MLCC 2018
Deep Learning

Lorenzo Rosasco
UNIGE-MIT-IIT

What? Classification

Object classification

What's in this image?

Note: beyond vision: classify graphs, strings, networks, time-series...

What makes the problem hard?

- Viewpoint

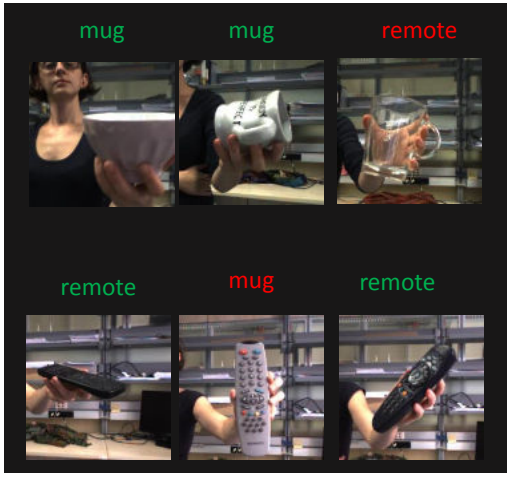
- Semantic variability

Note: Identification vs categorization...

Categorization: a learning approach

Training

Test



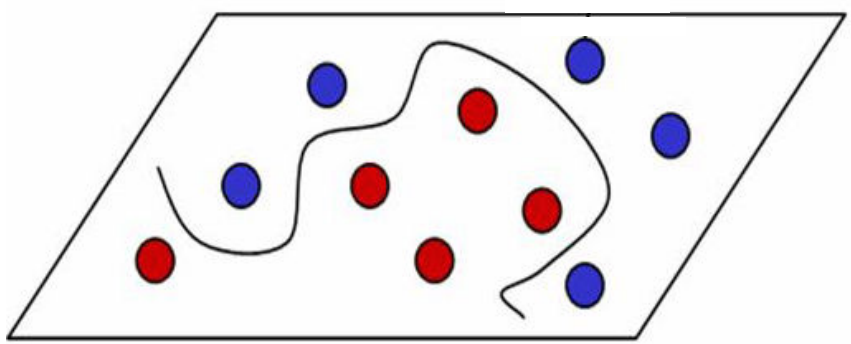
Supervised learning

Given

$$(x_1, y_1), \ldots, (x_n, y_n)$$

find f such that

$$\text{sign} f(x_{\text{new}}) = y_{\text{new}}$$



- $x \in \mathbb{R}^D$ a vectorization of an image
- $y = \pm 1$ a label (mug/remote)

Learning and data representation

Consider

$$f(x) = w^\top \Phi(x)$$

a two steps learning scheme is often considered

- *supervised* learning of w
- expert design or *unsupervised* learning of the **data representation** Φ

Learning and data representation

Consider

$$f(x) = w^\top \Phi(x)$$

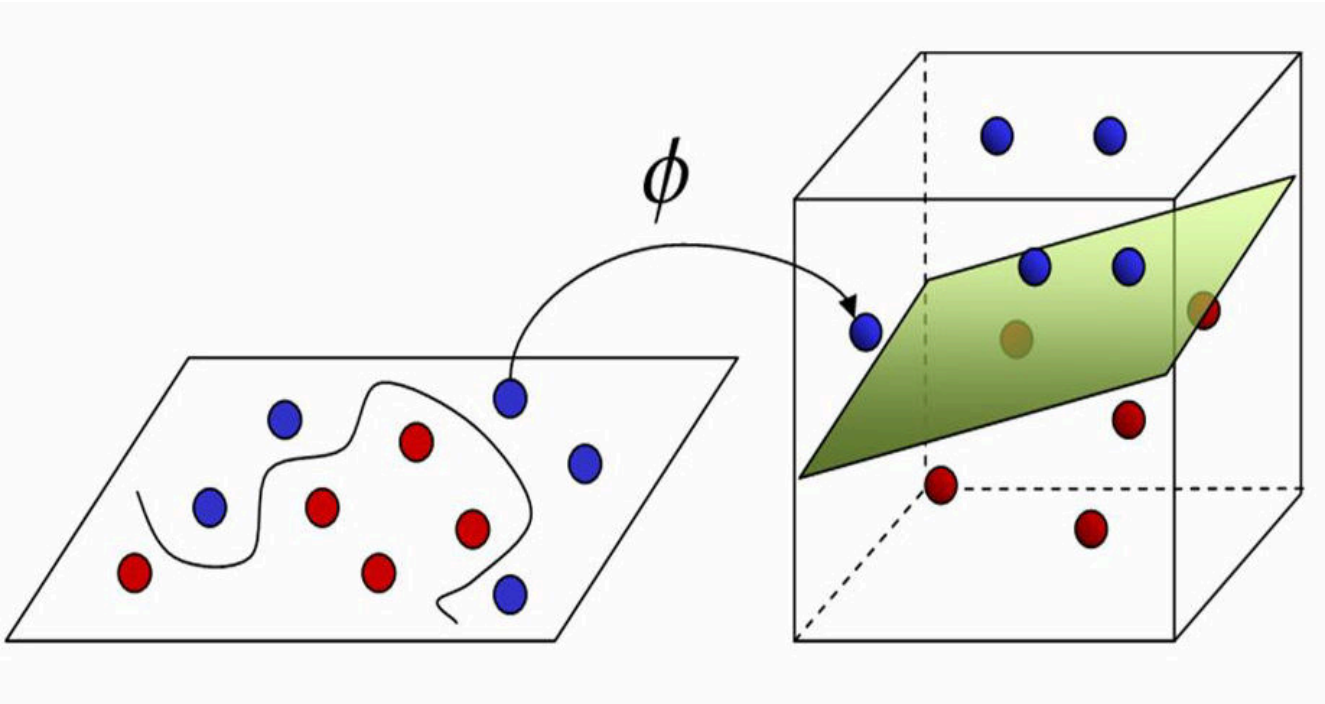
a two steps learning scheme is often considered

- *supervised* learning of w
- expert design or *unsupervised* learning of the **data representation** Φ

Data representation

$$\Phi : \mathbb{R}^D \to \mathbb{R}^p$$

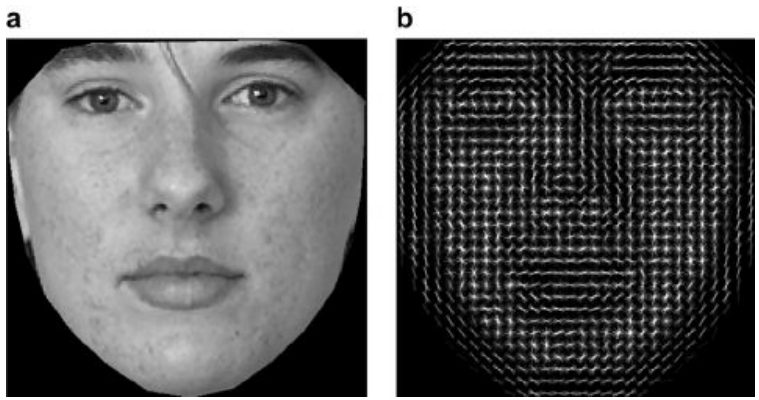
A mapping of data in a new **format** better suited for further processing



Data representation by design

Dictionaries of features

- Wavelet & friends.
- SIFT, HoG etc.



Kernels

- Classic off the shelf: Gaussian $K(x, x') = e^{-\|x-x'\|^2\gamma}$
- Structured input: kernels on histograms, graphs etc.

In practice all is multi-layer!
(an old slide)

Data representation schemes e.g. vision-speech, involve **multiple** (*layers*).

Pipeline

Raw data are often processed:

- first computing some of **low level** features,
- then learning some **mid level** representation,
- ...
- finally using **supervised** learning.

These stages are often done separately:

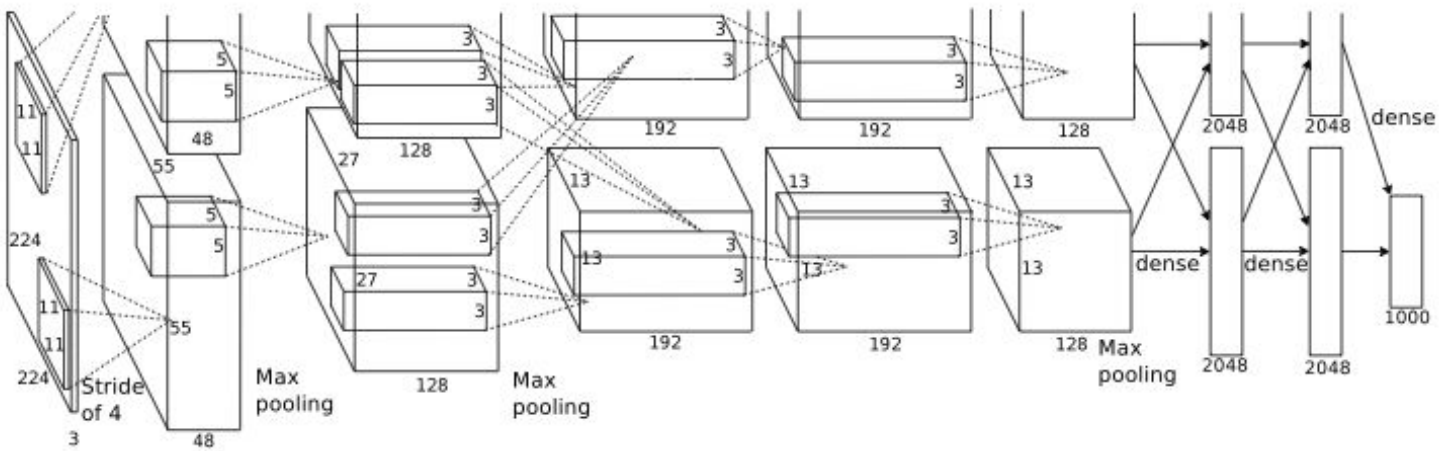
- good way to exploit unlabelled data...
- but is it possible to design **end-to-end** learning systems?

In practice all is deep-learning!
(updated slide)

Data representation schemes e.g. vision-speech, involve **deep learning**.

Pipeline

- Design some **wild- but "differentiable"** hierarchical architecture.
- Proceed with **end-to-end** learning!!



Architecture (rather than feature) engineering

Road Map

Part I: Basics neural networks

- Neural networks definition
- Optimization +approximation and statistics

Part II: One step beyond

- Auto-encoders
- Convolutional neural networks
- Tips and tricks

Part I: Basic Neural Networks

Shallow nets

$$f(x) = w^\top \Phi(x), \quad \underbrace{x \mapsto \Phi(x)}_{\textbf{Fixed}}$$

.

Shallow nets

$$f(x) = w^\top \Phi(x), \quad \underbrace{x \mapsto \Phi(x)}_{\textbf{Fixed}}$$

.

Examples

- Dictionaries

$$\Phi(x) = \cos(B^\top x) = (\cos(\beta_1^\top x), \ldots, \cos(\beta_p^\top x))$$

with $B = \beta_1, \ldots, \beta_p$ fixed frequencies.

Shallow nets

$$f(x) = w^\top \Phi(x), \quad \underbrace{x \mapsto \Phi(x)}_{\textbf{Fixed}}$$

.

Examples

- Dictionaries

$$\Phi(x) = \cos(B^\top x) = (\cos(\beta_1^\top x), \ldots, \cos(\beta_p^\top x))$$

with $B = \beta_1, \ldots, \beta_p$ fixed frequencies.

- Kernel methods

$$\Phi(x) = (e^{-\|\beta_1 - x\|^2}, \ldots, e^{-\|\beta_n - x\|^2})$$

with $\beta_1 = x_1, \ldots, \beta_n = x_n$ the input points.

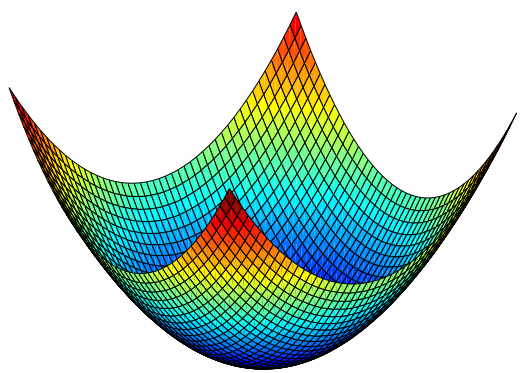
Shallow nets (cont.)

$$f(x) = w^\top \Phi(x), \quad \underbrace{x \mapsto \Phi(x)}_{\textbf{Fixed}}$$

Empirical Risk Minimization (ERM)

$$\min_w \sum_{i=1}^{n} (y_i - w^\top \Phi(x_i))^2$$

Note:
The function f depends linearly on w, the ERM problem is **convex**!



Interlude: optimization by Gradient Descent (GD)

Batch gradient descent

$$w_{t+1} = w_t - \gamma \nabla_w \widehat{\mathcal{E}}(w_t)$$

where

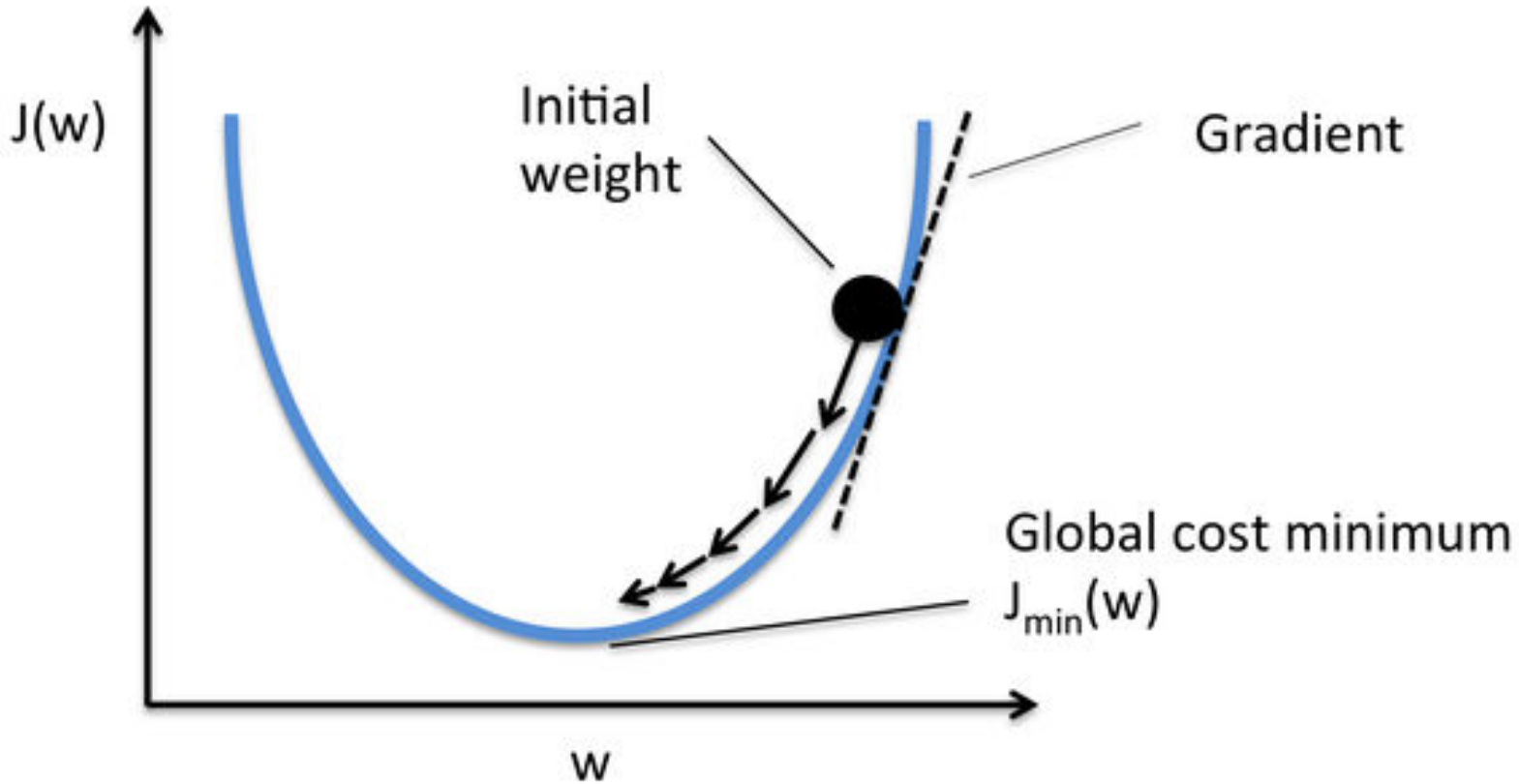
$$\widehat{\mathcal{E}}(w) = \sum_{i=1}^{n} (y_i - w^\top \Phi(x_i))^2$$

so that

$$\nabla_w \widehat{\mathcal{E}}(w) = -2 \sum_{i=1}^{n} \Phi(x_i)^\top (y_i - w^\top \Phi(x_i))$$

- **Constant step-size** depending on the *curvature* (Hessian norm)
- It is a **descent** method

Gradient descent illustrated



Stochastic gradient descent (SGD)

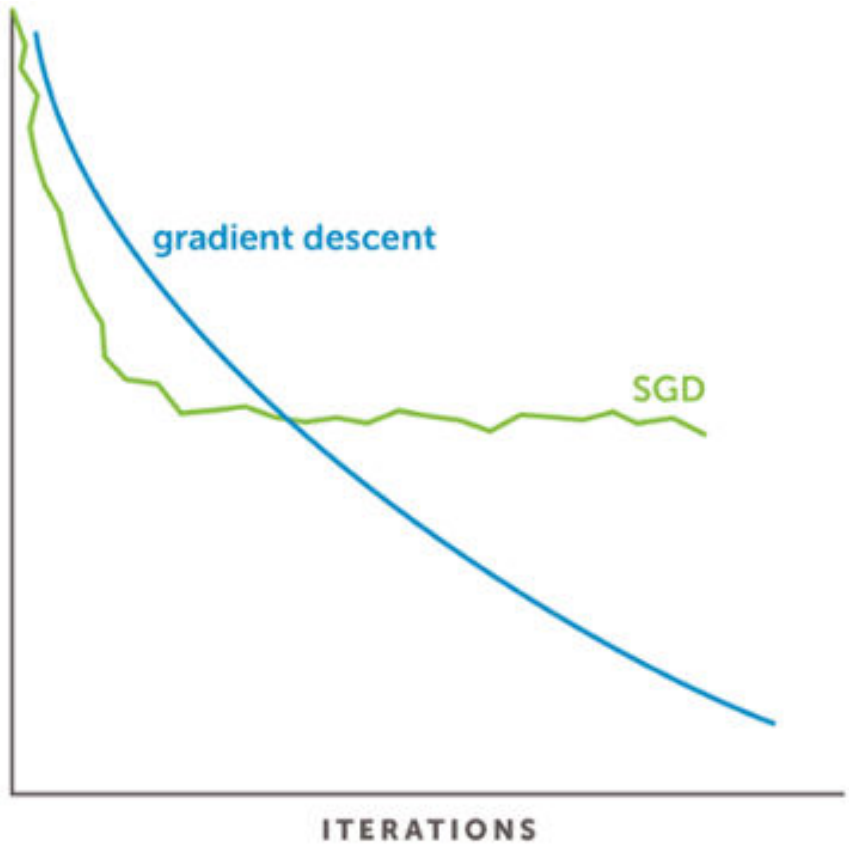
$$w_{t+1} = w_t + 2\gamma_t \Phi(x_t)^\top (y_t - w_t^\top \Phi(x_t))$$

Compare to

$$w_{t+1} = w_t + 2\gamma \sum_{i=1}^{n} \Phi(x_i)^\top (y_i - w_t^\top \Phi(x_i))$$

- Decaying step-size $\gamma = 1/\sqrt{t}$
- Lower **iteration cost**
- It is not a **descent** method (SG**D**?)
- Multiple passes (**epochs**) over data needed

SGD vs GD



Summary so far

Given data $(x_1, y_1), \ldots, (x_n, y_n)$ and a fixed representation Φ

- Consider
$$f(x) = w^\top \Phi(x)$$
- Find w by SGD
$$w_{t+1} = w_t + 2\gamma_t \Phi(x_t)^\top (y_t - w^\top \Phi(x_t))$$

Can we jointly learn Φ?

Neural Nets

Basic idea: **compose** simply **parameterized** representations

$$\Phi = \Phi_L \circ \cdots \circ \Phi_2 \circ \Phi_1$$

Neural Nets

Basic idea: **compose** simply **parameterized** representations

$$\Phi = \Phi_L \circ \cdots \circ \Phi_2 \circ \Phi_1$$

Let $d_0 = D$ and

$$\Phi_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

Neural Nets

Basic idea: **compose** simply **parameterized** representations

$$\Phi = \Phi_L \circ \cdots \circ \Phi_2 \circ \Phi_1$$

Let $d_0 = D$ and

$$\Phi_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

and in particular

$$\Phi_\ell = \sigma \circ W_\ell, \quad \ell = 1, \ldots, L$$

Neural Nets

Basic idea: **compose** simply **parameterized** representations

$$\Phi = \Phi_L \circ \cdots \circ \Phi_2 \circ \Phi_1$$

Let $d_0 = D$ and

$$\Phi_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

and in particular

$$\Phi_\ell = \sigma \circ W_\ell, \quad \ell = 1, \ldots, L$$

where

$$W_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

linear/affine

Neural Nets

Basic idea: **compose** simply **parameterized** representations

$$\Phi = \Phi_L \circ \cdots \circ \Phi_2 \circ \Phi_1$$

Let $d_0 = D$ and

$$\Phi_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

and in particular

$$\Phi_\ell = \sigma \circ W_\ell, \quad \ell = 1, \ldots, L$$

where

$$W_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \quad \ell = 1, \ldots, L$$

linear/affine and σ is a non linear map acting component-wise

$$\sigma : \mathbb{R} \to \mathbb{R}.$$

Deep neural nets

$$f(x) = w^\top \Phi_L(x), \qquad \underbrace{\Phi_L = \overline{\Phi}_L \circ \cdots \circ \overline{\Phi}_1}_{\text{compositional representation}}$$

$$\overline{\Phi}_1 = \sigma \circ W_1 \quad \ldots \quad \overline{\Phi}_L = \sigma \circ W_L$$

Deep neural nets

$$f(x) = w^\top \Phi_L(x), \qquad \underbrace{\Phi_L = \overline{\Phi}_L \circ \cdots \circ \overline{\Phi}_1}_{\text{compositional representation}}$$

$$\overline{\Phi}_1 = \sigma \circ W_1 \quad \dots \quad \overline{\Phi}_L = \sigma \circ W_L$$

ERM

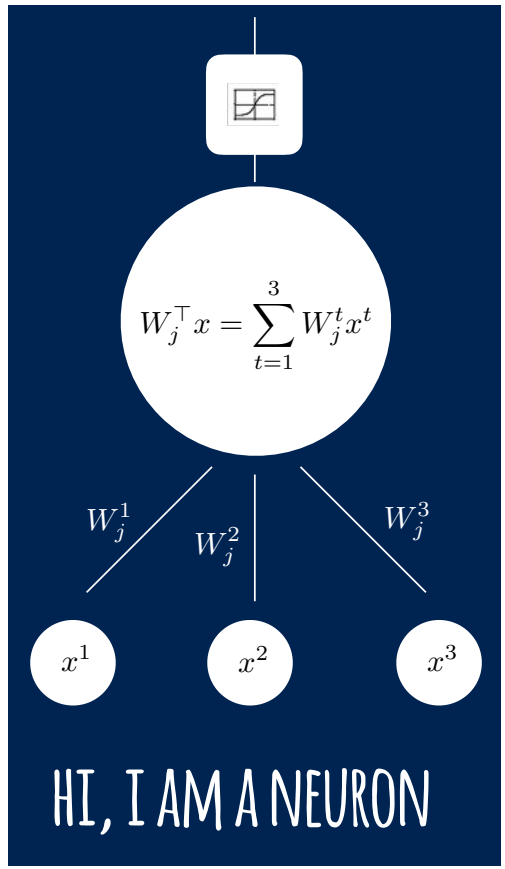
$$\min_{w,(W_j)_j} \frac{1}{n} \sum_{i=1}^{n} (y_i - w^\top \Phi_L(x_i))^2$$

Neural networks jargoon

$$\Phi_L(x) = \sigma(W_L \dots \sigma(W_2 \sigma(W_1 x)))$$

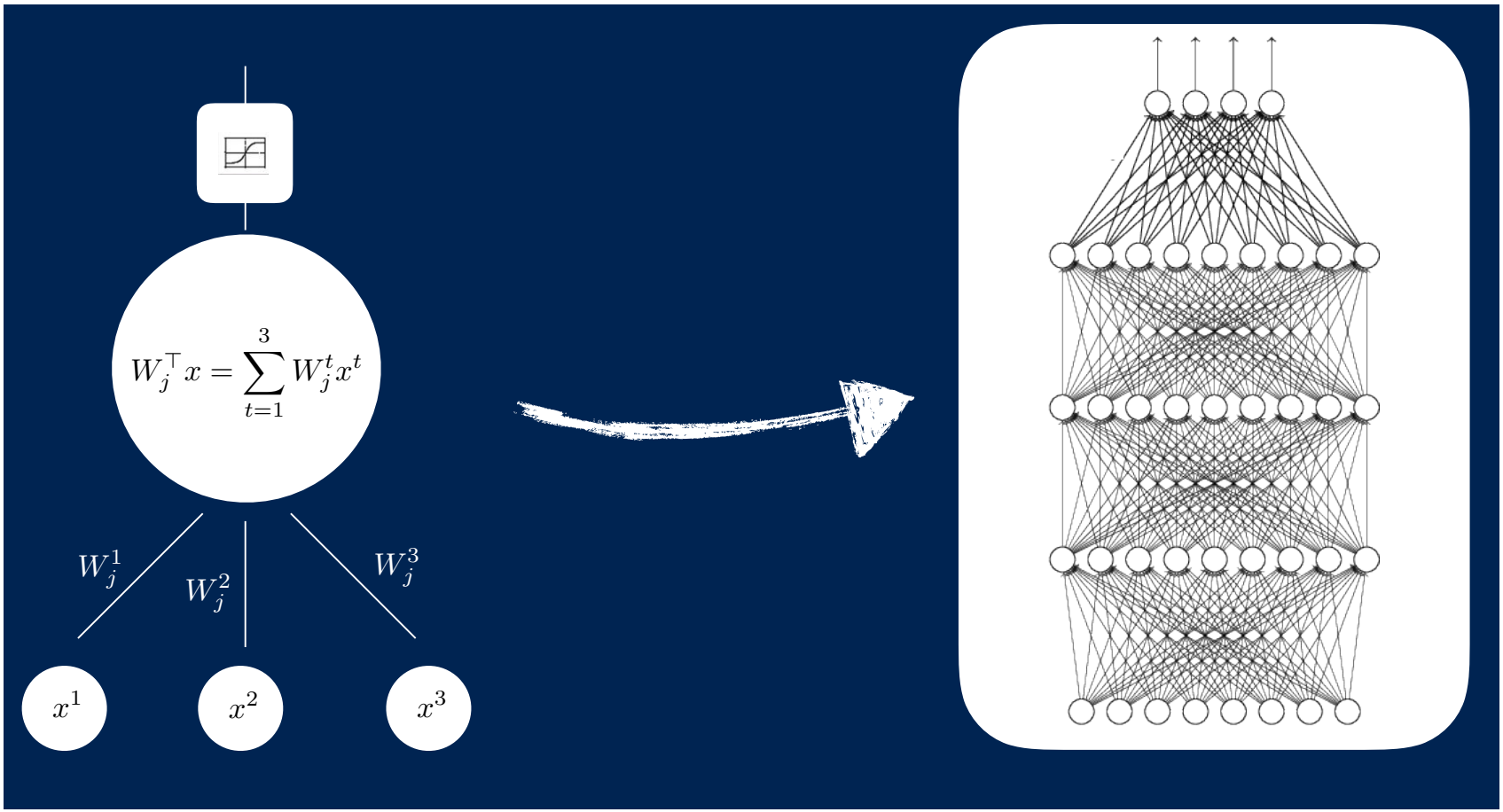
- Each intermediate representation corresponds to a **(hidden) layer**
- The dimensionalities $(d_\ell)_\ell$ correspond to the number of **hidden units**
- The non linearity σ is called **activation function**

Neural networks & neurons



- Each neuron compute an **inner product** based on a column of a weight matrix W
- The non-linearity σ is the **neuron activation** function.

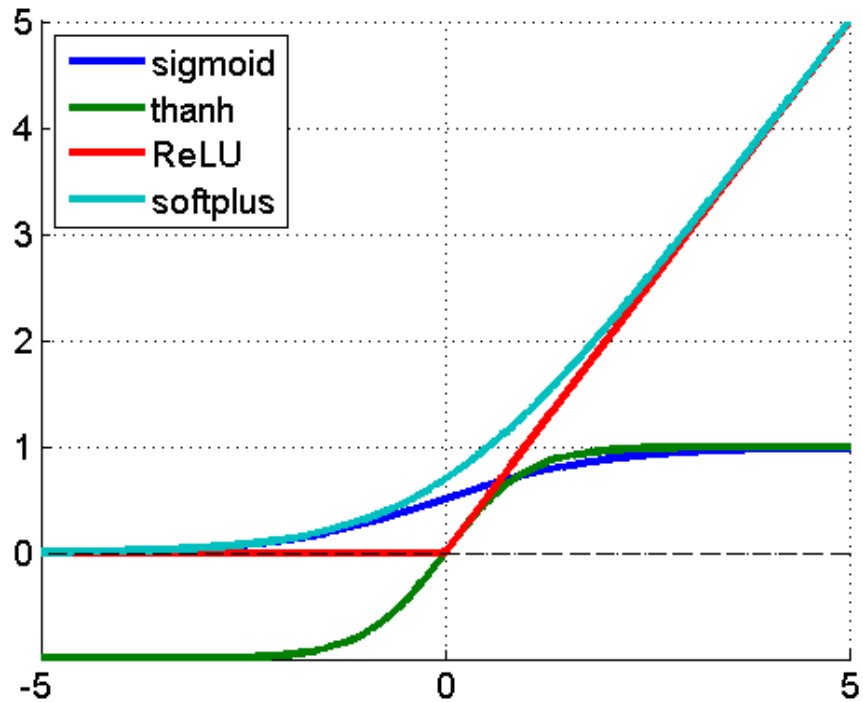
Deep neural networks



Activation functions

For $\alpha \in \mathbb{R}$ consider,

- **sigmoid** $s(\alpha) = 1/(1+e^{-\alpha})t$,
- **hyperbolic tangent** $s(\alpha) = (e^{\alpha} - e^{-\alpha})/(e^{\alpha} + e^{-\alpha})$,
- **ReLU** $s(\alpha) = |\alpha|_+$ (aka ramp, hinge),
- **Softplus** $s(\alpha) = \log(1+e^{\alpha})$.



Some questions

$$f_{w,(W_\ell)_\ell}(x) = w^\top \Phi_{(W_\ell)_\ell}(x), \qquad \Phi_{(W_\ell)_\ell} = \sigma(W_L \dots \sigma(W_2 \sigma(W_1 x)))$$

We have our model but:

- **Optimization:** Can we **train** efficiently?
- **Approximation:** Are we dealing with **rich** models?
- **Statistics:** How hard is it generalize from **finite data**?

Neural networks function spaces

Consider the non linear space of functions of the form
$f_{w,(W_\ell)_\ell} : \mathbb{R}^D \to \mathbb{R}$,

$$f_{w,(W_\ell)_\ell}(x) = w^\top \Phi_{(W_\ell)_\ell}(x), \qquad \Phi_{(W_\ell)_\ell} = \sigma(W_L \dots \sigma(W_2 \sigma(W_1 x)))$$

where $w, (W_\ell)_\ell$ may vary.

Very little structure. . . but we can :

- train by **gradient descent** (next)
- get (some) **approximation/statistical** guarantees (later)

One layer neural networks

Consider only one hidden layer:

$$f_{w,W}(x) = w^\top \sigma(Wx) = \sum_{j=1}^{u} w_j \sigma\left(x^\top W^j\right)$$

and ERM again

$$\sum_{i=1}^{n} (y_i - f_{w,W}(x_i))^2,$$

Computations

Consider

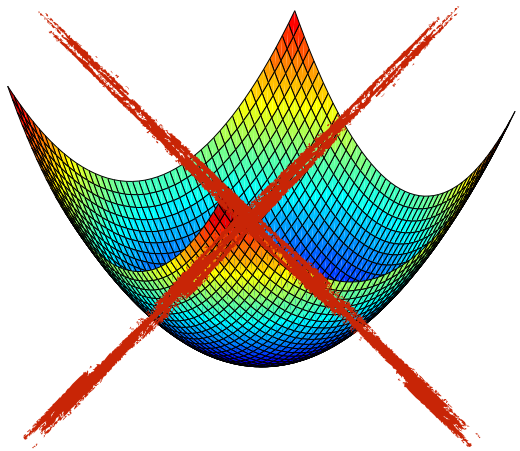
$$\min_{w,W} \widehat{\mathcal{E}}(w,W), \qquad \widehat{\mathcal{E}}(w,W) = \sum_{i=1}^{n} (y_i - f_{(w,W)}(x_i)))^2.$$

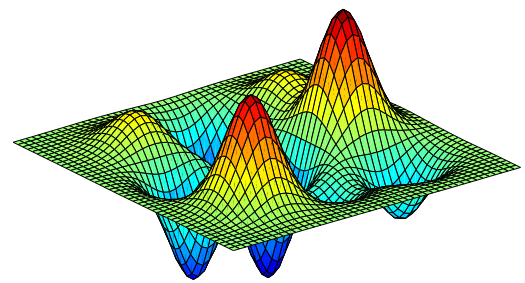
Computations

Consider

$$\min_{w,W} \widehat{\mathcal{E}}(w,W), \qquad \widehat{\mathcal{E}}(w,W) = \sum_{i=1}^{n} (y_i - f_{(w,W)}(x_i)))^2.$$

Problem is non-convex! (possibly smooth depending on σ)





Back-propagation & GD

Empirical risk minimization,

$$\min_{w,W} \widehat{\mathcal{E}}(w,W), \qquad \widehat{\mathcal{E}}(w,W) = \sum_{i=1}^{n} (y_i - f_{(w,W)}(x_i)))^2.$$

An approximate minimizer is computed via the following **gradient** method

$$\begin{aligned} w_j^{t+1} &= w_j^t - \gamma_t \frac{\partial \widehat{\mathcal{E}}}{\partial w_j}(w^t, W^t) \\ W_{j,k}^{t+1} &= W_{j,k}^t - \gamma_t \frac{\partial \widehat{\mathcal{E}}}{\partial W_{j,k}}(w^{t+1}, W^t) \end{aligned}$$

where the step-size $(\gamma_t)_t$ is often called learning rate.

Back-propagation & chain rule

Direct computations show that:

$$\frac{\partial \widehat{\mathcal{E}}}{\partial w_j}(w, W) = -2 \sum_{i=1}^{n} \underbrace{(y_i - f_{(w,W)}(x_i))}_{\Delta_{j,i}} h_{j,i}$$

$$\frac{\partial \widehat{\mathcal{E}}}{\partial W_{j,k}}(w, W) = -2 \sum_{i=1}^{n} \underbrace{(y_i - f_{(w,W)}(x_i)) w_j \sigma'(w_j^\top x)}_{\eta_{i,k}} x_i^k$$

Back-prop equations: $\eta_{i,k} = \Delta_{j,i} c_j \sigma'(w_j^\top x)$

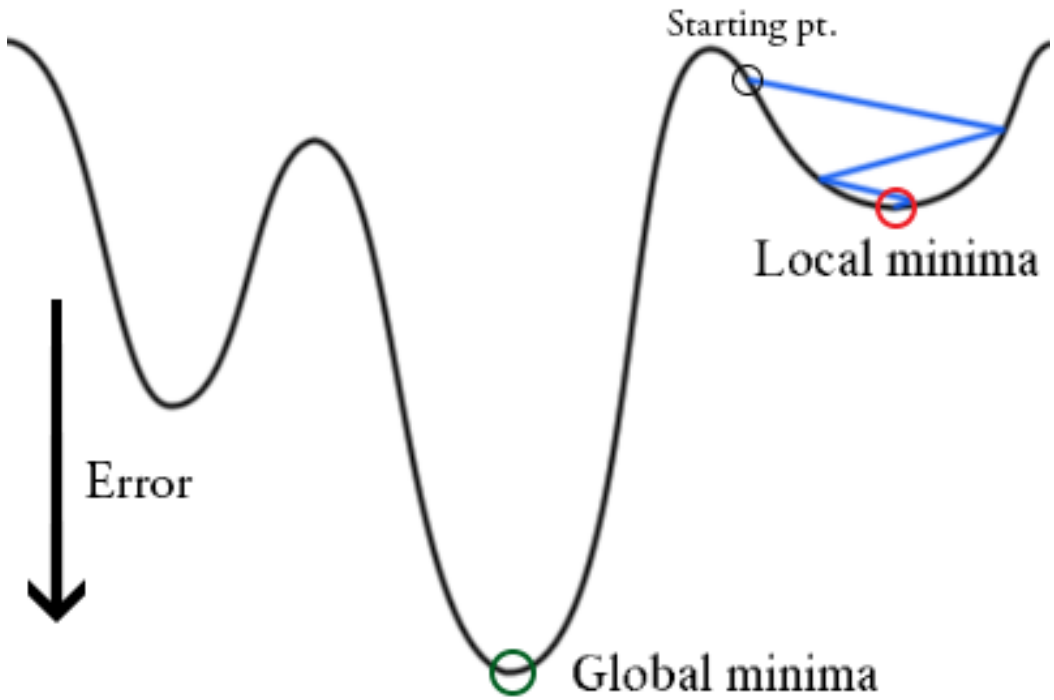
Using above equations, the updates are performed in two steps:

- **Forward pass** compute function values keeping weights fixed,
- **Backward pass** compute errors and propagate
- Hence the weights are updated.

SGD is typically preferred

$$
\begin{aligned}
w_j^{t+1} &= w_j^t - \gamma_t 2(y_t - f_{(w_t, W_t)}(x_t)))h_{j,t} \\
W_{j,k}^{t+1} &= W_{j,k}^t - \gamma_t 2(y_t - f_{(w_{t+1}, W_t)}(x_t)))w_j \sigma'(w_j^\top x) x_t^k
\end{aligned}
$$

Non convexity and SGD



Few remarks

- Optimization by **gradient methods**– typically SGD
- **Online** update rules are potentially biologically plausible– **Hebbian learning** rules describing neuron **plasticity**
- **Multiple layers** can be analogously considered
- **Multiple step-size per layers** can be considered
- **Initialization** is tricky- more later
- **NO** convergence guarantees
- More tricks later

Some questions

- What is the benefit of multiple layers?
- Why does stochastic gradient seem to work?

Wrapping up part I

- Learning classifier and representation
- From shallow to deep learning
- SGD and backpropagation

Coming up

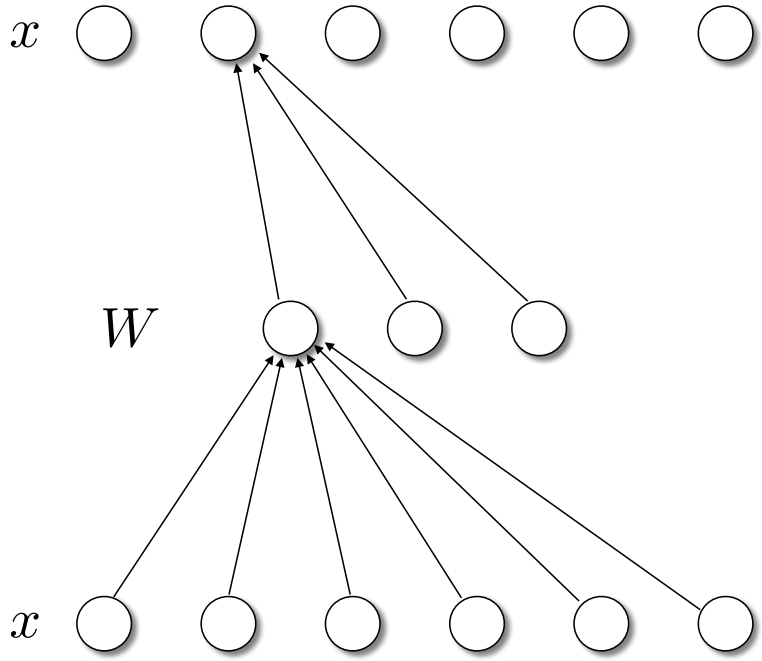
- Autoencoders and unsupervised data?
- Convolutional neural networks
- Tricks and tips

Part II:

Unsupervised learning with neural networks

- Because unlabeled data abound
- Because one could use obtained weight for initialize supervised learning (pre-training)

Auto-encoders



- A neural network with **one input layer, one output layer and one (or more) hidden layers** connecting them.
- The output layer has **equally** many nodes as the input layer,
- It is trained to **predict the input** rather than some target output.

Auto-encoders (cont.)

An auto encoder with one hidden layer of k units, can be seen as a **representation-reconstruction** pair:

$$\Phi : \mathbb{R}^D \to \mathcal{F}_k, \quad \Phi(x) = \sigma\left(Wx\right), \quad \forall x \in \mathbb{R}^D$$

with $\mathcal{F}_k = \mathbb{R}^k$, $k < d$ and

$$\Psi : \mathcal{F}_k \to \mathbb{R}^D, \quad \Psi(\beta) = \sigma\left(W'\beta\right), \quad \forall \beta \in \mathcal{F}_k.$$

Auto-encoders & dictionary learning

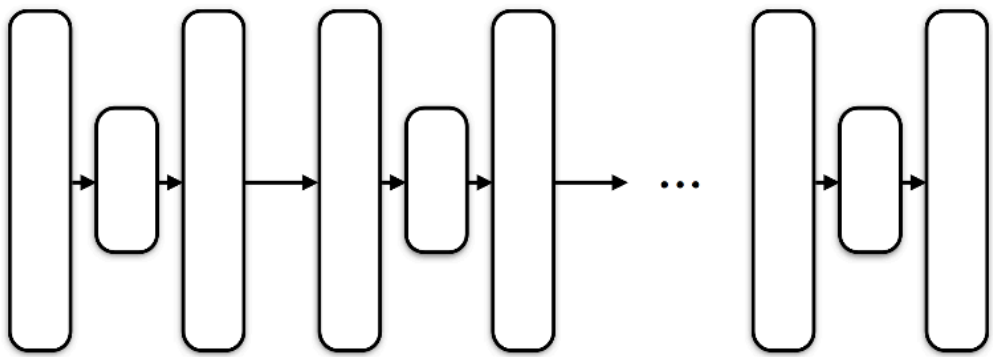
$$\Phi(x) = \sigma\left(Wx\right), \qquad \Psi(\beta) = \sigma\left(W'\beta\right)$$

- Reconstructive approaches have connections with so called **energy models** [LeCun et al.. . .]
- Possible **probabilistic/Bayesian** interpretations/variations (e.g. Boltzmann machine [Hinton et al.. . .])
- The above formulation is closely related to **dictionary learning**.
- The weights can be seen as dictionary **atoms**.

Stacked auto-encoders

Multiple layers of auto-encoders can be **stacked** [Hinton et al '06]...

$$\underbrace{(\Phi_1 \circ \Psi_1)}_{\text{Autoencoder}} \circ (\Phi_2 \circ \Psi_2) \cdots \circ (\Phi_\ell \circ \Psi_\ell)$$



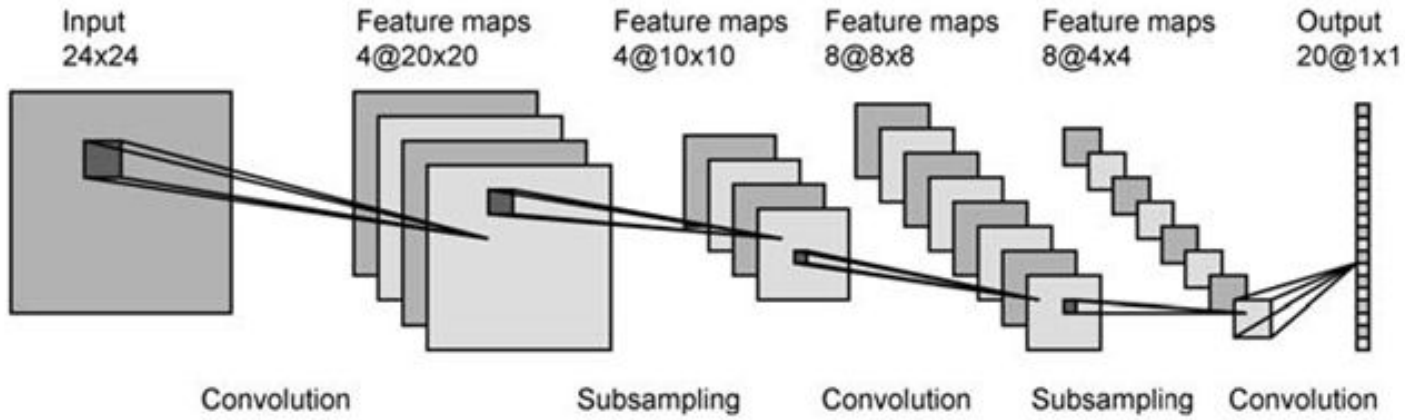
...with the potential of obtaining **richer** representations.

Are auto-encoders useful?

- Pre-training has not delivered as hoped: supervised training on big data-sets is best...

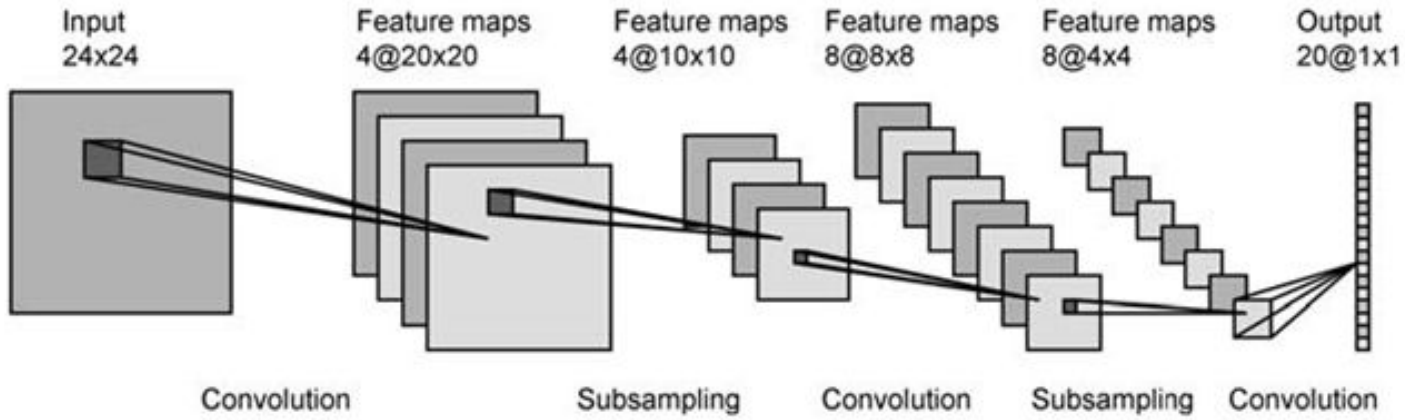
- Still a lot of work on the topic: variational autoencoders, denoising autoencoderes, sparse autoencoders...

Beyond reconstruction



In many applications the **connectivity** of neural networks is limited in a specific way.

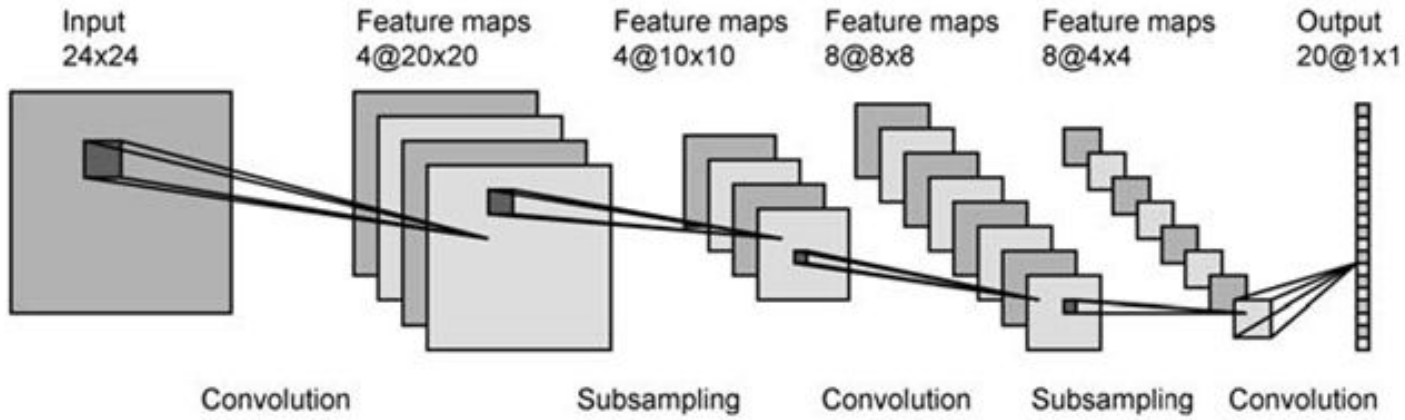
Beyond reconstruction



In many applications the **connectivity** of neural networks is limited in a specific way.

- Weights in the first few layers have **smaller support** and are **repeated**- weight sha ring.

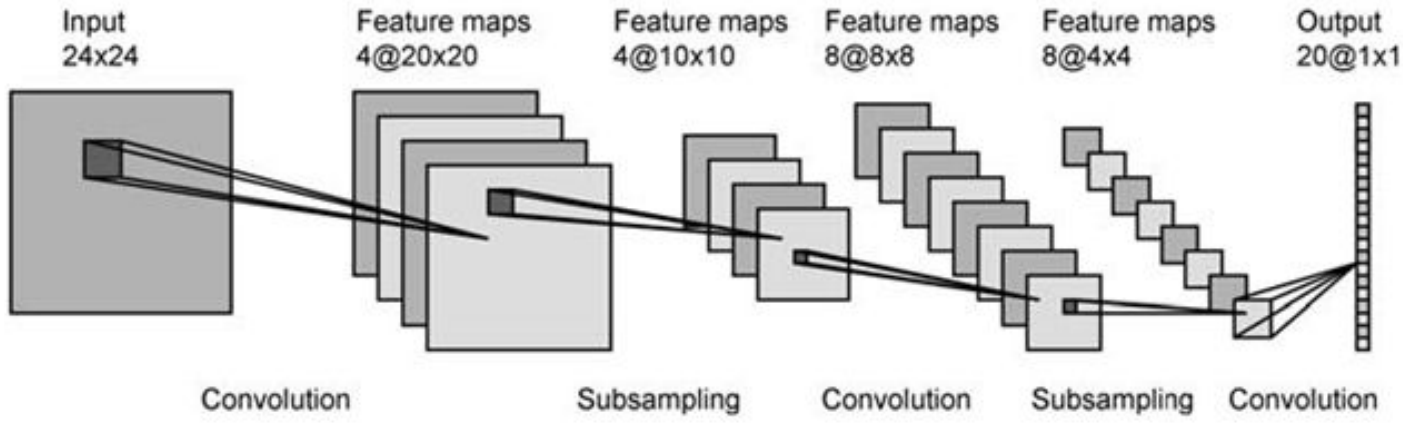
Beyond reconstruction



In many applications the **connectivity** of neural networks is limited in a specific way.

- Weights in the first few layers have **smaller support** and are **repeated**- weight sha ring.
- **Subsampling** (*pooling*) is interleaved with standard neural nets computations.

Beyond reconstruction



In many applications the **connectivity** of neural networks is limited in a specific way.

- Weights in the first few layers have **smaller support** and are **repeated**- weight sha ring.
- **Subsampling** (*pooling*) is interleaved with standard neural nets computations.

The obtained architectures are called **convolutional neural networks**.

Convolutional layers

Consider the composite representation

$$\Phi : \mathbb{R}^D \to \mathcal{F}, \quad \Phi = \sigma \circ W,$$

with

- representation by **filtering** $W : \mathbb{R}^D \to \mathcal{F}'$,
- representation by **pooling** $\sigma : \mathcal{F}' \to \mathcal{F}$.

Convolutional layers

Consider the composite representation

$$\Phi : \mathbb{R}^D \to \mathcal{F}, \quad \Phi = \sigma \circ W,$$

with

- representation by **filtering** $W : \mathbb{R}^D \to \mathcal{F}'$,
- representation by **pooling** $\sigma : \mathcal{F}' \to \mathcal{F}$.

Note: σ, W are more complex than in standard NN.

Convolution and filtering

The matrix W is made of blocks

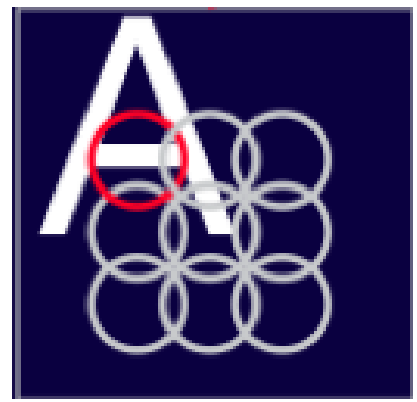
$$W = (G_{t_1}, \ldots, G_{t_T})$$

each block is a *convolution matrix* obtained transforming a vector (template) t, e.g.

$$G_t = (g_1 t, \ldots, g_N t).$$

e.g.

$$G_t = \begin{bmatrix} t^1 & t_2 & t_3 & \ldots & t^d \\ t^d & t^1 & t_2 & \ldots & t^{d-1} \\ t^{d-1} & t^d & t^1 & \ldots & t^{d-2} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ t^2 & t^3 & t^4 & \ldots & t^1 \end{bmatrix}$$



For all $x \in \mathbb{R}^D$,

$$W(x)(j,i) = x^\top g_i t_j$$

Convolution and filtering

The matrix W is made of blocks

$$W = (G_{t_1}, \ldots, G_{t_T})$$

then

$$Wx = (t_1 \star x), \ldots, (t_T \star x)$$

Note: Compare to standard neural nets where

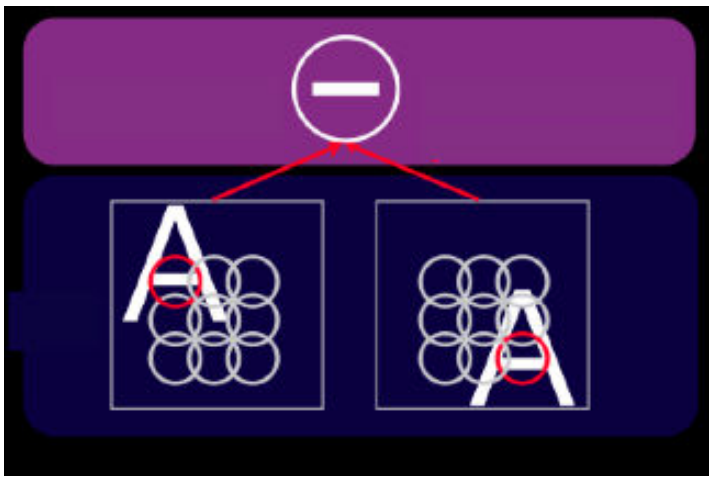
$$Wx = t_1^\top x, \ldots, t_T^\top x$$

Pooling

The **pooling** map **aggregates** (pools) the values corresponding to the same transformed template

$$x \star t = x^\top g_1 t, \ldots, x^\top g_N t,$$

and can be seen as a form of **subsampling**.



Pooling functions

Given a template t, let

$$\beta = \sigma(x \star t) = \left(\sigma(x^\top g_1 t), \ldots, \sigma(x^\top g_N t)\right).$$

for some non-linearity σ, e.g. $\sigma(\cdot) = |\cdot|_+$.

Pooling functions

Given a template t, let

$$\beta = \sigma(x \star t) = \left(\sigma(x^\top g_1 t), \ldots, \sigma(x^\top g_N t)\right).$$

for some non-linearity σ, e.g. $\sigma(\cdot) = |\cdot|_+$.

Examples of pooling

- max pooling

$$\max_{j=1,\ldots,N} \beta^j,$$

- average pooling

$$\frac{1}{N}\sum_{j=1}^{N} \beta^j,$$

- ℓ_p pooling

$$\|\beta\|_p = \left(\sum_{j=1}^{N} |\beta^j|^p\right)^{\frac{1}{p}}.$$

Why pooling?

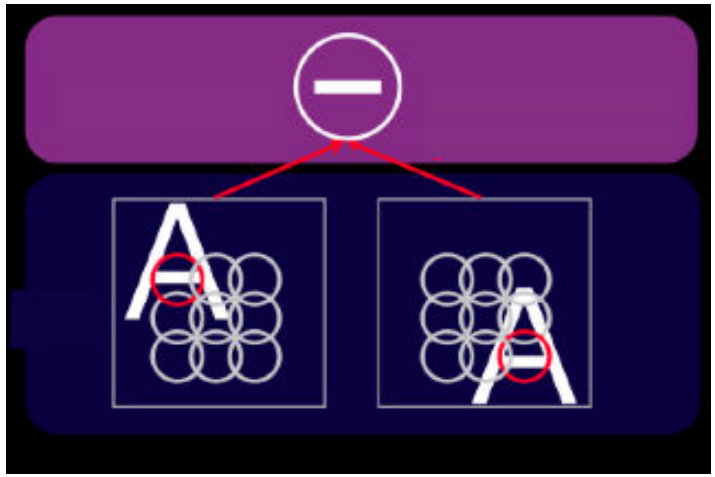
The intuition is that pooling can provide some form of robustness and even **invariance** to the transformations.

Invariance & selectivity

- A good representation should be **invariant** to **semantically irrelevant** transformations.
- Yet, it should be **discriminative** with respect to **relevant** information (**selective**).

Basic computations: simple & complex cells

(Hubel, Wiesel '62)



- Simple cells

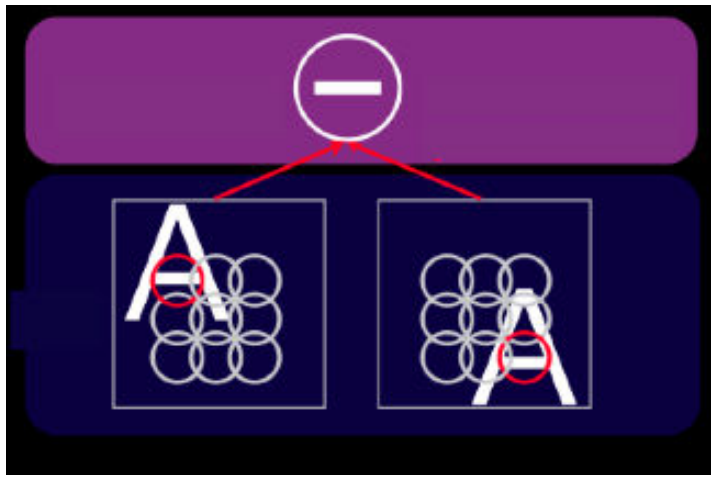
$$x \mapsto x^\top g_1 t, \ldots, x^\top g_N t$$

- Complex cells

$$x^\top g_1 t \ldots, x^\top g_N t \mapsto \sum_g |x^\top g t|_+$$

Basic computations: convolutional networks

(Le Cun '88)



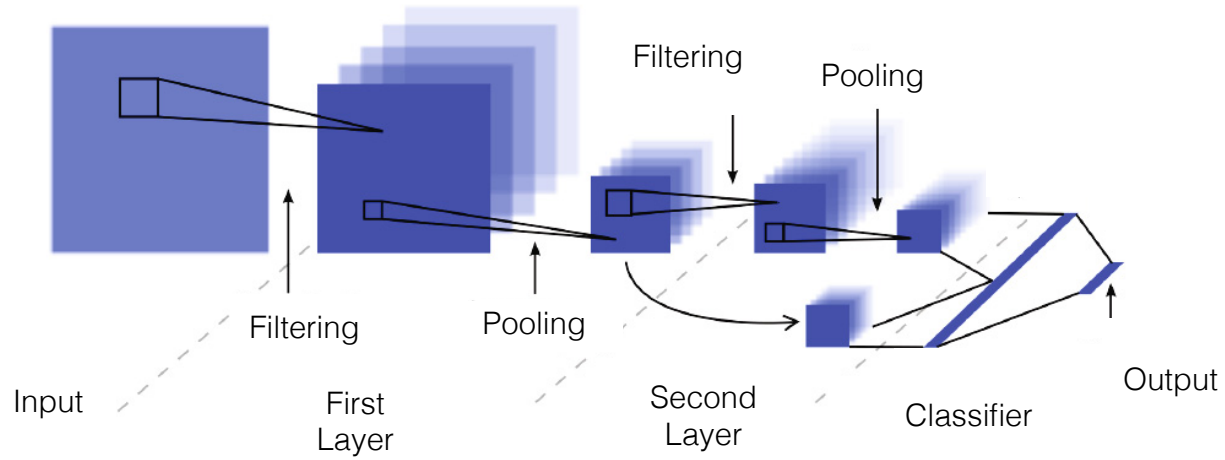
- Convolutional filters

$$x \mapsto x^\top g_1 t, \ldots, x^\top g_N t$$

- Subsampling/pooling

$$x^\top g_1 t \ldots, x^\top g_N t \mapsto \sum_g |x^\top g t|_+$$

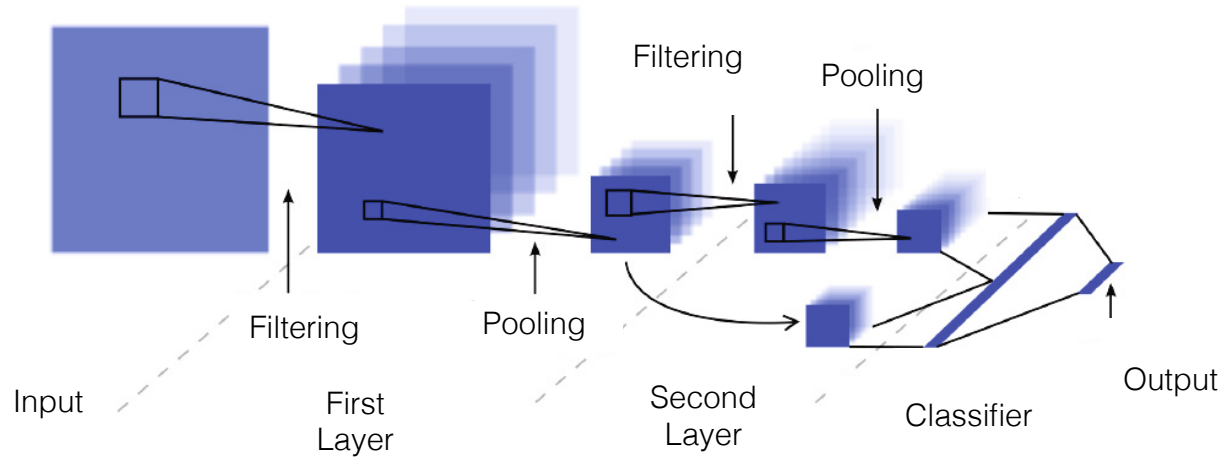
Deep convolutional networks



In practice:

- multiple convolution layers are **stacked**,

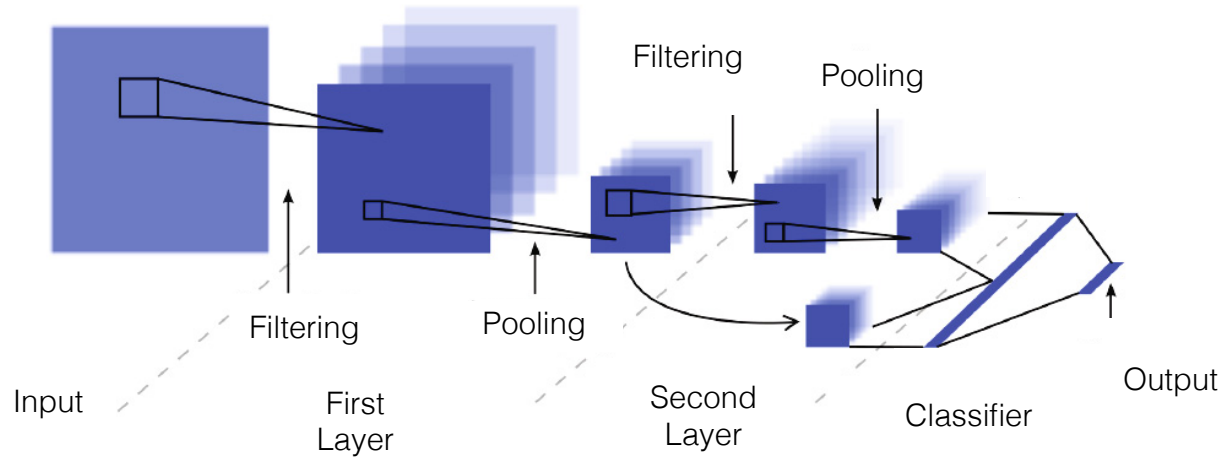
Deep convolutional networks



In practice:

- multiple convolution layers are **stacked**,
- pooling is not global, but over a subset of transformations (**receptive field**),

Deep convolutional networks



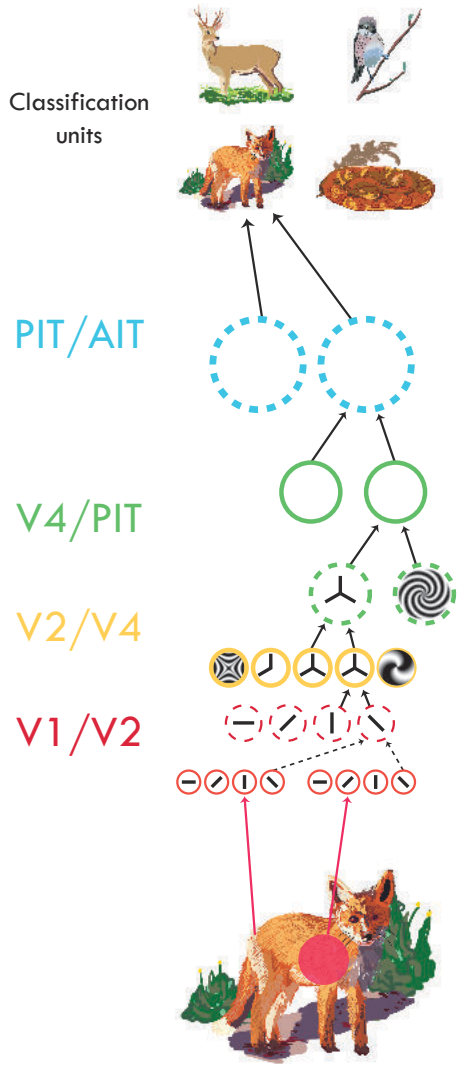
In practice:

- multiple convolution layers are **stacked**,
- pooling is not global, but over a subset of transformations (**receptive field**),
- the receptive fields size increases in **higher layers**.

A biological motivation

Visual cortex

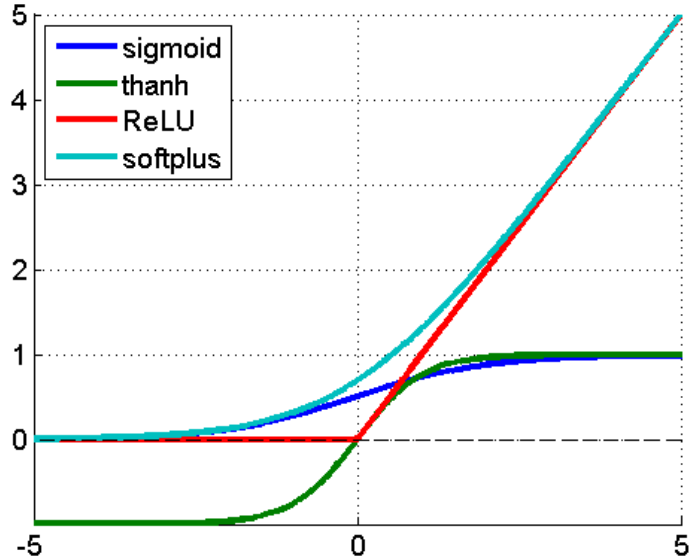
The processing in DCN has analogies with computational neuroscience models of the **information processing in the visual cortex see [Poggio et al. . . .]**.



Outline

Some Tricks of the Trade

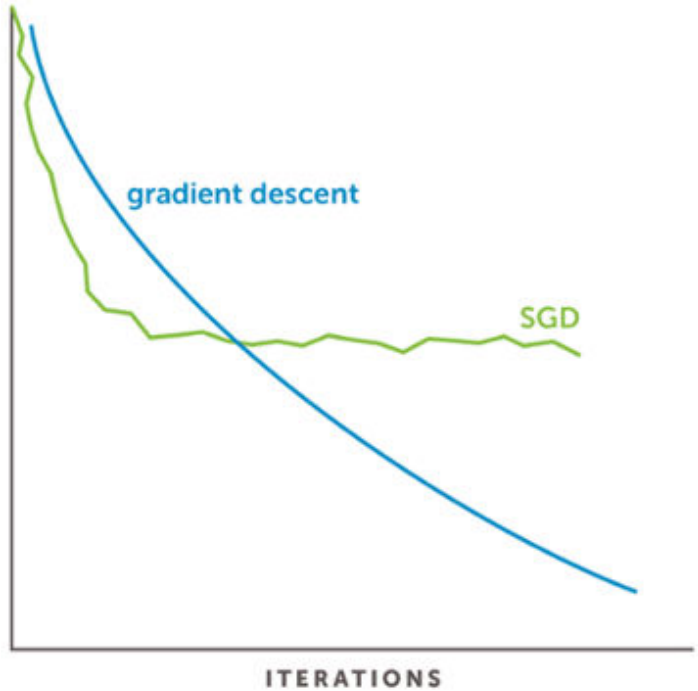
Which activation function?



- Biological motivation
- Rich function spaces
- Avoid vanishing gradient
- Fast gradient computation

ReLU: It has the last two properties! It seems to work best in practice!

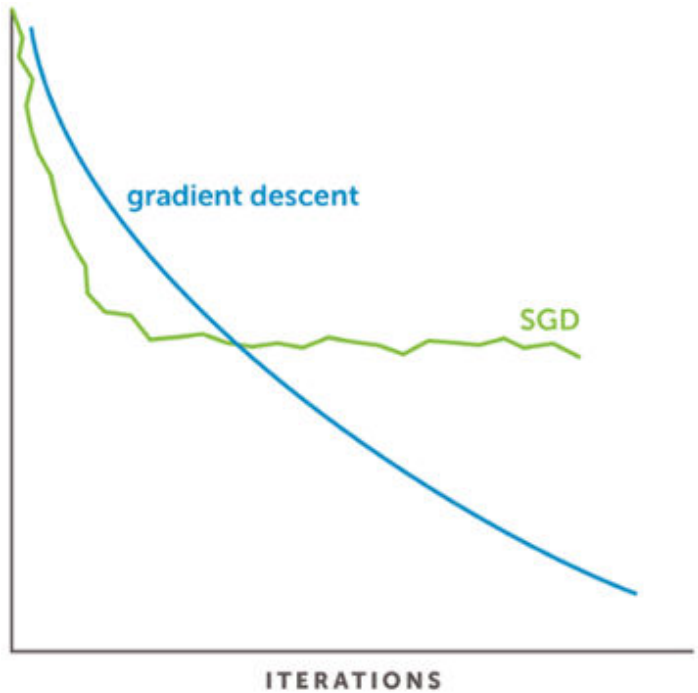
SGD is slow...



Accelerations

- **Momentum**
- Nesterov's method
- Adam
- Adagrad
- ...

Mini-Batch SGD



- GD: use all points each iteration to compute gradient
- SGD: use one point each iteration to compute gradient
- Mini-Batch: use a *mini-batch* of points each iteration to compute gradient

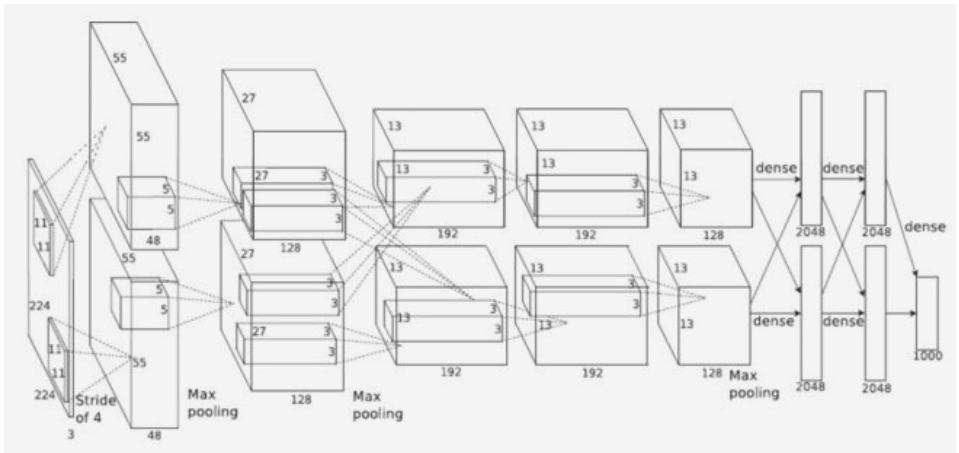
Why? Faster convergence/More stable behavior

Initialization: learning from scratch

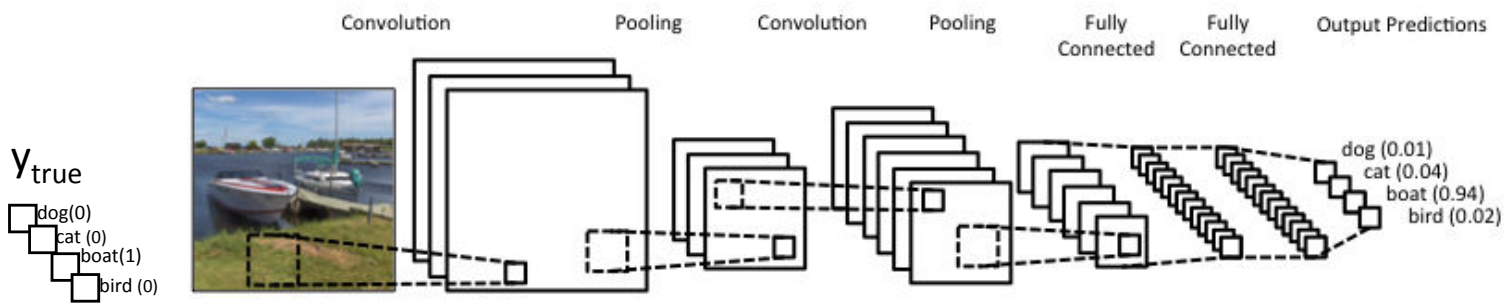
Large-scale Datasets

General Purpose GPUs

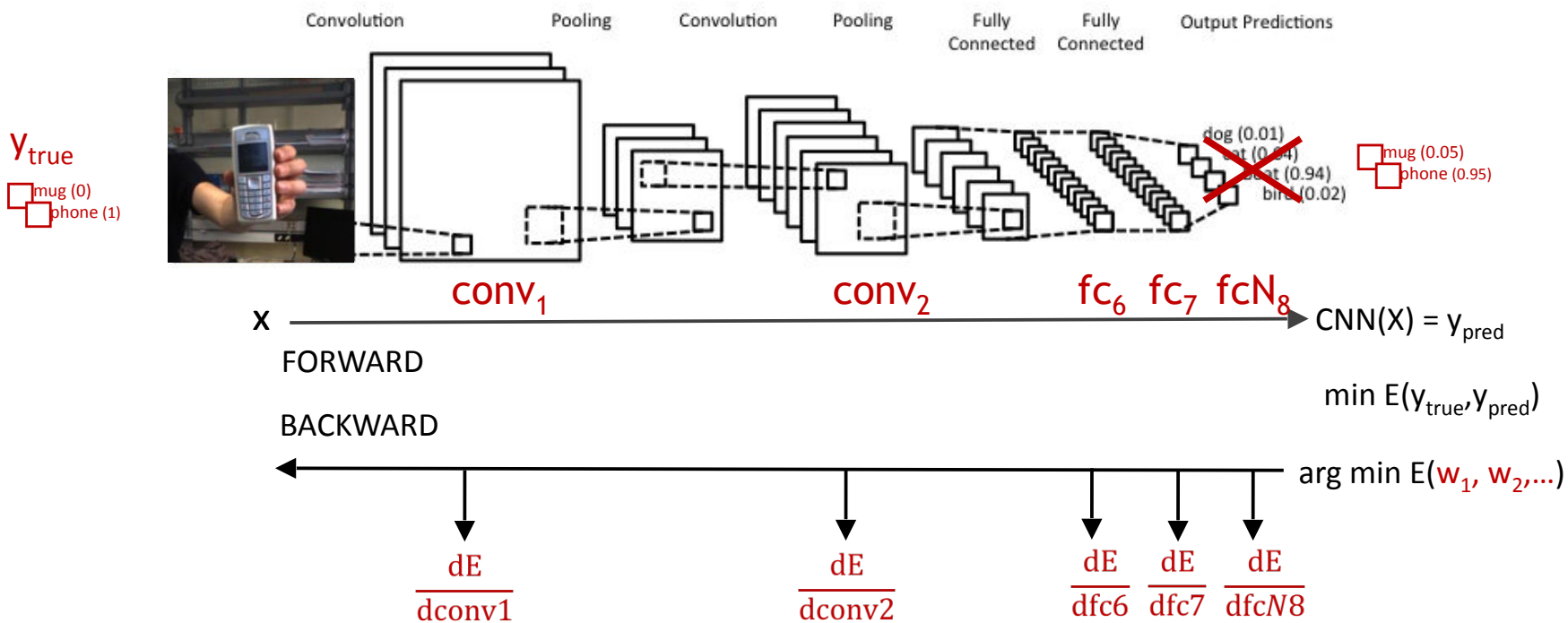
AlexNet
Krizhevsky
et al (2012)



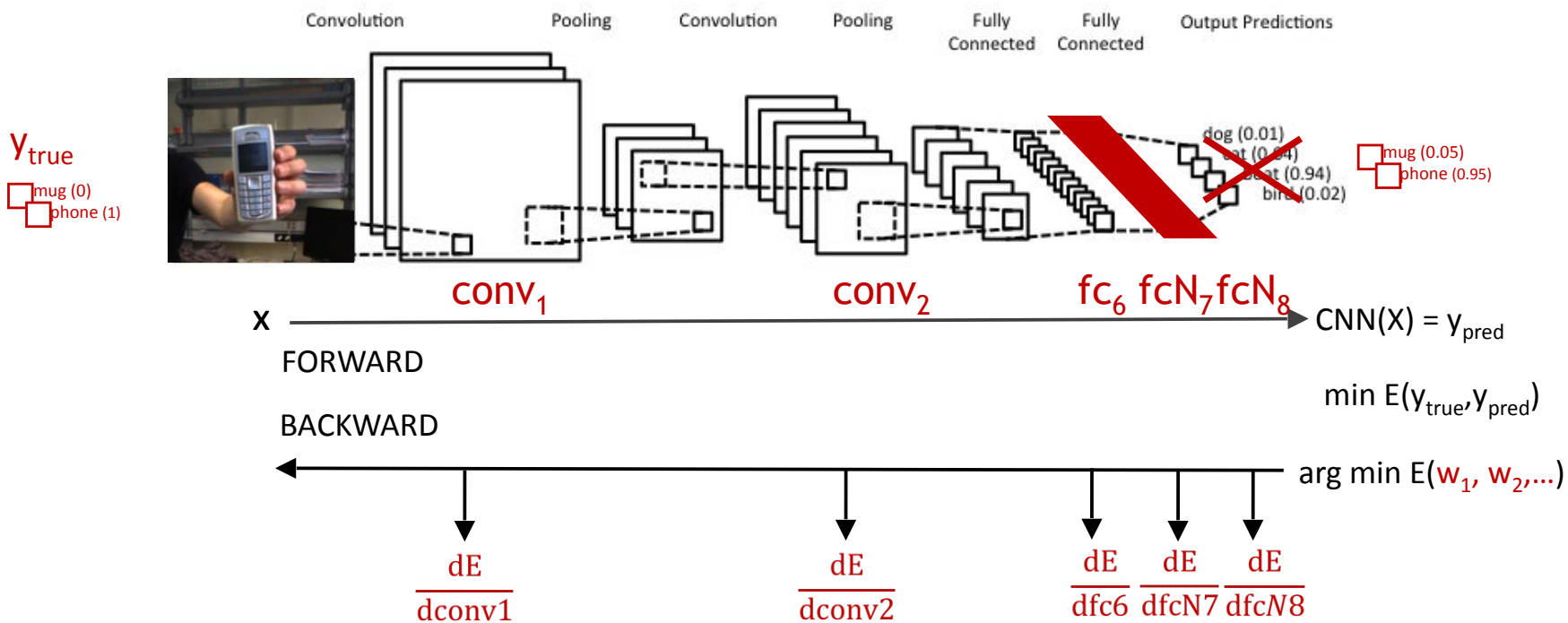
Initialization & fine tuning



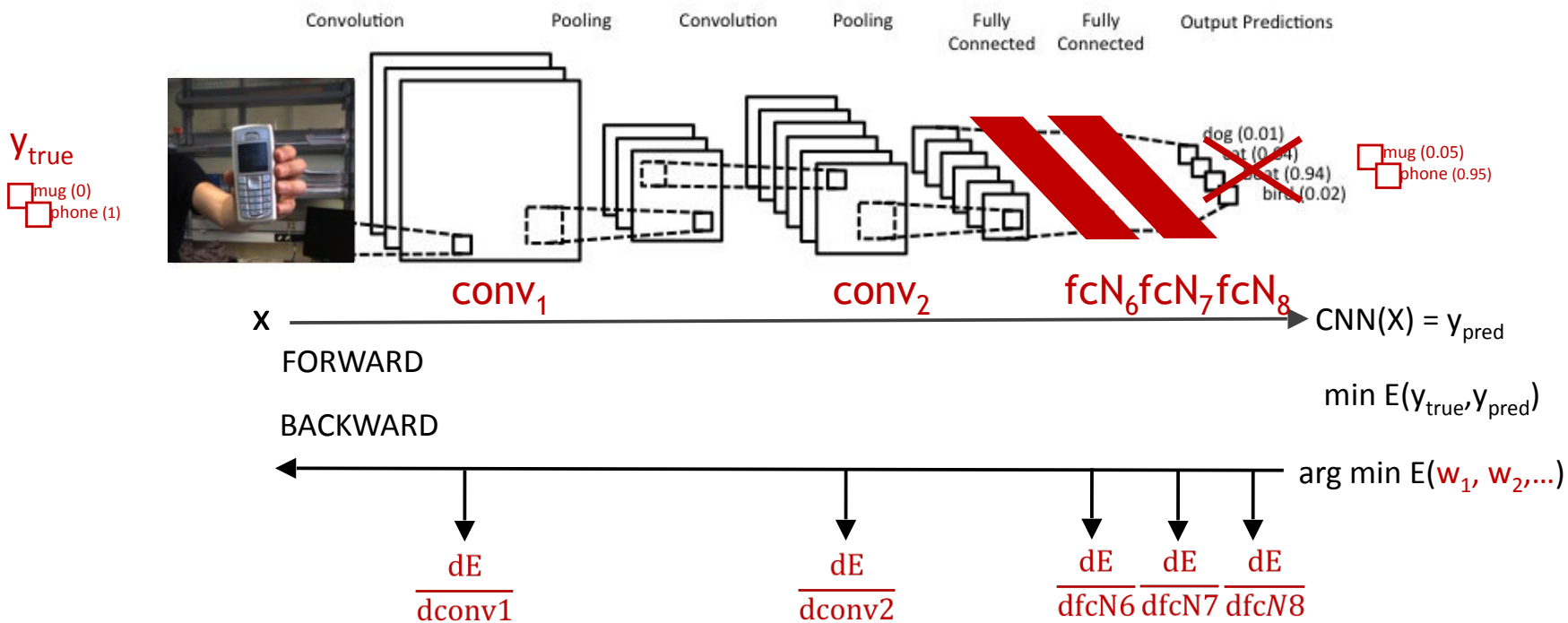
Initialization & fine tuning



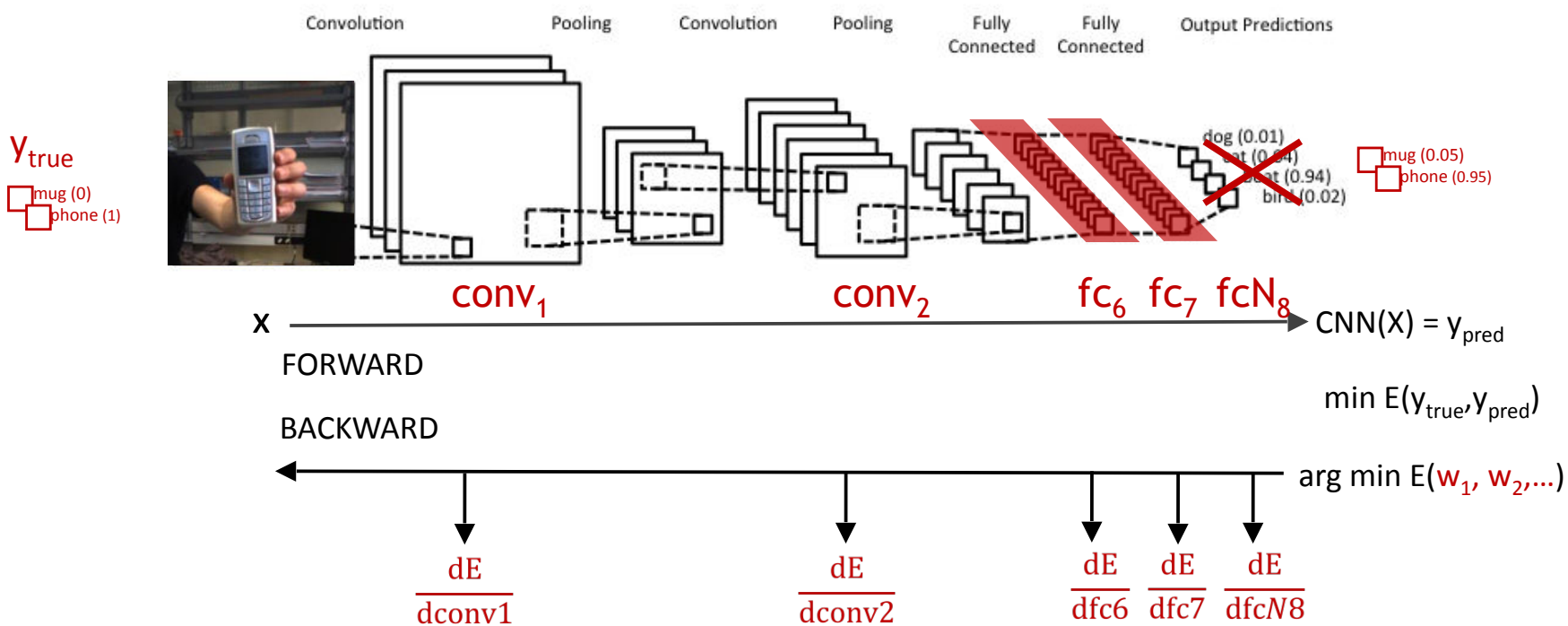
Initialization & fine tuning



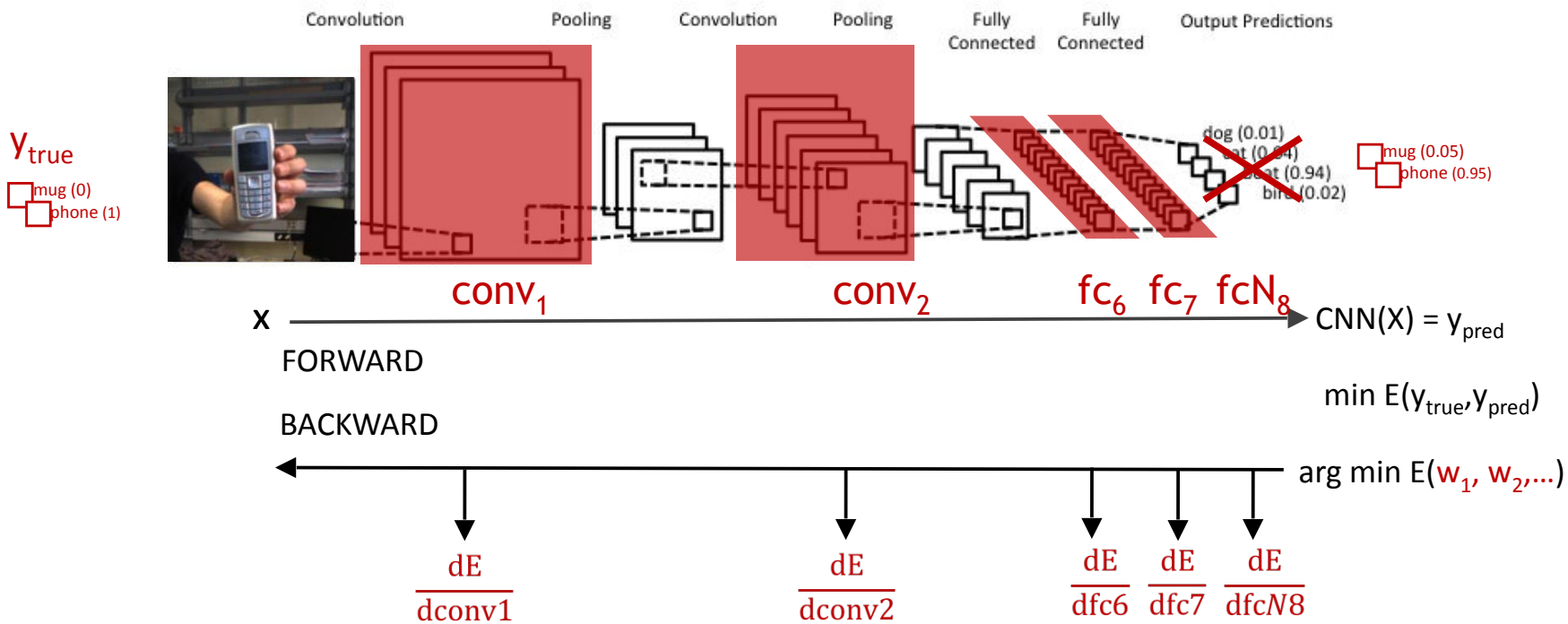
Initialization & fine tuning



Initialization & fine tuning



Initialization & fine tuning



- Learning layers from scratch/from pre-learned initialization
- Learning layers more/less aggressively using different step-sizes

Training protocol(s)

- Learning at different layers
  - Initialization
  - Learning rates
- Mini-batch size
- Further aspect: regularization!
  - Weight constraints
  - Drop-out
- Batch normalization
- ...

Wrapping up

- Unlabelled data and auto-encoders
- CNN: the power of weight sharing for learning
- Tips and tricks (fine tune!)

Final remarks

- Learning representations with deep-nets
- Learning deep-nets with back-prop
- CNN: the power of weight sharing for learning
- More deep-nets: Inception, GAN, Recurrent net, LSTM, ...

Final remarks

- Learning representations with deep-nets
- Learning deep-nets with back-prop
- CNN: the power of weight sharing for learning
- More deep-nets: Inception, GAN, Recurrent net, LSTM, ...

But why do they work?! Gotta be that they are like the brain...