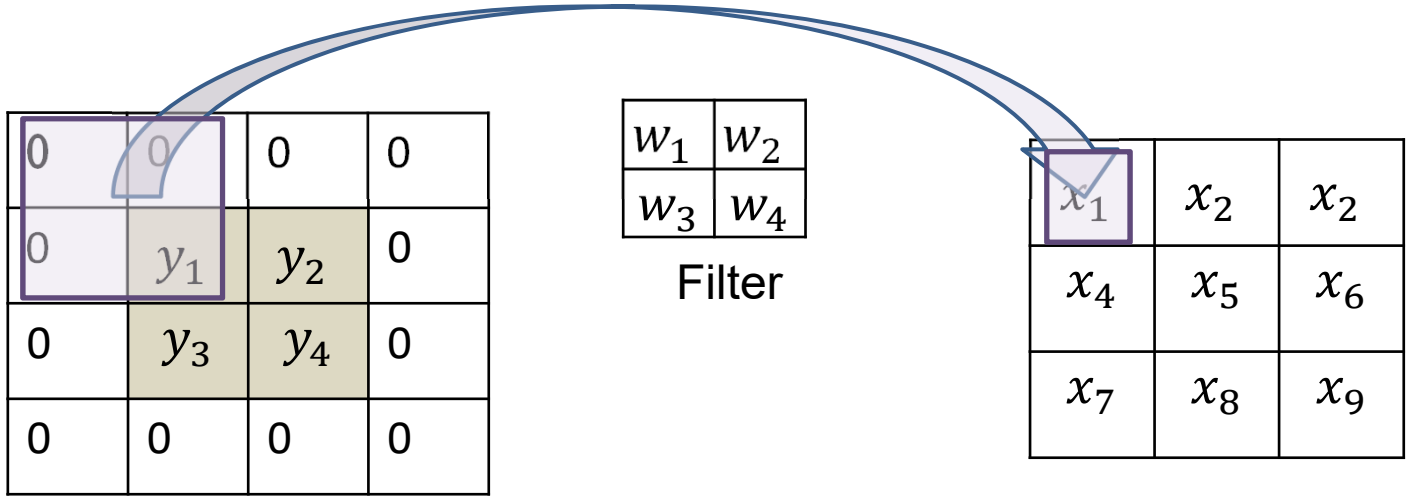
- Pooling
  - Used to aggregate information from a given layer
  - Usually Mean or Max operators are used for pooling
  - Example: Max pooling, stride 2



- Pooling provides some form of invariance to input deformations
- Pooling arithmetics

CNNs
Transpose convolution

- This is the reverse operation – to a convolution
  - Increases the input image size
    - Used for auto-encoders, object recognition, segmentation
  - Example: from 2x2 image to 3x3 image, 2x2 filter, Stride 1 with Padding



Note: more in (Dumoulin 2016), a guide to convolution arithmetic for Deep Learning

Transposed convolutions

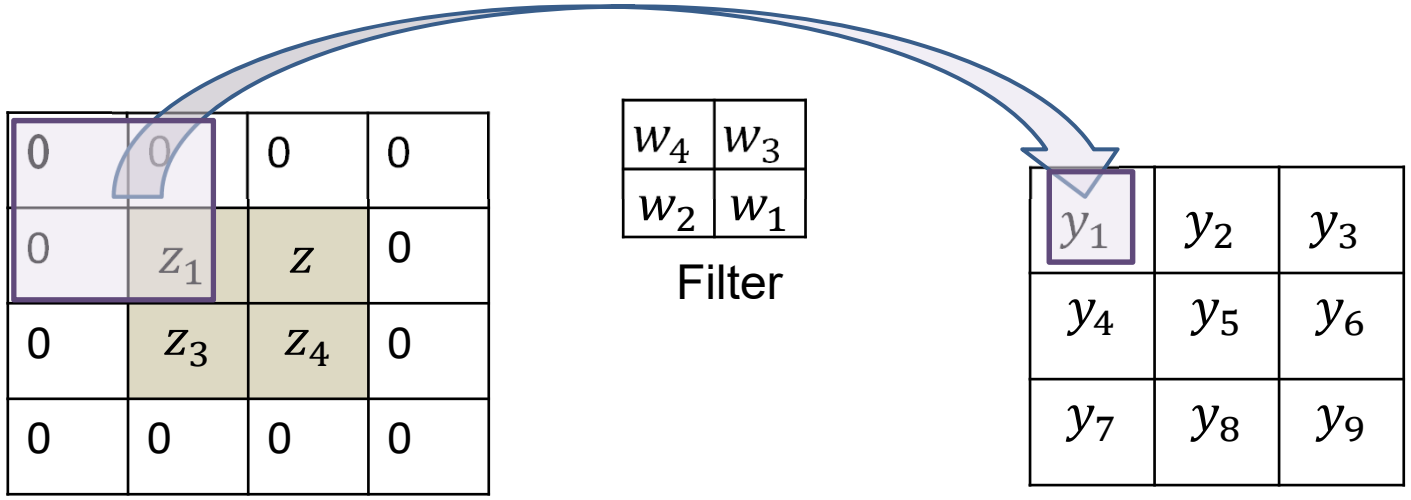
- Convolution
  - $x * w = z$, with $x \in R^9, z \in R^4$
  - $x = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix}, \mathrm{w} = \begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix}, \mathrm{z} = \begin{pmatrix} z_1 & z_2 \\ z_3 & z_4 \end{pmatrix}$

  - Convolution in matrix form
    - Lets flatten the vectors, the CNN convolution can be written in matrix form as:
    - $\mathrm{W}x = z$
    - $x = \begin{pmatrix} x_1 \\ \vdots \\ x_9 \end{pmatrix}, \mathrm{W} = \begin{pmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{pmatrix}, \mathrm{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix}$

- Transposed convolution
  - Transposed convolution in matrix form $y = W^T z, \quad z \in R^4 \; and \; y \in R^9$
  - $\mathrm{W}^{\mathrm{T}} = \begin{pmatrix} w_1 & 0 & 0 & 0 \\ w_2 & w_1 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ w_3 & 0 & w_1 & 0 \\ w_4 & w_3 & w_2 & w_1 \\ 0 & w_4 & 0 & w_2 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & w_4 & w_3 \\ 0 & 0 & 0 & w_4 \end{pmatrix}$

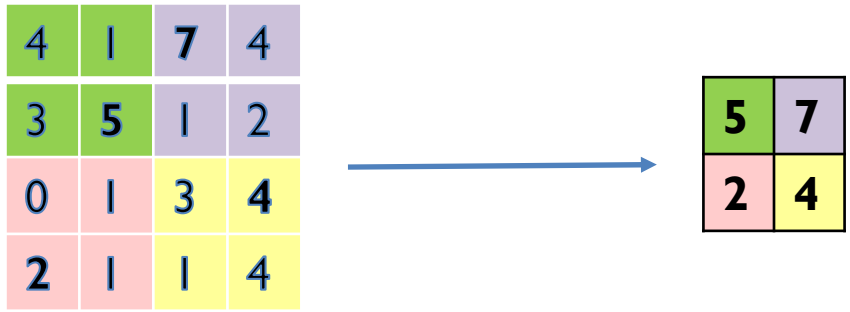
Transposed convoution

- Transposed convolution in convolutional form $y = z * w$

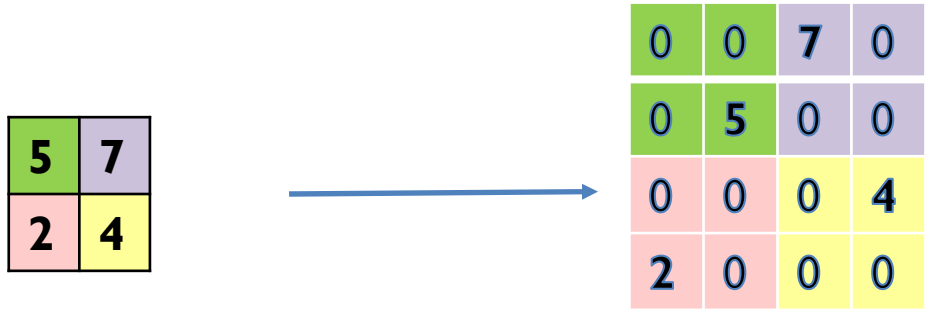


CNNs
Unpooling

- Reverse pooling operation
- Different solutions, e.g. unpooling a max pooling operation

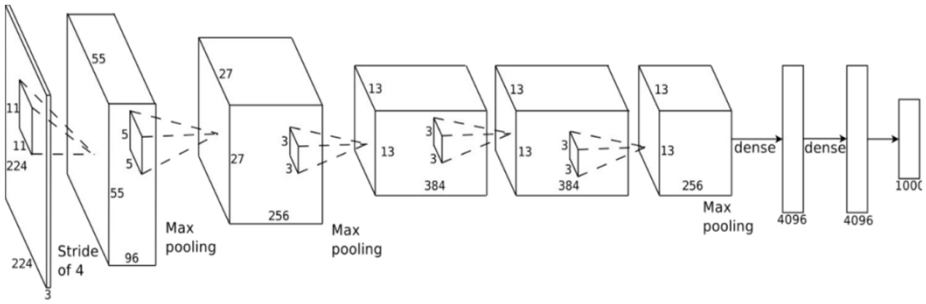


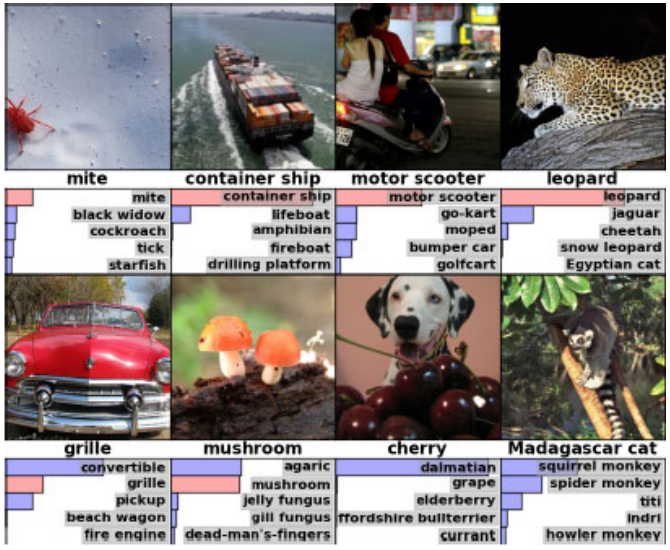
- Remember the positions of the max and fill the other positions with 0



CNNs–Classification (Krizhevsky et al. 2012)

- A landmark in object recognition - AlexNet
- ImageNet competition
  - Large Scale Visual Recognition Challenge (ILSVRC)
  - 1000 categories, 1.5 Million labeled training samples
  - Method: large convolutional net
  - 650K neurons, 630M synapses, 60M parameters
  - Trained with SGD on GPU



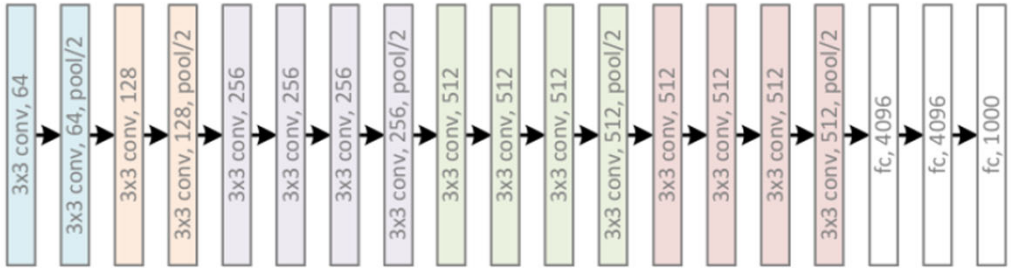


CNNs
Very Deep Nets trained with GPUs

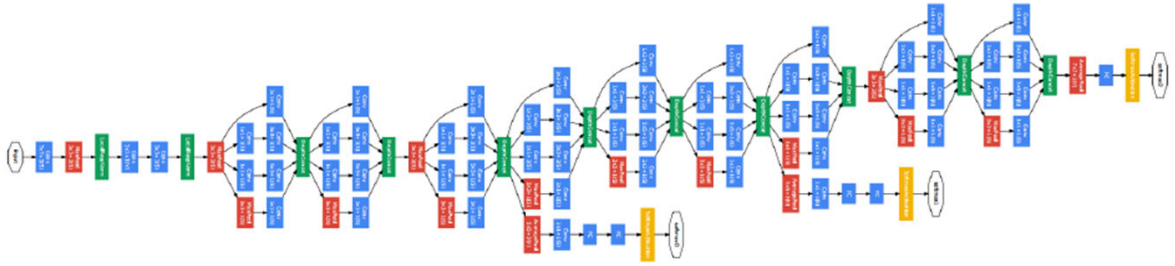
Deeper Nets with small filters – training time several days up to 1 or 2 weeks on ImageNet

Oxford, [Simonyan 2014], Parameters 138 M

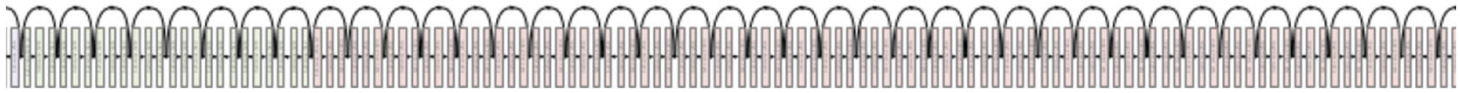
VGG, 16/19 layers, 2014



GoogleNet, 22 layers, 2014 Google, [Szegedy et al. 2015], Parameters 24 M



ResNet, 152 layers, 2015 MSRA, [He et al. 2016] , Parameters 60 M



Convolutional Nets
ILSVRC performance over the years

- Imagenet 2012 classification challenge

Rank	Name	Error rate	Description
1	**U. Toronto**	0.15315	Deep learning
2	U. Tokyo	0.26172	Hand-crafted features and learning models. Bottleneck.
3	U. Oxford	0.26979	
4	Xerox/INRIA	0.27058	

Object recognition over 1,000,000 images and 1,000 categories (2 GPU)

- ImageNet 2014 – Image classification challenge

Rank	Name	Error rate	Description
1	Google	0.06656	Deep learning
2	Oxford	0.07325	Deep learning
3	MSRA	0.08062	Deep learning

- ImageNet 2014 – object detection challenge

Rank	Name	Mean Average Precision	Description
1	Google	0.43933	Deep learning
2	CUHK	0.40656	Deep learning
3	DeepInsight	0.40452	Deep learning
4	UvA-Euvision	0.35421	Deep learning
5	Berkley Vision	0.34521	Deep learning

- ImageNet 2013 – image classification challenge

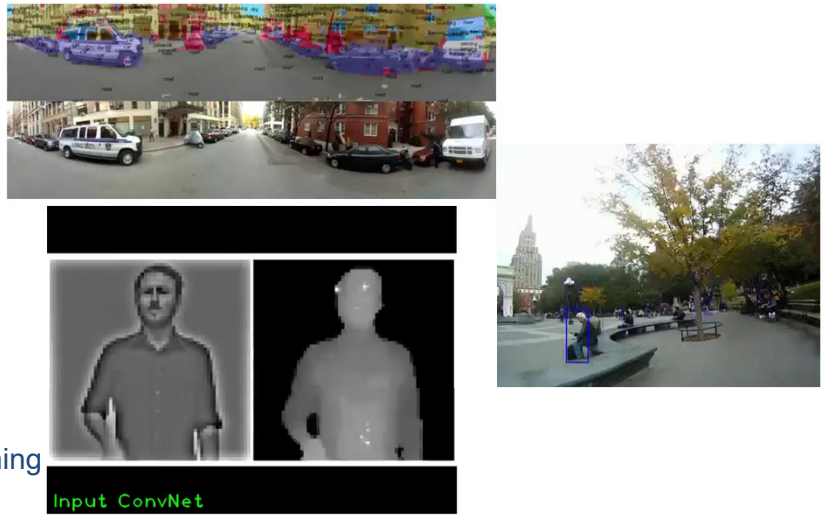
Rank	Name	Error rate	Description
1	NYU	0.11197	Deep learning
2	NUS	0.12535	Deep learning
3	Oxford	0.13555	Deep learning

MSRA, IBM, Adobe, NEC, Clarifai, Berkley, U. Tokyo, UCLA, UIUC, Toronto .... Top 20 groups all used deep learning

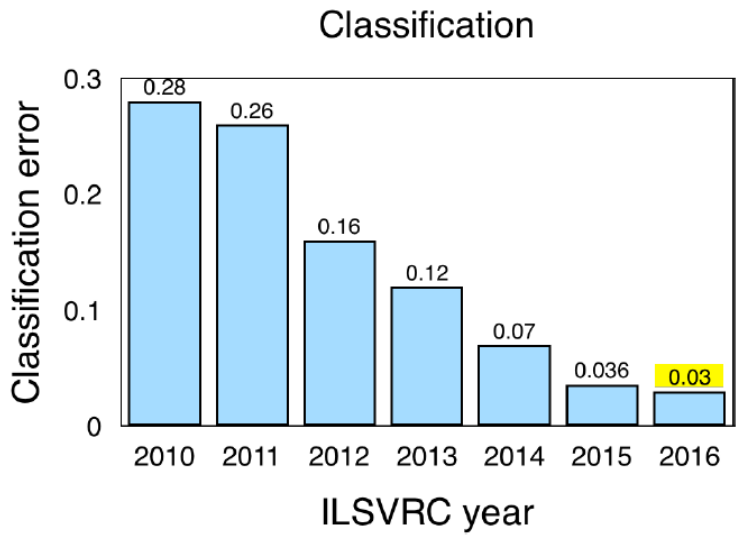
- ImageNet 2013 – object detection challenge

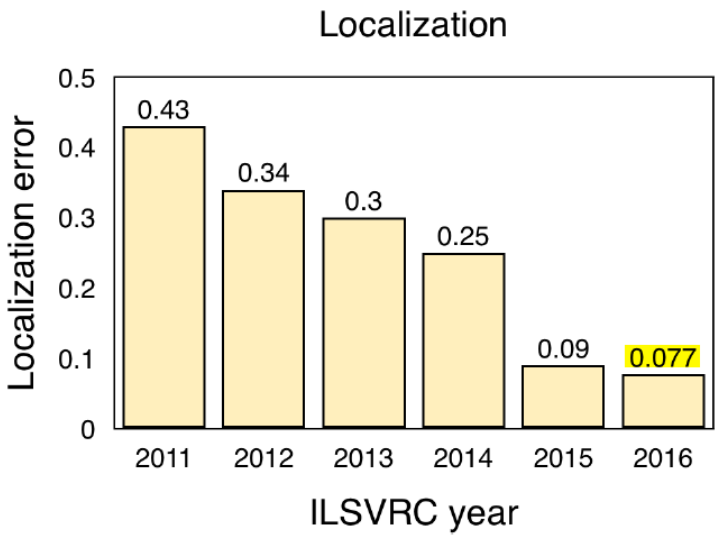
Rank	Name	Mean Average Precision	Description
1	UvA-Euvision	0.22581	Hand-crafted features
2	NEC-MU	0.20895	Hand-crafted features
3	NYU	0.19400	Deep learning

CNN examples



Convolutional Nets
ILSVRC performance over the years

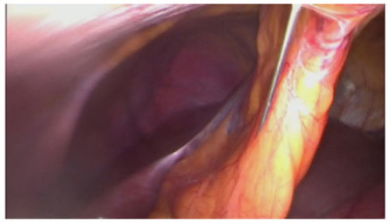


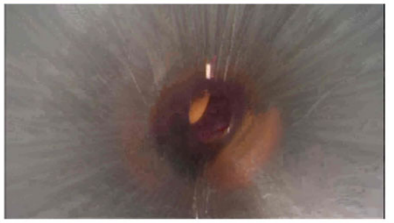


Classification
CNNs and Transfer Learning

- Training large NN requires
  - large amount of labeled data
  - Large GPU clusters
- Large labeled datasets are not available for all applications
- Deep Networks **pretrained** with large datasets like ImageNet are used for other applications after some retraining/ fine tuning:
  - Classification of images from different nature
  - Classification of objects in large size images
  - Object detection, Segmentation
  - Learning latent representations of images
- Remark
  - CNN trained on ImageNet have specific characteristics
    - e.g. input: 224x224 images, centered on the objects to be classified
    - How to adapt them to other collections?

Classification - Transfer learning - CNNs - Images from different nature,M2CAI Challenge (Cadene 2016)





- Endoscopic videos (large intestine)
  - resolution of 1920 x 1080, shot at 25 frame per second at the IRCAD research center in Strasbourg, France. 27 training videos ranging from 15mn to 1hour, 15 testing videos
- Used for: monitor surgeons, Trigger automatic actions
- Objective: classification, 1 of 8 classes for each frame
  - TrocarPlacement, Preparation, CalotTriangleDissection, ClippingCutting, GallbladderDissection, GallbladderPackaging, CleaningCoagulation, GallbladderRetraction
- Resnet 200 pretrained with ImageNet -> reaches 80% correct classification

Model	Input	Param.	Depth	Implem.	Forward (ms)	Backward (ms)
Vgg16	224	138M	16	GPU	185.29	437.89
InceptionV3 [2]	399	24M	42	GPU	**102.21**	311.94
ResNet-200 [3]	224	65M	200	GPU	273.85	687.48
InceptionV3	399	24M	42	CPU	19918.82	23010.15

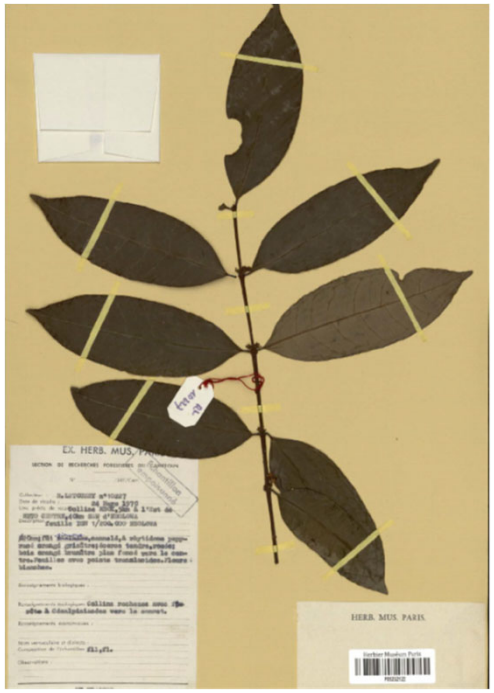
Table 1: Forward+Backward with batches of 20 images.

InceptionV3	Extraction (repres. of ImageNet)	60.53
InceptionV3	From Scratch (repres. of M2CAI)	69.13
InceptionV3	Fine-tuning (both representations)	79.06
ResNet200	**Fine-tuning (both representations)**	**79.24**

Table 2: Accuracy on the validation set.

Classification - Transfer learning - CNNs - Images from different nature, Plant classification (Wu 2017)

- Digitized plant collection from Museum of Natural History – Paris
- Largest digitized world collection (8 millions specimens)
- Goal
  - Identify plants characteristics for automatic labeling of worlwide plant collections
  - O(1000) classes, e.g. opposed/alternate leaves; simple/composed leaves; smooth/with teeth leaves, ….
- Pretrained ResNet



Classification - Fully convolutional nets

CNNs – Classification of large images (Durand 2016)

How to deal with complex scenes?

Pascal VOC style

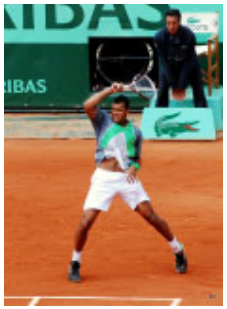
ImageNet style

- Working on datasets with complex scenes (large and cluttered background), not centered objects, variable size, ...

VOC07/ 12

MIT67

15 Scene



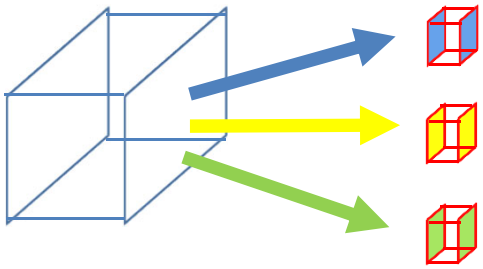
COCO

VOC12 Action

Converting Fully Convolutional Nets (FCN) to CNN

- Fully connected layers can be converted to convolutional nets
  - The following scheme is equivalent to 3 output cells fully connected to the input cells, but is expressed as a convolution
  - Colors correspondance below

FCN classical view

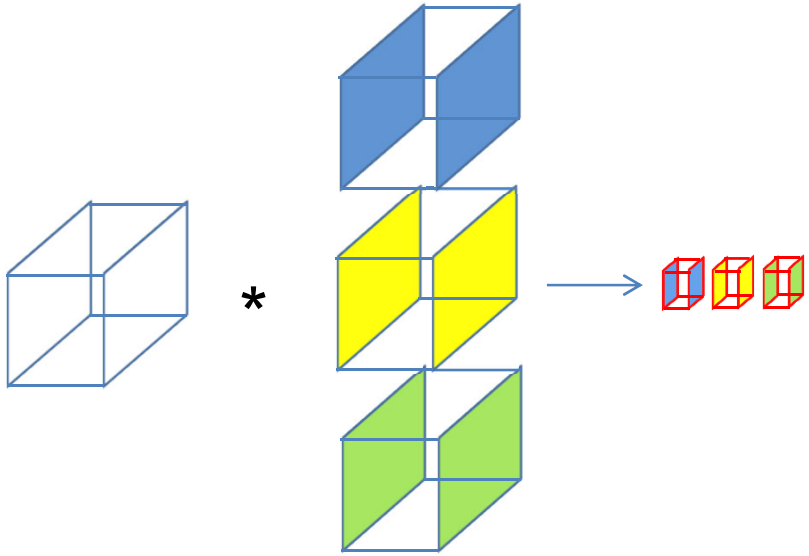


Cell layer
$nxnxC$

Each weight
vector is
$nxnxC$

Outputs
1 each

FCN convolutional view



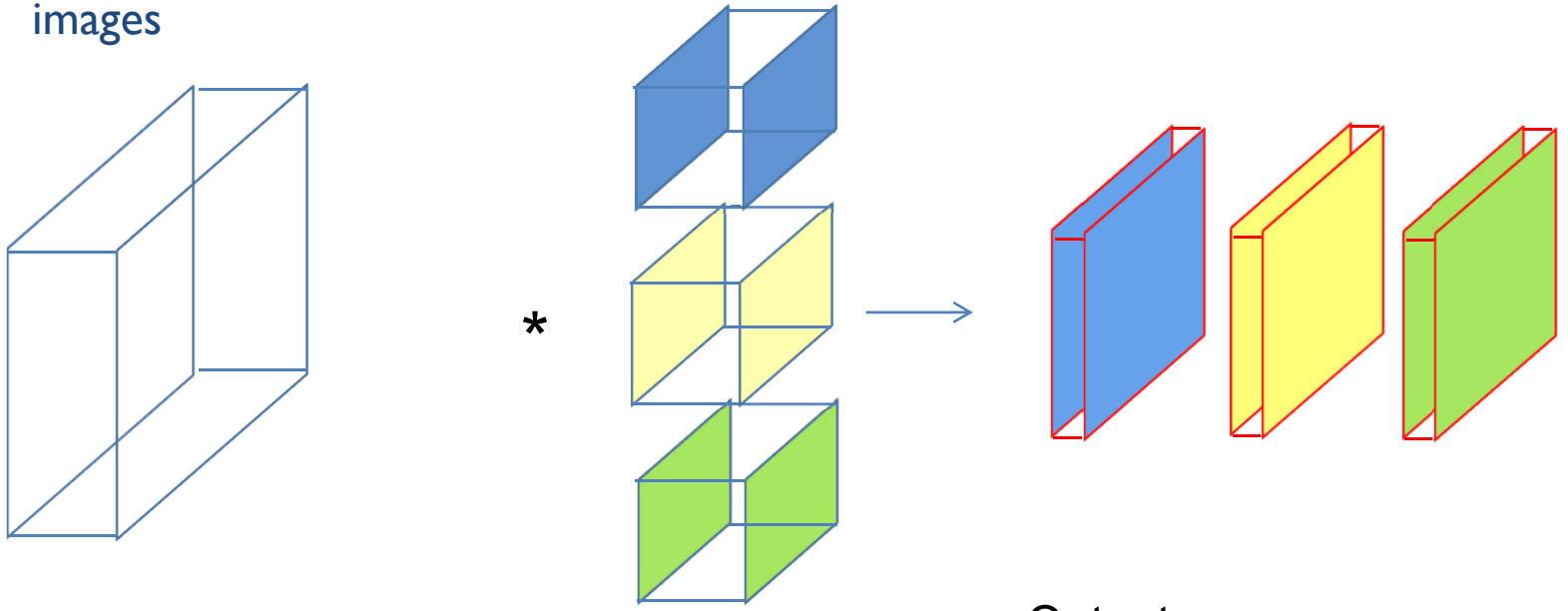
Cell layer
$nxnxC$

Weight layer
each is $nxnxC$

Outputs
1 each

Converting Fully Convolutional Nets (FCN) to CNN

- Fully connected layers can be converted to convolutional nets
  - This does not change anything if the input size is the size of the weight layer
  - It can be used as a convolution for larger input sizes, and then produces larger outputs
  - In this way, pre-trained networks can be used without retraining for larger images

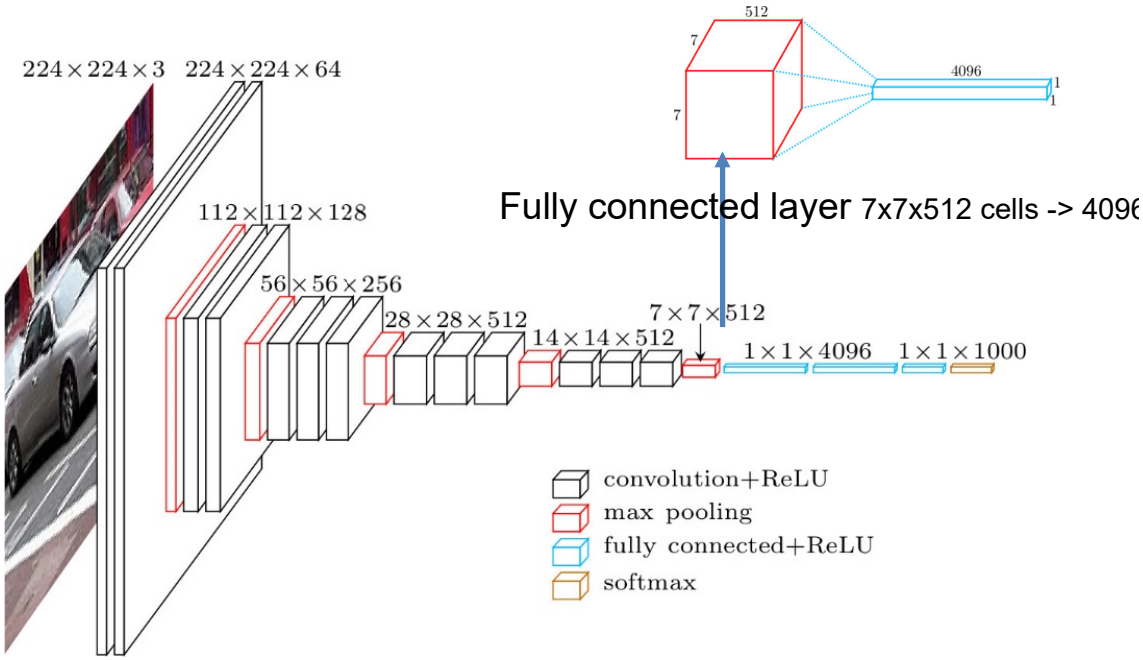


Cell layer
NxNxC

Weight layer
nxnxC each

Outputs
$(N-n+1)$x$(N-n+1)$x1 each

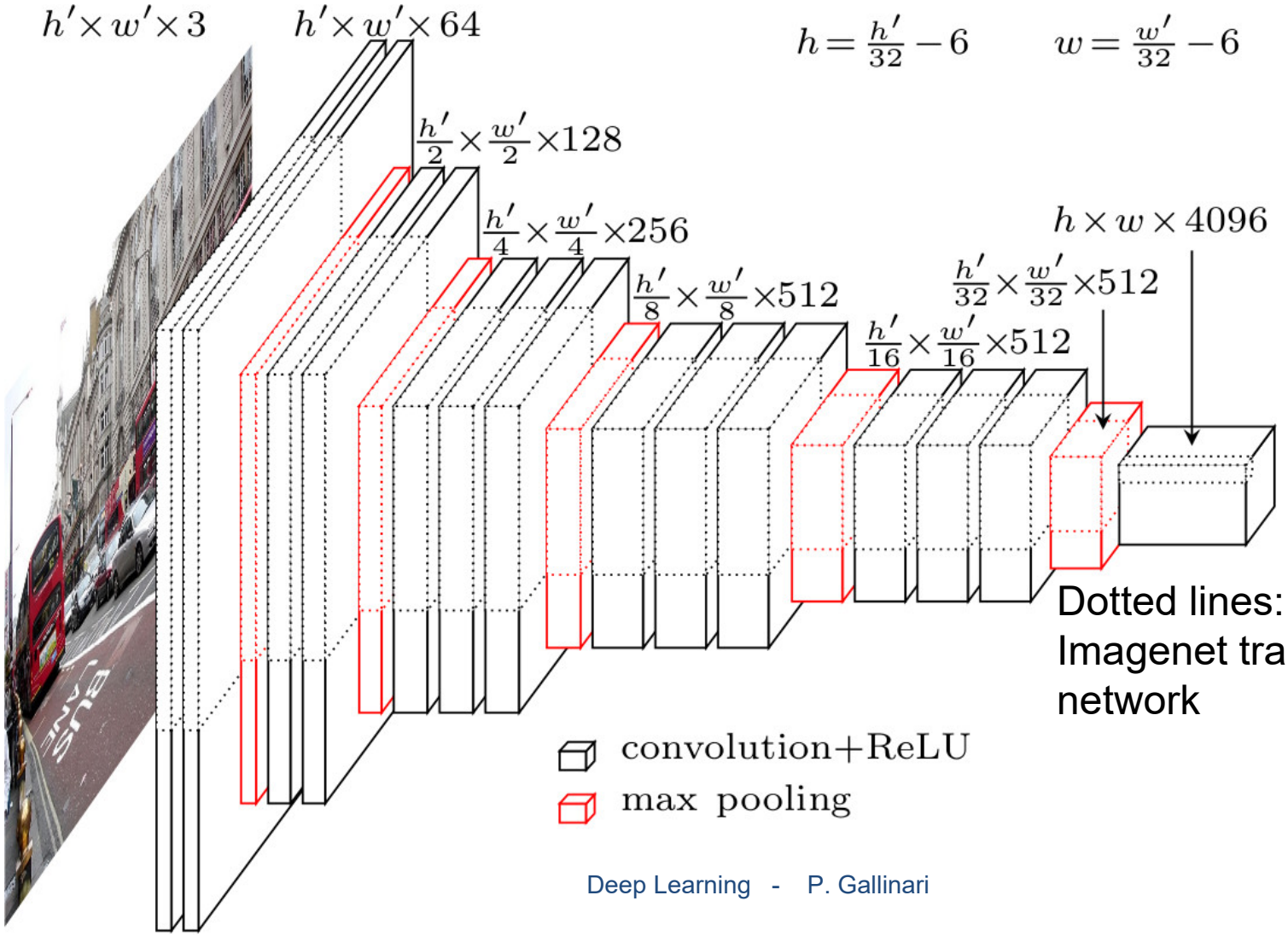
Classification - CNNs – Classification of large images (Durand 2016)
Sliding window => Convolutional Layers



ImageNet trained CNN

- Sliding window:
  - Use the ImageNet trained CNN as a sliding window (a convolution filter) on the large image
  - In order to do that, one must **convert the fully connected layer 7x7x512 cells → 4096 cells to a convolutional layer**

CNNs – Classification of large images (Durand 2016)
Sliding window => Convolutional Layers



Dotted lines: initial Imagenet trained network

CNNs – Classification of large images (Sermanet et al. 2014)
Sliding window => Convolutional Layers

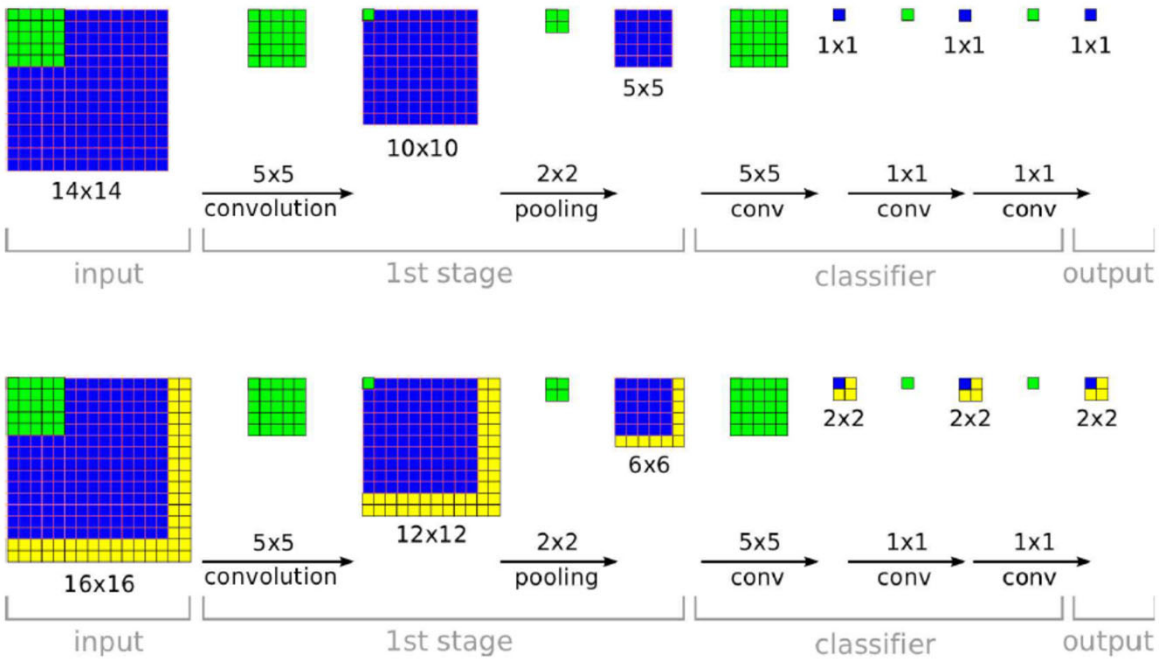
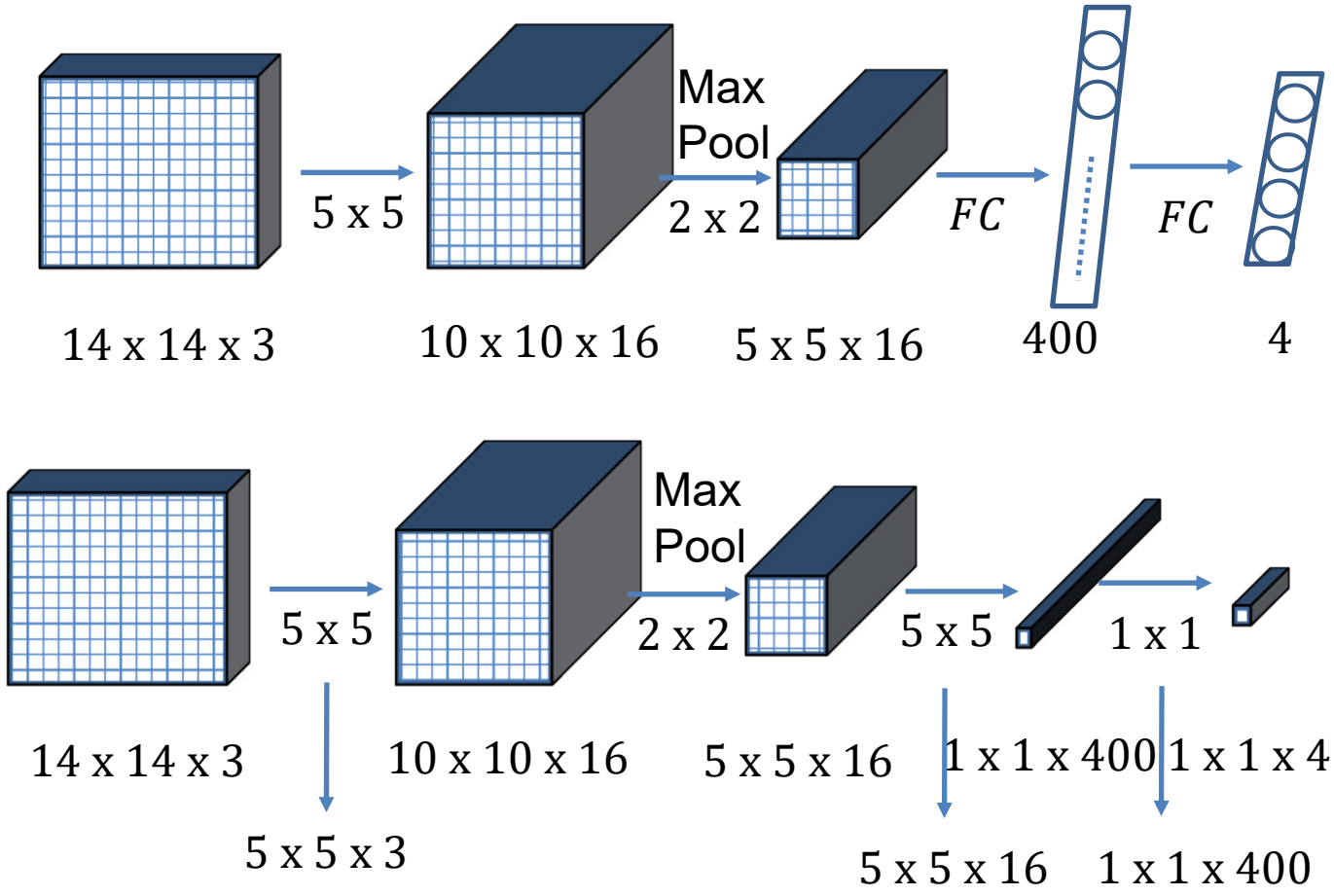


Figure 5: **The efficiency of ConvNets for detection.** During training, a ConvNet produces only a single spatial output (top). But when applied at test time over a larger image, it produces a spatial output map, e.g. 2x2 (bottom). Since all layers are applied convolutionally, the extra computation required for the larger image is limited to the yellow regions. This diagram omits the feature dimension for simplicity.

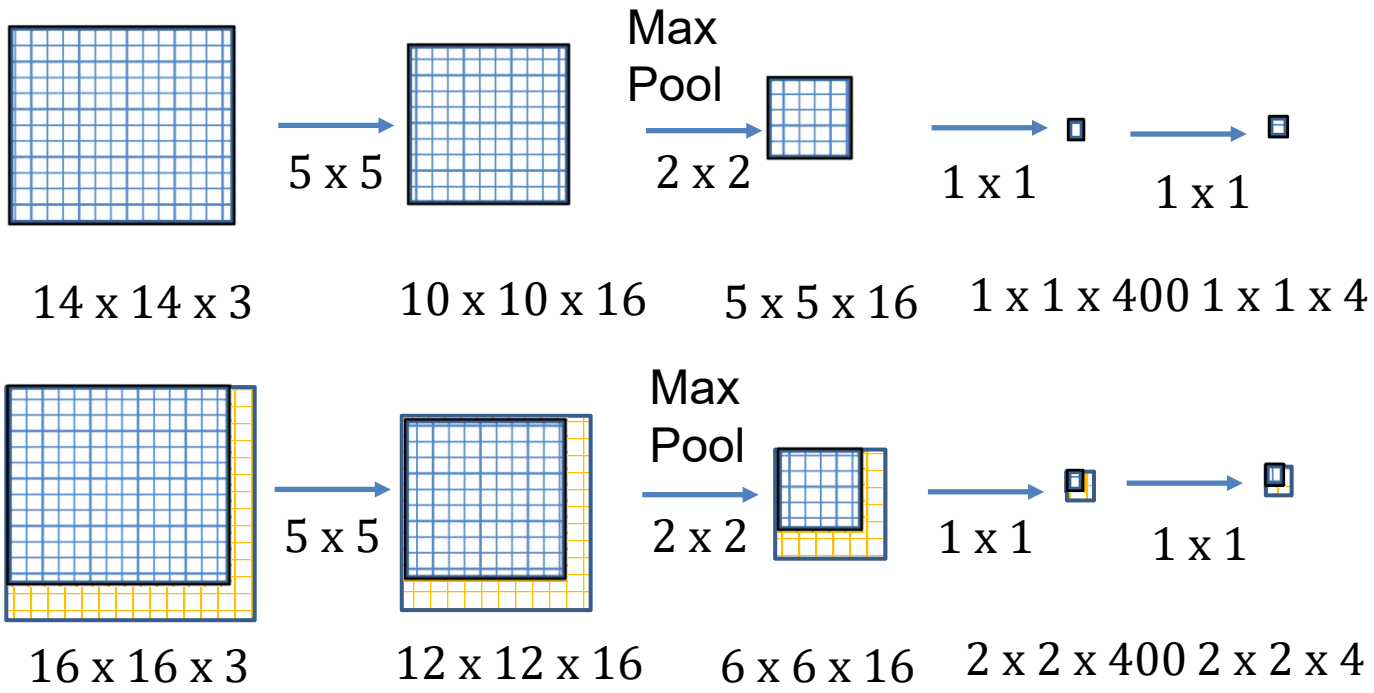
Fig: Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, Yann LeCun, OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks, 2014

From full connexion (FC) to convolutional layers



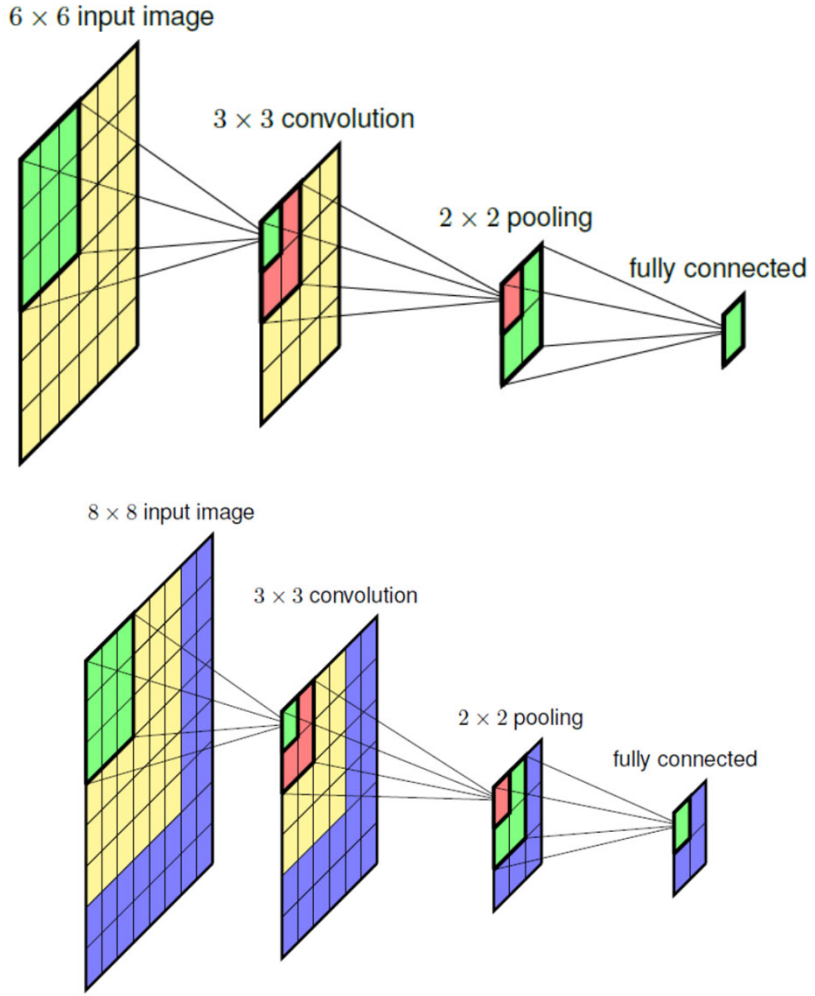
Sliding windows

To simplify, we drop the 3D channels



Convolutionnal sliding windows allows us to compute the 4 classes output for all the sliding windows positions (4 here) without redondant computations as a naive implementation of the sliding window would do.

Sliding windows (followed)



Application of the network shown in Figure 10.22 to a larger image in which the additional computation required corresponds to the blue regions.

Fig. Bishop 2024

CNNs – Classification of large images (Durand 2016)
Sliding window => Convolutional Layers