Leçon 7 et 8

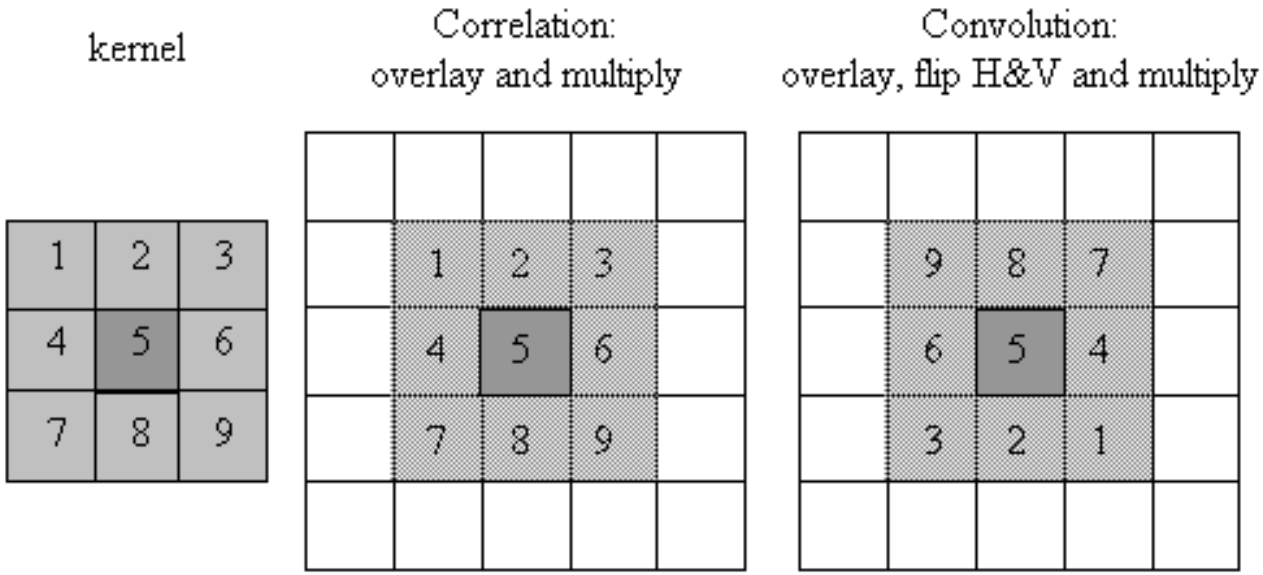
Plan de cours

- Convolutional Neural Networks
- Generative vs Discriminative methods
- Autoencoders
- Stochastic Networks – Boltzmann Machines
- Recurrent Neural Networks

Réseaux de neurones convolutionnels
Préliminaires

Le filtrage de l'image

Intuition: la corrélation croisée versus le produit de convolution



Merci à U. Rochester

Convolution vs corrélation

- Convolution - de l'intuition à l'équation

$$Conv[i][j] = P_{ij} \otimes K = \sum_{s=-\frac{k_w}{2}}^{\frac{k_w}{2}} \sum_{t=-\frac{k_h}{2}}^{\frac{k_h}{2}} P[i+s][j+t] * K[\frac{k_w}{2} - s][\frac{k_h}{2} - t]$$

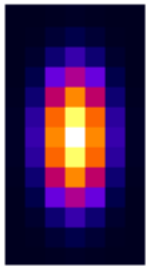
- P - une matrice de valeurs de pixels
- K - le noyau de filtrage

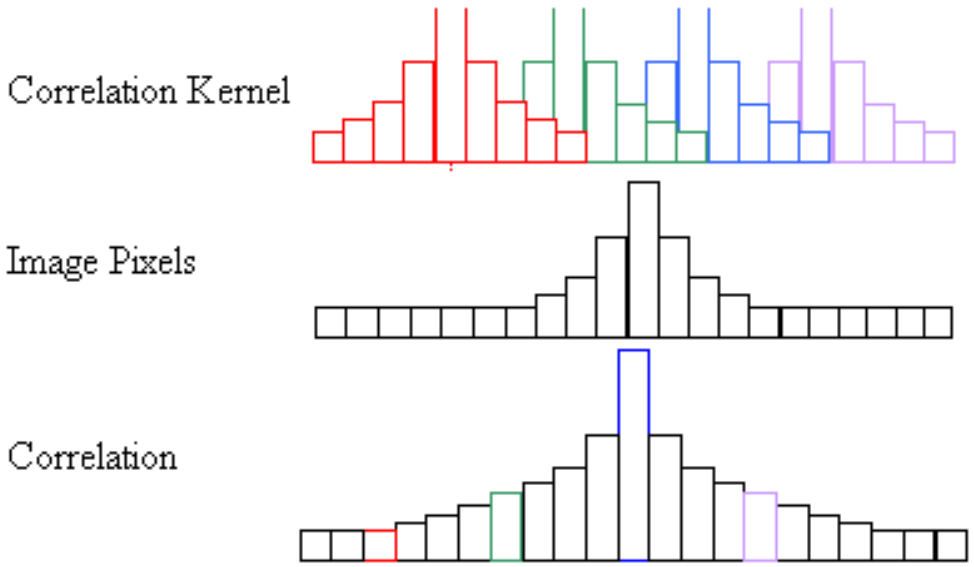
$$Corr[i][j] = \sum_{s=-\frac{k_w}{2}}^{\frac{k_w}{2}} \sum_{t=-\frac{k_h}{2}}^{\frac{k_h}{2}} P[i+s][j+t] * K[\frac{k_w}{2} + s][\frac{k_h}{2} + t]$$

- Corrélation - de l'intuition à l'équation

Quel est le résultat?

- Lorsque l'image correspond au noyau, il existe une corrélation positive élevée





- Mais, il est aussi possible de penser de l'opération comme une forme de filtrage linéaire

Linear functions

- Simplest: linear filtering.
  - Replace each pixel by a linear combination of its neighbors.
- The prescription for the linear combination is called the "convolution kernel".

10	5	3
4	5	1
1	1	7

Local image data

0	0	0
0	0.5	0
0	1	0.5

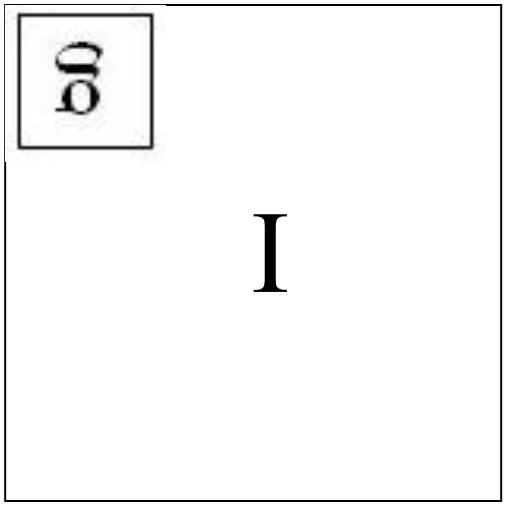
kernel

	7	

Modified image data

Convolution

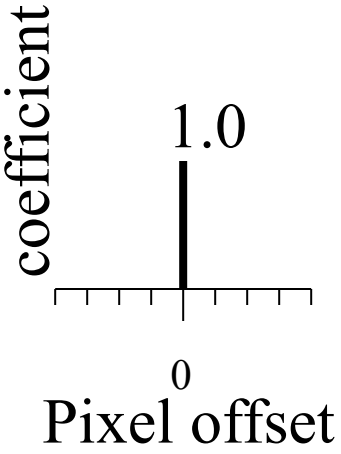
$$f[m,n] = I \otimes g = \sum_{k,l} I[m-k, n-l] g[k,l]$$



Merci à Friedman, MIT

Linear filtering (warm-up slide)

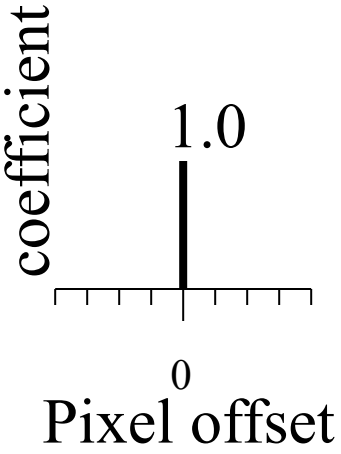
original



?

Linear filtering (warm-up slide)

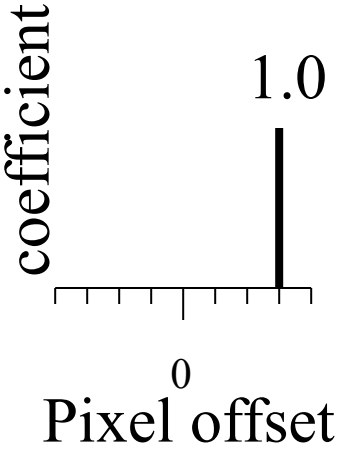
original



Filtered
(no change)

Linear filtering

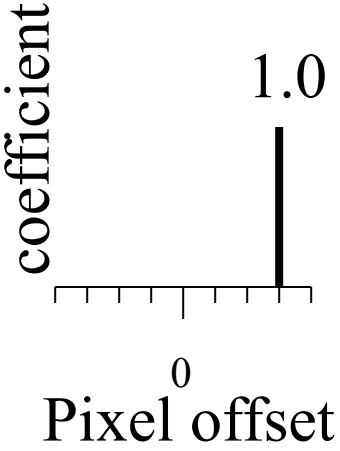
original

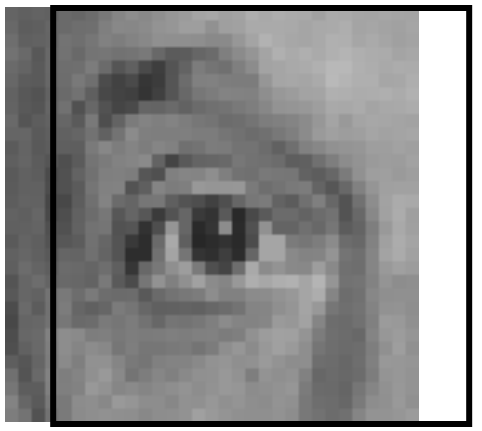


?

shift

original

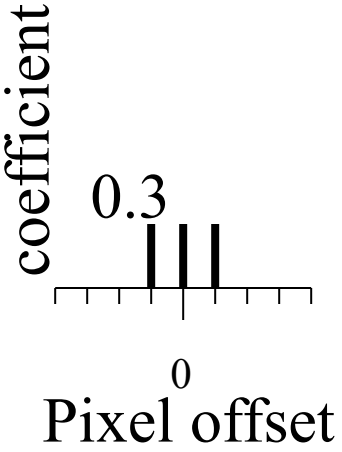




shifted

Linear filtering

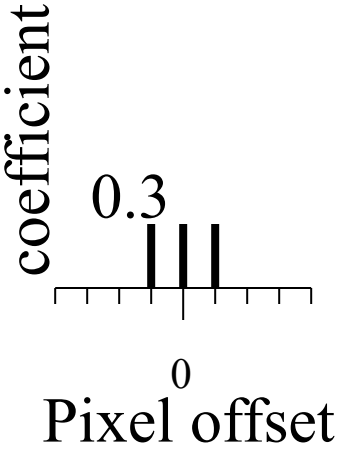
original



?

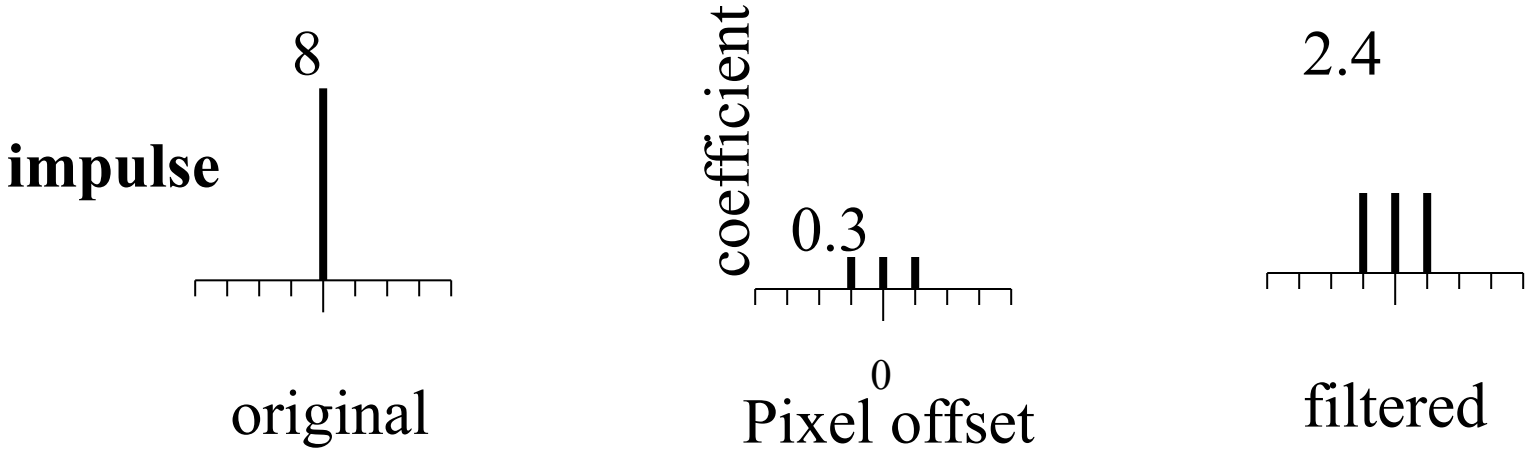
Blurring

original

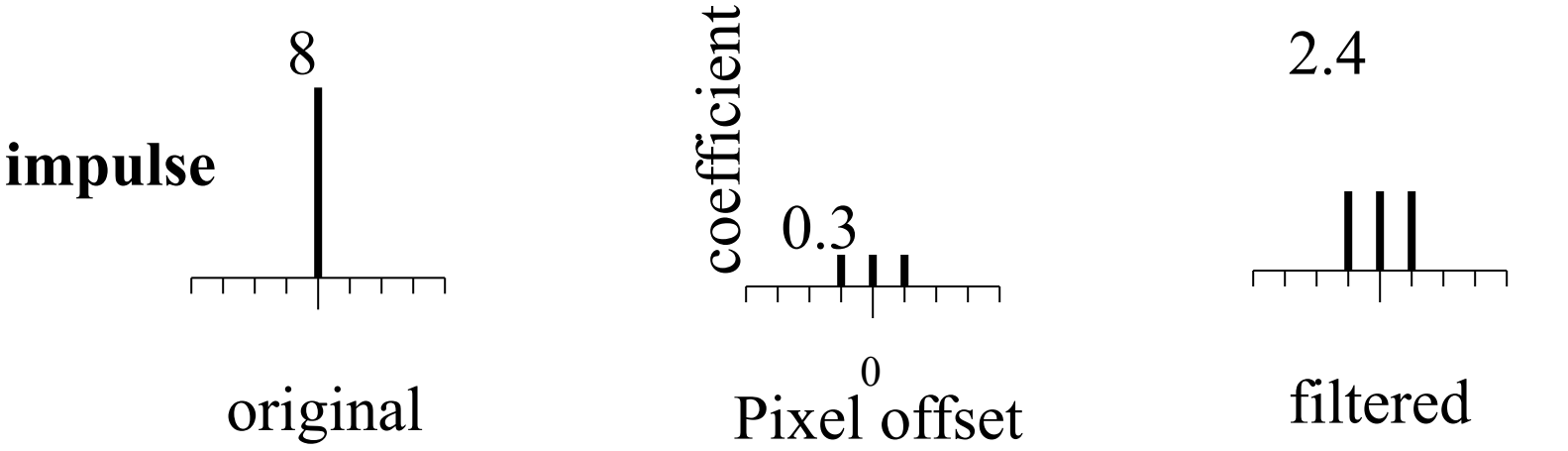


Blurred (filter applied in both dimensions).

Blur examples

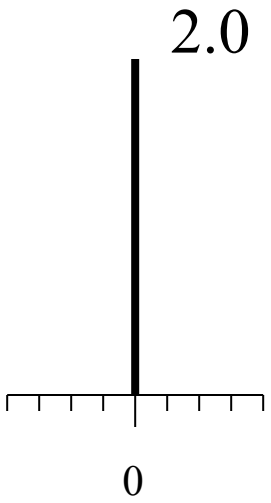


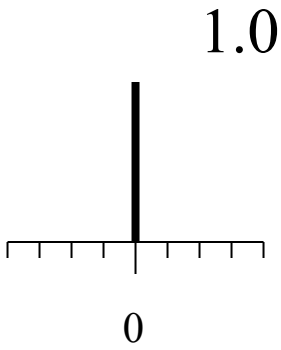
Blur examples



Linear filtering (warm-up slide)

original

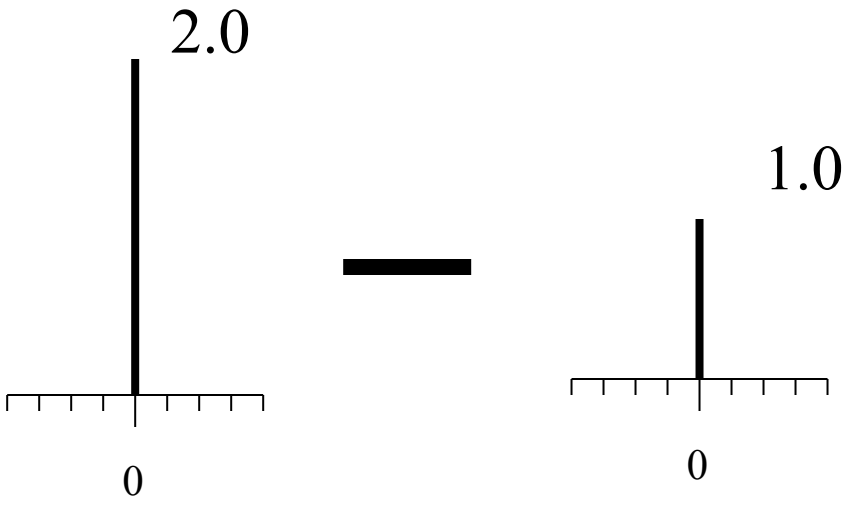




?

Linear filtering (no change)

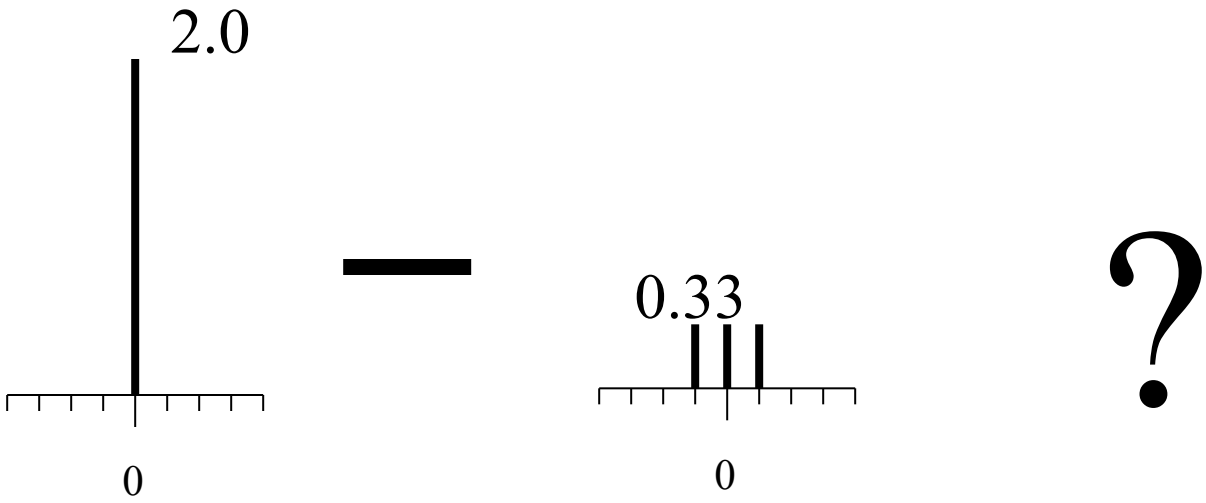
original



Filtered
(no change)

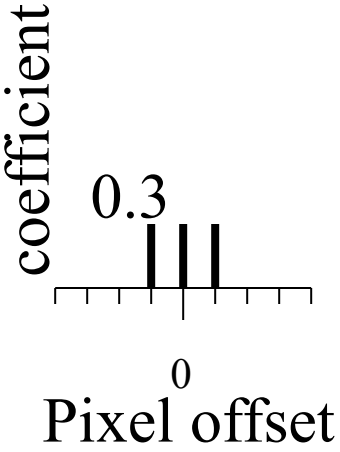
Linear filtering

original



(remember blurring)

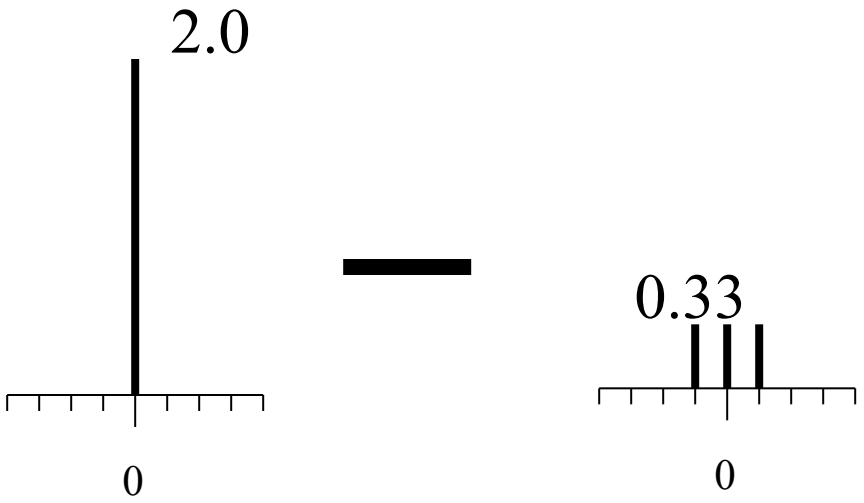
original

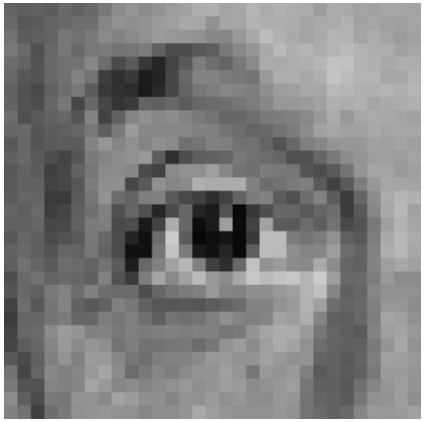


Blurred (filter applied in both dimensions).

Sharpening

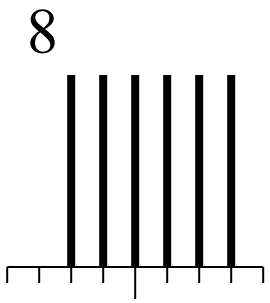
original



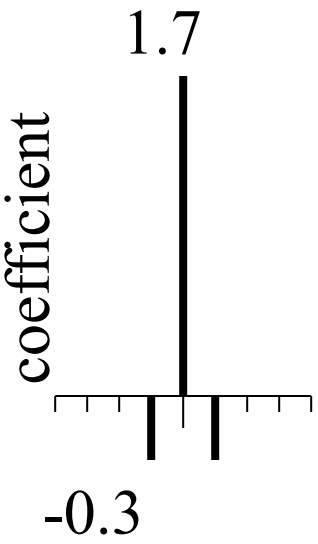


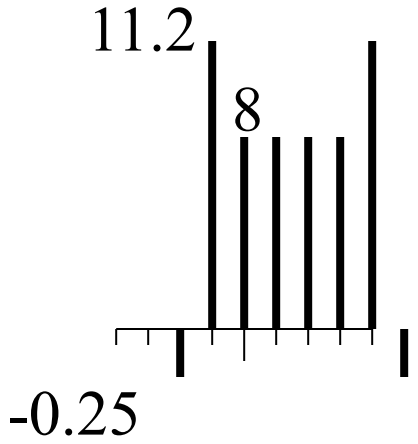
Sharpened
original

Sharpening example



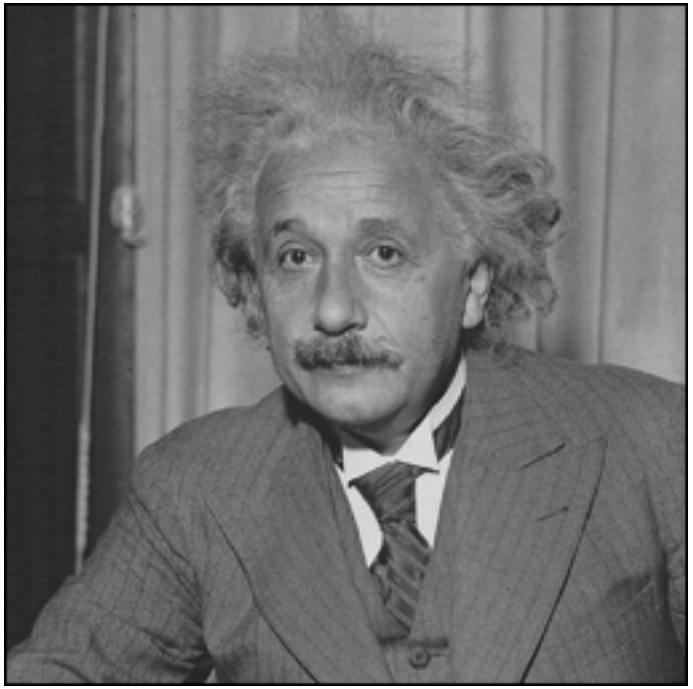
original



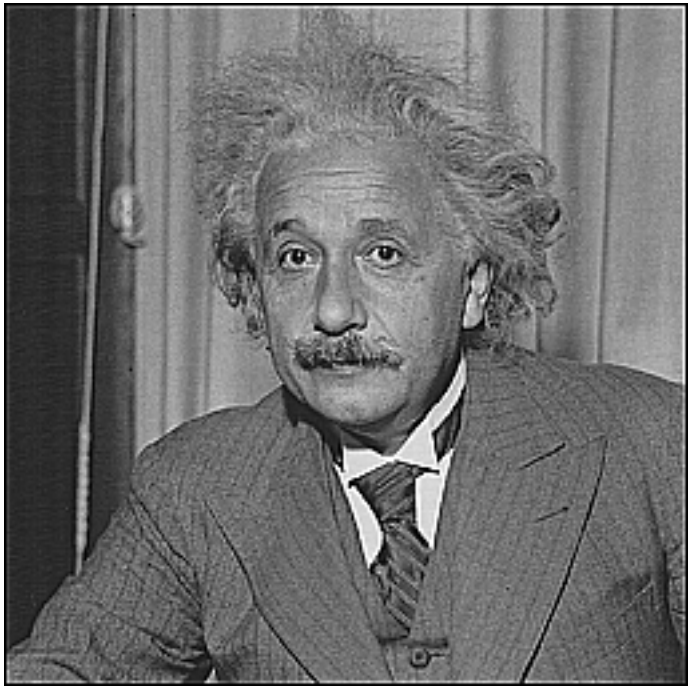


Sharpened (differences are accentuated; constant areas are left untouched).

Sharpening



before



after

Consider now the following filters

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

From left to right: a) A photograph, b) the photograph filtered with the Sobel operator $\mathbf{G}_x$, which emphasizes vertical edges, c) the photograph filtered with the Sobel operator $\mathbf{G}_y$, which emphasizes horizontal edges, d) the magnitue of the response of the pairs of filters at each spatial location, (but with the intensity flipped so that larger values are darker).

Convolutional neural networks

Convolutional neural networks (CNNs)

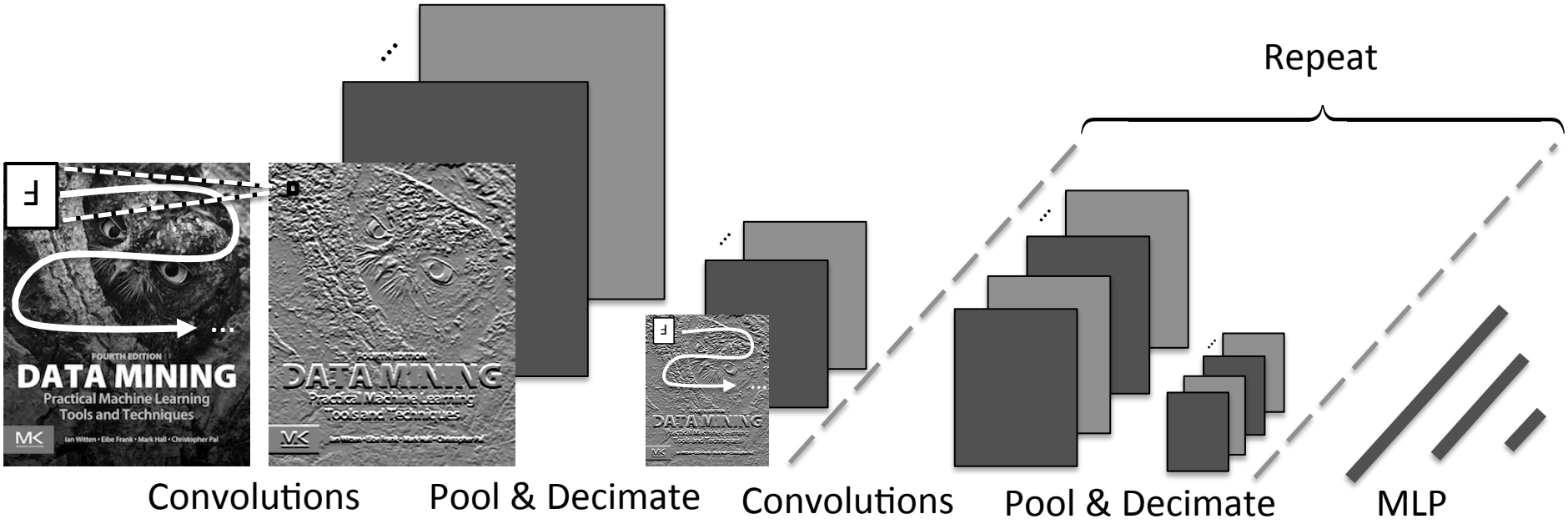
- Are a special kind of feedforward network that has proven *extremely* successful for image analysis
- Imagine filtering an image to detect edges, one could think of edges as a useful set of spatially organized 'features'
- Imagine now if one could learn many such filters jointly along with other parameters of a neural network on top
- Each filter can be implemented by multiplying a relatively small spatial zone of the image by a set of weights and feeding the result to an activation function – just like those discussed above for vanilla feedforward networks
- Because this filtering operation is simply repeated around the image using the same weights, it can be implemented using convolution operations
- The result is a CNN for which it is possible to learn both the filters and the classifier using SGD and the backpropagation algorithm

Deep CNNs

- In a convolutional neural network, once an image has been filtered by several learnable filters, each filter bank's output is often aggregated across a small spatial region, using the average or maximum value.
- Aggregation can be performed within non-overlapping regions, or using subsampling, yielding a lower-resolution layer of spatially organized features—a process that is sometimes referred to as "decimation"
- This gives the model a degree of invariance to small differences as to exactly where a feature has been detected.
- If aggregation uses the max operation, a feature is activated if it is detected anywhere in the pooling zone
- The result can be filtered and aggregated again

A typical CNN architecture

- Many feature maps are obtained from convolving learnable filters across an image
- Results are aggregated or pooled & decimated
- Process repeats until last set of feature maps are given to an MLP for final prediction



CNNs in practice

- LeNet and AlexNet architectures are canonical models
- While CNNs are designed to have a certain degree of translational invariance, augmenting data through global synthetic transformations like the cropping trick can increase performance significantly
- CNNs are usually optimized using mini-batch-based stochastic gradient descent, so practical discussions above about learning deep networks apply
- The use of GPU computing is typically *essential* to accelerate convolution operations significantly
- Resource issues related to the amount of CPU vs. GPU memory available are often important to consider

The ImageNet challenge

- Crucial in demonstrating the effectiveness of deep CNNs
- Problem: recognize object categories in Internet imagery
- The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) classification task - classify image from Flickr and other search engines into 1 of 1000 possible object categories
- Serves as a standard benchmark for deep learning
- The imagery was hand-labeled based on the presence or absence of an object belonging to these categories
- There are 1.2 million images in the training set with 732-1300 training images available per class
- A random subset of 50,000 images was used as the validation set, and 100,000 images were used for the test set where there are 50 and 100 images per class respectively

A plateau, then rapid advances

- "Top-5 error" is the % of times that the target label does not appear among the 5 highest-probability predictions
- Visual recognition methods not based on deep CNNs hit a plateau in performance at 25%

Name	Layers	Top-5 Error (%)	References
AlexNet	8	15.3	Krizhevsky et al. (2012)
VGG Net	19	7.3	Simonyan and Zisserman (2014)
ResNet	152	3.6	He et al. (2016)

- Note: the performance for human agreement has been measured at 5.1% top-5 error
- Smaller filters have been found to lead to superior results in deep networks: the methods with 19 and 152 layers use filters of size 3×3

Starting simply: image filtering

- When an image is filtered, the output can be thought of as another image that contains the filter's response at each spatial location
- Consider filtering a 1D vector **x** by multiplication with a matrix **W** that has a special structure, such as

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \begin{bmatrix} w_1 & w_2 & w_3 & & & \\ & w_1 & w_2 & w_3 & & \\ & & & \ddots & & \\ & & & w_1 & w_2 & w_3 \end{bmatrix}$$

where the elements left blank in the matrix above are zero and we have used a simple filter having only three non-zero coefficients and a "stride" of one

Correlation and convolution

- Suppose our filter is centered, giving the first vector element an index of –1, or an index of –*K*, where *K* is the "radius" of the filter, then 1D filtering can be written

$$\mathbf{y}[n] = \sum_{k=-K}^{K} \mathbf{w}[k]\mathbf{x}[n+k]$$

- Directly generalizing this filtering to a 2D image **X** and filter **W** gives the *cross-correlation*, **Y**=**W**★**X**, for which the result for row *r* and column *c* is

$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[j,k]\mathbf{X}[r+j,c+k]$$

- The *convolution* of an image with a filter, **Y**=**W***__X__ , is obtained by simply flipping the sense of the filter

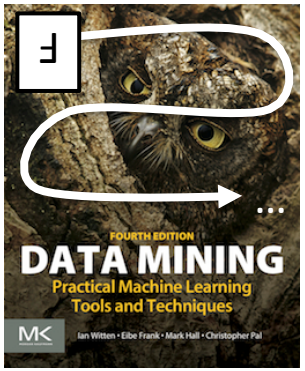
$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[-j,-k]\mathbf{X}[r+j,c+k]$$

Simple filtering example

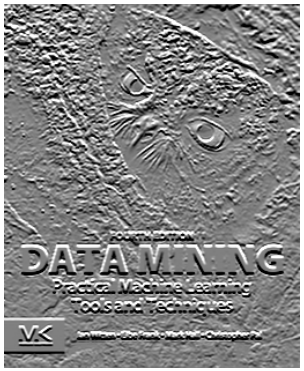
- Ex. consider the task of detecting edges in an image
- A well known technique is to filter an image with so-called "Sobel" filters, which involves convolving it with

$$\mathbf{W}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \mathbf{W}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

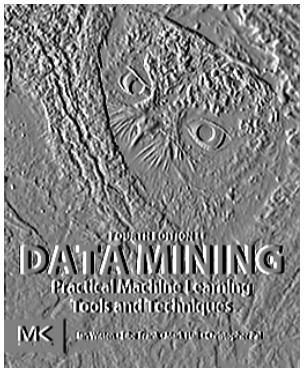
- Applied to the image X below, we have:



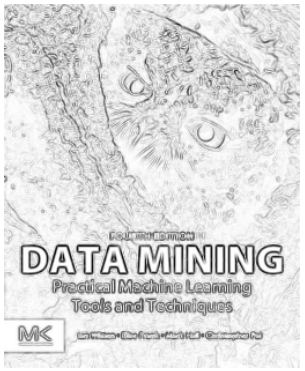
$\mathbf{X}$



$\mathbf{G}_x = \mathbf{W}_x * \mathbf{X}$



$\mathbf{G}_y = \mathbf{W}_y * \mathbf{X}$



$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$

Visualizing the filters learned by a CNN

- Learned edge-like filters and texture-like filters are frequently observed in the early layers of CNNs trained using natural images
- Since each layer in a CNN involves filtering the feature map below, so as one moves up the receptive fields become larger
- Higher- level layers learn to detect larger features, which often correspond to textures, then small pieces of objects

First Layer

Second Layer

Third Layer

(Imagery kindly provided by Matthew Zeiler)

- Above are the strongest activations of random neurons projecting the activation back into image space using the deconvolution approach of Zeiler and Fergus (2013).

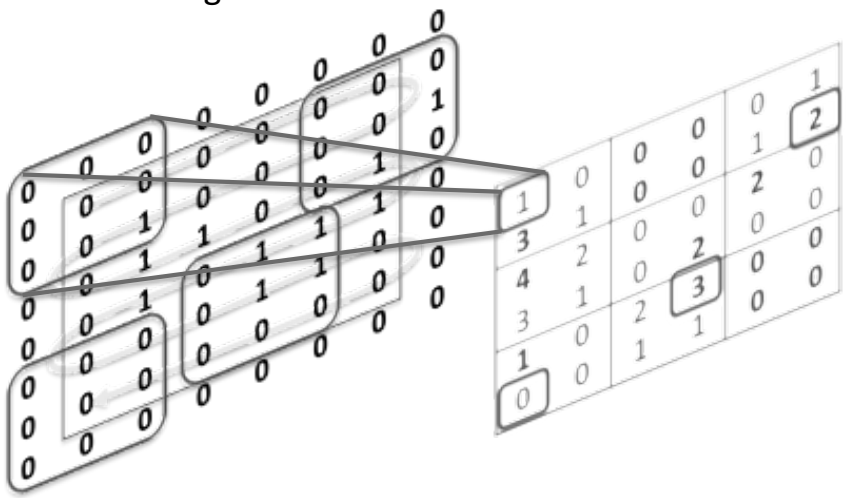
Simple example of: convolution, pooling, and decimation operations

Filter

-1	0	1
-2	0	2
-1	0	1

Image

Convolution



Max Pooling

3	3	0	0	2	2
3	3	0	0	2	2
4	4	2	2	2	2
4	4	2	2	2	2
1	1	3	3	0	0
1	1	3	3	0	0

Decimation

3	0	2
4	2	2
1	3	0

- An image is convolved with a filter; curved rectangular regions in the first large matrix depict a random set of image locations
- Maximum values within small 2×2 regions are indicated in bold in the central matrix
- The results are pooled, using max-pooling then decimated by a factor of two, to yield the final matrix

Convolutional layers and gradients

- Let's consider how to compute the gradients needed to optimize a convolutional network
- At a given layer we have i=1…$N^{(l)}$ feature filters and corresponding feature maps
- The convolutional kernel matrices $\mathbf{K}_i$ contain flipped weights with respect to kernel weight matrices $\mathbf{W}_i$
- With activation function act(), and for each feature type i, a scaling factor g_i and bias matrix $\mathbf{B}_i$, the feature maps are matrices $\mathbf{H}_i(\mathbf{A}_i(\mathbf{X}))$ and can be visualized as a set of images given by

$$\mathbf{H}_i = g_i \,\mathrm{act}\left[\mathbf{K}_i * \mathbf{X} + \mathbf{B}_i\right] = g_i \,\mathrm{act}\left[\mathbf{A}_i(\mathbf{X})\right]$$

Convolutional layers and gradients

- The loss is a function of the $N^{(l)}$ feature maps for a given layer, $L = L(\mathbf{H}_1^{(l)},\ldots,\mathbf{H}_{N^{(l)}}^{(l)})$
- Define **h**=vec(**H**), **x**=vec(**X**), **a**=vec(**A**), where the vec() function returns a vector with stacked columns of the given matrix argument,
- Choose an act() function that operates elementwise on an input matrix of pre-activations and has scale parameters of 1 and biases of 0.
- Partial derivatives of hidden layer output with respect to input **X** of the convolutional units are

$$\frac{\partial L}{\partial \mathbf{X}} = \sum_i \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{X}} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} = \sum_i \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}} \frac{\partial \mathbf{h}_i}{\partial \mathbf{a}_i} \frac{\partial L}{\partial \mathbf{h}_i} = \sum_i [\mathbf{W}_i * \mathbf{D}_i], \mathbf{D}_i = dL/\partial \mathbf{A}_i$$

Convolutional layers and gradients

- Matrix $\mathbf{D}_i$ in previous side is a matrix containing the partial derivative of the elementwise act() function's input with respect to its pre-activation value for the i^{th} feature type, organized according to spatial positions given by row *j* and column *k*.
- Intuitively, the result is a sum of the convolution of each of the (zero padded) filters $\mathbf{W}_i$ with an image-like matrix of derivatives $\mathbf{D}_i$.
- The partial derivatives of the hidden layer output are

$$\frac{\partial L}{\partial \mathbf{W}_i} = \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{W}_i} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} = \left[\mathbf{X}^{\dagger} * \mathbf{D}_i \right],$$

- Where $\mathbf{X}^{\dagger}$ is the row & column-flipped version of **X**

Pooling and subsampling layers

- What are the consequences of backpropagating gradients through max or average pooling layers?
- In the former case, the units that are responsible for the maximum within each zone *j*, *k* —the "winning units"— get the backpropagated gradient
- For average pooling, the averaging is simply a special type of convolution with a fixed kernel that computes the (possibly weighted) average of pixels in a zone
  - the required gradients are therefore like std conv. layers
- The subsampling step either samples every n^{th} output, or avoids needless computation by only evaluating every n^{th} pooling computation

Implementing CNNs

- Convolutions are very well suited for acceleration using GPUs
- Since graphics hardware can accelerate convolutions by an *order of magnitude* or more over CPU implementations, they play an important often *critical* role in training CNNs
- An experimental turn-around time of days rather than weeks makes a huge difference to model development times!
- Can also be challenging to construct software for learning a convolutional neural network in such a way that alternative architectures can be explored
- Early GPU implementations were hard to extend, newer tools allow for both fast computation and flexible high-level programming primitives
- Many software tools allow gradient computations and the backpropagation algorithm for large networks to be almost completely automated.

Bibliographic Notes & Further Reading

Convolutional Networks

- Modern convolutional neural networks are widely acknowledged as having their roots with the "neocognitron" proposed by Fukushima (1980); however
- The work of LeCun et al. (1998) on the LeNet convolutional network architecture has been extremely influential.
- The MNIST dataset containing 28×28 pixel images of handwritten digits has been popular in deep learning research community since 1998
- However, it was the ImageNet challenge (Russakovsky et al., 2015), with a variety of much higher resolutions, that catapulted deep learning into the spotlight in 2012.
  - The winning entry from the University of Toronto (Krizhevsky et al. , 2012) processed the images at a resolution of 256×256 pixels.
  - Up till then, CNNs were simply incapable of processing such large volumes of imagery at such high resolutions in a reasonable amount of time.

Bibliographic Notes & Further Reading

Convolutional Networks

- Krizhevsky et al. (2012)'s dramatic ImageNet win used a GPU accelerated convolutional neural networks.
  - This spurred a great deal of development, reflected in rapid subsequent advances in visual recognition performance and on the ImageNet benchmark.
- In the 2014 challenge, the Oxford Visual Geometry Group and a team from Google pushed performance even further using much deeper architectures: 16-19 weight layers for the Oxford group, using tiny 3×3 convolutional filters (Simonyan and Zisserman, 2014); 22 layers, with filters up to 5×5 for the Google team (Szegedy et al., 2015).
- The 2015 ImageNet challenge was won by a team from Microsoft Research Asia (MSRA) using an architecture with 152 layers (He et al., 2015), using tiny 3×3 filters combined with "shortcut" connections that skip over layers, and pooling and decimating the result of multiple layers of small convolution operations

Bibliographic Notes & Further Reading

Convolutional Networks

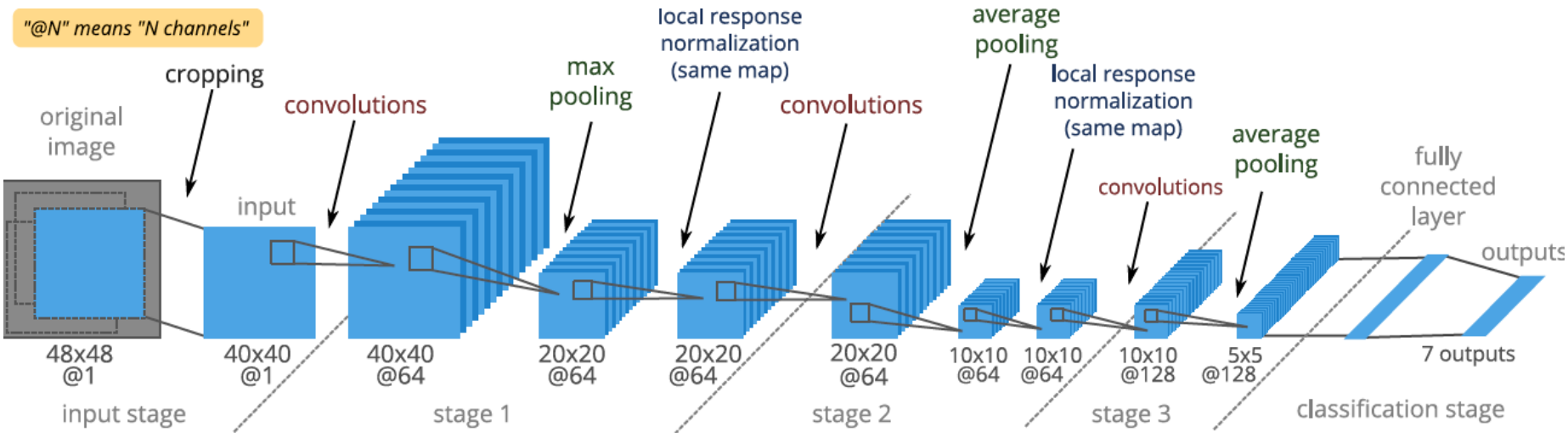
- Good parameter initialization can be critical for the success of neural networks, as discussed in LeCun et al. (1998)'s classic work and the more recent work of Glorot and Bengio (2010).
- Krizhevsky et al. (2012)'s convolutional network of rectified linear units (ReLUs) initialized weights using 0-mean isotropic Gaussian distributions with a standard deviation of 0.01, and initialized the biases to 1 for most hidden convolutional layers as well as their model's hidden fully connected layers.
- They observed that this initialization accelerated the early phase of learning by providing ReLUs with positive inputs.

Les réseaux convolutionelle

Voir les notes de cours supplémentaires sur le sujet disponible sur moodle

A Common Pattern for Convolutional Neural Networks

Convolutional Layers followed by a Multilayer Perceptron



- Four layers, 3 convolution-pool-decimate stages using max or average pooling followed by a fully-connected hidden layer, then the output layer
- Example: C++ and Cuda implementation of Krizhevsky et al
- Inputs are images of size 40x40, cropped randomly

Why are Deep Neural Networks so Hot?

- **Large-Vocabulary Speech Recognition**
  Significant increase in performance (Dahl, Yu, Deng & Acero, 2012)
  - Deep Neural Network : 16-23% relative error rate reduction over the previous state-of-the-art

- **Visual Recognition of 1000 Classes**
  Top results on the ImageNet contest (Krizhevsky, Sutskever & Hinton, 2012), in 1.2 million images
  - Deep Neural Network : 15% top-5 error rate
  - Second best entry: 26% top-5 error rate

- **Face Verification**
  Near human performance, 97.35% accuracy on the Labeled Faces in the Wild (LFW) evaluation, reducing the error of the state of the art at the time by more than 27% (Facebook Research, 2014)

Labeled Faces in the Wild

Menu

- LFW Home
  - Mailing
  - Explore
  - Download
  - Train/Test
  - Results
  - Information
  - Errata
  - Reference
  - Resources
  - Contact
  - Support
  - Changes
- Part Labels
- UMass Vision

Labeled Faces in the Wild Home

NEW RESULTS PAGE:

WE HAVE RECENTLY UPDATED AND CHANGED THE FORMAT AND CONTENT OF OUR RESULTS PAGE. PLEASE REFER TO THE NEW TECHNICAL REPORT FOR DETAILS OF THE CHANGES.

Welcome to Labeled Faces in the Wild, a database of face photographs designed for studying the problem of unconstrained face recognition. The data set contains more than 13,000 images of faces collected from the web. Each face has been labeled with the name of the person pictured. 1680 of the people pictured have two or more distinct photos in the data set. The only constraint on these faces is that they were detected by the Viola-Jones face detector. More details can be found in the technical report below.

Deep Learning and Face Verification

- **Face verification** – problem definition: given two facial images, do they belong to the same identity or not.
- The Labeled Faces in the Wild (LFW) de facto std. evaluation
  - 13,000 images of faces collected from the web (press photos)
  - 5749 identities, 1680 identities have two or more examples
- Taigman, Yang, Ranzato and Wolf, "***Deepface***: Closing the gap to human-level performance in face verification", CVPR 2014.
  - Near human level performance for face verification, **97.35%** accuracy on the Labeled Faces in the Wild (LFW) Benchmark
- Human Level Performance, ***DeepID2***: **99.15%** Sun, Wang, and Tang "Deep Learning Face Representation by Joint Identification-Verification"
- Human Performance: **97.53%** (cropped), **99.20%** (aligned)

Restricted vs. Unrestricted Results

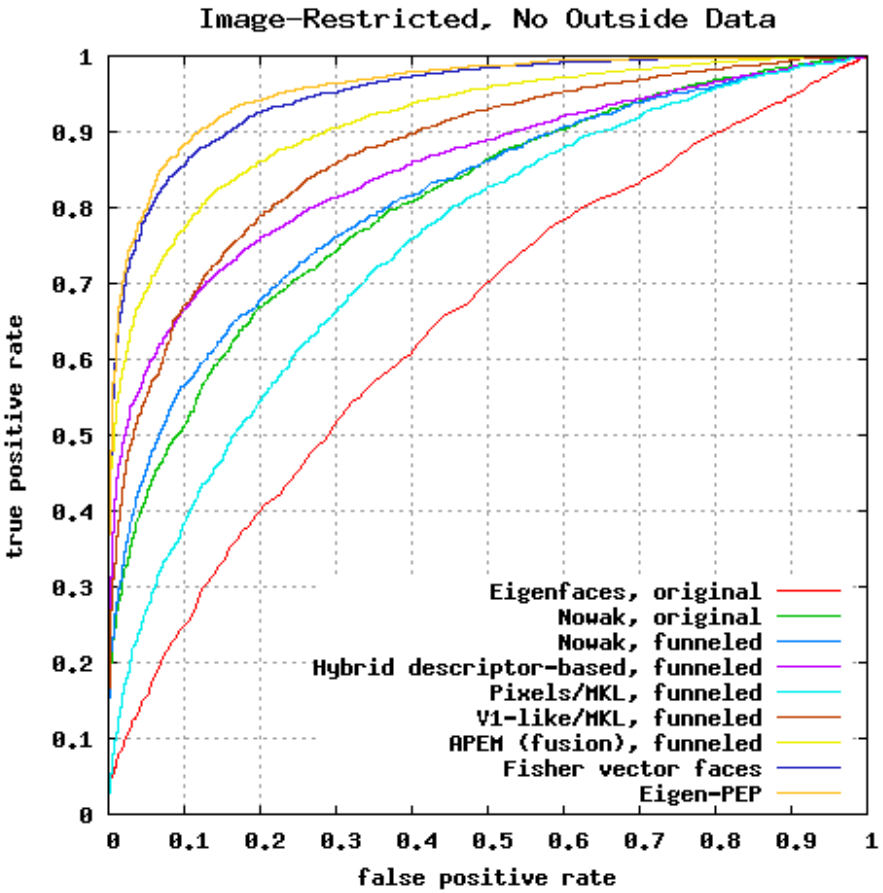


Figure 2: ROC curves averaged over 10 folds of View 2.

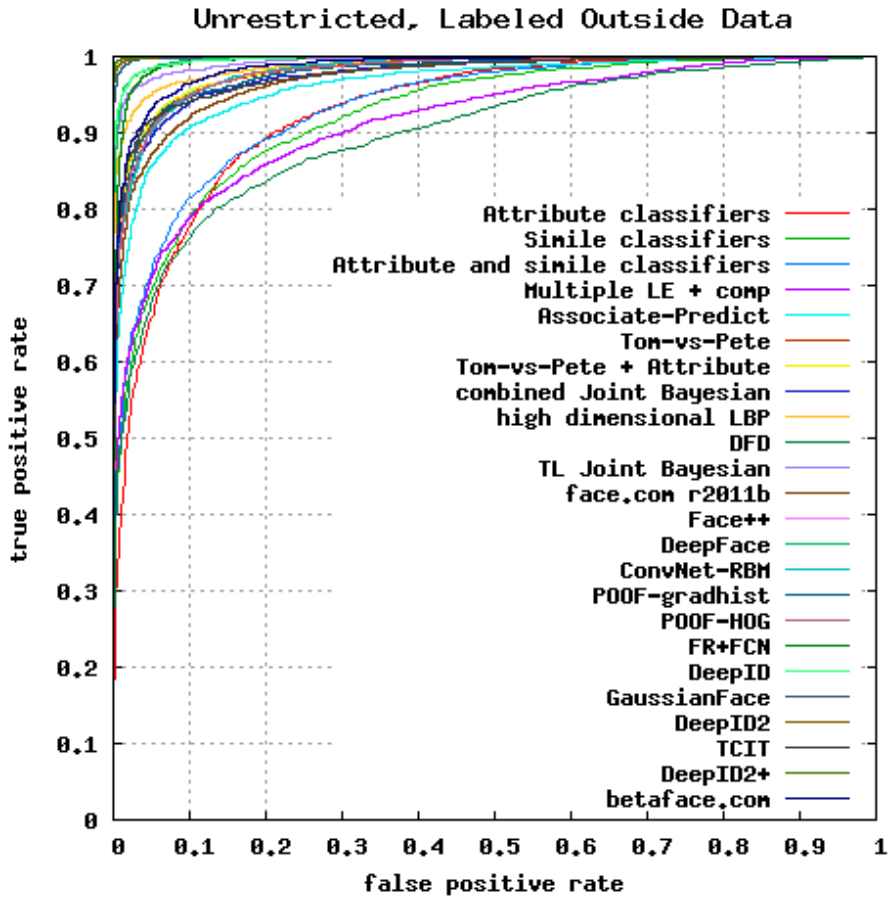
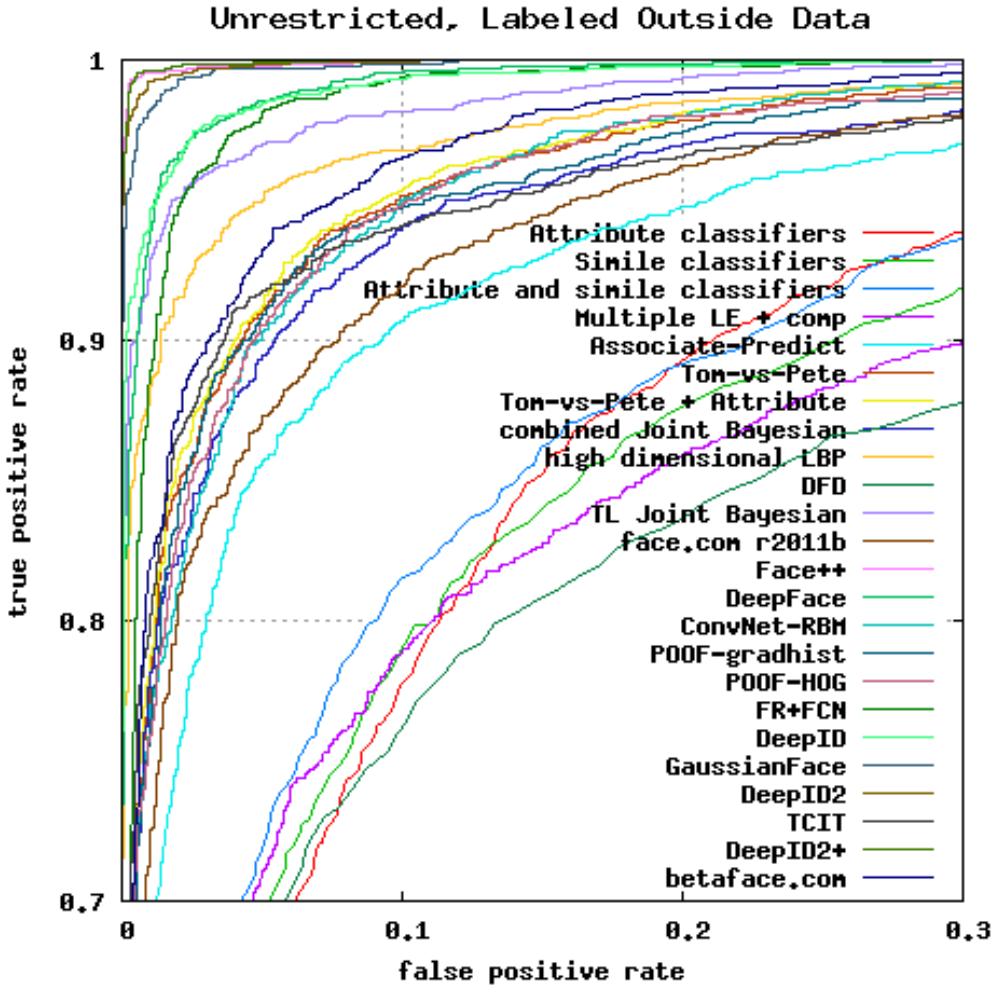


Figure 6: ROC curves averaged over 10 folds of View 2.
[click image to toggle zoom]

Unrestricted Results – Note How Deep Networks Dominate



Face++	0.9950 ± 0.0036
DeepID2+	0.9947 ± 0.0012
DeepID2	0.9915 ± 0.0013
GaussianFace	0.9852 ± 0.0066
DeepFace-ensemble	0.9735 ± 0.0025
ConvNet-RBM	0.9252 ± 0.0038
POOF-gradhist	0.9313 ± 0.0040
POOF-HOG	0.9280 ± 0.0047
FR+FCN	0.9645 ± 0.0025
DeepID	0.9745 ± 0.0026
TCIT	0.9333 ± 0.0124
betaface.com	0.9428 ± 0.0044
Human, funneled	**0.9920**
Human, cropped	0.9753
Human, inverse mask	0.9427

Figure 6: ROC curves averaged over 10 folds of View 2.

Where do gains come from ? (Highlights)

- ***DeepFace*** used the Social Face Dataset: **4.4 million labelled faces** from 4,030 people, each with 800-1200 faces
- ***DeepID2*** used CelebFaces+: **200k images**, 10k identities

Transforming from a recognition to verification task

Deepface trained ConvNet to recognize 3.8k identities

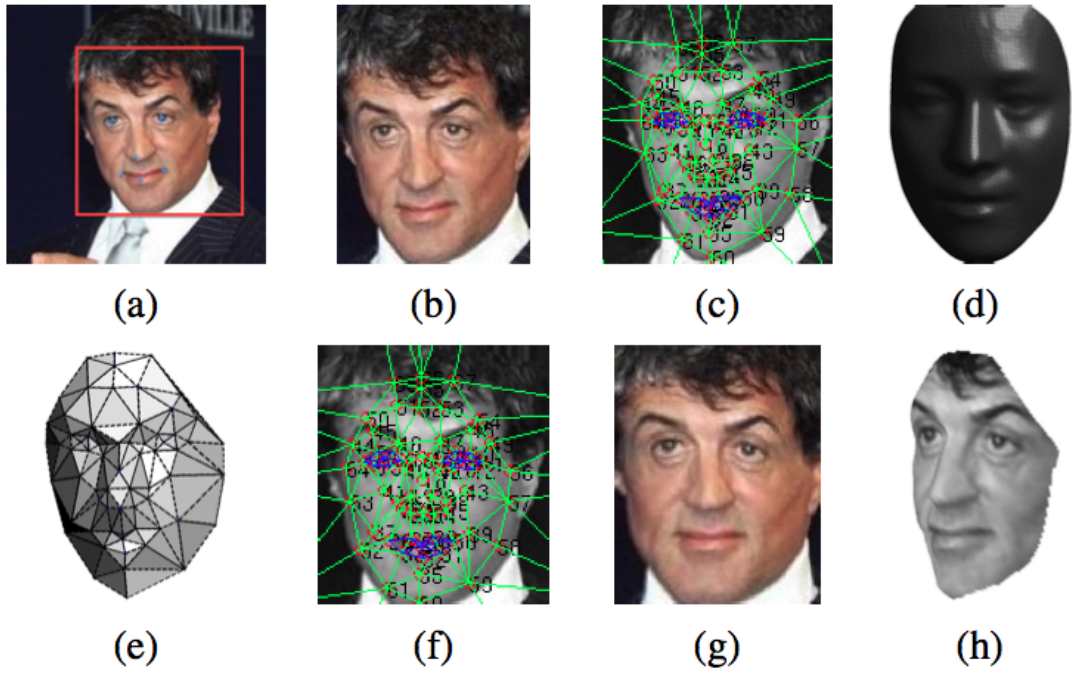
- 95.92% - ConvNet with 3D alignment (**frontalization**)
  - Simply take inner product of normalized features
- **97.00**% - Linear SVM on chi squared similarity vectors

DeepID2 trained to recognize 8.2k identities

- **99.15**% - Modified loss function to include verification
  - **Cropped patches** at different positions and scales according to 21 facial keypoints of 2D aligned image

DeepFace vs DeepID2 Face Processing

DeepFace strategy (*right*): uses 57 facial keypoints. A reference 3D shape is used to warp an induced 3D model and frontalize the face (g).



DeepID2 strategy (*botton*): uses 21 facial landmarks to define the locations of patches that are extracted at different scales. A greedy algorithm is used to filter down to 25 patches

DeepFace, Architecture and Components

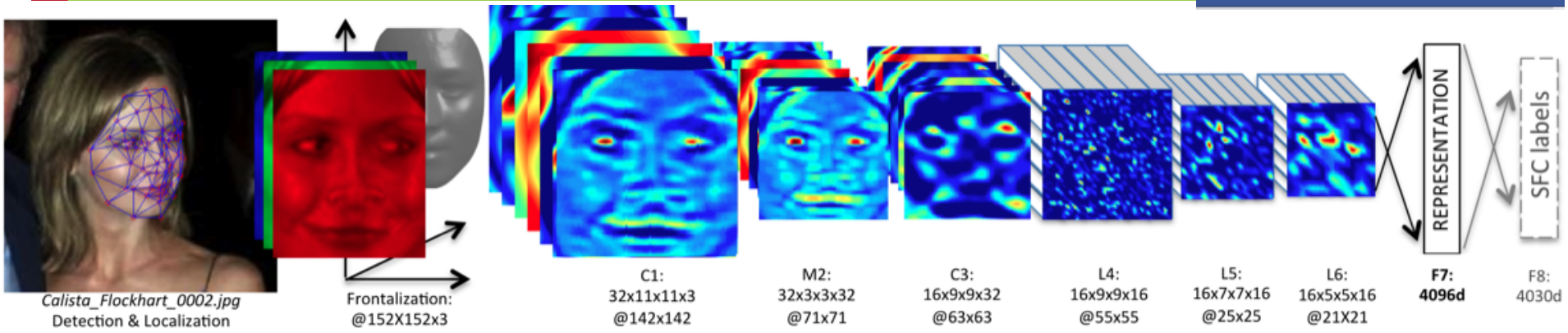


Figure 2. **Outline of the *DeepFace* architecture**. A front-end of a single convolution-pooling-convolution filtering on the rectified input, followed by three locally-connected layers and two fully-connected layers. Colors illustrate feature maps produced at each layer. The net includes more than 120 million parameters, where more than 95% come from the local and fully connected layers.

Where do gains come from ?

- 87.9% - ConvNet, centre crop of face detection
- 91.4% - 3D alignment and hand engineered features (LPBs)
- 94.3% - ConvNet with 2D alignment
- 97.0% - ConvNet with 3D alignment (frontalization)
- 97.4% - Ensemble

Other points: DeepID2 Loss Modifications

- They minimize the cross entry loss

$$\text{Ident}(f, t, \theta_{id}) = -\sum_{i=1}^{n} -p_i \log \hat{p}_i = -\log \hat{p}_t \ ,$$

- with an additional L2 verification cost

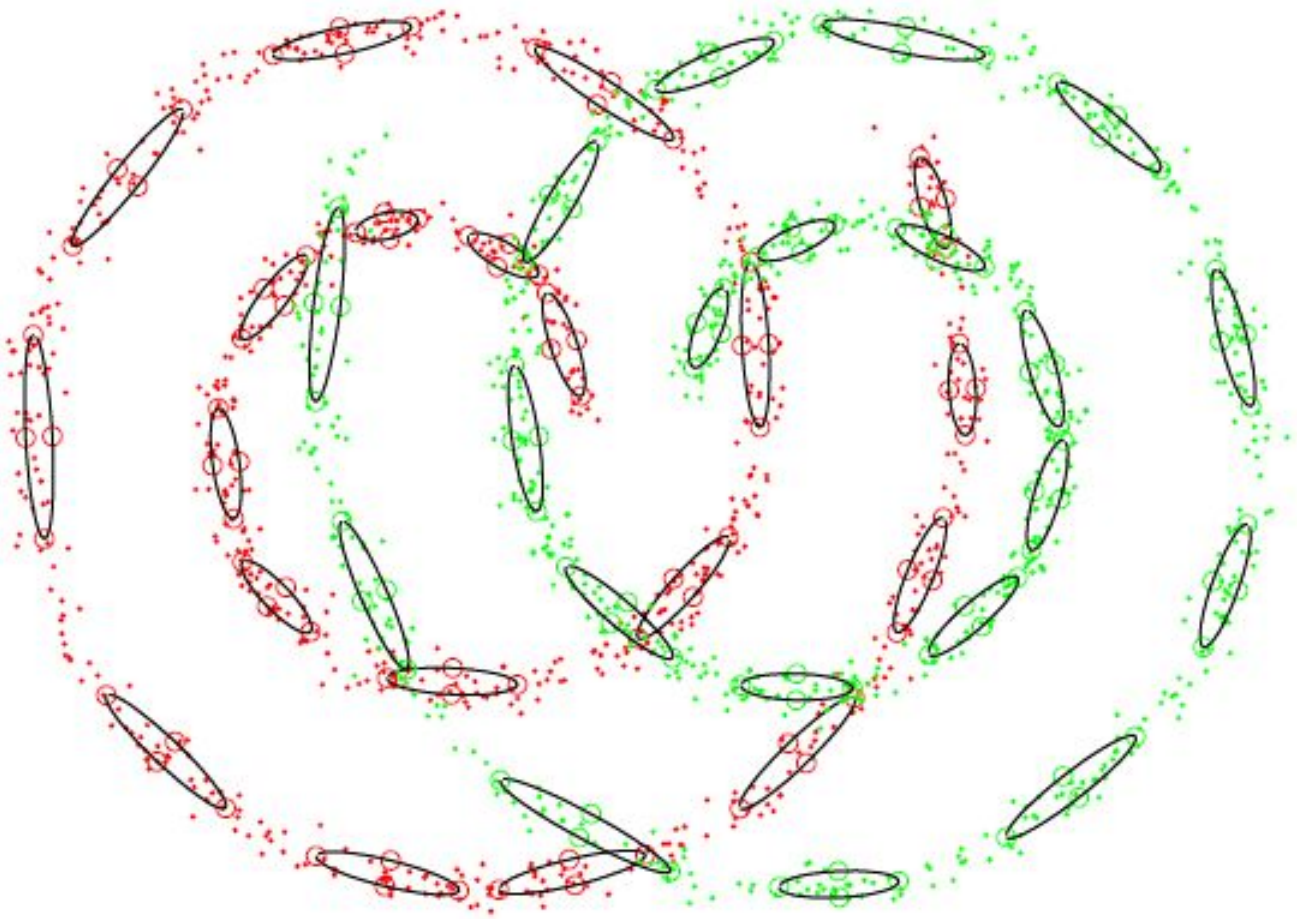
$$\text{Verif}(f_i, f_j, y_{ij}, \theta_{ve}) = \begin{cases} \frac{1}{2} \|f_i - f_j\|_2^2 & \text{if } y_{ij} = 1 \\ \frac{1}{2} \max\left(0, m - \|f_i - f_j\|_2\right)^2 & \text{if } y_{ij} = -1 \end{cases} ,$$

- Where f_i f_j are the network last layer feature vectors y_{ij}=1 means f_i and f_j were from the same identity

Les approches génératives vs. discriminatives vs hybrides

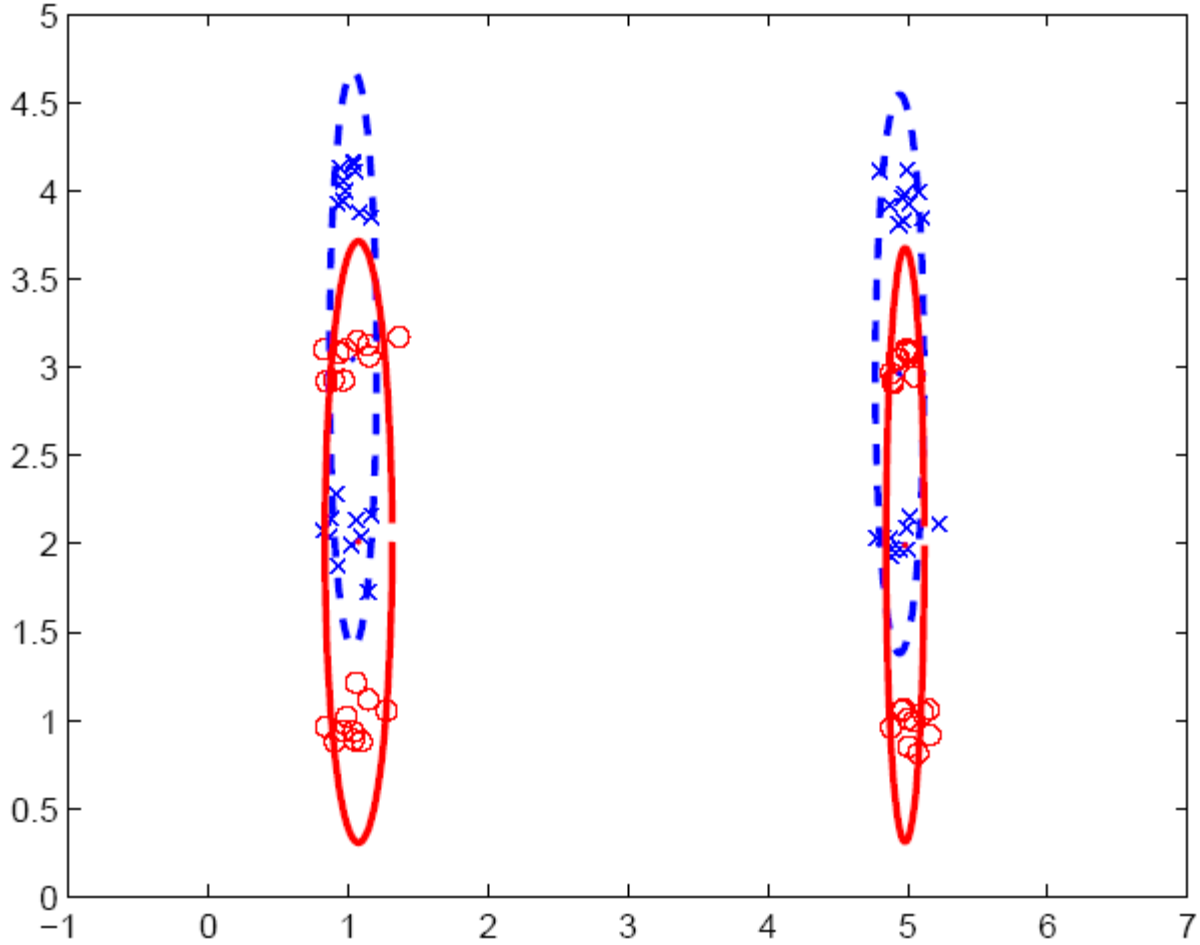
Chris Pal

Exemples : 2 spirales / 2 classes, chacun avec 20 composants



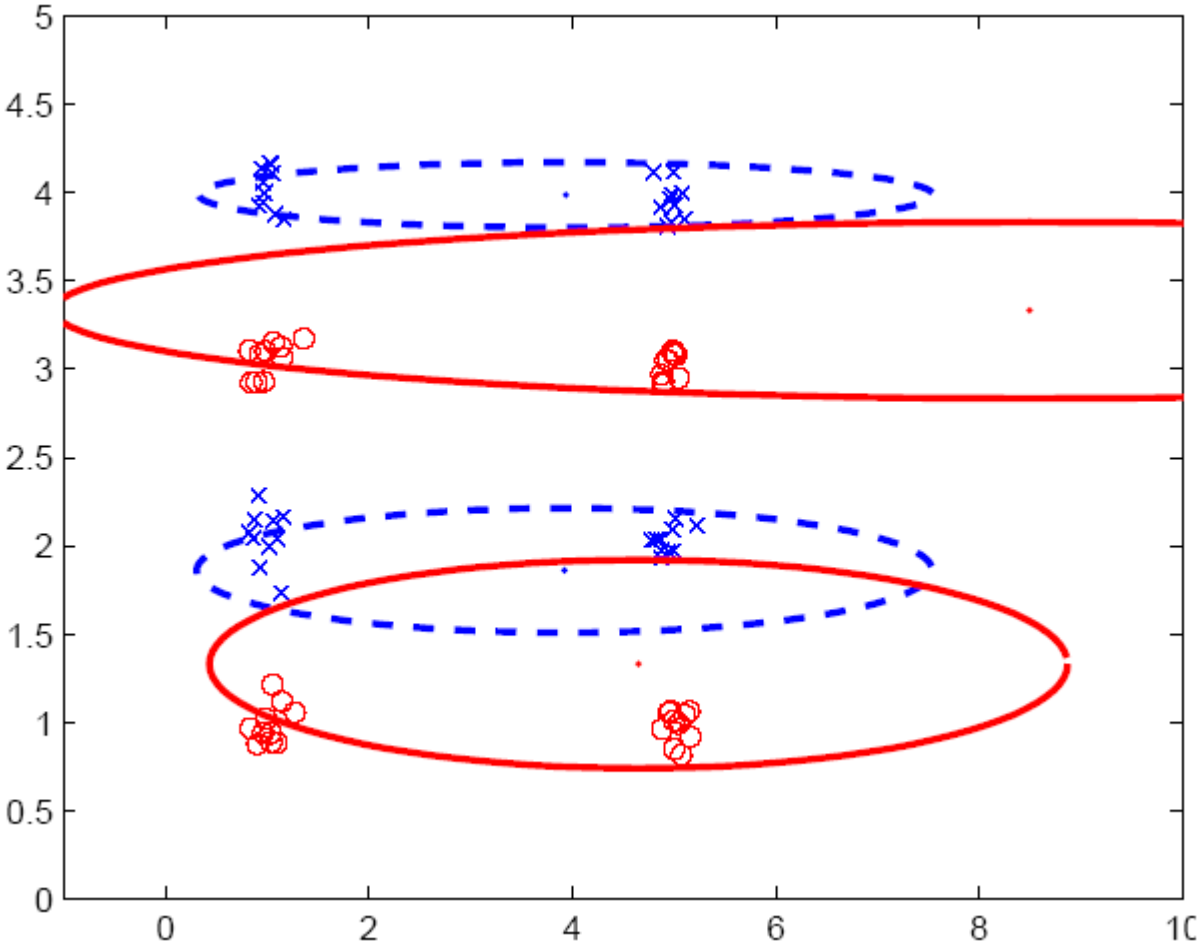
Apprentissage = simplement maximisation de la vraisemblance du modèle

Exemple : groupement



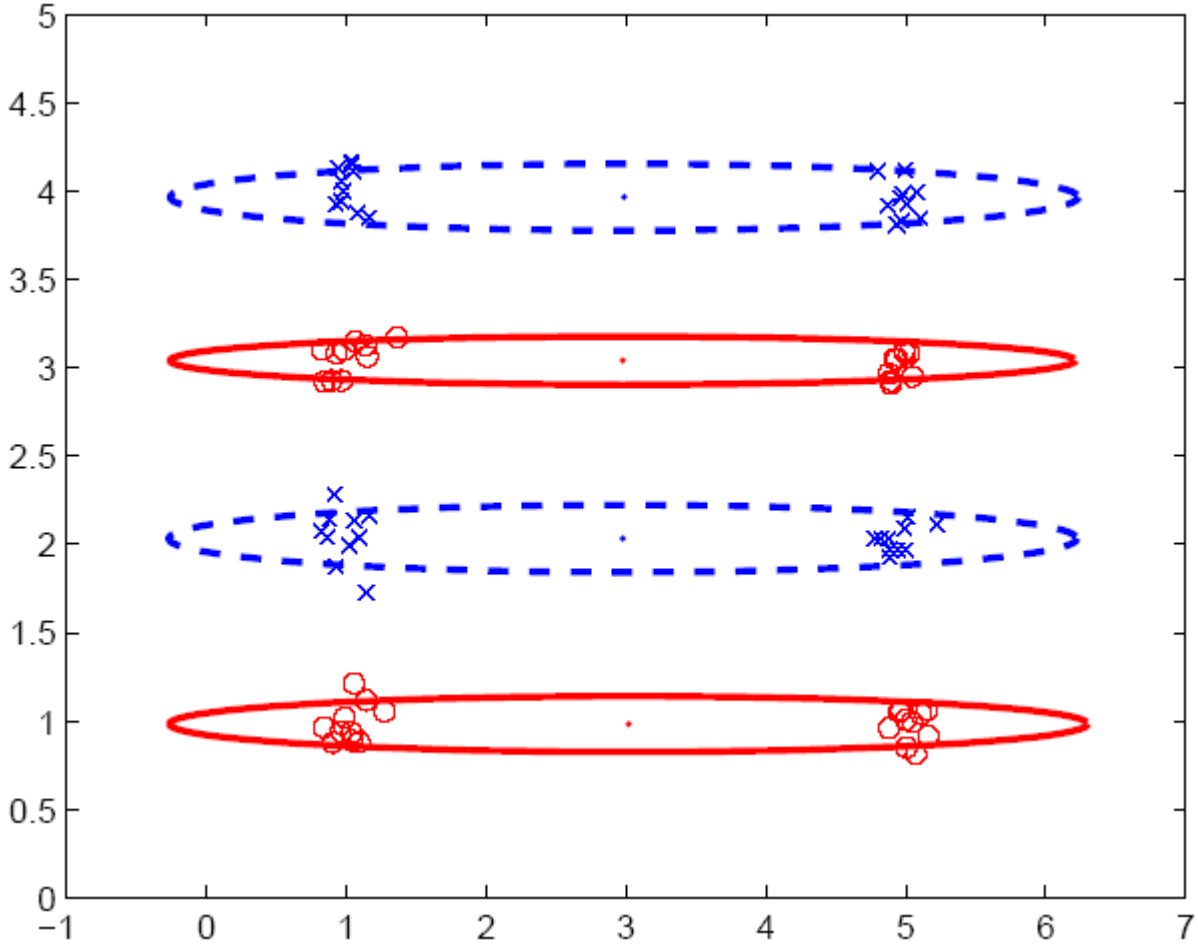
- Groupement des « o » rouges et des « x » bleus

Exemple : classification



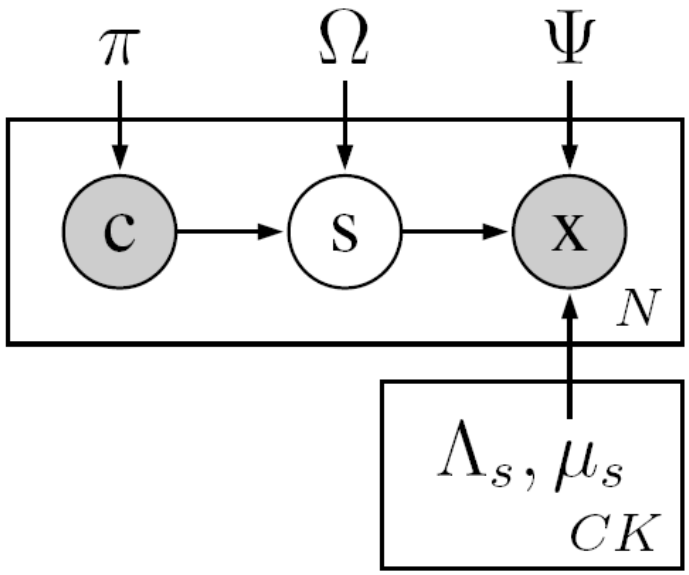
- Classification des « o » rouges et des « x » bleus

Méthode hybride



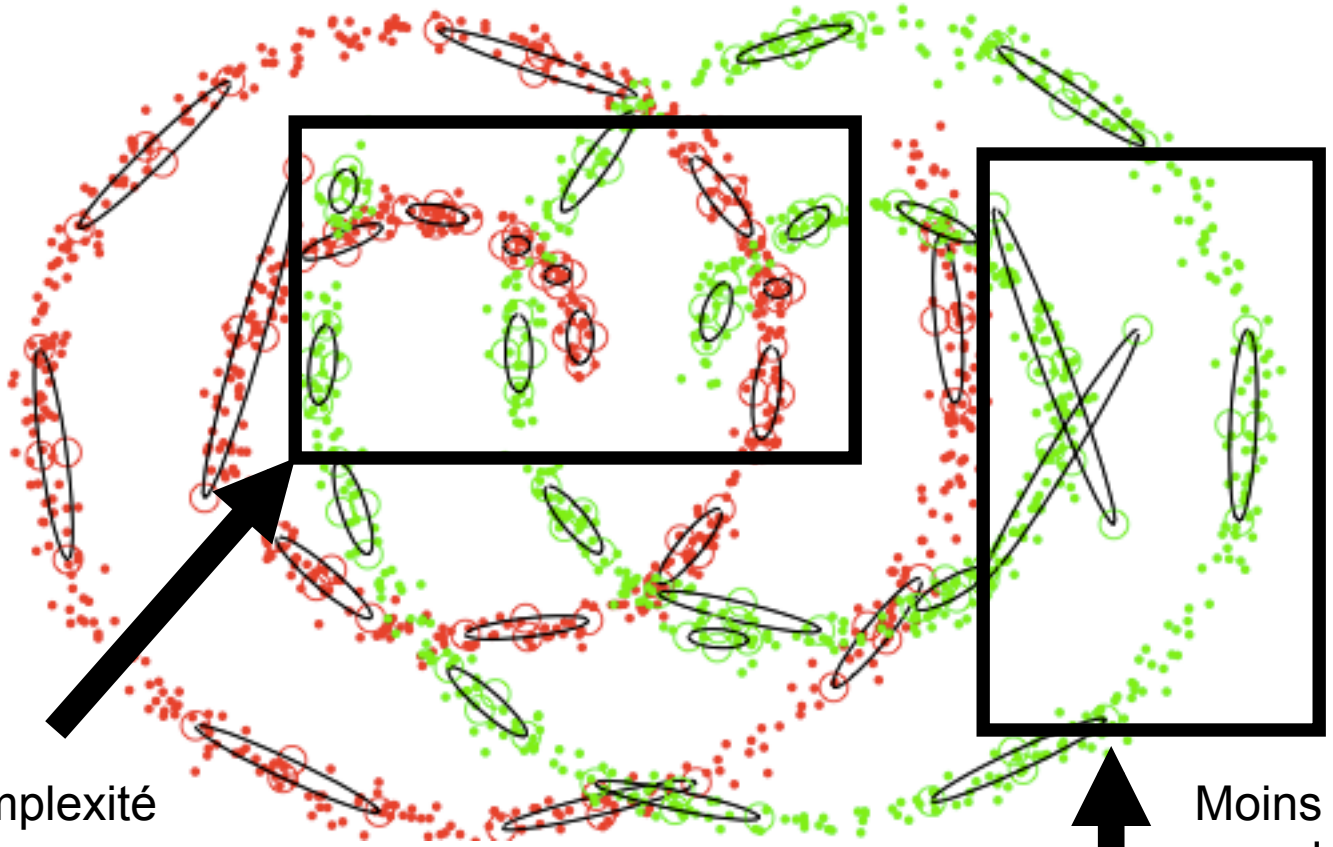
- Combinaison des méthodes de groupement et de classification

Exemple : un modèle de mélange gaussien



$$
\begin{aligned}
\mathrm{P}(c \mid \pi) &= \exp[\pi^T c] \\
\mathrm{P}(s \mid c, \Omega) &= \exp[c^T \Omega\, s] \\
\mathrm{P}(x \mid s, \mu_s, \Lambda_s, \Psi) &= \mathcal{N}_d(\mu_s, \Lambda_s \Lambda_s^T + \Psi)
\end{aligned}
$$

Approche hybride



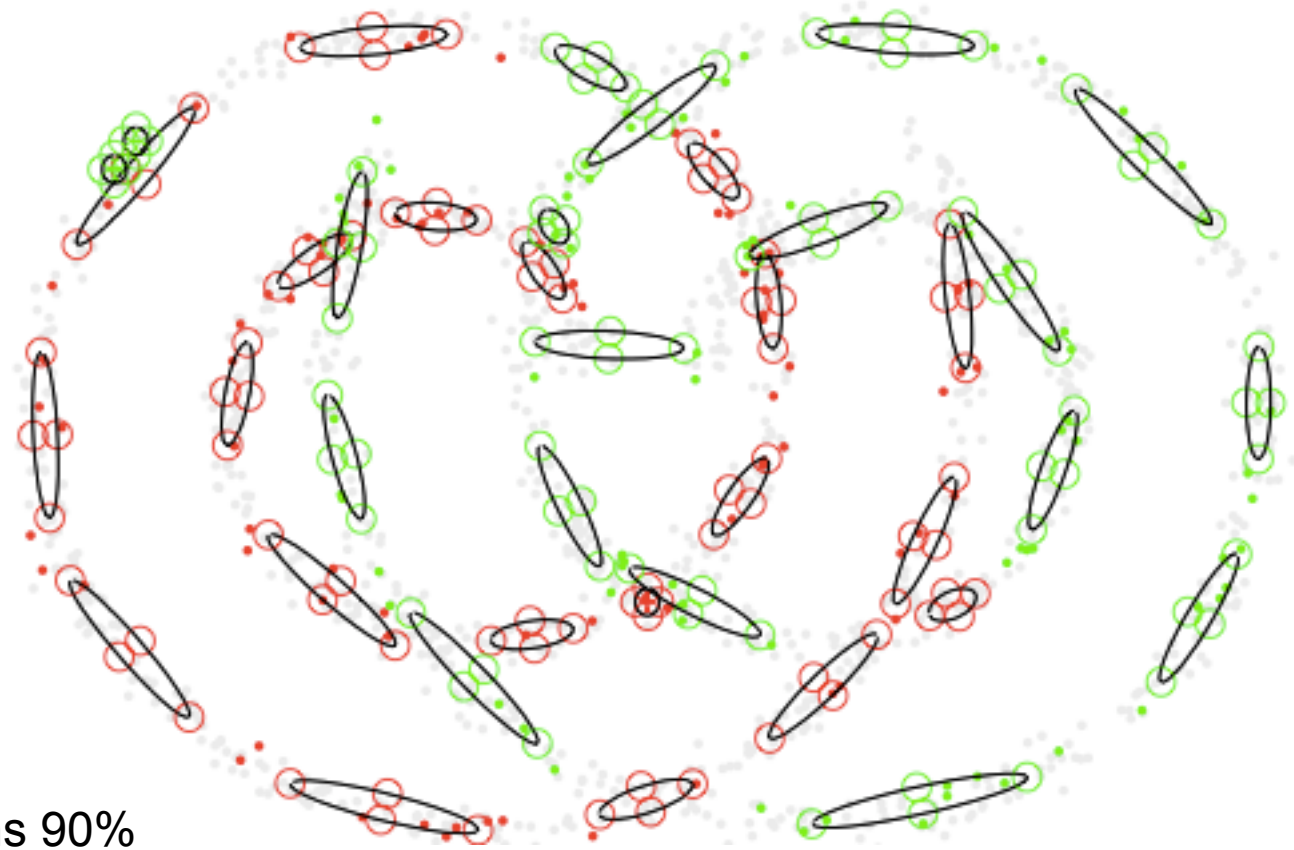
Plus de complexité

Moins de complexité

La fonction objectif

$$L(\theta) = \alpha \log P(c \mid x) + \beta \log P(x \mid c)$$

Approche hybride (semi-supervisé)



Noter: Sans 90% des étiquettes (les points en gris)

La fonction objectif

$$L(\theta) = \alpha \log P(c \mid x) + \beta \log P(x)$$

Problème : la segmentation ou la classification des pixels

(a) lasso labeling

(b) ground truth

MSR Cambridge / Berkeley Data

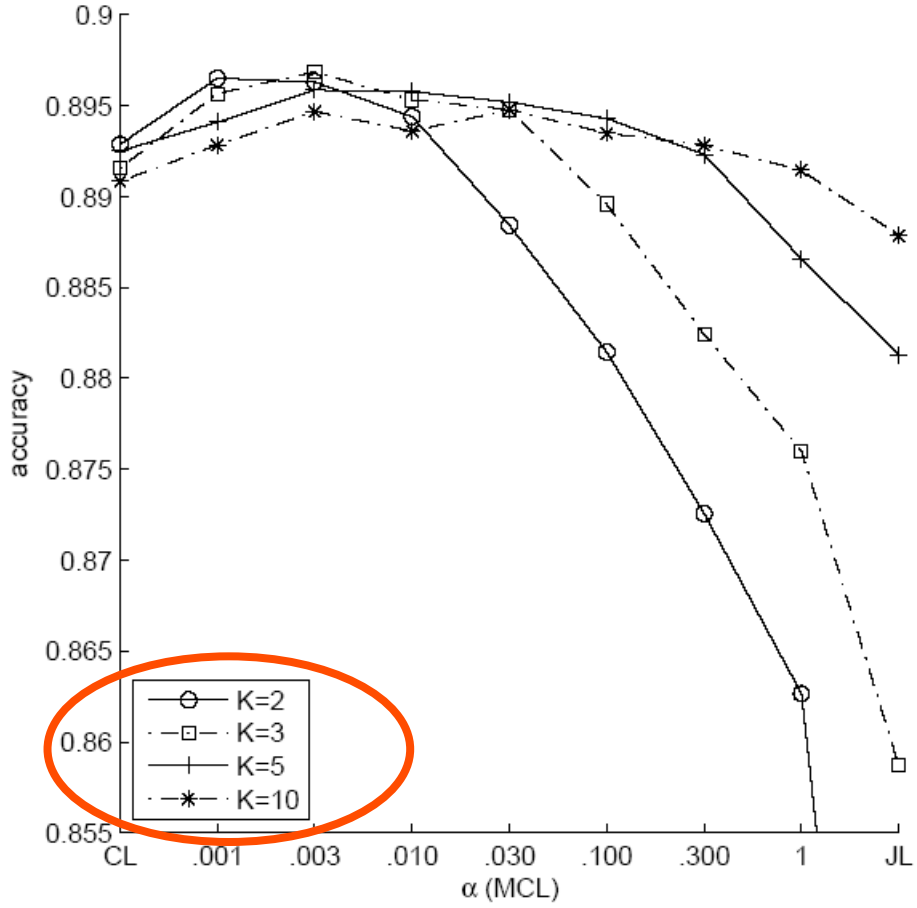
Apprentissage par descente de gradient espérée

$$\mathcal{L}^{\alpha}_{c|x,x|c} = (1+\alpha)\mathcal{L}_{c,x} - \mathcal{L}_x - \alpha\mathcal{L}_c.$$

$$f(\mu_s, \Lambda_s, \Psi) = \log \mathcal{N}_d(\mu_s, \Sigma_s = \Lambda_s \Lambda_s^T + \Psi)$$

$$\begin{aligned}
\frac{\partial}{\partial \mu_s} f &= \Sigma_s^{-1}(x - \mu_s) \\
\frac{\partial}{\partial \Lambda_s} f &= \Sigma_s^{-1}(S - \Sigma_s)\Sigma_s^{-1}\Lambda_s \\
\frac{\partial}{\partial \Psi} f &= \frac{1}{2}\operatorname{diag}\left(\Sigma_s^{-1}(S - \Sigma_s)\Sigma_s^{-1}\right)
\end{aligned}$$

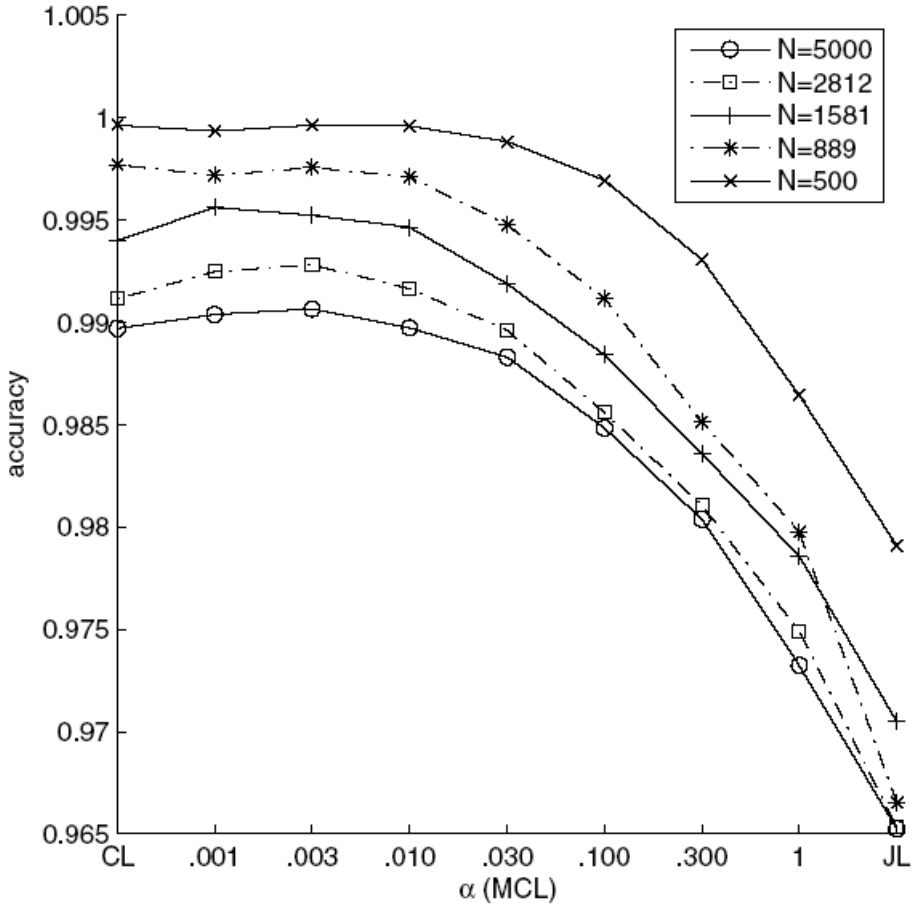
Notez : la performance vs. la complexité du modèle

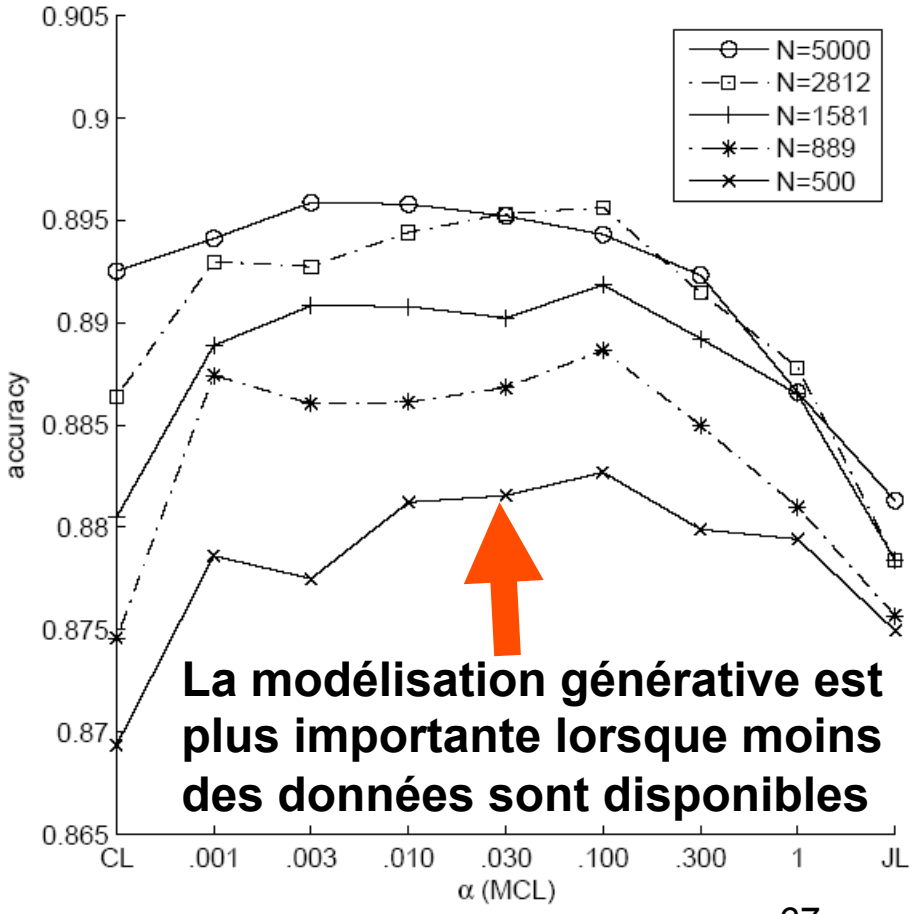


Optimisation de l'objective conjointe bénéficie plus substantiellement de la complexité supplémentaire du modèle.

Changements dans la performance avec plus de données

Précision sur l'ensemble d'apprentissage vs. Précision sur l'ensemble de test





Autoencoders

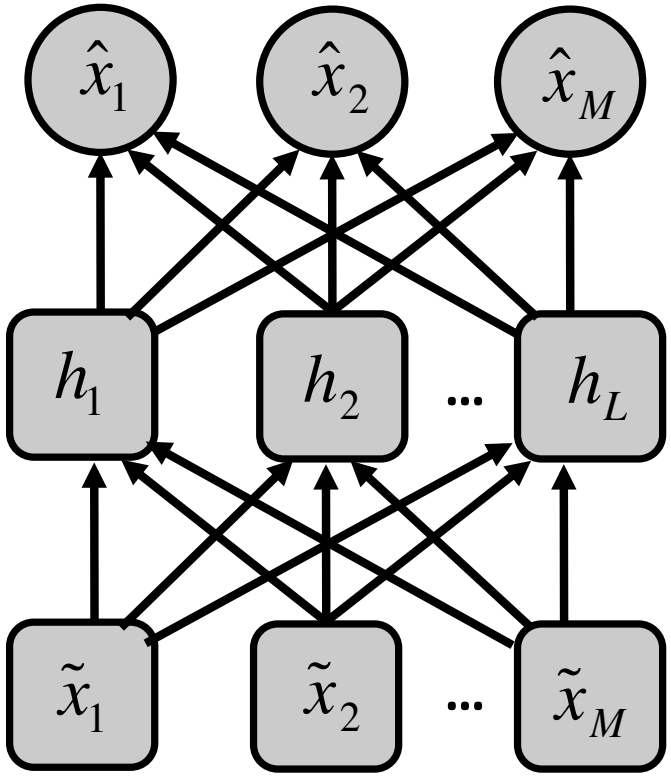
(Reconstructive Learning or Compressive Learning)

Autoencoders

- Used for unsupervised learning
- It is a network that learns an efficient coding of its input.
- The objective is simply to reconstruct the input, but through the intermediary of a compressed or reduced-dimensional representation.
- If the output is formulated using probability, the objective function is to optimize $p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}})$, that is, the probability that the model gives a random variable **x** the value $\hat{\mathbf{x}}$ given the observation $\tilde{\mathbf{x}}$, where $\hat{\mathbf{x}} = \tilde{\mathbf{x}}$.
- In other words, the model is trained to predict its own input—but it must map it through a representation created by the hidden units of a network.

A simple autoencoder

- Predicts its own input, ex. $p(\hat{\mathbf{x}} \mid \tilde{\mathbf{x}}) = p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}}; \mathbf{f}(\tilde{\mathbf{x}}))$
- Going through an encoding, $\mathbf{e} = \mathbf{h} = \mathrm{act}(\mathbf{a}^{(1)})$



where

$$\mathbf{f}(\tilde{\mathbf{x}}) = \mathbf{f}\left(\mathbf{d}\left(\mathbf{e}(\tilde{\mathbf{x}})\right)\right),$$

$$\mathbf{d} = \mathrm{out}(\mathbf{a}^{(2)}),$$

$$\mathbf{a}^{(2)} = \mathbf{W}^{\mathrm{T}}\mathbf{h} + \mathbf{b}^{(2)},$$

$$\mathbf{h} = \mathrm{act}(\mathbf{a}^{(1)}),$$

$$\mathbf{a}^{(1)} = \mathbf{W}\tilde{\mathbf{x}} + \mathbf{b}^{(1)}$$

Autoencoders

- Since the idea of an autoencoder is to compress the data into a lower-dimensional representation, the number L of hidden units used for encoding is less than the number M in the input and output layers
- Optimizing the autoencoder using the negative log probability over a data set as the objective function leads to the usual forms
- Like other neural networks it is typical to optimize autoencoders using backpropagation with mini-batch based SGD

Linear autoencoders and PCA

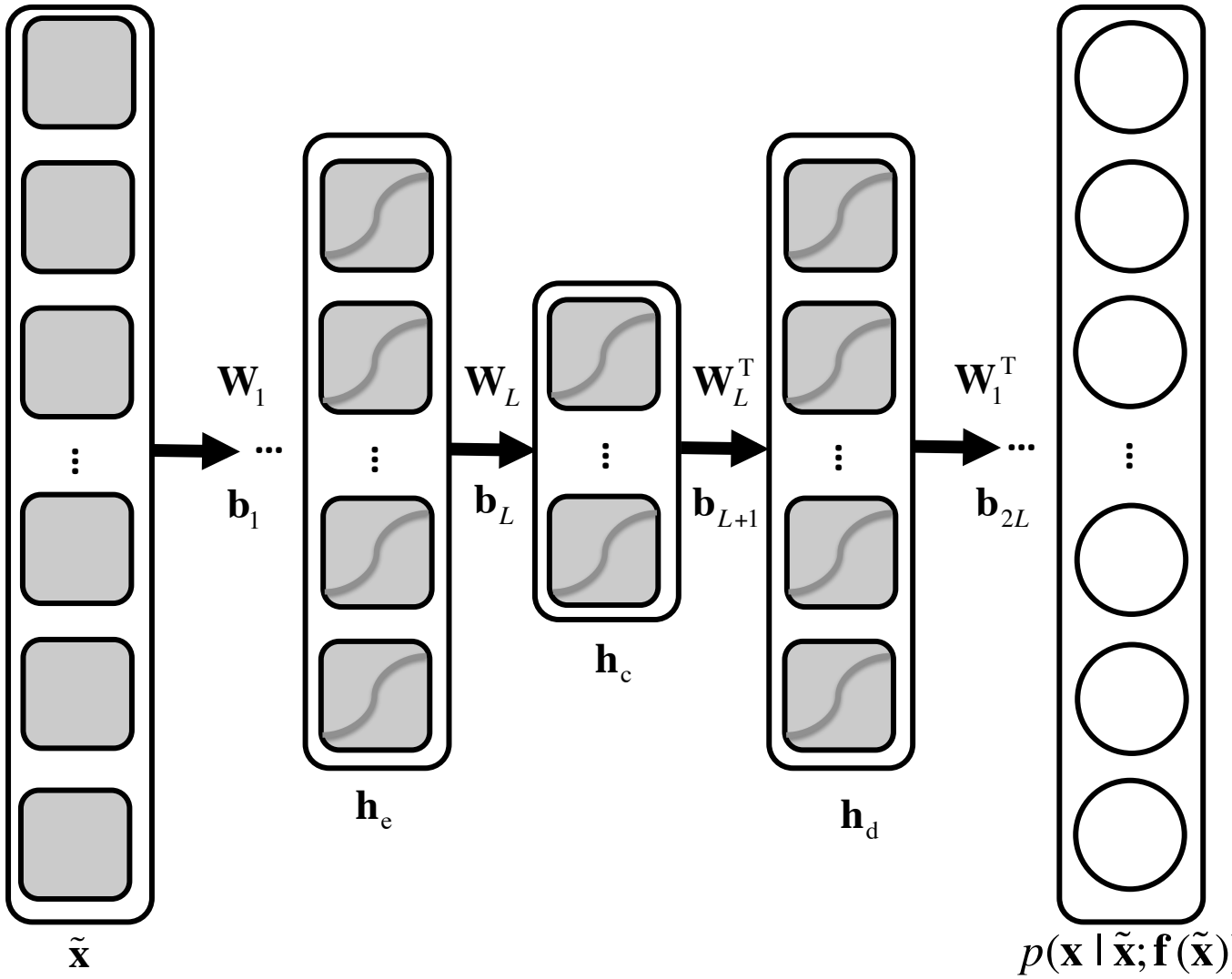
- Both the encoder activation function act() and the output activation function out() in the simple autoencoder model could be defined as the sigmoid function
- However, it can be shown that with no activation function, $\mathbf{h}^{(i)}=\mathbf{a}^{(i)}$, the resulting "linear autoencoder" will find the same subspace as PCA, (assuming a squared-error loss function and normalizing the data using mean centering)
  - Can be shown to be optimal in the sense that any model with a non-linear activation function would require a weight matrix with more parameters to achieve the same reconstruction error
- Even with non-linear activation functions such as a sigmoid, optimization finds solutions where the network operates in the linear regime, replicating the behavior of PCA
- This might seem discouraging; however, using a neural network with even one hidden layer to create much more flexible transformations, and
  - There is growing evidence deeper models can learn more useful representations

Deep autoencoders

- When building autoencoders from more flexible models, it is common to use a *bottleneck* in the network to produce an under-complete representation, providing a mechanism to obtain an encoding of lower dimension than the input.
- Deep autoencoders are able to learn low-dimensional representations with smaller reconstruction error than PCA using the same number of dimensions.
- Can be constructed by using L layers to create a hidden layer representation $\mathbf{h}_c^{(L)}$ of the data, and following this with a further L layers $\mathbf{h}_d^{(L+1)} \ldots \mathbf{h}_d^{(2L)}$ to decode the representation back into its original form
- The j=1,…,2L weight matrices for each of the i=1,…,L encoding and decoding layers are constrained by

$$\mathbf{W}_{L+i} = \mathbf{W}_{L+1-i}^{\mathrm{T}}$$

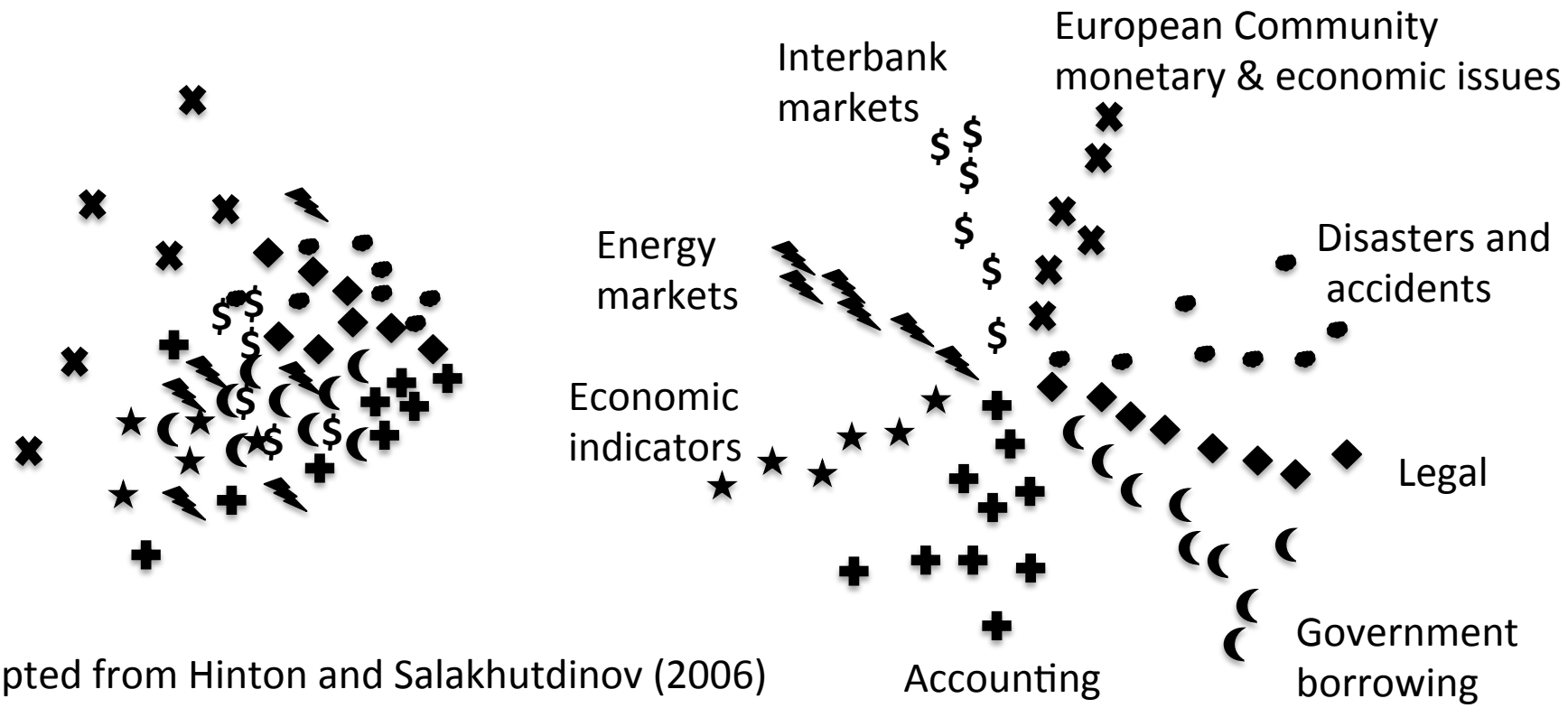
A deep autoencoder



$$\mathbf{f}(\mathbf{x}) = \mathbf{f}_{\mathbf{d}}(\mathbf{a}_{\mathbf{d}}^{(2L)}(\ldots\mathbf{h}_{\mathbf{d}}^{(L+1)}(\mathbf{a}_{\mathbf{d}}^{(L+1)}(\mathbf{h}_{\mathbf{c}}^{(L)}(\mathbf{a}_{\mathrm{e}}^{(L)}(\ldots\mathbf{h}_{\mathrm{e}}^{(1)}(\mathbf{a}_{\mathrm{e}}^{(1)}(\mathbf{x}))))))).$$

Deep autoencoders

- A comparison of data projected into a 2D space with PCA (left) vs a deep autoencoder (right) for a text dataset
- The non-linear autoencoder can arrange the learned space in such that it better separates natural groupings of the data



Adapted from Hinton and Salakhutdinov (2006)

Training autoencoders

- Deep autoencoders are an effective framework for non-linear dimensionality reduction
- It can be difficult to optimize autoencoders; being careful about activation function choice and initialization can help
- Once such a network has been built, the top-most layer of the encoder, the code layer $\mathbf{h}_c$, can be input to a supervised classification procedure
- One can pre-train a discriminative neural net with an autoencoder
- One can also use a composite loss from the start, with a reconstructive (unsupervised) and a discriminative (supervised) criterion

$$L(\theta) = (1 - \lambda) L_{\text{sup}}(\theta) + \lambda L_{\text{unsup}}(\theta)$$

  where λ is a hyperparameter that balances the two objectives
- Another approach to training autoencoders is based on pre-training by stacking two-layered restricted Boltzmann machines RBMs

Denoising autoencoders

- Autoencoders can be trained layerwise, using autoencoders as the underlying building blocks
- One can use greedy layerwise training strategies involving plain autoencoders to train deep autoencoders, but attempts to do this for networks of even moderate depth has been problematic
- Procedures based on stacking denoising autoencoders have been found to work better
- Denoising autoencoders are trained to remove different types of noise that has been added synthetically to their input
- Autoencoder inputs can be corrupted with noise such as: Gaussian noise; masking noise, where some elements are set to 0; and salt-and-pepper noise, where some elements are set to minimum and maximum input values (such as 0 and 1)

Bibliographic Notes & Further Reading

Autoencoders

- Hinton and Salakhutdinov (2006) noted that it has been known since the 1980s that deep autoencoders, optimized through backpropagation, could be effective for non-linear dimensionality reduction.
- The key limiting factors were the small size of the datasets used to train them, coupled with low computation speeds; plus the old problem of local minima.
- By 2006, datasets such as the MNIST digits and the 20 Newsgroups collection were large enough, and computers were fast enough, for Hinton and Salakhutdinov to present compelling results illustrating the advantages of deep autoencoders over principal component analysis.
  - Their experimental work used generative pre-training to initialize weights to avoid problems with local minima.

Bibliographic Notes & Further Reading

Autoencoders

- Bourlard and Kamp (1988) provide a deep analysis of the relationships between autoencoders and principal component analysis.
- Vincent et al. (2010) proposed stacked denoising autoencoders and found that they outperform both stacked standard autoencoders and models based on stacking restricted Boltzmann machines.
- Cho and Chen (2014) produced state-of-the-art results on motion capture sequences by training deep autoencoders with rectified linear units using hybrid unsupervised and supervised learning.

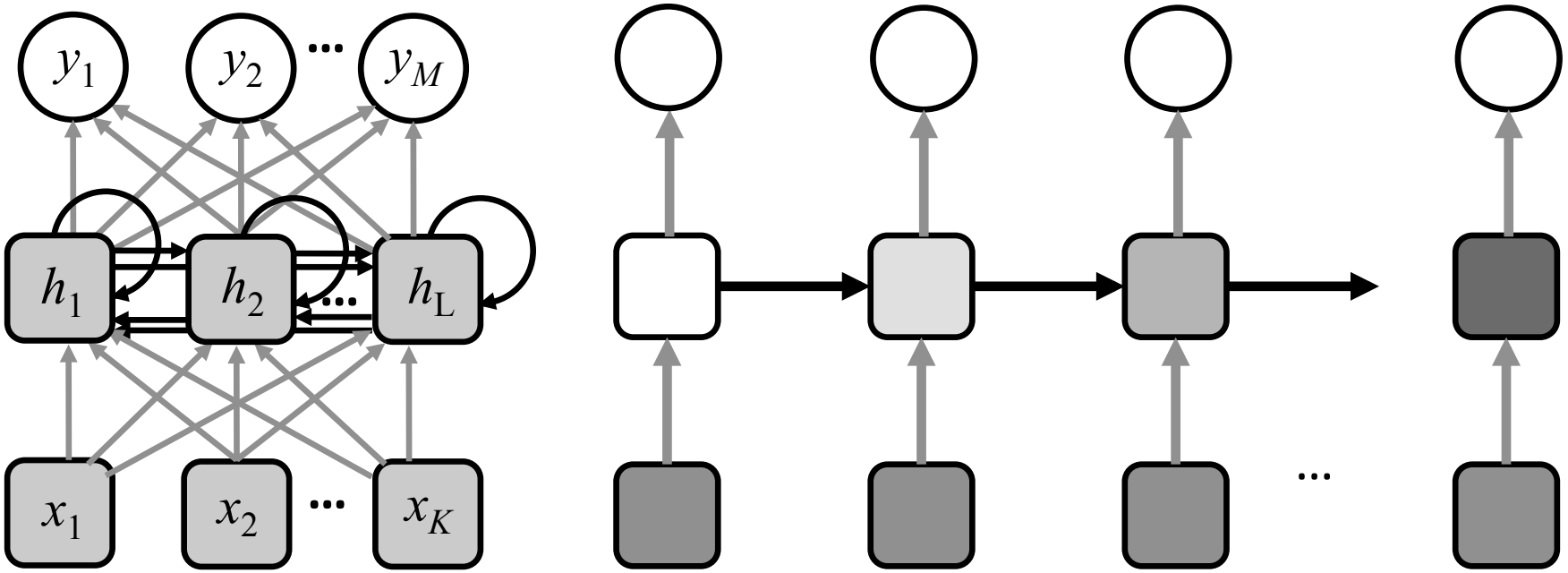
Recurrent neural networks

Recurrent neural networks

- Recurrent neural networks are networks with connections that form directed cycles.
- As a result, they have an internal state, which makes them prime candidates for tackling learning problems involving sequences of data—such as handwriting recognition, speech recognition, and machine translation.
- A feedforward network can be transformed into a recurrent network by adding connections from all hidden units h_i to h_j.
- Each hidden unit has connections to both itself and other hidden units.
- Imagine unfolding a recurrent network over time by following the sequence of steps that perform the underlying computation.
- Like a hidden Markov model, a recurrent network can be unwrapped and implemented using the same weights and biases at each step to link units over time.

Recurrent neural networks (RNNs)

- An RNN can be unwrapped and implemented using the same weights and biases at each step to link units over time as shown below
- The resulting unwrapped RNN is similar to a hidden Markov model, but keep in mind that the hidden units in RNNs are not stochastic



Recurrent neural networks (RNNs)

- Recurrent neural networks apply linear matrix operations to the current observation and the hidden units from the previous time step, and the resulting linear terms serve as arguments of activation functions act():

$$\mathbf{h}_t = \text{act}(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{o}_t = \text{act}(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$$

- The same matrix $\mathbf{U}_h$ is used at each time step
- The hidden units in the previous step $\mathbf{h}_{t-1}$ influence the computation of $\mathbf{h}_t$ where the current observation contributes to a $\mathbf{W}_h\mathbf{x}$ term that is combined with $\mathbf{U}_h\mathbf{h}_{t-1}$ and bias $\mathbf{b}_h$ terms
- Both $\mathbf{W}_h$ and $\mathbf{b}_h$ are typically replicated over time
- The output layer is modeled by a classical neural network activation function applied to a linear transformation of the hidden units, the operation is replicated at each step.

The loss, exploding and vanishing gradients

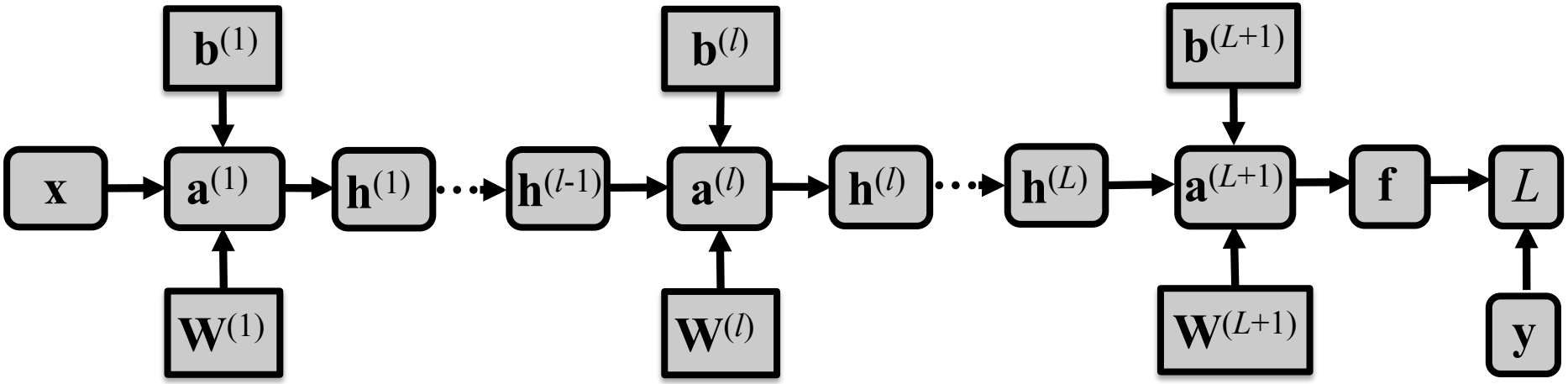
- The loss for a particular sequence in the training data can be computed either at each time step or just once, at the end of the sequence.
- In either case, predictions will be made after many processing steps and this brings us to an important problem.
- The gradient for feedforward networks decomposes the gradient of parameters at layer l into a term that involves the product of matrix multiplications of the form $\mathbf{D}^{(l)}\mathbf{W}^{\mathsf{T}(l+1)}$ (see the analysis for feedforward networks above)
- A recurrent network uses the same matrix at each time step, and over many steps the gradient can very easily either diminish to zero or explode to infinity—just as the magnitude of any number other than one taken to a large power either approaches zero or increases indefinitely

Backprop Reprise

Visualizing backpropagation

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \qquad \mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \qquad \mathbf{a}^{(L+1)} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}$$

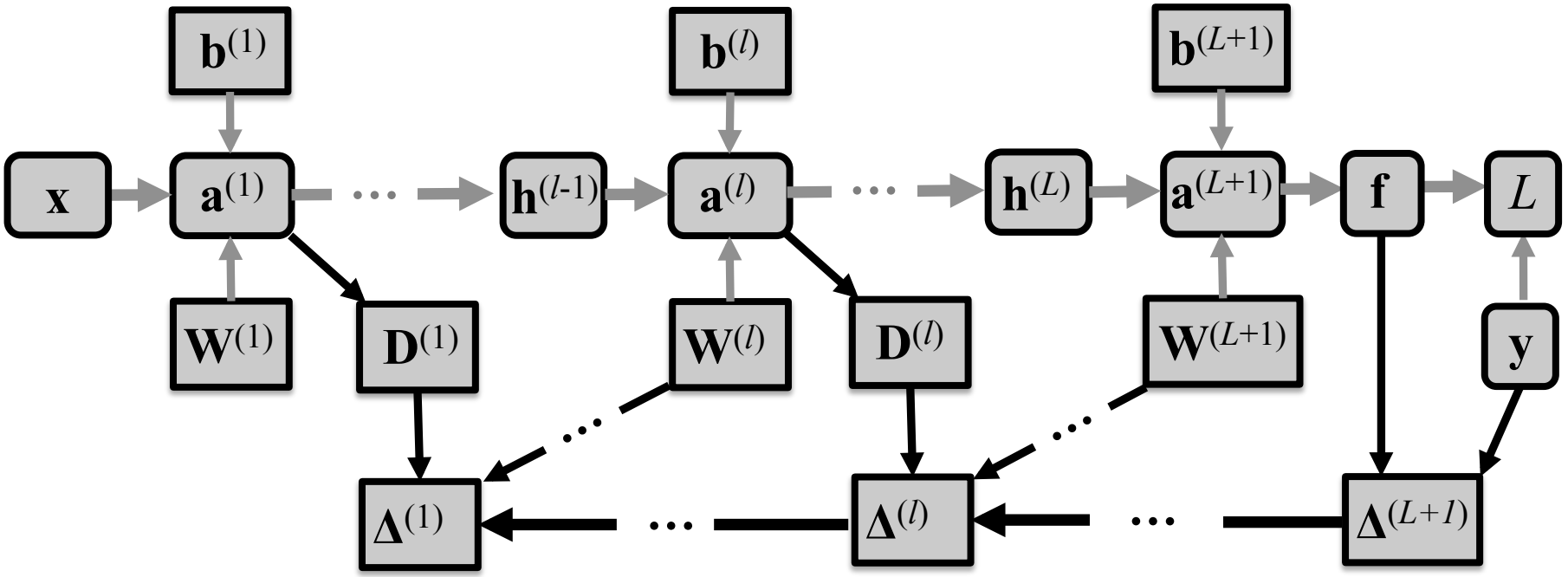
$$\mathbf{h}^{(1)} = \text{act}(\mathbf{a}^{(1)}) \qquad \mathbf{h}^{(l)} = \text{act}(\mathbf{a}^{(l)}) \qquad \mathbf{f} = \text{out}(\mathbf{a}^{(L+1)})$$



- In the forward propagation phase we compute terms of the form above
- The figure above is a type of computation graph, (which is different from the probability graphs we saw earlier)

Visualizing backpropagation

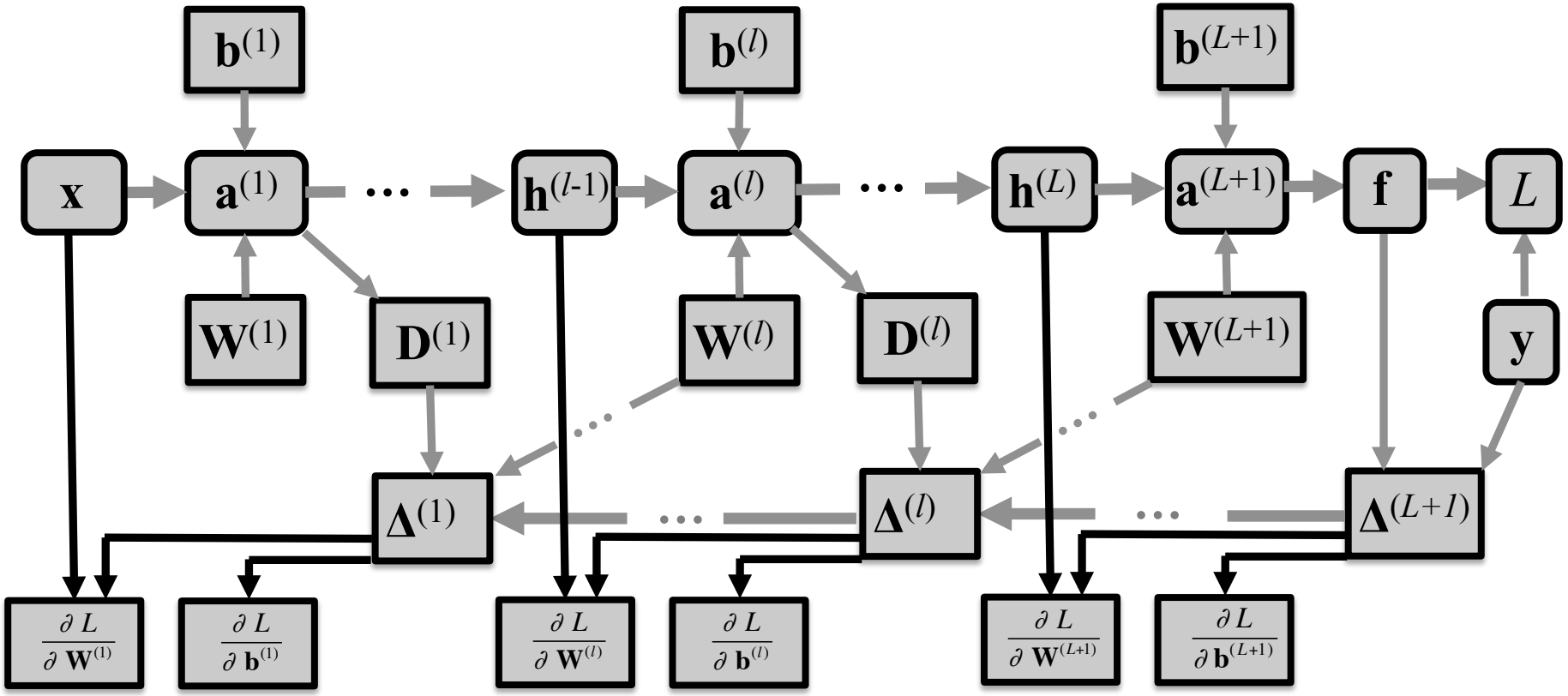
- In the backward propagation phase we compute terms of the form below



$$\Delta^{(l)} = \mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)} \qquad \frac{\partial\,\mathbf{h}^{(l)}}{\partial\,\mathbf{a}^{(l)}} = \mathbf{D}^{(l)} \qquad \frac{\partial\,\mathbf{a}^{(l+1)}}{\partial\,\mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)} \qquad \frac{\partial\,L}{\partial\,\mathbf{a}^{(L+1)}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] = -\Delta^{(L+1)}$$

Visualizing backpropagation

- We update the parameters in our model using the simple computations below



$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = -\Delta^{(1)}\mathbf{x}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(1)}} = -\Delta^{(1)} \qquad \frac{\partial L}{\partial \mathbf{W}^{(l)}} = -\Delta^{(l)}\mathbf{h}_{(l-1)}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(l)}} = -\Delta^{(l)}$$

A general form for gradients

- The gradients for the k^{th} vector of parameters of the l^{th} network layer can therefore be computed using products of matrices of the following form

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(l)}} = -\mathbf{H}_k^{(l-1)} \mathbf{D}^{(l)} \mathbf{W}^{\mathrm{T}(l+1)} \cdots \mathbf{D}^{(L)} \mathbf{W}^{\mathrm{T}(L+1)} \boldsymbol{\Delta}^{(L+1)}, \qquad \frac{\partial L}{\partial \boldsymbol{\theta}^{(l)}} = -\boldsymbol{\Delta}^{(l)} \hat{\mathbf{h}}_{(l-1)}^{\mathrm{T}}$$

- When l=1, $\hat{\mathbf{h}}_{(0)} = \hat{\mathbf{x}}$, the input data with a 1 appended
- Note: since **D** is usually diagonal the corresponding matrix-vector multiply can be transformed into an element-wise product $\circ$ by extracting the diagonal for **d**

$$\boldsymbol{\Delta}^{(l)} = \mathbf{D}^{(l)} (\mathbf{W}^{\mathrm{T}(l+1)} \boldsymbol{\Delta}^{(l+1)}) = \mathbf{d}^{(l)} \circ (\mathbf{W}^{\mathrm{T}(l+1)} \boldsymbol{\Delta}^{(l+1)})$$

Dealing with exploding gradients

- The use of L_1 or L_2 regularization can mitigate the problem of exploding gradients by encouraging weights to be small.
- Another strategy is to simply detect if the norm of the gradient exceeds some threshold, and if so, scale it down.
- This is sometimes called gradient (norm) clipping where for a gradient vector $\mathbf{g}$ and threshold T,

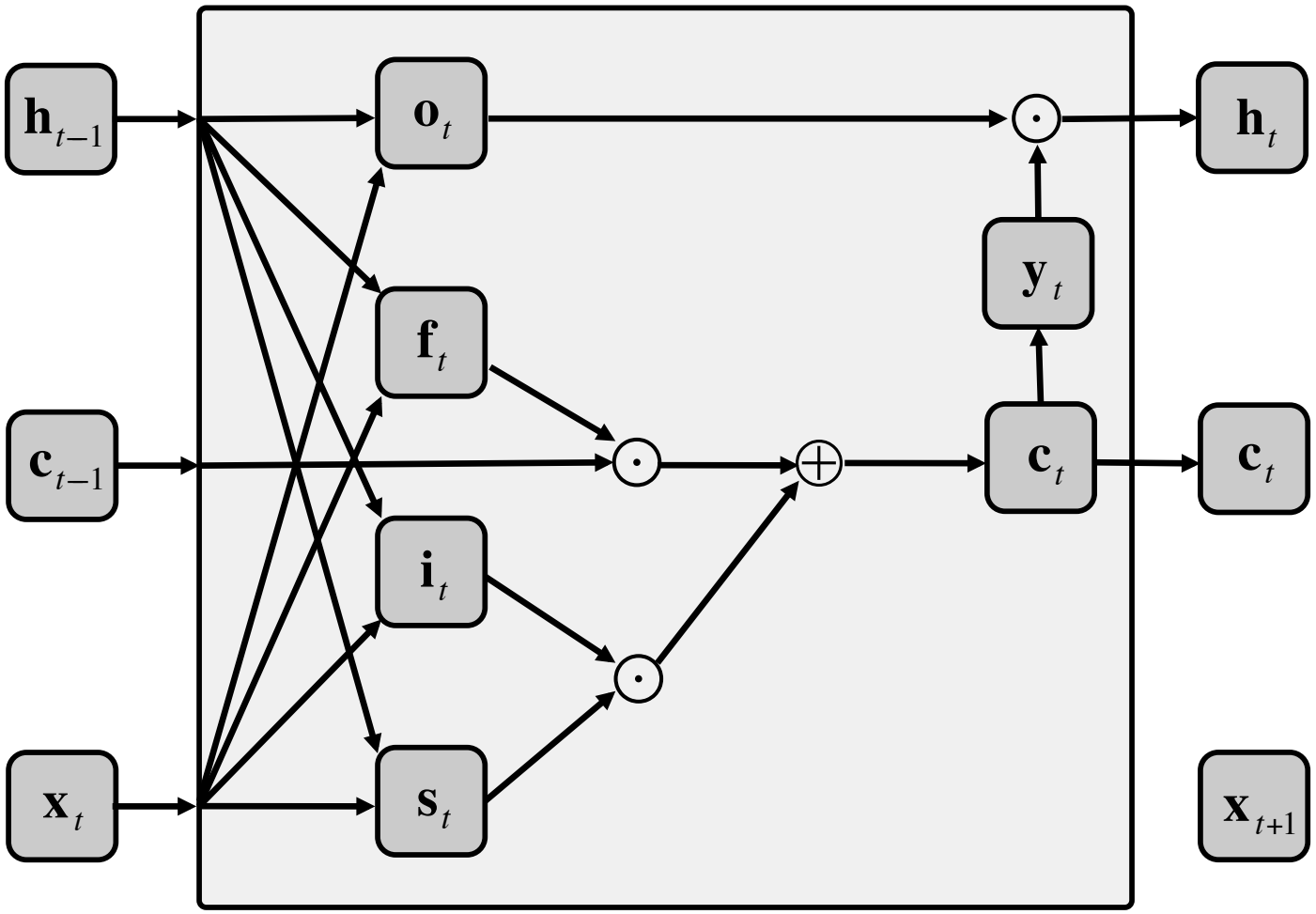
$$\text{if } \|\mathbf{g}\| \geq T \text{ then } \mathbf{g} \leftarrow \frac{T}{\|\mathbf{g}\|}\mathbf{g}$$

where T is a hyperparameter, which can be set to the average norm over several previous updates where clipping was not used.

LSTMs and vanishing gradients

- The so-called “long short term memory” (LSTM) RNN architecture was specifically created to address the vanishing gradient problem.
- Uses a combination of hidden units, elementwise products and sums between units to implement gates that control “memory cells”.
- Memory cells are designed to retain information without modification for long periods of time.
- They have their own input and output gates, which are controlled by learnable weights that are a function of the current observation and the hidden units at the previous time step.
- As a result, *backpropagated error terms from gradient computations can be stored and propagated backwards without degradation*.
- The original LSTM formulation consisted of *input gates* and *output gates*, but *forget gates* and “peephole weights” were added later.
- Below we present the most popular variant of LSTM RNNs which does not include peephole weights, but which does use forget gates.
- The architecture is complex, but has produced state-of-the-art results on a wide variety of problems.

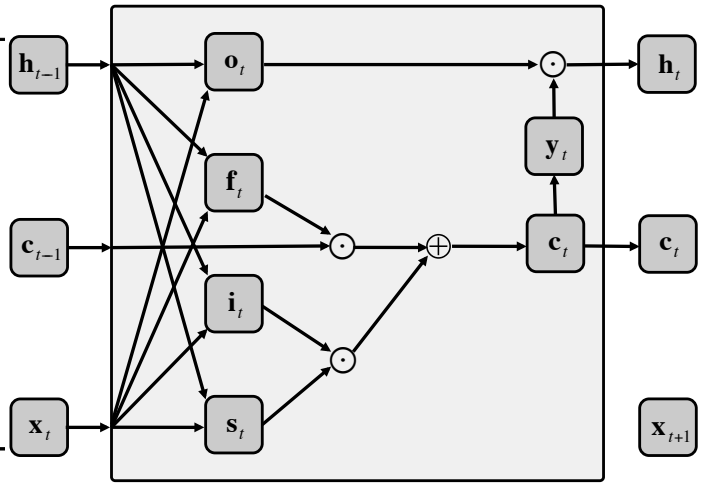
LSTM achitecture



LSTM achitecture

- At each time step there are three types of gates: input $\mathbf{i}_t$, forget $\mathbf{f}_t$, and output $\mathbf{o}_t$.
- Each are a function of both the underlying input $\mathbf{x}_t$ at time *t* as well as the hidden units at time *t*-1, $\mathbf{h}_{t-1}$
- Each gate multiplies $\mathbf{x}_t$ by its own gate specific **W** matrix, by its own **U** matrix, and adds its own bias vector **b**.
- This is usually followed by the application of a sigmoidal elementwise non-linearity.

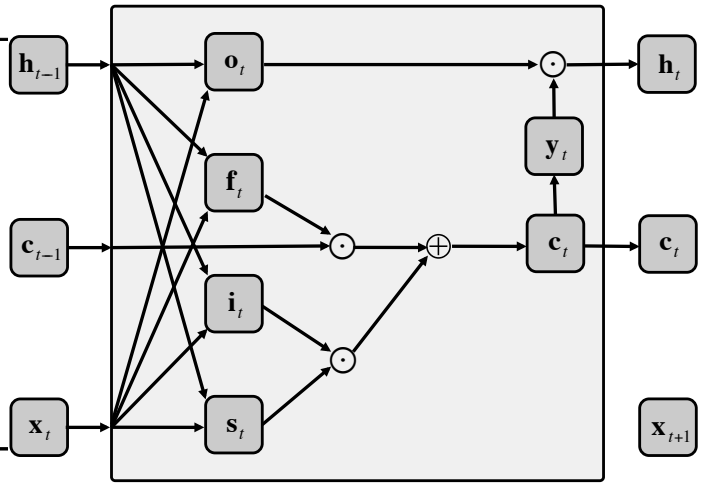
LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c)$



LSTM achitecture

- At each time step *t*, input gates $\mathbf{i}_t$ are used to determine when a potential input given by $\mathbf{s}_t$ is important enough to be placed into the memory unit or cell, $\mathbf{c}_t$
- Forget gates $\mathbf{f}_t$ allow memory unit content to be erased
- Output gates $\mathbf{o}_t$ determine whether $\mathbf{y}_t$, the content of the memory units transformed by activation functions, should be placed in the hidden units $\mathbf{h}_t$
- Typical gate activation functions and their dependencies are shown below

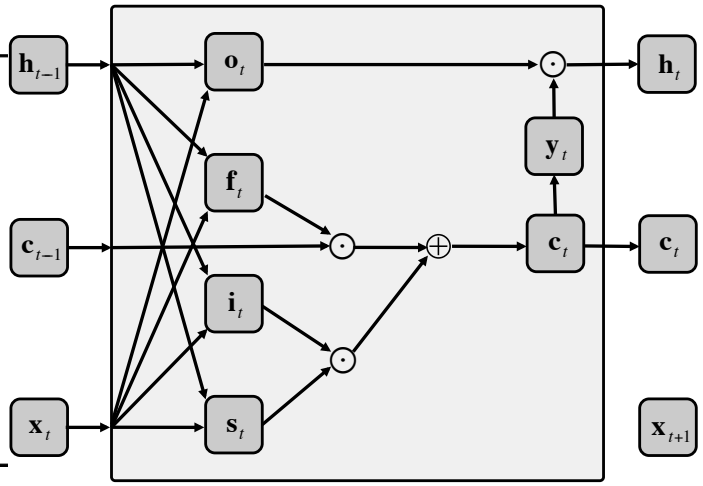
LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$



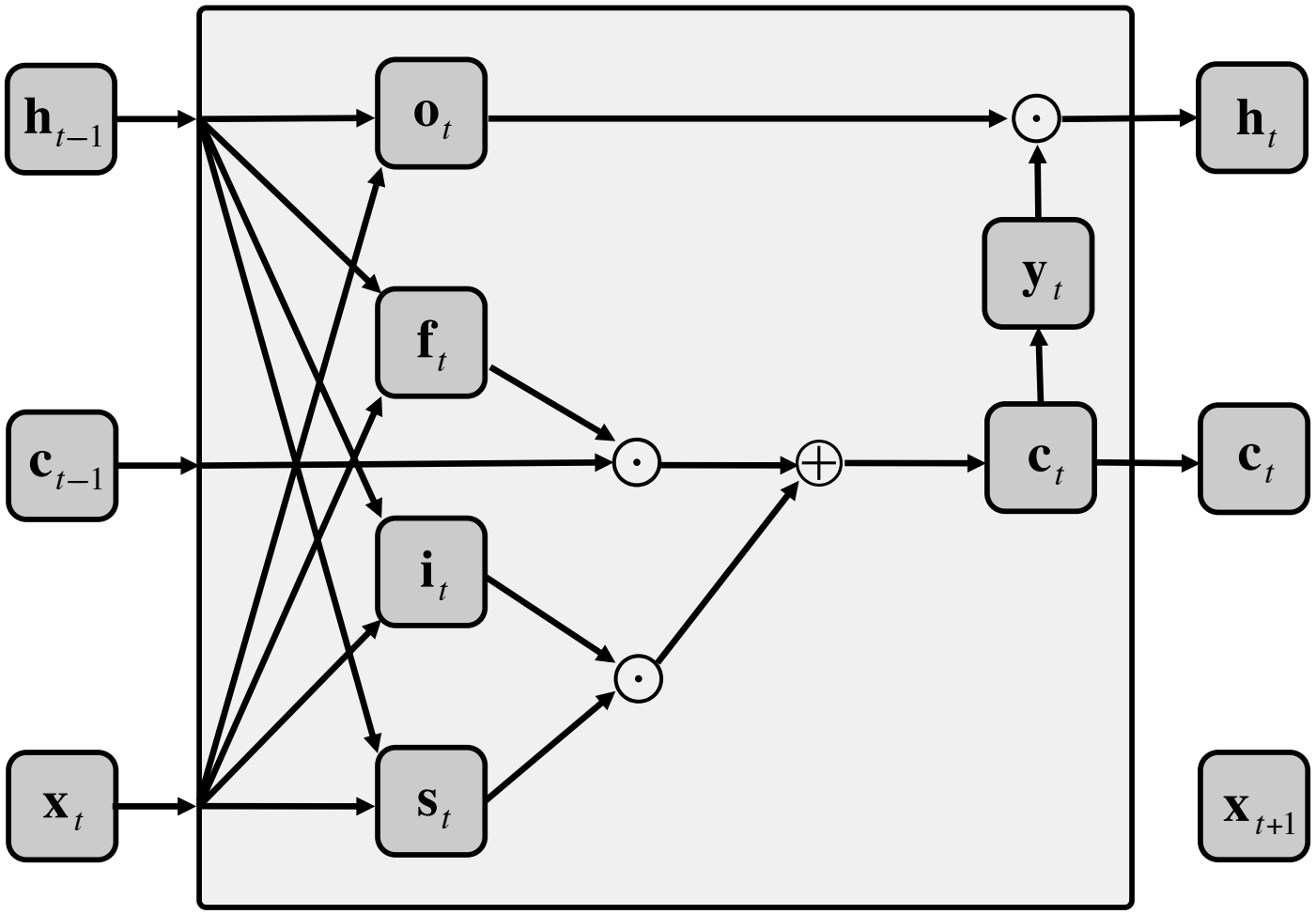
LSTM achitecture

- The final gating is implemented as an elementwise product between the output gate and the transformed memory contents, $\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
- Memory units are typically transformed by the tanh function prior to the gated output, such that $\mathbf{y}_t = \tanh(\mathbf{c}_t)$
- Memory units or cells are updated by $\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$ an elementwise product between the forget gates and the previous contents of the memory units, plus the elementwise product of the input gates and the new potential inputs .

LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$



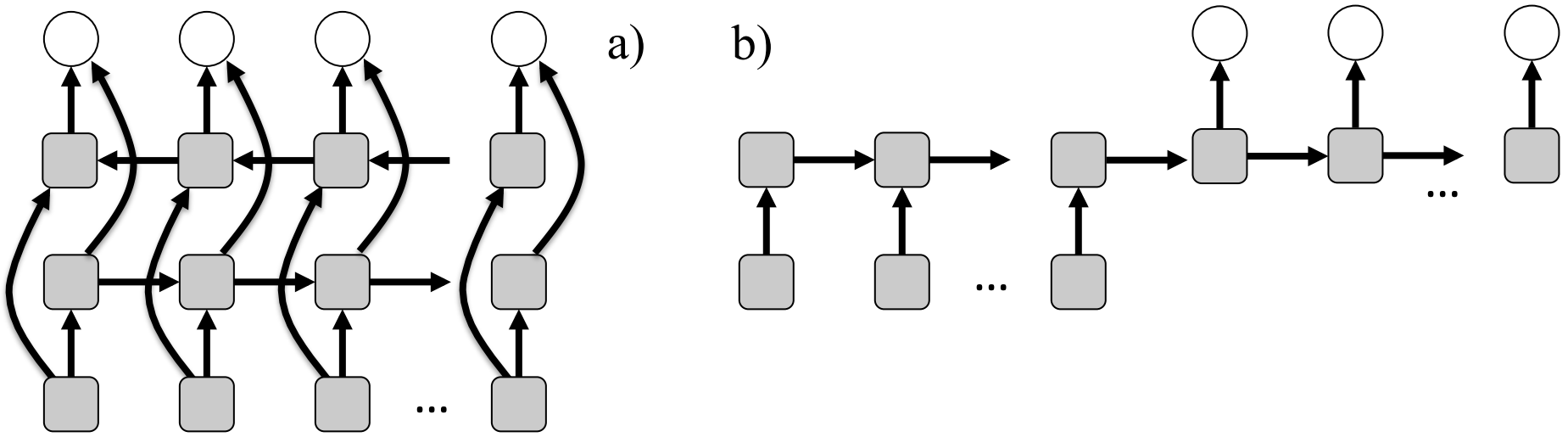
LSTM achitecture



Other RNN architectures

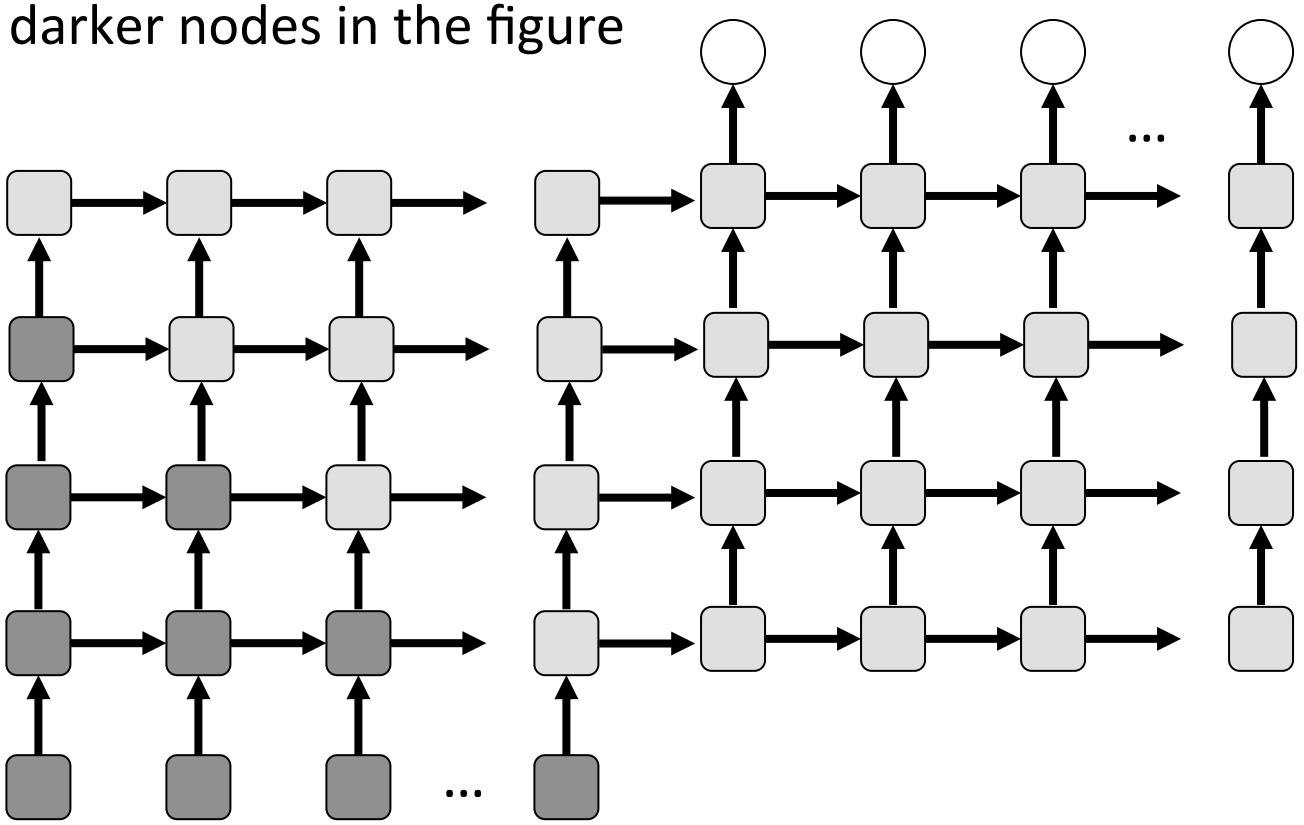
a) Recurrent networks can be made bidirectional, propagating information in both directions
   - They have been used for a wide variety of applications, including protein secondary structure prediction and handwriting recognition

b) An "encoder-decoder" network creates a fixed-length vector representation for variable-length inputs, the encoding can be used to generate a variable-length sequence as the output
   - Particularly useful for machine translation



Deep encoder-decoder architectures

- Given enough data, a deep encoder-decoder architecture such as that below can yield results that compete with translation systems that have been hand-engineered over decades of research.
- The connectivity structure means that partial computations in the model can flow through the graph in a wave, illustrated by the darker nodes in the figure



Stochastic methods

Boltzmann machines

- Are a type of Markov random field often used for unsupervised learning
- Unlike the units of a feedforward neural network, the units in Boltzmann machines correspond to random variables, such as are used in Bayesian networks
- Older variants of Boltzmann machines were defined using exclusively binary variables, but models with continuous and discrete variables are also possible
- They became popular prior to the impressive results of convolutional neural networks on the ImageNet challenge, but have since waned in popularity because they are more difficult to work with

Boltzmann machines

- To create a Boltzmann machine we partitioning variables into ones that are visible, using a *D*-dimensional binary vector **v**, and ones that are hidden, defined by a *K*-dimensional binary vector **h**
- A Boltzmann machine is a joint probability model of the form

$$p(\mathbf{v},\mathbf{h};\theta) = \frac{1}{Z(\theta)}\exp(-E(\mathbf{v},\mathbf{h};\theta)), \quad Z(\theta) = \sum_{\mathbf{v}}\sum_{\mathbf{h}}\exp(-E(\mathbf{v},\mathbf{h};\theta)),$$

$$E(\mathbf{v},\mathbf{h};\theta) = -\frac{1}{2}\mathbf{v}^T\mathbf{A}\mathbf{v} - \frac{1}{2}\mathbf{h}^T\mathbf{B}\mathbf{h} - \mathbf{v}^T\mathbf{W}\mathbf{h} - \mathbf{a}^{\mathrm{T}}\mathbf{v} - \mathbf{b}^{\mathrm{T}}\mathbf{h},$$

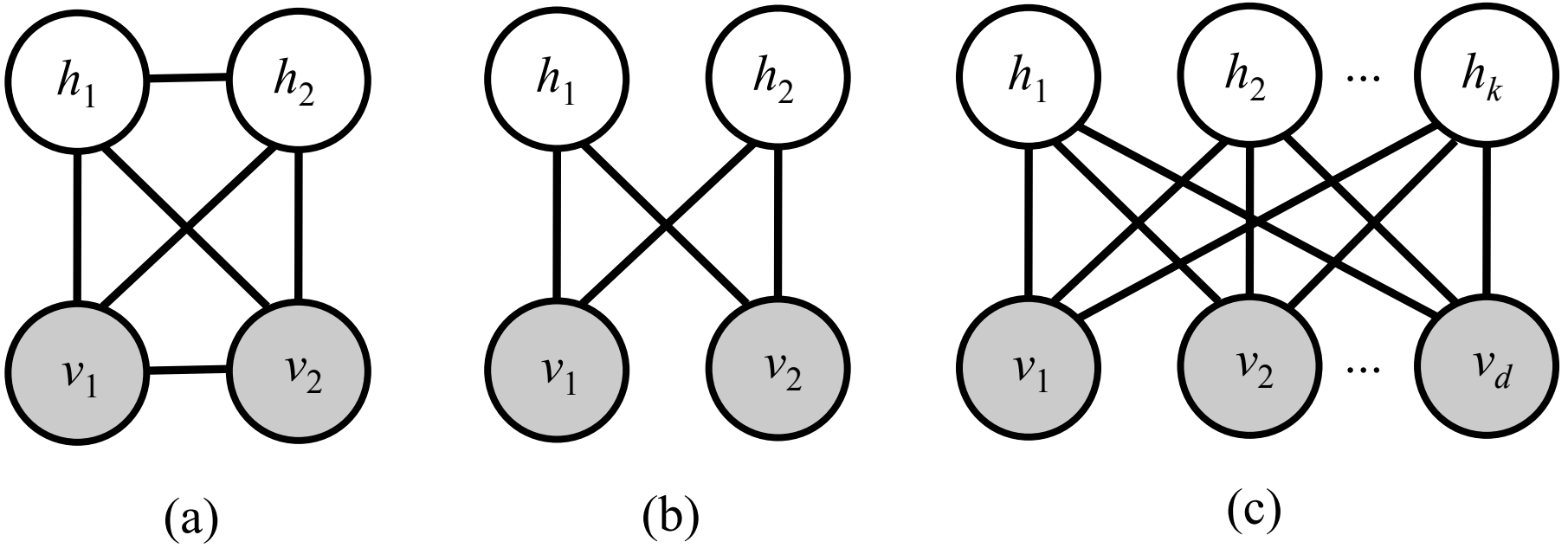
- where E(**v**,**h**;θ) is the energy function
- Z(θ) normalizes *E* so that it defines a valid joint probability
- matrices **A**, **B** and **W** encode the visible-to-visible, hidden-to-hidden and the visible-to-hidden variable interactions respectively
- vectors **a** and **b** encode the biases associated with each variable
- matrices **A** and **B** are symmetric, and their diagonal elements are 0

Boltzmann machines

(a) Boltzman Machines are binary Markov random field with pairwise connections between all variables

(b) Restricted Boltzmann machines (RBMs) do not have connections between the variables in a layer

(c) RBMs can be extended to many variables as shown



Key feature of Boltzmann machines

- A key feature of Boltzmann machines (and binary Markov random fields in general) is that the conditional distribution of one variable given the others is a sigmoid function whose argument is a weighted linear combination of the states of the other variables

$$p(h_j = 1 \mid \mathbf{v}, \mathbf{h}_{\neg j}; \theta) = \text{sigmoid}\left(\sum_{i=1}^{D} W_{ij} v_i + \sum_{k=1}^{K} B_{jk} h_k + b_j \right),$$

$$p(v_i = 1 \mid \mathbf{h}, \mathbf{v}_{\neg i}; \theta) = \text{sigmoid}\left(\sum_{j=1}^{K} W_{ij} h_j + \sum_{d=1}^{D} A_{id} v_d + c_i \right),$$

where the notation $\neg i$ indicates all elements with subscript other than *i*.

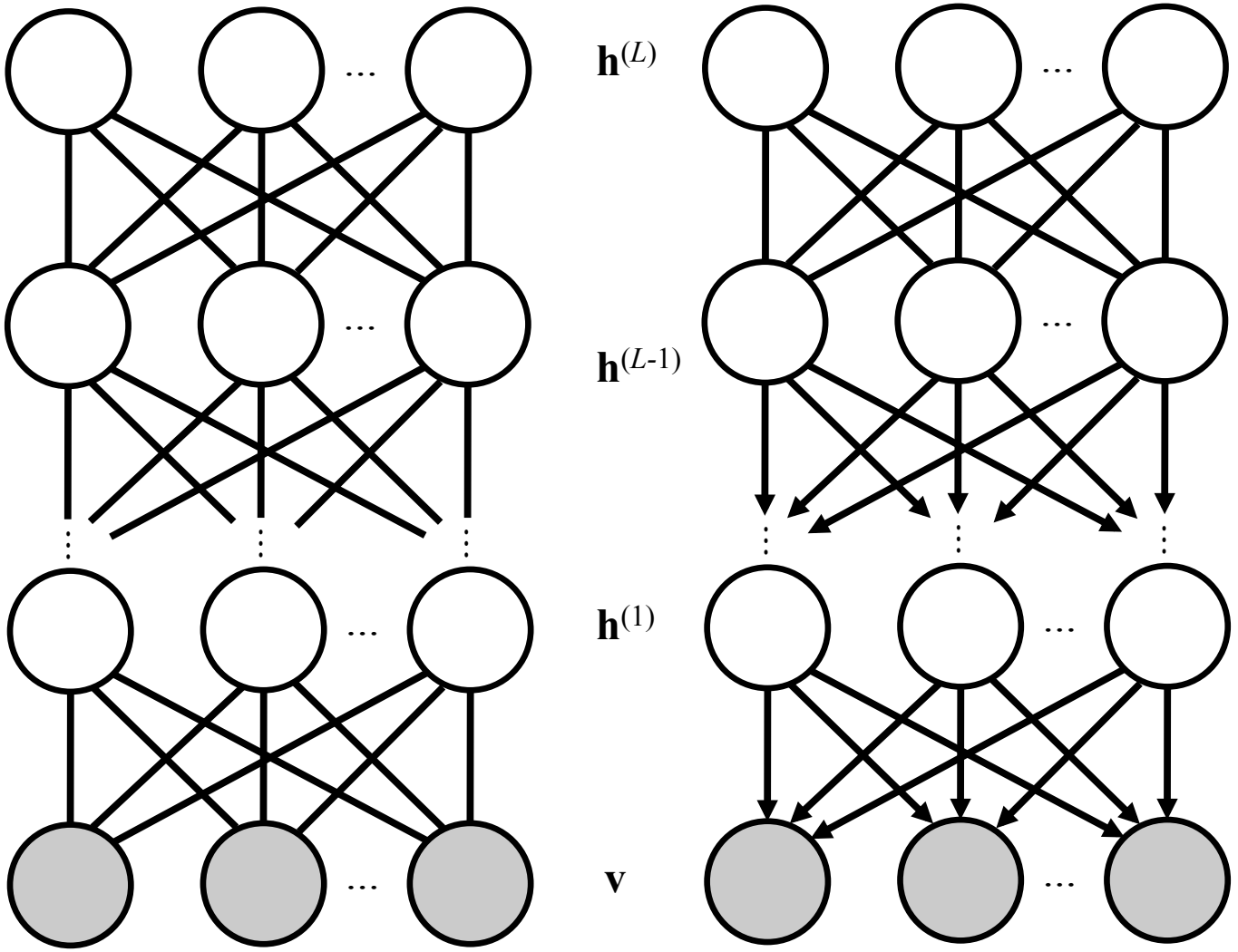
Contrastive divergence

- Running a Gibbs sampler for a Boltzmann machine often requires many iterations,
- A technique called “contrastive divergence” is a popular alternative that initializes the sampler to the observed data instead of randomly and performs a limited number of Gibbs updates.
- In an RBM a sample can be generated from the sigmoid distributions for all the hidden variables given the observed; then samples can be generated for the observed variables given the hidden variable sample
- This single step often works well in practice, although the process of alternating the sampling of hidden and visible units can be continued for multiple steps.

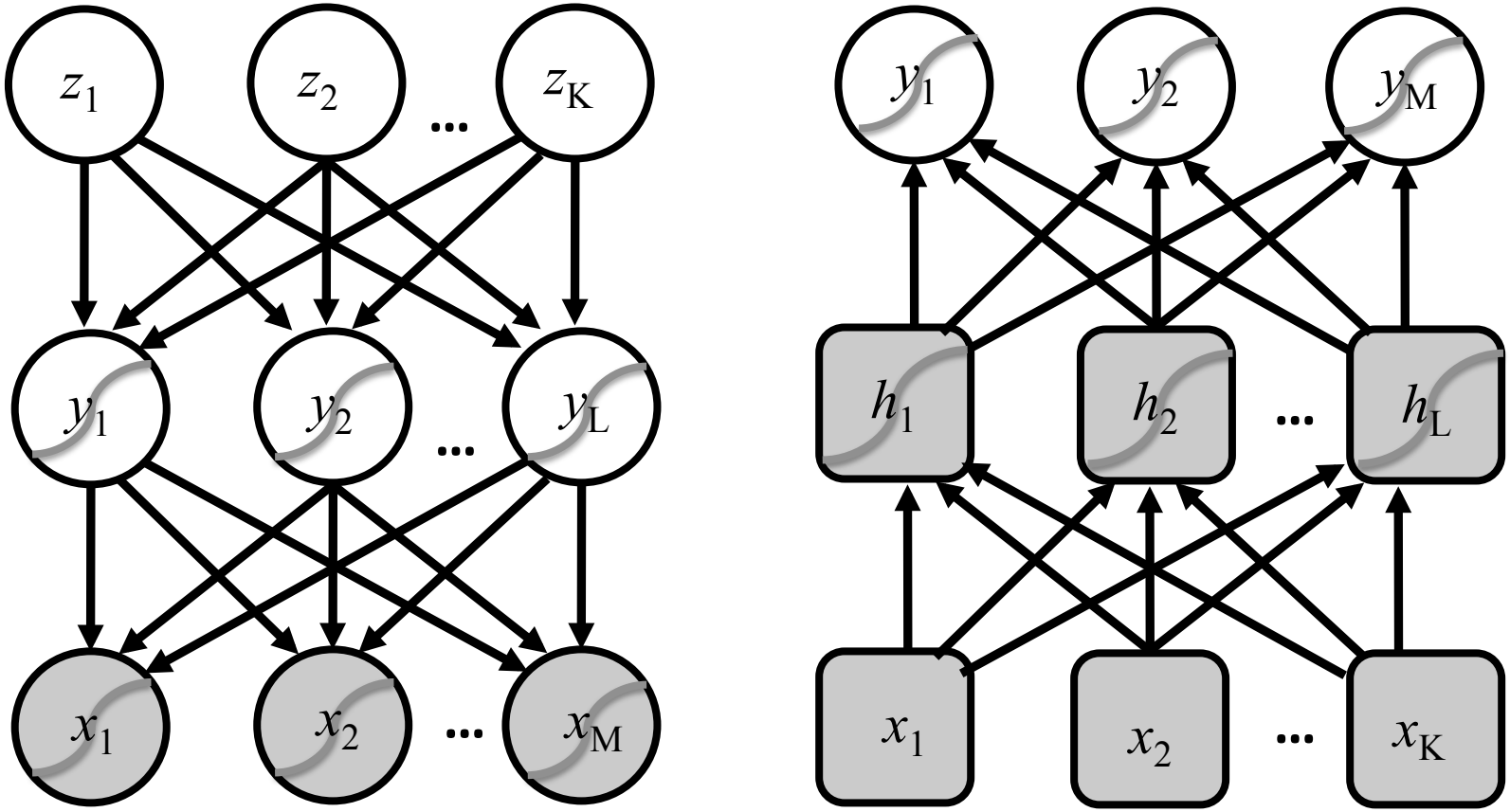
Deep RBMs and deep belief networks

- Deep Boltzmann machines involve coupling layers of random variables using restricted Boltzmann machine connectivity
- While any deep Bayesian network is technically a deep belief network, the term “deep belief network” has become strongly associated with a particular type of deep architecture that can be constructed by training restricted Boltzmann machines incrementally.
- The procedure is based on converting the lower part of a growing model into a Bayesian belief network, adding an RBM for the upper part of the model, then continuing the training, conversion and stacking process.

A deep RBM vs a deep belief network



A sigmoidal belief network vs a neural network



Bibliographic Notes & Further Reading

Stochastic methods – Boltzmann machines

- The history of Markov random fields has roots in statistical physics in the 1920s with so-called "Ising models" of ferromagnetism
- Our presentation of Boltzmann machines follows Hinton and Sejnowski (1983), but we use matrix-vector notation and our exposition more closely resembles formulations such as that of Salakhutdinov and Hinton (2009)
- Harmonium networks proposed in Smolensky (1986) are essentially equivalent to what are now commonly referred to as restricted Boltzmann machines
- Contrastive divergence was proposed by Hinton (2002)

Bibliographic Notes & Further Reading

Stochastic methods – Boltzmann machines

- The idea of using unsupervised pre-training to initialize deep networks using stacks of restricted Boltzmann machines was popularized by Hinton and Salakhutdinov (2006)
- Salakhutdinov and Hinton (2009) give further details on the use of deep Boltzmann machines and training procedures for deep belief networks, including other nuances for greedy training of deep restricted Boltzman machines
- Neal (1992) introduced sigmoidal belief networks
- Welling et al. (2004) showed how to extend Boltzmann machines to categorical and continuous variables using exponential-family models
- A greedy layer-wise training procedure for deep Boltzmann machines was proposed by Hinton and Salakhutdinov (2006) and refined by Murphy (2012)

Bibliographic Notes & Further Reading

Stochastic methods – Boltzmann machines

- Hybrid supervised and unsupervised learning procedures for restricted Boltzman machines were proposed by McCallum et al. (2005) and further explored by Larochelle and Bengio (2008).
- Vincent et al. (2010) proposed the autoencoder approach to unsupervised pre-training; they also explored various layer-wise stacking and training strategies and compared stacked restricted Boltzmann machines with stacked autoencoders.

Bibliographic Notes & Further Reading

Recurrent neural networks

- Graves et al. (2009) demonstrate how recurrent neural networks are particularly effective at handwriting recognition,
- Graves et al. (2013) apply recurrent neural networks to speech.
- The form of gradient clipping presented above was proposed by Pascanu et al. (2013).
- Hochreiter and Schmidhuber (1997) is the seminal work on the "Long Short-term Memory" architecture for recurrent neural networks;
  - our explanation follows Graves and Schmidhuber (2005)'s formulation.
- Greff et al. (2015)'s paper "LSTM: A search space odyssey" explored a wide variety of variants and finds that:
  a) none of them significantly outperformed the standard LSTM architecture; and
  b) forget gates and the output activation function were the most critical components. Forget gates were added by Gers et al. (2000).

Bibliographic Notes & Further Reading

Recurrent neural networks

- IRNNs were proposed by Le et al. (2015)
- Chung et al. (2014) proposed gated recurrent units
- Schuster and Paliwal (1997) proposed bidirectional recurrent neural networks
- Chen and Chaudhari (2004) used bi-directional networks for protein structure prediction; Graves et al. (2009) used them for handwriting recognition
- Cho et al. (2014) used encoder-decoder networks for machine translation, while Sutskever et al. (2014) proposed deep encoder-decoder networks and used them with massive quantities of data
- For further accounts of advances in deep learning and a more extensive history of the field, consult the reviews of LeCun et al. (2015), Bengio (2009), and Schmidhuber (2015)