

CENG 783

On Machine Learning and  
Optimization

Sinan Kalkan

# Optimization Methods for Large-Scale Machine Learning

Léon Bottou\*

Frank E. Curtis†

Jorge Nocedal‡

June 16, 2016

## Abstract

This paper provides a review and commentary on the past, present, and future of numerical optimization algorithms in the context of machine learning applications. Through case studies on text classification and the training of deep neural networks, we discuss how optimization problems arise in machine learning and what makes them challenging. A major theme of our study is that large-scale machine learning represents a distinctive setting in which the stochastic gradient (SG) method has traditionally played a central role while conventional gradient-based nonlinear optimization techniques typically falter. Based on this viewpoint, we present a comprehensive theory of a straightforward, yet versatile SG algorithm, discuss its practical behavior, and highlight opportunities for designing algorithms with improved performance. This leads to a discussion about the next generation of optimization methods for large-scale machine learning, including an investigation of two main streams of research on techniques that diminish noise in the stochastic directions and methods that make use of second-order derivative approximations.

# Now

- Introduction to ML
  - Problem definition
  - Classes of approaches
  - K-NN
  - Support Vector Machines
  - Softmax classification / logistic regression
  - Parzen Windows
- Optimization
  - Gradient Descent approaches
  - A flavor of other approaches



# Introduction to Machine learning

Uses many figures and material from the following website:  
[http://www.byclb.com/TR/Tutorials/neural\\_networks/ch1\\_1.htm](http://www.byclb.com/TR/Tutorials/neural_networks/ch1_1.htm)

# Overview

## Training

Data with label  
(label: human, animal etc.)



Extract Features

- Size
- Texture
- Color
- Histogram of oriented gradients
- SIFT
- Etc.

“Learn”

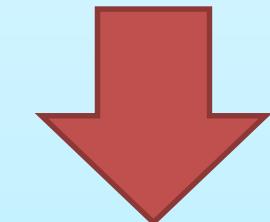


## Testing

Extract Features



Human or Animal or  
5  
...



Learned Models or Classifiers

# Content

- Problem definition
- General approaches
- Popular methods
  - kNN
  - Linear classification
  - Support Vector Machines
  - Parzen windows

# Problem Definition

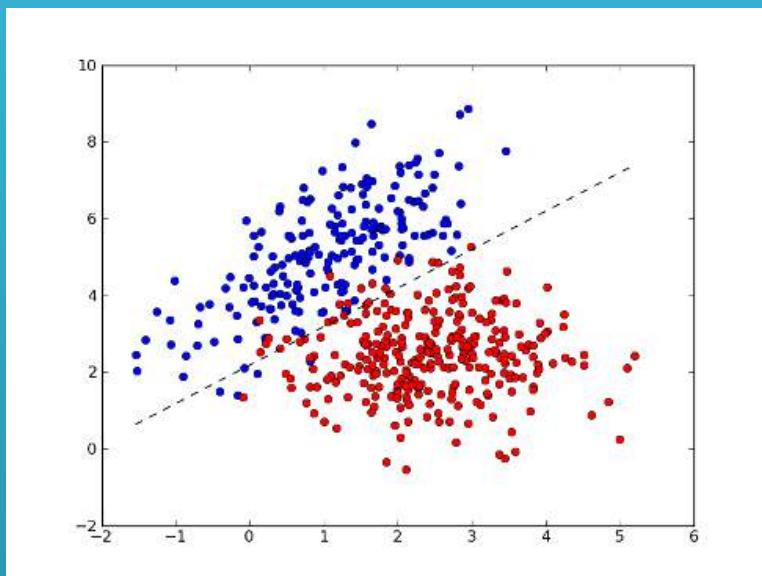
- Given
  - Data: a set of instances  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$  sampled from a space  $\mathbf{X} \in \mathbb{R}^d$ .
  - Labels: the corresponding labels  $y_i$  for each  $\mathbf{x}_i$ .  $y_i \in \mathbf{Y} \in \mathbb{R}$ .
- Goal:
  - Learn a mapping from the space of “data” to the space of labels, i.e.,
    - $\mathcal{M}: \mathbf{X} \rightarrow \mathbf{Y}$ .
    - $y = f(\mathbf{x})$ .

# Issues in Machine Learning

- Hypothesis space
- Loss / cost / objective function
- Optimization
- Bias vs. variance
- Test / evaluate
- Overfitting, underfitting

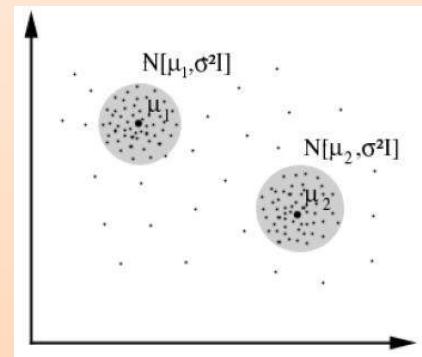
# General Approaches

## Discriminative

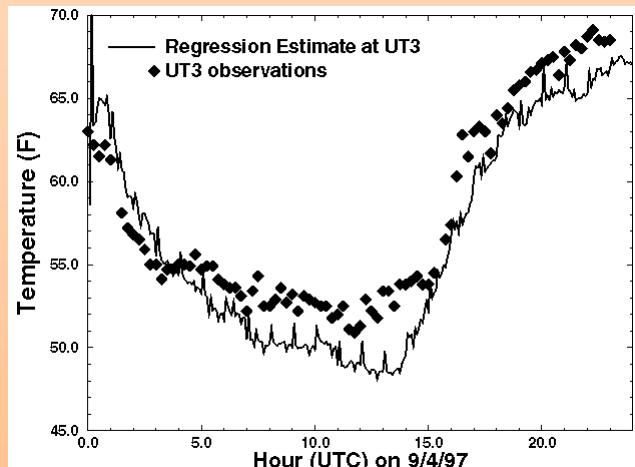


Find separating line  
(in general: hyperplane)

## Generative



Learn a model for each class.

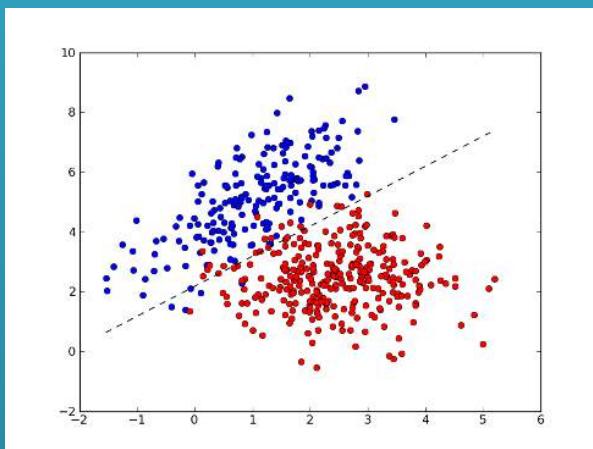


Fit a function to data  
(regression).

# General Approaches (cont'd)

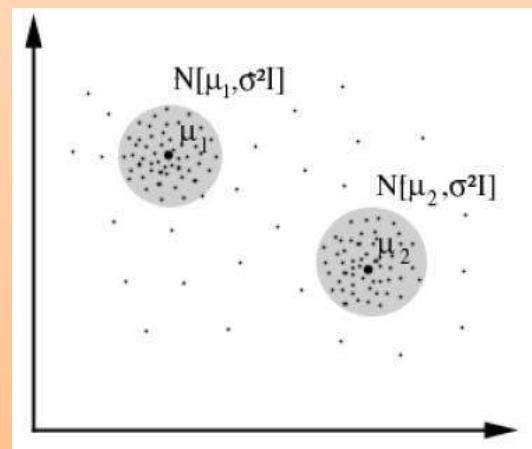
## Discriminative

- Support Vector Machines
- Artificial Neural Networks
- Conditional Random Fields
- K-NN



## Generative

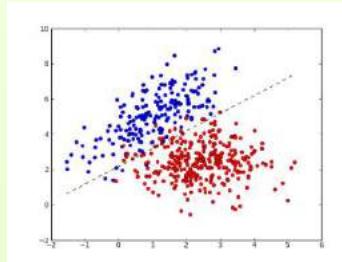
- Regression
- Markov Random Fields
- Bayesian Networks/Learning
- Clustering via Gaussian Mixture Models,  
*Parzen Windows etc.*
- ...



# General Approaches (cont'd)

## Supervised

Instance	Label
	elma
	elma
	armut
	elma
	armut



e.g. SVM

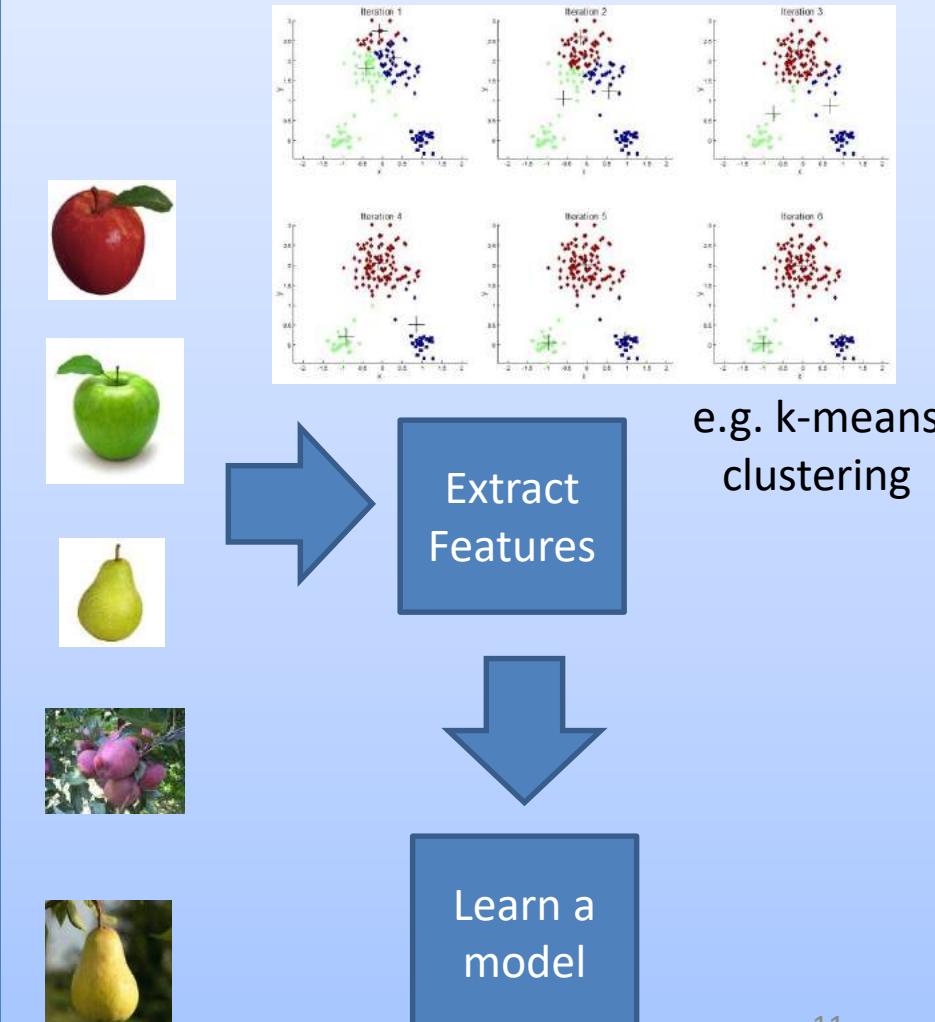


Extract  
Features



Learn a  
model

## Unsupervised



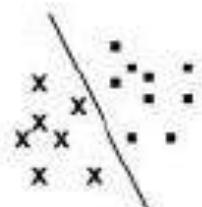
# General Approaches (cont'd)

	<b>Generative</b>	<b>Discriminative</b>
<b>Supervised</b>	Regression Markov Random Fields Bayesian Networks	Support Vector Machines Neural Networks Conditional Random Fields Decision Tree Learning
<b>Unsupervised</b>	Gaussian Mixture Models Parzen Windows	K-means Self-Organizing Maps

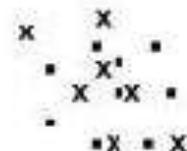
# Now

	Generative	Discriminative
Supervised	Softmax/Logistic Regression	Support Vector Machines
Unsupervised	Parzen Windows	

# Feature Space

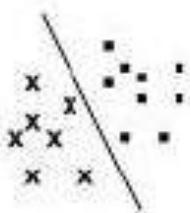


Good features



Bad features

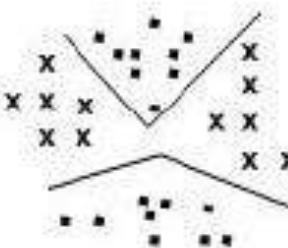
(a)



Linear separability



Non-linear separability



Multi-modal

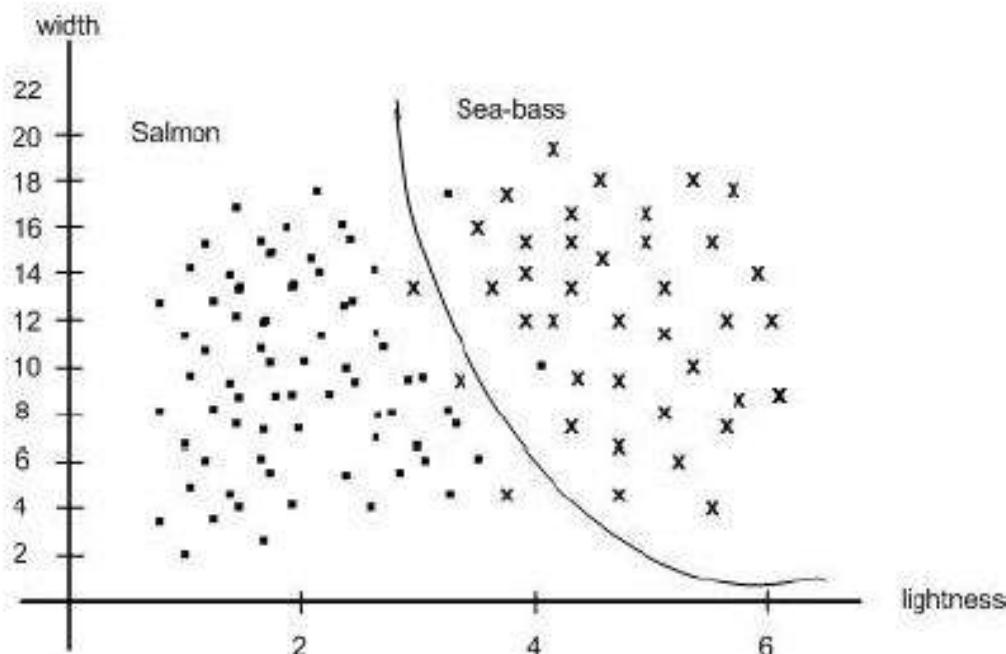
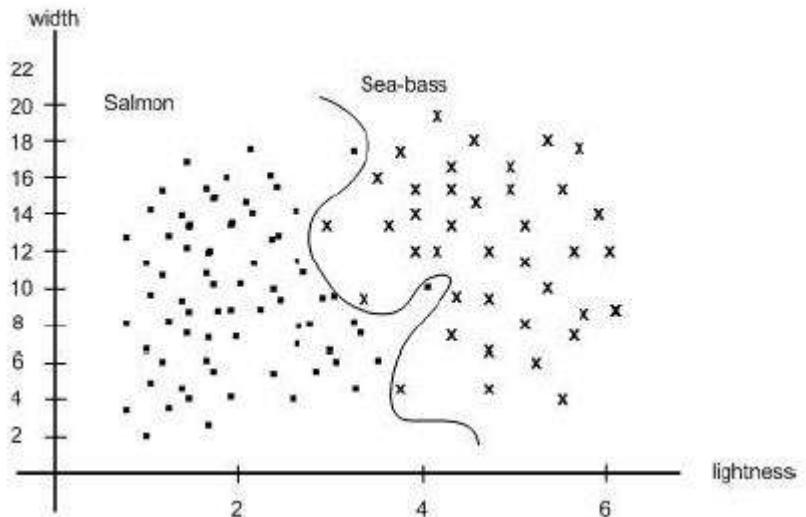


Highly correlated

(b)

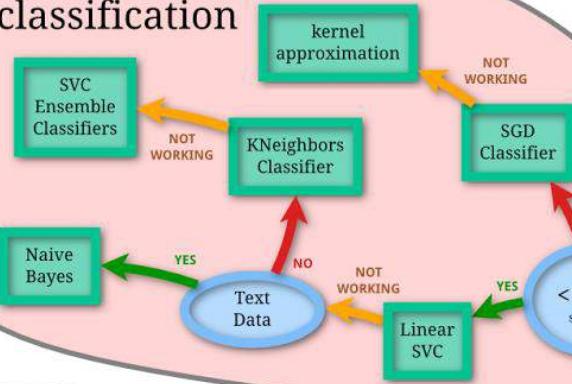
# General pitfalls/problems

- Overfitting
- Underfitting
- Occams' razor

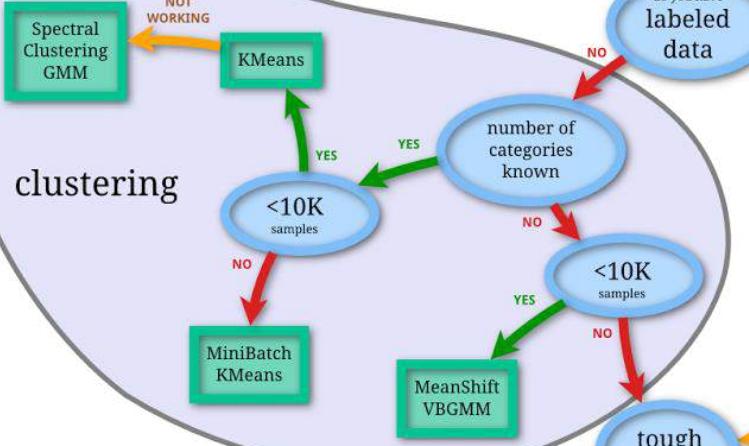


# scikit-learn algorithm cheat-sheet

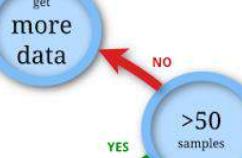
## classification



## clustering



get more data



predicting a category



predicting a quantity

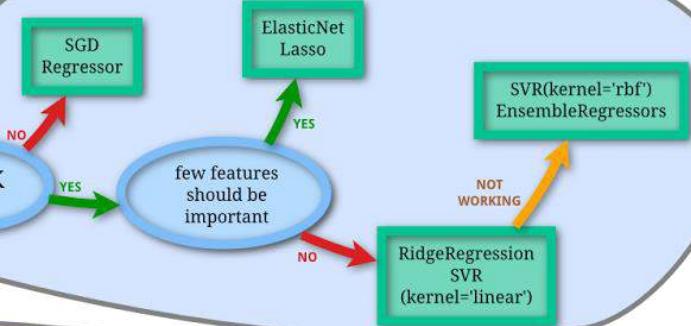
few features should be important

just looking

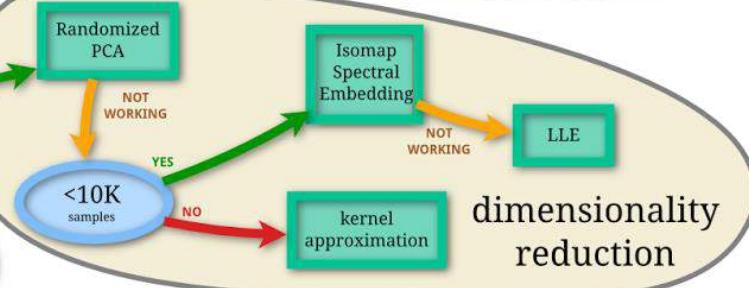
predicting structure

START

## regression



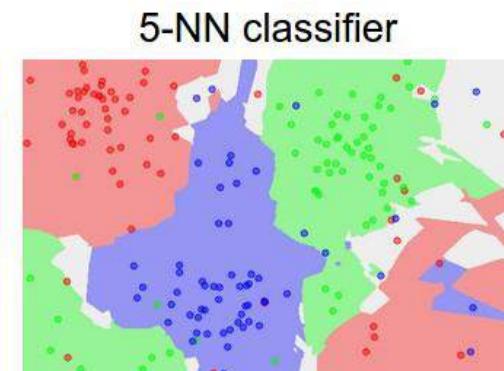
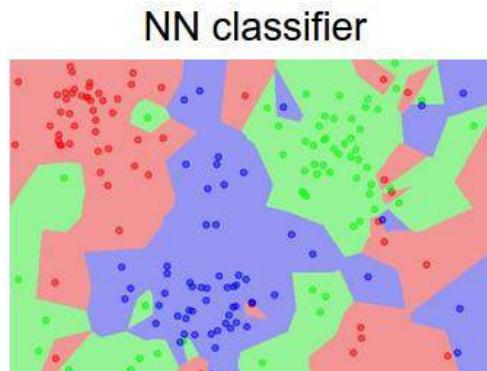
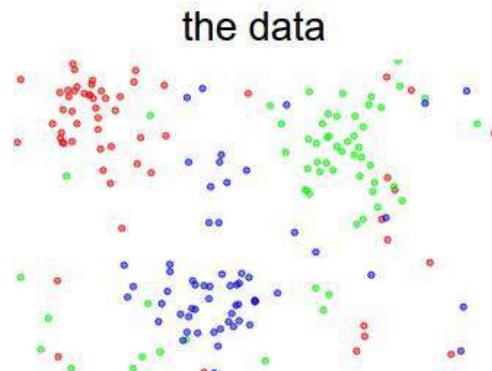
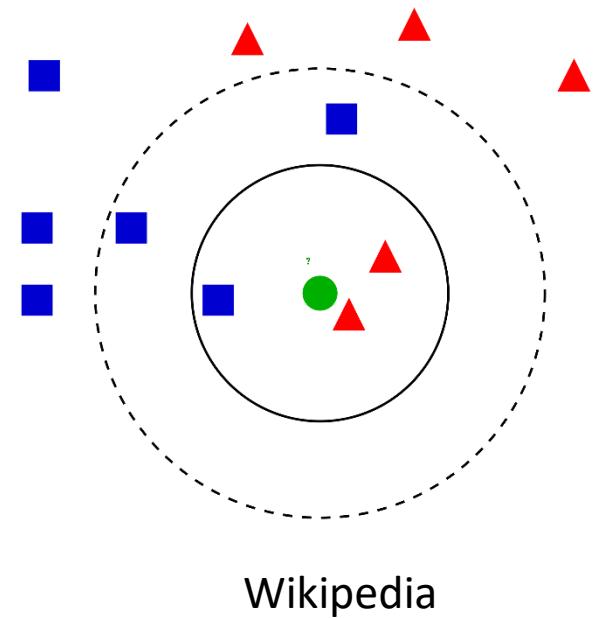
## dimensionality reduction

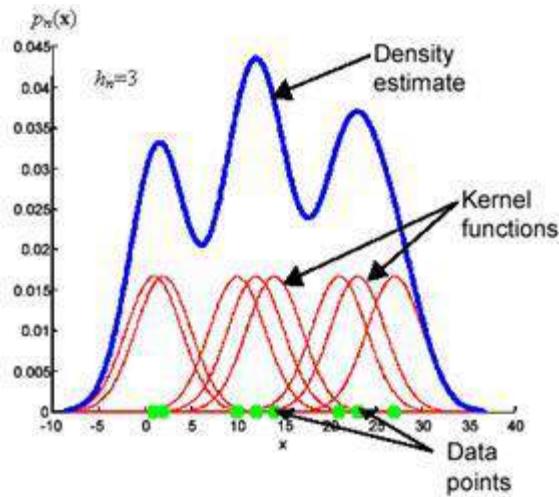


# K-NEAREST NEIGHBOR

# A very simple algorithm

- Advantages:
  - No training
  - Simple
- Disadvantages:
  - Slow testing time (more efficient versions exist)
  - Needs a lot of memory





A non-parametric method

# PARZEN WINDOWS

# Probability and density

- Probability of a continuous probability function,  $p(x)$ , satisfies the following:
  - Probability to be between two values:

$$P(a < x < b) = \int_a^b p(x)dx$$

- $p(x) > 0$  for all real  $x$ .
- And

$$\int_{-\infty}^{\infty} p(x)dx = 1$$

- In 2-D:
  - Probability for  $\mathbf{x}$  to be inside region  $R$ :

$$P = \int_R p(\mathbf{x})d\mathbf{x}$$

- $p(\mathbf{x}) > 0$  for all real  $\mathbf{x}$ .
- And

$$\int_{-\infty}^{\infty} p(\mathbf{x})d\mathbf{x} = 1$$

# Density Estimation

- Basic idea:

$$P = \int_R p(\mathbf{x}) d\mathbf{x}$$

- If  $R$  is small enough, so that  $p(\mathbf{x})$  is almost constant in  $R$ :
  - $P = \int_R p(\mathbf{x}) d\mathbf{x} \approx p(\mathbf{x}) \int_R d\mathbf{x} = p(\mathbf{x})V$
  - $V$ : volume of region  $R$ .

- If  $k$  out of  $n$  samples fall into  $R$ , then

$$P = k/n$$

- From which we can write:

$$\frac{k}{n} = p(\mathbf{x})V \Rightarrow p(\mathbf{x}) = \frac{k/n}{V}$$

# Parzen Windows

- Assume that:
  - $R$  is a hypercube centered at  $\mathbf{x}$ .
  - $h$  is the length of an edge of the hypercube.
  - Then,  $V = h^2$  in 2D and  $V = h^3$  in 3D etc.
- Let us define the following window function:

$$w\left(\frac{\mathbf{x}_i - \mathbf{x}_k}{h}\right) = \begin{cases} 1, & |\mathbf{x}_i - \mathbf{x}_k|/h < 1/2 \\ 0, & \text{otherwise} \end{cases}$$

- Then, we can write the number of samples falling into  $R$  as follows:

$$k = \sum_{i=1}^n w\left(\frac{\mathbf{x}_i - \mathbf{x}_k}{h}\right)$$

# Parzen Windows (cont'd)

- Remember:  $p(\mathbf{x}) = \frac{k/n}{V}$ .
- Using this definition of  $k$ , we can rewrite  $p(\mathbf{x})$  (in 2D):

$$p(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \frac{1}{h^2} w\left(\frac{\mathbf{x}_i - \mathbf{x}}{h}\right)$$

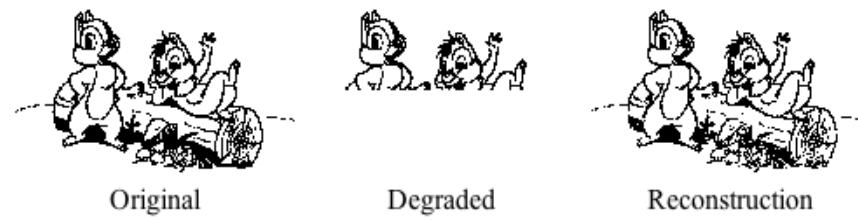
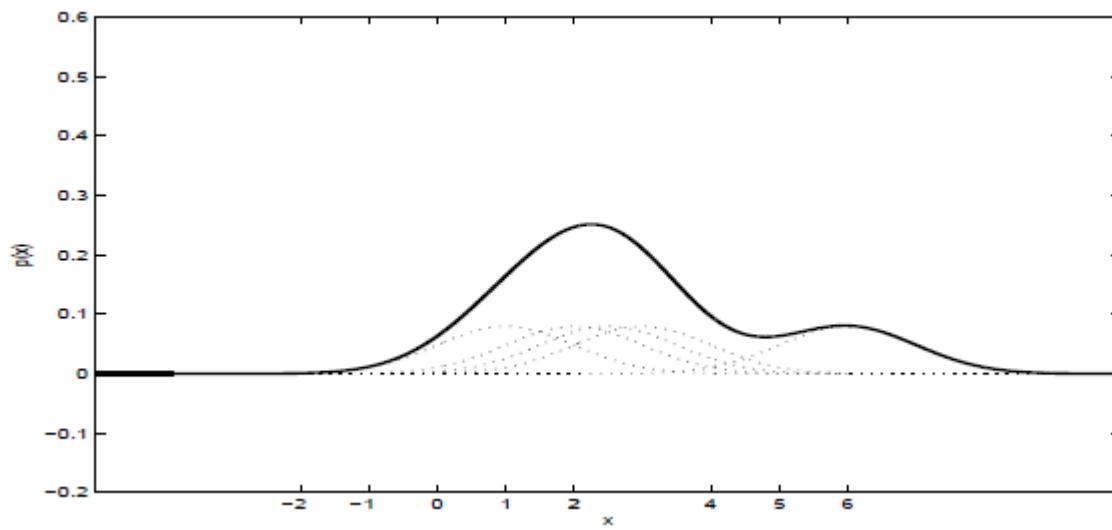
- Interpretation: Probability is the contribution of window functions fitted at each sample!

# Parzen Windows (cont'd)

- The type of the window function can add different flavors.
- If we use Gaussian for the window function:

$$p(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(\mathbf{x}_i - \mathbf{x})^2}{2\sigma^2}\right)$$

- Interpretation: Fit a Gaussian to each sample.



# **LINEAR CLASSIFICATION**

# Linear classification

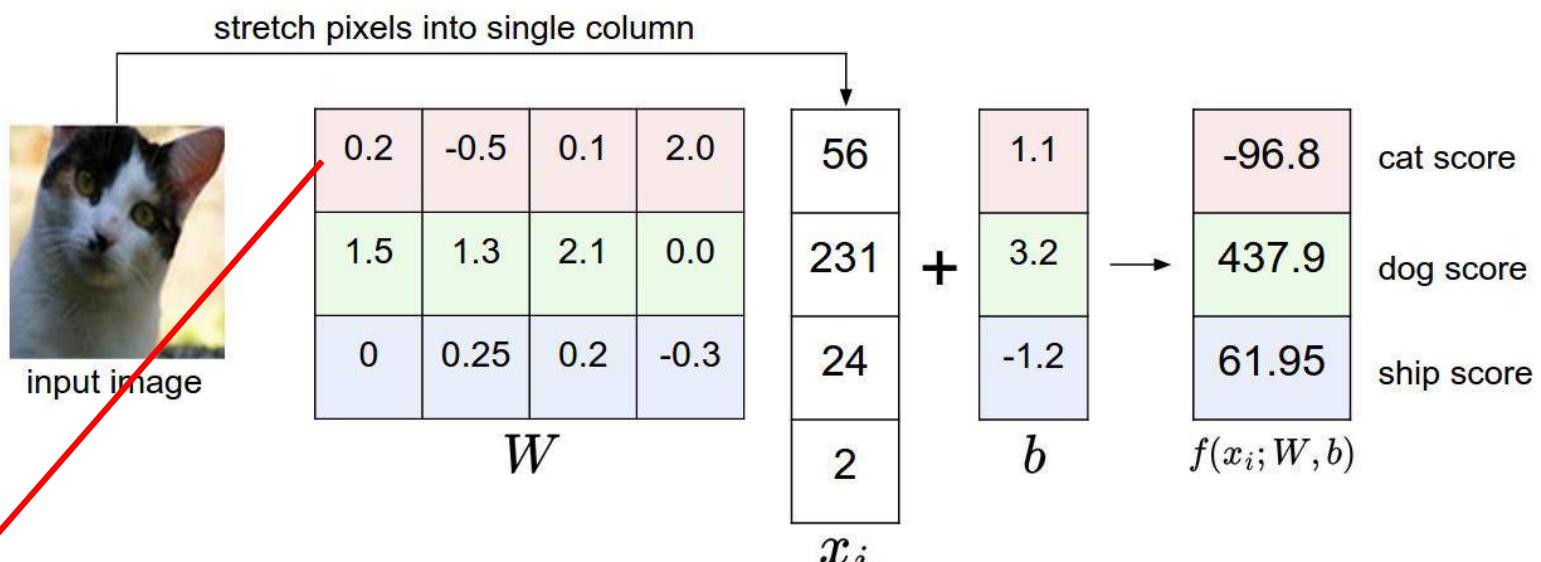
- Linear classification relies on the following score function:

$$f(x_i; W, b) = Wx_i + b$$

Or

$$f(x_i; W, b) = Wx_i$$

- where the bias is implicitly represented in  $W$  and  $x_i$



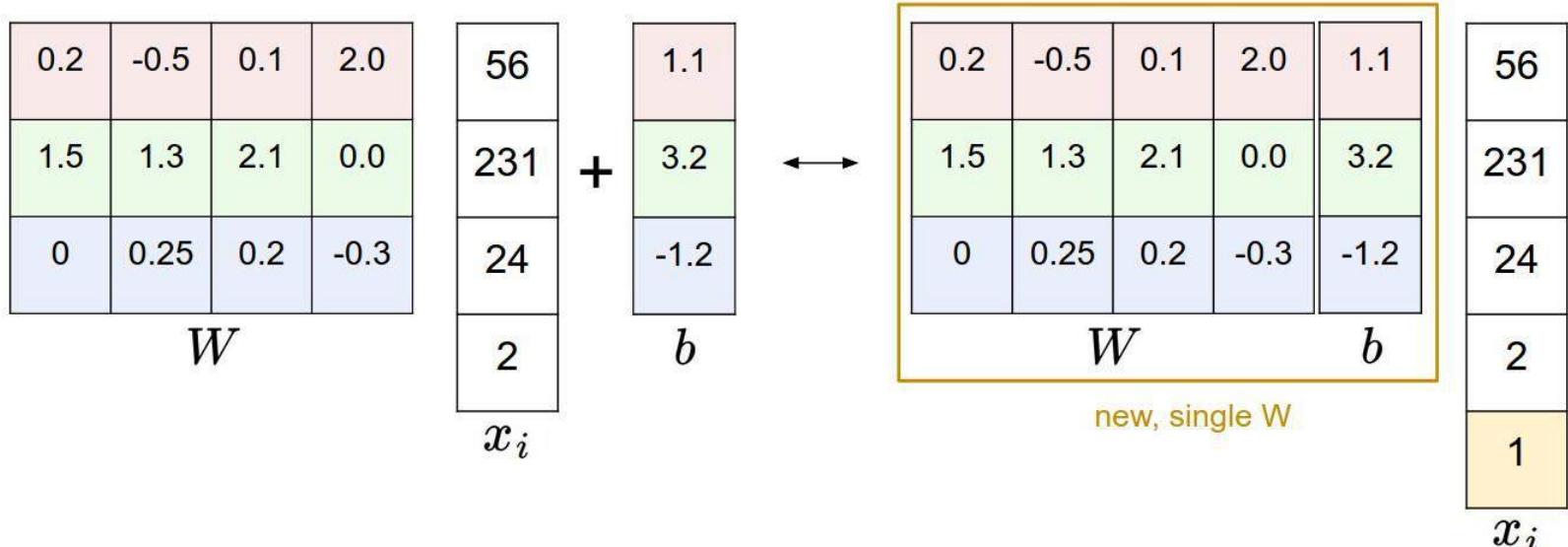
One row per class

# Linear classification

- We can rewrite the score function as:

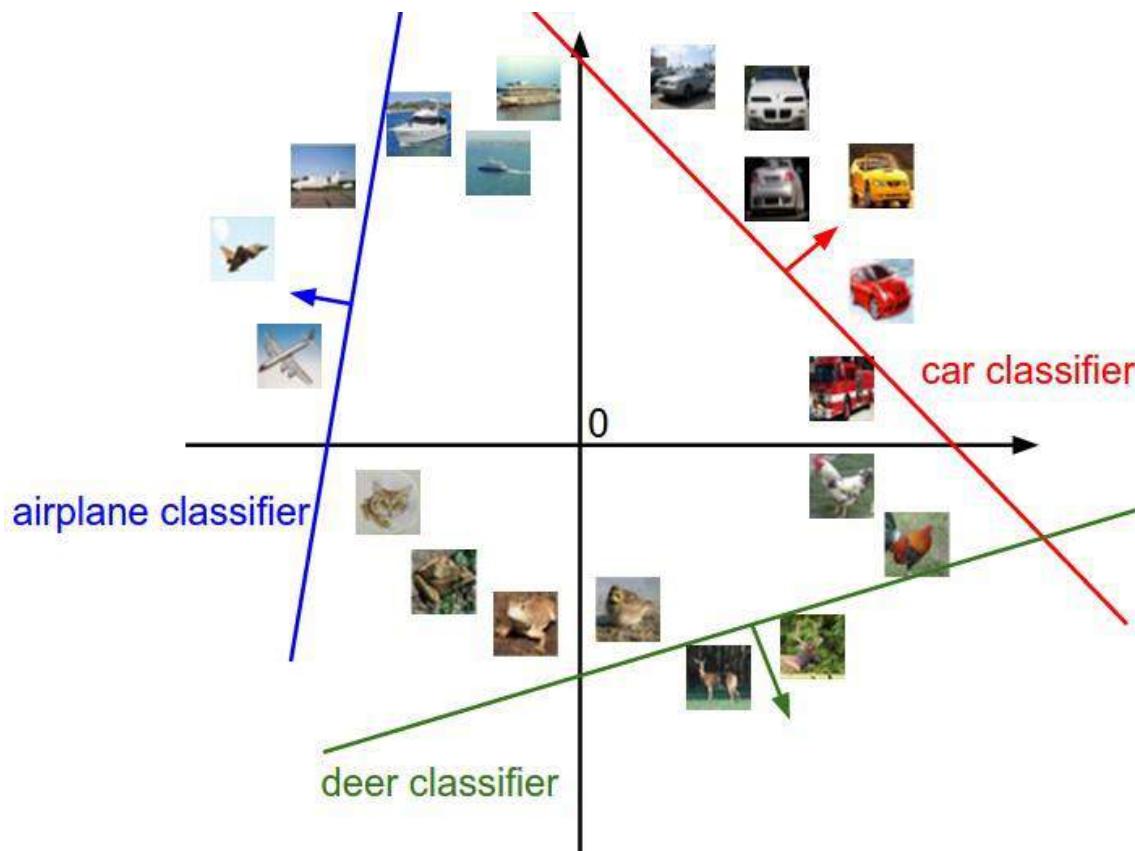
$$f(x_i; W, b) = Wx_i$$

- where the bias is implicitly represented in  $W$  and  $x_i$



# Linear classification: One interpretation

- Since an image can be thought as a vector, we can consider them as points in high-dimensional space.



One interpretation of:

$$f(x_i; W, b) = Wx_i + b$$

- Each row describes a line for a class, and "b"

# Linear classification: Another interpretation

- Each row in  $W$  can be interpreted as a template of that class.
  - $f(x_i; W, b) = Wx_i + b$  calculates the inner product to find which template best fits  $x_i$ .
  - Effectively, we are doing Nearest Neighbor with the “prototype” images of each class.



# Loss function

- A function which measures how good our parameters (weights) are.
  - Other names: cost function, objective function
- Let  $s_j = f(x_i; W)_j$
- An example loss function:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

Or equivalently:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

- This forces the distances to other classes to be more than  $\Delta$  (the margin)



# Example

- Consider our scores for  $x_i$  to be  $s = [13, -7, 11]$  and assume  $\Delta$  as 10.
- Then,

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10)$$

# Regularization

- In practice, there are many possible solutions leading to the same loss value.
  - Based on the requirements of the problem, we might want to penalize certain solutions.
- E.g.,

$$R(W) = \sum_i \sum_j W_{i,j}^2$$

- which penalizes large weights.
  - Why do we want to do that?

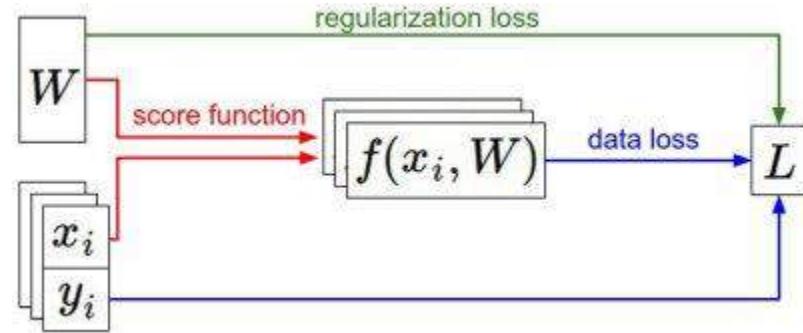
# Combined Loss Function

- The loss function becomes:

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

- If you expand it:

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [\max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)] + \lambda \sum_i \sum_j W_{i,j}^2$$

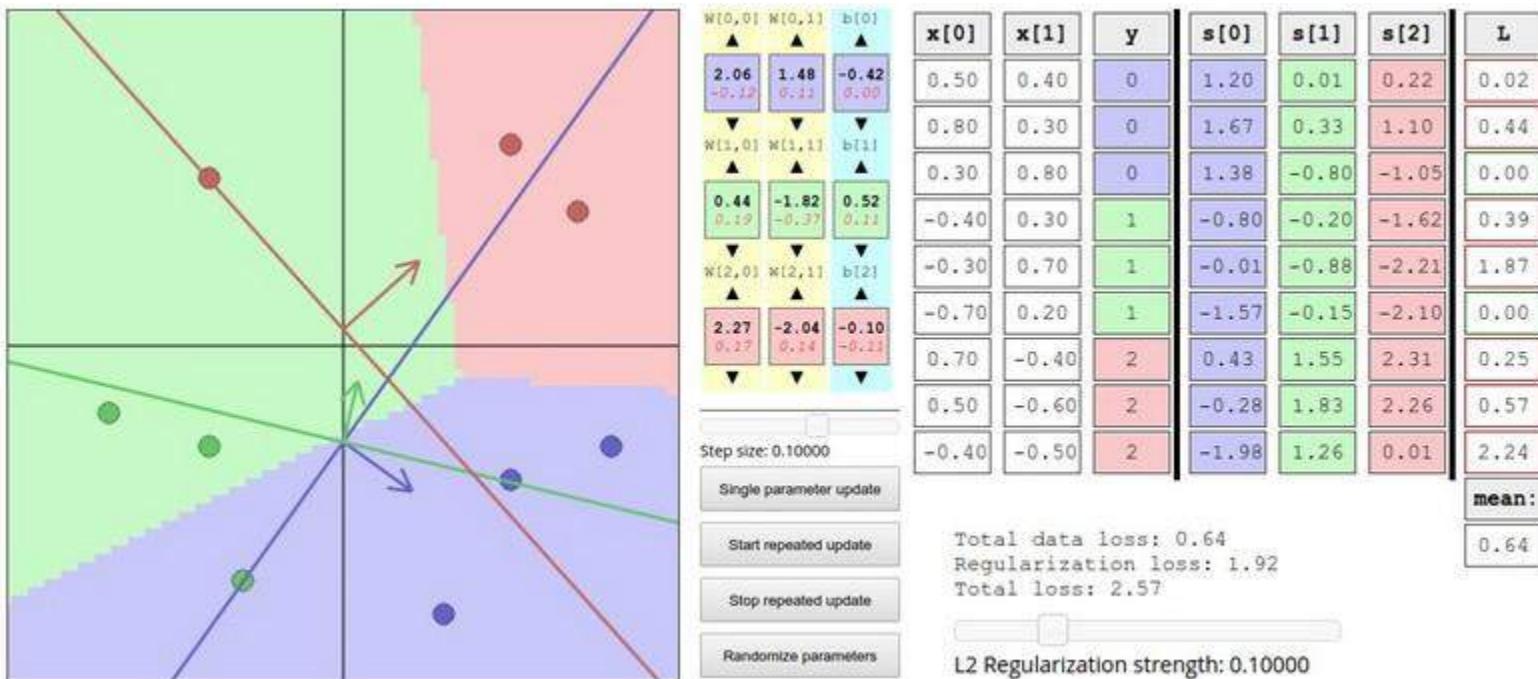


Hyper parameters  
(estimated using validation set)

# Hinge Loss, or Max-Margin Loss

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [\max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)] + \lambda \sum_i \sum_j W_{i,j}^2$$

# Interactive Demo



An alternative formulation of

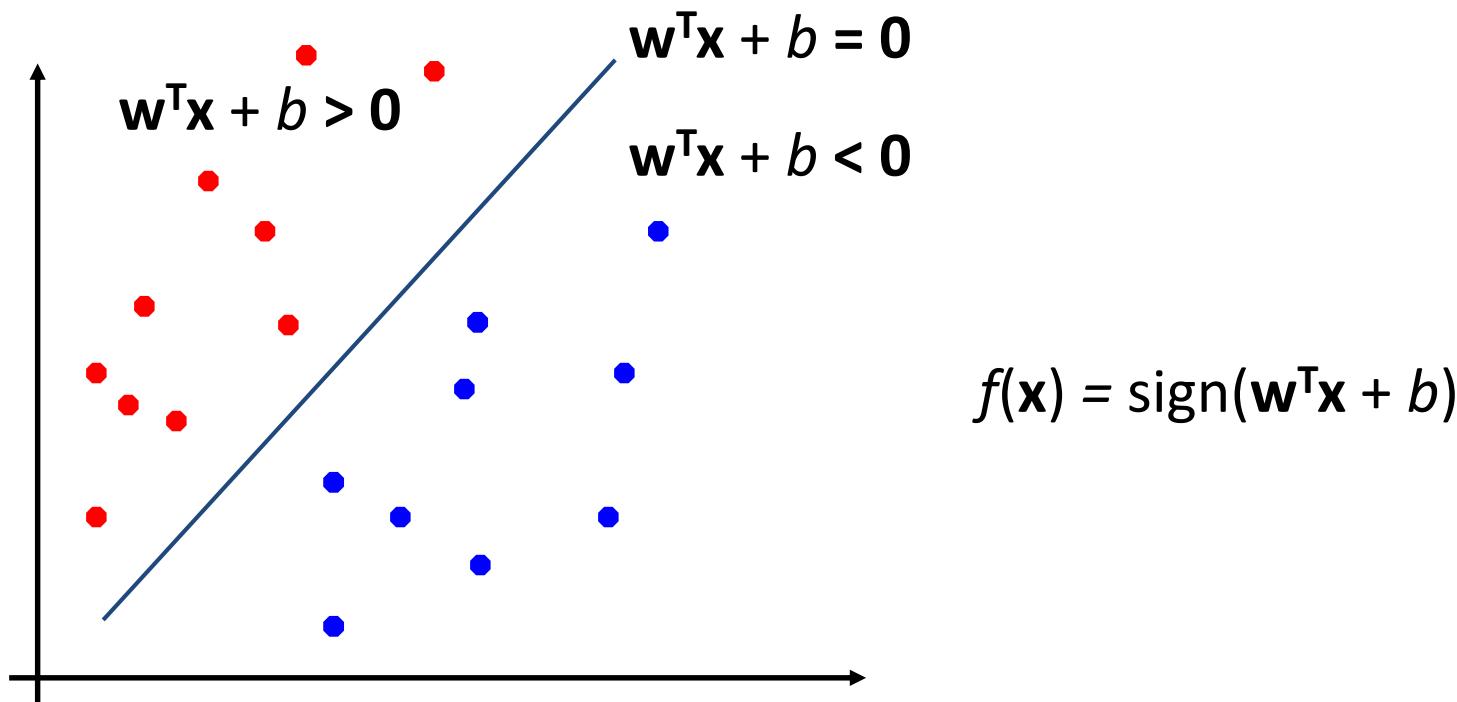
# **SUPPORT VECTOR MACHINES**

Barrowed mostly from the slides of:

- Machine Learning Group, University of Texas at Austin.
- Mingyue Tan, The University of British Columbia

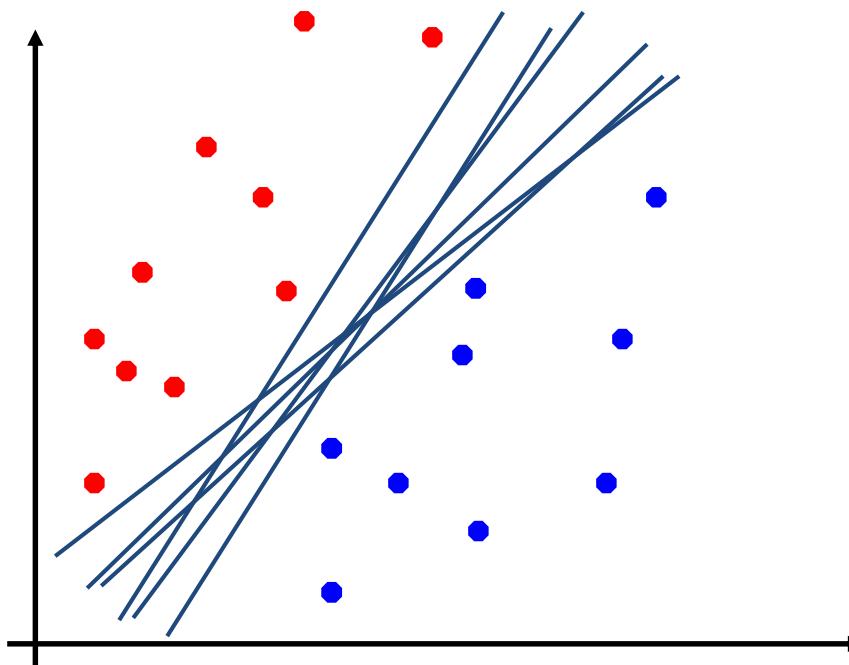
# Linear Separators

- Binary classification can be viewed as the task of separating classes in feature space:



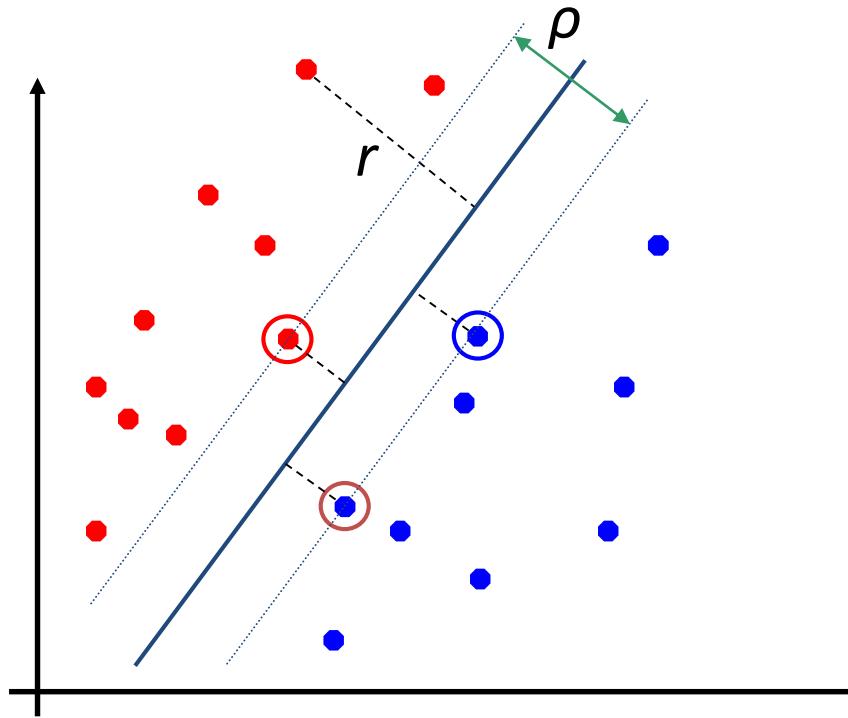
# Linear Separators

- Which of the linear separators is optimal?



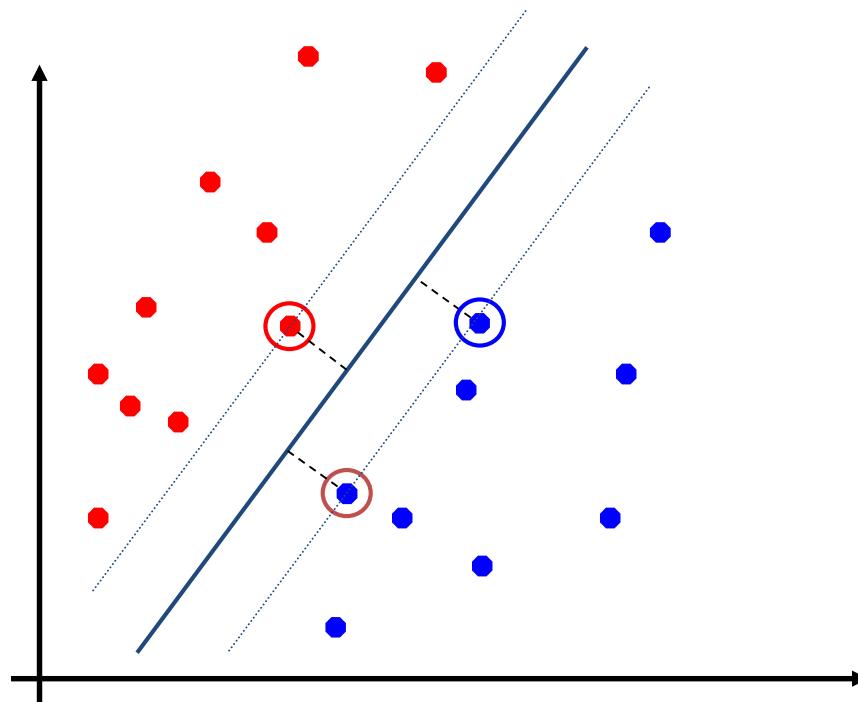
# Classification Margin

- Distance from example  $\mathbf{x}_i$  to the separator is  $r = \frac{\mathbf{w}^T \mathbf{x}_i + b}{\|\mathbf{w}\|}$
- Examples closest to the hyperplane are ***support vectors***.
- ***Margin*  $\rho$**  of the separator is the distance between support vectors.

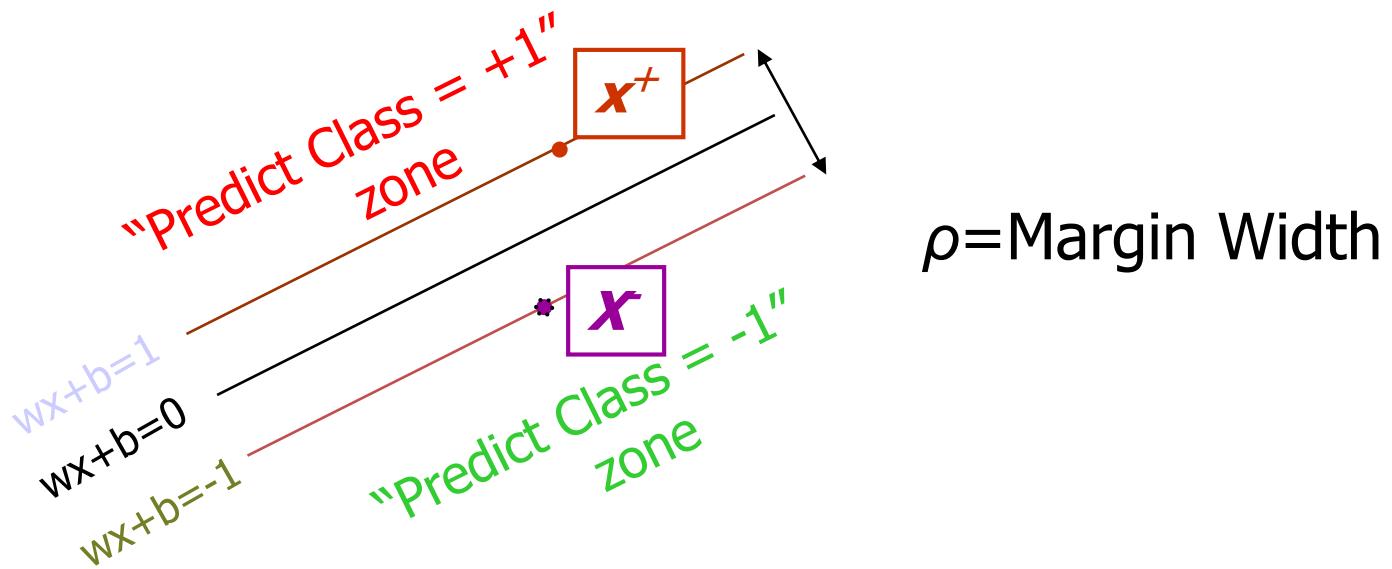


# Maximum Margin Classification

- Maximizing the margin is good according to intuition.
- Implies that only support vectors matter; other training examples are ignorable.



# Linear SVM Mathematically



What we know:

- $w \cdot x^+ + b = +1$
- $w \cdot x^- + b = -1$
- $w \cdot (x^+ - x^-) = 2$

$$\rho = \frac{(x^+ - x^-) \cdot w}{|w|} = \frac{2}{|w|}$$

# Linear SVMs Mathematically (cont.)

- Then we can formulate the *optimization problem*:

Find  $\mathbf{w}$  and  $b$  such that

$$\rho = \frac{2}{\|\mathbf{w}\|} \text{ is maximized}$$

$$\text{and for all } (\mathbf{x}_i, y_i), i=1..n : y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$$

Which can be reformulated as:

Find  $\mathbf{w}$  and  $b$  such that

$$\Phi(\mathbf{w}) = \| \mathbf{w} \| ^2 = \mathbf{w}^T \mathbf{w} \text{ is minimized}$$

$$\text{and for all } (\mathbf{x}_i, y_i), i=1..n : y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$$

# Lagrange Multipliers

- Given the following optimization problem:  
**minimize**  $f(x, y)$  **subject to**  $g(x, y) = c$
- We can formulate it as:  
$$L(x, y, \lambda) = f(x, y) + \lambda(g(x, y) - c)$$
- and set the derivative to zero:  
$$\nabla_{x,y,\lambda} L(x, y, \lambda) = 0$$

# Lagrange Multipliers

- Main intuition:
  - The gradients of  $f$  and  $g$  are parallel at the maximum

$$\nabla f = \lambda \nabla g$$

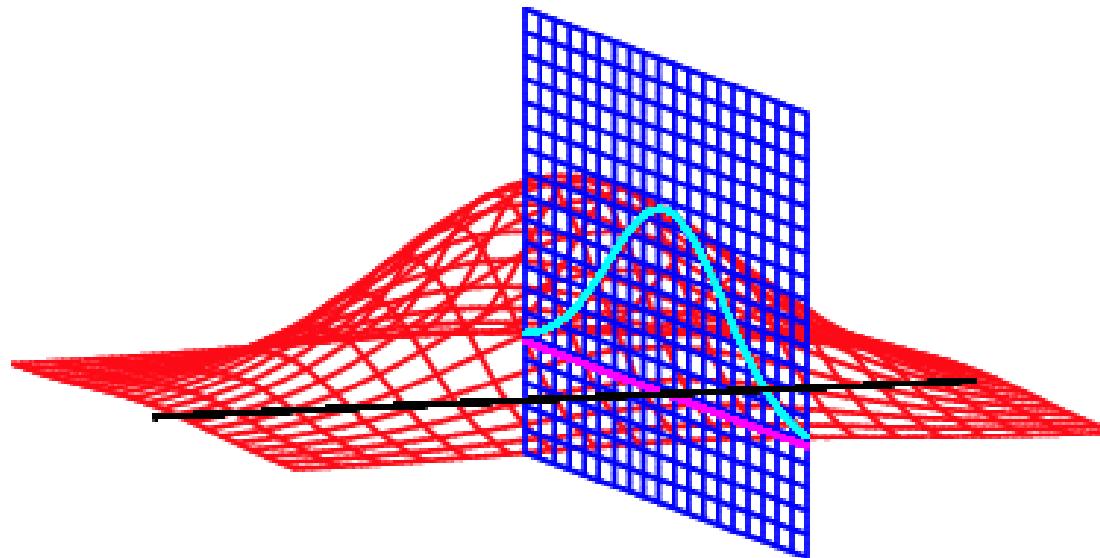


Fig: <http://mathworld.wolfram.com/LagrangeMultiplier.html>

# Lagrange Multipliers

- See the following for proof
  - [http://ocw.mit.edu/courses/mathematics/18-02sc-multivariable-calculus-fall-2010/2.-partial-derivatives/part-c-lagrange-multipliers-and-constrained-differentials/session-40-proof-of-lagrange-multipliers/MIT18\\_02SC\\_notes\\_22.pdf](http://ocw.mit.edu/courses/mathematics/18-02sc-multivariable-calculus-fall-2010/2.-partial-derivatives/part-c-lagrange-multipliers-and-constrained-differentials/session-40-proof-of-lagrange-multipliers/MIT18_02SC_notes_22.pdf)
- A clear example:
  - <http://tutorial.math.lamar.edu/Classes/CalcIII/LagrangeMultipliers.aspx>
- More intuitive explanation:
  - <http://www.slimy.com/~steuard/teaching/tutorials/Lagrange.html>

- In the SVM problem the Lagrangian is

$$L_P \equiv \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^l \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{w} + b) + \sum_{i=1}^l \alpha_i$$

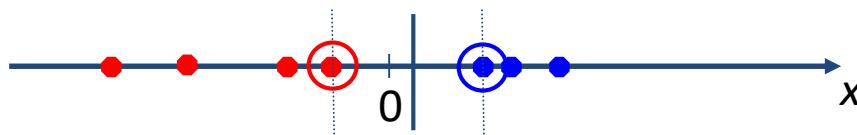
$$\alpha_i \geq 0, \forall i$$

- From the derivatives = 0 we get

$$\mathbf{w} = \sum_{i=1}^l \alpha_i y_i \mathbf{x}_i, \sum_{i=1}^l \alpha_i y_i = 0$$

# Non-linear SVMs

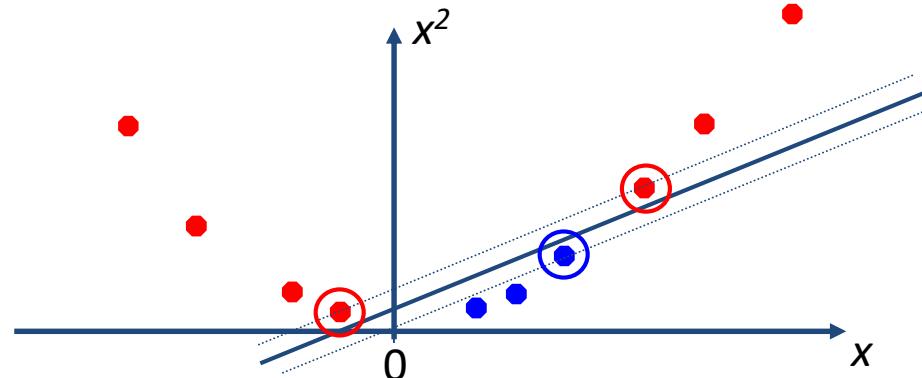
- Datasets that are linearly separable with some noise work out great:



- But what are we going to do if the dataset is just too hard?



- How about... mapping data to a higher-dimensional space:



# **SOFTMAX OR LOGISTIC CLASSIFIERS**

# Softmax classifier – cross-entropy loss

- Cross-entropy:  $H(p, q) = E_p[-\log q] = -\sum_j p_j \log q_j$
- In our case,
  - $p$  denotes the correct probabilities of the categories. In other words,  $p_j = 1$  for the correct label and  $p_j = 0$  for other categories.
  - $q$  denotes the estimated probabilities of the categories
- But, our scores are not probabilities!
  - One solution: Softmax function:  $f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$
  - It maps arbitrary ranges to probabilities
- Using the normalized values, we can define the cross-entropy loss for classification problem now:

$$L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) = -f_{y_i} + \log \sum_j e^{f_j}$$

# logistic loss

- A special case of cross-entropy for binary classification:

$$H(p, q) = - \sum_j p_j \log q_j = -p_j \log q_j - (1 - p_j) \log(1 - q_j)$$

- Softmax function reduces to the logistic function (see [1] for the derivation):

$$\frac{1}{1 + e^{-x}}$$

- And the loss becomes:

$$L_i = -\log\left(\frac{1}{1 + e^{-f}}\right)$$

# Why take logarithm of probabilities?

- Maps probability space to logarithmic space
- Multiplication becomes addition
  - Multiplication is a very frequent operation with probabilities
- Speed:
  - Addition is more efficient
- Accuracy:
  - Considering loss in representing real numbers, addition is friendlier
- Since log-probability is negative, to work with positive numbers, we usually negate the log-probability

# Softmax classifier: One interpretation

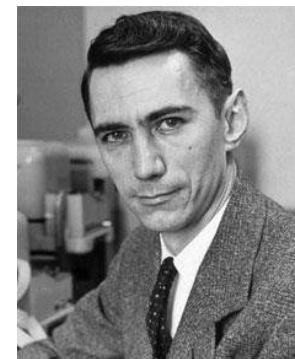
- Information theory
  - Cross-entropy between a true distribution and an estimated one:

$$H(p, q) = - \sum_x p(x) \log q(x).$$

- In our case,  $p = [0, \dots, 1, 0, \dots 0]$ , containing only one 1, at the correct label.
- Since  $H(p, q) = H(p) + D_{KL}(p||q)$ , we are minimizing the Kullback-Leibler divergence.

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}.$$

# Information Entropy



Claude Elwood Shannon  
(1916-2001)

*("A Mathematical Theory of Communication", 1948)*

- Number of bits to represent a coin-pair:

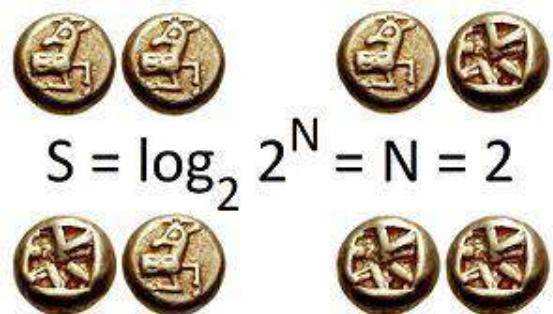
$$\log_2 4 = 2$$

- In fact, this is:

$$\log_2 \frac{1}{p_{coin}} = \log_2 \frac{1}{0.25} = 2$$

- Optimal number of bits to represent an event with probability  $p$ :

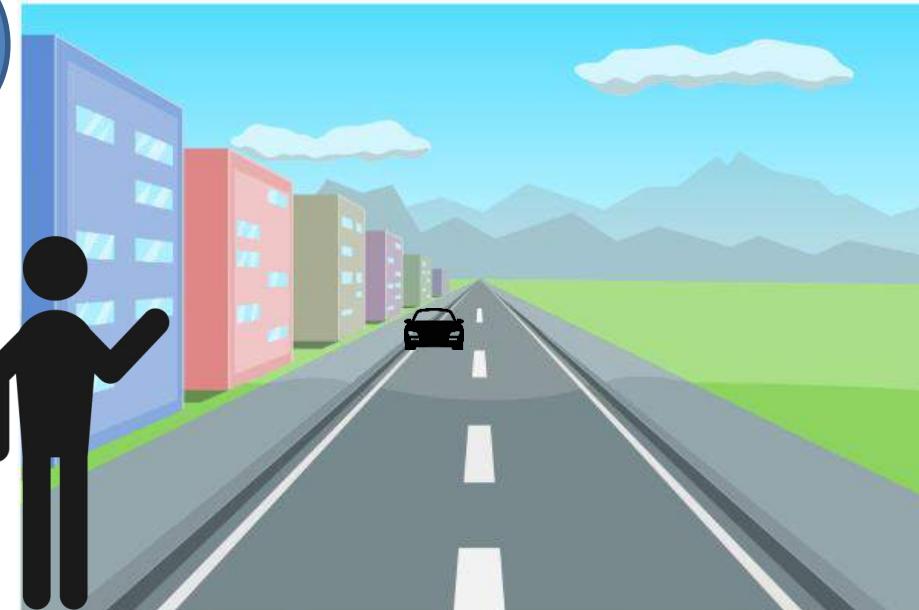
$$\log_2 \frac{1}{p}$$



$$S = \log_2 2^N = N = 2$$

1 : Tesla  
01 : Mazda  
10111: Tofaş

# Cross-entropy, Entropy



- Assume that each bit is expensive
  - So, we are interested in the minimal/optimal coding

# Cross-entropy, Entropy



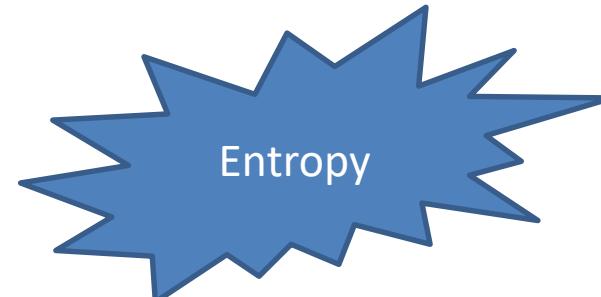
- For an optimal setting, we can assign bits to code information based on their probabilities
- The smallest number of bits on avg. to represent an event with probability  $p$ :  $\log 1/p$
- Optimal bits to represent tofaş cars:

$$b_{\text{tofas}} = \log \frac{1}{p_{\text{tofaş}}}$$

- The optimal encoding then requires:

$$H(p) = \sum_i p_i \log \frac{1}{p_i} = - \sum_i p_i \log p_i$$

Car	Probability
Tofaş	0.8
Mazda	0.15
Tesla	0.05

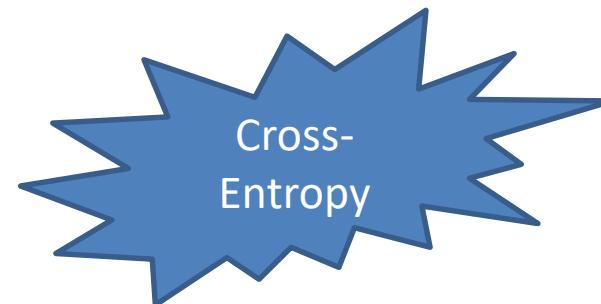


# Cross-entropy, Entropy

- Entropy assumes that the data follows the «correct» distribution.
- If the estimated/current distribution (call it  $q$ ) is somewhat “wrong”, how can we quantify the number of bits required?

$$H(p) = \sum_i p_i \log \frac{1}{p_i} = - \sum_i p_i \log p_i$$

$$H(p, q) = \sum_i p_i \log \frac{1}{q_i} = - \sum_i p_i \log q_i$$



# Kullback-Leibler Divergence

- Difference between cross-entropy and entropy  
(this is zero when  $p_i$  equals  $q_i$ ):

$$\begin{aligned} KL(p \parallel q) &= \sum_i p_i \log \frac{1}{q_i} - \sum_i p_i \log \frac{1}{p_i} \\ &= \sum_i p_i \log \frac{p_i}{q_i} \end{aligned}$$

# More on xentropy, entropy and KL-divergence

- <https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/>
- <https://www.youtube.com/watch?v=ErfnhcEV108>

# Softmax classifier: Another interpretation

- Probabilistic view

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}.$$

- In our case, we are minimizing the negative log likelihood.
- Therefore, this corresponds to Maximum Likelihood Estimation (MLE).

# Numerical Stability

- Exponentials may become very large. A trick:

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C \sum_j e^{f_j}} = \frac{e^{f_{y_i} + \log C}}{\sum_j e^{f_j + \log C}}$$

- Set  $\log C = -\max_j f_j$ .

See the following link for more information:

<http://www.nowozin.net/sebastian/blog/streaming-log-sum-exp-computation.html>

# SVM loss vs. cross-entropy loss

- SVM is happy when the classification satisfies the margin
  - Ex: if score values = [10, 9, 9] or [10, -10, -10]
    - SVM loss is happy if the margin is 1
  - SVM is local
- cross-entropy always wants better
  - cross-entropy is global

# A critical look at softmax

- Be Careful What You Backpropagate: A Case For Linear Output Activations & Gradient Boosting

<https://arxiv.org/pdf/1707.04199.pdf>

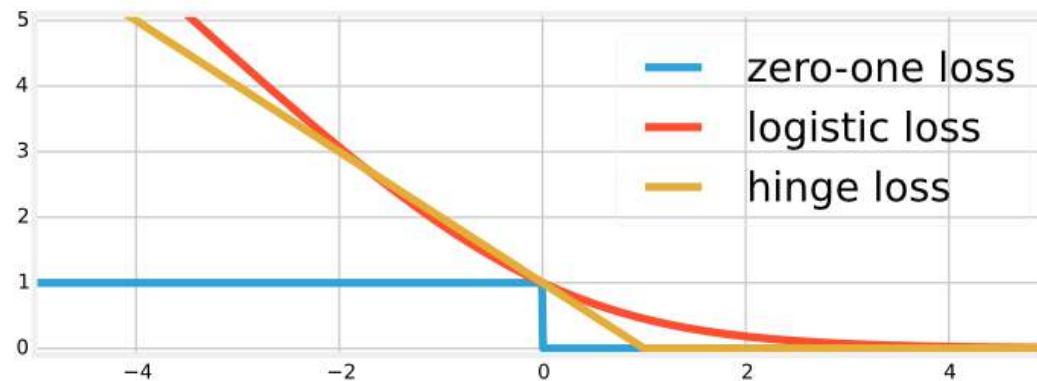
# 0-1 Loss

- Minimize the # of cases where the prediction is wrong:

$$L = \sum_i \mathbf{1}(f(x_i; W, b)_{y_i} \neq \hat{y}_i)$$

Or equivalently,

$$L = \sum_i \mathbf{1}(\hat{y}_i f(x_i; W, b)_{y_i} < 0)$$



# Absolute Value Loss, Squared Error Loss

$$L_i = \sum_j |s_j - y_j|^q$$

- $q = 1$ : absolute value loss
- $q = 2$ : square error loss.

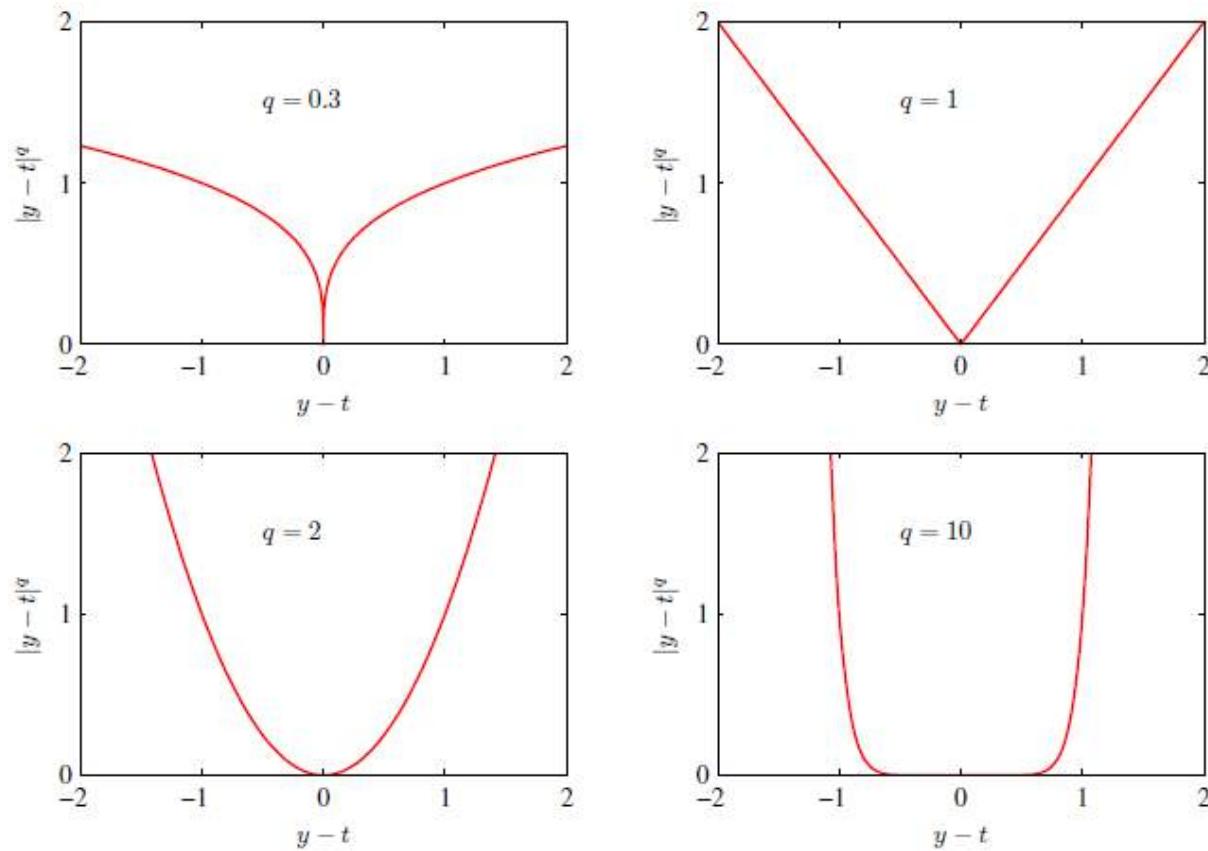


Figure 1.29 Plots of the quantity  $L_q = |y - t|^q$  for various values of  $q$ .

Bishop

# **MORE ON LOSS FUNCTIONS**

# Constraint for loss functions

- For a function to be loss function, it should be a Lipschitz function.
- A function  $f$  is a Lipschitz function iff.  
$$|f(x) - f(y)| \leq C|x - y|$$
for all  $x, y$ ;  $C$  is a constant independent of  $x$  and  $y$ .
- Any function with bounded first derivative must be Lipschitz.

# Visualizing Loss Functions

- If you look at one of the example loss functions:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + 1)$$

- Since  $W$  has too many dimensions, this is difficult to plot.
- We can visualize this for one weight direction though, which can give us some intuition about the shape of the function.
  - E.g., start from an arbitrary  $W_0$ , choose a direction  $W_1$  and plot  $L(W_0 + \alpha W_1)$  for different values of  $\alpha$ .

# Visualizing Loss Functions

- Example:

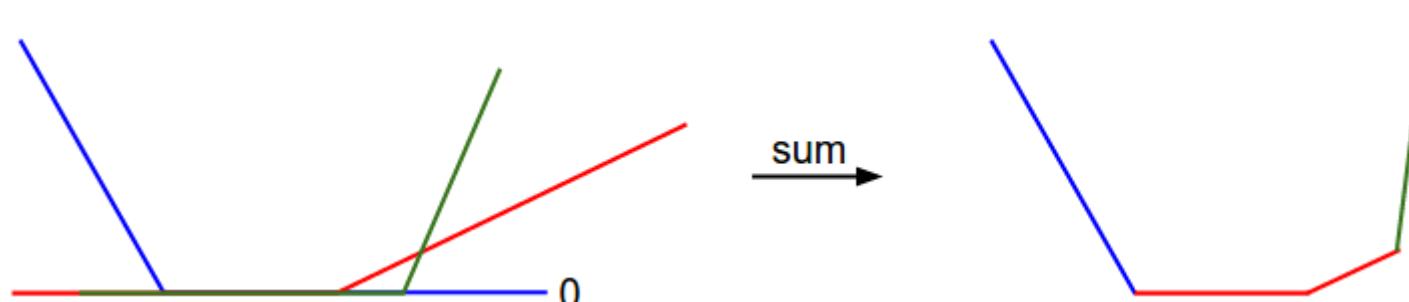
$$L_0 = \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1)$$

$$L_1 = \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1)$$

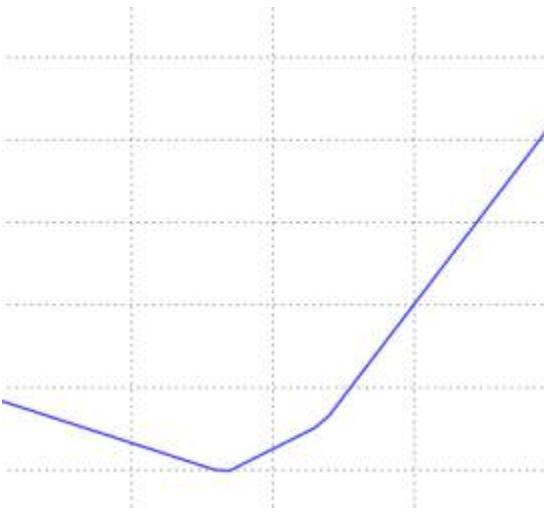
$$L_2 = \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1)$$

$$L = (L_0 + L_1 + L_2)/3$$

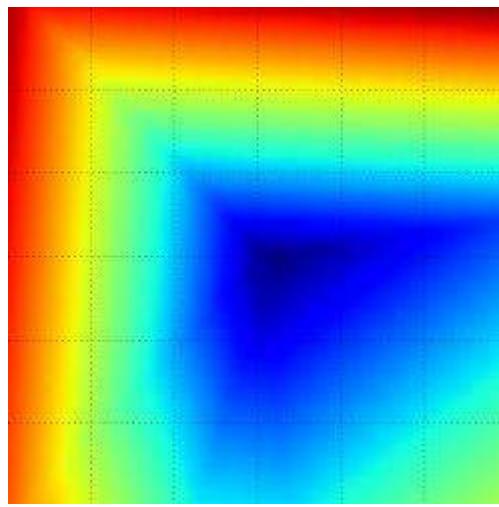
- If we consider just  $w_0$ , we have many linear functions in terms of  $w_0$  and loss is a linear combination of them.



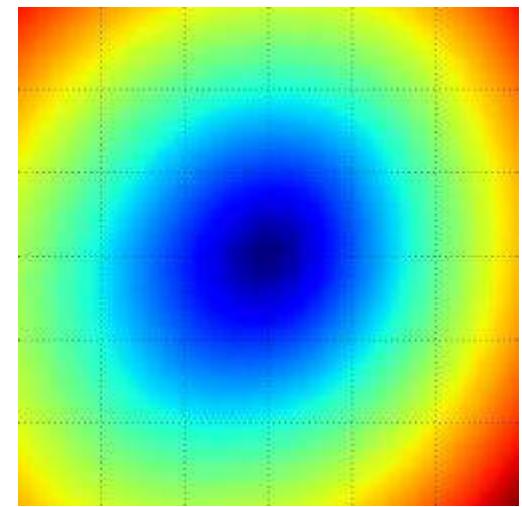
# Visualizing Loss Functions



Loss along one direction



Loss along two directions



Loss along two directions  
(averaged over many samples)

- You see that this is a convex function.
  - Nice and easy for optimization
- When you combine many of them in a neural network, it becomes non-convex.

# Another approach for visualizing loss functions

- 0-1 loss:

$$L = \mathbb{1}(f(x) \neq y)$$

or equivalently as:

$$L = \mathbb{1}(yf(x) > 0)$$

- Square loss:

$$L = (f(x) - y)^2$$

in binary case:

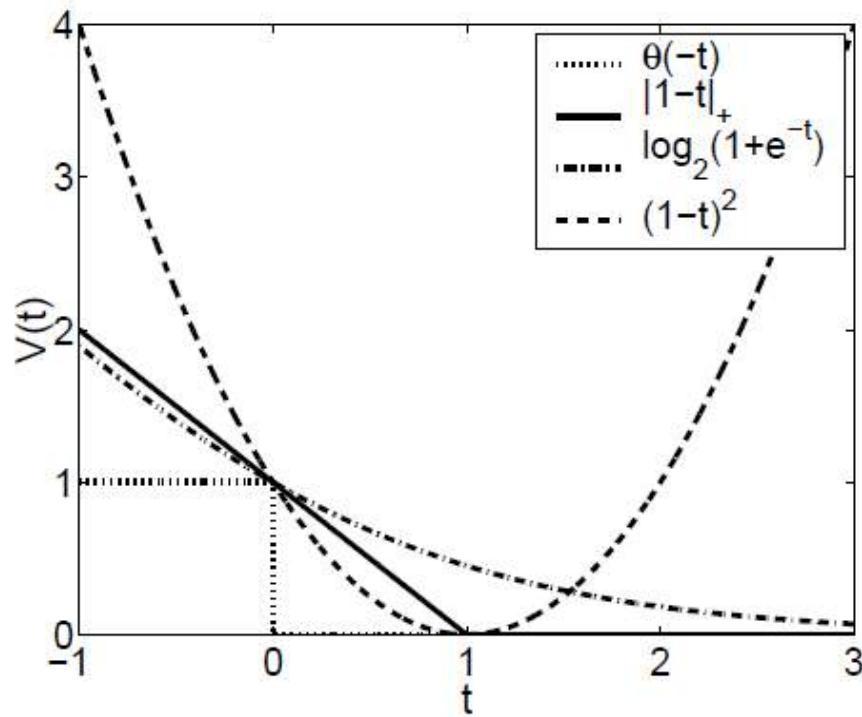
$$L = (1 - yf(x))^2$$

- Hinge-loss

$$L = \max(1 - yf(x), 0)$$

- Logistic loss:

$$L = (\ln 2)^{-1} \ln(1 + e^{-yf(x)})$$



Various loss functions used in classification. Here  $t = yf(x)$ .

Rosacco et al., 2003

# **SUMMARY OF LOSS FUNCTIONS**

# Linear Classifier: SVM

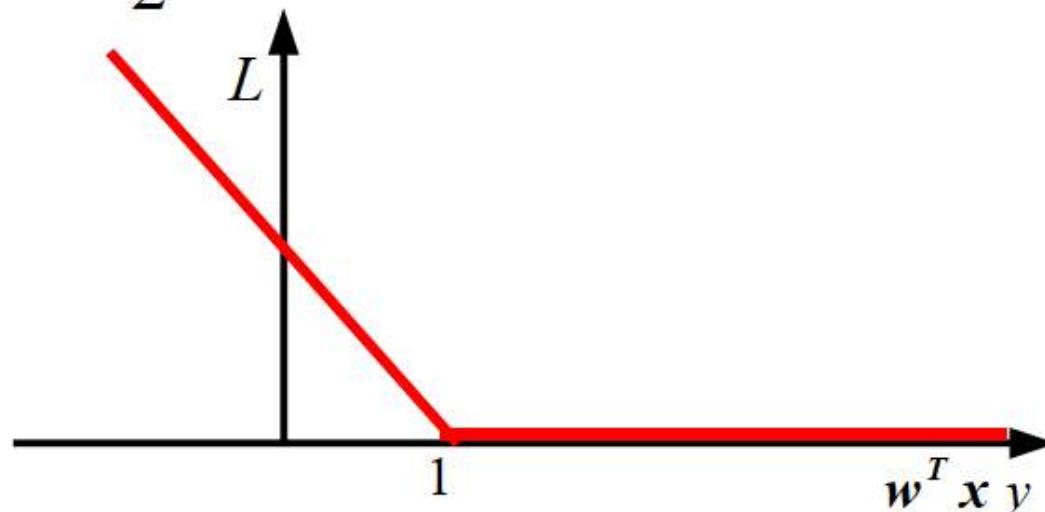
Input:  $\mathbf{x} \in R^D$

Binary label:  $y \in \{-1, +1\}$

Parameters:  $\mathbf{w} \in R^D$

Output prediction:  $\mathbf{w}^T \mathbf{x}$

Loss:  $L = \frac{1}{2} \|\mathbf{w}\|^2 + \lambda \max [0, 1 - \mathbf{w}^T \mathbf{x} y]$



Hinge Loss

# Linear Classifier: Logistic Regression

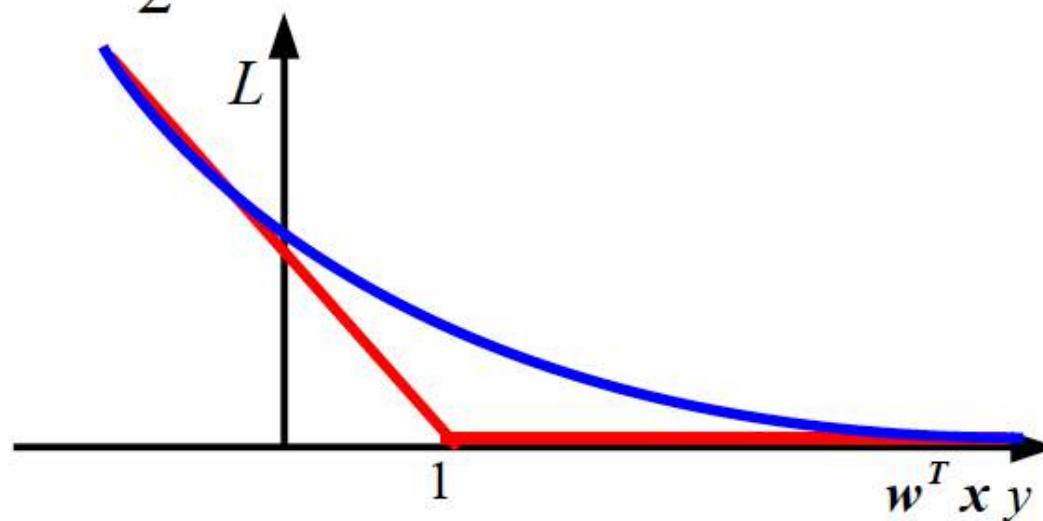
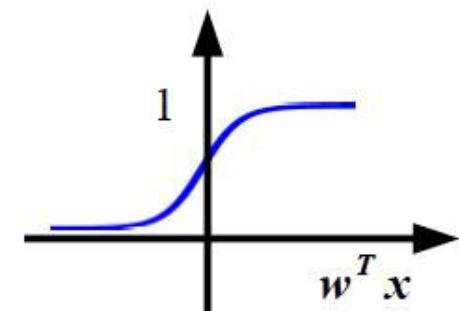
Input:  $\mathbf{x} \in R^D$

Binary label:  $y \in \{-1, +1\}$

Parameters:  $\mathbf{w} \in R^D$

Output prediction:  $p(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T \mathbf{x}}}$

Loss:  $L = \frac{1}{2} \|\mathbf{w}\|^2 - \lambda \log(p(y|\mathbf{x}))$



Log Loss

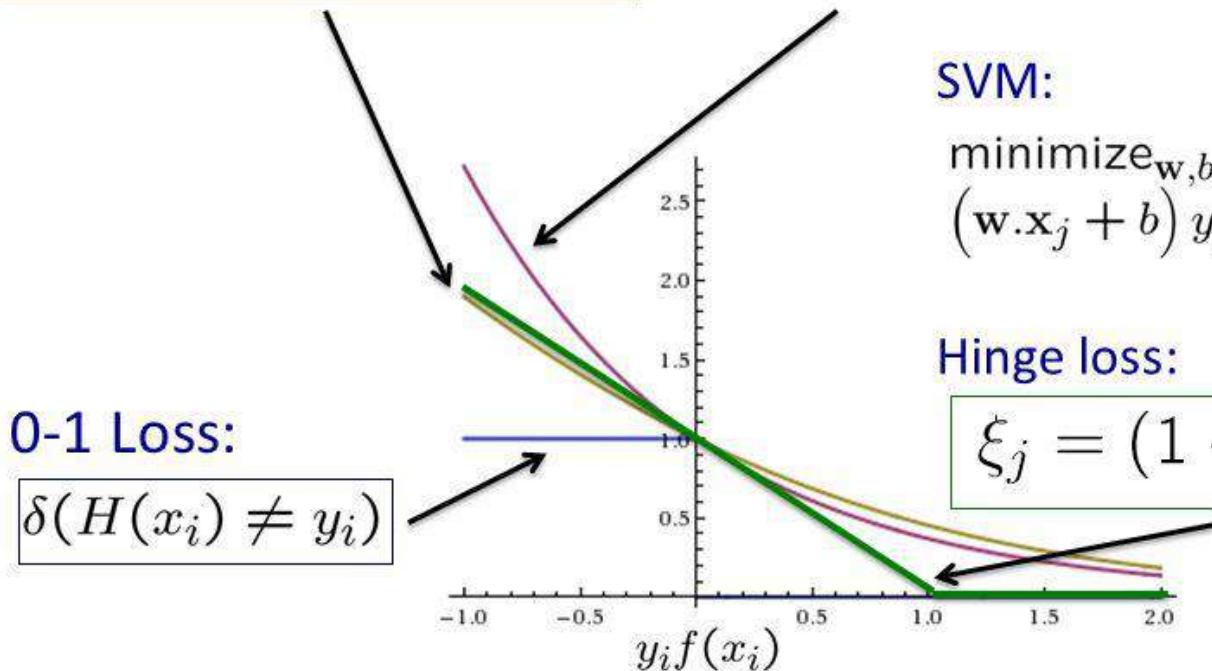
## Side Note: Different Losses

Logistic regression:

$$\sum_{i=1}^m \ln(1 + \exp(-y_i f(x_i)))$$

Boosting :

$$\frac{1}{m} \sum_i \exp(-y_i f(x_i)) = \prod_t Z_t$$



SVM:

$$\begin{aligned} & \text{minimize}_{\mathbf{w}, b} \quad \mathbf{w} \cdot \mathbf{w} + C \sum_j \xi_j \\ & (\mathbf{w} \cdot \mathbf{x}_j + b) y_j \geq 1 - \xi_j, \quad \forall j \\ & \xi_j \geq 0, \quad \forall j \end{aligned}$$

Hinge loss:

$$\xi_j = (1 - f(x_i) y_i)_+$$

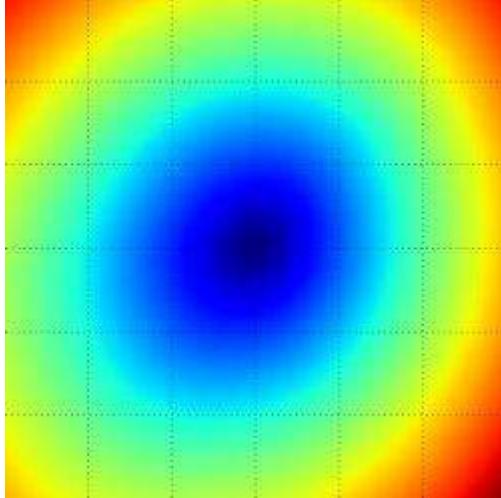
0-1 Loss:

$$\delta(H(x_i) \neq y_i)$$

All our new losses approximate 0/1 loss!

# Sum up

- 0-1 loss is not differentiable/helpful at training
  - It is used in testing
- Other losses try to cover the “weakness” of 0-1 loss
- Hinge-loss imposes weaker constraint compared to cross-entropy
- For classification: use hinge-loss or cross-entropy loss
- For regression: use squared-error loss, or absolute difference loss



**SO, WHAT DO WE DO WITH A LOSS  
FUNCTION?**

# Optimization strategies

- We want to find  $W$  that minimizes the loss function.
  - Remember that  $W$  has lots of dimensions.
- Naïve idea: random search
  - For a number of iterations:
    - Select a  $W$  randomly
    - If it leads to better loss than the previous ones, select it.
  - This yields 15.5% accuracy on CIFAR after 1000 iterations (chance: 10%)

# Optimization strategies

- Second idea: random local search
  - Start at an arbitrary position (weight  $W$ )
  - Select an arbitrary direction  $W + \delta W$  and see if it leads to a better loss
    - If yes, move along
  - This leads to **21.4%** accuracy after 1000 iterations
  - This is actually a variation of simulated annealing
- A better idea: follow the gradient

# A quick reminder on gradients / partial derivatives

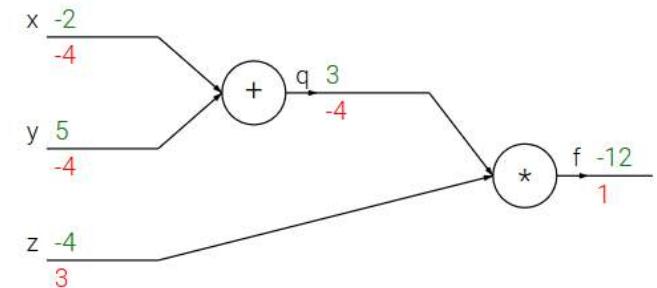
- In one dimension:  $\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h)-f(x)}{h}$
- In practice:
  - $\frac{df[n]}{dn} = \frac{f[n+h]-f[n]}{h}$
  - $\frac{df[n]}{dn} = \frac{f[n+h]-f[n-h]}{2h}$  (centered difference – works better)
- In many dimensions:
  1. Compute gradient numerically with finite differences
    - Slow
    - Easy
    - Approximate
  2. Compute the gradient analytically
    - Fast
    - Exact
    - Error-prone to implement
- In practice: implement (2) and check against (1) before testing

# A quick reminder on gradients / partial derivatives

- If you have a many-variable function, e.g.,  $f(x, y) = x + y$ , you can take its derivative wrt either  $x$  or  $y$ :
  - $\frac{df(x,y)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h,y)-f(x,y)}{h} = 1$
  - Similarly,  $\frac{df(x,y)}{dy} = 1$
  - In fact, we should denote them as follows since they are “partial derivatives” or “gradients on x or y”:
    - $\frac{\partial f}{\partial x}$  and  $\frac{\partial f}{\partial y}$
- Partial derivative tells you the rate of change along a single dimension at a point.
  - E.g., if  $\partial f / \partial x = 1$ , it means that a change of  $x_0$  in  $x$  leads to the same amount of change in the value of the function.
- Gradient is a vector of partial derivatives:
  - $\nabla f = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]$

# A quick reminder on gradients / partial derivatives

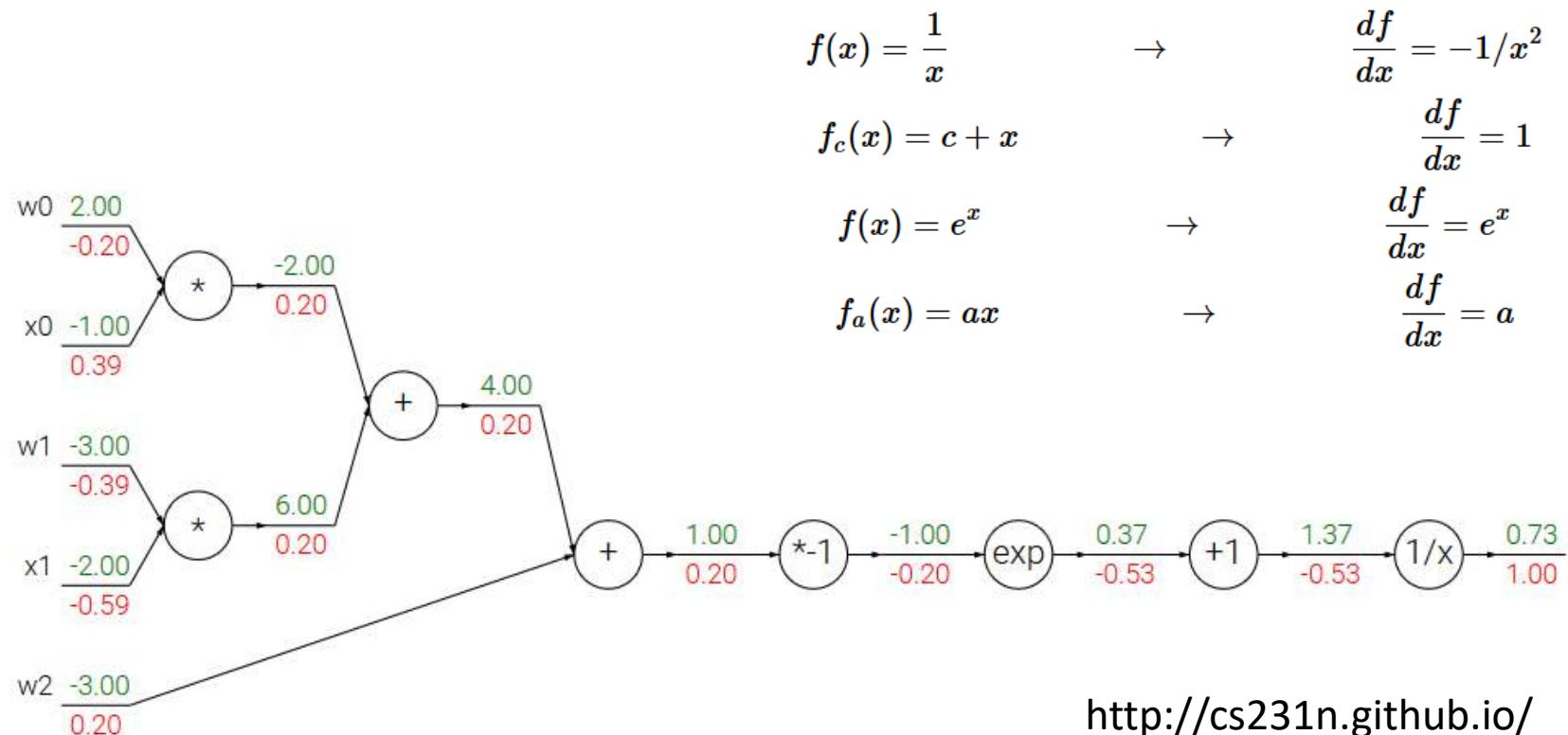
- A simple example:
  - $f(x, y) = \max(x, y)$
  - $\nabla f = [\mathbf{1}(x \geq y), \mathbf{1}(y \geq x)]$
- Chaining:
  - What if we have a composition of functions?
  - E.g.,  $f(x, y, z) = q(x, y)z$  and  $q(x, y) = x + y$
  - $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z$
  - $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = z$
  - $\frac{\partial f}{\partial z} = q(x, y) = x + y$
- Back propagation
  - Local processing to improve the system globally
  - Each gate locally determines to increase or decrease the inputs



# Partial derivatives and backprop

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

- Which has many gates:



# In fact, that was the sigmoid function

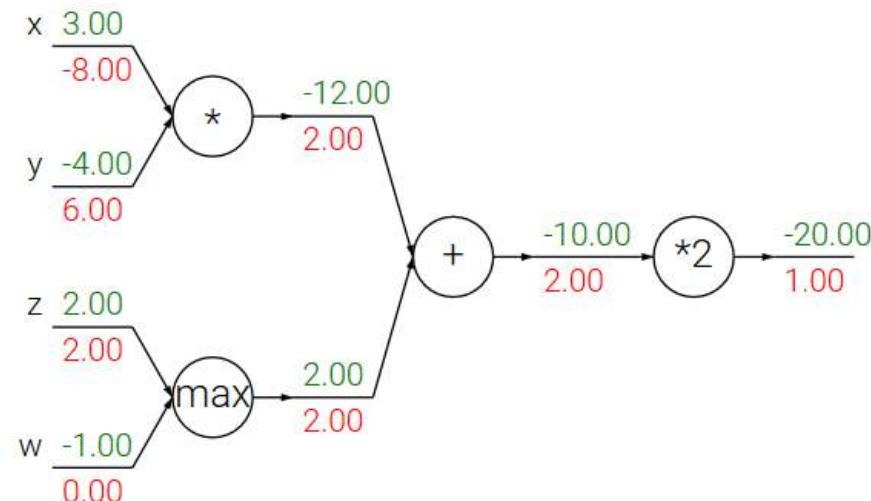
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\rightarrow \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

- We will combine all these gates into a single gate and call it a neuron

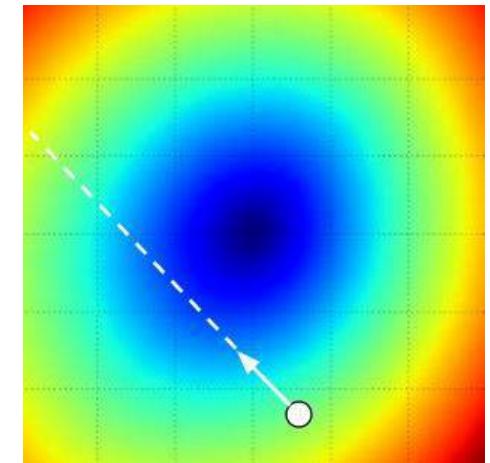
# Intuitive effects

- Commonly used operations
  - Add gate
  - Max gate
  - Multiply gate
- Due to the effect of the multiply gate, when one of the inputs is large, the small input gets the large gradient
  - This has a negative effect in NNs
  - When one of the weights is unusually large, it effects the other weights.
  - Therefore, it is better to normalize the input / weights



# Optimization strategies: gradient

- Move along the negative gradient (since we wish to go down)
  - $W - s \frac{\partial L(W)}{\partial W}$
  - $s$ : step size
- Gradient tells us the direction
  - Choosing how much to move along that direction is difficult to determine
  - This is also called the learning rate
  - If it is small: too slow to converge
  - If it is big: you may overshoot and skip the minimum



# Optimization strategies: gradient

- Take our loss function:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + 1)$$

- Its gradient wrt  $w_j$  is:

$$\frac{\partial L_i}{\partial w_j} = \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + 1 > 0) x_i$$

- For the “winning” weight, this is:

$$\nabla_{w_{y_i}} L_i = - \left( \sum_{j \neq y_i} \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i$$

# Gradient Descent

- Update the weight:

$$W_{new} \leftarrow W - s \frac{\partial L(W)}{\partial W}$$

- This computes the gradient after seeing all examples to update the weight.
  - Examples can be on the order of millions or billions
- Alternative:
  - Mini-batch gradient descent: Update the weights after, e.g., 256 examples
  - Stochastic (or online) gradient descent: Update the weights after each example
  - People usually use batches and call it stochastic.
  - Performing an update after one example for 100 examples is more expensive than performing an update at once for 100 examples due to matrix/vector operations

# A GENERAL LOOK AT OPTIMIZATION

# Mathematical Optimization

Nonlinear Optimization

Convex Optimization

Least-squares

LP

# Mathematical Optimization

---

(mathematical) optimization problem

$$\begin{aligned} & \text{minimize} && f_0(x) \\ & \text{subject to} && f_i(x) \leq b_i, \quad i = 1, \dots, m \end{aligned}$$

- $x = (x_1, \dots, x_n)$ : optimization variables
- $f_0 : \mathbf{R}^n \rightarrow \mathbf{R}$ : objective function
- $f_i : \mathbf{R}^n \rightarrow \mathbf{R}, i = 1, \dots, m$ : constraint functions

**optimal solution**  $x^*$  has smallest value of  $f_0$  among all vectors that satisfy the constraints

# Convex Optimization

---

$$\begin{aligned} & \text{minimize} && f_0(x) \\ & \text{subject to} && f_i(x) \leq b_i, \quad i = 1, \dots, m \end{aligned}$$

- objective and constraint functions are convex:

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)$$

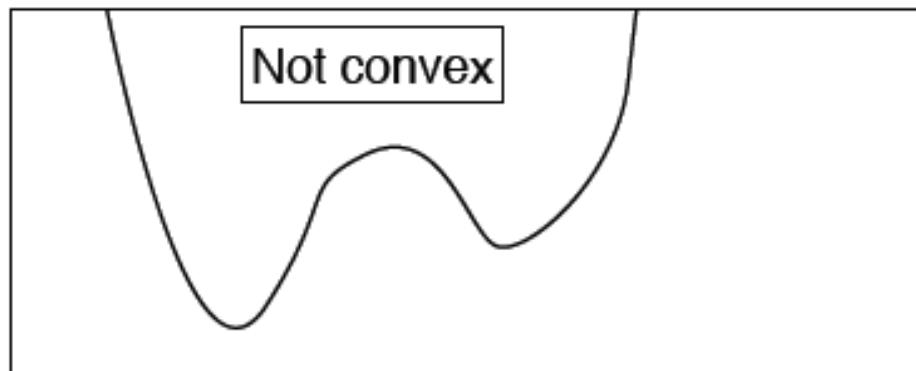
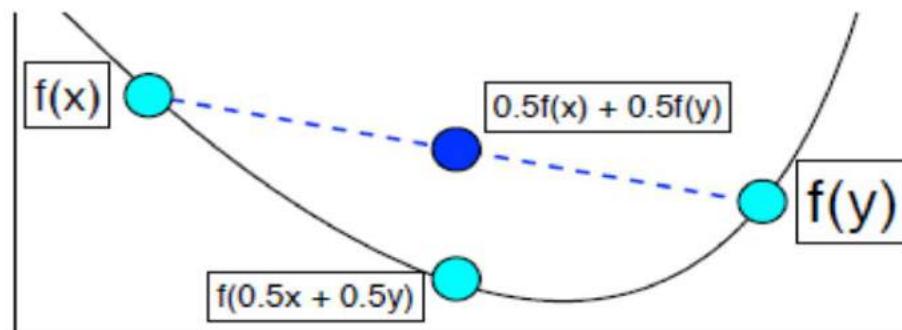
if  $\alpha + \beta = 1$ ,  $\alpha \geq 0$ ,  $\beta \geq 0$

- includes least-squares problems and linear programs as special cases

# Interpretation

---

- Function's value is below the line connecting two points

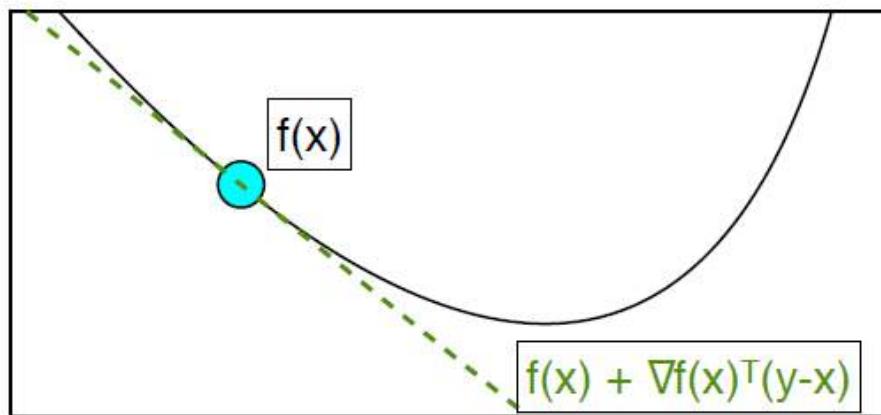


# Another interpretation

A *differentiable function  $f$  is convex if for all  $x$  and  $y$  we have*

$$f(y) \geq f(x) + \nabla f(x)^T(y - x),$$

- The function is globally *above the tangent at  $x$ .*



# Example convex functions

---

Some simple convex functions:

- $f(x) = c$
- $f(x) = a^T x$
- $f(x) = x^T A x$  (for  $A \succeq 0$ )
- $f(x) = \exp(ax)$
- $f(x) = x \log x$  (for  $x > 0$ )
- $f(x) = \|x\|^2$
- $f(x) = \|x\|_p$
- $f(x) = \max_i\{x_i\}$

Some other notable examples:

- $f(x, y) = \log(e^x + e^y)$
- $f(X) = \log \det X$  (for  $X$  positive-definite).
- $f(x, Y) = x^T Y^{-1} x$  (for  $Y$  positive-definite)

# Operations that conserve convexity

---

- ① Non-negative weighted sum:

$$f(x) = \theta_1 f_1(x) + \theta_2 f_2(x).$$

- ② Composition with affine mapping:

$$g(x) = f(Ax + b).$$

- ③ Pointwise maximum:

$$f(x) = \max_i\{f_i(x)\}.$$

---

Show that least-residual problems are convex for any  $\ell_p$ -norm:

$$f(x) = \|Ax - b\|_p$$

We know that  $\|\cdot\|_p$  is a norm, so it follows from (2).

$$\|x + y\|_p \leq \|x\|_p + \|y\|_p$$

Show that SVMs are convex:

$$f(x) = \frac{1}{2}\|x\|^2 + C \sum_{i=1}^n \max\{0, 1 - b_i a_i^T x\}.$$

Know first term is convex, for the other terms use (3) on the two (convex) arguments, then use (1) to put it all together.

# Why convex optimization?

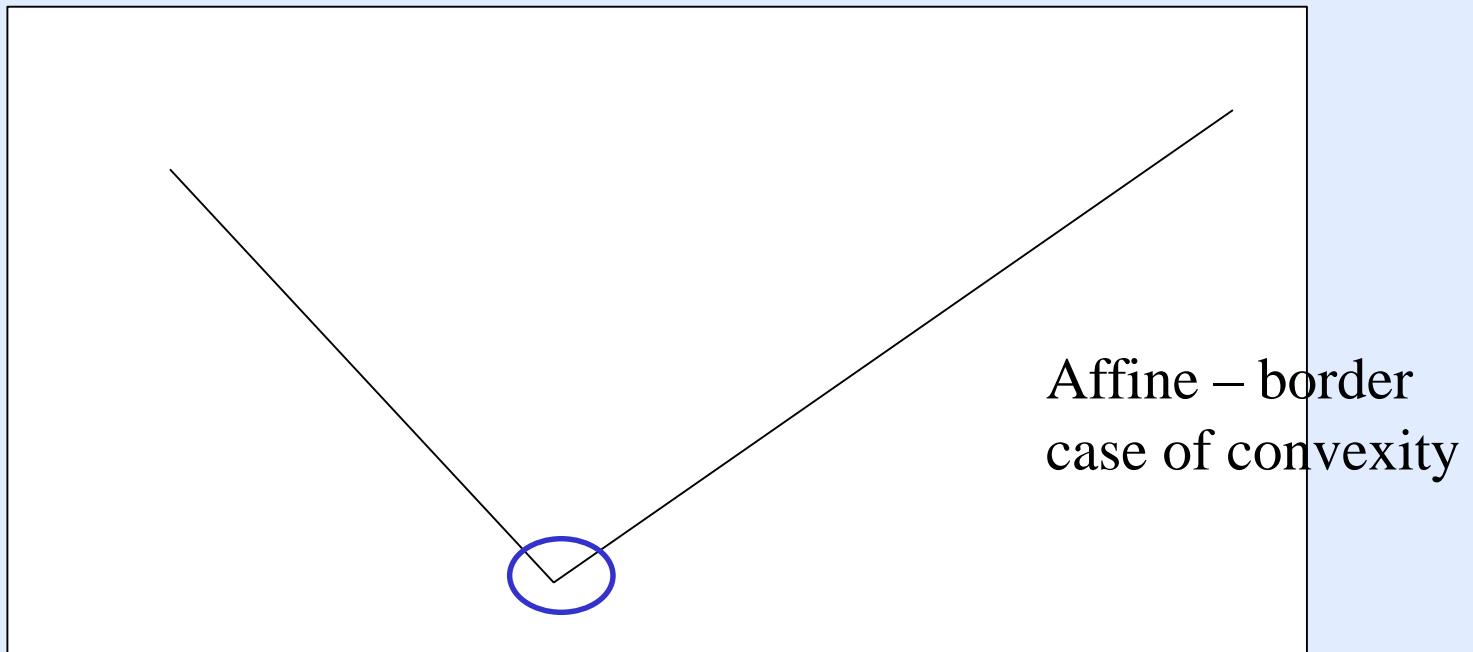
- Can't solve most OPs
  - E.g. NP Hard, even high polynomial time too slow
- Convex OPs
  - (Generally) No analytic solution
  - Efficient algorithms to find (global) solution
  - Interior point methods (basically Iterated Newton) can be used:
    - $\sim [10-100] * \max\{p^3, p^2m, F\}$ ; F cost eval. obj. and constr. f
  - At worst solve with general IP methods (CVX), faster specialized

# Convex Function

---

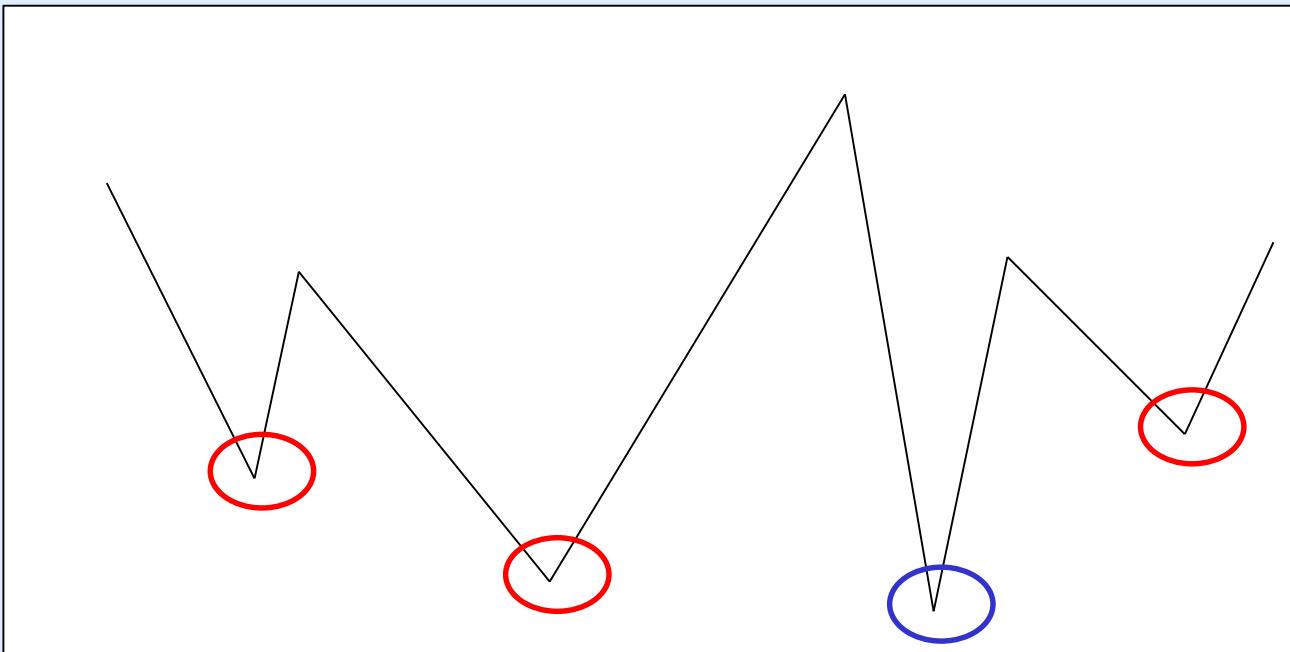
- Easy to see why convexity allows for efficient solution
- Just “slide” down the objective function as far as possible and will reach a minimum

# Convex vs. Non-convex Ex.



- Convex, min. easy to find

# Convex vs. Non-convex Ex.



- Non-convex, easy to get stuck in a local min.
- **Can't rely on only local search techniques**

# Non-convex

---

- Some non-convex problems highly multi-modal, or NP hard
- Could be forced to search all solutions, or hope stochastic search is successful
- Cannot guarantee best solution, inefficient
- Harder to make performance guarantees with approximate solutions

# Mathematical Optimization

## Nonlinear Optimization

### Convex Optimization

#### Least-squares

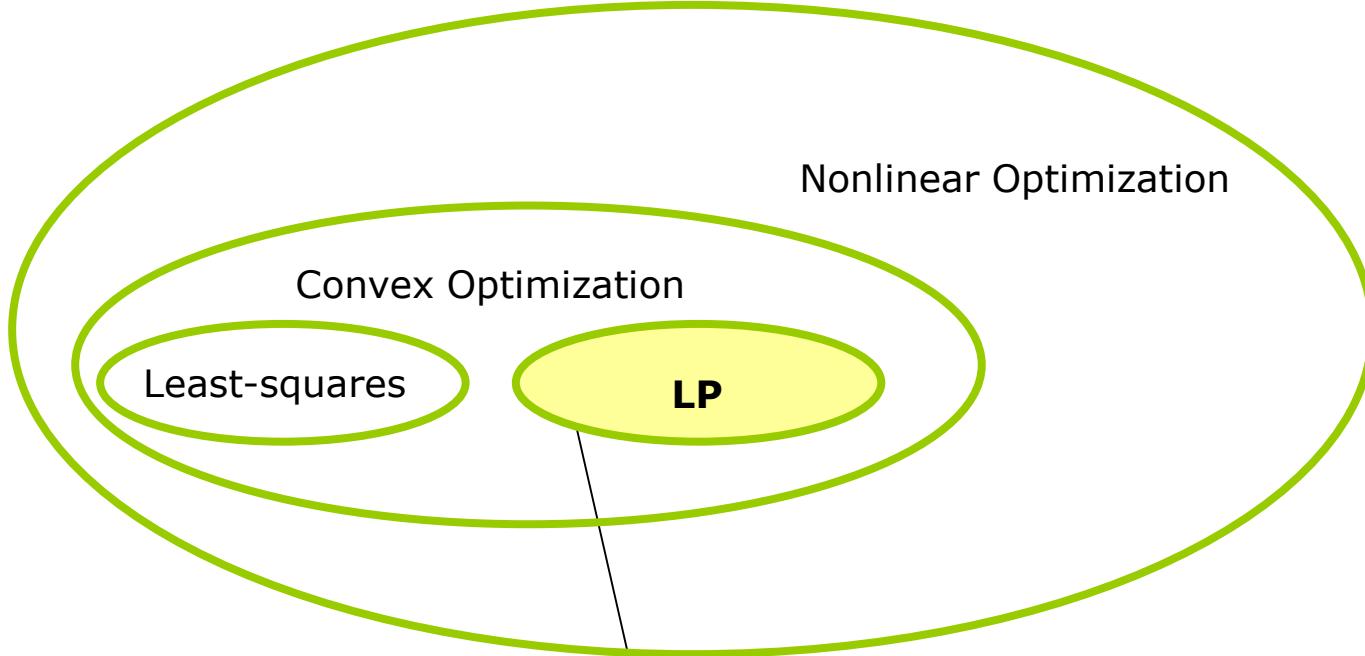
#### LP

$$\text{minimize} \quad \|Ax - b\|_2^2$$

- Analytical solution
- Good algorithms and software
- High accuracy and high reliability
- Time complexity:  $C \cdot n^2 k$

A mature technology!

# Mathematical Optimization

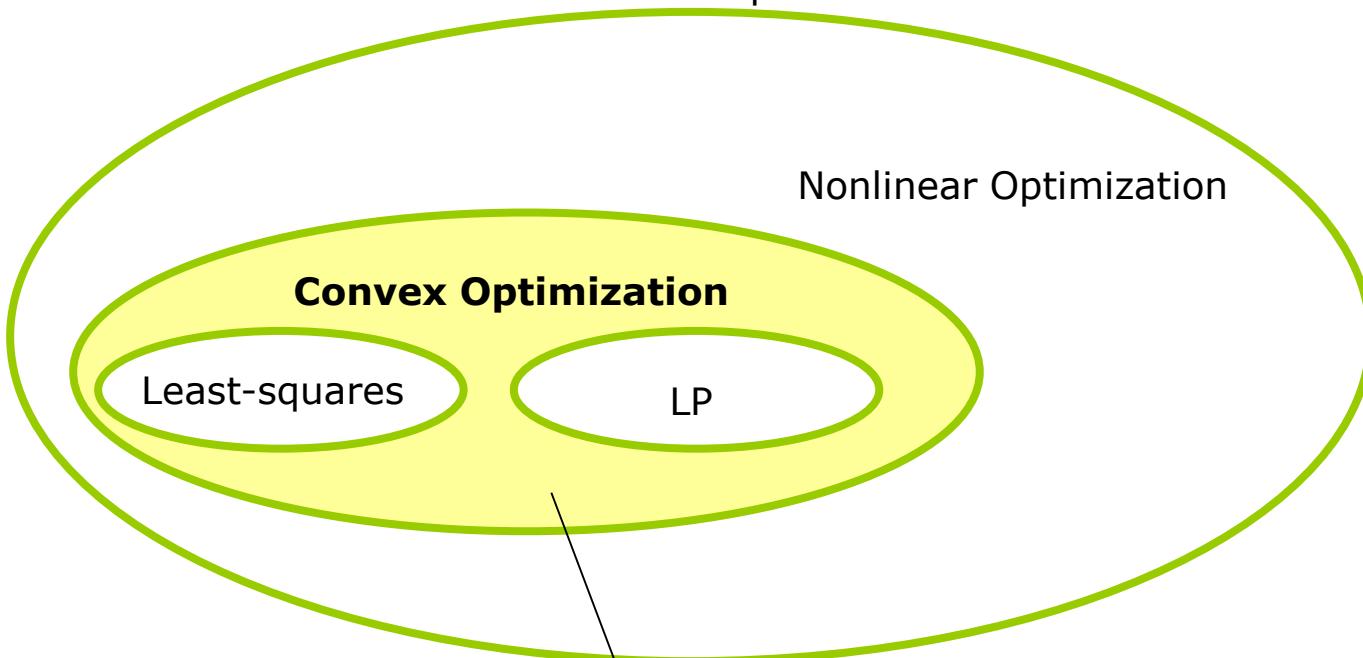


minimize  $c^T x$   
subject to  $a_i^T x \leq b_i$   
 $i = 1, \dots, m$

- No analytical solution
- Algorithms and software
- Reliable and efficient
- Time complexity:  $C \cdot n^2 m$

Also a mature technology!

# Mathematical Optimization



$$\begin{aligned} \text{minimize } & f_0(x) \\ \text{subject to } & f_i(x) \leq b_i, \quad i = 1, \dots, m \end{aligned}$$

- No analytical solution
- Algorithms and software
- Reliable and efficient
- Time complexity (roughly)  
 $\propto \max\{n^3, n^2m, F\}$

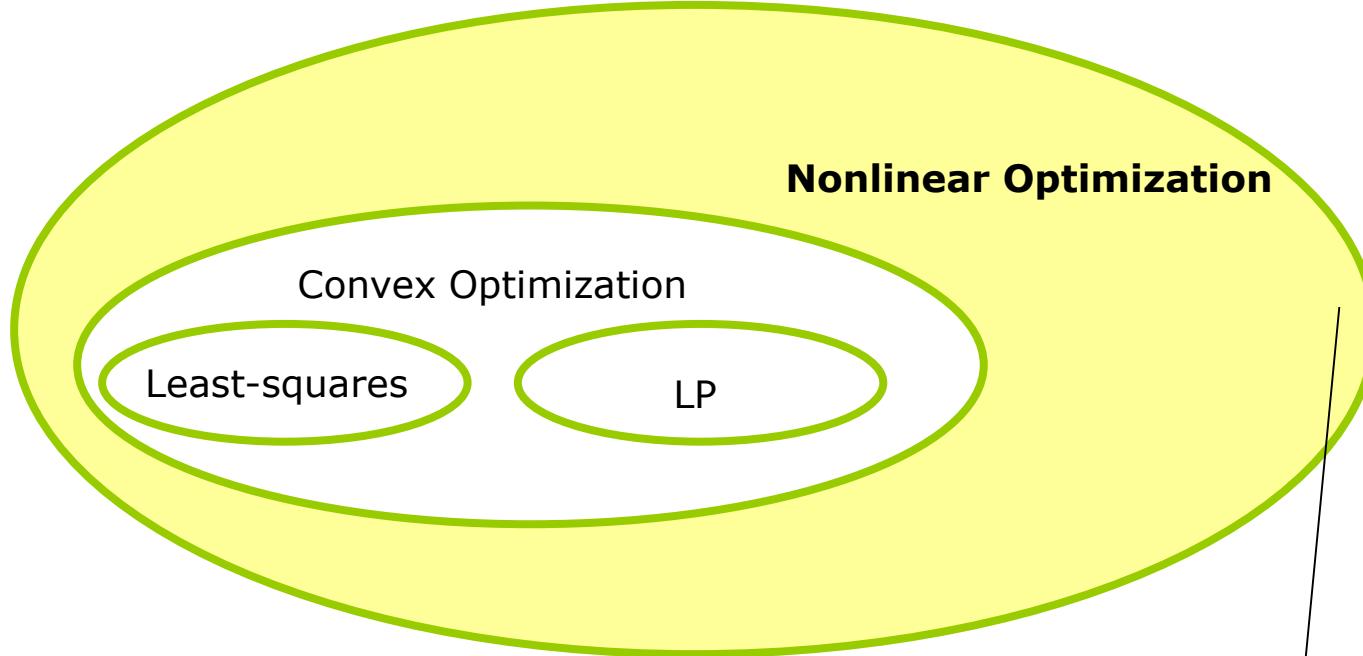
$F$  is cost of evaluating  $f_i$ 's and their first and second derivatives

Almost a ~~mature~~ technology!

117

Rong Jin

## Mathematical Optimization



- Sadly, no effective methods to solve
- Only approaches with some compromise
- Local optimization: "*more art than technology*"
- Global optimization: greatly compromised efficiency
- Help from convex optimization
  - 1) Initialization 2) Heuristics 3) Bounds

Far from a technology! (something to avoid)

# Why Study Convex Optimization

---

With only a bit of exaggeration, we can say that, if you formulate a practical problem as a convex optimization problem, then you have solved the original problem. **If not, there is little chance you can solve it.**

-- Section 1.3.2, p8, *Convex Optimization*

# Recognizing Convex Optimization Problems

---

- often difficult to recognize
- many tricks for transforming problems into convex form
- surprisingly many problems can be solved via convex optimization

$$\begin{aligned} & \text{minimize} && f_0(x) \\ & \text{subject to} && f_i(x) \leq b_i, \quad i = 1, \dots, m \end{aligned}$$

# Least-squares

---

A *least-squares* problem is an optimization problem with no constraints (*i.e.*,  $m = 0$ ) and an objective which is a sum of squares of terms of the form  $a_i^T x - b_i$ :

$$\text{minimize } f_0(x) = \|Ax - b\|_2^2 = \sum_{i=1}^k (a_i^T x - b_i)^2. \quad (1.4)$$

Here  $A \in \mathbf{R}^{k \times n}$  (with  $k \geq n$ ),  $a_i^T$  are the rows of  $A$ , and the vector  $x \in \mathbf{R}^n$  is the optimization variable.

# Analytical Solution of Least-squares

$$\text{minimize } f_0(x) = \|Ax - b\|_2^2 = \sum_{i=1}^k (a_i^T x - b_i)^2.$$

- Set the derivative to zero:
  - $\frac{df_0(x)}{dx} = 0$
  - $(A^T A)2x - 2Ab = 0$
  - $(A^T A)x = Ab$
- Solve this system of linear equations

# Linear Programming (LP)

---

Another important class of optimization problems is *linear programming*, in which the objective and all constraint functions are linear:

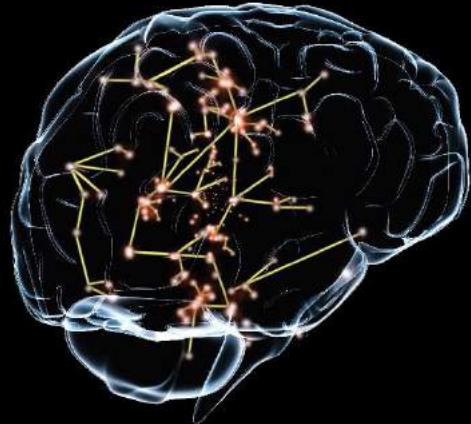
$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && a_i^T x \leq b_i, \quad i = 1, \dots, m. \end{aligned} \tag{1.5}$$

Here the vectors  $c, a_1, \dots, a_m \in \mathbf{R}^n$  and scalars  $b_1, \dots, b_m \in \mathbf{R}$  are problem parameters that specify the objective and constraint functions.

- no analytical formula for solution
- reliable and efficient algorithms and software

# To sum up

- We introduced some important concepts in machine learning and optimization
- We introduced popular machine learning methods
- We talked about loss functions and how we can optimize them using gradient descent



# CENG 783 – Deep Learning

Artificial neural networks

# Now

- Neurons
- Neuron models
- Perceptron learning
- Multi-layer perceptrons
- Backpropagation

It all starts with  
a neuron

# Some facts about human brain

- $\sim 86$  billion neurons
- $\sim 10^{15}$  synapses
- 

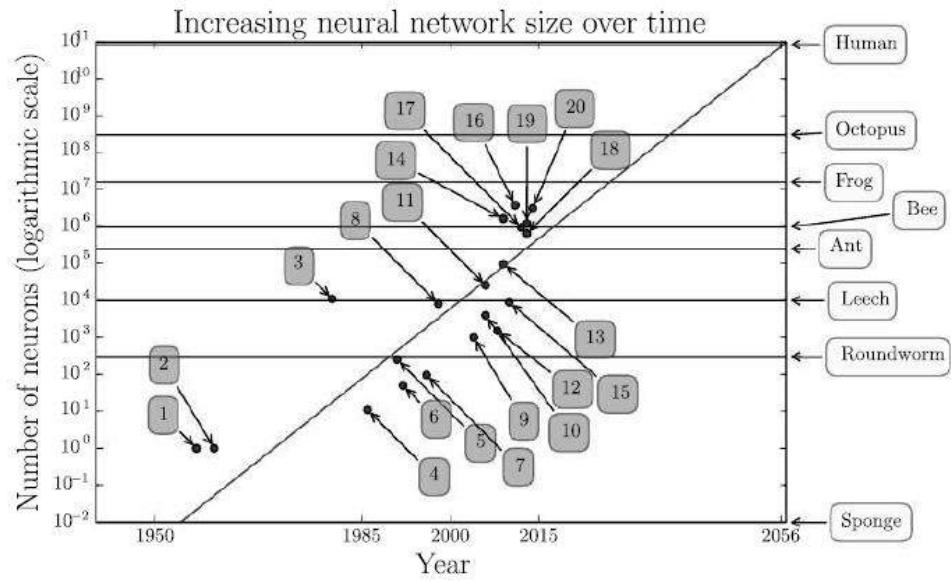
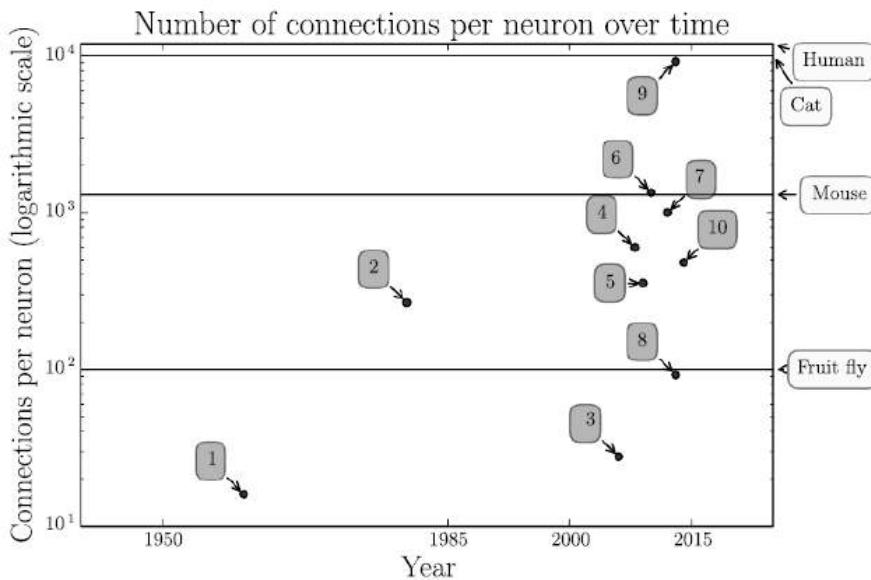
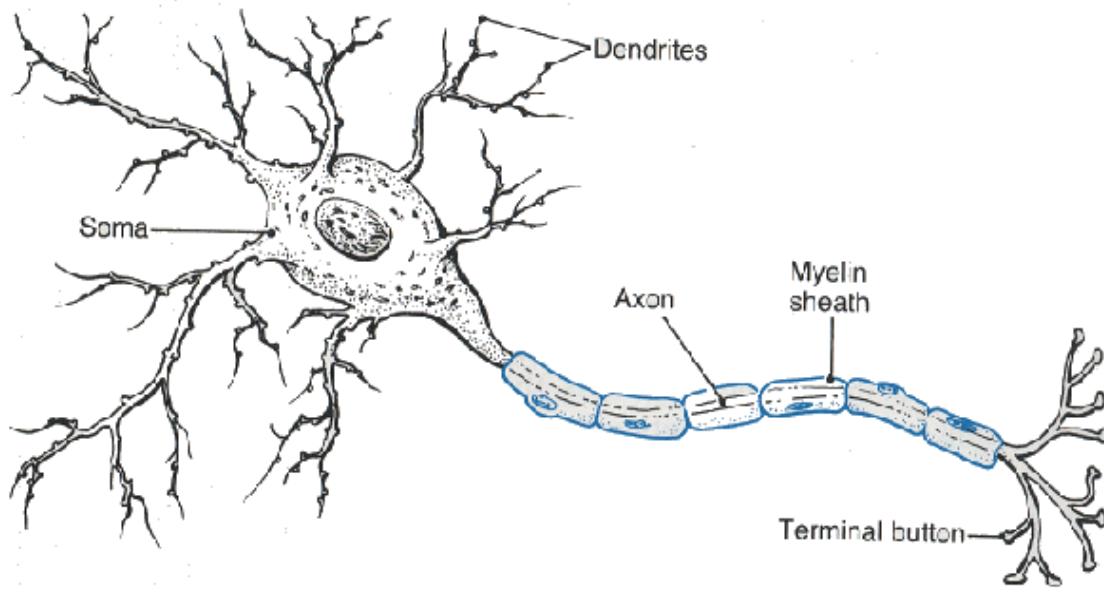


Fig: I. Goodfellow

# Neuron

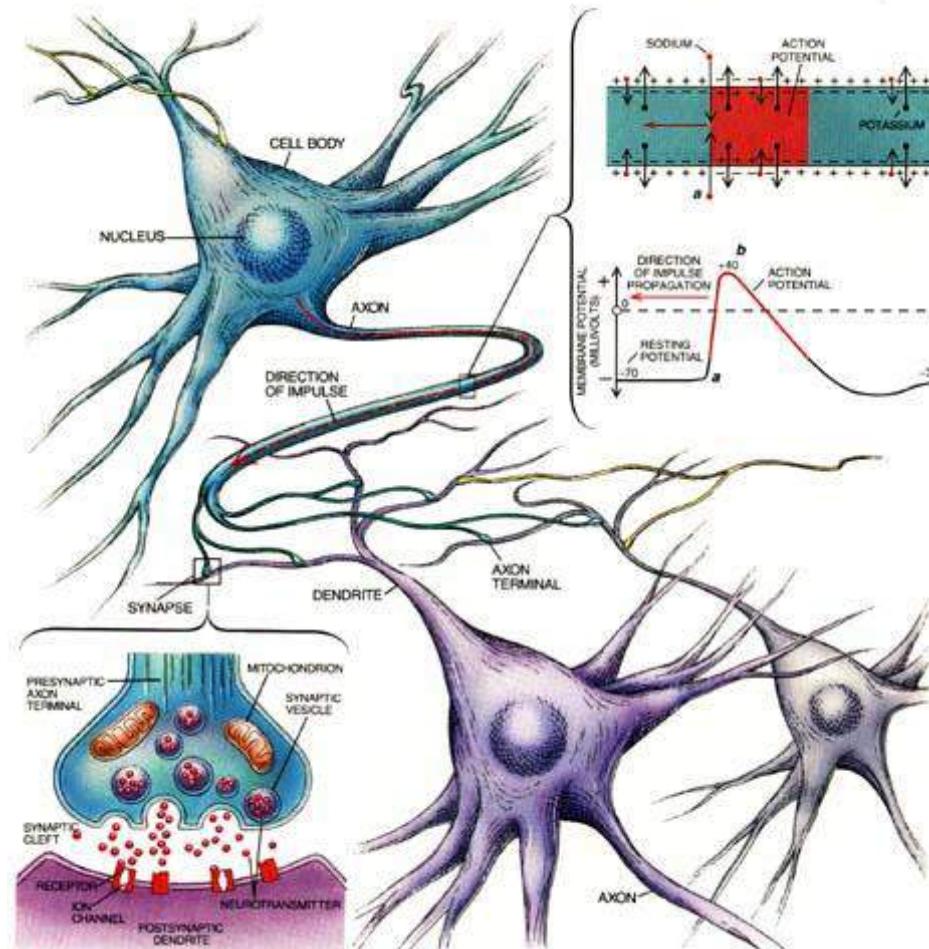
The basic information processing element of neural systems. The neuron

- receives input signals generated by other neurons through its dendrites,
- integrates these signals in its body,
- then generates its own signal (a series of electric pulses) that travel along the axon which in turn makes contacts with dendrites of other neurons.
- The points of contact between neurons are called synapses.

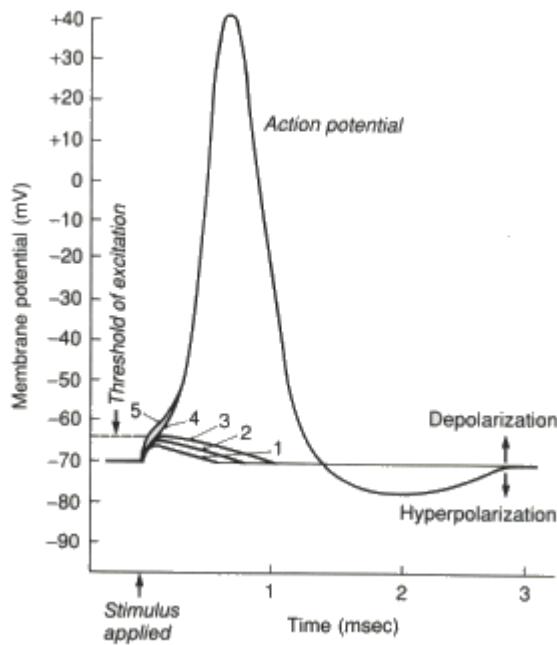


# Neuron

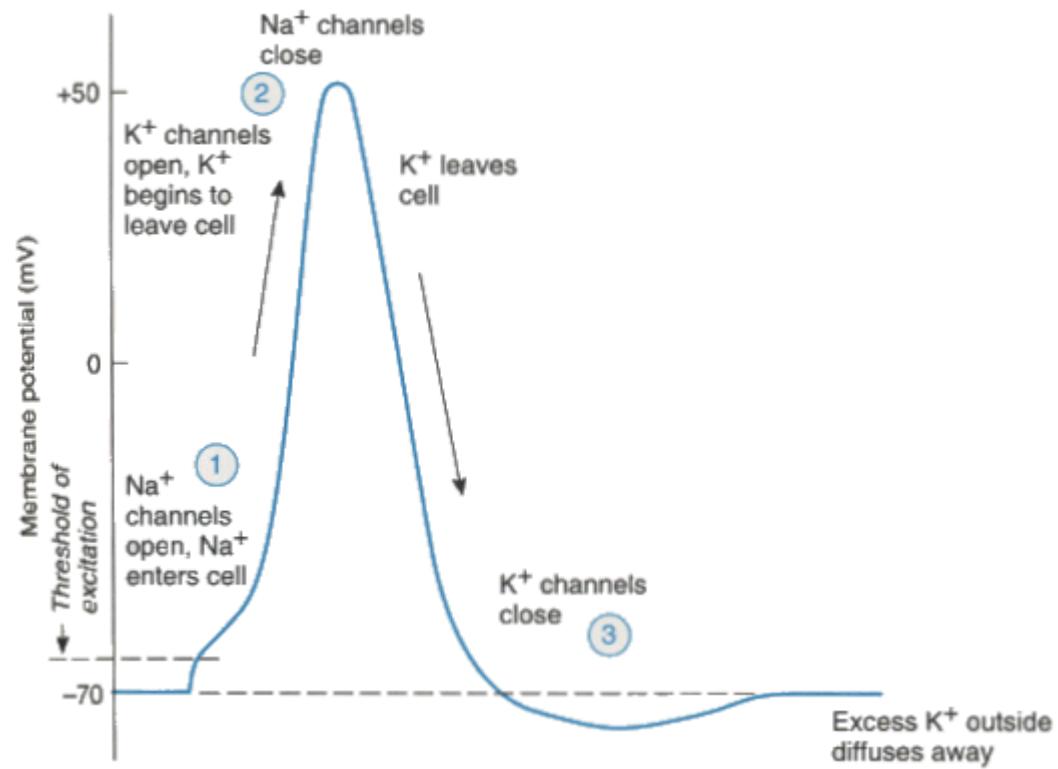
- The pulses generated by the neuron travels along the axon as an electrical wave.
- Once these pulses reach the synapses at the end of the axon open up chemical vesicles exciting the other neuron.



# Neuron

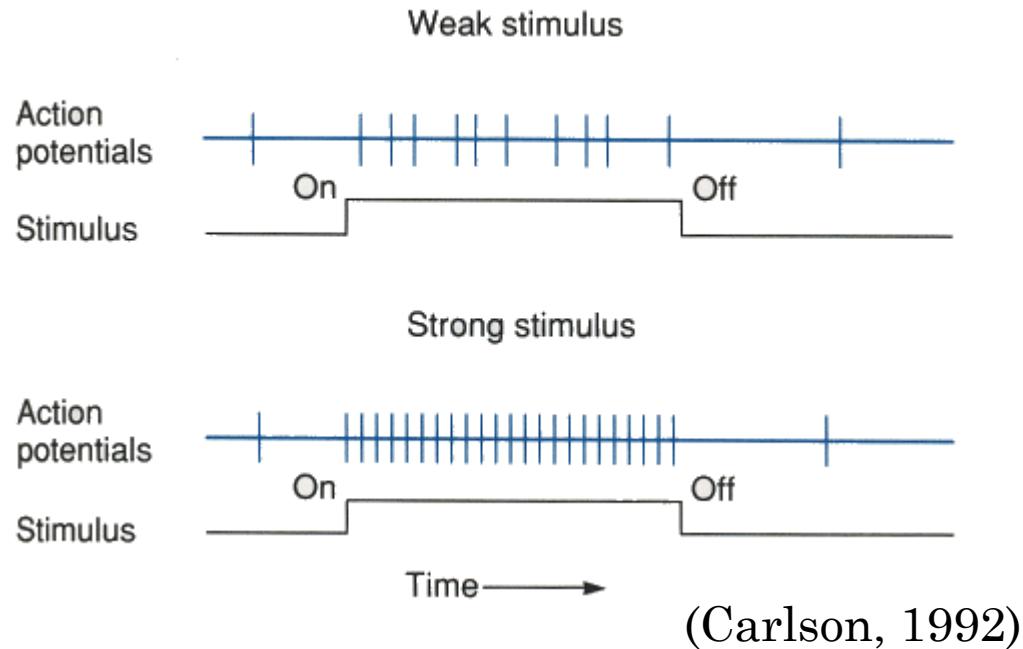


(Carlson, 1992)

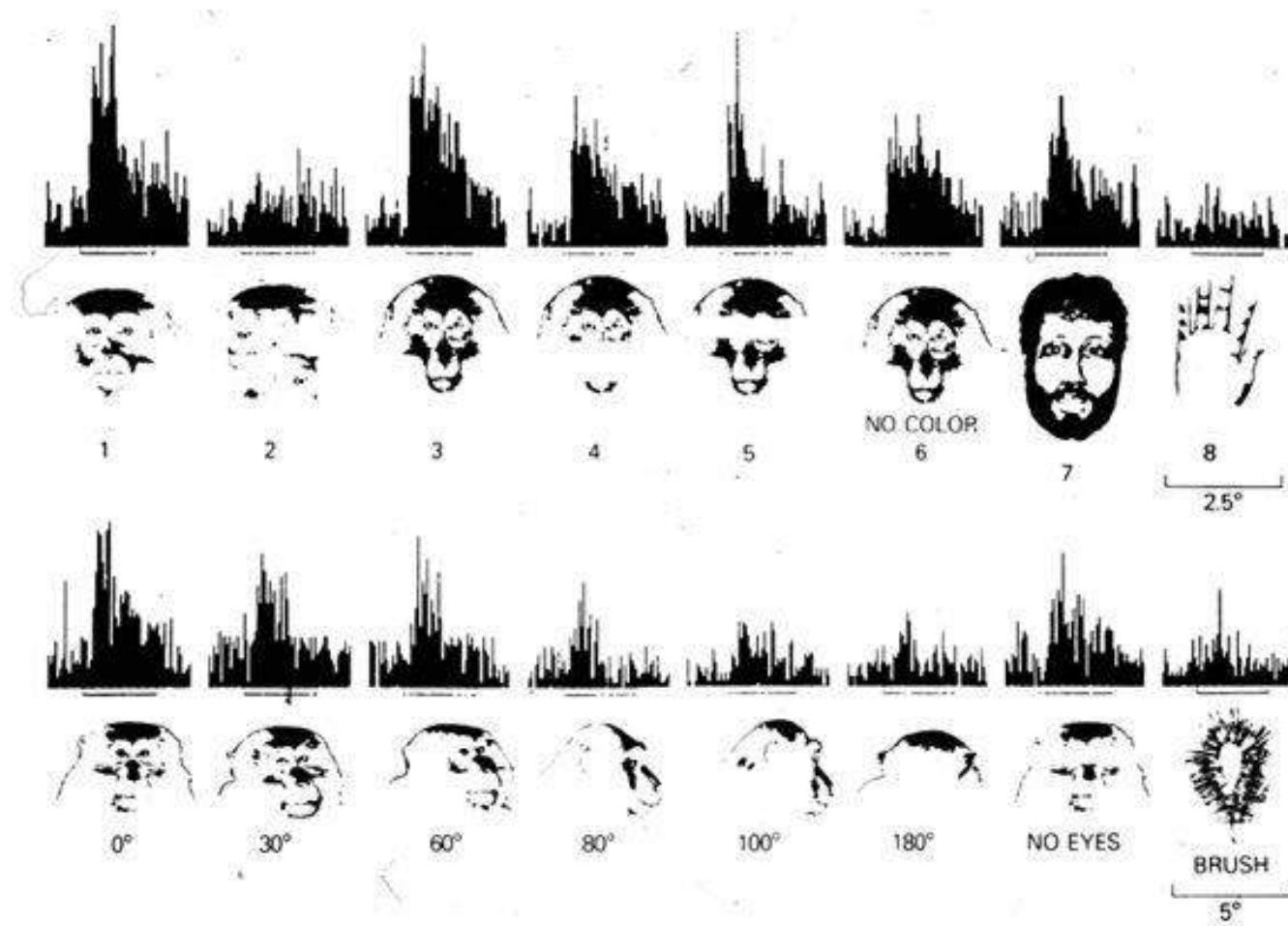


(Carlson, 1992)

# The biological neuron - 2



# Face selectivity in IT



# Artificial neuron

# History of artificial neurons

- Threshold Logic Unit, or Linear Threshold Unit, a.k.a. McCulloch Pitts Neurons – 1943
- Perceptron by Rosenblatt
  - “This model already considered more flexible weight values in the neurons, and was used in machines with adaptive capabilities. The representation of the threshold values as a bias term was introduced by Bernard Widrow in 1960 – see ADALINE.”
- “In the late 1980s, when research on neural networks regained strength, neurons with more continuous shapes started to be considered. The possibility of differentiating the activation function allows the direct use of the gradient descent and other optimization algorithms for the adjustment of the weights. Neural networks also started to be used as a general function approximation model. The best known training algorithm called backpropagation has been rediscovered several times but its first development goes back to the work of Paul Werbos”

# On the Origin of Deep Learning

Haohan Wang

Bhiksha Raj  
*Language Technologies Institute*  
School of Computer Science  
Carnegie Mellon University

HAOHANW@CS.CMU.EDU

BHIKSHA@CS.CMU.EDU

Table 1: Major milestones that will be covered in this paper

Year	Contributer	Contribution
300 BC	Aristotle	introduced Associationism, started the history of human's attempt to understand brain.
1873	Alexander Bain	introduced Neural Groupings as the earliest models of neural network, inspired Hebbian Learning Rule.
1943	McCulloch & Pitts	introduced MCP Model, which is considered as the ancestor of Artificial Neural Model.
1949	Donald Hebb	considered as the father of neural networks, introduced Hebbian Learning Rule, which lays the foundation of modern neural network.
1958	Frank Rosenblatt	introduced the first perceptron, which highly resembles modern perceptron.
1974	Paul Werbos	introduced Backpropagation
1980	Teuvo Kohonen	introduced Self Organizing Map
	Kunihiko Fukushima	introduced Neocogitron, which inspired Convolutional Neural Network
1982	John Hopfield	introduced Hopfield Network
1985	Hilton & Sejnowski	introduced Boltzmann Machine
1986	Paul Smolensky	introduced Harmonium, which is later known as Restricted Boltzmann Machine
	Michael I. Jordan	defined and introduced Recurrent Neural Network
1990	Yann LeCun	introduced LeNet, showed the possibility of deep neural networks in practice
1997	Schuster & Paliwal	introduced Bidirectional Recurrent Neural Network
	Hochreiter & Schmidhuber	introduced LSTM, solved the problem of vanishing gradient in recurrent neural networks
2006	Geoffrey Hinton	introduced Deep Belief Networks, also introduced layer-wise pretraining technique, opened current deep learning era.
2009	Salakhutdinov & Hinton	introduced Deep Boltzmann Machines
2012	Geoffrey Hinton	introduced Dropout, an efficient way of training neural networks

## Bain on Neural Networks

ALAN L. WILKES AND NICHOLAS J. WADE

*University of Dundee, Dundee, Scotland*

In his book *Mind and body* (1873), Bain set out an account in which he related the processes of associative memory to the distribution of activity in neural groupings—or neural networks as they are now termed. In the course of this account, Bain anticipated certain aspects of connectionist ideas that are normally attributed to 20th-century authors—most notably Hebb (1949). In this paper we reproduce Bain's arguments relating neural activity to the workings of associative memory which include an early version of the principles enshrined in Hebb's neurophysiological postulate. Nonetheless, despite their prescience, these specific contributions to the connectionist case have been almost entirely ignored. Eventually, Bain came to doubt the practicality of his own arguments and, in so doing, he seems to have ensured that his ideas concerning neural groupings exerted little or no influence on the subsequent course of theorizing in this area. © 1997 Academic Press

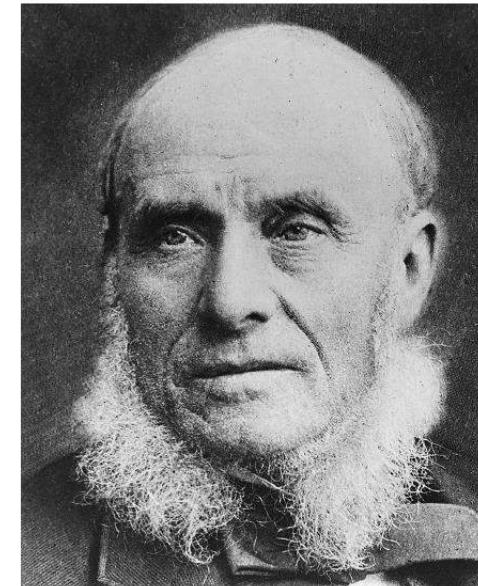


FIG. 1. Alexander Bain in 1892 from a photograph in his *Autobiography* (1904).

Alexander Bain (1818–1903), see Fig. 1, is best known for his textbooks *The senses and the intellect* (1855) and *The emotions and the will* (1859), in which he offered an interpretation of mental phenomena within an associationist framework (for further biographical detail, see Hearnshaw, 1964). Specifically, he tried to match quantitative estimates of the associations held in memory to the neural structure of the brain. It was this exercise that first drew Bain into confronting the potential properties of neural groupings or networks. In the course of thinking about these issues, he was led to speculate on how the internal structure of neural groupings could physically grow to reflect the contingencies of experience and how this same internal structure could come to support the variety of associative links typically found in memory.

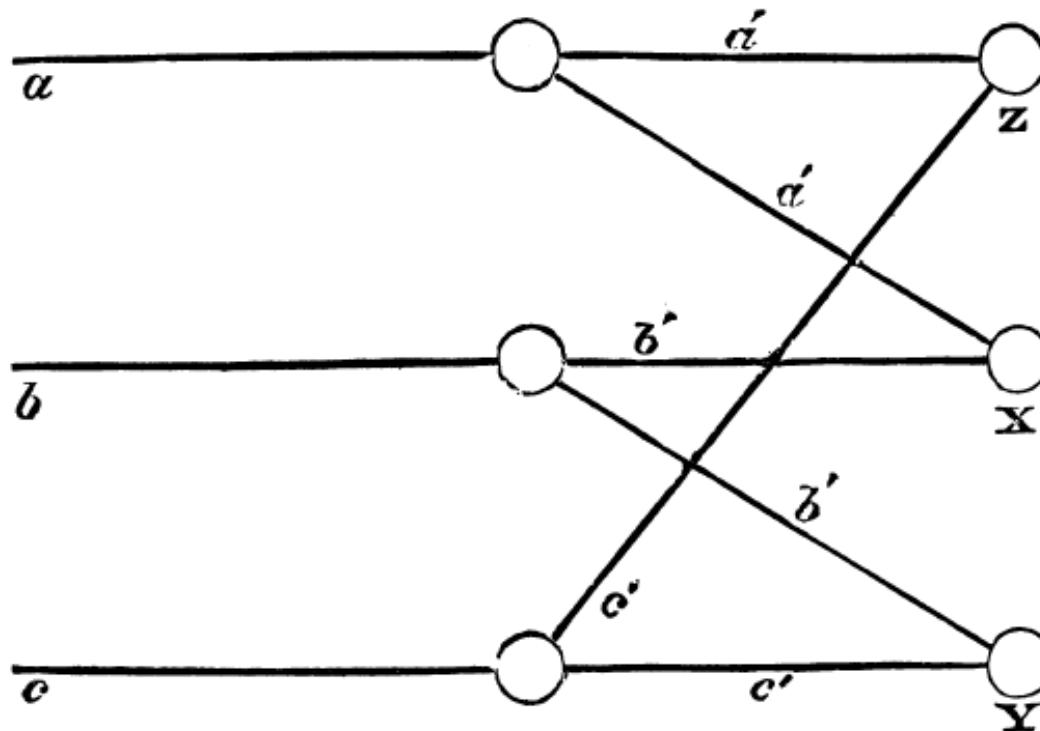


FIG. 2. Bain's diagram illustrating the way in which the connections in a neural network can channel activation in different directions:

It requires us to assume, not merely fibres multiplying by ramification through the cell junctions, but also an extensive arrangement of cross connections. We can typify it in this way. Suppose  $a$  enters a cell junction, and is replaced by several branches,  $a', a''$  etc;  $b$  in like manner, is multiplied into  $b', b''$  etc. Let one of the branches of  $a$  or  $a'$ , pass into some second cell, and a branch of  $b$ , or  $b'$ , pass into the same, and let one of the emerging branches be  $X$ ; we have then a means of connecting united  $a$  and  $b$  with  $X$ ; and in some other crossing, a branch of  $b$  may unite with a branch of  $c$ , from which crossing  $Y$  emerges and so on. . . . By this plan we comply with the primary condition of assigning a separate outcome to every different combination of sensory impressions.

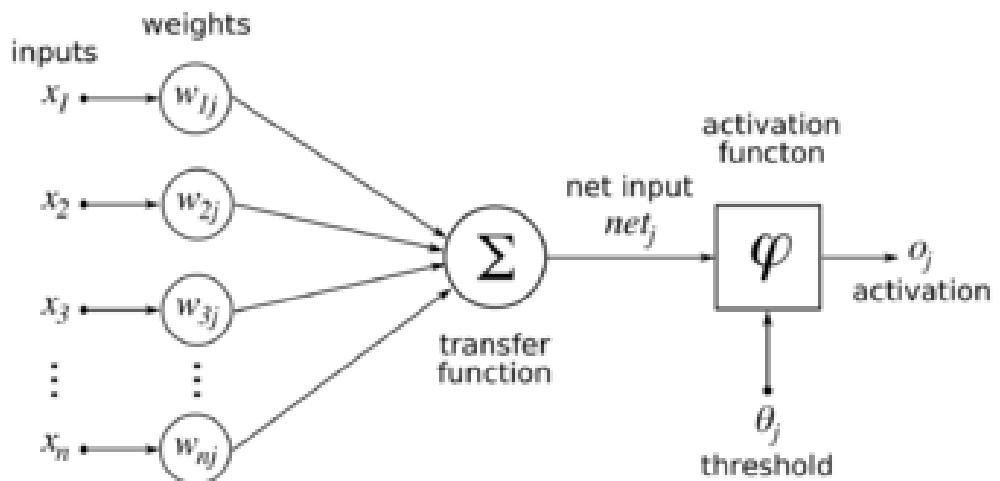
The diagram shows the arrangement. The fibre  $a$  branches into two  $a', a''$ ; the fibre  $b$  into  $b', b''$ ;  $c', c''$ . One of the branches of  $a$  unites with one of the branches of  $b$ , or  $a', b'$  in a cell  $X$ ;  $b', c'$  unite in  $Y$ ;  $a', c'$  in  $Z$ . (1873, pp. 110, 111)

# McCulloch-Pitts Neuron (McCulloch & Pitts, 1943)

- Binary input-output
- Can represent Boolean functions.
- No training.

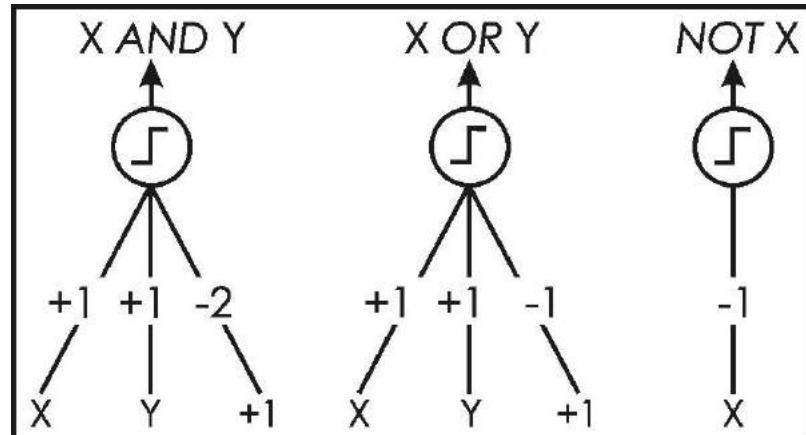
$$net = \sum_i (w_{y,x_i} x_i) + w_{y,x_b}$$

$$f(net) = \begin{cases} 0, & net < 0 \\ 1, & net \geq 0 \end{cases}$$



# McCulloch-Pitts Neuron

- Implement AND:
  - Let  $w_{x1}$  and  $w_{x2}$  to be 1, and  $w_{xb}$  to be -2.
- When input is 1 & 1; net is 0.
- When one input is 0; net is -1.
- When input is 0 & 0; net is -2.



[http://www.webpages.ttu.edu/dleverin/neural\\_network/neural\\_networks.html](http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html)

# McCulloch-Pitts Neuron

Wikipedia:

- “Initially, only a simple model was considered, with binary inputs and outputs, some restrictions on the possible weights, and a more flexible threshold value. Since the beginning it was already noticed that **any boolean function could be implemented by networks of such devices**, what is easily seen from the fact that one can implement the AND and OR functions, and use them in the disjunctive or the conjunctive normal form. Researchers also soon realized that cyclic networks, with feedbacks through neurons, could define dynamical systems with memory, but most of the research concentrated (and still does) on strictly feed-forward networks because of the smaller difficulty they present.”

# McCulloch-Pitts Neuron

- Binary input-output is a big limitation
- Also called

*“[...] caricature models since they are intended to reflect one or more neurophysiological observations, but without regard to realism [...]”*

-- Wikipedia

- No training! No learning!
- They were useful in inspiring research into connectionist models

# Hebb's Postulate (Hebb, 1949)

- “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased”

In short: **Neurons that fire together, wire together.**

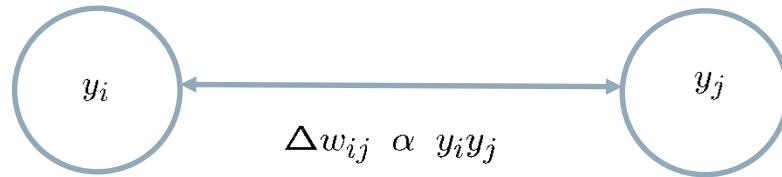
In other words:

$$w_{ij} \propto x_i x_j$$



# Hebb's Learning Law

- Very simple to formulate as a learning rule:



- If the activation of the neurons,  $y_1$  and  $y_2$ , are both on (+1) then the weight between the two neurons grow. (Off: 0)
- Else the weight between remains the same.
- However, when bipolar activation  $\{-1, +1\}$  scheme is used, then the weights can also decrease when the activation of two neurons does not match.

# THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN<sup>1</sup>

F. ROSENBLATT

*Cornell Aeronautical Laboratory*

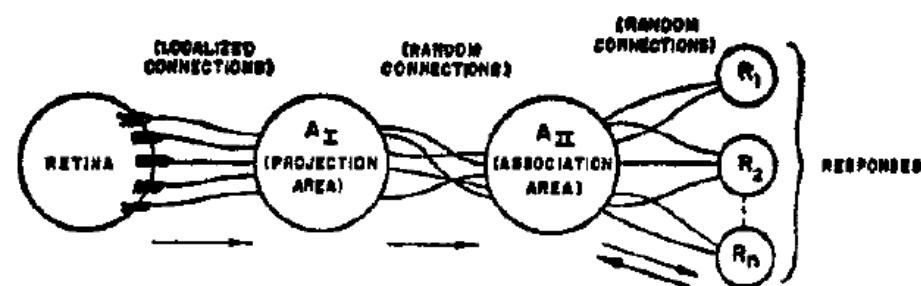
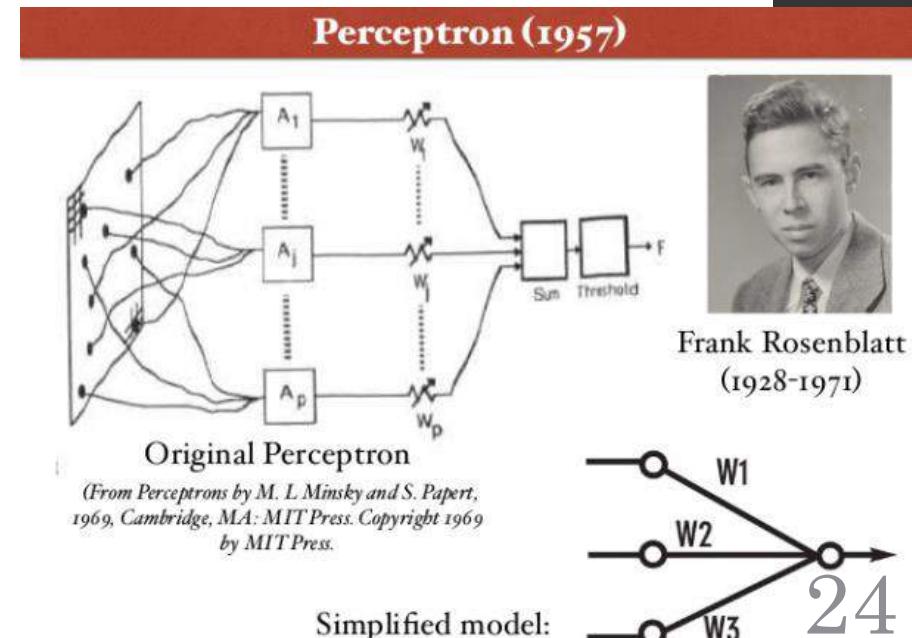


FIG. 1. Organization of a perceptron.





[https://www.youtube.com/watch?v=cNxadbrN\\_aI](https://www.youtube.com/watch?v=cNxadbrN_aI)

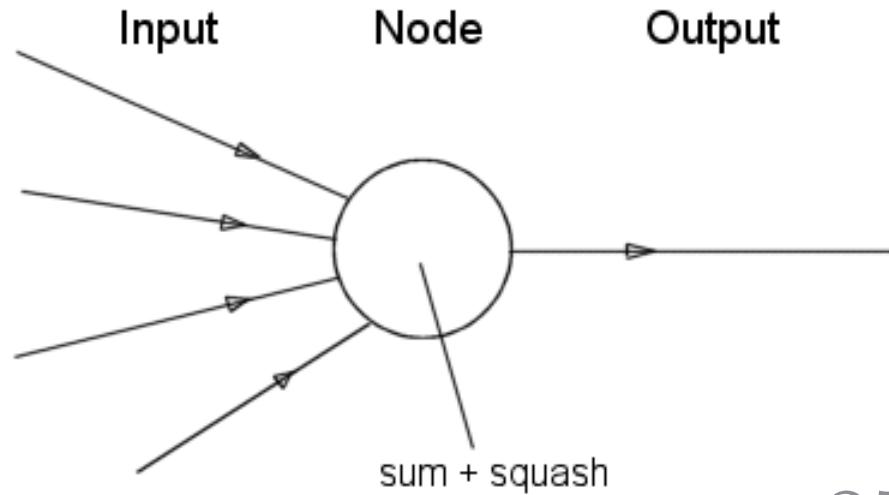
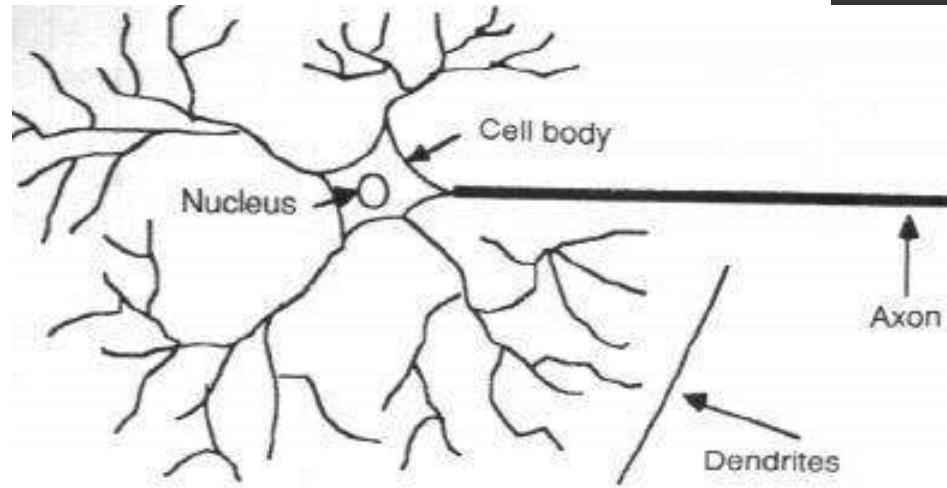
<https://www.youtube.com/watch?v=aygSMgK3BEM>

# And many others

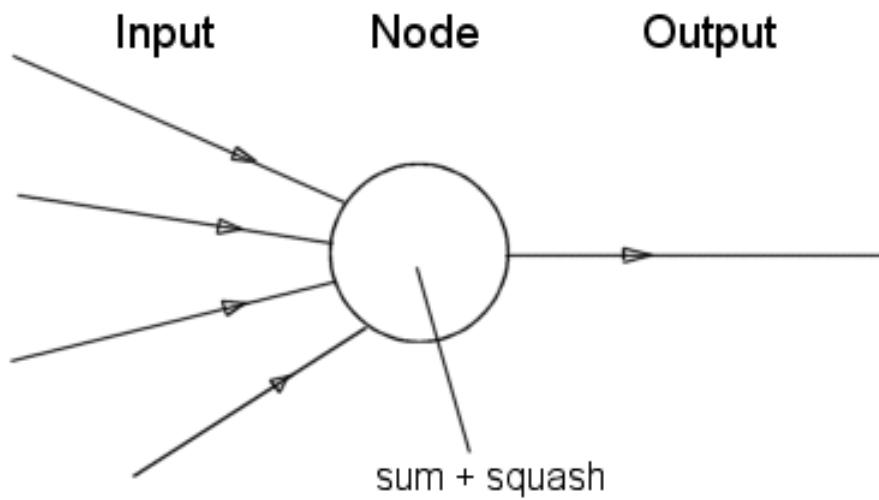
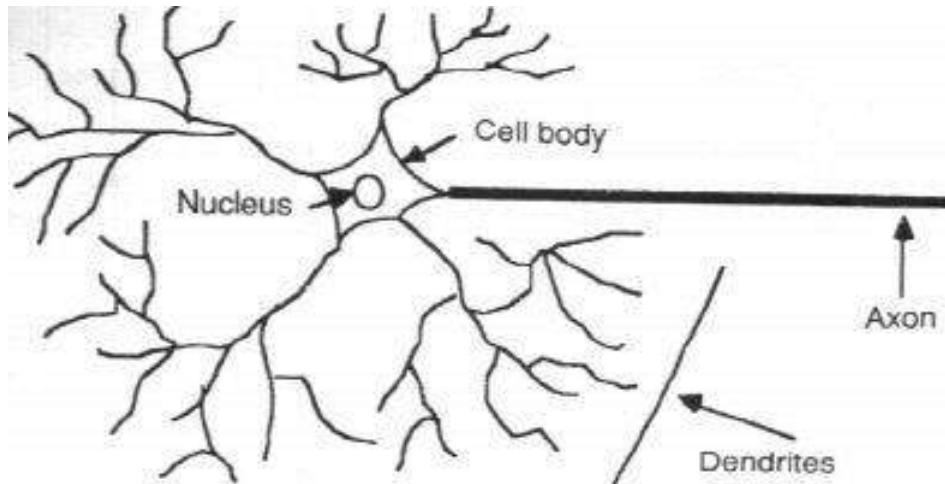
- Widrow & Hoff, 1962
- Grossberg, 1969
- Kohonen, 1972
- von der Malsburg, 1973
- Narendra & Thathchar, 1974
- Palm, 1980
- Hopfield, 1982

# Let's go back to a biological neuron

- A biological neuron has:
  - Dendrites
  - Soma
  - Axon
- Firing is continuous, unlike most artificial neurons
- Rather than the response function, the firing rate is critical



- Neurone vs. Node



- Very crude abstraction
- Many details overseen

“Spherical cow” problem!

# More on this

- <https://medium.com/intuitionmachine/neurons-are-more-complex-than-what-we-have-imagined-b3dd00a1dcd3>

# Spherical cow

Q: How does a physicist milk a cow?

A: Well, first let us consider a spherical cow...

Or

[https://en.wikipedia.org/wiki/Spherical\\_cow](https://en.wikipedia.org/wiki/Spherical_cow)

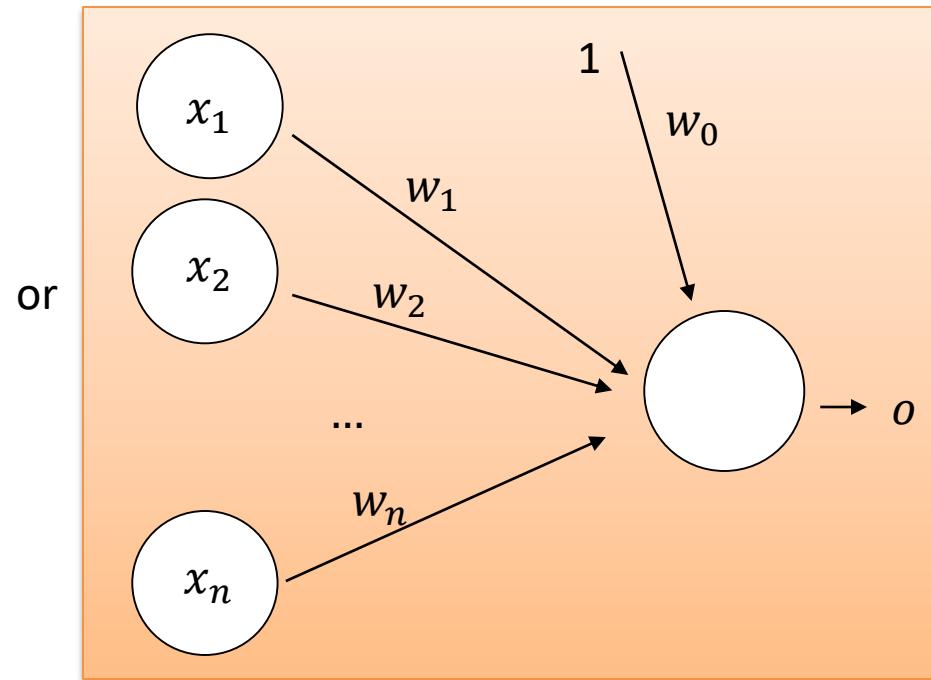
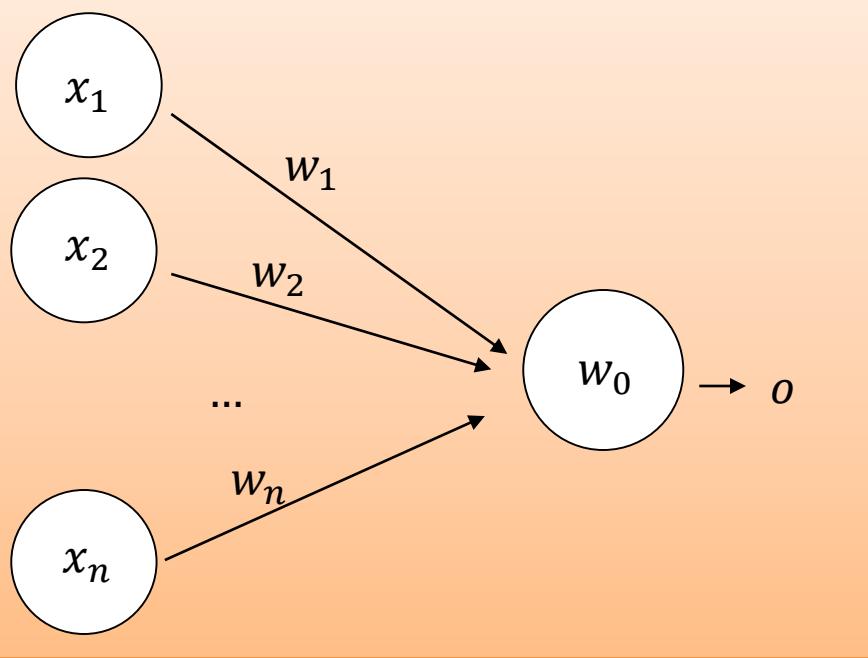
"Milk production at a dairy farm was low, so the farmer wrote to the local university, asking for help from academia. A multidisciplinary team of professors was assembled, headed by a theoretical physicist, and two weeks of intensive on-site investigation took place. The scholars then returned to the university, notebooks crammed with data, where the task of writing the report was left to the team leader. Shortly thereafter the physicist returned to the farm, saying to the farmer, "I have the solution, but it only works in the case of spherical cows in a vacuum"."



<https://www.washingtonpost.com/news/wonk/wp/2013/09/04/the-coase-theorem-is-widely-cited-in-economics-ronald-coase-hated-it/>

# Let us take a closer look at perceptrons

- Initial proposal of connectionist networks
- Rosenblatt, 50's and 60's
- Essentially a linear discriminant composed of nodes and weights



Activation Function

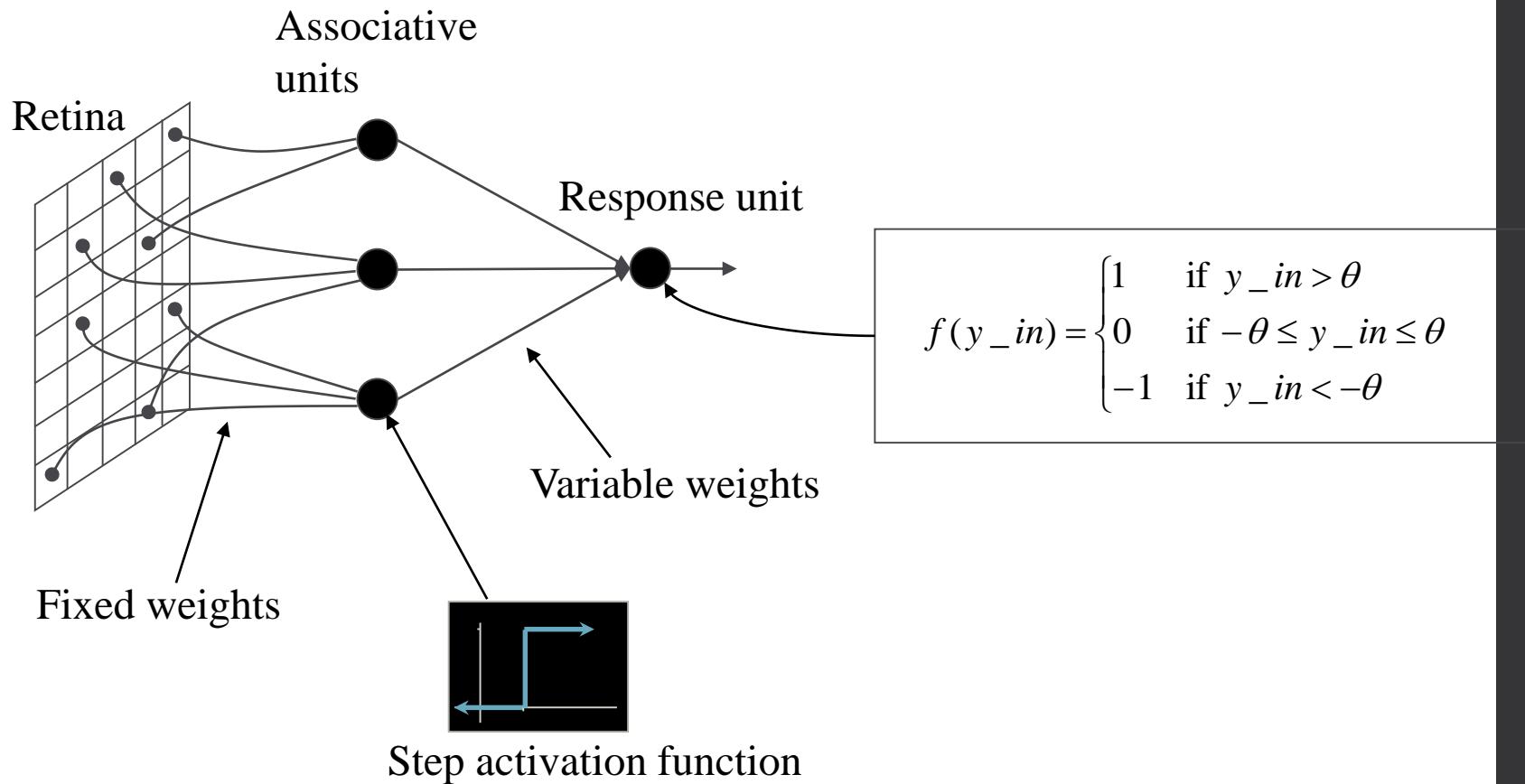
$$o(\mathbf{x}) = \begin{cases} 1, & w_0 + w_1x_1 + \dots + w_nx_n > 0 \\ 0, & \text{otherwise} \end{cases}$$

Or, simply

$$o(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x})$$

where  $\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases}$

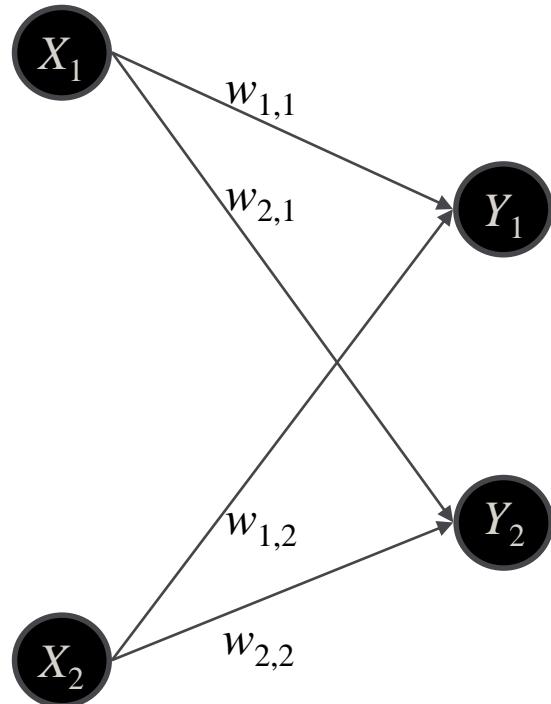
# Perceptron – clearer structure



Slide adapted from Dr. Nigel Crook from Oxford Brookes University  
Slide credit: Erol Sahin

# Perceptron - activation

Simple matrix multiplication, as we have seen in the previous lecture



$$\begin{aligned}\mathbf{Wx} &= \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \begin{bmatrix} w_{1,1}x_1 + w_{1,2}x_2 \\ w_{2,1}x_1 + w_{2,2}x_2 \end{bmatrix} \\ &= \begin{bmatrix} \sum_{j=1}^2 w_{1,j}x_j \\ \sum_{j=1}^2 w_{2,j}x_j \end{bmatrix}\end{aligned}$$

# Motivation for perceptron learning

*(No gradient descent yet)*

- We have estimated an output  $o$ 
  - But the target was  $t$
- Error (simply):  $t - o$
- Let us update each weight such that we “learn” from the error:
  - $w_i \leftarrow w_i + \Delta w_i$
  - where  $\Delta w_i \propto (t - o)$
- We somehow need to distribute the error to the weights.  
How?
  - Distribute the error according to how much they contributed to the error: Bigger input contributes more to the error.
  - Therefore:  $\Delta w_i \propto (t - o)x_i$

# An example

- Consider  $x_i = 0.8, t = 1, o = -1$ 
  - Then,  $(t - o)x_i = 1.6$
  - Which will increase the weight
  - Which makes sense considering the output and the target

# Perceptron training rule

- Update weights

$$w_i \leftarrow w_i + \Delta w_i$$

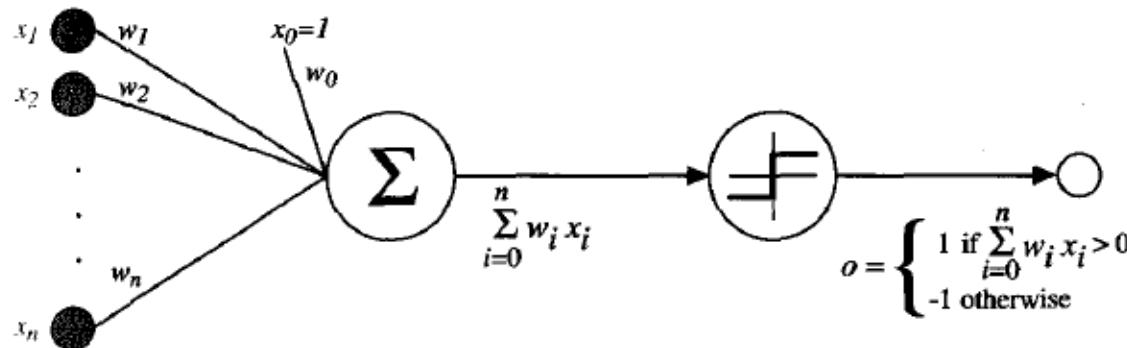
- How to determine  $\Delta w_i$ ?

$$\Delta w_i \leftarrow \eta(t - o)x_i$$

—  $\eta$ : learning rate – can be slowly decreased

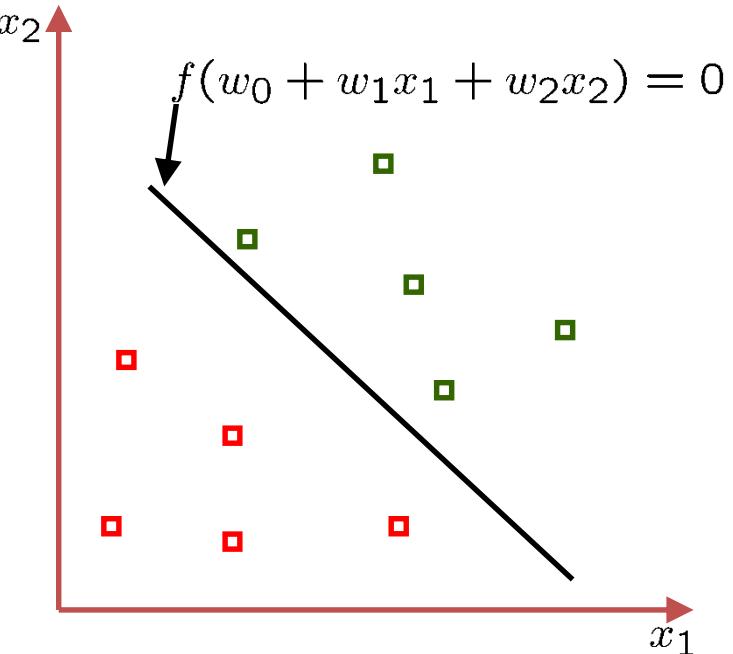
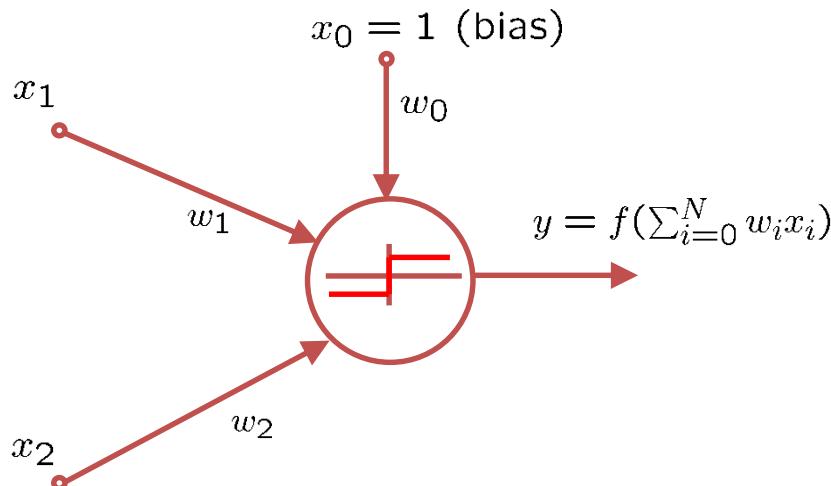
—  $t$ : target/desired output

—  $o$ : current output



# Perceptron - intuition

- A perceptron defines a hyperplane in N-1 space: a line in 2-D (two inputs), a plane in 3-D (three inputs),....
- The perceptron is a linear classifier: It's output is -1 on one side of the plane, and 1 for the other.
- Given a linearly separable problem, the perceptron learning rule guarantees convergence.

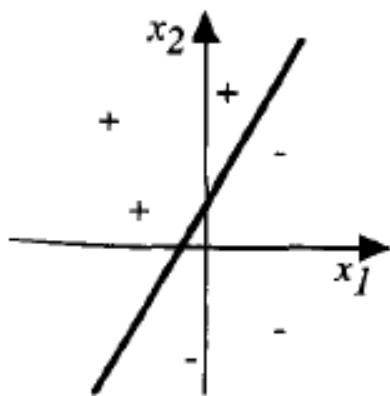


# Problems with perceptron

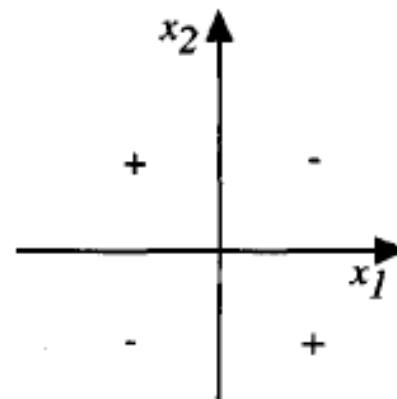
- Perceptron unit is non-linear
- However, it provides zero gradient (due to thresholding function), which makes it unsuitable to gradient descent in multi-layer networks.

# Problems with perceptron learning

- Can only learn linearly separable classification.



linearly separable



**not** linearly separable

# Gradient Descent

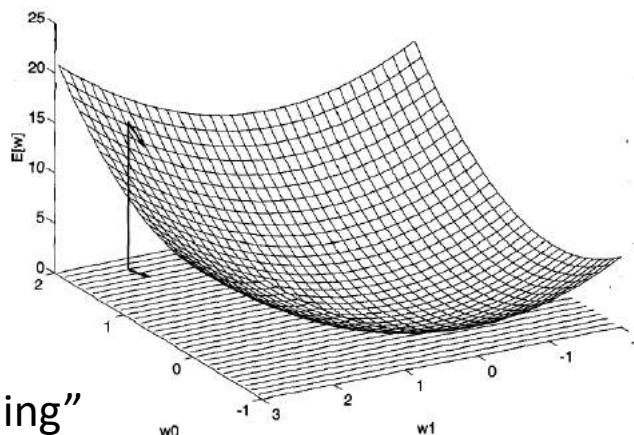
- Consider **unthresholded** perceptron:

$$o(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$$

- We can calculate the error of the perceptron:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- We can guide the search for better weights by using the gradient of this error function.



# Gradient Descent Rule

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$$

- Determine  $\Delta\mathbf{w}$  based on the error function:

$$\Delta\mathbf{w} \leftarrow -\eta \nabla E(\mathbf{w})$$

- For the individual weights:

$$\Delta w_i \leftarrow -\eta \frac{\partial E}{\partial w_i}$$

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)\end{aligned}$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d) (-x_{id})$$

$$\Delta w_i \leftarrow \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

# Gradient Descent Training Algorithm

**GRADIENT-DESCENT(*training-examples*,  $\eta$ )**

*Each training example is a pair of the form  $\langle \vec{x}, t \rangle$ , where  $\vec{x}$  is the vector of input values, and  $t$  is the target output value.  $\eta$  is the learning rate (e.g., .05).*

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - Initialize each  $\Delta w_i$  to zero.
  - For each  $\langle \vec{x}, t \rangle$  in *training-examples*, Do
    - Input the instance  $\vec{x}$  to the unit and compute the output  $o$
    - For each linear unit weight  $w_i$ , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \quad (\text{T4.1})$$

- For each linear unit weight  $w_i$ , Do

$$w_i \leftarrow w_i + \Delta w_i \quad (\text{T4.2})$$

# Stochastic Gradient Descent or Incremental Gradient Descent

- Difficulties of gradient descent:
  - Convergence to a local minimum can be quite slow
  - If there are multiple local minima, no guarantee on finding the global minimum
- One alternative:
  - update the weight after seeing each sample.

$$\Delta w_i \leftarrow \eta(t - o)x_i$$

Compare to (in standard GD):

- Error effectively becomes (per data):

$$E_d(w) = \frac{1}{2}(t_d - o_d)^2$$

$$\Delta w_i \leftarrow \eta \sum_{d \in D} (t_d - o_d)x_{id}$$

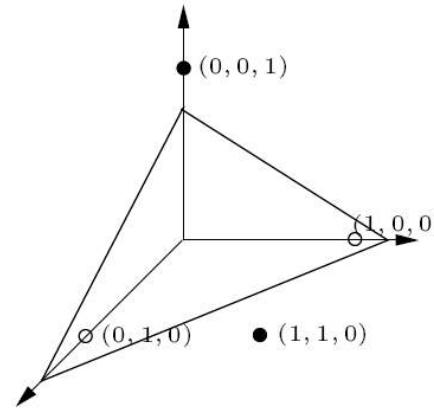
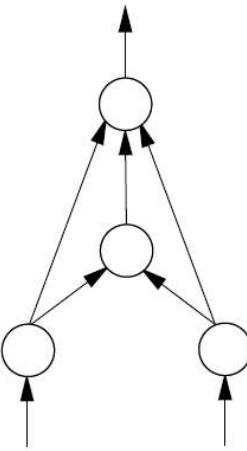
(Delta rule, least-mean-square-rule, Adaline rule or Widrof-Hoff rule)

# Notes on convergence

- Perceptron learning:
  - Output is thresholded
  - Converges after a finite number of iterations to a hypothesis that perfectly classifies the data
  - Condition: data is linearly separable
- Gradient descent (delta rule):
  - Output is not thresholded
  - Converges asymptotically to the minimum error hypothesis
  - Condition: unbounded time
  - Does not require linear separation.

# The limitations of a perceptron: A hidden neuron may help

Input	Output
001	$\Rightarrow$ 0
010	$\Rightarrow$ 1
100	$\Rightarrow$ 1
110	$\Rightarrow$ 0



# **LET'S GET MULTI-LAYER**

# Multi-layer Networks

Information flow is unidirectional

Data is presented to *Input layer*

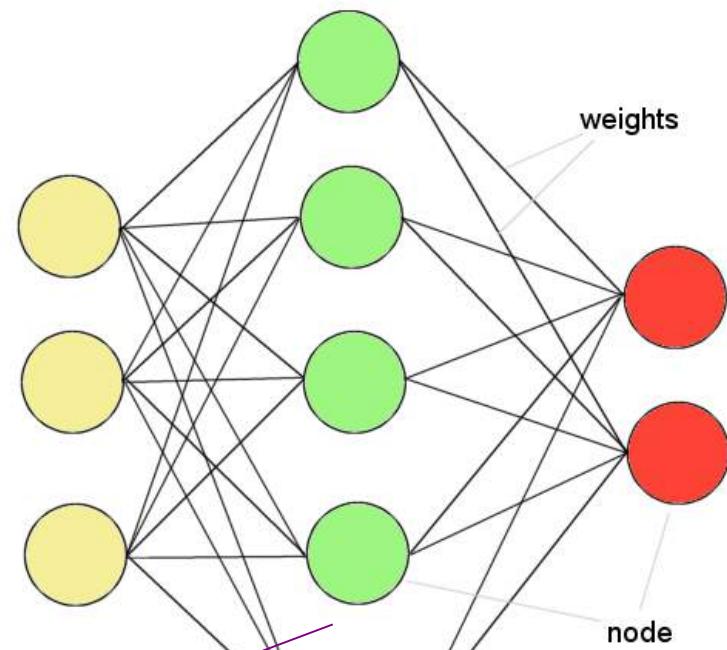
Passed on to *Hidden Layer*

Passed on to *Output layer*

Information is distributed

Information processing is parallel

Input              Hidden              Output



Information

Internal representation (interpretation) of data

# Multi-layered Networks

- To be able to have solutions for linearly non-separable cases, we need a **non-linear** and **differentiable** unit.

$$o = \sigma(\vec{w} \cdot \vec{x})$$

$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

where:

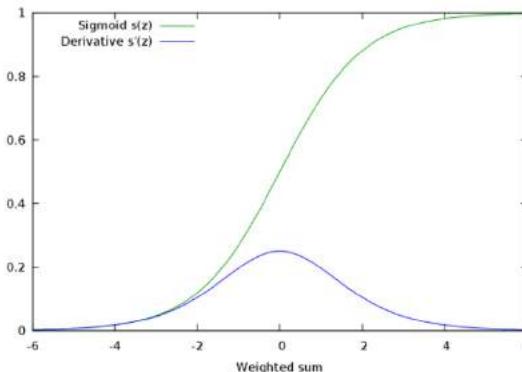
$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Let's denote the sigmoid function as  $\sigma(x) = \frac{1}{1 + e^{-x}}$ .

The derivative of the sigmoid is  $\frac{d}{dx} \sigma(x) = \sigma(x)(1 - \sigma(x))$ .

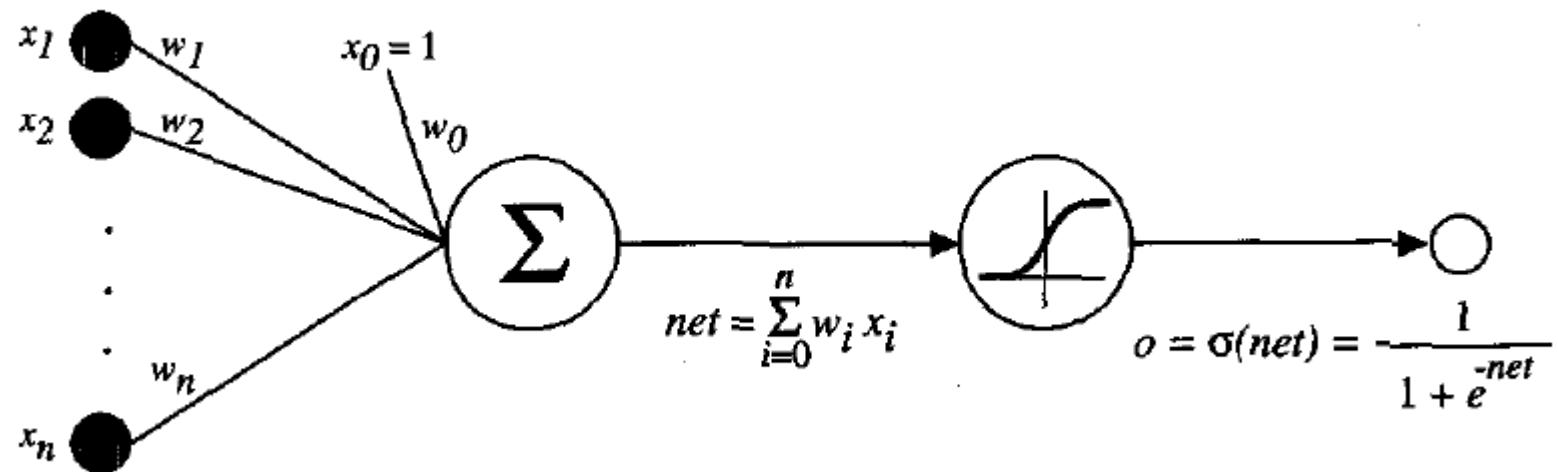
Here's a detailed derivation:

$$\begin{aligned}\frac{d}{dx} \sigma(x) &= \frac{d}{dx} \left[ \frac{1}{1 + e^{-x}} \right] \\ &= \frac{d}{dx} (1 + e^{-x})^{-1} \\ &= -(1 + e^{-x})^{-2} (-e^{-x}) \\ &= \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} \\ &= \frac{1}{1 + e^{-x}} \cdot \frac{(1 + e^{-x}) - 1}{1 + e^{-x}} \\ &= \frac{1}{1 + e^{-x}} \cdot \left( 1 - \frac{1}{1 + e^{-x}} \right) \\ &= \sigma(x) \cdot (1 - \sigma(x))\end{aligned}$$



- Sigmoid (logistic) function
- Output is in (0,1)
- Since it maps a large domain to (0,1) it is also called **squashing function**
- Alternatives: *tanh*

# Perceptron with sigmoid function



# Why do we need to learn backpropagation?

- “Many frameworks implement backpropagation for us, why do we need to learn?”
  - This is not a blackbox. There are many problems/issues involved. You can only deal with them if you have a good understanding of backpropagation.

<https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.7zawffou2>

# Backpropagation algorithm

- Let us re-define the error function since we have many outputs:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$$

- For one data:

$$E_d(\mathbf{w}) = \frac{1}{2} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$$

- For each output unit  $k$ , calculate its error term  $\delta_k$ :

$$\begin{aligned}\delta_k &= -\partial E_d(\mathbf{w}) / \partial o_k \\ \delta_k &= o_k(1 - o_k)(t_k - o_k)\end{aligned}$$

- For each hidden unit  $h$ , calculate its error term  $\delta_h$ :

$$\delta_h = o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh} \delta_k$$

- Update every weight  $w_{ji}$

$$w_{ji} = w_{ji} + \eta \delta_j x_{ji}$$

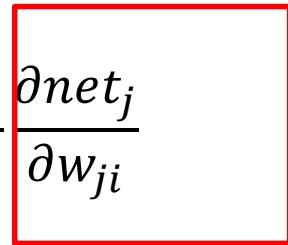
Derivative of the  
Sigmoid function

# Derivation of backpropagation

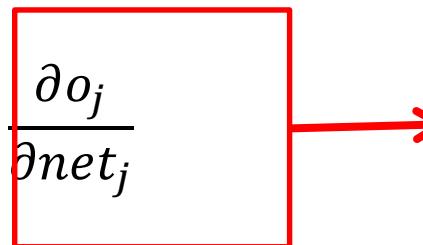
- Derivation of the output unit weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- Expand  $\frac{\partial E_d}{\partial w_{ji}}$ :

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$


- Expand  $\frac{\partial E_d}{\partial net_j}$ :

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$


Derivative of sigmoid  
 $o_j(1 - o_j)$

Therefore:

$$\frac{\partial}{\partial o_j} \frac{1}{2} \sum_k (t_k - o_k)^2 = \frac{\partial}{\partial o_j} \frac{1}{2} (t_k - o_k)^2 = -(t_j - o_j)$$


$$\begin{aligned}\Delta w_{ji} &= -\eta \frac{\partial E_d}{\partial w_{ji}} \\ &= \eta (t_j - o_j) o_j (1 - o_j) x_{ij}\end{aligned}$$


# Derivation of backpropagation

- Derivation of the output unit weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

Therefore:

$$\begin{aligned}\Delta w_{ji} &= -\eta \frac{\partial E_d}{\partial w_{ji}} \\ &= \eta(t_j - o_j)o_j(1 - o_j)x_{ij}\end{aligned}$$


$\delta_j$   
(error term for unit j)

# Derivation of backpropagation

- Derivation of the hidden unit weights

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- Expand  $\frac{\partial E_d}{\partial w_{ji}}$ :

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

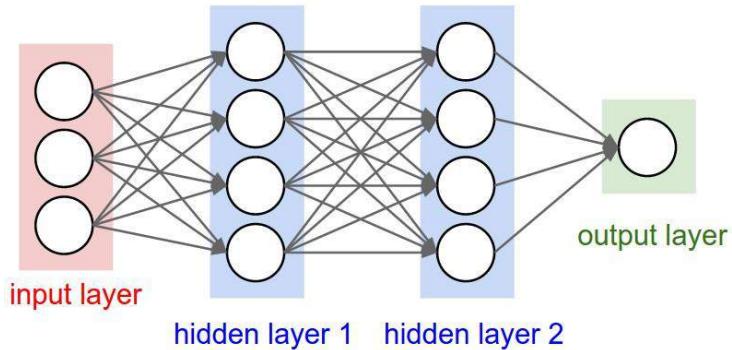
- Expand  $\frac{\partial E_d}{\partial net_j}$ :

$$\begin{aligned} \frac{\partial E_d}{\partial net_j} &= \sum_k \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\ &\quad \text{---} \delta_k \text{ ---} \\ &= \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\ &= w_{kj} o_j (1 - o_j) \end{aligned}$$

Therefore:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta \delta_j x_{ji} = \eta x_{ji} [o_j(1 - o_j) \sum_k \delta_k w_{kj}]$$

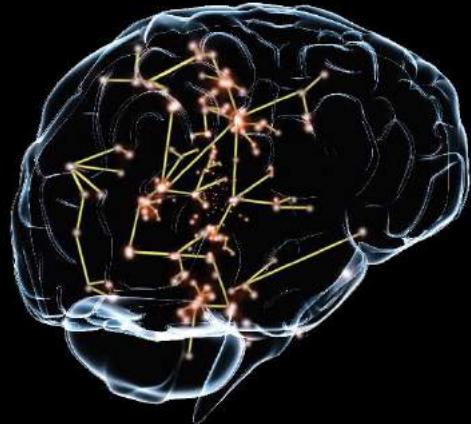
# Forward pass



```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

# Backpropagation vs. numerical differentiation

- Backpropagation:
  - $O(|W|)$
- Numerical differentiation
  - $O(|W|^2)$



# CENG 783 – Deep Learning

Week 4: Artificial neural networks

# Notes

- HW
  - Extension to 18th of May.
- Project proposals
- Opportunity to work in a TUBITAK project

# Now

- Details of an ANN
  - Activation function
  - Gradient Descent strategies
  - Setting the learning rate
  - Representational capacity
  - Overfitting/generalization
  - When to stop training

# Problems of back propagation with sigmoid

- It is extremely slow, if it does converge.
- It may get stuck in a local minima.
- It is sensitive to initial conditions.
- It may start oscillating.

# Backprop in deep networks

- Local minima may not be as severe as it is feared
  - If one weight gets into local minima, other weights provide escape routes
  - The more weights, the more escape routes
- Add a momentum term
- Use stochastic gradient descent, rather than true gradient descent
  - This means we have different error surfaces for each data
  - If stuck in local minima in one of them, the others might help
- Train multiple networks with different initial weights
  - Select the best one

# Backprop

- Very powerful - can learn any function, given enough hidden units!
- Have the same problems of Generalization vs. Memorization.
  - With too many units, we will tend to memorize the input and not generalize well. Some schemes exist to “prune” the neural network.
- Networks require extensive training, many parameters to fiddle with. Can be extremely slow to train. May also fall into local minima.
- Inherently parallel algorithm, ideal for multiprocessor hardware.
- Despite the cons, a very powerful algorithm that has seen widespread successful deployment.

# Now, let us look at alternative aspects

- Loss functions
  - Hinge-loss, softmax loss, squared-error loss, ...
  - We will not look at them here again
- Activation functions
  - Sigmoid, tanh, ReLU, Leaky ReLU, parametric ReLU, maxout
- Backpropagation strategy:
  - True Gradient Descent, Stochastic Gradient Descent, Mini-batch Gradient Descent, RMSprop, AdaDelta, AdaGrad, Adam

# Activation Functions

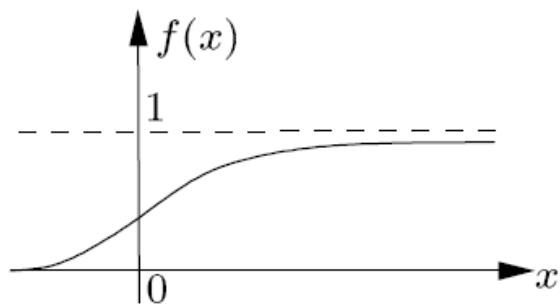
# Activation function

- Sigmoid / logistic function

The computational power is increased by the use of a squashing function.  
In the original paper the logistic function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

is used.



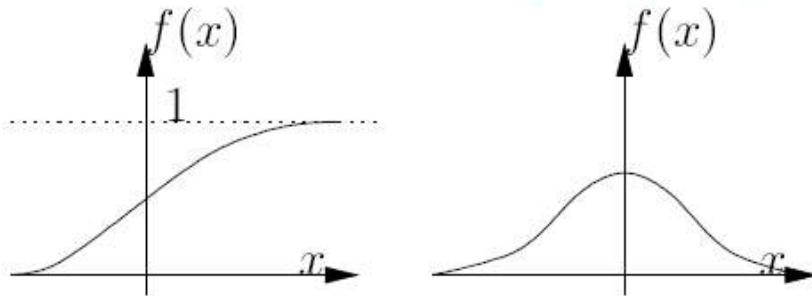
# Activation function

Logistic function  $o_{pj} = \frac{1}{1+e^{-net_{pj}}}$  has nice features:

- The derivative is expressable in terms of the function itself:

$$\frac{\partial o_{pj}}{\partial net_{pj}} = o_{pj}(1 - o_{pj})$$

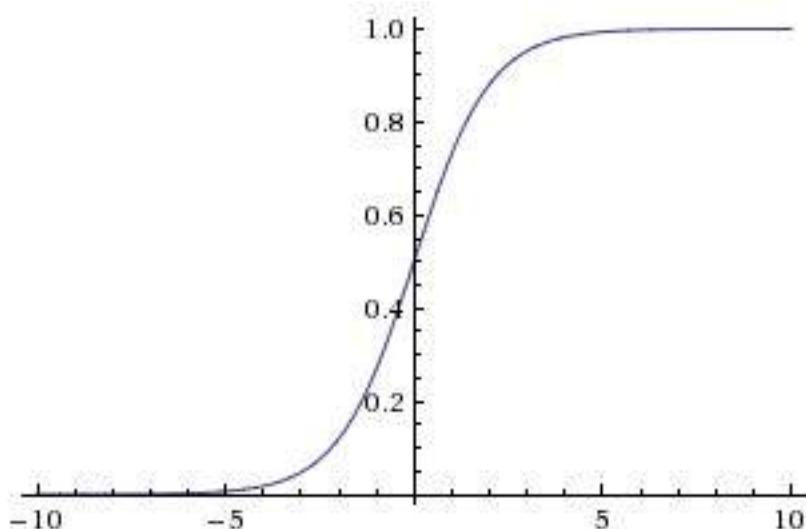
- The derivative is a “bump” that pushes uncommitted nodes to change weights.
- Likewise, weights are prevented from blowing up.
- **The downside is that it is hard to change large weights!**



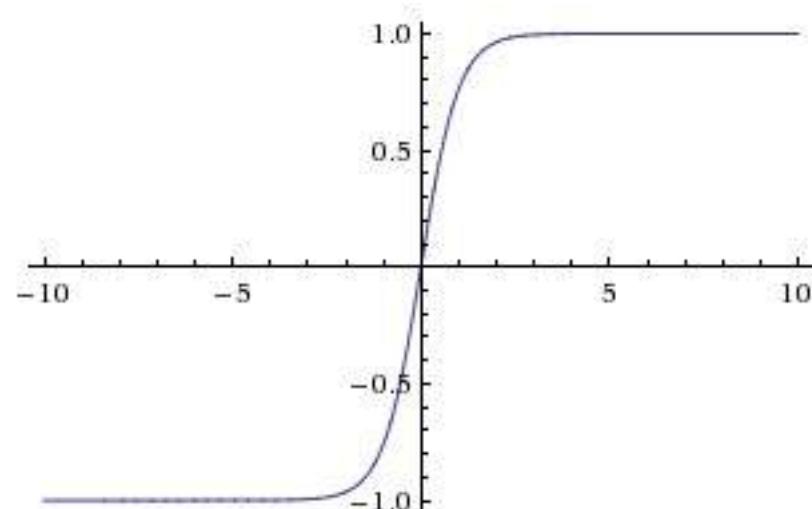
- How about  $f(\cdot) = \tanh(\cdot)$  which squashes the input into the  $[-1 : +1]$ ?
- All we need to make sure it that  $f(\cdot)$  is differentiable.

# Activation Functions

- sigmoid vs tanh



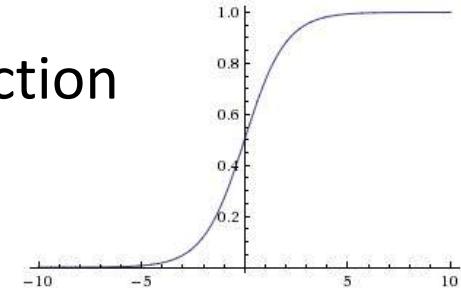
Derivative:  $\sigma(x)(1 - \sigma(x))$



Derivative:  $(1 - \tanh^2(x))$

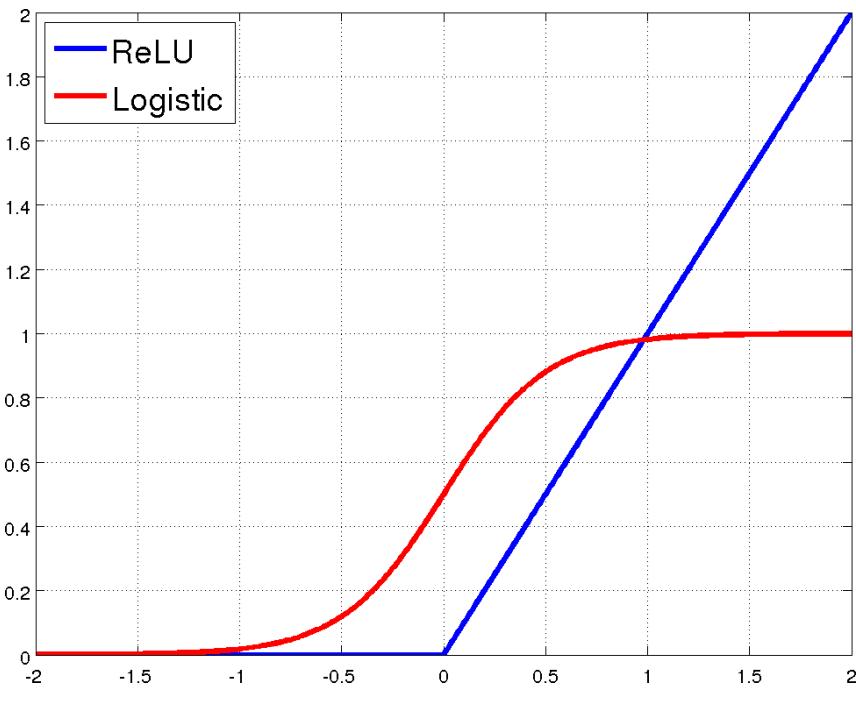
# Pros and Cons

- Sigmoid is an historically important activation function
  - But nowadays, rarely used
- Sigmoid drawbacks
  1. It gets saturated, if the activation is close to zero or one
    - This leads to very small gradient, which disallows “transfer”ing the feedback to earlier layers
    - Initialization is also very important for this reason
  2. It is not zero-centered (not very severe)
- Tanh
  - Similar to the sigmoid, it saturates
  - However, it is zero-centered.
  - Tanh is **always** preferred over sigmoid
  - Note:  $\tanh(x) = 2\sigma(2x) - 1$



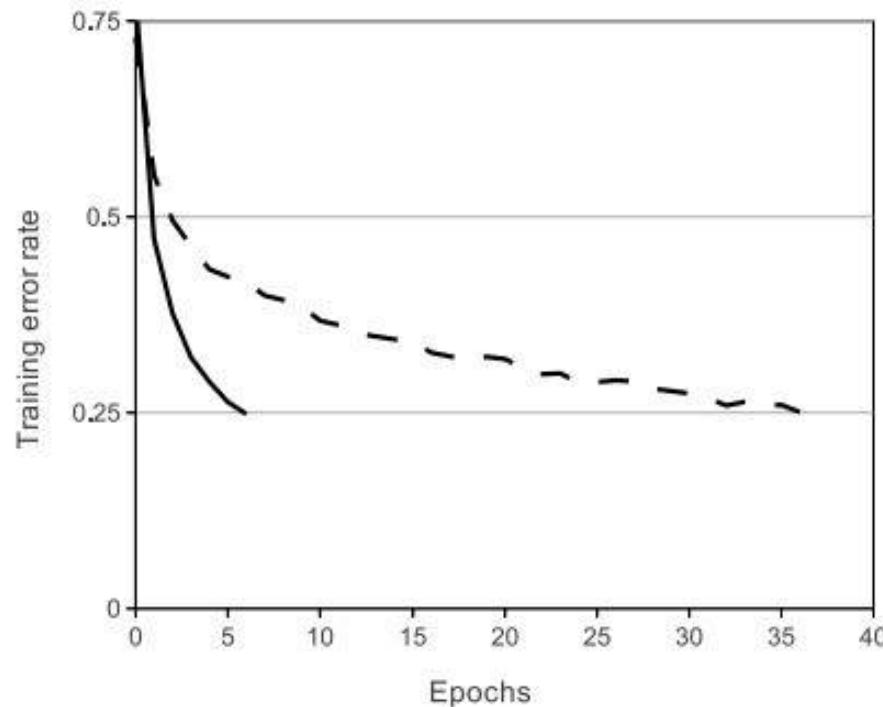
# Rectified Linear Units (ReLU)

Vinod Nair and Geoffrey Hinton (2010). Rectified linear units improve restricted Boltzmann machines, ICML.



$$f(x) = \max(0, x)$$

Derivative:  $\mathbf{1}(x > 0)$



[Krizhevsky et al., NIPS12]

# ReLU: biological motivation

$$f(I) = \begin{cases} \left[ \tau \log \left( \frac{E+RI-V_r}{E+RI-V_{th}} \right) + t_{ref} \right]^{-1}, & \text{if } E + RI > V_{th} \\ 0, & \text{if } E + RI \leq V_{th} \end{cases}$$

where  $t_{ref}$  is the refractory period (minimal time between two action potentials),  $I$  the input current,  $V_r$  the resting potential and  $V_{th}$  the threshold potential (with  $V_{th} > V_r$ ), and  $R$ ,  $E$ ,  $\tau$  the membrane resistance, potential and time constant. The most commonly used activation func-

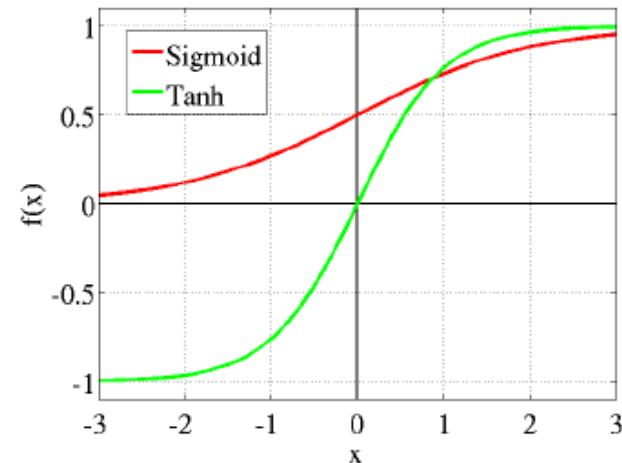
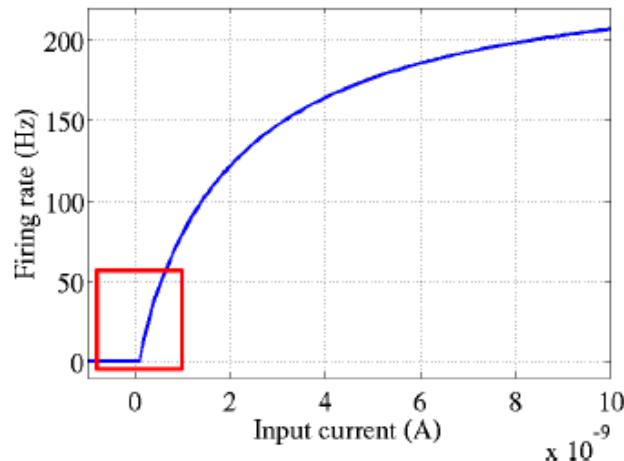
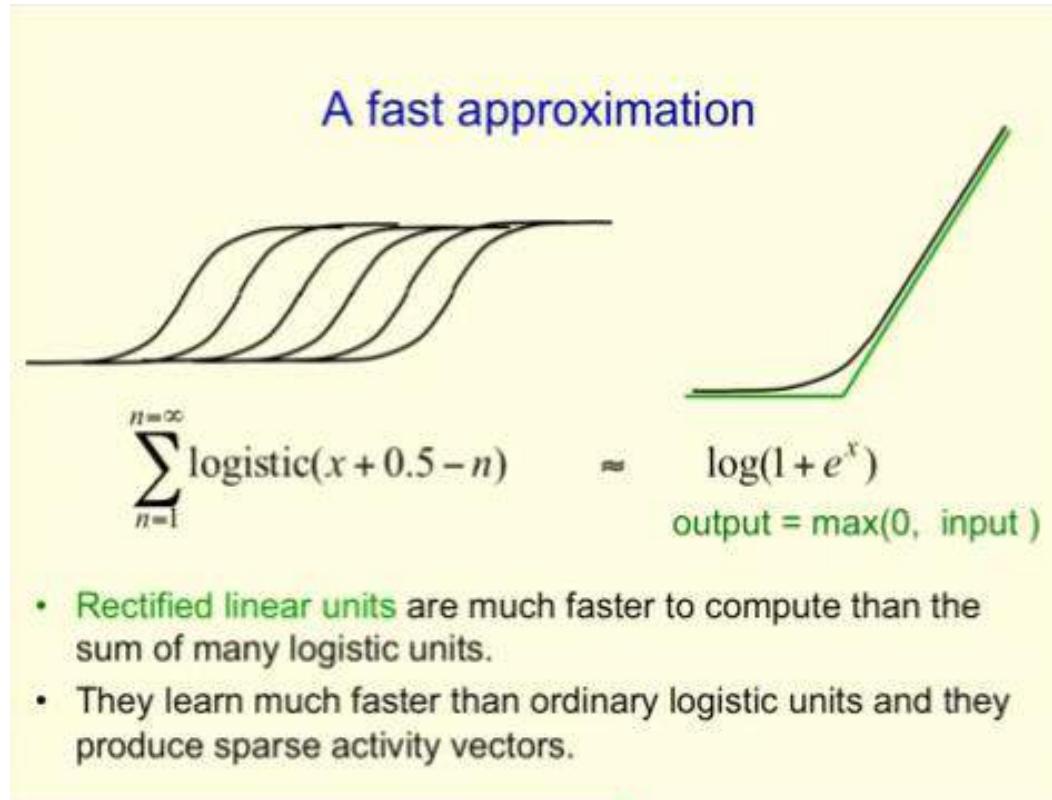


Figure 1: *Left:* Common neural activation function motivated by biological data. *Right:* Commonly used activation functions in neural networks literature: logistic sigmoid and hyperbolic tangent (*tanh*).

# Rectified Linear Units: Another Perspective

Hinton argues that this is a form of model averaging



# ReLU: Pros and Cons

- Pros:
  - It converges much faster (claimed to be 6x faster than sigmoid/tanh)
    - It overfits very fast and when used with e.g. dropout, this leads to very fast convergence
  - It is simpler and faster to compute (simple comparison)
- Cons:
  - A ReLU neuron may “die” during training
  - A large gradient may update the weights such that the ReLU neuron may never activate again
    - Avoid large learning rate

# ReLU

- See the following site for more in-depth analysis

<http://www.jefkine.com/general/2016/08/24/formulating-the-relu/>

# Leaky ReLU

- $f(x) = \mathbf{1}(x < 0)(\alpha x) + \mathbf{1}(x \geq 0)(x)$ 
  - When  $x$  is negative, have a non-zero slope ( $\alpha$ )
- If you learn  $\alpha$  during training, this is called parametric ReLU (PReLU)

Kaiming He, Xiangyu Zhang, Shaoqing Ren,  
Jian Sun (2015) Delving Deep into  
Rectifiers: Surpassing Human-Level  
Performance on ImageNet Classification

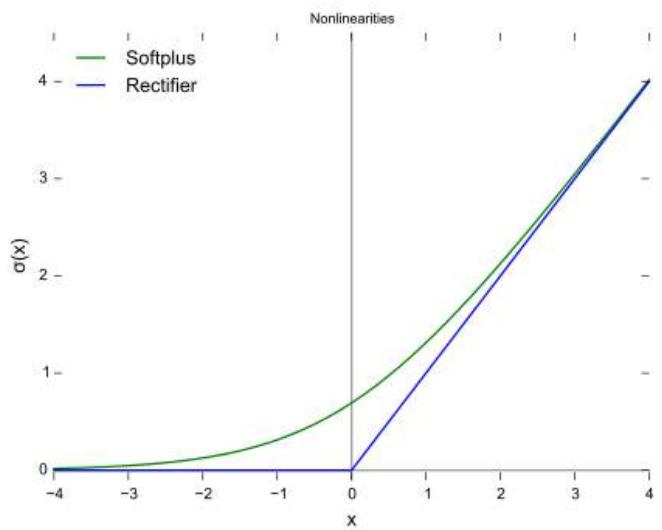
# Maxout

“Maxout Networks” by Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, Yoshua Bengio, 2013.

- $\max(w_1^T x + b_1, w_2^T x + b_2)$
- ReLU, Leaky ReLU and PReLU are special cases of this
- Drawback: More parameters to learn!

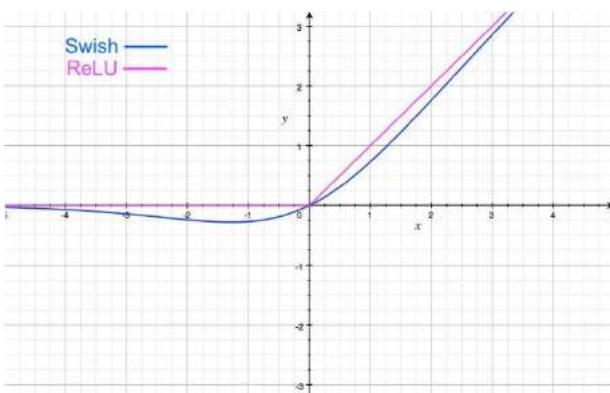
# Softplus

- A smooth approximation to the ReLU unit:  
 $f(x) = \ln(1 + e^x)$
- Its derivative is the sigmoid function:  
 $f'(x) = 1/(1 + e^{-x})$



# Swish: A Self-Gated Activation Function

*"The choice of activation functions in deep networks has a significant effect on the training dynamics and task performance. Currently, the most successful and widely-used activation function is the Rectified Linear Unit (ReLU). Although various alternatives to ReLU have been proposed, none have managed to replace it due to inconsistent gains. In this work, we propose a new activation function, named Swish, which is simply  $f(x)=x \cdot \text{sigmoid}(x)$ . Our experiments show that Swish tends to work better than ReLU on deeper models across a number of challenging datasets. For example, simply replacing ReLUs with Swish units improves top-1 classification accuracy on ImageNet by 0.9% for Mobile NASNet-A and 0.6% for Inception-ResNet-v2. The simplicity of Swish and its similarity to ReLU make it easy for practitioners to replace ReLUs with Swish units in any neural network."*



## SEARCHING FOR ACTIVATION FUNCTIONS

Prajit Ramachandran\*, Barret Zoph, Quoc V. Le  
Google Brain  
[{prajit,barretzoph,qvl}@google.com](mailto:{prajit,barretzoph,qvl}@google.com)

# Activation Functions: To sum up

- Don't use sigmoid
- If you really want, use tanh but it is worse than ReLU and its variants
- ReLU: be careful about dying neurons
- Leaky ReLU and Maxout: Worth trying

# DEMO

- <http://playground.tensorflow.org/#activation=tanh&regularization=L2&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.24725&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification>

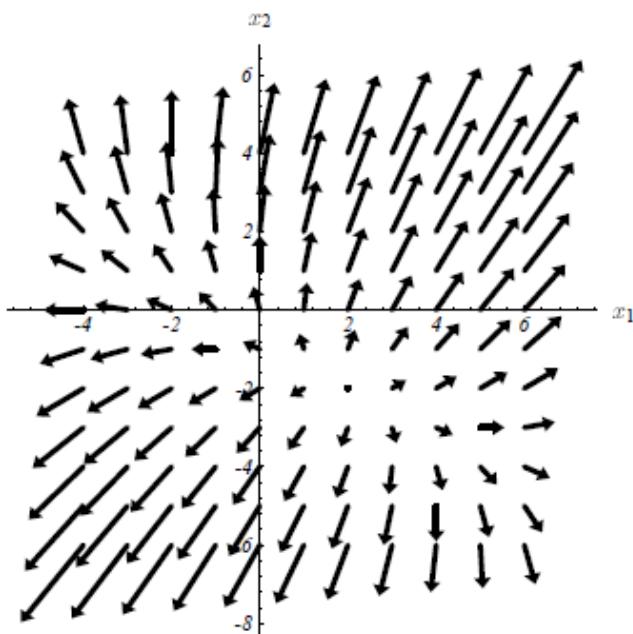
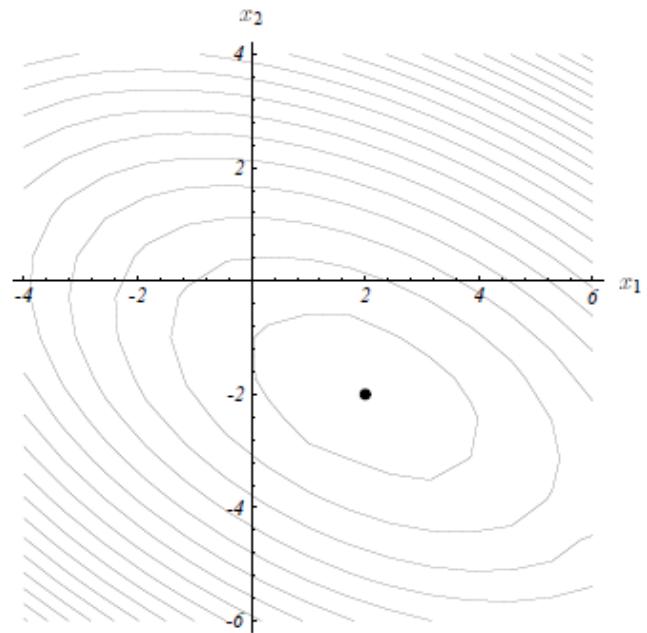
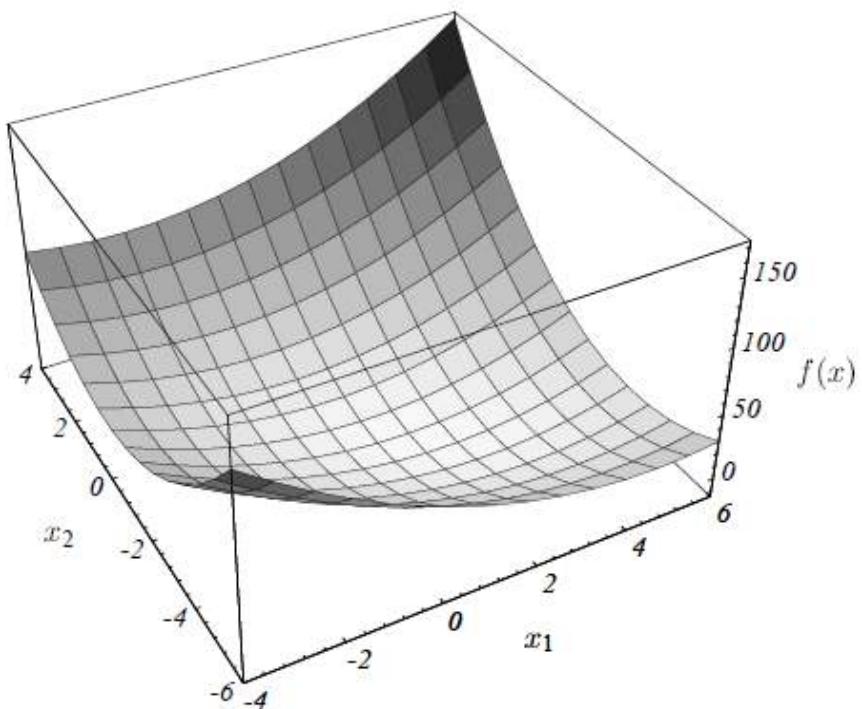
# Interactive introductory tutorial

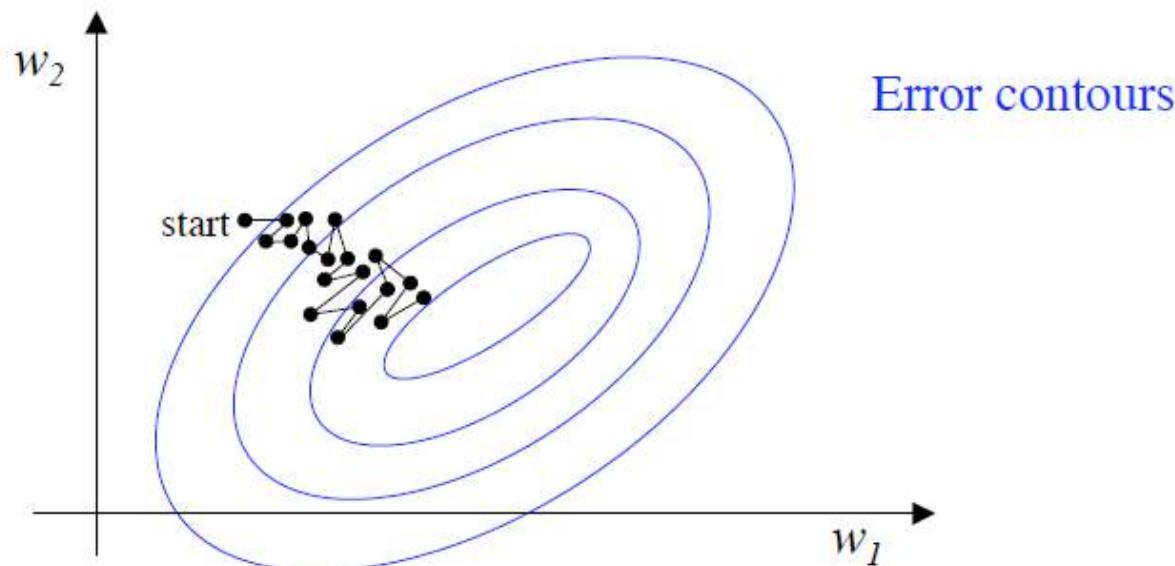
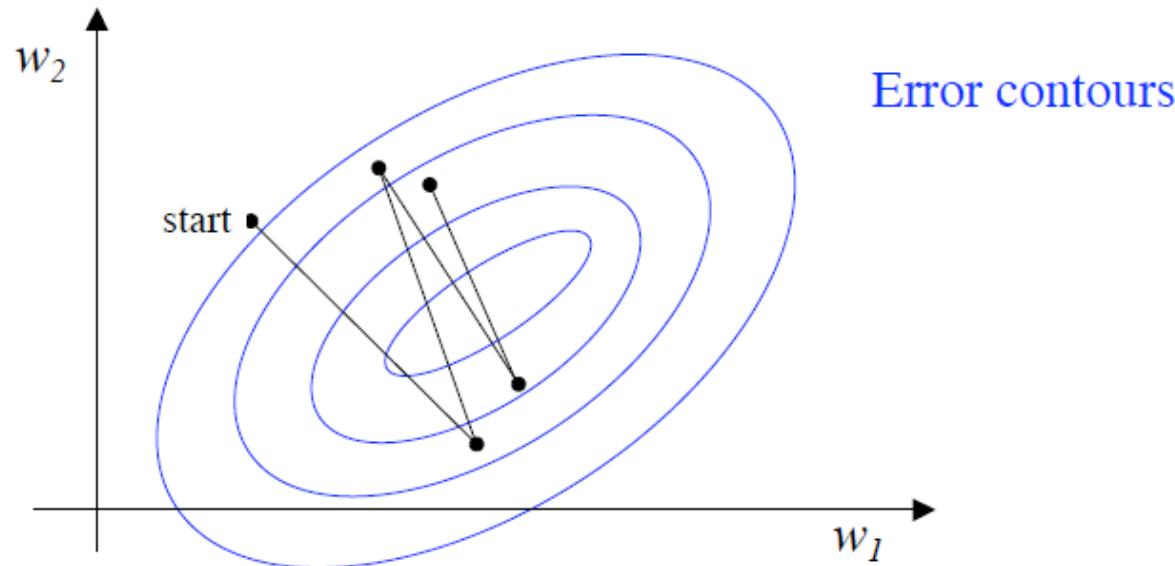
<https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/>

# **BACK PROPAGATION / MINIMIZATION STRATEGIES**

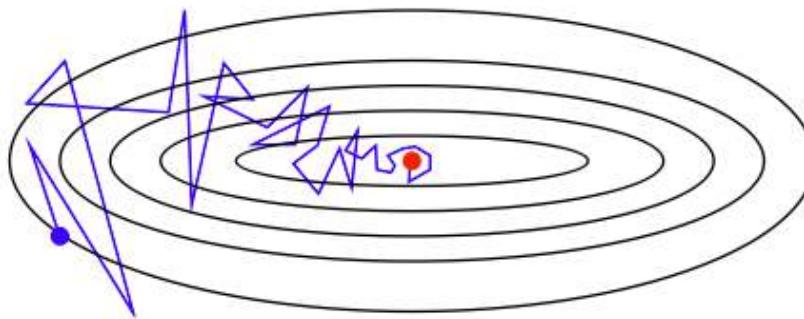
# Schemes of training

- True/Standard Gradient Descent
- Stochastic Gradient Descent
- Steepest Gradient Descent
- Momentum Gradient Descent
- Curricular training

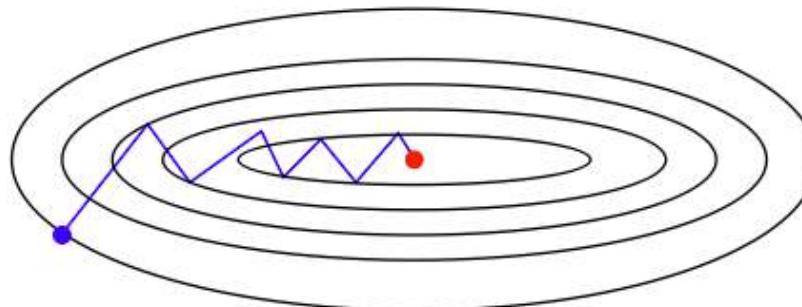




Stochastic Gradient Descent



Batch Gradient Descent



forth the following as possible causes for this phenomenon: (i) LB methods over-fit the model; (ii) LB methods are attracted to saddle points; (iii) LB methods lack the *explorative* properties of SB methods and tend to zoom-in on the minimizer closest to the initial point; (iv) SB and LB methods converge to qualitatively different minimizers with differing generalization properties. The data presented in this paper supports the last two conjectures.

The main observation of this paper is as follows:

The lack of generalization ability is due to the fact that large-batch methods tend to converge to *sharp minimizers* of the training function. These minimizers are characterized by a significant number of large positive eigenvalues in  $\nabla^2 f(x)$ , and tend to generalize less well. In contrast, small-batch methods converge to *flat minimizers* characterized by having numerous small eigenvalues of  $\nabla^2 f(x)$ . We have observed that the loss function landscape of deep neural networks is such that large-batch methods are attracted to regions with sharp minimizers and that, unlike small-batch methods, are unable to escape basins of attraction of these minimizers.

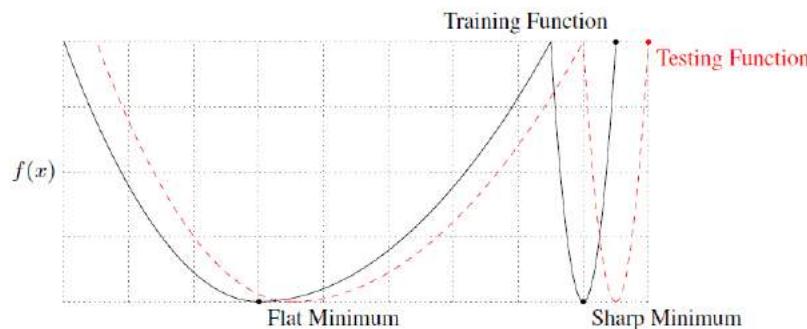


Figure 1: A Conceptual Sketch of Flat and Sharp Minima. The Y-axis indicates value of the loss function and the X-axis the variables (parameters)

---

## On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima

---

Nitish Shirish Keskar\*  
Northwestern University  
Evanston, IL 60208  
[keskar.nitish@u.northwestern.edu](mailto:keskar.nitish@u.northwestern.edu)

Dheevatsa Mudigere  
Intel Corporation  
Bangalore, India  
[dheevatsa.mudigere@intel.com](mailto:dheevatsa.mudigere@intel.com)

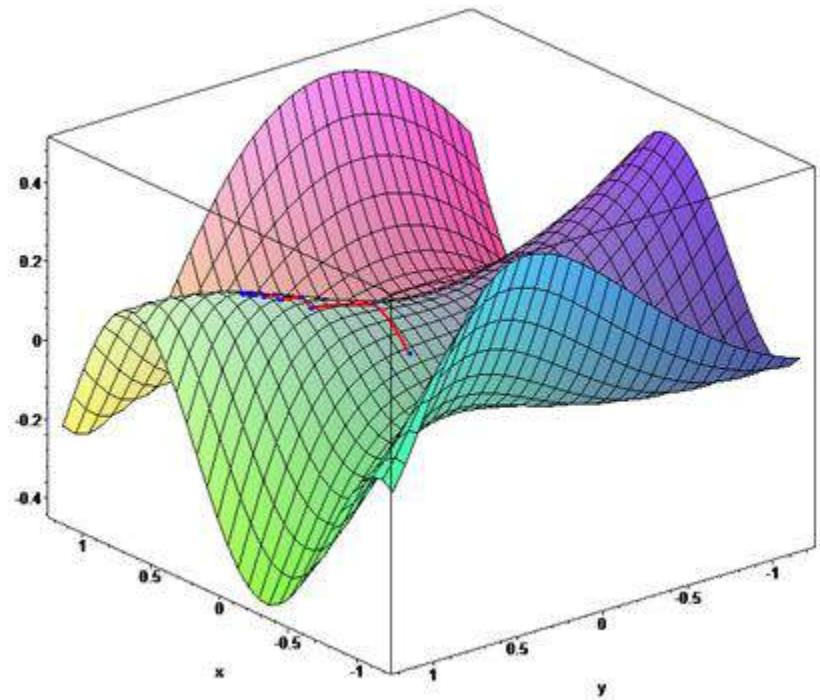
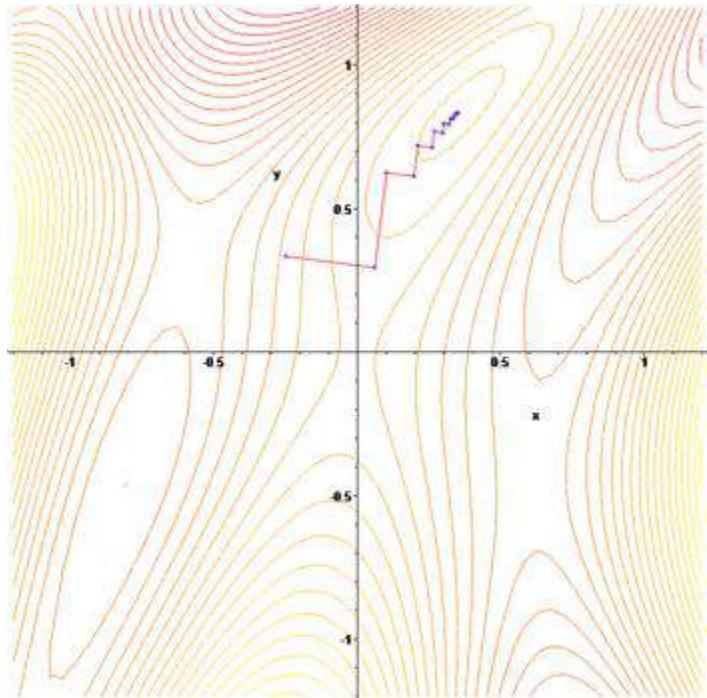
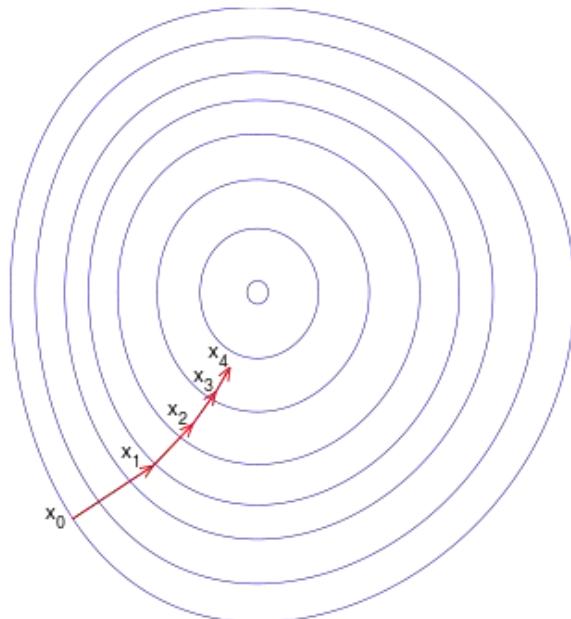
Jorge Nocedal  
Northwestern University  
Evanston, IL 60208  
[j-nocedal@northwestern.edu](mailto:j-nocedal@northwestern.edu)

Mikhail Smelyanskiy  
Intel Corporation  
Santa Clara, CA 95054  
[mikhail.smelyanskiy@intel.com](mailto:mikhail.smelyanskiy@intel.com)

Ping Tak Peter Tang  
Intel Corporation  
Santa Clara, CA 95054  
[peter.tang@intel.com](mailto:peter.tang@intel.com)

2017

# Gradient descent



# Second order methods

- Newton's method for optimization:

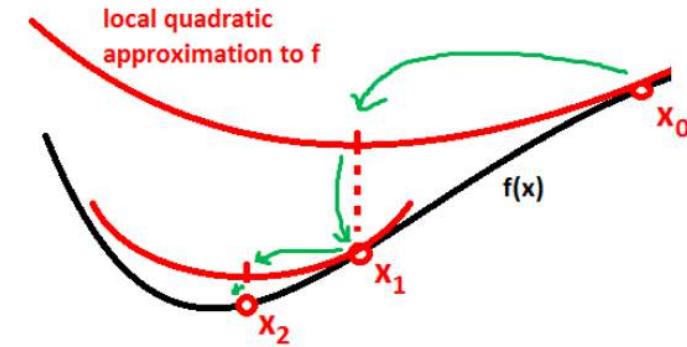
- $w \leftarrow w - [Hf(w)]^{-1} \nabla f(w)$

- where  $Hf(w)$  is the Hessian

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

- Hessian gives a better feeling about the surface
  - It gives information about the curvature of surface

# Intuition behind Newton's method



- Newton's method assumes that the function ( $f(x)$ ) we are trying to minimize is quadratic, and aims to find the minimum ( $x + \delta$ ), where  $f'(x + \delta) = 0$ .
- From Taylor expansion:

$$f(x + \delta) = f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2$$

- Solving for  $\delta$ :

$$\frac{d}{d\delta} \left[ f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2 \right] = 0$$

which yields:

$$\delta = -f'(x)/f''(x)$$

- In high-dimensional cases,  $f'(x)$  is replaced by  $\nabla f(x)$  and  $f''(x)$  by  $Hf(x)$ .

# Compare this to Newton's method for finding the roots

- To find a root  $r$  of a function ( $f(x)$ ), i.e.,  $f(r) = 0$ :

$$x_{k+1} = x_k + \frac{f(x_k)}{f'(x_k)}$$

- In optimization, we wish to end up with  $f'(x) = 0$  with:

$$x_{k+1} = x_k + \frac{f'(x_k)}{f''(x_k)}$$

# Newton's method for optimization

- $w \leftarrow w - [Hf(w)]^{-1} \nabla f(w)$ 
  - Makes bigger steps in shallow curvature
  - Smaller steps in steep curvature
- Note that there is no hyper-parameter! (if you wish you can add a step size, but this is not necessary)
- Disadvantage:
  - Too much memory requirement
  - For 1 million parameters, this means a matrix of 1 million x 1 million → ~ **3725 GB RAM**
  - Alternatives exist to get around the memory problem (quasi-Newton methods, Limited-memory BFGS)
    - Active research area → A suitable project topic 😊

# RPROP (Resilience Propagation)

- Instead of the magnitude, use the sign of the gradients

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & , \text{ else} \end{cases} \quad (4)$$

where  $0 < \eta^- < 1 < \eta^+$

- Motivation: If the sign of a gradient has changed, that means we have “overshot” a minima
- Advantage: Faster to run/converge
- Disadvantage: More complex to implement

A Direct Adaptive Method for Faster Backpropagation Learning:  
The RPROP Algorithm

# RPROP (Resilience Propagation)

```
For all weights and biases{  
    if ( $\frac{\partial E}{\partial w_{ij}}(t - 1) * \frac{\partial E}{\partial w_{ij}}(t) > 0$ ) then {  
         $\Delta_{ij}(t) = \text{minimum}(\Delta_{ij}(t - 1) * \eta^+, \Delta_{max})$   
         $\Delta w_{ij}(t) = -\text{sign}(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t)$   
         $w_{ij}(t + 1) = w_{ij}(t) + \Delta w_{ij}(t)$   
    }  
    else if ( $\frac{\partial E}{\partial w_{ij}}(t - 1) * \frac{\partial E}{\partial w_{ij}}(t) < 0$ ) then {  
         $\Delta_{ij}(t) = \text{maximum}(\Delta_{ij}(t - 1) * \eta^-, \Delta_{min})$   
         $w_{ij}(t + 1) = w_{ij}(t) - \Delta w_{ij}(t - 1)$   
         $\frac{\partial E}{\partial w_{ij}}(t) = 0$   
    }  
    else if ( $\frac{\partial E}{\partial w_{ij}}(t - 1) * \frac{\partial E}{\partial w_{ij}}(t) = 0$ ) then {  
         $\Delta w_{ij}(t) = -\text{sign}(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t)$   
         $w_{ij}(t + 1) = w_{ij}(t) + \Delta w_{ij}(t)$   
    }  
}
```

A Direct Adaptive Method for Faster Backpropagation Learning:  
The RPROP Algorithm

# Gradient Descent with Line Search

- Gradient descent:

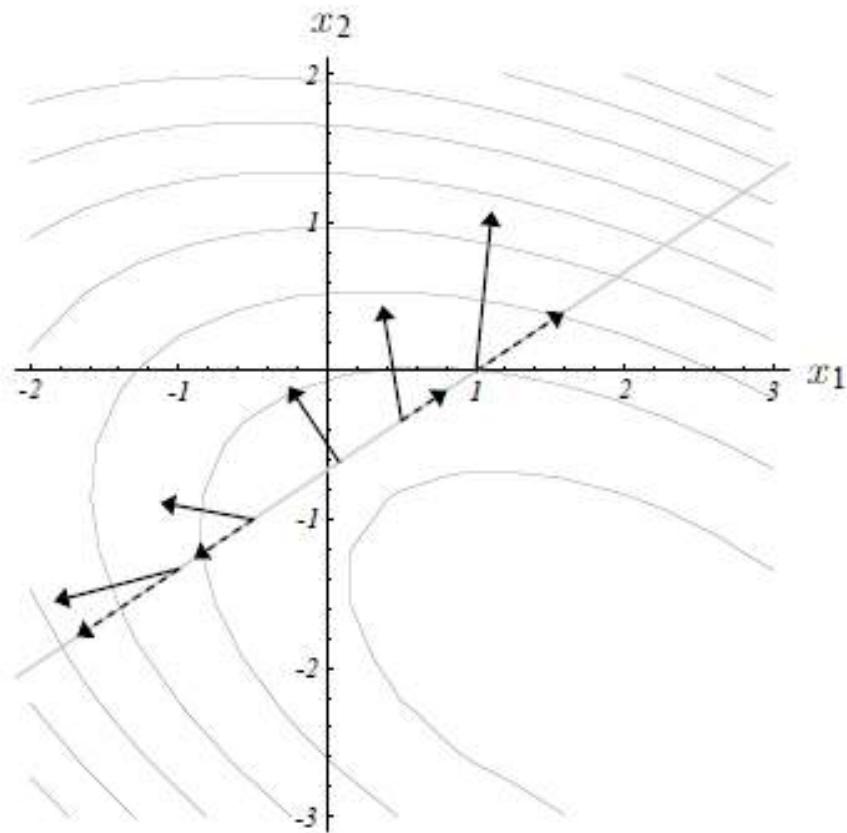
$$w_{ij}^t = w_{ij}^{t-1} + s \ dir_{ij}^{t-1}$$

where  $dir_{ij}^{t-1} = -\partial L / \partial w_{ij}$

- Gradient descent with line search:

- Choose  $s$  such that  $L$  is minimized along  $dir_{ij}^{t-1}$ .

- Set  $\frac{dL(w_{ij}^t)}{ds} = 0$  to find the optimal  $s$ .



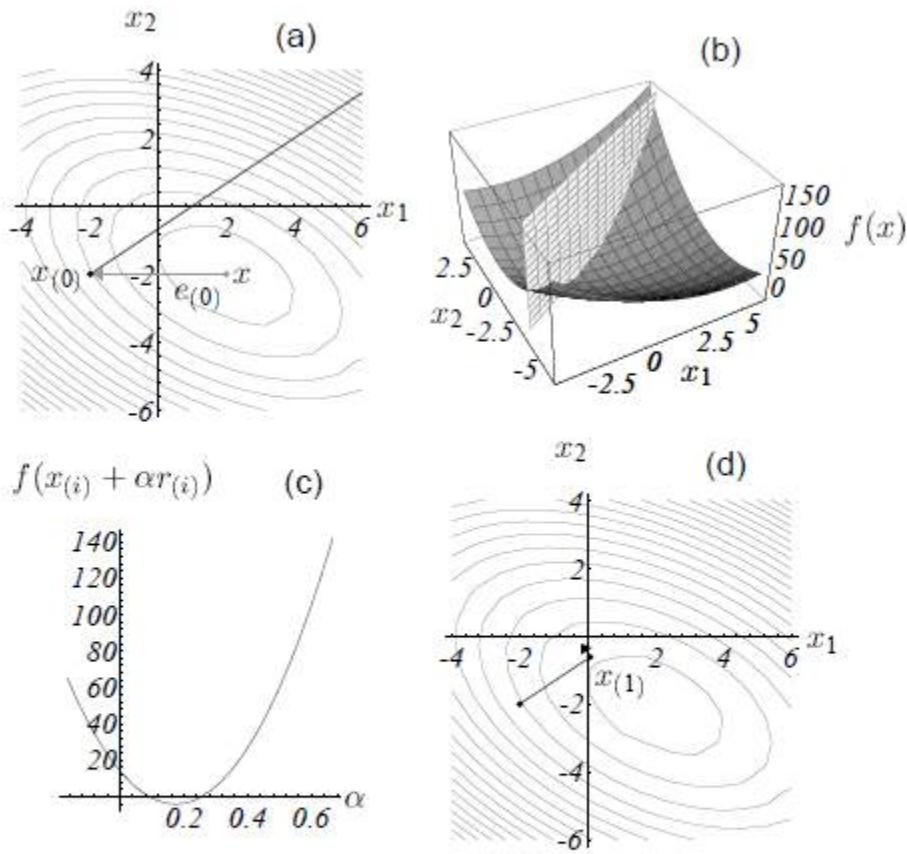


Figure 6: The method of Steepest Descent. (a) Starting at  $[-2, -2]^T$ , take a step in the direction of steepest descent of  $f$ . (b) Find the point on the intersection of these two surfaces that minimizes  $f$ . (c) This parabola is the intersection of surfaces. The bottommost point is our target. (d) The gradient at the bottommost point is orthogonal to the gradient of the previous step.

# Gradient Descent with Line Search

$$w_{ij}^t = w_{ij}^{t-1} + s \ dir_{ij}^{t-1}$$

- Set  $\frac{dL(w_{ij}^t)}{ds} = 0$  to find the optimal  $s$ .
- $\frac{dL(w_{ij}^t = w_{ij}^{t-1} + s \ dir_{ij}^{t-1})}{ds} = \frac{dL}{dw_{ij}^t} \frac{dw_{ij}^t}{ds} = \frac{dL}{dw_{ij}^t} dir_{ij}^{t-1} = 0$

$$\frac{dL}{dw_{ij}^t} \frac{dw_{ij}^t}{ds} = \frac{dL}{dw_{ij}^t} dir_{ij}^{t-1} = 0$$

- Interpretation:
  - Choose  $s$  such that: the gradient direction at the new position is orthogonal to the current direction
- This is called **steepest gradient descent**
- Problem: makes zig-zag

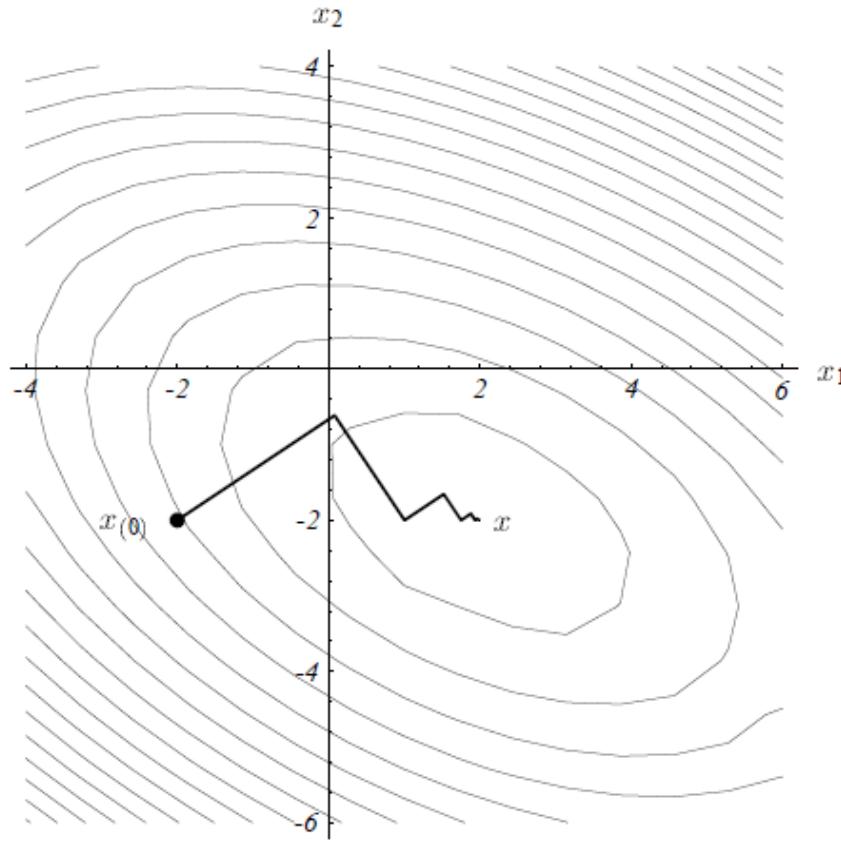
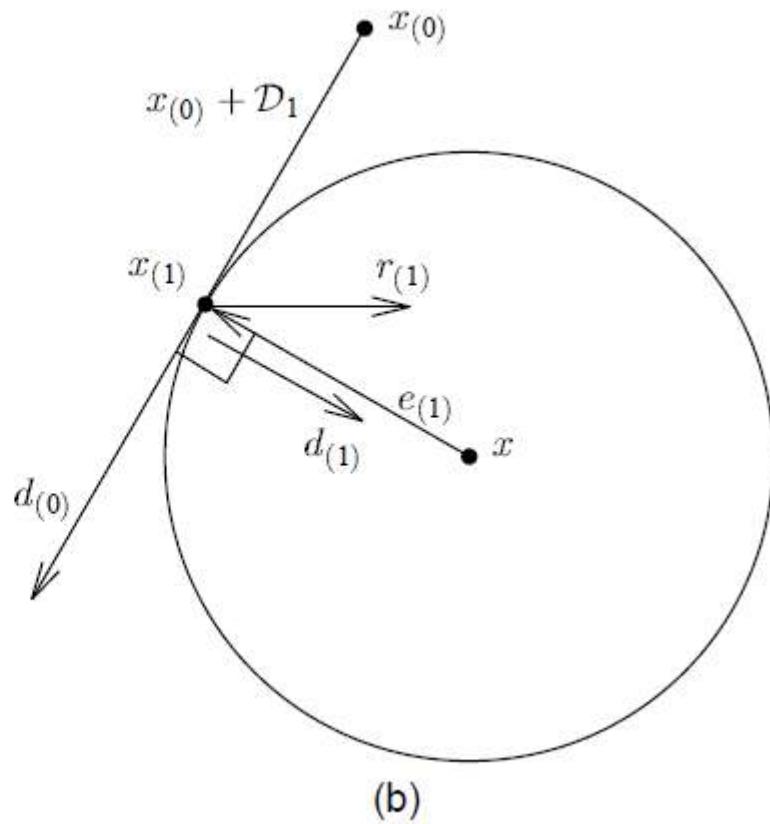
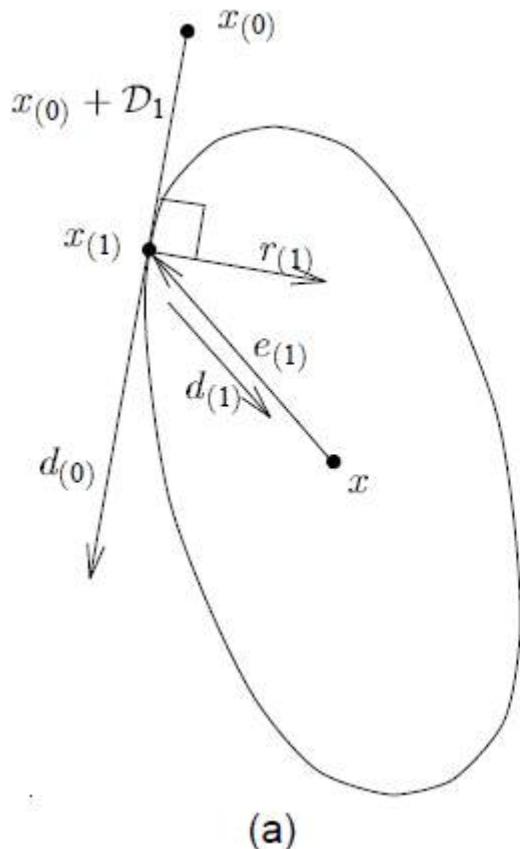


Figure 8: Here, the method of Steepest Descent starts at  $[-2, -2]^T$  and converges at  $[2, -2]^T$ .

# Conjugate Gradient Descent

- Motivation



# Conjugate Gradient Descent

- Two vectors are conjugate ( $A$ -orthogonal) if:  
$$u^T A v = 0$$
- We assume that the error surface has the quadratic form:

$$f(x) = \frac{1}{2} x^T A x - b^T x + c$$

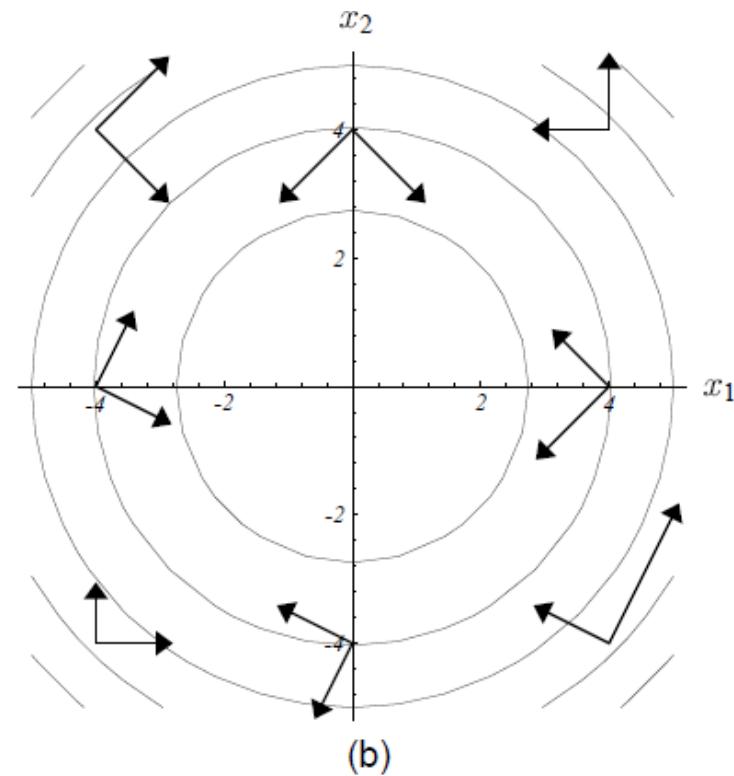
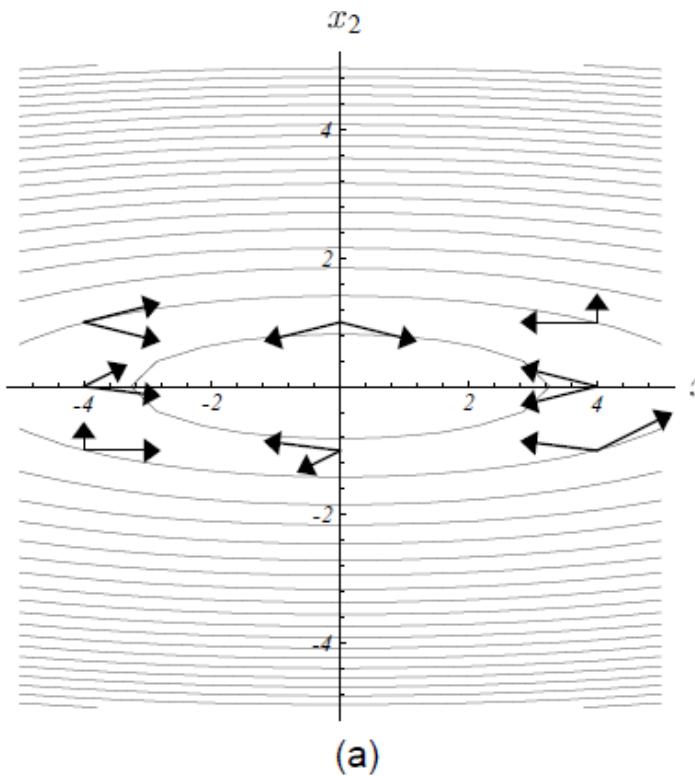


Figure 22: These pairs of vectors are  $A$ -orthogonal . . . because these pairs of vectors are orthogonal.

# Conjugate Gradient Descent

- $dir_{ij}^t = -\frac{\partial E(w_{ij}^t)}{\partial w_{ij}^t} + \beta dir_{ij}^{t-1}$
- By assuming quadratic form etc.:

$$\beta = \frac{\sum_{i,j} \left( \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} - \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} \right) \cdot \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}}{\sum_{i,j} \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} \cdot \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}}$$

# Conjugate Gradient Descent

- Or simply as:

$$\beta = \frac{(\nabla E_{new} - \nabla E_{old}) \cdot \nabla E_{new}}{(\nabla E_{old})^2}$$

- Interpretation:

- Rewrite this as:

$$\beta = \frac{\nabla E_{new}^2}{\nabla E_{old}^2} - \frac{\nabla E_{old} \cdot \nabla E_{new}}{\nabla E_{old}^2}$$

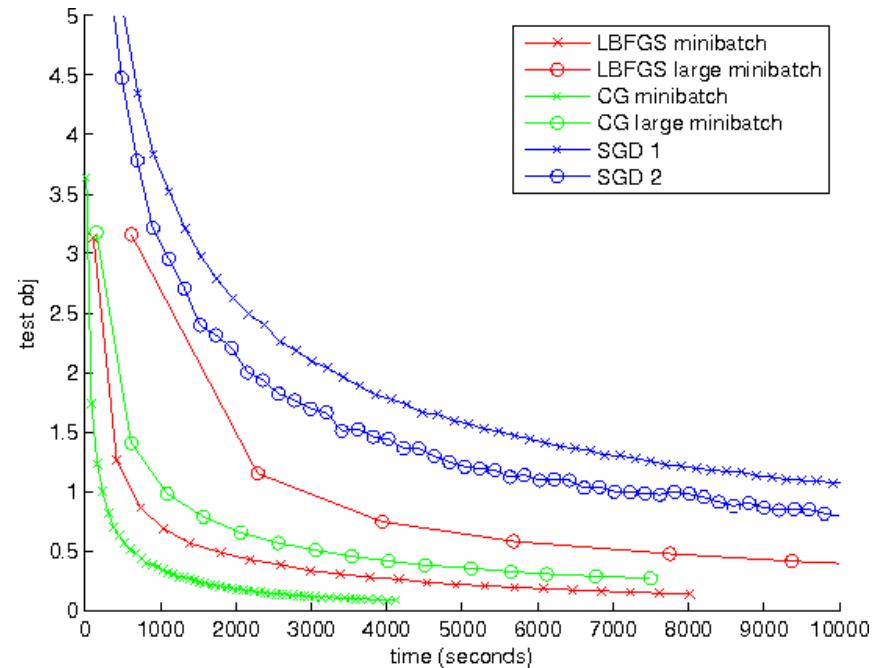
- If the new direction suggests a radical turn, rely more on the old direction!

- For more detailed motivation and derivations, see:

Jonathan Richard Shewchuk, “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain”, 1994.

# Steepest and Conjugate Gradient Descent: Cons and Pros

- Pros:
  - Faster to converge than, e.g., stochastic gradient descent (even mini-batch)
- Cons:
  - They don't work well on saddle points
  - Computationally more expensive
  - In 2D:
    - Steepest descent is  $O(n^2)$
    - Conjugate descent is  $O(n^{3/2})$

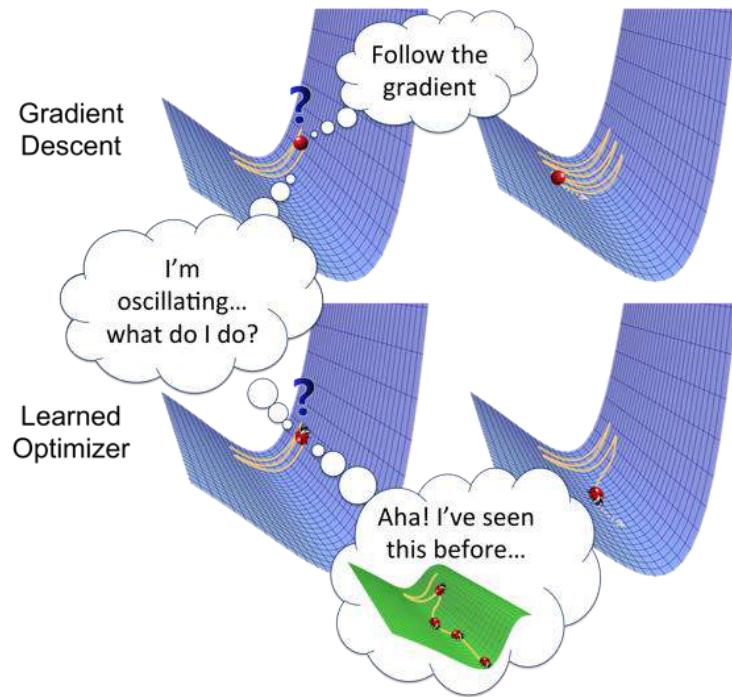


Le et al., “On optimization methods for deep learning”, 2011.

# Online Interactive Tutorial

<http://www.benfrederickson.com/numerical-optimization/>

- <http://bair.berkeley.edu/blog/2017/09/12/learning-to-optimize-with-rl>



# Genetic Algorithms

General strategy:

- Randomly choose weights and encode them on a string of bits (chromosomes).
- Determine a “fitness” function (e.g. error function).
- Use genetic operators \*mutation, crossover) to construct new strings.
- Use “survival of the fittest” to produce better and better strings.

Some observations/comments:

- Selection of fitness and operators is crucial to its effectiveness.
- Search is global, not fooled by local minima.
- Fitness (or error in this case) function need not be differentiable.
- Search is rather blind, since it does not use the  $\nabla$  info.
- It can be a good method for initialization, to be used for a gradient method.

# **CHALLENGES OF THE ERROR SURFACE**

# Challenges

- Local minima
- Saddle points
- Cliffs
- Valleys

# Local minima

- Solutions
  - Momentum
    - Make weight update depend on the previous one as well:
$$\Delta w_{ij}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n - 1)$$
    - $0 \leq \alpha < 1$ : momentum (constant)
  - Incremental update
  - Large training data
  - Adaptive learning rate
  - Good initialization
  - Different minimization strategies

- For smaller networks, local minima are more problematic
  - For large-size networks, most local minima are equivalent and yield similar performance on a test set.
  - The probability of finding a “bad” (high value) local minimum is non-zero for small-size networks and decreases quickly with network size.
  - Struggling to find the global minimum on the training set (as opposed to one of the many good local ones) is not useful in practice and may lead to overfitting.

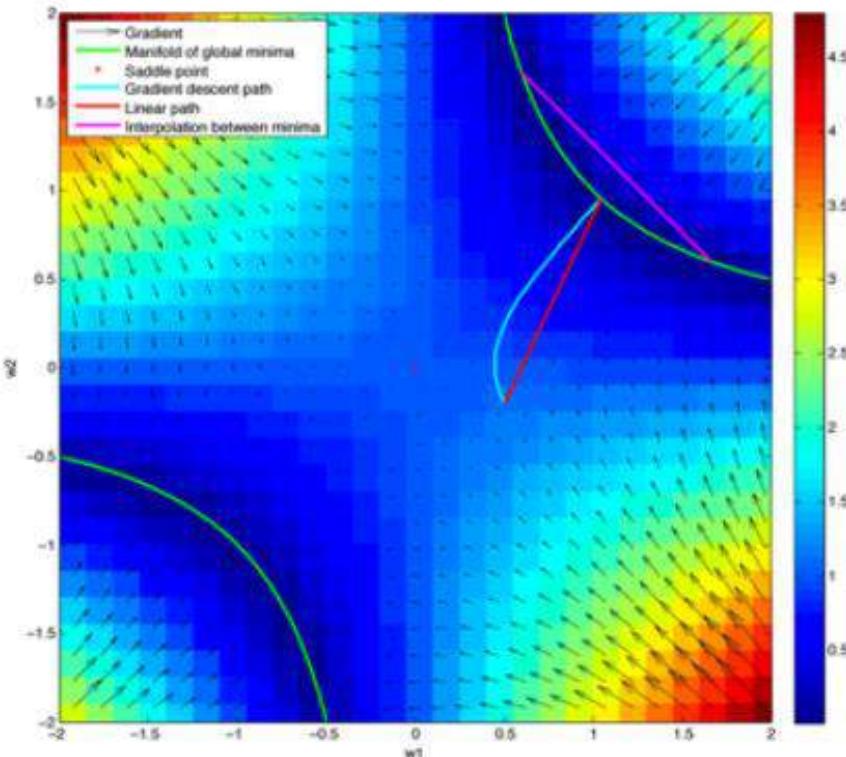
---

## The Loss Surfaces of Multilayer Networks

---

# Do neural nets have saddle points?

- Saxe et al, 2013:
- neural nets without nonlinearities have many saddle points
- all the minima are global
- all the minima form a connected manifold



## Do neural nets have saddle points?

- Dauphin et al 2014: Experiments show neural nets do have as many saddle points as random matrix theory predicts
- Choromanska et al 2015: Theoretical argument for why this should happen
- Major implication: **most minima are good, and this is more true for big models.**
- Minor implication: the reason that *Newton's method* works poorly for neural nets is its attraction to the ubiquitous saddle points.

# Valleys, Cliffs and Exploding Gradients

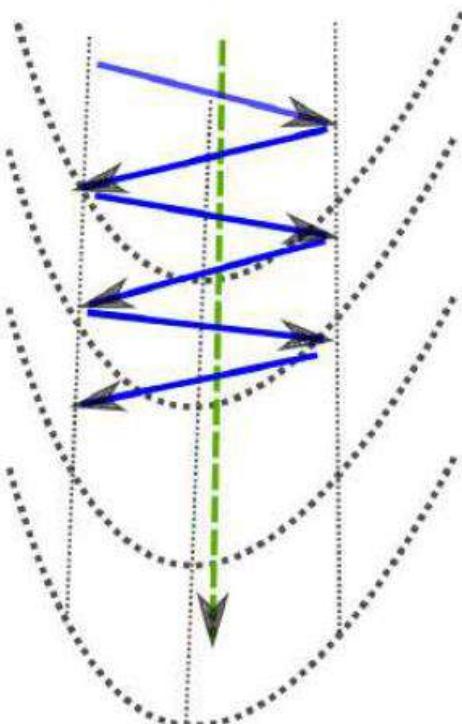


Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the quickly rising sides of the valley (going left or right), and other directions have a low curvature, corresponding to the smooth slope of the valley (going down, dashed arrow). Most second-order methods, as well as momentum or gradient averaging methods are meant to address that problem, by increasing the step size in the direction of the valley (where it pays off the most in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations (blue full arrows). The objective is to smoothly go down, staying at the bottom of the valley (green dashed arrow).

# Valleys, Cliffs and Exploding Gradients

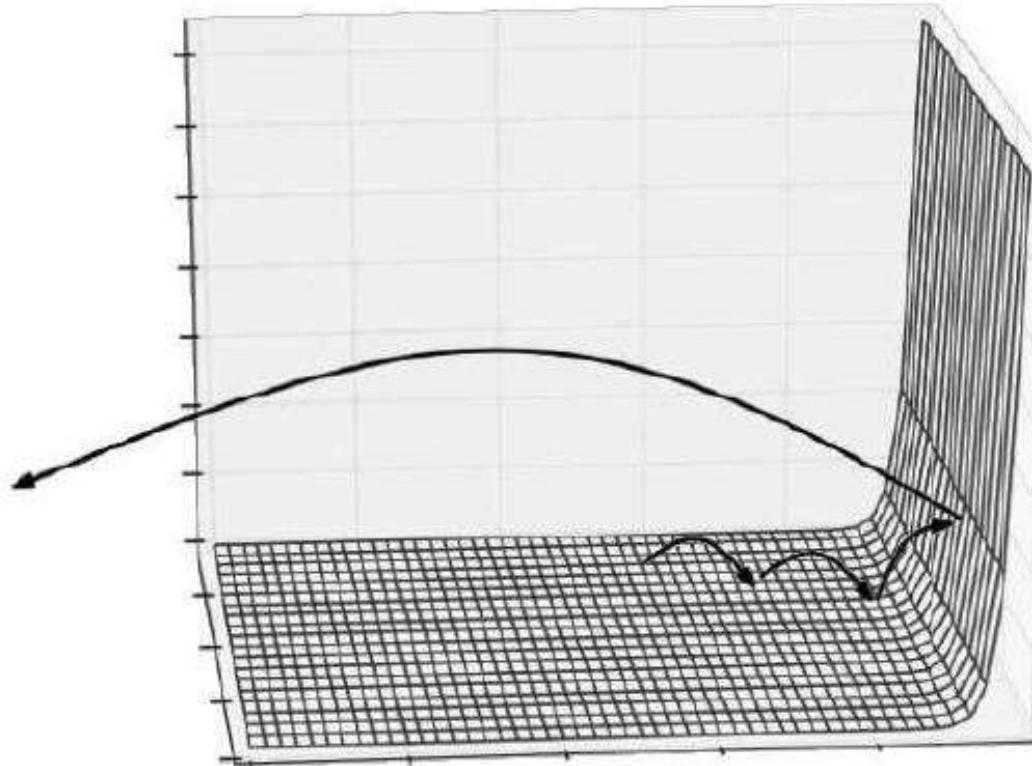


Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly ruining a lot of the optimization work that had been done. Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

# Valleys, Cliffs and Exploding Gradients

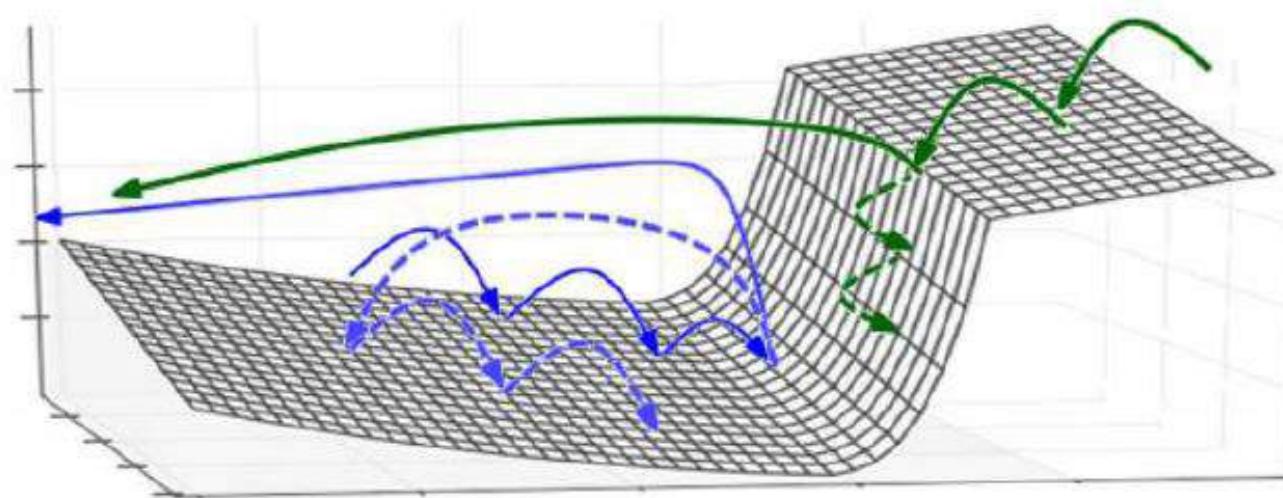


Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (dotted arrows trajectories) helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below (bold arrows trajectories). Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

# **USING MOMENTUM TO IMPROVE STEPS**

# Momentum

- Maintain a “memory”

$$\Delta w(t + 1) \leftarrow \mu \Delta w(t) - \eta \nabla E$$

where  $\mu$  is called the momentum term

- Momentum filters oscillations on gradients (i.e., oscillatory movements on the error surface)
- $\mu$  is typically initialized to 0.9.
  - It is better if it anneals from 0.5 to 0.99 over multiple epochs

# Momentum

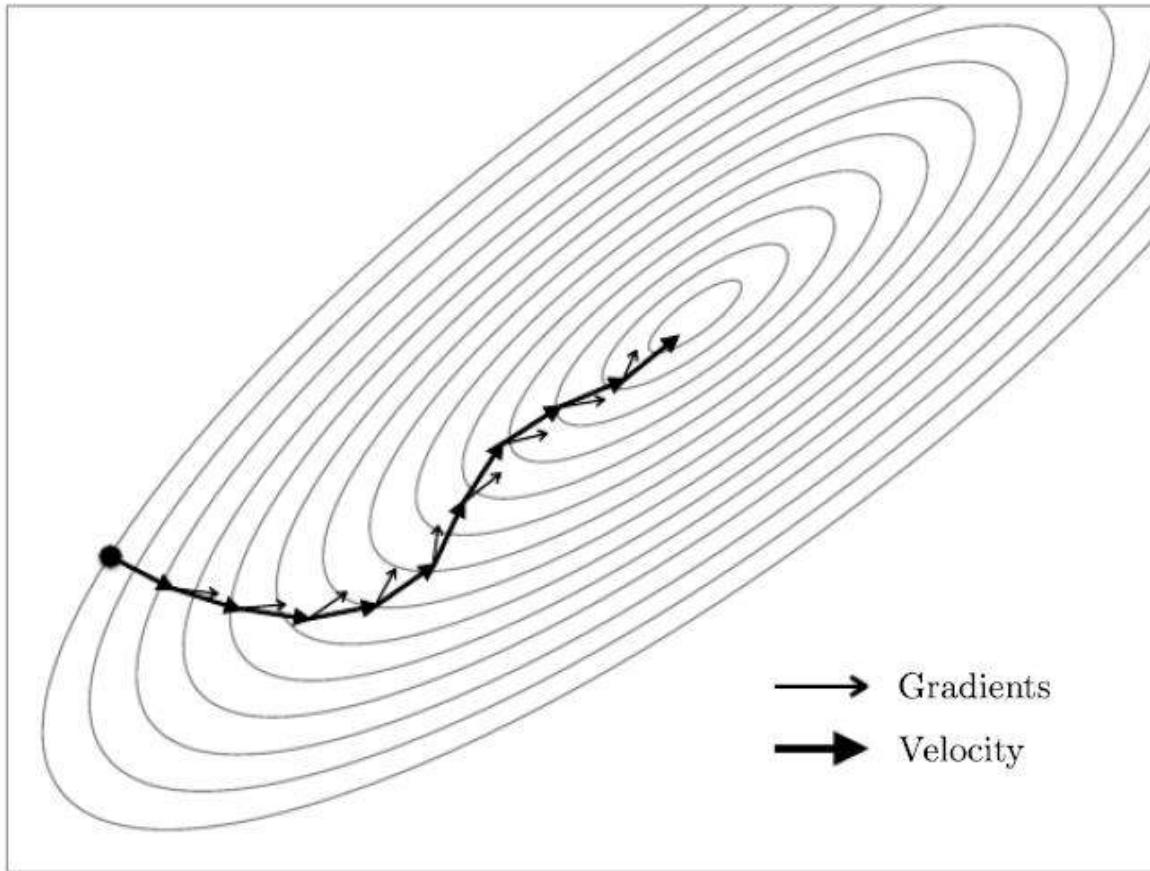


Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.

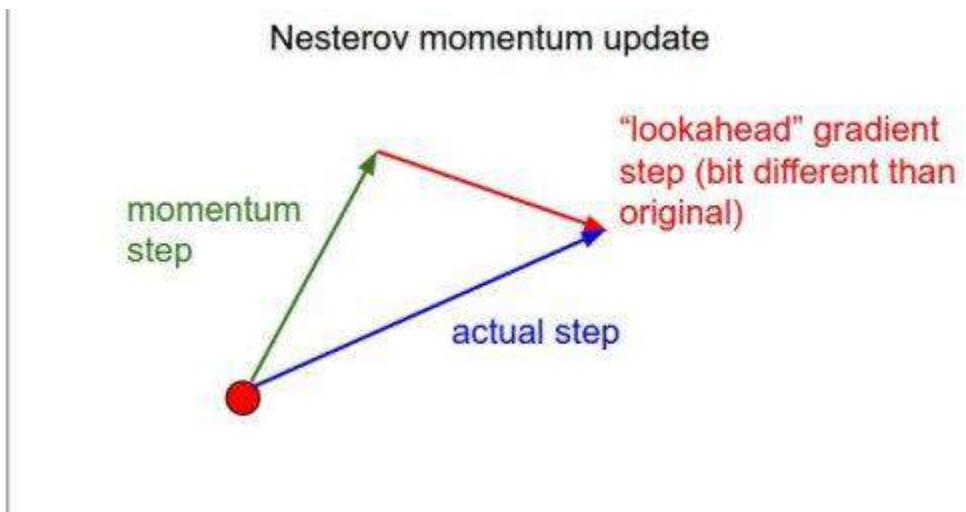
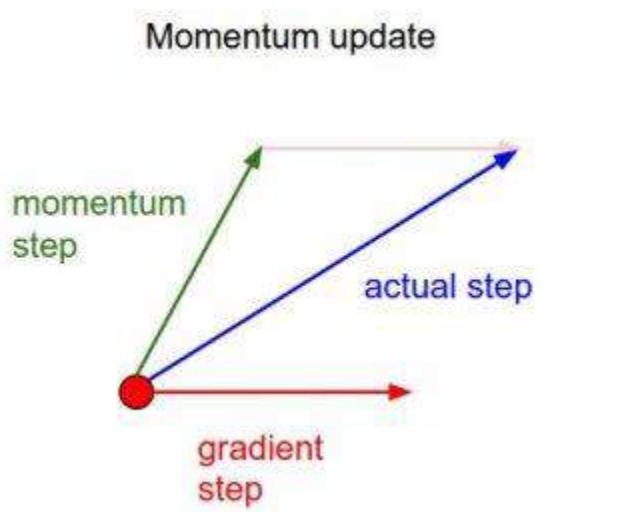
# Nesterov Momentum

- Use a “lookahead” step to update:

$$w_{\text{ahead}} \leftarrow w + \mu \Delta w(t)$$

$$\Delta w(t+1) \leftarrow \mu \Delta w(t) - \eta \nabla E_{\text{ahead}}$$

$$w \leftarrow w + \Delta w(t+1)$$



# Momentum vs. Nesterov Momentum

- When the learning rate is very small, they are equivalent.
  - When the learning rate is sufficiently large, Nesterov Momentum performs better (it is more responsive).
  - See for an in-depth comparison:
- 

On the importance of initialization and momentum in deep learning

---

Ilya Sutskever<sup>1</sup>  
James Martens  
George Dahl  
Geoffrey Hinton

ILYASU@GOOGLE.COM  
JMARTENS@CS.TORONTO.EDU  
GDAHL@CS.TORONTO.EDU  
HINTON@CS.TORONTO.EDU

# Demo (and further reading)

<http://distill.pub/2017/momentum/>

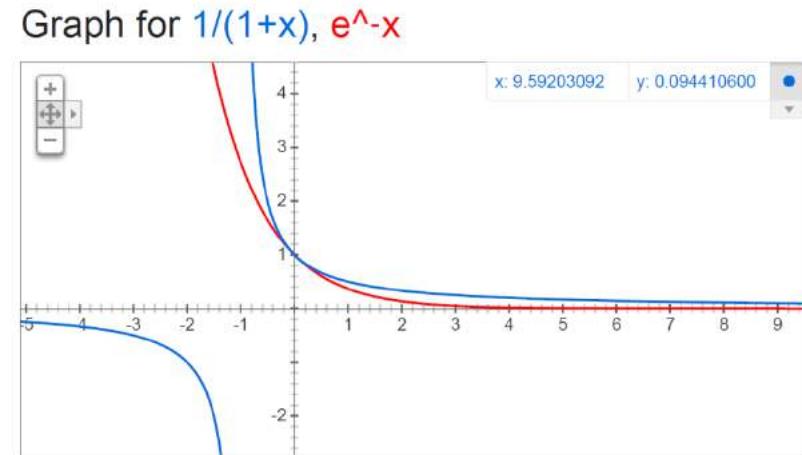
# **SETTING THE LEARNING RATE**

# Alternatives

- Single global learning rate
  - Adaptive Learning Rate
  - Adaptive Learning Rate with Momentum
- Per-parameter learning rate
  - AdaGrad
  - RMSprop
  - Adam
  - AdaDelta

# Annealing the learning rate (Global)

- Step decay
  - $\eta' \leftarrow \eta \times c$ , where  $c$  could be 0.5, 0.4, 0.3, 0.2, 0.1 etc.
- Exponential decay:
  - $\eta = \eta_0 e^{-kt}$ , where  $t$  is iteration number
  - $\eta_0, k$ : hyperparameters
- $1/t$  decay:
  - $\eta = \eta_0 / (1 + kt)$
- If you have time, keep decay small and train longer



# Adagrad (Per parameter)

- Higher the gradient, lower the learning rate
- Accumulate square of gradients elementwise (initially  $r = 0$ ):

$$r \leftarrow r + \left( \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2$$

- Update each parameter/weight based on the gradient on that:

$$\Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W}$$

---

**Algorithm 8.4** The Adagrad algorithm

---

**Require:** Global learning rate  $\eta$ ,

**Require:** Initial parameter  $\theta$

Initialize gradient accumulation variable  $r = 0$ ,  
while Stopping criterion not met do

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .

    Set  $\mathbf{g} = \mathbf{0}$

**for**  $i = 1$  to  $m$  **do**

        Compute gradient:  $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

**end for**

    Accumulate gradient:  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g}^2$  (square is applied element-wise)

    Compute update:  $\Delta \theta \leftarrow -\frac{\eta}{\sqrt{r}} \mathbf{g}$ . % ( $\frac{1}{\sqrt{r}}$  applied element-wise)

    Apply update:  $\theta \leftarrow \theta + \Delta \theta_t$

**end while**

---

Journal of Machine Learning Research 12 (2011) 2121-2159

Submitted 3/10; Revised 3/11; Published 7/11

## Adaptive Subgradient Methods for Online Learning and Stochastic Optimization\*

**John Duchi**

Computer Science Division  
University of California, Berkeley  
Berkeley, CA 94720 USA

JDUCHI@CS.BERKELEY.EDU

**Elad Hazan**

Technion - Israel Institute of Technology  
Technion City  
Haifa, 32000, Israel

EHAZAN@IE.TECHNION.AC.IL

**Yoram Singer**

Google  
1600 Amphitheatre Parkway  
Mountain View, CA 94043 USA

SINGER@GOOGLE.COM

# RMSprop (Per parameter)

- Similar to Adagrad
- Calculates a **moving average of square of the gradients**
- Accumulate square of gradients (initially  $r = 0$ ):

$$r \leftarrow \rho r + (1 - \rho) \left( \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2$$

- $\rho$  is typically [0.9, 0.99, 0.999]
- Update each parameter/weight based on the gradient on that:

$$\Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W}$$

---

**Algorithm 8.5** The RMSprop algorithm

---

**Require:** Global learning rate  $\eta$ , decay rate  $\rho$ .

**Require:** Initial parameter  $\theta$

Initialize accumulation variables  $r = 0$

while Stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .

    Set  $\mathbf{g} = \mathbf{0}$

**for**  $i = 1$  to  $m$  **do**

        Compute gradient:  $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

**end for**

    Accumulate gradient:  $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g}^2$

    Compute parameter update:  $\Delta \theta = -\frac{\eta}{\sqrt{r}} \odot \mathbf{g}$ . % ( $\frac{1}{\sqrt{r}}$  applied element-wise)

    Apply update:  $\theta \leftarrow \theta + \Delta \theta$

**end while**

---

# RMSprop with Nesterov Momentum

---

**Algorithm 8.6** RMSprop algorithm with Nesterov momentum

---

**Require:** Global learning rate  $\eta$ , decay rate  $\rho$ , momentum coefficient  $\alpha$ .

**Require:** Initial parameter  $\theta$ , initial velocity  $v$ .

Initialize accumulation variable  $r = 0$

**while** Stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .

    Compute interim update:  $\theta \leftarrow \theta + \alpha v$

    Set  $g = 0$

**for**  $i = 1$  to  $m$  **do**

        Compute gradient:  $g \leftarrow g + \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

**end for**

    Accumulate gradient:  $r \leftarrow \rho r + (1 - \rho)g^2$

    Compute velocity update:  $v \leftarrow \alpha v - \frac{\eta}{\sqrt{r}} \odot g$ . % ( $\frac{1}{\sqrt{r}}$  applied element-wise)

    Apply update:  $\theta \leftarrow \theta + v$

**end while**

---

# Adam (per parameter)

- Similar to RMSprop + momentum
- Incorporates first & second order moments
- Bias correction needed to get rid of bias towards zero at initialization

---

**Algorithm 8.7** The Adam algorithm

---

**Require:** Step-size  $\alpha$

**Require:** Decay rates  $\rho_1$  and  $\rho_2$ , constant  $\epsilon$

**Require:** Initial parameter  $\theta$

Initialize 1st and 2nd moment variables  $s = \mathbf{0}$ ,  $r = \mathbf{0}$ ,

Initialize timestep  $t = 0$

**while** Stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .

    Set  $\mathbf{g} = \mathbf{0}$

**for**  $i = 1$  to  $m$  **do**

        Compute gradient:  $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

**end for**

$t \leftarrow t + 1$

    Get biased first moment:  $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

    Get biased second moment:  $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g}^2$

    Compute bias-corrected first moment:  $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

    Compute bias-corrected second moment:  $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

    Compute update:  $\Delta\theta = -\alpha \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \epsilon} \mathbf{g}$  % (operations applied element-wise)

    Apply update:  $\theta \leftarrow \theta + \Delta\theta$

**end while**

# Adadelta (per parameter)

- Incorporates second-order gradient information

---

**Algorithm 8.8** The Adadelta algorithm

---

**Require:** Decay rate  $\rho$ , constant  $\epsilon$

**Require:** Initial parameter  $\theta$

Initialize accumulation variables  $r = \mathbf{0}$ ,  $s = \mathbf{0}$ ,

**while** Stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .

    Set  $\mathbf{g} = \mathbf{0}$

**for**  $i = 1$  to  $m$  **do**

        Compute gradient:  $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

**end for**

    Accumulate gradient:  $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g}^2$

    Compute update:  $\Delta \theta = -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} \mathbf{g}$  % (operations applied element-wise)

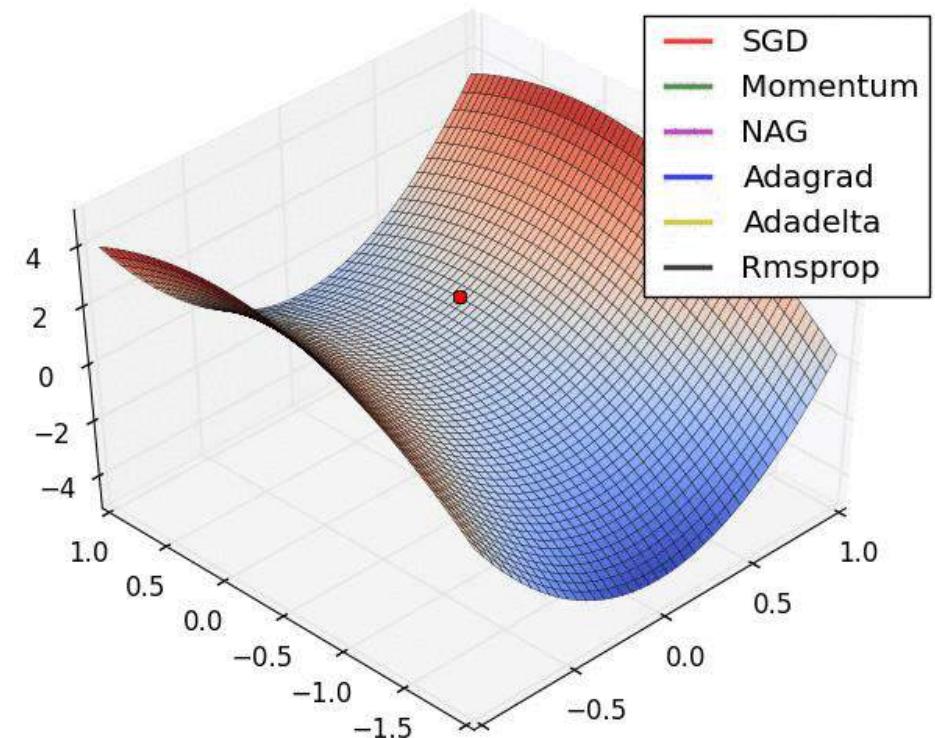
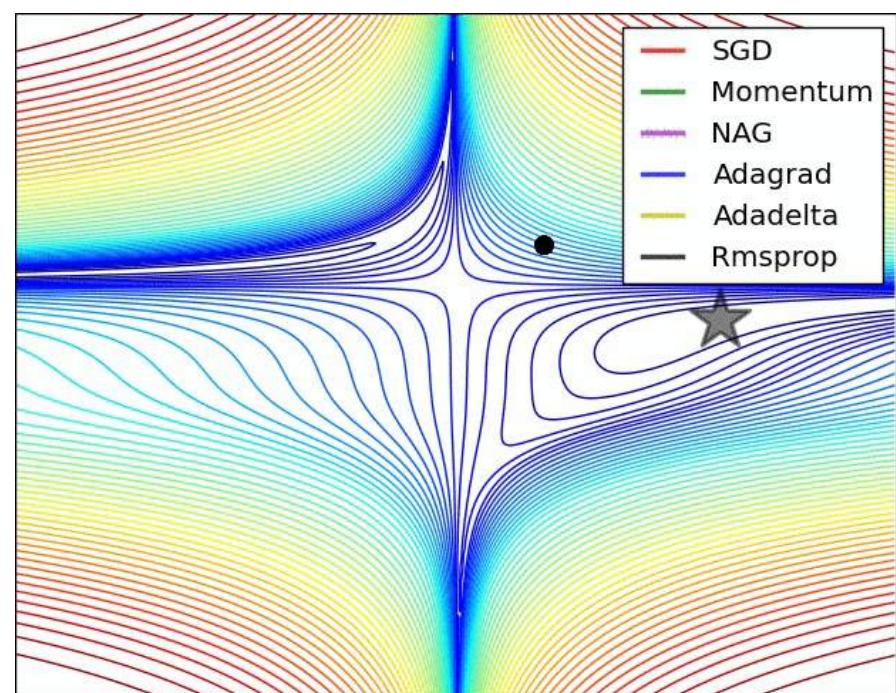
    Accumulate update:  $\mathbf{s} \leftarrow \rho \mathbf{s} + (1 - \rho) [\Delta \theta]^2$

    Apply update:  $\theta \leftarrow \theta + \Delta \theta$

**end while**

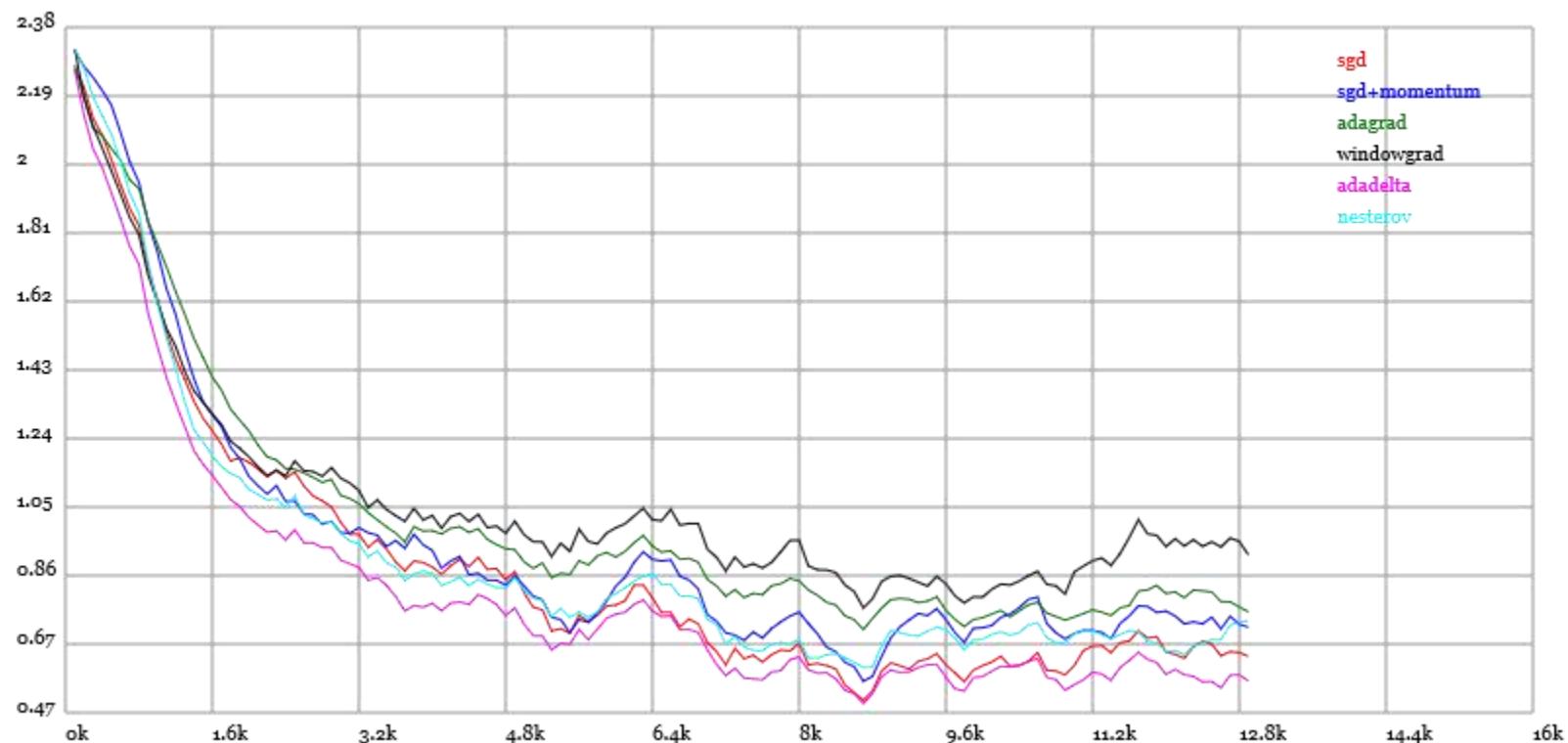
---

# Comparison



NAG: Nesterov's Accelerated Gradient

# Comparison



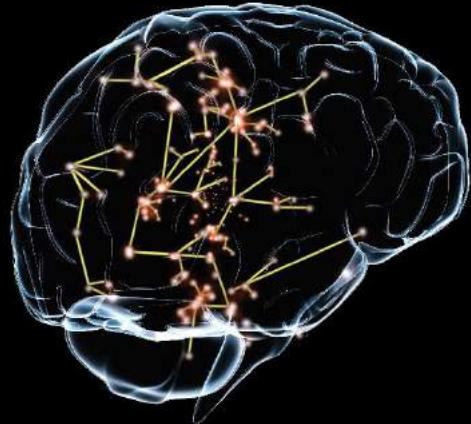
- When SGD+momentum is tuned for hyperparameters, it can outperform Adam etc.
- There are methods that try to finetune the hyper-parameters:

## **YellowFin and the Art of Momentum Tuning**

<https://arxiv.org/abs/1706.03471>

# To sum up

- Different problems seem to favor different per-parameter methods
- Adam seems to perform better among per-parameter adaptive learning rate algorithms
- SGD+Nesterov momentum seems to be a fair alternative



# CENG 783 – Deep Learning

Week 5: Artificial neural networks

# Notes

- HW
- Project proposals
- Opportunity to work in a TUBITAK project

- <http://aiweirdness.com/post/171451900302/do-neural-nets-dream-of-electric-sheep>



Left: A man is holding a dog in his hand  
Right: A woman is holding a dog in her hand  
*Image: @SouperSarah*

# Today

- Representational capacity
- Overfitting, convergence, when to stop
- Data preprocessing and initialization
- Tips and tricks
- Final remarks

# Representational capacity

# Representational capacity

- Boolean functions:
  - Every Boolean function can be represented exactly by a neural network
  - The number of hidden layers might need to grow with the number of inputs
- Continuous functions:
  - Every bounded continuous function can be approximated with small error with two layers
- Arbitrary functions:
  - Three layers can approximate any arbitrary function

Universal approximation theorem:

Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2 (4), 303-314

Kurt Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks, 4(2), 251-257.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.

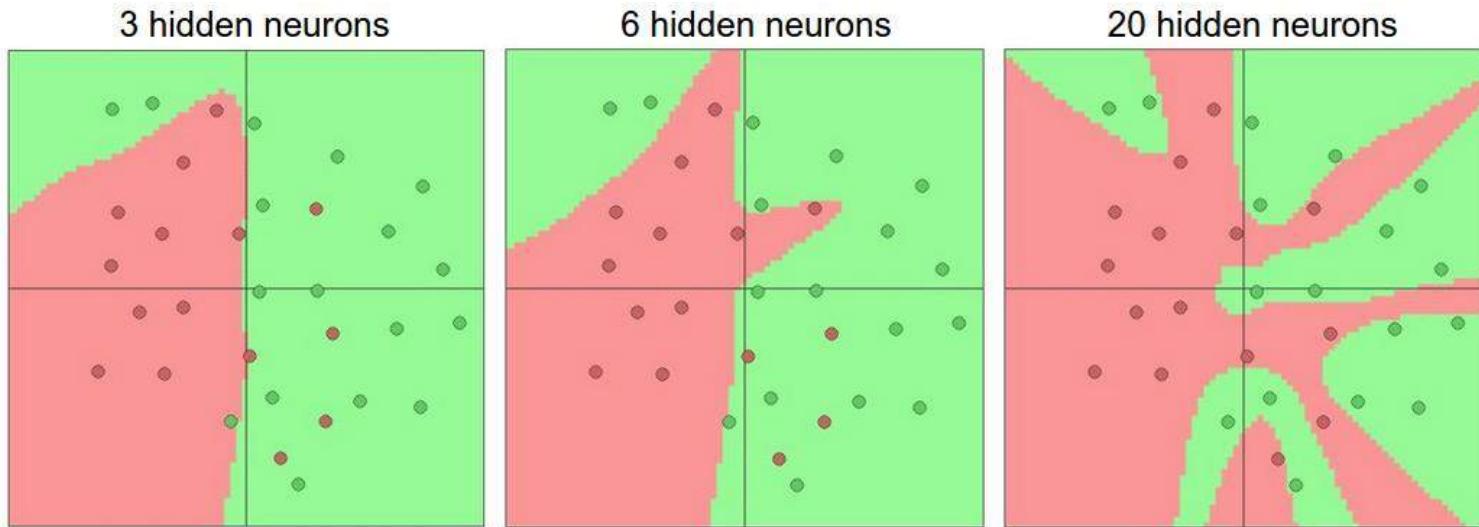
# Representational Capacity: Why go deeper if 3 layers is sufficient?

- Going deeper helps convergence in “big” problems.
- Going deeper in “old-fashion trained” ANNs does not help much in accuracy
  - However, with different training strategies or with Convolutional Networks, going deeper matters

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., & LeCun, Y. (2015, February). The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics* (pp. 192-204).

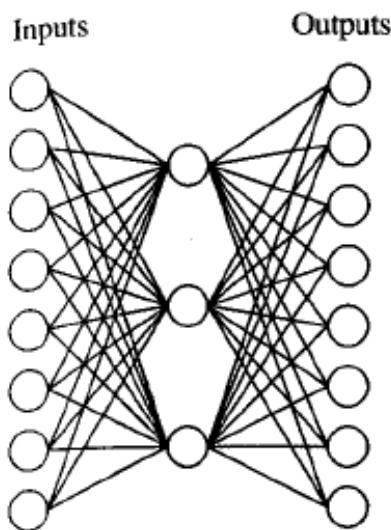
# Representational Capacity

- More hidden neurons → capacity to represent more complex functions



- Problem: overfitting vs. generalization
  - We will discuss the different strategies to help here (L2 regularization, dropout, input noise, using a validation set etc.)

# What the hidden units represent



Input	Hidden Values			Output		
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

**FIGURE 4.7**

Learned Hidden Layer Representation. This  $8 \times 3 \times 8$  network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

# Number of hidden neurons

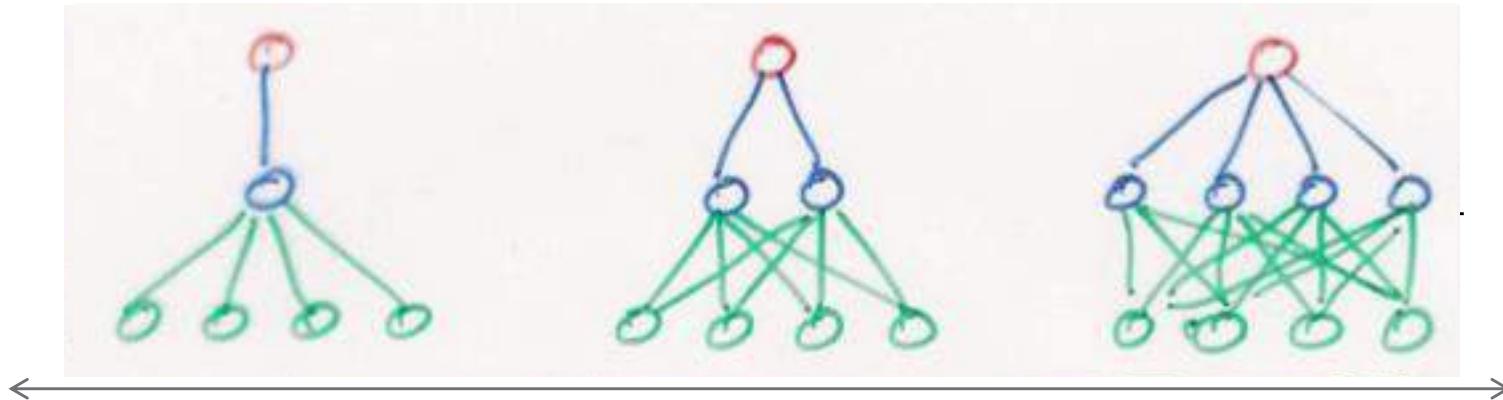
- Several rule of thumbs (Jeff Heaton)
  - The number of hidden neurons should be between the size of the input layer and the size of the output layer.
  - The number of hidden neurons should be  $2/3$  the size of the input layer, plus the size of the output layer.
  - The number of hidden neurons should be less than twice the size of the input layer.

# Number of hidden layers

- Depends on the nature of the problem
  - Linear classification? → No hidden layers needed
  - Non-linear classification?

# Model Complexity

- Models range in their flexibility to fit arbitrary data



**simple model**

**low capacity**

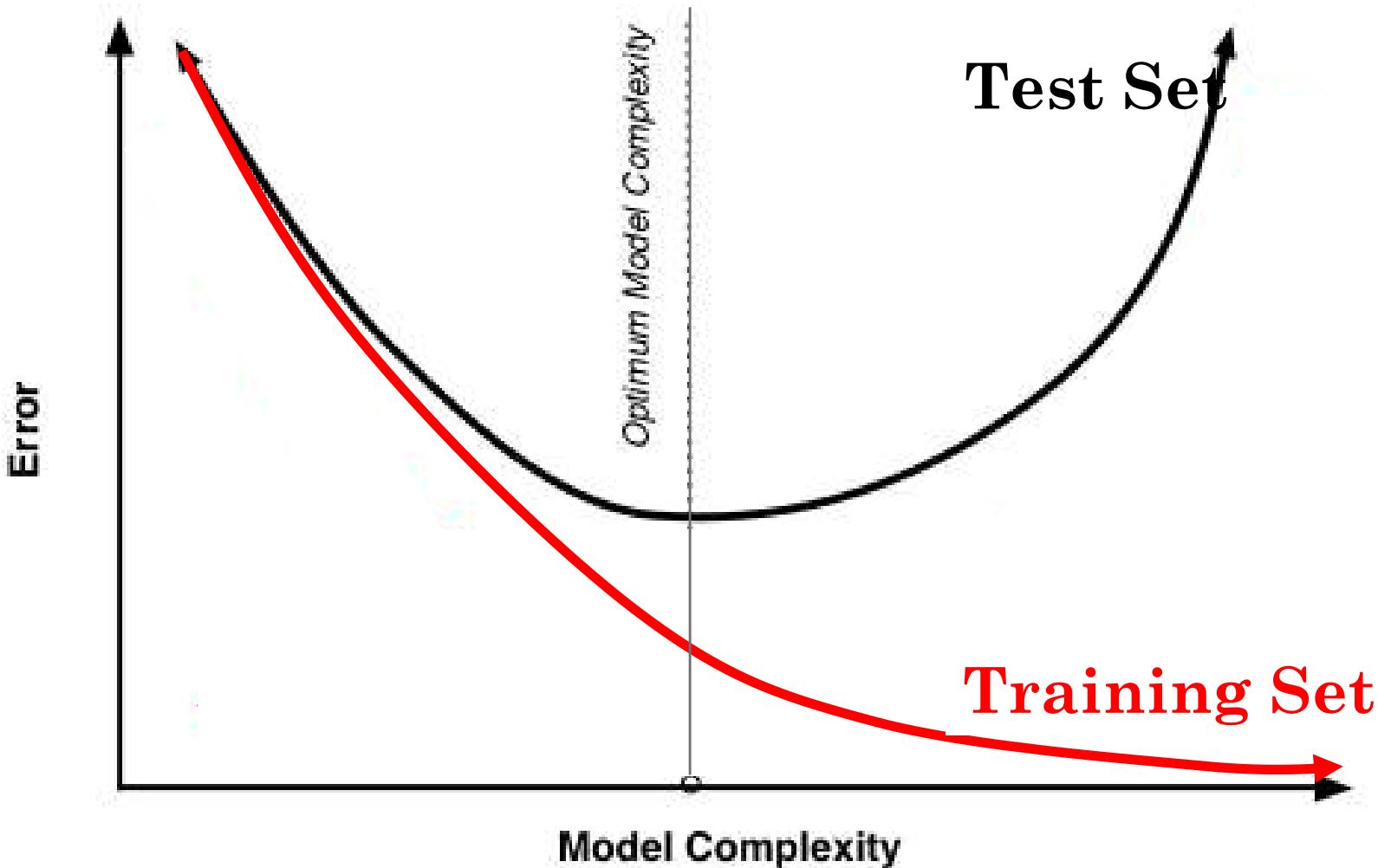
**small capacity may prevent it from representing all structure in data**

**complex model**

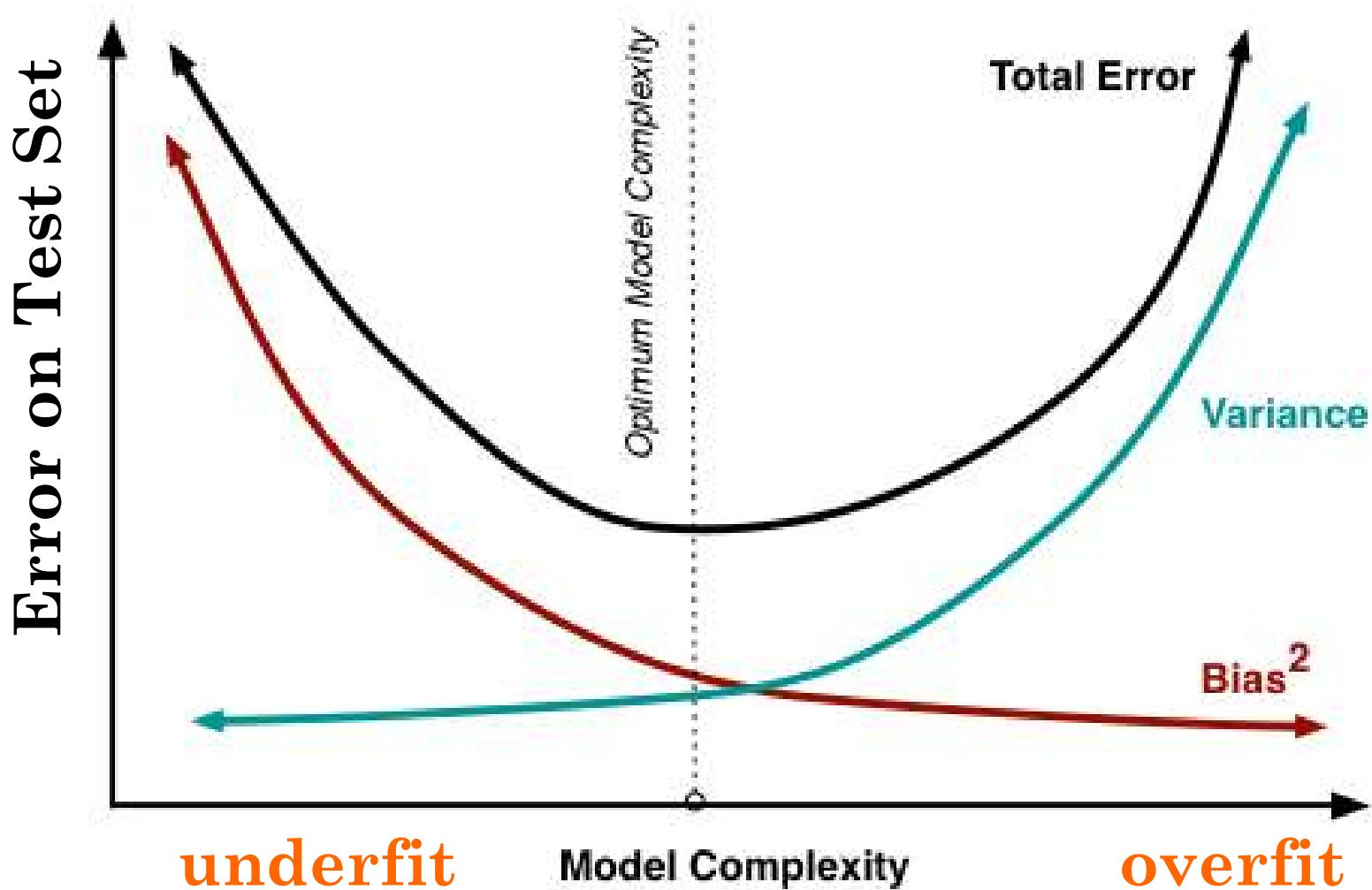
**high capacity**

**large capacity may allow it to memorize data and fail to capture regularities**

# Training Vs. Test Set Error



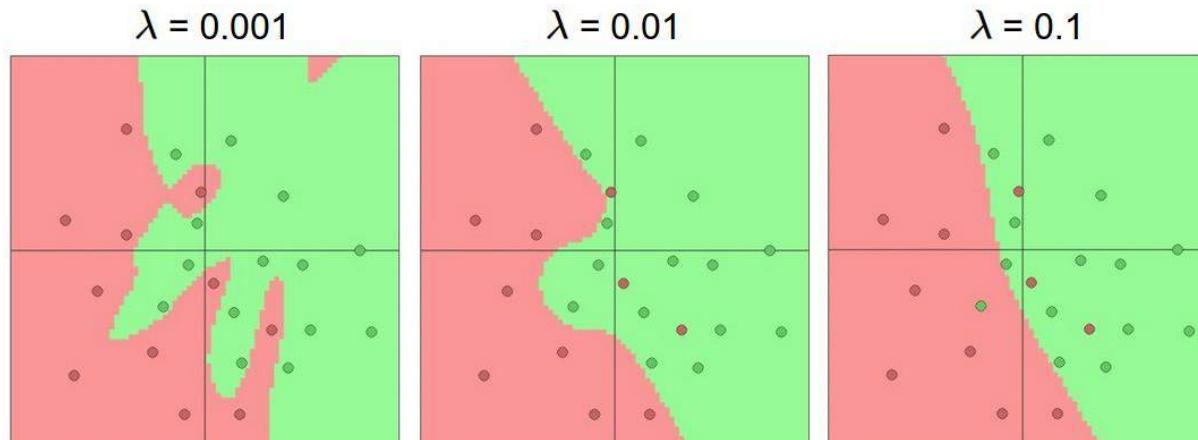
# Bias-Variance Trade Off



# **OVERFITTING, CONVERGENCE, AND WHEN TO STOP**

# Overfitting

- Occurs when training procedure fits not only regularities in training data but also noise.
  - Like memorizing the training examples instead of learning the statistical regularities
- Leads to poor performance on test set
- Most of the practical issues with neural nets involve avoiding overfitting



# Avoiding Overfitting

- Increase training set size
  - Make sure effective size is growing;  
redundancy doesn't help
- Incorporate domain-appropriate bias into model
  - Customize model to your problem
- Set hyperparameters of model
  - number of layers, number of hidden units per layer,  
connectivity, etc.
- **Regularization techniques**
  - “smoothing” to reduce model complexity

# Incorporating Domain-Appropriate Bias Into Model

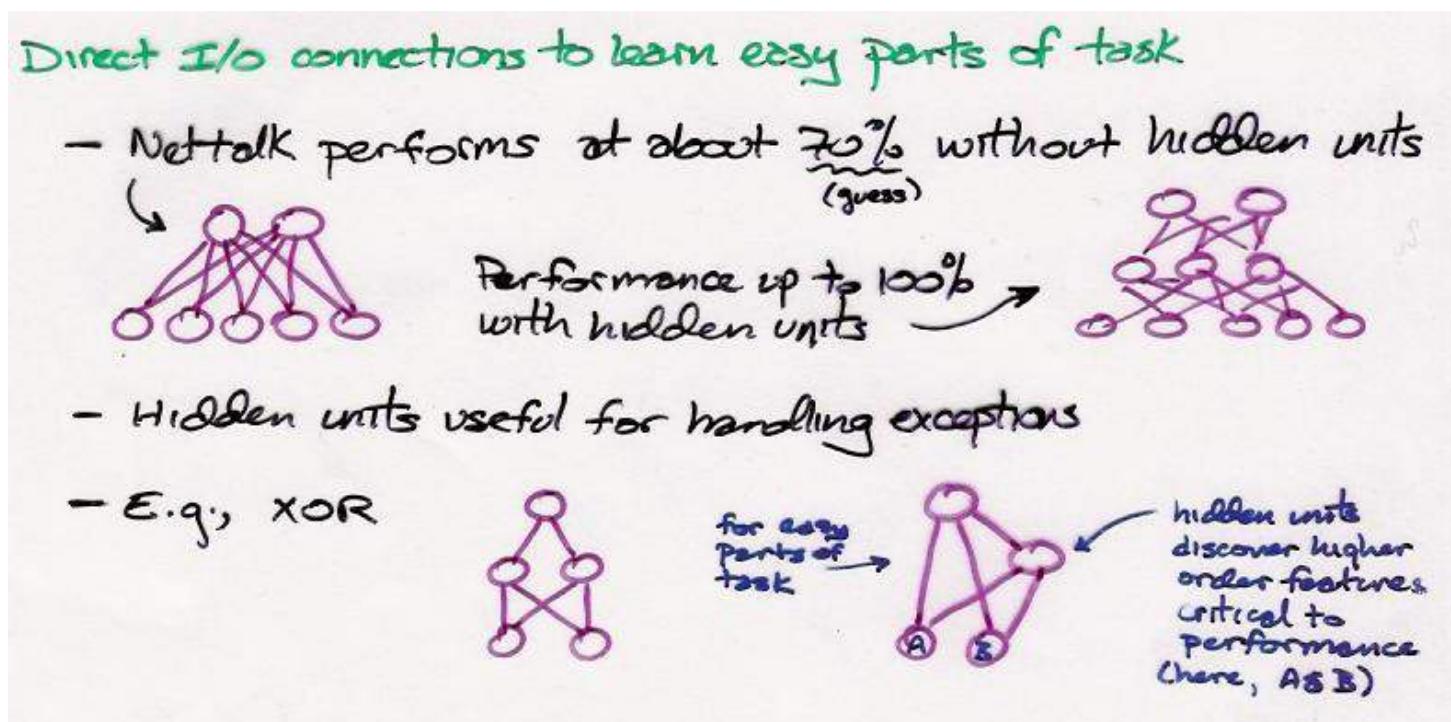
- Input representation
- Output representation
- Architecture
  - # layers, connectivity
  - e.g., convolutional nets, residual connections etc.
- Activation function
- Error function

# Customizing Networks

- Neural nets can be customized based on the problem domain
  - choice of error function
  - choice of activation function
- Domain knowledge can be used to impose domain-appropriate bias on model
  - bias is good if it reflects properties of the data set
  - bias is harmful if it conflicts with properties of data

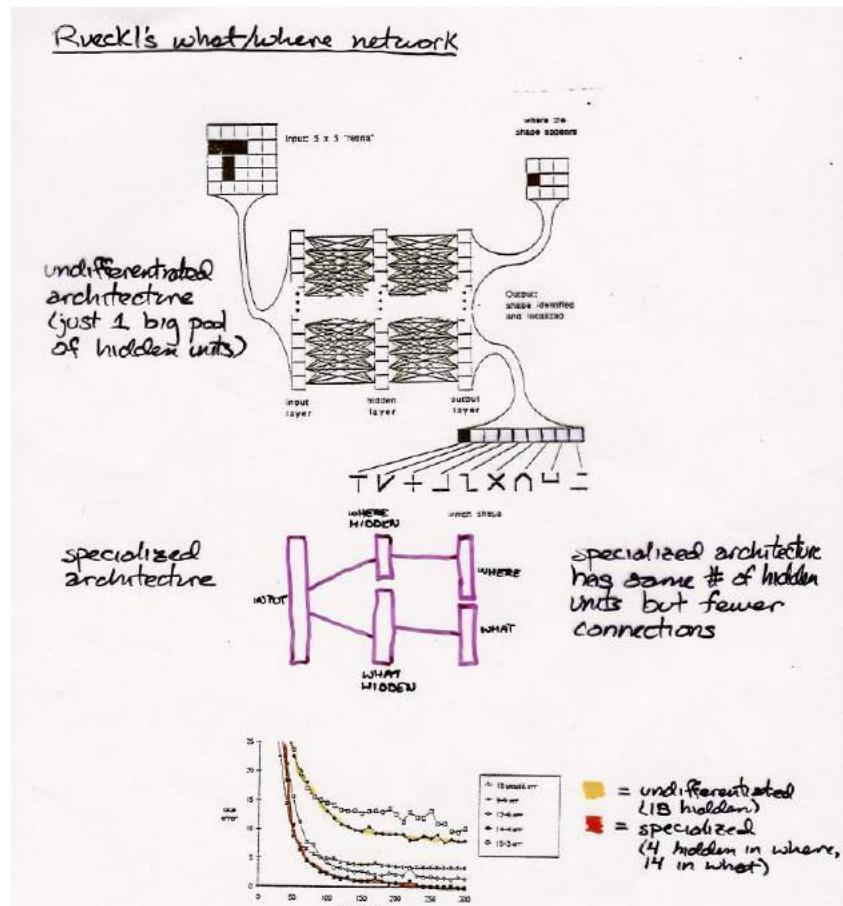
# Adding bias into a model

- Adding hidden layers or direct connections based on the problem



# Adding bias into a model

- Modular architectures
  - Specialized hidden units for special problems



- By splitting hidden units, Rueckl tells system how difficult each task is and that information relevant to one task is not relevant to the other
- Relation to two cortical visual systems (parietal & temporal pathways)

# Adding bias into a model

- Local or specialized receptive fields
  - E.g., in CNNs
- Constraints on activities
- Constraints on weights

## Constraints on activities

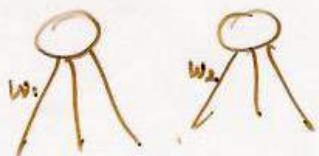
e.g. reduce amount of information flowing through net by encouraging binary-valued hidden units

$$E = \sum_p \sum_i (d_i^p - o_i^p)^2 + \sum_{h \in \text{hidden}} o_h(1-o_h)$$



## Constraints among weights

E.g., T-C problem: Each hidden unit should detect the same feature, but shifted in position



Set  $w_1 = w_2$  initially

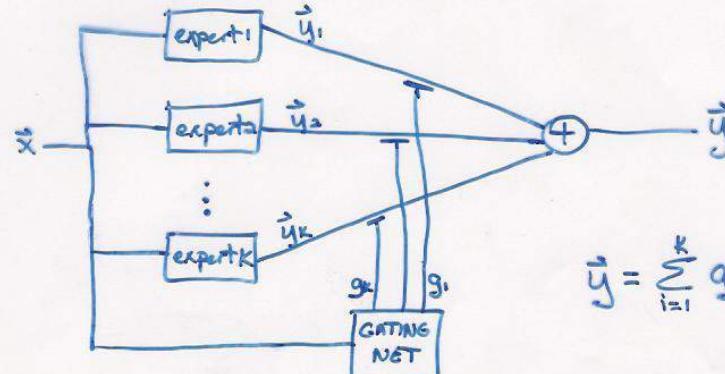
$$\Delta w_1 = \Delta w_2 = -\varepsilon (\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2})$$

# Adding bias into a model

- Use different error functions (e.g., cross-entropy)
- Use specialized activation functions

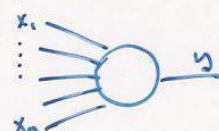
## Specialized Activation Functions

mixture of Experts (Jacobs, Jordan, Nowlan, & Hinton 91)



$$E = -\ln \sum_{i=1}^K g_i e^{-\frac{1}{2}(d-y_i)^2}$$

## Radial Basis Functions



$$\text{net}_i = \sum (x_j - w_{ij})^2 / s_{ij}$$

$$y_i = e^{-\text{net}_i}$$



## Normalized Exponential Transform, a.k.a. Softmax (Bialek 91)

$$\begin{matrix} \hat{y} & 0000 \\ & 11X \\ y & 0000 \\ & XXX \\ h & 0000 \\ & XVVV \end{matrix}$$

$$y_i = \sum w_{ij} h_j \quad (\text{linear})$$

$$\hat{y}_i = \frac{e^{w_i}}{\sum_j e^{w_j}}$$

for classification:  
 $E = -\ln \hat{y}_d$

NOTE:  $0 \leq \hat{y}_i \leq 1$

23

Slide Credit: Michael Mozer

# Adding bias into a model

- Introduce other parameters
  - Temperature
  - Saliency of input

## Designing bias into the net (contd.)

Introduce parameters other than weights/biases and perform gradient descent in these parameters as well.

- E.g., "temperature" (steepleness of sigmoid)



$$O_i = \frac{1}{1 + e^{-\text{net}_i/T_i}}$$

$$\text{Compute } \frac{\partial E}{\partial T_i} \quad \Delta T_i = -\epsilon \frac{\partial E}{\partial T_i}$$

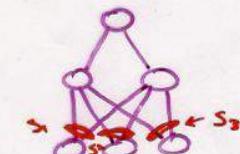
- E.g., input salience term

In the "real world" many inputs are irrelevant to task at hand. Would like to suppress them.

$$\text{net}_i^{\text{layer}_2} = \sum_{j=1}^m w_{ij} O_j S_j$$

↑ activity  
of unit  $j$   
in layer 1

saliency  
unit  $j$  in  
layer 2  
( $\delta-1$ )



$$\text{Compute } \frac{\partial E}{\partial S_j} \quad \Delta S_j = -\epsilon \frac{\partial E}{\partial S_j}$$

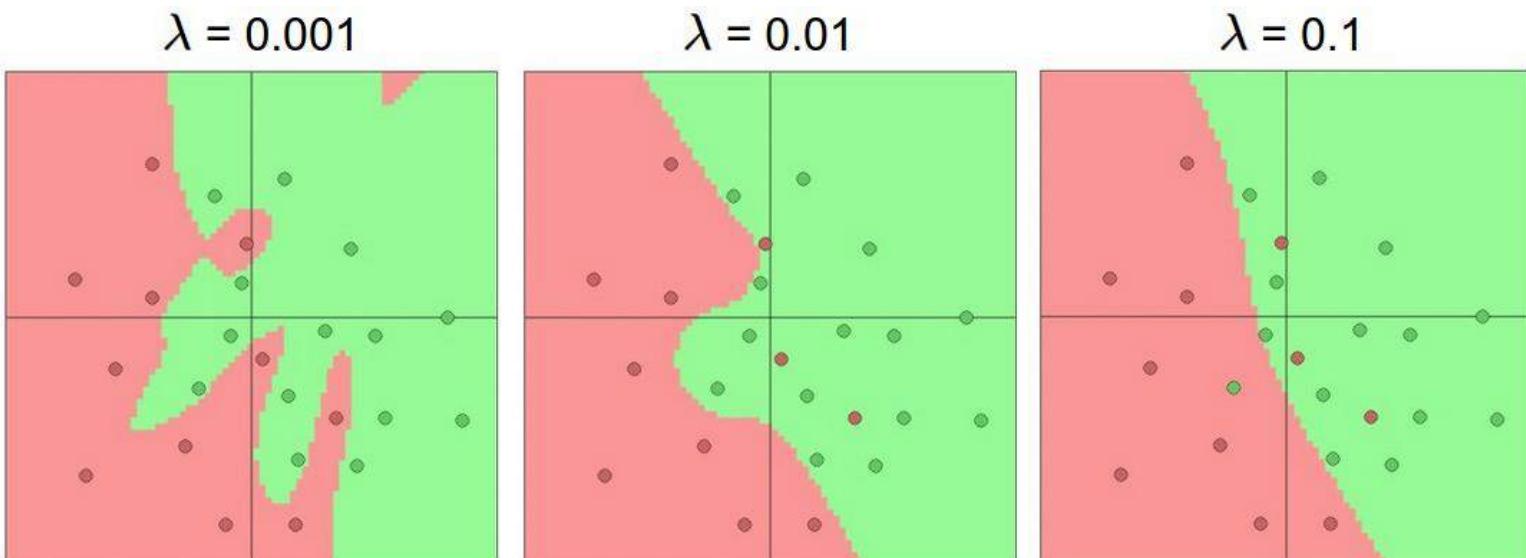
Equivalent to changing all outgoing weights from input unit simultaneously

- These parameters allow you to cut across weight space diagonally ( $\text{low } T \equiv$  turning up all weights coming into unit;  $\text{low } S \equiv$  turning down all weights coming from unit)

# Regularization

- Regularization strength can effect overfitting

$$\frac{1}{2} \lambda w^2$$



# Regularization

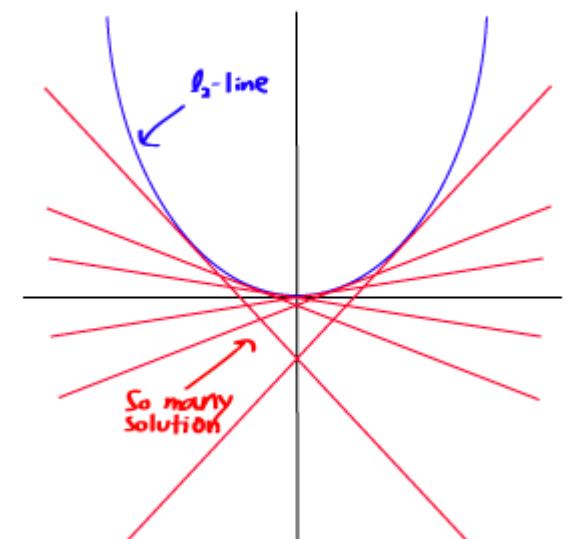
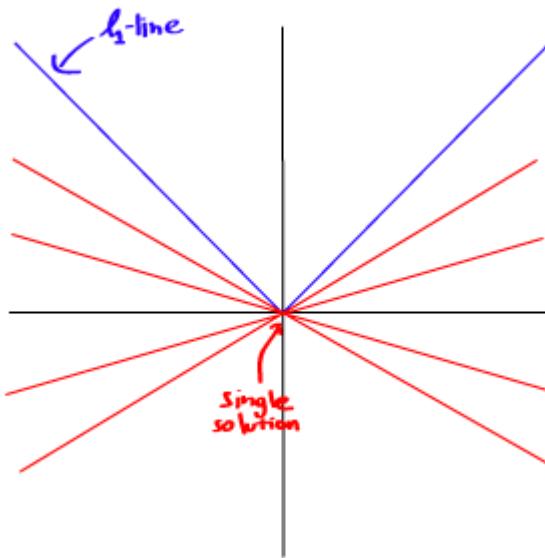
- L2 regularization:  $\frac{1}{2} \lambda w^2$ 
  - Very common
  - Penalizes peaky weight vector, prefers diffuse weight vectors
- L1 regularization:  $\lambda |w|$ 
  - Enforces sparsity (some weights become zero)
  - Leads to input selection (makes it noise robust)
  - Use it if you require sparsity / feature selection
- Can be combined:  $\lambda_1 |w| + \lambda_2 w^2$
- Regularization is not performed on the bias; it seems to make no significant difference

# L1 vs. L2 optimization

$$\min \|x\|_1 \text{ subject to } Ax = b$$

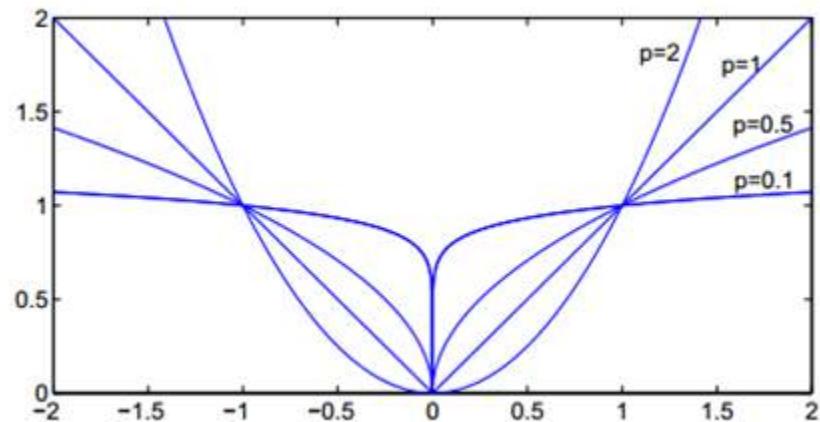
$$\min \|x\|_2 \text{ subject to } Ax = b$$

Harder to solve since it is not smooth. Single L1 solution.



# L0 regularization

- $L_0 = (\sum_i x_i^0)^{1/0}$
- How to compute the zeroth power and zeroth-root?
- Mathematicians approximate this as:
  - $L_0 = \#\{i \mid x_i \neq 0\}$
  - The cardinality of non-zero elements
- This is a strong enforcement of sparsity.
- However, this is non-convex
  - L1 norm is the closest convex form



L<sub>1</sub> penalty: The L<sub>1</sub> penalty is generally used as a substitute for the L<sub>0</sub> penalty, where the L<sub>0</sub> "norm" is just the number of non-zero components of the vector. The L<sub>0</sub> norm can't effectively be used in practice due to the time complexity of the combinatorial problem it creates, but the L<sub>1</sub> norm serves as an adequate substitute. The gradient on the L<sub>1</sub> norm is constant with respect to the magnitude of each vector component, so all inputs are reduced equally. As an example, the L<sub>1</sub> norm of (1, 3) is equal to the L<sub>1</sub> norm of (0, 4). This forces most of the inputs to be extremely close to 0. So **use the L<sub>1</sub> norm when you want sparsity.**

L<sub>2</sub> penalty: The critical difference for the L<sub>2</sub> norm is that the gradient is linear in the magnitude of each component of the vector. Thus, small values are favored, but it's more favorable to decrease a large value than it is to make a small value even smaller because the gradient for small values is very small. As an example, the L<sub>2</sub> norm of (1, 3) is root 10, and decreasing the first component by 1 results in a vector of (0, 3) with L<sub>2</sub> norm of root(9) = 3 but decreasing the second component by 1 results in a vector of (1, 2) with L<sub>2</sub> norm of root(5) < 3. Thus, it's more favorable to decrease the larger components of the vector. Thus, **use the L<sub>2</sub> norm when you don't want large activations.**

Hope this served as an alright answer. There are certainly people here that know more about this than I do (and I'm not overly confident in the completeness of my answer), so I'd love to be corrected or for someone to tell me that I actually do understand the gist of this.

# Regularization

- Enforce an upper bound on weights:
  - Max norm:
    - $\|w\|_2 < c$
    - Helps the gradient explosion problem
    - Improvements reported
- Dropout:
  - At each iteration, *drop* a number of neurons in the network
  - Use a neuron's activation with probability  $p$  (a hyperparameter)
  - Adds stochasticity!

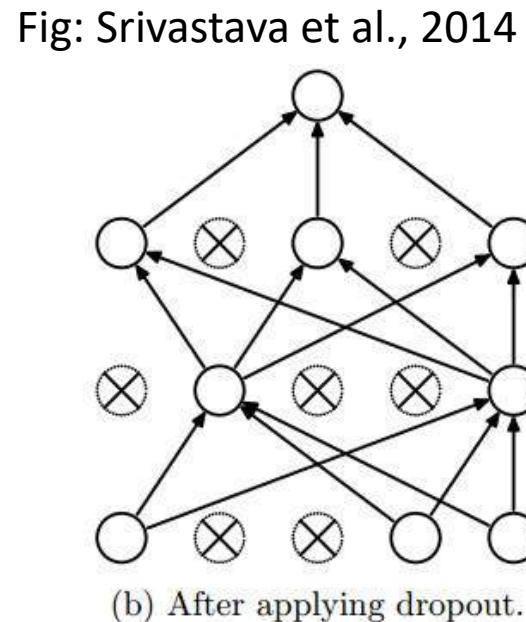
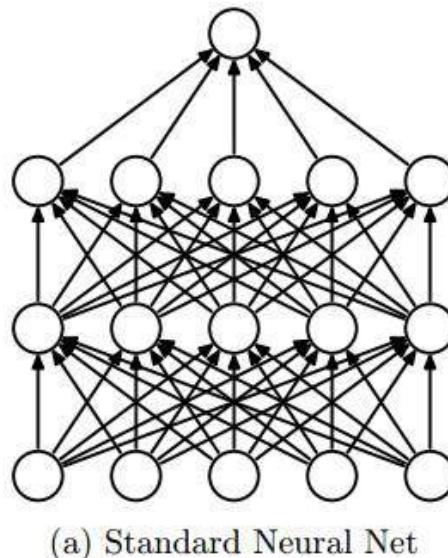


Fig: Srivastava et al., 2014

# Regularization: Dropout

- Feed-forward only on active units
- Can be trained using SGD with mini-batch
  - Back propagate only “active” units.
- One issue:
  - Expected output  $x$  with dropout:
  - $E[x] = px + (1 - p)0$
- To have the same scale at testing time (no dropout), multiply test-time activations with  $p$ .

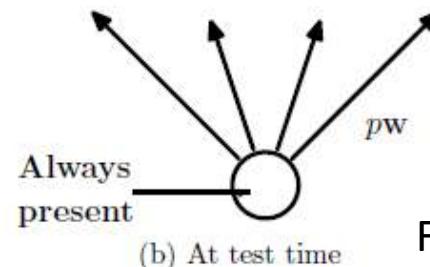
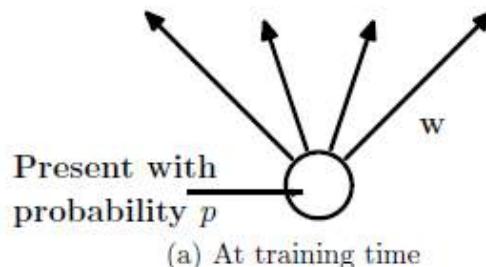


Fig: Srivastava et al.<sup>31</sup> 2014

# Regularization: Dropout

**Training-time:**

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p # first dropout mask
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p # second dropout mask
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

**Test-time:**

```
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
out = np.dot(W3, H2) + b3
```

# Regularization: Inverted Dropout

Perform scaling while dropping at training time!

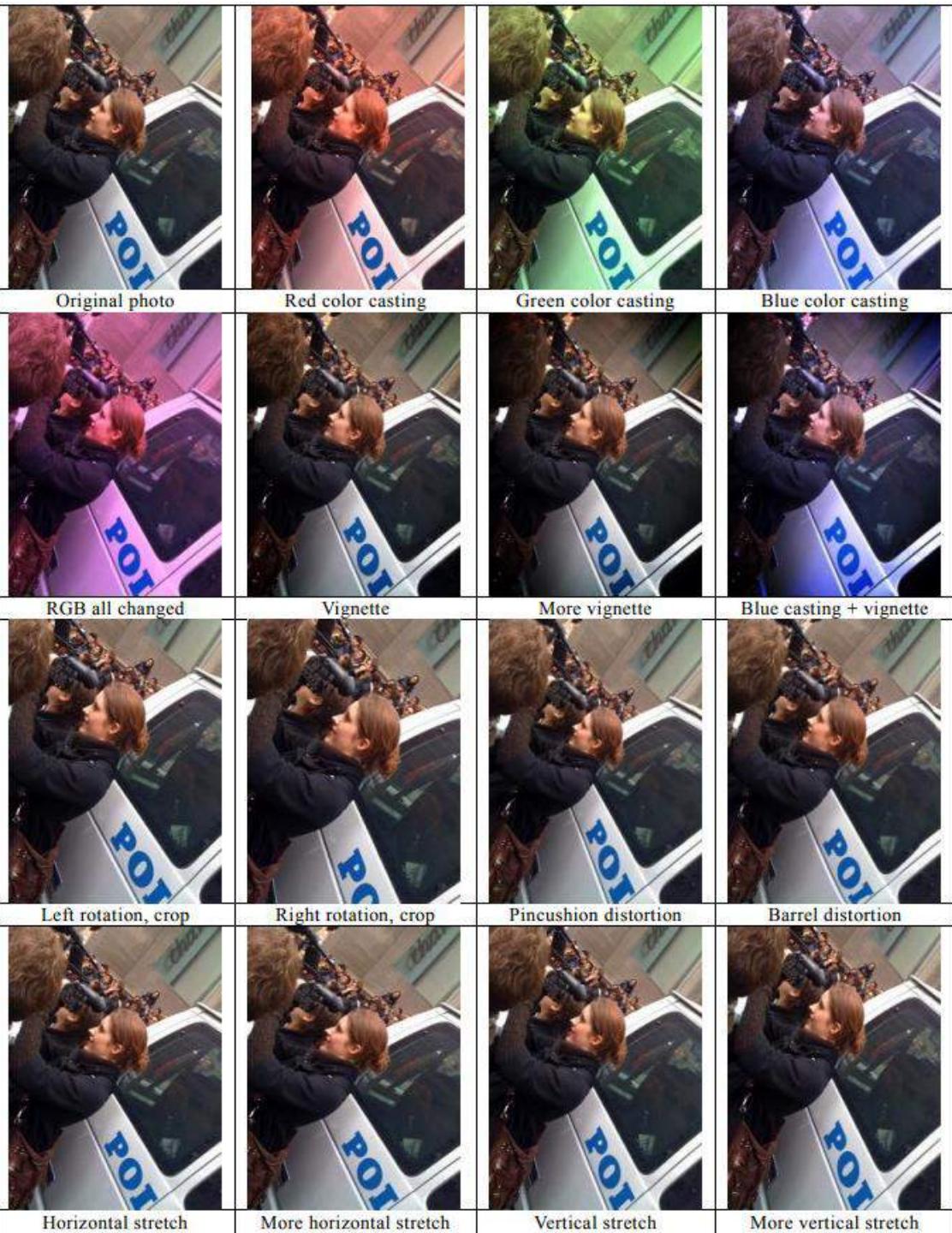
**Training-time:**

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

**Test-time:**

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

# Data Augmentation



# Regularization Summary

- L2 regularization
- Inverted dropout with  $p = 0.5$  (tunable)

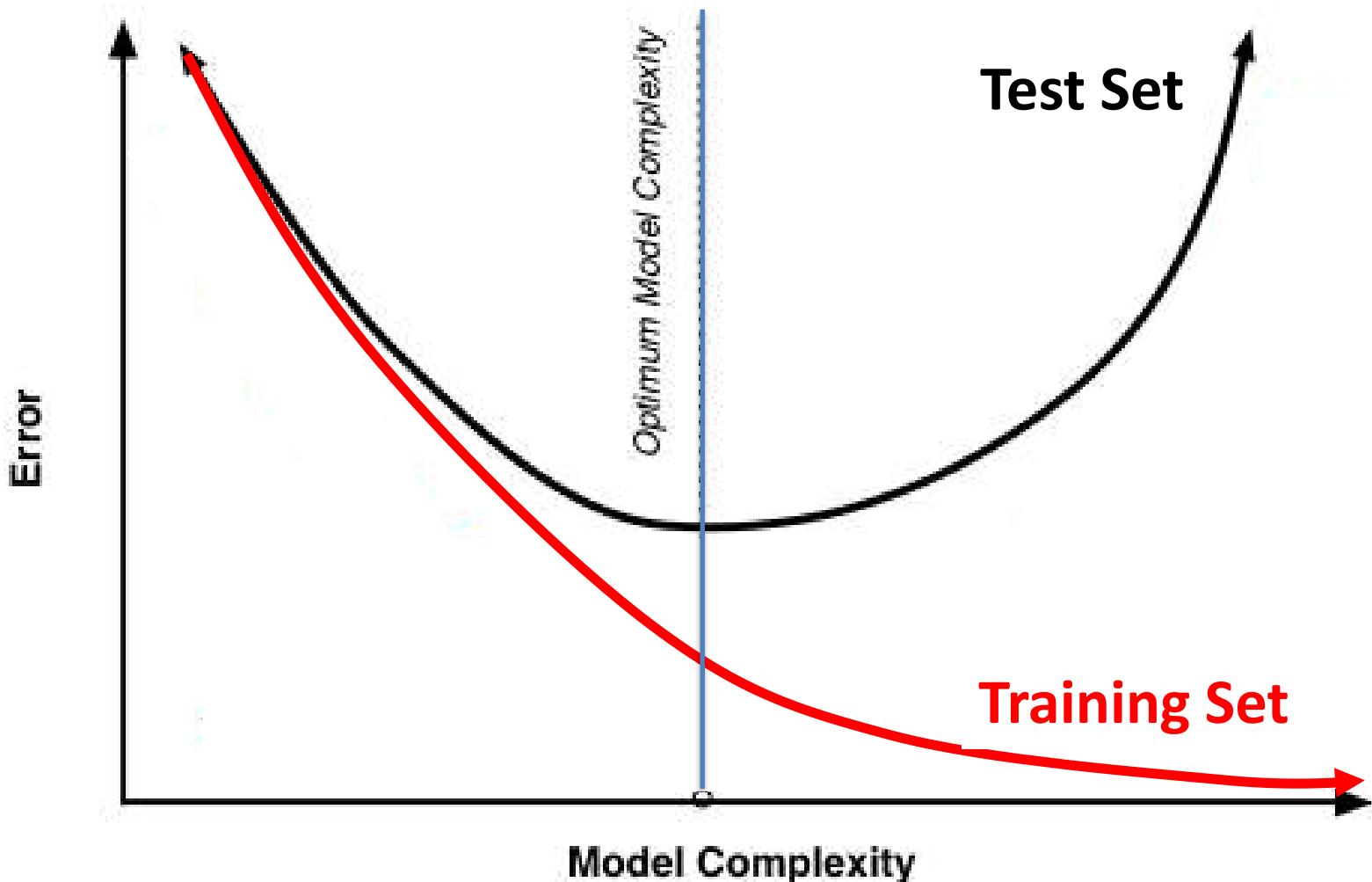
# When To Stop Training

- 1. Train  $n$  epochs; lower learning rate; train  $m$  epochs
  - bad idea: can't assume one-size-fits-all approach
- 2. Error-change criterion
  - stop when error isn't dropping
  - recommendation: criterion based on % drop over a window of, say, 10 epochs
    - 1 epoch is too noisy
    - absolute error criterion is too problem dependent
  - Another idea: train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)

# When To Stop Training

- 3. Weight-change criterion
  - Compare weights at epochs  $(t - 10)$  and  $t$  and test:
$$\max_i |w_i^t - w_i^{t-10}| < q$$
  - Don't base on length of overall weight change vector
  - Possibly express as a percentage of the weight
  - Be cautious: small weight changes at critical points can result in rapid drop in error

# Training Vs. Test Set Error



# **DATA PREPROCESSING AND WEIGHT INITIALIZATION**

# Data Preprocessing

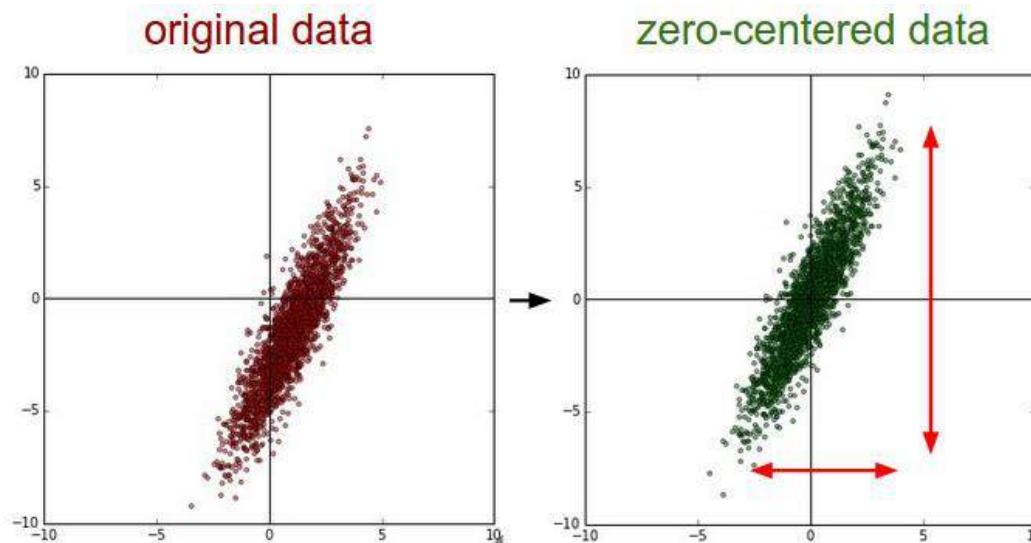
- Mean subtraction
- Normalization
- PCA and whitening

# Data Preprocessing: Mean subtraction

- Subtract the mean for each dimension:

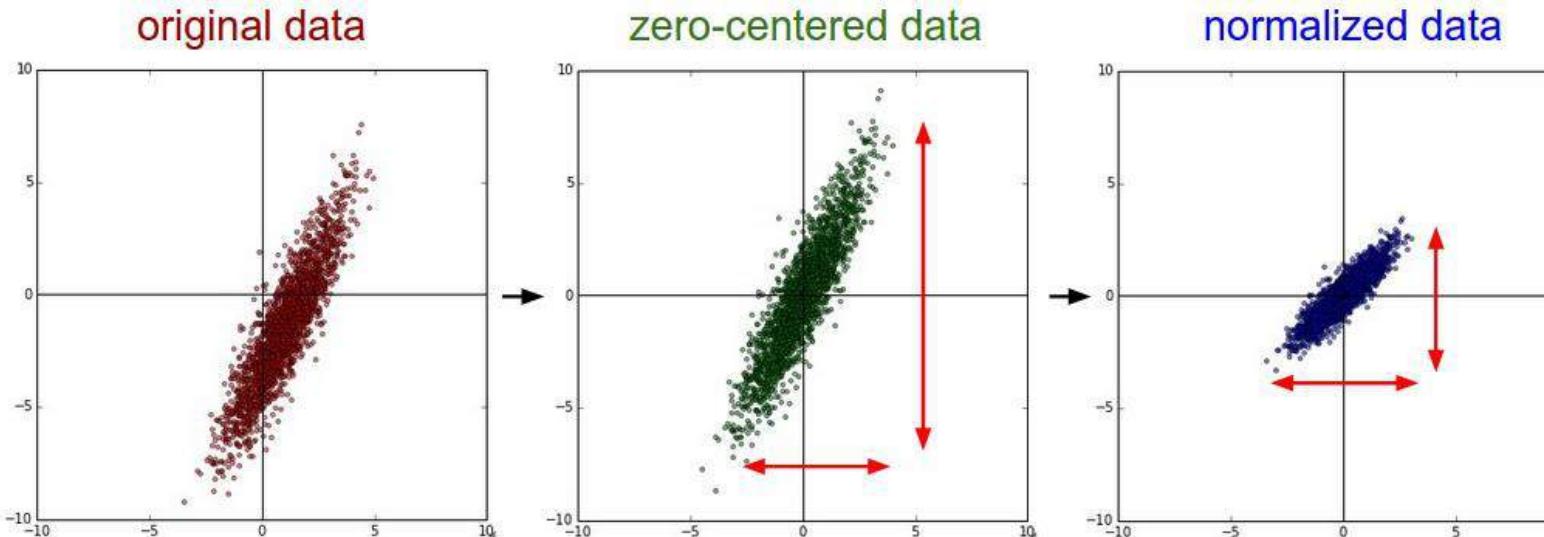
$$x'_i = x_i - \hat{x}_i$$

- Effect: Move the data center (mean) to coordinate center



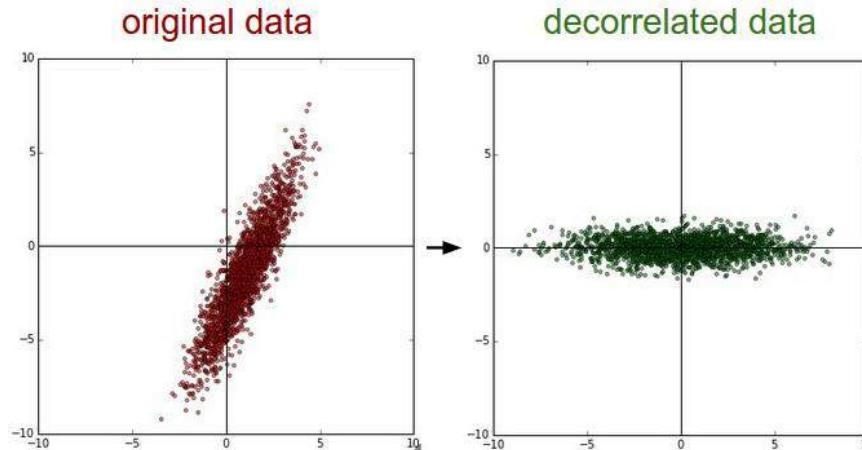
# Data Preprocessing: Normalization (or conditioning)

- Necessary if you believe that your dimensions have different scales
    - Might need to reduce this to give equal importance to each dimension
  - Normalize each dimension by its std. dev. after mean subtraction:
- $$x'_i = x_i - \mu_i$$
- $$x''_i = x'_i / \sigma_i$$
- Effect: Make the dimensions have the same scale



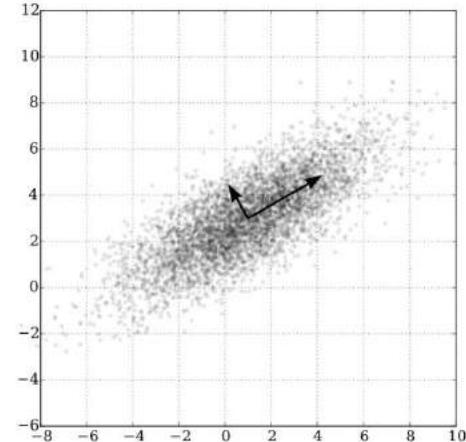
# Data Preprocessing: Principle Component Analysis

- First center the data
- Find the eigenvectors  $e_1, \dots, e_n$
- Project the data onto the eigenvectors:
  - $x_i^R = x_i \cdot [e_1, \dots, e_n]$
- This corresponds to rotating the data to have the eigenvectors as the axes
- If you take the first  $M$  eigenvectors, it corresponds to dimensionality reduction



# Reminder: PCA

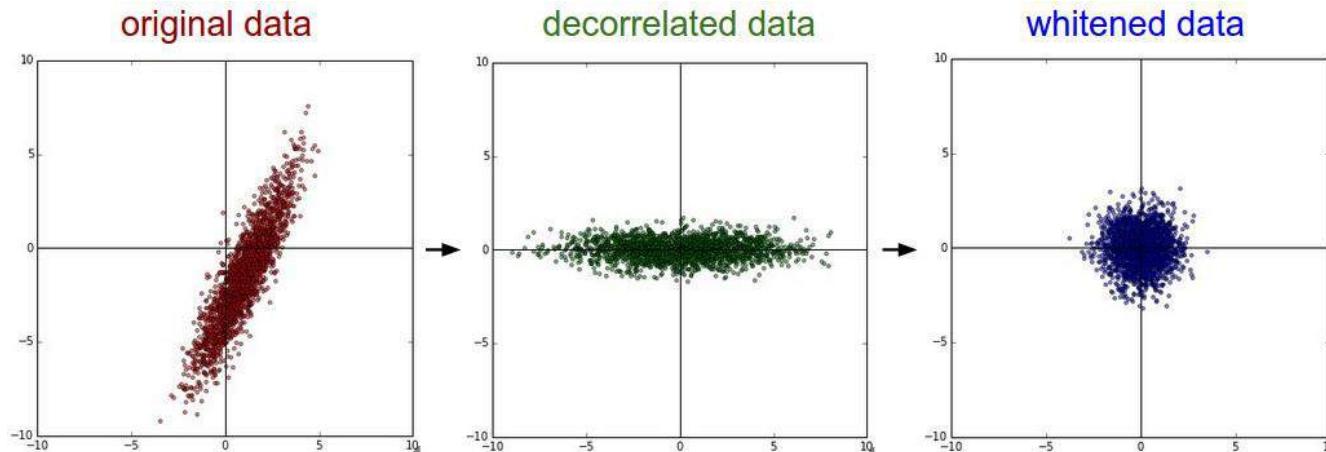
- Principle axes are the eigenvectors of the covariance matrix:



$$\Sigma = \begin{bmatrix} E[(X_1 - \mu_1)(X_1 - \mu_1)] & E[(X_1 - \mu_1)(X_2 - \mu_2)] & \cdots & E[(X_1 - \mu_1)(X_n - \mu_n)] \\ E[(X_2 - \mu_2)(X_1 - \mu_1)] & E[(X_2 - \mu_2)(X_2 - \mu_2)] & \cdots & E[(X_2 - \mu_2)(X_n - \mu_n)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(X_n - \mu_n)(X_1 - \mu_1)] & E[(X_n - \mu_n)(X_2 - \mu_2)] & \cdots & E[(X_n - \mu_n)(X_n - \mu_n)] \end{bmatrix}.$$

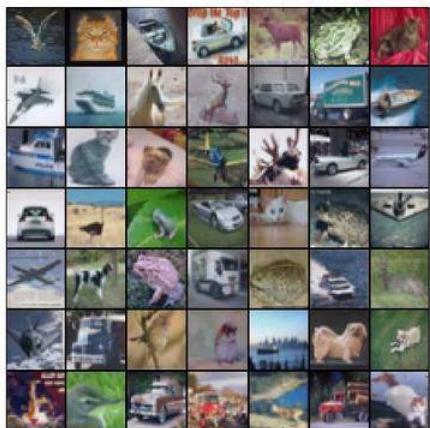
# Data Preprocessing: Whitening

- Normalize the scale with the norm of the eigenvalue:  
$$x_i^w = x_i^R / ([\lambda_1, \dots, \lambda_n] + \epsilon)$$
- $\epsilon$ : a very small number to avoid division by zero
- This stretches each dimension to have the same scale.
- Side effect: this may exaggerate noise.

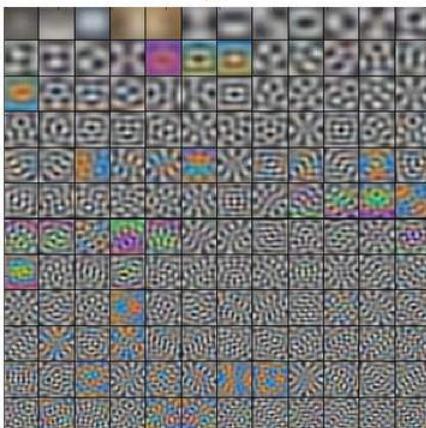


# Data Preprocessing: Example

original images



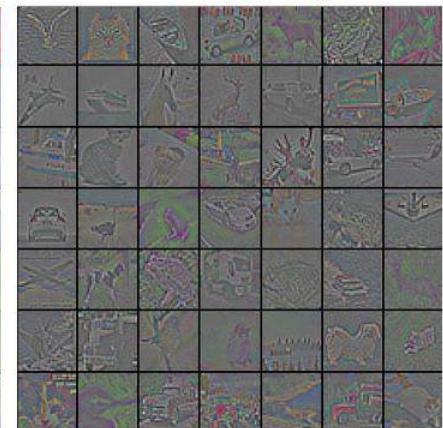
top 144 eigenvectors



reduced images



whitened images



# Data Preprocessing: Summary

- We mostly don't use PCA or whitening
  - They are computationally very expensive
  - Whitening has side effects
- It is quite crucial and common to zero-center the data
- Most of the time, we see normalization with the std. deviation

# Weight Initialization

- Zero weights
  - Wrong!
  - Leads to updating weights by the same amounts for every input
  - Symmetry!
- Initialize the weights randomly to small values:
  - Sample from a small range, e.g.,  $\text{Normal}(0,0.01)$
  - Don't initialize too small
- The bias may be initialized to zero
  - For ReLU units, this may be a small number like 0.01.

**Note: None of these provide guarantees. Moreover, there is no guarantee that one of these will always be better.**

# Initial Weight Normalization

- Problem:
  - Variance of the output changes with the number of inputs
  - If  $s = \sum_i w_i x_i$ :

$$\begin{aligned}
 \text{Var}(s) &= \text{Var}\left(\sum_i^n w_i x_i\right) \\
 &= \sum_i^n \text{Var}(w_i x_i) \\
 &= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \\
 &= \sum_i^n \text{Var}(x_i) \text{Var}(w_i) \\
 &= (n \text{Var}(w)) \text{Var}(x)
 \end{aligned}$$

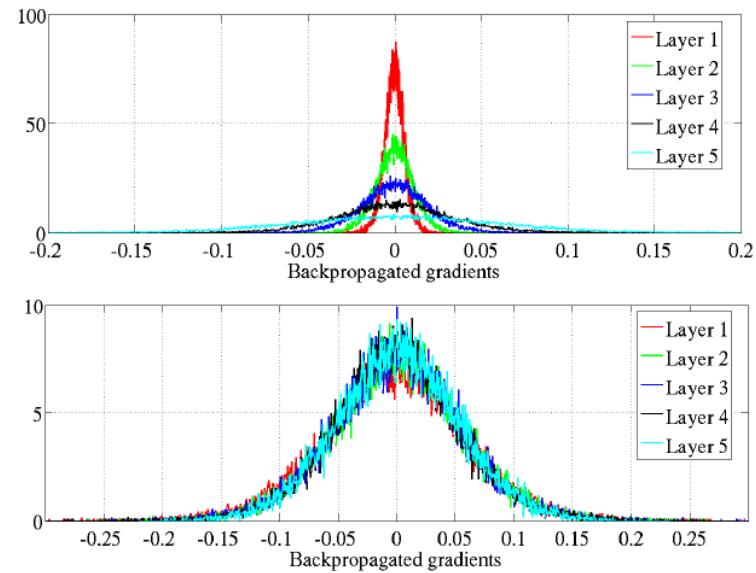


Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

Glorot & Bengio, “Understanding the difficulty of training deep feedforward neural networks”, 2010.

$$\text{Var}(X) = E[(X - \mu)^2]$$

# Initial Weight Normalization

- **Solution:**

- Get rid of  $n$  in

$$Var(s) = (n \ Var(w))Var(x)$$

- How?

- $w_i = rand(0,1)/\sqrt{n}$

- Why?

- $Var(aX) = a^2Var(X)$

- If the number of inputs & outputs are not fixed:

- $w_i = rand(0,1) \times \frac{2}{\sqrt{n_{in}+n_{out}}}$

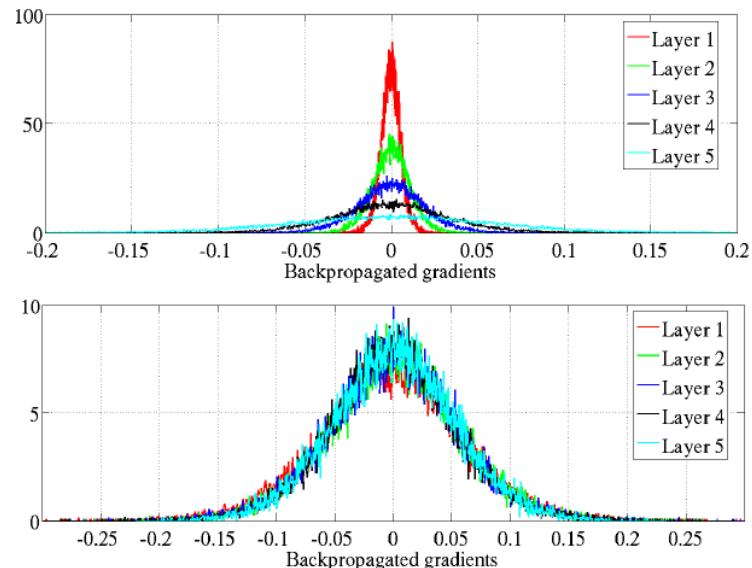


Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

Glorot & Bengio, "Understanding the difficulty of training deep feedforward neural networks", 2010.

# Alternative: Batch Normalization

- Normalization is differentiable
  - So, make it part of the model (not only at the beginning)
  - I.e., perform normalization during every step of processing
- More robust to initialization
- Issue: How to normalize at test time?
  1. Store means and variances during training, or
  2. Calculate mean & variance over your test data

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;  
Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

Ioffe & Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", 2015.

# To sum up

- Initialization and normalization are crucial
- Different initialization & normalization strategies may be needed for different deep learning methods
  - E.g., in CNNs, normalization might be performed only on convolution etc.
- More on this later

# **LOSS FUNCTIONS, AGAIN**

# Loss functions

- A **single correct** label case (classification):
  - Hinge loss:
    - $L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$
  - Soft-max:
    - $L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right)$

# Loss functions

- **Many correct labels case:**
  - Binary prediction for each label, independently:
    - $L_i = \sum_j \max(0, 1 - y_{ij}f_j)$
    - $y_{ij} = +1$  if example  $i$  is labeled with label  $j$ ; otherwise  $y_{ij} = -1$ .
  - Alternatively, train logistic loss for each label (0 or 1):

$$L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))$$

# Loss functions

- Regression case (a continuous label):
  - L2 Norm squared (L2 Loss):

- $L_i = \|f - y_i\|_2^2$

- $\frac{\partial L_i}{\partial f_j} = f_j - (y_i)_j$

- L1 Norm:

- $L_i = \|f - y_i\|_1 = \sum_j |f_j - (y_i)_j|$

- $\frac{\partial L_i}{\partial f_j} = sign(f_j - (y_i)_j)$

Reminder:

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}.$$

# Mean-squared error Loss: Caution

- MSE loss asks for a more difficult constraint:
  - Learn to output a response that is exactly the same as the correct label
  - This is harder to train
- Compare, e.g., softmax:
  - Which asks only one response to be maximum than others.

# Loss functions

- What if we want to predict a graph, tree etc.? Something that has structure.
  - **Structured loss**: formulate loss such that you minimize the distance to a correct structure
  - Not very common

# **ISSUES & PRACTICAL ADVICES**

# Issues & tricks

- Vanishing gradient
  - Saturated units block gradient propagation (why?)
  - A problem especially present in recurrent networks or networks with a lot of layers
- Overfitting
  - Drop-out, regularization and other tricks.
- Tricks:
  - Unsupervised pretraining
- Batch normalization (each unit's preactivation is normalized)
  - Helps keeping the preactivation non-saturated
  - Do this for mini-batches (adds stochasticity)
  - Backprop needs to be updated

# Unsupervised pretraining

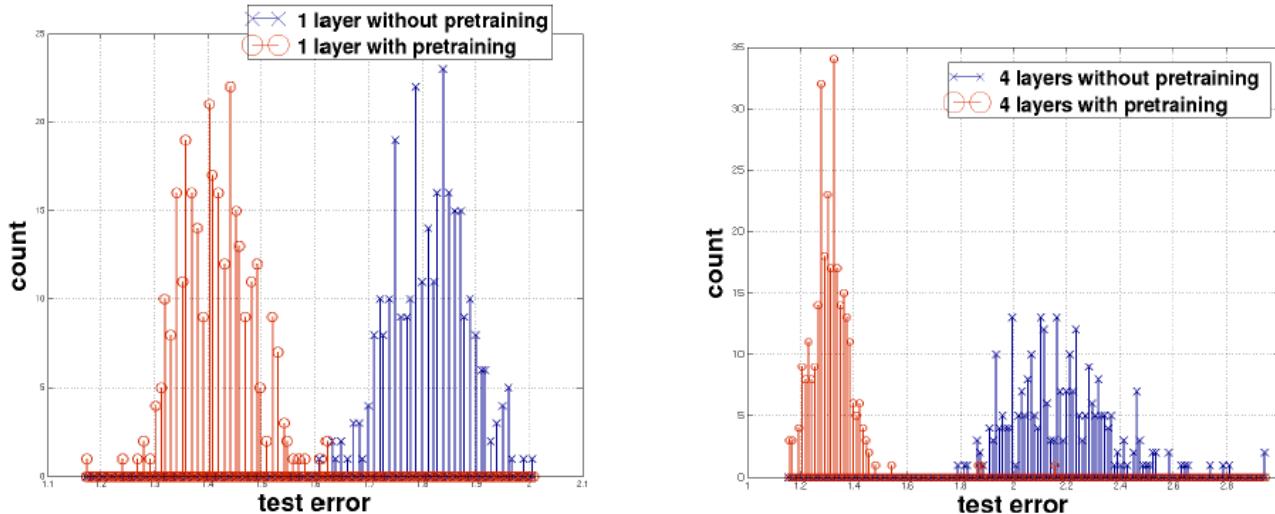


Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left:** 1 hidden layer. **Right:** 4 hidden layers.

Journal of Machine Learning Research 11 (2010) 620-660

Submitted 8/09; Published 2/10

## Why Does Unsupervised Pre-training Help Deep Learning?

Dumitru Erhan\*  
Yoshua Bengio  
Aaron Courville  
Pierre-Antoine Manzagol  
Pascal Vincent

Département d'informatique et de recherche opérationnelle  
Université de Montréal  
2920, chemin de la Tour  
Montréal, Québec, H3T 1J8, Canada

DUMITRU.ERHAN@UMONTREAL.CA  
YOSHUA.BENGIO@UMONTREAL.CA  
AARON.COURVILLE@UMONTREAL.CA  
PIERRE-ANTOINE.MANZAGOL@UMONTREAL.CA  
PASCAL.VINCENT@UMONTREAL.CA

Samy Bengio  
Google Research  
1600 Amphitheatre Parkway  
Mountain View, CA, 94043, USA

BENGIO@GOOGLE.COM

# Unsupervised pretraining

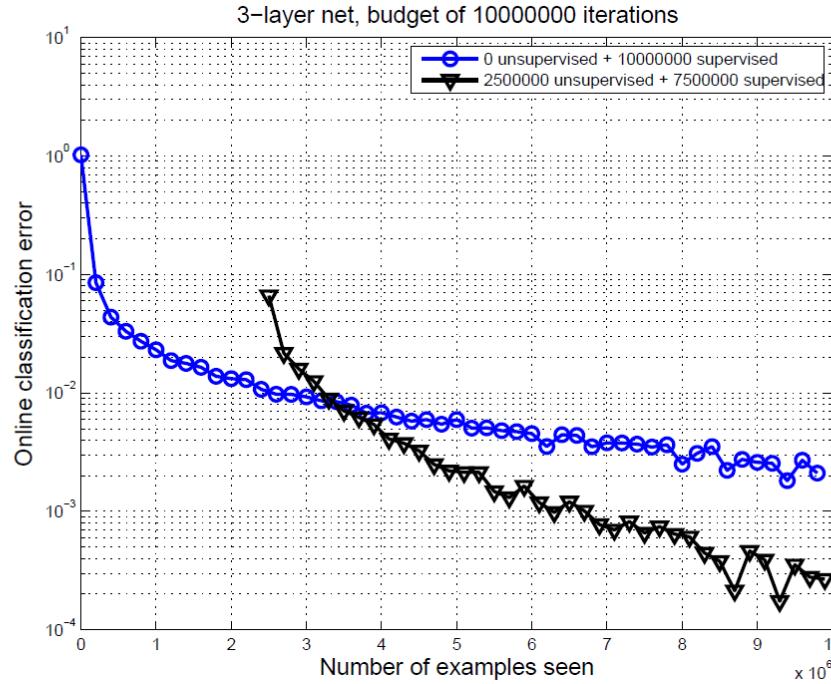


Figure 7: Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the beginning. The first 2.5 million examples are used for unsupervised pre-training (of a stack of denoising auto-encoders). The oscillations near the end are because the error rate is too close to zero, making the sampling variations appear large on the log-scale. Whereas with a very large training set regularization effects should dissipate, one can see that without pre-training, training converges to a poorer apparent local minimum: unsupervised pre-training helps to find a better minimum of the online error. Experiments performed by Dumitru Erhan.

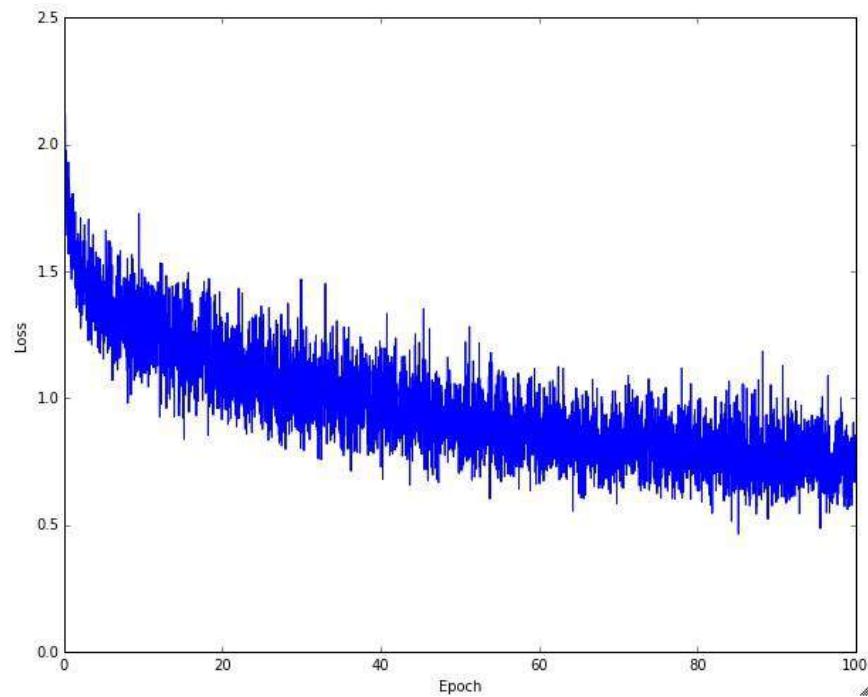
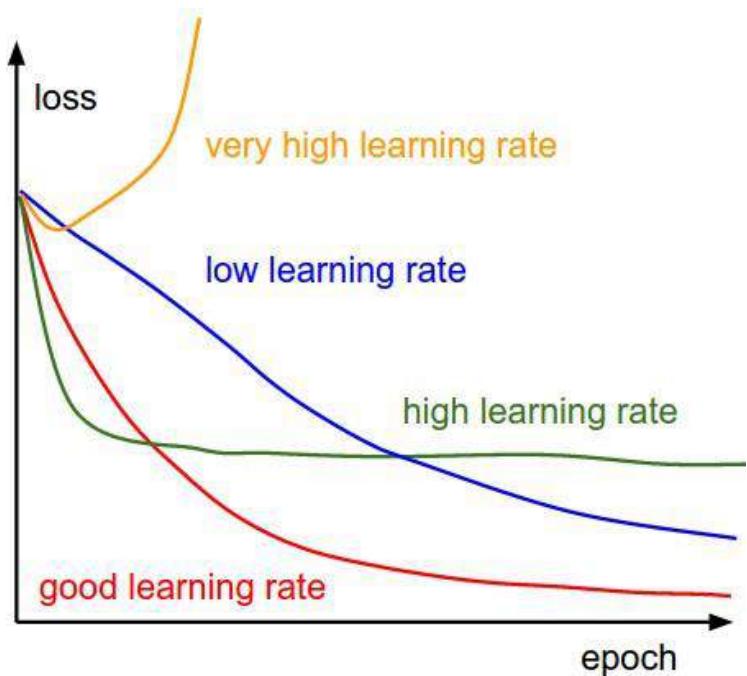
# What if things are not working?

- Check your gradients by comparing them against numerical gradients
  - More on this at: <http://cs231n.github.io/neural-networks-3/>
  - Check whether you are using an appropriate floating point representation
    - Be aware of floating point precision/loss problems
  - Turn off drop-out and other “extra” mechanisms during gradient check
  - This can be performed only on a few dimensions
- Regularization loss may dominate the data loss
  - First disable regularization loss & make sure data loss works
  - Then add regularization loss with a big factor
  - And check the gradient in each case

# What if things are not working?

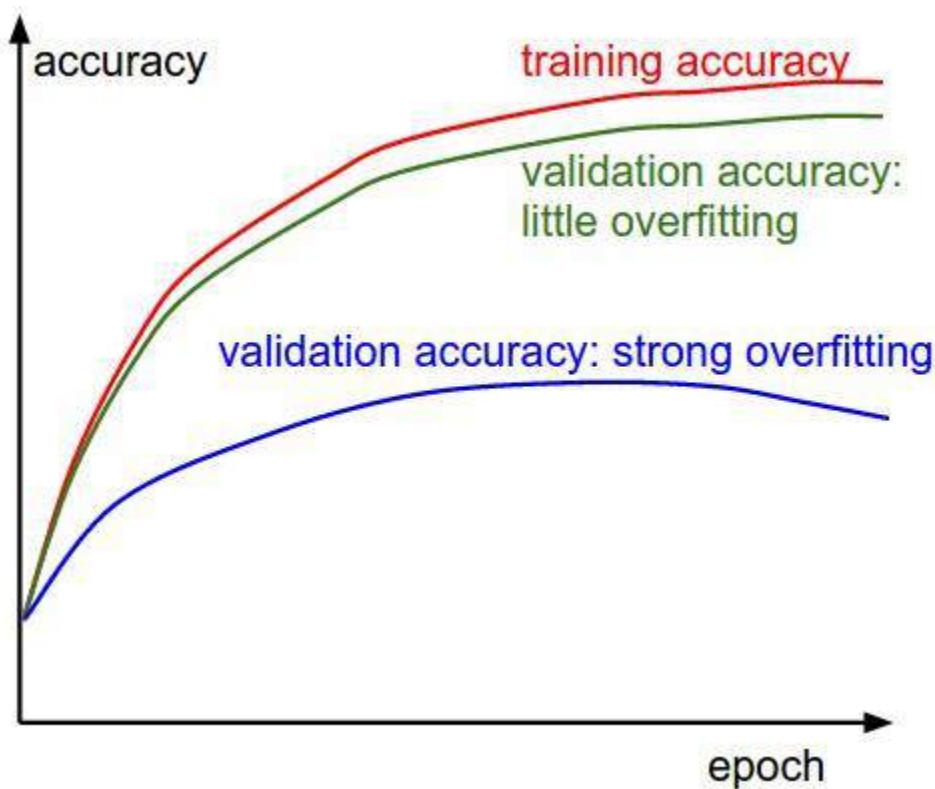
- Have a feeling of the initial loss value
  - For CIFAR-10 with 10 classes: because each class has probability of 0.1, initial loss is  $-\ln(0.1)=2.302$
  - For hinge loss: since all margins are violated (since all scores are approximately zero), loss should be around 9 (+1 for each margin).
- Try to overfit on a tiny subset of the dataset
  - The cost should reach to zero if things are working properly

# What if things are not working?



Learning rate might be too low;  
Batch size might be too small

# What if things are not working?

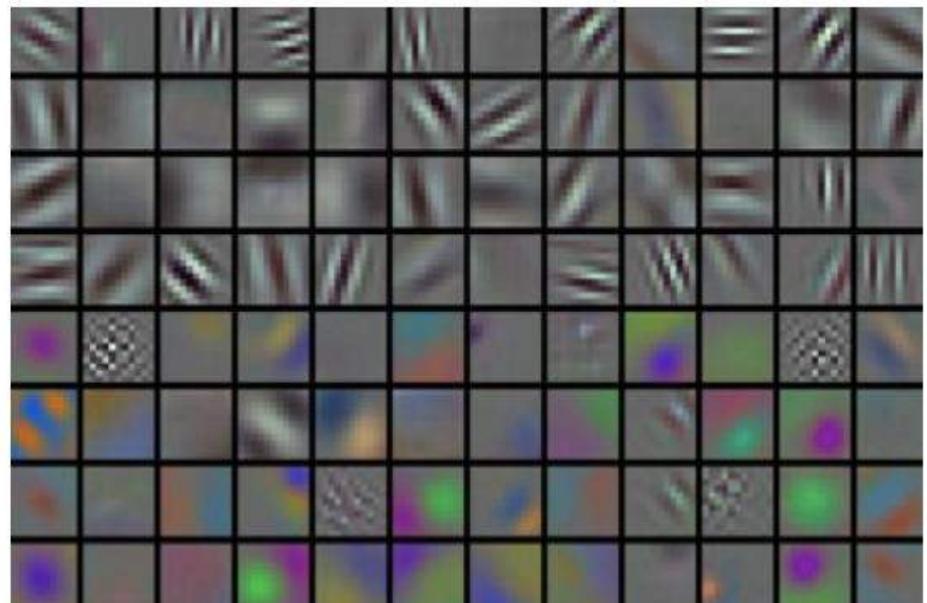
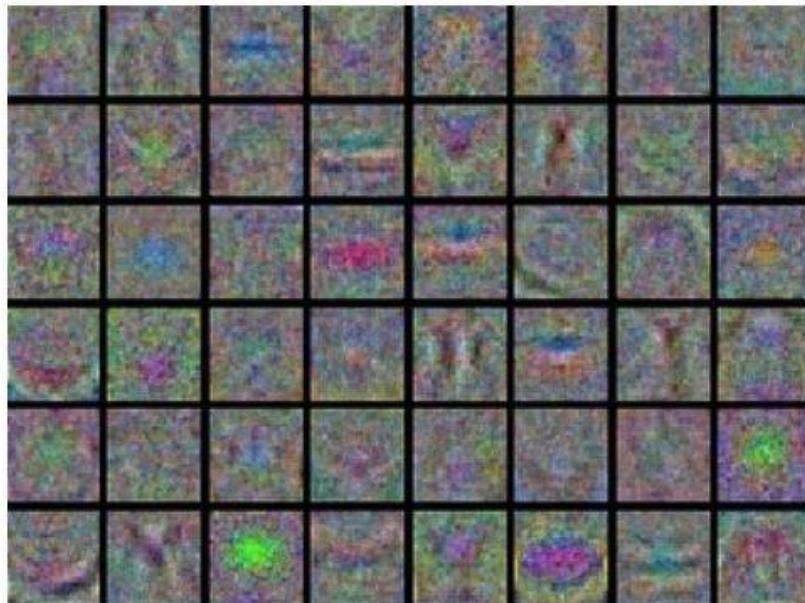


# What if things are not working?

- Plot the histogram of activations per layer
  - E.g., for tanh functions, we expect to see a diverse distribution of values between [-1,1]

# What if things are not working?

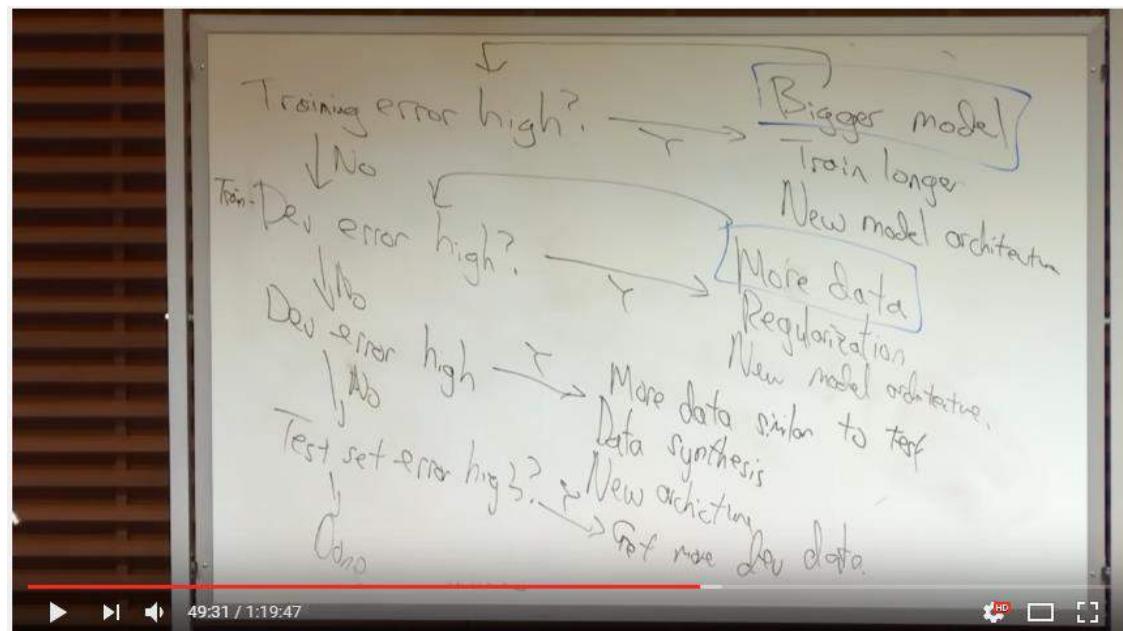
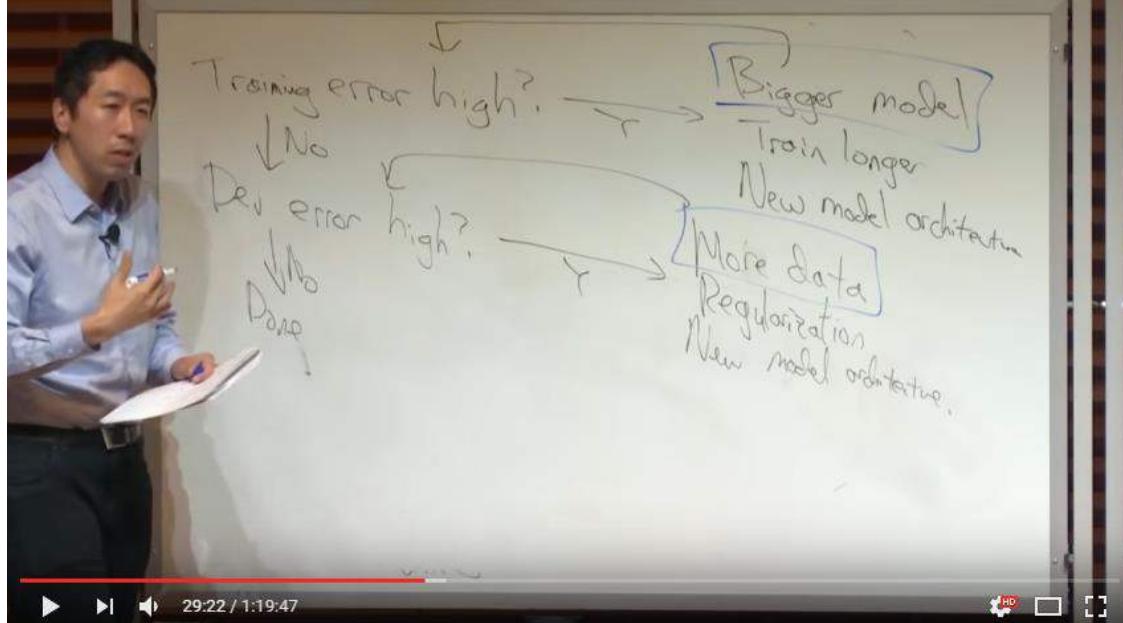
- Visualize your layers (the weights)



Examples of visualized weights for the first layer of a neural network. **Left:** Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. **Right:** Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

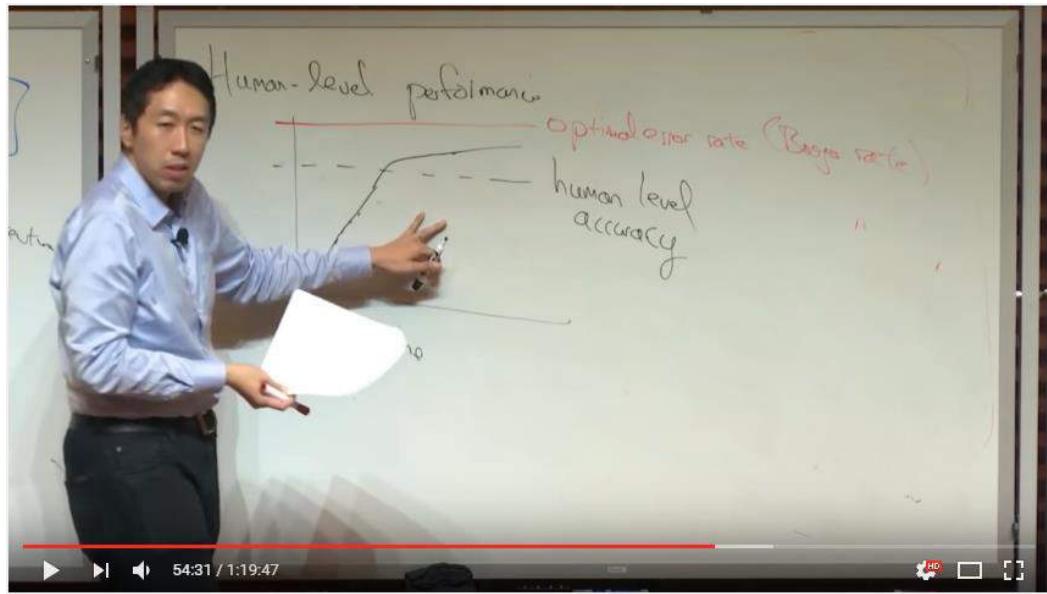
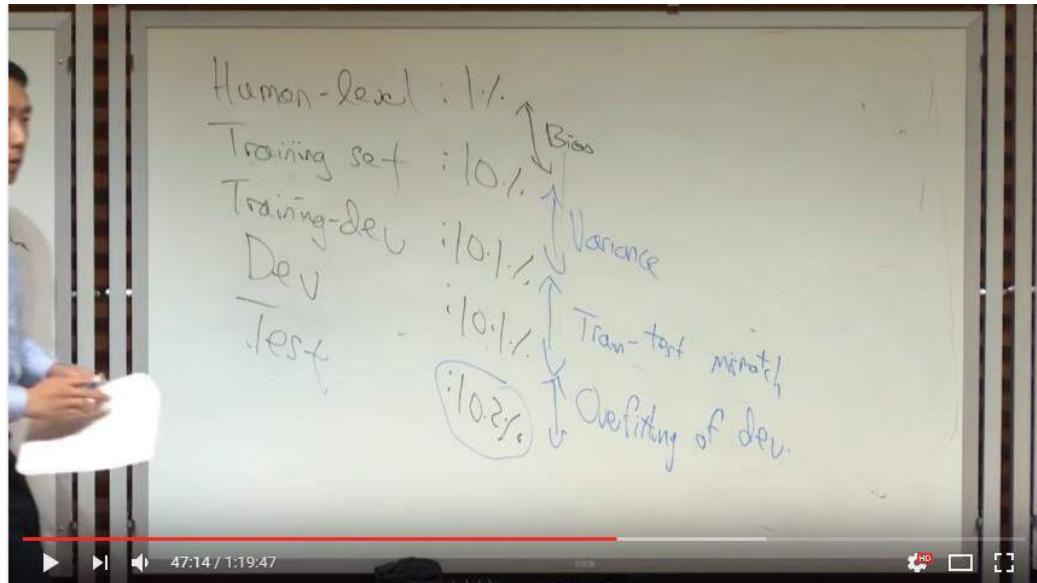
# Andrew Ng's suggestions

- “In DL, the coupling between bias & variance is weaker compared to other ML methods:
  - We can train a network to have high bias and variance.”
- “Dev (validation) and test sets should come from the same distribution. Dev&test sets are like problem specifications.
  - This requires especially attention if you have a lot of data from simulated environments etc. but little data from the real test environment.”



# Andrew Ng's suggestions

- “Knowing the human performance level gives information about the problem of your network:
  - If training error is far from human performance, then there is a bias error.
  - If they are close but validation has more error (compared to the diff between human and training error), then there is variance problem.”
- “After surpassing human level, performance increases only very slowly/difficult.
  - One reason: There is not much space for improvement (only tiny little details). Problem gets much harder.
  - Another reason: We get labels from humans.”



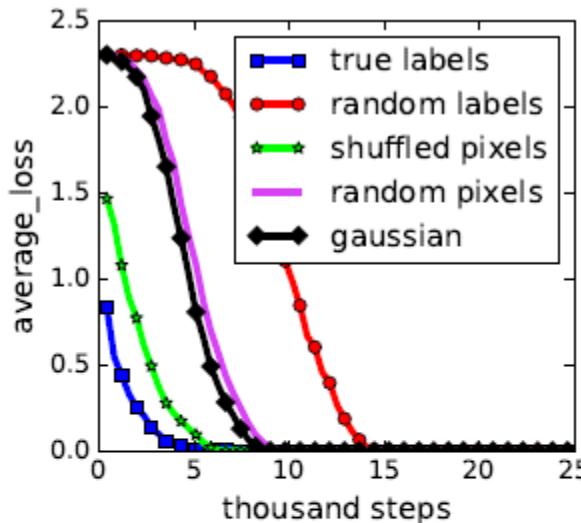
# Also read the following

- 37 reasons why your neural network is not working:
  - <https://medium.com/@slavivanov/4020854bd607>

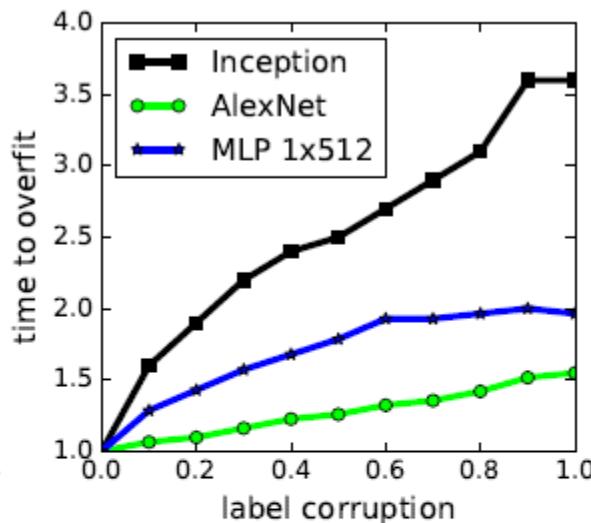
# What is best then?

- Which algorithm to choose?
  - No answer yet
  - See Tom Schaul (2014)
  - RMSprop and AdaDelta seems to be slightly favorable; however, no best algorithm
- SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam are the most widely used ones

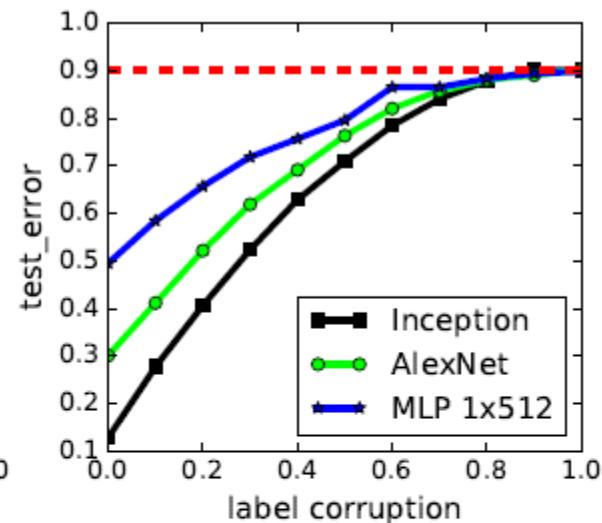
# Luckily, deep networks are very powerful



(a) learning curves



(b) convergence slowdown



(c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Regularization is turned off in the experiments.  
When you turn on regularization, the networks  
perform worse.

UNDERSTANDING DEEP LEARNING REQUIRES RE-THINKING GENERALIZATION

Chiyuan Zhang\*  
Massachusetts Institute of Technology  
chiyuan@mit.edu

Benjamin Recht†  
University of California, Berkeley  
recht@berkeley.edu

Samy Bengio  
Google Brain  
bengio@google.com

Oriol Vinyals  
Google DeepMind  
vinyals@google.com

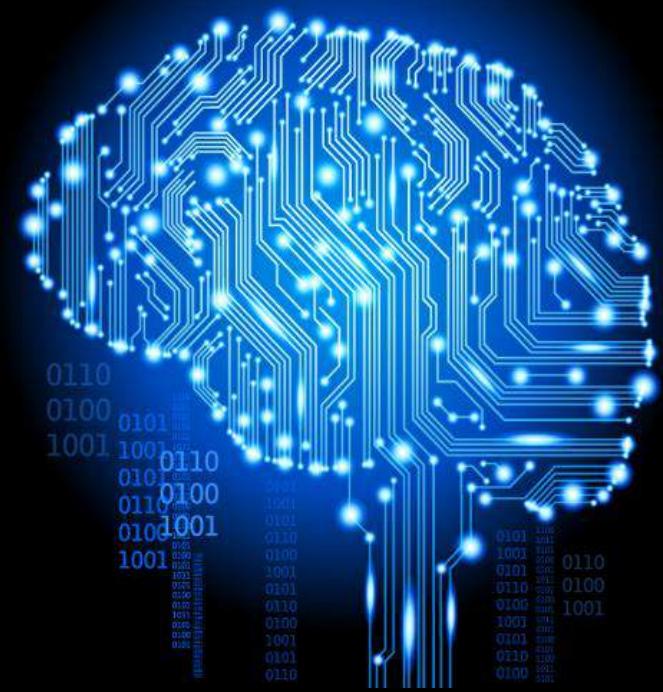
Moritz Hardt  
Google Brain  
mrtz@google.com

# Concluding remarks for the first part

- Loss functions
- Gradients of loss functions for minimizing them
  - All operations in the network should be differentiable
- Gradient descent and its variants
- Initialization, normalization, adaptive learning rate, ...
- Overall, you have learned most of the tools you will use in the rest of the course.

# CENG 783

## Special topics in Deep Learning



© AlchemyAPI

*Week 6 –  
Intro to CVPR & Human Vision*

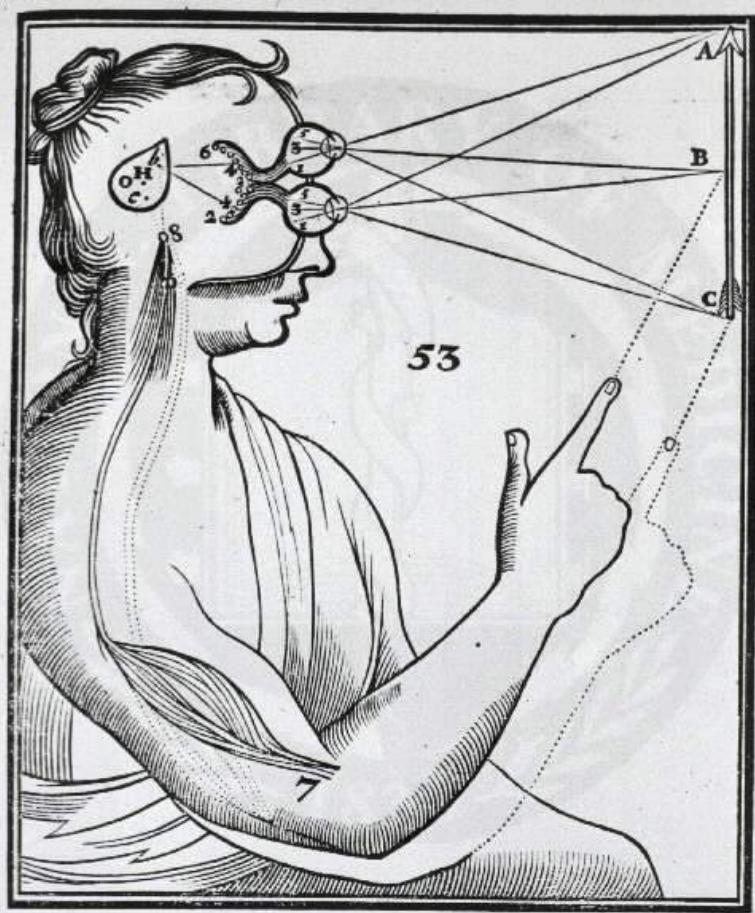
Sinan Kalkan



# today

- A quick introduction to Computer Vision and Pattern Recognition
  - The problem(s) that we are trying to solve
  - The challenges
  - The approaches, especially the hand-designed feature-based ones
- Human vision system in a nutshell
  - Main steps of processing starting from the retina
  - Hierarchies of features in human vision system
- What can we learn for CVPR from human vision system?

# What is Vision?

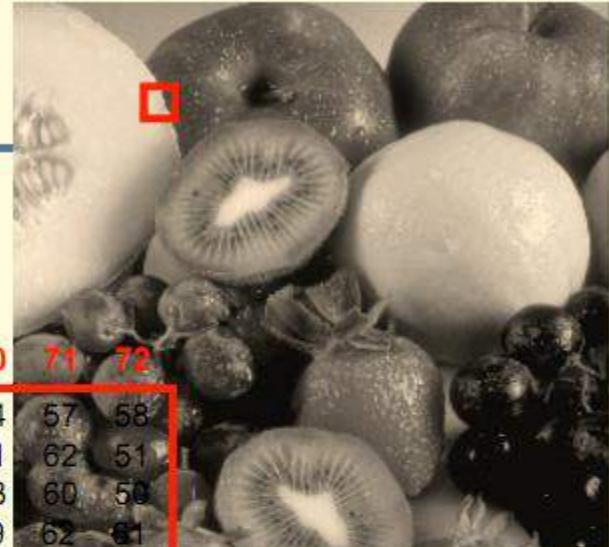


- \* David Marr:
  - \* “Understand **what** is **where** by looking”



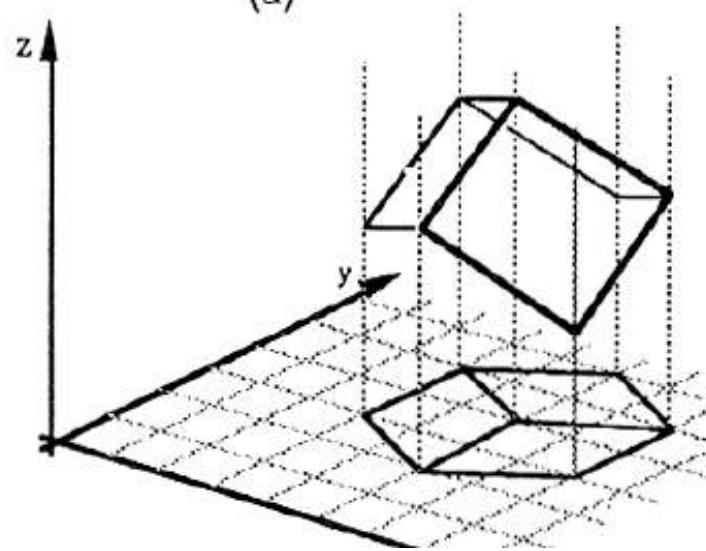
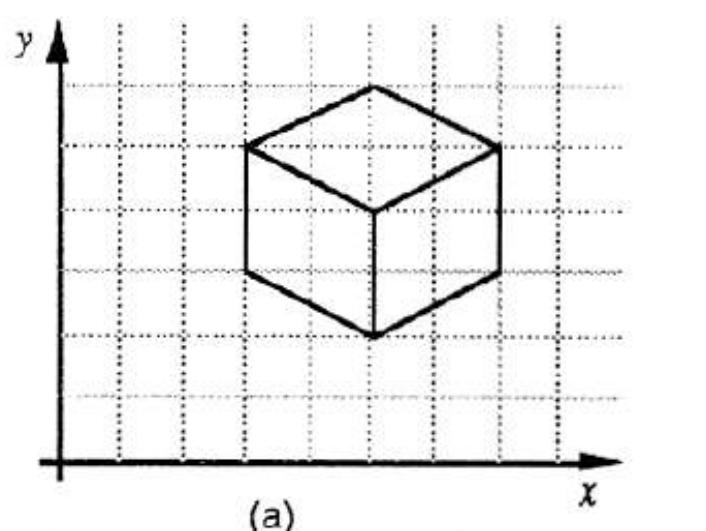
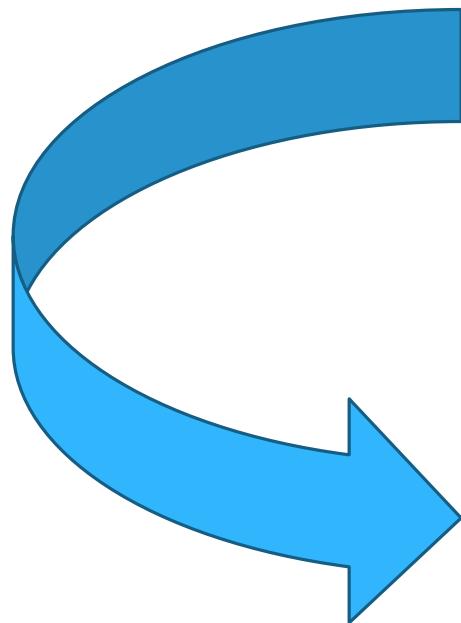
# The starting point

## Grayscale Image



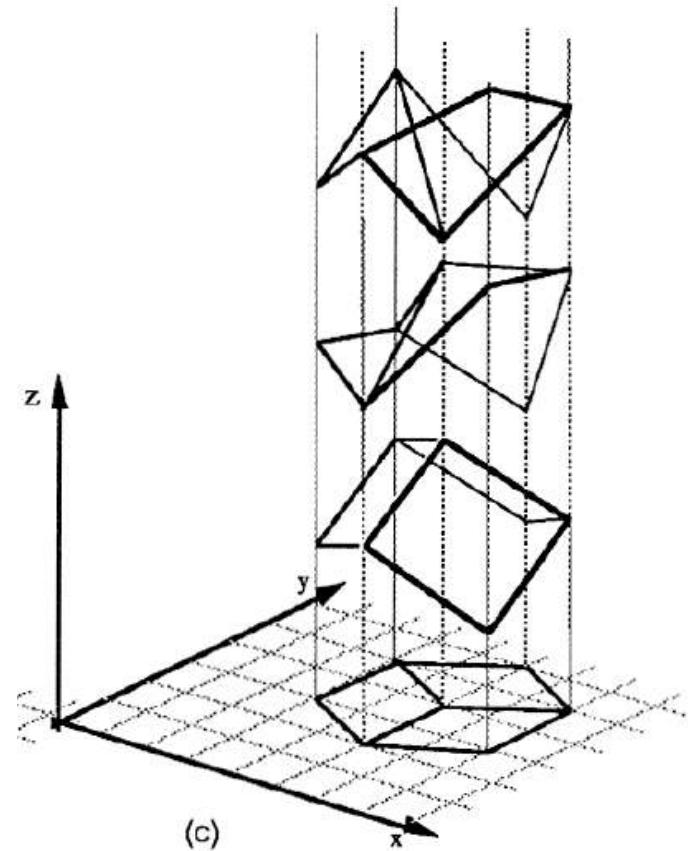
x =	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	
y =	41	210	209	204	202	197	247	143	71	64	80	84	54	54	57	58
	42	206	196	203	197	195	210	207	56	63	58	53	53	61	62	51
	43	201	207	192	201	198	213	156	69	65	57	55	52	53	60	59
	44	216	206	211	193	202	207	208	57	69	60	55	77	49	62	61
	45	221	206	211	194	196	197	220	56	63	60	55	46	97	58	106
	46	209	214	224	199	194	193	204	173	64	60	59	51	62	56	48
	47	204	212	213	208	191	190	191	214	60	62	66	76	51	49	55
	48	214	215	215	207	208	180	172	188	69	72	55	49	56	52	56
	49	209	205	214	205	204	196	187	196	86	62	66	87	57	60	48
	50	208	209	205	203	202	186	174	185	149	71	63	55	55	45	56
	51	207	210	211	199	217	194	183	177	209	90	62	64	52	93	52
	52	208	205	209	209	197	194	183	187	187	239	58	68	61	51	56
	53	204	206	203	209	195	203	188	185	183	221	75	61	58	60	60
	54	200	203	199	236	188	197	183	190	183	196	122	63	58	64	66
	55	205	210	202	203	199	197	196	181	173	186	105	62	57	64	63

# Vision: An inverse problem

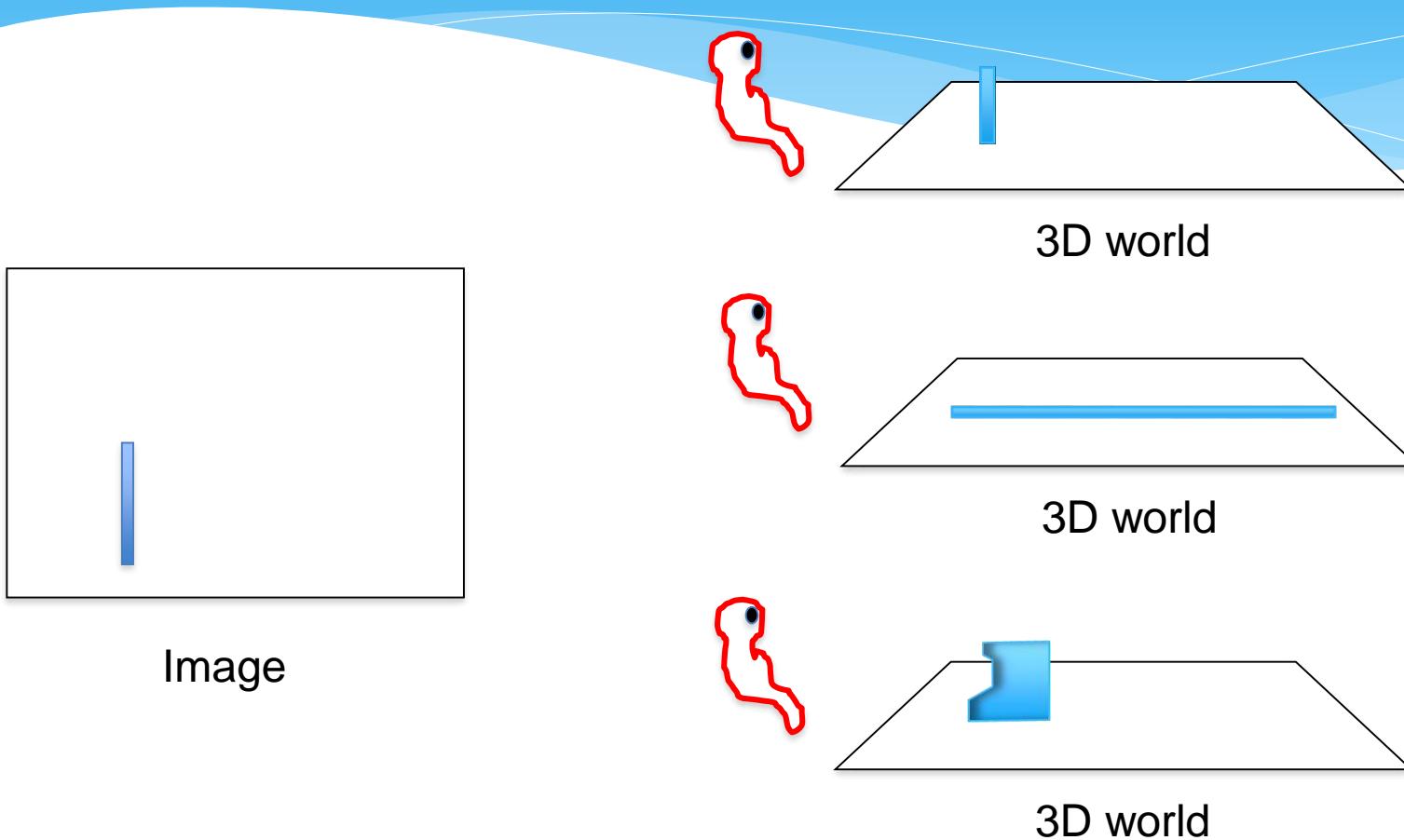


# Vision: An inverse problem (2)

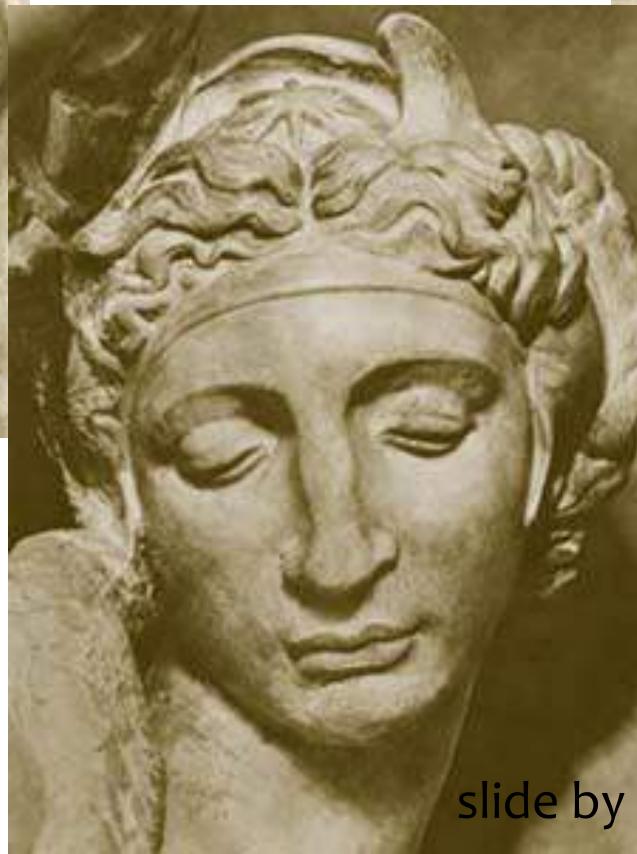
- \* In fact, it is **ill-posed**.
- \* A problem is well-posed if:
  - \* A solution exists
  - \* The solution is unique
  - \* The solution depends continuously on the data,



# Vision: An inverse problem (3)



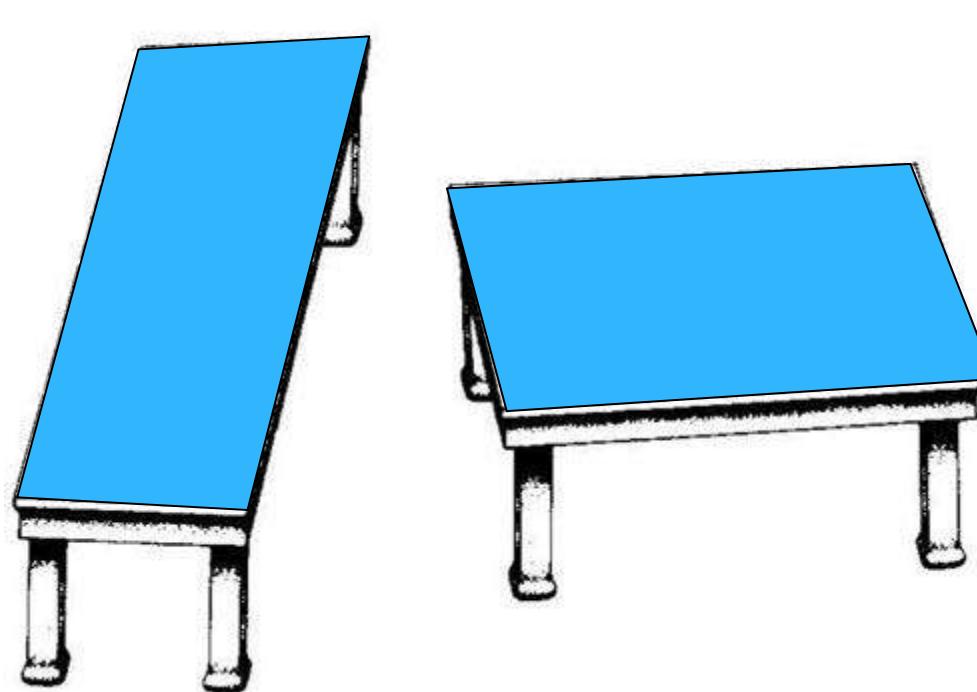
# Challenges 1: view point variation



Michelangelo 1475-1564

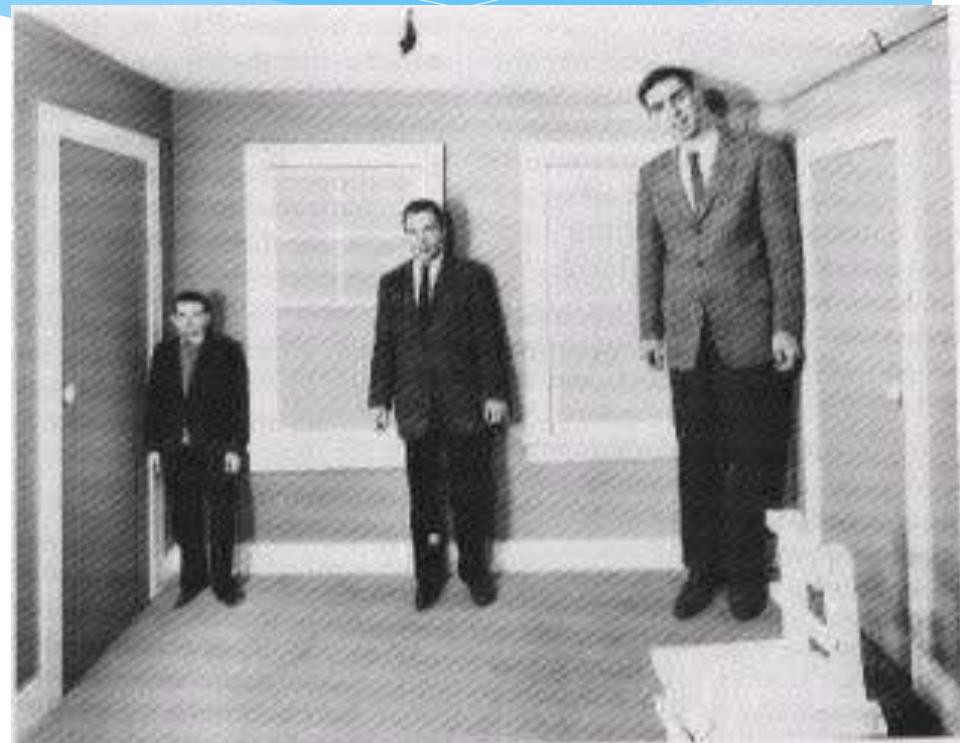
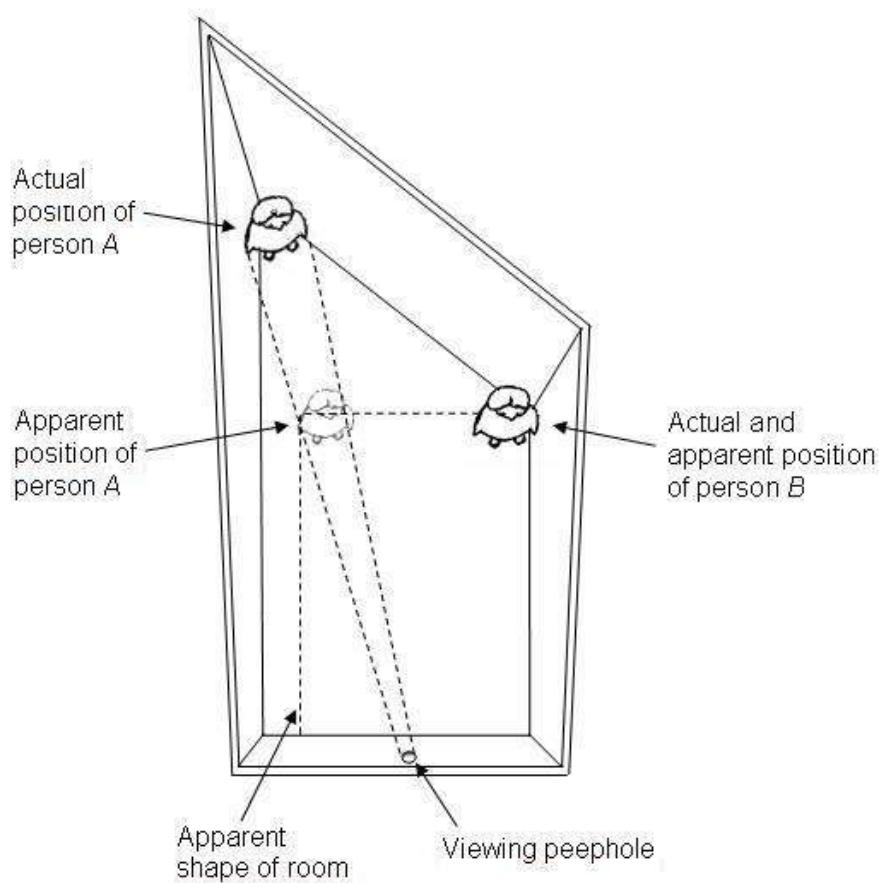
slide by Fei Fei, Fergus & Torralba

# Challenges 1: view point variation



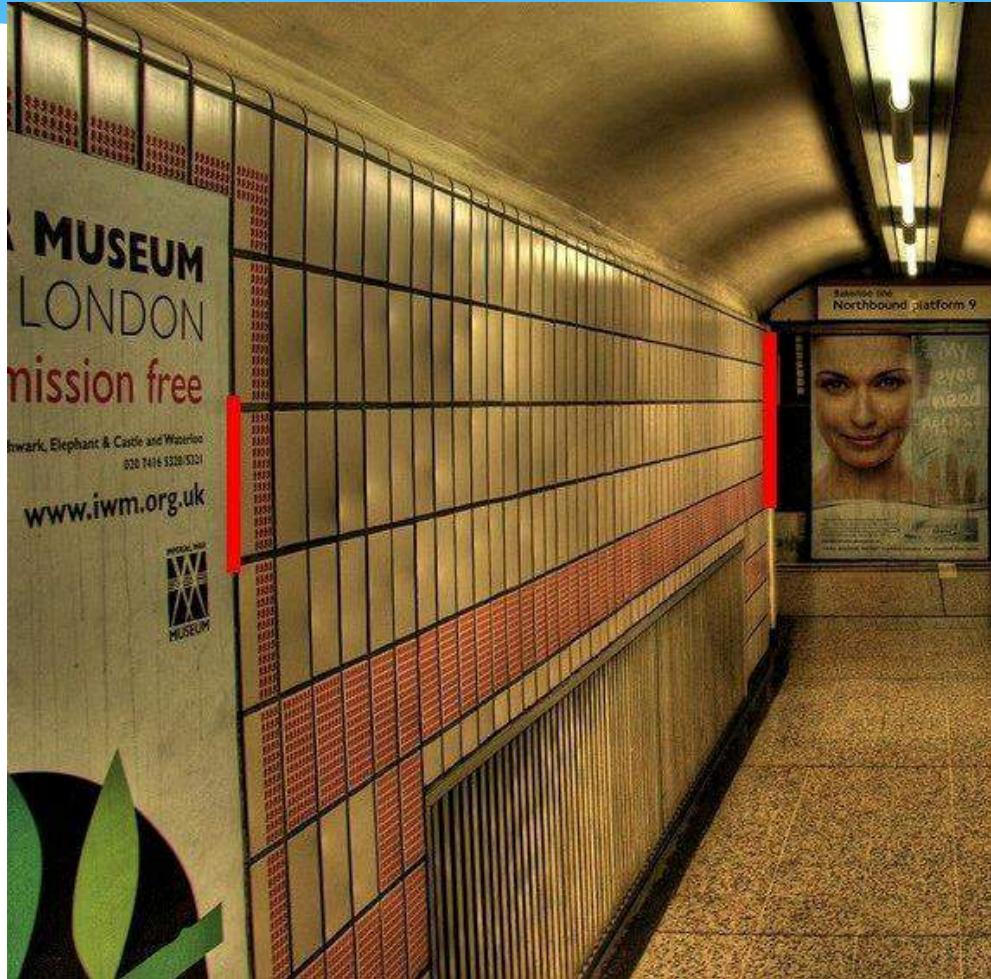
by Roger Shepard ("Turning the Tables")

# Challenges 1: view point variation



Ames room

# Challenges 1: view point variation



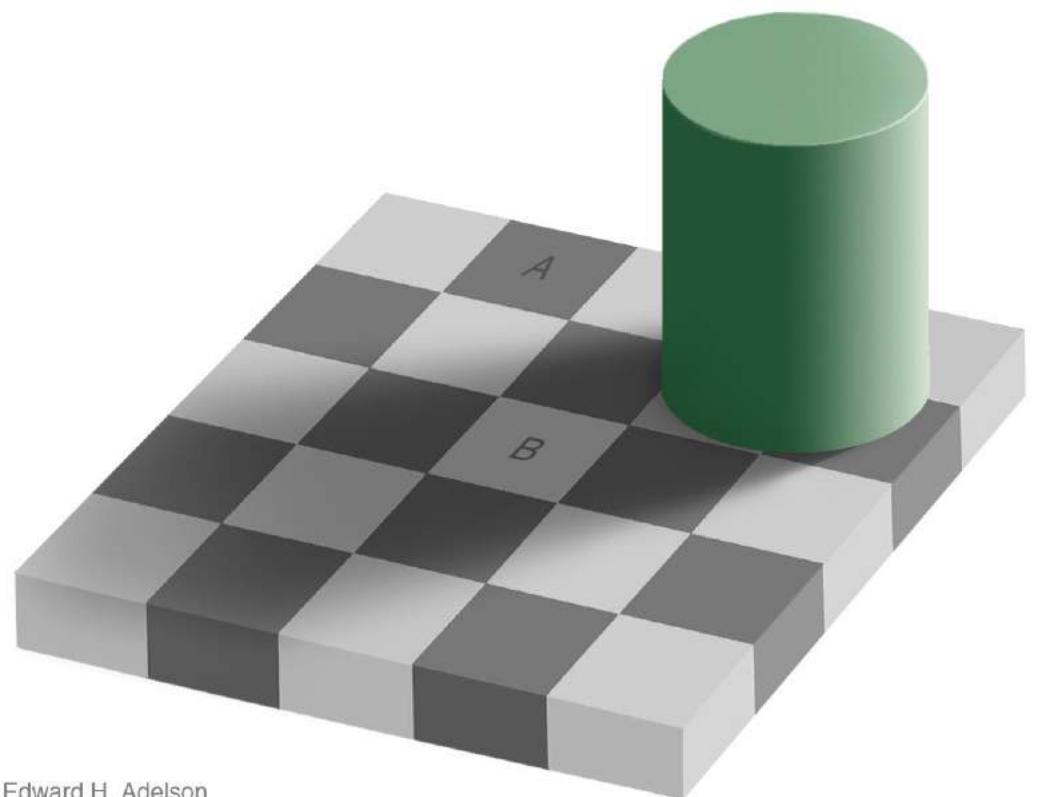
(c) 2006 Walt Anthony

## Challenges 2: illumination



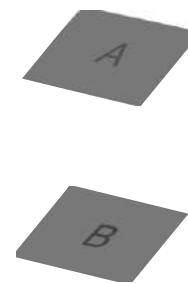
slide credit: S. Ullman

# Challenges 2: illumination

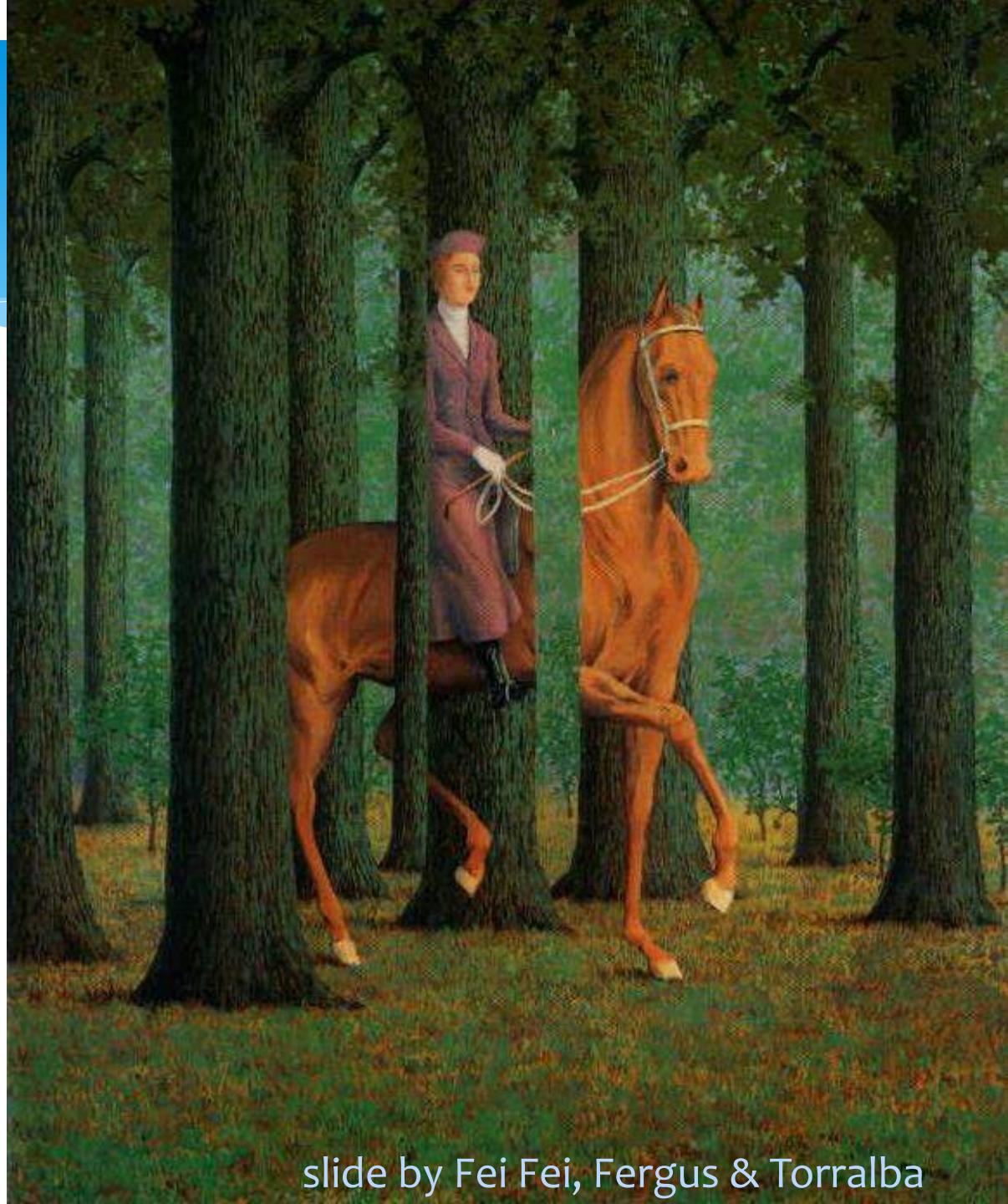


Edward H. Adelson

# Challenges 2: illumination



## Challenge 3: occlusion



Magritte, 1957

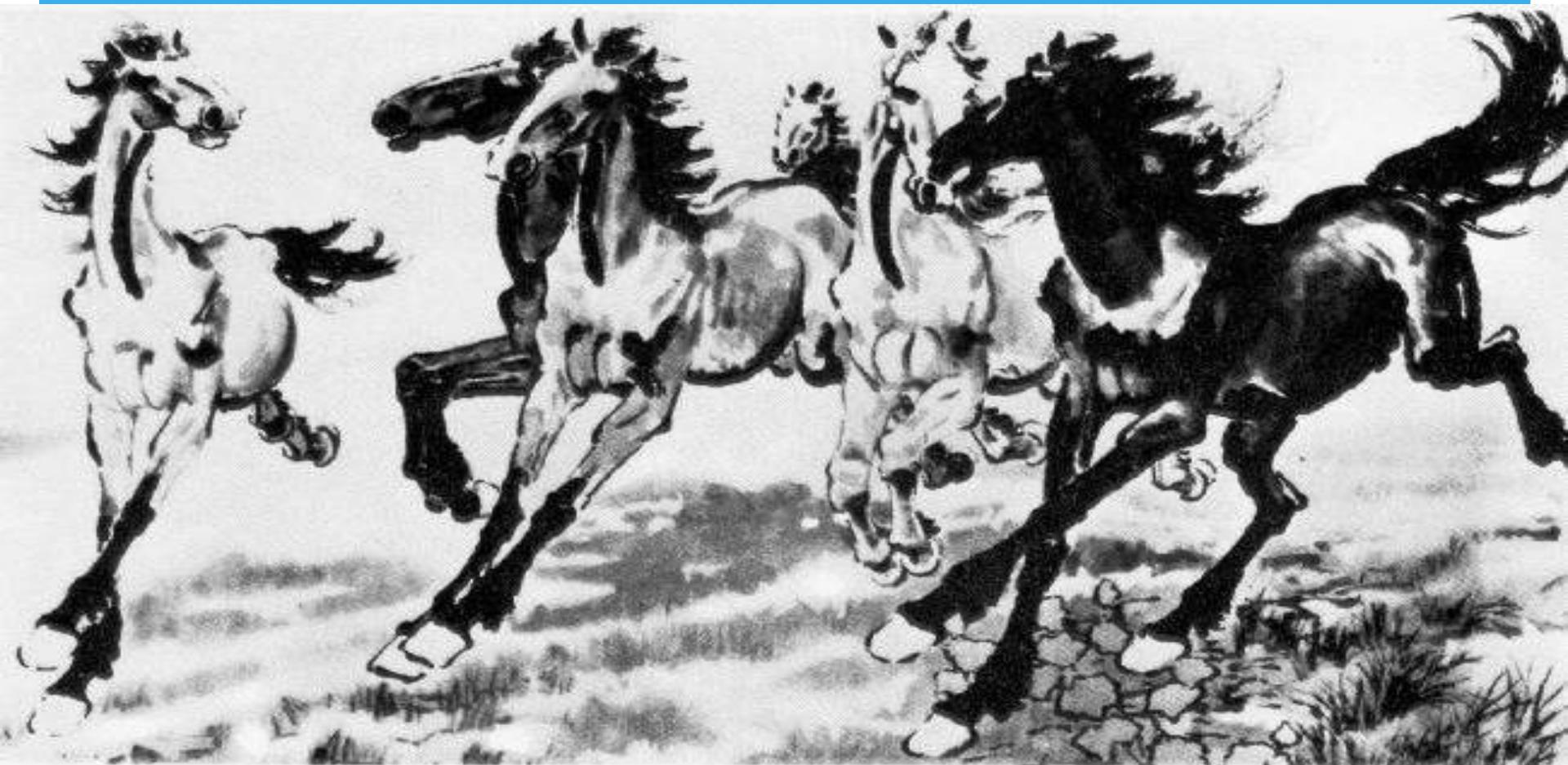
slide by Fei Fei, Fergus & Torralba

## Challenge 4: scale



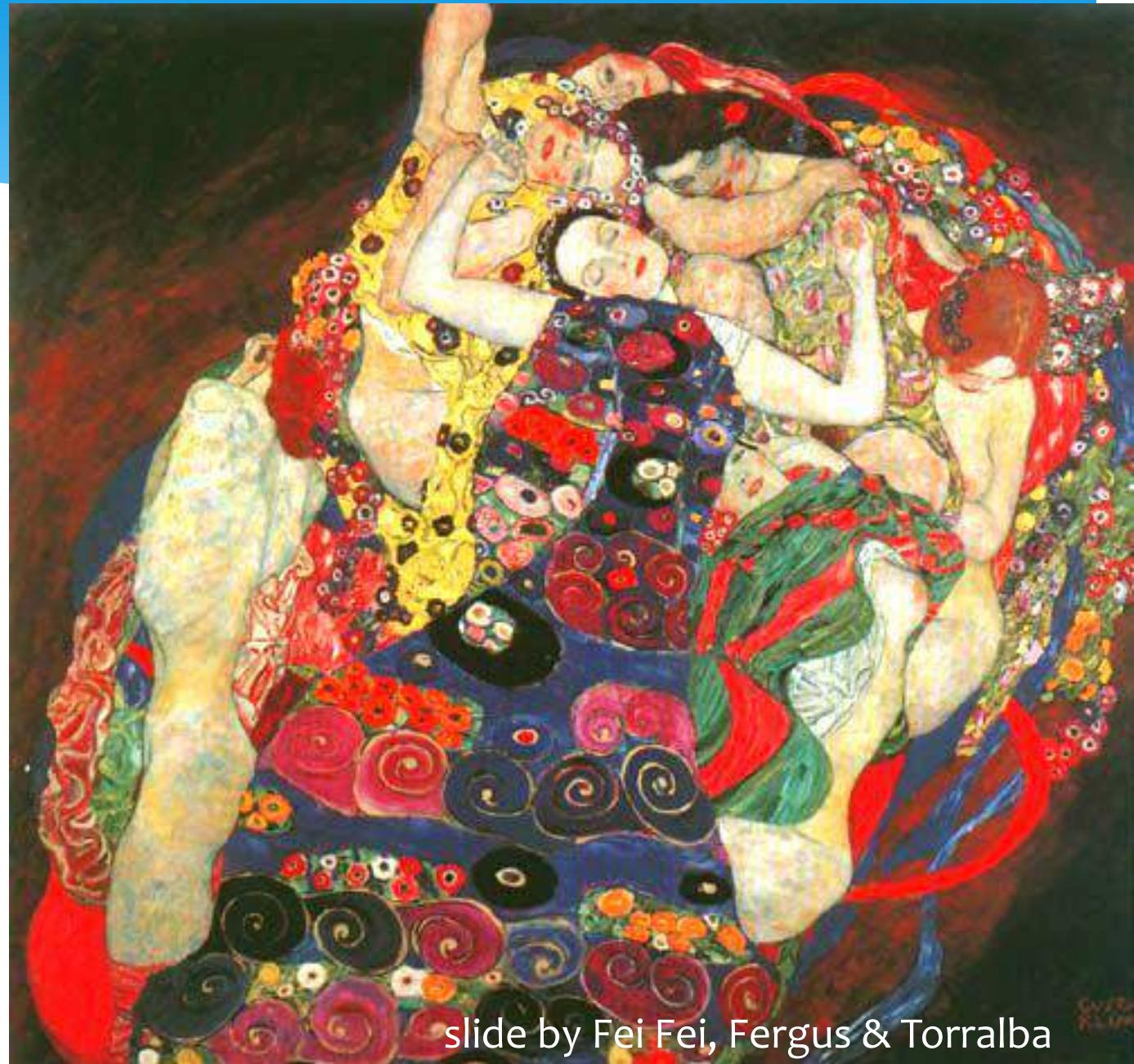
slide by Fei Fei, Fergus & Torralba

## Challenges 5: deformation



Xu, Beihong 1943

## Challenges 6: background clutter



Klimt, 1913

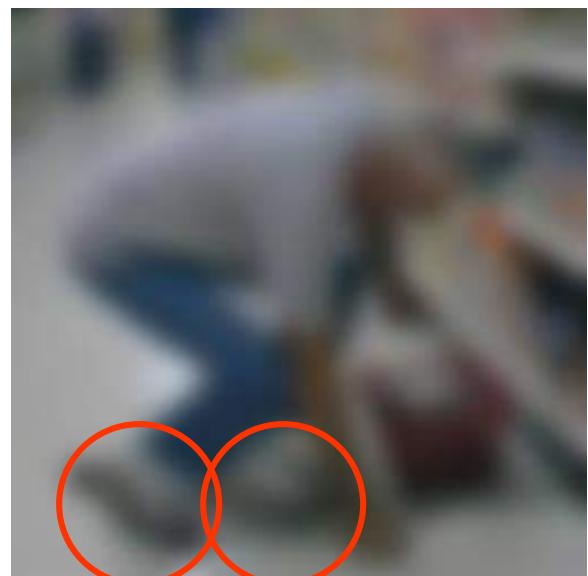
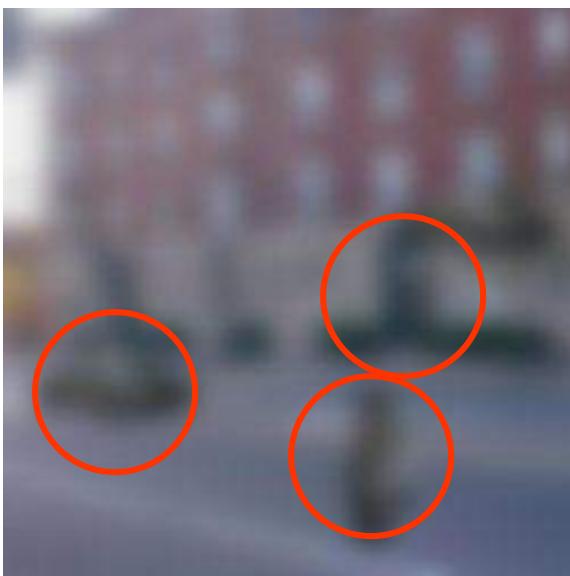
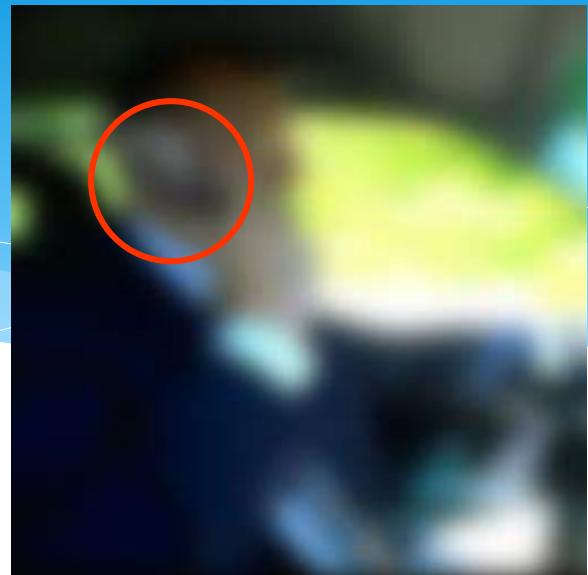
slide by Fei Fei, Fergus & Torralba

## Challenges 7: object intra-class variation



slide by Fei-Fei, Fergus & Torralba

## Challenges 8: local ambiguity



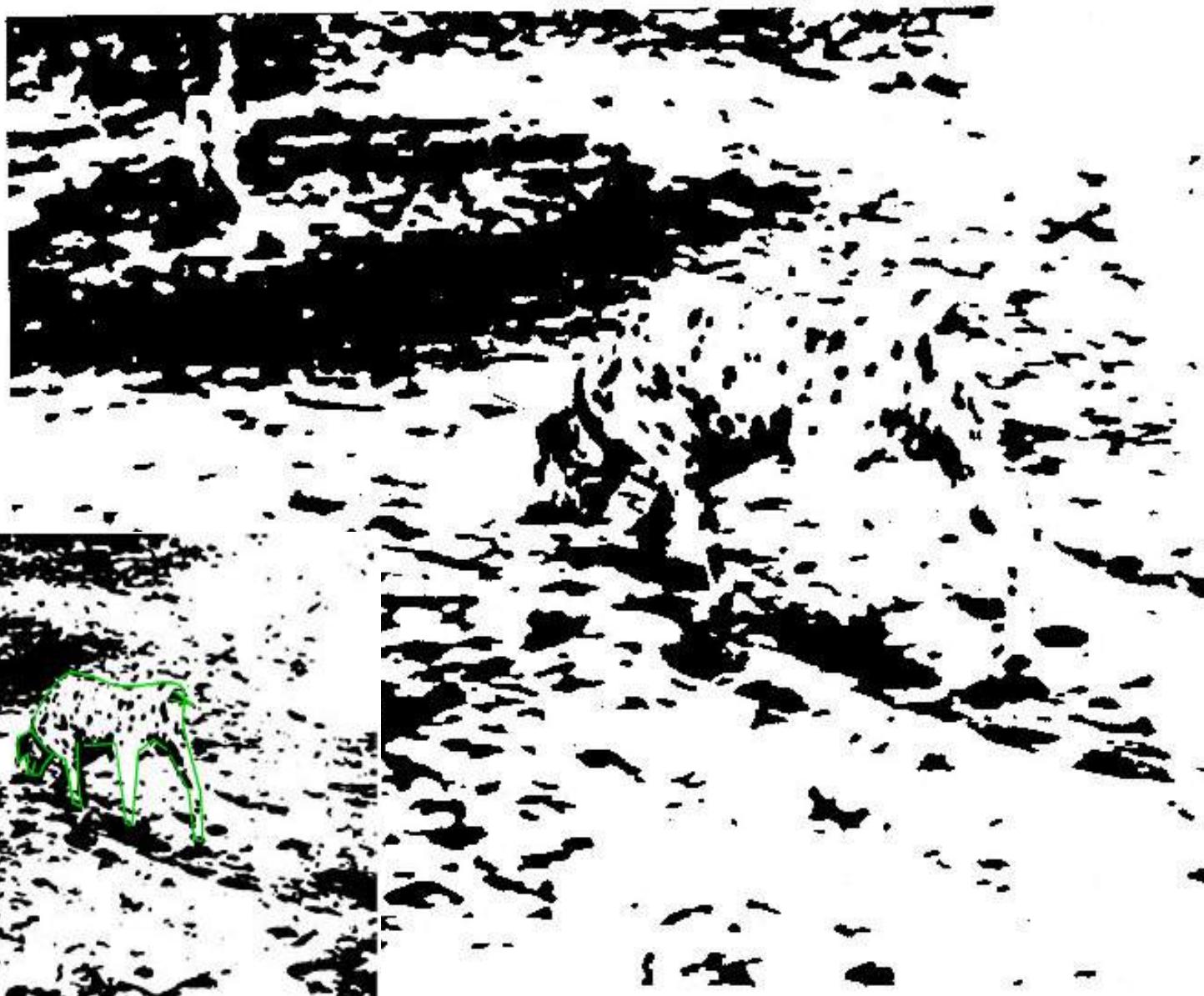
slide by Fei-Fei, Fergus & Torralba





slide credit: A. Torralba

## Challenges 8: Prior Knowledge

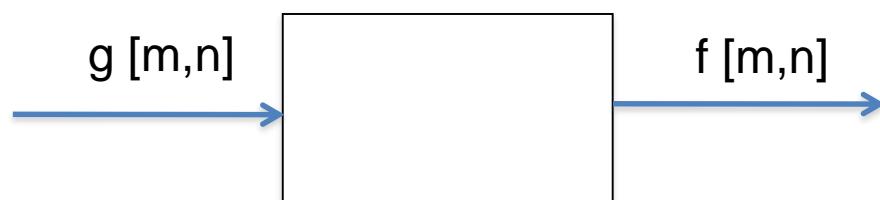


Gregory R (1970) "The intelligent eye", Fotoğraf: RC James

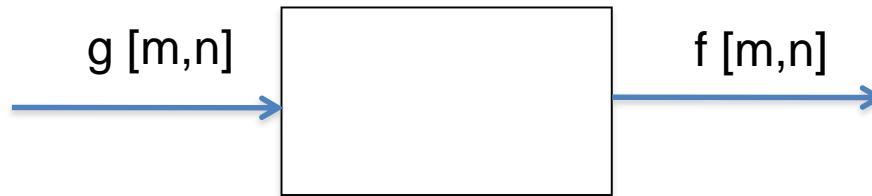
# Now

- Image Gradient and Orientation
- Edges
- Corners
- Hessian matrix

# Filtering



# Linear filtering



For a linear system, each output is a linear combination of all the input values:

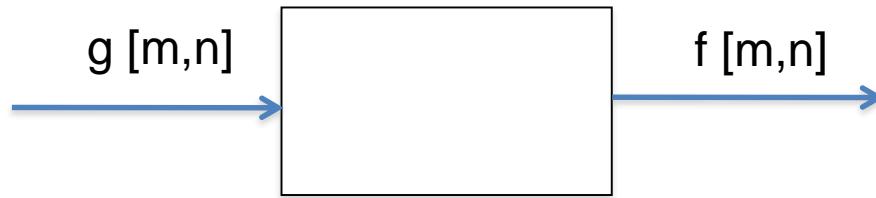
$$f[m,n] = \sum_{k,l} h[m,n,k,l] g[k,l]$$

In matrix form:

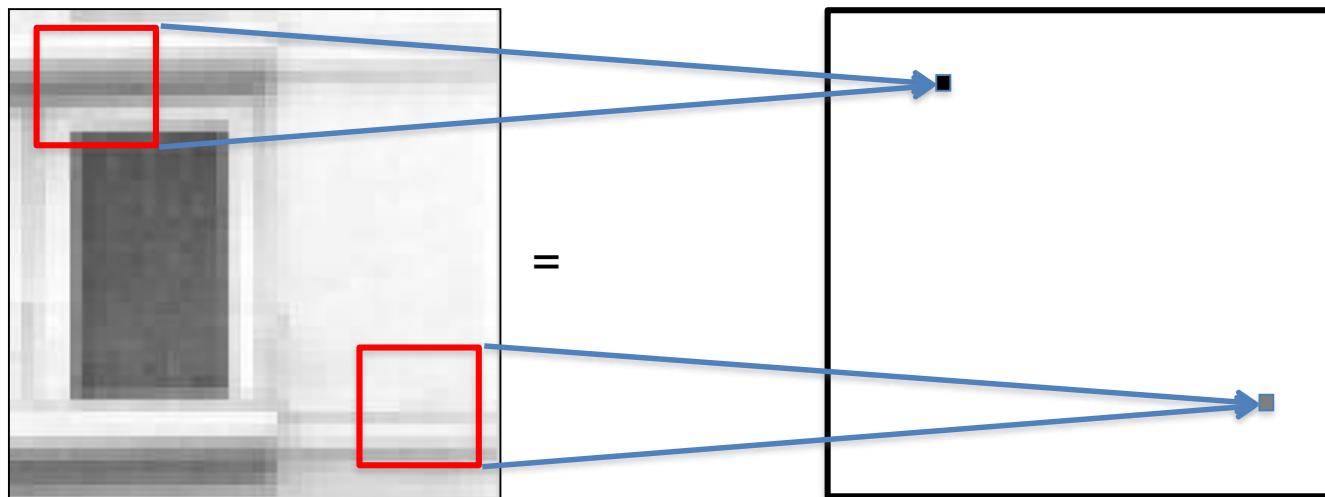
$$\mathbf{F} = \mathbf{H} \mathbf{G}$$



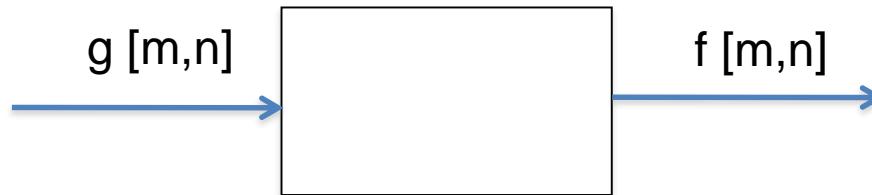
# Linear filtering



$$f[m,n] = I \otimes g = \sum_{k,l} h[m-k, n-l] g[k, l]$$



# Linear filtering



$$f[m,n] = I \otimes g = \sum_{k,l} h[m-k, n-l] g[k, l]$$

$m=0 \ 1 \ 2 \ \dots$

111	115	113	111	112	111	112	111
135	138	137	139	145	146	149	147
163	168	188	196	206	202	206	207
180	184	206	219	202	200	195	193
189	193	214	216	104	79	83	77
191	201	217	220	103	59	60	68
195	205	216	222	113	68	69	83
199	203	223	228	108	68	71	77



-1	2	-1
-1	2	-1
-1	2	-1

$g[m,n]$

$h[m,n]$

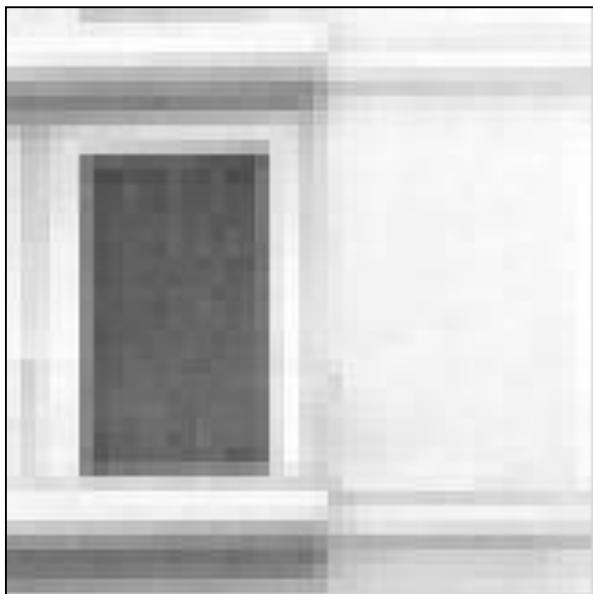
=

?	?	?	?	?	?	?	?	?
?	-5	9	-9	21	-12	10	?	?
?	-29	18	24	4	-7	5	?	?
?	-50	40	142	-88	-34	10	?	?
?	-41	41	264	-175	-71	0	?	?
?	-24	37	349	-224	-120	-10	?	?
?	-23	33	360	-217	-134	-23	?	?
?	?	?	?	?	?	?	?	?

$f[m,n]$

# Impulse

$$f[m, n] = I \otimes g = \sum_{k, l} h[m - k, n - l]g[k, l]$$

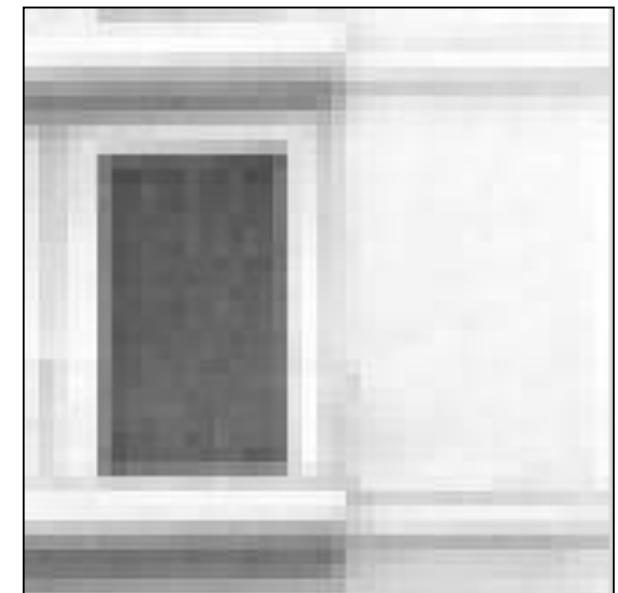


$g[m, n]$

$\otimes$

0	0	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	0	0	0

$h[m, n]$

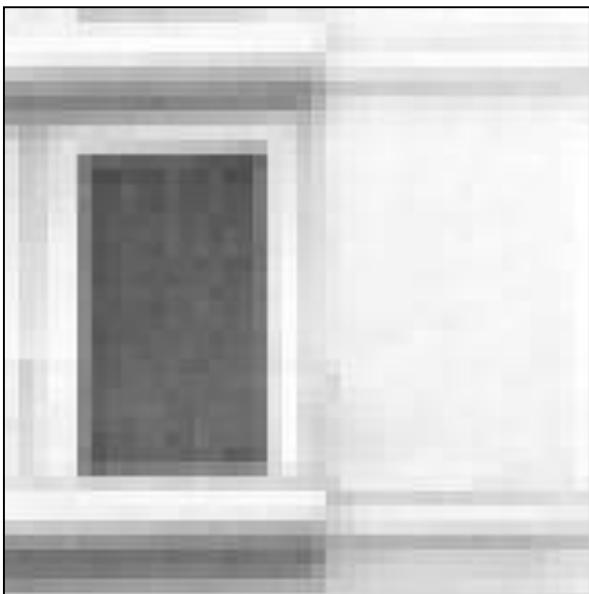


$f[m, n]$

# Shifts

$$f[m, n] = I \otimes g = \sum_{k, l} h[m - k, n - l]g[k, l]$$

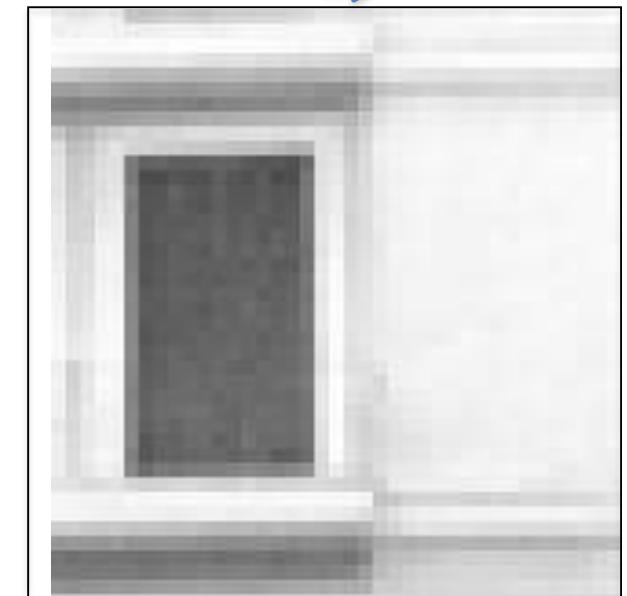
2pixels



$\otimes$

0	0	0	0	0
0	0	0	0	0
0	0	0	0	1
0	0	0	0	0
0	0	0	0	0

$h[m, n]$



$g[m, n]$

$f[m, n]$

# Rectangular filter



$g[m,n]$

$\otimes$



$h[m,n]$

=



$f[m,n]$

# Rectangular filter



$g[m,n]$

$\otimes$



=

$h[m,n]$



$f[m,n]$

# Rectangular filter



$g[m,n]$

$\otimes$

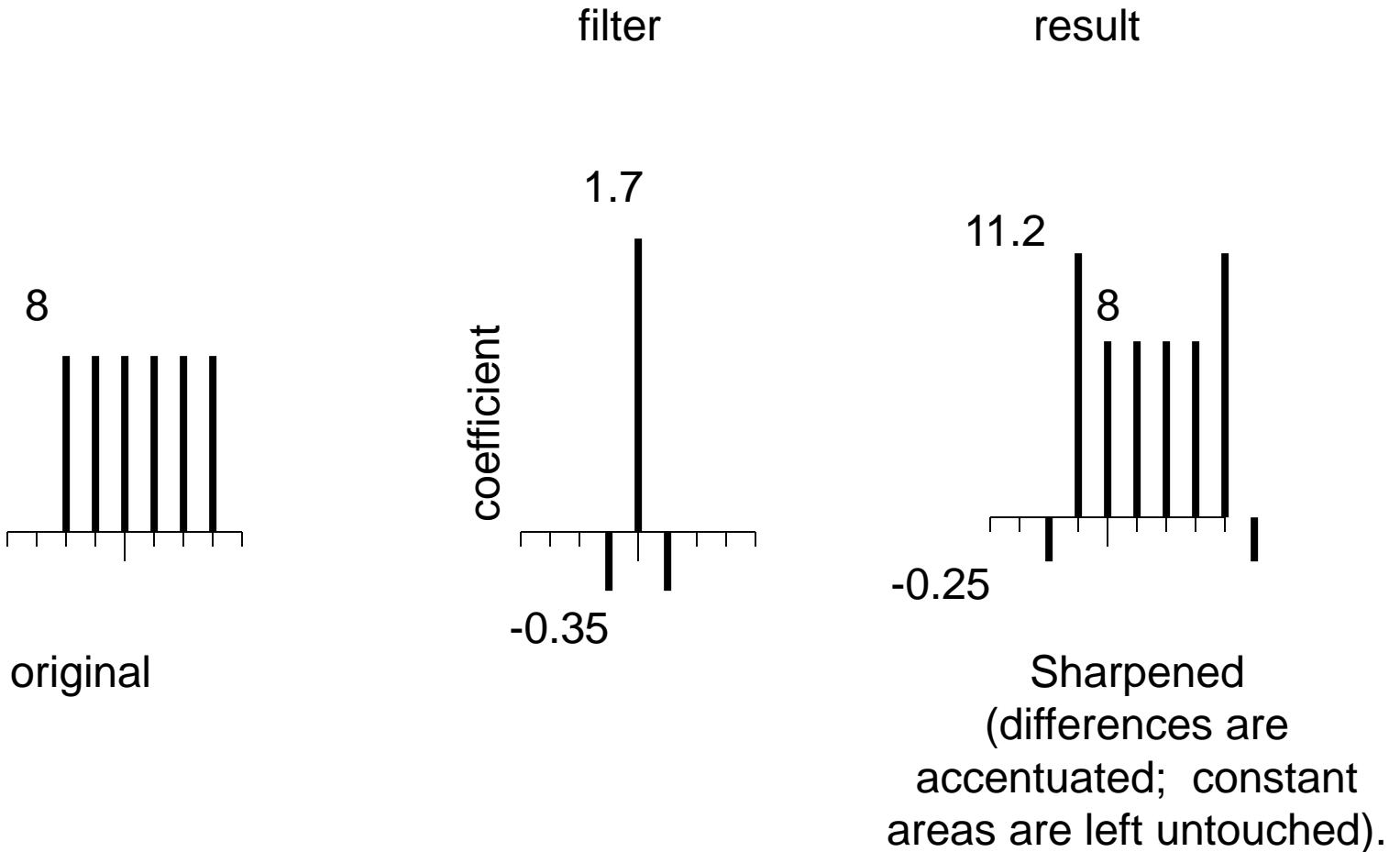
$h[m,n]$

=

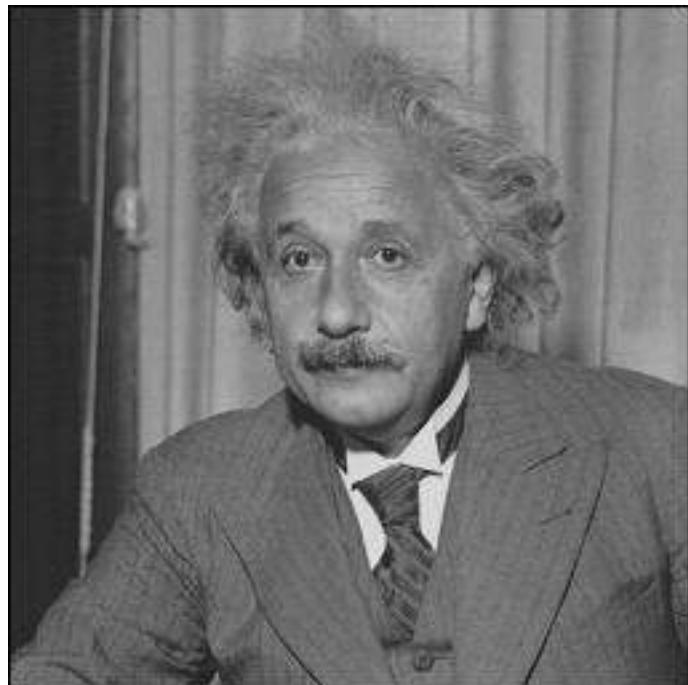


$f[m,n]$

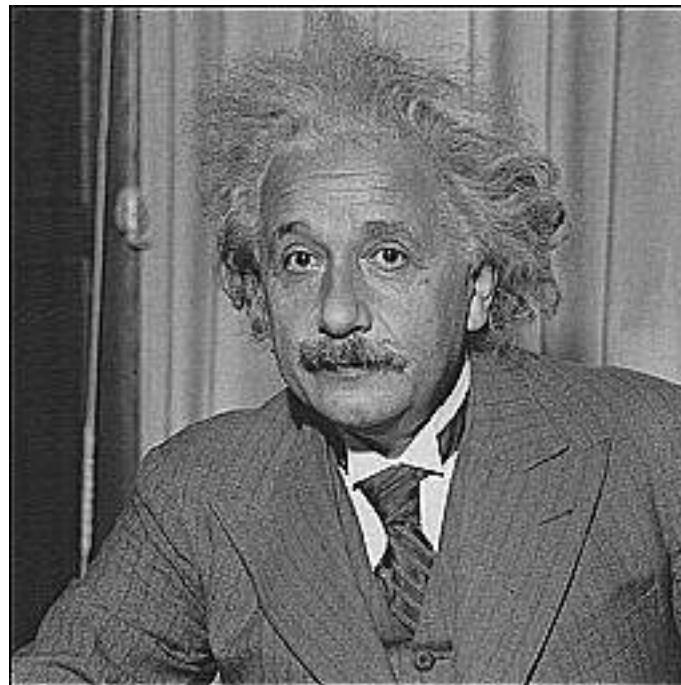
# Sharpening example



# Sharpening



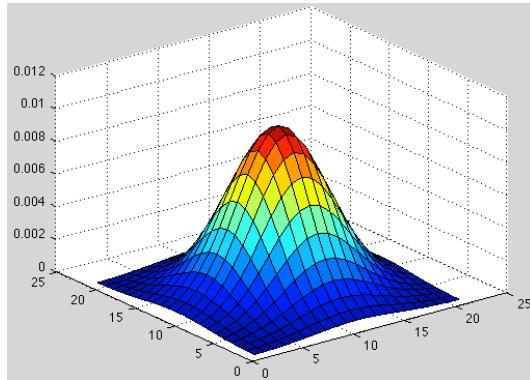
**before**



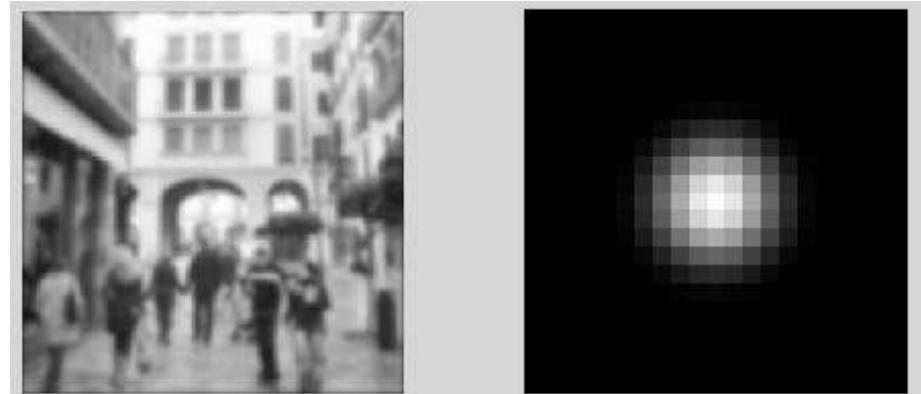
**after**

# Gaussian filter

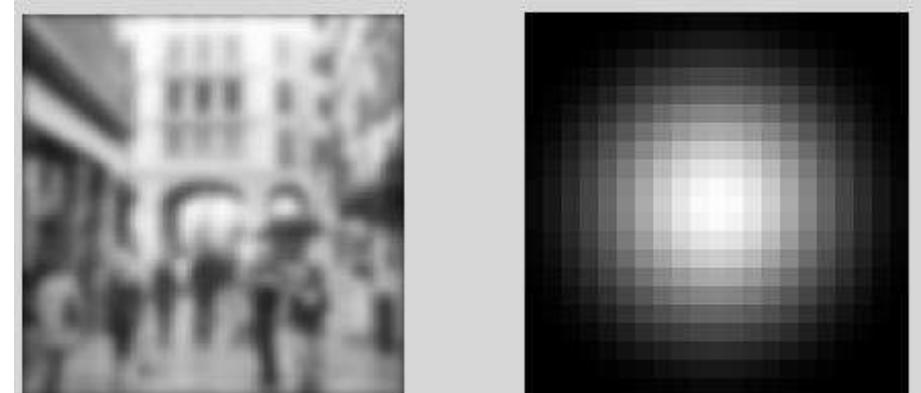
$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



$\sigma=1$

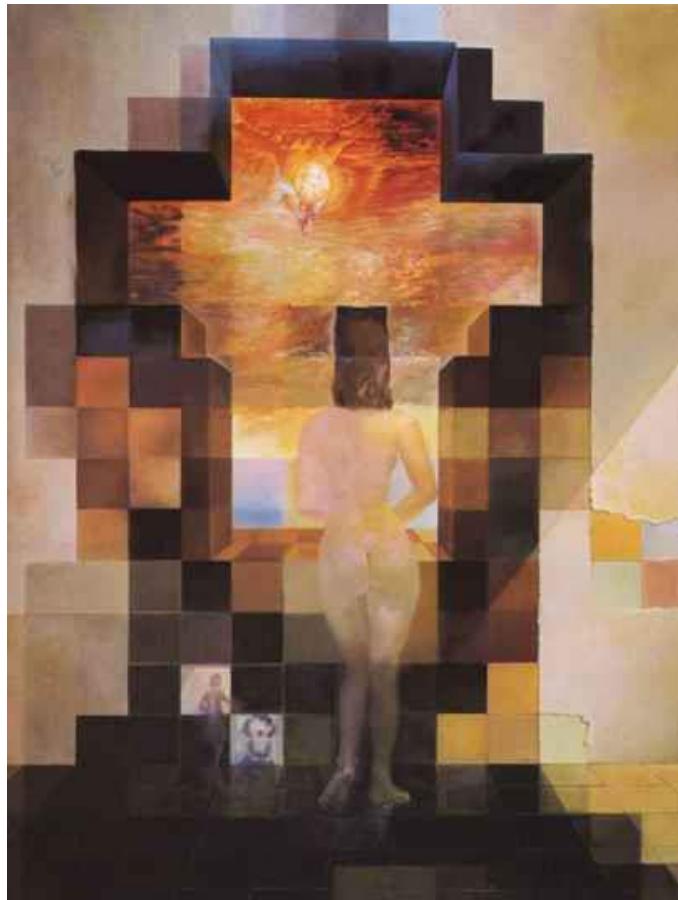


$\sigma=2$

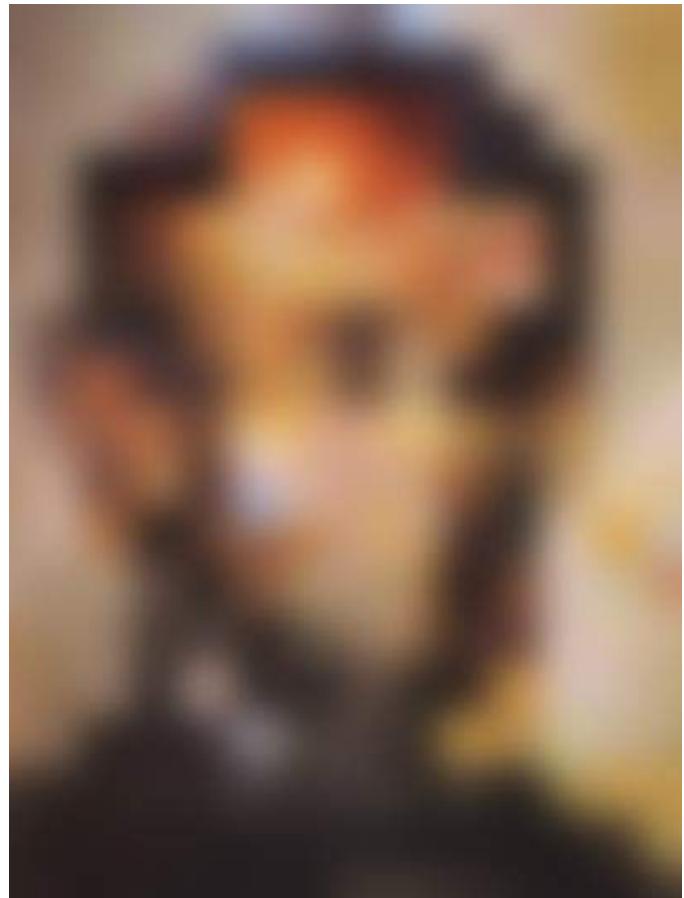
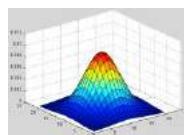


$\sigma=4$

# Global to Local Analysis



Dali



$$[-1 \ 1]$$



$g[m,n]$



$$[-1, 1]$$

$h[m,n]$



$f[m,n]$



$$[-1 \ 1]^T$$

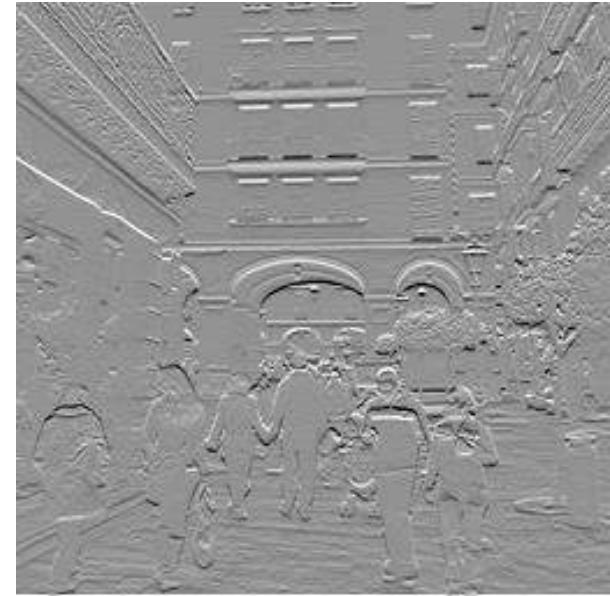


$g[m,n]$



$$\otimes \quad [-1, 1]^T =$$

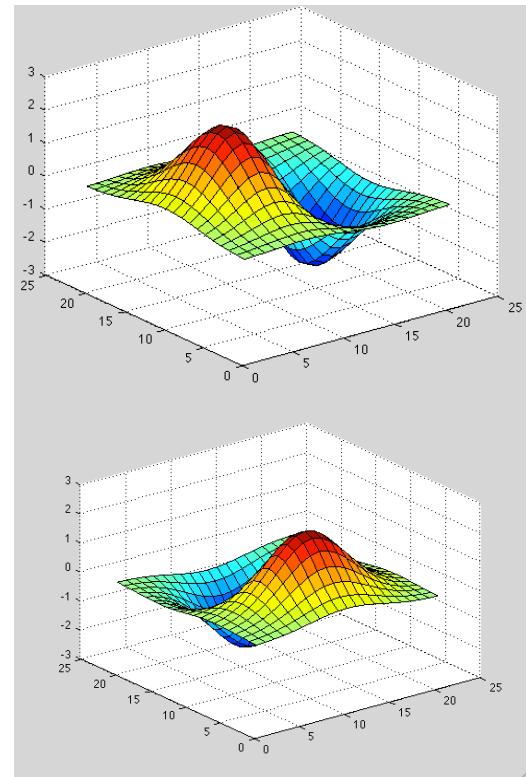
$$h[m,n]$$



$f[m,n]$

$$h_x(x,y) = \frac{\partial h(x,y)}{\partial x} = \frac{-x}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$h_y(x,y) = \frac{\partial h(x,y)}{\partial y} = \frac{-y}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



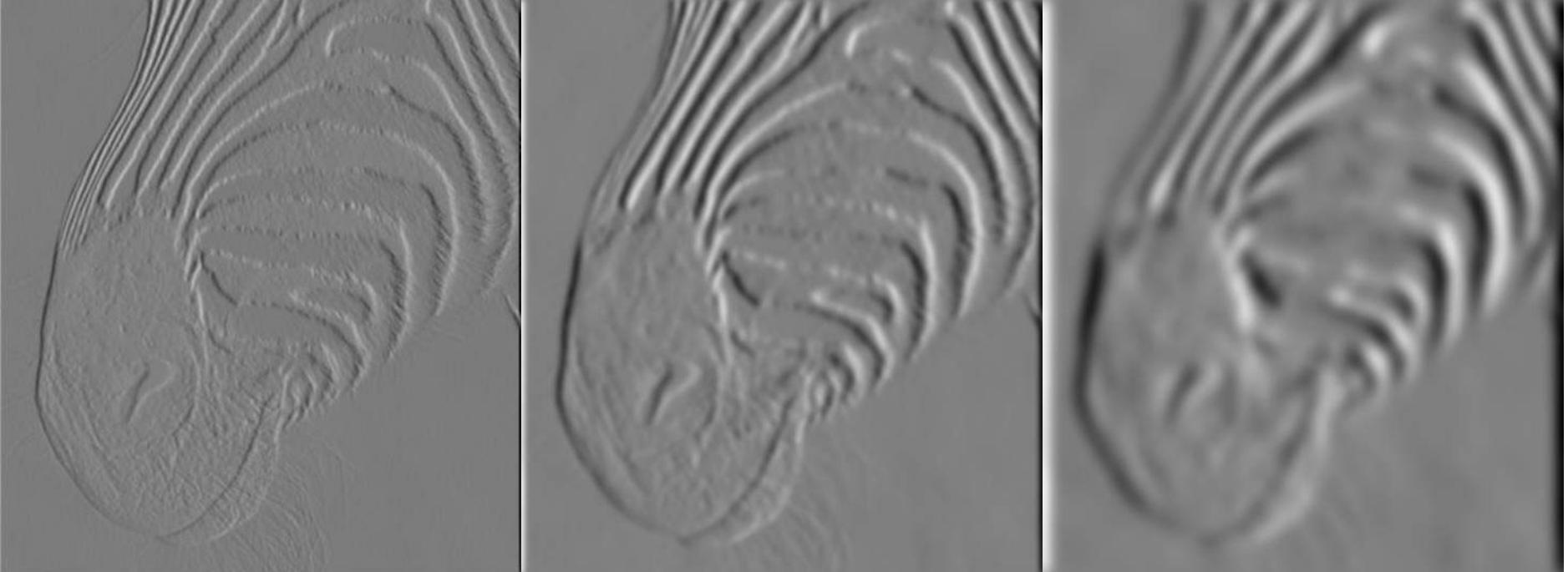
Magnitude:  $h_x(x,y)^2 + h_y(x,y)^2$

*Edge strength*

Angle:

$$\arctan\left(\frac{h_y(x,y)}{h_x(x,y)}\right)$$

*Edge orientation*



1 pixel

3 pixels

7 pixels

The scale of the smoothing filter affects derivative estimates, and also the semantics of the edges recovered.



Gradient magnitudes at scale 1

Gradient magnitudes at scale 2

### Issues:

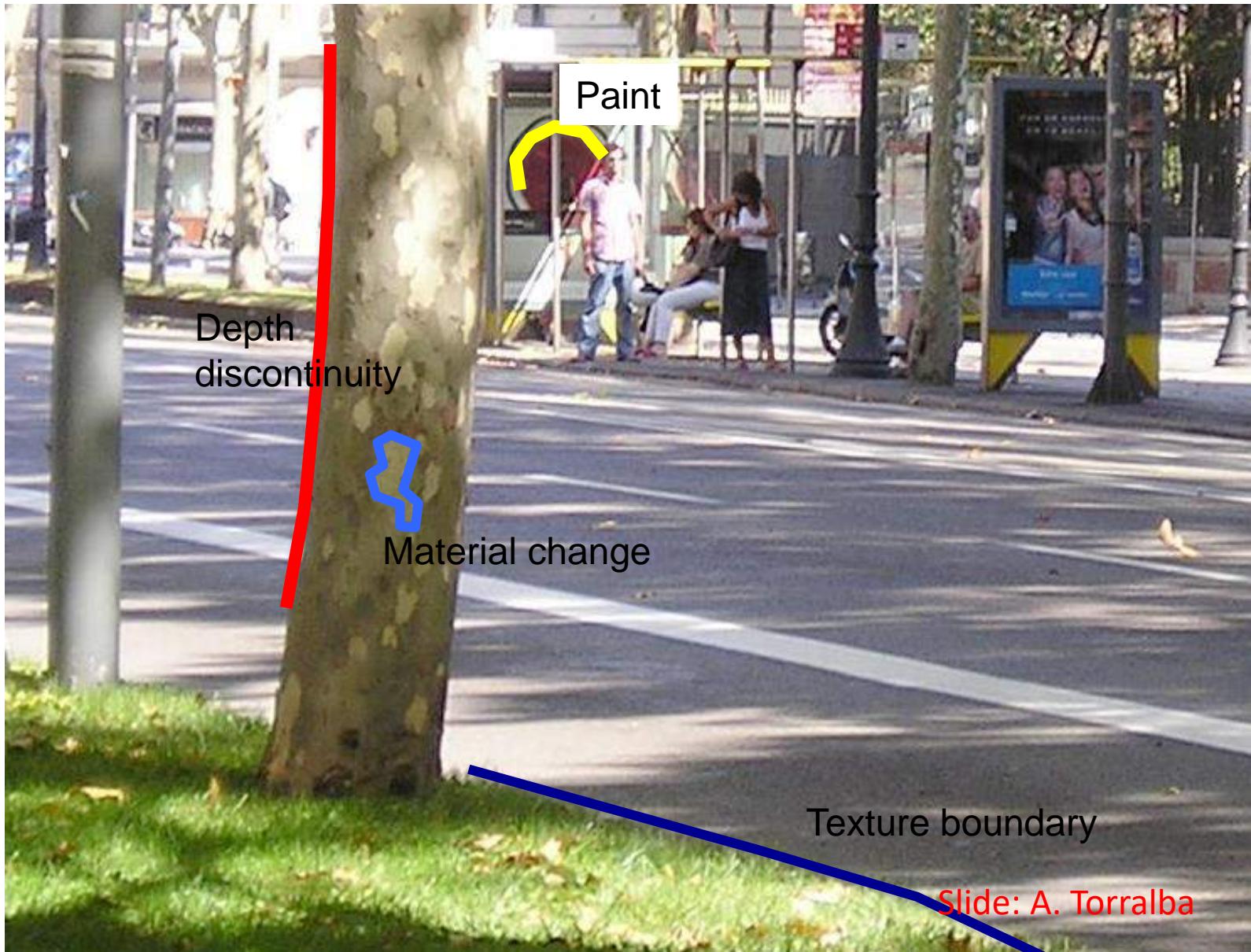
- 1) The gradient magnitude at different scales is different; which should we choose?
- 2) The gradient magnitude is large along thick trail; how do we identify the significant points?
- 3) How do we link the relevant points up into curves?
- 4) Noise.

The scale of the smoothing filter affects derivative estimates, and also the semantics of the edges recovered.

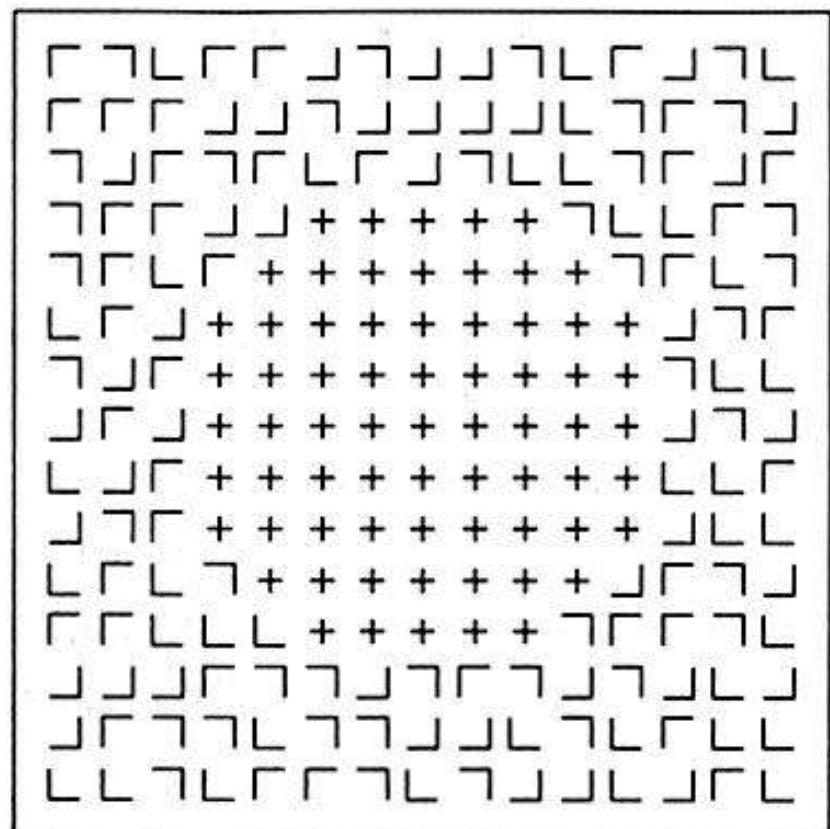
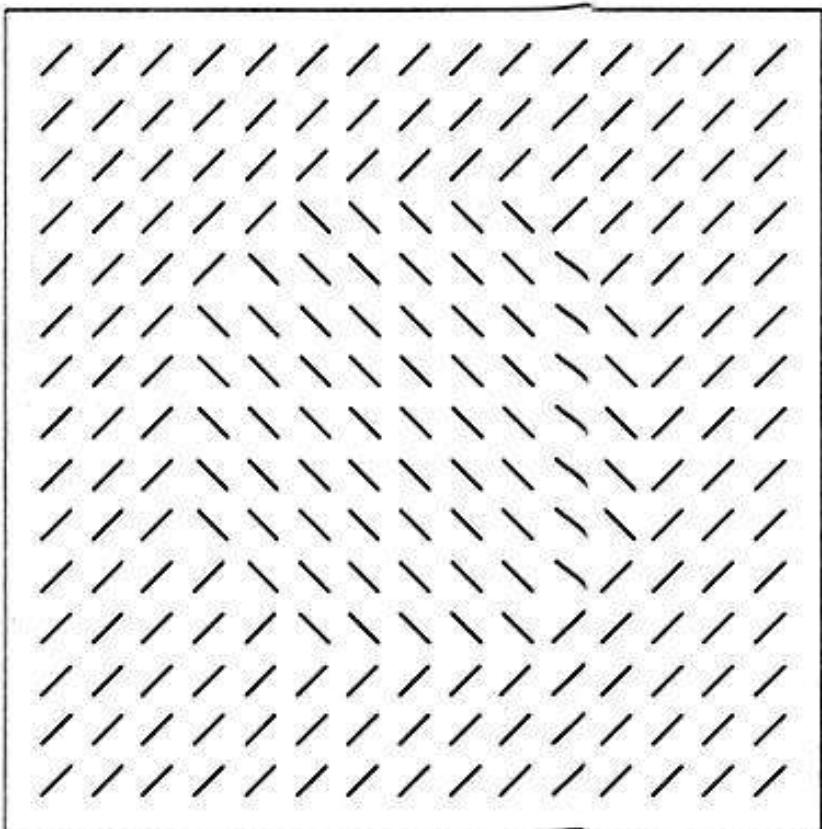
# What is an edge?



# What is an edge?



# Edges





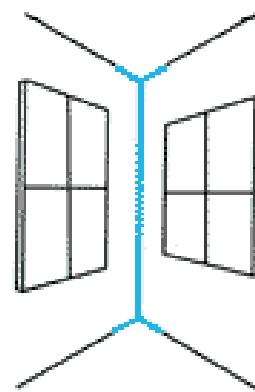
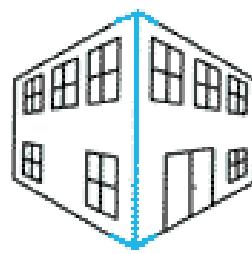
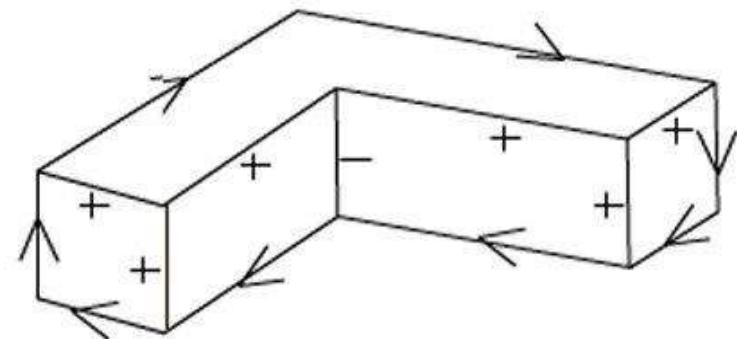
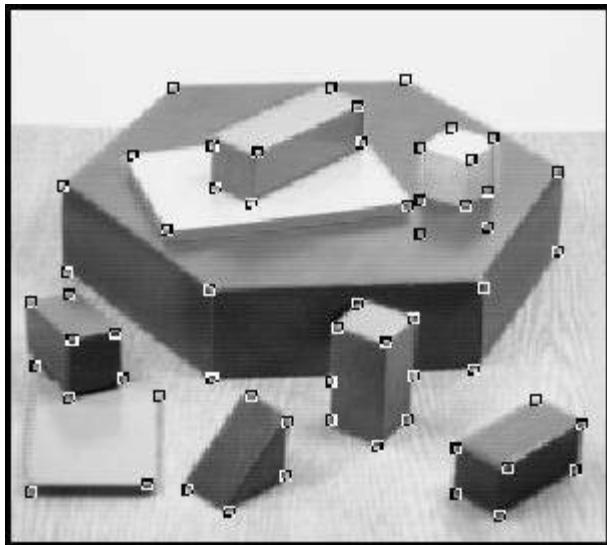
Gaussian gradient  
boundaries



Human boundaries

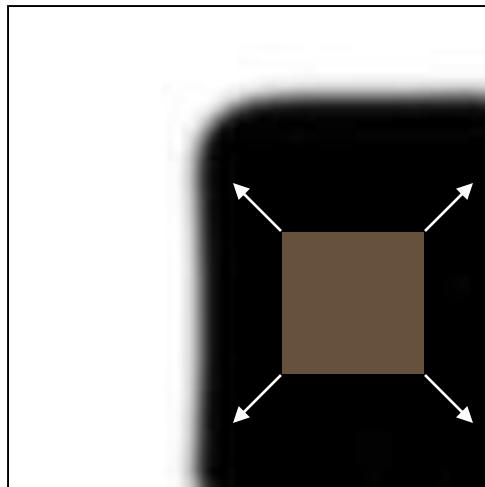
# Corners, Junctions

- Non-accidental features (Witkin & Tenenbaum, 1983)

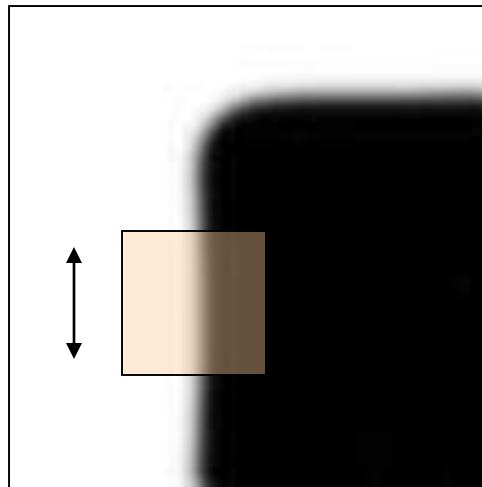


# Corners as distinctive interest points

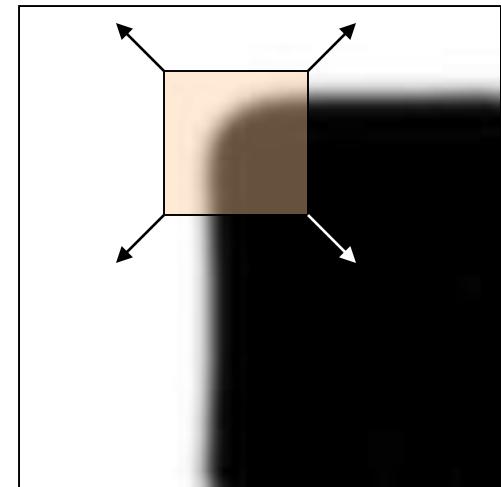
- Shifting a window in *any direction* should give *a large change* in intensity



“flat” region:  
no change in  
all directions



“edge”:  
no change along  
the edge  
direction



“corner”:  
significant  
change in all  
directions

# Harris Detector formulation

Change of intensity for the shift  $[u, v]$ :

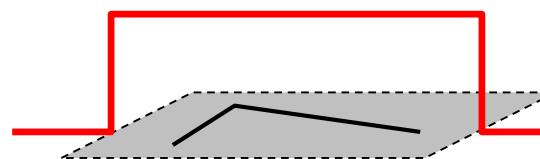
$$E(u, v) = \sum_{x, y} w(x, y) [I(x + u, y + v) - I(x, y)]^2$$

Diagram illustrating the components of the Harris detector formula:

- Window function (green oval)
- Shifted intensity (green oval)
- Intensity (green oval)

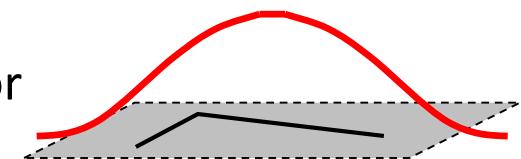
Arrows point from the labels to their corresponding terms in the equation.

Window function  $w(x, y) =$



1 in window, 0 outside

or



Gaussian

# Taylor Series for 2D Functions

$$f(x+u, y+v) = f(x, y) + u f_x(x, y) + v f_y(x, y) +$$

**First partial derivatives**

$$\frac{1}{2!} [u^2 f_{xx}(x, y) + uv f_{xy}(x, y) + v^2 f_{yy}(x, y)] +$$

**Second partial derivatives**

$$\frac{1}{3!} [u^3 f_{xxx}(x, y) + u^2 v f_{xxy}(x, y) + u v^2 f_{xyy}(x, y) + v^3 f_{yyy}(x, y)]$$

**Third partial derivatives**

+ ... (Higher order terms)

First order approx

$$f(x+u, y+v) \approx f(x, y) + u f_x(x, y) + v f_y(x, y)$$

## Harris Corner Derivation

$$\sum [I(x+u, y+v) - I(x, y)]^2$$

$$\approx \sum [I(x, y) + uI_x + vI_y - I(x, y)]^2 \quad \text{First order approx}$$

$$= \sum u^2 I_x^2 + 2uv I_x I_y + v^2 I_y^2$$

$$= \sum \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \quad \text{Rewrite as matrix equation}$$

$$= \begin{bmatrix} u & v \end{bmatrix} \left( \sum \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

# Harris Detector formulation

This measure of change can be approximated by:

$$E(u, v) \approx [u \ v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

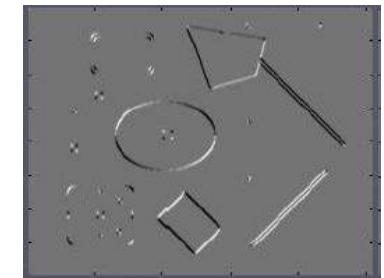
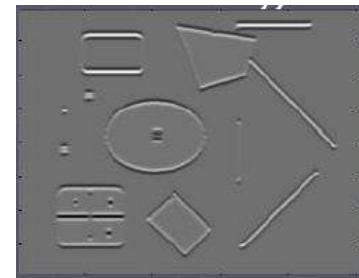
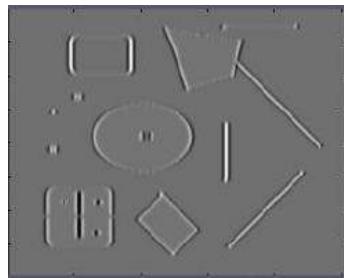
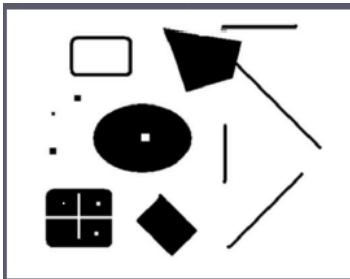
where  $M$  is a  $2 \times 2$  matrix computed from image derivatives:

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Sum over image region – area  
we are checking for corner

Gradient with  
respect to x, times  
gradient with  
respect to y

# Harris Detector formulation



where  $M$  is a  $2 \times 2$  matrix computed from image derivatives:

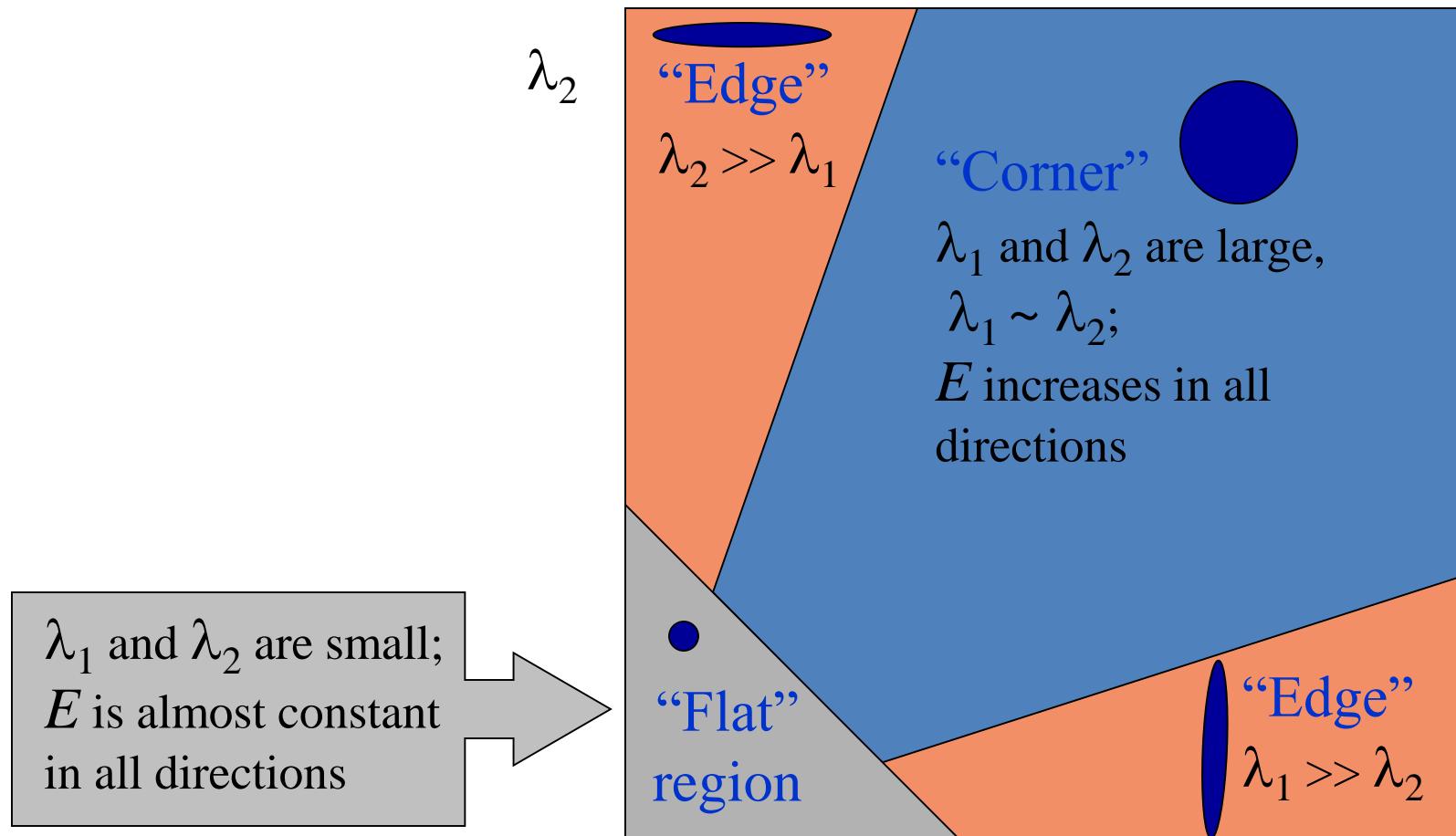
$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Sum over image region – area  
we are checking for corner

Gradient with  
respect to x, times  
gradient with  
respect to y

# Interpreting the eigenvalues of M

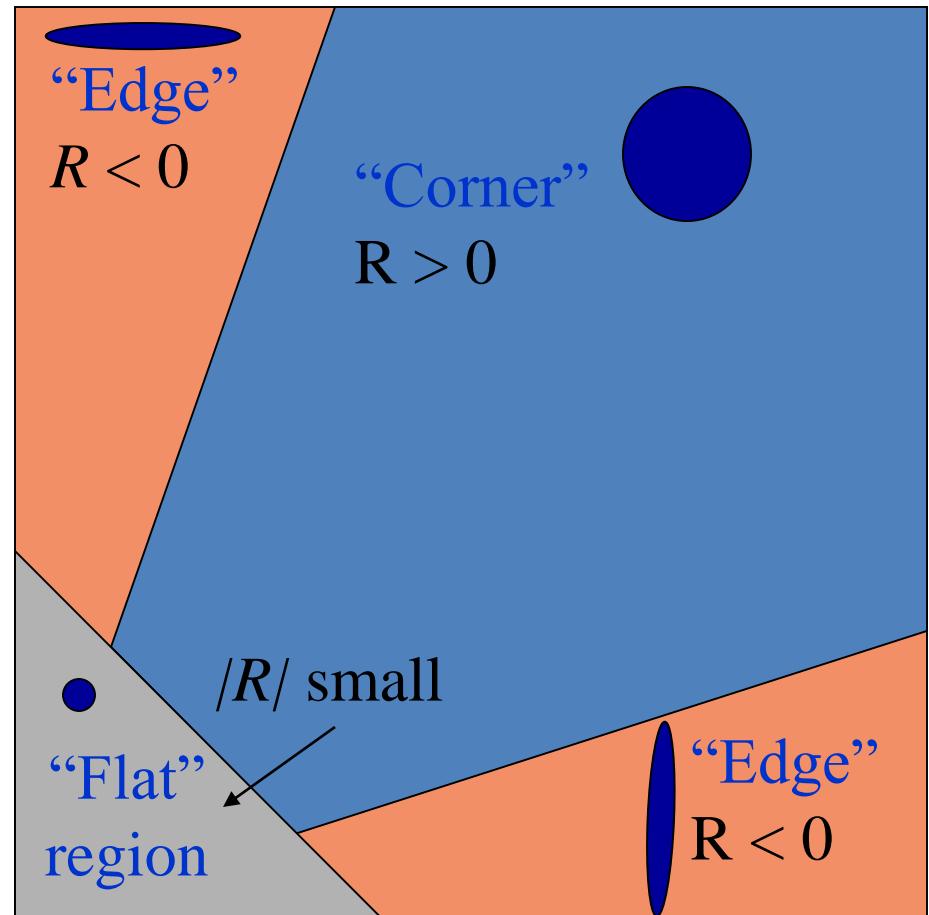
Classification of image points using eigenvalues of  $M$ :



# Corner response function

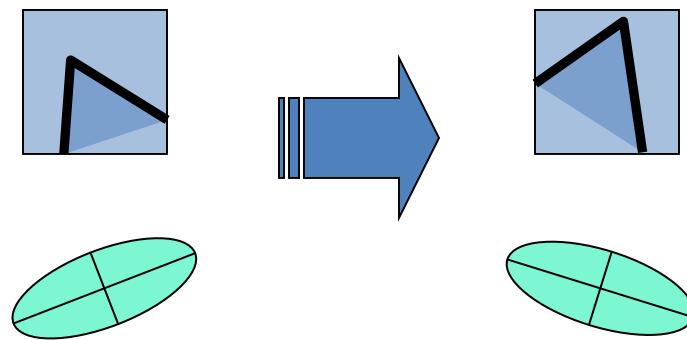
$$R = \det(M) - \alpha \operatorname{trace}(M)^2 = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$$

$\alpha$ : constant (0.04 to 0.06)



# Harris Detector: Properties

- Rotation invariance

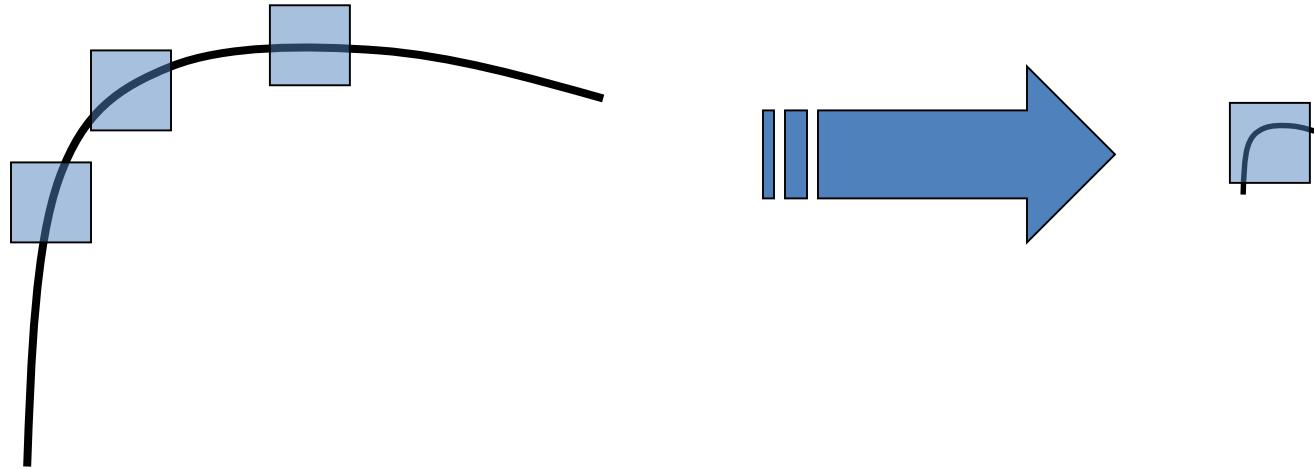


Ellipse rotates but its shape (i.e. eigenvalues) remains the same

*Corner response  $R$  is invariant to image rotation*

# Harris Detector: Properties

- Not invariant to image scale



All points will be  
classified as **edges**

Corner !

# Hessian Matrix

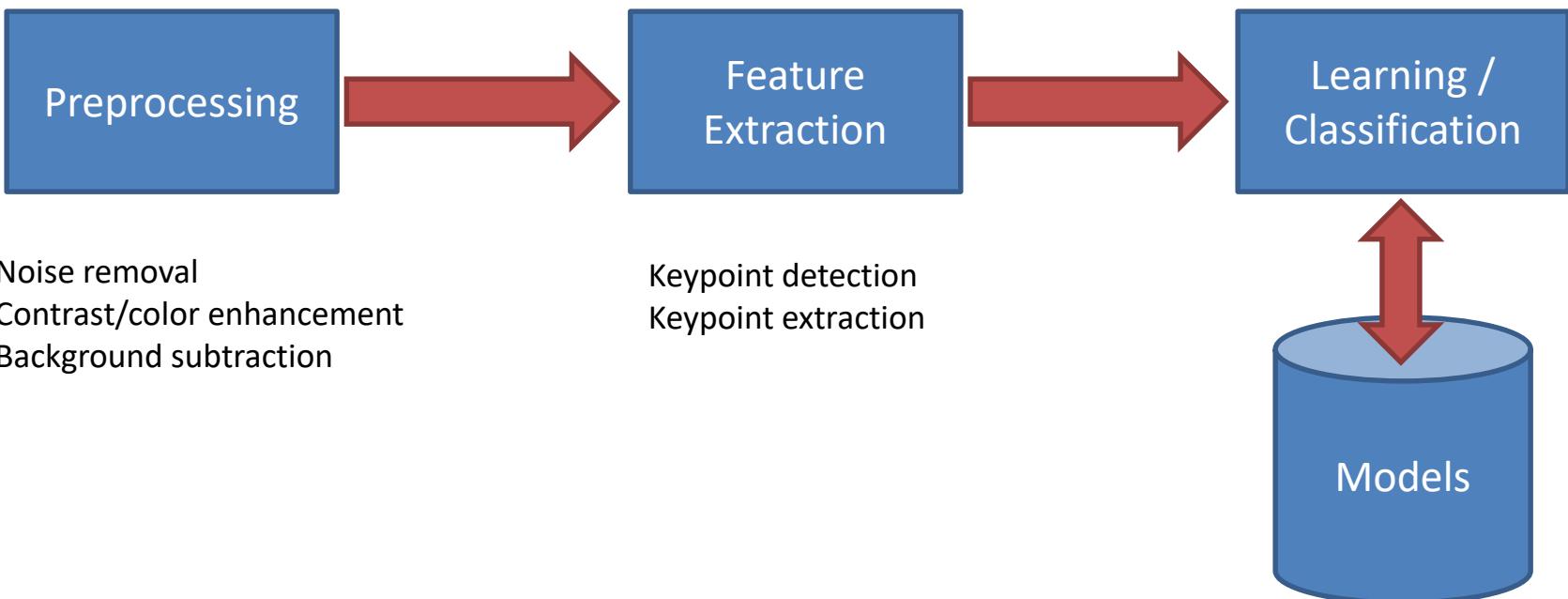
- The eigenvalues of the 2D case can be used for corner & edge detection.

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Harris:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

# Outline of approaches in Computer Vision & Pattern Recognition



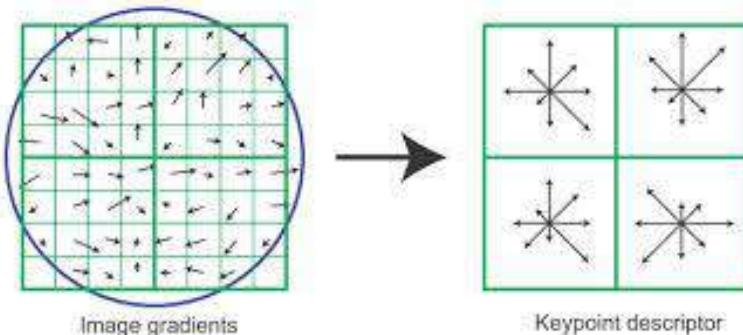
# **FEATURES WIDELY USED IN COMPUTER VISION**

# Feature Extraction: Overview

- Two phases:
  - Keypoint (interest point) detection
- Desired properties:
  - Repeatable
  - Scale invariant
  - Rotation invariant
  - Translation invariant



- Keypoint description



# Feature Extraction:

## Keypoint (interest point) detection

- What can be keypoints?
  - Edges
  - Corners
  - Segments / blobs
  - ...



(a) Color Labels (ACA)



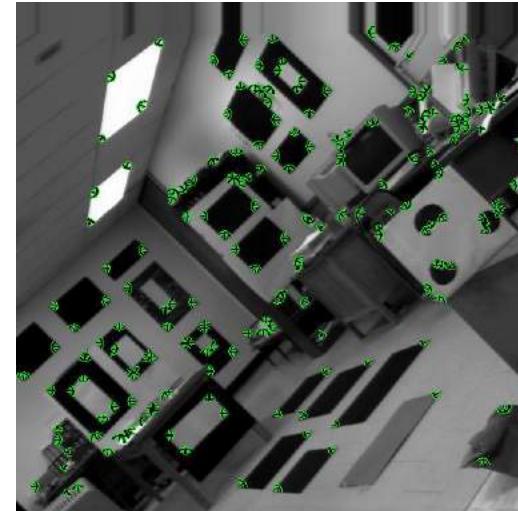
(b) Texture Classes



(c) Crude Segmentation



(d) Final Segmentation



An important issue: at what scale?

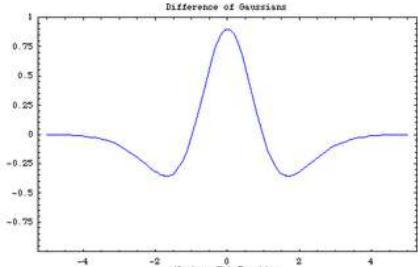
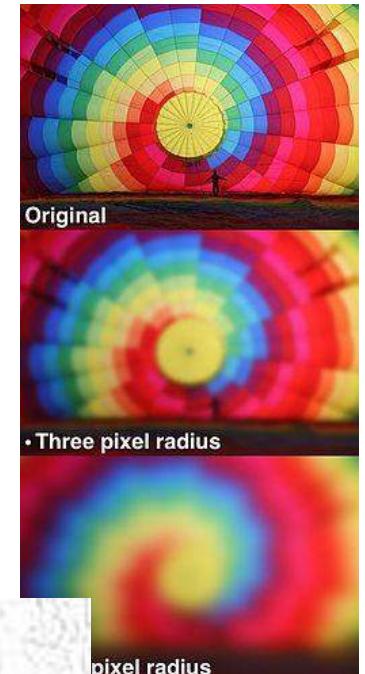
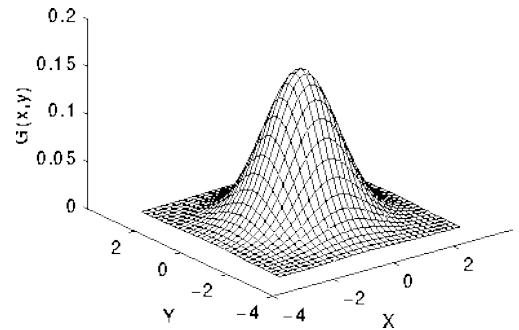
# Feature Extraction: Keypoint (interest point) detection

- Difference of Gaussians
  - Gaussian is a smoothing function:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- When a difference of Gaussians of different widths is applied, we get edge detection.

$$\Gamma_{\sigma_1, \sigma_2}(x) = I * \frac{1}{\sigma_1 \sqrt{2\pi}} e^{-(x^2)/(2\sigma_1^2)} - I * \frac{1}{\sigma_2 \sqrt{2\pi}} e^{-(x^2)/(2\sigma_2^2)}.$$



\*

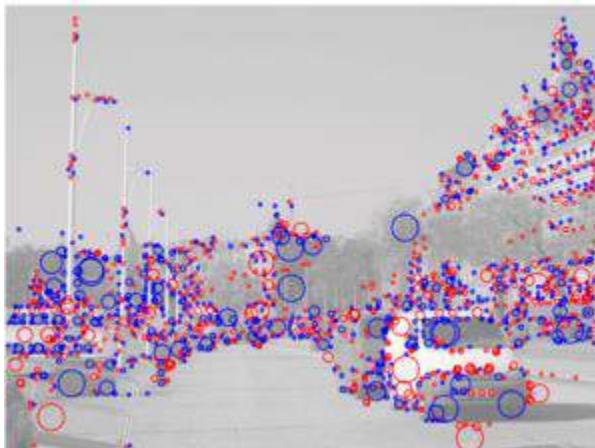
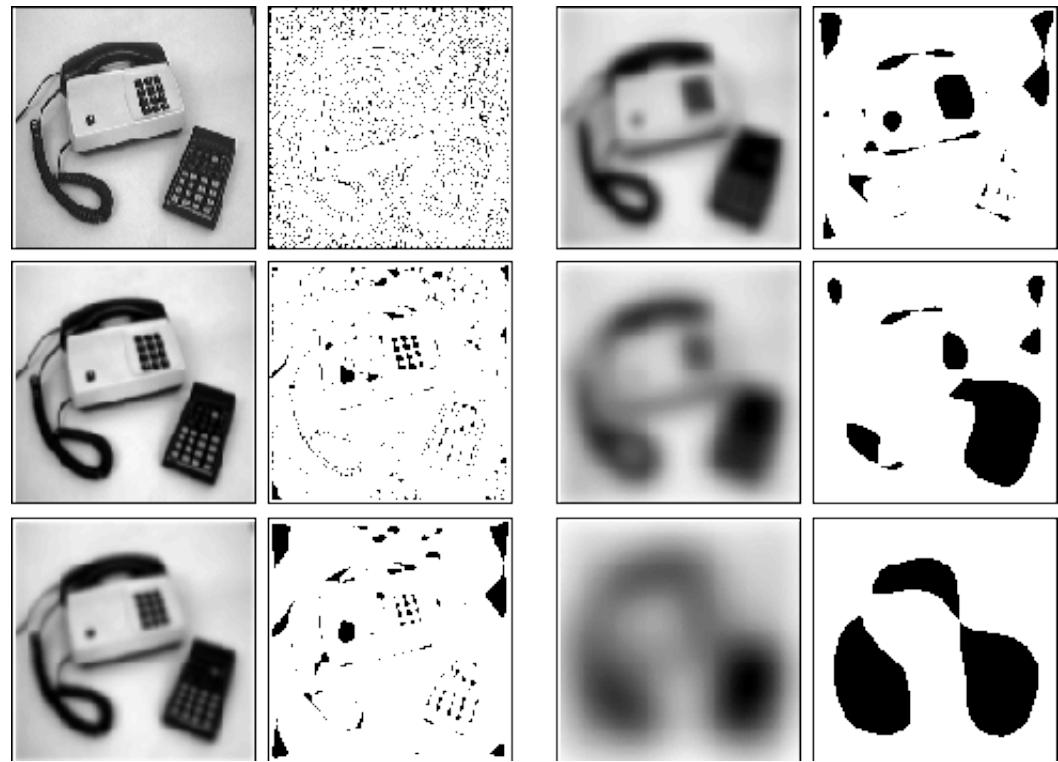


=



# Feature Extraction: Scale-space representation

- A single scale (width of the kernel) is not sufficient.
  - **Solution:** apply DoG or LoG at different scales, and find the maximum among scales as well as space.



# Feature Extraction: Keypoint description

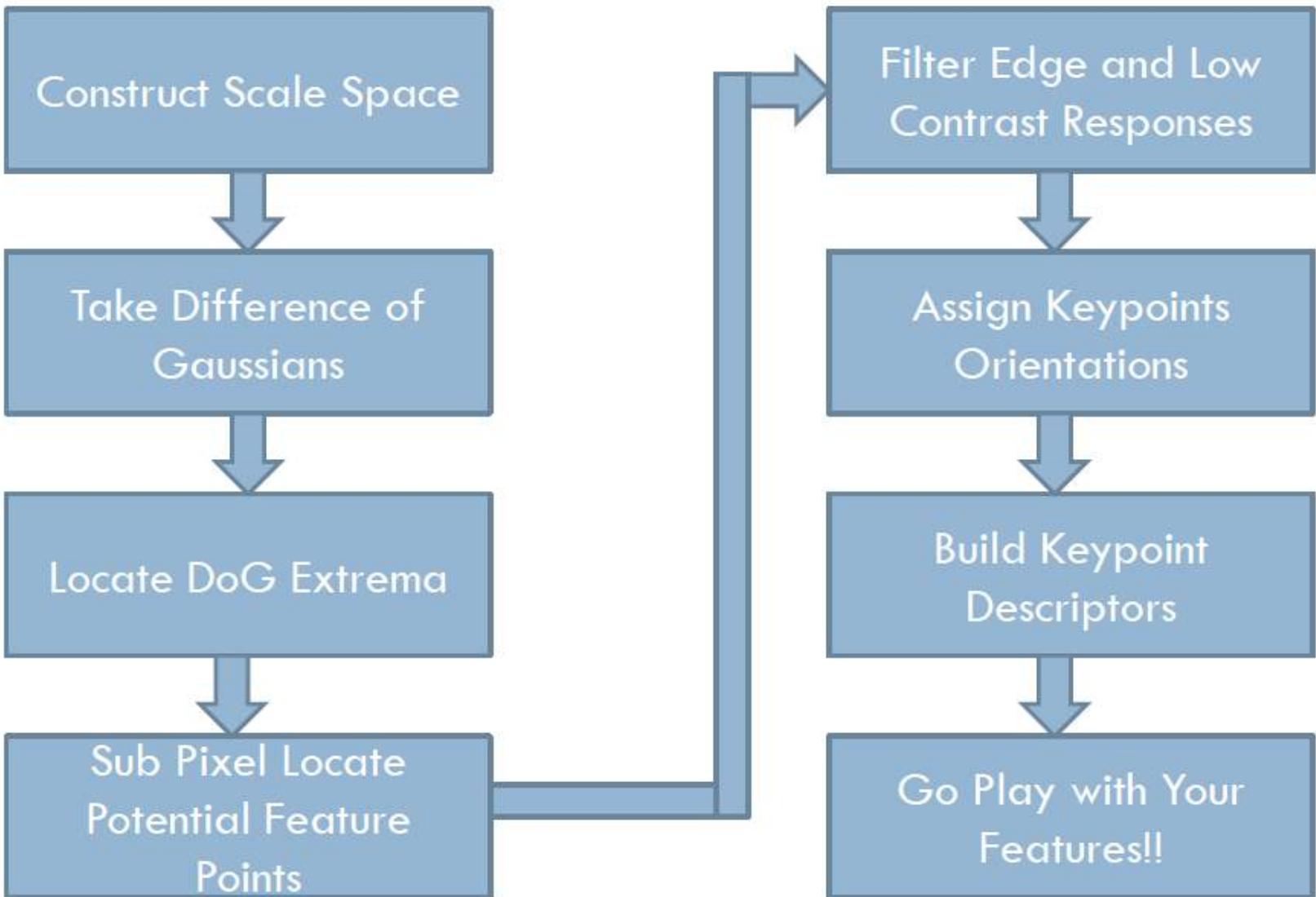
- Keypoint detection gives us “interesting point”s and a rough estimate of the scales (widths).
- We need to represent the information at these points:
  - Color
  - Texture
  - Gradient response
  - Image orientation
  - ...



# Feature Extraction: Keypoint descriptors

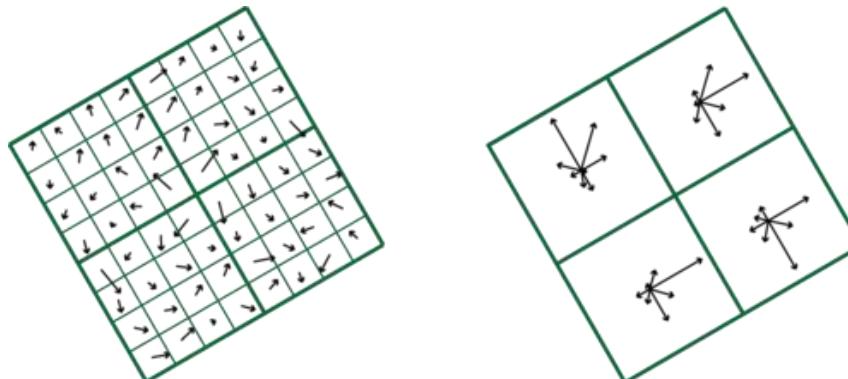
- Distribution-based
  - Scale-Invariant Feature Transform, Speeded-up Robust Features, ...
- Frequency or filter-based
  - Fourier Descriptors, Gabor Filters, ...
- Differential descriptors
  - Steerable filters, Histogram of Oriented Gradients, ...
- Moment-based
  - Zernike moments, ...
- Others
  - Local Binary Patterns
  - Global Binary Patterns

# Feature Extraction: Scale-Invariant Feature Transform



# Feature Extraction: Scale-Invariant Feature Transform (SIFT)

1. Scale gives the width of the interest point.
2. Compute peak orientation to determine the orientation of the interest region
  - Weighted by the gradient and Gaussian centered at the interest point
3. Divide the region into  $4 \times 4$  grids, and for each grid, compute local gradient orientation histogram (8-bin each)
4. This gives us a feature vector of size  $4 \times 4 \times 8 = 128$ .



Lowe, David G. (1999). Object recognition from local scale-invariant features. *ICCV*.

Lowe, David G. (2004). Distinctive image features from scale-invariant key points. *IJCV*.

# Feature Extraction: Speeded-up Robust Features (SURF)

1. Uses Hessian as the detector.
2. Scale-space is used similar to SIFT.
3. Orientation is estimated from a circular region around the interest point.
  - Gradient and orientation are extracted using a fast version of Haar wavelets.
4. The region is divided into a  $4 \times 4$  grid. In each grid,  $\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|$  are computed.
5. This gives us a feature vector of size  $4 \times 4 \times 4 = 64$ .

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix},$$

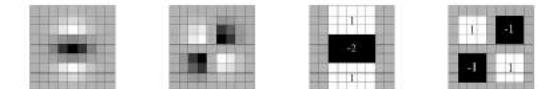
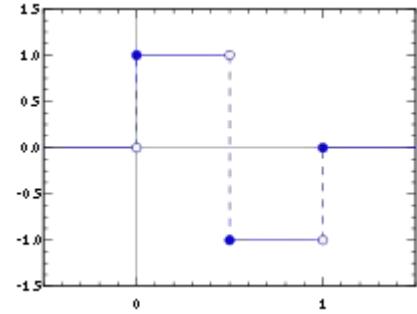
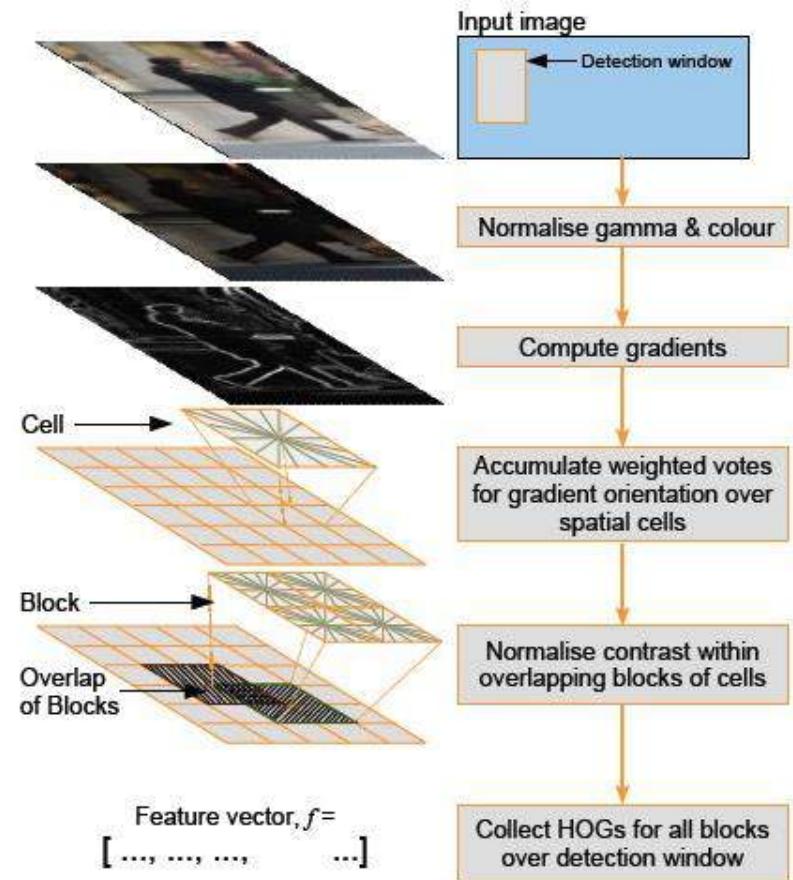


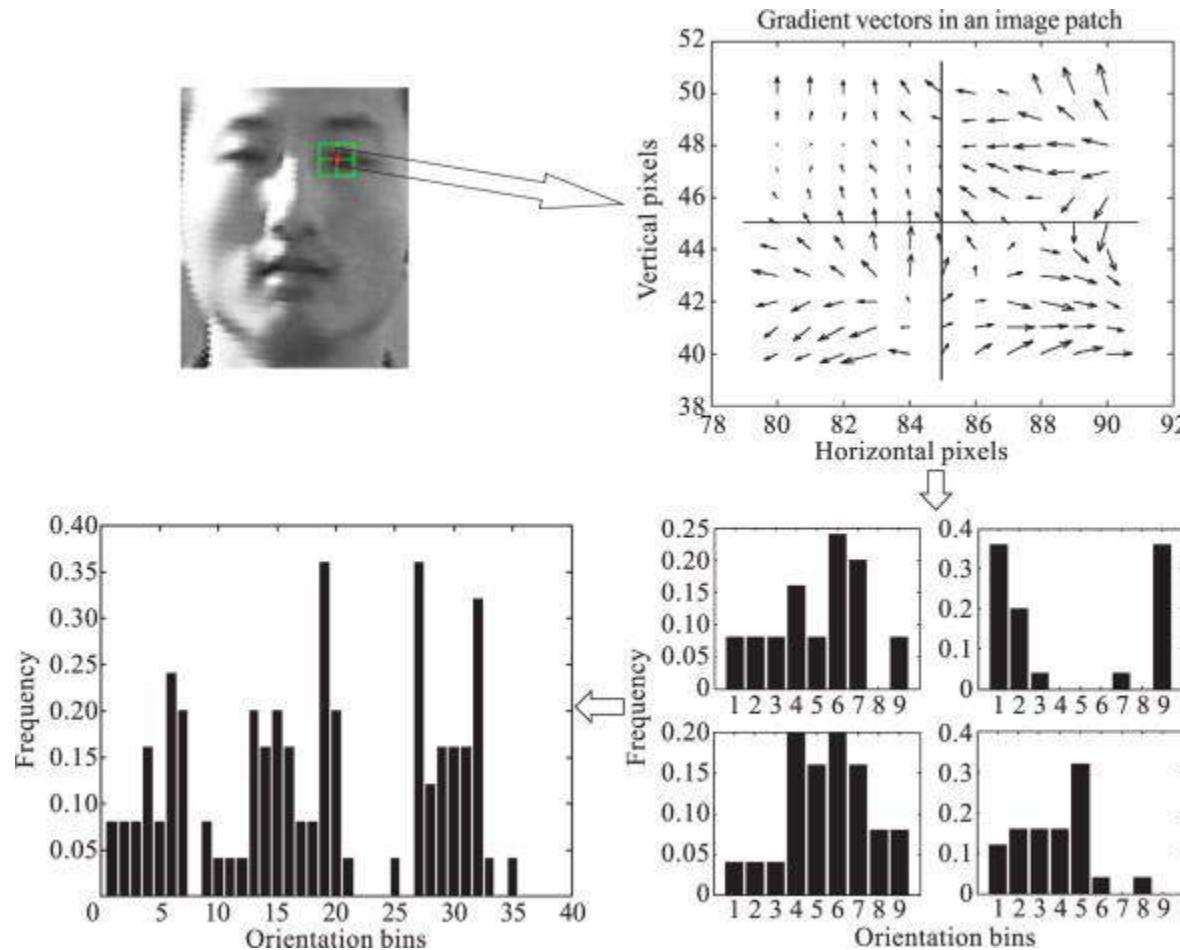
Fig. 1. Left to right: the (discretised and cropped) Gaussian second order partial derivatives in  $y$ -direction and  $xy$ -direction, and our approximations thereof using box filters. The grey regions are equal to zero.

# Feature Extraction: Histogram of Oriented Gradients (HOG)

- Dense feature extraction
  - The image is divided into windows.
- Each window is divided into blocks of cells.
  - In each cell, a histogram of orientations weighted by gradients is calculated.
- The concatenation of histograms is the feature vector.



# Feature Extraction: Histogram of Oriented Gradients (HOG)

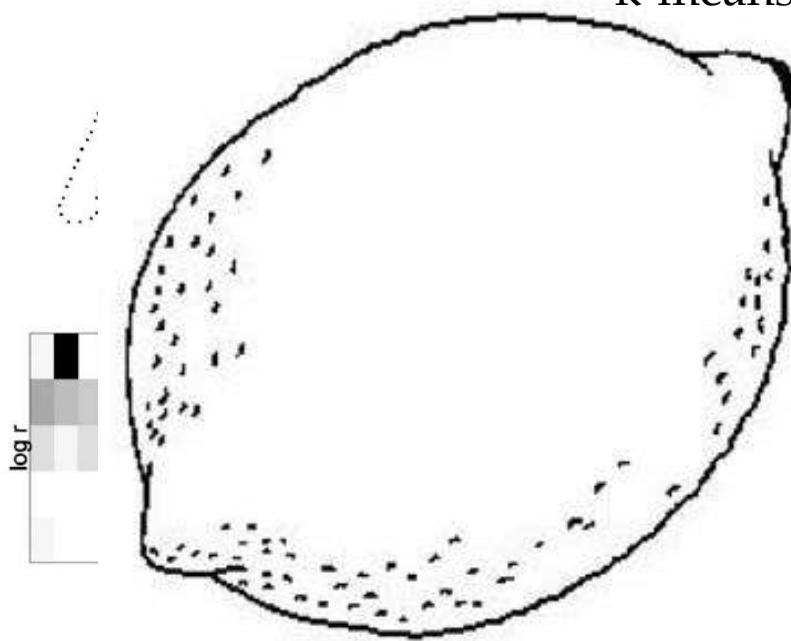


Shu et al., "Histogram of Oriented Gradients for Face Recognition", 2011.

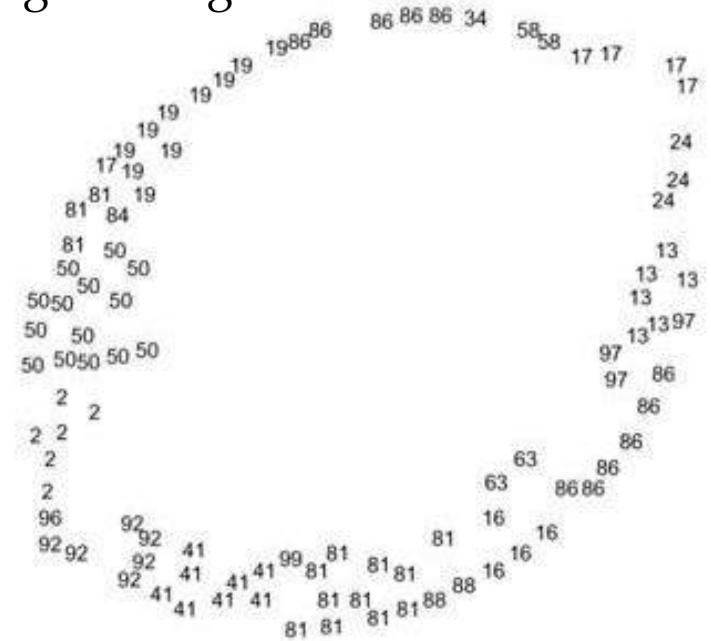
# Feature Extraction: Shapeme Histograms

- By Mori et al. (2005)

Shape Context histograms +  
k-means clustering + histogram

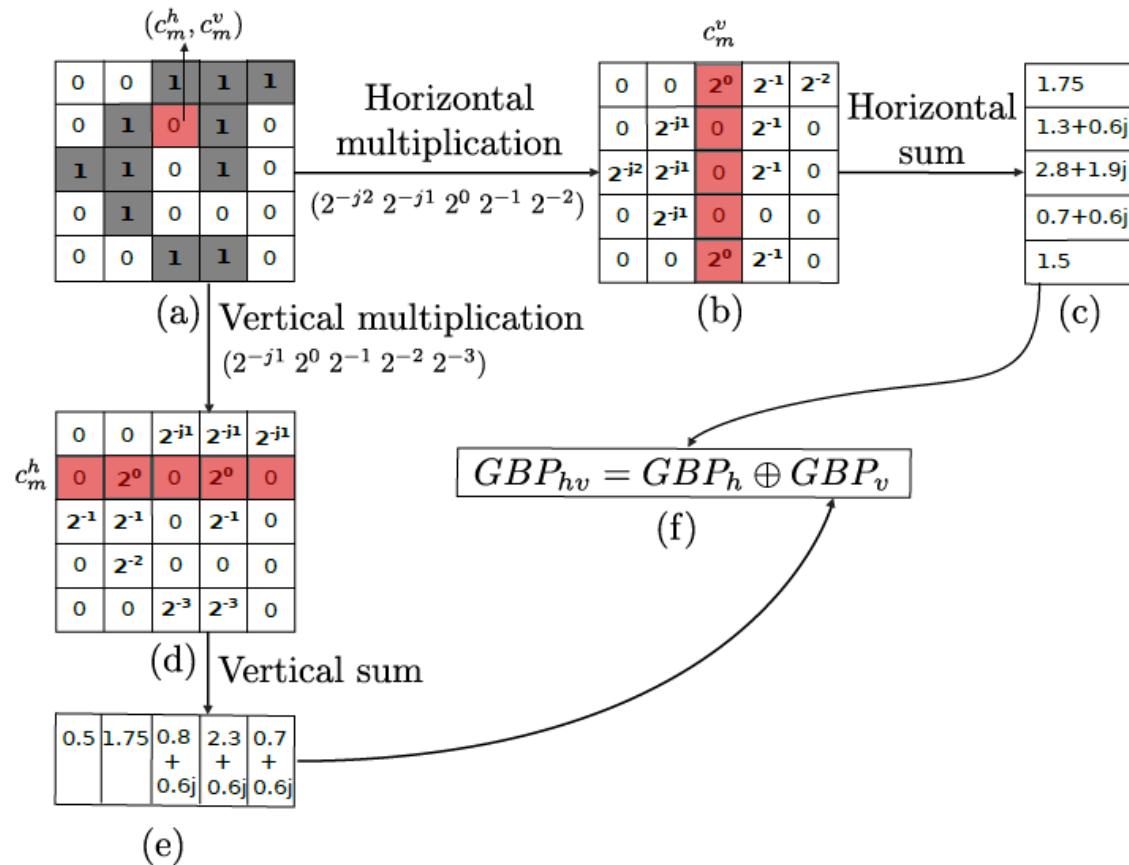


Shape Context (Belongi et al., 2002)



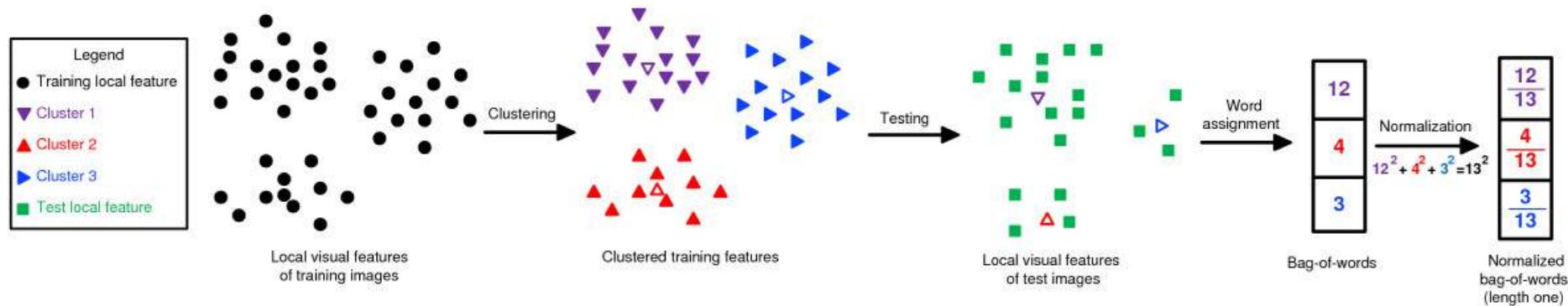
# Feature Extraction: Global Binary Patterns

- By Sivri & Kalkan (2013).



# Feature Extraction: Bag of Words

- Instead of using features directly, first group them into categories (“words”) and then use those words.
  - More efficient
  - Better performance



# **EARLY VISION**

# Sensory Coding

- Efficient Coding Hypothesis
  - “The goal of early vision (or, early visual processing) is to provide an efficient representation of the incoming visual signal”
  - (Field, 1987; Hateren, 1998; Bell & Sejnowski, 1996, 1997; Olshausen & Field, 1996; Hyvarinen, 2010)
- For a review & critics:
  - (Simoncelli & Olshausen, 2001; Simoncelli, 2003)

Independent Component Analysis:

$$\text{Image} = s_1 \cdot \text{Filter}_1 + s_2 \cdot \text{Filter}_2 + \dots + s_k \cdot \text{Filter}_k$$

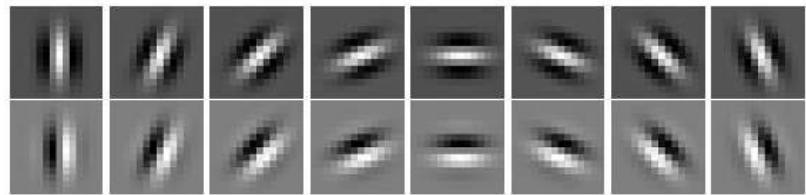
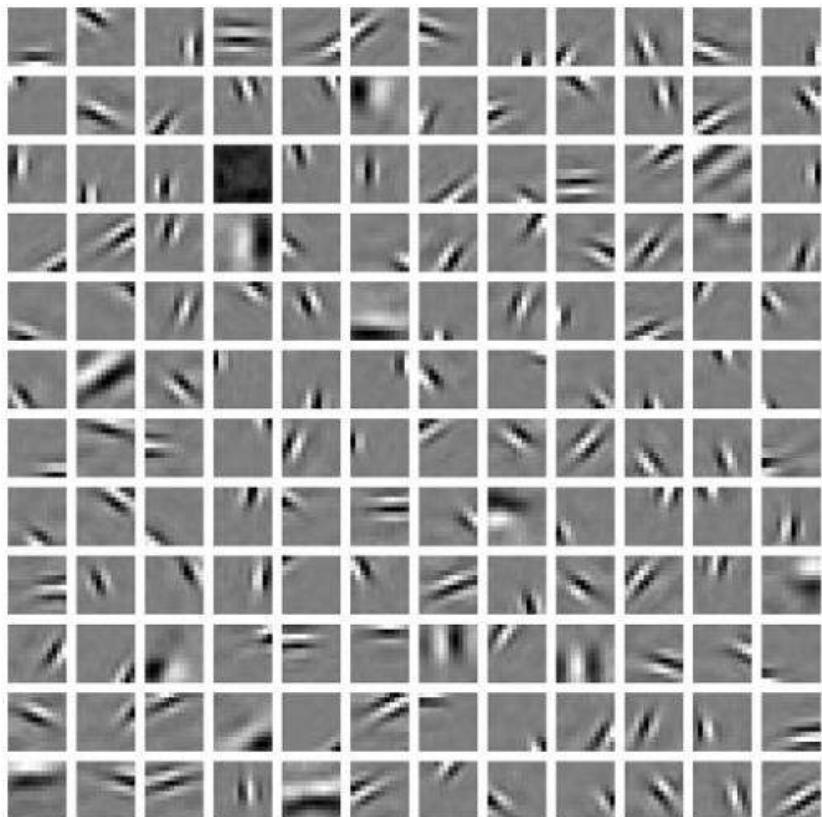


Figure 3: Real (first row) and imaginary (second row) parts of eight orientation Gabor wavelets.  
(Kalkan et al., 2008)



(Olshausen & Field, 1996)

(Hyvarinen, 2010)

# Gabor Filter

- Gaussian multiplied by a sinusoidal.

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \exp\left(i\left(2\pi\frac{x'}{\lambda} + \psi\right)\right)$$

Real

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi\frac{x'}{\lambda} + \psi\right)$$

Imaginary

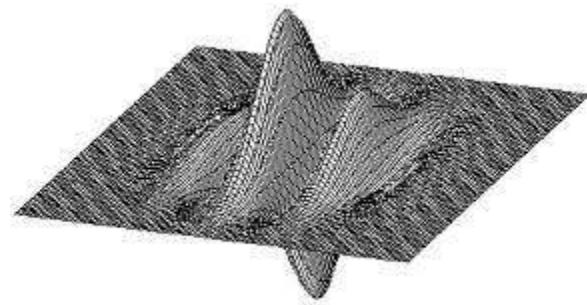
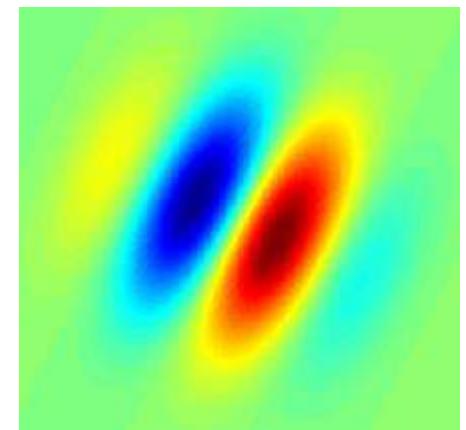
$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \sin\left(2\pi\frac{x'}{\lambda} + \psi\right)$$

where

$$x' = x \cos \theta + y \sin \theta$$

and

$$y' = -x \sin \theta + y \cos \theta$$



# Gabor Filters vs. Cortical Receptive Fields

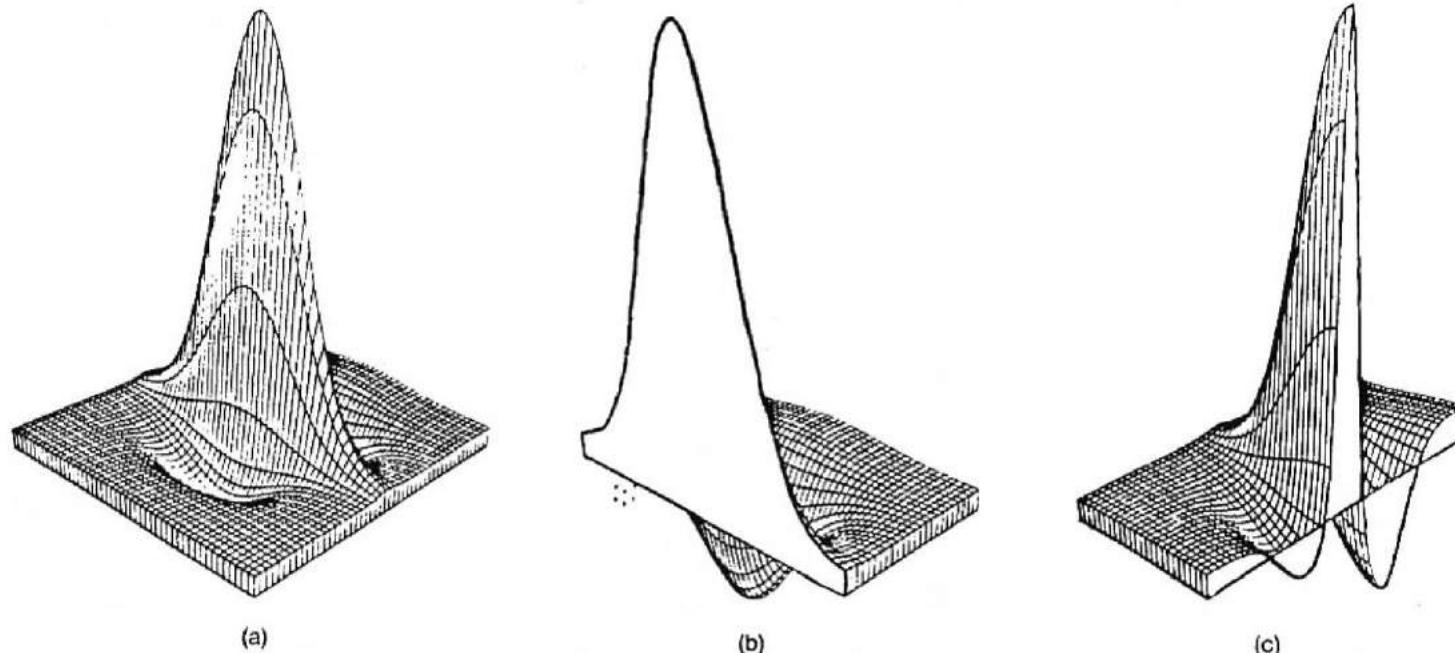
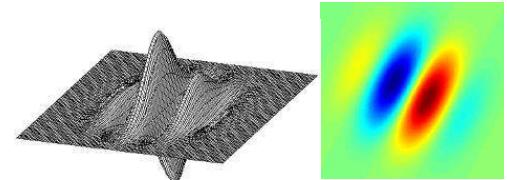


Fig. 1. Perspective plot of the two-dimensional profile of sensitivity of the model of the receptive field employed in this analysis. (a) The whole shape. (b) A cross section in the axis parallel to the preferred orientation of the receptive field, which highlights the Gaussian distribution used to describe the height ( $y_h$ ) dimension. (c) A cross section in the dimension orthogonal to the preferred orientation, which highlights the d-DOG-S function used to model the structure of the excitatory and inhibitory zones in the receptive field.

Reprinted from Journal of the Optical Society of America A, Vol. 5, page 598, April 1988  
Copyright © 1988 by the Optical Society of America and reprinted by permission of the copyright owner.

Two-dimensional spatial structure of receptive fields in  
monkey striate cortex

# Gabor Filters vs. Cortical Receptive Fields

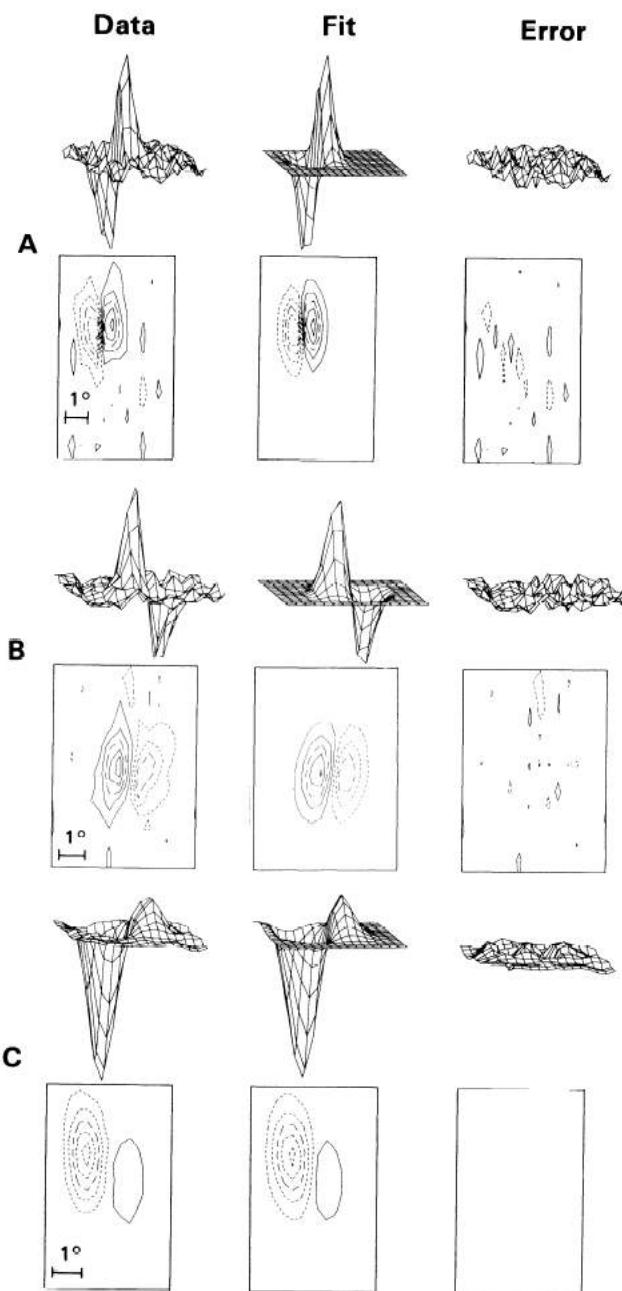


FIG. 2. The two-dimensional (2D) Gabor filter fits simple cell 2D spatial response profiles. Each part of this figure illustrates a 2D spatial response profile, the corresponding least-squared error best-fitting 2D Gabor filter, and the residual error, that function of space which remains after the 2D Gabor filter has been subtracted from the data.

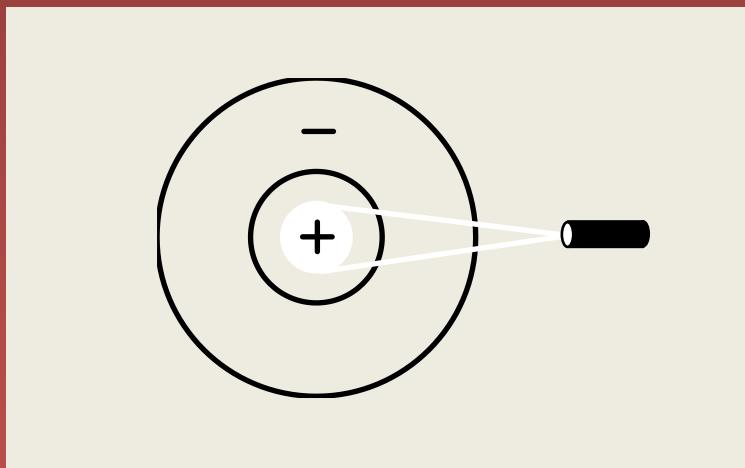
JOURNAL OF NEUROPHYSIOLOGY  
Vol. 58, No. 6, December 1987. Printed in U.S.A.

An Evaluation of the Two-Dimensional Gabor Filter  
Model of Simple Receptive Fields in Cat Striate Cortex

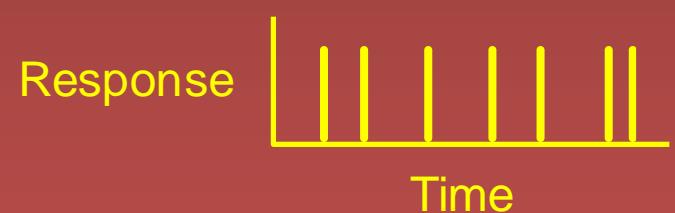
JUDSON P. JONES AND LARRY A. PALMER

# Retinal Receptive Fields

Receptive field structure in ganglion cells:  
On-center Off-surround



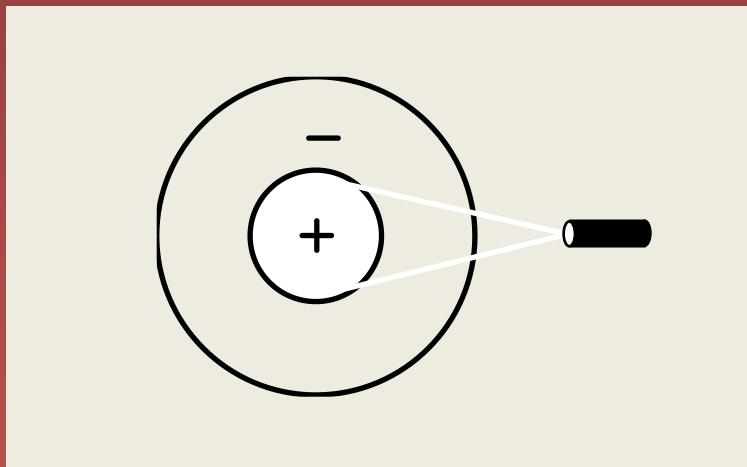
Stimulus condition



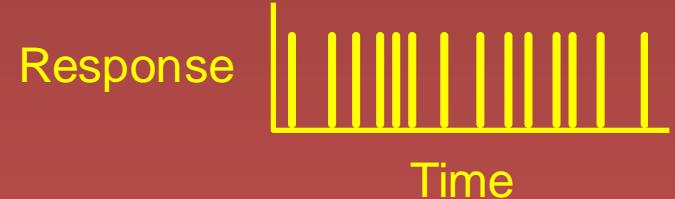
Electrical response

# Retinal Receptive Fields

Receptive field structure in ganglion cells:  
On-center Off-surround



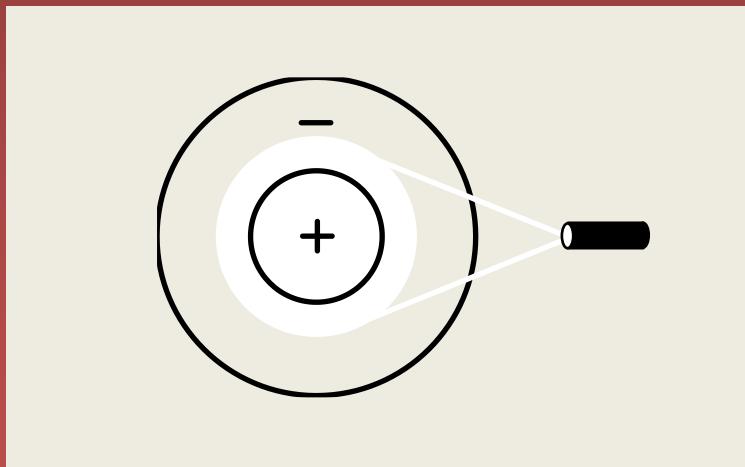
Stimulus condition



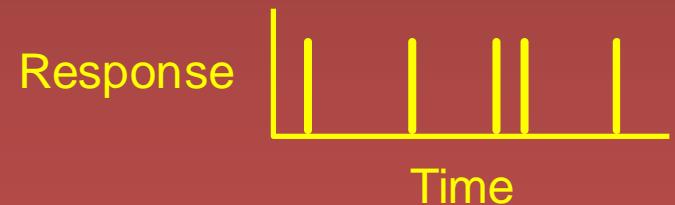
Electrical response

# Retinal Receptive Fields

Receptive field structure in ganglion cells:  
On-center Off-surround



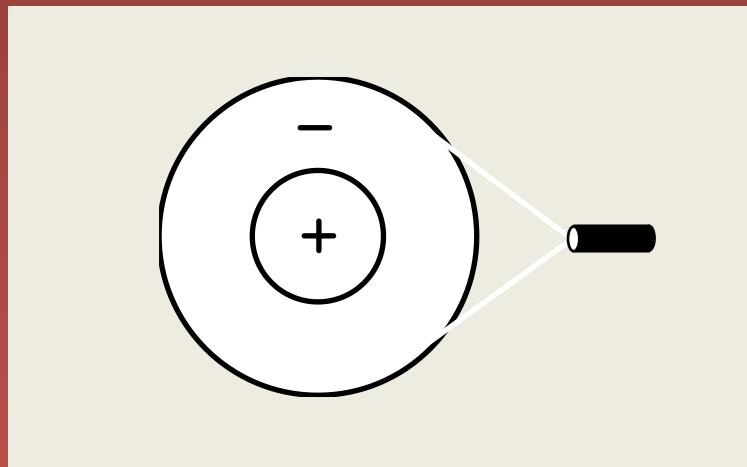
Stimulus condition



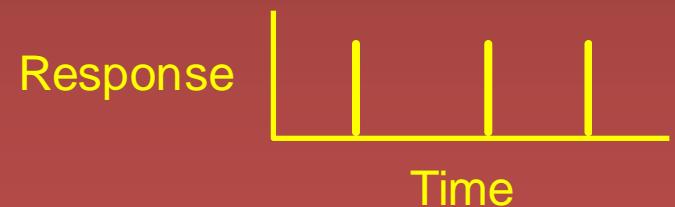
Electrical response

# Retinal Receptive Fields

Receptive field structure in ganglion cells:  
On-center Off-surround



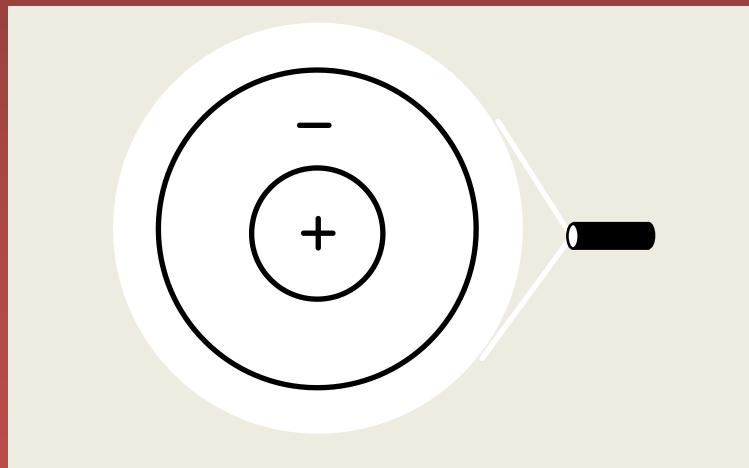
Stimulus condition



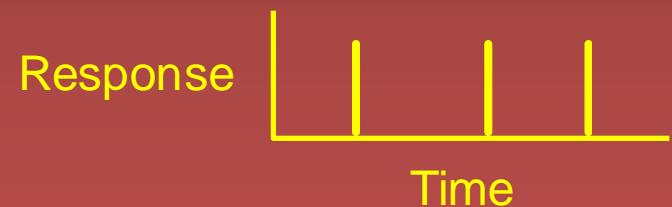
Electrical response

# Retinal Receptive Fields

Receptive field structure in ganglion cells:  
On-center Off-surround



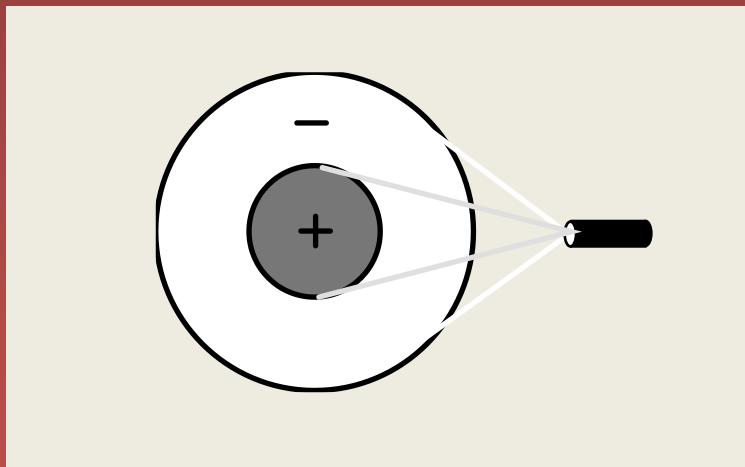
Stimulus condition



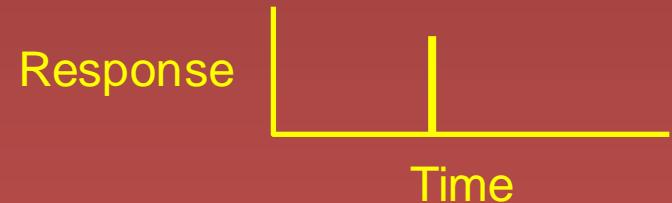
Electrical response

# Retinal Receptive Fields

Receptive field structure in ganglion cells:  
On-center Off-surround



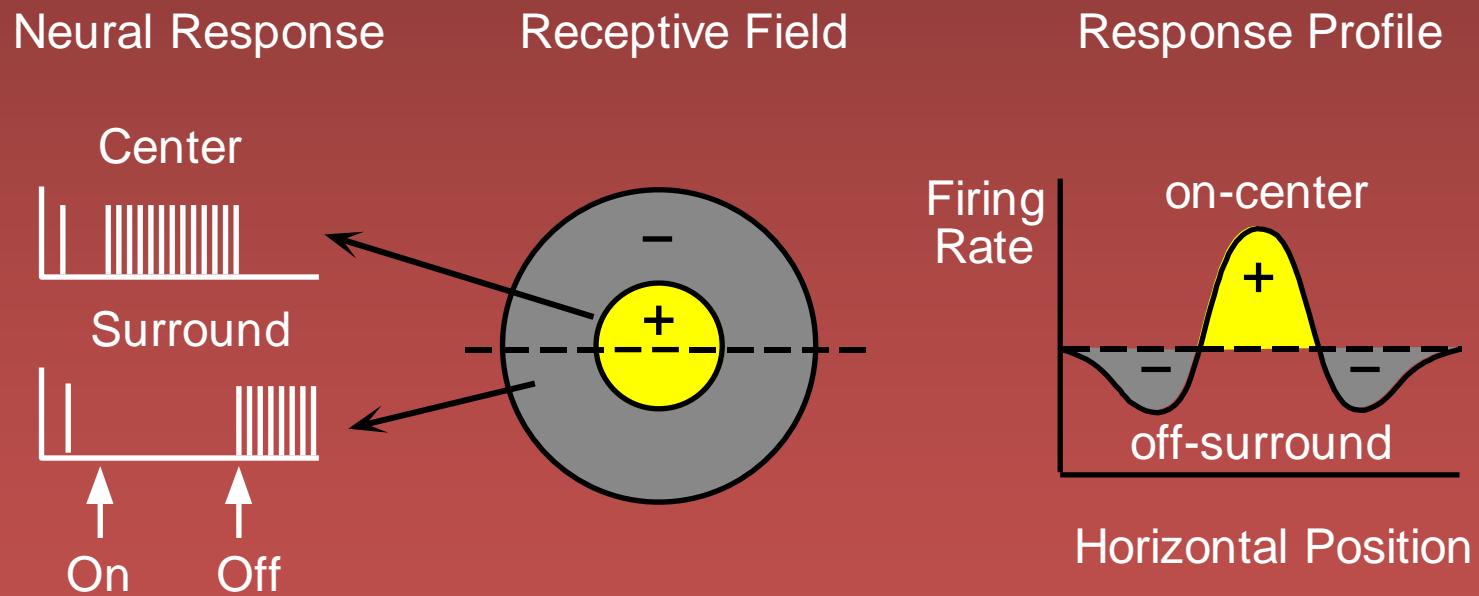
Stimulus condition



Electrical response

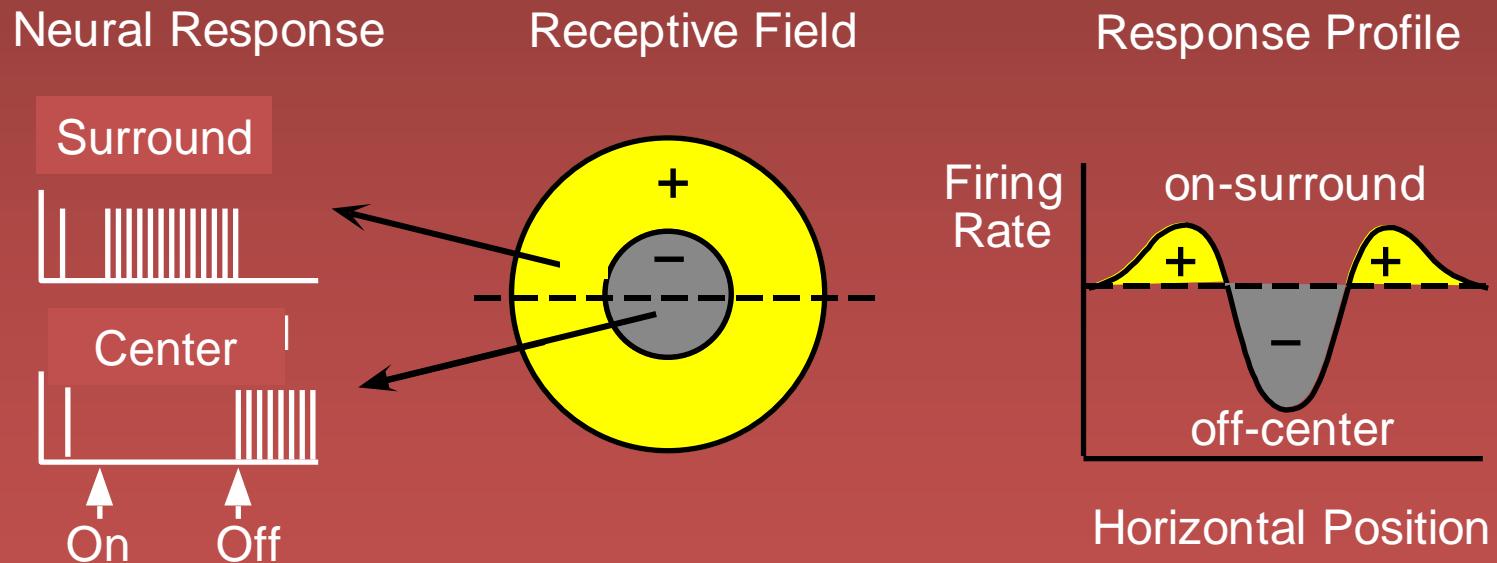
# Retinal Receptive Fields

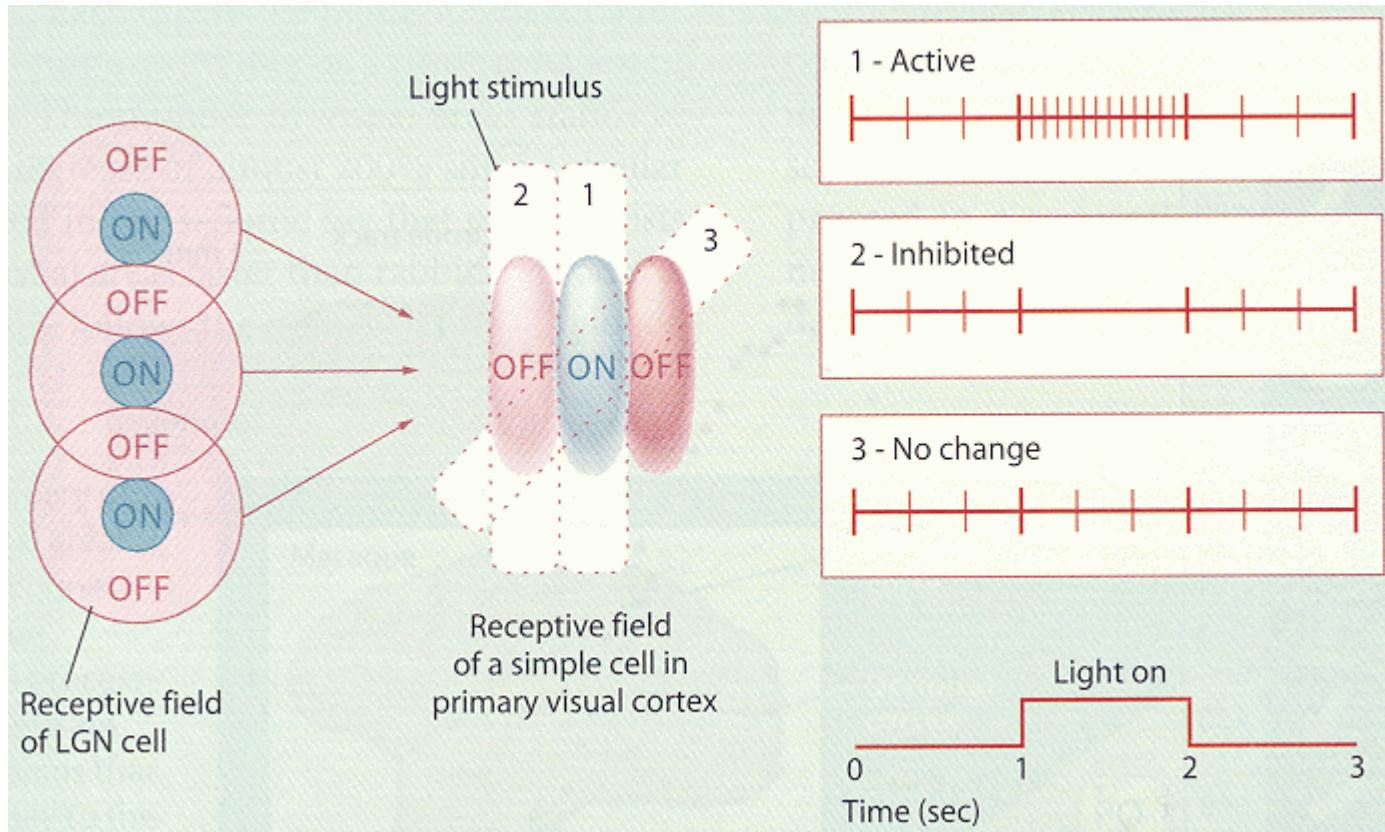
## RF of On-center Off-surround cells



# Retinal Receptive Fields

## RF of Off-center On-surround cells

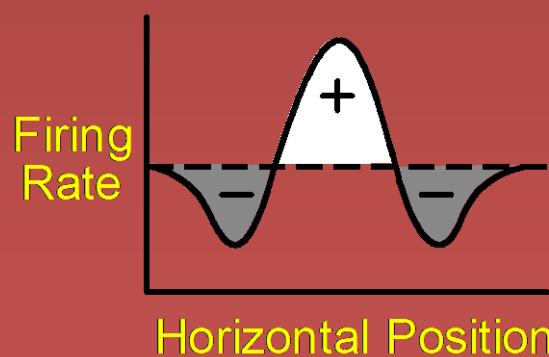
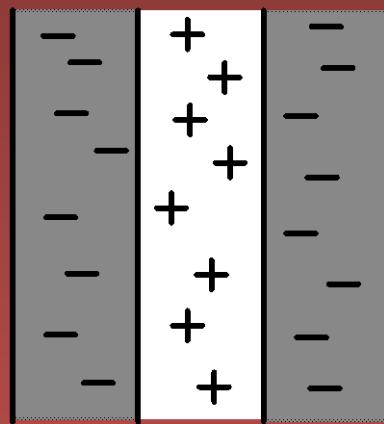




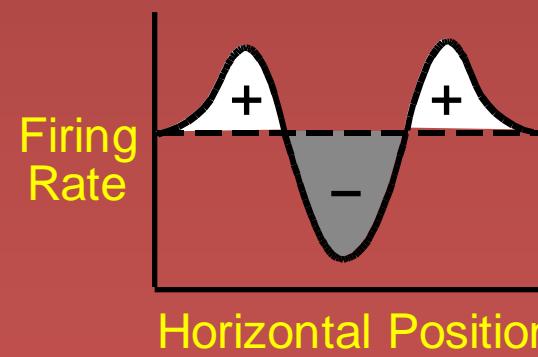
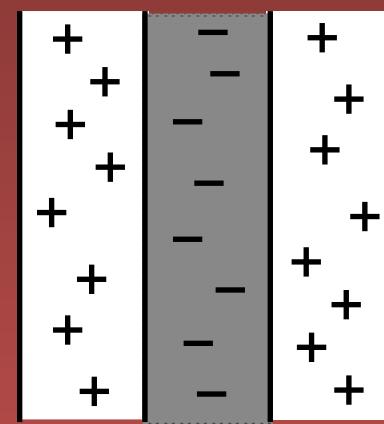
# Cortical Receptive Fields

## Simple Cells: “Line Detectors”

A. Light Line Detector



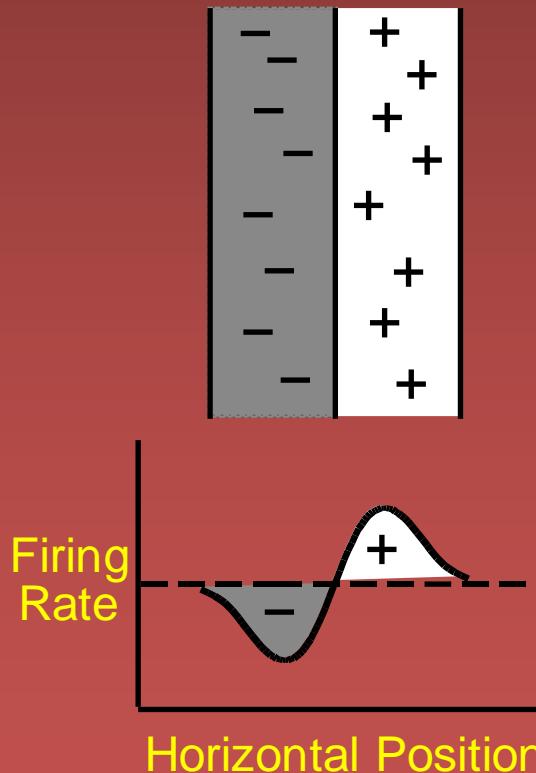
B. Dark Line Detector



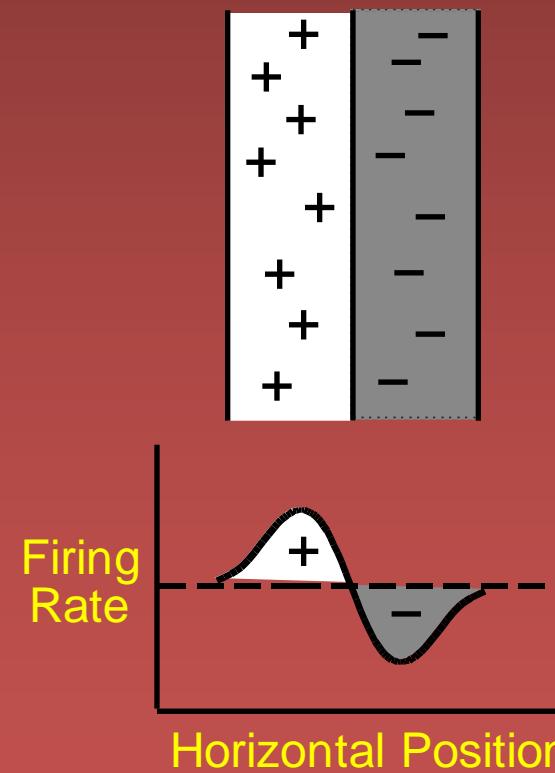
# Cortical Receptive Fields

## Simple Cells: “Edge Detectors”

C. Dark-to-light Edge Detector



D. Light-to-dark Edge Detector



# So, what does it say?

- The early ‘low-level filtering’ achieves efficient encoding by finding intensity changes.
  - With Gabor-like filters.
- This is one of the important evidence for the hypothesis that our visual system is tuned to the statistical regularities in the environment.

(Slides are mostly from Norbert Krueger)

# **HUMAN VISION IN A NUTSHELL**

IEEE Trans Pattern Anal Mach Intell. 2013 Aug; 35(8):1847-71.

# Deep Hierarchies in the Primate Visual Cortex: What Can We Learn For Computer Vision?

Norbert Krüger, Peter Janssen, Sinan Kalkan, Markus Lappe, Aleš Leonardis, Justus Piater,  
Antonio J. Rodriguez-Sánchez, Laurenz Wiskott

**Abstract**—Computational modeling of the primate visual system yields insights of potential relevance to some of the challenges that computer vision is facing, such as object recognition and categorization, motion detection and activity recognition or vision-based navigation and manipulation. This article reviews some functional principles and structures that are generally thought to underlie the primate visual cortex, and attempts to extract biological principles that could further advance computer vision research. Organized for a computer vision audience, we present *functional principles* of the *processing hierarchies* present in the primate visual system considering recent discoveries in neurophysiology. The hierarchical processing in the primate visual system is characterized by a sequence of different levels of processing (in the order of ten) that constitute a *deep hierarchy* in contrast to the *flat* vision architectures predominantly used in today's mainstream computer vision. We hope that the functional description of the deep hierarchies realized in the primate visual system provides valuable insights for the design of computer vision algorithms, fostering increasingly productive interaction between biological and computer vision research.

**Index Terms**—Computer Vision, Deep Hierarchies, Biological Modeling

## 1 INTRODUCTION

The history of computer vision now spans more than half a century. However, general, robust, complete satisfactory solutions to the major problems such as large-scale object, scene and activity recognition and categorization, as well as vision-based manipulation are still beyond reach of current machine vision systems. Biological visual systems, in particular those of primates, seem to accomplish these tasks almost effortlessly and have been, therefore, often used as an inspiration for computer vision researchers.

Interactions between the disciplines of “biological vision” and “computer vision” have varied in intensity throughout

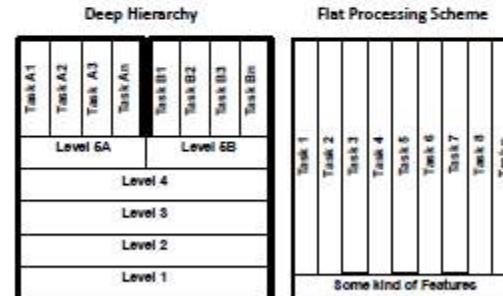


Fig. 1. Deep hierarchies and flat processing schemes

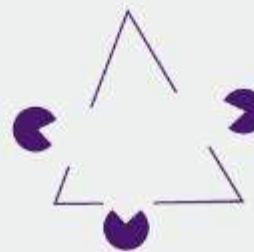
The Mærsk McKinney Møller Institute

Denmark

University of Southern

Copyrighted material

# VISION



David Marr

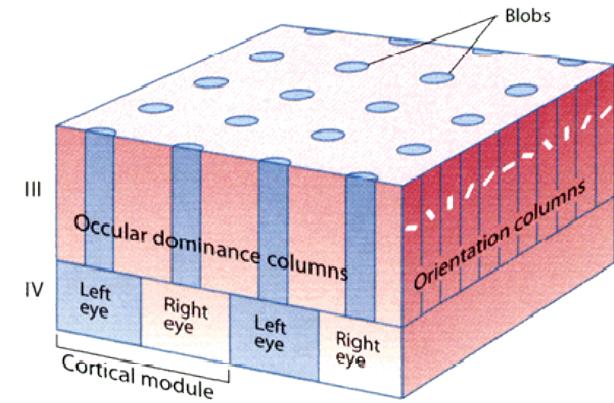
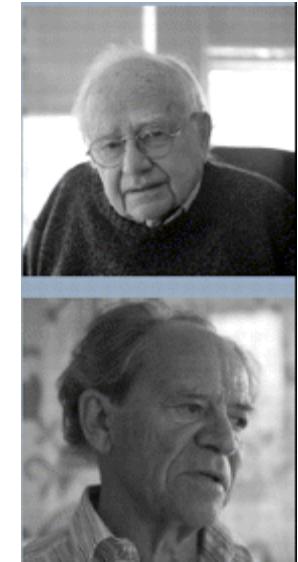
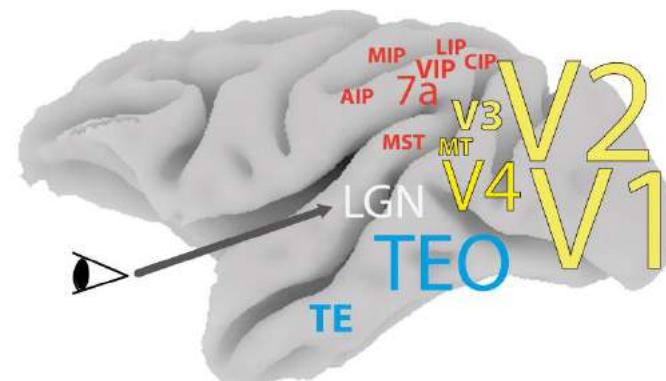
FOREWORD BY  
Shimon Ullman

AFTERWORD BY  
Tomaso Poggio

Copyrighted material

David Marr (1982): *Vision. A Computational Investigation into the Human Representation and Processing of Visual Information.*

David Hubel and Torsten Wiesel

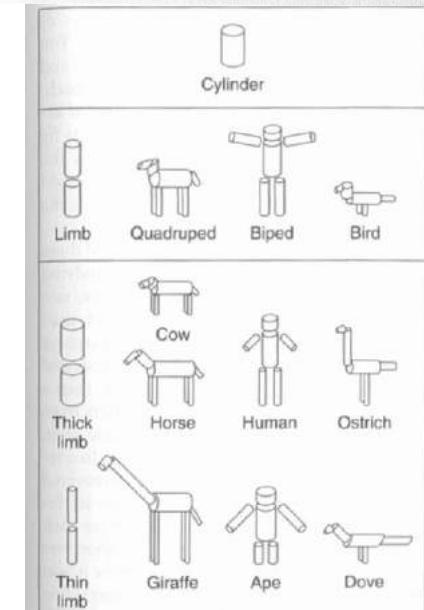
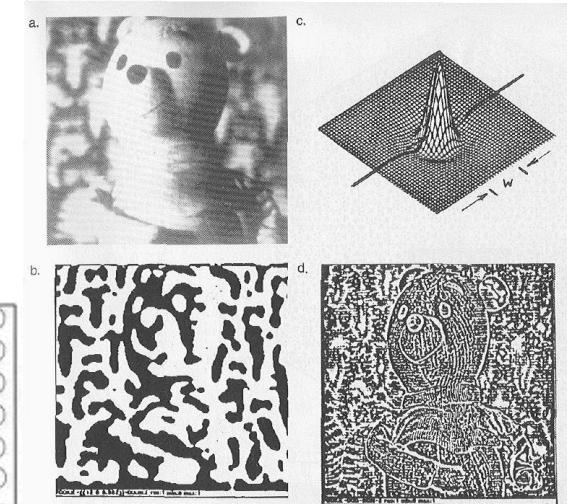
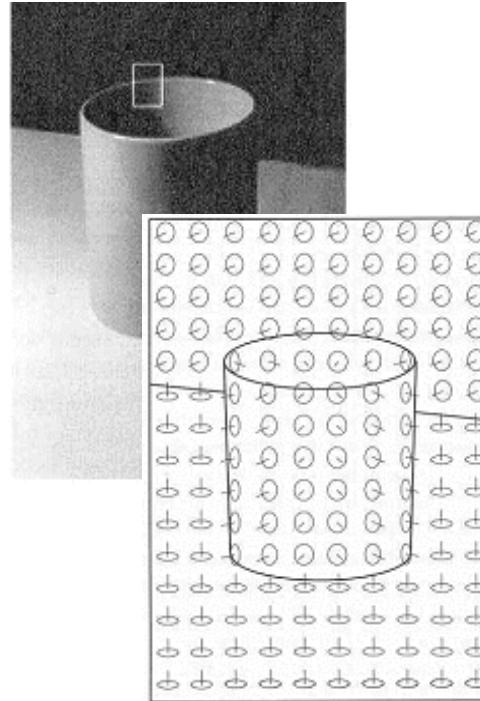


(Aus Gazzaniga et al., 1998)

The Nobel Prize in Medicine 1981

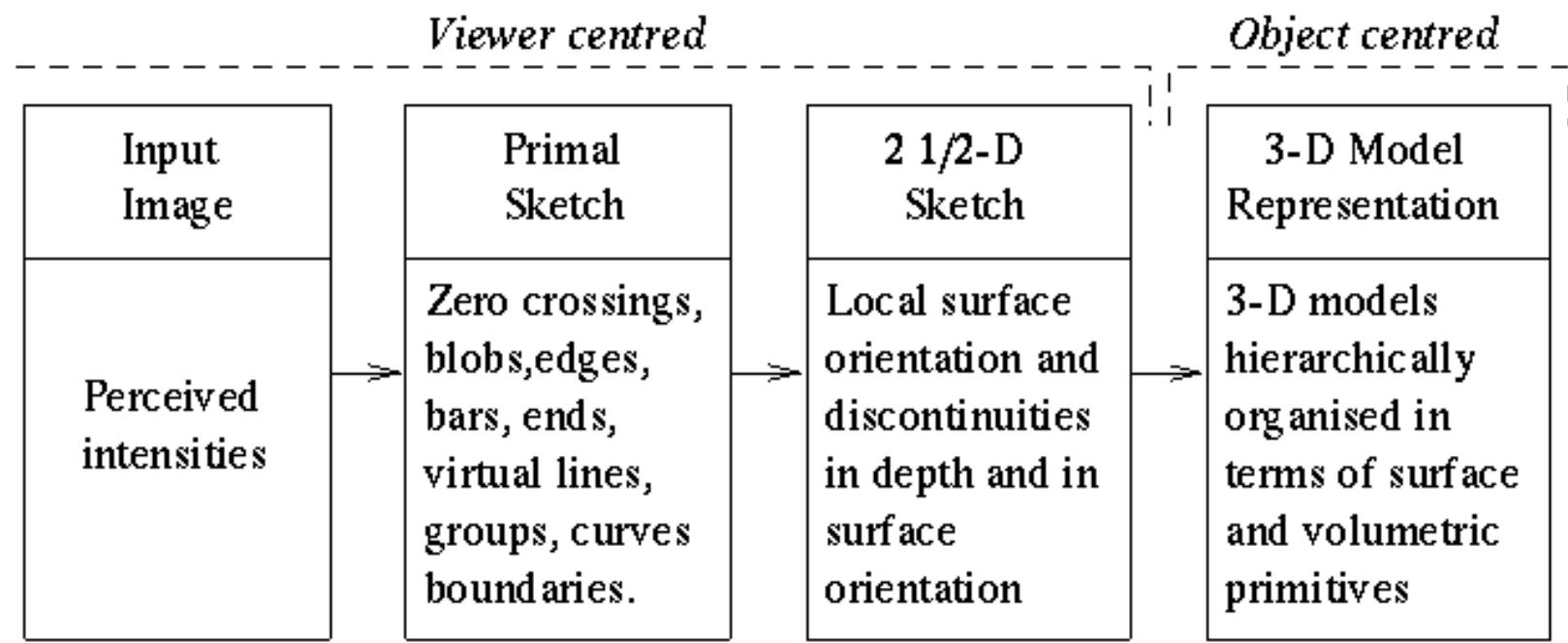
# Some remarks on the interaction of human vision research and computer vision

- David Marr 1982: Vision: A computational investigation into the human representation and processing of visual information
- 3 Stages
  - Primal Sketch: Multi-scale Edge Detection
  - 2.5D Sketch: Viewer centered Scene Representation
  - 3D Sketch: Object Centered Representation

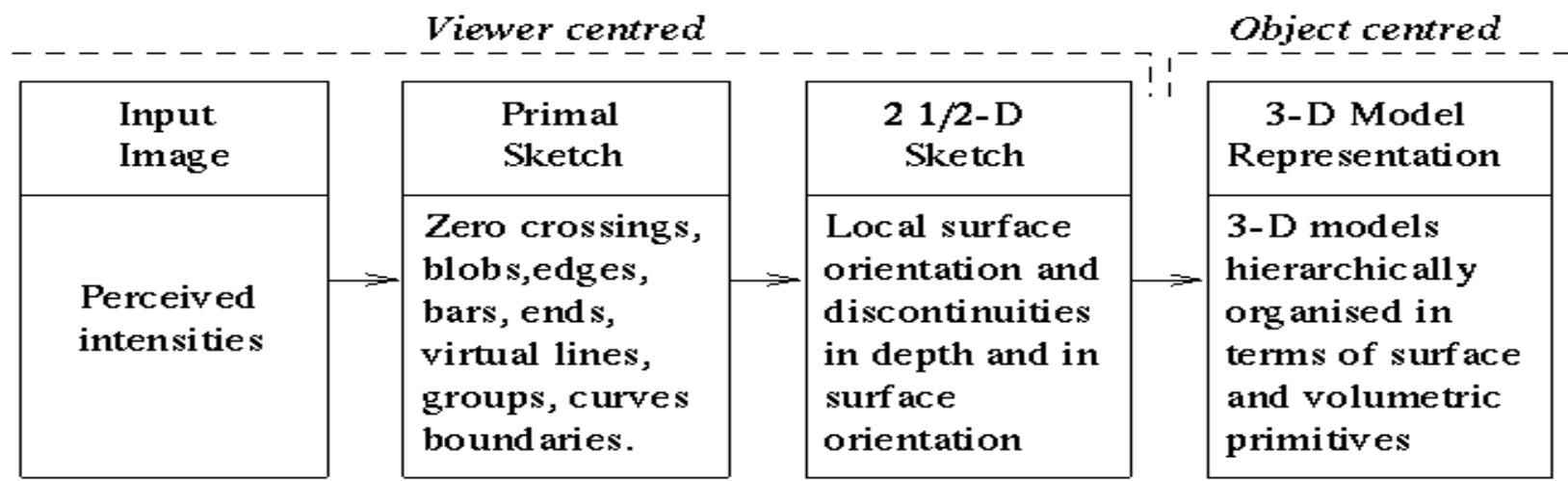


# Marr's Paradigm

- Marr: “Visual processing is modular”

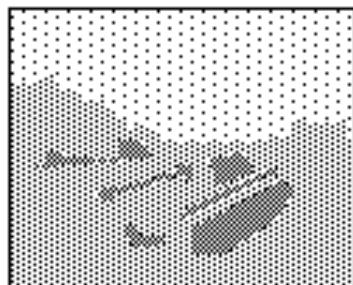


# Marr's Paradigm



[http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL\\_COPIES/GOMES1/marr.html](http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/GOMES1/marr.html)

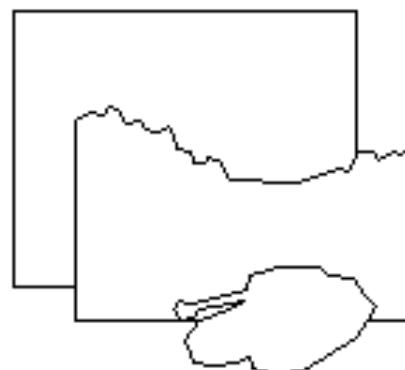
input image



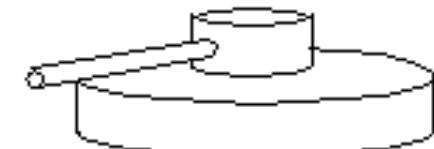
edge image



2<sup>1</sup>/<sub>2</sub>-D sketch



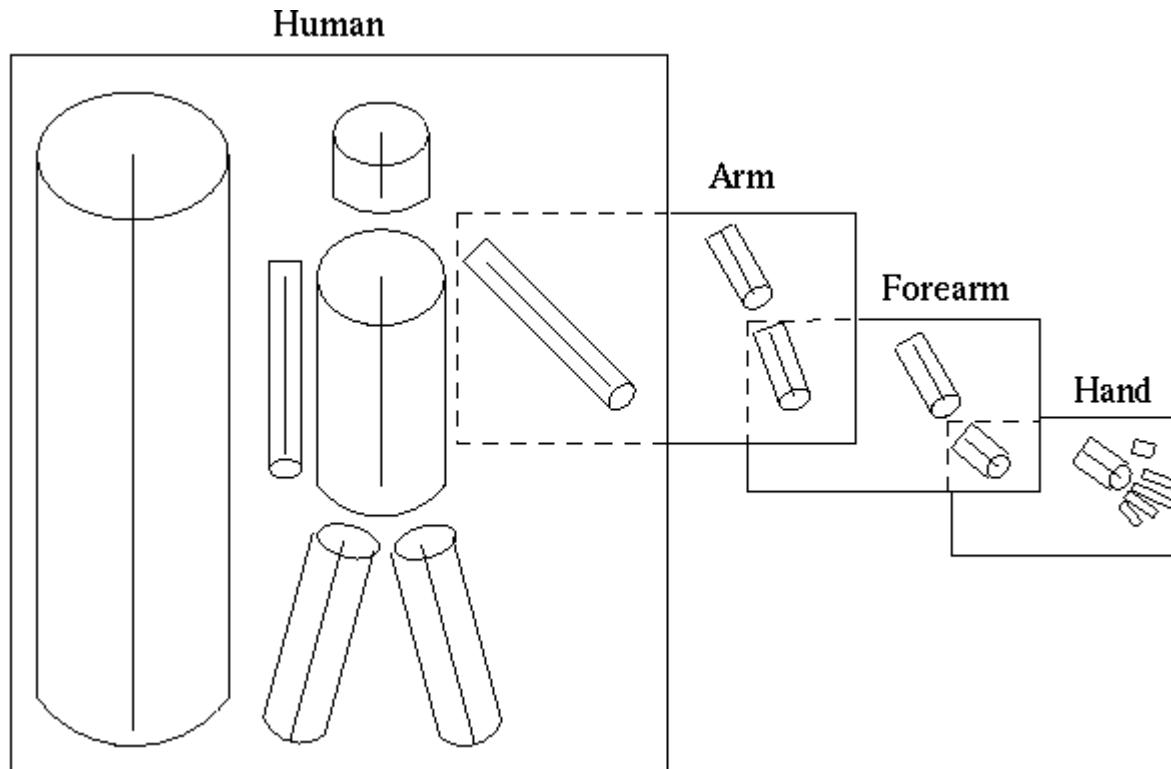
3-D model



From  
S. Lehar:

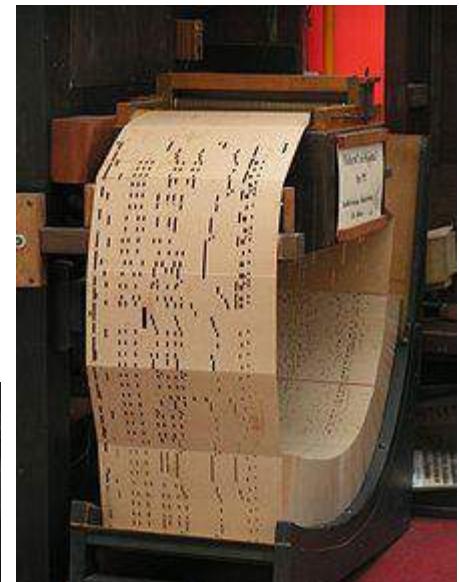
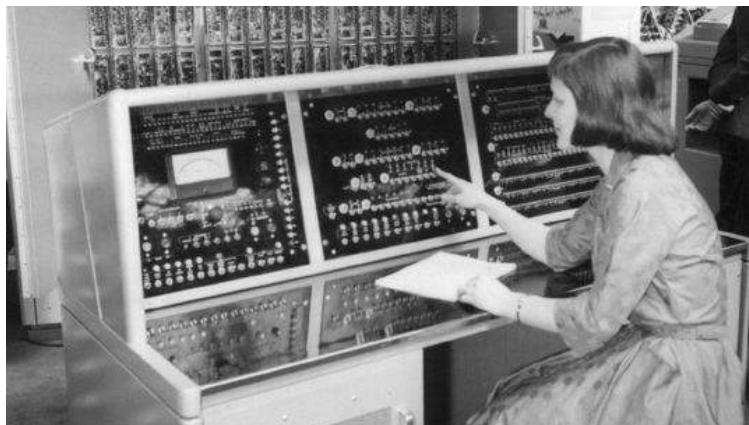
# Marr's Paradigm

- Marr's 3D Model description:



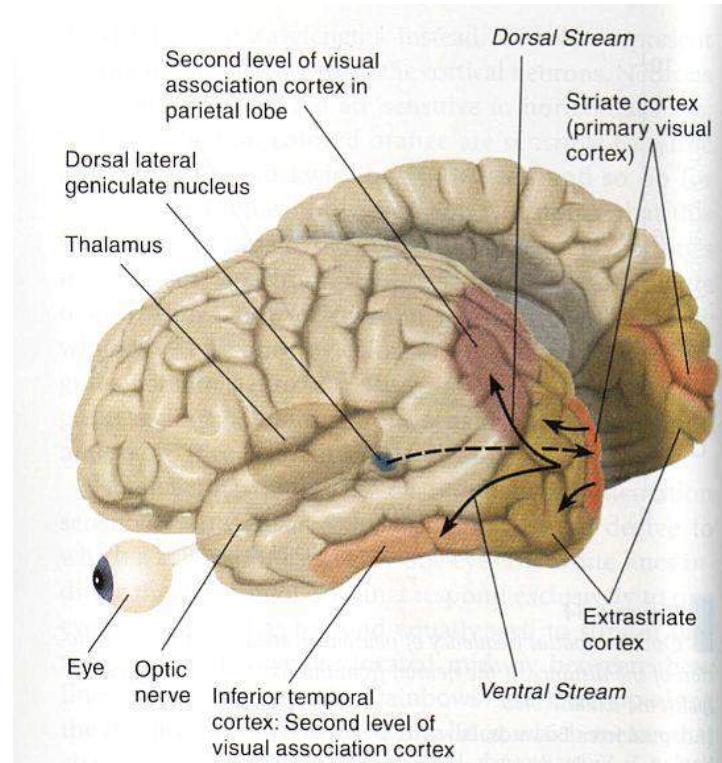
# Why did that ‘fail’? Two reasons

- The project was too ambitious at Marr’s time
  - Lack of knowledge on low-level modalities
    - Optic flow
    - Edge detection
    - Stereo
    - Structure-from-Motion
- Lack of computational resources
  - Slow clock frequency
  - No GPUs



# 'Computer Vision' and 'Biological Vision'

- In the 80<sup>th</sup> and 90<sup>th</sup> there was a strong link
- This link has been kind of diluted from 'both sides'
  - Computer Vision became a sub-discipline of Machine Learning
  - Many neurophysiologists have given up on understanding the brain on a functional level
- 'Biologically inspired' got a somehow bad reputation
  - Not efficient
  - Everything could somehow be biologically inspired



# Maybe a restart is worthwhile

- Much better understanding of early vision
- Significantly larger computational resources
- Still many unsolved problems in CV
- Aim of the paper
  - Distill essential knowledge on the human visual system for Engineers

IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, DOI:10.1109/TPAMI.2012.272 AUTHOR FINAL DRAFT

IEEE Trans Pattern Anal Mach Intell. 2013 Aug;35(8):1847-71.

## Deep Hierarchies in the Primate Visual Cortex: What Can We Learn For Computer Vision?

Norbert Krüger, Peter Janssen, Sinan Kalkan, Markus Lappe, Aleš Leonardis, Justus Piater, Antonio J. Rodríguez-Sánchez, Laurenz Wiskott

**Abstract**—Computational modeling of the primate visual system yields insights of potential relevance to some of the challenges that computer vision is facing, such as object recognition and categorization, motion detection and activity recognition or vision-based navigation and manipulation. This article reviews some functional principles and structures that are generally thought to underlie the primate visual cortex, and attempts to extract biological principles that could further advance computer vision research. Organized for a computer vision audience, we present *functional principles of the processing hierarchies* present in the primate visual system considering recent discoveries in neurophysiology. The hierarchical processing in the primate visual system is characterized by a sequence of different levels of processing (in the order of ten) that constitute a *deep hierarchy* in contrast to the *flat* vision architectures predominantly used in today's mainstream computer vision. We hope that the functional description of the deep hierarchies realized in the primate visual system provides valuable insights for the design of computer vision algorithms, fostering increasingly productive interaction between biological and computer vision research.

**Index Terms**—Computer Vision, Deep Hierarchies, Biological Modeling

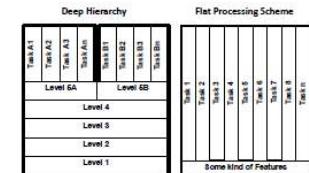
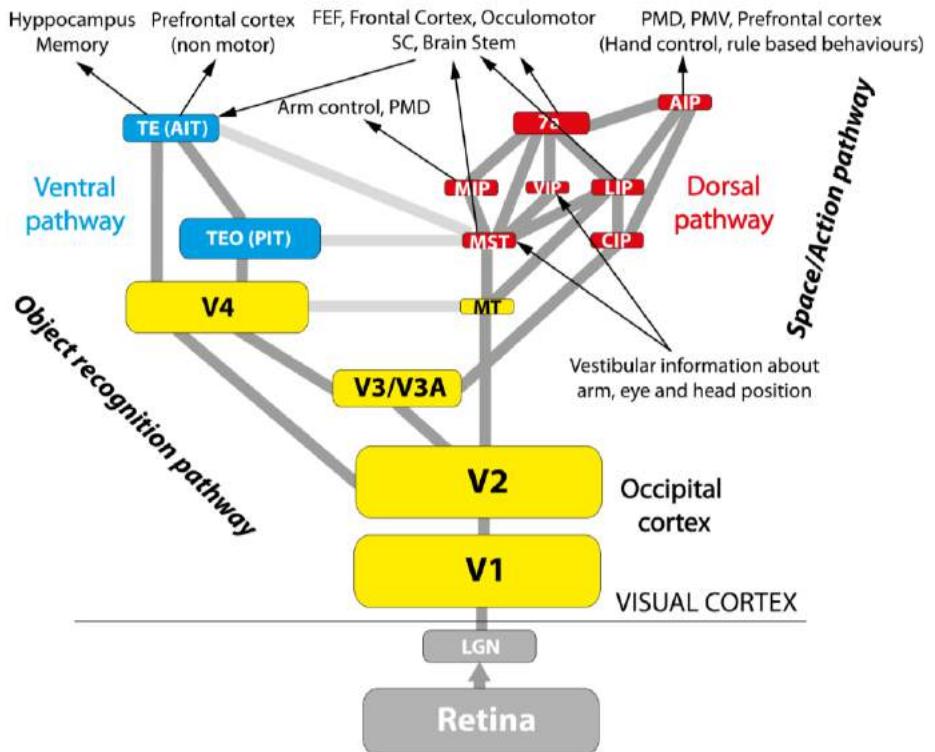


Fig. 1. Deep hierarchies and flat processing schemes

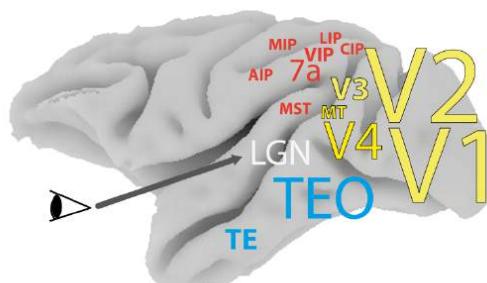
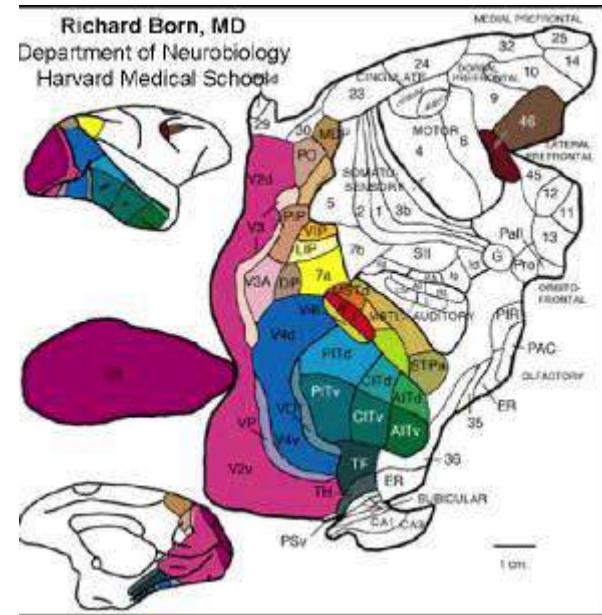
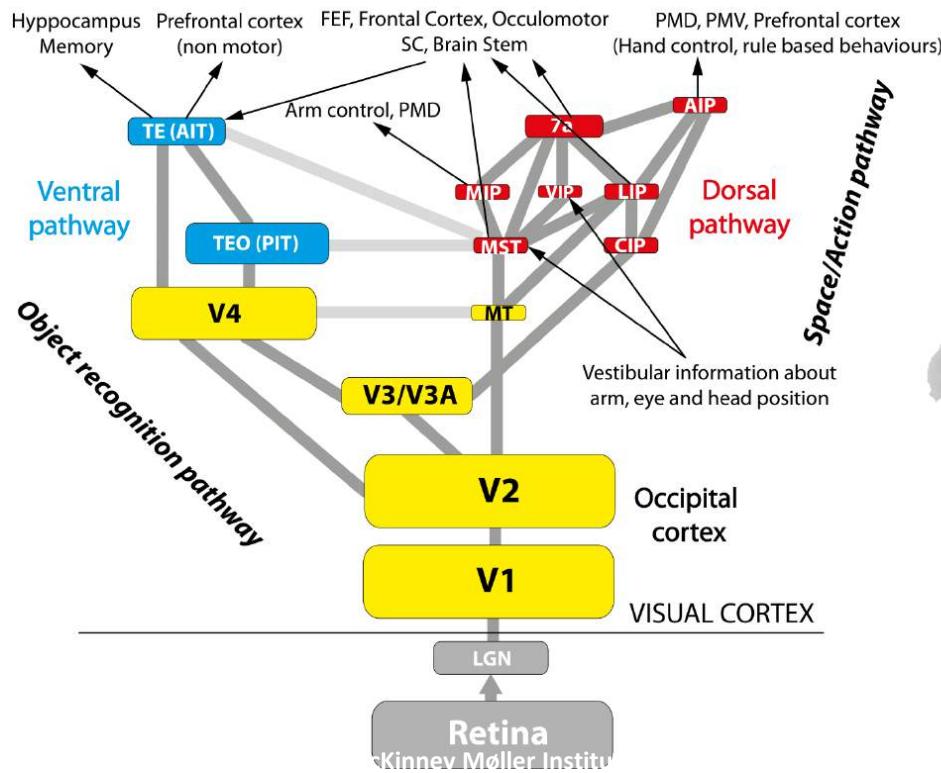
# Overview

- The primate's vision system: A deep Hierarchy
- Reflections



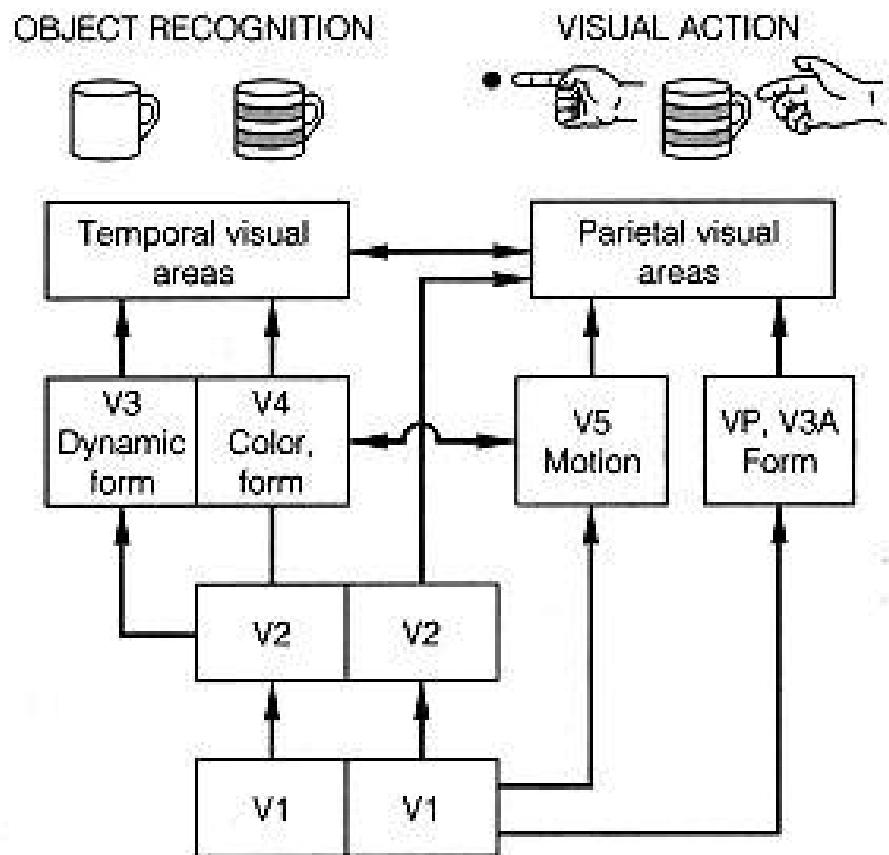
# Basic facts

- 55% of the neo-cortex of the primate brain is concerned with vision
- Division in
  - Occipital Cortex
  - Dorsal Pathway
  - Ventral Pathway

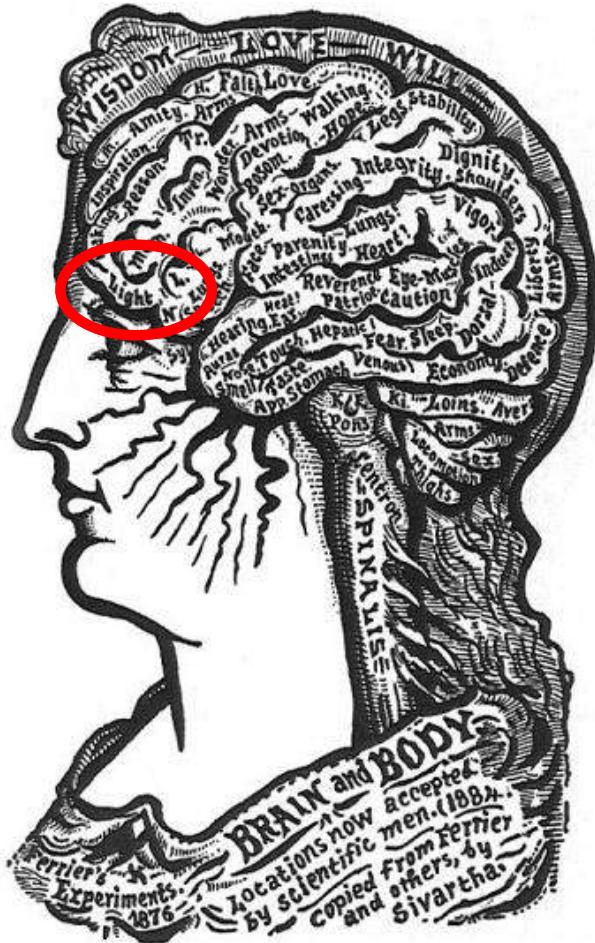


# Vision for recognition & action

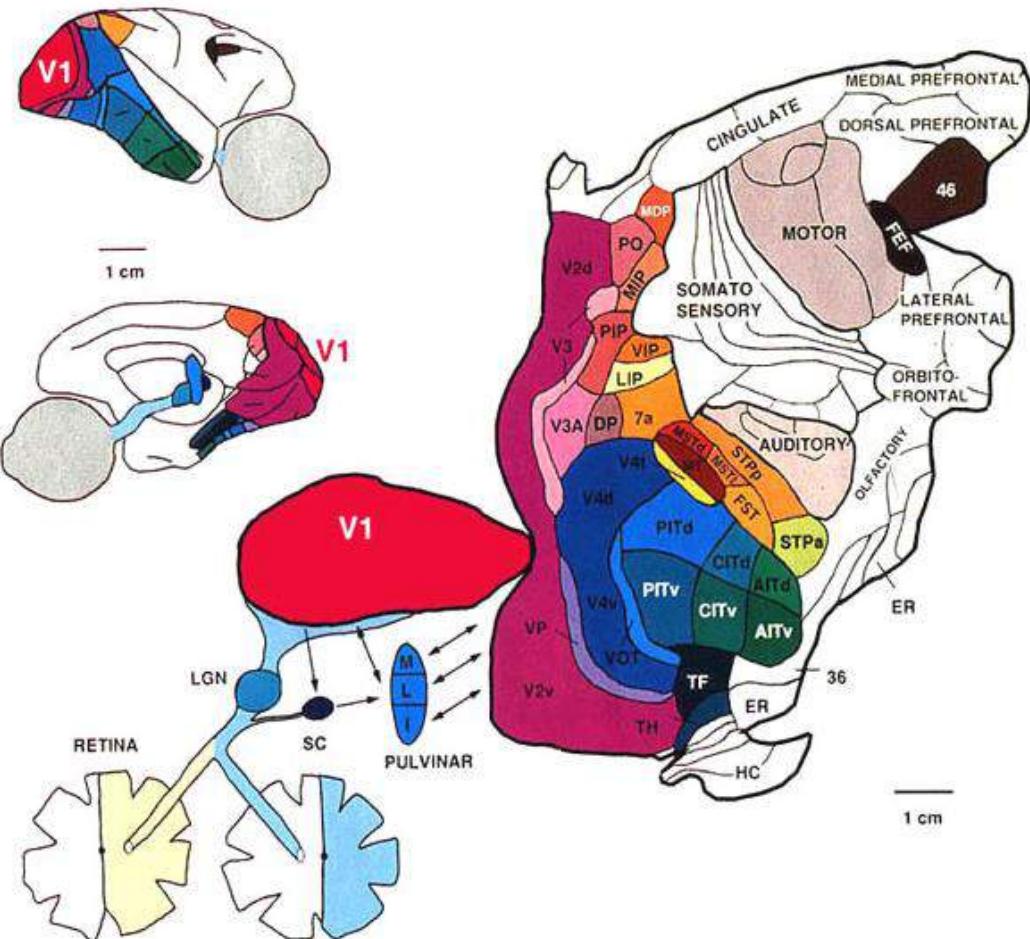
- Ventral pathway:
  - Provides “what”
  - Object recognition
- Dorsal pathway:
  - Provides “where”
  - Vision for action



# Brain Maps

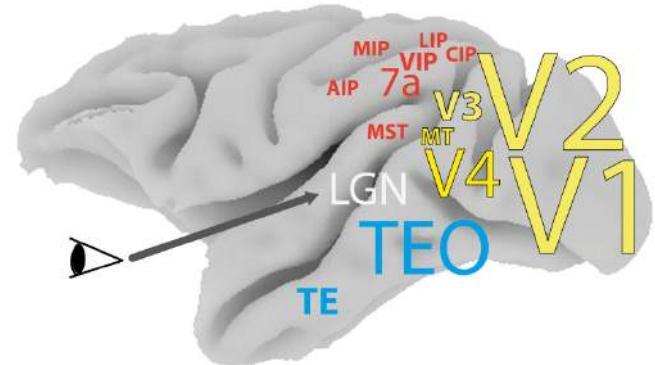
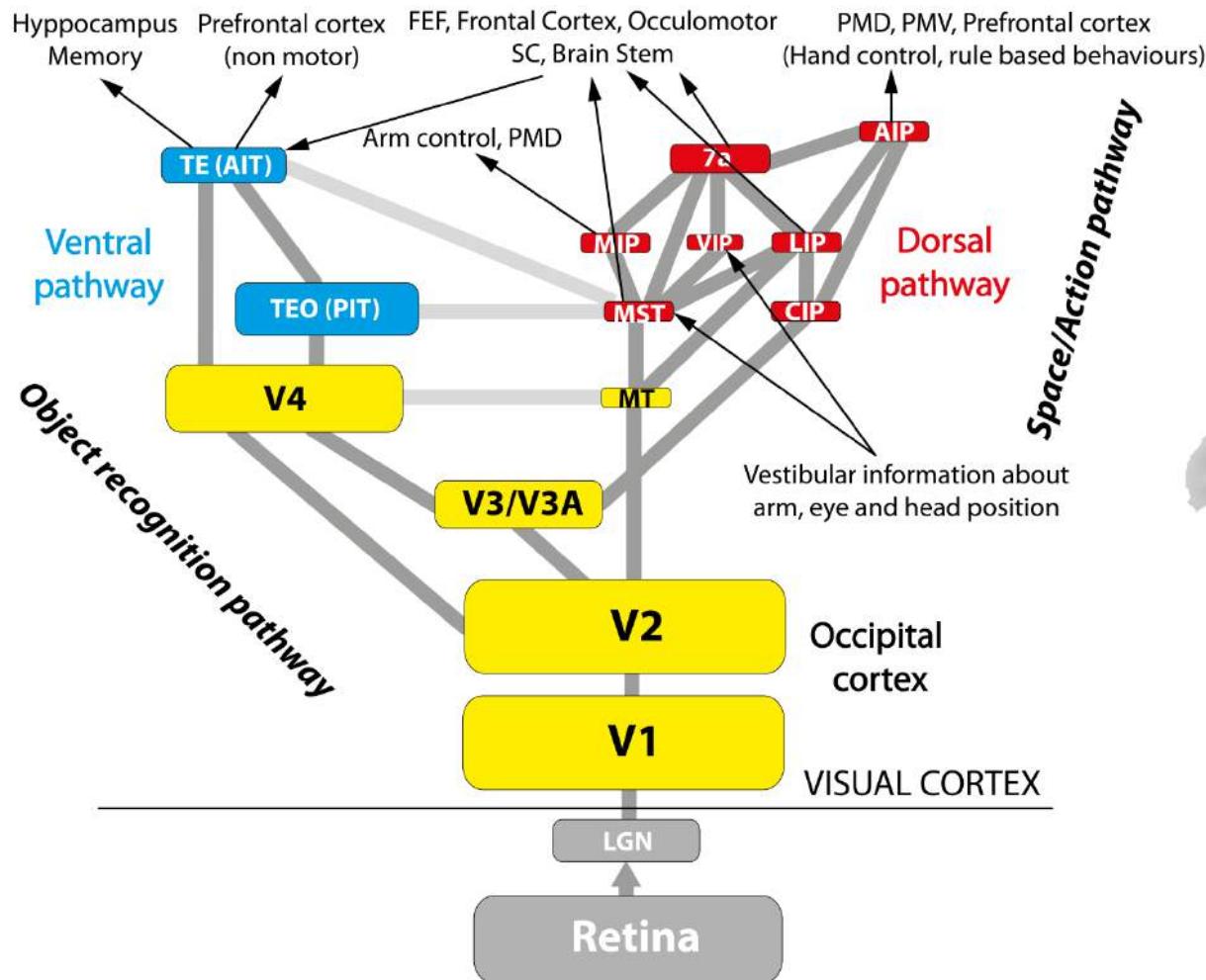


Dr. Alesha Sivartha in the late 1800s  
(published in his metaphysical book *The Book of Life: The Spiritual and Physical Constitution of Man*)



From: van Essen 1992

# Half of the brain in 15 minutes



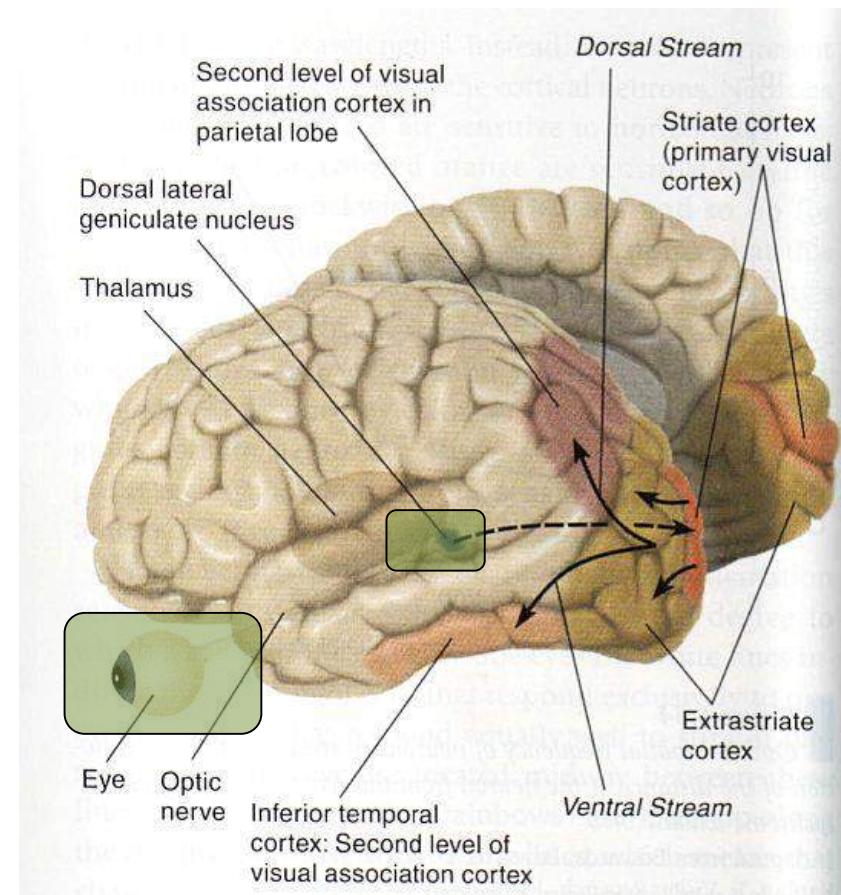
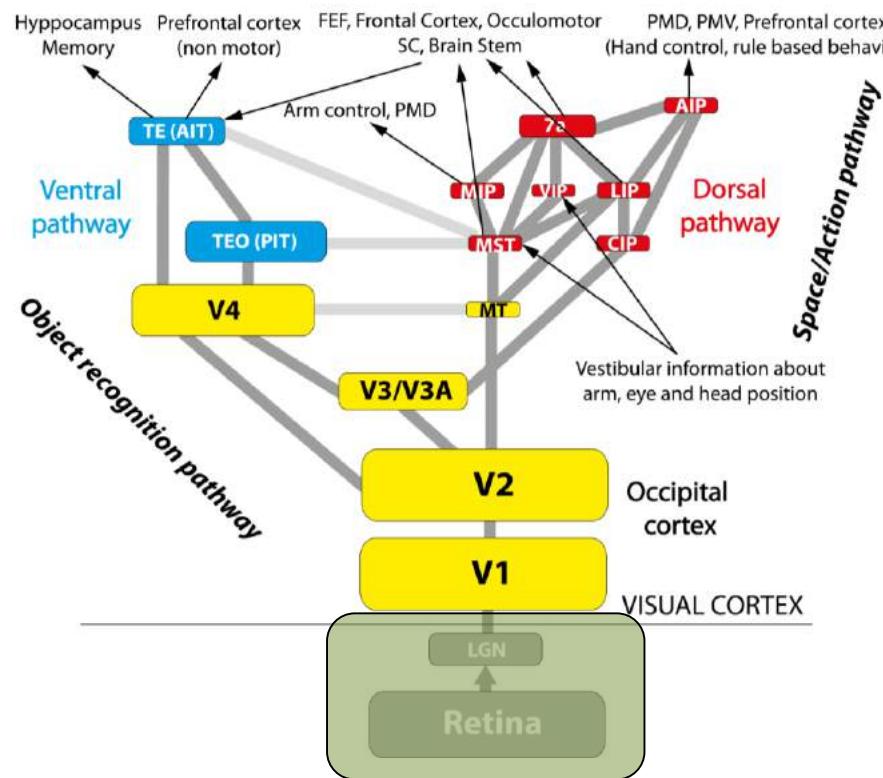
# Basic Facts

Area	Size (mm <sup>2</sup> )	RFS	Latency (ms)	co/bi lat.	rt/st/cl/co	CI/SI/PI/OI	Function
Sub-cortical processing							
Retina LGN	1018 0.1	0.01 0.1	20-40 30-40	bl co	+/-/- +/-/-	-/-/- -/-/-	sensory input, contrast computation relay, gating
Occipital / Early Vision							
V1 V2 V3/V3A/VP V4/VOT/V4t MT	1120 1190 325 650 55	3 4 6 8 7	30-40 40 50 70 50	co co co co co	+/-/+ +/-/+ +/-/+ +/-/+ +/-/+	-/-/- -/-/- -/-/- +/-/- +/-/+	generic feature processing generic feature processing generic feature processing generic feature processing / color motion
Sum	3340						
Ventral Pathway / What (Object Recognition and Categorization)							
TEO TE	590 180	3-5 10-20	70 80-90	co bl	(+)/-/- -/-/+	?/-/? +/-/+(-)	object recognition and categorization
Sum	770						
Dorsal Pathway / Where and How (Coding of Action Relevant Information)							
MST CIP VIP 7a LIP AIP MIP	60 ? 40 115 55 35 55	>30 ? 10-30 >30 12-20 5-7 10-20	60-70 ? 50-60 90 50 60 100	bl bl bl bl cl bl co	+/-/+/- +/-/? -/-/- (+)/-/- +/-/- ?/-/+/- +/-/?	1 +/-/? 1 ?/-/+/- ?/-/- ?/-/+/- 1	optic flow, self-motion, pursuit 3D orientation of surfaces optic flow, touch, near extra personal space Optic flow, heading salience, saccadic eye movements grasping reaching
Sum	585						

TABLE 1

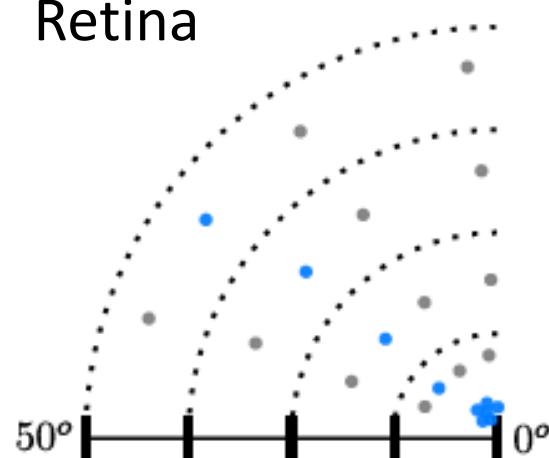
Basic facts on the different areas of the macaque visual cortex based on different sources [44], [28], [95], [141], [161] *First column: Name of Area. Second column: Size of area in mm<sup>2</sup>. '?' indicates that this information is not available. Third column: Average receptive field size in degrees at 5 degree of eccentricity. Fourth column: Latency in milliseconds. Fifth Column: Contra versus bilateral receptive fields. Sixth Column: Principles of organization: Retinotopic (rt), spatiotopic (st), clustered (cl) columnar (co) Seventh Column: Invariances in representation of shape: Cue-Invariance (CI), Size Invariance (SI), Position Invariance (PI), Occlusion Invariance (OI). '1' indicates that this entry is irrelevant for the information coded in these areas. Eighth Column: Function associated to a particular area.*

# Pre-cortical Areas

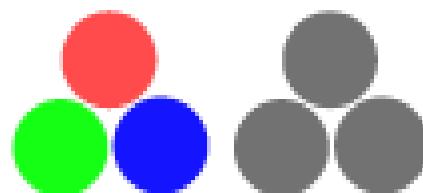
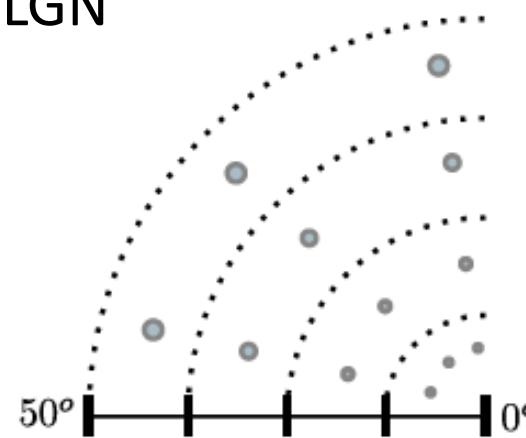


# Precortical Areas

Retina



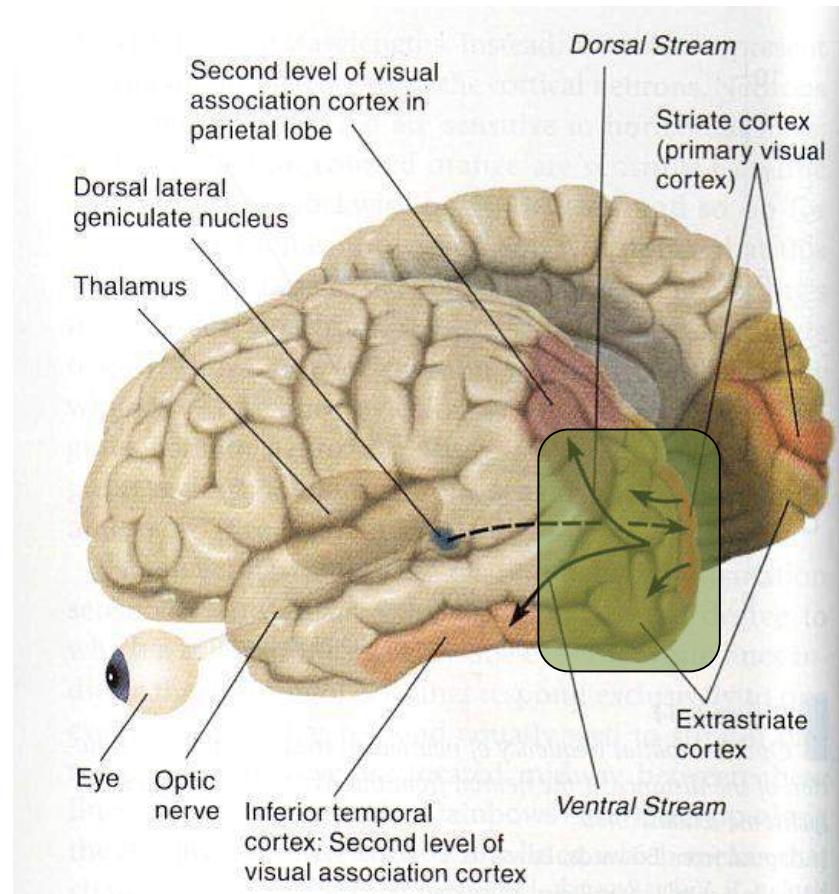
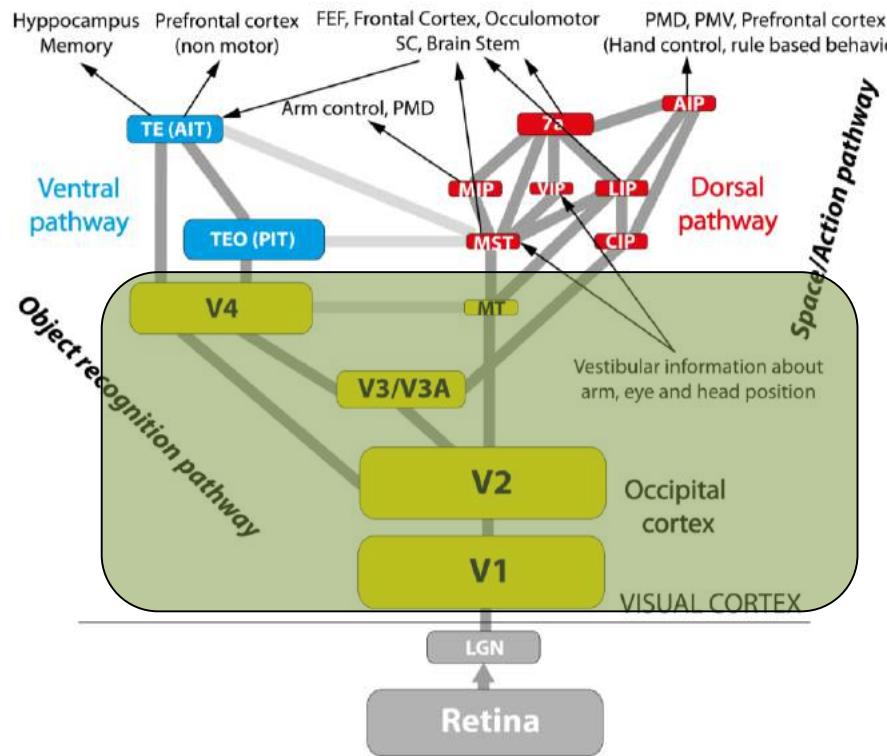
LGN



- No Feature Transformation
- Preparing for Stereo

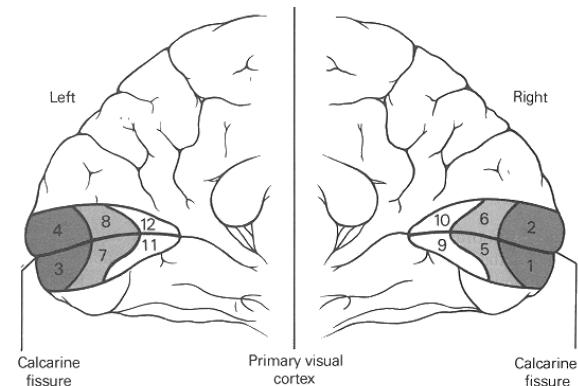
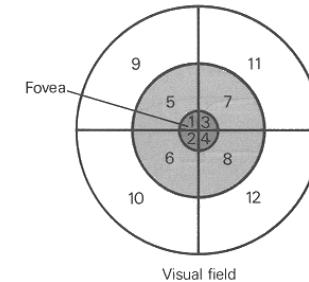


# Occipital Cortex

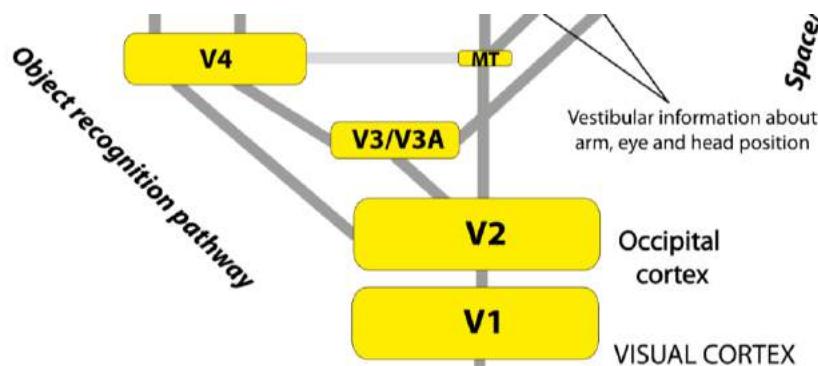


# Occipital Cortex

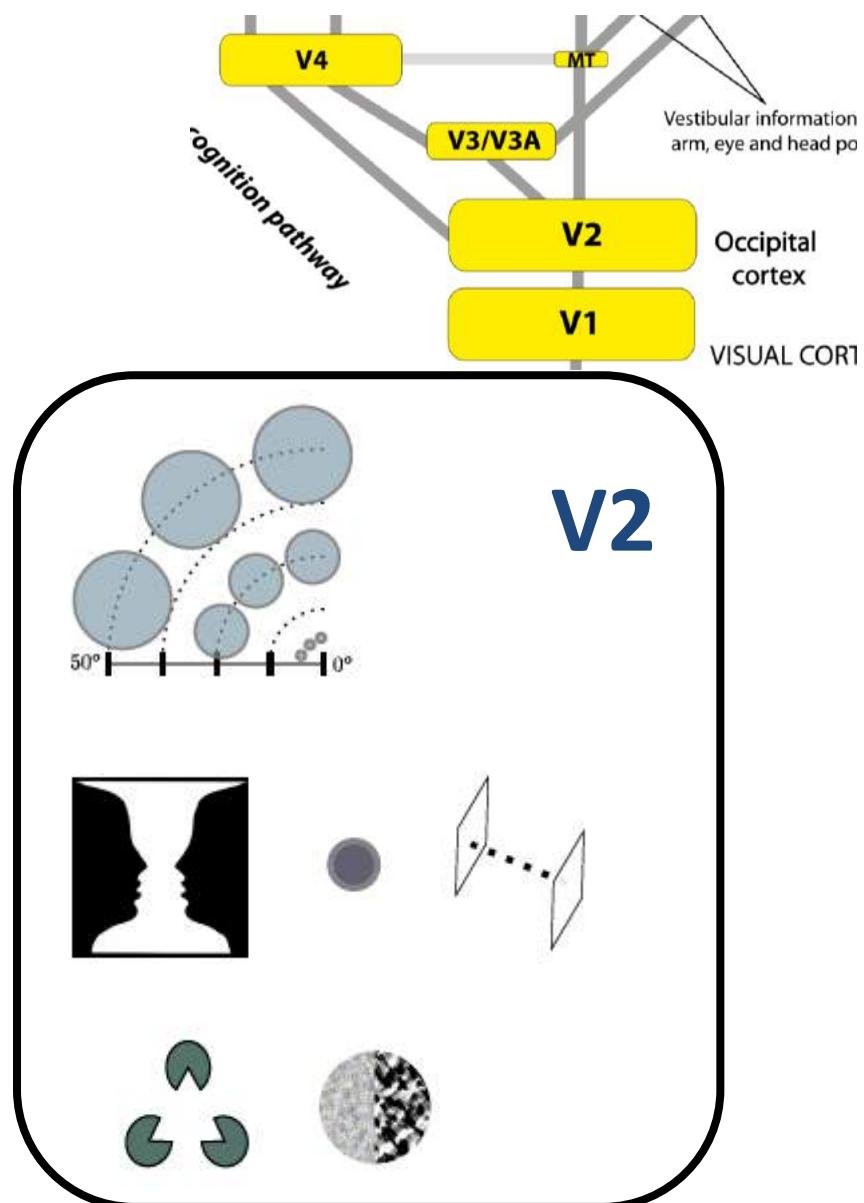
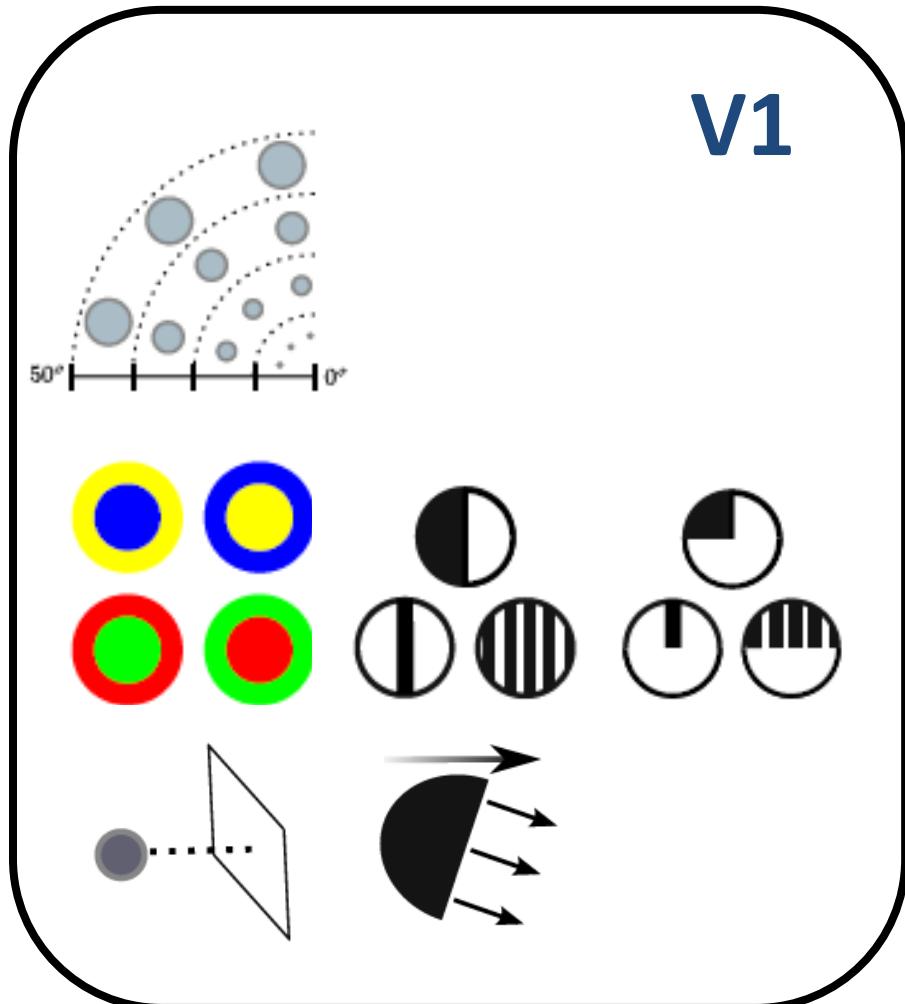
- More than 70% of the visual cortex
  - Occipital Cortex 3340mm<sup>2</sup>
  - Ventral Pathway 770mm<sup>2</sup>
  - Dorsal Pathway 585mm<sup>2</sup>
- Processing
  - Task unspecific generic scene representation



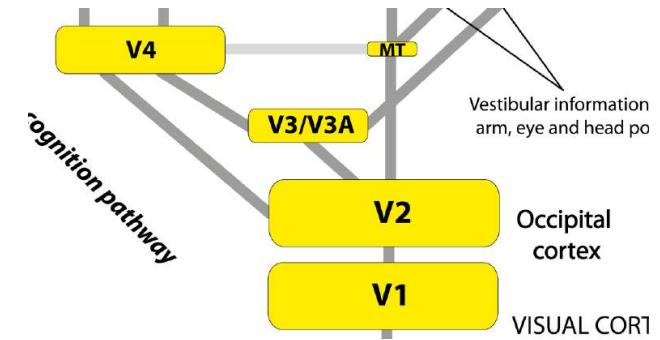
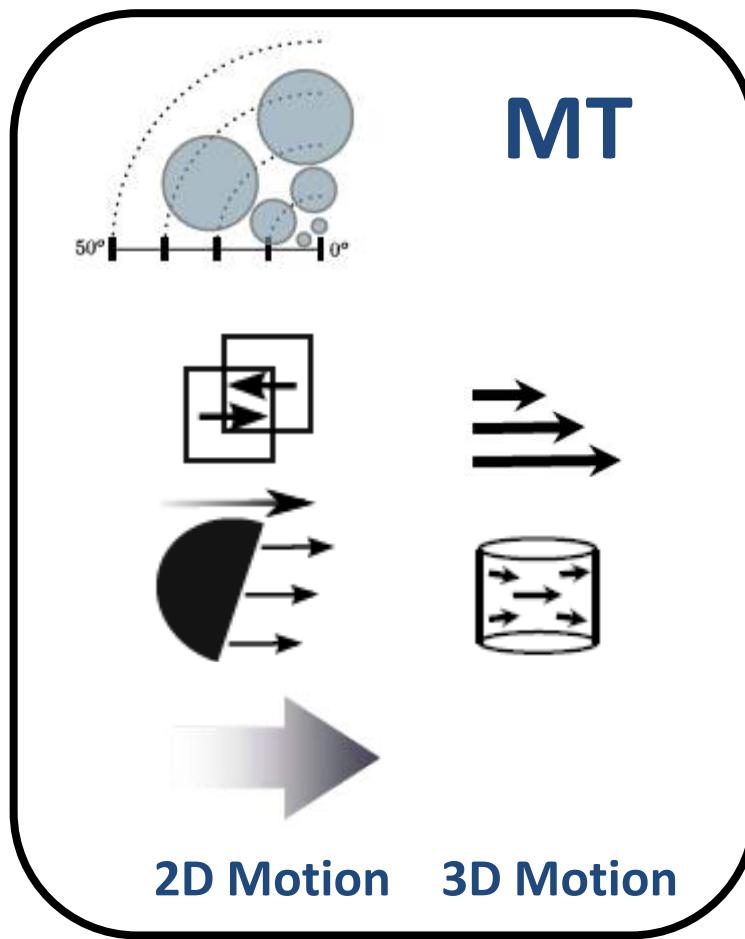
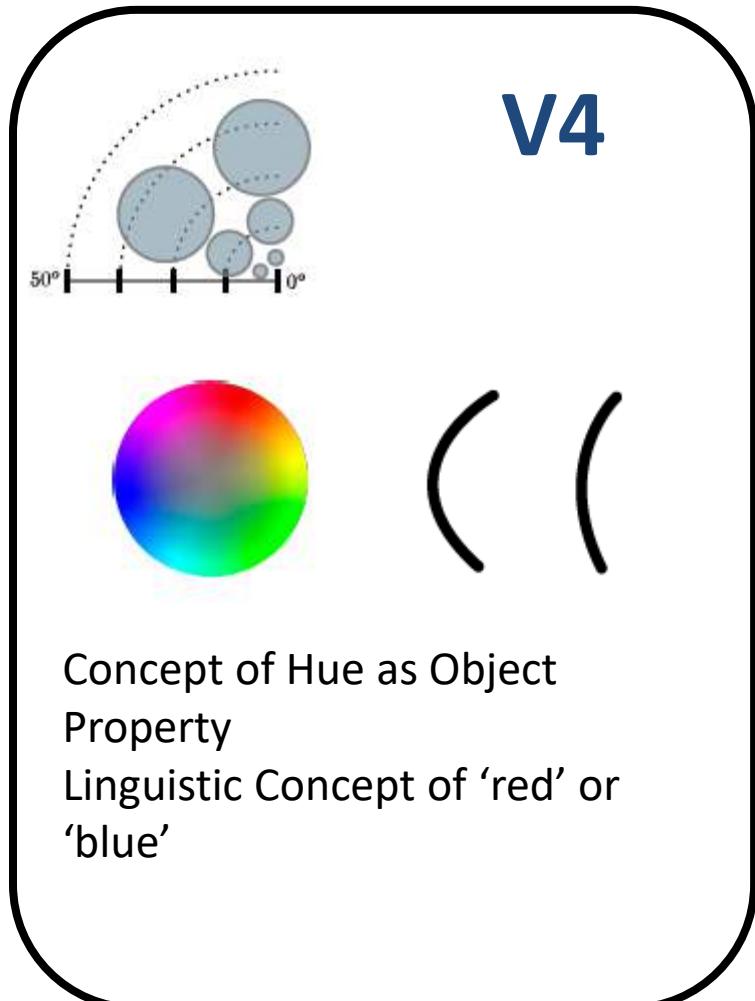
Retinotopic Organization



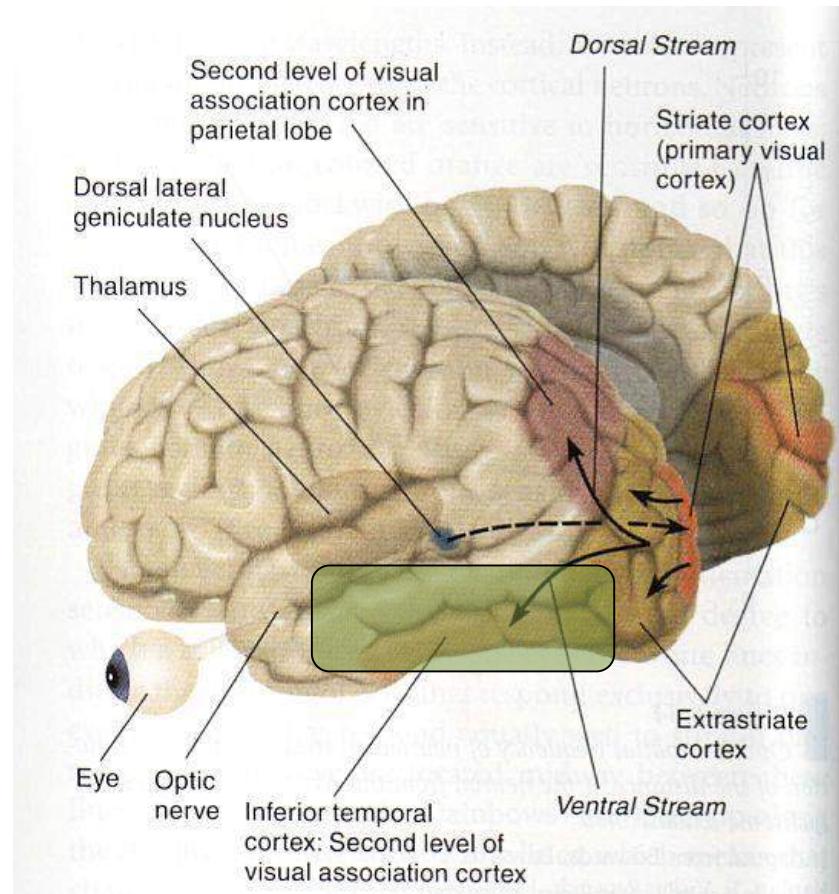
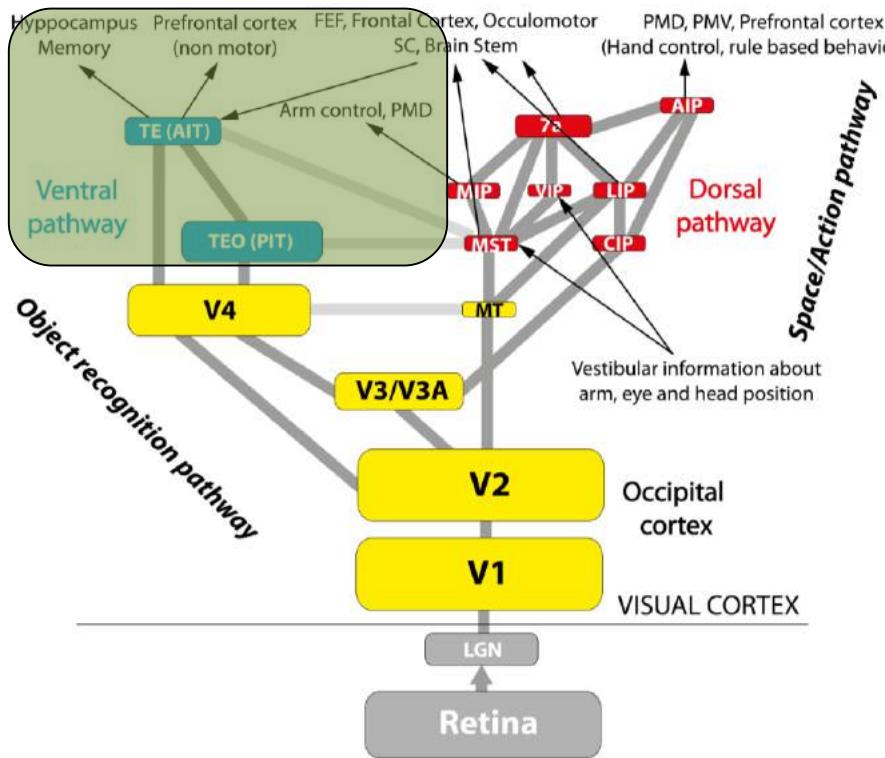
# Occipital Cortex: V1 and V2



# V4 and MT

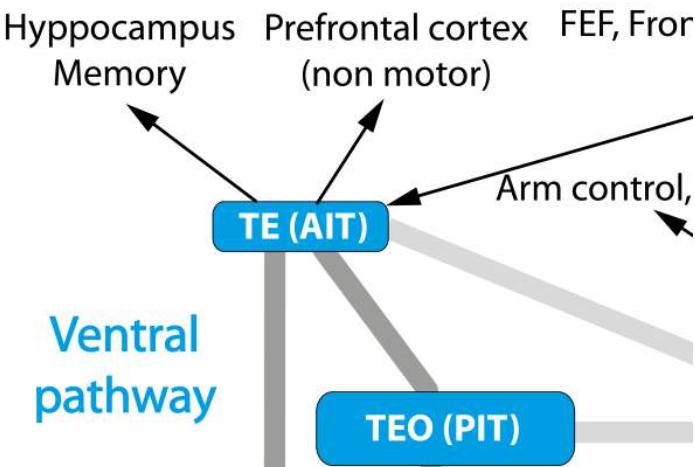


# Ventral Pathway

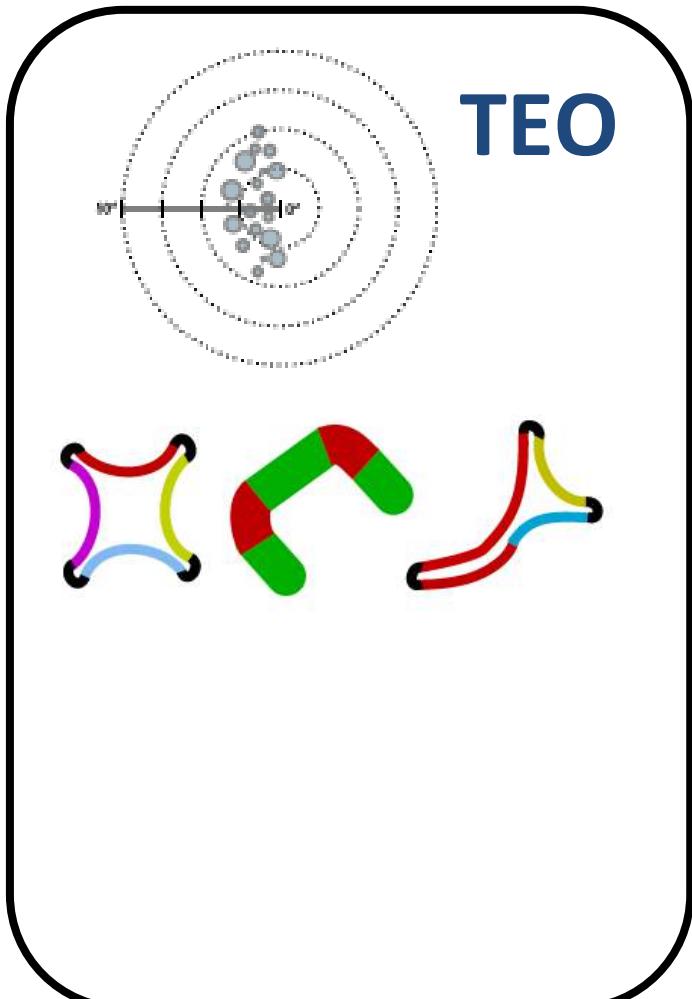


# Ventral Pathway

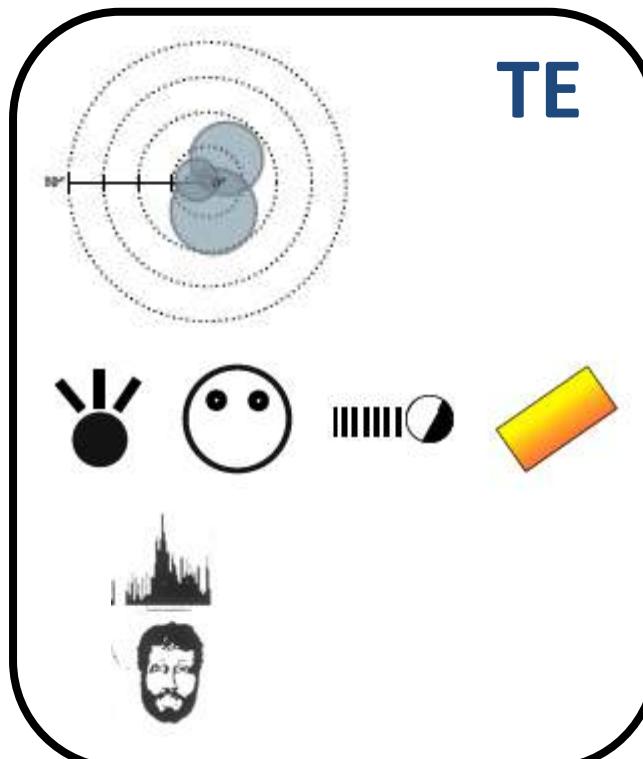
- Ca. 17% of the visual cortex
  - Occipital Cortex 3340mm<sup>2</sup>
  - Ventral Pathway 770mm<sup>2</sup>
  - Dorsal Pathway 585mm<sup>2</sup>
- Processing
  - Object Recognition and Categorization
  - Many suggestions for how to divide into areas



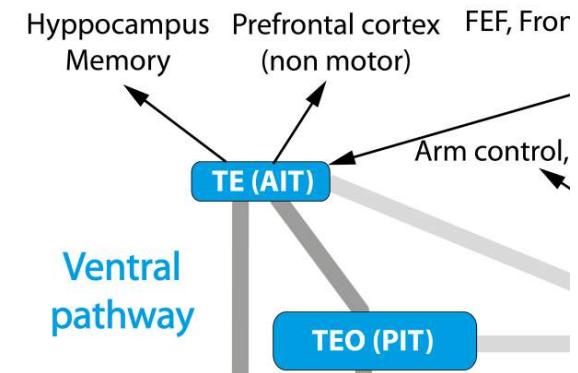
# Ventral Pathway: TEO and TE



Orientation and shape selectivity.



Selectivity to complex  
features / shapes

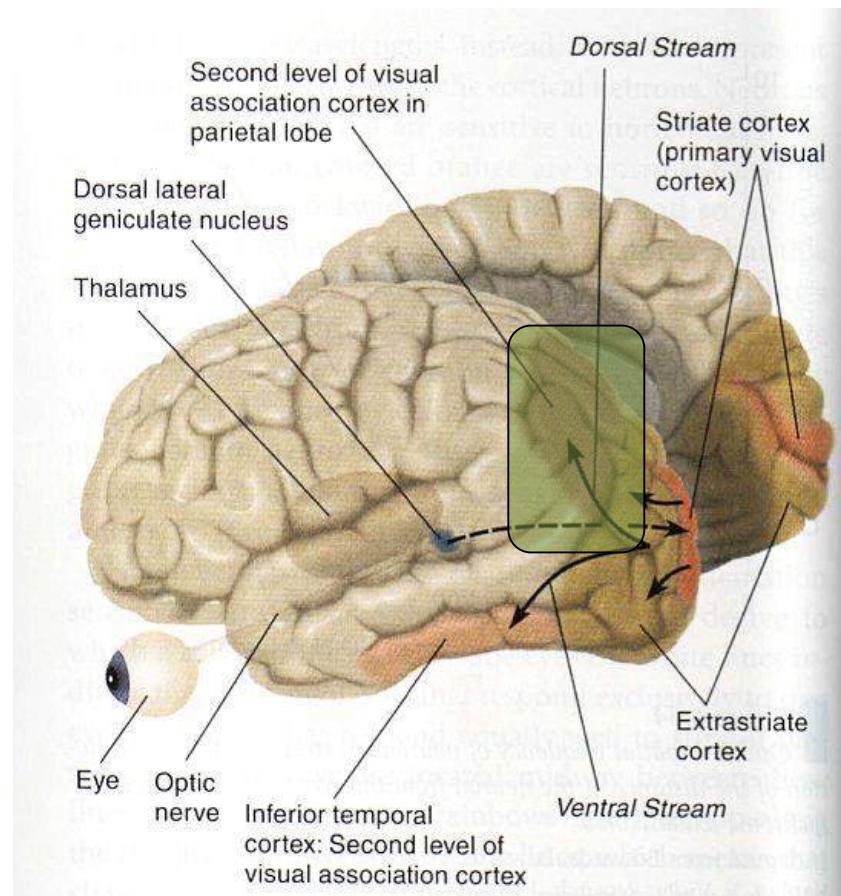
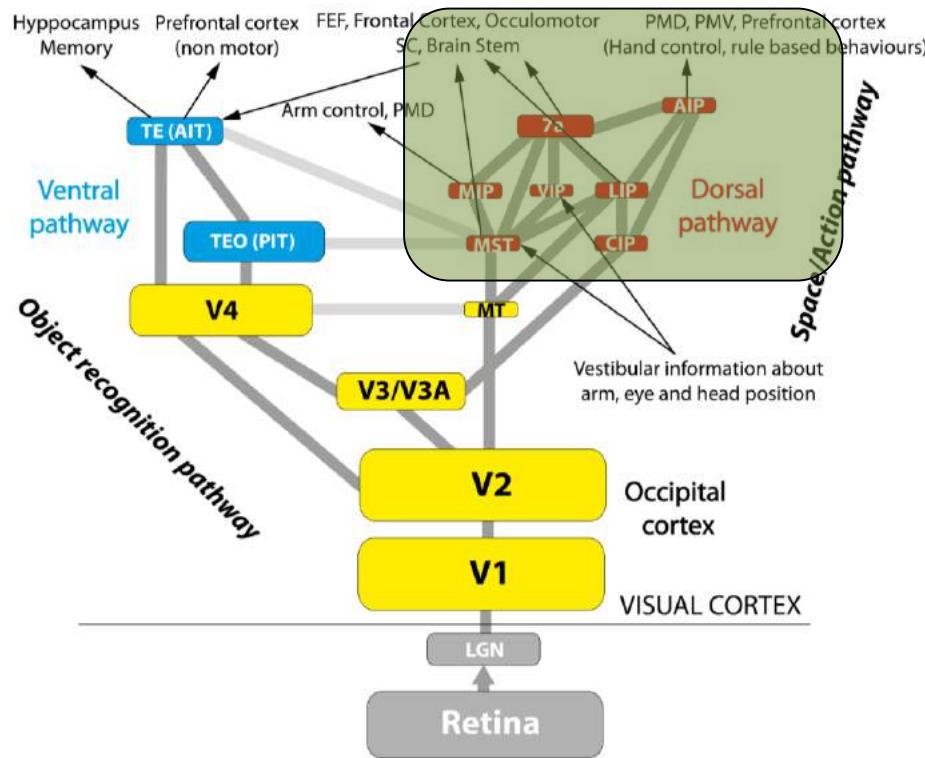


feature, it is known that the selectivity increases gradually along the ventral stream, along the parallel to the stimulus axis within columns. This is true for a variety of features we have examined, including the cortical selectivity column, which is similar to the stimulus axis selectivity of orientation within the columns of V1. The wider and larger modules in TE is characteristic of the hypercolumns and extended modules in V1. The stimulus selectivity does not increase gradually from the center to the border. The discontinuity is coincident with the hypercolumns or columns representing different features; however, the discontinuity is stable at the same level of feature complexity. For example, in a space of very high dimensionality, a specific set of features can be mapped onto a hypercolumn. The selectivity of each feature is determined by the size of the hypercolumn (2).

The spatial organization of TEO modules suggests an invariant representation of object features along their spatial extent, such as color or similarity between objects. As

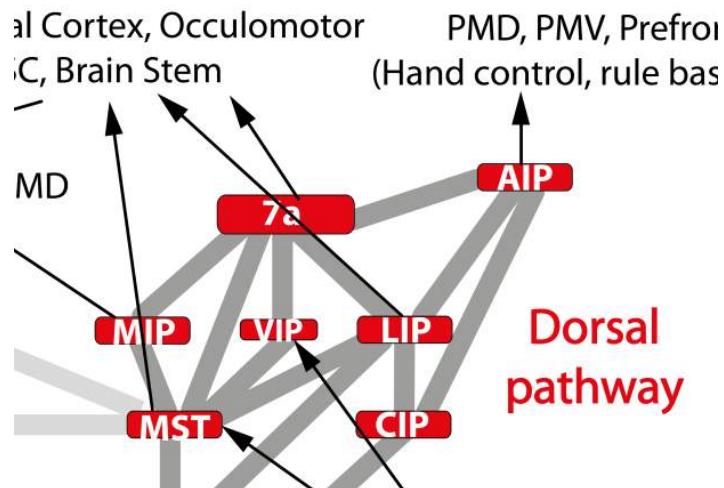
Tanaka

# Dorsal Pathway

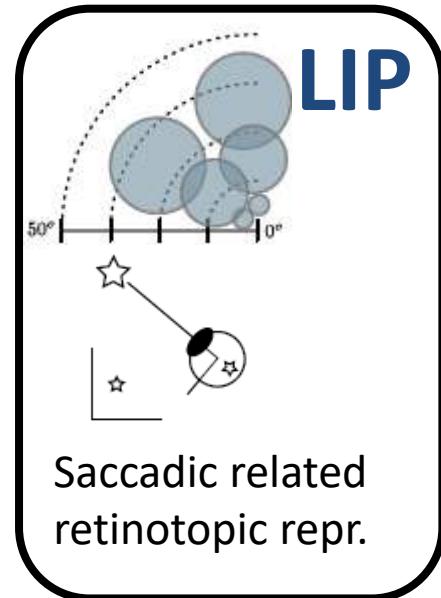
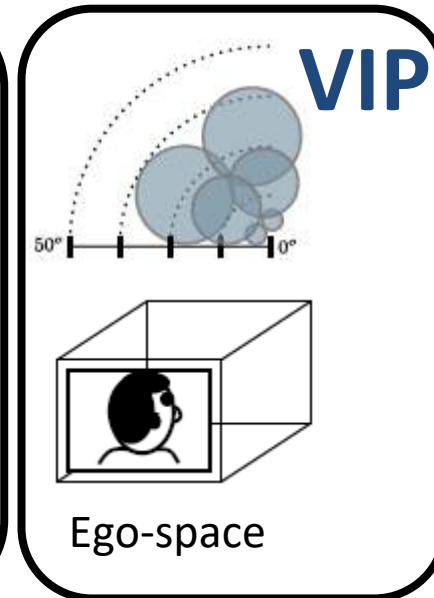
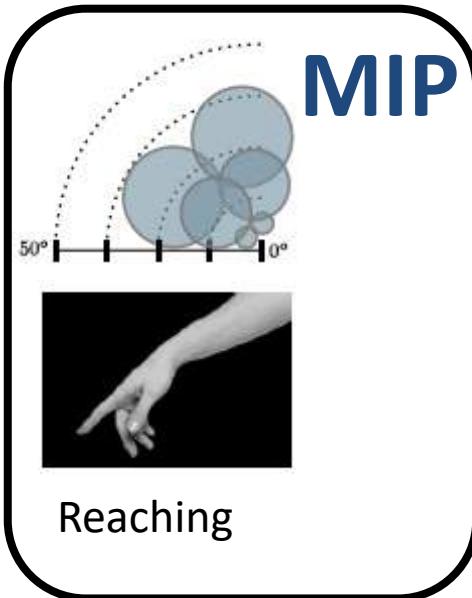
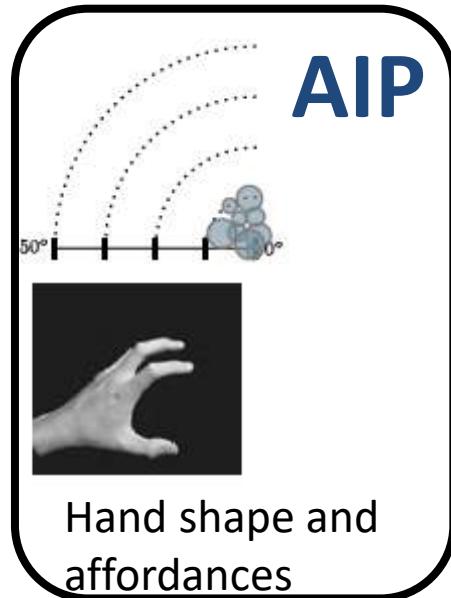
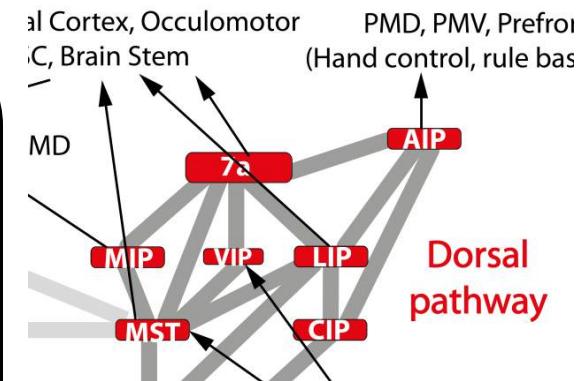
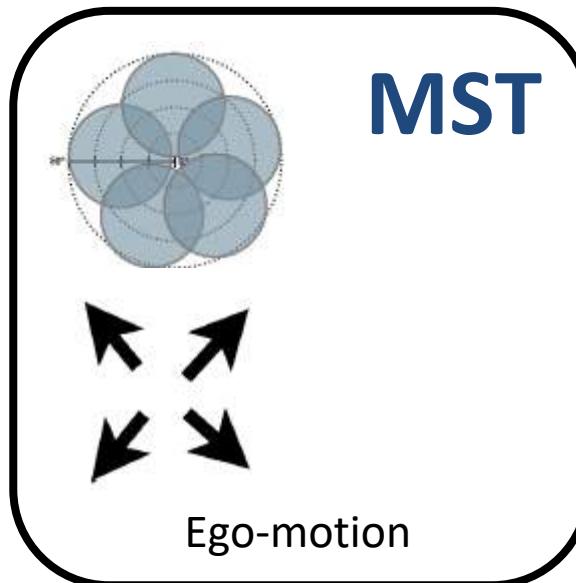
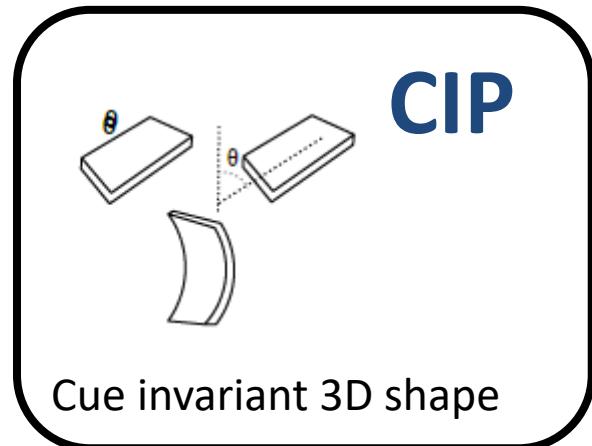


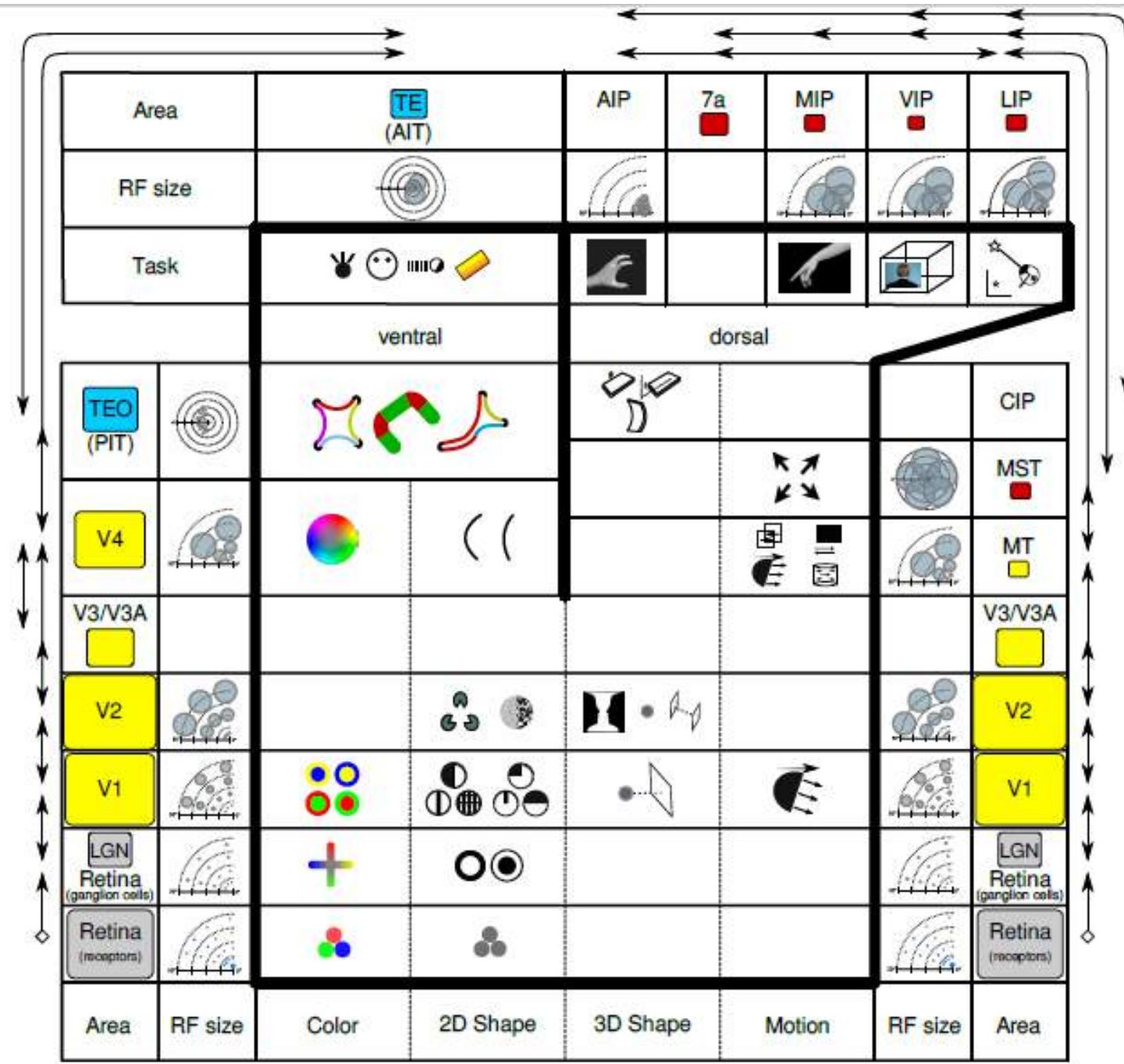
# Dorsal Pathway

- Ca 15% of the visual cortex
  - Occipital Cortex 3340mm<sup>2</sup>
  - Ventral Pathway 770mm<sup>2</sup>
  - Dorsal Pathway 585mm<sup>2</sup>
- Processing
  - Much less known than Ventral Pathway
  - Many more distinguished areas
  - Coding visual information related to action and position in space



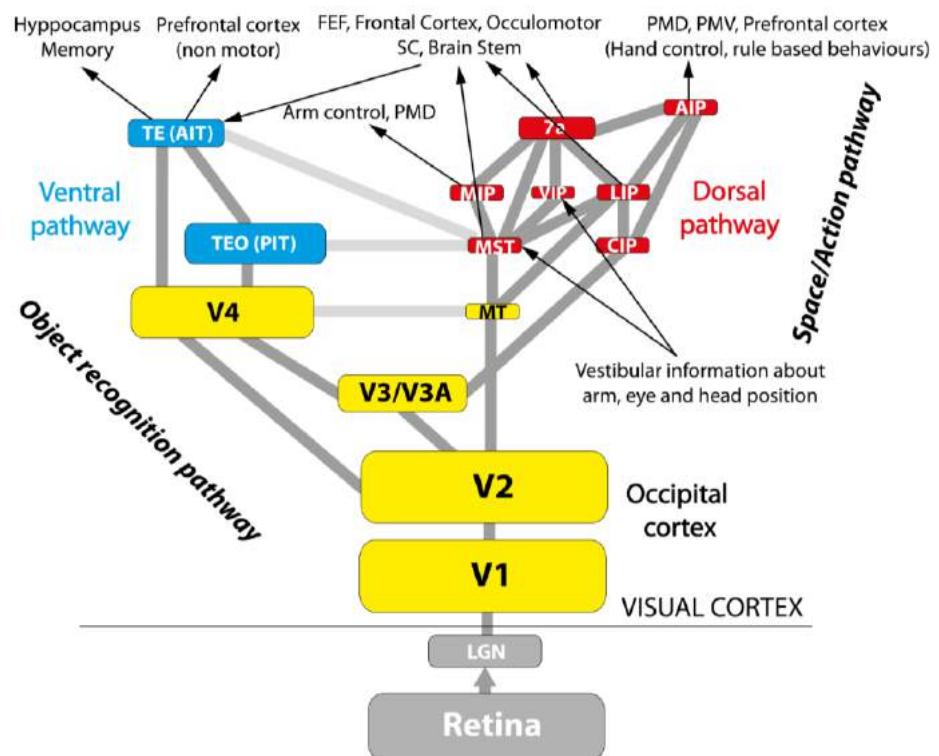
# Dorsal Pathway





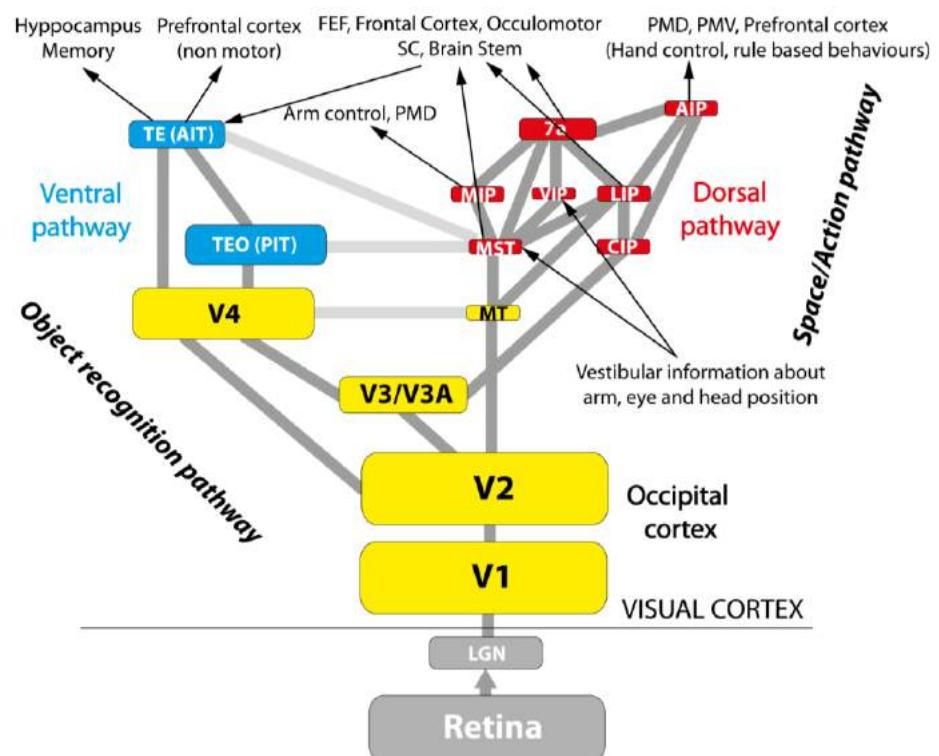
# Overview

- Some annoying prior remarks
- The primate's vision system: A deep Hierarchy
- What can we learn?

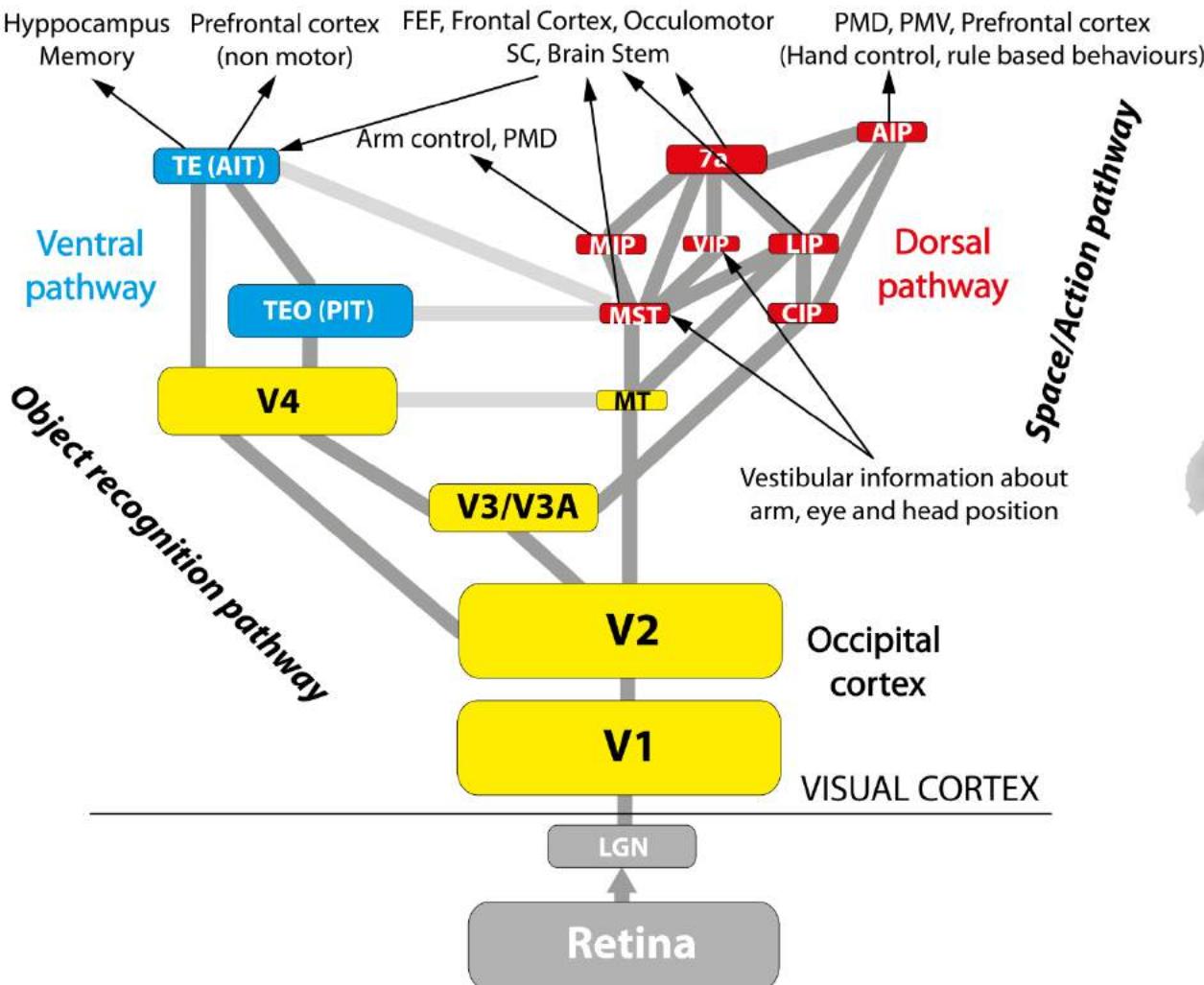


# What can we learn?

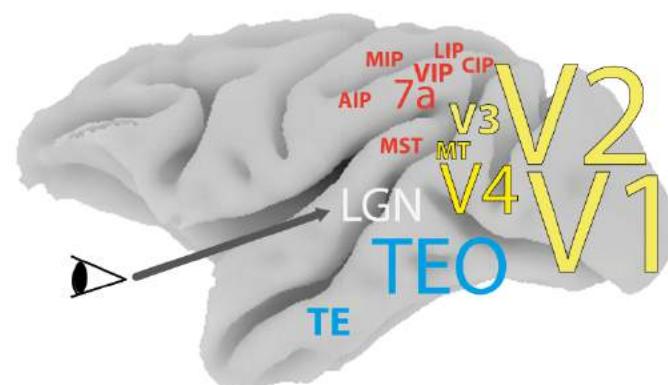
- The primate's vision
  - uses a *huge amount of resources* in the brain organized in a *deep hierarchy*
  - devotes its main vision resources to a *two step computation of a rich set of descriptors* in a *generic representation*



# Deep Hierarchy

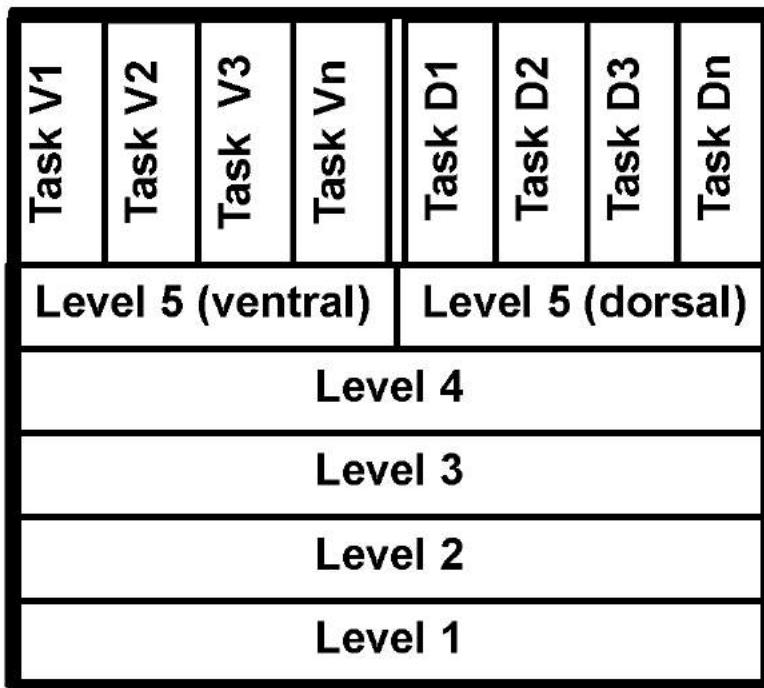


*Space/Action pathway*

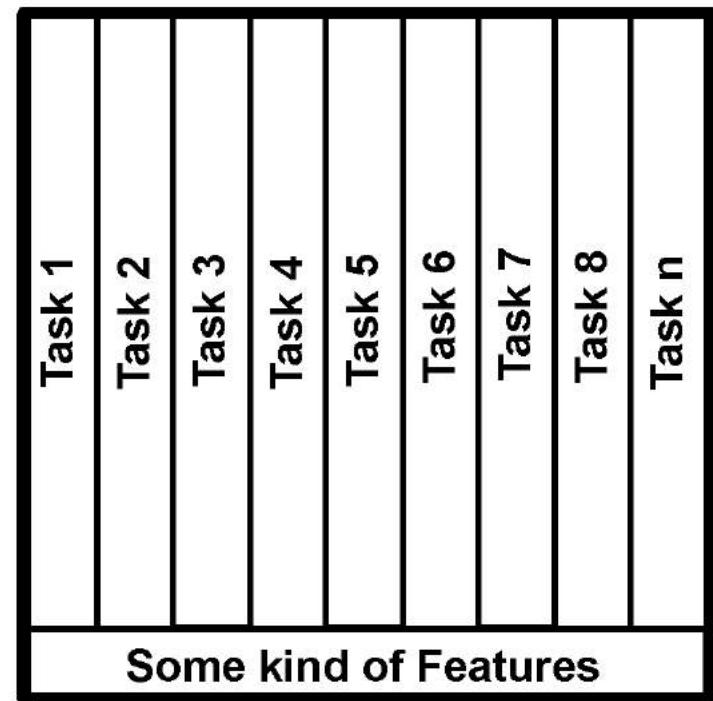


# Flat versus deep Hierarchies

**Deep Hierarchy**



**Flat Hierarchy**



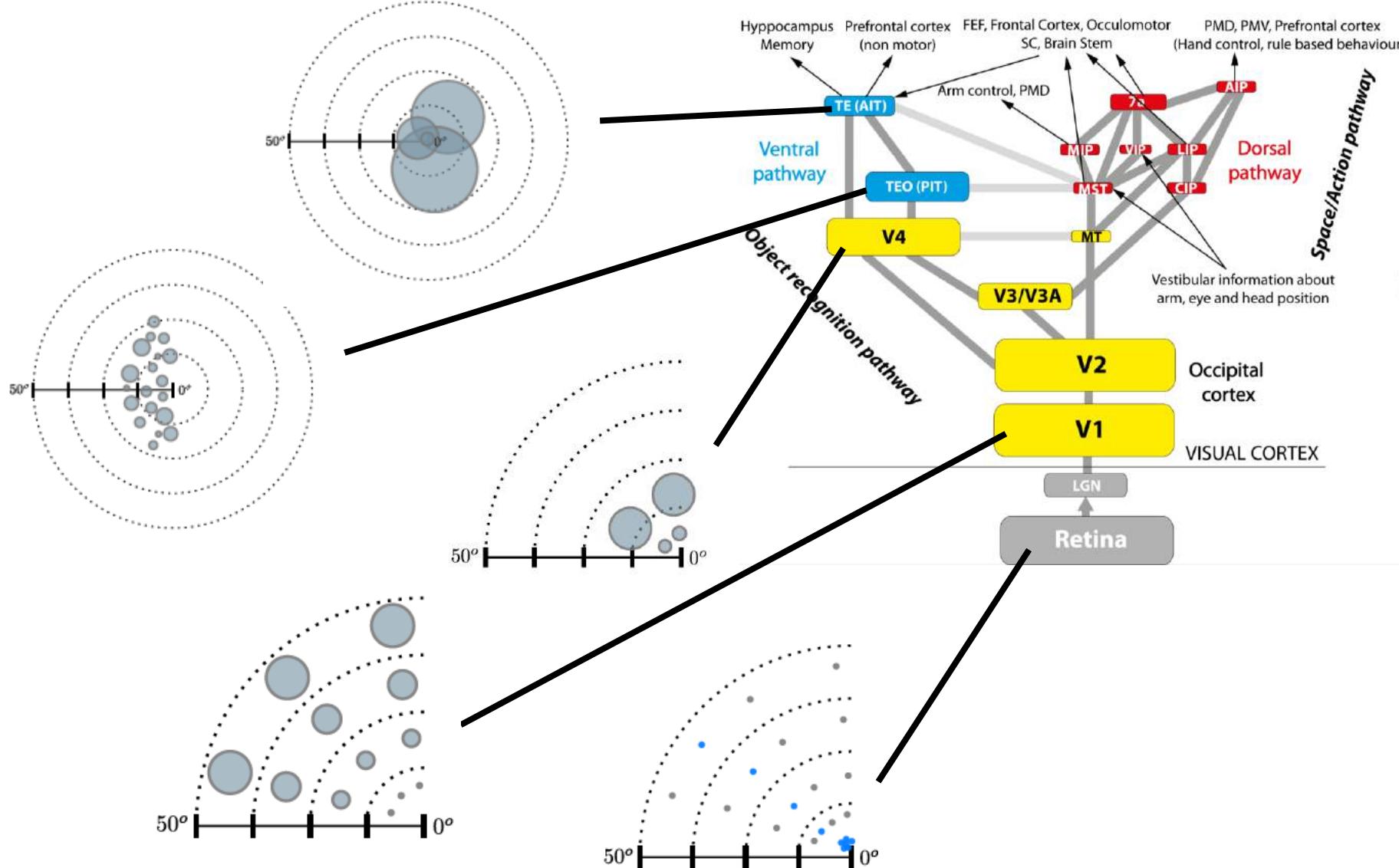
# Example of a flat hierarchy



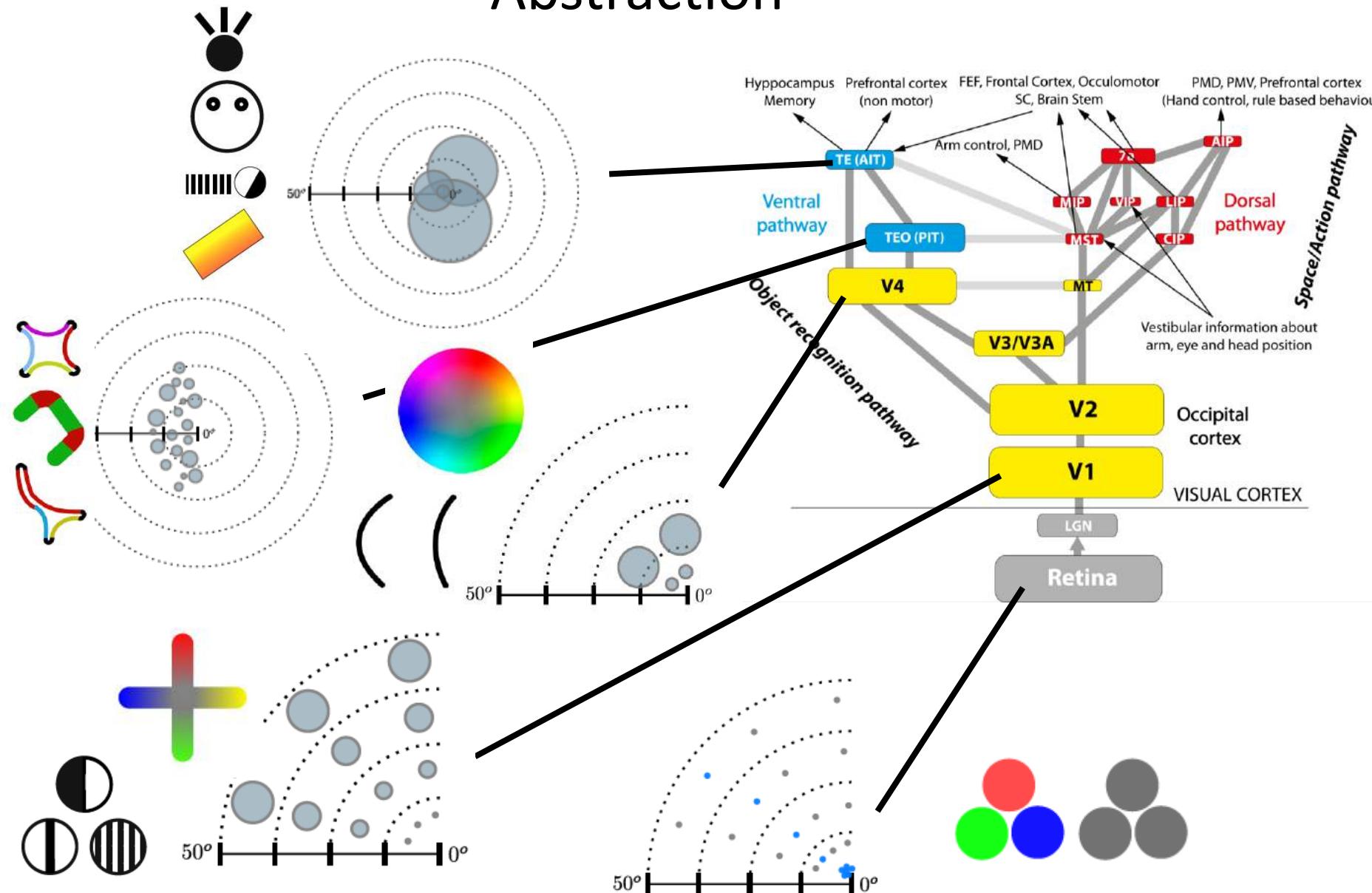
The frog does not seem to see or, at any rate, is not concerned with the detail of stationary parts of the world around him. He will starve to death surrounded by food if it is not moving. His choice of food is determined only by size and movement. He will leap to capture any object the size of an insect or worm, providing it moves like one. He can be fooled easily not only by a bit of dangled meat but by any moving small object. His sex life is conducted by sound and touch. His choice of paths in escaping enemies does not seem to be governed by anything more devious than leaping to where it is darker. Since he is equally at home in water and on land, why should it matter where he lights after jumping or what particular direction he takes? He does remember a moving thing providing it stays within his field of vision and he is not distracted.

J. Y. Lettvin et al. (1959). What the frog's eye tells the frog's brain. Proceedings of the Institute of Radio Engineers

# Increasing Level of Abstraction

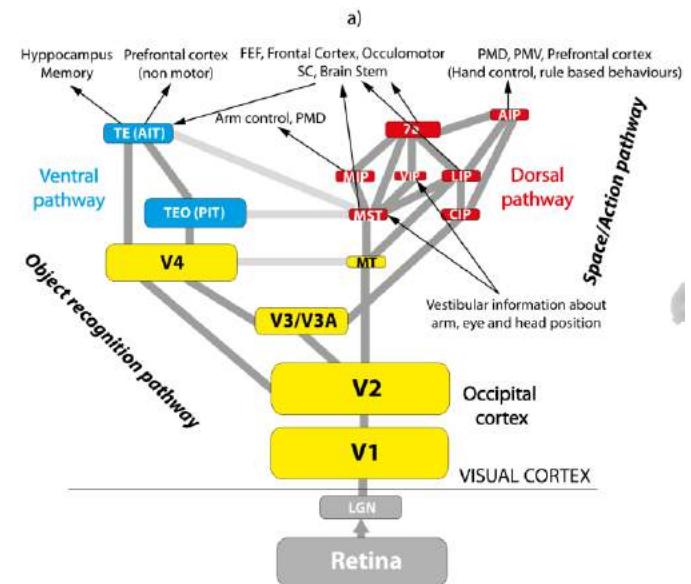
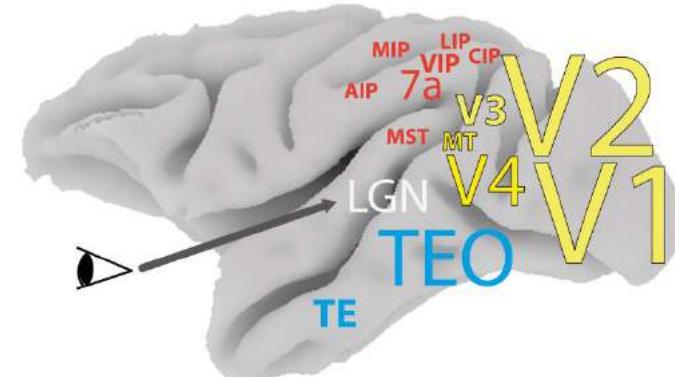


# Increasing Level of Abstraction



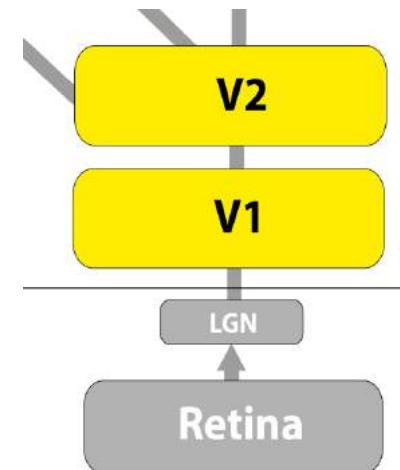
# Flat versus deep hierarchies

- Flat Hiererachies are inefficient
  - No sharing of computational resources
  - Transfer of experience across tasks is facilitated within the same representations

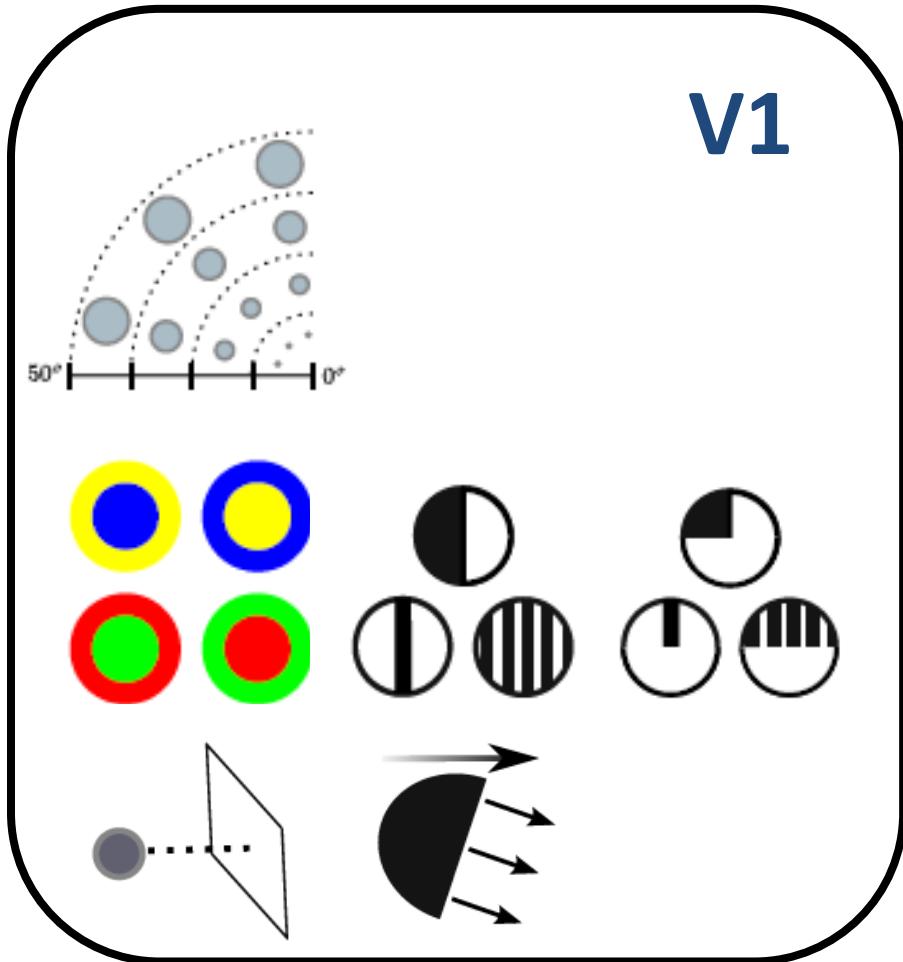


# What can we learn?

- The primate's vision
  - uses a *huge amount of resources* in the brain organized in a *deep hierarchy*
  - **devotes its main vision resources to a *two step computation of a rich set of descriptors in a generic representation***

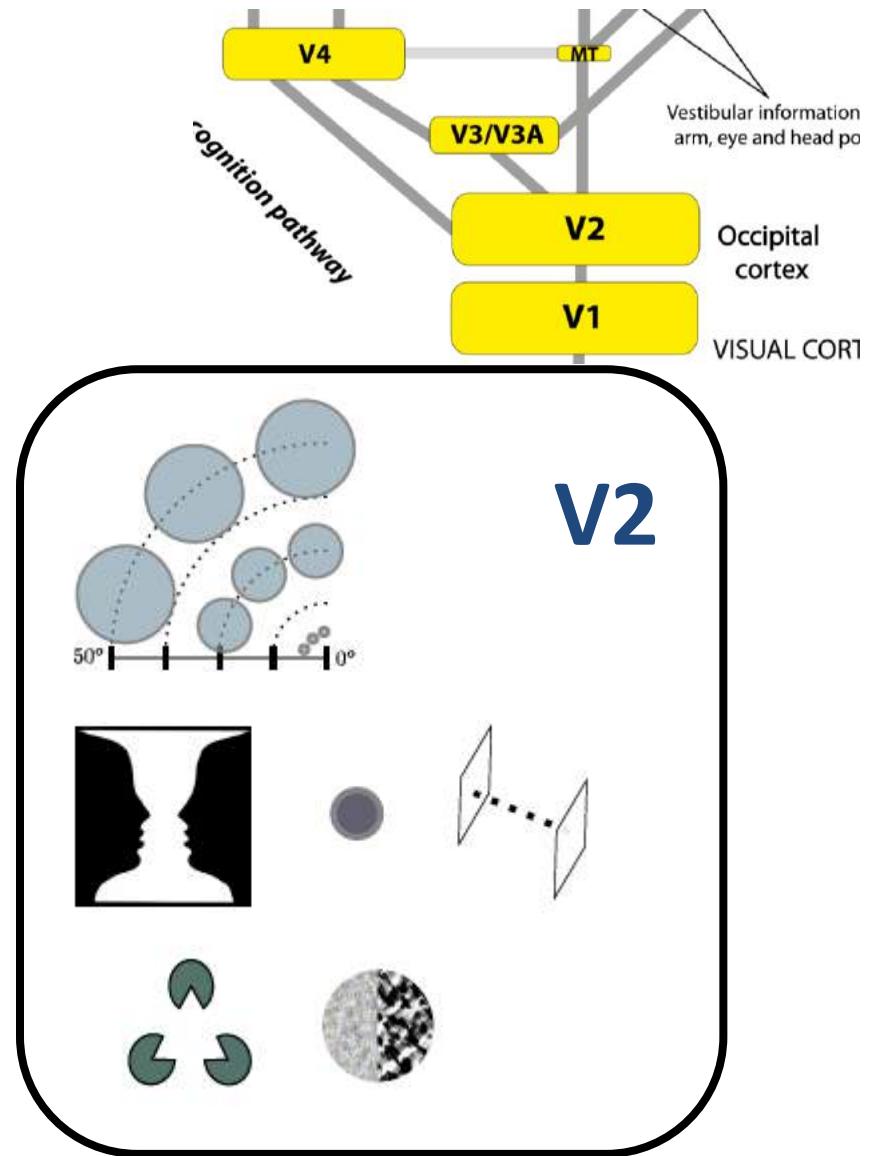


# V1 and V2: A Zoo of Features



V1

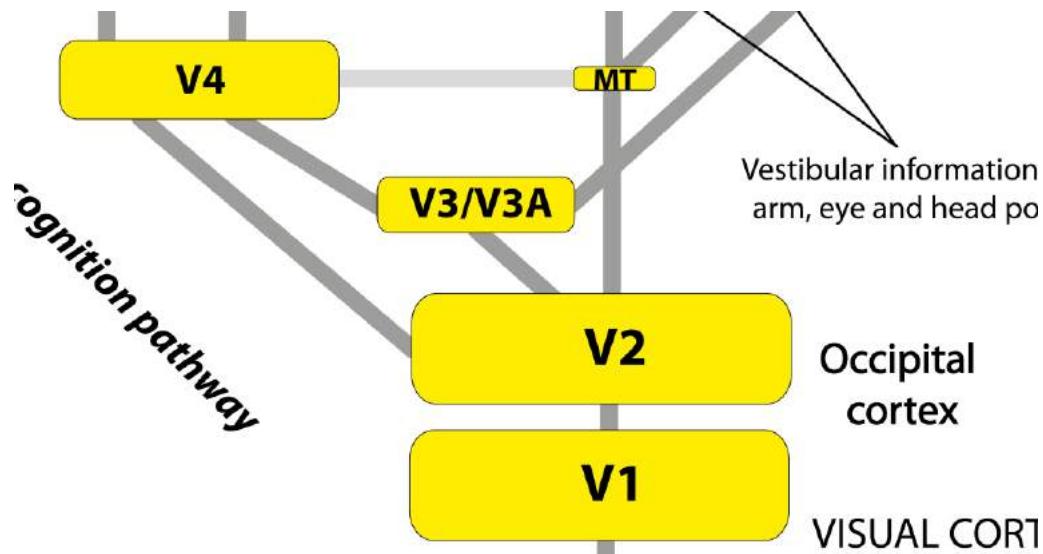
Semi-Global Processing



V2

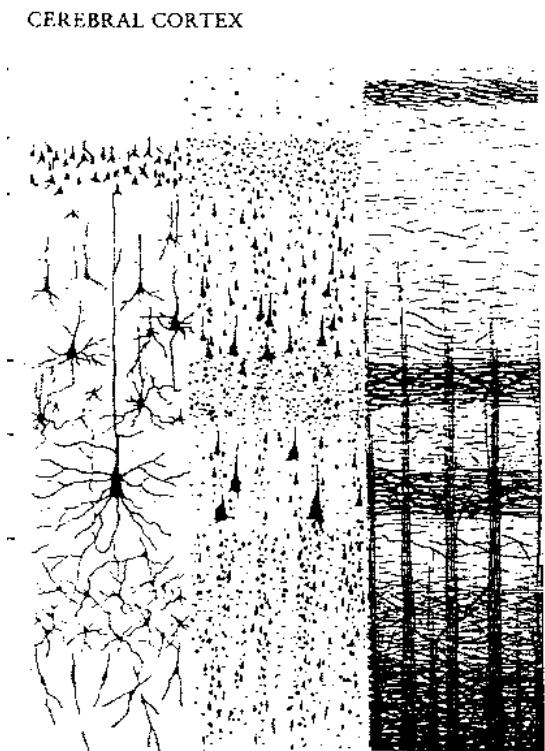
# Perspective

- A stable, rich and disambiguated generic scene representation

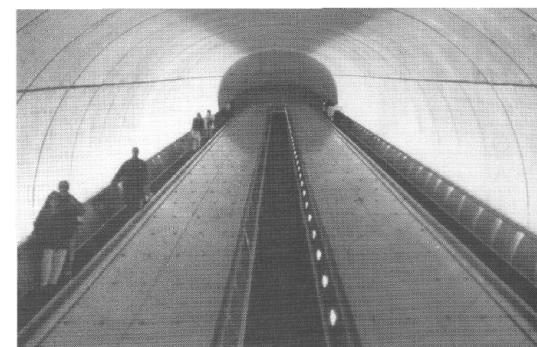
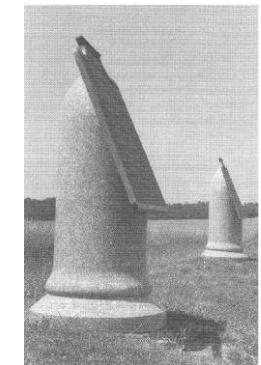
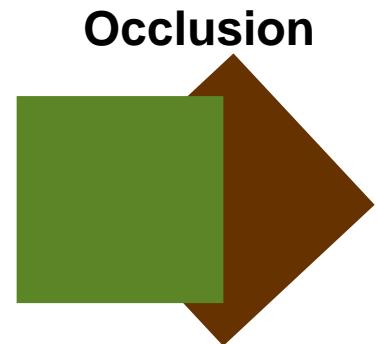
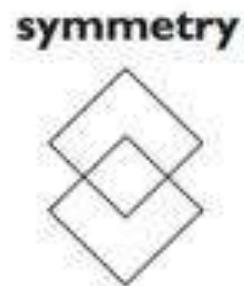
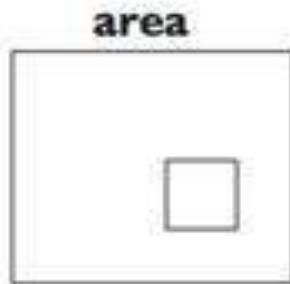
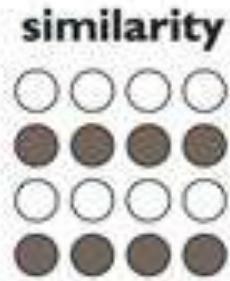
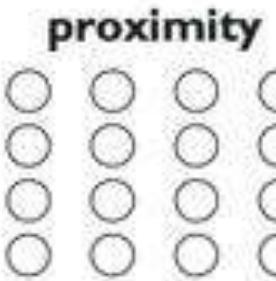


# What does human vision suggest?

- There is evidence that a good part of the local feature processing in V1 works from scratch
  - Hubel and Wiesel, Bonhoefer et al.
- However, lateral connections are probably heavily influenced by statistics of experience
- Gestalt laws and pictorial depth cues are used only after 6 months of development
- Possible Approach
  - Understand how extractable 3D features in structured regions relate to not yet computed 3D structures at homogeneous areas
  - If there are strong conditional probabilities that would be a good sign
  - Integrate semantic scene understanding and depth processing

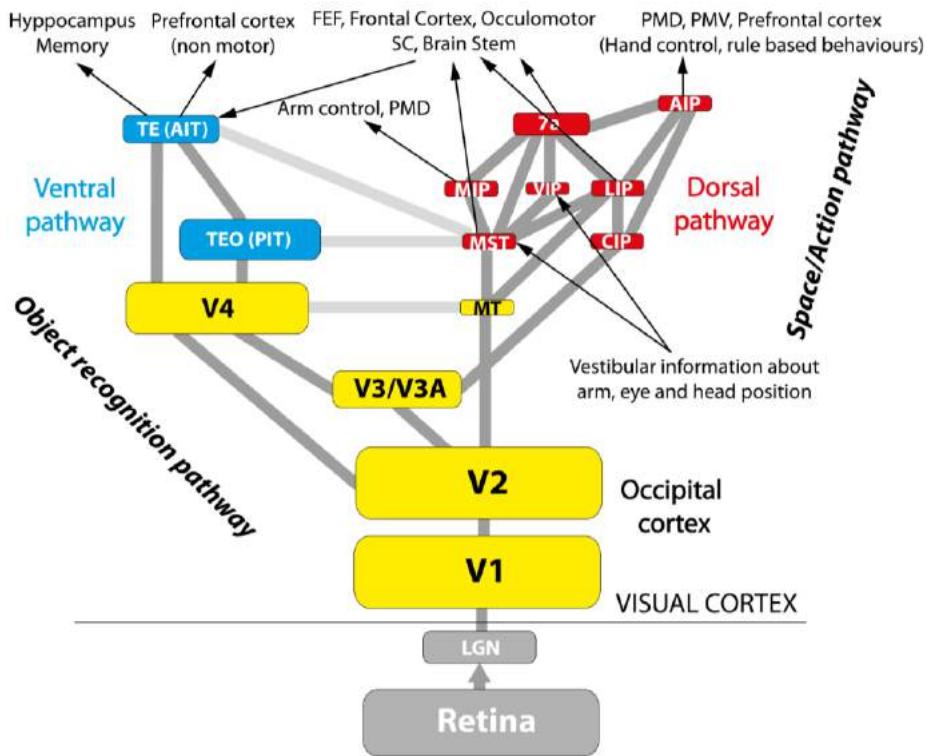


# Gestalt laws and Pictorial Depth Cues



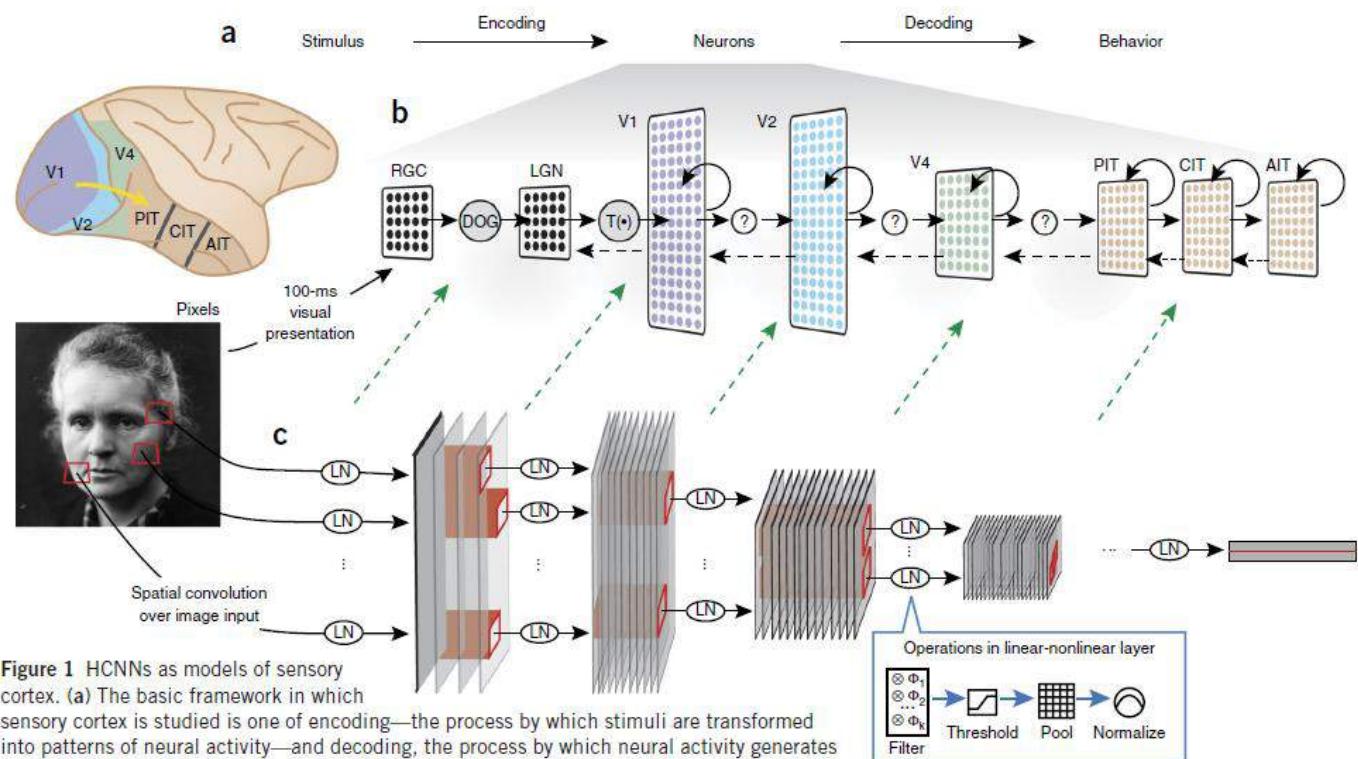
# Conclusion

- The primate's vision system: A deep Hierarchy
- Can we learn something?
- What can we learn?
  - Deep Hierarchies
  - Large resources to generic scene representation
- My personal believe:
  - We need more structure than Deep ANNs provide
- But
  - I might actually be wrong



# **FINAL REMARKS**

# Many similarities between deep learning and our brain



Yamins, Dicarlo, "Using goal-driven deep learning models to understand sensory cortex", Nature Neuroscience, 2016.

# Sensory Coding

- Efficient Coding Hypothesis
  - “The goal of early vision (or, early visual processing) is to provide an efficient representation of the incoming visual signal”
  - (Field, 1987; Hateren, 1998; Bell & Sejnowski, 1996, 1997; Olshausen & Field, 1996; Hyvarinen, 2010)
- For a review & critics:
  - (Simoncelli & Olshausen, 2001; Simoncelli, 2003)

Independent Component Analysis:

$$\text{Image} = s_1 \cdot \text{Filter}_1 + s_2 \cdot \text{Filter}_2 + \dots + s_k \cdot \text{Filter}_k$$

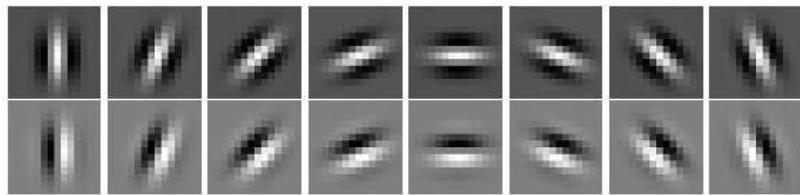
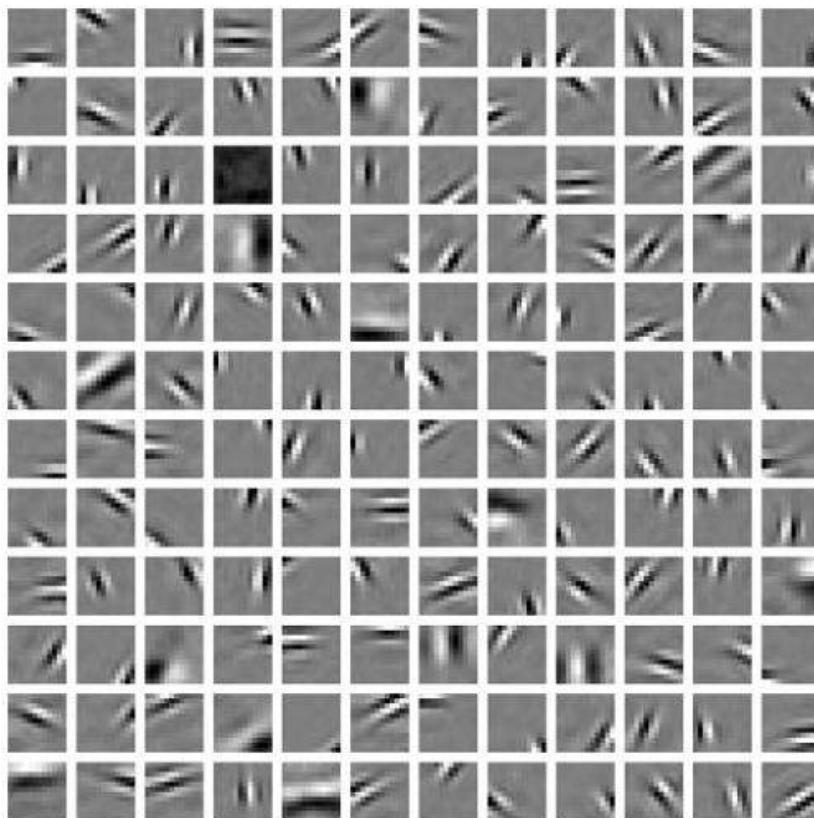


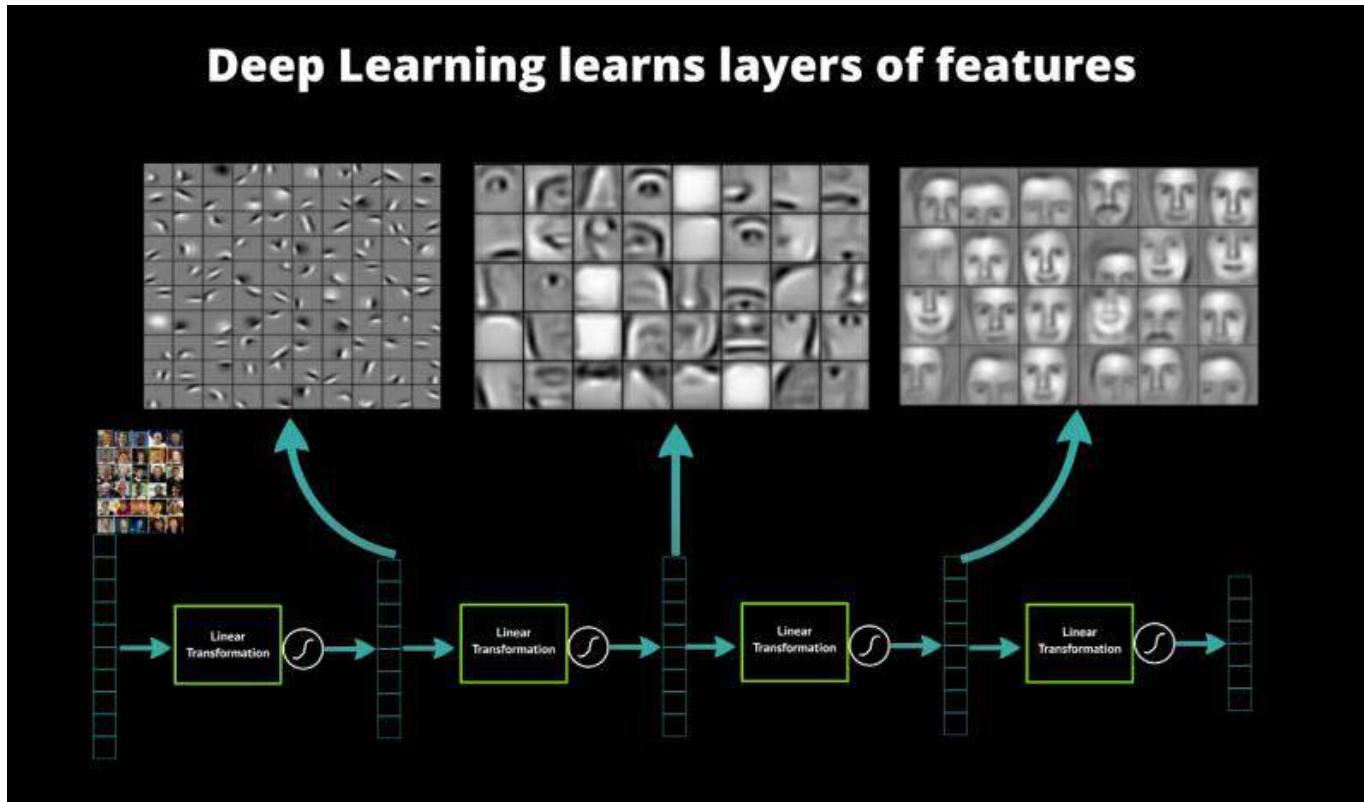
Figure 3: Real (first row) and imaginary (second row) parts of eight orientation Gabor wavelets.  
(Kalkan et al., 2008)



(Olshausen & Field, 1996)

(Hyvarinen, 2010)

# Compare the basis functions of patches with the following

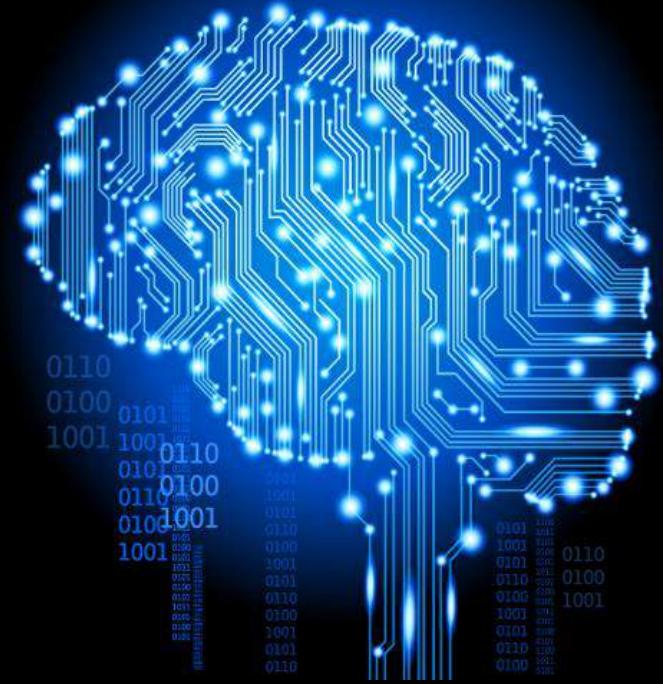


# CENG 783

# Deep Learning

*Week – 7  
Convolutional Neural Networks  
(cont.)*

Sinan Kalkan



© AlchemyAPI



# Today

- CNN operations
  - Convolution
  - Pooling
  - Non-linearity
  - Normalization
- CNN Architectures
- Training a CNN
- Practical issues

# More on connectivity

## Small RF & Stacking

- E.g., 3 CONV layers of 3x3 RFs
- Pros:
  - Same extent for these example figures
  - Non-linearity added with 2<sup>nd</sup> and 3<sup>rd</sup> layers → More expressive! More representational capacity!
  - Less parameters:  $3 \times [(3 \times 3 \times C) \times C] = 27CxC$
- Cons?

## Large RF & Single Layer

- 7x7 RFs of single CONV layer
- Cons:
  - Linear operation in one layer
  - More parameters:  $(7 \times 7 \times C) \times C = 49CxC$
- Pros?

So, we prefer a stack of small filter sizes against big ones

# Convolution layer: How to train?

- How to backpropagate?
  - Calculate error for each neuron, add up gradients but update only one set of weights per slice
- More on this later...

# Implementation Details: Numpy example

- Suppose input is volume is  $X$  of shape (11,11,4)
- Depth slice at depth d (i.e., channel d):  $X[:, :, d]$
- Depth column at position (x,y):  $X[x, y, :]$
- F: 5, P:0 (no padding), S=2
  - Output volume (V) width, height =  $(11-5+0)/2+1 = 4$
- Example computation for some neurons in first channel:

```
V[0,0,0] = np.sum(X[:5,:5,:] * w0) + b0
V[1,0,0] = np.sum(X[2:7,:5,:] * w0) + b0
V[2,0,0] = np.sum(X[4:9,:5,:] * w0) + b0
V[3,0,0] = np.sum(X[6:11,:5,:] * w0) + b0
```

- Note that this is just along one dimension

# Implementation Details: Numpy example

- A second activation map (channel):

```
V[0,0,1] = np.sum(X[:5,:5,:]*W1) + b1
```

```
V[1,0,1] = np.sum(X[2:7,:5,:]*W1) + b1
```

```
V[2,0,1] = np.sum(X[4:9,:5,:]*W1) + b1
```

```
V[3,0,1] = np.sum(X[6:11,:5,:]*W1) + b1
```

```
V[0,1,1] = np.sum(X[:5,2:7,:]*W1) + b1 (example of going along y)
```

```
V[2,3,1] = np.sum(X[4:9,6:11,:]*W1) + b1 (or along both)
```

- To utilize fast processing of matrix multiplications, we will convert each **w x h x d** volume (input & weights) into a single vector using **im2col** function.
  - This leads to redundancy since the receptive fields overlap
  - We can then have just a single dot product: **np.dot(W\_row, X\_col)**
  - The result needs to be shaped back

**Summary.** To summarize, the Conv Layer:

- Accepts a volume of size  $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters  $K$ ,
  - their spatial extent  $F$ ,
  - the stride  $S$ ,
  - the amount of zero padding  $P$ .
- Produces a volume of size  $W_2 \times H_2 \times D_2$  where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$  (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces  $F \cdot F \cdot D_1$  weights per filter, for a total of  $(F \cdot F \cdot D_1) \cdot K$  weights and  $K$  biases.
- In the output volume, the  $d$ -th depth slice (of size  $W_2 \times H_2$ ) is the result of performing a valid convolution of the  $d$ -th filter over the input volume with a stride of  $S$ , and then offset by  $d$ -th bias.

# Unshared convolution

- In some cases, sharing the weights do not make sense
  - When?
- Different parts of the input might require different types of features
- In such a case, we just have a network with local connectivity
- E.g., a face.
  - Features are not repeated across the space.

MULTI-SCALE CONTEXT AGGREGATION BY  
DILATED CONVOLUTIONS

## Dilated Convolution

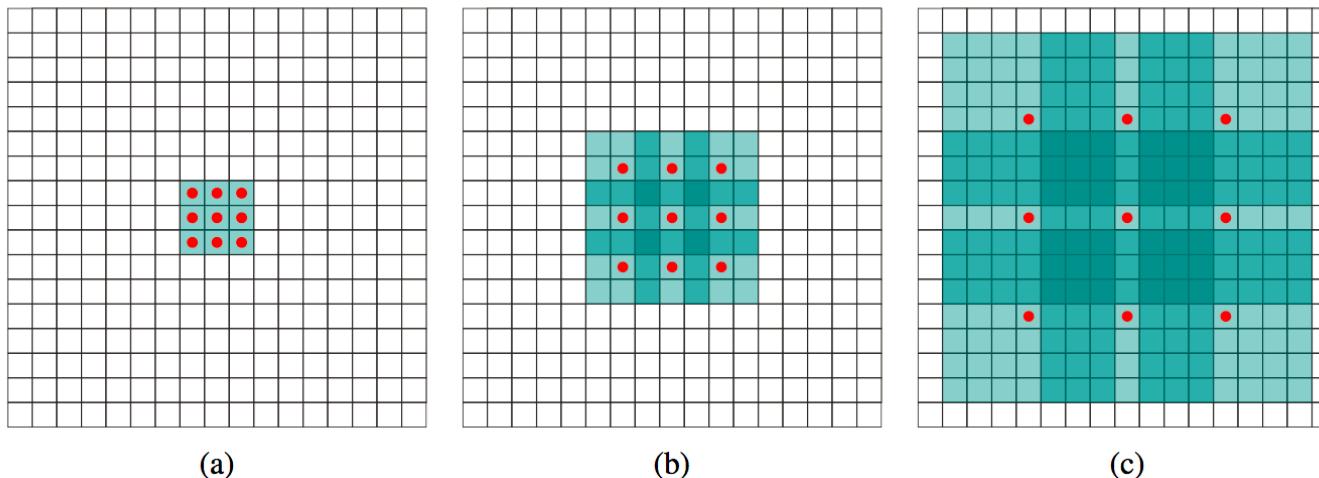
**Fisher Yu**  
Princeton University**Vladlen Koltun**  
Intel Labs

Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a)  $F_1$  is produced from  $F_0$  by a 1-dilated convolution; each element in  $F_1$  has a receptive field of  $3 \times 3$ . (b)  $F_2$  is produced from  $F_1$  by a 2-dilated convolution; each element in  $F_2$  has a receptive field of  $7 \times 7$ . (c)  $F_3$  is produced from  $F_2$  by a 4-dilated convolution; each element in  $F_3$  has a receptive field of  $15 \times 15$ . The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

# Deconvolution

Jonathan Long\*

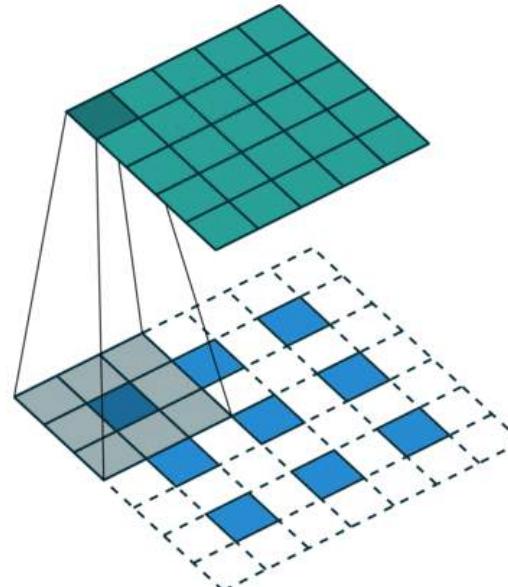
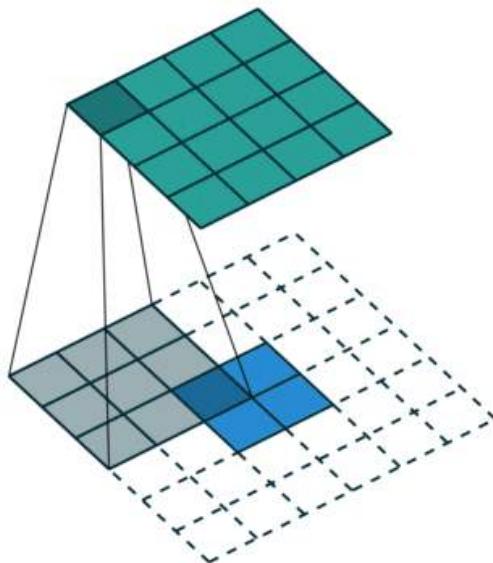
Evan Shelhamer\*

Trevor Darrell

UC Berkeley

{jonlong, shelhamer, trevor}@cs.berkeley.edu

In a sense, upsampling with factor  $f$  is convolution with a *fractional* input stride of  $1/f$ . So long as  $f$  is integral, a natural way to upsample is therefore *backwards convolution* (sometimes called *deconvolution*) with an *output* stride of  $f$ . Such an operation is trivial to implement, since it simply reverses the forward and backward passes of convolution.



# Convolution demos

- [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)
- <http://cs231n.github.io/assets/conv-demo/index.html>
- <https://ezyang.github.io/convolution-visualizer/index.html>

# Efficient convolution

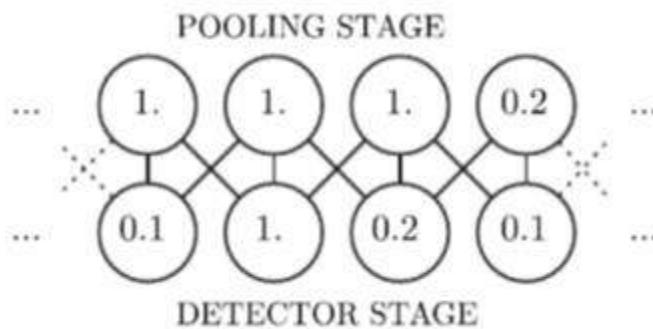
- Convolution in time-domain means multiplication in frequency domain
  - For some problem sizes, converting the signals to the frequency domain, performing multiplication in the frequency domain and transforming them back is more efficient
- A kernel is called separable if it can be written as a product of two vectors:

$$\frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \frac{1}{4} [1 \ 2 \ 1] = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

- Separable kernels can be stored with less memory if you store just the vectors
- Moreover, processing using the vectors is faster than using naïve convolution

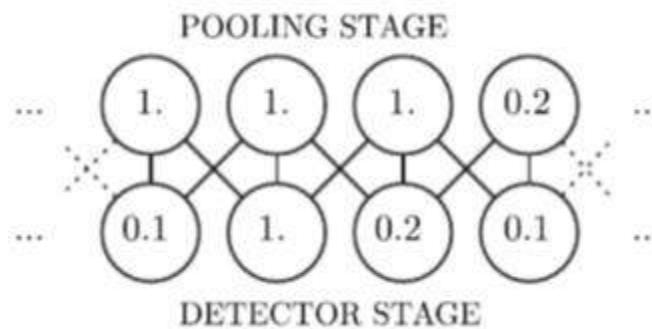
Pooling  
layer / operation

# Pooling

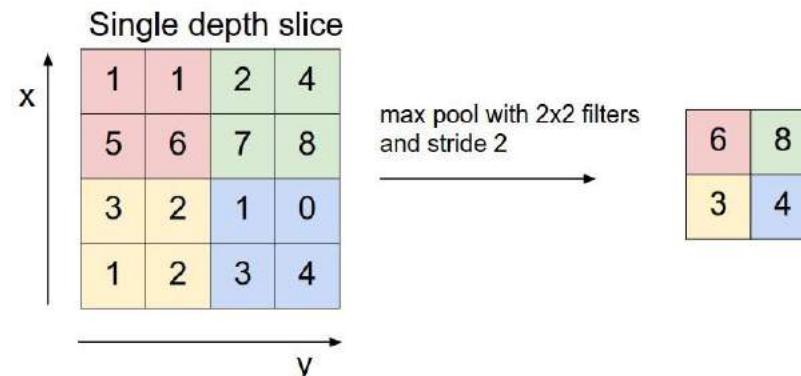
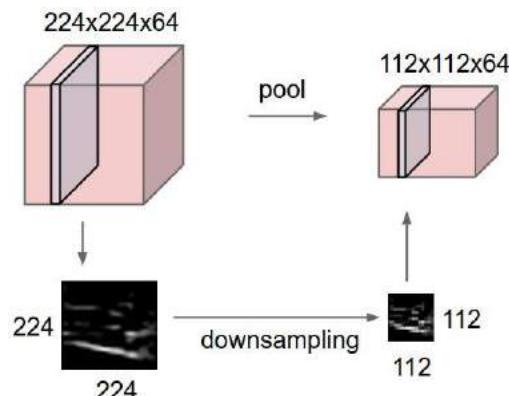


- Apply an **operation** on the “detector” results **to combine or to summarize** the answers of a set of units.
  - Applied to each channel (depth slice) **independently**
  - The operation has to be differentiable of course.
- Alternatives:
  - Maximum
  - Sum
  - Average
  - Weighted average with distance from the center pixel
  - L2 norm
  - Second-order statistics?
  - ...
- Different problems may perform better with different pooling methods
- Pooling can be overlapping or non-overlapping

# Pooling

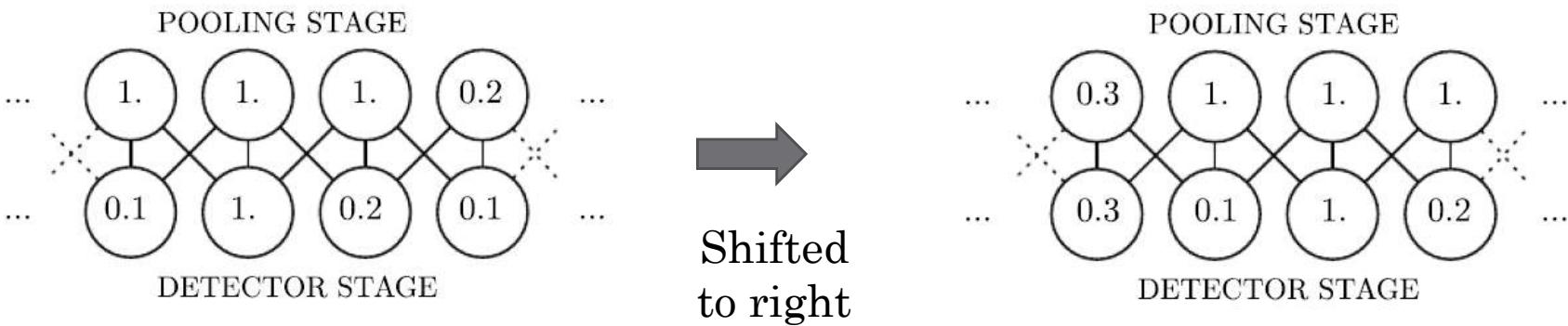


- Example
  - Pooling layer with filters of size 2x2
  - With stride = 2
  - Discards %75 of the activations
  - Depth dimension remains unchanged
- Max pooling with F=3, S=2 or F=2, S=2 are quite common.
  - Pooling with bigger receptive field sizes can be destructive
- Avg pooling is an obsolete choice. Max pooling is shown to work better in practice.

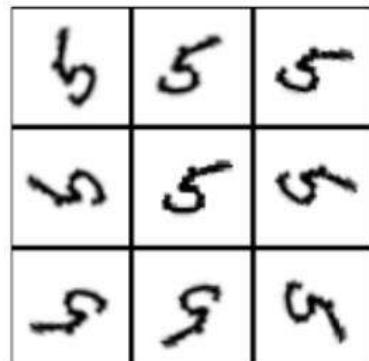


# Pooling

- Pooling provides invariance to **small** translation.

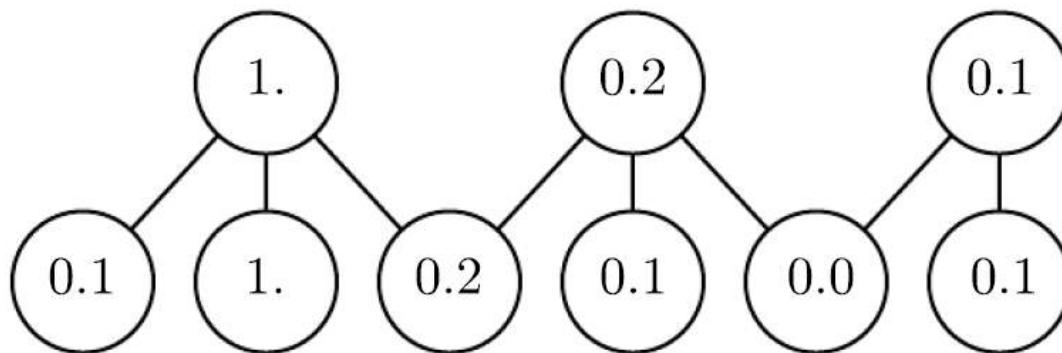


- If you pool over different convolution operators, you can gain invariance to different transformations.



# Pooling can downsample

- Especially needed when to produce an output with fixed-length on varying length input.



Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau *et al.*, 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau *et al.*, 2011). This approach yields a different set of pooling regions for each image. Another approach is to *learn* a single pooling structure that is then applied to all images (Jia *et al.*, 2012).

# Pooling can downsample

- If you want to use the network on images of varying size, you can arrange this with pooling (with the help of convolutional layers)
- E.g., to the last classification layer, provide pooling results of four quadrants in the image.
  - Invariant on the size.

# Backpropagation for pooling

- E.g., for a max operation:
- Remember the derivative of  $\max(x, y)$ :
  - $f(x, y) = \max(x, y)$
  - $\nabla f = [1(x \geq y), 1(y \geq x)]$
- Interpretation: only routing the gradient to the input that had the biggest value in the forward pass.
- This requires that we save the index of the max activation (sometimes also called *the switches*) so that gradient “routing” is handled efficiently during backpropagation.

Model	Error (%)	# parameters
without data augmentation		
Model A	12.47%	$\approx 0.9$ M
Strided-CNN-A	13.46%	$\approx 0.9$ M
ConvPool-CNN-A	10.21%	$\approx 1.28$ M
ALL-CNN-A	10.30%	$\approx 1.28$ M
Model B	10.20%	$\approx 1$ M
Strided-CNN-B	10.98%	$\approx 1$ M
ConvPool-CNN-B	9.33%	$\approx 1.35$ M
ALL-CNN-B	9.10%	$\approx 1.35$ M
Model C	9.74%	$\approx 1.3$ M
Strided-CNN-C	10.19%	$\approx 1.3$ M
ConvPool-CNN-C	9.31%	$\approx 1.4$ M
ALL-CNN-C	9.08%	$\approx 1.4$ M

# Recent developments

- “*Striving for Simplicity: The All Convolutional Net proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while.*”

# For more on pooling

- This is for binary features though.

## A Theoretical Analysis of Feature Pooling in Visual Recognition

Y-Lan Boureau<sup>2,3</sup>

Jean Ponce<sup>1,2</sup>

Yann LeCun<sup>3</sup>

YLAN@CS.NYU.EDU

JEAN.PONCE@ENS.FR

YANN@CS.NYU.EDU

<sup>1</sup>Laboratoire d’Informatique de l’Ecole Normale Supérieure, 45, rue d’Ulm 75230 Paris CEDEX 05, France

<sup>2</sup>INRIA - WILLOW Project (INRIA/ENS/CNRS UMR 8548), 23, avenue d’Italie, 75214 Paris, France

<sup>3</sup>Courant Institute of Mathematical Sciences New York University, NY 10003, USA

### Abstract

Many modern visual recognition algorithms incorporate a step of spatial ‘pooling’, where the outputs of several nearby feature detectors are combined into a local or global ‘bag of features’, in a way that preserves task-related information while removing irrelevant details. Pooling is used to achieve invariance to image transformations, more compact representations, and better robustness to noise and clutter. Several papers have shown that the details of the pooling operation can greatly influence the performance, but studies have so far been purely empirical. In this paper, we show that the reasons underlying the performance of various pooling methods are obscured by several confounding factors, such as the link between the sample cardinality in a spatial pool and the resolution at which low-level features have been extracted. We provide a detailed theoretical analysis of max pooling and average pooling, and give extensive empirical comparisons for object recognition tasks.

### 1. Introduction

Modern computer vision architectures often comprise a spatial pooling step, which combines the responses of feature detectors obtained at nearby locations into some statistic that summarizes the joint distribution of the features over some region of interest. The idea of feature pooling originates in Hubel and Wiesel’s seminal work on complex cells in the visual cortex (1962), and is related to Koenderink’s concept of locally orderless images (1999). Pooling features over a local neighborhood to create invariance to small transformations of the input

is used in a large number of models of visual recognition. The pooling operation is typically a sum, an average, a max, or more rarely some other commutative (i.e., independent of the order of the contributing features) combination rule. Biologically-inspired models of image recognition that use feature pooling include the neocognitron (Fukushima & Miyake, 1982), convolutional networks which use average pooling (LeCun et al., 1989; 1998), or max pooling (Ranzato et al., 2007; Jarrett et al., 2009), the HMAX class of models which uses max pooling (Serre et al., 2005), and some models of the primary visual cortex area V1 (Pinto et al., 2008) which use average pooling. Many popular methods for feature extraction also use pooling, including SIFT (Lowe, 2004), histograms of oriented gradients (HOG) (Dalal & Triggs, 2005) and their variations. In these methods, the dominant gradient orientations are measured in a number of regions, and are pooled over a neighborhood, resulting in a local histogram of orientations. Recent recognition systems often use pooling at a higher level to compute local or global bags of features. This is done by vector-quantizing feature descriptors and by computing the codeword counts over local or global areas (Sivic & Zisserman, 2003; Lazebnik et al., 2006; Zhang et al., 2007; Yang et al., 2009), which is equivalent to average-pooling vectors containing a single 1 at the index of the codeword, and 0 everywhere else (1-of- $k$  codes).

In general terms, the objective of pooling is to transform the joint feature representation into a new, more usable one that preserves important information while discarding irrelevant detail, the crux of the matter being to determine what falls in which category. For example, the assumption underlying the computation of a histogram is that the average feature activation matters, but exact spatial localization does not. Achieving invariance to changes in position or lighting conditions, robustness to clutter, and compactness of representation, are all common goals of pooling.

The success of the spatial pyramid model (Lazebnik et al.,

# Convolution & pooling

- Provide strong bias on the model and the solution
- They directly affect the overall performance of the system

Non-linearity  
Layer

# Non-linearity

- Sigmoid
- Tanh
- ReLU and its variants
  - The common choice
  - Faster
  - Easier (in backpropagation etc.)
  - Avoids saturation issues
- ...

# Normalization Layer

# Normalization layer

- Normalize the responses of neurons
- Necessary especially when the activations are not bounded (e.g., a ReLU unit)
- Replaced by drop-out, batch normalization, better initialization etc.
- Has little impact
- Not favored anymore

- From Krizhevsky et al. (2012):

generalization. Denoting by  $a_{x,y}^i$  the activity of a neuron computed by applying kernel  $i$  at position  $(x, y)$  and then applying the ReLU nonlinearity, the response-normalized activity  $b_{x,y}^i$  is given by the expression

$$b_{x,y}^i = a_{x,y}^i / \left( k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a_{x,y}^j)^2 \right)^\beta$$

where the sum runs over  $n$  “adjacent” kernel maps at the same spatial position, and  $N$  is the total number of kernels in the layer. The ordering of the kernel maps is of course arbitrary and determined before training begins. This sort of response normalization implements a form of lateral inhibition inspired by the type found in real neurons, creating competition for big activities amongst neuron outputs computed using different kernels. The constants  $k$ ,  $n$ ,  $\alpha$ , and  $\beta$  are hyper-parameters whose values are determined using a validation set; we used  $k = 2$ ,  $n = 5$ ,  $\alpha = 10^{-4}$ , and  $\beta = 0.75$ . We

# Fully-Connected Layer

# Fully-connected layer

- At the top of the network for mapping the feature responses to output labels
- Full connectivity
- Can be many layers
- Various activation functions can be used

# FC & Conv Layer Conversion

- CONV to FC conversion:
  - weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).
- FC to CONV conversion:
  - Treat the neurons of the next layer as the different channels.

# CNN Architectures

```
INPUT -> [[CONV -> RELU]*N -> POOL?] *M -> [FC -> RELU]*K -> FC
```

where the `*` indicates repetition, and the `POOL?` indicates an optional pooling layer. Moreover, `N >= 0` (and usually `N <= 3`), `M >= 0`, `K >= 0` (and usually `K < 3`). For example, here are some common ConvNet architectures you may see that follow this pattern:

- `INPUT -> FC`, implements a linear classifier. Here `N = M = K = 0`.
- `INPUT -> CONV -> RELU -> FC`
- `INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC`. Here we see that there is a single CONV layer between every POOL layer.
- `INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC` Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation.

# Demo

<http://scs.ryerson.ca/~aharley/vis/conv/>



# General rule of thumbs: The input layer

- The size of the input layer should be divisible by 2 many times
  - Hopefully a power of 2
- E.g.,
  - 32 (e.g. CIFAR-10),
  - 64,
  - 96 (e.g. STL-10), or
  - 224 (e.g. common ImageNet ConvNets),
  - 384, and 512 etc.

# General rule of thumbs: The conv layer

- Small filters with stride 1
- Usually zero-padding applied to keep the input size unchanged
- In general, for a certain  $F$ , if you choose

$$P = (F - 1)/2,$$

the input size is preserved (for  $S=1$ ):

$$\frac{W - F + 2P}{S} + 1$$

- Number of filters:
  - A convolution channel is more expensive compared to fully-connected layer.
  - We should keep this as small as possible.

# General rule of thumbs: The pooling layer

- Commonly,
  - $F=2$  with  $S=2$
  - Or:  $F=3$  with  $S=2$
- Bigger  $F$  is very destructive

# Taking care of downsampling

- At some point(s) in the network, we need to reduce the size
- If conv layers do not downsize, then only pooling layers take care of downsampling
- If conv layers also downsize, you need to be careful about strides etc. so that
  - (i) the dimension requirements of all layers are satisfied and
  - (ii) all layers tile up properly.
- $S=1$  seems to work well in practice
- However, for bigger input volumes, you may try bigger strides

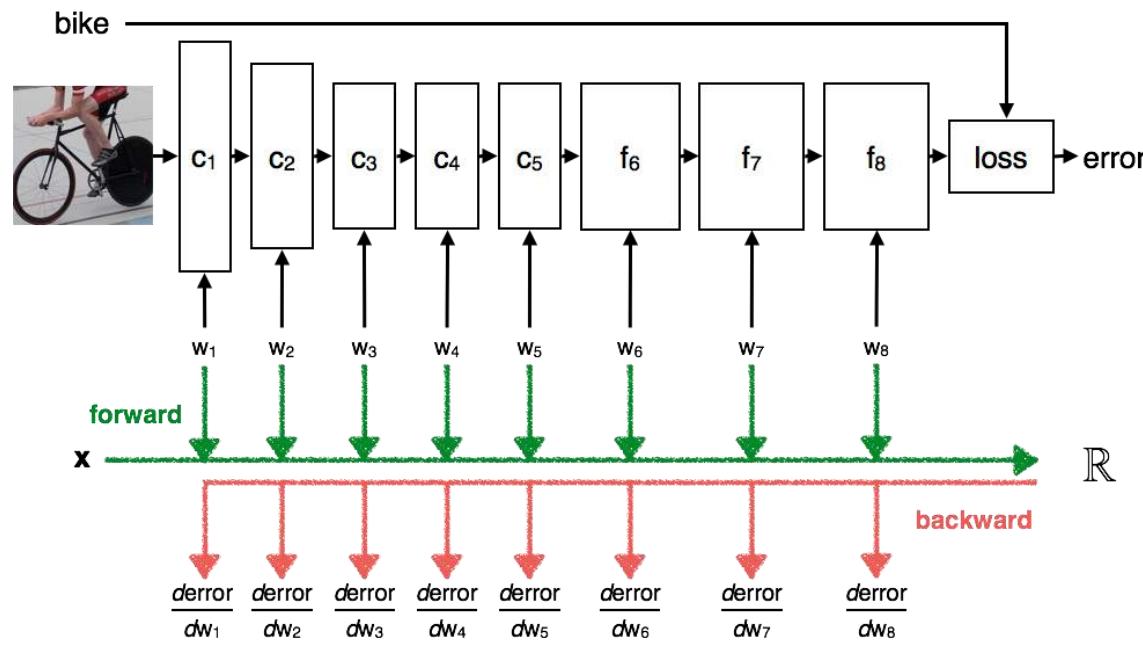


Fig: <http://www.robots.ox.ac.uk/~vgg/practicals/cnn/>

# Training a CNN

# Feed-forward pass

$$x_{ij}^{\ell} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \omega_{ab} y_{(i+a)(j+b)}^{\ell-1}$$

Note that this is written in terms of the weight indices.

Then, let us say that we add an immediate non-linearity after each convolution layer:

$$y_{ij}^{\ell} = \sigma(x_{ij}^{\ell}).$$

# Backpropagation

- Convolution Layer:
  - Calculate the gradient on the weights of the filter summing up over all the expressions that the weights were used in

$$\frac{\partial E}{\partial \omega_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{ij}^\ell} \frac{\partial x_{ij}^\ell}{\partial \omega_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{ij}^\ell} y_{(i+a)(j+b)}^{\ell-1}$$

$$\frac{\partial E}{\partial x_{ij}^\ell} = \frac{\partial E}{\partial y_{ij}^\ell} \frac{\partial y_{ij}^\ell}{\partial x_{ij}^\ell} = \frac{\partial E}{\partial y_{ij}^\ell} \frac{\partial}{\partial x_{ij}^\ell} (\sigma(x_{ij}^\ell)) = \frac{\partial E}{\partial y_{ij}^\ell} \sigma'(x_{ij}^\ell)$$

# Backpropagation

Gradient for the previous layer:

$$\frac{\partial E}{\partial y_{ij}^{\ell-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{(i-a)(j-b)}^\ell} \frac{\partial x_{(i-a)(j-b)}^\ell}{\partial y_{ij}^{\ell-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{(i-a)(j-b)}^\ell} \omega_{ab}$$

Note that we have convolution here

- However, the indices have flipped!

# Backpropagation

- Max pooling and fully-connected layers are straight-forward

# How to initialize the weights?

- Option 1: randomly
  - This has been shown to work nicely in the literature
- Option 2:
  - Train/obtain the “filters” elsewhere and use them as the weights
  - Unsupervised pre-training using image patches (windows)
  - Avoids full feedforward and backward pass, allows the search to start from a better position
  - You may even skip training the convolutional layers

# CONVOLUTIONAL CLUSTERING FOR UNSUPERVISED LEARNING

Aysegul Dundar, Jonghoon Jin, and Eugenio Culurciello

Purdue University, West Lafayette, IN 47907, USA

{adundar, jhjin, euge}@purdue.edu

Table 3: Classification error on MNIST.

(a) Algorithms that learn the filters unsupervised.				
Algorithm	600	1000	3000	All
Zhao et al. (2015) (auto-encoder)	8.4%	6.40%	4.76%	-
Rifai et al. (2011) (contractive auto-encoder)	6.3%	4.77%	3.22%	1.14%
<b>This work (2 layers + multi dict.)</b>	<b>2.8%</b>	<b>2.5%</b>	<b>1.4%</b>	<b>0.5%</b>

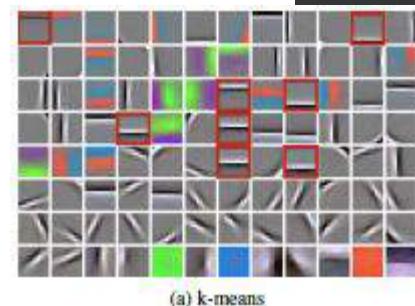
(b) Supervised and semi-supervised algorithms.				
Algorithm	600	1000	3000	All
LeCun et al. (1998) (convnet)	7.68%	6.45%	3.35%	-
Lee (2013) (psuedo-label)	5.03%	3.46%	2.69%	-
Zhao et al. (2015) (semi-supervised auto-encoder)	3.31%	2.83%	2.10%	0.71%
Kingma et al. (2014) (generative models)	2.59%	2.40%	2.18%	0.96%
Rasmus et al. (2015) (semi-supervised ladder)	-	1.0%	-	-

## 3.1 LEARNING FILTERS WITH K-MEANS

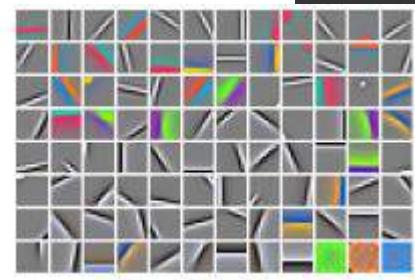
Our method for learning filters is based on the k-means algorithm. The classic k-means algorithm finds cluster centroids that minimize the distance between points in the Euclidean space. In this context, the points are randomly extracted image patches and the centroids are the filters that will be used to encode images. From this perspective, k-means algorithm learns a dictionary  $D \in \mathbb{R}^{n \times k}$  from the data vector  $w^{(i)} \in \mathbb{R}^n$  for  $i = 1, 2, \dots, m$ . The algorithm finds the dictionary as follows:

$$\begin{aligned} s_j^{(i)} &:= \begin{cases} D^{(j)^T} w^{(i)} & \text{if } j = \operatorname{argmax}_l |D^{(l)^T} w^{(i)}|, \\ 0 & \text{otherwise,} \end{cases} \\ D &:= WS^T + D, \\ D^{(j)} &:= \frac{D^{(j)}}{\|D^{(j)}\|_2}, \end{aligned} \tag{1}$$

where  $s^{(i)} \in \mathbb{R}^k$  is the code vector associated with the input  $w^{(i)}$ , and  $D^{(j)}$  is the  $j$ 'th column of the dictionary  $D$ . The matrices  $W \in \mathbb{R}^{n \times m}$  and  $S \in \mathbb{R}^{k \times m}$  have the columns  $w^{(i)}$  and  $s^{(i)}$ , respectively.  $w^{(i)}$ 's are randomly extracted patches from input images that have the same dimension as the dictionary vectors,  $D^{(j)}$ .



(a) k-means



(b) convolutional k-means

# Now

- More on CNNs
  - Memory constraints
  - Design choices
  - Visualizing and understanding CNNs
  - Popular CNN models
  - CNN applications

# Memory & performance considerations

# Memory

- Main sources of memory load:
- Activation maps:
  - Training: They need to be kept during training so that backpropagation can be performed
  - Testing: No need to keep the activations of earlier layers
- Parameters:
  - The weights, their gradients and also another copy if momentum is used
- Data:
  - The originals + their augmentations
- If all these don't fit into memory,
  - Use batches
  - Decrease the size of your batches

# Memory constraints

- Using smaller RFs with stacking means more memory since you need to store more activation maps
- In such memory-scarce cases,
  - the first layer may use bigger RFs with  $S > 1$
  - information loss from the input volume may be less critical than the following layers
- E.g., AlexNet uses RFs of  $11 \times 11$  and  $S = 4$  for the first layer.

# Trade-offs in architecture

- Between filter size and number of layers
  - [Keep the layer widths fixed.](#)
  - Deeper networks with smaller filter sizes perform better (if you keep the overall computational complexity fixed)
- Between layer width and number of layers
  - [Keep the size of the filters fixed.](#)
  - Increase the number of layers or the depth
  - Increasing depth improves performance
- Between filter size and layer width
  - [Keep the number of layers fixed.](#)
  - No significant difference

---

This CVPR2015 paper is the Open Access version, provided by the Computer Vision Foundation.  
The authoritative version of this paper is available in IEEE Xplore.

Convolutional Neural Networks at Constrained Time Cost

Kaiming He

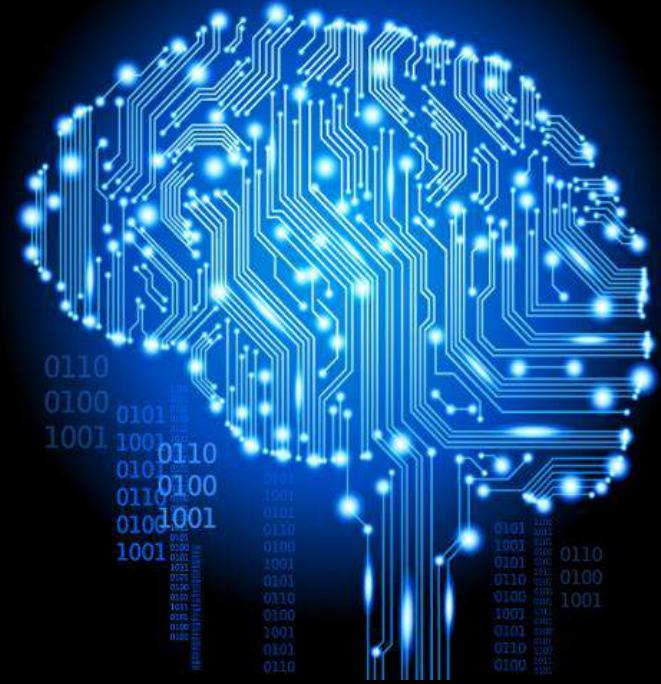
Jian Sun  
Microsoft Research  
[{kahe,jiansun}@microsoft.com](mailto:{kahe,jiansun}@microsoft.com)

# CENG 783

# Deep Learning

*Week – 9  
Convolutional Neural Networks  
(cont.)*

Sinan Kalkan



© AlchemyAPI



# Today

- Transfer learning: Using a trained CNN
- Visualizing and understanding CNNs
- Popular CNN models
- CNN applications

Transfer learning:  
using trained CNN  
& fine-tuning

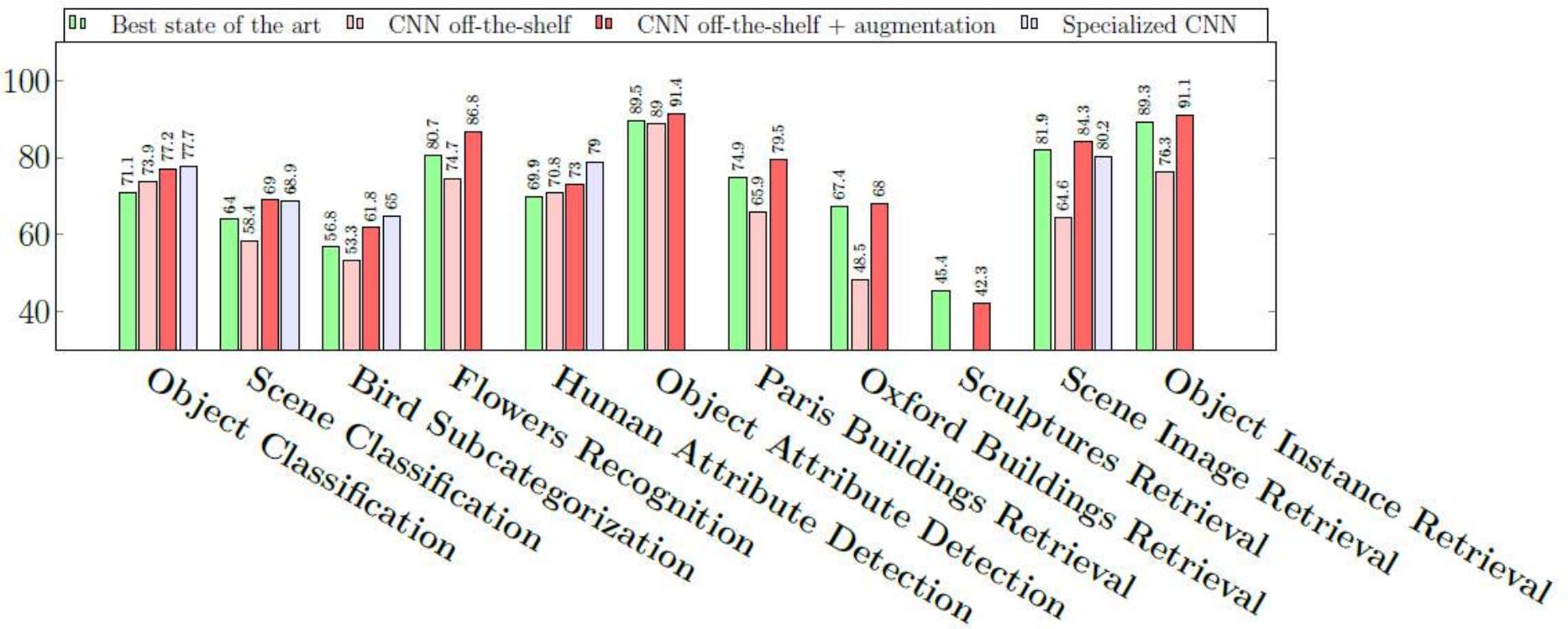
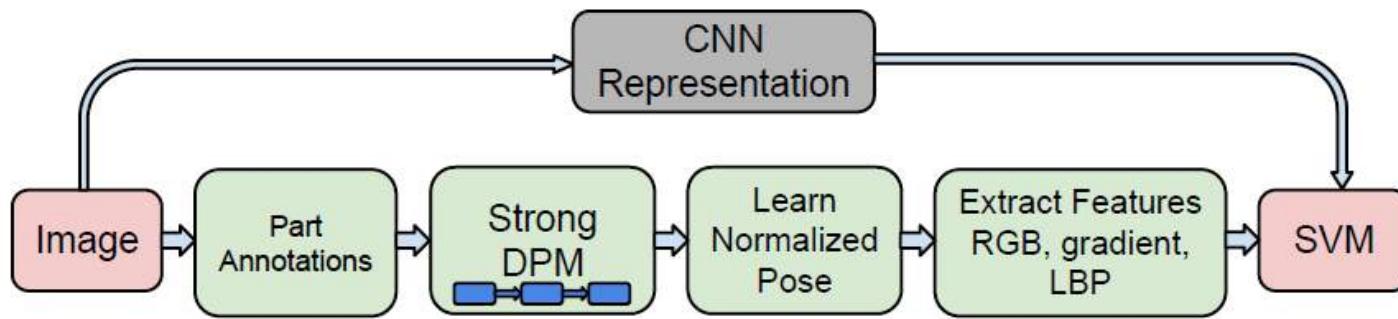
# Using trained CNN

- Also called transfer learning
  - Rare to train a CNN from scratch!
- Take a trained CNN, e.g., AlexNet
  - Use a trained CNN as a feature detector:
    - Remove the last fully-connected layer
    - The activations of the remaining layer are called CNN codes
    - This yields a 4096 dimensional feature vector for AlexNet
    - Now, add a fully-connected layer for your problem and train a linear classifier on your dataset.
  - Alternatively, fine-tune the whole network with your new layer and outputs
    - You may limit updating only to the last layers because earlier layers are generic, and quite dataset independent
- Pre-trained CNNs:
  - Model Zoo of Caffe: <https://github.com/BVLC/caffe/wiki/Model-Zoo>

# CNN Features off-the-shelf: an Astounding Baseline for Recognition

Ali Sharif Razavian Hossein Azizpour Josephine Sullivan Stefan Carlsson  
CVAP, KTH (Royal Institute of Technology)  
Stockholm, Sweden  
[{razavian,azizpour,sullivan,stefanc}@csc.kth.se](mailto:{razavian,azizpour,sullivan,stefanc}@csc.kth.se)

2014



# Finetuning

1. If the new dataset is **small** and **similar** to the original dataset used to train the CNN:
  - Finetuning the whole network may lead to overfitting
  - **Just train the newly added layer**
2. If the new dataset is **big** and **similar** to the original dataset:
  - The more, the merrier: go ahead and **train the whole network**
3. If the new dataset is **small** and **different** from the original dataset:
  - Not a good idea to train the whole network
  - However, add your new layer not to the top of the network, since those parts are very dataset (problem) specific
  - **Add your layer to earlier parts of the network**
4. If the new dataset is **big** and **different** from the original dataset:
  - We can “**finetune**” the **whole network**
  - This amounts to a new training problem by initializing the weights with those of another network

# More on finetuning

- You cannot change the architecture of the trained network (e.g., remove layers) **arbitrarily**
- The **sizes** of the layers can be varied
  - For convolution & pooling layers, this is straightforward
  - For the fully-connected layers: you can convert the fully-connected layers to convolution layers, which makes it size-independent.
- You should use small learning rates while fine-tuning

# See also:

Preprint release. Full citation: Yosinski J, Clune J, Bengio Y, and Lipson H. *How transferable are features in deep neural networks?* In Advances in Neural Information Processing Systems 27 (NIPS '14), NIPS Foundation, 2014.

---

## How transferable are features in deep neural networks?

---

Jason Yosinski,<sup>1</sup> Jeff Clune,<sup>2</sup> Yoshua Bengio,<sup>3</sup> and Hod Lipson<sup>4</sup>

<sup>1</sup> Dept. Computer Science, Cornell University

<sup>2</sup> Dept. Computer Science, University of Wyoming

<sup>3</sup> Dept. Computer Science & Operations Research, University of Montreal

<sup>4</sup> Dept. Mechanical & Aerospace Engineering, Cornell University

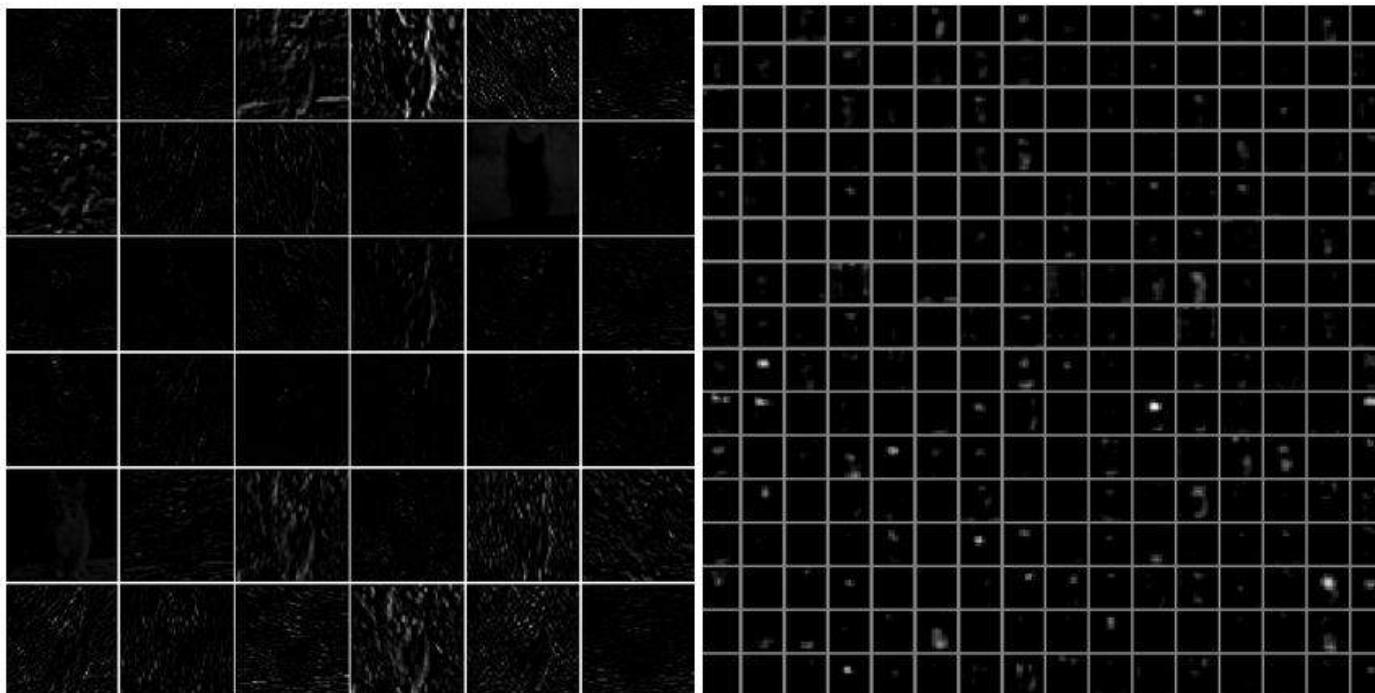
# Visualizing and Understanding CNNs

# Many different mechanisms

- Visualize layer activations
- Visualize the weights (i.e., filters)
- Visualize examples that maximally activate a neuron
- Visualize a 2D embedding of the inputs based on their CNN codes
- Occlude parts of the window and see how the prediction is affected
- Data gradients

# Visualize activations during training

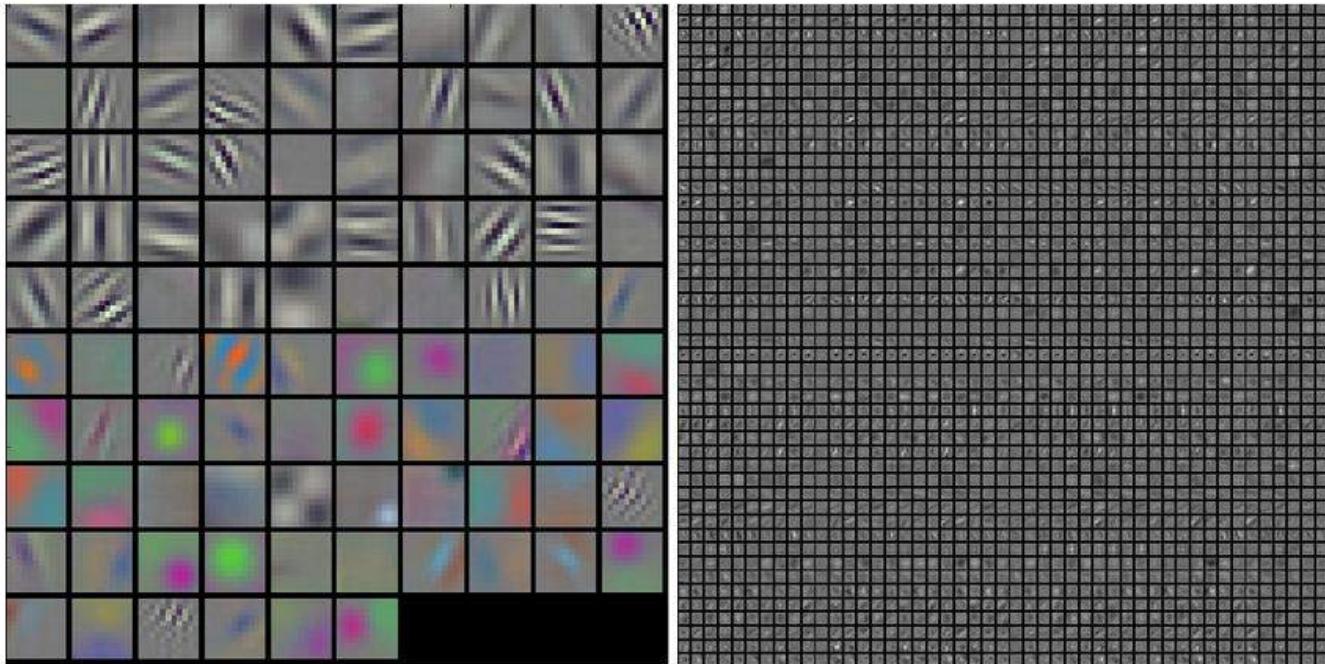
- Activations are dense at the beginning.
  - They should get sparser during training.
- If some activation maps are all zero for many inputs, dying neuron problem => high learning rate in the case of ReLUs.



Typical-looking activations on the first CONV layer (left), and the 5th CONV layer (right) of a trained AlexNet looking at a picture of a cat. Every box shows an activation map corresponding to some filter. Notice that the activations are sparse (most values are zero, in this visualization shown in black) and mostly local.

# Visualize the weights

- We can directly look at the filters of all layers
- First layer is easier to interpret
- Filters shouldn't look noisy



Typical-looking filters on the first CONV layer (left), and the 2nd CONV layer (right) of a trained AlexNet. Notice that the first-layer weights are very nice and smooth, indicating nicely converged network. The color/grayscale features are clustered because the AlexNet contains two separate streams of processing, and an apparent consequence of this architecture is that one stream develops high-frequency grayscale features and the other low-frequency color features. The 2nd CONV layer weights are not as interpretable, but it is apparent that they are still smooth, well-formed, and absent of noisy patterns.

# Visualize the inputs that maximally activate a neuron

- Keep track of which images activate a neuron most

Rich feature hierarchies for accurate object detection and semantic segmentation  
Tech report (v5)

Ross Girshick Jeff Donahue Trevor Darrell Jitendra Malik  
UC Berkeley  
`{rbg, jdonahue, trevor, malik}@eecs.berkeley.edu`



Maximally activating images for some POOL5 (5th pool layer) neurons of an AlexNet. The activation values and the receptive field of the particular neuron are shown in white. (In particular, note that the POOL5 neurons are a function of a relatively large portion of the input image!) It can be seen that some neurons are responsive to upper bodies, text, or specular highlights.

# Embed the codes in a lower-dimensional space

- Place images into a 2D space such that images which produce similar CNN codes are placed close.
- You can use, e.g., t-Distributed Stochastic Neighbor Embedding (t-SNE)



t-SNE embedding of a set of images based on their CNN codes. Images that are nearby each other are also close in the CNN representation space, which implies that the CNN "sees" them as being very similar. Notice that the similarities are more often class-based and semantic rather than pixel and color-based. For more details on how this visualization was produced the associated code, and more related visualizations at different scales refer to [t-SNE visualization of CNN codes](#).

# Occlude parts of the image

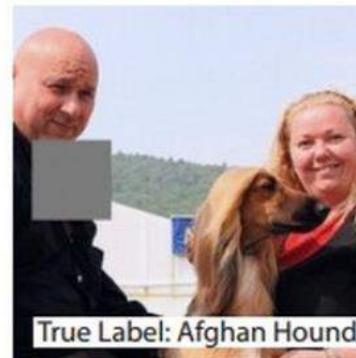
- Slide an “occlusion window” over the image
- For each occluded image, determine the class prediction confidence/probability.



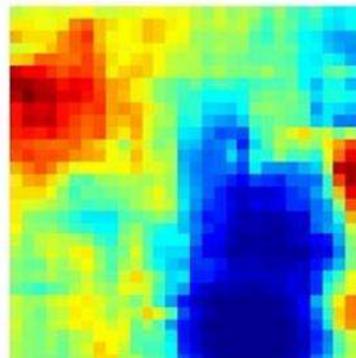
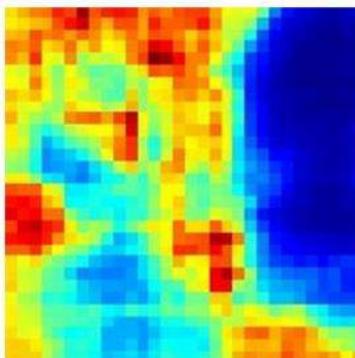
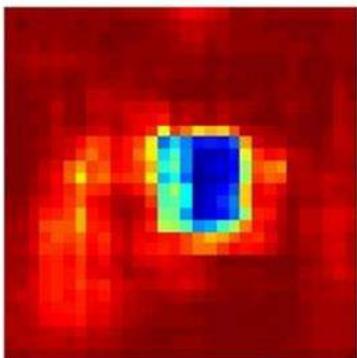
True Label: Pomeranian



True Label: Car Wheel



True Label: Afghan Hound



Three input images (top). Notice that the occluder region is shown in grey. As we slide the occluder over the image we record the probability of the correct class and then visualize it as a heatmap (shown below each image). For instance, in the left-most image we see that the probability of Pomeranian plummets when the occluder covers the face of the dog, giving us some level of confidence that the dog's face is primarily responsible for the high classification score. Conversely, zeroing out other parts of the image is seen to have relatively negligible impact.

# Data gradients

- Option 1: Generate an image that maximizes the class score.

More formally, let  $S_c(I)$  be the score of the class  $c$ , computed by the classification layer of the ConvNet for an image  $I$ . We would like to find an  $L_2$ -regularised image, such that the score  $S_c$  is high:

$$\arg \max_I S_c(I) - \lambda \|I\|_2^2, \quad (1)$$

where  $\lambda$  is the regularisation parameter. A locally-optimal  $I$  can be found by the back-propagation

- Use: Gradient ascent!

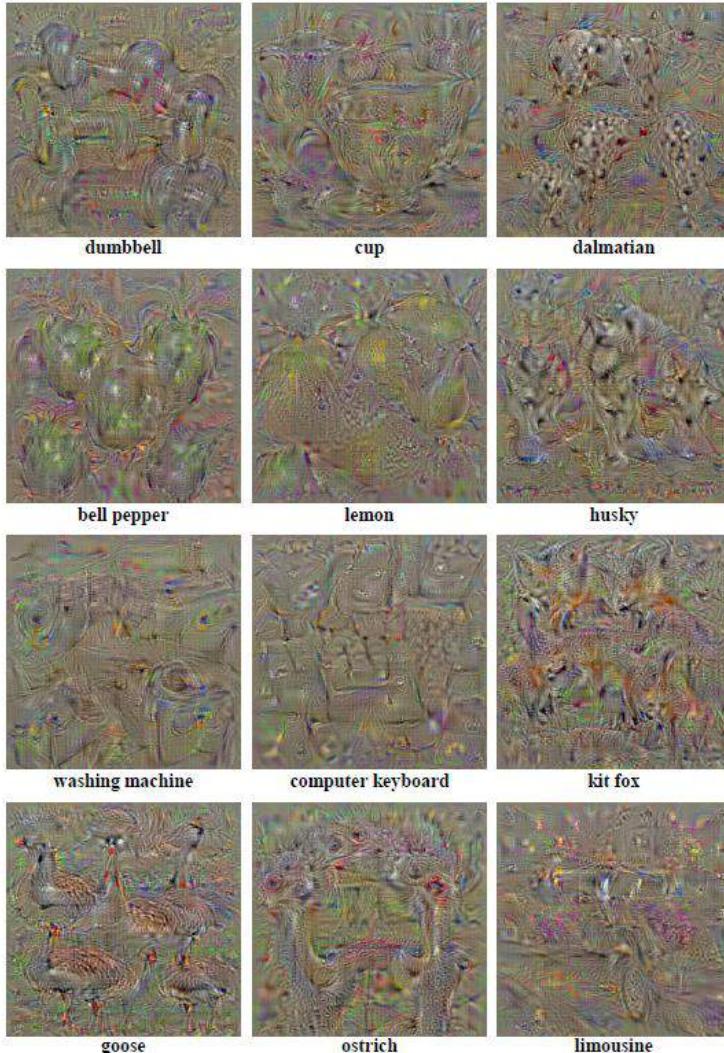


Figure 1: Numerically computed images, illustrating the class appearance models, learnt by a ConvNet, trained on ILSVRC-2013. Note how different aspects of class appearance are captured in a single image. Better viewed in colour.

---

## Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

---

Karen Simonyan

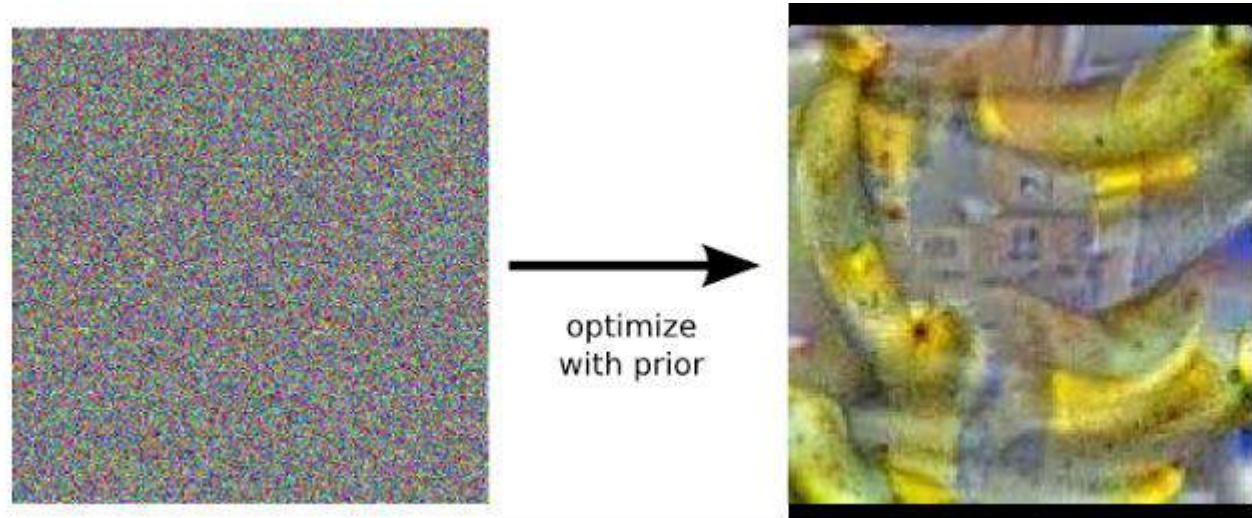
Visual Geometry Group, University of Oxford  
{karen, vedaldi, az}@robots.ox.ac.uk

Andrea Vedaldi

Andrew Zisserman

2014

# DeepDream



# Data gradients

- The gradient with respect to the input is high for pixels which are on the object

We start with a motivational example. Consider the linear score model for the class  $c$ :

$$S_c(I) = w_c^T I + b_c, \quad (2)$$

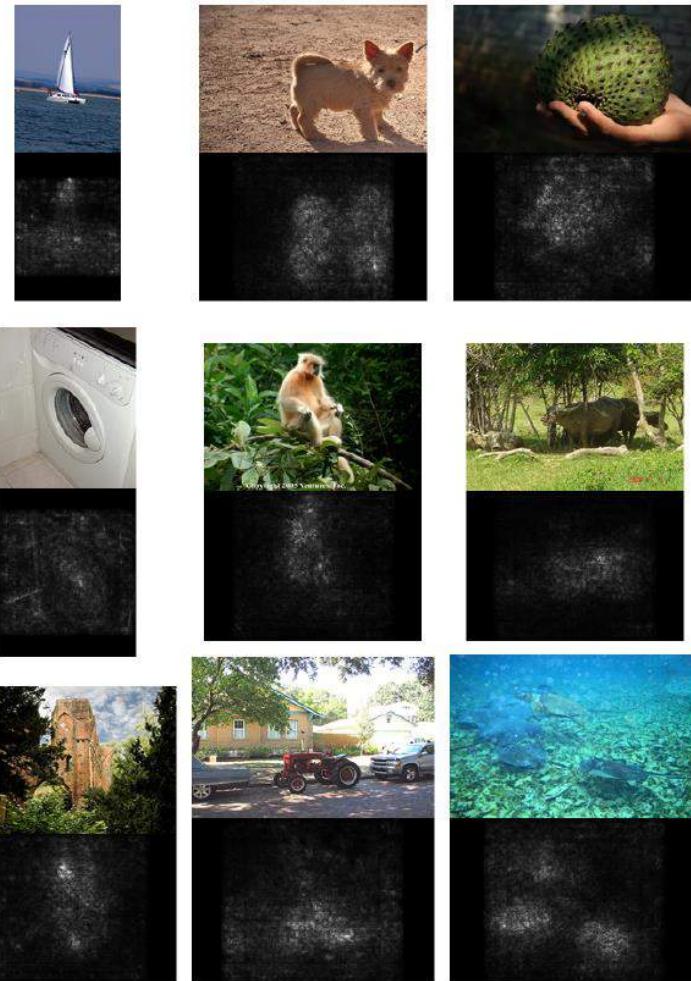
where the image  $I$  is represented in the vectorised (one-dimensional) form, and  $w_c$  and  $b_c$  are respectively the weight vector and the bias of the model. In this case, it is easy to see that the magnitude of elements of  $w$  defines the importance of the corresponding pixels of  $I$  for the class  $c$ .

In the case of deep ConvNets, the class score  $S_c(I)$  is a highly non-linear function of  $I$ , so the reasoning of the previous paragraph can not be immediately applied. However, given an image  $I_0$ , we can approximate  $S_c(I)$  with a linear function in the neighbourhood of  $I_0$  by computing the first-order Taylor expansion:

$$S_c(I) \approx w^T I + b, \quad (3)$$

where  $w$  is the derivative of  $S_c$  with respect to the image  $I$  at the point (image)  $I_0$ :

$$w = \frac{\partial S_c}{\partial I} \Big|_{I_0}. \quad (4)$$



---

## Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

---

# Feature inversion

- Learns to reconstruct an image from its representation

This section introduces our method to compute an approximate inverse of an image representation. This is formulated as the problem of finding an image whose representation best matches the one given [34]. Formally, given a representation function  $\Phi : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}^d$  and a representation  $\Phi_0 = \Phi(\mathbf{x}_0)$  to be inverted, reconstruction finds the image  $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$  that minimizes the objective:

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^{H \times W \times C}}{\operatorname{argmin}} \ell(\Phi(\mathbf{x}), \Phi_0) + \lambda \mathcal{R}(\mathbf{x}) \quad (1)$$

where the loss  $\ell$  compares the image representation  $\Phi(\mathbf{x})$  to the target one  $\Phi_0$  and  $\mathcal{R} : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}$  is a regulariser capturing a *natural image prior*.

- Regularization term here is the key factor

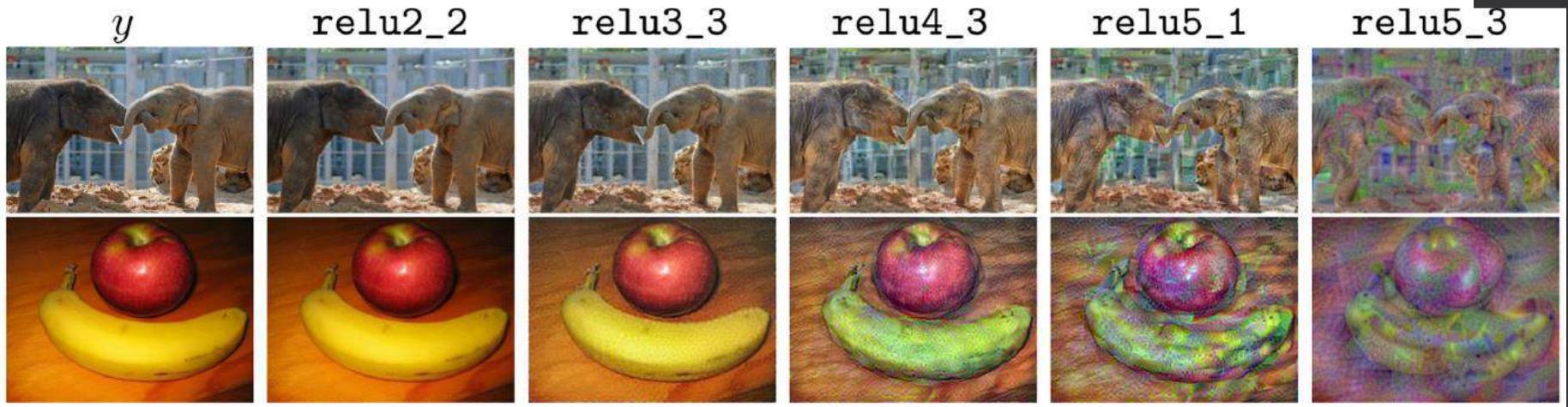
## Understanding Deep Image Representations by Inverting Them

Aravindh Mahendran  
University of Oxford

Andrea Vedaldi  
University of Oxford



Figure 1. **What is encoded by a CNN?** The figure shows five possible reconstructions of the reference image obtained from the 1,000-dimensional code extracted at the penultimate layer of a reference CNN[13] (before the softmax is applied) trained on the ImageNet data. From the viewpoint of the model, all these images are practically equivalent. This image is best viewed in color/screen.



Mahendran and Vedaldi, "Understanding Deep Image Representations by Inverting Them", CVPR 2015  
Figure from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016.

# Visualization distill.pub

- <https://distill.pub/2017/feature-visualization/>
- And a newer one:

<https://distill.pub/2018/building-blocks/>

# Fooling ConvNets

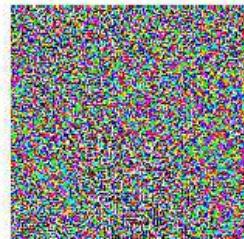
- Given an image  $I$  labeled as  $l_1$ , find minimum “ $r$ ” (noise) such that  $I + r$  is classified as a different label,  $l_2$ .
- I.e., minimize:

$$\text{loss}(I + r, l_2) + c|r|$$



$x$   
“panda”  
57.7% confidence

$+ .007 \times$



$\text{sign}(\nabla_x J(\theta, x, y))$   
“nematode”  
8.2% confidence

=



$\epsilon \text{sign}(\nabla_x J(\theta, x, y))$   
“gibbon”  
99.3 % confidence

## EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES

Ian J. Goodfellow, Jonathon Shlens & Christian Szegedy  
Google Inc., Mountain View, CA  
[{goodfellow,shlens,szegedy}@google.com](mailto:{goodfellow,shlens,szegedy}@google.com)

### Intriguing properties of neural networks

Christian Szegedy  
Google Inc.

Wojciech Zaremba  
New York University

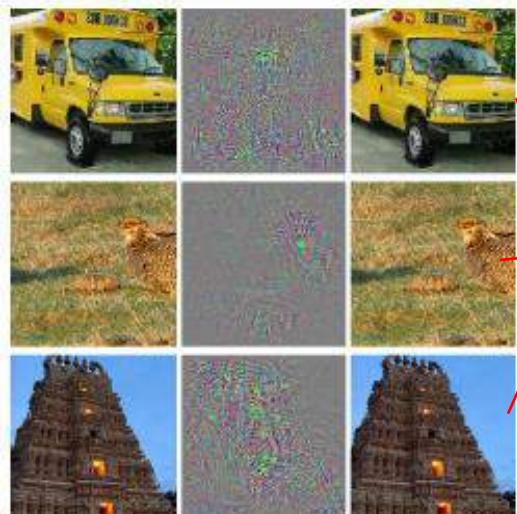
Ilya Sutskever  
Google Inc.

Joan Bruna  
New York University

Dumitru Erhan  
Google Inc.

Ian Goodfellow  
University of Montreal

Rob Fergus  
New York University  
Facebook Inc.



Ostrich

# More on adversarial examples

- How to classify adversarial examples?
  - You need to train your network against them!
  - That is very expensive and training against all kinds of adversarial examples is not possible
  - However, training against adversarial examples increases accuracy on non-adversarial examples as well.
- They are still an unsolved aspect/issue in neural networks
- Adversarial are problems of any learning method
- See I. Goodfellow for more on adversarial examples:
  - <http://www.kdnuggets.com/2015/07/deep-learning-adversarial-examples-misconceptions.html>

# Popular CNN models

# Gradient-Based Learning Applied to Document Recognition

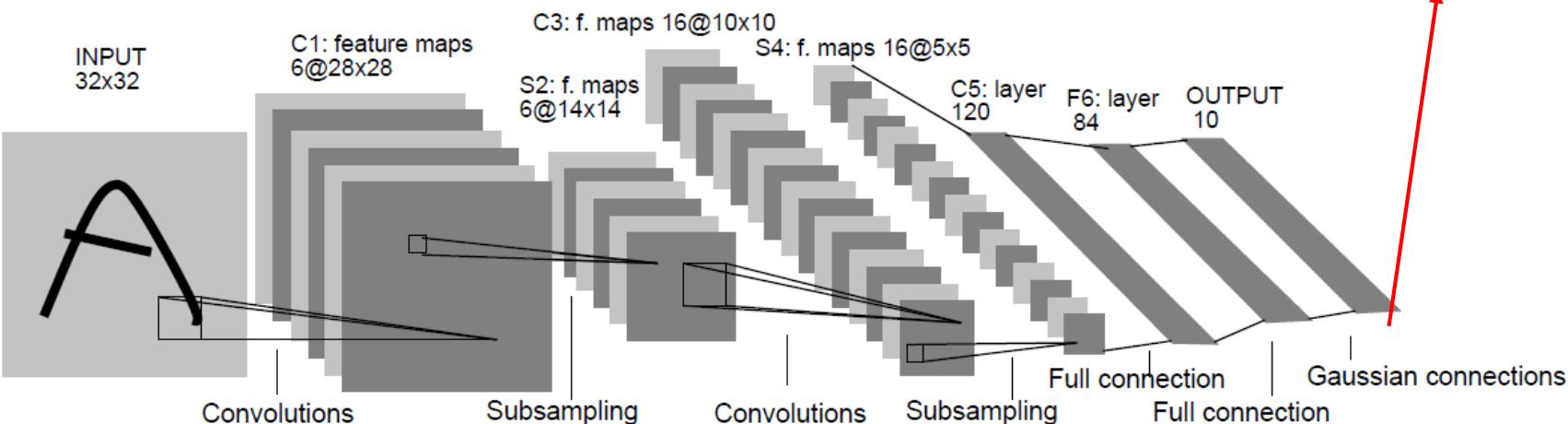
Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

## LeNet (1998)

- For reading zip codes and digits

Euclidean RBF:

$$y_i = \sum_j (x_j - w_{ij})^2.$$



$F=5 \times 5$

$S=1$

$P=0$

$F=2 \times 2$

$S=2$

$\text{sigm}(\alpha avg + b)$   
 $\alpha$  &  $b$ : trainable

$F=5 \times 5$

$S=1$

$P=0$

$F=2 \times 2$

$S=2$

$\text{sigm}(\alpha avg + b)$   
 $\alpha$  &  $b$ : trainable

$F=5 \times 5$

$S=1$

$P=0$

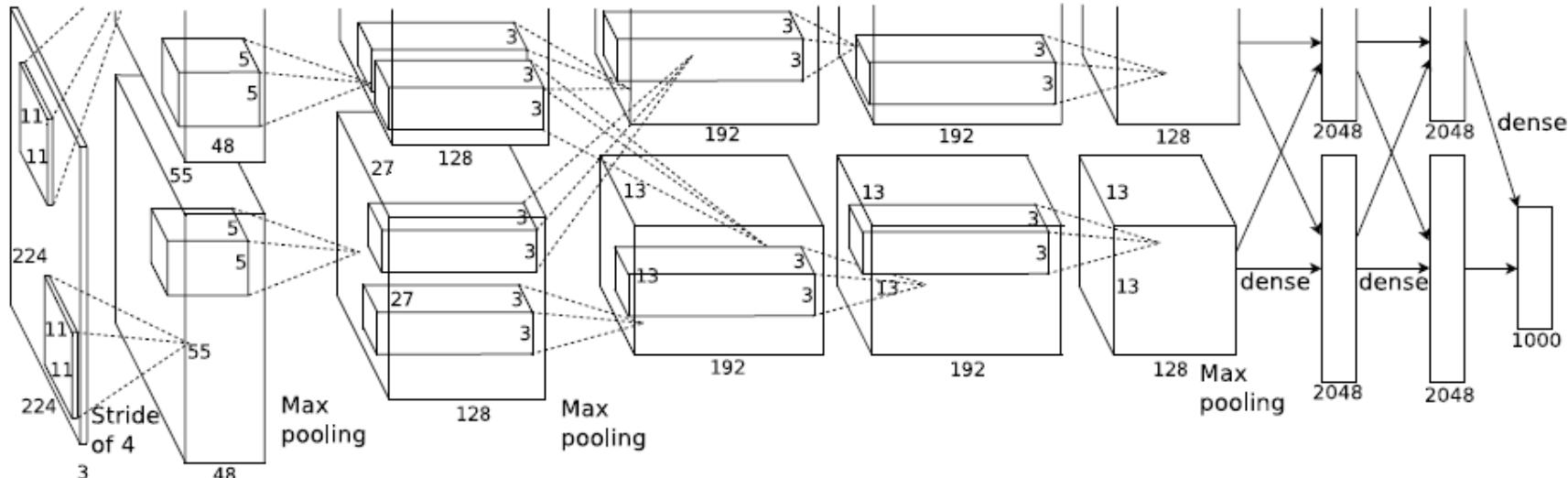
# AlexNet (2012)

Alex Krizhevsky  
University of Toronto  
kriz@cs.utoronto.ca

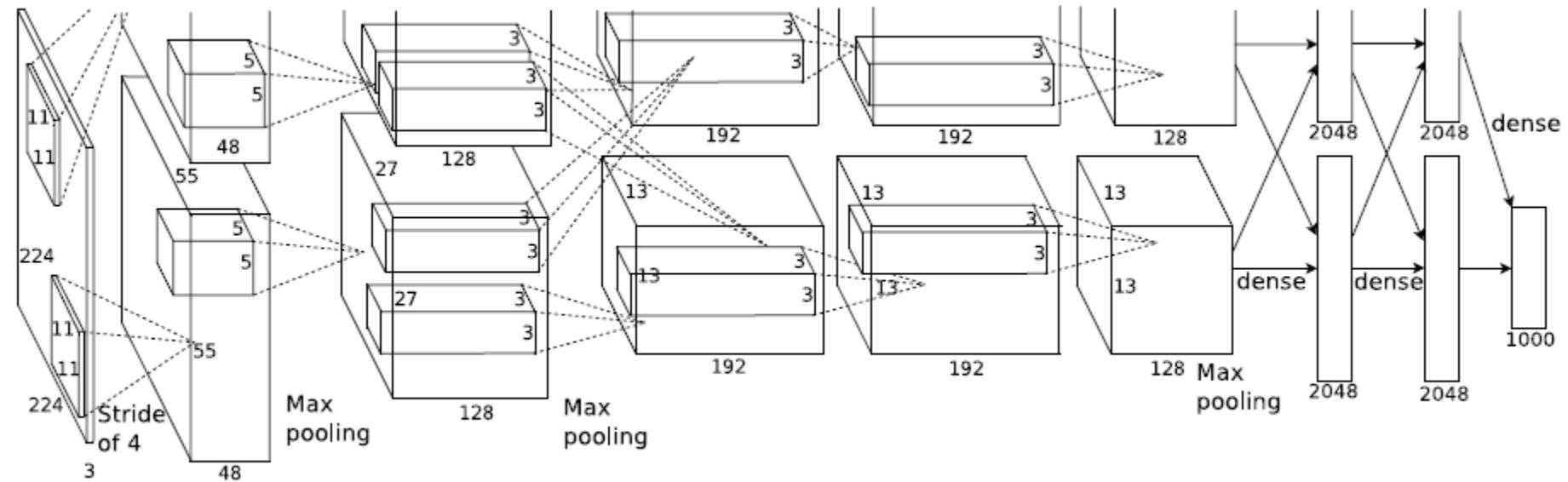
Ilya Sutskever  
University of Toronto  
ilya@cs.utoronto.ca

Geoffrey E. Hinton  
University of Toronto  
hinton@cs.utoronto.ca

- Popularized CNN in computer vision & pattern recognition
- ImageNet ILSVRC challenge 2012 winner
- Similar to LeNet
  - Deeper & bigger
  - Many CONV layers on top of each other (rather than adding immediately a pooling layer after a CONV layer)
  - Uses GPU
- 650K neurons. 60M parameters. Trained on 2 GPUs for a week.

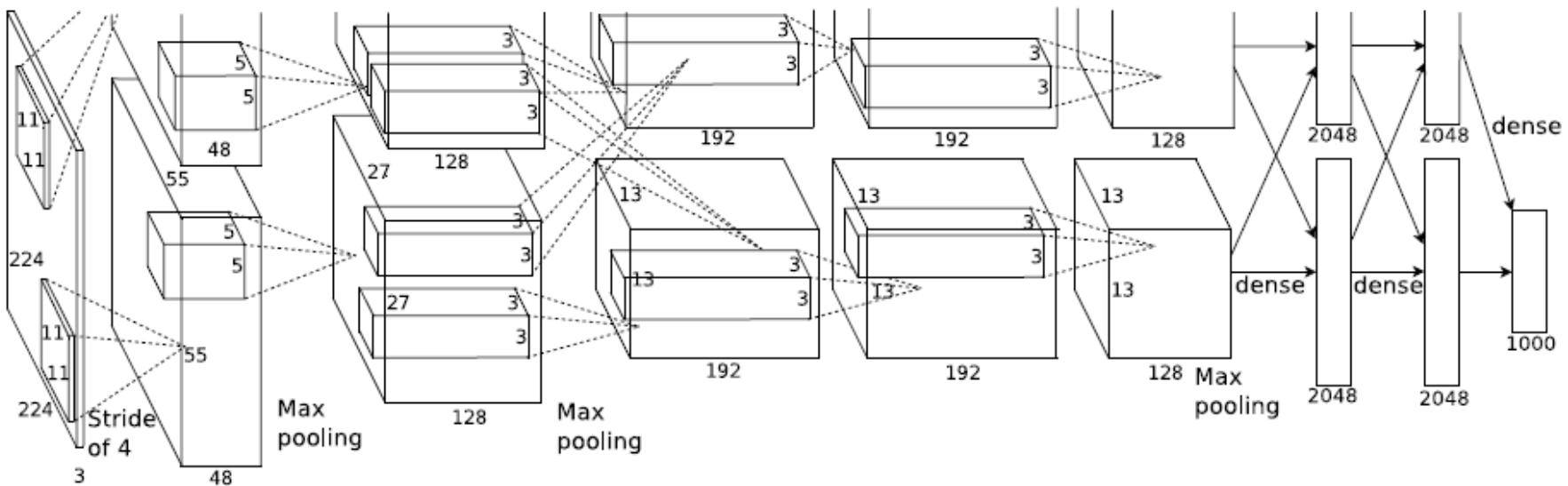


# AlexNet (2012) Details



- Since the network is too big to fit in on GPU, it is divided into two.
- Note the cross connections between the “pathways”.
- Uses ReLU as non-linearity after every convolutional and fully-connected layer.
- Normalization layer is placed after the first & the second convolutional layers.
- Max-pooling layer is placed only after the normalization layers & the fifth convolutional layer.
- Last layer is a soft-max.

# AlexNet (2012) Training



- Data augmentation & dropout are used during training to avoid overfitting.
- Stochastic Gradient Descent with a batchsize of 128 examples is used.
- Momentum with coefficient 0.9 is employed.
- Weight decay (L2 regularization cost) with factor 0.0005 is also used in the loss function.
- Weights are initialized from a zero-mean Gaussian distribution with 0.01 std.
- Learning rate started with 0.01 and **manually** divided by 10 when the validation error rate stopped improving.
- Trained on 1.2 million images, which took 5-6 days on two GPUs.

# AlexNet (2012): The learned filters

- Do you notice anything strange with the filters?

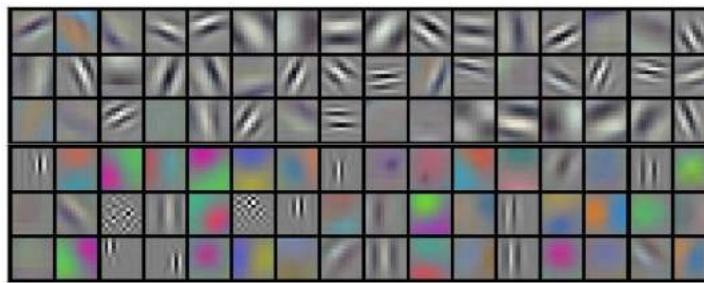
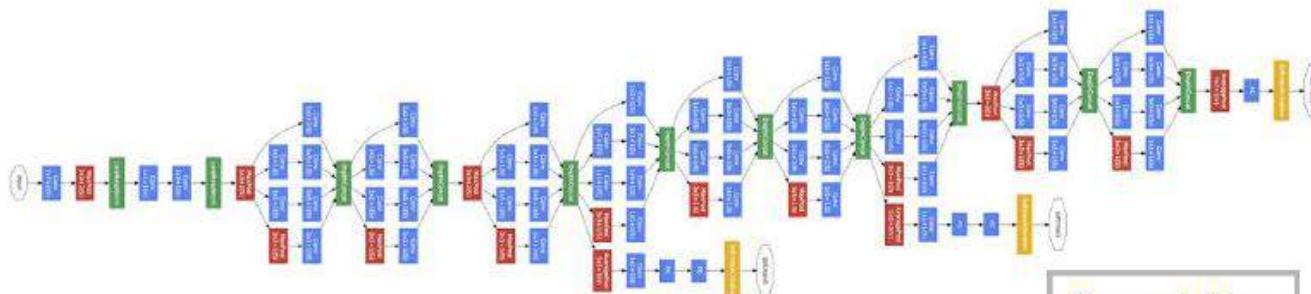


Figure 3: 96 convolutional kernels of size  $11 \times 11 \times 3$  learned by the first convolutional layer on the  $224 \times 224 \times 3$  input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

Figure 3 shows the convolutional kernels learned by the network's two data-connected layers. The network has learned a variety of frequency- and orientation-selective kernels, as well as various colored blobs. Notice the specialization exhibited by the two GPUs, a result of the restricted connectivity described in Section 3.5. The kernels on GPU 1 are largely color-agnostic, while the kernels on GPU 2 are largely color-specific. This kind of specialization occurs during every run and is independent of any particular random weight initialization (modulo a renumbering of the GPUs).

# GoogleNet (2014)

- ImageNet 2014 winner
- Contributions:
  - Inception module
    - Dramatically reduced parameters (from 60M in AlexNet to 4M)
    - Avg Pooling at the top, instead of fully-connected layer → Reduced number of parameters
- Motivation:
  - Going bigger (in depth or width) means too many parameters.
  - Go bigger by maintaining sparse connections.



Convolution  
Pooling  
Softmax  
Other

Christian Szegedy  
Google Inc.

Wei Liu  
University of North Carolina, Chapel Hill

Yangqing Jia  
Google Inc.

Pierre Sermanet  
Google Inc.

Scott Reed  
University of Michigan

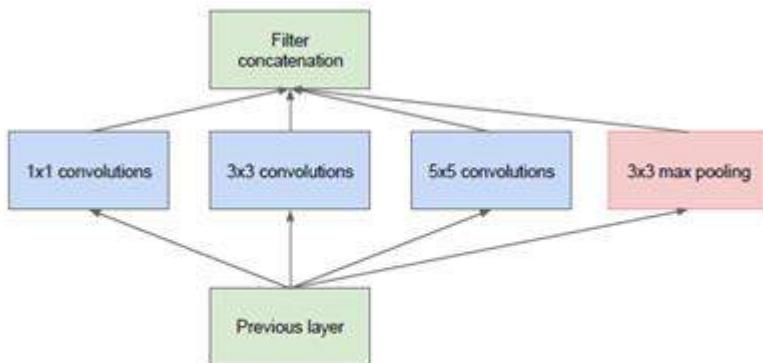
Dragomir Anguelov  
Google Inc.

Dumitru Erhan  
Google Inc.

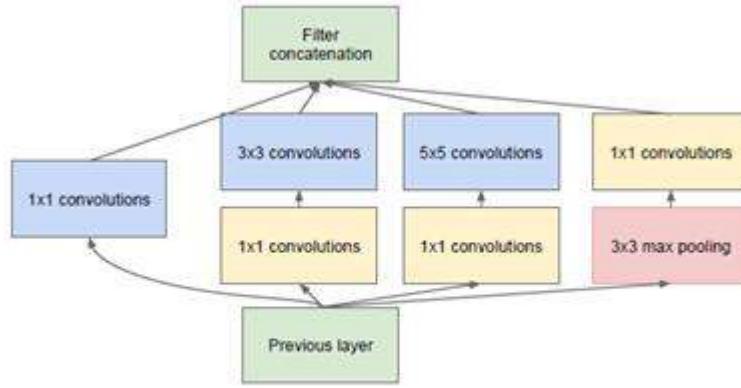
Vincent Vanhoucke  
Google Inc.

Andrew Rabinovich  
Google Inc.

# Inception module: “network in network” (inspired from Lin et al., 2013)

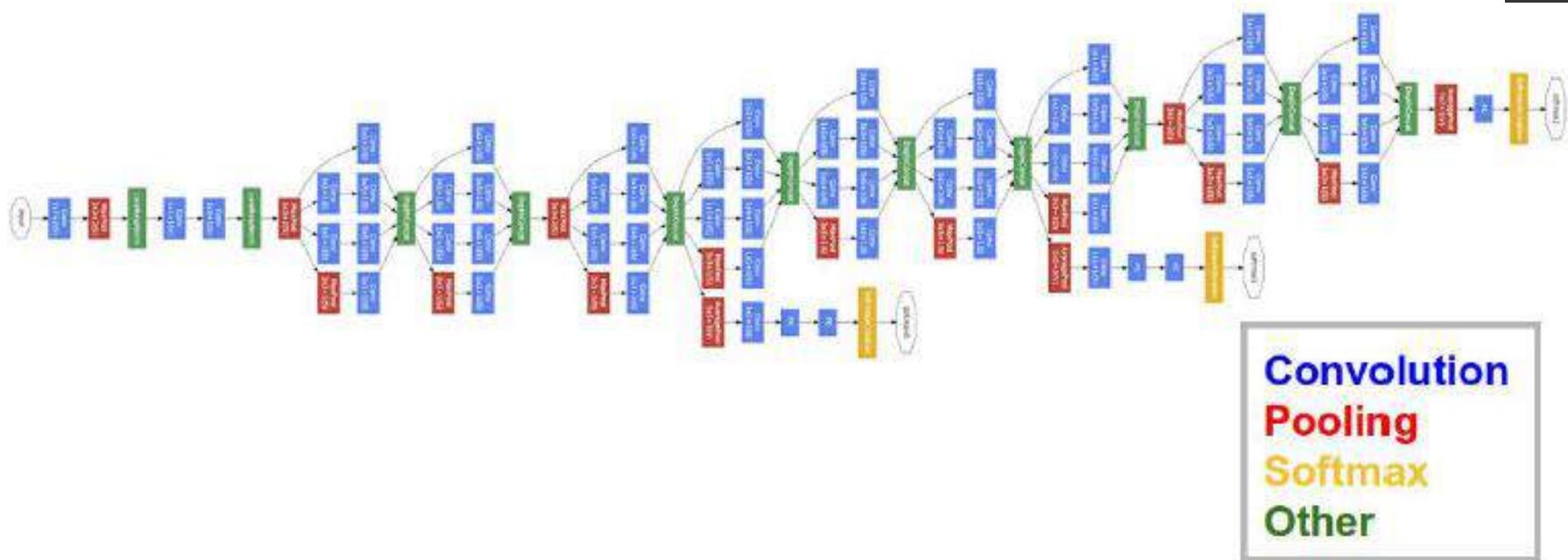


(a) Inception module, naïve version



(b) Inception module with dimension reductions

- Concatenation is performed along the “columns” (depth).
  - The output of inception layers must have the same size.
- The naïve version has a tendency to blow up in number of channels.
  - Why? Max-pooling does not change the number of channels. When concatenated with other filter responses, number of channels increase with every layer.
  - Solution: Do 1x1 convolution to decrease the number of channels.
    - Also called “bottleneck”.
- In order to decrease the computational complexity of 3x3 and 5x5 pooling, they are also preceded by 1x1 convolution (i.e., the number of channels are reduced).

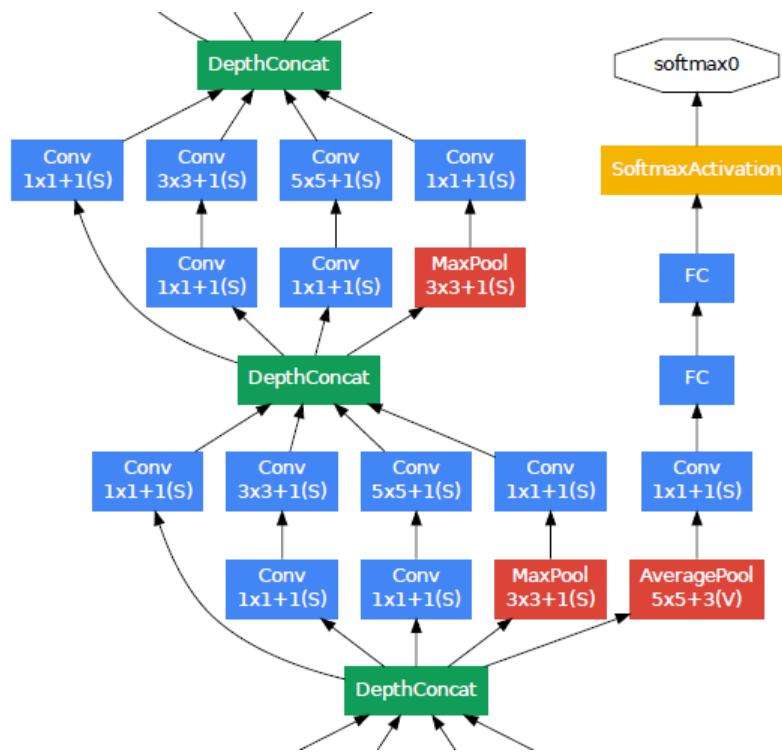


One of the main beneficial aspects of this architecture is that it allows for increasing the number of units at each stage significantly without an uncontrolled blow-up in computational complexity. The ubiquitous use of dimension reduction allows for shielding the large number of input filters of the last stage to the next layer, first reducing their dimension before convolving over them with a large patch size. Another practically useful aspect of this design is that it aligns with the intuition that visual information should be processed at various scales and then aggregated so that the next stage can abstract features from different scales simultaneously.

For the inception (4a) module, the input to

- 1x1 convolution is  $14 \times 14 \times 512$ .
  - 3x3 convolution is  $14 \times 14 \times 96$ .
  - 5x5 convolution is  $14 \times 14 \times 16$ .
  - Poolproj is  $14 \times 14 \times 512$ .

- Since the intermediate layers learn to discriminate features specific to a class, we can directly link them to the loss term.
  - Encourages these layers to become more discriminative
  - Increases propagation of gradient signal to earlier stages



# GoogleNet: Training

- ReLU after all layers
- Max pooling in inception modules as well as a whole layer occasionally
- Avg pooling instead of fully-connected layers
  - Only a minor change in the accuracy (0.6%)
  - However, less number of parameters
- Other usual tricks (e.g., dropout, augmentation etc.) are used.
- Trained on CPUs using a distributed machine learning system.
- SGD with momentum (0.9).
- Fixed learning rate scheme with 4% decrease every 8 epochs
- They trained many different models with different initializations and parameters. They combined these models using different methods and tricks. There is no single training method that yields the results they achieved.

# VGGNet (2014)

- ImageNet runner up in 2014
- Contribution:
  - Use small RFs & increase depth as much as possible
  - 16 CONV/FC layers.
  - 3x3 CONVs and 2x2 pooling from beginning to the end
- Although performs slightly worse than GoogleNet in image classification, VGGNet may perform better at other tasks (such as transfer learning problems).
- Downside: Needs a lot of memory & parameters (140M)

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as “conv⟨receptive field size⟩-⟨number of channels⟩”. The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input ( $224 \times 224$ RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

# ResNet (2015)

Kaiming He    Xiangyu Zhang    Shaoqing Ren    Jian Sun  
Microsoft Research  
[{kahe,v-xiangz,v-shren,jiansun}@microsoft.com](mailto:{kahe,v-xiangz,v-shren,jiansun}@microsoft.com)

- Increasing the depth naively may not give you better performance after a number of depths
  - Why?
    - This is shown to be not due to overfitting (since training error also gets worse) or vanishing gradients (suitable non-linearities used)
    - Accuracy is somehow saturated. Not clear why. Though reported in several studies.
- Solution: Make shortcut connections

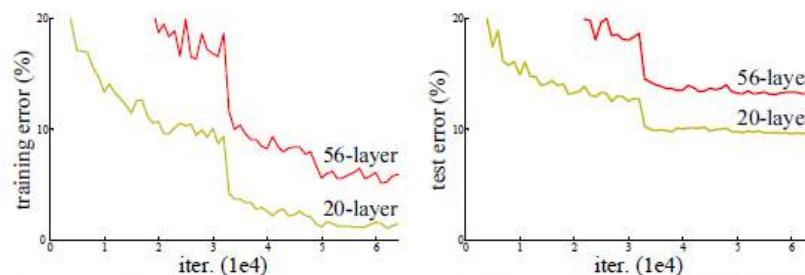


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

# ResNet (2015)

- Residual (shortcut) connections

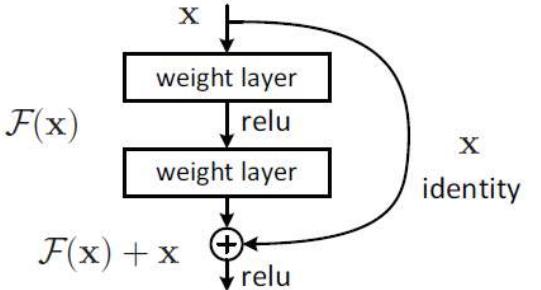


Figure 2. Residual learning: a building block.

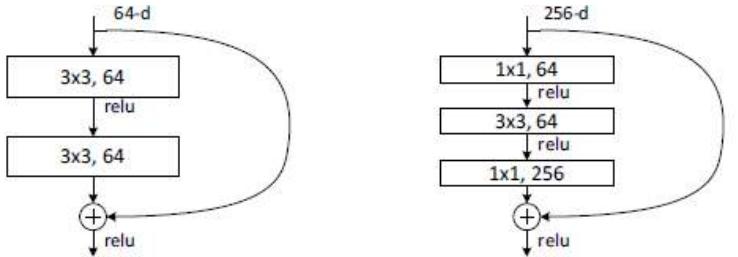
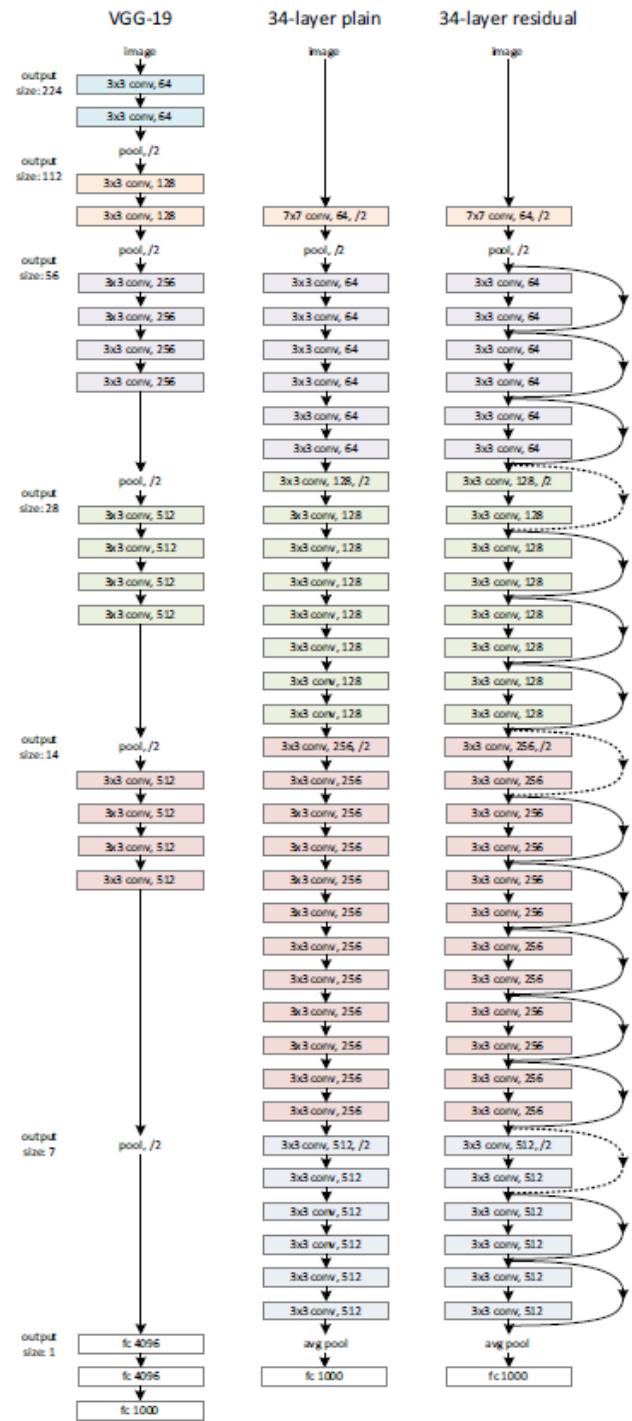


Figure 5. A deeper residual function  $\mathcal{F}$  for ImageNet. Left: a building block (on  $56 \times 56$  feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.



# ResNet (2015)

- Residual (shortcut) connections

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43 <sup>†</sup>
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	<b>19.38</b>	<b>4.49</b>

Table 4. Error rates (%) of single-model results on the ImageNet validation set (except <sup>†</sup> reported on the test set).

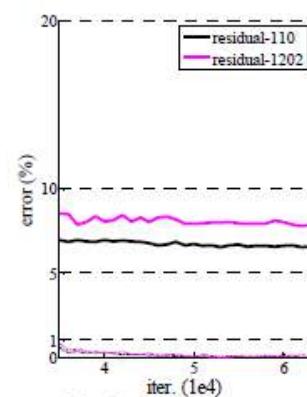
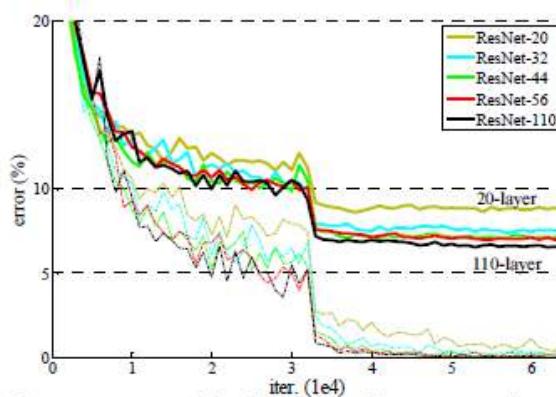
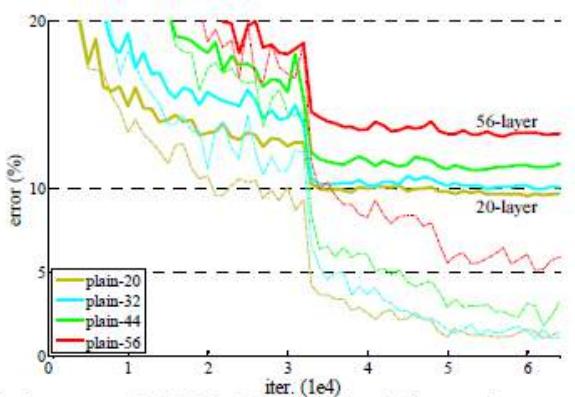
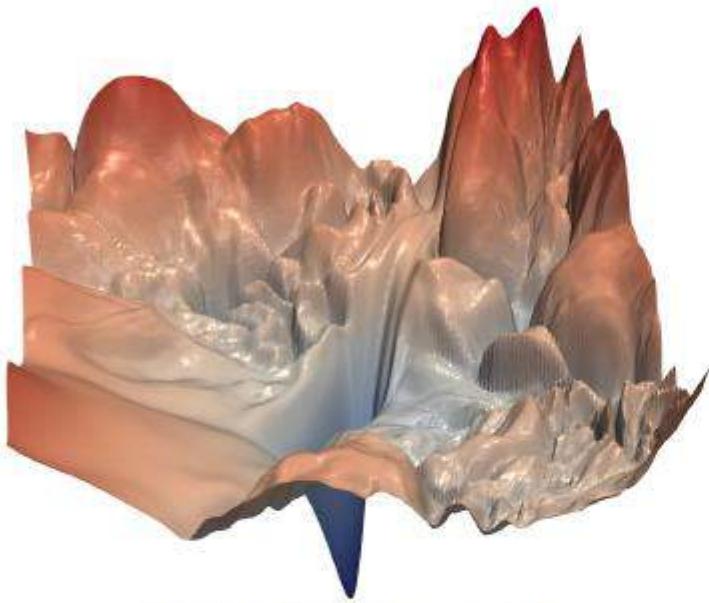
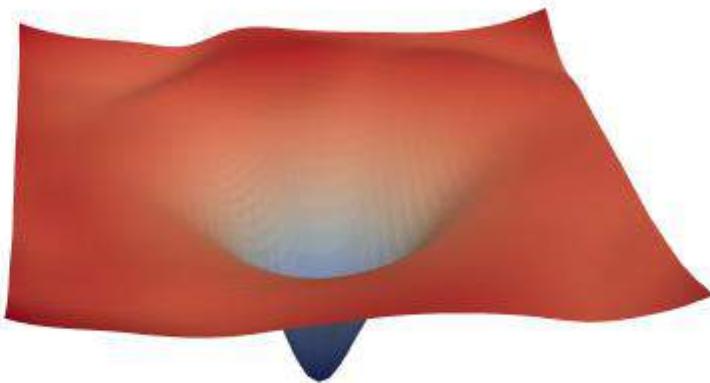


Figure 6. Training on CIFAR-10. Dashed lines denote training error, and bold lines denote testing error. **Left:** plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle:** ResNets. **Right:** ResNets with 110 and 1202 layers.

# Effect of residual connections



(a) without skip connections



(b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The vertical axis is logarithmic to show dynamic range. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

VISUALIZING THE LOSS LANDSCAPE OF NEURAL NETS

2018

Hao Li<sup>1</sup>, Zheng Xu<sup>1</sup>, Gavin Taylor<sup>2</sup>, Christoph Studer<sup>3</sup>, Tom Goldstein<sup>1</sup>

<sup>1</sup>University of Maryland, College Park, <sup>2</sup>United States Naval Academy, <sup>3</sup>Cornell University  
 [{haoli,xuzh,tomg}@cs.umd.edu](mailto:{haoli,xuzh,tomg}@cs.umd.edu), [taylor@usna.edu](mailto:taylor@usna.edu), [studer@cornell.edu](mailto:studer@cornell.edu)

# Recent work

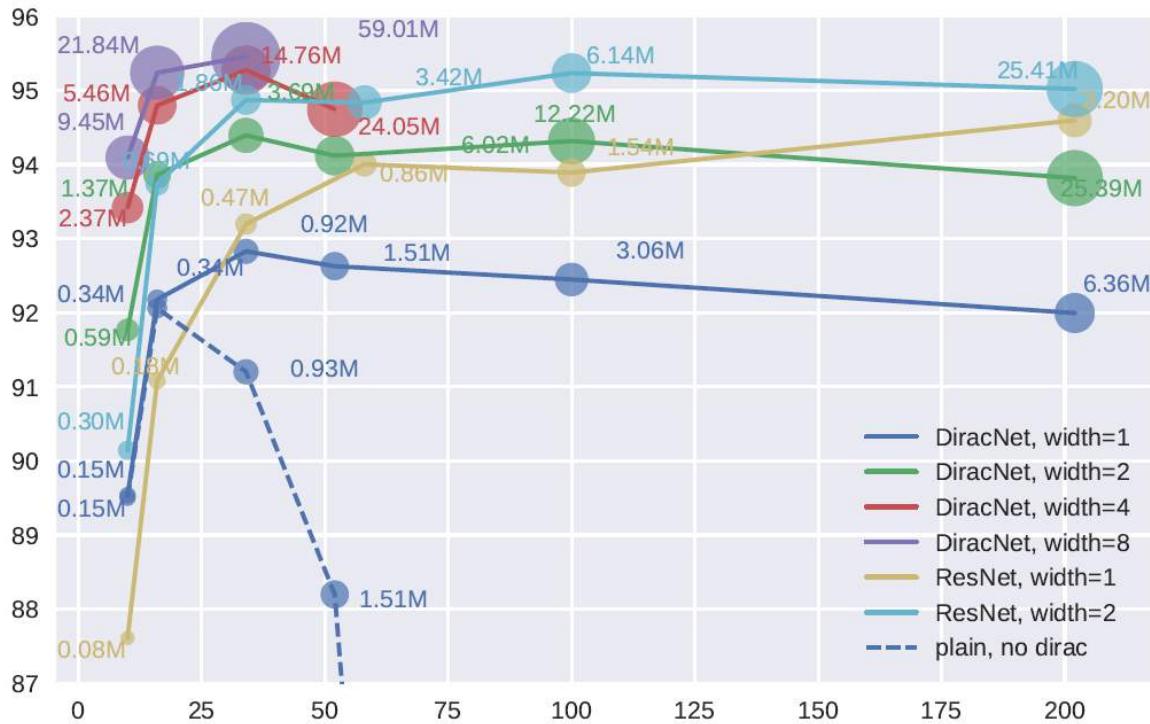
## DiracNets: Training Very Deep Neural Networks Without Skip-Connections

Sergey Zagoruyko  
 sergey.zagoruyko@enpc.fr  
 Nikos Komodakis  
 nikos.komodakis@enpc.fr

Université Paris-Est, École des Ponts  
 ParisTech  
 Paris, France

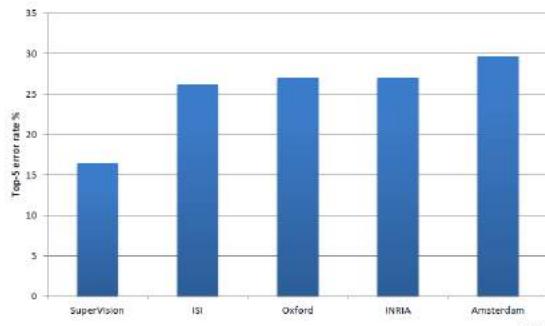
### Abstract

Deep neural networks with skip-connections, such as ResNet, show excellent performance in various image classification benchmarks. It is though observed that the initial motivation behind them - training deeper networks - does not actually hold true, and the benefits come from increased capacity, rather than from depth. Motivated by this, and inspired from ResNet, we propose a simple Dirac weight parameterization, which allows us to train very deep plain networks without skip-connections, and achieve nearly the same performance. This parameterization has a minor computational cost at training time and no cost at all at inference. We're able to achieve 95.5% accuracy on CIFAR-10 with 34-layer deep plain network, surpassing 1001-layer deep ResNet, and approaching Wide ResNet. Our parameterization also mostly eliminates the need of careful initialization in residual and non-residual networks. The code and models for our experiments are available at <https://github.com/szagoruyko/diracnets>



## ImageNet Classification 2012

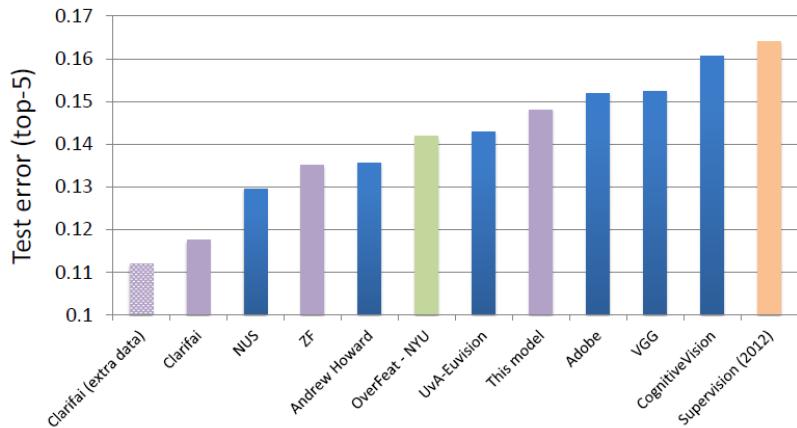
- Krizhevsky et al. -- 16.4% error (top-5)
- Next best (non-convnet) – 26.2% error



Slide: R. Fergus 31

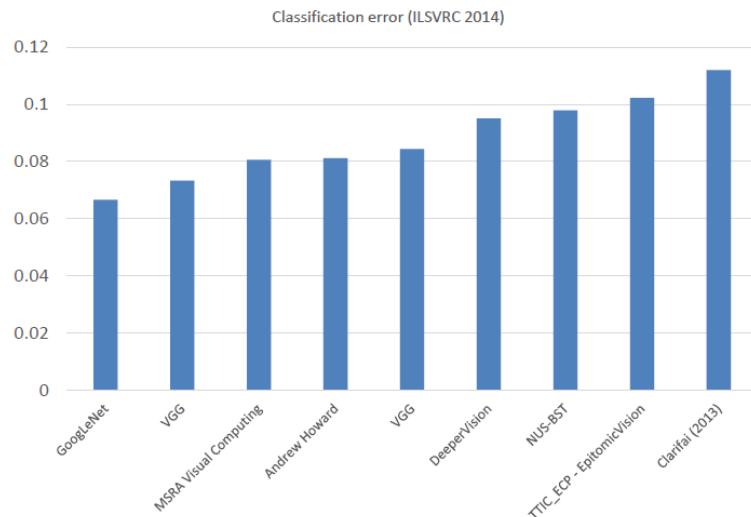
## ImageNet Classification 2013 Results

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



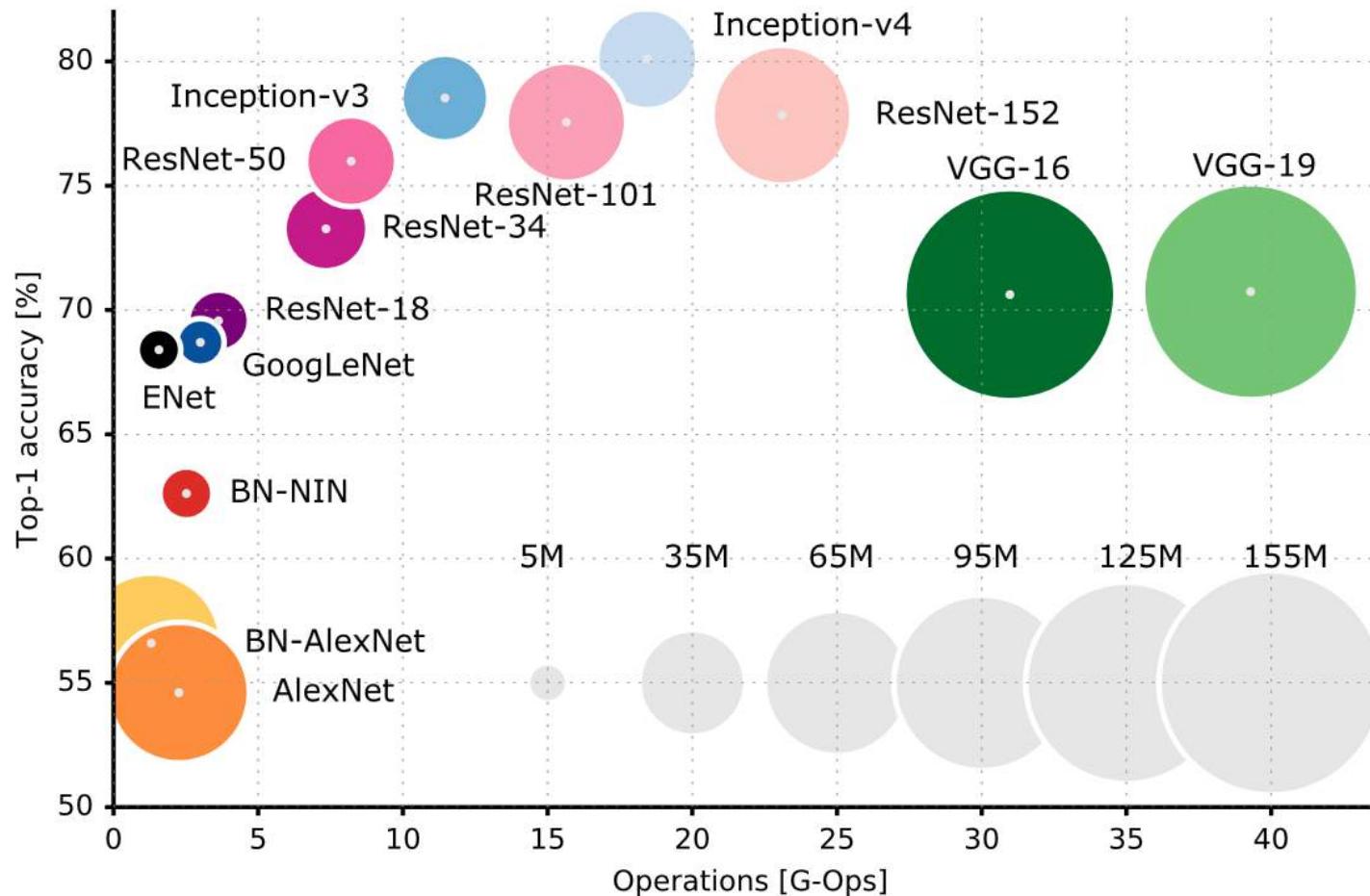
Slide: R. Fergus 32

## ImageNet Classification 2014 Results



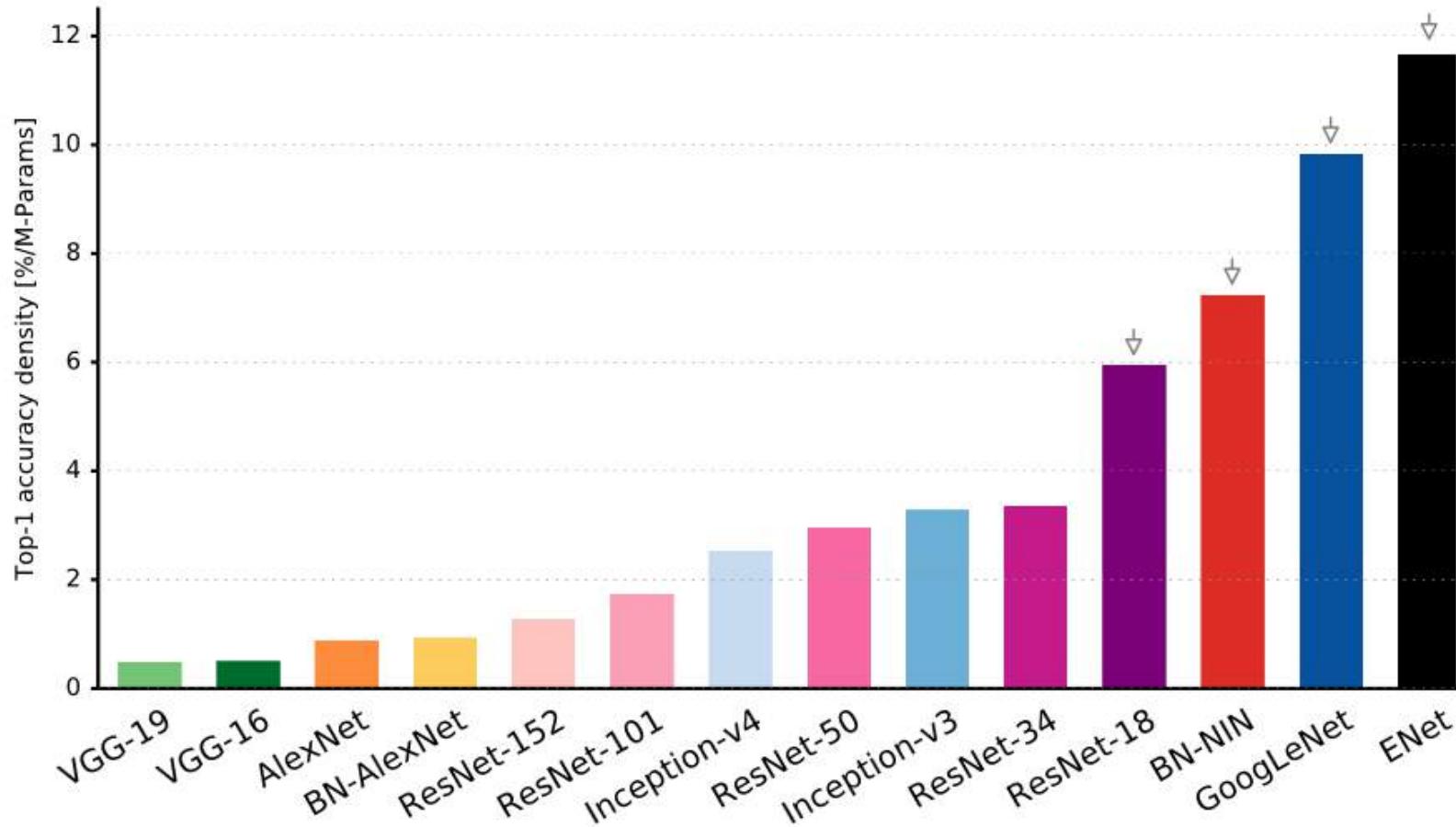
# Comparison:

<http://culurciello.github.io/tech/2016/06/04/nets.html>



# Comparison:

<http://culurciello.github.io/tech/2016/06/04/nets.html>



# Going deep may not be the only answer

## Shallow Networks for High-Accuracy Road Object-Detection

Khalid Ashraf, Bichen Wu, Forrest N. Iandola, , Matthew W. Moskewicz, Kurt Keutzer  
Electrical Engineering and Computer Sciences Department, UC Berkeley

{ashrafkhalid, bichen}@berkeley.edu, {forresti, moskewcz, keutzer}@eecs.berkeley.edu

### Abstract

*The ability to automatically detect other vehicles on the road is vital to the safety of partially-autonomous and fully-autonomous vehicles. Most of the high-accuracy techniques for this task are based on R-CNN or one of its faster variants. In the research community, much emphasis has been applied to using 3D vision or complex R-CNN variants to achieve higher accuracy. However, are there more straightforward modifications that could deliver higher accuracy? Yes. We show that increasing input image resolution (i.e. upsampling) offers up to 12 percentage-points higher accuracy compared to an off-the-shelf baseline. We also find situations where earlier/shallower layers of CNN provide higher accuracy than later/deeper layers. We further show that shallow models and upsampled images yield competitive accuracy. Our findings contrast with the current trend towards deeper and larger models to achieve high accuracy in domain specific detection tasks.*

# Recent work

## **“Towards Principled Design of Deep Convolutional Networks: Introducing SimpNet”**

- Major winning Convolutional Neural Networks (CNNs), such as VGGNet, ResNet, DenseNet, \etc, include tens to hundreds of millions of parameters, which impose considerable computation and memory overheads. This limits their practical usage in training and optimizing for real-world applications. On the contrary, light-weight architectures, such as SqueezeNet, are being proposed to address this issue. However, they mainly suffer from low accuracy, as they have compromised between the processing power and efficiency. These inefficiencies mostly stem from following an ad-hoc designing procedure. In this work, we discuss and propose several crucial design principles for an efficient architecture design and elaborate intuitions concerning different aspects of the design procedure. Furthermore, we introduce a new layer called {\it SAF-pooling} to improve the generalization power of the network while keeping it simple by choosing best features. Based on such principles, we propose a simple architecture called {\it SimpNet}. We empirically show that SimpNet provides a good trade-off between the computation/memory efficiency and the accuracy solely based on these primitive but crucial principles. SimpNet outperforms the deeper and more complex architectures such as VGGNet, ResNet, WideResidualNet \etc, on several well-known benchmarks, while having 2 to 25 times fewer number of parameters and operations. We obtain state-of-the-art results (in terms of a balance between the accuracy and the number of involved parameters) on standard datasets, such as CIFAR10, CIFAR100, MNIST and SVHN. The implementations are available at \url{[this https URL](https://arxiv.org/abs/1802.06205)}.

# CNN applications

# Object Detection: Regions with CNN (R-CNN)

Region-based Convolutional Networks for  
Accurate Object Detection and Segmentation

Ross Girshick, Jeff Donahue, *Student Member, IEEE*, Trevor Darrell, *Member, IEEE*, and  
Jitendra Malik, *Fellow, IEEE*

2015

- A very straightforward application
- Start from a pre-trained model (trained on imagenet)
- Finetune using the new data available
- There are faster versions (called Fast-RCNN) by sharing computations performed in convolutions on different regions

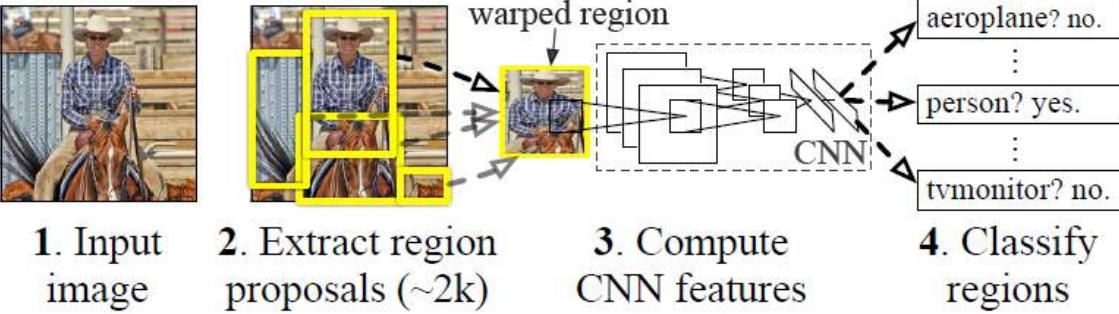


Fig. 1. Object detection system overview. Our system (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a large convolutional network (CNN), and then (4) classifies each region using class-specific linear SVMs. We trained an R-CNN that achieves a mean average precision (mAP) of 62.9% on PASCAL VOC 2010. For comparison, [21] reports 35.1% mAP using the same region proposals, but with a spatial pyramid and bag-of-visual-words approach. The popular deformable part models perform at 33.4%. On the 200-class ILSVRC2013 detection dataset, we trained an R-CNN with a mAP of 31.4%, a large improvement over OverFeat [19], which had the previous best result at 24.3% mAP.

# Faster R-CNN

## Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun

(2015)

- Replaced region-proposal method with a network
- Followed by “Fast R-CNN” classifier
- The two networks share some convolutional layers

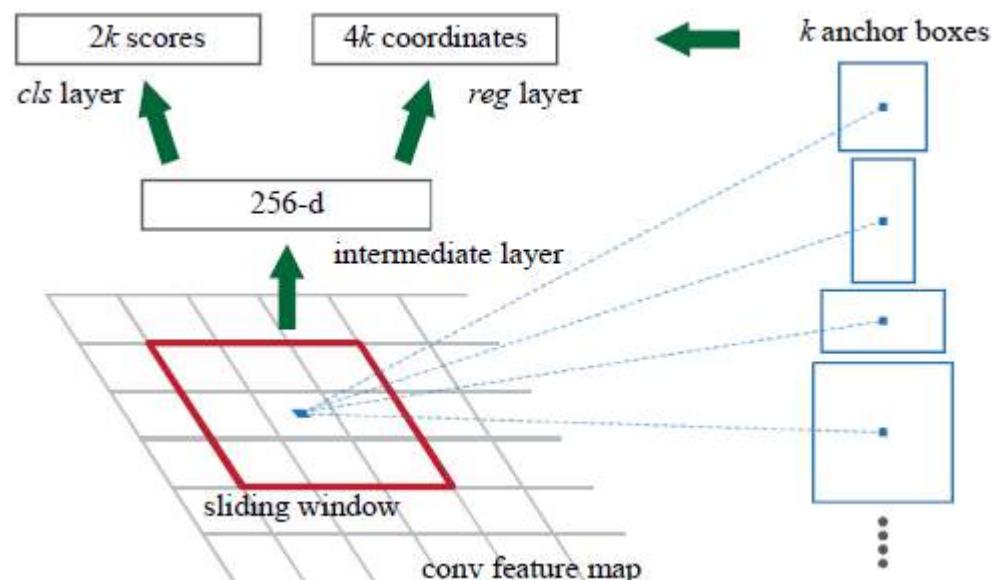
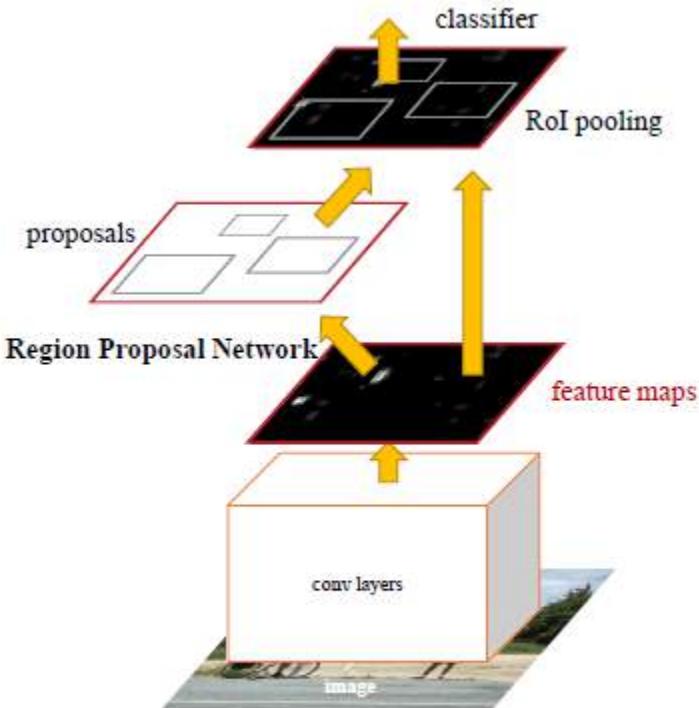
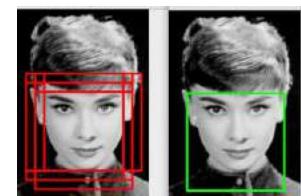


Figure 2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the ‘attention’ of this unified network.

Non-  
maximum  
suppression



# Object Detection: Mask R-CNN

## Mask R-CNN

Kaiming He   Georgia Gkioxari   Piotr Dollár   Ross Girshick  
Facebook AI Research (FAIR)  
2017

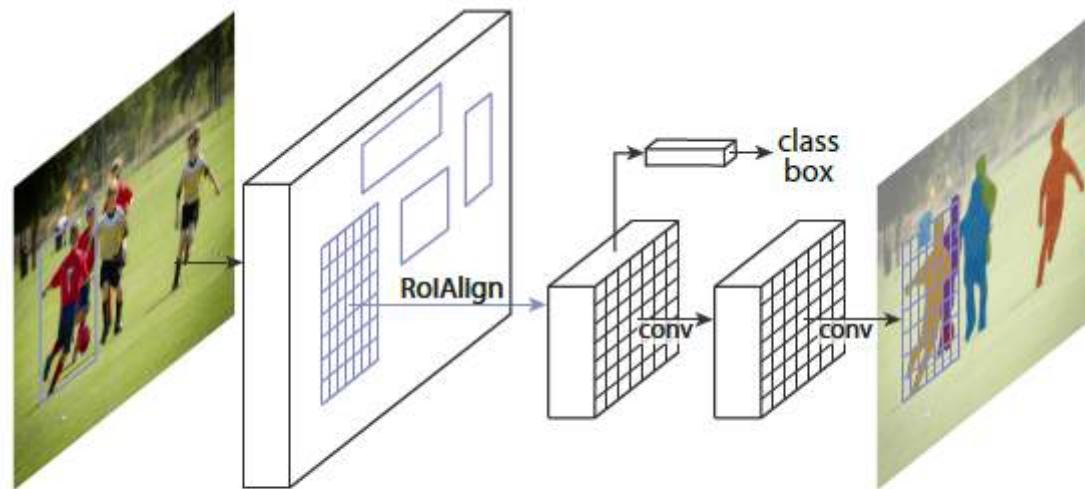


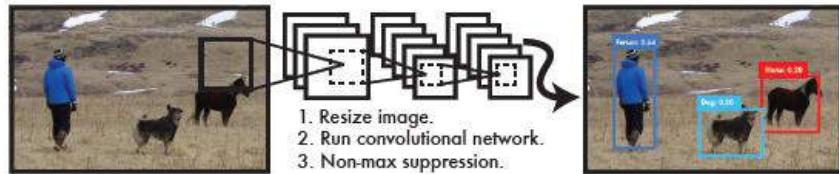
Figure 1. The Mask R-CNN framework for instance segmentation



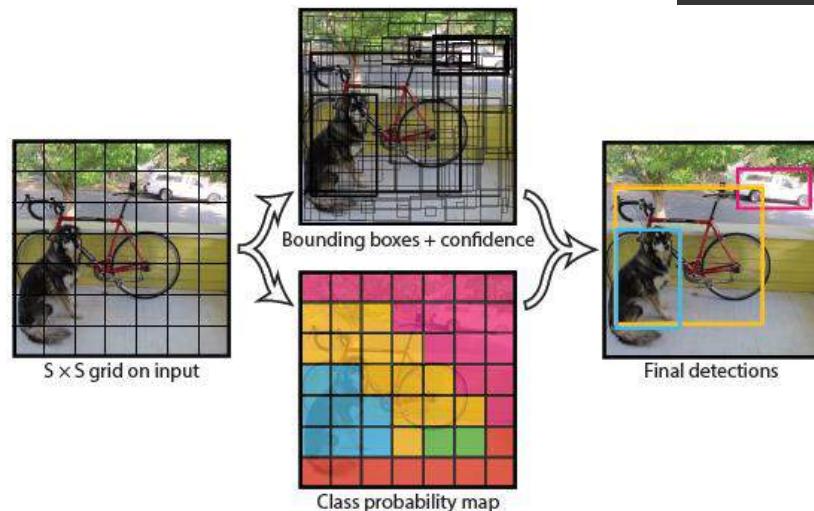
Figure 2. Mask R-CNN results on the COCO test set. These results are based on ResNet-101 [19], achieving a *mask AP* of 35.7 and running at 5 fps. Masks are shown in color, and bounding box, category, and confidences are also shown.

Single-stage method:

- Region-proposal & classification steps are combined.



**Figure 1:** The YOLO Detection System. Processing images with YOLO is simple and straightforward. Our system (1) resizes the input image to  $448 \times 448$ , (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.



**Figure 2:** The Model. Our system models detection as a regression problem. It divides the image into an  $S \times S$  grid and for each grid cell predicts  $B$  bounding boxes, confidence for those boxes, and  $C$  class probabilities. These predictions are encoded as an  $S \times S \times (B * 5 + C)$  tensor.

# Single-Shot Detector (SSD)

SSD: Single Shot MultiBox Detector

Wei Liu<sup>1</sup>, Dragomir Anguelov<sup>2</sup>, Dumitru Erhan<sup>3</sup>, Christian Szegedy<sup>3</sup>,  
Scott Reed<sup>4</sup>, Cheng-Yang Fu<sup>1</sup>, Alexander C. Berg<sup>1</sup>

(2016)

- Similar to YOLO, SSD is a single-stage detector
- Allows more bounding boxes per location
- 8372 BBs per class per image

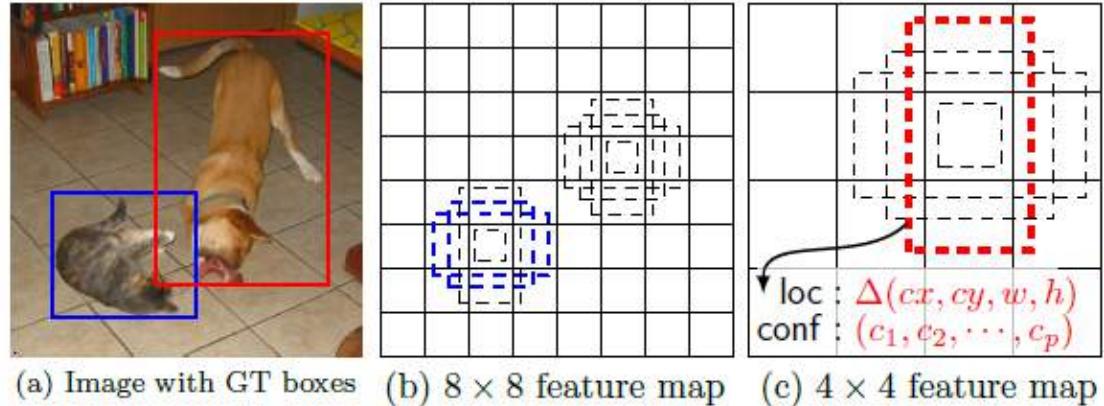


Fig. 1: **SSD framework.** (a) SSD only needs an input image and ground truth boxes for each object during training. In a convolutional fashion, we evaluate a small set (e.g. 4) of default boxes of different aspect ratios at each location in several feature maps with different scales (e.g.  $8 \times 8$  and  $4 \times 4$  in (b) and (c)). For each default box, we predict both the shape offsets and the confidences for all object categories ( $(c_1, c_2, \dots, c_p)$ ). At training time, we first match these default boxes to the ground truth boxes. For example, we have matched two default boxes with the cat and one with the dog, which are treated as positives and the rest as negatives. The model loss is a weighted sum between localization loss (e.g. Smooth L1 [6]) and confidence loss (e.g. Softmax).

# Feature Pyramid Networks for Object Detection

Tsung-Yi Lin<sup>1,2</sup>, Piotr Dollár<sup>1</sup>, Ross Girshick<sup>1</sup>,  
Kaiming He<sup>1</sup>, Bharath Hariharan<sup>1</sup>, and Serge Belongie<sup>2</sup>

<sup>1</sup>Facebook AI Research (FAIR)

<sup>2</sup>Cornell University and Cornell Tech

(2017)

- Used for feature extraction
- Can be used for region proposal, object detection or segmentation.

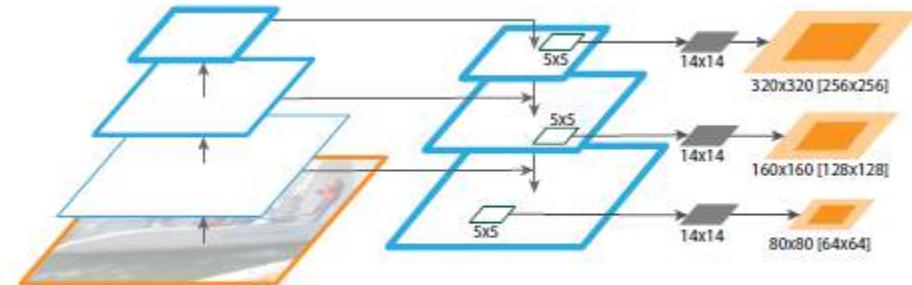
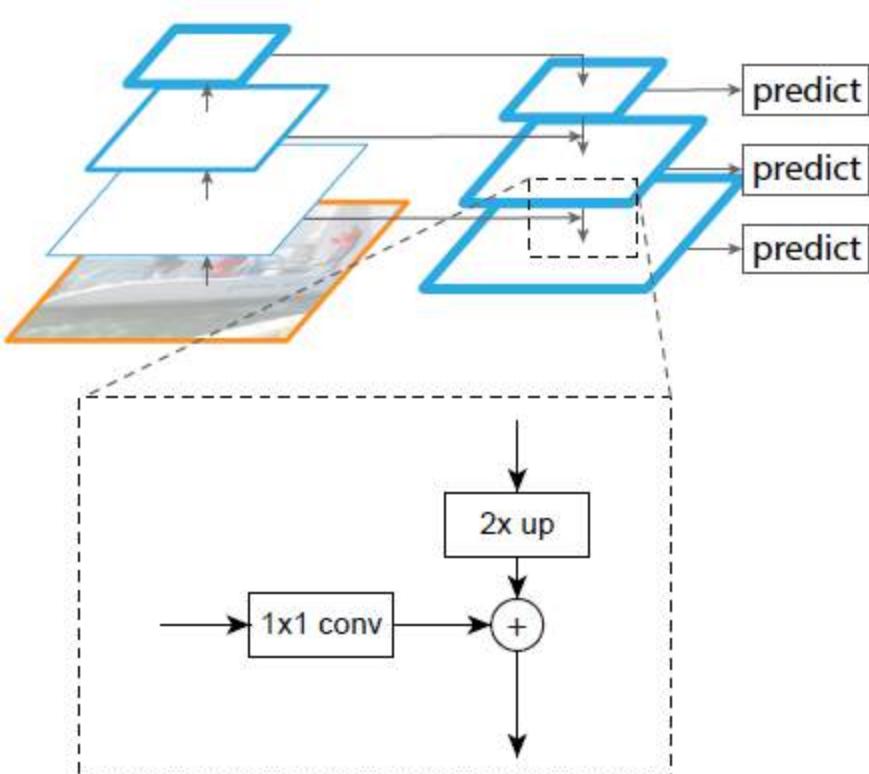


Figure 4. FPN for object segment proposals. The feature pyramid is constructed with identical structure as for object detection. We

# Focal Loss for Dense Object Detection (RetinaNet)

## Focal Loss for Dense Object Detection

Tsung-Yi Lin Priya Goyal Ross Girshick Kaiming He Piotr Dollár  
Facebook AI Research (FAIR)

2017

The highest accuracy object detectors to date are based on a two-stage approach popularized by R-CNN, where a classifier is applied to a sparse set of candidate object locations. In contrast, one-stage detectors that are applied over a regular, dense sampling of possible object locations have the potential to be faster and simpler, but have trailed the accuracy of two-stage detectors thus far. In this paper, we investigate why this is the case. We discover that the extreme foreground-background class imbalance encountered during training of dense detectors is the central cause. We propose to address this class imbalance by reshaping the standard cross entropy loss such that it down-weights the loss assigned to well-classified examples. Our novel Focal Loss focuses training on a sparse set of hard examples and prevents the vast number of easy negatives from overwhelming the detector during training. To evaluate the effectiveness of our loss, we design and train a simple dense detector we call RetinaNet. Our results show that when trained with the focal loss, RetinaNet is able to match the speed of previous one-stage detectors while surpassing the accuracy of all existing state-of-the-art two-stage detectors. Code is at:

<https://github.com/facebookresearch/Detectron>.

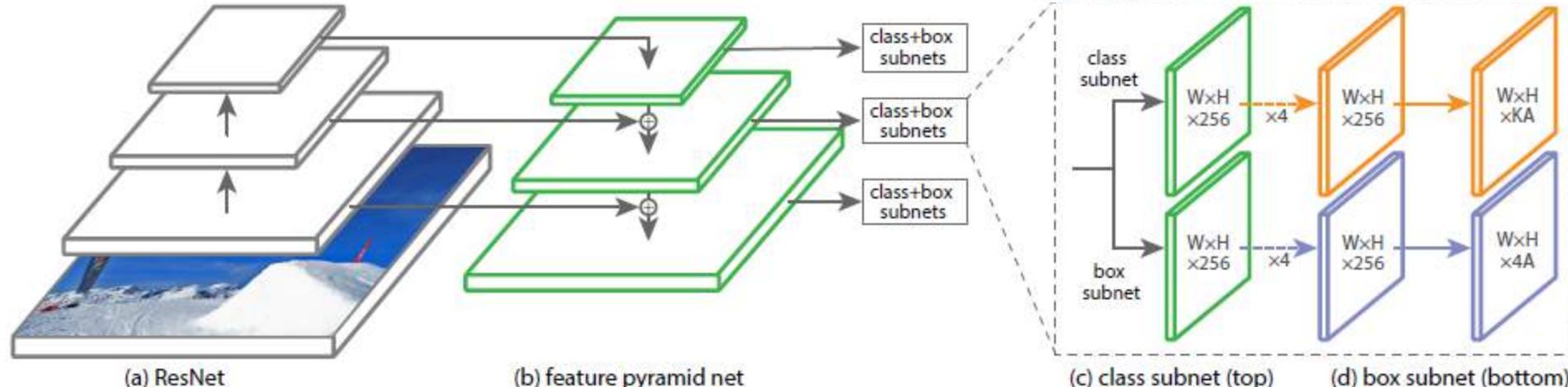


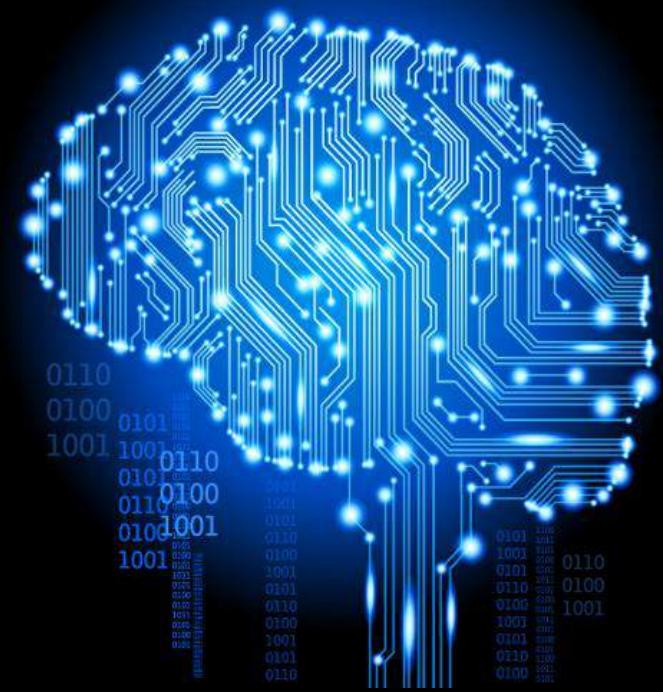
Figure 3. The one-stage **RetinaNet** network architecture uses a Feature Pyramid Network (FPN) [20] backbone on top of a feedforward ResNet architecture [16] (a) to generate a rich, multi-scale convolutional feature pyramid (b). To this backbone RetinaNet attaches two subnetworks, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d). The network design is intentionally simple, which enables this work to focus on a novel focal loss function that eliminates the accuracy gap between our one-stage detector and state-of-the-art two-stage detectors like Faster R-CNN with FPN [20] while running at faster speeds.

# CENG 783

# Deep Learning

*Week – 10  
Convolutional Neural Networks (cont.)  
Recurrent Neural Networks*

Sinan Kalkan



© AlchemyAPI



# Today

- Finalize CNN Applications & CNN
  - Style transfer
- Recurrent Neural Networks

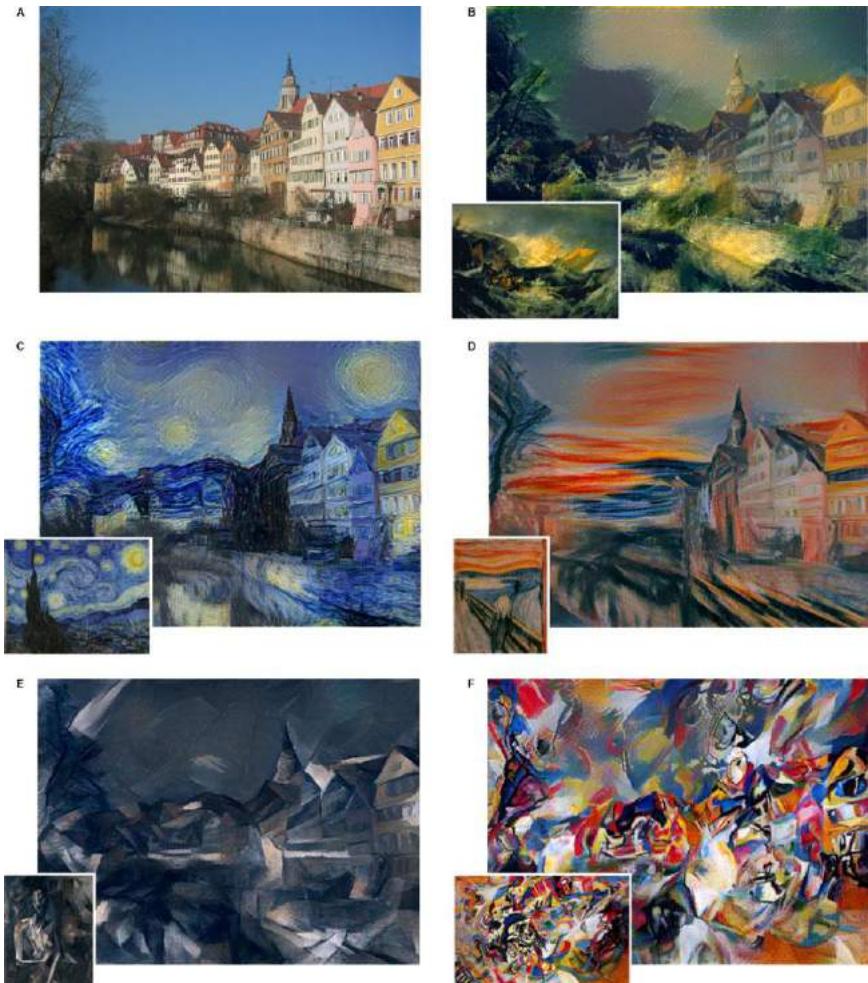
# A Neural Algorithm of Artistic Style

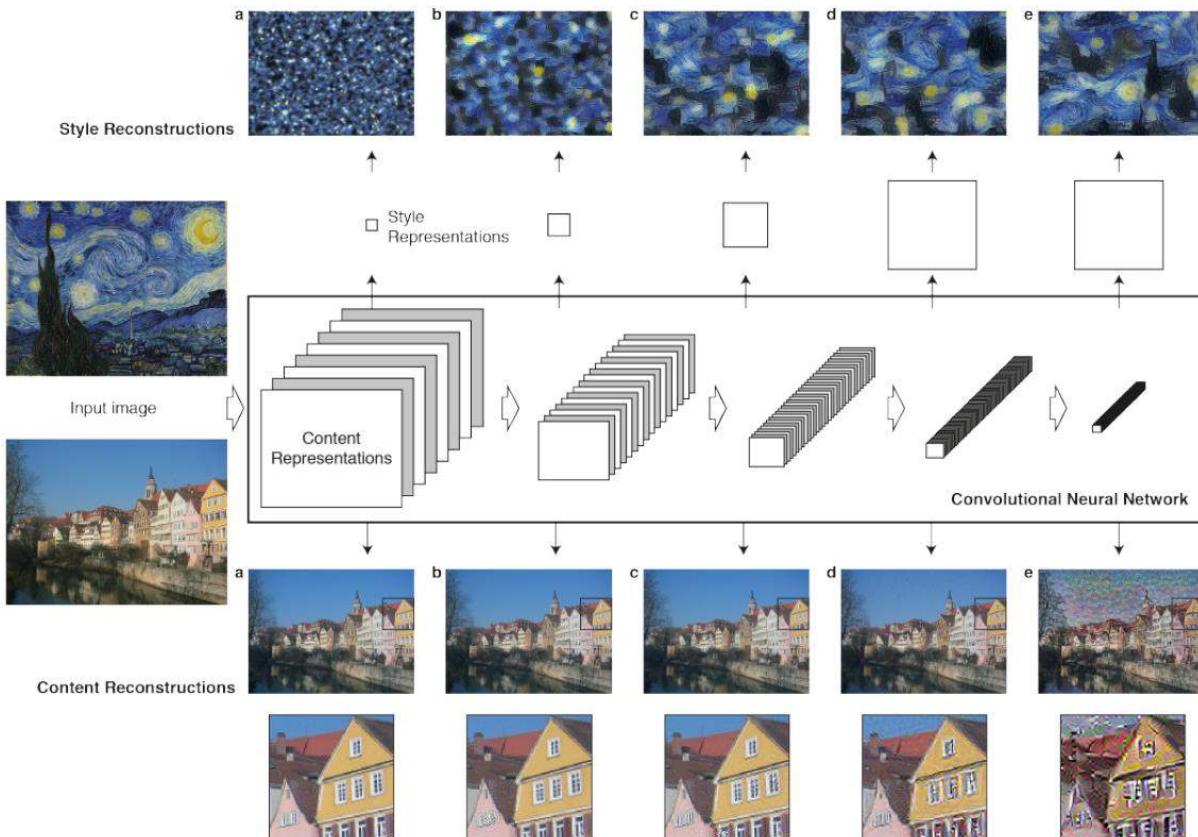
Leon A. Gatys,<sup>1,2,3\*</sup> Alexander S. Ecker,<sup>1,2,4,5</sup> Matthias Bethge<sup>1,2,4</sup>

2015

Takes 19 layer VGG as the base (no FC layers)

Max pooling is replaced by avg pooling since it produced more appealing results





**Figure 1: Convolutional Neural Network (CNN).** A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions.** We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the network's responses in a particular layer. We reconstruct the input image from layers 'conv1\_1' (a), 'conv2\_1' (b), 'conv3\_1' (c), 'conv4\_1' (d) and 'conv5\_1' (e) of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (a,b,c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). **Style Reconstructions.** On top of the original CNN representations we built a new feature space that captures the style of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from style representations built on different subsets of CNN layers ('conv1\_1' (a), 'conv1\_1' and 'conv2\_1' (b), 'conv1\_1', 'conv2\_1' and 'conv3\_1' (c), 'conv1\_1', 'conv2\_1', 'conv3\_1' and 'conv4\_1' (d), 'conv1\_1', 'conv2\_1', 'conv3\_1', 'conv4\_1' and 'conv5\_1' (e)). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

**Content reconstruction:** gradient descent on a white-noise image to find an image that matches the filter responses.

image. So let  $\vec{p}$  and  $\vec{x}$  be the original image and the image that is generated and  $P^l$  and  $F^l$  their respective feature representation in layer  $l$ . We then define the squared-error loss between the two feature representations

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2 . \quad (1)$$

**Style representation:**

Gram matrix  $G^l \in \mathcal{R}^{N_l \times N_l}$ , where  $G_{ij}^l$  is the inner product between the vectorised feature map  $i$  and  $j$  in layer  $l$ :

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l. \quad (3)$$

## Style reconstruction:

To generate a texture that matches the style of a given image (Fig 1, style reconstructions), we use gradient descent from a white noise image to find another image that matches the style representation of the original image. This is done by minimising the mean-squared distance between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated. So let  $\vec{a}$  and  $\vec{x}$  be the original image and the image that is generated and  $A^l$  and  $G^l$  their respective style representations in layer  $l$ . The contribution of that layer to the total loss is then

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2 \quad (4)$$

and the total loss is

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^L w_l E_l \quad (5)$$

where  $w_l$  are weighting factors of the contribution of each layer to the total loss (see below for specific values of  $w_l$  in our results).

Overall loss:

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$



Figure 3: Detailed results for the style of the painting *Composition VII* by Wassily Kandinsky. The rows show the result of matching the style representation of increasing subsets of the CNN layers (see Methods). We find that the local image structures captured by the style representation increase in size and complexity when including style features from higher layers of the network. This can be explained by the increasing receptive field sizes and feature complexity along the network's processing hierarchy. The columns show different relative weightings between the content and style reconstruction. The number above each column indicates the ratio  $\alpha/\beta$  between the emphasis on matching the content of the photograph and the style of the artwork (see Methods).

# Fully Convolutional Networks for Semantic Segmentation

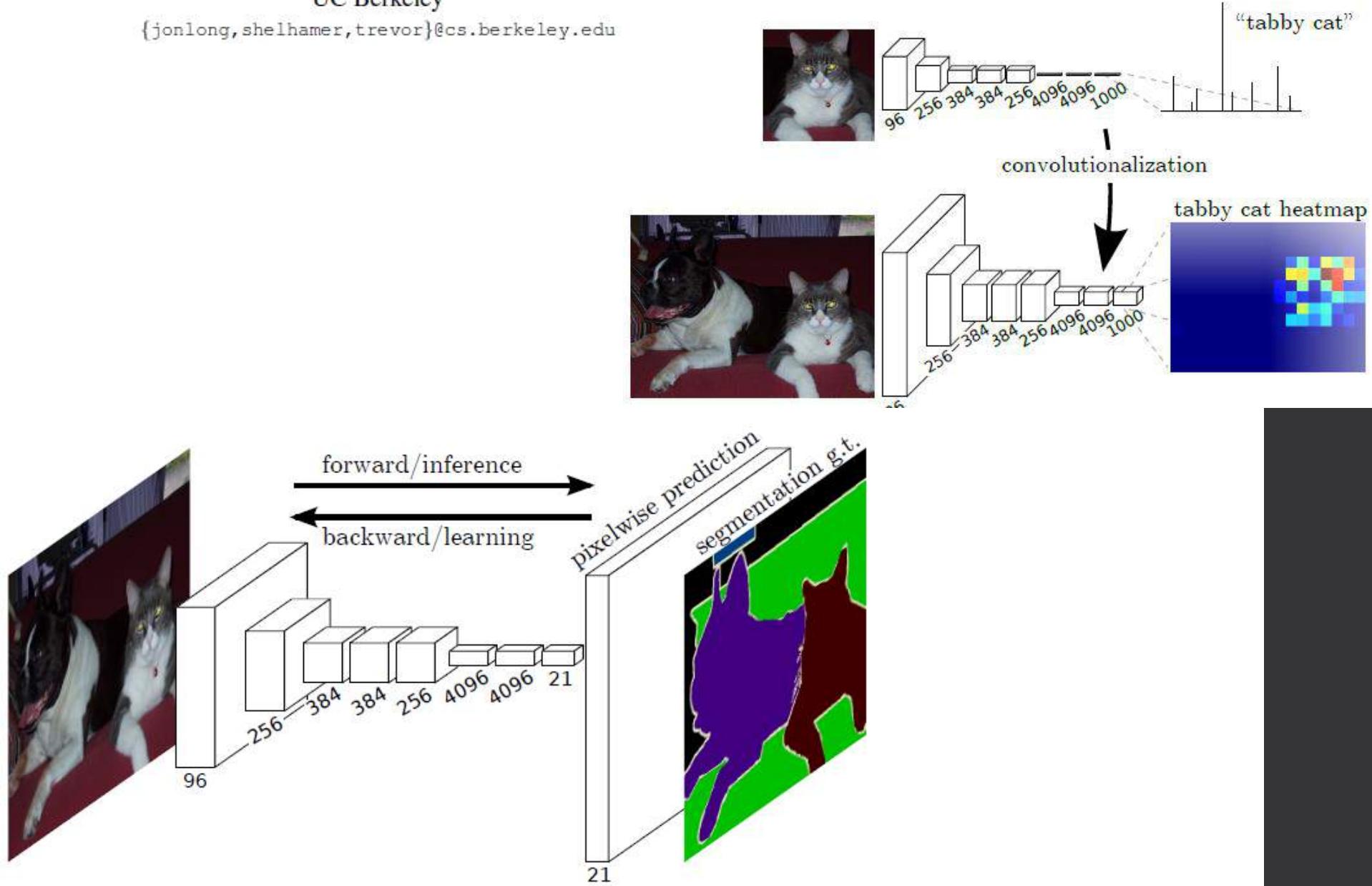
Jonathan Long\*

Evan Shelhamer\*

Trevor Darrell

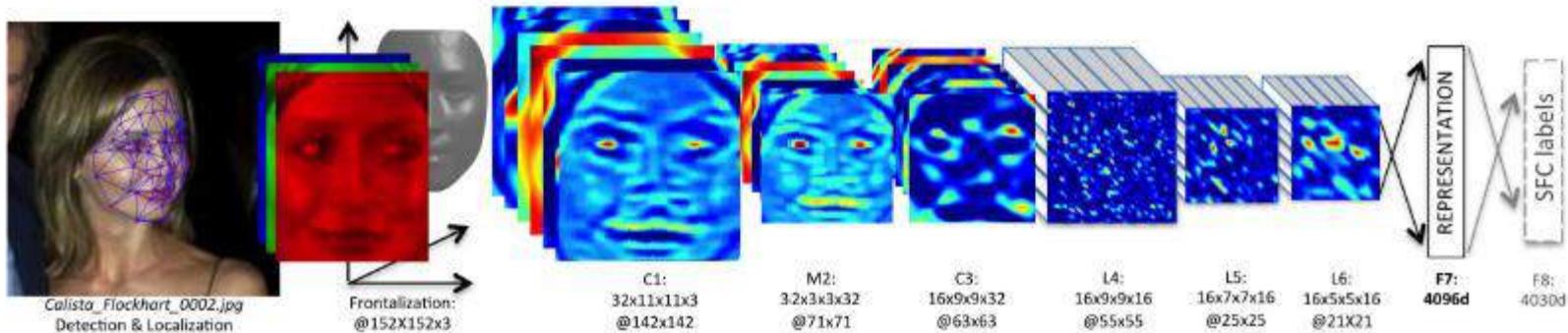
UC Berkeley

{jonlong, shelhamer, trevor}@cs.berkeley.edu



# Industry Deployment

- Used in Facebook, Google, Microsoft
- Image Recognition, Speech Recognition, ....
- Fast at test time



Taigman et al. DeepFace: Closing the Gap to Human-Level Performance in Face Verification,  
CVPR'14

# Other Applications

- Tracking (Bazzani et. al. 2010, and many others)



- Pose estimation (Toshev et al. 2013, Jain et al., 2013, ...)



- Caption generation (Vinyals et al. 2015, Xu et al. 2015, ...)



# Convolutional networks for music recommendation

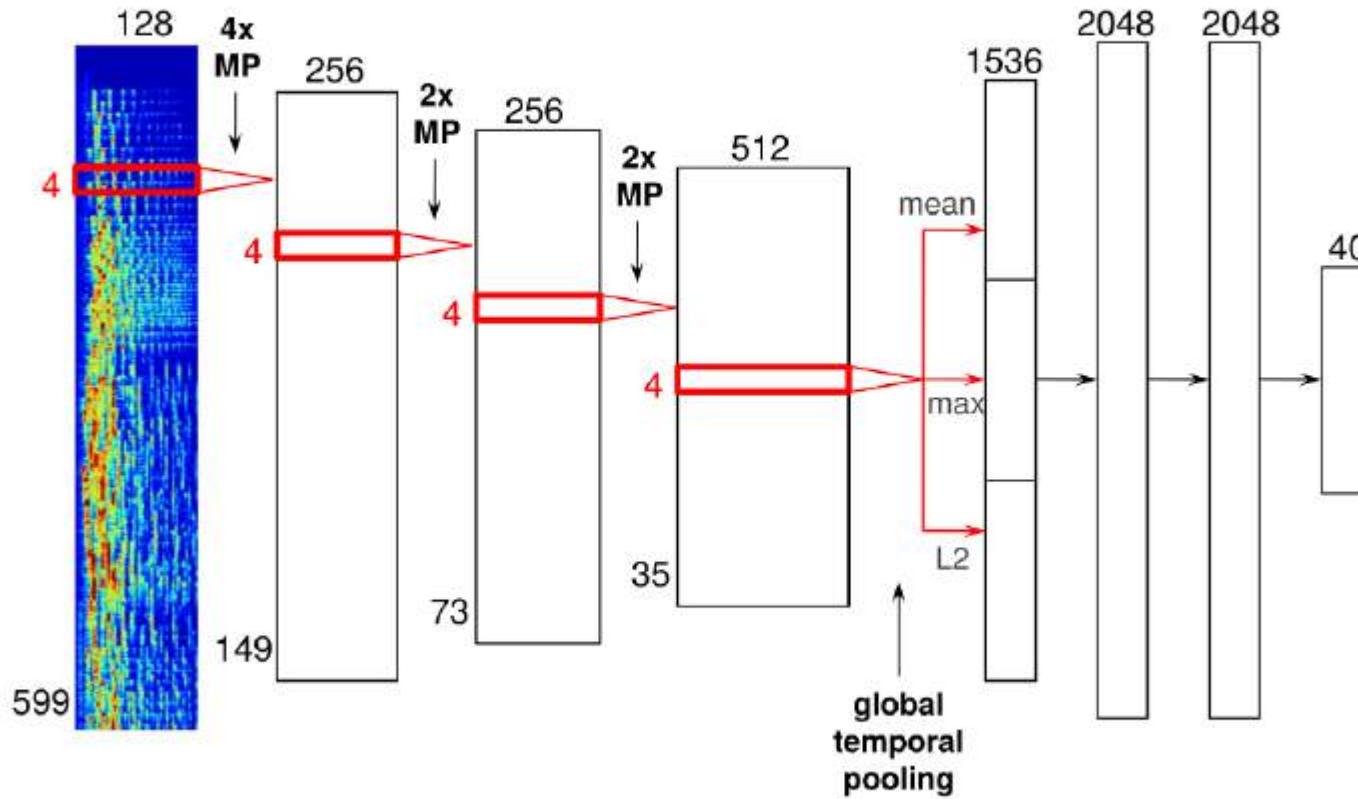


Image from: <http://benanne.github.io/2014/08/05/spotify-cnns.html>

Related work: Van den Oord, Dieleman & Schrauwen. Deep content-based music recommendation. In NIPS 2013

To wrap up

# CNNs: summary & future directions

- Less parameters
- Allows going deeper
- High flexibility
  - In operations
  - In organization of layers
  - In the overall architecture etc.
- Future directions:
  - Understanding them better
  - Making them deeper, faster and more efficient
  - Getting a trained expensive network and shrink it into a smaller cheaper one.
  - ...

# Binary networks

## XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks

Mohammad Rastegari<sup>†</sup>, Vicente Ordonez<sup>†</sup>, Joseph Redmon\*, Ali Farhadi<sup>†\*</sup>

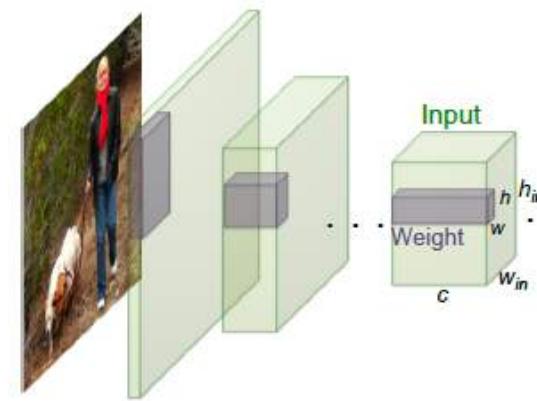
Allen Institute for AI<sup>†</sup>, University of Washington\*

{mohammadr, vicenteor}@allenai.org

{pjreddie, ali}@cs.washington.edu

**Abstract.** We propose two efficient approximations to standard convolutional neural networks: Binary-Weight-Networks and XNOR-Networks. In Binary-Weight-Networks, the filters are approximated with binary values resulting in  $32\times$  memory saving. In XNOR-Networks, both the filters and the input to convolutional layers are binary. XNOR-Networks approximate convolutions using primarily binary operations. This results in  $58\times$  faster convolutional operations and  $32\times$  memory savings. XNOR-Nets offer the possibility of running state-of-the-art networks on CPUs (rather than GPUs) in real-time. Our binary networks are simple, accurate, efficient, and work on challenging visual tasks. We evaluate our approach on the ImageNet classification task. The classification accuracy with a Binary-Weight-Network version of AlexNet is only 2.9% less than the full-precision AlexNet (in top-1 measure). We compare our method with recent network binarization methods, BinaryConnect and BinaryNets, and outperform these methods by large margins on ImageNet, more than 16% in top-1 accuracy.

# Binary networks



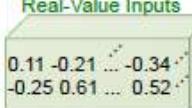
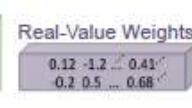
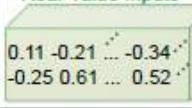
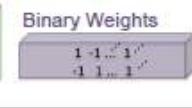
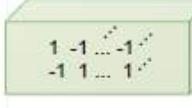
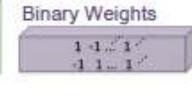
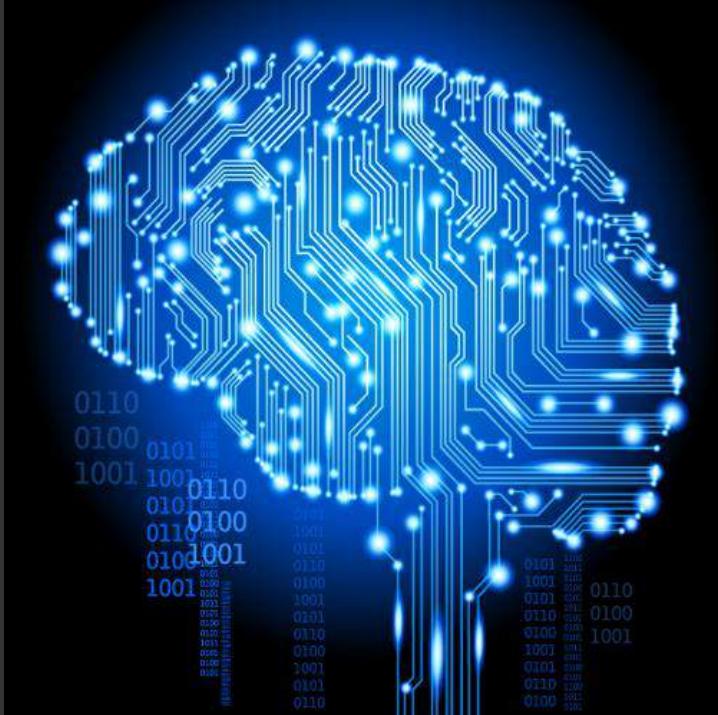
	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Time Saving on CPU (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	<b>Real-Value Inputs</b>  <b>Real-Value Weights</b> 	+ , - , $\times$	1x	1x	%56.7
Binary Weight	<b>Real-Value Inputs</b>  <b>Binary Weights</b> 	+ , -	~32x	~2x	%53.8
BinaryWeight Binary Input (XNOR-Net)	<b>Binary Inputs</b>  <b>Binary Weights</b> 	XNOR , bitcount	~32x	~58x	%44.2

Fig. 1: We propose two efficient variations of convolutional neural networks. **Binary-Weight-Networks**, when the weight filters contains binary values. **XNOR-Networks**, when both weigh and input have binary values. These networks are very efficient in terms of memory and computation, while being very accurate in natural image classification. This offers the possibility of using accurate vision techniques in portable devices with limited resources.

# CENG 783

## Special topics in Deep Learning



© AlchemyAPI

*Recurrent Neural Networks*



# Sequence Labeling/Modeling: **Motivation**

# Why do we need them?

*Foreign Minister.*



FOREIGN MINISTER.



THE SOUND OF

# Different types of sequence learning/recognition problems

- Sequence Classification
  - A sequence to a label
  - E.g., recognizing a single spoken word
  - Length of the sequence is fixed
  - Why RNNs then? Because sequential modeling provides robustness against translations and distortions.
- Segment Classification
  - Segments in a sequence correspond to labels
- Temporal Classification
  - General case: sequence (input) to sequence (label) modeling.
  - No clue about where input or label starts.

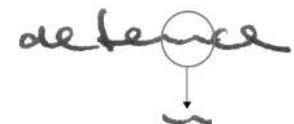


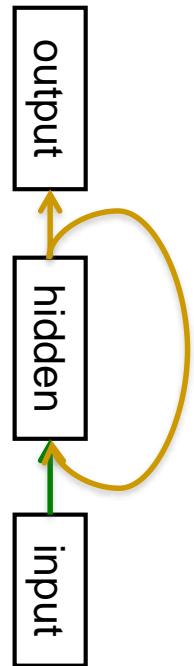
Fig. 2.3 Importance of context in segment classification. The word ‘defence’ is clearly legible. However the letter ‘n’ in isolation is ambiguous.

# Recurrent Neural Networks

# Recurrent Neural Networks (RNNs)



Feed-forward  
networks

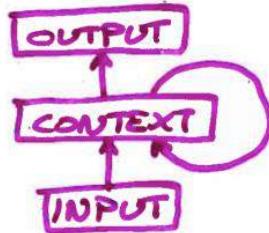


Recurrent  
networks

- RNNs are very powerful because:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.
- More formally, RNNs are Turing complete.

# Recurrent Neural Nets

- Temporal pattern recognition



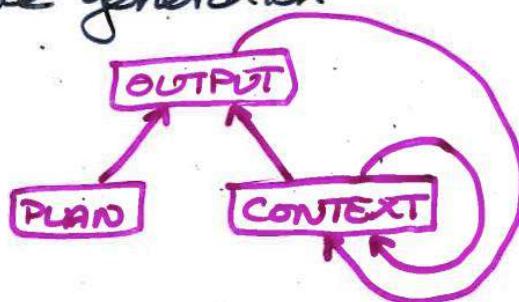
INPUT<sub>1</sub>, INPUT<sub>2</sub>, INPUT<sub>3</sub>, ..., INPUT<sub>t</sub>  
⇒ OUTPUT<sub>t</sub>

e.g., speech recognition

e.g., event recognition

e.g., natural language understand

- Sequence generation



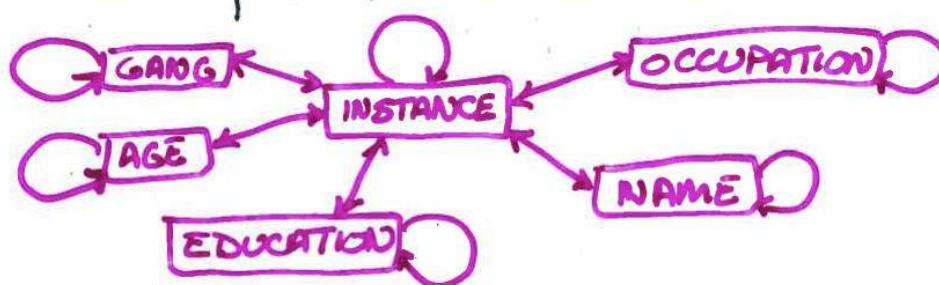
PLAN ⇒ OUTPUT<sub>1</sub>, OUTPUT<sub>2</sub>,  
OUTPUT<sub>3</sub>, ..., OUTPUT<sub>t</sub>

e.g., speech production

e.g., motor control

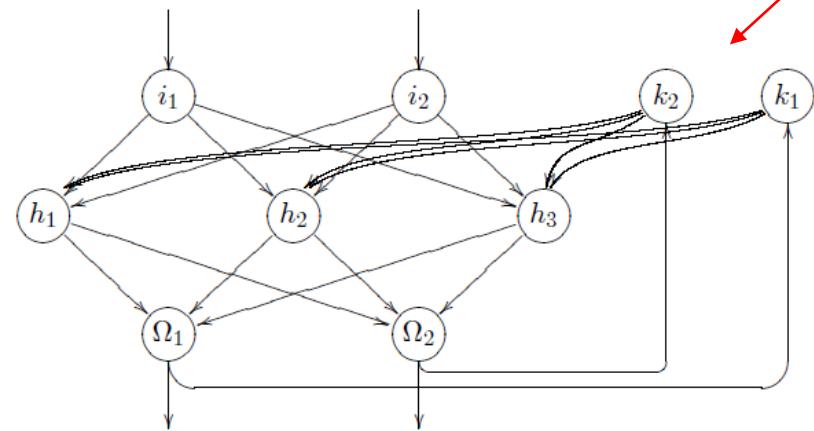
e.g., planning and acting

- Pattern completion / constraint satisfaction

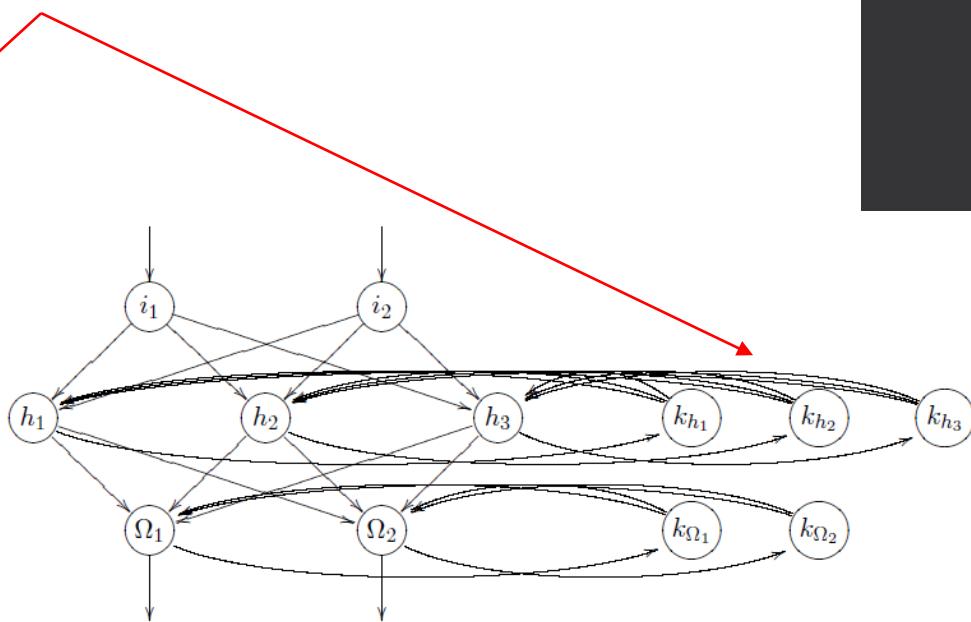


# Some examples

“context” neurons



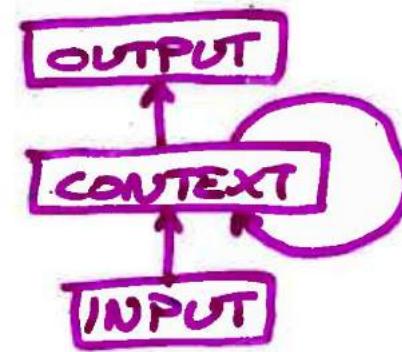
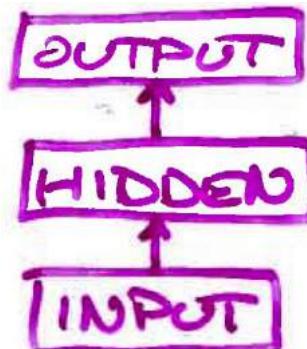
Jordan Networks



Elman Networks

# Challenge

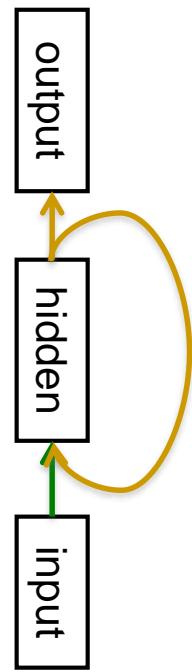
- Back propagation is designed for feedforward nets
- What would it mean to back propagate through a recurrent network?
  - error signal would have to travel back in time



# Unfolding

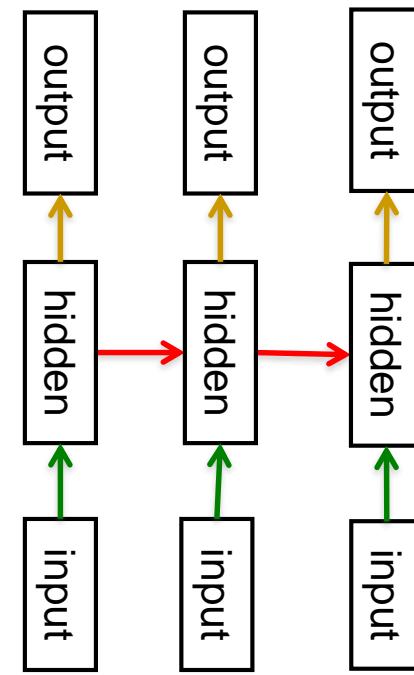


Feed-forward  
networks



Recurrent  
networks

Unfolding



t=0      t=1      t=2

time →

# Unfolding (another example)

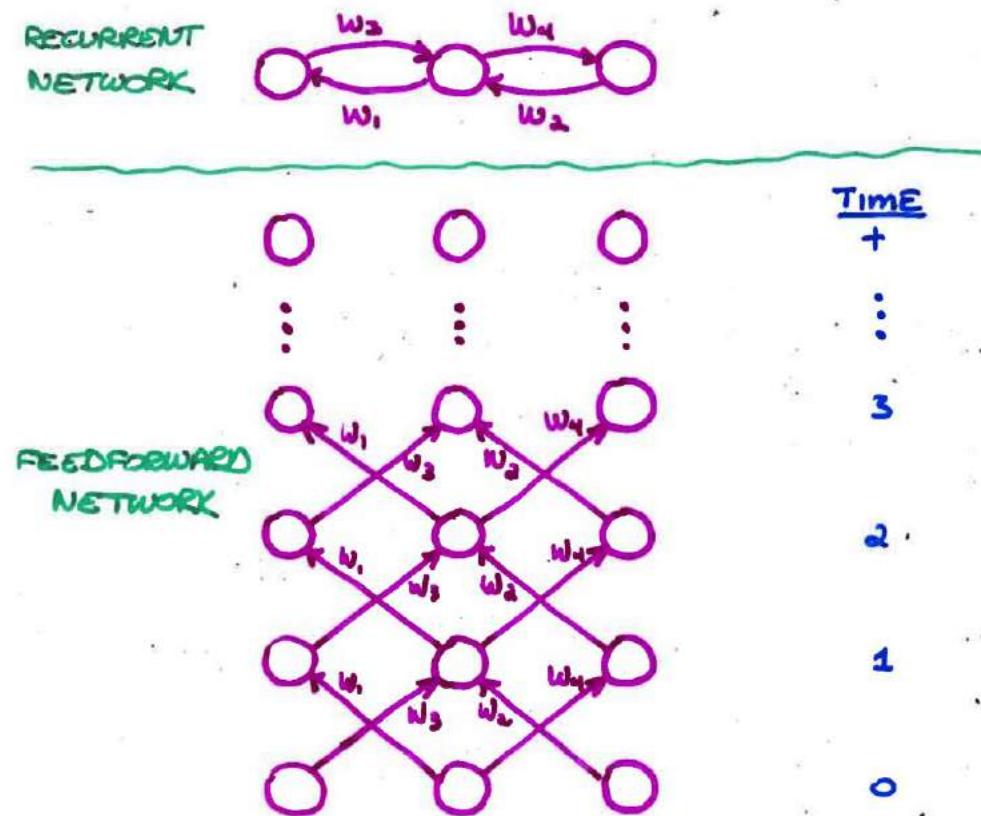
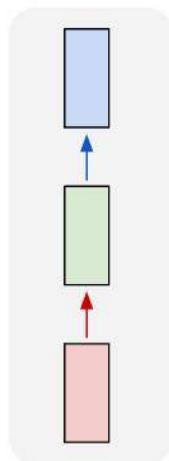
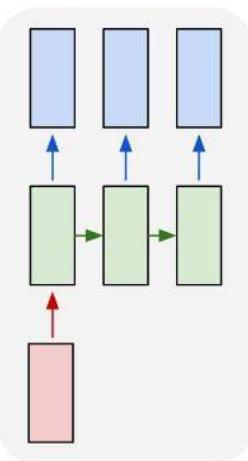


Figure: Michael Mozer

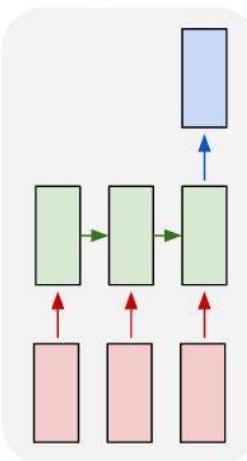
one to one



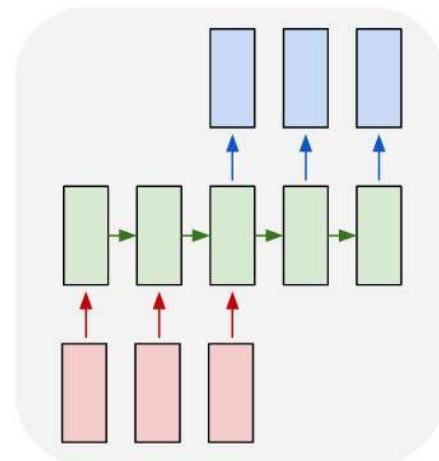
one to many



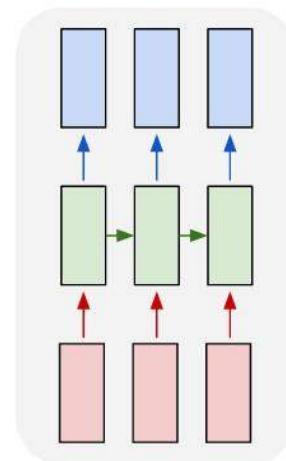
many to one



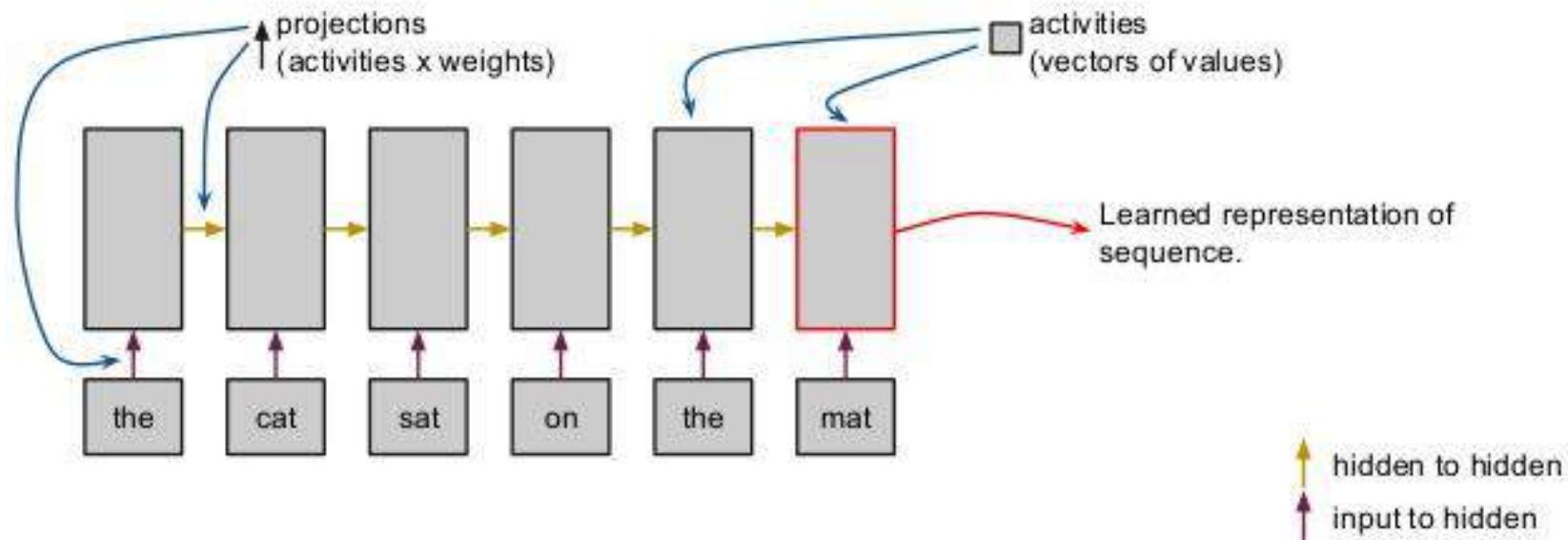
many to many



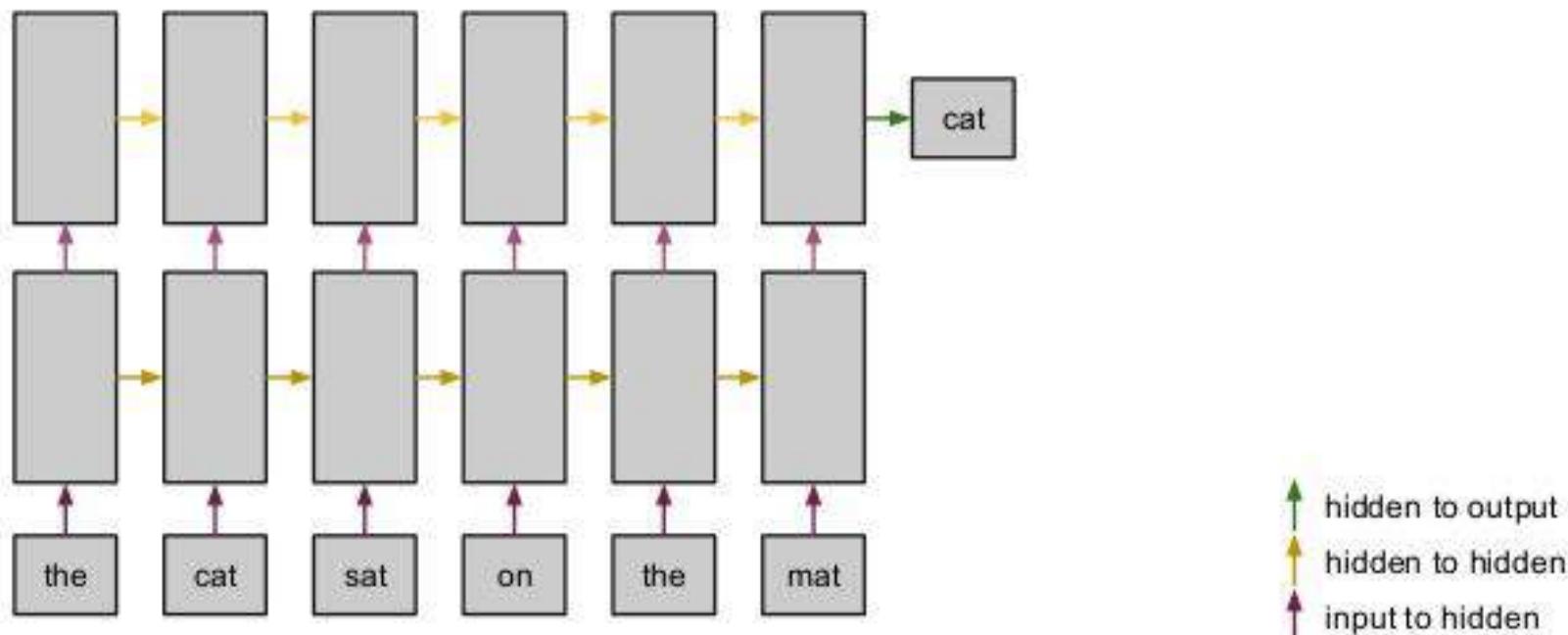
many to many



# How an RNN works



# You can stack them too



# Unfolding implications

- Entails duplication of weights => weight sharing
- Sharing weights means their gradients will be accumulated over time and reflected on the weights
- Unfolded network has the **same dynamics of the RNN** for a **fixed number of time steps!**

# Back-propagation Through Time

# Reminder: Backpropagation with weight constraints

- It is easy to modify the backprop algorithm to incorporate linear constraints between the weights.
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
  - So if the weights started off satisfying the constraints, they will continue to satisfy them.

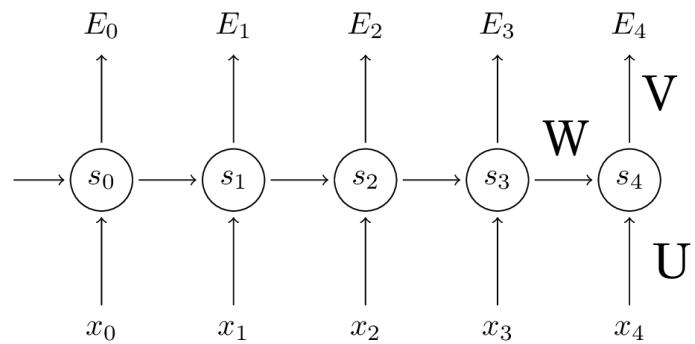
*To constrain:*  $w_1 = w_2$   
*we need:*  $\Delta w_1 = \Delta w_2$

*compute:*  $\frac{\partial E}{\partial w_1}$  and  $\frac{\partial E}{\partial w_2}$

*use*  $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$  *for*  $w_1$  *and*  $w_2$

# Backpropagation through time

- We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.
- We can also think of this training algorithm in the time domain:
  - The forward pass builds up a stack of the activities of all the units at each time step.
  - The backward pass peels activities off the stack to compute the error derivatives at each time step.
  - After the backward pass we add together the derivatives at all the different times for each weight.



Cross-entropy loss:

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

$$\begin{aligned} E(y, \hat{y}) &= \sum_t E_t(y_t, \hat{y}_t) \\ &= - \sum_t y_t \log \hat{y}_t \end{aligned}$$

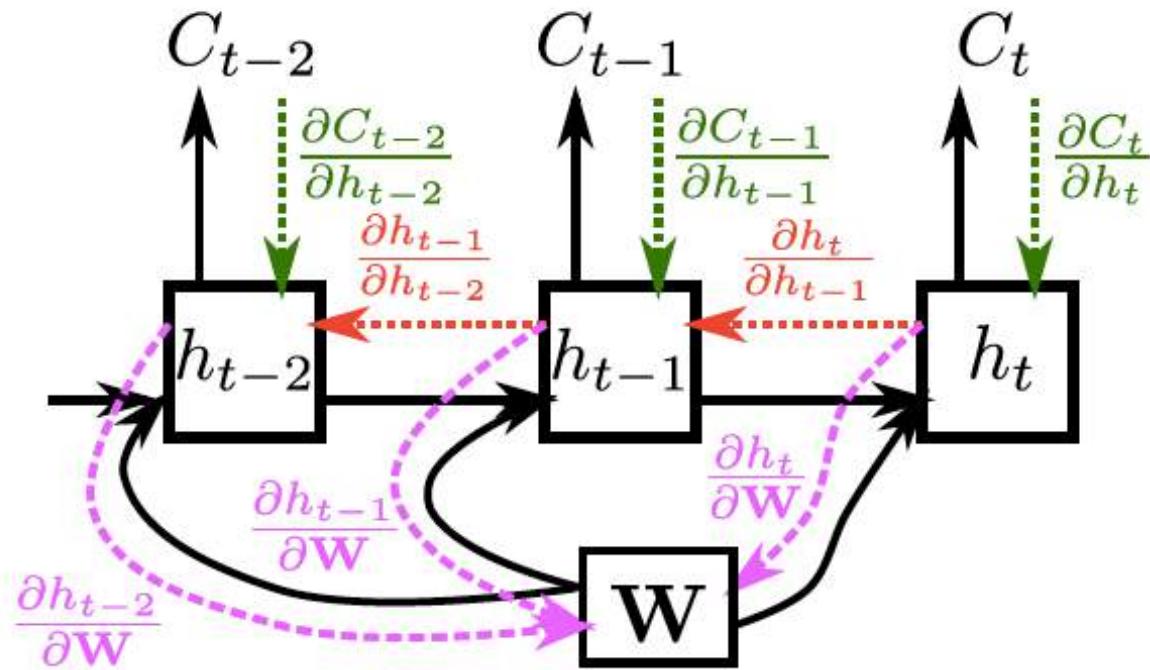
$$\begin{aligned} s_t &= \tanh(Ux_t + Ws_{t-1}) \\ \hat{y}_t &= \text{softmax}(Vs_t) \end{aligned}$$

Accumulate errors over time  
(treat the whole sequence as a  
training example):

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

$$\begin{aligned} \frac{\partial E_3}{\partial V} &= \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial V} \\ &= \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial z_3} \frac{\partial z_3}{\partial V} \\ &= (\hat{y}_3 - y_3) \otimes s_3 \\ z_3 &= Vs_3 \end{aligned}$$

$$\begin{aligned} \frac{\partial E_3}{\partial W} &= \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial W} \\ \frac{\partial E_3}{\partial W} &= \sum_{k=0}^3 \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W} \end{aligned}$$



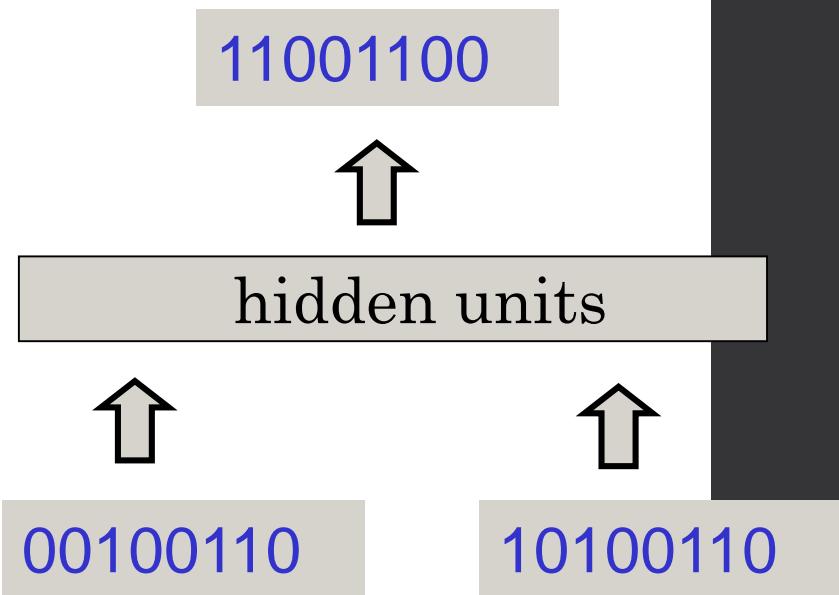
$$\frac{\partial C_t}{\partial \mathbf{W}} = \sum_{t'=1}^t \frac{\partial C_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial \mathbf{W}}, \text{ where } \frac{\partial h_t}{\partial h_{t'}} = \prod_{k=t'+1}^t \frac{\partial h_k}{\partial h_{k-1}}$$

# An irritating extra issue

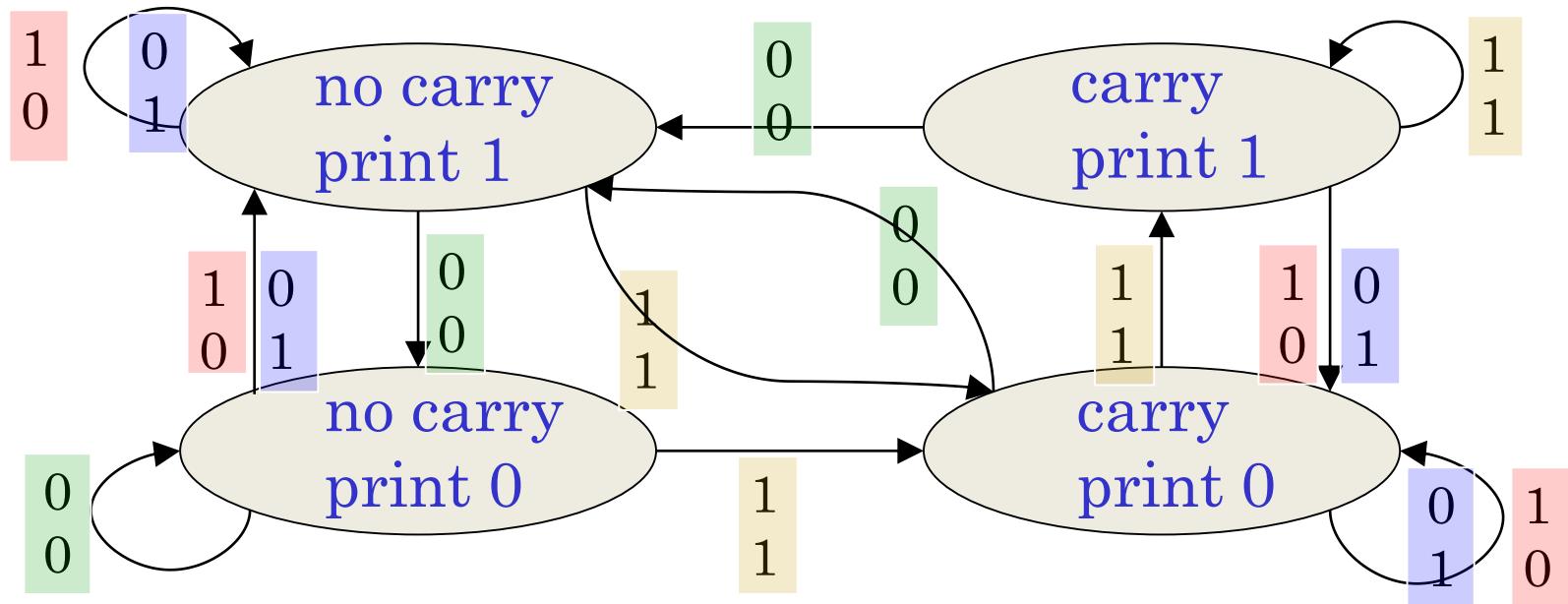
- We need to specify the initial activity state of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
  - Start off with an initial random guess for the initial states.
  - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
  - Adjust the initial states by following the negative gradient.

# A good toy problem for a recurrent network

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture efficiently.
  - We must decide in advance the maximum number of digits in each number.
  - The processing applied to the beginning of a long number does not generalize to the end of the long number because it uses different weights.
- As a result, feedforward nets do not generalize well on the binary addition task.



# The algorithm for binary addition



This is a finite state automaton. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

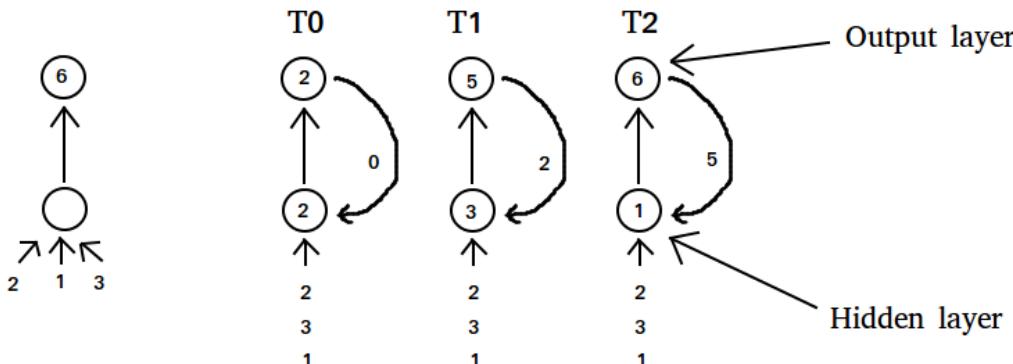
# A recurrent net for binary addition

- The network has two input units and one output unit.
- It is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
  - It takes one time step to update the hidden units based on the two input digits.
  - It takes another time step for the hidden units to cause the output.

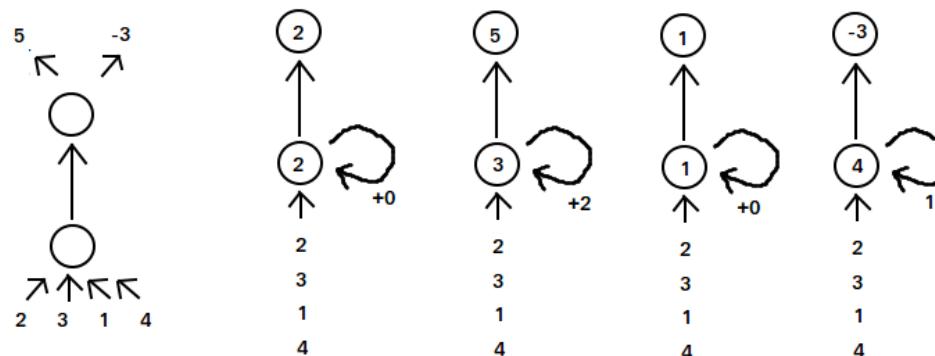
$$\begin{array}{r} 00110100 \\ 01001101 \\ \hline 10000001 \end{array}$$

← time

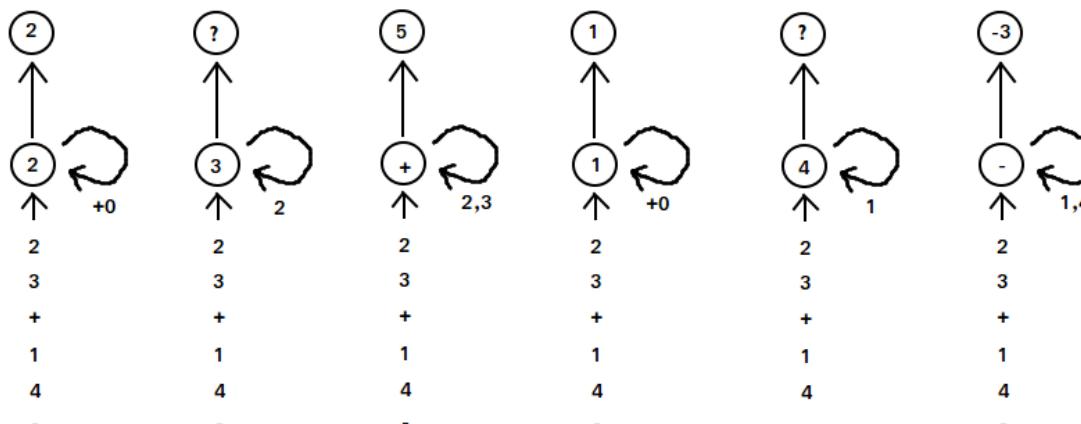
## Sum of three numbers



## Addition and subtraction in the same network



## Generalized network: Any length, any operation



# The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, it's very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.

# BPTT vs RTRL

One complaint about BPTT is that it requires saving activity states of all units at all previous times.

Space complexity of BPTT:  $O(NT)$

Time complexity of BPTT:  $O(N^2T)$

$N$ : # units

$T$ : # time steps

Williams & Zipser have developed an alternative algorithm called real-time recurrent learning (RTRL) which does not require storing activity states.

Space complexity of RTRL:  $O(N^3)$

Time complexity of RTRL:  $O(N^4)$

Both RTRL and BPTT compute exact gradient of the error with respect to the weights.

∴ have the same power and limitations

For large  $T$ , RTRL may be more efficient.

Not particularly useful in practice.

# Problem With BPTT and RTRL

while BPTT is in principle capable of learning relationships among temporal events, in practice it is weak.

E.g., detecting contingencies spanning temporal gaps

...  $e_1 \dots e_a \dots$

$e_a$  dependent on  $e_1$

Input is a sequence of symbols: A, B, C, D, E, F, X, Y

Task is to predict next symbol in sequence

Sample sequences:

X A B X  
Y A B Y  
↑ gap=2 ↑

X A B C D E F X  
Y A B C D E F Y  
↑ gap=6 ↑

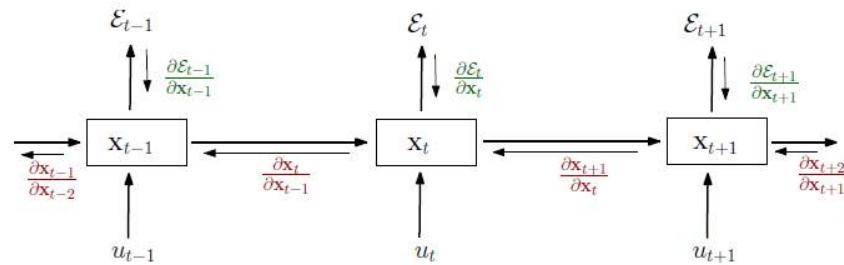
Learning two-sequence training set with a sequence-recg. architecture and BPTT is not reliable for gaps of 4 or more

gap	% failures after 10k epochs
2	0
4	36
6	92
8	100

(Mozer, 1992)

Problem: BPTT ok at discovering structure that is local in time, but not good at handling structure at a more global scale (long temporal intervals, & involving high order statistics).

# Exploding and vanishing gradients problem



- Solution 1:** Gradient clipping for exploding gradients:

**Algorithm 1** Pseudo-code for norm clipping

```

 $\hat{g} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$ 
if  $\|\hat{g}\| \geq \text{threshold}$  then
     $\hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|} \hat{g}$ 
end if

```

- For vanishing gradients: Regularization term that penalizes changes in the magnitudes of back-propagated gradients

$$\Omega = \sum_k \Omega_k = \sum_k \left( \frac{\left\| \frac{\partial \mathcal{E}}{\partial \mathbf{x}_{k+1}} \frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_k} \right\|}{\left\| \frac{\partial \mathcal{E}}{\partial \mathbf{x}_{k+1}} \right\|} - 1 \right)^2$$

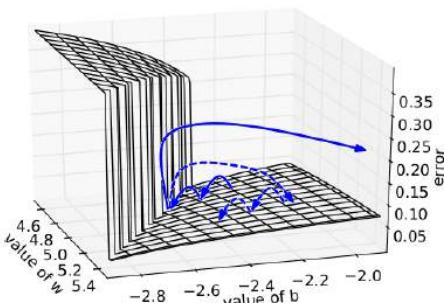
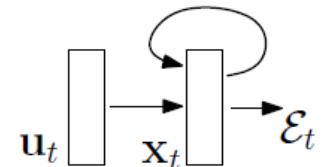


Figure 6. We plot the error surface of a single hidden unit recurrent network, highlighting the existence of high curvature walls. The solid lines depicts standard trajectories that gradient descent might follow. Using dashed arrow the diagram shows what would happen if the gradients is rescaled to a fixed size when its norm is above a threshold.



## On the difficulty of training recurrent neural networks

Razvan Pascanu

PASCANUR@IRO.UMONTREAL.CA

Université de Montréal, 2920, chemin de la Tour, Montréal, Québec, Canada, H3T 1J8

Tomas Mikolov

T.MIKOLOV@GMAIL.COM

Speech@FIT, Brno University of Technology, Brno, Czech Republic

Yoshua Bengio

YOSHUA.BENGIO@UMONTREAL.CA

Université de Montréal, 2920, chemin de la Tour, Montréal, Québec, Canada, H3T 1J8

# Exploding and vanishing gradients problem

- Solution 2:
  - Use methods like LSTM

# Long Short Term Memory (LSTM) Networks

# RNN

- Basic block diagram

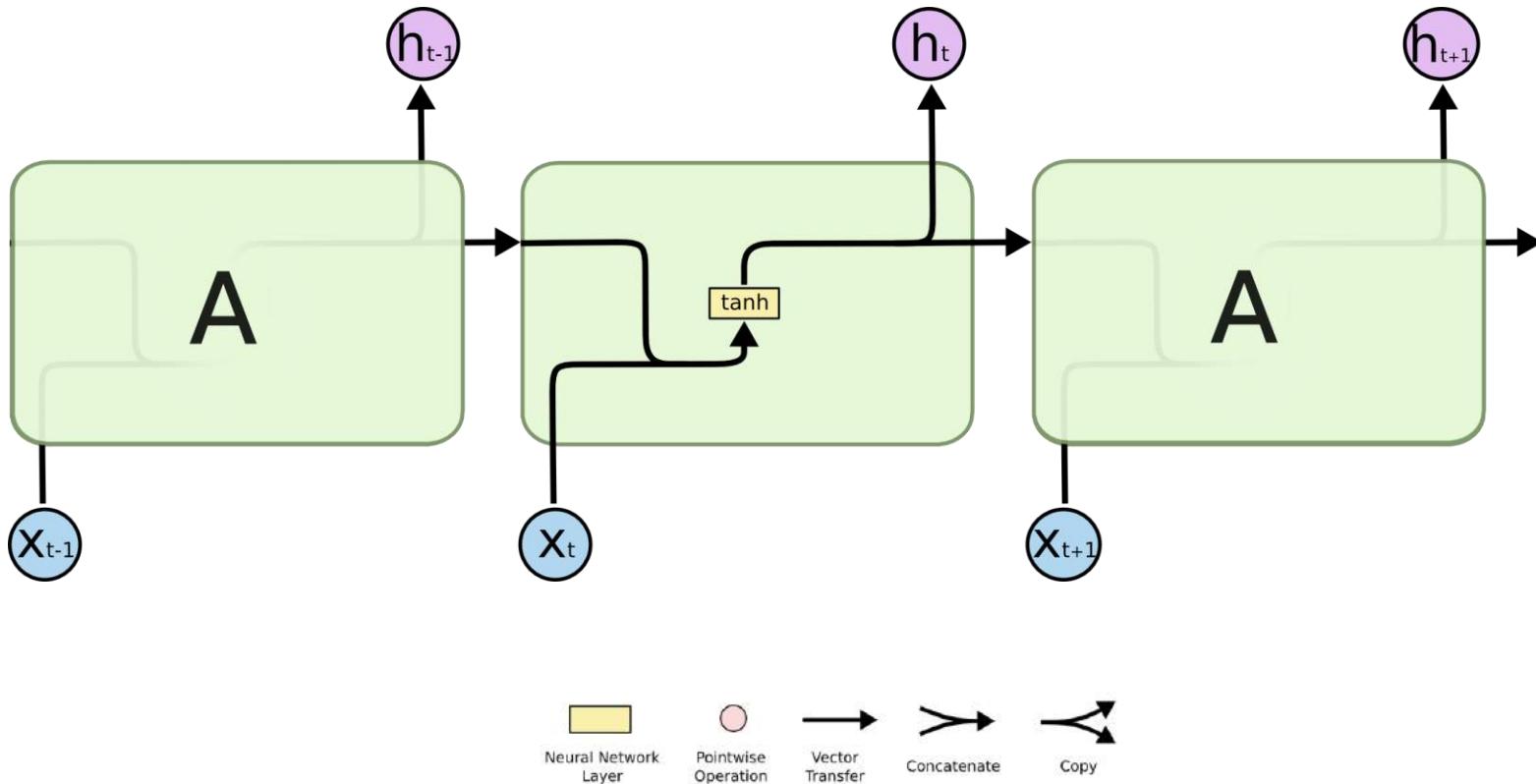
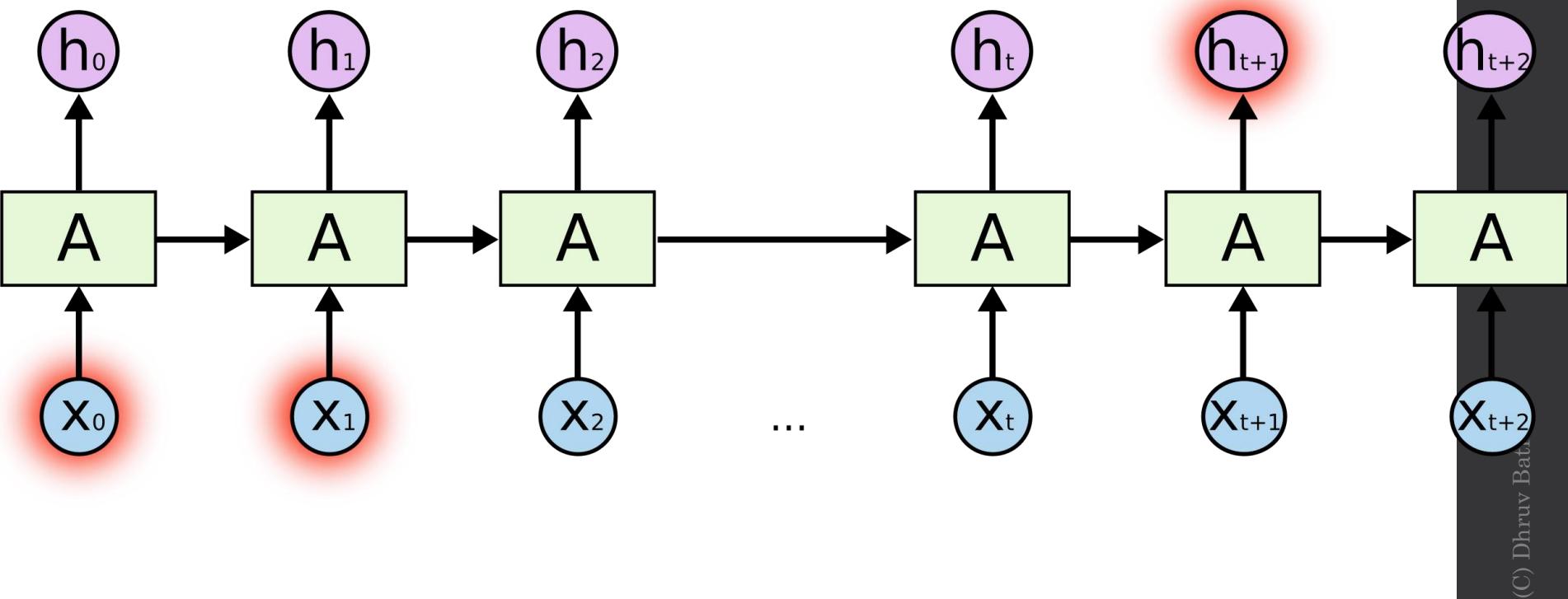


Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# Key Problem

- Learning long-term dependencies is hard



# Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.

# Meet LSTMs

- How about we explicitly encode memory?

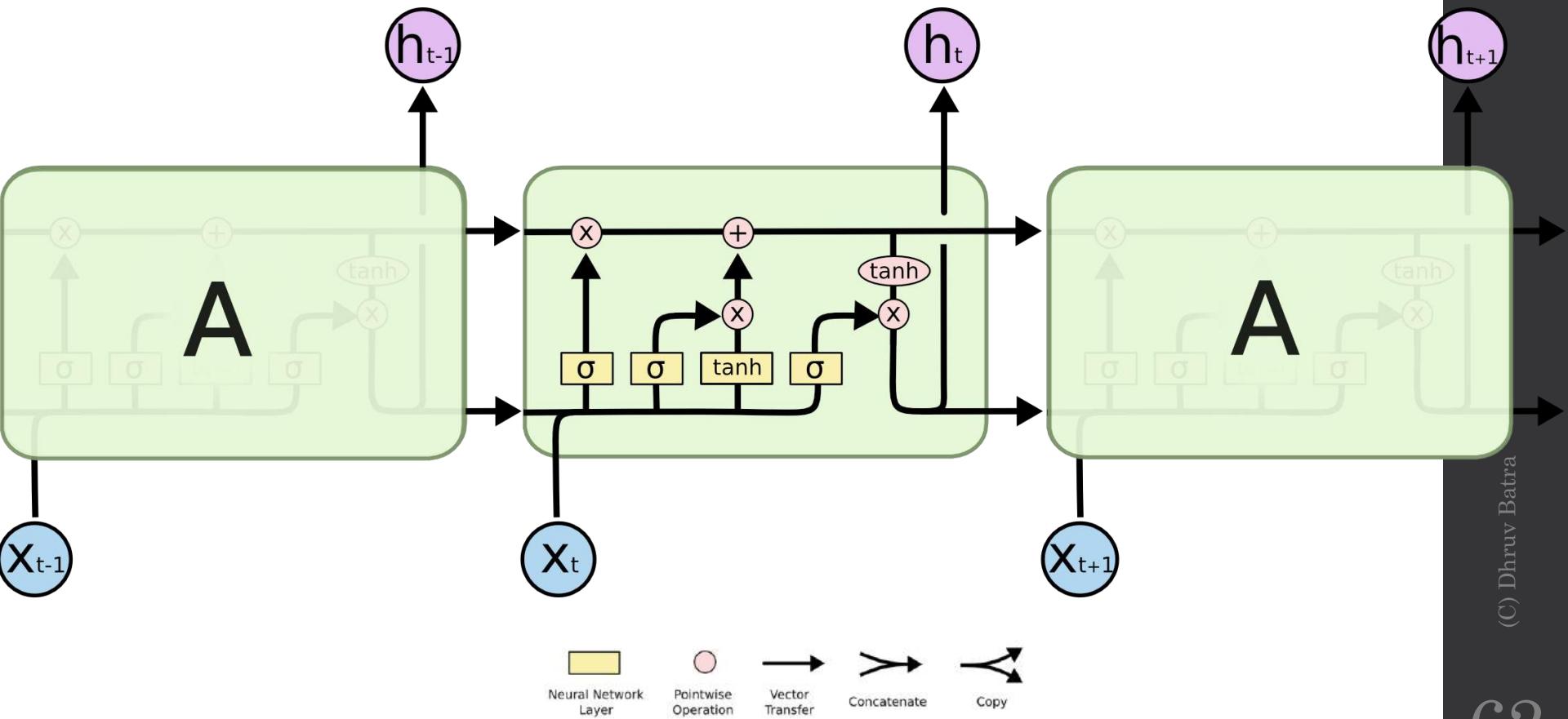


Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# LSTM in detail

- We first compute an activation vector,  $a$ :

$$a = W_x x_t + W_h h_{t-1} + b$$

- Split this into four vectors of the same size:

$$a_i, a_f, a_o, a_g \leftarrow a$$

- We then compute the values of the gates:

$$i = \sigma(a_i)$$

$$f = \sigma(a_f)$$

$$o = \sigma(a_o)$$

$$g = \tanh(a_g)$$

where  $\sigma$  is the sigmoid.

- The next cell state  $c_t$  and the hidden state  $h_t$ :

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

where  $\odot$  is the element-wise product of vectors

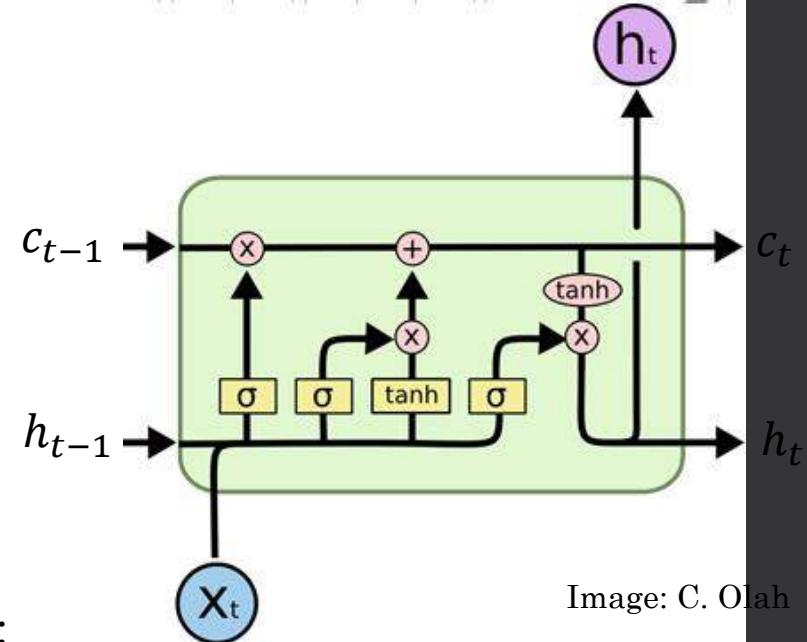


Image: C. Olah

Alternative formulation:

$$i_t = g(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = g(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = g(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

# LSTMs Intuition: Memory

- Cell State / Memory

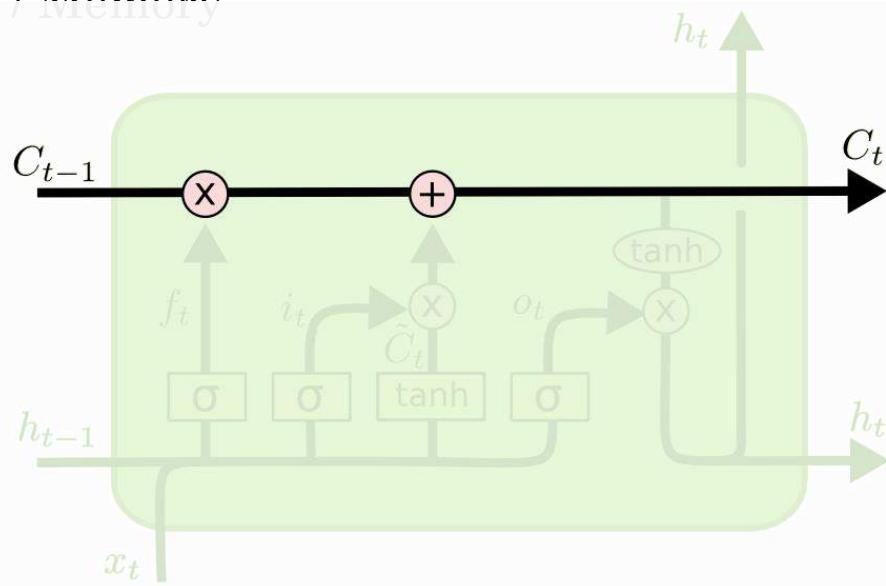
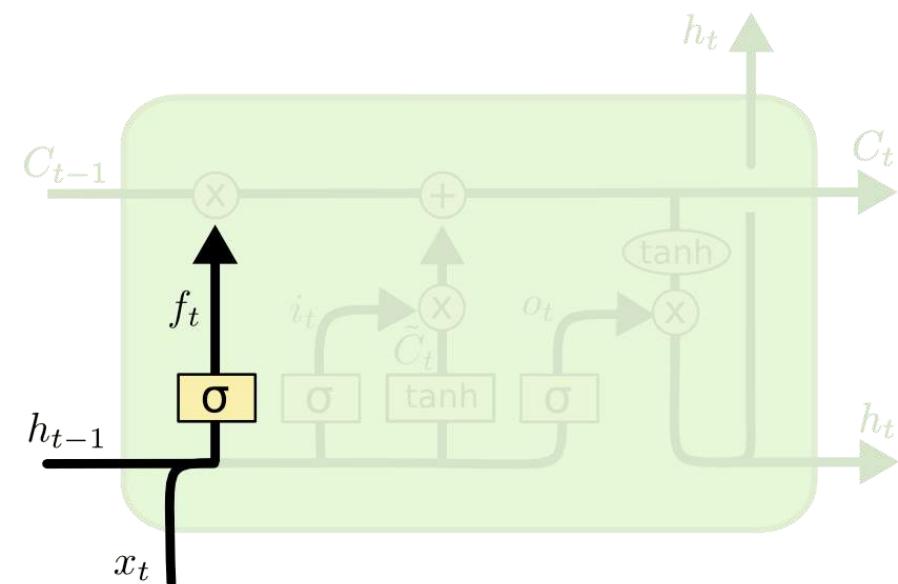


Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# LSTMs Intuition: Forget Gate

- Should we continue to remember this “bit” of information or not?

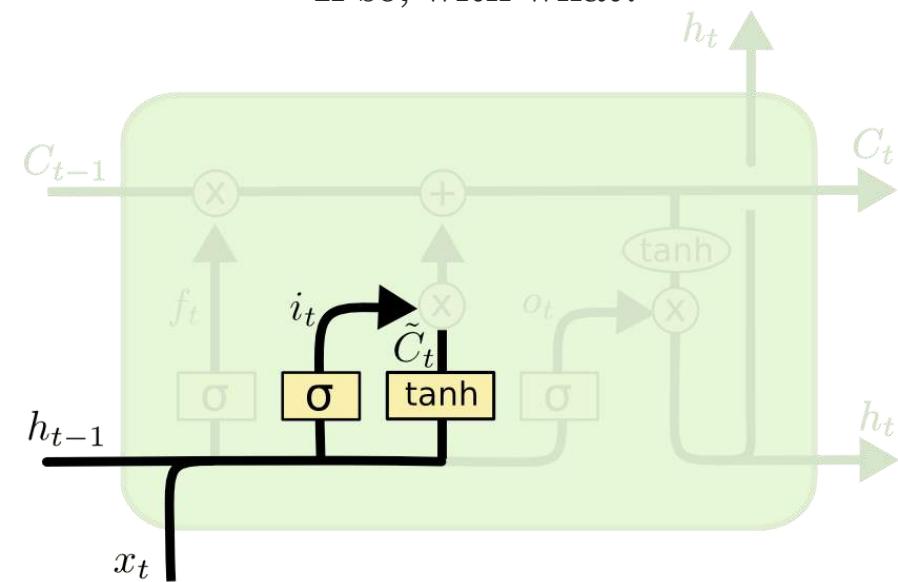


$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# LSTMs Intuition: Input Gate

- Should we update this “bit” of information or not?
  - If so, with what?

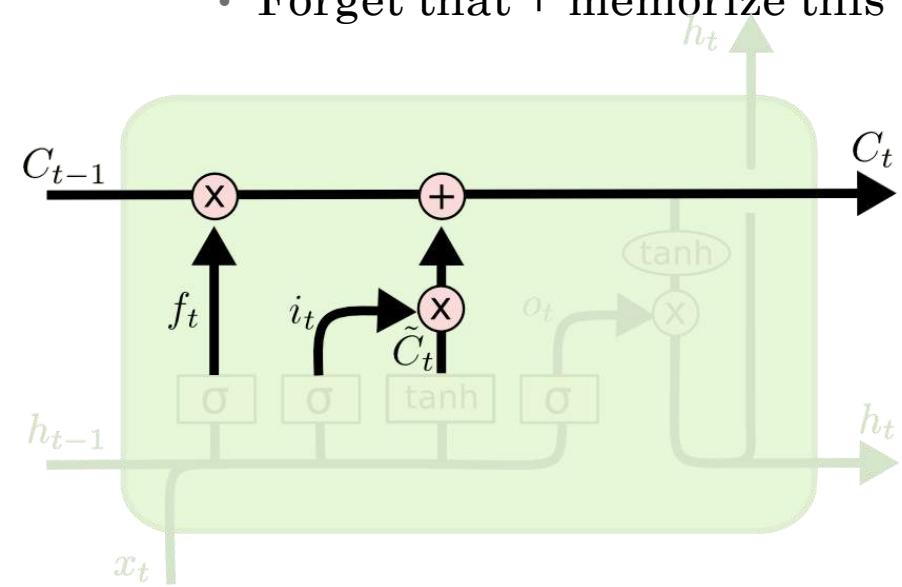


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

# LSTMs Intuition: Memory Update

- Forget that + memorize this

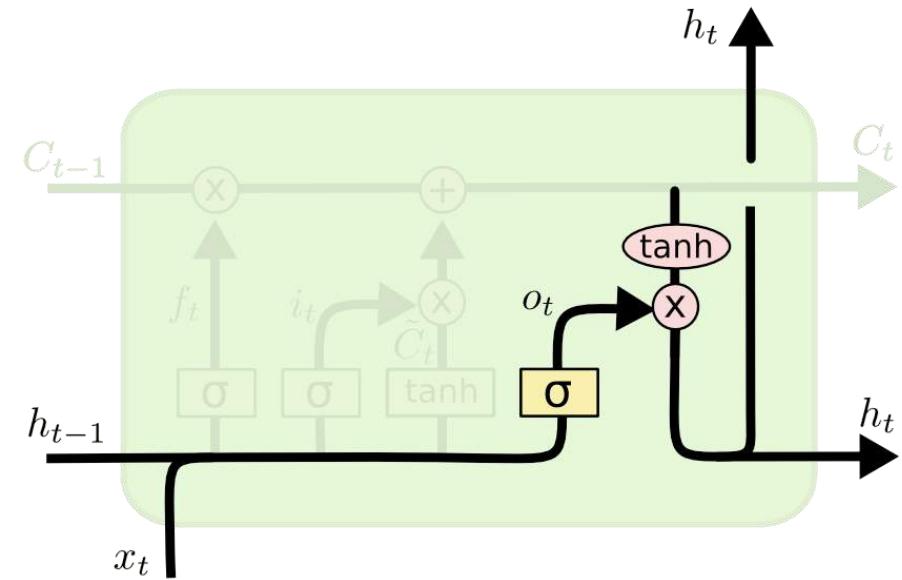


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# LSTMs Intuition: Output Gate

- Should we output this “bit” of information to “deeper” layers?



$$o_t = \sigma (W_o [ h_{t-1}, x_t ] + b_o)$$

$$h_t = o_t * \tanh (C_t)$$

# LSTMs

- A pretty sophisticated cell

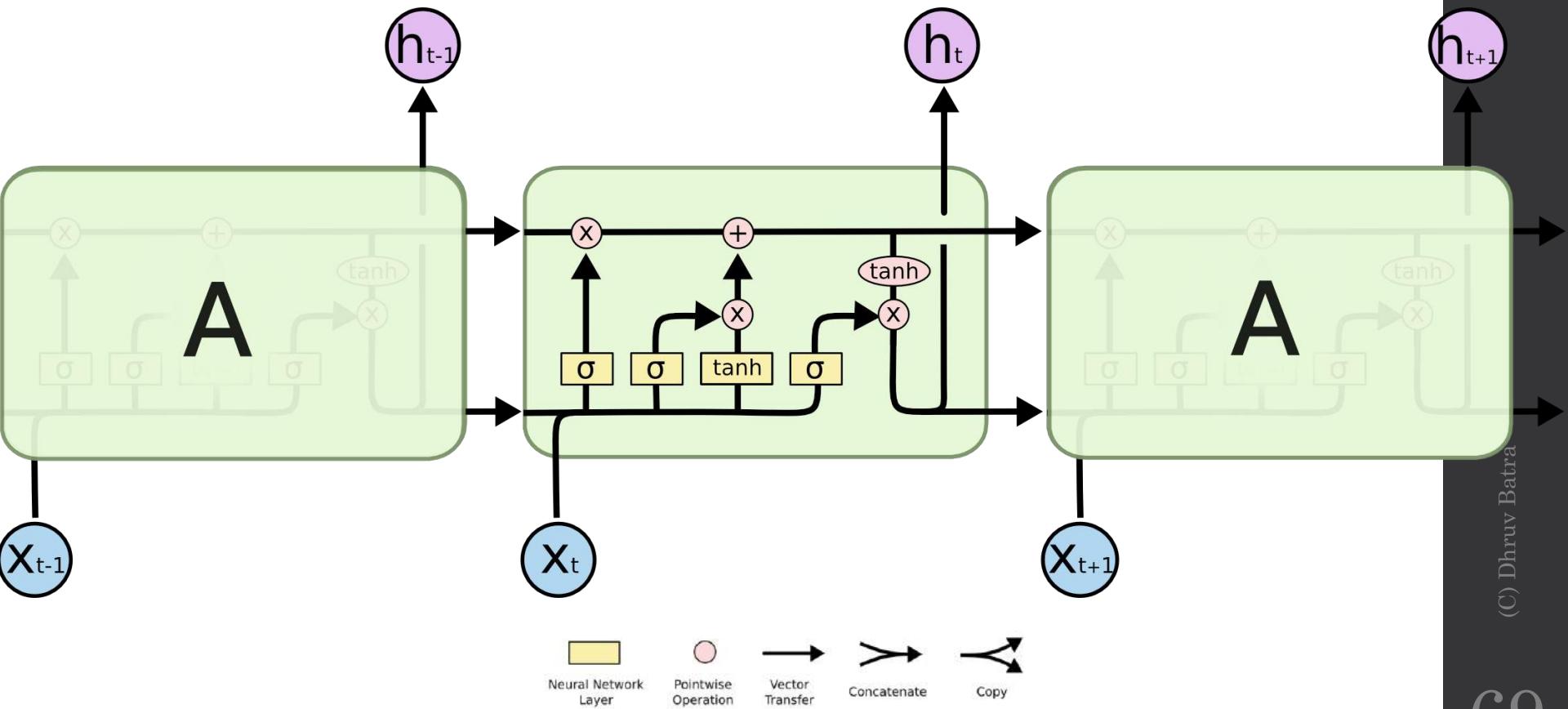
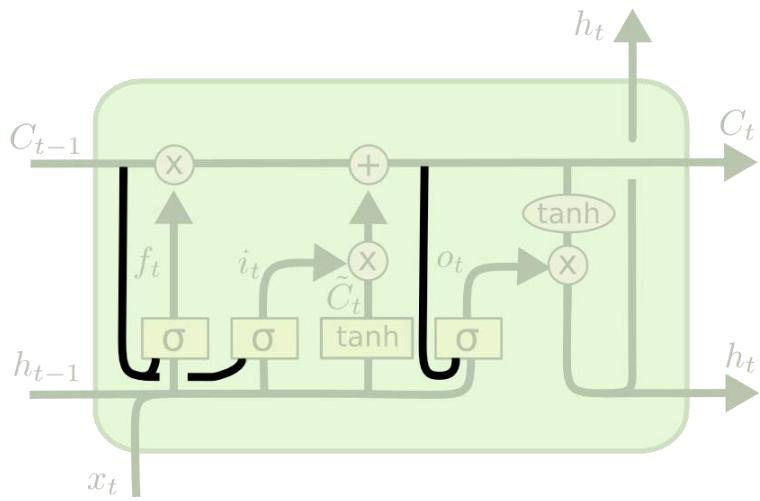


Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# LSTM Variants #1: Peephole Connections

- Let gates see the cell state / memory



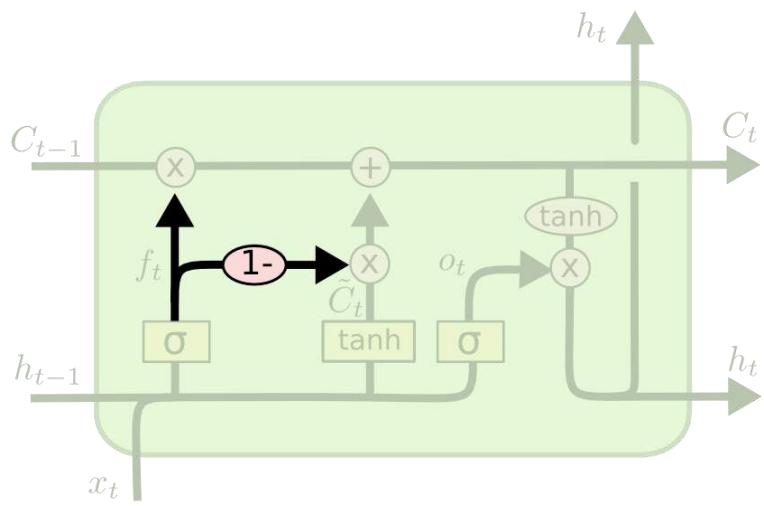
$$f_t = \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i)$$

$$o_t = \sigma(W_o \cdot [C_t, h_{t-1}, x_t] + b_o)$$

# LSTM Variants #2: Coupled Gates

- Only memorize new if forgetting old

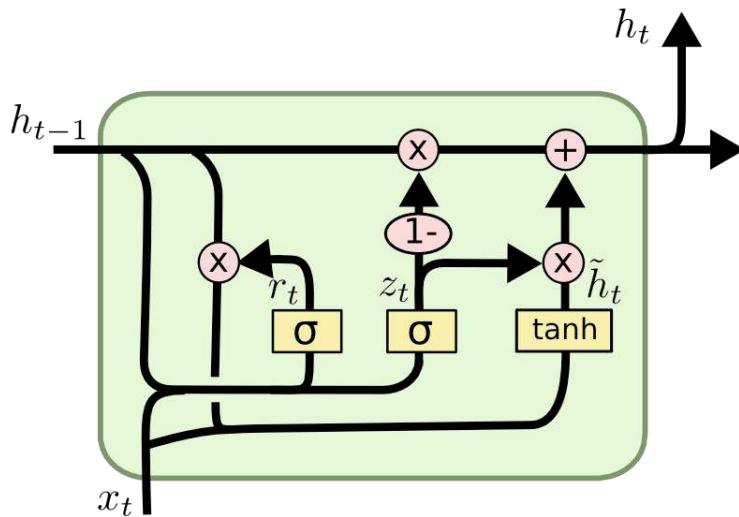


$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

Image Credit: Christopher Olah (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)

# LSTM Variants #3: Gated Recurrent Units

- Changes:
  - No explicit memory; memory = hidden output
  - Z = memorize new and forget old



$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

# References

- A very detailed explanation with nice figures

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Shaojie Bai<sup>1</sup> J. Zico Kolter<sup>2</sup> Vladlen Koltun<sup>3</sup>

4 Mar 2018

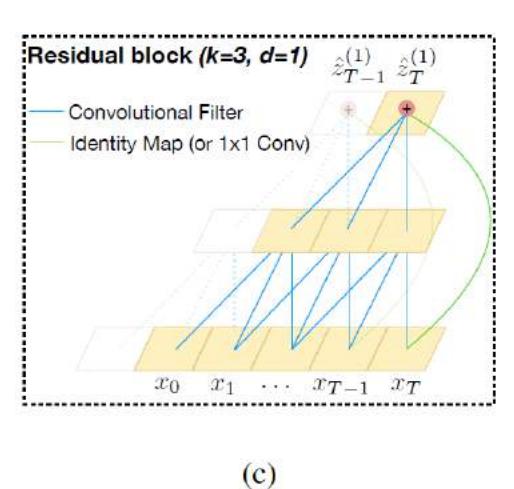
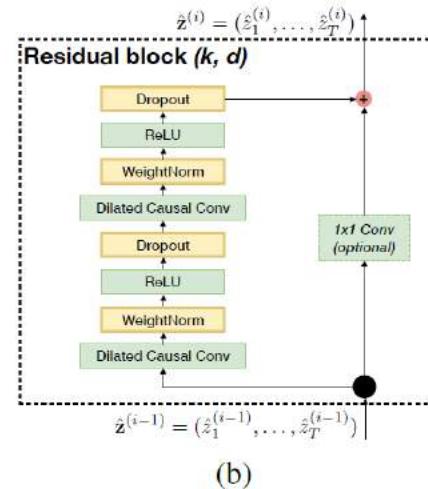
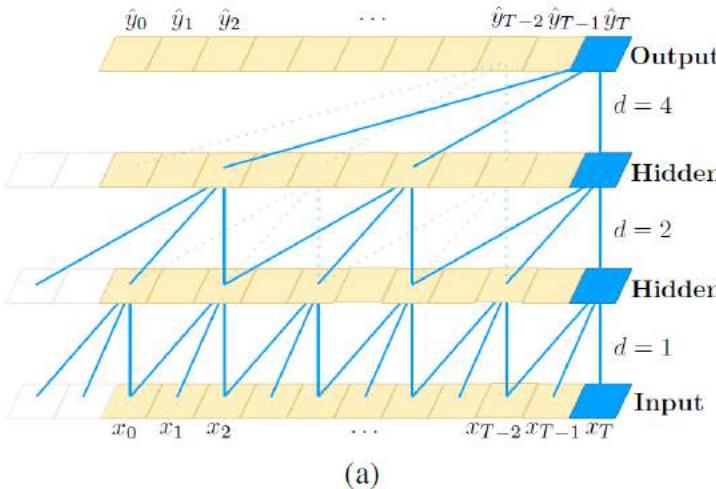


Figure 1. Architectural elements in a TCN. (a) A dilated causal convolution with dilation factors  $d = 1, 2, 4$  and filter size  $k = 3$ . The receptive field is able to cover all values from the input sequence. (b) TCN residual block. An  $1 \times 1$  convolution is added when residual input and output have different dimensions. (c) An example of residual connection in a TCN. The blue lines are filters in the residual function, and the green lines are identity mappings.

chine translation (van den Oord et al., 2016; Kalchbrenn et al., 2016; Dauphin et al., 2017; Gehring et al., 2017a); This raises the question of whether these successes of convolutional sequence modeling are confined to specific application domains or whether a broader reconsideration of the association between sequence processing and recurrent networks is in order.

We address this question by conducting a systematic empirical evaluation of convolutional and recurrent architecture

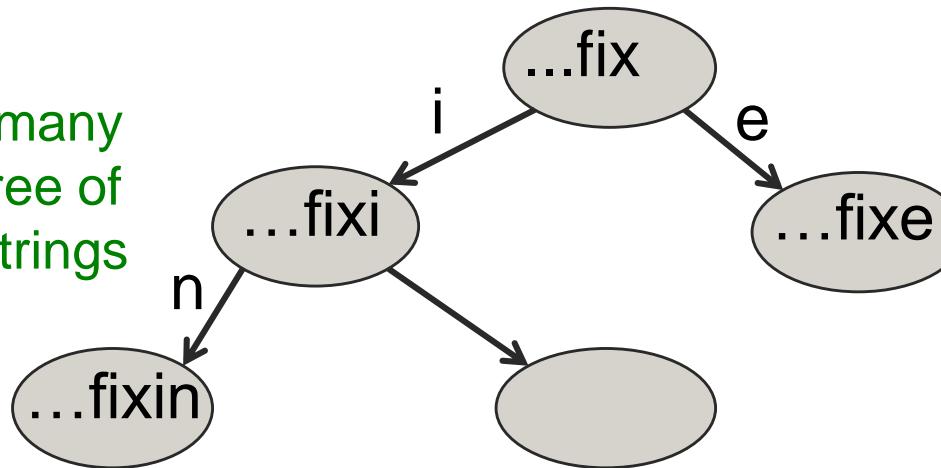
Example:  
Character-level  
Text Modeling

# Modeling text: Advantages of working with characters

- The web is composed of character strings.
- Any learning method powerful enough to understand the world by reading the web ought to find it trivial to learn which strings make words (**this turns out to be true, as we shall see**).
- Pre-processing text to get words is a big hassle
  - What about morphemes (**prefixes, suffixes** etc)
  - What about subtle effects like “sn” words?
  - What about New York?
  - What about Finnish
    - ymmartämättömyydellänsakaan

# A sub-tree in the tree of all character strings

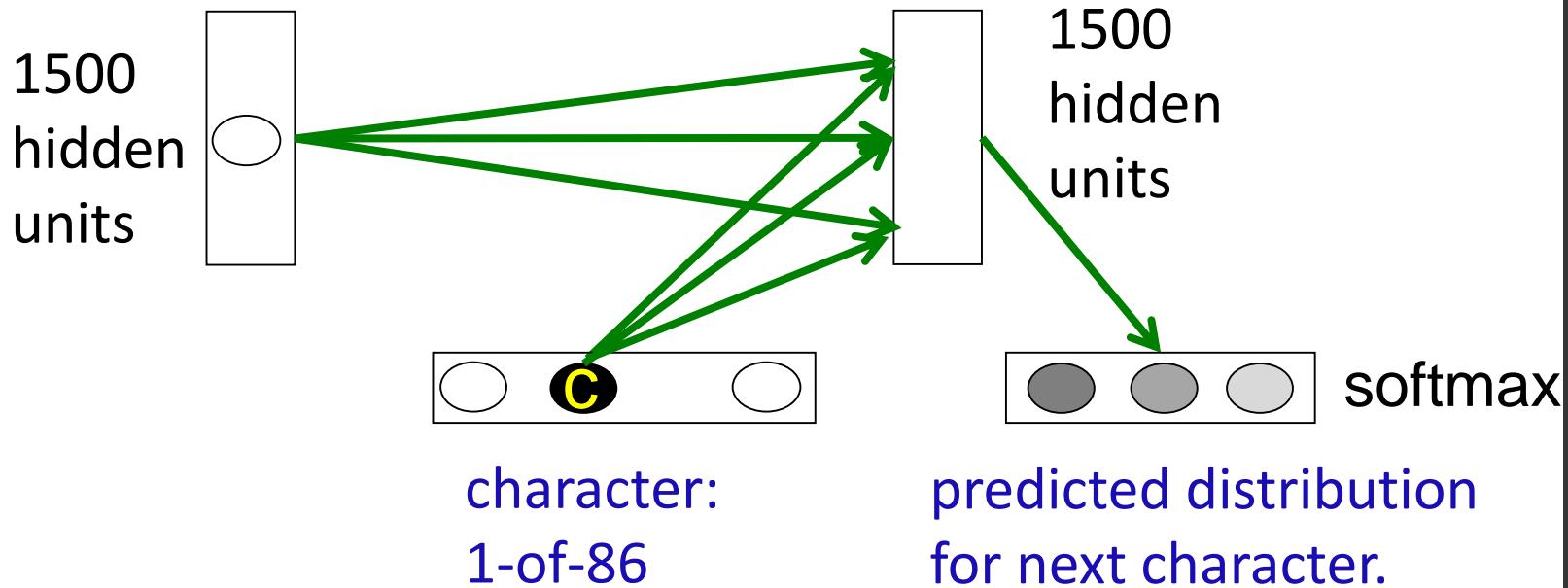
There are exponentially many nodes in the tree of all character strings of length N.



In an RNN, each node is a hidden state vector. The next character must transform this to a new node.

- If the nodes are implemented as hidden states in an RNN, different nodes can share structure because they use distributed representations.
- The next hidden representation needs to depend on the **conjunction** of the current character and the current hidden representation.

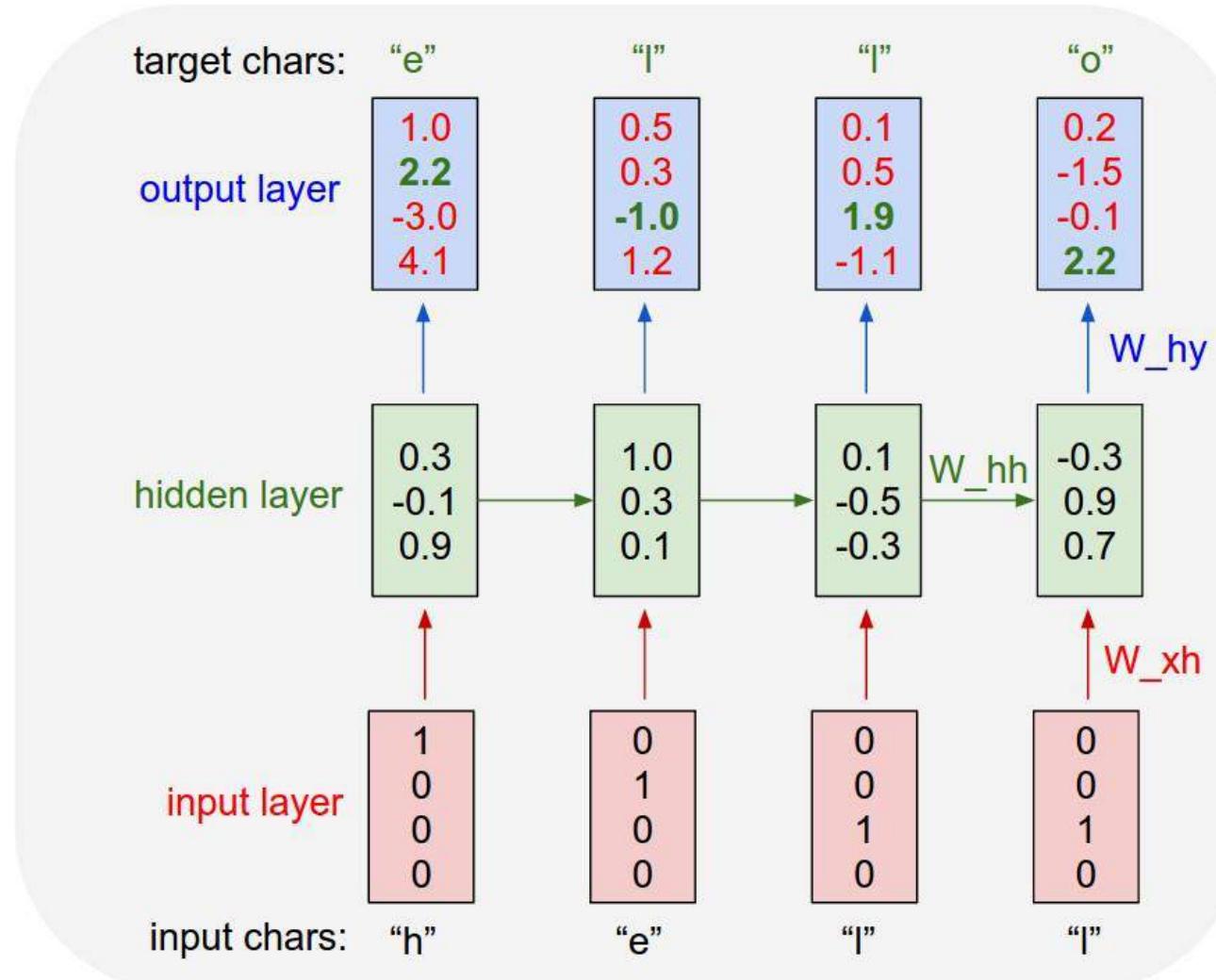
# An obvious recurrent neural net



It's a lot easier to predict 86 characters than 100,000 words.

# A simple scenario

- Alphabet: h, e, l, o
- Text to train to predict: “hello”



# Sample predictions (when trained on the works of **Shakespeare**):

- 3-level RNN  
with 512  
hidden nodes  
in each layer

PANDARUS:

Alas, I think he shall be come approached and the day  
When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,  
Breaking and strongly should be buried, when I perish  
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and  
my fair nues begun out of the fact, to be conveyed,  
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

# Sample predictions (when trained on Wikipedia):

- Using LSTM

Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict. Copyright was the succession of independence in the slop of Syrian influence that was a famous German movement based on a more popular servicious, non-doctrinal and sexual power post. Many governments recognize the military housing of the [[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]], that is sympathetic to be to the [[Punjab Resolution]] (PJS) [<http://www.humah.yahoo.com/guardian.cfm/7754800786d17551963s89.htm>] Official economics Adjoint for the Nazism, Montgomery was swear to advance to the resources for those Socialism's rule, was starting to signing a major tripad of aid exile.]]

# Sample predictions

(when trained on Latex documents):

- Using multi-layer LSTM

For  $\bigoplus_{n=1,\dots,m} \mathcal{L}_{m,n} = 0$ , hence we can find a closed subset  $H$  in  $\mathcal{H}$  and any sets  $F$  on  $X$ ,  $U$  is a closed immersion of  $S$ , then  $U \rightarrow T$  is a separated algebraic space.

*Proof.* Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by  $\coprod Z \times_U U \rightarrow V$ . Consider the maps  $M$  along the set of points  $\text{Sch}_{fppf}$  and  $U \rightarrow U$  is the fibre category of  $S$  in  $U$  in Section, ?? and the fact that any  $U$  affine, see Morphisms, Lemma ???. Hence we obtain a scheme  $S$  and any open subset  $W \subset U$  in  $\text{Sh}(G)$  such that  $\text{Spec}(R') \rightarrow S$  is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that  $f_i$  is of finite presentation over  $S$ . We claim that  $\mathcal{O}_{X,x}$  is a scheme where  $x, x', s'' \in S'$  such that  $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}'_{X',x'}$  is separated. By Algebra, Lemma ?? we can define a map of complexes  $\text{GL}_{S'}(x'/S'')$  and we win.  $\square$

To prove study we see that  $\mathcal{F}|_U$  is a covering of  $X'$ , and  $T_i$  is an object of  $\mathcal{F}_{X/S}$  for  $i > 0$  and  $\mathcal{F}_p$  exists and let  $\mathcal{F}_i$  be a presheaf of  $\mathcal{O}_X$ -modules on  $C$  as a  $\mathcal{F}$ -module. In particular  $\mathcal{F} = U/\mathcal{F}$  we have to show that

$$\widetilde{\mathcal{M}}^\bullet = \mathcal{I}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)_{fppf}^{\text{opp}}, (\text{Sch}/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longrightarrow (U, \text{Spec}(A))$$

is an open subset of  $X$ . Thus  $U$  is affine. This is a continuous map of  $X$  is the inverse, the groupoid scheme  $S$ .

*Proof.* See discussion of sheaves of sets.  $\square$

The result for prove any open covering follows from the less of Example ???. It may replace  $S$  by  $X_{\text{spaces},\text{étale}}$  which gives an open subspace of  $X$  and  $T$  equal to  $S_{\text{Zar}}$ , see Descent, Lemma ???. Namely, by Lemma ?? we see that  $R$  is geometrically regular over  $S$ .

**Lemma 0.1.** Assume (3) and (3) by the construction in the description.

Suppose  $X = \lim |X|$  (by the formal open covering  $X$  and a single map  $\underline{\text{Proj}}_X(\mathcal{A}) = \text{Spec}(B)$  over  $U$  compatible with the complex

$$\text{Set}(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that  $\mathcal{Q} \rightarrow \mathcal{C}_{Z/X}$  is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If  $T$  is surjective we may assume that  $T$  is connected with residue fields of  $S$ . Moreover there exists a closed subspace  $Z \subset X$  of  $X$  where  $U$  in  $X'$  is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1)  $f$  is locally of finite type. Since  $S = \text{Spec}(R)$  and  $Y = \text{Spec}(R)$ .

*Proof.* This is form all sheaves of sheaves on  $X$ . But given a scheme  $U$  and a surjective étale morphism  $U \rightarrow X$ . Let  $U \cap U = \coprod_{i=1,\dots,n} U_i$  be the scheme  $X$  over  $S$  at the schemes  $X_i \rightarrow X$  and  $U = \lim_i X_i$ .  $\square$

The following lemma surjective restrocomposes of this implies that  $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{x,\dots,x}$ .

**Lemma 0.2.** Let  $X$  be a locally Noetherian scheme over  $S$ ,  $E = \mathcal{F}_{X/S}$ . Set  $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$ . Since  $\mathcal{I}^n \subset \mathcal{I}^n$  are nonzero over  $i_0 \leq p$  is a subset of  $\mathcal{J}_{n,0} \circ A_2$  works.

**Lemma 0.3.** In Situation ???. Hence we may assume  $q' = 0$ .

*Proof.* We will use the property we see that  $p$  is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where  $K$  is an  $F$ -algebra where  $\delta_{n+1}$  is a scheme over  $S$ .  $\square$

From Ilya Sutskever (using a variant of character-level RNN)

He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters' sisters in lower coil trains were always operated on the line of the **ephemeral** street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS). The B every chord was a "strongly cold internal palette pour even the white blade."

# Some completions produced by the model

- Sheila thrunge<sup>s</sup> (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6<sup>th</sup> try)
- The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. (one of the first 10 tries for a model trained for longer).

# What does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers.
- It is good at balancing quotes and brackets.
  - It can count brackets: **none**, **one**, **many**
- It knows a lot about syntax but its very hard to pin down exactly what form this knowledge has.
  - Its syntactic knowledge is not modular.
- It knows a lot of weak semantic associations
  - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable.

# RNNs for predicting the next word

- Tomas Mikolov and his collaborators have trained quite large RNNs on quite large training sets using BPTT.
  - They do better than feed-forward neural nets.
  - They do better than the best other models.
  - They do even better when averaged with other models.
- RNNs require much less training data to reach the same level of performance as other models.
- RNNs improve faster than other methods as the dataset gets bigger.
  - This is going to make them very hard to beat.

# Word-level RNN for news title generation

<https://github.com/larspars/word-rnn>

## Click-o-Tron



[3D Video Brings Clean Energy To The Real Economy](#)



[David Beckham & Victoria Beckham's Talk Show Gets 2014 Golden Girls Love](#)

The area is still in such big power, but I have some advice for those this ye...



[White House Attacks The Obama Plan](#)

Women learned from their families in a post-9/11 New York Times story that is...



[Not The Same Thing: New Rule Can't](#)



### LATEST STORIES

[Family Of Batman Killed Up To Time In Jail Was Worst Political Idea Ever](#)



[How To Create Your Own Hair In Less Than 3 Minutes](#)



['Ways Kids': Happy Mother's Day](#)

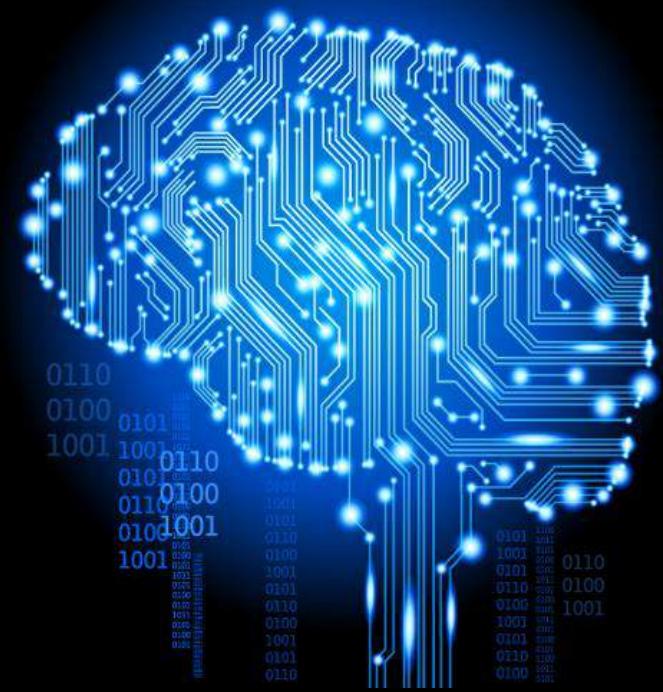
<http://clickotron.com/>

# CENG 783

# Deep Learning

*Week – 10  
Recurrent Neural Networks  
(contd.)*

Sinan Kalkan



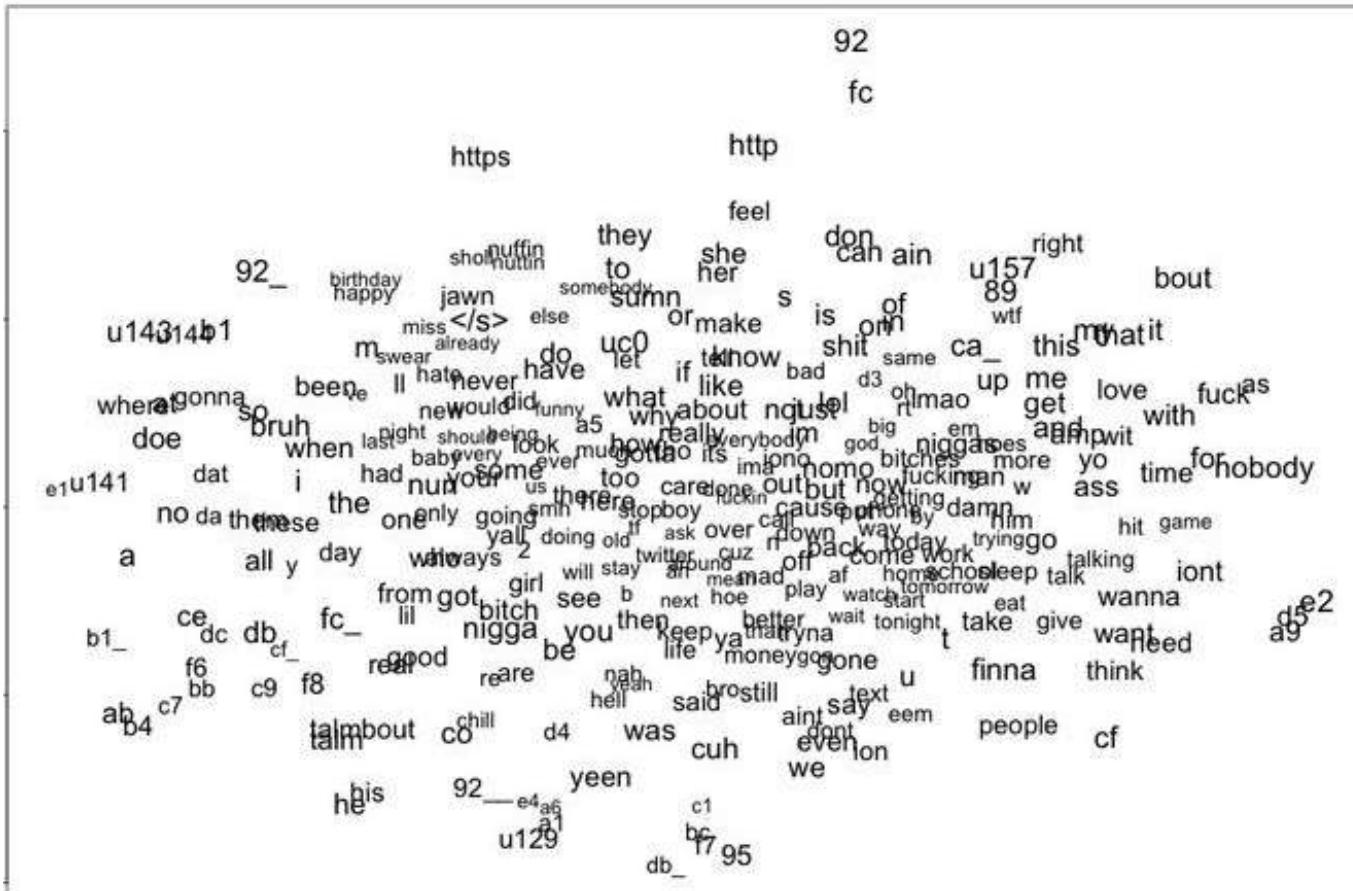
© AlchemyAPI



# Today

- Finalize RNNs
  - Word-level text modelling
  - Image captioning
  - Neural machine translation
  - Echo state networks
  - Time-delay neural networks
- Neural Turing Machines
- NOTES:
  - Final Exam date: 31 May, 17:00.
  - HW3 to be announced next week.
  - Project demos and papers due: 6 June.

## A two dimensional reduction of the vector space model using t-SNE



# Word Embedding (word2vec)

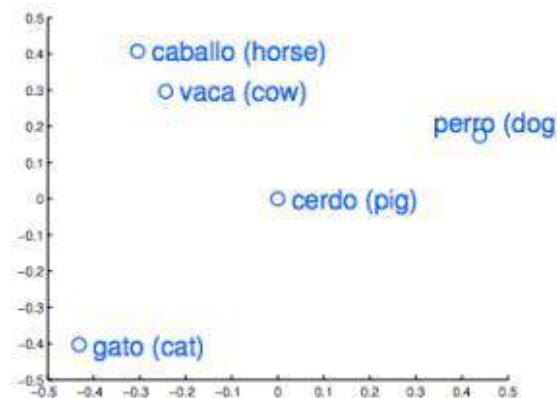
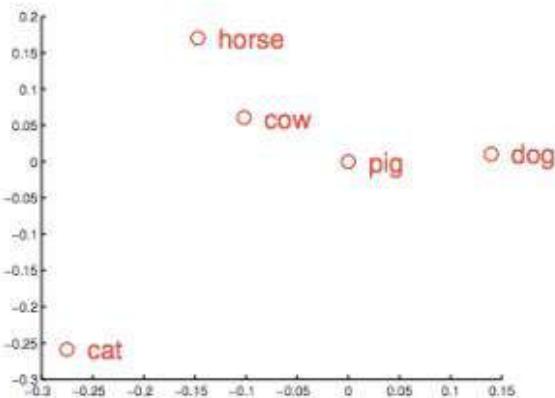
# Why do we embed words?

- 1-of-n encoding is not suitable to learn from
  - It is sparse
  - Similar words have different representations
  - Compare the pixel-based representation of images: Similar images/objects have similar pixels
- Embedding words in a map allows
  - Encoding them with fixed-length vectors
  - “Similar” words having similar representations
  - Allows complex reasoning between words:
    - king - man + woman = queen

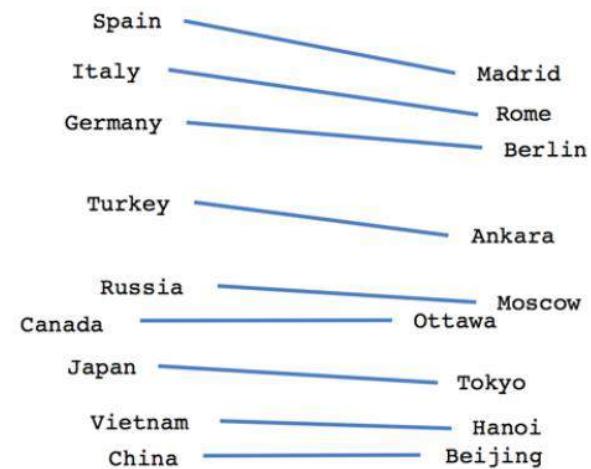
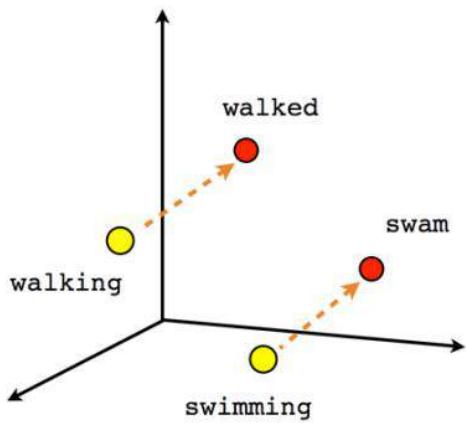
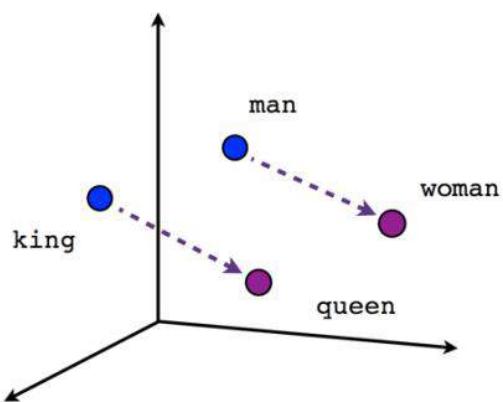
EXPRESSION	NEAREST TOKEN
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Monreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

Table 1: Mikolov et al. [3] showcase simple additive properties of their word embeddings.

# More examples



# More examples

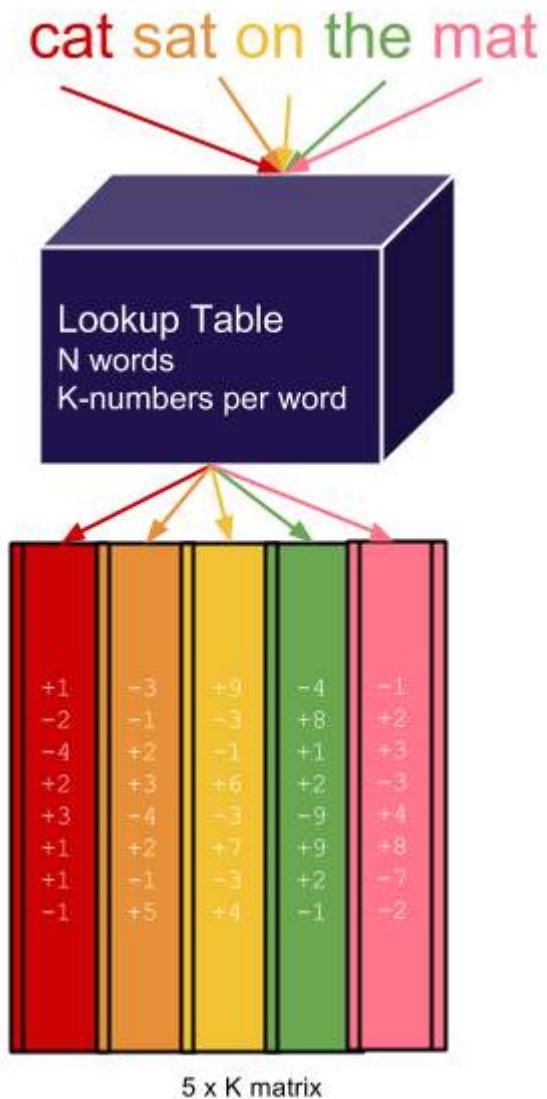


# More examples

- Geopolitics: *Iraq - Violence = Jordan*
- Distinction: *Human - Animal = Ethics*
- *President - Power = Prime Minister*
- *Library - Books = Hall*

# Using word embeddings

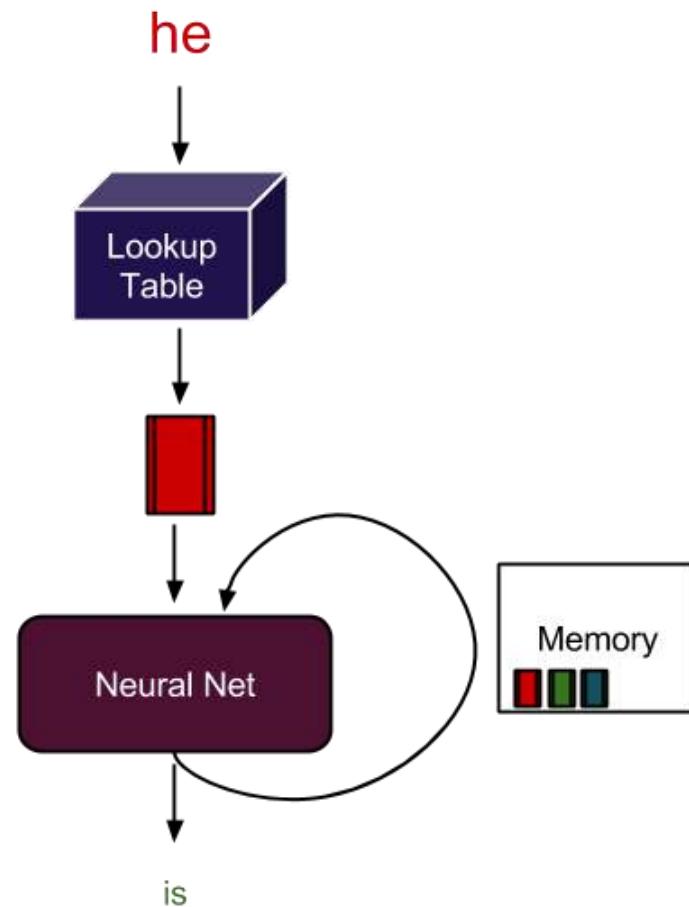
- E.g., for language modeling
- Given “I am eating ...”, a language model can predict what can come next.
  - This both requires syntax and semantics (context)
- Before deep learning, n-gram (2-gram, 3-gram) models were state of the art.



# Using word embeddings



# Using word embeddings



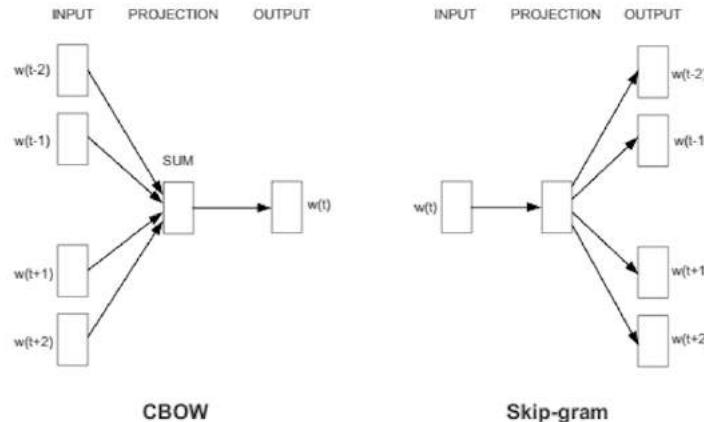
# word2vec

- “Similarity” to Sweden (cosine distance between their vector representations)

Word	Cosine distance
<hr/>	
norway	0.760124
denmark	0.715460
finland	0.620022
switzerland	0.588132
belgium	0.585835
netherlands	0.574631
iceland	0.562368
estonia	0.547621
slovenia	0.531408

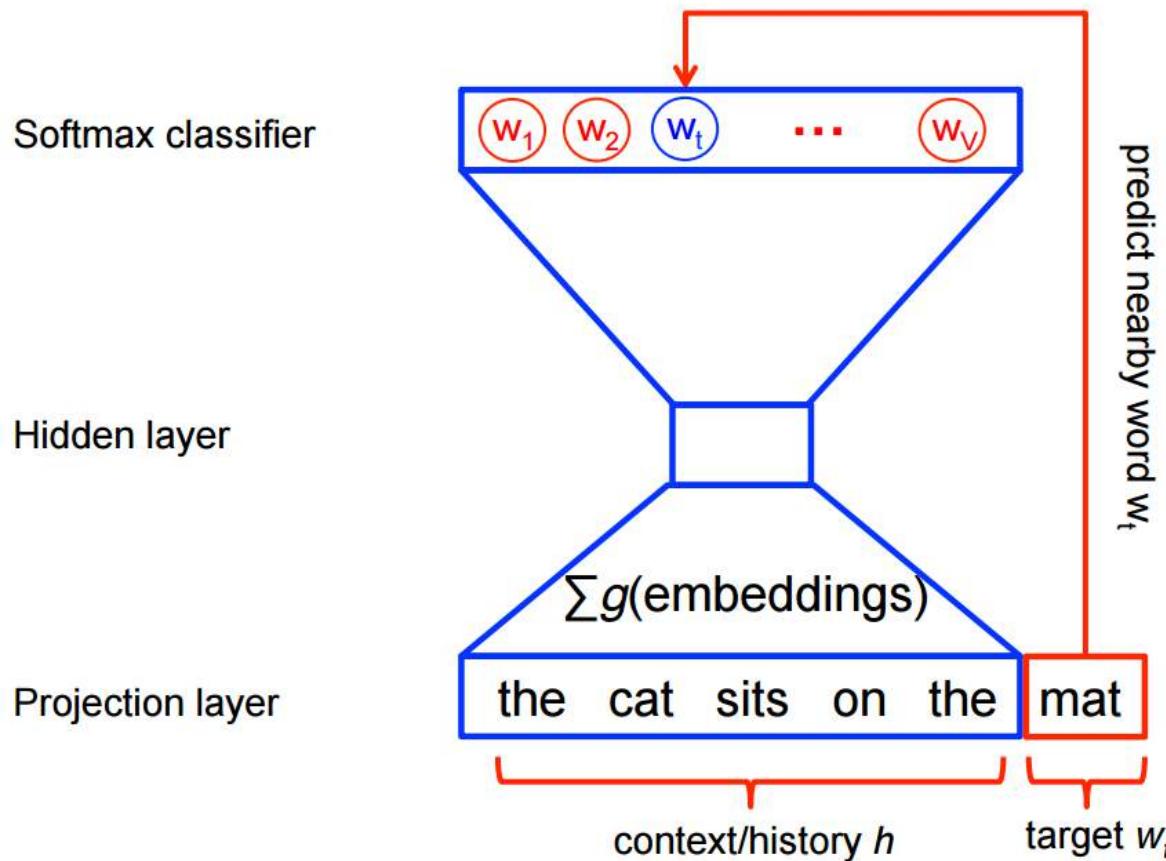
# Two different ways to train

1. Using context to predict a target word (~ continuous bag-of-words)
2. Using word to predict a target context (skip-gram)
  - Produces more accurate results on large datasets
  - If the vector for a word cannot predict the context, the mapping to the vector space is adjusted
  - Since similar words should predict the same or similar contexts, their vector representations should end up being similar



# Two different ways to train

1. Using context to predict a target word (~ continuous bag-of-words)



# Two different ways to train

## 2. Using word to predict a target context (skip-gram)

- Produces more accurate results on large datasets

- Given a sentence:

the quick brown fox jumped over the lazy dog

- For each word, take context to be

(N-words to the left, N-words to the right)

- If  $N = 1$  (context, word):

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...

# Some details

- CBOW is called continuous BOW since the context is regarded as a BOW and it is continuous.
- In both approaches, the networks are composed of linear units
- The output units are usually normalized with the softmax
- According to Mikolov:
  - “*Skip-gram: works well with small amount of the training data, represents well even rare words or phrases.*
  - *CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words”*

# An extra issue: Sampling

- Greedy sampling: Take the most likely word at each step

vs

- Beam search (or alternatives): Consider  $k$  most likely words at each step, and expand search.

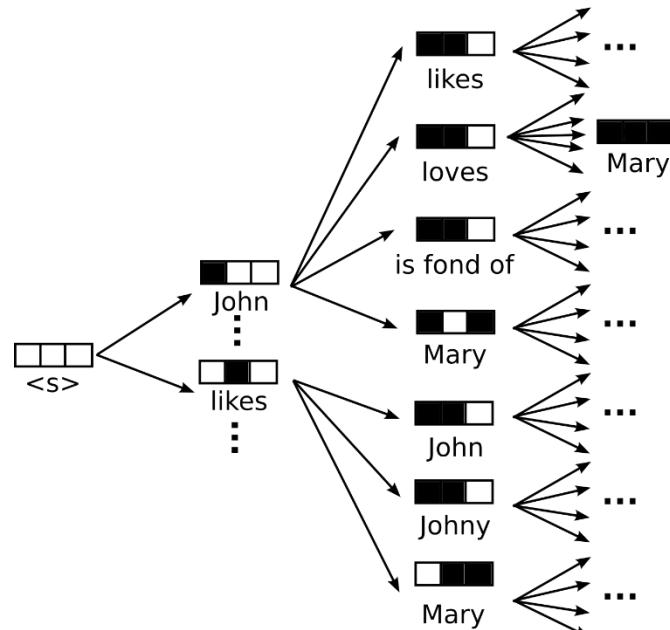


Figure: <http://mttalks.ufal.ms.mff.cuni.cz/index.php>

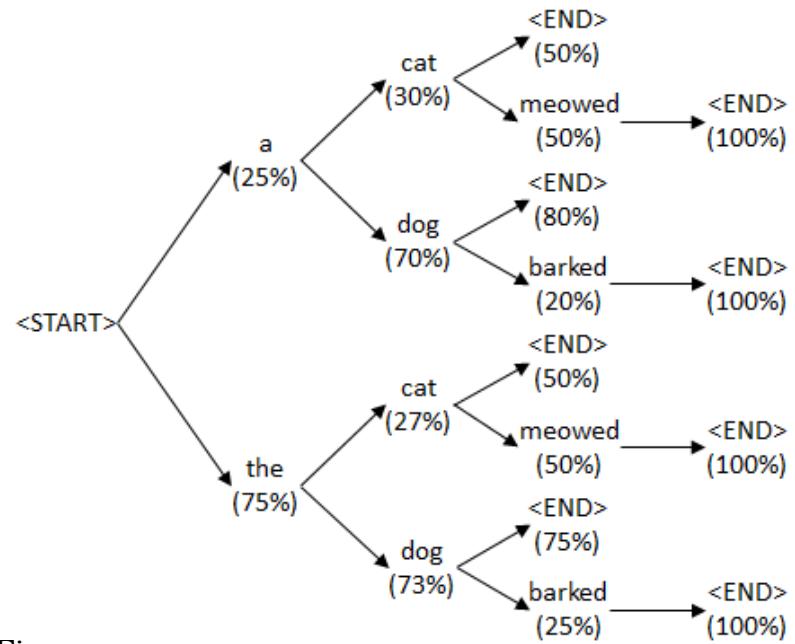


Figure:  
<https://geekyisawesome.blogspot.com.tr/2016/10/using-beam-search-to-generate-most.html>



a man is playing tennis on a tennis court



a train is traveling down the tracks at a train station



a cake with a slice cut out of it



a bench sitting on a patch of grass next to a sidewalk

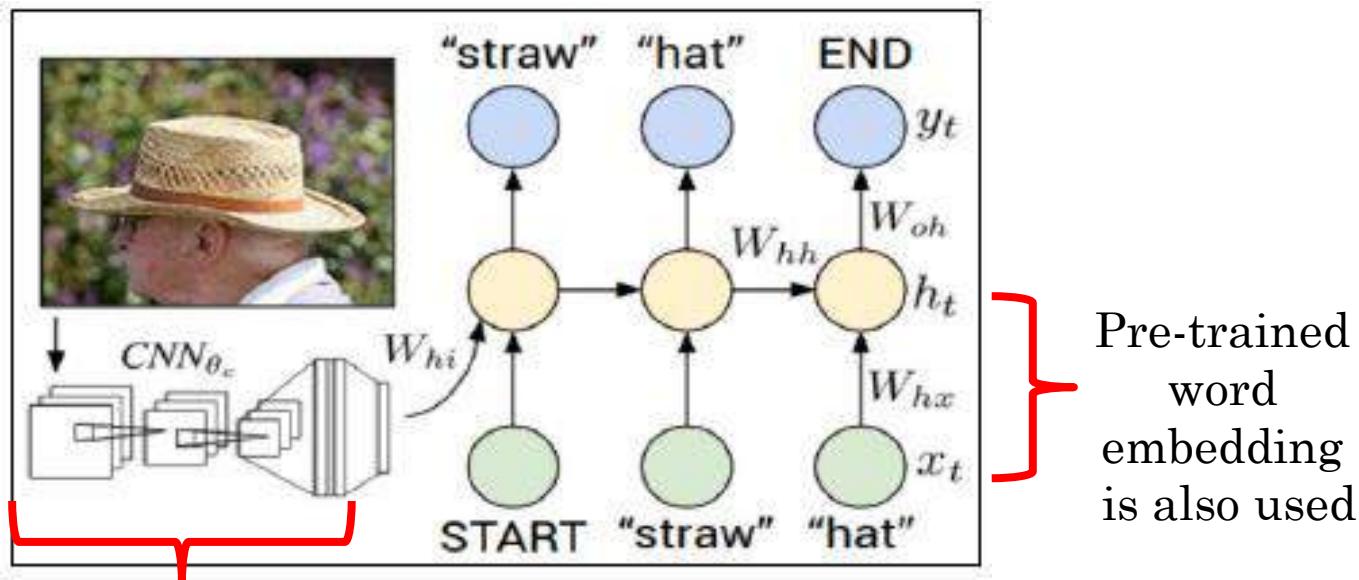
Fig: <https://github.com/karpathy/neuraltalk2>

# Example: Image Captioning

# Demo video

<https://vimeo.com/146492001>

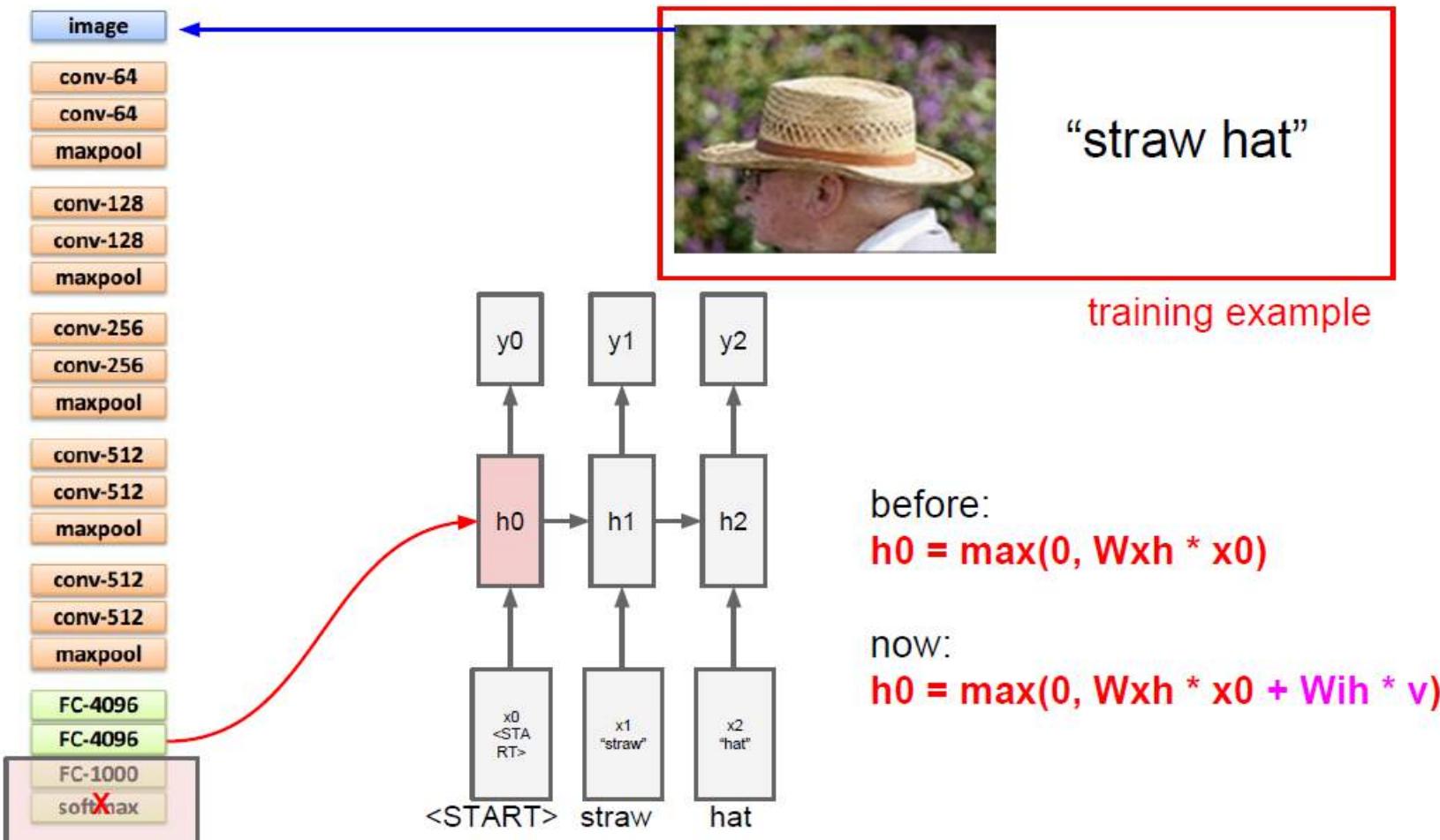
# Overview



Pre-trained CNN  
(e.g., on imagenet)

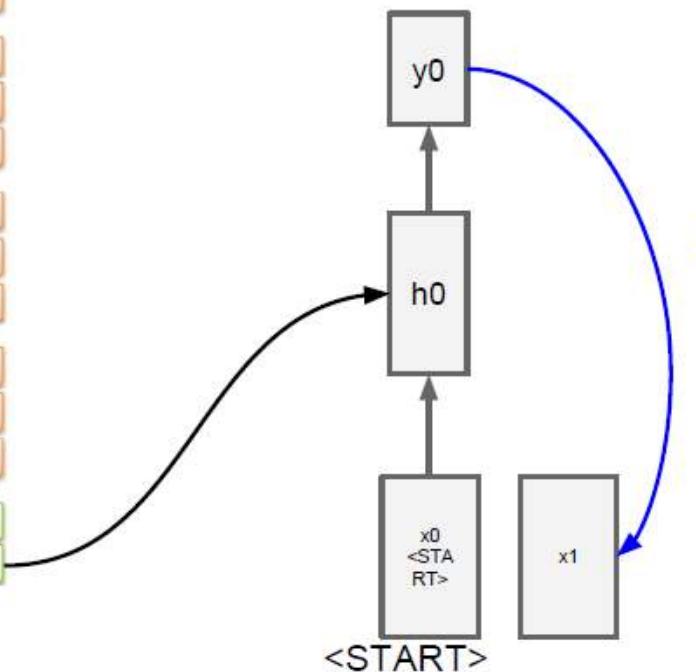
Image: Karpathy

# Training



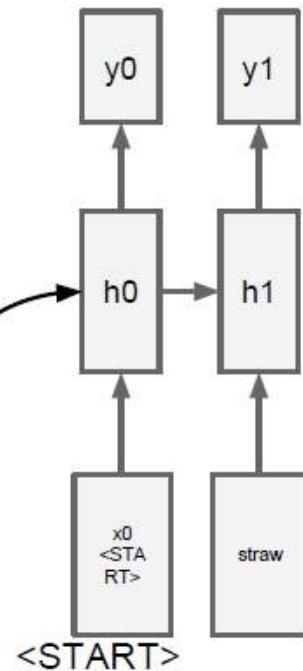


test image



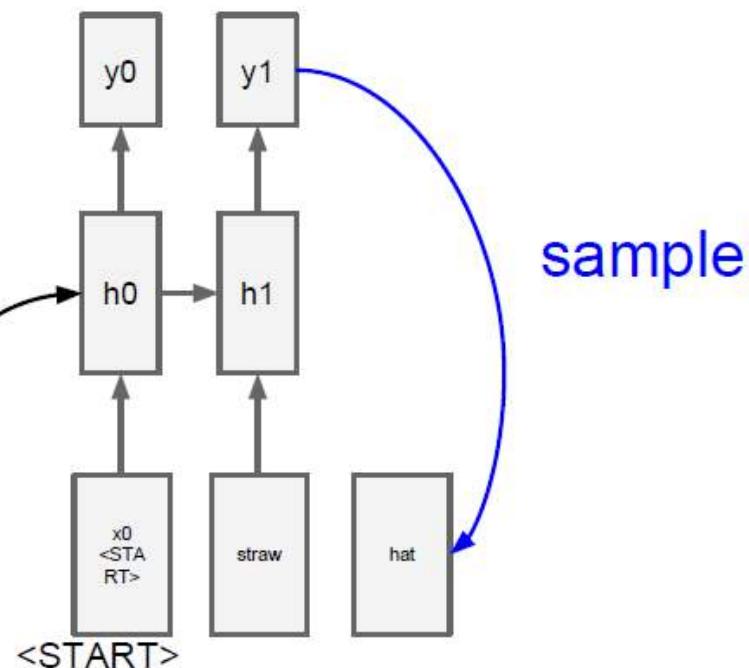


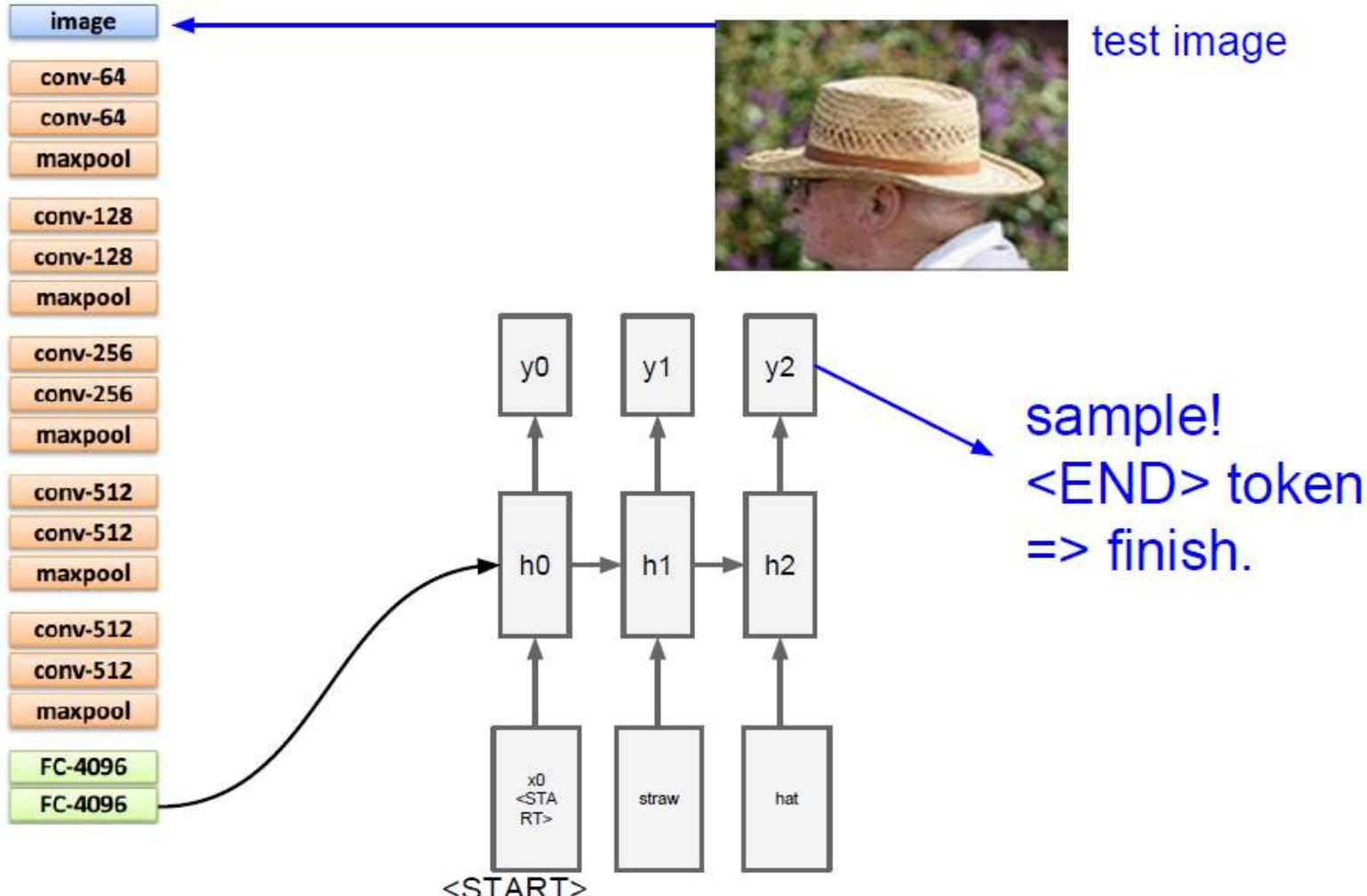
test image





test image





**Learning Phrase Representations using RNN Encoder–Decoder  
for Statistical Machine Translation**

**Kyunghyun Cho**

Bart van Merriënboer Caglar Gulcehre

Université de Montréal

firstname.lastname@umontreal.ca

Dzmitry Bahdanau

Jacobs University, Germany

d.bahdanau@jacobs-university.de

**Fethi Bougares Holger Schwenk**

Université du Maine, France

Université de Montréal, CIFAR Senior Fellow

firstname.lastname@lium.univ-lemans.fr

**Yoshua Bengio**

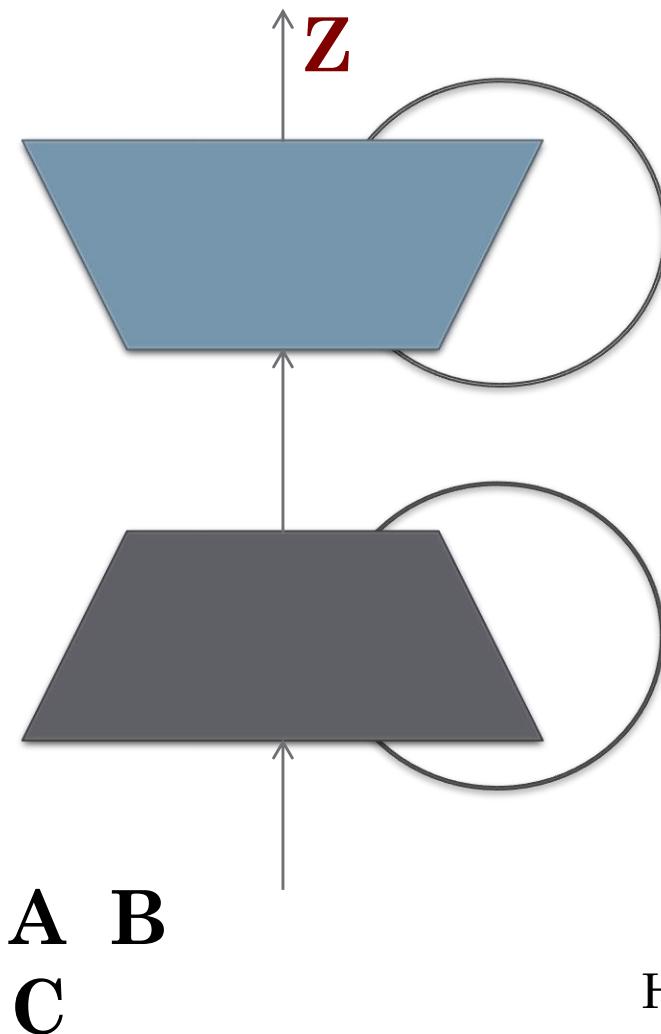
find.me@on.the.web

# Example: Neural Machine Translation

# *Neural Machine Translation*

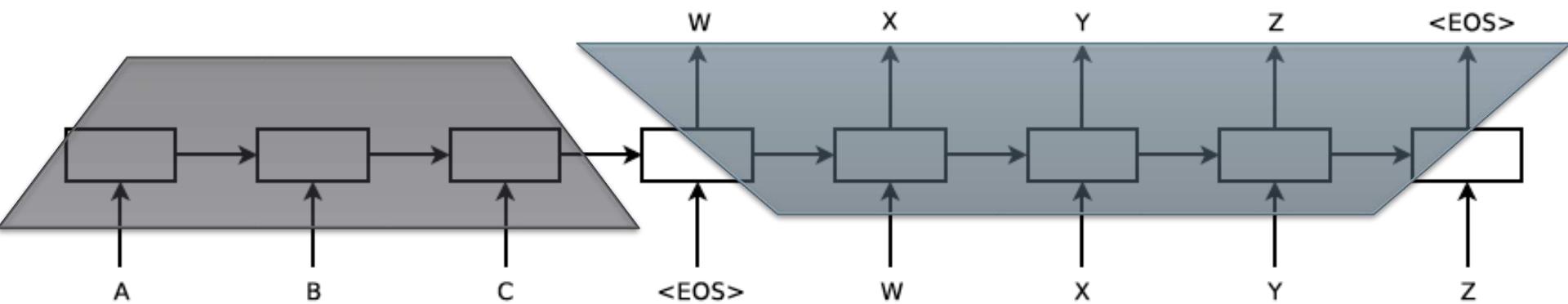
W X Y

- Model



# *Neural Machine Translation*

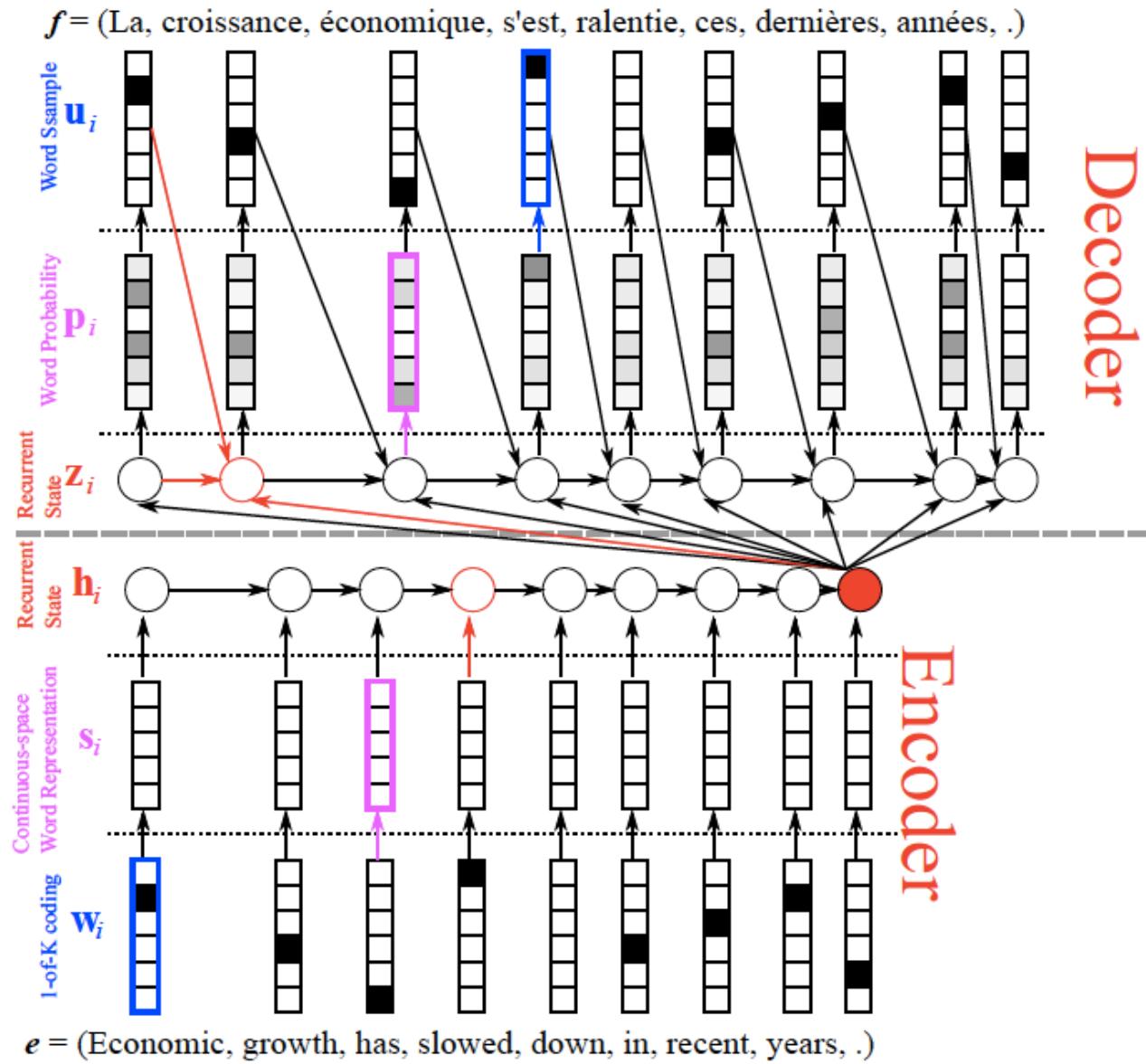
- Model



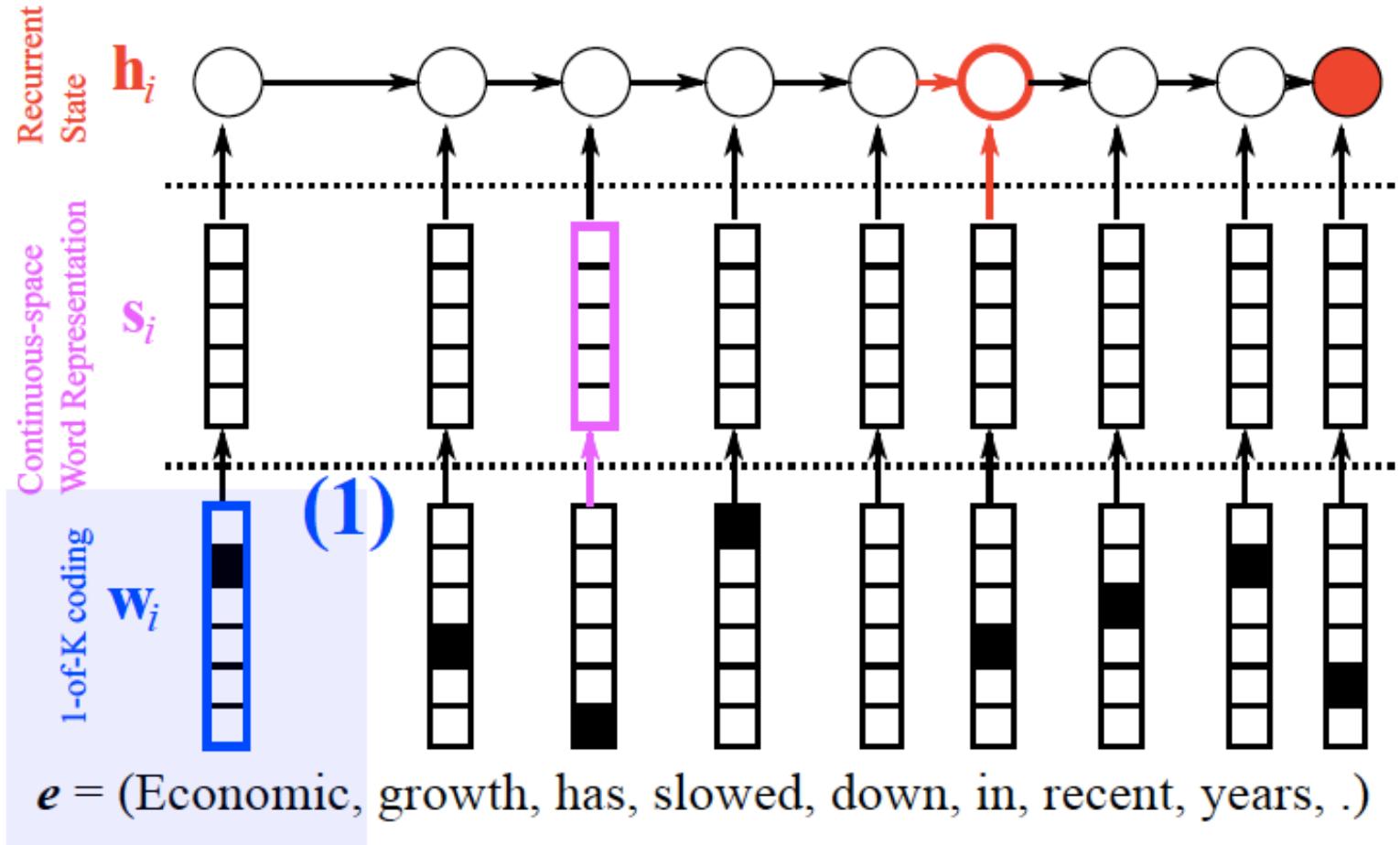
Each box is an LSTM or GRU cell.

# Neural Machine Translation

- Model- *encoder*

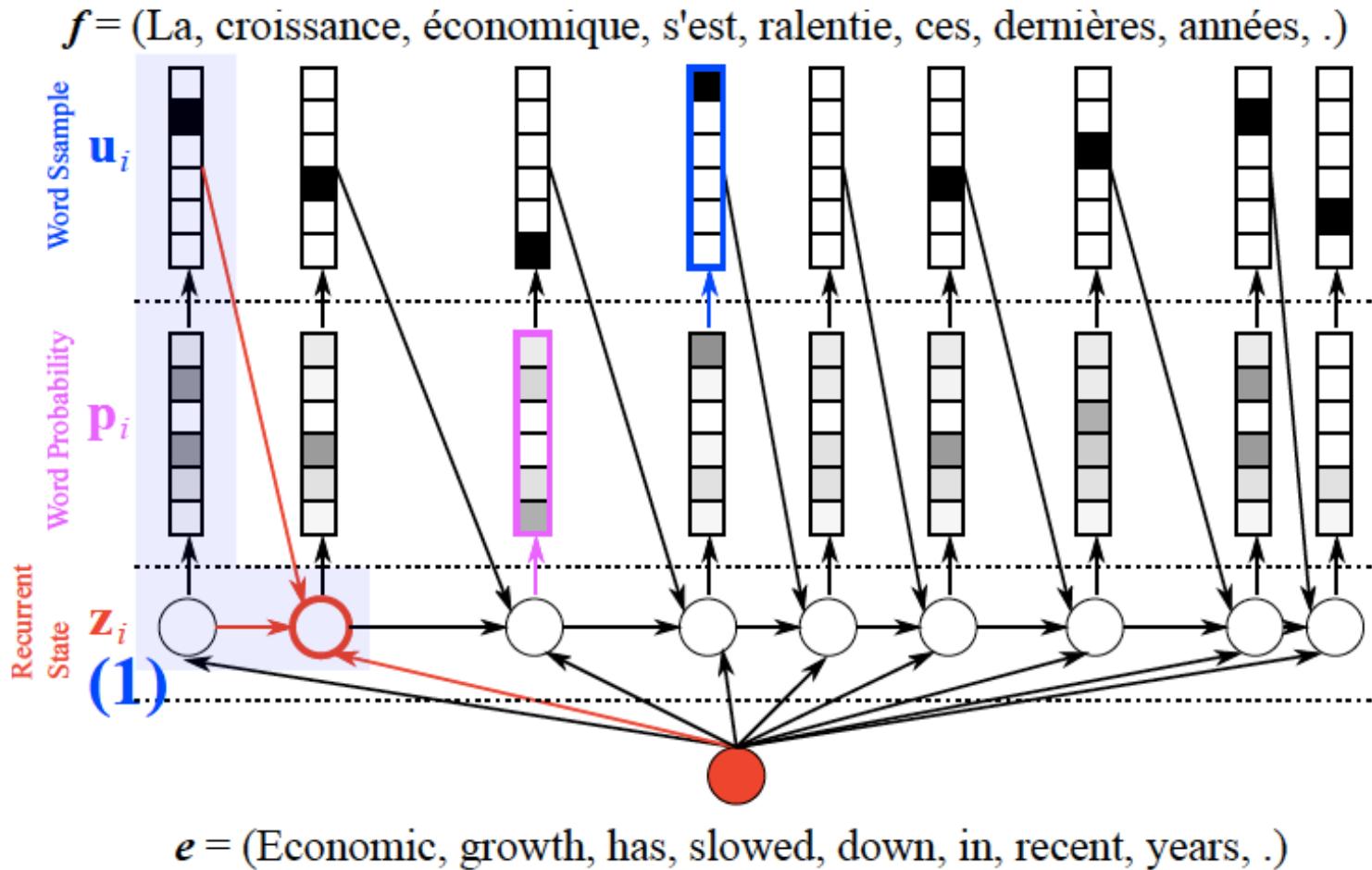


# Neural Machine Translation



# Neural Machine Translation

- Model-decoder



# Decoder in more detail

Given

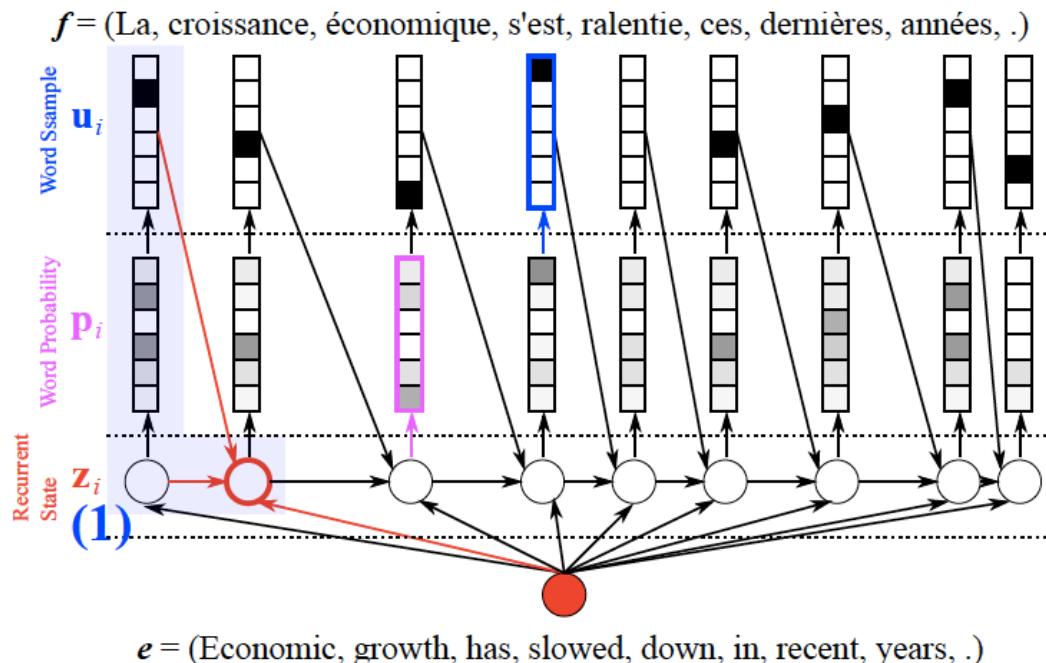
- (i) the “summary” ( $\mathbf{c}$ ) of the input sequence,
- (ii) the previous output / word ( $y(t - 1)$ )
- (iii) the previous state ( $h(t - 1)$ )

the hidden state of the decoder is:

$$h(t) = f(h(t - 1), y(t - 1), \mathbf{c})$$

Then, we can find the most likely next word:

$$P(y(t) | y(t - 1), y(t - 2), \dots, \mathbf{c}) = g(h(t), y(t - 1), \mathbf{c})$$

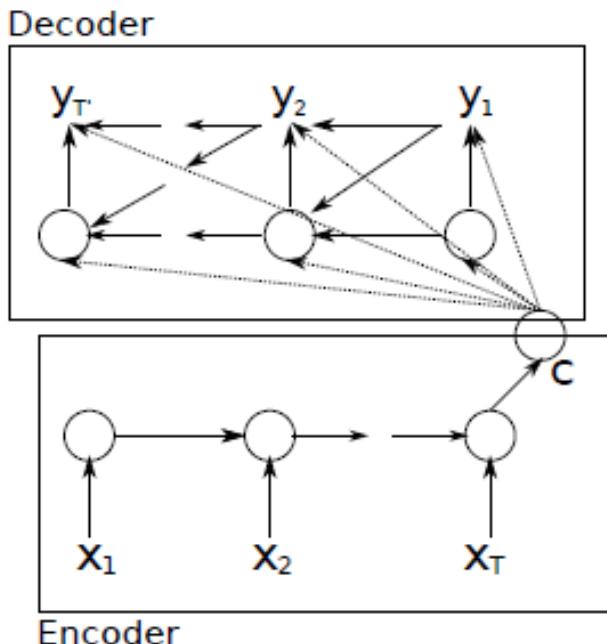


$f, g$ : activation functions of our choice. For  $g$ , we need a function that maps to probabilities, e.g., softmax.

# Encoder-decoder

- Jointly trained to maximize

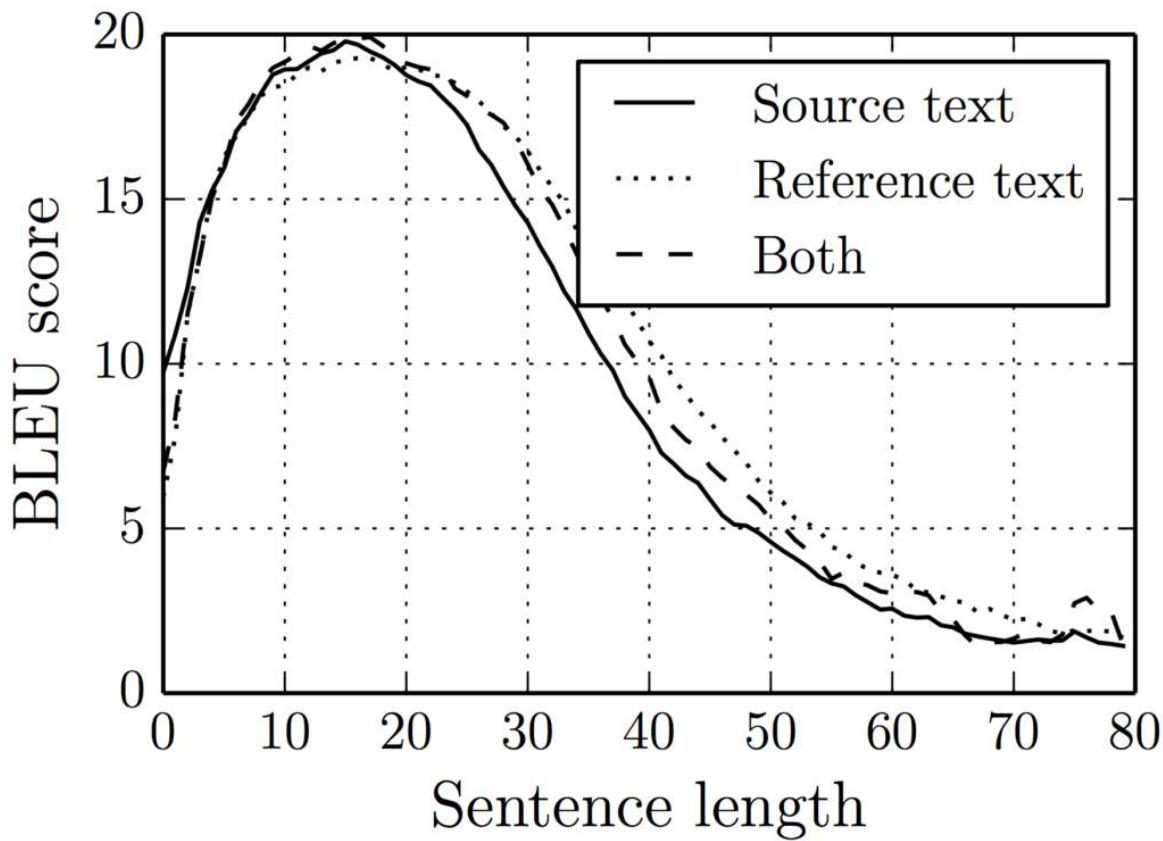
$$\max_{\theta} \frac{1}{N} \sum_{n=1}^N \log p_{\theta}(\mathbf{y}_n \mid \mathbf{x}_n),$$



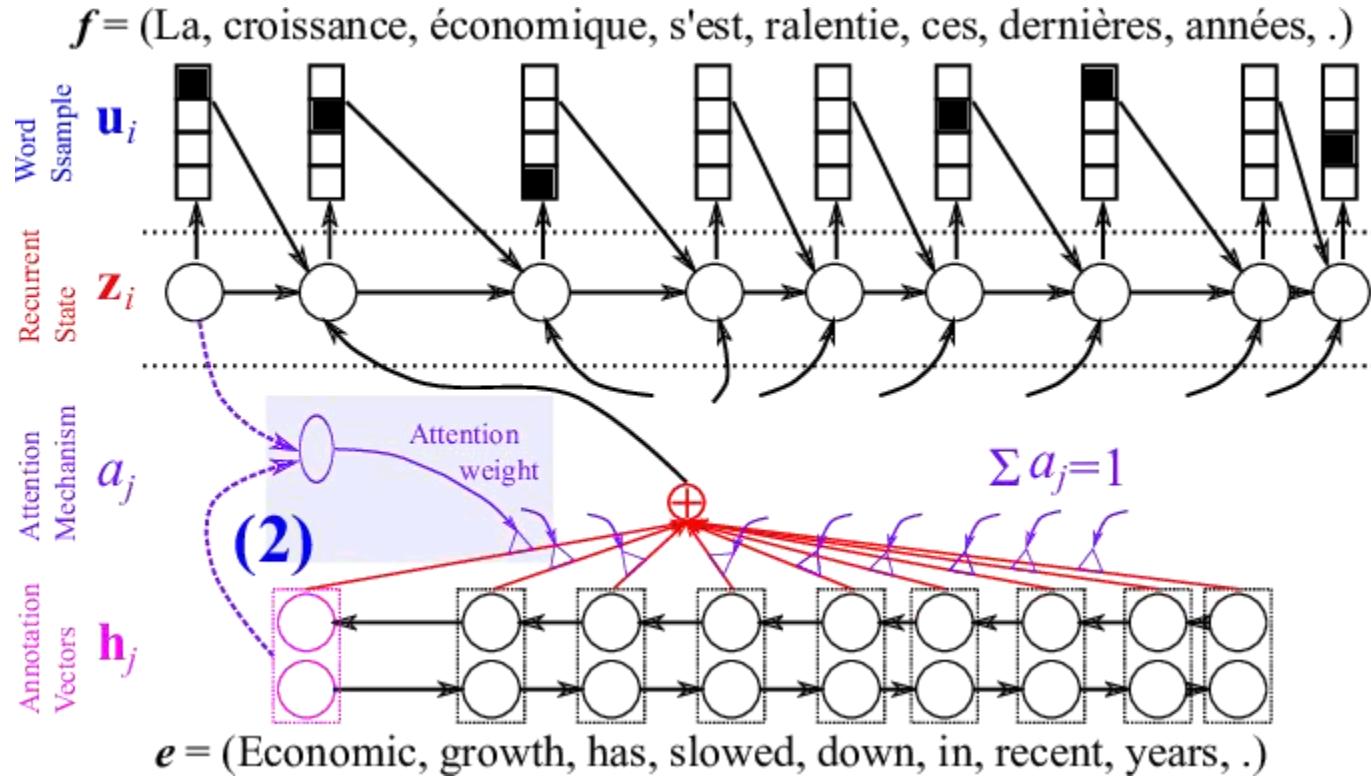
# Check the following tutorial

- [http://smerity.com/articles/2016/google\\_nmt\\_arch.html](http://smerity.com/articles/2016/google_nmt_arch.html)

# More on attention



# More on attention



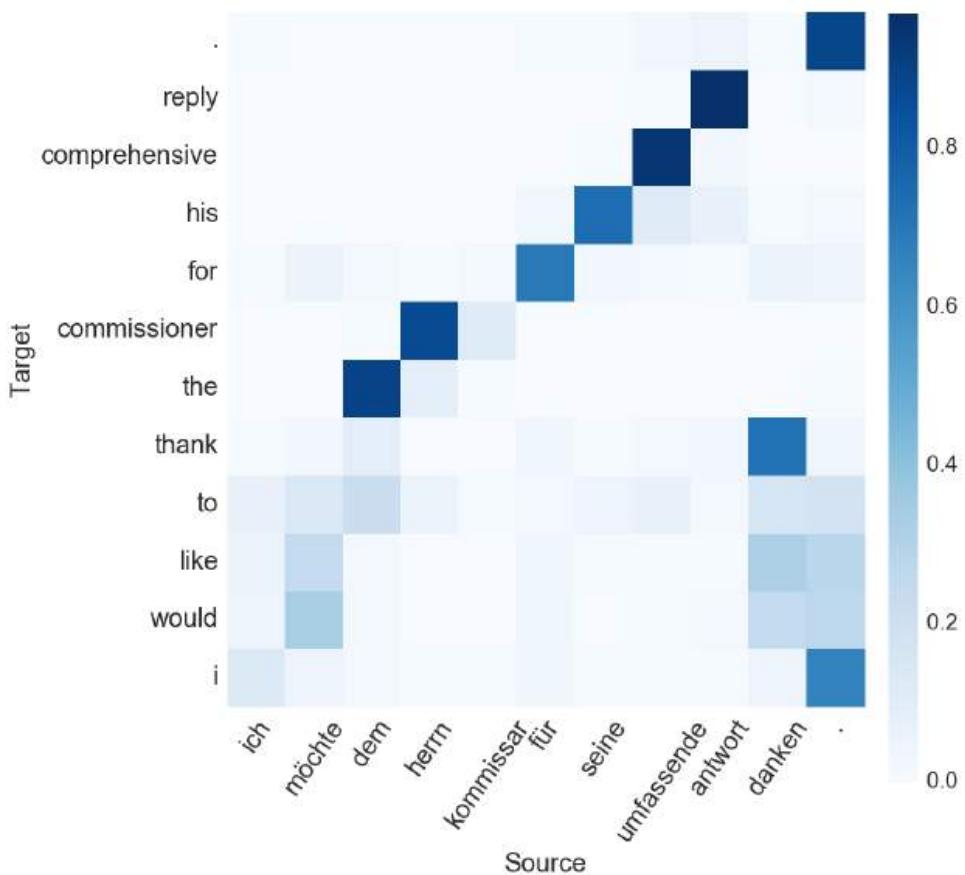
Attention mechanism: A two-layer neural network.

Input:  $z_i$  and  $h_j$

Output:  $e_j$ , a scalar for the importance of word  $j$ .

The scores of words are normalized:  $a_j = \text{softmax}(e_j)$

# More on attention



What does Attention in Neural Machine Translation  
Pay Attention to?

Hamidreza Ghader and Christof Monz  
Informatics Institute, University of Amsterdam, The Netherlands  
h.ghader, c.monz@uva.nl

2017

# NMT can be done at char-level too

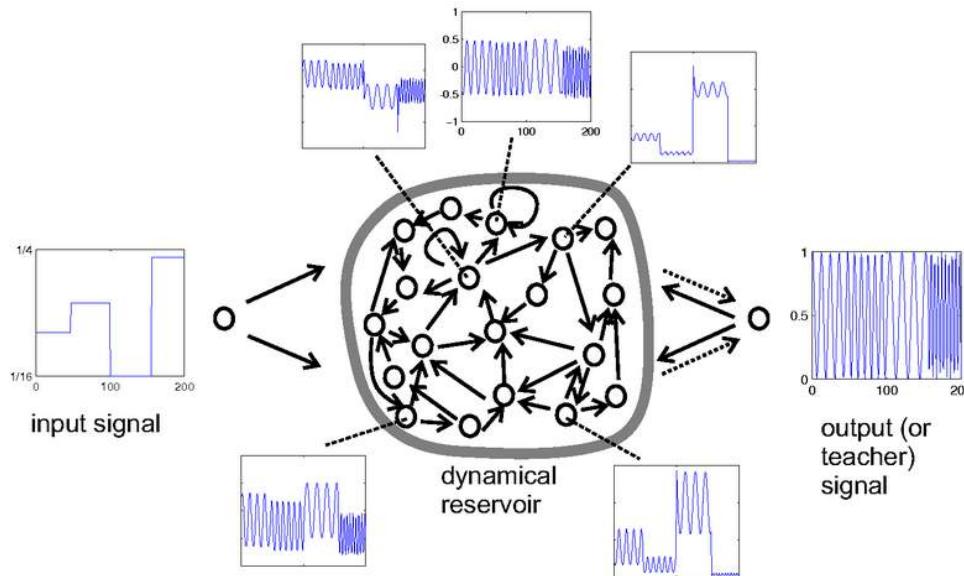
- <http://arxiv.org/abs/1603.06147>

# This can be done with CNNs

Jonas Gehring  
Michael Auli  
David Grangier  
Denis Yarats  
Yann N. Dauphin  
Facebook AI Research

2017

. la maison de Léa <end> .



# Echo State Networks

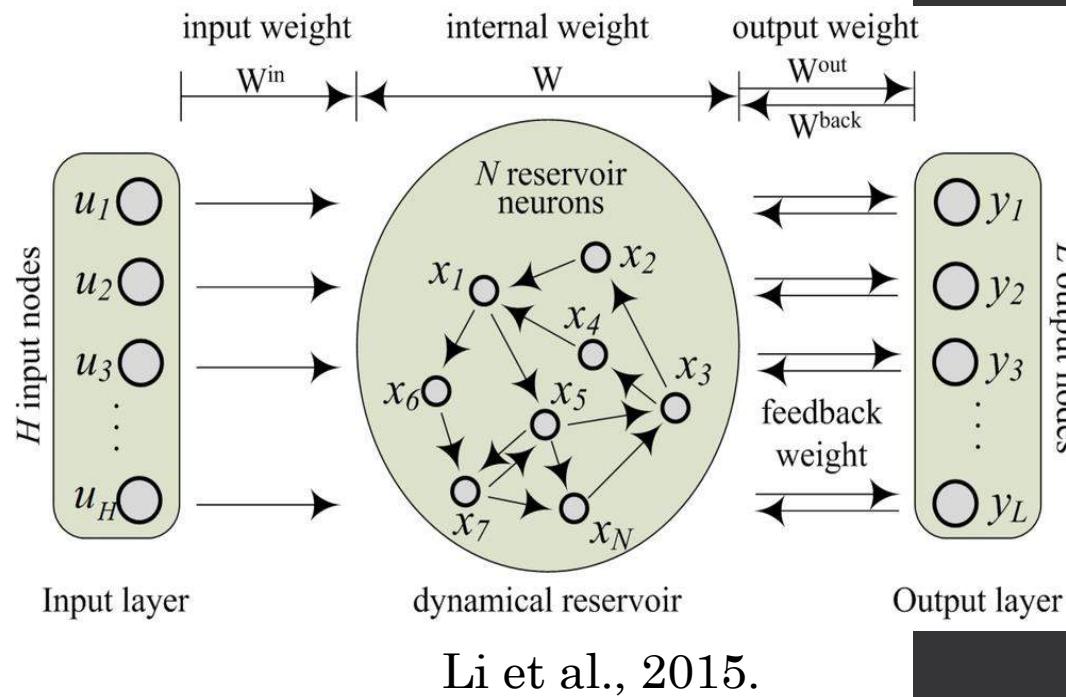
Reservoir Computing

# Motivation

- “Schiller and Steil (2005) also showed that in traditional training methods for RNNs, where all weights (not only the output weights) are adapted, **the dominant changes are in the output weights**. In cognitive [neuroscience](#), a related mechanism has been investigated by Peter F. Dominey in the context of modelling sequence processing in mammalian [brains](#), especially speech recognition in humans (e.g., Dominey 1995, Dominey, Hoen and Inui 2006). Dominey was the first to explicitly state the principle of reading out target information from a randomly connected RNN. The basic idea also informed a model of temporal input discrimination in biological neural networks (Buonomano and Merzenich 1995).”

# Echo State Networks (ESN)

- Reservoir of a set of neurons
  - Randomly initialized and fixed
  - Run input sequence through the network and keep the activations of the reservoir neurons
  - Calculate the “readout” weights using linear regression.
- Has the benefits of recurrent connections/networks
- No problem of vanishing gradient



# The reservoir

- Provides non-linear expansion
  - This provides a “kernel” trick.
- Acts as a memory
- Parameters:
  - $W_{in}$ ,  $W$  and  $\alpha$  (leaking rate).
- Global parameters:
  - Number of neurons: The more the better.
  - Sparsity: Connect a neuron to a fixed but small number of neurons.
  - Distribution of the non-zero elements: Uniform or Gaussian distribution.  $W_{in}$  is denser than  $W$ .
  - Spectral radius of  $W$ : Maximum absolute eigenvalue of  $W$ , or the width of the distribution of its non-zero elements.
    - Should be less than 1. Otherwise, chaotic, periodic or multiple fixed-point behavior may be observed.
    - For problems with large memory requirements, it should be bigger than 1.
  - Scale of the input weights.

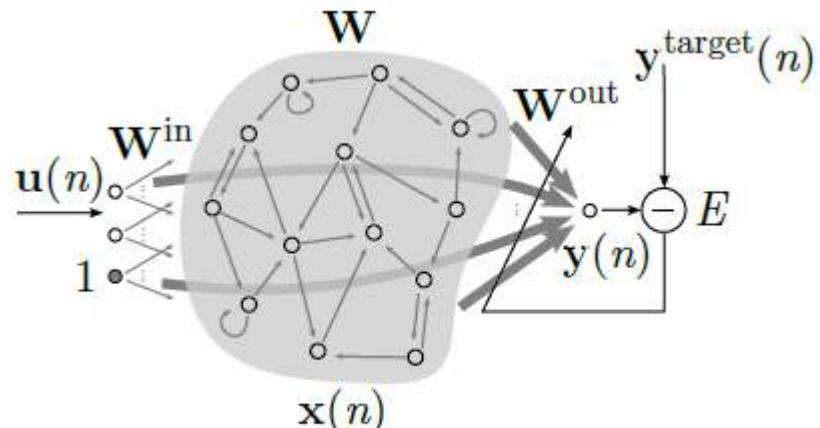


Fig. 1: An echo state network.

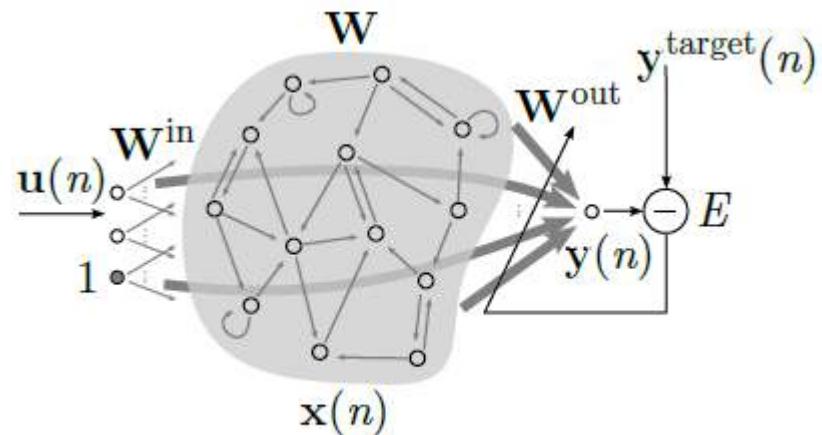
# A Practical Guide to Applying Echo State Networks

Mantas Lukoševičius

$$\tilde{\mathbf{x}}(n) = \tanh(\mathbf{W}^{\text{in}}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1)), \quad (2)$$

$$\mathbf{x}(n) = (1 - \alpha)\mathbf{x}(n-1) + \alpha\tilde{\mathbf{x}}(n), \quad (3)$$

where  $\mathbf{x}(n) \in \mathbb{R}^{N_x}$  is a vector of reservoir neuron activations and  $\tilde{\mathbf{x}}(n) \in \mathbb{R}^{N_x}$  is its update, all at time step  $n$ ,  $\tanh(\cdot)$  is applied element-wise,  $[;\cdot]$  stands for a vertical vector (or matrix) concatenation,  $\mathbf{W}^{\text{in}} \in \mathbb{R}^{N_x \times (1+N_u)}$  and  $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$  are the input and recurrent weight matrices respectively, and  $\alpha \in (0, 1]$  is the leaking rate. Other sigmoid wrappers can be used besides the tanh, which however is the most common choice. The model is also sometimes used without the leaky integration, which is a special case of  $\alpha = 1$  and thus  $\tilde{\mathbf{x}}(n) \equiv \mathbf{x}(n)$ .



$$y(n) = \mathbf{W}^{\text{out}}[1; \mathbf{u}(n); \mathbf{x}(n)],$$

Fig. 1: An echo state network.

again stands for a vertical vector (or matrix) concatenation. An additional nonlinearity can be applied to  $y(n)$  in (4), as well as feedback connections  $\mathbf{W}^{\text{fb}}$  from  $y(n-1)$  to  $\tilde{\mathbf{x}}(n)$  in (2). A graphical

# Training ESN

$$\mathbf{Y}^{\text{target}} = \mathbf{W}^{\text{out}} \mathbf{X}$$

Probably the most universal and stable solution to (8) in this context is ridge regression, also known as regression with Tikhonov regularization:

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}} \mathbf{X}^T \left( \mathbf{X} \mathbf{X}^T + \beta \mathbf{I} \right)^{-1}, \quad (9)$$

where  $\beta$  is a regularization coefficient explained in Section 4.2, and  $\mathbf{I}$  is the identity matrix.

Overfitting (regularization):

$$\mathbf{W}^{\text{out}} = \arg \min_{\mathbf{W}^{\text{out}}} \frac{1}{N_y} \sum_{i=1}^{N_y} \left( \sum_{n=1}^T (y_i(n) - y_i^{\text{target}}(n))^2 + \beta \|\mathbf{w}_i^{\text{out}}\|^2 \right),$$

# Beyond echo state networks

- Good aspects of ESNs  
Echo state networks can be trained very fast because they just fit a linear model.
- They demonstrate that it's very important to initialize weights sensibly.
- They can do impressive modeling of one-dimensional time-series.
  - but they cannot compete seriously for high-dimensional data.
- Bad aspects of ESNs  
They need many more hidden units for a given task than an RNN that learns the hidden→hidden weights.

# Similar models

- Liquid State Machines (Maas et al., 2002)
  - A spiking version of Echo-state networks
- Extreme Learning Machines
  - Feed-forward network with a hidden layer.
  - Input-to-hidden weights are randomly initialized and never updated

# Time Delay Neural Networks

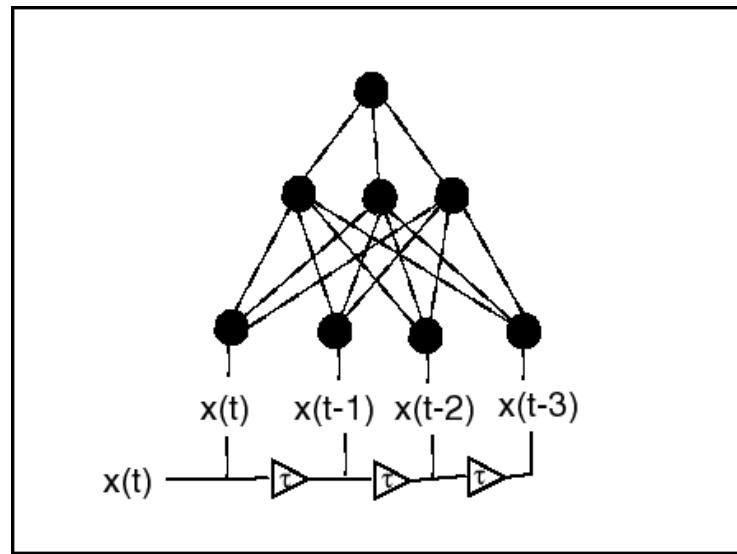


Fig: <https://www.willamette.edu/~gorr/classes/cs449/Temporal/tappedDelay.htm>

# Skipped points

# Skipping

- Stability
- Continuous-time recurrent networks
- Attractor networks

Stability of Discrete Time Recurrent Neural Networks and Nonlinear optimization problems

Dr. Nikita Barabanov, and Jayant Singh

**Abstract** We consider the method of Reduction of Dissipativity Domain to prove global Lyapunov stability of Discrete Time Recurrent Neural Networks. The standard and advanced criteria for Absolute Stability of these essentially nonlinear systems produce rather weak results. The method mentioned above is proved to be more powerful. It involves a multi-step procedure with maximization of special nonconvex functions over polytopes on every step. We derive conditions which guarantee an existence of at most one point of local maximum for such functions over every hyperplane. This nontrivial result is valid for wide range of neuron transfer functions.

---

# An Empirical Exploration of Recurrent Network Architectures

---

**Rafal Jozefowicz**

Google Inc.

RAFALJ@GOOGLE.COM

**Wojciech Zaremba**

New York University, Facebook<sup>1</sup>

WOJ.ZAREMBA @GMAIL.COM

**Ilya Sutskever**

Google Inc.

ILYASU@GOOGLE.COM

Under review as a conference paper at ICLR 2016

---

## VISUALIZING AND UNDERSTANDING RECURRENT NETWORKS

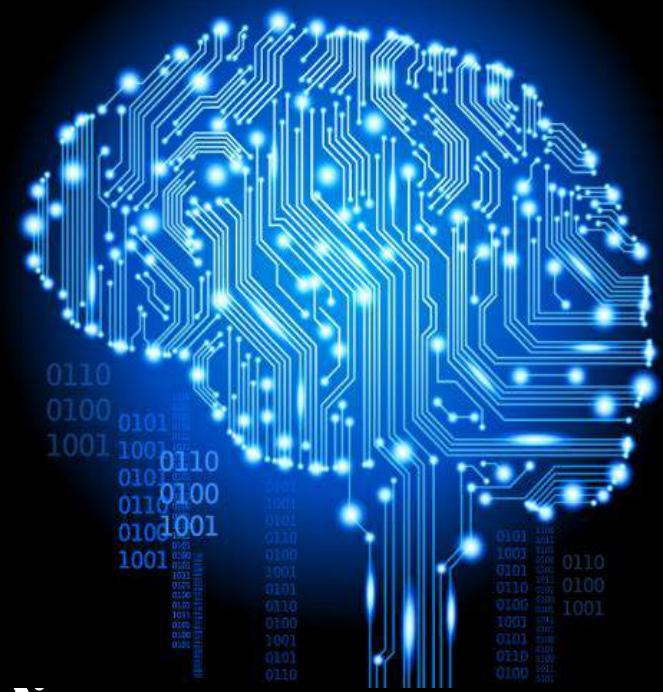
**Andrej Karpathy\***      **Justin Johnson\***      **Li Fei-Fei**  
Department of Computer Science, Stanford University  
`{karpathy, jcjohns, feifeili}@cs.stanford.edu`

# CENG 783

# Deep Learning

*Week – 12*  
*NTM, AEs*

Sinan Kalkan



© AlchemyAPI



# Today

- Neural Turing Machines
- Auto-encoders
- NOTES:
  - Final Exam date: 31 May, 17:00.
  - HW3 announced. Due date: 24 May, 23:55.
  - Project demos and papers due: 6 June.
  - **Extra lecture: 29 May 9:40-12:30**

# Neural Turing Machines

# Why need other mechanisms?

- We mentioned before that RNNs are Turing Complete, right?
- The issues are:
  - The vanishing/exploding gradients (LSTM and other tricks address these issues)
  - However, parameters explode in LSTMs with the number of layers (and stacks)
  - The answer to addressing bigger networks with less parameters is a better abstraction of the computational components, e.g., in a form similar to Turing machines

Bilbo travelled to the cave. Gollum dropped the ring there. Bilbo took the ring.

Bilbo went back to the Shire. Bilbo left the ring there. Frodo got the ring.

Frodo journeyed to Mount-Doom. Frodo dropped the ring there. Sauron died.

Frodo went back to the Shire. Bilbo travelled to the Grey-havens. The End.

Where is the ring? A: Mount-Doom

Where is Bilbo now? A: Grey-havens

Where is Frodo now? A: Shire

# Turing Machine

Wikipedia:

Following Hopcroft and Ullman (1979, p. 148), a (one-tape) Turing machine can be formally defined as a 7-tuple  $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$  where

- $Q$  is a finite, non-empty set of states
- $\Gamma$  is a finite, non-empty set of *tape alphabet symbols*
- $b \in \Gamma$  is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$  is the set of *input symbols*
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  is a *partial function* called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.) If  $\delta$  is not defined on the current state and the current tape symbol, then the machine halts.<sup>[21]</sup>
- $q_0 \in Q$  is the *initial state*
- $F \subseteq Q$  is the set of *final* or *accepting states*. The initial tape contents is said to be *accepted* by  $M$  if it eventually halts in a state from  $F$ .

Anything that operates according to these specifications is a Turing machine.



(a) Alan Turing

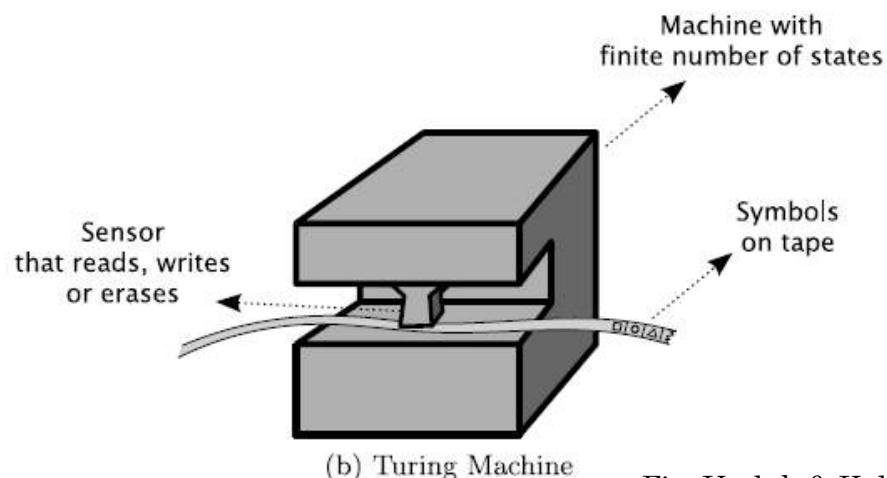
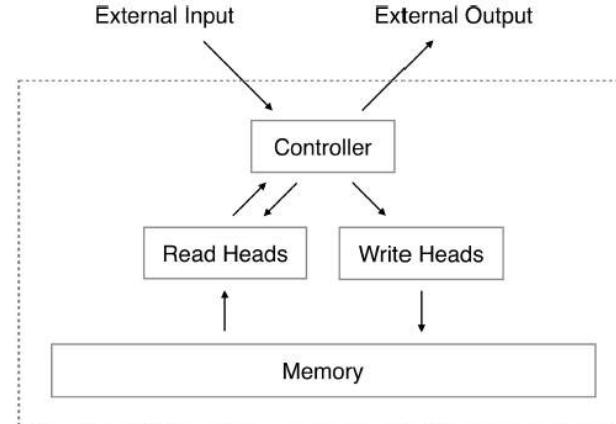


Fig: Ucoluk & Kalkan, 2012

# Neural Turing Machines

- If we make every component differentiable, we can train such a complex machine
- Accessing only a part of the network is problematic
  - Unlike a computer (TM), we need a ‘blurry’ access mechanism



**Figure 1: Neural Turing Machine Architecture.** During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.

---

## Neural Turing Machines

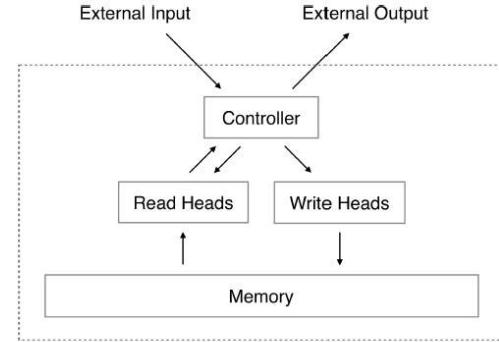
---

Alex Graves  
Greg Wayne  
Ivo Danihelka

gravesa@google.com  
gregwayne@google.com  
danihelka@google.com

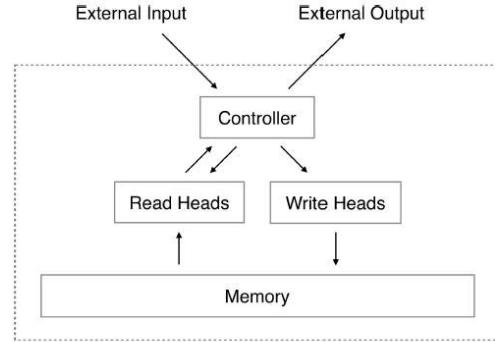
# Neural Turing Machines: Reading

- Let memory  $\mathbf{M}$  be an  $N \times M$  matrix
  - $N$ : the number of “rows”
  - $M$ : the size of each row (vector)
- Let  $\mathbf{M}_t$  be the memory state at time  $t$
- $w_t$ : a vector of weightings over  $N$  locations emitted by the read head at time  $t$ . Since the weights are normalized:
 
$$\sum_i w_t(i) = 1, \quad 0 \leq w_t(i) \leq 1, \forall i$$
- $\mathbf{r}_t$ : the read vector of length  $M$ :
 
$$\mathbf{r}_t \leftarrow \sum_i w_t(i) \mathbf{M}_t(i).$$
- which is differentiable, and therefore, trainable.



**Figure 1: Neural Turing Machine Architecture.** During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.

# Neural Turing Machines: Writing



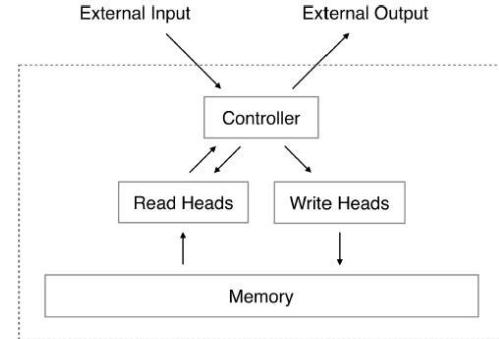
**Figure 1: Neural Turing Machine Architecture.** During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.

- Writing = erasing content + adding new content
  - Inspired from LSTM's forgetting and addition gates.
- Erasing: Multiply with an erase vector  $\mathbf{e}_t \in [0,1]^M$ 
$$\hat{\mathbf{M}}_t(i) \leftarrow \mathbf{M}_{t-1}(i)[\mathbf{1} - w_t(i)\mathbf{e}_t]$$

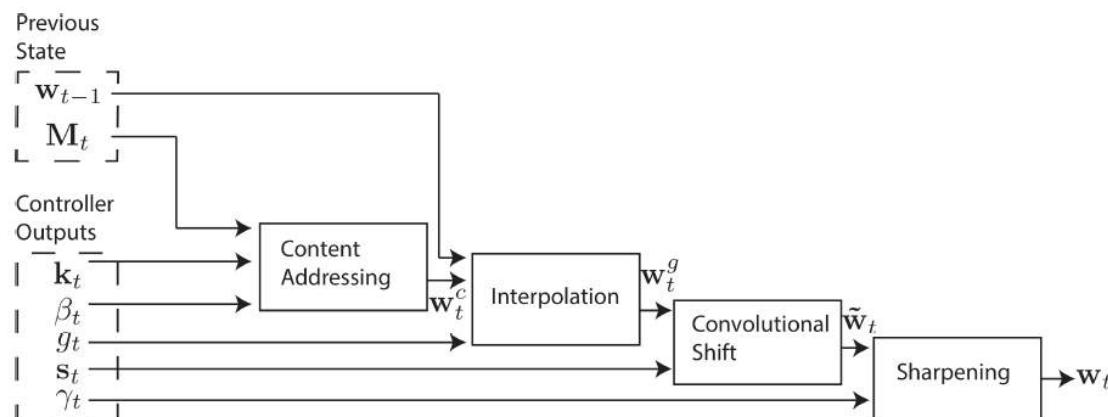
$\mathbf{1}$ : vector of ones. Multiplication here is pointwise.
- Adding: Add an add vector  $\mathbf{a}_t \in [0,1]^M$ :
$$\mathbf{M}_t(i) \leftarrow \hat{\mathbf{M}}_t(i) + w_t(i)\mathbf{a}_t$$

# Neural Turing Machines: Addressing

- Content-based addressing
- Location-based addressing
  - In a sense, use variable “names” to access content



**Figure 1: Neural Turing Machine Architecture.** During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.



**Figure 2: Flow Diagram of the Addressing Mechanism.** The key vector,  $k_t$ , and key strength,  $\beta_t$ , are used to perform content-based addressing of the memory matrix,  $M_t$ . The resulting content-based weighting is interpolated with the weighting from the previous time step based on the value of the *interpolation gate*,  $g_t$ . The *shift weighting*,  $s_t$ , determines whether and by how much the weighting is rotated. Finally, depending on  $\gamma_t$ , the weighting is sharpened and used for memory access.

# Neural Turing Machines: Content-based Addressing

- Each head (reading or writing head) produces an  $M$  length key vector  $\mathbf{k}_t$ 
  - $\mathbf{k}_t$  is compared to each vector  $\mathbf{M}_t(i)$  using a similarity measure  $K[.,.]$ , e.g., cosine similarity:

$$K[\mathbf{u}, \mathbf{v}] = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|}$$

- From these similarity measures, we obtain a vector of “addressing”:

$$w_t^c(i) \leftarrow \frac{\exp(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(i)])}{\sum_j \exp(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(j)])}$$

- $\beta_t$ : amplifies or attenuates the precision of the focus

# Neural Turing Machines: Location-based Addressing

- Important for e.g. iteration over memory locations, or jumping to an arbitrary memory location
- First: Interpolation between addressing schemes using “interpolation gate”  $g_t$ :

$$\mathbf{w}_t^g \leftarrow g_t \mathbf{w}_t^c + (1 - g_t) \mathbf{w}_{t-1}$$

- If  $g_t = 1$ : weight from content-addressable component is used
- If  $g_t = 0$ : weight from previous step is used
- Second: rotationally shift weight to achieve location-based addressing using convolution:

$$\hat{w}_t(i) \leftarrow \sum_{j=0}^{N-1} w_t^g(j) s_t(i-j)$$

- $\mathbf{s}_t$ : shift amount. Three elements for how “much” to shift left, right or keep as it is.
- It needs to be “sharp”. To keep it sharp, each head emits a scalar  $\gamma^t \geq 1$ :

$$w_t(i) \leftarrow \frac{\hat{w}_t(i)^{\gamma^t}}{\sum_j \hat{w}_t(j)^{\gamma^t}}$$

# Neural Turing Machines: Controller Network

- Free parameters
  - The size of the memory
  - Number of read-write heads
  - Range of allowed rotation shifts
  - Type of the neural network for controller
- Alternatives:
  - A recurrent network such as LSTM with its own memory
    - These memory units might be considered like “registers” on the CPU
  - A feed-forward network
    - Can use the memory to achieve recurrence
    - More transparent

# Neural Turing Machines: Training

- Binary targets
  - Logistic sigmoid output layers
  - Cross-entropy loss
- Other schemes possible
- Tasks:
  - Copy from input to output
  - Repeat Copy: Make n copies of the input
  - Associative recall: Present a part of a sequence to recall the remaining part
  - N-gram: Learn distribution of 6-grams and make predictions for the next bit based on this distribution
  - Priority sort: Associate a priority as part of each vector and as the target place the sequence according to the priority

Task	#Heads	Controller Size	Memory Size	Learning Rate	#Parameters
Copy	1	100	$128 \times 20$	$10^{-4}$	17,162
Repeat Copy	1	100	$128 \times 20$	$10^{-4}$	16,712
Associative	4	256	$128 \times 20$	$10^{-4}$	146,845
N-Grams	1	100	$128 \times 20$	$3 \times 10^{-5}$	14,656
Priority Sort	8	512	$128 \times 20$	$3 \times 10^{-5}$	508,305

Table 1: NTM with Feedforward Controller Experimental Settings

Task	#Heads	Controller Size	Memory Size	Learning Rate	#Parameters
Copy	1	100	$128 \times 20$	$10^{-4}$	67,561
Repeat Copy	1	100	$128 \times 20$	$10^{-4}$	66,111
Associative	1	100	$128 \times 20$	$10^{-4}$	70,330
N-Grams	1	100	$128 \times 20$	$3 \times 10^{-5}$	61,749
Priority Sort	5	$2 \times 100$	$128 \times 20$	$3 \times 10^{-5}$	269,038

Table 2: NTM with LSTM Controller Experimental Settings

Task	Network Size	Learning Rate	#Parameters
Copy	$3 \times 256$	$3 \times 10^{-5}$	1,352,969
Repeat Copy	$3 \times 512$	$3 \times 10^{-5}$	5,312,007
Associative	$3 \times 256$	$10^{-4}$	1,344,518
N-Grams	$3 \times 128$	$10^{-4}$	331,905
Priority Sort	$3 \times 128$	$3 \times 10^{-5}$	384,424

Table 3: LSTM Network Experimental Settings

# Neural Turing Machines: Training

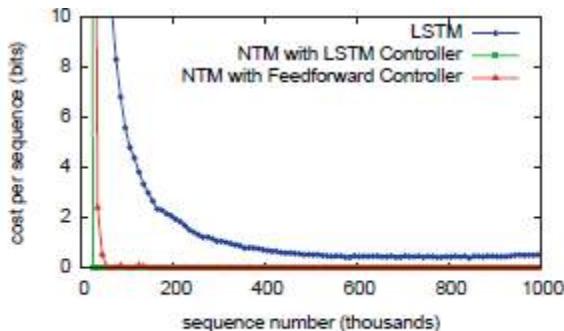


Figure 3: Copy Learning Curves.

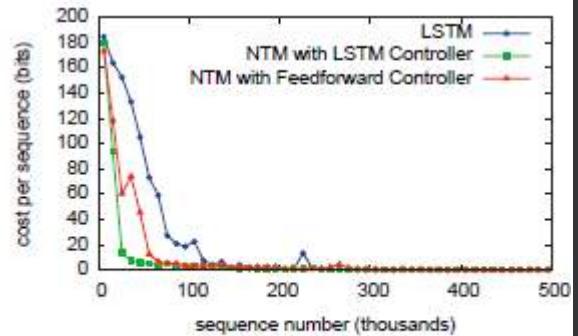


Figure 7: Repeat Copy Learning Curves.

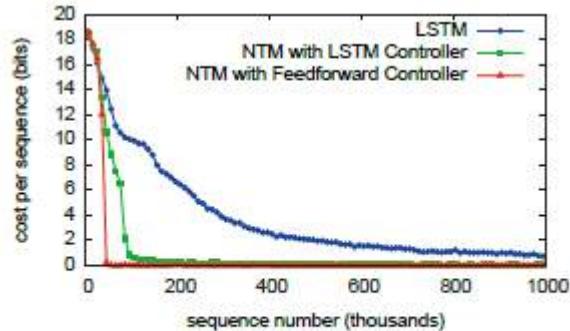


Figure 10: Associative Recall Learning Curves for NTM and LSTM.

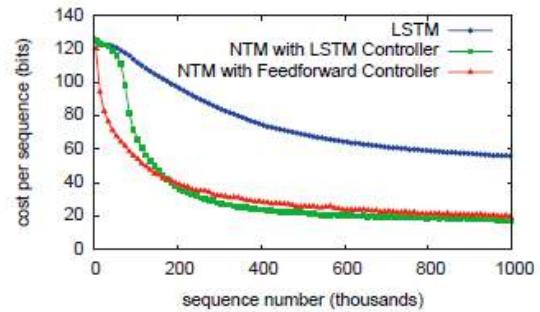


Figure 18: Priority Sort Learning Curves.

# Reinforced version

Under review as a conference paper at ICLR 2016

---

## REINFORCEMENT LEARNING NEURAL TURING MACHINES - REVISED

**Wojciech Zaremba<sup>1,2</sup>**  
New York University  
Facebook AI Research  
[woj.zaremba@gmail.com](mailto:woj.zaremba@gmail.com)

**Ilya Sutskever<sup>2</sup>**  
Google Brain  
[ilyasu@google.com](mailto:ilyasu@google.com)

Other  
variants/attempts

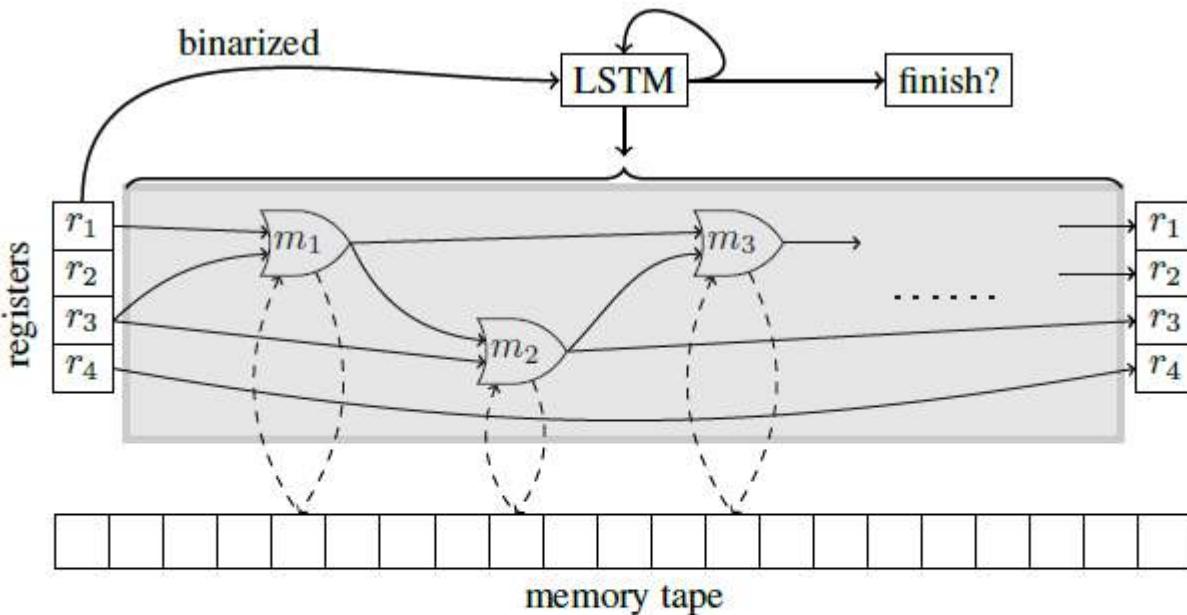


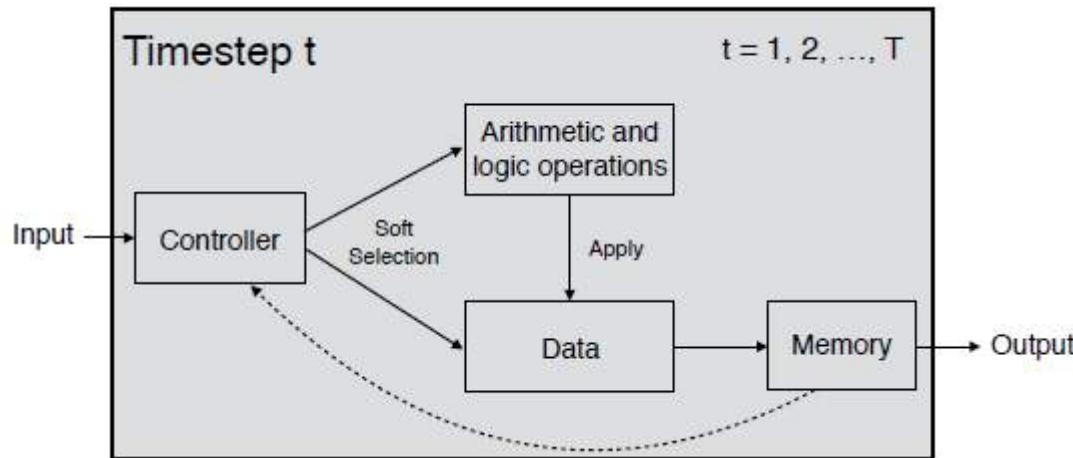
Figure 2: One timestep of the NRAM architecture with  $R = 4$  registers. The LSTM controller gets the „binarized” values  $r_1, r_2, \dots$  stored in the registers as inputs and outputs the description of the circuit in the grey box and the probability of finishing the execution in the current timestep (See Sec. 3.3 for more detail). The weights of the solid thin connections are outputted by the controller. The weights of the solid thick connections are trainable parameters of the model. Some of the modules (i.e. READ and WRITE) may interact with the memory tape (dashed connections).

Published as a conference paper at ICLR 2016

## NEURAL RANDOM-ACCESS MACHINES

Karol Kurach\* & Marcin Andrychowicz\* & Ilya Sutskever  
 Google  
 {kkurach, marcina, ilyasu}@google.com

# Neural Programmer



Published as a conference paper at ICLR 2016

## NEURAL PROGRAMMER: INDUCING LATENT PROGRAMS WITH GRADIENT DESCENT

**Arvind Neelakantan\***  
University of Massachusetts Amherst  
[arvind@cs.umass.edu](mailto:arvind@cs.umass.edu)

**Quoc V. Le**  
Google Brain  
[qvl@google.com](mailto:qvl@google.com)

**Ilya Sutskever**  
Google Brain  
[ilyasu@google.com](mailto:ilyasu@google.com)

# Pointer networks

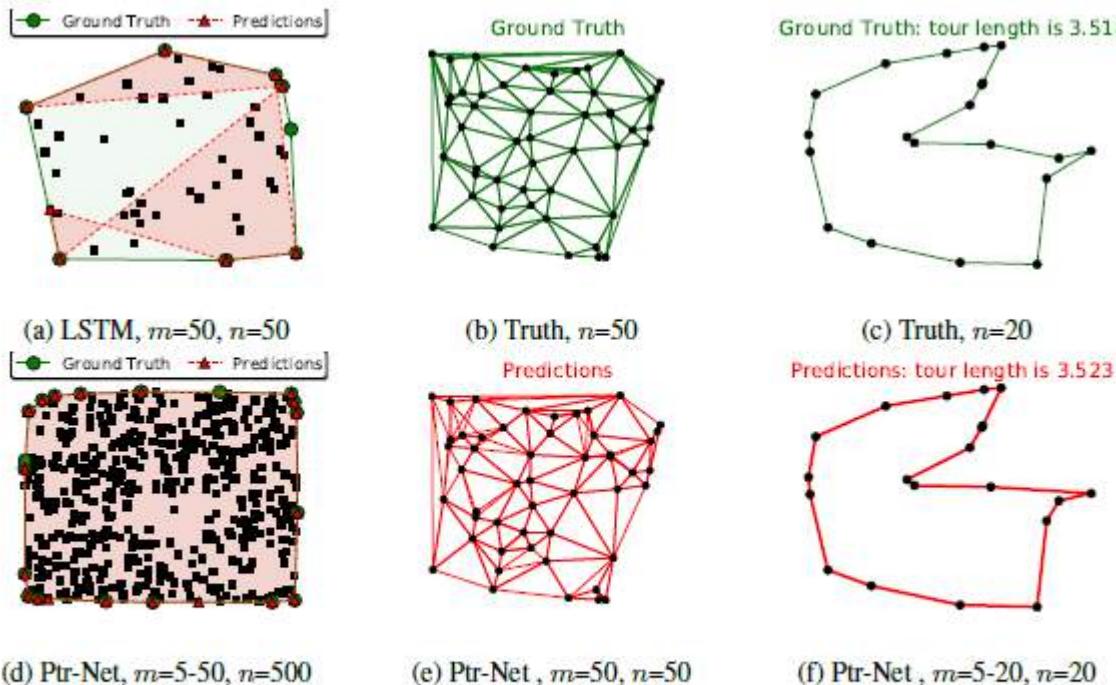


Figure 3: Examples of our model on Convex hulls (left), Delaunay (center) and TSP (right), trained on  $m$  points, and tested on  $n$  points. A failure of the LSTM sequence-to-sequence model for Convex hulls is shown in (a). Note that the baselines cannot be applied to a different length from training.

# Memory Networks

Bilbo travelled to the cave. Gollum dropped the ring there. Bilbo took the ring.

Bilbo went back to the Shire. Bilbo left the ring there. Frodo got the ring.

Frodo journeyed to Mount-Doom. Frodo dropped the ring there. Sauron died.

Frodo went back to the Shire. Bilbo travelled to the Grey-havens. The End.

Where is the ring? A: Mount-Doom

Where is Bilbo now? A: Grey-havens

Where is Frodo now? A: Shire

Published as a conference paper at ICLR 2015

## MEMORY NETWORKS

**Jason Weston, Sumit Chopra & Antoine Bordes**  
Facebook AI Research  
770 Broadway  
New York, USA  
[{jase, spchopra, abordes}@fb.com](mailto:{jase, spchopra, abordes}@fb.com)

# Universal Turing Machine

The NLP Engine: A Universal Turing Machine for NLP

Jiwei Li<sup>1</sup> and Eduard Hovy<sup>2</sup>

<sup>1</sup>Computer Science Department, Stanford University, Stanford, CA 94305

<sup>2</sup>Language Technology Institute, Carnegie Mellon University, Pittsburgh, PA 15213  
jiweil@stanford.edu    ehover@andrew.cmu.edu

2015

# Inferring and Executing Programs for Visual Reasoning

Justin Johnson<sup>1</sup>

Bharath Hariharan<sup>2</sup>

Laurens van der Maaten<sup>2</sup>

Judy Hoffman<sup>1</sup>

Li Fei-Fei<sup>1</sup>

C. Lawrence Zitnick<sup>2</sup>

Ross Girshick<sup>2</sup>

<sup>1</sup>Stanford University

<sup>2</sup>Facebook AI Research

2017



How many chairs are at the table?

Is there a pedestrian in my lane?



Is the person with the blue hat touching the bike in the back?

Is there a matte cube that has the same size as the red metal object?

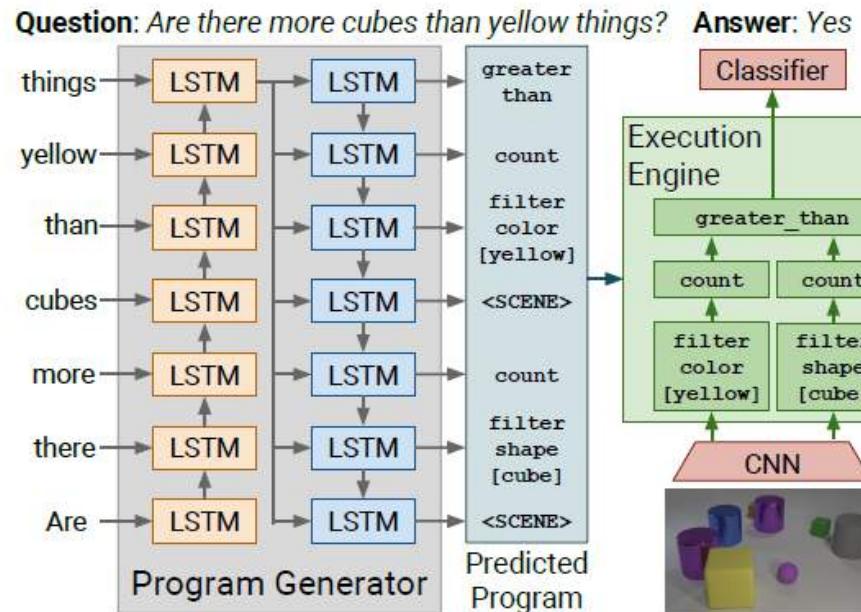


Figure 2. System overview. The **program generator** is a sequence-to-sequence model which inputs the question as a sequence of words and outputs a program as a sequence of functions, where the sequence is interpreted as a prefix traversal of the program’s abstract syntax tree. The **execution engine** executes the program on the image by assembling a neural module network [2] mirroring the structure of the predicted program.

# Newer studies

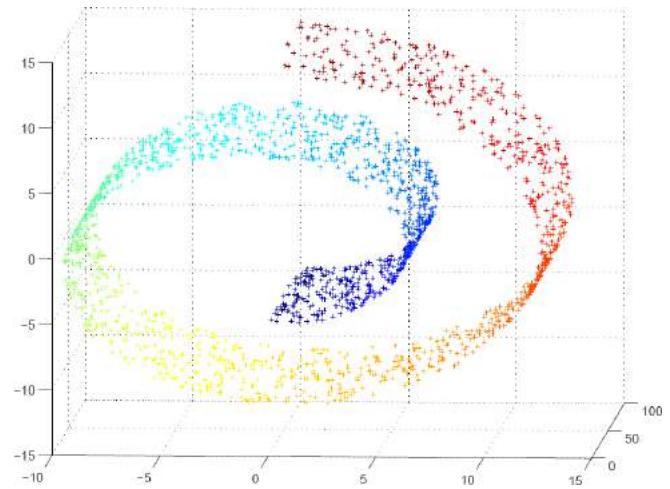
- <https://deepmind.com/blog/differentiable-neural-computers/>
- Differentiable Neural Machines

# Unsupervised pre-training with Auto-encoders

# Now

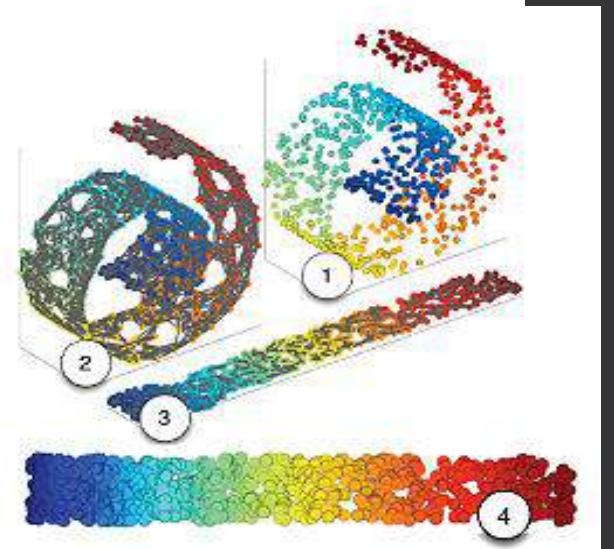
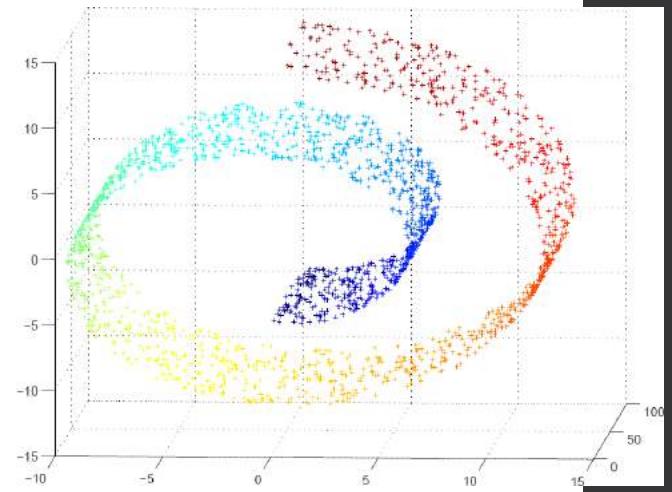
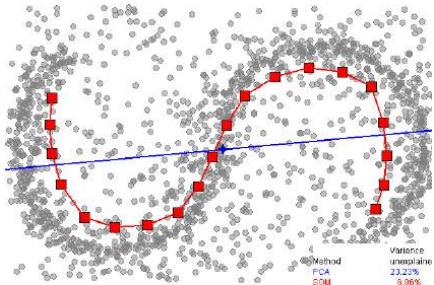
- Manifold Learning
  - Principle Component Analysis
  - Independent Component Analysis
- Autoencoders
- Sparse autoencoders
- K-sparse autoencoders
- Denoising autoencoders
- Contraction autoencoders

# Manifold Learning



# Manifold Learning

- Discovering the “hidden” structure in the high-dimensional space
- Manifold: “hidden” structure.
- Non-linear dimensionality reduction



[http://www.convexoptimization.com/dattorro/manifold\\_learning.html](http://www.convexoptimization.com/dattorro/manifold_learning.html)

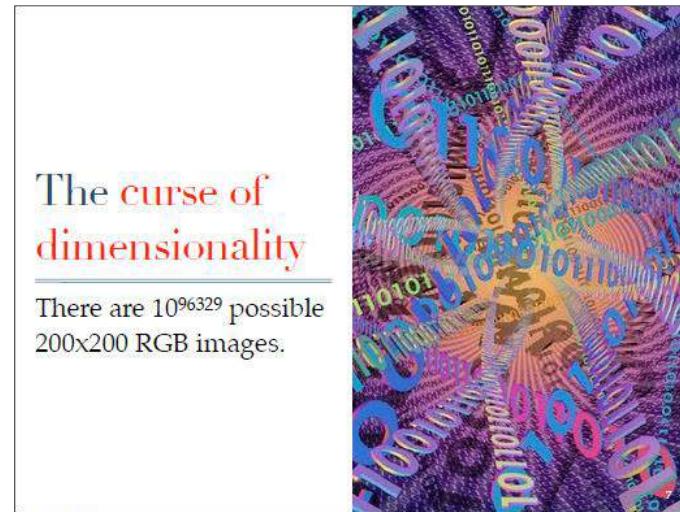
# Manifold Learning

- Many approaches:
  - Self-Organizing Map (Kohonen map/network)
  - **Auto-encoders**
  - Principles curves & manifolds: Extension of PCA
  - Kernel PCA, Nonlinear PCA
  - Curvilinear Component Analysis
  - Isomap: Floyd-Marshall + Multidimensional scaling
  - Data-driven high-dimensional scaling
  - Locally-linear embedding
  - ...

# Manifold learning

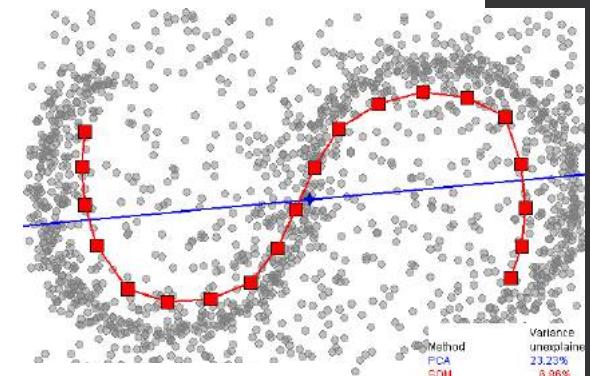
- Autoencoders learn lower-dimensional manifolds embedded in higher-dimensional manifolds
- Assumption: “Natural data in high dimensional spaces concentrates close to lower dimensional manifolds”
  - Natural images occupy a very small fraction in a space of possible images

(Pascal Vincent)



# Manifold Learning

- Many approaches:
  - Self-Organizing Map (Kohonen map/network)



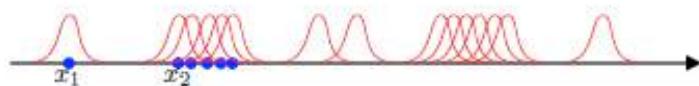
## Algorithm [ edit ]

1. Randomize the map's nodes' weight vectors
2. Grab an input vector  $\mathbf{D}(t)$
3. Traverse each node in the map
  1. Use the Euclidean distance formula to find the similarity between the input vector and the map's node's weight vector
  2. Track the node that produces the smallest distance (this node is the best matching unit, BMU)
4. Update the nodes in the neighborhood of the BMU (including the BMU itself) by pulling them closer to the input vector
  1.  $\mathbf{W}_v(s + 1) = \mathbf{W}_v(s) + \Theta(u, v, s) \alpha(s)(\mathbf{D}(t) - \mathbf{W}_v(s))$
5. Increase  $s$  and repeat from step 2 while  $s < \lambda$

A variant algorithm:

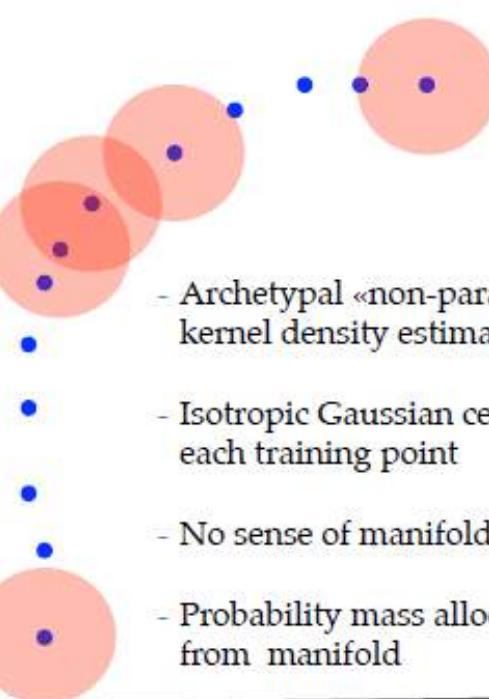
1. Randomize the map's nodes' weight vectors
2. Traverse each input vector in the input data set
  1. Traverse each node in the map
    1. Use the Euclidean distance formula to find the similarity between the input vector and the map's node's weight vector
    2. Track the node that produces the smallest distance (this node is the best matching unit, BMU)
  2. Update the nodes in the neighborhood of the BMU (including the BMU itself) by pulling them closer to the input vector
    1.  $\mathbf{W}_v(s + 1) = \mathbf{W}_v(s) + \Theta(u, v, s) \alpha(s)(\mathbf{D}(t) - \mathbf{W}_v(s))$
3. Increase  $s$  and repeat from step 2 while  $s < \lambda$

# Non-parametric density estimation

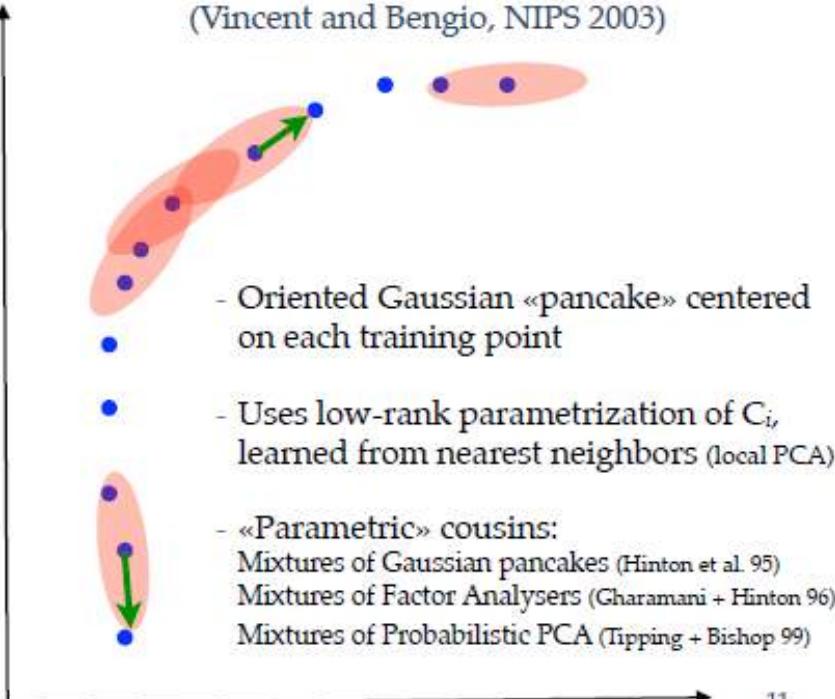


$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n \mathcal{N}(x; x_i, C_i)$$

Classical Parzen Windows  
density estimator



Manifold Parzen Windows  
density estimator  
(Vincent and Bengio, NIPS 2003)



# Non-local manifold Parzen windows

(Bengio, Larochelle, Vincent, NIPS 2006)

**Isotropic Parzen:**

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n \mathcal{N}(x; x_i, \underbrace{\sigma^2 I}_{\text{isotropic}})$$

**Manifold Parzen:**

(Vincent and Bengio, NIPS 2003)

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n \mathcal{N}(x; x_i, \underbrace{C_i}_{d_M \text{ high variance directions from PCA on } k \text{ nearest neighbors}})$$

**Non-local manifold Parzen:**  $\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n \mathcal{N}(x; \underbrace{\mu(x_i), C(x_i)}_{d_M \text{ high variance directions output by neural network trained to maximize likelihood of } k \text{ nearest neighbors}})$

12

# Principle Component Analysis (PCA)

- Principle Components:
  - Orthogonal directions with most **variance**
  - **Eigen-vectors** of the **co-variance** matrix

- Mathematical background:
  - **Orthogonality:**
    - Two vectors  $\vec{u}$  and  $\vec{v}$  are orthogonal iff  $\vec{u} \cdot \vec{v} = 0$
  - **Variance:**

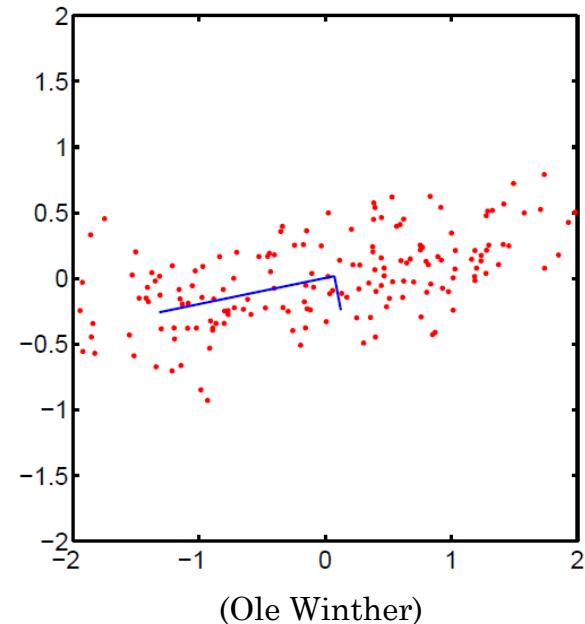
$$\sigma(X)^2 = Var(X) = E[(X - \mu)^2] = \sum_i p(x_i)(x_i - \mu)^2$$

where the (weighted) mean,  $\mu = E[X] = \sum_i p(x_i)x_i$ .

If  $p(x_i) = 1/N$ :

$$Var(X) = \frac{1}{N} \sum_i (x_i - \mu)^2$$

$$\mu = \frac{1}{N} \sum_i x_i$$



# Mathematical background for PCA: Covariance

- Co-variance:
  - Measures how two random variables change wrt each other:

$$\begin{aligned} \text{Cov}(X, Y) &= E[(X - E[X])(Y - E[Y])] \\ &= \frac{1}{N} \sum_i (\textcolor{red}{x}_i - E[X])(\textcolor{blue}{y}_i - E[Y]) \end{aligned}$$

- If big values of X & big values of Y “co-occur” and small values of X & small values of Y “co-occur” → high co-variance.
- Otherwise, small co-variance.

# Mathematical background for PCA: Covariance Matrix

- Co-variance Matrix:
  - Denoted usually by  $\Sigma$
  - For an  $n$ -dimensional space:

$$\Sigma_{ij} = \text{Cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)]$$

$$\Sigma = \begin{bmatrix} E[(X_1 - \mu_1)(X_1 - \mu_1)] & E[(X_1 - \mu_1)(X_2 - \mu_2)] & \cdots & E[(X_1 - \mu_1)(X_n - \mu_n)] \\ E[(X_2 - \mu_2)(X_1 - \mu_1)] & E[(X_2 - \mu_2)(X_2 - \mu_2)] & \cdots & E[(X_2 - \mu_2)(X_n - \mu_n)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(X_n - \mu_n)(X_1 - \mu_1)] & E[(X_n - \mu_n)(X_2 - \mu_2)] & \cdots & E[(X_n - \mu_n)(X_n - \mu_n)] \end{bmatrix}.$$

# Mathematical background for PCA: Covariance Matrix

- Co-variance Matrix:

$$\begin{aligned}\Sigma_{ij} &= \text{Cov}(X_i, X_j) \\ &= E[(X_i - \mu_i)(X_j - \mu_j)]\end{aligned}$$

- Properties

1.  $\Sigma = E(\mathbf{XX}^T) - \boldsymbol{\mu}\boldsymbol{\mu}^T$

2.  $\Sigma$  is positive-semidefinite and symmetric.

3.  $\text{cov}(\mathbf{AX} + \mathbf{a}) = \mathbf{A} \text{ cov}(\mathbf{X}) \mathbf{A}^T$

4.  $\text{cov}(\mathbf{X}, \mathbf{Y}) = \text{cov}(\mathbf{Y}, \mathbf{X})^T$

5.  $\text{cov}(\mathbf{X}_1 + \mathbf{X}_2, \mathbf{Y}) = \text{cov}(\mathbf{X}_1, \mathbf{Y}) + \text{cov}(\mathbf{X}_2, \mathbf{Y})$

6. If  $p = q$ , then  $\text{var}(\mathbf{X} + \mathbf{Y}) = \text{var}(\mathbf{X}) + \text{cov}(\mathbf{X}, \mathbf{Y}) + \text{cov}(\mathbf{Y}, \mathbf{X}) + \text{var}(\mathbf{Y})$

7.  $\text{cov}(\mathbf{AX} + \mathbf{a}, \mathbf{B}^T \mathbf{Y} + \mathbf{b}) = \mathbf{A} \text{ cov}(\mathbf{X}, \mathbf{Y}) \mathbf{B}$

8. If  $\mathbf{X}$  and  $\mathbf{Y}$  are independent or uncorrelated, then  $\text{cov}(\mathbf{X}, \mathbf{Y}) = \mathbf{0}$

(Wikipedia)

$M$  is called *positive-semidefinite* (or sometimes *nonnegative-definite*) if

$$x^* M x \geq 0$$

for all  $x$  in  $\mathbb{C}^n$  (or, all  $x$  in  $\mathbb{R}^n$  for the real matrix).

(Wikipedia)

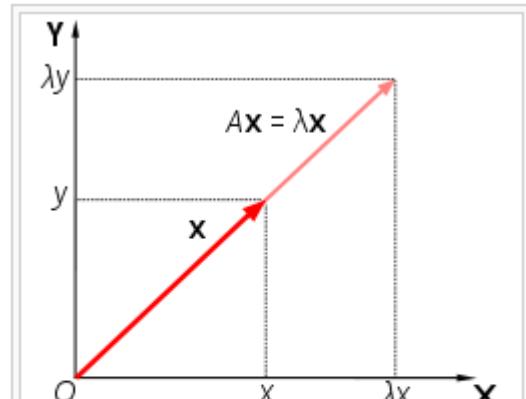
# Mathematical background for PCA: Eigenvectors & Eigenvalues

- **Eigenvectors** and **eigenvalues**:
  - $\vec{v}$  is an eigenvector of a square matrix  $A$  if

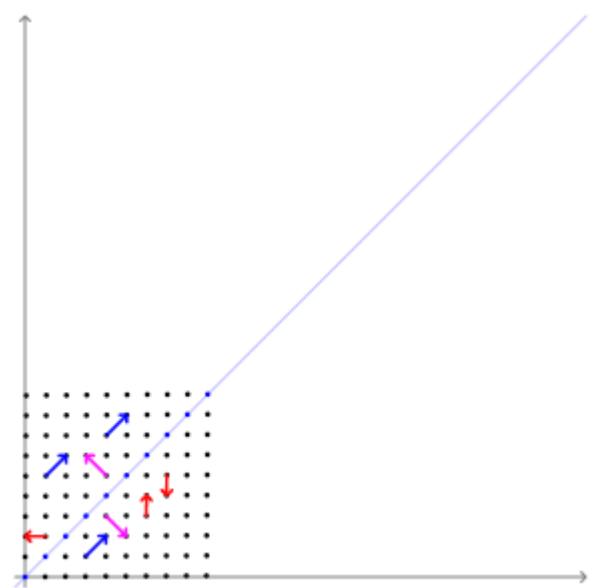
$$A\vec{v} = \lambda\vec{v}$$

where  $\lambda$  is the eigenvalue (scalar) associated with  $\vec{v}$ .

- Interpretation:
  - “Transformation”  $A$  does not change the direction of the vector.
  - It changes the vector’s scale, i.e., the eigenvalue.
- Solution:
  - $(A - \lambda I)\vec{v} = 0 \rightarrow$  has a solution when the determinant  $|A - \lambda I|$  is zero.
  - Find the eigenvalues, then plug in those values to get the eigenvectors.



Matrix  $A$  acts by stretching the vector  $x$ ,  $\square$  not changing its direction, so  $x$  is an eigenvector of  $A$ .



The transformation matrix  $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$  preserves the direction of vectors parallel to  $v = (1, -1)^T$  (in purple) and  $w = (1, 1)^T$  (in blue). The vectors in red are not parallel to either eigenvector, so, their directions are changed by the transformation. See also: An extended

# Mathematical background for PCA: Eigenvectors & Eigenvalues Example

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

- Setting the determinant  $|A - \lambda I|$  to zero:

$$p(\lambda) = |A - \lambda I| = 3 - 4\lambda + \lambda^2 = 0,$$

- The roots:  $\lambda = 1$  and  $\lambda = 3$
- If you plug in those eigenvalues, for  $\lambda = 1$ :

$$(A - I)\mathbf{v} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{Bmatrix} v_1 \\ v_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix},$$

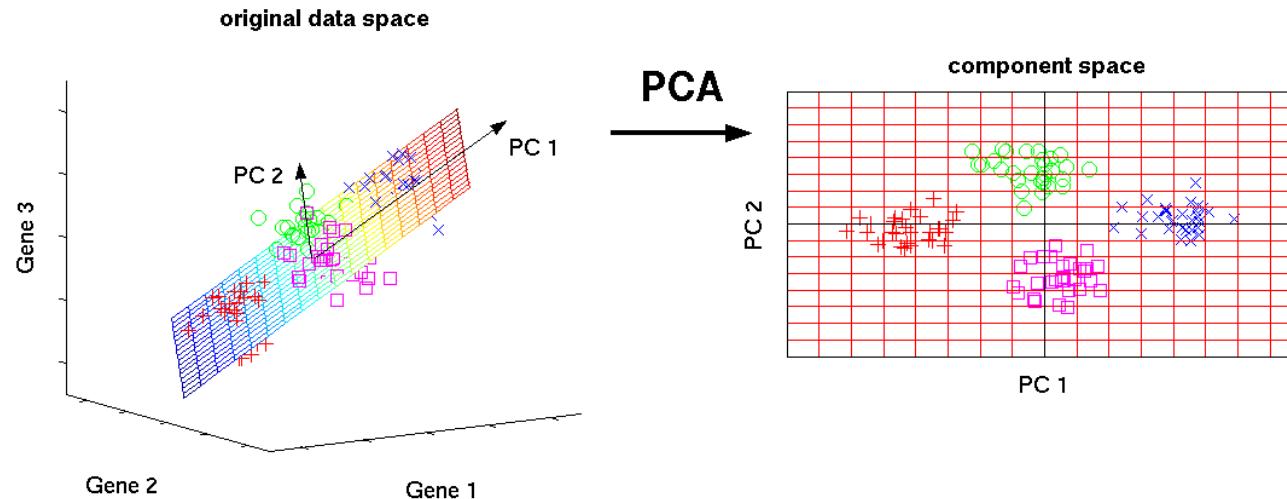
which gives  $\mathbf{v}_1 = \{1, -1\}$ . For  $\lambda = 3$ :

$$(A - 3I)\mathbf{w} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{Bmatrix} w_1 \\ w_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix},$$

which gives  $\mathbf{v}_2 = \{1, 1\}$ .

# PCA allows also dimensionality reduction

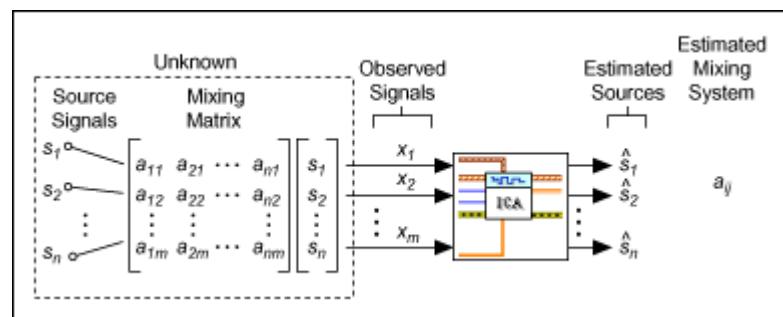
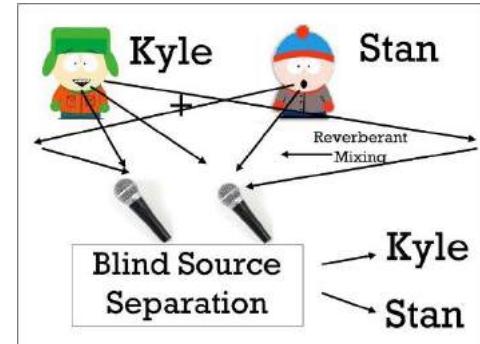
- Discard components whose eigenvalue is negligible.



See the following tutorial for more on PCA:  
[http://www.cs.princeton.edu/picasso/mats/PCA-Tutorial-Intuition\\_jp.pdf](http://www.cs.princeton.edu/picasso/mats/PCA-Tutorial-Intuition_jp.pdf)

# Independent Component Analysis (ICA)

- PCA assumes **Gaussianity**:
  - Data along a component should be explainable by a mean and a variance.
  - This may be violated by real signals in the nature.
- ICA:
  - Blind-source separation of **non-Gaussian** and **mutually-independent** signals.
- Mutual independence:

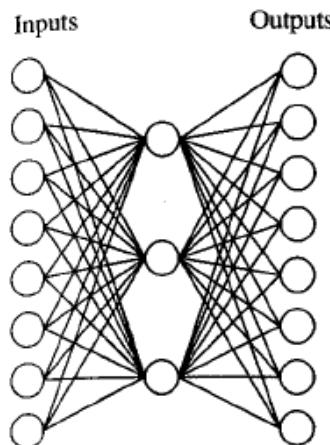


<https://cnx.org/contents/-ElGohfq@1.4:gFEtO206@1/Independent-Component-Analysis>

# Autoencoders

# Autoencoders

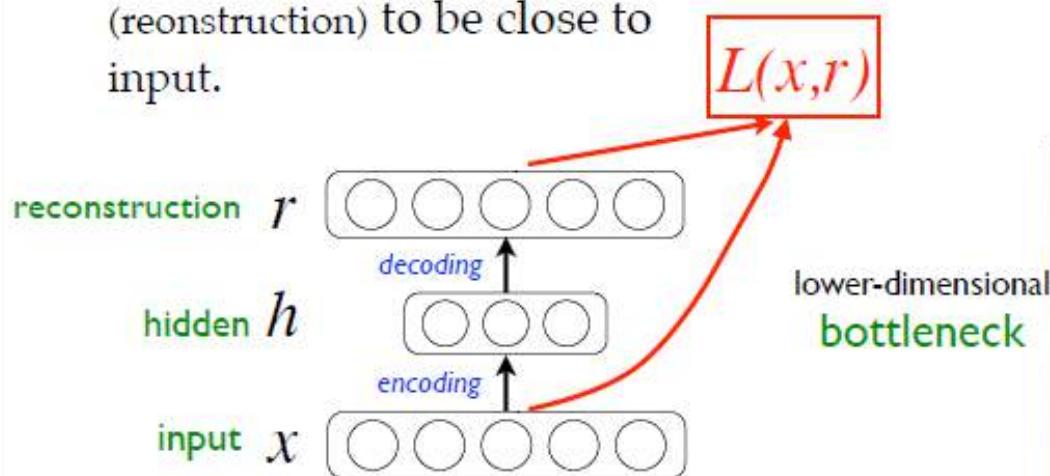
- Universal approximators
  - So are Restricted Boltzmann Machines
- Unsupervised learning
- Dimensionality reduction
- $\mathbf{x} \in \mathbb{R}^D \Rightarrow \mathbf{h} \in \mathbb{R}^M$  s.t.  $M < D$



Input	Hidden Values			Output		
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

# Autoencoders: MLPs used for «unsupervised» representation learning

- + Make **output** layer same size as **input** layer
- + Have **target** = **input**
- + **Loss** encourages output (reconstruction) to be close to input.



Autoencoders are also called

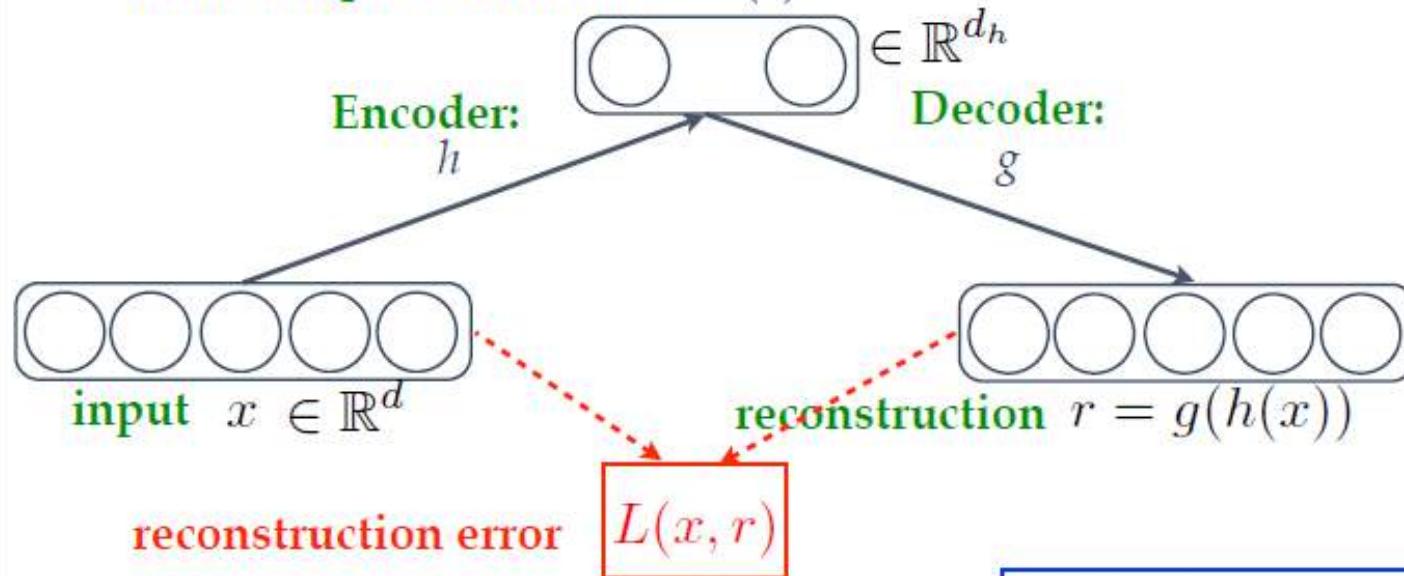
- Autoencoders
- Auto-associators
- Diabolo networks
- Sandglass-shaped net



The Diabolo

# Auto-Encoders (AE) for learning representations

hidden representation  $\mathbf{h} = h(\mathbf{x})$



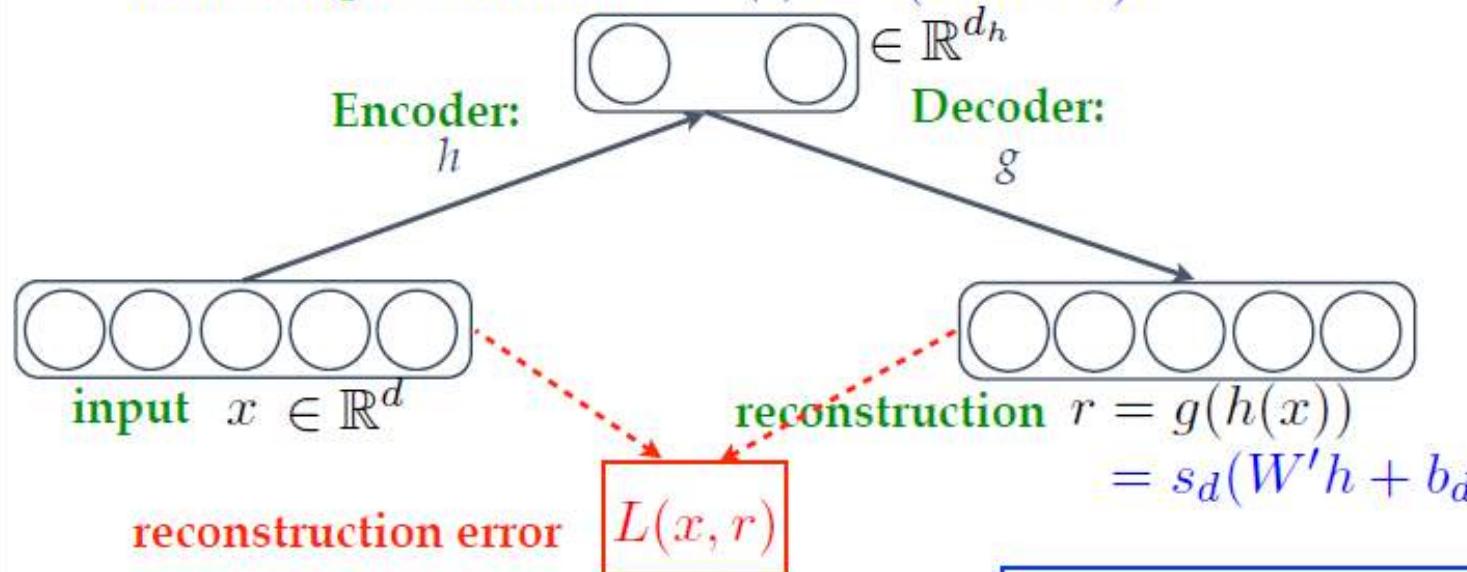
$$\text{Minimize} \quad \mathcal{J}_{\text{AE}} = \sum_{x \in D} L(x, g(h(x)))$$

19

# Auto-Encoders (AE) for learning representations

Typical form

hidden representation  $\mathbf{h} = h(\mathbf{x}) = s(W\mathbf{x} + b)$



squared error:  $\|x - r\|^2$   
or Bernoulli cross-entropy

Minimize

$$\mathcal{J}_{\text{AE}} = \sum_{x \in D} L(x, g(h(x)))$$

19

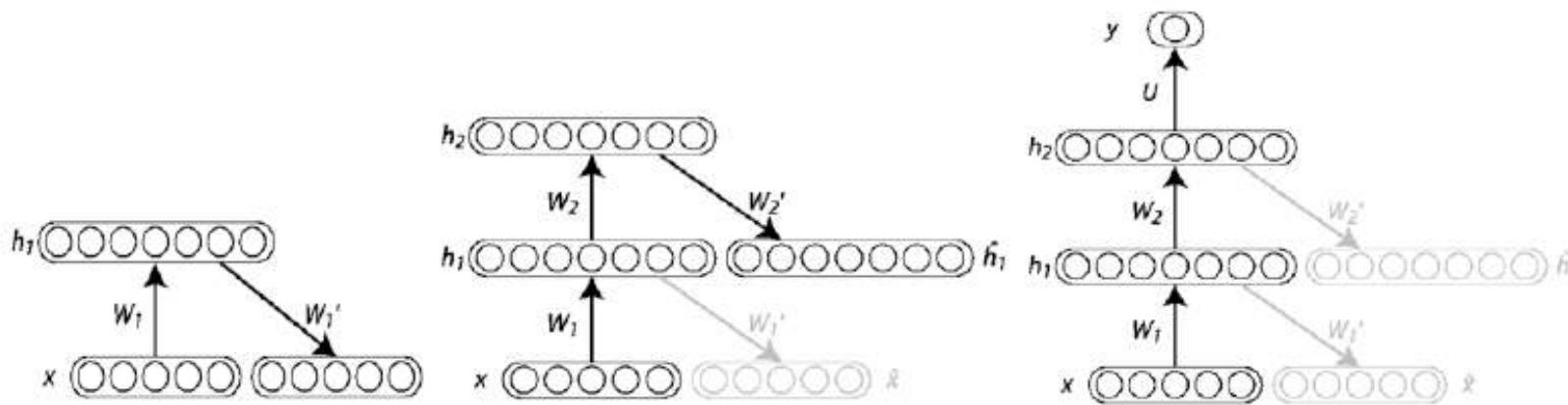
# conection between Linear auto-encoders and PCA

$d_h < d$  (bottleneck, undercomplete representation):

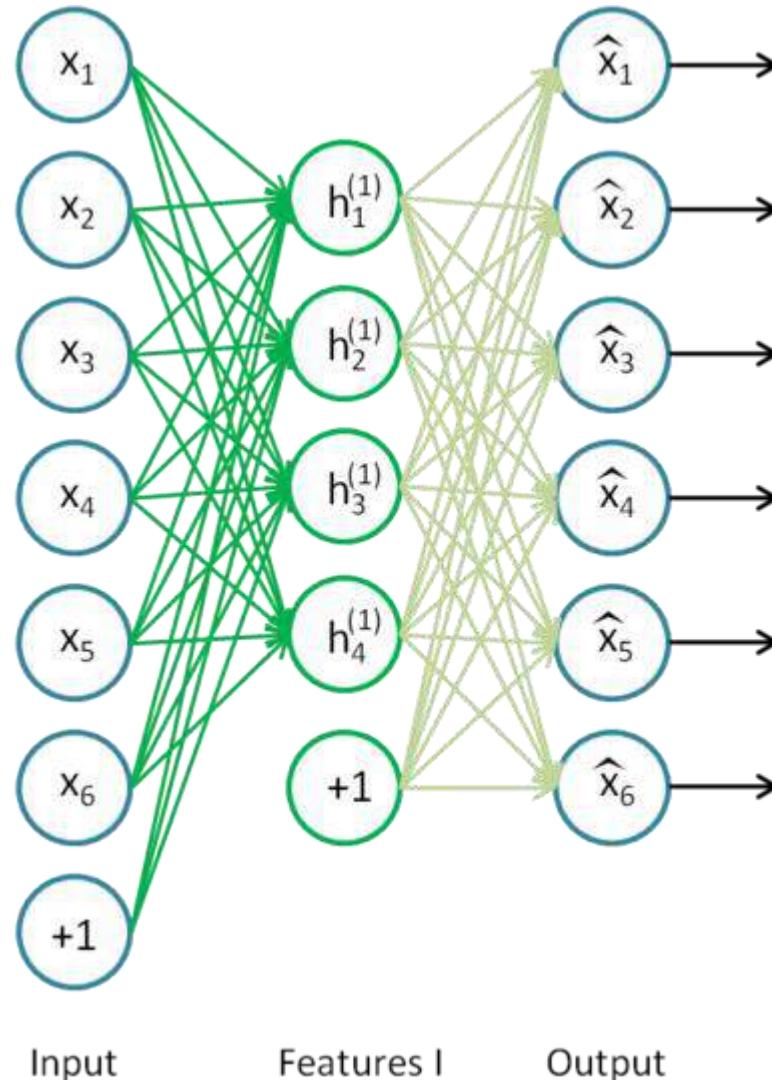
- With linear neurons and squared loss
  - ➡ autoencoder learns same **suspace** as PCA
- Also true with a single sigmoidal hidden layer, if using linear output neurons with squared loss [Baldi & Hornik 89] and untied weights.
- Won't learn the exact same **basis** as PCA, but  $W$  will span the same **subspace**.

# Greedy Layer-Wise Pre-training with Auto-Encoders

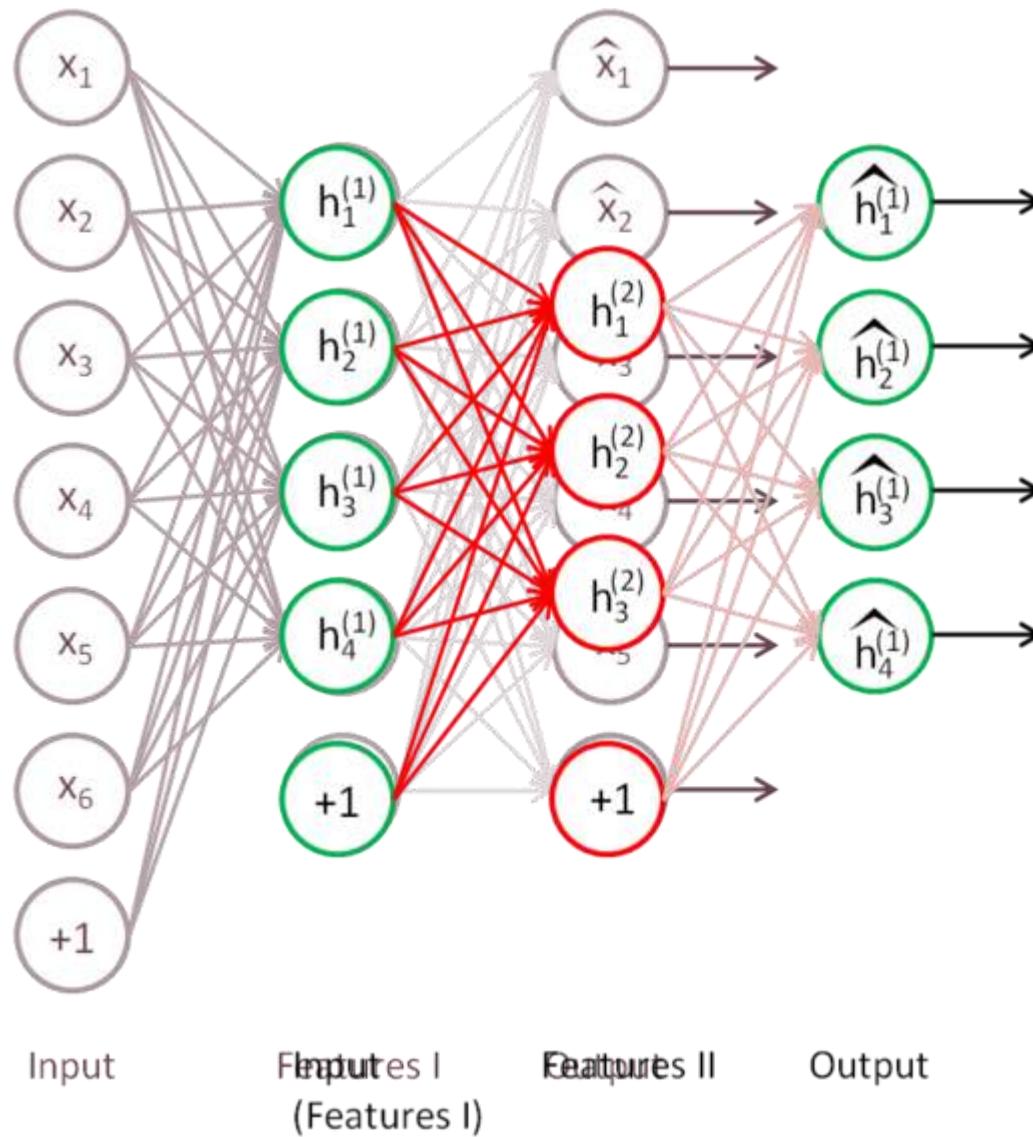
Stacking basic Auto-Encoders [Bengio et al. 2007]



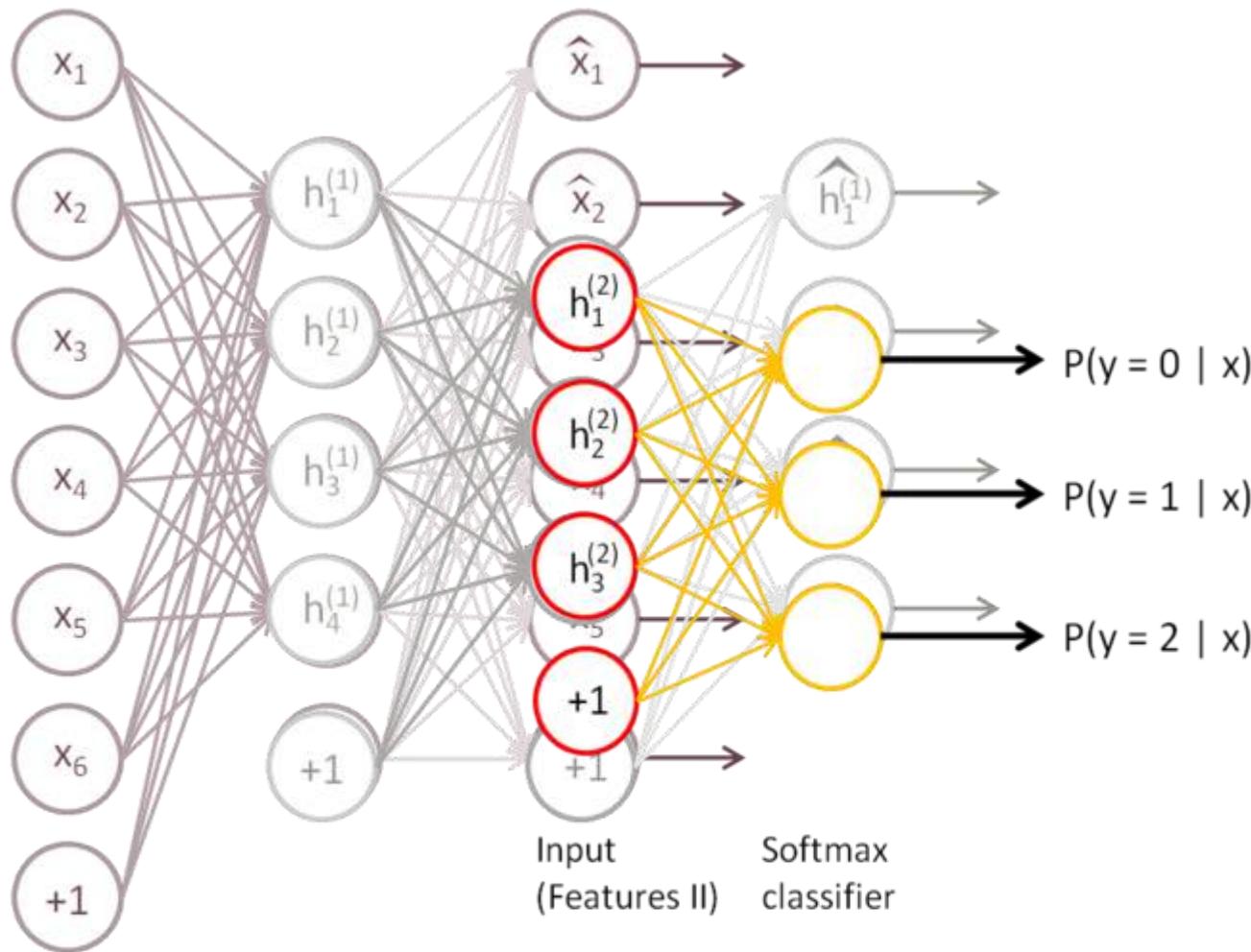
# Stacking autoencoders: learn the first layer



# Stacking autoencoders: learn the second layer



# Stacking autoencoders: Add, e.g., a softmax layer for mapping to output



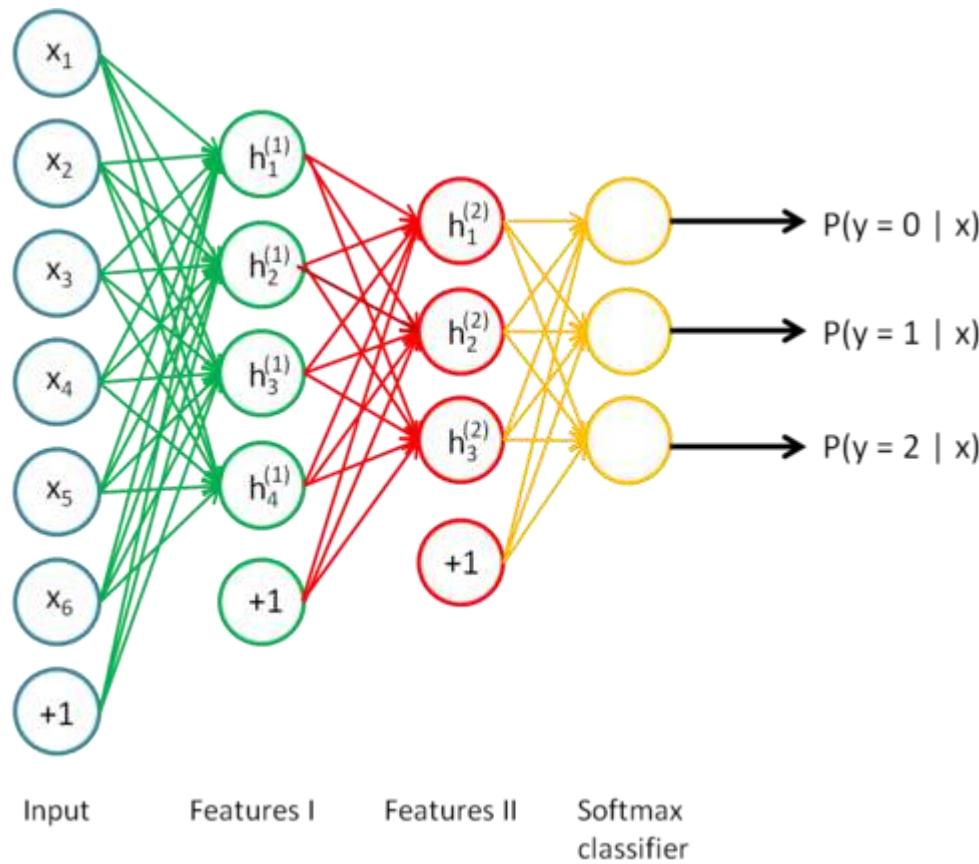
Input

Features I

Outputs II

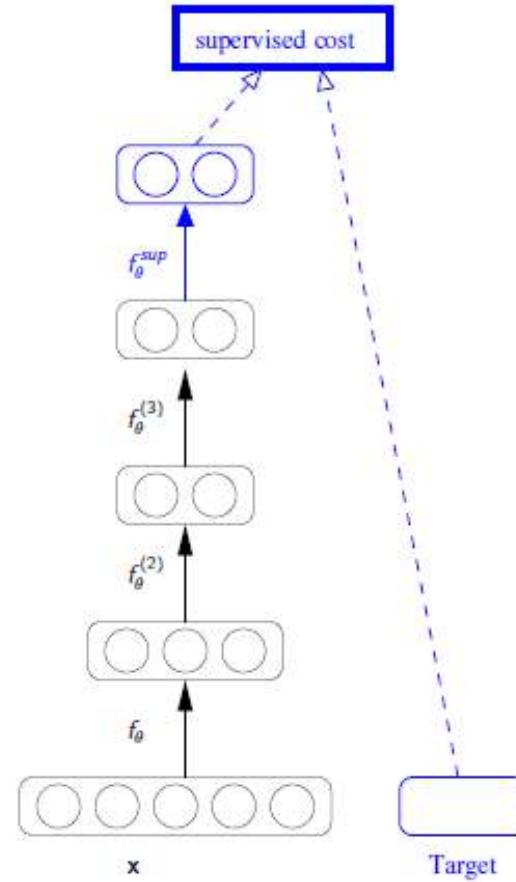
Output

# Stacking autoencoders: Overall



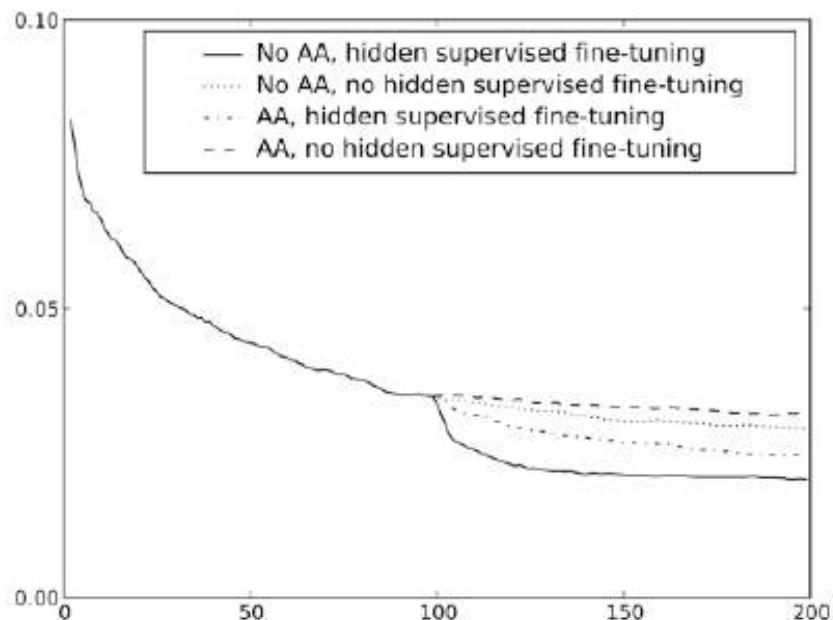
# Supervised fine-tuning

- Initial deep mapping was learnt in an **unsupervised** way.
- **initialization** for a **supervised** task.
- Output layer** gets added.
- Global fine tuning by gradient descent on **supervised criterion**.



# Supervised Fine-Tuning is Important

- Greedy layer-wise unsupervised pre-training phase with RBMs or auto-encoders on MNIST
- Supervised phase with or without unsupervised updates, with or without fine-tuning of hidden layers



## Classification performance on benchmarks:

- Pre-training basic auto-encoder stack **better** than no pre-training
- Basic auto-encoder stack **almost** matched RBM stack...

# Basic auto-encoders not as good feature learners as RBMs...

## What's the problem?

- Traditional autoencoders were for **dimensionality reduction** ( $d_h < d_x$ )
- Deep learning success seems to depend on ability to learn **overcomplete representations** ( $d_h > d_x$ )
- Overcomplete basic autoencoder yields trivial useless solutions: **identity mapping!**
- Need for alternative **regularization/constraining**



26

# Making auto- encoders learn **over-complete** representations

That are not one-to-one mappings

# Wait, what do we mean by over-complete?

- Remember distributed representations?

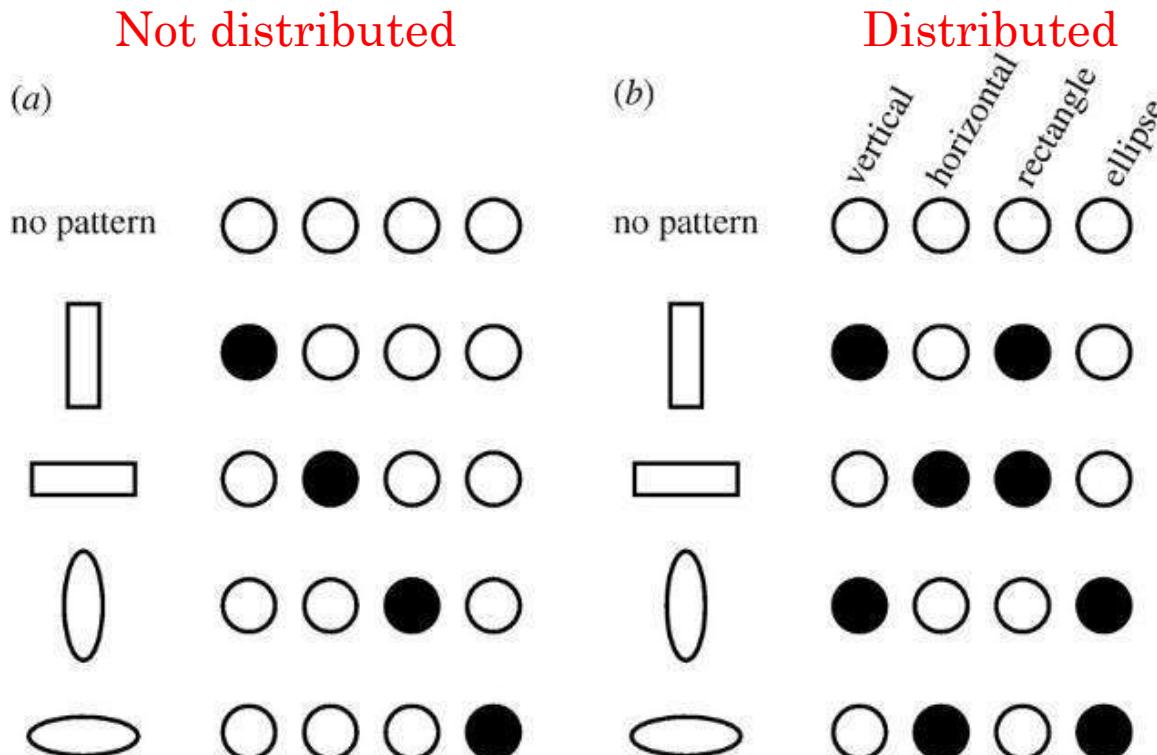
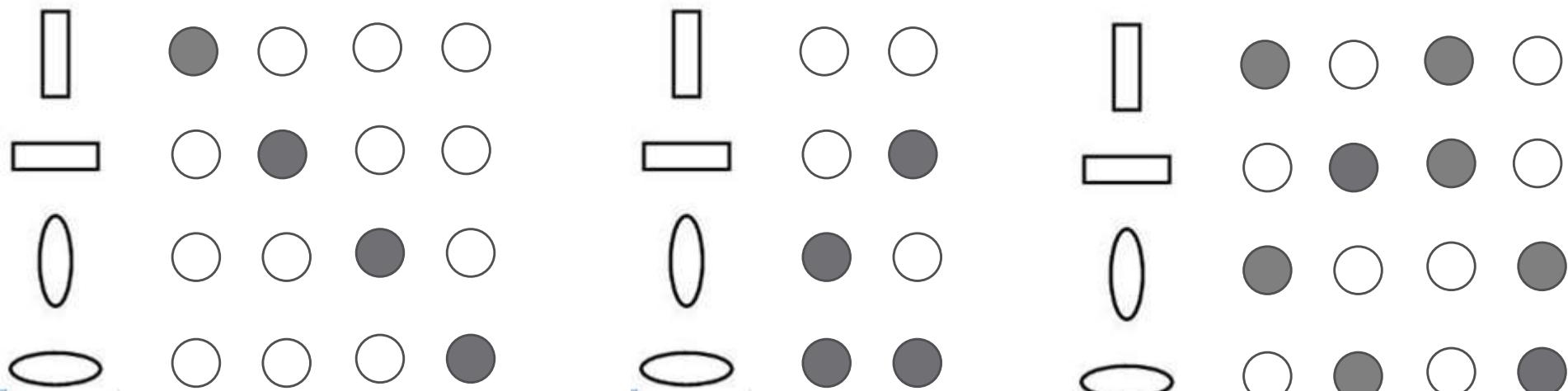


Figure Credit: Moontae Lee

# Distributed vs. undercomplete vs. overcomplete representations

- Four categories could also be represented by two neurons:



Not Distributed  
(over complete)

Distributed  
(Under complete)

Distributed  
(Over complete)

# Over-complete = sparse (in distributed representations)

- Why sparsity?
    1. Because our brain relies on sparse coding.
      - Why does it do so?
        - a. Because it is adapted to an environment which is composed of and can be sensed through the combination of primitive items/entities.
        - b. *"Sparse coding may be a general strategy of neural systems to augment memory capacity. To adapt to their environments, animals must learn which stimuli are associated with rewards or punishments and distinguish these reinforced stimuli from similar but irrelevant ones. Such task requires implementing stimulus-specific associative memories in which only a few neurons out of a population respond to any given stimulus and each neuron responds to only a few stimuli out of all possible stimuli."*
      - c. Theoretically, it has shown that it increases capacity of memory.
- Wikipedia

# Over-complete = sparse (in distributed representations)

- Why sparsity?
- 2. Because of information theoretical aspects:
  - Sparse codes have lower entropy compared to non-sparse ones.
- 3. It is easier for the consecutive layers to learn from sparse codes, compared to non-sparse ones.

Olshausen & Field,  
“Sparse coding with  
an overcomplete  
basis set: A  
strategy employed  
by V1?”, 1997

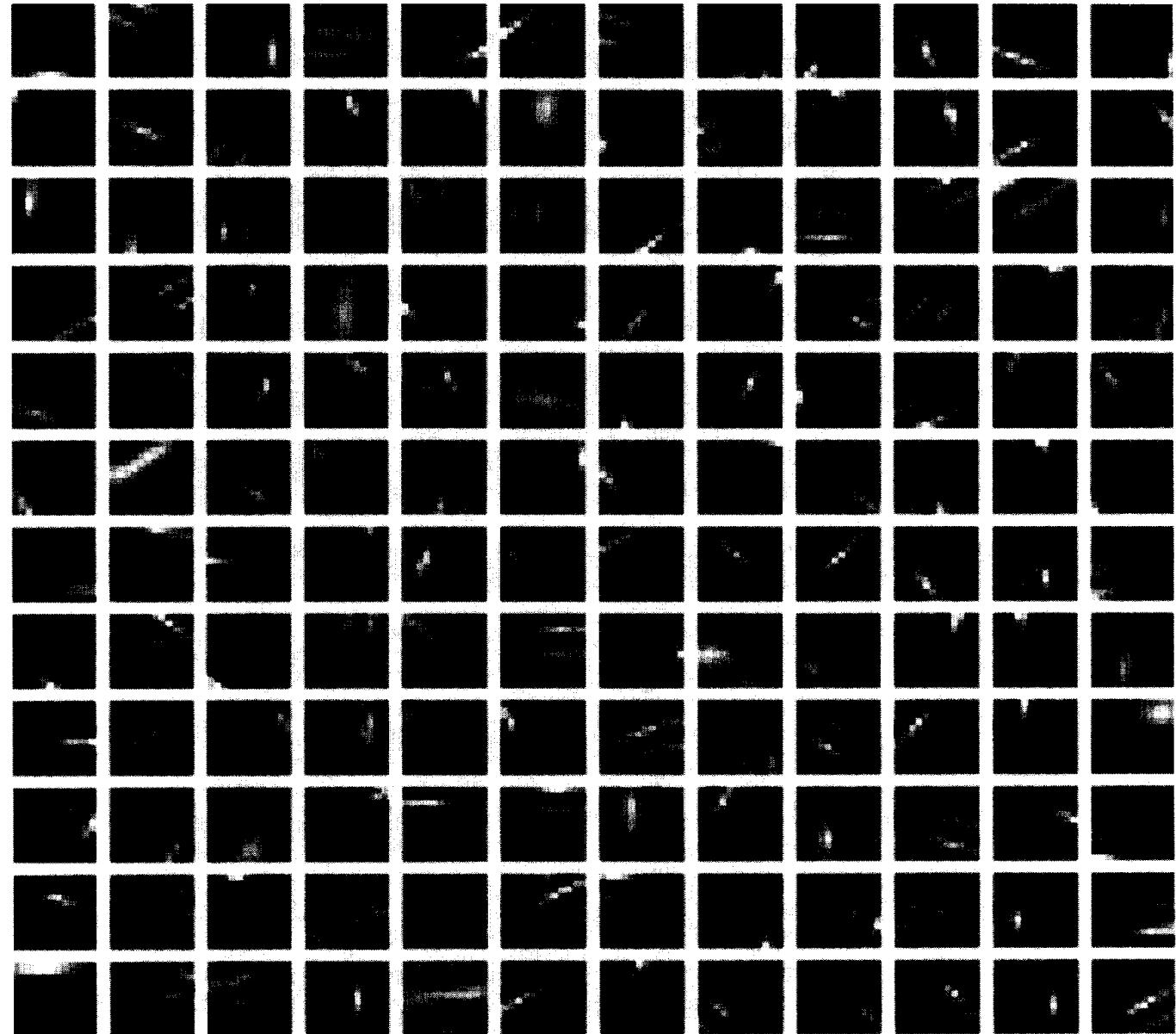


FIGURE 7. The set of 144 basis functions learned by the sparse coding algorithm. The basis functions are totally overlapping (i.e., the entire set codes for the same image patch). All have been normalized to fill the grey scale, but with zero always represented by the same grey level.

# Mechanisms for enforcing over-completeness

- Use stochastic gradient descent
- Add sparsity constraint
  - Into the loss function (sparse autoencoder)
  - Or, in a hard manner ( $k$ -sparse autoencoder)
- Add stochasticity / randomness
  - Add noise: Denoising Autoencoders, Contraction Autoencoders
  - Restricted Boltzmann Machines

---

## Why Regularized Auto-Encoders learn Sparse Representation?

---

Devansh Arpit  
Yingbo Zhou  
Hung Q. Ngo  
Venu Govindaraju

DEVANSHA@BUFFALO.EDU  
YINGBOZH@BUFFALO.EDU  
HUNGNGO@BUFFALO.EDU  
GOVIND@BUFFALO.EDU

# Auto-encoders with SGD

# Simple neural network

- Input:  $\mathbf{x} \in R^{\textcolor{red}{n}}$
- Hidden layer:  $\mathbf{h} \in R^{\textcolor{blue}{m}}$

$$\mathbf{h} = f_1(W_1 \mathbf{x})$$

- Output layer:  $\mathbf{y} \in R^{\textcolor{red}{n}}$

$$\mathbf{y} = f_2(W_2 f_1(W_1 \mathbf{x}))$$

- Squared-error loss:

$$L = \frac{1}{2} \sum_{d \in D} \| \mathbf{x}_d - \mathbf{y}_d \|^2$$

- For training, use SGD.
- You may try different activation functions for  $f_1$  and  $f_2$ .

# Sparse Autoencoders

# Sparse autoencoders

- Input:  $\mathbf{x} \in R^n$
- Hidden layer:  $\mathbf{h} \in R^m$

$$\mathbf{h} = f_1(W_1\mathbf{x})$$

- Output layer:  $\mathbf{y} \in R^n$

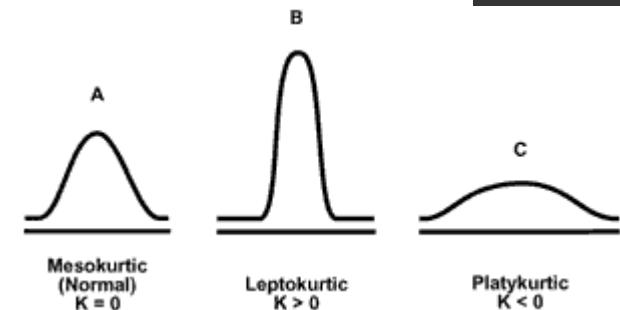
$$\mathbf{y} = f_2(W_2f_1(W_1\mathbf{x}))$$

Over-completeness and sparsity:

- Require
  - $m > n$ , and
  - Hidden neurons to produce only little activation for any input  
→ i.e., sparsity.
- How to enforce sparsity?

# Enforcing sparsity: alternatives

- How?
- Solution 1:  $\lambda |w|$ 
  - We have seen before that this enforces sparsity.
  - However, this is not strong enough.
- Solution 2
  - Limit on the amount of average total activation for a neuron throughout training!
- Solution 3
  - Kurtosis:  $\frac{\mu_4}{\sigma^4} = \frac{E[(X-\mu)^4]}{(E[(X-\mu)^2])^2}$
  - Calculated over the activations of the whole network.
  - High kurtosis  $\rightarrow$  sparse activations.
  - “Kurtosis has only been studied for response distributions of model neurons where negative responses are allowed. It is unclear whether kurtosis is actually a sensible measure for realistic, non-negative response distributions.” -  
[http://www.scholarpedia.org/article/Sparse\\_coding](http://www.scholarpedia.org/article/Sparse_coding)
- And many many other ways...



# Enforcing sparsity: a popular choice

- Limit the amount of total activation for a neuron throughout training!
- Use  $\rho_i$  to denote the activation of neuron  $x$  on input  $i$ .  
The average activation of **the neuron over the training set**:

$$\hat{\rho}_i = \frac{1}{m} \sum_i^m \rho_i$$

- Now, to enforce sparsity, we limit to  $\hat{\rho}_i = \rho_0$ .
- $\rho_0$ : A small value.
  - Yet another hyperparameter which may be tuned.
  - typical value: 0.05.
- The neuron must be inactive most of the time to keep its activations under the limit.

# Enforcing sparsity

$$\hat{\rho}_i = \frac{1}{m} \sum_i^m \rho_i$$

- How to limit  $\hat{\rho}_i = \rho_0$ ? How do we add integrate this as a penalty term into the loss function?
  - $\rho_0$  is called the sparsity parameter.
- Use Kullback-Leibler divergence:

$$\sum_i KL(\rho_0 \parallel \hat{\rho}_i)$$

Or, equivalently as (since this is between two Bernoulli variables with mean  $\rho_0$  and  $\hat{\rho}_i$ ):

$$\sum_i \rho_0 \log \frac{\rho_0}{\hat{\rho}_i} + (1 - \rho_0) \log \frac{1 - \rho_0}{1 - \hat{\rho}_i}$$

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}.$$

# Backpropagation and training

$$S = \beta \sum_i \rho_0 \log \frac{\rho_0}{\hat{\rho}_i} + (1 - \rho_0) \log \frac{1 - \rho_0}{1 - \hat{\rho}_i}$$

$$\frac{dS}{d\rho_i} = \beta \left( -\rho_0 \frac{1}{\hat{\rho}_i \ln 10} + (1 - \rho_0) \frac{1}{(1 - \hat{\rho}_i) \ln 10} \right)$$

- If you use **ln** in KL:

$$\frac{dS}{d\rho_i} = \beta \left( -\frac{\rho_0}{\hat{\rho}_i} + \frac{1 - \rho_0}{1 - \hat{\rho}_i} \right)$$

- So, if we integrate into the original error term:

$$\delta_h = o_h(1 - o_h) \cdot \left( \left( \sum_k w_{kh} \delta_k \right) + \beta \left( -\frac{\rho_0}{\hat{\rho}_h} + \frac{1 - \rho_0}{1 - \hat{\rho}_h} \right) \right)$$

- Need to change  $\textcolor{red}{o_h(1 - o_h)}$  if you use a different activation function.

## Reminder

-For each hidden unit  $h$ , calculate its error term  $\delta_h$ :

$$\delta_h = o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh} \delta_k$$

-Update every weight  $w_{ji}$

$$w_{ji} = w_{ji} + \eta \delta_j x_{ji}$$

# Backpropagation and training

$$S = \beta \sum_i \rho_0 \log \frac{\rho_0}{\hat{\rho}_i} + (1 - \rho_0) \log \frac{1 - \rho_0}{1 - \hat{\rho}_i}$$

- Do you see a problem here?
- $\hat{\rho}_i$  should be calculated over the training set.
- In other words, we need to go through the whole dataset (or batch) once to calculate  $\hat{\rho}_i$ .

# Loss & decoders & encoders

- Be careful about the range of your activations and the range of the output
- Real-valued input:
  - Encoder: use sigmoid
  - Decoder: no need for non-linearity.
  - Loss: Squared-error Loss
  - Vincent et al. (2010):

For real-valued  $\mathbf{x}$ , that is,  $\mathbf{x} \in \mathbb{R}^d$ :  $X|\mathbf{z} \sim \mathcal{N}(\mathbf{z}, \sigma^2 \mathbf{I})$ , that is,  $X_j|\mathbf{z} \sim \mathcal{N}(z_j, \sigma^2)$ .

This yields  $L(\mathbf{x}, \mathbf{z}) = L_2(\mathbf{x}, \mathbf{z}) = C(\sigma^2) \|\mathbf{x} - \mathbf{z}\|^2$  where  $C(\sigma^2)$  denotes a constant that depends only on  $\sigma^2$  and that can be ignored for the optimization.

- Binary-valued input:
  - Encoder: use sigmoid.
  - Decoder: use sigmoid.
  - Loss: use cross-entropy loss:

$$-\sum_j [\mathbf{x}_j \log \mathbf{z}_j + (1 - \mathbf{x}_j) \log(1 - \mathbf{z}_j)] :$$

# Loss & decoders & encoders

- Kullback-Leibler divergence assumes that the variables are in the range [0,1].
  - I.e., you are bound to use sigmoid for the hidden layer if you use KL to limit the activations of hidden units.

# k-Sparse Autoencoder

- Note that it doesn't have an activation function!
- Non-linearity comes from k-selection.

### $k$ -Sparse Autoencoders:

#### Training:

1) Perform the feedforward phase and compute

$$\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$

2) Find the  $k$  largest activations of  $\mathbf{z}$  and set the rest to zero.

$$\mathbf{z}_{(\Gamma)^c} = 0 \quad \text{where } \Gamma = \text{supp}_k(\mathbf{z})$$

3) Compute the output and the error using the sparsified  $\mathbf{z}$ .

$$\hat{\mathbf{x}} = \mathbf{W}\mathbf{z} + \mathbf{b}'$$

$$E = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2$$

3) Backpropagate the error through the  $k$  largest activations defined by  $\Gamma$  and iterate.

#### Sparse Encoding:

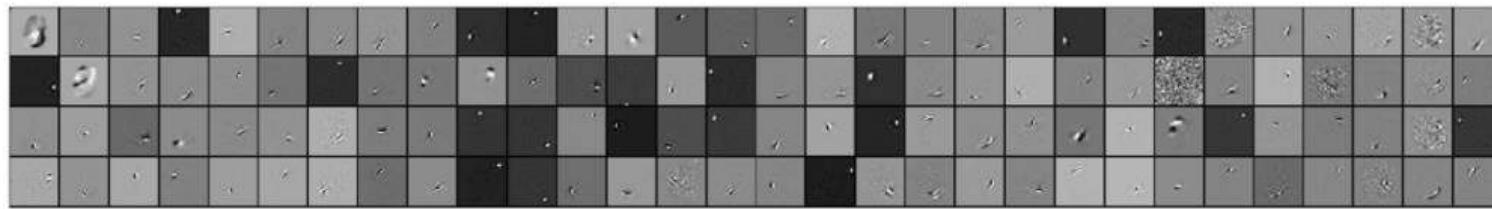
Compute the features  $\mathbf{h} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$ . Find its  $\alpha k$  largest activations and set the rest to zero.

$$\mathbf{h}_{(\Gamma)^c} = 0 \quad \text{where } \Gamma = \text{supp}_{\alpha k}(\mathbf{h})$$

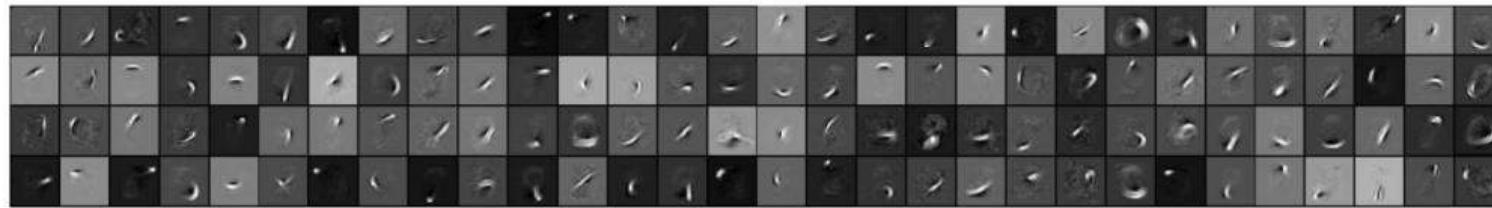
---

## $k$ -Sparse Autoencoders

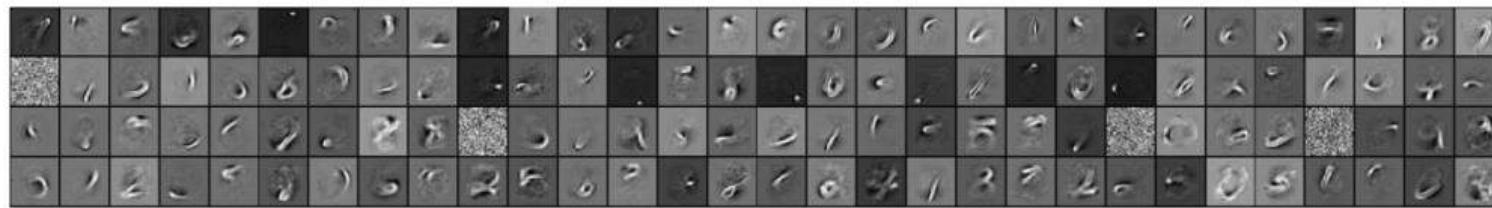
---



(a)  $k = 70$



(b)  $k = 40$



(c)  $k = 25$



(d)  $k = 10$

# Denoising Auto-encoders (DAE)

Journal of Machine Learning Research 11 (2010) 3371-3408

Submitted 5/10; Published 12/10

## Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion

**Pascal Vincent**

*Département d'informatique et de recherche opérationnelle  
Université de Montréal  
2920, chemin de la Tour  
Montréal, Québec, H3T 1J8, Canada*

[PASCAL.VINCENT@UMONTREAL.CA](mailto:PASCAL.VINCENT@UMONTREAL.CA)

**Hugo Larochelle**

*Department of Computer Science  
University of Toronto  
10 King's College Road  
Toronto, Ontario, M5S 3G4, Canada*

[LAROCHEH@CS.TORONTO.EDU](mailto:LAROCHEH@CS.TORONTO.EDU)

**Isabelle Lajoie**

**Yoshua Bengio**

**Pierre-Antoine Manzagol**

*Département d'informatique et de recherche opérationnelle  
Université de Montréal  
2920, chemin de la Tour  
Montréal, Québec, H3T 1J8, Canada*

[ISABELLE.LAJOIE.1@UMONTREAL.CA](mailto:ISABELLE.LAJOIE.1@UMONTREAL.CA)

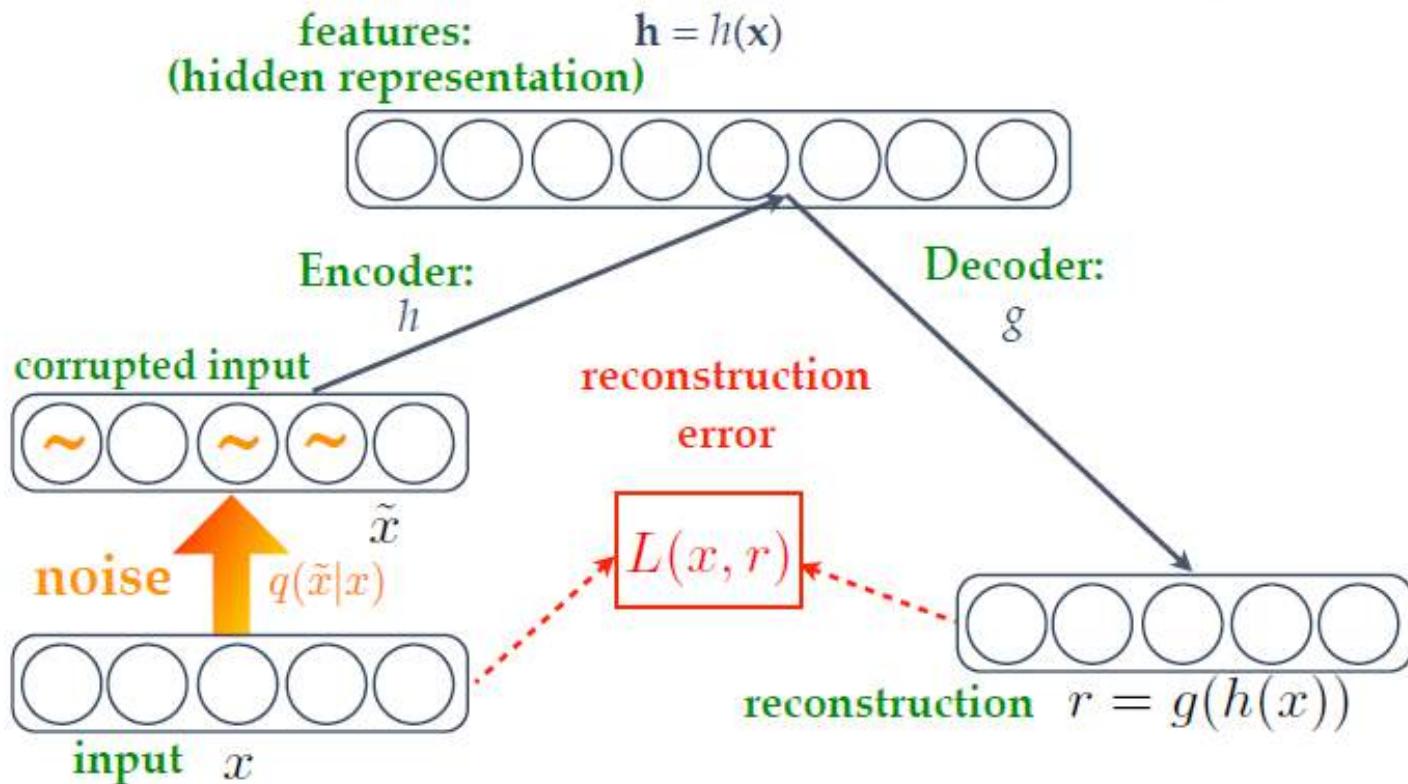
[YOSHUA.BENGIO@UMONTREAL.CA](mailto:YOSHUA.BENGIO@UMONTREAL.CA)

[PIERRE-ANTOINE.MANZAGOL@UMONTREAL.CA](mailto:PIERRE-ANTOINE.MANZAGOL@UMONTREAL.CA)

# Denoising Auto-encoders

- Simple idea:
  - randomly corrupt some of the inputs (as many as half of them) – e.g., set them to zero.
  - Train the autoencoder to *reconstruct the input from a corrupted version of it.*
  - The auto-encoder is to *predict the corrupted (i.e. missing) values from the uncorrupted values.*
  - This requires capturing the joint distribution between a set of variables
- A stochastic version of the auto-encoder.

# Denoising auto-encoder (DAE)



- learns robust & useful features
- easier to train than RBM features
- yield similar or better classification performance (as deep net pre-training)

Minimize:

$$\mathcal{J}_{\text{DAE}}(\theta) = \sum_{x \in D} \mathbb{E}_{q(\tilde{x}|x)} [L(x, g(h(\tilde{x})))]$$

# Denoising auto-encoder (DAE)

- + Autoencoder training minimizes:

$$\mathcal{J}_{\text{AE}}(\theta) = \sum_{x \in D} L(x, g(h(\tilde{x}))$$

- + Denoising autoencoder training minimizes

$$\mathcal{J}_{\text{DAE}}(\theta) = \sum_{x \in D} \mathbb{E}_{q(\tilde{x}|x)} [L(x, g(h(\tilde{x})))]$$

Cannot compute expectation exactly

⇒ use stochastic gradient descent,  
**sampling corrupted inputs  $\tilde{x}|x$**

## Possible corruptions q:

- zeroing pixels at random  
(now called «dropout» noise)
- additive Gaussian noise
- salt-and-pepper noise
- ...

29

# Loss in DAE

- You may give extra emphasis on “corrupted” dimensions:

$$L_{2,\alpha}(\mathbf{x}, \mathbf{z}) = \alpha \left( \sum_{j \in \mathcal{I}(\tilde{\mathbf{x}})} (\mathbf{x}_j - \mathbf{z}_j)^2 \right) + \beta \left( \sum_{j \notin \mathcal{I}(\tilde{\mathbf{x}})} (\mathbf{x}_j - \mathbf{z}_j)^2 \right),$$

where  $\mathcal{I}(\tilde{\mathbf{x}})$  denotes the indexes of the components of  $\mathbf{x}$  that were corrupted.

Or, in cross-entropy-based loss:

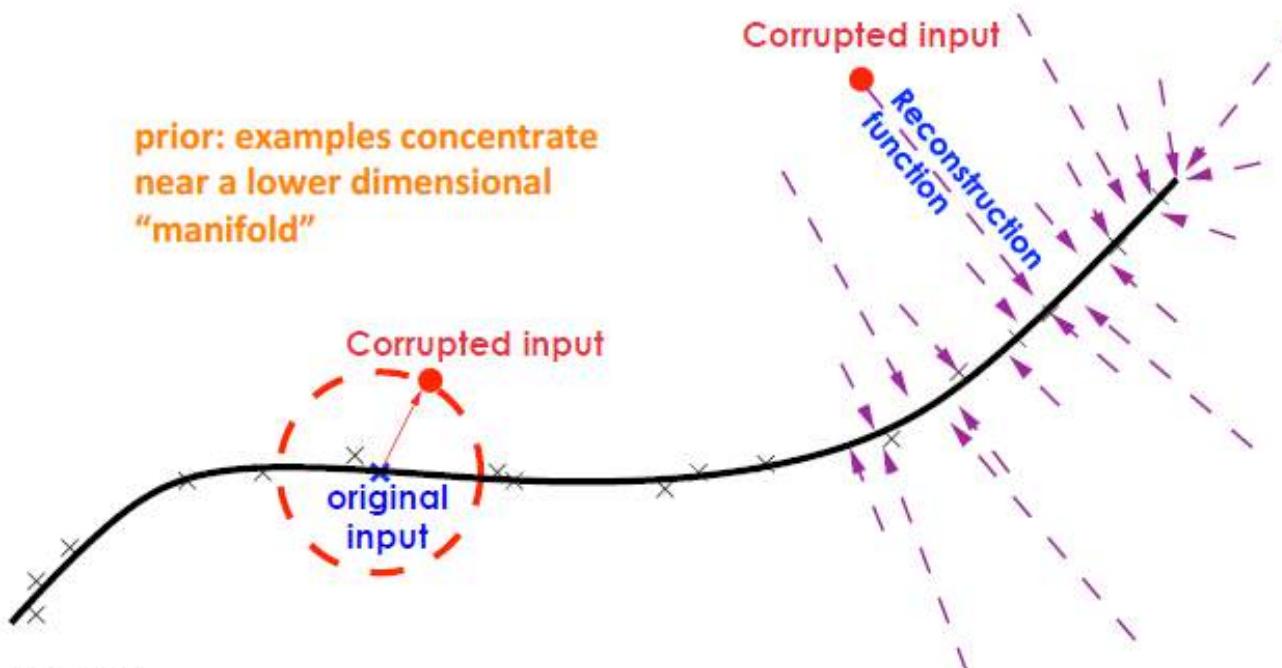
$$\begin{aligned} L_{\text{H},\alpha}(\mathbf{x}, \mathbf{z}) &= \alpha \left( - \sum_{j \in \mathcal{I}(\tilde{\mathbf{x}})} [\mathbf{x}_j \log \mathbf{z}_j + (1 - \mathbf{x}_j) \log(1 - \mathbf{z}_j)] \right) \\ &\quad + \beta \left( - \sum_{j \notin \mathcal{I}(\tilde{\mathbf{x}})} [\mathbf{x}_j \log \mathbf{z}_j + (1 - \mathbf{x}_j) \log(1 - \mathbf{z}_j)] \right). \end{aligned}$$

# Denoising Auto-encoders

- To undo the effect of a corruption induced by the noise, the **network needs to capture the statistical dependencies between the inputs.**
- This can be interpreted from many perspectives (see Vincent et al., 2008):
  - the manifold learning perspective,
  - stochastic operator perspective.

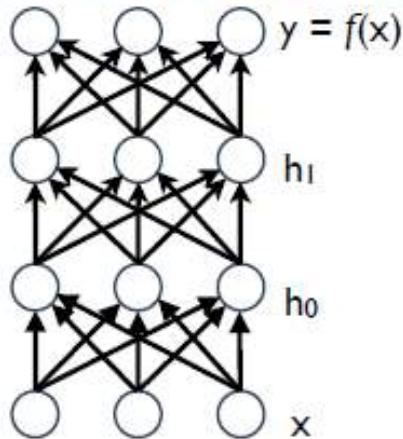
# Denoising auto-encoders: manifold interpretation

- DAE learns to «project back» corrupted input onto manifold.
- Representation  $h \approx$  location on the manifold



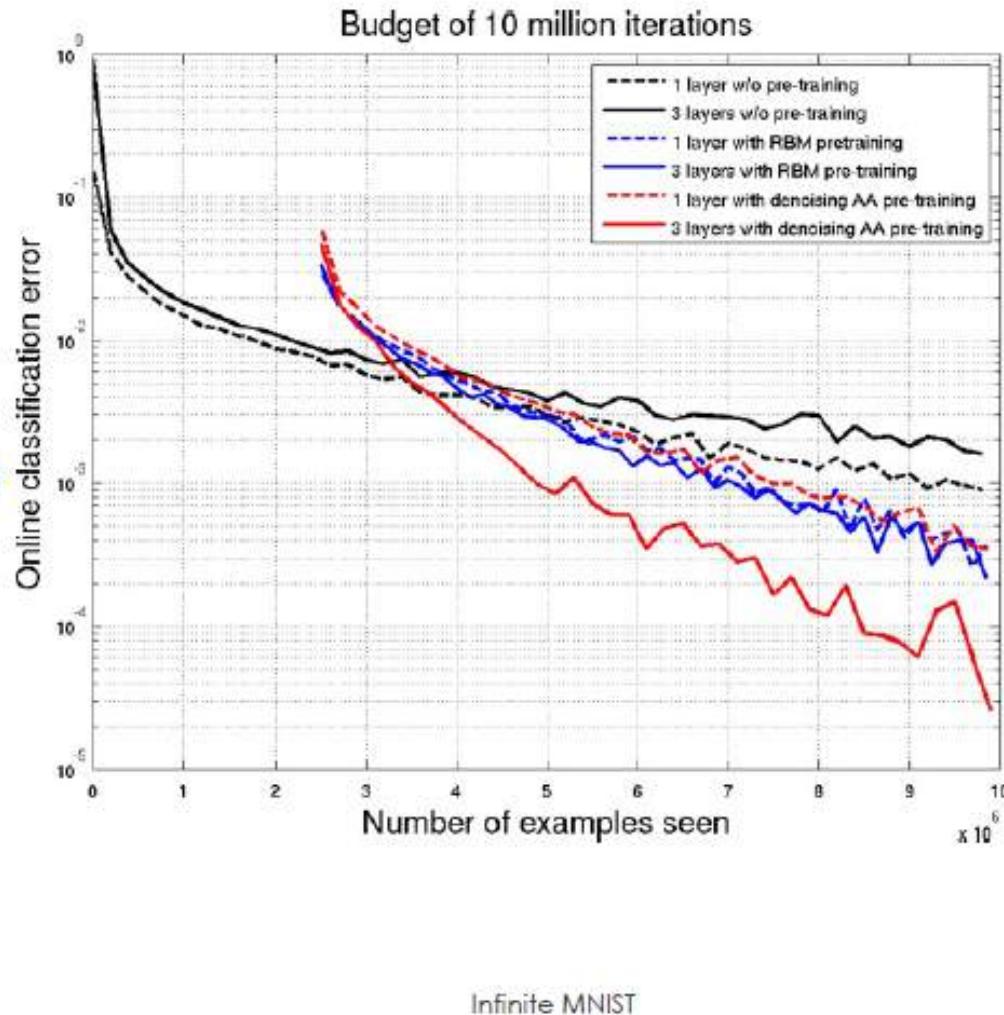
32

# Stacked Denoising Auto-Encoders (SDAE)



## Advantages over stacking RBMs

- No partition function, can measure training criterion
- Very flexible: encoder & decoder can use any parametrization (more layers...)
- Performs as well or better than stacking RBMs for unsupervised pre-training



# Types of corruption

- Gaussian Noise (additive, isotropic)
- Masking Noise
  - Set a randomly selected subset of input to zero for each sample (the fraction ratio is constant, a parameter)
- Salt-and-pepper Noise:
  - Set a randomly selected subset of input to **maximum** or **minimum** for each sample (the fraction ratio is constant, a parameter)

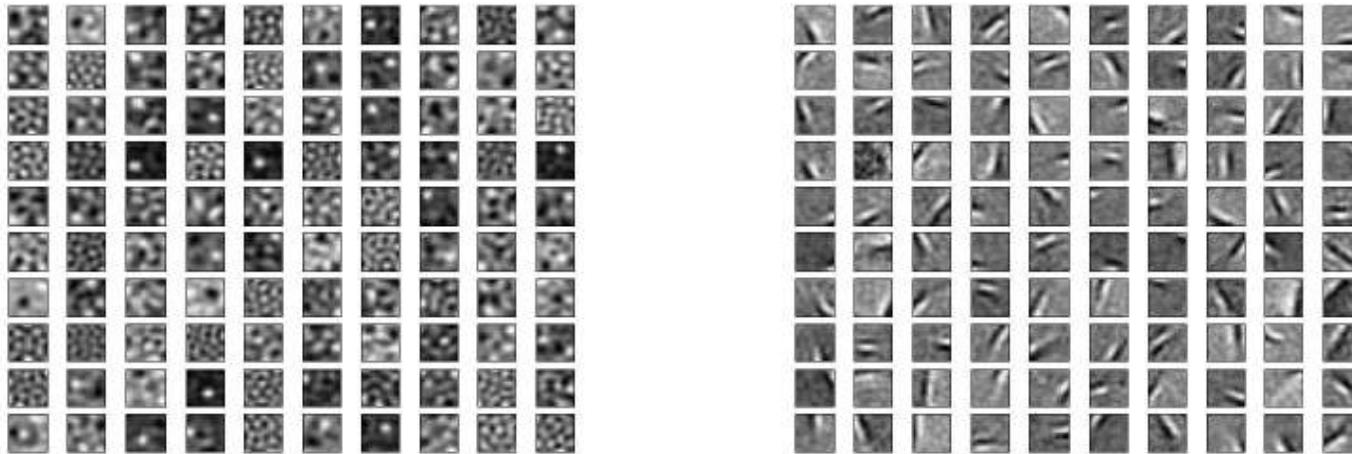


Figure 6: Weight decay vs. Gaussian noise. We show typical filters learnt from natural image patches in the over-complete case (200 hidden units). *Left*: regular autoencoder with weight decay. We tried a wide range of weight-decay values and learning rates: filters never appeared to capture a more interesting structure than what is shown here. Note that some local blob detectors are recovered compared to using no weight decay at all (Figure 5 right). *Right*: a denoising autoencoder with additive Gaussian noise ( $\sigma = 0.5$ ) learns Gabor-like local oriented edge detectors. Clearly the filters learnt are qualitatively very different in the two cases.

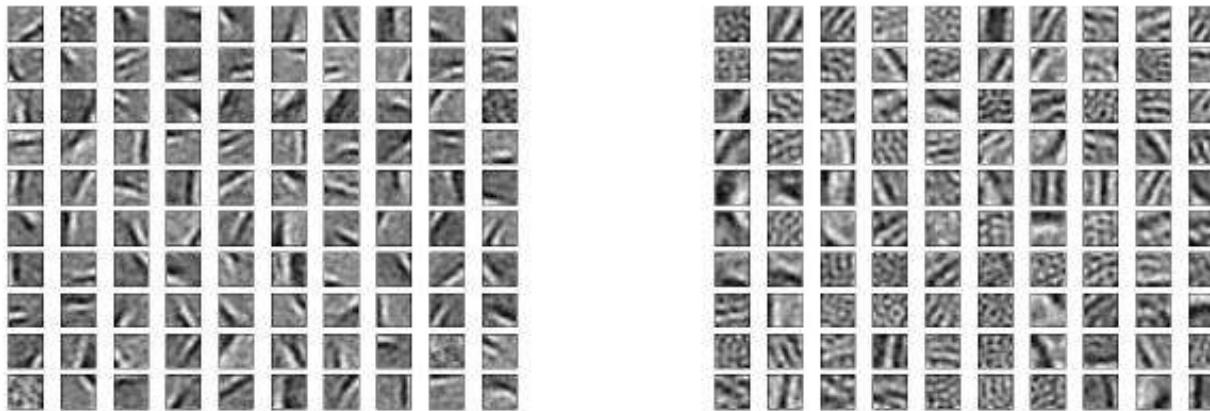


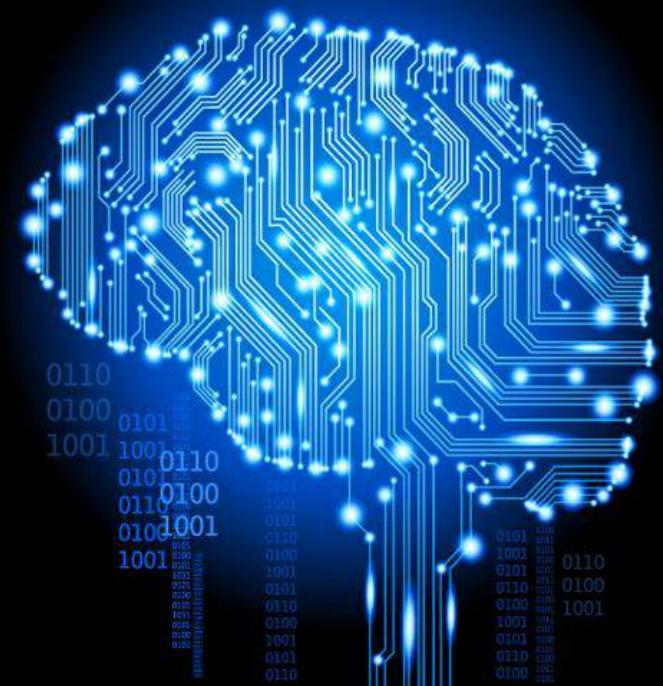
Figure 7: Filters obtained on natural image patches by denoising autoencoders using other noise types. *Left*: with 10% salt-and-pepper noise, we obtain oriented Gabor-like filters. They appear slightly less localized than when using Gaussian noise (contrast with Figure 6 right). *Right*: with 55% zero-masking noise we obtain filters that look like oriented gratings. For the three considered noise types, denoising training appears to learn filters that capture meaningful natural image statistics structure.

# Training DAE

- Training algorithm does not change
  - However, you may give different emphasis on the error of reconstruction of the corrupted input.
- SGD is a popular choice
- Sigmoid is a suitable choice unless you know what you are doing.

# CENG 783

## Special topics in Deep Learning



© AlchemyAPI

*Week 14  
AE, Generative Models*

Sinan Kalkan



# Today

- Finalize Autoencoders
  - Contractive AE
  - Convolutional AE
- Generative models
  - Hopfield Networks
  - Boltzmann Machines
  - Restricted Boltzmann Machines
  - Generative Adversarial Networks
  - Variational Autoencoders
- NOTES:
  - Final Exam date: 31 May, 17:00.
  - HW3 announced. Due date: 24 May, 23:55.
  - Project demos and papers due: 6 June.
  - **Extra lecture: 29 May 9:40-12:30**

# Contractive Auto-encoder

# Encouraging representation to be insensitive to corruption

- \* DAE encourages **reconstruction** to be insensitive to input corruption
- \* Alternative: encourage **representation** to be **insensitive**

$$\mathcal{J}_{\text{SCAE}}(\theta) = \sum_{x \in D} L(x, g(h(x))) + \lambda \mathbb{E}_{q(\tilde{x}|x)} [\|h(x) - h(\tilde{x})\|^2]$$

**Reconstruction error**      **stochastic regularization term**

# From stochastic to analytic penalty

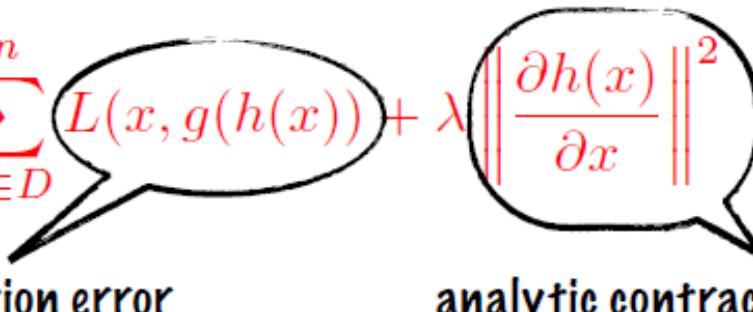
- \* SCAE stochastic regularization term:  $\mathbb{E}_{q(\tilde{x}|x)} [\|h(x) - h(\tilde{x})\|^2]$
- \* For small additive noise  $\tilde{x}|x = x + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I)$
- \* Taylor series expansion yields  $h(x + \epsilon) = h(x) + \frac{\partial h}{\partial x} \epsilon + \dots$
- \* It can be showed that

$$\underbrace{\mathbb{E}_{q(\tilde{x}|x)} [\|h(x) - h(\tilde{x})\|^2]}_{\text{stochastic (SCAE)}} \approx \underbrace{\sigma^2 \left\| \frac{\partial h}{\partial x}(x) \right\|_F^2}_{\text{analytic (CAE)}}$$

35

# Contractive Auto-Encoder (CAE)

(Rifai, Vincent, Muller, Glorot, Bengio, ICML 2011)

- \* Minimize  $\mathcal{J}_{\text{CAE}} = \sum_{x \in D}^n L(x, g(h(x))) + \lambda \left\| \frac{\partial h(x)}{\partial x} \right\|^2$   


Reconstruction error      analytic contractive term
- \* For training examples, encourages both:
  - small reconstruction error
  - representation insensitive to small variations around example

# Computational considerations

## CAE for a simple encoder layer

We defined  $\mathbf{h} = h(\mathbf{x}) = s(\underbrace{Wx + b}_a)$

Further suppose:  $s$  is an elementwise non-linearity  
 $s'$  its first derivative.

Let  $J(x) = \frac{\partial h}{\partial x}(x)$

$$J_j = s'(b + x^T W_j) W_j \quad \text{where } J_j \text{ and } W_j \text{ represent j}^{\text{th}} \text{ row}$$

CAE penalty is:  $\|J\|_F^2 = \sum_{j=1}^{d_h} s'(a_j)^2 \|W_j\|^2$

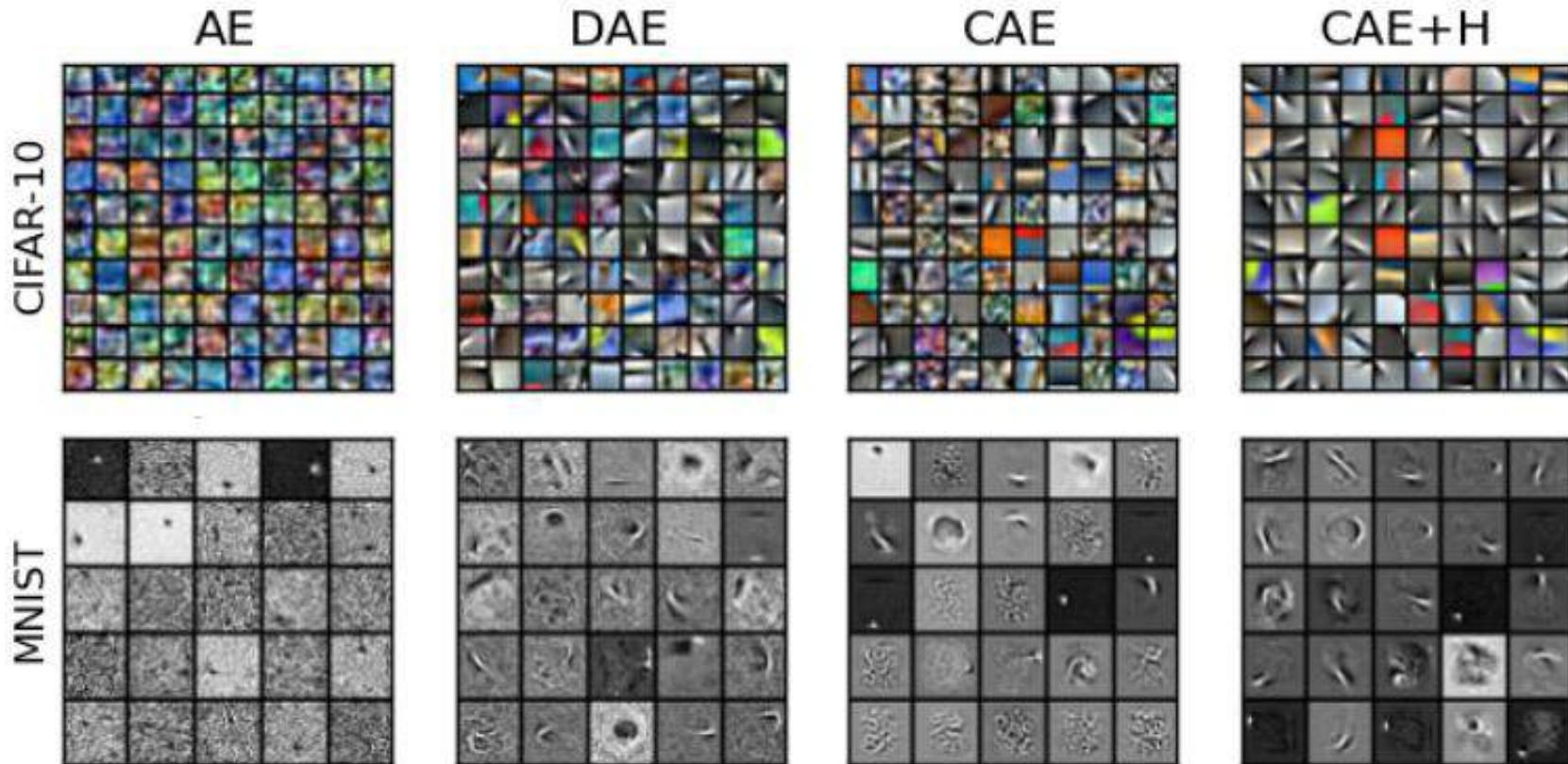
Compare to L<sub>2</sub> weight decay:  $\|W\|_F^2 = \sum_{j=1}^{d_h} \|W_j\|^2$

Same complexity:  
 $O(d_h d)$

Gradient backprop  
wrt parameters:  
 $O(d_h d)$

37

# Learned filters



39

11

(Pascal Vincent)

# Convolutional AE

Stacked Convolutional Auto-Encoders for  
Hierarchical Feature Extraction

Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber

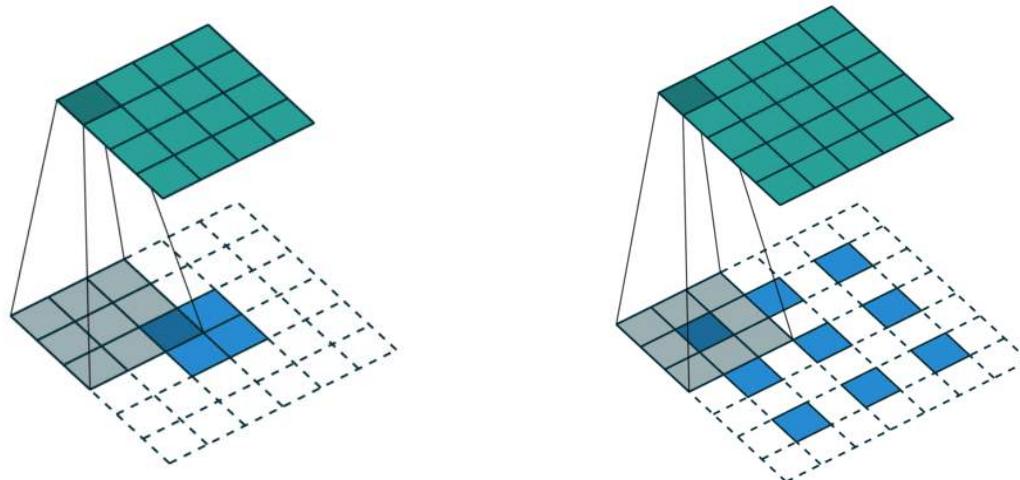
Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA)

Lugano, Switzerland

{jonathan, ueli, dan, juergen}@idsia.ch

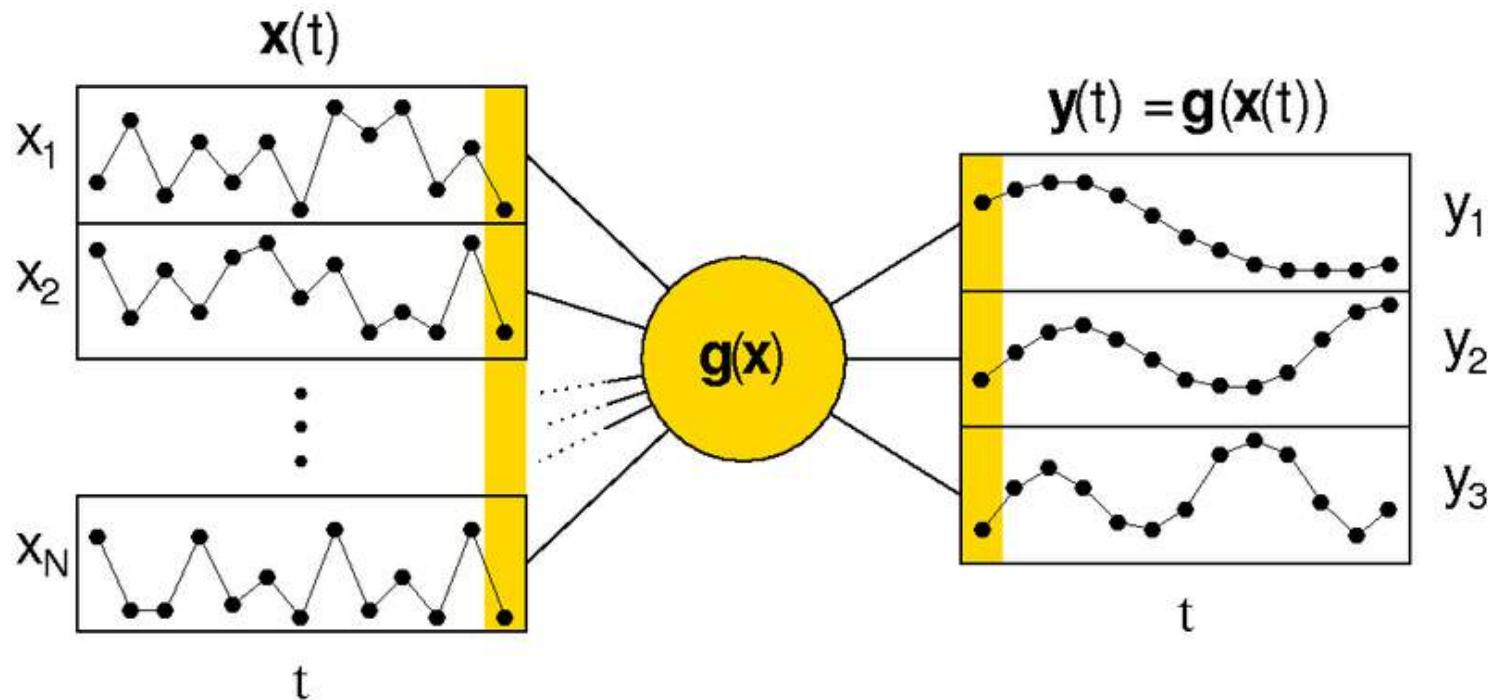
2011

- Encoder:
  - Standard convolutional layer
  - You may use pooling (e.g., max-pooling)
  - Pooling is shown to regularize the features in the encoder  
(Masci et al., 2011)
- Decoder:
  - Deconvolution
- Loss is MSE.



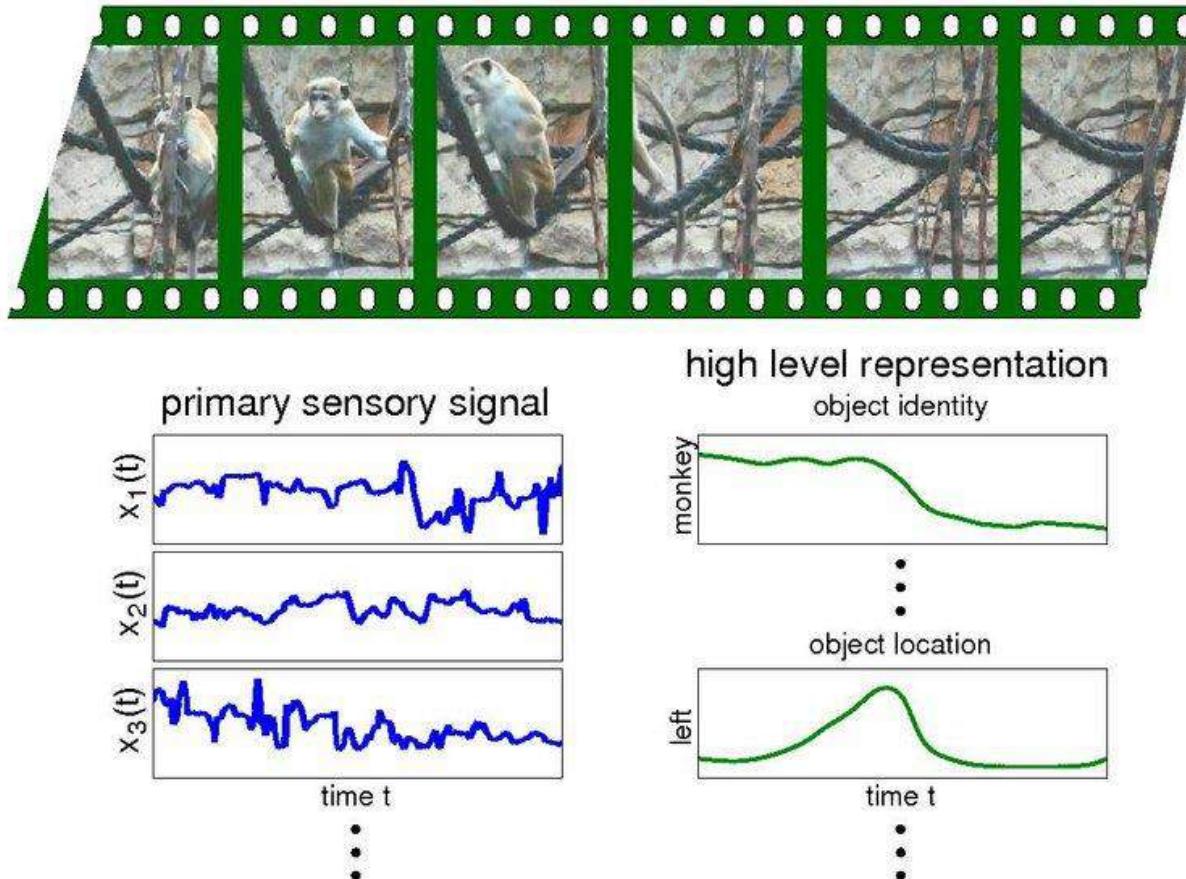
Principles other  
than ‘sparsity’?

# Slowness

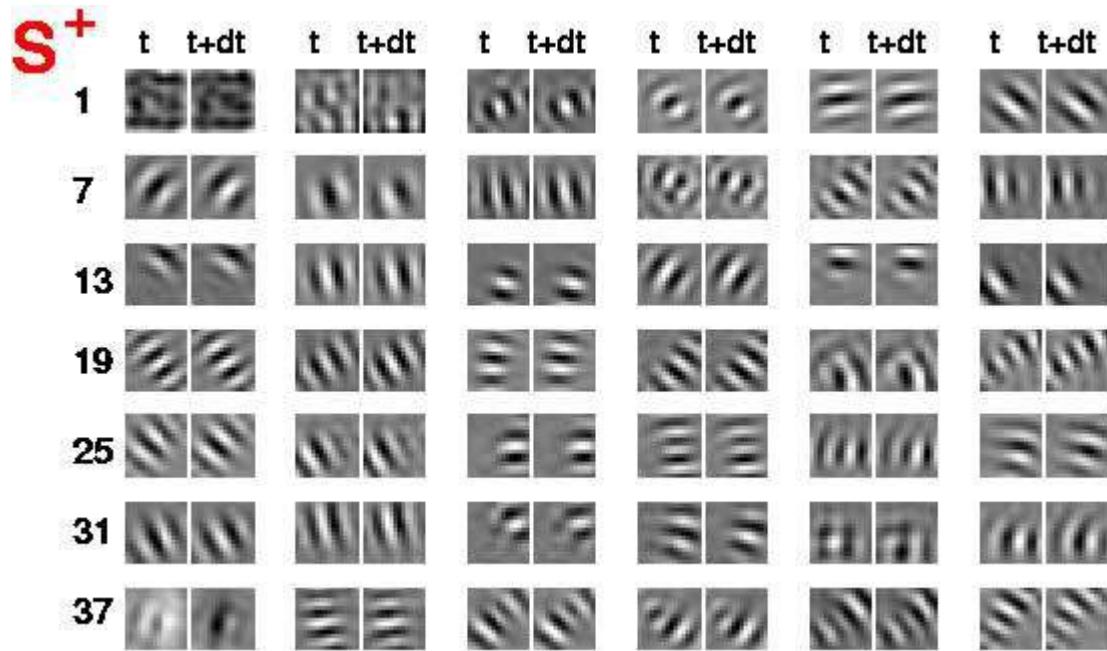


# Slow Feature Analysis (SFA)

from Wiskott et al.



# Slow Feature Analysis (SFA)



Optimal stimuli for the slowest components  
extracted from natural image sequences.

# Visualizing the layers

# Visualizing the layers

- Question: What is the input that activates a hidden unit  $h_i$  **most**?

- i.e., we are after  $\mathbf{x}^*$ :

$$\mathbf{x}^* = \arg \max_{\mathbf{x} \text{ s.t. } \|\mathbf{x}\|=\rho} h_i(W, \mathbf{x})$$

- For the first layer:

$$x_j = \frac{w_{ij}}{\sqrt{\sum_k (w_{ik})^2}}$$

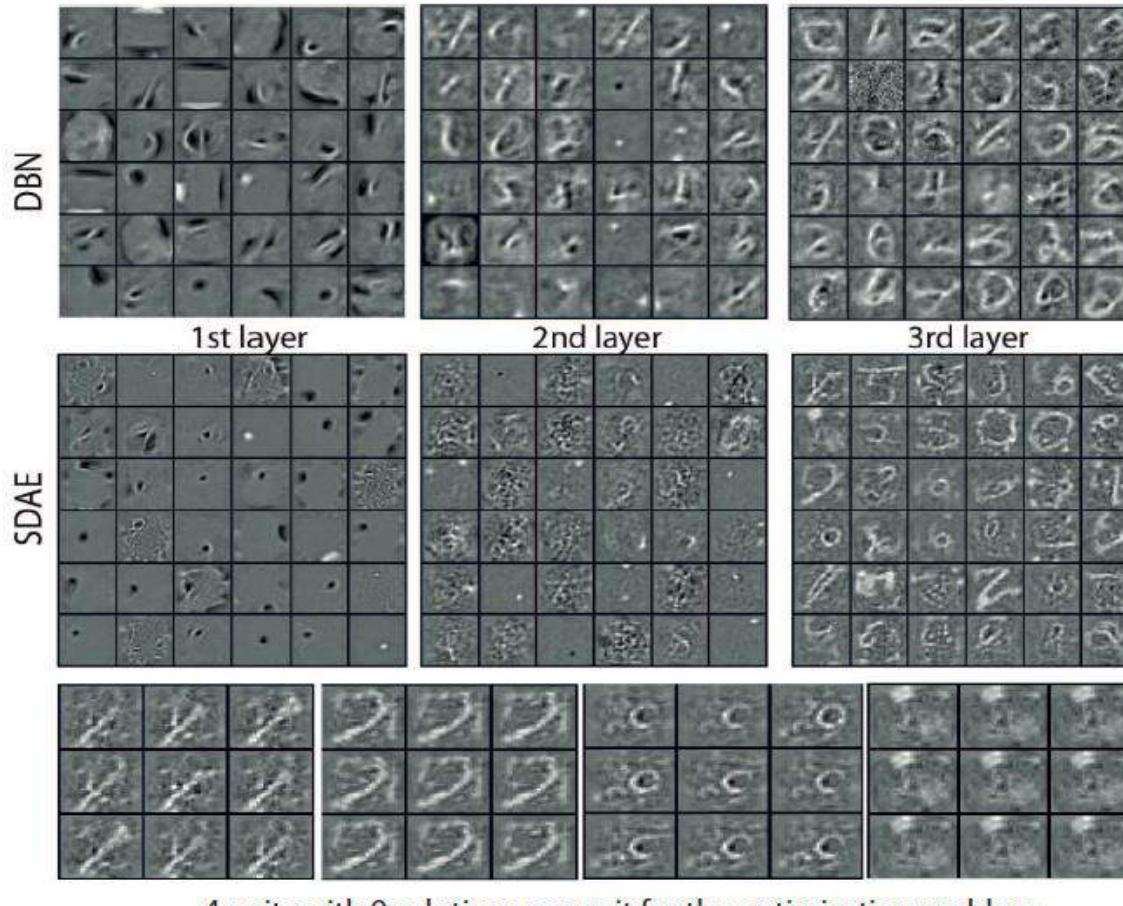
where we assume that  $\sum_i x_i^2 \leq 1$ , and hence normalize the weights to match the range of the input values.

- How about the following layers?
    - Gradient ascent (**not descent**): find the gradient of  $h_i(W, \mathbf{x})$  w.r.t  $\mathbf{x}$  and move  $\mathbf{x}$  in the direction of the gradient since we want to maximize  $h_i(W, \mathbf{x})$ .

# Visualizing the layers

- Activation maximization:
  - Gradient ascent to maximize  $h_i(W, \mathbf{x})$ .
  - Start with **randomly generated input** and move towards the gradient.
  - Luckily, different random initializations yield very similar filter-like responses.
- Applicable to any network for which we can calculate the gradient  $\partial h_i(W, \mathbf{x}) / \partial \mathbf{x}$
- Need to tune parameters:
  - Learning rate
  - Stopping criteria
- Have the same problems of gradient descent
  - The space is non-convex
  - Local maxima etc.

# Activation Maximization results



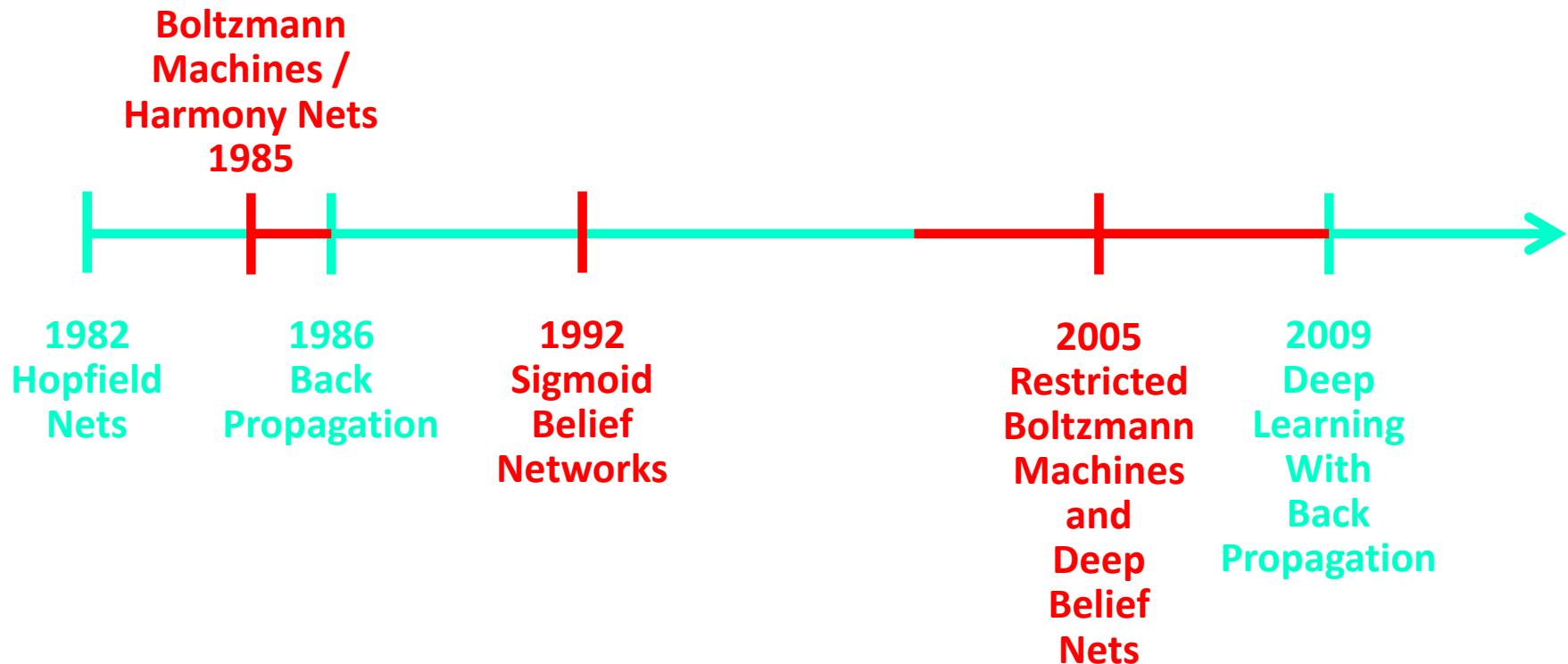
# New studies

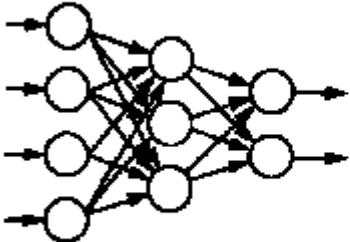
- Unsupervised pretraining with setting noise as target:
- <https://arxiv.org/abs/1704.05310>

# Hopfield Networks & Boltzmann Machines

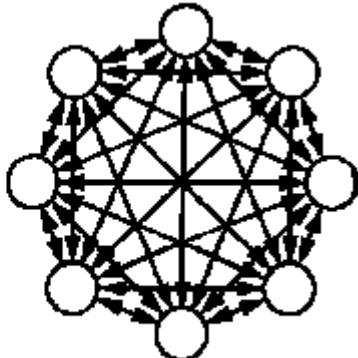
Generative Modeling

# A Brief History Of Deterministic And Stochastic Networks

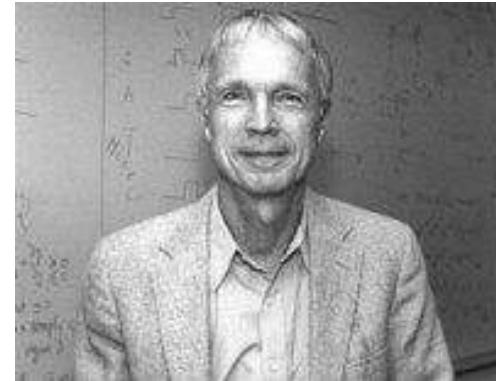




RUMMELHART



HOPFIELD

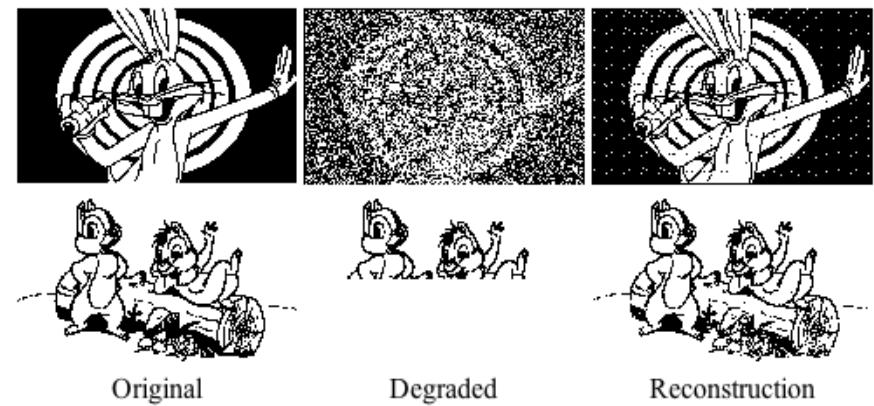
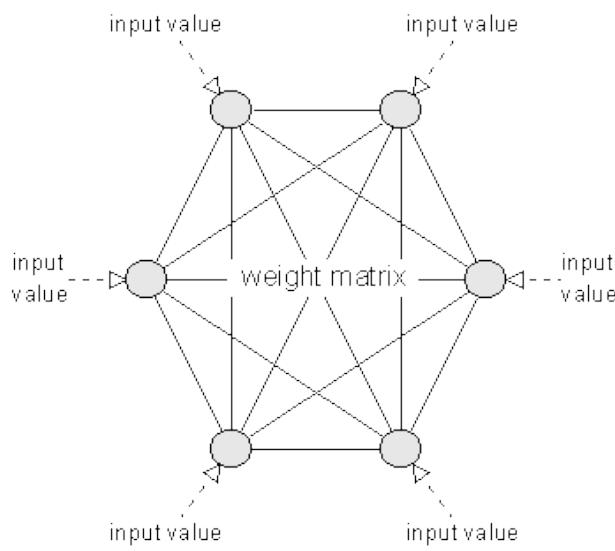


*Neural networks and physical systems with emergent collective computational properties, Hopfield and Tank, PNAS, 1982.*

# Hopfield Networks

# Hopfield networks

- No input-output differentiation. Every neuron is both an input & output
- An **undirected weighted** graph  $G = (V, E)$ 
  - The vertices ( $V$ ) are the neurons
  - The edges ( $E$ ) are connections between the neurons (all neurons are connected)
  - The edges are real valued:  $w_{ij} \in R$  ( $w_{ij} = w_{ji}$  and  $w_{ii} = 0$ )
  - The activations are binary (+1/-1 or 1/0)
- Content addressable memory

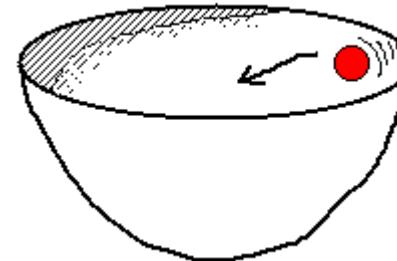


- Training on a set of patterns causes them to become attractors
- Degraded input is mapped to nearest attractor



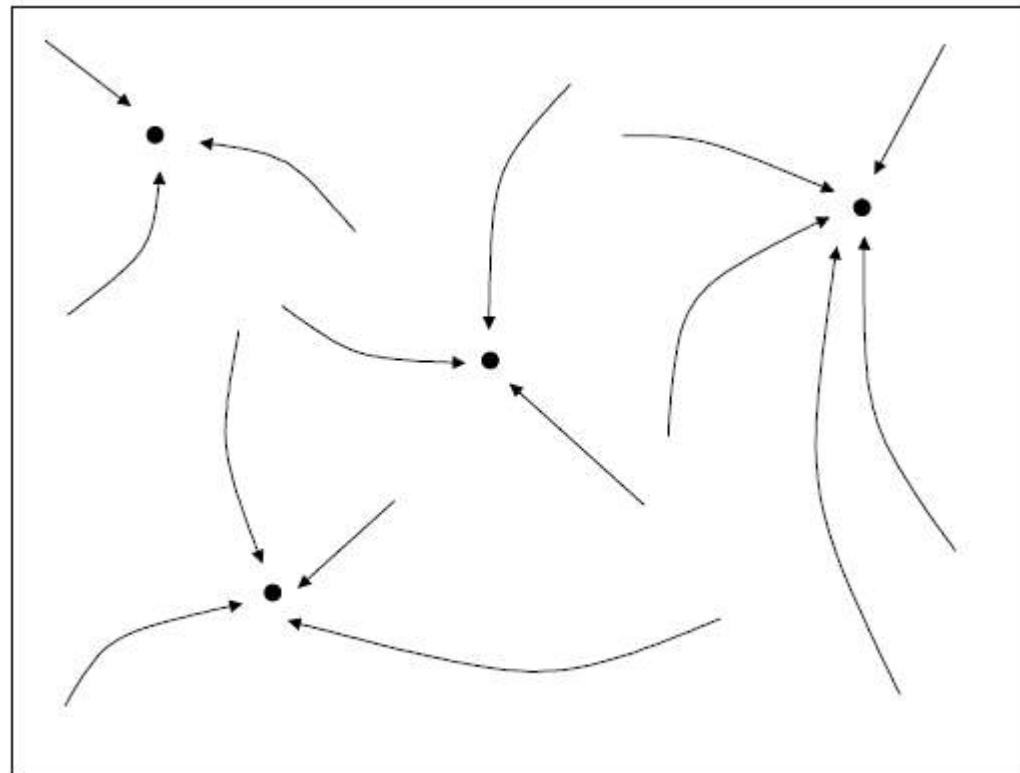
# Content-addressable memories in physical systems

- Physical systems can also act as a content-addressable memory (CAM).
- Consider the time evolution of the system in its state space. Some systems can be configured to have locally stable points. If the system is released from an initial state then it will roll down to the closest stable state.



# Defining CAM

- CAM can be defined as a system whose stable points can be set as a set of pre-defined states.
- The stored patterns divide the state space into locally stable points, called “basins of attraction” in dynamical systems theory.



# State of a neuron

- Let  $s_i$  denote the state (value/activation) of neuron  $i$ .

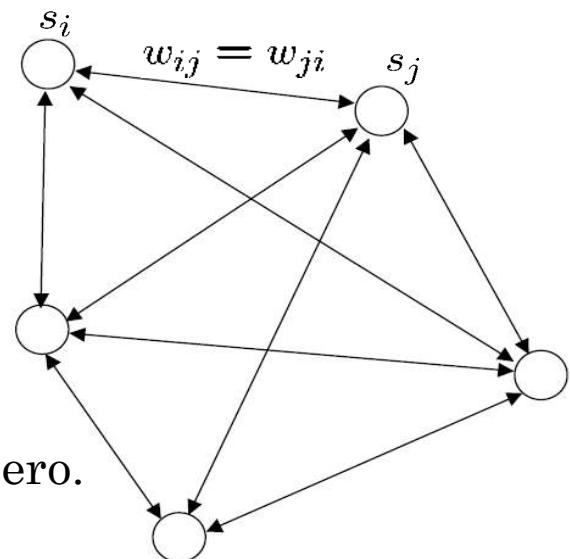
- Note that  $s_i = -1$  or  $+1$

- Then,

$$s_i \leftarrow \begin{cases} +1, & \sum_j w_{ij} s_j \geq \theta_i \\ -1, & \text{otherwise} \end{cases}$$

- $\theta_i$ : threshold of neuron  $i$ . Mostly we set this zero.
- In short:

$$s_i = \operatorname{sign}\left(\left[\sum_j w_{ij} s_j\right] - \theta_i\right)$$



# Updating neurons

- Three possible schemes:
  - Synchronously: all units updated at each step.
  - Asynchronously I: at each time step select a random unit for update.
  - Asynchronously II: each unit independently chooses to update itself with some probability per unit time.
- Use asynchronously I and keep updating until no neuron changes its state.

# Learning to store a single pattern

- Consider memorizing the pattern  $\xi$ . The condition is then

$$sgn\left(\sum_j w_{ij}\xi_j\right) = \xi_i$$

for all  $i$ .

- One solution is to have

$$w_{ij} = \frac{1}{N}\xi_i\xi_j$$

since  $\xi_j^2 = 1$ .

- If more than half of the bits are the same as  $\xi$  then the network will “recall”  $\xi$ . That is  $\xi$  is an attractor of the system.
- Question: what will happen if more than half of the bits are different from  $\xi$ ?

# Learning to store **many** patterns

Make  $w_{ij}$  a superposition of correlation terms:

$$w_{ij} = \frac{1}{N} \sum_{\mu} \xi_i^{\mu} \xi_j^{\mu}$$

## Observations

- weights can be positive or negative
- weight change positive if  $\xi_i^{\mu} = \xi_j^{\mu}$
- weight change negative otherwise

“An associative memory model using the Hebb rule for all possible pairs  $(i, j)$  with binary units and asynchronous updateing, is usually called a Hopfield model.”

# Overall procedure

1. Determine the weights for the patterns to be stored (i.e., learning)
2. When you want to recall a pattern,
  - a. set the neurons with the available parts of the pattern and
  - b. let the network converge to a stable state (a pattern)

# Example

Two patterns

$$\xi^1 = (-1, -1, -1, +1)$$

$$\xi^2 = (+1, +1, +1, +1)$$

Compute weights

$$w_{ij} = \frac{1}{4} \sum_{\mu=1}^2 \xi_i^\mu \xi_j^\mu$$

$$w = \frac{1}{4} \begin{bmatrix} 2 & 2 & 2 & 0 \\ 2 & 2 & 2 & 0 \\ 2 & 2 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Recall

$$s_i = Sgn \left( \sum_j w_{ij} s_j \right)$$

Input  $(-1, -1, -1, +1)$  →  $(-1, -1, -1, +1)$  stable

Input  $(-1, +1, +1, +1)$  →  $(+1, +1, +1, +1)$  stable

Input  $(-1, -1, -1, -1)$  →  $(-1, -1, -1, -1)$  spurious

# Harmony or Energy

- We can define a scalar for the energy of the state of the network:

$$E = - \sum_i \sum_{j < i} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

or

$$E = -\frac{1}{2} \sum_i \sum_j w_{ij} s_i s_j + \sum_i \theta_i s_i$$

- Harmony (H) = negative of the energy
- This is called energy since when you update neurons randomly, it either decreases or stays the same.
- Repeatedly updating the network will eventually make the network converge to a local minimum, i.e., a stable state.

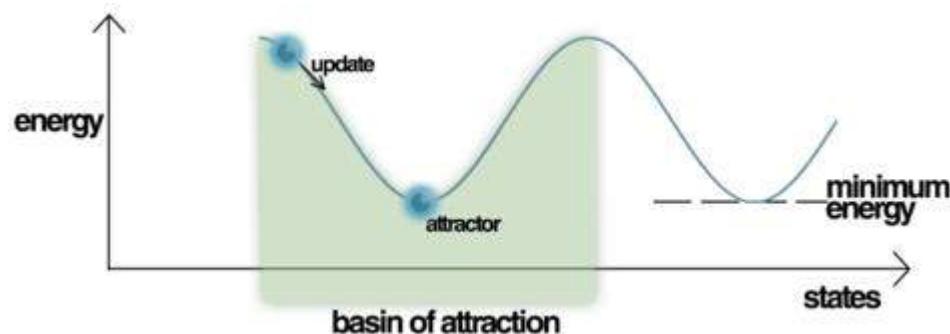


Fig: Wikipedia

# Why does energy stay constant or decrease?

- Let us use  $S'_i$  for the updated value of  $S_i$ :

$$S'_i = \text{sign} \left( \sum_j w_{ij} S_j \right)$$

- Then, the change in the energy can be calculated as follows (assume that  $w_{ii} = 0$ ):

$$\begin{aligned}\Delta E = E' - E &= - \sum_j w_{ij} S'_i S_j - \left( - \sum_j w_{ij} S_i S_j \right) \\ &= 2S_i \sum_j w_{ij} S_j\end{aligned}$$

- The term  $(S_i \sum_j w_{ij} S_j)$  is negative due to the definition given in the first item.
- Therefore,  $E'$  is lower than  $E$  if  $S'_i \neq S_i$ .
  - If  $S'_i = S_i$ ,  $\Delta E = 0$ .

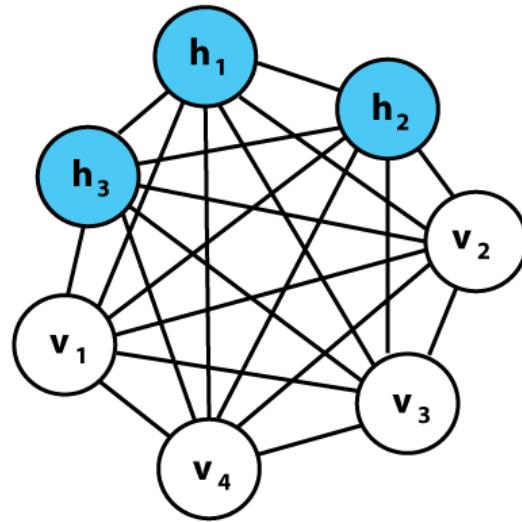
# Skipping many details

- Stability conditions
- Storage capacity
- Increasing robustness
- Extension for continuous-valued patterns
- ...



Ludwig Boltzmann  
(1844 – 1906)

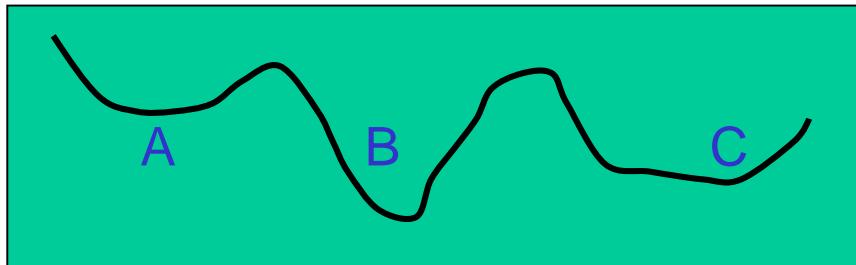
# Boltzmann Machines



# Motivation

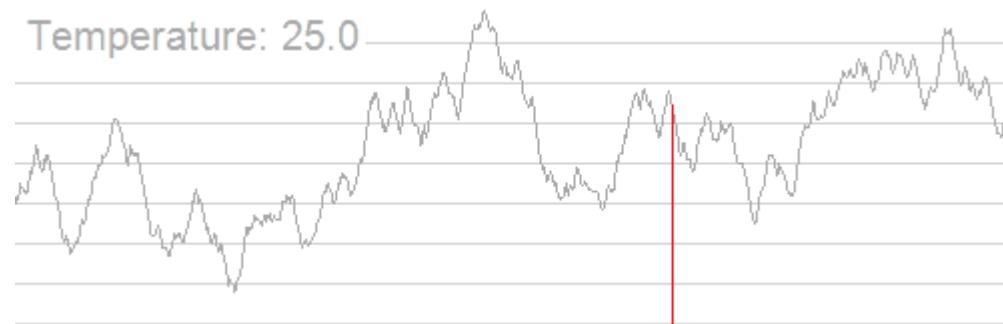
---

- A Hopfield net always makes decisions that reduce the energy.
  - This makes it impossible to escape from local minima.
- We can use random noise to escape from poor minima.
  - Start with a lot of noise so its easy to cross energy barriers.
  - This may mean we occasionally increase the energy
  - Slowly reduce the noise so that the system ends up in a deep minimum. This is “simulated annealing”.



# Simulated annealing

- With a probability dependent on the temperature, choose to either stay in the current state or move to a neighboring state.



# Boltzmann (Gibbs) Distribution

- Probability distribution of particles in a system over possible states:  
$$F(state) \propto e^{-\epsilon/kT}$$

$\epsilon$ : the energy of the state,  $k$ : Boltzmann's constant,  $T$ : temperature.

- The probability that a system will be in a certain state:

$$p_i = \frac{e^{-\varepsilon_i/kT}}{\sum_{j=1}^M e^{-\varepsilon_j/kT}}$$

where  $\varepsilon_i$  is the energy of state  $i$ .

- Interpretation: states with lower energy are more probable.
- The denominator is called the partition function (denoted by  $Q$  or  $Z$  in the literature):

$$Z = \sum_{j=1}^M e^{-\varepsilon_j/kT}$$

# Boltzmann constant

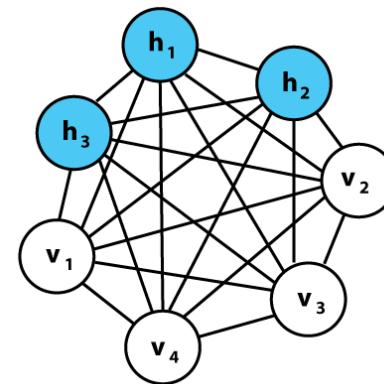
- From Wikipedia:

The **Boltzmann constant** ( $k_B$  or  $k$ ), named after [Ludwig Boltzmann](#), is a physical constant relating [energy](#) at the individual particle level with [temperature](#). It is the [gas constant](#)  $R$  divided by the [Avogadro constant](#)  $N_A$ :

$$k = \frac{R}{N_A}.$$

The Boltzmann constant has the dimension [energy](#) divided by [temperature](#), the same as [entropy](#). The accepted value in SI units is  $1.380\ 648\ 52(79) \times 10^{-23}\ \text{J/K}$ .

# Boltzmann Machines



- By Hinton & Sejnowski (1985)
- Boltzmann machines can be seen as the **stochastic counterpart of Hopfield nets**

- In fact, they have the same energy definition:

$$E = - \sum_i \sum_{j < i} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

- However, we have hidden neurons now
  - The availability of hidden variables → bigger class of distributions that can be modeled → in principle, we can model distributions of arbitrary complexity
- Moreover, it is stochastic

# Boltzmann Machines (cont'd)

- They are powerful machines
- Can be used to learn internal representations
- Can be used to represent and learn difficult combinatoric problems (given sufficient time)
- However, if the connectivity is unconstrained, learning/inference is not tractable

# Probability of a neuron's state

- Turning on a neuron  $i$  (i.e.,  $s_i$  is changed to 1 from 0) causes change  $\Delta E_i$  in energy:

$$\begin{aligned}\Delta E_i &= E_{i=0} - E_{i=1} \\ &= -k T \ln(Z p_{i=0}) - (-k T \ln(Z p_{i=1})) \\ &= -k T \ln\left(\frac{Z p_{i=0}}{Z p_{i=1}}\right) = -k T \ln\left(\frac{p_{i=0}}{p_{i=1}}\right) = -k T \ln\left(\frac{1 - p_{i=1}}{p_{i=1}}\right)\end{aligned}$$

Using:  
 $p_i = \frac{e^{-\varepsilon_i/kT}}{Z}$

- $k$  is a constant. Assume that the artificial concept of temperature ( $T$ ) “absorbs” that:

$$\begin{aligned}-\frac{\Delta E_i}{T} &= \ln\left(\frac{1 - p_{i=1}}{p_{i=1}}\right) \\ \exp\left(-\frac{\Delta E_i}{T}\right) &= \frac{1}{p_{i=1}} - 1\end{aligned}$$

- This yields the famous logistic / sigmoid function:

$$p_{i=1} = \frac{1}{1 + \exp\left(-\frac{\Delta E_i}{T}\right)}$$

# Interpretation of a state's probability

$$p_{i=1} = \frac{1}{1 + \exp\left(-\frac{\Delta E_i}{T}\right)}$$

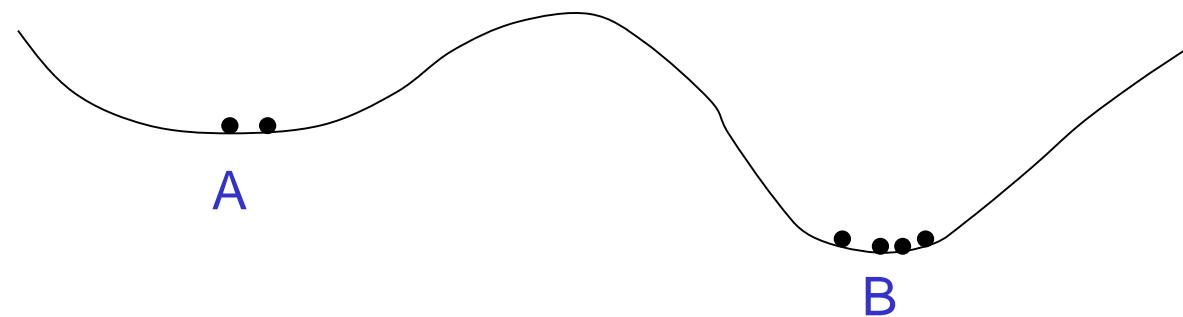
- a. If  $T = 0$ , the probability for turning on the neuron is  $\approx 1$  if  $\Delta E_i$  is positive (energy reduced). If  $\Delta E_i$  is negative,  $p_{i=1} \approx 0$ .
  - Does it become equal to a Hopfield network then?
- b. If  $T$  is high ( $\rightarrow \infty$ ), then  $p_{i=1} \approx 1/2$ .
  - Half the chance is given to updating the neuron.
- c. For a fixed  $T$ , if  $\Delta E_i$  is zero, same as case – (b).
- d. For a fixed  $T$ , if  $\Delta E_i$  is very high, same as case – (a).
  - When the temperature is big, the network covers the whole state space.
  - In the cooling phase, when the temperature is small, the network converges to a minima, hopefully the global one.

# How temperature affects transition probabilities

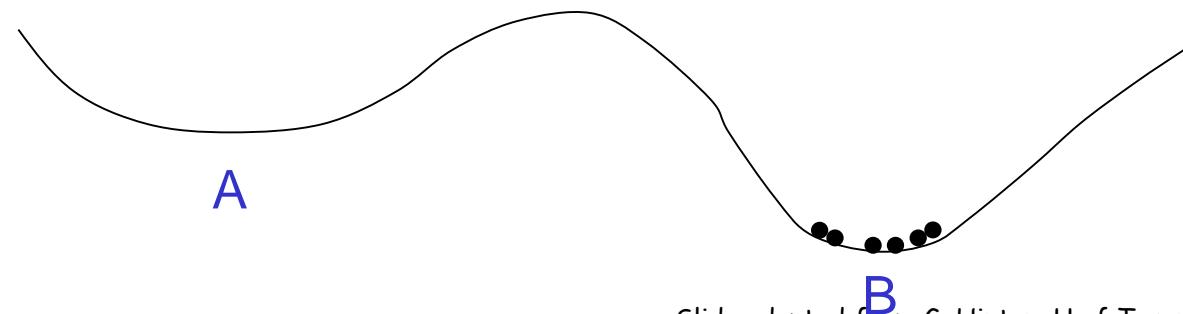
$$p(A \rightarrow B) = 0.2$$

$$p(A \leftarrow B) = 0.1$$

High temperature  
transition  
probabilities



Low temperature  
transition  
probabilities



# Energy change in Hopfield vs. Boltzmann Machines

- Energy decreases in Hopfield networks
- Although we have the same definition, why can energy increase in Boltzmann machines?
  - Because state updates that normally would not be possible in a Hopfield network is possible in Boltzmann machines.

An example of how weights define a distribution

$$p(\mathbf{v}^\alpha, \mathbf{h}^\beta) = \frac{e^{-E^{\alpha\beta}}}{\sum_{\gamma\delta} e^{-E^{\gamma\delta}}}$$

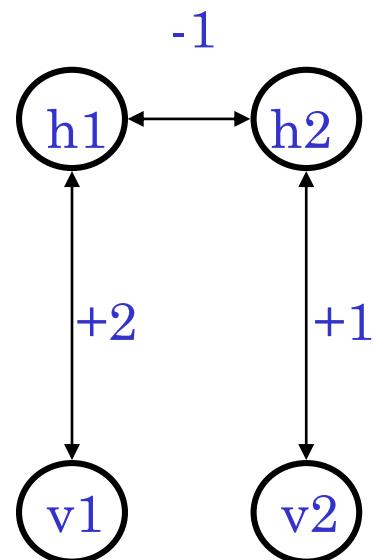
$\mathbf{v}$	$\mathbf{h}$	$-E$	$e^{-E}$	$p(\mathbf{v}, \mathbf{h})$	$p(\mathbf{v})$
1 1	1 1	2	7.39	.186	
1 1	1 0	2	7.39	.186	
1 1	0 1	1	2.72	.069	0.466
1 1	0 0	0	1	.025	
1 0	1 1	1	2.72	.069	
1 0	1 0	2	7.39	.186	
1 0	0 1	0	1	.025	0.305
1 0	0 0	0	1	.025	
0 1	1 1	0	1	.025	
0 1	1 0	0	1	.025	
0 1	0 1	1	2.72	.069	0.144
0 1	0 0	0	1	.025	
0 0	1 1	-1	0.37	.009	
0 0	1 0	0	1	.025	
0 0	0 1	0	1	.025	0.084
0 0	0 0	0	1	.025	
total = 39.70					

0.466

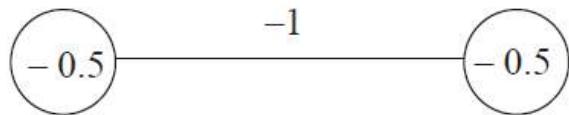
0.305

0.144

0.084



# An example



$$E = -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{ij} x_i x_j + \sum_{i=1}^n \theta_i x_i$$

**Fig. 14.2.** A flip-flop network  
(Threshold is -0.5 for both units)

In this network, there are four possible states with the following energies:

$$\begin{array}{lll} E_{00} = & 0.0 & E_{10} = & -0.5 \\ E_{01} = & -0.5 & E_{11} = & 0.0 \end{array}$$

You can calculate transition probabilities between states:

$$p_{00 \rightarrow 01} = \frac{1}{2} \left( \frac{1}{1 + \exp(-0.5)} \right) = 0.31 \quad (T = 1)$$

# An example

This can be summarized in the following transition matrix:

$$P = \begin{pmatrix} \frac{1}{1+e^{1/2}} & \frac{1}{2} \frac{1}{1+e^{-1/2}} & \frac{1}{2} \frac{1}{1+e^{-1/2}} & 0 \\ \frac{1}{2} \frac{1}{1+e^{1/2}} & \frac{1}{1+e^{-1/2}} & 0 & \frac{1}{2} \frac{1}{1+e^{1/2}} \\ \frac{1}{2} \frac{1}{1+e^{1/2}} & 0 & \frac{1}{1+e^{-1/2}} & \frac{1}{2} \frac{1}{1+e^{1/2}} \\ 0 & \frac{1}{2} \frac{1}{1+e^{-1/2}} & \frac{1}{2} \frac{1}{1+e^{-1/2}} & \frac{1}{1+e^{1/2}} \end{pmatrix} = \begin{pmatrix} 0.38 & 0.31 & 0.31 & 0 \\ 0.19 & 0.62 & 0 & 0.19 \\ 0.19 & 0 & 0.62 & 0.19 \\ 0 & 0.31 & 0.31 & 0.38 \end{pmatrix}$$

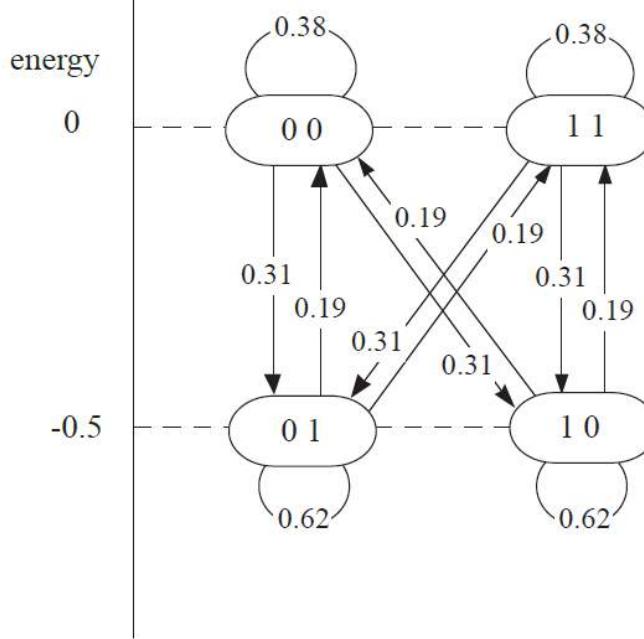


Fig. 14.3. Transition probabilities of a flip-flop network ( $T = 1$ )

# An example

Let us start from an arbitrary state. This has the following distribution:

$$\mathbf{v}_0 = (0.25, 0.25, 0.25, 0.25)$$

If we update this distribution using the transition matrix:

$$\begin{aligned}\mathbf{v}_1 &= \mathbf{v}_0 P \\ &= (0.25, 0.25, 0.25, 0.25) \begin{pmatrix} 0.38 & 0.31 & 0.31 & 0 \\ 0.19 & 0.62 & 0 & 0.19 \\ 0.19 & 0 & 0.62 & 0.19 \\ 0 & 0.31 & 0.31 & 0.38 \end{pmatrix} \\ &= (0.19, 0.31, 0.31, 0.19)\end{aligned}$$

# An example

We can continue doing that:

$$\mathbf{v}_t = \mathbf{v}_{t-1} \mathbf{P}.$$

In thermal equilibrium:

$$\mathbf{v} = \mathbf{v}\mathbf{P}$$

We could also calculate it directly in this case since the network is so small:

$$\begin{aligned}\mathbf{v} &= \frac{1}{e^{-E_{00}} + e^{-E_{01}} + e^{-E_{10}} + e^{-E_{11}}} (e^{-E_{00}}, e^{-E_{01}}, e^{-E_{10}}, e^{-E_{11}}) \\ &= \frac{1}{2} \left( \frac{1}{1 + e^{0.5}} \right) (1, e^{0.5}, e^{0.5}, 1) \\ &= (0.19, 0.31, 0.31, 0.19).\end{aligned}$$

# An example

We can continue doing that:

$$\mathbf{v}_t = \mathbf{v}_{t-1} \mathbf{P}.$$

However, the transition matrix is also updated in Boltzmann machines:

$$\mathbf{v}_t = \mathbf{v}_{t-1} \mathbf{P} = \mathbf{v}_{t-2} \mathbf{P}^2 = \cdots = \mathbf{v}_0 \mathbf{P}^t$$

Once the stable state is reached, the transition probabilities also stabilize:

$$\lim_{t \rightarrow \infty} \mathbf{P}^t = \begin{pmatrix} 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \end{pmatrix}.$$

# Equilibrium

- We select a neuron and update its state according to the following probability:

$$p_{i=1} = \frac{1}{1 + \exp\left(-\frac{\Delta E_i}{T}\right)}$$

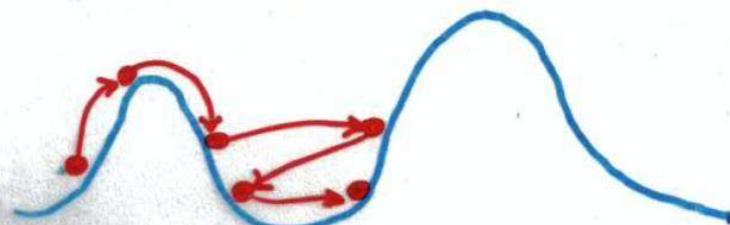
- If this is repeated long enough for a certain temperature, the state of the network will depend on the state's energy, and not on the initial state.
- In this condition, the log probabilities of global states become linear in their energies → Boltzmann distribution!
- This is called **thermal equilibrium**.
- Start from a high temperature, gradually decrease it until thermal equilibrium, we may converge to a distribution where energy level is close to the global minimum. → Simulated Annealing.

## Thermal Equilibrium

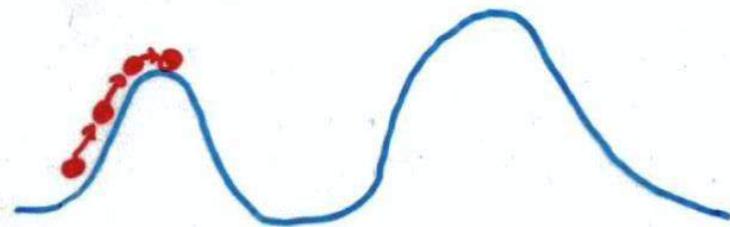
Start the system running at some temperature  $T$

Immediate behavior of system will be dependent on initial state

At high temperatures, system quickly moves away from initial state



At low temperatures, system takes much longer



# Thermal Equilibrium

- How do we understand we have reached it?
  - The **average activation of neurons** don't change over time.
  - I.e., the **probability of being in a state** does not change.
- How can we know that we can reach thermal equilibrium?
  - “*any network that always went downhill in some Lyapunov function in its deterministic version is guaranteed to reach a thermal equilibrium in its stochastic version*” (S. Roweis)
  - In the deterministic case, our Lyapunov function is:
$$E = - \sum_{i,j < i} w_{ij} s_i s_j$$
- The initial state is not important!
- At low temperature:
  - There is a strong bias for states with low energy
  - But this is too slow to reach
- At high temperature:
  - Not a strong bias for low energy
  - Equilibrium is reached faster

# Thermal equilibrium

Thermal equilibrium is a difficult concept!

- It does not mean that the system has settled down into the lowest energy configuration.
- The thing that settles down is the probability distribution over configurations.

The best way to think about it is to imagine a huge ensemble of systems that all have exactly the same energy function.

- The probability distribution is just the fraction of the systems that are in each possible configuration.
- We could start with all the systems in the same configuration, or with an equal number of systems in each possible configuration.
- After running the systems stochastically in the right way, we eventually reach a situation where the number of systems in each configuration remains constant even though any given system keeps moving between configurations

# Training

- The neurons are divided into two:
  - Visible units:  $V$
  - Hidden units:  $H$
- Distribution over the training set:  $P^+(V)$
- The distribution over global states converges as the Boltzmann machine reaches thermal equilibrium.
  - $P^-(V)$  to denote this distribution
- Our goal: Approximate the real distribution  $P^+(V)$  from  $P^-(V)$
- The similarity bw the distributions:

$$G = D_{KL}(P^+(V), P^-(V)) = \sum_v P^+(v) \ln \left( \frac{P^+(v)}{P^-(v)} \right)$$

summation over **all possible states of  $V$ .**

- $G$  is a function of weights.
  - We can use gradient descent on  $G$  to update the weights to minimize it.

# Training (cont'd)

- Two phases:
  - Positive phase: visible units are initialized to a random sample from the training set.
  - Negative phase: the network runs freely. The units are not initialized to external data.
- Then:
$$\frac{\partial G}{\partial w_{ij}} = \frac{1}{R} [p_{ij}^+ - p_{ij}^-]$$
  - R: learning rate
  - $p_{ij}^+$ : probability that **both units are on** at thermal equilibrium on the positive phase.  
 $E[s_i s_j]$
  - $p_{ij}^-$ : probability that **both units are on** at thermal equilibrium on the negative phase.
- $w_{ij} = w_{ij} - \frac{\partial G}{\partial w_{ij}}$
- Needs only local information (compare it to backprop)

See the following for the derivation of the learning rule:

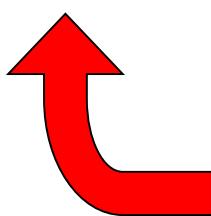
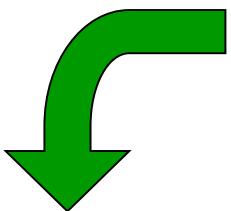
Hinton, Geoffrey E., and Terrence J. Sejnowski. "Learning and relearning in Boltzmann machines." *Parallel distributed processing: Explorations in the microstructure of cognition* 1 (1986): 282-317.

# Training (cont'd)

- Another interpretation (Rojas, 1996):

Note that Boltzmann learning resembles Hebbian learning. The values of  $\langle x_i x_j \rangle_{fixed}$  and  $\langle x_i x_j \rangle_{free}$  correspond to the entries of the stochastic correlation matrix of network states. The second term is subtracted and is interpreted as Hebbian “forgetting”. This controlled loss of memory should prevent the network from learning false, spontaneously generated states. Obviously, Boltz-

# Why do we need the negative phase?

$$p(\mathbf{v}) = \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{u}} \sum_{\mathbf{g}} e^{-E(\mathbf{u}, \mathbf{g})}}$$


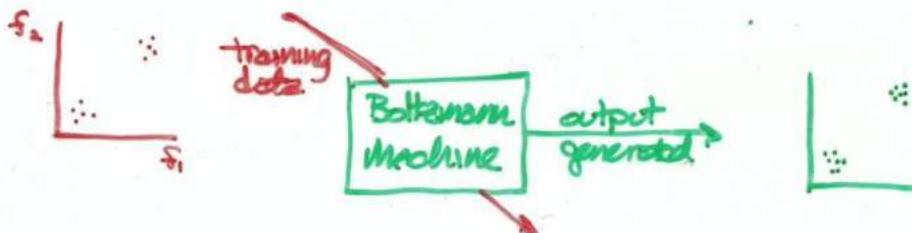
The positive phase finds hidden configurations that work well with  $\mathbf{v}$  and lowers their energies.

The negative phase finds the joint configurations that are the best competitors and raises their energies.

## Boltzmann Machine learning algorithm

Goal: Model structure of environment

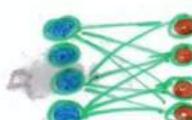
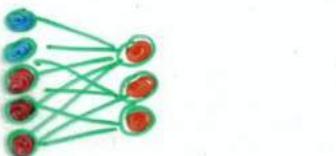
Serve as generative model of environment



Can be applied to both supervised & unsupervised learning.



Restricted architecture: Harmony network  
(Smolensky, 1987)



- inputs
- outputs
- hidden

### Terminology

clamped: frozen value of input or output unit

clamped phase: all inputs & outputs clamped  
aka, "wake phase", "reality phase"

unclamped phase: all units allowed to change activity levels  
aka "sleep phase", "fantasy phase"

# The batch learning algorithm

---

- Positive phase
  - Clamp a datavector on the visible units.
  - Let the hidden units reach thermal equilibrium at a temperature of 1 (may use annealing to speed this up)
  - Sample  $s_i s_j$  for all pairs of units
  - Repeat for all datavectors in the training set.
- Negative phase
  - Do not clamp any of the units
  - Let the whole network reach thermal equilibrium at a temperature of 1
  - Sample  $s_i s_j$  for all pairs of units
  - Repeat many times to get good estimates
- Weight updates
  - Update each weight by an amount proportional to the difference in  $\langle s_i s_j \rangle$  in the two phases.

# Why Boltzmann Machine Failed

## Too slow

- loop over training epochs
  - loop over training examples
  - loop over 2 phases (+ and -)
  - loop over annealing schedule for T
  - loop until thermal equilibrium reached
  - loop to sample  $\langle o_i o_j \rangle$

## Sensitivity to annealing schedule

## Difficulty determining when equilibrium is reached

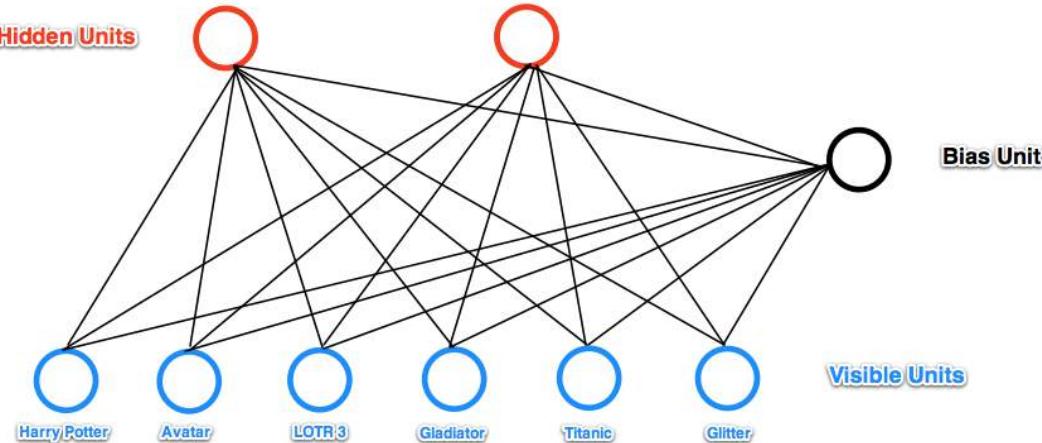
As learning progresses, weights get larger, energy barriers get hard to break -> becomes even slower

## Back prop was invented shortly after

- The need to perform pattern completion wasn't necessary for most problems (feedforward nets sufficed)

# References

- R. Rojas, “Neural Networks”, Ch14, 1996.
- Hinton, Geoffrey E., and Terrence J. Sejnowski. "Learning and relearning in Boltzmann machines." *Parallel distributed processing: Explorations in the microstructure of cognition* 1 (1986): 282-317.
- S. Roweis, “Boltzmann Machines”, lecture notes:  
<https://www.cs.nyu.edu/~roweis/notes/boltz.pdf>
- For links and inspirations to physical systems (like Ising models and renormalization group theory), see:
  - <https://charlesmartin14.wordpress.com/2015/03/25/why-does-deep-learning-work/>
  - <https://charlesmartin14.wordpress.com/2015/04/01/why-deep-learning-works-ii-the-renormalization-group>



<http://blog.echen.me/2011/07/18/introduction-to-restricted-boltzmann-machines/>

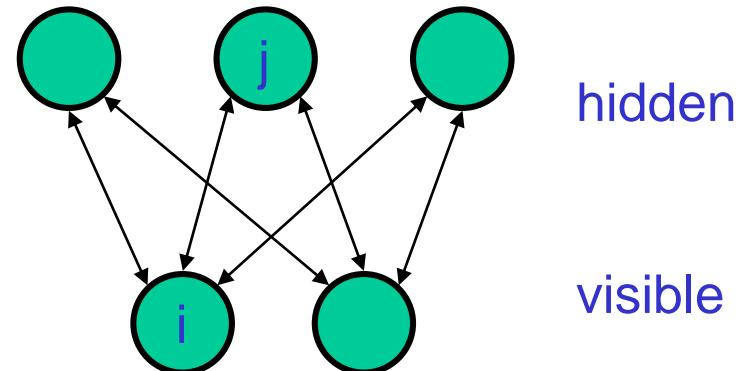
# Restricted Boltzmann Machines

# Restricted Boltzmann Machines (RBM)

- Invented by Smolensky (1986), improved by Hinton et al. (2006)
- RBM: Boltzmann Machine with restricted connectivity
  - Connections between hidden-visible units only!
- Smolensky called it Harmonium
  - Also called Harmony networks
- A simple learning method is used
  - Contrastive Divergence

# Restricted Boltzmann Machines

- We restrict the connectivity to make inference and learning easier.
  - Only one layer of hidden units.
  - No connections between hidden units.
- In an RBM it only takes one step to reach thermal equilibrium when the visible units are clamped.
  - So we can quickly get the exact value of : $\langle s_i s_j \rangle_v$



$$p(s_j = 1) = \frac{1}{1 + e^{-(b_j + \sum_{i \in vis} s_i w_{ij})}}$$

# Overview of training

- Again, we should update the weights according to the “positive” phase and the “negative” phase:

$$\Delta w_{ij} = R(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model})$$

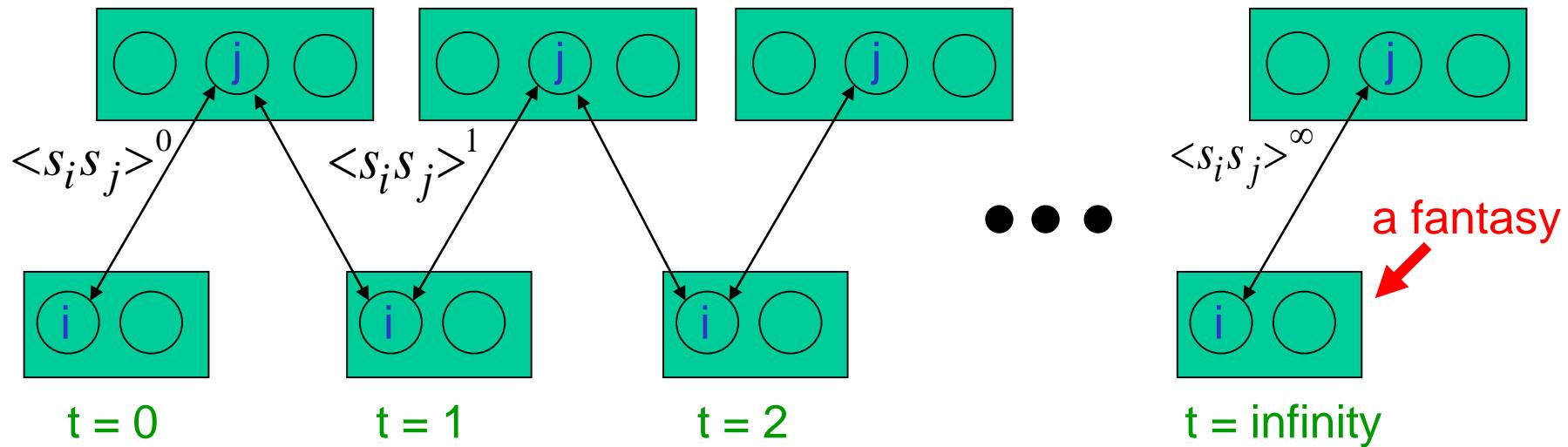
- For the “data” part:
  - Clamp the visible units with a pattern and determine the visible units with probability:

$$p(h_j = 1 | v) = \sigma\left(b_j + \sum_i v_i w_{ij}\right) = \frac{1}{1 + \exp(b_j + \sum_i v_i w_{ij})}$$

- The “model” part is more problematic and slow.
- Shortcut (Hinton 2002):

$$\Delta w_{ij} = R(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recons})$$

# A picture of the Boltzmann machine learning algorithm for an RBM

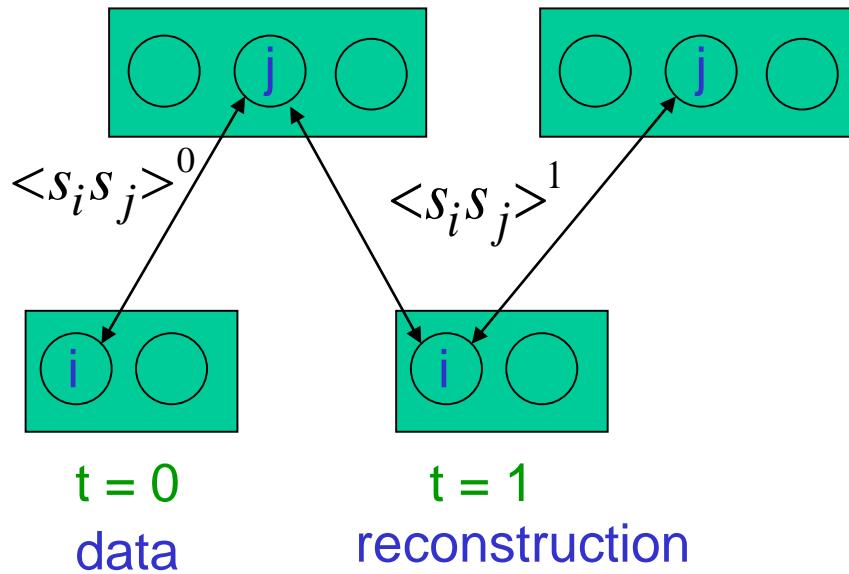


Start with a training vector on the visible units.

Then alternate between updating all the hidden units in parallel and updating all the visible units in parallel.

$$\Delta w_{ij} = \epsilon (\langle s_i s_j \rangle^0 - \langle s_i s_j \rangle^\infty)$$

# A surprising short-cut



Start with a training vector on the visible units.

Update all the hidden units in parallel

Update the all the visible units in parallel to get a “reconstruction”.

Update the hidden units again.

$$\Delta w_{ij} = \epsilon (\langle s_i s_j \rangle^0 - \langle s_i s_j \rangle^1)$$

This is not following the gradient of the log likelihood. But it works very well.

# Overview of training

- Take a training sample  $v$ , sample a hidden vector using:

$$p(h_j = 1 | v) = \sigma\left(b_j + \sum_i v_i w_{ij}\right)$$

- Compute the outer product of  $v$  and  $h$  and call this the “positive gradient”.
- Sample a reconstruction  $v'$  from  $h$  using:

$$p(v'_i = 1 | h) = \sigma\left(b_i + \sum_j h_j w_{ij}\right)$$

then resample the hidden activations  $h'$  from this.

- Compute the outer product of  $v'$  and  $h'$  and call this the “negative gradient”.
- Update the weights:

$$\Delta w_{ij} = R(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recons})$$

# Skipping many details

- Collecting the statistics in the binary or real-valued cases
- Initial values for the weights
- Overfitting
- Momentum
- ...

Hinton, “A Practical Guide to Training Restricted Boltzmann Machines”, TR, 2010.

# RBM vs. Autoencoders

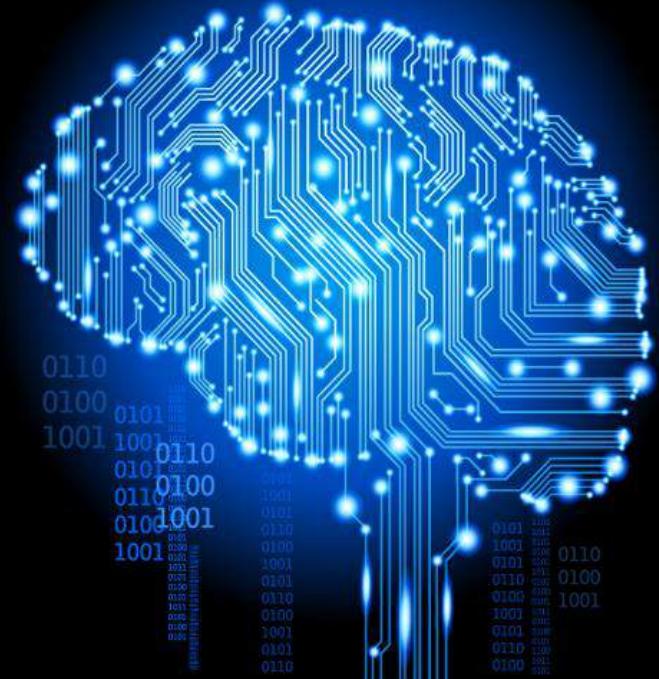
- Both try to encode an input and to minimize error in its reconstructions from its encoding
- The objective functions minimized are different
  - RBMs directly model the joint distribution of the hidden & visible units
  - AEs rely on the score of the learned decoding.
- There are many powerful extensions of both approaches
- Also many links, e.g., Vincent, “A Connection Between Score Matching and Denoising Autoencoders”, 2011.

# CENG 783

## Special topics in Deep Learning

*Week 15  
Generative Models*

Sinan Kalkan



© AlchemyAPI

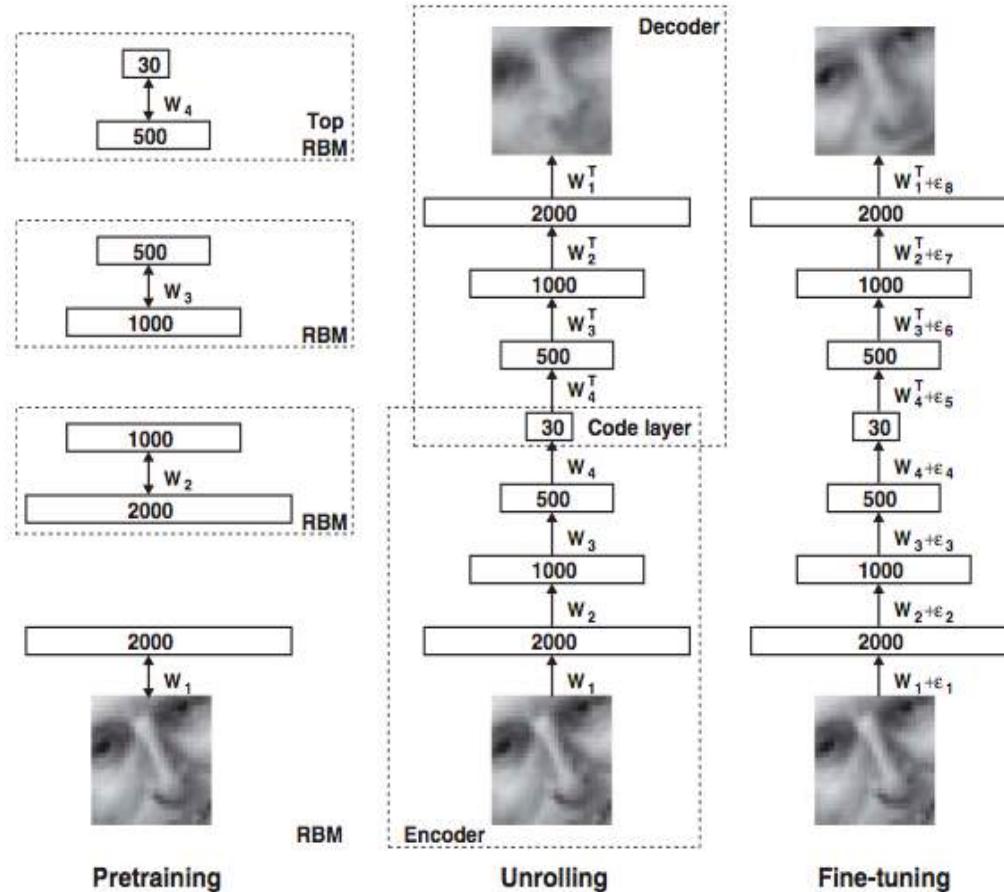


# Today

- Finalize RBMs
  - Stacking
  - Deep Belief Nets
  - Deep Boltzmann Machines
- Generative Adversarial Networks
- Variational Autoencoders
- Autoregressive models
- Deep Reinforcement Learning
- NOTES:
  - Final Exam date: 31 May, 17:00.
  - Project demos and papers due: 6 June.

# Stacking RBMs

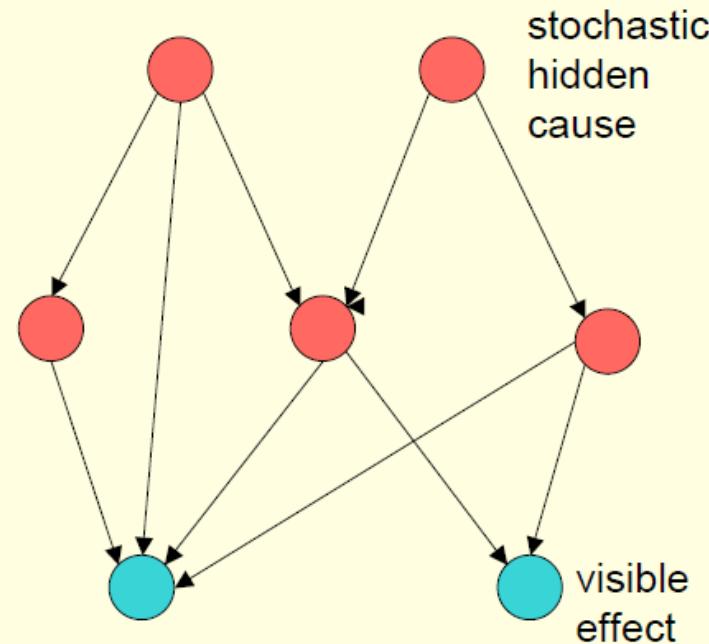
# Deep RBM Autoencoder



**Fig. 1.** Pretraining consists of learning a stack of restricted Boltzmann machines (RBMs), each having only one layer of feature detectors. The learned feature activations of one RBM are used as the “data” for training the next RBM in the stack. After the pretraining, the RBMs are “unrolled” to create a deep autoencoder, which is then fine-tuned using backpropagation of error derivatives.

# Belief Nets

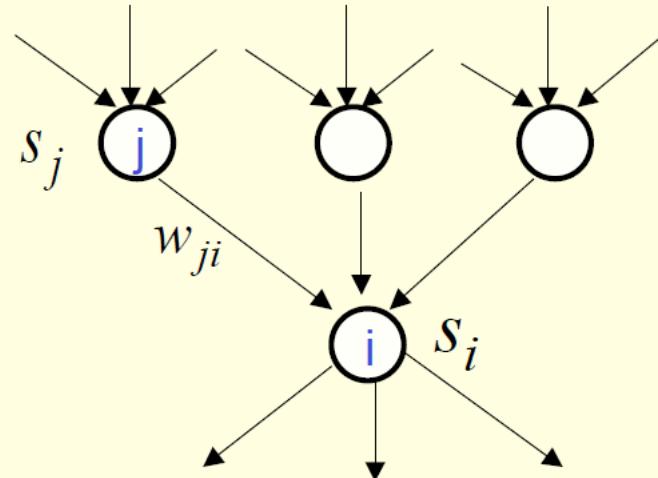
- A belief net is a directed acyclic graph composed of stochastic variables.
- We get to observe some of the variables and we would like to solve two problems:
- **The inference problem:** Infer the states of the unobserved variables.
- **The learning problem:** Adjust the interactions between variables to make the network more likely to generate the observed data.



We will use nets composed of layers of stochastic binary variables with weighted connections. Later, we will generalize to other types of variable.

# The learning rule for sigmoid belief nets

- Learning is easy if we can get an unbiased sample from the posterior distribution over hidden states given the observed data.
- For each unit, maximize the log probability that its binary state in the sample from the posterior would be generated by the sampled binary states of its parents.



$$p_i \equiv p(s_i = 1) = \frac{1}{1 + \exp(-\sum_j s_j w_{ji})}$$

$$\Delta w_{ji} = \varepsilon s_j (s_i - p_i)$$



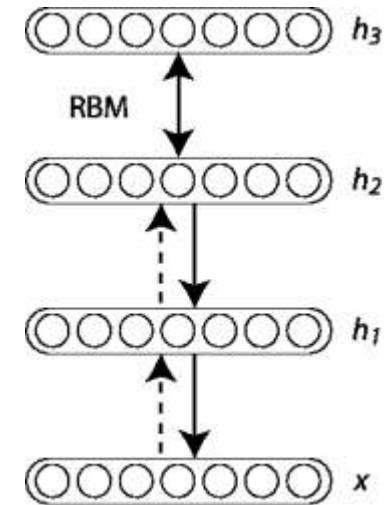
learning  
rate

# DBN

- A stacked RBM
- First used by Hinton & Salakhutdinov (2006)
- Models the distribution:

$$P(x, h^1, \dots, h^\ell) = \left( \prod_{k=0}^{\ell-2} P(h^k | h^{k+1}) \right) P(h^{\ell-1}, h^\ell)$$

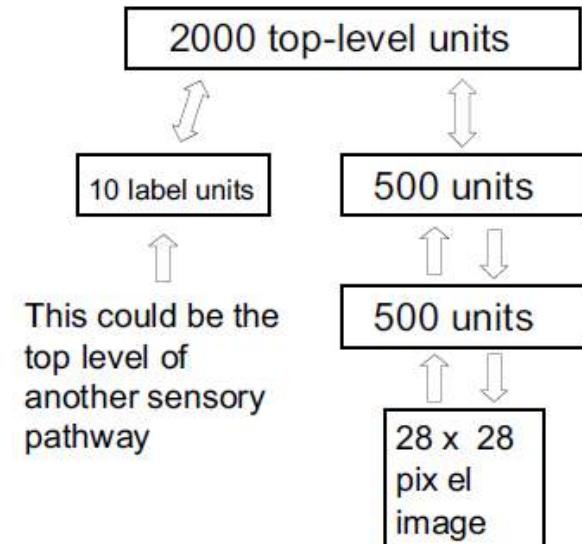
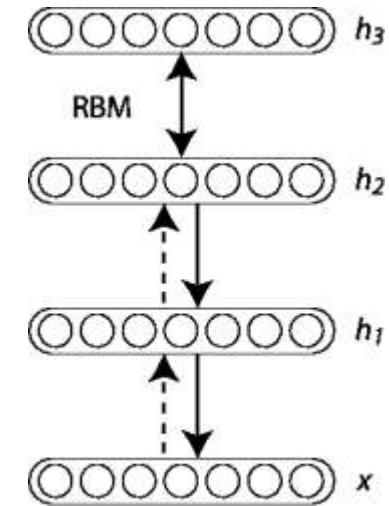
- Training is similar to autoencoders:



1. Train the first layer as an RBM that models the raw input  $x = h^{(0)}$  as its visible layer.
2. Use that first layer to obtain a representation of the input that will be used as data for the second layer. Two common solutions exist. This representation can be chosen as being the mean activations  $p(h^{(1)} = 1 | h^{(0)})$  or samples of  $p(h^{(1)} | h^{(0)})$ .
3. Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of that RBM).
4. Iterate (2 and 3) for the desired number of layers, each time propagating upward either samples or mean values.
5. Fine-tune all the parameters of this deep architecture with respect to a proxy for the DBN log-likelihood, or with respect to a supervised training criterion (after adding extra learning machinery to convert the learned representation into supervised predictions, e.g. a linear classifier).

# DBN

- Not a Boltzmann Machine
- It is a hybrid between stacked RBM & logistic belief nets.



Hinton et al., 2006.

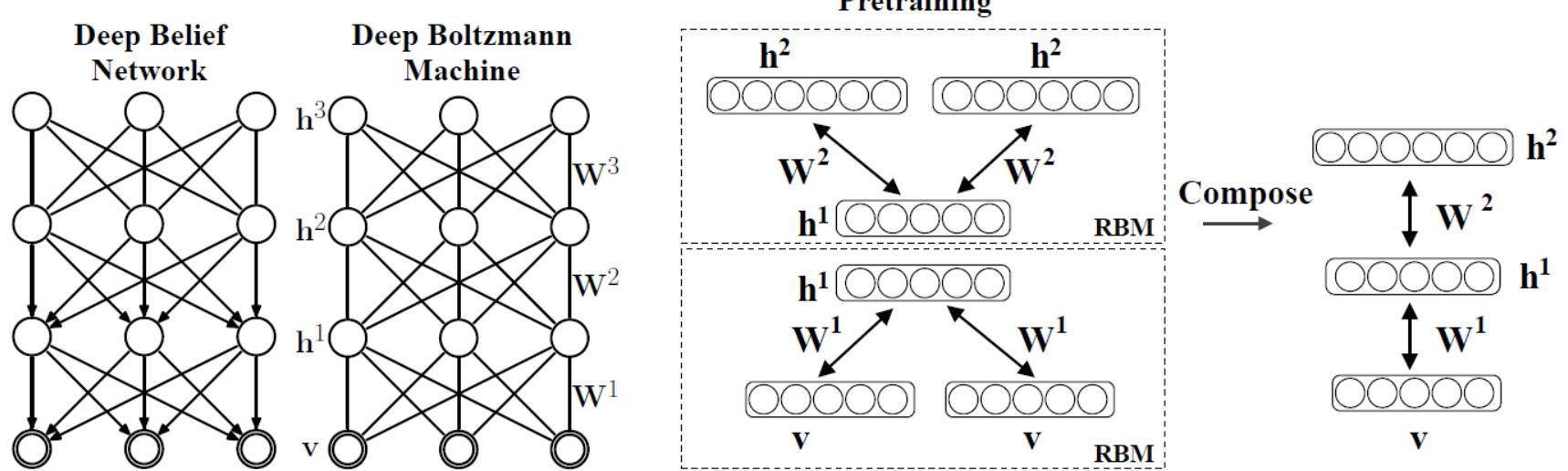


Figure 2: **Left:** A three-layer Deep Belief Network and a three-layer Deep Boltzmann Machine. **Right:** Pretraining consists of learning a stack of modified RBM's, that are then composed to create a deep Boltzmann machine.

# Generative Adversarial Networks

# Generative Adversarial Networks (GANs)

- Originally proposed by Ian Goodfellow in 2014
  - It all started in a pub ☺
- 

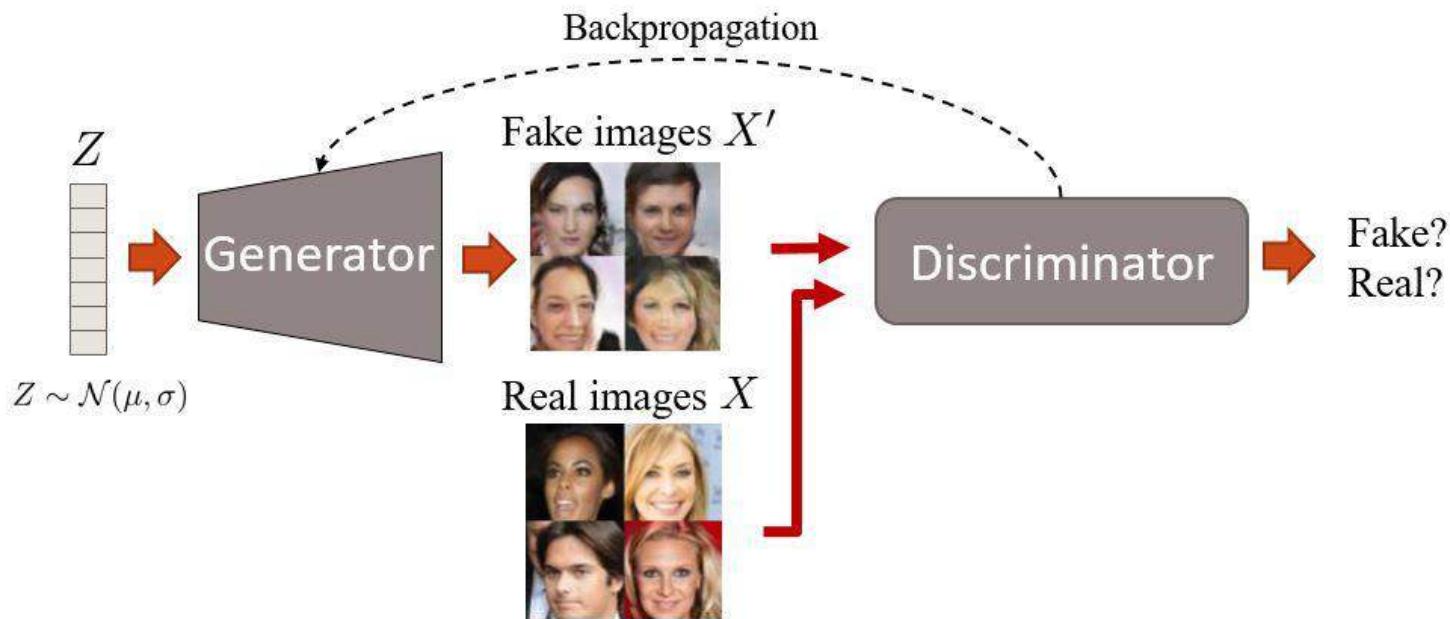
## Generative Adversarial Nets

---

Ian J. Goodfellow, Jean Pouget-Abadie\*, Mehdi Mirza, Bing Xu, David Warde-Farley,  
Sherjil Ozair†, Aaron Courville, Yoshua Bengio‡

Département d'informatique et de recherche opérationnelle  
Université de Montréal  
Montréal, QC H3C 3J7

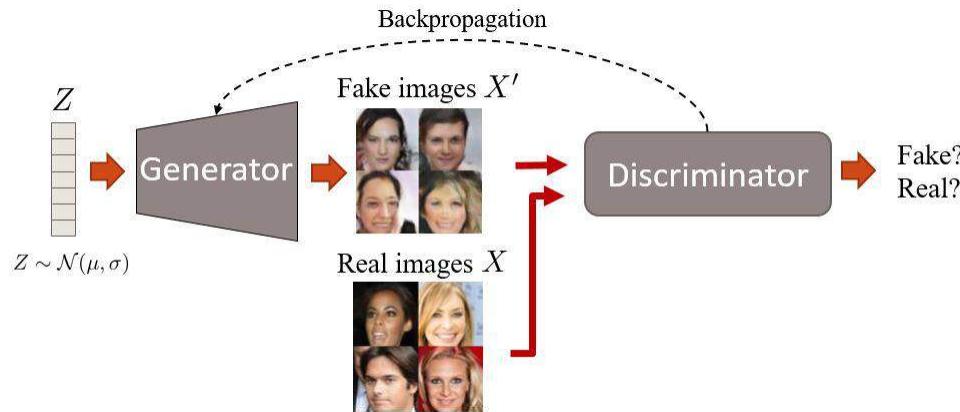
# Generative Adversarial Networks (GANs)



We have two networks:

- Generator (G): Generates a fake image given a noise (embedding) vector ( $z$ )
- Discriminator (D): Discriminates whether an image is fake or real.

# Generative Adversarial Networks (GANs)



- With two competing networks, we solve the following minimax game:

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

$D(x)$ : Probability that  $x$  is real (came from data).

$\log (1 - D(G(z)))$  is minimized by  $G$ .

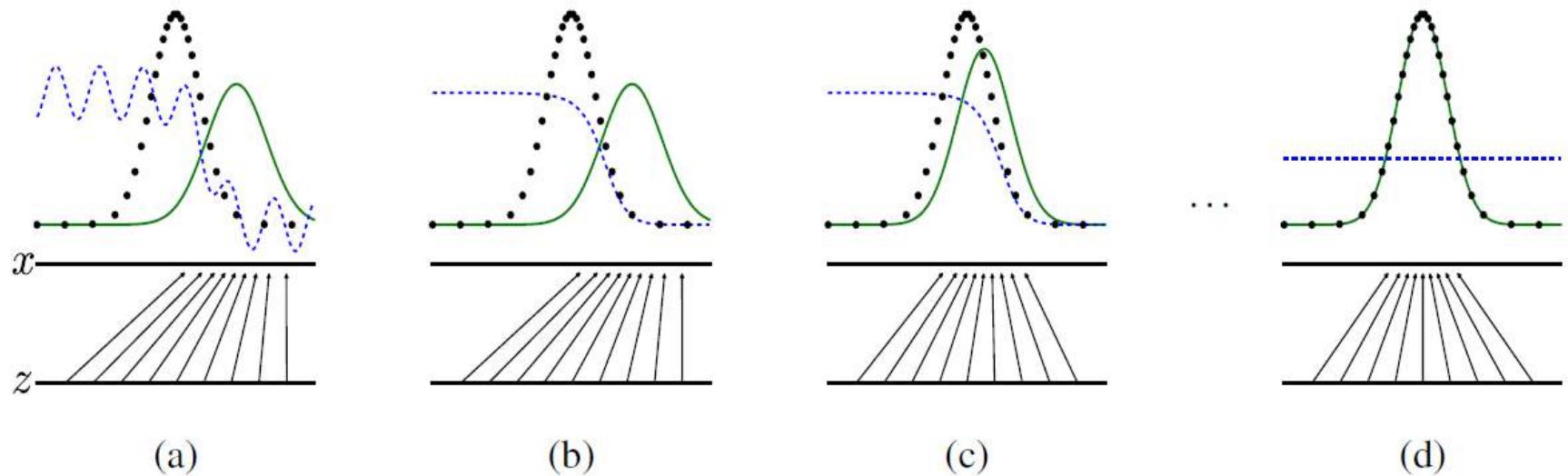


Figure 1: Generative adversarial nets are trained by simultaneously updating the **discriminative distribution** ( $D$ , blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line)  $p_{\text{data}}$  from those of the **generative distribution**  $p_g$  (G) (green, solid line). The lower horizontal line is the domain from which  $z$  is sampled, in this case uniformly. The horizontal line above is part of the domain of  $x$ . The upward arrows show how the mapping  $x = G(z)$  imposes the non-uniform distribution  $p_g$  on transformed samples.  $G$  contracts in regions of high density and expands in regions of low density of  $p_g$ . (a) Consider an adversarial pair near convergence:  $p_g$  is similar to  $p_{\text{data}}$  and  $D$  is a partially accurate classifier. (b) In the inner loop of the algorithm  $D$  is trained to discriminate samples from data, converging to  $D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$ . (c) After an update to  $G$ , gradient of  $D$  has guided  $G(z)$  to flow to regions that are more likely to be classified as data. (d) After several steps of training, if  $G$  and  $D$  have enough capacity, they will reach a point at which both cannot improve because  $p_g = p_{\text{data}}$ . The discriminator is unable to differentiate between the two distributions, i.e.  $D(x) = \frac{1}{2}$ .

---

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

---

**for** number of training iterations **do**

**for**  $k$  steps **do**

- Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
- Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data generating distribution  $p_{\text{data}}(x)$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

**end for**

- Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
- Update the generator by descending its stochastic gradient:

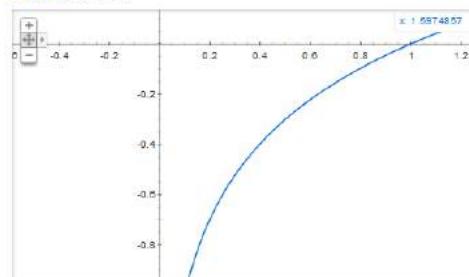
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

**end for**

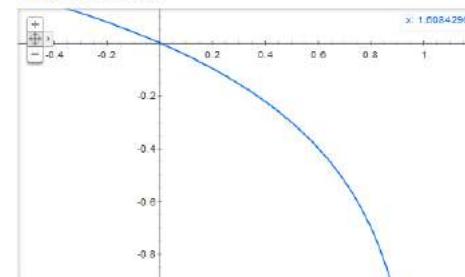
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

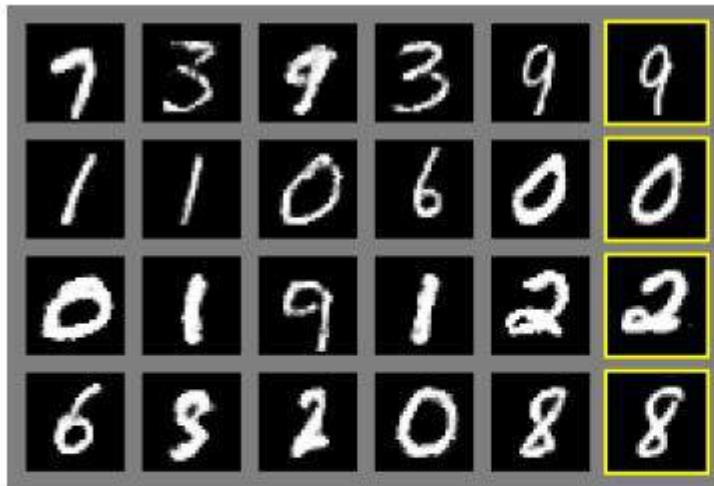
---

log(x) grafiği



log(1-x) grafiği





a)



b)



c)



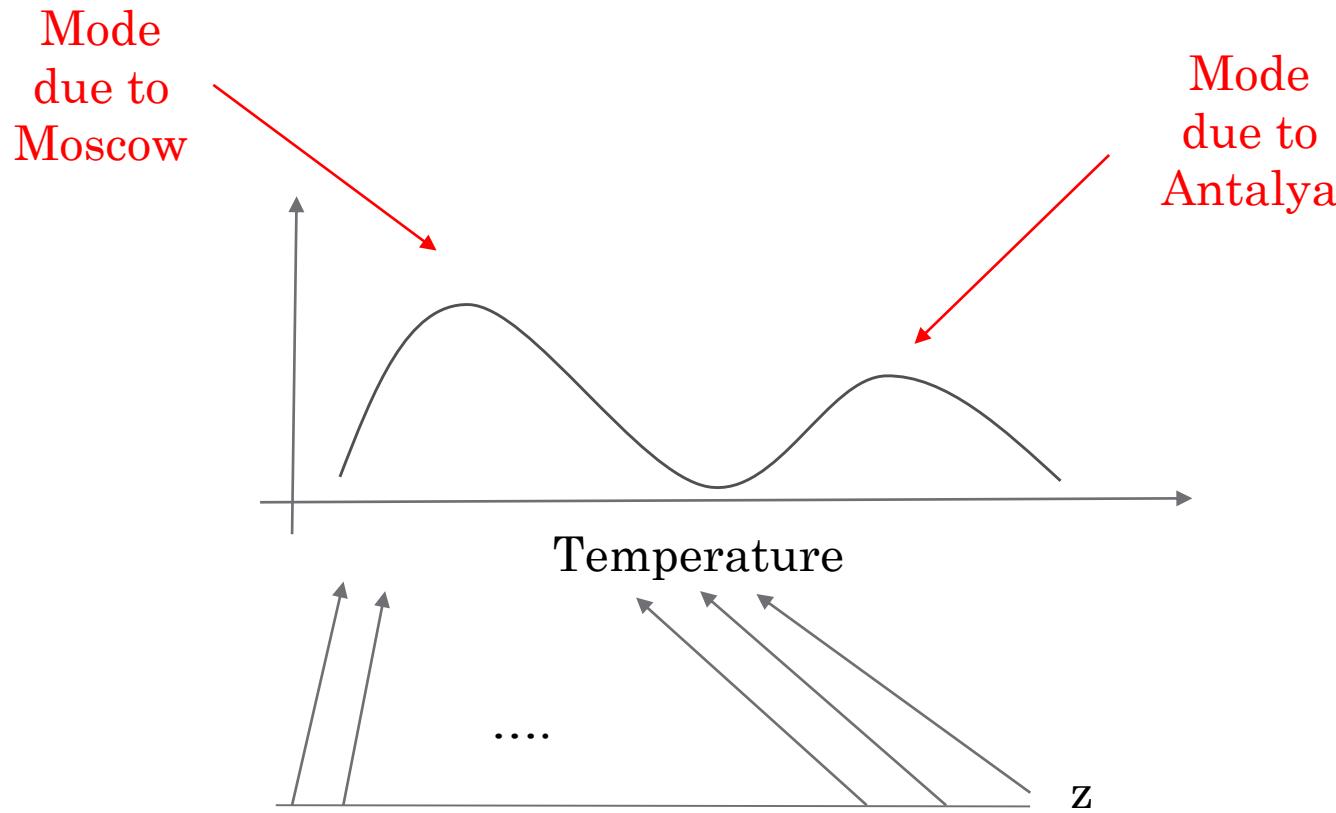
d)

Figure 2: Visualization of samples from the model. Rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. Samples are fair random draws, not cherry-picked. Unlike most other visualizations of deep generative models, these images show actual samples from the model distributions, not conditional means given samples of hidden units. Moreover, these samples are uncorrelated because the sampling process does not depend on Markov chain mixing. a) MNIST b) TFD c) CIFAR-10 (fully connected model) d) CIFAR-10 (convolutional discriminator and “deconvolutional” generator)

Fig: Goodfellow et al., 2014.

# Mode collapse in GANs

- Problem:
  - The generator network maps the different  $z$  (embedding/noise) values into similar images.



# Mode collapse in GANs

- Solutions:
  - Changing the training procedure (use batch discrimination instead of individual discrimination)
  - Experience replay (show old fake images again and again)
  - Use a different loss (+ enforce diversity)
  - ...

# Deep Convolutional GAN

- GAN with convolutional layers
- More stable

## Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.



## UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS

Alec Radford & Luke Metz

indico Research

Boston, MA

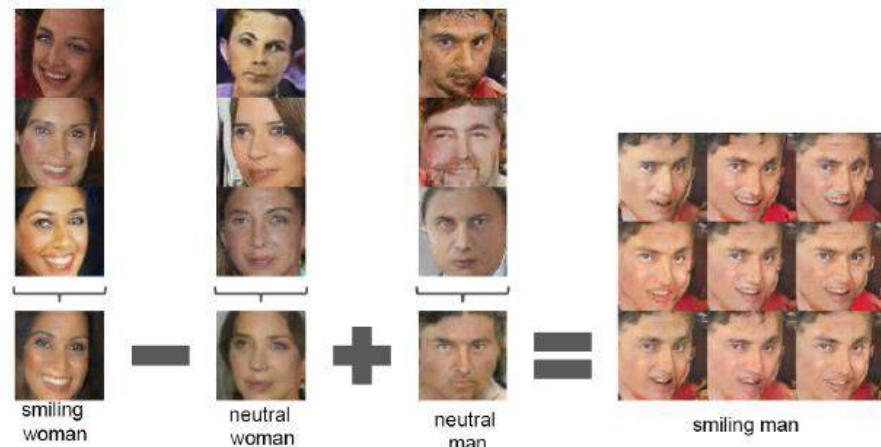
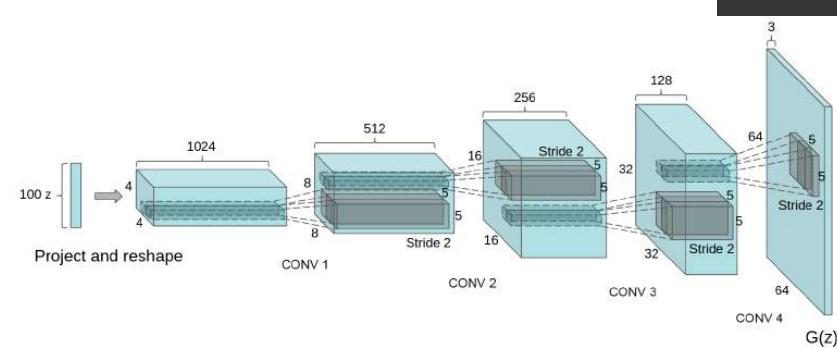
{alec,luke}@indico.io

Soumith Chintala

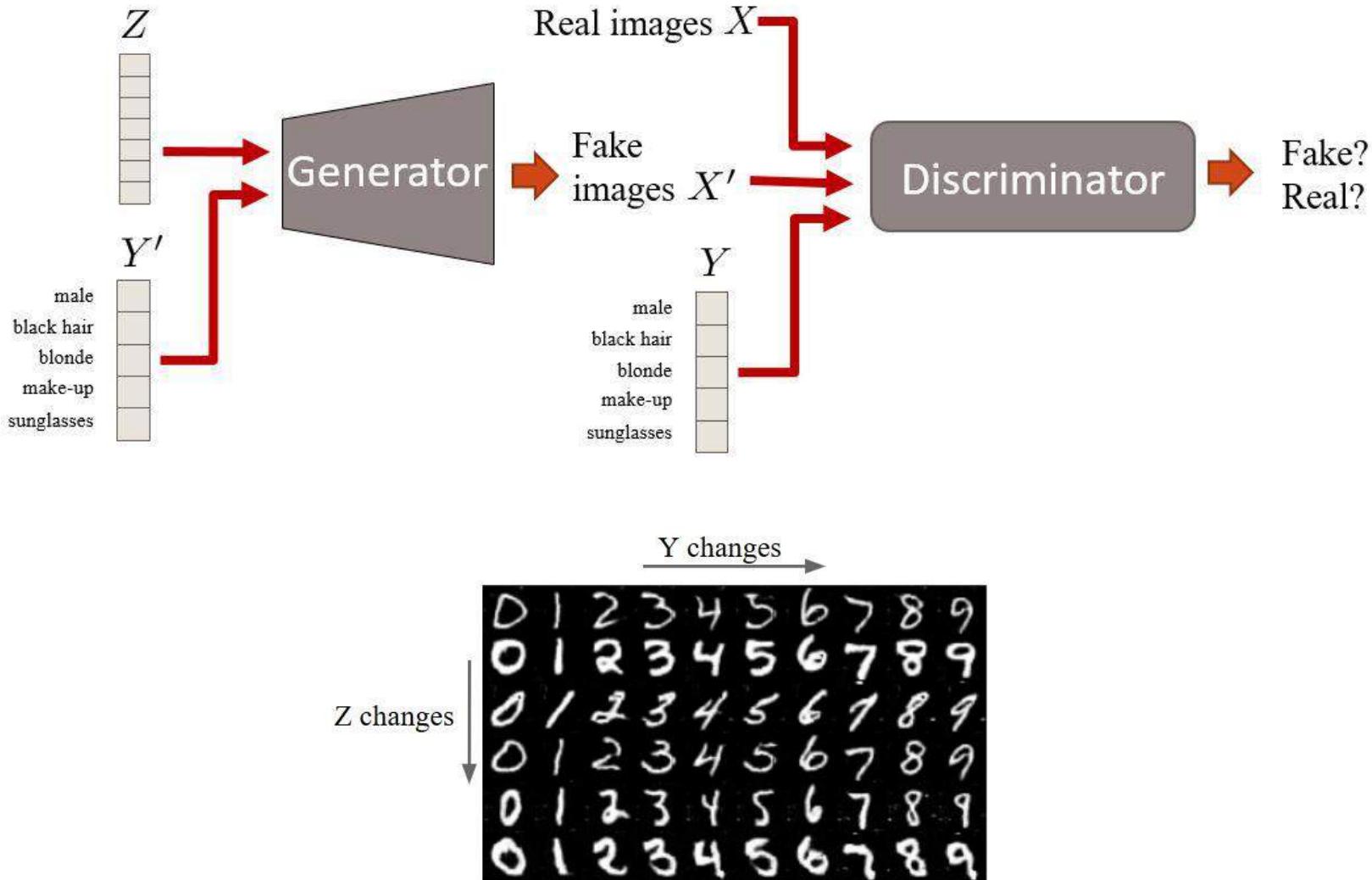
Facebook AI Research

New York, NY

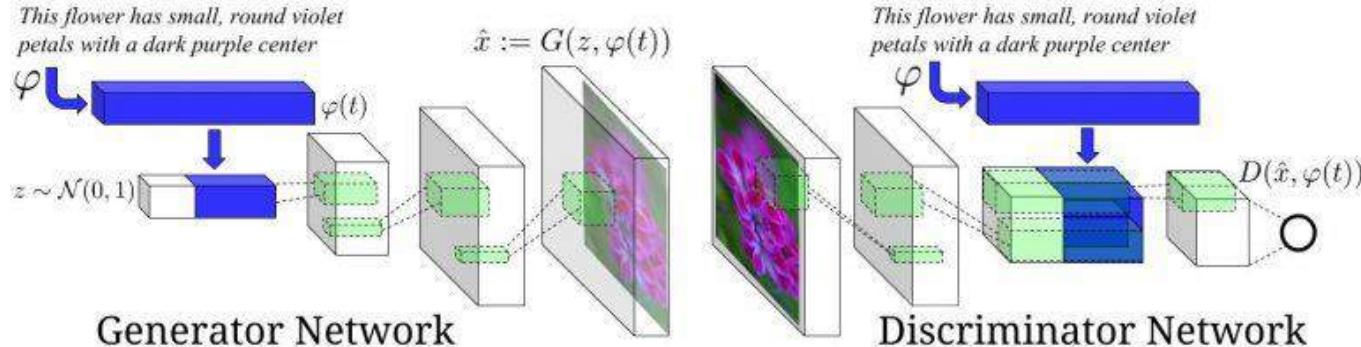
soumith@fb.com



# Conditional GANs



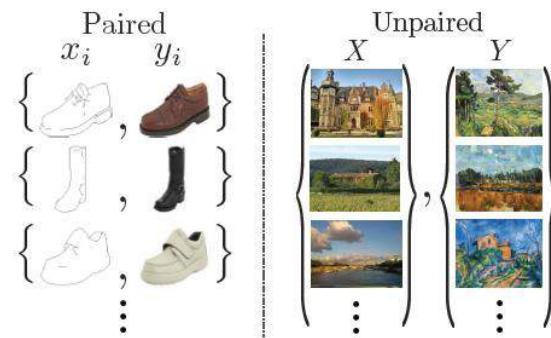
# Text to image with GANs



*(this small bird has a pink breast and crown, and black primaries and secondaries)*



# Cycle GAN



## Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Jun-Yan Zhu\*      Taesung Park\*      Phillip Isola      Alexei A. Efros  
Berkeley AI Research (BAIR) laboratory, UC Berkeley

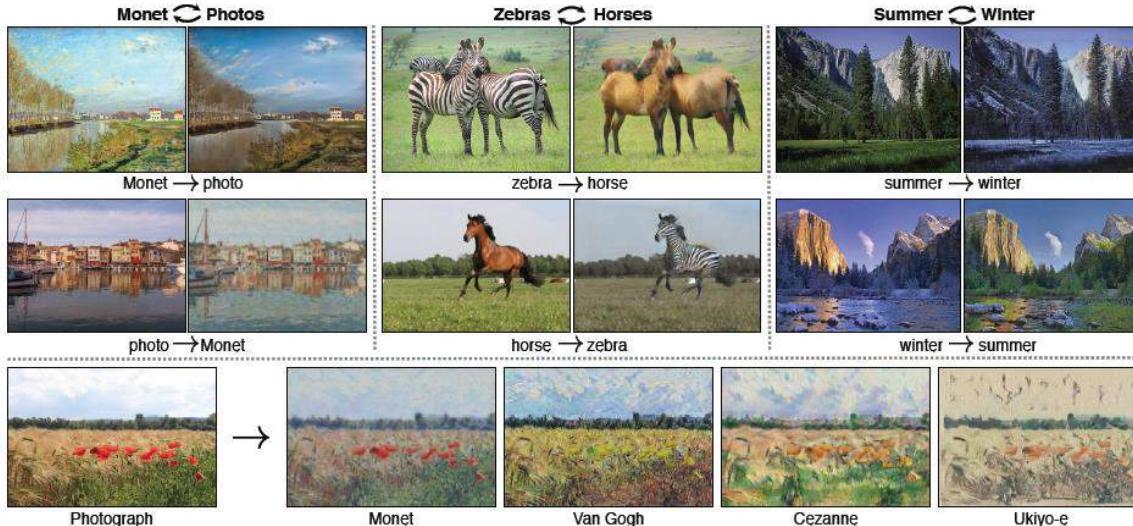


Figure 1: Given any two unordered image collections  $X$  and  $Y$ , our algorithm learns to automatically “translate” an image from one into the other and vice versa: (*left*) 1074 Monet paintings and 6753 landscape photos from Flickr; (*center*) 1177 zebras and 939 horses from ImageNet; (*right*) 1273 summer and 854 winter Yosemite photos from Flickr. Example application (*bottom*): using a collection of paintings of a famous artist, learn to render a user’s photograph into their style.

<https://junyanz.github.io/CycleGAN/>

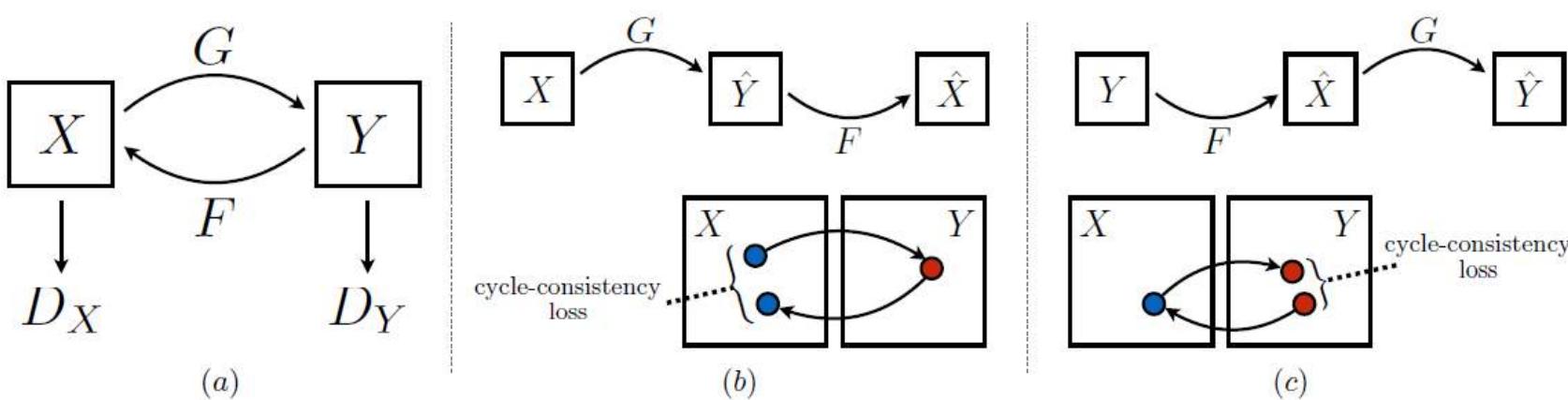


Figure 3: (a) Our model contains two mapping functions  $G : X \rightarrow Y$  and  $F : Y \rightarrow X$ , and associated adversarial discriminators  $D_Y$  and  $D_X$ .  $D_Y$  encourages  $G$  to translate  $X$  into outputs indistinguishable from domain  $Y$ , and vice versa for  $D_X$ ,  $F$ , and  $X$ . To further regularize the mappings, we introduce two “cycle consistency losses” that capture the intuition that if we translate from one domain to the other and back again we should arrive where we started: (b) forward cycle-consistency loss:  $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$ , and (c) backward cycle-consistency loss:  $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

$$\begin{aligned} \mathcal{L}(G, F, D_X, D_Y) = & \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ & + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ & + \lambda \mathcal{L}_{\text{cyc}}(G, F), \end{aligned}$$

$$\begin{aligned} \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = & \mathbb{E}_{y \sim p_{\text{data}}(y)} [\log D_Y(y)] \\ & + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log(1 - D_Y(G(x)))] \end{aligned}$$

$$\begin{aligned} \mathcal{L}_{\text{cyc}}(G, F) = & \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(G(x)) - x\|_1] \\ & + \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(F(y)) - y\|_1]. \end{aligned}$$

# Cycle GAN

Kış



Yaz



<https://junyanz.github.io/CycleGAN/>

<https://www.digitaltrends.com/cool-tech/nvidia-ai-winter-summer-car/>



# The zoo of GANs

- <https://deephunt.in/the-gan-zoo-79597dc8c347>

# Variational-AE

# Variational Inference

- Why need VI?
  - Intractability
    1. *Intractability:* the case where the integral of the marginal likelihood  $p_{\theta}(\mathbf{x}) = \int p_{\theta}(\mathbf{z})p_{\theta}(\mathbf{x}|\mathbf{z}) d\mathbf{z}$  is intractable (so we cannot evaluate or differentiate the marginal likelihood), where the true posterior density  $p_{\theta}(\mathbf{z}|\mathbf{x}) = p_{\theta}(\mathbf{x}|\mathbf{z})p_{\theta}(\mathbf{z})/p_{\theta}(\mathbf{x})$  is intractable (so the EM algorithm cannot be used), and where the required integrals for any reasonable mean-field VB algorithm are also intractable. These intractabilities are quite common and appear in cases of moderately complicated likelihood functions  $p_{\theta}(\mathbf{x}|\mathbf{z})$ , e.g. a neural network with a nonlinear hidden layer.
    2. *A large dataset:* we have so much data that batch optimization is too costly; we would like to make parameter updates using small minibatches or even single datapoints. Sampling-based solutions, e.g. Monte Carlo EM, would in general be too slow, since it involves a typically expensive sampling loop per datapoint.

Diederik P. Kingma  
Machine Learning Group  
Universiteit van Amsterdam  
dpkingma@gmail.com

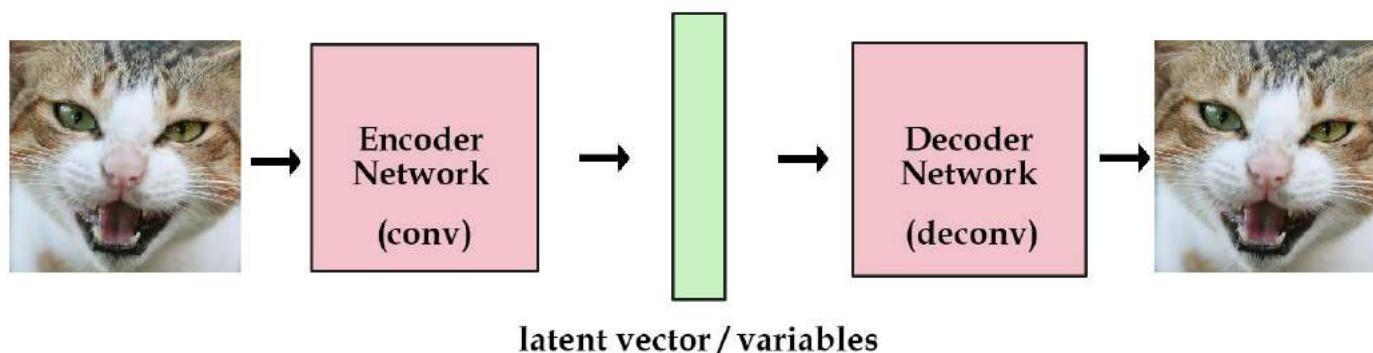
Max Welling  
Machine Learning Group  
Universiteit van Amsterdam  
welling.max@gmail.com

2013

- $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1}^N$ : The dataset consisting of  $N$  i.i.d. samples of continuous or discrete variable  $\mathbf{x}$ .
- We assume that the data is generated by some random process, involving unobserved continuous random variable  $\mathbf{z}$ .
  1. A random  $\mathbf{z}^{(i)}$  is generated from some prior distribution  $p_{\theta^*}(\mathbf{z})$ .
  2. A value  $\mathbf{x}^{(i)}$  is generated from a conditional distribution  $p_{\theta^*}(\mathbf{x} | \mathbf{z})$ .
- **Condition:**  $p_{\theta^*}(\mathbf{z})$  and  $p_{\theta^*}(\mathbf{x} | \mathbf{z})$  are differentiable ( $\theta^*$  are parameters).
- **Limitation:**  $\mathbf{z}^{(i)}$  and parameters  $\theta^*$  (true parameters) are unknown.
- **Problem:**  $p_{\theta}(\mathbf{x})$  and  $p_{\theta}(\mathbf{z} | \mathbf{x}) = p_{\theta}(\mathbf{x} | \mathbf{z})p_{\theta}(\mathbf{z})/p_{\theta}(\mathbf{x})$  are intractable.

## Solution:

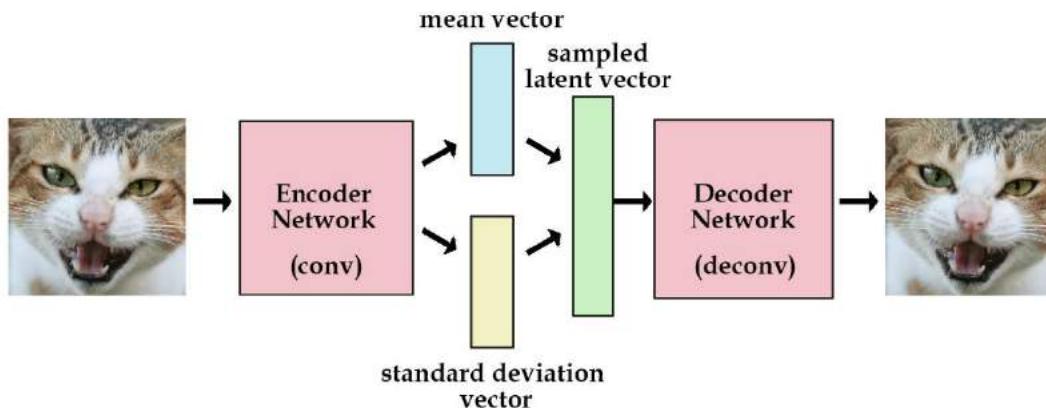
- Replace intractable true posterior  $p_\theta(\mathbf{z} | \mathbf{x})$  with a recognition model  $q_\phi(\mathbf{z} | \mathbf{x})$ .
- $q_\phi(\mathbf{z} | \mathbf{x})$ : probabilistic encoder. Produces a probability distribution over  $\mathbf{z}$  given  $\mathbf{x}$ .
- $p_\theta(\mathbf{x} | \mathbf{z})$ : probabilistic decoder. Produces a probability distribution over  $\mathbf{x}$  given  $\mathbf{z}$ .



# A practical issue

- Problematic with very high variance and impractical
- Solution: Reparameterize  $\tilde{z} \sim q_\phi(z|x)$  using a differentiable transformation  $g_\phi(\epsilon, x)$  with an auxiliary noise variable  $\epsilon$ :

$$\tilde{z} = g_\phi(\epsilon, x) \quad \text{with} \quad \epsilon \sim p(\epsilon)$$



# Auto-Encoding Variational Bayes

Diederik P. Kingma  
Machine Learning Group  
Universiteit van Amsterdam  
dpkingma@gmail.com

Max Welling  
Machine Learning Group  
Universiteit van Amsterdam  
welling.max@gmail.com 2013

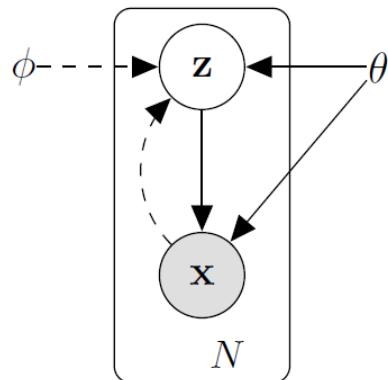


Figure 1: The type of directed graphical model under consideration. Solid lines denote the generative model  $p_{\theta}(z)p_{\theta}(x|z)$ , dashed lines denote the variational approximation  $q_{\phi}(z|x)$  to the intractable posterior  $p_{\theta}(z|x)$ . The variational parameters  $\phi$  are learned jointly with the generative model parameters  $\theta$ .

# Training

- How can we know  $q_\lambda(z|x)$  approximates  $p(z|x)$  well?

$$\begin{aligned} KL(q_\lambda(z|x) \parallel p(z|x)) &= \\ E_q[\log q_\lambda(z|x)] - E_q[\log p(x, z)] + \log p(x) \end{aligned}$$

- Goal: find parameters  $\lambda$  that minimizes this divergence.
- However, this is impossible to compute because of  $p(x)$
- Let us re-write the equation:

$$\log p(x) = \underbrace{-E_q[\log q_\lambda(z|x)] + E_q[\log p(x, z)]}_{\text{Evidence Lower Bound (ELBO)}} + KL(q_\lambda(z|x) \parallel p(z|x))$$

- KL divergence is always greater than or equal to zero
- This means that minimizing KL divergence is equivalent to maximizing the ELBO term (note that  $p(x)$  is constant given the dataset)

- ELBO can be re-written as follows for a single data point:

$$ELBO_i(\lambda) = E_{q_\lambda(z|x_i)}[\log p(x_i|z)] - KL(q_\lambda(z|x_i) \parallel p(z))$$

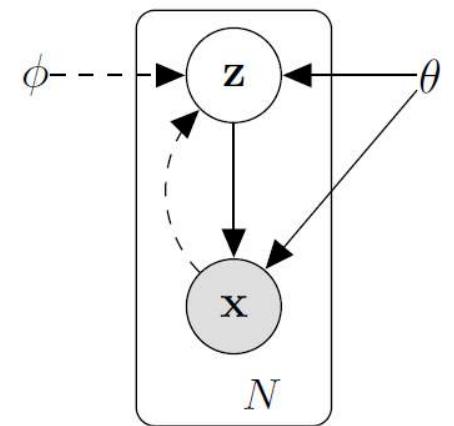
- In the neural net language, this becomes:

$$ELBO_i(\lambda) = E_{q_\theta(z|x_i)}[\log p_\phi(x_i|z)] - KL(q_\theta(z|x_i) \parallel p(z))$$

- ELBO is the negative of the loss function:

$$ELBO_i(\lambda) = -L_i(\theta, \phi)$$

$$= - \left( \underbrace{E_{q_\theta(z|x_i)}[\log p_\phi(x_i|z)]}_{\text{reconstruction loss}} - \underbrace{KL(q_\theta(z|x_i) \parallel p(z))}_{\text{regularizer}} \right)$$



# Resources

- <https://jaan.io/what-is-variational-autoencoder-vae-tutorial/>
- <http://kvfrans.com/variational-autoencoders-explained/>
- <https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf>
- <https://arxiv.org/pdf/1606.05908.pdf>

# Autoregressive Generative Models

# Autoregressive models

- Used in time-series analysis/modeling
- Predict  $x(t)$  from  $x(t - 1), x(t - 2), \dots, x(t - n)$  – e.g., as follows
  - $x(t) = \alpha_0 + \alpha_1 x(t - 1) + \alpha_2 x(t - 2) + \dots + \alpha_n x(t - n) + \epsilon_t$
  - This is an  $n^{th}$  order autoregressor
- This essentially corresponds to modeling  $p(x)$  as follows:

$$p(x) = p(x_0) \prod_{i=1}^n p(x_i | x_{i<})$$

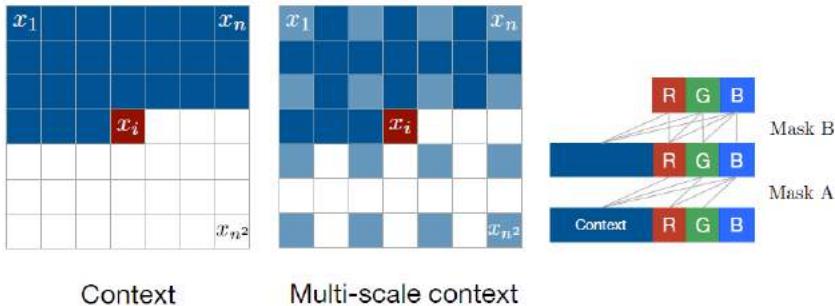
# PixelCNN & PixelRNN

Aäron van den Oord  
Nal Kalchbrenner  
Koray Kavukcuoglu  
Google DeepMind

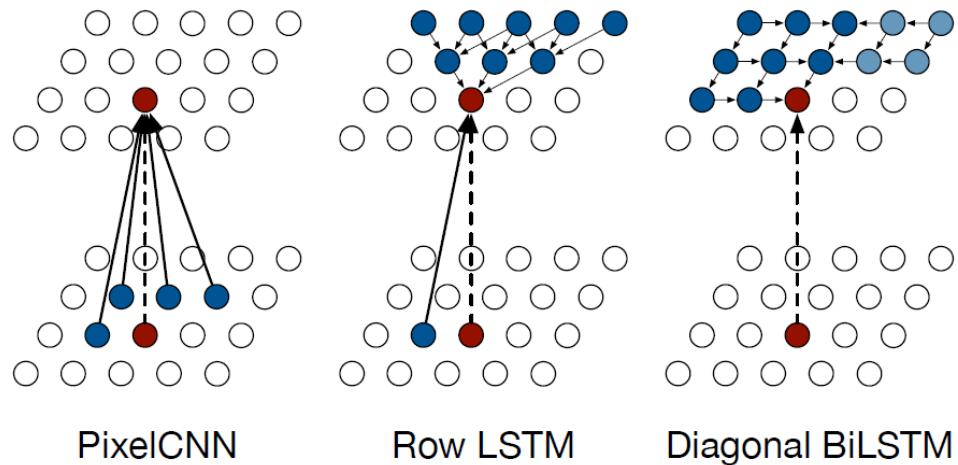
AVDNOORD@GOOGLE.COM  
NALK@GOOGLE.COM  
KORAYK@GOOGLE.COM

2016

- Estimate the next pixel given all the pixels up to that point.



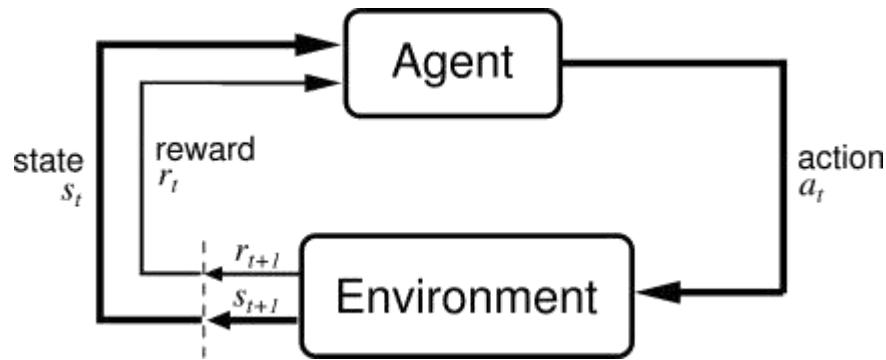
**Figure 2. Left:** To generate pixel  $x_i$  one conditions on all the previously generated pixels left and above of  $x_i$ . **Center:** To generate a pixel in the multi-scale case we can also condition on the subsampled image pixels (in light blue). **Right:** Diagram of the connectivity inside a masked convolution. In the first layer, each of the RGB channels is connected to previous channels and to the context, but is not connected to itself. In subsequent layers, the channels are also connected to themselves.



**Figure 4.** Visualization of the input-to-state and state-to-state mappings for the three proposed architectures.

# Deep Reinforcement Learning

# Reinforcement Learning



The agent receives reward  $r_t$  for its actions.

# More formally

- An agent's behavior is defined by a policy,  $\pi$ :

$$\pi: \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$$

$\mathcal{S}$ : The space of states.

$\mathcal{A}$ : The space of actions.

- The “return” from a state is usually:

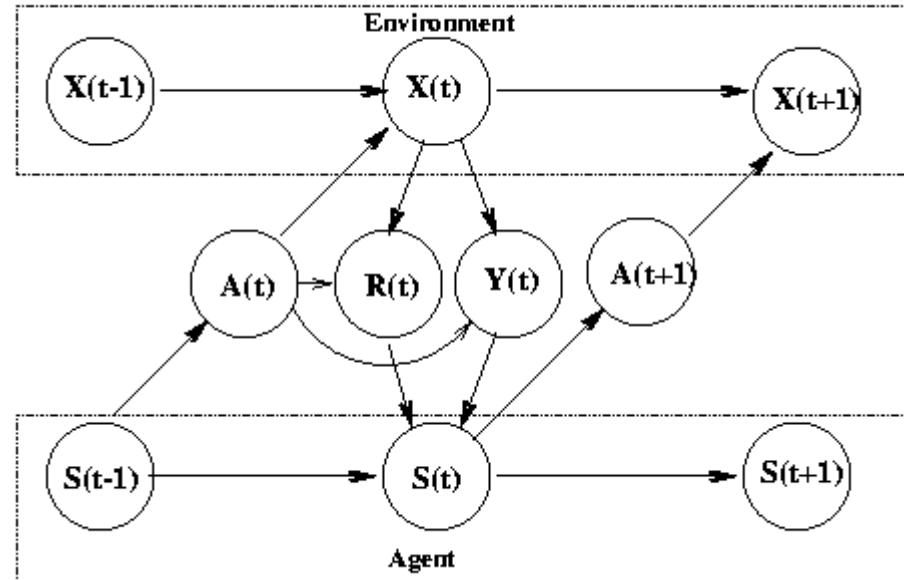
$$R_t = \sum_{i=t}^T \gamma^{(i-t)} r(s_i, a_i)$$

$r(s_i, a_i)$ : the reward for action  $a_i$  in state  $s_i$ .

$\gamma$ : discount factor.

- Goal: Learn a policy that maximizes the expected return from the starting position:

$$\mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]$$



# More formally

- We can define an expected return for taking action  $a_t$  at state  $s_t$ :

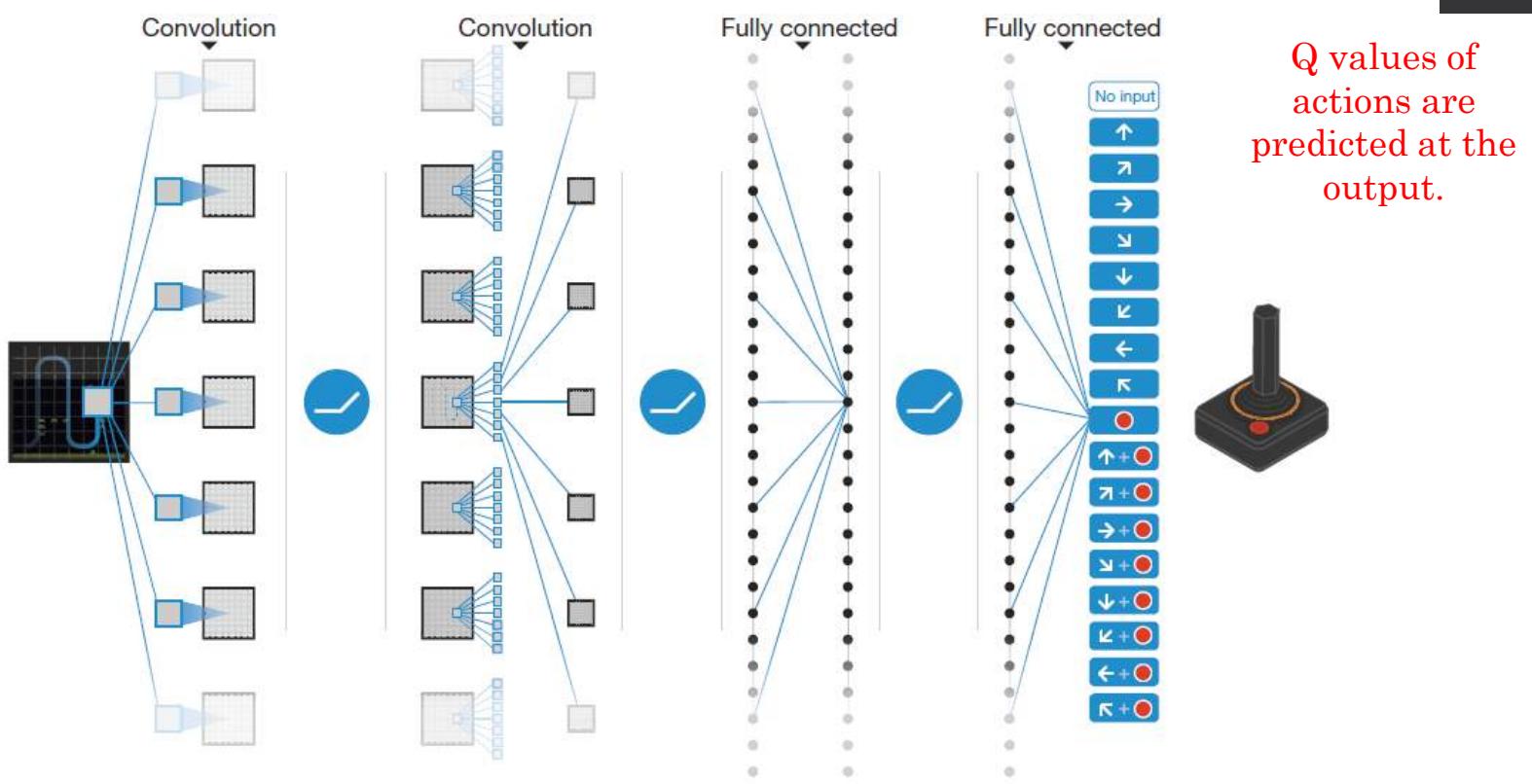
$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i > t} \sim E, a_{i > t} \sim \pi} [R_t | s_t, a_t]$$

- This can be rewritten as (called the Bellman equation):

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E} [r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi} [Q^\pi(s_{t+1}, a_{t+1})]]$$

# Reinforcement Learning in/with Deep Networks

- Two general approaches:
  - Value gradients
  - Policy gradients



**Figure 1 | Schematic illustration of the convolutional neural network.** The details of the architecture are explained in the Methods. The input to the neural network consists of an  $84 \times 84 \times 4$  image produced by the preprocessing map  $\phi$ , followed by three convolutional layers (note: snaking blue line

symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is,  $\max(0, x)$ ).

# LETTER

doi:10.1038/nature14236

## Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

network. We refer to a neural network function approximator with weights  $\theta$  as a Q-network. A Q-network can be trained by adjusting the parameters  $\theta_i$  at iteration  $i$  to reduce the mean-squared error in the Bellman equation, where the optimal target values  $r + \gamma \max_{a'} Q^*(s', a')$  are substituted with approximate target values  $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ , using parameters  $\theta_i^-$  from some previous iteration. This leads to a sequence of loss functions  $L_i(\theta_i)$  that changes at each iteration  $i$ ,

$$L_i(\theta_i) = \mathbb{E}_{s,a,r} [(\mathbb{E}_{s'}[y|s,a] - Q(s,a;\theta_i))^2]$$

## LETTER

doi:10.1038/nature14236

# Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

**Algorithm 1: deep Q-learning with experience replay.**

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

        With probability  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$

        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

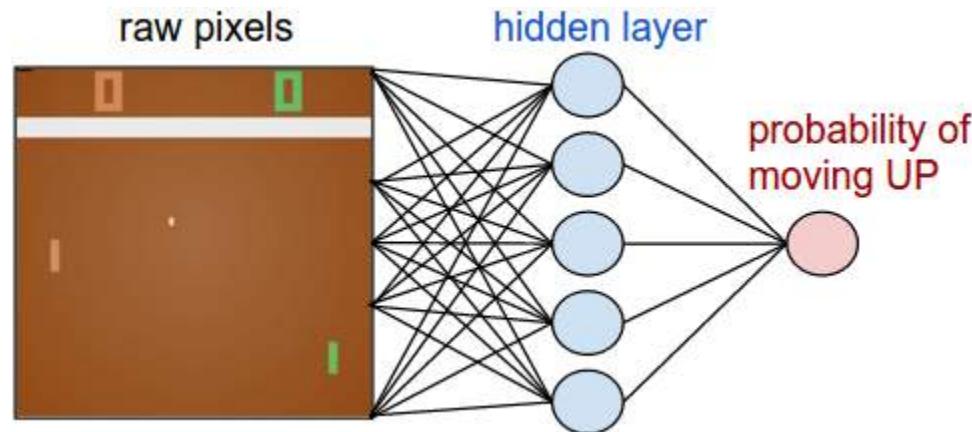
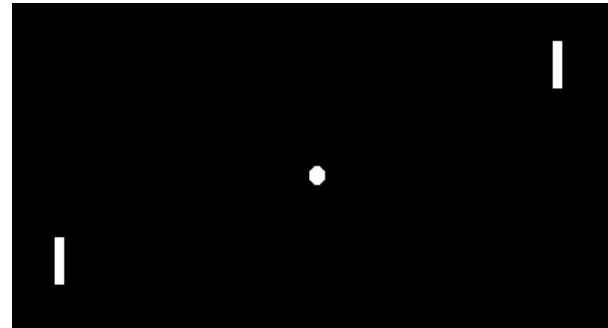
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

        Every C steps reset  $\hat{Q} = Q$

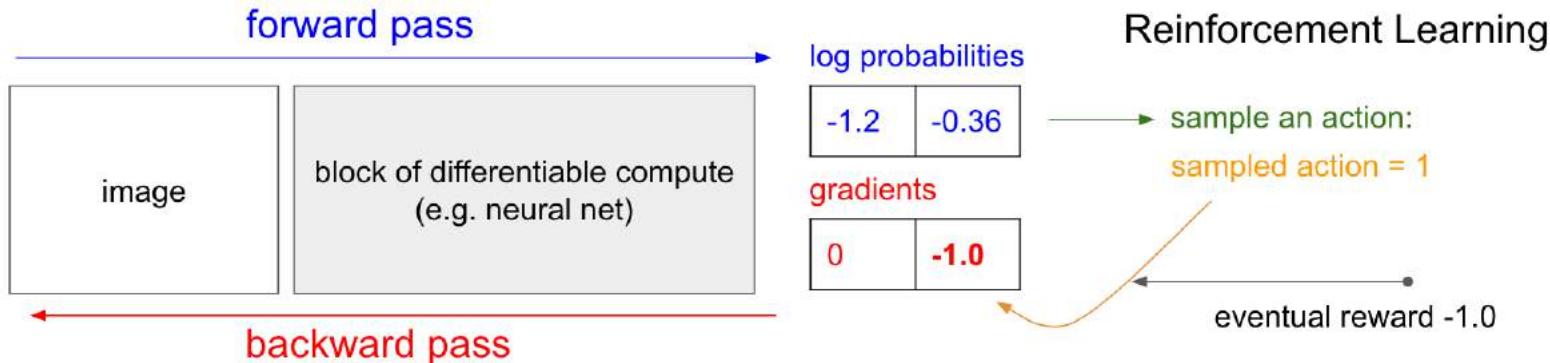
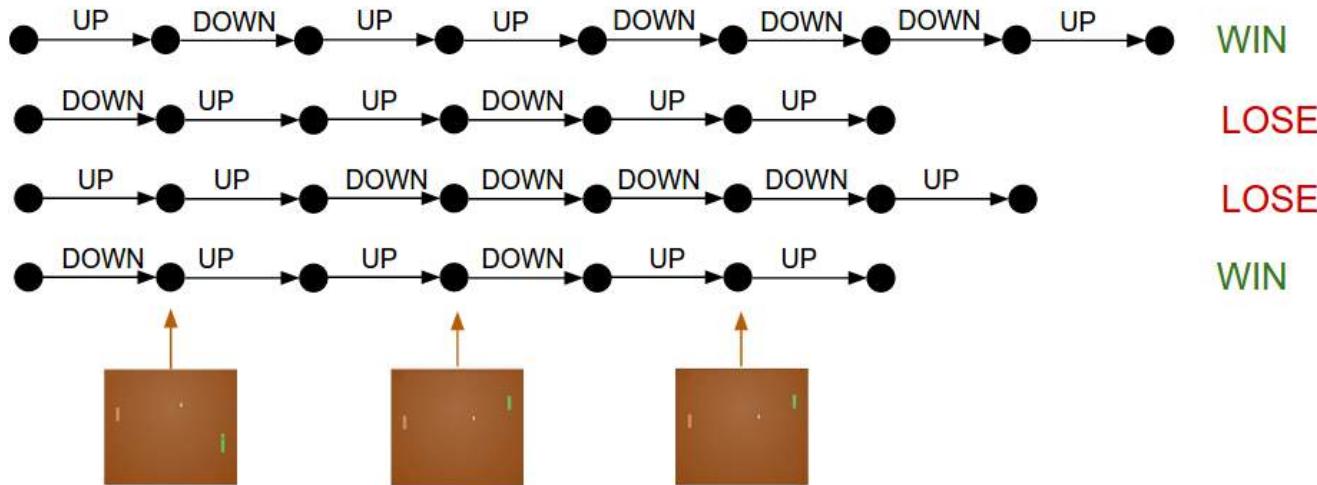
**End For**

**End For**

# Policy gradients

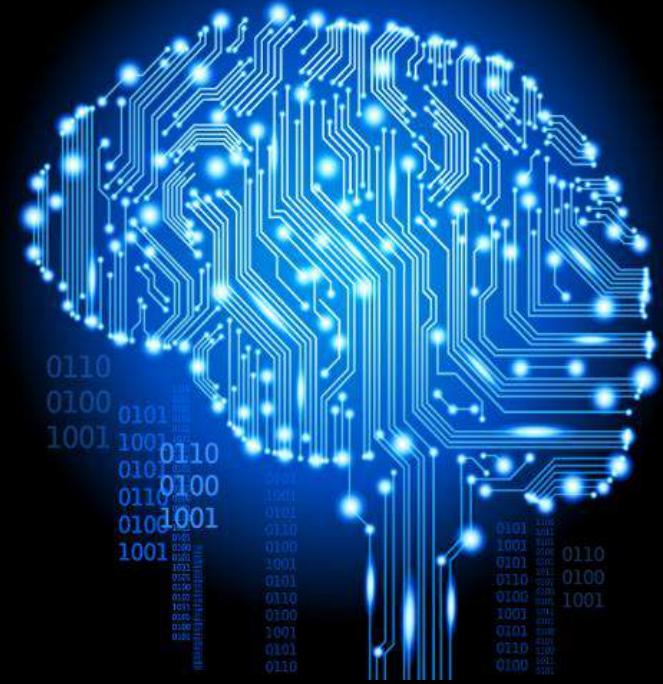


# Policy gradients



# CENG 783

# Deep Learning



© AlchemyAPI

*Week – 6*  
*Convolutional Neural Networks*

Sinan Kalkan



# Motivation

- A fully-connected network has too many parameters
  - On CIFAR-10:
    - Images have size 32x32x3 → one neuron in hidden layer has 3072 weights!
    - With images of size 512x512x3 → one neuron in hidden layer has 786,432 weights!
    - This explodes quickly if you increase the number of neurons & layers.
  - Alternative: enforce local connectivity!

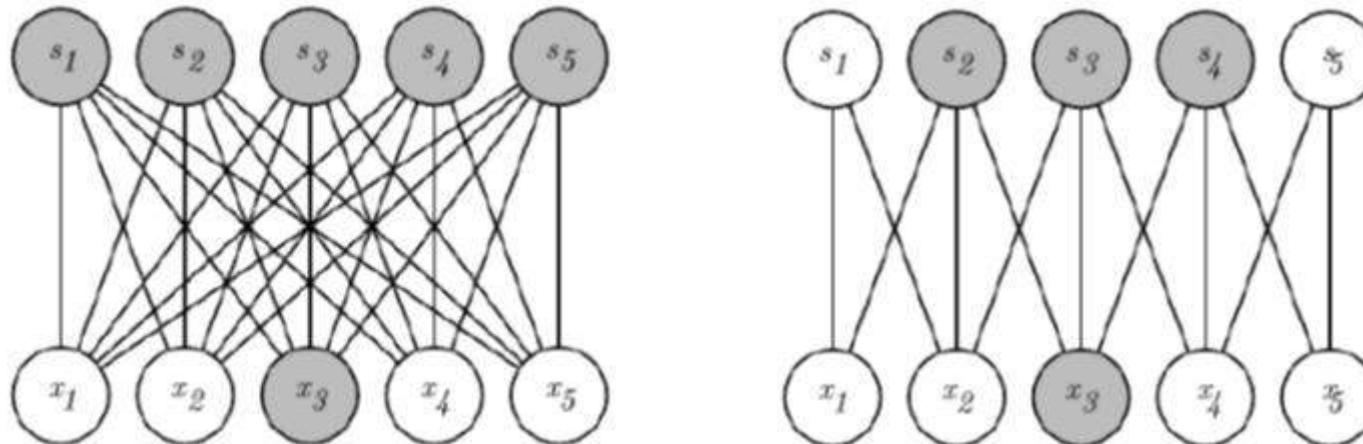
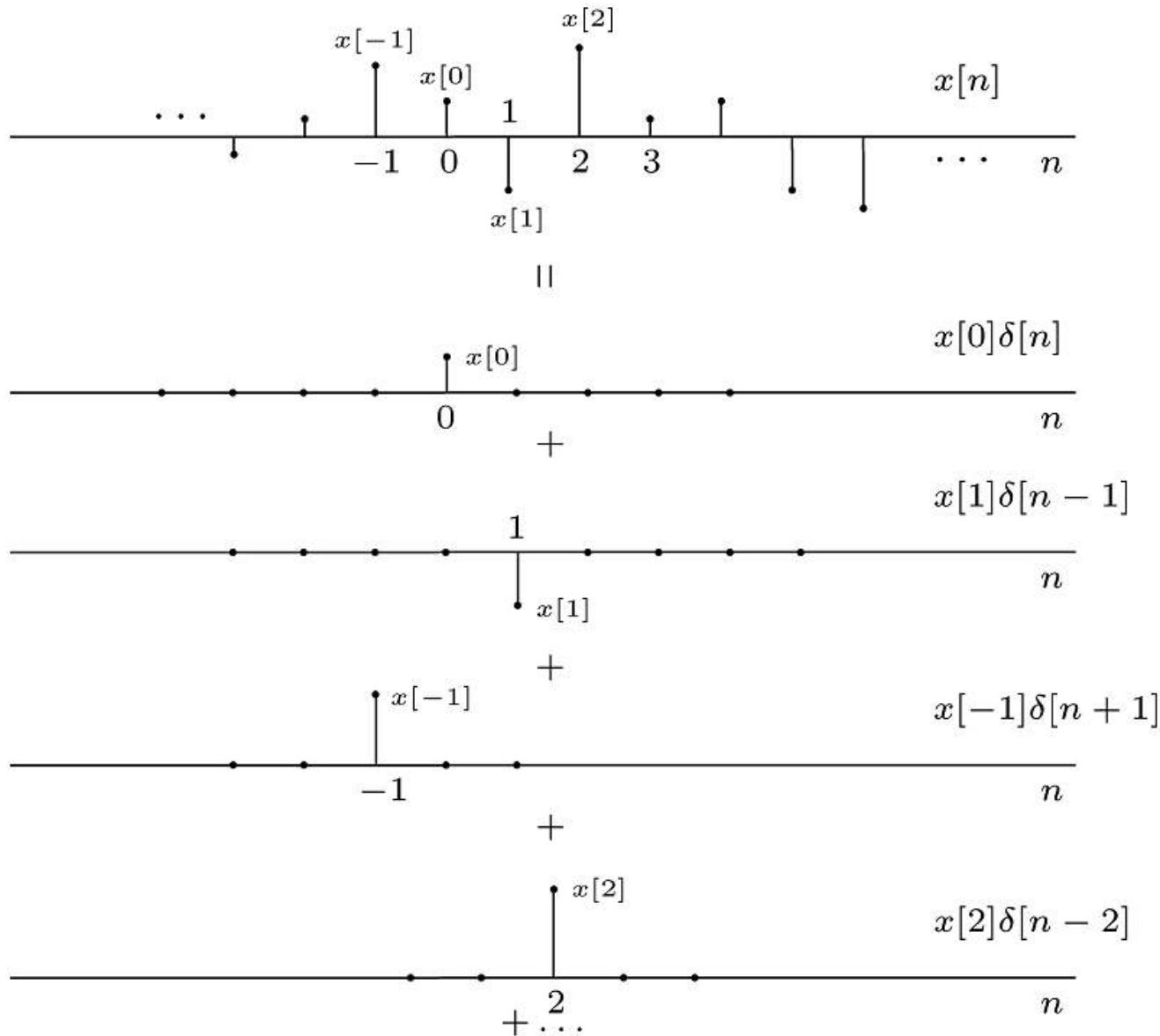


Figure: Goodfellow et al., “Deep Learning”, MIT Press, 2016.

# Convolution

# Formulating Signals in Terms of Impulse Signal



# Formulating Signals in Terms of Impulse Signal

$$x[n] = \dots + x[-2]\delta[n+2] + x[-1]\delta[n+1] + x[0]\delta[n] + x[1]\delta[n-1] + \dots$$

$$x[n] = \sum_{k=-\infty}^{+\infty} \underbrace{x[k]}_{\text{Coefficients}} \underbrace{\delta[n-k]}_{\text{Basic Signals}}$$

Important to note  
the “-” sign

## The Sifting Property of the Unit Sample

To sift: Eleme<sup>k</sup>

# Unit Sample Response

- Now suppose the system is **LTI**, and define the *unit sample response*  $h[n]$ :

$$\delta[n] \longrightarrow h[n]$$
$$\Downarrow$$

From **Time-Invariance**:

$$\delta[n - k] \longrightarrow h[n - k]$$

From **Linearity**:

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k] \delta[n - k] \longrightarrow y[n] = \underbrace{\sum_{k=-\infty}^{+\infty} x[k] h[n - k]}_{\text{convolution sum}} = x[n] * h[n]$$

# Conclusion

The output of *any* DT LTI System is a convolution of the input signal with the unit-sample response, *i.e.*

$$\text{Any DT LTI} \longleftrightarrow y[n] = x[n] * h[n]$$

$$= \sum_{k=-\infty}^{+\infty} x[k] h[n - k]$$

As a result, any DT LTI Systems are *completely characterized* by its unit sample response

# Power of convolution

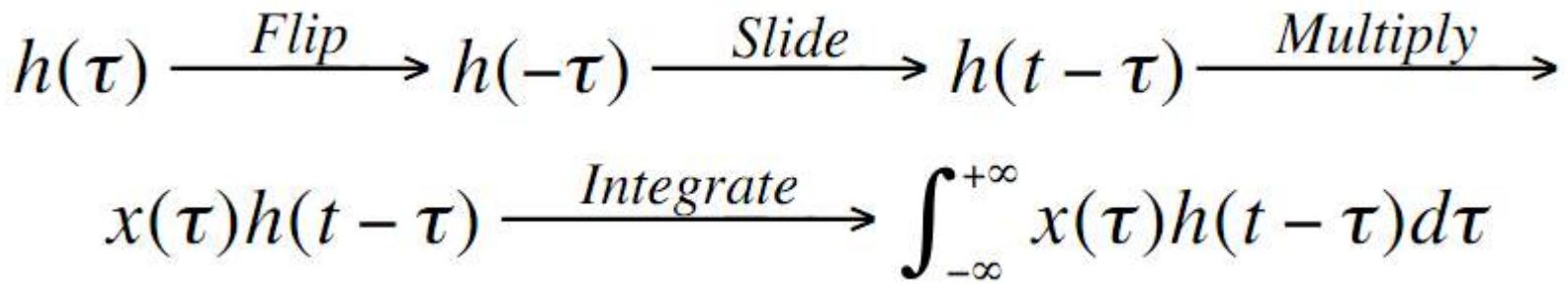
- Describe a “system” (or operation) with a very simple function (impulse response).
- Determine the output by convolving the input with the impulse response

# Convolution

- Definition of continuous-time convolution

$$x(t) * h(t) = \int x(\tau)h(t - \tau) d\tau$$

$$y(t) = x(t) * h(t) \equiv \int_{-\infty}^{+\infty} x(\tau)h(t - \tau)d\tau$$



# Convolution

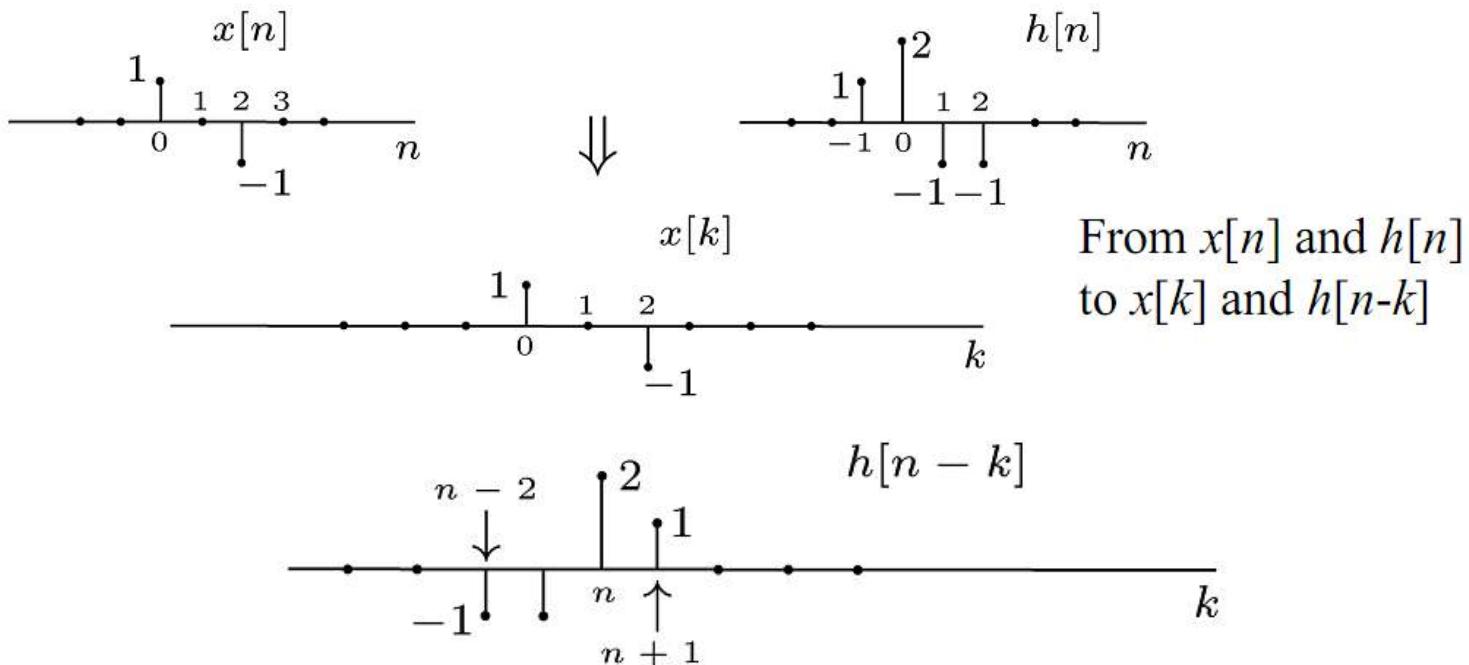
- Definition of discrete-time convolution

$$x[n] * h[n] = \sum x[k]h[n - k]$$

Choose the value of  $n$  and consider it fixed

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n - k]$$

↑                              ↑  
View as functions of  $k$  with  $n$  fixed



# Discrete-time 2D Convolution

- For images, we need two-dimensional convolution:

$$s[i, j] = (I * K)[i, j] = \sum_m \sum_n I[m, n] K[i - m, j - n]$$

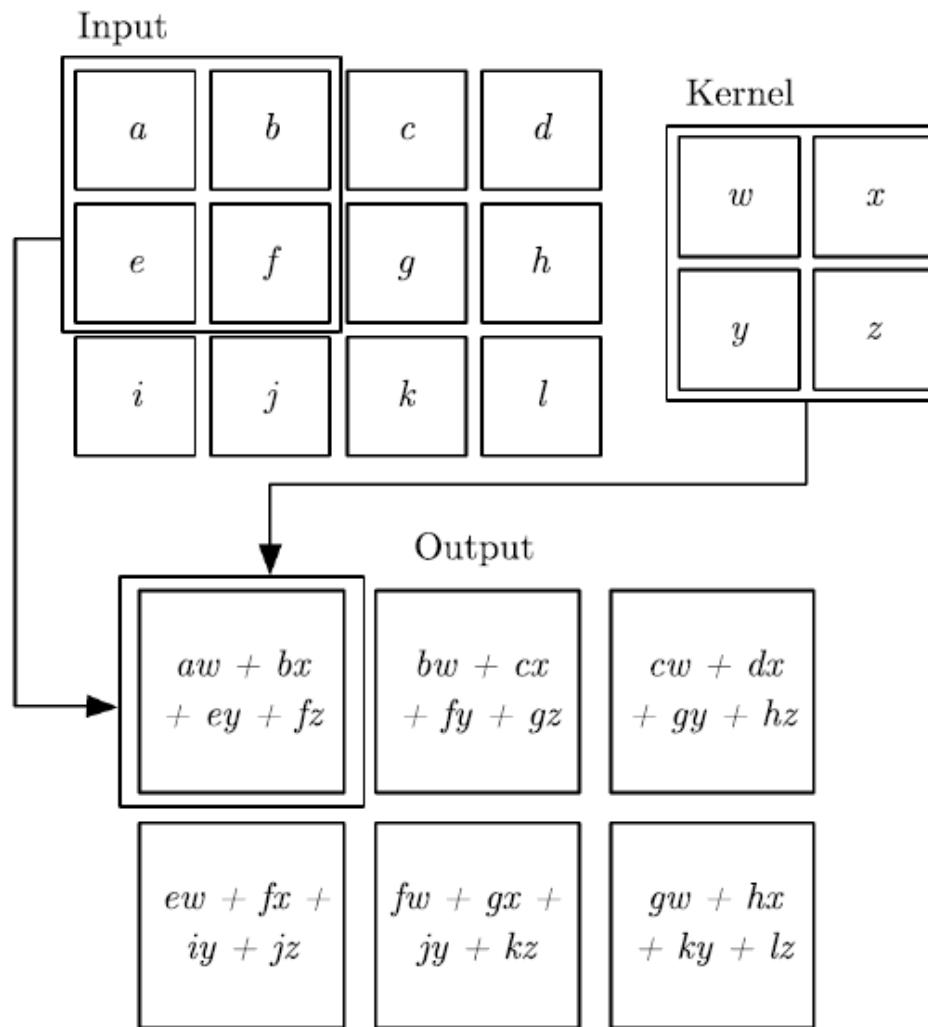
- These multi-dimensional arrays are called tensors
- We have commutative property:

$$s[i, j] = (I * K)[i, j] = \sum_m \sum_n I[i - m, j - n] K[m, n]$$

- Instead of subtraction, we can also write (easy to derive by a change of variables). This is called cross-correlation:

$$s[i, j] = (I * K)[i, j] = \sum_m \sum_n I[i + m, j + n] K[m, n]$$

# Example multi-dimensional convolution



# Filtering

- Convolutional
  - Dependencies are local
  - Translation equivariance
  - Tied filter weights (few params)
  - Stride 1,2,... (faster, less mem.)



Input

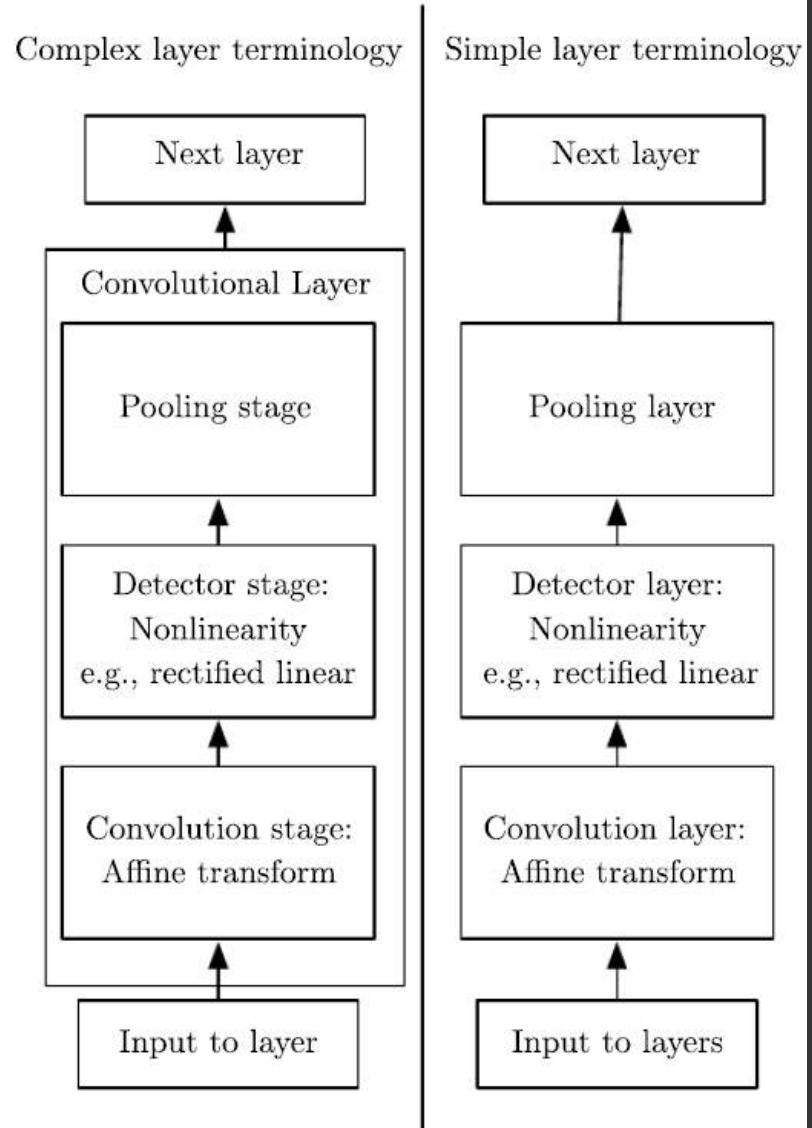


Feature Map

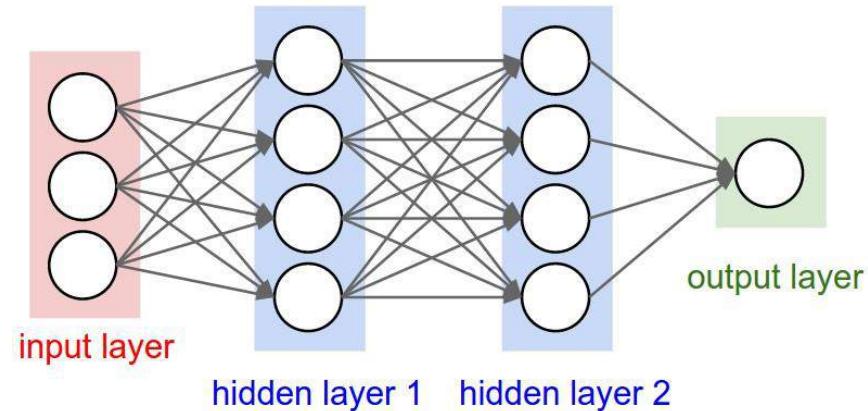
# Overview of CNN

# CNN layers

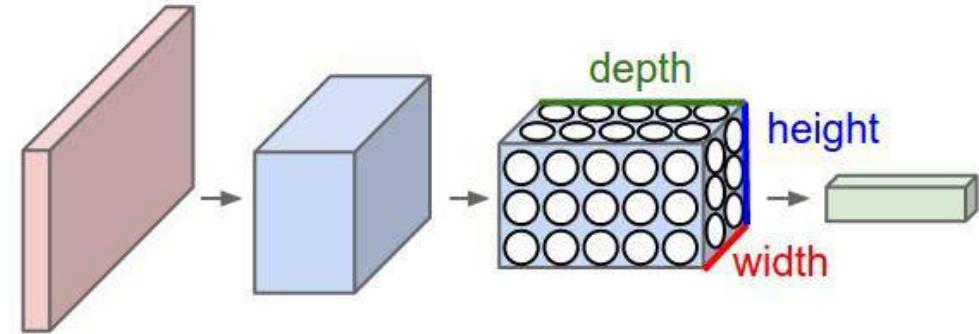
- Stages of CNN:
  - Convolution (in parallel) to produce pre-synaptic activations
  - Detector: Non-linear function
  - Pooling: A summary of a neighborhood
- Pooling of a rectangular region:
  - Max
  - Average
  - L2 norm
  - Weighted average acc. to the distance to the center
  - ...



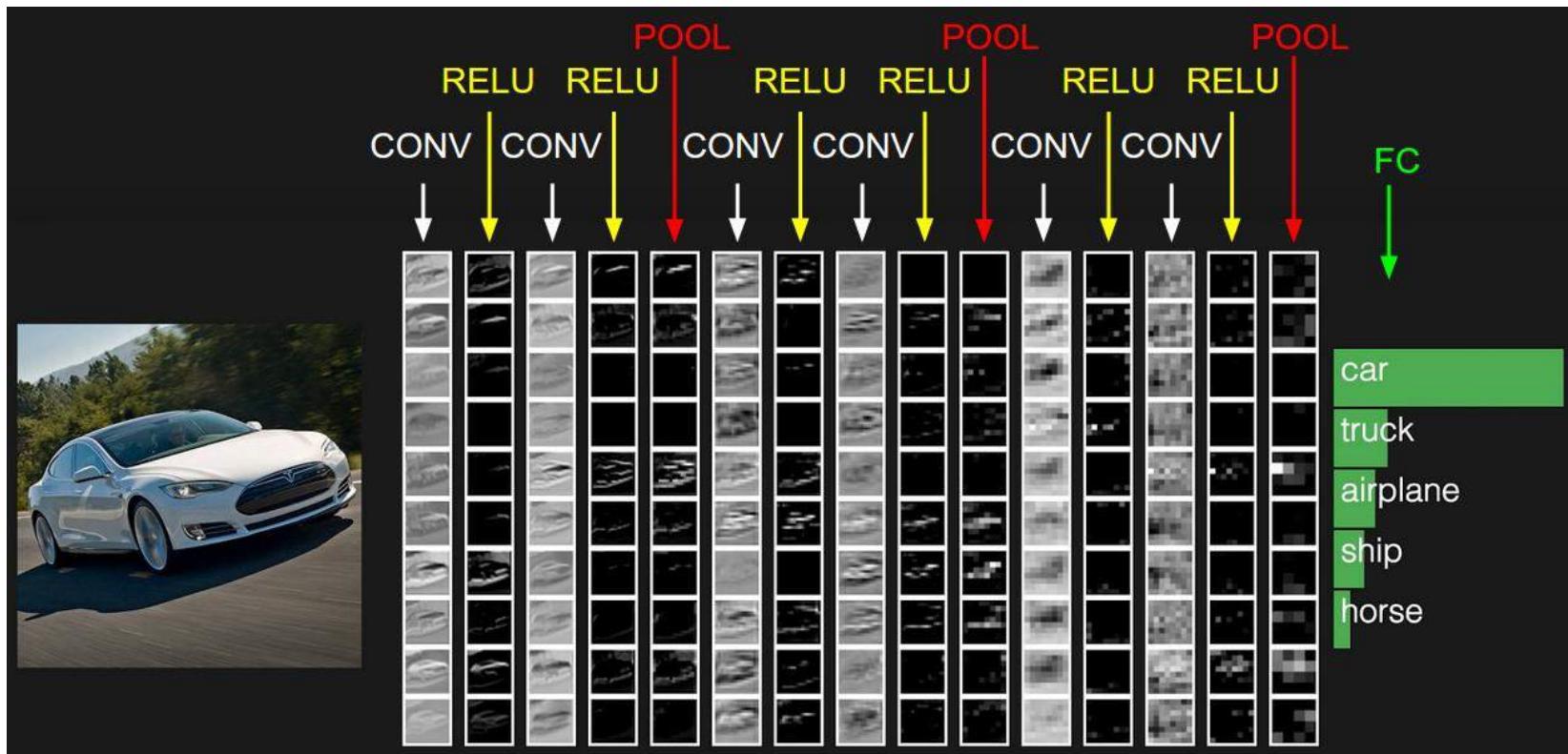
# Regular ANN



# CNN



# An example architecture



# Convolution in CNN

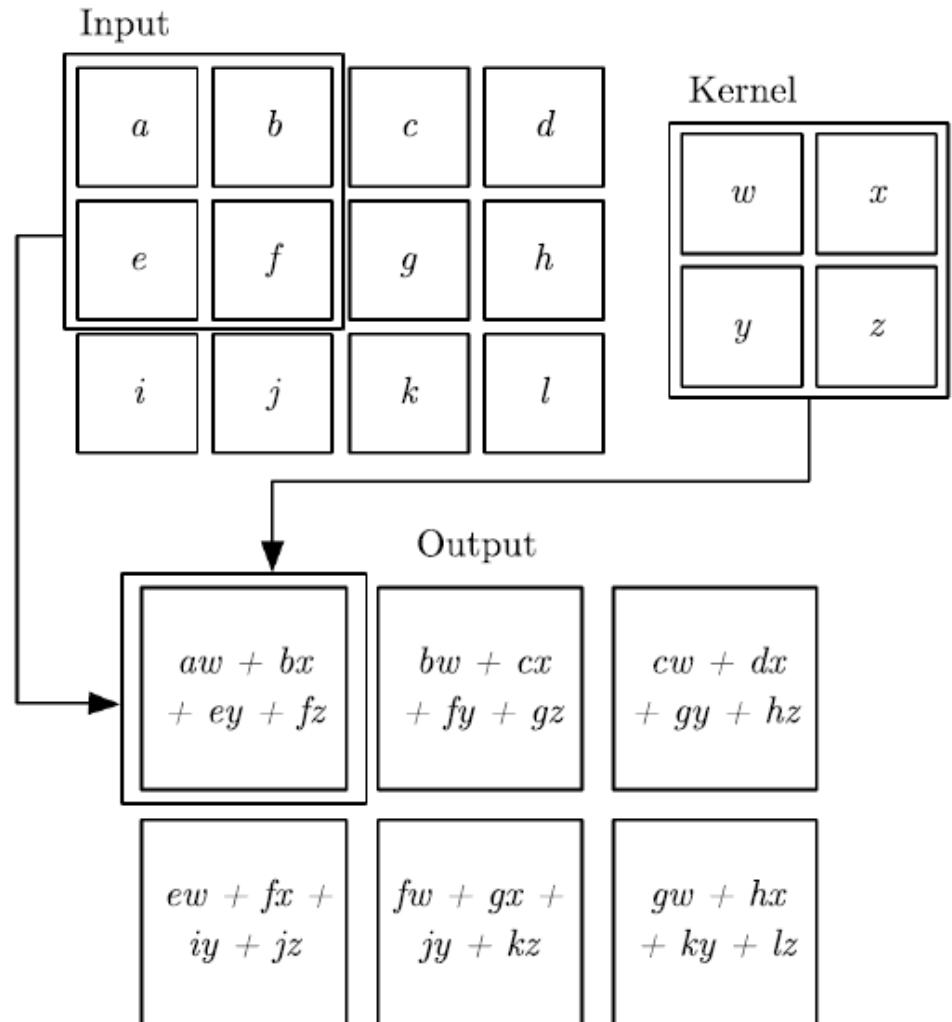
# Convolution in CNN

$$y(t) = x(t) * h(t) = \int x(\tau)h(t - \tau) d\tau$$

- $x(t)$ : input
- $h(t)$ : impulse response (in engineering), kernel (in CNN)
- $y(t)$ : output, feature map (in CNN)
- And  $h(t) = 0$  for  $t < 0$ . Otherwise, it means it uses future input.
- Effectively, we are learning the filters that best work for our problem.

# Convolution in CNN

- The weights correspond to the kernel
- The **weights are shared** in a channel (depth slice)
- We are effectively **learning filters** that respond to some part/entities/visual-cues etc.



# Local connectivity in CNN = Receptive fields

- Each neuron is connected to only a local neighborhood, i.e., receptive field
- The size of the receptive field → another hyper-parameter.

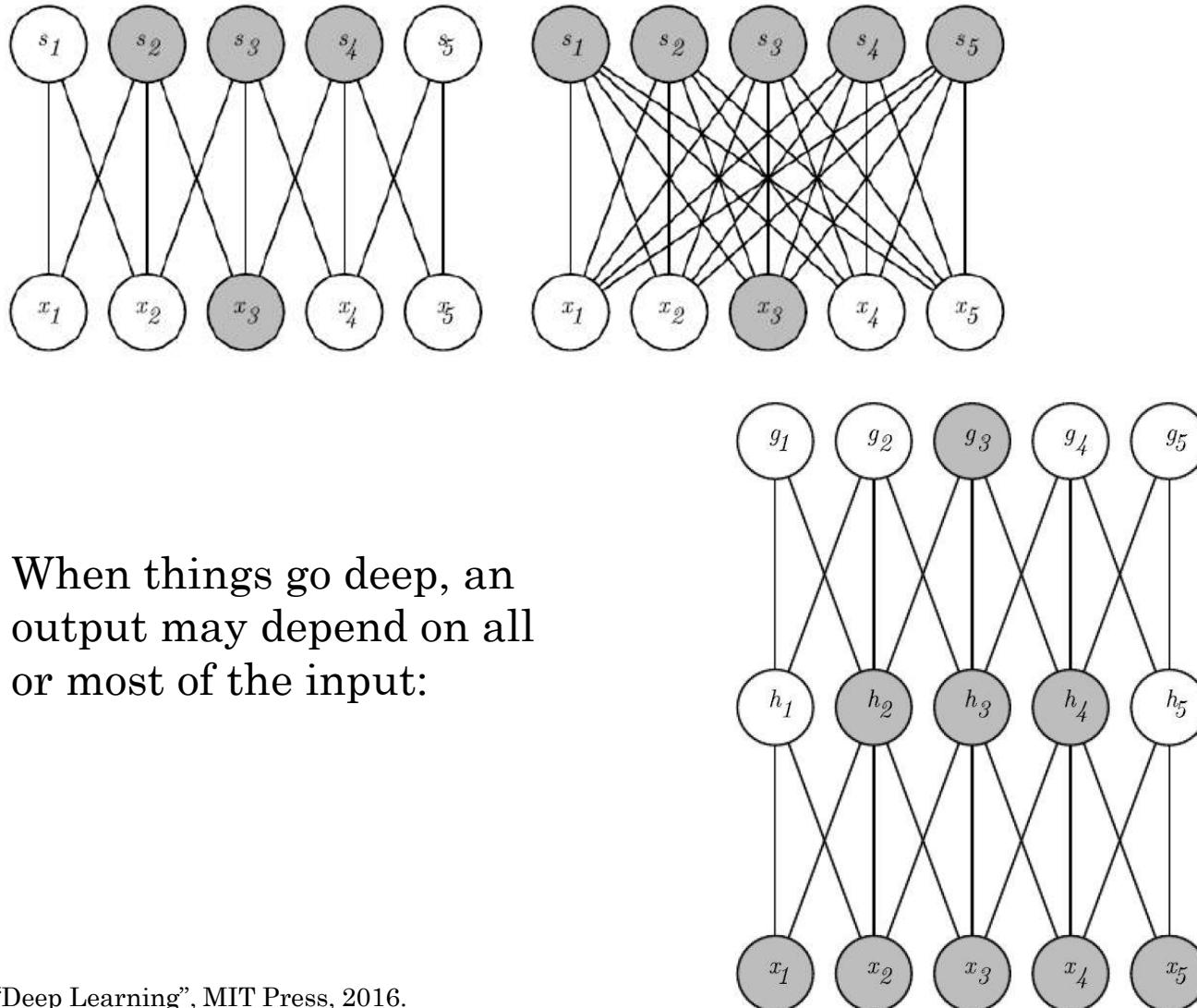
Having looked at convolution in general  
and convolution in CNN:

## Why convolution in neural networks?

- In regular ANNs, each output node considers all data.
  - Matrix multiplication in essence.
  - This is redundant in most cases.
  - Expensive.
- Instead, a smaller matrix multiplication operation.
  - Sparse interaction.
  - Can extract meaningful entities, such edges, corners, that occupy less space.
  - This reduces space requirements, and improves speed.
- Regular ANN:  $m \times n$  parameters
- With CNN (if we restrict the output to link to  $k$  units):  $n \times k$  parameters.

Having looked at convolution in general  
and convolution in CNN:

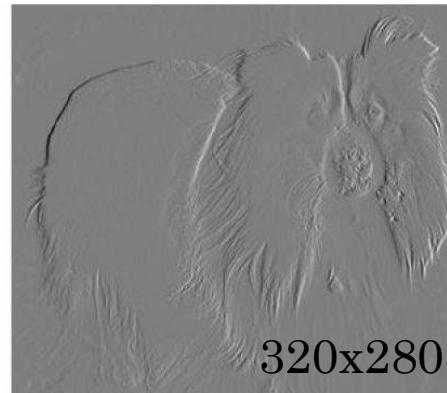
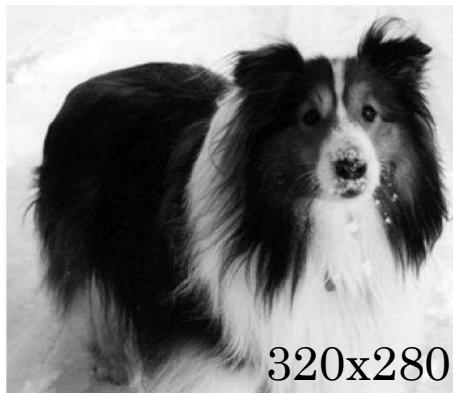
Why convolution in neural networks?



# Having looked at convolution in general and convolution in CNN:

## Why convolution in neural networks?

- Parameter sharing
  - In regular ANN, each weight is independent
- In CNN, a layer might re-apply the same convolution and therefore, share the parameters of a convolution
  - Reduces storage and learning time



- With ANN:  $320 \times 280 \times 320 \times 280$  multiplications
- With CNN:  $320 \times 280 \times 3$  multiplications

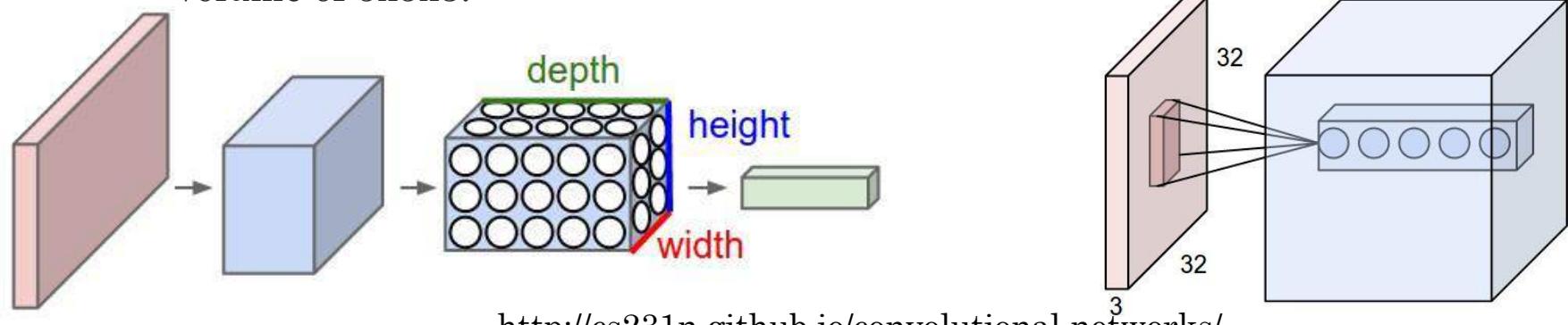
Having looked at convolution in general  
and convolution in CNN:

## Why convolution in neural networks?

- Equivariant to translation
  - The output will be the same, just translated, since the weights are shared.
- Not equivariant to scale or rotation.

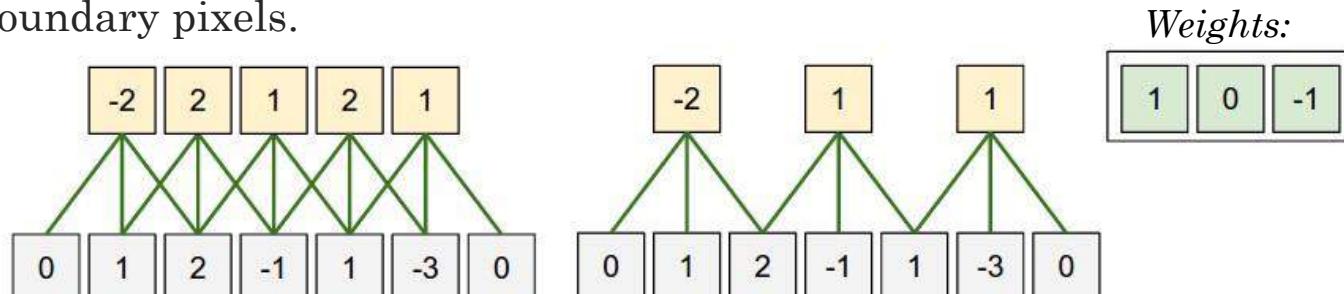
# Connectivity in CNN

- Local
  - The behavior of a neuron does not change other than being restricted to a subspace of the input.
- Each neuron is connected to slice of the previous layer
- A layer is actually a volume having a certain **width x height** and **depth** (or channel)
- A neuron is connected to a subspace of **width x height** but to **all channels** (depth)
- Example: CIFAR-10
  - Input:  $32 \times 32 \times 3$  (3 for RGB channels)
  - A neuron in the next layer with receptive field size  $5 \times 5$  has input from a volume of  $5 \times 5 \times 3$ .



# Important parameters

- Depth (number of channels)
  - We will have more neurons getting input from the same receptive field
  - This is similar to the hidden neurons with connections to the same input
  - These neurons learn to become selective to the presence of different signal in the same receptive field
- Stride
  - The amount of space between neighboring receptive fields
  - If it is small, RFs overlap more
  - If it is big, RFs overlap less
- How to handle the boundaries?
  - i. Option 1: Don't process the boundaries. Only process pixels on which convolution window can be placed fully.
  - ii. Option 2: Zero-pad the input so that convolution can be performed at the boundary pixels.



# Padding illustration

- Only convolution layers are shown.
- Top: no padding → layers shrink in size.
- Bottom: zero padding → layers keep their size fixed.

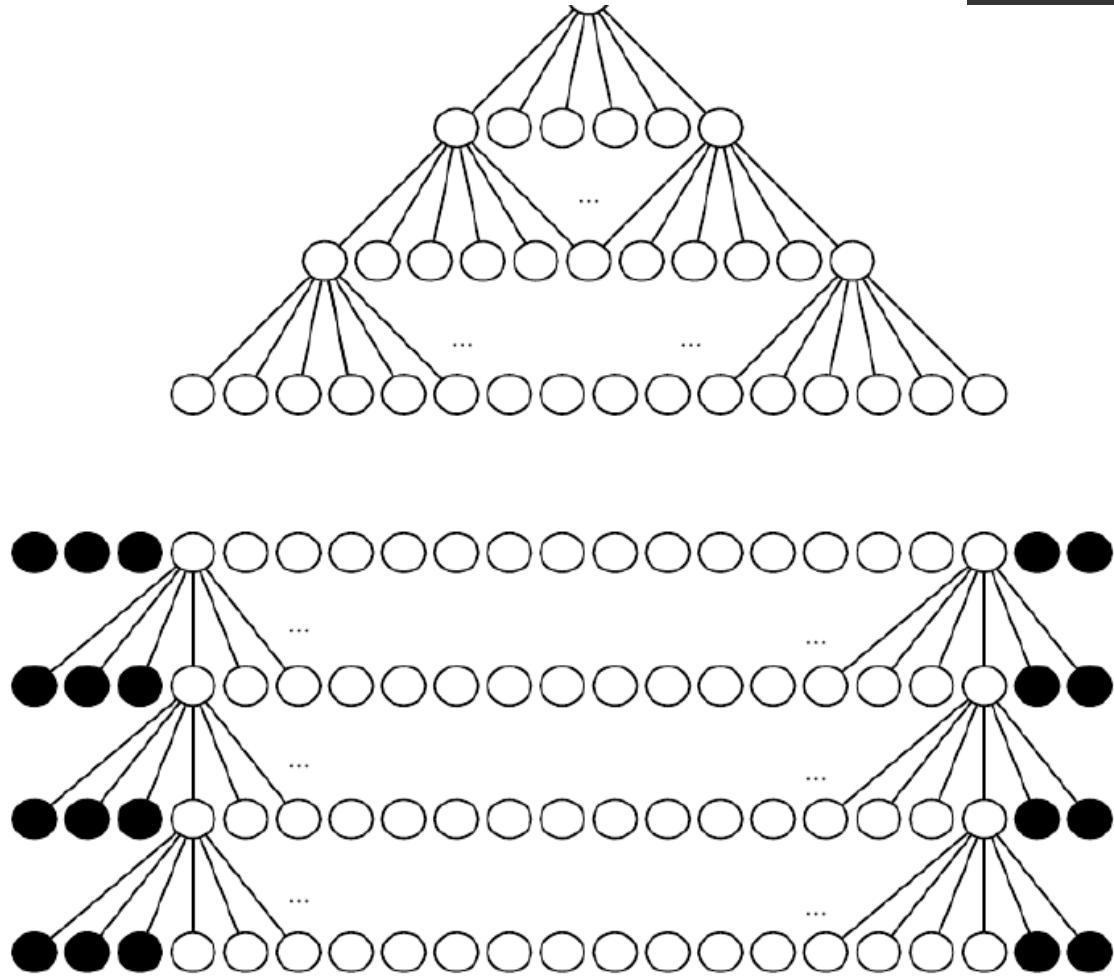


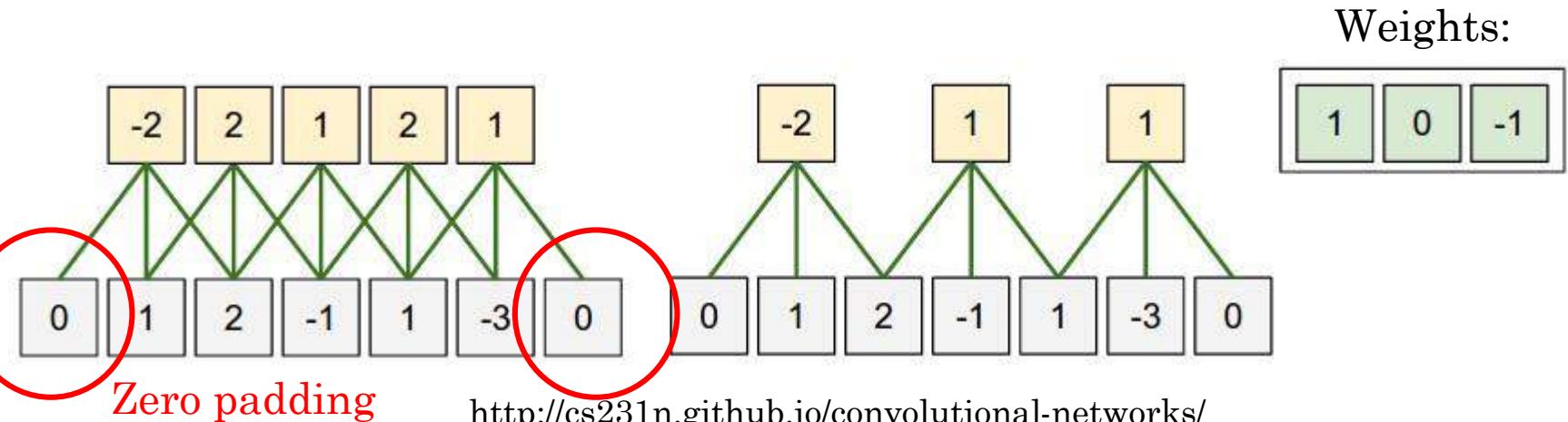
Figure 9.11: *The effect of zero padding on network size*: Consider a convolutional network with a kernel of width six at every layer. In this example, do not use any pooling, so only the convolution operation itself shrinks the network size. *Top*) In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive and some shrinking is inevitable in this kind of architecture. *Bottom*) By adding five implicit zeroes to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

# Size of the network

- Along a dimension:
  - $W$ : Size of the input
  - $F$ : Size of the receptive field
  - $S$ : Stride
  - $P$ : Amount of zero-padding
- Then: the number of neurons in the output:

$$\frac{W - F + 2P}{S} + 1$$

- If this number is not an integer, your strides are incorrect and your neurons cannot tile nicely to cover the input volume



# Size of the network

- Arranging these hyperparameters can be problematic

- Example:

- If  $W=10$ ,  $P=0$ , and  $F=3$ , then

$$\frac{W - F + 2P}{S} + 1 = \frac{10 - 3 + 0}{S} + 1 = \frac{7}{S} + 1$$

i.e.,  $S$  cannot be an integer other than 1 or 7.

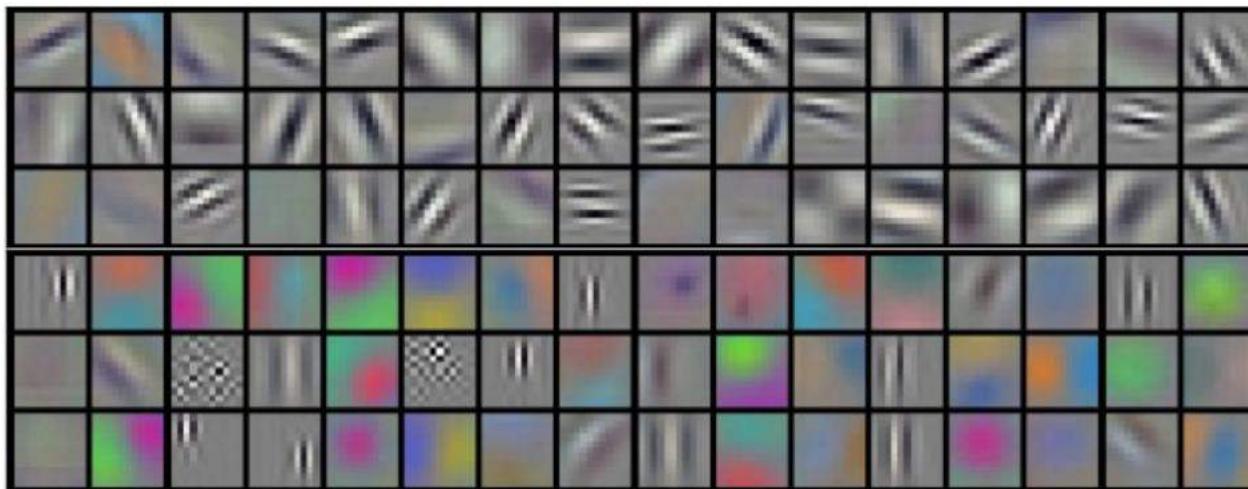
- Zero-padding is your friend here.

# Real example – AlexNet (Krizhevsky et al., 2012)

- Images: 227x227x3
- W=227, F=11, S=4, P=0  $\rightarrow \frac{227-11}{4} + 1 = 55$  (the size of the convolutional layer)
- Convolution layer: 55x55x96 neurons (96: the depth, the number of channels)
- The neurons in the first layer:  $55 \times 55 \times 96 = 290,400$  neurons
  - Each has  $11 \times 11 \times 3$  receptive field  $\rightarrow 363$  weights and 1 bias
  - Then,  $290,400 \times 364 = 105,705,600$  parameters just for the first convolution layer (if no sharing)
  - Very high number

# Real example – AlexNet (Krizhevsky et al., 2012)

- However, we can share the parameters
  - For each channel (slice of depth), have the same set of weights
  - If 96 channels, this means 96 different set of weights
  - Then,  $96 \times 364 = 34,944$  parameters
  - 364 weights shared by  $55 \times 55$  neurons in each channel



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55\*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55\*55 distinct locations in the Conv layer output volume.