Introduction to Deep Learning

Julien Mille

INSA Centre Val de Loire - Département GSI, 5A, option ACAD

Université de Tours - Master BDMA

Laboratoire d'Informatique Fondamentale
et Appliquée de Tours (LIFAT)

Outline

Introduction

What is Deep Learning ?

- A kind of statistical machine learning algorithms
- Good old Neural Networks, with more layers/modules
- Non-linear, hierarchical, abstract representations of data
- Flexible models with any input/output type and size
- Differentiable functional programming (automatic differentiation)

Introduction

- "Classical" learning systems

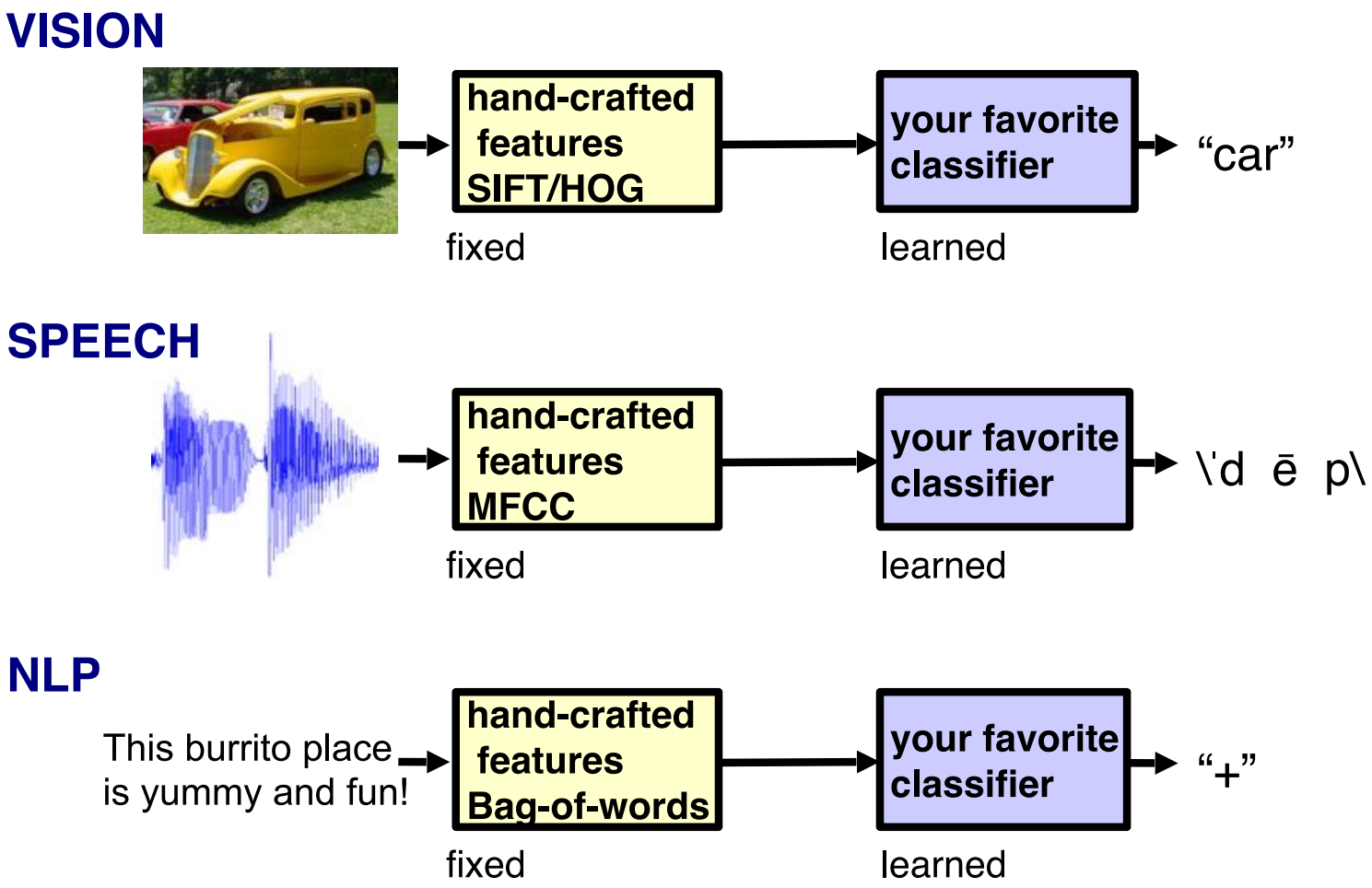
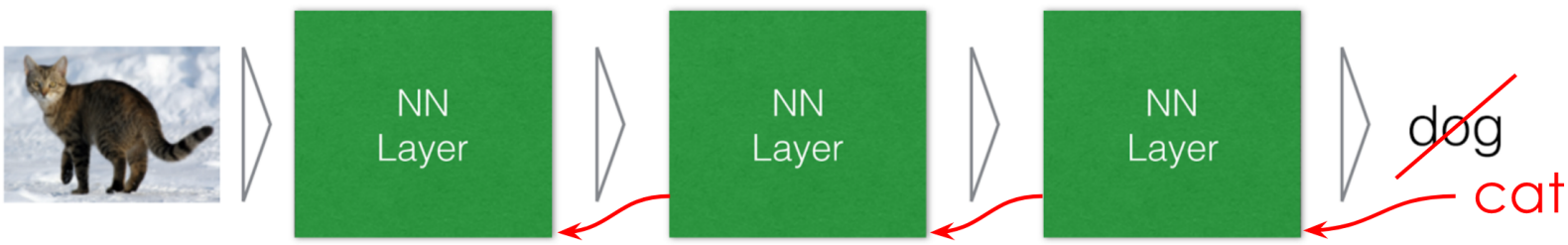


Figure by A. Erdem

Introduction

- "Classifical" learning system

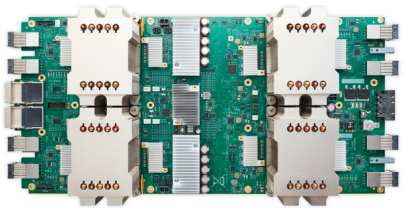
- Deep learning system



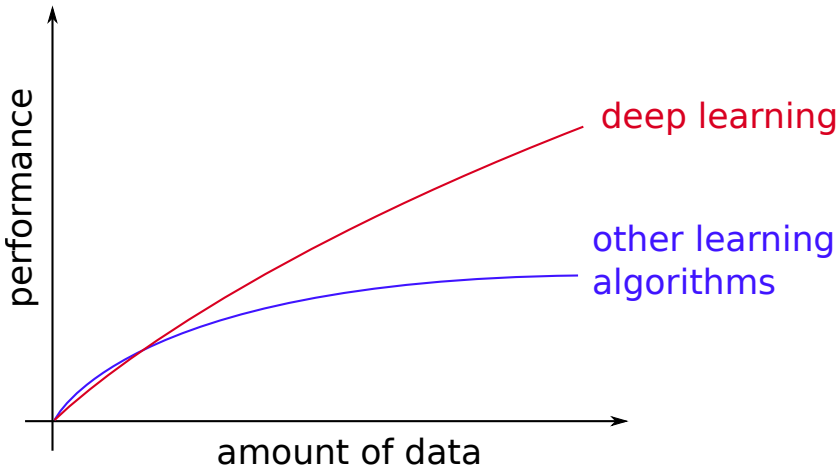
Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Introduction

Why Deep Learning now ?



Computing power (GPUs)



Labeled datasets


theano

Microsoft CNTK
PYTORCH
Caffe2
dmlc mxnet
gensim
spaCy

Open source frameworks

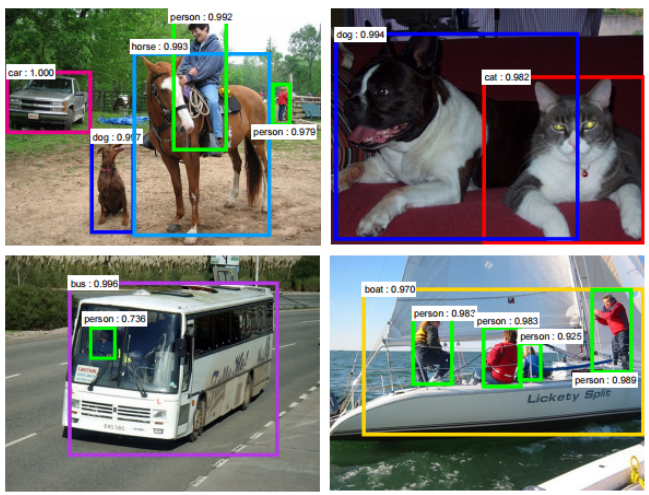
Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Introduction

- Deep Learning in Vision

[Krizhevsky 2012]

[Ciresan et al. 2013]



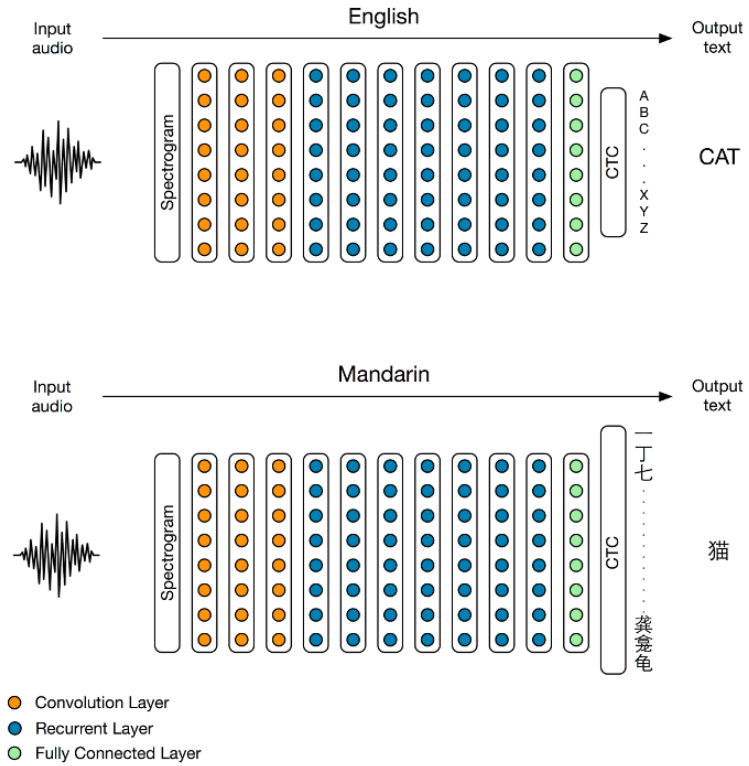
[Faster R-CNN - Ren 2015]

[NVIDIA dev blog]

Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Introduction

- Deep Learning in speech processing

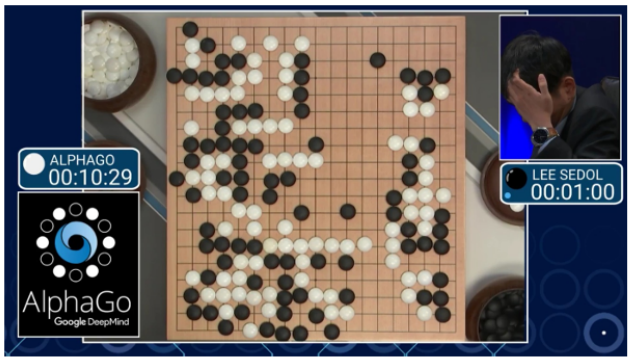


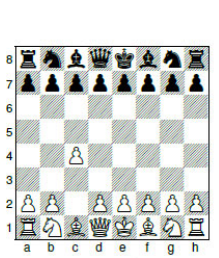
[Baidu 2014]

Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Introduction

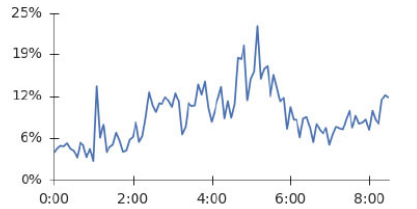
- Deep Learning in games





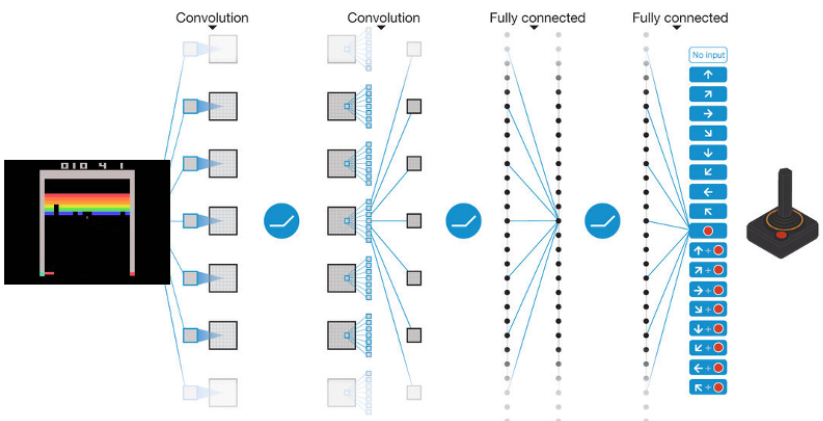
w 20/30/0, b 8/40/2

A10: English Opening

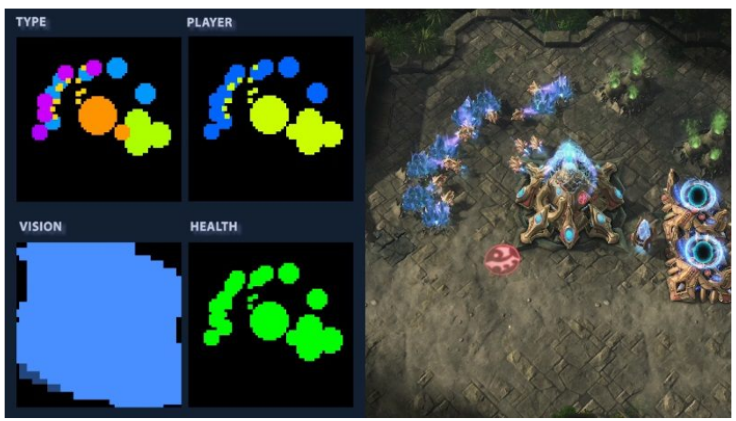


1...e5 g3 d5 cxd5 ♘f6 ♗g2 ♘xd5 ♘f3

[Deepmind AlphaGo / Zero 2017]



[Atari Games - DeepMind 2016]



[Starcraft 2 for AI research]

Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Introduction

- Learning a hierarchy of increasingly abstract representations

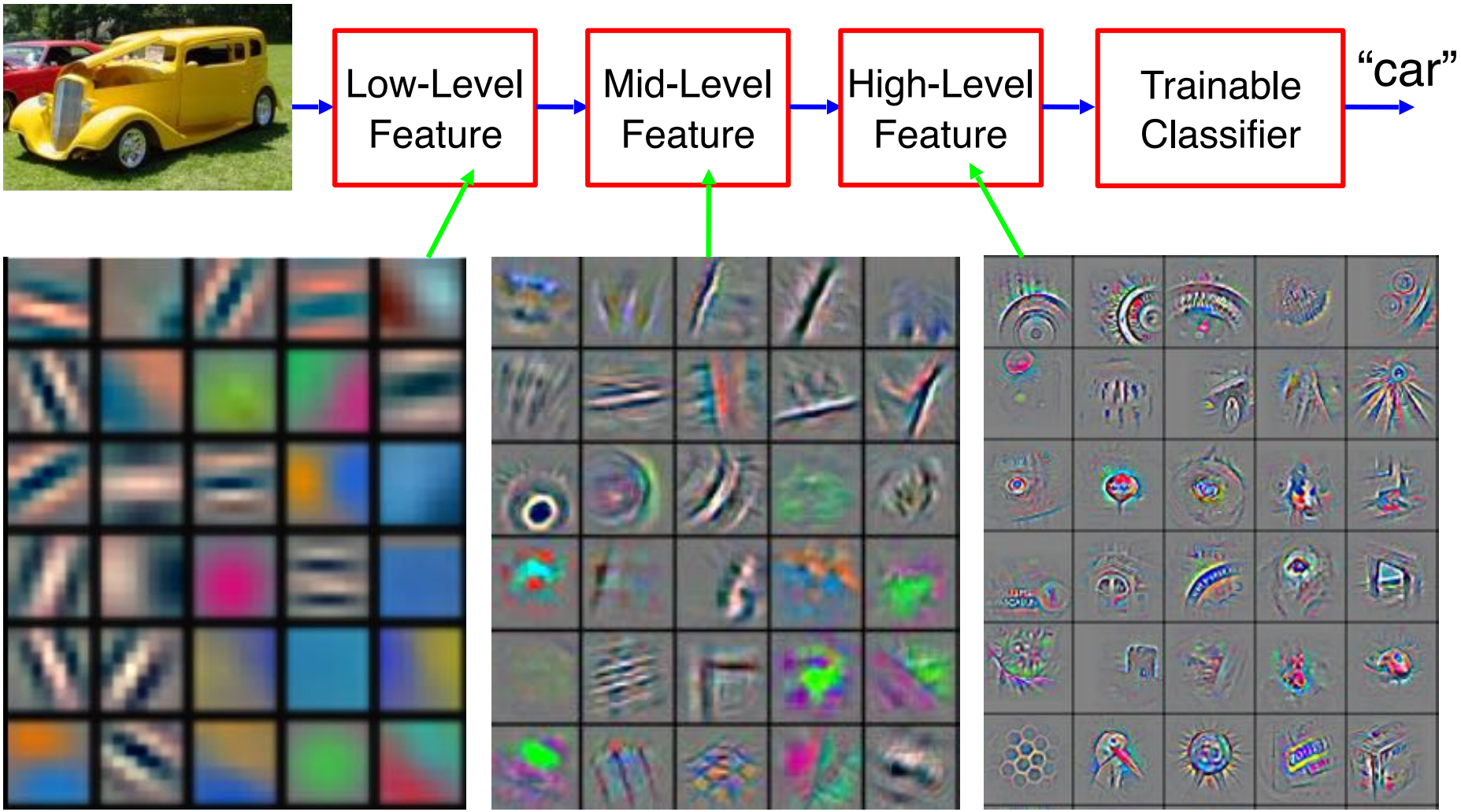


Figure by Y.Lecun and M.A. Ranzato

Introduction

Deep Learning $\rightarrow$ End-to-End learning

- A hierarchy of trainable feature transforms.
- Each module transforms its input representation into a higher-level one.
- Low-level features are shared among categories.
- As the level increases, features are increasingly global and invariant.

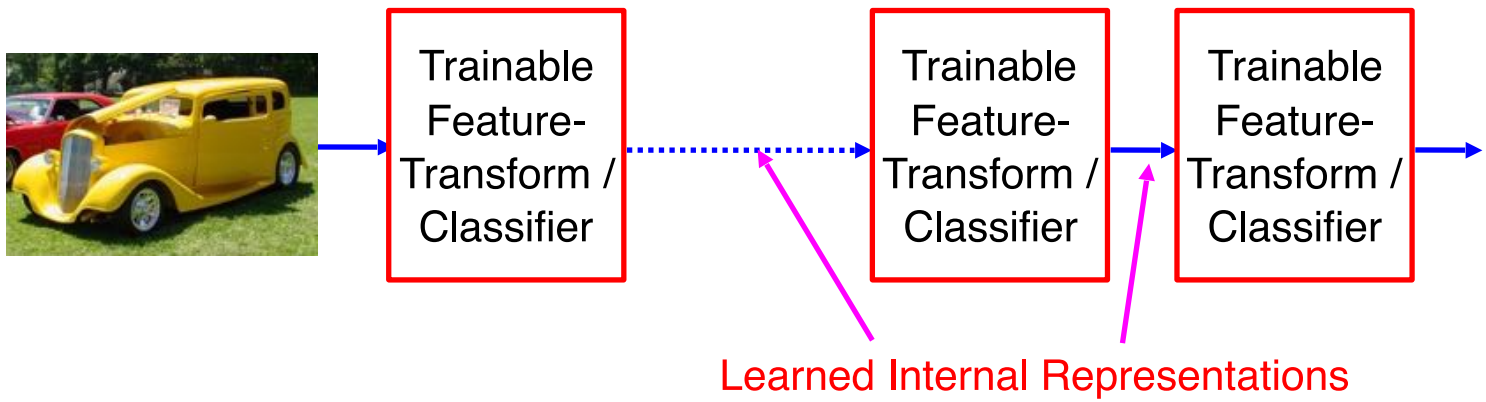
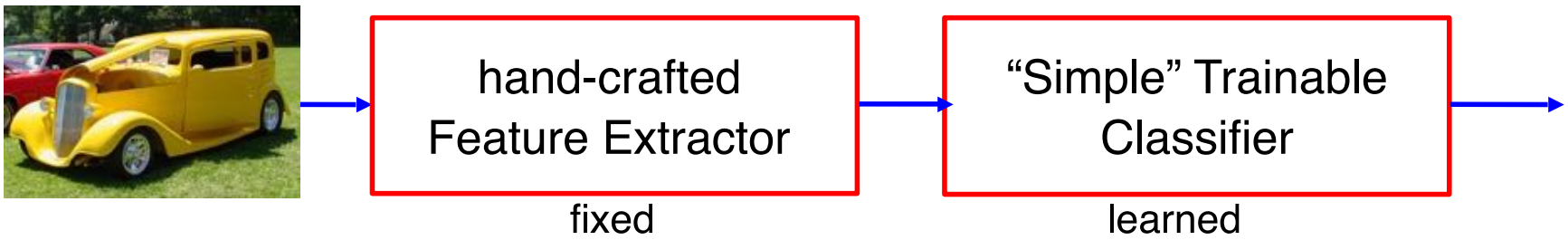


Figure by Y.Lecun and M.A. Ranzato

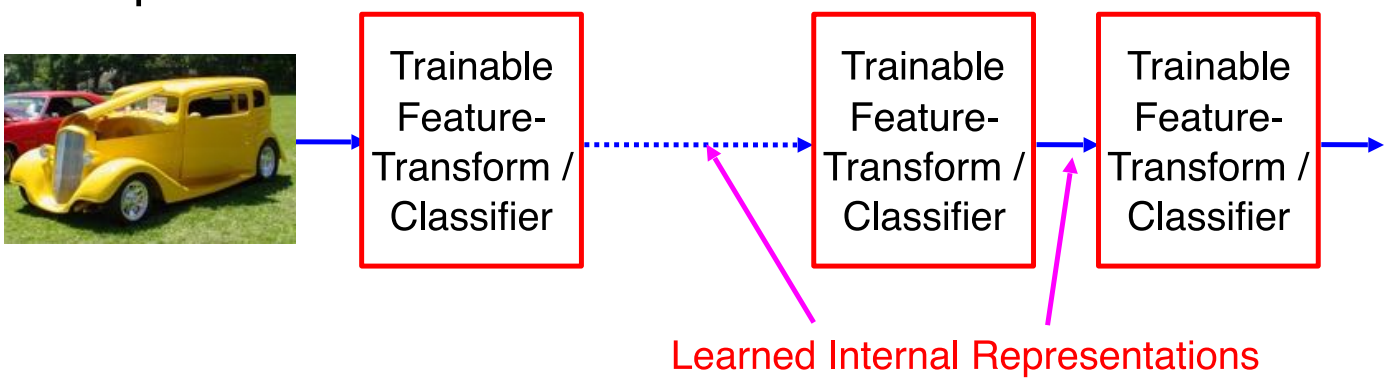
Introduction

"Shallow" vs Deep Learning

- "Shallow" models



- "Deep" models



Figures by Y.Lecun and M.A. Ranzato

Introduction

- Real data examples for a given task are usually not spreaded everywhere in **input space**, but rather clustered on a low-dimension "manifold", also referred to as **latent space**.
- Example : images of faces, of size $200 \times 200 \rightarrow$ each sample in the input space is a vector in $\mathbb{R}^{40000}$
- But the number of features (degrees of freedom) leading to plausible images of faces is much smaller : orientation, lighting, positions of face elements, shapes, skin color, hair type, *etc*.

Figures by G.Perarneau *et al*

Introduction

Why features should be learnt ?

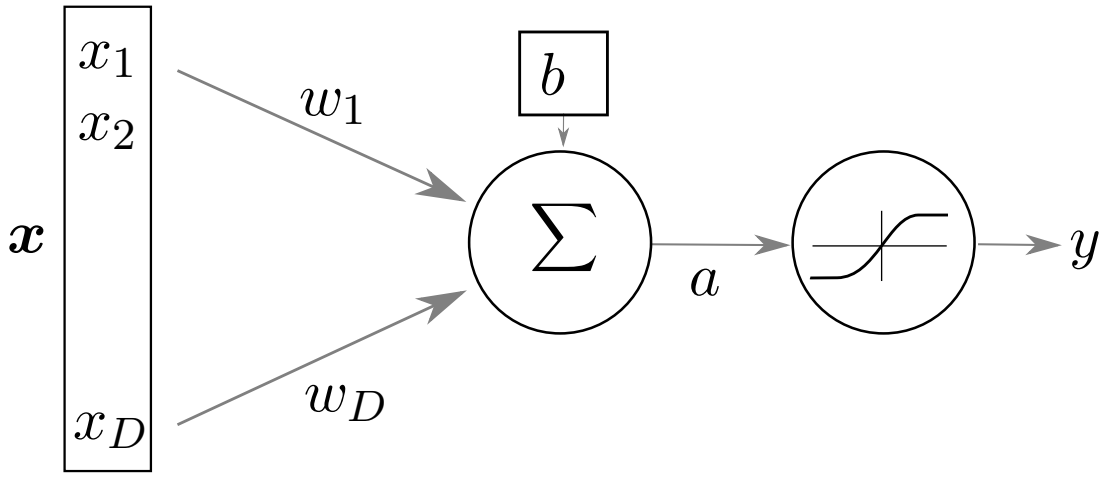
- There is a lot of redundancy in the input space.
- A simple fact : in images, neighboring pixels very often look the same.
- There is much less redundancy in the latent space.
- Learning features lets the system decide itself how to deal with this redundancy.

Outline

Outline

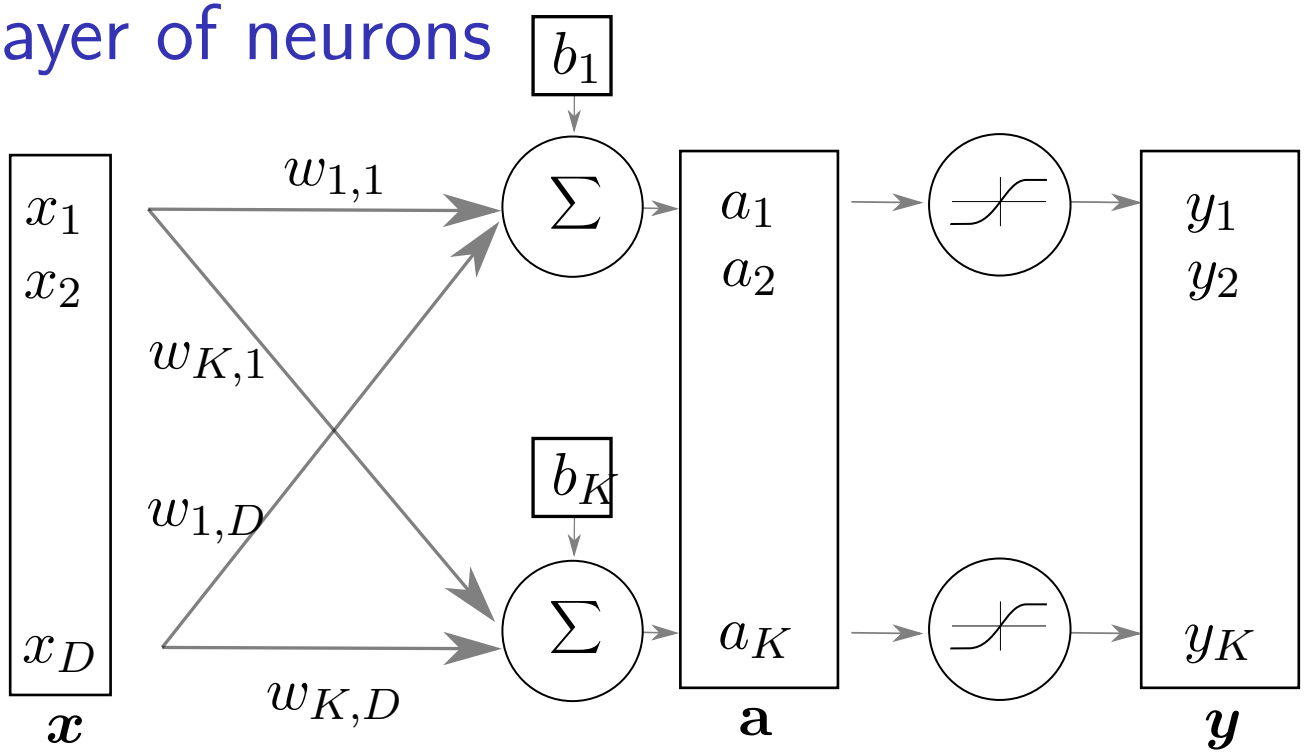
A single neuron

- A single neuron



- Weighted sum of inputs plus bias : $a = \mathbf{w}^\mathsf{T}\boldsymbol{x} + b = b + \sum_{j=1}^{D} w_j x_j$
- Output of neuron : activation function applied to a :
  $y = f(a) = f(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b)$

A single layer of neurons



- D inputs, K neurons (inputs, outputs and biases are column vectors)
- $\mathbf{W}$ = weight matrix of size $K \times D$, $\mathbf{b}$ = bias vector of size K
- $\mathbf{a} = \mathbf{W}\boldsymbol{x} + \mathbf{b}$
- Output of layer : activation function f applied element-wise to $\mathbf{a}$:

$$\boldsymbol{y} = \mathbf{f}(\mathbf{a}) = [f(a_1) \ \cdots \ f(a_K)]^\mathsf{T}$$

- Each input is connected to each output : known as **dense** or **fully-connected** layer

Common activation functions

Element-wise activation functions :

- Identity : $f(a) = a$
- Sigmoid :
  - Logistic function : $f(a) = \dfrac{1}{1 + e^{-a}}$ (often referred to as *the* sigmoid)
  - Hyperbolic tangent : $f(a) = \tanh(a)$
  - Arctangent : $f(a) = \arctan(a)$
- ReLU (Rectified Linear Unit) : $f(a) = \max(0, a)$

The softmax function

- A.k.a normalized exponential
- Outputs a vector whose components sum to 1 :

$$\text{softmax}(\mathbf{a}) = \frac{1}{\sum_{j=1}^{K} e^{a_j}} \begin{bmatrix} e^{a_1} \\ \vdots \\ e^{a_K} \end{bmatrix}$$

- Usually, used for 1-of-K representation (*one-hot* encoding) in the last layer of neural-network based classifier.

The 3 sets of supervised learning

Training set

- For fitting the **parameters** of the model (for neural networks : weights and biases)
- Should not overfit !

Validation set (not always used)

- For adjusting the **hyperparameters** of the model (for neural networks, this can be the size of hidden layers)
- Can be used for regularization

Test set

- Used to provide an unbiased evaluation of a final model which was fit on the training dataset
- Follows the same probability distribution than training set
- Evaluate **generalization** ability of model (handling samples which were never seen during training)

Classification and loss

- We have a **training set** $(\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1...N}$. Each $\boldsymbol{x}_i$ is a feature vector of size D. Each $\boldsymbol{y}_i$ is the desired output (label) of size K.
- For a classification task, K is the number of classes, and $\boldsymbol{y}_i$ is a 1-of-K encoding of the label of the i^{th} sample.
- The **loss function** measures, for a given sample, the discrepancy between the output of the model (the neural network) and the label. Common loss functions ($\mathbf{p}, \mathbf{q} \in \mathbb{R}^K$) :
  - Mean squared error : $\mathcal{L}(\mathbf{p}, \mathbf{q}) = \frac{1}{K} \|\mathbf{p} - \mathbf{q}\|^2$
  - Mean absolute error : $\mathcal{L}(\mathbf{p}, \mathbf{q}) = \frac{1}{K} \sum_{j=1}^{K} |p_j - q_j|$
  - Cross-entropy (negative log-likelihood) loss (for p_j and q_j between 0 and 1) : $\mathcal{L}(\mathbf{p}, \mathbf{q}) = -\sum_{j=1}^{K} q_j \log p_j$

Cost minimization

- The parameters of the model $\mathcal{M}$ are represented by vector θ.
- The output of the model for a given sample is denoted by $\mathcal{M}(\boldsymbol{x}_i; \theta)$.
- The total **cost function** ($=$ **error**) is the sum of losses on the entire training set.
- Training the model $=$ iteratively modify parameters θ in order to minimize the cost function :

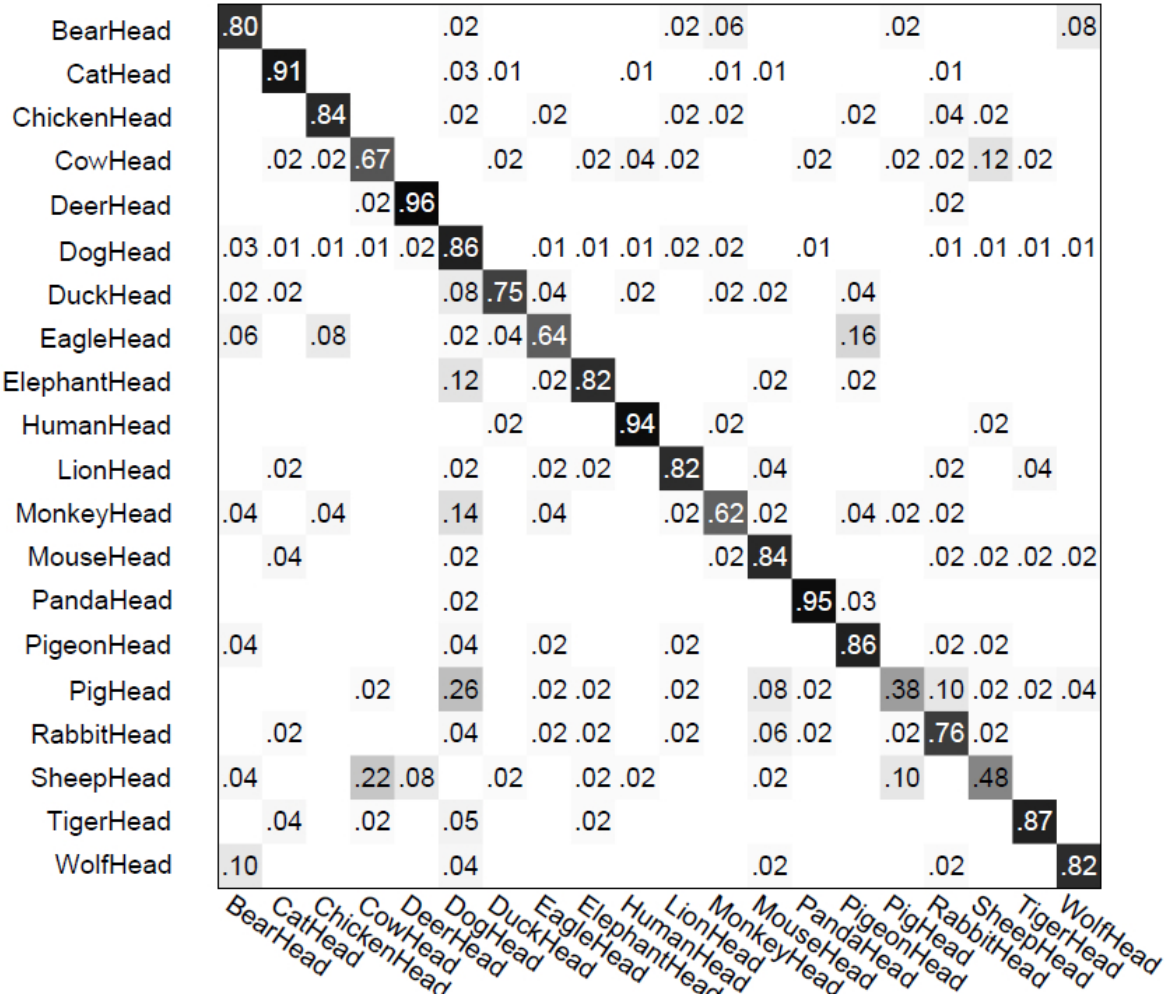
$$\mathcal{C}(\theta) = \sum_{i=1}^{N} \mathcal{L}(\mathcal{M}(\boldsymbol{x}_i; \theta), \boldsymbol{y}_i)$$

- The cost function is **differentiable** with respect to each parameter. When the learning procedure has converged to a **local minimum**, we should have, ideally :

$$\frac{\mathrm{d}\mathcal{C}}{\mathrm{d}\theta} = \mathbf{0}$$

Confusion matrix

- At test time, one wishes the cost function to be small on the test set
- In addition, one often wishes to know in which class the errors are
  $\rightarrow$ generate a confusion matrix (a row $=$ an actual class, a column $=$ a predicted class)



Backpropagation

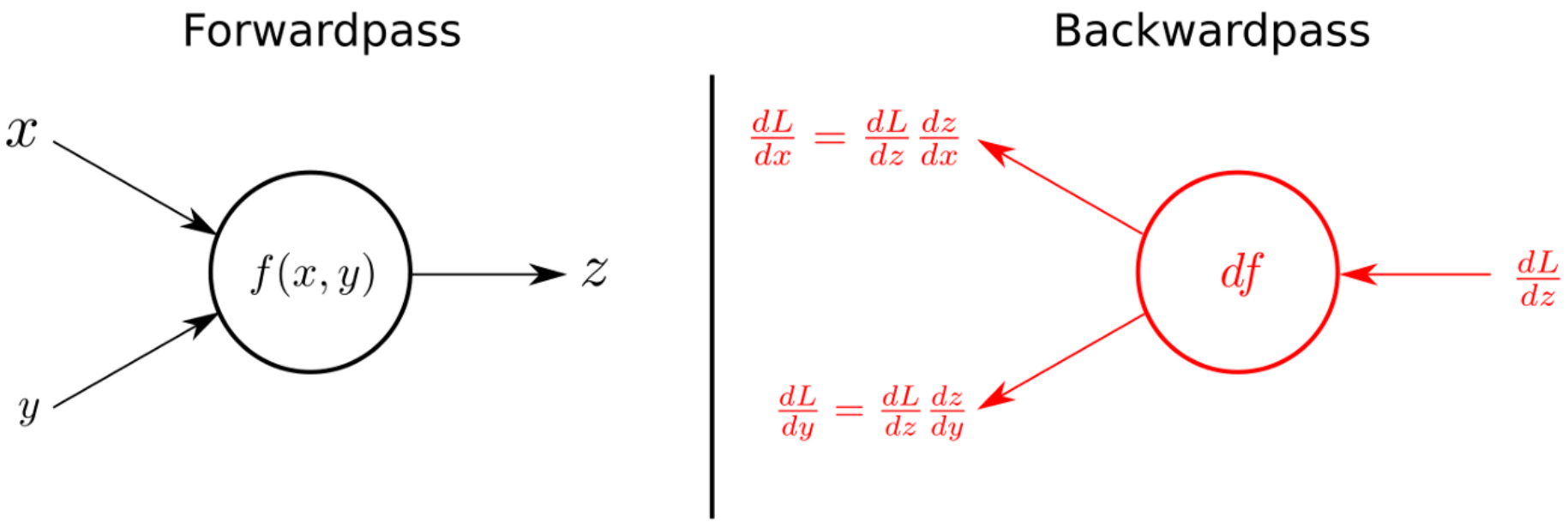
- Minimization of the cost function by **gradient descent** : at a given iteration t, for each parameter θ_j in θ,

$$\theta_j^{t+1} \leftarrow \theta_j^t - \alpha \frac{\partial \mathcal{C}}{\partial \theta_j},$$

  where α is the **learning rate**.
- **Backpropagation** : parameters (weights and biases) are updated by descending order of layer.
- Let $w_{i,j,l}$ be the weight corresponding to input j and output i in layer l. Derivative $\dfrac{\partial \mathcal{C}}{\partial w_{i,j,l}}$ depends on $\dfrac{\partial \mathcal{C}}{\partial w_{\cdot,\cdot,l+1}} \rightarrow$ chain rule of derivation.
- Forward pass : feed a sample at the input of the network, compute activation and output for each neuron in ascending order of layer.
- Backward pass : evaluate derivative of cost function for parameters and update these parameters, for each neuron in descending order of layer.

Backpropagation

- Forward pass : feed a sample at the input of the network, compute activation and output for each neuron in ascending order of layer.
- Backward pass : evaluate derivative of cost function for parameters and update these parameters, for each neuron in descending order of layer.



Batch vs stochastic gradient descent

- Batch gradient descent = average $\frac{\partial \mathcal{L}(\mathcal{M}(\boldsymbol{x}_i;\theta),\boldsymbol{y}_i)}{\partial \theta_j}$ over all samples $\boldsymbol{x}_i$ to update each $\theta_j \rightarrow$ Computationally expensive !

$$\theta_j^{t+1} \leftarrow \theta_j^t - \frac{\alpha}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}(\mathcal{M}(\boldsymbol{x}_i;\theta),\boldsymbol{y}_i)}{\partial \theta_j}.$$

- Stochastic gradient descent = randomly shuffle training samples, pick a sample $\boldsymbol{x}_i$ and update each θ_j :

$$\theta_j^{t+1} \leftarrow \theta_j^t - \alpha\frac{\partial \mathcal{L}(\mathcal{M}(\boldsymbol{x}_i;\theta),\boldsymbol{y}_i)}{\partial \theta_j}.$$

- Stochastic minibatch gradient descent = randomly shuffle samples, pick a small subset of samples and update each θ_j with derivatives averaged over this small subset.

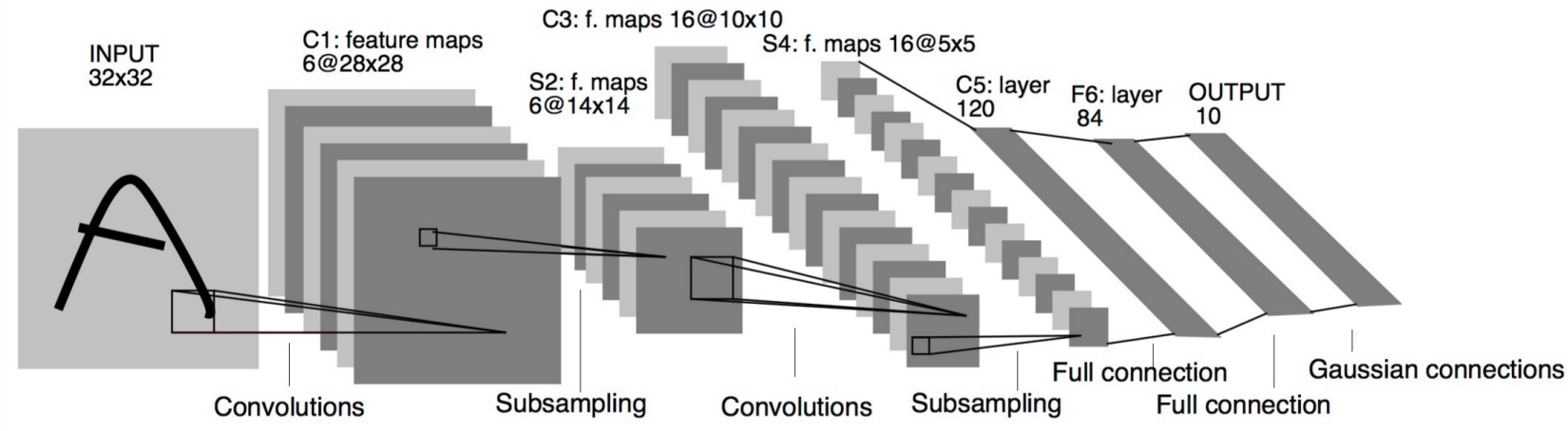
Training a model

- The cost function is **non-convex** $\rightarrow$ has many local minima
- Gradient descent converges to a **local minimum** (we hope that it is a good one !)
- Dependence on initialization (typically, weights are randomly drawn from a zero-mean normal distribution. Biases are set to 0)
- Different sets of parameters can lead to the same classification
- An **epoch** is **one** pass of gradient descent (batch, stochastic or minibatch) over the **whole** training set
- Training usually needs a large number of epochs

Outline

ConvNets : introduction

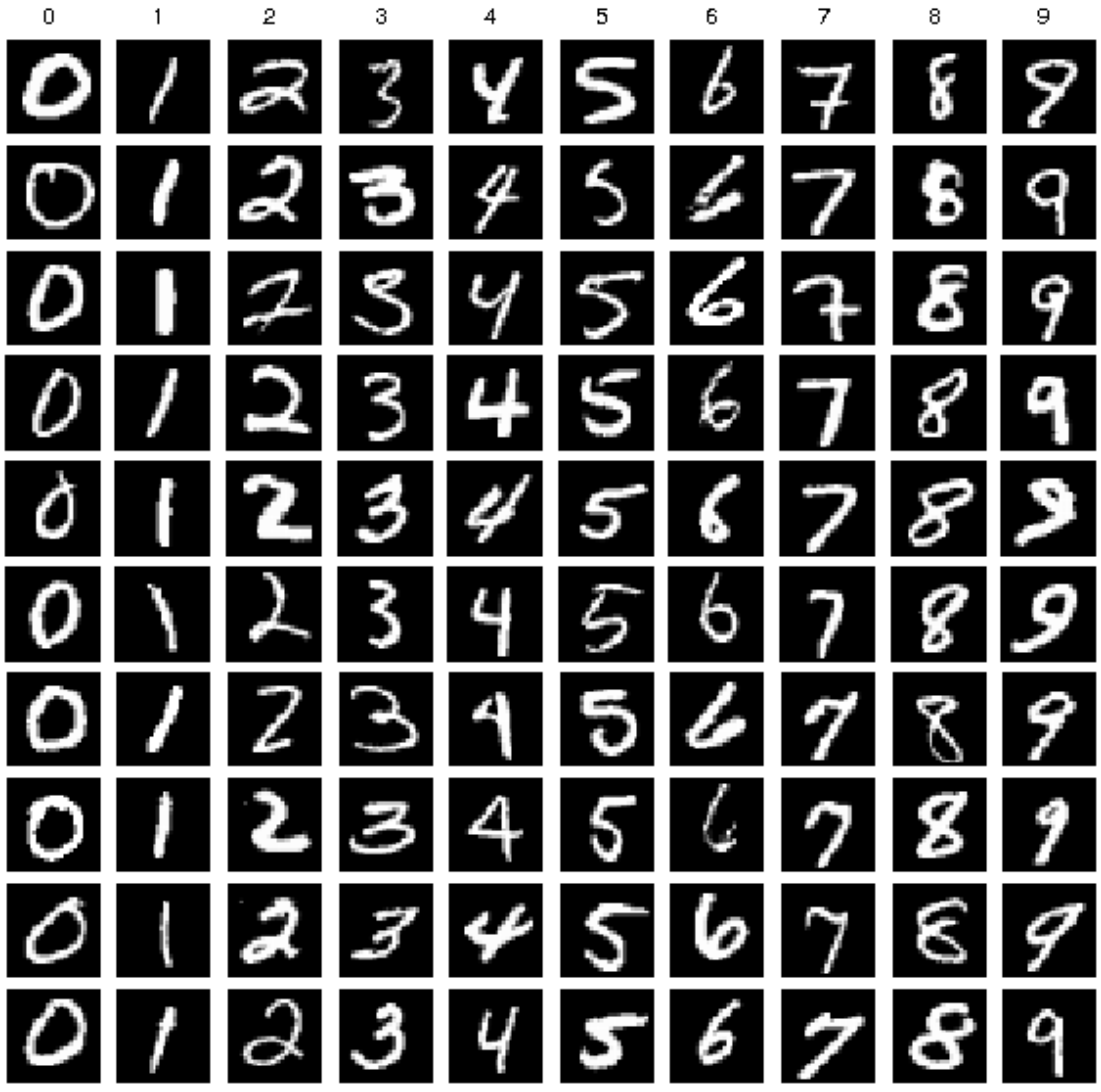
- Introduced in late 1990s by Yann Lecun *et al* for digit recognition



The LeNet5 network

ConvNets : introduction

- Introduced in late 1990s by Yann Lecun *et al* for digit recognition



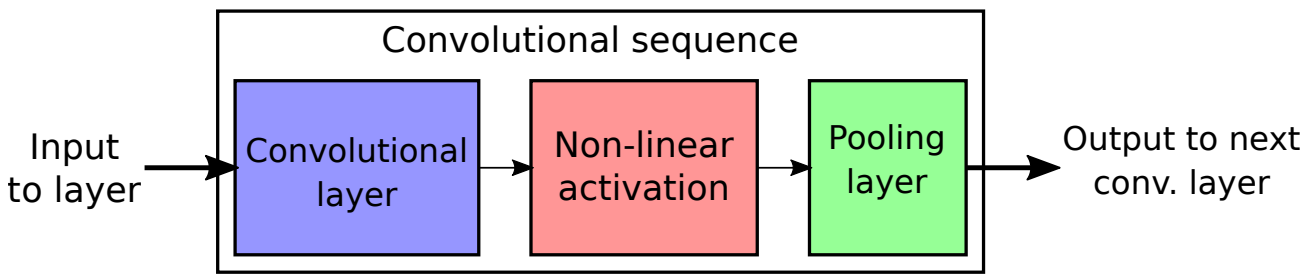
The MNIST dataset

Why ConvNets ?

- For a fully connected layer of input size N and output size M, the number of parameters to learn is $N(M+1)$.
- If a full image was flattened and fed into a fully-connected layer of neurons...
  - For a small image size, say 200×200, and 50 neurons in the layer, $200 \times 200 \times 51$ parameters to learn, for a single layer $\rightarrow$too many parameters.
  - Spatial layout would be destroyed.
- Conversely, ConvNets gradually decrease image size, and gradually increase feature vector size $\rightarrow$ spatial layout is progressively encoded into the successive layers.

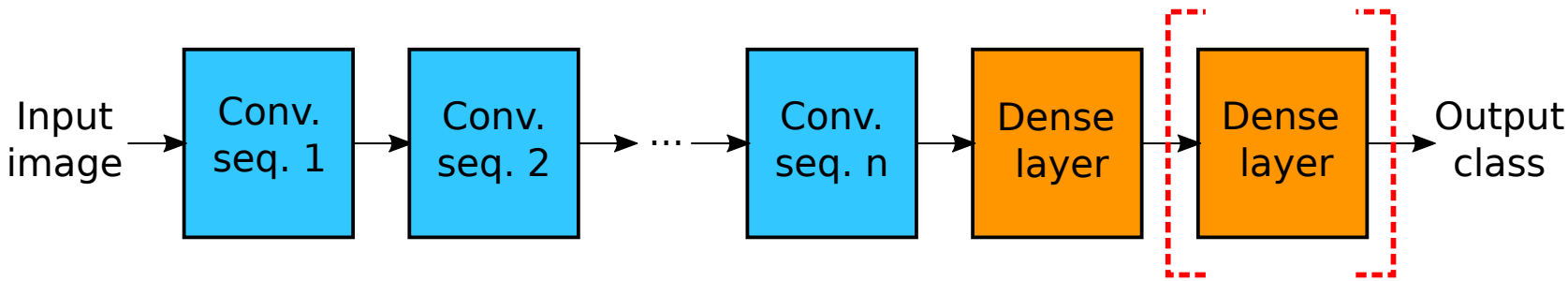
A convolutional sequence

- Input = feature maps (or input image) of size $W \times H \times D$.
- Output = features maps of size $W' \times H' \times D'$ (usually, $W' \leq W$ and $H' \leq H$ and $D' \geq D$).
- A convolutional sequence is made up of three layers :
  - A convolutional layer.
  - A non-linear activation layer (sigmoid, tanh, arctan, ReLU, ...).
  - A pooling layer (average, max, ...).
- Can handle variable-sized input



A full ConvNet

- Usually, one or two fully-connected (dense) layer(s) after a sequence of convolutional groups.
- If the goal is classification, the softmax function is chosen as activation in the last layer : desired outputs are 1-of-K encoding of class labels (outputs sum to 1)



Convolution (reminder ?)

- Mathematically speaking, in a continuous space setting, the convolution product between two functions $f : \mathcal{D} \mapsto \mathbb{R}$ and $g : \mathcal{D} \mapsto \mathbb{R}$, is another function defined by

$$(f * g)(\boldsymbol{p}) = \int_{\mathcal{D}} f(\boldsymbol{p} - \boldsymbol{y}) g(\boldsymbol{y}) \mathrm{d}\boldsymbol{y}$$

- Properties : bilinear, associative and commutative
- In a discrete 2D setting, the convolution between an image f and a filter (or mask) g (of size $K \times L$) outputs a new image h, such that

$$h[x, y] = \sum_{i=0}^{K-1} \sum_{j=0}^{L-1} f\left[x - i + \frac{K}{2}, y - j + \frac{L}{2}\right] g[i, j]$$

- Filter is usually centered $\rightarrow$ K and L are odd

Cross-correlation

- What is actually done in a convolutional layer is known as **cross-correlation** = convolution without flipping the mask :

$$h[x,y] = \sum_{i=0}^{K-1} \sum_{j=0}^{L-1} f\left[x+i-\frac{K}{2}, y+j-\frac{L}{2}\right] g[i,j]$$

- Each pixel value in the output image is a weighted sum of neighboring pixel values in the input image
- Numerical example (3×3 mask) :

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

Input image

0	1	2
2	2	0
0	1	2

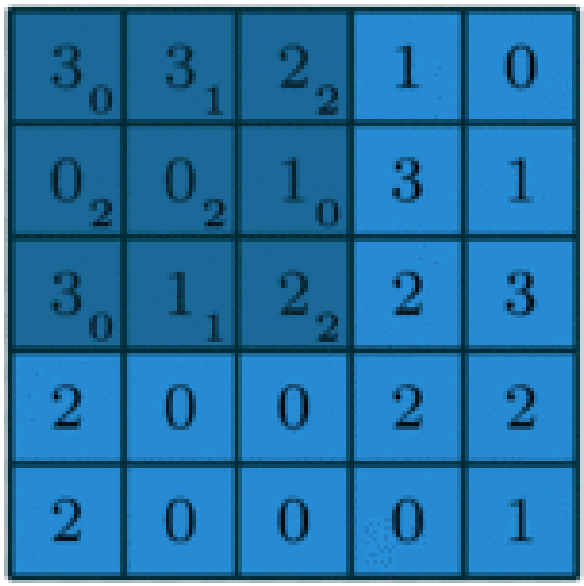
Mask

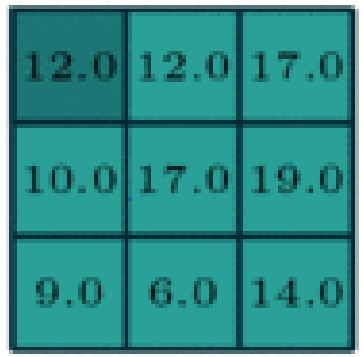
12	12	17
10	17	19
9	6	14

Output image

Convolution

- Filter is slided over the image : the same weights are applied on overlapping areas
- Numerical example :



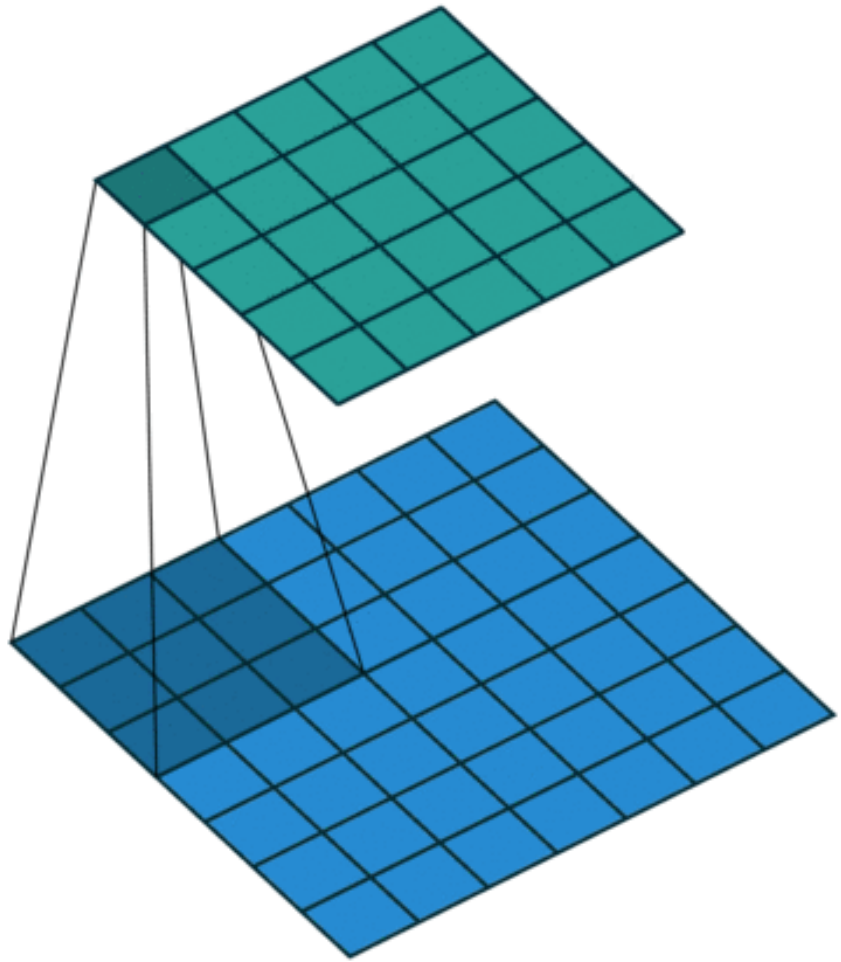


- Input image size $W \times H$, mask size=$K \times L$. If no padding is applied (and stride=1), output image size=$(W - K + 1) \times (H - L + 1)$

Figures by V. Dumoulin, `https://github.com/vdumoulin/conv_arithmetic`

Padding and stride

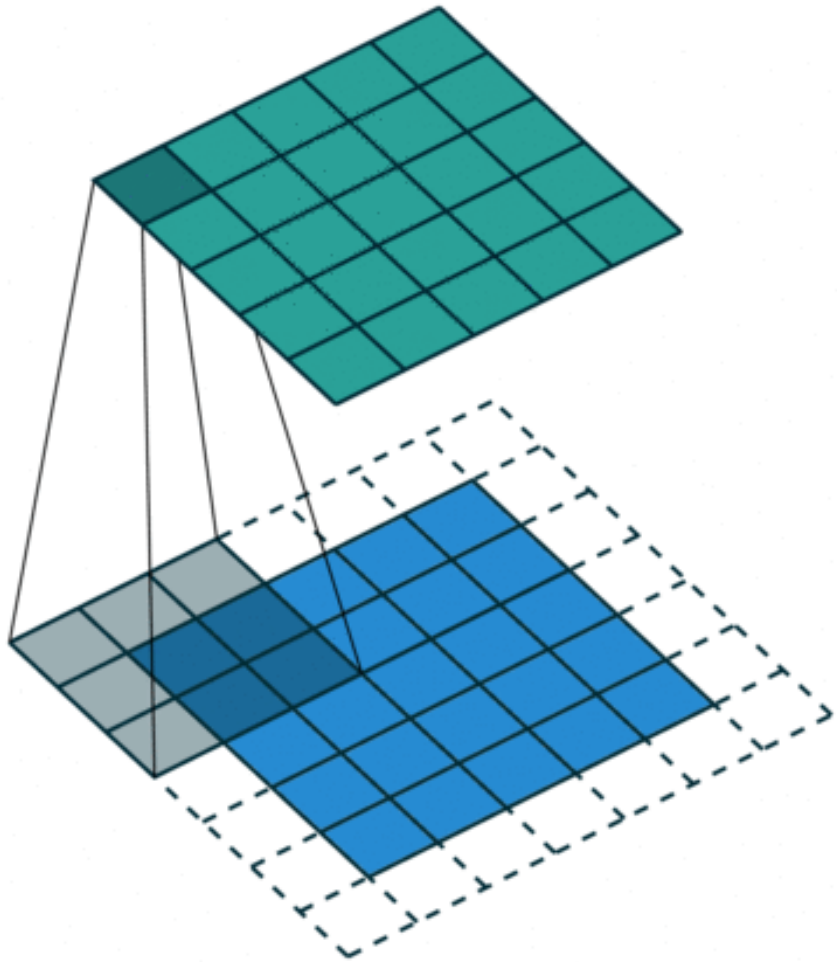
- Filter size$=3 \times 3$, stride$=1$, no padding



Figures by V. Dumoulin, `https://github.com/vdumoulin/conv_arithmetic`

Padding and stride

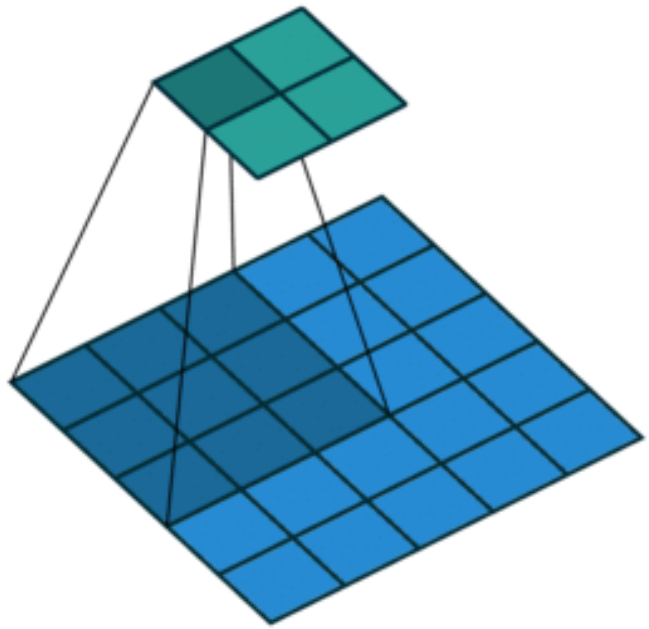
- Filter size=3×3, stride=1, horizontal padding=1, vertical padding=1



Figures by V. Dumoulin, `https://github.com/vdumoulin/conv_arithmetic`

Padding and stride

- Filter size=3×3, stride=2, no padding



Figures by V. Dumoulin, `https://github.com/vdumoulin/conv_arithmetic`

Convolutional sequence

Convolutional layer

- Filter weights and biases are learned !
- Filter size is much smaller than image size. Typically, 3×3, 5×5, ...
- Number of parameters independent from the width and height of the input feature maps.
- In practice, input feature maps have several features per pixel (depth is not necessarily 1) $\rightarrow$ 3D convolution.
- Example : filter size=$k \times k$, nb input features (depth)=A, nb output features=B, total number of parameters of layer is $(k \times k \times A + 1) \times B$ ($+1$ comes from the bias) $\rightarrow$ several orders of magnitude smaller than a fully connected layer.

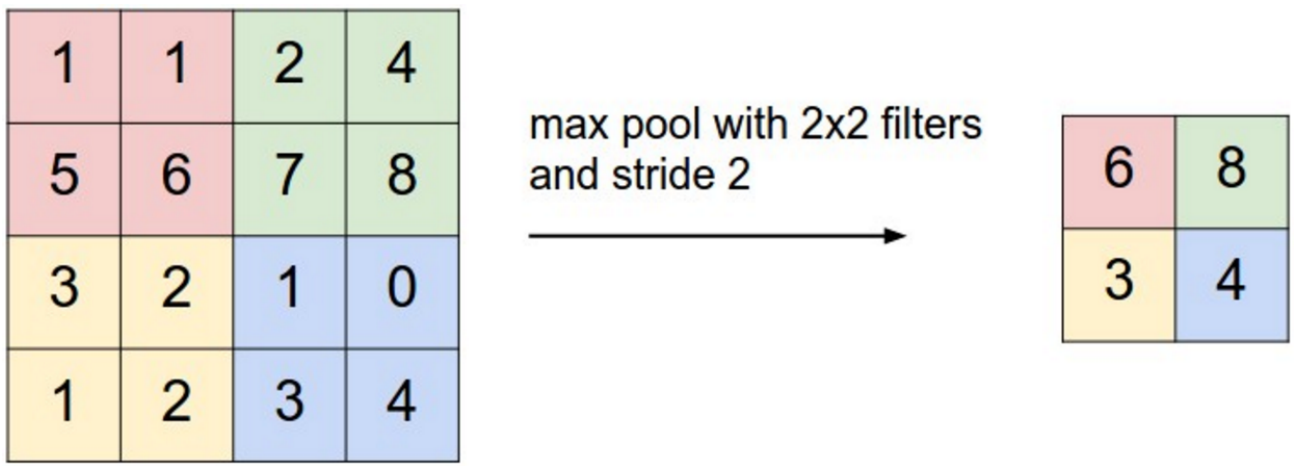
Non-linear activation layer

- Very often, ReLU$(x) = \max(0, x)$ is used
- Activation function is applied elementwise : for each output feature of each pixel
- No parameter to learn here

Convolutional sequence

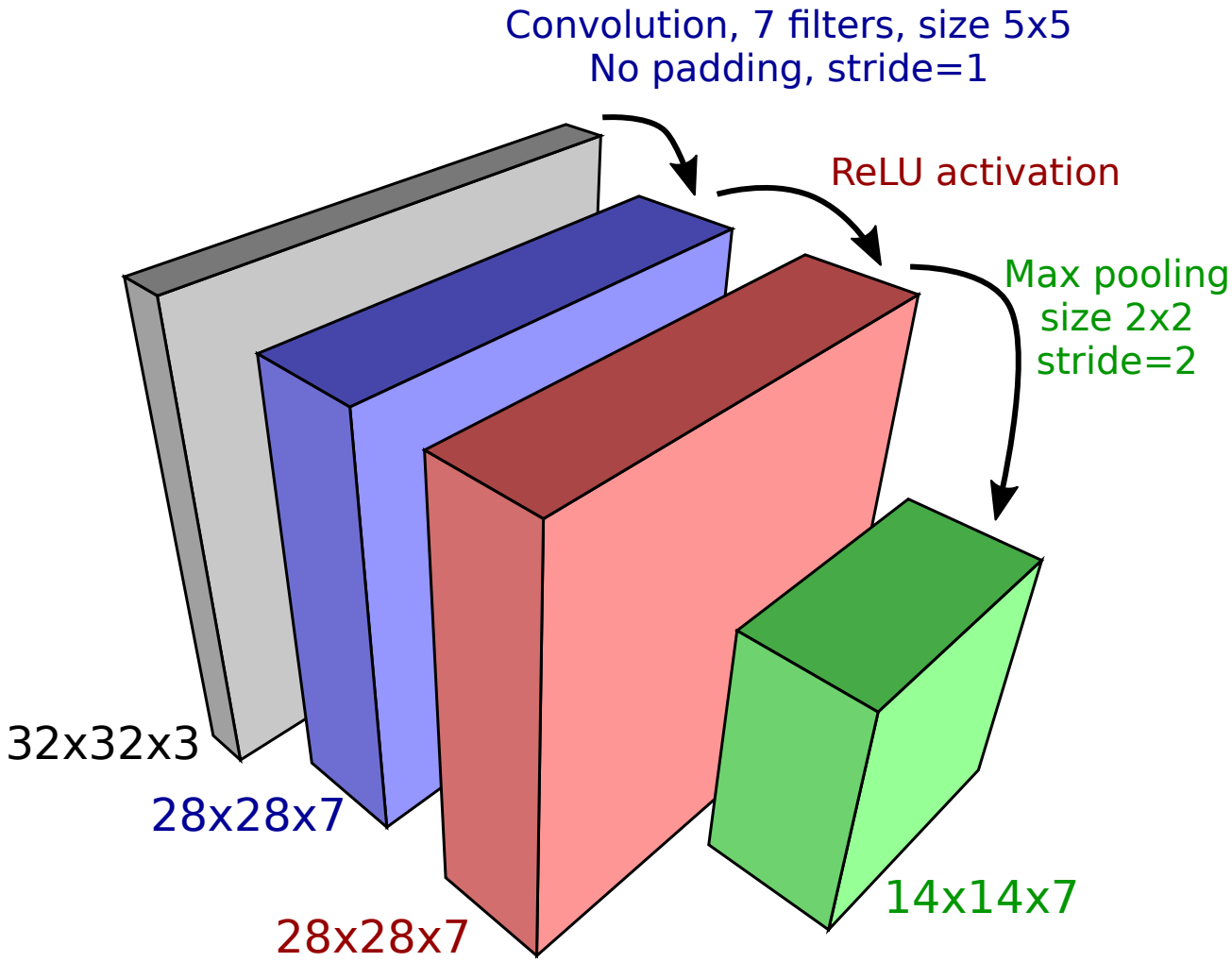
Pooling layer

- Goal : reduce the **spatial** size of the feature map $\rightarrow$ **downsampling**
- Number of output features B is unchanged, pooling is performed for each of these B features.
- No parameter to learn here
- Average or maximum taken over $p \times p$ squares of each feature map.
- Squares are usually non-overlapping : stride $= p$. Example : max pooling, $p = 2$



Size of feature maps

- Example of a convolutional sequence with sizes of feature maps



Parameter initialization

What should not be done : all zero initialization

- If all neurons compute the same output, they will undergo the same parameter update.
- No source of difference between neurons if their weights are initialized to the same value.

Small random numbers

- Sample from a normal distribution (zero-mean, unit standard deviation).
- Problem with the above suggestion : the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.
- The variance of each neuron's output can be normalized to 1 by scaling its weight vector by the square root of its number of inputs

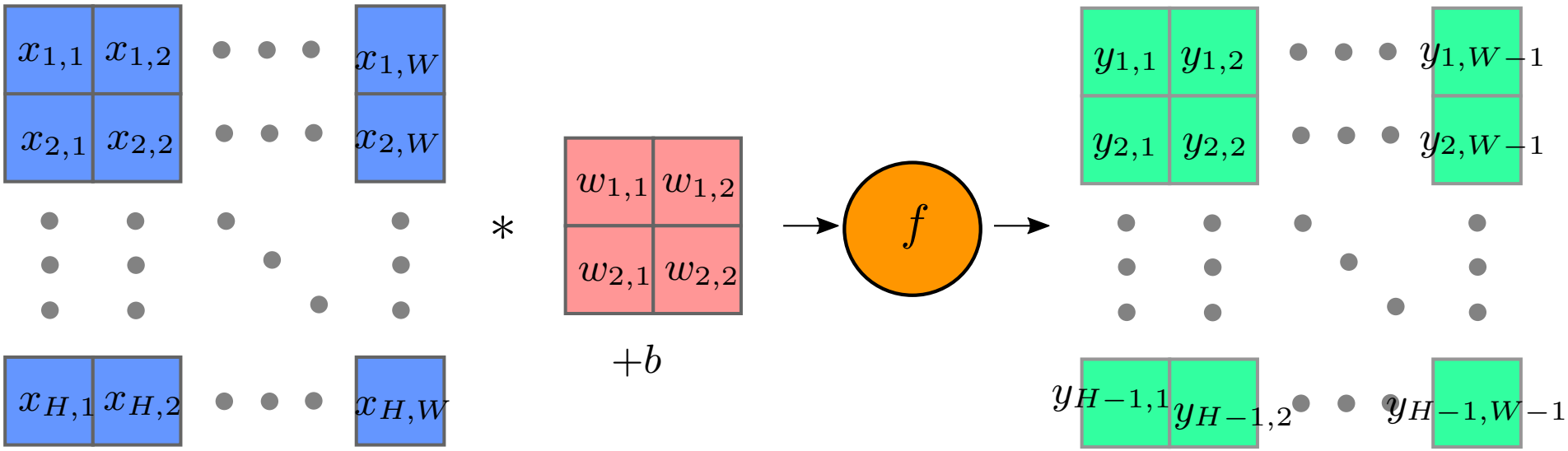
$$\mathbf{W} \sim \frac{1}{\sqrt{n}}\mathcal{N}(0,1)$$

with $n =$ number of input features.

Backpropagation

Backpropagation through convolutional layer + activation layer

- Consider the following example : convolutional layer with a 2×2 mask with bias, stride=1, no padding.
- Non-linear activation f.
- Input feature map size $= W \times H$, so output feature map size $= (W-1) \times (H-1)$



Backpropagation

- In the forward pass, each output $y_{i,j}$ is computed as

$$\begin{aligned} y_{i,j} &= f(w_{1,1}x_{i,j} + w_{1,2}x_{i,j+1} + w_{2,1}x_{i+1,j} + w_{2,2}x_{i+1,j+1} + b) \\ &= f\left(b + \sum_{\substack{\lambda=1..2 \\ \gamma=1..2}} w_{\lambda,\gamma}x_{i+\lambda-1,j+\gamma-1}\right) \end{aligned}$$

- Each weight $w_{\lambda,\gamma}$ and the bias b takes part in the calculation of every $y_{i,j}$
- During the backward pass, assume that each $\dfrac{\partial \mathcal{L}}{\partial y_{i,j}}$ has just been computed (coming from the following pooling layer)

Backpropagation

- For a given weight $w_{\lambda,\gamma}$:

$$\frac{\partial \mathcal{L}}{\partial w_{\lambda,\gamma}} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{\lambda,\gamma}}$$

$$= \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} x_{i+\lambda-1,j+\gamma-1} f' \left(b + \sum_{\substack{\lambda'=1..2 \\ \gamma'=1..2}} w_{\lambda',\gamma'} x_{i+\lambda'-1,j+\gamma'-1} \right)$$

- For the bias b,

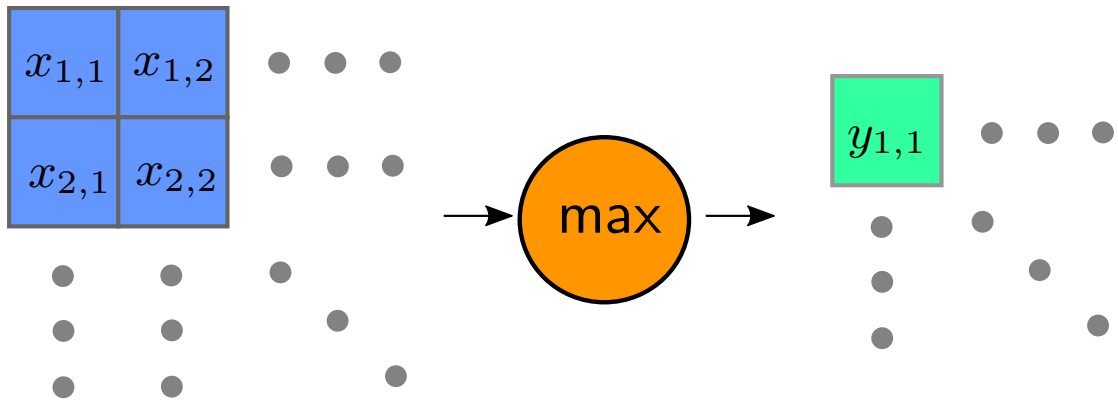
$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial b}$$

$$= \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} f' \left(b + \sum_{\substack{\lambda'=1..2 \\ \gamma'=1..2}} w_{\lambda',\gamma'} x_{i+\lambda'-1,j+\gamma'-1} \right)$$

Backpropagation

Backpropagation in pooling layer

- Max pooling, 2×2, stride=2 (no overlap)
- Input feature map size $= W \times H$, so output feature map size $= \frac{W}{2} \times \frac{H}{2}$



$$y_{i,j} = \max\{x_{2i-1,2j-1}\ ,\ x_{2i-1,2j}\ ,\ x_{2i,2j-1}\ ,\ x_{2i,2j}\}$$

$$= \max_{\substack{\lambda=-1..0 \\ \gamma=-1..0}} x_{2i+\lambda,2j+\gamma}$$

Backpropagation

- During the backward pass, assume that each $\frac{\partial \mathcal{L}}{\partial y_{i',j'}}$ has just been computed (coming from the following convolutional layer)
- The $\max$ function is not differentiable BUT...
- ... there's a trick : during the forward pass, store, for each $y_{i',j'}$, the position in feature map x which led to the maximum :

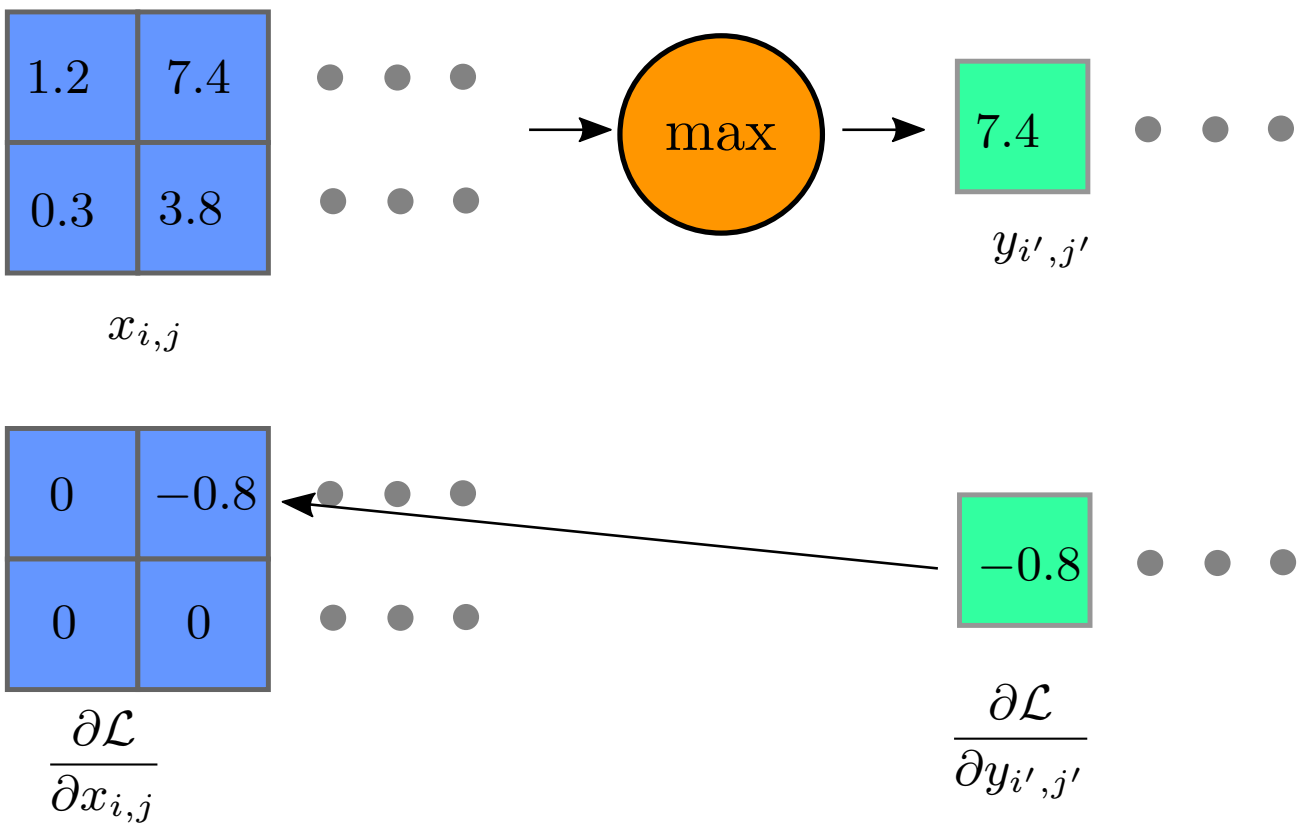
$$(\lambda^*, \gamma^*) = \underset{\substack{\lambda=-1..0 \\ \gamma=-1..0}}{\text{argmax}}\, x_{2i'+\lambda, 2j'+\gamma}$$

- A given input $x_{i,j}$ takes part in the computation of $y_{i',j'}$, where $i' = \lfloor (i-1)/2 \rfloor + 1$ and $j' = \lfloor (j-1)/2 \rfloor + 1$. Hence,

$$\frac{\partial \mathcal{L}}{\partial x_{i,j}} \leftarrow \begin{cases} \frac{\partial \mathcal{L}}{\partial y_{i',j'}} & \text{if } i = 2i' + \lambda^*, j = 2j' + \gamma^* \\ 0 & \text{otherwise} \end{cases}$$

Backpropagation

- Example : during forward pass, $x_{1,2}$ has the maximum value among $\{x_{1,1}\ ,\ x_{1,2}\ ,\ x_{2,1}\ ,\ x_{2,2}\}$
- During backward pass, $\dfrac{\partial \mathcal{L}}{\partial y_{1,1}}$ is backpropagated to $\dfrac{\partial \mathcal{L}}{\partial x_{1,2}}$
- The other $\dfrac{\partial \mathcal{L}}{\partial x_{i,j}}$ are set to 0



Regularization : parameter norm penalties

- The goal of regularization is to prevent overfitting (limit generalization error).
- First technique : add a **regularization term** (some norm over the vector of parameters) in the cost function

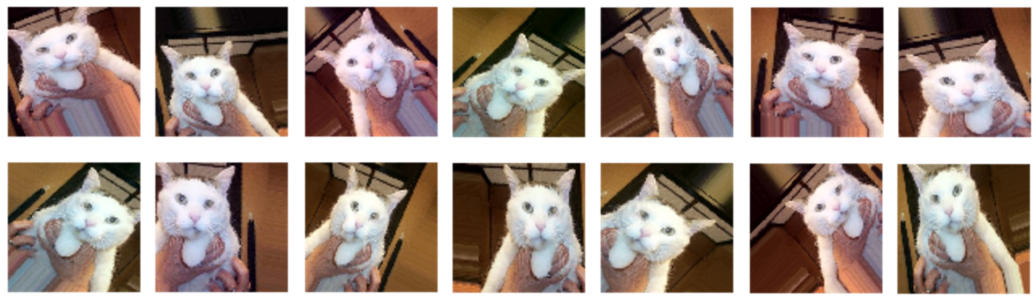
$$\mathcal{C}(\theta) = \sum_{i=1}^{N} \mathcal{L}(\mathcal{M}(\boldsymbol{x}_i; \theta), \boldsymbol{y}_i) + \lambda \|\theta\|$$

  where λ is a hyperparameter.
- For neural networks, $\|\theta\|$ penalizes only the weights, and leave the biases
- L^2 norm (the most common form of regularization) $\rightarrow$ encourages the network to use all of its inputs a little, rather than some of its inputs a lot
- L^1 norm $\rightarrow$ leads the weight vectors to become **sparse** during optimization

Regularization : data augmentation

- Train on more data ! Of course, in practice, the amount of data is limited.
- Create "fake" data and add it to the training set

- Data augmentation by affine transformation :



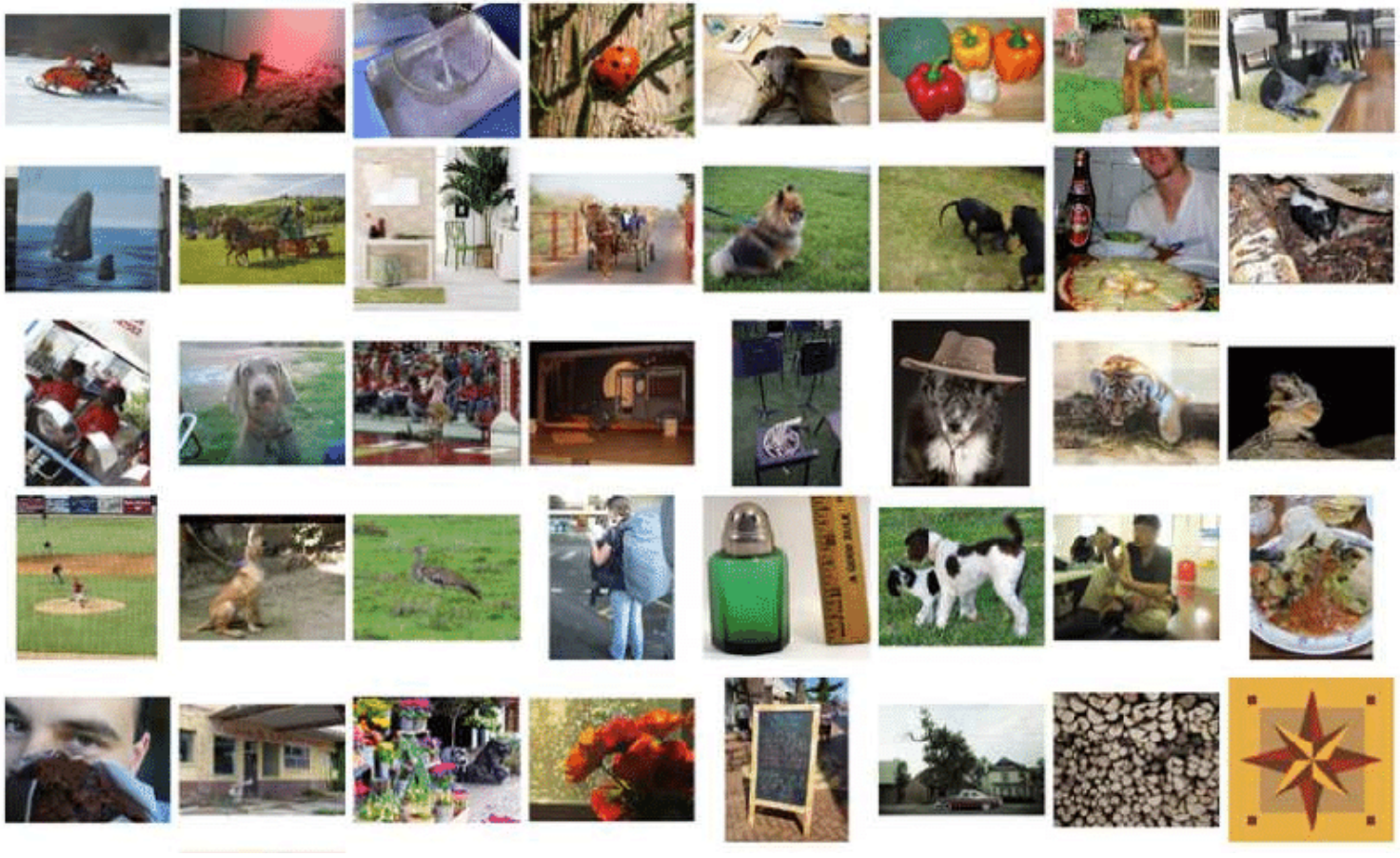
- Another possibility is data augmentation by injecting noise in the inputs

Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Outline

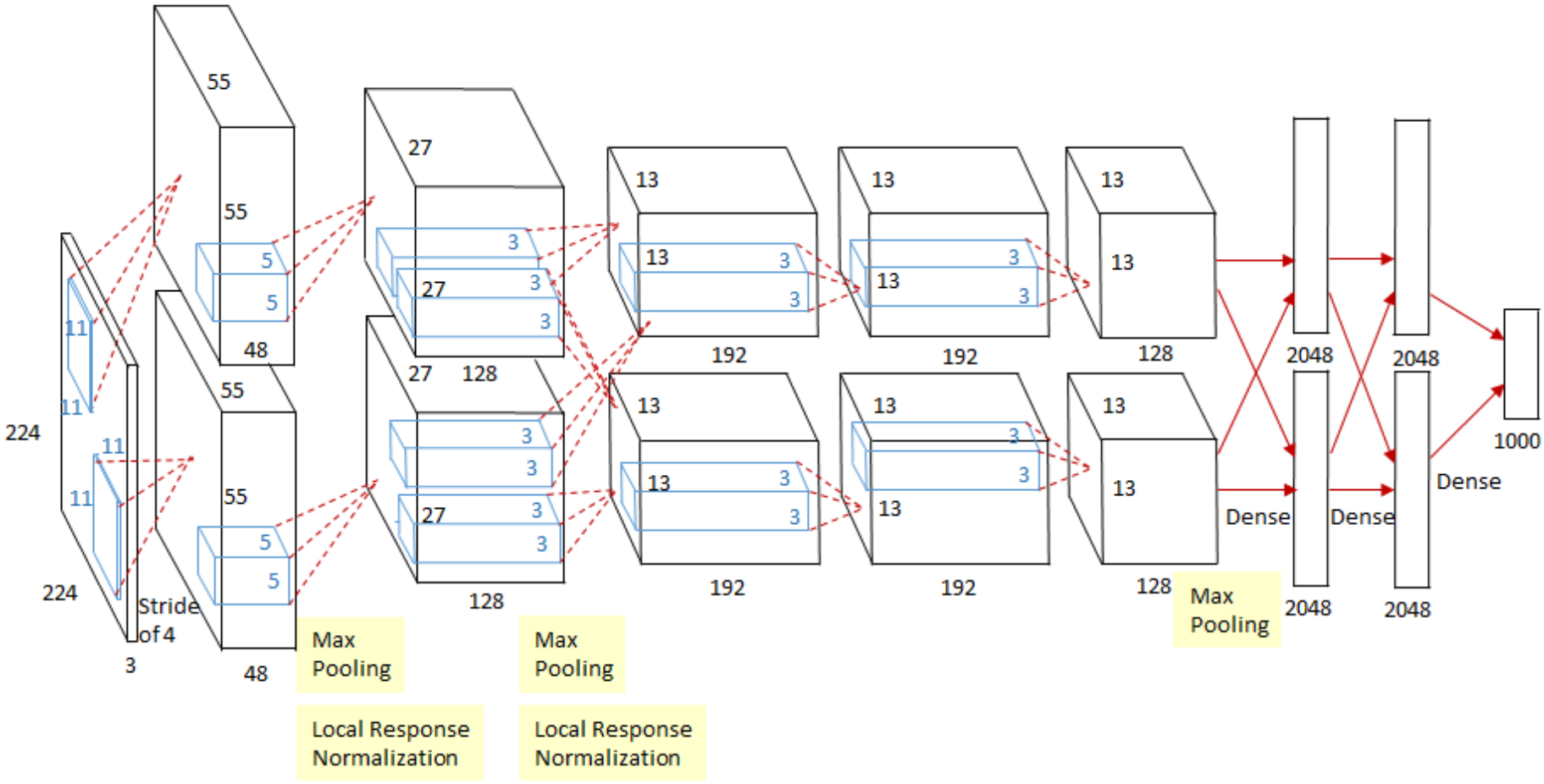
The ImageNet dataset

- The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale.



AlexNet

- Won the ILSVRC 2012 challenge
- [A. Krizhevsky, I. Sutskever, G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *NIPS* 2012]



AlexNet : Local response normalization (LRN)

- Not used much in other architectures...
- In neurobiology, "lateral inhibition" = capacity of an excited neuron to subdue its neighbors.
- Favors detection of high frequency features with a large response.
- If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors.
- Useful with ReLU activations, which are unbounded.
- LRN will dampen the responses that are uniformly large in any given local neighborhood. If all the values are large, then normalizing those values will diminish all of them.

AlexNet : Local response normalization (LRN)

- Normalization is performed **across** filters of the same convolutional layer (after ReLU activation).
- At a fixed spatial position (i, j) :

$$y_{i,j}^{k} = \frac{x_{i,j}^{k}}{\left(\alpha_1 + \alpha_2 \sum_{\lambda=k-n/2}^{k+n/2} \left(x_{i,j}^{\lambda}\right)^2\right)^{\beta}}$$

- $x_{i,j}^{k}$ = the k^{th} input feature to the LRN layer (output of the k^{th} filter of the previous convolutional layer, after ReLU activation).
- $y_{i,j}^{k}$ = the k^{th} output of the LRN layer.
- n = size of neighborhood (= set of neighboring filters) ($\lambda - n/2$ and $\lambda + n/2$ are bounded between 0 and $N - 1$, where N is the number of filters).
- α_1, α_2, β, n are hyperparameters.
- No parameter to learn here.

AlexNet in detail

- Input : $224 \times 224 \times 3$ input images
- 1st Convolutional layer : 96 filters of size $11 \times 11 \times 3$ (stride $= 4$, no padding) $\rightarrow 55 \times 55 \times 96$ feature maps
- ReLU
- Max-pooling layer : 3×3 (stride $= 2$) $\rightarrow 27 \times 27 \times 96$ feature maps
- Local Response Normalization
- 2nd Convolutional layer : 256 filters of size $5 \times 5 \times 48$ (stride $= 1$, padding$=2$) $\rightarrow 27 \times 27 \times 256$ feature maps
- ReLU
- Max-pooling layer : 3×3 (stride $= 2$) $\rightarrow 13 \times 13 \times 256$ feature maps
- Local Response Normalization

AlexNet in detail

- 3rd Convolutional layer : 384 filters of size $3 \times 3 \times 256$ (stride $= 1$, padding$=1$) $\rightarrow 13 \times 13 \times 384$ feature maps
- 4th Convolutional layer : 384 filters of size $13 \times 13 \times 192$ (stride $= 1$, padding$=1$) $\rightarrow 13 \times 13 \times 384$ feature maps
- 5th Convolutional layer : 256 filters of size $3 \times 3 \times 192$ (stride $= 1$, padding$=1$) $\rightarrow 13 \times 13 \times 256$ feature maps
- Max-pooling layer : 3×3 (stride $= 2$) $\rightarrow 6 \times 6 \times 256$ feature maps
- 1st Fully connected layer : 4096 neurons
- 2nd Fully connected layer : 4096 neurons
- 3st Fully connected layer : 1000 neurons
- In total, there are 60 million parameters need to be trained !

Use of pretrained models

- Training a model on ImageNet from scratch takes days or weeks.
- Many models trained on ImageNet and their weights are publicly available !
- We can perform fine-tuning for transfer learning
- Retraining the/some parameters of the network (given enough data)
- Truncate the last layer(s) of the pre-trained network
- Train a classification model from these features on a new classification task (early layers are frozen, only late layers are trained)

Outline

Outline

Why recurrent neural networks ?

Feedforward neural networks

- All input values of a given sample are considered independent of each other.
- When processing sequential data, values at different time steps would also be considered independent $\rightarrow$ does not take advantage of time coherence !
- Cannot handle variable-length sequential data, *e.g* sentences.

Recurrent neural networks

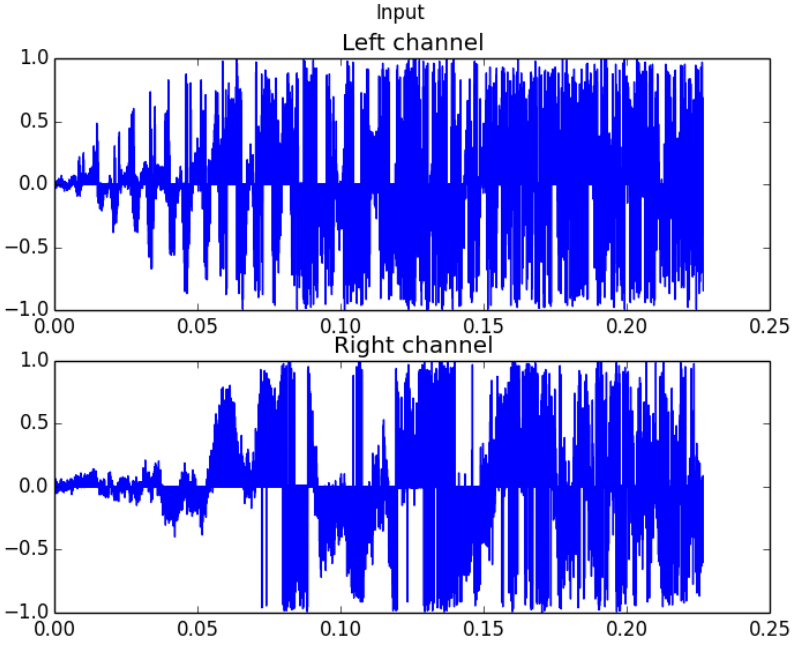
- Make use of **sequential** information
- Output is made dependent on previous computations
- Recurrent neurons have a **memory** = internal **hidden state**
- Can handle variable-length sequences

Sequential data

- Let $\mathbf{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T)$ be a sequence of T vectors of size D
- Processing sequential data is roughly equivalent to predict what comes next :

$$p(\mathbf{X}) = \prod_{t=1}^{T} p(\boldsymbol{x}_t | \boldsymbol{x}_1, ..., \boldsymbol{x}_{t-1})$$

- Examples of sequential data :
  - Sound wave : $D = 1$ (mono) or $D = 2$ (stereo)



Sequential data

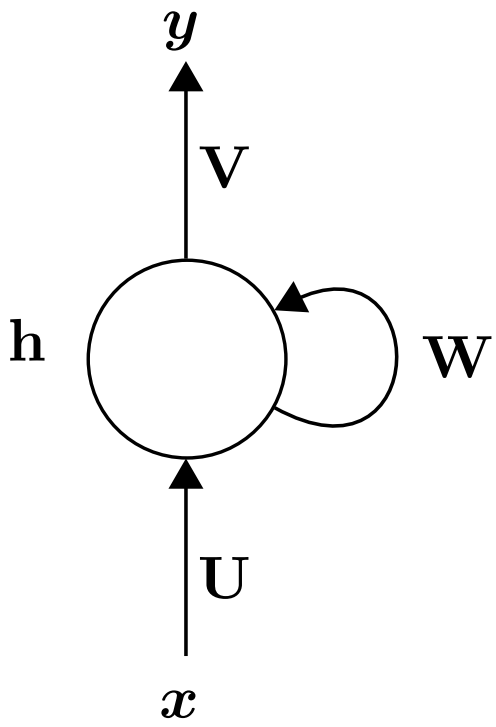
- Sentences : sequence of words
- Initial encoding of words : 1-of-V vectors, where V is the size of vocabulary (huge !)

The	man	is	wearing	a	hat
0	**1**	0	0	0	0
⋮	0	**1**	⋮	⋮	⋮
0	⋮	0	⋮	**1**	⋮
1	⋮	⋮	0	0	⋮
0	⋮	⋮	**1**	⋮	0
⋮	⋮	⋮	0	⋮	**1**
0	0	0	0	⋮	0

- Words are re-encoded in a space of smaller dimension (embedding)

Recurrent neurons

- A recurrent **layer** of neurons (input and output are vectors !) :
  - $\boldsymbol{x}$ = input vector of size D
  - $\boldsymbol{y}$ = output vector of size K
  - $\mathbf{h}$ = hidden state vector of size H
  - $\mathbf{U} = H \times D$ weight matrix
  - $\mathbf{V} = K \times H$ weight matrix
  - $\mathbf{W} = H \times H$ weight matrix



- The hidden state are the "memory" of the neuron. It is calculated based on the previous hidden state and the current input :

$$\mathbf{h}_t = f_1(\mathbf{U}\boldsymbol{x}_t + \mathbf{W}\mathbf{h}_{t-1})$$

- The output is calculated based on the hidden state :

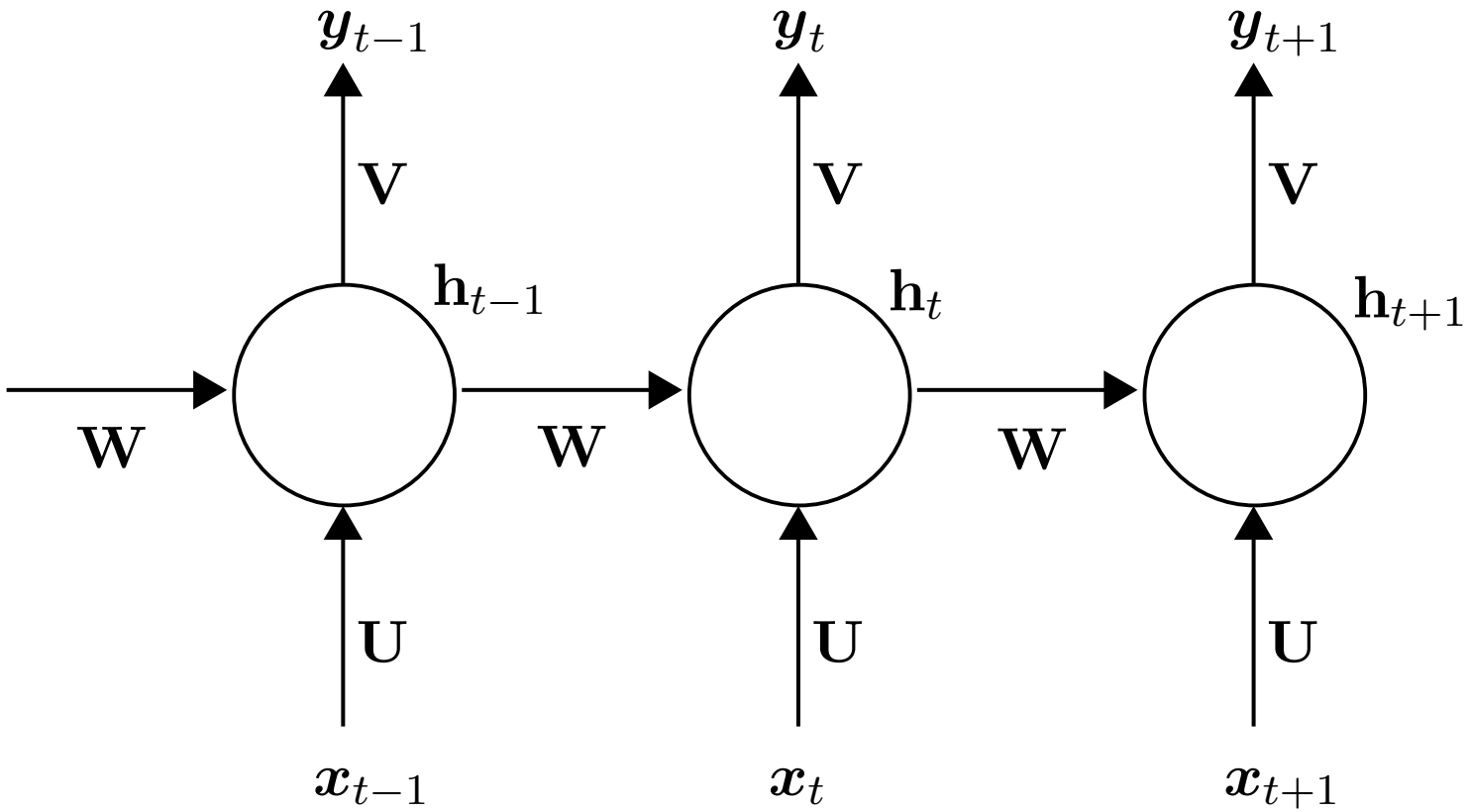
$$\boldsymbol{y}_t = f_2(\mathbf{V}\mathbf{h}_t)$$

Recurrent neurons

- The hidden state captures information about what happened in **all the previous time steps** (in practice, it typically cannot capture information from too many time steps ago)
- Unlike a traditional deep neural network, which uses different parameters at each layer, a RNN shares the same parameters ($\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ above) across all time steps
- We are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.

Recurrent neurons

- The same reccurent layer, **unfolded** in time :



Outline

Backpropagation through time (BPTT)

- The full sequence is one training sample $\rightarrow$ the loss for one sample is the sum of the losses at each time step.

$$\mathcal{L}(\boldsymbol{y}, \overline{\boldsymbol{y}}) = \sum_{t=1}^{T} \mathcal{L}(\boldsymbol{y}_t, \overline{\boldsymbol{y}}_t)$$

  where $\overline{\boldsymbol{y}}$ is the desired output (label) (sample index i is dropped for convenience)
- Let $\mathcal{L}_t$ be a shortened notation for $\mathcal{L}(\boldsymbol{y}_t, \overline{\boldsymbol{y}}_t)$.
- To train the recurrent layer, we should compute $\frac{\partial \mathcal{L}_t}{\partial \mathbf{U}}$, $\frac{\partial \mathcal{L}_t}{\partial \mathbf{V}}$ and $\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}}$ for every t
- We are differentiating real numbers and vectors with respect to vectors and matrices! $\rightarrow$ matrix calculus

Backpropagation through time (BPTT)

- From now on, we will reason on a single recurrent neuron with $D = 1$, $K = 1$, $H = 1$.
- Input x_t, output y_t and hidden state h_t are scalars.
- Weights u, v, w are scalars too.

$$\begin{aligned} h_t &= f_1(ux_t + wh_{t-1}) \\ y_t &= f_2(vh_t) \end{aligned}$$

- For v, we have the easy relation

$$\frac{\partial \mathcal{L}_t}{\partial v} = \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial v} = \frac{\partial \mathcal{L}_t}{\partial y_t} h_t f_2'(vh_t)$$

Backpropagation through time (BPTT)

- For w, we have the following **recurrent** relation :

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial w} &= \frac{\partial \mathcal{L}_t}{\partial y_t}\frac{\partial y_t}{\partial w} \\ &= \frac{\partial \mathcal{L}_t}{\partial y_t}\frac{\partial h_t}{\partial w} v f_2'(v h_t)\end{aligned}$$

with

$$\begin{aligned}\frac{\partial h_t}{\partial w} &= \frac{\partial}{\partial w}\left\{ f_1(u x_t + w h_{t-1}) \right\} \\ &= \left(h_{t-1} + w\frac{\partial h_{t-1}}{\partial w} \right) f_1'(u x_t + w h_{t-1}) \\ &= \ldots\end{aligned}$$

- Recurrence stops as $\dfrac{\partial h_0}{\partial w} = 0.$

Backpropagation through time (BPTT)

- Similarly, for u, we have the following **recurrent** relation :

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial u} &= \frac{\partial \mathcal{L}_t}{\partial y_t}\frac{\partial y_t}{\partial u} \\ &= \frac{\partial \mathcal{L}_t}{\partial y_t}\frac{\partial h_t}{\partial u} v f_2'(v h_t)\end{aligned}$$

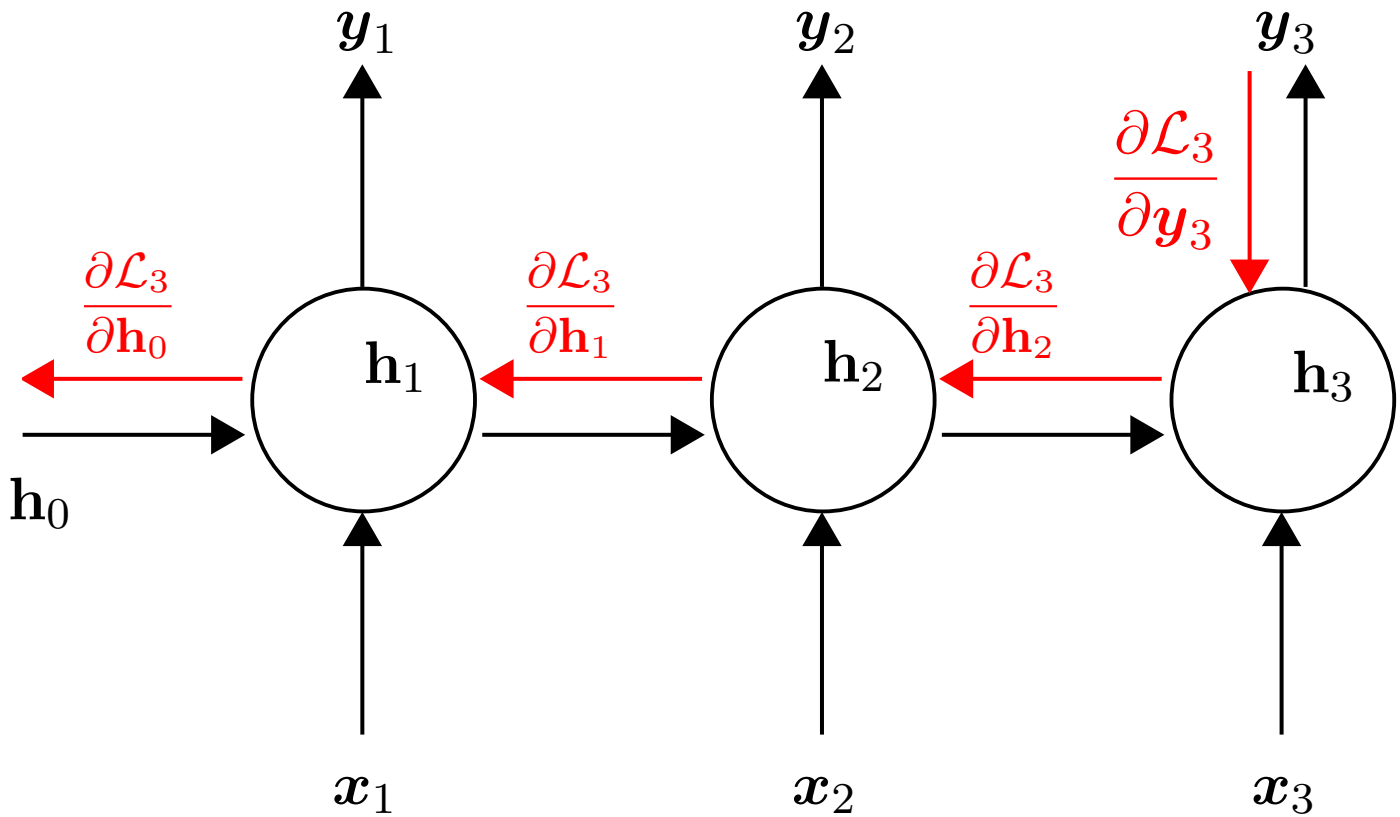
with

$$\begin{aligned}\frac{\partial h_t}{\partial u} &= \frac{\partial}{\partial u}\left\{ f_1(u x_t + w h_{t-1}) \right\} \\ &= \left(x_t + w \frac{\partial h_{t-1}}{\partial u} \right) f_1'(u x_t + w h_{t-1}) \\ &= \ldots\end{aligned}$$

- Recurrence stops as $\dfrac{\partial h_0}{\partial u} = 0.$

Backpropagation through time (BPTT)

- Backpropagation through time for a given loss at time step $t = 3$:



Outline

Neural Language Processing

- Possible tasks in Neural Language Processing (NLP) : neural machine translation (NMT), visual question answering, chatbots, ...
- A recurrent neural network learns a language model, assigning a probability to a sequence of words
- Plausible sequences have higher probabilities :

$$p(\text{“I like cats“}) > p(\text{”I table cats“})$$
$$p(\text{”I like cats“})) > p(\text{”like I cats“})$$

- Words are initially represented as 1-of-V vectors $\rightarrow$ vocabulary size V is huge !
- For NLP, inputs of recurrent neural networks are **embeddings**

Word embeddings

- Word embeddings have a size much smaller than V
- Example :

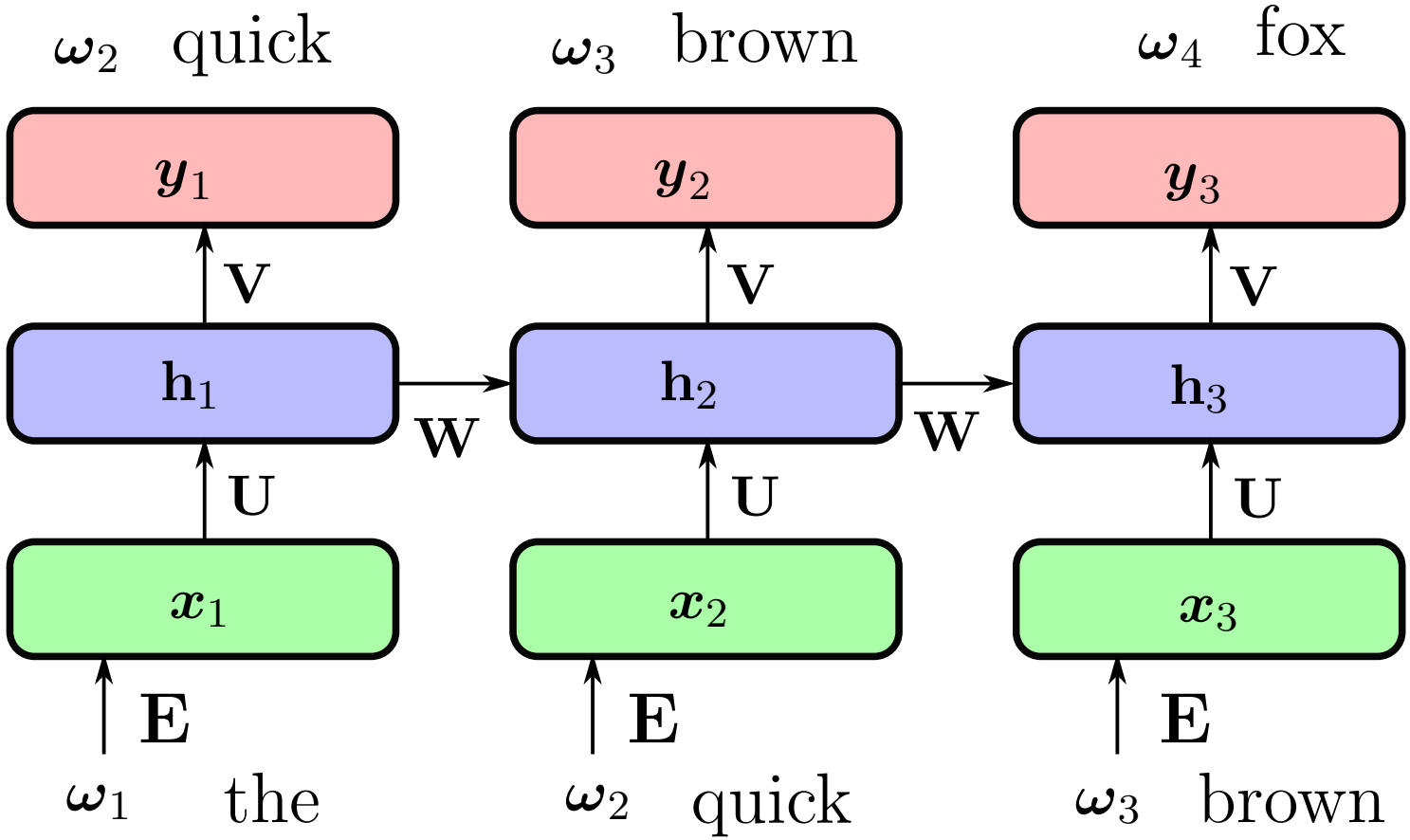
  Apple : $[1.11, 2.24, 7.88]$
  Orange : $[1.01, 2.04, 7.22]$
  Car : $[8.41, 2.34, -1.28]$
  Table : $[-1.41, 7.34, 3.01]$
- Embeddings are different (*e.g.* in terms of Euclidean distance) if corresponding words are semantically different
- Initial sequence : $(\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, ...\boldsymbol{\omega}_T)$
- Sentence fed as input to the RNN : $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ...\boldsymbol{x}_T)$, with

$$\boldsymbol{x}_t = \mathbf{E}\boldsymbol{\omega}_t$$

  where $\mathbf{E}$ is the **embedding** operator (projection)

Language modeling

- Input : sequence $(\omega_1, \omega_2, ...\omega_T)$
- Output : shifted sequence $(\omega_2, \omega_3, ...\omega_{T+1})$



Figures by O. Grisel and C. Ollion, `https://github.com/m2dsupsdlclass`

Language modeling

- Neural machine translation (NMT)
- The hidden state at the last iteration encodes the memory for the entire sentence

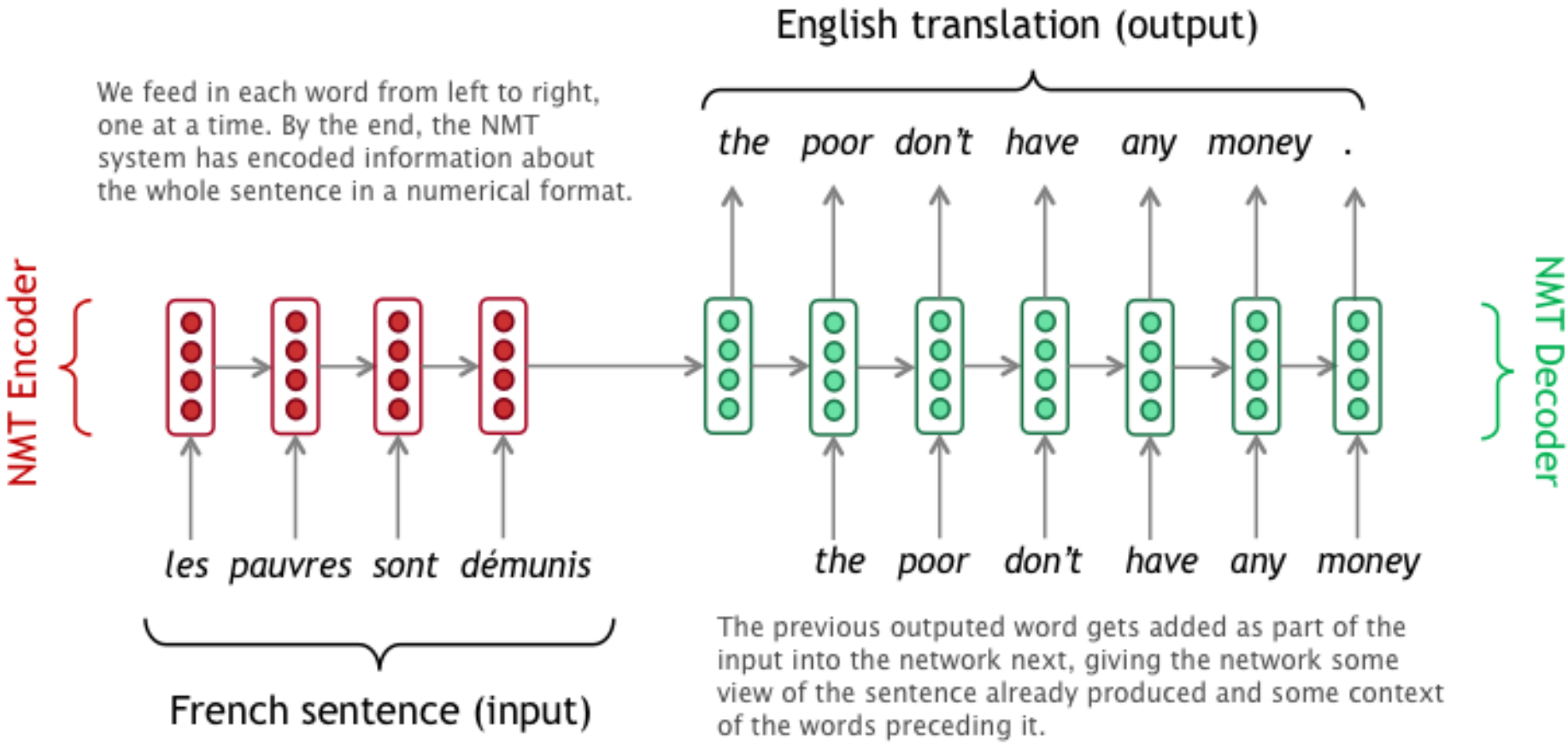


Figure by A. See, `http://web.stanford.edu/class/cs224n`

Conclusion

Deep learning models

- are powerful, for many many tasks !
- need A LOT of annotated data,
- move the problem of feature engineering to architecture engineering

Forthcoming challenges

- Explainability of learnt features
- Make architectures less time and memory-consuming (decrease the number of layers/parameters without performance loss)
- Public debate on algorithms, artificial intelligence, ethics... (what AI can do, what it cannot do, what it should not be used for...)