Introduction to Deep Learning

Sorbonne Université - Master 2 - Probabilités et Finance
- P. Gallinari, patrick.gallinari@lip6.fr, http://www-connex.lip6.fr/~gallinar/
Olivier Schwander, olivier.schwandler@lip6.fr
Année 2021-2022

Course Outline and Organization

- Introductory ML course with a focus on Neural Networks and Deep Learning
- Organization
  - Part I: Introduction and classical machine learning by B. Wilbertz
  - Part 2: Neural Networks and Deep Learning – P. Gallinari – O. Schwander
    - Courses 15 h
    - Practice and exercises 9 h
- Outline
  - Neural Networks and Deep Learning
    - Introductory Concepts - Perceptron-Adaline
    - Linear Regression and Logistic Regression - Optimization Basics
    - Multilayer Perceptrons – Generalization Properties
    - Convolutional Neural Networks – Vision applications
    - Recurrent Neural Networks – Language applications
    - Unsupervised Learning: Generative Models

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
      - Chapters3, 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

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)

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

Context -Examples
Language

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd 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 shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

train more

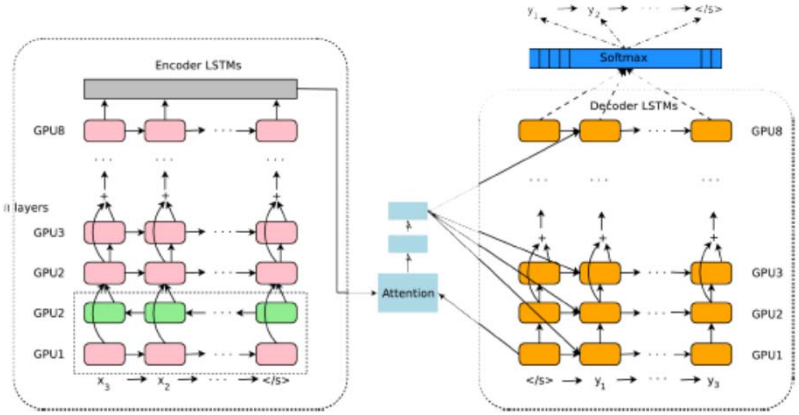
```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

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.
```

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)

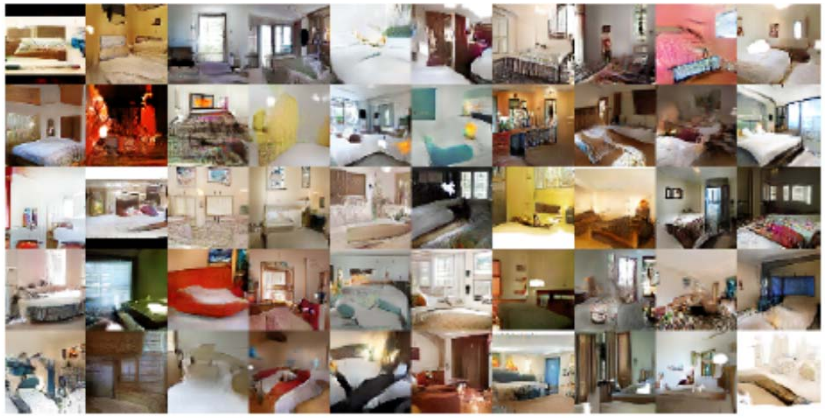
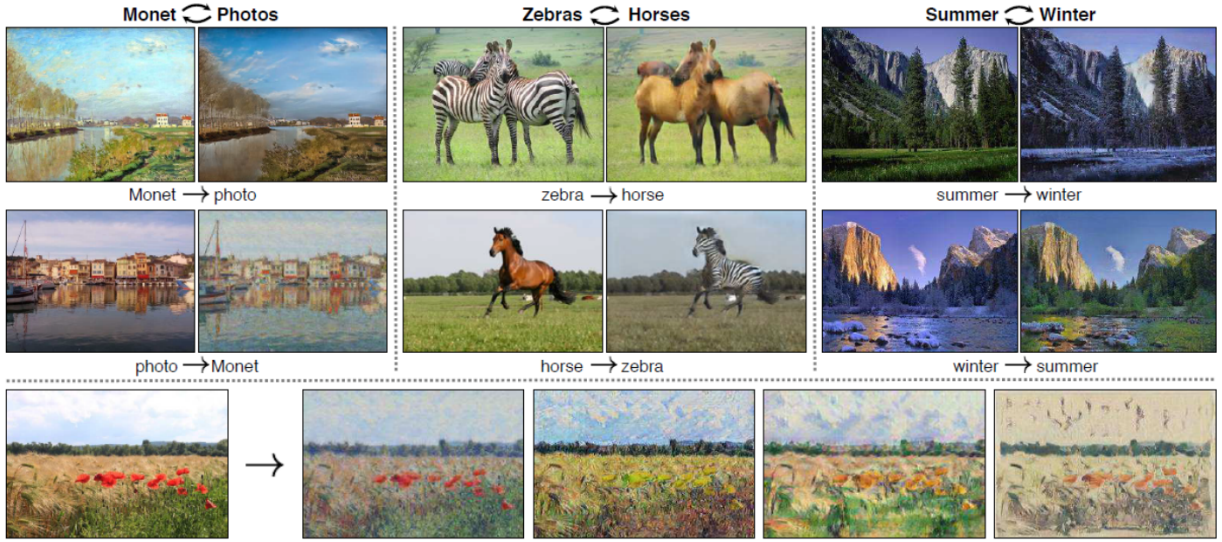


Figure 3: Generated bedrooms after five epochs of training. There appears to be evidence of visual under-fitting via repeated noise textures across multiple samples such as the base boards of some of the beds.

CycleGan
Image Translation, (Zhu 2017)



Context-Examples
Games (not considered in this course)

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

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

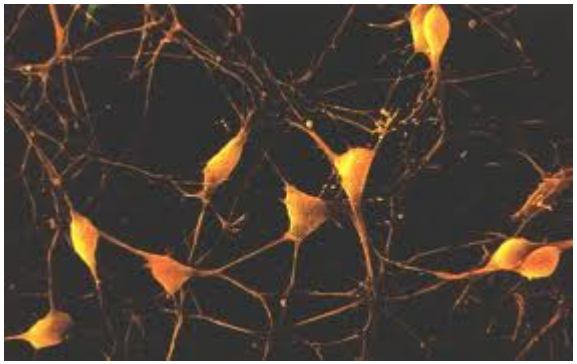
AlphaGo, AlphaGo Zero, Alpha Zero from Google DeepMind (2015, 2017)

Introductory NN concepts

Intuitive introduction via 2 simple –historical- models

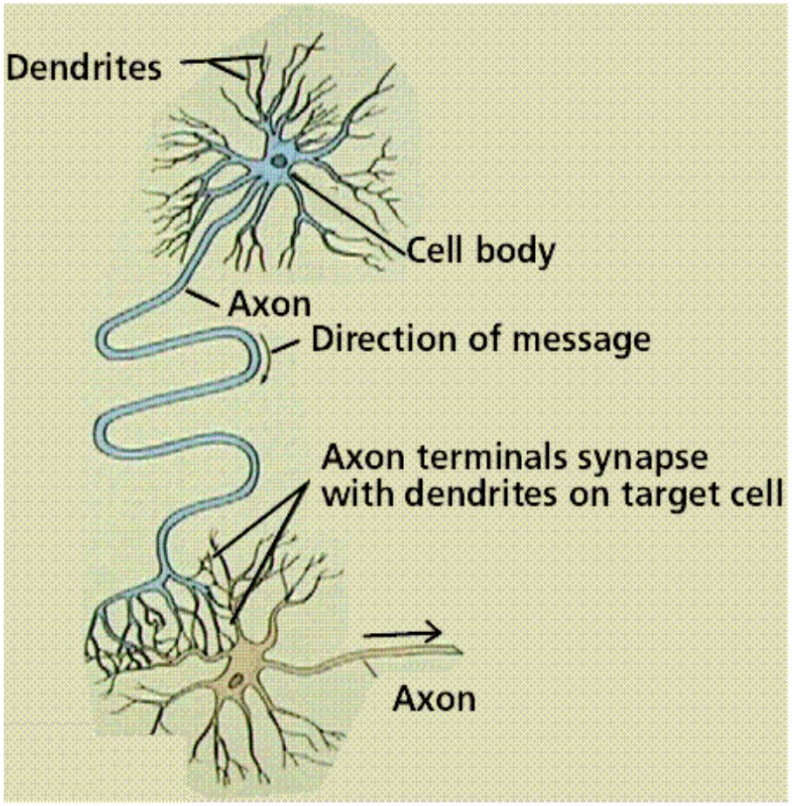
Perceptrons and Adalines

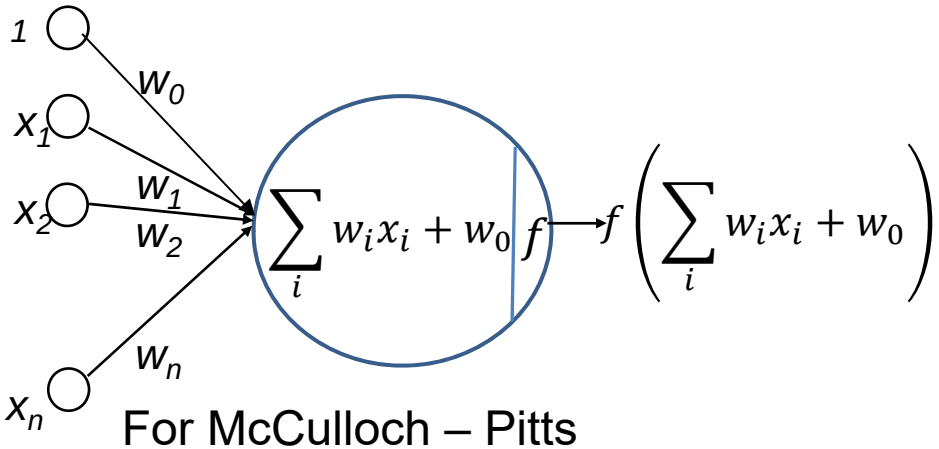
Neural Networks inspired Machine Learning
Brain metaphor



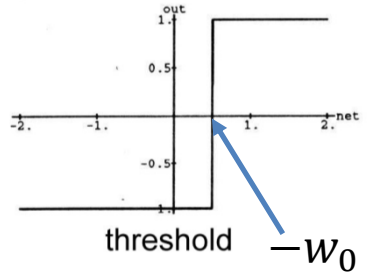
- Artificial Network Networks are an important paradigm in Statistical Machine learning and Artificial Intelligence
- Human brain is used as a source of inspiration and as a **metaphor** for developing Artificial NN
  - Human brain is a dense network 10^{11} of simple computing units, the neurons. Each 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





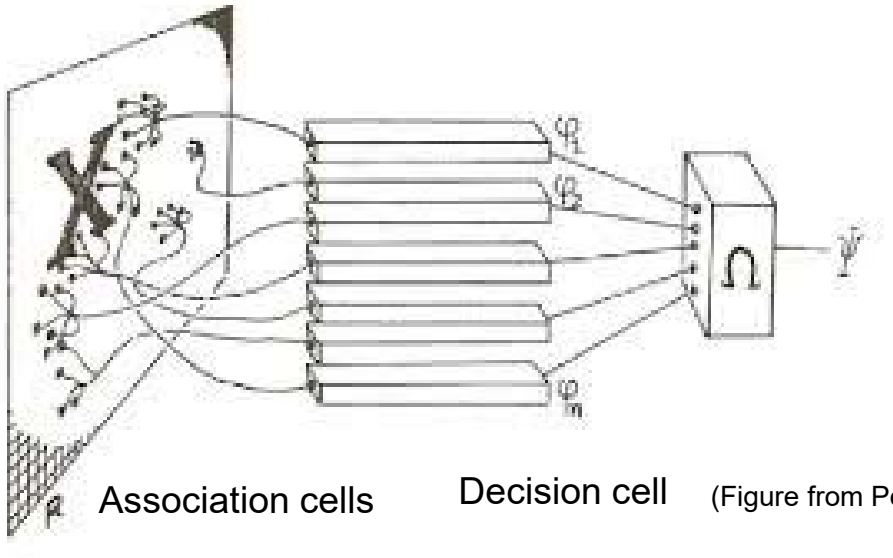
For McCulloch – Pitts neuron, f is a threshold (sign) function



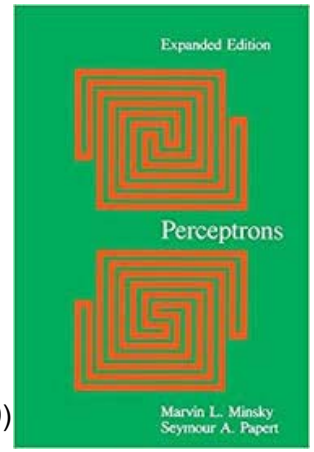
$$f(x) = \begin{cases} 1 \; if \sum_i (w_i x_i + w_0) > 0 \\ -1 \; otherwise \end{cases}$$

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

Perceptron (1958 Rosenblatt)

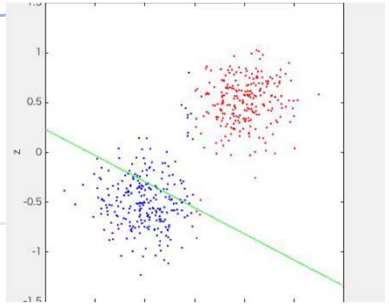


(Figure from Perceptrons, Minsky and Papert 1969)



- The decision cell is a threshold function (McCulloch – Pitts neuron)
  - $F(\boldsymbol{x}) = sgn(\sum_{i=1}^{n} w_i x_i + \mathrm{w}_0)$
- This simple perceptron can perform 2 classes classification

Perceptron Algorithm (2 classes)



Data

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

Output

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

Training set
Classifier specificatio

Initialize $\boldsymbol{w}$ (0)

Repeat (t)

Choose an example $(\boldsymbol{x}(t), y(t))$

if $y(t)\boldsymbol{w}(t).\boldsymbol{x}(t) \leq 0$ then $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \epsilon y(t)\boldsymbol{x}(t)$

Stochastic Algorithm

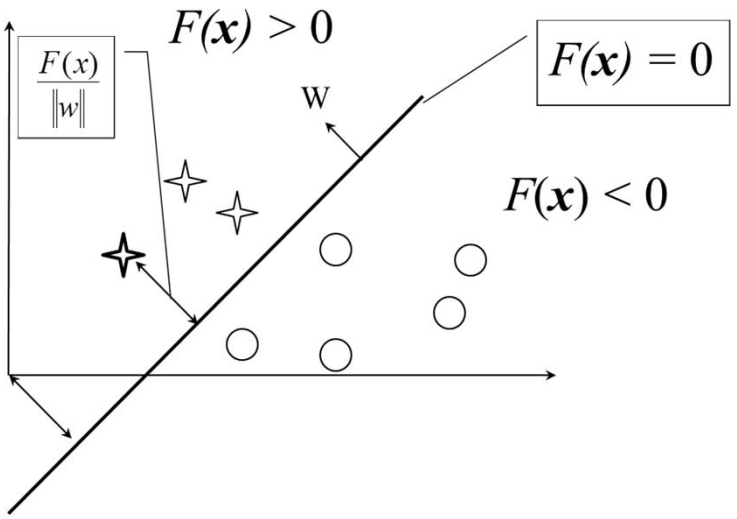
Until convergence

- 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

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

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



Perceptron algorithm performs a stochastic gradient descent

- Loss function
  - $C = -\sum_{(x,y)\text{missclassified}} \boldsymbol{w}.\boldsymbol{x}y = -\sum_{(x,y)\text{miss-classified}} \mathrm{c}(\boldsymbol{x}, \mathrm{y})$
  - Objective : minimize C
- gradient
  - $grad_{\boldsymbol{w}} C = \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n}\right)^T$ with $\frac{\partial C}{\partial w_i} = -\sum_{(x,d)\text{missclassified}} \boldsymbol{x}y$
- Learning rule
  - Stochastic gradient descent for minimizing loss C
  - Repeat (t)
    - Choose an example $(\boldsymbol{x}(t), y(t))$
    - $\boldsymbol{w}(\mathrm{t}) = \boldsymbol{w}(\mathrm{t}-1) - \epsilon\, grad_{\boldsymbol{w}} \mathrm{c}(\boldsymbol{x}, \mathrm{y})$

Multi-class generalization

- Usual approach: one vs all
  - p classes = p " 2 class problems " : class C_i against the others
    - Learn p discriminant functions $F_i(\boldsymbol{x}), i = 1 \ldots p$
    - Decision rule: $\boldsymbol{x} \in Ci$ if $F_i(\boldsymbol{x}) > F_j(\boldsymbol{x})$ 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 $D = \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^N, y^N)\}$ a data sample. If
    - $R = \max_{1 \le i \le N} \|\boldsymbol{x}^i\|$
    - $\sup_{\boldsymbol{w}} \min_i y^i (\boldsymbol{w}.\boldsymbol{x}^i) > \rho$ (ρ is called a margin)
    - The training sequence is presented a sufficient number of time
  - The algorithm will converge after at most $\left[\frac{R^2}{\rho^2}\right]$ 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 $l \le \frac{1+4\ln R/\rho - \ln \eta}{-\ln(1-\epsilon)} [R^2/\rho^2]$ steps
    - with probability $1-\eta$, test error is less than ϵ

Link between training and generalization performance

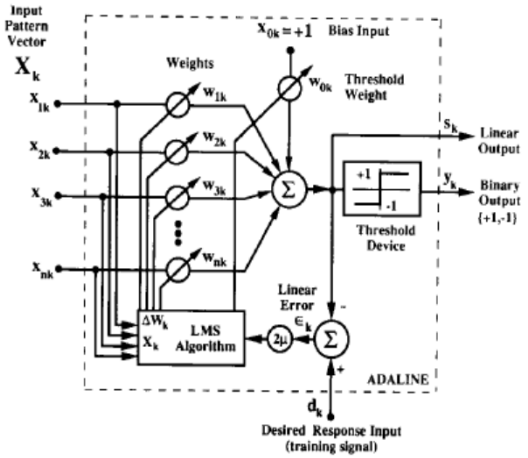
Convergence proof (Novikof)

- Hyp: lets take w^* / $\|w^*\| = 1$
  - $w_0 = 0, w_{t-1}$ is the weight vector before the t^{th} correction
  - $w_t = w_{t-1} + \epsilon y^t x^t$
  - $w_t . w^* = w_{t-1} . w^* + \epsilon y^t x^t . w^* \geq w_{t-1} . w^* + \epsilon\rho$
  - By induction $w_t . w^* \geq t\epsilon\rho$

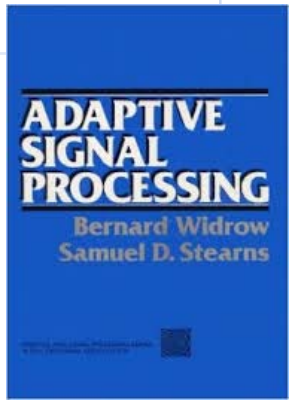
  - $\|w_t\|^2 = \|w_{t-1}\|^2 + 2\epsilon y^t w_{t-1} . x^t + \epsilon^2 \|x^t\|^2$
  - $\|w_t\|^2 \leq \|w_{t-1}\|^2 + \epsilon^2 \|x_t\|^2$ since $y^t w_{t-1} . x^t < 0$ (remember that x^t is incorrectly classified)
  - $\|w_t\|^2 \leq \|w_{t-1}\|^2 + \epsilon^2 \mathrm{R}^2$
  - By induction $\|w_t\|^2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

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

Adaline – Adaptive Linear Element
(Widrow - Hoff 1959)



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

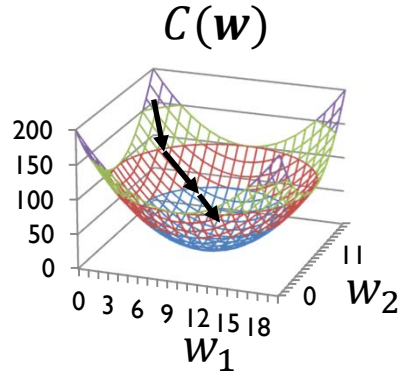


- « Least Mean Square » LMS algorithm
  - Loss: $c(\boldsymbol{x}, \boldsymbol{y}) = ||\boldsymbol{y} - F(\boldsymbol{x})||^2$
  - Algorithm: Stochastic Gradient Descent (Robbins – Monro (1951))

Iterate

Choose an example $(\boldsymbol{x}(t), y(t))$

$w(t+1) = w(t) - \epsilon \nabla_w \, c(\boldsymbol{x}, \boldsymbol{y})$



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

Adaline example motivating the need for adaptivity from an engineering perspective

- **Adaptive** noise cancelling

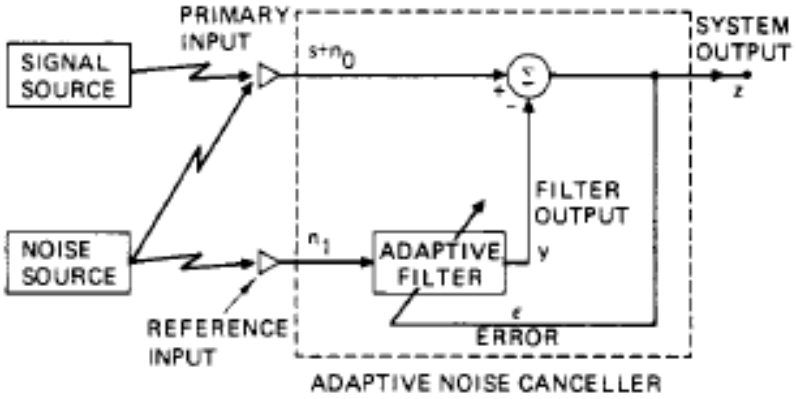


Fig. 1. The adaptive noise cancelling concept.

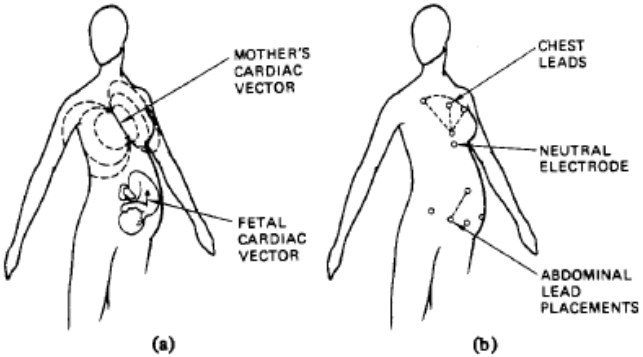


Fig. 14. Cancelling maternal heartbeat in fetal electrocardiography. (a) Cardiac electric field vectors of mother and fetus. (b) Placement of leads.

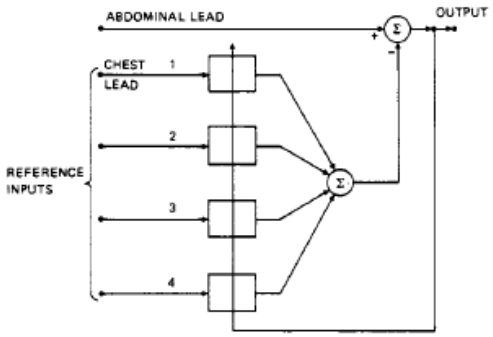


Fig. 15. Multiple-reference noise canceller used in fetal ECG experiment.

Widrow-Science in Action - YouTut

Fig. from Adaptive Signal Processing, Widrow, Stearn

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
  - $Min\ E[z^2] = E[s^2] + Min\ E[(n_0 - y)^2]$
- 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 $z - s = n_0 - y$)

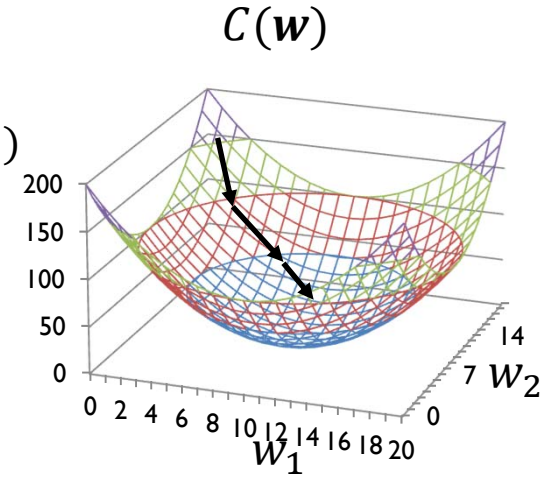
Introductory concepts
Summary of key ideas

- 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 $(\boldsymbol{x}, \boldsymbol{y})$
- 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 ≠ Optimization
  - One wants to learn functions that generalize well

Optimisation : gradient methods – introduction

Optimization
Batch gradient algorithms

- Batch gradient general scheme
  - Training Data Set
    - $D = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^N, \boldsymbol{y}^N)\}$
  - Objective
    - Optimize a loss function $C(\boldsymbol{w}) = \sum_{i=1}^{N} c_{\mathrm{w}}(\boldsymbol{x}^i, \boldsymbol{y}^i)$
      - Sum of invidual losses $c_{\mathrm{w}}(.,.)$ on each example $(\boldsymbol{x}^i, \boldsymbol{y}^i)$
  - Principle
    - Initialize $\boldsymbol{w} = \boldsymbol{w}(0)$
    - Iterate until convergence
      - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \epsilon(t)\boldsymbol{\Delta}_{\mathbf{w}}(t)$



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

Optimization
Batch second order gradients

- Consider a quadratic approximation of the loss function
  - C is approximated via a parabola
    - $C(w) = \mathrm{C}\big(\mathrm{w(t)}\big) + \big(\mathrm{w} - \mathrm{w(t)}\big)^{\mathrm{T}} \nabla C\big(w(t)\big) + \frac{1}{2}\big(\mathrm{w} - \mathrm{w(t)}\big)^{T} H\big(\mathrm{w} - \mathrm{w(t)}\big)$
    - where $\mathrm{w(t)}$ is the parameter vector at time t
      - H is the Hessian of $C(.) : H_{ij} = \frac{\partial^2 C}{\partial w_i \partial w_j}$
  - Differentiating w.r.t. w
    - $\nabla C(w) = \nabla C\big(w(t)\big) + H(w - w(t))$
  - The minimum of C is obtained for
    - $\nabla C(w) = 0$
  - Several iterative methods could be used
    - E.g. Newton
      - $w(t+1) = w(\mathrm{t}) - \mathrm{H}^{-1} \nabla C(w(t))$
      - Complexity $O(n^3)$ for the inverse + partial derivatives
      - In practice on makes use of quasi-Newton methods : H^{-1} is approximated iteratively

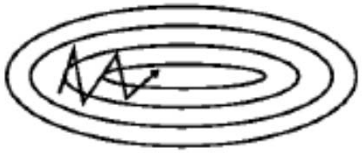
Optimization
Stochastic Gradient algorithms

- Objectives
  - Training NNs involves finding the parameters $\boldsymbol{w}$ 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 redundency 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
    - $D = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^N, \boldsymbol{y}^N)\}$
  - Loss function
    - $C(\boldsymbol{w}) = \sum_{i=1}^{N} c_{\mathrm{w}}(\boldsymbol{x}^i, \boldsymbol{y}^i)$
  - All the algorithms are given in vector form
- Basic Stochastic Gradient Descent
  - Initialise $\boldsymbol{w}(0)$
  - Iterate until stop criterion
  - sample un exemple $(\boldsymbol{x}(t), \boldsymbol{y}(t))$
  - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \epsilon \nabla_{\boldsymbol{w}} c(\boldsymbol{x}(t), \boldsymbol{y}(t))$
  - Rq: might produce a lot of oscillations
- Momentum
  - Dampens oscillations
    - $\boldsymbol{m}(\boldsymbol{t}) = \boldsymbol{\gamma m}(\boldsymbol{t-1}) + \epsilon \nabla_{\boldsymbol{w}} c(\boldsymbol{x}(t), \mathbf{y}(t))$
    - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \boldsymbol{m}(\boldsymbol{t})$

(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 $\boldsymbol{g}(t) = \nabla_{\boldsymbol{w}} c(\boldsymbol{x}(t), \boldsymbol{y}(t))$ — 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:
    - $w_i(t+1) = w_i(t) - \frac{\epsilon}{\sqrt{r_i(t)+\epsilon'}} \nabla_{w_i} c(\boldsymbol{x}(t), \boldsymbol{y}(t))$ — Scalar
  - In vector form
    - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \frac{\epsilon}{\sqrt{\boldsymbol{r}(t)+\epsilon'}} \odot \nabla_{\boldsymbol{w}} c(\boldsymbol{x}(t), \boldsymbol{y}(t))$ — Vector
    - $\odot$ elementwise multiplication, ϵ' $(\approx 10^{-8})$ avoids dividing by 0, $\frac{\epsilon}{\sqrt{\boldsymbol{r}(t)+\epsilon'}}$ 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
    - $\boldsymbol{r}(t) = \gamma r(t-1) + (1-\gamma)\boldsymbol{g}(t) \odot \boldsymbol{g}(t), \quad 0 < \gamma < 1$
    - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \frac{\epsilon}{\sqrt{\boldsymbol{r}(t)+\epsilon'}} \odot \nabla_{\boldsymbol{w}} c(\boldsymbol{x}(t), \boldsymbol{y}(t))$ — Vector

Optimization
SGD algorithm with momentum and Adaptive learning rate

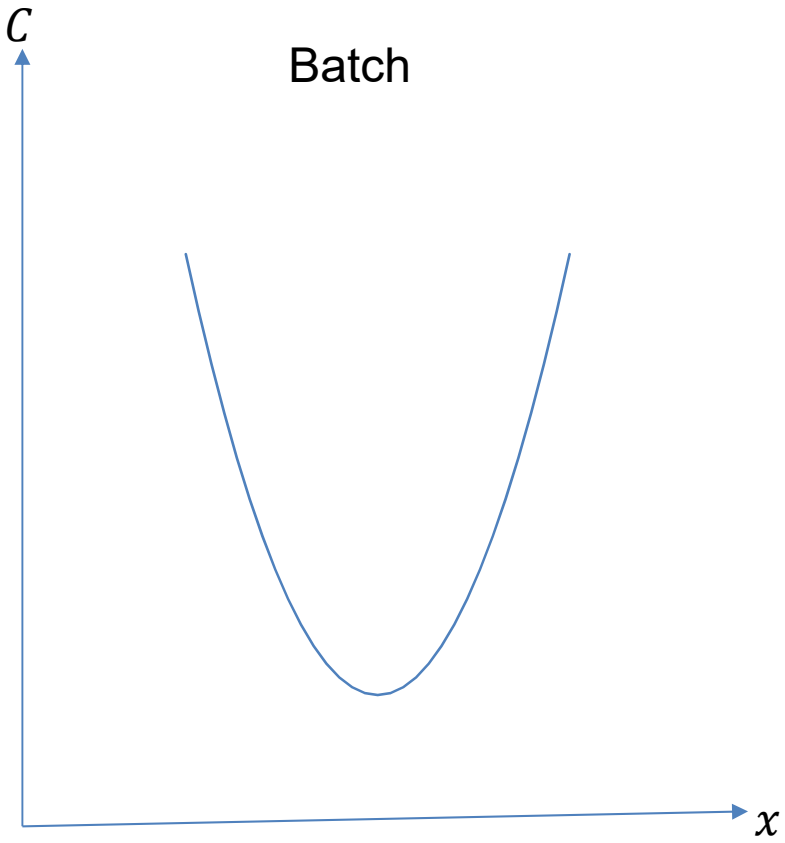
- Adam (adaptive moment estimation)
  - Computes
    - Adaptive learning rates for each parameter
    - An exponentially decaying avarage of past gradients (momentum)
    - An exponentially decaying average of past squared gradients (like RMSprop)
  - Iteration t
    - Momentum term : $\boldsymbol{m}(\boldsymbol{t}) = \gamma_1 \boldsymbol{m}(t-1) + \epsilon(1-\gamma_1)\boldsymbol{g}(t)$
    - Gradient variance term: $\mathrm{r}(t) = \gamma_2 \boldsymbol{r}(t-1) + \epsilon(1-\gamma_2)\boldsymbol{g}(t) \odot \boldsymbol{g}(t)$
    - $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \frac{\epsilon}{\sqrt{\boldsymbol{r}(t)}+\epsilon\prime} \odot \boldsymbol{m}(t)$
    - 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 $\boldsymbol{m}$: $\boldsymbol{m}(\boldsymbol{t}) = \frac{\boldsymbol{m}(\boldsymbol{t})}{1-\gamma_1^t}$
        - Correct bias of $\boldsymbol{r}$: $\mathbf{r}(t) = \frac{\boldsymbol{r}(\boldsymbol{t})}{1-\gamma_2^t}$

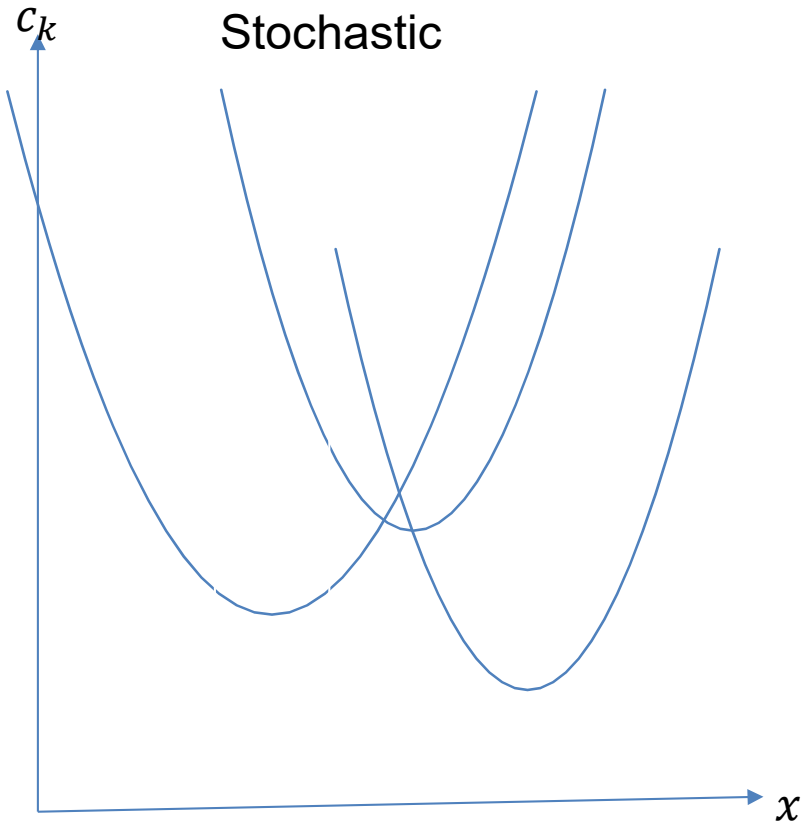
Batch vs stochastic gradient

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

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
  -

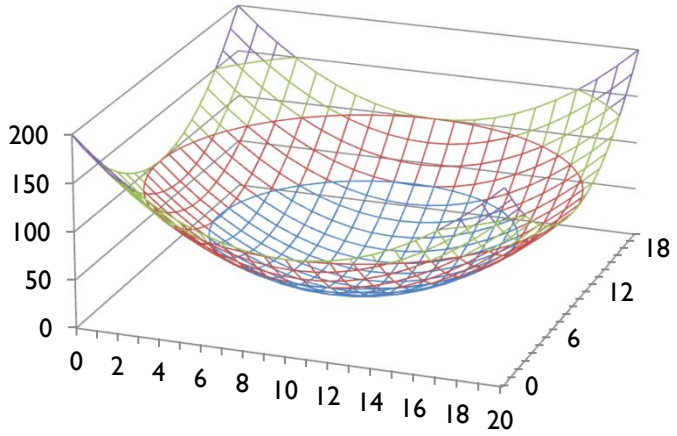
Regression and Logistic Regression

Regression

- Linear regression
  - Objective : predict real values
  - Training set
    - $(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^N, y^N)$
    - $\boldsymbol{x} \in R^n, y \in R$: single output regression
  - Linear model
    - $F(x) = \boldsymbol{w}.\boldsymbol{x} = \sum_{i=0}^{n} w_i x_i$ with $x_0 = 1$
  - Loss function
    - Mean square error
      - $C = \frac{1}{2}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.\boldsymbol{x}^i\right)^2$
  - Steepest descent gradient (batch)
    - $\boldsymbol{w} = \boldsymbol{w}(t) - \epsilon \nabla_w C$, $\nabla_w C = (\frac{\partial C}{\partial w_1}, \dots, \frac{\partial C}{\partial w_n})^T$
    - $\frac{\partial C}{\partial w_k} = \frac{1}{2}\sum_{i=1}^{N}\frac{\partial}{\partial w_k}\left(y^i - \boldsymbol{w}.\boldsymbol{x}^i\right)^2 = -\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.\boldsymbol{x}^i\right)x_{\boldsymbol{k}}^i$ for component w_k
    - $\boldsymbol{w} = \boldsymbol{w}(t) + \epsilon\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.\boldsymbol{x}^i\right)\boldsymbol{x}^i$ in vector form

Regression

- Geometry of mean squares



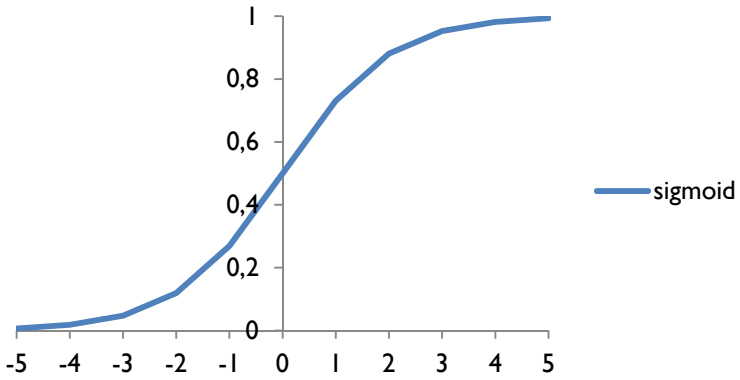
- Regression with multiple outputs $\mathbf{y} \in R^p$
  - Simple extension: p independent linear regressions

Probabilistic Interpretation

- Statistical model of linear regression
  - $y = \boldsymbol{w}.\boldsymbol{x} + \epsilon$, 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
    - $p(y\,|\boldsymbol{x}; \boldsymbol{w}) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y-\boldsymbol{w}.\boldsymbol{x})^2}{2\sigma^2})$
  - Likelihood
    - $L(w) = \prod_{i=1}^{N} p(y^i\,|\boldsymbol{x}^i; \boldsymbol{w})$
      - Likelihood is a function of $\boldsymbol{w}$, it is computed on the training set
  - Maximum likelihood principle
    - Choose the parameters $\boldsymbol{w}$ maximizing $L(\boldsymbol{w})$ or any incresing function of $L(\boldsymbol{w})$
  - In practice, one optimizes the log likelihood $l(\boldsymbol{w}) = logL(\boldsymbol{w})$
    - $l(\boldsymbol{w}) = Nlog\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y^i - \boldsymbol{w}.\boldsymbol{x}^i)^2$
    - 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 classfication
- For classification a proper model is logistic regression
  - $F_w(\boldsymbol{x}) = g(\boldsymbol{w}.\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}.\boldsymbol{x})}$
  - 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 ∈ {0,1}, we make a Bernoulli hypothesis for the posterior distribution
  - $p(y = 1|\boldsymbol{x}; \boldsymbol{w}) = F_w(\boldsymbol{x})$ et $p(y = 0|\boldsymbol{x}; \boldsymbol{w}) = 1 - F_w(\boldsymbol{x})$
  - In compact format
    - $p(y|\boldsymbol{x}; \boldsymbol{w}) = (F_w(\boldsymbol{x}))^y \left(1 - F_w(\boldsymbol{x})\right)^{1-y}$ with y ∈ {0,1}
- Likelihood
  - $L(\boldsymbol{w}) = \prod_{i=1}^{N} \left(F_w(\boldsymbol{x}^i)\right)^{y^i} \left(1 - F_w(\boldsymbol{x}^i)\right)^{1-y^i}$
- Log-likelihood
  - $l(\boldsymbol{w}) = \sum_{i=1}^{N} y^i log F_w(\boldsymbol{x}^i) + (1 - y^i)(\log(1 - F_w(\boldsymbol{x}^i))$
    - This is minus the cross-entropy between the target and the estimated posterior distribution
  - Steepest descent algorithm (batch) for minimizing cross entropy
    - Componentwise: $\frac{\partial l(\boldsymbol{w})}{\partial w_k} = \sum_{i=1}^{N} \left(y^i - F_w(\boldsymbol{x}^i)\right) \boldsymbol{x}_k^i$
    - **Vector** form: $\nabla_w l = \sum_{i=1}^{N} \left(y^i - F_w(\boldsymbol{x}^i)\right) \boldsymbol{x}^i$
    - Algorithm
      - $\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla_w C = \boldsymbol{w} + \epsilon \sum_{i=1}^{N} \left(y^i - F_w(\boldsymbol{x}^i)\right) \boldsymbol{x}^i$

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: $\mathrm{y} = (1,0,\ldots,0)^T$
  - Class 2 : $\mathrm{y} = (0,1,\ldots,0)^T$
  - …
  - Class p: $\mathrm{y} = (0,0,\ldots,1)^T$
- $F_{\boldsymbol{W}}(x)$ is a vector valued function with values in R^p
  - Its component i is a **softmax function** (generalizes the sigmoid)
    - $F_{\boldsymbol{W}}(\boldsymbol{x})_i = \frac{\exp(\boldsymbol{w}_i.\boldsymbol{x})}{\sum_{j=1}^{p}\exp(\boldsymbol{w}_j.\boldsymbol{x})}$
      - Note : here $\boldsymbol{w}_j \in R^n$ is a vector
- The probabilistic model for the posterior is a multinomial distribution
  - $p(Class = i|\boldsymbol{x};\boldsymbol{W}) = \frac{\exp(\boldsymbol{w}_i.\boldsymbol{x})}{\sum_{j=1}^{p}\exp(\boldsymbol{w}_j.\boldsymbol{x})}$ (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

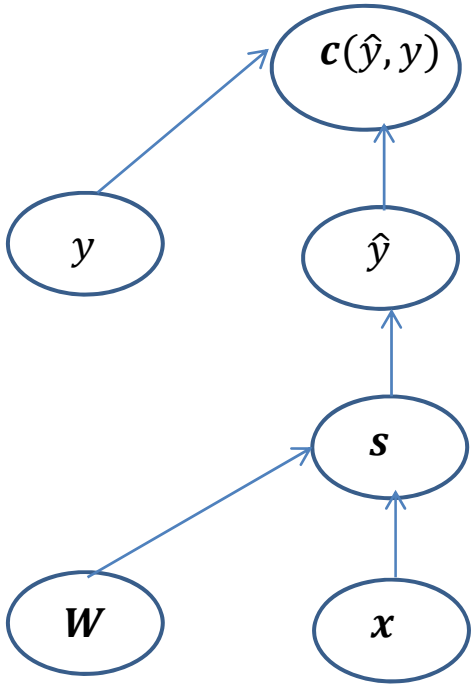
- These results extend to non linear models, e.g. when $F_w(x)$ is a NN
- Non linear regression
  - Max likelihood is equivalent to MSE loss optimization under the Gaussian hyp.
    - $y = F_w(x) + \epsilon, \epsilon \sim N(0, \sigma^2)$
    - $p(y\,|\boldsymbol{x}; \boldsymbol{w}) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y-\boldsymbol{F}(\boldsymbol{x}))^2}{2\sigma^2})$
  - $\log - \text{likelihood } l(w)$
    - $l(\boldsymbol{w}) = N log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y^i - \boldsymbol{F}(\boldsymbol{x}^i)\right)^2$
- 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 $F(x) = p(y = 1|\boldsymbol{x})$ 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

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

y: target



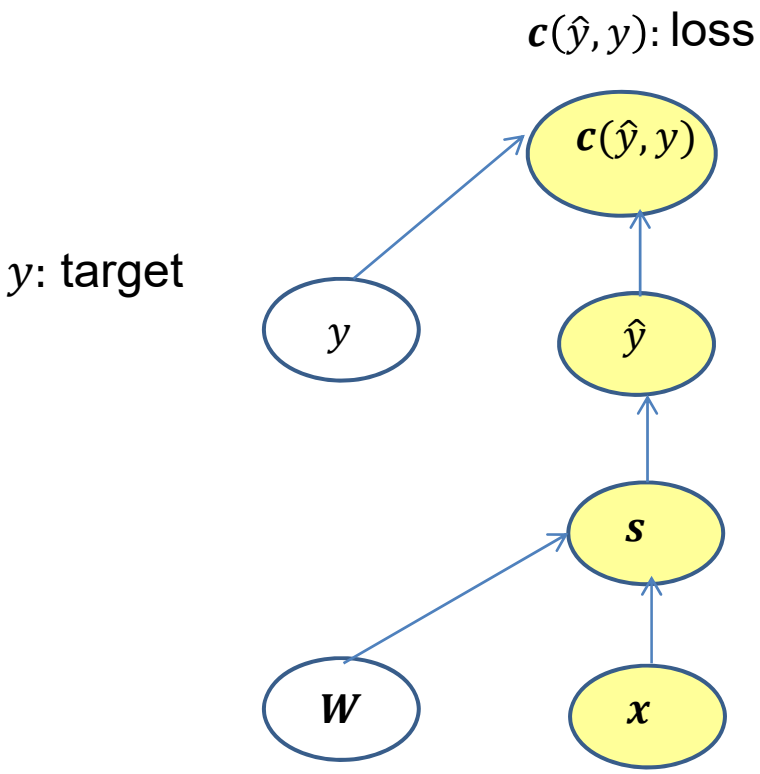
Forward propagation:

$s = \boldsymbol{w}.\boldsymbol{x}$

$\hat{y} = \sigma(s)$

Logistic regression – Computational graph - SGD

- Forward pass



Forward propagation:

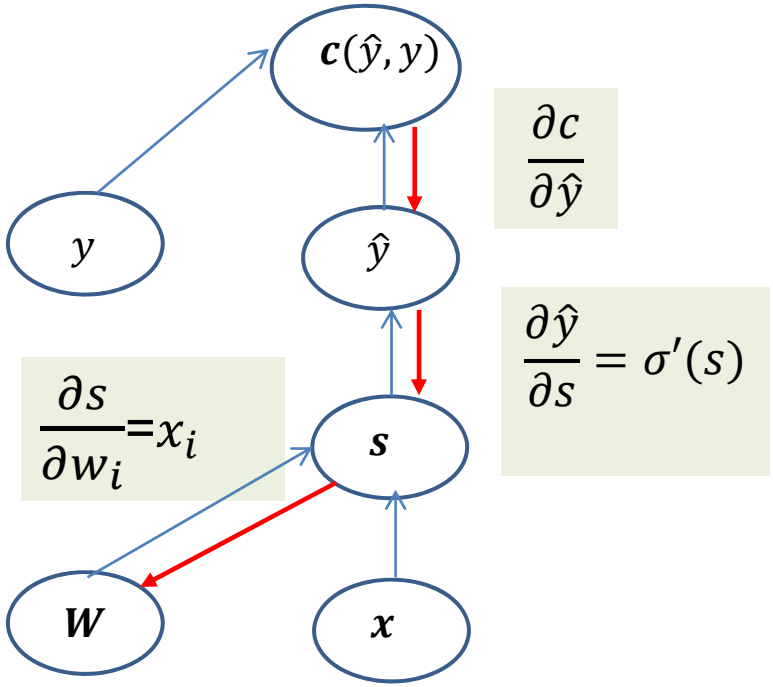
$s = \boldsymbol{w}.\boldsymbol{x}$

$\hat{y} = \sigma(s)$

Logistic regression – Computational graph - SGD

- Backward pass

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



y: target

$\frac{\partial c}{\partial \hat{y}}$

$\frac{\partial \hat{y}}{\partial s} = \sigma'(s)$

$\frac{\partial s}{\partial w_i} = x_i$

Backward propagation:

$$\frac{\partial c}{\partial s} = \frac{\partial c}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial s}$$

$$\frac{\partial c}{\partial w_i} = \frac{\partial c}{\partial s} \frac{\partial s}{\partial w_i}$$

$$\frac{\partial c}{\partial w_i} = \frac{\partial c}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial s} \frac{\partial s}{\partial w_i}$$

Chain Rule

Logistic regression – Computational graph - SGD

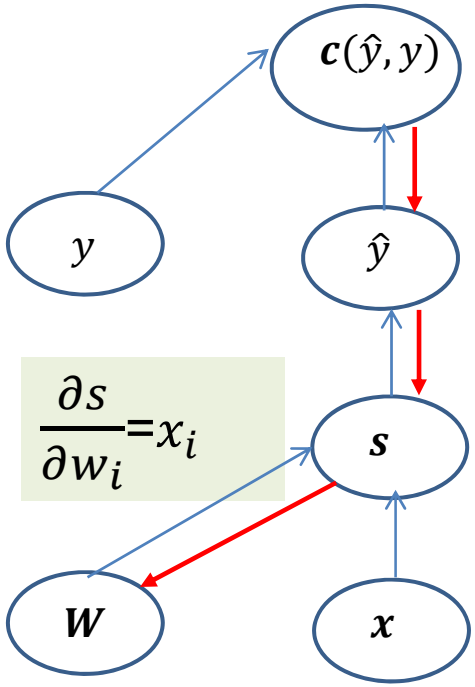
- Backward pass

For the cross entropy loss

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

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

y: target



$$\frac{\partial c}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial \hat{y}}{\partial s} = \sigma'(s)$$

$$\frac{\partial s}{\partial w_i} = x_i$$

Backward propagation:

$$\frac{\partial c}{\partial s} = \frac{\partial c}{\partial \hat{y}}\sigma'(s)$$

$$\frac{\partial c}{\partial w_i} = \frac{\partial c}{\partial s}x_i$$

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

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 $R = E_{x,y}\left[\left(y - h(\boldsymbol{x})\right)^2\right]$
- The minimum of R, $\mathrm{Min}_h R$, is obtained for $h^*(\boldsymbol{x}) = E_y[y|\boldsymbol{x}]$
- The risk R pour the family of functions $F_w(\boldsymbol{x})$ decomposes as follows:
  - $R = E_{x,y}[\left(y - F_w(\boldsymbol{x})\right)^2]$
  - $R = E_{x,y}\left[\left(y - E_y[y|\boldsymbol{x}]\right)^2\right] + E_{x,y}[\left(E_y[y|\boldsymbol{x}] - F_w(\boldsymbol{x})\right)^2]$
- Let us consider $E_y\left[\left(y - E_y[y|\boldsymbol{x}]\right)^2\right]$
  - This term is independent of the model $F_w(.)$ and only depends on the problem characteristics (the data distribution).
  - It represents the min error that could be obtained for this data distribution
  - $h^*(x) = E_y[y|\boldsymbol{x}]$ est the optimal solution to $\mathrm{Min}_h R$
- Minimizing $E_{x,y}[\left(y - F_w(\boldsymbol{x})\right)^2]$ is equivalent to minimizing $E_{x,y}\left[\left(E_y[y|\boldsymbol{x}] - F_w(\boldsymbol{x})\right)^2\right]$
  - The optimal solution $F_{w*}(\boldsymbol{x}) = \mathrm{argmin}_{\mathrm{w}} E_{x,y}\left[\left(E_y[y|\boldsymbol{x}] - F_w(\boldsymbol{x})\right)^2\right]$ is the best mean square approximation of $E[y|\boldsymbol{x}]$

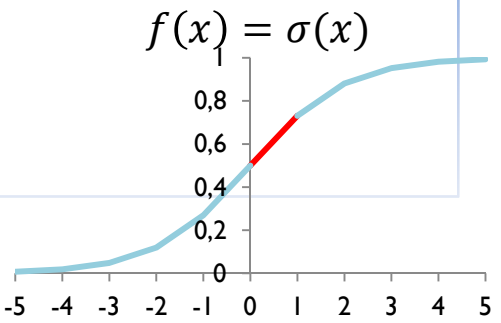
Probabilistic interpretation of NN outputs

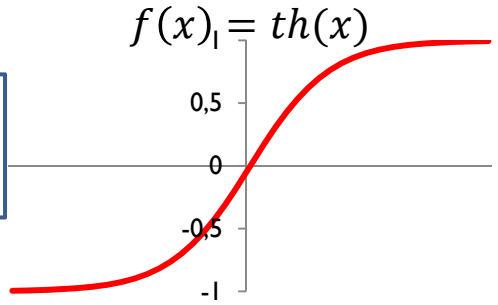
- Classification
  - Let us consider multi-class classification with one hot encoding of the target outputs
    - i.e. $\boldsymbol{y} = (0, \dots, 0, 1, 0, \dots, 0)^T$ with a 1 at position i if the target is class i and zero everywhere else
    - $h_i^* = E_y[y|x] = 1 * P(C_i|x) + 0 * \big(1 - P(C_i|x)\big) = P(C_i|x)$
    - 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)

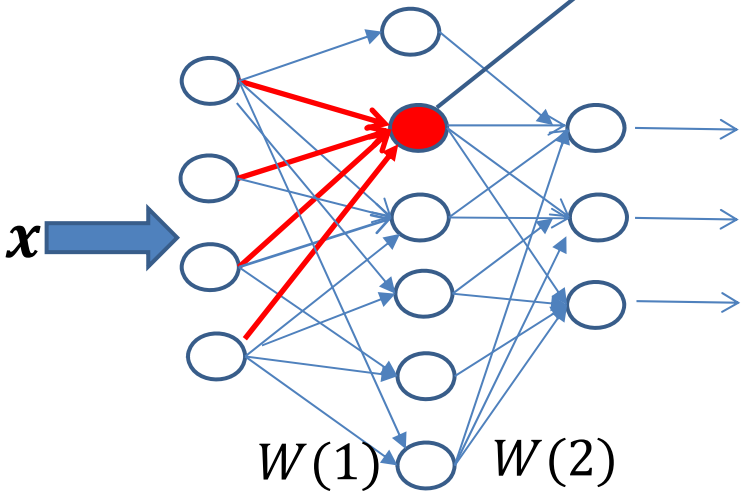
- Neurons arranged into layers
- Each neuron is a non linear unit, e.g.





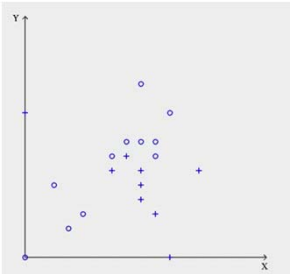
$f(\boldsymbol{w}.\boldsymbol{x})$

$\boldsymbol{w}$: **cell weight vector**



$$\hat{\boldsymbol{y}} = F_w(\boldsymbol{x}) = f\odot(W(2)f\odot(W(1)\boldsymbol{x}))$$

http://playground.tensorflow.org/



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

Multi-layer Perceptron - Training

- **Stochastic Gradient Descent** - The algorithm is called **Back-Propagation**
  - Pick one example $(\boldsymbol{x}, \boldsymbol{y})$ or a **Mini Batch** $\{(\boldsymbol{x^i}, \boldsymbol{y^i})\}$ sampled from the training set
    - Here the algorithm is described for 1 example and for the sigmoid $\sigma()$ non linearity
  - Forward pass
    - $\hat{\boldsymbol{y}} = F_w(\boldsymbol{x}) = f\odot(W(2)f\odot(W(1)\boldsymbol{x}))$
  - Compute error
    - $c(\boldsymbol{y}, \hat{\boldsymbol{y}})$, e.g. mean square error or cross entropy
  - Backward pass
    - efficient implementation of chain rule
    - $w_{ij} = w_{ij} - \epsilon \frac{\partial c(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}}$

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

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 funtion into the program for computing the derivatives
  - All modern DL framework implement AD

Notations – matrix derivatives

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

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

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

Vector by vector

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

Matrix by scalar

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

Scalar by matrix

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

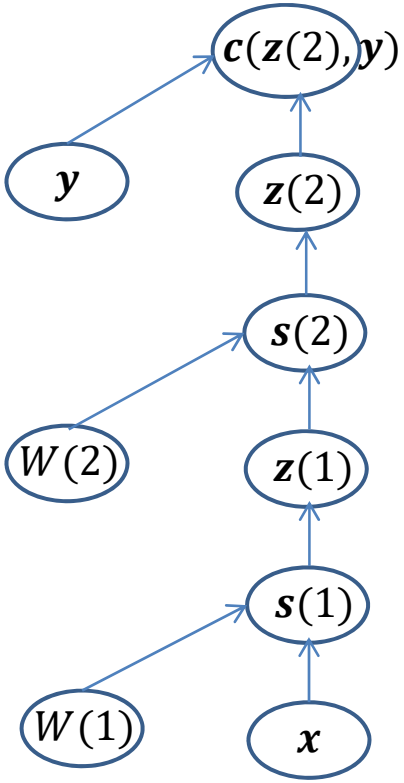
Multi-layer Perceptron - Training

- Computational graph

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

Here, $\boldsymbol{z}(2) = \hat{\boldsymbol{y}}$

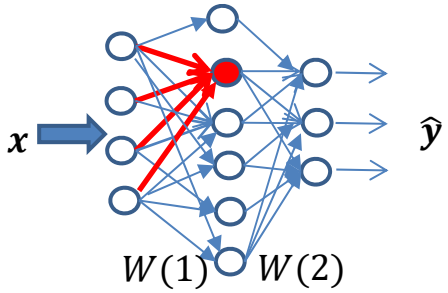
$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

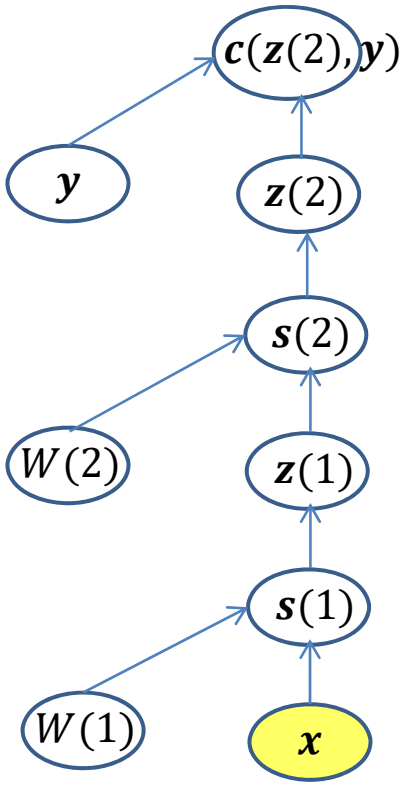


Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

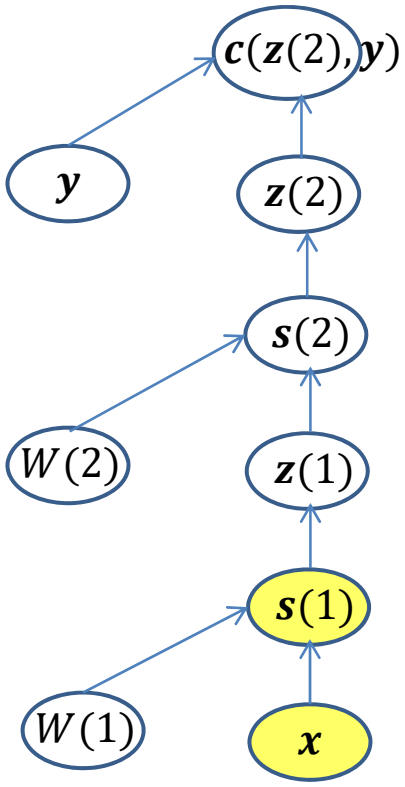
$$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$$
$$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$$

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

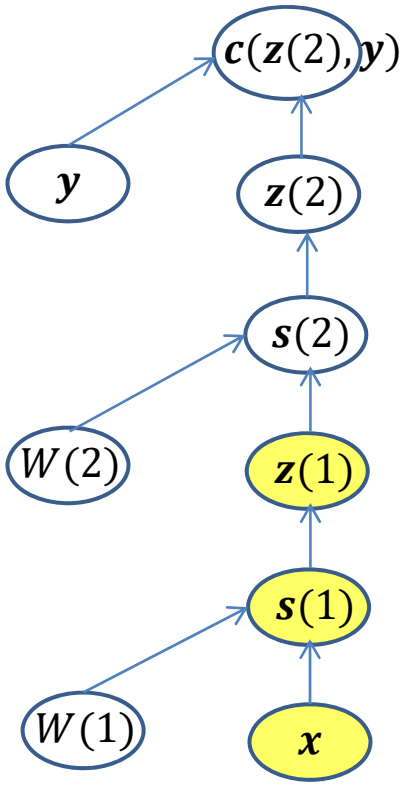
$$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$$
$$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$$

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

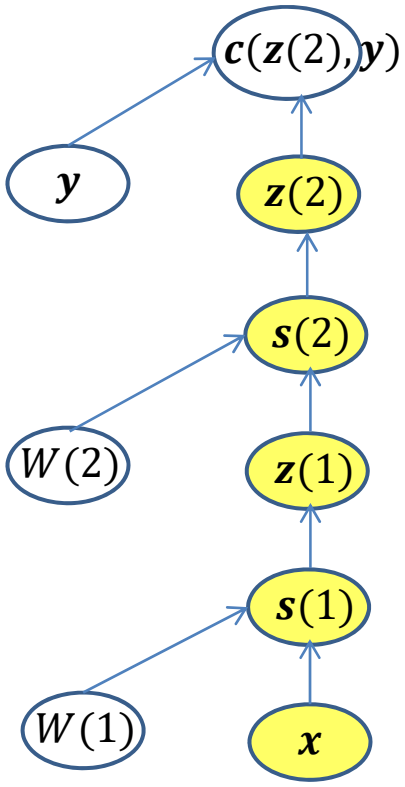
$$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$$
$$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$$

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

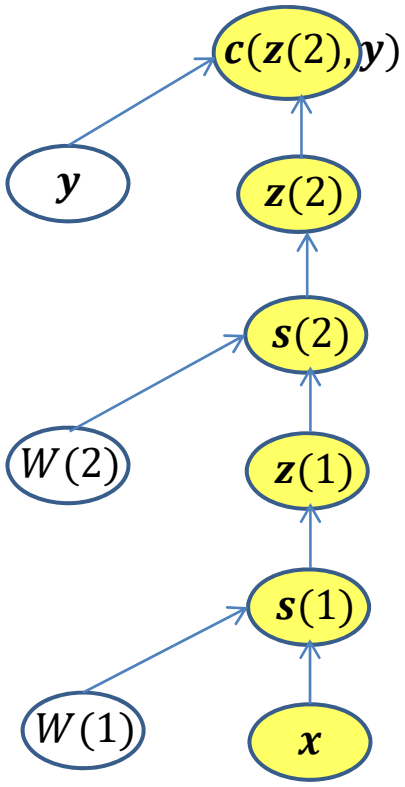
Note: $\boldsymbol{x}$, $\boldsymbol{s}(n)$, $\boldsymbol{x}(n)$ are vectors

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

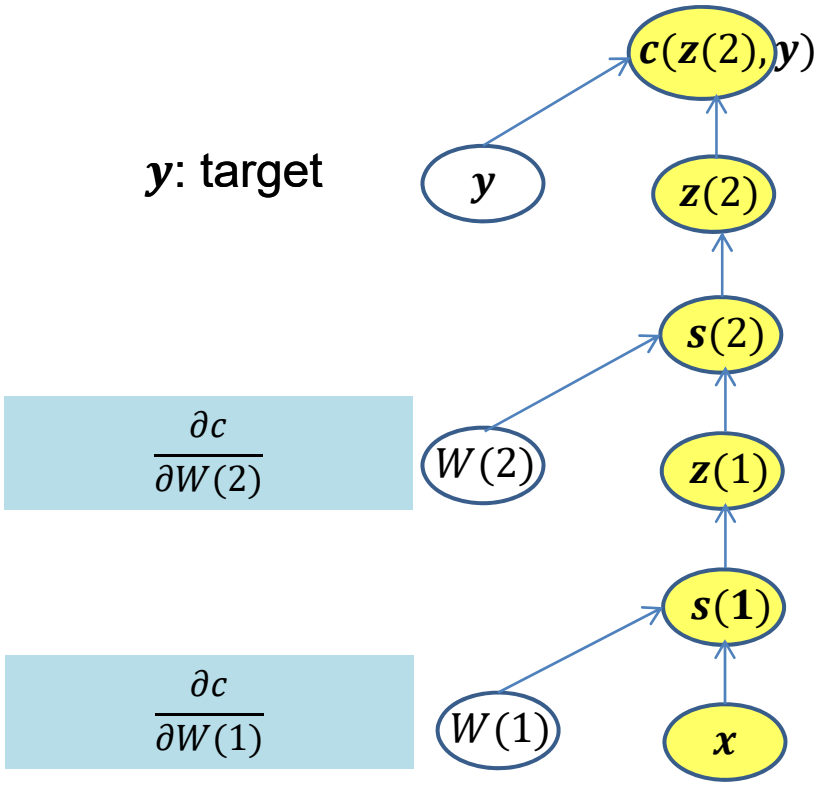
$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

Note: $\boldsymbol{x}$, $\boldsymbol{s}(n)$, $\boldsymbol{x}(n)$ are vectors

Multi-layer Perceptron - Training

- Back Propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



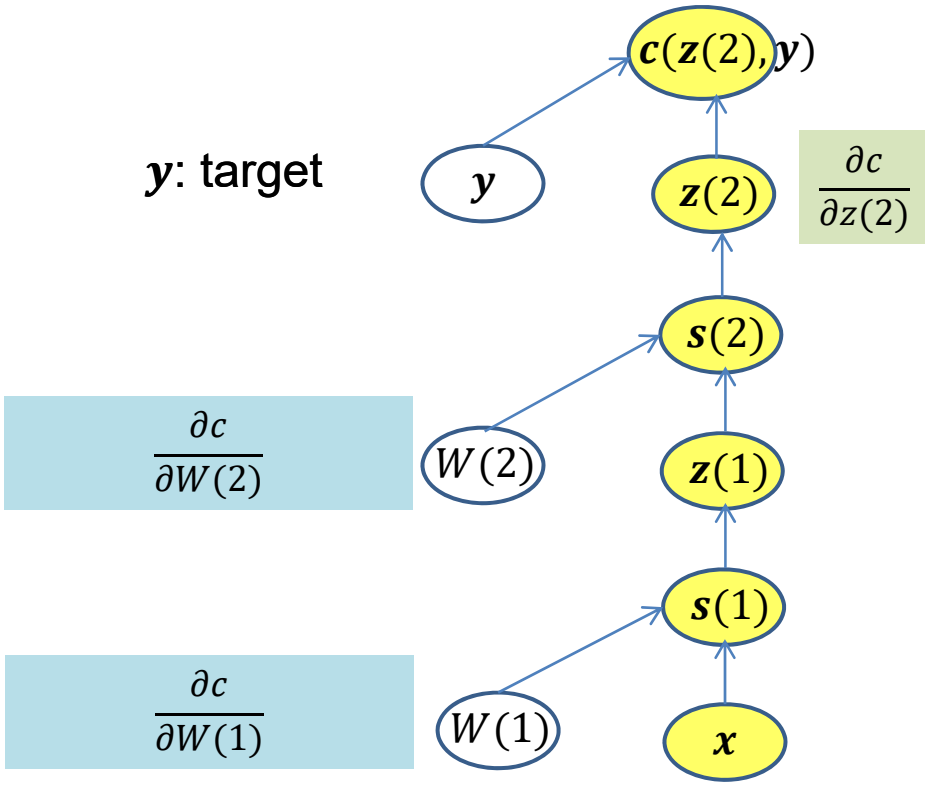
$$W = W - \epsilon \frac{\partial c}{\partial W}$$

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

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1)\frac{\partial c}{\partial \boldsymbol{s}(n)}$$

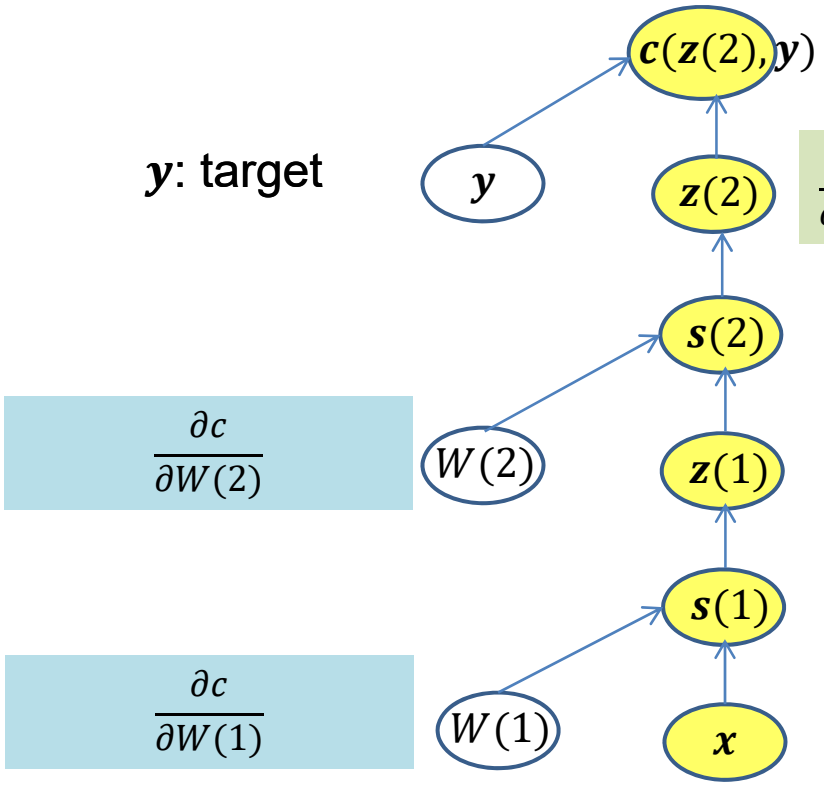
$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

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

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1) \frac{\partial c}{\partial \boldsymbol{s}(n)}$$

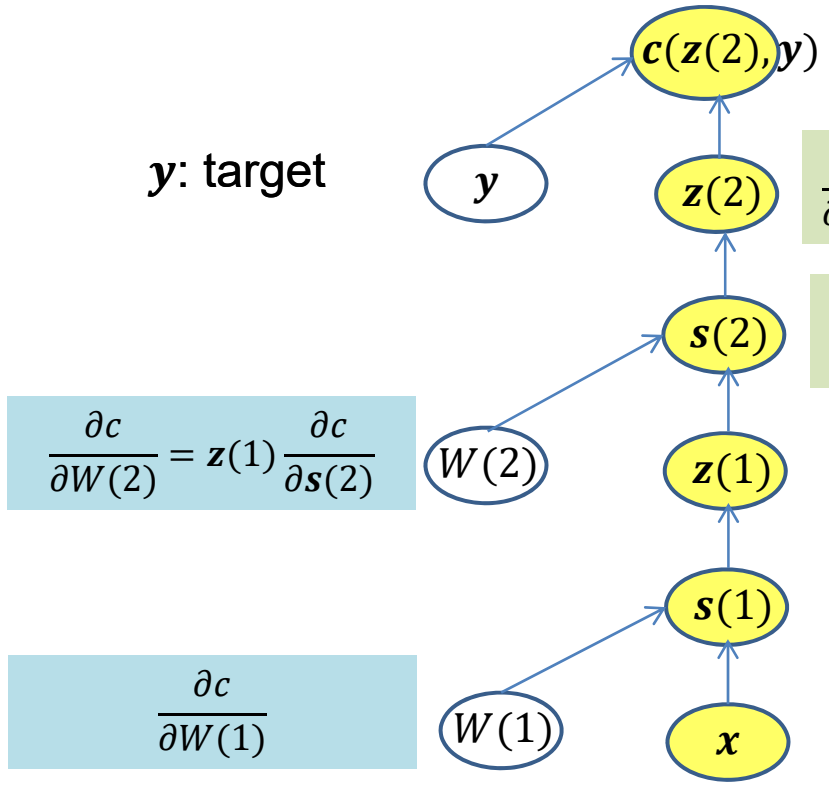
$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

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

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1)\frac{\partial c}{\partial \boldsymbol{s}(n)}$$

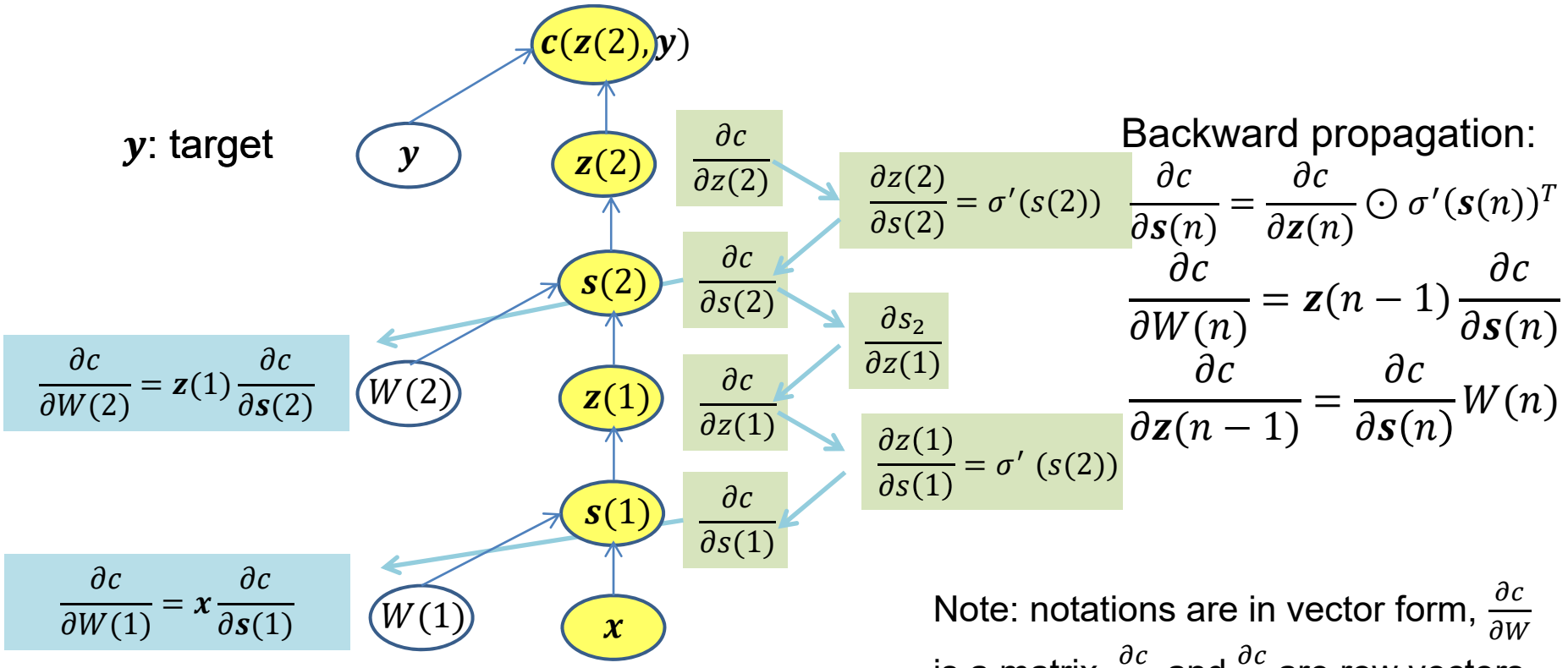
$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

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

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

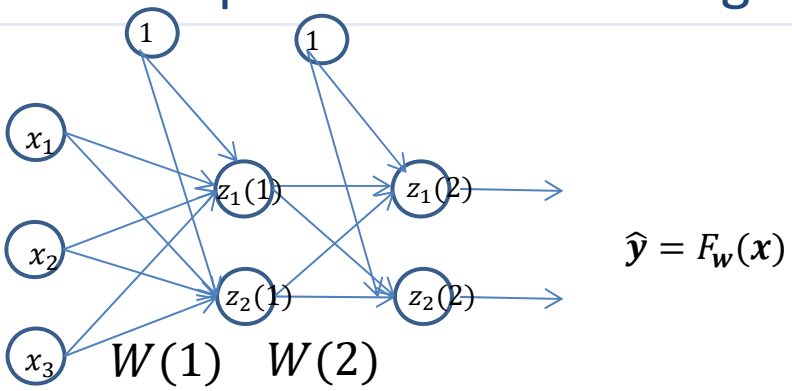
$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1)\frac{\partial c}{\partial \boldsymbol{s}(n)}$$

$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

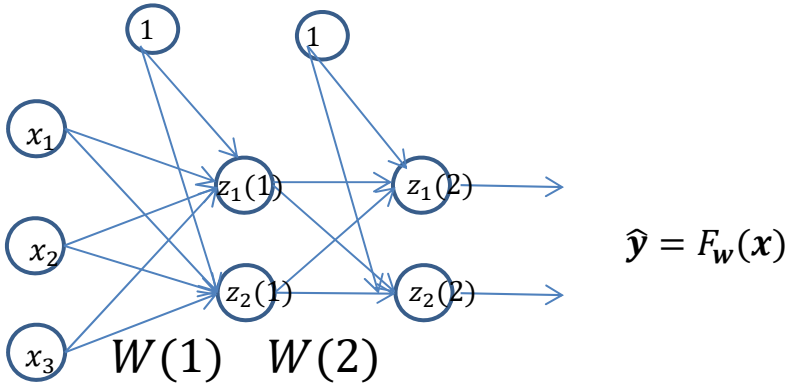
Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are row vectors of the appropriate size

Multi-layer Perceptron – SGD Training – example - notations



- Notations
  - $\boldsymbol{z}(i)$ 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 $i+1$
  - $\hat{\boldsymbol{y}}$ 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 – Detailed derivation for a 1 hidden layer network (MSE loss + sigmoid units) - forward pass

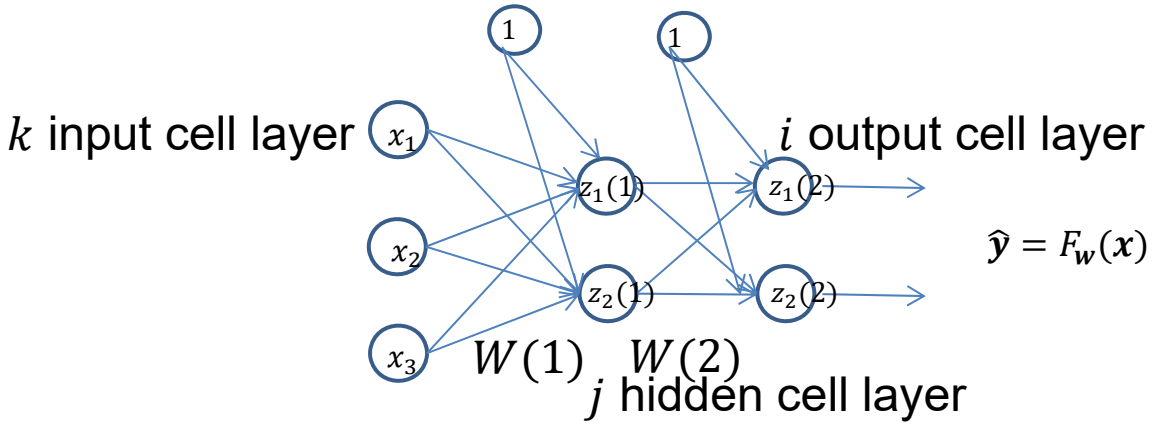


$\hat{\boldsymbol{y}} = F_{\boldsymbol{w}}(\boldsymbol{x})$

- For example x
  - The activations of all the neurons from layer 1 are computed in parallel
  - $\boldsymbol{s}(1) = W(1)\boldsymbol{x}$ then $\boldsymbol{z}(1) = g(\boldsymbol{s}(1))$
    - with $g(\boldsymbol{s}(1)) = (g(\boldsymbol{s}_1(1)), g(\boldsymbol{s}_2(1)))^{\mathrm{T}}$
  - 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.
  - $\boldsymbol{s}(2) = W(2)\boldsymbol{z}(1)$ then $\hat{\boldsymbol{y}} = \boldsymbol{z}(2) = g(\boldsymbol{s}(2))$
    -

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



  - $s_j(1) = \sum_k w_{jk}(1)x_k$, $z_j(1) = g(s(1))$
  - $s_i(2) = \sum_j w_{ij}(2)z_j(1)$, $z_i(2) = g(s_i(2))$
    - $s_i(2) = \sum_j w_{ij}(2)g(\sum_k w_{jk}(1)x_k)$, $z_i(2) = g(\sum_j w_{ij}(2)g(\sum_k w_{jk}(1)x_k))$
- 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
  - Upgrade rule for weight w_{ij}, layer m: $w_{ij}(m) = w_{ij}(m) + \Delta w_{ij}(m)$
  - 2nd weight layer
    - $\Delta w_{ij}(2) = -\epsilon \frac{\partial C}{\partial w_{ij}(2)} = -\epsilon \frac{\partial C}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{w_{ij}(2)}$
    - $\Delta w_{ij}(2) = \epsilon (y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial s_i(2)} \frac{\partial s_i(2)}{\partial w_{ij}(2)}$
    - $\Delta w_{ij}(2) = \epsilon (y_i - \hat{y}_i) g'(s_i(2)) z_j(1)$
    - $\Delta w_{ij}(2) = \epsilon e_i(2) z_j(1)$, with $e_i(2) = (y_i - \hat{y}_i) g'(s_i(2))$
  - 1st weight layer
    - $\Delta w_{ij}(1) = -\epsilon \frac{\partial C}{\partial w_{ij}(1)} = -\epsilon \frac{\partial C}{\partial z_j(1)} \frac{\partial z_j(1)}{\partial w_{ij}(1)}$
      - $\frac{\partial C}{\partial z_j(1)} = \sum_{i\ parents\ of\ j} \frac{\partial C}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_j(1)} = -\sum_i (y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial s_i(2)} \frac{\partial s_i(2)}{\partial z_j(1)}$
      - $\frac{\partial C}{\partial z_j(1)} = -\sum_i (y_i - \hat{y}_i) g'(s_i(2)) w_{ij}(2)$

Multi-layer Perceptron – SGD derivation
Detailed derivation (MSE loss + sigmoid units)

- $\frac{\partial z_j(1)}{\partial w_{jk}(1)} = \frac{\partial z_j(1)}{\partial s_j(1)} \frac{\partial s_j(1)}{\partial w_{jk}(1)} = g'\big(s_j(1)\big) z_k$

- $\Delta w_{jk}(1) = \epsilon \sum_{i\ parents\ of\ j} (y_i - \hat{y}_i) g'(s_i(2)) w_{ij}(2)\ g'\big(s_j(1)\big) x_k$
- $\Delta w_{jk}(1) = \epsilon e_j(1) x_k$ with $e_j = g'\big(s_j(1)\big) \sum_{i\ parents\ of\ j} e_i w_{ij}(2)$

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
  - $Min_W c = \frac{1}{N}\sum_{k=1}^{N} c(F(x^k), y^k)$
  - With $F(x) = F_l \circ \cdots \circ F_1(x)$
- Rewritten as a constrained optimisation problem
  - $Min_W c = \frac{1}{N}\sum_{k=1}^{N} c(z^k(l), y^k)$
  - Subject to
$$\begin{cases} z^k(l) = F_l(z^k(l-1), W(l)) \\ z^k(l-1) = F_{l-1}(z^k(l-2), W(l-1)) \\ \dots \\ z^k(1) = F_1(x^k, W(1)) \end{cases}$$
  - 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 pure SGD, i.e. $N = 1$
  - So that we drop the index k
- The Lagrangian associated to the optimization problem is
  - $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^{l} \lambda_i^T (z(i) - F_i(z(i-1), W(i))$
  - Unknowns to be estimated:
    - $z(i), W(i), \lambda_i, i = 1 \dots l,$

Back Propagation and Adjoint

- We want to solve for the Lagrangian
  - $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^{l} \lambda_i^T (z(i) - F_i(z(i-1), W(i))$
  - with unknowns: $z(i), W(i), \lambda_i, i = 1, \dots, l$
- The partial derivatives of the Lagrangian are
  - $\frac{\partial \mathcal{L}}{\partial z(l)} = -\lambda_l^T + \frac{\partial c(z(l),y)}{\partial z(l)}$ for the last layer l
  - $\frac{\partial \mathcal{L}}{\partial z(i)} = -\lambda_i^T + \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i),W(i+1))}{\partial z(i)}, \; i = 1, \dots, l-1$ for intermediate layer i
  - $\frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_i^T \frac{\partial F_i(z(i-1),W(i))}{\partial W(i)}, \; \mathrm{i} = 1 \dots \mathrm{l}$
  - $\frac{\partial \mathcal{L}}{\partial \lambda_i} = z(i) - F_i(z(i-1), W(i)) \; , \; \mathrm{i} = 1 \dots \mathrm{l}$
  - 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 ... l, represent the constraints
  - One wants $\frac{\partial \mathcal{L}}{\partial \lambda_i} = 0, \quad i = 1 \dots l$
  - 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 $(z(i) - F_i\big(z(i-1), W(i)\big) = 0$ (forward pass), one can choose λ_i^T as we want
  - Let us choose the λs such that $\frac{\partial \mathcal{L}}{\partial z(i)} = 0, \forall\, i$
  - The λs can be computed backward Starting at $i = l$ down to 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
    - $\frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_{i+1}^T \frac{\partial F_i(z(i-1),W(i))}{\partial W(i)}$, $\forall\, i$

      - $\frac{\partial F_i(z(i-1),W(i))}{\partial W(i)} = \frac{\partial z(i)}{\partial W(i)}$ 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 $z(i) - F_{l-1}\big(z(i-1), W(i)\big) = 0, i = 1 \dots l$

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
  - $\frac{\partial c}{\partial z} - \lambda^T \frac{\partial g}{\partial z} = 0$, rewriten as:

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

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
      - $\hat{\boldsymbol{y}} \in R^p, \boldsymbol{y} \in \{-1,1\}^p$
  - Hinge, logistic losses are used as proxies for the classification loss

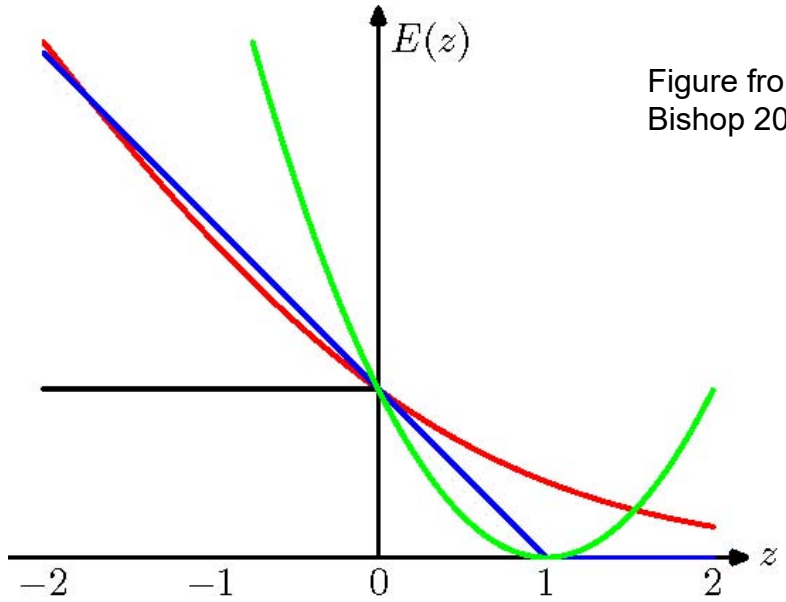


Figure from Bishop 2006

z coordinate: $z = \hat{\boldsymbol{y}}.\boldsymbol{y}$ (margin)

$$C_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2$$
$$C_{hinge}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = [1 - \hat{\boldsymbol{y}}.\boldsymbol{y}]_+ = \max(0, 1 - \hat{\boldsymbol{y}}.\boldsymbol{y})$$
$$C_{logistic}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \ln(1 + \exp(-\hat{\boldsymbol{y}}.\boldsymbol{y}))$$

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 $g(x) = \sum_{j=1}^{n} v_j f(\mathbf{w}_j . . \mathbf{x})$ is dense in the space of continuous functions on the unit cube $C(I)$. i.e. $\forall h \in C(I) et \; \forall \epsilon > 0, \exists \; g : |g(x) - h(x)| < \epsilon$ 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 $g(x) = \sum_{j=1}^{n} v_j f(\mathbf{w}_j . . \mathbf{x})$ and a set $D \subset I$ such that $measure(D) = 1 - \epsilon(D)$ 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 $g(x) = \sum_{j=1}^{n} v_j f(\mathbf{w}_j . . \mathbf{x})$ is uniformely dense on the compact sets $C(R^n)$ - the space of continuous functions on R^n
  - Notes:
    - None of these result is constructive
    - Recent review of approximation properties of NN: Guhring et al., 2020, Expressivity of deep neural networks, arXiv:2007.04759

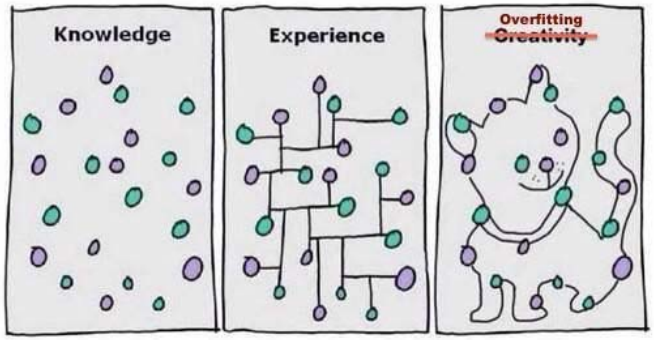
Complexity control

Bias – Variance

Overtraining and regularization

Generalization and Model Selection

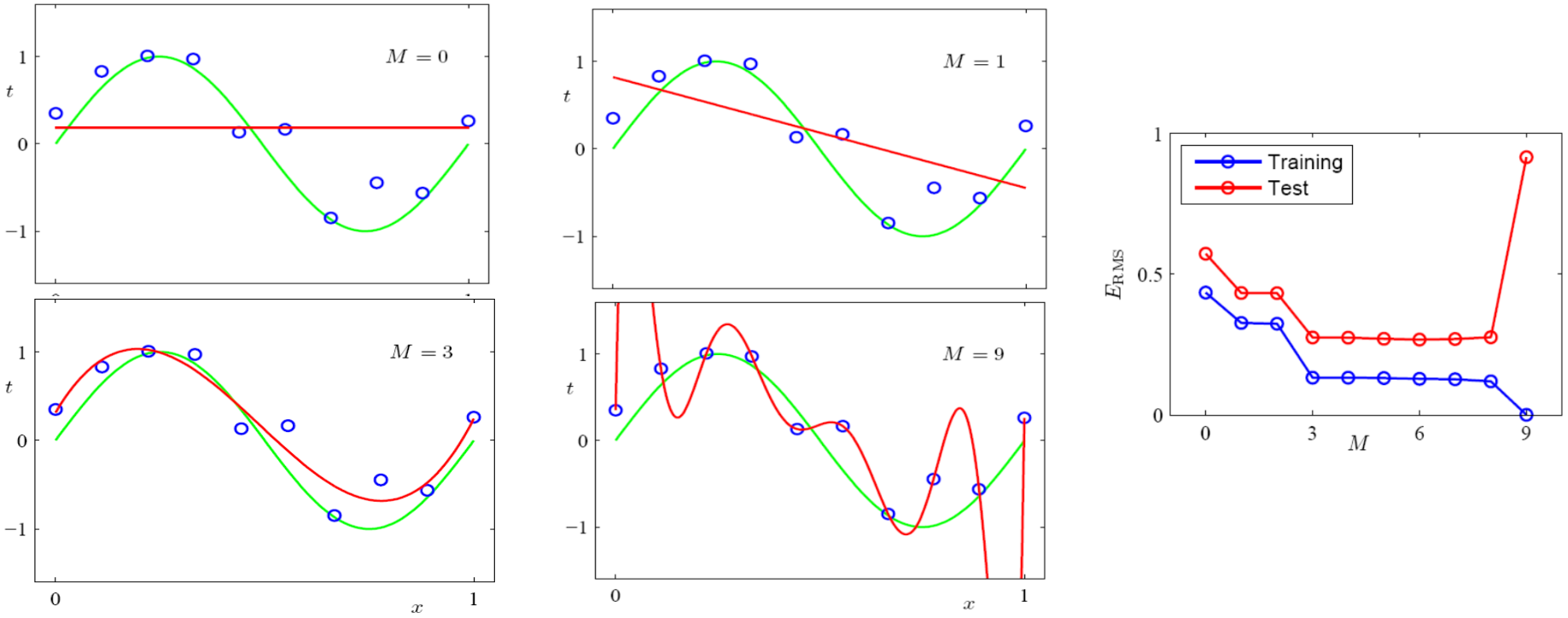
- Complex models sometimes perform worse than simple linear models
  - Overfitting/ generalization problem



- 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



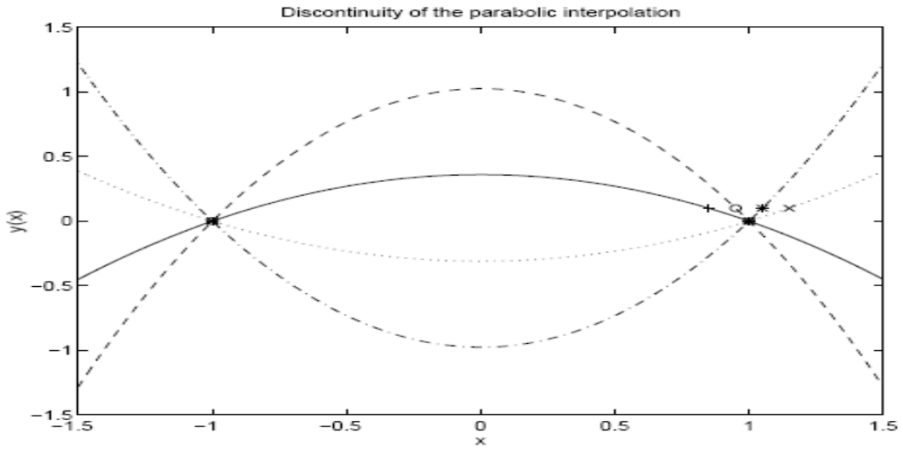
- Model complexity shall be controlled (learned) during training
  - How?

Complexity control

- One shall optimize the risk while controling 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

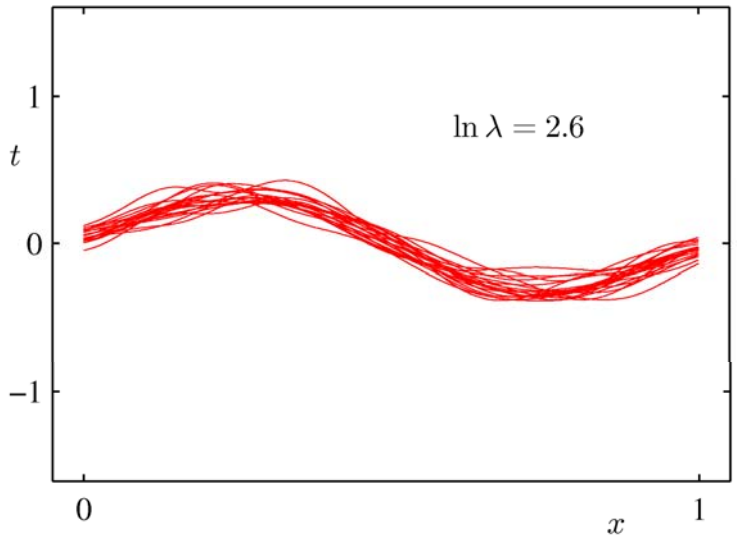
- Illustrates the problem of model selection, puts in evidence the influence of the complexity of the model
  - Remember: MSE risk decomposition
    - $E_{x,y}\left[\left(y - F_w(\boldsymbol{x})\right)^2\right] = E_{x,y}\left[\left(y - E_y[y|\boldsymbol{x}]\right)^2\right] + E_{x,y}\left[\left(E_y[y|\boldsymbol{x}] - F_w(\boldsymbol{x})\right)^2\right]$
    - Let $h^*(x) = E_y[y|\boldsymbol{x}]$ be the optimal solution for the minimization of this risk
  - In practice, the number of training data for estimating $E_y[y|\boldsymbol{x}]$ 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: $D_1, D_2, \ldots$
      - Learn $F_w(\boldsymbol{x}, D)$ for each of these datasets
      - Compute the mean of the empirical errors obtained on these different datasets

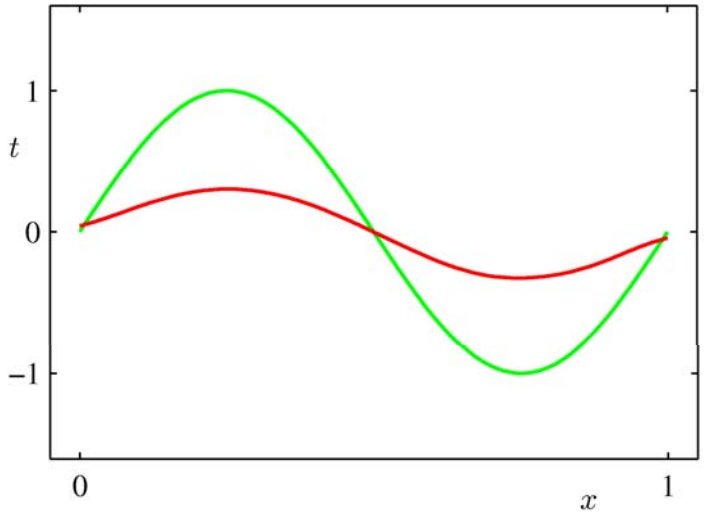
Bias-variance decomposition

- Let us consider the quadratic error $\left(F(x;D) - h^*(x)\right)^2$ 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 $E_{D\sim p(D)}[F_w(x;D)]$ denote the expectation w.r.t. the distribution of $D, p(D)$
- $(F_w(x;D) - h^*(x))^2$ decomposes as:
  - $(F_w(x;D) - h^*(x))^2 = (F_w(x;D) - E_D[F_w(x;D)] + E_D[F_w(x;D)] - h^*(x))^2$
  - $(F_w(x;D) - h^*(x))^2 = \begin{array}{l}(F_w(x;D) - E_D[F_w(x;D)])^2 + (E_D[F_w(x;D)] - h^*(x))^2 \\ + 2(F_w(x;D) - E_D[F_w(x;D)])(E_D[F_w(x;D)] - h^*(x))\end{array}$
- Expectation w.r.t. D distribution decomposes as:
  - $E_D[(F_w(x;D) - h^*(x))^2] = (E_D[F_w(x;D)] - h^*(x))^2 + E_D[(F_w(x;D) - E_D[F_w(x;D)])^2]$
  - $= \quad bias^2 \quad + \quad variance$
- 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)

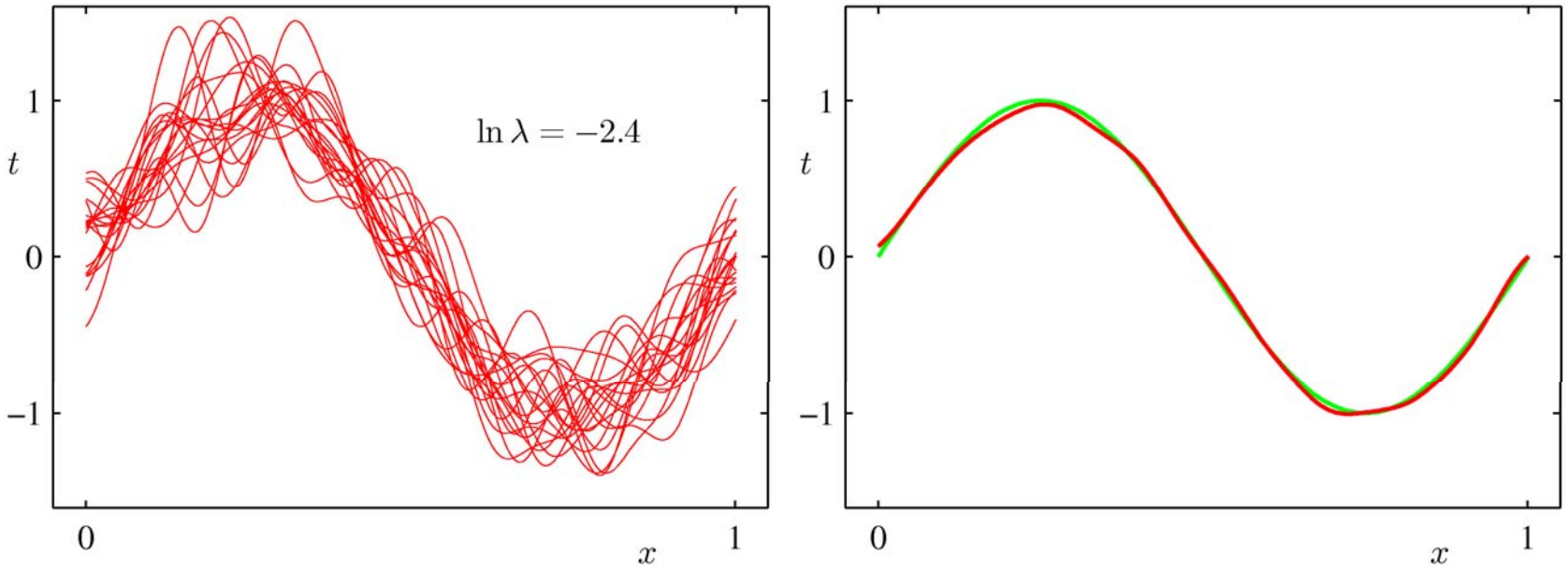
- Example: 100 data sets from the sinusoidal, varying the degree of regularization
  - Model: gaussian basis function, Learning set size = 25, λ is the regularization parameter
    - High values of λ correspond to simple models, low values to more complex models
  - Left 20 of the 100 models shown
  - 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)

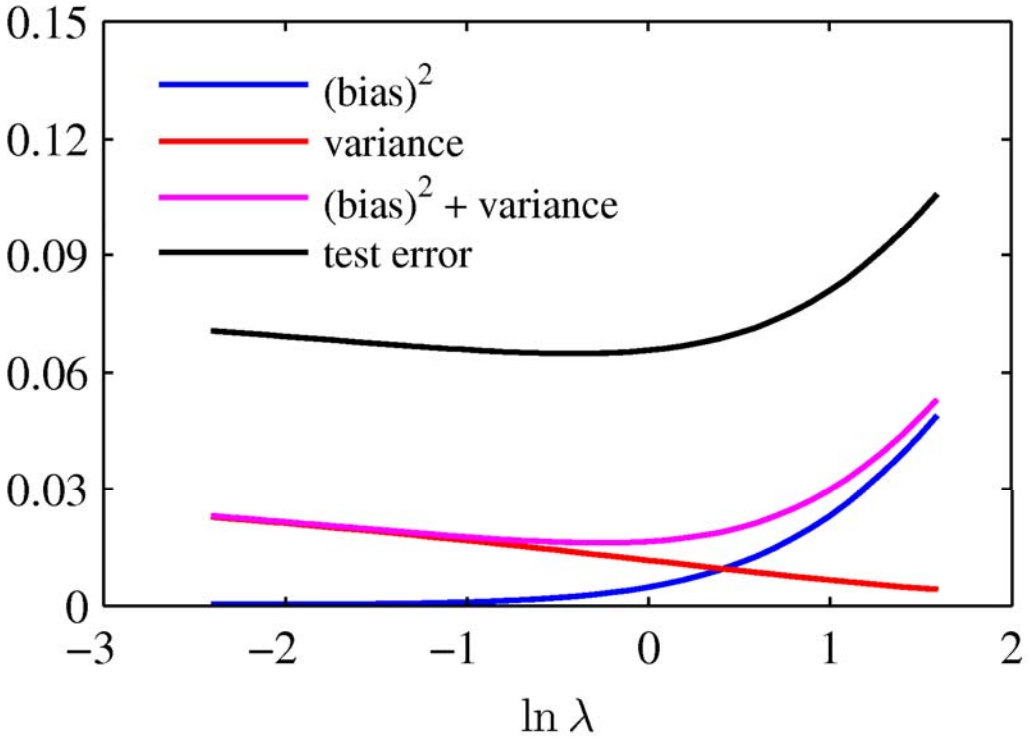
- Example: 100 data sets from the sinusoidal, varying the degree of regularization
  - Same setting as before
    - Figure illustrates low bias and high variance ($\lambda = 0.09$)



  - Remark
    - The mean of several complex models behaves well here (reduced variance)
      - $\rightarrow$ 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
  - $C = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.x^i\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\left|w_j\right|^q$
    - $q = 2$ regularization L_2, $q = 1$ regularization L_1 also known as « Lasso »

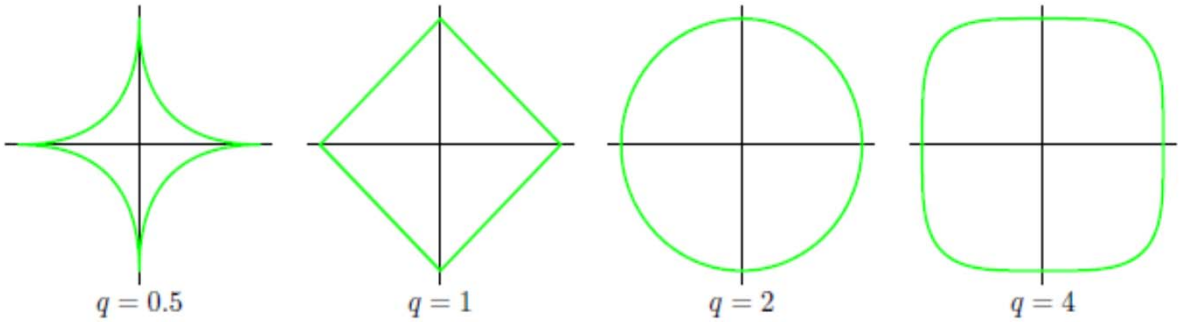


Fig. from Bishop 2006

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

Régularisation

- Solve
  - $Min_{\boldsymbol{w}}\, C = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.x^i\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\left|w_j\right|^q, \lambda > 0$
- Amounts at solving the following constrained optimization problem
  - $Min_{\boldsymbol{w}}\, C = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.x^i\right)^2$
  - Under constraint $\sum_{j=1}^{n}\left|w_j\right|^q \leq s$ 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 $w_1^* = 0$.

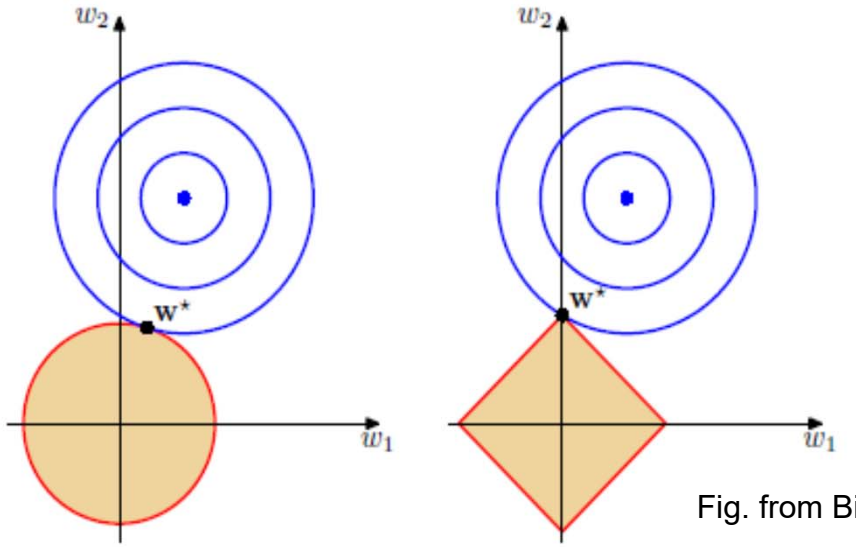


Fig. from Bishop 2006

Regularization

- Penalization L_2
  - Loss
    - $C = C_1 + \lambda \sum_{j=1}^{n} |w_j|^2$
  - Gradiant
    - $\nabla_{\boldsymbol{w}} C = \lambda \boldsymbol{w} + \nabla_{\boldsymbol{w}} C_1$
  - Update
    - $\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} C = (1 - \epsilon\lambda)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} C_1$
    - Penalization is proportional to $\boldsymbol{w}$
- Penalization L_1
  - Loss
    - $C = C_1 + \lambda \sum_{j=1}^{n} |w_j|^1$
  - Gradiant
    - $\nabla_{\boldsymbol{w}} C = \lambda sign(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} C_1$
    - $sign(\boldsymbol{w})$ is the sign of $\boldsymbol{w}$ applied to each component of $\boldsymbol{w}$
  - Update
    - $\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} C = \boldsymbol{w} - \epsilon\lambda sign(\boldsymbol{w}) - \epsilon \nabla_{\boldsymbol{w}} C_1$
    - Penalization is constant with sign $sign(\boldsymbol{w})$

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

Generalization in modern Deep Learning

- Deep Learning models often do not follow the common complexity / performance wisdom
  - Extremely large models / with no complexity control (like e.g. regularization or early stopping), may reach good performance, better than models trained with the usual complexity control ingredients
  - Observed in modern deep learning
    - High complexity models with zero train error may not overfit and lead to accurate predictions on unseen data
      - This observation questions the usual claim and the theoretical beliefs such as Bias – Variance dilemma
- Example
  - Double descent phenomenon
    - Based on (Belkin 2019) and (Nakkiran 2020)

Generalization in modern Deep Learning - Double Descent

- Observed by different authors but formalized as a general concept in (Belkin 2019)
- General message
  - Learning curves as a function of model capacity (complexity) exhibit a two regimes phenomenon coined as « double descent »
  - Classical regime corresponds to under-parameterized models and exhibits the classical U shaped curve corresponding to the bias-variance intruition
    - Models do not achieve perfect interpolation
    - The test risk first decreases and then increases when the model starts interpolating
  - Modern interpolation regime corresponds to over-parameterized models
    - Models may achieve near zero train error, i.e. near perfect interpolation
    - Test risk value may decrease below the level of the best classical regime risk value

Generalization in modern Deep Learning - Double Descent Intuition (Belkin 2019)

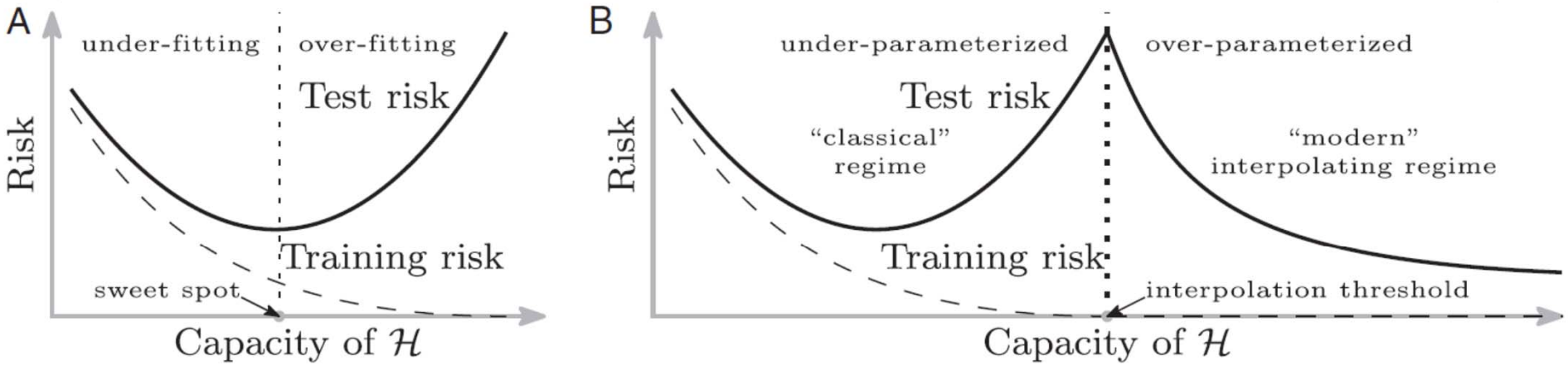


Fig. 1. Curves for training risk (dashed line) and test risk (solid line). (*A*) The classical U-shaped risk curve arising from the bias–variance trade-off. (*B*) The double-descent risk curve, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high-capacity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

- All the models to the right of the interpolation threshold have a zero training error
- Tentative explanation
  - The notion of « capacity of the function class » does not fit the inductive bias appropriate for the problem and cannot explain the observed behavior
  - The inductive bias seems to be the smoothness of a function as measured by a certain function space norm

Generalization in modern Deep Learning - Double Descent Intuition (Belkin 2019)

- Caracterization on classification problems
  - Model: Random Fourier Features
  - Equivalent to 1 hidden layer NN with fixed weights in the first layer
    - i.e. only the last weight layers are learned, aka convex problem
    - Because of the lnearity of the trainable component, the complexity can be measured by the number of basis functions (nb of hidden cells)
      - Or at least this provides a proxy for the complexity
- RFF
  - Consider a class of function denoted $\mathcal{H}_N : h(x): R^d \rightarrow R$
    - With $h(x) = \sum_{k=1}^{N} a_k \phi(x; v_k)$ with $\phi(x; v) = \exp(i < v, x >)$ - (the complex exponential)
    - Where the $v_1, \dots, v_N$ are sampled independently from the standard normal distribution in R^d
    - The $\phi(x; v)$ are N complex basis functions
    - This may be implemented as a NN with $2N$ basis functions corresponding to the real and imaginary parts of ϕ
  - Learning procedure
    - Given a training set $(x^1, y^1) \dots (x^n, y^n)$, train via ERM, i.e. minimize $\frac{1}{n}\sum_{i=1}^{n}(h(x^i) - y^i)^2$
    - When the minimizer is not unique (always the case when $N > n$) choose the one with coefficients $(a_1, \dots, a_N)$ of minimum l_2 norm, i.e. the smoothest one

Generalization in modern Deep Learning - Double Descent Intuition (Belkin 2019)

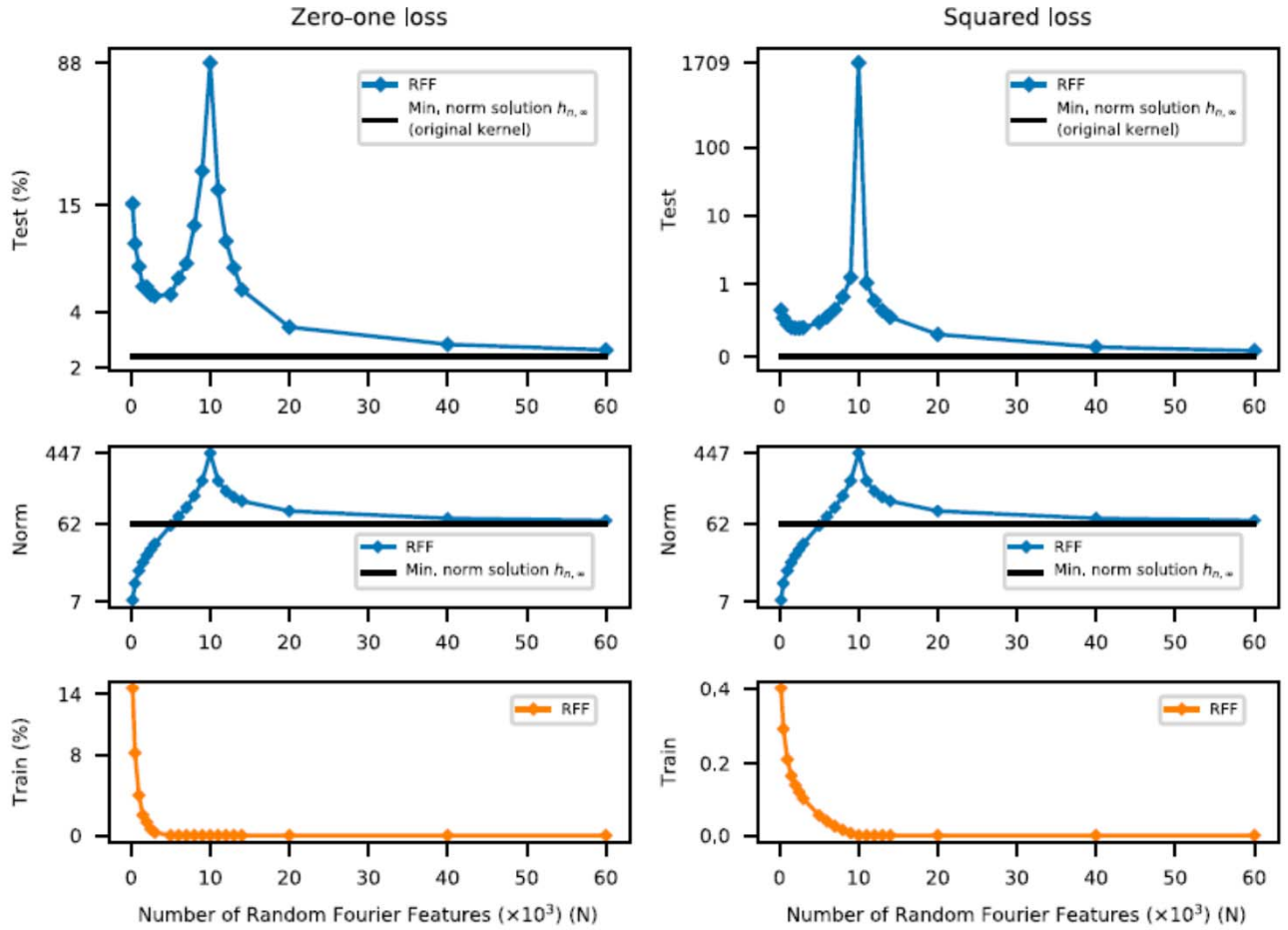


Fig. 2. Double-descent risk curve for the RFF model on MNIST. Shown are test risks (log scale), coefficient ℓ_2 norms (log scale), and training risks of the RFF model predictors $h_{n,N}$ learned on a subset of MNIST ($n = 10^4$, 10 classes). The interpolation threshold is achieved at $N = 10^4$.

Generalization in modern Deep Learning - Double Descent Intuition (Nakkiran 2020)

- Caracterize the double descent phenomenon for
  - A large variety of NN models: CNN, ResNet, Transformers
  - Several settings: model-wise, epoch-wise, sample-wise
- Propose a measure of complexity calleds « effective model complexity »
  - For non linear models, the number of parameters is not a characterization of the function class complexity

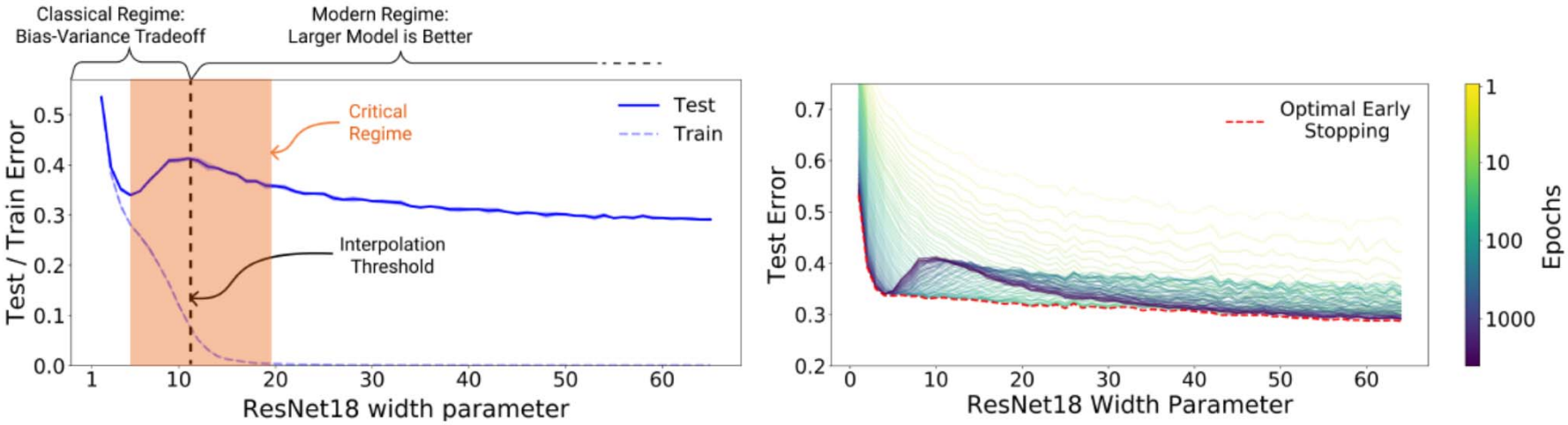


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

Generalization in modern Deep Learning - Double Descent Intuition (Nakkiran 2020)

- Effective model complexity (EMC)
  - A training procedure $\mathcal{T}$ takes as input a training set $\mathrm{D} = \{(x^1, y^1), ..., (x^n, y^n)\}$ and outputs a classifier $\mathcal{T}(D)$
  - The effective complexity of $\mathcal{T}$ w.r.t. the distribution $\mathcal{D}$ of D is the maximum number of samples n on which $\mathcal{T}$ achieves on average a zero training error
- The EMC of training procedure $\mathcal{T}$ w.r.t. distribution $\mathcal{D}$ and parameter`$\epsilon > 0$, is defined as:
  - $EMC_{\mathcal{D},\epsilon}(\mathcal{T}) = \max\{n | E_{D\sim\mathcal{D}^n}[Error_D(T(D))] \leq \epsilon$
- Regimes
  - Under-parameterized: $EMC_{\mathcal{D},\epsilon}(\mathcal{T})$ smaller than n, increasing EMC will decrease the test error
  - Over-parameterized: $EMC_{\mathcal{D},\epsilon}(\mathcal{T})$ larger than n, increasing EMC will decrease the test error
  - Critical: $EMC_{\mathcal{D},\epsilon}(\mathcal{T})$ around n, increasing EMC may decrease or increase the test error (see figure)

Generalization in modern Deep Learning - Double Descent Intuition (Nakkiran 2020)

- Different settings for characterizing the double-descent phenomenon
  - i.e. the phenomenon appears under each setting and not only under the Model-wise setting characterized by Belkin et al.
  - Model-wise
    - Models of increasing size, fixed large number of training steps
  - Epoch-wise
    - Fixed large architecture
  - Sample-wise
    - Fixed model and training procedure, change the number of training samples

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 ressources
    - sophisticated libraries
    - large computing power e.g. GPU clusters
    - large datasets, …

DeepMind

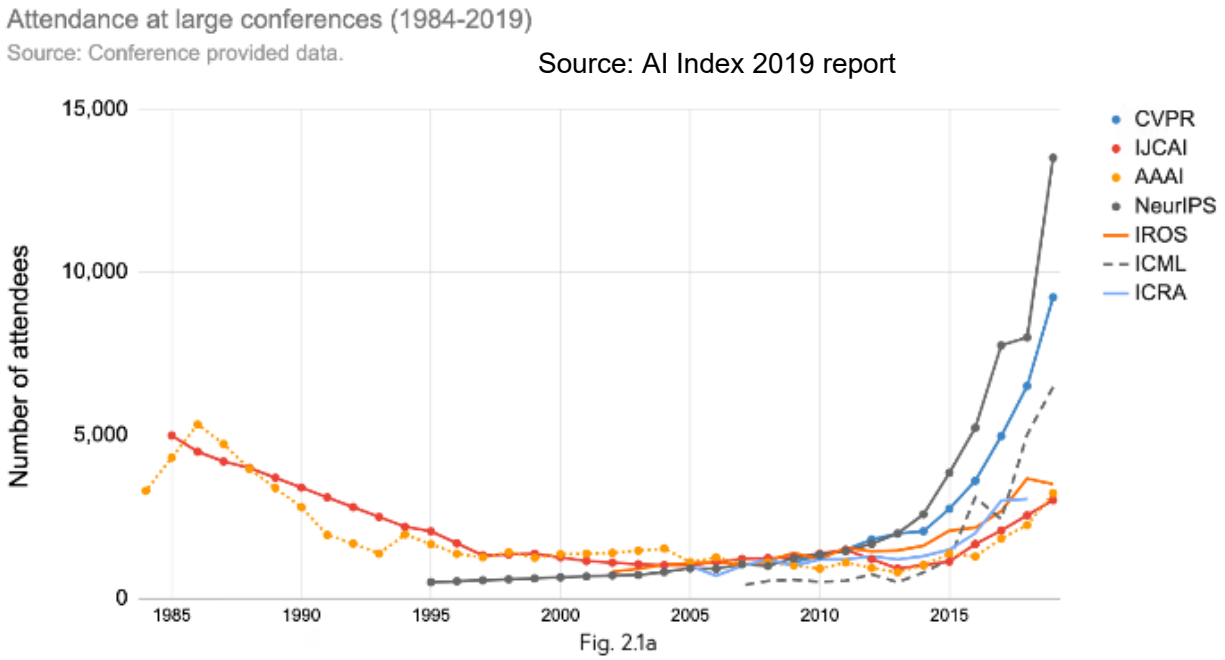
Baidu IDL 百度深度学习实验室 Relentless for Ultimate Intelligence

NVIDIA DEEP LEARNING INSTITUTE

Facebook AI Research

Interlude – ML conference attendance growth

- ML and AI conference Attendence



Source: AI Index 2019 report

Fig. 2.1a

- NIPS (Neurips)
  - 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 popular 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

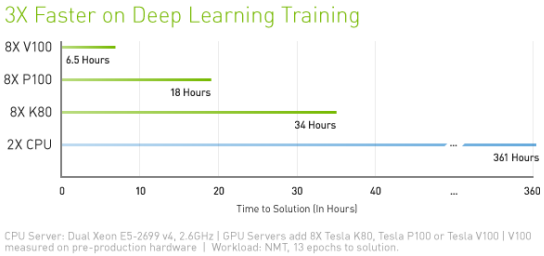
Loss criterion
- Cross entropy

Train for 20 epochs

Evaluate performance on test set

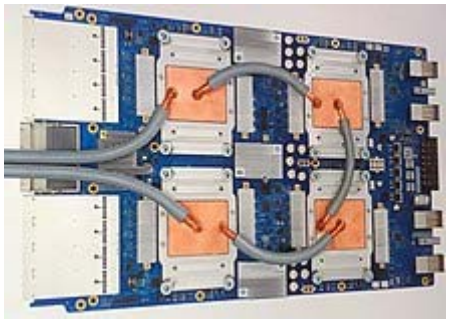
Interlude – Hardware

- 2017 - NVIDIA V100 – optimized for Deep Learning



- "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."

- Google Tensor Processor Unit – TPU V3



- 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

Motivations and historical folklore
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, …

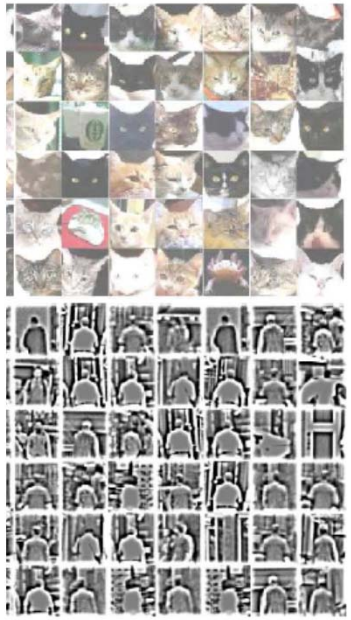


Figure 3. Top: Top 48 stimuli of the best neuron from the test set. Bottom: The optimal stimulus according to numerical constraint optimization.

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

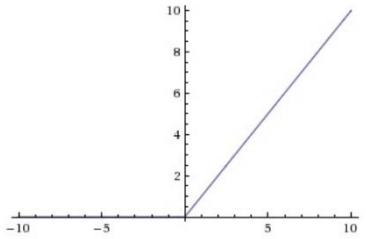
Useful Deep Learning heuristics

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

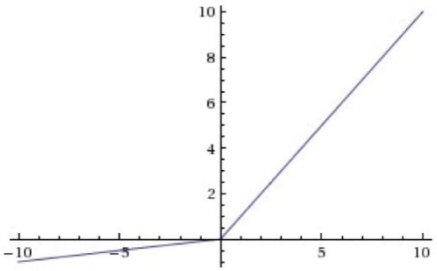
We introduce some of these ideas

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)
    - $g(\boldsymbol{x}) = \max(0, b + \boldsymbol{w}.\boldsymbol{x})$
      - Rectified units allow to draw activations to 0 (used for sparse representations) + derivative remain large when unit is active



  - Leaky RELU (used for internal layers)
    - $g(\boldsymbol{x}) = \begin{cases} b + \boldsymbol{w}.\boldsymbol{x} \text{ if } b + \boldsymbol{w}.\boldsymbol{x} > 0 \\ 0.01(b + \boldsymbol{w}.\boldsymbol{x}) \text{ otherwise} \end{cases}$
      - Introduces a small derivative when $b + \boldsymbol{w}.\boldsymbol{x} < \boldsymbol{0}$
  - Maxout
    - $g(\boldsymbol{x}) = \max_i(b_i + \boldsymbol{w}_i.\boldsymbol{x})$
    - 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
      - $g(\boldsymbol{x}) = softmax(\boldsymbol{b} + W\boldsymbol{x}) = \frac{e^{b_i + (Wx)_i}}{\sum_{j=1}^{p} e^{b_j + (Wx)_j}}$

Deep Learning Tricks
Normalisation

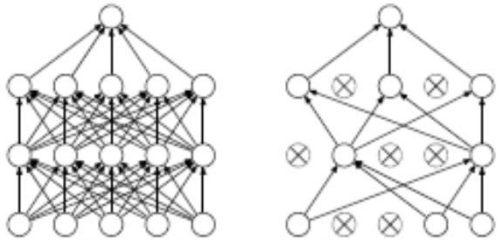
- Units: Batch Normalization (Ioffe 2015)
  - Normalize the activations of the units (hidden units) so as to coordinate the gradients accross layers
  - Let $B = \{x^1, \dots, x^N\}$ be a mini batch, $h_i(x^j)$ the activation of hidden unit i for input x^j before non linearity
  - Training
    - Set $h'_i(x^j) = \frac{h_i(x^j) - \mu_i}{\sigma_i + \epsilon}$ 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
  - One of the most effective tricks for learning with deep NNs

- 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 clippling 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

Figure from Srivastava 2014



(a) Standard Neural Net (b) After applying dropout.

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.

- 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

Dropout

- Dropout for a single unit
  - Let p be the dropout probability
  - Consider a neuron i with inputs $\boldsymbol{x} \in R^n$ and weight vector $\boldsymbol{w} \in R^n$ including the bias term
  - The activation of neuron i is $z_i = f(\boldsymbol{w}.\boldsymbol{x})$ with f a non linear function (e.g. Relu)
  - Let b_i a binomial variable of parameter $1 - p$
  - Original dropout
    - Training phase
      - $z_i = b_i f(\boldsymbol{w}.\boldsymbol{x}), b_i \in \{0,1\}$
    - Test phase
      - $z_i = \frac{1}{1-p} f(\boldsymbol{w}.\boldsymbol{x})$
  - Inverted dropout
    - Training phase
      - $z_i = \frac{1}{1-p} b_i f(\boldsymbol{w}.\boldsymbol{x}), b_i \in \{0,1\}$
    - Test phase
      - $z_i = f(\boldsymbol{w}.\boldsymbol{x})$
  - 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 $\mathrm{E}[B(m, p)] = mp$
    - L^2 normalization could be used together with dropout but reduced

CNN: Convolutional Neural Nets

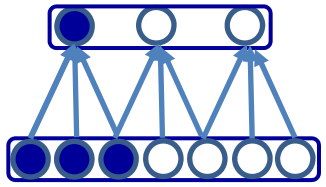
Introduction
Classification
Object detection
Image segmentation

CNNs

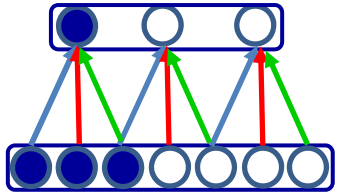
- CNNs were developped in the late 80ies for image and speech applications
- Deep CNNs were successfully used for image applications (classification and segmentation) in the 2010s – starting with the ImageNet competition, and for speech recognition.
  - Their use has been extended to handle several situations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

CNNs
principle

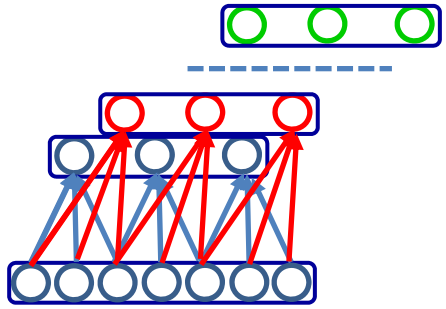
- Exploit local characteristics of the data via local connections
  - e.g. images (2 D), speech signal (1 D)



- 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)

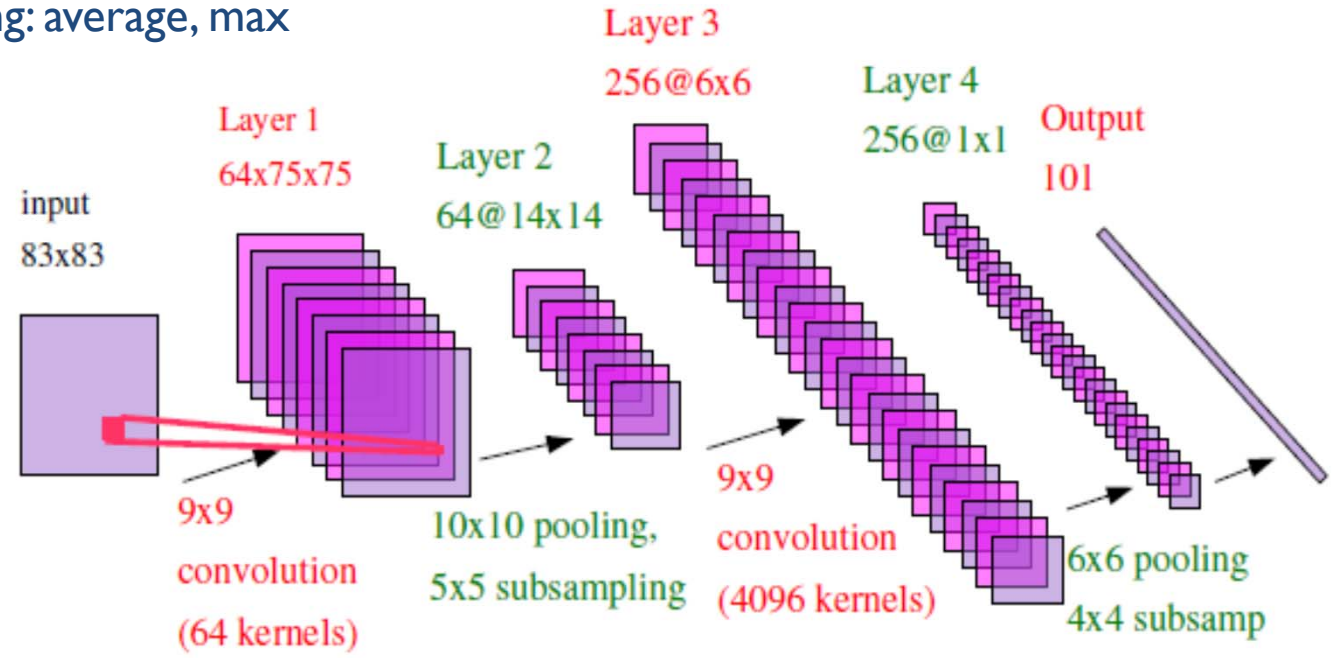


- Several convolution filters can be learned simultaneously
  - This corresponds to applying a set of local filters on the input signal
    - e.g edge detectors at different angles for an image
    - here colors indicate similar **weight vectors,** not weight values as above



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



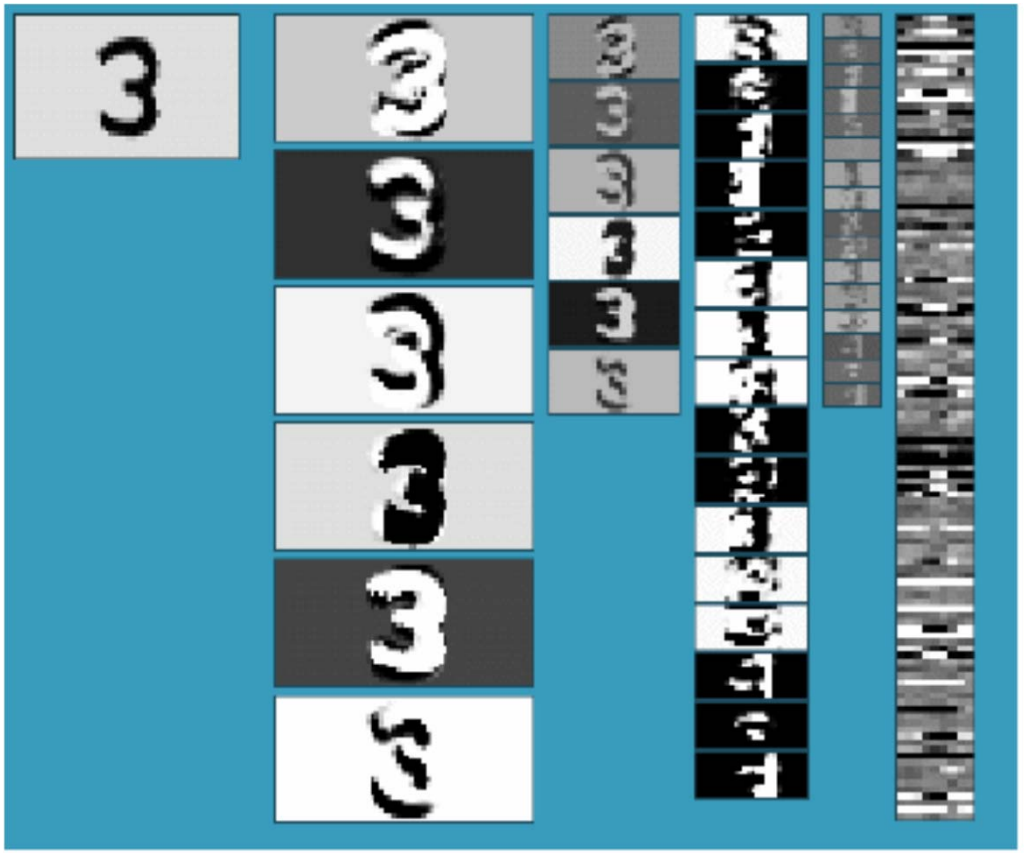
parameters 64x9x9=5184, 256x9x9=20736, 256x101 = 60916

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 tranformation 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.

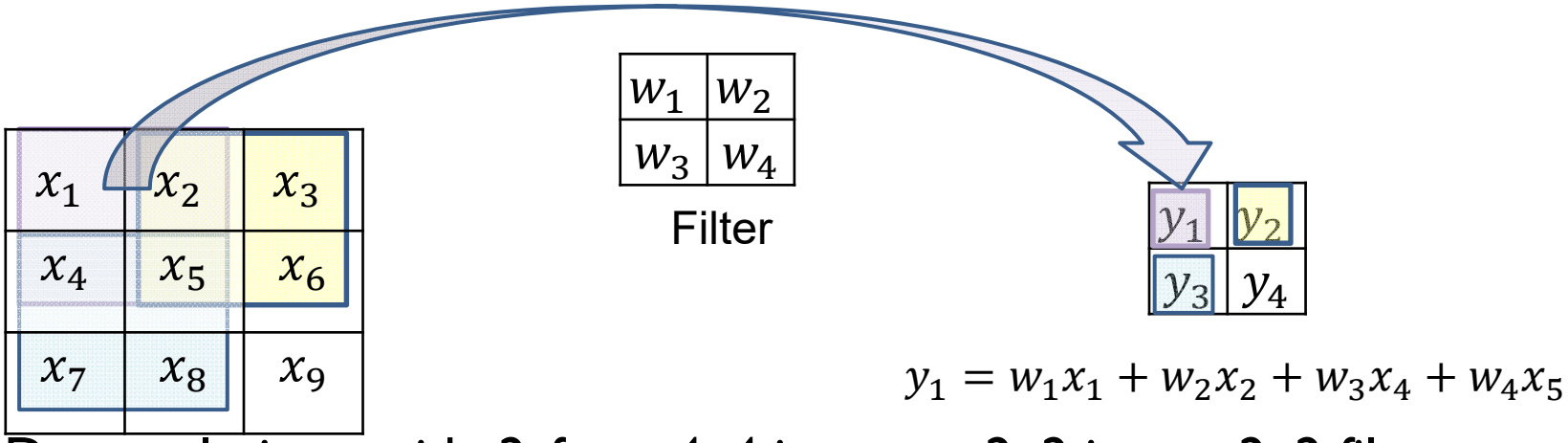
CNNs visualization

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



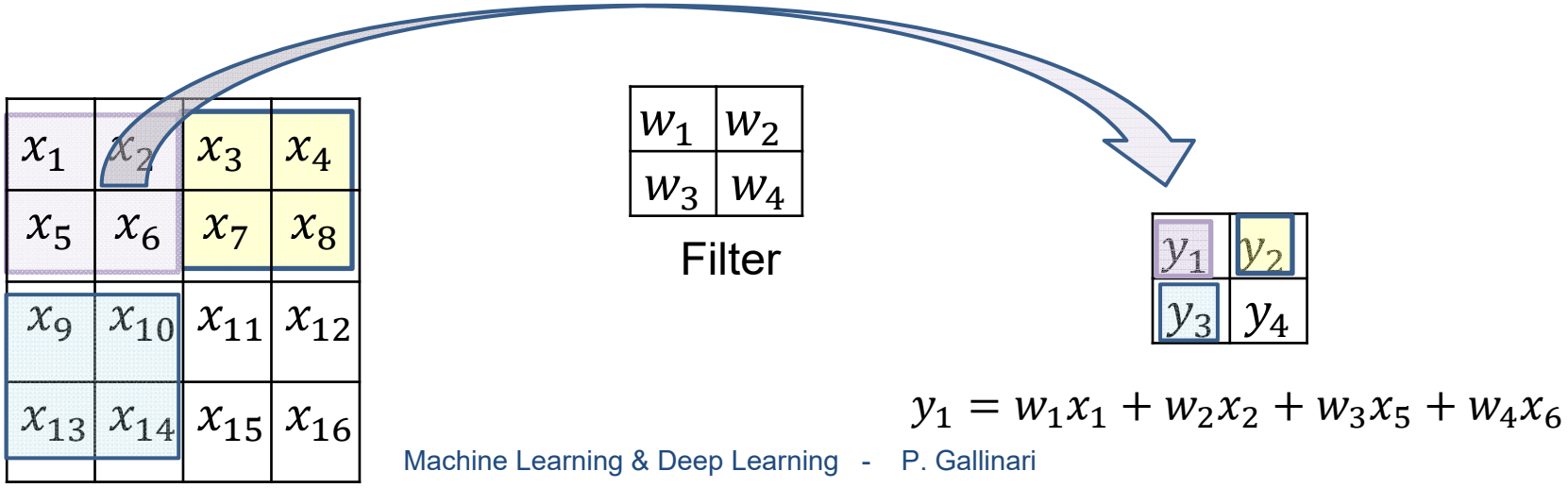
CNNs
Convolution: filter size and stride

- 2D convolution, stride 1, from 3x3 image to 2x2 image, 2x2 filter



$$y_1 = w_1x_1 + w_2x_2 + w_3x_4 + w_4x_5$$

- 2 D convolution, stride 2, from 4x4 image to 2x2 image, 2x2 filter



$$y_1 = w_1x_1 + w_2x_2 + w_3x_5 + w_4x_6$$

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

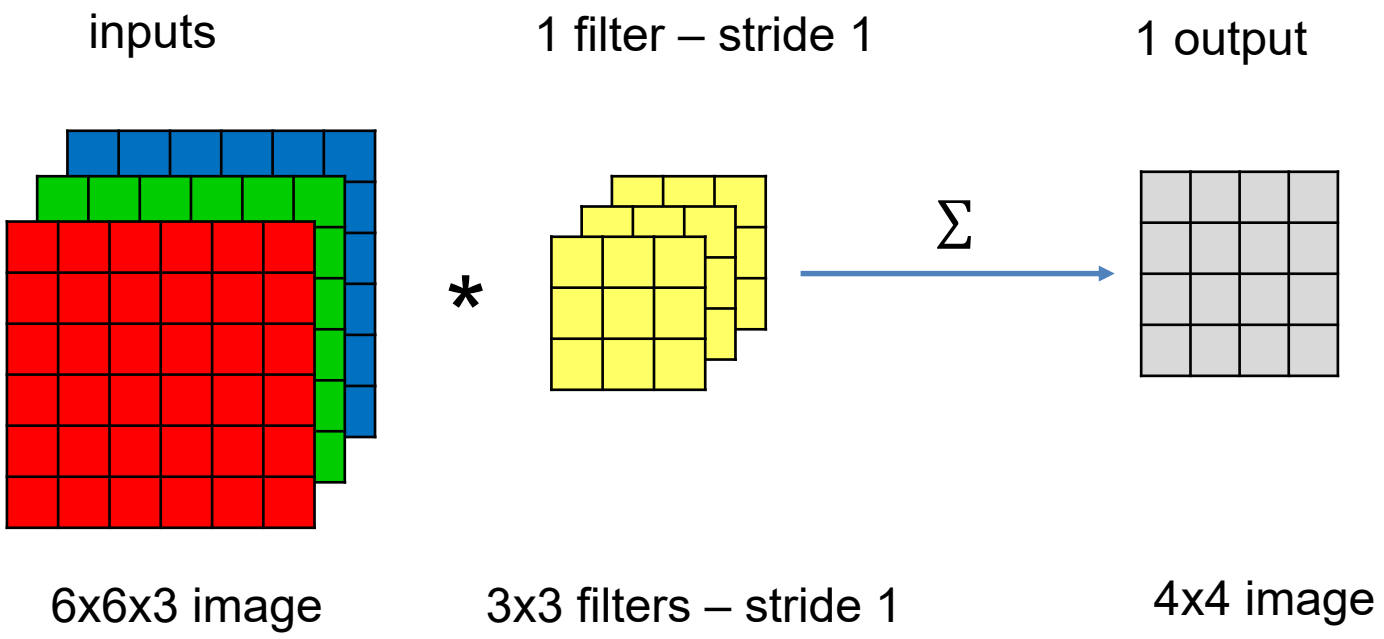
CNNs
Convolutions arithmetics

- Input image nxn, filter $f\,\mathrm{x}f$, padding p, stride s
- Output image is $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ x $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$
- Floor function [.]
  - 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 $j = i + a, a \in \{0, \ldots, s-1\}$ 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

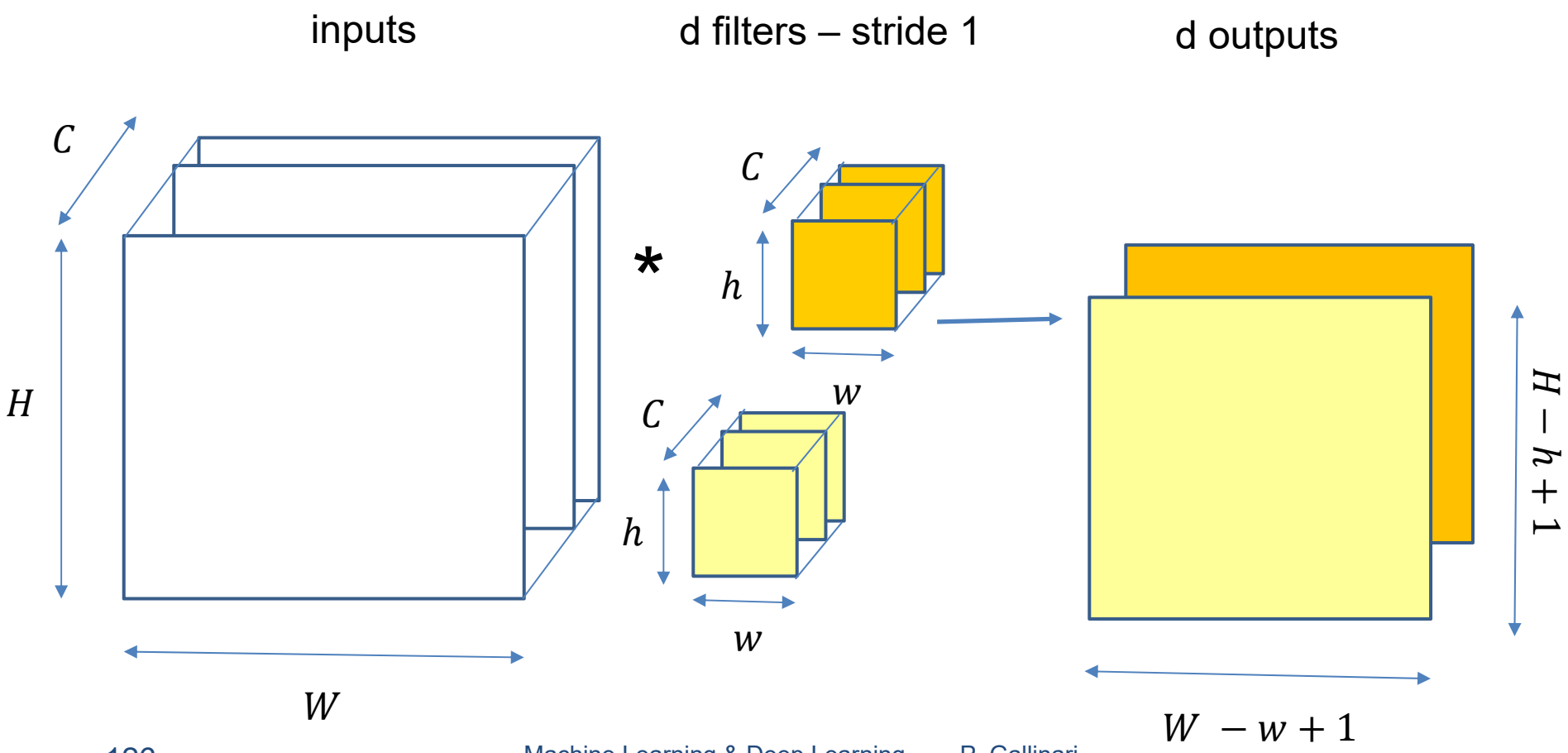
CNNs
on multiple channels, e.g. RGB images

- 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.



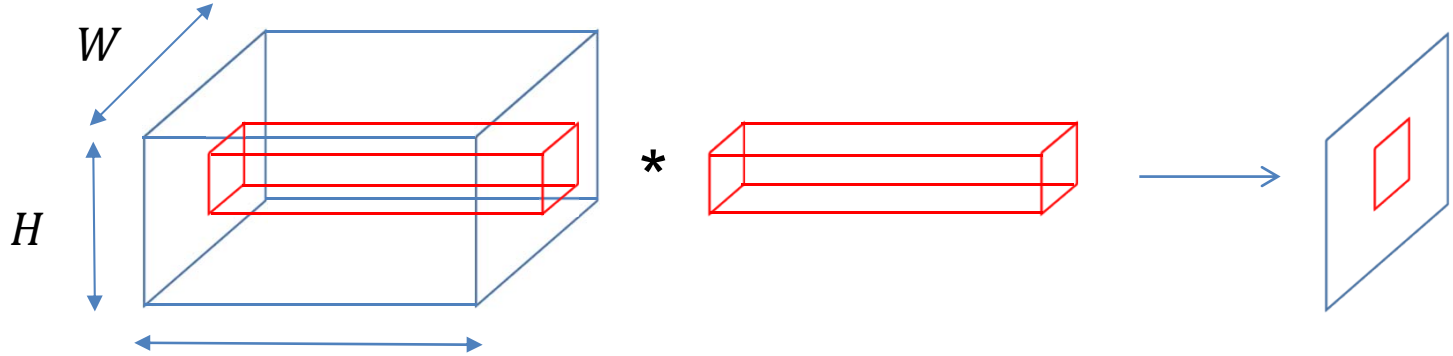
CNNs
on multiple channels

- 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 HxWxC volume to a HxWxC' volume with $C' < C$, by using C', 1x1 convolutions

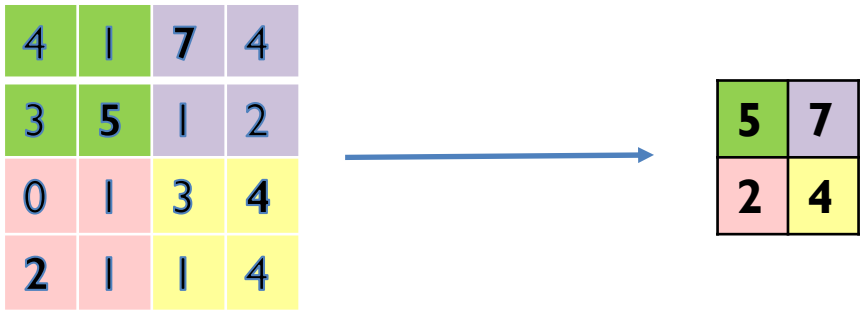


HxWxC 1x1xC HxWx1

$C' = 1$ convolution in this example

CNNs
Pooling

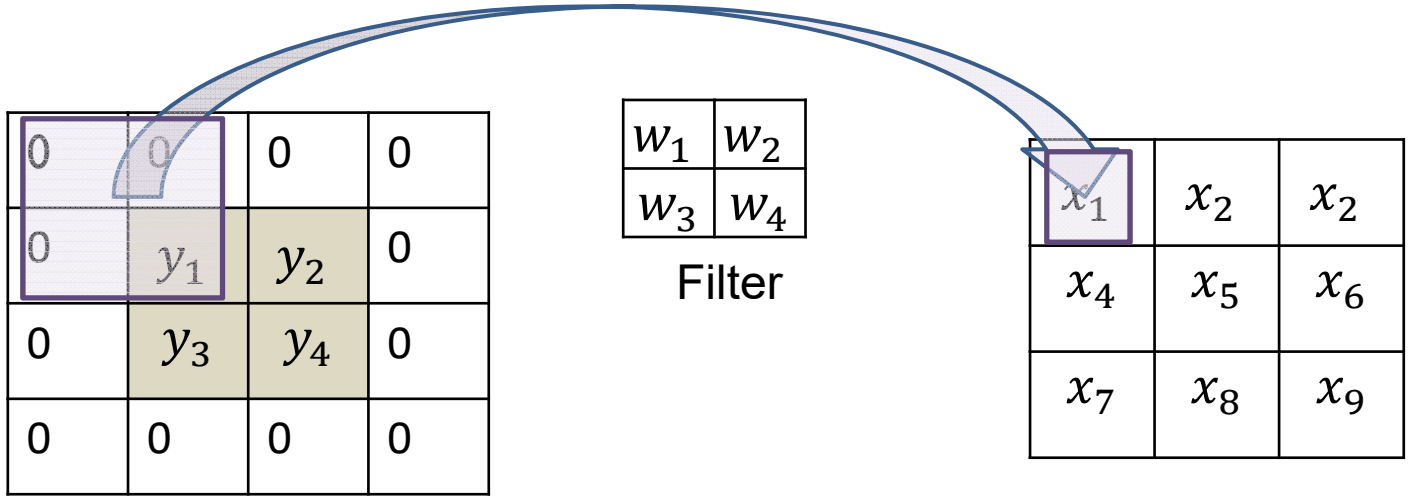
- Pooling
  - Used to aggregate information from a given layer
  - Usually Mean or Max operators are used for pooling
  - Example: Max pooling, stride 2



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

CNNs
Transposed 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

Transposed convolutions

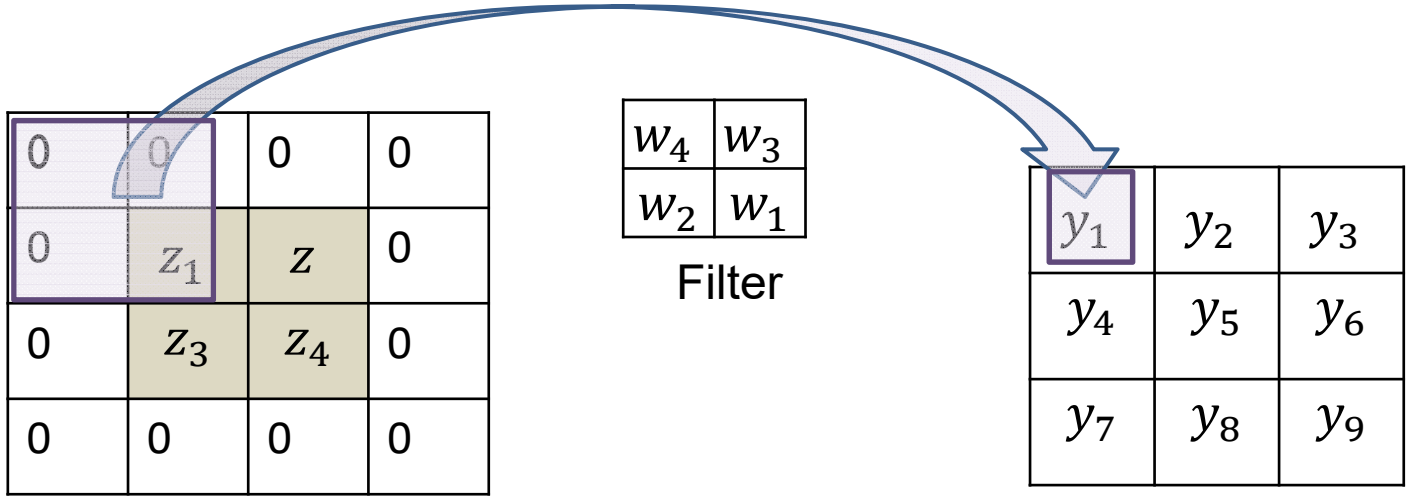
- Convolution
  - $x * w = z$, with $x \in R^9, z \in R^4$
  - $x = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix}, \mathrm{w} = \begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix}, \mathrm{z} = \begin{pmatrix} z_1 & z_2 \\ z_3 & z_4 \end{pmatrix}$

  - Convolution in matrix form
    - Lets flatten the vectors, the CNN convolution can be written in matrix form as:
    - $\mathrm{W}x = z$
    - $x = \begin{pmatrix} x_1 \\ \vdots \\ x_9 \end{pmatrix}, \mathrm{W} = \begin{pmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{pmatrix}, \mathrm{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix}$

- Transposed convolution
  - Transposed convolution in matrix form $y = W^T z, \quad z \in R^4 \; and \; y \in R^9$
  - $\mathrm{W}^{\mathrm{T}} = \begin{pmatrix} w_1 & 0 & 0 & 0 \\ w_2 & w_1 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ w_3 & 0 & w_1 & 0 \\ w_4 & w_3 & w_2 & w_1 \\ 0 & w_4 & 0 & w_2 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & w_4 & w_3 \\ 0 & 0 & 0 & w_4 \end{pmatrix}$

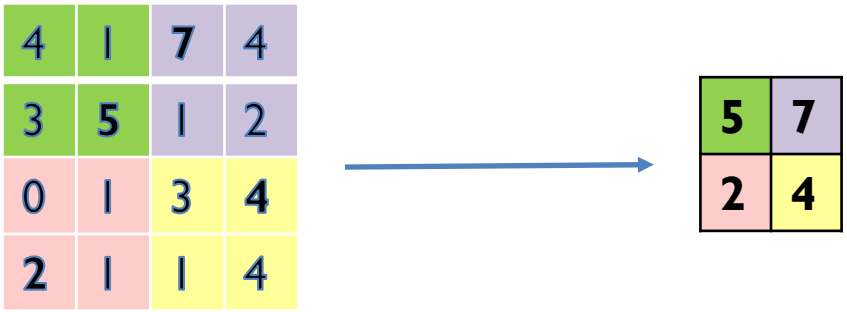
Transposed convoution

- Transposed convolution in convolutional form $y = z * w$

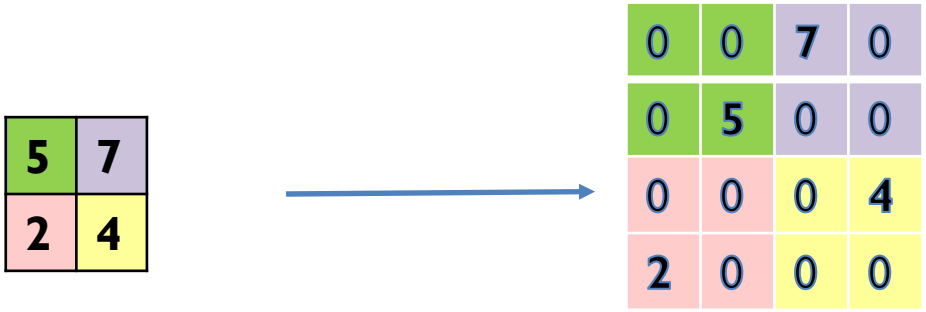


CNNs
Unpooling

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

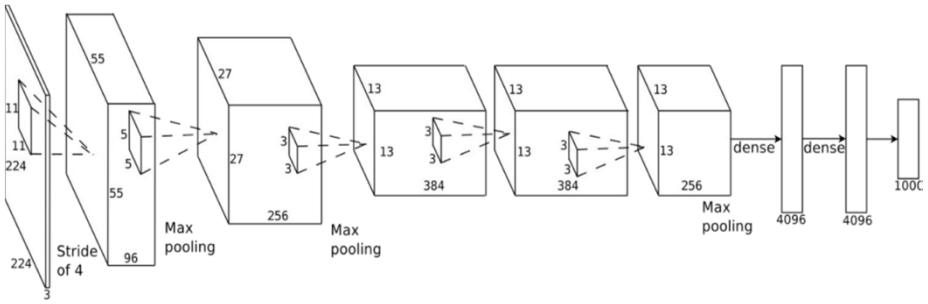


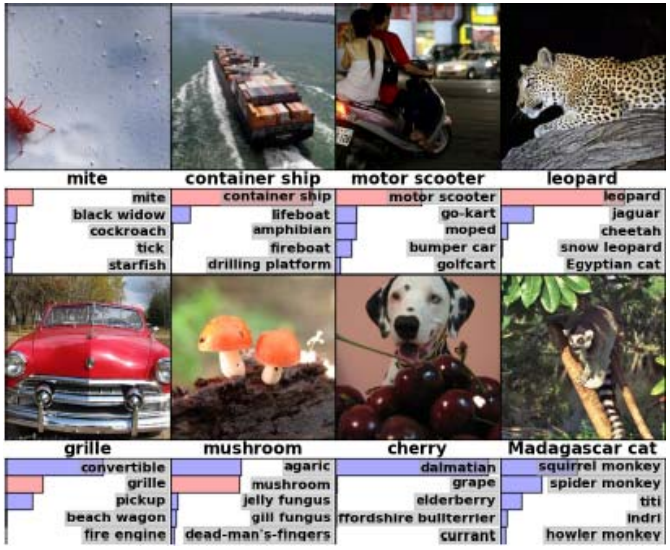
- Remember the positions of the max and fill the other positions with 0



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



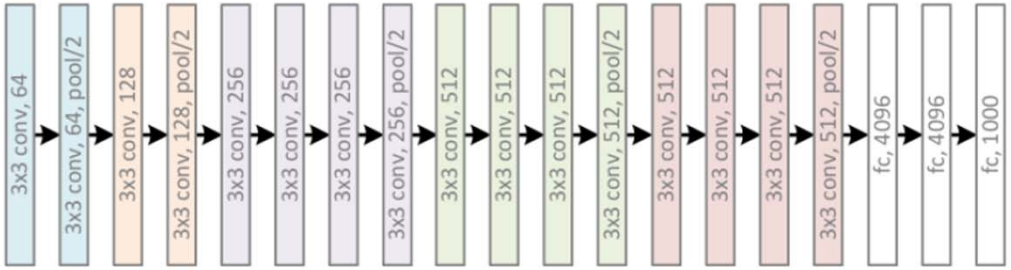


CNNs
Very Deep Nets trained with GPUs

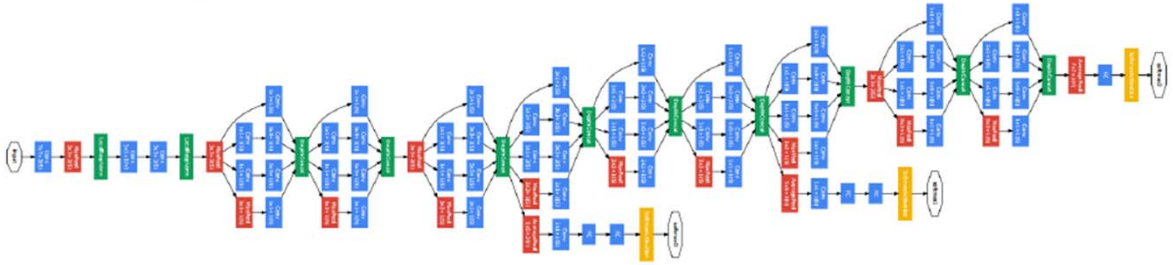
Deeper Nets with small filters – training time several days up to 1 or 2 weeks on ImageNet

Oxford, [Simonyan 2014], Parameters 138 M

VGG, 16/19 layers, 2014



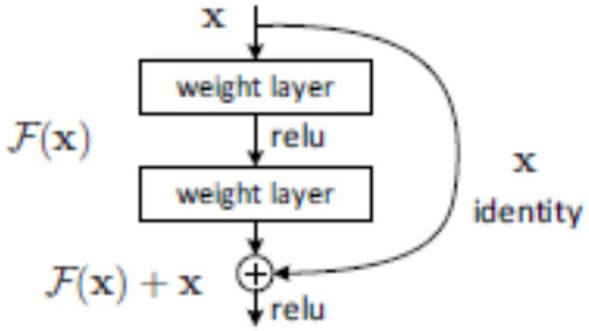
GoogleNet, 22 layers, 2014 Google, [Szegedy et al. 2015], Parameters 24 M



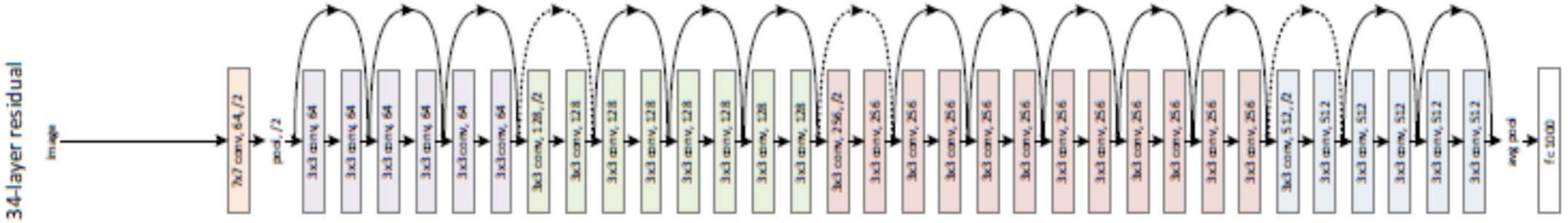
ResNet, 152 layers, 2015 MSRA, [He et al. 2016] , Parameters 60 M

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

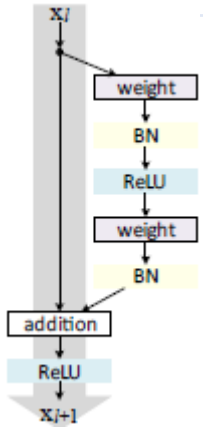


Fig. He 2016, original ResNet block

  - 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
  - $\frac{\partial C}{\partial x_t} = \frac{\partial C}{\partial x_T}\frac{\partial x_T}{\partial x_t} = \frac{\partial C}{\partial x_T}(1 + \frac{\partial}{\partial x_t}\sum_{i=t}^{T-1} F(x_i, W_i))$
  - 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
    - $X(t_1) = X(t_0) + hF\left(X(t_0), \theta_{t_0}\right)$
    - $X(t_2) = X(t_1) + hF\left(X(t_1), \theta_{t_1}\right), \ldots$

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)

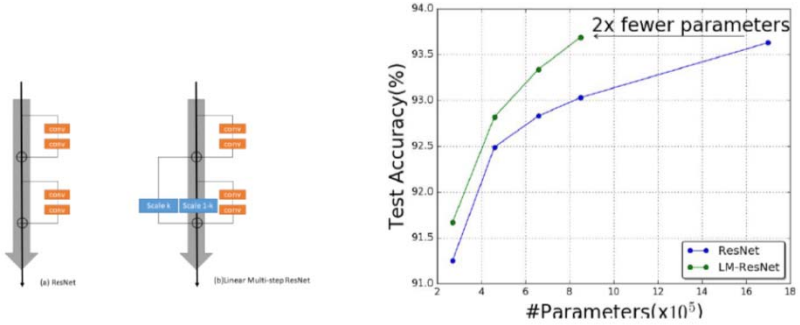


Figure 2: LM-architecture is an efficient structure that enables ResNet to achieve same level of accuracy with only half of the parameters on CIFAR10.

- 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. Bottleneck.
3	U. Oxford	0.26979	
4	Xerox/INRIA	0.27058	

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

- ImageNet 2014 – 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 2014 – 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

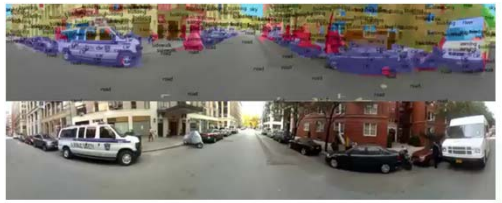
Rank	Name	Error rate	Description
1	NYU	0.11197	Deep learning
2	NUS	0.12535	Deep learning
3	Oxford	0.13555	Deep learning

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

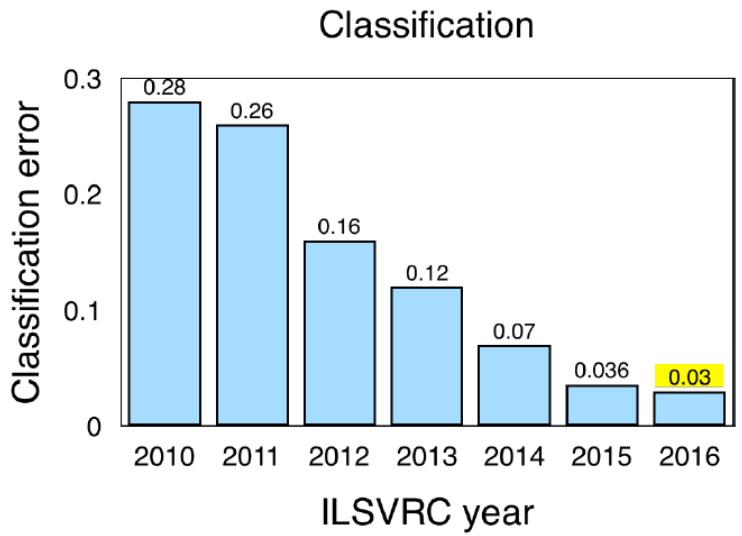
- ImageNet 2013 – object detection challenge

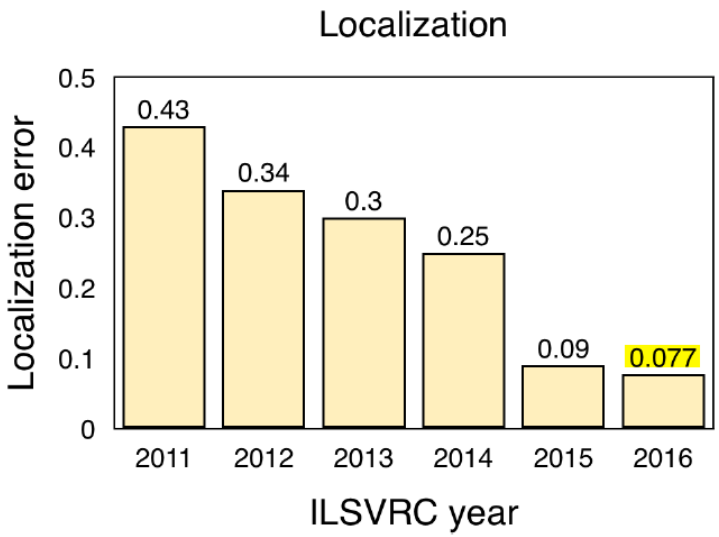
Rank	Name	Mean Average Precision	Description
1	UvA-Euvision	0.22581	Hand-crafted features
2	NEC-MU	0.20895	Hand-crafted features
3	NYU	0.19400	Deep learning

CNN examples



Convolutional Nets
ILSVRC performance over the years

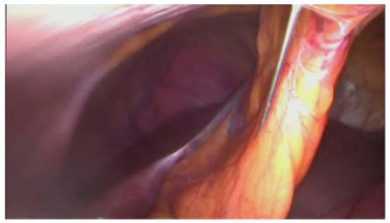


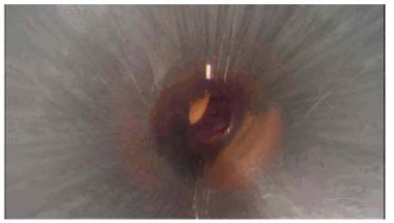


Classification
CNNs and Transfer Learning

- 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

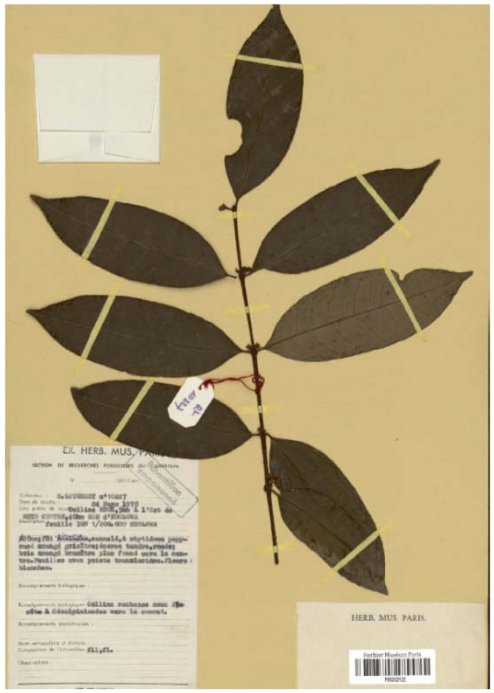
Table 1: Forward+Backward with batches of 20 images.

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 worlwide plant collections
  - O(1000) classes, e.g. opposed/alternate leaves; simple/composed leaves; smooth/with teeth leaves, ….
- Pretrained ResNet



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

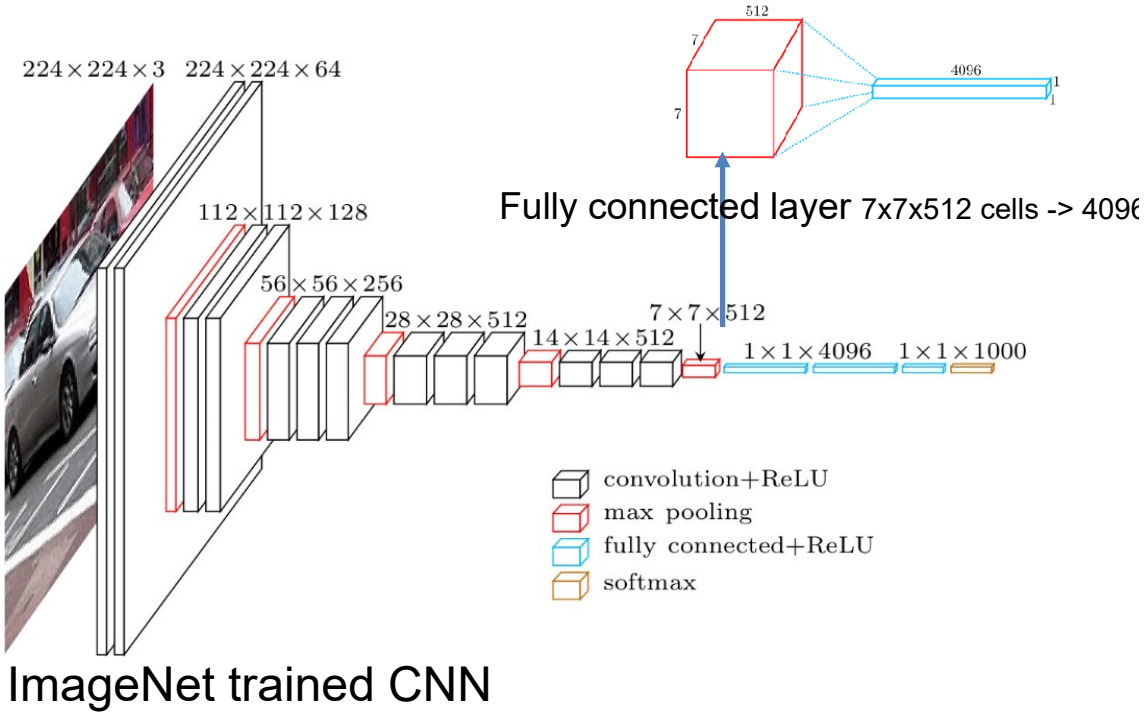
MIT67

15 Scene

COCO

VOC12 Action

Classification - CNNs – Classification of large images (Durand 2016)
Sliding window => Convolutional Layers



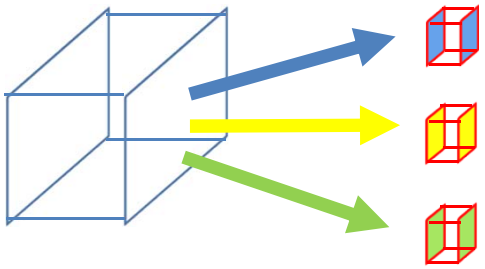
ImageNet trained CNN

- 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 7x7x512 cells → 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

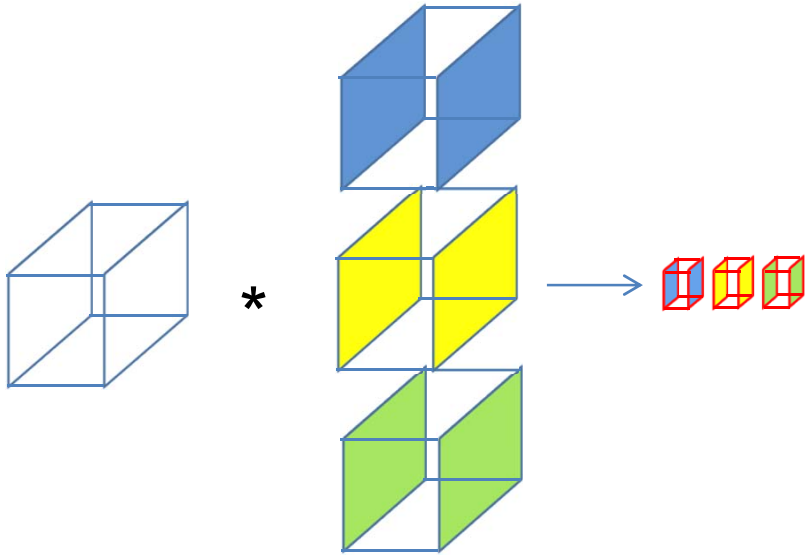


Cell layer
$nxnxC$

Each weight
vector is
$nxnxC$

Outputs
1 each

FCN convolutional view



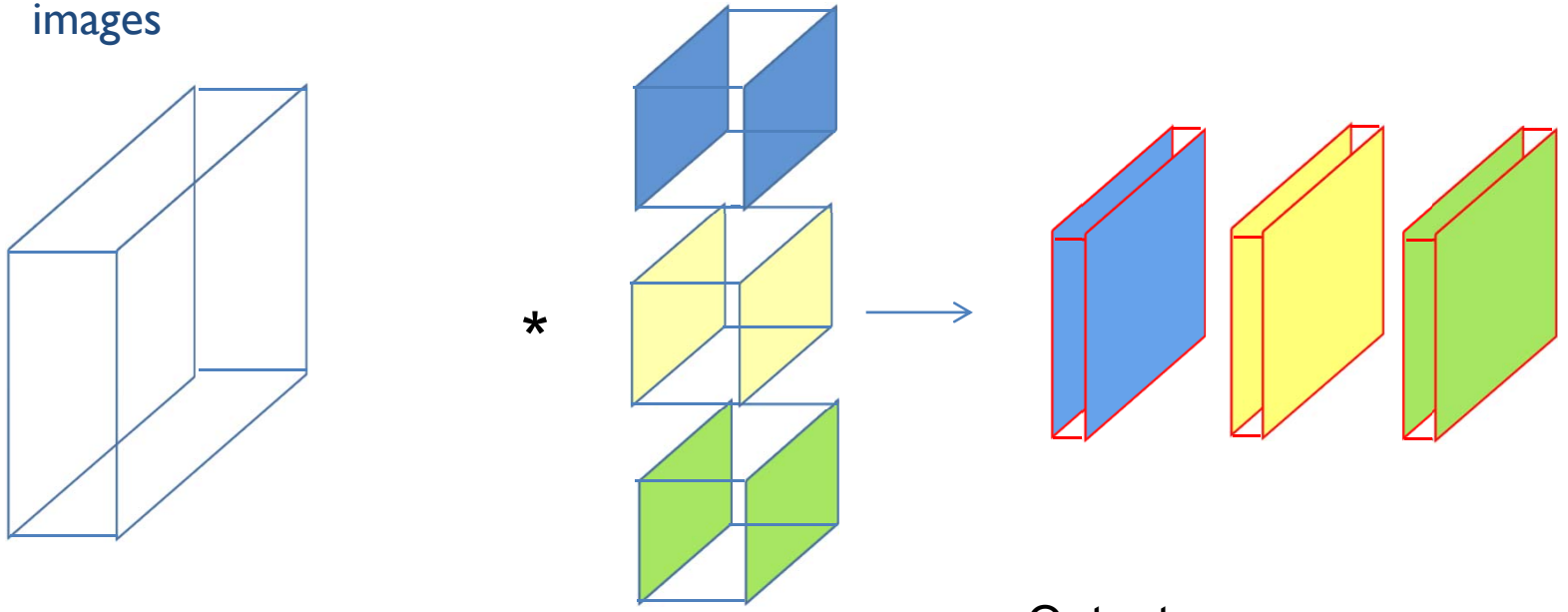
Cell layer
$nxnxC$

Weight layer
each is $nxnxC$

Outputs
1 each

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

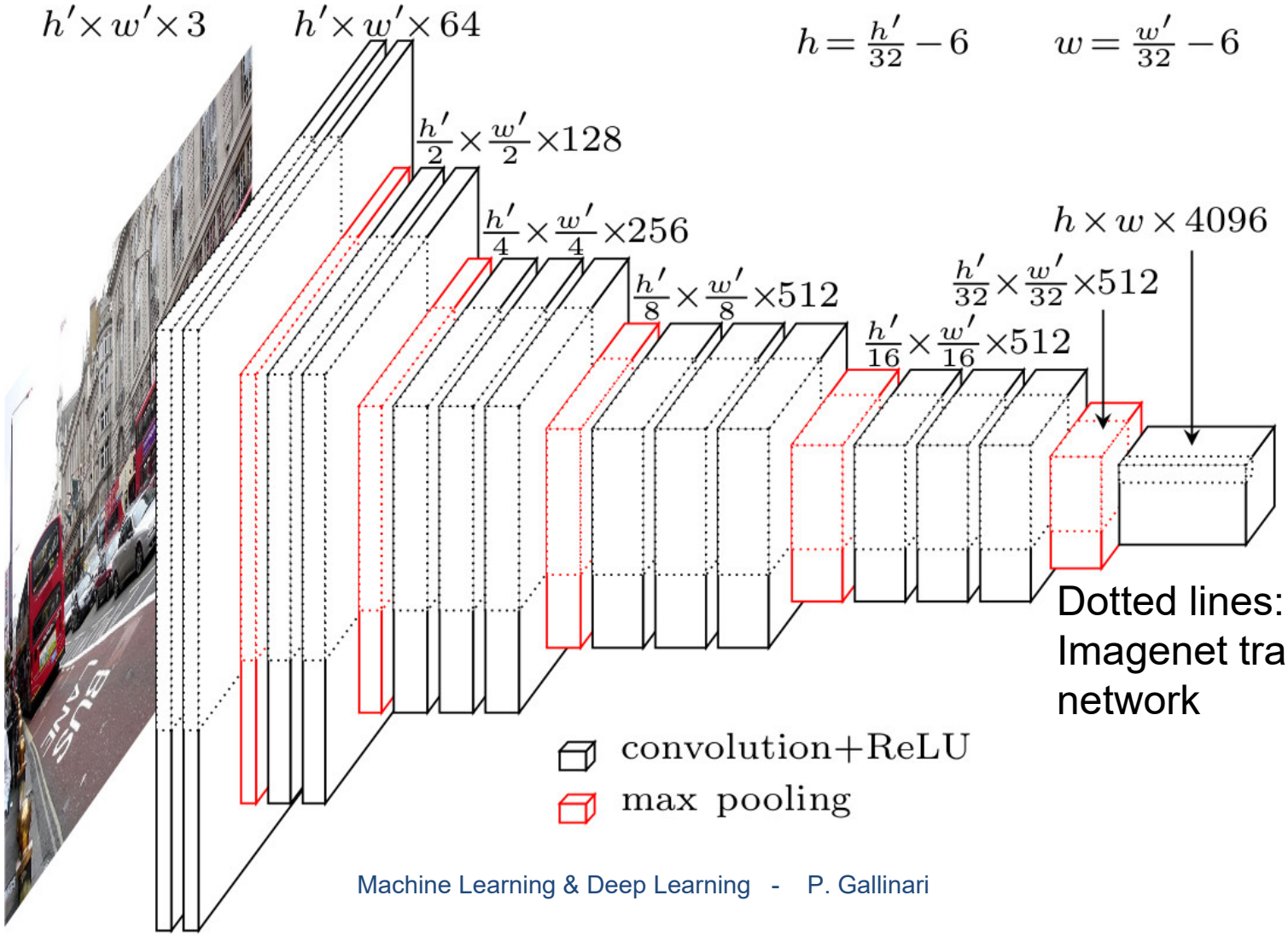


Cell layer
NxNxC

Weight layer
nxnxC each

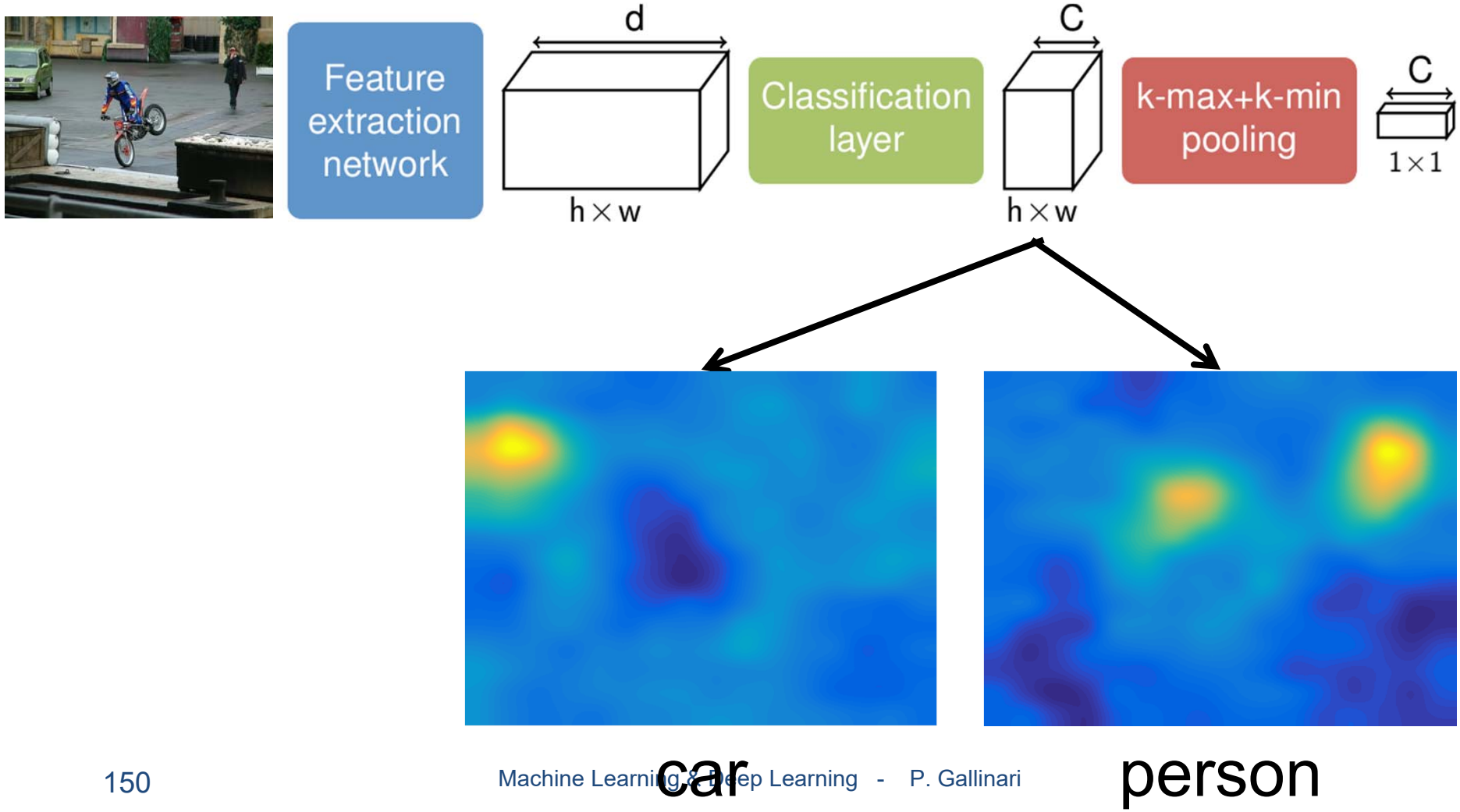
Outputs
$(N-n+1)$x$(N-n+1)$x1 each

CNNs – Classification of large images (Durand 2016)
Sliding window => Convolutional Layers



Dotted lines: initial Imagenet trained network

CNNs – Classification of large images (Durand 2016)
Sliding window => Convolutional Layers



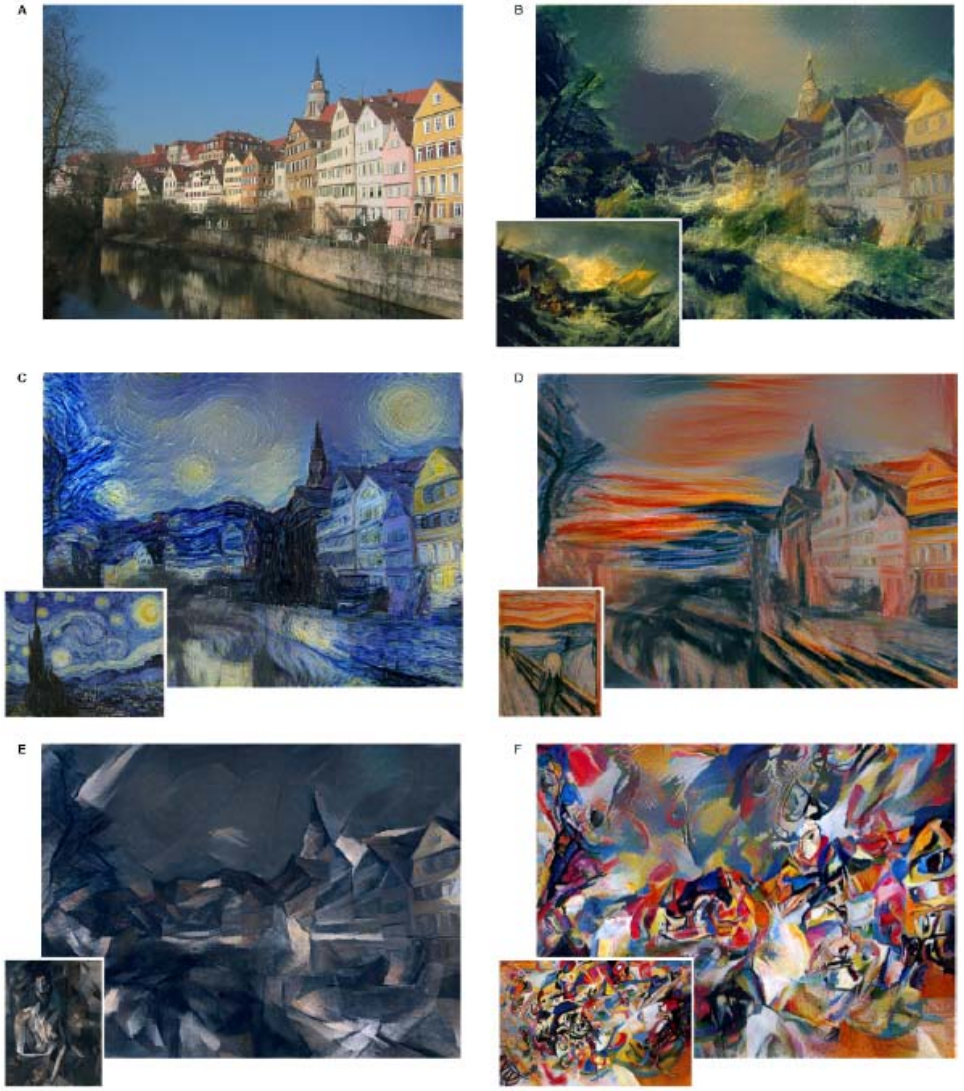
CNN : A neural algorithm of Artistic Style (Gatys et al. 2016)

Generate images by combining content and style

Makes use of a discriminatively trained CNN

Image generation

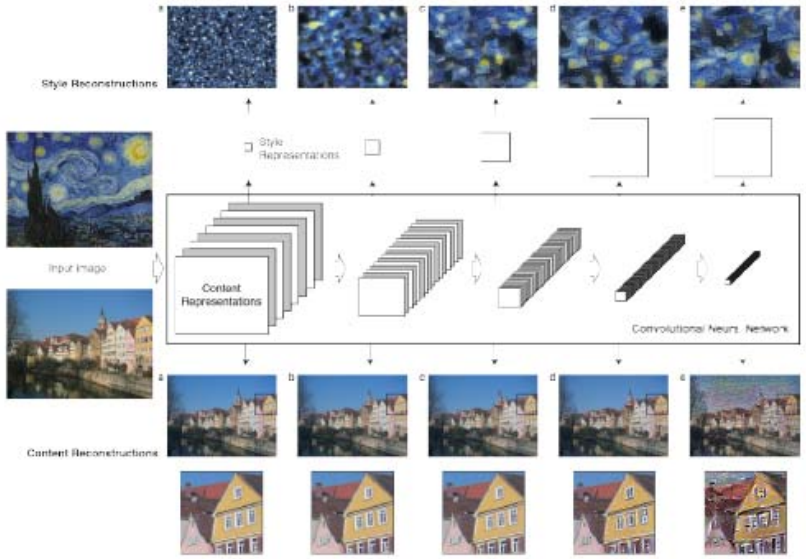
- inverse problem on the CNN



https://deepart.io

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

- Idea (simplified)
  - Use a pre-trained ImageNet NN
  - $\boldsymbol{c}$ input content image, F_c a filter representation of $\boldsymbol{c}$
  - $\boldsymbol{a}$ input art image, G_a a filter correlation representation of $\boldsymbol{a}$
  - $\boldsymbol{x}$ 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
    - $\hat{\boldsymbol{x}} = argmin_x(L)$
    - Solved by gradient



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

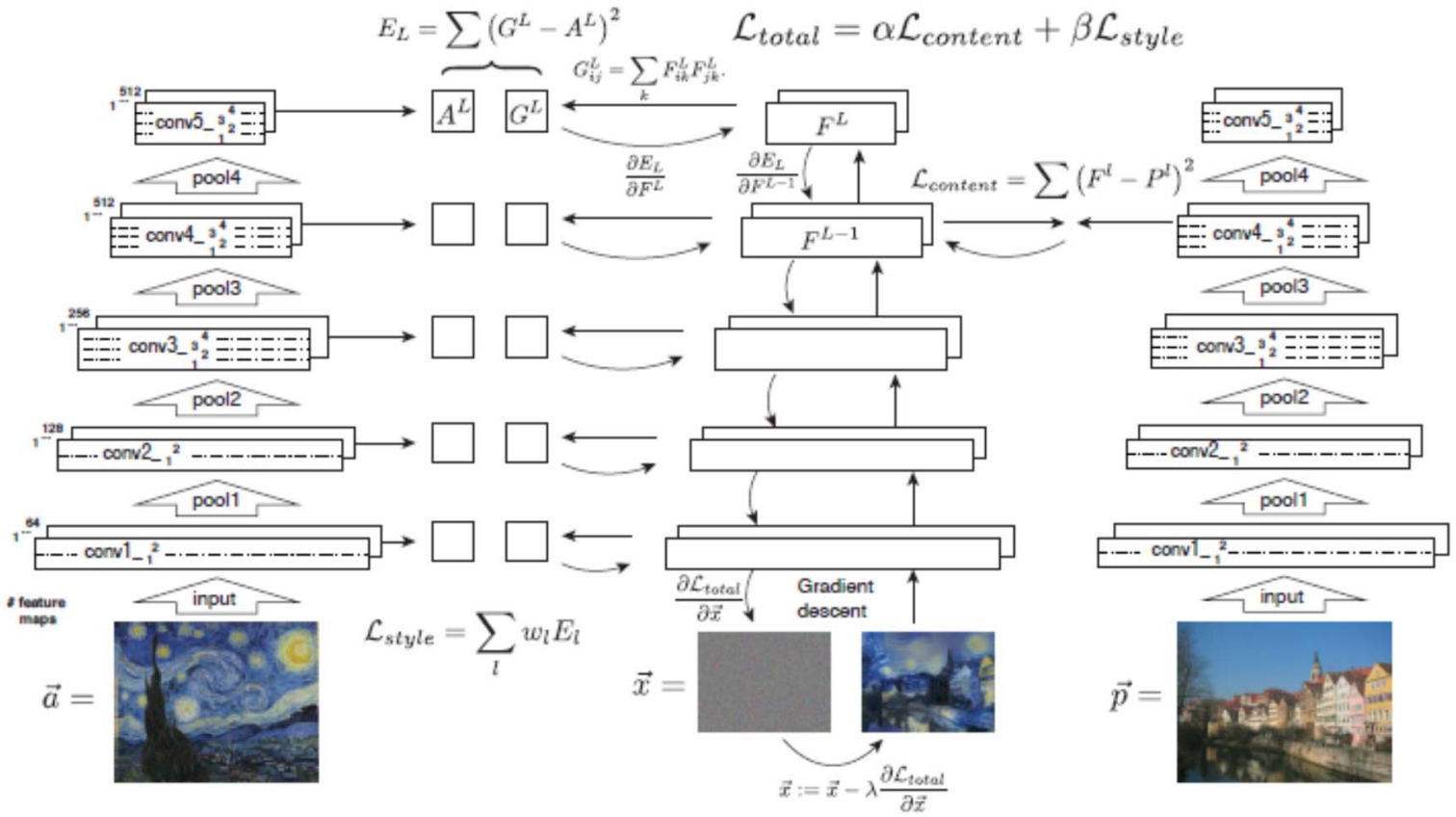
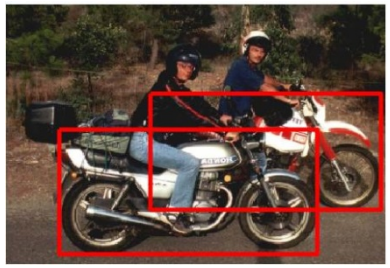


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image $\vec{a}$ is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image $\vec{p}$ is passed through the network and 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: $IoU = \frac{area(B \cap \hat{B})}{\text{area}(B U \hat{B})}$
- 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 SxS grid
  - Each grid cell predicts
    - B bounding boxes and confidence for these boxes
      - 5 numbers per box: (x, y): $box\ center$, (w, h): $box\ dimension$, $confidence$
      - $confidence = P(Object).IoU(target, pred)$
        - $P(Object)$ 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
      - $P(Class|Object)$
      - Note: at inference time they use the following score
        - $P(Class|object).P(Object).IoU(target, pred)$ instead of $P(Class|Object)$
          - 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

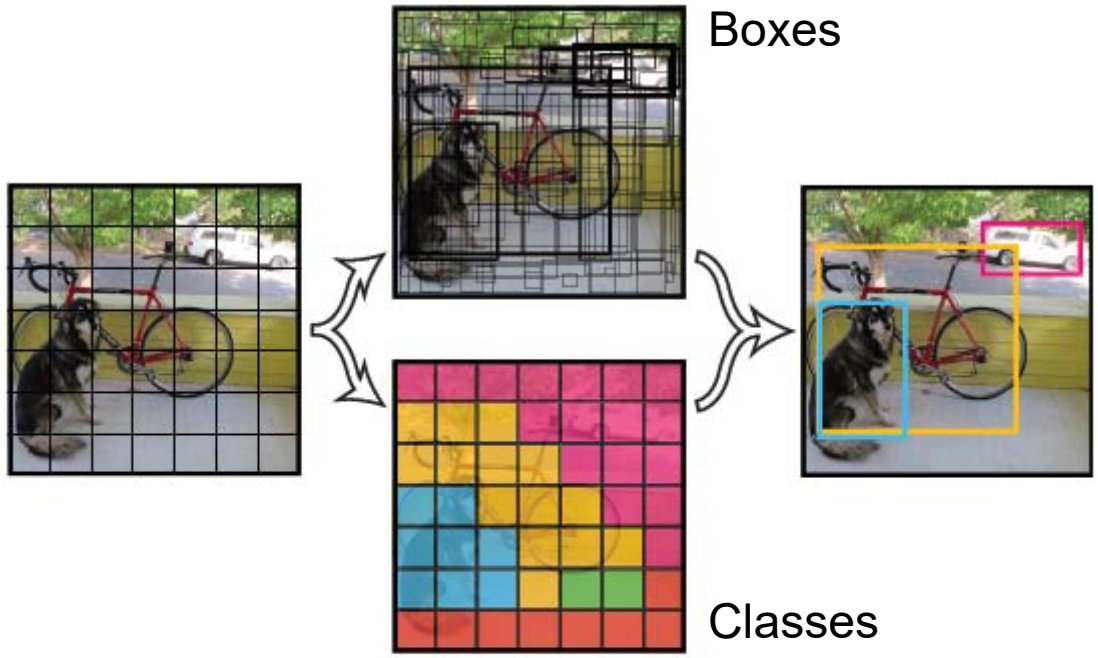


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an even grid and simultaneously predicts bounding boxes, confidence in those boxes, and class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

CNNs for Object detection
Case study: YOLO (Redmon 2015) - Network Design

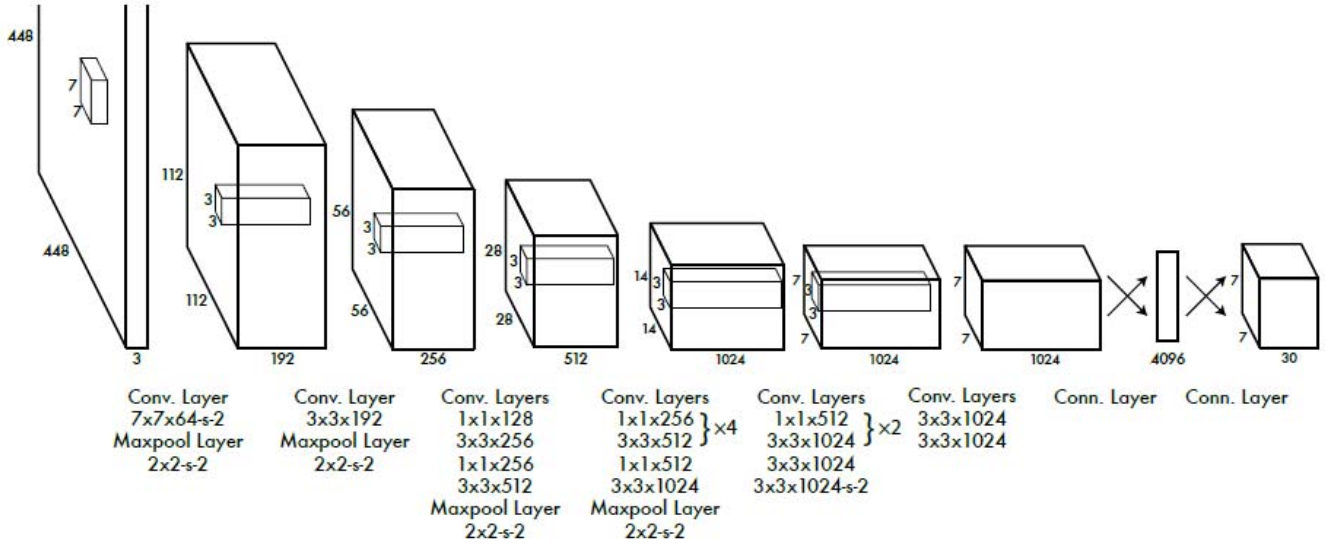


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224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

Output : SxSx(Bx5 + C) tensor

for Pascal Voc dataset: SxSx(Bx5 + C) = 7x7x(2x5 + 20)

With B: # boxes and C: # classes

Several 1x1xn convolutional structures to reduce the feature space from preceeding 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:

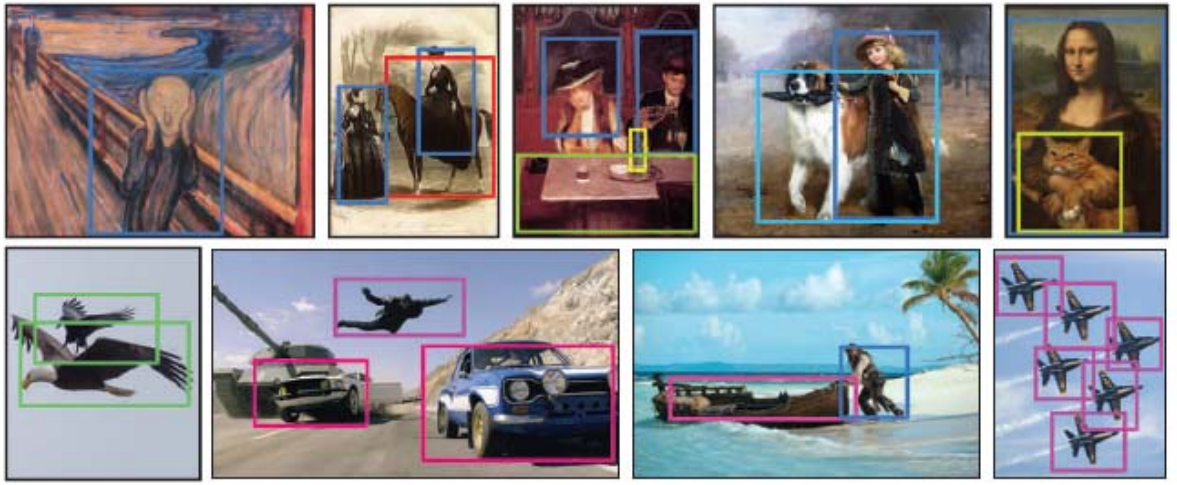
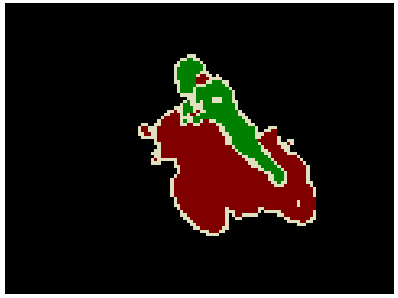


Figure 6: **Qualitative Results.** YOLO running on artwork and natural images. It is mostly accurate although it does think one person in an image is an airplane.

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 developped
    - Encoder – Decoder architectures with skip connections
      - Encoder are similar to the ones used for classification and decoders use Transpose Convolutions and Unpooling
    - Dilated or a Trous 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

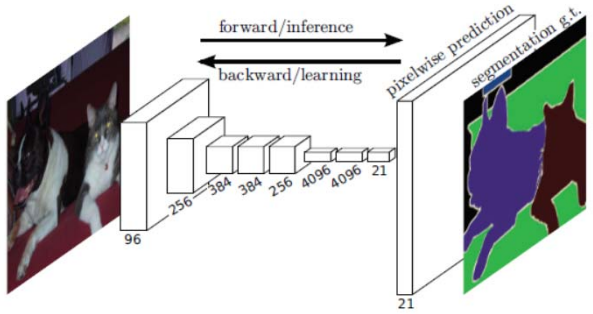


Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

- Fully connected -> convolutional trick

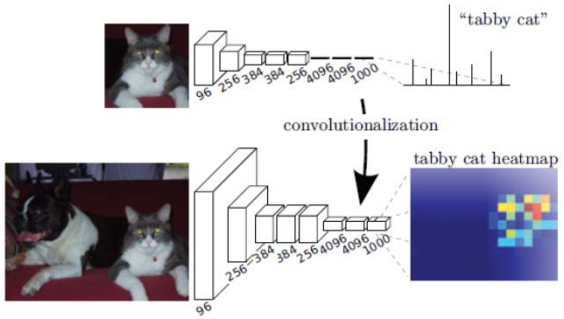


Figure 2. Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss (as in Figure 1) produces an efficient machine for end-to-end dense learning.

- End to end training for segmentation

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 x 32 upsampling
  - Improved results where obtained by combining several resolutions in the DNN

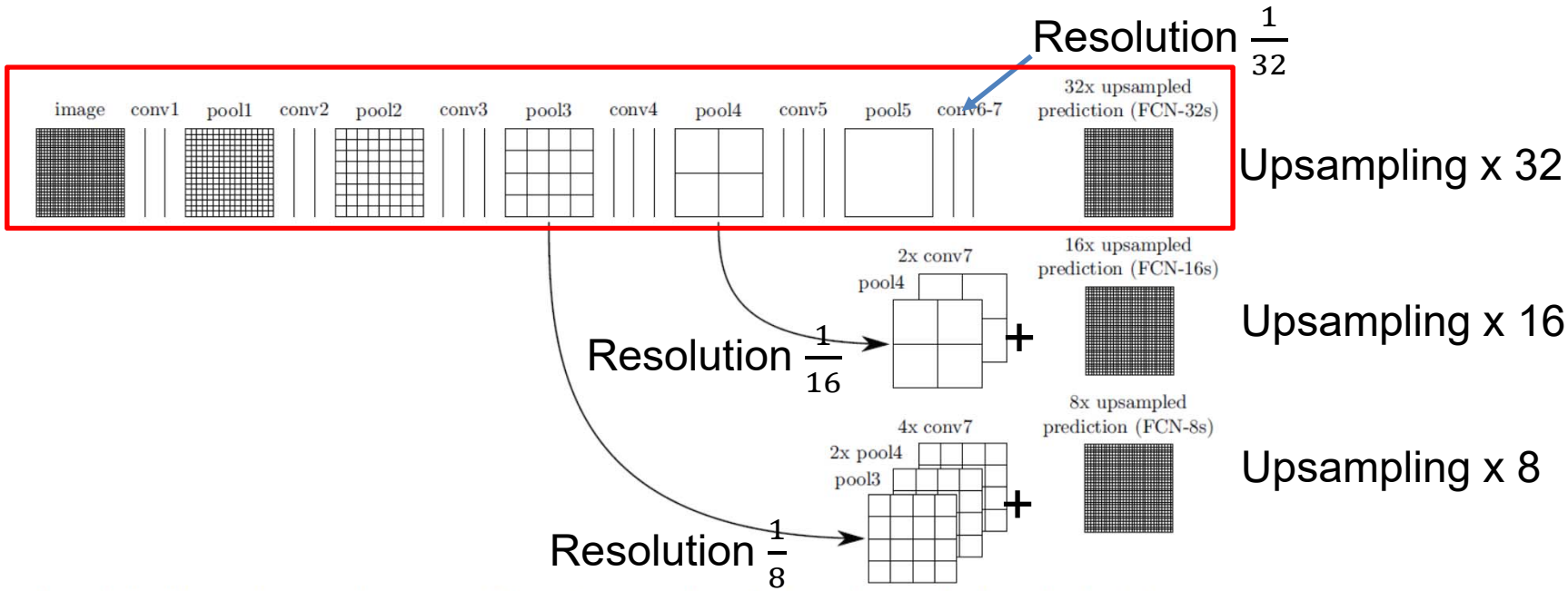


Figure 3. Our DAG nets learn to combine coarse, high layer information with fine, low layer information. Pooling and prediction layers are shown as grids that reveal relative spatial coarseness, while intermediate layers are shown as vertical lines. First row (FCN-32s): Our single-stream net, described 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

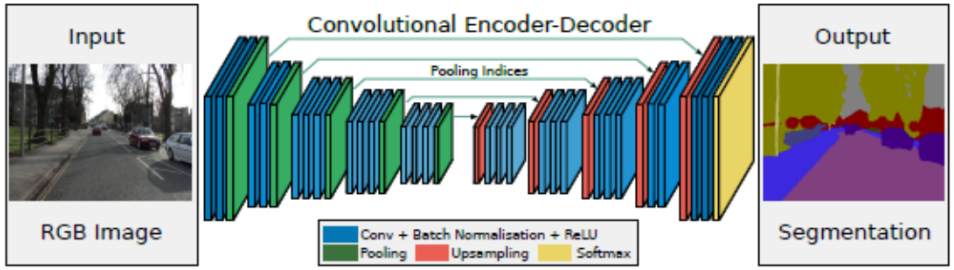


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to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

SegNet – (Badrinarayanan 2017)

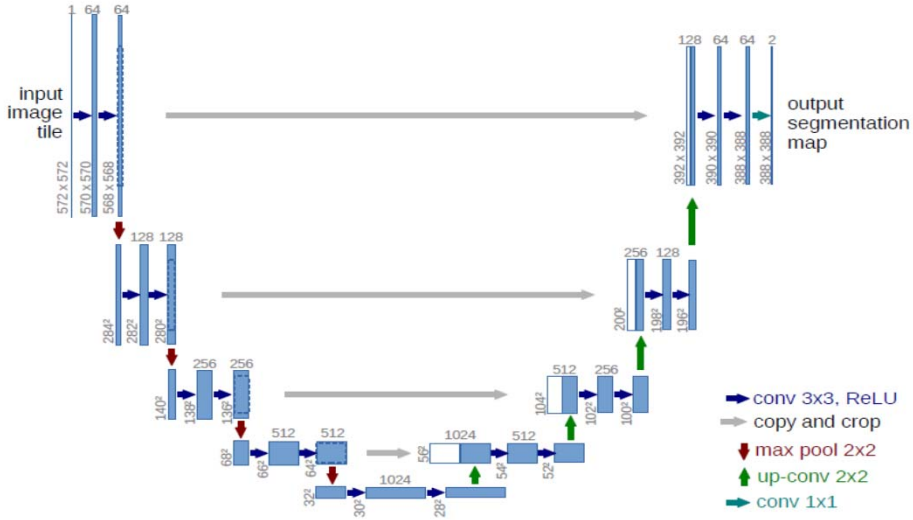


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows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

U-Net, (Ronneberger 2015)

- Noisy data for vision
  - Random rotations
  - Random flips
  - Random shifts
  - Random “zooms”
  - Recolorings

Recurrent networks

RNNs
Examples of tasks and sequence types

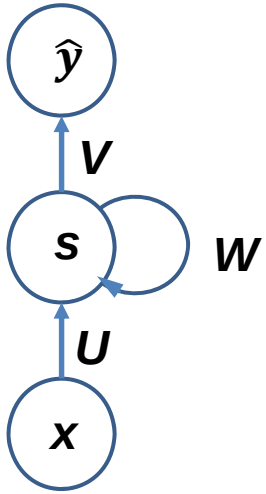
- Sequence classification
  - Input: sequence, output: class
    - Time series classification
    - Sentence classification (topic, polarity, sentiment, etc.)
- Sequence generation
  - Input: initial state (fixed vector), output: sequence
    - Text Generation
    - Music
- Sequence to sequence transduction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

RNNs

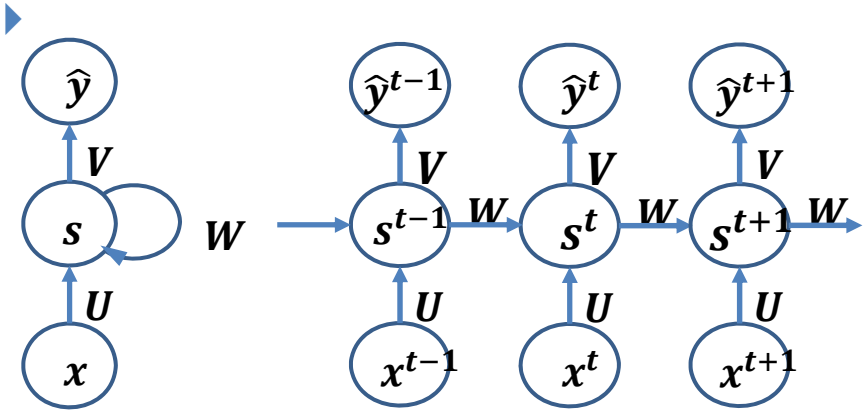
- Several formulations of RNN where proposed in the late 80s, early 90s
  - They faced several limitations and were not successful for applications
    - Recurrent NN are difficult to train
    - They have a limited memory capacity
- Mid 2000s successful attempts to implement RNN
  - e.g. A. Graves for speech and handwriting recognition
  - new models where proposed which alleviate some of these limitations
- Today
  - RNNs are used for a variety of applications e.g., speech decoding, translation, language generation, etc
  - They became SOTA for sequence processing tasks around 2015. In 2020 alternative NN ideas (Transformers) have replaced RNNs for many discrete sequence modeling tasks.
- In this course
  - We briefly survey some of the developments from the 90s
  - We introduce recent developments

Dynamics of RNN

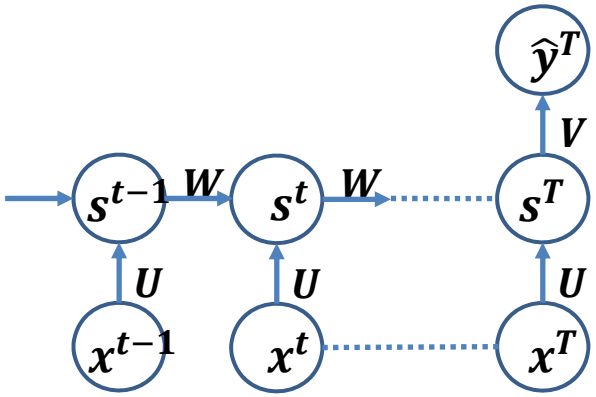
- We consider different tasks corresponding to different dynamics
  - They are illustrated for a simple RNN with loops on the hidden units
  - This can be extended to more complex architectures
  - However, RNNs used today all make use of local connections similar to this simple RNN
- Basic architecture



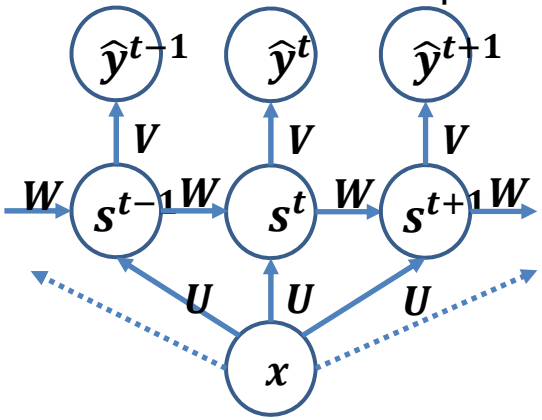
RNNs
Dynamics of RNN – unfolding the RNN



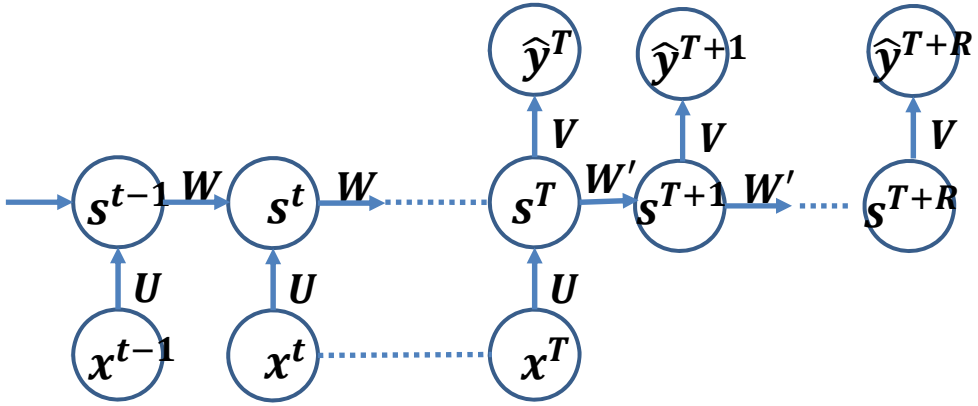
Many to many, e.g. speech or handwriting decoding, Part of Speech Tagging



Many to one, e.g. sequence classification



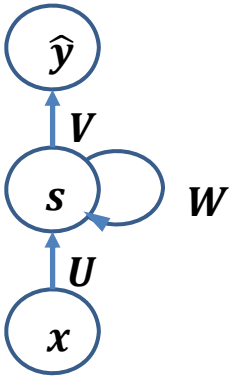
One to many, e.g. image annotation

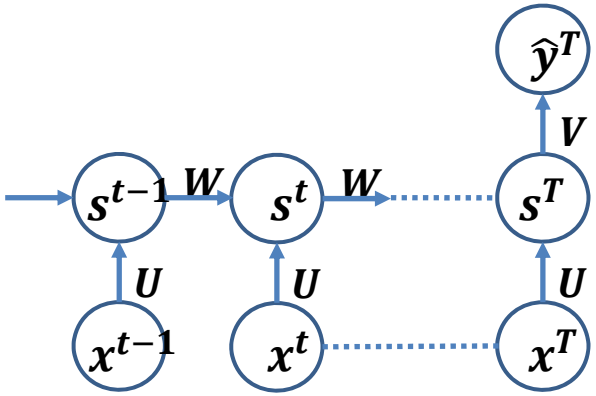


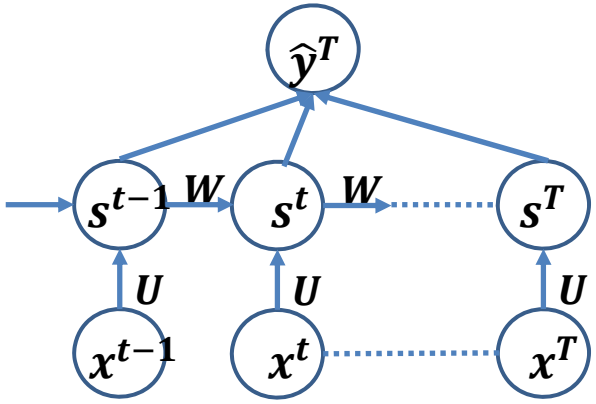
Many to many, e.g. translation

RNNs
Dynamics of RNN – unfolding the RNN

- Different ways to compute sequence encodings







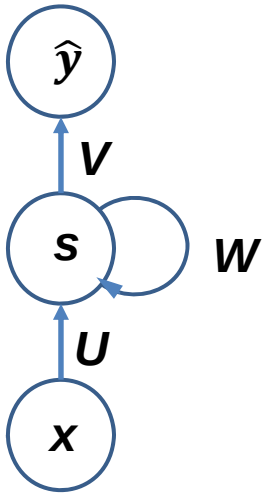
- The final state s^T encodes the sentence

- The whole state sequence encodes the input sequence – usually better: take elementwise max or mean of the hidden states.
- More on that on Attention and Transformers

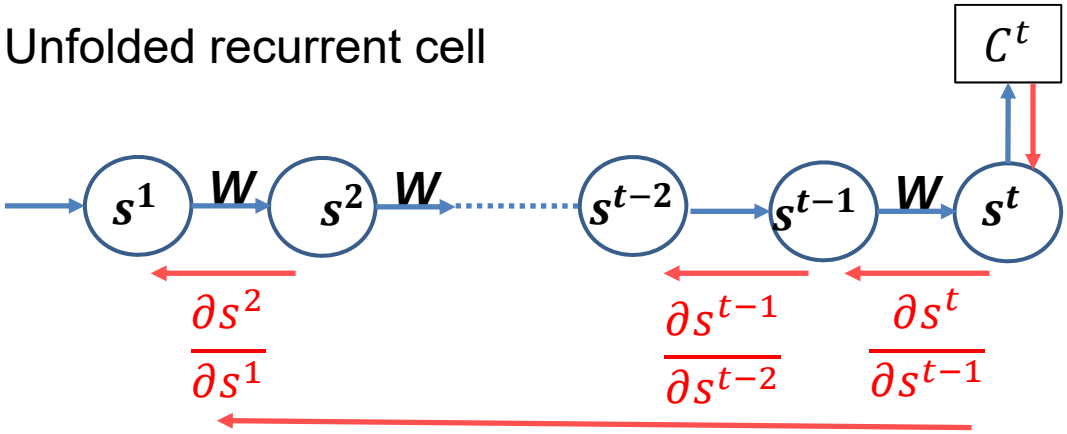
RNNs
Back Propagation Through Time

- 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 NNs
  - 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 w.r.t. size of the sequence
- Several attempts to solve these problems
  - We introduce a popular family of recurrent units that became SOTA around 2015:
    - Gated units (GRU, LSTMs)

- Vanishing gradient problem
  - Consider a many to many mapping problem such as decoding or building a language model (more on that later)



Unfolded recurrent cell



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

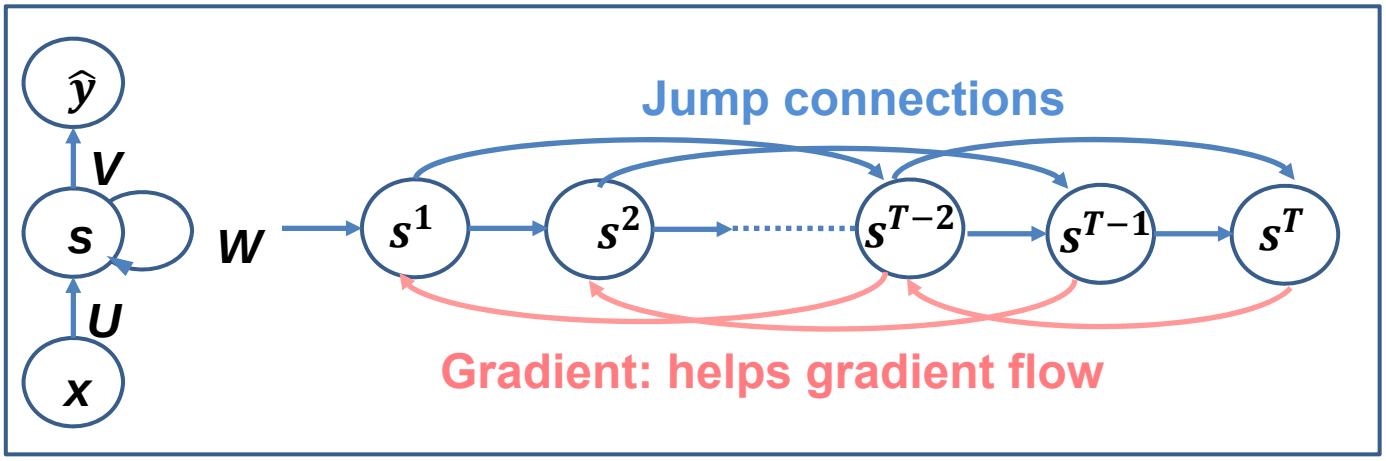
Gradient flow: vanishing gradient

$$\frac{\partial C^t}{\partial s^1} = \frac{\partial s^2}{\partial s^1} \text{x} \dots \text{x} \frac{\partial s^t}{\partial s^{t-1}} \frac{\partial C^t}{\partial s^t}$$

If any of these quantities is small, the gradient from C^t gets smaller and smaller

RNNs - Gated Units
Long Short Term memory (LSTM – Hochreiter 1997)
Gated Recurrent Units (GRU – Cho 2014)

- Introducing « Jump connections » - similar to Resnet



Past value

New candidate value:

$$s^t = (1 - z^t) \odot s^{t-1} + z^t \odot s'^t$$

Gating mechanism

$$s'^t = \tanh(Ux^t + Ws^{t-1})$$
$$z^t = \sigma(U_z x^t + W_z s^{t-1})$$

$\odot$ is the Hadamard product
U_z and W_z learned by SGD

RNNs
Gated Recurrent Units (GRU – Cho 2014)

- Skip connection with Forget Gate + Reset Gate

Past value

New candidate value:

$$\boldsymbol{s}^t = (1 - \boldsymbol{z}^t) \odot \boldsymbol{s}^{t-1} + \boldsymbol{z}^t \odot \boldsymbol{s}'^t$$

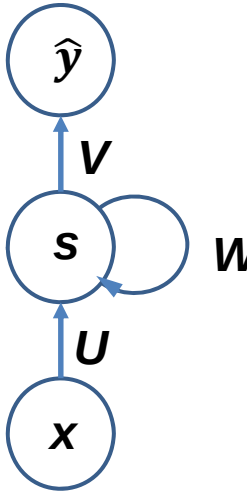
Gating mechanism

$$\boldsymbol{s}'^t = \tanh(U\boldsymbol{x}^t + W(r^t \odot s^{t-1}))$$

Forget gate $\boldsymbol{z}^t = \sigma(U_z\boldsymbol{x}^t + W_z\boldsymbol{s}^{t-1})$

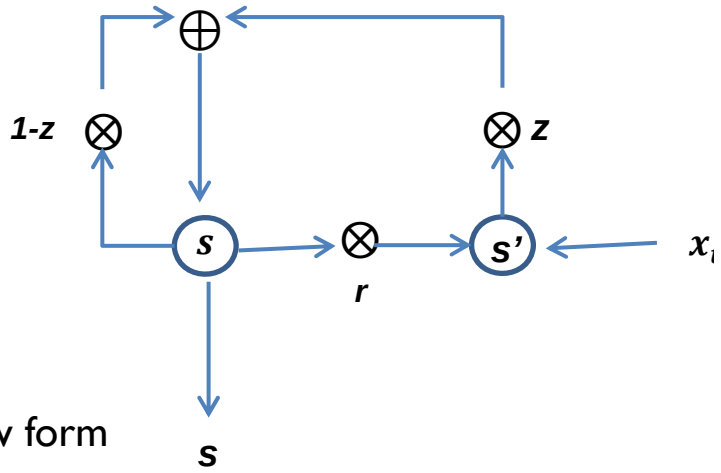
Reset Gate $\boldsymbol{r}^t = \sigma(U_r\boldsymbol{x}^t + W_r\boldsymbol{s}^{t-1})$

$\odot$ is the Hadamard product



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'^t_j
  - $\boldsymbol{s}^t = (1 - \boldsymbol{z}^t) \odot \boldsymbol{s}^{t-1} + \boldsymbol{z}^t \odot \boldsymbol{s}'^t$
  - 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'^t_j
    - 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 $\boldsymbol{s}^{t-1}$
  - $\boldsymbol{z}^t = \sigma(U_z \boldsymbol{x}^t + W_z \boldsymbol{s}^{t-1})$
- The new value $\boldsymbol{s}'^t$ is a classical recurrent unit where the values at time $t-1$ are gated by a reset unit $\boldsymbol{r}_t$
  - $\boldsymbol{s}'^t = \tanh(U\boldsymbol{x}^t + W(r^t \odot s^{t-1}))$
- The reset unit $\boldsymbol{r}^t$ 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)
  - $\boldsymbol{r}^t = \sigma(U_r \boldsymbol{x}^t + W_r \boldsymbol{s}^{t-1})$

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 forgeting 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

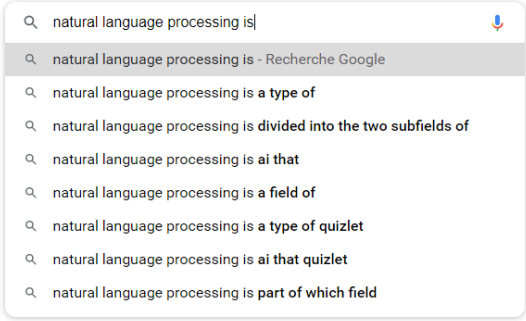
RNNs Future

- RNNs variants (GRU, LSTM) became the dominant approach around 2015, for several tasks including speech recognition, translation, text generation etc
- These last years (2019-2020) they have become superseded by other approaches for many of these tasks
  - Transformers are now more frequently used for a large variety of tasks dealing with discrete sequences, in NLP for example

Language models

- Objective:
  - Probability models of sequences $(x^1, x^2, \ldots, x^t)$
  - Items may be words, characters, character ngrams, word pieces, etc
  - Formally: given a sequence of items, what is the probability of the next item?
    - $p(x^t | x^{t-1}, \ldots, x^1)$
- Example
  - « S'il vous plaît… dessine-moi …» what next ?
  - « $x^1 x^2 x^3$ … … … … … … … x^{t-1} … » what is x^t ?
- Language models in everyday use
  - Sentence completion
    - Search engine queries
    - Smartphone messages, etc
  - Speech recognition, handwriting recognition, etc

Google



Language models

- Language models can be used to compute the probability of a piece of text
- Let $(x^1, x^2, \ldots, x^{\mathrm{T}})$ be a sequence of text, its probability according to a language model is:
  - $p(x^1, x^2, \ldots, x^{\mathrm{T}}) = \prod_{t=1}^{T} p(x^t | x^{t-1}, \ldots, x^1)$
    - With $p(x^t | x^{t-1}, \ldots, x^1)$ computed by the language model

Language models
How to learn a language model - n-grams

- A simple solution: n-grams
  - n-grams are sequences of n consecutive words (or characters, or any items)
  - Language model is based on n-gram statistics
  - Markov assumption
    - x^t only depends on the $n-1$ preceding words
      - $p(x^t|x^{t-1},\dots,x^1)=p(x^t|x^{t-1},\dots,x^{t-n+1})$
    - Use Bayes formula $p(x^t|x^{t-1},\dots,x^{t-n+1}) = \frac{p(x^t,x^{t-1},\dots,x^{t-n+1})}{p(x^{t-1},\dots,x^{t-n+1})}$

n-gram probability

n-1-gram probability

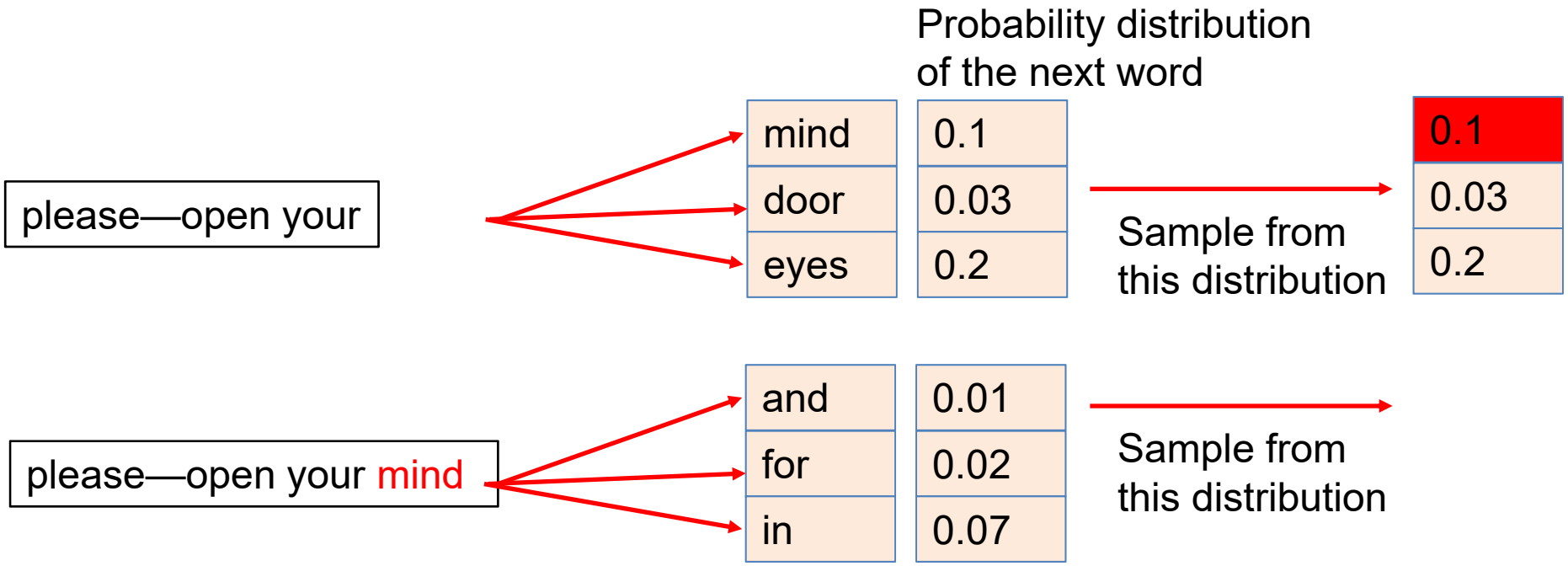
  - Given large text collections, it is possible to compute estimates of the posterior probabilities
    - An estimate could be $\hat{p}(x^t|x^{t-1},\dots,x^{t-n+1}) = \frac{count(x^t,x^{t-1},\dots,x^{t-n+1})}{count(x^{t-1},\dots,x^{t-n+1})}$
    - Where $count(x^t,x^{t-1},\dots,x^{t-n+1})$ is the number 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 dictionnary of 10 k words, there could be
    - 10^{4x2} bigrams
    - 10^{4x3} trigrams, etc
    - Note: the number of n-grams in a language is smaller than 10^{4xn} 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 $n > 10$

Language models
n-grams – text generation

- Any language model can be used for text generation



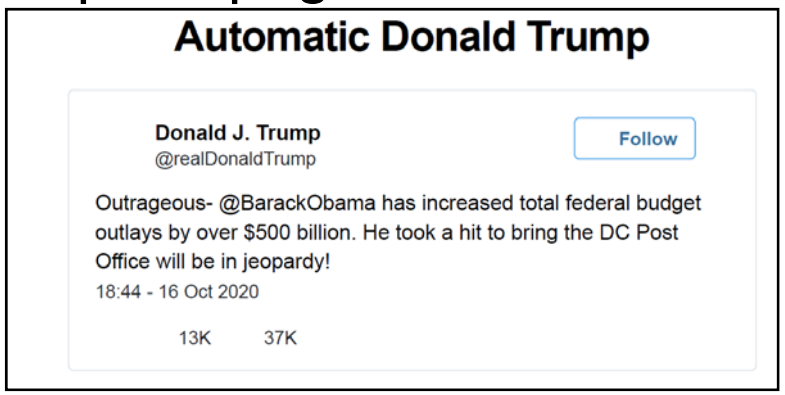
- One can generate text of any length

Language models
n-grams – text generation

- Example from https://projects.haykranen.nl/markov/demo/
  - 4 gram trained on the Wikipedia article on Calvin and Hobbes
  - Generated text

Rosalyn is a standary children used each otherwise as he stereotypically comic stand for an impulsive real-life Watterson's stuffed tiger, much as "grounded in reality rathmore spacious circle: because associety The club has said they have the archive shifting into low art some of the strip was one larger than Calvin articipate indulges in his hands attribute red-and-black pants, magenta socks and Susie Derkins specifically characters like school where were printerestrainstory

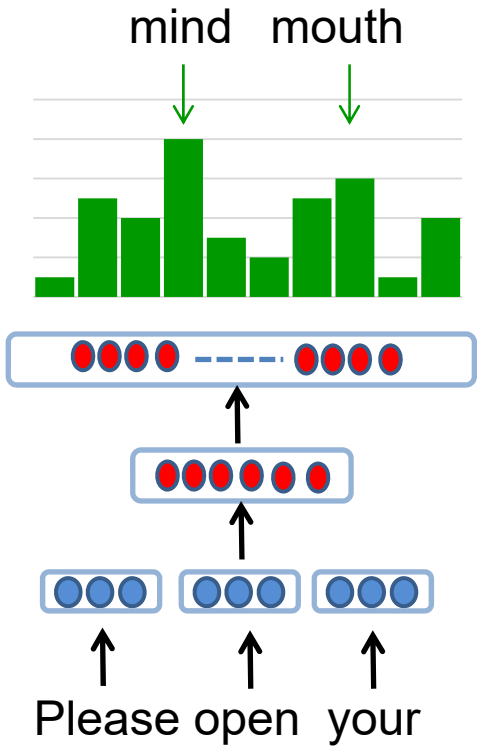
- Example from https://filiph.github.io/markov/



Language models
Neural networks

- Fixed input size NN

- The NN could be typically a convolutional NN with all the input word representations sharing the same weights
- It could also be made fully convolutional
- Less sensitive than n-grams to sparsity



- Posterior estimate of the next word
- Classification layer, softmax among all vocabulary words
- Hidden layer(s)
- Word representation, e.g. w2Vec
- Input sentence, one hot encoding

RNNs
Language models

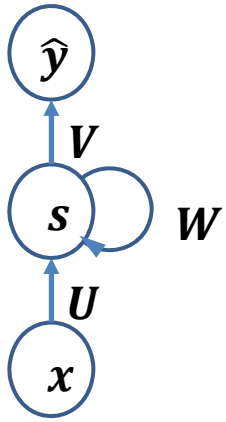
- RNNs offer an alternative approach to non recurrent NNs
- Objective:
  - Probability models of sequences $(x^1, x^2, \ldots, x^t)$
  - Estimate with RNNs:
    - $p(x^t | x^{t-1}, \ldots, x^1)$

prediction

memory

$\hat{\boldsymbol{y}}^t = g(V\boldsymbol{s}^t)$

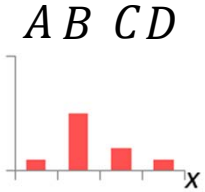
$s^t = f(Ws^{t-1} + Ux^t)$



- g is typically a softmax
- f could be a sigmoid, Relu, …
- x will usually be a word/ item representation learned from large corpora

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 $\hat{\boldsymbol{y}}^t$ for each time step
    - $\hat{\boldsymbol{y}}^t$ is a distribution on the word dictionary
      - This is the estimated posterior probability distribution given past subsequence
      - If the dictionary is $V = \{A, B, C, D\}$:



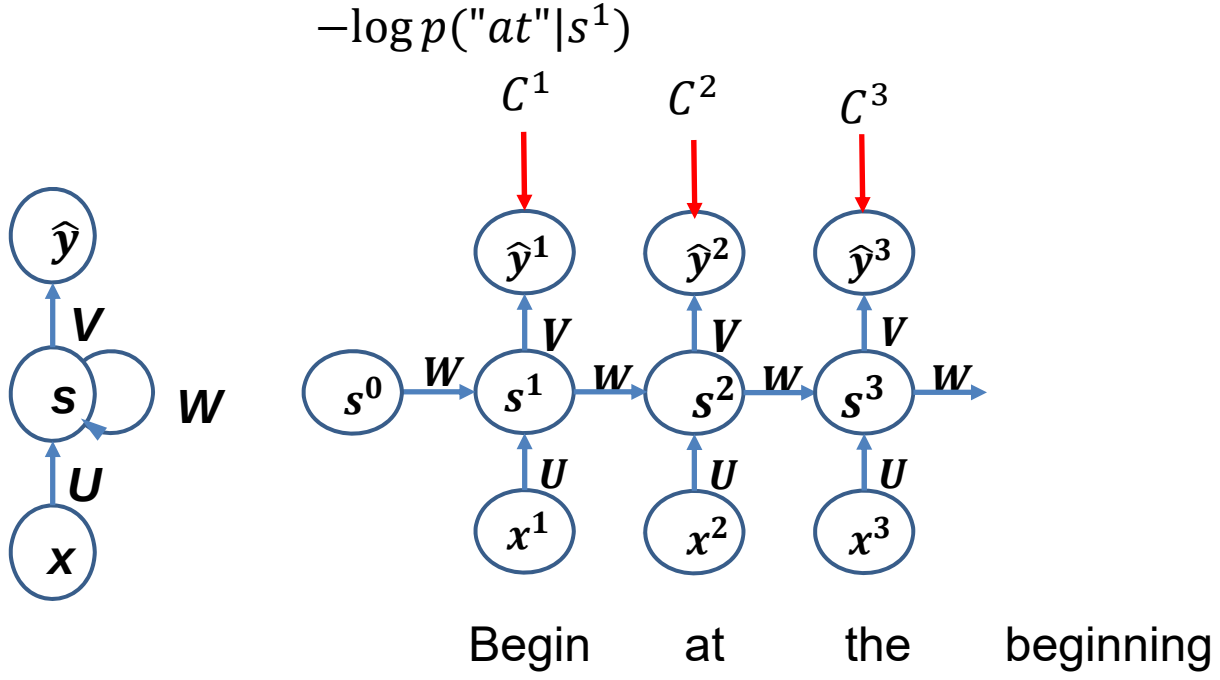
$$\hat{\boldsymbol{y}}^{\boldsymbol{t}} = \boldsymbol{P}(\boldsymbol{x}^{\boldsymbol{t+1}} | \boldsymbol{s}^{\boldsymbol{t}})$$

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

Recurrent neural networks Language models

- Training

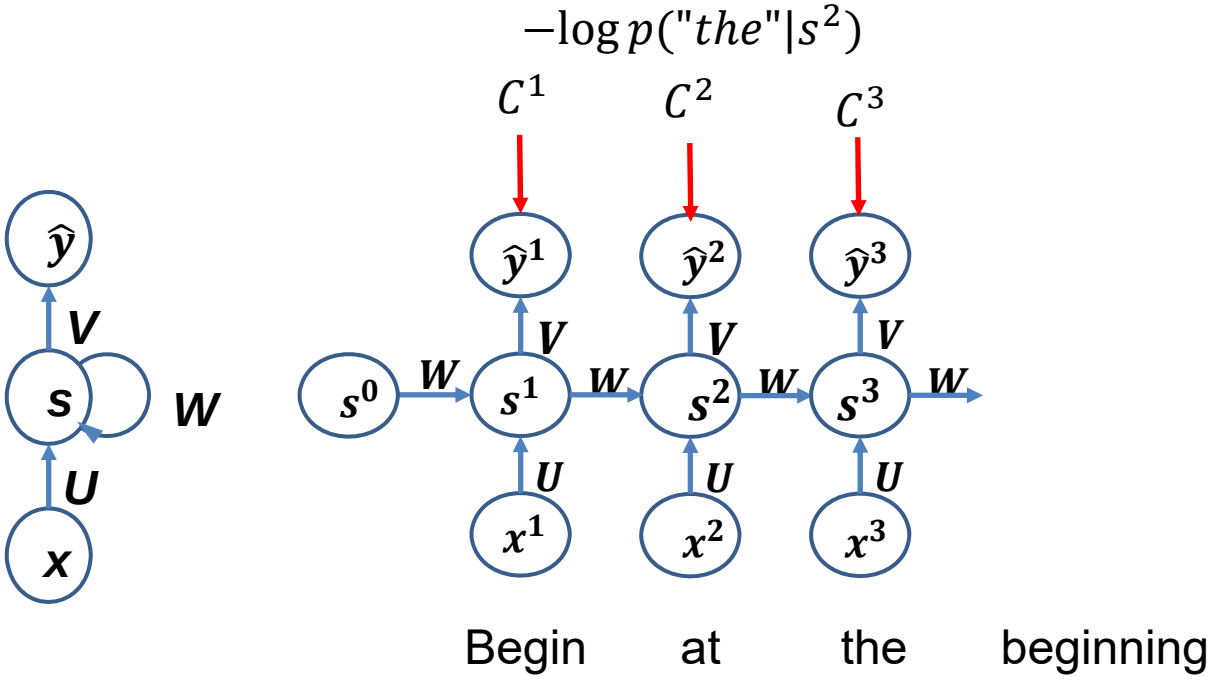
$$\hat{\boldsymbol{y}}^{\boldsymbol{t}} = \boldsymbol{P}(\boldsymbol{x}^{\boldsymbol{t+1}}|\boldsymbol{s}^{\boldsymbol{t}})$$



Recurrent neural networks Language models

- Training

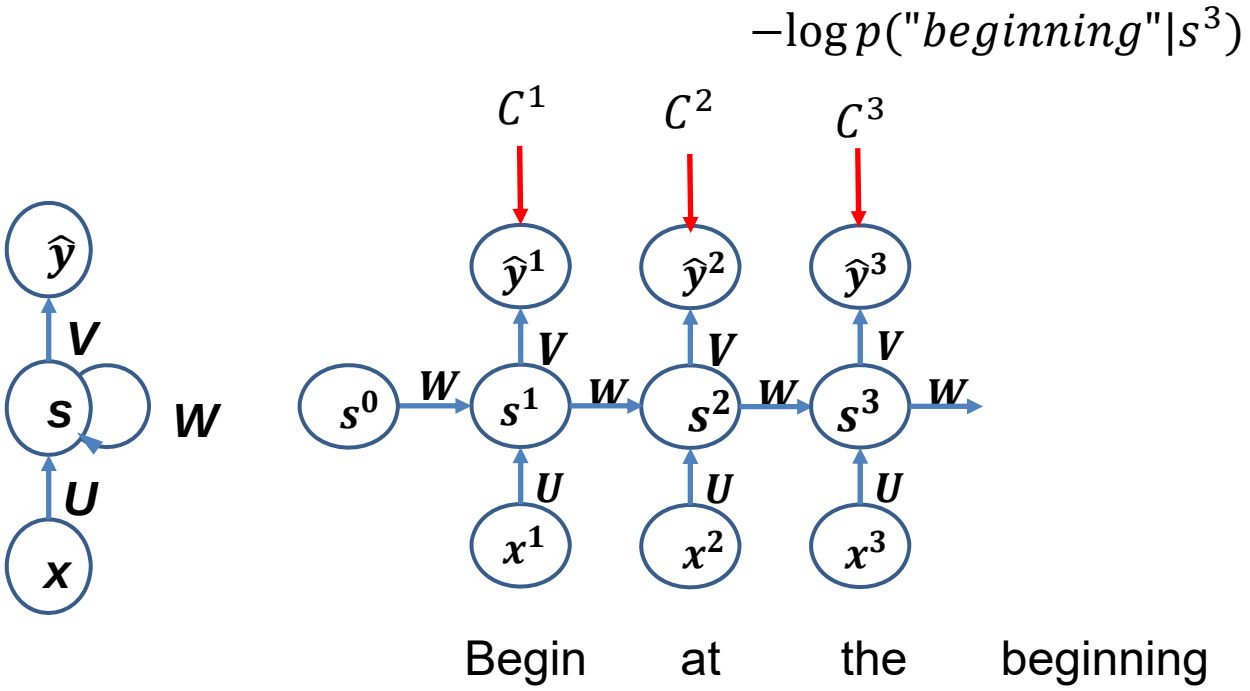
$$\hat{\boldsymbol{y}}^{\boldsymbol{t}} = \boldsymbol{P}(\boldsymbol{x}^{\boldsymbol{t+1}}|\boldsymbol{s}^{\boldsymbol{t}})$$



Recurrent neural networks Language models

- Training

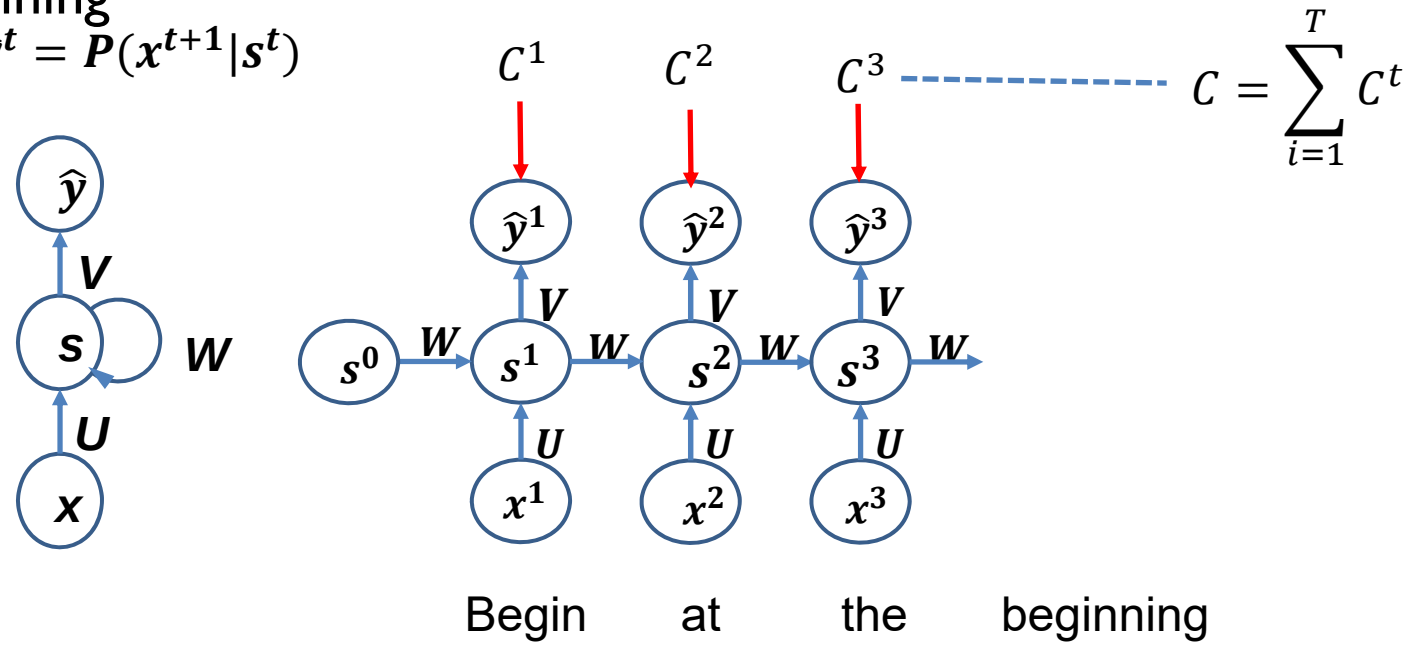
$$\hat{y}^t = P(x^{t+1}|s^t)$$



Recurrent neural networks Language models

- Training

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



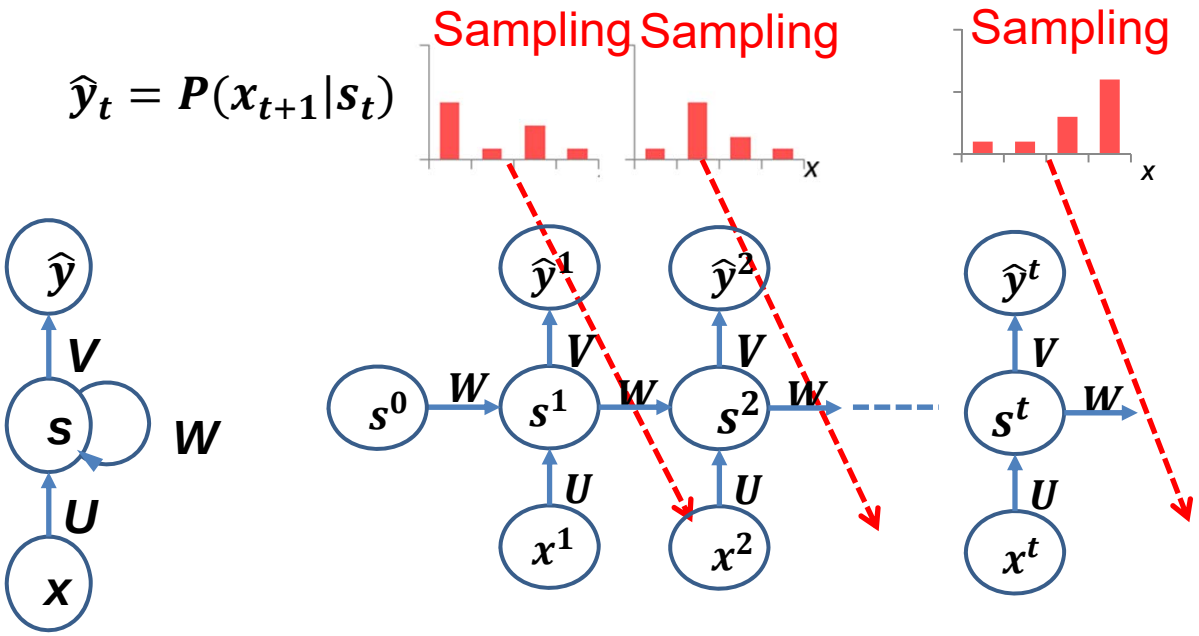
- Note
  - Weights are shared: only one U, one V, one W for the whole NN

Recurrent neural networks Language models

- Training algorithm: Back Propagation Through Time - BPTT
  - Consider a sequence of words $(x^1, x^2, \dots, x^T)$ aka an example from the training set
  - Loss function for a sequence : $C = \sum_{i=1}^{T} C^t$
    - SGD: compute the loss for the sequence (actually a batch of sequences), compute the gradient and upfate the parameters
    - Recall weights are shared: only one U, one V, one W
  - Example: update of the shared W weights
    - Gradient of the loss for the whole sequence: compute the derivatives w.r.t. each C^t and sum them:
      - $\frac{\partial C}{\partial W} = \sum_{t=1\dots T} \frac{\partial C^t}{\partial W}$
    - Gradient of the loss for the loss at time t, C^t:
      - $\frac{\partial C^t}{\partial W} = \sum_{i=1}^{t} \left(\frac{\partial C^t}{\partial W}\right)_{(i)}$ where $\left(\frac{\partial C^t}{\partial W}\right)_{(i)}$ is the gradient of the loss w.r.t. weight at position $i \leq t$
        - Backpropagate over time steps $i = 1 \dots t$, summing the gradient: BPTT
  - This training regime is called teacher forcing
    - Successive sequential inputs correspond to the true sequence
    - Different during inference (see next slide)

RNNs
Language models

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



RNNs
Language models – Word representation

- Words, characters, n-grams, word pieces 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 embeding technique (e.g. Word2Vec, see later)
      - Advantage, this can be done by using very large text collections
      - These representations could then be used for downstream tasks (e.g. classification)
    - On line while training the language model
      - In this case, the xs are initialized at random values in R^n and are learned by backpropagating the error, together with the other parameters
      - We usually loose the benefit of training on large corpora

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 discribed 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

Semantics: words
How to encode words according to their semantic meaning

- **Representing words as discrete symbols**
  - In traditional NLP, we regard words as discrete symbols: Words can be represented by **one-hot vectors** - Each word is a distinct symbol
  - **Example**: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".
    - motel = [0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 0]
    - hotel = [0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0]
      - These two vectors are orthogonal.
      - There is **no natural notion of similarity** for one-hot vectors!
  - **Vector dimension** = number of words in vocabulary (e.g., 500,000)
    - Very large dimensional discrete space - Problem for machine learning - sparsity

Semantics: words

- Instead: learn to encode similarity in the vectors themselves
  - GloVe (Pennington et al. 2014)

Nearest words to frog:

1. frogs
2. toad
3. litoria
4. leptodactylidae
5. rana
6. lizard
7. eleutherodactylus

litoria

leptodactylidae

rana

eleutherodactylus

Words in vector space
Representing words by their context

- Distributional semantics: A word's meaning is given by the words that frequently appear close-by
  - One of the most successful ideas of modern statistical NLP!
- When a word w appears in a text, its context is the set of words that appear nearby (within a fixed-size window).
  - Use the many contexts of w to build up a representation of w

…government debt problems turning into **banking** crises as happened in 2009…
…saying that Europe needs unified **banking** regulation to replace the hodgepodge…
…India has just given its **banking** system a shot in the arm…

context words will represent ***banking***

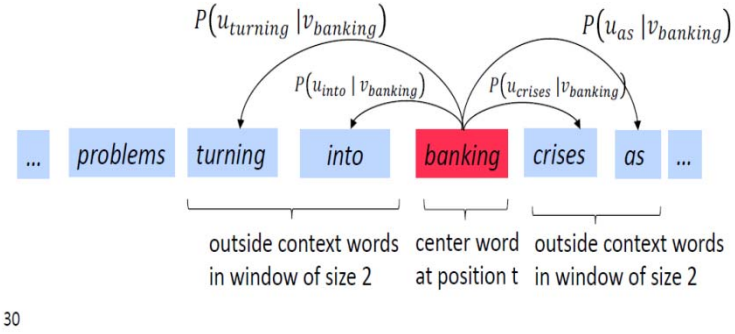
Words in vector space
Representing words by their context

- Word embeddings
  - We represent words by vectors so that words with similar contexts share « close » representations in the vector space

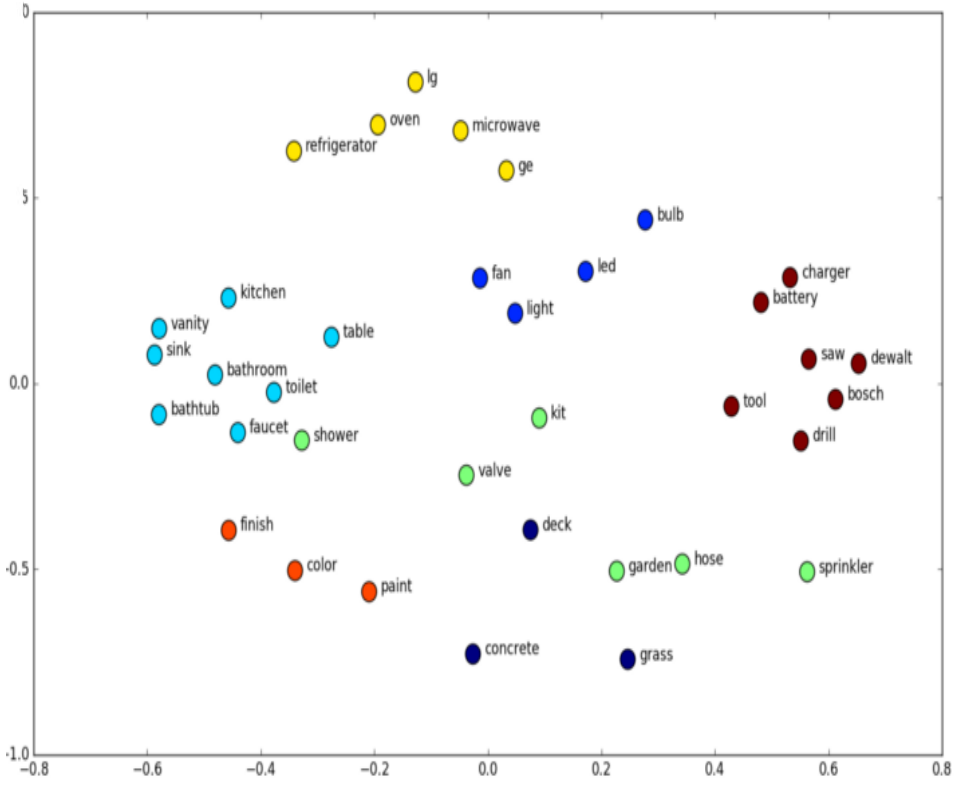
$$\boldsymbol{banking} = \begin{bmatrix} 0.87 \\ 0.45 \\ -0.34 \\ -0.63 \\ 0.23 \\ 0.16 \end{bmatrix}$$

- Key idea
  - These representations are learned from very large corpora for representing a large variety of situations/ contexts
    - No nead for supervision
  - These embeddings will be used for doswnstream tasks, e.g. classification

Word embeddings
Word2Vec – Mikolov et al. 2013

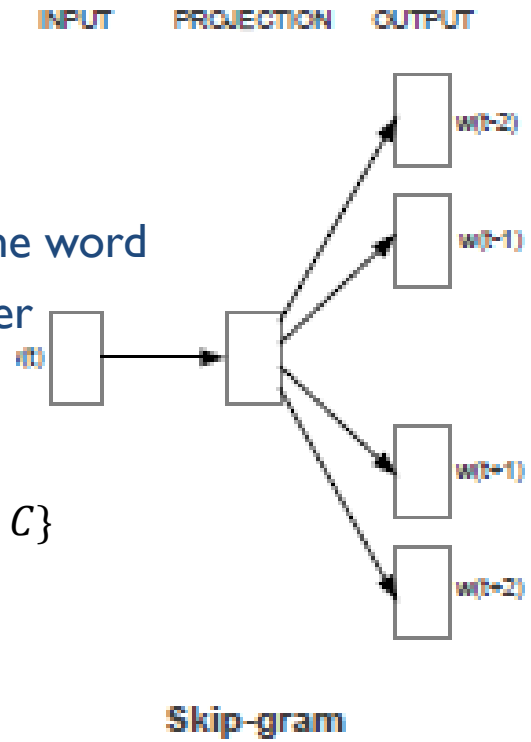


$$p(o|c) = \frac{\exp(u_o . v_c)}{\sum_{w \in Vocabulary} \exp(u_w . v_c)}$$



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

- Skip Gram model
  - Task
    - Given a sequence of words, predict context words from the central word
    - The context is typically 4 words before and 4 after
  - Input and output word representations are learned jointly
    - (random initialization)
  - The projection layer is linear followed by a sigmoid
  - 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



Skip-gram

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
  - $p(w_{out}|w_{in}) = \frac{\exp(\boldsymbol{v}_{w_{out}}.\boldsymbol{v}_{w_{in}})}{\sum_{w=1}^{V}\exp(\boldsymbol{v}_{w}.\boldsymbol{v}_{w_{in}})}$
    - Where $\boldsymbol{v}_w$ is the learned representation of the w vector (the hidden layer), $\boldsymbol{v}_{w_{out}}.\boldsymbol{v}_{w_{in}}$ 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
        - $l(w_{in}, w_{Out}) = \log\sigma\left(v_{w_{out}}.v_{w_{in}}\right) + \sum_{i=1}^{k}\log\sigma(-v_{w_i}.v_{w_{in}}))$
        - with $\sigma(x) = \frac{1}{1+\exp(-x)}$

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 $p(w_{out}|w_{in}) = \frac{\exp(\boldsymbol{v}_{w_{out}}.\boldsymbol{v}_{w_{in}})}{\sum_{w=1}^{V}\exp(\boldsymbol{v}_{w}.\boldsymbol{v}_{w_{in}})}$ in W2V
    - $\boldsymbol{v}_{w_{out}}.\boldsymbol{v}_{w_{in}}$ is replaced by $\sum_{z_i \in\ ngram(w_{in})} \boldsymbol{v}_{w_{out}}.\ z_i$
    - And the same for $\boldsymbol{v}_{w}.\boldsymbol{v}_{w_{in}}$

Generative Adversarial Networks - GANs

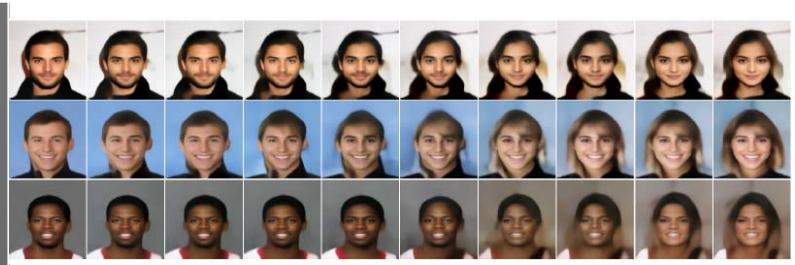
Ian J. Goodfellow, et al. 2014

Generative models

- Objective
  - Learn a probability distribution model from data samples
    - Given $x^1, \ldots, x^N \in R^n$ learn to approximate their underlying distribution $\mathcal{X}$
    - For complex distributions, there is no analytical form, and for large size spaces (R^n) approximate methods (e.g. MCMC) might fail
    - Deep generative models recently attacked this problem with the objective of handling large dimensions and complex distributions

https://en.wikipedia.org/wiki/Edmond_de_Belamy
432 k$ Christies in 2018

Xie et al. 2019
artificial smoke



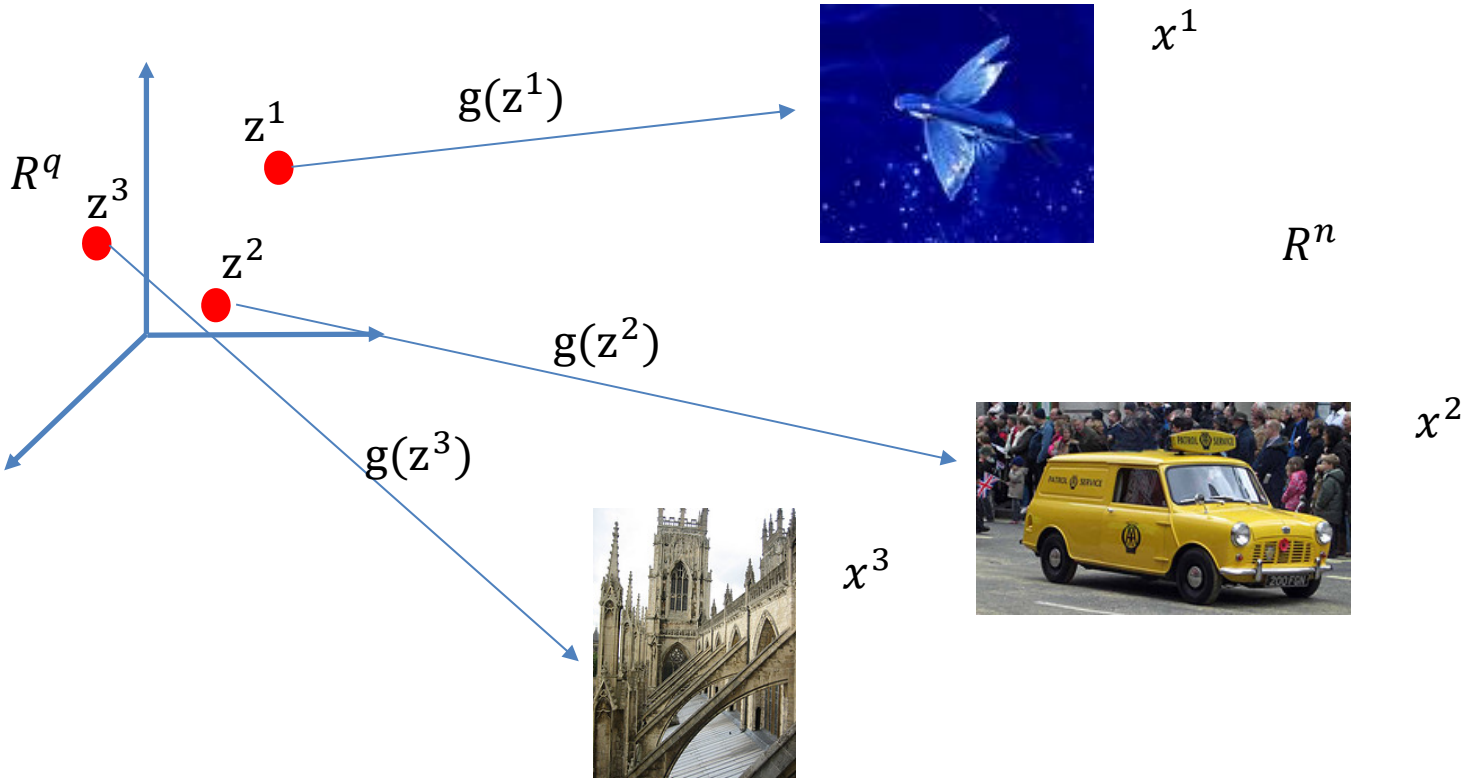
De Bezenac et al. 2021
Generating female images from male ones

Generative models

- Objective
  - General setup of deep generative models
    - Learn a generator nework $g_\theta: R^q \rightarrow R^n$ that transforms a latent distribution $\mathcal{Z} \subset R^q$ to match a target distribution $\mathcal{X}$
      - $\mathcal{Z}$ is usually a simple distribution e.g. Gaussian from which it is easy to sample, $q < n$
      - This is unlike traditional statistics where an analytic expression for the distribution is sought
    - Once trained the generator can be used for:
      - Sampling from the latent space:
        - $z \in R^q \sim \mathcal{Z}$ and then generate synthetic data via $g_\theta(.), g_\theta(z) \in R^n$
      - When possible, density estimation $p_\theta(x) = \int \; p_\theta(x|z)p_Z(z)dz$
        - with $p_\theta(x|z)$ a function of g_θ

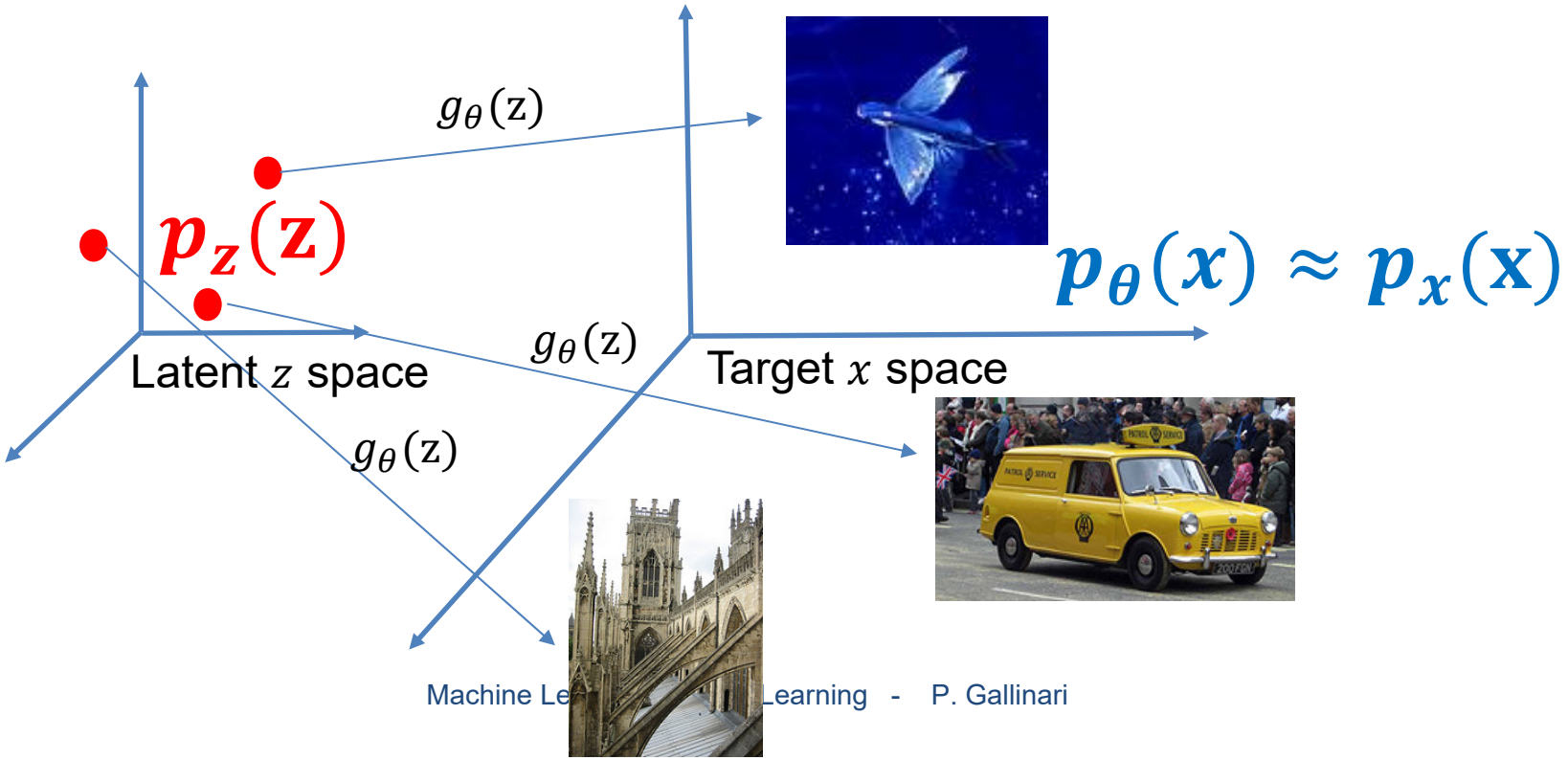
Generative models intuition

- Let $\{z^1, \dots, z^N\}, z^i \in R^q$ and $\{x^1, \dots, x^N\}, x^i \in R^n$, two sets of points in different spaces
  - Provided a sufficiently powerful model $g(x)$, it should be possible to learn complex deterministic mappings associating the two sets:



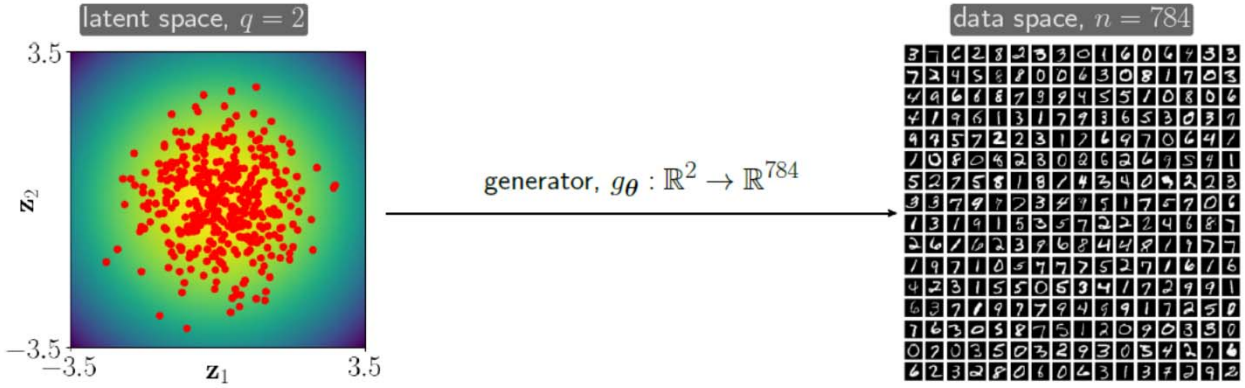
Generative models intuition

- Given distributions on a latent space $p_z(\mathrm{z})$, and on the data space $p_x(x)$, it is possible to map $p_z(\mathrm{z})$ onto $p_x(x)$
  - g_θ defines a distribution on the target space $p_x(g_\theta(z)) = p_\theta(x)$
    - $p_\theta(x)$ is the generated data distribution, objective: $p_\theta(x) \approx p_x(x)$
  - Data generation: sample $\mathrm{z} \sim Z$, transform with g_θ, $g_\theta(z)$

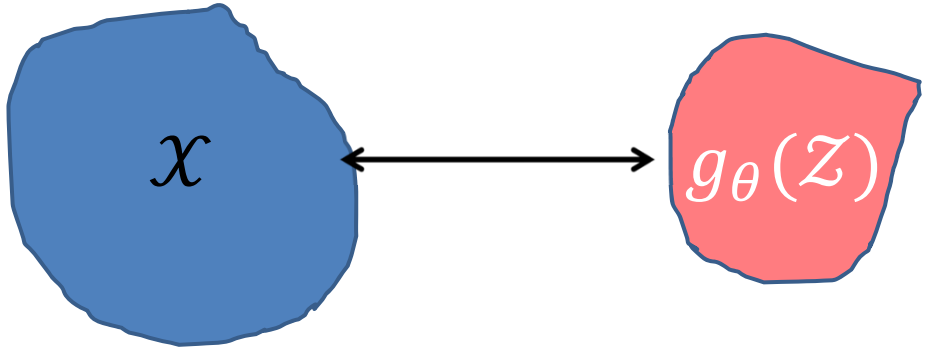


Generative models intuition

- Data generation: sample z~Z, transform with g_θ, $g_\theta(z)$

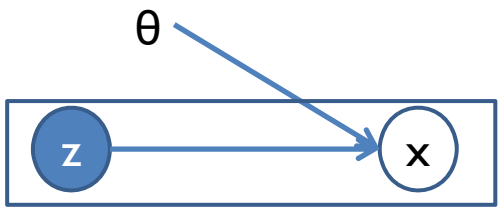


- Important issue
  - How to compare predicted distribution $p_\theta(x)$ and target distribution $p_X(x)$?



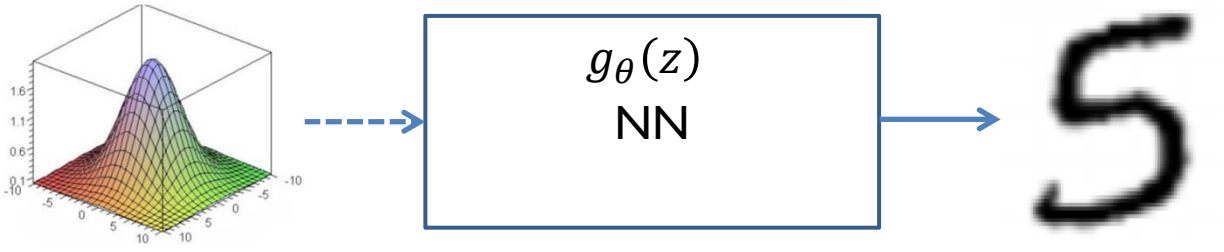
GANs

- Generative latent variable model



  - Given Samples $x^1, \dots, x^N \in R^n$, with $x \sim \mathcal{X}$, latent space distribution $z \sim \mathcal{Z}$ e.g $z \sim \mathcal{N}(0, \mathrm{I})$, use a NN to learn a possibly complex mapping $g_\theta : R^q \rightarrow R^n$ such that:

$$p_x(g_\theta(z)) = p_\theta(x) \approx p_x(x)$$



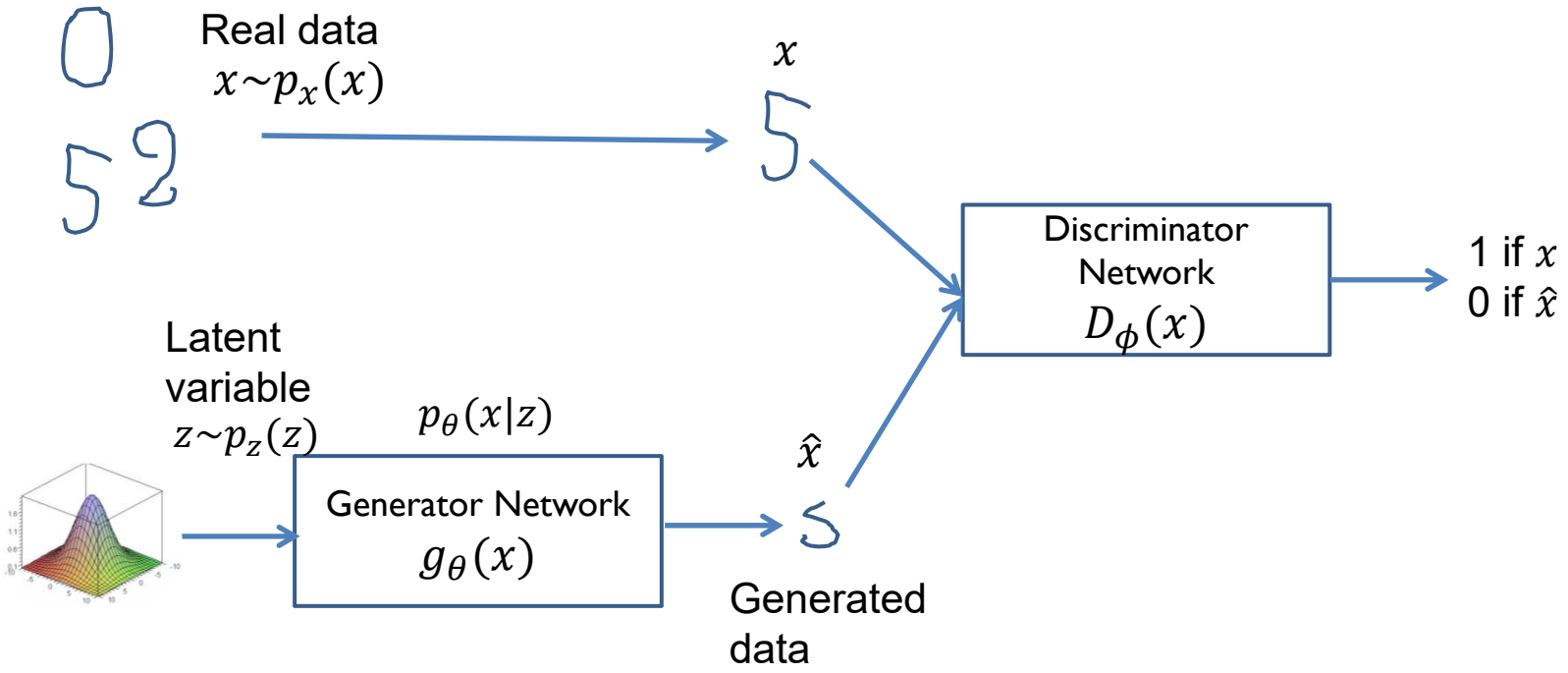
- Different solutions for measuring the similarity between $p_\theta(x)$ and $p_x(x)$
  - In this course: binary classification
- Note:
  - Once trained, sample from z directly generates the samples $g_\theta(z)$
  - Different from VAEs and Flows where the NN $g_\theta(.)$ generate distribution parameters

GANs – Adversarial training as binary classification

- Principle
  - A **generative** network generates data after sampling from a latent distribution
  - A **discriminant** network tells if the data comes from the generative network or from real samples
    - The discriminator will be used to measure the distance between the distributions $p_\theta(x)$ and $p_x(x)$
  - The two networks are trained together
    - The generative network tries to fool the discriminator, while the discriminator tries to distinguish between true and artificially generated data
    - The problem is formulated as a MinMax game
    - The Discriminator will force the Generator to be « clever » and learn the data distribution
- Note
  - No hypothesis on the existence of a density function
    - i.e. no density estimate (Flows), no lower bound (VAEs)

GANs – Adversarial training as binary classification
Intuition - Training

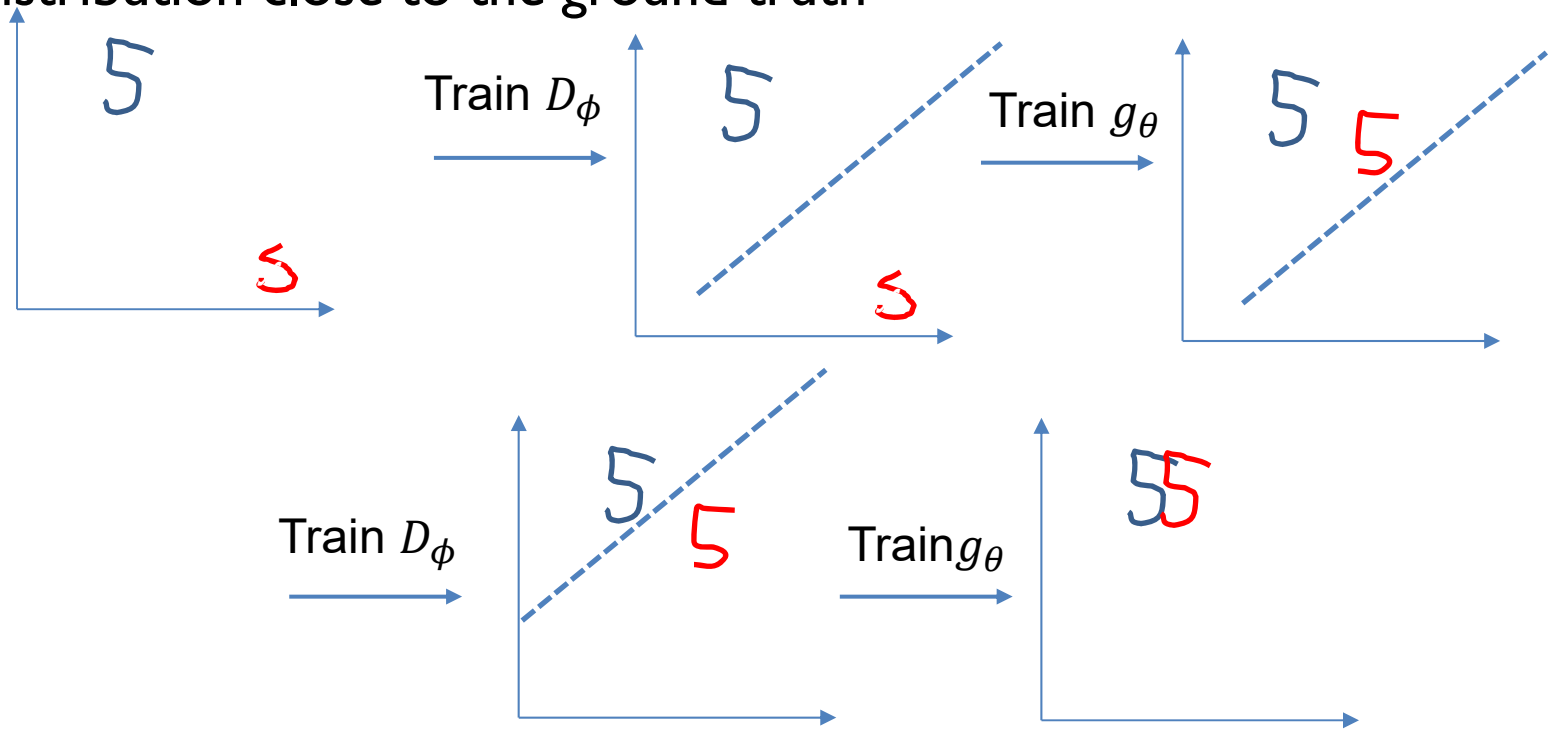
- Discriminator is presented alternatively with true (x) and fake ($\hat{x} = g_\theta(z)$)data



D_ϕ and g_θ are typically MLPs/Deep CNNs/…

GAN – Adversarial training as binary classification
Intuition - Training

- Algorithm alternates between optimizing D_ϕ (separate true and generated data) and g_θ (generate data as close as possible to true examples) – Once trained, G should be able to generate data witha distribution close to the ground truth



GANs - Adversarial training as binary classification
Loss function (Goodfellow et al. 2014)

- $x \sim p_x(x)$ distribution over data x
- $z \sim p_z(z)$ prior on z, usually a simple distribution (e.g. Normal distribution)
- Loss
  - $\min_{\theta} \max_{\phi} L(D_\phi, g_\theta) = E_{x \sim p_x(x)}[log D_\phi(x)] + E_{z \sim p_z(z)}[\log\left(1 - D_\phi(g_\theta(z))\right)]$
    - $g_\theta: R^q \rightarrow R^n$ mapping from the latent (z) space to the data (x) space
    - $D_\phi: R^n \rightarrow [0,1]$ probability that x comes from the data rather than from the generator g_θ
    - If g_θ is fixed, $L(D_\phi, g_\theta)$ is a classical binary cross entropy for D_ϕ, distinguishing real and fake examples
  - Note:
    - Training is equivalent to find D_{ϕ^*}, g_{θ^*} such that
      - $D_{\phi^*} \in arg \max_{\phi} L(D_\phi, g_{\theta^*})$ and $g_{\theta^*} \in arg \min_{\theta} L(D_{\phi^*}, g_\theta)$
      - Saddle point problem
        - instability
- Practical training algorithm
  - Alternates optimizing (maximizing) w.r.t. D_ϕ optimizing (minimizing) w.r.t. g_θ

GAN- Adversarial training as binary classification Equilibrium analysis (Goodfellow et al. 2014)

- The seminal GAN paper provides an analysis of the solution that could be obtained at equilibrium
- Let us define
  - $L(D_\phi, g_\theta) = E_{x\sim p_x(x)}[logD_\phi(x)] + E_{\mathrm{x}\sim p_\theta(x)}[\log\left(1 - D_\phi(x)\right)]$
    - with $p_x(x)$ the true data distribution and $p_\theta(x)$ the distribution of generated data
      - Note that this is equivalent to the $L(D, G)$ definition on the slide before
- If g_θ and D_ϕ have sufficient capacity
  - Computing $\underset{\theta}{argmin}$
    - $g^* = \underset{\theta}{argmin} \max_\phi L\left(D_\phi, g_\theta\right)$
  - Is equivalent to compute
    - $g^* = argmin_\theta D_{JS}(p_x, p_\theta)$ with $D_{JS}(,)$ the Jenson-Shannon dissimilarity measure between distributions
    - The loss function of a GAN quantifies the similarity between the real sample distribution and the generative data distribution by JSD when the discriminator is optimal
  - If the optimum is reached
    - $D(x) = \frac{1}{2}$ for all $x \rightarrow$ Equilibrium

GAN equilibrium analysis (Goodfellow et al. 2014)
Prerequisite KL divergence

- Kullback Leibler divergence
  - Measure of the difference between two distributions p and q
  - Continuous variables
    - $D_{KL}(p(y)||q(y)) = \int_y (\log \frac{p(y)}{q(y)}) p(y) dy$
  - Discrete variables
    - $D_{KL}(p(y)||q(y)) = \sum_i (\log \frac{p(y_i)}{q(y_i)}) p(y_i)$
- Property
  - $D_{KL}(p(y)||q(y)) \geq 0$
  - $D_{KL}(p(y)||q(y)) = 0$ iff $p = q$
    - $D_{KL}(p(y)||q(y)) = -E_{p(y)}\left[log \frac{q(y)}{p(y)}\right] \geq -\log E_{p(y)}\left[\frac{q(y)}{p(y)}\right] \geq 0$
      - where the first inequality is obtained via Jensen inequality
  - note: D_{KL} is asymmetric, symmetric versions exist, e.g. Jensen-Shannon divergence

GAN equilibrium analysis (Goodfellow et al. 2014) - proof

- For a given generator G, the optimal discriminator is
  - $\mathrm{D}^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g\,(x)}$
    - Let $f(y) = a\,log(y) + b\,log(1-y)$, with $a, b, y > 0$
    - $\frac{df}{dy} = \frac{a}{y} - \frac{b}{1-y}, \frac{df}{dy} = 0 \Leftrightarrow y = \frac{a}{a+b}$ and this is a max
    - $Max_D\, L(D,G) = E_{x\sim p_{data}(x)}[logD(x)] + E_{\mathrm{x}\sim p_g(x)}[\log(1-D(x))]$ is then obtained for:
      - $\mathrm{D}^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g\,(x)}$

GAN equilibrium analysis (Goodfellow et al. 2014) - proof

- Let $C(G) = \max_D L(G, D) = L(G, D^*)$
- It si easily verfied that:
  - $C(G) = -\log 4 + KL\left(p_{data}(x); \frac{p_{data}(x)+p_g(x)}{2}\right) + KL\left(p_g(x); \frac{p_{data}(x)+p_g(x)}{2}\right)$
  - Since $KL(p; q) \geq 0$ and $KL(p; q) = 0$ iff $p = q$
    - $C(G)$ is minimum for $p_g = p_{data}$ with $D^*(x) = \frac{1}{2}$
    - At equilibrium, GAN training optimises Jenson-Shannon Divergence, $JSD(p; q) = \frac{1}{2} KL\left(p; \frac{p+q}{2}\right) + \frac{1}{2} KL\left(q; \frac{p+q}{2}\right)$ between p_g and p_{data}
- Summary
  - The loss function of a GAN quantifies the similarity between the real sample distribution and the generative data distribution by JSD when the discriminator is optimal
- Note
  - $\frac{p_{data}(x)}{p_g(x)} = \frac{p(x|y=1)}{p(x|y=0)} = k \frac{p(y=1|x)}{p(y=0|x)} = k \frac{D^*(x)}{1-D^*(x)}$ with $k = \frac{p(y=0)}{p(y=1)}$
  - The discriminator is used to implicitely measure the discrepancy between the distributions

Adversarial training as binary classification
Training GANs

- Training alternates optimization (SGD) on D_ϕ and g_θ
  - In the alternating scheme, g_θ usually requires more steps than D_ϕ+ different batch sizes
- It is known to be highly unstable with two pathological problems
  - Oscillation: no convergence
  - Mode collapse: G collapses on a few modes only of the target distribution (produces the same few patterns for all z samplings)
  - Low dimensional supports (Arjovsky 2017): $p_x$and p_θ may lie on low dimensional manifold that do not intersect.
    - It is then easy to find a discriminator, without p_θ close to p_x
  - Lots of heuristics, lots of theory, but
    - Behavior is still largely unexplained, best practice is based on heuristics

GANs examples
Deep Convolutional GANs (Radford 2015 - historical example) - Image generation

- LSUN bedrooms dataset - over 3 million training examples –

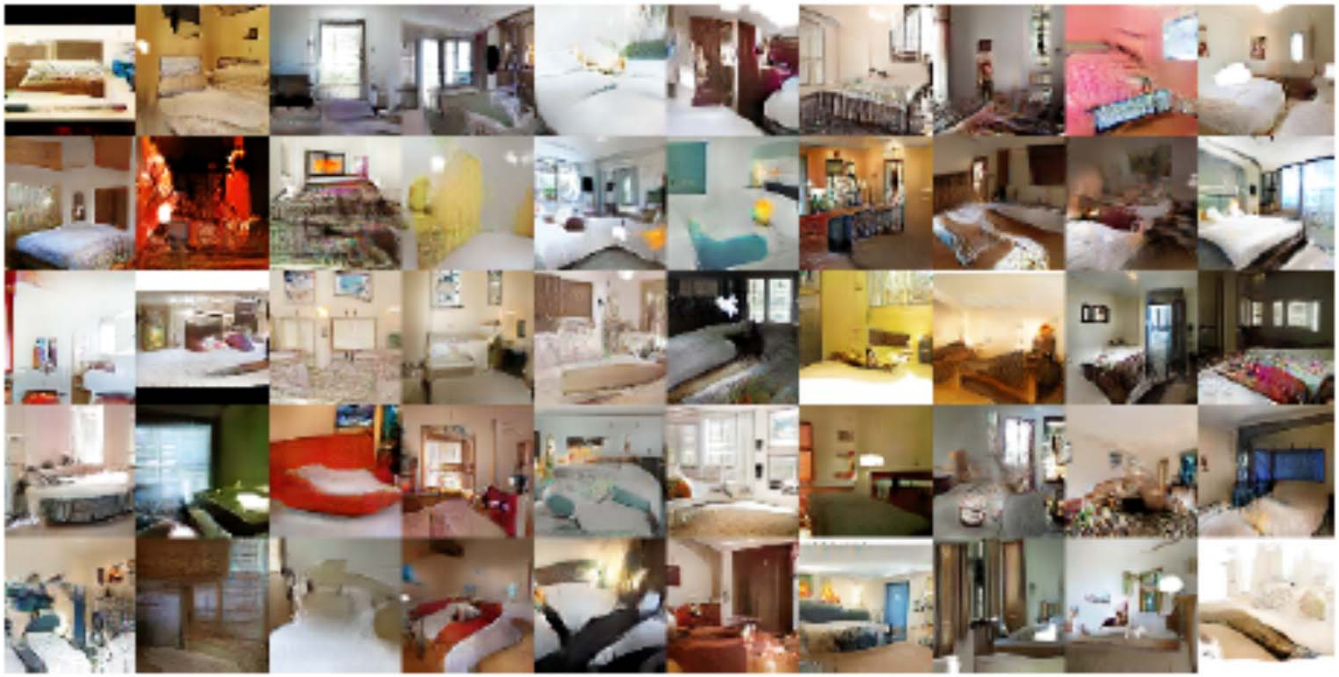
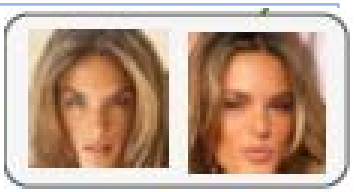


Figure 3: Generated bedrooms after five epochs of training. There appears to be evidence of visual under-fitting via repeated noise textures across multiple samples such as the base boards of some of the beds.

Fig. Radford 2015

Gan example
MULTI-VIEW DATA GENERATION WITHOUT VIEW SUPERVISION (Chen 2018)



- Objective
  - Generate images by disantangling content and view
    - e.g. content 1 person, View: position, illumination, etc
  - 2 latent spaces: view and content
    - Generate image pairs: same item with 2 different views
    - Learn to discriminate between generated and real pairs

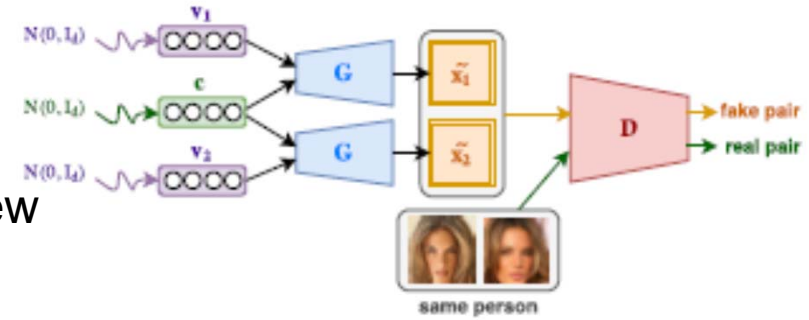
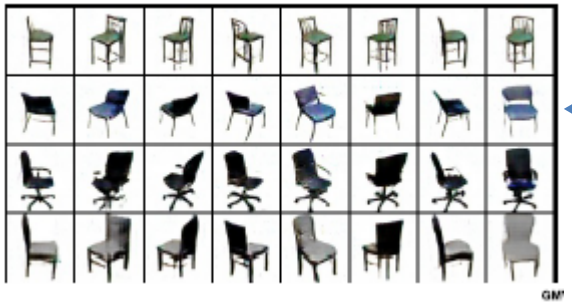


Fig. Chen 2018

Column = view

Column = view



1 row = 1 content

Conditional GANs (Mirza 2014)

- The initial GAN models distributions by sampling from the latent Z space
- Many applications require to condition the generation on some data
  - e.g.: text generation from images, in-painting, super-resolution, etc
- Conditional GANs
  - Both the generator and the discriminator are conditioned on variable y – corresponding to the conditioning data

$$\min_{\theta}\max_{\phi} V(D,g) = E_{x\sim p_x(x)}[logD_{\phi}(x|y)] + E_{z\sim p(z)}[\log\Big(1 - D_{\phi}(g_{\theta}(z|y))\Big)]$$

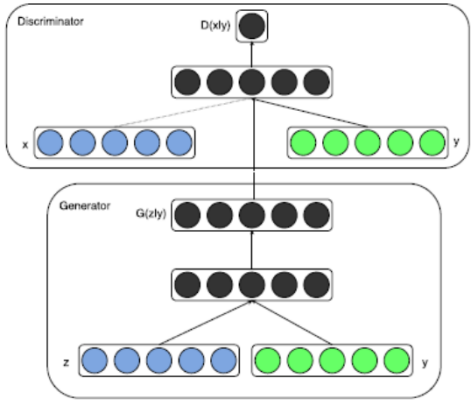


Fig. (Mirza 2014)

Conditional GANs example
Generating images from text (Reed 2016)

- Objective
  - Generate images from text caption
  - Model: GAN conditioned on text input
- Compare different GAN variants on image generation
- Image size 64x64

Fig. from Reed 2016

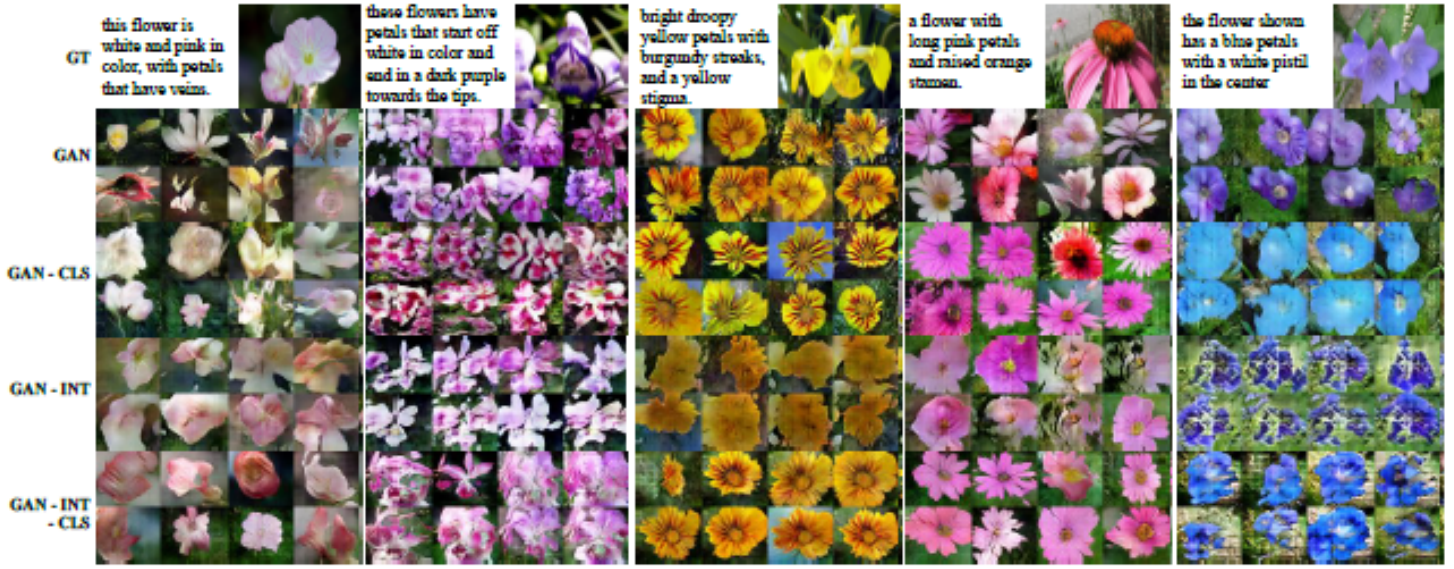


Figure 4. Zero-shot generated flower images using GAN, GAN-CLS, GAN-INT and GAN-INT-CLS. All variants generated plausible images. Although some shapes of test categories were not seen during training (e.g. columns 3 and 4), the color information is preserved.

Cycle GANs (Zhu 2017)

- Objective
  - Learn to « translate » images without aligned corpora
    - 2 corpora available with input and output samples, but no pair alignment between images
  - Given two unaligned corpora, a conditional GAN can learn a correspondance between the two distributions (by sampling the two distributions), however this does not guaranty a correspondance between input and output
- Approach
  - (Zhu 2017) proposed to add a « consistency » constraint similar to back translation in language
    - This idea has been already used for vision tasks in different contexts
    - Learn two mappings
      - $G: X \rightarrow Y$ and F: $Y \rightarrow X$ such that:
      - $F \circ G(x) \simeq x$ and G $\circ F(y) \simeq$y
      - and two discriminant functions $D_Y$and D_X

Cycle GANs (Zhu 2017)

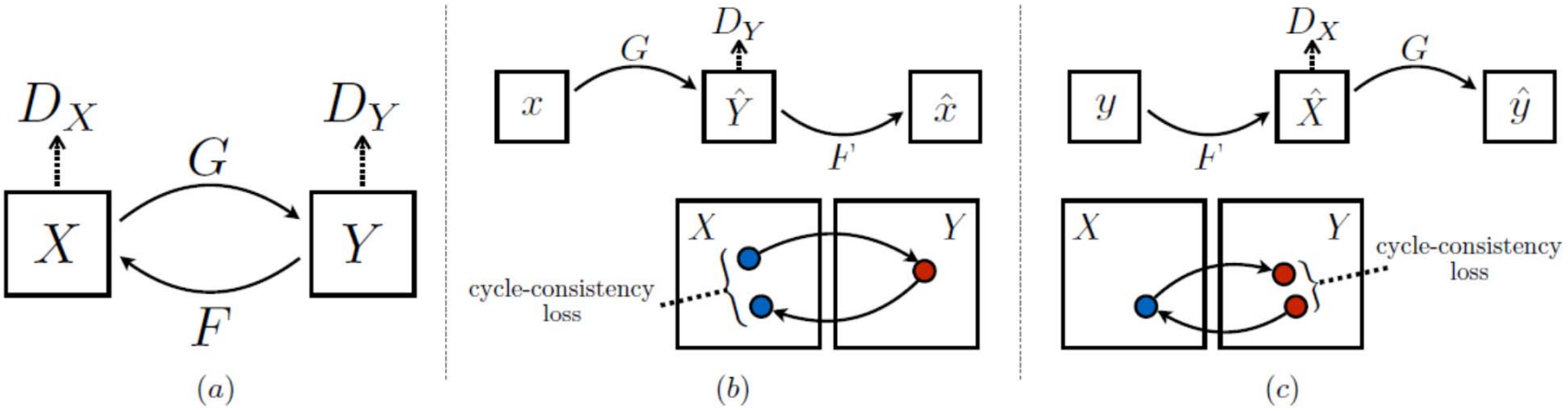


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X, F, and X. To further regularize the mappings, we introduce two "cycle consistency losses" that capture the intuition that if we translate from one domain to the other and back again we should arrive where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

Fig (Zhu 2017)

Cycle GANs (Zhu 2017)

- Training
  - The loss combines two conditional GAN losses (G, D_Y) and (F, D_X) and a cycle consistency loss
  - $C_{cycle}(F,G) = E_{p_{data}(x)}[\|F(G(x)) - x\|_1] + E_{p_{data}(y)}[\|G(F(y)) - y\|_1]$
  - $C = L(G, D_Y) + L(F, D_X) + C_{cycle}(F,G)$
  - Note: they replaced the usual $V(G, D_Y)$ and $V(F, D_X)$ term by a mean square error term, e.g.:
    - $L(G, D_Y) = E_{pdata(y)}[(D_Y(y) - 1)^2] + E_{data(x)}[D_Y(G(x))]$

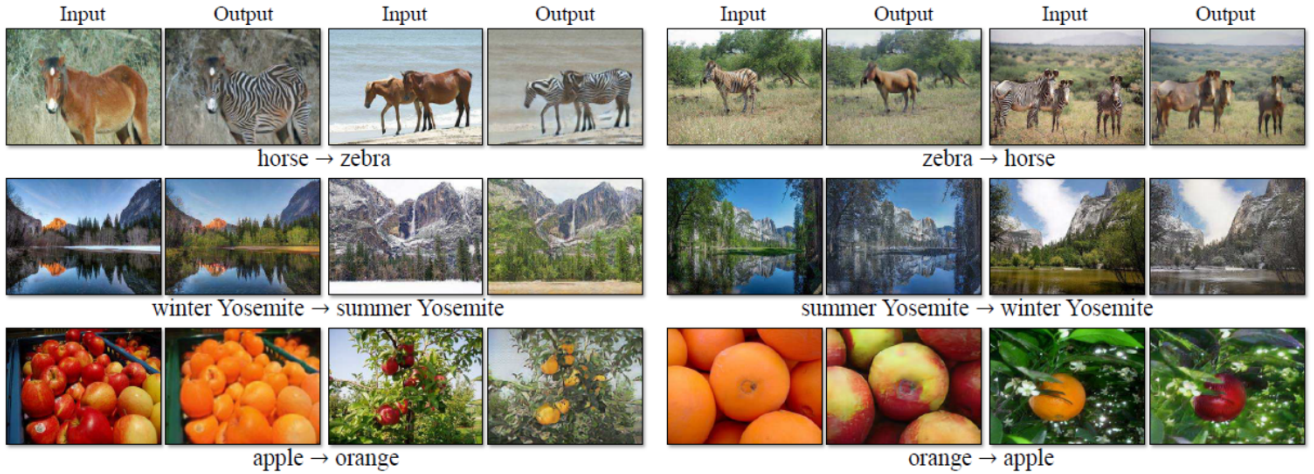
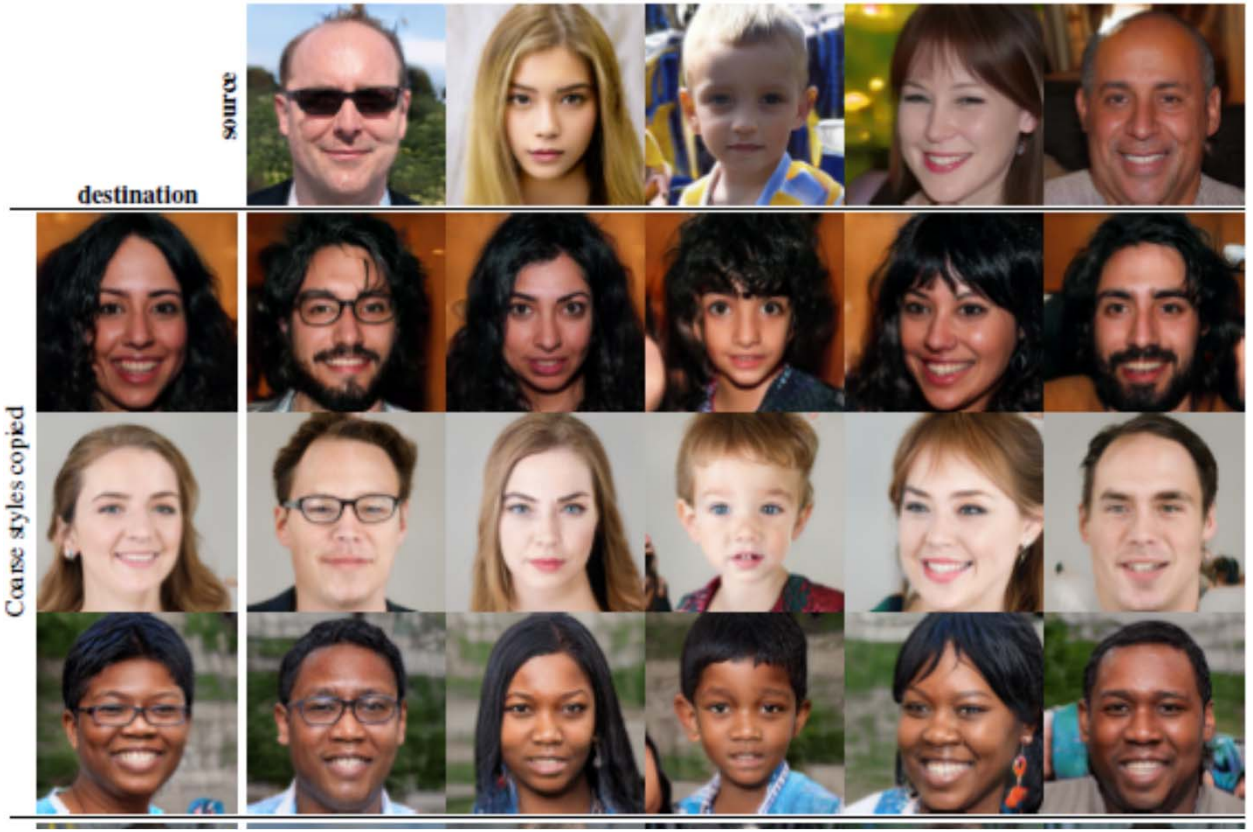


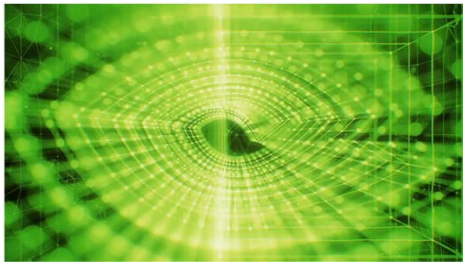
Figure 7: Results on several translation problems. These images are relatively successful results – please see our website for more comprehensive results.

Input Monet Van Gogh Cezanne Ukiyo-e

Large size demos
Style GAN (Karras et al. 2019)

- (Karras et al. 2019 - NVIDIA) – Style GAN





GANs summary

- Likelihood free training
- Optimize quality of generated samples
- Difficult to train
  - arXiv (2021-07-16) : **Showing 1–50 of 5,211 results for all: gans**
  - Mode collapse, convergence, ..
  - No best method, relies on heuristics (e.g. normalization)
  - Works on very high dimensional spaces

References: papers used as illustrations for the presentation

- General

  - 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). ttps://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 Icml-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*.