

Introduction to Machine Learning & Deep Learning

Sorbonne Université – Master DAC. Gallinari, patrick.gallinari@lip6.fr, <http://www-connex.lip6.fr/~gallinar/>

Nicolas Baskiotis, Benjamin Piwowarski, Laure Soulier, nom.prenom@lip6.fr

Année 2020-2021

Course Outline and Organization

- ▶ Introductory ML course with a focus on Neural Networks and Deep Learning
- ▶ Organization
 - ▶ Courses 14 x 2 h – P. Gallinari
 - ▶ Practice and exercises 14 x 2 h
- ▶ Outline
 - ▶ Introduction
 - ▶ Basic Concepts of Machine Learning
 - ▶ Neural Networks and Deep Learning
 - ▶ Introductory Concepts of - Perceptron-Adaline
 - ▶ Linear Regression and Logistic Regression - Optimization Basics
 - ▶ Multilayer Perceptrons – Generalization Properties
 - ▶ Convolutional Neural Networks – Vision applications
 - ▶ Recurrent Neural Networks – Language applications
 - ▶ Support Vector Machines
 - ▶ Gaussian processes
 - ▶ Unsupervised Learning
 - ▶ EM algorithm and density mixture
 - ▶ Variational methods

Ressources

- ▶ Books
 - ▶ The following two books cover the course (more or less)
 - ▶ Deep Learning, An MIT Press book, I. Goodfellow, Y. Bengio and A. Courville, 2017
 - <http://www.deeplearningbook.org/>
 - ▶ Pattern recognition and Machine Learning, C. Bishop, Springer, 2006
 - Chapters 3, 4, 5, 6, 7, 9,
 - ▶ Many other books can be profitable, e.g.
 - ▶ The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition, T. Hastie, R. Tibshirani, J. Friedman, Springer, 2009
 - Version pdf accessible : <http://statweb.stanford.edu/~tibs/ElemStatLearn/>
 - ▶ Bayesian Reasoning and Machine Learning, D. Barber, Cambridge University Press, 2012
 - Version pdf accessible : <http://www.cs.ucl.ac.uk/staff/d.barber/brml/>
- ▶ Courses
 - ▶ Several on line ressources, covering this topic and others
 - ▶ Course slides and material: Machine Learning, Deep Learning for Vision, Natural Language Processing, ...
 - ▶ MOOCs: e.g. Andrew Ng ML course on Coursera
- ▶ Software Platforms
 - ▶ ... introduced in the practice sessions



Machine Learning General Framework

- 4 learning problems
- Risk, Empirical Risk

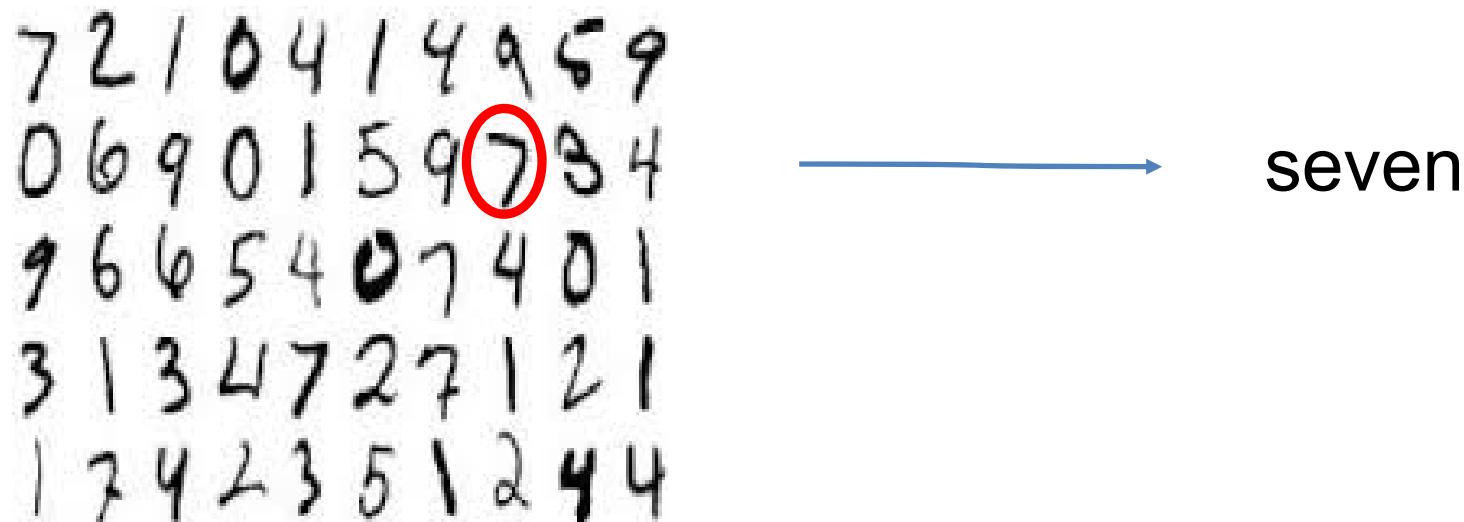
4 learning problems

- ▶ ML develops generic methods for solving different types of problems
- ▶ Usually, one makes the distinction between 4 problem types
 - ▶ Supervised
 - ▶ Unsupervised
 - ▶ Semi-supervised
 - ▶ Reinforcement

4 learning problems

Supervised learning

- ▶ Training set: couples (inputs, target) $(x^1, y^1), \dots, (x^N, y^N)$
- ▶ Objective : learn to associate inputs to outputs
 - ▶ With good generalization properties
- ▶ Classical problems: classification, regression, ranking



- ▶ Most applications today fall under the supervised learning paradigm

4 learning problems

Unsupervised learning

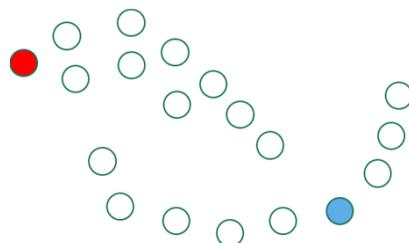
- ▶ **Training set**
 - ▶ Only input data x^1, \dots, x^N , no target
- ▶ **Objective**
 - ▶ Extract some regularities from data
 - ▶ Similarities, relations between items, latent factors explaining data generation
- ▶ **Use**
 - ▶ Density estimation, clustering, latent factors identification, generative models



4 learning problems

Semi-supervised learning

- ▶ Task
 - ▶ Similar to supervised learning
- ▶ Training set
 - ▶ Small number of labeled data $(x^1, y^1), \dots, (x^N, y^N)$
 - ▶ Large number of unlabeled data x^{N+1}, \dots, x^{N+M}
- ▶ Objective
 - ▶ Extract information from unlabeled data useful for labeling examples
 - ▶ e.g. structure
 - ▶ Joint learning from the two datasets

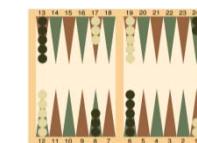
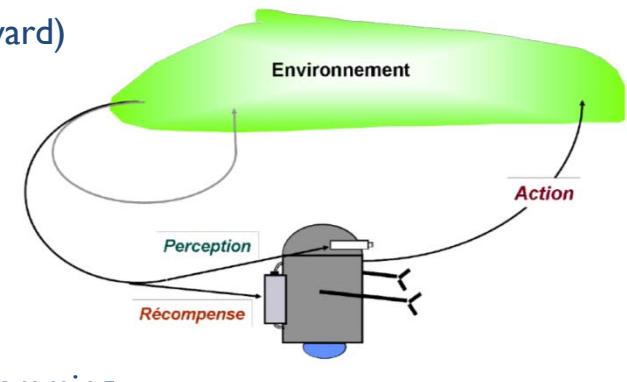


- ▶ Use
 - ▶ When large amounts of data are available and labeling is costly

4 learning problems

Reinforcement learning

- ▶ Training set
 - ▶ Couples (input, qualitative target)
 - ▶ x^i 's may be sequences (temporal credit assignment), y^i are qualitative targets (e.g. 0,1), deterministic or stochastic
- ▶ Paradigm
 - ▶ Learning by exploring the environment, using reinforcement signals (reward)
 - ▶ Exploration/ exploitation paradigm
- ▶ Use
 - ▶ command, sequential decision, robotics, two players game, dynamic programming, ...
 - ▶ RL for games
 - ▶ Backgammon (TD Gammon Thesauro 1992)
 - ▶ Trained on 1.5 M plays
 - ▶ Plays against itself
 - ▶ Deep RL
 - ▶ AlphaGo (2015), AlphaGo Zero (2017)
 - ▶ Alphazero (2017)



Risk – Empirical Risk

Probabilistic formalism

- ▶ Data
 - ▶ Random vectors (\mathbf{z}) generated from distribution $p(\mathbf{z})$
- ▶ Learning model
 - ▶ $\mathcal{F} = \{F_\theta\}_\theta$ with θ the model parameters, usually real parameters
- ▶ Loss
 - ▶ $c_\theta(\mathbf{z})$ for model F_θ and example \mathbf{z}
- ▶ Risk
 - ▶ $R_\theta = E_{\mathbf{z}}[c_\theta(\mathbf{z})] = \int_{\mathbf{z}} c_\theta(\mathbf{z}) p(\mathbf{z}) d\mathbf{z}$
- ▶ Optimal solution
 - ▶ $F_{\theta^*} = \operatorname{argmin}_\theta R_\theta$

Risk – Empirical Risk

Learning from examples

- ▶ Data

- ▶ $D = \{\mathbf{z}^i\}_{i=1..N}$

- ▶ Empirical risk

- ▶ $C = \frac{1}{N} \sum_{i=1}^N c_\theta(\mathbf{z}^i)$

- ▶ Empirical risk minimization principle

- ▶ F_{θ^*} minimizing the theoretical risk is approximated by $F_{\hat{\theta}}$ minimizing the empirical risk
- ▶ Is that sufficient ? Answer is No

- ▶ Inductive framework

- ▶ We will consider the following ML framework

- ▶ The model learns on an available training set
 - ▶ Once trained parameters are fixed and the model can be used for inference and/or evaluated on a test set

Example of generic ML problems

▶ Classification

- ▶ $\mathbf{z} = (\mathbf{x}, y), y \in \{0,1\}$
- ▶ F_θ threshold functions
- ▶ R : probability of incorrect classification
- ▶ C : error frequency

$$c_\theta(\mathbf{z}) = \begin{cases} 0 & \text{if } y = F_\theta(\mathbf{x}) \\ 1 & \text{otherwise} \end{cases}$$

▶ Regression

- ▶ $\mathbf{z} = (\mathbf{x}, y), y \in \mathbb{R}$
- ▶ F_θ real functions (e.g. linear NNs)
- ▶ R : expectation of quadratic error
- ▶ C : sum of quadratic errors

$$c_\theta(\mathbf{z}) = \|y - F_\theta(\mathbf{x})\|^2$$

▶ Density estimation

- ▶ $\mathbf{z} = \mathbf{x}$
- ▶ F_θ real functions
- ▶ R : likelihood (expectation)
- ▶ C : empirical estimator of likelihood (sum)

$$c_\theta(\mathbf{z}) = -\ln p_\theta(\mathbf{x})$$

Neural Networks and Deep Learning

Context

Context

Deep Learning today

- ▶ Deep Learning is today the most popular paradigm in data science
- ▶ Popularized since 2006, first by some academic actors and then by big players (GAFAs, BATs, etc)
- ▶ It has initiated a « paradigm shift » in the field of data science / AI and definitely changed the way one will exploit data
 - ▶ e.g. key players have made available development platforms (e.g. TensorFlow, PyTorch)
 - ▶ Allowing the development in a « short time » of complex processing chains
 - ▶ Making complex DL methods available for a large community
 - ▶ This shift will most probably influence other scientific domains as well in a near future
 - ▶ More generally, knowledge based procedures are progressively replaced by learning machines

Context - Examples

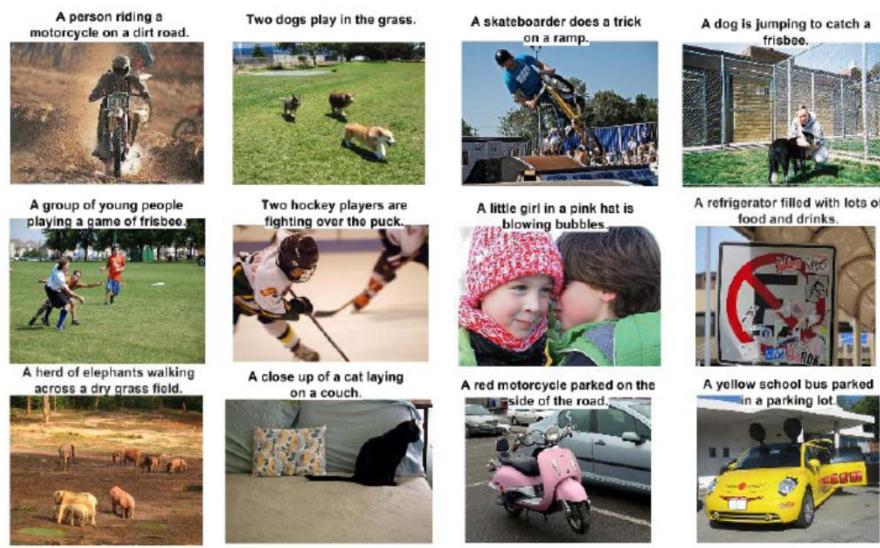
Computer Vision



Segmentation +
classification, Mask
R-CNN, (He 2017)

Image Captioning (Vinyals 2015)

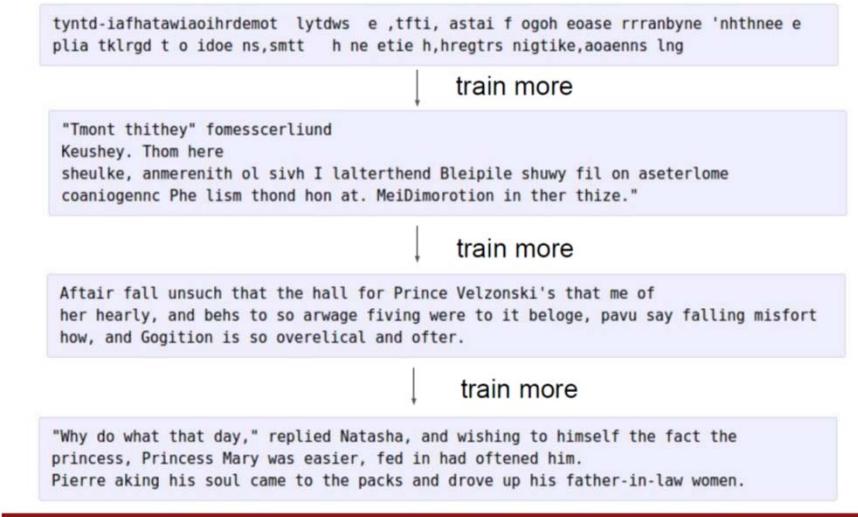
Mach



Describes without errors Describes with minor errors Somewhat related to the image Unrelated to the image

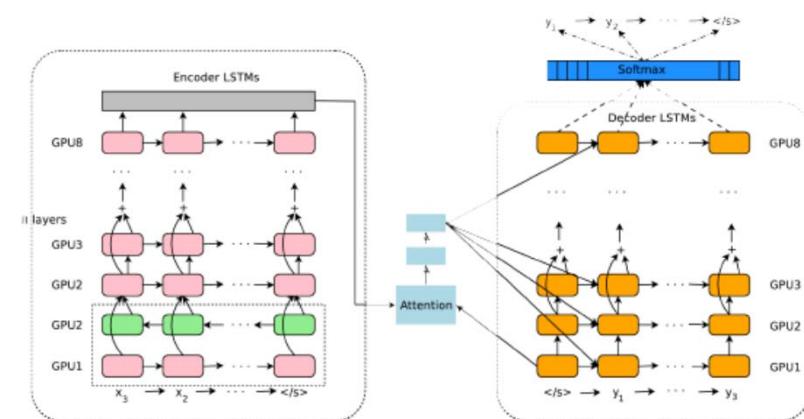
Figure 5. A selection of evaluation results, grouped by human rating.

Context -Examples Language



Language generation, Training
on Tolstoy's War and Peace a
character language model,
(Karpathy 2015-
<https://karpathy.github.io/2015/05/21/rnneffectiveness/>)

Google Translation
model, (Wu 2016)



Context -Examples

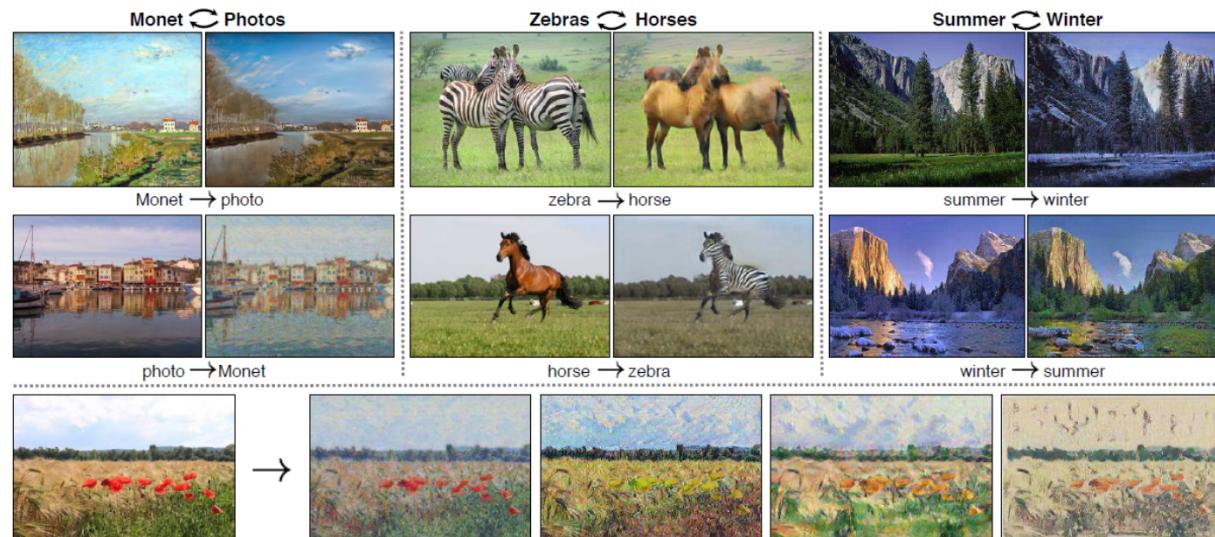
Generative models

Image generation, (Radford 2015)



Figure 3: Generated bedrooms after five epochs of training. There appears to be evidence of visual under-fitting via repeated noise textures across multiple samples such as the base boards of some of the beds.

CycleGan
Image Translation, (Zhu 2017)



Context-Examples

Games (not considered in this course)

▶



Atari games, Self trained on 49 games, (Mnih 2013, 2015)

Figure 1: Screen shots from five Atari 2600 Games: (Left-to-right) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

AlphaGo, AlphaGo Zero, Alpha Zero from Google DeepMind (2015, 2017)

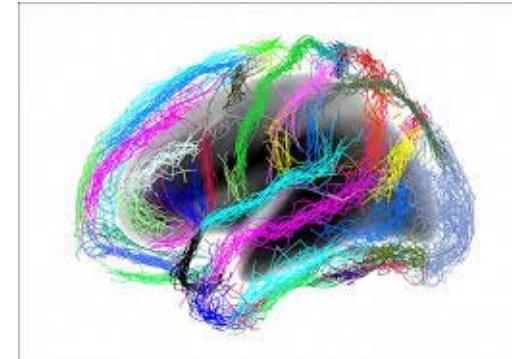
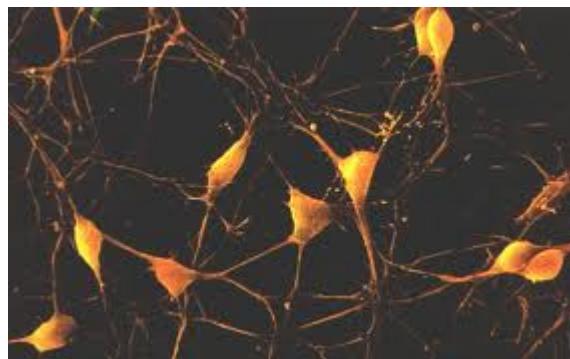


Introductory NN concepts

Intuitive introduction via 2 simple –historical- models
Perceptrons and Adalines

Neural Networks inspired Machine Learning

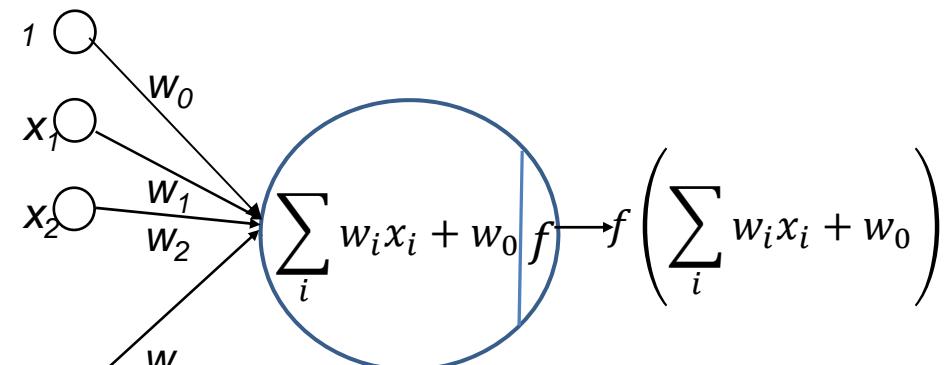
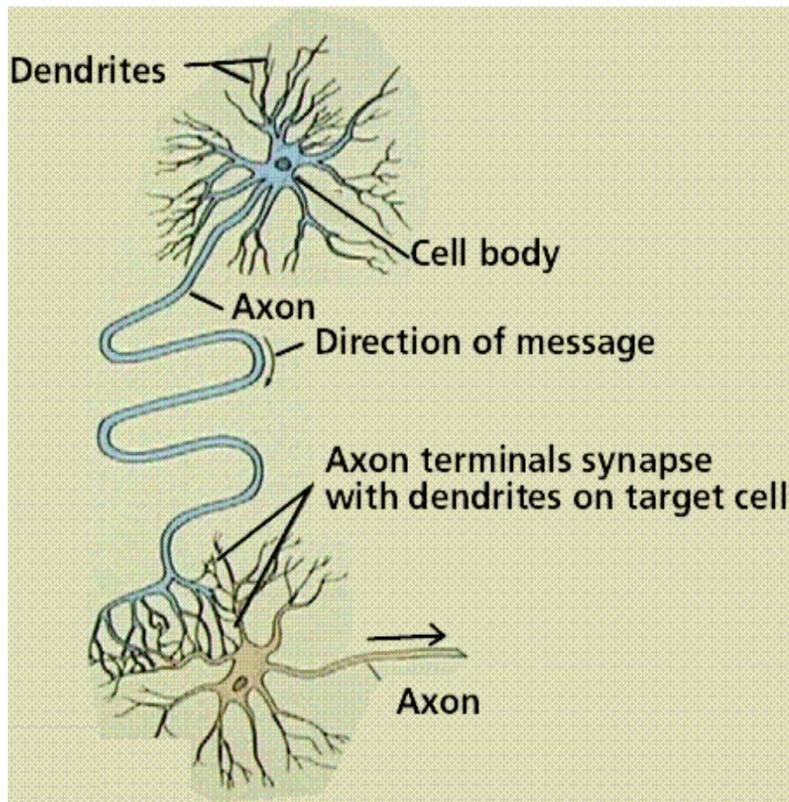
Brain metaphor



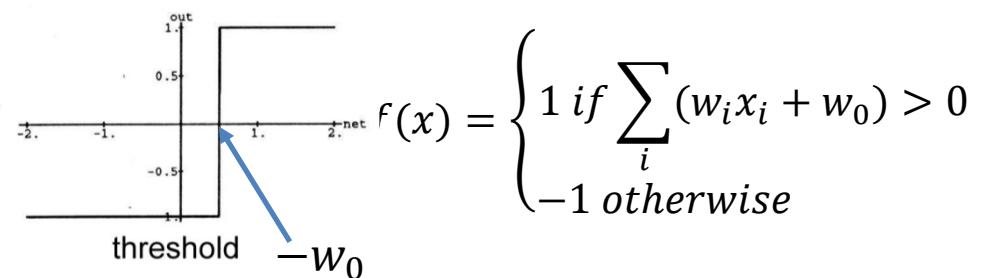
- ▶ Artificial Network Networks are an important paradigm in Statistical Machine learning and Artificial Intelligence
- ▶ Human brain is used as a source of inspiration and as a **metaphor** for developing Artificial NN
 - ▶ Human brain is a dense network 10^{11} of simple computing units, the neurons. Each neuron is connected – in mean- to 10^4 neurons.
 - ▶ Brain as a computation model
 - ▶ Distributed computations by simple processing units
 - ▶ Information and control are distributed
 - ▶ Learning is performed by observing/ analyzing huge quantities of data and also by trials and errors

Formal Model of the Neuron

McCulloch – Pitts 1943

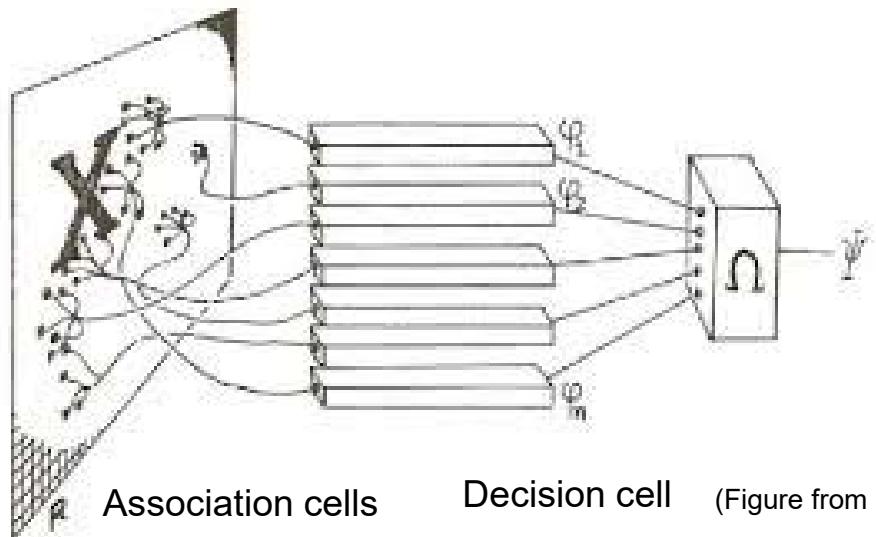


For McCulloch – Pitts
neuron, f is a threshold (sign)
function

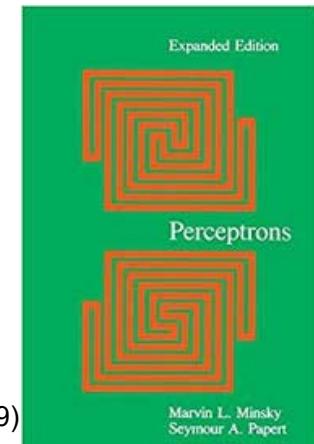


A synchronous assembly of neurons is capable of universal computations (aka equivalent to a Turing machine)

Perceptron (1958 Rosenblatt)

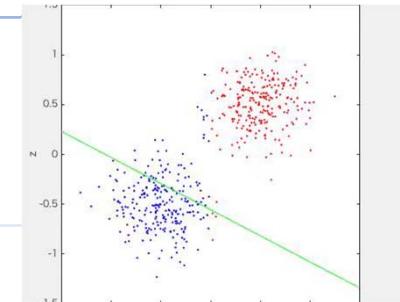


(Figure from Perceptrons, Minsky and Papert 1969)



- ▶ The decision cell is a threshold function (McCulloch – Pitts neuron)
 - ▶ $F(\mathbf{x}) = \text{sgn}(\sum_{i=1}^n w_i x_i + w_0)$
- ▶ This simple perceptron can perform 2 classes classification

Perceptron Algorithm (2 classes)



Data

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

Output

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

Initialize $w (0)$

Repeat (t)

Choose an example $(x(t), y(t))$

if $y(t)w(t) \cdot x(t) \leq 0$ then $w(t + 1) = w(t) + \epsilon y(t)x(t)$

Until convergence

Training set
Classifier specification

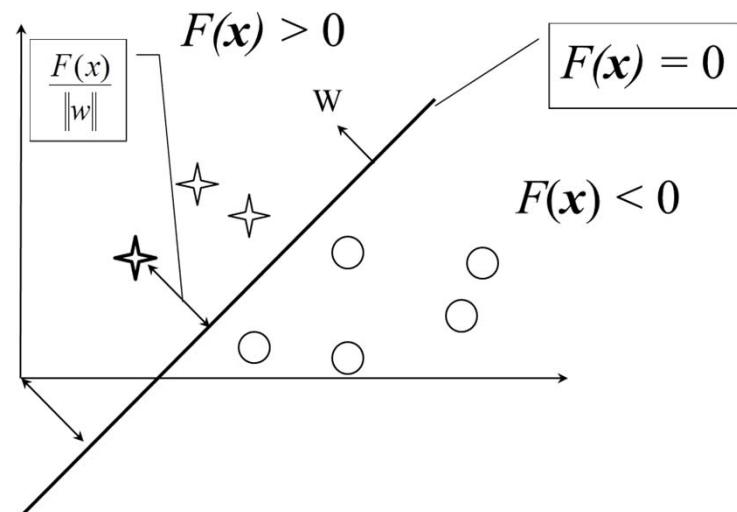
Stochastic
Algorithm

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

Linear discriminant function

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

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



Perceptron algorithm performs a stochastic gradient descent

▶ Loss function

- ▶ $C = -\sum_{(x,y) \text{ missclassified}} \mathbf{w} \cdot \mathbf{x}y = -\sum_{(x,y) \text{ miss-classified}} c(x, y)$
- ▶ Objective : minimize C

▶ gradient

- ▶ $\text{grad}_{\mathbf{w}} C = \left(\frac{\partial C}{\partial w_1}, \dots, \frac{\partial C}{\partial w_n} \right)^T$ with $\frac{\partial C}{\partial w_i} = -\sum_{(x,d) \text{ missclassified}} \mathbf{x}y$

▶ Learning rule

- ▶ Stochastic gradient descent for minimizing loss C
- ▶ Repeat (t)
 - ▶ Choose an example $(x(t), y(t))$
 - ▶ $\mathbf{w}(t) = \mathbf{w}(t-1) - \epsilon \text{grad}_{\mathbf{w}} c(\mathbf{x}, \mathbf{y})$

Multi-class generalization

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

Perceptron properties (1958 Rosenblatt)

► **Convergence theorem** (Novikof, 1962)

- ▶ Let $D = \{(x^1, y^1), \dots, (x^N, y^N)\}$ a data sample. If
 - ▶ $R = \max_{1 \leq i \leq N} \|x^i\|$
 - ▶ $\sup_w \min_i y^i(w \cdot x^i) > \rho$ (ρ is called a margin)
 - ▶ The training sequence is presented a sufficient number of time
 - ▶ The algorithm will converge after at most $\left\lceil \frac{R^2}{\rho^2} \right\rceil$ corrections

► **Generalization bound** (Aizerman, 1964)

- ▶ If in addition we provide the following stopping rule:
 - ▶ Perceptron stops if after correction number k , the next $m_k = \frac{1+2 \ln k - \ln \eta}{-\ln(1-\epsilon)}$ data are correctly recognized
- ▶ Then
 - ▶ the perceptron will converge in at most $l \leq \frac{1+4 \ln R/\rho - \ln \eta}{-\ln(1-\epsilon)} [R^2/\rho^2]$ steps
 - ▶ with probability $1 - \eta$, test error is less than ϵ

Link between training and generalization performance

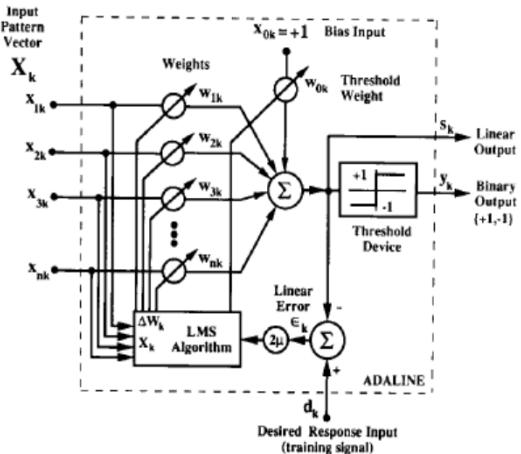
Convergence proof (Novikof)

- ▶ Hyp: let's take $w^* / \|w^*\| = 1$
 - ▶ $w_0 = 0, w_{t-1}$ is the weight vector before the t^{th} correction
 - ▶ $w_t = w_{t-1} + \epsilon y^t x^t$
 - ▶ $w_t \cdot w^* = w_{t-1} \cdot w^* + \epsilon y^t x^t \cdot w^* \geq w_{t-1} \cdot w^* + \epsilon \rho$
 - ▶ By induction $w_t \cdot w^* \geq t \epsilon \rho$

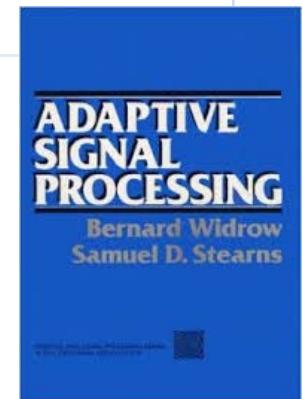
- ▶ $\|w_t\|^2 = \|w_{t-1}\|^2 + 2\epsilon y^t w_{t-1} \cdot x^t + \epsilon^2 \|x^t\|^2$
- ▶ $\|w_t\|^2 \leq \|w_{t-1}\|^2 + \epsilon^2 \|x^t\|^2$ since $y^t w_{t-1} \cdot x^t < 0$ (remember that x^t is incorrectly classified)
- ▶ $\|w_t\|^2 \leq \|w_{t-1}\|^2 + \epsilon^2 R^2$
- ▶ By induction $\|w_t\|^2 \leq t \epsilon^2 R^2$

- ▶ $t \epsilon \rho \leq w_t \cdot w^* \leq \|w_t\| \|w^*\| \leq \sqrt{t} \epsilon R \|w^*\|$
- ▶ $t \leq \frac{R^2}{\rho^2} \|w^*\|^2 = \frac{R^2}{\rho^2}$

Adaline – Adaptive Linear Element (Widrow - Hoff 1959)



$$\text{Linear unit: } F(x) = \sum_i w_i x_i + w_0$$

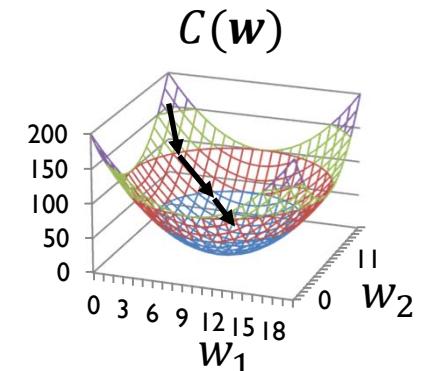


- ▶ « Least Mean Square » LMS algorithm
 - ▶ Loss: $c(x, y) = \|y - F(x)\|^2$
 - ▶ Algorithm: Stochastic Gradient Descent (Robbins – Monro (1951))

Iterate

Choose an example $(x(t), y(t))$
 $w(t + 1) = w(t) - \epsilon \nabla_w c(x, y)$

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



Adaline example motivating the need for adaptivity from an engineering perspective

► Adaptive noise cancelling

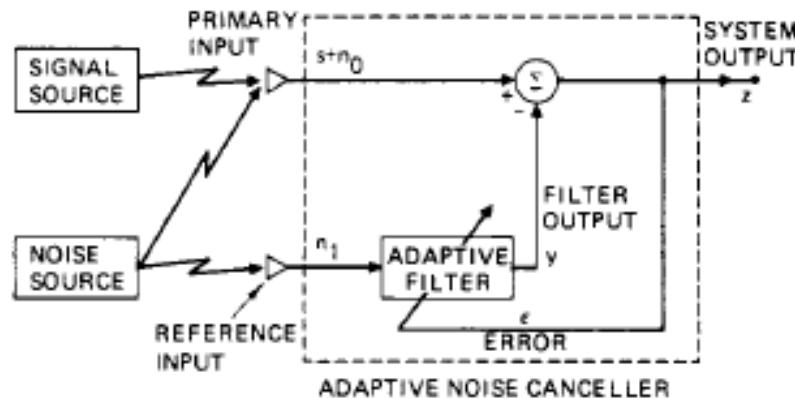


Fig. 1. The adaptive noise cancelling concept.



Widrow-Science in Action - YouTuk

Fig. from Adaptive Signal Processing, Widrow, Stearn

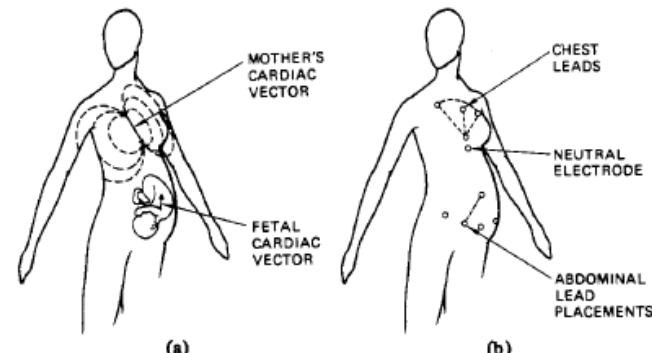


Fig. 14. Cancelling maternal heartbeat in fetal electrocardiography. (a) Cardiac electric field vectors of mother and fetus. (b) Placement of leads.

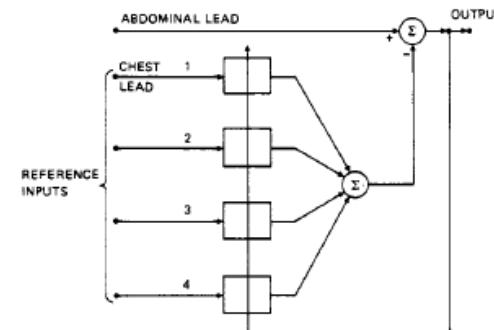


Fig. 15. Multiple-reference noise canceller used in fetal ECG experiment.

Heartbeat cancelling
Objective: get z as close as possible to the baby signal s

Adaline – heartbeat cancelling detailed

- ▶ With the notations of the Figure
- ▶ Hyp.:
 - ▶ s, n_0, n_1, y are stationary with zero means
 - ▶ s is uncorrelated with n_0, n_1 and then y
- ▶ Filtering scheme
 - ▶ output $z = s + n_0 - y$
 - ▶ Loss function to be minimized $E[z^2]$
- ▶ Then
 - ▶ $z^2 = s^2 + (n_0 - y)^2 + 2s(n_0 - y)$
 - ▶ $E[z^2] = E[s^2] + E[(n_0 - y)^2] + 2E[s(n_0 - y)]$
 - ▶ $E[z^2] = E[s^2] + E[(n_0 - y)^2]$ since s and $(n_0 - y)$ are not correlated
- ▶ So that
 - ▶ $\text{Min } E[z^2] = E[s^2] + \text{Min } E[(n_0 - y)^2]$
- ▶ When the filter is trained to minimize $E[z^2]$, it also minimizes $E[(n_0 - y)^2]$
- ▶ Then y is the best LMS estimate of n_0 , and z is the best LMS estimate of signal s (since $z - s = n_0 - y$)

Introductory concepts

Summary of key ideas

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



Optimisation : gradient methods – introduction



Optimization

Batch gradient algorithms

▶ Batch gradient general scheme

▶ Training Data Set

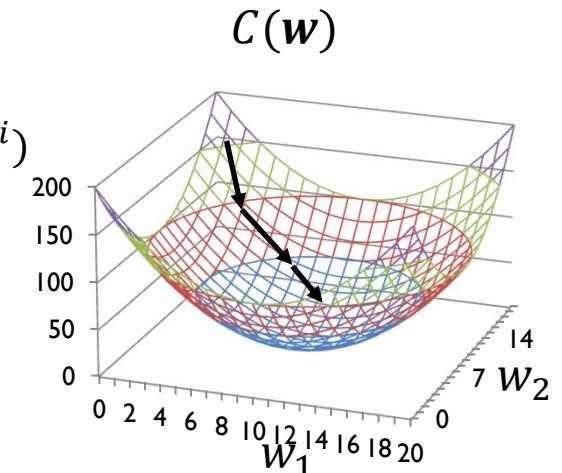
- ▶ $D = \{(x^1, y^1), \dots, (x^N, y^N)\}$

▶ Objective

- ▶ Optimize a loss function $C(\mathbf{w}) = \sum_{i=1}^N c_{\mathbf{w}}(x^i, y^i)$
 - Sum of individual losses $c_{\mathbf{w}}(\cdot, \cdot)$ on each example (x^i, y^i)

▶ Principle

- ▶ Initialize $\mathbf{w} = \mathbf{w}(0)$
- ▶ Iterate until convergence
 - $\mathbf{w}(t+1) = \mathbf{w}(t) + \epsilon(t) \Delta_{\mathbf{w}}(t)$



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

Optimization

Batch second order gradients

- ▶ Consider a quadratic approximation of the loss function

- ▶ C is approximated via a parabola

- $C(w) = C(w(t)) + (w - w(t))^T \nabla C(w(t)) + \frac{1}{2} (w - w(t))^T H (w - w(t))$

- where $w(t)$ is the parameter vector at time t

- H is the Hessian of $C(\cdot)$: $H_{ij} = \frac{\partial^2 C}{\partial w_i \partial w_j}$

- ▶ Differentiating w.r.t. w

- $\nabla C(w) = \nabla C(w(t)) + H(w - w(t))$

- ▶ The minimum of C is obtained for

- $\nabla C(w) = 0$

- ▶ Several iterative methods could be used

- ▶ E.g. Newton

- $w(t + 1) = w(t) - H^{-1} \nabla C(w(t))$

- Complexity $O(n^3)$ for the inverse + partial derivatives

- In practice one makes use of quasi-Newton methods : H^{-1} is approximated iteratively

Optimization

Stochastic Gradient algorithms

- ▶ Objectives
 - ▶ Training NNs involves finding the parameters w by optimizing a loss
- ▶ Difficulties
 - ▶ Deep NN have a large number of parameters and meta-parameters, the loss is most often a non linear function of these parameters: the optimization problem is non convex
 - ▶ Optimization for Deep NN is often difficult:
 - ▶ Multiple local minima with high loss, might not be a problem in high dimensional spaces
 - ▶ Flat regions: plateaus -> 0 gradients, saddle points -> pb for 2nd order methods
 - ▶ Sharp regions: gradients may explode
 - ▶ Deep architectures: large number of gradient multiplications may often cause gradient vanishing or gradient exploding
- ▶ Solutions
 - ▶ There is no unique answer to all these challenges
 - ▶ The most common family of optimization methods for Deep NN is based on **stochastic gradient algorithms**
 - ▶ **Exploit the redundancy in the data, at the cost of high variance in gradient estimates**
 - ▶ Deep Learning has developed several heuristic training methods
 - ▶ They are provided in the different toolboxes (Pytorch etc)
 - ▶ Some examples follow

Optimization

Stochastic gradient algorithms (From Ruder 2016)

- ▶ Data + Loss
 - ▶ Training Data Set
 - ▶ $D = \{(x^1, y^1), \dots, (x^N, y^N)\}$
 - ▶ Loss function
 - ▶ $C(w) = \sum_{i=1}^N c_w(x^i, y^i)$
 - ▶ All the algorithms are given in vector form
- ▶ Basic Stochastic Gradient Descent
 - ▶ Initialise $w(0)$
 - ▶ Iterate until stop criterion
 - ▶ sample un exemple $(x(t), y(t))$
 - ▶ $w(t + 1) = w(t) - \epsilon \nabla_w c(x(t), y(t))$
 - ▶ Rq: might produce a lot of oscillations
- ▶ Momentum
 - ▶ Dampens oscillations
 - ▶ $m(t) = \gamma m(t - 1) + \epsilon \nabla_w c(x(t), y(t))$
 - ▶ $w(t + 1) = w(t) - m(t)$



(a) SGD without momentum



(b) SGD with momentum

Figures from (Ruder 2016)

Optimization

SGD algorithms with Adaptive learning rate

▶ Adagrad

- ▶ One learning rate for each parameter w_i at each time step t

- ▶ Iteration t

- ▶ Compute gradient $\mathbf{g}(t) = \nabla_{\mathbf{w}} c(\mathbf{x}(t), \mathbf{y}(t))$ Vector

- ▶ Accumulate squared gradients for each component $r_i(t) = r_i(t-1) + (g_i(t))^2$ Scalar
- kind of gradient variance
- Sum of the squared gradients up to step t

- ▶ Componentwise:

- ▶ $w_i(t+1) = w_i(t) - \frac{\epsilon}{\sqrt{r_i(t)+\epsilon'}} \nabla_{w_i} c(\mathbf{x}(t), \mathbf{y}(t))$ Scalar

- ▶ In vector form

- ▶ $\mathbf{w}(t+1) = \mathbf{w}(t) - \frac{\epsilon}{\sqrt{r(t)+\epsilon'}} \odot \nabla_{\mathbf{w}} c(\mathbf{x}(t), \mathbf{y}(t))$ Vector

- ▶ \odot elementwise multiplication, $\epsilon' (\approx 10^{-8})$ avoids dividing by 0, $\frac{\epsilon}{\sqrt{r(t)+\epsilon'}}$ is a vector with components $\frac{\epsilon}{\sqrt{r_i(t)+\epsilon'}}$

- ▶ Default : learning rate shrinks too fast

▶ RMS prop

- ▶ Replace $r(t)$ in Adagrad by an exponentially decaying average of past gradients

- ▶ $\mathbf{r}(t) = \gamma \mathbf{r}(t-1) + (1-\gamma) \mathbf{g}(t) \odot \mathbf{g}(t), \quad 0 < \gamma < 1$

- ▶ $\mathbf{w}(t+1) = \mathbf{w}(t) - \frac{\epsilon}{\sqrt{\mathbf{r}(t)+\epsilon'}} \odot \nabla_{\mathbf{w}} c(\mathbf{x}(t), \mathbf{y}(t))$ Vector

Optimization

SGD algorithm with momentum and Adaptive learning rate

▶ Adam (adaptive moment estimation)

▶ Computes

- ▶ Adaptive learning rates for each parameter
- ▶ An exponentially decaying average of past gradients (momentum)
- ▶ An exponentially decaying average of past squared gradients (like RMSprop)

▶ Iteration t

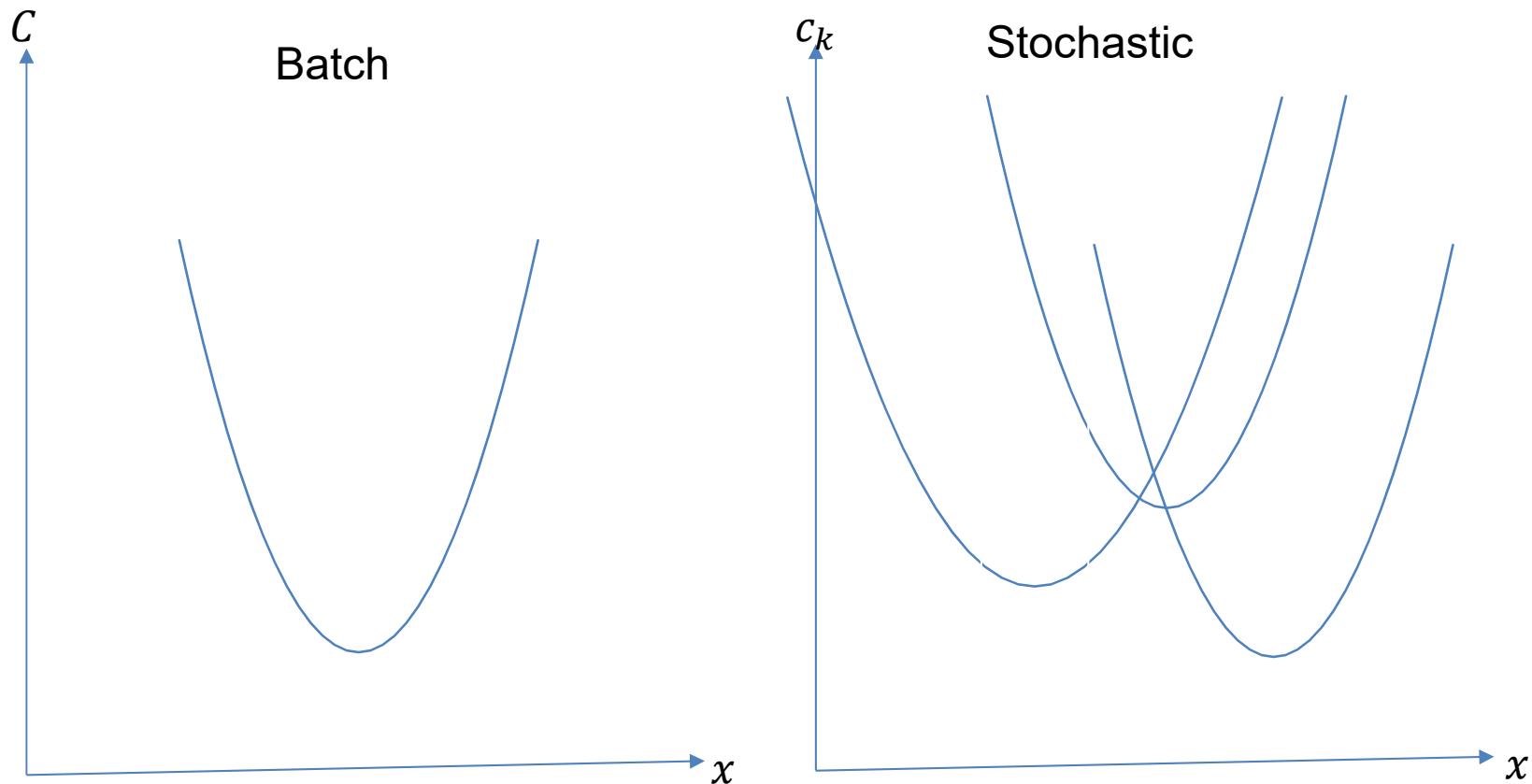
- ▶ Momentum term : $\mathbf{m}(t) = \gamma_1 \mathbf{m}(t - 1) + \epsilon(1 - \gamma_1) \mathbf{g}(t)$
- ▶ Gradient variance term: $\mathbf{r}(t) = \gamma_2 \mathbf{r}(t - 1) + \epsilon(1 - \gamma_2) \mathbf{g}(t) \odot \mathbf{g}(t)$
- ▶ $\mathbf{w}(t + 1) = \mathbf{w}(t) - \frac{\epsilon}{\sqrt{\mathbf{r}(t) + \epsilon'}} \odot \mathbf{m}(t)$
- ▶ Bias correction
 - The 2 moments are initialized at 0, they tend to be biased towards 0, the following correction terms reduce this effect
 - Correct bias of \mathbf{m} : $\mathbf{m}(t) = \frac{\mathbf{m}(t)}{1 - \gamma_1^t}$
 - Correct bias of \mathbf{r} : $\mathbf{r}(t) = \frac{\mathbf{r}(t)}{1 - \gamma_2^t}$

Batch vs stochastic gradient



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

C : global loss
 c_k : individual (pattern k) loss



Optimization Summary

- ▶ Which method to use?
 - ▶ No « one solution for all problems »
 - ▶ For large scale applications, Adam is often used today as a default choice together with minibatches
 - ▶ But... simple SGD with heuristic learning rate decay can sometimes be competitive ...
- ▶ Batch, mini batch, pure SGD
 - ▶ Stochastic methods exploit data redundancy
 - ▶ Mini batch well suited for GPU
 - ▶



Regression and Logistic Regression

Regression

▶ Linear regression

▶ Objective : predict real values

▶ Training set

▶ $(\mathbf{x}^1, y^1), \dots, (\mathbf{x}^N, y^N)$

▶ $\mathbf{x} \in R^n, y \in R$: single output regression

▶ Linear model

▶ $F(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^n w_i x_i$ with $x_0 = 1$

▶ Loss function

▶ Mean square error

$$\square C = \frac{1}{2} \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i)^2$$

▶ Steepest descent gradient (batch)

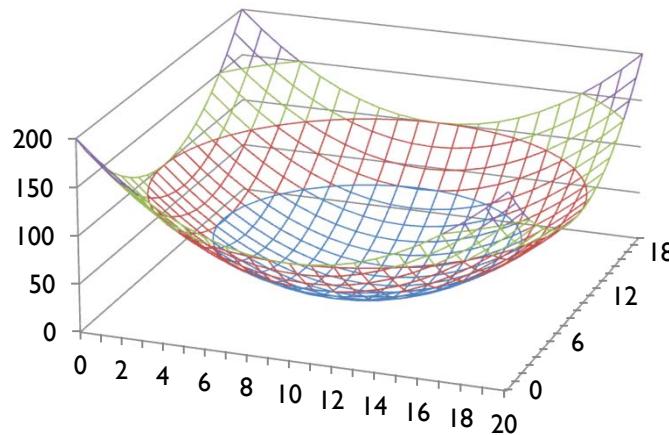
▶ $\mathbf{w} = \mathbf{w}(t) - \epsilon \nabla_{\mathbf{w}} C, \nabla_{\mathbf{w}} C = \left(\frac{\partial C}{\partial w_1}, \dots, \frac{\partial C}{\partial w_n} \right)^T$

▶ $\frac{\partial C}{\partial w_k} = \frac{1}{2} \sum_{i=1}^N \frac{\partial}{\partial w_k} (y^i - \mathbf{w} \cdot \mathbf{x}^i)^2 = - \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i) x_k^i$ for component w_k

▶ $\mathbf{w} = \mathbf{w}(t) + \epsilon \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i) \mathbf{x}^i$ in vector form

Regression

► Geometry of mean squares



- Regression with multiple outputs $\mathbf{y} \in R^p$
 - Simple extension: p independent linear regressions

Probabilistic Interpretation

- ▶ Statistical model of linear regression
 - ▶ $y = \mathbf{w} \cdot \mathbf{x} + \epsilon$, where ϵ is a random variable (error term)
 - ▶ Hypothesis ϵ is i.i.d. Gaussian
 - ▶ $\epsilon \sim N(0, \sigma^2)$, $p(\epsilon) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{\epsilon^2}{2\sigma^2})$
 - ▶ The posterior distribution of y is then
 - ▶ $p(y | \mathbf{x}; \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y - \mathbf{w} \cdot \mathbf{x})^2}{2\sigma^2})$
 - ▶ Likelihood
 - ▶ $L(\mathbf{w}) = \prod_{i=1}^N p(y^i | \mathbf{x}^i; \mathbf{w})$
 - Likelihood is a function of \mathbf{w} , it is computed on the training set
 - ▶ Maximum likelihood principle
 - ▶ Choose the parameters \mathbf{w} maximizing $L(\mathbf{w})$ or any increasing function of $L(\mathbf{w})$
 - ▶ In practice, one optimizes the log likelihood $l(\mathbf{w}) = \log L(\mathbf{w})$
 - ▶
$$l(\mathbf{w}) = N \log \left(\frac{1}{\sqrt{2\pi}\sigma} \right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i)^2$$
 - ▶ This is the MSE criterion
- ▶ This provides a probabilistic interpretation of regression
 - ▶ Under a gaussian hypothesis max likelihood is equivalent to MSE minimization

Logistic regression

- ▶ Linear regression can be used (in practice) for regression or classification
- ▶ For classification a proper model is logistic regression

- ▶ $F_w(x) = g(w \cdot x) = \frac{1}{1+\exp(-w \cdot x)}$

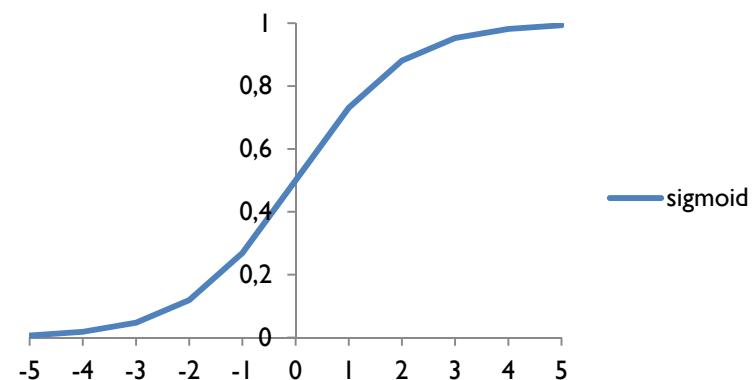
- ▶ Logistic (or sigmoid) function

- ▶ $g(z) = \frac{1}{1+\exp(-z)}$

- hint

- $g'(z) = g(z)(1 - g(z))$

- ▶ Hyp: $y \in \{0,1\}$



Logistic regression

Probabilistic interpretation

- ▶ Since $y \in \{0,1\}$, we make a Bernoulli hypothesis for the posterior distribution

- ▶ $p(y = 1|\mathbf{x}; \mathbf{w}) = F_w(\mathbf{x})$ et $p(y = 0|\mathbf{x}; \mathbf{w}) = 1 - F_w(\mathbf{x})$
- ▶ In compact format
 - $p(y|\mathbf{x}; \mathbf{w}) = (F_w(\mathbf{x}))^y (1 - F_w(\mathbf{x}))^{1-y}$ with $y \in \{0,1\}$

- ▶ Likelihood

- ▶ $L(\mathbf{w}) = \prod_{i=1}^N (F_w(\mathbf{x}^i))^{y^i} (1 - F_w(\mathbf{x}^i))^{1-y^i}$

- ▶ Log-likelihood

- ▶ $l(\mathbf{w}) = \sum_{i=1}^N y^i \log F_w(\mathbf{x}^i) + (1 - y^i) \log(1 - F_w(\mathbf{x}^i))$
 - This is minus the cross-entropy between the target and the estimated posterior distribution

- ▶ Steepest descent algorithm (batch) for minimizing cross entropy

- ▶ Componentwise:
$$\frac{\partial l(\mathbf{w})}{\partial w_k} = \sum_{i=1}^N (y^i - F_w(\mathbf{x}^i)) \mathbf{x}_k^i$$
- ▶ Vector form:
$$\nabla_{\mathbf{w}} l = \sum_{i=1}^N (y^i - F_w(\mathbf{x}^i)) \mathbf{x}^i$$
- ▶ Algorithm
 - $\mathbf{w} = \mathbf{w} - \epsilon \nabla_{\mathbf{w}} C = \mathbf{w} + \epsilon \sum_{i=1}^N (y^i - F_w(\mathbf{x}^i)) \mathbf{x}^i$

Multivariate logistic regression

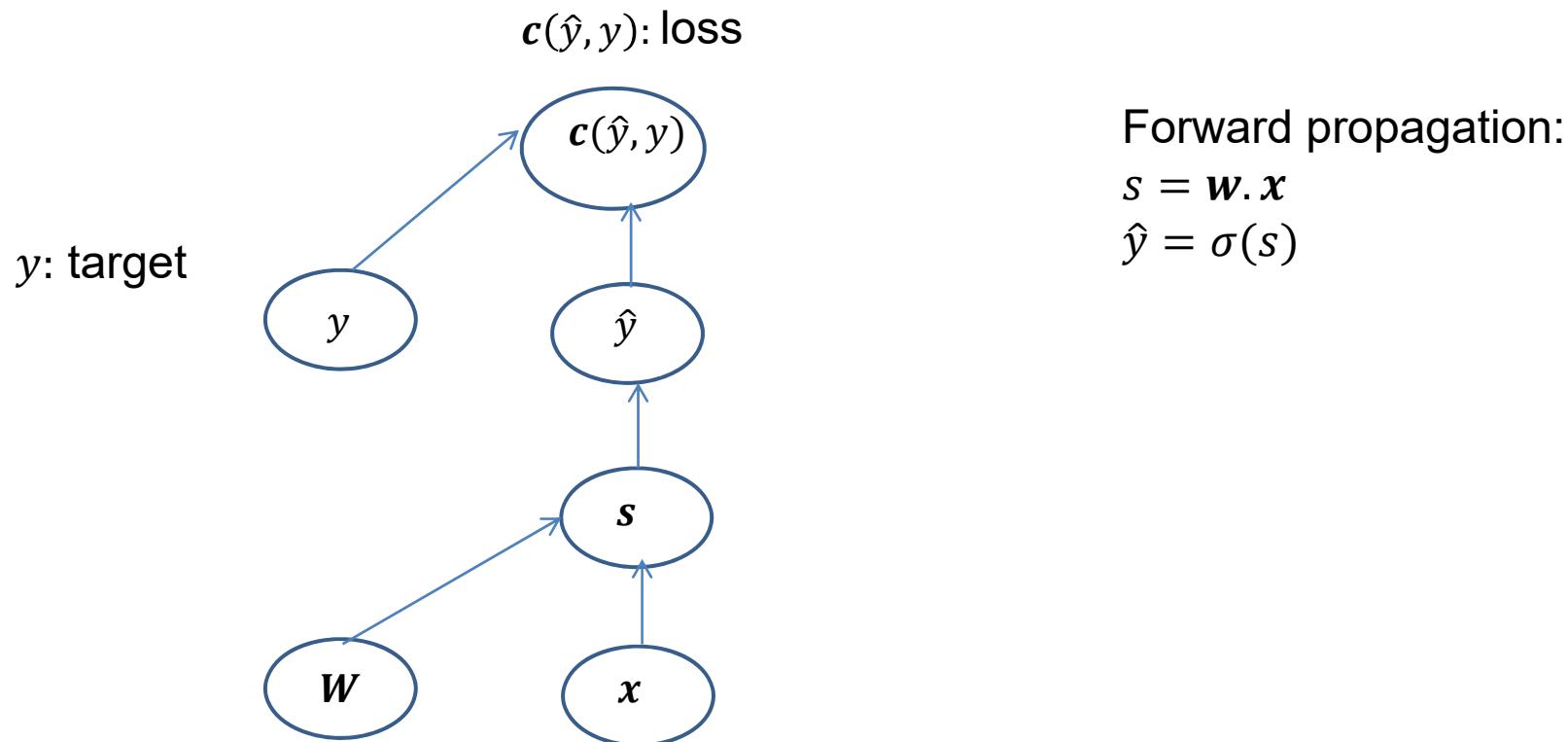
- ▶ Consider a p class classification problem
- ▶ Classes are encoded by “one hot” indicator vectors. Each vector is of dimension p
 - ▶ Class 1: $y = (1, 0, \dots, 0)^T$
 - ▶ Class 2 : $y = (0, 1, \dots, 0)^T$
 - ▶ ...
 - ▶ Class p : $y = (0, 0, \dots, 1)^T$
- ▶ $F_{\mathbf{W}}(\mathbf{x})$ is a vector valued function with values in R^p
 - ▶ Its component i is a **softmax function** (generalizes the sigmoid)
 - ▶ $F_{\mathbf{W}}(\mathbf{x})_i = \frac{\exp(\mathbf{w}_i \cdot \mathbf{x})}{\sum_{j=1}^p \exp(\mathbf{w}_j \cdot \mathbf{x})}$
 - Note : here $\mathbf{w}_j \in R^n$ is a vector
- ▶ The probabilistic model for the posterior is a multinomial distribution
 - ▶ $p(\text{Class} = i | \mathbf{x}; \mathbf{w}) = \frac{\exp(\mathbf{w}_i \cdot \mathbf{x})}{\sum_{j=1}^p \exp(\mathbf{w}_j \cdot \mathbf{x})}$ (softmax)
- ▶ Training algorithm
 - ▶ As before, one may use a gradient method for maximizing the log likelihood.
 - ▶ When the number of classes is large, computing the soft max is prohibitive, alternatives are required

Probabilistic interpretation for non linear models

- ▶ These results extend to non linear models, e.g. when $F_w(x)$ is a NN
- ▶ Non linear regression
 - ▶ Max likelihood is equivalent to MSE loss optimization under the Gaussian hyp.
 - ▶ $y = F_w(x) + \epsilon, \epsilon \sim N(0, \sigma^2)$
 - ▶ $p(y | \mathbf{x}; \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y - F(\mathbf{x}))^2}{2\sigma^2}\right)$
 - ▶ log – likelihood $l(w)$
 - ▶
$$l(\mathbf{w}) = N \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (y^i - F(\mathbf{x}^i))^2$$
- ▶ Classification
 - ▶ Max likelihood is equivalent to cross entropy maximization under Bernoulli/multinomial distribution
 - 2 classes: if y is binary and we make the hypothesis that it is conditionnally Bernoulli with probability $F(x) = p(y = 1|x)$ we get the cross entropy loss
 - More than 2 classes: same as logistic regression with multiple outputs
 - XXmultinoulli distribution ?

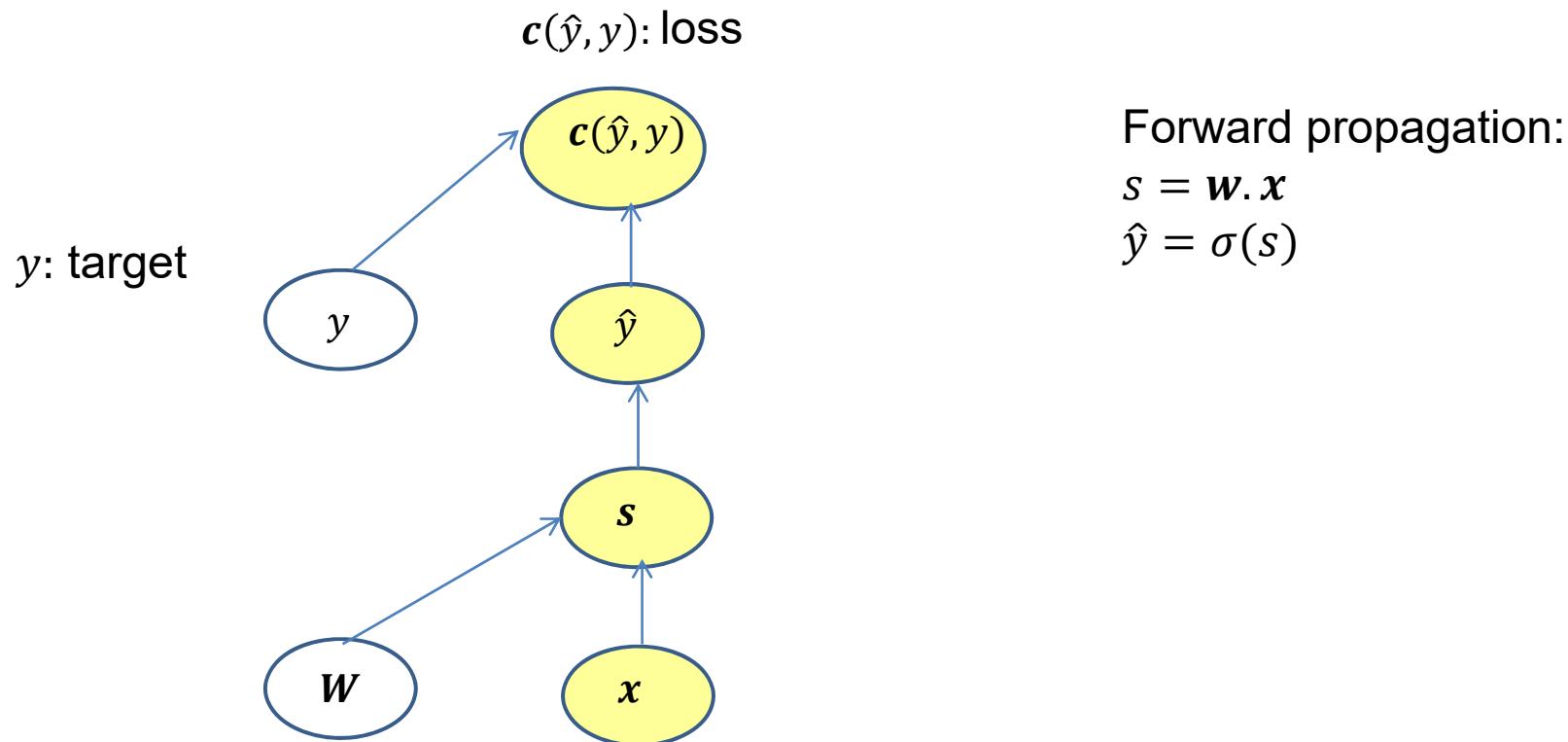
Logistic regression – Computational graph -SGD

▶ Forward pass



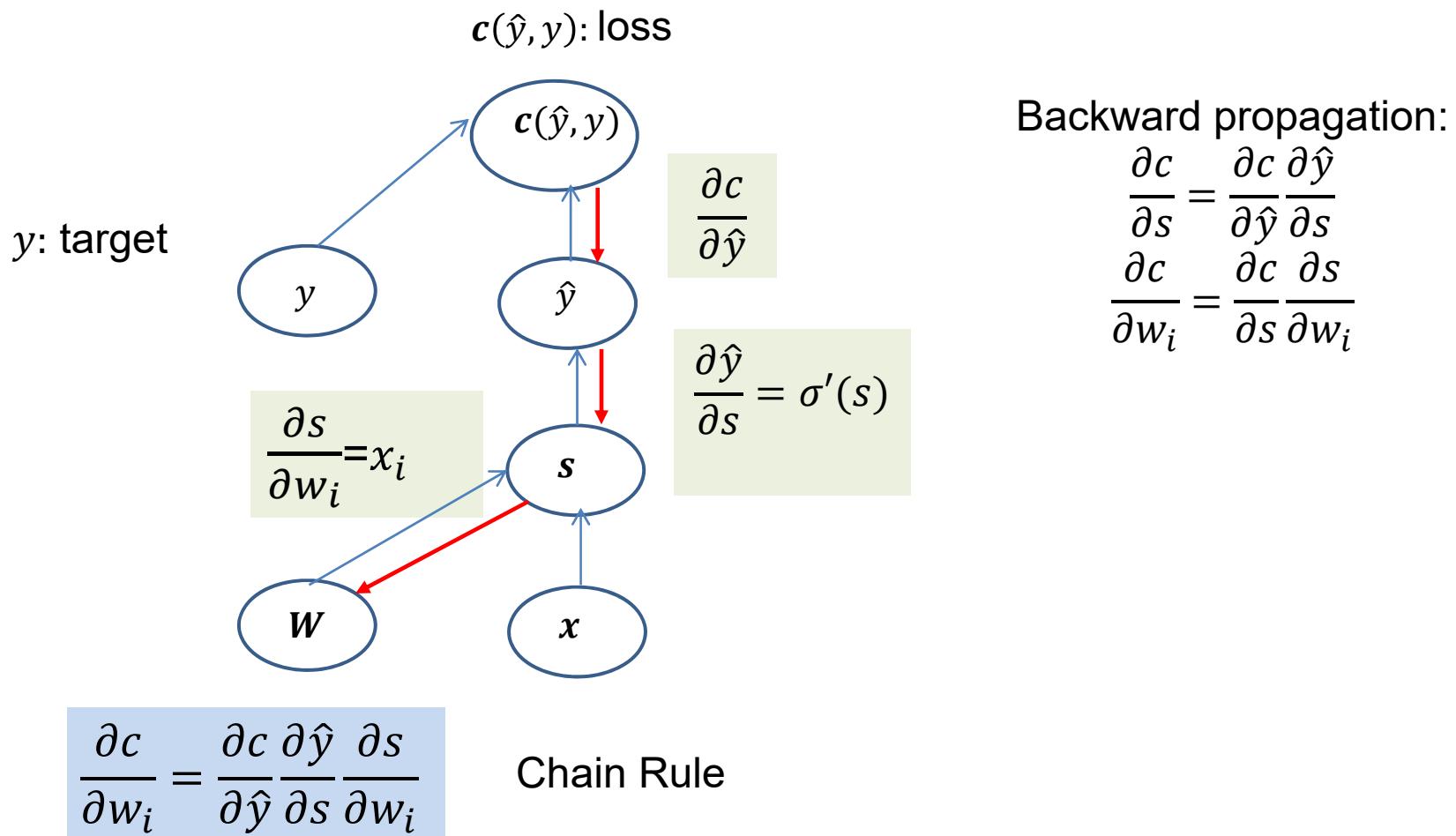
Logistic regression – Computational graph - SGD

▶ Forward pass



Logistic regression – Computational graph - SGD

▶ Backward pass



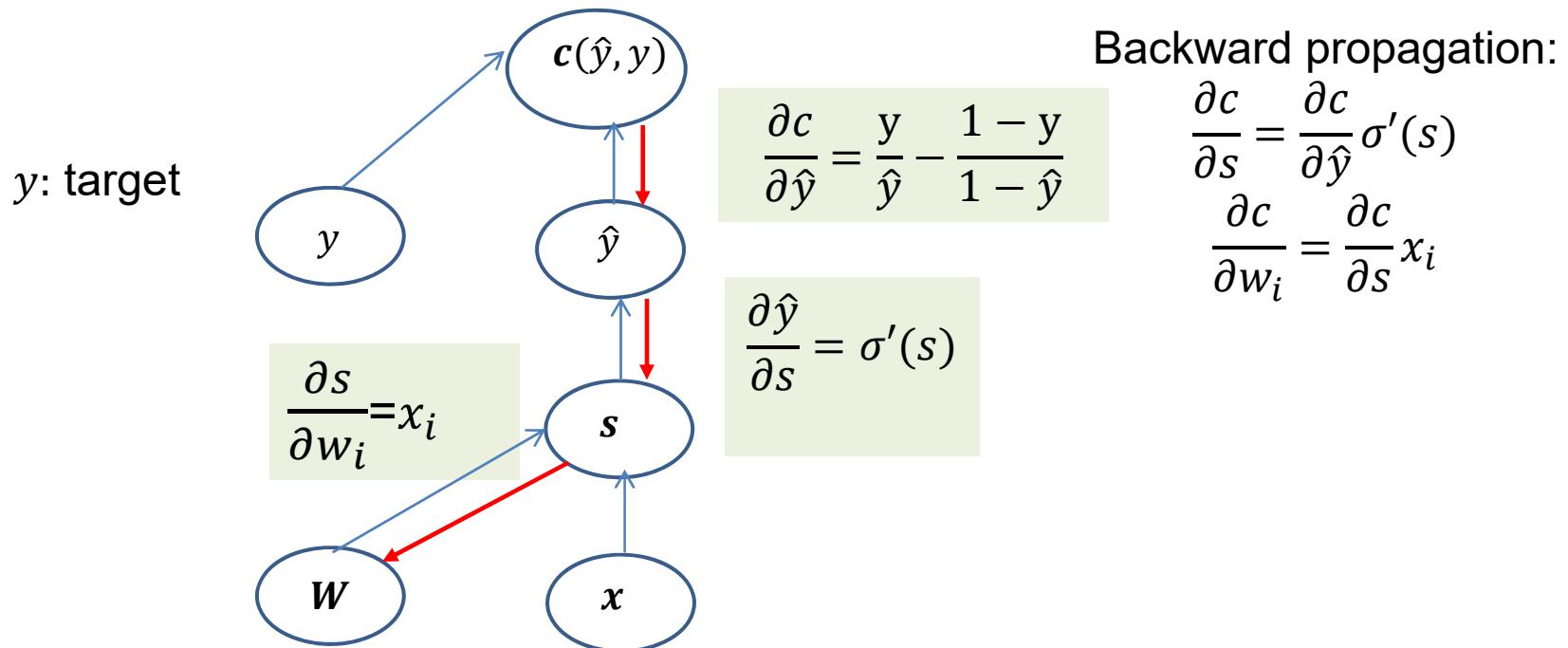
Logistic regression – Computational graph - SGD

▶ Backward pass

For the cross entropy loss

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

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



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

Probabilistic interpretation of NN outputs

Mean Square loss

- ▶ Derived here for multivariate regression (1 output), trivial extension to multiple outputs
- ▶ Holds for any continuous functional (regression, logistic regression, NNs, etc)
- ▶ Risk $R = E_{x,y} [(y - h(x))^2]$
- ▶ The minimum of R , $\text{Min}_h R$, is obtained for $h^*(x) = E_y[y|x]$
- ▶ The risk R pour the family of functions $F_w(x)$ decomposes as follows:
 - ▶ $R = E_{x,y} [(y - F_w(x))^2]$
 - ▶ $R = E_{x,y} [(y - E_y[y|x])^2] + E_{x,y} [(E_y[y|x] - F_w(x))^2]$
- ▶ Let us consider $E_y [(y - E_y[y|x])^2]$
 - ▶ This term is independent of the model $F_w(\cdot)$ and only depends on the problem characteristics (the data distribution).
 - ▶ It represents the min error that could be obtained for this data distribution
 - ▶ $h^*(x) = E_y[y|x]$ est the optimal solution to $\text{Min}_h R$
- ▶ Minimizing $E_{x,y} [(y - F_w(x))^2]$ is equivalent to minimizing $E_{x,y} [(E_y[y|x] - F_w(x))^2]$
 - ▶ The optimal solution $F_{w^*}(x) = \text{argmin}_w E_{x,y} [(E_y[y|x] - F_w(x))^2]$ is the best mean square approximation of $E[y|x]$

Probabilistic interpretation of NN outputs

▶ Classification

- ▶ Let us consider multi-class classification with one hot encoding of the target outputs
 - ▶ i.e. $\mathbf{y} = (0, \dots, 0, 1, 0, \dots, 0)^T$ with a 1 at position i if the target is class i and zero everywhere else
 - ▶ $h_i^* = E_y[y|x] = 1 * P(C_i|x) + 0 * (1 - P(C_i|x)) = P(C_i|x)$
 - ▶ i.e. $F_{w^*}()$ is the best LMS approximation of the Bayes discriminant function (which is the optimal solution for classification with 0/1 loss)
- ▶ More generally with binary targets
 - ▶ $h_i^* = P(y_i = 1|x)$

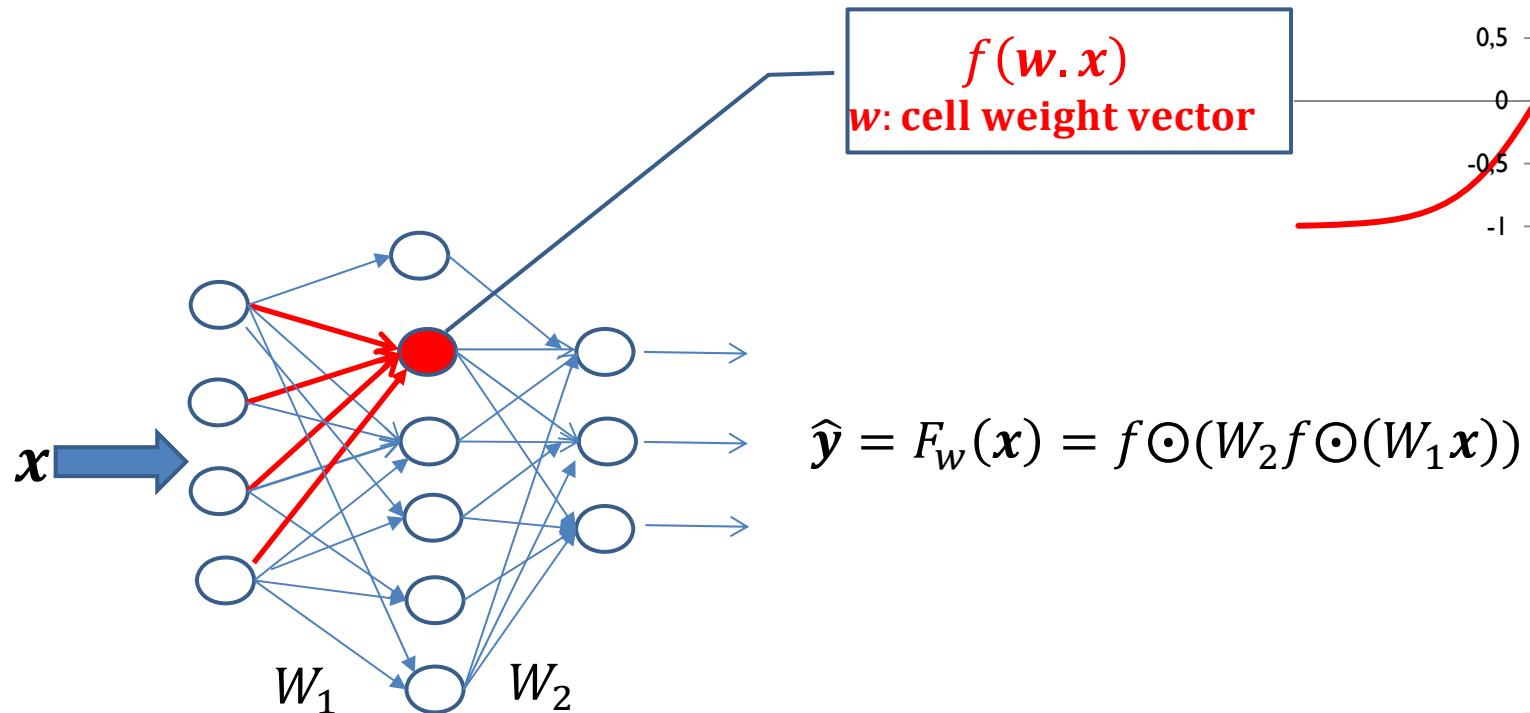
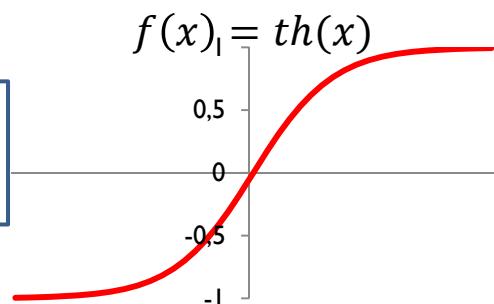
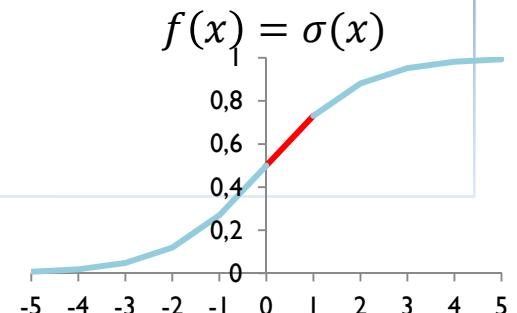
▶ Note

- ▶ Similar results hold for the cross entropy criterion
- ▶ Precision on the computed outputs depends on the task
 - ▶ Classification: precision might not be so important (max decision rule, one wants the correct class to be ranked above all others)
 - ▶ Posterior probability estimation: precision is important

Multi-layer Perceptron

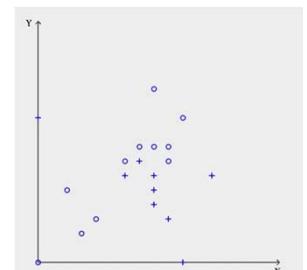
Multi-layer Perceptron (Hinton – Sejnowski – Williams 1986)

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



<http://playground.tensorflow.org/>

Note: \odot is a pointwise operator, if $x = (x_1, x_2)$, $f \odot ((x_1, x_2)) = (f(x_1), f(x_2))$
 Machine Learning & Deep Learning - P. Gallinari



Multi-layer Perceptron - Training

▶ **Stochastic Gradient Descent** - The algorithm is called **Back-Propagation**

- ▶ Pick one example (x, y) or a **Mini Batch** $\{(x^i, y^i)\}$ sampled from the training set
 - ▶ Here the algorithm is described for 1 example and for the sigmoid $\sigma()$ non linearity
- ▶ Forward pass
 - $\hat{y} = F_w(x) = f \odot (W_2 f \odot (W_1 x))$
- ▶ Compute error
 - $c(y, \hat{y})$, e.g. mean square error or cross entropy
- ▶ Backward pass
 - ▶ efficient implementation of chain rule
 - ▶ $w_{ij} = w_{ij} - \epsilon \frac{\partial c(y, \hat{y})}{\partial w_{ij}}$

Note: \odot is a pointwise operator, if $x = (x_1, x_2)$, $f \odot ((x_1, x_2)) = (f(x_1), f(x_2))$

Algorithmic differentiation

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

Notations – matrix derivatives

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}, \alpha \in R, W: p \times q$$

Vector by scalar

$$\frac{\partial x}{\partial \alpha} = \begin{pmatrix} \frac{\partial x_1}{\partial \alpha} \\ \vdots \\ \frac{\partial x_n}{\partial \alpha} \end{pmatrix}$$

Scalar by vector

$$\frac{\partial \alpha}{\partial x} = \left(\frac{\partial \alpha}{\partial x_1}, \dots, \frac{\partial \alpha}{\partial x_n} \right)$$

Vector by vector

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Matrix by scalar

$$\frac{\partial W}{\partial \alpha} = \begin{pmatrix} \frac{\partial w_{11}}{\partial \alpha} & \dots & \frac{\partial w_{1q}}{\partial \alpha} \\ \vdots & \ddots & \vdots \\ \frac{\partial w_{p1}}{\partial \alpha} & \dots & \frac{\partial w_{pq}}{\partial \alpha} \end{pmatrix}$$

Scalar by matrix

$$\frac{\partial \alpha}{\partial W} = \begin{pmatrix} \frac{\partial \alpha}{\partial w_{11}} & \dots & \frac{\partial \alpha}{\partial w_{p1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \alpha}{\partial w_{1q}} & \dots & \frac{\partial \alpha}{\partial w_{pq}} \end{pmatrix}$$

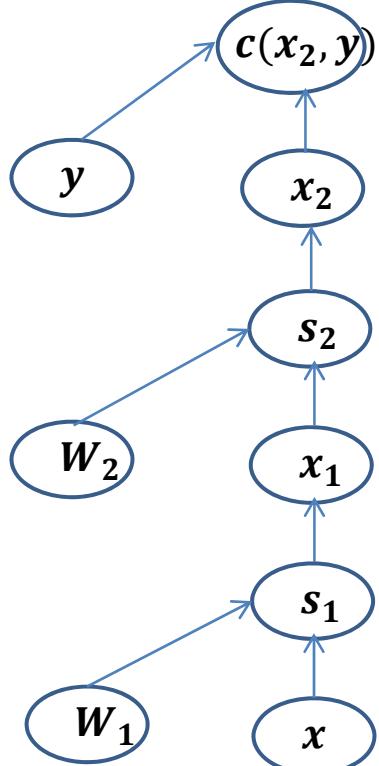
Multi-layer Perceptron - Training

▶ Computational graph

$c(x_2, y)$: loss

Here, $x_2 = \hat{y}$

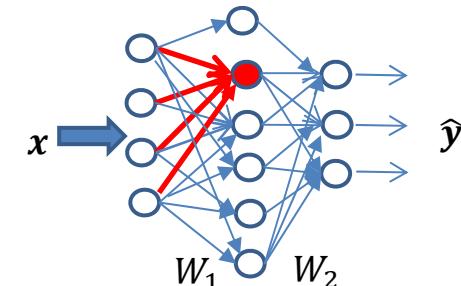
y : target



Forward propagation:

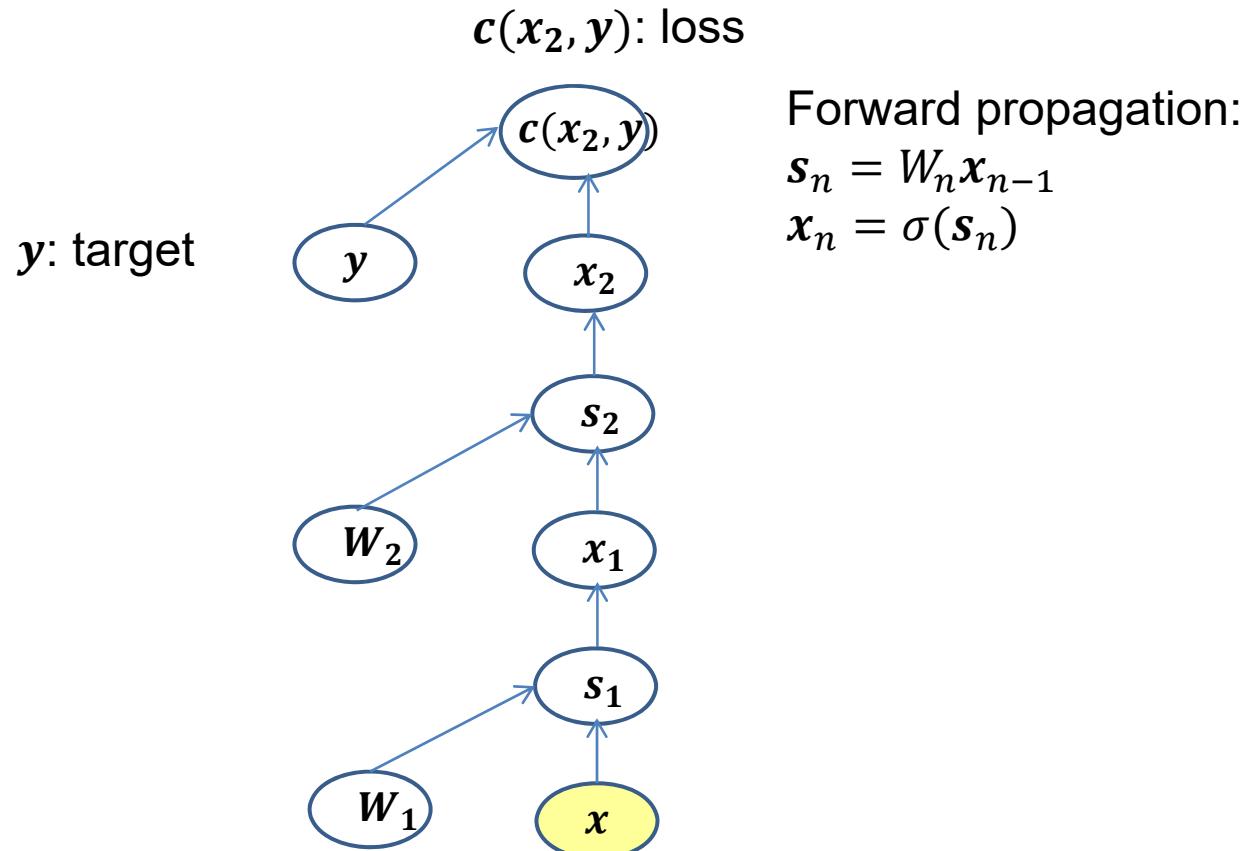
$$s_n = W_n x_{n-1}$$

$$x_n = \sigma(s_n)$$



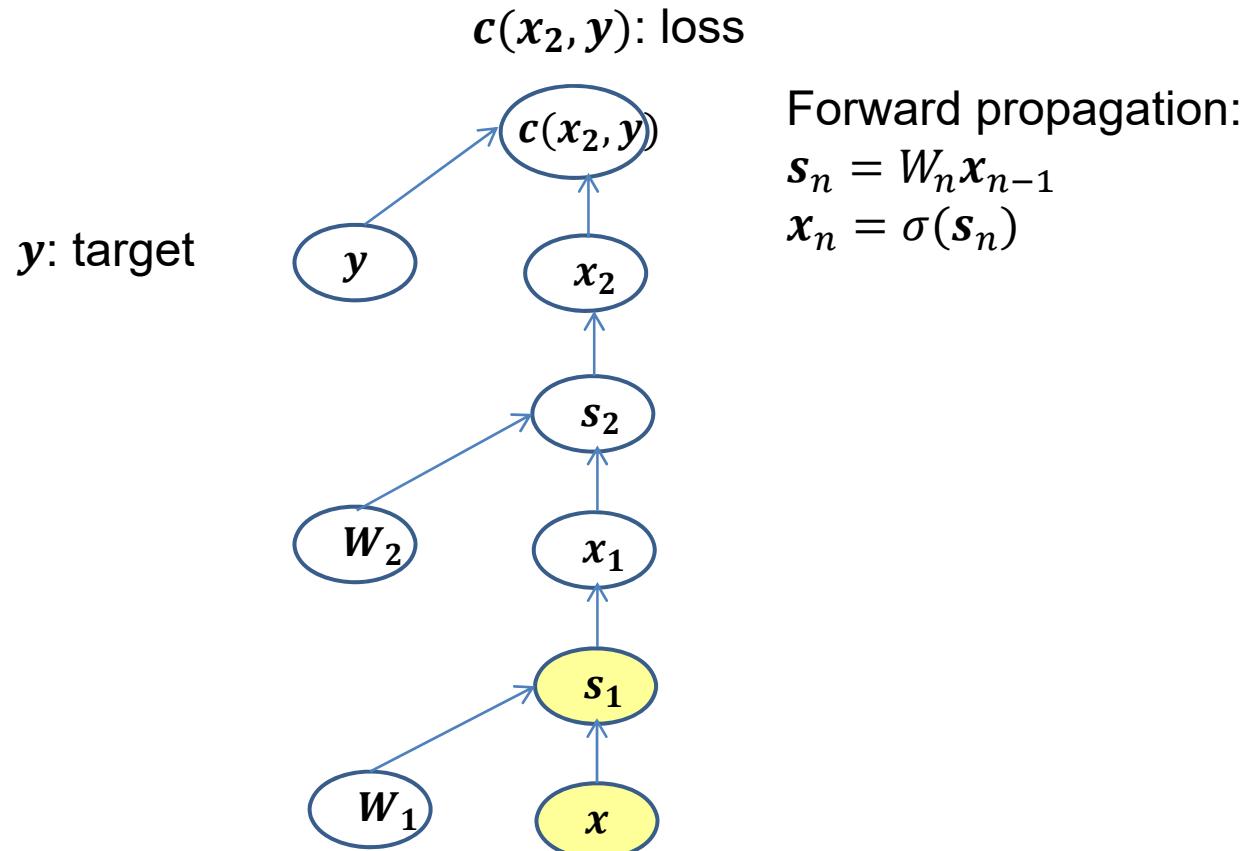
Multi-layer Perceptron - Training

▶ Forward pass



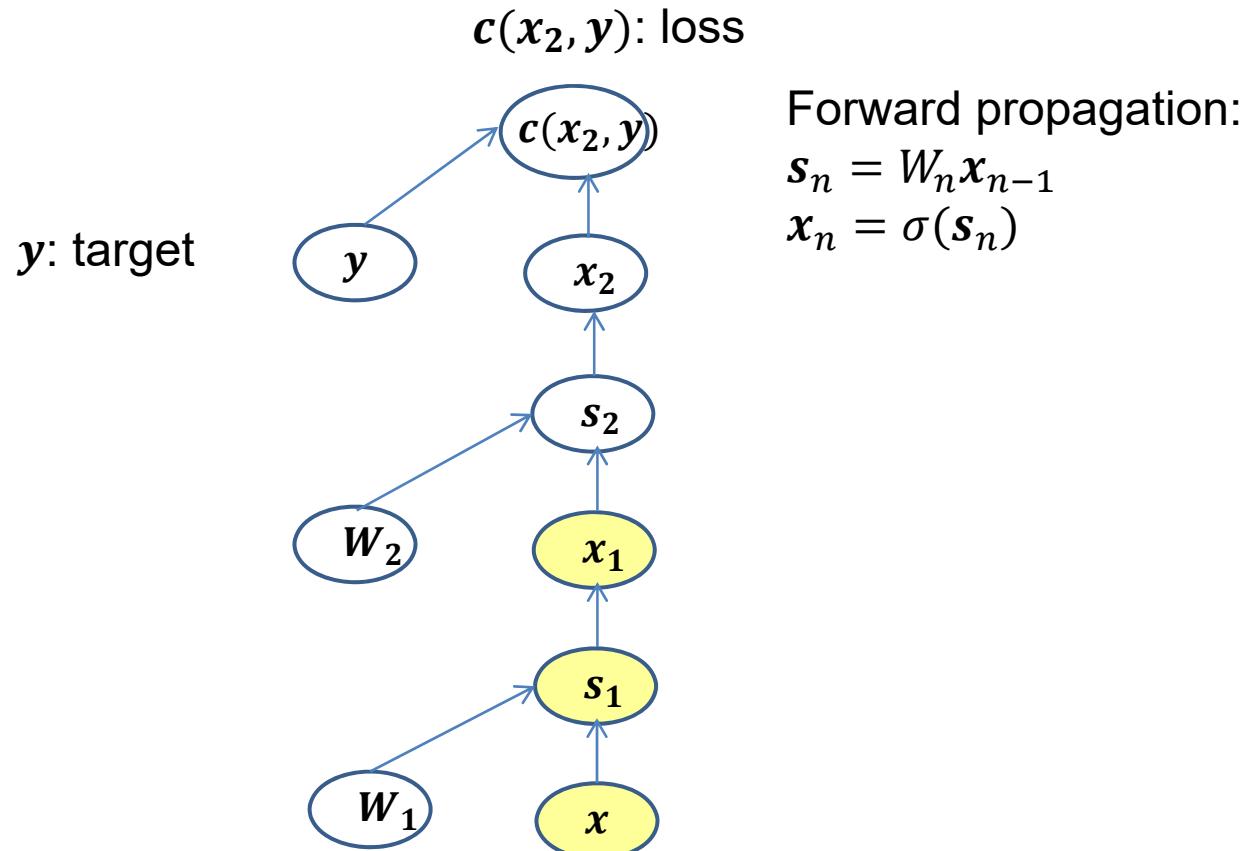
Multi-layer Perceptron - Training

▶ Forward pass



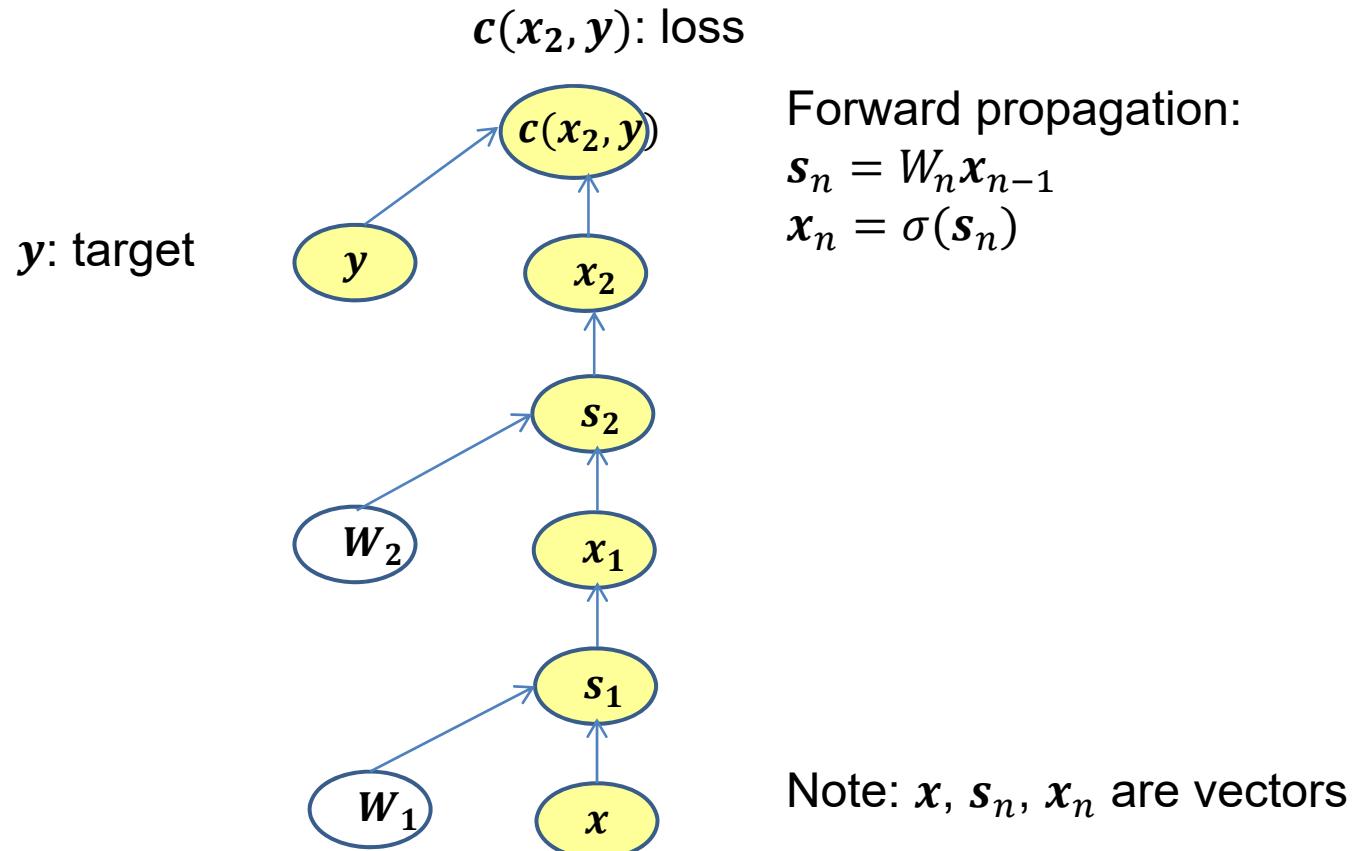
Multi-layer Perceptron - Training

▶ Forward pass



Multi-layer Perceptron - Training

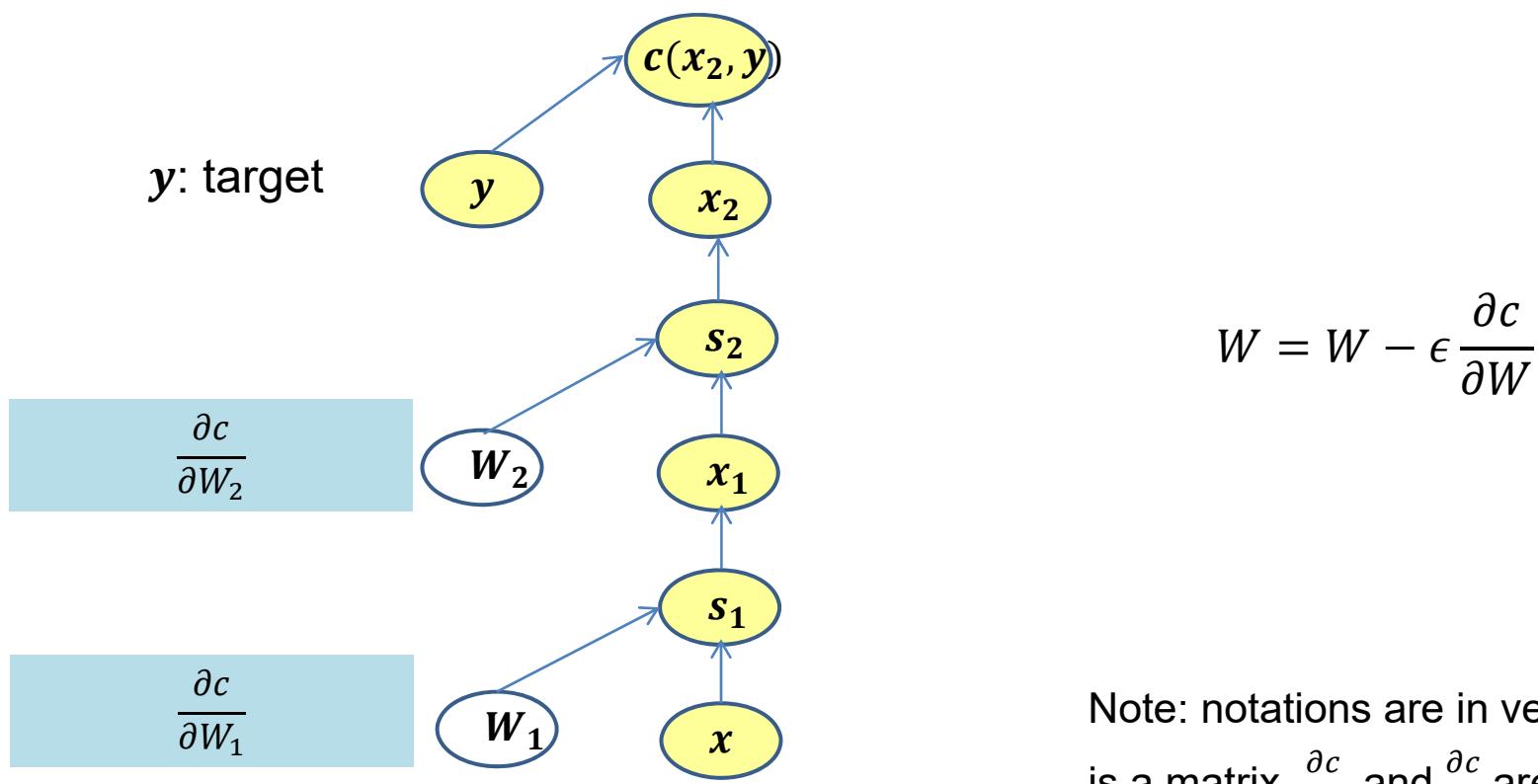
▶ Forward pass



Multi-layer Perceptron - Training

- ▶ Back Propagation: Reverse Mode Differentiation

$c(x_2, y)$: loss

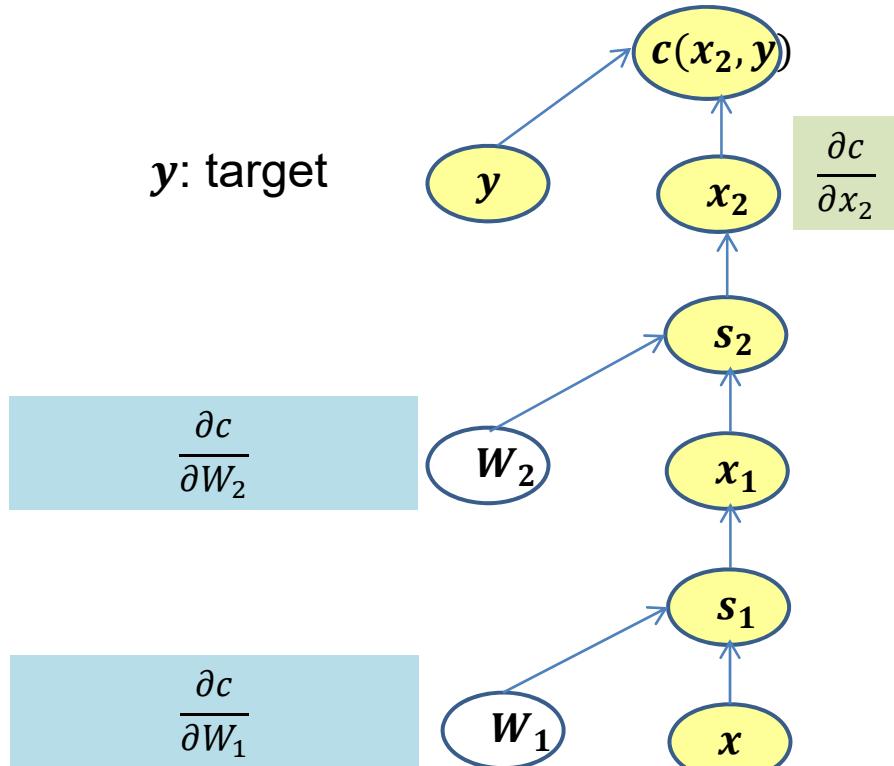


Note: notations are in vector form, $\frac{\partial c}{\partial w}$ is a matrix, $\frac{\partial c}{\partial x}$ and $\frac{\partial c}{\partial s}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- ▶ Back propagation: Reverse Mode Differentiation

$c(x_2, y)$: loss



Backward propagation:

$$\frac{\partial c}{\partial s_n} = \frac{\partial c}{\partial x_n} \odot \sigma'(s_n)^T$$

$$\frac{\partial c}{\partial W_n} = x_{n-1} \frac{\partial c}{\partial s_n}$$

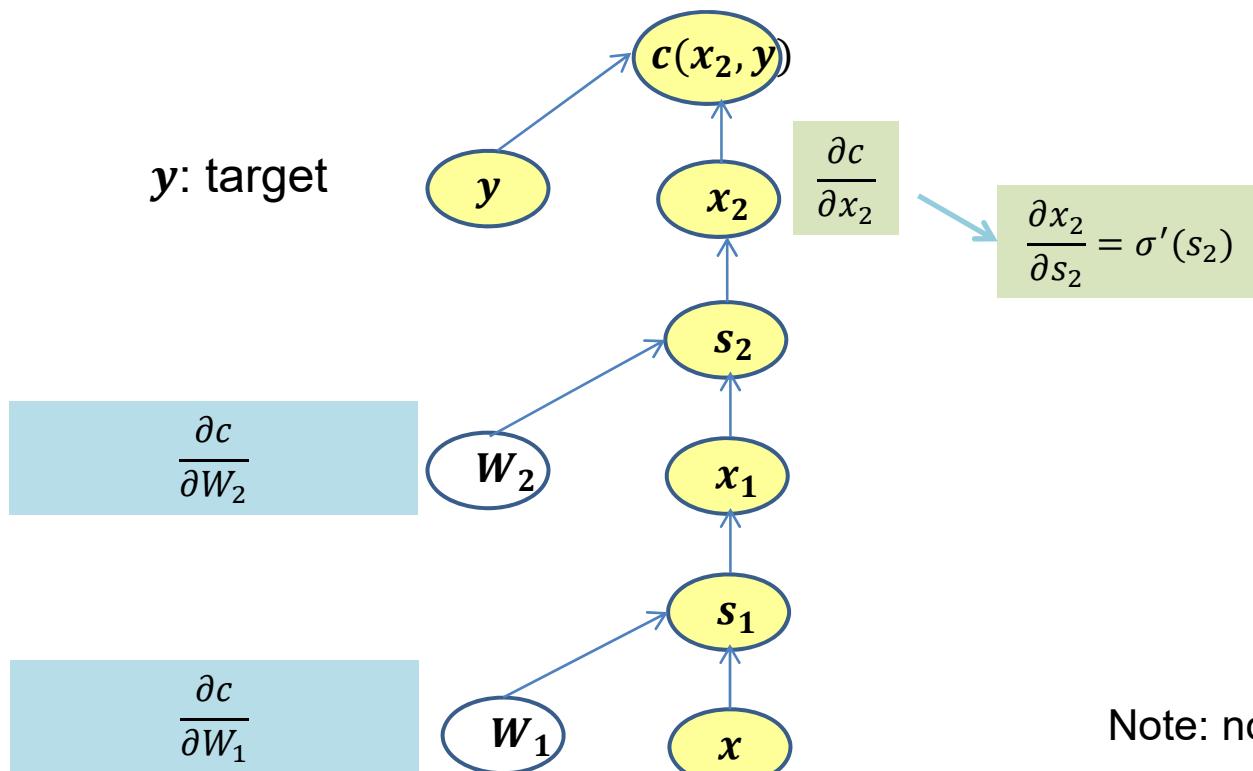
$$\frac{\partial c}{\partial x_{n-1}} = \frac{\partial c}{\partial s_n} W_n$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial x}$ and $\frac{\partial c}{\partial s}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- ▶ Back propagation: Reverse Mode Differentiation

$c(x_2, y)$: loss



Backward propagation:

$$\frac{\partial c}{\partial s_n} = \frac{\partial c}{\partial x_n} \odot \sigma'(s_n)^T$$

$$\frac{\partial c}{\partial W_n} = x_{n-1} \frac{\partial c}{\partial s_n}$$

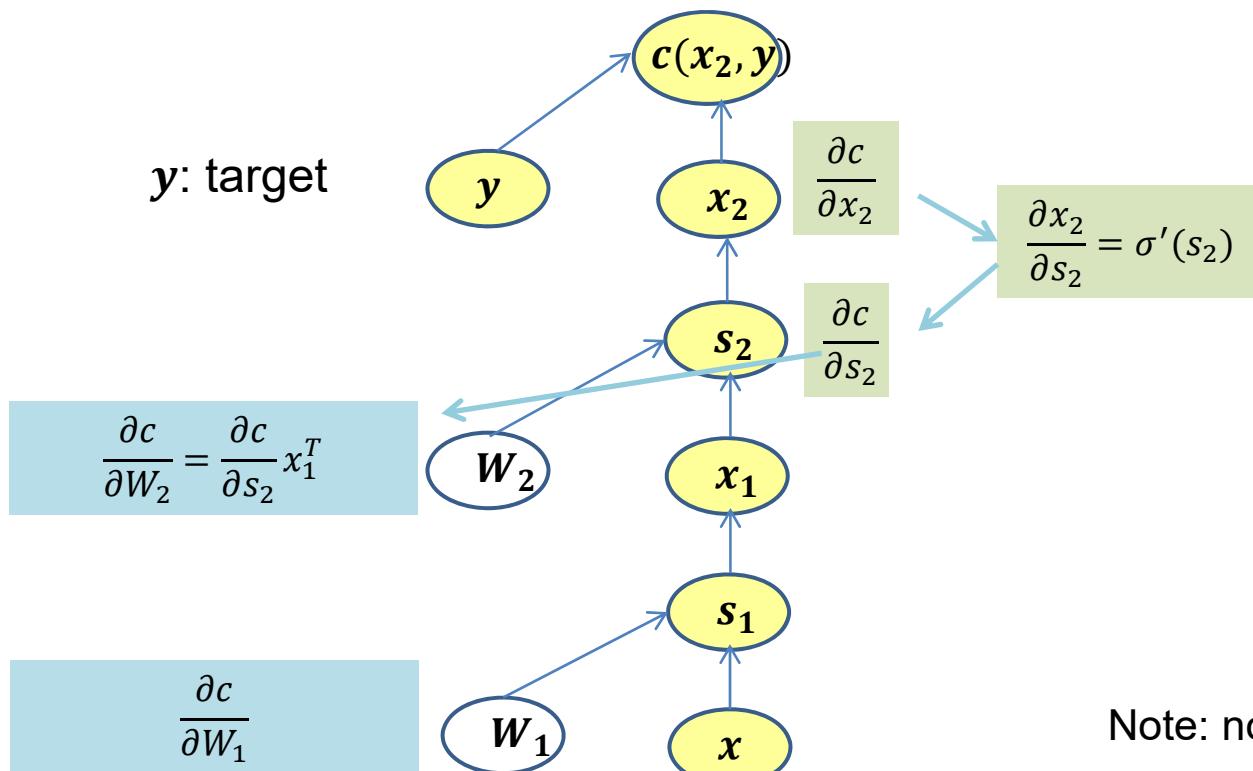
$$\frac{\partial c}{\partial x_{n-1}} = \frac{\partial c}{\partial s_n} W_n$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial x}$ and $\frac{\partial c}{\partial s}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- ▶ Back propagation: Reverse Mode Differentiation

$c(x_2, y)$: loss



Backward propagation:

$$\frac{\partial c}{\partial s_n} = \frac{\partial c}{\partial x_n} \odot \sigma'(s_n)^T$$

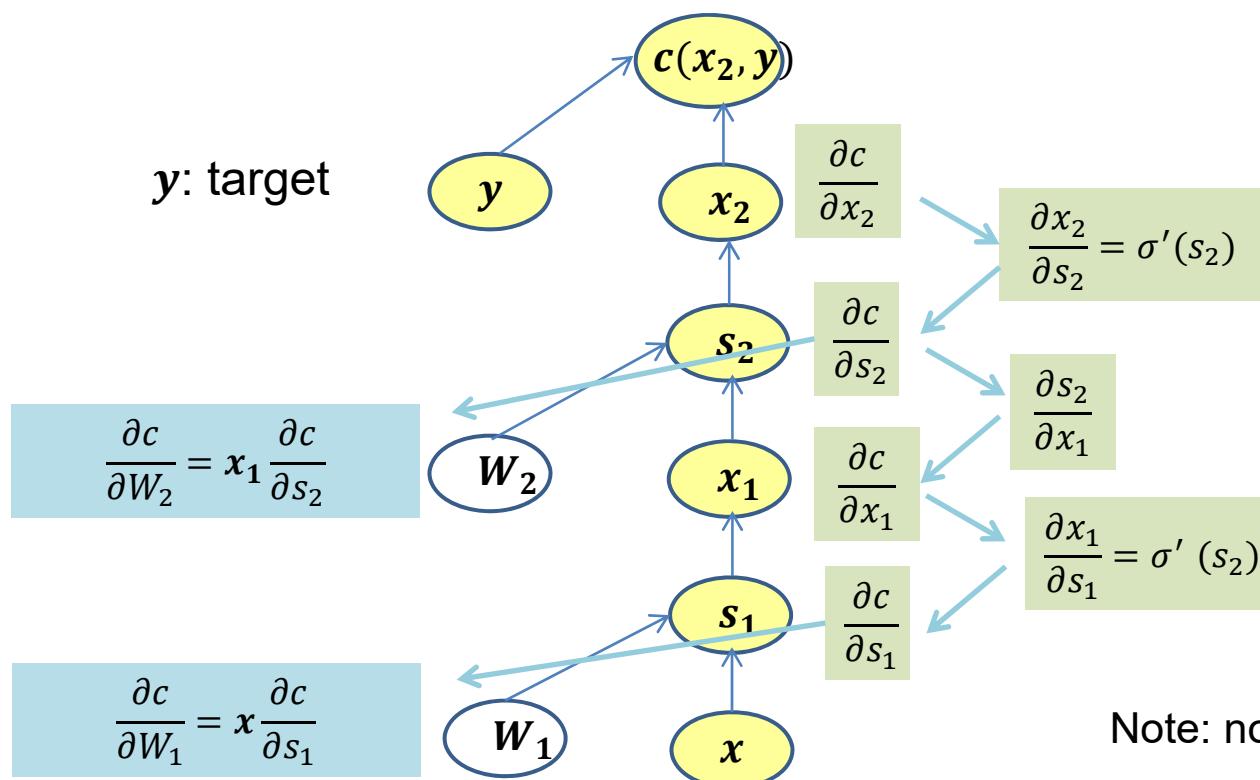
$$\frac{\partial c}{\partial W_n} = x_{n-1} \frac{\partial c}{\partial s_n}$$

$$\frac{\partial c}{\partial x_{n-1}} = \frac{\partial c}{\partial s_n} W_n$$

Note: notations are in vector form, $\frac{\partial c}{\partial w}$ is a matrix, $\frac{\partial c}{\partial x}$ and $\frac{\partial c}{\partial s}$ are column vectors of the appropriate size

Multi-layer Perceptron - Training

- ▶ Back propagation: Reverse Mode Differentiation
 $c(x_2, y)$: loss



Backward propagation:

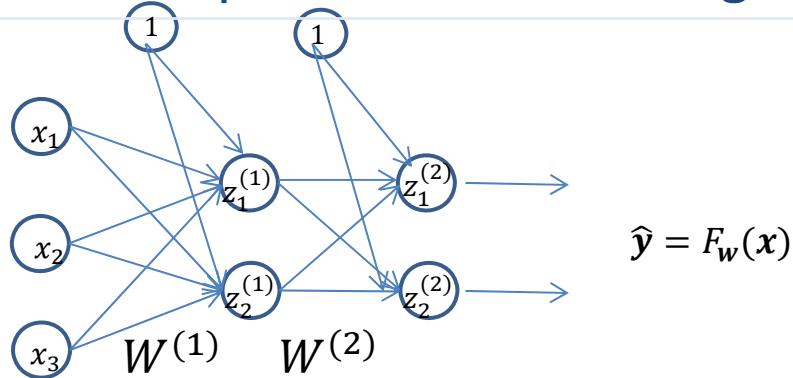
$$\frac{\partial c}{\partial s_n} = \frac{\partial c}{\partial x_n} \odot \sigma'(s_n)^T$$

$$\frac{\partial c}{\partial W_n} = x_{n-1} \frac{\partial c}{\partial s_n}$$

$$\frac{\partial c}{\partial x_{n-1}} = \frac{\partial c}{\partial s_n} W_n$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial x}$ and $\frac{\partial c}{\partial s}$ are row vectors of the appropriate size

Multi-layer Perceptron – SGD Training - example

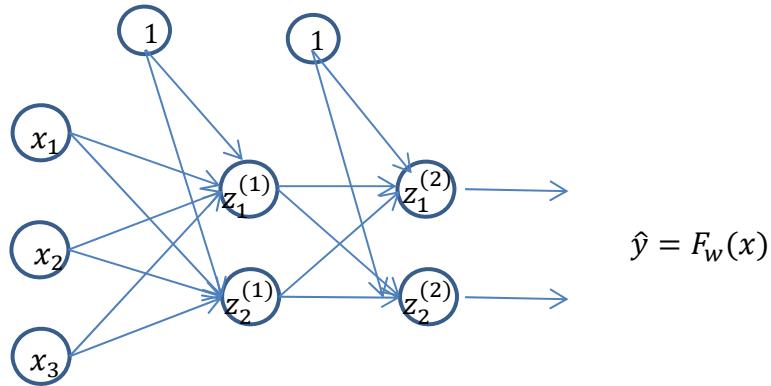


► Notations

- $\mathbf{z}^{(i)}$ activation vector for layer i
- $z_j^{(i)}$ activation of neuron j in layer i
- $W^{(i+1)}$ weight matrix from layer i to layer $i + 1$, including bias weights
- $w_{jk}^{(i)}$ weight from cell k on layer i to cell j on layer $i + 1$
- $\hat{\mathbf{y}}$ computed output
- $\hat{y}_1 = z_1^{(2)} = g(w_{10}^{(2)} + w_{11}^{(2)}z_1^{(1)} + w_{12}^{(2)}z_2^{(1)})$
- $z_1^{(1)} = g(w_{10}^{(1)} + w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3)$
- $W^{(1)} = \begin{pmatrix} w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{pmatrix}$

Multi-layer Perceptron – SGD Training – example

Inference : forward pass



- ▶ For example x
 - ▶ The activations of all the neurons from layer 1 are computed in parallel
 - ▶ $a^{(1)} = W^{(1)}x$ then $z^{(1)} = g(a^{(1)})$
 - with $g(a^{(1)}) = (g(a_1^{(1)}), g(a_2^{(1)}))^T$
 - ▶ The activations of cells on layer 1 are then used as inputs for layer 2. The activations of cells in layer 2 are computed in parallel.
 - ▶ $a^{(2)} = W^{(2)}z^{(1)}$ then $\hat{y} = z^{(2)} = g(a^{(2)})$
 -

Multi-layer Perceptron – SGD Training Summary (MSE loss + sigmoid units)

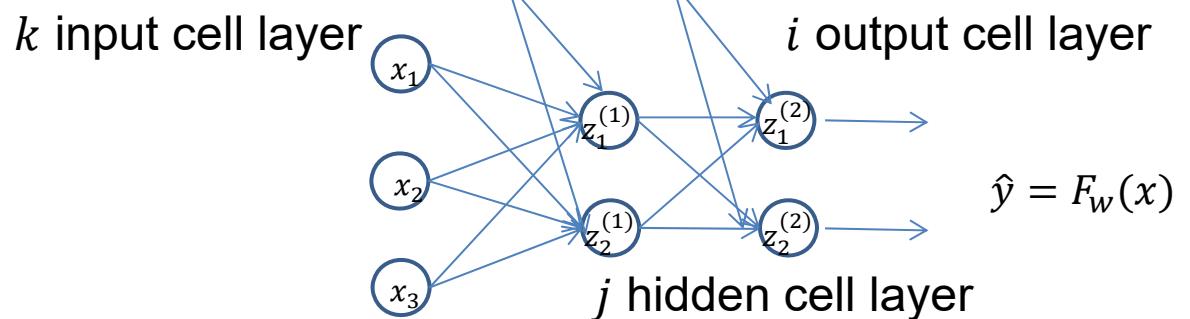
- ▶ The algorithm is described for a MSE loss – similar derivations for other losses
 - ▶ MLP with $M + 1$ layers of cells numbered 0 (input layer), ..., M (output layer), M weight layers numbered W^1, \dots, W^M , $w_{ij}^{(m)}$ is the weight from cell j in layer $m - 1$ to cell i in layer m (and is one of the components of W^m)
- ▶ Algorithm
 - ▶ Sample an example $(x, y), x \in R^n, y \in R^p$
 - ▶ Compute output $\hat{y} = F_W(x), \hat{y} \in R^p$
 - ▶ Compute difference $\delta = (y - \hat{y}) = (y_1 - \hat{y}_1, \dots, y_p - \hat{y}_p)^T$
 - ▶ Back propagate this error from the last weight layer to the first weight layer:
 - $w_{ij}^{(m)} = w_{ij}^{(m)} + \Delta w_{ij}^{(m)}$ → update equation for layer m and weight w_{ij}^m
 - $\Delta w_{ij}^{(m)} = \epsilon e_i^{(m)} z_j^{(m-1)}$ → gradient for $w_{ij}^{(m)}$
 - « e » is the quantity that will be back propagated
 - $e_i^{(M)} = \delta_i g'(a_i^{(M)})$ if i is an output cell with $\delta_i = (y_i - \hat{y}_i)$
 - $e_i^{(m)} = g'(a_i^{(m)}) \sum_{h \text{ parents of } i} e_h^{(m+1)} w_{hi}^{(m+1)}$ if i is not an output cell

Multi-layer Perceptron – SGD derivation

Detailed derivation for a 1 hidden layer network (MSE loss + sigmoid units)

▶ Forward pass

- ▶ Indices used below for this detailed derivation: i output cell layer, j hidden cell layer, k input cell layer



- ▶ $a_j^{(1)} = \sum_k w_{jk}^{(1)} x_k, z_j^{(1)} = g(a_j^{(1)})$
- ▶ $a_i^{(2)} = \sum_j w_{ij}^{(2)} z_j^{(1)}, z_i^{(2)} = g(a_i^{(2)})$
 - ▶ $a_i^{(2)} = \sum_j w_{ij}^{(2)} g(\sum_k w_{jk}^{(1)} x_k), z_i^{(2)} = g(\sum_j w_{ij}^{(2)} g(\sum_k w_{jk}^{(1)} x_k))$

▶ Loss

- ▶ $c = \frac{1}{2} \sum_i (y_i - \hat{y}_i)^2 = \frac{1}{2} \sum_i \left(y_i - g(\sum_j w_{ij}^{(2)} z_j^{(1)}) \right)^2$

Multi-layer Perceptron – SGD derivation

Detailed derivation for a 1 hidden layer network (MSE loss + sigmoid units)

▶ Backward (derivative) pass

▶ 2nd weight layer

- ▶ $\Delta w_{ij}^{(2)} = -\epsilon \frac{\partial C}{\partial w_{ij}^{(2)}} = -\epsilon \frac{\partial C}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_{ij}^{(2)}}$
- ▶ $\Delta w_{ij}^{(2)} = \epsilon(y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial a_i^{(2)}} \frac{\partial a_i^{(2)}}{\partial w_{ij}^{(2)}}$
- ▶ $\Delta w_{ij}^{(2)} = \epsilon(y_i - \hat{y}_i)g'(a_i^{(2)})z_j^{(1)}$
- ▶ $\Delta w_{ij}^{(2)} = \epsilon e_i^{(2)}z_j^{(1)}$, with $e_i^{(2)} = (y_i - \hat{y}_i)g'(a_i^{(2)})$

▶ 1st weight layer

- ▶ $\Delta w_{jk}^{(1)} = -\epsilon \frac{\partial C}{\partial w_{jk}^{(1)}} = -\epsilon \frac{\partial C}{\partial z_j^{(1)}} \frac{\partial z_j^{(1)}}{\partial w_{jk}^{(1)}}$
- $\frac{\partial C}{\partial z_j^{(1)}} = \sum_i \text{parents of } j \frac{\partial C}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_j^{(1)}} = -\sum_i (y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial a_i^{(2)}} \frac{\partial a_i^{(2)}}{\partial z_j^{(1)}}$
- $\frac{\partial C}{\partial z_j^{(1)}} = -\sum_i (y_i - \hat{y}_i)g'(a_i^{(2)})w_{ij}^{(2)}$

Multi-layer Perceptron – SGD derivation

Detailed derivation (MSE loss + sigmoid units)

- $\frac{\partial z_j^{(1)}}{\partial w_{jk}^{(1)}} = \frac{\partial z_j^{(1)}}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{jk}^{(1)}} = g'(a_j^{(1)}) x_k$
- ▶ $\Delta w_{jk}^{(1)} = \epsilon \sum_{i \text{ parents of } j} (y_i - \hat{y}_i) g'(a_i^{(2)}) w_{ij}^{(2)} g'(a_j^{(1)}) x_k$
- ▶ $\Delta w_{jk}^{(1)} = \epsilon e_j^{(1)} x_k$ with $e_j = g'(a_j^{(1)}) \sum_{i \text{ parents of } j} e_i w_{ij}^{(2)}$

Back Propagation and Adjoint

- ▶ BP is an instance of a more general technique: the Adjoint method
- ▶ Adjoint method
 - ▶ has been designed for computing **efficiently** the sensitivity of a loss to the parameters of a function (e.g. weights, inputs or any cell value in a NN).
 - ▶ Can be used to solve different constrained optimization problems (including BP)
 - ▶ Is used in many fields like control, geosciences
 - ▶ Interesting to consider the link with the adjoint formulation since this opens the way to generalization of the BP technique to more general problems
 - ▶ e.g. continuous NNs (Neural ODE)

Back Propagation and Adjoint

- ▶ Learning problem

- ▶ $\text{Min}_W c = \frac{1}{N} \sum_{k=1}^N c(F(x^k), y^k)$
- ▶ With $F(x) = F_l \circ \dots \circ F_1(x)$

- ▶ Rewritten as a constrained optimisation problem

- ▶ $\text{Min}_W c = \frac{1}{N} \sum_{k=1}^N c(z^k(l), y^k)$

- ▶ Subject to
$$\begin{cases} z^k(l) = F_l(z^k(l-1), W(l)) \\ z^k(l-1) = F_{l-1}(z^k(l-2), W(l-1)) \\ \quad \quad \quad \vdots \\ z^k(1) = F_1(x^k, W(1)) \end{cases}$$

- ▶ Note

- ▶ z and W are vectors of the appropriate size
- ▶ e.g. $z(i)$ is $n_z(i) \times 1$ and $W(i)$ is $n_W(i) \times 1$

Back Propagation and Adjoint

- ▶ For simplifying, one considers SGD, i.e. $N = 1$
- ▶ The Lagrangian associated to the optimization problem is
 - ▶ $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^l \lambda_i^T (z(i) - F_i(z(i-1), W(i)))$
- ▶ The partial derivatives of the Lagrangian are
 - ▶ $\frac{\partial \mathcal{L}}{\partial z(l)} = -\lambda_l^T + \frac{\partial c(z(l), y)}{\partial z(l)}$
 - ▶ $\frac{\partial \mathcal{L}}{\partial z(i)} = -\lambda_i^T + \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i), W(i+1))}{\partial z(i)}, \quad i = 1 \dots l-1$
 - ▶ $\frac{\partial \mathcal{L}}{\partial W_i} = \lambda_i^T \frac{\partial F_i(z(i-1), W(i))}{\partial W(i)}, \quad i = 1 \dots l$
 - ▶ $\frac{\partial \mathcal{L}}{\partial \lambda_i} = z(i) - F_i(z(i-1), W(i)), \quad i = 1 \dots l$

▶ Note

- ▶ $\frac{\partial \mathcal{L}}{\partial z(i)}$ is $1 \times n_z(i)$, $\frac{\partial \mathcal{L}}{\partial W_i}$ is $1 \times n_W(i)$, $\frac{\partial \mathcal{L}}{\partial \lambda_i}$ is $1 \times n_\lambda(i)$, λ_i is $n_z(i) \times 1$, $\frac{\partial F_{i+1}(z(i), W(i+1))}{\partial z(i)}$ is $n_z(i+1) \times n_z(i)$, $\frac{\partial c(z(l), y)}{\partial z(l)}$ is $1 \times n_z(l)$, $\frac{\partial F_i(z(i-1), W(i))}{\partial W(i)}$ is $n_z(i) \times n_W(i)$

Back Propagation and Adjoint

▶ Forward equation

- ▶ $\frac{\partial \mathcal{L}}{\partial \lambda_i} = z(i) - F_i(z(i-1), W(i))$, $i = 1 \dots l$, represent the constraints
- ▶ One wants $\frac{\partial \mathcal{L}}{\partial \lambda_i} = 0$, $i = 1 \dots l$
- ▶ Starting from $i = 1$ up to $i = l$, this is exactly the forward pass of BP

▶ Backward equation

- ▶ Remember the Lagrangian
 - ▶ $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^l \lambda_i^T (z(i) - F_i(z(i-1), W(i)))$
 - ▶ Since one imposes $(z(i) - F_i(z(i-1), W(i))) = 0$ (forward pass), one can choose λ_i^T as we want
 - ▶ Let us choose the λ s such that $\frac{\partial \mathcal{L}}{\partial z(i)} = 0, \forall i$
 - ▶ The λ s can be computed backward Starting at $i = l$ down to $i = 1$
 - ▶ $\lambda_l^T = \frac{\partial c(z(l), y)}{\partial z(l)}$
 - ▶ ...
 - ▶ $\lambda_i^T = \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i), w(i+1))}{\partial z(i)} = \lambda_{i+1}^T \frac{\partial z(i+1)}{\partial z(i)}$

Back Propagation and Adjoint

► Derivatives

- All that remains is to compute the derivatives of \mathcal{L} wrt the W_i

$$\triangleright \frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_{i+1}^T \frac{\partial F_i(z(i-1), W(i))}{\partial W(i)}, \forall i$$

□ $\frac{\partial F_i(z(i-1), W(i))}{\partial W(i)}$ easy to compute

Back Propagation and Adjoint – Algorithm Recap

- ▶ Recap, BP algorithm with Adjoint

- ▶ Forward

- ▶ Solve forward $\frac{\partial \mathcal{L}}{\partial \lambda_i} = 0$

- ▶ $z(1) = F_1(z(0), W(1))$
 - ▶ ...
 - ▶ $z(i) = F_i(z(i-1), W(i))$

- ▶ Backward

- ▶ Solve backward $\frac{\partial \mathcal{L}}{\partial z(i)} = 0$

- ▶ $\lambda_l^T = \frac{\partial c(z(l), y)}{\partial z(l)}$
 - ▶ ...
 - ▶ $\lambda_i^T = \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i), w(i+1))}{\partial z(i)} = \lambda_{i+1}^T \frac{\partial z(i+1)}{\partial z(i)}$

- ▶ Derivatives

- $\frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_{i+1}^T \frac{\partial F_i(z(i-1), W(i))}{\partial W(i)}, \forall i$

Adjoint method – Adjoint equation

- ▶ Let us consider the Lagrangian written in a simplified form
 - ▶ $\mathcal{L}(x, W) = c(z(l), y) - \lambda^T g(z, W)$
 - ▶ z, W represent respectively all the variables of the NN and all the weights
 - ▶ z is a $1 \times n_z$ vector, and W is a $1 \times n_W$ vector
 - ▶ $g(z, W) = 0$ represents the constraints written in an implicit form
 - here the system $z(i) - F_{l-1}(z(i-1), W(i)) = 0, i = 1 \dots l$

The derivative of $\mathcal{L}(x, W)$ wrt W is

- ▶
$$\frac{d\mathcal{L}(x, W)}{dW} = \frac{\partial c}{\partial z} \frac{\partial z}{\partial W} - \lambda^T \left(\frac{\partial g}{\partial z} \frac{\partial z}{\partial W} + \frac{\partial g}{\partial W} \right)$$
- ▶
$$= \left(\frac{\partial c}{\partial z} - \lambda^T \frac{\partial g}{\partial z} \right) \frac{\partial z}{\partial W} + \lambda^T \frac{\partial g}{\partial W}$$
- ▶ In order to avoid computing $\frac{\partial z}{\partial W}$, choose λ such that
 - ▶ $\frac{\partial c}{\partial z} - \lambda^T \frac{\partial g}{\partial z} = 0$, rewritten as:

$$\frac{\partial g^T}{\partial z} \lambda = - \frac{\partial c}{\partial z} \quad <<<<<< \text{Adjoint Equation}$$

Adjoint method

- ▶ λ is determined from the Adjoint equation
 - ▶ Different options for solving λ , depending on the problem
 - ▶ For MLPs, the hierarchical structure leads to the backward scheme

Multi-layer Perceptron – stochastic gradient

▶ Note

- ▶ The algorithm has been detailed for « pure » SGD, i.e. one datum at a time
- ▶ In practical applications, one uses mini-batch implementations
- ▶ This accelerates GPU implementations
- ▶ The algorithm holds for any differentiable loss/ model
- ▶ Deep Learning on large architectures makes use of SGD variants, e.g. Adam

Loss functions

- ▶ Depending on the problem, and on model, different loss functions may be used
- ▶ Mean Square Error
 - ▶ For regression
- ▶ Classification, Hinge, logistic, cross entropy losses
 - ▶ Classification loss
 - ▶ Number of classification errors
 - ▶ Exemples
 - $\hat{\mathbf{y}} \in R^p, \mathbf{y} \in \{-1,1\}^p$
 - ▶ Hinge, logistic losses are used as proxies for the classification loss

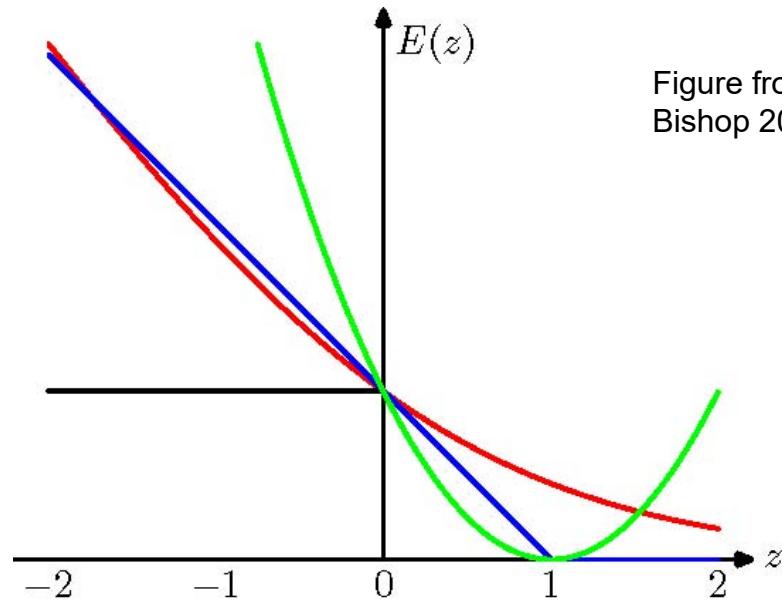


Figure from
Bishop 2006

z coordinate: $z = \hat{\mathbf{y}} \cdot \mathbf{y}$ (margin)

$$C_{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = ||\hat{\mathbf{y}} - \mathbf{y}||^2$$

$$C_{hinge}(\hat{\mathbf{y}}, \mathbf{y}) = [1 - \hat{\mathbf{y}} \cdot \mathbf{y}]_+ = \max(0, 1 - \hat{\mathbf{y}} \cdot \mathbf{y})$$

$$C_{logistic}(\hat{\mathbf{y}}, \mathbf{y}) = \ln(1 + \exp(-\hat{\mathbf{y}} \cdot \mathbf{y}))$$

Approximation properties of MLPs

- ▶ Results based on functional analysis
 - ▶ (Cybenko 1989)
 - ▶ Theorem 1 (regression): Let f be a continuous saturating function, then the space of functions $g(x) = \sum_{j=1}^n v_j f(\mathbf{w}_j \cdot \mathbf{x})$ is dense in the space of continuous functions on the unit cube $C(I)$. i.e. $\forall h \in C(I) \text{ et } \forall \epsilon > 0, \exists g : |g(x) - h(x)| < \epsilon \text{ on } I$
 - ▶ Theorem 2 (classification): Let f be a continuous saturating function. Let F be a decision function defining a partition on I . Then $\forall \epsilon > 0$, there exists a function $g(x) = \sum_{j=1}^n v_j f(\mathbf{w}_j \cdot \mathbf{x})$ and a set $D \subset I$ such that $\text{measure}(D) = 1 - \epsilon(D)$ and $|g(x) - F(x)| < \epsilon$ on D
 - ▶ .
 - ▶ (Hornik et al., 1989)
 - ▶ Theorem 3 : For any increasing saturating function f , and any probability measure m on R^n , the space of functions $g(x) = \sum_{j=1}^n v_j f(\mathbf{w}_j \cdot \mathbf{x})$ is uniformly dense on the compact sets $C(R^n)$ - the space of continuous functions on R^n
 - ▶ Note:
 - ▶ None of these results is constructive

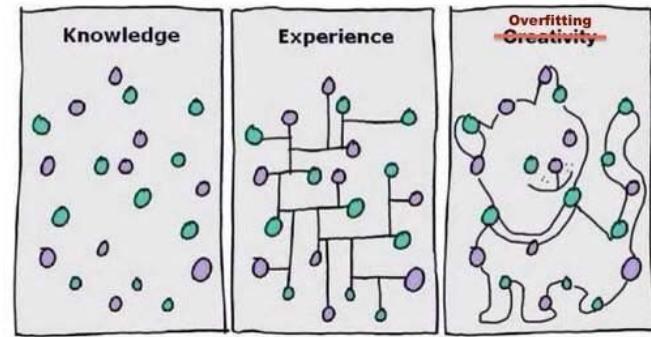
Complexity control

Bias – Variance

Overtraining and regularization

Generalization and Model Selection

- ▶ Complex models sometimes perform worse than simple linear models
 - ▶ Overfitting/ generalization problem

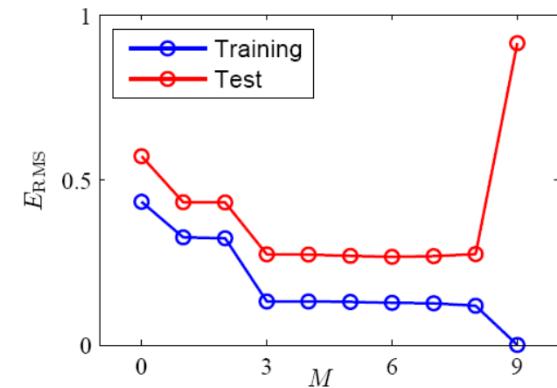
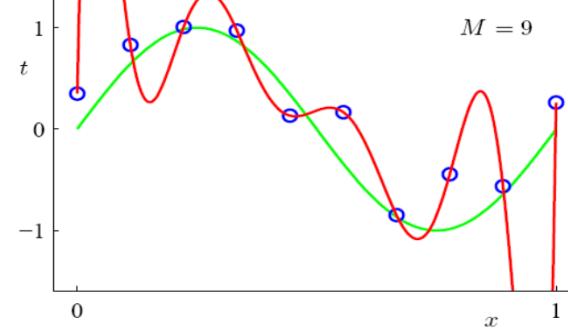
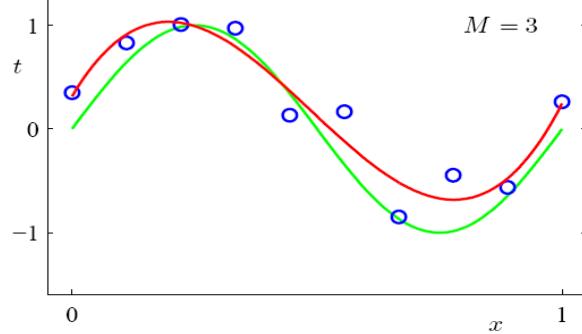
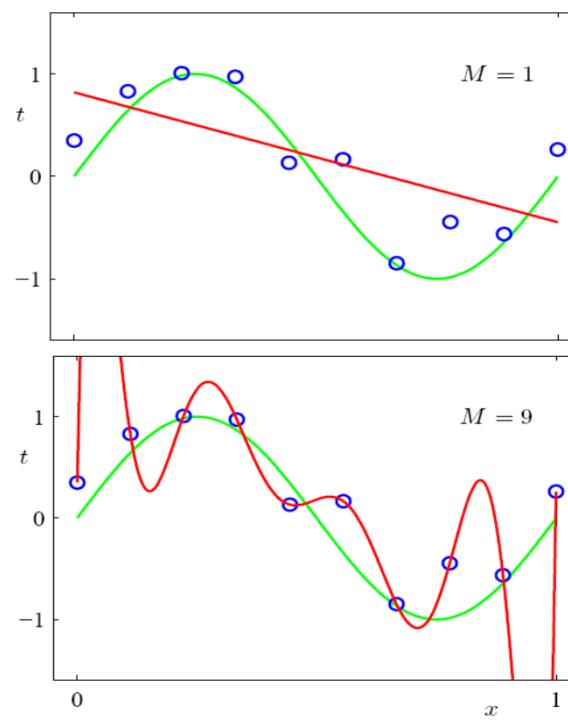
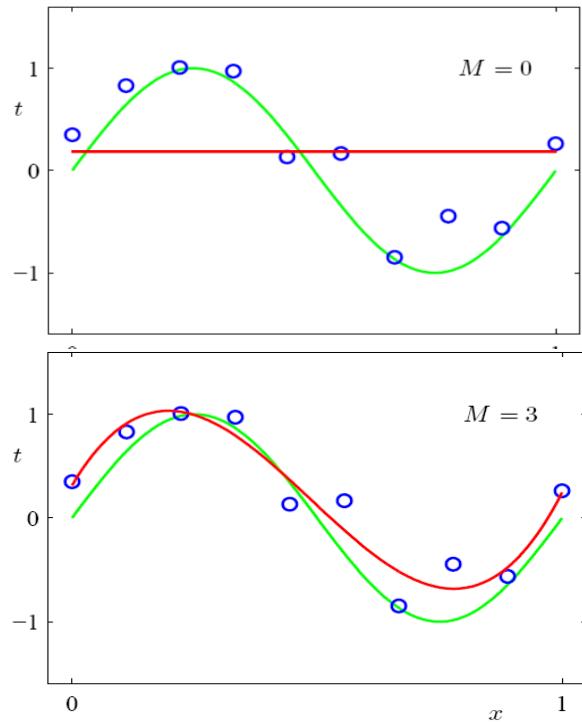


- ▶ Empirical Risk Minimization is not sufficient
 - ▶ The model complexity should be adjusted both to the task and to the information brought by the examples
 - ▶ Both the model parameters and the model capacity should be learned
 - ▶ Lots of practical method and of theory has been devoted to this problem

Complexity control

Overtraining / generalization for regression

- ▶ Example (Bishop 06) fit of a sinusoid with polynomials of varying degrees



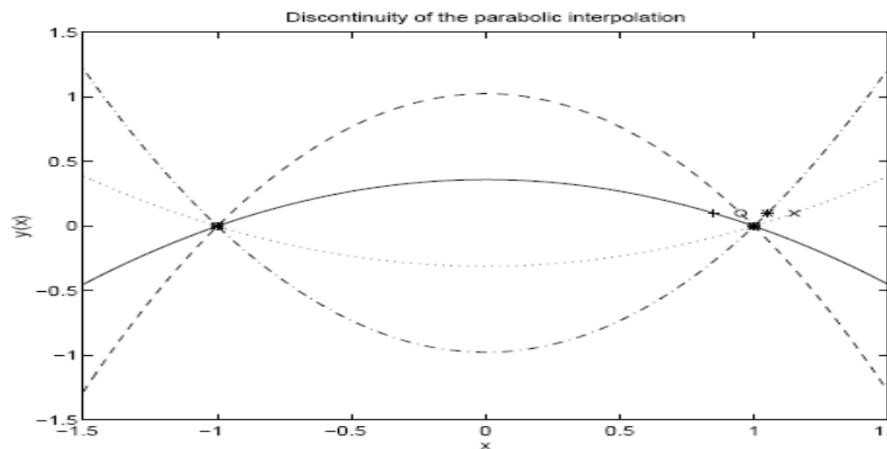
- ▶ Model complexity shall be controlled (learned) during training
 - ▶ How?

Complexity control

- ▶ One shall optimize the risk while controlling the complexity
- ▶ Several methods
 - ▶ Régularisation (Hadamard ... Tikhonov)
 - ▶ Theory of ill posed problems
 - ▶ Minimization of the structural risk (Vapnik)
 - ▶ Algebraic estimators of generalization error (AIC, BIC, LOO, etc)
 - ▶ Bayesian learning
 - ▶ Provides a statistical explanation of regularization
 - ▶ Regularization terms appear as priors on the parameter distribution
 - ▶ Ensemble methods
 - ▶ Boosting, bagging, etc
 - ▶ Many others especially in the Deep NN literature (seen later)

Regularisation

- ▶ Hadamard
 - ▶ A problem is well posed if
 - ▶ A solution exists
 - ▶ It is unique and stable
 - ▶ Example of ill posed problem (Goutte 1997)



- ▶ Tikhonov
 - ▶ Proposes methods pour transforming a ill posed problem into a “well” posed one

Bias-variance decomposition

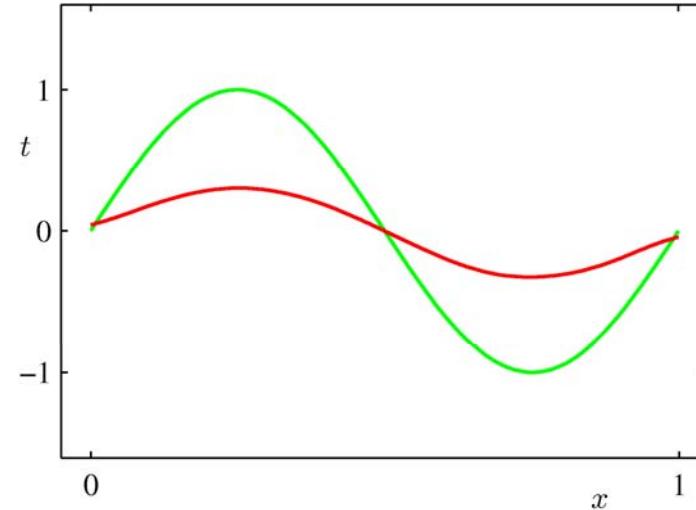
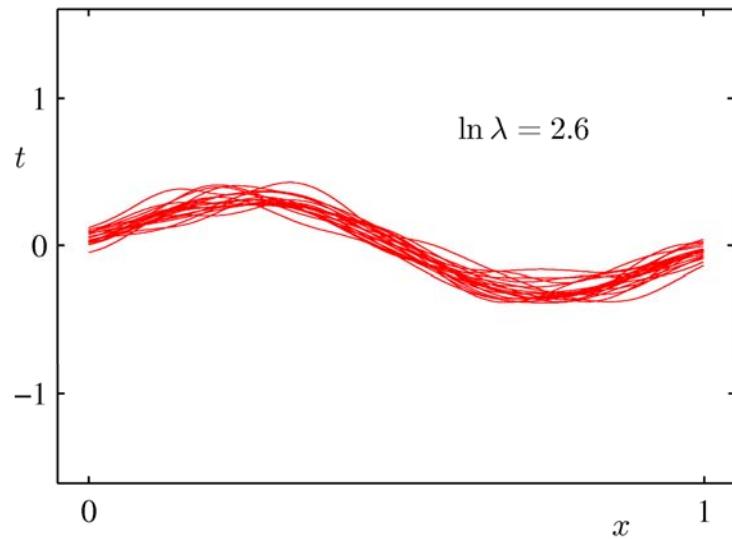
- ▶ Illustrates the problem of model selection, puts in evidence the influence of the complexity of the model
 - ▶ Remember: MSE risk decomposition
 - ▶ $E_{x,y} \left[(y - F_w(x))^2 \right] = E_{x,y} \left[(y - E_y[y|x])^2 \right] + E_{x,y} \left[(E_y[y|x] - F_w(x))^2 \right]$
 - ▶ Let $h^*(x) = E_y[y|x]$ be the optimal solution for the minimization of this risk
 - ▶ In practice, the number of training data for estimating $E_d[d|x]$ is limited
 - ▶ The estimation will depend on the training set D
 - ▶ Uncertainty due to the training set choice for this estimator can be measured as follows
 - Sample a series of training sets, all of size N : D_1, D_2, \dots
 - Learn $F_w(x, D)$ for each of these datasets
 - Compute the mean of the empirical errors obtained on these different datasets

Bias-variance decomposition

- ▶ Let us consider the quadratic error $(F_w(x; D) - h^*(x))^2$ for a datum x and for the solution $F_w(x; D)$ obtained with the training set D (in order to simplify, we consider a 1 dimensional real output, extension to multidimensional outputs is trivial)
 - ▶ Let $E_D[F_w(x; D)]$ denote the expectation wrt the distribution of D
- ▶ $(F_w(x; D) - h^*(x))^2$ decomposes as:
 - ▶ $(F_w(x; D) - h^*(x))^2 = (F_w(x; D) - E_D[F_w(x; D)] + E_D[F_w(x; D)] - h^*(x))^2$
 - ▶ $(F_w(x; D) - h^*(x))^2 = (F_w(x; D) - E_D[F_w(x; D)])^2 + (E_D[F_w(x; D)] - h^*(x))^2 + 2(F_w(x; D) - E_D[F_w(x; D)])(E_D[F_w(x; D)] - h^*(x))$
- ▶ Expectation $E_D[(F_w(x; D) - h^*(x))^2]$ wrt D distribution decomposes as:
 - ▶ $E_D[(F_w(x; D) - h^*(x))^2] = (E_D[F_w(x; D)] - h^*(x))^2 + E_D[(F_w(x; D) - E_D[F_w(x; D)])^2] = bias^2 + variance$
- ▶ Intuition
 - ▶ Choosing the right model requires a compromise between flexibility and simplicity
 - ▶ Flexible model : low bias – strong variance
 - ▶ Simple model : strong bias – low variance

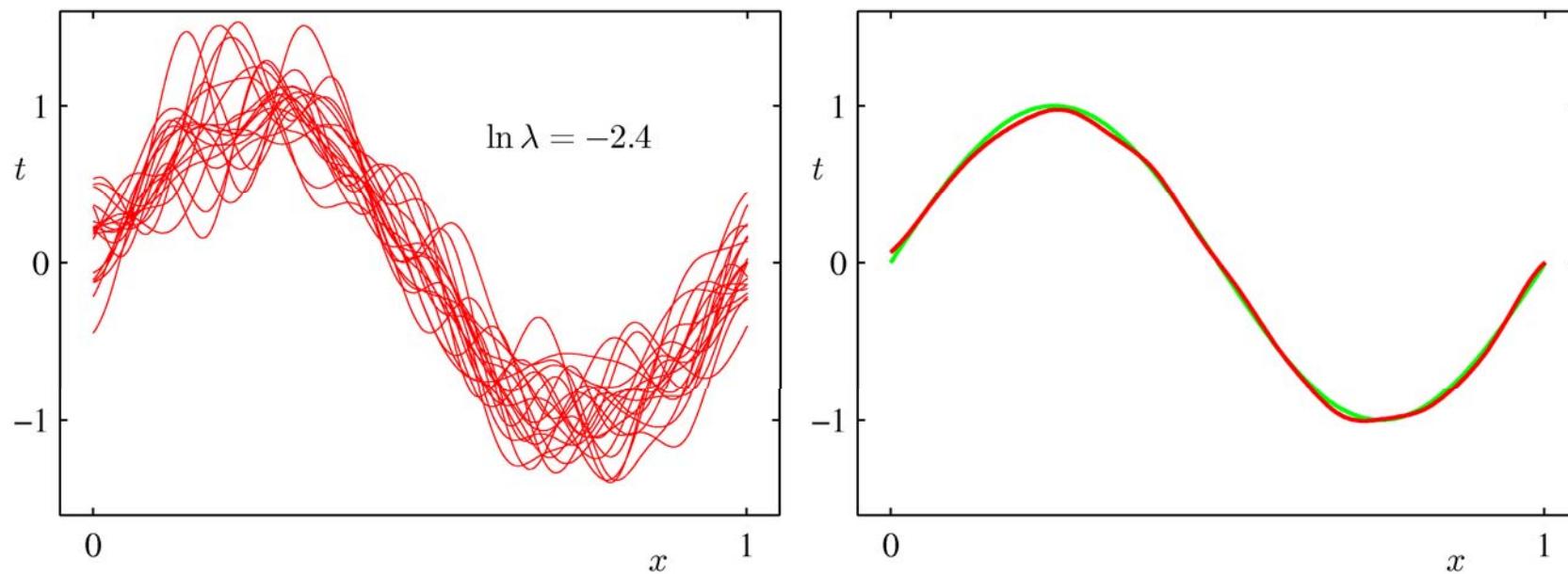
The Bias-Variance Decomposition (Bishop PRML 2006)

- ▶ Example: 100 data sets from the sinusoidal, varying the degree of regularization
 - ▶ Model: gaussian basis function, Learning set size = 25, λ is the regularization parameter
 - High values of λ correspond to simple models, low values to more complex models
 - ▶ Left 20 of the 100 models shown
 - ▶ Right : average of the 100 models (red), true sinusoid (green)
 - ▶ Figure illustrates high bias and low variance ($\lambda = 13$)



The Bias-Variance Decomposition (Bishop PRML 2006)

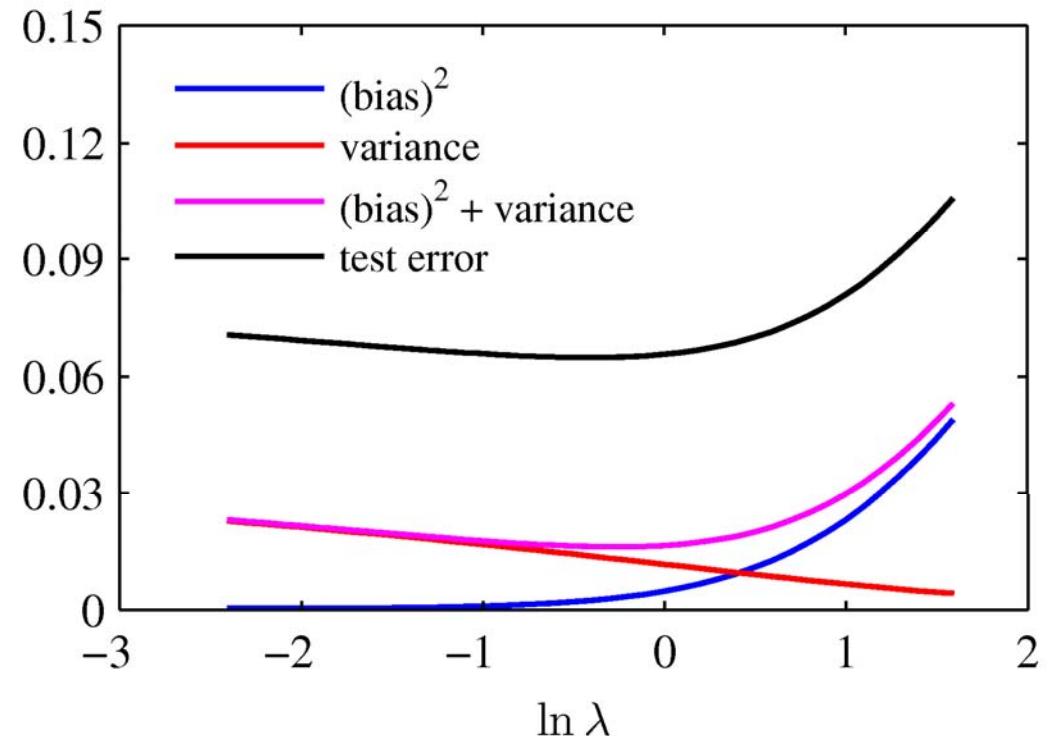
- ▶ Example: 100 data sets from the sinusoidal, varying the degree of regularization
 - ▶ Same setting as before
 - Figure illustrates low bias and high variance ($\lambda = 0.09$)



- ▶ Remark
 - The mean of several complex models behaves well here (reduced variance)
 - → leads to ensemble methods

The Bias-Variance Decomposition (Bishop PRML 2006)

- ▶ From these plots, we note that an over-regularized model (large λ) will have a high bias, while an under-regularized model (small λ) will have a high variance.



Regularisation

- ▶ Principle: control the solution variance by constraining function F
 - ▶ Optimise $C = C_1 + \lambda C_2$
 - ▶ C is a compromise between
 - ▶ C_1 : reflects the objective e.g. MSE, Entropie, ...
 - ▶ C_2 : constraints on the solution (e.g. weight distribution)
 - ▶ λ : constraint weight
- ▶ Regularized mean squares
 - ▶ For the linear multivariate regression
 - ▶ $C = \frac{1}{N} \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i)^2 + \frac{\lambda}{2} \sum_{j=1}^n |w_j|^q$
 - ▶ $q = 2$ regularization L_2 , $q = 1$ regularization L_1 also known as « Lasso »

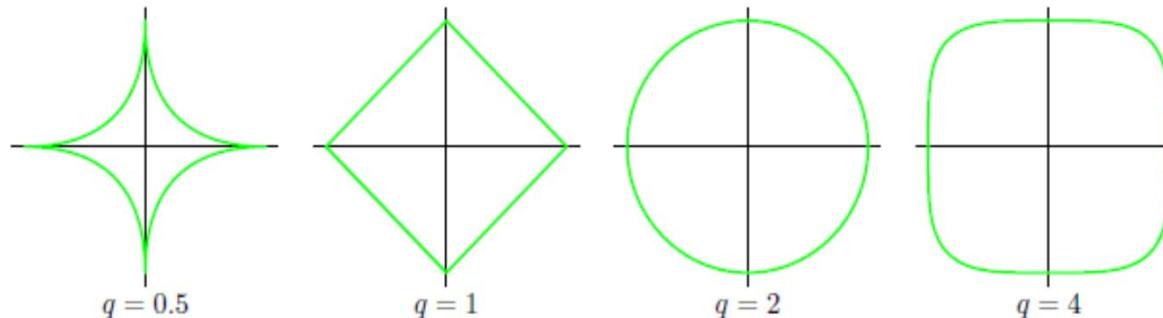


Fig. from Bishop 2006

Figure 3.3 Contours of the regularization term in (3.29) for various values of the parameter q .

Régularisation

▶ Solve

$$\triangleright \text{Min}_{\mathbf{w}} C = \frac{1}{N} \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i)^2 + \frac{\lambda}{2} \sum_{j=1}^n |w_j|^q, \lambda > 0$$

▶ Amounts at solving the following constrained optimization problem

$$\triangleright \text{Min}_{\mathbf{w}} C = \frac{1}{N} \sum_{i=1}^N (y^i - \mathbf{w} \cdot \mathbf{x}^i)^2$$

▶ Under constraint $\sum_{j=1}^n |w_j|^q \leq s$ for a given value of s

▶ Effect of this constraint

Figure 3.4 Plot of the contours of the unregularized error function (blue) along with the constraint region (3.30) for the quadratic regularizer $q = 2$ on the left and the lasso regularizer $q = 1$ on the right, in which the optimum value for the parameter vector \mathbf{w} is denoted by \mathbf{w}^* . The lasso gives a sparse solution in which $w_1^* = 0$.

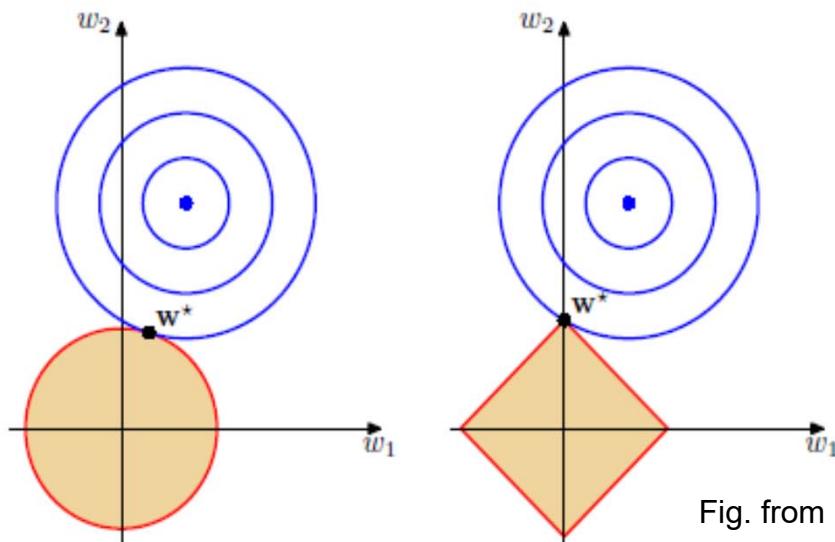


Fig. from Bishop 2006

Regularization

▶ Penalization L_2

▶ Loss

$$\triangleright C = C_1 + \lambda \sum_{j=1}^n |w_j|^2$$

▶ Gradiant

$$\triangleright \nabla_w C = \lambda w + \nabla_w C_1$$

▶ Update

$$\triangleright w = w - \epsilon \nabla_w C = (1 - \epsilon \lambda) w - \epsilon \nabla_w C_1$$

▶ Penalization is proportional to w

▶ Penalization L_1

▶ Loss

$$\triangleright C = C_1 + \lambda \sum_{j=1}^n |w_j|^1$$

▶ Gradiant

$$\triangleright \nabla_w C = \lambda sign(w) + \nabla_w C_1$$

▶ $sign(w)$ is the sign of w applied to each component of w

▶ Update

$$\triangleright w = w - \epsilon \nabla_w C = w - \epsilon \lambda sign(w) - \epsilon \nabla_w C_1$$

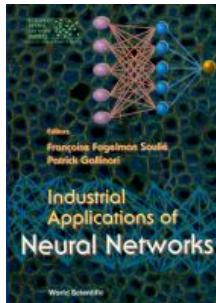
▶ Penalization is constant with sign $sign(w)$

Other ideas for improving generalization in NNs

- ▶ Learning rate decay
- ▶ Early stopping
- ▶ Data augmentation
 - ▶ By adding noise
 - with early work from Matsuoka 1992 ; Grandvallet and Canu 1994 ; Bishop 1994
 - And many new developments for Deep learning models
 - ▶ By generating new examples (synthetic, or any other way)
- ▶ Note: Bayesian learning and regularization
 - ▶ Regularization parameters correspond to priors on these model variables

Summary

- ▶ Non linear machines were widely developed in the 90^{ies}
- ▶ Foundations for modern statistical machine learning
- ▶ Foundations for statistical learning theory
- ▶ Real world applications



- ▶ Also during this period
 - ▶ Recurrent Neural Networks
 - ▶ Extension of back propagation
 - ▶ Reinforcement Learning
 - ▶ Early work mid 80ies
 - ▶ Sutton – Barto Book 1998, including RL + NN



Deep learning



Interlude: new actors – new practices

- ▶ GAFA (Google, Apple, Facebook, Amazon) , BAT (Baidu, Tencent, Alibaba), ..., Startups, are shaping the data world
- ▶ Research
 - ▶ Big Tech. actors are leading the research in DL
 - ▶ Large research groups
 - ▶ Google Brain, Google Deep Mind, Facebook FAIR, Baidu AI lab, Baidu Institute of Deep Learning, etc
 - ▶ Standard development platforms, dedicated hardware, etc
 - ▶ DL research requires access to resources
 - ▶ sophisticated libraries
 - ▶ large computing power e.g. GPU clusters
 - ▶ large datasets, ...



百度深度学习实验室
Relentless for Ultimate Intelligence

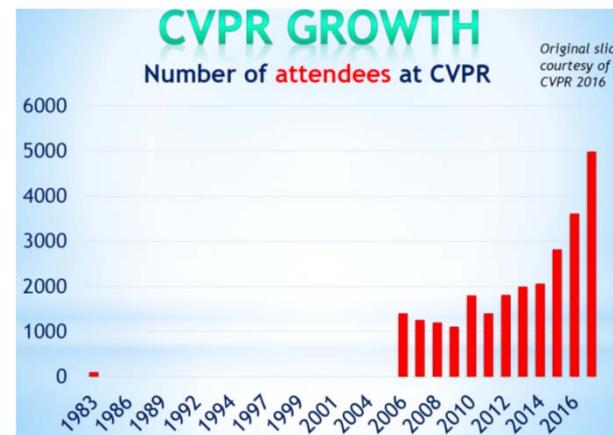
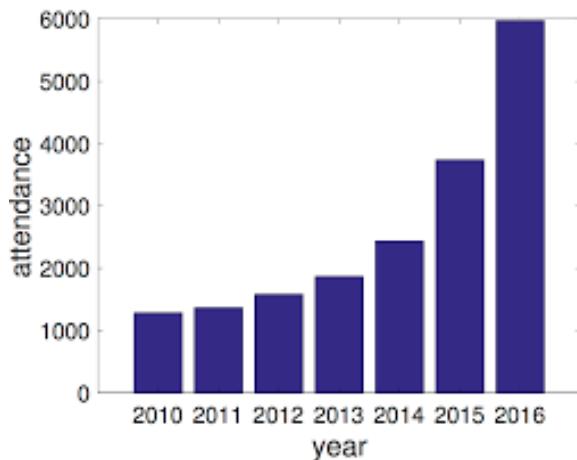


Facebook AI
Research



Interlude – ML conference attendance growth

- ▶ Nips attendance (T. Sejnowski)
- ▶ CVPR attendance



- ▶ 2017 sold out 1 week after registration opening, 7000 participants
- ▶ 2018, 2k inscriptions sold in 11 mn!

Interlude – Deep Learning platforms

- ▶ Deep Learning platforms offer
 - ▶ Classical DL models
 - ▶ Optimization algorithms
 - ▶ Automatic differentiation
 - ▶ Popular options/ tricks
 - ▶ Pretrained models
 - ▶ CUDA/ GPU/ CLOUD support
- ▶ Contributions by large open source communities: lots of code available
- ▶ Easy to build/ train sophisticated models
- ▶ Among the most populars platforms:
 - ▶ TensorFlow - Google Brain - Python, C/C++
 - ▶ PyTorch – Facebook- Python
 - ▶ Caffe – UC Berkeley / Caffe2 Facebook, Python, MATLAB
 - ▶ Higher level interfaces
 - ▶ e.g. Keras for TensorFlow
- ▶ And also:
 - ▶ PaddlePaddle (Baidu), MXNet (Amazon), Mariana (Tencent), PA 2.0 (Alibaba),



Interlude - Modular programming: Keras simple example MLP

From <https://keras.io/>

```
import keras  
from keras.models import Sequential  
from keras.layers import Dense, Dropout, Activation  
from keras.optimizers import SGD
```

```
# Load and format training and test data  
# Not shown - (x_train, y_train), (x_test, y_test)
```

```
model = Sequential()  
model.add(Dense(64, activation='relu', input_dim=20))  
model.add(Dense(64, activation='relu'))  
model.add(Dense(10, activation='softmax'))
```

```
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)  
model.compile(loss='categorical_crossentropy',  
              optimizer=sgd,  
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train,  
          epochs=20,  
          batch_size=128)  
score = model.evaluate(x_test, y_test, batch_size=128)
```

Load Training – Test data

Specify NN architecture:
• here basic MLP with 3 weight layers

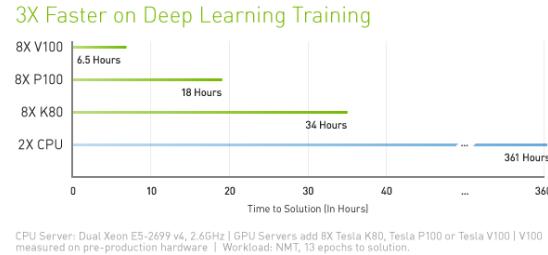
Optimisation algorithm
• SGD
Loss criterion
• Cross entropy

Train for 20 epochs

Evaluate performance on test set

Interlude – Hardware

- ▶ 2017 - NVIDIA V100 – optimized for Deep Learning
- ▶ 2017 - Google Tensor Processor Unit



- ▶ “With 640 Tensor Cores, Tesla V100 is the world’s first GPU to break the 100 teraflops (TFLOPS) barrier of deep learning performance. The next generation of [NVIDIA NVLink™](#) connects multiple V100 GPUs at up to 300 GB/s to create the world’s most powerful computing servers.”



- ▶ Cloud TPU



Motivations

- ▶ Learning representations
 - ▶ Handcrafted versus learned representation
 - ▶ Often complex to define what are good representations
 - ▶ General methods that can be used for
 - ▶ Different application domains
 - ▶ Multimodal data
 - ▶ Multi-task learning
 - ▶ Learning the latent factors behind the data generation
 - ▶ Unsupervised feature learning
 - ▶ Useful for learning data/ signal representations
- ▶ Deep Neural networks
 - ▶ Learn high level/ abstract representations from raw data
 - ▶ Key idea: stack layers of neurons to build deep architectures
 - ▶ Find a way to train them

High Level Representations in Videos – Google (Le et al. 2012)

- ▶ Objective
 - ▶ Learn high level representations without teacher
 - ▶ 10 millions images 200x200 from YouTube videos
 - ▶ Auto-encoder 10^9 connexions
- ▶ « High level » detectors
 - ▶ Show test images to the network
 - ▶ E.g. faces
 - ▶ Look for neurons with maximum response
- ▶ Some neurons respond to high level characteristics
 - ▶ Faces, cats, silhouettes, ...

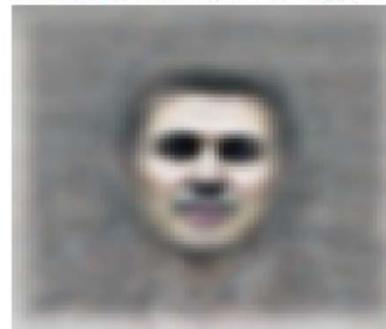


Figure 3. Top: Top 48 stimuli of the best neuron from the test set. Bottom: The optimal stimulus according to numerical constraint optimization.

Top: most responsive stimuli on the test set for the neuron. Bottom: Most responsive human body neuron on the test set for the human body neuron.

Useful Deep Learning heuristics

Deep NN make use of several (essential) heuristics for training large architecture: type of units, normalization, optimization...

We introduce some of these ideas

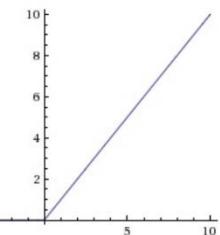
Deep Learning Tricks

Neuron units

- In addition to the logistic or tanh units, used in the 90s , other forms are used in deep architectures – Some of the popular forms are:

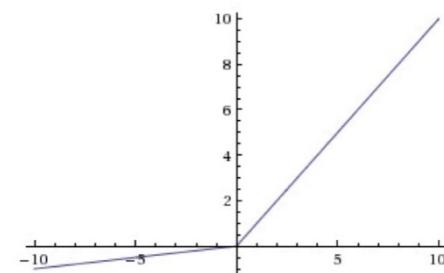
- ▶ RELU - Rectified linear units (used for internal layers)

- $g(\mathbf{x}) = \max(0, b + \mathbf{w} \cdot \mathbf{x})$
 - Rectified units allow to draw activations to 0 (used for sparse representation when unit is active)



- ▶ Leaky RELU (used for internal layers)

- $g(\mathbf{x}) = \begin{cases} b + \mathbf{w} \cdot \mathbf{x} & \text{if } b + \mathbf{w} \cdot \mathbf{x} > 0 \\ 0.01(b + \mathbf{w} \cdot \mathbf{x}) & \text{otherwise} \end{cases}$
 - Introduces a small derivative when $b + \mathbf{w} \cdot \mathbf{x} < 0$



- ▶ Maxout

- $g(\mathbf{x}) = \max_i (b_i + \mathbf{w}_i \cdot \mathbf{x})$
 - Generalizes the rectified unit
 - There are multiple weight vectors for each unit

- ▶ Softmax (used for output layer)

- ▶ Used for classification with a 1 out of p coding (p classes)
 - Ensures that the sum of predicted outputs sums to 1

- $$g(\mathbf{x}) = softmax(\mathbf{b} + W\mathbf{x}) = \frac{e^{b_i + (W\mathbf{x})_i}}{\sum_{j=1}^p e^{b_j + (W\mathbf{x})_j}}$$

Deep Learning Tricks

Normalisation

- ▶ Units: Batch Normalization (Ioffe 2015)
 - ▶ Normalize the activations of the units (hidden units) so as to coordinate the gradients across layers
 - ▶ Let $B = \{x^1, \dots, x^N\}$ be a mini batch, $h_i(x^j)$ the activation of hidden unit i for input x^j before non linearity
 - ▶ Training
 - ▶ Set $h'_i(x^j) = \frac{h_i(x^j) - \mu_i}{\sigma_i + \epsilon}$ where μ_i is the mean of the activities of hidden unit i on batch B , and σ_i its standard deviation
 - ▶ μ_i and σ_i are estimated on batch B , ϵ is a small positive number
 - ▶ The output of unit i is then $z_i = \gamma_i h'_i(x^j) + \beta_i$
 - Where γ and β are learned via SGD
 - ▶ Testing
 - ▶ μ_i and σ_i for test are estimated as a moving average during training, and need not be recomputed on the whole training dataset
- ▶ Note
 - ▶ No clear agreement if BN should be performed before or after non linearity
 - ▶ L^2 normalization could be used together with BN but reduced
- ▶ Gradient
 - ▶ Gradient clipping
 - ▶ Avoid very large gradient steps when the gradient becomes very large - different strategies work similarly in practice.
 - ▶ Let $\nabla_w c$ be the gradient computed over a minibatch
 - ▶ A possible clipping strategy is (Pascanu 2013)
 - $\nabla_w c = \frac{\nabla_w c}{\|\nabla_w c\|} v$, with v a norm threshold

Dropout

► Dropout (Srivastava 2014)

► Training

- Randomly drop units at training time
 - Parameter: dropout percentage p
 - Each unit is dropped with probability p
 - This means that it is inactive in the forward and backward pass

► Testing

- Initial paper (Srivastava 2014)

- Keep all the units
- Multiply the units activation by p during test
 - The expected output for a given layer during the test phase should be the same as during the training phase

► Inverted Dropout

- Current implementations use « inverted dropout » - easier implementation: the network does not change during the test phase (see next slide)
 - Units are dropped with probability p
 - Multiplies activations by $\frac{1}{1-p}$ during training, and keep the network untouched during testing

► Effects

- Increases independence between units and better distributes the representation
- Interpreted as an ensemble model; reduces model variance

Figure from Srivastava 2014

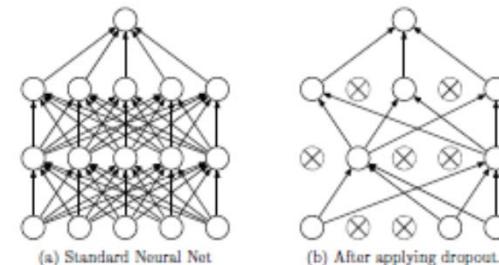


Figure 1: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout

▶ Dropout for a single unit

- ▶ Let p be the dropout probability
- ▶ Consider a neuron i with inputs $\mathbf{x} \in R^n$ and weight vector $\mathbf{w} \in R^n$ including the bias term
- ▶ The activation of neuron i is $z_i = f(\mathbf{w} \cdot \mathbf{x})$ with f a non linear function (e.g. Relu)
- ▶ Let b_i a binomial variable of parameter $1 - p$

▶ Original dropout

- ▶ Training phase
 - $z_i = b_i f(\mathbf{w} \cdot \mathbf{x}), b_i \in \{0,1\}$
- ▶ Test phase
 - $z_i = \frac{1}{1-p} f(\mathbf{w} \cdot \mathbf{x})$

▶ Inverted dropout

- ▶ Training phase
 - $z_i = \frac{1}{1-p} b_i f(\mathbf{w} \cdot \mathbf{x}), b_i \in \{0,1\}$
- ▶ Test phase
 - $z_i = f(\mathbf{w} \cdot \mathbf{x})$

▶ Note

- ▶ The total number of neurons dropped at each step is the sum of Bernoullis b_i , it follows a binomial distribution $B(m, p)$ where m is the number of neurons on the layer of neuron i .
- ▶ Its expectation is the $E[B(m, p)] = mp$
- ▶ L^2 normalization could be used together with dropout but reduced

CNN: Convolutional Neural Nets

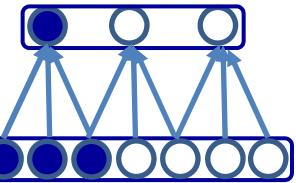
Introduction
Classification
Object detection
Image segmentation

CNNs

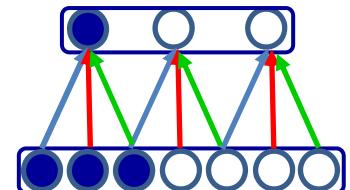
- ▶ CNNs were developed in the late 80ies for image and speech applications
- ▶ Deep CNNs were successfully used for image applications (classification and segmentation) in the 2010s – starting with the ImageNet competition, and for speech recognition.
 - ▶ Their use has been extended to handle several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

CNNs principle

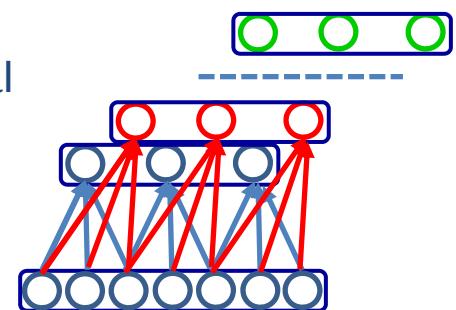
- ▶ Exploit local characteristics of the data via local connections
 - ▶ e.g. images (2 D), speech signal (1 D)



- ▶ Local connections are constrained to have shared weight vectors
 - ▶ This is equivalent to convolve a unique weight vector with the input signal
 - ▶ Think of a local edge detector for images
 - ▶ The 3 hidden cells here share the same weight vector
 - (blue, red, green weight values)

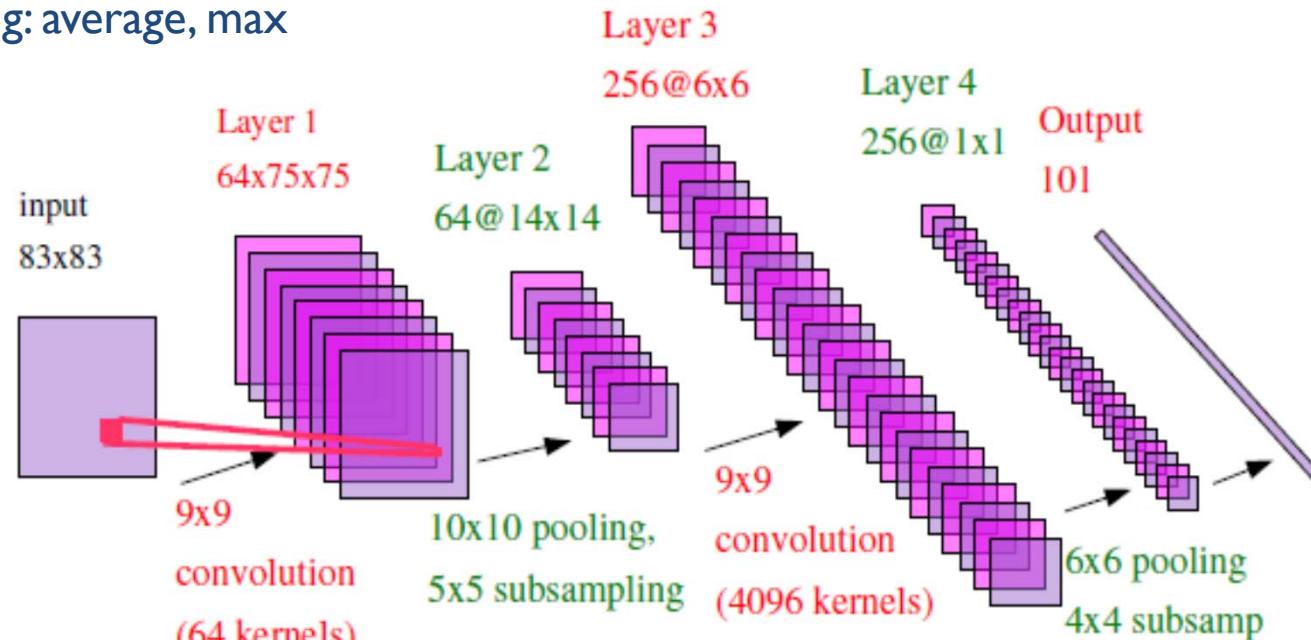


- ▶ Several convolution filters can be learned simultaneously
 - ▶ This corresponds to applying a set of local filters on the input signal
 - ▶ e.g edge detectors at different angles for an image
 - ▶ here colors indicate similar weight vectors, not weight values as above



CNNs example

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



parameters $64 \times 9 \times 9 = 5184$, $256 \times 9 \times 9 = 20736$, $256 \times 101 = 60916$

CNNs

▶ In Convnet

- ▶ The first hidden layer consists in 64 different convolution kernels over the initial input, resulting in 64 different mapping of the input
- ▶ The second hidden layer is a sub-sampling layer with a pooling transformation applied to each matrix representation of the first hidden layer
- ▶ etc
- ▶ Last layer is a classification layer, fully connected

▶ More generally

- ▶ CNNs alternate convolution, and pooling layers, and a fully connected layer at the top.

CNNs visualization

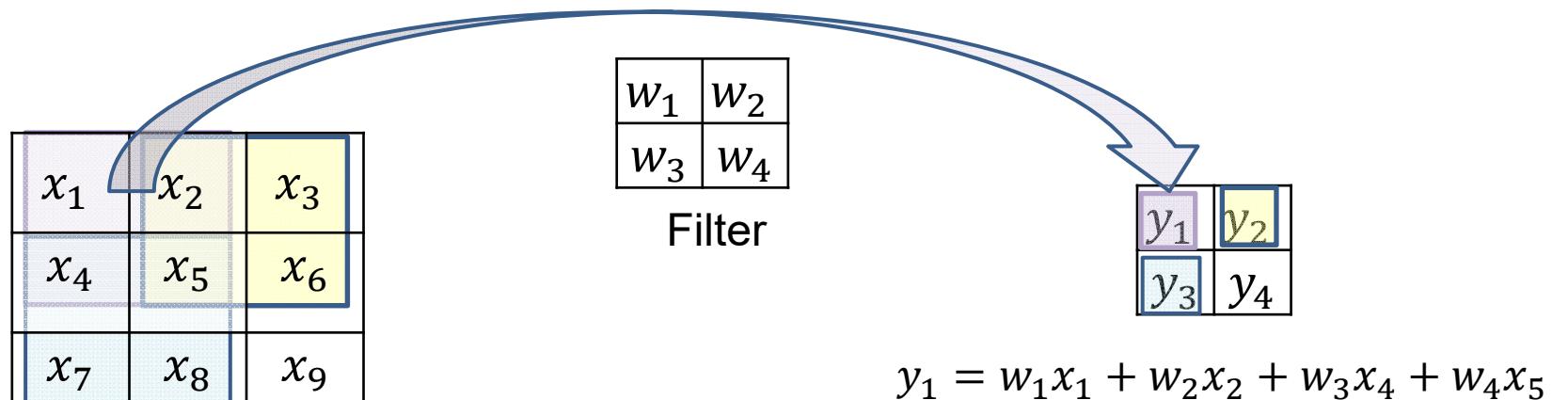
- ▶ Hand writing recognition (Y. LeCun Bell labs 1989)



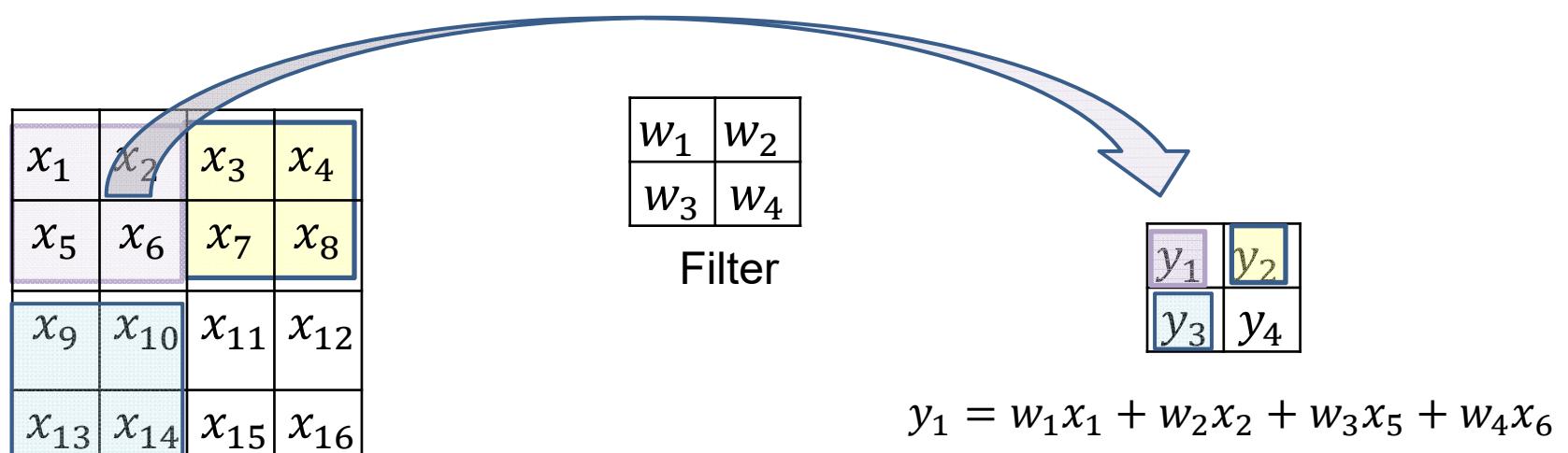
CNNs

Convolution: filter size and stride

- ▶ 2D convolution, stride 1, from 3×3 image to 2×2 image, 2×2 filter



- ▶ 2D convolution, stride 2, from 4×4 image to 2×2 image, 2×2 filter



CNNs Padding

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

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

CNNs

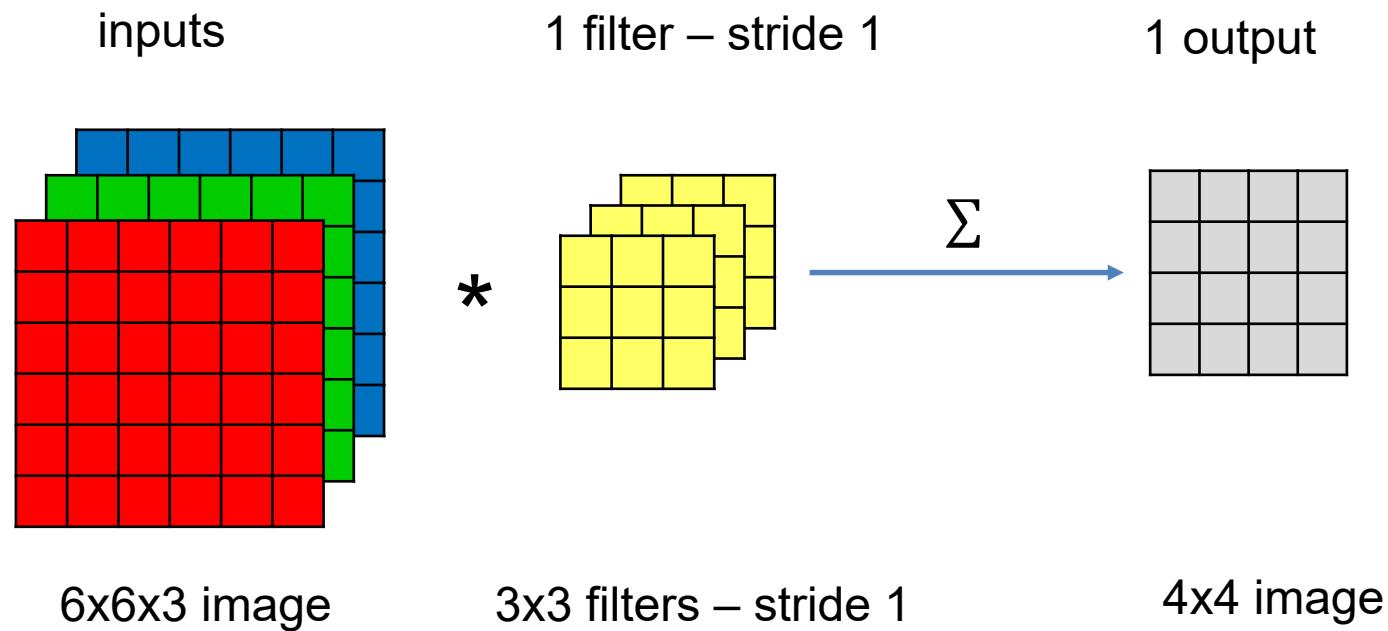
Convolutions arithmetics

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

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

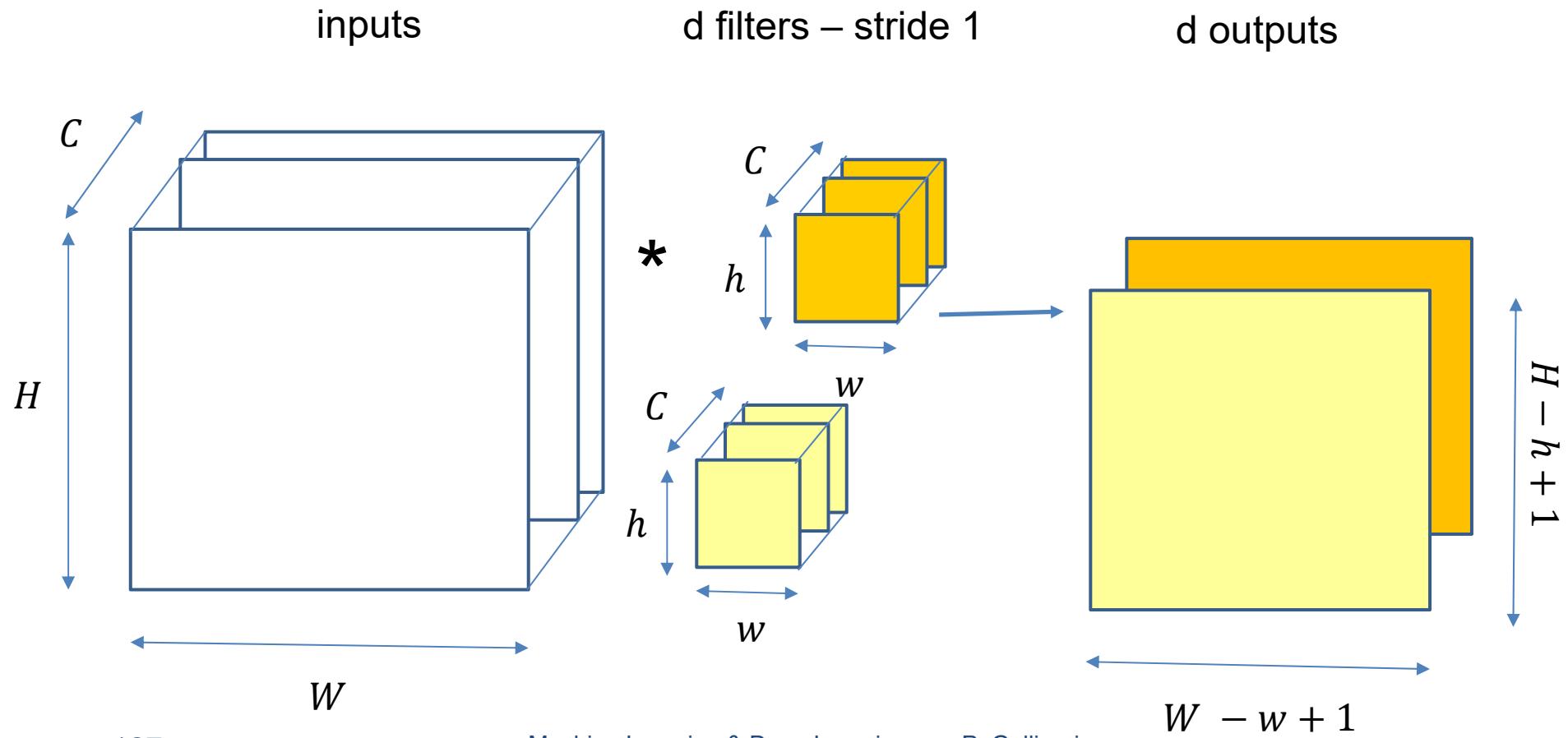
CNNs on multiple channels, e.g. RGB images

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



CNNs on multiple channels

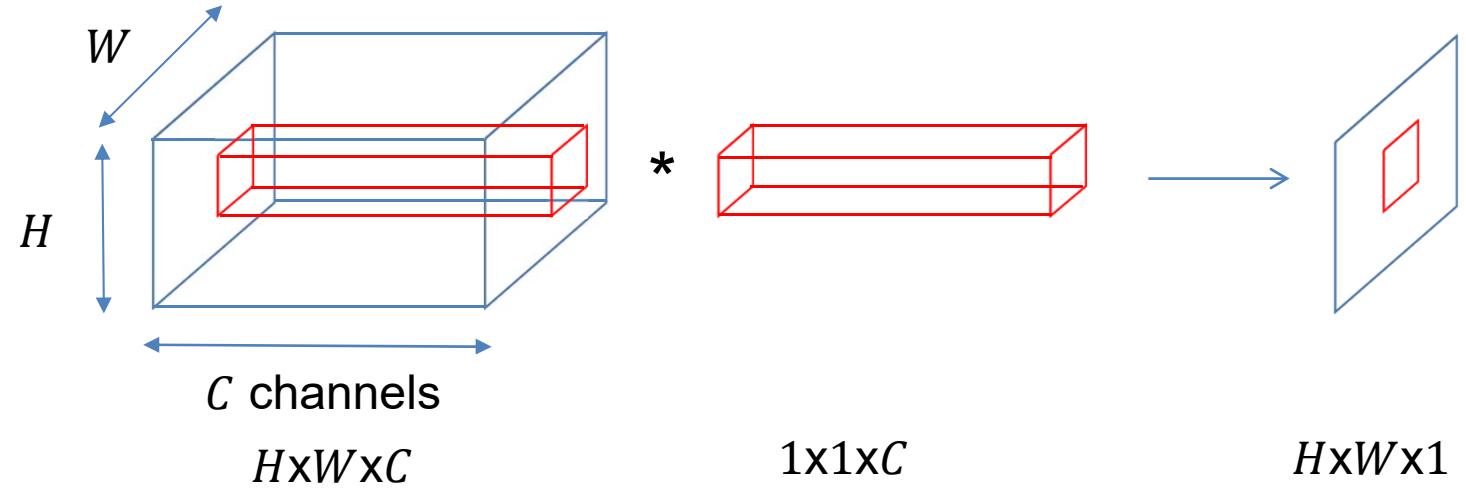
- ▶ This generalizes to any number of input channels, and filters
 - ▶ Below C input channels and 2 outputs



CNNs

1x1 convolutions on multiple channels

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

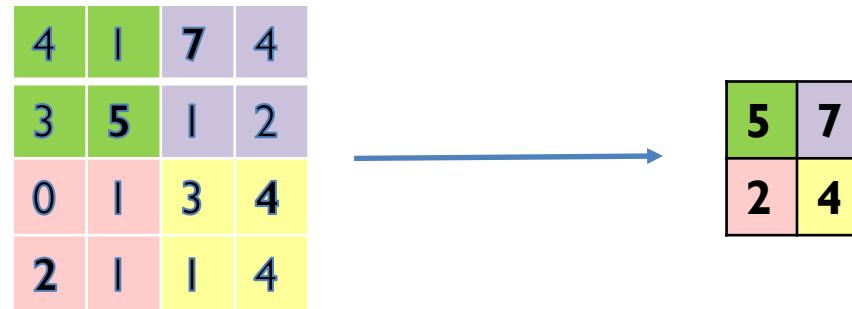


$C' = 1$ convolution in
this example

CNNs

Pooling

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

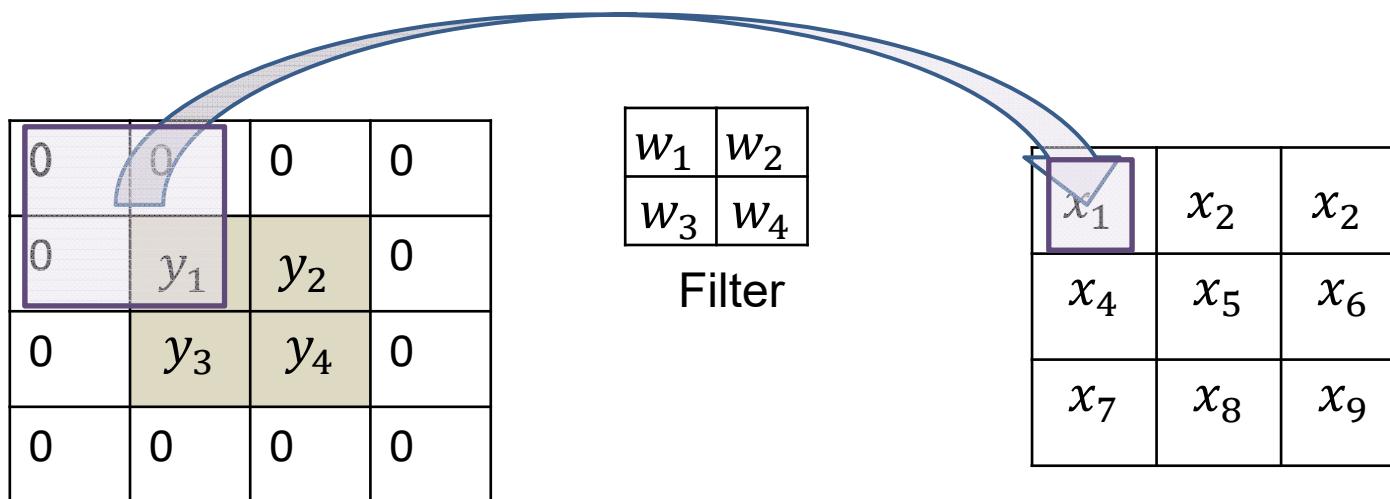


- ▶ Pooling provides some form of invariance to input deformations
- ▶ Pooling arithmetics

CNNs

Transpose convolution

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

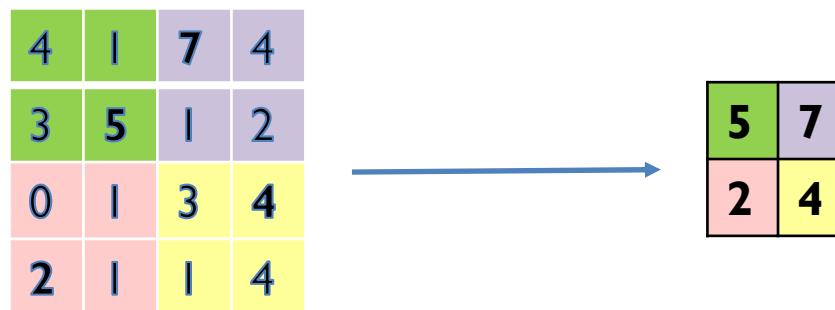


Note: more in (Dumoulin 2016), a guide to convolution arithmetic for Deep Learning

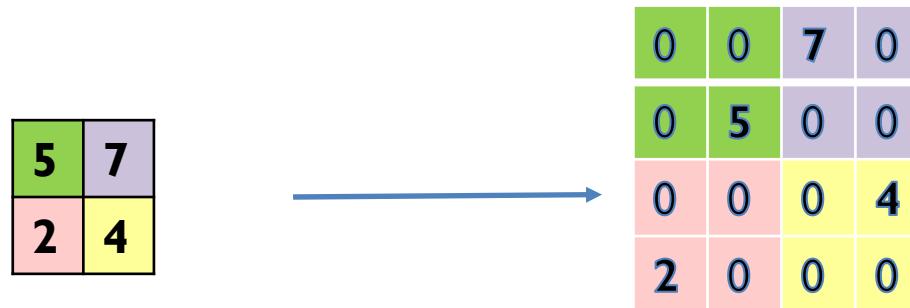
CNNs

Unpooling

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

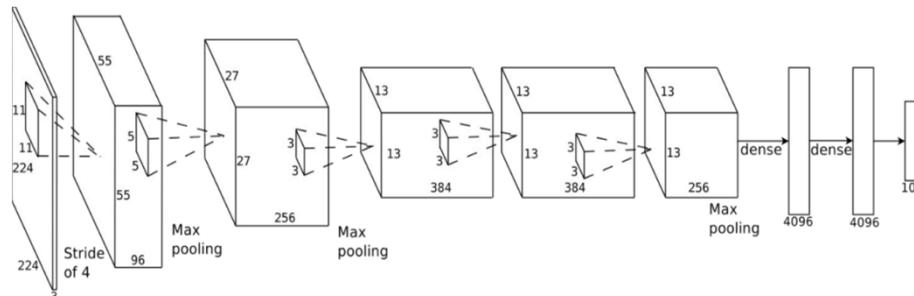


- ▶ Remember the positions of the max and fill the other positions with 0



CNNs–Classification (Krizhevsky et al. 2012)

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



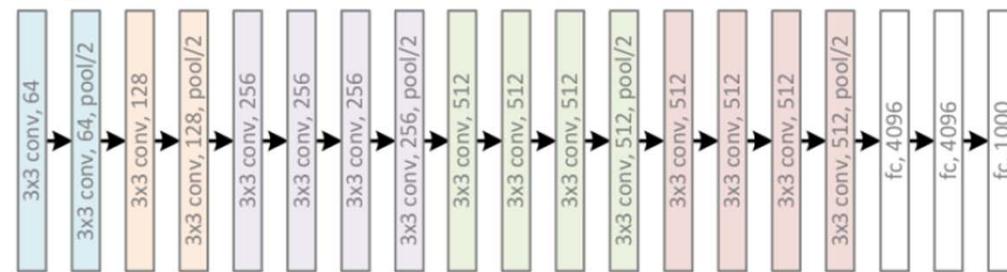
CNNs

Very Deep Nets trained with GPUs

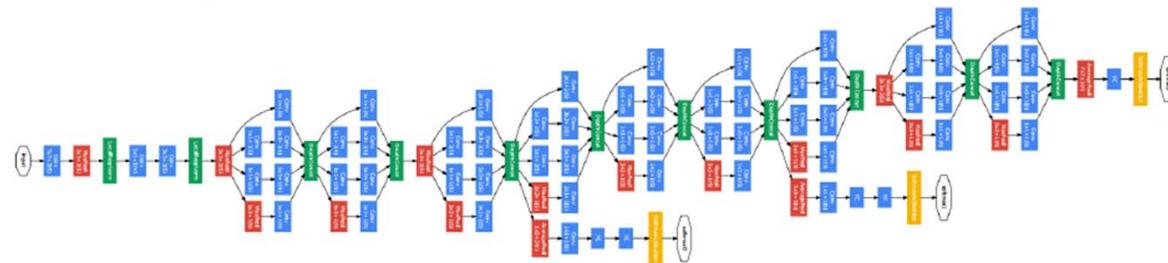
Deeper Nets with small filters – training time several days up to 1 or 2 weeks on ImageNet

Oxford, [Simonyan 2014], Parameters 138 M

VGG, 16/19 layers, 2014



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



ResNet, 152 layers, 2015

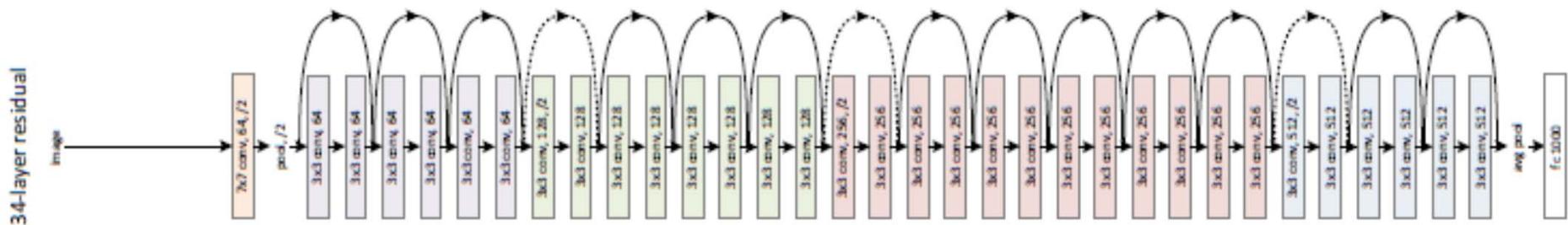
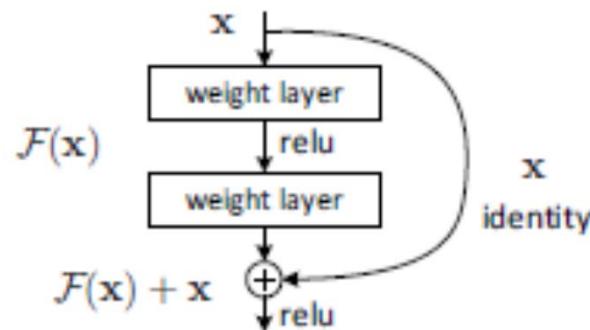


MSRA, [He et al. 2016] , Parameters 60 M

CNNs

ResNet [He et al. 2016]

- ▶ 152 ResNet 1st place ILSVRC classification competition
- ▶ Other ResNets 1st place ImageNet detection, 1st place ImageNet localization, MS-COCO detection and segmentation
- ▶ Main characteristics
 - ▶ Building block
 - ▶ Identity helps propagating gradients
 - ▶ Reduces the vanishing effect
 - ▶ $F(x)$ is called the residual
 - ▶ Similar ideas used in other models
 - ▶ Deep network with small convolution filters
 - ▶ Mainly 3x3 convolutional filters



CNNs

ResNet [He et al. 2016b]

▶ ResNet block

- ▶ $x_{t+1} = x_t + F(x_t, W_t)$
- ▶ $x_T = x_t + \sum_{i=t}^{T-1} F(x_i, W_i)$

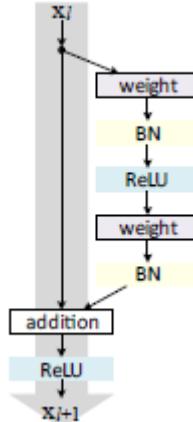


Fig. He 2016, original ResNet block

- ▶ The feature x_T on the last layer can be represented as the feature x_t of layer t plus a residual $\sum_{i=t}^{T-1} F(x_i, W_i)$

▶ ResNet Backward equation

- ▶ $\frac{\partial C}{\partial x_t} = \frac{\partial C}{\partial x_T} \frac{\partial x_T}{\partial x_t} = \frac{\partial C}{\partial x_T} \left(1 + \frac{\partial}{\partial x_t} \sum_{i=t}^{T-1} F(x_i, W_i) \right)$
- ▶ Gradient $\frac{\partial C}{\partial x_t}$ can be decomposed in two additive term
 - ▶ $\frac{\partial C}{\partial x_T}$ propagates this gradient to any unit
 - ▶ $\frac{\partial}{\partial x_t} \sum_{i=t}^{T-1} F(x_i, W_i)$ propagates through the weight layers

CNNs

ResNet as a discretization scheme for ODEs (Optional)

- ▶ Ordinary Differential Equation

- ▶ $\frac{dX}{dt} = F(X(t), \theta(t)), X(0) = X_0$ (1)

- ▶ Resnet module can be interpreted as a numerical discretization scheme for the ODE:

- ▶ $X_{t+1} = X_t + G(X_t, \theta_t)$ - ResNet module (2)
 - ▶ $X_{t+1} = X_t + hF(X_t, \theta_t), h \in [0,1]$ (simple rewriting of (2) replacing $G()$ with $hF()$)
 - ▶ $\frac{X_{t+1} - X_t}{h} = F(X_t, \theta_t)$
 - ▶ Forward Euler Scheme for the ODE (1)
 - ▶ h time step
 - ▶ Note: this type of additive structure (2) is also present in LSTM and GRU units (see RNN section)

- ▶ Resnet

- ▶ Input X_t , output X_{t+1}
 - ▶ Multiple Resnet modules implement a discretization scheme for the ODE
 - ▶ $X(t_1) = X(t_0) + hF(X(t_0), \theta_{t_0})$
 - ▶ $X(t_2) = X(t_1) + hF(X(t_1), \theta_{t_1}), \dots$

CNNs

Resnet as a discretization scheme for ODEs

- ▶ This suggests that alternative discretization schemes will correspond to alternative Resnet like NN models
 - ▶ Backward Euler, Runge-Kutta, linear multi-step ...
- ▶ Example (Lu 2018) linear multi-step discretization scheme
 - ▶ $X_{t+1} = (1 - k_t)X_t + k_t X_{t-1} + F(X_t, \theta_t)$

Fig. (Lu 2018)

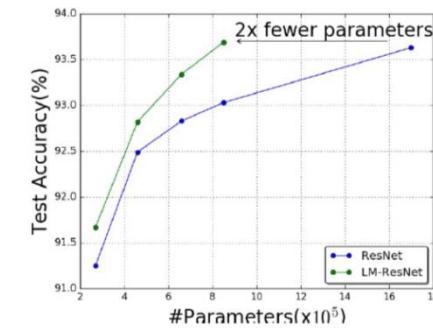
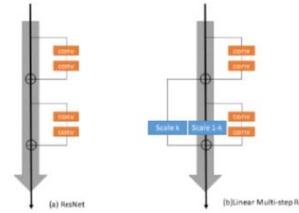


Figure 2: LM-architecture is an efficient structure that enables ResNet to achieve same level of accuracy with only half of the parameters on CIFAR10.

- ▶ Applications
 - ▶ Classification (a la ResNet)
 - ▶ Modeling dynamical systems

Convolutional Nets

ILSVRC performance over the years

- Imagenet 2012 classification challenge

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

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

- ImageNet 2013 – image classification challenge

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

- ImageNet 2013 – object detection challenge

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

- ImageNet 2013 – image classification challenge

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

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

- ImageNet 2013 – object detection challenge

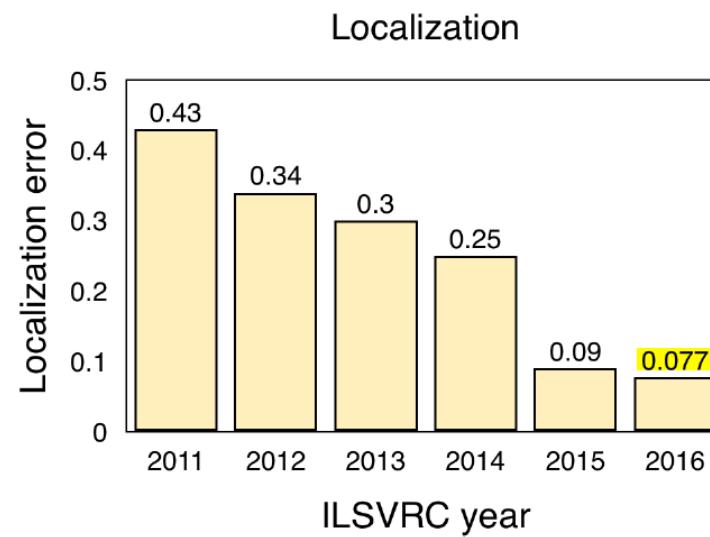
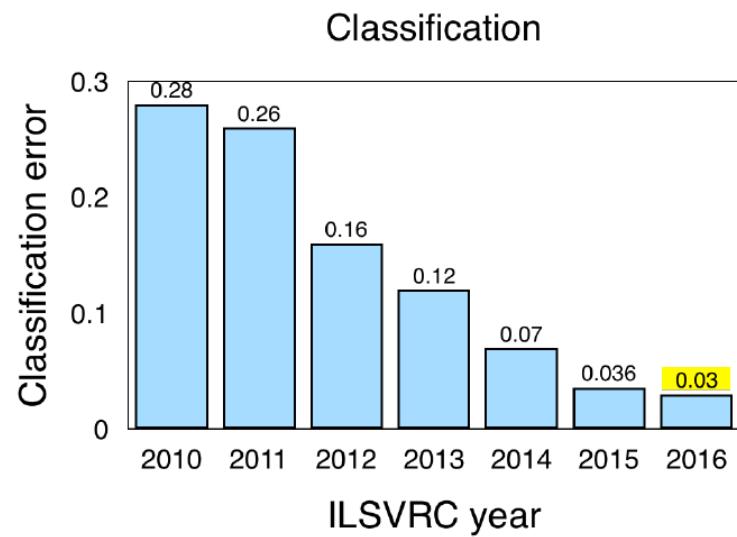
Rank	Name	Mean Average Precision	Description
1	UvA-Euvision	0.22581	Hand-crafted features
2	NEC-MU	0.20895	Hand-crafted features
3	NYU	0.19400	Deep learning

CNN examples



Convolutional Nets

ILSVRC performance over the years



Classification

CNNs and Transfer Learning

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

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



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

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

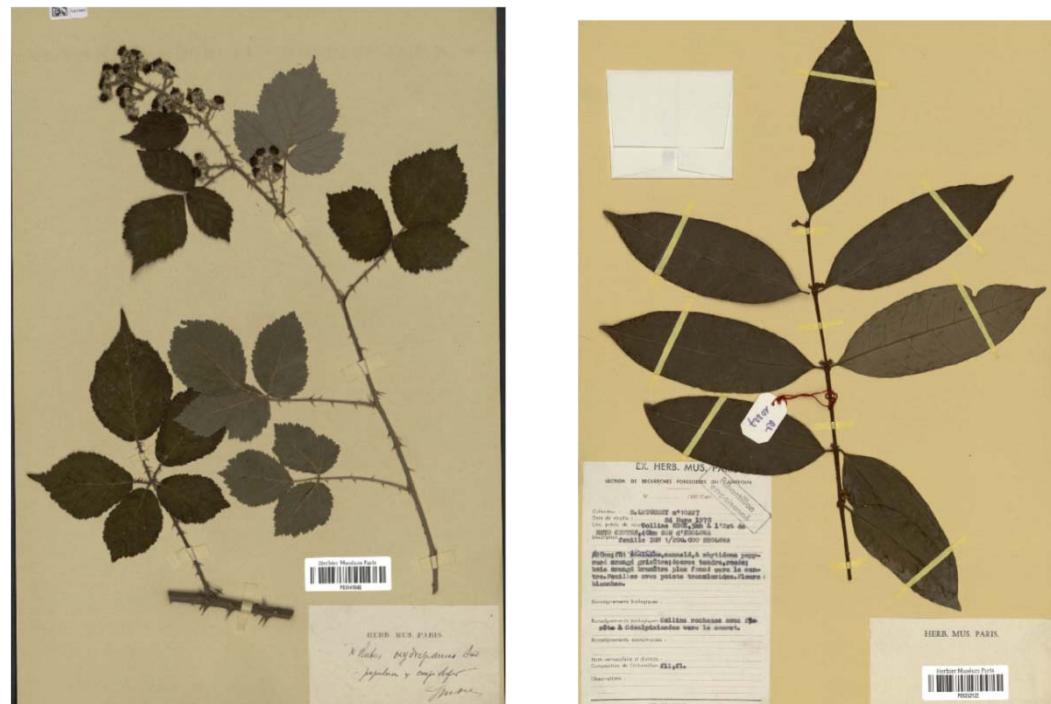
Table 1: Forward+Backward with batches of 20 images.

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

Table 2: Accuracy on the validation set.

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

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



Machine Learning & Deep Learning - P. Gallinari

Classification - Fully convolutional nets

CNNs – Classification of large images (Durand 2016)

How to deal with complex scenes?

Pascal VOC style

ImageNet style



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



VOC07/12

MIT67

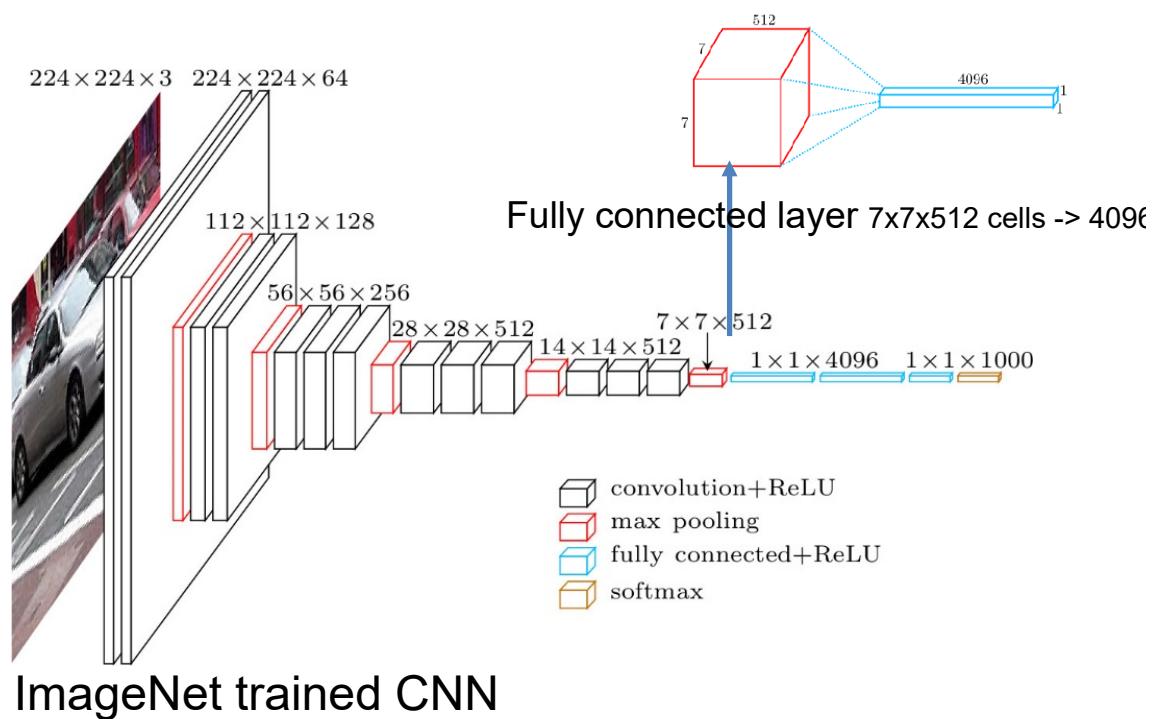
15 Scene

COCO

VOC12 Action

Classification - CNNs – Classification of large images (Durand 2016)

Sliding window => Convolutional Layers

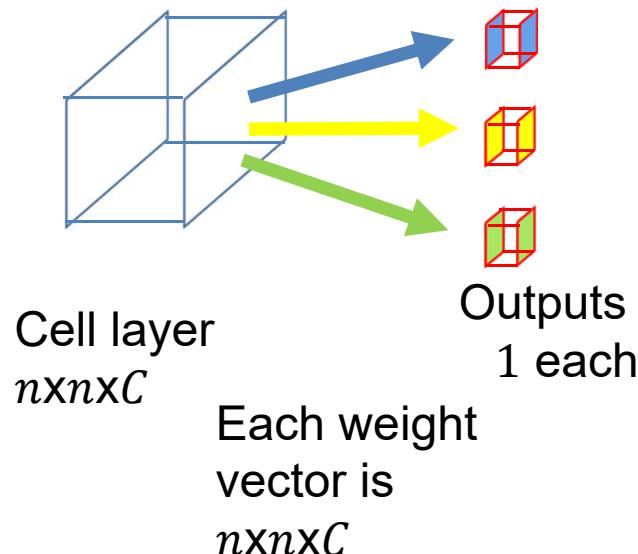


- ▶ Sliding window:
 - ▶ Use the ImageNet trained CNN as a sliding window (a convolution filter) on the large image
 - ▶ In order to do that, one must **convert the fully connected layer $7 \times 7 \times 512$ cells $\rightarrow 4096$ cells to a convolutional layer**

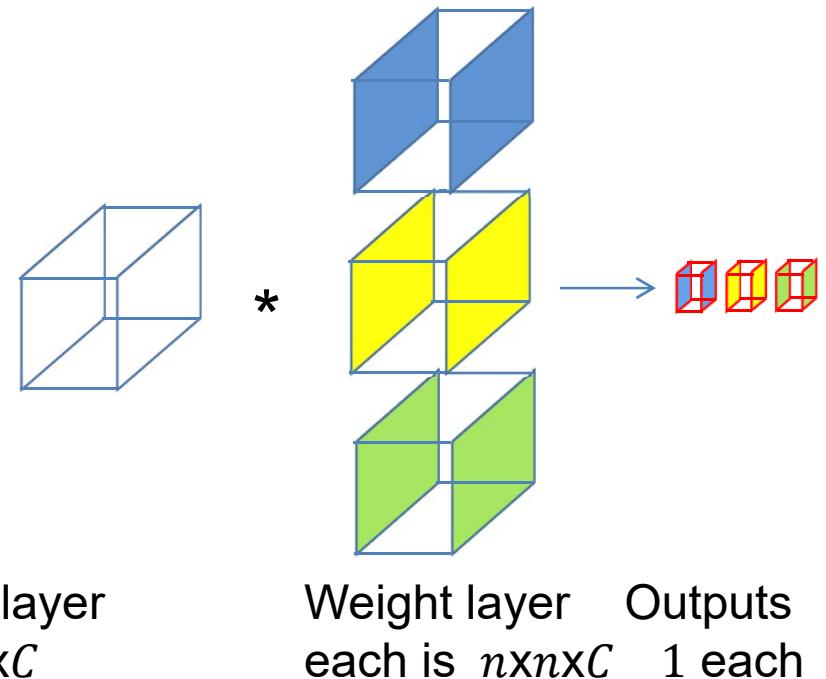
Converting Fully Convolutional Nets (FCN) to CNN

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

FCN classical view

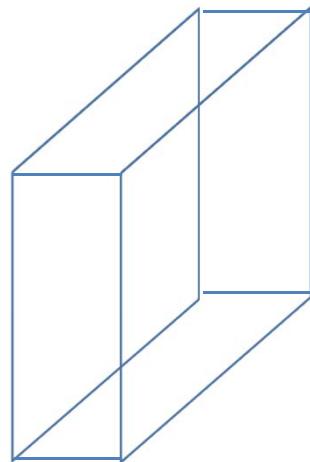


FCN convolutional view



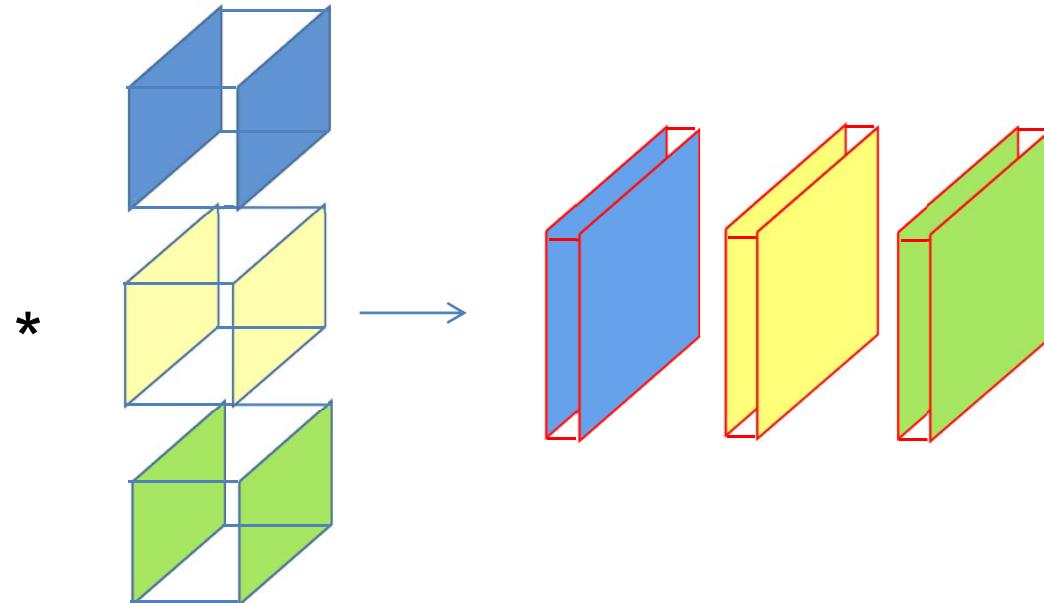
Converting Fully Convolutional Nets (FCN) to CNN

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



Cell layer
 $N \times N \times C$

146

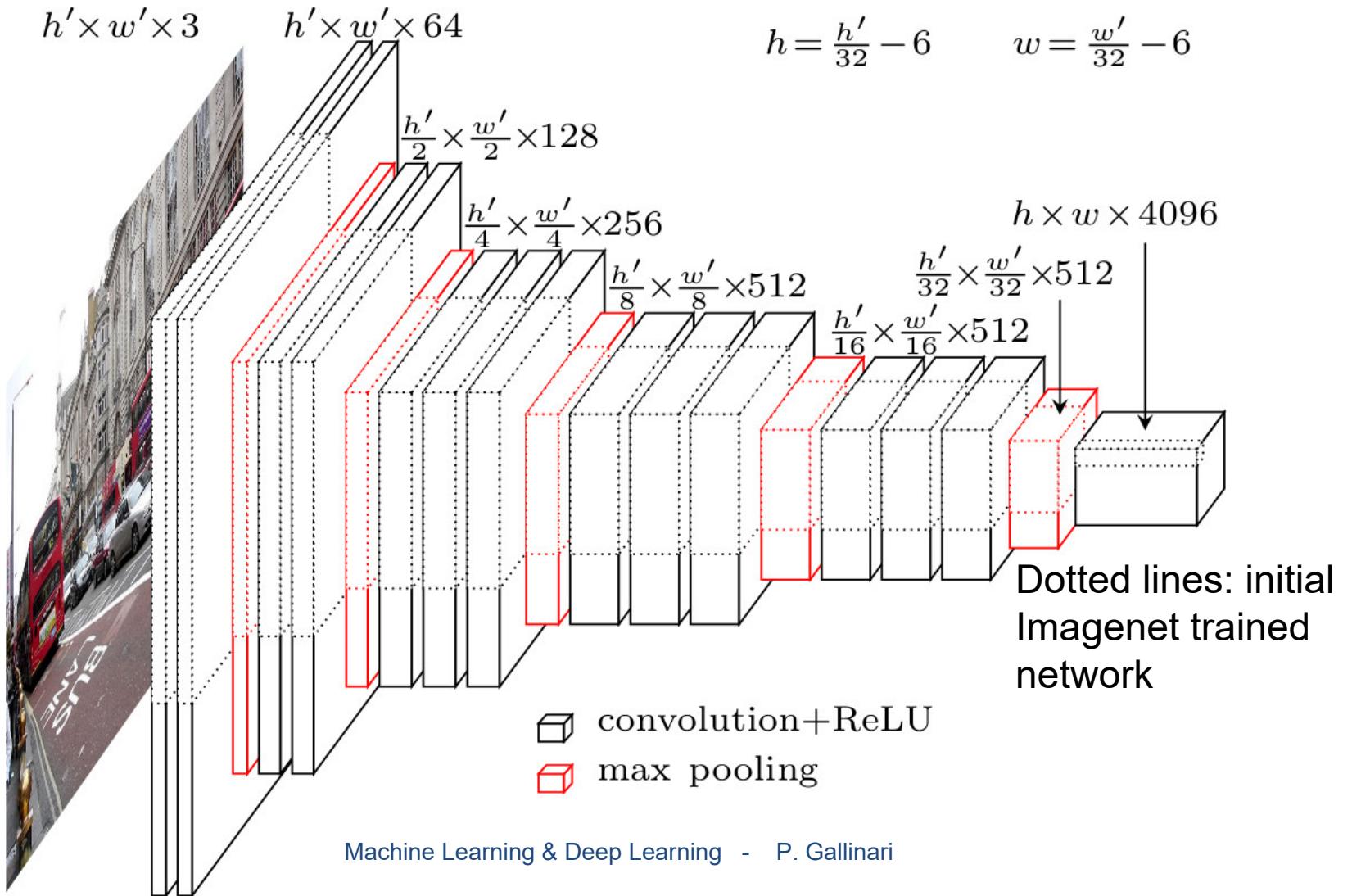


Weight layer
 $n \times n \times C$ each

Outputs
 $(N - n + 1) \times (N - n + 1) \times 1$ each

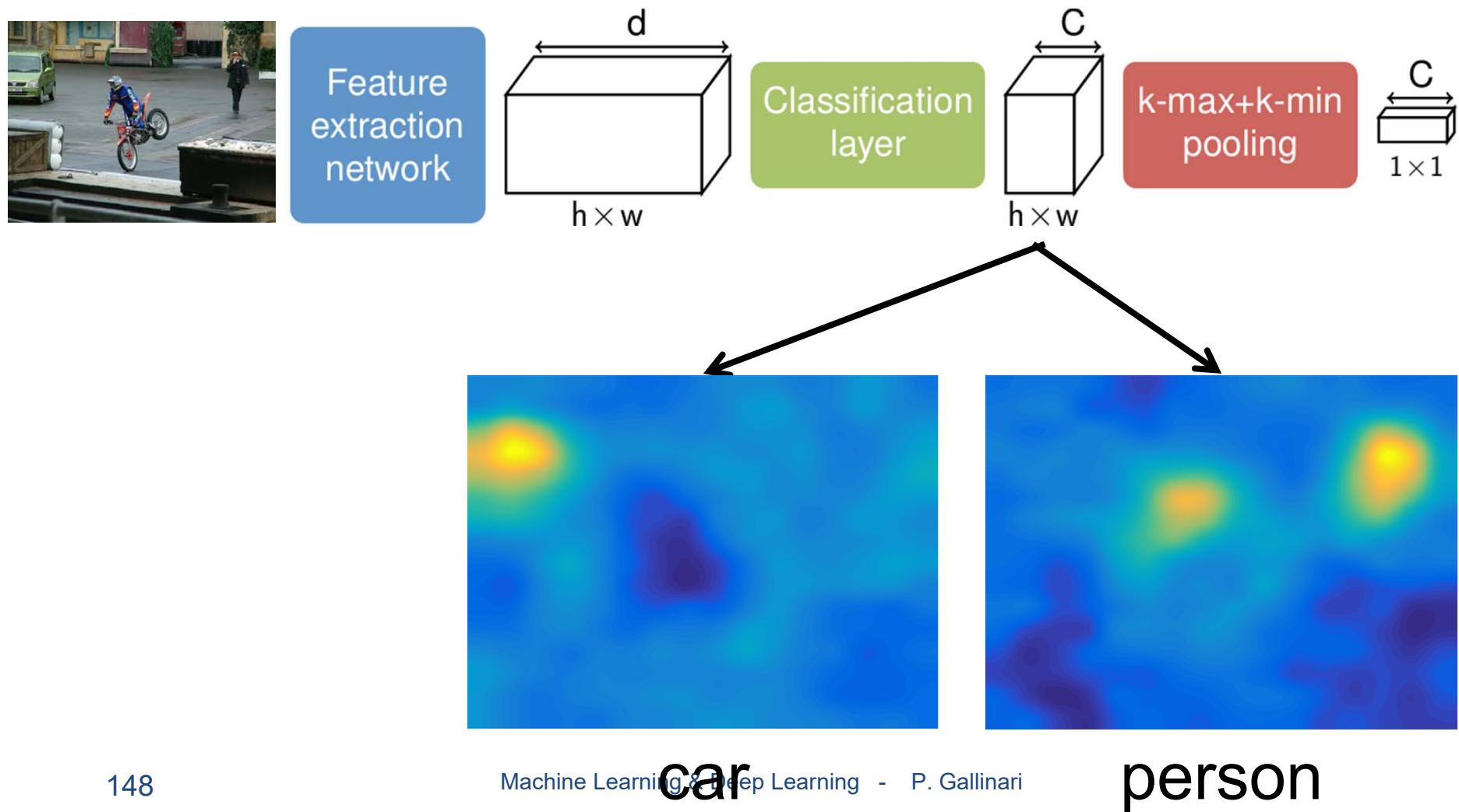
CNNs – Classification of large images (Durand 2016)

Sliding window => Convolutional Layers



CNNs – Classification of large images (Durand 2016)

Sliding window => Convolutional Layers



CNN : A neural algorithm of Artistic Style

(Gatys et al. 2016)

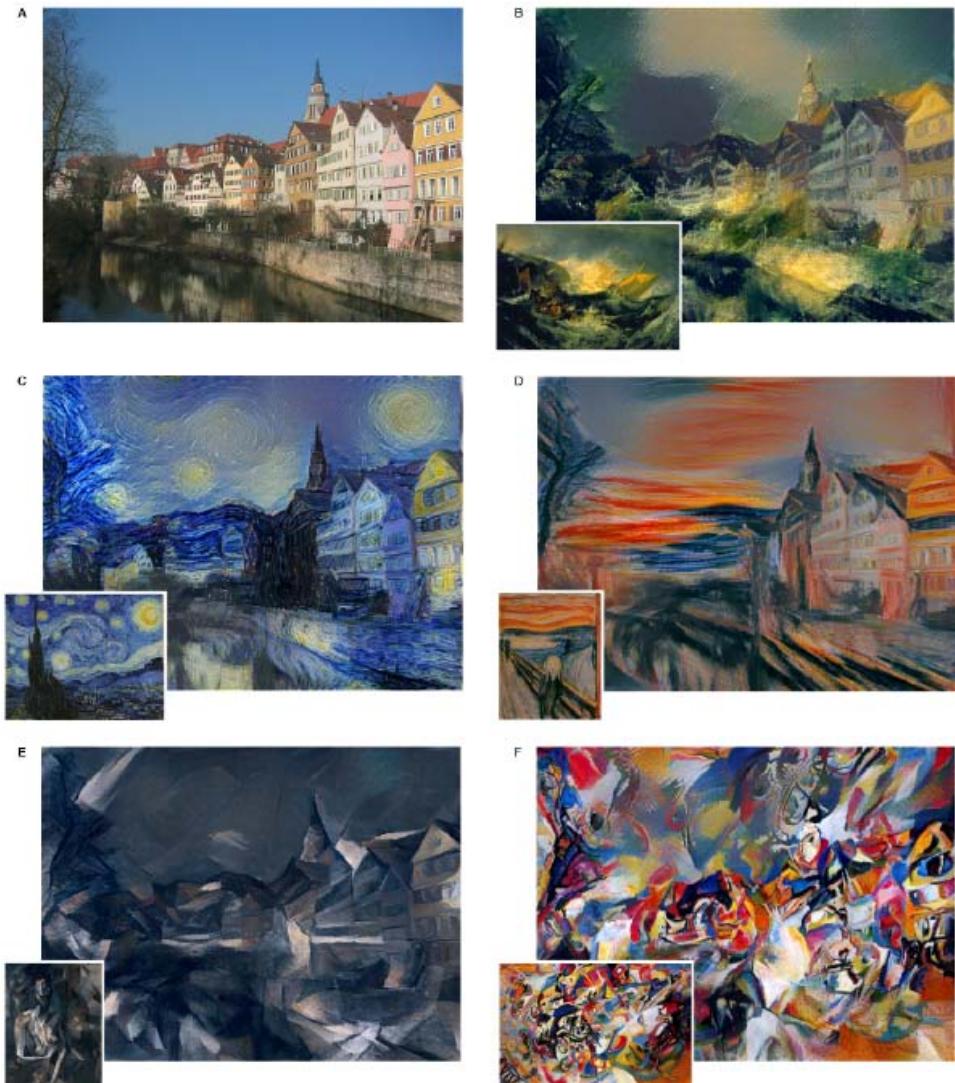
Generate images by combining content and style

Makes use of a discriminatively trained CNN

Image generation

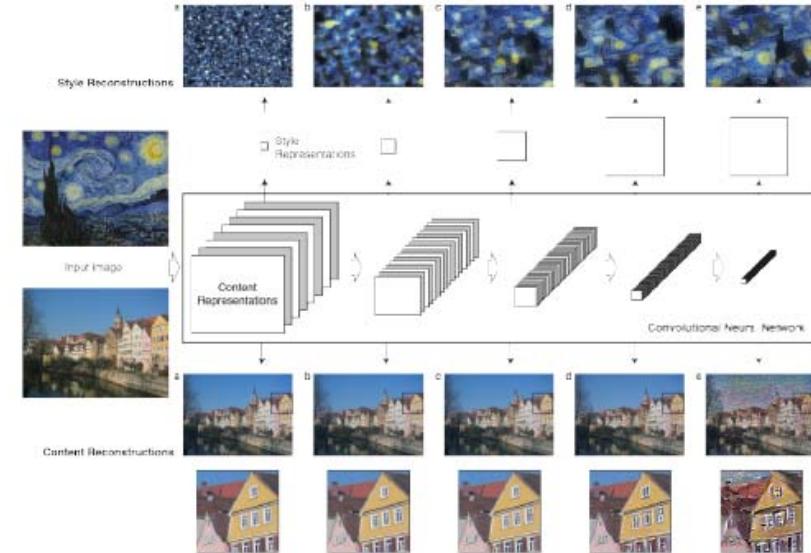
- ▶ inverse problem on the CNN

<https://deepart.io>



CNN : A neural algorithm of Artistic Style (Gatys et al. 2016)

- ▶ Idea (simplified)
 - ▶ Use a pre-trained ImageNet NN
 - ▶ c input content image, F_c a filter representation of c
 - ▶ a input art image, G_a a filter correlation representation of a
 - ▶ x a white noise image, F_x and G_x the corresponding filter and filter correlation representations
 - ▶ loss:
 - ▶ $L = \|F_c - F_x\|^2 + \alpha \|G_a - G_x\|^2$
- ▶ Generated image
 - ▶ Solve an inverse problem
 - ▶ $\hat{x} = \operatorname{argmin}_x(L)$
 - ▶ Solved by gradient



CNN : A neural algorithm of Artistic Style (Gatys et al. 2016)

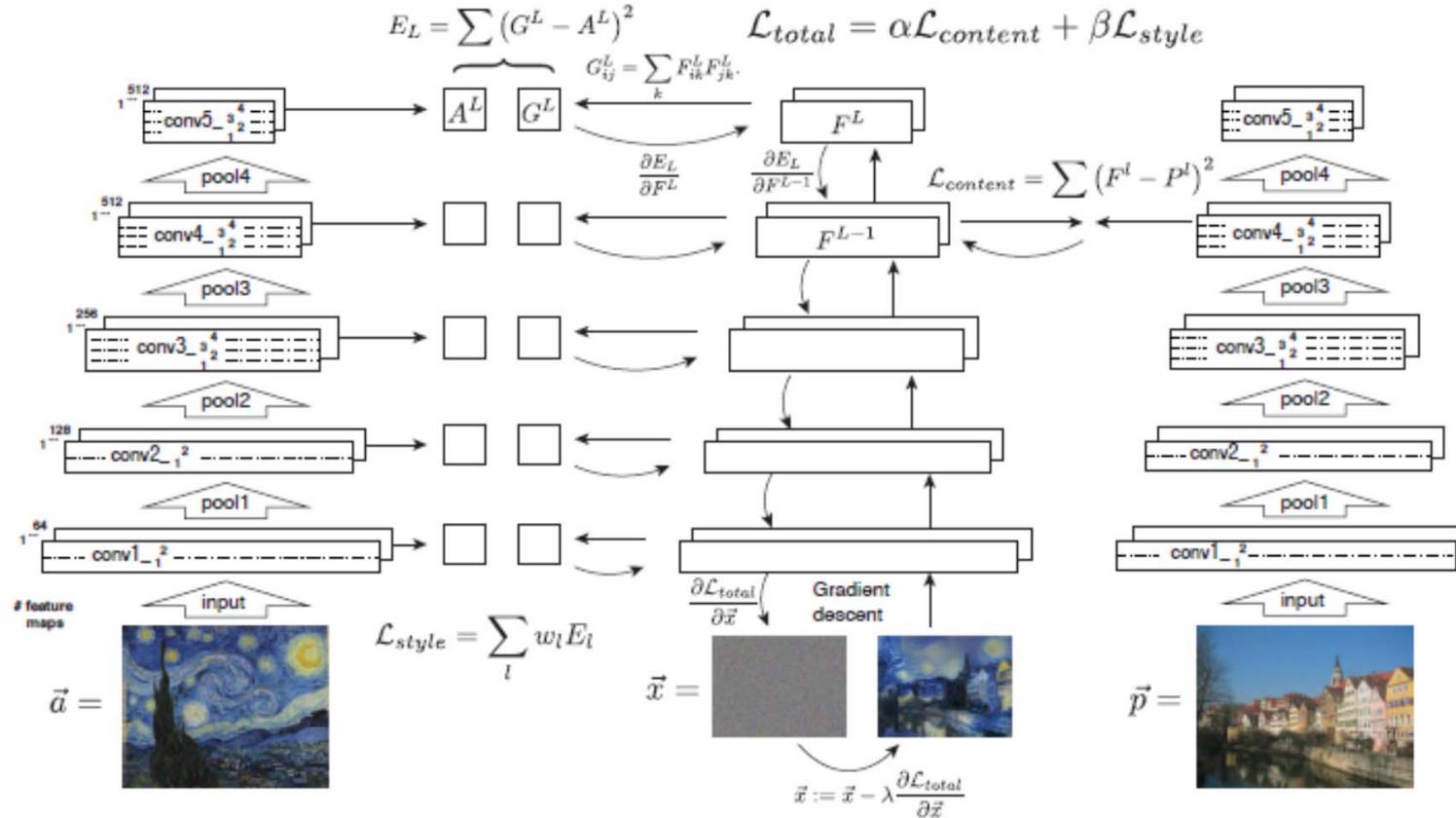


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).

Object detection

- ▶ Objective: predicting classes and location of objects in an image
 - ▶ Usually the output of the predictor is a series of bounding boxes with an object class label
- ▶ Performance measure
 - ▶ Let B a target bounding box and \hat{B} the predicted one
 - ▶ Intersection over Union: $IoU = \frac{\text{area}(B \cap \hat{B})}{\text{area}(B \cup \hat{B})}$
- ▶ Training
 - ▶ Supervised training, e.g. Pascal Voc Dataset



```
# PASCAL Annotation Version 1.00 Image filename :  
"TUDarmstadt/PNGImages/motorbike-testset/motorbikes040-rt.png"  
Image size (X x Y x C) : 400 x 275 x 3  
Database : "The TU Darmstadt Database"  
Objects with ground truth : 2 { "PASmotorbikeSide" "PASmotorbikeSide" }  
# Note that there might be other objects in the image # for which ground truth data has  
not been provided.  
# Top left pixel co-ordinates : (1, 1)  
# Details for object 1 ("PASmotorbikeSide")  
Original label for object 1 "PASmotorbikeSide" : "motorbikeSide"  
Bounding box for object 1 "PASmotorbikeSide" (Xmin, Ymin) - (Xmax, Ymax) : (57, 133)  
- (329, 265)  
# Details for object 2 ("PASmotorbikeSide")  
Original label for object 2 "PASmotorbikeSide" : "motorbikeSide"  
Bounding box for object 2 "PASmotorbikeSide" (Xmin, Ymin) - (Xmax, Ymax) : (153, 95)  
(396, 218)
```

CNNs for Object detection

Case study: YOLO (Redmon 2015), <https://goo.gl/bEs6Cj>

- ▶ Classical CNN architecture
- ▶ Divides the input image into a $S \times S$ grid
 - ▶ Each grid cell predicts
 - ▶ B bounding boxes and confidence for these boxes
 - 5 numbers per box: (x, y) : box center, (w, h) : box dimension, confidence
 - $\text{confidence} = P(\text{Object}).\text{IoU}(\text{target}, \text{pred})$
 - $P(\text{Object})$ is the probability that an object appears in a grid cell
 - ▶ The class probability for the object if any (only one object/ cell grid), i.e. 1 prediction / cell
 - $P(\text{Class}|\text{Object})$
 - Note: at inference time they use the following score
 - $P(\text{Class}|\text{object}).P(\text{Object}).\text{IoU}(\text{target}, \text{pred})$ instead of $P(\text{Class}|\text{Object})$
 - ▶ This includes confidence
 - Only the boxes/classes with the higher score are kept

CNNs for Object detection

Case study: YOLO (Redmon 2015)



Fig. Redmon 2015

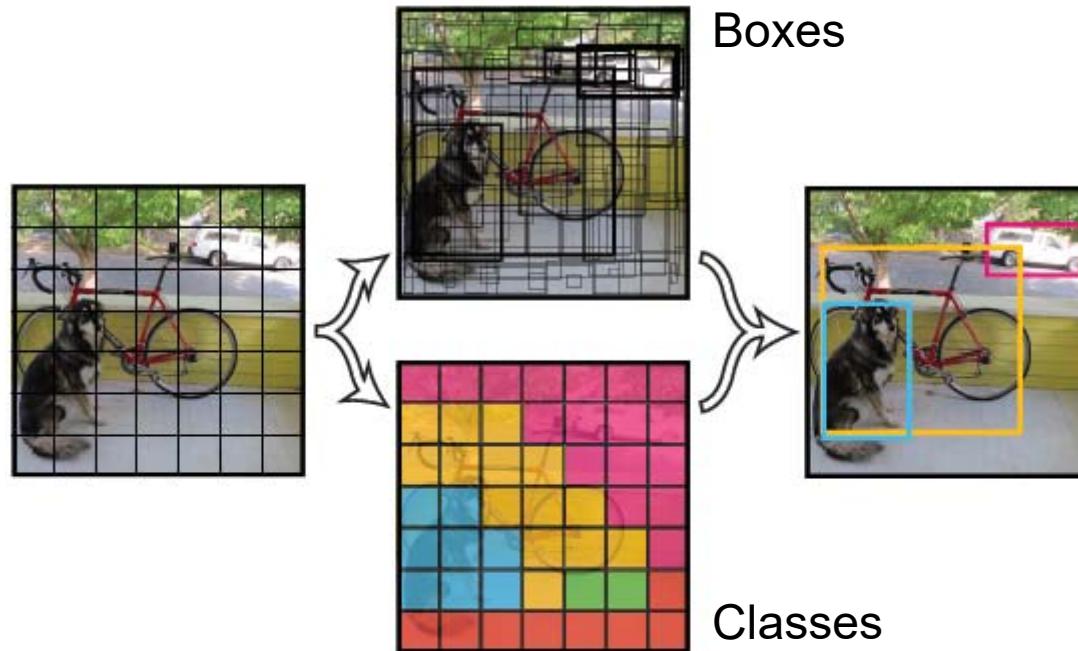


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an even grid and simultaneously predicts bounding boxes, confidence in those boxes, and class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

CNNs for Object detection

Case study: YOLO (Redmon 2015) - Network Design

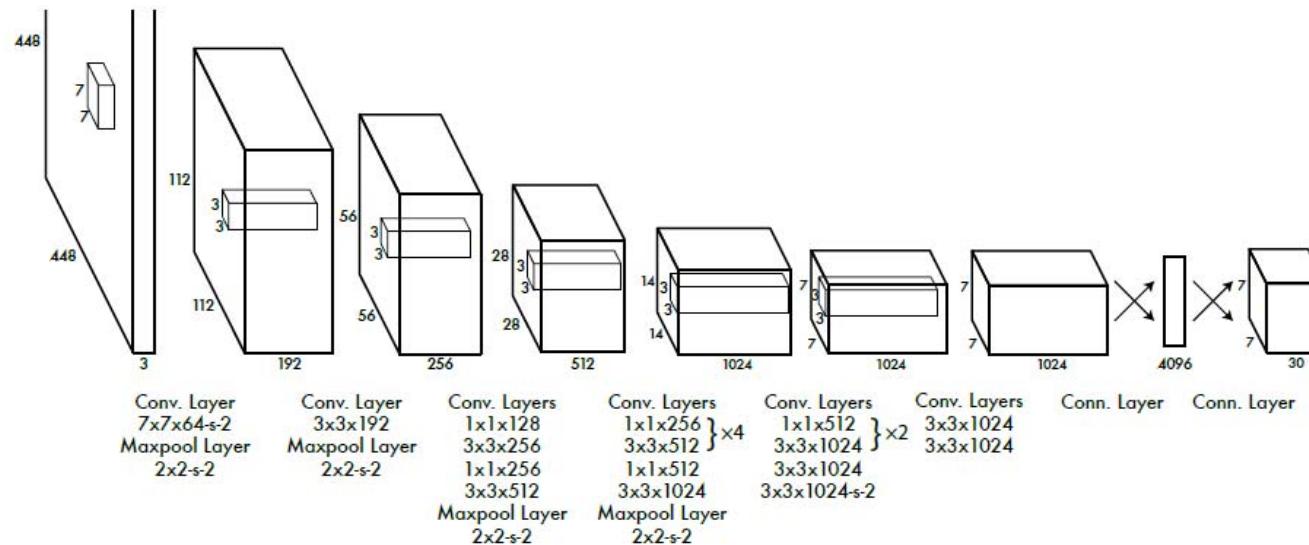


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

Output : $S \times S \times (B \times 5 + C)$ tensor

for Pascal Voc dataset: $S \times S \times (B \times 5 + C) = 7 \times 7 \times (2 \times 5 + 20)$

With B : # boxes and C : # classes

Several $1 \times 1 \times n$ convolutional structures to reduce the feature space from preceding layers

CNNs for Object detection

Case study: YOLO (Redmon 2015) - Design and Training

- ▶ Pretrained on ImageNet 1000 class
- ▶ Remove classification layer and replace it with 4 convolutional layers + 2 Fully Connected layers
- ▶ Activations: Linear for the last layer, leaky reLu for the others
- ▶ Requires a lot of know-how (design, training strategy, tricks, etc)
 - ▶ Not described here – see paper...
- ▶ Improved versions followed the initial paper
- ▶ Generalizes to other types of images:

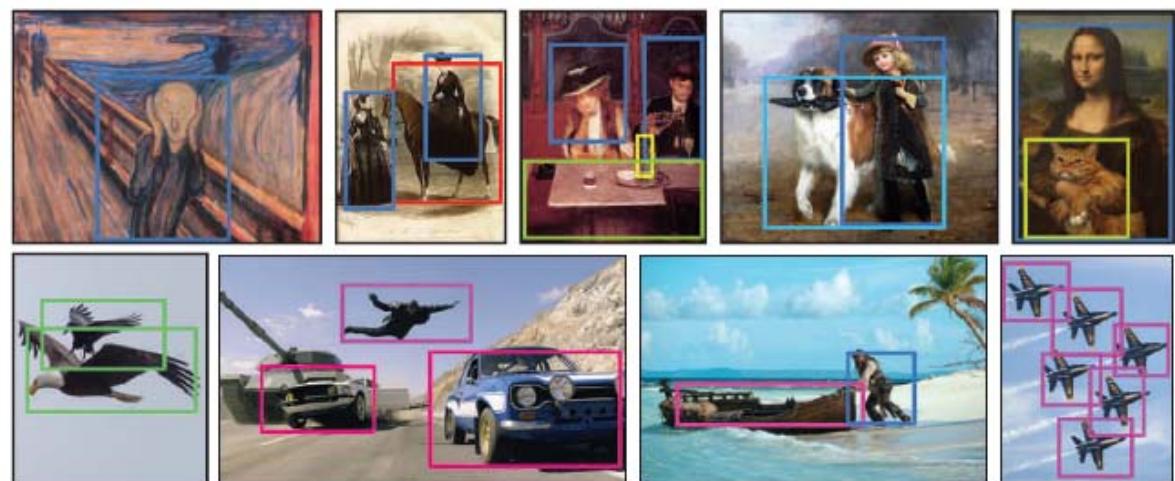
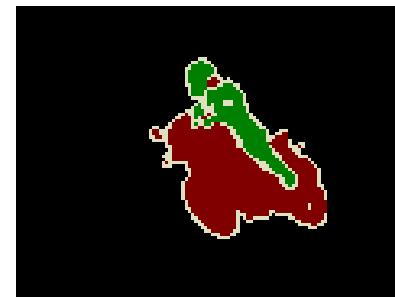


Image Semantic Segmentation

- ▶ **Objective**
 - ▶ Identify the different objects in an image



- ▶ **Deep learning**
 - ▶ handles segmentation as pixel classification
 - ▶ re-uses network trained for image classification by making them fully convolutional
 - ▶ Currently, SOTA is Deep Learning
- ▶ **Main datasets**
 - ▶ **Voc2012**, <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/>
 - ▶ **MSCOCO**, <http://mscoco.org/explore/>

CNNs for Image Semantic Segmentation

- ▶ DL for segmentation massively re-uses CNN architectures pretrained for classification
 - ▶ This is another example of transfer learning
 - ▶ Here the goal is to generate classification **at the pixel level** and not at the global image level
 - ▶ Means that the output should be the same size (more or less) as the original image, with each pixel labeled by an object Id.
 - ▶ Full connections: too many parameters
 - How to keep a pixelwise precision with a low number of parameters
 - ▶ Two solutions have been developed
 - ▶ Encoder – Decoder architectures with skip connections
 - Encoder are similar to the ones used for classification and decoders use Transpose Convolutions and Unpooling
 - ▶ Dilated or atrous convolutions : remove the Pooling/Unpooling operation

CNNs for Image Semantic Segmentation

Encoder-Decoder - Fully Convolutional Nets (Shelhamer 2016)

- ▶ One of the first contribution to DL semantic segmentation, introduces several ideas
- ▶ Auto-encoder with skip connections

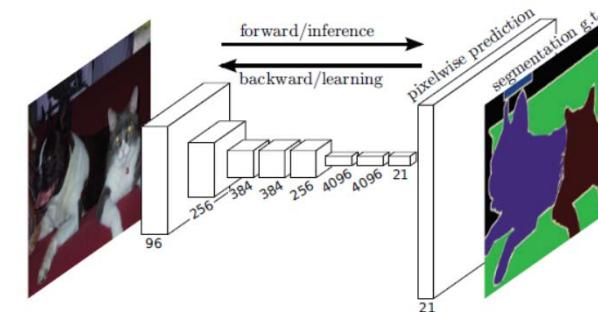


Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation

- ▶ Fully connected -> convolutional trick

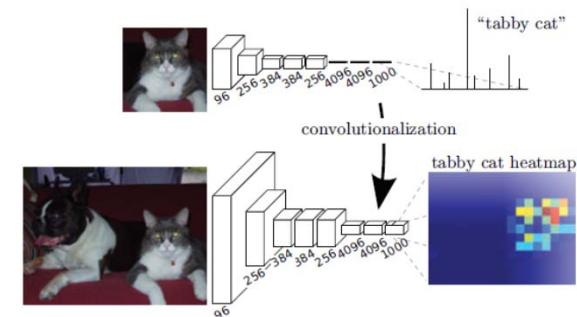


Figure 2. Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss (as in Figure 1) produces an efficient machine for end-to-end dense learning.

CNNs for Image Semantic Segmentation

Encoder-Decoder - Fully Convolutional Nets (Shelhamer 2016)

- ▶ FCN architecture: **upsampling** and **skip connections**
 - ▶ Training loss = per pixel cross entropy
 - ▶ Their initial pipeline (red rectangle) requires $\times 32$ upsampling
 - ▶ Improved results were obtained by combining several resolutions in the DNN

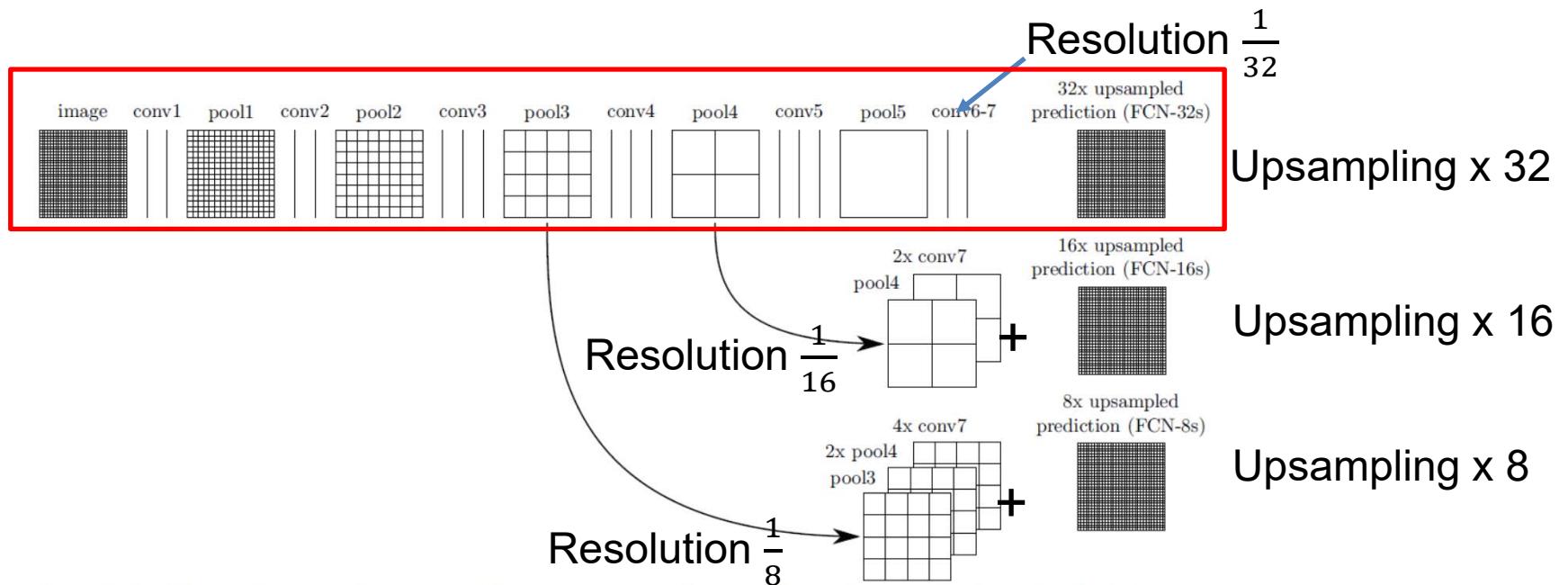


Figure 3. Our DAG nets learn to combine coarse, high layer information with fine, low layer information. Pooling and prediction layers are shown as grids that reveal relative spatial coarseness, while intermediate layers are shown as vertical lines. First row (FCN-32s): Our single-stream net, described in Section 4.1, upsamples stride 32 predictions back to pixels in a single step. Second row (FCN-16s): Combining predictions from both the final layer and the pool4 layer, at stride 16, lets our net predict finer details, while retaining high-level semantic information. Third row (FCN-8s): Additional predictions from pool3, at stride 8, provide further precision.

Segmentation

Encoder-Decoder - Other models based on the same ideas

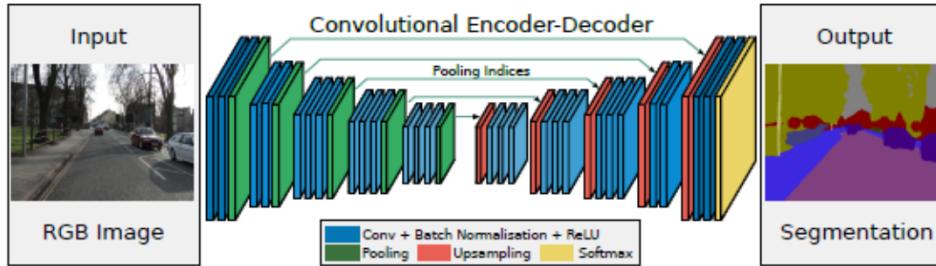
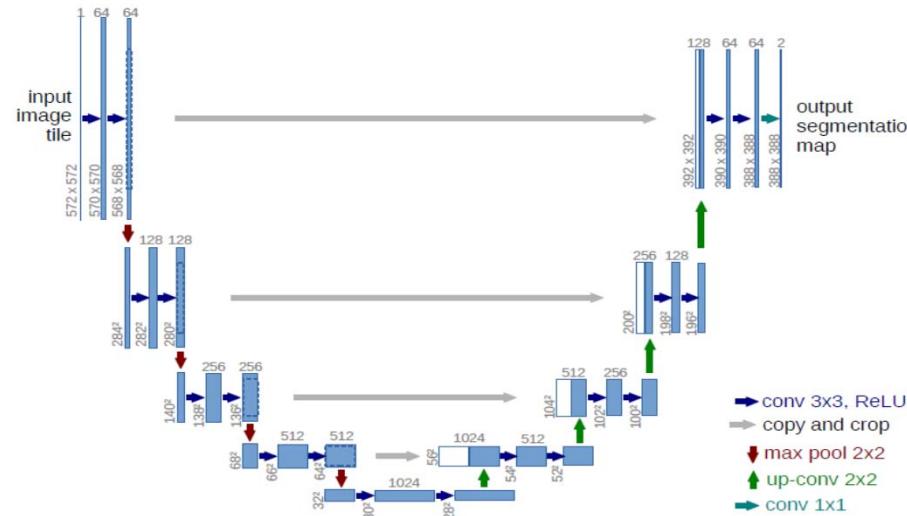


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

SegNet – (Badrinarayanan 2017)



U-Net, (Ronneberger 2015)

Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

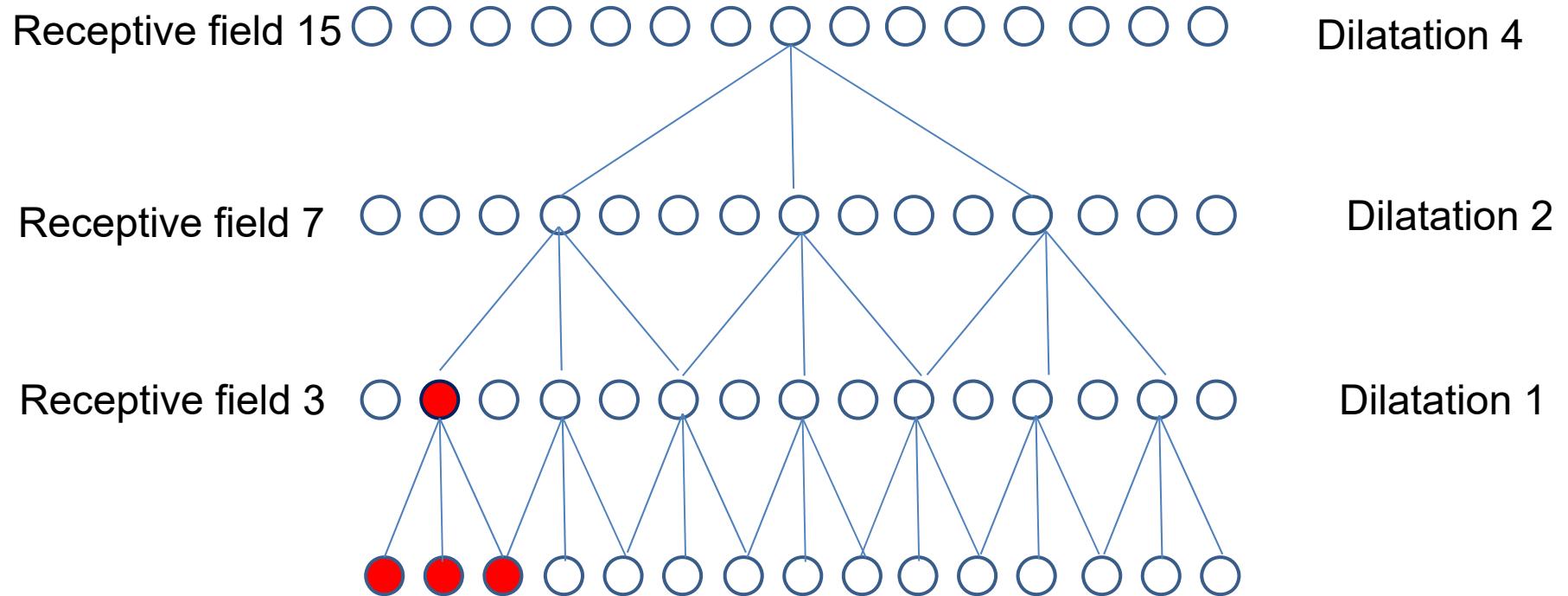
Segmentation

Dilated convolutions (Yu 2016)

- ▶ Pooling used for classification is not adapted to segmentation
 - ▶ The link with individual pixels is lost
- ▶ Proposed method
 - ▶ Start from a Deep CNN trained from classification.
 - ▶ Remove the last Fully Connected and Pooling layers
 - ▶ Replace them with Dilated Convolution layers
 - ▶ Dilated convolution layers organized hierarchically allow to keep large feature maps for individual neurons with a « small » number of connections
 - ▶ Size of the input is the same as the size of the output
 - No downsampling as with pooling, i.e. keep the resolution

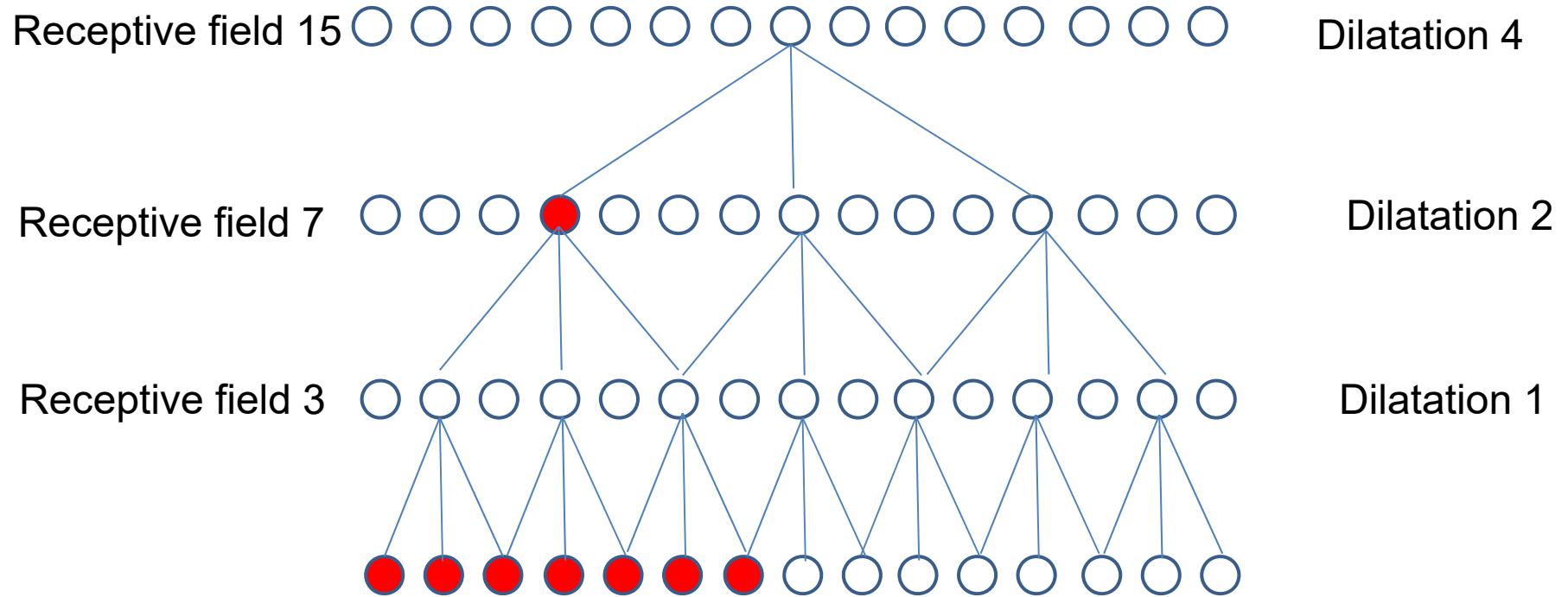
Segmentation Dilated convolutions (Yu 2016)

► 1 D example



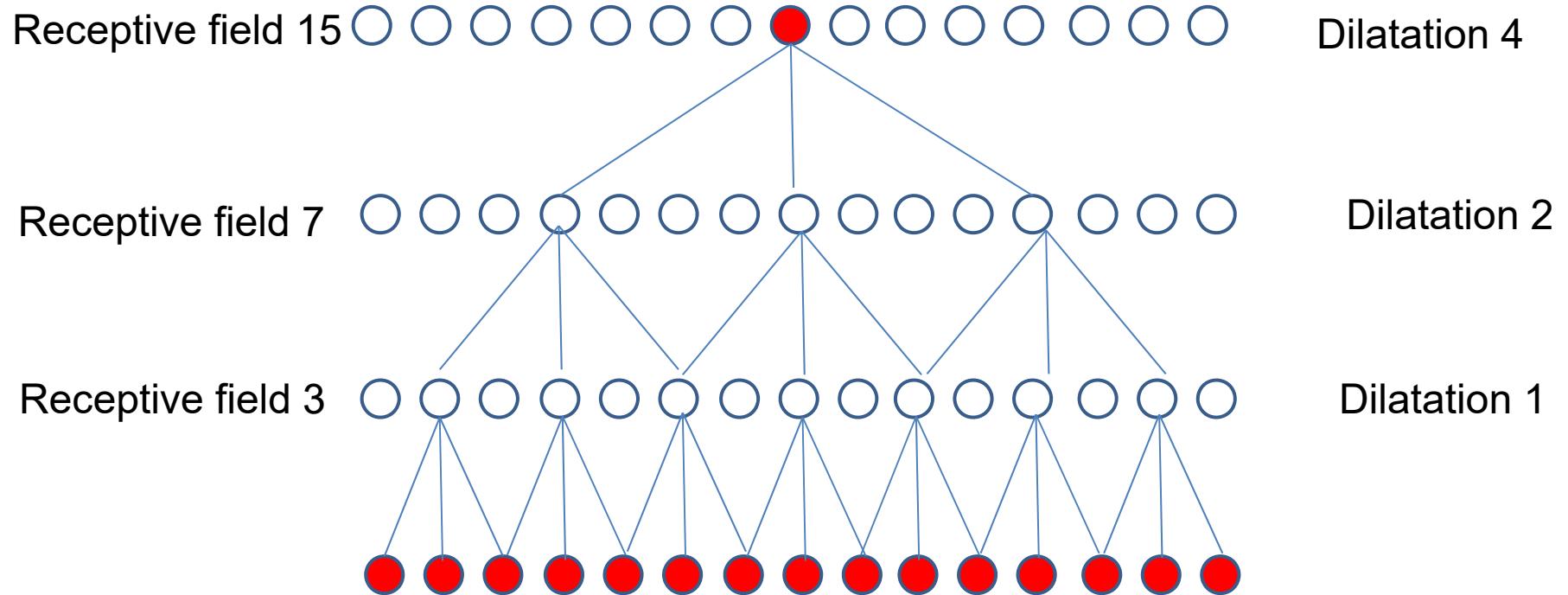
Segmentation Dilated convolutions (Yu 2016)

► 1 D example



Segmentation Dilated convolutions (Yu 2016)

► 1 D example



Segmentation

Dilated convolutions (Yu 2016)

► In 2 D

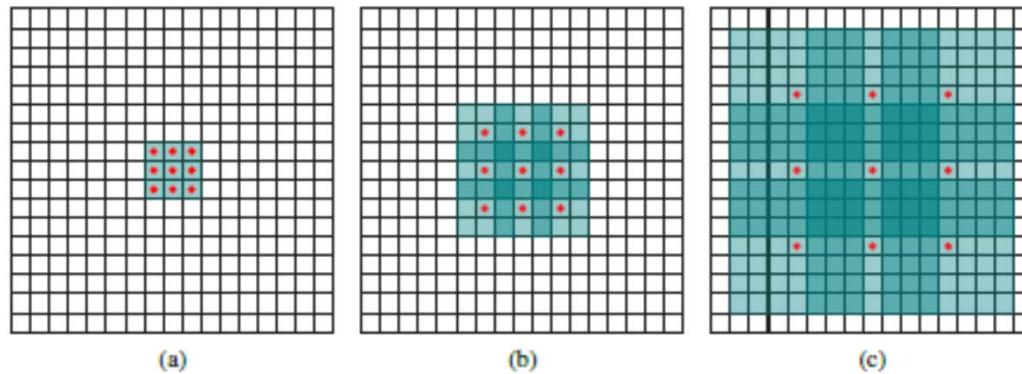


Fig from (Yu 2016)

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

► More recent architectures use improved versions of these two ideas

▶ Noisy data for vision

- ▶ Random rotations
- ▶ Random flips
- ▶ Random shifts
- ▶ Random “zooms”
- ▶ Recolorings



Recurrent networks



RNNs

Examples of tasks and sequence types

- ▶ **Sequence classification**
 - ▶ **Input: sequence, output: class**
 - ▶ Time series classification
 - ▶ Sentence classification (topic, polarity, sentiment, etc.)
- ▶ **Sequence generation**
 - ▶ **Input: initial state (fixed vector), output: sequence**
 - ▶ Music
 - ▶ Text
- ▶ **Sequence to sequence transduction**
 - ▶ **Input: sequence, output: sequence**
 - ▶ Natural language processing: Named Entity recognition
 - ▶ Speech recognition: speech signal to word sequence
 - ▶ Translation

RNNs

- ▶ Several formulations of RNN where proposed in the late 80s, early 90s
 - ▶ They faced several limitations and were not successful for applications
 - Recurrent NN are difficult to train
 - They have a limited memory capacity
- ▶ Mid 2000s successful attempts to implement RNN
 - ▶ e.g. A. Graves for speech and handwriting recognition
 - ▶ new models where proposed which alleviate some of these limitations
- ▶ Today
 - ▶ RNNs SOTA for a variety of applications e.g., speech decoding, translation, language generation, etc
- ▶ In this course
 - ▶ We briefly survey some of the developments from the 90s
 - ▶ We introduce recent developments

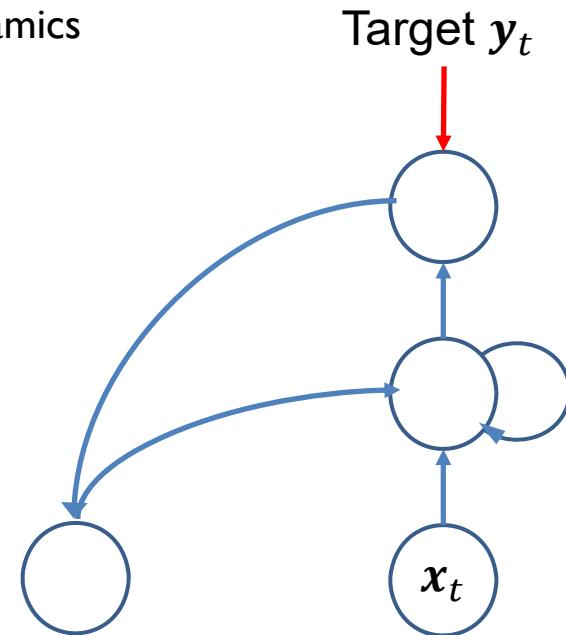
RNNs

- ▶ Imagine a NN with feedback loops, i.e. no more a DAG

- ▶ This transforms the NN into a dynamical/ state-space system

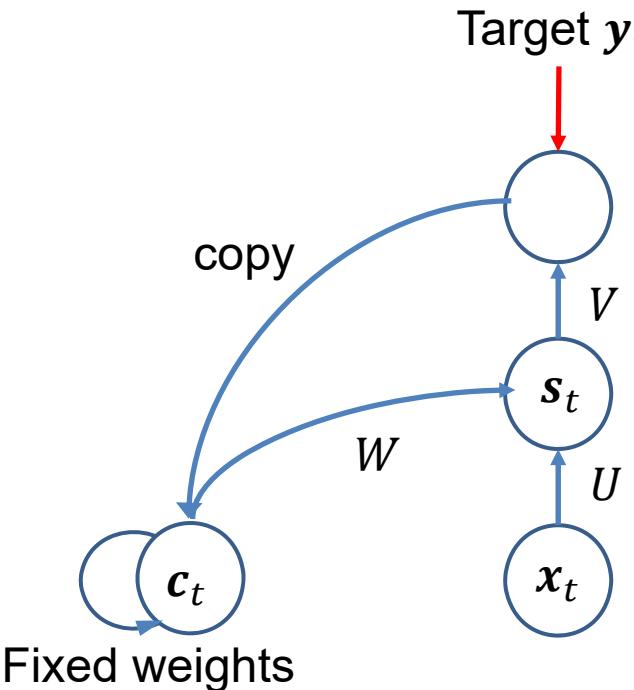
- ▶ Information can circulate according to different dynamics
 - Convergence, stable state?
 - ▶ Supervision can occur at different times
 - ▶ Inputs: fixed, sequences, etc....

- ▶ Two main families
 - ▶ Global connections
 - ▶ Local connections
- ▶ In practice, only a limited class of RNNs is used for applications

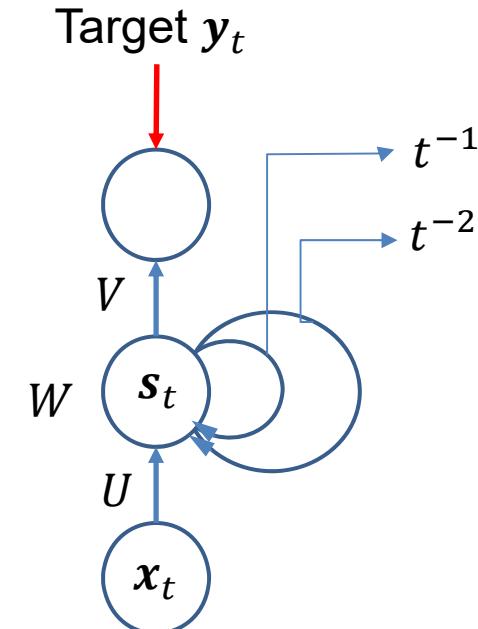


RNNs local connections (90s)

- ▶ Several local connection architectures proposed in the 90s

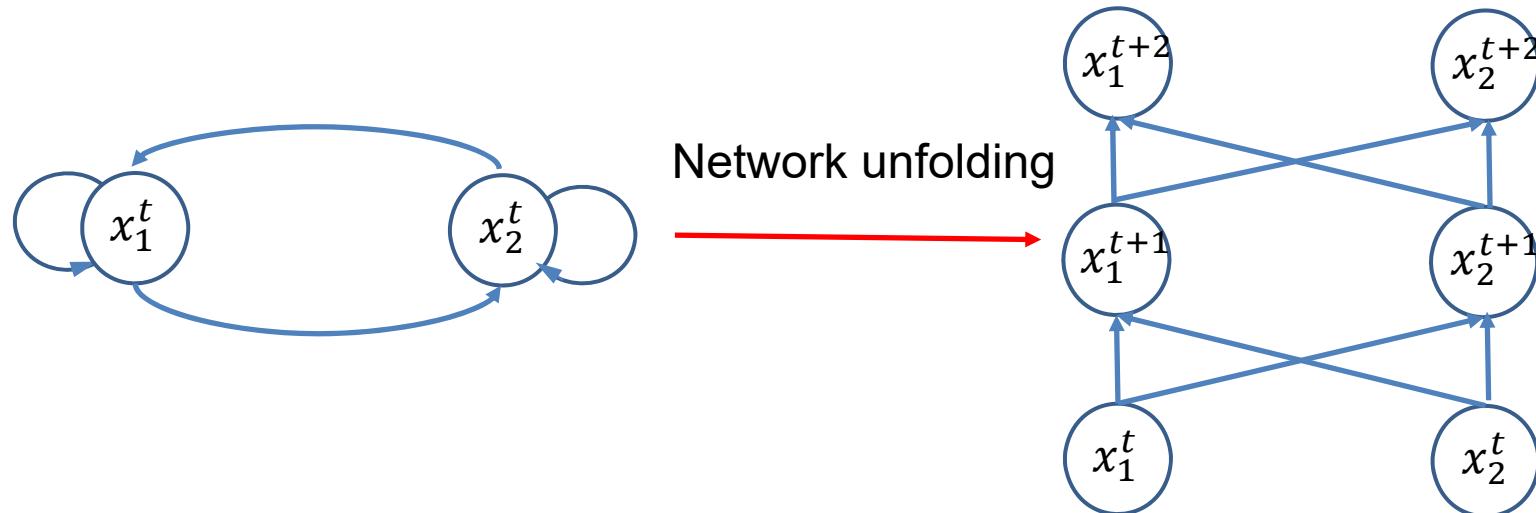


Only the forward weights are learned:
 $\text{SGD } \mathbf{s}_t = f(W\mathbf{c}_t) + U\mathbf{x}_t$



All weights learned
 $\mathbf{s}_t = f(W\mathbf{s}_{t-1}) + U\mathbf{x}_t$

RNNs global recurrences (90s)



► Algorithm

- Back Propagation Through Time (BPTT)
- For general sequences: $O(n^4)$ if n units

Fig. (Pearlmutter, 1995, IEEE Trans. on Neural Networks – nice review paper on RNN form the 90s)

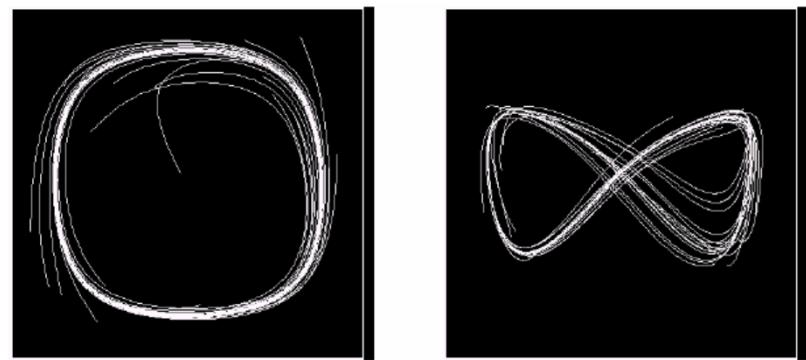
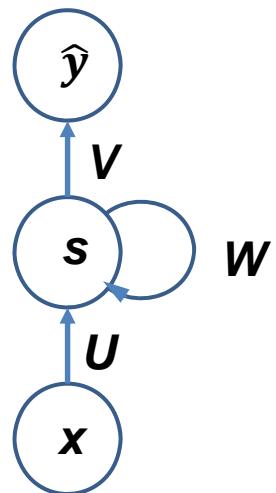


Fig. 9. The output states y_1 and y_2 plotted against each other for a 1000 time unit run, with all the units in the network perturbed by a random amount about every 40 units of time. The perturbations in the circle network (left) were uniform in ± 0.1 , and in the figure-eight network (right) in ± 0.005 .

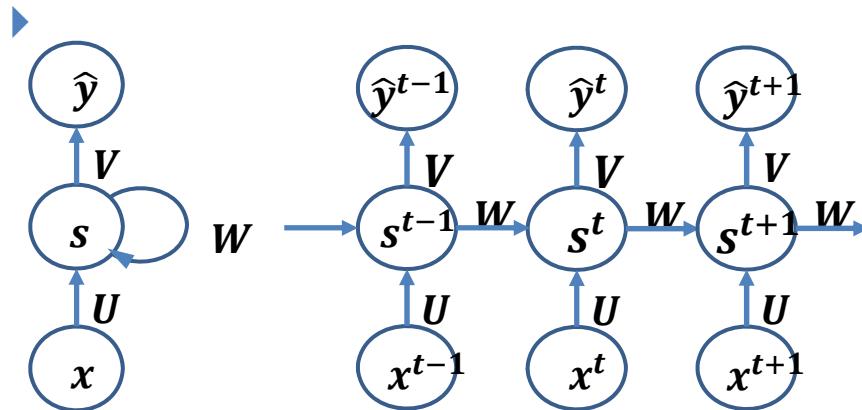
Dynamics of RNN

- ▶ We consider different tasks corresponding to different dynamics
 - ▶ They are illustrated for a simple RNN with loops on the hidden units
 - ▶ This can be extended to more complex architectures
 - ▶ However, RNNs used today all make use of local connections similar to this simple RNN
- ▶ Basic architecture

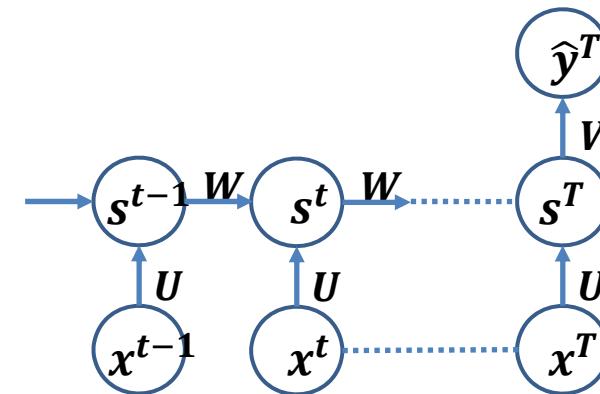


RNNs

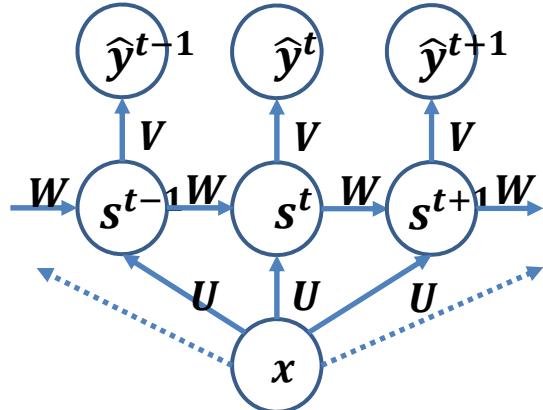
Dynamics of RNN – unfolding the RNN



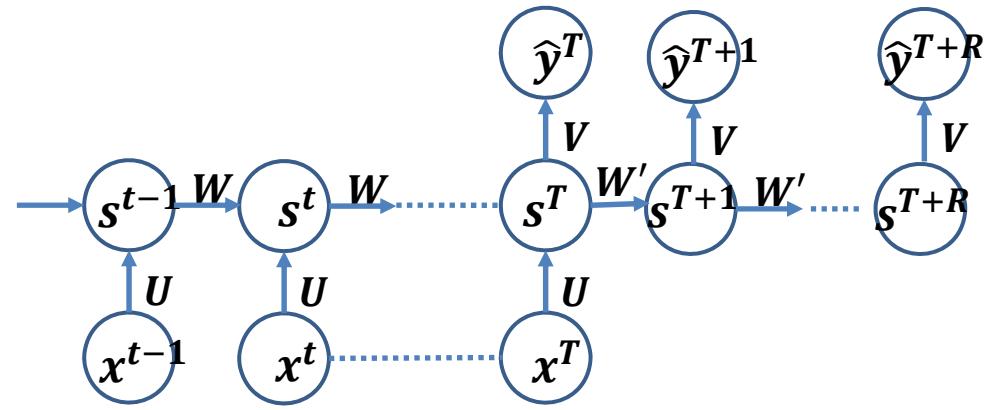
Many to many, e.g. speech
or handwriting decoding



Many to one, e.g.
sequence classification



One to many, e.g. image
annotation



Many to many, e.g.
translation

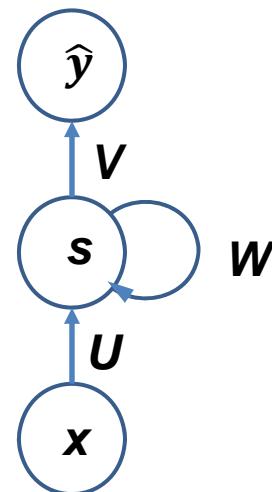
RNNs

Back Propagation Through Time

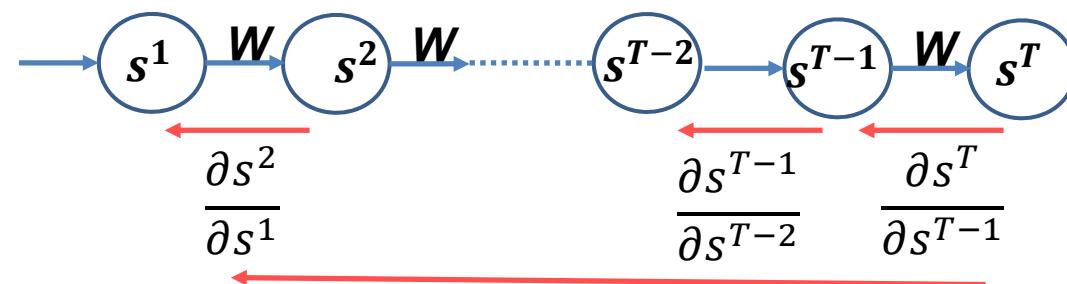
- ▶ By unfolding the RNN, one can see that one builds a Deep NN
- ▶ Training can be performed via SGD like algorithms
- ▶ This is called Back Propagation Through Time
- ▶ Automatic Differentiation can be used for training the RNNs
- ▶ RNNs suffer from the same problems as the other Deep NN
 - ▶ Gradient exploding
 - ▶ Solution: gradient clipping
 - ▶ Gradient vanishing
 - ▶ In a vanilla RNN, gradient information decreases exponentially with the size of the sequence
 - ▶ Plus limited memory
 - ▶ Again exponential decay of the memory wrt size of the sequence
- ▶ Several attempts to solve these problems
 - ▶ We introduce as an example a popular family of recurrent units:
 - ▶ Gated units (GRU, LSTMs)

RNNs

Recurrent units: Long Short Term memory (LSTM – Hochreiter 1997),
Gated Recurrent Units (GRU – Cho 2014)



Unfolded recurrent cell



$$s^{t+1} = f(Ws^t + Ux^{t+1})$$

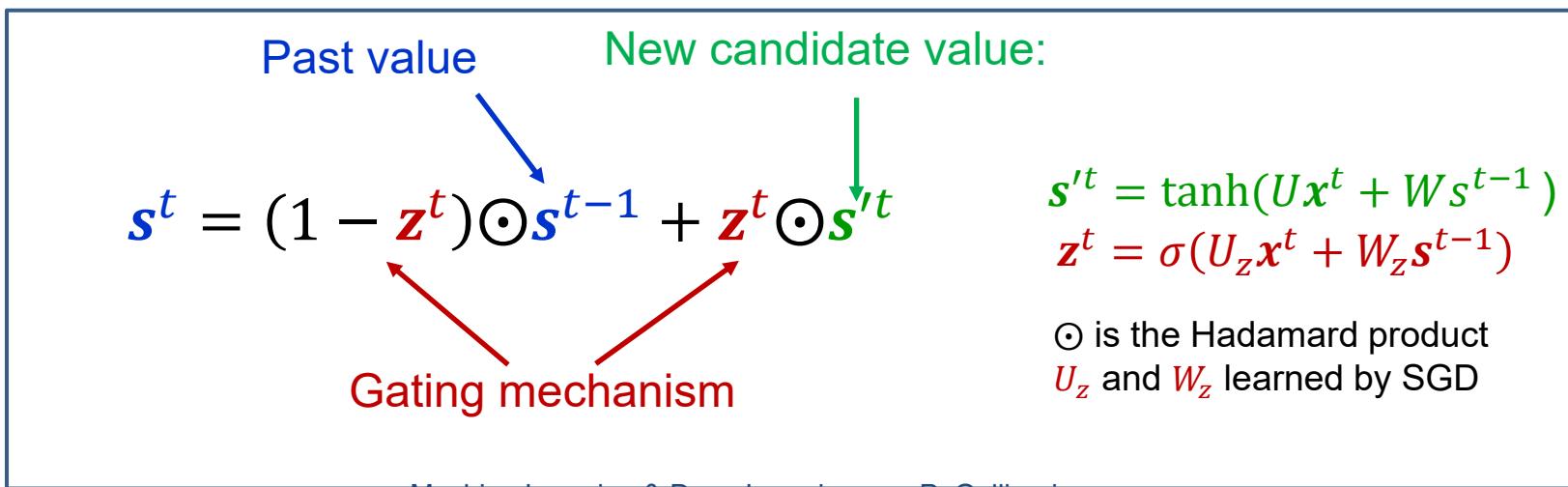
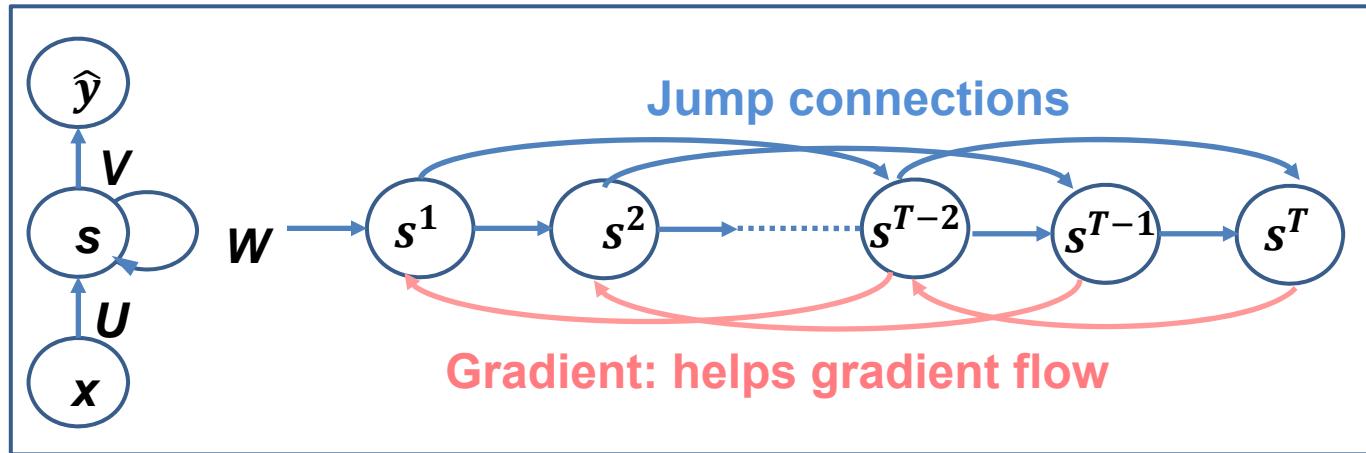
Gradient flow: vanishing
gradient

RNNs - Gated Units

Long Short Term memory (LSTM – Hochreiter 1997)

Gated Recurrent Units (GRU – Cho 2014)

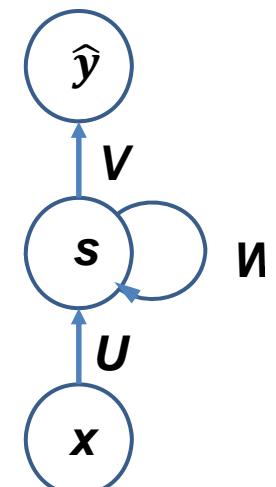
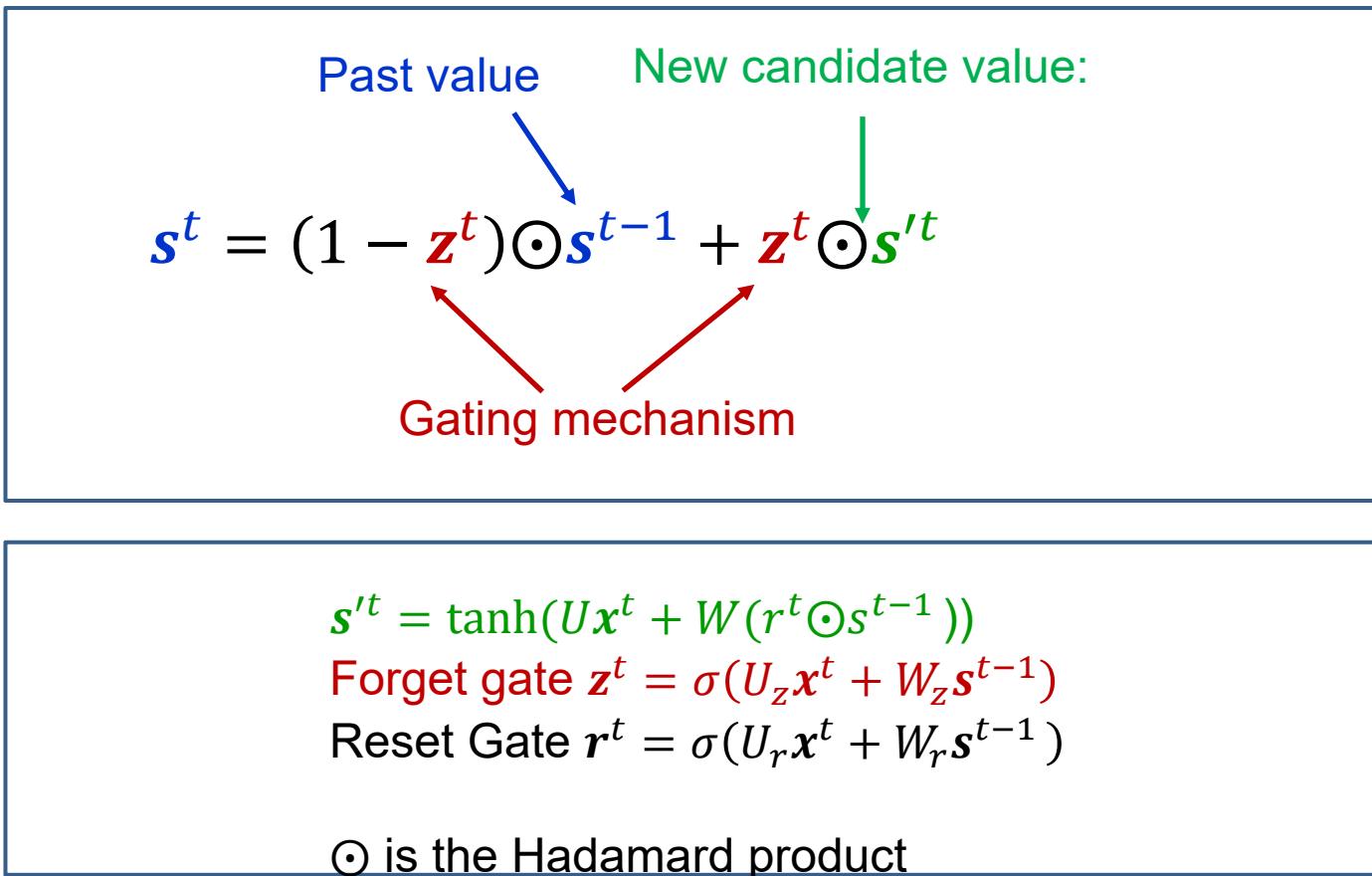
- ▶ Introducing « Jump connections » - similar to Resnet



RNNs

Gated Recurrent Units (GRU – Cho 2014)

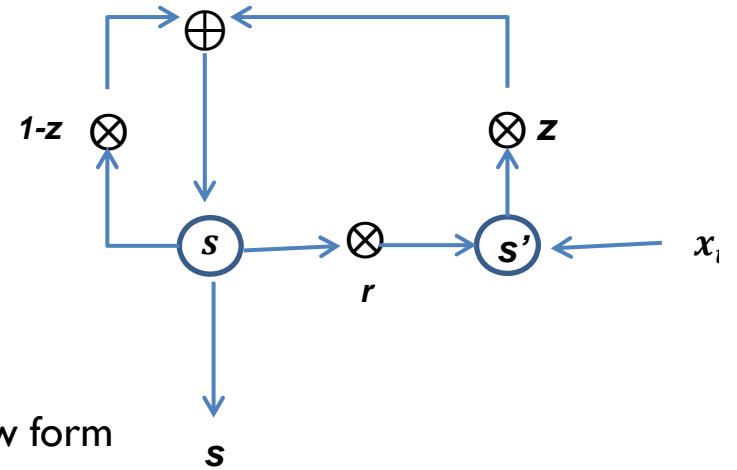
- Skip connection with Forget Gate + Reset Gate



RNNs

Gated Recurrent Units (GRU – Cho 2014)

- ▶ The output s_j^t of cell j is a weighted sum of the cell output at time $t - 1$, s_j^{t-1} and a new value of the cell s'_j^t
 - ▶ $s^t = (1 - z^t) \odot s^{t-1} + z^t \odot s'^t$
 - ▶ z is a gating function
 - ▶ If $z = 0$, s_j^t is a simple copy of s_j^{t-1}
 - ▶ If $z = 1$ it takes the new value s'_j^t
 - ▶ w.r.t the classical recurrent unit formulation, this new form allows us to remember the value of the hidden cell at a given time in the past and reduces the vanishing gradient phenomenon



RNNs

Gated Recurrent Units (GRU – Cho 2014) - followed

- ▶ The gating function is a function of the current input at time t and the past value of the hidden cell s^{t-1}
 - ▶ $\mathbf{z}^t = \sigma(U_z \mathbf{x}^t + W_z s^{t-1})$
- ▶ The new value s'^t is a classical recurrent unit where the values at time $t - 1$ are gated by a reset unit r_t
 - ▶ $s'^t = \tanh(U \mathbf{x}^t + W(r^t \odot s^{t-1}))$
- ▶ The reset unit r^t allows us to forget the previous hidden state and to start again a new modeling of the sequence
 - ▶ This is similar to a new state in a HMM (but it is soft)
 - ▶ $r^t = \sigma(U_r \mathbf{x}^t + W_r s^{t-1})$

RNNs

Gated Recurrent Units (GRU – Cho 2014)

- ▶ There are two main novelties in this unit
 - ▶ The z gating function which implements skip connections and acts for reducing the vanishing gradient effect
 - ▶ The r gating function which acts for forgetting the previous state and starting again a new subsequence modeling with no memory
- ▶ Each unit adapts its specific parameters, i.e. each may adapt its own time scale and memory size
- ▶ Training
 - ▶ is performed using an adaptation of backpropagation for recurrent nets
 - ▶ All the functions – unit states and gating functions are learned from the data using some form of SGD

Long short term memory - LSTM

- ▶ This was initially proposed in 1997 (Hochreiter et al.) and revised later.
- ▶ State of the art on several sequence prediction problems
 - ▶ Speech, handwriting recognition, translation
 - ▶ Used in conjontions with other models e.g. HMMs or in standalone recurrent neural networks
 - ▶ The presentation here is based on (Graves 2012)

Long short term memory

- ▶ In the LSTM, there are 3 gating functions

- ▶ i: input gating
- ▶ o: output gating
- ▶ f: forget gating

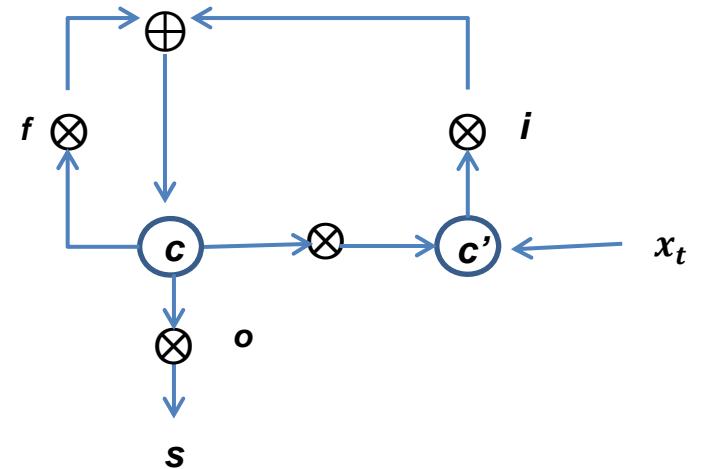
- ▶ Difference with the gated recurrent cell

- ▶ Similarities

- ▶ Both use an additive form for computing the hidden cell state (c) here.
 - This additive component reduces the vanishing gradient effect and allows us to keep in memory past state values.
- ▶ Both use a reset (called here forget (f)) gate
 - The reset permits to start from a new « state » a subsequence prediction

- ▶ Differences

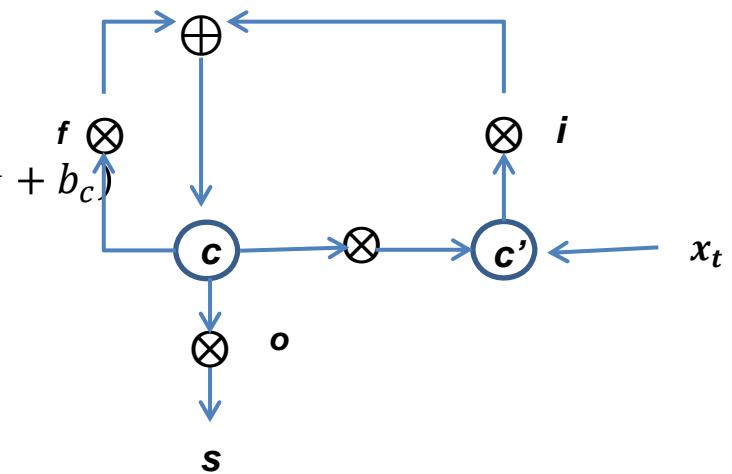
- ▶ No output gating in the GRU
- ▶ Reset does not play exactly the same role



Long short term memory

- ▶ For the forward pass, the different activations are computed as follows and the this order

- ▶ $i^t = \sigma(W_{xi}x^t + W_{hi}s^{t-1} + W_{ci}c^{t-1} + b_i)$
- ▶ $f^t = \sigma(W_{xf}x^t + W_{hf}s^{t-1} + W_{cf}c^{t-1} + b_f)$
- ▶ $c^t = f_t \odot c^{t-1} + i_t \odot \tanh(W_{xc}x^t + W_{hc}s^{t-1} + b_c)$
- ▶ $o^t = \sigma(W_{xo}x^t + W_{ho}s^{t-1} + W_{co}c^{t-1} + b_o)$
- ▶ $s^t = o^t \tanh(c^t)$



- ▶ c_t^i is a memory of cell i at time t , c_t is computed as for the GRU as a sum of c_{t-1} and of the new memory content $c'_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$
- ▶ o is an output gate
- ▶ σ is a logistic function
- ▶ W_{ci}, W_{cf}, W_{co} are diagonal matrices

RNNs

Language models

- ▶ **Objective:**
 - ▶ Probability models of sequences (x^1, x^2, \dots, x^t)
 - ▶ Items may be words or characters, or character bigrams, etc
 - ▶ Estimate:
 - ▶ $p(x^t | x^{t-1}, \dots, x^1)$
- ▶ **Example**
 - ▶ « S'il vous plaît... dessine-moi ...» what next ?
 - ▶ « $x^1 x^2 x^3 \dots \dots \dots \dots x^{t-1} \dots$ » what is x^t ?
- ▶ **Language models in everyday use**
 - ▶ Sentence completion
 - ▶ Search engine queries
 - ▶ Smartphone messages, etc
 - ▶ Speech recognition, handwriting recognition, etc



Language models

n-grams

► A simple solution: n-grams

- ▶ n-grams are sequences of n words (or characters, or any items)
- ▶ Given large text collections, it is possible to compute estimates of the posterior probabilities

$$\triangleright p(x^t | x^{t-1}, \dots, x^{t-n+1}) = \frac{p(x^t, x^{t-1}, \dots, x^{t-n+1})}{p(x^{t-1}, \dots, x^{t-n+1})}$$

$$\triangleright \text{An estimate could be } \hat{p}(x^t | x^{t-1}, \dots, x^{t-n+1}) = \frac{\text{count}(x^t, x^{t-1}, \dots, x^{t-n+1})}{\text{count}(x^{t-1}, \dots, x^{t-n+1})}$$

▶ Where $\text{count}(x^t, x^{t-1}, \dots, x^{t-n+1})$ is the number of occurrences of the sequence in the corpus

Language models

n-grams

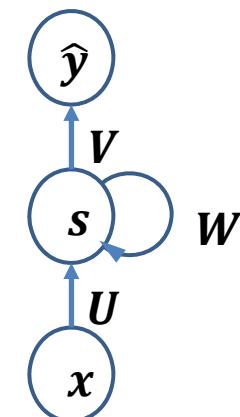
► Sparsity problem

- ▶ In order to get good estimates, this requires large text quantities
- ▶ The larger n is, the larger the training corpus should be
- ▶ For a dictionary of 10 k words, there could be
 - $10^{4 \times 2}$ bigrams
 - $10^{4 \times 3}$ trigrams, etc
 - Note: the number of n-grams in a language is smaller than $10^{4 \times n}$ but still extremely large and grows exponentially with n
 - The model size increases exponentially with n
- ▶ n-gram counting is limited to relatively short sequences
 - Only large companies like Google could afford computing/ storing estimates for $n > 10$
- ▶

RNNs

Language models

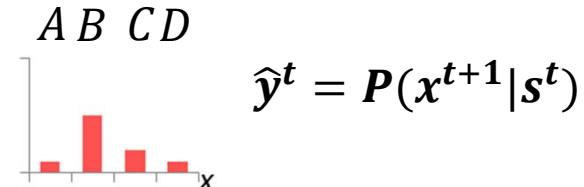
- ▶ **Objective:**
 - ▶ Probability models of sequences (x^1, x^2, \dots, x^t)
 - ▶ Estimate with RNNs:
 - ▶ $p(x^t | x^{t-1}, \dots, x^1)$
- ▶ $\hat{y}^t = g(Vs^t)$
- ▶ $s^t = f(Ws^{t-1} + Ux^t)$
- ▶ g is typically a softmax
- ▶ f could be a sigmoid, Relu, ...



Recurrent neural networks Language models

▶ Training

- ▶ Use a corpus of text, e.g. a sequence of words (x^1, x^2, \dots, x^T)
- ▶ Feed the sequence into the RNN, one word at a time
- ▶ Compute the output distribution \hat{y}^t for each time step
 - ▶ \hat{y}^t is a distribution on the word dictionary
 - This is the estimated posterior probability distribution given past subsequence
 - If the dictionary is $V = \{A, B, C, D\}$:

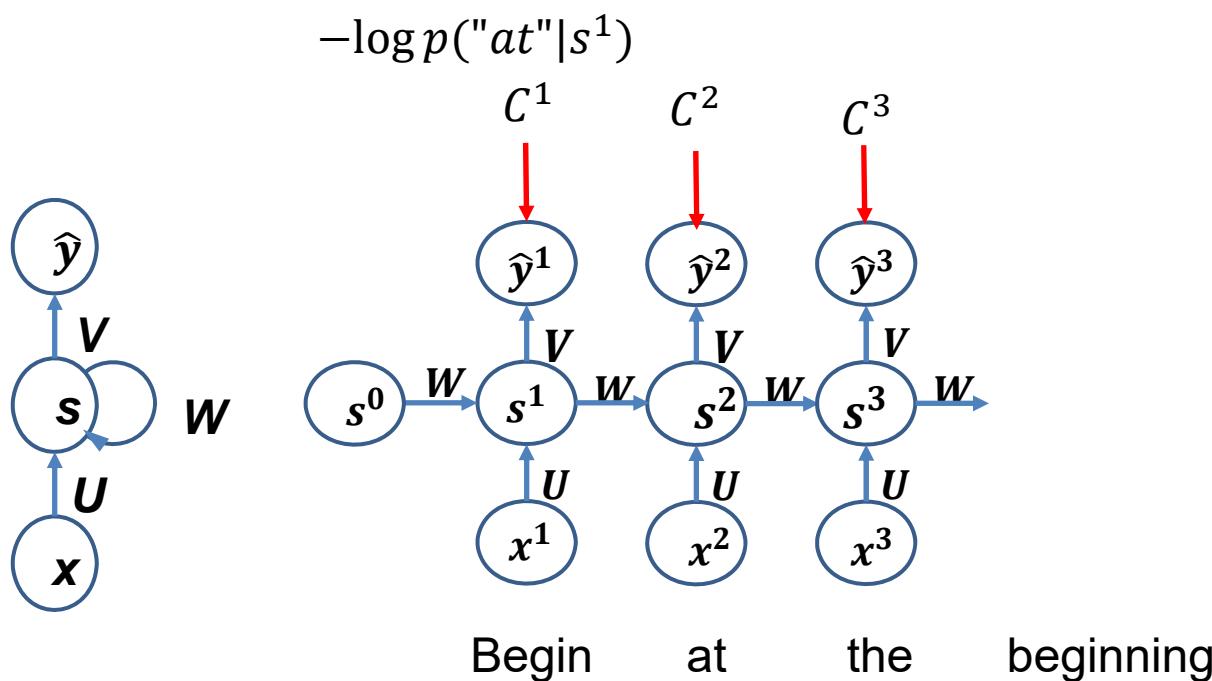


- Loss function
 - Classically the cross entropy between the predicted distribution \hat{y}^t and the target distribution y^t
 - $C^t = C(\hat{y}^t, y^t) = - \sum_{i=1}^{|V|} y_i^t \log \hat{y}_i^t$
 - Loss over the corpus $C = \sum_{t=1}^T C^t$
 - In practice, one uses a mini batch of sentences (x^1, x^2, \dots, x^T) sampled from the corpus and use SGD

Recurrent neural networks Language models

▶ Training

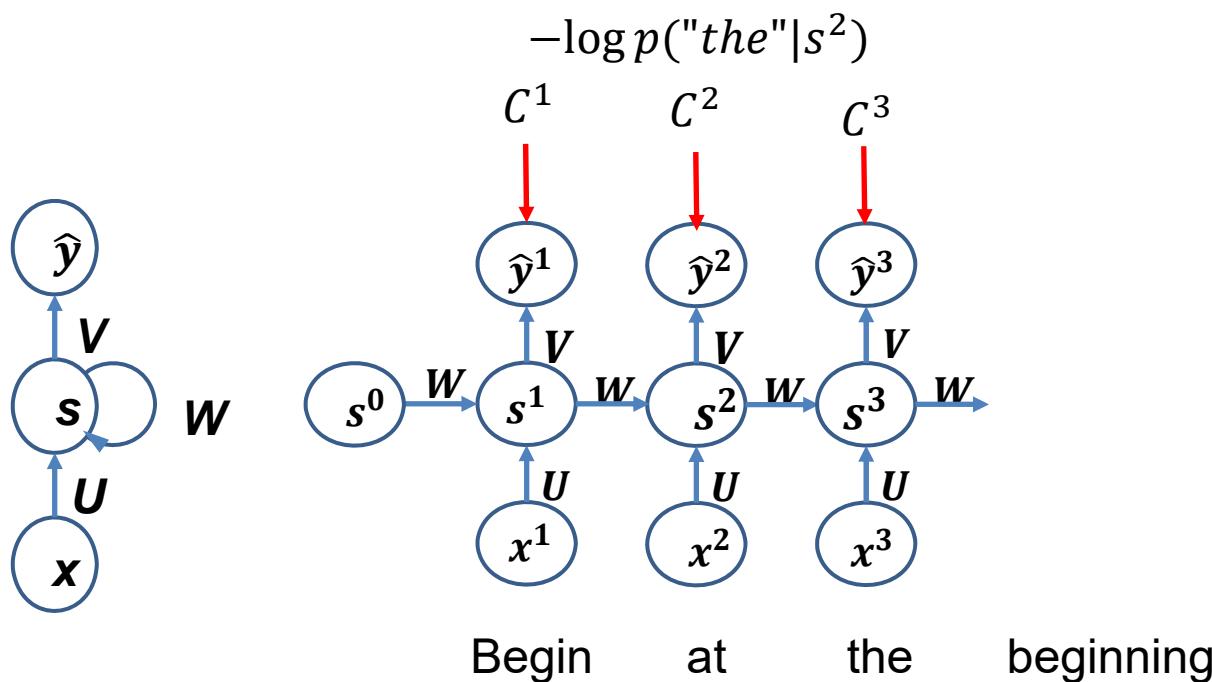
$$\hat{y}^t = P(x^{t+1}|s^t)$$



Recurrent neural networks Language models

▶ Training

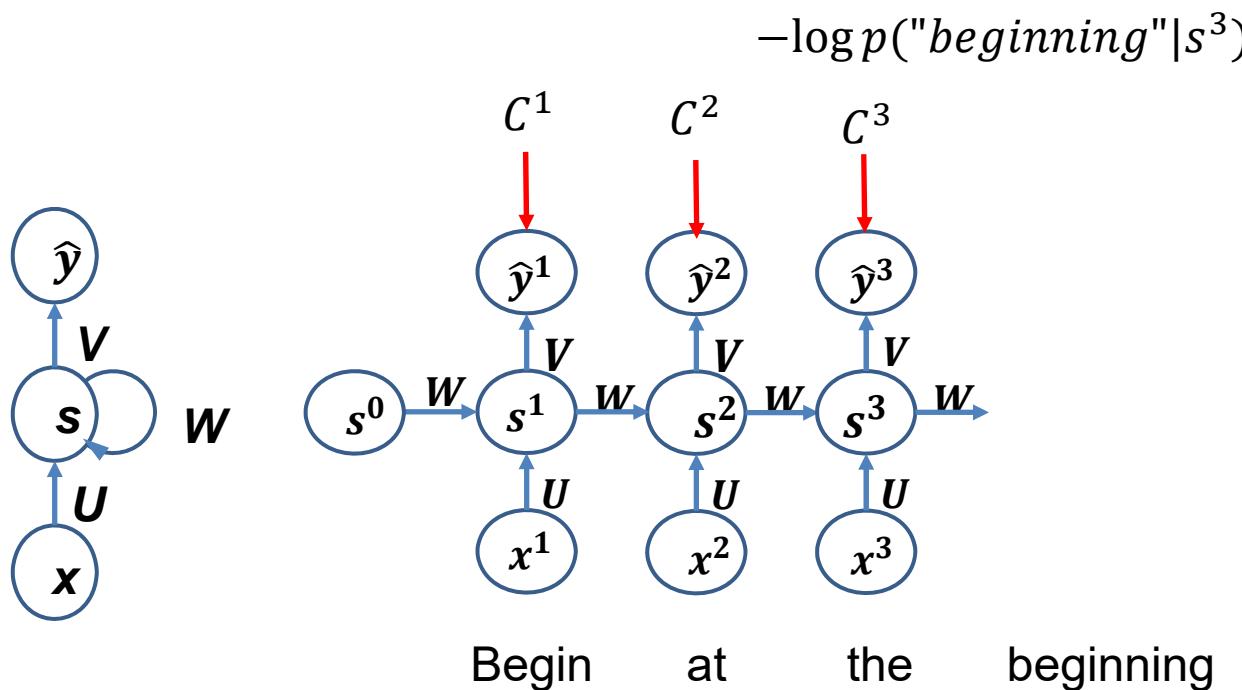
$$\hat{y}^t = P(x^{t+1}|s^t)$$



Recurrent neural networks Language models

▶ Training

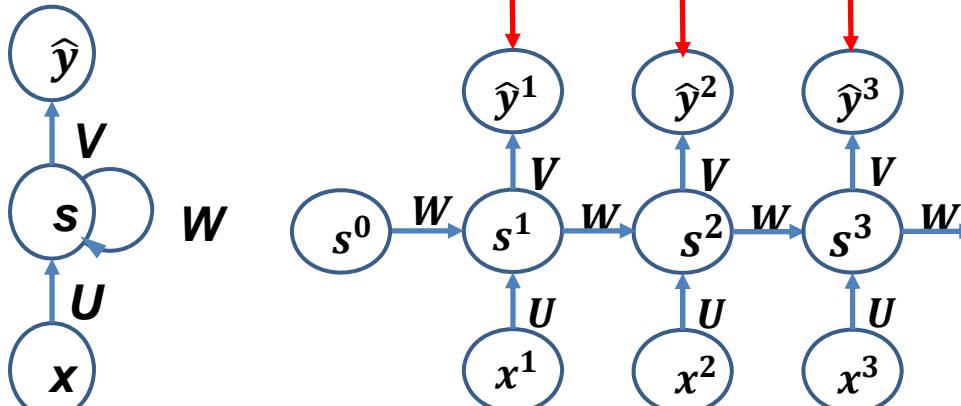
$$\hat{y}^t = P(x^{t+1}|s^t)$$



Recurrent neural networks Language models

▶ Training

$$\hat{y}^t = P(x^{t+1}|s^t) \quad C^1 \quad C^2 \quad C^3 \quad \dots \quad C = \sum_{i=1}^T C^i$$



Begin at the beginning

▶ BPTT

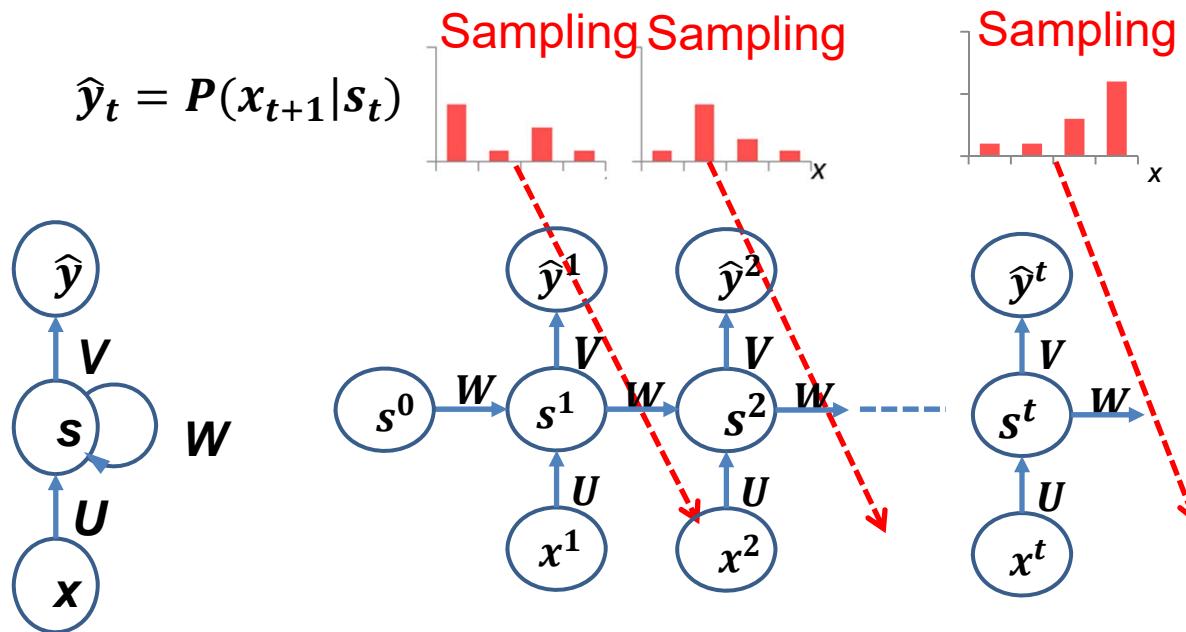
- ▶ Weights are shared: only one U , one V , one W
- ▶ Compute the derivatives wrt each C^i and sum them: $\frac{\partial C}{\partial W} = \sum_{i=1}^t \frac{\partial C^i}{\partial W}$

RNNs

Language models

Inference

- ▶ Suppose the RNN has been trained
- ▶ Inference processes by sampling from the predicted distribution



RNNs

Language models

- ▶ Words, characters, n-grams are all discrete data
- ▶ How to represent them
 - ▶ The usual way is to embed the words, etc in a continuous space of high dimension e.g. R^{200} , i.e. each word will be a vector in R^{200}
 - ▶ This could be done
 - ▶ off line using some embedding technique (e.g. Word2Vec, see later)
 - Advantage, this can be done by using very large text collections
 - ▶ On line while training the language model
 - In this case, the x s are initialized at random values in R^n and are learned by backpropagating the error, together with the other parameters

Language models – example

(Karpathy 2015- <https://karpathy.github.io/2015/05/21/rnneffectiveness/>)

- ▶ Training on Tolstoy's War and Peace a character language model
 - ▶ Stacked recurrent networks (LSTM)

tyntd-iafhatawiaoahrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tkldg t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

↓ train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwyl fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

↓ train more

Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.

↓ train more

"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.

Learning word vector representations

Word2Vec model (Mikolov et al. 2013a, 2013b)

▶ Goal

- ▶ Learn word representations
 - ▶ Words or language entities belong to a discrete space
 - ▶ They could be described using one hot encoding, but this is meaningless
 - ▶ How to represent these entities with meaningful representations?
- ▶ Word2Vec model
 - ▶ Learn robust vector representation of words that can be used in different Natural Language Processing or Information retrieval tasks
 - ▶ Learn word representations in phrase contexts
 - ▶ Learn using **very** large text corpora
 - ▶ Learn efficient, low complexity transformations
- ▶ Successful and influential work that gave rise to many developments and extensions

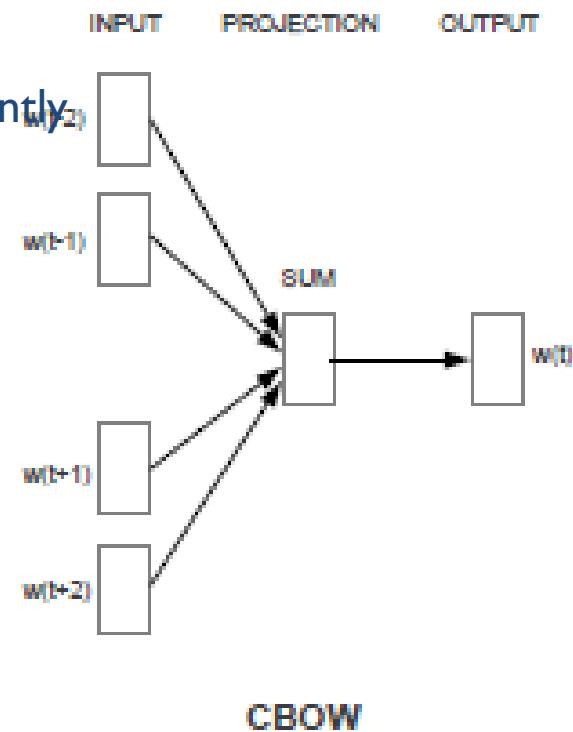
Learning word vector representations

(Mikolov et al. 2013a, 2013b)

▶ CBOW model

▶ Task

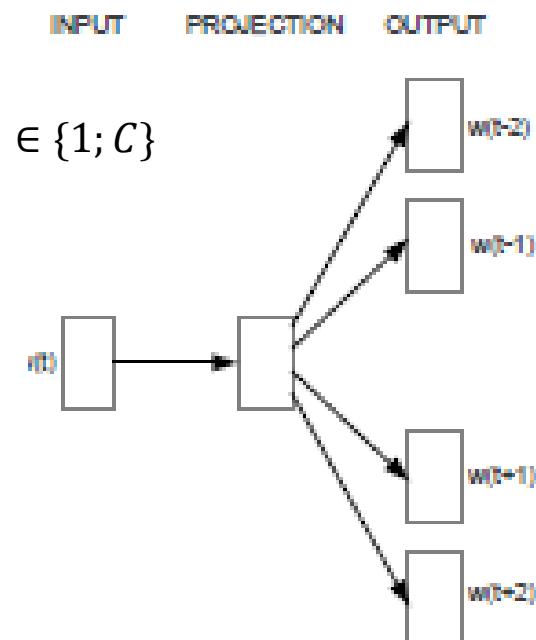
- ▶ Predict the middle word of a sequence of words
- ▶ Input and output word representations are learned jointly
 - ▶ (random initialization)
- ▶ The projection layer is linear followed by a sigmoid
- ▶ Word weight vectors in the projection layer are shared (all the weight vectors are the same)
- ▶ The output layer computes a hierarchical softmax
 - ▶ See later
 - ▶ This allows computing the output in $O(\log_2(\text{dictionary size}))$ instead of $O(\text{dictionary size})$
- ▶ The context is typically 4 words before and 4 after



Learning word vector representations (Mikolov et al. 2013a, 2013b)

▶ Skip Gram model

- ▶ Similar to the CBOW model, except that the context is predicted from the central word instead of the reverse
- ▶ Input and outputs have different representations for the same word
- ▶ The output is computed using a hierarchical softmax classifier
- ▶ Output words are sampled less frequently if they are far from the input word
 - ▶ i.e. if the context is $C = 5$ words each side, one selects $R \in \{1; C\}$ and use R words for the output context



Learning word vector representations (Mikolov et al. 2013a, 2013b)

- ▶ Skip gram model
 - ▶ Loss average log probability
 - ▶ $L = \frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t)$
 - ▶ Where T is the number of words in the whole sequence used for training (roughly number of words in the corpus) and c is the context size
 - ▶ $p(w_{out} | w_{in}) = \frac{\exp(\nu_{w_{out}} \cdot \nu_{w_{in}})}{\sum_{w=1}^V \exp(\nu_w \cdot \nu_{w_{in}})}$
 - ▶ Where ν_w is the learned representation of the w vector (the hidden layer), $\nu_{w_{out}} \cdot \nu_{w_{in}}$ is a dot product and V is the vocabulary size
 - ▶ Note that computing this softmax function is impractical since it is proportional to the size of the vocabulary
 - ▶ In practice, this can be reduced to a complexity proportional to $\log_2 V$ using a binary tree structure for computing the softmax
 - Other alternatives are possible to compute the softmax in a reasonable time
 - In Mikolov 2013: simplified version of negative sampling
 - $l(w_{in}, w_{out}) = \log \sigma(\nu_{w_{out}} \cdot \nu_{w_{in}}) + \sum_{i=1}^k \log \sigma(-\nu_{w_i} \cdot \nu_{w_{in}}))$
 - with $\sigma(x) = \frac{1}{1+\exp(-x)}$

Learning word vector representations (Mikolov et al. 2013a, 2013b)

- ▶ Properties
 - ▶ « analogical reasoning »
 - ▶ This model learns analogical relationships between terms in the representation space
 - ▶ i.e. term pairs that share similar relations are share a similar geometric transformation in the representation space
 - ▶ Example for the relation « capital of »
 - ▶ In the vector space
 - Paris – France + Italy = Rome
 - At least approximatively
 - i.e. Rome is the nearest vector to
 - Paris – France + Italy
 - ▶ Reasoning via more complex inferences
 - ▶ is however difficult:
 - ▶ Combination of transformations
 - ▶ to infer more complex facts is not effective

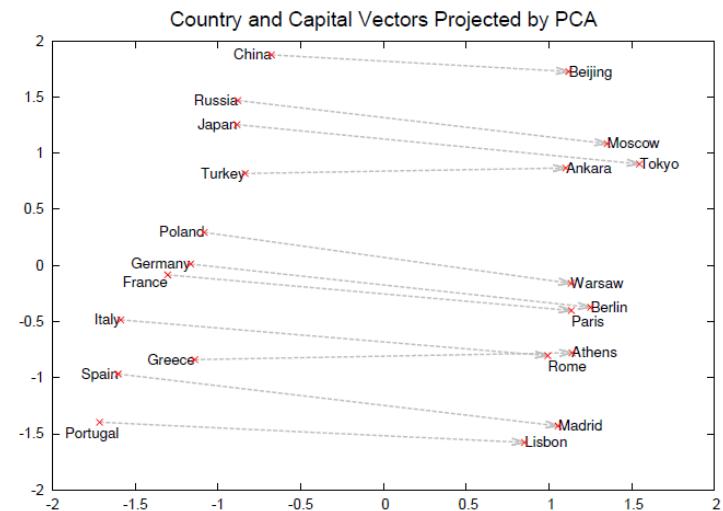


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

Figure from Mikolov 2013

Learning word vector representations (Mikolov et al. 2013a, 2013b)

- ▶ Paris – France + Italy = Rome

Table 8: Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Word2Vec extensions, example of FastText

- ▶ After W2V, several similar ideas and extensions have been published
 - ▶ Among the more popular are Glove (Pennington 2014) and FastText (Bojanowski 2017)
 - ▶ Vector representations learned on large corpora with these methods are made available
 - ▶ FastText is a simple extension of the skipgram model in W2V, where n-grams are used as text units instead of words in W2V
 - ▶ Consider the word « where » and 3-grams. « where » will be represented as:
 - <wh, whe, her, ere, re>, with « < » and « > » special « begin » and « end » characters
 - A vector representation z_i is associated to each n-gram i
 - The word representation is simply the sum of the n-gram representations of the word description
 - ▶ Remember $p(w_{out}|w_{in}) = \frac{\exp(\mathbf{v}_{w_{out}} \cdot \mathbf{v}_{w_{in}})}{\sum_{w=1}^V \exp(\mathbf{v}_w \cdot \mathbf{v}_{w_{in}})}$ in W2V
 - ▶ $\mathbf{v}_{w_{out}} \cdot \mathbf{v}_{w_{in}}$ is replaced by $\sum_{z_i \in \text{ngram}(w_{in})} \mathbf{v}_{w_{out}} \cdot z_i$
 - ▶ And the same for $\mathbf{v}_w \cdot \mathbf{v}_{w_{in}}$

Language models - Evaluation

- ▶ A classical criterion for evaluating language models is **perplexity**
- ▶ $PP(model\ q) = \left(\frac{1}{q(x^1, \dots, x^T)} \right)^{1/T} = \left(\prod_{t=1}^T \frac{1}{\sum_{i=1}^{|V|} y_i^t \hat{y}_i^t} \right)^{1/T}$
 - ▶ Where $q()$ is the probability estimate of the model
 - ▶ Perplexity $PP(model\ q)$ is estimated on a test set of sentences
 - ▶ Lower is better
 - ▶ Interpretations
 - ▶ Weighted average branching factor of a language: average nb of words following another word
 - e.g. for random digit sequences, perplexity is 10
 - ▶ Perplexity estimates on the WSJ corpus (1.5 M words test corpus, dictionary size = 20 k words) for n-gram models

Unigram	Bigram	Trigram
962	170	109

Fig. from XX

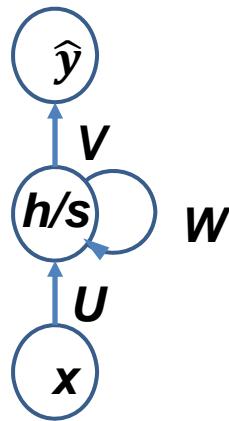
Translation

- ▶ NN have been used for a long time in translation systems (as an additional component, e.g. for reranking or as language model)
- ▶ Recently translation systems have been proposed that are based on recurrent neural networks with GRU or LSTM units.
 - ▶ Initial papers: Sutskever et al. 2014, Cho et al. 2014
- ▶ General principle
 - ▶ Sentence to sentence translation
 - ▶ Use an encoder-decoder architecture
 - ▶ Encoding is performed using a RNN on the input sentence (e.g. English)
 - ▶ This transforms a variable length sequence into a fixed size vector which encodes the whole sentence
 - ▶ Starting with this encoding, another RNN generates the translated sentence (e.g. French)
 - ▶ Instead of using a fixed length encoding, current systems use an **attention mechanism**

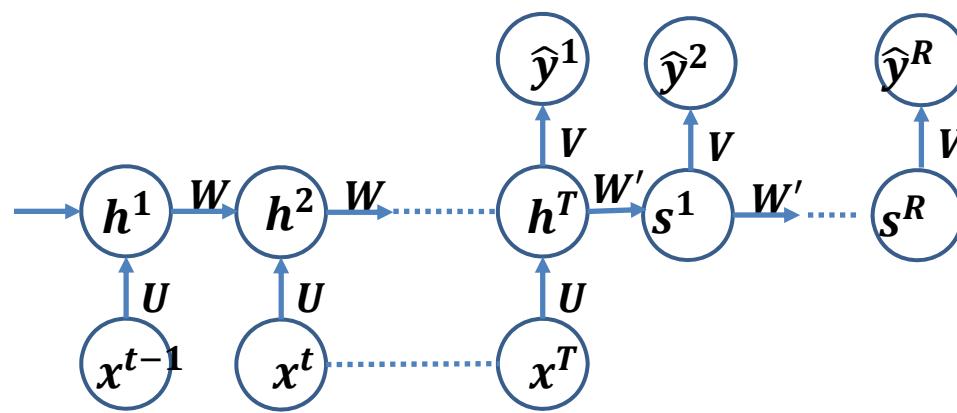
Encoder-Decoder paradigm: example of neural translation – (Cho et al. 2014, Sutskever et al. 2014)

- ▶ First attempts for DL Machine Translation with RNNs

Recurrent NN



Unfolded recurrent NN for translation



- ▶ Proof of concept, did not match SOTA, several improvements since this first attempt

Translation

- ▶ Let
 - ▶ x^1, \dots, x^T be an input sentence
 - ▶ $y^1, \dots, y^{T'}$ be an output sentence
 - ▶ Note that T and T' are most often different and that the word order in the two sentences is also generally different
- ▶ Objective
 - ▶ Learn $p(y^1, \dots, y^{T'} | x^1, \dots, x^T)$
 - ▶ Encoder
 - ▶ Reads each symbol of the input sentence sequentially using a RNN
 - ▶ After each symbol the state of the RNN is changed according to $\mathbf{h}^t = f(x^t, \mathbf{h}^{t-1})$
 - ▶ After reading the sentence, the final state is $\mathbf{h}^T = \mathbf{v}$
 - ▶ Decoder
 - ▶ Generates the output sequence by predicting the next symbol y^t given $\mathbf{s}^{t-1}, y^{t-1}$ and the vector \mathbf{v}
 - $\mathbf{s}^t = f(\mathbf{y}^{t-1}, \mathbf{s}^{t-1}, \mathbf{v})$
 - $p(y^t | y^{t-1}, \dots, y^1, \mathbf{v}) = g(y^{t-1}, \mathbf{s}^t, \mathbf{v})$
- ▶ Training: cross-entropy loss
 - ▶ $\max_{\theta} \frac{1}{N} \sum_{n=1}^N \log p_{\theta}(y_s^n | x_s^n)$, where x_s^n and y_s^n are sentences and p_{θ} is the translation model, N is the number of sentences

Translation

- ▶ Typical architecture
 - ▶ RNN with 1000 hidden cells
 - ▶ Word embeddings of dimension between 100 and 1000
 - ▶ Softmax at the output for computing the word probabilities
 - ▶ Of the order of 100 M parameters

Google Neural Machine Translation System

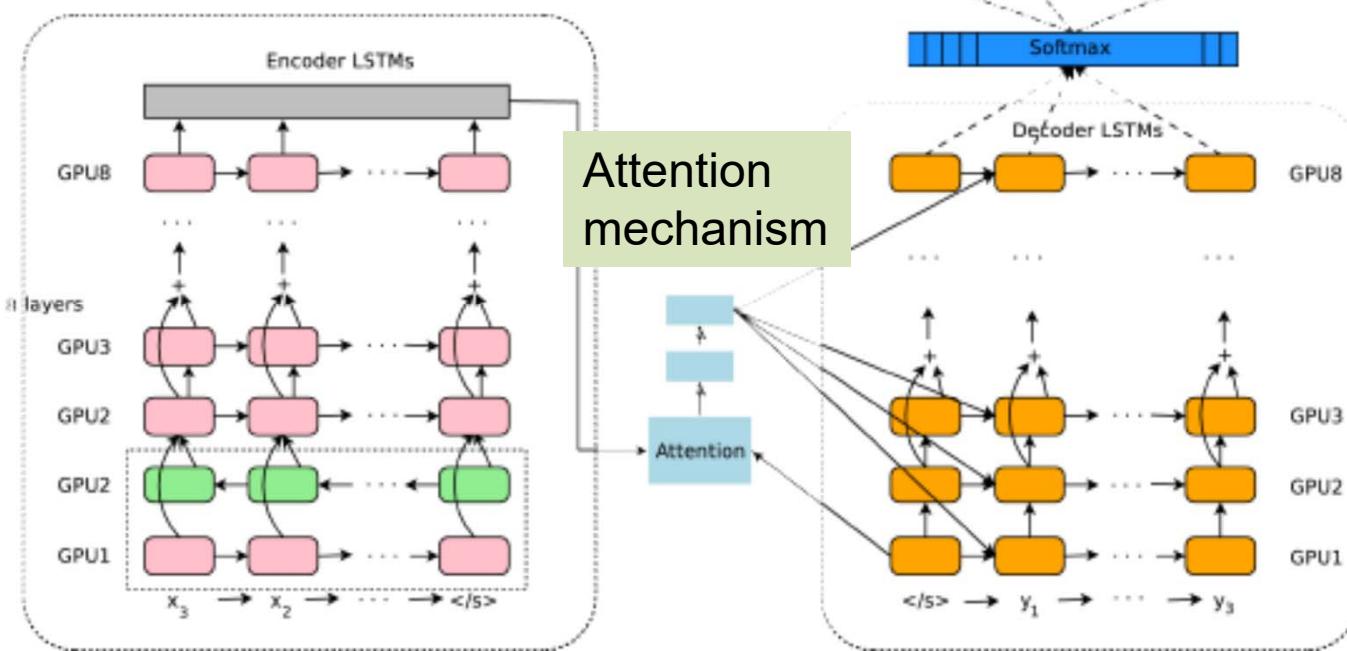
(Wu et al 2016)

<https://research.googleblog.com/2016/09/a-neural-network-for-machine.html>

▶ General Architecture

Encoder: 8 stacked LSTM RNN + residual connections

Decoder: 8 stacked LSTM RNN + residual connections + Softmax output layer



Neural image caption generator (Vinyals et al. 2015)

▶ Objective

- ▶ Learn a textual description of an image
 - ▶ i.e. using an image as input, generate a sentence that describes the objects and their relation!

▶ Model

- ▶ Inspired by a translation approach but the input is an image
 - ▶ Use a RNN to generate the textual description, word by word, provided a learned description of an image via a deep CNN

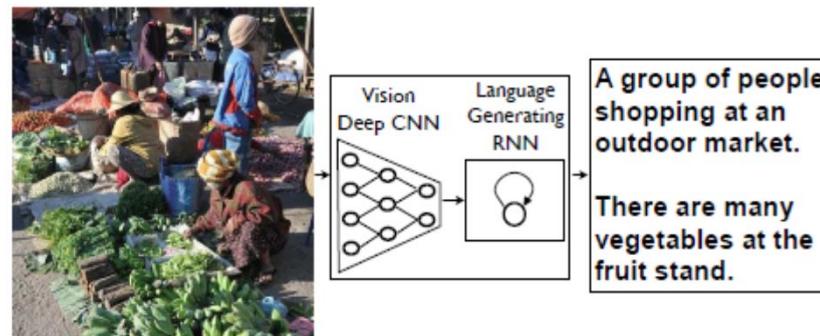


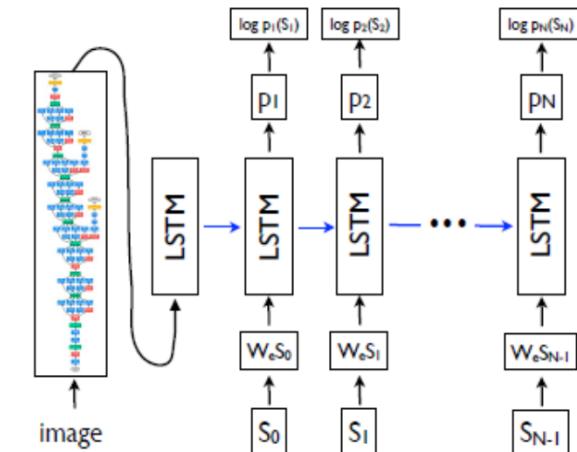
Figure 1. NIC, our model, is based end-to-end on a neural network consisting of a vision CNN followed by a language generating RNN. It generates complete sentences in natural language from an input image, as shown on the example above.

Machine Learning & Deep Learning - P. Gallinari

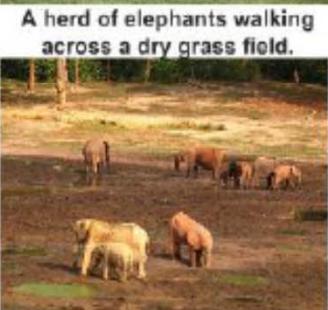
Neural image caption generator (Vinyals et al. 2015)

▶ Loss criterion

- ▶ $\max_{\theta} \sum_{I,S} \log p(S|I; \theta)$
 - ▶ Where (I, S) is an associated couple (Image, Sentence)
 - ▶ Notations correspond to the figure
- ▶ $\log p(S|I; \theta) = \sum_{t=1}^N \log p(S_t|I, S_0, \dots, S_{t-1})$
- ▶ $p(S_t|I, S_0, \dots, S_{t-1})$ is modeled with a RNN with S_0, \dots, S_{t-1} encoded into the hidden state h_t of the RNN
- ▶ Here $s^{t+1} = f(s^t, x_t)$ is modelled using a RNN with LSTM cells
- ▶ For encoding the image, a CNN is used



Neural image caption generator (Vinyals et al. 2015)



Describes without errors

Describes with minor errors

Somewhat related to the image

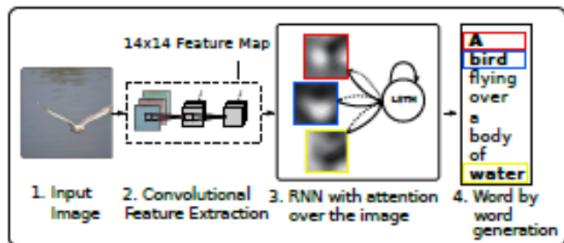
Unrelated to the image

Figure 5. A selection of evaluation results, grouped by human rating.

Attention Mechanism

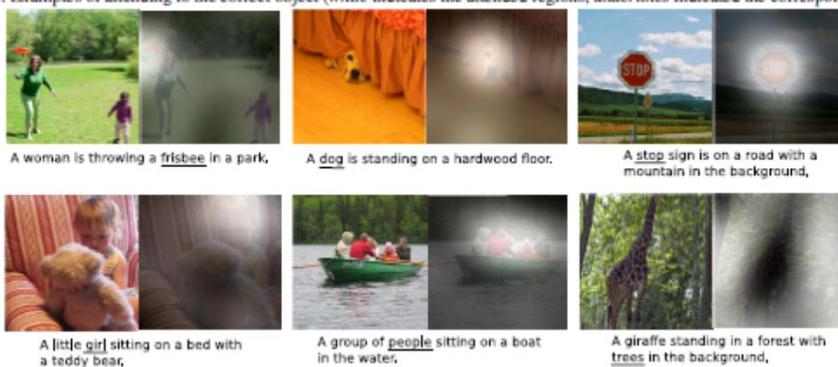
Attention mechanism

- ▶ Objective: focus on specific parts of the data representation for taking the current decision
 - ▶ Implemented as an additional differentiable modules in several architectures
- ▶ Illustration: attention on image while generating sentences

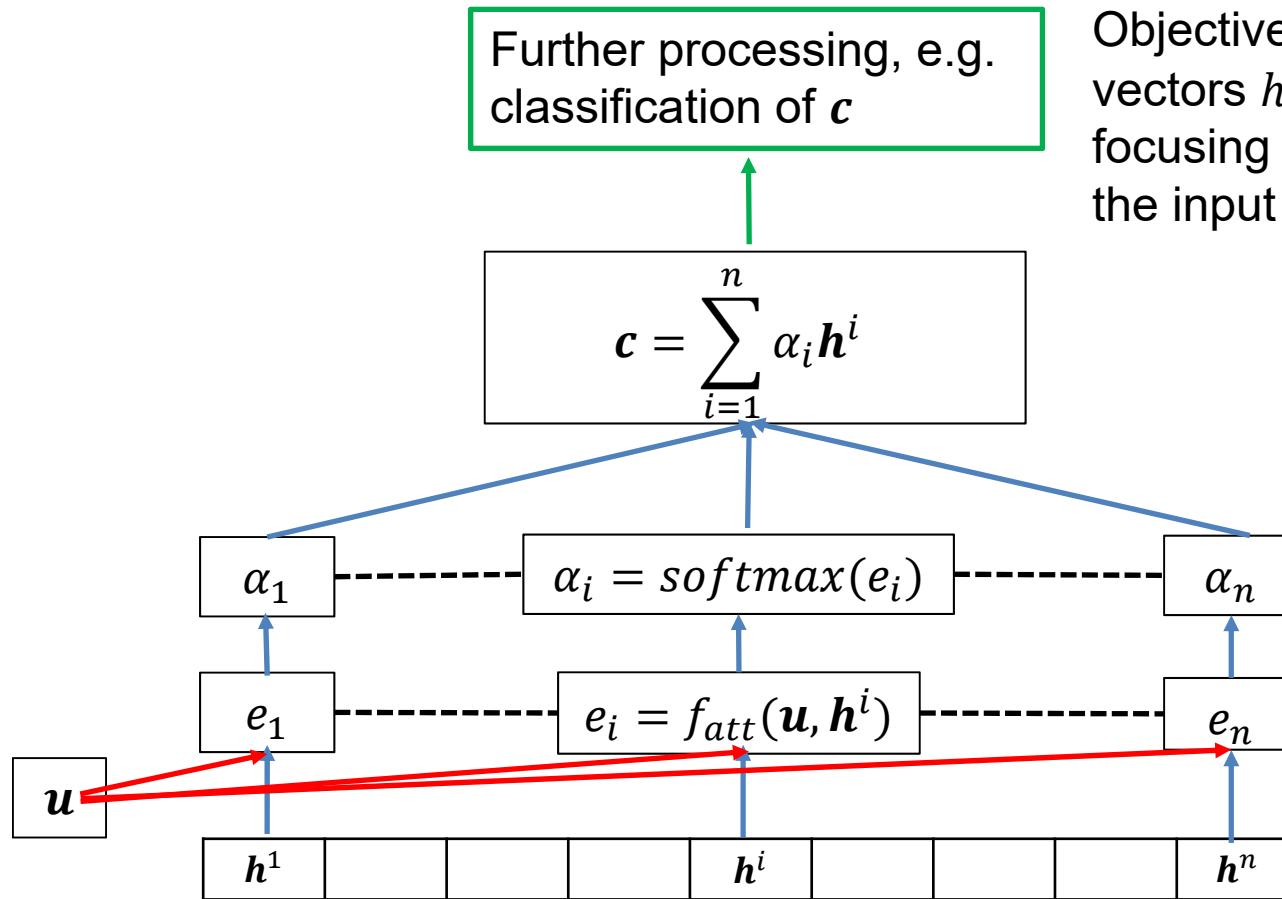


Figs. from Xu et al. 2015

Figure 4. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicate the corresponding word)



Attention mechanism



Objective: learn a combination of input vectors \mathbf{h}^i with attention weights focusing on the most relevant parts of the input signal \mathbf{h}

$\mathbf{h}^i \in R^d$: input, e.g. embedding or hidden output (e.g. RNN hidden layer)

$\mathbf{u} \in R^d$: additional info.

$\mathbf{c} \in R^d$: context vector

$e_i \in R$: attention factor

$\alpha_i \in R$: attention coefficient

$$softmax(e_i) = \frac{\exp(e_i)}{\sum_{j=1}^n \exp(e_j)}$$

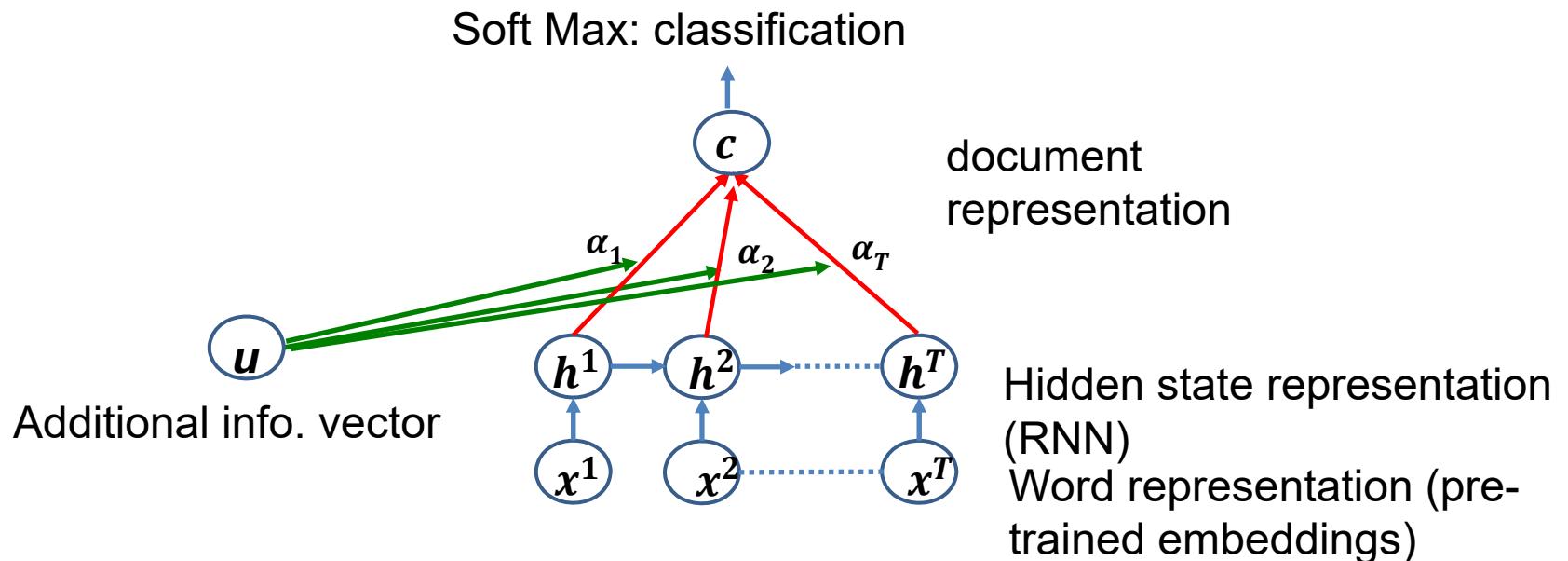
Attention mechanism

- ▶ Different attention functions f_{att} :
 - ▶ Additive
 - ▶ $f_{att}(\mathbf{u}, \mathbf{h}^i) = \mathbf{v}^T \tanh(W_1 \mathbf{h}^i + W_2 \mathbf{u}), \mathbf{v} \in R^d, \mathbf{h}^i \in R^d, W_1 : d \times d, W_2 : d \times d$
 - ▶ Multiplicative
 - ▶ $f_{att}(\mathbf{u}, \mathbf{h}^i) = \mathbf{u}^T W \mathbf{h}^i, \mathbf{u} \in R^d, W : d \times d$
 - ▶ Many variants of these formulations

Attention mechanism

For document classification (adapted from Yang et al. 2016)

- ▶ Objective: classify documents using a sequential model of attention
 - ▶ Document : word sequence w^1, \dots, w^T
 - ▶ Objective: classify the document among predefined classes – learning criterion: log likelihood
 - ▶ Word sequence encodings (e.g. pretrained via Word2Vec): x^1, \dots, x^T
 - ▶ Corresponding hidden state sequence: h^1, \dots, h^T obtained via a Recurrent NN



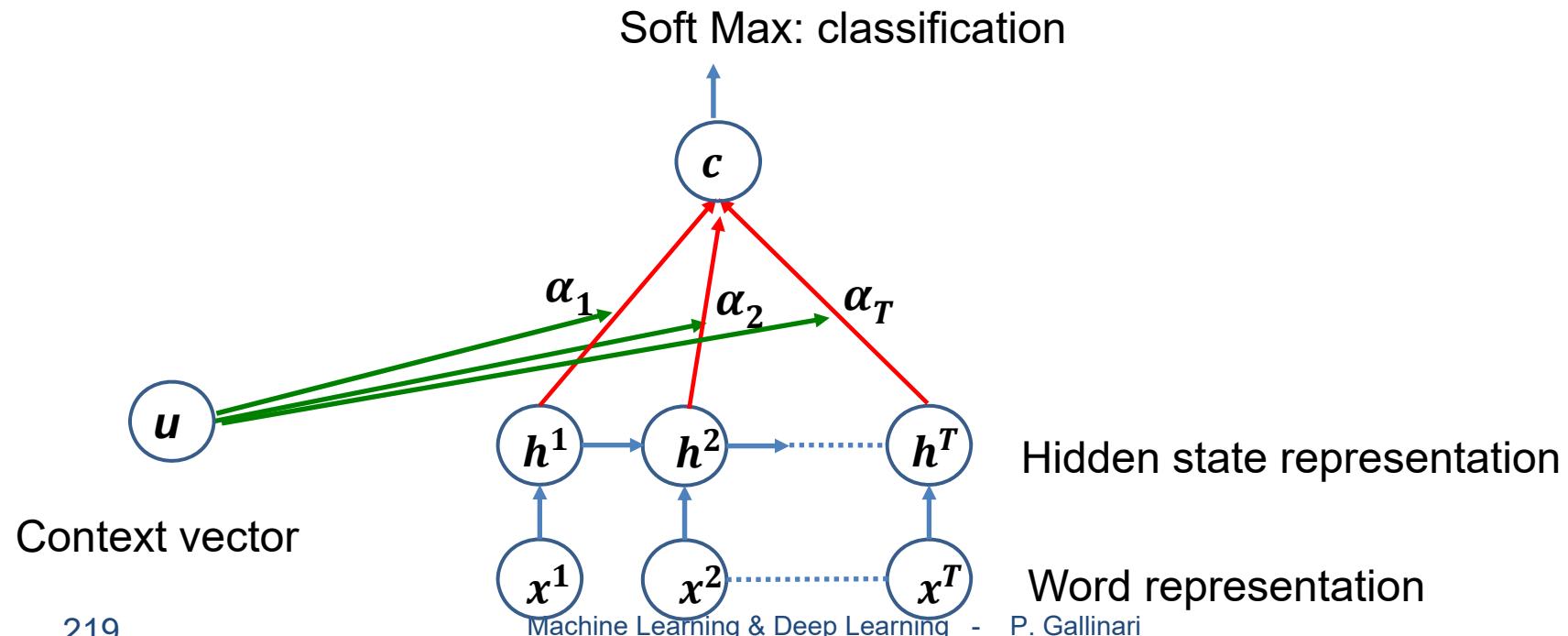
Attention mechanism

For document classification (adapted from Yang et al. 2016)

- ▶ $v_j = \tanh(W\mathbf{h}^j + \mathbf{b})$ (vector)
- ▶ $\alpha_j = \frac{\exp(v_j \cdot \mathbf{u})}{\sum_t v_t \cdot \mathbf{u}}$: attention weight (real value)
- ▶ $\mathbf{c} = \sum_{j=1}^T \alpha_j \mathbf{h}^j$: fixed size document representation (vector)
- ▶ \mathbf{u} : context vector to be learned (vector)

Parameters to be learned:

- Attention W, u
- Others: RNN, Softmax classifier



Attention mechanism

For document classification (adapted from Yang et al. 2016)

▶ Illustration (Yang et al. 2016)

- ▶ Yelp reviews: ratings from 1 to 5 (5 is the best)
- ▶ Classification = sentiment/ polarity classification
- ▶ Hierarchical attention: word and sentence levels
- ▶ Blue = word weight in the decision
- ▶ Red = sentence weight (hierarchical attention model – 2 levels: sentences and words within a sentence)

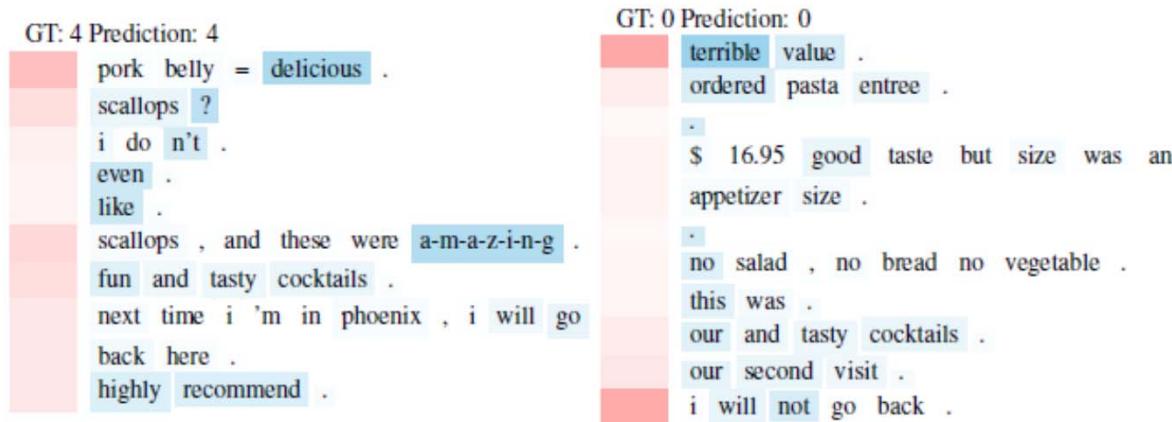


Figure 5: Documents from Yelp 2013. Label 4 means star 5, label 0 means star 1.

Attention mechanism for translation (adapted from Bahdanau et al. 2015)

► Context: **classical** Encoder – Decoder framework for translation

► Encoder

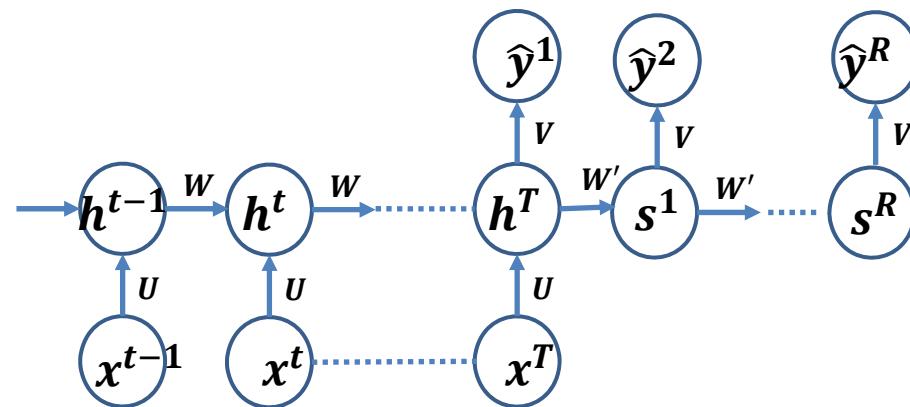
- Input sentence $\{x^1, \dots, x^T\}$ word embeddings
- Encoder: $\mathbf{h}^t = f_h(x^t, \mathbf{h}^{t-1})$ implemented via a RNN / LSTM
 - \mathbf{h}^t is the hidden state for input x^t
- $\mathbf{c} = q(\mathbf{h}^1, \dots, \mathbf{h}^T)$ for the Encoder-Decoder framework, typically $\mathbf{c} = \mathbf{h}^T$ the last hidden state for the input sentence

► Decoder

- Output sentence $\{y^1, \dots, y^R\}$ for simplification input and output sentence taken at the same length
- $p(y^t | y^1, \dots, y^{t-1}, \mathbf{c}) = g(y^{t-1}, \mathbf{s}^t, \mathbf{c})$ implemented via a RNN or LSTM + softmax
 - \mathbf{s}^t is the hidden state of the decoder for output y^t
 - Decoding is conditionned on a unique vector \mathbf{c} for the whole sentence

Attention mechanism for translation (adapted from Bahdanau et al. 2015)

- ▶ Context: **classical** Encoder – Decoder framework for translation



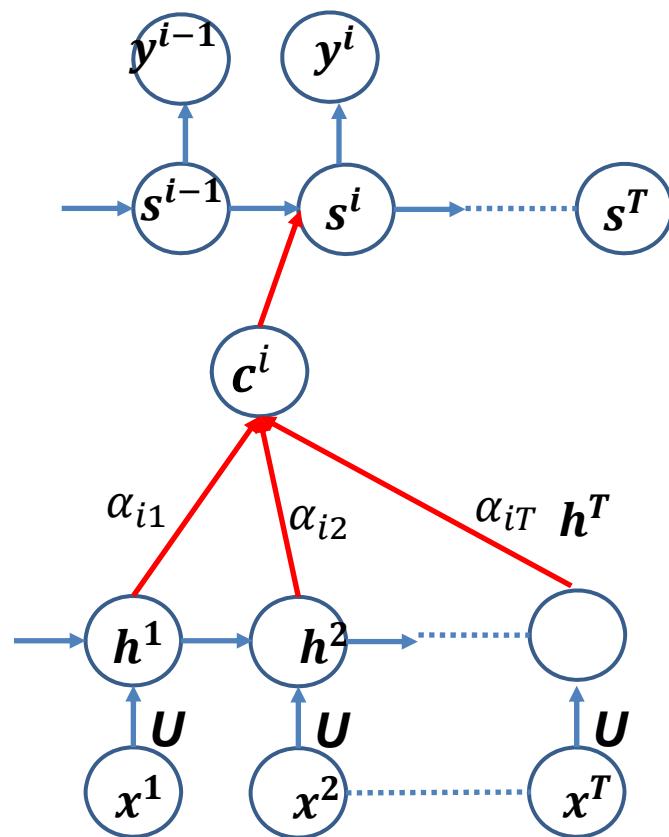
Attention mechanism for translation (adapted from Bahdanau et al. 2015)

▶ Attention mechanism

- ▶ Instead of conditionning the output y^i on the whole context $\mathbf{c} = \mathbf{h}^T$, the attention mechanism will use as context \mathbf{c}_i a linear combination of the $\mathbf{h}^t, t = 1 \dots T$
 - ▶ One \mathbf{c}_i is computed for each y^i instead of a common context \mathbf{c} for all y^i s
- ▶ The encoder is the same as before
- ▶ Decoder
 - ▶ $p(y^i | y^1, \dots, y^{i-1}, x) = g(y^{i-1}, s^i, c_i)$
 - ▶ $s^i = f(s^{i-1}, y^{i-1}, c_i)$
- ▶ Context vector
 - ▶ $e_{ij} = a(s^{i-1}, h^j)$ computed via a simple MLP for example
 - ▶ $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})}$ weight of h^j when decoding y^i
 - ▶ $c^i = \sum_{j=1}^T \alpha_{ij} h^j$ context vector
- ▶ The whole system is trained end to end

Attention mechanism for translation (adapted from Bahdanau et al. 2015)

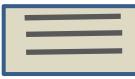
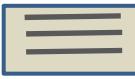
▶ Attention mechanism



Self Attention (Lin 2017)

- ▶ Sometimes, there is no additional information \mathbf{u}
- ▶ Self attention – point-wise description
 - ▶ Score for input \mathbf{h}^i
 - ▶ $e_i = f_{att}(\mathbf{h}^i) = \mathbf{v}^T \tanh(W\mathbf{h}^i), W : dx d, \mathbf{h}^i : dx 1, \mathbf{v} : dx 1, e_i : 1x 1, i = 1..n$
 - ▶ Normalized score for input \mathbf{h}^i
 - ▶ $\alpha_i = \text{softmax}(e_i) : 1x 1, i = 1..n$
 - ▶ Context vector for the whole sequence $[\mathbf{h}^1, \dots, \mathbf{h}^n]$
 - ▶ $\mathbf{c} = \sum_{i=1}^n \alpha_i \mathbf{h}^i : dx 1$
 - ▶ Self attention – vector-wise description
 - ▶ $H = [\mathbf{h}^1, \dots, \mathbf{h}^n] : dx n$
 - ▶ $\mathbf{e} = f_{att}(H) = \mathbf{v}^T \tanh(WH), W : dx d, H : dx n, \mathbf{v} : dx 1, \mathbf{e} : 1x n$
 - ▶ $\boldsymbol{\alpha} = \text{softmax}(\mathbf{e}) = [\dots, \text{softmax}(e_i), \dots] : 1x n$
 - ▶ $\mathbf{c} = H \boldsymbol{\alpha}^T : dx 1$

Self Attention (Lin 2017)

- ▶ Multiple representations/ multi-head representations
 - ▶ objective: instead of learning a unique representation for a given position, learn multiple representations c corresponding to multiple attentions
 - ▶ $E = V^T \tanh(WH), E : rxn, V : dxr, W : dxd, H : dxn$
 - 1 row per head (r heads)
 - ▶ $A = \text{softmax}(E) : rxn$, softmax by rows, 1 row per head
 - ▶ $C = HA^T, C : dxr$, 1 column per head
 - ▶ Penalization term enforces orthogonality of A and decreases redundancy in the vector components of A
 - ▶ $\|AA^T - I\|_F^2$

Self attention (Lin 2017)

- ▶ Example: sentence representation (Lin 2017)
 - ▶ Objective encode a variable length sentence in a fixed dimensional vector for a classification task, e.g. sentiment classification
 - ▶ Direct application of the self attention model described before
 - ▶ $H = [\mathbf{h}^1, \dots, \mathbf{h}^n]$: $d \times n$, hidden representation of the sentence obtained from a RNN e.g. LSTM, \mathbf{h}^i corresponds to the encoding of word w_i + past sequence of words
 - ▶ $E = V^T \tanh(WH), A = \text{softmax}(E), C = HA^T, \|AA^T - I\|_F^2$
 - ▶ Here the attention vectors representation C replaces alternatives like:
 - ▶ the last hidden state of an encoder in RNNs or any other combination of the h states, k-max pooling, etc
 - ▶ The classification module operates on C , and all the components can be trained together

Self attention (Lin 2017)

- ▶ Example: sentence representation (Lin 2017)

Fig. from (Lin 2017)

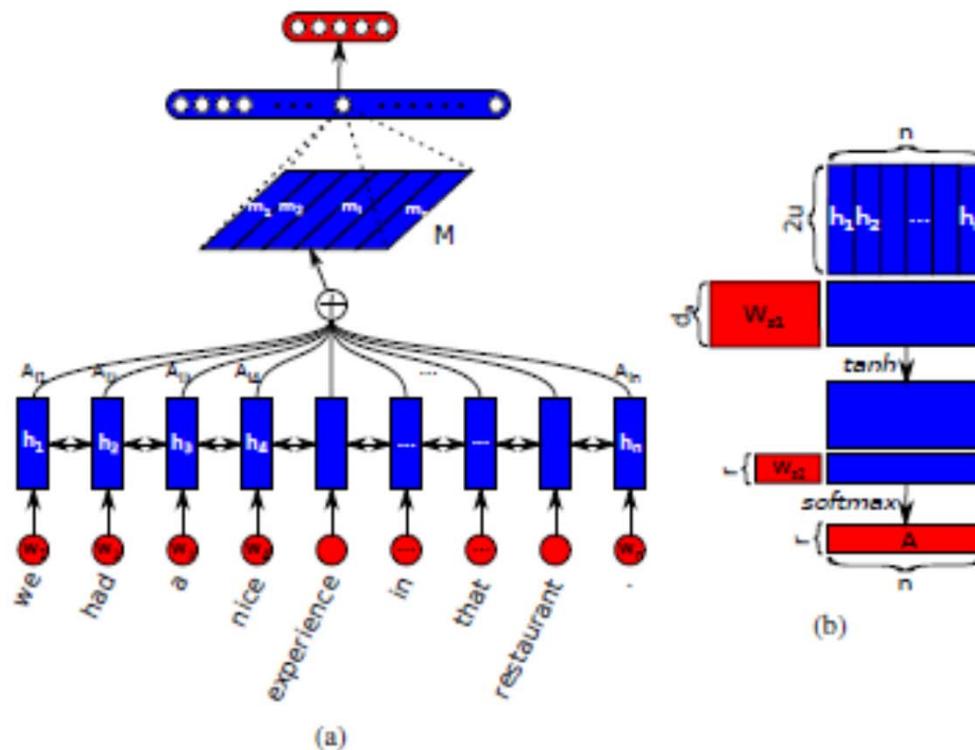


Figure 1: A sample model structure showing the sentence embedding model combined with a fully connected and softmax layer for sentiment analysis (a). The sentence embedding M is computed as multiple weighted sums of hidden states from a bidirectional LSTM (h_1, \dots, h_n), where the summation weights (A_{t1}, \dots, A_{tn}) are computed in a way illustrated in (b). Blue colored shapes stand for hidden representations, and red colored shapes stand for weights, annotations, or input/output.

Key-value attention (as in Vaswani 2017)

- ▶ Vector \mathbf{h}^i is separated into two components $\mathbf{h}^i = \begin{pmatrix} \mathbf{k}^i \\ \mathbf{v}^i \end{pmatrix}$, where \mathbf{k}^i is called the key and \mathbf{v}^i the value
 - ▶ \mathbf{k}^i is used to compute attention and \mathbf{v}^i to compute the new representation
- ▶ Self attention with key-values – point-wise description
 - ▶ Score for input \mathbf{h}^i
 - ▶ $e_i = f_{att}(\mathbf{k}^i) = \mathbf{u} \cdot \mathbf{k}^i, \mathbf{u}, \mathbf{k}^i : d_k \times 1, i = 1..n$
 - ▶ Normalized score for \mathbf{k}^i
 - ▶ $\alpha_i = softmax(e_i) = \frac{\exp(\mathbf{u} \cdot \mathbf{k}^i)}{\sum_{j=1..n} \exp(\mathbf{u} \cdot \mathbf{k}^j)} : 1 \times 1, i = 1..n$
 - ▶ Context vector for the whole sequence $[\mathbf{h}^1, \dots, \mathbf{h}^n]$
 - ▶ $\mathbf{c} = \sum_{i=1}^n \alpha_i \mathbf{v}^i : d_v \times 1$

Transformer Networks

Initial paper: Vaswani 2017

Story Telling and Illustrations used in the slides:

J.Alammar 2018 - <http://jalammar.github.io/illustrated-transformer/>

P.Bloem 2019 - <http://www.peterbloem.nl/blog/transformers>

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

- ▶ Transformer networks were proposed in 2017
- ▶ They implement a self attention mechanism
- ▶ They became SOTA technology for many NLP problems
- ▶ Transformer blocks are now a basic component of the NN zoo

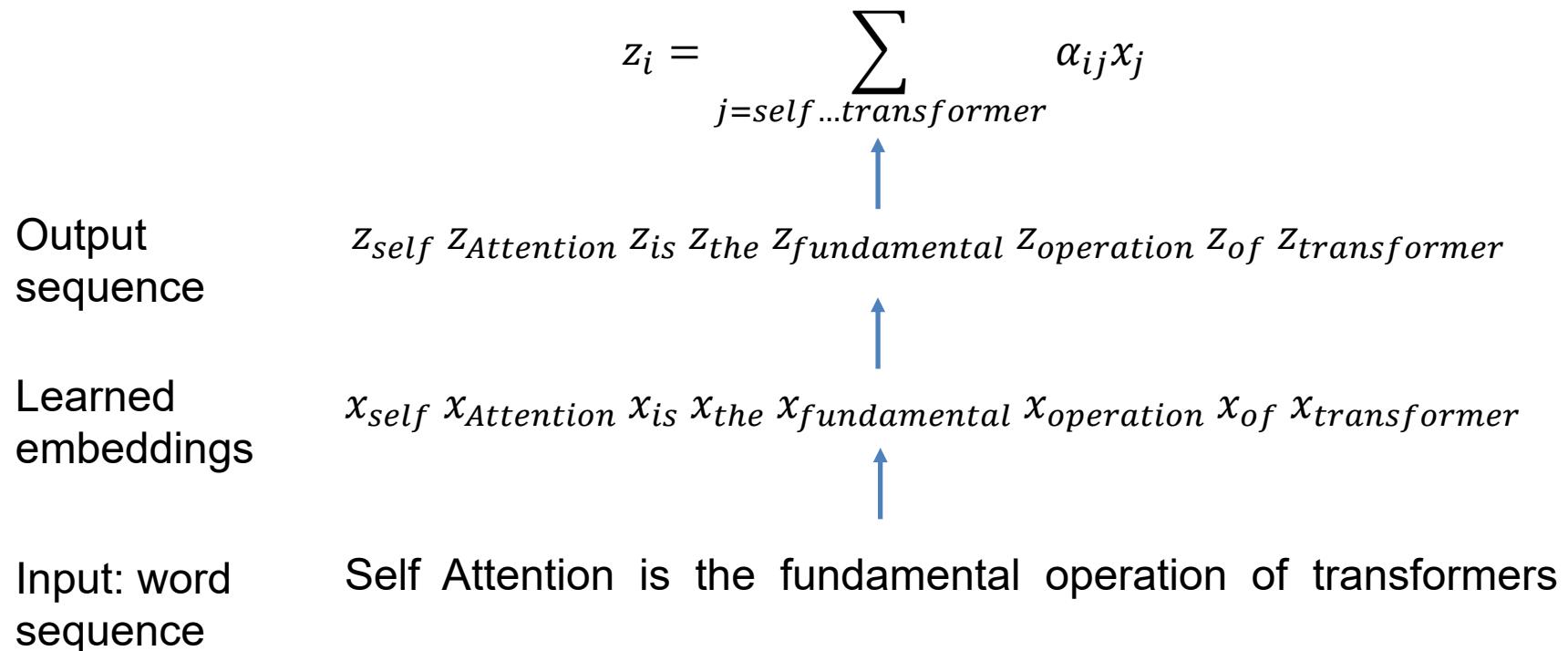
Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Self Attention

- ▶ **Self Attention is the fundamental operation of transformers**
 - ▶ Self attention is a sequence to sequence operation
 - ▶ Let x_1, x_2, \dots, x_T and z_1, z_2, \dots, z_T be respectively the input and output vector sequence
 - ▶ Self attention computes the output sequence as:
 - ▶ $z_i = \sum_j \alpha_{ij} x_j$
 - ▶ With w_{ij} a normalized attention score
 - ▶ A simple version of the normalized score could be:
 - $e_{ij} = x_i \cdot x_j$
 - $\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp e_{ij}}{\sum_k \exp e_{ik}}$
 - ▶ α_{ij} measures how x_i and x_j are important for predicting y_i

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)
Self Attention

- ▶ Self Attention is the fundamental operation of transformers



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

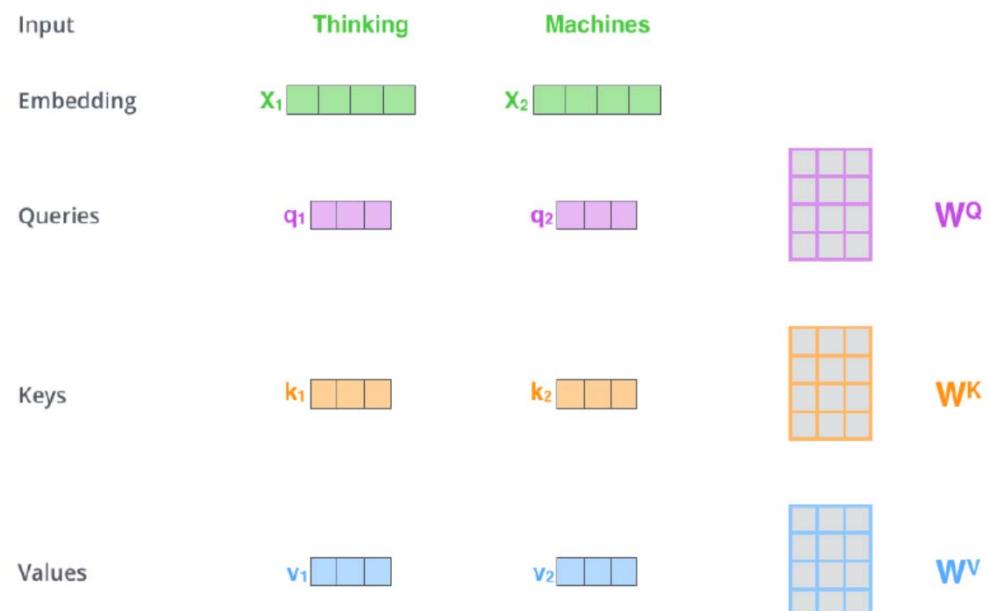
Self Attention

▶ Note

- ▶ Self Attention sees the input as a set and not as a sequence
- ▶ Permutation in the inputs simply results in a permutation of the outputs
- ▶ An additional mechanism should be used

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019) - Self Attention – Queries, keys, values

- ▶ Current transformers make use of a more complex self attention mechanism
- ▶ I. For each embedding x_i create 3 vectors as linear transformation of x_i : query, key, value
 - ▶ query: $q_i = W_q x_i$
 - ▶ key: $k_i = W_k x_i$
 - ▶ value: $v_i = W_v x_i$



Multiplying x_1 by the W^Q weight matrix produces q_1 , the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

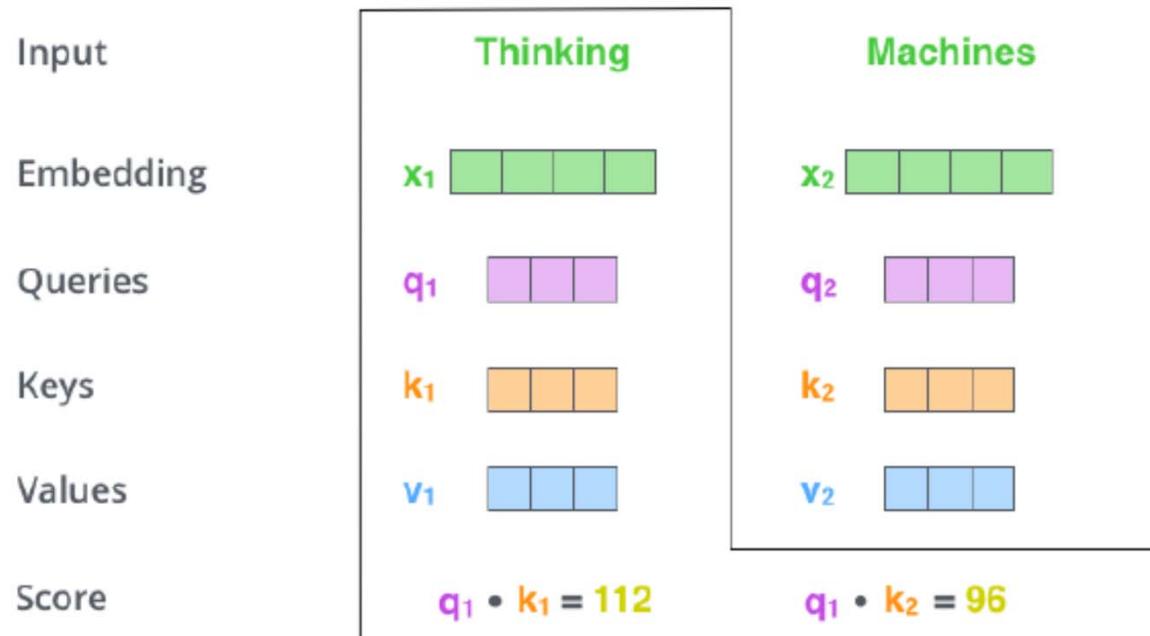
Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Self Attention

- ▶ x_i is used for three roles:
 - ▶ Query: it is compared to every vector to establish the weights for its own output z_i
 - ▶ Key: it is compared to every vector to establish the weights for the output z_j
 - ▶ Value: it is used in the weighted sum to compute each output vector
- ▶ Separating the roles in three vectors q_i, k_i, v_i all linear transformation of x_i gives more flexible model
 - ▶ q_i and k_j will be used for computing the attention score:
 - ▶ $e_{ij} = q_i \cdot k_j$
 - ▶ $\alpha_{ij} = \text{softmax}(e_{ij})$
 - ▶ v_j will be used for the output item
 - ▶ $z_i = \sum_j \alpha_{ij} v_j$

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

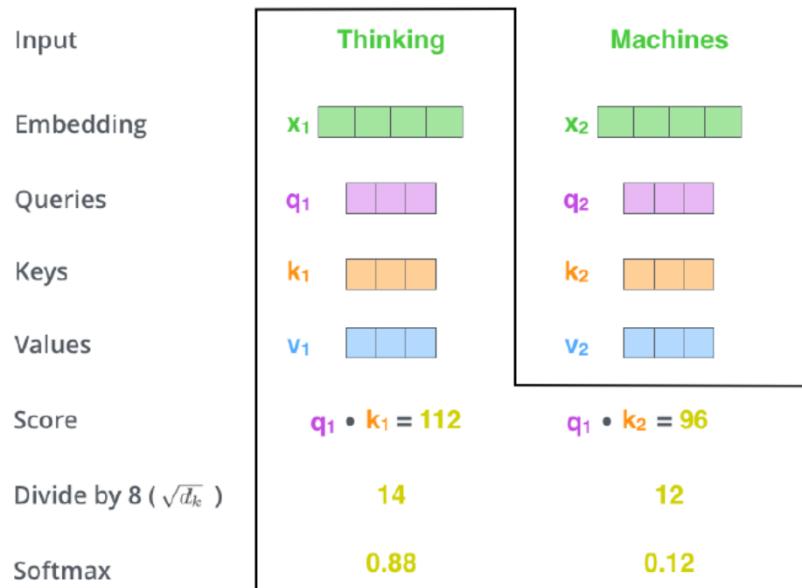
- ▶ 2. Compute score from query and key
 - ▶ Dot product of query and key value for each word
 - ▶ $e_{ij} = q_i \cdot k_j$



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

► 3. Normalize and softmax

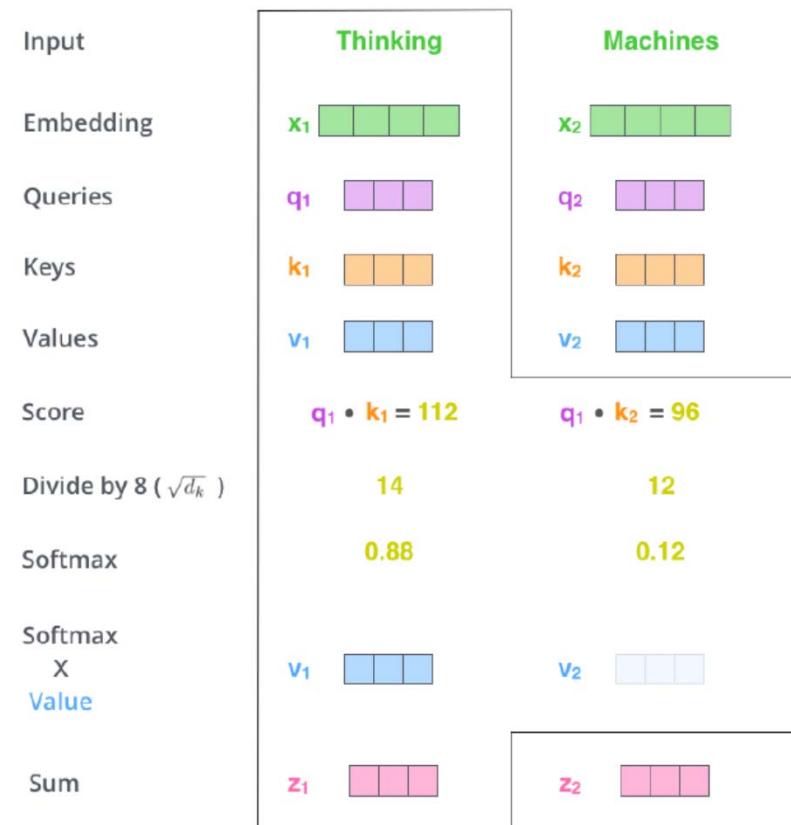
- Divide by the square root of the dimension of the key vectors (8 in the figure)
 - $e_{ij} = \frac{q_i \cdot k_j}{\sqrt{k}}$, with k the dimension of the q, k, v vectors
- Compute softmax
 - $\alpha_{ij} = \text{softmax}(e_{ij})$
- The softmax value indicates the weight of each word in the sequence for position 1 in the example



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

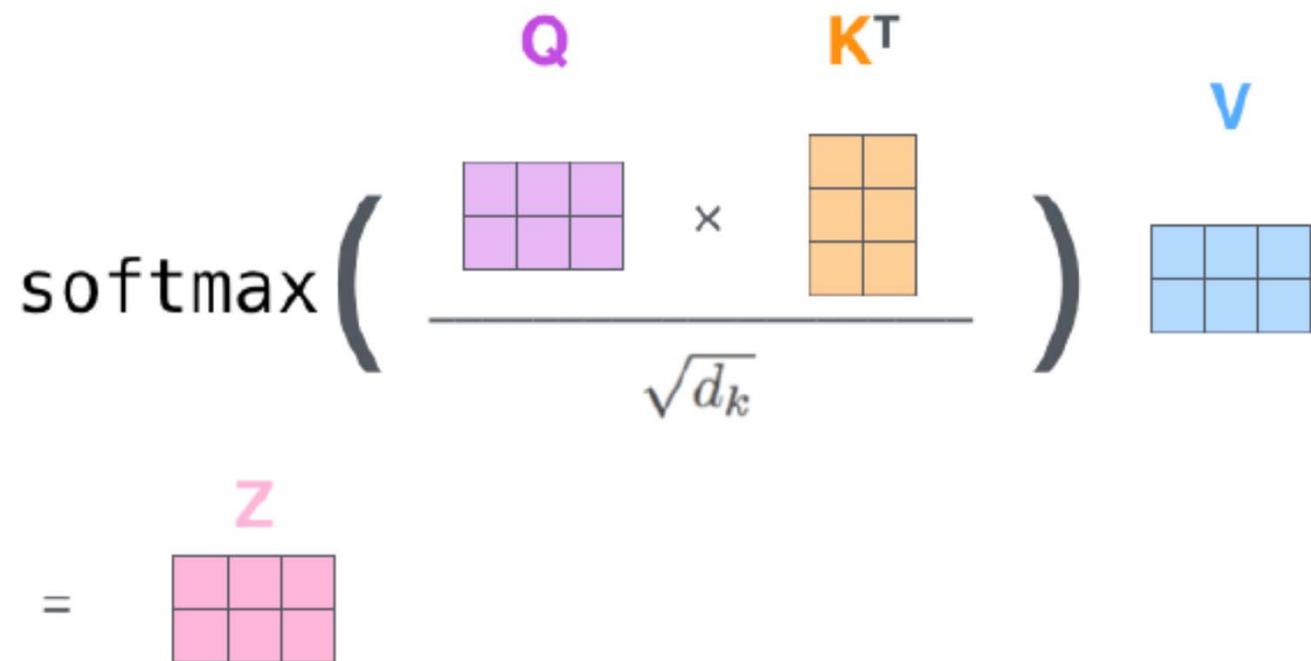
► 4. Compute the output of the self attention layer at position 1

- Multiply each value vector v by the softmax score
- Sum up the weighted value vectors
- $$z_i = \sum_j \alpha_{ij} v_j$$



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

- ▶ Compute the output of the self attention layer at position 1
 - ▶ Matrix form

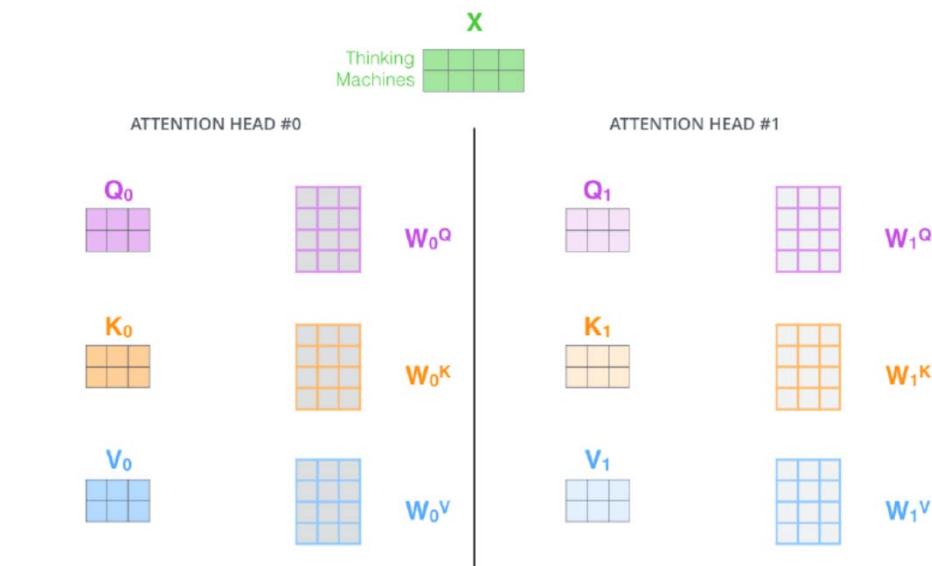
$$\text{softmax} \left(\frac{\begin{matrix} \mathbf{Q} & \mathbf{K^T} \\ \times & \end{matrix}}{\sqrt{d_k}} \right) \mathbf{V}$$
$$= \mathbf{Z}$$


The self-attention calculation in matrix form

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019))

▶ Multi-head self attention

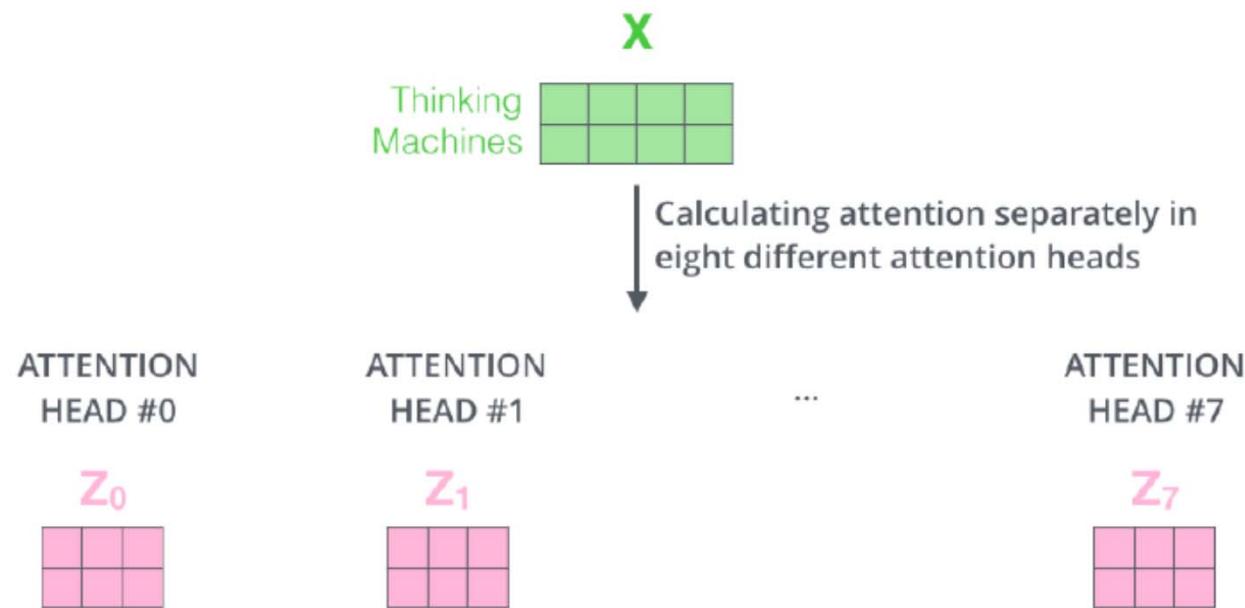
- ▶ Duplicate the self attention mechanism
- ▶ Allows to focus on different parts of the input sequence and to encode different relations between elements of the input sequence
- ▶ Matrices for the different heads are denoted W_q^r, W_k^r, W_v^r with r the index of head r



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the $WQ/WK/WV$ matrices to produce Q/K/V matrices.

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

- ▶ Compute one output for each head



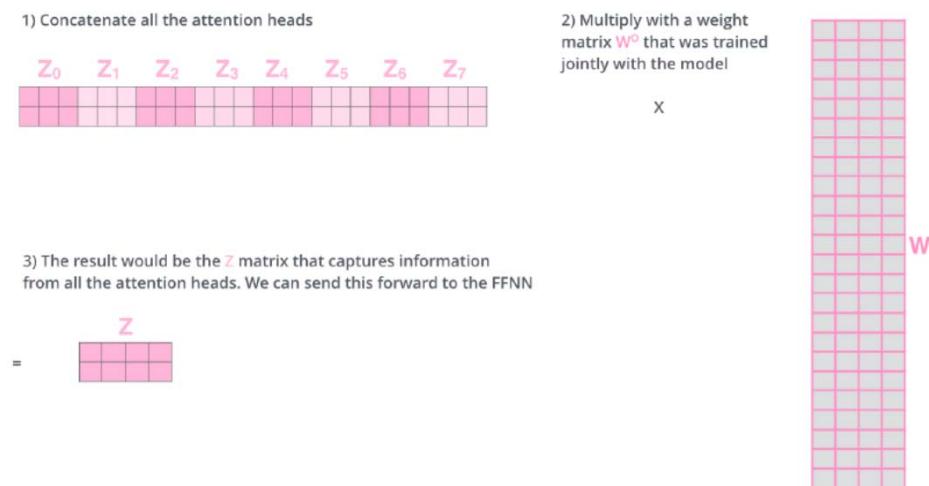
Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019))

- ▶ Multi-head self attention
- ▶ Two usual ways of applying multi-head
- ▶ 1. Cut the embedding vector x_i into chunks and generate q, k, v from each chunk
 - ▶ e.g. if the embedding is size 256 and we have 8 heads, each chunk will be of size 32, the W_q^r, W_k^r, W_v^r are of size 32x32
- ▶ 2. Apply each head to the whole vector
 - ▶ W_q^r, W_k^r, W_v^r are of size 256x256

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

▶ Global output

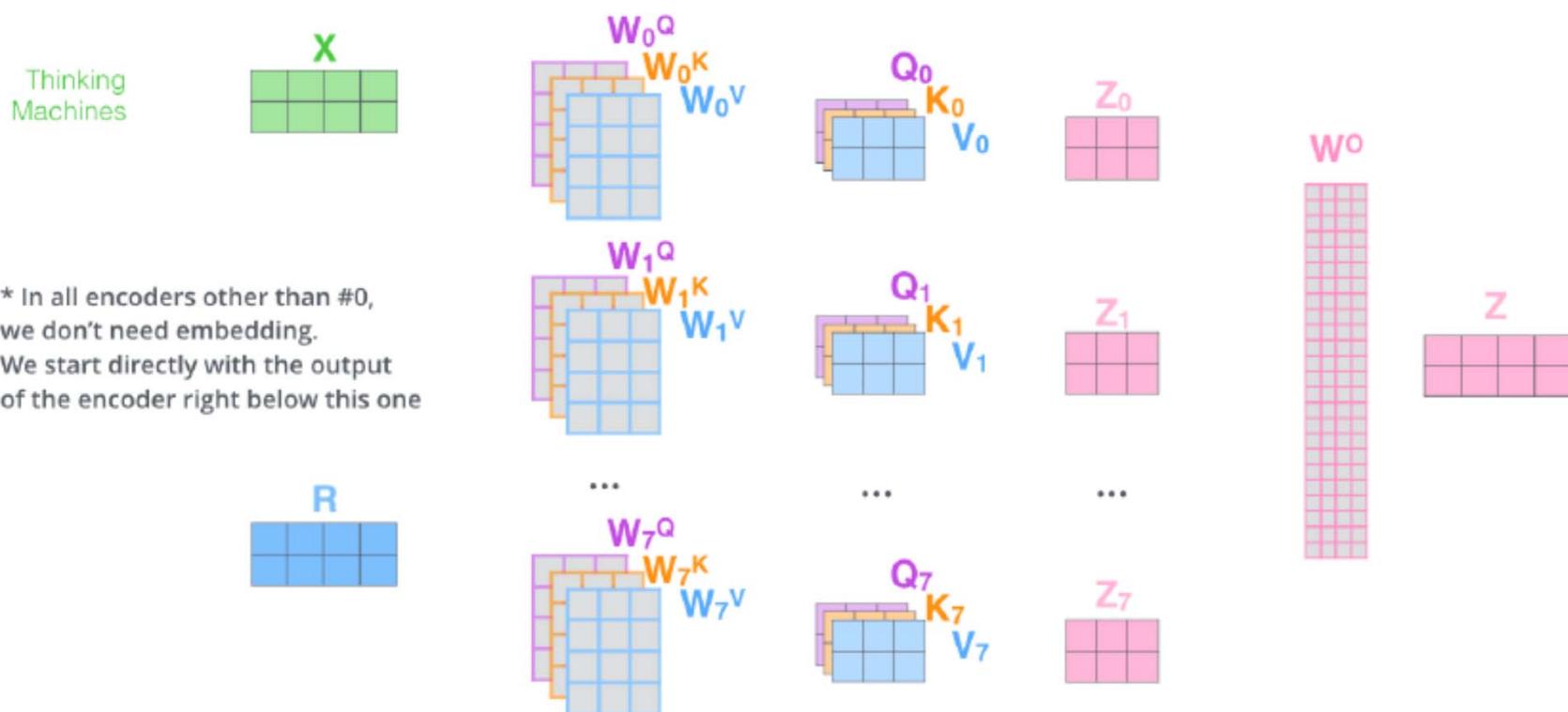
- ▶ Concatenate the individual head outputs
- ▶ Combine them with an additional matrix W^0 in order to produce an output of size k , for example the initial size of the embeddings



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

▶ Summary of multi-head self attention

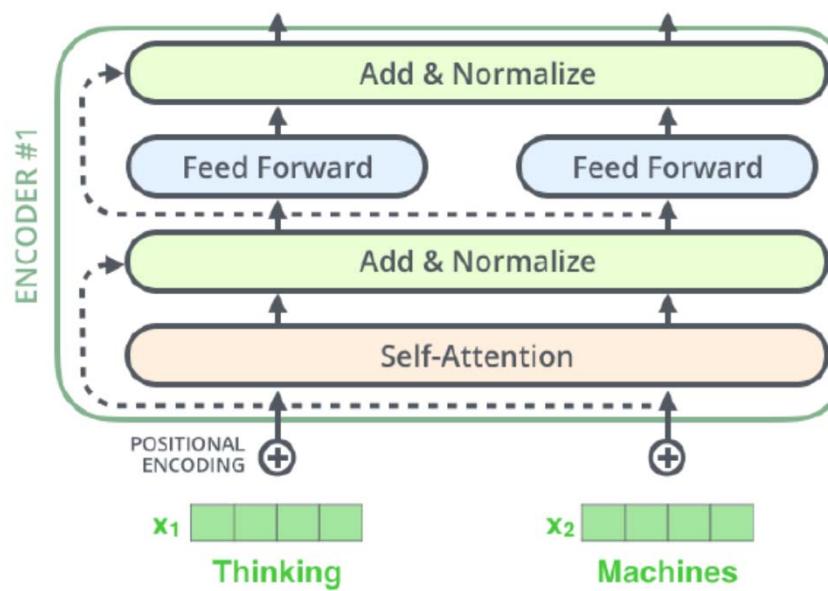
- 1) This is our input sentence* each word*
- 2) We embed each word*
- 3) Split into 8 heads. We multiply X or R with weight matrices
- 4) Calculate attention using the resulting $Q/K/V$ matrices
- 5) Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Transformer module

- ▶ A transformer module combines different operations and is roughly defined as follows (several variants)
- ▶ The example takes two word as input and outputs two transformed encodings

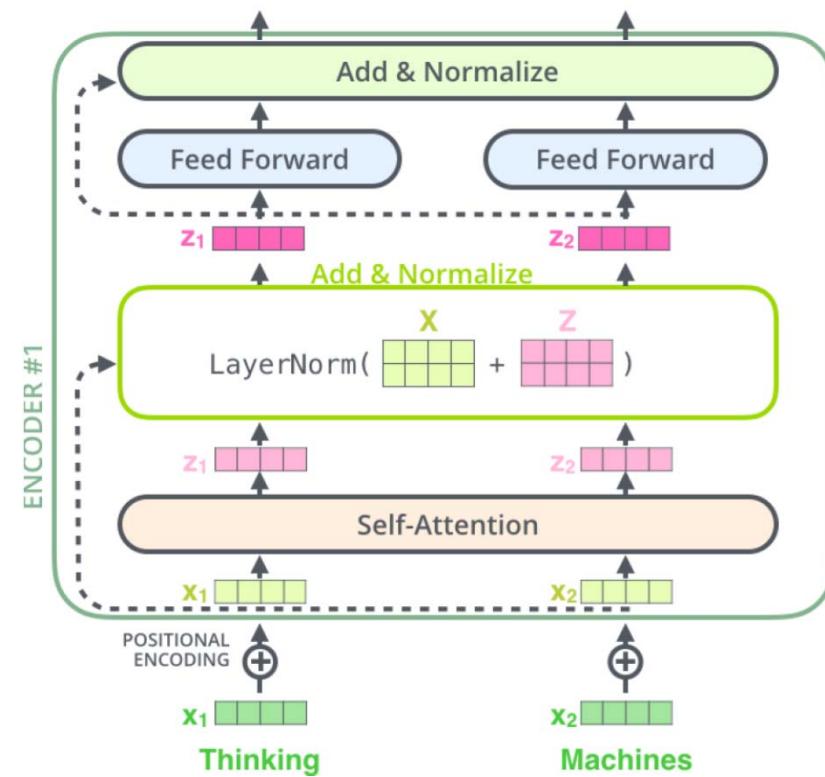


- Normalization layers (layer normalization)
- Multiple self attention modules per encoder
- Residual (skip) connections like in ResNet
- Positional encoding

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Transformer module

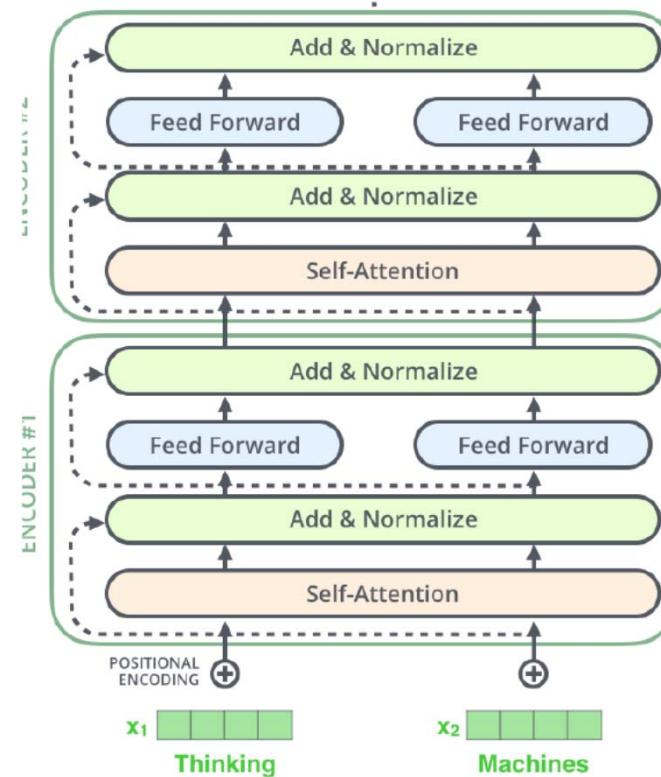
- ▶ Add and normalized detailed
- ▶ Residual connections are added before normalization



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Transformer architecture

- ▶ Stack multiple transformer modules



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Transformer architecture

▶ Attention: word dependencies

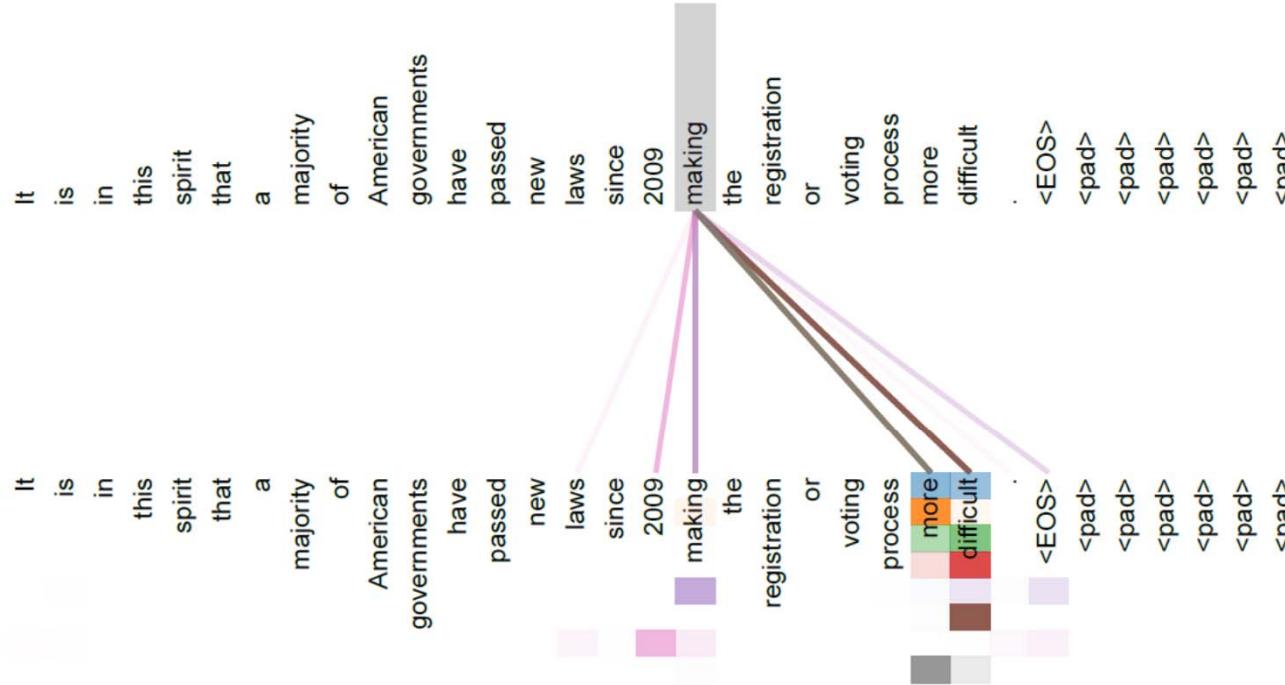


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb ‘making’, completing the phrase ‘making...more difficult’. Attentions here shown only for the word ‘making’. Different colors represent different heads. Best viewed in color.

Fig. (Vaswani 2017)

Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

▶ Positional encoding

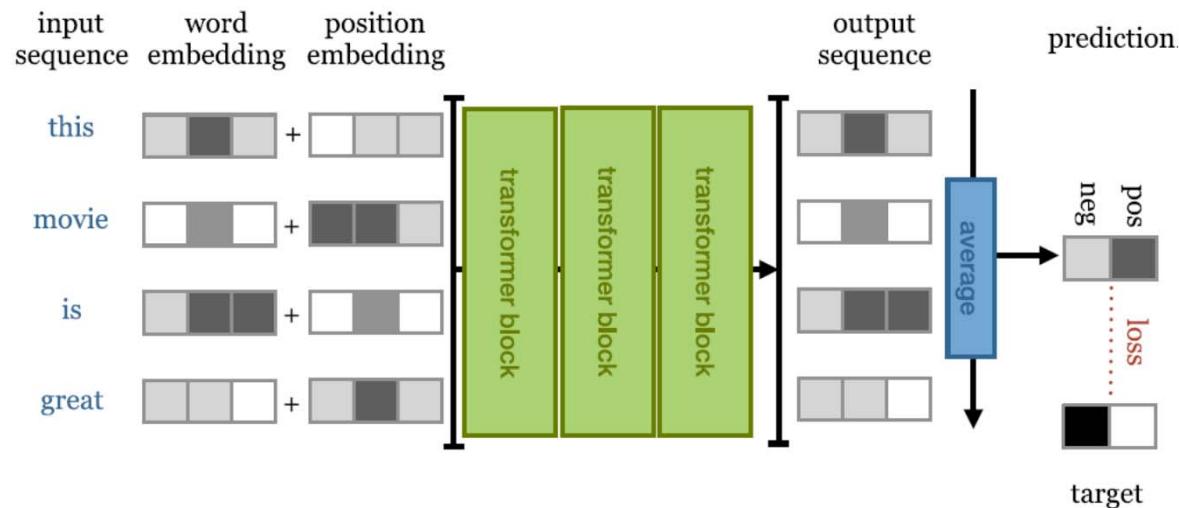
- ▶ In order to account for the word order, the model makes use of a positional encoding together with the first word embeddings (1st transformer module)
 - ▶ An information is added to each input embedding which helps determining the position of the word in the sentence.
 - ▶ This information is added to the input embeddings at the bottom of the transformer module
 - ▶ The encoding can be learned like word embeddings – requires learning one embedding for each position
 - ▶ The encoding can be defined according to some function $f: N \rightarrow R^k$
 - ▶ In the original transofmer paper, the encoding is defined as follows
 - Let PE denote the positional encoding, $PE \in R^d \times R^n$, i.e. vector of length n , size of the sequence, and each positional encoding is of size d (same size as embeddings v).
 - $PE_{-}(pos, 2i) = \sin(pos / 10000^{\frac{2i}{d}})$, $PE_{-}(pos, 2i + 1) = \cos(pos / 10000^{\frac{2i}{d}})$
 - With pos the position in the sequence and $i \in \{1, \dots, d\}$ the dimension in the position vector

Transformer networks

Example: classifier (Bloem 2019)

- ▶ Binary classifier for word sequences

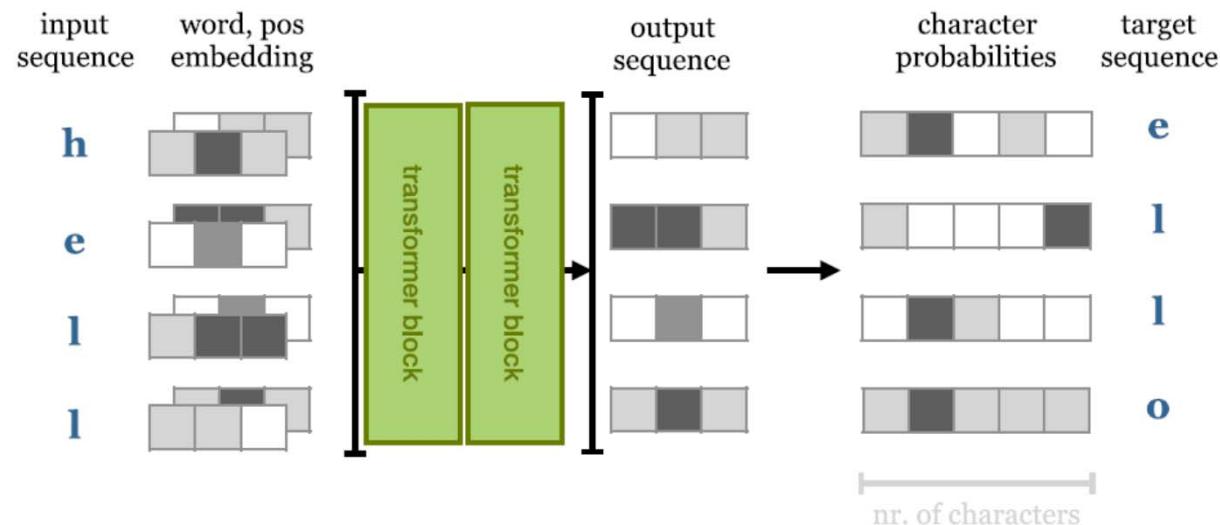
- ▶ Targets: positive/ negative
- ▶ The output sequence is averaged in order to produce a fixed size vector



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018, P. Bloem 2019)

Example: text generation transformer - autoregressive model

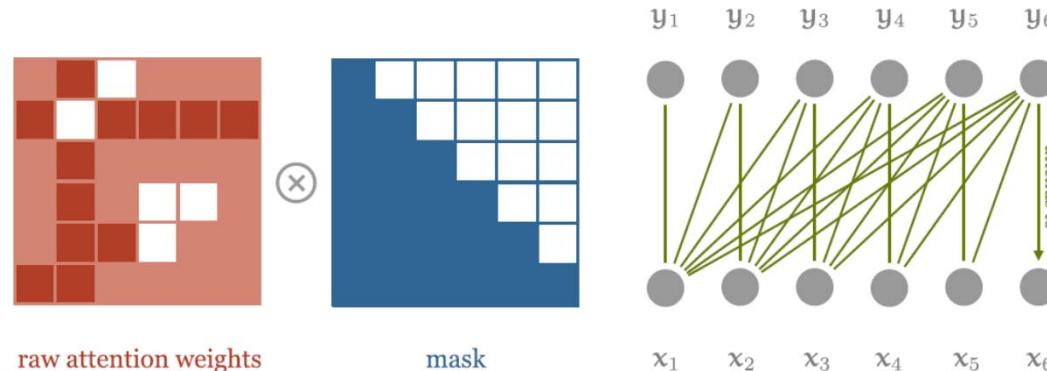
- ▶ Character level transformer for predicting next character from an input sequence
 - ▶ Input: a sequence
 - ▶ Output next character for each point in the sequence
 - ▶ i.e. the target sequence is the input shifted one character to the left
 - ▶ Example with a words vocabulary



Transformer networks

Example: text generation transformer - autoregressive model (Bloem 2019)

- ▶ Because the transformer has access to the whole « h e l l » sequence, prediction for « e l l » becomes trivial
- ▶ If one wants to learn an autoregressive model one should prevent the transformer to look forward in the sequence
- ▶ Character level transformer for predicting next character from an input sequence
- ▶ For that one makes use of a MASK to the matrix of ot products before the softmax in the self attention module



- ▶ Note: multiplication here is the elementwise multiplication

Transformer networks

Example: text generation transformer - autoregressive model (Bloem 2019)

- ▶ Example followed
- ▶ Training from sequences of length 256, using 12 transformer blocks and 256 embedding dimensions
- ▶ After training, let the model generate characters from a 256 character sequence seed
 - ▶ For a sequence of 256 input characters the Transformer generates a distribution for the new character (257^{e}).
 - ▶ Sample from this distribution and feed back to the input for predicting the next (258^{e}) character, etc

Sample output (training from 10^8 characters from Wikipedia including markups):

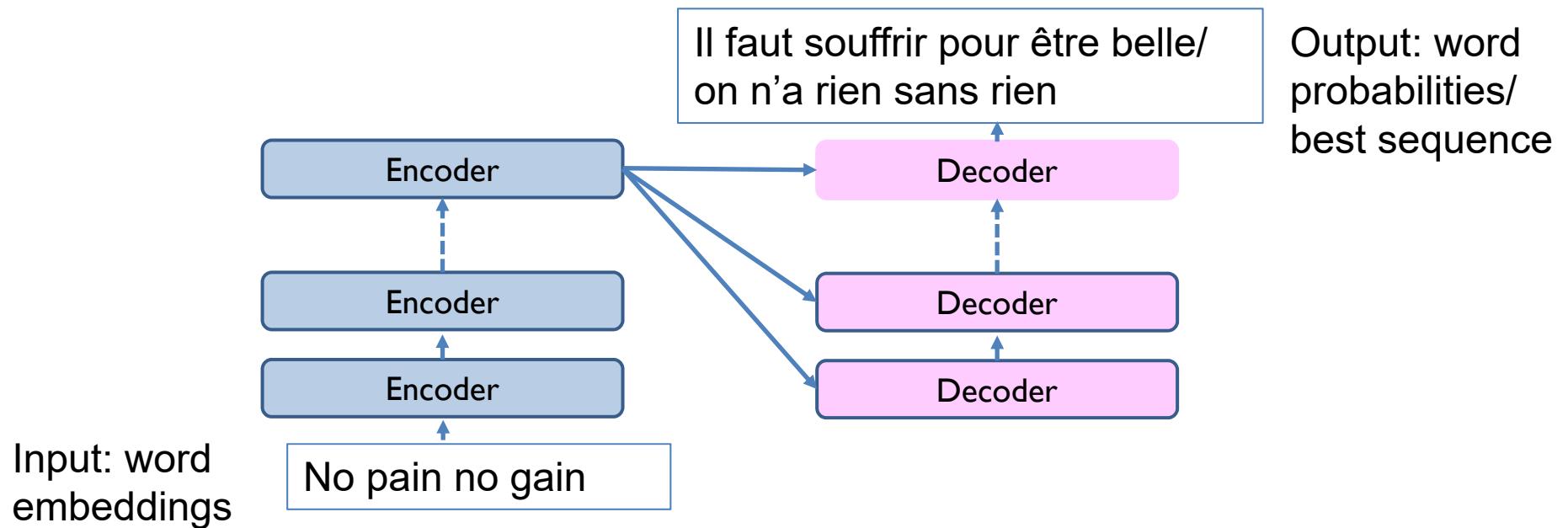
1228X Human & Rousseau. Because many of his stories were originally published in long-forgotten magazines and journals, there are a number of [[anthology|anthologies]] by different collators each containing a different selection. His original books have been considered an anthologie in the [[Middle Ages]], and were likely to be one of the most common in the [[Indian Ocean]] in the [[1st century]]. As a result of his death, the Bible was recognised as a counter-attack by the [[Gospel of Matthew]] (1177-1133),...

Transformer networks

- ▶ Large size development of Transformers
 - ▶ BERT
 - ▶ GPT2

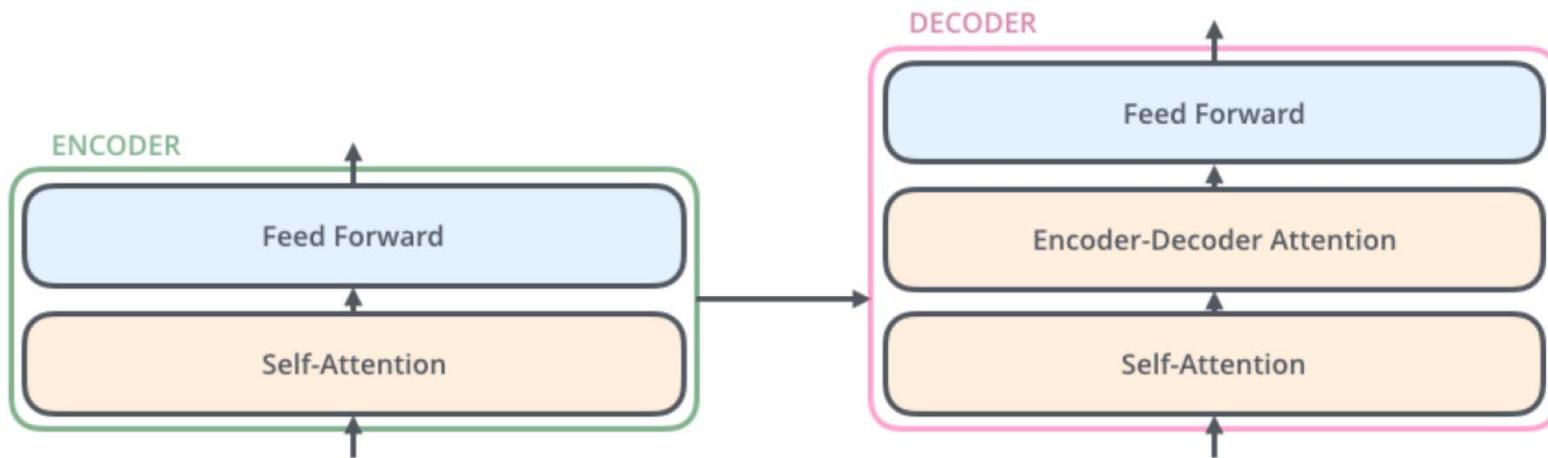
Historical side: Transformer networks (Vaswani 2017)

- ▶ The first implementation of Transformer was proposed by (Vaswani 2017) in an encoder-decoder scheme
- ▶ General scheme
 - ▶ Stacks of encoder/ decoder modules
 - ▶ Encoder (resp. decoders) have the same structure but do not share parameters



Transformer networks (Vaswani 2017, illustrations J. Alammar 2018)

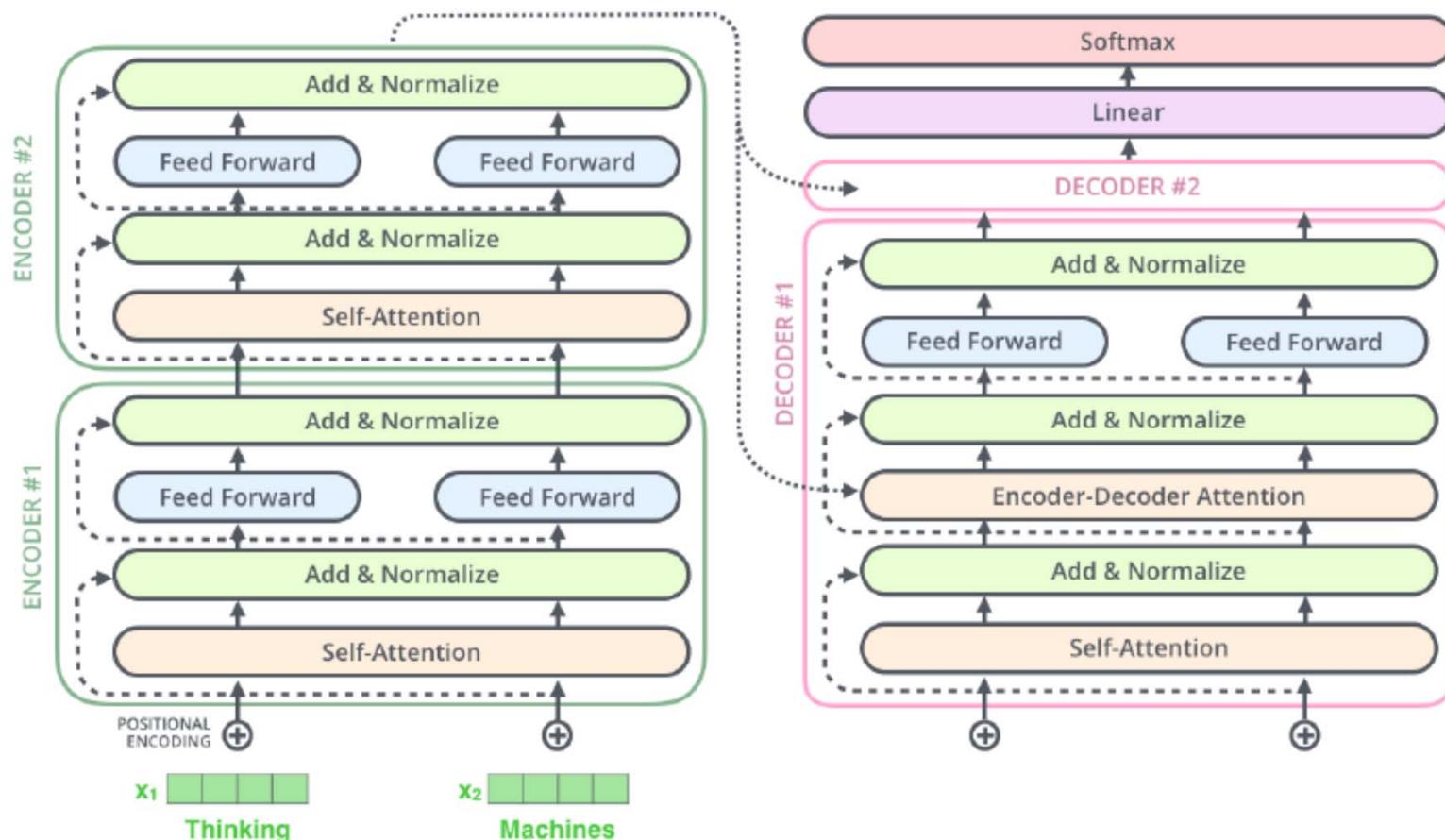
▶ Encoder-Decoder modules structure



- ▶ **Encoder**
 - ▶ Input flows through a self attention layer – encoding of a word in the sequence will depend on the other words
 - ▶ Outputs of the self attention layer are fed in a feed-forward NN. The same network is used for each word position
- ▶ **Decoder**
 - ▶ The decoder has an additional encoder-decoder attention layer that focuses on relevant parts of the input

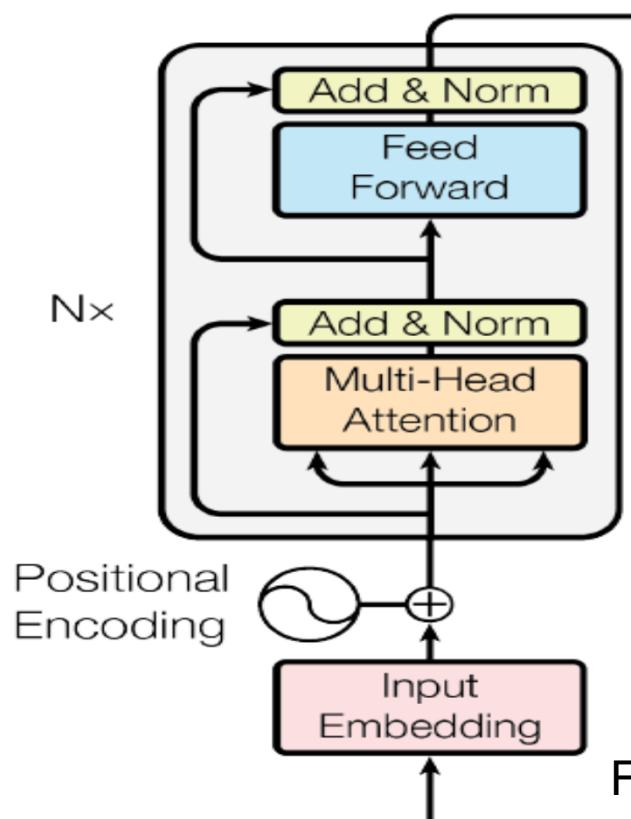
Historical side: Transformer networks (Vaswani 2017)
 illustration: J. Alammar 2018

▶ Encoder + Decoder modules



Historical side: Transformer networks (Vaswani 2017)

▶ Full encoder module



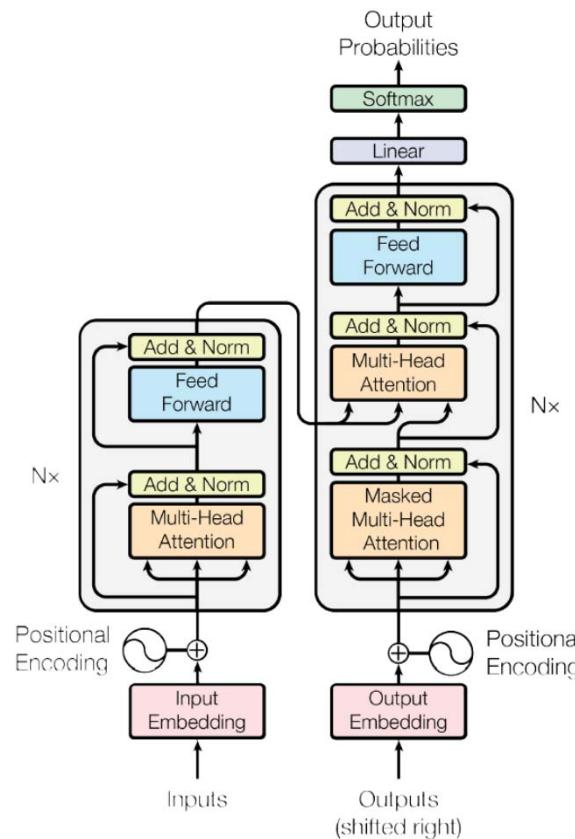
- Normalization layers
- Skip connections like in ResNet
- Multiple self attention modules per encoder
- Positional encoding
- Normalization layers
- Skip connections like in ResNet
- Multiple self attention modules per encoder/ multi-head
- Positional encoding

Fig. (Vaswani 2017)

Historical side: Transformer networks (Vaswani 2017)

▶ Decoder

- ▶ The encoder top layer outputs attention vectors (k, v) , they are processed by the decoder
- ▶ Two main changes w.r.t. encoder
 - ▶ Masked self-attention on previously generated outputs
 - The self attention can only attend to earlier positions in the output sequence, future positions are masked
 - ▶ Encoder – decoder attention
 - U comes from the previous decoder layer and keys/ values K, V from the encoder last layer output



References: papers used as illustrations for the presentation

- ▶ Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein Generative Adversarial Networks. In Proceedings of The 34th International Conference on Machine Learning (pp. 1–32). Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In ICCV (pp. 2223–2232).
- ▶ Badrinarayanan, V., Kendall, A., & Cipolla, R. (2017). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12), 2481–2495.
- ▶ Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation By Jointly Learning To Align and Translate. In Iclr 2015. <https://doi.org/10.1146/annurev.neuro.26.041002.131047>
- ▶ Baydin Atilim Gunes , Barak A. Pearlmutter, Alexey Andreyevich Radul, Automatic differentiation in machine learning: a survey. CoRR abs/1502.05767 (2017)
- ▶ Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2016). Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5, 135–146.
- ▶ Cadène R., Thomas Robert, Nicolas Thome, Matthieu Cord:M2CAI Workflow Challenge: Convolutional Neural Networks with Time Smoothing and Hidden Markov Model for Video Frames Classification. CoRR abs/1610.05541 (2016)
- ▶ Chen M. Denoyer L., Artieres T. Multi-view Generative Adversarial Networks without supervision, 2017 , <https://arxiv.org/abs/1711.00305>.
- ▶ Chen, L. C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2018). DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4), 834–848. <https://doi.org/10.1109/TPAMI.2017.2699184>
- ▶ Cho, K., Gulcehre, B. van M.C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. 2014. Learning Phrase Representations using RNN Encoder – Decoder for Statistical Machine Translation. EMNLP 2014 (2014), 1724–1734.
- ▶ Cybenko, G. (1993). Degree of approximation by superpositions of a sigmoidal function. *Approximation Theory and Its Applications*, 9(3), 17–28.
- ▶ Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. In arxiv.org/abs/1603.07285 (pp. 1–31).
- ▶ Durand T. , Thome, N. and Cord M., WELDON: Weakly Supervised Learning of Deep Convolutional Neural Networks, CVPR 2016.
- ▶ Frome, A., Corrado, G., Shlens, J., Bengio, S., Dean, J., Ranzato, M.A. and Mikolov, T. 2013. DeViSE: A Deep Visual-Semantic Embedding Model. NIPS 2013 (2013).
- ▶ Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. In CVPR (pp. 2414–2423).

References: papers used as illustrations for the presentation

- ▶ Goodfellow I, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio , Generative adversarial nets, NIPS 2014, 2672-2680
- ▶ Goodfellow, I., Pouget-Abadie, J., & Mirza, M. (2014). Generative Adversarial Networks. NIPS, 2672--2680.
- ▶ He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016a. Deep residual learning for image recognition. In CVPR, 770–778.
- ▶ He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016b. Identity mappings in deep residual networks. In ECCV, 630–645.
- ▶ He, K., Gkioxari, G., Dollar, P., & Girshick, R. (2017). Mask R-CNN. Proceedings of the IEEE International Conference on Computer Vision, 2017–Octob, 2980–2988.
- ▶ Hornik, K. (1991). Approximation Capabilities of Multilayer Feedforward Networks [J]. Neural Networks, 4(2), 251–257.
- ▶ Ioffe S., Szegedy C.: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 1995, <http://arxiv.org/abs/1502.03167>
- ▶ Jalammar 2018 - <http://jalammar.github.io/illustrated-transformer/>
- ▶ Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Zitnick, C. L., & Girshick, R. (2017). CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning. In CVPR (pp. 1988–1997). <https://doi.org/10.1109/CVPR.2017.215>
- ▶ Johnson, J., Hariharan, B., van der Maaten, L., Hoffman, J., Fei-Fei, L., Zitnick, C. L., & Girshick, R. (2017). Inferring and Executing Programs for Visual Reasoning. In ICCV (pp. 3008–3017). <https://doi.org/10.1109/ICCV.2017.325>
- ▶ Krizhevsky, A., Sutskever, I. and Hinton, G. 2012. Imagenet classification with deep convolutional neural networks. Advances in Neural Information. (2012), 1106–1114.
- ▶ Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J. and Ng, A. 2012. Building high-level features using large scale unsupervised learning. Proceedings of the 29th International Conference on Machine Learning (ICML-12). (2012), 81–88.
- ▶ Lerer, A., Gross, S., & Fergus, R. (2016). Learning Physical Intuition of Block Towers by Example. In Icml (pp. 430–438). Retrieved from <http://arxiv.org/abs/1603.01312>
- ▶ Lin, M., Chen, Q., & Yan, S. (2013). Network In Network. In arxiv.org/abs/1312.4400. <https://doi.org/10.1109/ASRU.2015.7404828>
- ▶ Lin, Z., Feng, M., Santos, C. N. dos, Yu, M., Xiang, B., Zhou, B., & Bengio, Y. (2017). A Structured Self-attentive Sentence Embedding. In ICLR.
- ▶ Mathieu, M., Couprie, C., & LeCun, Y. (2016). Deep multi-scale video prediction beyond mean square error. In ICLR (pp. 1–14). Retrieved from <http://arxiv.org/abs/1511.05440>
- ▶ Mirza, M., & Osindero, S. (2014). Conditional Generative Adversarial Nets. In arxiv.org/abs/1411.1784.
- ▶ Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. In NIPS Deep Learning Workshop. <https://doi.org/10.1038/nature14236>
- ▶ Pearlmutter B.A., Gradient calculations for dynamic recurrent neural networks: a survey, IEEE Trans on NN, 1995

References: papers used as illustrations for the presentation

- ▶ Pennington, J., Socher, R. and Manning, C.D. 2014. GloVe : Global Vectors for Word Representation. EMNLP 2014 (2014), 1532–1543.
- ▶ Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In arxiv.org/abs/1511.06434 (pp. 1–15). <https://doi.org/10.1051/0004-6361/201527329>
- ▶ Radford, Luke Metz, Soumith Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2016, <http://arxiv.org/abs/1511.06434>
- ▶ Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. In CVPR (pp. 779–788).
- ▶ Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B. and Lee, H. 2016. Generative Adversarial Text to Image Synthesis. Icml (2016), 1060–1069.
- ▶ Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., & Lee, H. (2016). Generative Adversarial Text to Image Synthesis. In Icml (pp. 1060–1069).
- ▶ Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In MICCAI 2015: Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015 (pp. 234–241).
- ▶ Ruder S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.
- ▶ Shelhamer, E., Long, J., Darrell, T., Shelhamer, E., Darrell, T., Long, J., ... Darrell, T. (2015). Fully Convolutional Networks for Semantic Segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 3431–3440).
- ▶ Srivastava N., Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research 15(1): 1929-1958 (2014)
- ▶ Sutskever, I., Vinyals, O. and Le, Q. V 2014. Sequence to sequence learning with neural networks. Advances in Neural Information Processing Systems (NIPS) (2014), 3104–3112.
- ▶ Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention Is All You Need. In NIPS.
- ▶ Vinyals, O., Toshev, A., Bengio, S. and Erhan, D. 2015. Show and Tell: A Neural Image Caption Generator, CVPR 2015: 3156-3164
- ▶ Widrow, B., Glover, J. R., McCool, J. M., Kaunitz, J., Williams, C. S., Hearn, R. H., ... Goodlin, R. C. (1975). 1975 Adaptive noise cancelling: Principles and applications. Proceedings of the IEEE, 63(12), 1692–1716. <https://doi.org/10.1109/PROC.1975.10036>
- ▶ Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, Jeffrey Dean, Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, Technical Report, 2016.

References: papers used as illustrations for the presentation

- ▶ Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., ... Bengio, Y. (2015). Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Icmi-2015* (pp. 2048–2057). <https://doi.org/10.1109/72.279181>
- ▶ Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., & Hovy, E. (2016). Hierarchical Attention Networks for Document Classification. In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language
- ▶ Yu, F., & Koltun, V. (2016). Multi-Scale Context Aggregation by Dilated Convolutions. In *ICLR*, arxiv.org/abs/1511.07122.