Neural Networks

(P-ITEEA-0011)

Introduction to the course
Single layer perceptron

Akos Zarandy
Lecture 1
September 10, 2019

Outline

- Administration: requirements of the course
- Machine learning – Machine intelligence
- Artificial neuron
- Perceptron

Course requirements: Signature requirements

- Mandatory **attendance** 80% (lectures and practice sessions)
- **Short quiz** at every practice session.
  - You have to reach at least **60% of all points**
- **Lab report**: one can be skipped
- **Paper based test**: minimum 50%
- **Computer-based test**: minimum 50%

Course requirements: Lab Reports

- Lab reports are short summaries of the previous practice session
- You will have to work in teams of 3 (**talent program alone**)
- Submission: on the main page of the course until 4 am the day before the next practice session
- Contents:
  - Your names, your email addresses, the time and date of the practice session
  - A brief description of the new methods/techniques and their mathematical background (if applicable) we used
  - A general description of the dataset we used (with examples from the dataset) (if applicable)
  - If we used any new network architectures, a detailed description of that specific architecture.
- You may use Internet, however you must cite that source, else your report will not be accepted. The same goes for too similar lab reports.

Course requirements: Midterm project

- Not mandatory in general
  - Mandatory for the **talent program**
- Required to earn an offered grade
- You will need to apply for it after it is announced
- Once you choose a task, nobody else can, so there will be no possibility of changing your task, or cancelling your selection
- You will have to submit an acceptable solution, otherwise your final score will be reduced by 20%

Course requirements: Tests

- **Paper-based test**
  - 15. October
  - Theoretic questions and paper based calculations
  - In the time and location of the lecture
  - You need to score at least 50% to pass

- **Computer-based test**
  - Considered to be a part of the exam
  - The test will be held at the end of the semester, it will be 3-4 hours long
  - The test will be graded on the spot
  - You need to score at least 50% to pass

Course requirements: Exam and grade

- **Exam**
  - Oral exam
- **Offered grade**
  - Only a 4 or 5 can be received
  - Limits on the offered grades:
    - > 85% of the short quizzes, the closed-room test
  - Midterm project required, final grade depends on it
- **Early exam**
  - There will also be an exam in the first of the exam period (before the computer-based test) for those students who excelled most during the semester. This exam is invite-only by the lecturers, and if you are invited, you are excused from the computer-based test

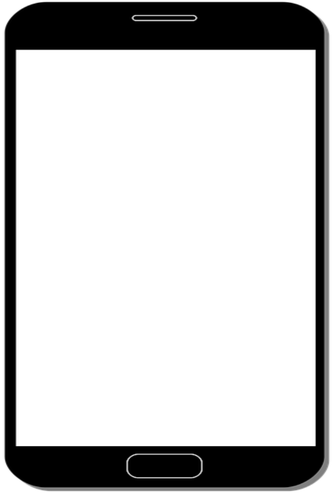
Detailed description of the requirements on the webpage of the course: http://users.itk.ppke.hu/~konso1/neural_networks

Outline

- Administration: requirements of the course
- Machine learning – Machine intelligence
- Artificial neuron
- Perceptron

Machine learning, machine intelligence

- What is intelligence?
- The ability to acquire and apply knowledge and skills.
- The definition changes continuously



Machine learning, machine intelligence

- What is intelligence?
- The ability to acquire and apply knowledge and skills.

Intelligence is the ability to adapt to change

„Stephen Hawking”

Providing computers the ability to learn without being explicitly programmed:

Involves: programming, Computational statistics, mathematical optimization, image processing, natural language processing etc...

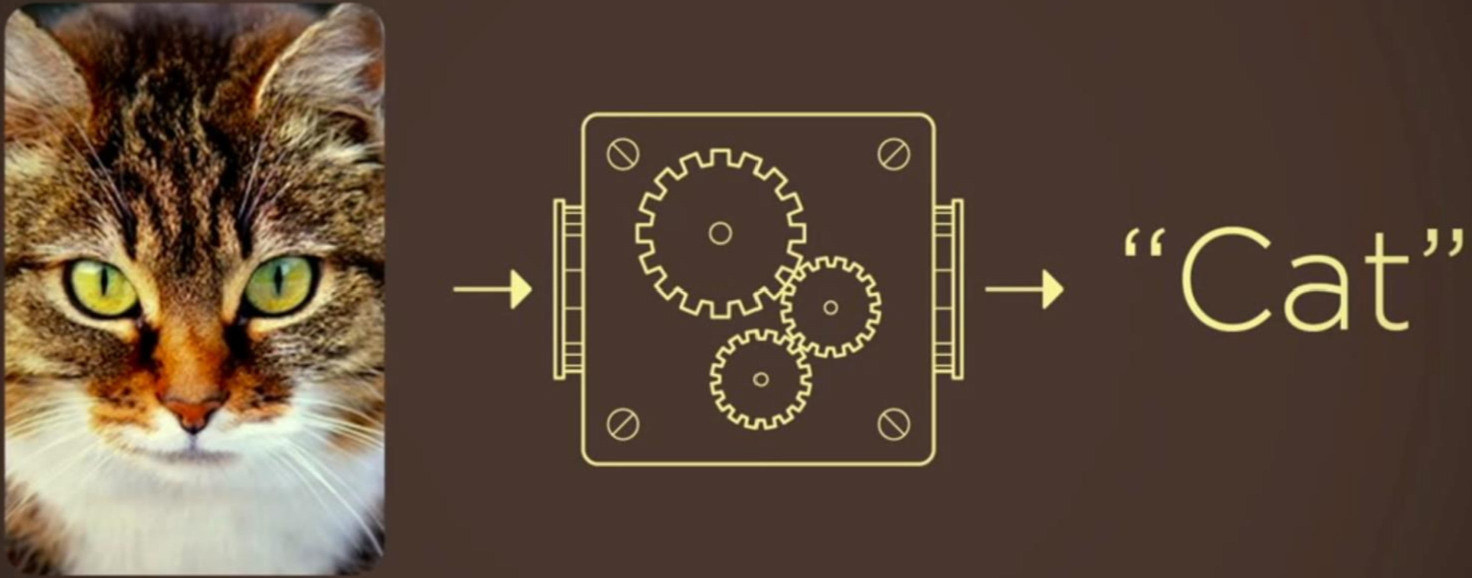
Conventional approach

- Trivial, or at least analitically solvable tasks
  - Well established mathematical solution exist or at least can be derived
- Example:
  - Finding well defined data constellations in a database
  - Formal verification of the operation is easy

Machine learning approach

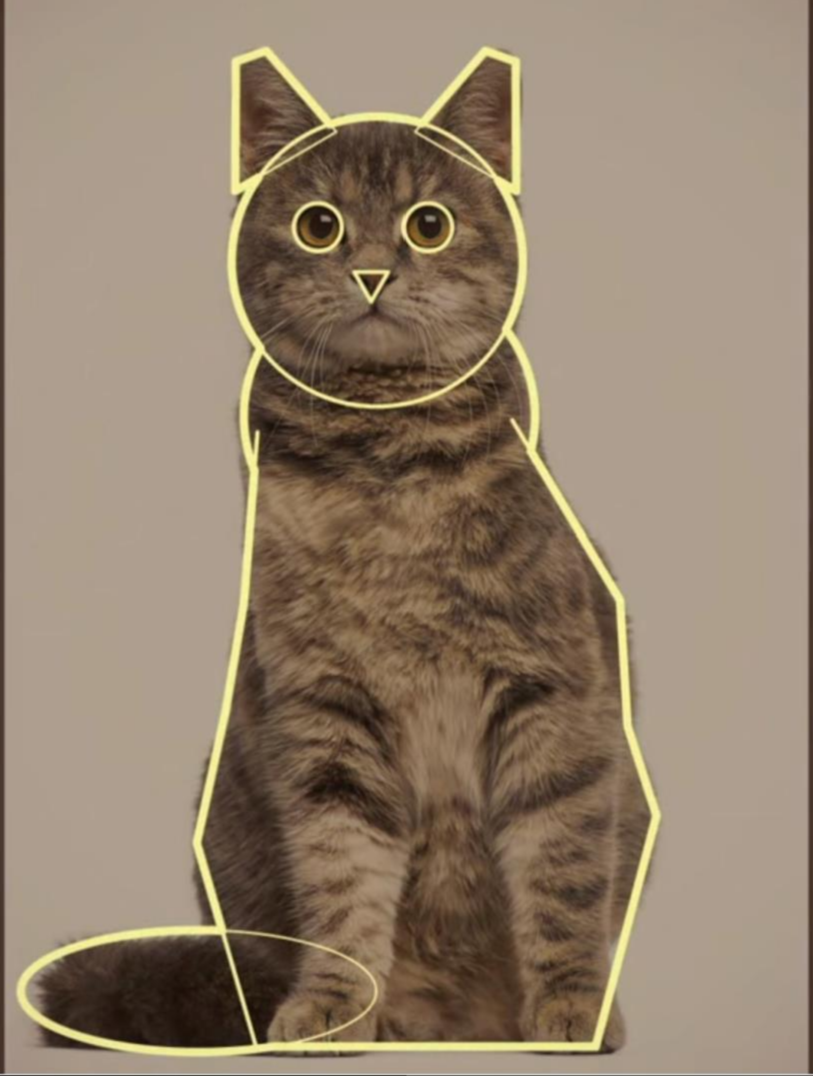
- Complex underspecified tasks
  - No exact mathematical solution exists, the function to be implemented is not known
- Example:
  - Searching for “strange” data constellations in a database
  - Verification of the operation is difficult

In case of very complex problems, verification of the operation is very difficult. Typically done by exhaustive testing in case of machine learning.

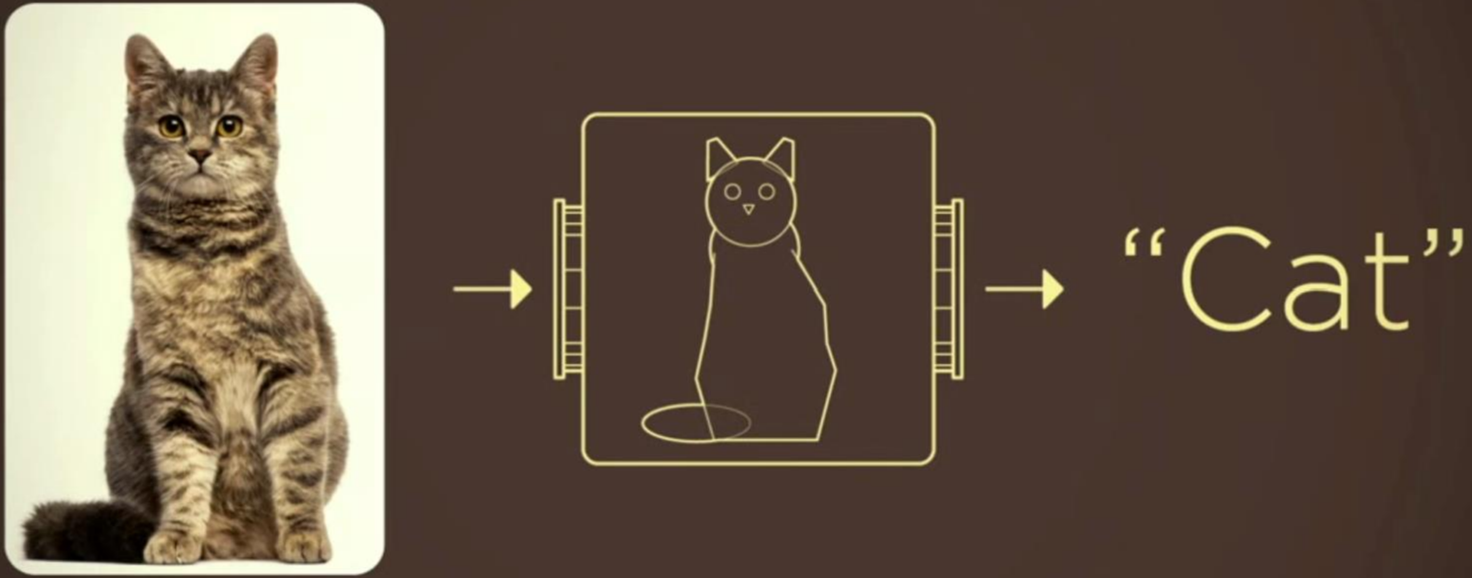


Crecits: Fei Fei Li

Crecits: Fei Fei Li

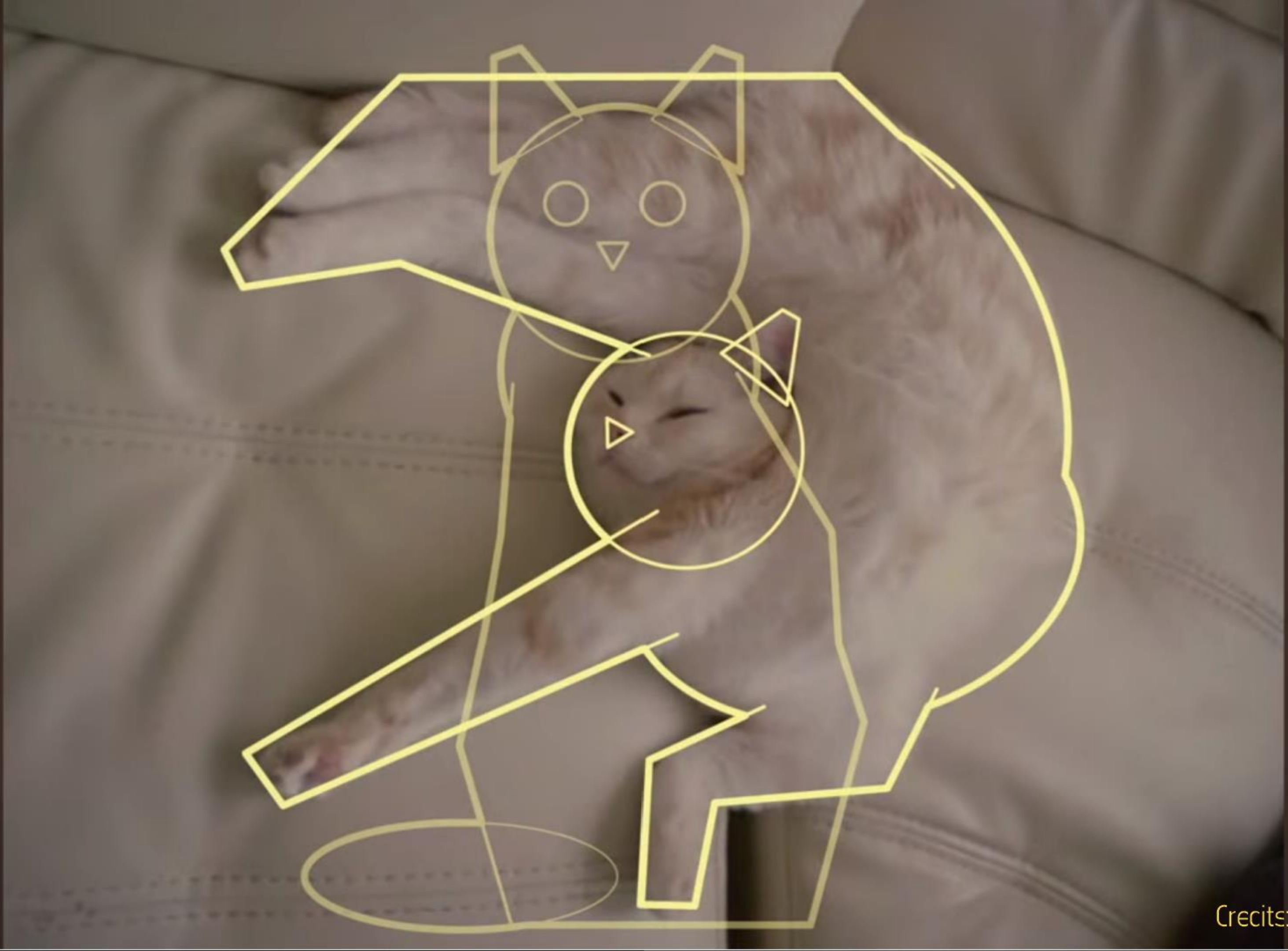


Crecits: Fei Fei Li



Crecits: Fei Fei Li

Crecits: Fei Fei Li



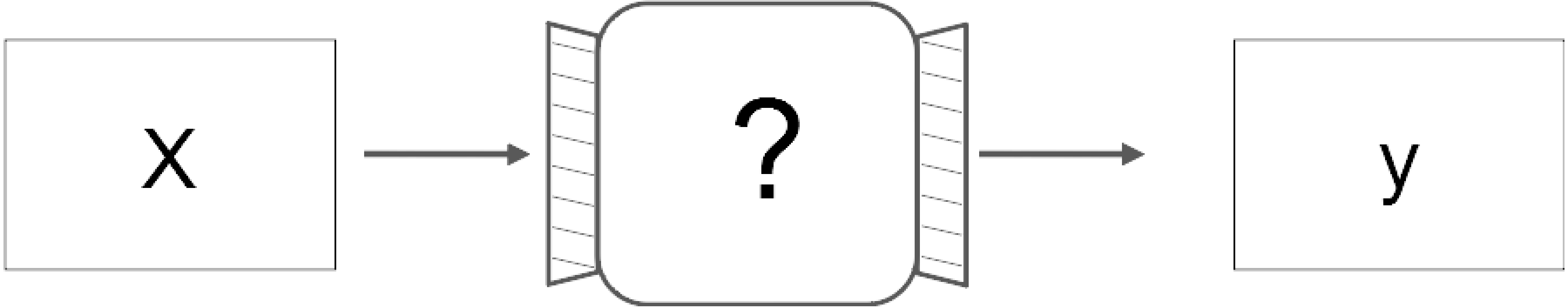
Crecits: Fei Fei Li


Crecits: Fei Fei Li

General truth: there are no general truths

Machine learning

We consider each task as an input-output problem



X: scalar, vector, array or a sequence of these (incl. text)

size(X) vs size(Y)
Data reduction
Data generation

Y: Decision or scalar, vector, array or a sequence of these (incl. text)

Conquests of machine learning

- 1952 Arthur Samuel (IBM): First machine learning program playing checkers

Arthur Samuel coined the term „machine learning”

Conquests of machine learning

- 1952 Arthur Samuel (IBM): First machine learning program playing checkers
- 1997 IBM Deep Blue Beats Kasparov

First match (1996 Nov):

Kasparov–Deep Blue (4–2)

Second Match (1997 May):

Deep Blue–Kasparov (3½–2½)

Conquests of machine learning

- 1952 Arthur Samuel (IBM): First machine learning program playing checkers
- 1997 IBM Deep Blue Beats Kasparov
- 2011 IBM Watson: Beating human champions in Jeopardy

It's a 4-letter term for a summit; the first 3 letters mean a type of simian : **Apex**

4-letter word for a vantage point or a belief : **View**

Music fans wax rhapsodic about this Hungarian's "Transcendental Etudes" : **Franz Liszt**

Conquests of machine learning

- 1952 Arthur Samuel (IBM): First machine learning program playing checkers

- 1997 IBM Deep Blue Beats Kasparov
- 2011 IBM Watson: Beating human champions in Jeopardy
- 2014 Deep face algorithm Facebook

Reached 97.35% accuracy
Human performance is around 97%

Conquests of machine learning

- 1952 Arthur Samuel (IBM): First machine learning program playing checkers

- 1997 IBM Deep Blue Beats Kasparov
- 2011 IBM Watson: Beating human champions in Jeopardy
- 2014 Deep face algorithm Facebook
- 2016 Alpha go: deep learning

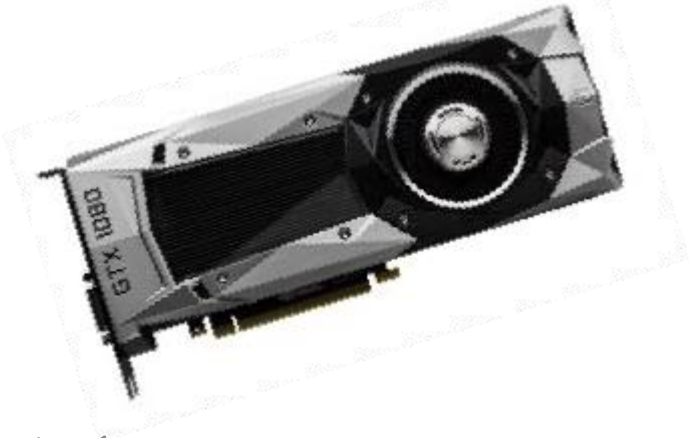
Fan Hui (5-0)
Lee Sedol (4-1)
99.8% win rate against other Go programs

Deep learning - why now?

1. Appearance of machine learning methods and frameworks, optimization know-how, new tools for rapid experimentation

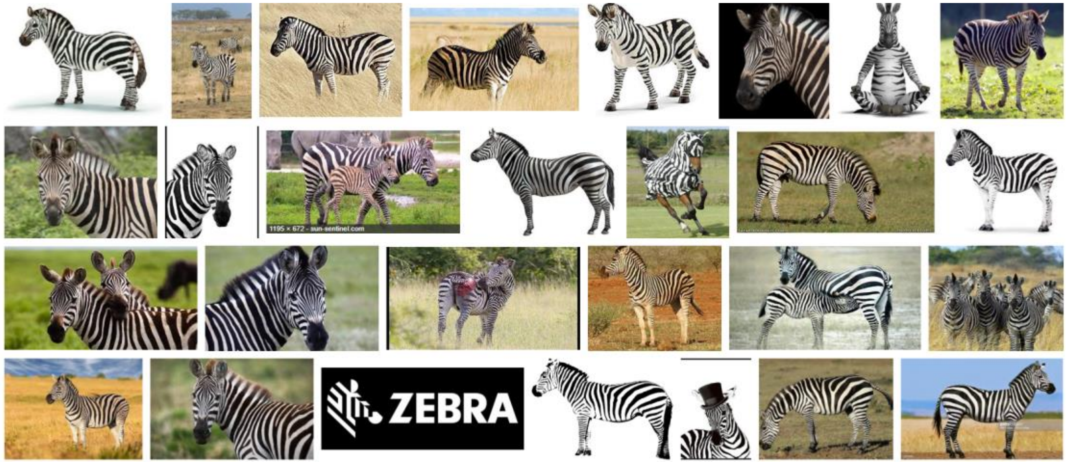
Deep learning - why now?

1. Appearance of machine learning methods and frameworks, optimization know-how, new tools for rapid experimentation
2. New architectures are available for computation
   – (1980: VIC-20 5kb RAM, MOS 6502 CPU 1.02Mhz)
   – (2018: NVIDIA GeForce GTX 1080, 8GB RAM, 1733 MHz, 2560 cores)



Deep learning - why now?

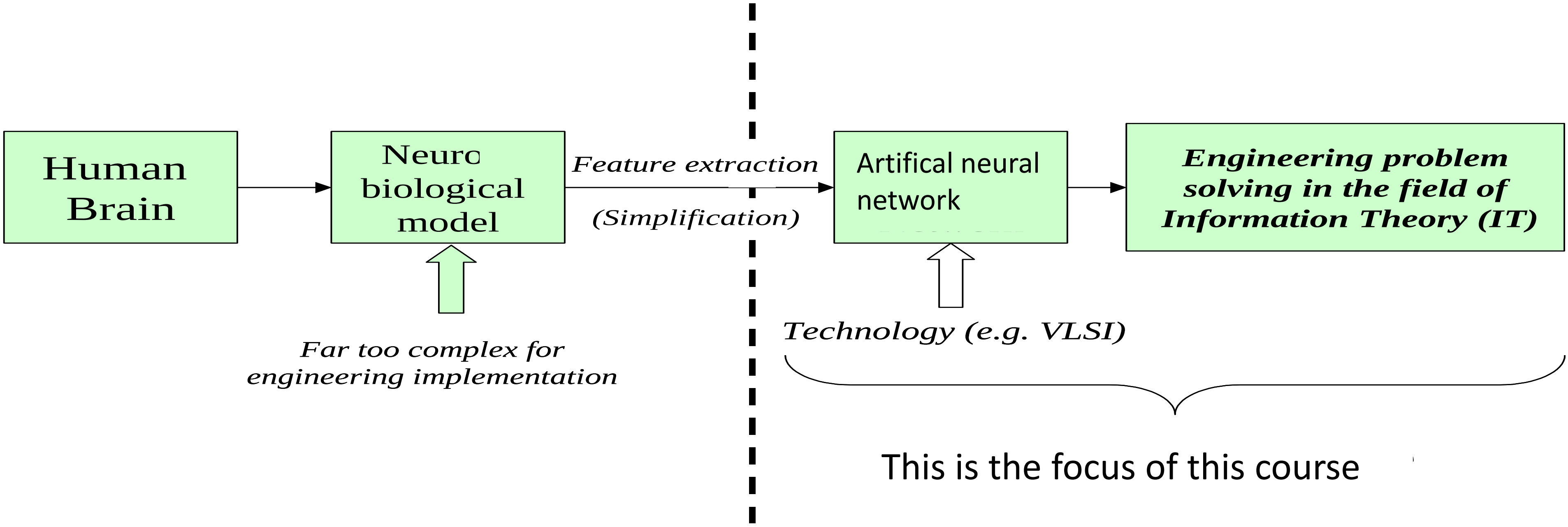
1. Appearance of machine learning methods and frameworks, optimization know-how, new tools for rapid experimentation
2. New architectures are available for computation
   - (1980: VIC-20 5kb RAM, MOS 6502 CPU 1.02Mhz)
   - (2018: NVIDIA GeForce GTX 1080, 8GB RAM, 1733 MHz, 2560 cores)
3. Vast amount of data is available
   - Billions of labeled images available quasi free



Outline

- Administration: requirements of the course
- Machine learning – Machine intelligence
- Artificial neuron
- Perceptron

Copying the brain?

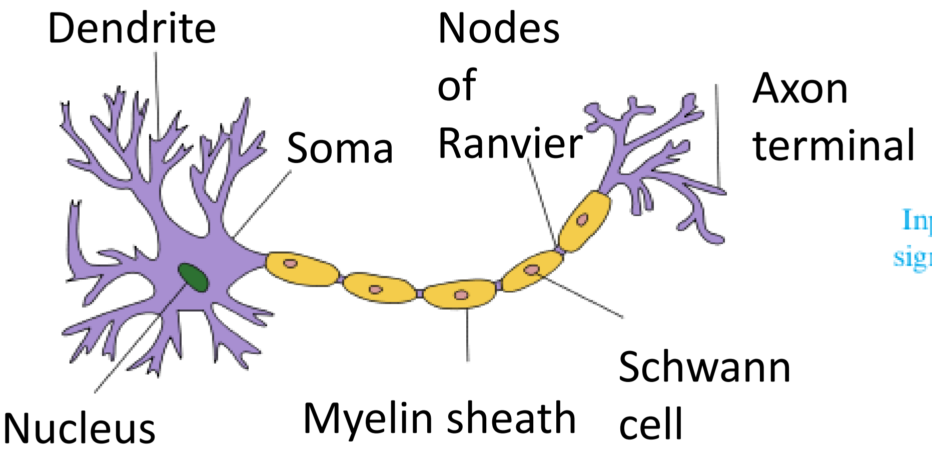


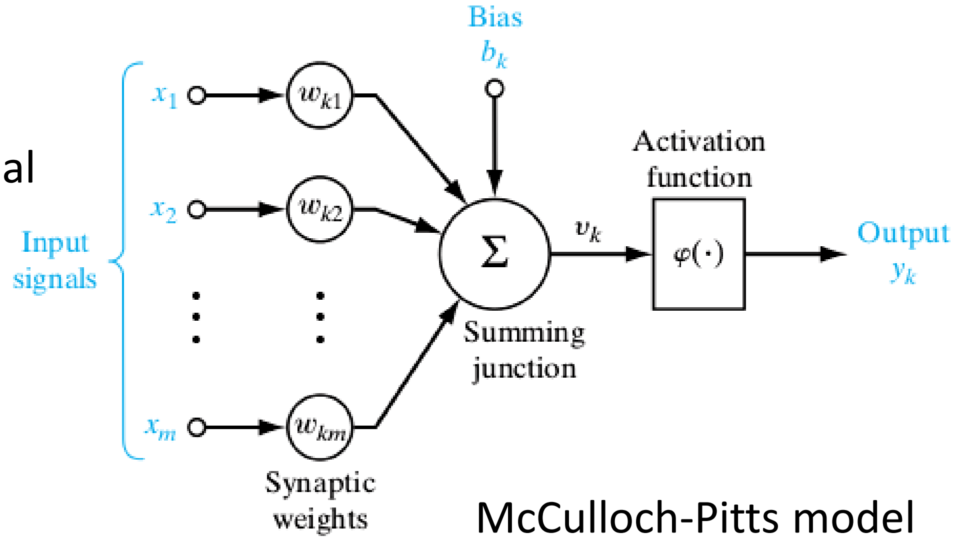
History of the artificial neural networks

- Artificial neuron model, 40’s (McCulloch-Pitts, J. von Neumann);
- Synaptic connection strenghts increase for usage, 40’s (Hebb)
- Perceptron learning rule, 50’s (Rosenblatt);
- ADALINE, 60’s (Widrow)
- Critical review ,70’s (Minsky)
- Feedforward neural nets, 80’s (Cybenko, Hornik, Stinchcombe..)
- Back propagation learning, 80’s (Sejnowsky, Grossberg)
- Hopfield net, 80’s (Hopfield, Grossberg);
- Self organizing feature map, 70’s - 80’s (Kohonen)
- CNN, 80’s-90’s (Roska, Chua)
- PCA networks, 90’s (Oja)
- Applications in IT, 90’s - 00’s
- SVMs, statistical machines 2000-2010’s
- Deep learning, Convolutional Neural Networks 2010-

The artificial neuron (McCulloch-Pitts)

- The artificial neuron is an information processing unit that is basic constructing element of an artificial neural network.
- Extracted from the biological model



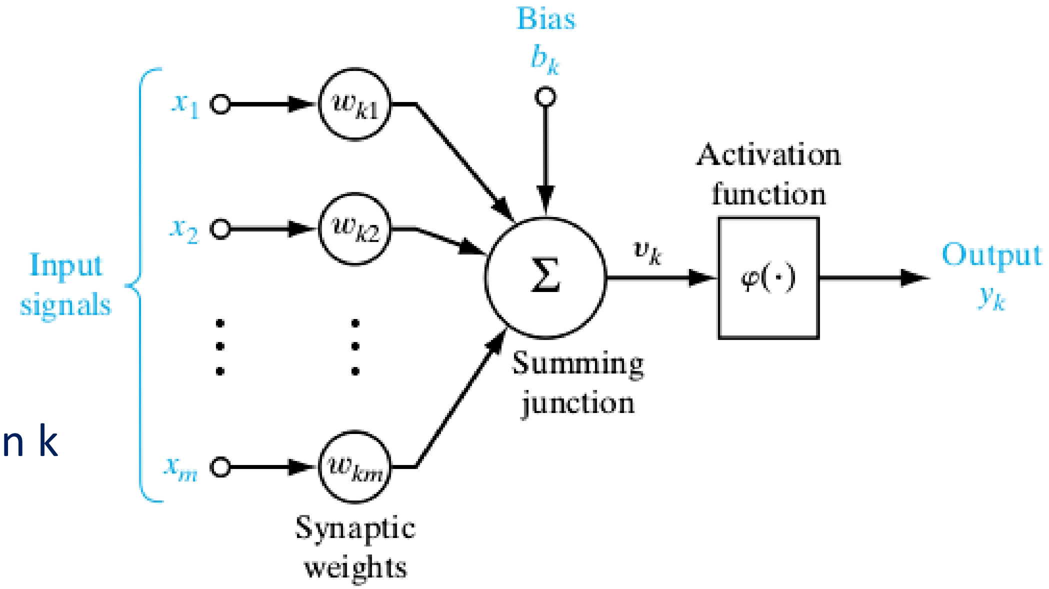


McCulloch-Pitts model

The artificial neuron

- Receives input through its synapsis (x_i)
- Synapsis are weighted (w_i)
  - if $w_i > 0$: amplified input from that source (excitatory input)
  - if $w_i < 0$: attenuated input from that source (inhibitory input)
- A *b* value biases the sum to enable asymmetric behavior
- A weighted sum is calculated
- Activation function shapes the output signal

x_i : input vector
w_{ki} : weight coefficient vector of neuron k
b_k : bias value of neuron k
o_k : output value of neuron k



The artificial neuron

- Output equation:

$$y_k = \varphi\left(\sum_{i=1}^{m} w_{ki} x_i + b_k\right)$$

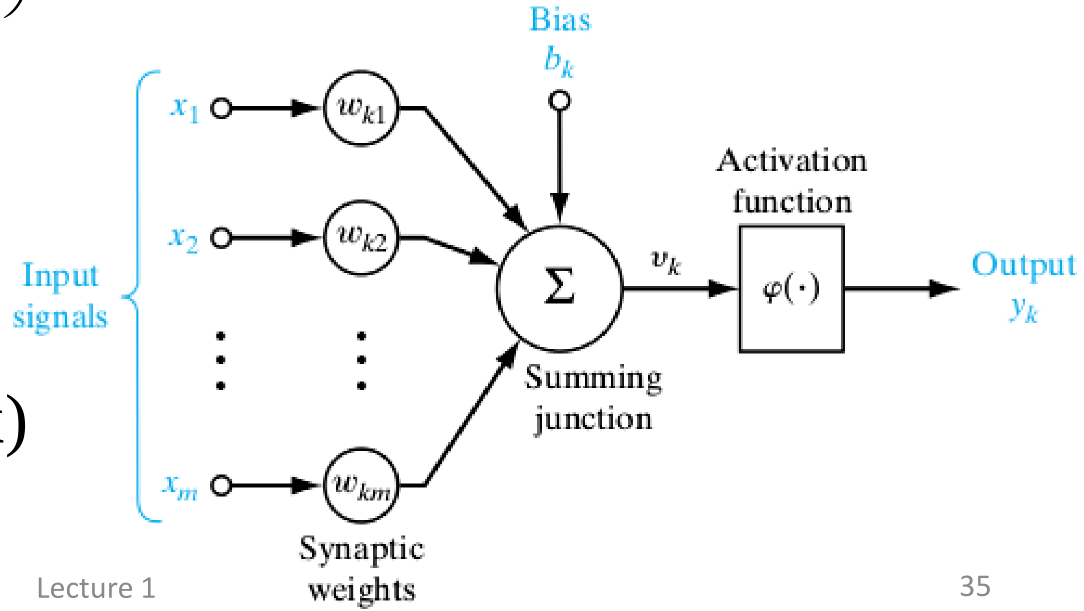
x_i : input vector (*i: 1….m*)
w_{ki} : weight coefficient vector of neuron k
b_k : bias value of neuron k
o_k : output value of neuron k

- Bias can be included as:

$w_0 = b$

$x_0 = 1$

$$y_k = \varphi\left(\sum_{i=0}^{m} w_{ki} x_i\right) = \varphi(\mathbf{w}^T \mathbf{x})$$



Activation functions (1)

- Activation function: φ(.)
  - Always a nonlinear function
  - Typically it clamps the output (introduces boundaries)
  - Monotonic increasing function
  - Differentiable
    - Important from theoretical point of view
  - Or at least continuous (except in simplified cases)
    - Sophisticated training algorithms require continuity

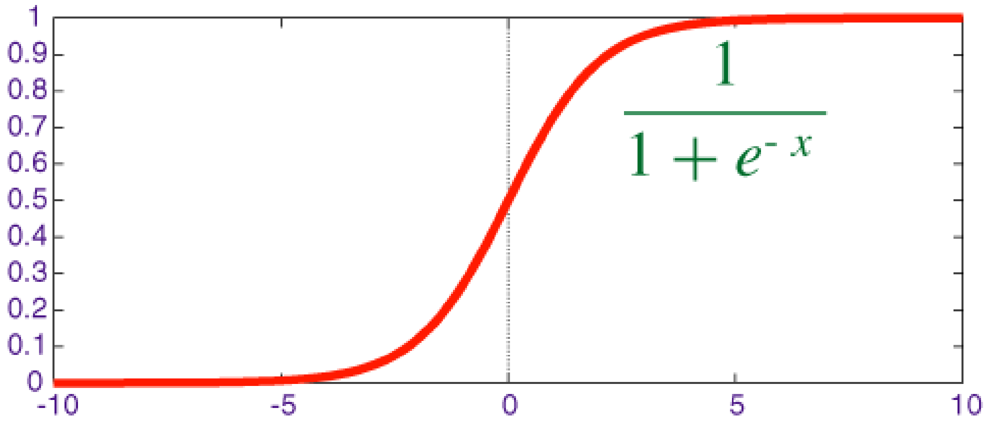
Activation functions (2)

- **Sigmoid** (or logistic) function is a widely activation function

$$y = \varphi(u) = \frac{1}{1+e^{-\lambda u}}$$

- where

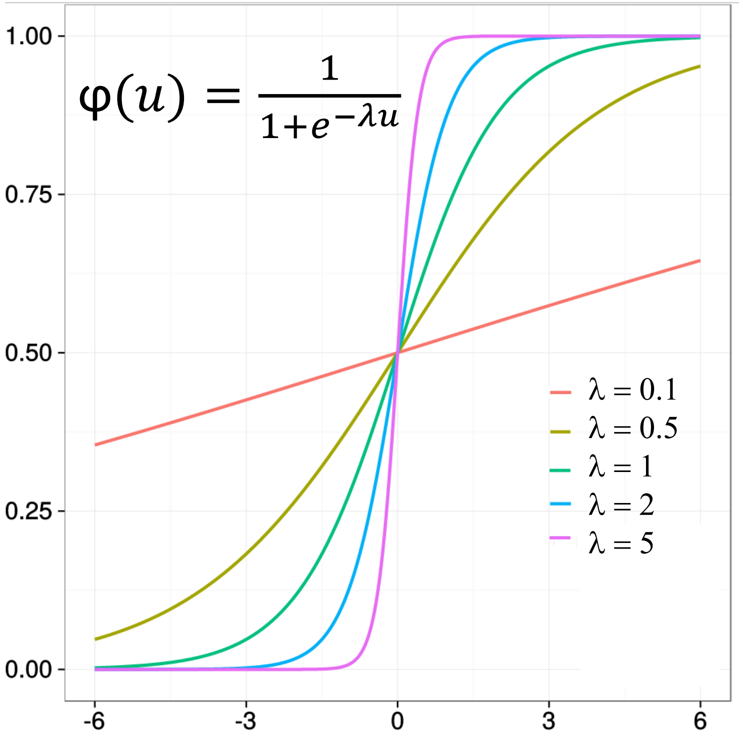
$$u = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$



Activation functions (3)

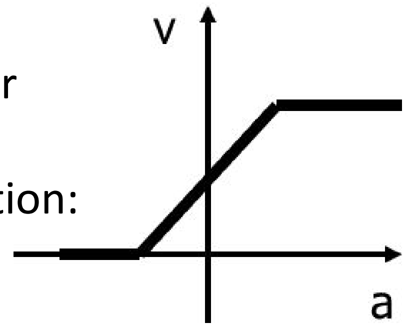
soft nonlinearity

(continuously differentiable)



hard nonlinearity

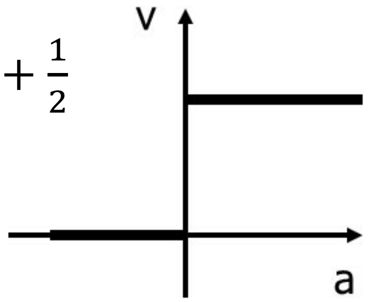
piece-wise linear implementation of sigmoid function:



$\lambda \longrightarrow \infty$

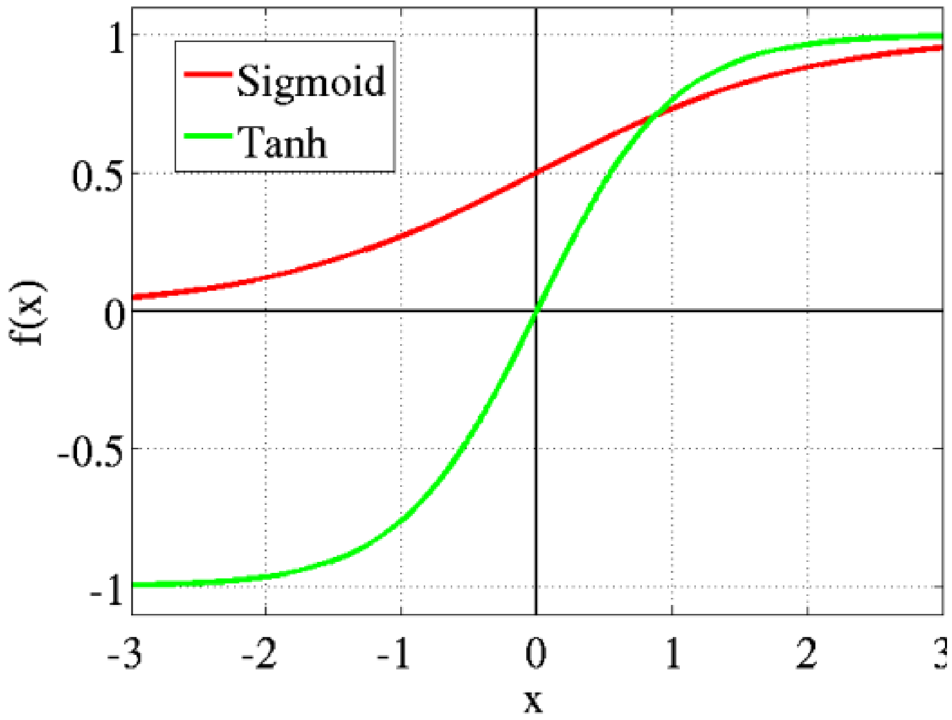
Step (threshold)function

$$\varphi(u) = \frac{sign(u)}{2} + \frac{1}{2}$$



Activation function (4)

- Bipolar activation function: tanh
- Continuously differentiable
- Monotonic
- Useful, when bipolar output is expected
- Hard approximations:
  - Piece-wise
  - Step-wise



Elementary set separation by a single neuron (1)

- Let us use φ(.) step nonlinear function for siplicity:

$$y = \varphi(u) = \frac{sign(u)}{2} + \frac{1}{2} = \begin{cases} 1, & \text{if } u \geq 0 \\ 0, & \text{else} \end{cases}$$

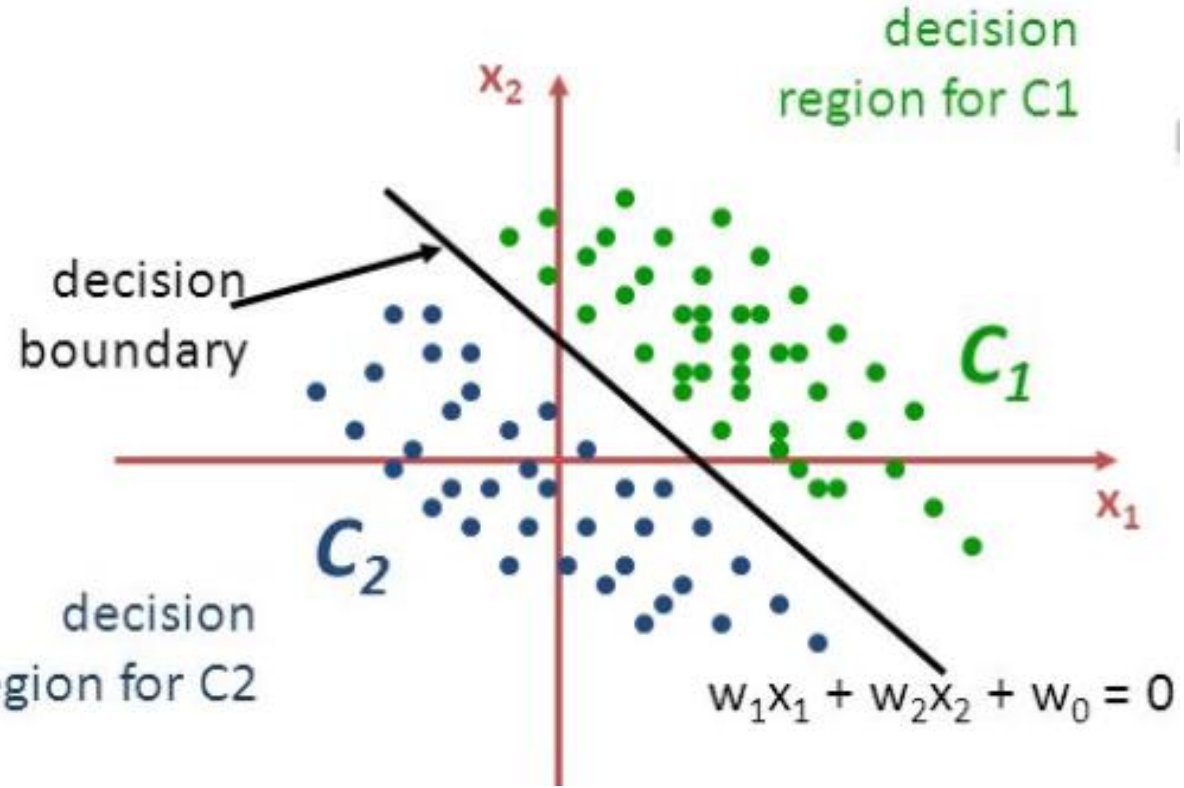
- The output of the neuron will be binary:

$$y = \varphi(u) = \frac{sign(\mathbf{w}^T \boldsymbol{x})}{2} + \frac{1}{2} = \begin{cases} 1, & \text{if } \mathbf{w}^T \boldsymbol{x} \geq 0 \\ 0, & \text{else} \end{cases}$$

DECISION!

Elementary set separation by a single neuron (2)

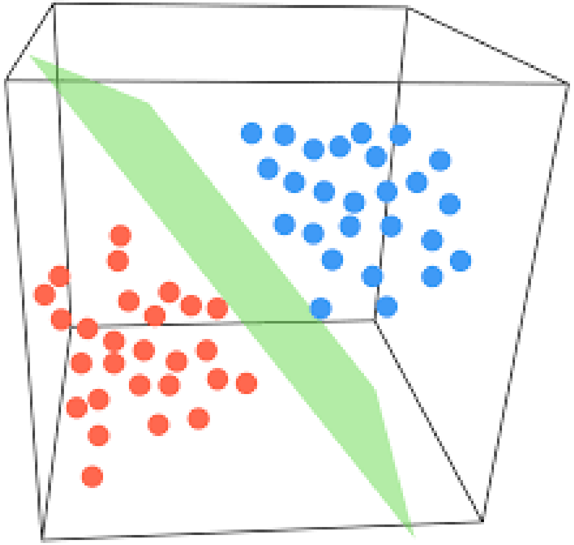
- in a 2-D input space, the hyper plane is a straight line.
- Above the line is classified: +1 (C1: yes)
- Below the line is classified : 0 (C2: no).



Elementary set separation by a single neuron (3)

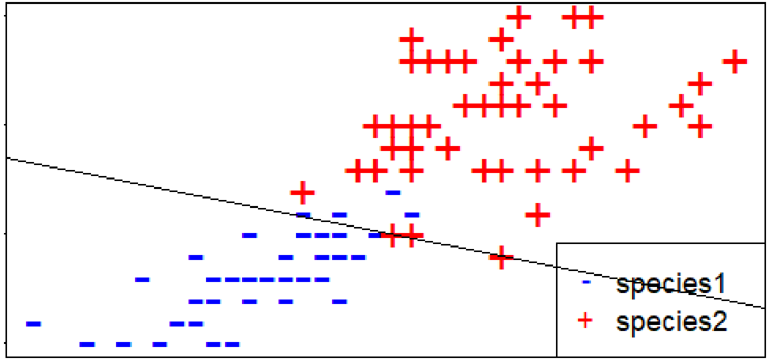
- Neuron with *m* inputs has an *m* dimensional input space
- Neuron makes a linear decision for a 2 class problem
- The decision boundary is a hyperplane defined:

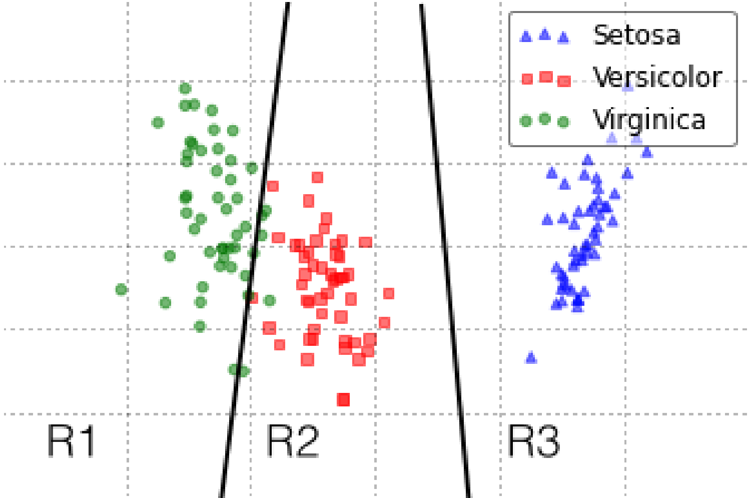
$$\mathbf{w}^T\mathbf{x} = 0$$



Why it is so important to use set separation by hyper plane? (1)

- Most logic functions has this complexity (OR, AND)
- There are plenty of mathematical and computational task which can be derived to a set separation problem by a linear hyper plane
- Application of multiple hyper plane provides complex decision boundary

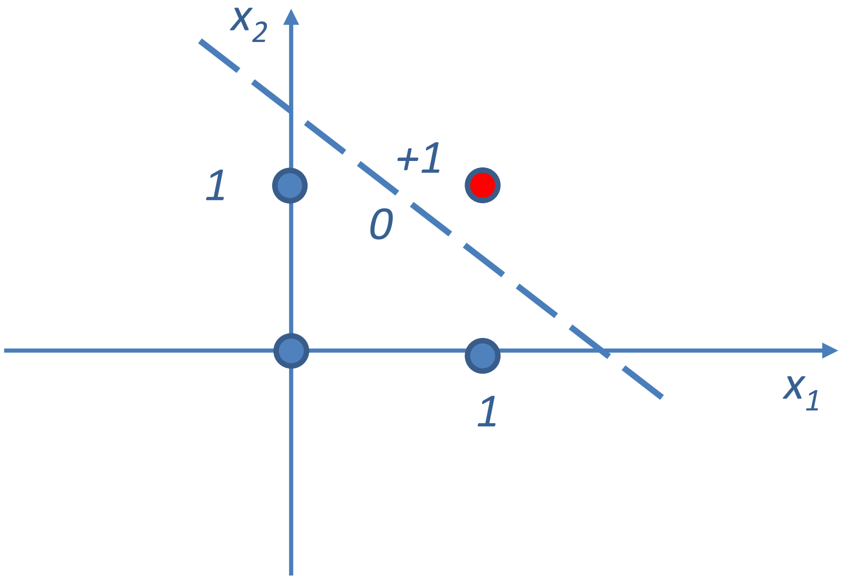




Implementation of a single logical function by a single neuron (1)

AND

x_1	x_2	$x_1 x_2$
0	0	0
0	1	0
1	0	0
1	1	1



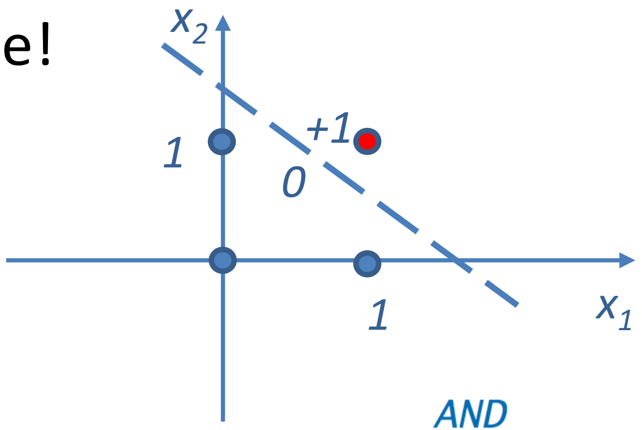
- The truth table of the logical AND function.
- 2-D AND input space and decision boundary

Implementation of a single logical function by a single neuron (2)

- We need to figure out the separation surface!
- Mathematically is the following equation:

$$-1.5 + x_1 + x_2 = 0$$

w_0=-1.5; w_1=1; w_2=1;



AND

x_1	x_2	x_1x_2
0	0	0
0	1	0
1	0	0
1	1	1

- The weight vector is:

$$y = \frac{sign(u)}{2} + \frac{1}{2} = \begin{cases} 1, & if\ \ u \geq 0 \\ 0, & else \end{cases}$$

w = (−1.5, 1, 1).

$$u = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$

$$x_0 = 1$$

Implementation of a single logical function by a single neuron (3)

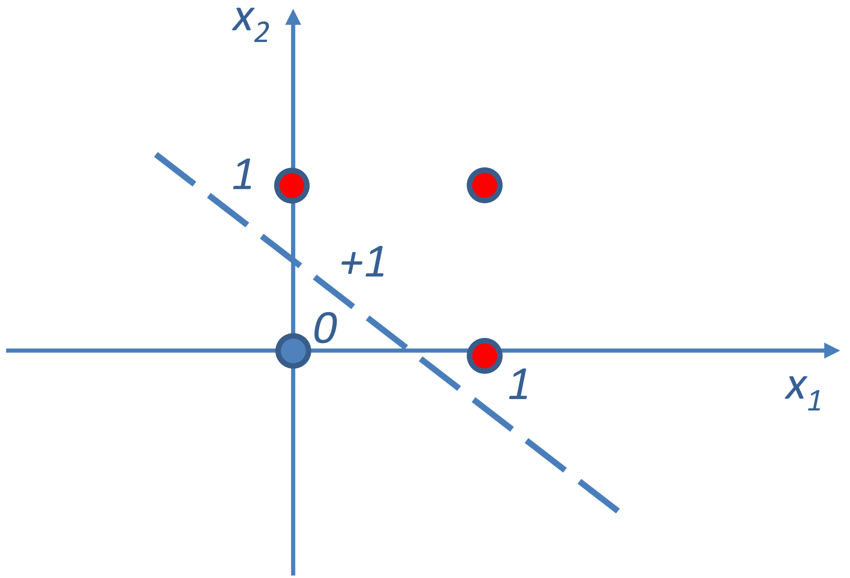
- Furthermore instead of 2D, we can actually come up with the *R* dimensional AND function.
- The weights corresponding to the inputs are all 1 and threshold should be *R* – 0.5. As a result the actual weights of the neuron are the following:

$$\mathbf{w}^{\mathrm{T}} = \left(-\left(R-0.5\right),1, \dots ,1\right)$$

Implementation of a single logical function by a single neuron (4)

OR

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1



- The truth table of the logical OR function.

$\mathbf{w}$ =(−0.5, 1, 1).

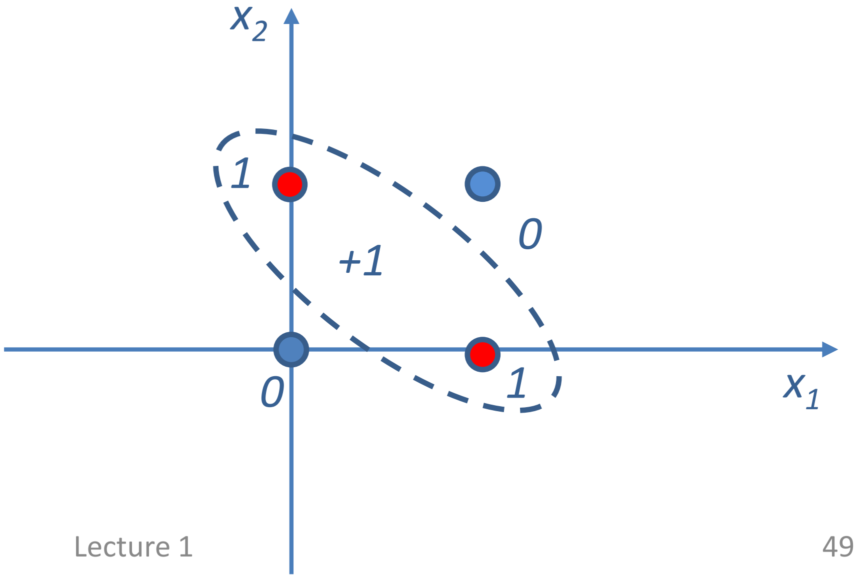
- 2-D OR input space and decision boundary

Implementation of a single logical function by a single neuron (5)

- However we cannot implement every logical function by a linear hyper plane.
- Exclusive OR (XOR) cannot be implemented by a single neuron (linearly not separable)

XOR

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

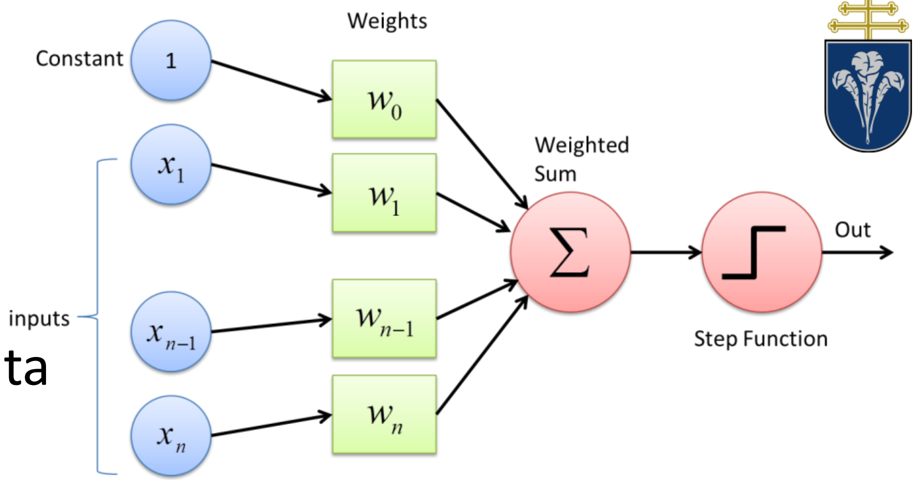


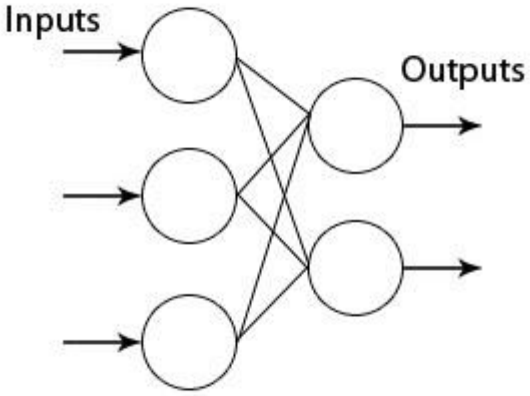
Outline

- Administration: requirements of the course
- Machine learning – Machine intelligence
- Artificial neuron
- Perceptron

Perceptron

- One or a set of neurons sharing the same input
- Typically used for decision making
- Multiple decisions from the same data
- Activation function
  - Originally step function
  - Sigmoid or Tanh or their piece-wise linear approximation is used nowadays
  - Sophisticated training algorithms require differentiable or at least continuous functions

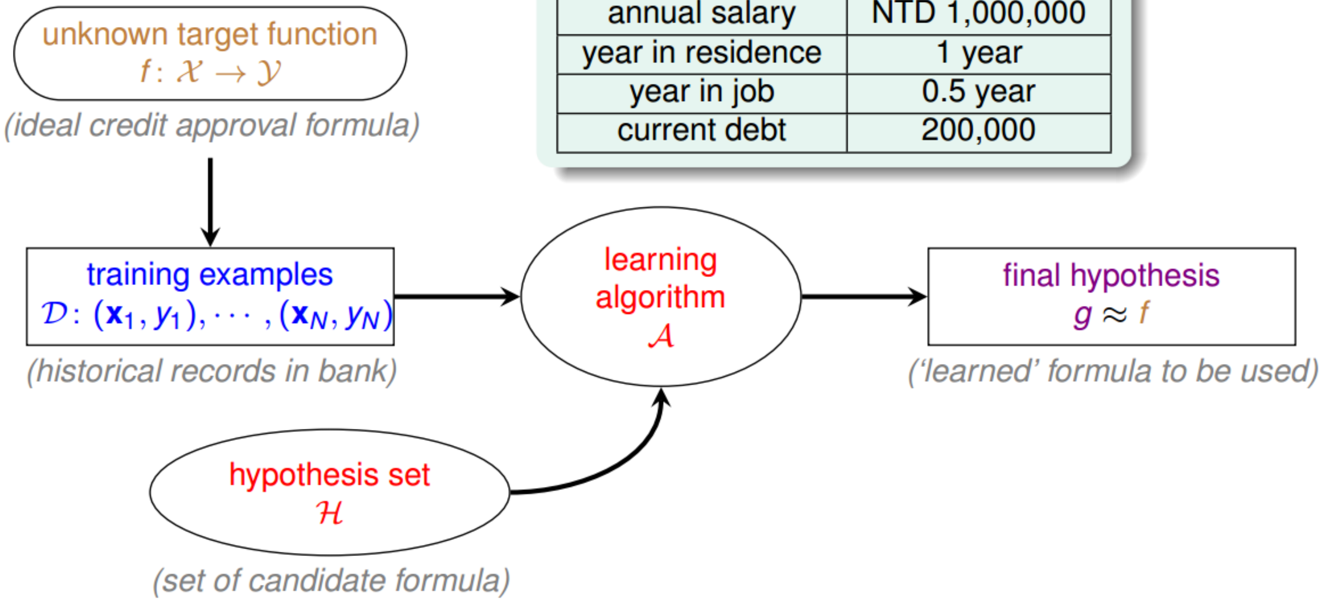




Credit Approval Problem Revisited

Applicant Information

age	23 years
gender	female
annual salary	NTD 1,000,000
year in residence	1 year
year in job	0.5 year
current debt	200,000



what hypothesis set can we use?

A Simple Hypothesis Set: the 'Perceptron'

age	23 years
annual salary	NTD 1,000,000
year in job	0.5 year
current debt	200,000

- For $\mathbf{x} = (x_1, x_2, \cdots, x_d)$ '**features of customer**', compute a weighted 'score' and

$$\text{approve credit if} \quad \sum_{i=1}^{d} w_i x_i > \text{threshold}$$

$$\text{deny credit if} \quad \sum_{i=1}^{d} w_i x_i < \text{threshold}$$

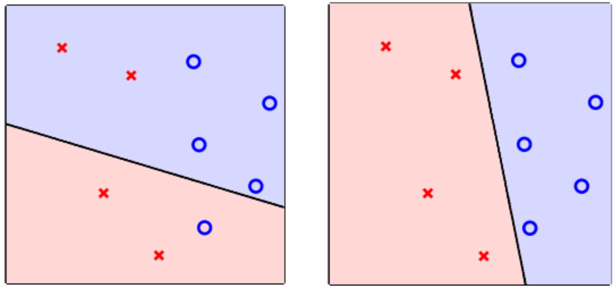
- $\mathcal{Y}$: $\{+1(\textbf{good}), -1(\textbf{bad})\}$, 0 ignored—linear formula $h \in \mathcal{H}$ are

$$h(\mathbf{x}) = \text{sign}\left(\left(\sum_{i=1}^{d} w_i x_i\right) - \text{threshold}\right)$$

called 'perceptron' hypothesis historically

Perceptrons in $\mathbb{R}^2$

$$h(\mathbf{x}) = \text{sign}\left(w_0 + w_1 x_1 + w_2 x_2\right)$$



- customer features $\mathbf{x}$: points on the plane (or points in $\mathbb{R}^d$)
- labels y: $\circ$ (+1), $\times$ (-1)
- hypothesis h: **lines** (or hyperplanes in $\mathbb{R}^d$)
  —positive on one side of a line, negative on the other side
- different line classifies customers differently

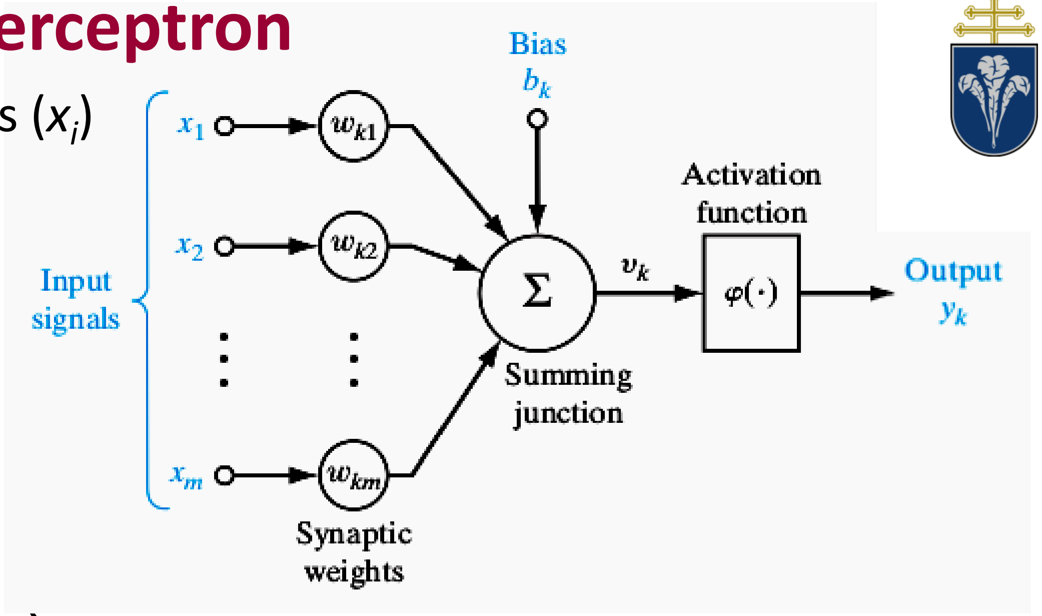
perceptrons $\Leftrightarrow$ **linear (binary) classifiers**

Outline

- Properties of the perceptron
- Input-output pairs
- Perceptron learning method
- Perceptron learning example
- Proof of convergence
- Good material:
  http://hagan.okstate.edu/4_Perceptron.pdf

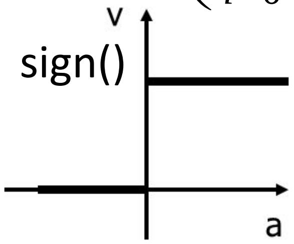
The Perceptron

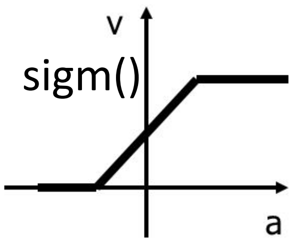
- Receives input through its synapsis (x_i)
- Synapsis are weighted (w_i)
- A b value biases the sum
  to enable asymmetric behavior
- A weighted sum is calculated
- Activation function applied



$$y_k = \varphi\left(\sum_{i=0}^{m} w_{ki} x_i\right) = \varphi(\mathbf{w}^T\mathbf{x})$$

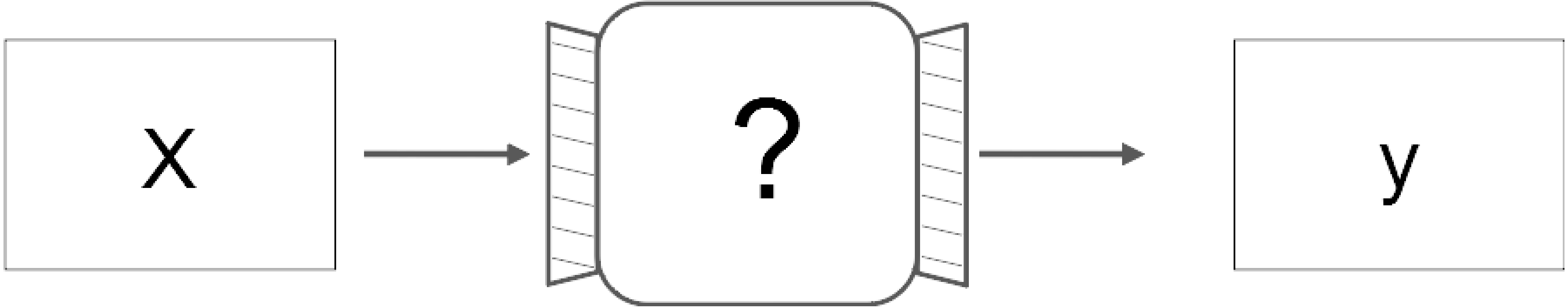
x_i : input vector
w_{ki} : weight coefficient vector of neuron k
b_k : bias value of neuron k
o_k : output value of neuron k





Neural Networks

Perceptron is an Input → Output device



As opposed to **Traditional Computers** where
- the math of the functionality is known
- the known math should be programmed

At **Neural Networks**

- the math behind the functionality is unknown
- the functionality is “illustrated” with examples

Function illustrated by examples

- Given a set of input-output pairs

  $x_j \rightarrow d_j$ (x_j: input vector; d_j: desired output)

	X		d
#	X_1	X_2	
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1

- Number of input vectors
  - Finite/limited set (*e.g.* AND function)
    - Equivalent with a look-up-table (LUT), math known
    - Mathematically it is correct to define a function by listing all the IO pairs
    - Goal: generate a simpler than LUT decision making device through learning
  - Infinite/open set (customers of a bank asking for a loan)
    - Math behind is unknown, cannot be coded directly
    - Goal: generate the function through learning
    - It should predict well the output of a previously unknown/untested input (**GENERALIZATION**)

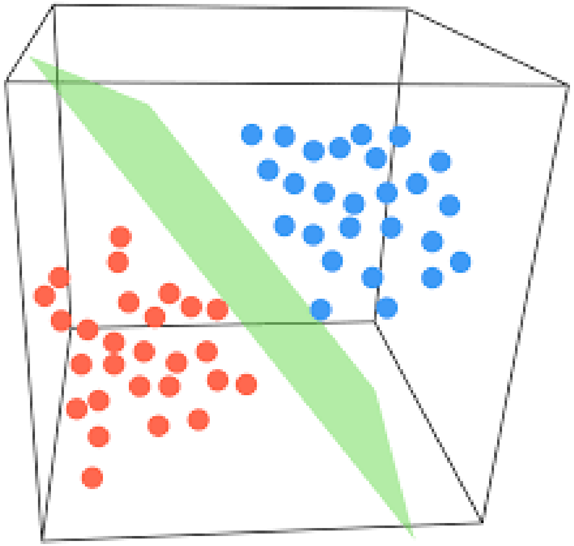
	X				d
#	age	gender	debt	salary	
1	25	M(1)	25	100	**Y(1)**
2	22	F(2)	18	80	**Y(1)**
3	65	M(1)	3000	200	**N(0)**
.	.	.	.	.	.
.	.	.	.	.	.

Good news: we can use the same learning/training method!!!

Linear separability

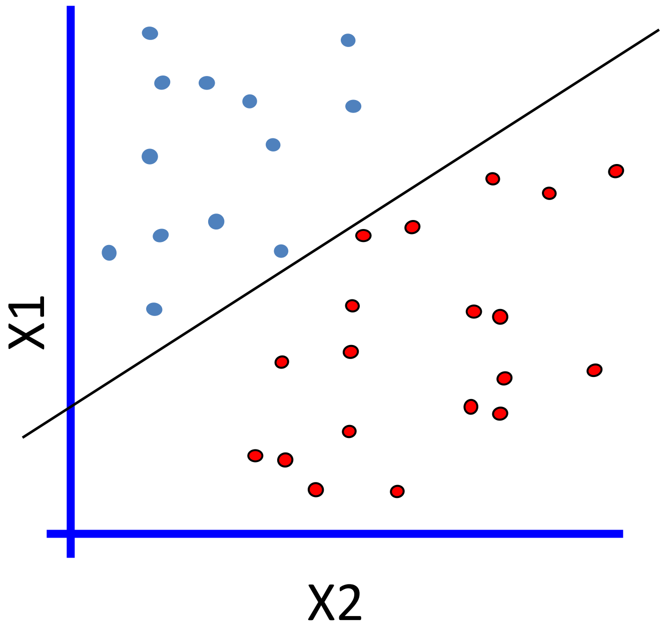
- Today, we assume that the IO sets are linearly separable
- The decision boundary is a hyperplane defined:

$$\mathbf{w}^T\mathbf{x} = 0$$



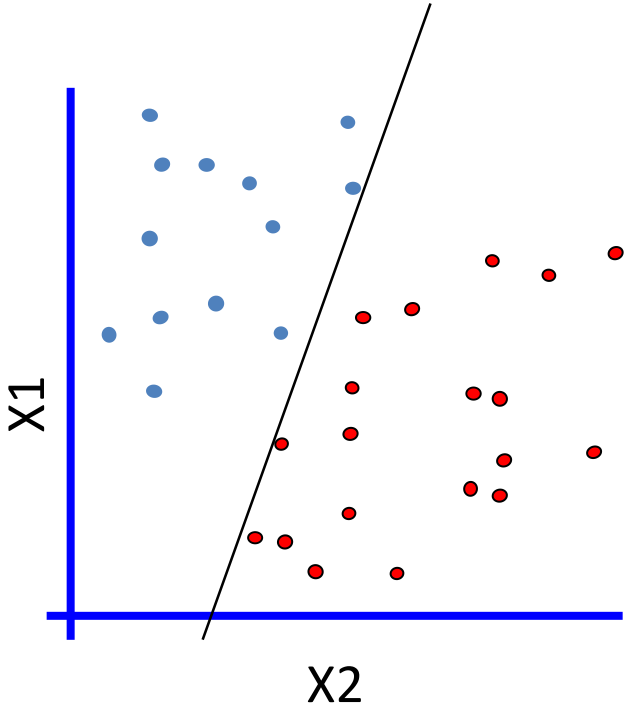
- Positive side of the hyperplane is classified: +1 (yes)
- Negative side of the hyperplane is classified : 0 (no).

Which boundary surface to use, if there are many?



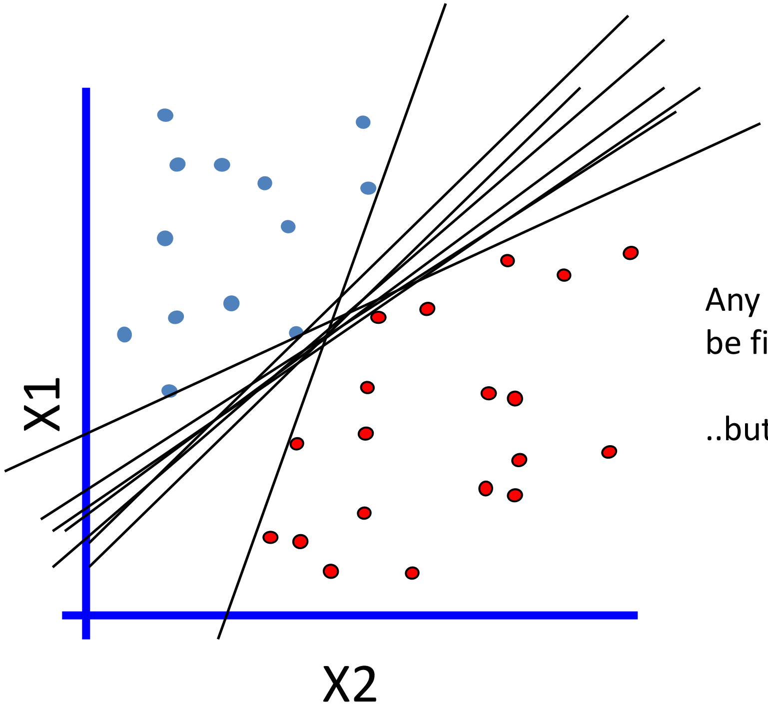
How would you classify this data?

Which boundary surface to use, if there are many?



How would you classify this data?

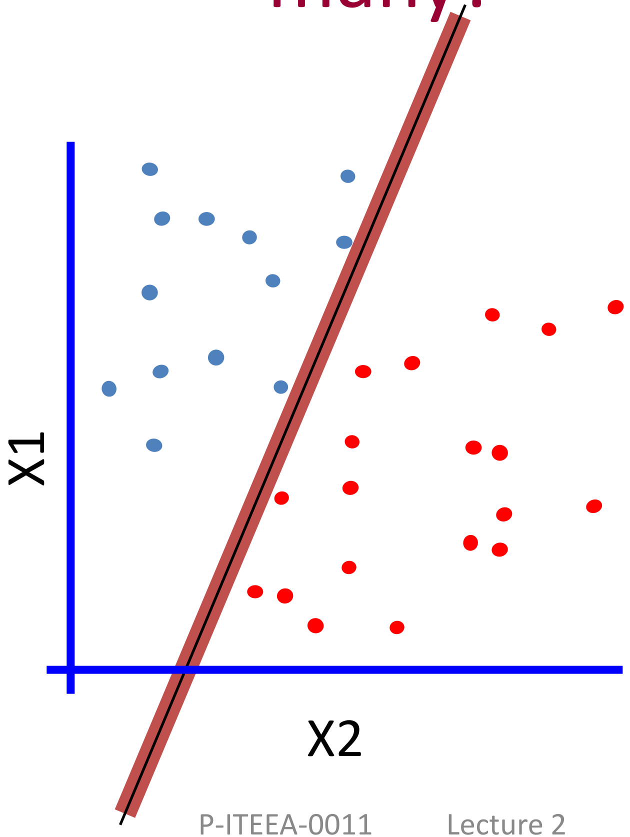
Which boundary surface to use, if there are many?



Any of these would be fine..

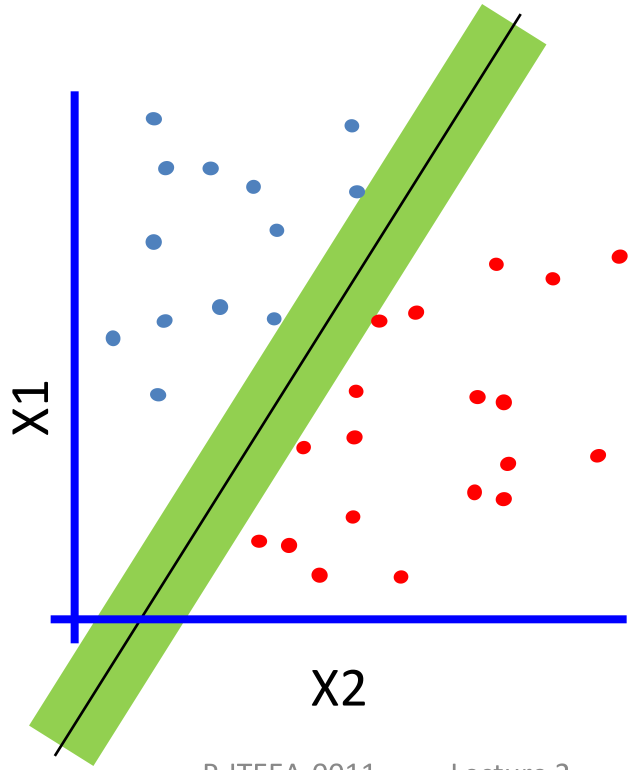
..but which is best?

Which boundary surface to use, if there are many?



<u>Maximum Margin:</u>
Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a data point.

Which boundary surface to use, if there are many?



Maximum Margin:
Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a data point.

What does learning mean?

- Given an annotated dataset

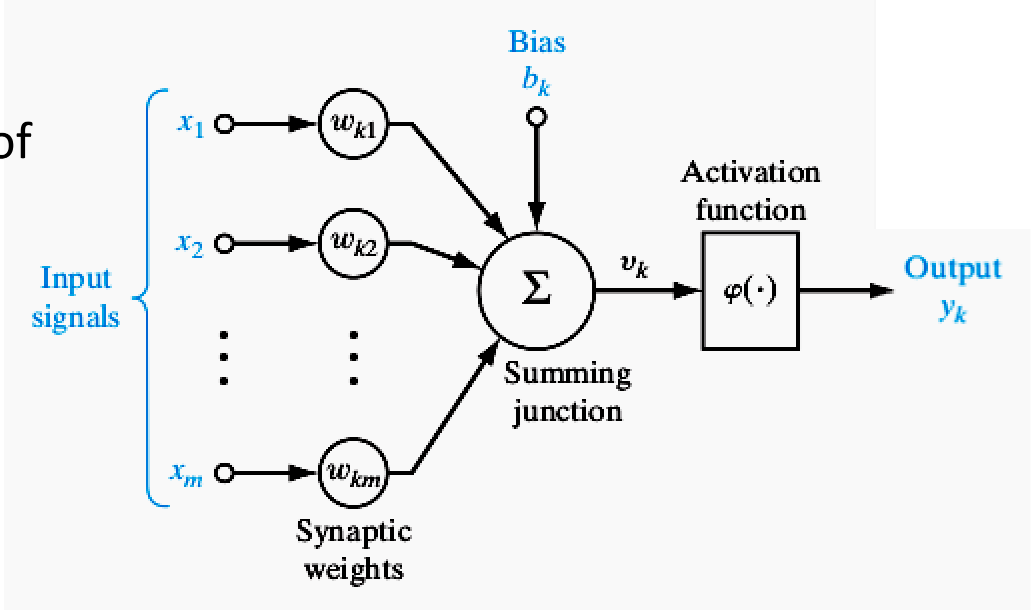
  $\mathbf{x}_j \rightarrow d_j$

- Given the parametric equation of the perceptron

$$y = sign(\mathbf{w}^T \mathbf{x})$$

- Goal: find the optimal $\mathbf{w}_{opt}$ weights (parameters), where for each j

$$d_j = sign(\mathbf{w}_{opt}^T \mathbf{x}_j)$$



The learning algorithm: Datasets

- Training set
  - *Set of input – desired output pairs*
  - *Will be used for training*

$$X^{+}=\{\mathbf{x}:\ d=+1\}$$
$$X^{-}=\{\mathbf{x}:\ d=0\}$$

- Test set
  - Used, when we have large set of input vectors (not used today)
  - *Set of input – desired output pairs*
  - *Will be used for testing and scoring the result*
- We assumed that X^{+} and X^{-} must be linearly separable
- We are looking for an optimal parameter set:

$$X^{+}=\{\mathbf{x}:\ \mathbf{w}_{\text{opt}}^{\mathrm{T}}\mathbf{x}>0\},$$
$$X^{-}=\{\mathbf{x}:\ \mathbf{w}_{\text{opt}}^{\mathrm{T}}\mathbf{x}<0\}.$$

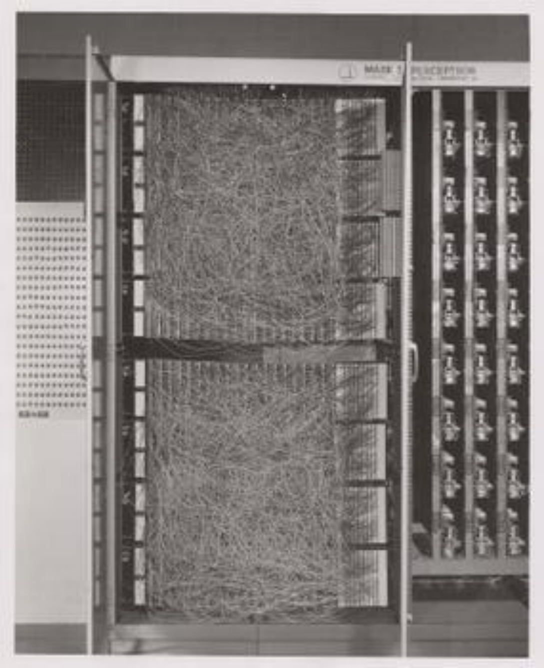
The learning algorithm: Recursive algorithm

- We have to develop a recursive algorithm called learning, which can learn the weight step by step, based on observing
  - the **(i) input,**
  - the **(ii) weight vector**,
  - the **(iii) desired output,** and
  - the **(iv) actual output** of the system.
- This can be described formally as follows:

$$\mathbf{w}(k+1) = \Psi\big(\mathbf{x}(k), \mathbf{w}(k), d(k), y(k)\big) \quad \rightarrow \quad \mathbf{w}_{opt}$$

The learning algorithm: Perceptron Learning Algorithm

- In a more ambitious way it can be called intelligent, because
  - perceptron can learn through examples (adapt),
  - even the function parameters are fully hidden.
- Perceptron learning was introduced by Frank Rosenblatt 1958
  - Built a 20x20 image sensor
  - With analog perceptron
  - 400 weights controlled by electromotors



The learning algorithm: Recursive steps

1. *Initialization.*

   Set **w**(0)=**0** or **w**(0)=**rand**

2. *Activation.*

   Select a $\mathbf{x}_k \rightarrow d_k$ pair

3. *Computation of actual response*

   $$y(k) = sign\left(w^T(k)x(k)\right)$$

4. *Adaptation of the weight vector*

   $$\mathbf{w}(k+1) = \Psi\left(\mathbf{x}(k), \mathbf{w}(k), d(k), y(k)\right)$$

5. *Continuation*

   Until all responses of the perceptron are OK

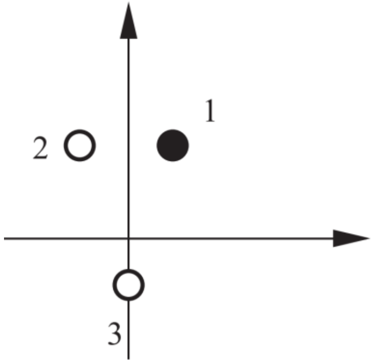
Weight update: very simple example

- Given a 3 input vector example

$$\mathbf{x}_1 = \begin{bmatrix}1\\2\end{bmatrix}, d_1 = 1;$$

$$\mathbf{x}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, d_2 = 0;$$

$$\mathbf{x}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, d_3 = 0;$$

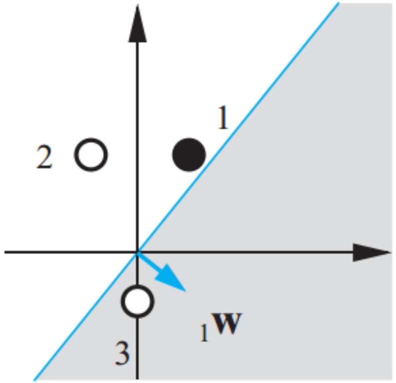


- Assume that bias is zero

  (decision boundary will cross the origo)

- Random initialization

$$\mathbf{w}^T(1) = [1 \quad -0.8];$$



Remember: the weight vector is orthogonal to the decision boundary!!!
Decision boundary: $x_1 - 0.8x_2 = 0$
Its orthogonal vector is: (1, -0.8)

Weight update: very simple example

- Test with the first input vector

$$\mathbf{x}_1 = \begin{bmatrix}1\\2\end{bmatrix}, d_1 = 1;$$

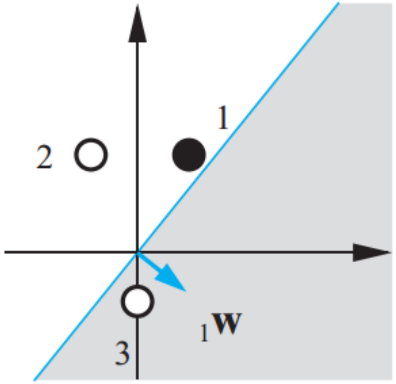
d_j-y_j > 0

$$\mathbf{w}^T(1) = [1 \quad -0.8];$$

$$y_1(1) = sign(w^T(1)x_1) = sign\left([1 \quad -0.8] \quad \begin{bmatrix}1\\2\end{bmatrix}\right) = sign(1 - 1.6) = 0$$

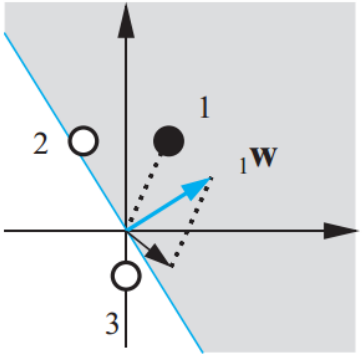
<u>The result is not OK! Positive misclassification: Instead of 1, the result is 0!!</u>

(The normal vector points to the positive side of the decision boundary.)



Idea: **<u>add</u>** the vector pointing to the positively misclassified point to the orthogonal vector of the decision boundary, to rotate it **<u>towards</u>** the point!
w(k+1)=w(k)+x_1

$$\mathbf{w}^T(2) = [1 + 1 \quad -0.8 + 2] = [2 \quad 1.2];$$



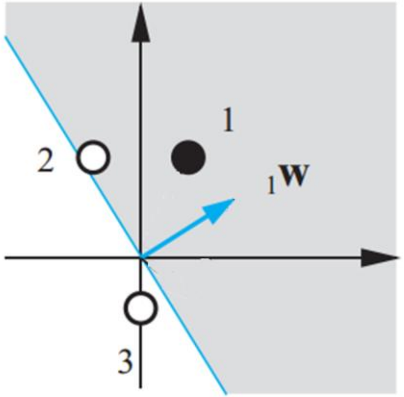
Weight update: very simple example

- Test with the second input vector $\mathbf{x}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, d_1 = 0;$ d_j-y_j < 0

$$\mathbf{w}^T(2) = [2 \quad 1.2];$$

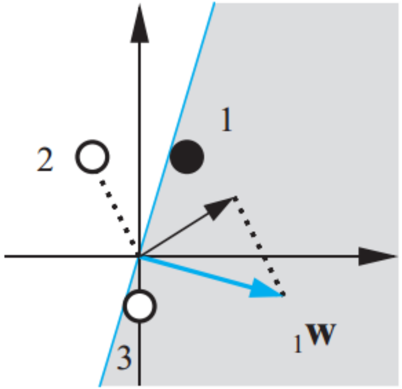
$$y_2(2) = sign(w^T(2)x_2) = sign\left([2 \quad 1.2] \quad \begin{bmatrix} -1 \\ 2 \end{bmatrix}\right) = sign(-2 + 2.4) = 1$$

The result is not OK! Negative misclassification: Instead of 0, the result is 1!!



Idea: **subtract** the vector pointing to the the negatively misclassified point to the orthogonal vector of the decision boundary, to rotate it **away** the point!
w(k+2)=w(k+1)-x_2

$$\mathbf{w}^T(3) = [2 - (-1) \quad 1.2 - 2] = [3 \quad -0.8]$$



Weight update: very simple example

- Test with the third input vector

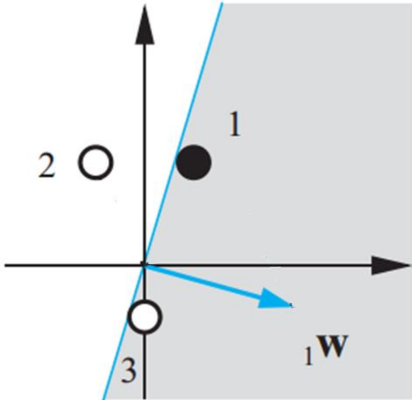
$$\mathbf{x}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, d_3 = 0;$$

d_j-y_j < 0

$$\mathbf{w}^T(3) = [3 \quad -0.8];$$

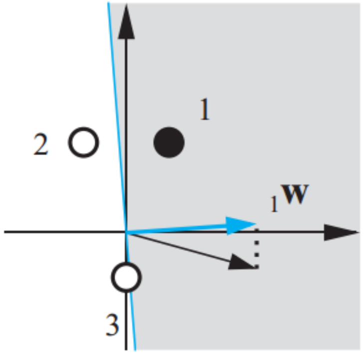
$$y_3(3) = sign(w^T(3)x_3) = sign\left([3 \quad -0.8] \quad \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right) = sign(0 + 0.8) = 1$$

The result is not OK! Negative misclassification: Instead of 0, the result is 1!!



Again: **subtract** the vector pointing to the negatively misclassified point to the orthogonal vector of the decision boundary, to rotate it **away** the point!
w(k+3)=w(k+2)-x_3

$$\mathbf{w}^T(4) = [3 - 0 \quad -0.8 - (-1)] = [3 \quad 0.2];$$



Weight update: very simple example

- Start again:
  - Test with the again with the first vector

    The result is OK!

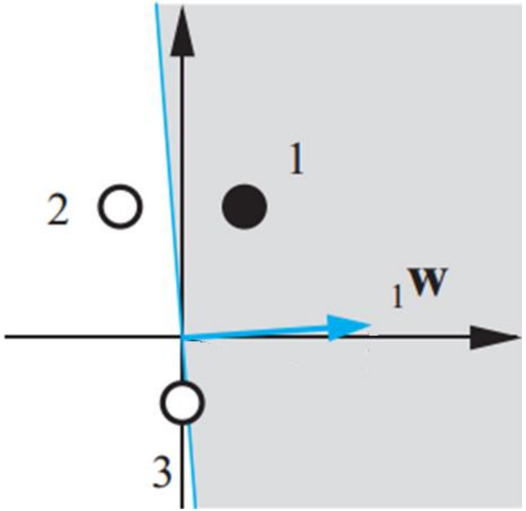
    - Do not modify!!!
  - Test with the again with the second vector

    The result is OK!

    - Do not modify!!!
  - Test with the again with the third vector

    The result is OK!

    - Do not modify!!!
- Since all input vectors are correctly classified: we are ready



Formalization of the update rules

- Positive misclassification : ADD

  $\varepsilon = d_j - y_j = 1$ $\quad w(k+1)=w(k)+x_j$

- Negative misclassification : SUBTRACT

  $\varepsilon = d_j - y_j = -1$ $\quad w(k+1)=w(k)-x_j$

- Correct classification : DO NOTHING

  $\varepsilon = d_j - y_j = 0$ $\quad w(k+1)=w(k)$

- In general:

$$w(k+1)=w(k)+\varepsilon\, x_j$$

The learning algorithm: Adaptation

We were looking for a recursive function:

$$\mathbf{w}(k+1) = \Psi\big(\mathbf{x}(k), \mathbf{w}(k), d(k), y(k)\big)$$

In general:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \varepsilon\eta\boldsymbol{x}_j$$

where is the error function

$$\varepsilon(k) = d(k) - y(k)$$

and

$$d(k) = \begin{cases} 0 & if \quad \mathbf{x}(k) \text{ belongs to class } \mathrm{X}^+ \\ -1 & \text{if} \quad \mathbf{x}(k) \text{ belongs to class } \mathrm{X}^- \end{cases},$$

η is the learning rate
(η controls the learning speed and should be positive)

AND

Weight update strategy

- Apply all the input vectors in one after the others, selecting them randomly
- Instance update
  - Update the weights after each input
- Batch update
  - Add up the modifications
  - Update the weights with the sum of the modifications, after all the inputs were applied
- Mini batch
  - Select a smaller batch of input vectors, and do with that as in the batch mode

Perceptron Convergence theorem (1)

Assumptions:

- ***w**(0)=0*
- *the input space is linearly separable, therefore **w**$_o$ (stands for **w**$_{optimal}$) exists:*

$$x \in X^+: \quad w_o^T x > 0: \ d = 1$$

$$x \in X^-: \quad w_o^T x < 0: \ d = 0$$

- *Let us denote* $\tilde{x} = -x$

$$\tilde{x} \in \tilde{X}^-: \quad w_o^T \tilde{x} > 0: \ d = 1$$

For the proof, see also: Simon Haykins: Neural Networks and Learning Machines, *Section 1.3:* http://dai.fmph.uniba.sk/courses/NN/haykin.neural-networks.3ed.2009.pdf

Perceptron Convergence theorem (2)

- Idea:
  - During the training, the network will be activated with those input vectors (one after the other), where the decision is wrong, hence non zero adaptation is needed:

$$x(j) \in X^{+}: \quad w^{T}(j)x(j) < 0, \quad y = 0, \quad d = 1$$

$$x(j) \in \tilde{X}^{-}: \quad w^{T}(j)x(j) < 0, \quad y = 0, \quad d = 1$$

  - Note: The error function is always positive ($\varepsilon = 1$)

Perceptron Convergence theorem (3)

- According to the learning method:
- $w(n+1)=w(0)+\eta x(0)+\eta x(1) +\eta x(2) +\eta x(3)+\ldots +\eta x(n)$
  - where

$$x(j)\in X^{+}: \quad w^{T}(j)x(j) < 0, \quad y=-1, \ d=1$$

or

$$x(j)\in \tilde{X}^{-}: \quad w^{T}(j)x(j) < 0, \quad y=-1, \ d=1$$

  - The decision boundary will be:

$$\eta w^{T}x=0$$

which means that η is a scaling factor, therefore it can be choosen for any positive number.

Let us use $\eta=1$, *therefore* $\eta\varepsilon=1$

Perceptron Convergence theorem (4)

- We will calculate $\|w(n+1)\|^2$ in two ways, and give an upper and a lower boundary, and it will turn out that an n_{max} exists, and beyond that the lower boundary is higher than the upper boundary (squeeze theorem, sandwitch lemma (*közrefogási elv, rendőr elv*))

Perceptron Convergence theorem (5)
lower limit (1)

According to the learning method, the presented input vectors are added up:

$$w(n+1) = w(0) + x(0) + x(1) + \ldots + x(n)$$

w(0)=0

Multiply it with w_o^T from the left:

$$w_o^T w(n+1) = w_o^T x(0) + w_o^T x(1) + \ldots + w_o^T x(n)$$

$$0 < \alpha \leq w_o^T x(j)$$

Because each input vector (or its opposite) were selected that way.

$$0 < \alpha = \min_{x(n) \in \{X^+, \tilde{X}^-\}} w_o^T x(n)$$

$$w_o^T w(n+1) \geq n\alpha$$

Perceptron Convergence theorem (6)
lower limit (2)

$$w_o^T w(n+1) \geq n\alpha$$

We apply Cauchy Schwarty inequality $\|a\|^2 \|b\|^2 \geq \|a^T b\|^2$

$$\|w_0^T\|^2 \|w(n+1)\|^2 \geq \|w_o^T w(n+1)\|^2 \geq n^2 \alpha^2$$

Lower limit:

$$\|w(n+1)\|^2 \geq \frac{n^2 \alpha^2}{\|w_0^T\|^2}$$

Lower limit proportional with n^2

Perceptron Convergence theorem (7) upper limit (1)

Let us have a different synthetization approach of *w(n+1)*:

$$w(k+1) = w(k) + x(k)$$

for k= 0 … n

Squared Euclidian norm:

$$\|w(k+1)\|^2 = \|w(k)\|^2 + \|x(k)\|^2 + 2w(k)^T x(k)$$

$$w(k)^T x(k) < 0$$

Because each input vector (or its opposite) were selected that way.

$$\|w(k+1)\|^2 \le \|w(k)\|^2 + \|x(k)\|^2$$

$$\|w(k+1)\|^2 - \|w(k)\|^2 \le \|x(k)\|^2$$

for k= 0 … n

Perceptron Convergence theorem (8) upper limit (2)

$$\|w(k+1)\|^2 - \|w(k)\|^2 \le \|x(k)\|^2$$

Telescoping sum: $\sum_{i=1}^{n}(a_{i+1} - a_i) = a_{n+1} - a_1$

Example: $\sum_{i=1}^{4}(a_{i+1} - a_i) = \cancel{a_2} - a_1 + \cancel{+a_3} - \cancel{a_2} + \cancel{+a_4} - \cancel{a_3} + +a_5 - \cancel{a_4} = $
$= \boldsymbol{a_5 - a_1}$

Summing up the upper term:

$$\sum_{k=0}^{n}\left(\|w(k+1)\|^2 - \|w(k)\|^2\right) \le \sum_{k=0}^{n}\|x(k)\|^2$$

Note that there is a telescoping sum in the left hand side.

$$\|w(n+1)\|^2 - \|w(0)\|^2$$

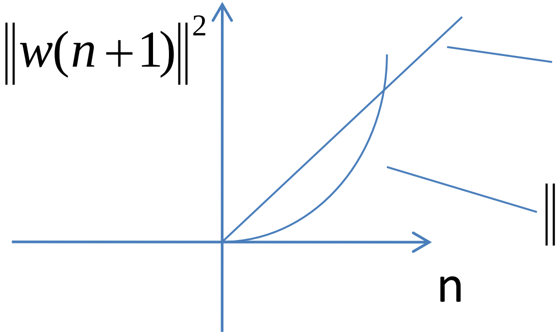
$$0 < \beta = \max_{x(k)\in\{X^+,\tilde{X}^-\}}\|x(k)\|^2$$

$\|w(0)\|^2 = 0$

$$\|w(n+1)\|^2 \le (n+1)\beta$$

Upper limit linearly proportional with n

Perceptron Convergence theorem (9)
comparing upper and lower limits



$$\|w(n+1)\|^2 \le (n+1)\beta$$

$$\|w(n+1)\|^2 \ge \frac{n^2\alpha^2}{\|w_0^T\|^2}$$

Linear upper limit and squared lower limit cannot grow unlimitedly

n_{max} should exist

$$n_{max} = \frac{\beta\|w_0\|^2}{\alpha^2}$$

Neural Networks

(P-ITEEA-0011)

Multilayer Perceptron
Back-propagation algorithm

Akos Zarandy

Lecture 3

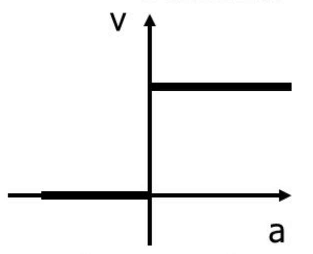
September 24, 2019

Contents

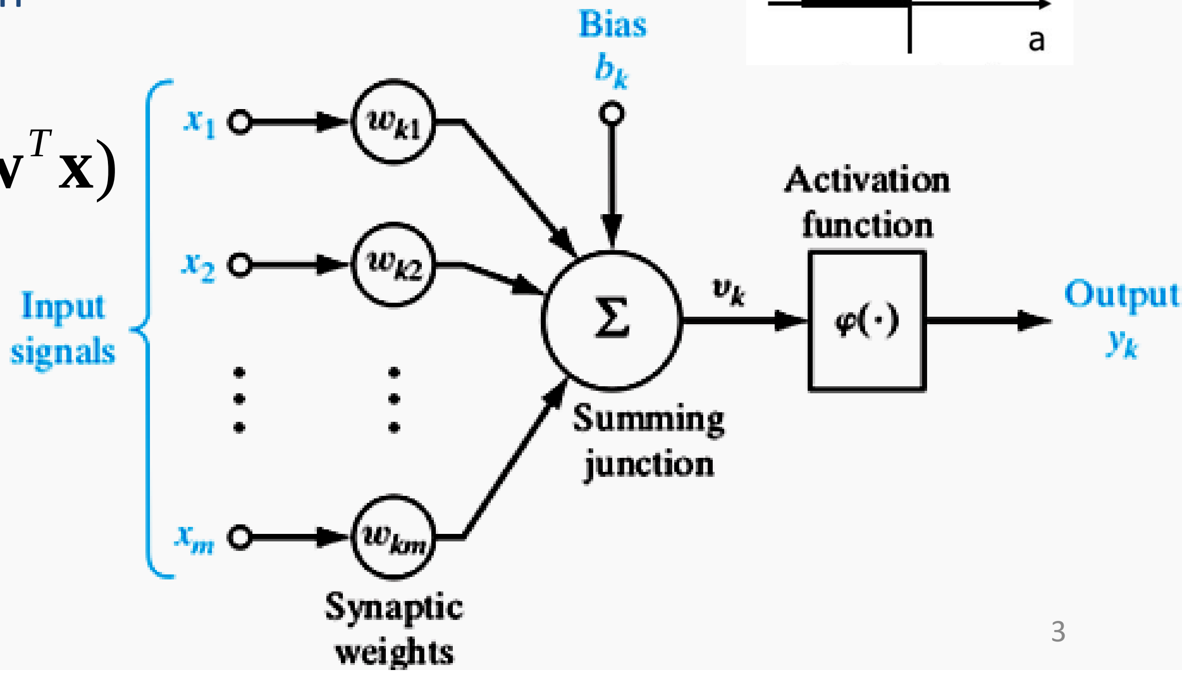
- Recall
  - Single-layer perceptron and its learning method
- Multilayer perceptron
  - Topology
  - Operation
- Representation
- Blum and Li construction
- Learning
  - Back-propagation

Single-layer Perceptron

- Receives input through its synapses (x_i)
- Synapses are weighted (w_i) (including bias)
- A weighted sum is calculated
- Nonlinear activation function



$$y_k = \varphi\left(\sum_{i=0}^{m} w_{ki} x_i\right) = \varphi(\mathbf{w}^T \mathbf{x})$$



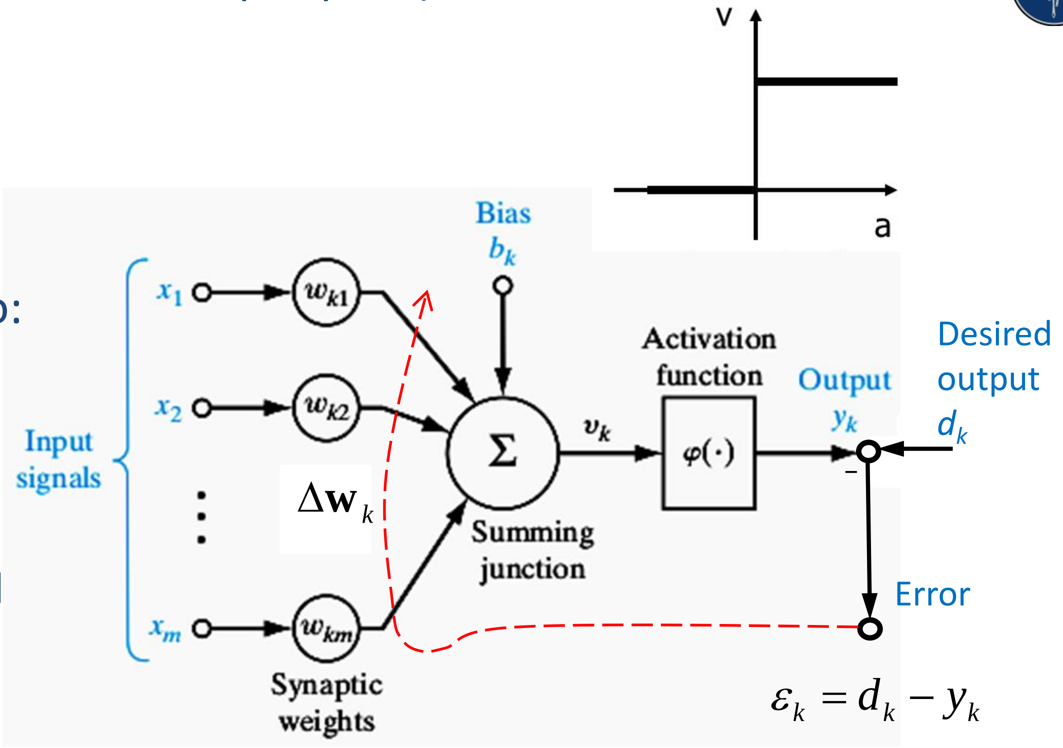
x_i : input vector
w_{ki} : weight coefficient vector
v_k : weighted sum
b_k : bias value of neuron k
o_k : output value of neuron k

Single-layer perceptron training: *Error correction*

- Had a training set (known input desired output pairs)
  - $x_i \rightarrow d_i$
- Apply the input vector (x_i)
- Calculate the output
- If output is false
- Modify the weights according to:

$$\Delta \mathbf{w}_k = \eta \, \varepsilon_k \mathbf{x}_k$$

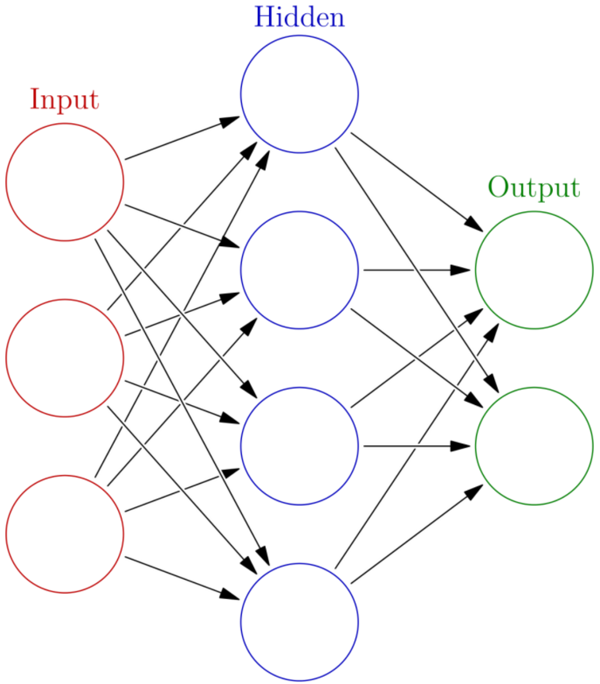
- Operation:
  - When error is positive the contribution of $w_{ki}x_i$ should be increased
- Convergence is proven in case of linearly separable task



Linear separability requirement is a major limitation of the single layer perceptron!

Multilayer perceptron

- Different names of Multilayer perceptron
  - Feed forward neural networks (FFNN)
  - Fully connected neural networks
- Multilayer neural network
  - Input layer
  - Hidden layers (one or multiple)
  - Output layer
  - The outputs are the inputs of the next layer
  - Many hidden layers → deep network
- Multiple inputs, multiple outputs
- The output is typically not binary
- Used practically in all deep neural networks!



Can solve linearly non-separable problems!

Topology and naming

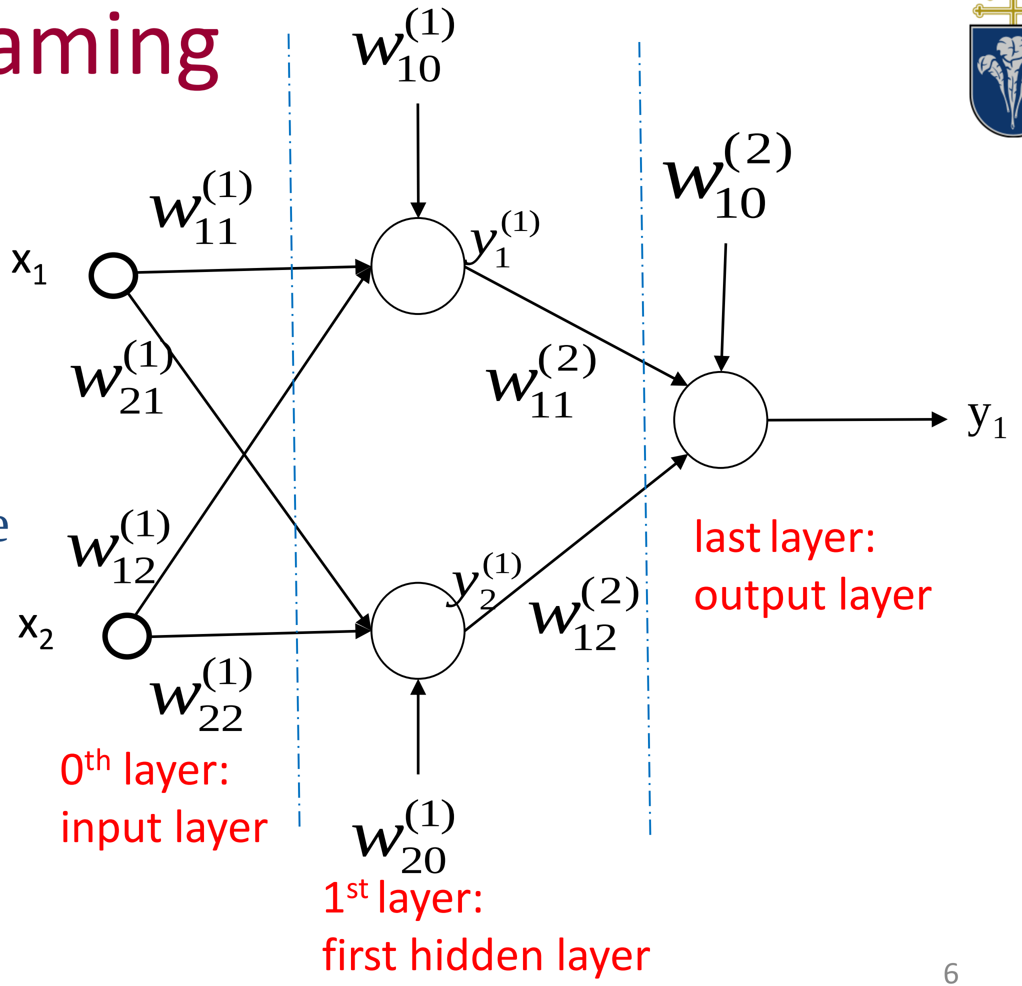
- Weights: $w_{ij}^{(l)}$
  - Arrives to the l^{th} layer
  - Comes from the j^{th} neuron from the $(l\text{-}1)^{th}$ layer
  - Arrives to the i^{th} neuron of the l^{th} layer

$w_{ij}^{(l)}$

destination layer

Destination neuron

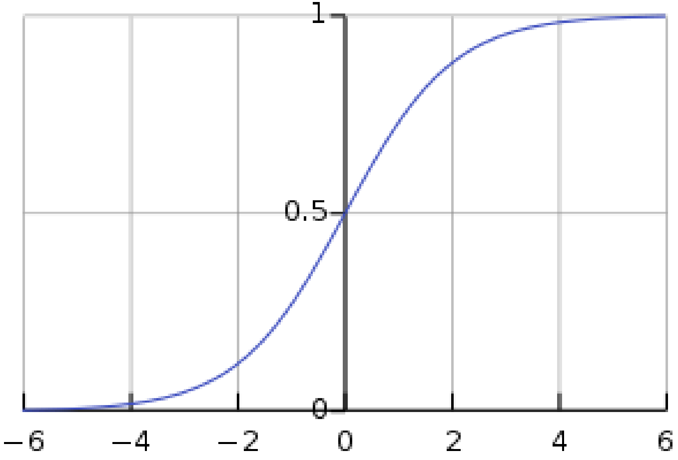
source neuron



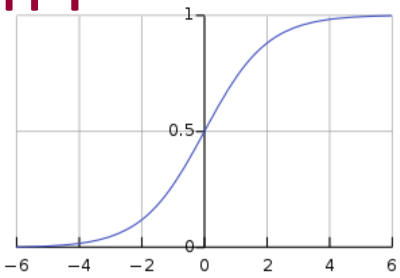
Activation function I

- Sigmoid function
  - Continuous
  - Continuously differentiable
  - It is used in the output layer of the fully connected neural network

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



Derivative of sigmoid function I



$$Sigmoid(x) = \frac{1}{1+e^{-x}} \qquad \frac{d}{dx}S(x) = \frac{d}{dx}\frac{1}{1+e^{-x}}$$

quotient rule:

$$\frac{d}{dx}f = \frac{(denominator * \frac{d}{dx}numerator) - (numerator * \frac{d}{dx}denominator)}{denominator^2}$$

$$\frac{d}{dx}S(x) = \frac{(1+e^{-x})(0)-(1)(-e^{-x})}{(1+e^{-x})^2}$$

$$\frac{d}{dx}S(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

This is the correct result,
but it is not in a nice form.

Derivative of sigmoid function II

$$\frac{d}{dx}S(x) = \frac{e^{-x}}{(1+e^{-x})^2} \Rightarrow \frac{d}{dx}S(x) = \frac{1-1+e^{-x}}{(1+e^{-x})^2}$$

$$\frac{d}{dx}S(x) = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$

reduction

$$\frac{d}{dx}S(x) = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2}$$

Multiply out

$$\frac{d}{dx}S(x) = \frac{1}{(1+e^{-x})}(1 - \frac{1}{1+e^{-x}})$$

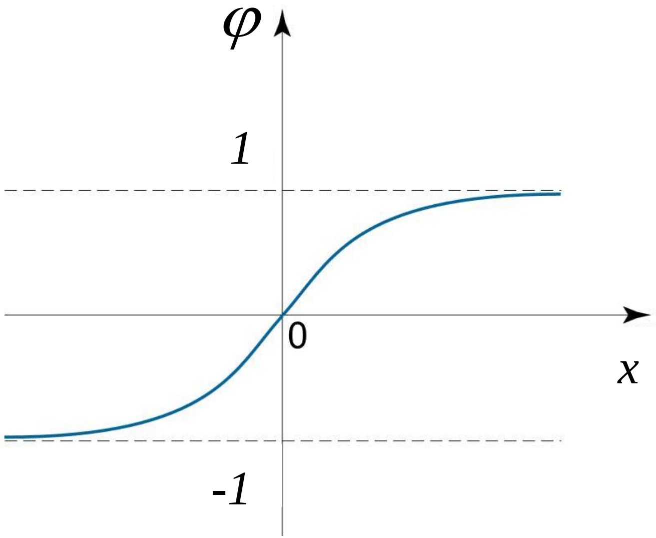
$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}S(x) = S(x)(1-S(x))$$

Much nicer form!

Activation function II

- Hyperbolic tangent function
  - Continuous
  - Continuously differentiable
  - It is used in the output layer of the fully connected neural network



$$\varphi(x) = \tanh(x)$$

$$\frac{d}{dx}\varphi(x) = 1 - \tanh^2(x) = (1 - \tanh(x))\,(1 + \tanh(x))$$

Activation function III

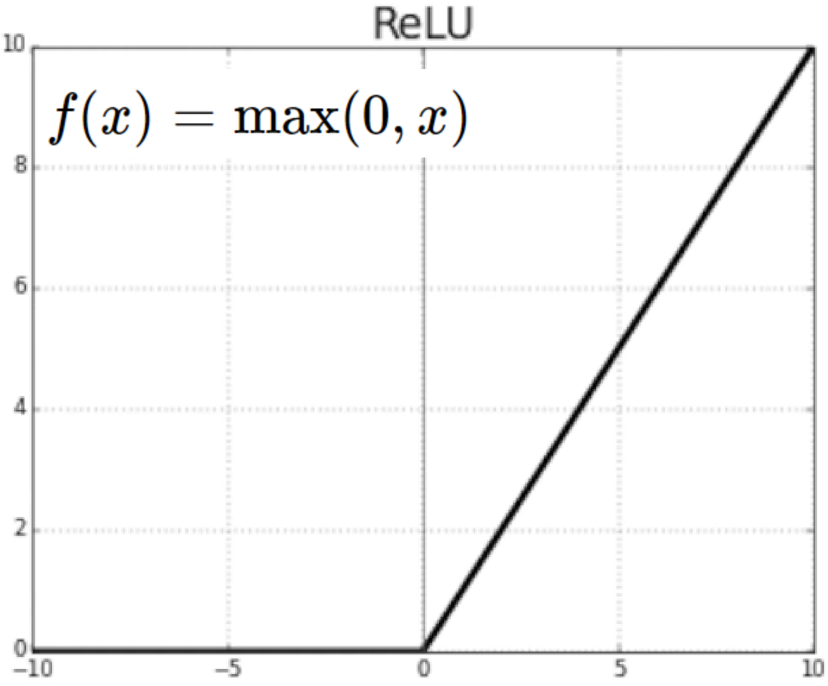
- Rectified Linear Unit (ReLU)

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

  - Most commonly used nonlinearity in hidden layers of deep neural networks

- Derivative of ReLU

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



Operation

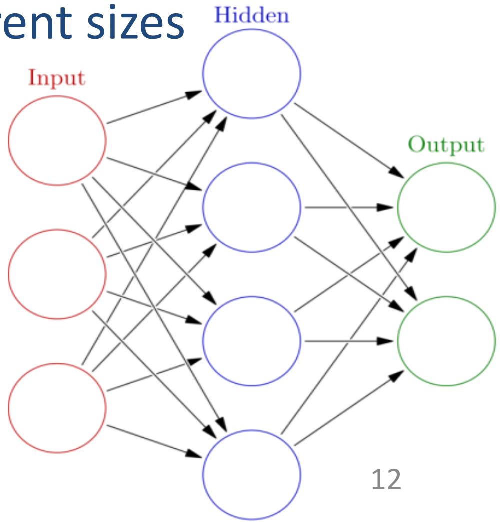
- Signal flows through the network progresses left to right

- The output of the network:

$$Net(\mathbf{x}, \mathbf{W}) = \varphi^{(L)}\left(\boldsymbol{w}^{(L)}\varphi^{(L-1)}\left(\boldsymbol{w}^{(L-1)} \quad \dots \quad \varphi^{(2)}\left(\boldsymbol{w}^{(2)}\varphi^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}\right)\right)\right)\right)$$

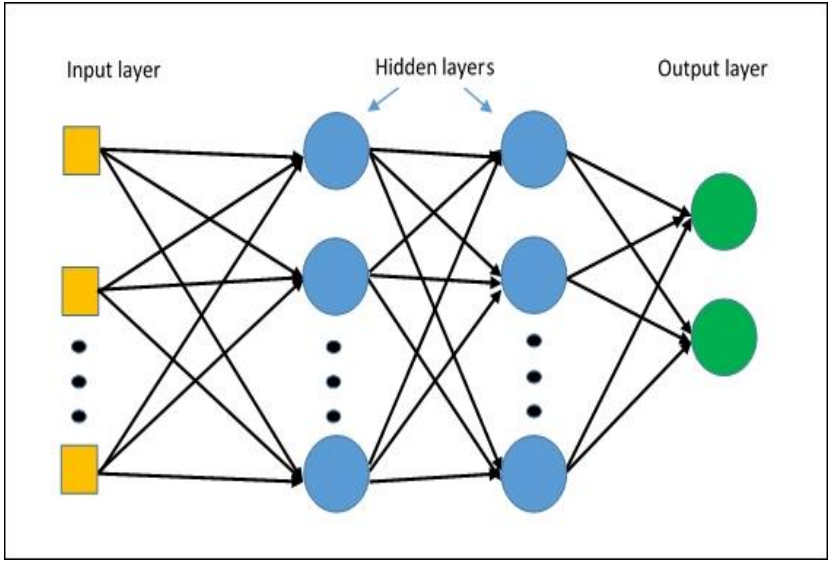
- Where the weights are matrices at each layer with different sizes

$$\boldsymbol{W}: \ \left(\boldsymbol{w}^{(L)}, \boldsymbol{w}^{(L-1)}, \dots \boldsymbol{w}^{(1)}\right)$$



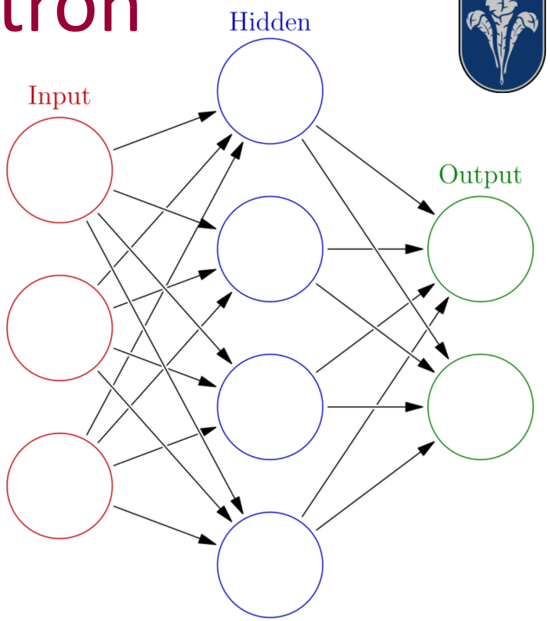
- Different activation functions for different layers
- Number of layers: L, neurons in l^{th} layer: n^l

Forward (signal) propagation



- Calculate the output of the first hidden layer
  $$\boldsymbol{y}^{(1)} = \varphi\left(\boldsymbol{w}^{(1)}\boldsymbol{x}\right)$$
- Calculate the output of the second hidden layer using the output of the first hidden layer as the input
  $$\boldsymbol{y}^{(2)} = \varphi\left(\boldsymbol{w}^{(2)}\boldsymbol{y}^{(1)}\right)$$
  $\boldsymbol{x},\boldsymbol{y}^{(k)}$ are vectors
  $\boldsymbol{w}^{(k)}$ are matrices
- . . .
  $$\boldsymbol{y}^{(L)} = \varphi\left(\boldsymbol{w}^{(L)}\boldsymbol{y}^{(L-1)}\right)$$
- Calculate the output of the output layer using the output of the last hidden layer as the input

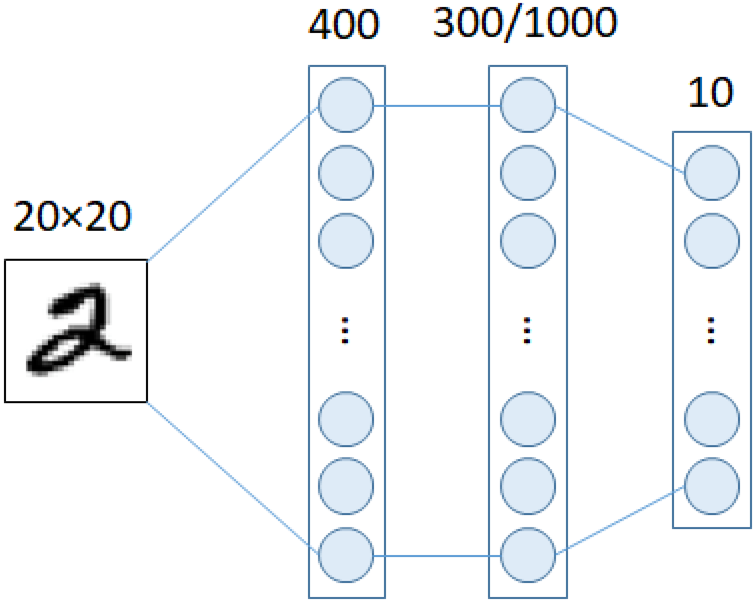
Usage of Multilayer Perceptron



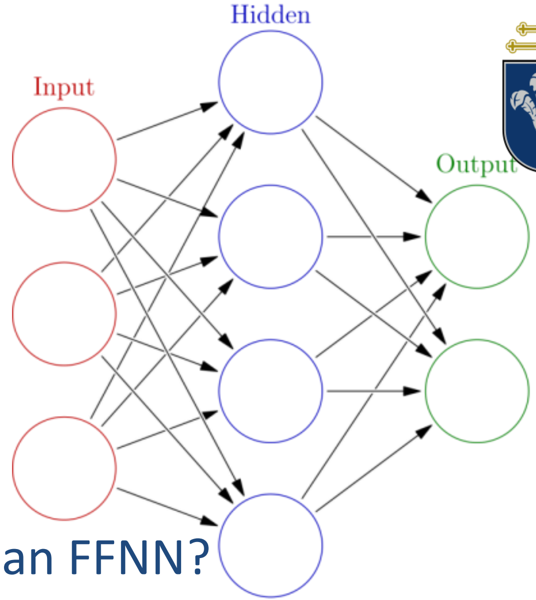
- Multilayer perceptrons are used for
  - Classification
    - Supervised learning for classification
    - Given inputs and class labels
  - Approximation
    - Approximate an arbitrary function with arbitrary precision

Classification example

- Classification of the hand written figures
  - MNIST data base: 20x20 binary images
  - The output is a one of ten code



Approximation



- When solving engineering task by FFNN we are faced with the following theoretical questions:

1. Representation
   – What kind of functions can be Approximated by an FFNN?
2. Learning
   – How to set up the weights to solve a specific task?
3. Generalization
   – If only limited knowledge is available about the task which is to be solved, then how the FFNN is going to generalize this knowledge?

Approximation (Representation)

- Can it approximate all the function?
- With what precision?

$$\left.\begin{array}{c}\forall F(\mathbf{x}) \in \mathcal{F} \\ \varepsilon > 0\end{array}\right\} \rightarrow \exists \mathbf{w} : \left\| F(\mathbf{x}) - Net(\mathbf{x}, \mathbf{w}) \right\| < \varepsilon$$

- The notation || || defines a norm used in $\mathcal{F}$ space

$$\int \underset{\mathbf{X}}{\cdots} \int \left(F(\mathbf{x}) - Net(\mathbf{x}, \mathbf{w}) \right)^p \mathbf{d}x, \ldots \mathbf{d}x_N < \varepsilon$$

Representation – Theorem 1

- Theorem (Harnik, Stinchambe, White 1989)
- Every function in L^p can be represented arbitrarily closely approximation by a neural net
- More precisely for each

$$F(x) \in L^p$$
$$\forall \varepsilon > 0, \exists \mathbf{w}$$
$$\int \underset{\mathbf{x}}{\cdots} \int \left(F(\mathbf{x}) - Net(\mathbf{x}, \mathbf{w}) \right)^p \mathbf{d}x, \ldots \mathbf{d}x_N < \varepsilon$$

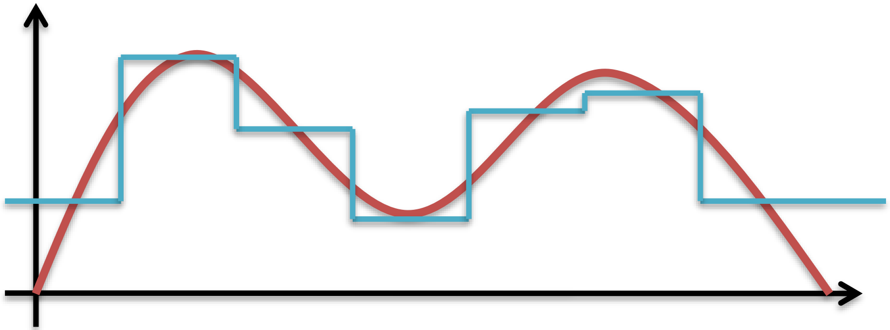
Recall:

$$L^1 : \int \underset{\mathbf{x}}{\cdots} \int \left(F(x) \right) \mathbf{d}x, \ldots \mathbf{d}x_N < \infty$$
$$L^2 : \int \underset{\mathbf{x}}{\cdots} \int \left(F(x) \right)^2 \mathbf{d}x, \ldots \mathbf{d}x_N < \infty$$
$$L^p : \int \underset{\mathbf{x}}{\cdots} \int \left(F(x) \right)^p \mathbf{d}x, \ldots \mathbf{d}x_N < \infty$$

- Since it is out of the focus of the course this proof will not be presented here

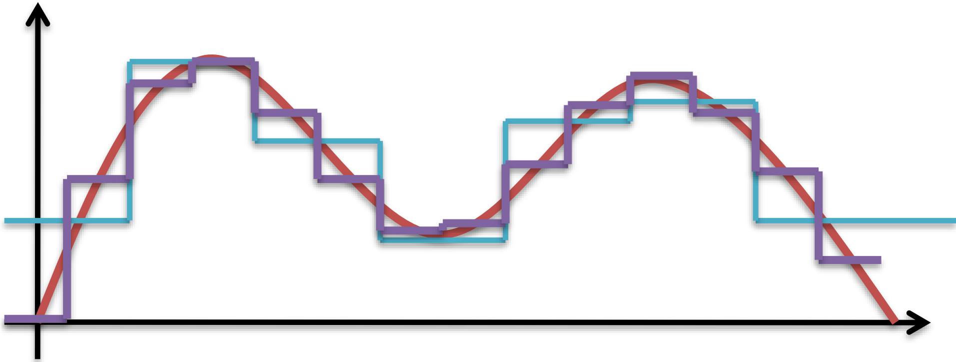
Representation – Blum and Li theorem

- Theorem: $F(x) \in L^2$
  $\forall \varepsilon > 0, \exists \mathbf{w}$
  $$\int \cdots_{\mathbf{x}} \int \left(F(\mathbf{x}) - Net(\mathbf{x}, \mathbf{w}) \right)^2 \mathbf{d}x, \ldots \mathbf{d}x_N < \varepsilon$$
- Proof:
  - Using the step functions: S
  - From elementary integral theory it is clear every function can be approximated by appropriate step function sequence



Representation – Blum and Li theorem

- From elementary integral theory it is clear every function can be approximated by appropriate step function sequence
  - The step function can have arbitrary narrow steps
  - For example each step could be divided into two sub-steps
  - Therefore we can synthetize a function with arbitrary precision

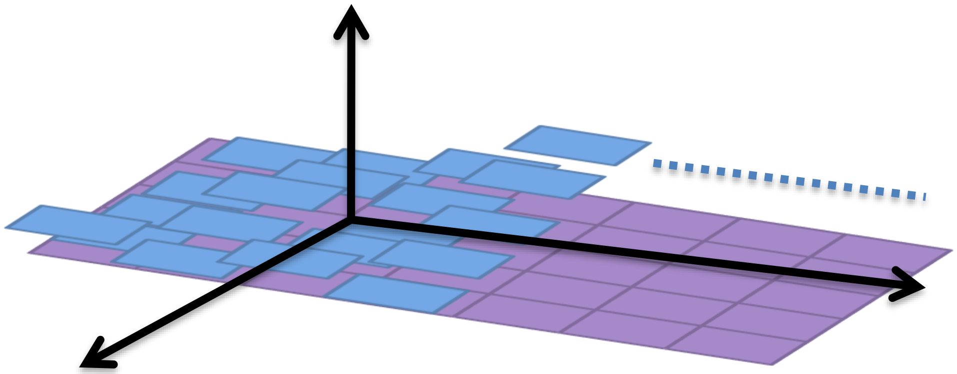


$$I(X)=\begin{cases}1 & \text{if } \mathbf{x}\in X\\ 0 & \text{else}\end{cases}$$

$$F(x)\cong\underbrace{\sum_i F(x_i)I(x_i)}_{s(x)}$$

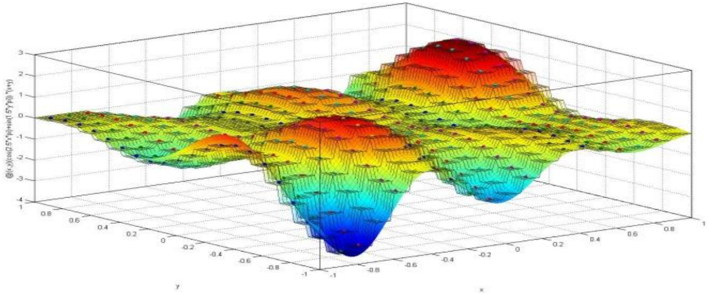
Representation – Blum and Li construction

- This construction …
  - … has no dimensional limits
  - … has no equidistance restrictions on tiles (partitions)
  - … can be further fined, and the approximation can be any precise
- 2 dimensional example
  - The tiles are the top of the columns for each approximation cell



Blum and Li – Limitations

- The size of the FFNN constructed via this method is quite big
- Consider the task on the picture, where there are 1000 by 1000 cell to approximate the function



- General case:
  ~2 Million neurons are needed
- Smoother approximation needs more
- The network architecture is synthetized (constracted), the weights are generated
- We are after to find a less complicated architectures

Learning

$$\mathbf{w}_{\mathrm{opt}} : \min_{\mathbf{w}} \left\| \mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right\|^2 = \min_{\mathbf{w}} \int .. \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right)^2 dx_1 \ldots dx_N$$

- Nor minimization task neither construction is possible most cases
  - Complete information would be needed about F(x), however it is typically unknown
  - Known in the input-output pairs only (limited positions in input space)
- Weak learning in incomplete environment, instead of using F(x)

$$\tau^{(K)} = \left\{ \left(\mathbf{x}_k , d_k \right); k = 1, \ldots, K \right\}$$

- A training set is being constructed of observations

Learning

- Rather than minimizing the error function

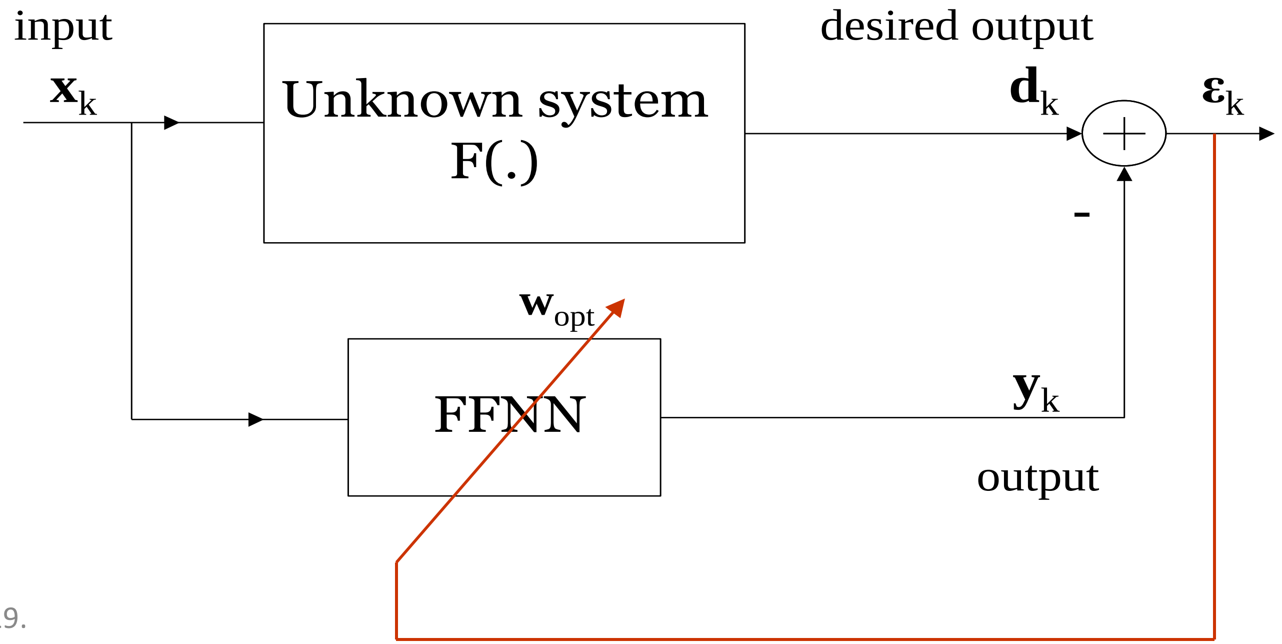
$$\mathbf{w}_{\text{opt}} : \min_{\mathbf{w}} \left\| \mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right\|^2 = \min_{\mathbf{w}} \int .. \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right)^2 dx_1 ... dx_N$$

- The approximation is the best achievable
  - F function is known in a limited positions (training set)

$$\mathbf{w}_{\text{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \left(d_k - Net(\mathbf{x}_k, \mathbf{w}) \right)^2$$

Learning

$$\mathbf{w}_{opt} : \min_{\mathbf{w}} \left\| \mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right\|^2 = \cancel{\min_{\mathbf{w}} \int..\int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right)^2 dx_1 \ldots dx_N}$$



Learning

- The questions are the following
  - What is the relationship of these optimal weights?

$$\mathbf{w}_{\mathrm{opt}} \overset{???}{\Leftrightarrow} \mathbf{w}_{\mathrm{opt}}^{(K)}$$

$$\mathbf{w}_{\mathrm{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \left(d_k - Net\left(\mathbf{x}_k, \mathbf{w}\right)\right)^2$$

  - How this new objective function should be minimized as quickly as possible?

Statistical learning theory

- Empirical error

$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

- Theoretical error

$$\left\|\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w})\right\|^2 = \int_X \cdots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w})\right)^2 dx_1 \ldots dx_N$$

- Let us have $\mathbf{x}_k$ random variables subject to uniform distribution

Statistical learning theory

- $\mathbf{x}_k$ random variable, where d=F($\mathbf{x}$)

$$\lim_{k\to\infty} = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net\left(\mathbf{x}_k, \mathbf{w}\right)\right)^2 = \mathrm{E}\left(d - Net(\mathbf{x},\mathbf{w})\right)^2 =$$

$$\int_X \cdots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}\left(\mathbf{x},\mathbf{w}\right)\right)^2 p(\mathbf{x})dx_1 \ldots dx_N =$$

Because it is ~ constant due to the uniformity

$$\frac{1}{|X|}\int_X \cdots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}\left(\mathbf{x},\mathbf{w}\right)\right)^2 dx_1 \ldots dx_N \;\square$$

$$\int_X \cdots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}\left(\mathbf{x},\mathbf{w}\right)\right)^2 dx_1 \ldots dx_N$$

Statistical learning theory

- Therefore

$$\underset{K\to\infty}{\text{l.i.m.}}\,\mathbf{w}_{\text{opt}} = \mathbf{w}_{\text{opt}}^{(K)}$$

- Where l.i.m. means: lim in mean

$$\lim_{K\to\infty} R_{emp}\left(\mathbf{w}\right) = R_{th}\left(\mathbf{w}\right)$$

$$\lim_{K\to\infty}\frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net\left(\mathbf{x}_k,\mathbf{w}\right)\right)^2 = \int_X \dots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}\left(\mathbf{x},\mathbf{w}\right)\right)^2 dx_1 \dots dx_N$$

Weak learning is satifactory!

Learning – in practice

- Learning based on the training set:

$$\tau^{(K)} = \{(\mathbf{x}_k, d_k); k = 1, \ldots, K\}$$

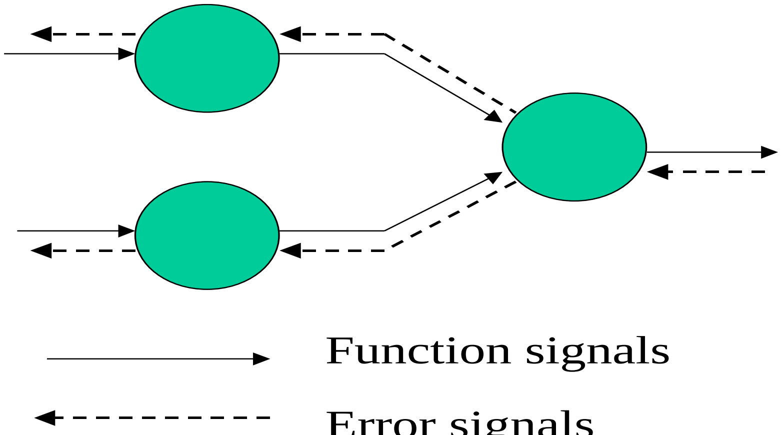
- Minimize the empirical error function (R_{emp})

$$\mathbf{w}_{\text{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \underbrace{\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2}_{E_k} = \min_{\mathbf{w}} R_{emp}(\mathbf{w})$$

- Learning is a multivariate optimization task

Learning

- The Rosenblatt algorithm is inapplicable,
  - the error and desired output in the hidden layers of the FFNN **is unknown**
- Someway the error of the whole network has to be distributed to the internal neurons, in a feedback way



Forward propagation of function signals and back-propagation of errors signals

Sequential back propagation

- Adapting the weights of the FFNN (recursive algorithm)

$$w_{ij}^{(l)}(k+1) = w_{ij}^{(l)}(k) + \Delta w_{ij}^{(l)}(k)$$

$$\Delta w_{ij}^{(l)}(k) = ?$$

- The weights are modified towards the differential of the error function (delta rule):

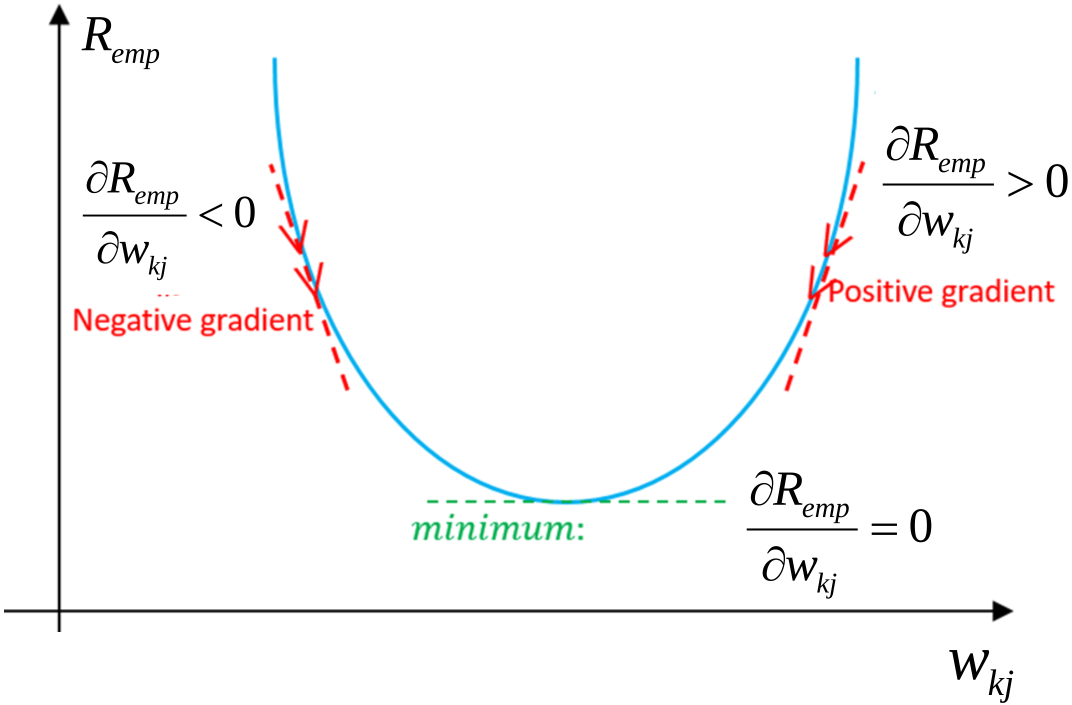
$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial R_{emp}}{\partial w_{ij}^{(l)}}$$

- The elements of the training set adapted by the FFNN sequentially

$$R_{emp} = R_{emp}(y(\mathbf{x}), d)$$

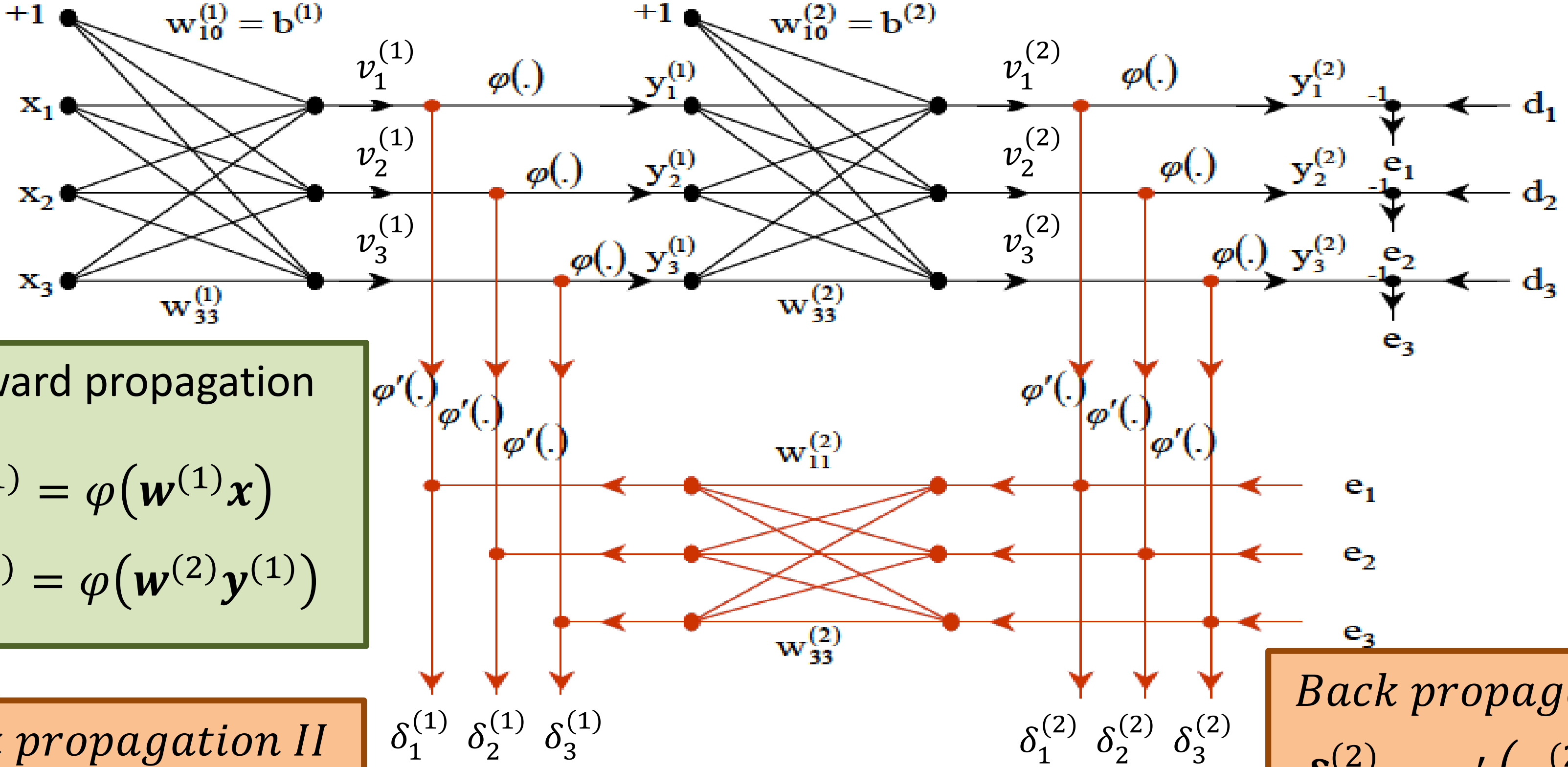
Delta (learning) rule

- If $\frac{\partial R_{emp}}{\partial w_{kj}} < 0$ than we have to increas w_{kj}, to get closer to the minimum.
  - $\Delta w_{kj} = -\eta \frac{\partial R_{emp}}{\partial w_{kj}}$
- If $\frac{\partial R_{emp}}{\partial w_{kj}} > 0$ than we have to decrease w_{kj}, to get closer to the minimum.
  - $\Delta w_{kj} = -\eta \frac{\partial R_{emp}}{\partial w_{kj}}$



η: learning rate parameter

Propagation and back propagation



Forward propagation

$$\boldsymbol{y}^{(1)} = \varphi\left(\boldsymbol{w}^{(1)}\boldsymbol{x}\right)$$

$$\boldsymbol{y}^{(2)} = \varphi\left(\boldsymbol{w}^{(2)}\boldsymbol{y}^{(1)}\right)$$

Back propagation I

$$\boldsymbol{\delta}_i^{(2)} = \varphi'\left(v_i^{(2)}\right) e_i$$

Back propagation II

$$\boldsymbol{\delta}_j^{(1)} = \varphi'\left(v_j^{(2)}\right) \sum_k \delta_k^{(2)} w_{kj}^2 = \varphi'\left(v_j^{(2)}\right) \boldsymbol{\delta}^{(2)} \boldsymbol{w}_j^2$$

Weight update: $\Delta w_{ji}^{(L)} = \eta \delta_j^{(L)} y_i^{(L-1)}$

Back-propagation

- Though we showed how to modify the weights with back propagation, its most important value that it can calculate the gradient
- The weight updates can be calculated with different optimization methods, after the gradients are calculated
- Various optimization method can drastically speed up the training (100x, 1000x)

Conclusion

- For known functions (according to Blum-Li)
  - One can define a Neural Network architecture
  - And generate the weights
  - That it can represent the known function with arbitrary precision

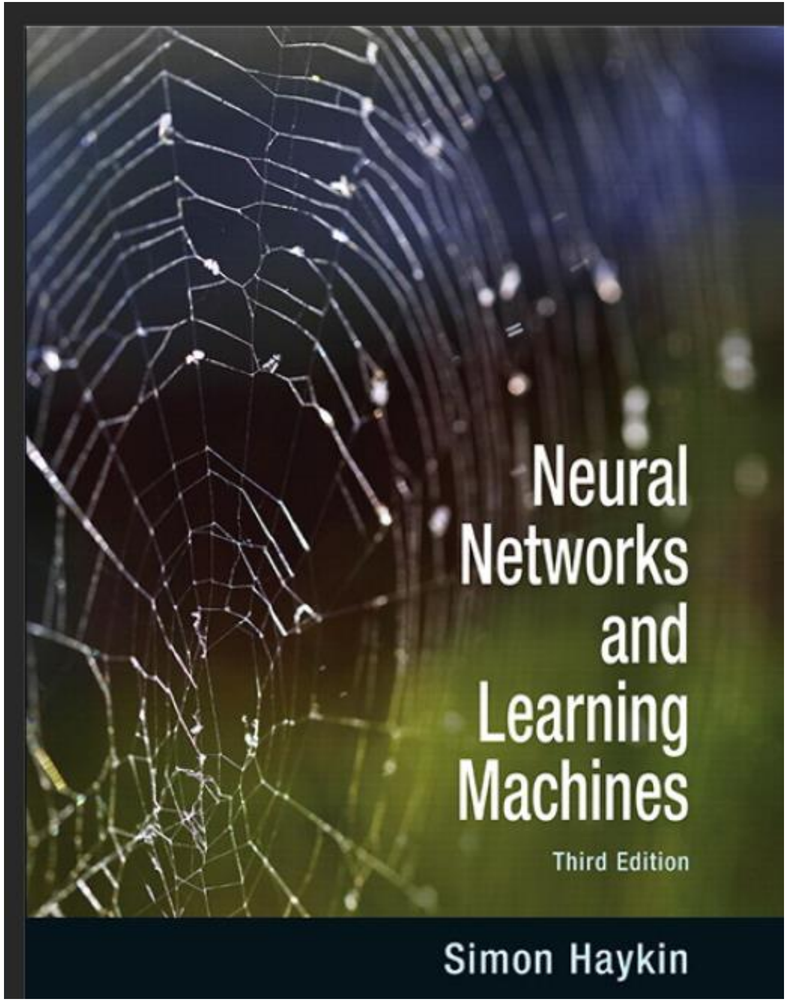
- For unknown but existing function defined by IO pairs (according to statistic learning)
  - One can find a Neural Network architecture
  - And train the network (optimize the weights)
  - Reach arbitrary precision with high number of IO pairs
  - The trained network will be able to well predict previously unknown IO pairs (generalization)

Implementing Neural Computing

- For a given task
  - Find large representative annotated data set
  - Find a suitable network architecture
    - Number of layers, neurons, activations, interconnection patterns
  - Find a learning/training method
    - Converges in acceptable time

Literature

- Simon Haykin: **Neural Networks: A Comprehensive Foundation**
- **Page 129-141**



Neural Networks

(P-ITEEA-0011)

Gradient based optimization methods

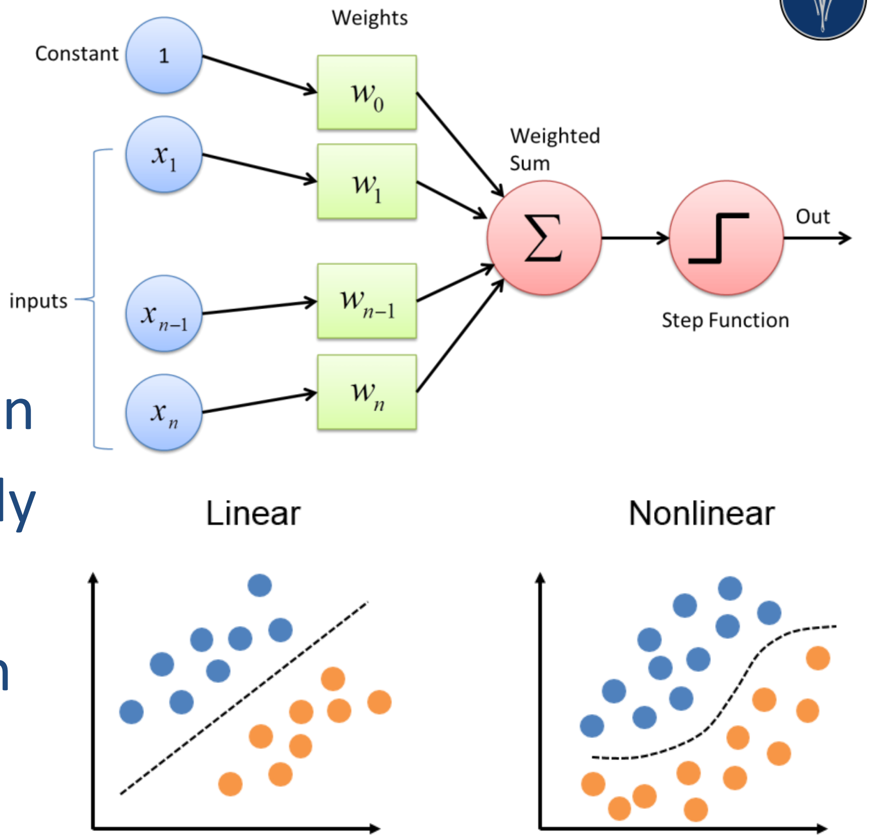
Akos Zarandy
Lecture 4
October 1, 2019

Contents

- Recall
  - Single- and multilayer perceptron and its learning method
- Mathematical background
- Simple gradient based optimizers
  - 1st and 2nd order optimizers
- Advanced optimizers
  - Momentum
  - ADAM

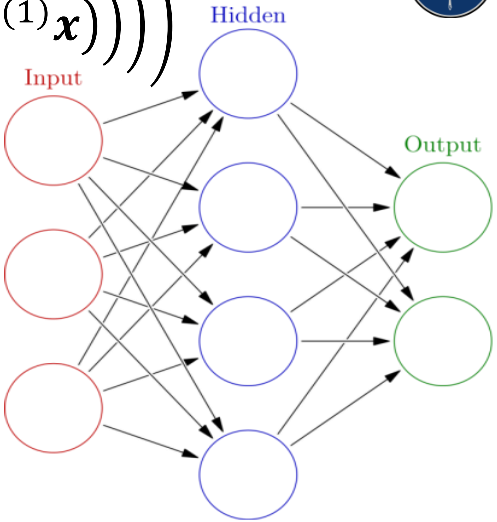
Recall: Single layer perceptron

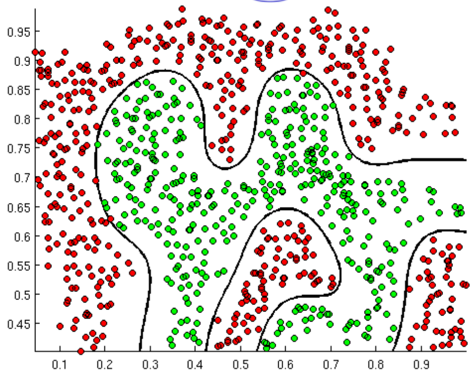
- $y = \varphi(\mathbf{w}^T \mathbf{x})$
- Decision boundary is a hyperplan
- Simple training method
- Convergence of training was proven
- Good for making decision in linearly separable cases
- In more complex decision situation
  - It turns out to be a toy



Recall: Multi-layer perceptron

- $Net(\mathbf{x}, \mathbf{W}) = \varphi^{(L)}\left(\boldsymbol{w}^{(L)}\varphi^{(L-1)}\left(\boldsymbol{w}^{(L-1)} \quad \ldots \quad \varphi^{(2)}\left(\boldsymbol{w}^{(2)}\varphi^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}\right)\right)\right)\right)$
- Can approximate an arbitrary function with arbitrary precision
- The same way, it can implement arbitrary decision boundary
- It can be trained even if F (or the boundary surface) is not known analytically or not even fully known
  - *Statistical learning:* It is enough to know equally distributed input/output pairs
- The partial gradient of the network can be also calculated for each weight coefficient or hidden layer neuron (back propagation)





What is learning (training)?

- Given:
  - Definition of the network architecture
    - Topology
    - Initial weights
    - Activation functions (nonlinearities)
    - Training set ($\boldsymbol{x}_i \rightarrow \boldsymbol{y}_i$)
- Goal:
  - Calculation of the optimal weight composition: W_{opt}

1. *Having a function to approximate*

$$\mathbf{w}_{\text{opt}} : \min_{\mathbf{w}} \left\| \mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right\|^2 = \min_{\mathbf{w}} \int .. \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right)^2 dx_1 ... dx_N$$

2. *Having a set of observations from a stochastic process*

$$\mathbf{w}_{\text{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \left(d_k - Net(\mathbf{x}_k, \mathbf{w}) \right)^2$$

Stochastic process is a process, where we cannot observe the exact values. In these processes, our observations are always corrupted with some random noise.

OPTIMIZATION!!!

Optimization

- Given an ***Objective function*** to optimize
  - Also called: Error function, Cost function, Loss function, Criterion
  - Derived from the network topology and the input/output pairs
- Function types:
  - Quadratic, in case of regression (stochastic process)

$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

  - Conditional log-likelihood, in case of classification (classification process)
    - The sum of the negative logarithmic likelihood (probability) is minimized

$$\Theta(\mathbf{w}) = \sum_{k=1}^{K} -logP(\mathbf{y_k}|\mathbf{x_k};\mathbf{w})$$

Optimizations

- *Here we always minimize the objective function*
  - *Parametric equation*
    - **x** *are the variables*
    - **w** *are the parameters*
- *Optimization targets to find the optimal weights*

$$\mathbf{w}_{opt} = \min f\left(\mathbf{x}, \mathbf{d}, Net(\mathbf{x},\mathbf{w})\right)$$

goals:

- *Acceptable error level*
- *Acceptable computational time assuming reasonable computational effort*

Mathematics behind: Function analysis

- Assumptions
  - Poor conditioning
  - Conditioning number (Ratio of Eugen values): $\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$

$$f(x) = A^{-1}x \qquad A \in \Re^{n \times n}$$

  - Applied functions should be Lipschitz continuous or have Lipschitz continuous derivate

$$\forall x, \forall y, \ \left| f(x) - f(y) \right| \leq \mathrm{L} \left\| \mathrm{x} - \mathrm{y} \right\|_2$$

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output. (e.g. Matrix inversion)

(where:
L is the Lipschitz constant)

Basic idea of Gradient Descent

- There is a function, where

$$f(x)$$

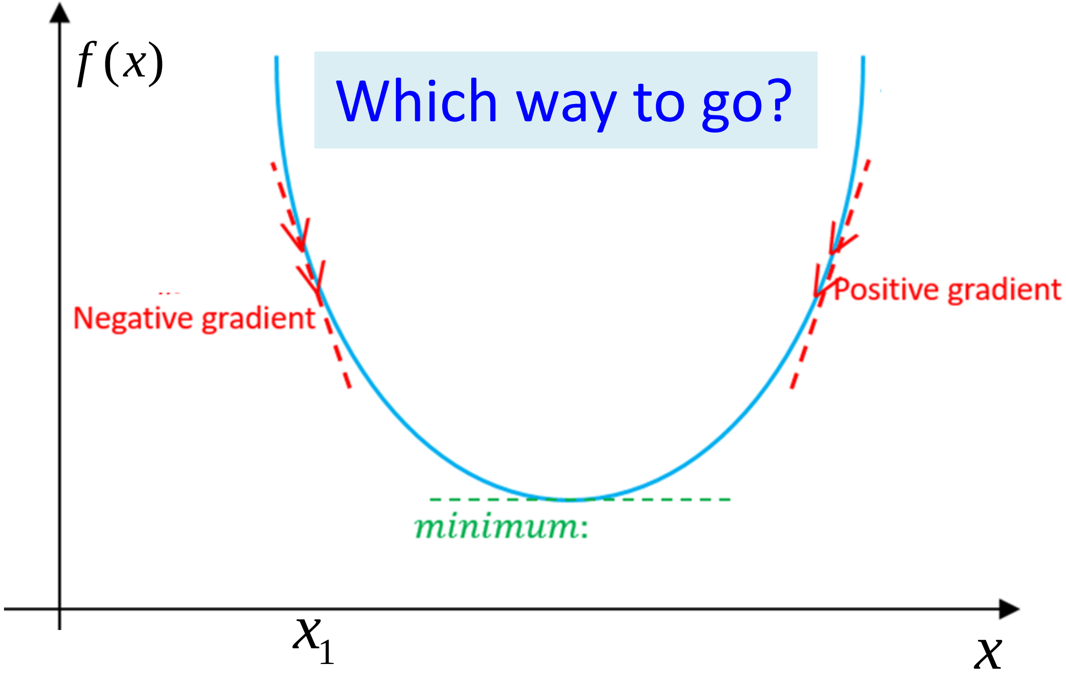
and

$$f'(x)$$

  - can be calculated at any points, but

$$f'(x) = 0$$

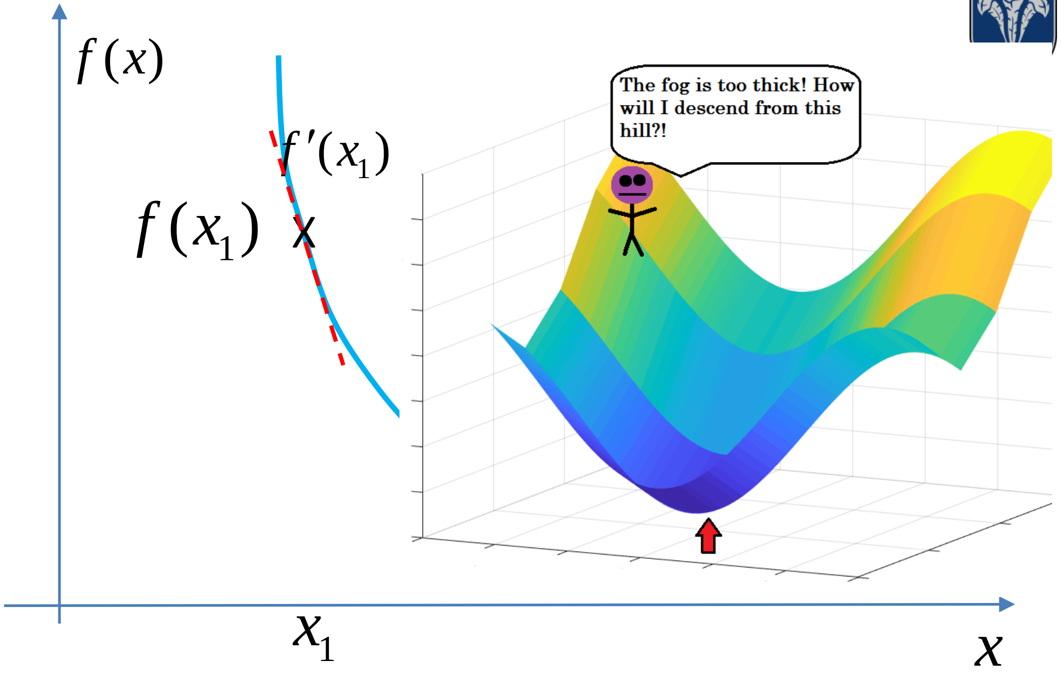
  - cannot.
  - Therefore the trace of the light blue line is not known.
  - We have to start out from one point (say x_1) and with an iterative method, we need to go towards the minimum



Basic idea of Gradient Descent

- We do not know where the curve is
- We know the value at $f(x_1)$
- We know the derivative at x_1

  $f'(x_1)$
- Which way to go?
- Idea: follow the descending gradient!



Basic idea of Gradient Descent

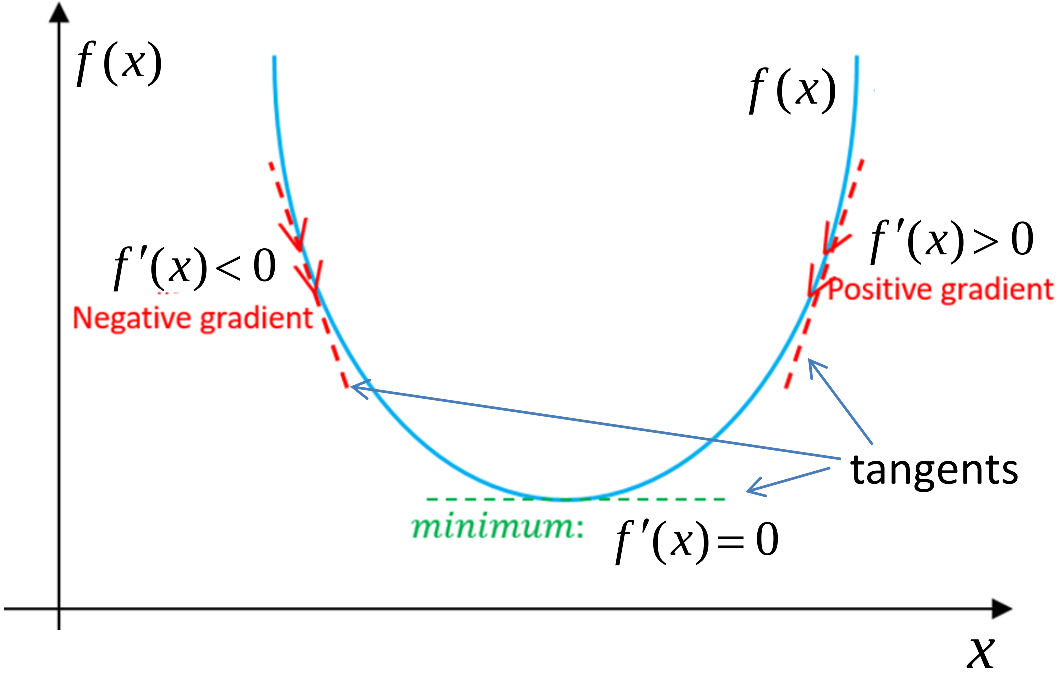
- Derivative means for small ε

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

- *therefore*

$$f\left(x-\varepsilon\, sign\left(f'(x)\right)\right) \leq f(x)$$

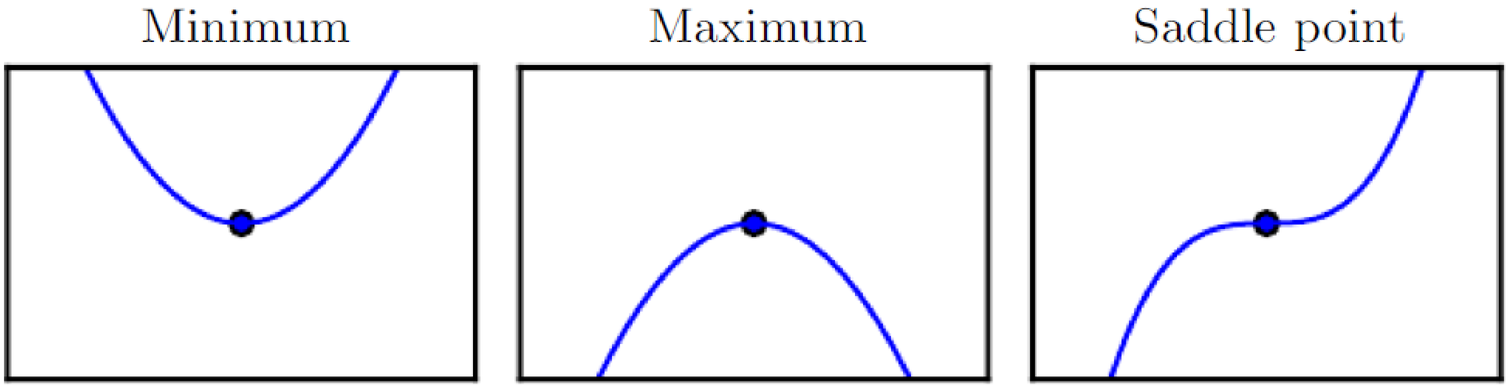
- This technique is called **Gradient Descent (Cauchy, 1847).**



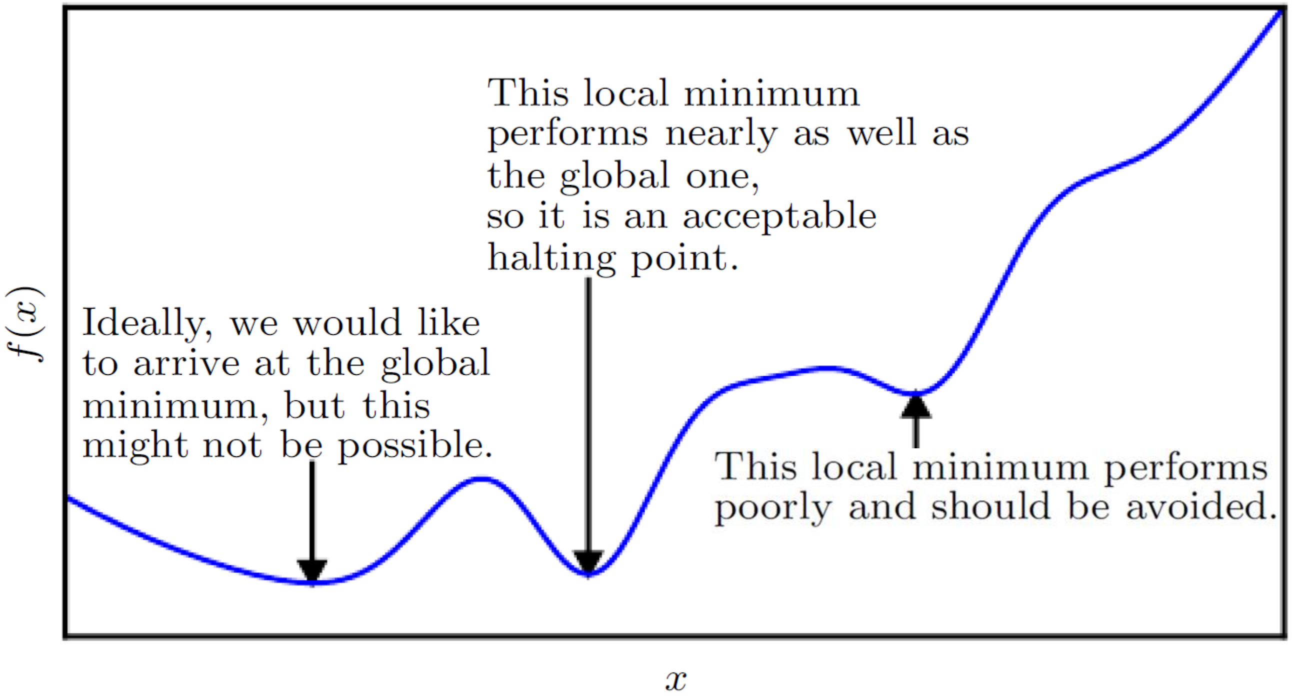
Optimization goal is to find the $f'(x)=0$ position.
(Critical or stationary points)

Stationary points

- Local minimum, where *f`(x)=0*, and *f(x)* is smaller than all neighboring points
- Local maximum, where *f`(x)=0*, and *f(x)* is larger than all neighboring points
- Saddle points, where *f`(x)=0*, and neither minimum nor maximum



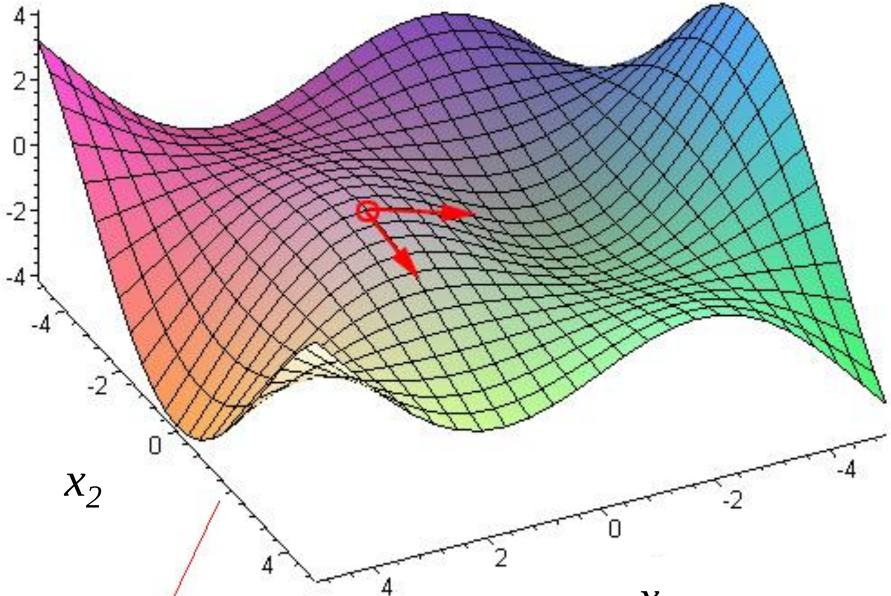
Local and global minimum



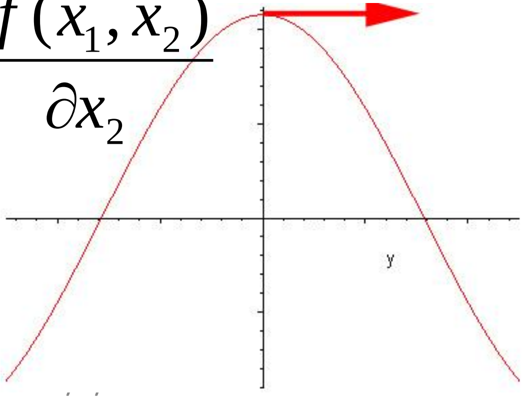
In neural network parameter optimization we usually settle for finding a value of f that is very low, but not necessarily minimal in any formal sense.

Multidimensional input functions I

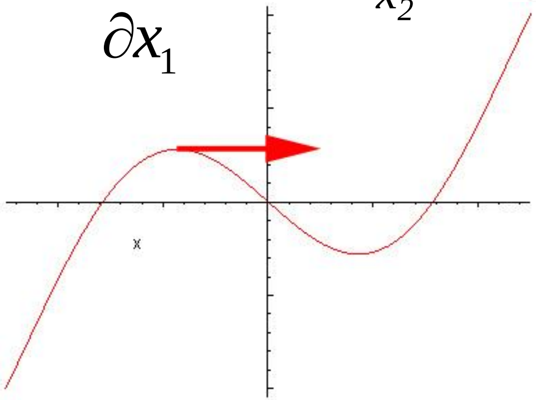
- In case of a vector scalar function
- In 2D, directional derivatives (slope towards x_1 and x_2):



$$\frac{\partial f(x_1, x_2)}{\partial x_2}$$



$$\frac{\partial f(x_1, x_2)}{\partial x_1}$$

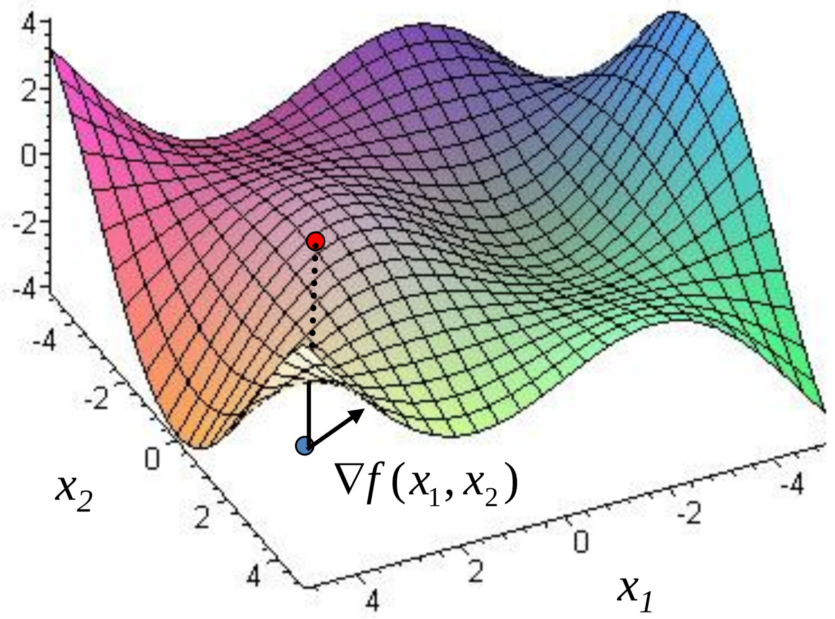


Multidimensional input functions II

- In case of a vector scalar function
- Gradient definition in 2D

$$f : R^2 \to R$$

$$\nabla f(x_1, x_2) := \begin{pmatrix} \dfrac{\partial f}{\partial x_1} & \dfrac{\partial f}{\partial x_2} \end{pmatrix}$$

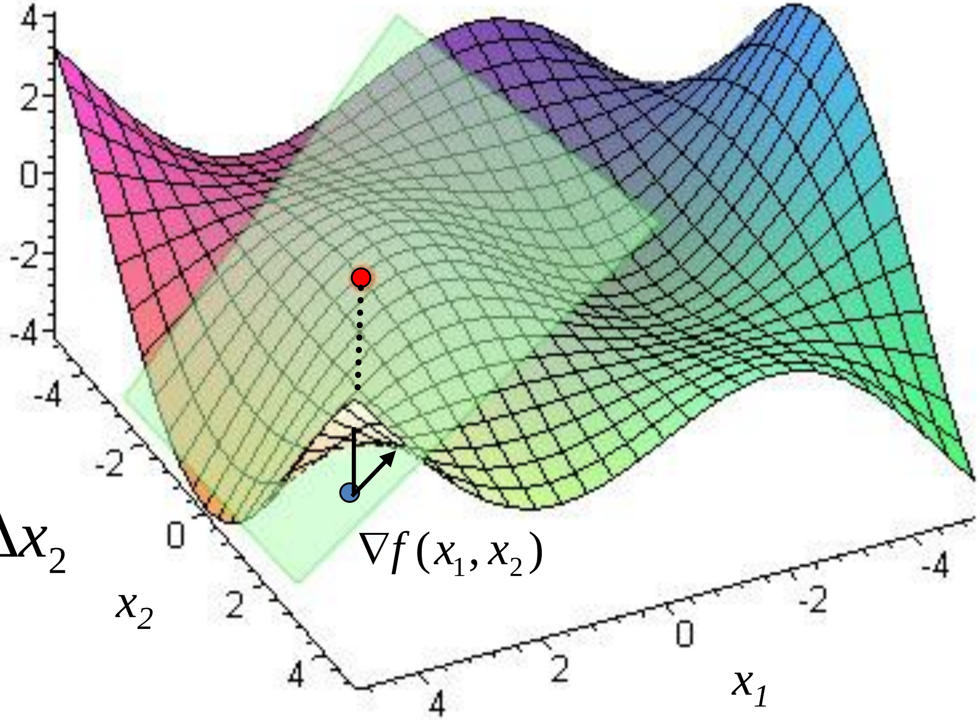


A vector in the in the x_1 - x_2 plane

Multidimensional input functions III

- The gradient defines (hyper) plane approximating the function infinitesimally at point $\mathbf{x}$ (x_1, x_2)

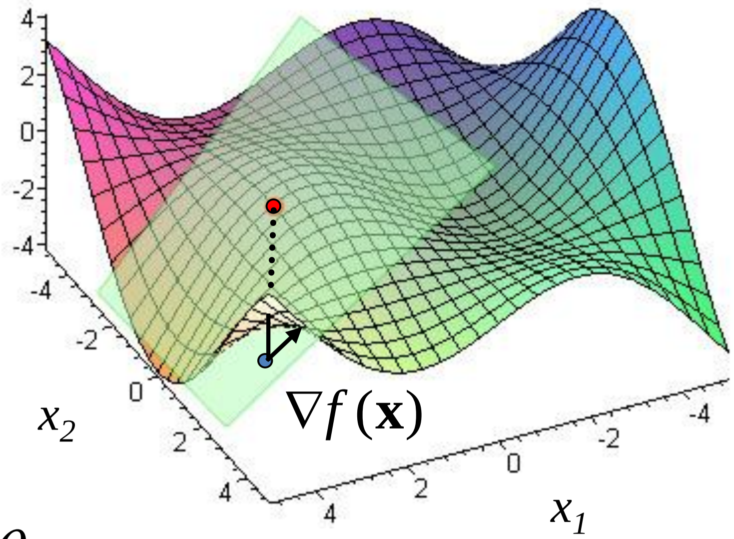
$$\Delta z = \frac{\partial f(x_1, x_2)}{\partial x_1} \cdot \Delta x_1 + \frac{\partial f(x_1, x_2)}{\partial x_2} \cdot \Delta x_2$$



Multidimensional input functions IV

- Directional derivative to an arbitrary direction **u** (**u** is unit vector) is the slope of f in that direction at point $\mathbf{x}$ (x_1, x_2):

$$\mathbf{u}^{\mathrm{T}}\nabla f(\mathbf{x})$$



- f decreases the fastest:

Not changing with u

$$\min_{u,u^{\mathrm{T}}u=1} \mathbf{u}^{\mathrm{T}}\nabla f(\mathbf{x}) = \min_{u,u^{\mathrm{T}}u=1} \|\mathbf{u}\|_2 \|\nabla f(\mathbf{x})\|_2 \cos\theta$$

minimum at 180

- **u** is opposite to the gradient!!!

New points towards steepest descent:

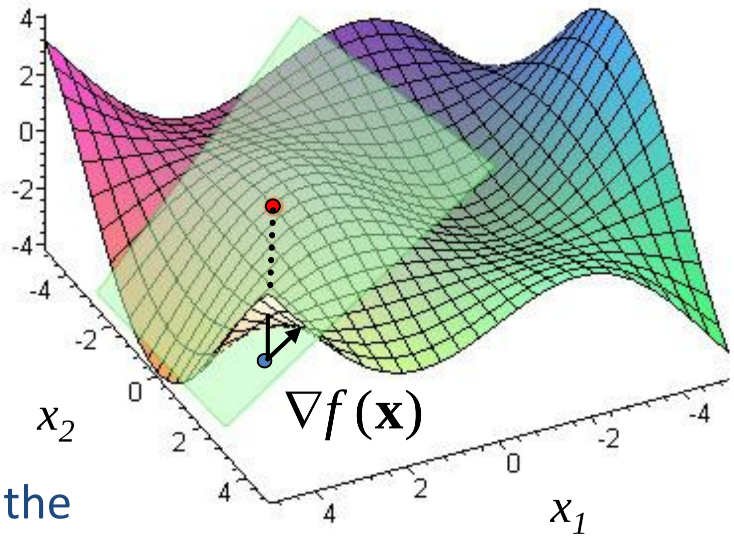
$$\mathbf{x}' = \mathbf{x} - \varepsilon\nabla f(\mathbf{x})$$

Gradient Descent in multidimensional input case

- Steepest gradient descent iteration

$$\mathbf{x}(n+1) = \mathbf{x}(n) - \varepsilon \nabla f\left(\mathbf{x}(n)\right)$$

- ε is the learning rate
- Choosing ε:
  - Small constant
  - Decreases as the iteration goes ahead
  - **Line search**: checked with several values, and the one selected, where $f(\mathbf{x})$ is the smallest
- Stopping condition of the gradient descent iteration
  - When the gradient is zero or close to zero



Jacobean Matrix

- Partial derivative of a vector → vector function
- Specifically, if we have a function $\mathbf{f}:\Re^m \to \Re^n$ then the Jacobian matrix $\mathbf{J} \in \Re^{n\times m}$ of $\mathbf{f}$ is defined such that: $\mathbf{J}_{i,j} = \frac{\partial}{\partial x_j} f(x_i)$

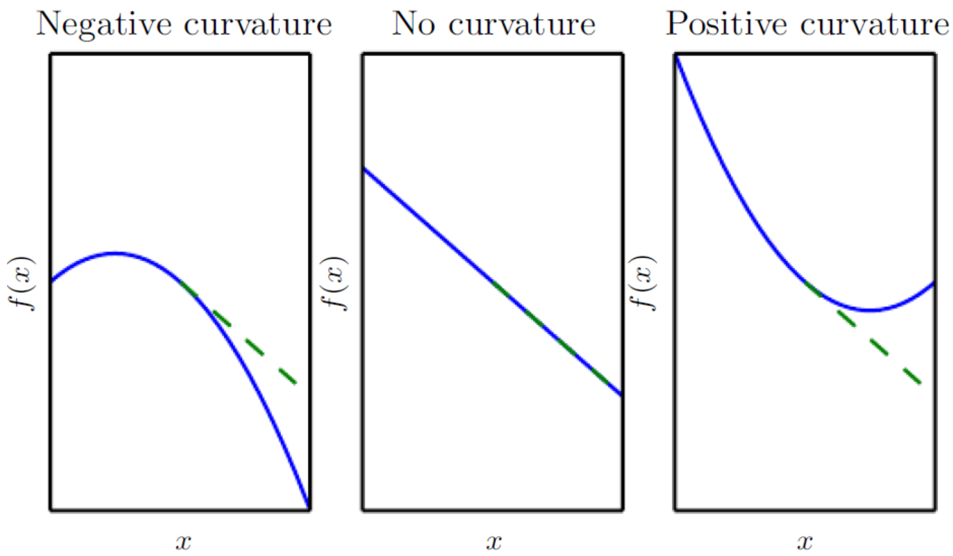
$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

2nd derivatives

- 2nd derivative determines the curvature of a line in 1D
- In nD, it is described by the Hessian Matrix

$$H\left(f(x_{i,j})\right) = \frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial^2}{\partial x_j \partial x_i} f(x)$$

- The Hessian is the Jacobian of the gradient.



2^{nd} order gradient descent method I

- 2^{nd} derivative in a specific direction: $\mathbf{u}^{\mathrm{T}}\mathbf{H}\mathbf{u}$
- Second-order Taylor series approximation to the function *f(x) around the current point* $\mathbf{x}_0$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}}\mathbf{g} + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}}\mathbf{H}(\mathbf{x}-\mathbf{x}_0)$$

where:
g: gradient at x_0
H: Hessian at x_0

- stepping towards the largest gradient:

$$\mathbf{x}_0 - \varepsilon\,\mathbf{g} \approx \mathbf{x} \quad \to \quad \mathbf{x}-\mathbf{x}_0 \approx -\varepsilon\,\mathbf{g}$$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0 - \varepsilon\,\mathbf{g}) \approx f(\mathbf{x}_0) - \varepsilon\,\mathbf{g}^{\mathrm{T}}\mathbf{g} + \frac{1}{2}\varepsilon^2\,\mathbf{g}^{\mathrm{T}}\mathbf{H}\mathbf{g}$$

2^{nd} order gradient descent method II

- Analyzing: $$f(\mathbf{x}_0 - \varepsilon\mathbf{g}) \approx \underbrace{f(\mathbf{x}_0)}_{\text{Original value}} - \underbrace{\varepsilon\mathbf{g}^{\mathrm{T}}\mathbf{g}}_{\text{Expected improvement}} + \underbrace{\frac{1}{2}\varepsilon^2\mathbf{g}^{\mathrm{T}}\mathbf{H}\mathbf{g}}_{\text{Correction due to curvature}}$$
- When the third term is too large, the gradient descent step can actually move uphill.
- When it *is zero or negative,* the Taylor series approximation predicts that increasing ε *forever will decrease f* forever.
- In practice, the Taylor series is unlikely to remain accurate for large ε*, so* one must resort to more heuristic choices of ε *in this case.*
- *When it is positive,* solving for the optimal step

$$\varepsilon^* = \frac{\mathbf{g}^{\mathrm{T}}\mathbf{g}}{\mathbf{g}^{\mathrm{T}}\mathbf{H}\mathbf{g}}$$

Simplest 2nd order Gradient descent method: Newton Method

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}} \nabla f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}} \mathbf{H}\big(f(\mathbf{x}_0)\big)(\mathbf{x}-\mathbf{x}_0)$$

- Replacing $(\mathbf{x}-\mathbf{x}_0) \rightarrow \Delta\mathbf{x}$ and differentiating it with $\Delta\mathbf{x}$, assuming that we can jump to a minima, where: $\nabla f(\mathbf{x}) \approx 0$

$$0 = \frac{\partial}{\partial \Delta\mathbf{x}}\left(\underbrace{f(\mathbf{x}_0)}_{\text{Constant}\rightarrow 0} + \underbrace{\Delta\mathbf{x}^{\mathrm{T}}}_{(\Delta x)' \rightarrow 1} \nabla f(\mathbf{x}_0) + \frac{1}{2} \underbrace{\Delta\mathbf{x}^{\mathrm{T}}\mathbf{H}}_{(½\,(\Delta x)^2)' \rightarrow \Delta x}\big(f(\mathbf{x}_0)\big)\Delta\mathbf{x} \right) = \nabla f(\mathbf{x}_0) + \mathbf{H}\big(f(\mathbf{x}_0)\big)\Delta\mathbf{x}$$

Newton optimization:

$$\Delta\mathbf{x} = -\mathbf{H}\big(f(\mathbf{x}_0)\big)^{-1} \nabla f(\mathbf{x}_0) \qquad \mathbf{x}(n+1) = \mathbf{x}(n) - \eta \mathbf{H}\Big(f\big(\mathbf{x}(n)\big)\Big)^{-1} \nabla f\big(\mathbf{x}(n)\big)$$

Properties of Newton optimization method

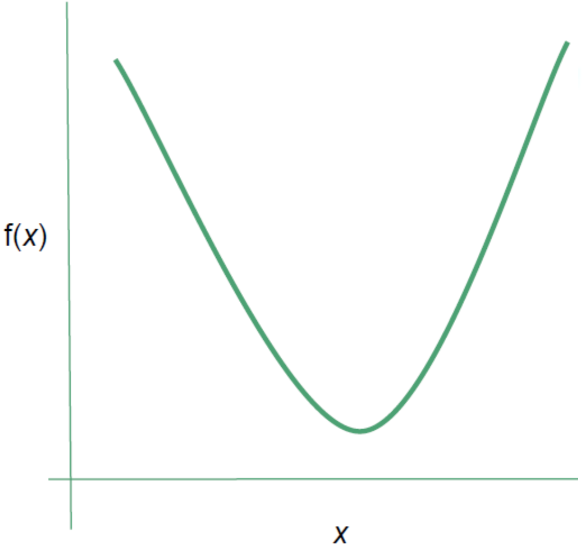
- When f *is a positive definite quadratic function, Newton's method* jumps in a single step to the minimum of the function directly.

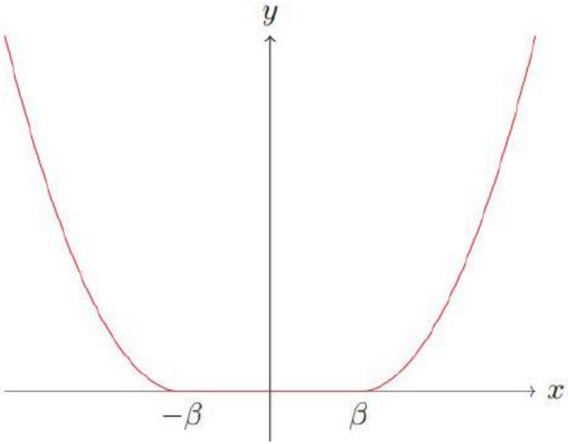
- *Newton's method can* reach the critical point much faster than 1st order gradient descent.

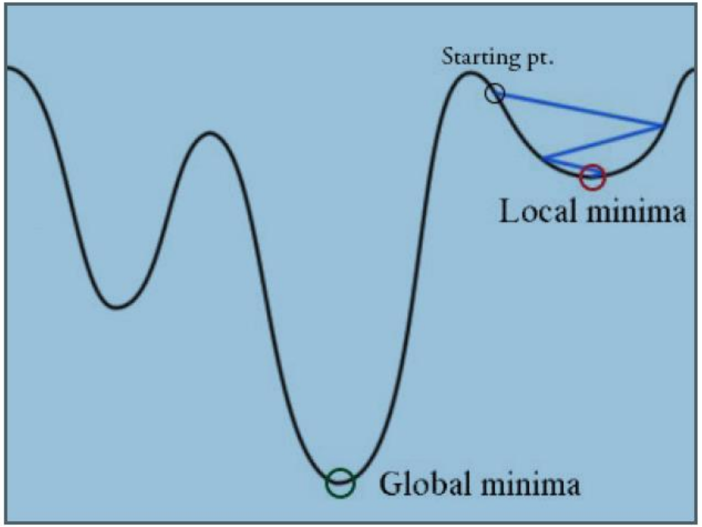
Newton optimization:

$$\Delta\mathbf{x} = -\mathbf{H}\big(f(\mathbf{x}_0)\big)^{-1}\nabla f(\mathbf{x}_0) \qquad \mathbf{x}(n+1) = \mathbf{x}(n) - \eta\mathbf{H}\big(f(\mathbf{x}(n))\big)^{-1}\nabla f(\mathbf{x}(n))$$

Convex and non-convex functions







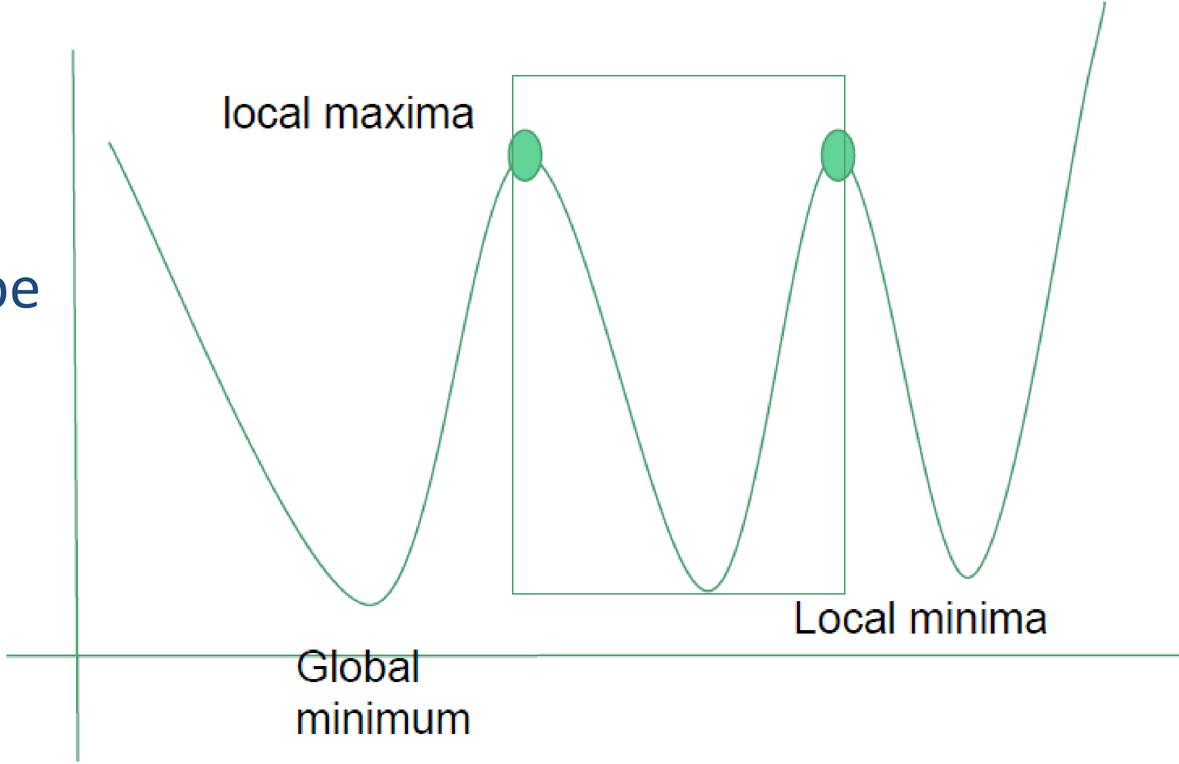
Strongly convex function:
1 local minimum

Non-Strongly convex function: infinity local touching minima with the same values

Non-convex function: multiple non-touching local minima with different values

Local optimization in non-convex case

- Optimization is done locally in a certain domain, where the function is assumed to be convex
- Multiple local optimization is used to find global minimum



Most commonly applied gradient descent methods

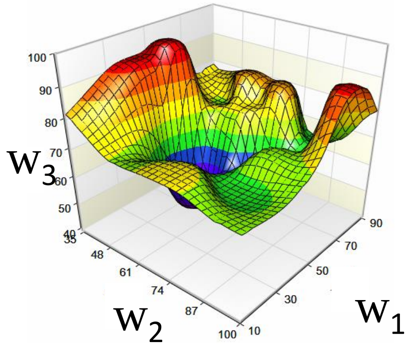
- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

What are we optimizing here?

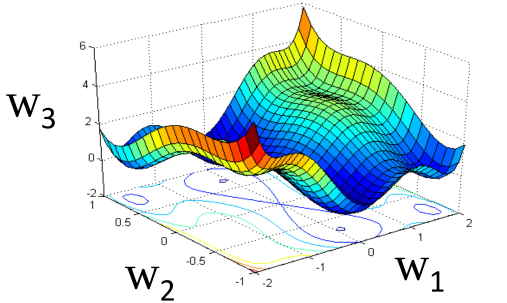
- Cost function in quadratic case for one $\mathbf{x}_i \rightarrow \mathbf{d}_i$ pair:

$$\mathcal{E}_i = \left(\mathbf{d}_i - Net(\mathbf{x}_i, \mathbf{w})\right)^2$$

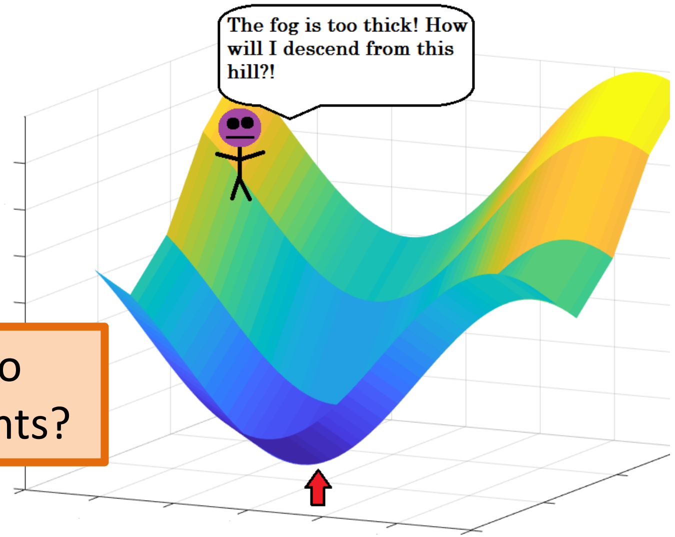
  - Error surface is in the $\mathbf{w}$ space
  - Error surface depends on the $\mathbf{x}_i \rightarrow \mathbf{d}_i$ pair
  - Moreover, we do not see the entire surface, just $\mathcal{E}$ and the gradients $\dfrac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}}$



Error surface for $\mathbf{x}_i \rightarrow \mathbf{d}_i$



Error surface for $\mathbf{x}_k \rightarrow \mathbf{d}_k$



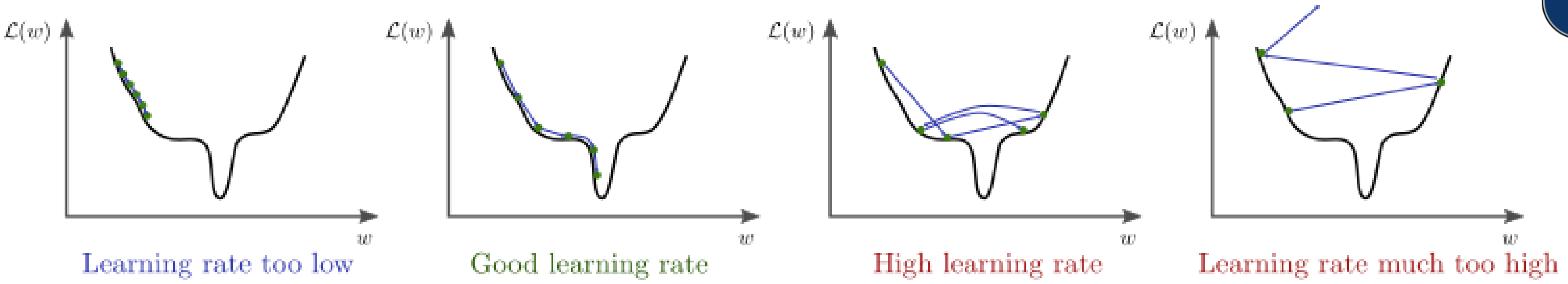
When and how to update the weights?

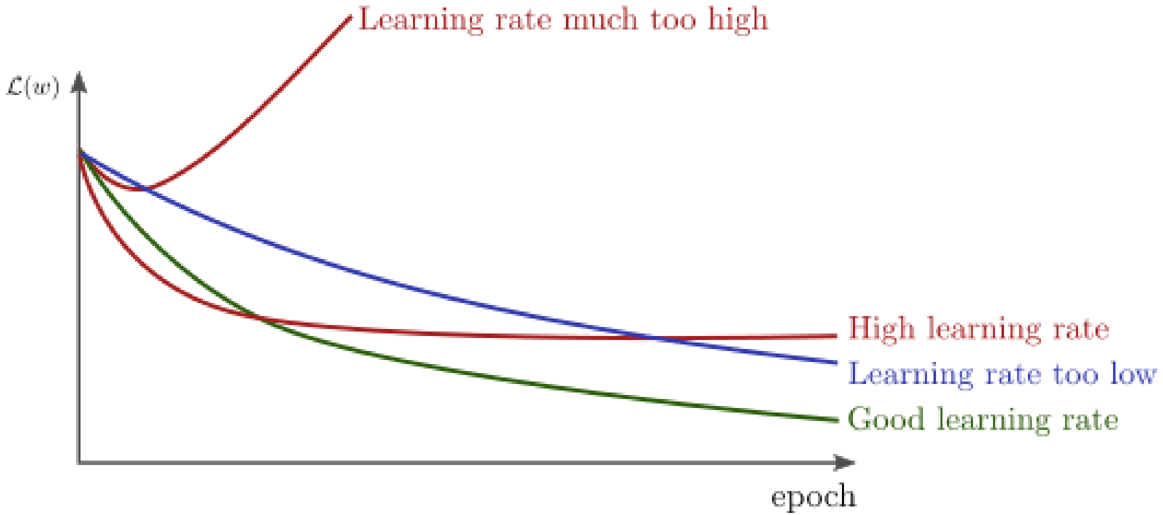
Update strategies

- Single vector update approach (instant update)
  - Weights are updated after each input vector
- Batched update approach
  - All the input vectors are applied
    - this is actually the correct entire error funtion, which is used by the original Gradient Descent Method
  - Updates (Δw_{ij}) are calculated for each vector, and averaged
  - Update is done with the averaged values (Δw_{ij}) after the entire batch is calculated
- Mini batch approach
  - When the number of inputs are very high (10^4-10^6), batch would be ineffective
  - Random selection of m input vectors (m is a few hundred)
  - Updates (Δw_{ij}) are calculated for each vector, and averaged
  - Update is done with the averaged values (Δw_{ij}) after the mini batch is calculated
  - Works efficiently when far away from minimum, but inaccurate close to minimum
  - Requires reducing learning rate

Remember, each approach optimizes different error surfaces!!!

How learning rate effects convergence?





Most commonly applied gradient descent methods

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
  - Nesterov momentum update
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

Stochastic Gradient Descent (SGD) algorithm

- Introduced in 1945
- Gradient Descent method, plus:
  - Applying mini batches
  - Changing the learning rate during the iteration

Learning rate at SGD

- Sufficient conditions to guarantee convergence of SGD:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \qquad \text{and} \qquad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

ϵ is the learning rate, also marked with η sometimes

- In practice:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \qquad \alpha = \frac{k}{\tau}$$

- After iteration τ *, it is common to leave* ε *constant*

Stochastic Gradient Descent algorithm

Algorithm Stochastic gradient descent (SGD) update at training iteration k

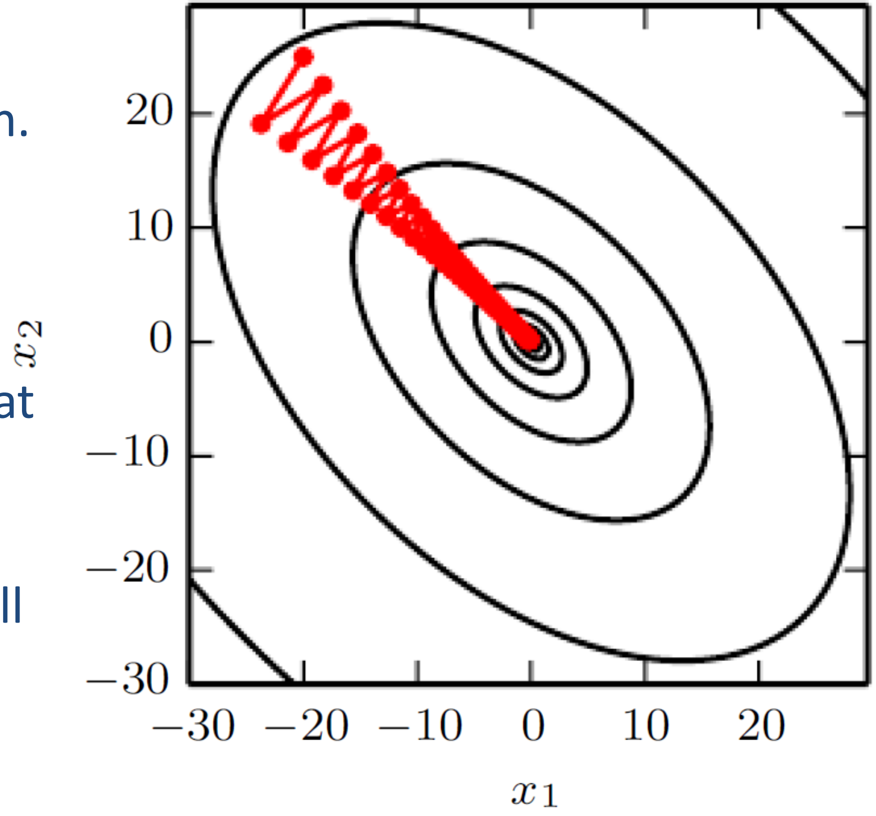
Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$
end while

where: L is the cost function

θ is the total set of $w_{i,j}^{(l)}$ (and all other parameters to optimize)

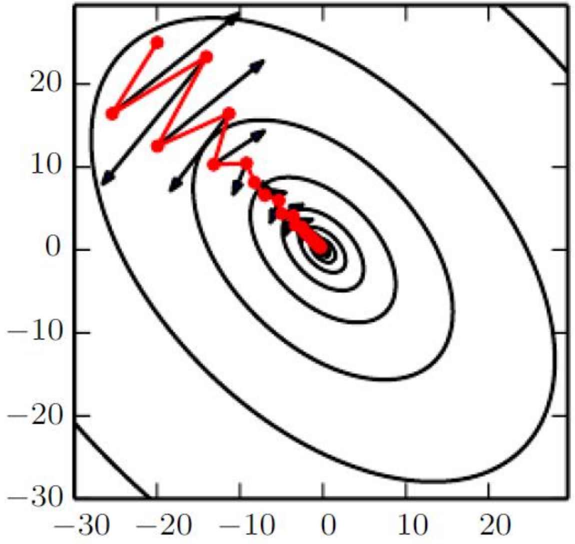
Stochastic Gradient Descent algorithm

- This very elongated quadratic function resembles a long canyon.
- Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature.
- Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration.



Momentum I

- Introduced in 1964
- Physical analogy
- The idea is to simulate a unity weight mass
- It flows through on the surface of the error function
- Follows Newton’s laws of dynamics
- Having *v* velocity
- Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.



Momentum II: velocity considerations

The update rule is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$. The larger α is relative to ϵ, the more previous gradients affect the current direction.

Terminal velocity is applied when it finds descending gradient permanently:

$$\frac{\epsilon ||\boldsymbol{g}||}{1 - \alpha}$$

Momentum III

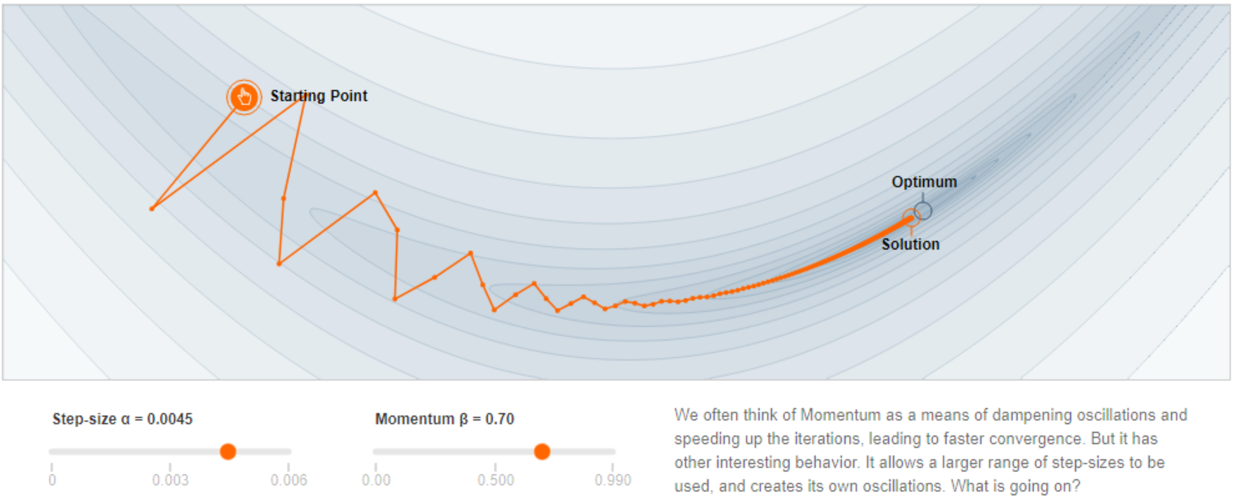
Algorithm Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

Momentum demo

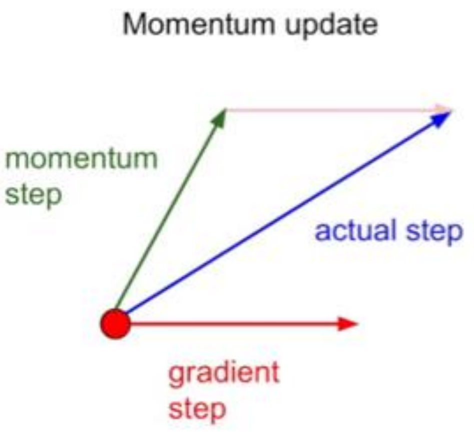
- What does the parameter of the momentum method means, and how to set them?
  – https://distill.pub/2017/momentum/

Why Momentum Really Works



Nesterov momentum update

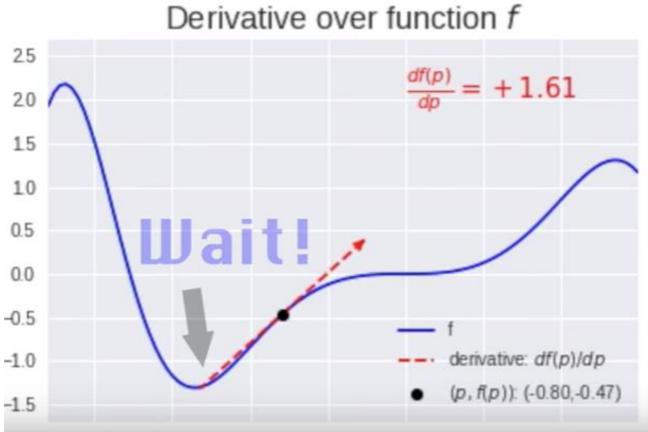
- It calculates the gradient not in the current point, but in the next point, and correct the velocity with the gradient over there (look ahead function)
- It does not runs through a minimum, because if there is a hill behind a minimum, than it starts decreasing the speed in time.



Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$



What if we make the learning rate adaptive as well, not just the velocity?

Most commonly applied gradient descent methods

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
  - Nesterov momentum update
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

AdaGrad algorithm

- The AdaGrad algorithm (2011) individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values

- The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate

- The net effect is greater progress in the more gently sloped directions of parameter space

- AdaGrad performs well for some but not all deep learning models

AdaGrad algorithm

Algorithm The AdaGrad algorithm

Remembers the entire history evenly

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSP algorithm

- The RMSProp algorithm (2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average

- In each step AdaGrad reduces the learning rate, therefore after a while it stops entirely!
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure

- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl

RMSP algorithm

Algorithm The RMSProp algorithm

> *The closer parts of the history are counted more strongly.*

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

ADAM algorithm (2014)

- The name “Adam” derives from the phrase “adaptive moments.”

- In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions.

- in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient.

- Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin

ADAM algorithm

s estimates the gradient from the history (moment)

r estimates the curvature of the gradient

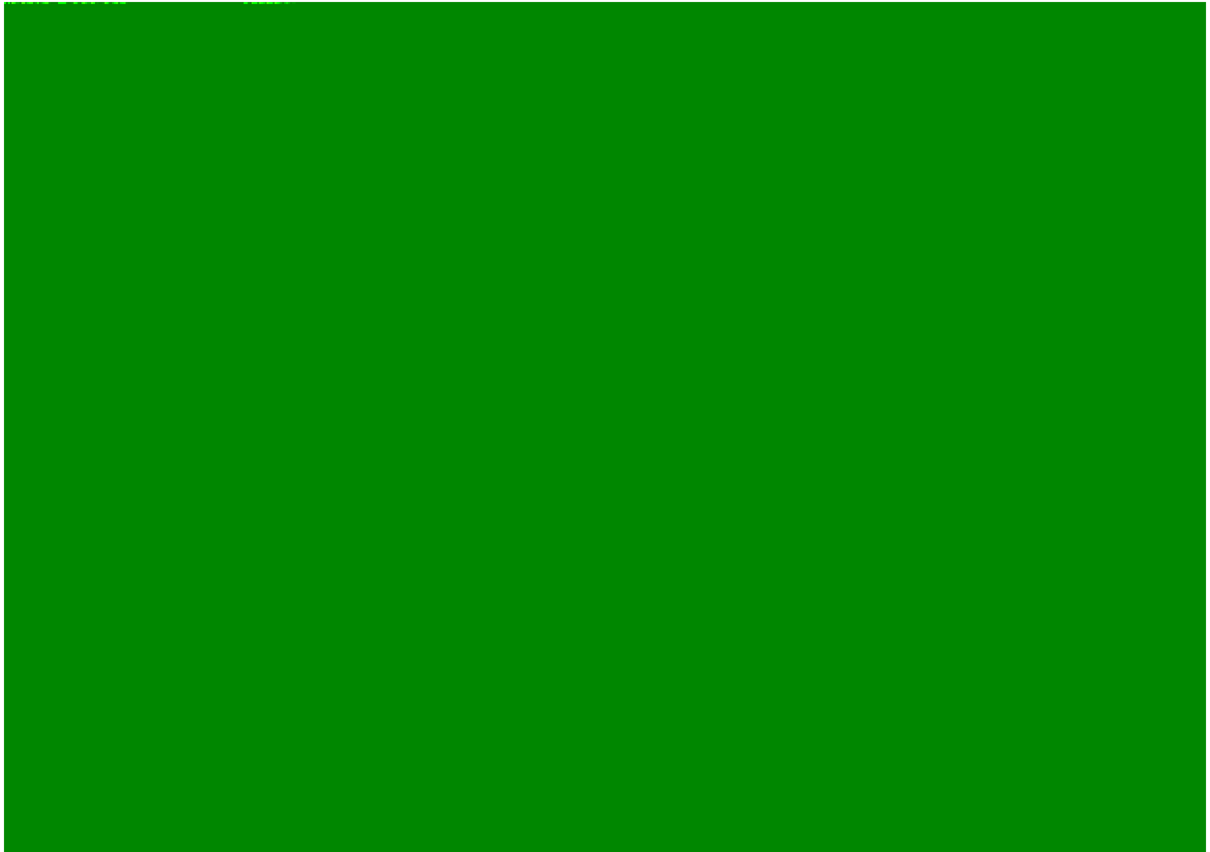
Booth of them are biased to reduce anomalies at the initialization

Algorithm The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while

Video comparing adaptive and non-adaptive methods

- Three optimizer types are compared:
  - SGD
  - Momentum types
    - Momentum
    - Nesterov AG
  - Adaptív
    - AdaGrad
    - AdaDelta
    - RmsProp
- Adaptive ones are the fastest
- SGD is very slow (stucked into saddle point)
- https://www.youtube.com/watch?v=nhqo0u1a6fw&t=306s



Most commonly applied gradient descent methods

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

Newton's algorithm

Algorithm Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of m examples
  while stopping criterion not met **do**
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Newton’s algorithm

- Typically not used, due to the computational complexity
- Parameter space much higher than first order (where it is already very high)

Back propagation

- We have seen last time how to calculate the gradient in a multilayer fully connected network using back propagation
  - The introduced method was based on gradient descent method
- However, being able to calculate gradient, we might select any of the above methods, which leads to orders of magnitude faster convergence

Neural Networks

Components and methods of deep neural networks

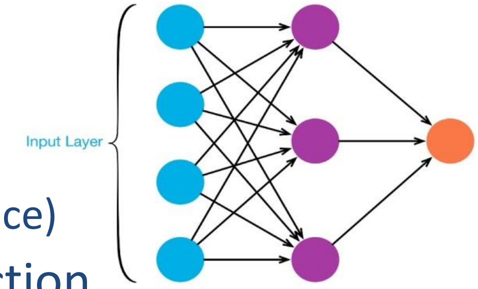
(P-ITEEA-0011)

Akos Zarandy
Lecture 5
October 8, 2019

Contents

- Recall
  - Optimization
  - Analysis of the different methods
- Activation functions
  - Various ReLUs
  - Softmax
- Error functions
  - Cross-entropy
  - Negative log-likelihood
- Regularization
  - Batch normalization,
  - Weight regularization
  -

We discussed…



- How to construct an Artifitial Neural Network
  - Architecture, parameters, signal propagation, recall (inference)
- How to calculate the local gradient from the error function
  - Error back propagation
- Update strategies
  - Batch approach: Error function based on all the training vectors
    (*K: Number of all the training vectors*)

$$e = \frac{1}{K}\sum_{k=1}^{K}(d_k - Net(x_k, w))^2$$

  - Instant update: Error function based on one training vector

$$e = (d_k - Net(x_k, w))^2$$

  - Mini batch approach: Error function based on a random subset of the training vectors
    ($m_b \approx 200$)

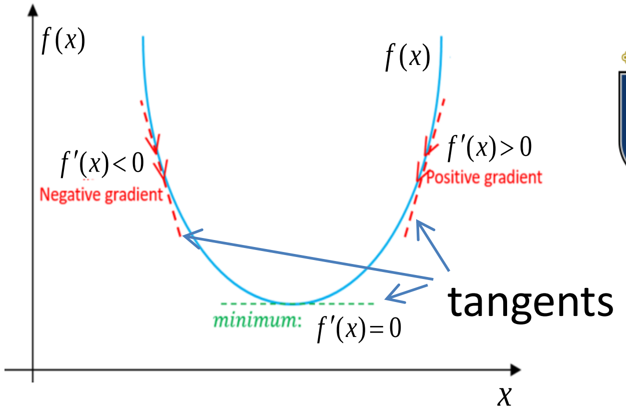
$$e = \frac{1}{m_b}\sum_{k=1}^{m_b}(d_k - Net(x_k, w))^2$$

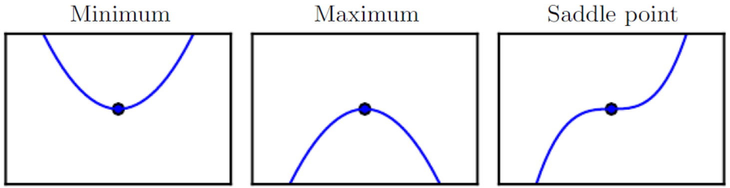
<u>**Epoch:**</u> *One Epoch is when the ENTIRE training set is passed forward and backward through the neural network only ONCE.*

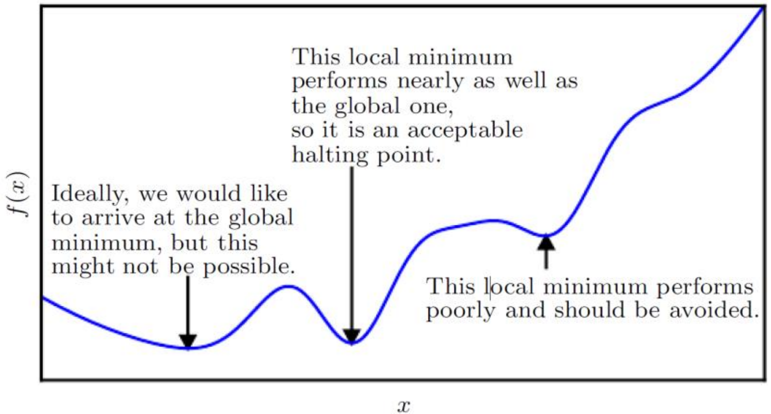
Epoch: time period (korszak in Hungarian)

As we discussed …

- Once the gradient is known, optimization of the network parameters can be done
- Gradient Descent Method
  - Always uses the total error function (all the training samples are used)
    - *Painfull to calcualte the gradient in case of a very large training set*
  - Easily stucks in saddle points
  - Stucks in local minima
  - Very slow!

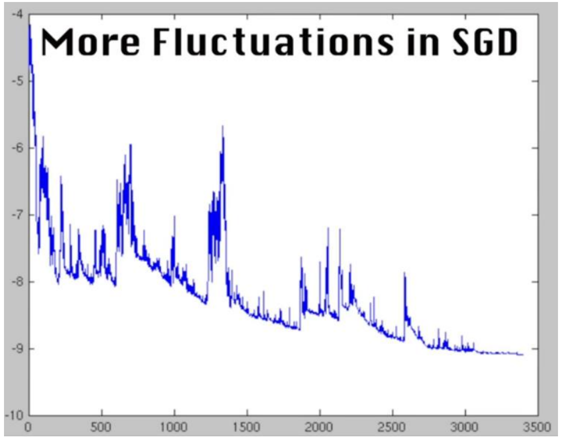


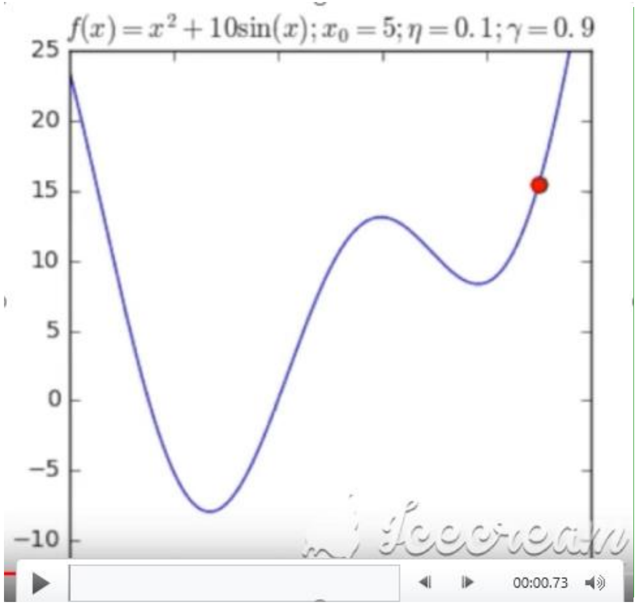




As we discussed …

- Stocastic Gradient Descent (SGD) Method
  - Uses a random subset of the training vectors (mini batches)
    - *One update is fast to calcualte*
  - The objective function changes stocastically with the minibatch selection
    - *More fluctuation in the objective function than in case of Gradient Descent*
    - *It helps to come out from local minima and saddles*
  - Decreases the learning rate during the training time to reduce overshoot
  - Still very slow! (Many update steps are needed)
- More advanced optimization methods required!

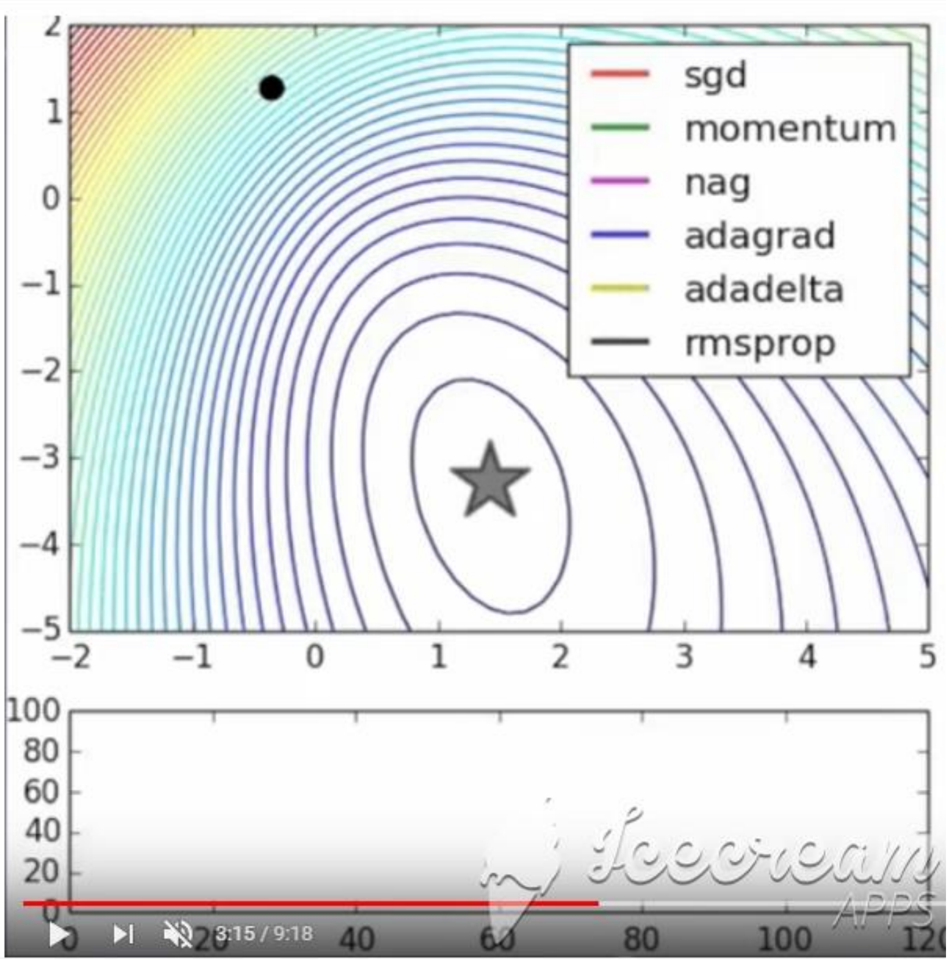




Comparing adaptive and non-adaptive methods

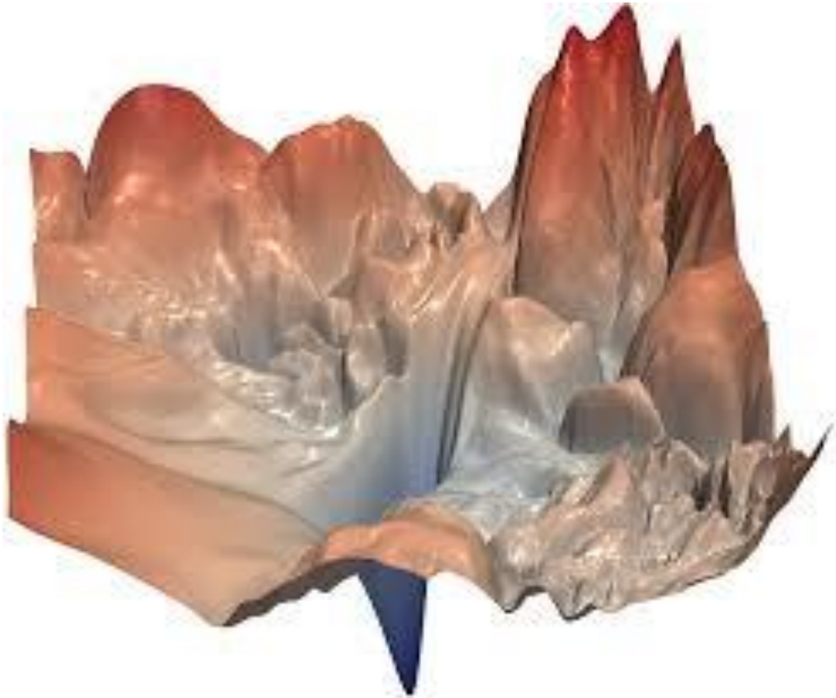
- Three optimizer types are compared:
  - SGD
  - Momentum types
    - Momentum
    - NAG
  - Adaptive
    - AdaGrad
    - AdaDelta
    - RmsProp
- Adaptive ones are the fastest
- SGD is very slow

- https://www.youtube.com/watch?v=nhqo0u1a6fw&t=306s

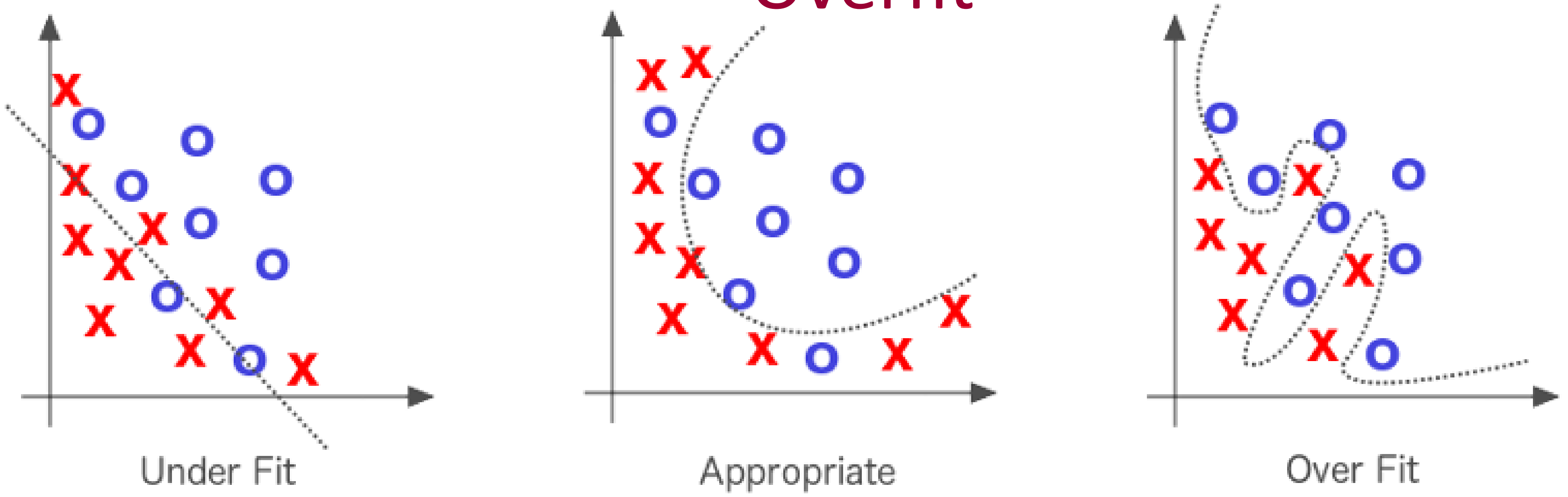


Do we have to reach the global minimum?

- Not really
- Global minimum means: **Overfitting**
- Overfitting: The network exactly learned the training vectors
- However, it loses the generalization capabilities



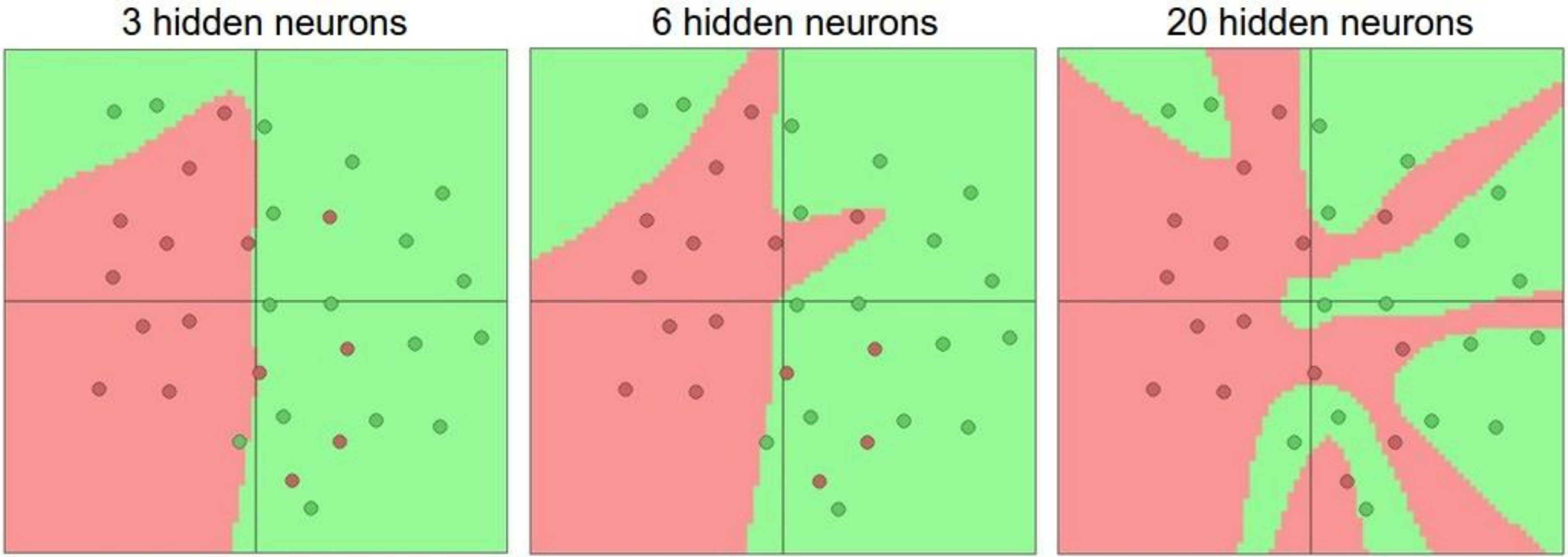
Overfit



Overfitting occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship

Losing the generalization capabilities!!!

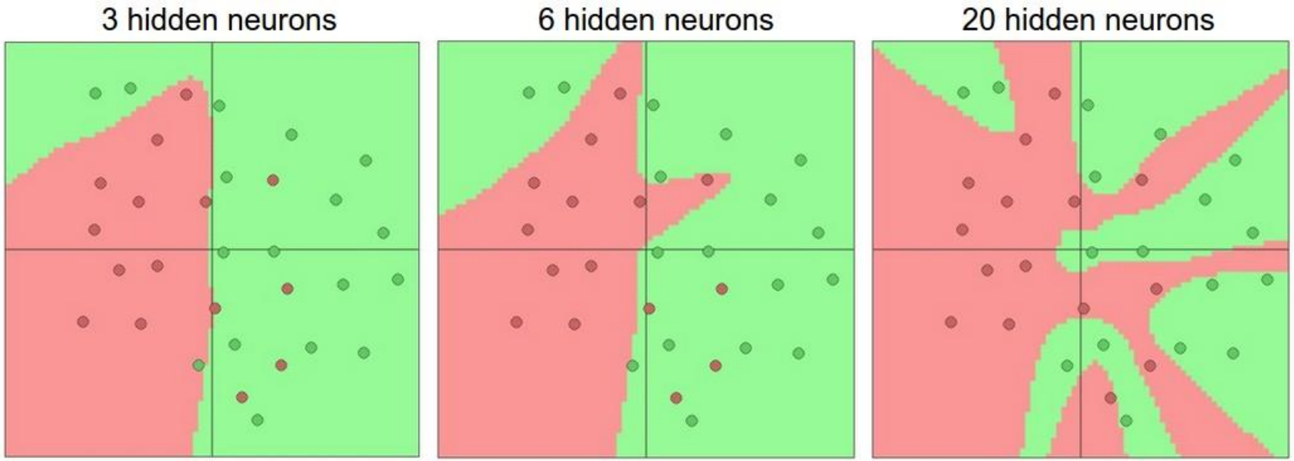
Network complexity vs. capacity



http://cs231n.github.io/neural-networks-1/

https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

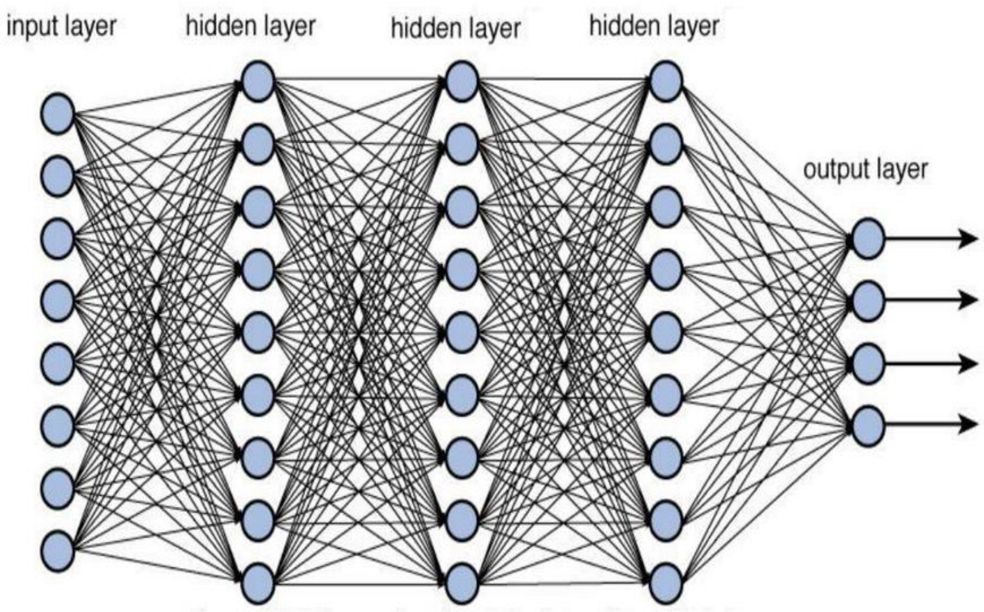
Network complexity vs. capacity



- In general, the more layers we have, and the more neurons there are, the larger the capacity.
- There is no adequate method to predict the required complexity.
- Even if a network is capable to learn a task, it is not guaranteed that it will.

Now we understand

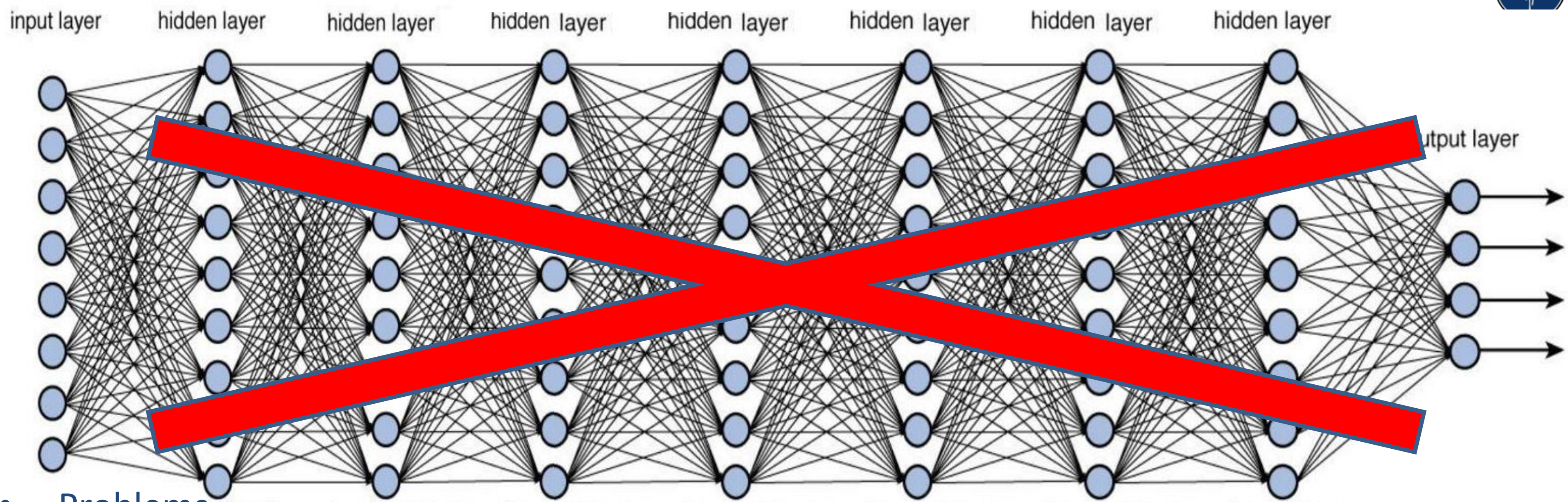
- Architecture of the multilayer fully connected neural networks
- Operation of these networks
- Derivation of the parameters
- Arbitrary function can be approximated if the neural network is complex enough



How to increase complexity on a smart way?

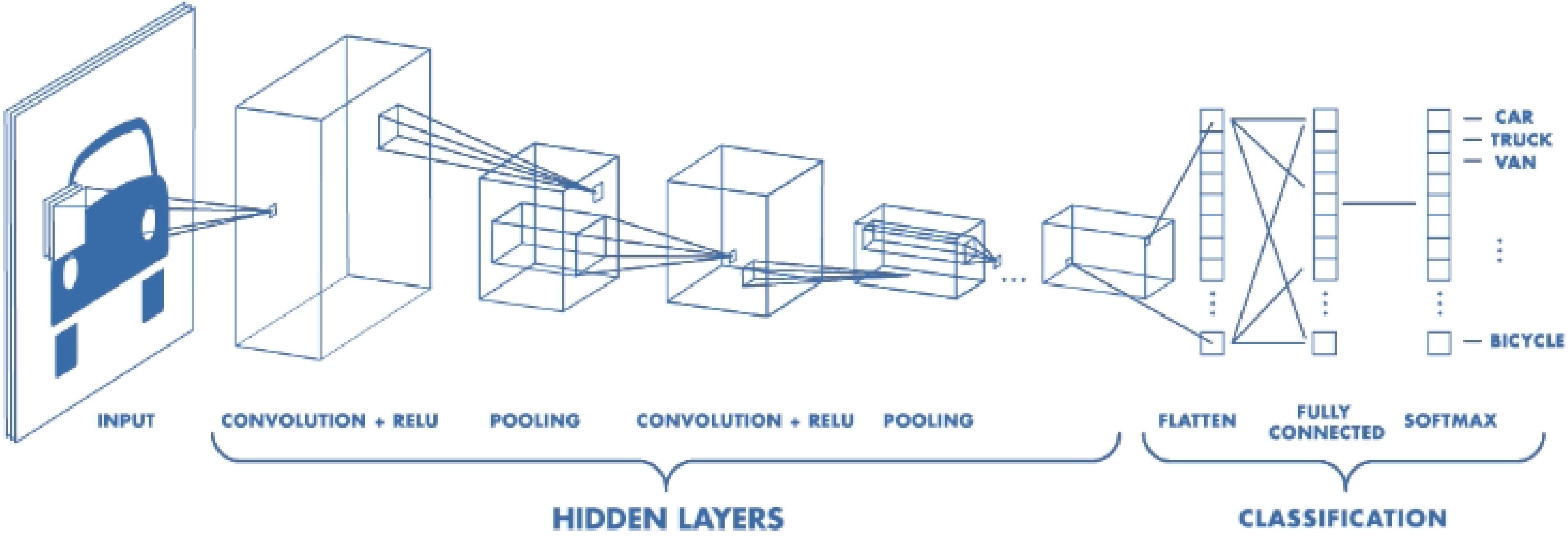
No-brainer solution: Increase the number of the hidden layers

D E E E E E E P



- Problems:
  - Number of free parameters are exploding
  - Numerical problems arises after using too many layers (*double precision limit*)
- Solution:
  - Try to mimic human brain:
  - Use hierarchical architectures!
  - Reusable components!

Hierarchical architecture of a deep neural network



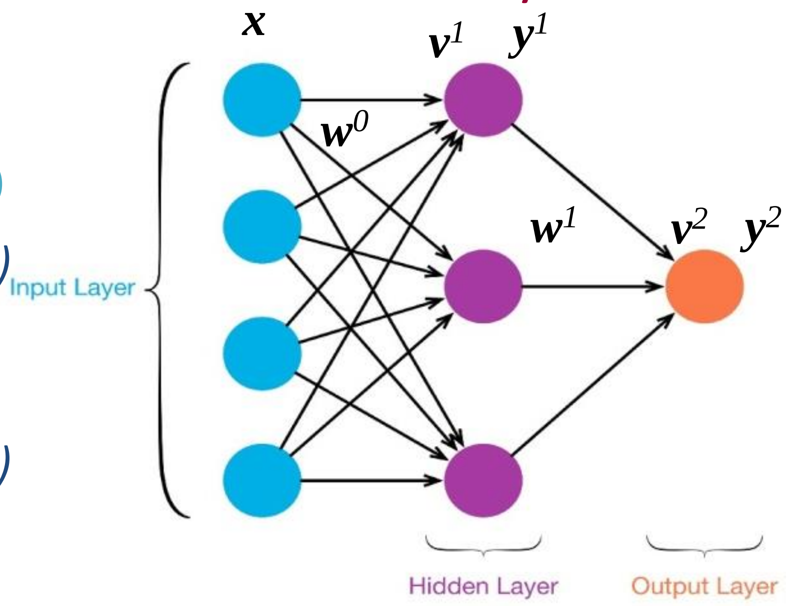
What are the building blocks of a hierarchical deep neural network?

Components and methods

- Activation functions
- Error (loss) functions
- Regularization
    - Batch normalization
    - L1 and L2 regularizations

Why do we need nonlinear activation function in the hidden layers?

”Repeated matrix multiplications interwoven with activation functions.” (Karpathy)



$\boldsymbol{v}^1 = \boldsymbol{w}^0 \boldsymbol{x}$ *(Summing junctions of layer 1)*

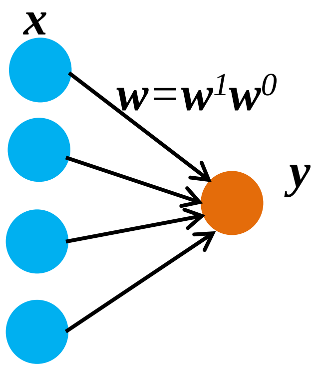
$\boldsymbol{y}^1 = \varphi(\boldsymbol{v}^1)$ *(Output of layer 1)*

$\boldsymbol{v}^2 = \boldsymbol{w}^1 \boldsymbol{y}^1$ *(Summing junctions of layer 2)*

If the neuron is linear: $\varphi(\boldsymbol{v}^1) = \boldsymbol{v}^1$ $\quad \boldsymbol{y}^1 = \boldsymbol{w}^0 \boldsymbol{x}$

$$\boldsymbol{v}^2 = \boldsymbol{w}^1 \boldsymbol{y}^1 = \boldsymbol{w}^1 \boldsymbol{w}^0 \boldsymbol{x} = \boldsymbol{w}\boldsymbol{x}$$

The two layers can be combined into an equivalent single layer network!



On the other hand, we could not approximate arbitrary kinds of functions, only linear ones!

Sigmoid function

- Sigmoid function compresses the output
- Used in classification,
  - The network calculates the probability of the yes and the no decisions at the same time

$$Net(\mathbf{x}_k, \mathbf{w}) = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x}) = P\big((\mathbf{y}_k|\mathbf{x}_k; \mathbf{w})\big)$$

- Probability of <u>**yes**</u> decision:

$$P\big((\mathbf{y}_k|\mathbf{x}_k; \mathbf{w})\big) = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x})$$

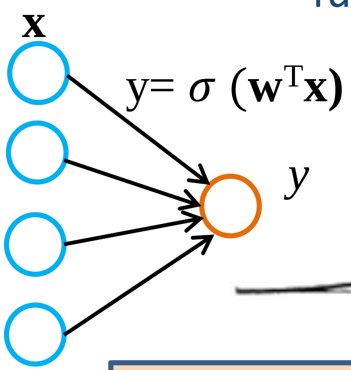
- Probability of <u>**no**</u> decision:

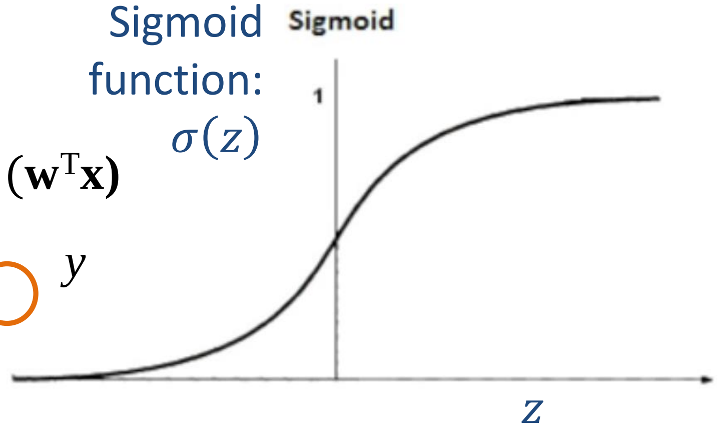
$$1 - P\big((\mathbf{y}_k|\mathbf{x}_k; \mathbf{w})\big) = 1 - \sigma\big(\mathbf{w}^{\mathrm{T}}\mathbf{x}\big) = \sigma(-\mathbf{w}^{\mathrm{T}}\mathbf{x})$$

- It generates the probability (ϕ) parameter of a Bernoulli distribution:

$$\sigma(z) + \sigma(-z) = 1$$

- When z is large or small, the derivative of the output is minimal (compresses the gradient)
  - It <u>significantly</u> slows down the training when quadratic loss function is used





<u>**Bernoulli Distribution**</u> is a distribution over a single binary random variable. (like flipping a coin: *head or tail*)
It is controlled by a single parameter $\phi \in [0,1]$, which gives the probability of the random variable being equal to 1

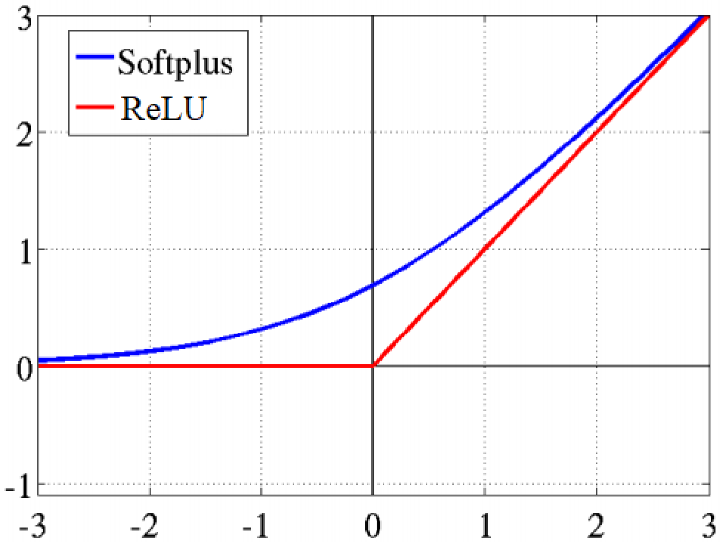
Pobability of head: $P(\mathrm{x} = 1) = \phi$

$P(\mathrm{x} = 0) = 1 - \phi$

Expectation: $\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = \phi$

ReLU: Rectified Linear Unit

- Very easy to calculate
  - Implementation is a simple sign comparison and replacing with 0 if negative
- Also easy to calculate its derivative
- Also called:
  - Ramp function
  - Half-wave rectifier
- Orders of magnitude learning speed advantage
  - Due to non-compressed gradient
- Smooth analytic approximation is the Softplus function
- Asymptotically reaches ReLU



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

ReLU

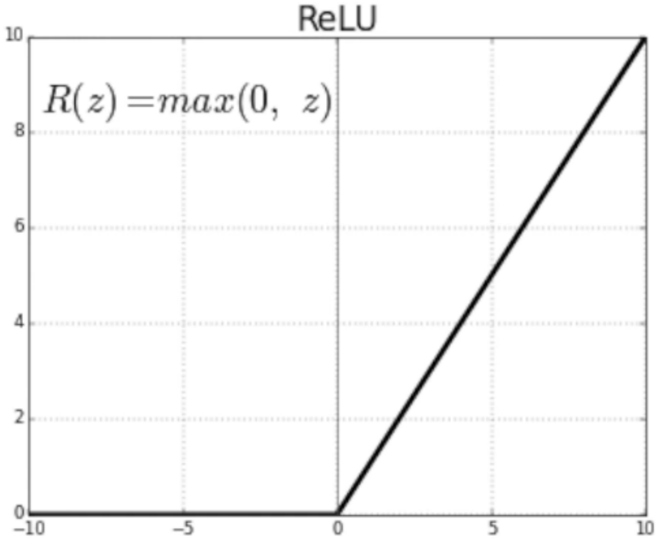
$$f(x) = \log(1 + e^x)$$

Softplus

Most used in hidden layers in deep neural networks (as of 2019)!

Dying ReLU problem

- During training it happens that the weight composition of a neuron got a certain combination in a high gradient situation (when large jump happens during the optimization), which leads to generate zero output from that point on.
  - Happens typically with large learning rate
  - E.g. a very large negative value appears in the bias position
- That neuron will output zero for each input vector from that point
  - Irreversible
  - No contribution to the decision
  - A usefull neuron selectively fires to a set of input vector having the same properties
- In some bad cases, even 40% of the neurons dies in coarse of a long training (Vanishing Gradient problem)



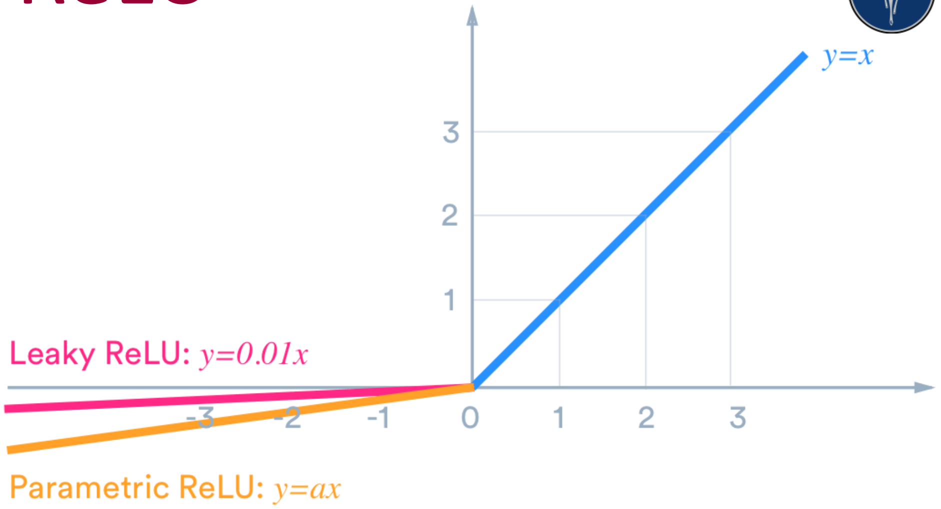
$$\mathbf{y}^{(L)} = R\left(\mathbf{w}^{(L-1)T}\mathbf{y}^{(L-1)} + b^{(L-1)}\right) = \mathbf{R}(v)$$

$$\Delta w_{ji}^{L} = \eta R'(v) e_i y_i^{(L-1)}$$

Avoid the absolute zero part!
Introduction of Leaky ReLU.

Leaky ReLU

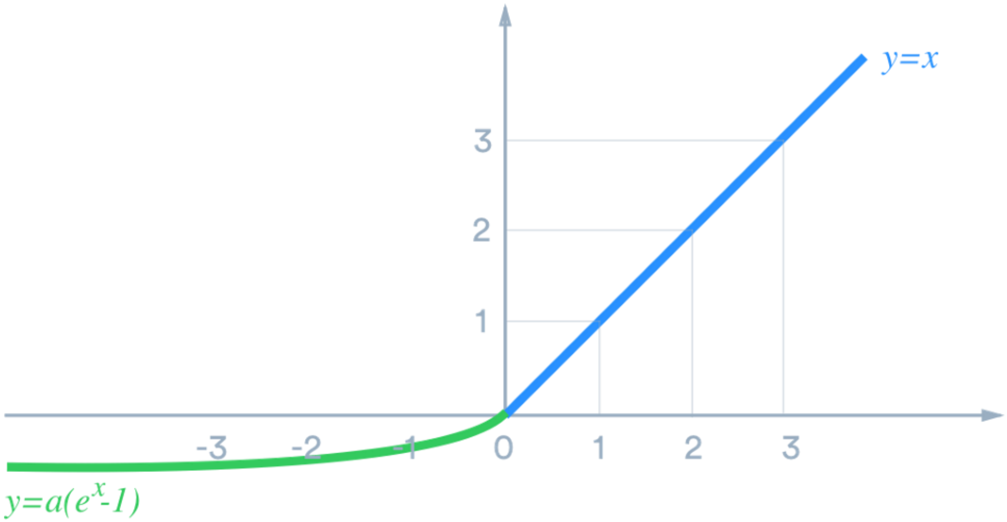
- No constant zero output
- Neurons do not die
- **Parametric ReLU**
  - Variation of leaky ReLU
  - a is a hyper-parameter:
    - Tuned during training
- Leaky ReLUs are not necessarily superior than normal ReLU
- It is an option, if normal ReLU is not performing well



$$f(x) = \max(x, ax)$$
$where:$
$a\ is\ a\ small\ positive\ number$

ELU: Exponential linear units

- Variation of leaky ReLU
  - No constant zero output
  - Neurons do not die
  - Mean activation closer to 0 in the negative region
- Obtains higher classification accuracy than ReLU, but requires more computations
- a is a hyper-parameter:
  - Tuned during training

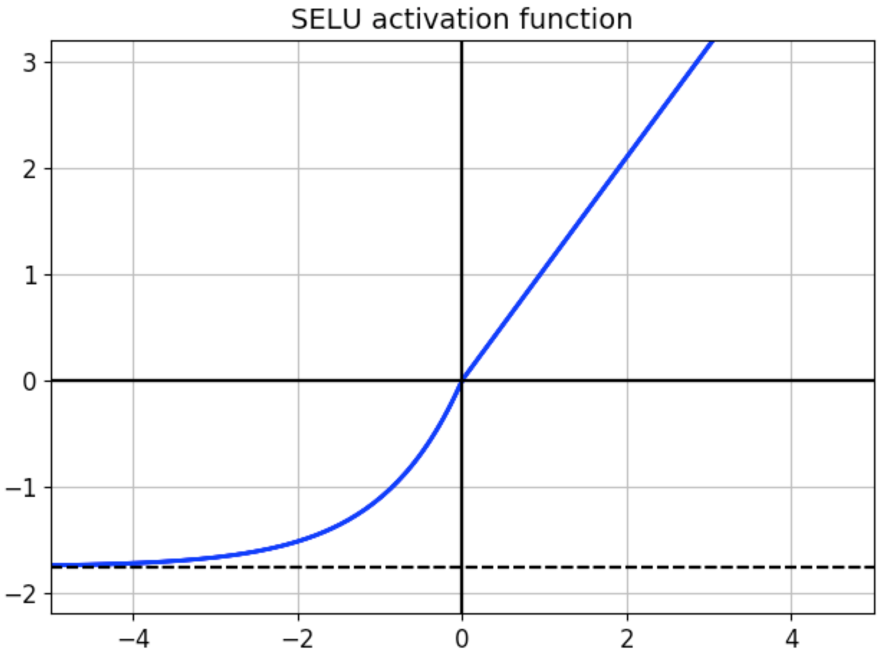


$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ a(e^x - 1) & \text{otherwise} \end{cases}$$

a is a hyper-parameter to be tuned and $a \geq 0$ is a constraint.

SELU: Scaled Exponential linear units

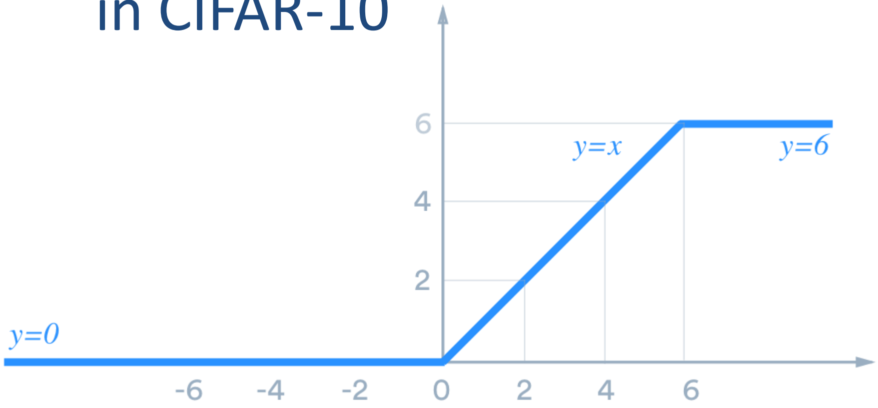
- Variation of leaky ELU
- Two fixed parameters
  - Not trained, but selected to be fixed
  - λ is the scaling parameter

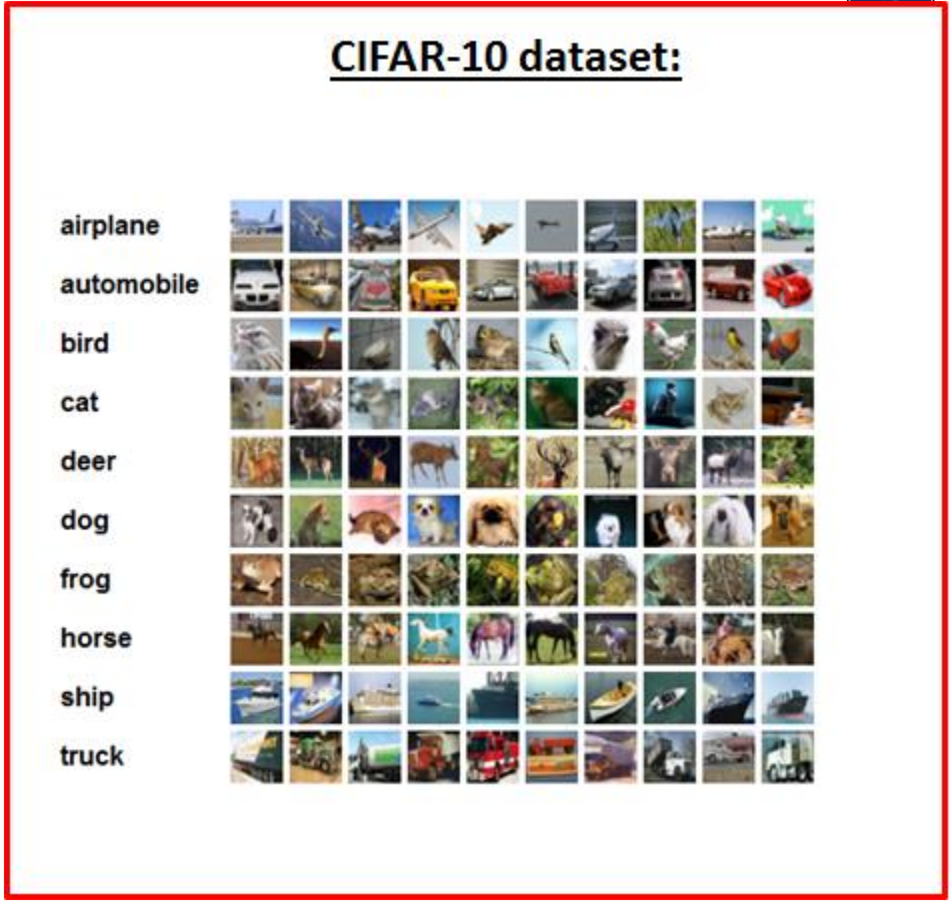


$$\mathrm{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leqslant 0 \end{cases}$$

ReLU6

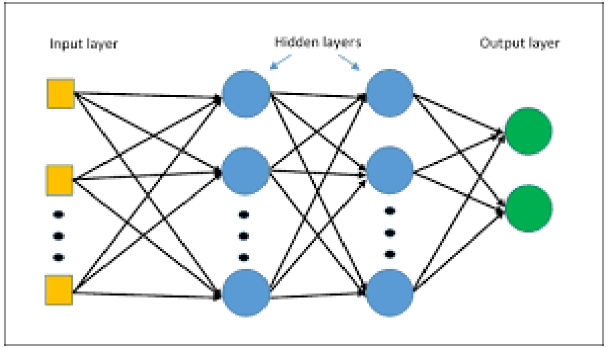
- Variation of ReLU
  - Capped at 6
  - 6 is a choosable parameter
- Shown to learn sparse features faster
- Turned out to be usefull in CIFAR-10





What do we expect from the activation functions?

- Strong nonlinearities to support approximation of wide range of functions
- To drive (during training) the individual neurons in the hidden layers to a parameter zone where it is
  - Silent for a set of input vectors
  - Active for another set of input vectors
- Letting the gradient go through them
- Work together with the loss function (select them in synchrony)



Loss functions

- Loss function determines the training process
  - Tells the net, whether an error is big or small, and penalize accordingly
  - There can be other errors, not just the difference of the output and the desired output
- Most used loss function types:
  - Quadratic, in case of regression

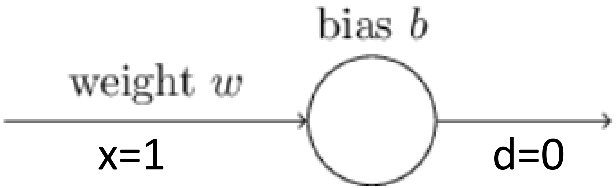
$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

  - Conditional log-likelihood, in case of classification
    The sum of the negative logarithmic likelihood is minimized

$$C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\left(-logP(\mathbf{y}_k|\mathbf{x}_k, \mathbf{w})\right)$$

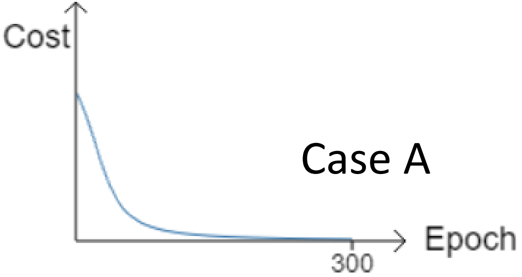
What is the problem with quadratic loss function in classification tasks?

- In case of classification, the convergence can be very slow
- Consider the following very simple case

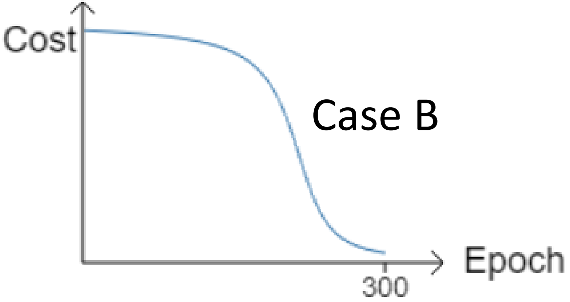


Check out the example!
http://neuralnetworksanddeeplearning.com/chap3.html

- **Case A**: Start the learning from w(0)=0.6, b(0)=0.9
  - Loss function decreases quickly



- **Case B**: Start the learning from w(0)=2, b(0)=2
  - Loss function decreases very slowly at the beginning



- Why is that?
  - Because the Δw is proportional with the gradient

Calculation of the gradient

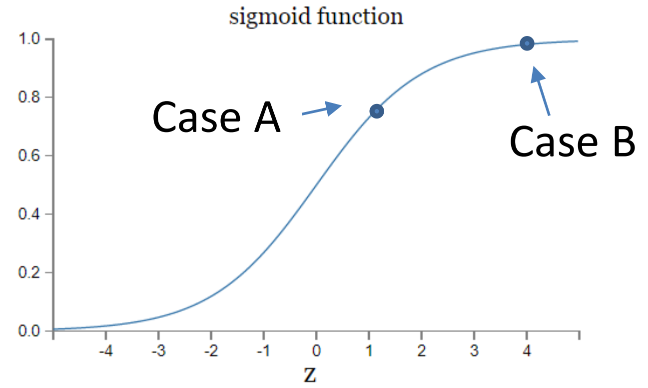
- Loss function:

$$L = {}^{1}/_{2}\,(d - y)^2, \quad where \quad y = \sigma(wx + b)$$

- Gradient, using chain rule:

$$\frac{\partial L}{\partial w} = (y - d)\,\sigma'(wx + b)\mathrm{x} = y\sigma'(wx + b)\mathrm{x}$$

- **Case A**: w(0)=0.6, b(0)=0.9, x=1, d=0
  - Slope of the gradient is fine: $(wx + b) = 1.5$
  - Fast convergence
- **Case B**: w(0)=2, b(0)=2, x=1, d=0
  - Slope of the gradient is very small: $(wx + b) = 4$
  - Very slow convergence



Sigmoid with quadratic loss function leads to very small gradient even at large error, when the argument of the sigmoid is a large value.

Introducing Cross Entropy

- Idea: replace the quadratic Loss function with a more appropriate Loss function: Try cross entropy!
- In general: $$\boxed{C = -\frac{1}{n}\sum_x \left[y \ln a + (1-y)\ln(1-a)\right]}$$
- $C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\Big(d_k\, log P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) + (1-d_k) log\big(1 - P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\big)\Big)$
  - Is it always positive?
    - d_k is either 0 or 1 (binary classification)
      - Either the first or the second term is zero
    - $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) = \sigma(\mathbf{w}\mathbf{x}_k + b)$
      - The probability is the output of the network
      - Due to the sigmoid, it is between 0 and 1
      - Therefore, its logarithm is negative

Introducing Cross Entropy

- Idea: replace the quadratic Loss function with a more appropriate Loss function: Try cross entropy!
- $C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\left(d_k\, logP(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) + (1-d_k)log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\right)$
  - Is it a good loss function?

    Good decision (small loss):

    - When d_k is 0 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 0, than $-log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\sim\mathbf{0}$
    - When d_k is 1 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 1, than $-log\left(P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\sim\mathbf{0}$

    Bad decision (large loss):

    - When d_k is 0 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 1, than $-log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\sim\infty$
    - When d_k is 1 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 0, than $-log\left(P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\sim\infty$

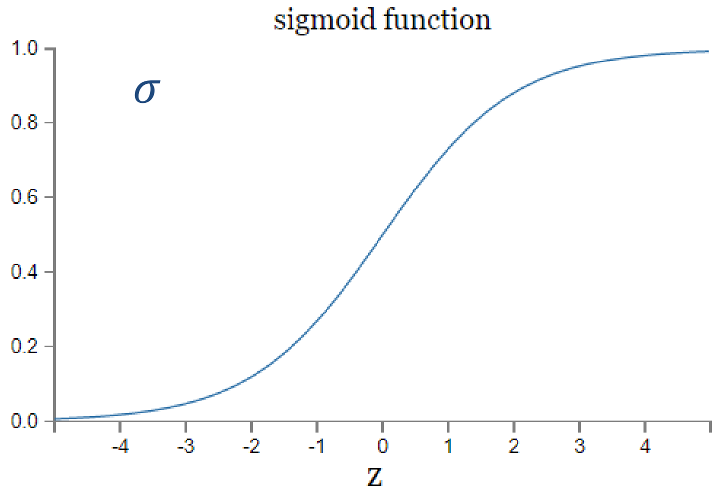
http://neuralnetworksanddeeplearning.com/chap3.html

Introducing Cross Entropy

- Why is cross entropy good?
- $C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\left(d_k\, logP(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) + (1-d_k)log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\right)$
  - Because its partial derivative does not contain σ'

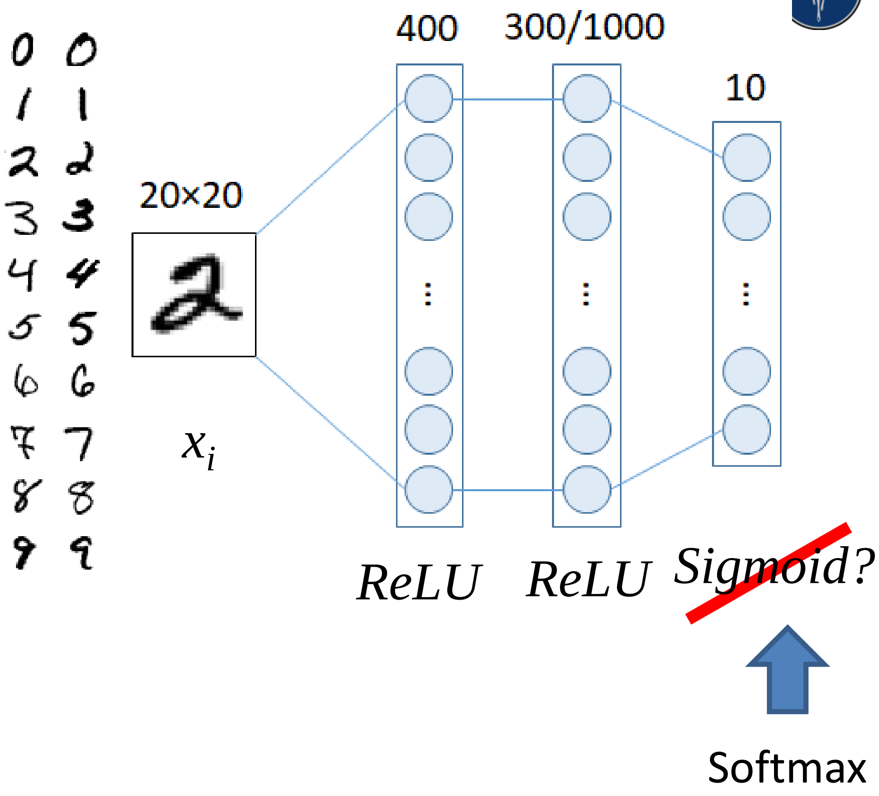
$$\frac{\partial C}{\partial w_j} = \frac{1}{K}\sum_{k=1}^{K} x_j(\sigma(\mathbf{wx}+\mathbf{b}) - d)$$

  - The gradient is proportional with the value of the sigmoid, and not with its derivative!



Probabilistic decision (n discrete categories)

- Assume we have annotated input vectors with *n* different classes (MNIST data base)
- Expect a probability distribution on the output layer!
  - $0 \leq y_i \leq 1$ sigmoid OK!
  - $\sum_{i=1}^{n} y_i = 1$ sigmoid NOT OK!



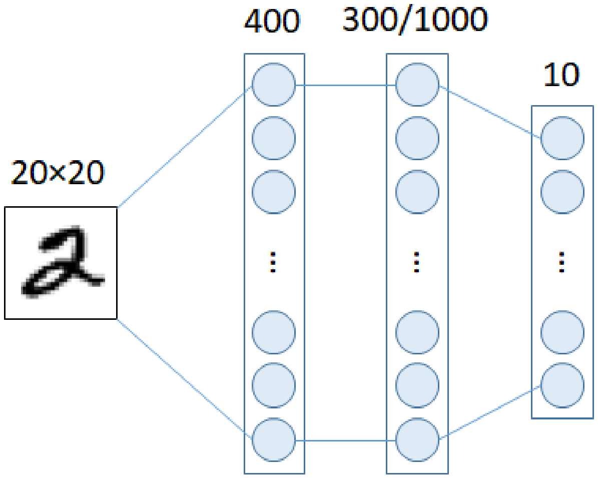
Softmax

- Mathematically:
  - Normalized exponential functions of the output units
- Probability distribution of n discrete classes:
  - One-of-n classes problems
  - $0 \leq y_i \leq 1$
  - $\sum_{i=1}^{n} y_i = 1$
- Architectural difference:
  - Previously learned activation functions were based on the inputs of one neuron
  - Softmax combines a layer of output neurons

$$y_i = softmax(v)_i$$

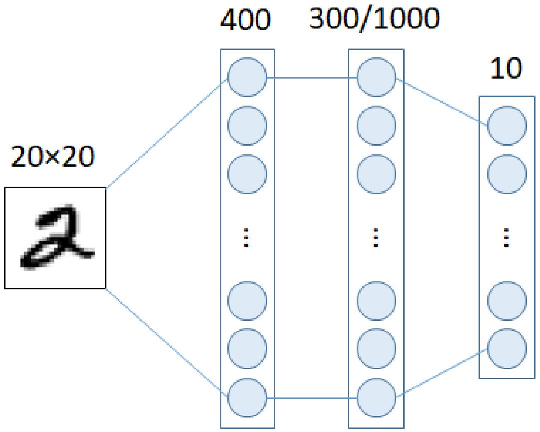
$$= \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$$

$$\boldsymbol{v} = \boldsymbol{w}^T \boldsymbol{x}$$



Properties of Softmax

- Generalization of sigmoid function for one-of-n class
- *Squashes* a vector of size n between 0 and 1
- Improves the interpretability of the output of a Neural Net
- Describes the probability distribution of a certain class
  - We may use the word ”confidence”
- Winner take all
  - exponential function strongly penalize the non-winners
  - Similar to lateral negative feedback in the natural neural systems



$$y_i = softmax(v)_i = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$$

$$\boldsymbol{v} = \boldsymbol{w}^T \boldsymbol{x}$$

EXAMPLE

Input pixels, $\mathbf{x}$

Forward propagation →

Feedforward output, $\boldsymbol{v}_i$

cat	dog	horse
5	4	2

Softmax function →

Softmax output, $S(\boldsymbol{v}_i)$

cat	dog	horse
0.71	0.26	0.04

Input images — Input values — Probability scores

https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/

EXAMPLE

Input pixels, $\mathbf{x}$

Forward propagation →

Feedforward output, $\boldsymbol{v}_i$

cat	dog	horse
5	4	2
4	2	8

Softmax function →

Softmax output, $S(\boldsymbol{v}_i)$

cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98

Input images

Input values

Probability scores

https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/

EXAMPLE

Input pixels, $\mathbf{x}$

Input images

Forward propagation →

Feedforward output, v_i

cat	dog	horse
5	4	2
4	2	8
4	4	1

Input values

Softmax function →

Softmax output, $S(v_i)$

cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

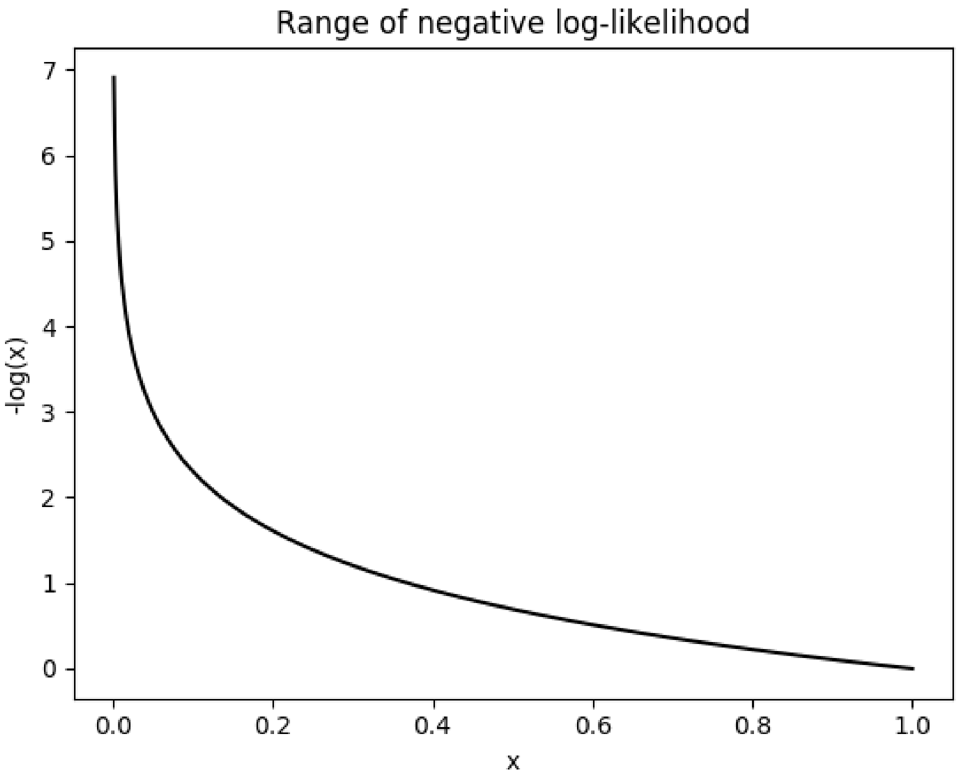
Probability scores

https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/

Loss function for softmax: Negative log-likelihood

- $L(\boldsymbol{y}) = \sum_{k=1}^{K} -\log(y)$
- The negative logarithm of the probability of the correct decision classes are summed up
- It is small, if the confidence of a good decision was high for a certain class
- Large, when the confidence is low
- Partial derivative of a softmax layer with negative log-likelihood:

$$\frac{\partial C}{\partial v_j} = y_j - 1$$



Example

Input pixels, $\mathbf{x}$ Softmax output, $\boldsymbol{S}(\boldsymbol{v}_i)$ Loss, $\mathbf{L}(a)$

cat	dog	horse		NLL
0.71	0.26	0.04		0.34
0.02	0.00	0.98	$\xrightarrow{-\log(a) \text{ at the correct classes}}$	0.02
0.49	0.49	0.02		0.71

The correct class is highlighted in red

Total: **1.07**

Correct classes are known because we are training

Predictor confidence of **horse** is high.

Predictor confidence of **dog** is low.

Probability scores for correct classes (want big numbers)

Negative log for correct class: (want small numbers)

When computing the loss, we can then see that higher confidence at the correct class leads to lower loss and vice-versa.

https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/

Data regularization techniques

- Modification of the input vectors and internal data and internal parameters of the net
- Targeting to perform better in generalization
- Increases the loss during training phase
- Puts the parameters further away from a minimum with an expectation of it will find a deeper minimum
- In many cases these are heuristic methods with mostly experimental and partial mathematical proof

Input vector normalization

- When the input vector contains high and small mean values in different vector positions it is usefull to normalize them
- Squeezes the number to the same range
- Speeds up the training process

Input vector: $x = \begin{pmatrix} 0{,}45 \\ 1589{,}2 \\ 0{,}00143 \end{pmatrix}$

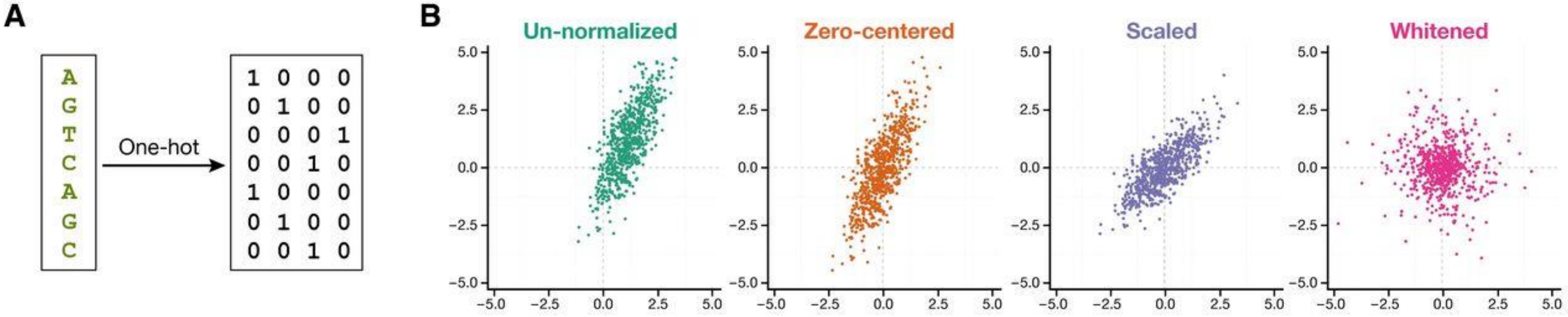
mean: $\bar{x} = \begin{pmatrix} 0{,}32 \\ 1423{,}2 \\ 0{,}00132 \end{pmatrix}$

deviation: $\sigma = \begin{pmatrix} 0{,}11 \\ 155{,}2 \\ 0{,}00042 \end{pmatrix}$

normalized input vector: $x_{normed} = \dfrac{x - \bar{x}}{\sigma} = \begin{pmatrix} 1{,}18 \\ 1{,}06 \\ 0{,}26 \end{pmatrix}$

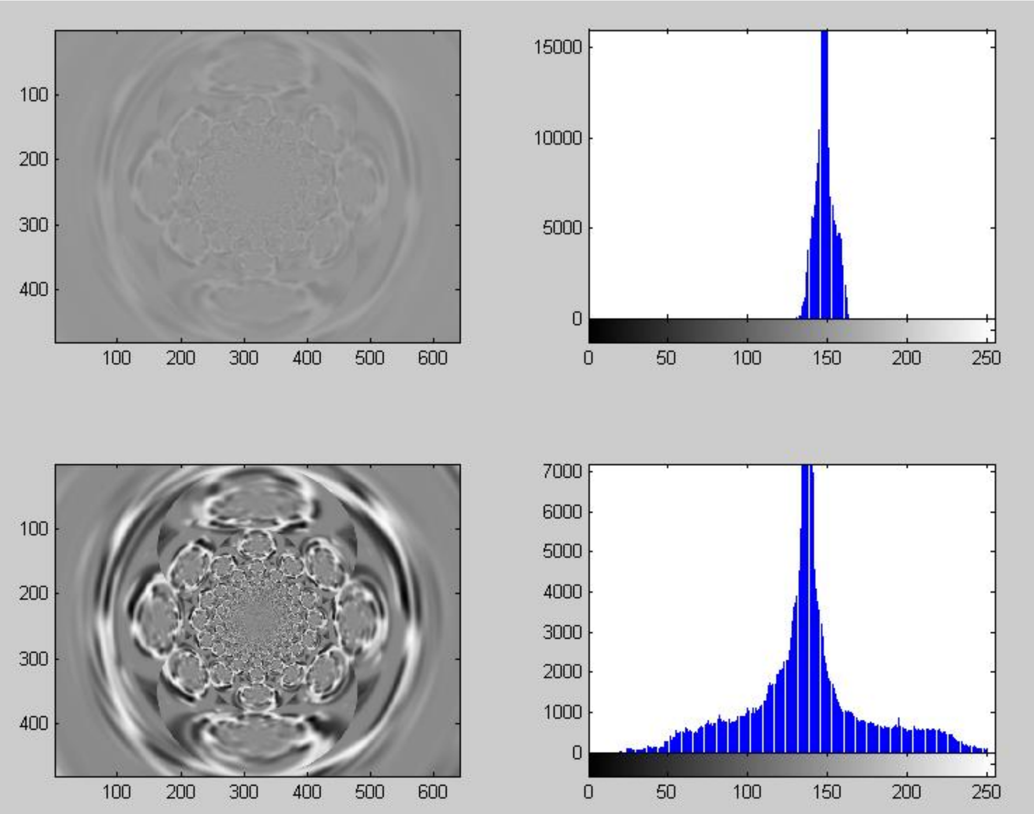
Input Normalization

- Different normalization strategies exists for different input types
- Showing it in two dimension, it shapes the input vector



Once you trained you net with a normalized training set, you have to apply normalization when a previously unseen vector (a new observation) is appled during inference.
OK, but how do you know the statistics?

Input normalization ezample



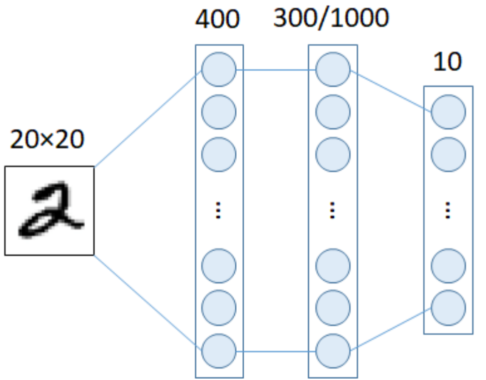
L1 and L2 regularization

- L1, L2, regularization modifies the weights
  - Rather than using MSE cost function
  - An extra term, the MSE of the weights is added (biases excluded)
  - Done on minibatch level
- Can be used with other cost function type as well
- Differenciable: back propagation works
- Why is it good?
  - Network preffers smaller weights
  - If a few large weights dominate the decision the network will lose fine generalization properties
  - In case of large weigths, the decisions are less distributed, the network is less error tolerant

$$C_0 = \frac{1}{2n}\sum_x \|y - a^L\|^2$$

$$C_{L_1} = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{n}\sum_w |w|$$

$$C_{L_2} = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{2n}\sum_w w^2$$



Batch normalization

- In very deep networks the distribution of the input vectors changes from layer to layer
  - The first layer got normalized input
  - The second layer somewhat shifts and twists on this normalization
  - And it goes on, and the (originally normalized) data propagating trough the layers will be lose its normalized properties (called „*covariance shift*”)
  - This will shift the neuron out of its zero centered position, where the activation function performs well (where the nonlinearity is)
- Solution: **normalization on each layers**!
- It also introduce a noise (loss function increase), which helps to avoid local minima and avoids overfitting

Batch Normalization

- Done on layer level like softmax
- Training:
  - Done on minibatch level
- Inferencing:
  - Do the normalization with the pre-calculated parameters of the entire training set
- Batch normalization is differenciable via chain rule
  - Back propagation can be applied for batch normalized layers
- Rewriting the normalization using probability terms:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

E: *the expectation*
Var: the variance

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

bias

weights

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

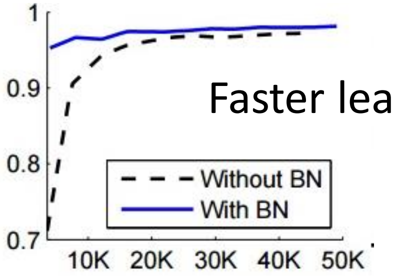
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

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

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

ϵ: avoid zero

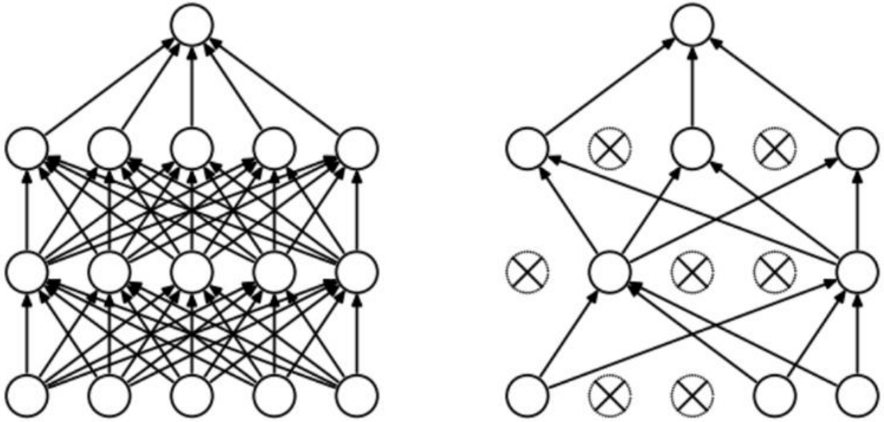
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



Faster learning

Dropout

- Idea of dropout method:
  - Use mini-batch training approach
  - For each minibatch, a random set of neurons from one or multiple hidden layer(s) (called **droppout layers**) is temporally deactivated
  - Selection and deactivation probability is *p*
  - In testing phase, use all the neurons, but multiply all the outputs with p, to account for the missing activation during training
- Requires more training steps, but each is simpler, due to reduced number of neurons
- No computational penalty in testing phase
- Use it for fully connected layers



(a) Standard Neural Net

(b) After applying dropout.

Reduces overfitting, because the network is forced to learn the functionality in different configurations using different neural paths.

Reasoning behind dropout

- Dropout can be considered as averaging of multiple thinned networks (“ensemble”)
- Dropout avoids training separate models
  – Would be very expensive
- Avoids computatinal penalty in the test phase
- But still gets benefits of ensemble methods

Intuitive explanation

Imagine that you have a team of workers and the overall goal is to learn how to erect a building. When each of the workers is overly specialized, if one gets sick or makes a mistake, the whole building will be severely affected. The solution proposed by “dropout” technique is to pick randomly every week some of the workers and send them to business trip. The hope is that the team overall still learns how to build the building and thus would be more resilient to noise or workers being on vacation.

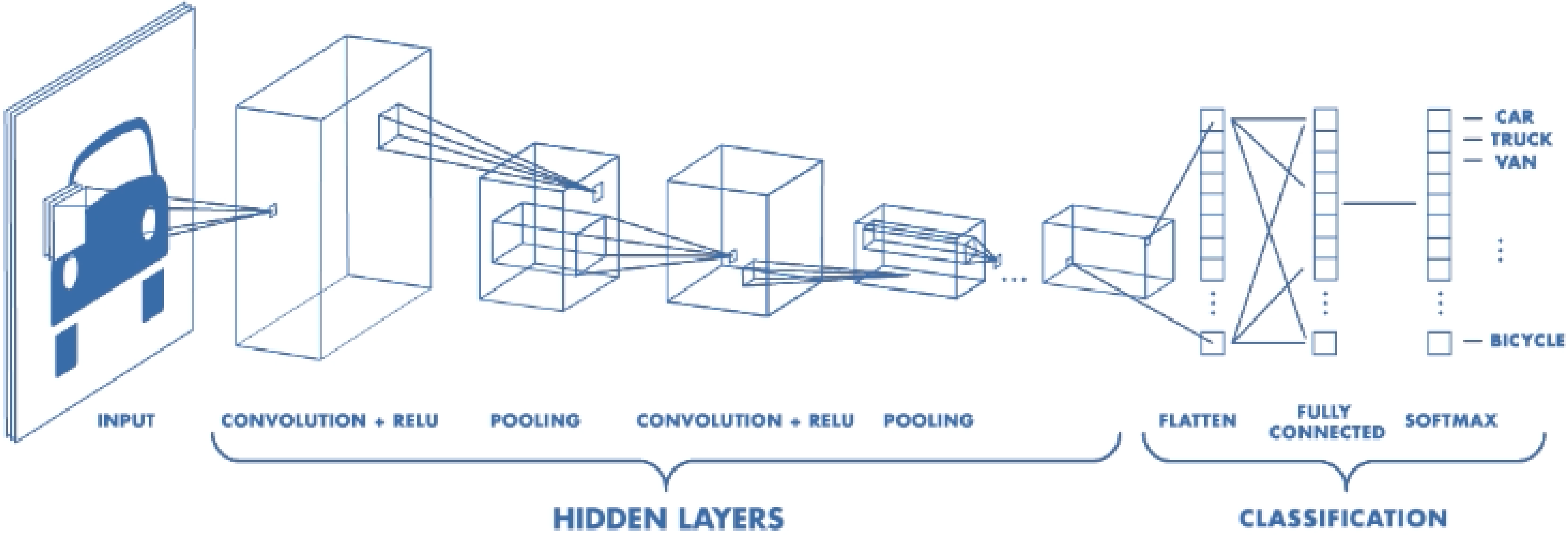
Neural Networks

Components and methods of deep neural networks II

(P-ITEEA-0011)

Akos Zarandy
Lecture 6
October 22, 2019

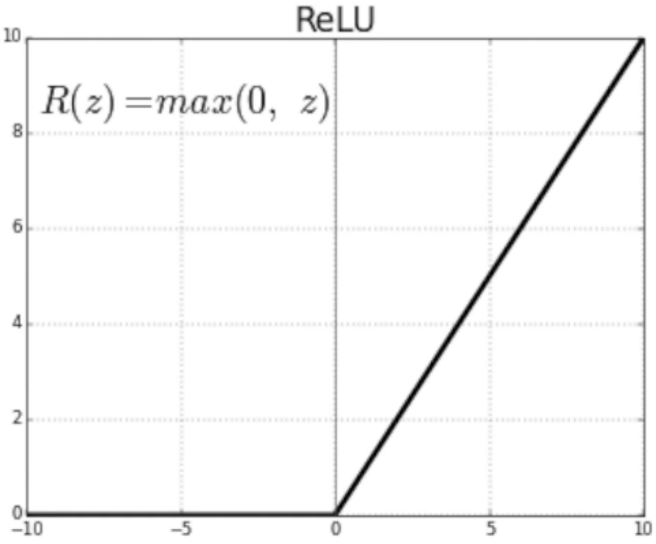
Hierarchical architecture of a deep neural network



What are the building blocks of a hierarchical deep neural network?

ReLU: Rectified Linear Unit

- Activation function
- Half-wave rectifier
- Not compressing the gradient
  - learns much faster
- ReLU types
  - Softmax
  - Leaky ReLU
  - ELU, SELU Relu6



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

ReLU

Most used in hidden layers in deep neural networks (as of 2019)!

Probability type loss: Cross Entropy and Softmax

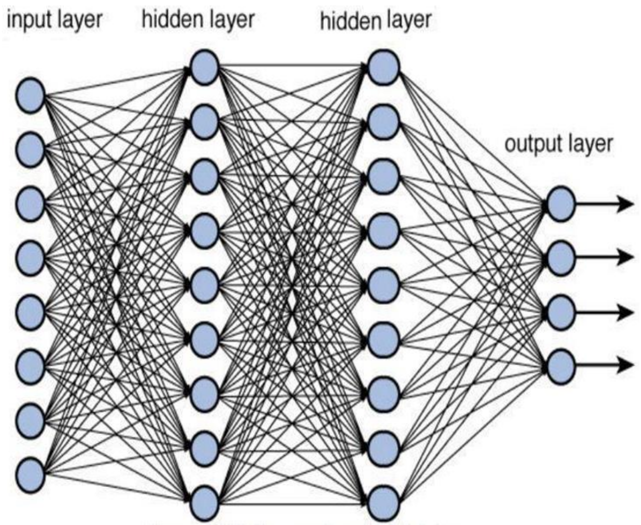
- Mathematically:
  - Normalized exponential functions of the units

$$y_i = softmax(v)_i = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$$

$$\boldsymbol{v} = \boldsymbol{w}^T \boldsymbol{x}$$

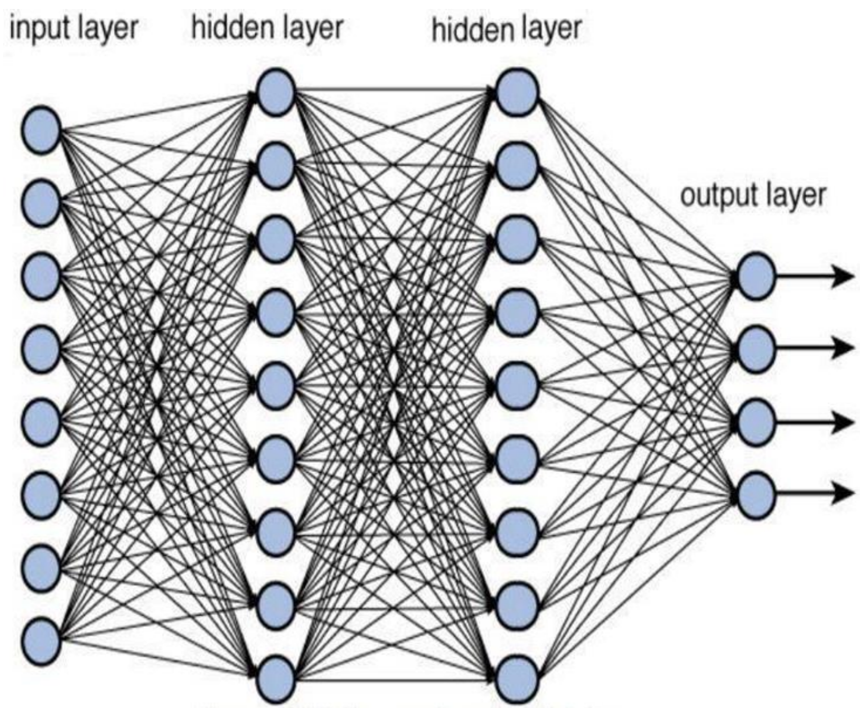
- Probability distribution of n discrete classes:
  - One-of-n classes problems
  - $0 < y_i < 1$
  - $\sum_{i=1}^{n} y_i = 1$

- Architectural difference:
  - Previously learned activation functions were based on the inputs of one neuron
  - Softmax combines a layer of output neurons



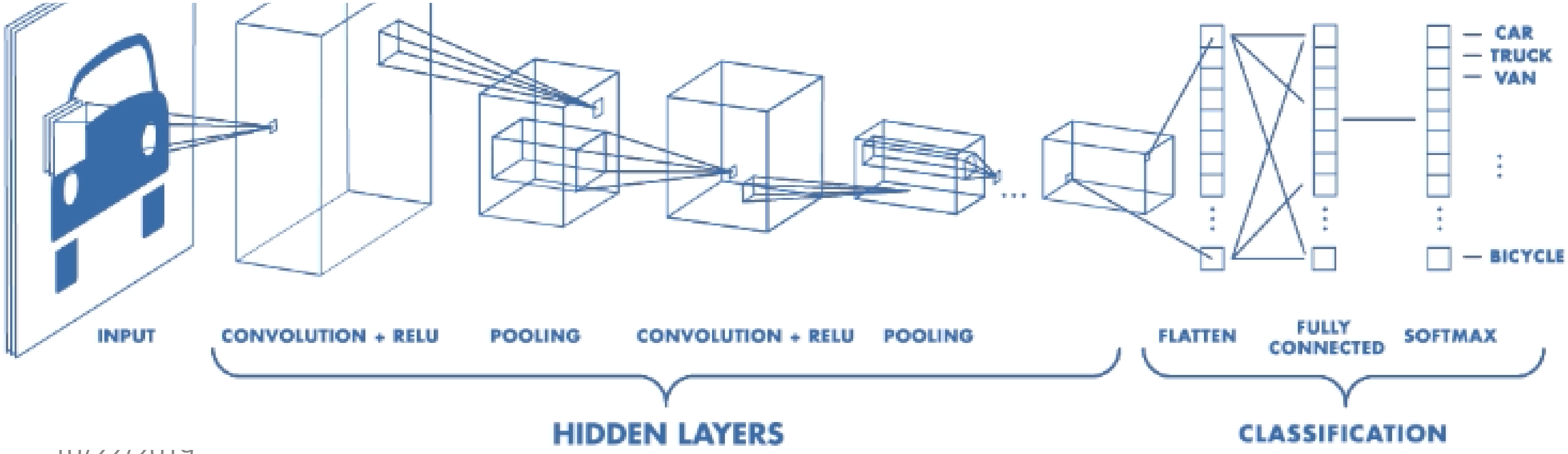
Data regularization techniques

- Modification of the input vectors or the internal data composition of the network
  - Input normalization
  - Batch normalization
- Modification of the cost function (involving the weight magnitudes)
  - L1 and L2 regularization (weight penalty)
- Temporal Modification of the net architecture in training phase
  - Dropout



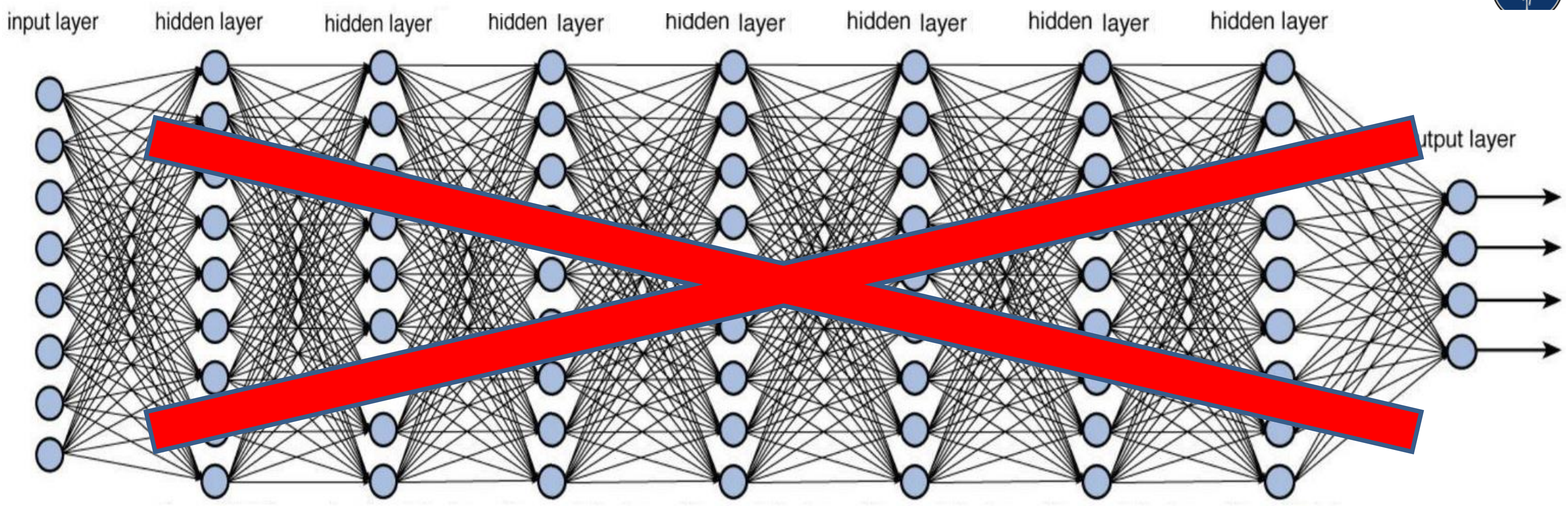
Contents

- Reducing the number of interconnections
  - Biological motivations
- Convolution
- Convolution layers in deep networks
- Pooling
- Regularization methods



No-brainer solution: Increase the number of the hidden layers

DEEEEEEP

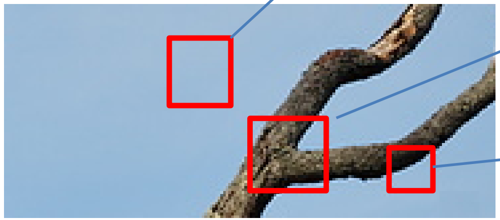


- Problems:
  - Not usefull for locally correlated data
  - Number of free parameters are exploding

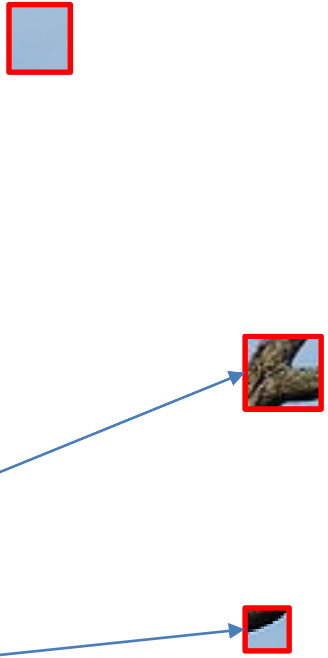
Locallity

- Spatial locallity:
  - Data points measured physically close to each other
  - e.g. image measured by a sensor array
  - Measurements, close to each other are similar (correlated)
  - Local feature: where local similarity is broken

uncorrelated



correlated



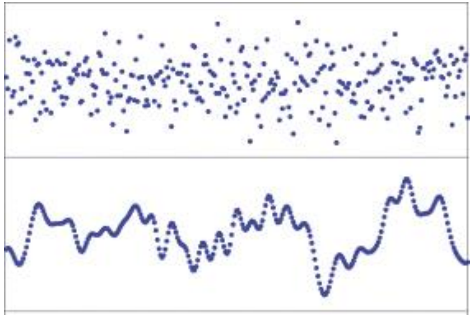
Locallity

- Spatial locallity:
  - Data points measured physically close to each other
  - e.g. image measured by a sensor array
  - Measurements, close to each other are similar (correlated)
  - Local feature: where local similarity is broken

uncorrelated

correlated

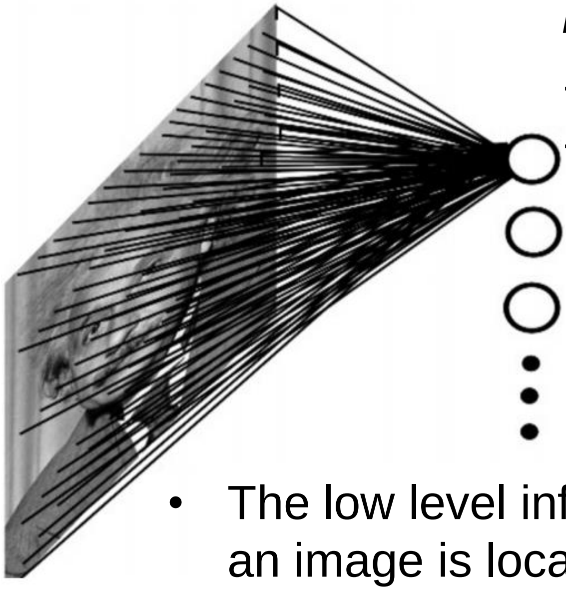
- Temporal locallity:
  - Data point sequence measured with the same sensor with small time difference
  - e.g. voice measured by a microphone
  - Measurement points, close to each other are similar (correlated)



Uncorrelated data series (noise)

Correlated data series (continiuous signal)

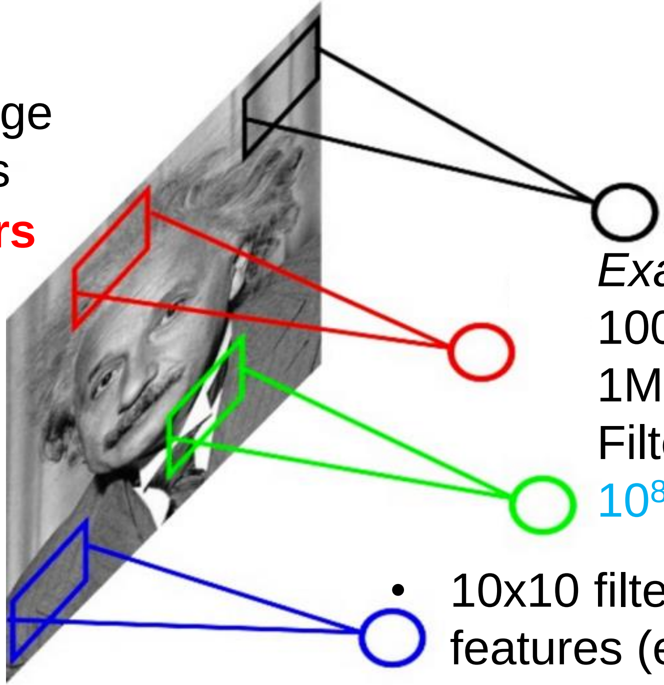
FULLY CONNECTED NEURAL NET



Example:
1000x1000 image
1M hidden units
10^{12} parameters

- The low level information on an image is local
- Makes no sense to involve distant pixels to the same function

LOCALLY CONNECTED NEURAL NET

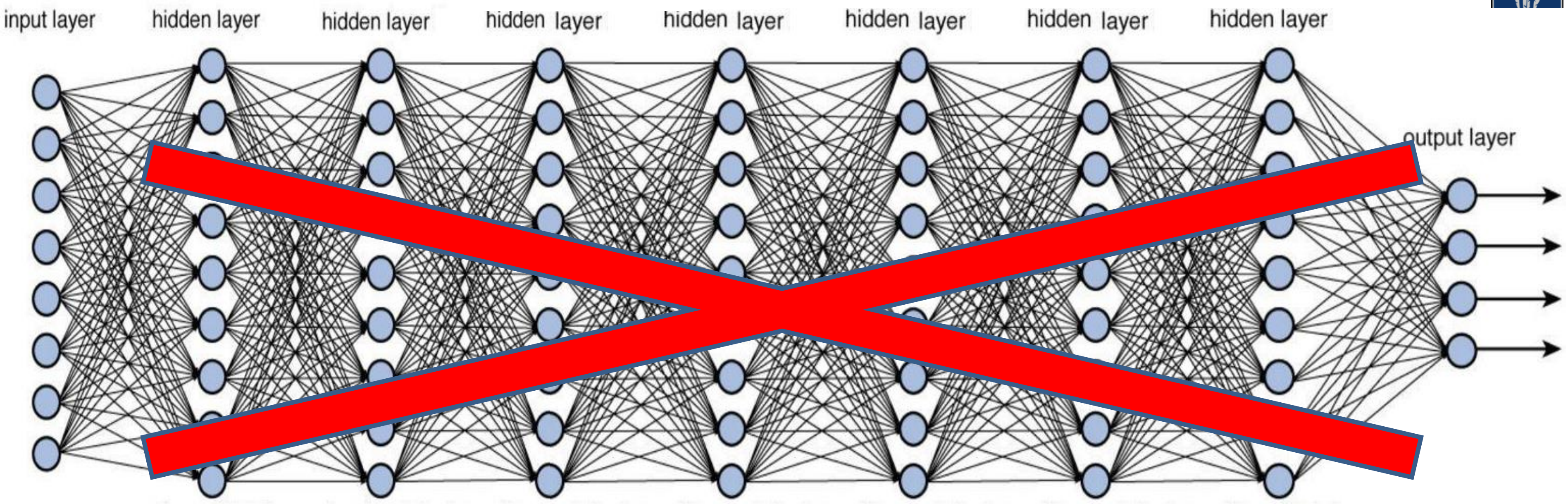


Example:
1000x1000 image
1M hidden units
Filter size: 10x10
10^{8} parameters

- 10x10 filters finds local features (edges)
- Why to apply different filters in different location?
- How do I know where to expect the edges?

No-brainer solution: Increase the number of the hidden layers

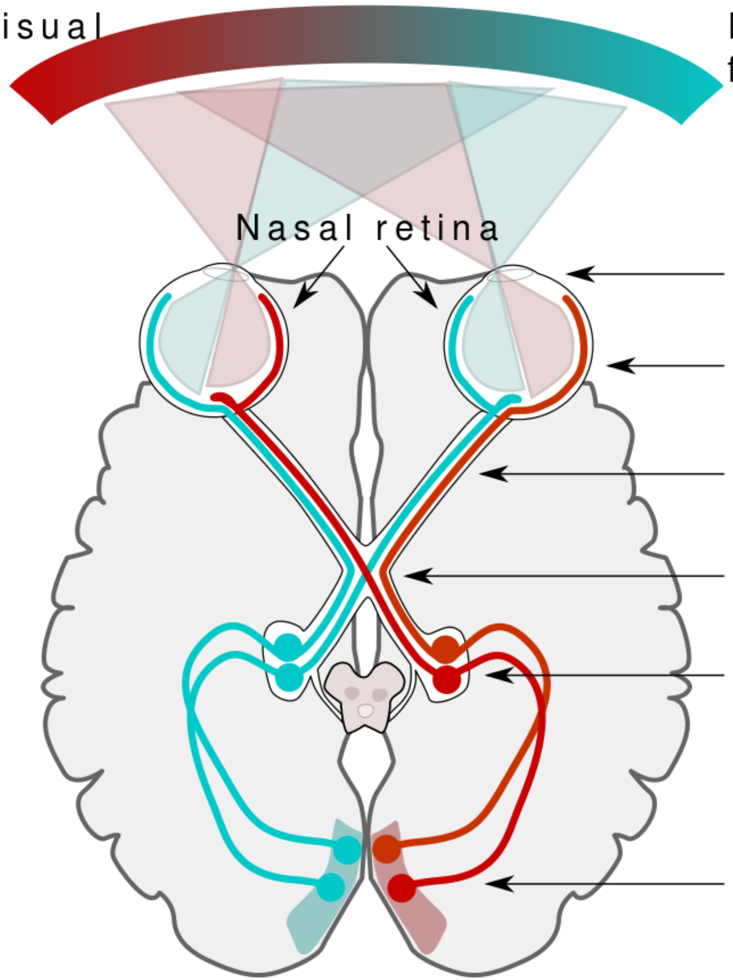
DEEEEEEP



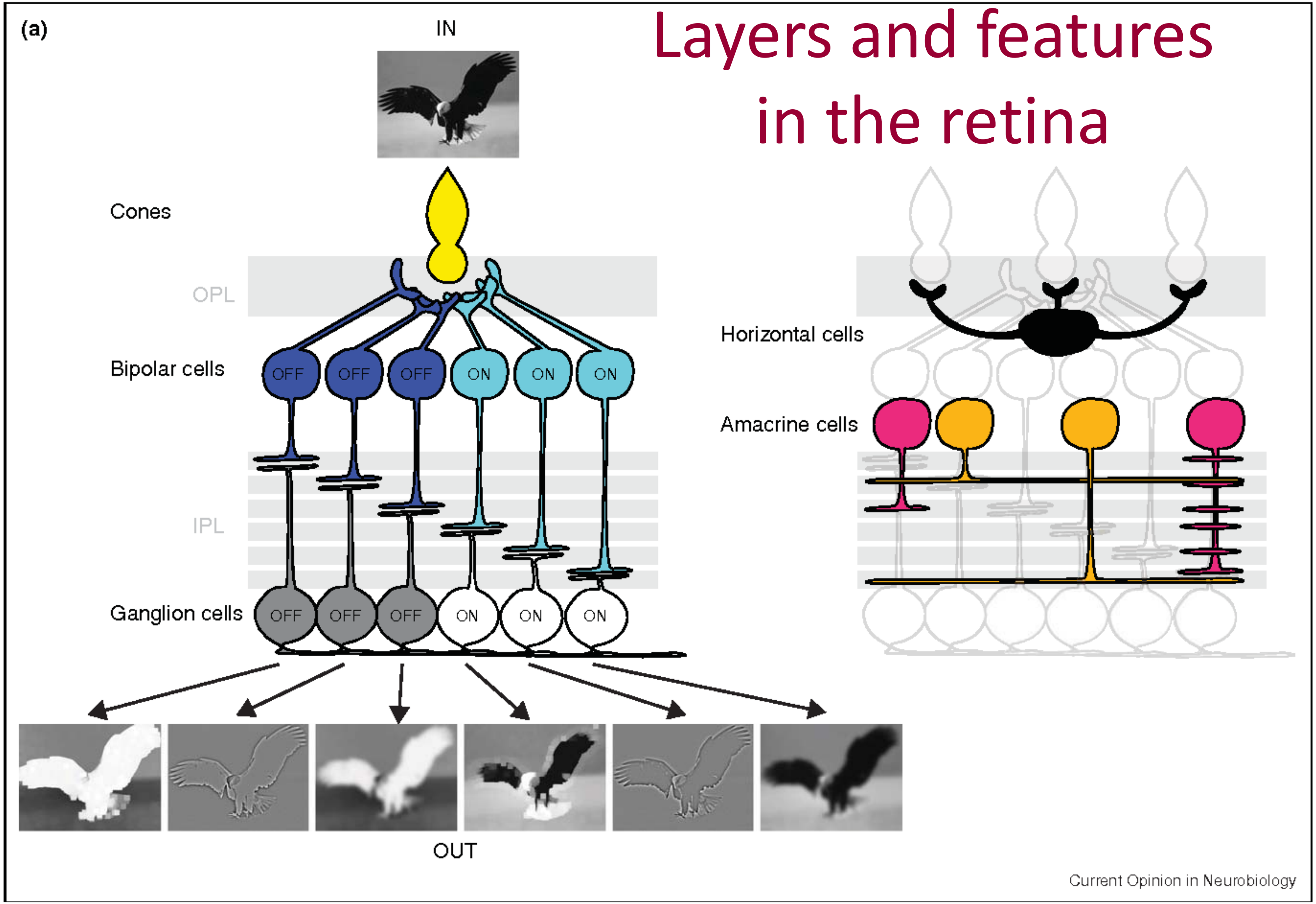
- Architecturel problem:
  - Why would be optimal to use one linear arrangement using the same data width everywhere?
  - Parallel, loop?
  - How human visual system does it?

What can we learn from Human Visual System?

- Hierarchy
  - Eyes, LGN, Visual Cortex
- Each organized to parallel layers
- Each responsible for extracting a different feature
- Fraction of the sensed data is transferred only
  - Image features and motions

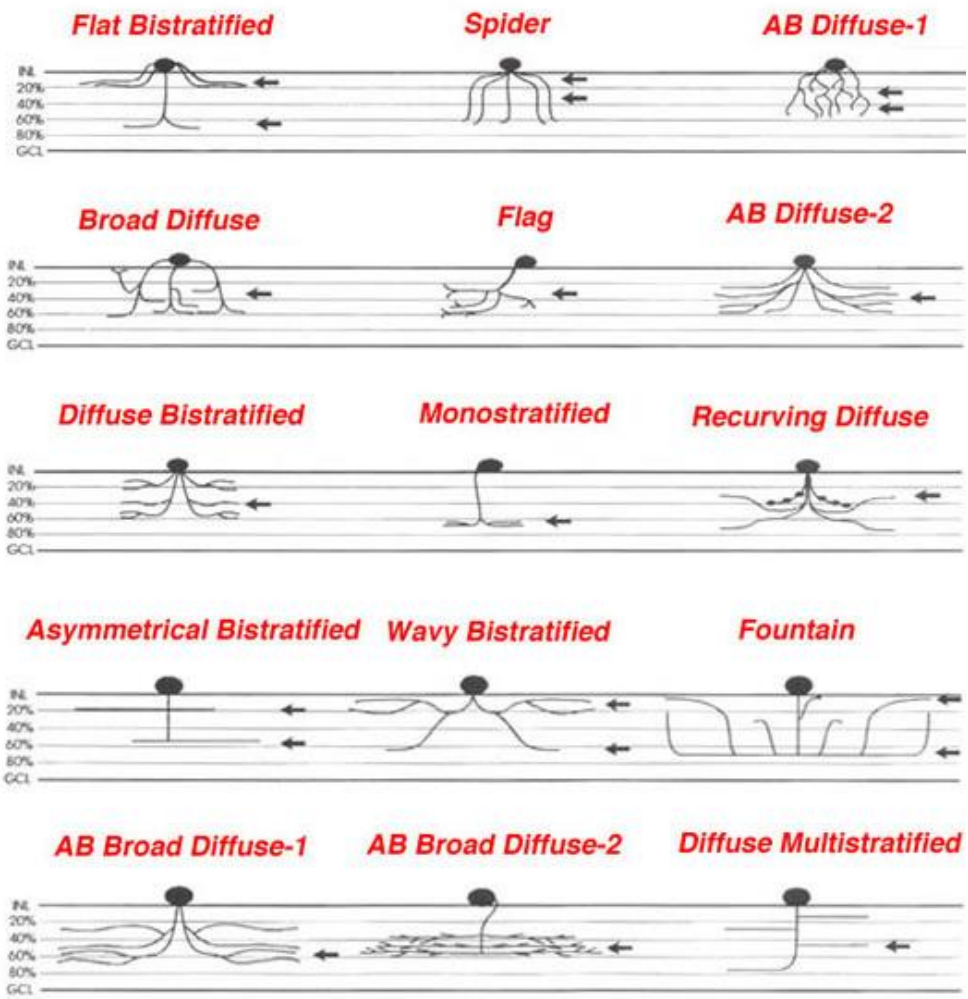


Layers and features in the retina



Retina cell layers

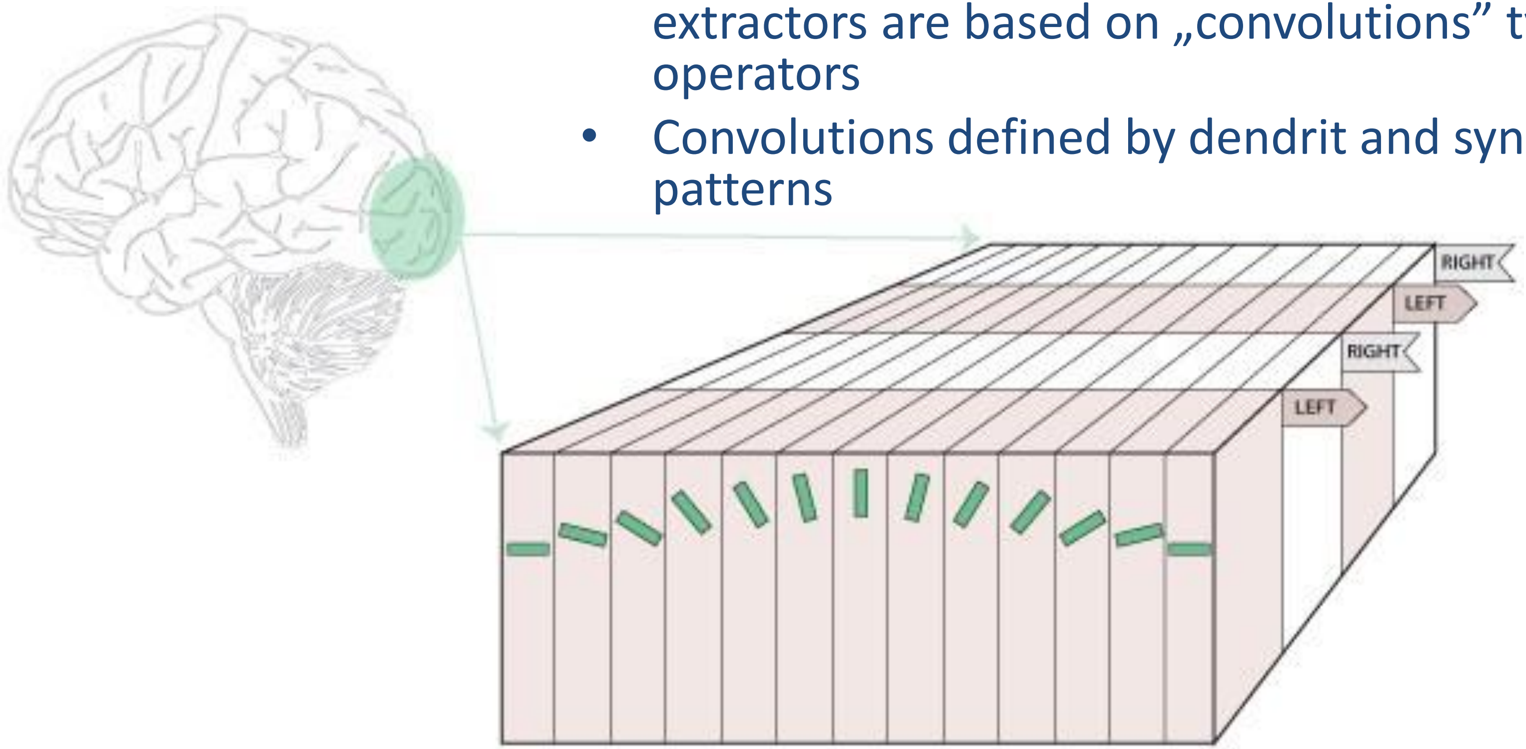
- Similar cells are forming layers (filter)
- A layer extracts the same local feature from the entire sensed image with convolution type operations
  - Contrast changes, color differences, motion direction, orientation
  - Dendritic tree and synapse weights defines the captured features
- Outputs are organized in separate channels



https://webvision.med.utah.edu/book/part-iii-retinal-circuits/roles-of-amacrine-cells/

Visual cortex

- Parallel blocks identifying edges with different orientation
- Both the retinal and the cortical local feature extractors are based on „convolutions” type operators
- Convolutions defined by dendrit and synapse patterns

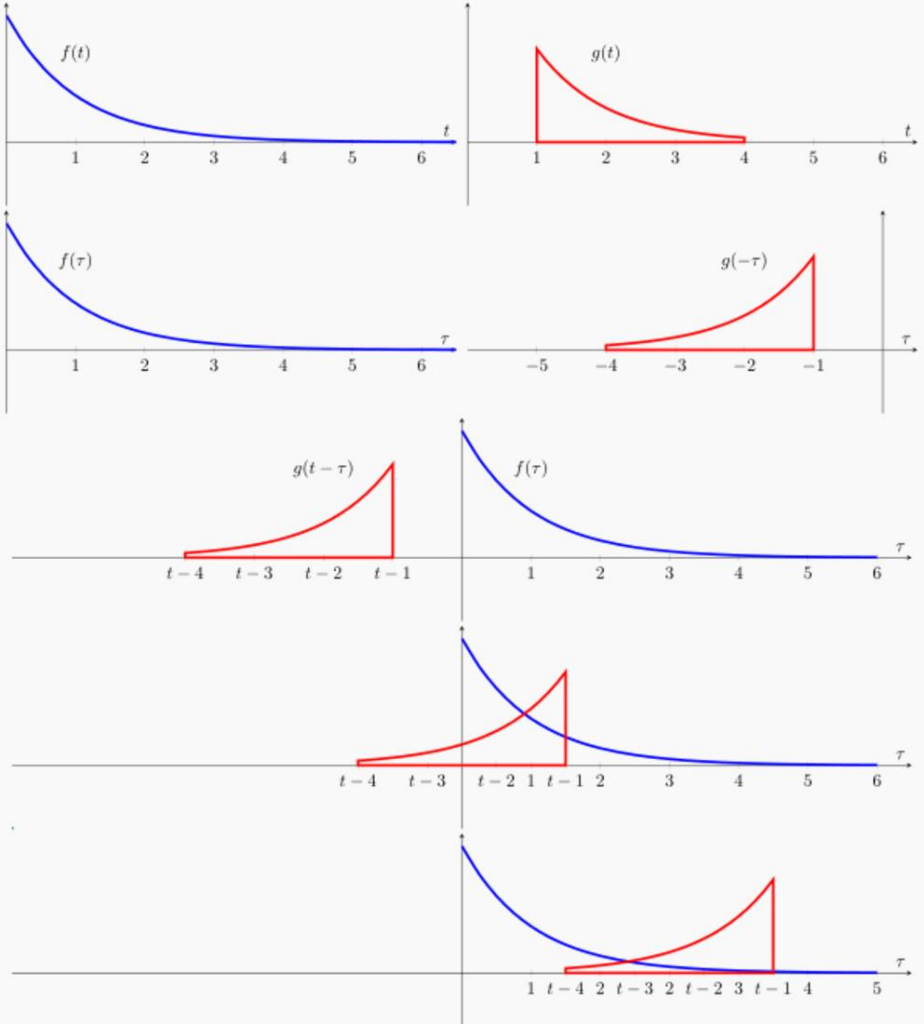


Convolution

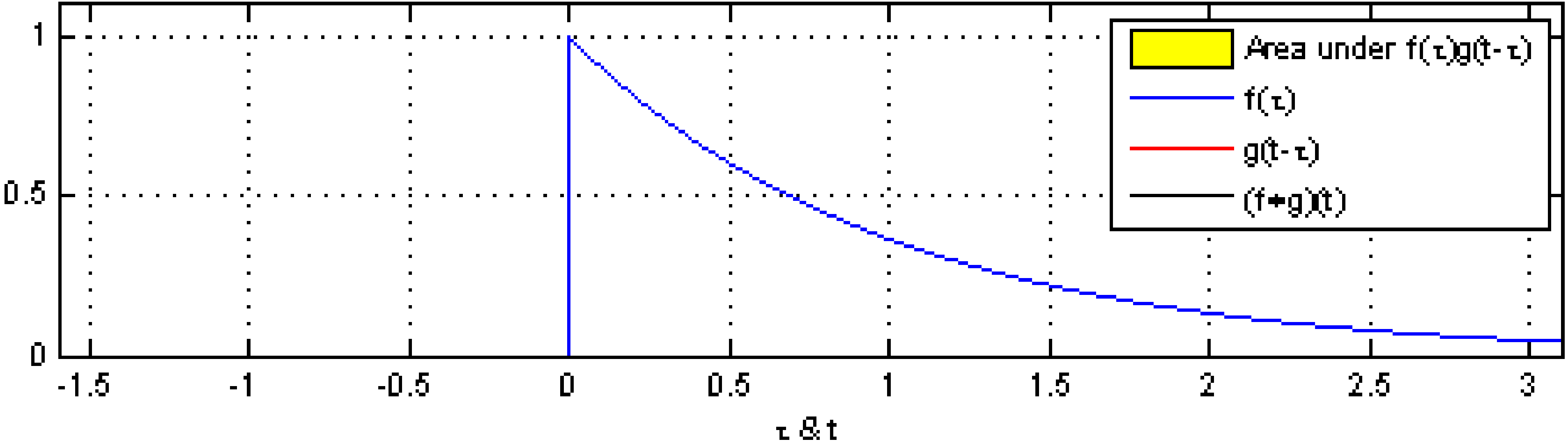
- Convolution is a mathematical operation that
  - does the integral of the product of 2 functions (signals),
  - with one of the signals flipped and shifted
- Mathetmatically:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau) g(\tau)\, d\tau$$

- Convolution is commutative



Visualization in 1D



1. Flipp *g* signal
2. Slide the flipped *g* over *f*
3. Integrate the product in continious space
   or Multiply and accumulate it in discrete space with each shift

Discrete convolution

- For continiuous:

$$(f*g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau$$

- For discrete functions:

$$(f*g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$
$$= \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

1D Numerical example

f function:

0	1	2	3	4

g function:

1	-1	2

flipped g function:

2	-1	1

Shifting the flipped g function over f

$f * g =$	0	1	1	3	5	2	8

1D Numerical example

f function:

0 1 2 3 4

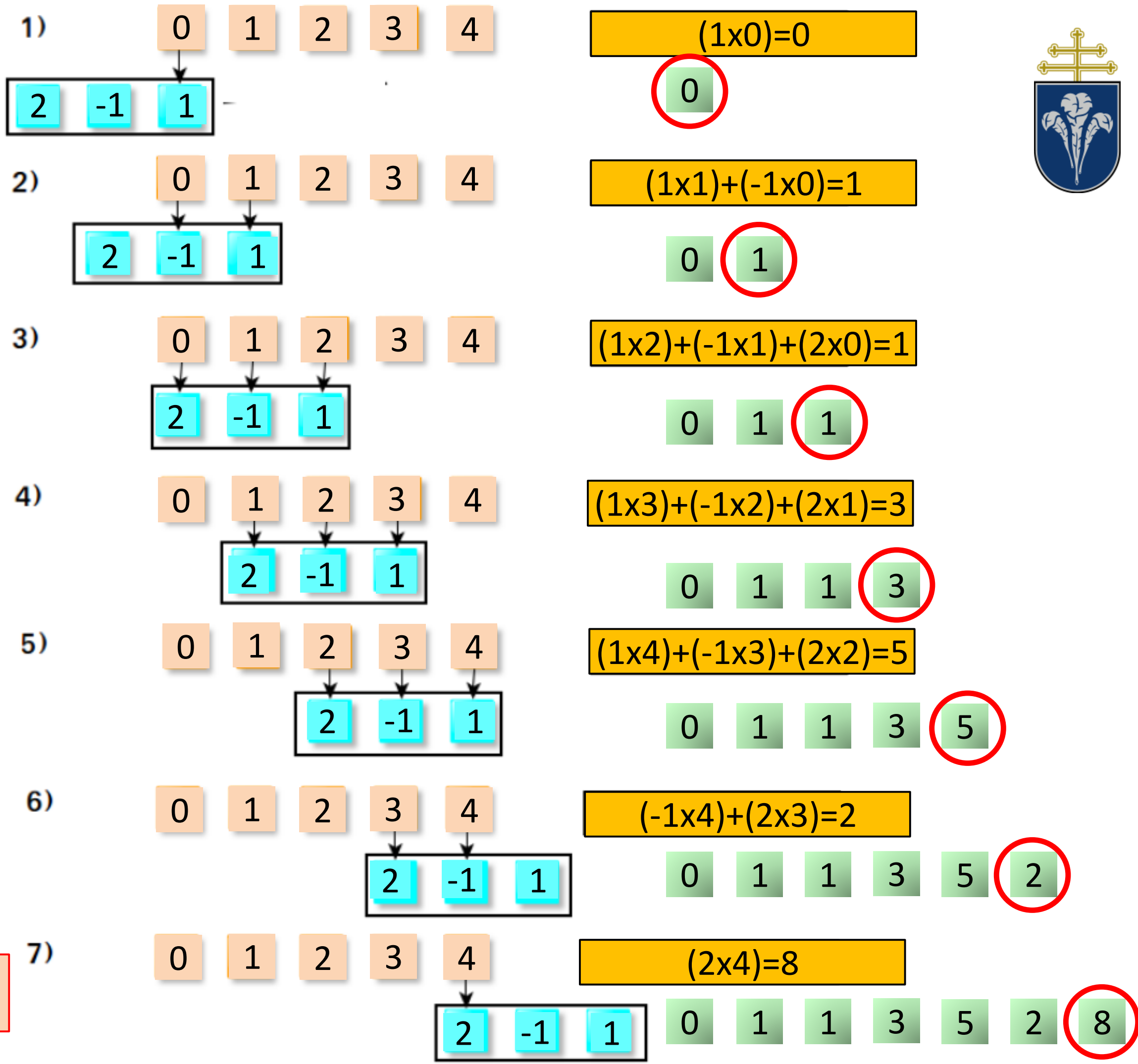
g function:

1 -1 2

flipped *g* function:

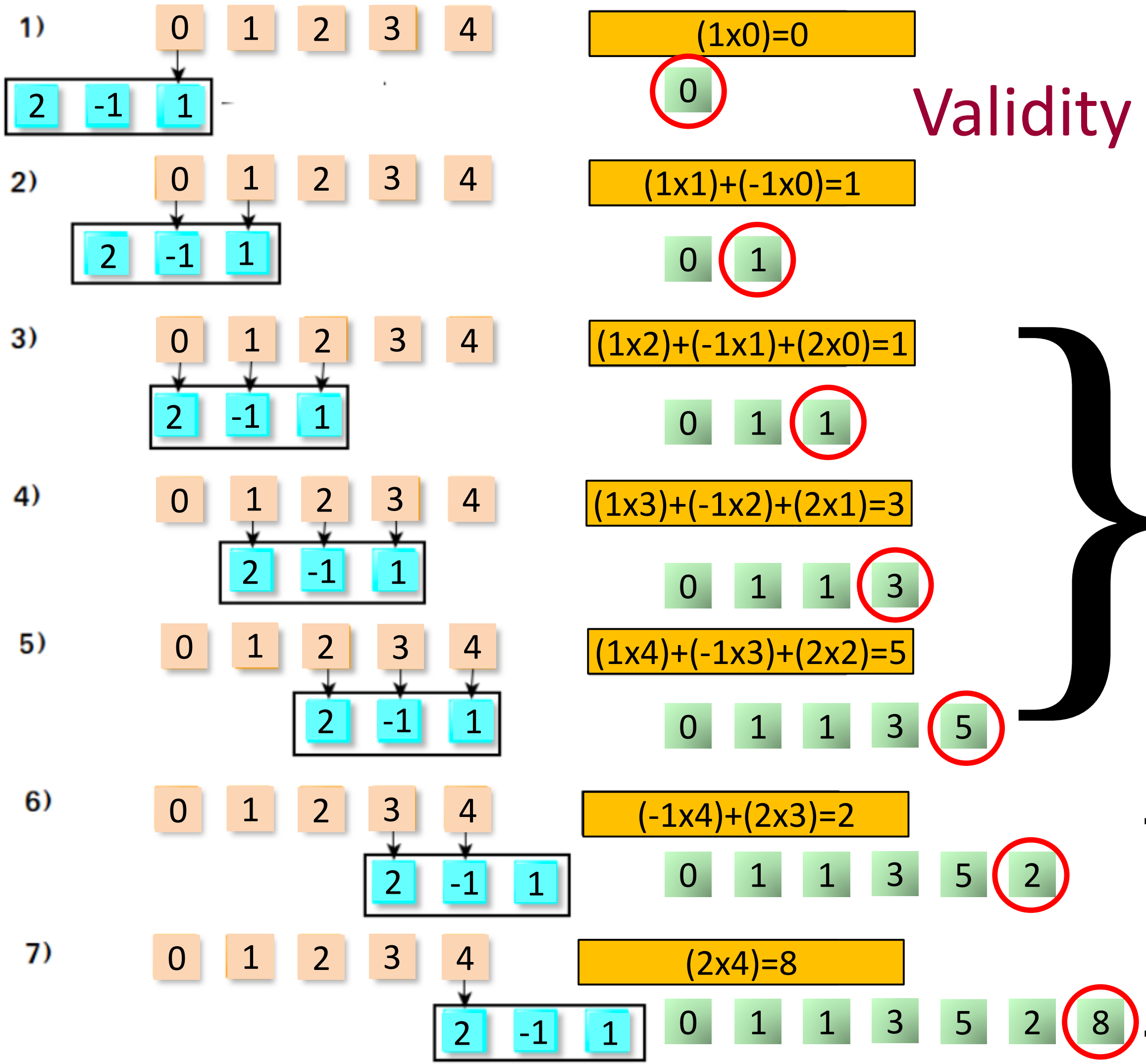
2 -1 1

Shifting the flipped *g* function over *f*



$f * g =$ 0 1 1 3 5 2 8

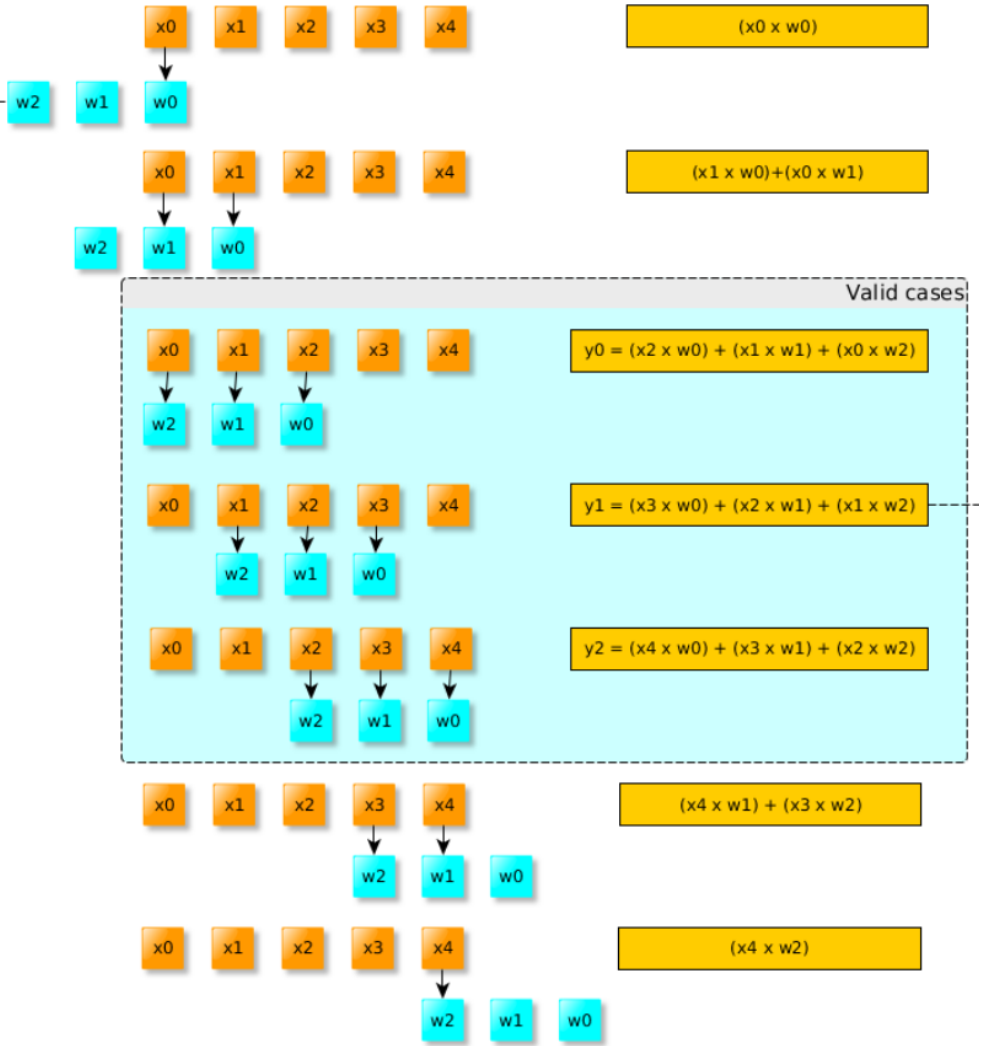
Validity vs Boundary position

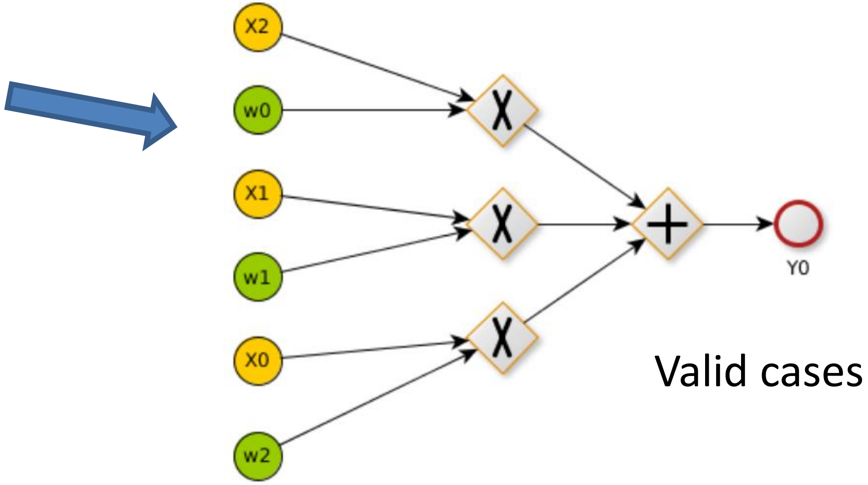


Def.: Valid positions: the flipped *g* is completely inside *f* *(fully overlapping positions)*

Def.: Boundary positions: *partially overlapping positions*

Computation graph



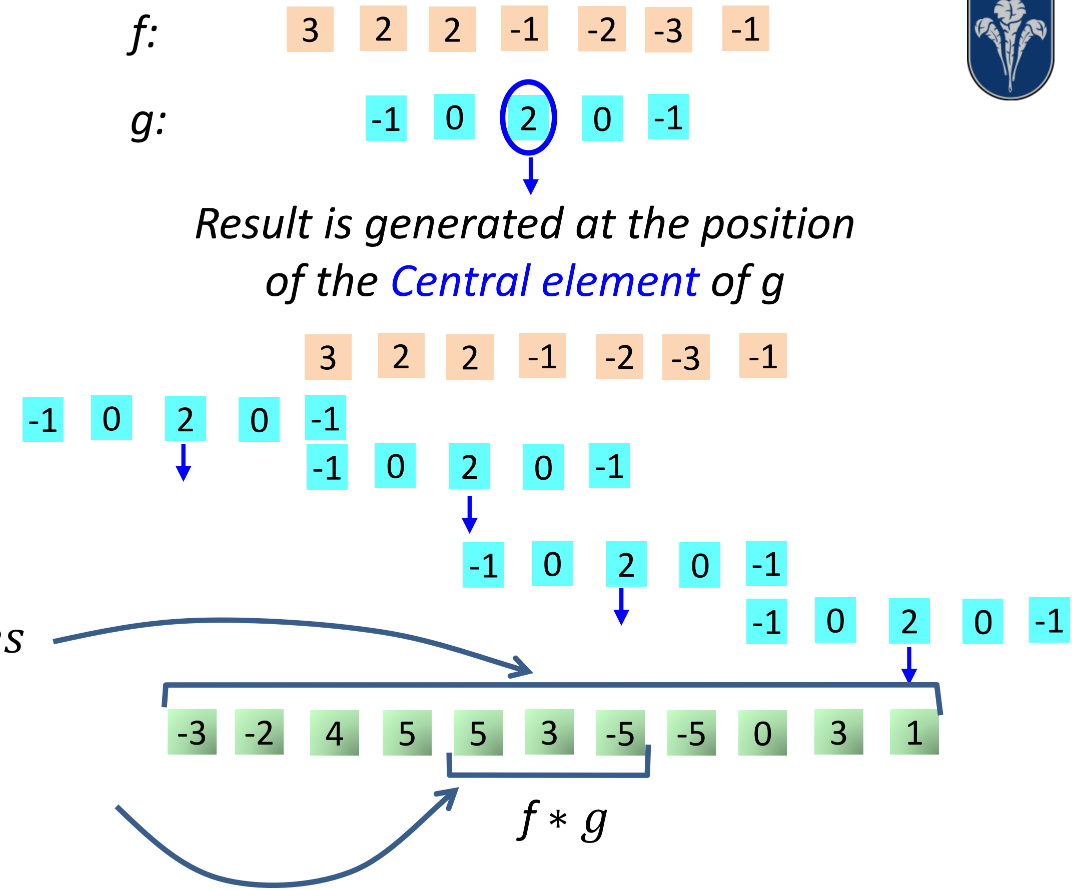


Size of the result

- In practice, convolution is used as a ***filter***, where
  - *f* is the measrurement data, *g* is the filter function descriptor (***kernel***)
  - size(*f*) ≫ size(*g*)

size(*f*)= n   size(*g*)= *k*   n≥*k*

$$\text{size}(f * g) = \begin{cases} n + k - 1 \quad (if\ all\ values\ counted) \\ \\ \end{cases}$$



In CNN, we calculate the valid values only!

Padding in 1D

- In many case, we use a sequence of convolution filters on the measured data blocks
- We do not want size changes on the data blocks
- To avoid size changes, we have to pad the data block with zeros at the boundaries
  - *k=size(g)* is odd: k=2p+1

    | p | n | p |

  - *k=size(g)* is even: k=2p
    - Padding is asymmetric:

      | p | n | p-1 |

f: 3 2 2 -1 -2 -3 -1

g: -1 0 2 0 -1

central element

side elements

g: -1 0 2 0 -1

Padded f: 0 0 3 2 2 -1 -2 -3 -1 0 0

Valid $f * g$ after padding: 4 5 5 3 -5 -5 0

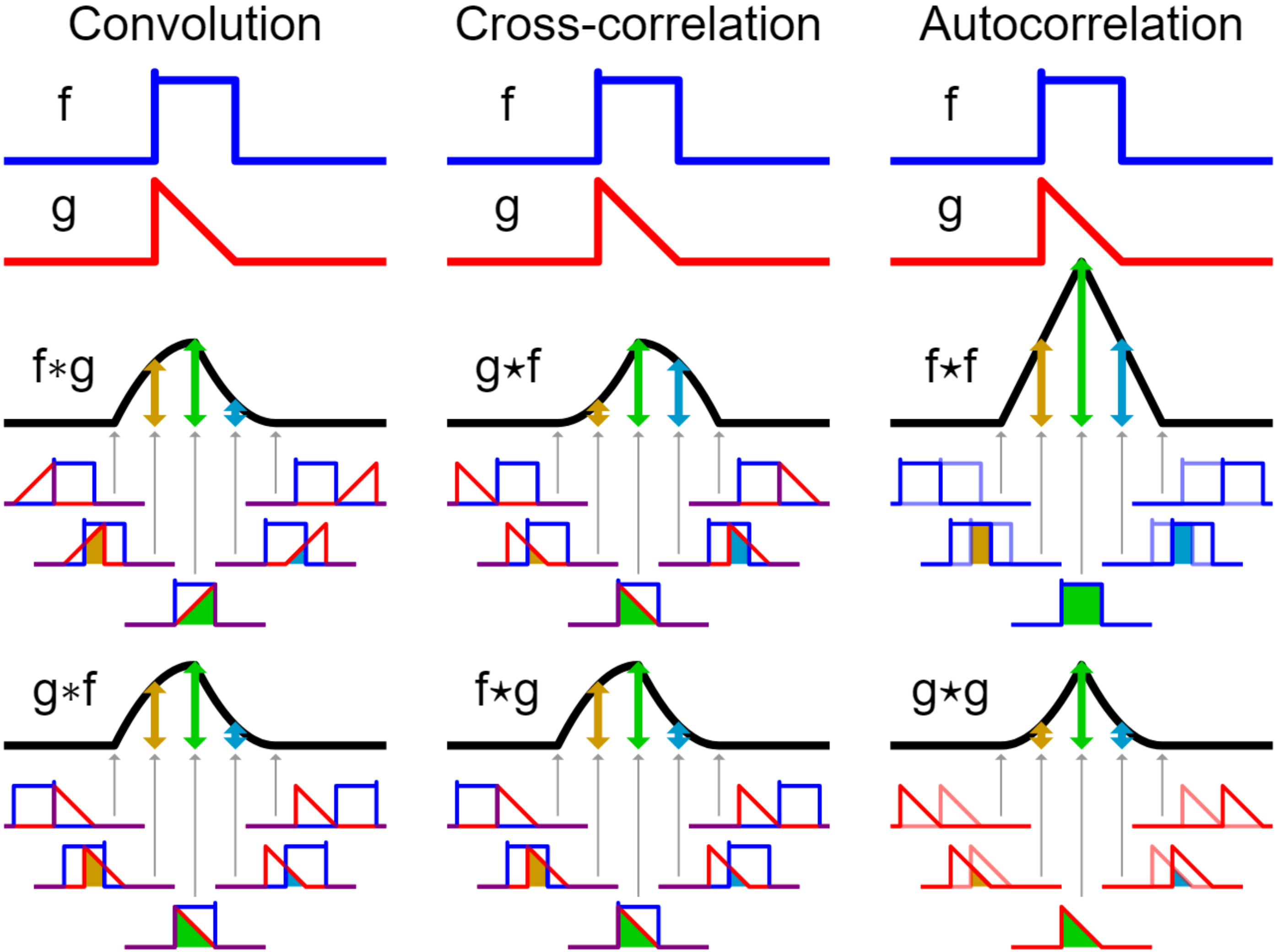
Convolution vs correlation I

- Cross-Correlation:

$$(f \star g)(\tau) \stackrel{\text{def}}{=}$$

$$= \int_{-\infty}^{\infty} f^*(t)\, g(t+\tau)\, dt$$

- f^* : complex conjugate
- When f is symmetric, $f * g = f \star g$ (otherwise not)



Convolution vs correlation II

- As the only difference is kernel flipping…
- Why convolution rather that correlation?
  - Commutativity, Associativity, Distributivity helps to prove mathematical statements
  - Since the network learns its own weights, it is invariant whether that flip is there or not (just a convention)
  - In many cases, correlation is implemented even when it is called convolution

2D convolution

$f \quad * \quad g$

$$\begin{bmatrix} 1 & 3 & 1 \\ 0 & -1 & 1 \\ 2 & 2 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$$

input

kernel

Scanning through the *f* function with the flipped *g* function

flip(kernel)

$$\begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \boxed{1} & \boxed{3} & 1 \\ \boxed{0} & \boxed{-1} & 1 \\ 2 & 2 & -1 \end{bmatrix}$$

1(-1)+3(0)+0(2)-1(1)=-2

$$\begin{bmatrix} 1 & \boxed{3} & \boxed{1} \\ 0 & \boxed{-1} & \boxed{1} \\ 2 & 2 & -1 \end{bmatrix}$$

3(-1)+1(0)-1(2)+1(1)=-4

$$\begin{bmatrix} 1 & 3 & 1 \\ \boxed{0} & \boxed{-1} & 1 \\ \boxed{2} & \boxed{2} & -1 \end{bmatrix}$$

0(-1)-1(0)+2(2)+2(1)=6

$$\begin{bmatrix} 1 & 3 & 1 \\ 0 & \boxed{-1} & \boxed{1} \\ 2 & \boxed{2} & \boxed{-1} \end{bmatrix}$$

-1(-1)+1(0)+2(2)-1(1)=4

result(valid)

$$\begin{bmatrix} -2 & -4 \\ 6 & 4 \end{bmatrix}$$

2D convolution: Example 2

$$g = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

kernel

1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

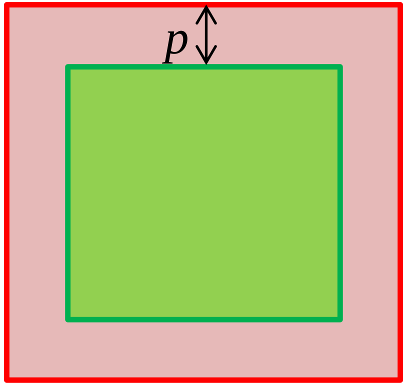
Image

4		

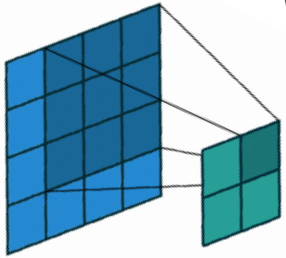
Convolved Feature

Padding in 2D

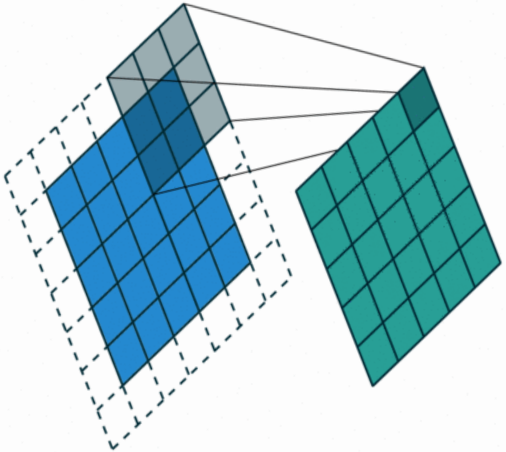
- Works the same way as in 1D
  - Boundary layers are added and filled up with zeros
  - Size *g* is *k* x *k*,
    - *where:* $k=2p+1$
  - Padding: p layers of zeros



Convolution without padding (valid results)

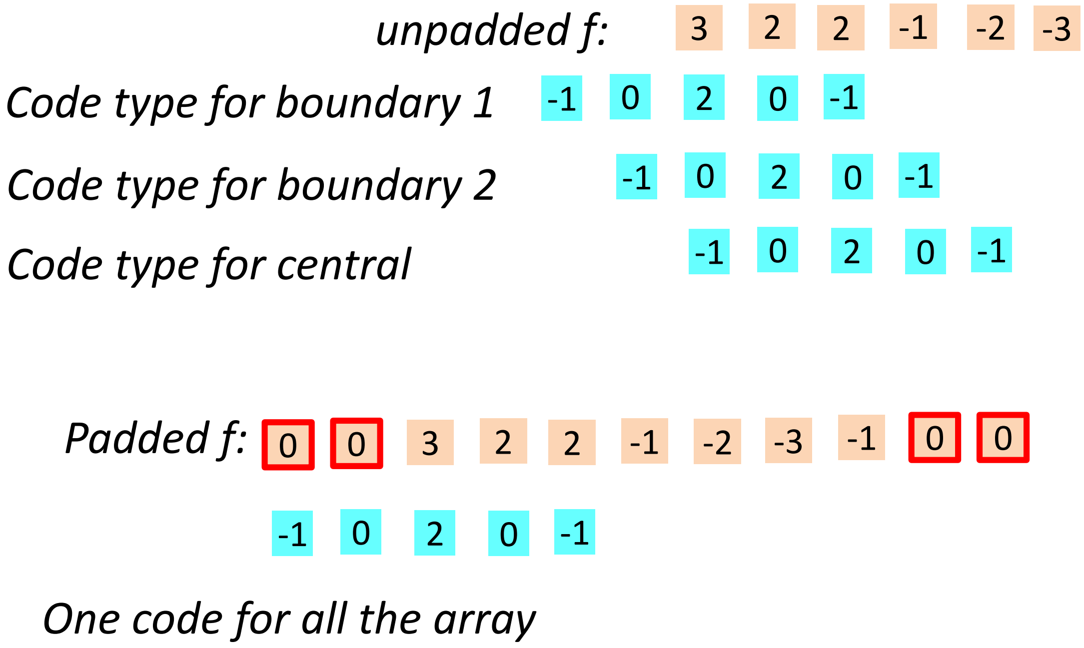


Convolution with padding (size unchanged)



Why use padding?

- Simplifies the execution code
- No branches
- Do not have to deal with the different calculation methods at the boundaries
- Same code runs in the entire array



Though it is more multiply-add operation, but as *f>>g* a branch free simpler code is more efficient

Parameter number and computational load

- Number of trainable free parameters:
  - k in 1D convolution | size(g)= k
  - k^2 in 2D convolution | size(g)= $k \times k$
- Operation number
  - $k*n$ for a padded 1D convolution | size(f)= n
  - $k^2 * n^2 = O(n^2)$ for a padded 2D convolution | size(f)= $n \times n$

Convolution theorem

- Convolution in the Fourier domain is a multiplication

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

and also:

$$\mathcal{F}\{f \cdot g\} = \mathcal{F}\{f\} * \mathcal{F}\{g\}$$

where:

$\mathcal{F}\{f\}$ is the Fourier transform for f

f can be vector or matrix

$\cdot$ is point-wise multiplication

- Therefore:

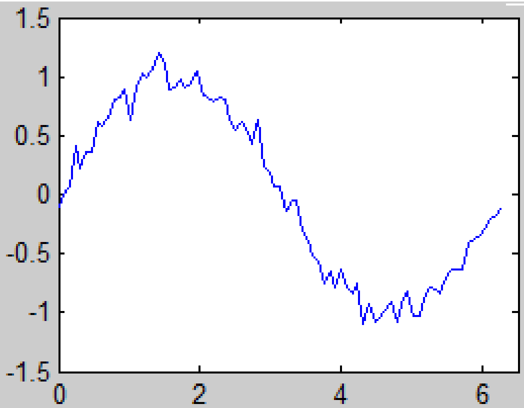
$$\boxed{f * g = \mathcal{F}^{-1}\{\, \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}}$$

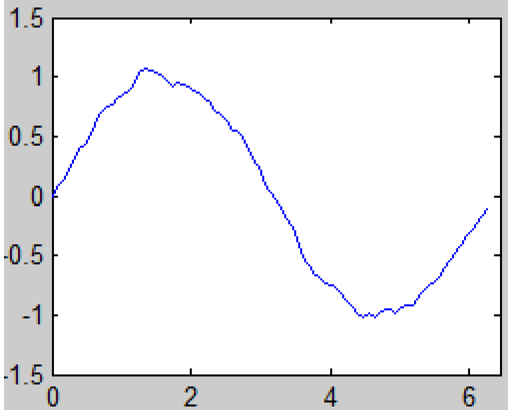
$$f \cdot g = \mathcal{F}^{-1}\{\, \mathcal{F}\{f\} * \mathcal{F}\{g\}\}$$

Convolution can be calculated with a Fourier and an inverse Fourier transformation and a point-wise multiplication. It reduces the computational complexity from $O(n^2)$ to $O(n \cdot \log n)$. (using FFT, assuming $n=2^i$)

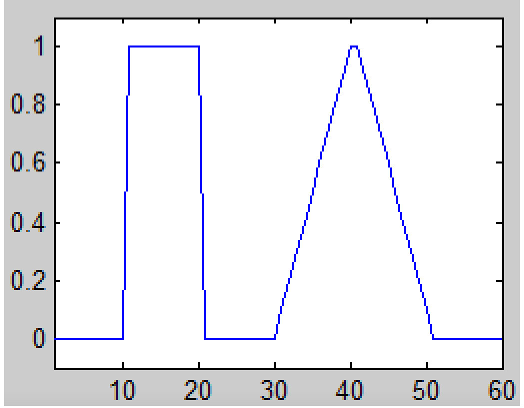
Usage of convolution I : 1D filtering

Smoothing noisy signal
Data lengths: 80 points
kernel: $\frac{1}{5}[\,1\;\;1\;\;1\;\;1\;\;1\,]$

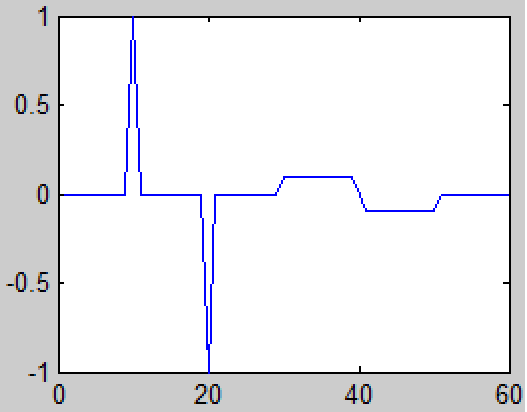




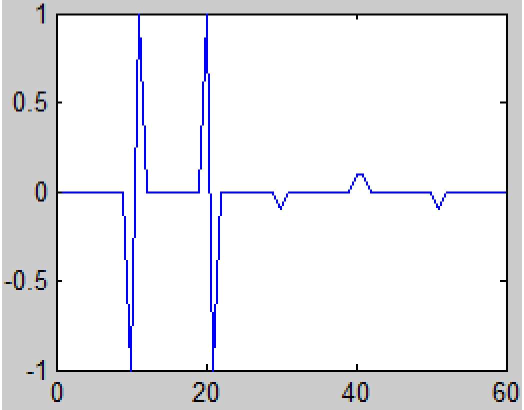
Signal differentiation



Data lengths: 60 points



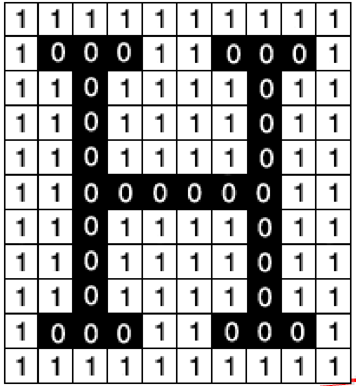
$\frac{dy}{dx}$ kernel: $\frac{1}{2}[1\;\;-1]$

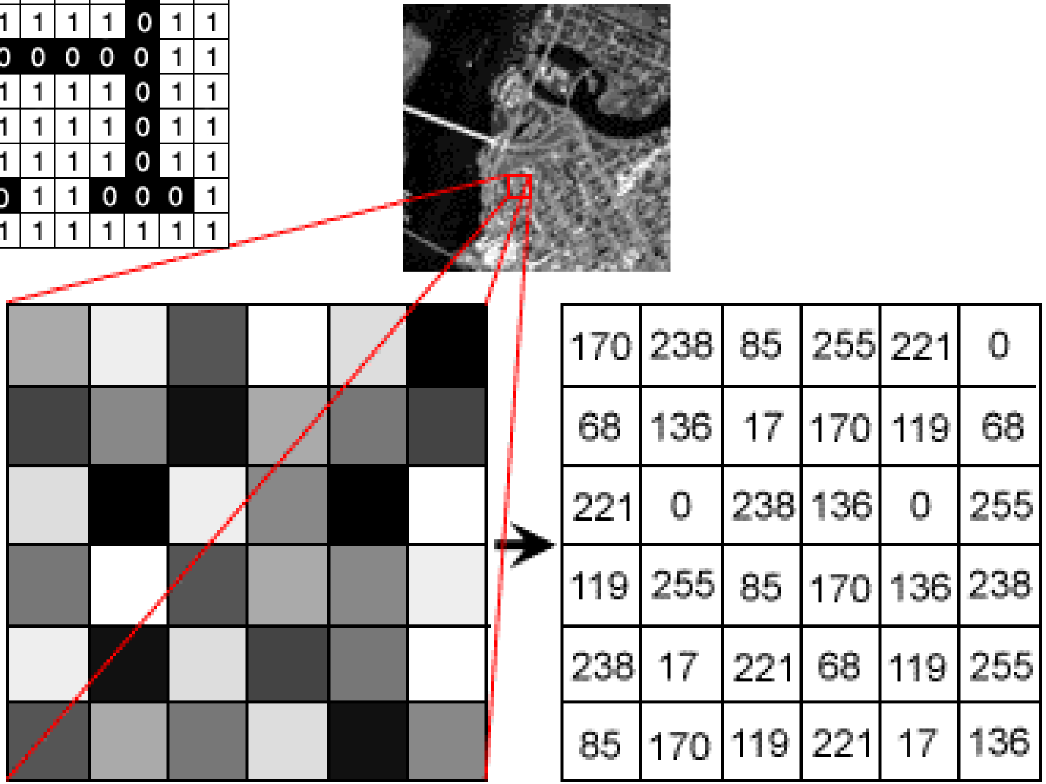


$\frac{d^2y}{dx^2}$ kernel: $\frac{1}{4}[-1\;\;2\;\;-1]$

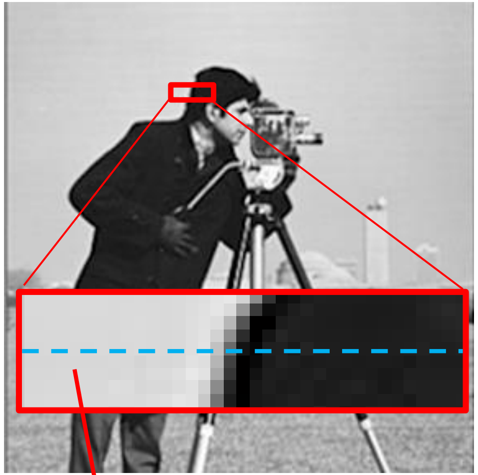
2D convolution: image filtering

- *What is a digital image?*
  - One-to-one mapping of a matrix and the pixels
  - Black-and-white image
    - Binary matrix
    - 0: black
    - 1: white
  - Monochrome (grayscale)
    - Matrix of (typically) 8 bit numbers
    - Values representing the brightness of the pixel
  - Color image
    - 3 matrices (R,G,B)

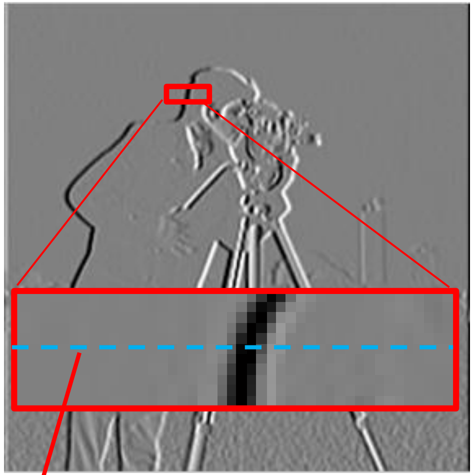




Usage of convolution II : 2D filtering
Sobel operation

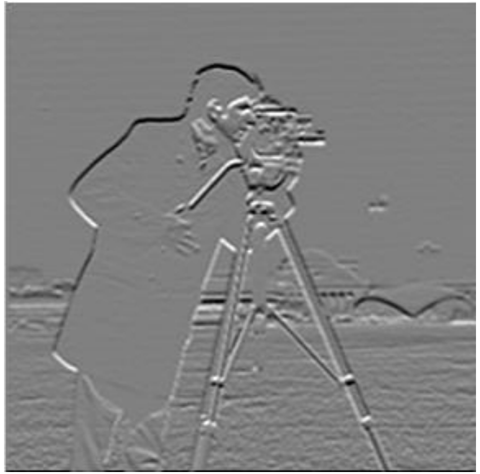


Cameraman



First derivative
(horizontal gradient)

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

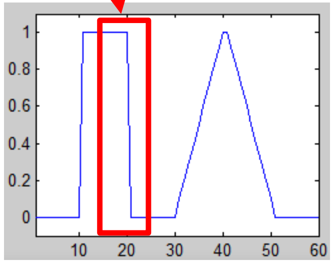


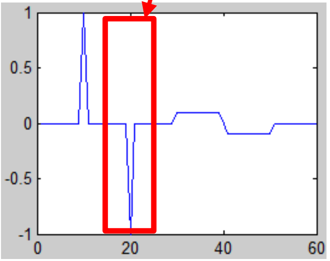
First derivative
(vertical gradient)

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & 1 \end{bmatrix}$$

First derivative
(diagonal gradient)

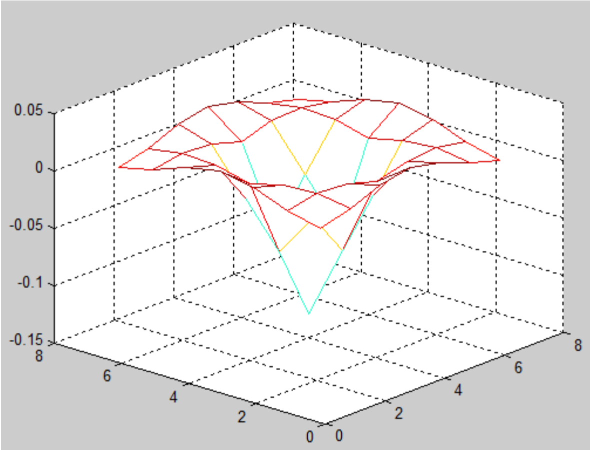
$$\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$





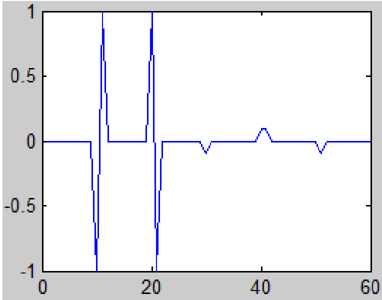
Usage of convolution III : 2D filtering

7x7 Laplacian of Gaussian kernel



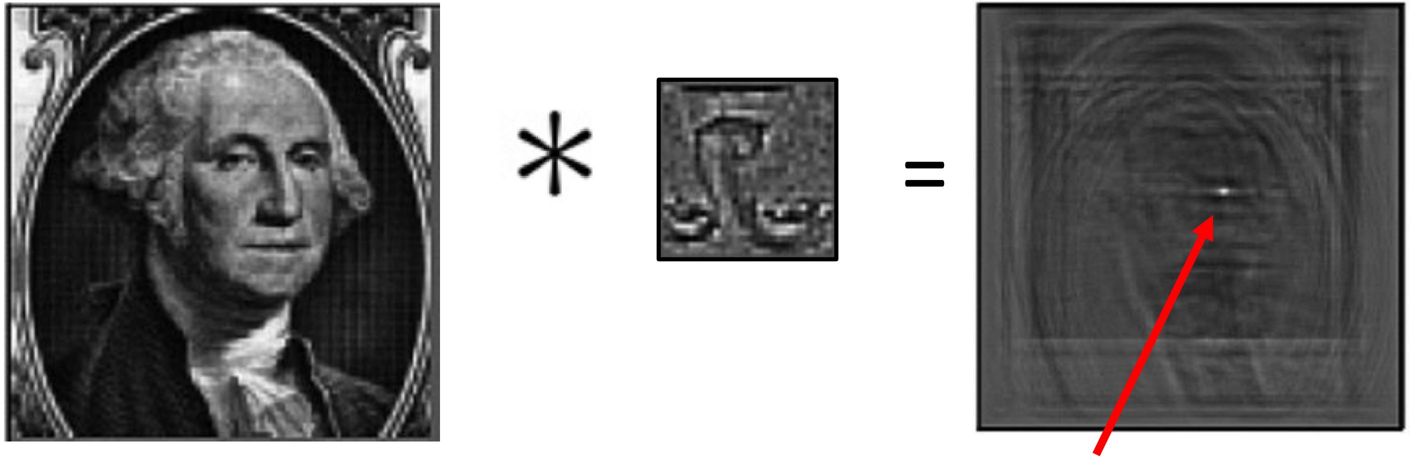
$$\begin{bmatrix} 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.3 & 0.4 & -0.7 & -1.3 & -0.7 & 0.4 & 0.3 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \end{bmatrix}$$

Second derivative of an image



Usage of convolution IV : 2D filtering

- Seeking for a known patter
- Large convolution kernel is applied
- Kernel size is equivalent with the size of the sought pattern



- Sensitive for rotation
- Scale variant

Filter responeded with a strong white peek in the matching position

Decomposition of large kernels I

- Convolution is associative

$$f * (g * h) = (f * g) * h$$

Example:

$$\begin{bmatrix} 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.3 & 0.4 & -0.7 & -1.3 & -0.7 & 0.4 & 0.3 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.5 & 0.2 \\ 0.5 & -3.1 & 0.5 \\ 0.2 & 0.5 & 0.2 \end{bmatrix} * \begin{bmatrix} 0 & 0.2 & 0.3 & 0.2 & 0 \\ 0.2 & 0.6 & 0.8 & 0.6 & 0.2 \\ 0.3 & 0.8 & 1.2 & 0.8 & 0.3 \\ 0.2 & 0.6 & 0.8 & 0.6 & 0.2 \\ 0 & 0.2 & 0.3 & 0.2 & 0 \end{bmatrix}$$

Laplacian of Gaussian kernel $(g * h)$ Laplacian (g) Gaussian kernel (h)

Number of operations: $49 * N_{pix}$ $9 * N_{pix} + 25 * N_{pix} = 34 * N_{pix}$

15% reduction of computational demand!!!

Decomposition of large kernels II

- Decomposition is not exact in most cases
  - In general case, it approximates the kernels with a limited accuracy only
- Neural nets does not sensitive for inaccurate decomposition
- Decomposition of larger kernels leads to higher savings!
- Wildly used!

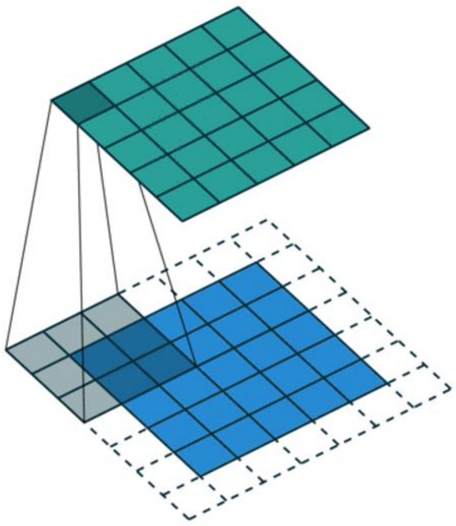
Stride

- **Stride** is the number of pixel what we slide the kernel
  - Horizontal stride
  - Vertical stride
- Down sampling the image
  - Size:

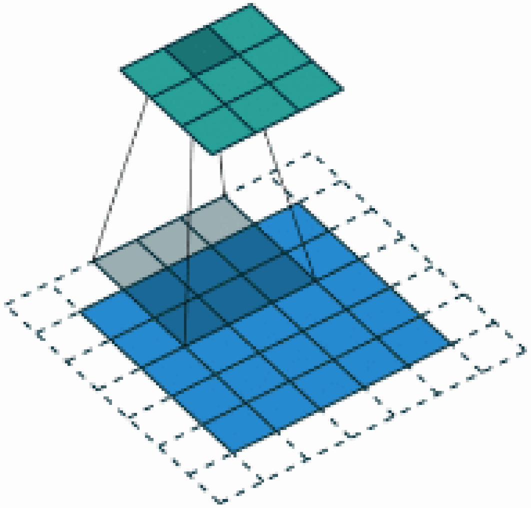
$$\frac{n+2p-k}{s}+1$$

  - where:

size(f)= n, size(g)= k,
p: padding, s: stride



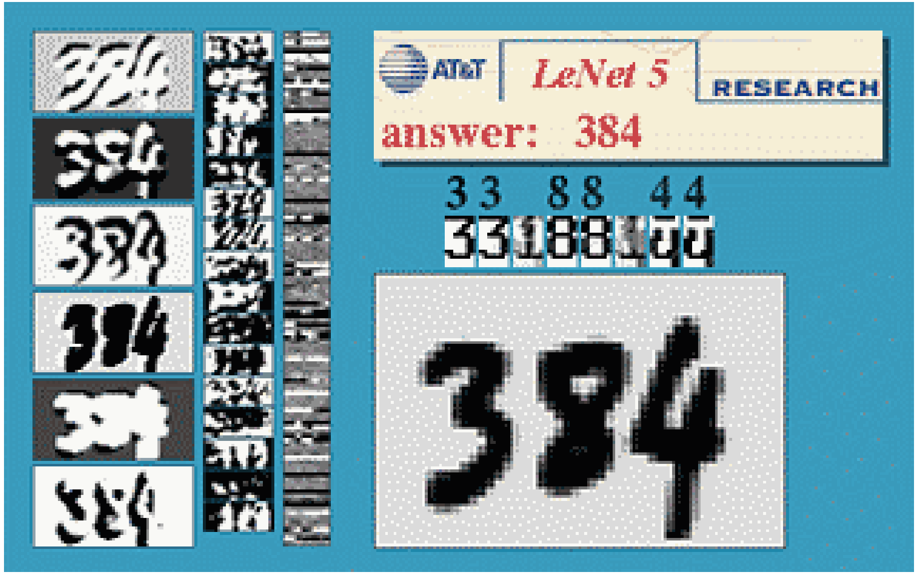
Padding:1, stride: 1



Padding:1, stride: 2

What is the role of a convolution?

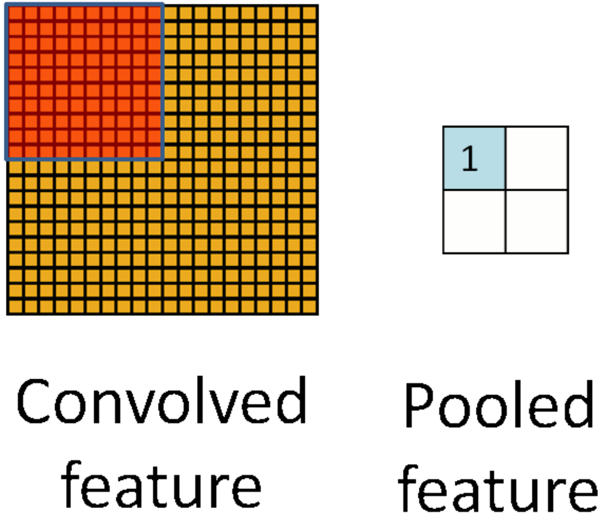
- The convolution emulates the response of an individual neuron
- Each convolutional neuron processes data only for its receptive field
  - Receptiv field: area covered by the g function
- Why not fully connected?
  - Reduces the number of the parameters (millions to a few dozens)
  - Avoids vanishing gradient problem, because one weight is tuned by a large number of data pathes
- Since the convolution is space invariant, detection will be space invariant also



Space invariance means here that the functionality of a 2D function is not changing in space. This enables the detection of a certain image feature anywhere on the image.

Pooling

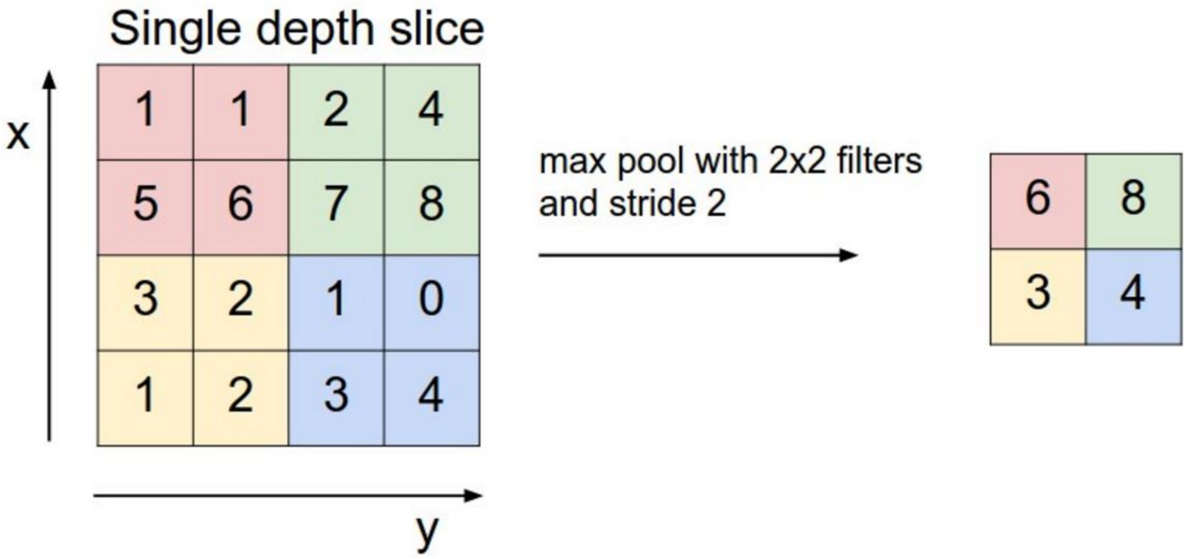
- Pooling summarizes statistically the extracted features from the same location on a feature map
- Mathematically, it is a local function over 1D or 2D data
  - input:
    - Segment of a vector in 1D
    - rectangular neighborhood in 2D
  - Function
    - Statistical (maximum: max-pool)
    - L2 norm
    - Weighted average (weights proportional of the distance of the central element)
- In most cases: stride > 1
  - This leads to downsampling
- Pooling introduces some shift invariancy



$s = 10$

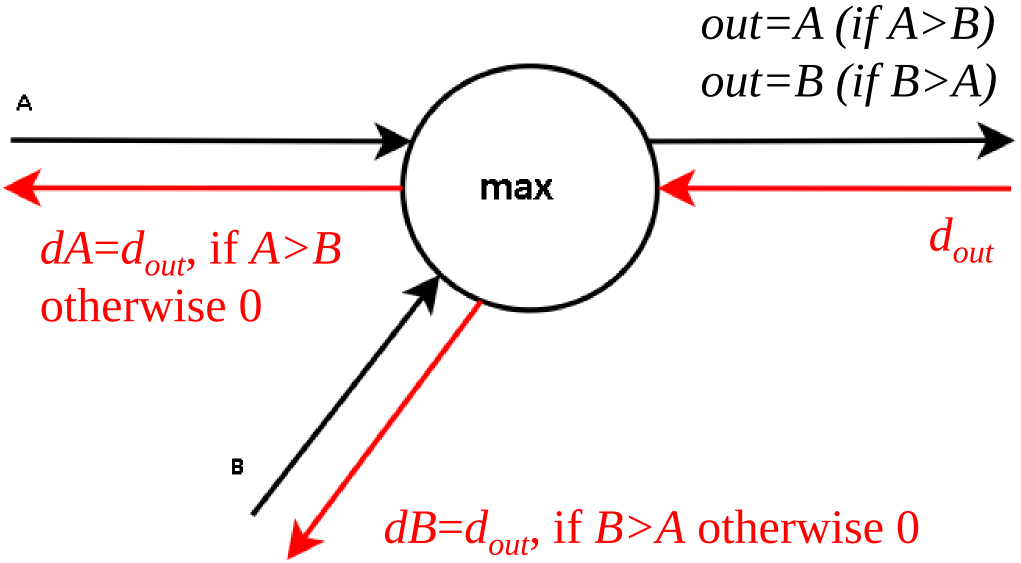
Max pooling

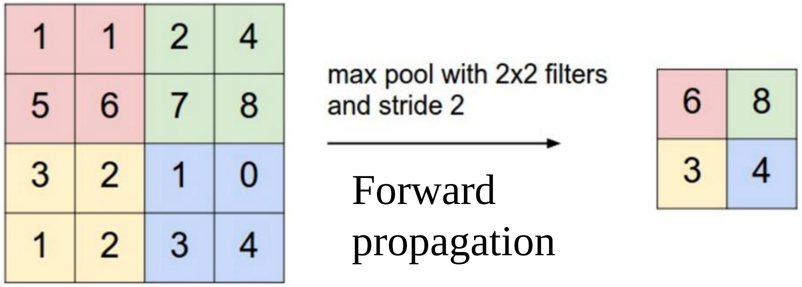
- Max pooling is the most used pooling in CNN
- Picks the largest value from a neighborhood
- Non-linear
- Statistical filter
- Downsampling depends on the stride

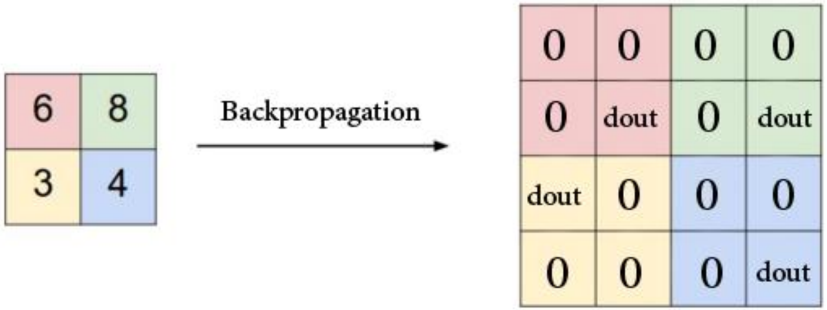


Backpropagation through max-pooling layer

- Maximum node acts as a router
- The d_{out} gradient is given to the input node, which has contributed (which was the biggest)
- The remaining positions will get zero, because they did not contributed to the error



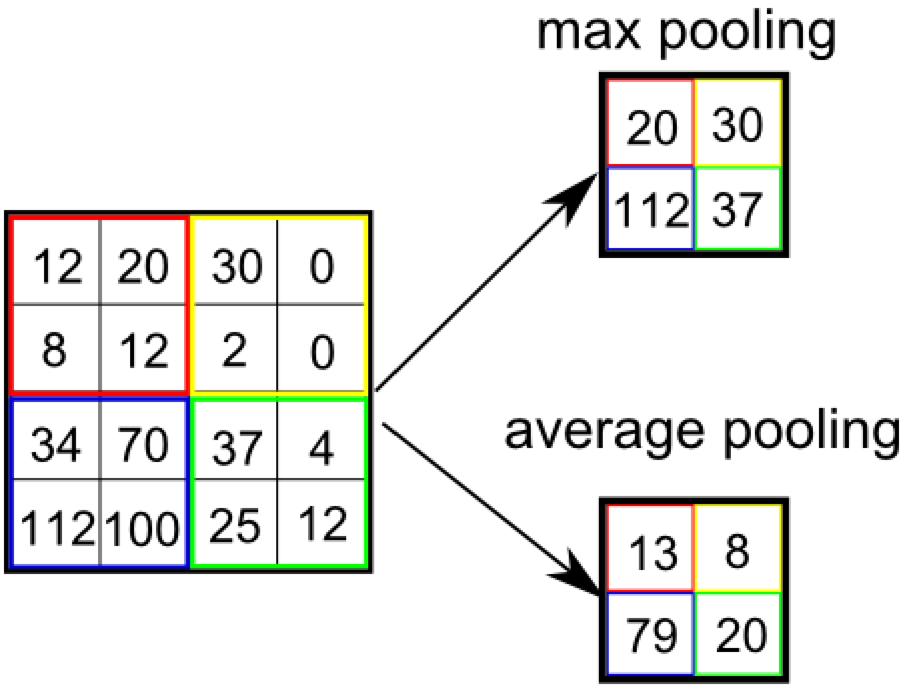


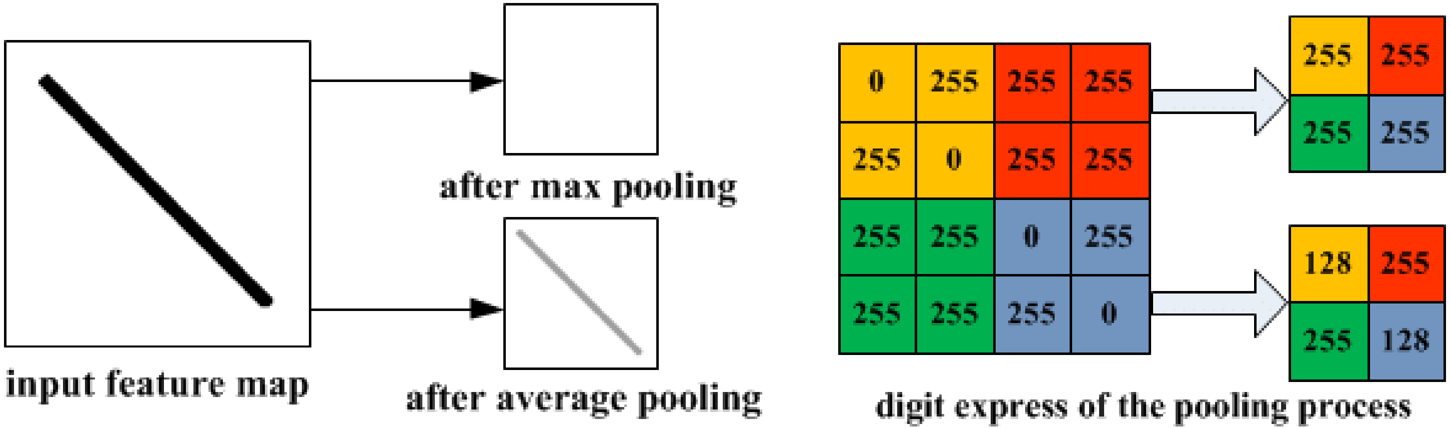


The maximum positions are stored

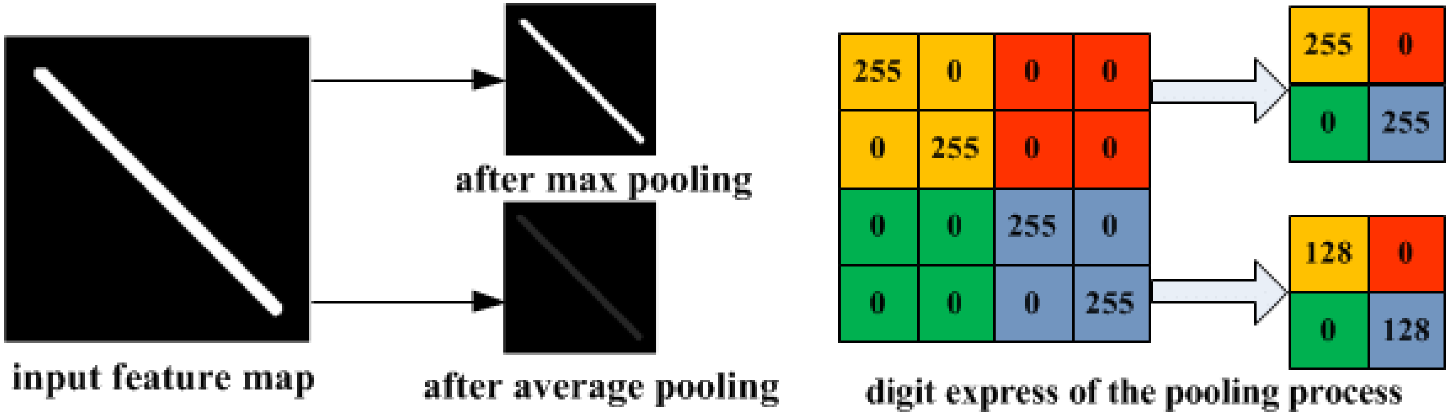
Average pooling

- Similar to max pooling, but uses the average



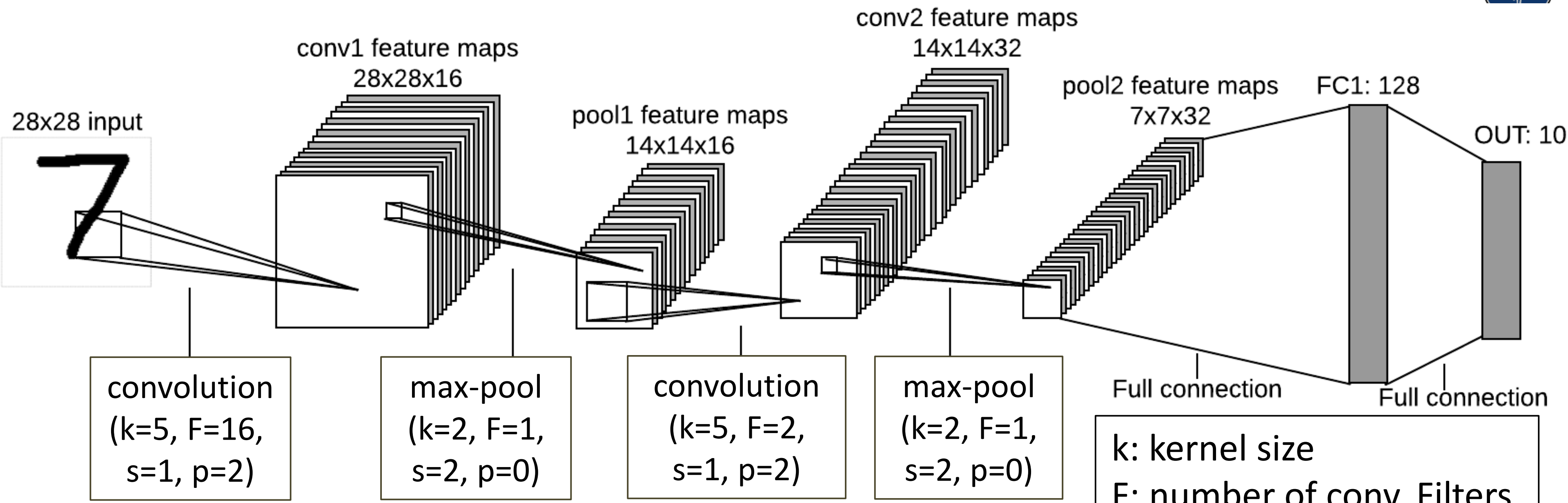


(a) Illustration of max pooling drawback



(b) Illustration of average pooling drawback

Architecture of a typical Convolution Neural Network



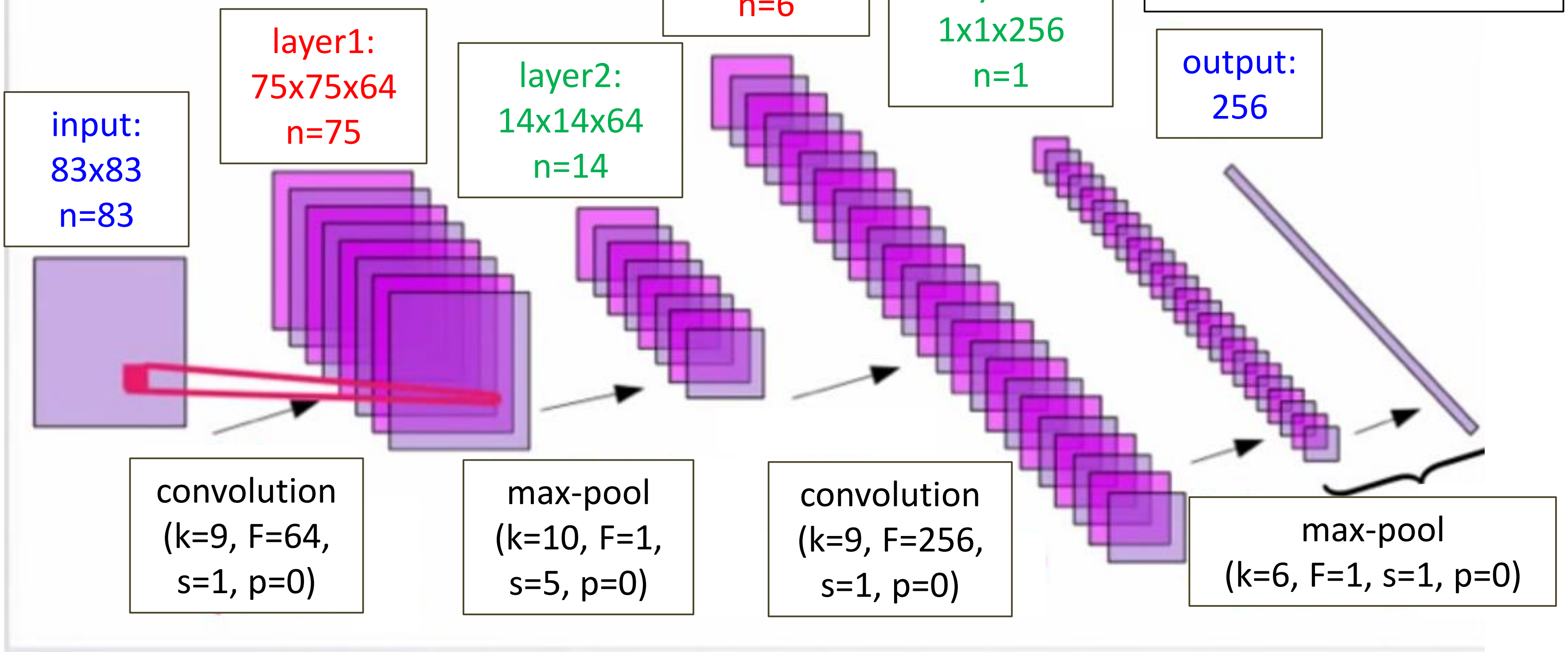
k: kernel size
F: number of conv. Filters
s: stride
p: zero padding size

- Input
- Parallel feature extractors (convolution layers w. RELU)
- Data reduction (pooling)
- Combination of the features – aggregating information (fully connected layer)
- Decision (fully connected layer with soft-max activation)

CNN example for data size reduction

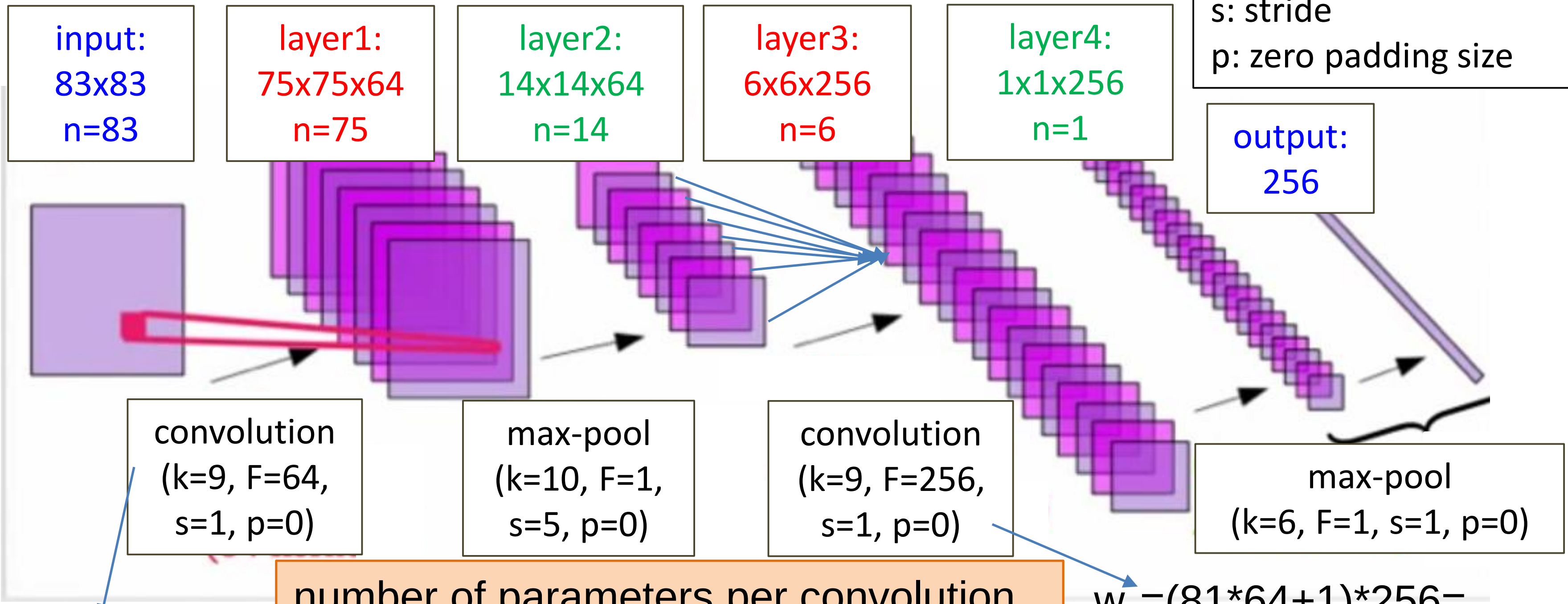
k: kernel size
F: number of conv. filters
s: stride
p: zero padding size

$$size = \frac{n + 2p - k}{s} + 1$$



Number of free parameters

k: kernel size
F: number of conv. filters
s: stride
p: zero padding size



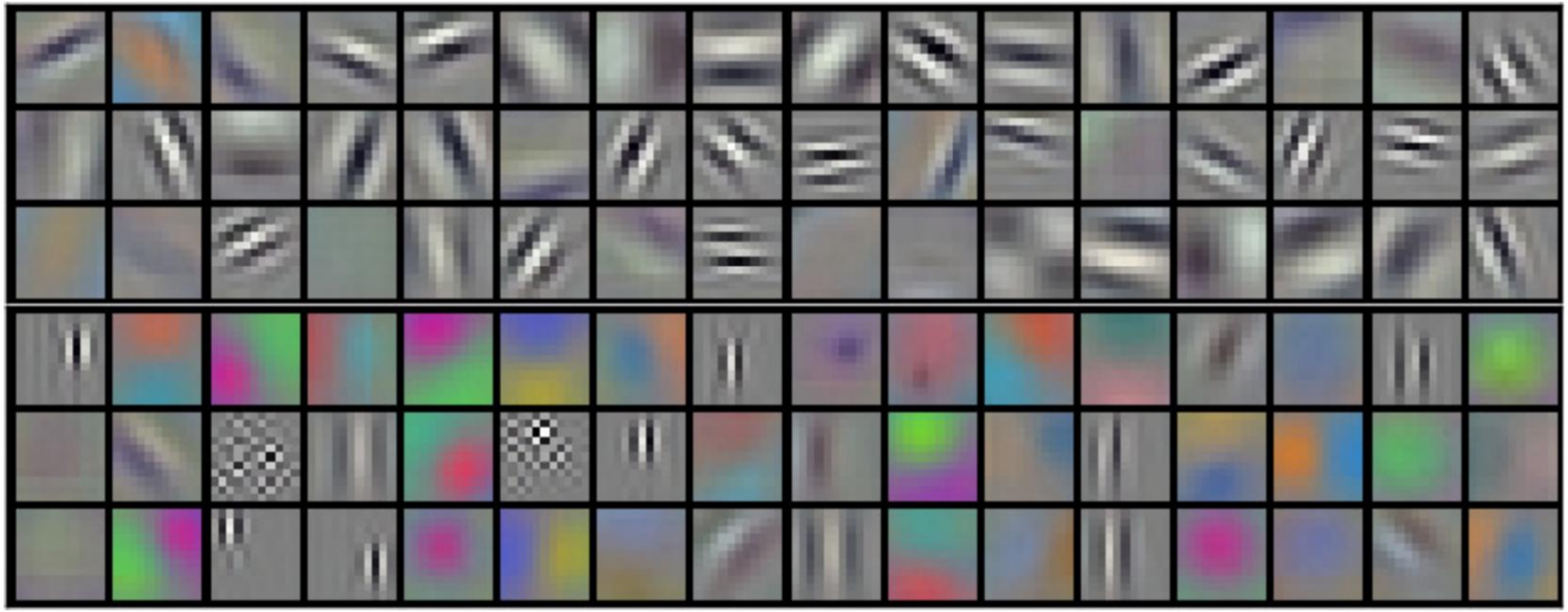
$w_{\#}=(81*1+1)*64=$
$=5{,}248$

number of parameters per convolution layer: $w_{\#}=(k^2\times n_i+1)\times n_o$
where:
n_i , n_o :number of input /output layers
+1 stands for the bias

$w_{\#}=(81*64+1)*256=$
$=1{,}327{,}360$

Each feature map receives input from each one from the previous layer

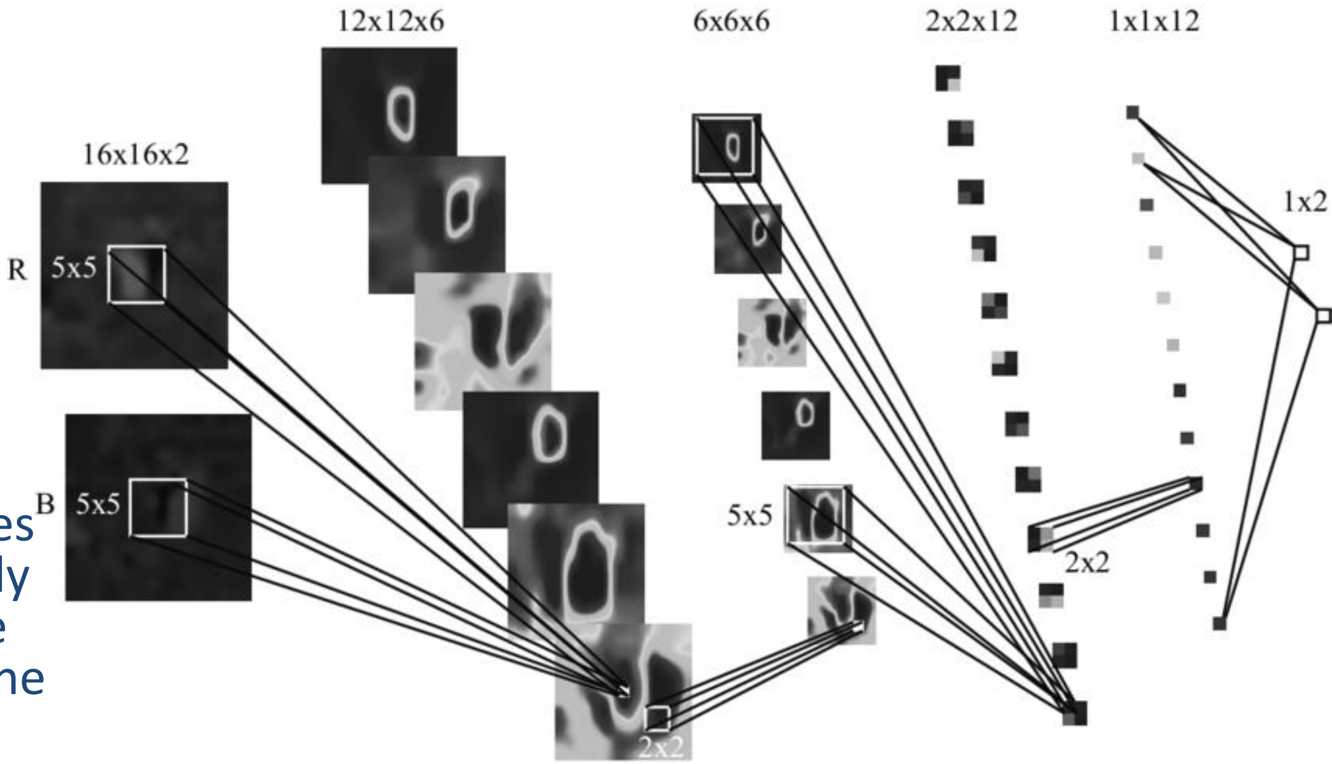
Typical features for the first layers



Individual feature maps gives high response to these patterns

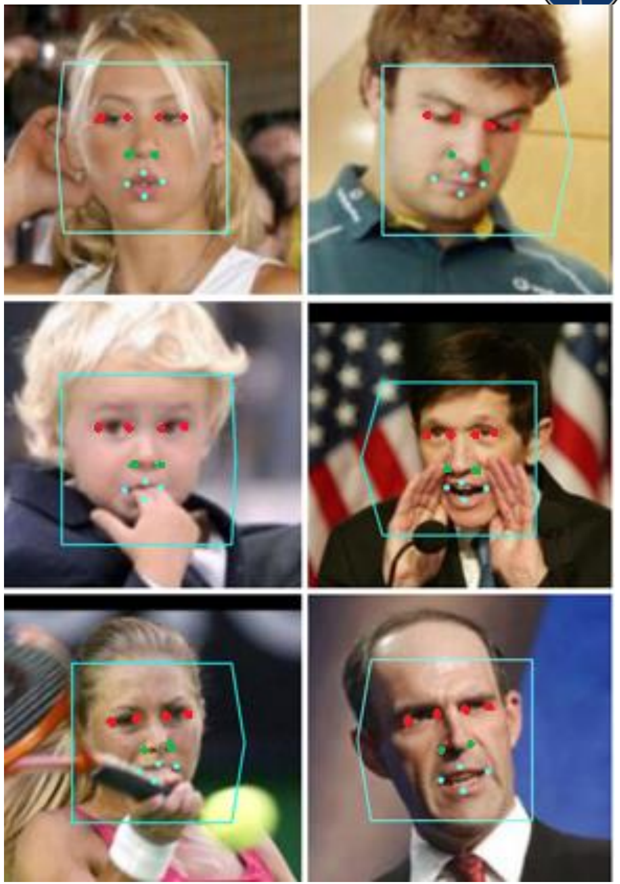
Combination of features

- The output of multiple feature maps can be combined to a feature map in the next layer with convolutions
- If 1x1 convolution kernel is applied, this enables weighted sum of multiple maps
- Ultimately, the features are combined by a fully connected layer in the classification part of the network



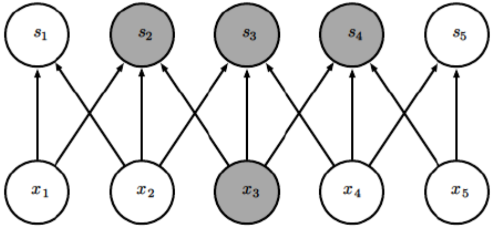
Why data size reduction is important?

- Methods of data size reduction
  - Pooling
  - Convolution with strides
  - Convolution without padding
- Information aggregation
- Reduces the chance of overfitting or vanishing gradient
- The distant local features are brought closer
  - One filter can cover multiple features from the previous layers

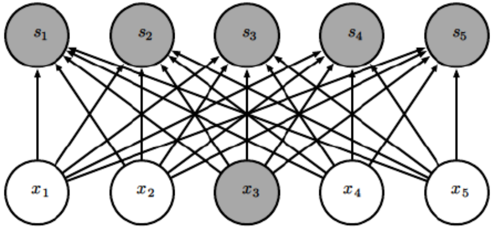


Properties of Convolutional Neural Networks I:

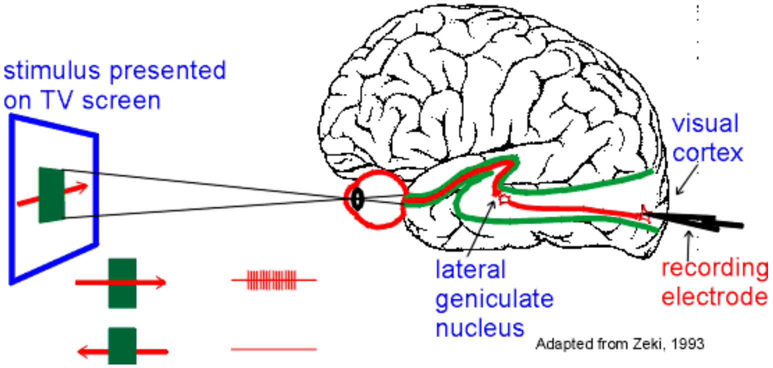
- Sparsity
  - The interconnection weights are just a fraction of the fully connected NN (the weight matrix between two layers are sparse)
  - A few dozen free parameter describes the operation of a layer
  - Receptive field organization similar to natural neural vision systems



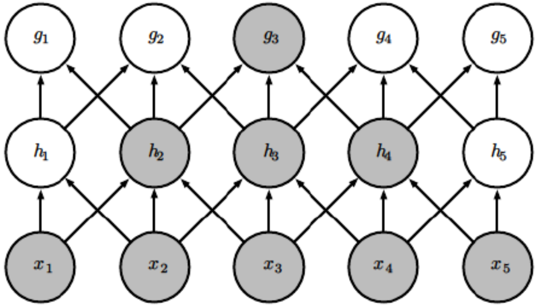
Sparse interconnections



Dense interconnections



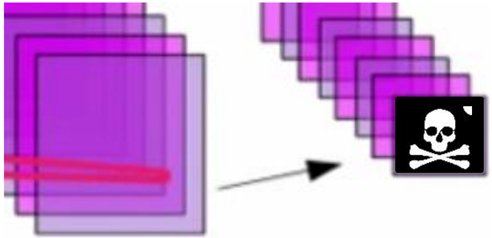
A neuron in visual cortex receives input from the receptive field only, which is a small piece of the visual field



Receptive field of an artifitial neuron

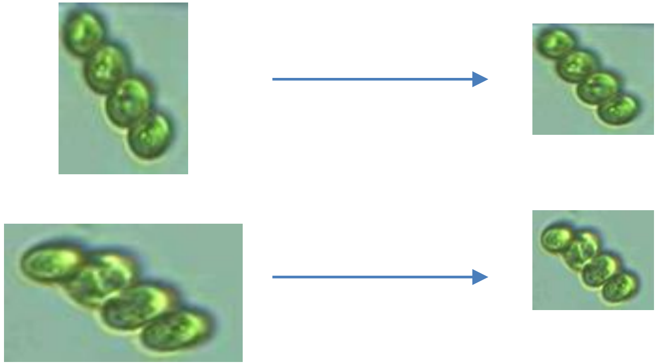
Properties of Convolutional Neural Networks II:

- Parameter sharing
  - Same parameters everywhere in the layer
  - Contribution to the gradient of a weight from many positions
  - Reduces the risk of overfitting
  - Reduces the risk of dying RELU (dying cell)
    - When it happens, an entire feature extractor on a layer is dying



Properties of Convolutional Neural Networks III:

- Variable input size
  - The input image is either resized or padded



Input images are
resized to the same size

Properties of Convolutional Neural Networks IV:

- Equivalent representation
  - Equvariance to translation
    - The output shifts with an input shift
  - In a fully connected neural network, each input is a dedicated channel for a certain input parameter-therefore the inputs cannot be swapped
    - Like bank example, one cannot replace the age input with the salary input
  - In CNN, the image can be shifted, because the inputs are not dedicated and the features are identified anywhere

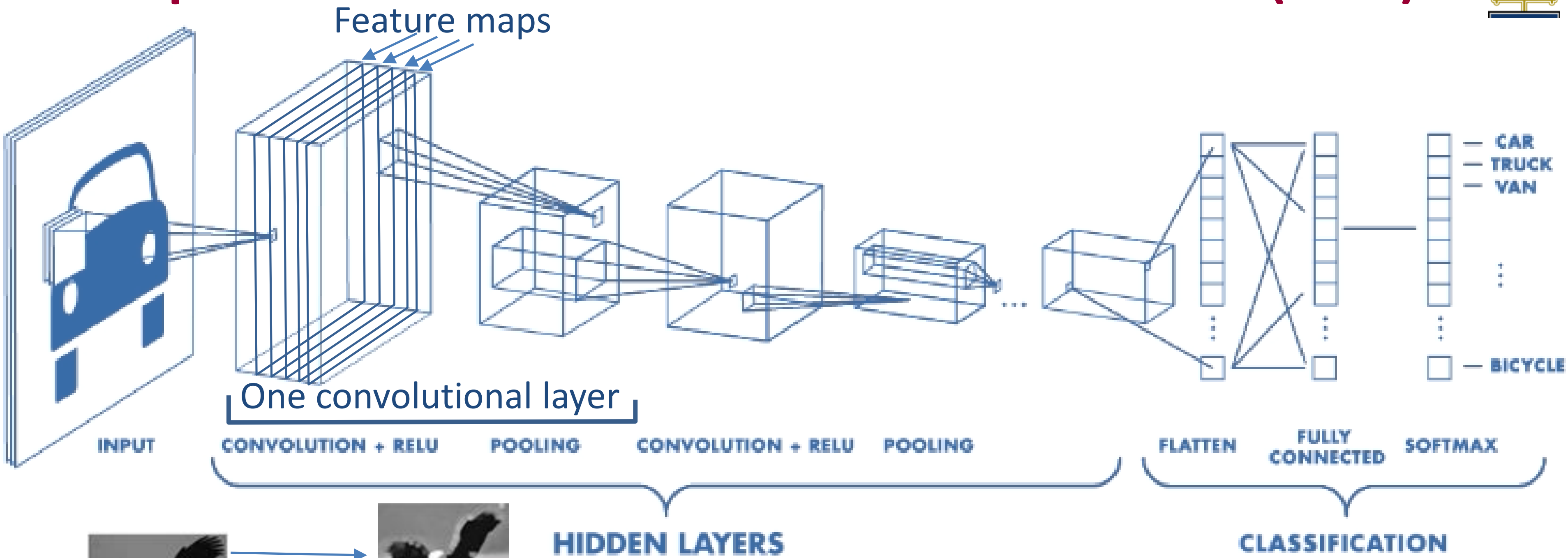
Neural Networks

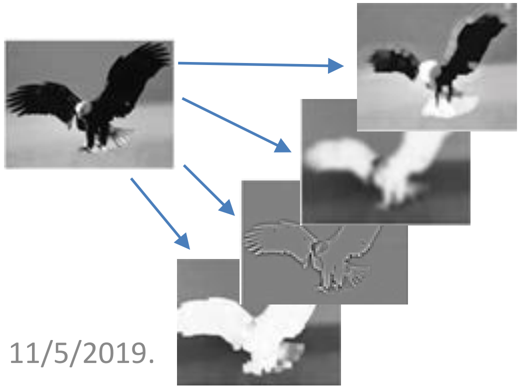
Convolutional Neural Networks

(P-ITEEA-0011)

Akos Zarandy
Lecture 7
November 5, 2019

Components of a convolutional neural network (CNN)





Contents

- Regularization and normalization methods
  - Local response normalization
  - Data augmentation
  - Early stopping
  - Ensembling
- Example CNN: AlexNet
- Segmentation

Regularization and optimization methods

- Different methods to increase the loss in the learning phase, but reduce overfitting and increase generalization capabilities
  - Local response normalization
  - Batch normalization
  - Data augmentation (Enriching the data set)
  - Early stopping
  - Ensemble methods
    - Network duplication
    - Bagging
    - Dropout

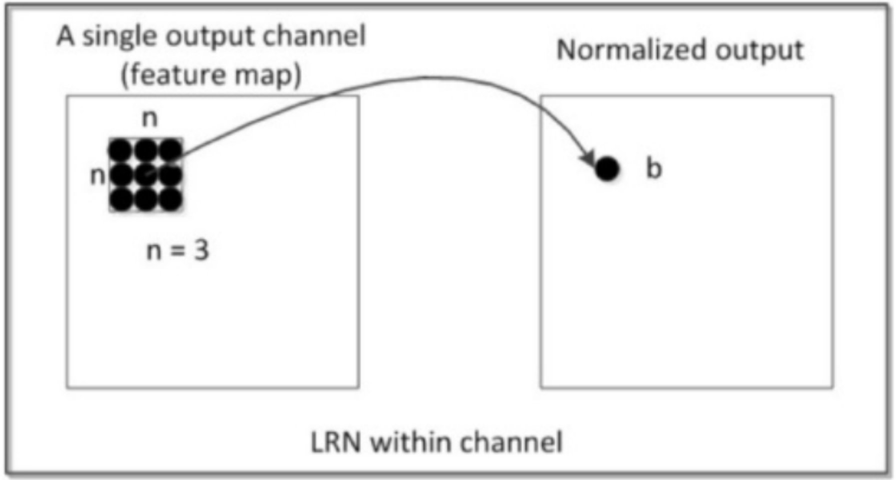
Ian Goodfellow: regularization is *"any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error"*

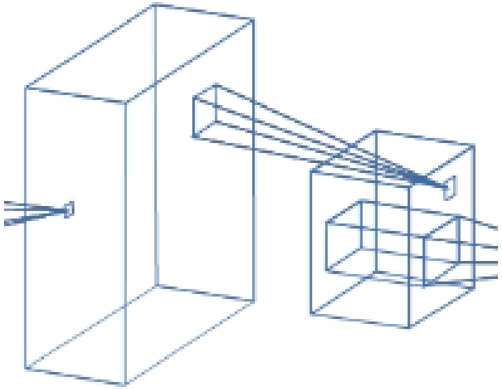
Local response normalization I

- Implementation of the Lateral inhibition from neurobiology
  - If a neuron starts spiking strongly in a layer it inhibits (suppresses) the of the neighboring cells
  - Winner take all (have a strong decision)
  - Balances the asymmetric responses of neurons in different areas of the layer
- Useful when we are dealing with ReLU neurons
  - Normalizes the unbounded activation of the ReLU neurons
    - Avoids concentrating and delivering large values through layers
  - It enhances high spatial frequencies by suppressing the local neighbors of the strongest neuron

https://towardsdatascience.com/difference-between-local-response-normalization-and-batch-normalization-272308c034ac

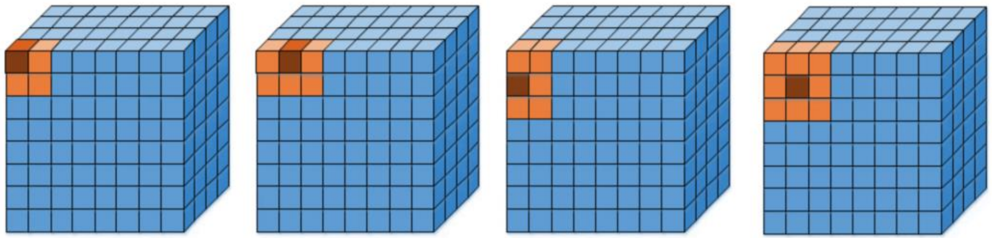
Local response normalization II



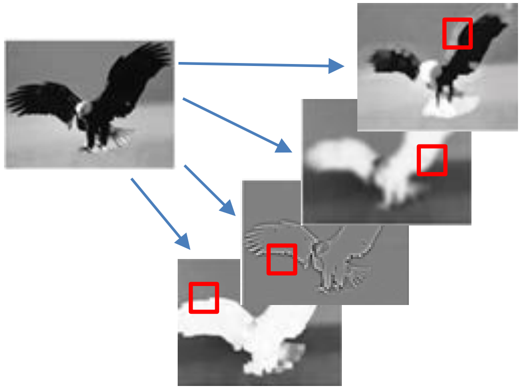


Looking from this side

Feature maps are facing to us

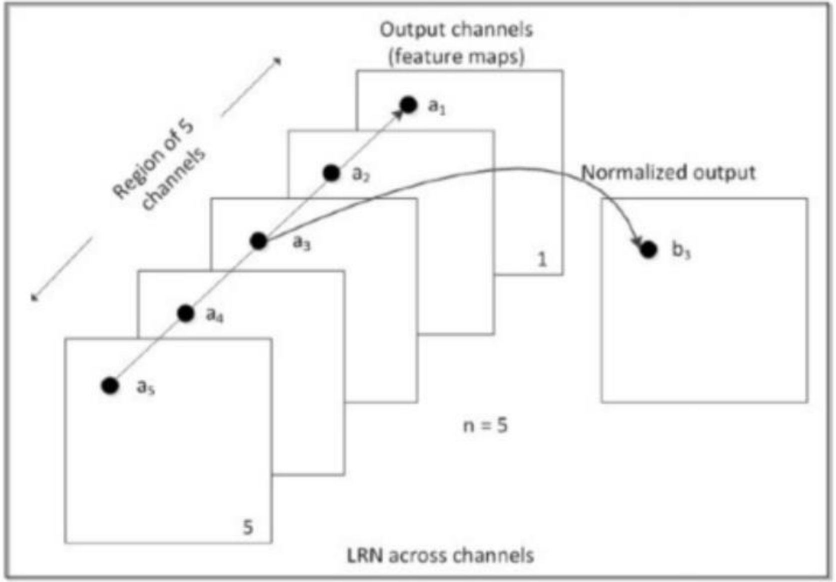


- Intra map normalization
  - 2D normalization within the same feature map
  - Balancing for different areas
  - Winner-take-all for neigbouring neurons in the same feature map (strongest response to the same transformation should win)

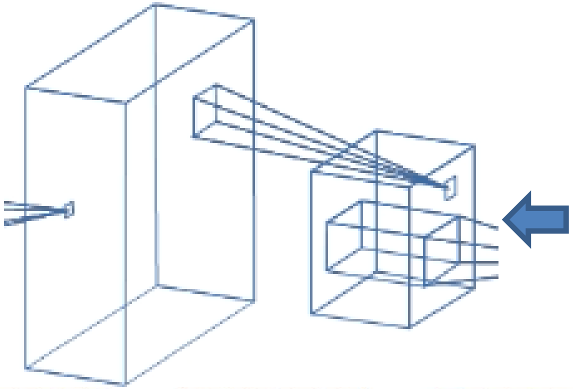


Which particular pattern responds the largest for the same transformation?

Local response normalization III

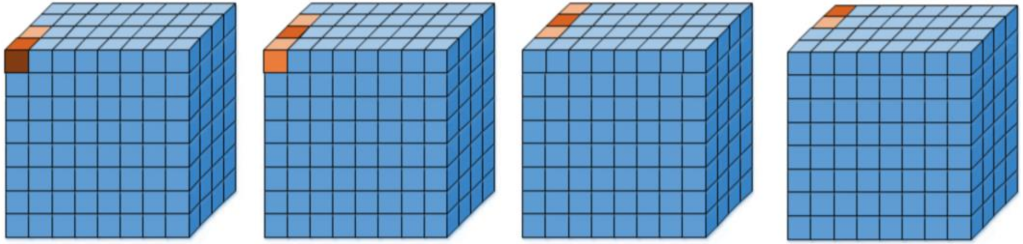


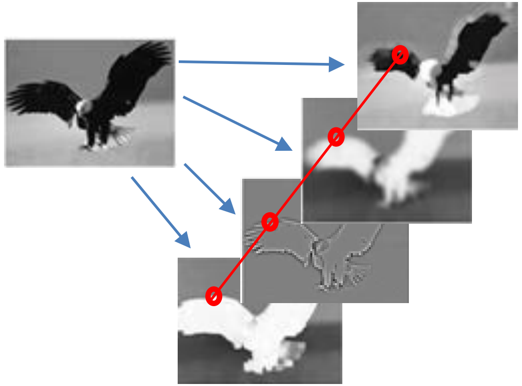
– Inter-map normalization
  - Normalization between the neighboring feature maps
  - Winner-take-all for the largest response with different transformation for the same input location



Looking from this side

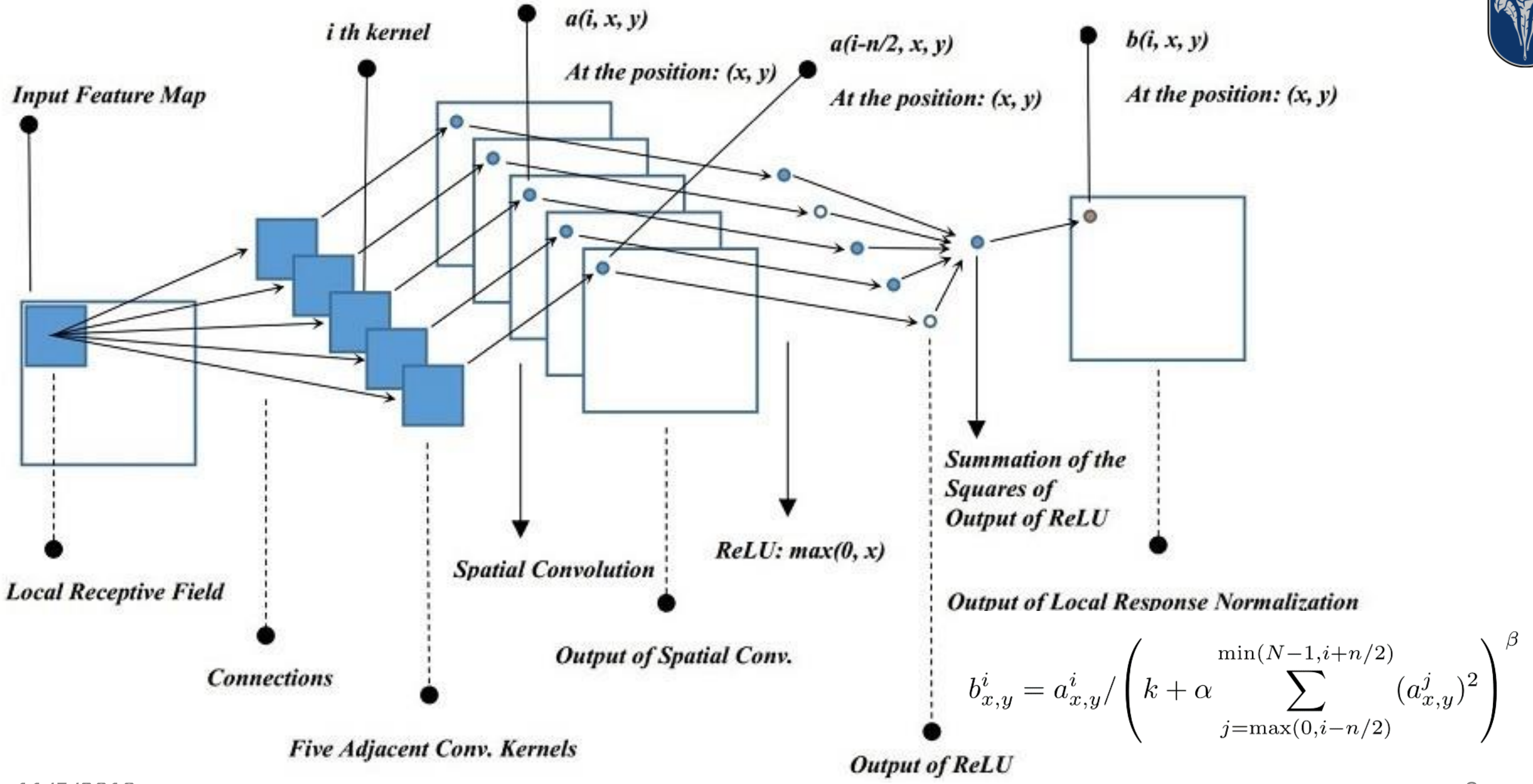
Feature maps are facing to us





Which particular tranformation responds the largest for the same pattern?

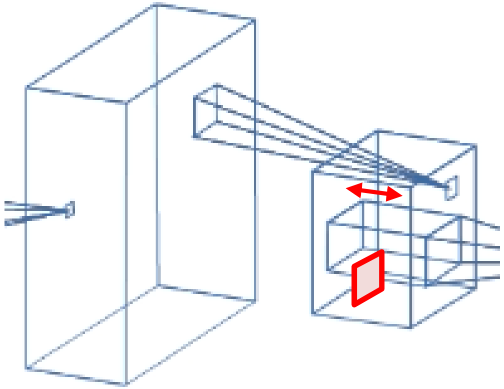
Calculation method of local response normalization



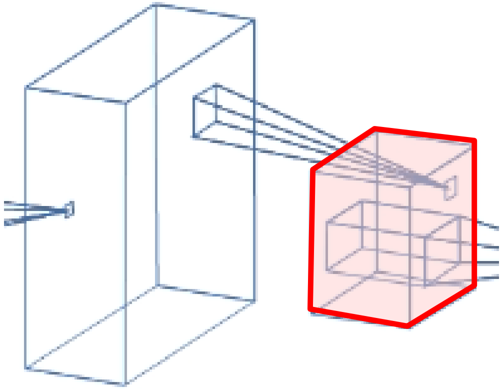
Local response norm. vs batch norm.

Both work within one convolutional layer

- Normalization either through the feature maps or within one feature map
- Normalization is done for one input image

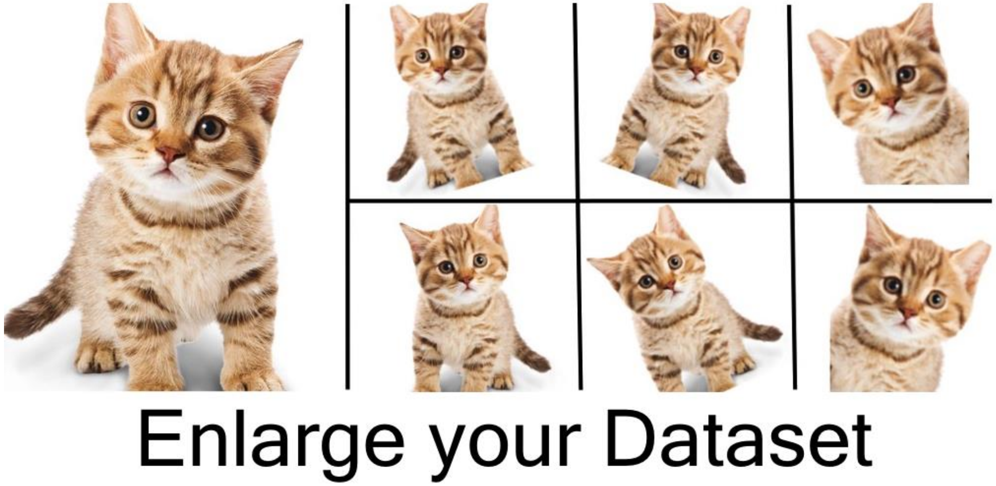


- Normalization done for all the pixels in all the feature maps within a layer
- Normalization is done for the entire batch



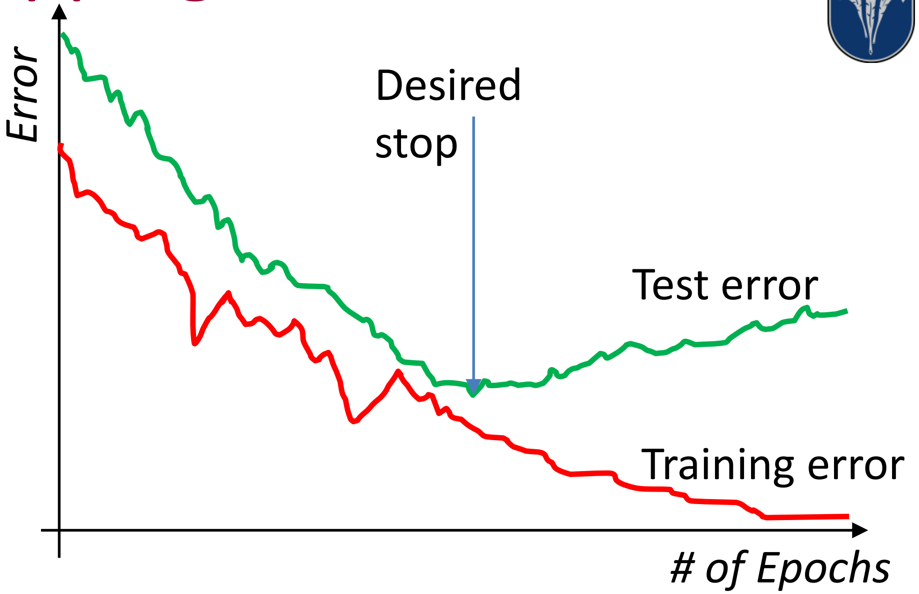
Data augmentation

- **Idea:**
  - *Increase the generalization capability of the net by enlarging the training set*
- Increase the number of the training vector by introducing fake (artificial) input-output pairs
- Typical methods
  - Translating
  - Slight rotation
  - Rescaling
  - Adding noise
  - Flipping
  - Cutting out parts
  - Manipulating with pixel values



Early stopping

- **Idea:**
  - Split data into training and test sets
  - At the end of each epoch (or, every N epochs):
    - evaluate the network performance on the test set
    - if the network outperforms the previous best model: save a copy of the network parameters at the current epoch



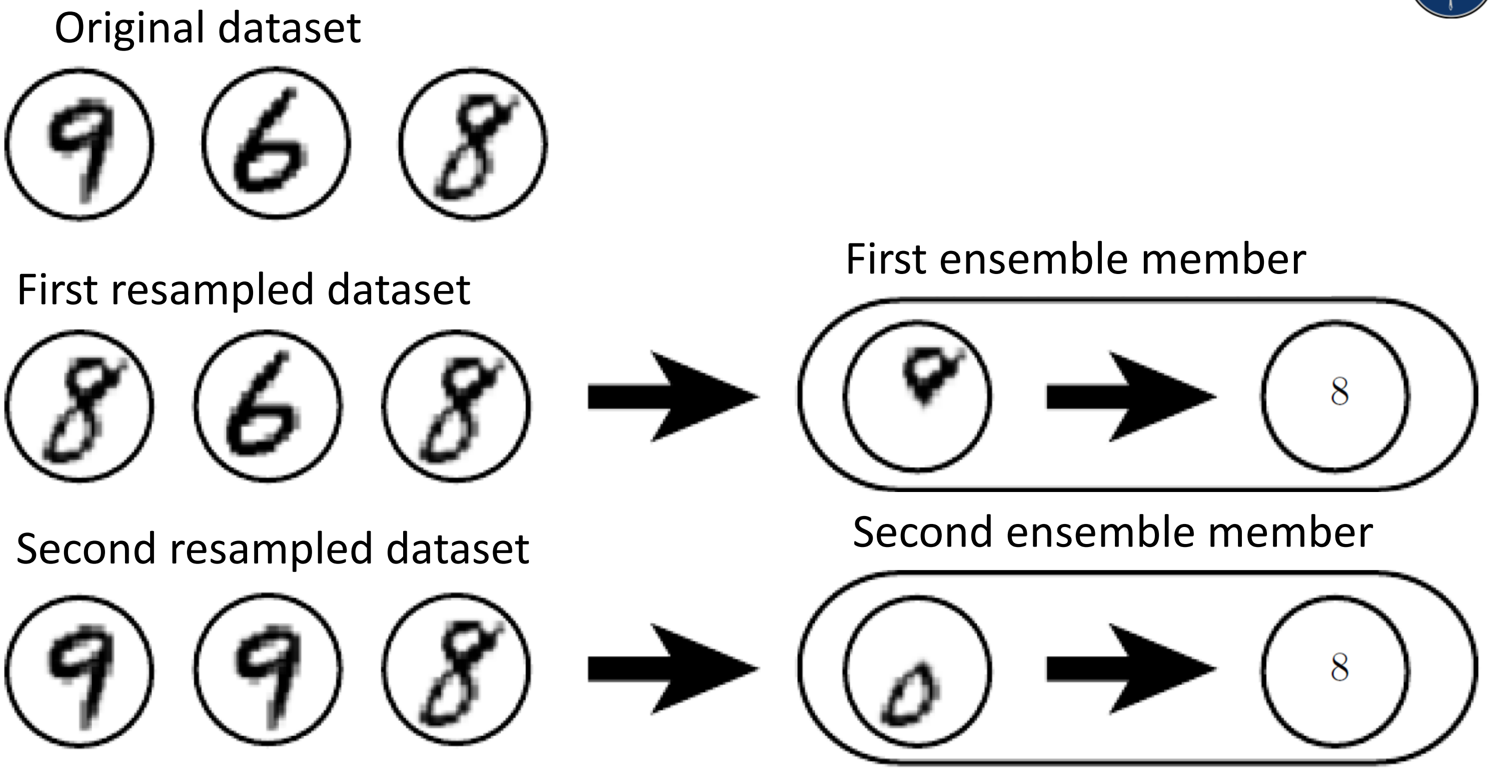
  - The best suboptimum is selected finally
  - Since the error function is not necessarily monotonic, the optimization goes on, but the suboptima are saved

Ensemble methods

- Idea of ensemble methods:
  - Generate multiple copies of your net
    - Same or slightly modified architectures
  - Train them separately
    - Using different subsets of the training sets
    - Different objective functions
    - Different optimization methods
  - The different trained models have independent error characteristics
  - Averaging the results will lead to smaller error
- Requires more computation and memory both in training and inferencing (testing) phase

Bagging

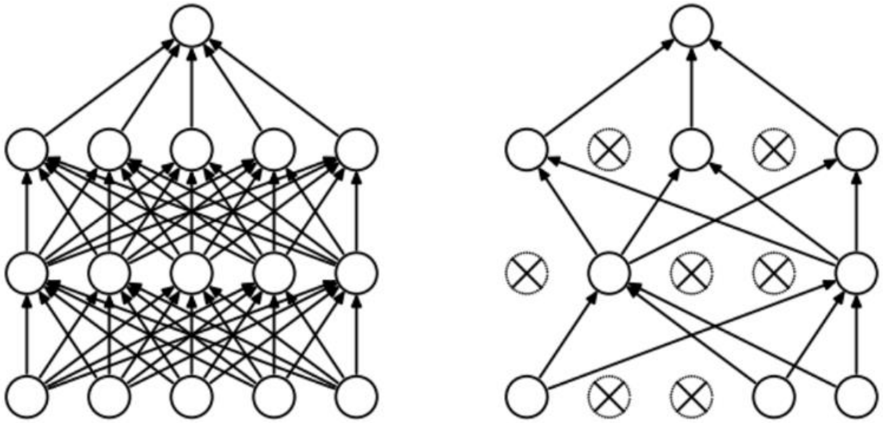
- Construct k different datasets
- Each with a subset of the data, but with duplications
- Trains with these
- Make result averaging



First learns the upper loop, the second the lower. When both say yes, it is an 8.

Dropout

- Idea of dropout method:
  - Use mini-batch training approach
  - For each minibatch, a random set of neurons from one or multiple hidden layer(s) (called **droppout layers**) is temporally deactivated
  - Deactivation probability is p
  - In testing phase, use all the neurons, but multiply all the outputs with p, to account for the missing activation during training
- Requires more training steps, but each is simpler, due to reduced number of neurons
- No computational penalty in testing phase
- Use it for fully connected layers



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

Reduces overfitting, because the network is forced to learn the functionality in different configurations using different neural paths.

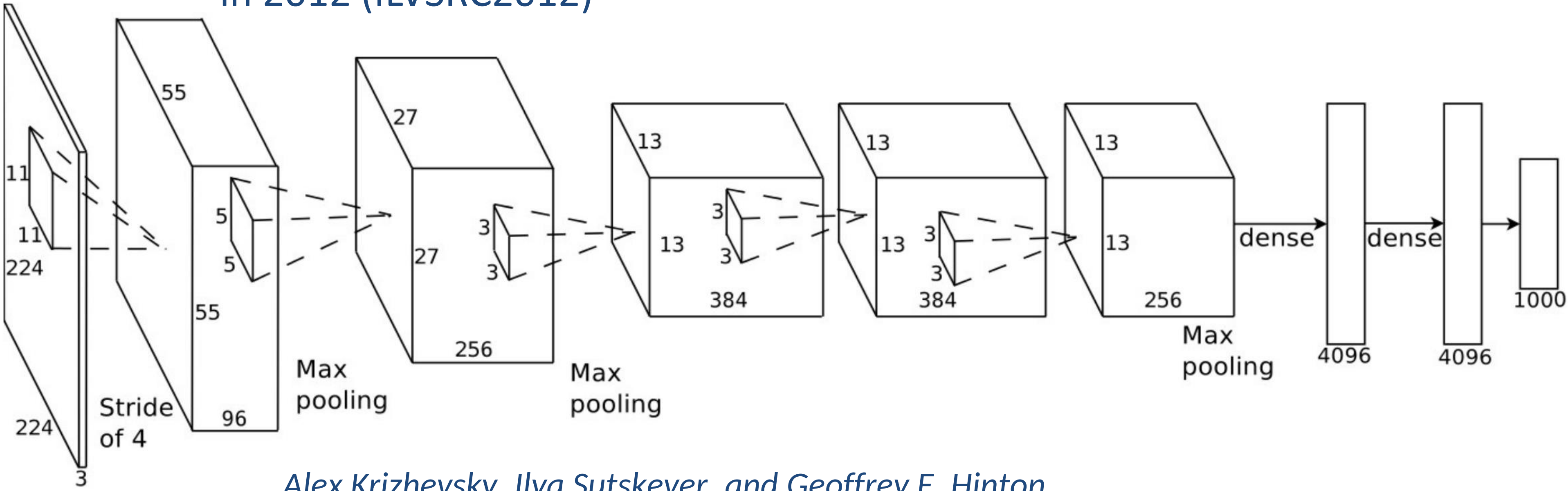
Summary of CNN

- Layers:
  - Convolution, fully connected
- Activation function
  - ReLU, SoftMax
- Data aggregation
  - Stride convolution, pooling
- Regularization
  - Test set, data, parameter, and architecture regularization

See, how it works in practice!

Alexnet

- First fully trained deep convolutional neural network
  - Won the ImageNet Large Scale Visual Recognition (ILVSRC) Challenge in 2012 (ILVSRC2012)



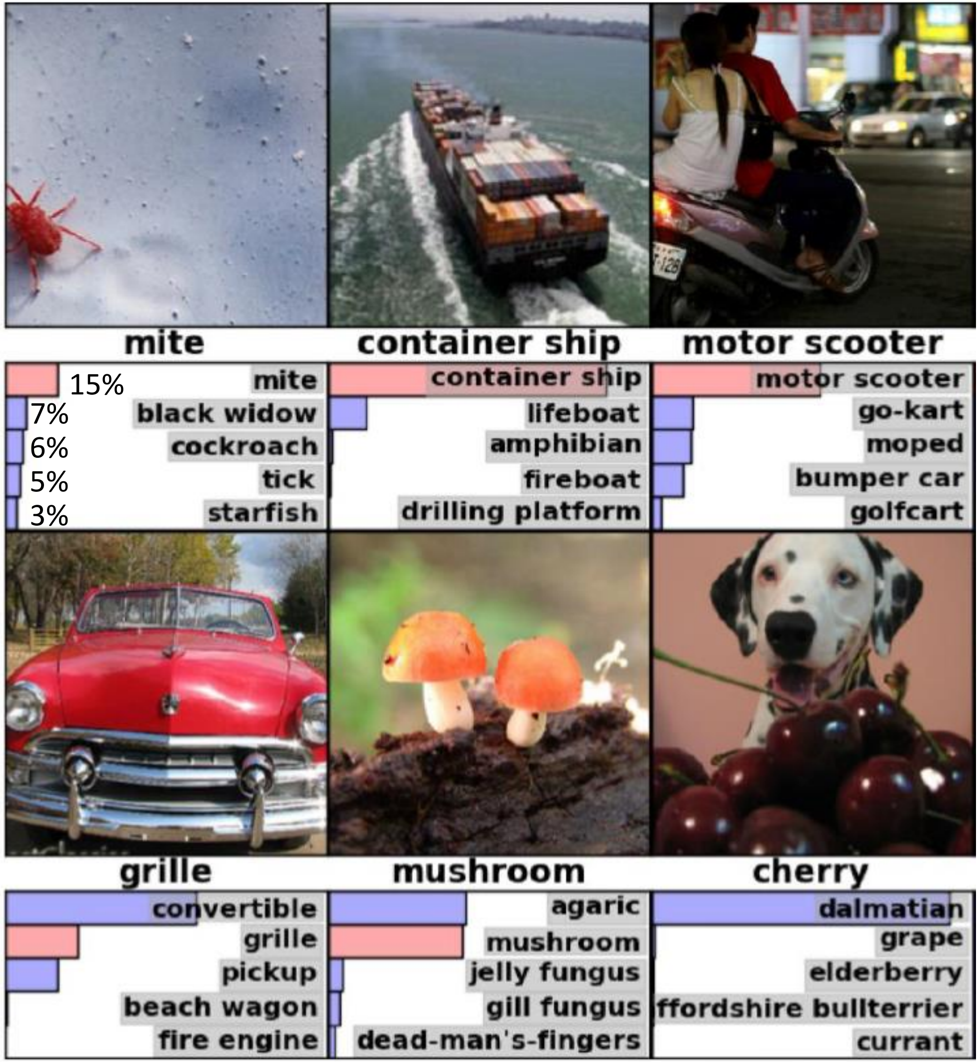
Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, "Imagenet classification with deep convolutional neural networks", Advances in neural information processing systems, 2012

ImageNet Large Scale Visual Recognition Challenge I

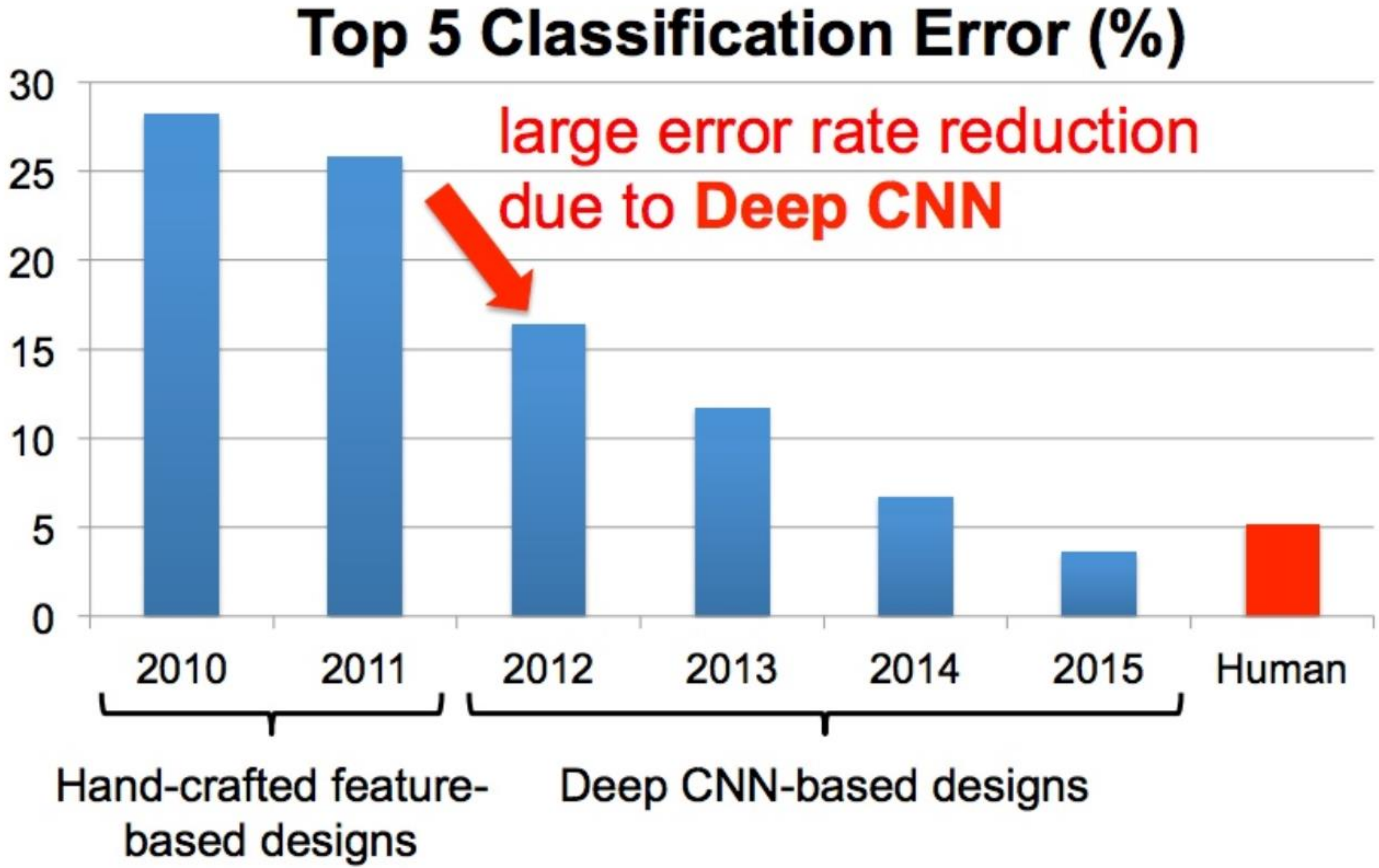
- ImageNet:
  - 15+ million labeled high-resolution images
  - 22000 categories
- ILSVRC uses a subset of ImageNet:
  - 1000 categories
  - ~1000 images per category
  - 1.2 million training images | 50 000 validation images | 150 000 testing images

ImageNet Large Scale Visual Recognition Challenge II

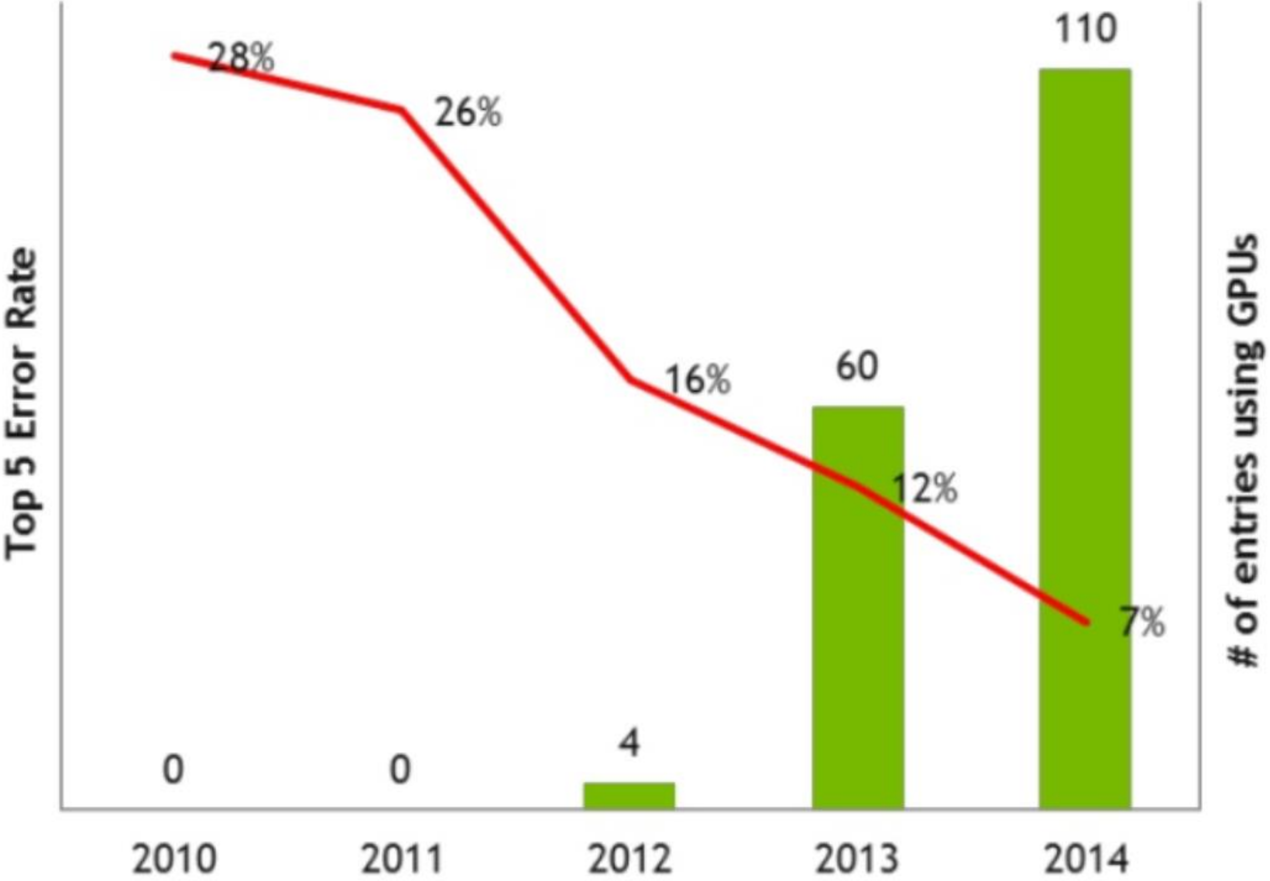
- Each image should be classified
  - Probability distribution
- Top 1 error rate:
  - What percentage was wrongly classified as highest probability? (38,9%)
- Top 5 error rate:
  - What percentage was not in the first five? (18.9%)



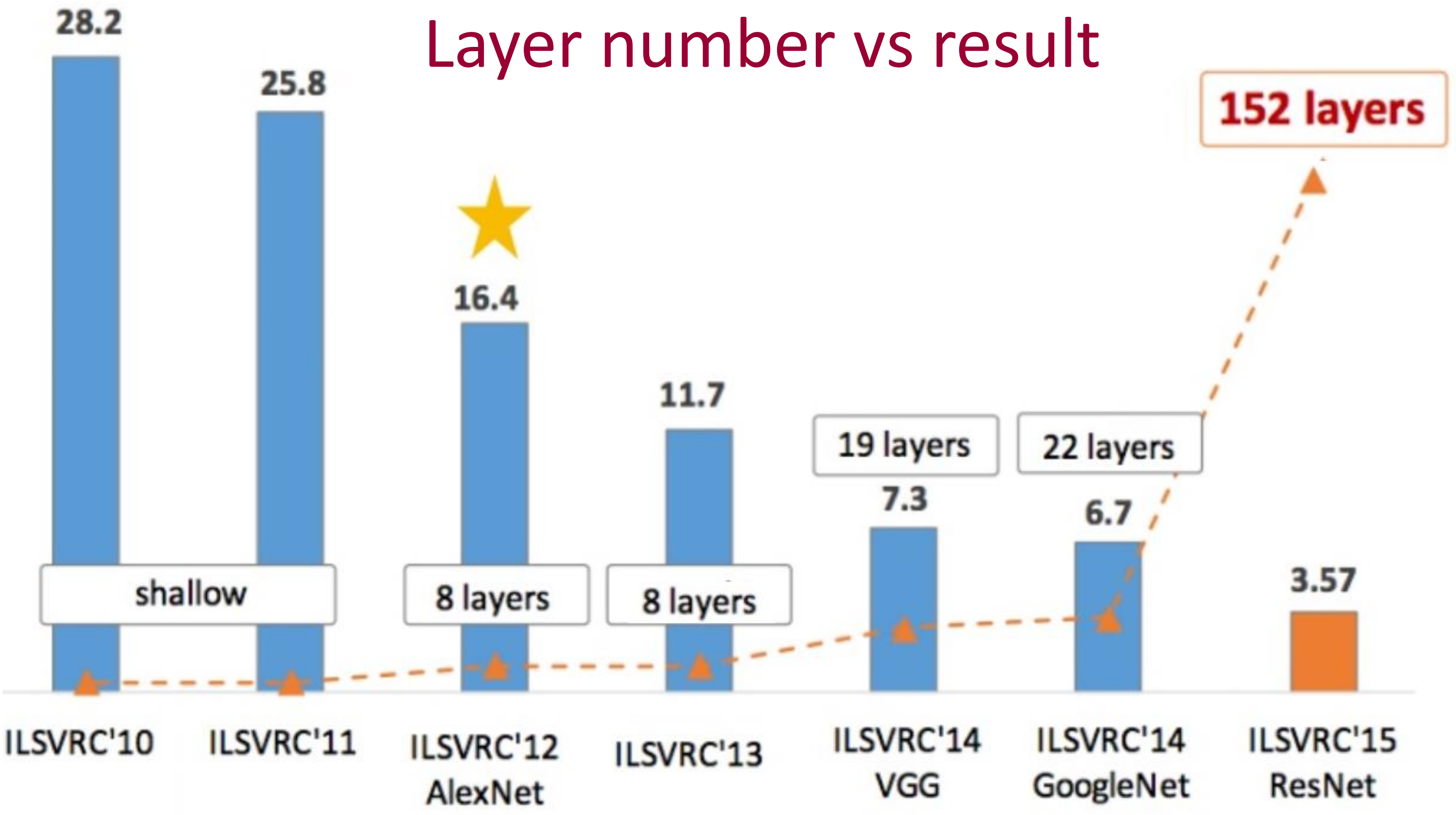
ILVSRC results



Teams used GPU in the challenge



Layer number vs result

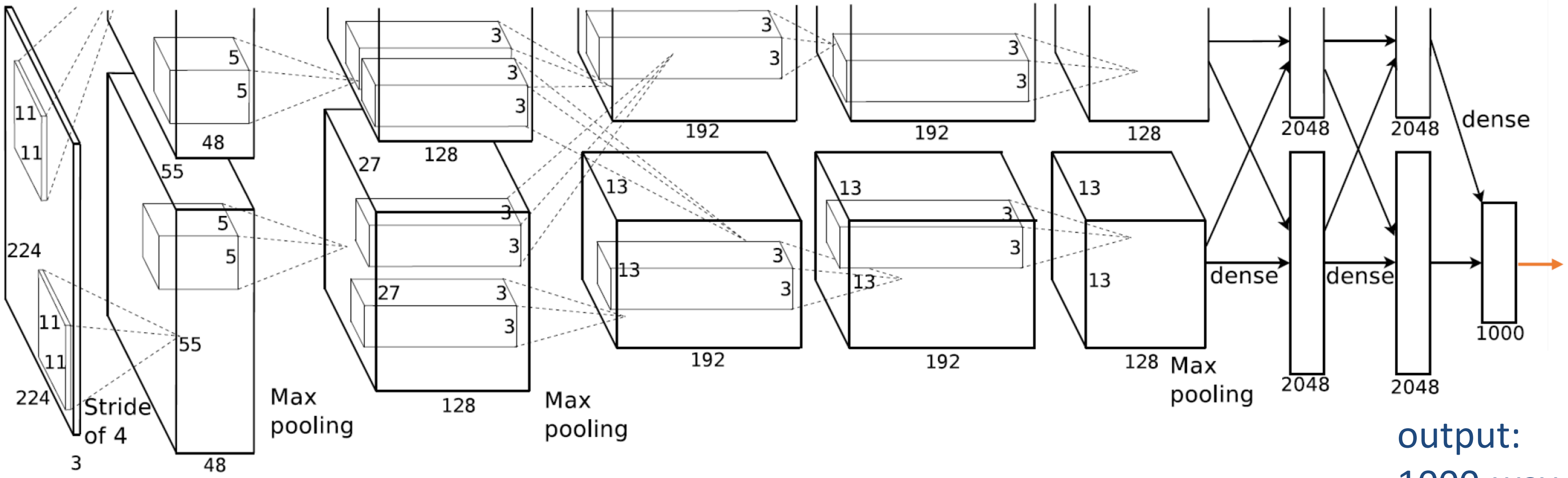


http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf

Architecture

input:
3 channels of the color images

3 fully connected layers



output:
1000-way Softmax

5 convolutional layers

Input normalization and Data augmentation I

Images were down-sampled and cropped to 256×256 pixels and normalized

- 1^{st} : image translations and horizontal reflections
  - random 224x224 patches + horizontal reflections from the 256x256 images
  - Testing: five 224x224 patches + horizontal reflections → averaging the predictions over the ten patches

Data augmentation II

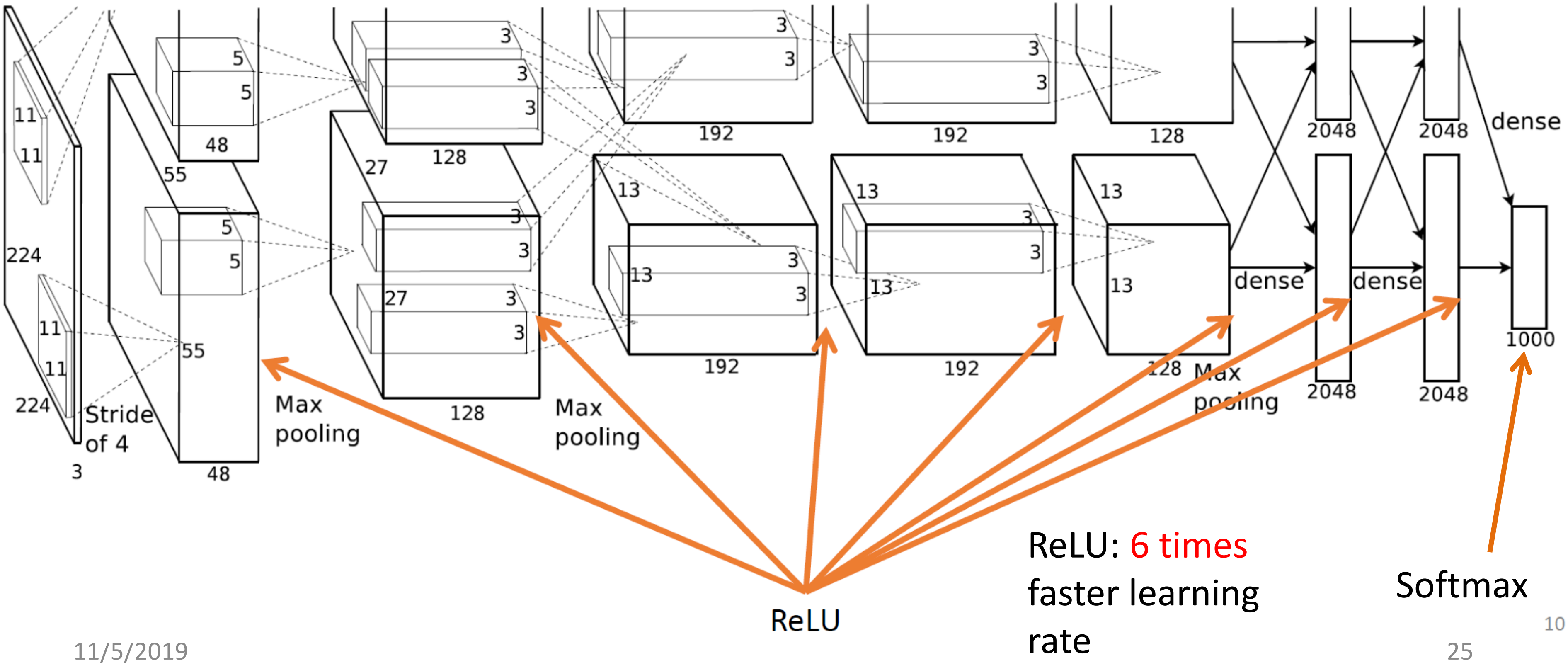
- 2^{nd} : change the intensity of RGB channels
  - PCA on the set of RGB pixel values throughout the ImageNet training set
  - To each RGB image pixel $\mathrm{I_{xy}} = [I_{xy}^R, I_{xy}^G, I_{xy}^B]$ following is added

$$\underbrace{[p_1, p_2, p_3]}_{\text{eigenvectors}}[\alpha_1 \underbrace{\lambda_1}_{}, \alpha_2 \underbrace{\lambda_2}_{}, \alpha_3 \underbrace{\lambda_3}_{}]^T \quad | \alpha_i \sim N(0, 0.1)$$

eigenvalues

  - Improvement:
    - top-1 error rate by 1%

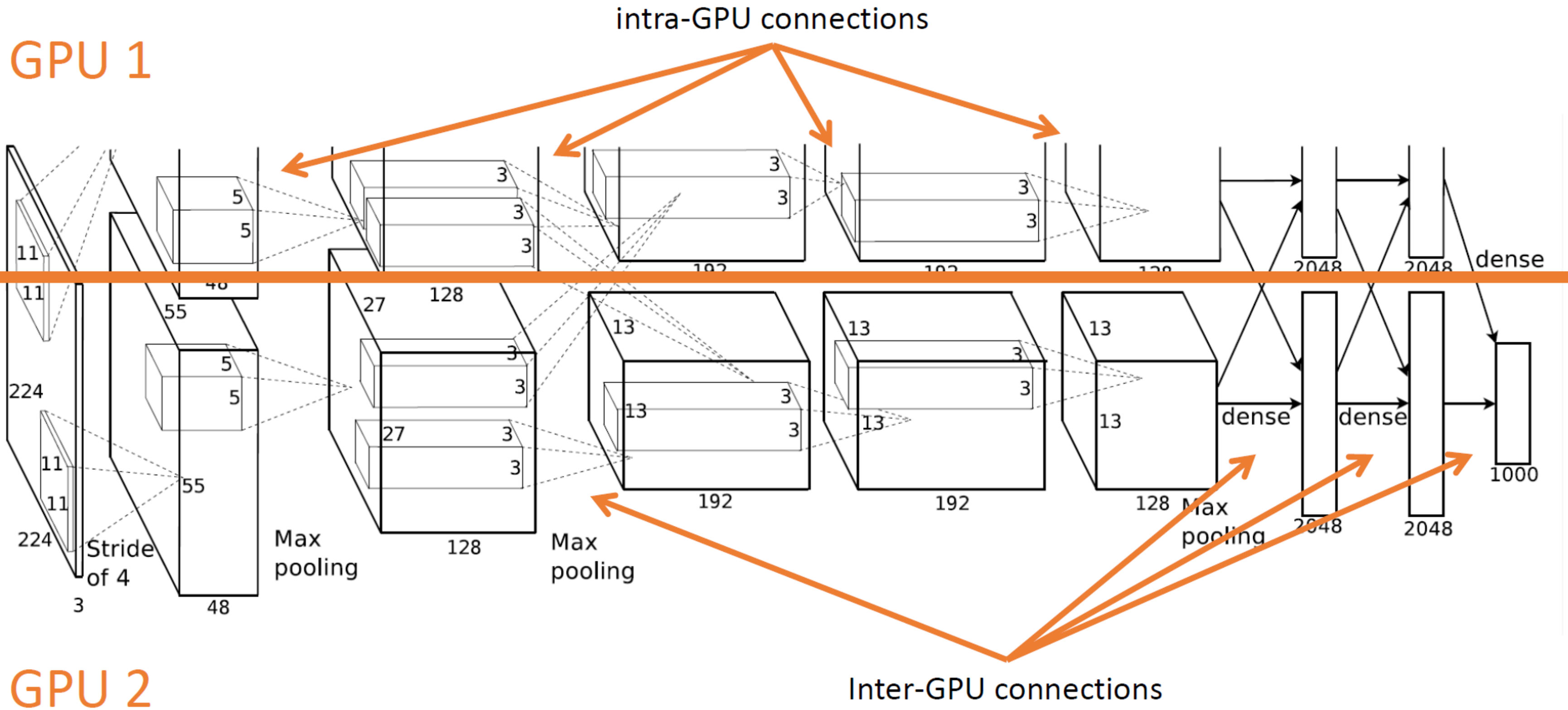
Activation function



Ensembling: duplicating the network I

- Train two architecturally identical copies of the network on two GPUs
  - Half of the neuron layers are on each GPU
    - GPUs communicate only in certain layers
  - Improvement (as compared with a net with half as many kernels in each convolutional layer trained on one GPU):
    - Top 1 error rate by 1.7%
    - Top 5 error rate by 1.2%

Ensembling: duplicating the network II



Local Response Normalization I

- ReLUs do not require input normalization to prevent them from saturating
- However, Local Response Normalization aids generalization

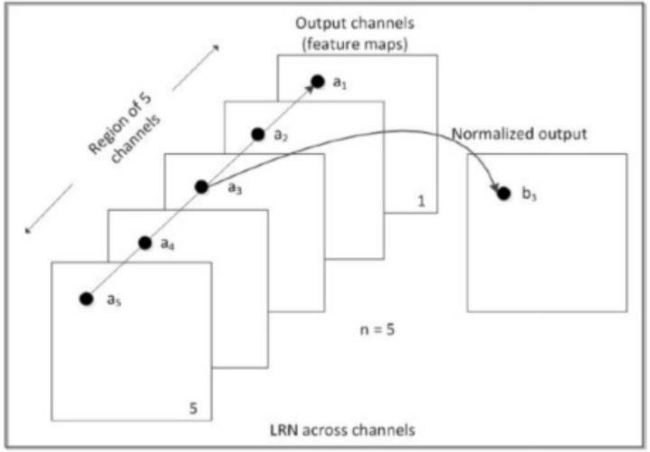
Activity of a neuron by applying kernel i at position (x,y)

$$b^{i}_{x,y} = a^{i}_{x,y} \Big/ \left(k + \alpha \sum_{j=\max\left(0,i-\frac{n}{2}\right)}^{\min\left(N-1,i+\frac{n}{2}\right)} \left(a^{j}_{x,y}\right)^{2} \right)^{\beta}$$

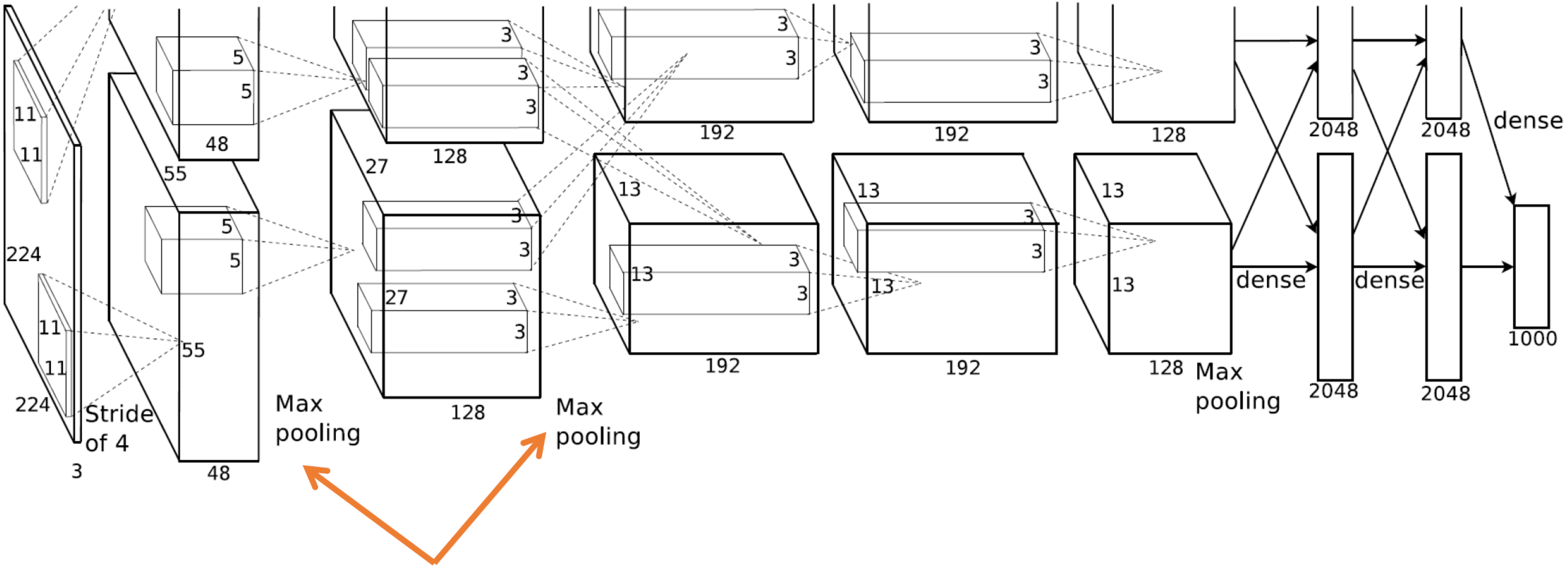
$$k = 2 \quad n = 5 \quad \alpha = 10^{-4} \quad \beta = 0.75$$

- Lateral inhibition (intra-map)
- Improvement:
  - Top error rate by 1.4%
  - Top 5 error rate by 1.2%

sum runs over n “adjacent” kernel maps at the same spatial position



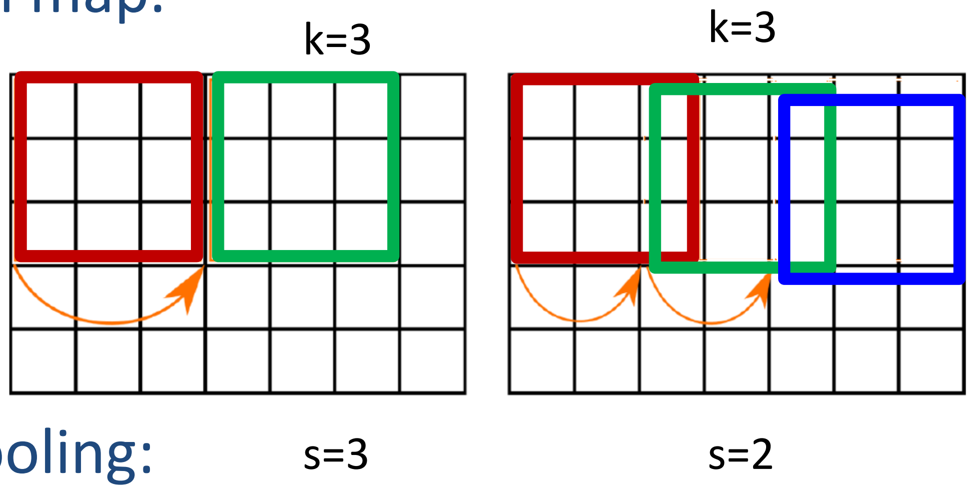
Local Response Normalization II



Local Response Normalization

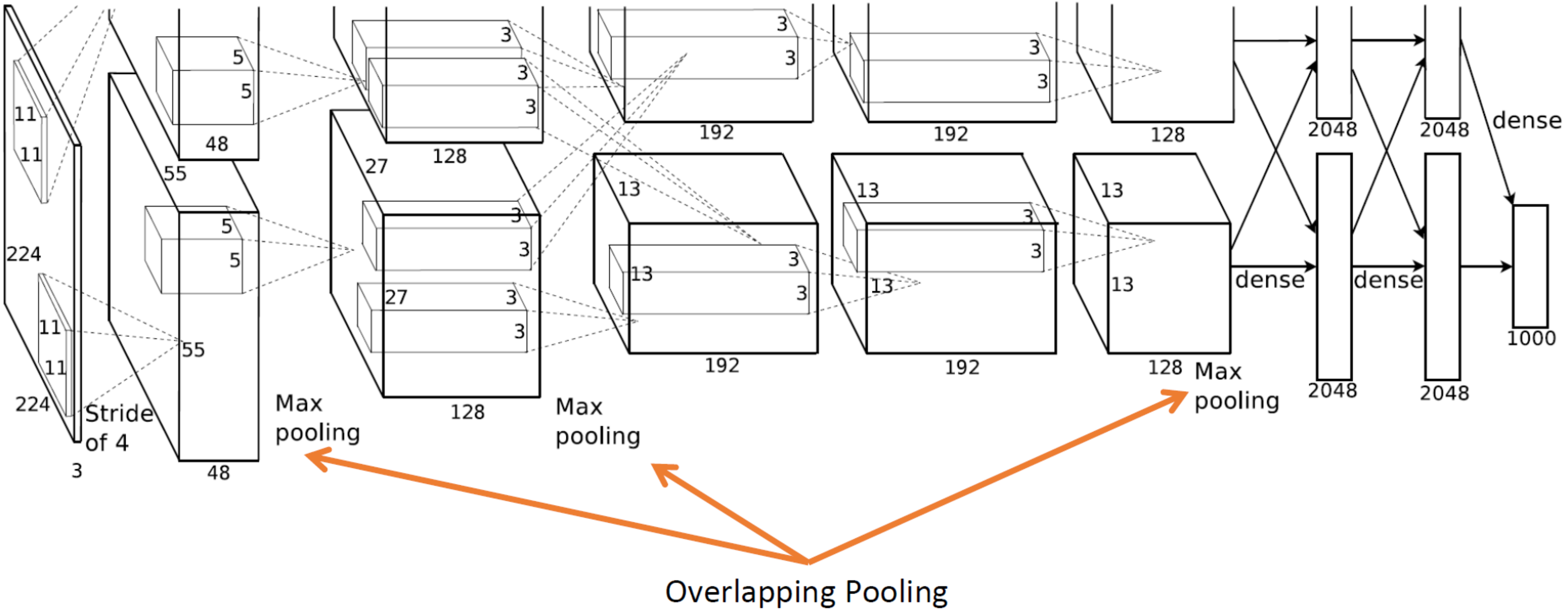
Overlapping Pooling I

- Pooling layers summarize the outputs of neighboring neurons in the same kernel map.
  – Overlapping pooling
  – s < k

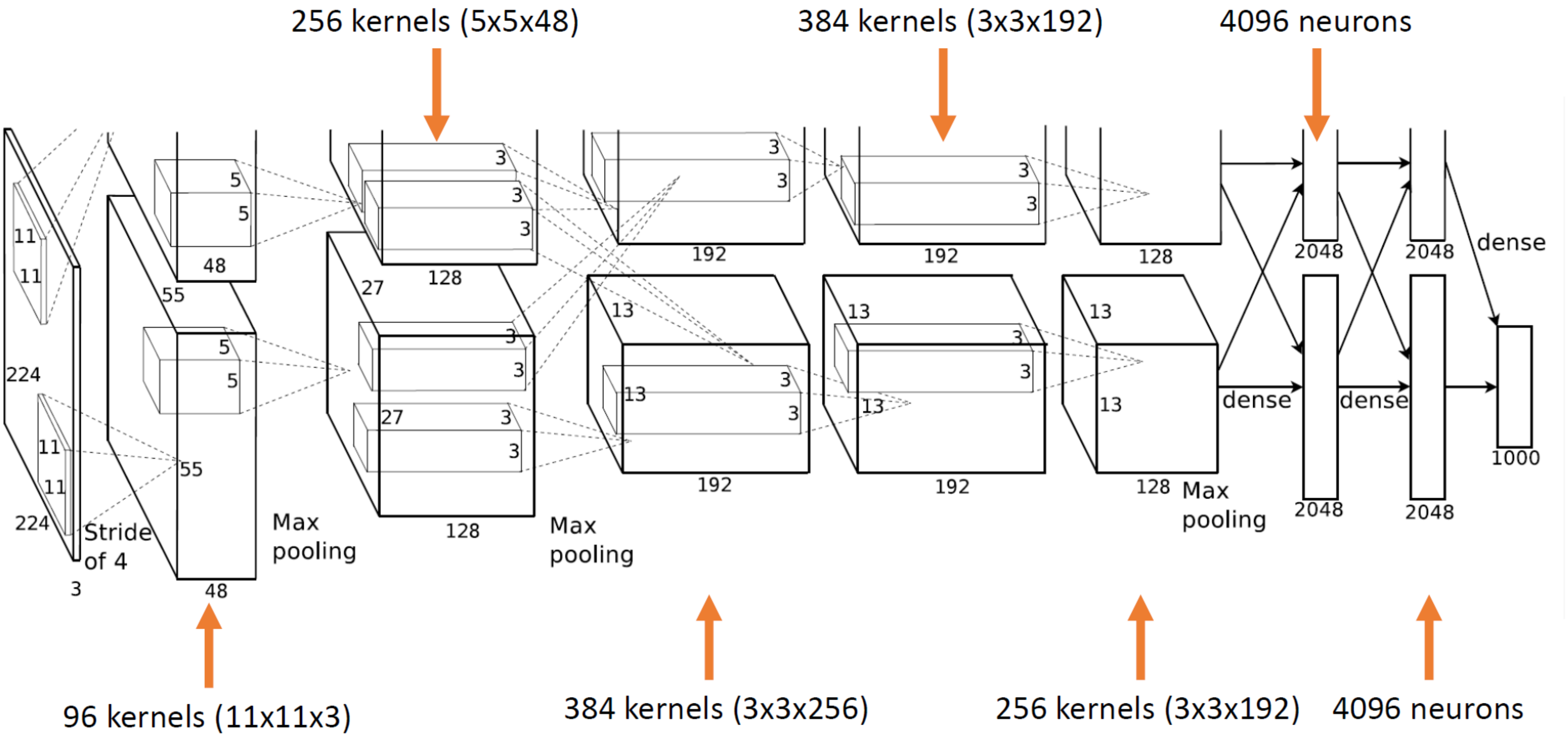


- Improvement using MaxPooling:
  – Top 1 error rate by 0.4%
  – Top 5 error rates by 0.3%

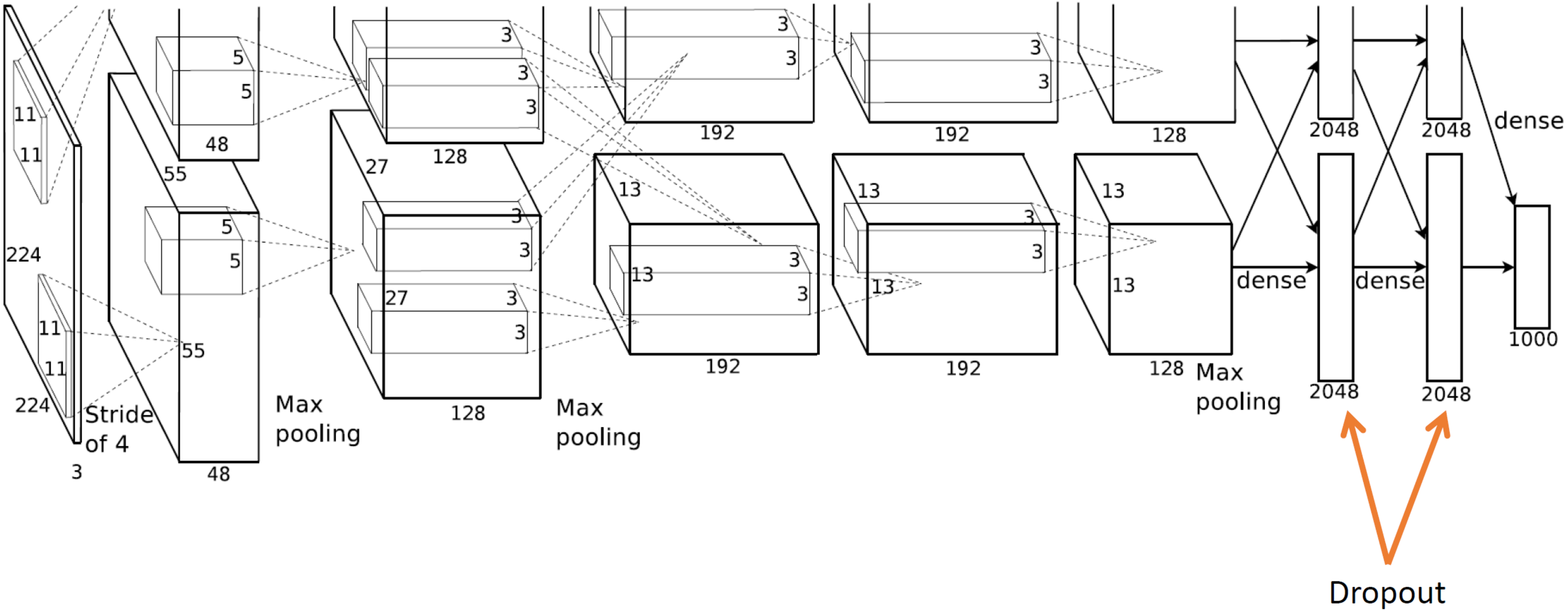
Overlapping Pooling II



Overall Architecture



Dropout layers



Training I

- Stochastic Gradient Descent (with momentum)
  - ADAM method was introduced in 2014 only (2 years later)
- Minimizing the negative log-likelihood (cross-entropy) loss function
- With L2 regularization (weight penalty):

$$L(w) = \sum_{i=1}^{N} \sum_{c=1}^{1000} -y_{ic} \log f_c(x_i) + \epsilon \|w\|_2^2$$

predicted probability of class c for image x

indicator that example i has label c

Training II

- SGD + Momentum with a batch size of 128
- Learning rate initialized at 0.01
  - divided by 10 if validation error rate stopped improving
- Update rule for weight w:

weight decay   learning rate

$$v_{i+1} := 0.9 * v_i - 0.0005 * \epsilon * w_i - \epsilon * \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$$

$$w_{i+1} := w_i + v_{i+1}$$

momentum

Gradient of Loss

- Training effort:
  - ~ 90 epochs → five to six days on two NVIDIA GTX 580 3GB GPUs

Look into the parameters!

- https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html
- 3 layer CNN
- Cifar 10 database
- 32x32 sized color images
- 10 classes
- 6000 images per class

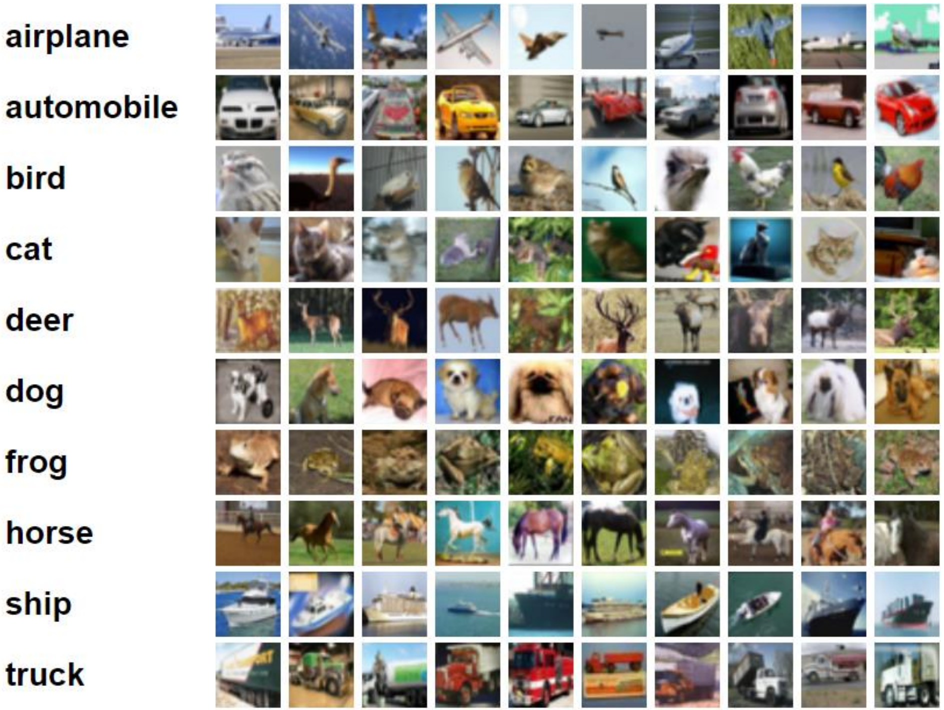
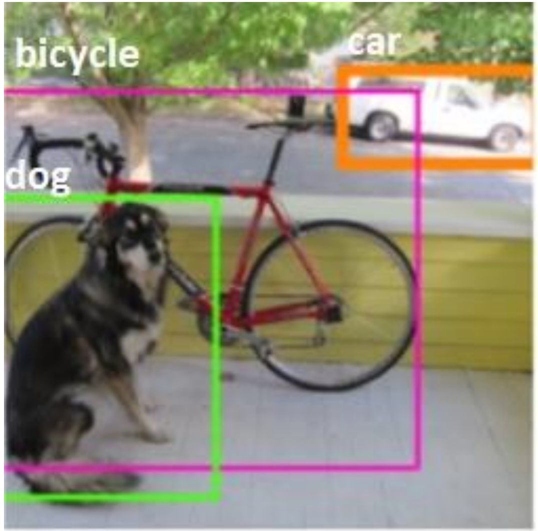


Image understanding beyond classification

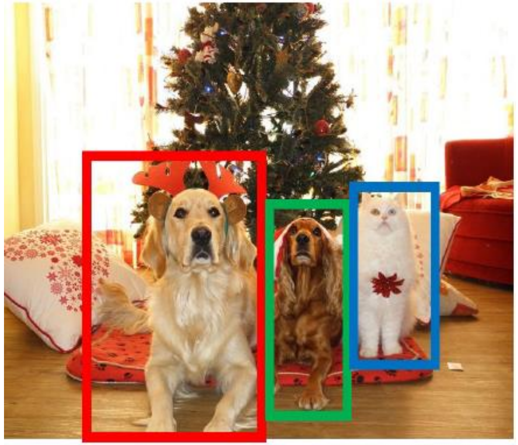
- ImageNet challenge:
  - One dominant object per image
- Real life problems:
  - Multiple objects
    - Same kind of objects
    - Different kinds of objects
    - Overlapping objects
  - Where are the objects?
    - Square them!
    - Find the boundary → Segmentation

container ship



Multiple decisions from each image!

Locality information!

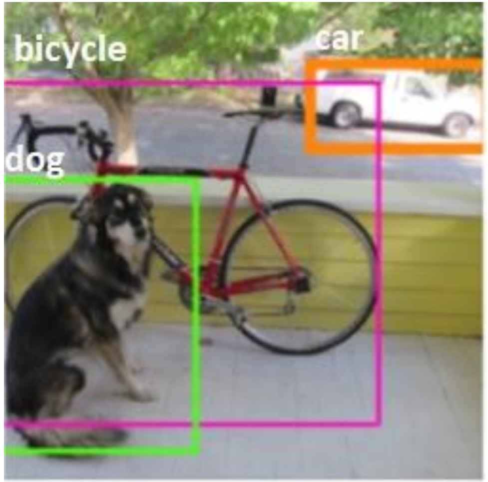


DOG, DOG, CAT

Object recognition

- One object per image
  - Task:
    - Classify image
    - Classes are known (one-of-n decision)

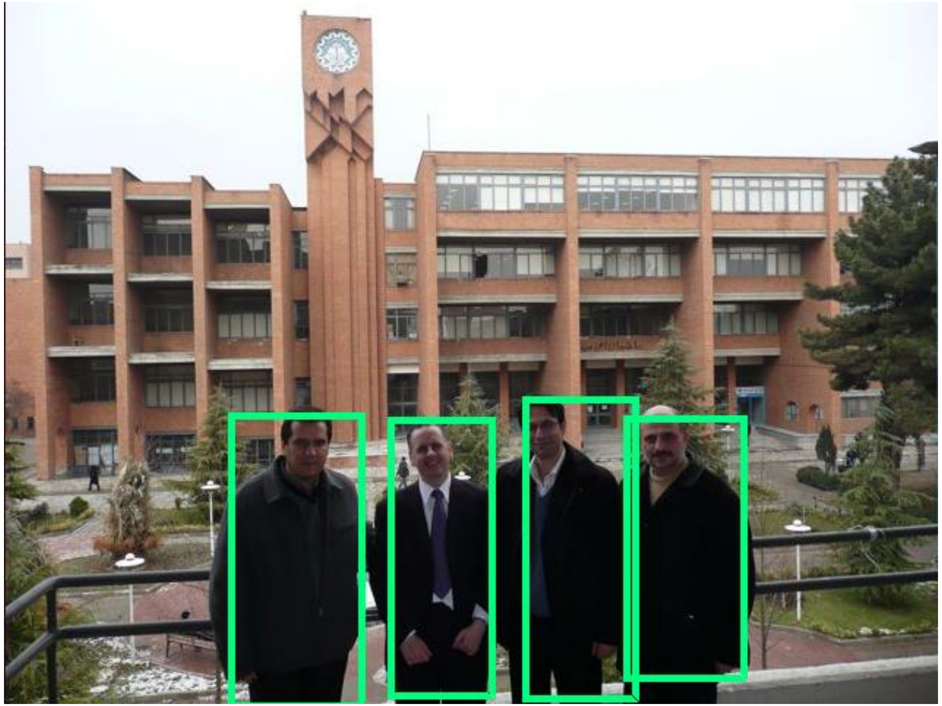
- Multiple object per image
  - Task:
    - Find and classify the objects
    - Find the bounding boxes



Object detection

- One or Multiple object per image
  - Task:
    - Find the objects
    - Identify them with bounding boxes

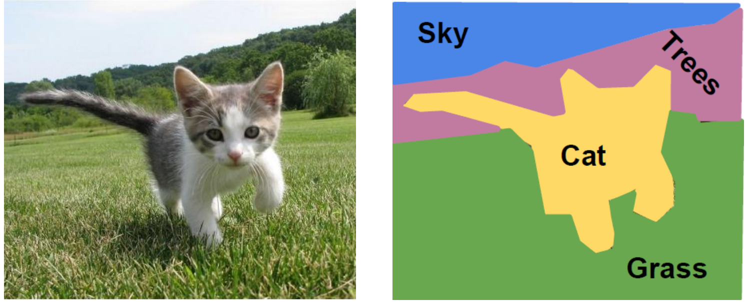
Area or pixel level one-of-two decision!



Segmentation

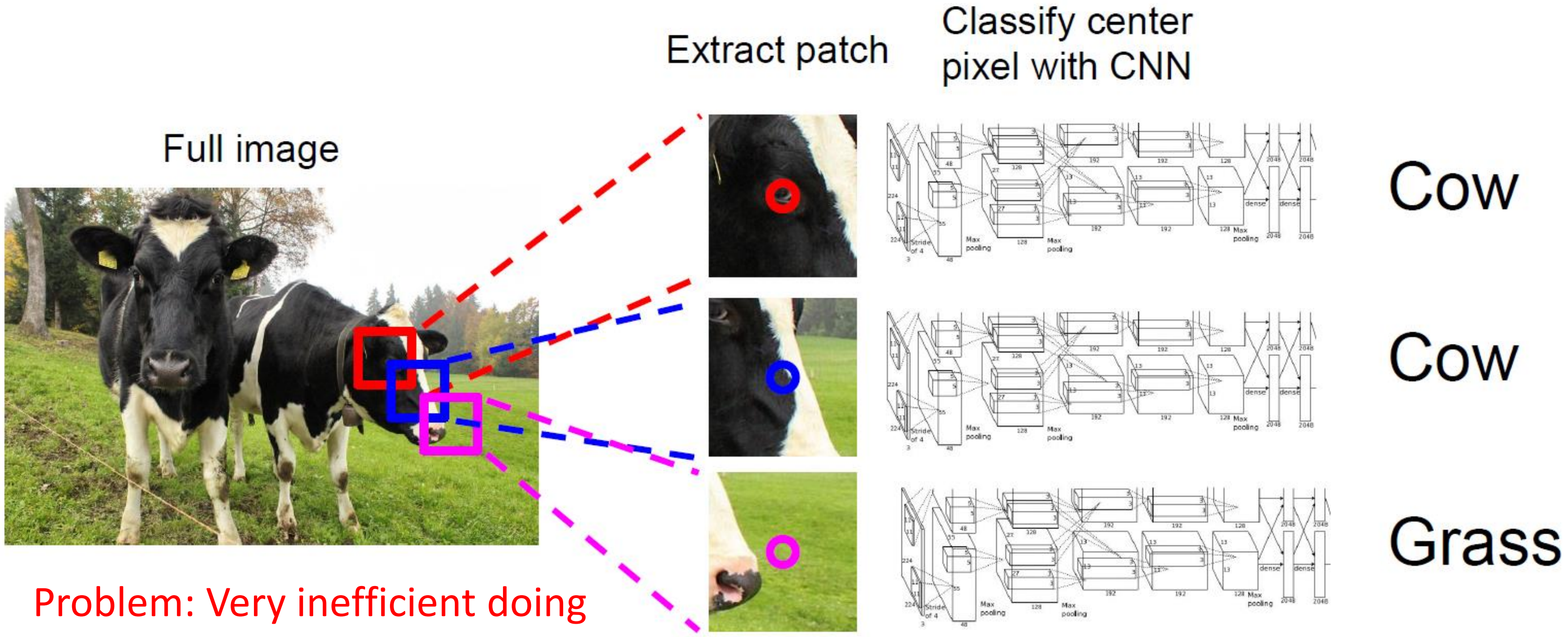
- Semantic Segmentation
  - Label each pixel in the image with a category label
  - Don't differentiate Instances, only care about pixels

Pixel level one-of-n classification!



- Semantic Instance Segmentation
  - Differentiate instances

Semantic Segmentation Idea I: Sliding Window

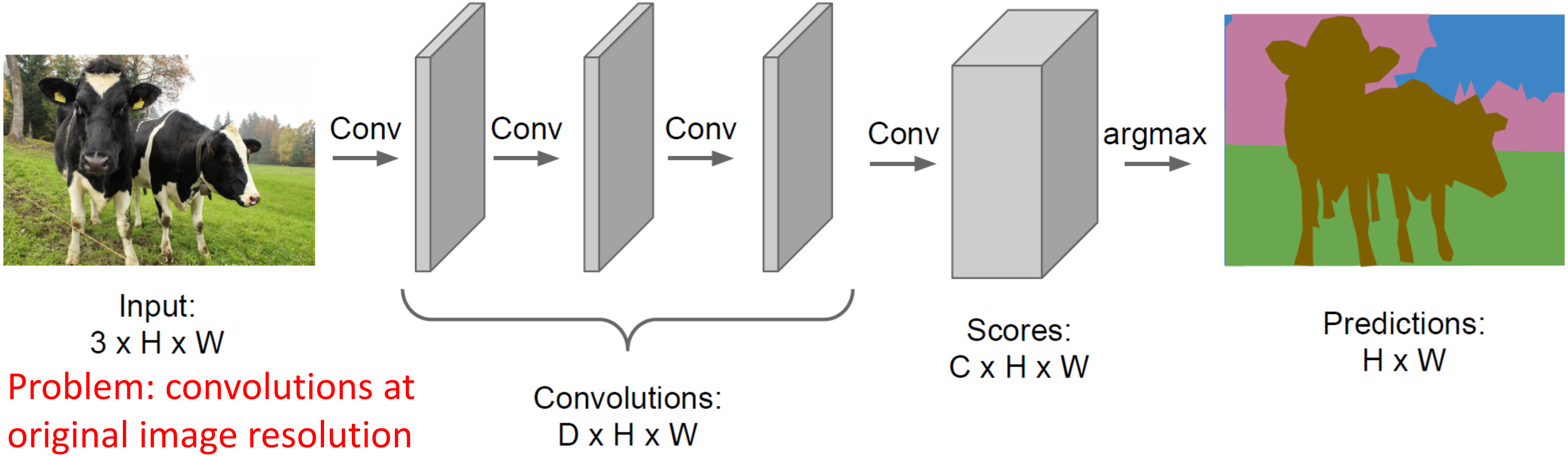


Problem: Very inefficient doing it pixel-wise! No reuse of shared features between overlapping Patches.

Farabet et al, "Learning Hierarchical Features for Scene Labeling," TPAMI 2013
Pinheiro and Collobert, "Recurrent Convolutional Neural Networks for Scene Labeling", ICML 2014

Semantic Segmentation Idea II: Fully Convolutional

Design a network as a bunch of convolutional layers
to make predictions for pixels all at once!



Problem: convolutions at original image resolution will be very expensive ...

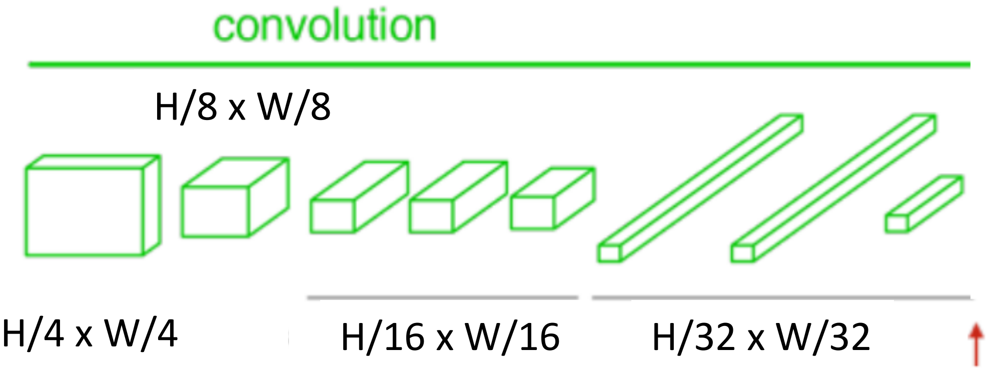
Semantic Segmentation Idea III: Fully Convolutional

Downsampling:
Pooling, strided convolution

Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Upsampling:
???

Input:
3 x H x W



Conv, pool,
nonlinearity

upsampling

Predictions:
H x W

Upsampling I: “Unpooling”

Nearest Neighbor

1	2
3	4

Input: 2 x 2

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

Output: 4 x 4

“Bed of Nails”

1	2
3	4

Input: 2 x 2

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

Output: 4 x 4

Upsampling I: "Unpooling"

Max Pooling
Remember which element was max!

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

Input: 4 x 4

5	6
7	8

Output: 2 x 2

. . .

Rest of the network

Max Unpooling
Use positions from pooling layer

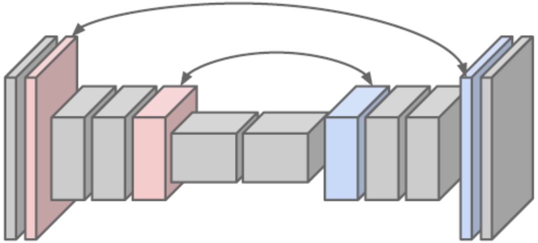
1	2
3	4

Input: 2 x 2

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

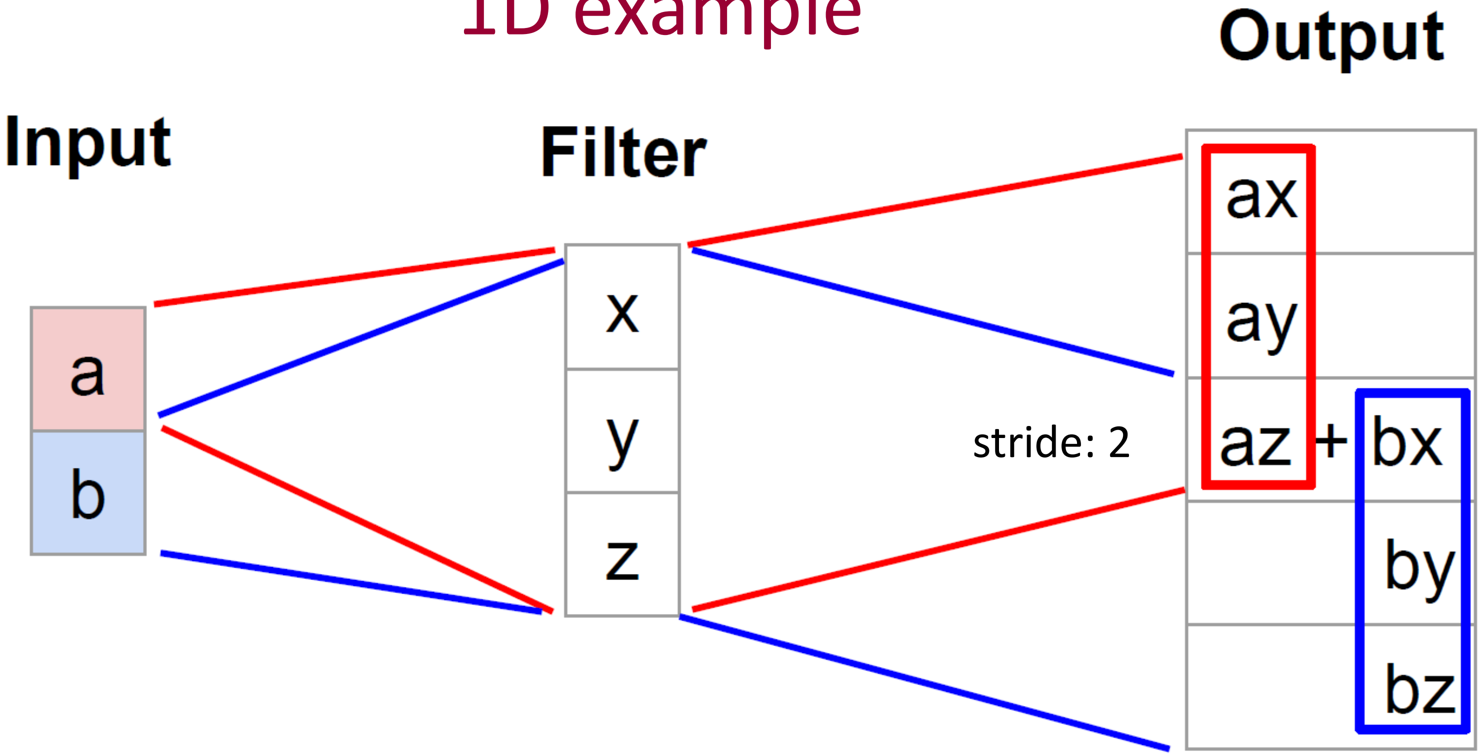
Output: 4 x 4

Corresponding pairs of downsampling and upsampling layers



Upsampling II: "transpose convolution"

1D example



Output contains copies of the filter weighted by the input, summing at where at overlaps in the output

Need to crop one pixel from output to make output exactly 2x input

2D transposed convolution

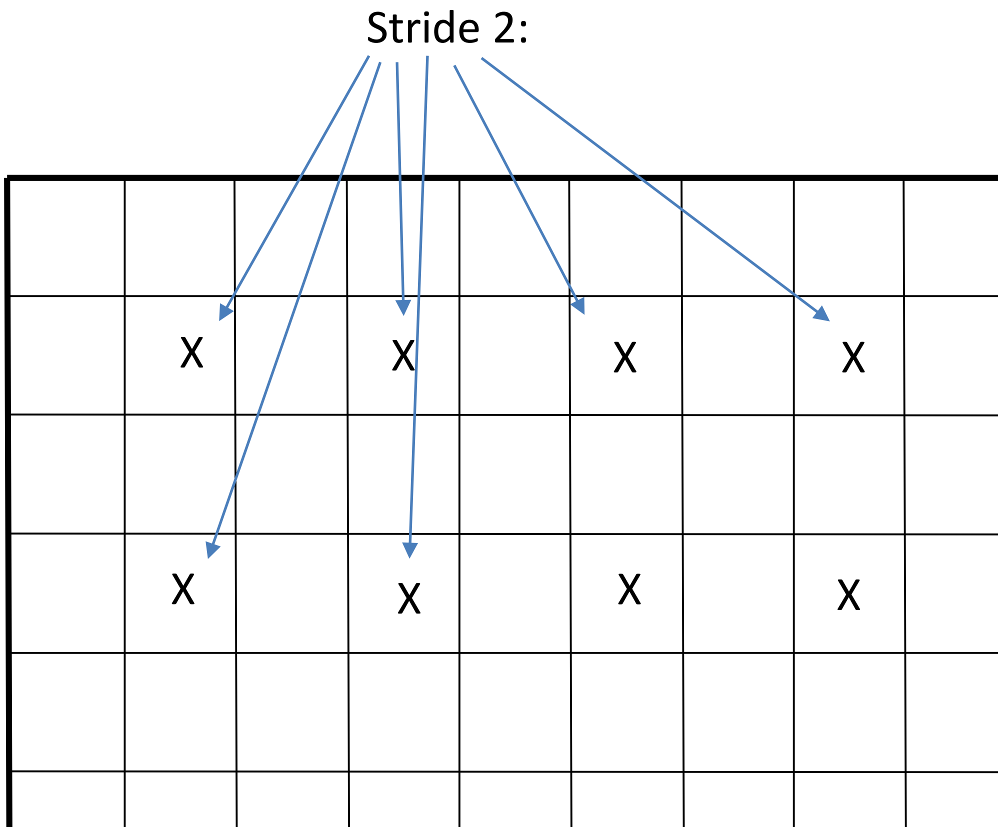
1	1	1
1	1	1
1	1	1

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps



2D transposed convolution

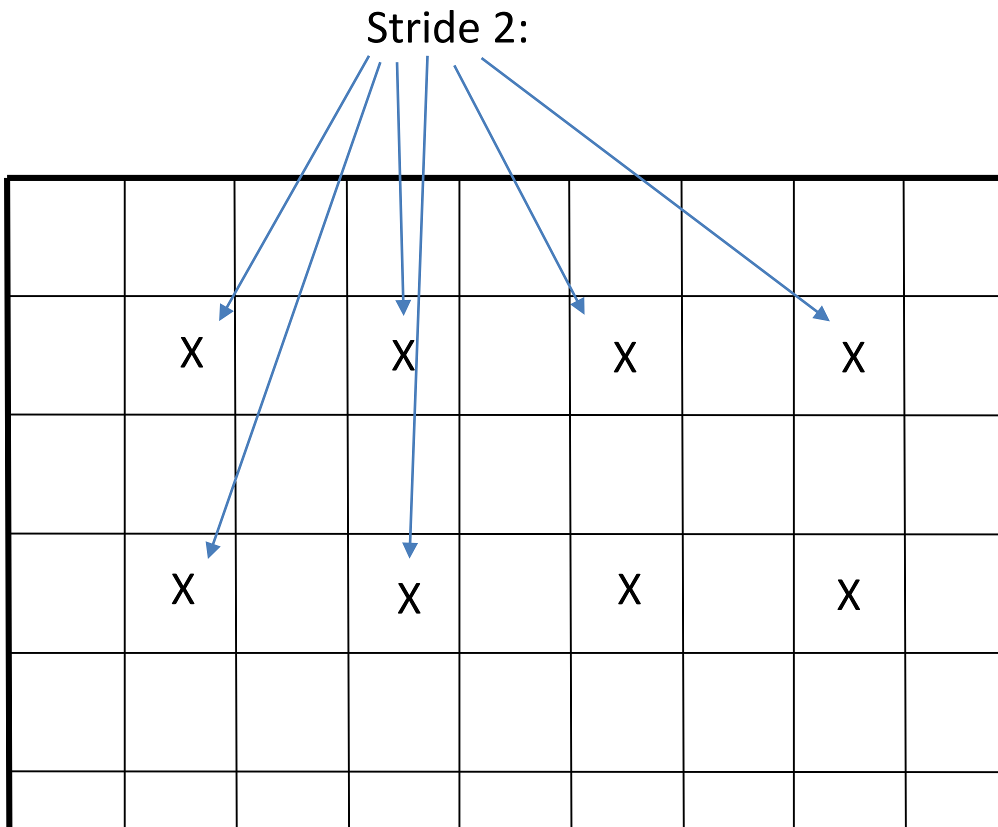
1	1	1
1	1	1
1	1	1

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps



2D transposed convolution

1	1	1
1	1	1
1	1	1

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1						
1	1	1	X		X		X	
1	1	1						
	X		X		X		X	

2D transposed convolution

2	2	2
2	2	2
2	2	2

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1						
1	1	1	X		X		X	
1	1	1						
	X		X		X		X	

2D transposed convolution

2	2	2
2	2	2
2	2	2

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1 2	2	2				
1	1	1 2	2	2	X		X	
1	1	1 2	2	2				
	X		X					

2D transposed convolution

3	3	3
3	3	3
3	3	3

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1 2	2	2				
1	1	1 2	2	2	X		X	
1	1	1 2	2	2				
	X		X		X		X	

2D transposed convolution

3	3	3
3	3	3
3	3	3

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1 2	2	2				
1	1	1 2	2	2	X		X	
1 3	1 3	1 2 3	2	2				
3	3	3	X		X		X	
3	3	3						

2D transposed convolution

4	4	4
4	4	4
4	4	4

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1 2	2	2				
1	1	1 2	2	2	X		X	
1 3	1 3	1 2 3	2	2				
3	3	3	X		X		X	
3	3	3						

2D transposed convolution

4	4	4
4	4	4
4	4	4

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1 2	2	2				
1	1	1 2	2	2	X		X	
1 3	1 3	1 2 3 4	2 4	2 4				
3	3	3 4	4	4	X		X	
3	3	3 4	4	4				

2D transposed convolution

4	4	4
4	4	4
4	4	4

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	1 2	2	2				
1	1	1 2	2	2	X		X	
1 3	1 3	1 2 3 4	2 4	2 4				
3	3	3 4	4	4	X		X	
3	3	3 4	4	4				

2D transposed convolution

4	4	4
4	4	4
4	4	4

kernel

1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Stride 2:

1	1	3	2	2				
1	1	3	2	2				
4	4	10	6	6				
3	3	7	4	4	X		X	
3	3	7	4	4				

2D transposed convolution

5	5	5
5	5	5
5	5	5

kernel

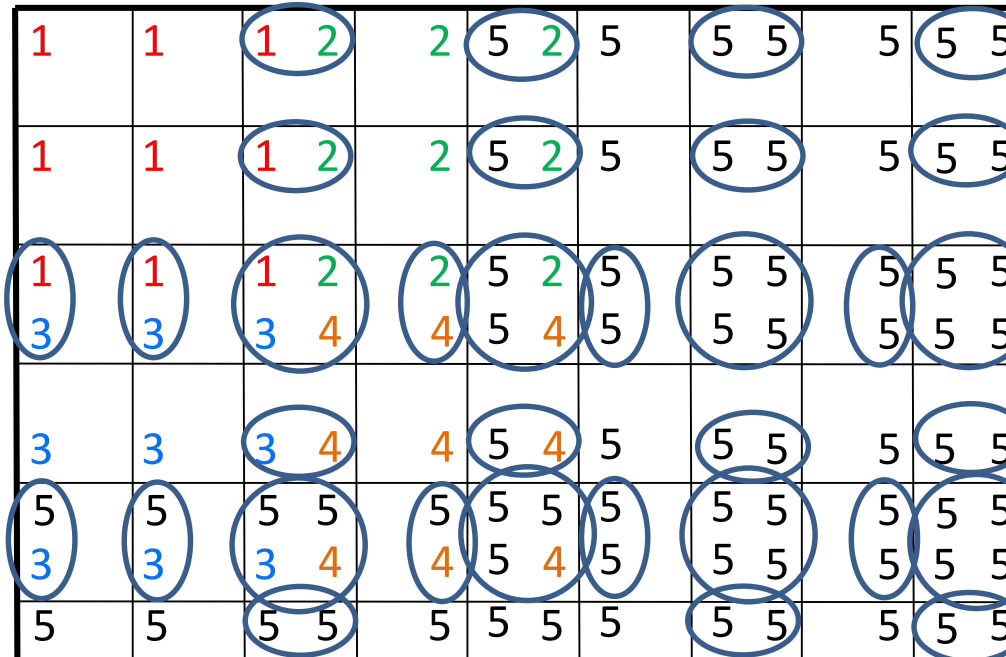
1	2	5	5
3	4	5	5
5	5	5	5
5	5	5	5

image

1. Kernel is weighted with the input pixel value
2. Placed to the stride positions
3. Summed up where overlaps

Note: the summing positions are not homogenious

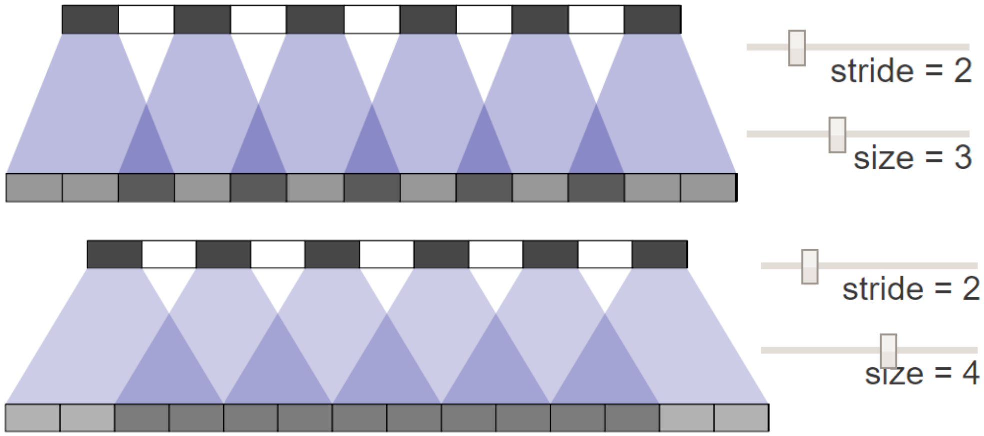
Stride 2:



Transpose convolution artefact:

Avoiding checkerboard effect

- Non-homgenious transpose convolution causes checkerboard patterns
- Balanced stripe and kernel size can make it homogenious



https://distill.pub/2016/deconv-checkerboard/

Fully Convolutional Network

Downsampling:
Pooling, strided convolution

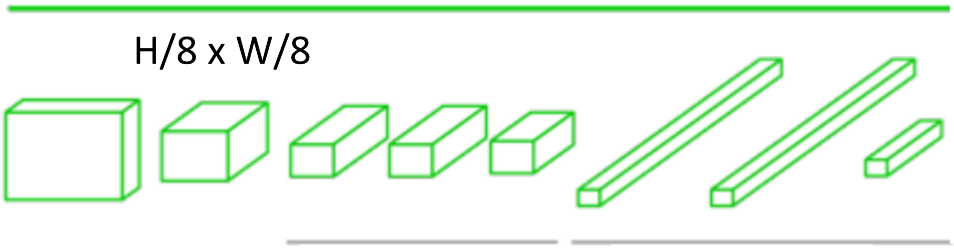
Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Upsampling:
Unpooling or strided transpose convolution

Input:
3 x H x W

convolution

H/8 x W/8



H/4 x W/4

H/16 x W/16

H/32 x W/32

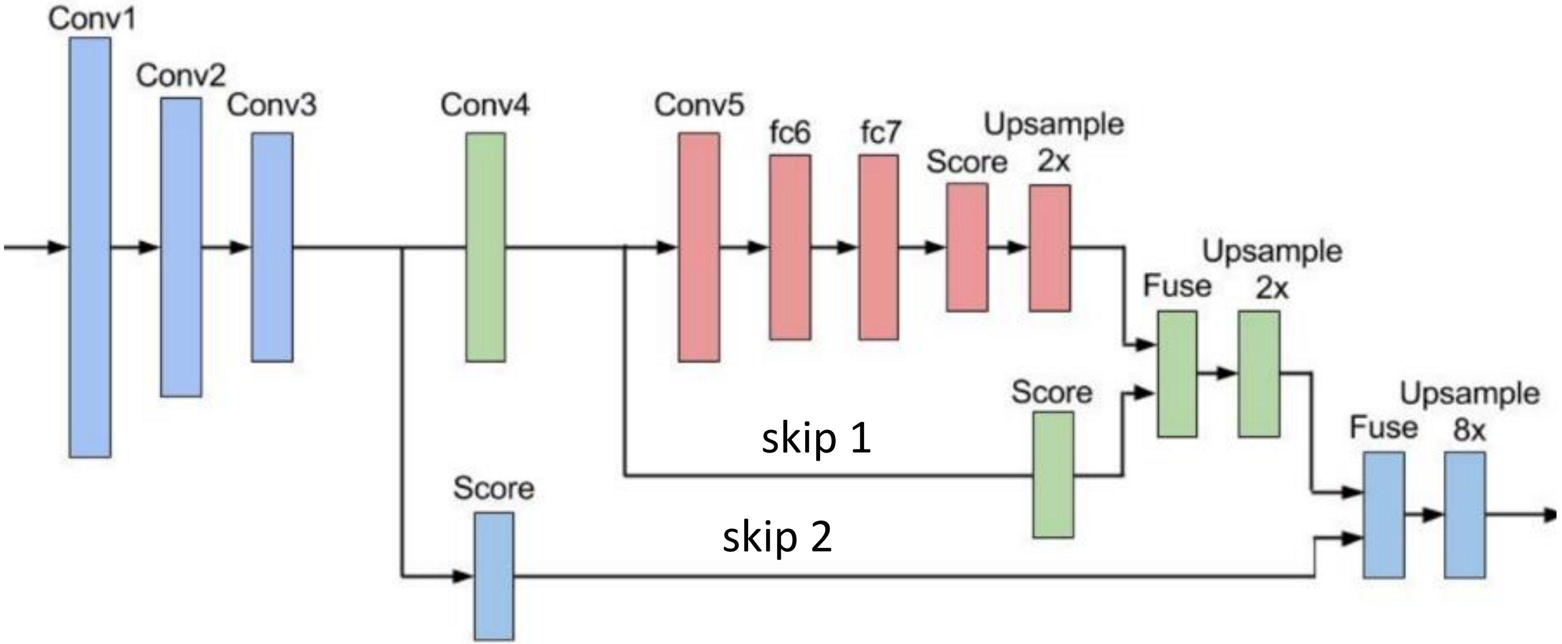
Conv, pool,
nonlinearity

upsampling

Predictions:
H x W

- As many output layers as many classes
- Pixel-wise Softmax output function is used with negative log-likelihood loss (multi-class cross entropy) function

Increasing spatial resolution in segmentation I

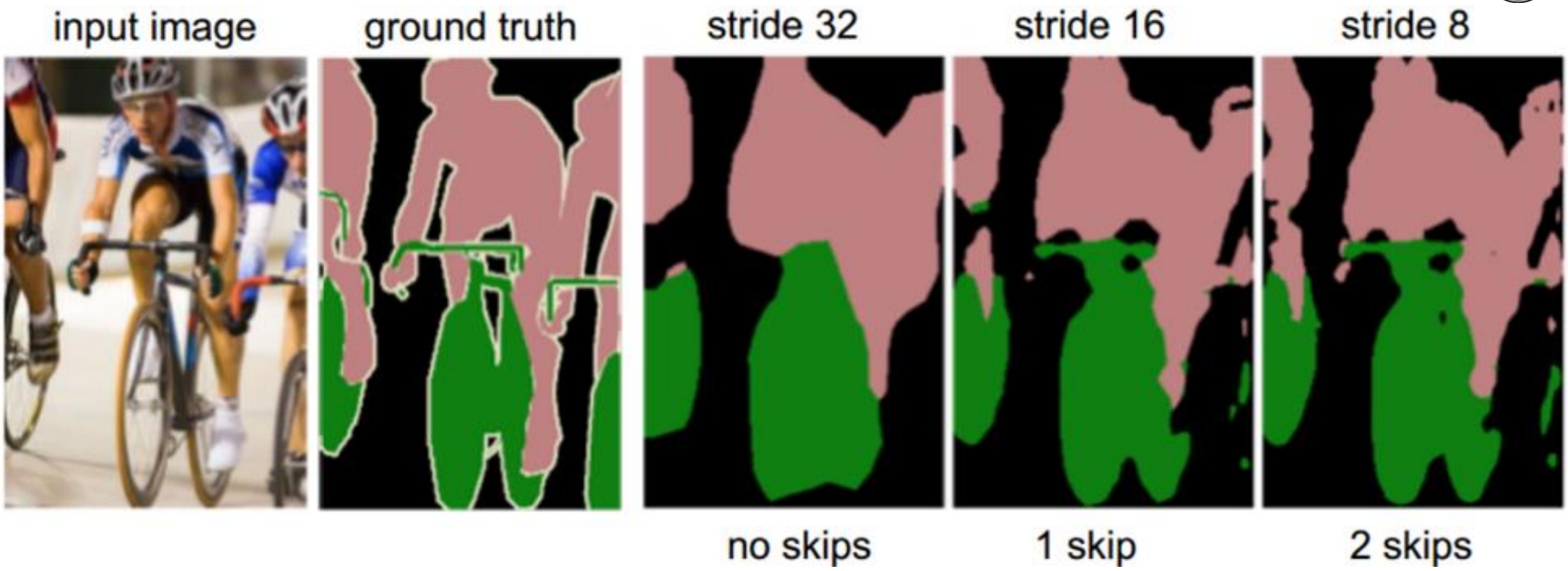


Higher resolution layers directly forwarded to transfer finer spatial information

Called “Skipping”. It skips using the coarser (more downsampled) layers

Can be considered of an ensembling of three networks

Increasing spatial resolution in segmentation II

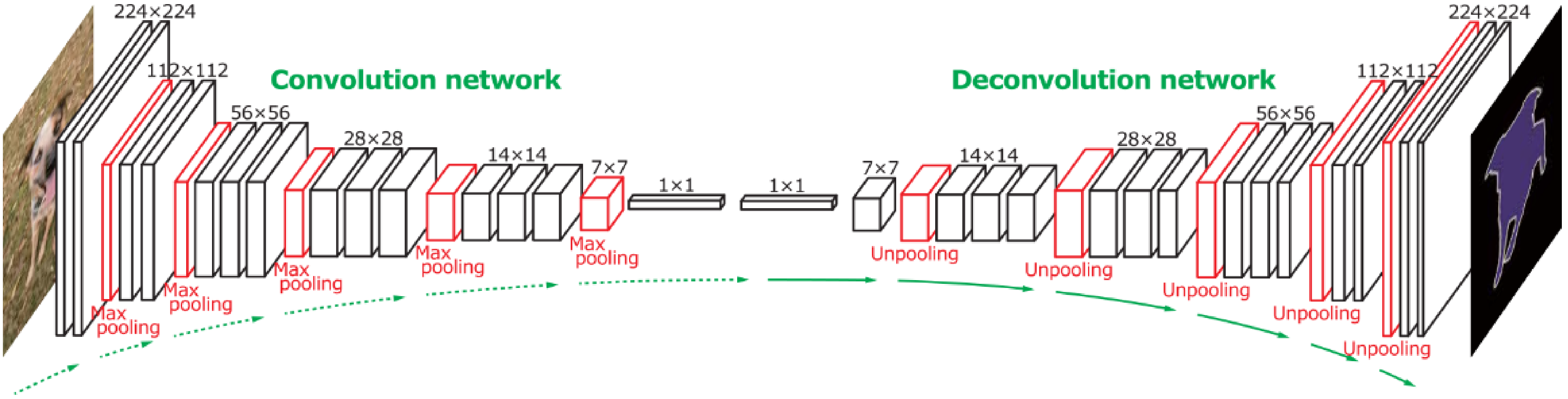


Inreasing spatial resolution as higher resolution layers are feed forward

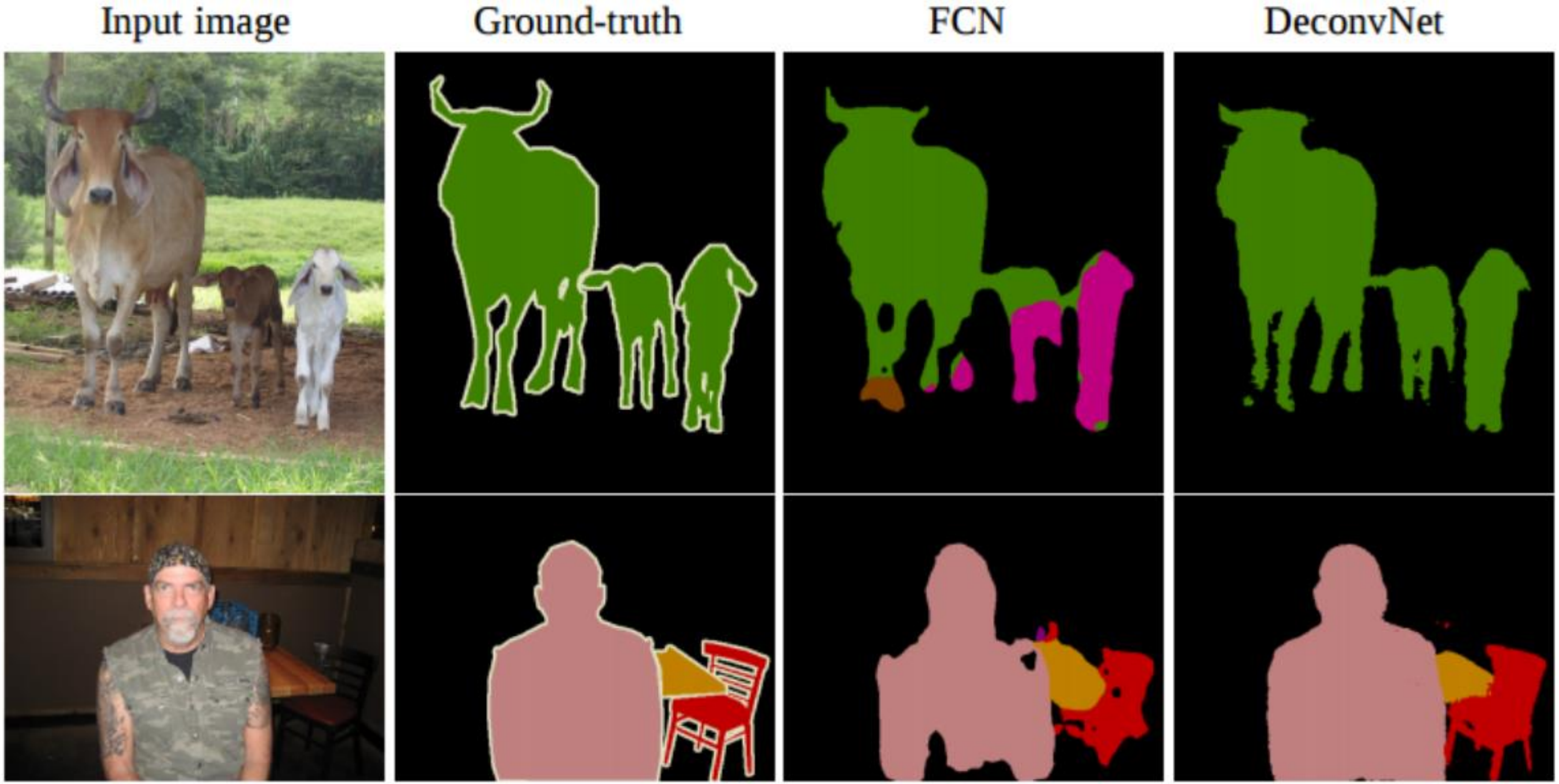
Information content is less squeezed to smaller layer

Deconvnet: Extreme segmentation I

- Fully symmetrical convolutional network
  - All convolution and pooling layers are reversed
- Two stage training (first side trained for classification first)
- Takes 6 days to train on titan GPU
- Output probability map same size as input



Deconvnet: Extreme segmentation II



Neural Networks

Semantic Segmentation

(P-ITEEA-0011)

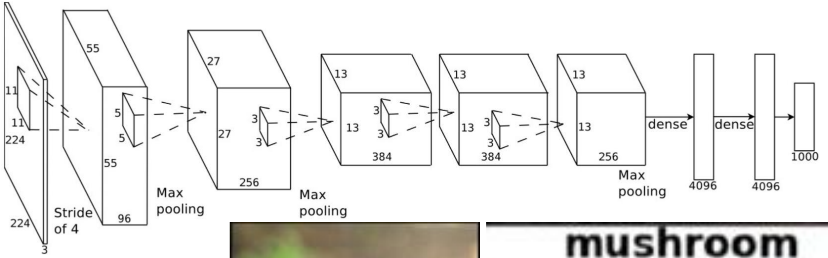
Akos Zarandy
Lecture 8
November 12, 2019

Announcement

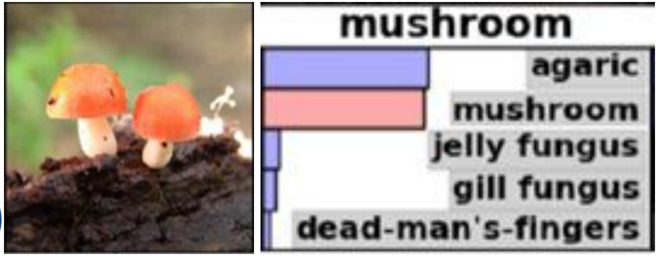
- Midterm project were taken by many people ☺
- Midterm project counts for those
  - Paper based test result is 5
    - Can get offered grade 4 or 5 based on the quality of the midterm project solution
  - Paper based test result is 5
    - Can get offered grade 3 only if the quality of the midterm project solution is satisfactory
    - One can go for better grade in exam period
- If somebody changes his/her mind about midterm project after this announcement, he or she has to write a letter to Soma Kontar **today!**

Short quiz 60% required!

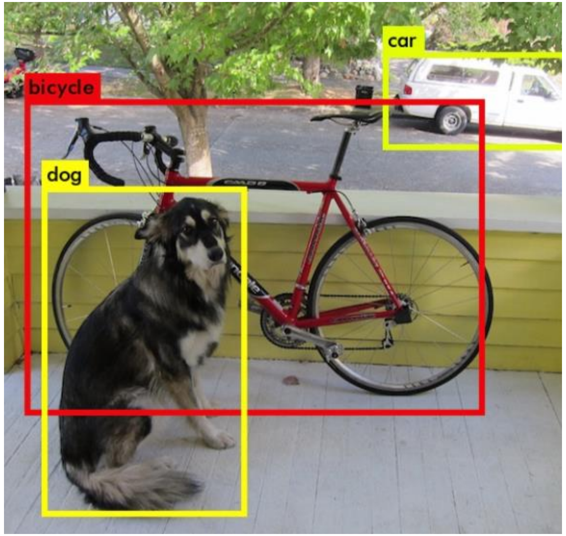
Recap



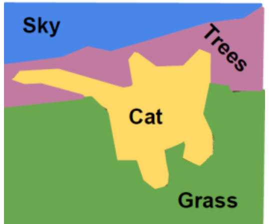
- Last Lecture we discussed
  - How to do image classification
    - Alexnet
    - One decision per image (*classification*)



  - Detection and Localization is more complex
    - Multiple (few) decision per image
      - *Regressions* for localization
      - *Classification* for detection



  - Pixel level Segmentation
    - Very high number of decisions (*classification*) per image

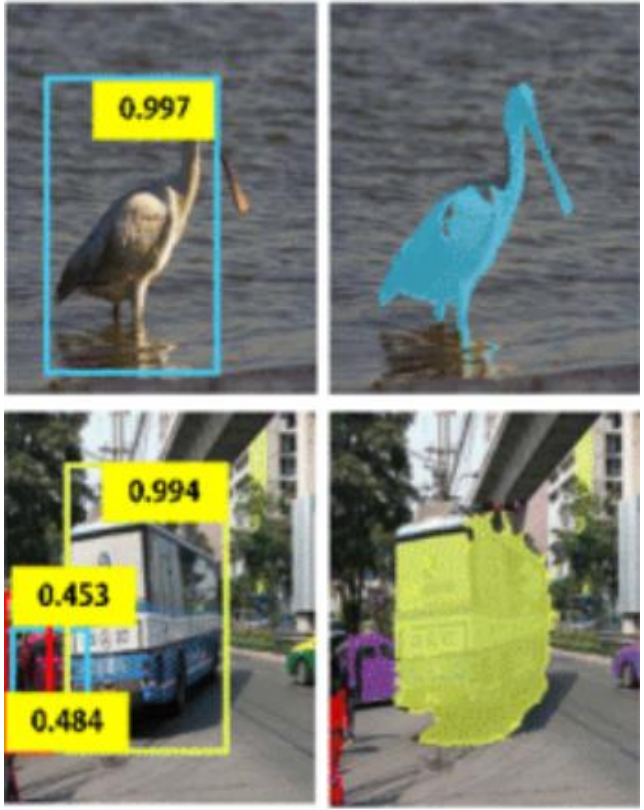


Contents

- Detection and Localization
  - PASCAL Database and Competion
  - R-CNN
    - Region proposal, Classification
    - Support Vector Machine (SVN), Bounding box refinement
  - Fast R-CNN
  - Faster R-CNN
- Semantic Image Segmentation
  - U-Net
  - DeConvNet
  - SegNet
    - Resolution controlling
    - Atrous convolutions, sub-pixel image combination

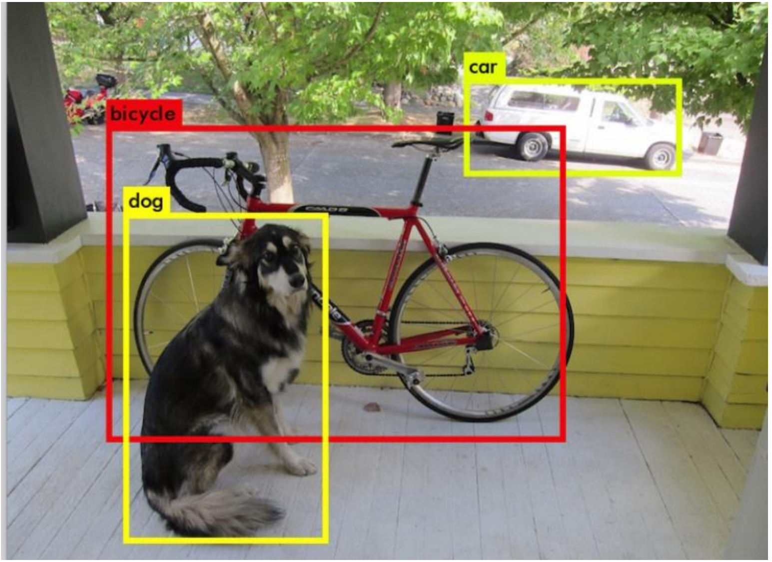
The PASCAL Object Recognition Database and Challange

- Annotated image database
  - Detection (squared objects)
  - Segmentation (segmented objects)
- Challenge
  - The PASCAL Visual Object Classes Challenge (PASCAL VOC)



Object detection/localization and classification

- Chicken and egg problem
  - You need to know that it is a bicycle before able to say that both a wheel part and a pipe segment belongs to the same object
  - You need to know that the red box contains an object before you can recognize it. (Cannot recognize a bicycle if you try it from separated parts)
- Our brain does it parallel
- How neural nets can solve it?
  - Detection by regression?
  - Detection by classification?



Detection as Regression?

(finding bounding box coordinates)

DOG, (x, y, w, h)
CAT, (x, y, w, h)
CAT, (x, y, w, h)
DUCK (x, y, w, h)

= 16 numbers

Detection as Regression?

(finding bounding box coordinates)

DOG, (x, y, w, h)
CAT, (x, y, w, h)

= 8 numbers

Detection as Regression?

(finding bounding box coordinates)

CAT, (x, y, w, h)
CAT, (x, y, w, h)
....
CAT (x, y, w, h)

= many numbers

Need variable sized outputs

Detection as Classification

(classify the content of each bounding boxes)

CAT? NO

DOG? NO

Detection as Classification

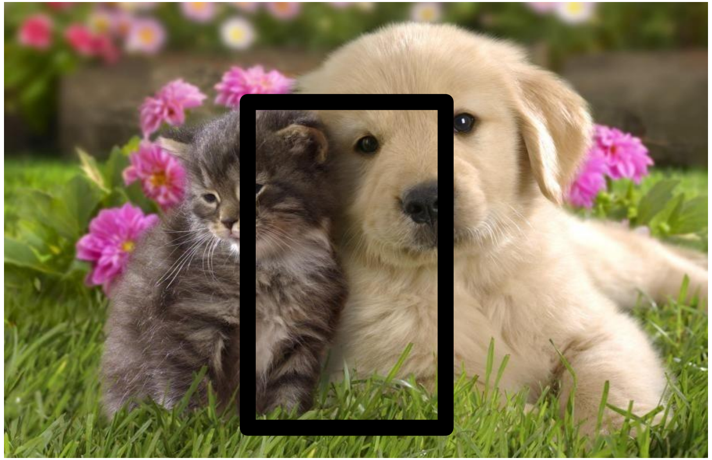
(classify the content of each bounding boxes)

CAT? YES!

DOG? NO

Detection as Classification

(classify the content of each bounding boxes)



CAT? NO

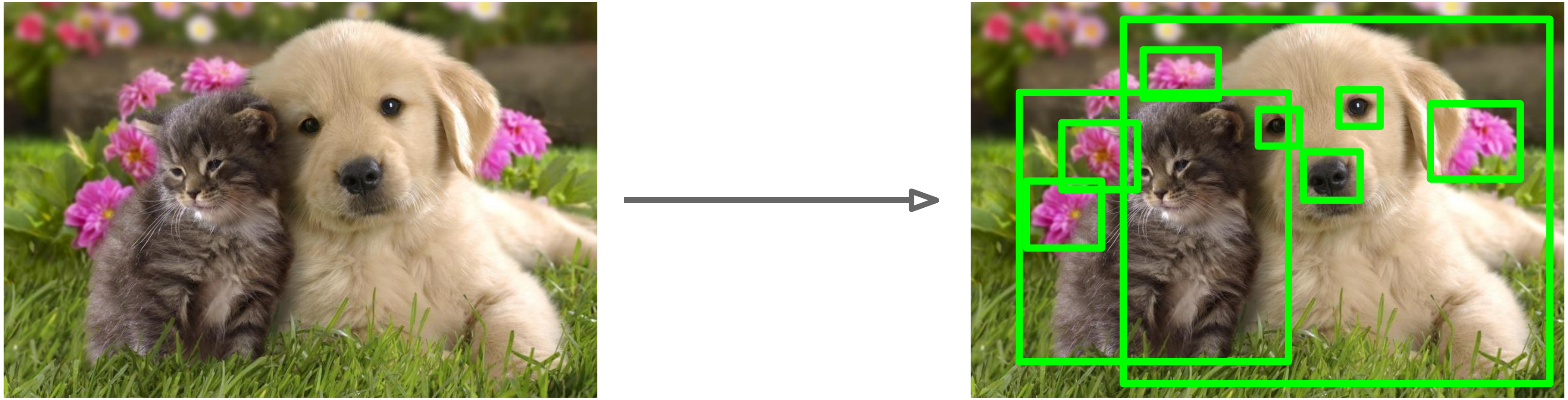
DOG? NO

Problem: Need to too test many positions and scales, and use a computationally demanding classifier (CNN)

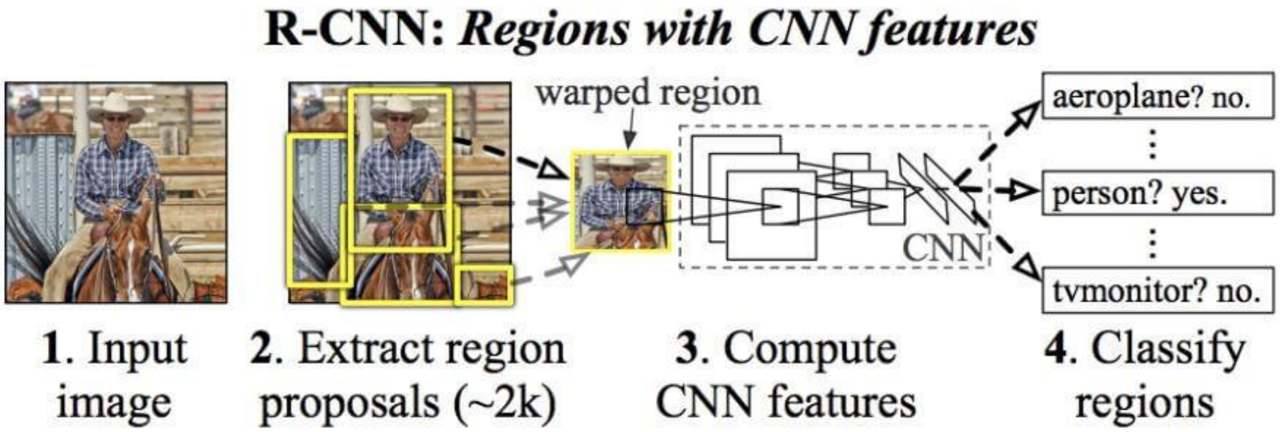
Solution: Only look at a tiny subset of possible positions

Region Proposals

- Find “blobby” image regions that are likely to contain objects
- “Class-agnostic” object detector
- Look for “blob-like” regions

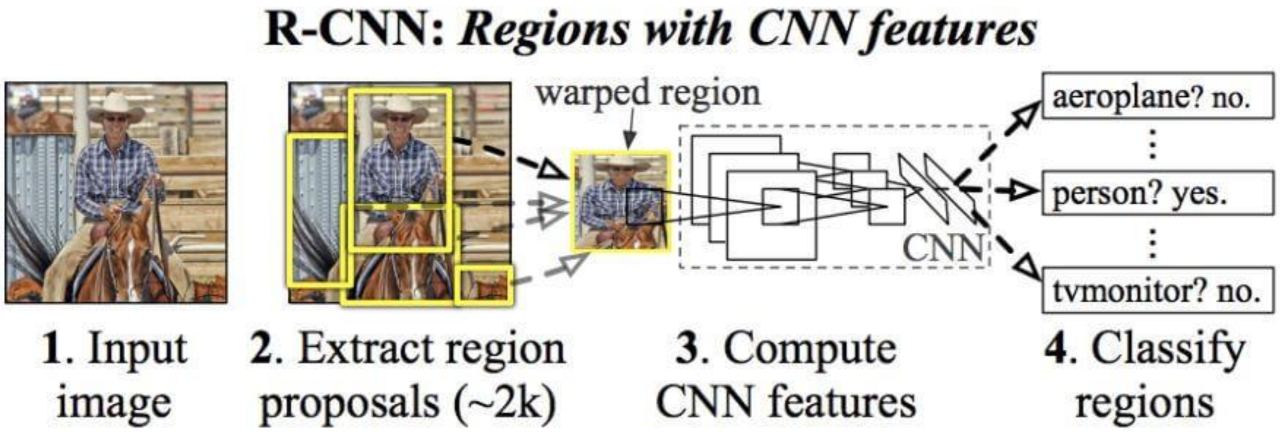


R-CNN in a Glance



1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

The R-CNN algorithm



1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

R-CNN: Region Proposal

- Requirements:
  - Propose a large number (up to 2000) of regions (boxes) with different sizes
  - Still much better than exhausting search with multi-scale sliding window (brute force)
  - Boxes should contain all the candidate objects with high probability
- R-CNN works with various Region proposal methods:
  - Objectness
  - Constrained Parametric Min-Cuts for Automatic Object Segmentation
  - Category Independent Object Proposals
  - Randomized Prim
  - Selective Search
- Selective Search is the fastest and provides best regions

R-CNN: Selective Search I

- Graph based segmentation (Felzenszwalb and Huttenlocher method)
  - cannot be used in this form, because one object is covered with multiple segments, moreover regions for occluded objects will not be covered
- Idea: oversegment it and apply scaled similarity based merging

Input image

Segmented image

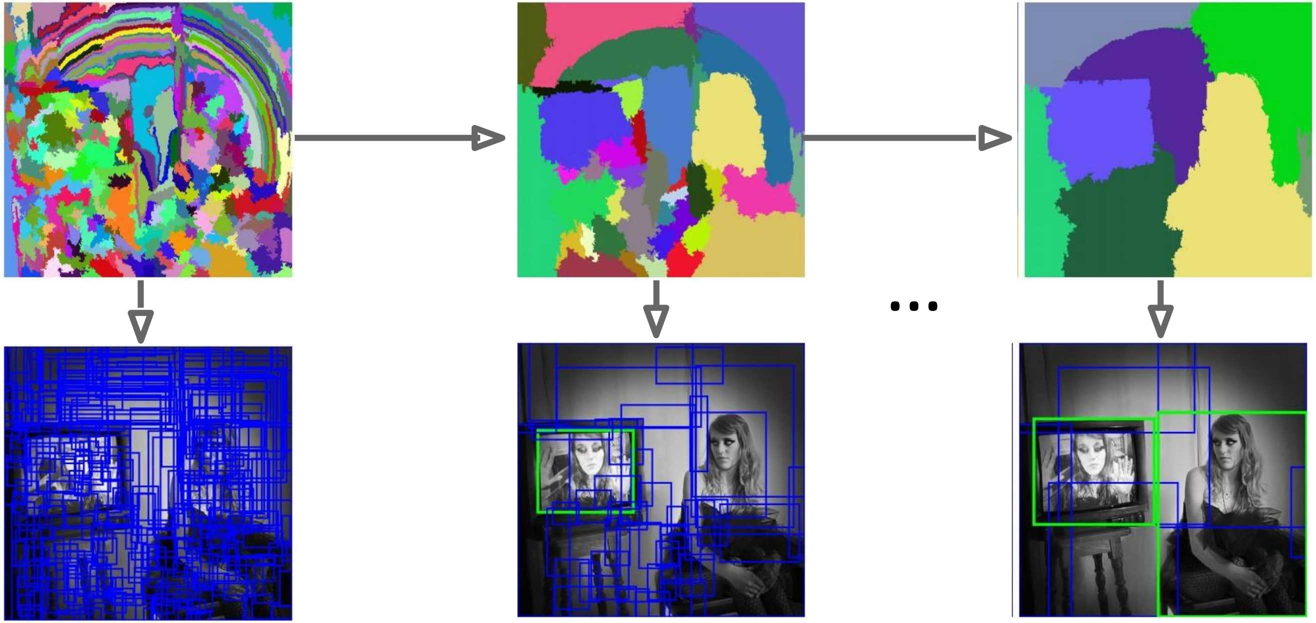
Oversegmented image

https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/

R-CNN: Selective Search II

Step-by-step merging regions at multiple scales based on similarities

Convert regions to boxes



Original fine scale

Step one merging

Step *n* merging

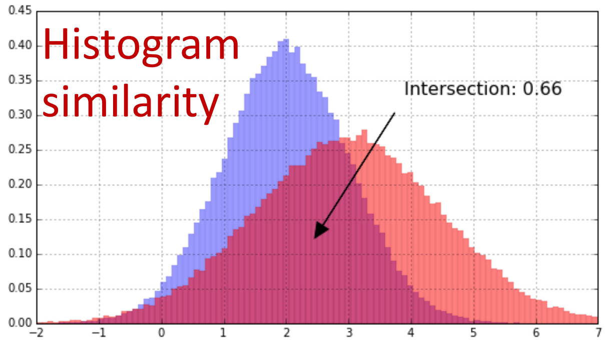
https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/

Similarity measures I

Color Similarity

- Generate color histogram of each segment (descriptor)
  - 25 bins/ color channels
  - Descriptor vector (c_i^k)size: 3x25=75
- Calculate histogram similarity for each region pair

$$s_{color}(r_i, r_j) = \sum_{k=1}^{75} \min(c_i^k, c_j^k)$$



c_i^k is the histogram value for the k[th] bin in color descriptor

Texture Similarity

- Texture features: Gaussian derivatives at 8 orientations in each pixel
  - 10 bins/color channels
  - Descriptor vector (t_i^k)size: 3x10x8=240
- Each region will have a texture histogram
- Calculate histogram similarity for each region pair

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{240} \min(t_i^k, t_j^k)$$

t_i^k is the histogram value for the k[th] bin in texture descriptor

https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/

Similarity measures II

Size Similarity

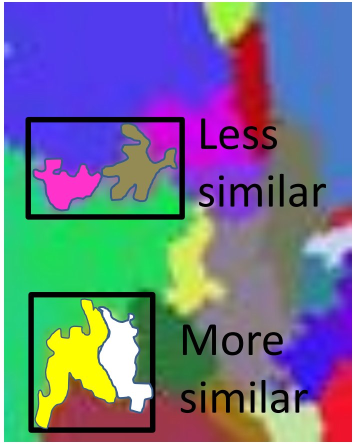
- Helps merging the smaller sized objects
- Since we do bottom up merging, the small segments will be merged first, because their size similarity score is higher

$$s_{size}(r_i, r_j) = 1 - \frac{size(r_i) + size(r_j)}{size(image)}$$

$size(image)$ is the size of the entire image in pixels

Shape Similarity

- Measures how well two regions are fit
  - How close they are
  - How large is the overlap



$$s_{fill}(r_i, r_j) = $$
$$= 1 - \frac{size(BB_{ij}) - size(r_i) - size(r_j)}{size(image)}$$

$size(BB_{ij})$ is the size of the bounding box of r_i and r_j

Similarity measures III

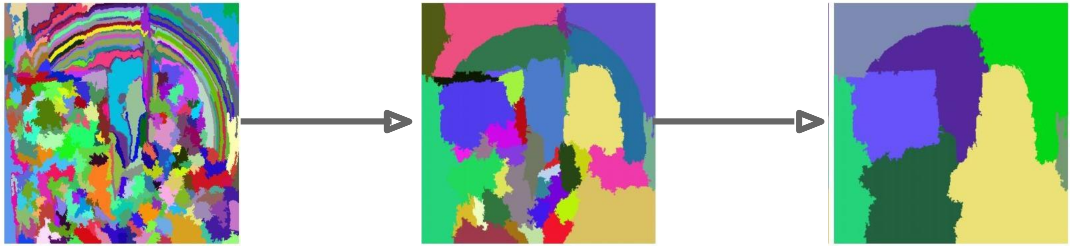
Final Similarity

- Linear combination of the four similarities

$$s_{final}(r_i, r_j) = a_1 s_{color}(r_i, r_j) + a_2 s_{texture}(r_i, r_j) + a_3 s_{shap}(r_i, r_j) + a_4 s_{fill}(r_i, r_j)$$

List or proposed region

1. Initial oversegmentation
2. Calculation the similarities
3. Merge the similar regions
4. The formed regions are added to the region list (*this ensures that there will be smaller and larger regions in the list as well*)
5. Goto 2



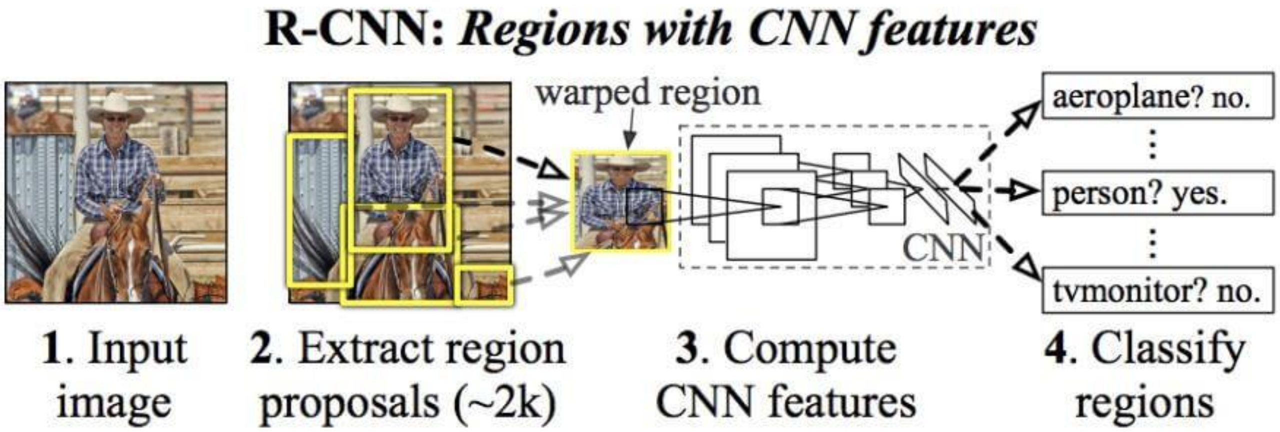
https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/

Proposed regions

- Few hundreds or few thousand boxes
- Includes all the objects with high probability
- Number of the boxes are much smaller than with brute force method
- C and python functions exist

https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/

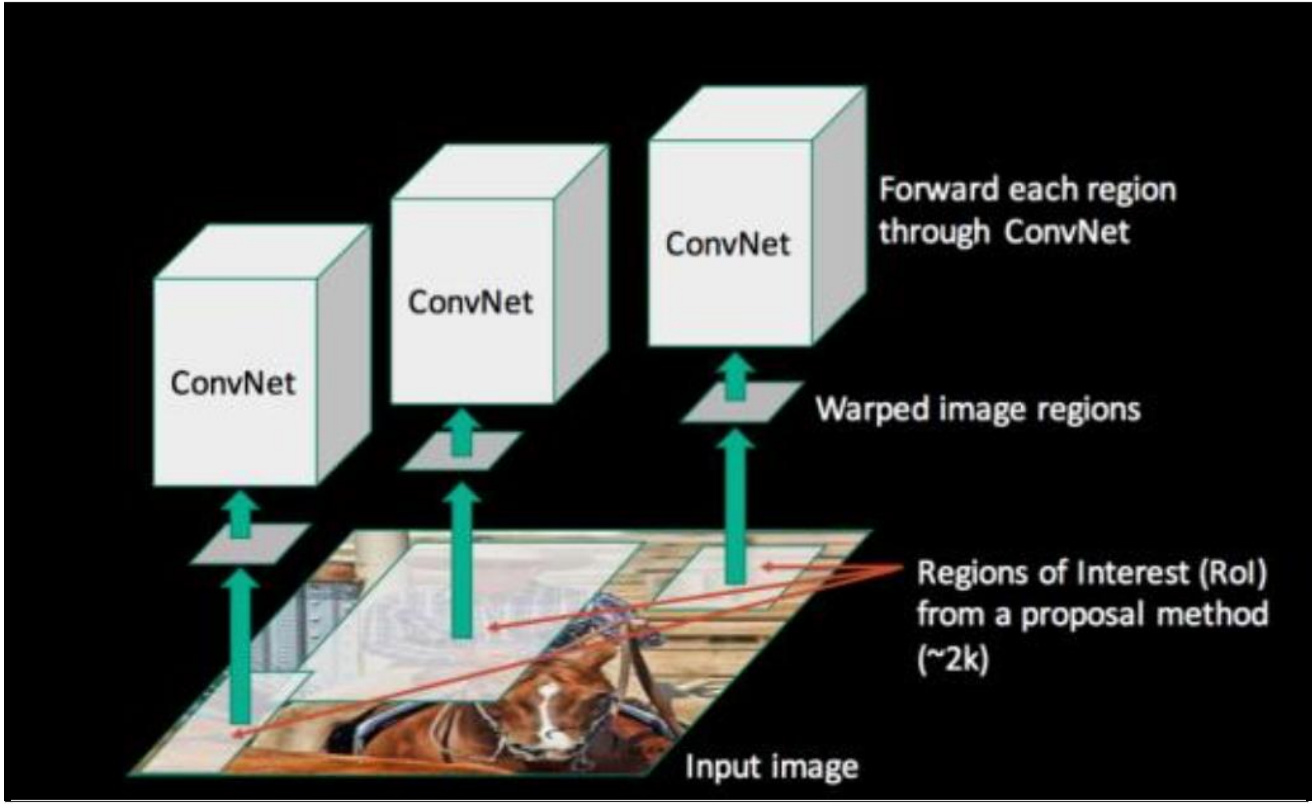
The R-CNN algorithm



1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

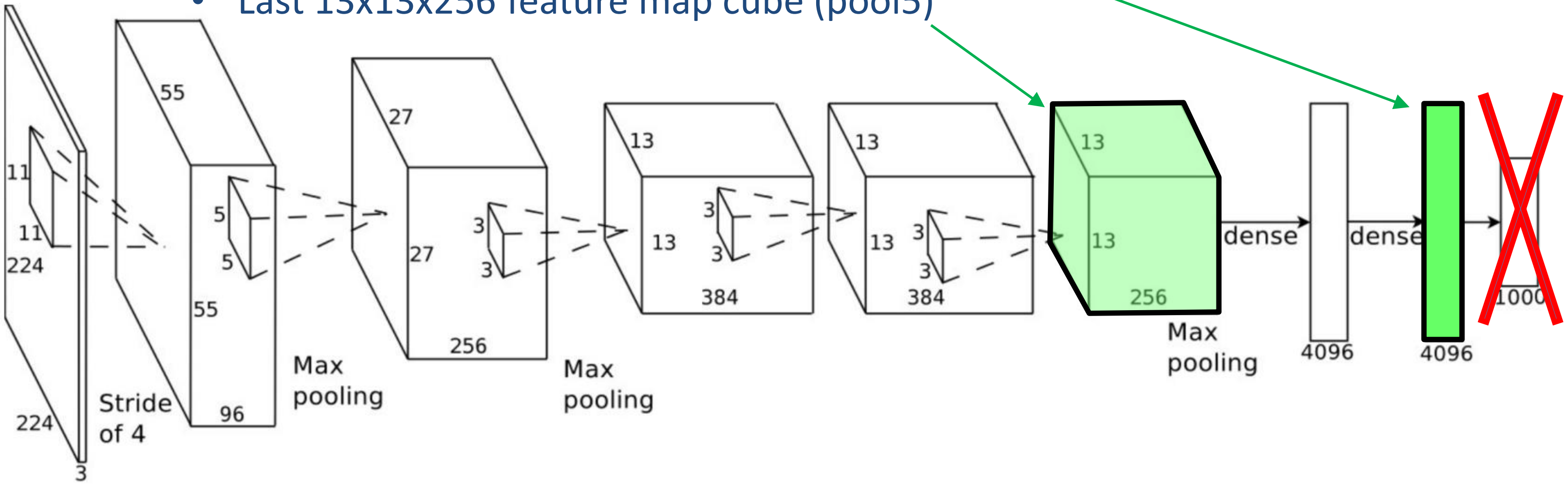
Computing the features of the regions

- Cut the regions one after the other
- Resize (warp) the regions to the input size of the ConvNet
- Calculate the features of the individual regions

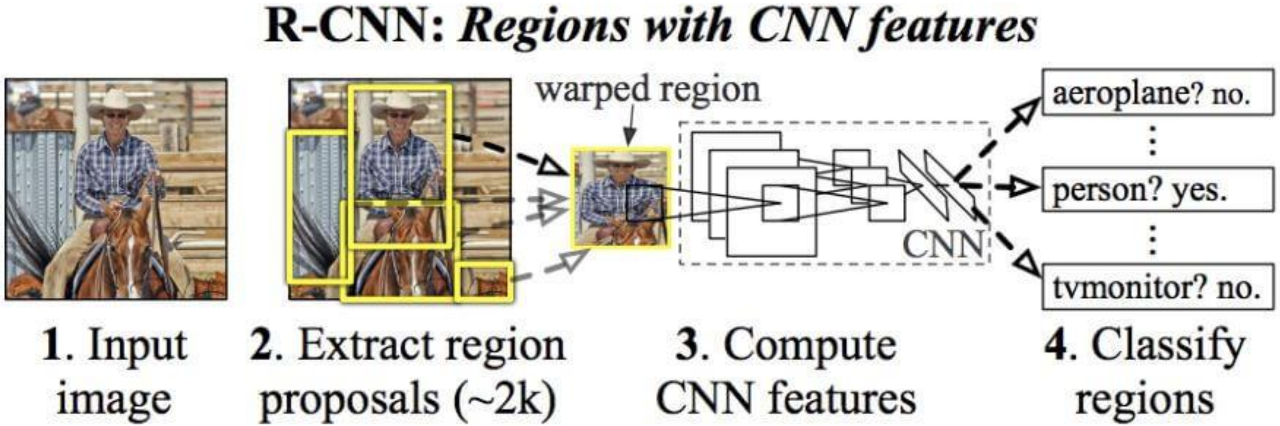


Convolution network

- Pre-trained AlexNet, later VGGNet
- The decision maker SoftMax layer was cut
  - Outputs:
    - 4096 long feature vectors from each region
    - Last 13x13x256 feature map cube (pool5)



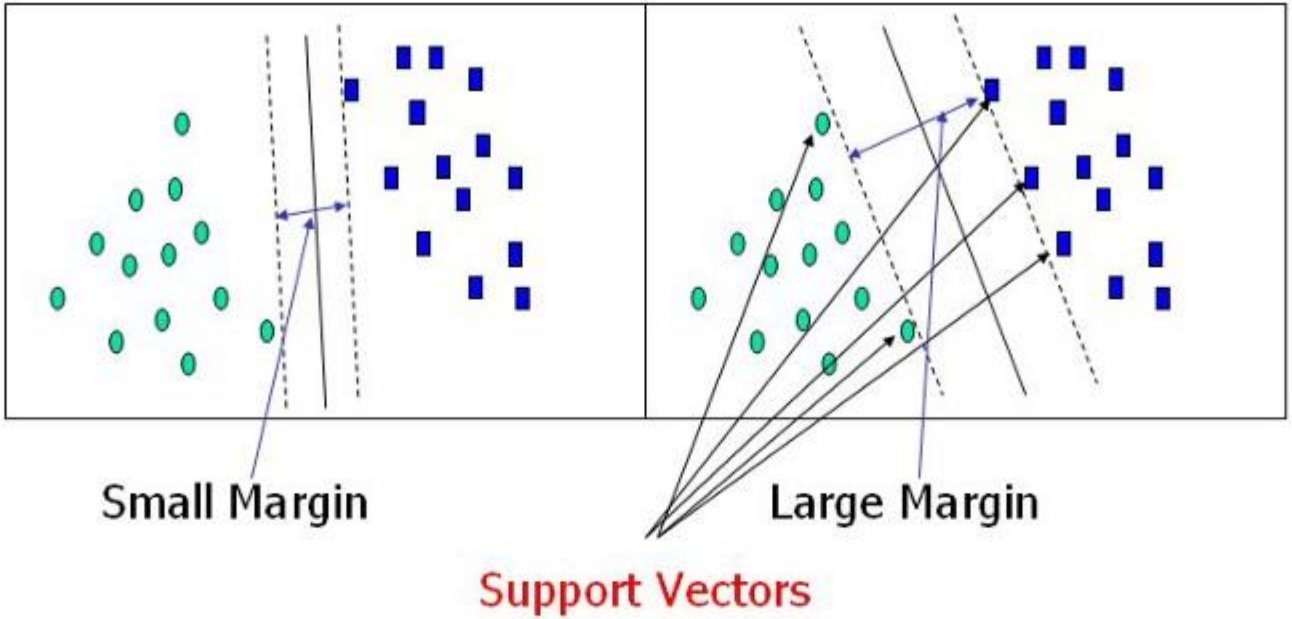
The R-CNN algorithm



1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

Linear Support Vector Machine

- **<u>Idea:</u>** Separate the data point in the data space with a boundary surface (hyperplane) with maximum margin
- Vectors pointing to the data points touching the margins are the support vectors
- The parameters of the optimal hyperplane is calculated with regression



- Similar to single layer perceptron, but optimized for maximum margin

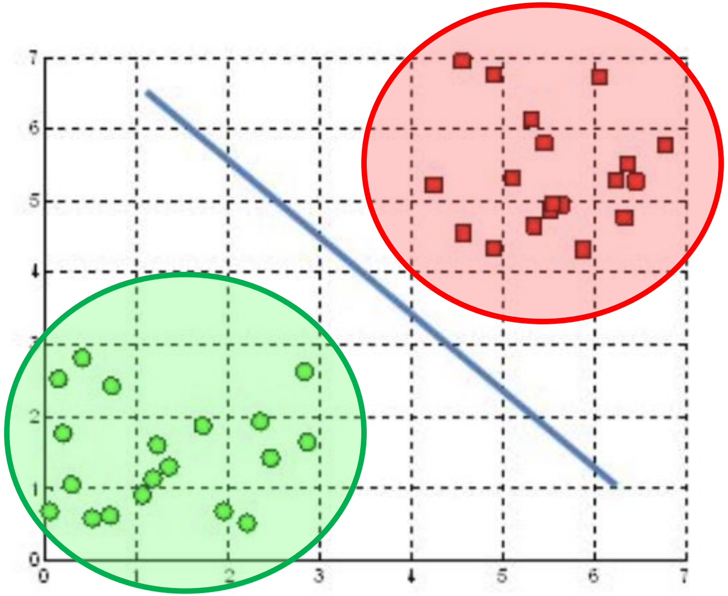
https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

Why SVM?

- Why not use simple the classification output of the AlexNet?
- During the training, the AlexNet/VGGNet is not trained
- Only SVM is trained
- The number of category is much smaller
  - Designed for 20-200 categories rather than 1000

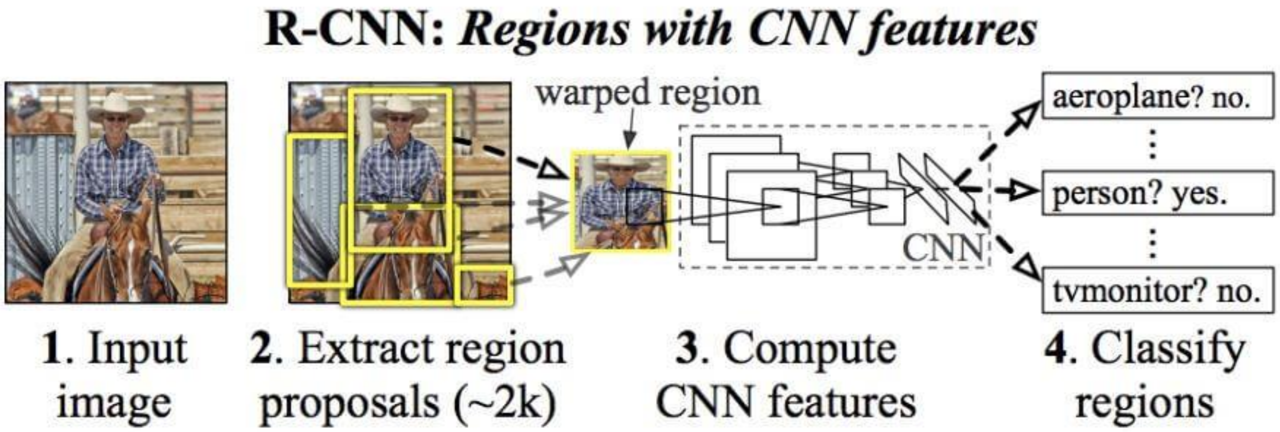
Decision with SVM

- As many separate SVM as many category we have



The result: Each region is categorized in every image classes.

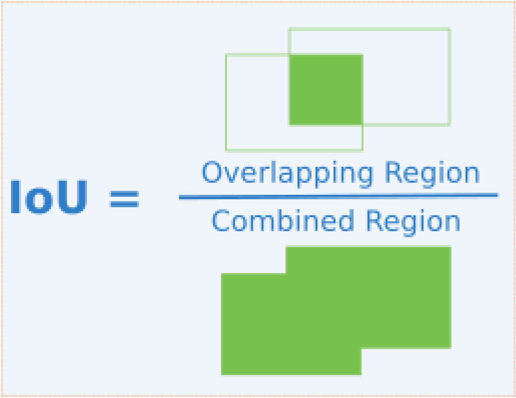
The R-CNN algorithm



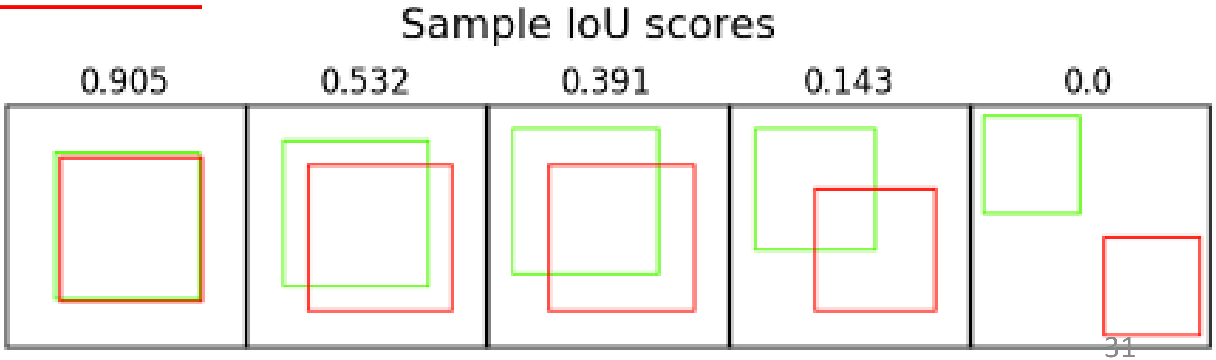
1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

Ranking, selecting, merging

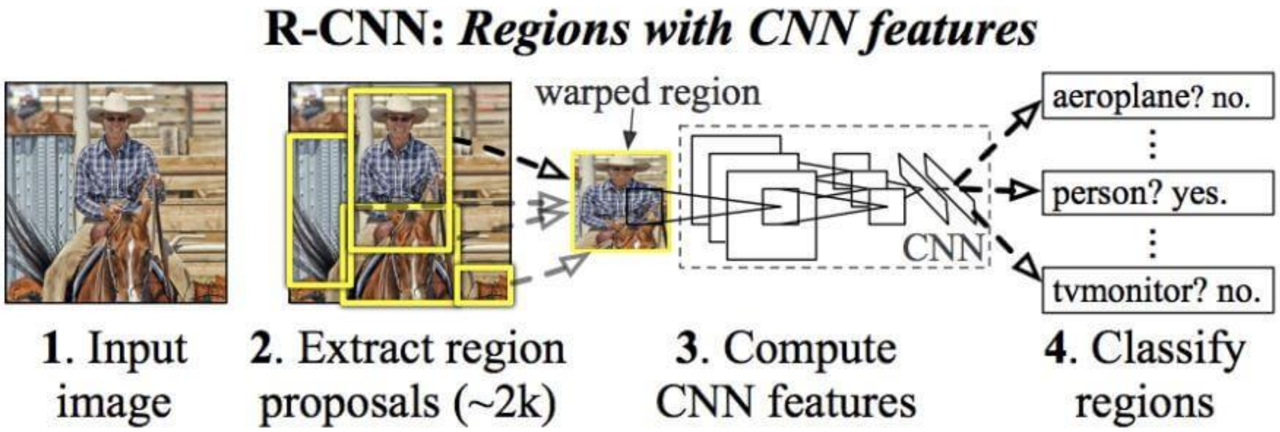
- Greedy non-maximum suppression
  - Regions with low classification probabilities are rejected
  - Regions with high Intersection over Union values (within the same category) are merged



- *Result:* **localized and classified object**

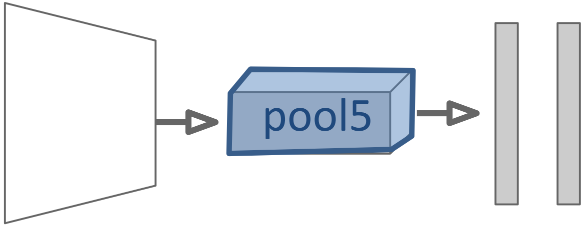


The R-CNN algorithm



1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

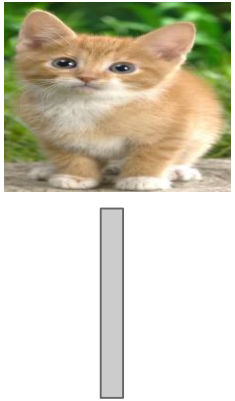
Bounding Box Regression

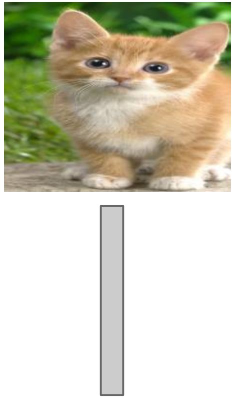


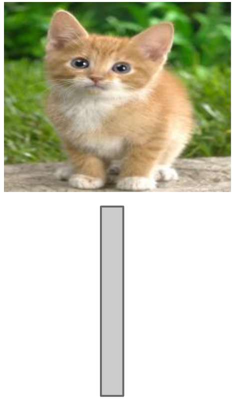
- Linear regression model
- One per object category
- Input: last feature map cube of the conv net (pool5)
- Output: size and position modification to the bounding box:
  - dx, dy, dw, dh

Training image regions

Input:
Cached feature map
cube (pool5)







Regression targets:
(dx, dy, dw, dh)
(normalized)

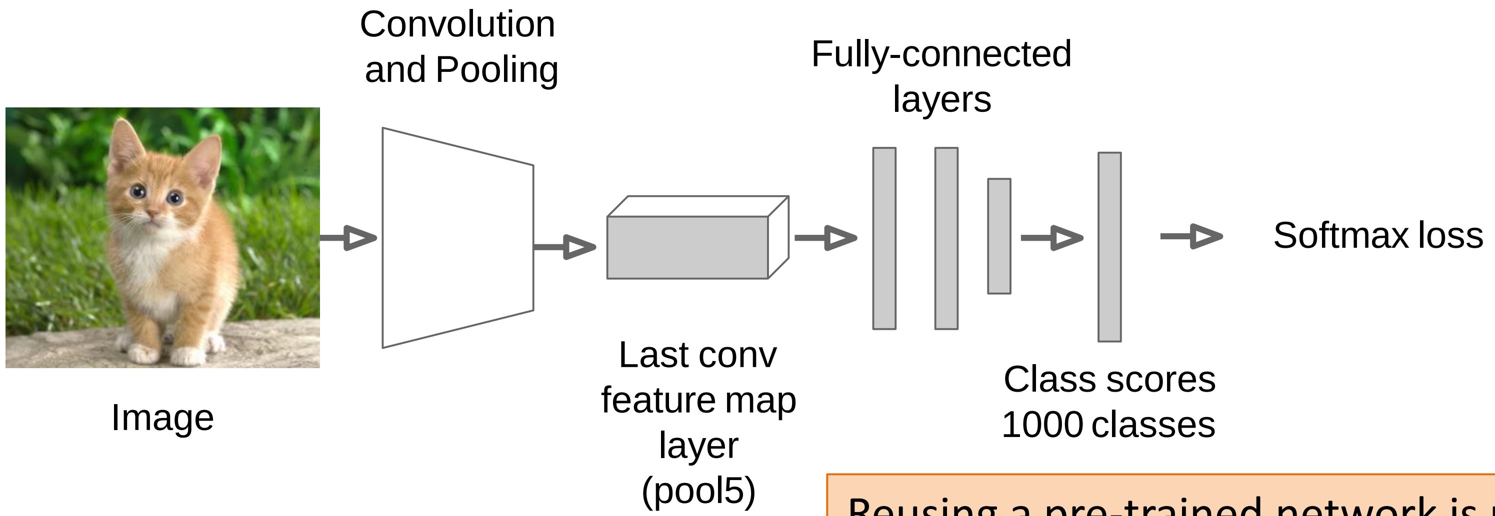
(0, 0, 0, 0)
Proposal is good

(.25, 0, 0, 0)
Proposal too
far to left

(0, 0, -0.125, 0)
Proposal too
wide

R-CNN Training

Step 1: Take a pretrained Convolutional Neural Network (e.g. AlexNet)



Reusing a pre-trained network is useful, if there is not enough data to train or if it provides good enough result. Fine tuning typically needed!

Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

R-CNN Training

Step 2: Extract features

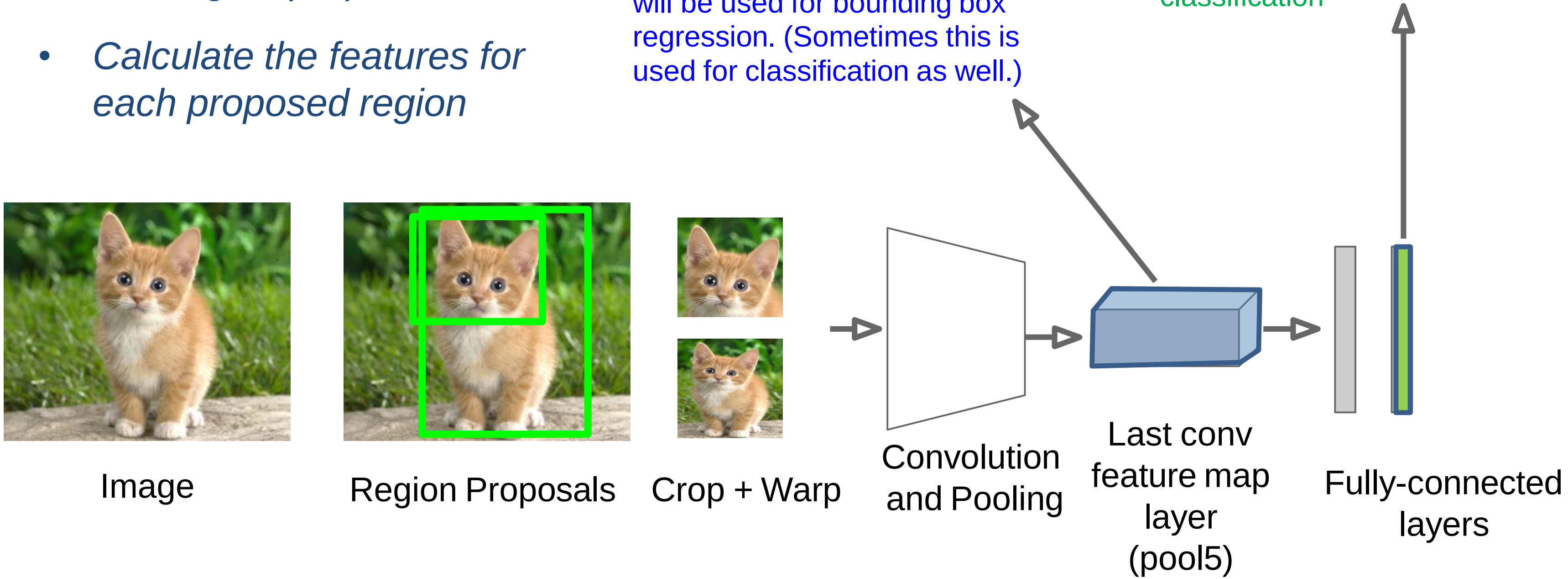
- *Go through the data base*
- *Use region proposal*
- *Calculate the features for each proposed region*

Save the feature cube to disk!

This feature cube describes the relative position information, and will be used for bounding box regression. (Sometimes this is used for classification as well.)

Save the feature vector to disk!

This feature vector describes the content, and will be used for classification



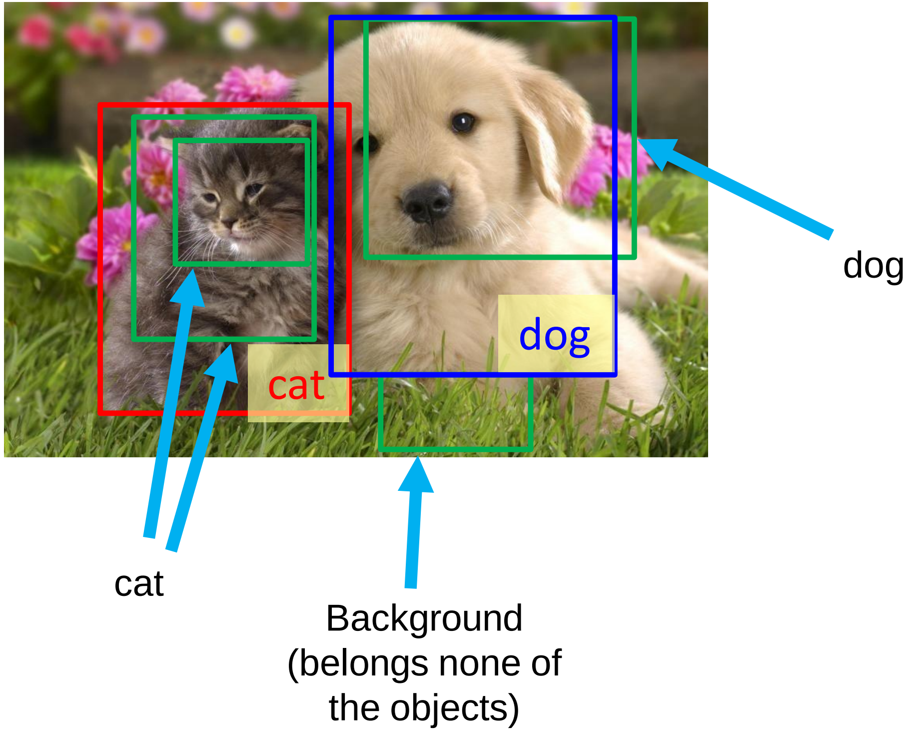
Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

R-CNN Training

Step 3: Identify which proposed region belongs to which object class

Based on the annotated image

Proposed region overlaps with the annotated image segment? (IoU)

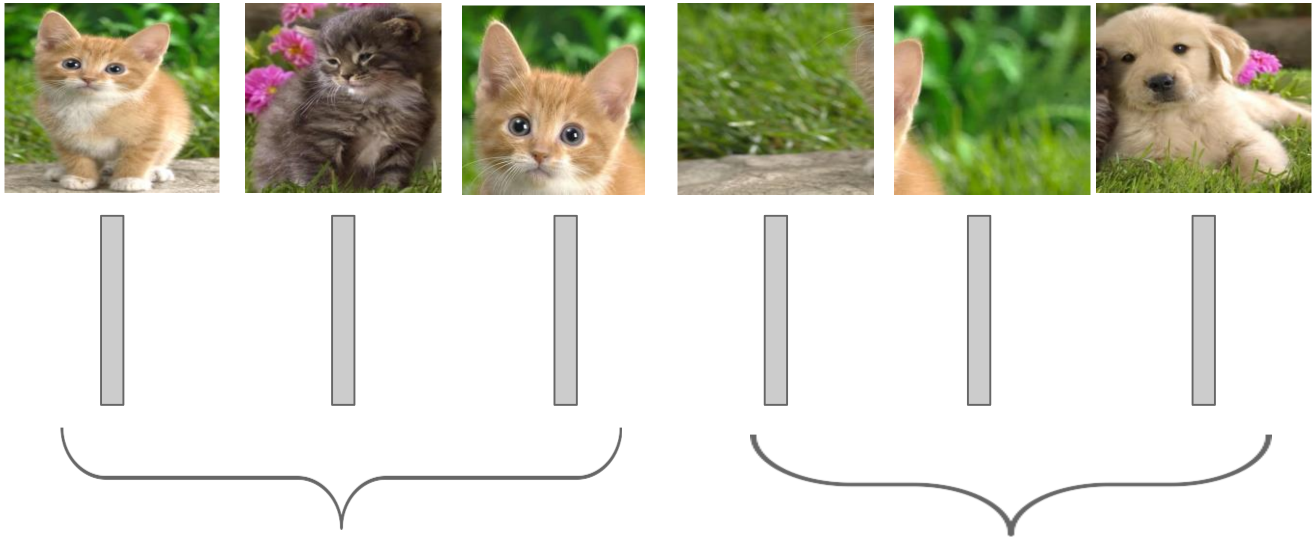


R-CNN Training

Step 4: Train one SVM per class to classify region features

Training image regions

Cached region
features vectors



Positive samples for cat SVM

Negative samples for cat SVM

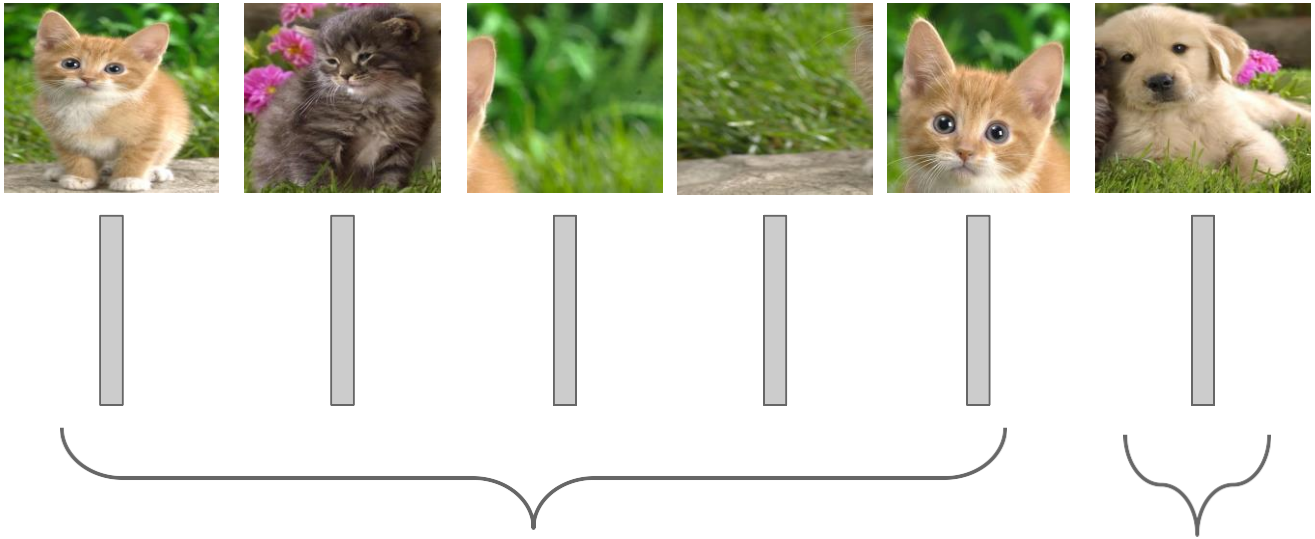
Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

R-CNN Training

Step 4: Train one SVM per class to classify region features

Training image regions

Cached region
features vectors



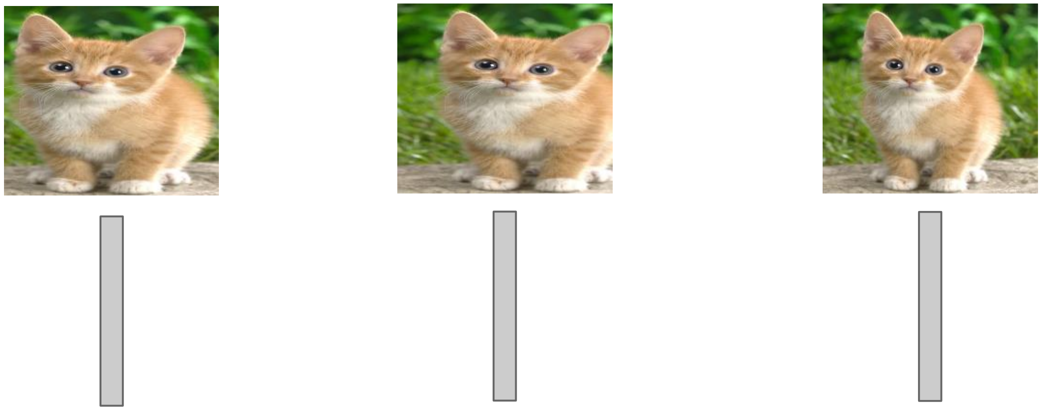
Negative samples for dog SVM

Positive samples for dog SVM

Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

R-CNN Training

Step 5 (bbox regression): For each class, train a linear regression model to map from cached features cubes to offsets/size of the boxes to fix “slightly wrong” position proposals



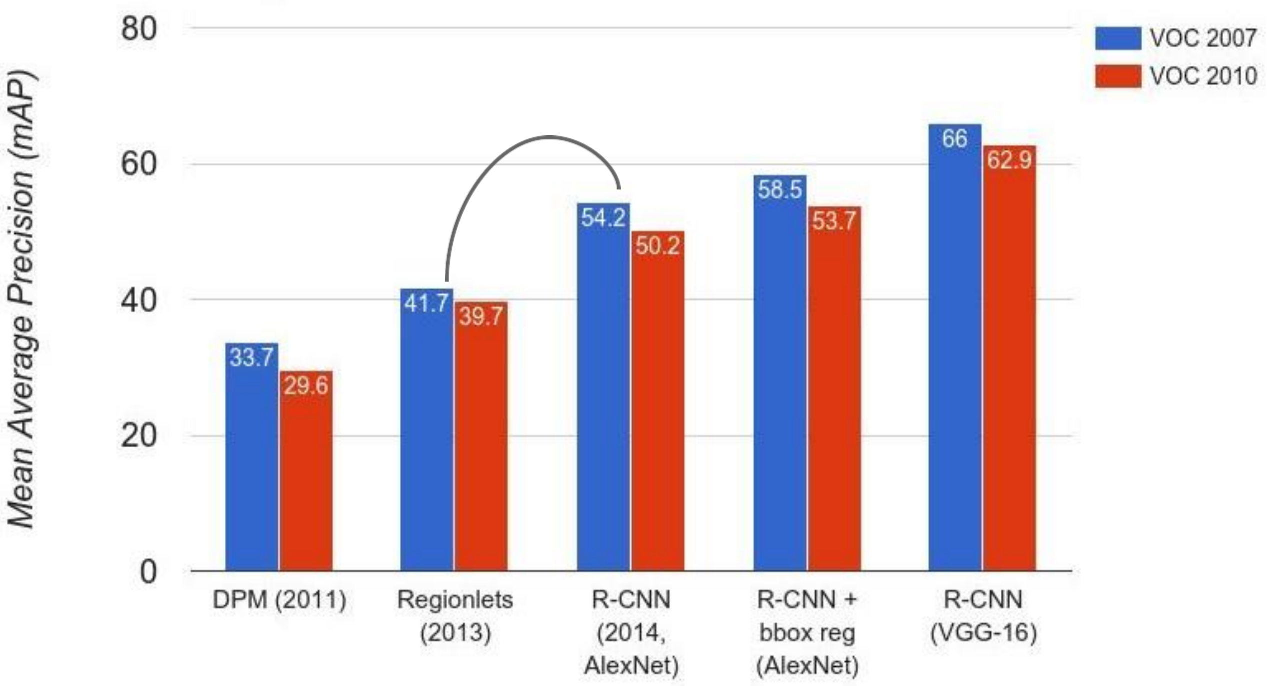
Training image regions

Cached region feature cube (pool5)

Regression targets (dx, dy, dw, dh) Normalized coordinates	(0, 0, 0, 0) Proposal is good	(.25, 0, 0, 0) Proposal too far to left	(0, 0, -0.125, 0) Proposal too wide

R-CNN Results

Big improvement (~25%)
compared to pre-CNN methods

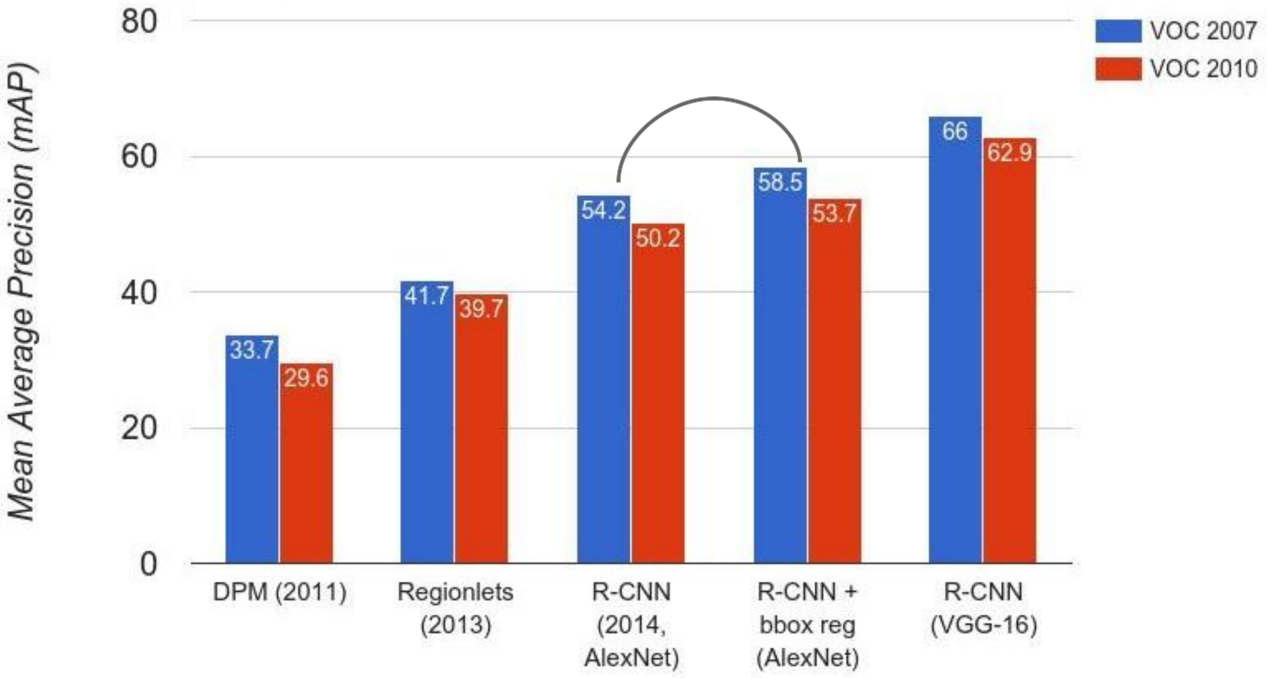


Wang et al, “Regionlets for Generic Object Detection”, ICCV 2013

Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

R-CNN Results

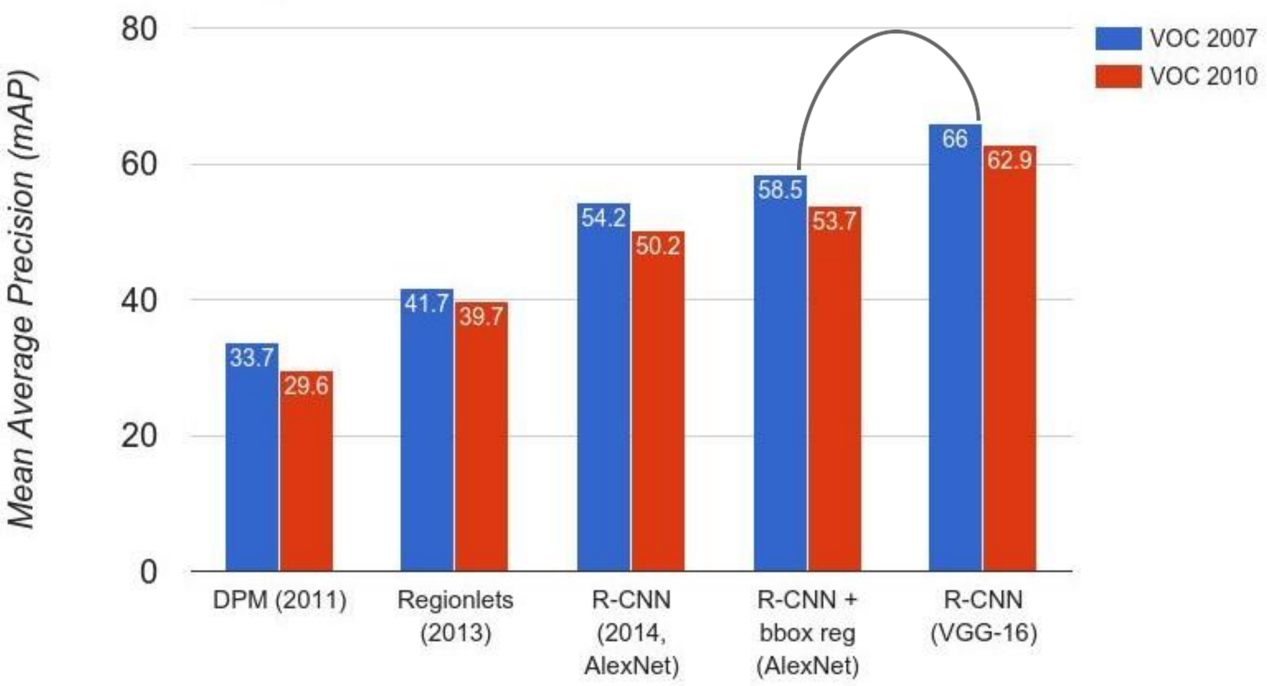
Bounding box regression helps a bit



Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

R-CNN Results

Features from a deeper network help a lot



Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li

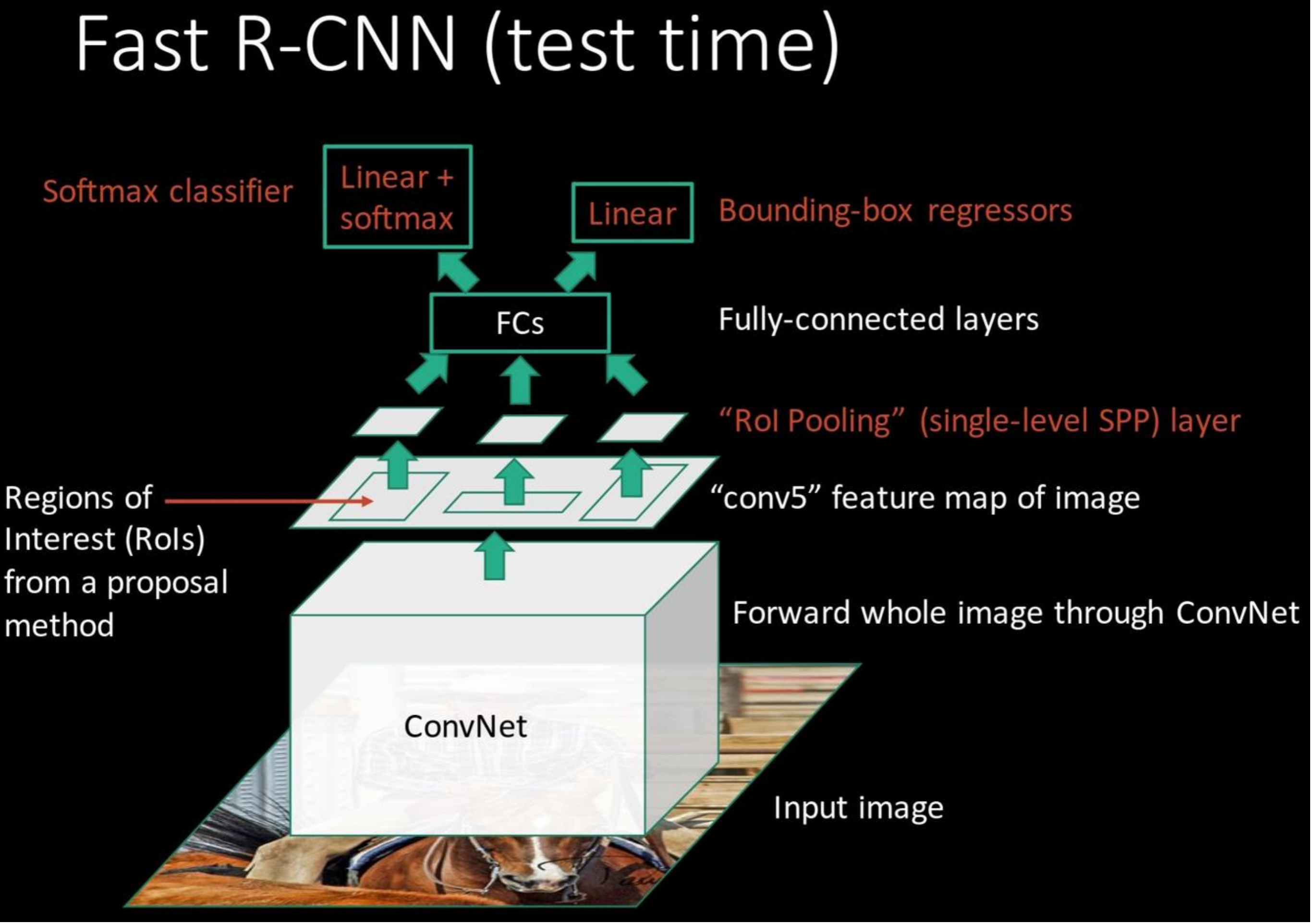
R-CNN Problems

1. Slow at test-time: need to run full forward pass of CNN for each region proposal
   - Recalculate the features again-and-again in the overlapping regions

2. SVMs and bbox regressors are post-hoc:
   - CNN features not updated in response to SVMs and regressors

3. Complex multistage training pipeline
   - Calculate the features for all the regions for all the training image first
   - Then train for SVM and bbox regressor separately

Slide Credits: Justin Johnson, Andrej Karpathy, Fei-Fei Li



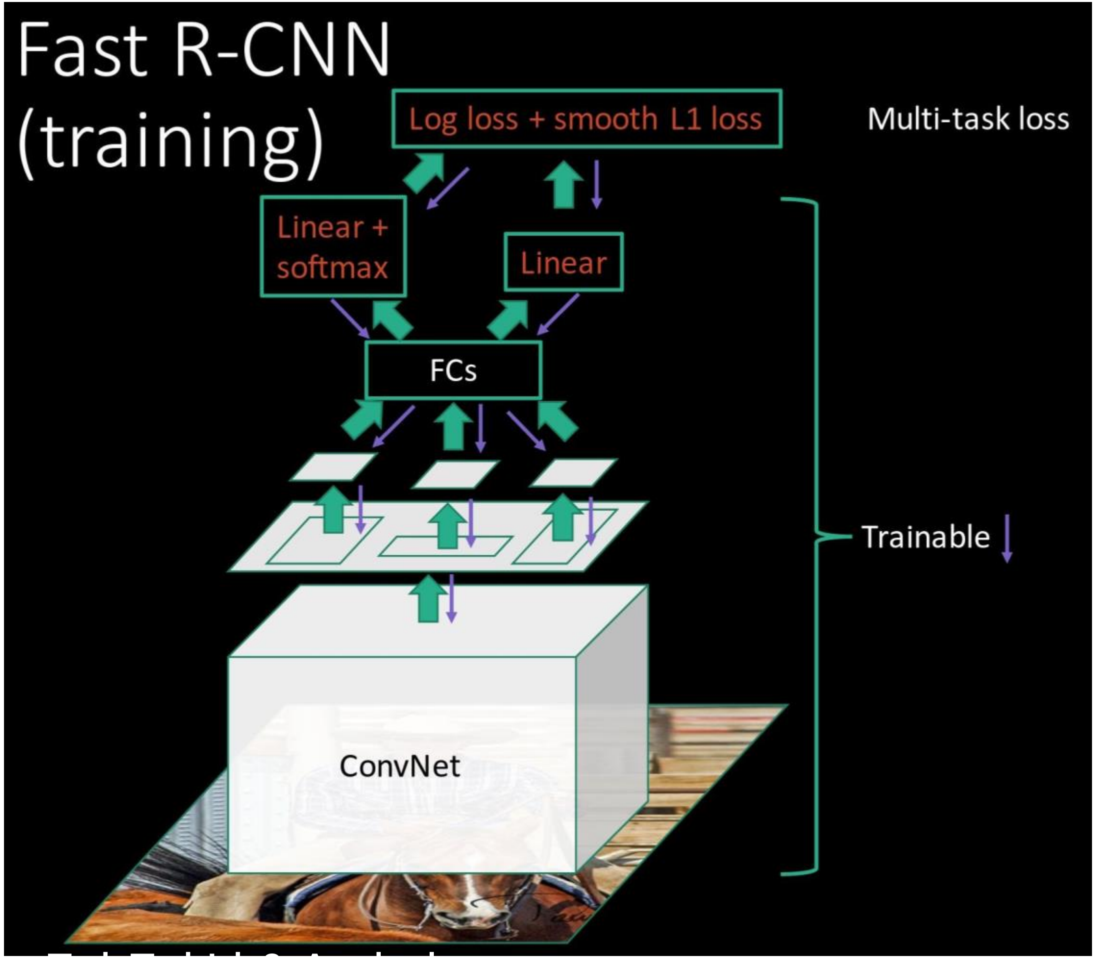
R-CNN Problem #1: Slow at test-time due to independent forward passes of the CNN

Solution: Share computation of convolutional layers between proposals for an image

Girschick, "Fast R-CNN", ICCV 2015

Slide credit: Ross Girschick

https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46



R-CNN Problem #2:
Post-hoc training: CNN not updated in response to final classifiers and regressors

R-CNN Problem #3:
Complex training pipeline

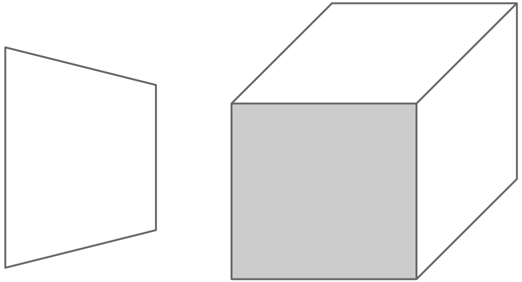
Solution:
Just train the whole system end-to-end all at once!

Slide credit: Ross Girschick

Fast R-CNN: Region of Interest Pooling

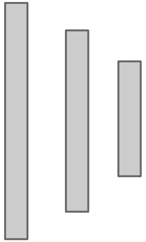
Hi-res input image:
3 x 800 x 600
with region
proposal

Convolution
and Pooling



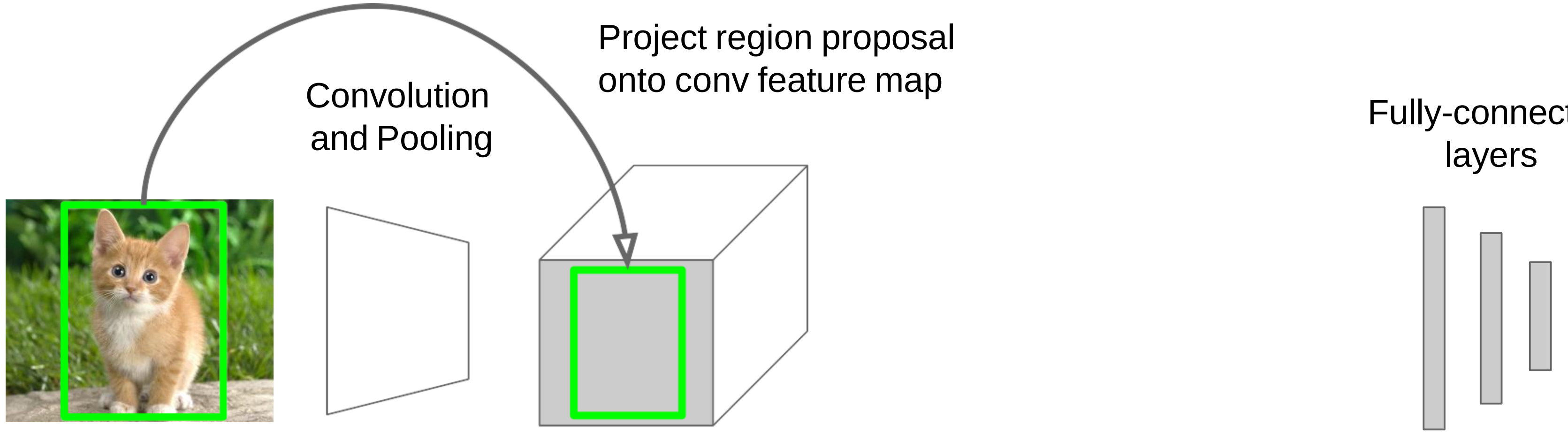
Hi-res conv features:
C x H x W
with region proposal

Fully-connected
layers



Problem: Fully-connected
layers expect low-res conv
features: C x h x w

Fast R-CNN: Region of Interest Pooling

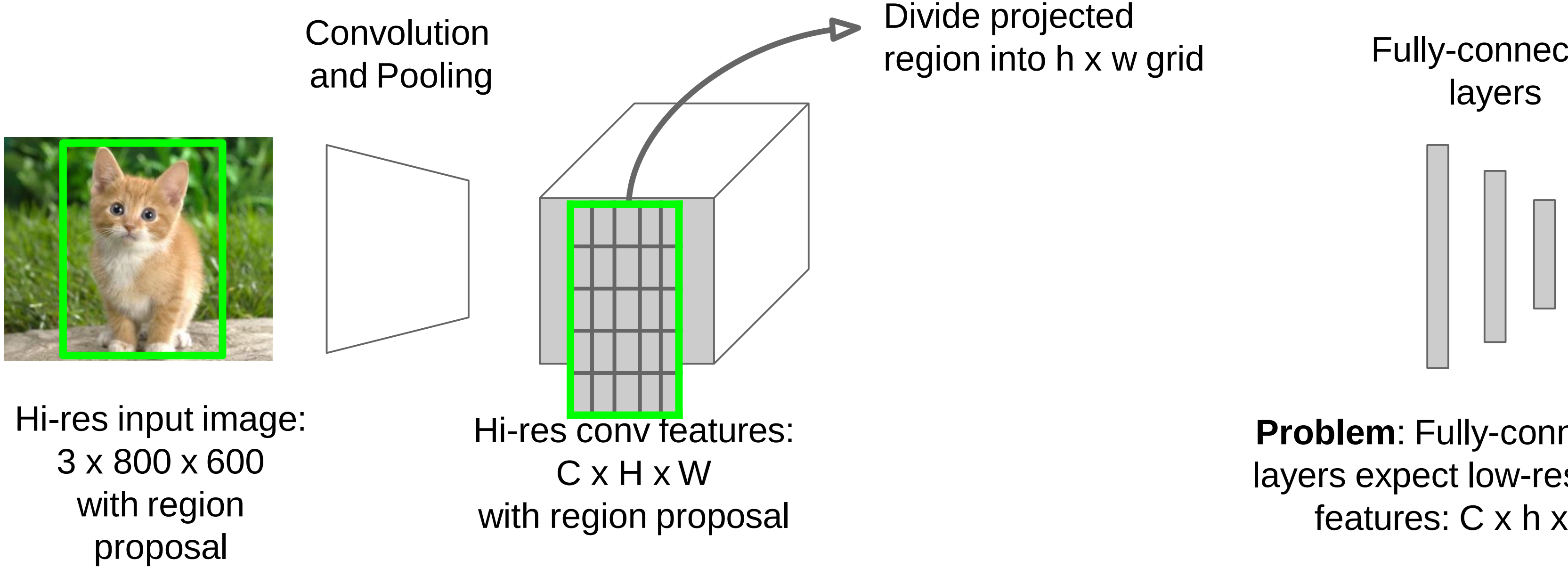


Hi-res input image:
3 x 800 x 600
with region
proposal

Hi-res conv features:
C x H x W
with region proposal

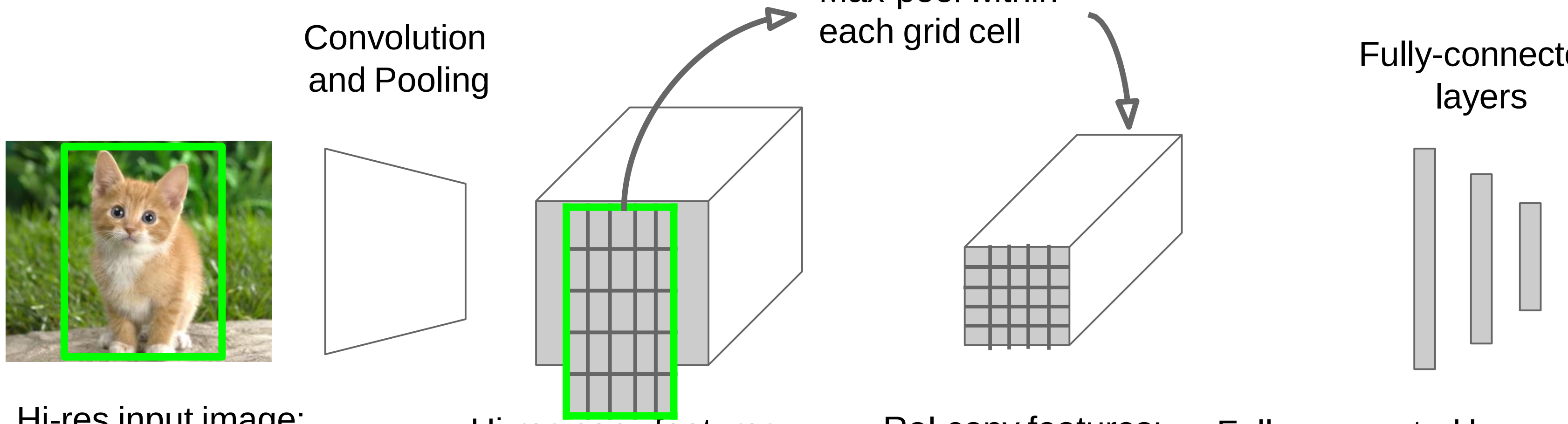
Problem: Fully-connected layers expect low-res conv features: C x h x w

Fast R-CNN: Region of Interest Pooling

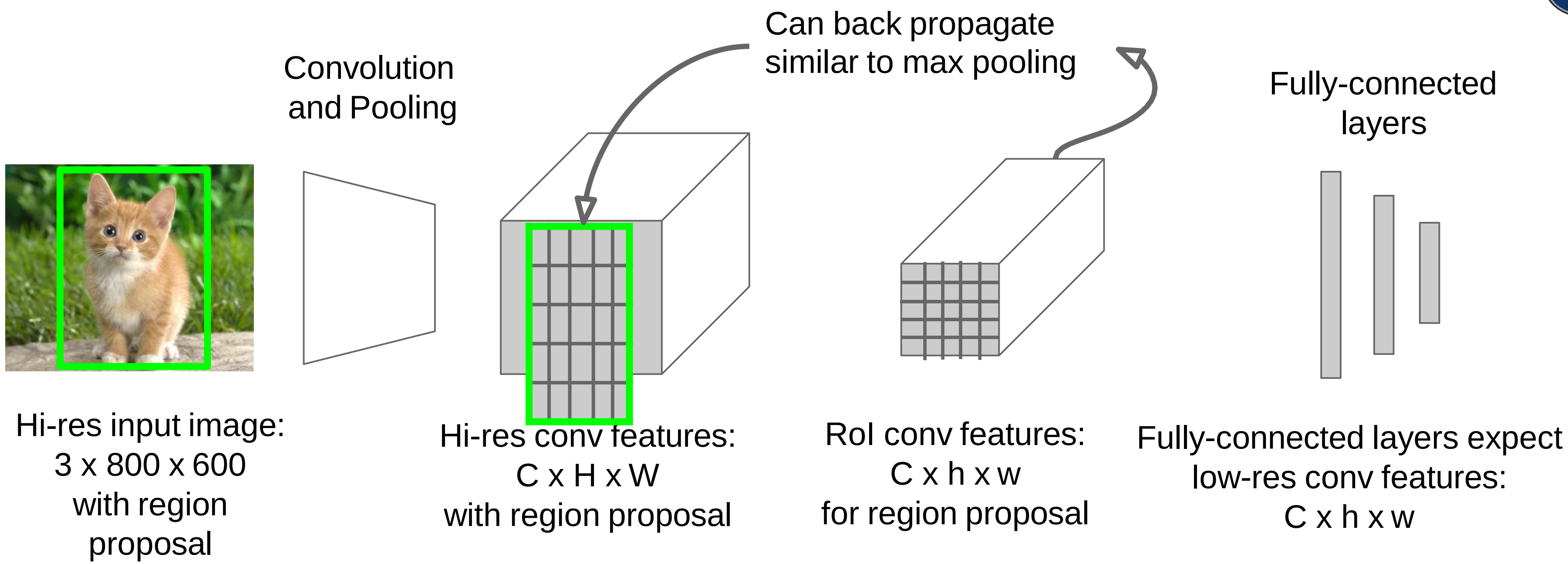


Problem: Fully-connected layers expect low-res conv features: C x h x w

Fast R-CNN: Region of Interest Pooling



Fast R-CNN: Region of Interest Pooling



Instead of SVM, a SoftMax layer makes the decision at Fast R-CNN.

Fast R-CNN Results

		R-CNN	Fast R-CNN
Faster!	Training Time:	84 hours	**9.5 hours**
	(Speedup)	1x	**8.8x**
FASTER!	Test time per image	47 seconds	**0.32 seconds**
	(Speedup)	1x	**146x**
Better!	mAP (VOC 2007)	66.0	**66.9**

Using VGG-16 CNN on Pascal VOC 2007 dataset

Fast R-CNN Results

		R-CNN	Fast R-CNN
Faster!	Training Time:	84 hours	**9.5 hours**
	(Speedup)	1x	**8.8x**
FASTER!	Test time per image	47 seconds	**0.32 seconds**
	(Speedup)	1x	**146x**
Better!	mAP (VOC 2007)	66.0	**66.9**

Using VGG-16 CNN on Pascal VOC 2007 dataset

Fast R-CNN Problem:

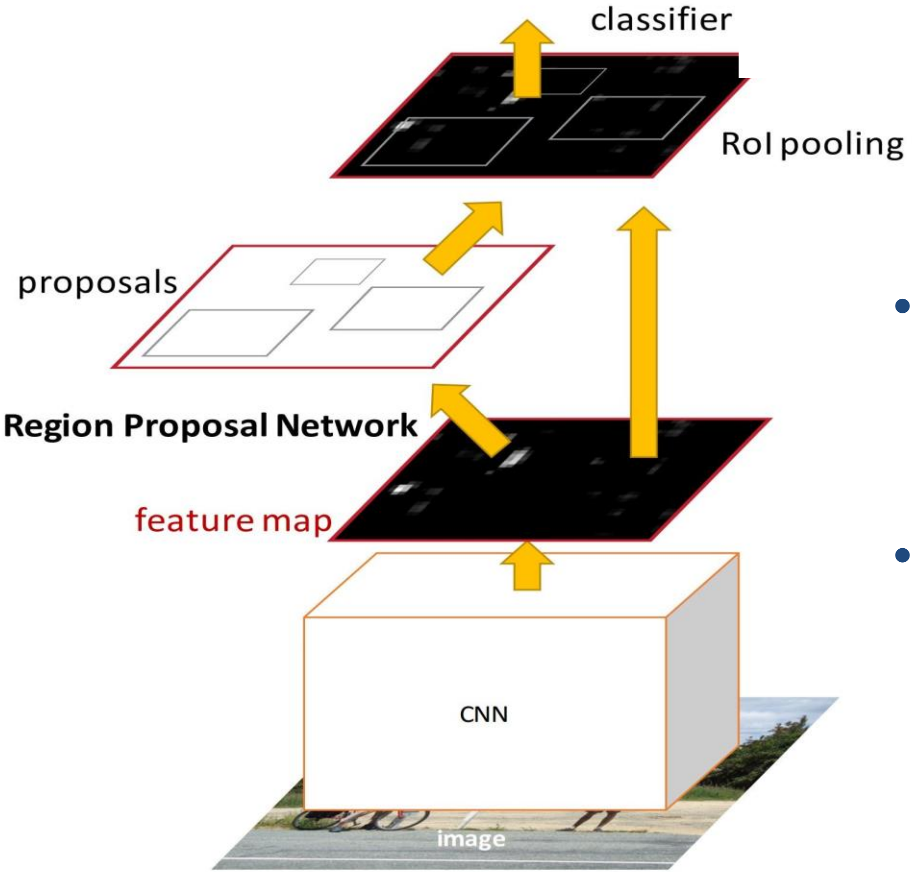
	R-CNN	**Fast R-CNN**
Test time per image without **Region Proposals**	47 seconds	**0.32 seconds**
(Speedup)	1x	**146x**
Test time per image with **Region Proposals**	50 seconds	**2 seconds**
(Speedup)	1x	**25x**

Fast R-CNN Problem Solution:

Test-time speeds don’t include region proposals
Just make the CNN do region proposals too!

	R-CNN	Fast R-CNN
Test time per image without **Region Proposals**	47 seconds	**0.32 seconds**
(Speedup)	1x	**146x**
Test time per image with **Region Proposals**	50 seconds	**2 seconds**
(Speedup)	1x	**25x**

Faster R-CNN:

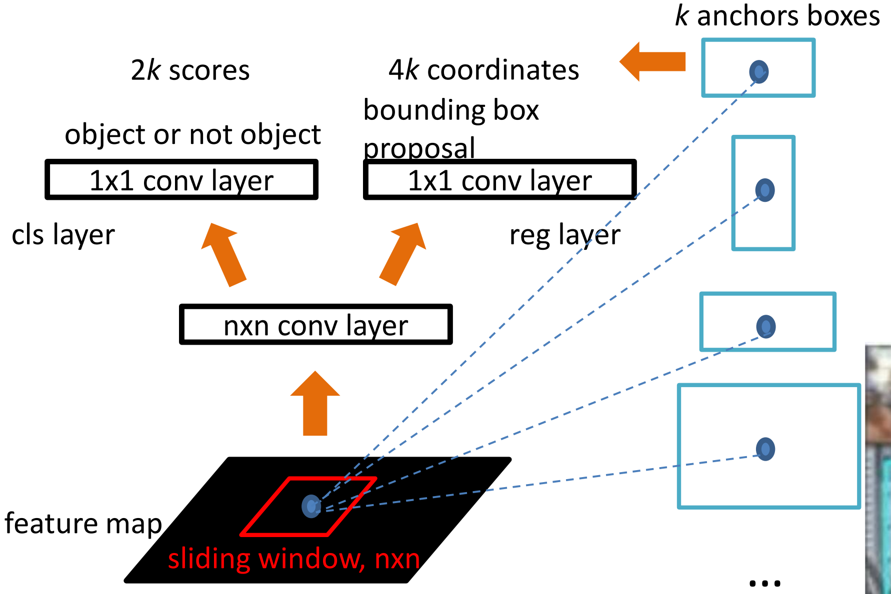


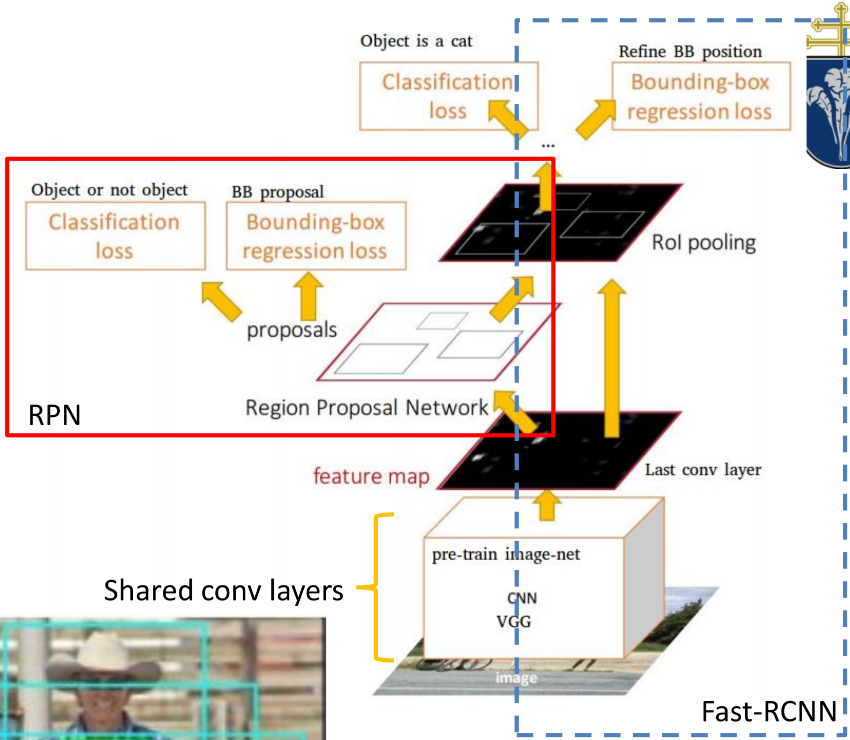
- Insert a **Region Proposal Network (RPN)** after the last convolutional layer
  - Reuse the CNN computation
- RPN trained to produce region proposals directly; no need for external region proposals!
- After RPN, use RoI Pooling and an upstream classifier and bbox regressor just like Fast R-CNN

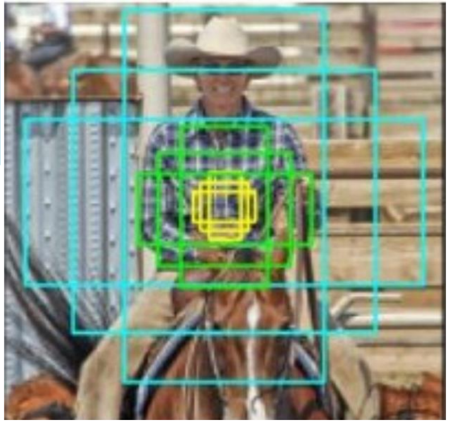
https://towardsdatascience.com/faster-rcnn-object-detection-f865e5ed7fc4

Faster-RCNN

Region Proposal Networks:







Anchors:
three rectangle
in three scales.

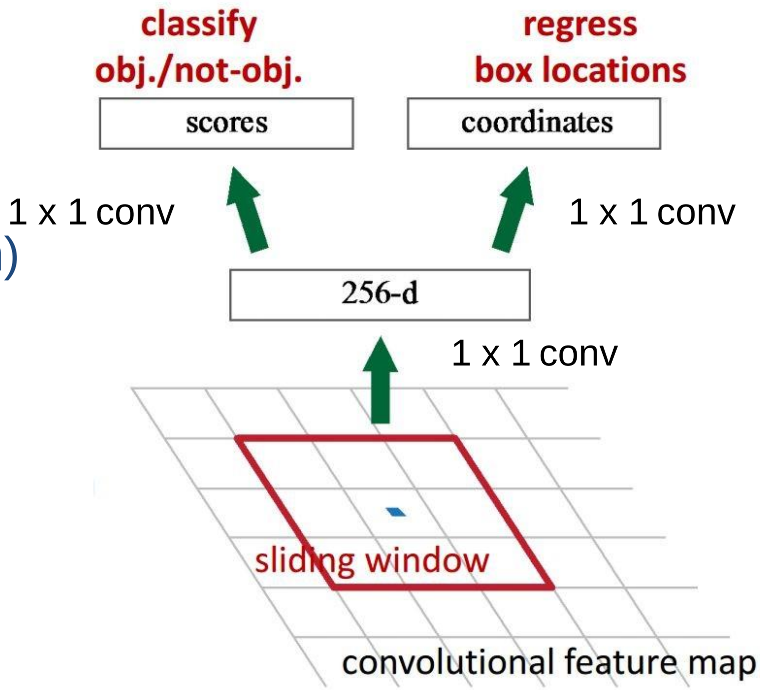
Faster R-CNN: Region Proposal Network

Slide a small window on the feature map
(very small computational effort per position)

Build a small network for:
- classifying object or not-object, (Binary decision)
- regressing bbox locations

Position of the sliding window provides localization information with reference to the image

Box regression provides finer localization information with reference to this sliding window



Slide credit: Kaiming He

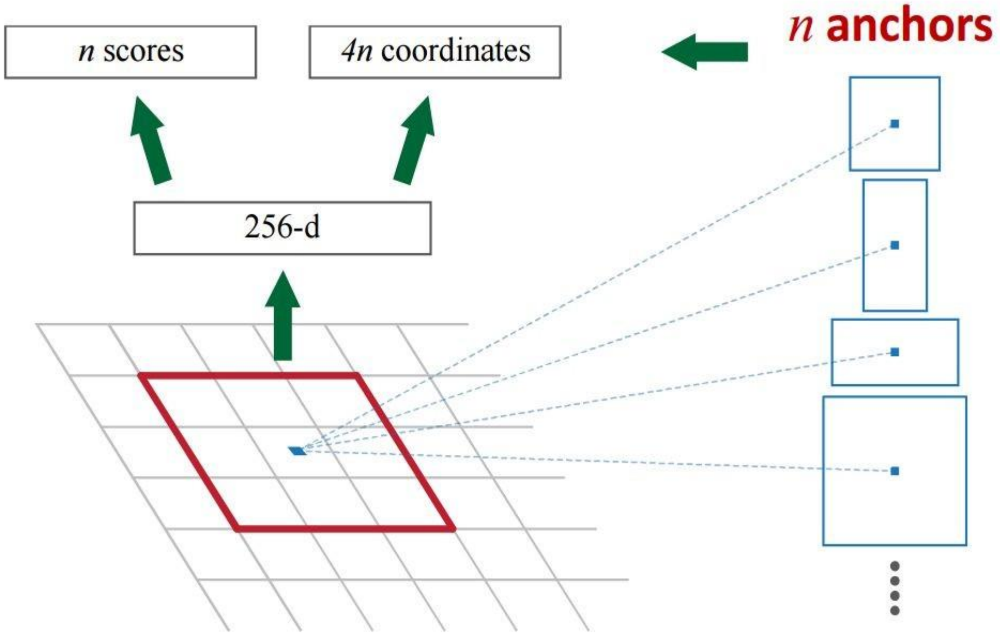
Faster R-CNN: Region Proposal Network

Use **N anchor boxes** at each location

Anchors are **translation invariant**: use the same ones at every location

Regression gives offsets from anchor boxes

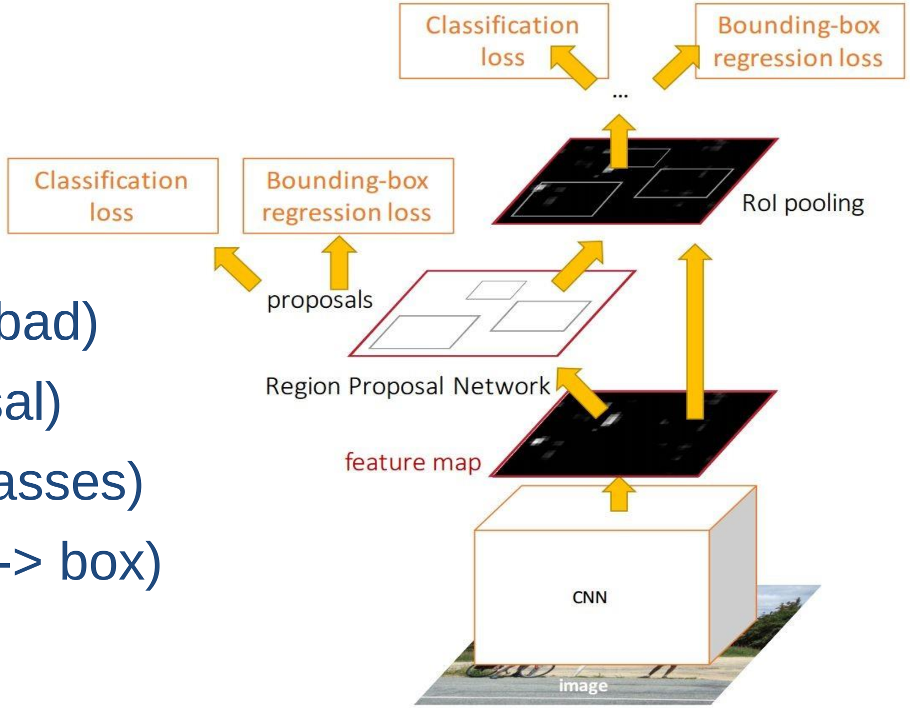
Classification gives the probability that each (regressed) anchor shows an object



Faster R-CNN: Training

One network, four losses

- RPN classification (anchor good / bad)
- RPN regression (anchor -> proposal)
- Fast R-CNN classification (over classes)
- Fast R-CNN regression (proposal -> box)



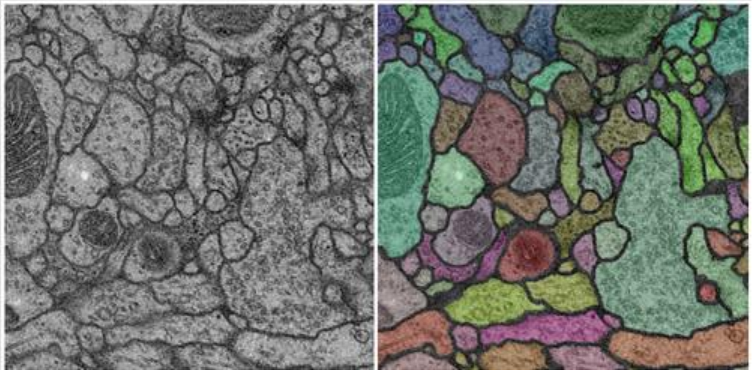
Slide credit: Ross Girschick

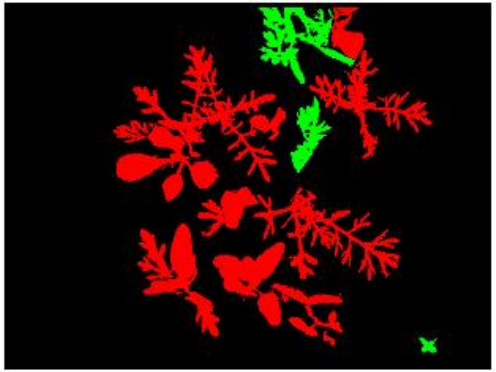
Faster R-CNN: Results

	R-CNN	**Fast R-CNN**	**Faster R-CNN**
Test time per image (with proposals)	50 seconds	2 seconds	**0.2 seconds**
(Speedup)	1x	25x	**250x**
mAP (VOC 2007)	66.0	**66.9**	**66.9**

Segmentation

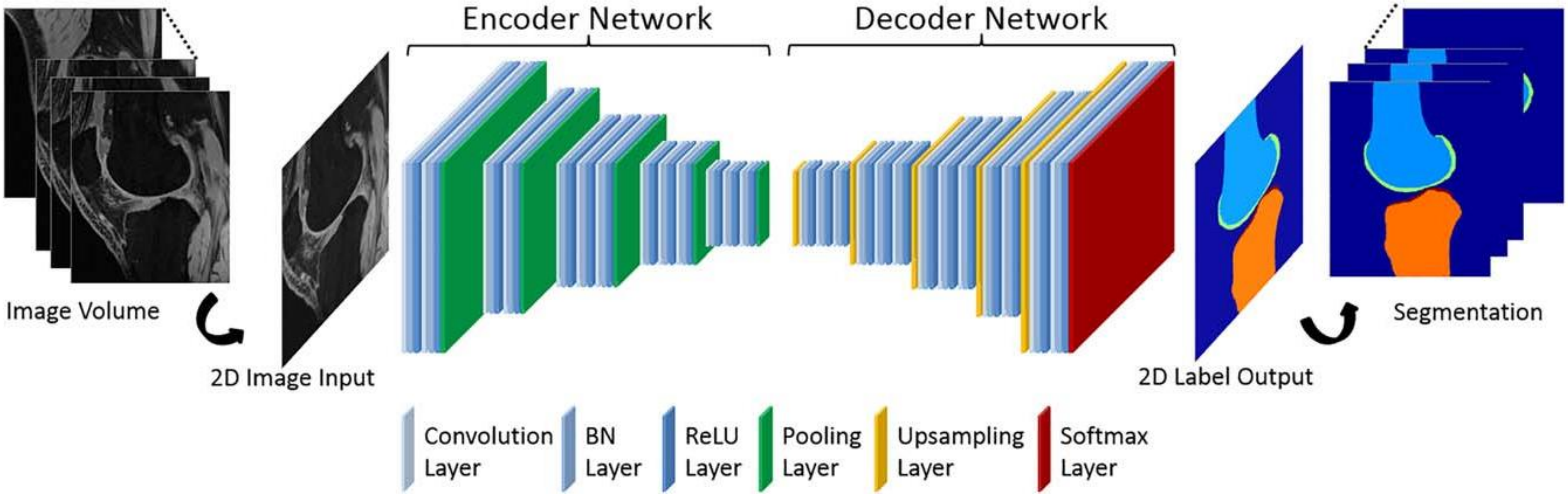
- Pixel-wise classification
  - Scene understanding
    - Autonomous driving
  - Medical imaging
  - Precision agriculture





Segmentation Architecture in General

- Same resolution is needed at the end

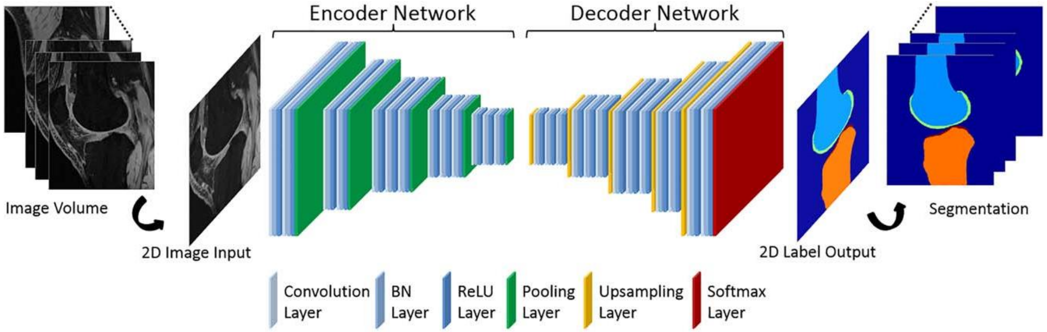


Contraction side

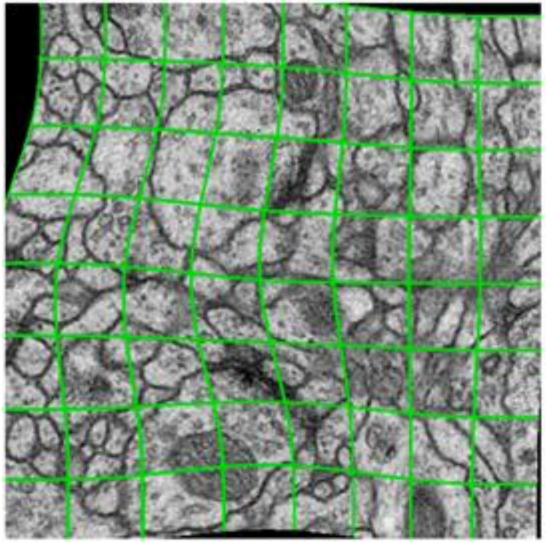
Expansion side

Segmentation Architecture in General

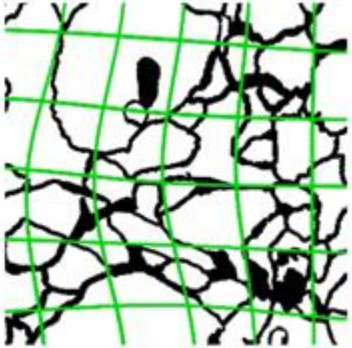
- Encoder Network: extract image features using deep convolutional network
  - Each layer: bank of trainable convolutional filters, followed by
  - ReLUs and max-pooling to downsample image features
- Decoder Network: upsamples feature map back to image resolution with final output having same number of channels as there are pixel classes
  - Deconvolution
  - Network mirrors encoder network
- Pixel-wise softmax over final feature map and cross-entropy loss function for training using some kind of SGD.



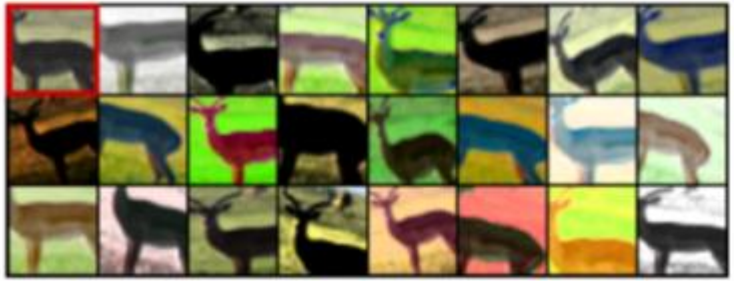
U-Net



resulting deformed image
(for visualization: no rotation, no shift, no extrapolation)



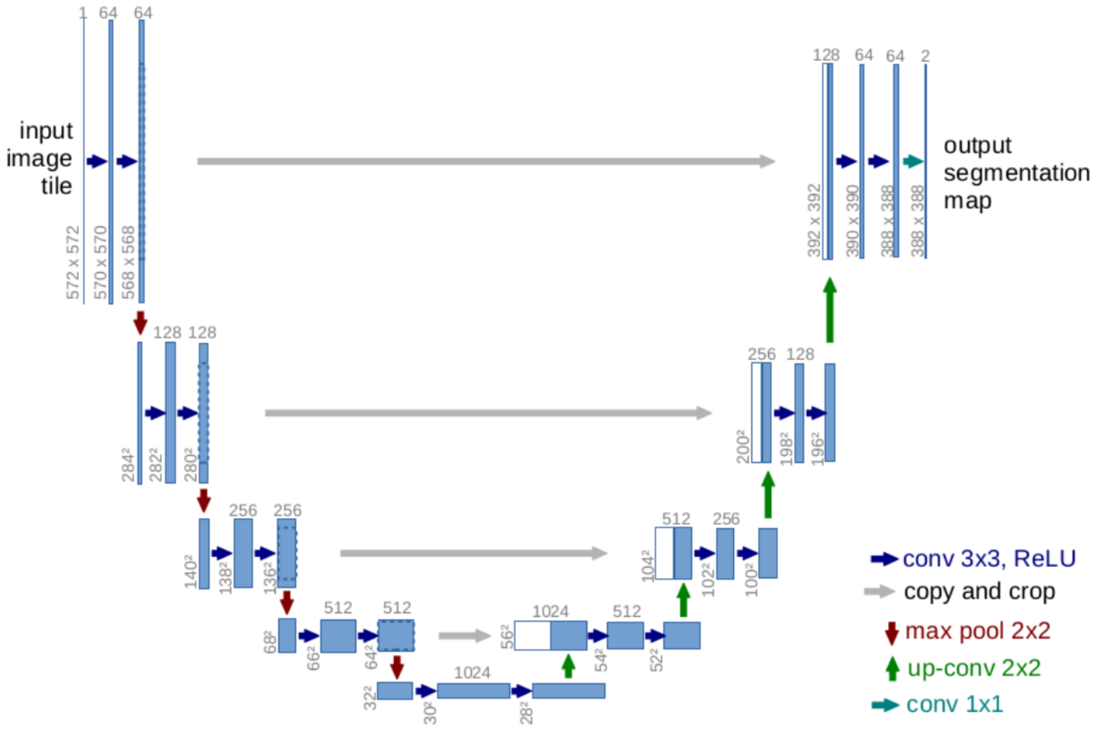
correspondingly deformed
manual labels



From [3]

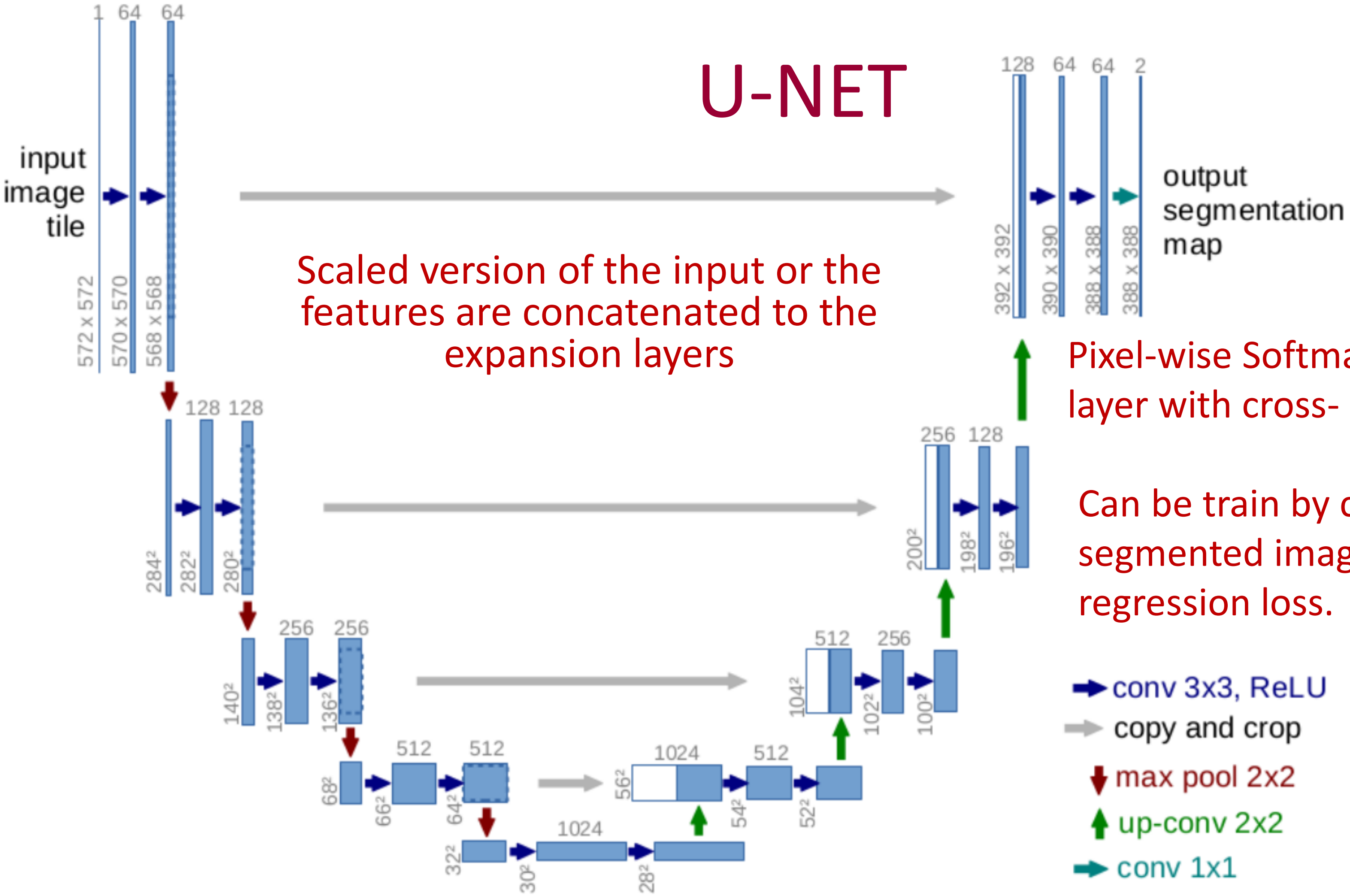
- Designed for biomedical image processing: cell segmentation
- Data augmentation via applying elastic deformations,
  - Natural since deformation is a common variation of tissue
  - Smaller dataset is enough

U-Net



- Concatenate features from encoder network
  - instead of reusing pooling indices
- Relatively shallow network with low computational demand
  - 3x3 convolution kernel size only
  - 2x2 max pooling
- No fully connected layer in the middle

U-NET



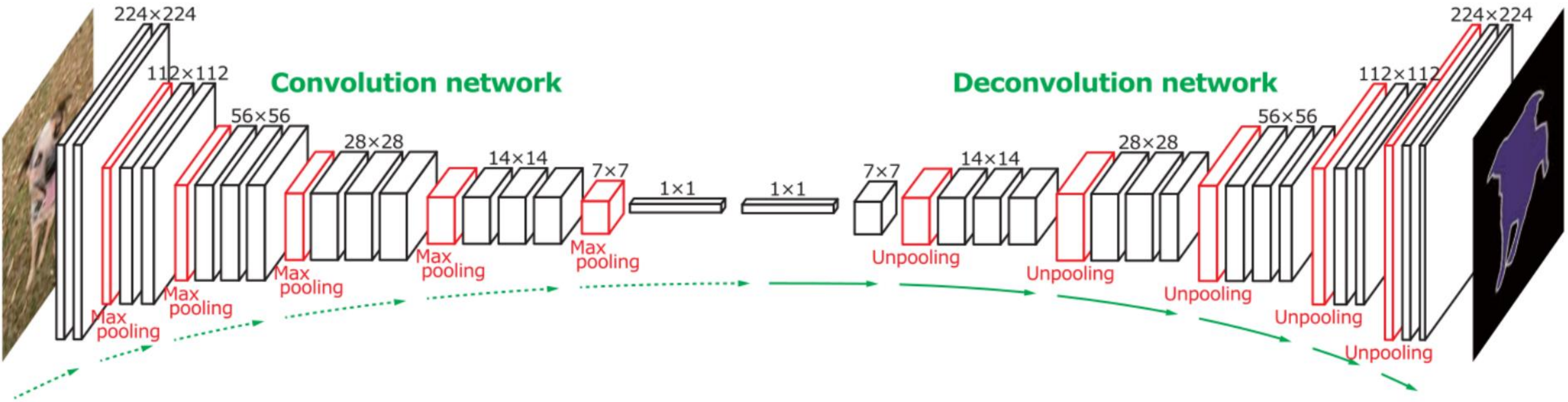
Scaled version of the input or the features are concatenated to the expansion layers

Pixel-wise Softmax at the last layer with cross- entropy loss.

Can be train by colored segmented image with regression loss.

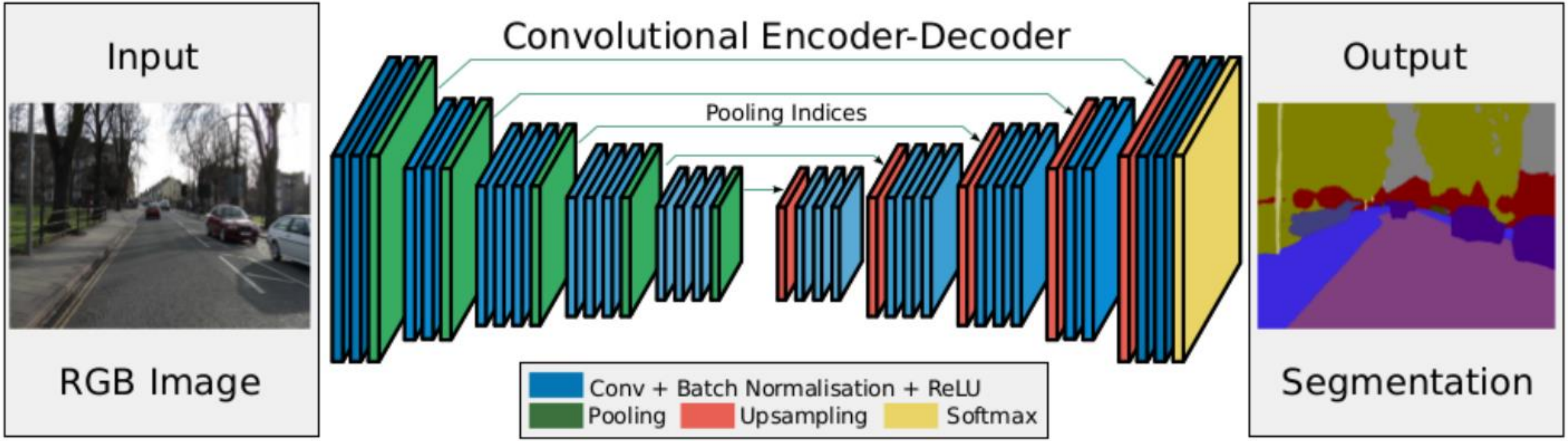
DeconvNet

- Instance-wise segmentation
- Two-stage training:
  - train on easy examples (cropped bounding boxes centered on a single object) first and
  - then more difficult examples



SegNet

- 13 convolutional layers from VGG-16
  - The original fully connected layers are discarded
- Max pooling indices (locations) are stored and sent to decoder
- Scene understanding



 https://towardsdatascience.com/review-segnet-semantic-segmentation-e66f2e30fb96

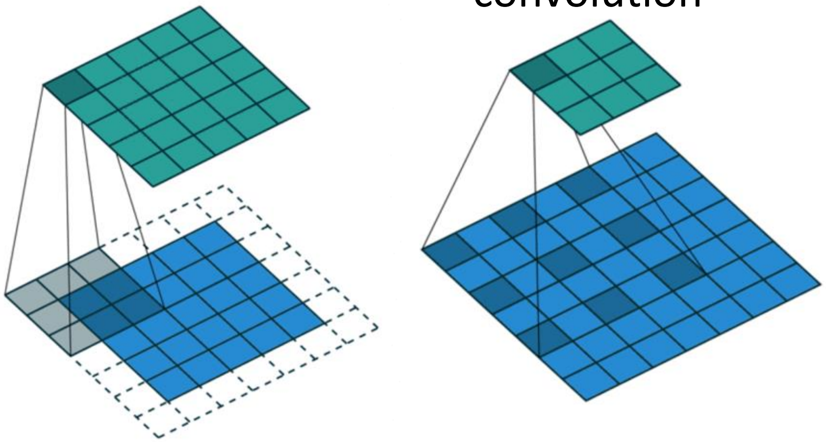
Avoiding resolution loss but no high computational load:

Atrous convolution

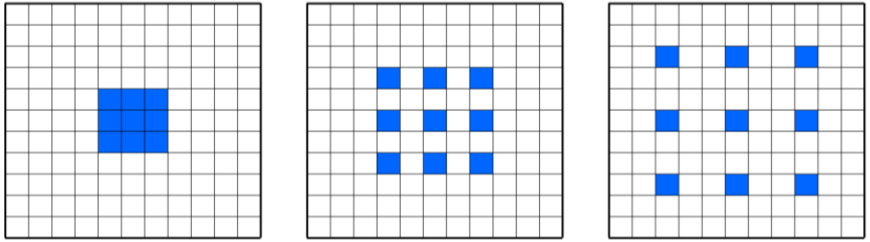
- How it works?
  - Blows up the kernel
  - Filling up the holes with zeros
    - Atrous means very dark (like the wholes between the values)
- Properties
  - Not doing downsampling
  - Not increasing computational load
  - But reaches larger neighborhood
  - Combines information from larger neighborhood

Normal convolution

Atrous (dilated) convolution

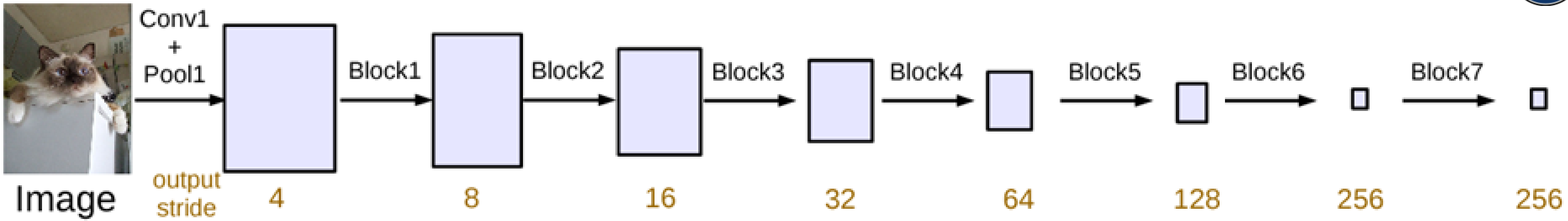


kernel

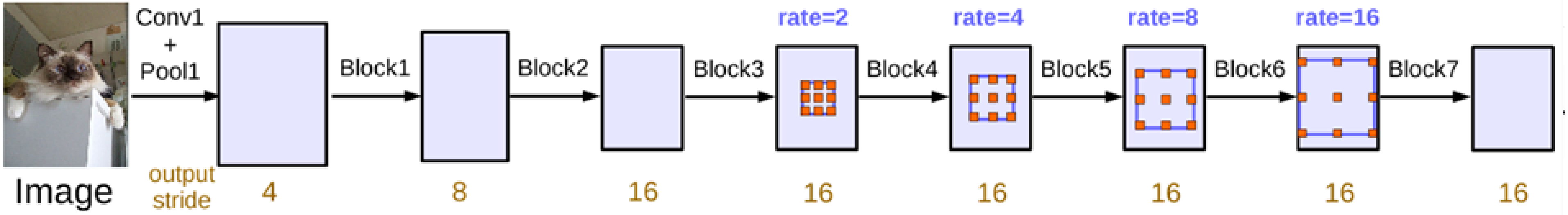


rate=1 rate=2 rate=3

Depth-to-Space



Normal convolution goes deeper with reducing resolution



Atrous convolution goes deeper without further reducing resolution

Filter size considerations

- **Small** field-of-view → accurate **localization**
- **Large** field-of-view → context **assimilation**
- Effective filter size increases (enlarge the field-of-view of filter)

$$n_o\colon k \times k \quad \rightarrow \quad n_a\colon \big(k + (k-1)(r-1)\big) \times \big(k + (k-1)(r-1)\big)$$

n_o : original convolution kernel size

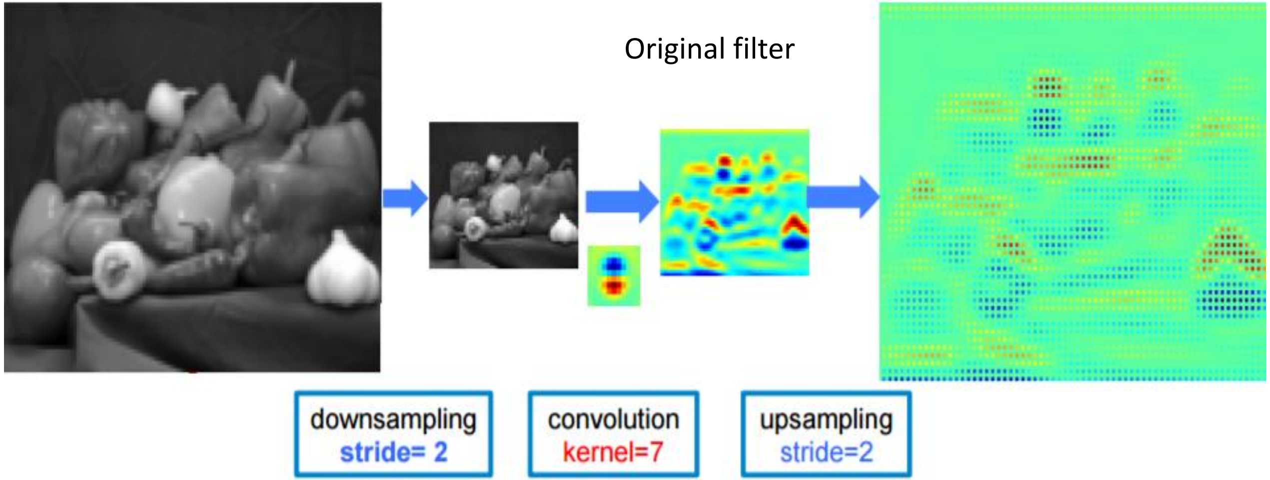
n_a : atrous convolution kernel size

r: rate

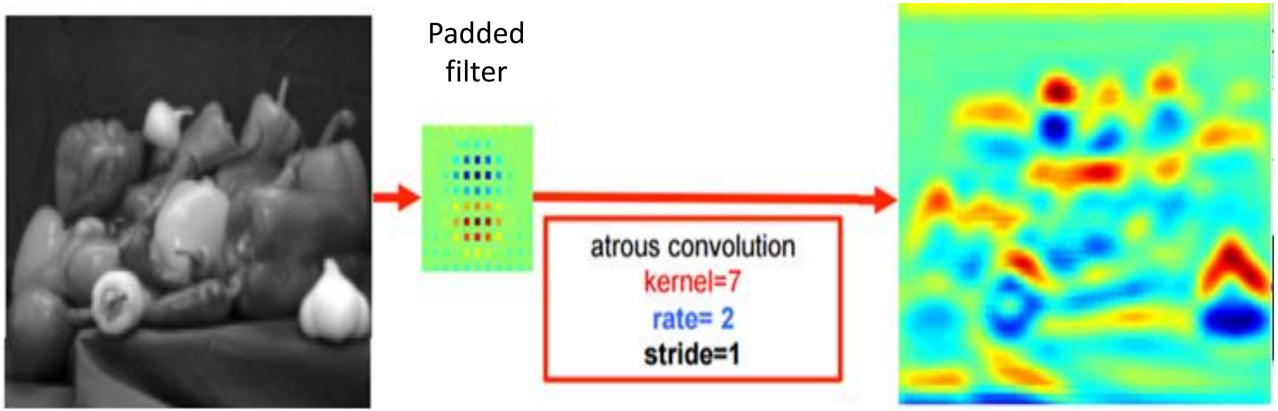
- However, we take into account only the non-zero filter values:
  - Number of filter parameters is the same
  - Number of operations per position is the same

Visualizing atrous convolution

Standard convolution

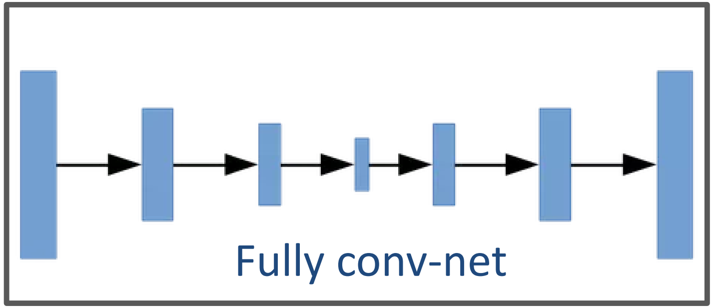


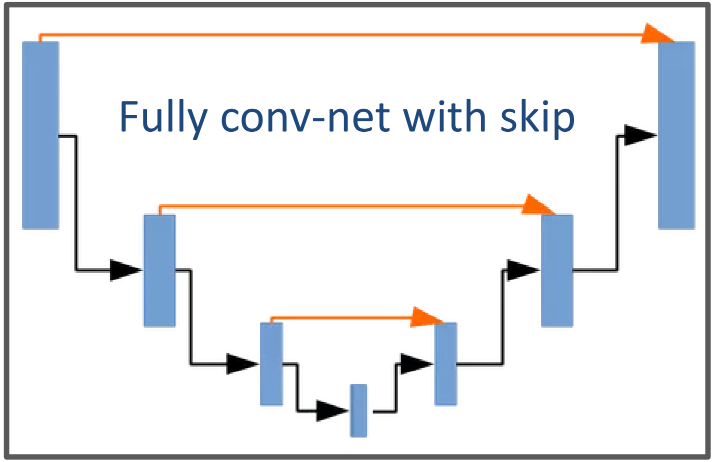
Atrous convolution

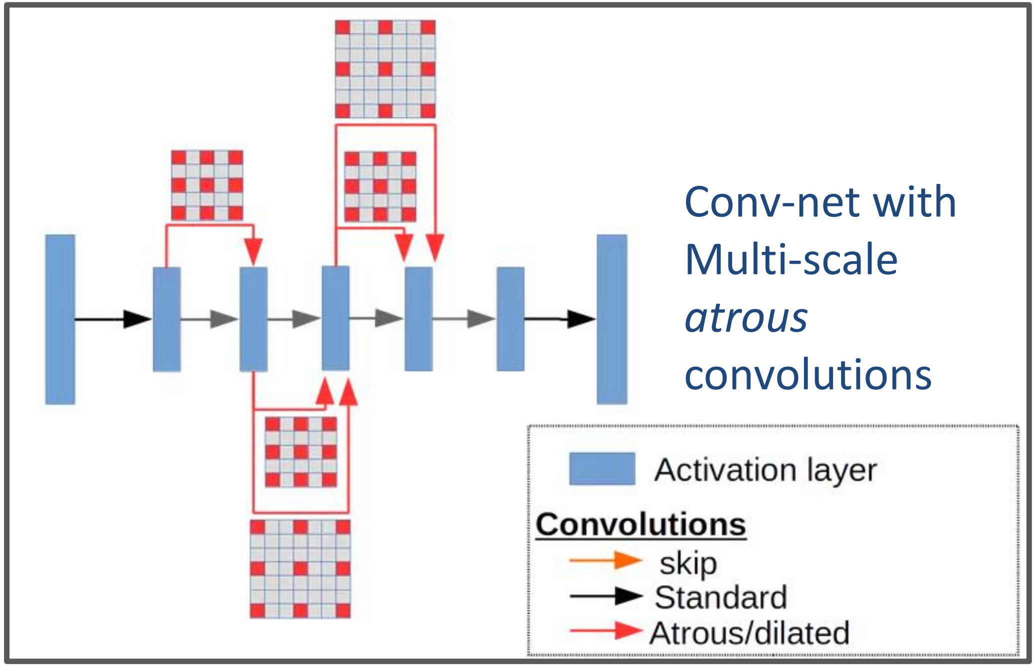


Chen, Liang-Chieh, et al. "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs." *arXiv preprint arXiv:1606.00915* (2016).

Semantic segmentation CNN arrangements







- How to solve reduced resolution?
  - Do not downsample !!!
    - Convolution on large images ⇒ Small FOV Enlarge kernel
    - Size O(n^2) more parameters ⇒ getting close to fully
    - Connected layer, slow training, overfitting
  - Atrous Convolution.
    - Large FOV with little parameters → Kill two birds with one stone!

Neural Networks

Unsupervised learning techniques

(P-ITEEA-0011)

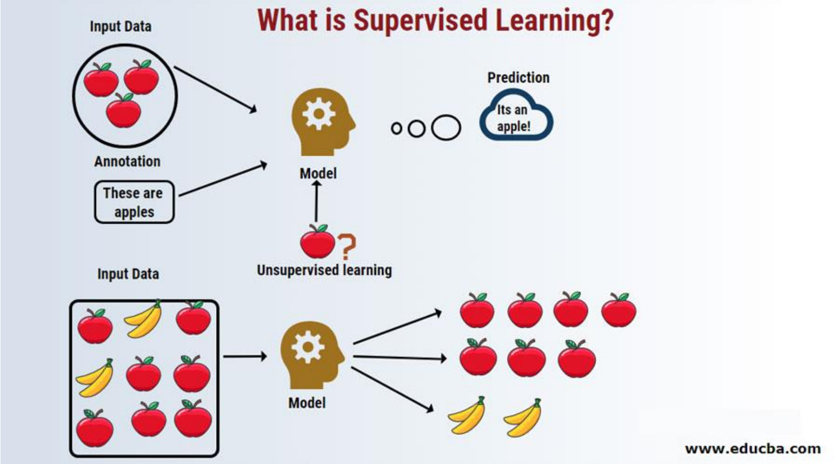
Akos Zarandy
Lecture 9
November 19, 2019

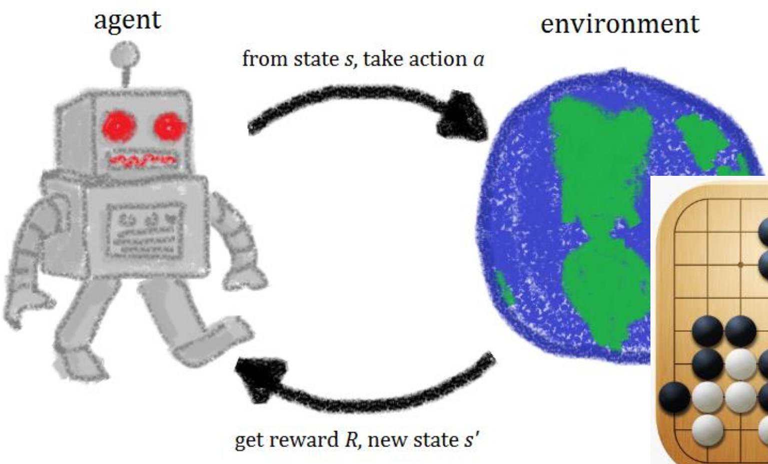
Contents

- Supervised vs unsupervised learning
- Unsupervised learning techniques
  - Curse of dimensionality
  - Principal component analysis (PCA)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
  - Autoencoder

Typical Machine Learning Types

- **Supervised Learning**
  - Learning from labeled examples (for which the answer is known)
- **Unsupervised Learning**
  - Learning from unlabeled examples (for which the answer is unknown)
- **Reinforcement Learning**
  - Learning by trial and feedback, like the “child learning” example





Supervised vs Unsupervised learning

- Supervised learning
  - We have prior knowledge of the desired output
    - Always have data set with ground truth (like image data sets with labels)
  - Typical tasks
    - Classification
    - Regression

- Unsupervised learning
  - No prior knowledge of the desired output
    - Received radio signals from deep space
  - Typical tasks
    - Clustering
    - Representation learning
    - Density estimation

We wish to learn the inherent structure of (patterns in) our data.

Use cases for unsupervised learning

- Exploratory analysis of a large data set
  - Clustering by data similarity
  - Enables verifying individual hypothesizes after analyzing the clustered data

- Dimensionality reduction
  - Represents data with less columns
  - Allows to present data with fewer features
  - Selects the relevant features
  - Enables less power consuming data processing, and/or human analysis

Curse of dimensionality

- What is it?
  - A name for **various problems that arise** when analyzing data in high dimensional space.
  - Dimensions = independent features in ML
    - Input vector size (different measurements, or number of pixels in an image)
  - Occurs when d (# dimensions) is large in relation to n (number of samples).

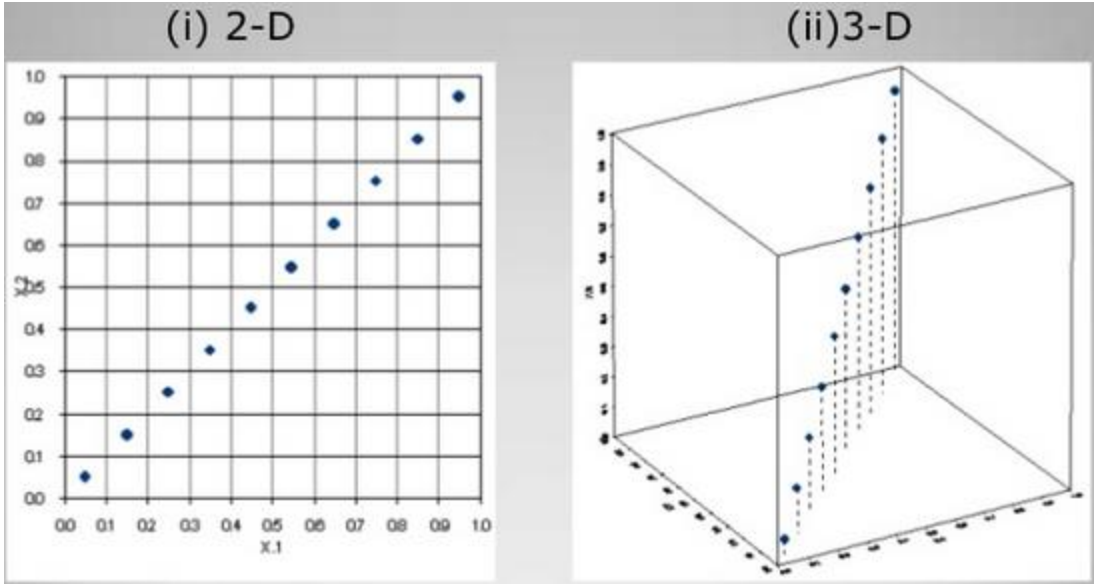
- Real life examples:
  - Genomics
    - We have ~20k genes, but disease sample sizes are often in the 100s or 1000s.

So what is this curse?

- **Sparse data:**
  - When the dimensionality d increases, the volume of the space increases so fast that the available data becomes **sparse, i.e. a few points in a large space**
  - Many features are not balanced, or are ‘rarely occur’ – sparse features
- **Noisy data**: More features can lead to increased noise → it is harder to find the true signal
- **Less clusters**: Neighborhoods with fixed k points are less concentrated as d increases.
- **Complex features**: High dimensional functions tend to have more complex features than low-dimensional functions, and hence harder to estimate

Data becomes sparse as dimensions increase

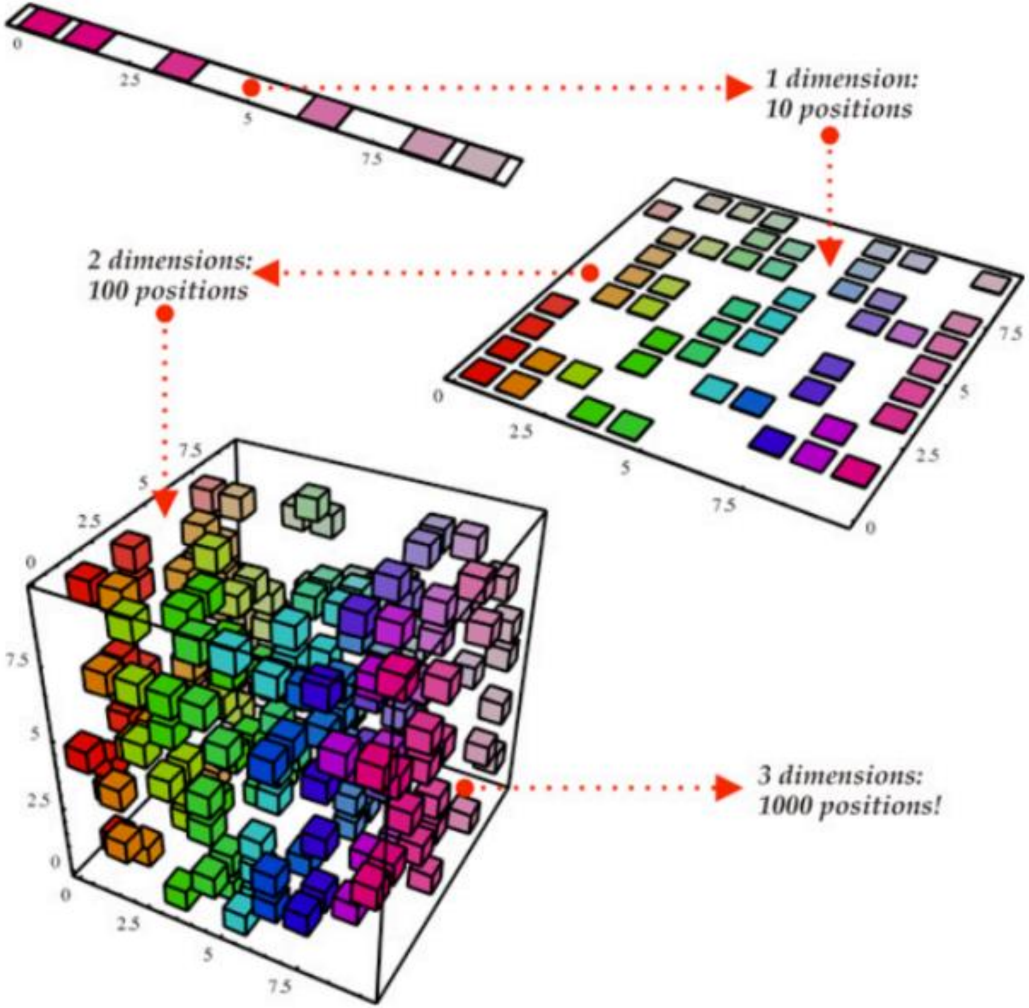
- A sample that maps 10% of the 1x1 squares in 2D represent only 1% of the 1x1x1 cubes in 3D



- There is an exponential increase in the search-space

Data sample number increase to avoid sparsity

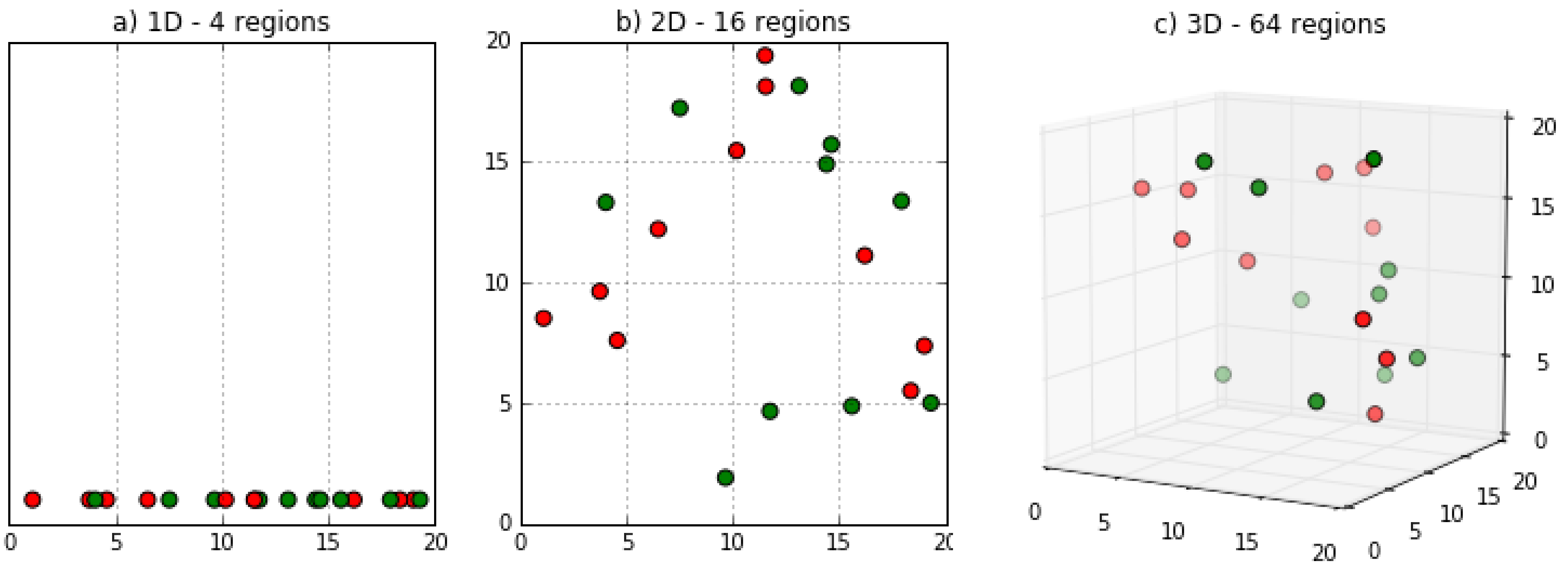
- e.g. 10 observations /dimension
  - 1D: 10 observations
  - 2D: 100 observations
  - 3D: 1000 observations
  - …



Curse of dim - Running complexity

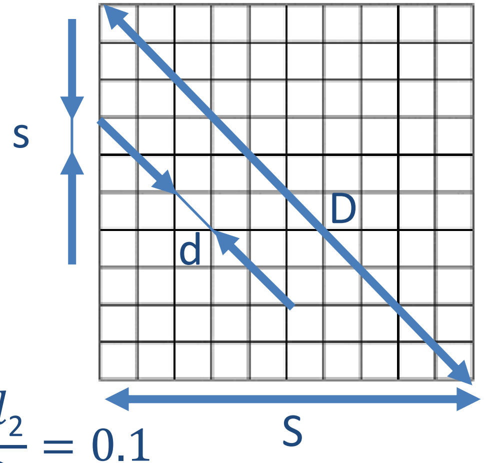
- Many data points (labeled measurements) are needed
- Complexity (running time) increase with dimension ***d***
- A lot of methods have at least $O(n*d^2)$ complexity, where ***n*** is the number of samples
- As ***d*** becomes large, this complexity becomes very costly.
  - Compute = $

Sparisty increase: More regions with the same number of data points



Distances in high dimension

- Assume, we have a unit side (2D) square, what we divided to 100 equal small squares
  - Calculate the ratio of the largest distance in a small square and the largest distance of the big square (in 2D)



$S_2=1$

$s_2=\sqrt[2]{\frac{1}{100}}=0.1$

$D_2=\sqrt{2}$

$d_2=0.1\sqrt{2}$

$$R_2=\frac{d_2}{D_2}=0.1$$

- Assume, we have a unit side 100D cube, what we divided to 100 equal small 100D cubes
  - Calculate the ratio Ratio of the largest distance in a small cube and the largest distance of the big cube (in 100D)
  - The average nearest neighbor distance is 95% of the largest distance!!!
  - Euclidian distance becomes meaningless, most two points are “far” from each others

$S_{100}=1$ $\quad s_{100}=\sqrt[100]{\frac{1}{100}}=0.95$

$D_{100}=\sqrt{100}=10$ $\quad d_{100}=\sqrt{100*0.95^2}=9.5$

$$R_{100}=\frac{d_{100}}{D_{100}}=0.95$$

Curse of dim - Some mathematical (weird) effects

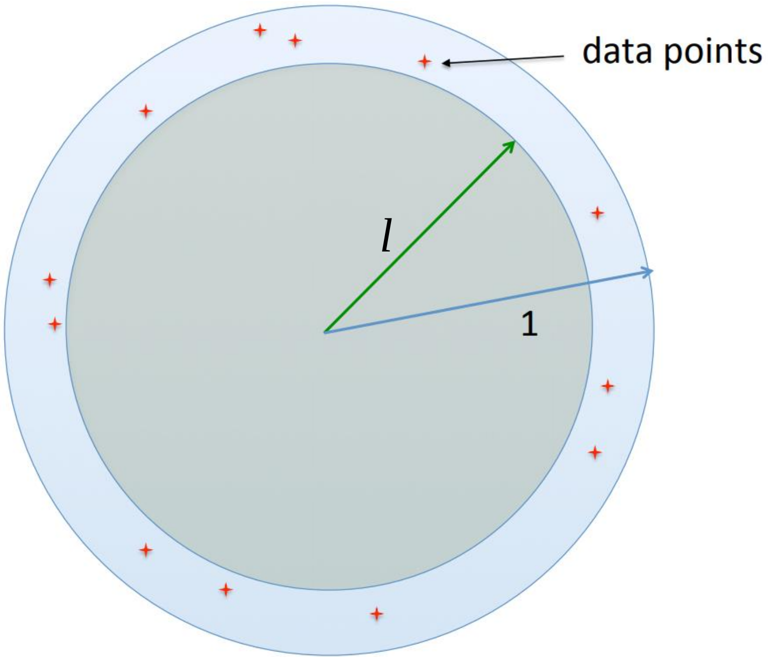
- Ratio between the volume of a sphere and a cube for **d**=3: $\frac{(\frac{4}{3})\pi r^3}{(2r)^3} \approx \frac{4r^3}{8r^3} \approx 0.5$
- When **d** tends to infinity the volume of the sphere (this ratio) tends to zero

d	3	5	10	20	30	50
ratio	0.52	0.16	0.0025	2.5E-08	2.0E-14	1.5E-28

- Most of the data is in the corner of the cube
  - Thus, Euclidian distance becomes meaningless, most two points are “far” from each others
- Very problematic for methods such as k-NN classification or k-means clustering because most of the neighbors are equidistant

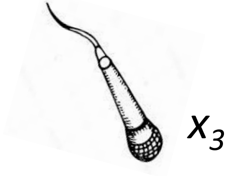
The nearest neighbor problem in a sphere

- Assume randomly distributed points in a sphere with a unit diameter
- The median of the nearest neighbors is l
- As dimension tends to infinity
  - The median of the nearest neighbors converges to 1



"The Curse of Dimensionality" by Raúl Rojas
https://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/dimensionality.pdf

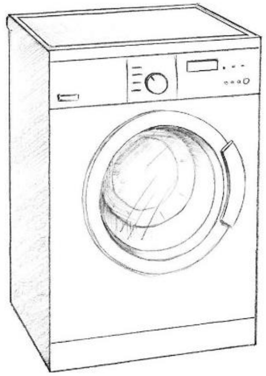
How to calculate dimensionality?



feature vectors (x)

observations (d)

	x_1	x_2	x_3	x_4
d_1	1	2	1	1
d_2	2	4	3.5	1
d_3	3	6	17	1

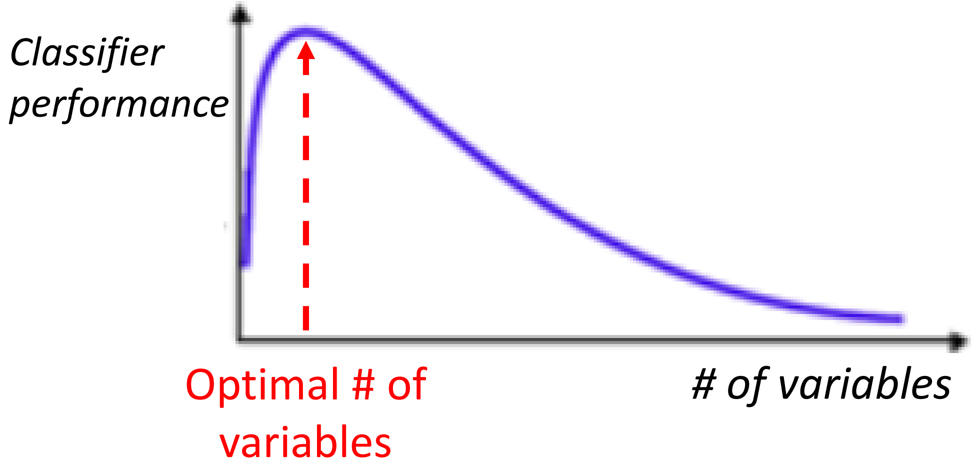


- How many dimensions does the data intrinsically have here?
  (How many independent coordinates?)
  - Two!
    - x1 = ½ * x2 (no additional information, correlated, not independent)
    - x4 is constant (carries no information at all!)

How to avoid the curse?

- Reduce dimensions
  - **Feature selection** - Choose only a subset of features
  - Use algorithms that transform the data into a lower dimensional space (example – PCA, t-SNE)

    **Both methods often result in information loss*
- Less is More
  - In many cases the information that is lost by discarding variables is made up for by a more accurate mapping/sampling in the lower-dimensional space



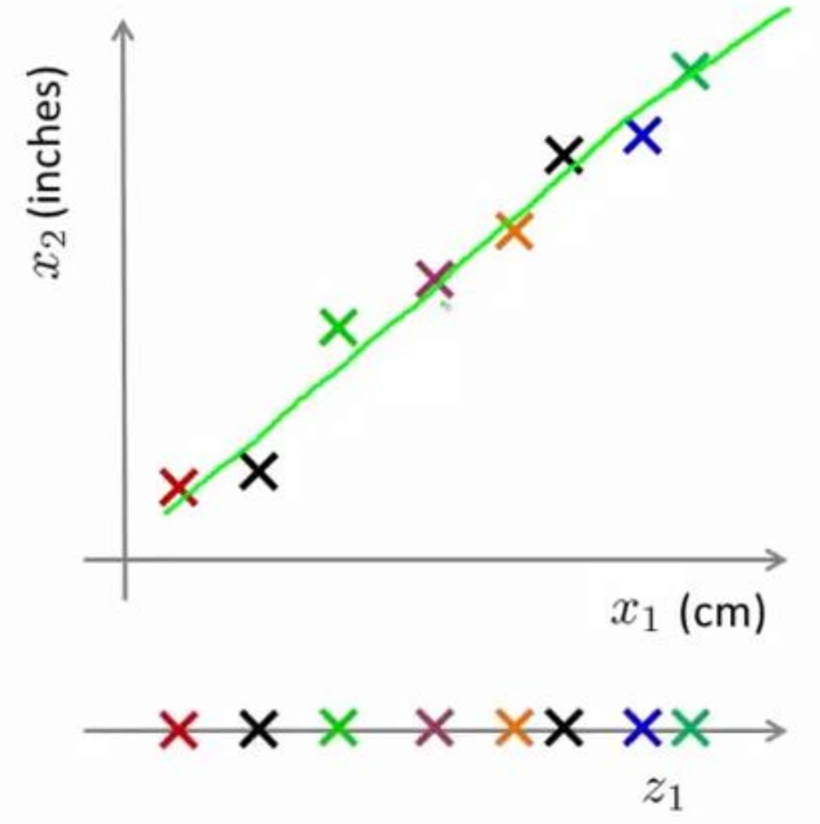
Principal component analysis

(PCA)

Dimensionality reduction goals

- Improve ML performance
- Compress data
- Visualize data (you can’t visualize >3 dimensions)
- Generate new complex features
  - Loosing the meaning of a feature
  - Combining temperature, sound and current to one feature will be meaningless for human (non-physical)

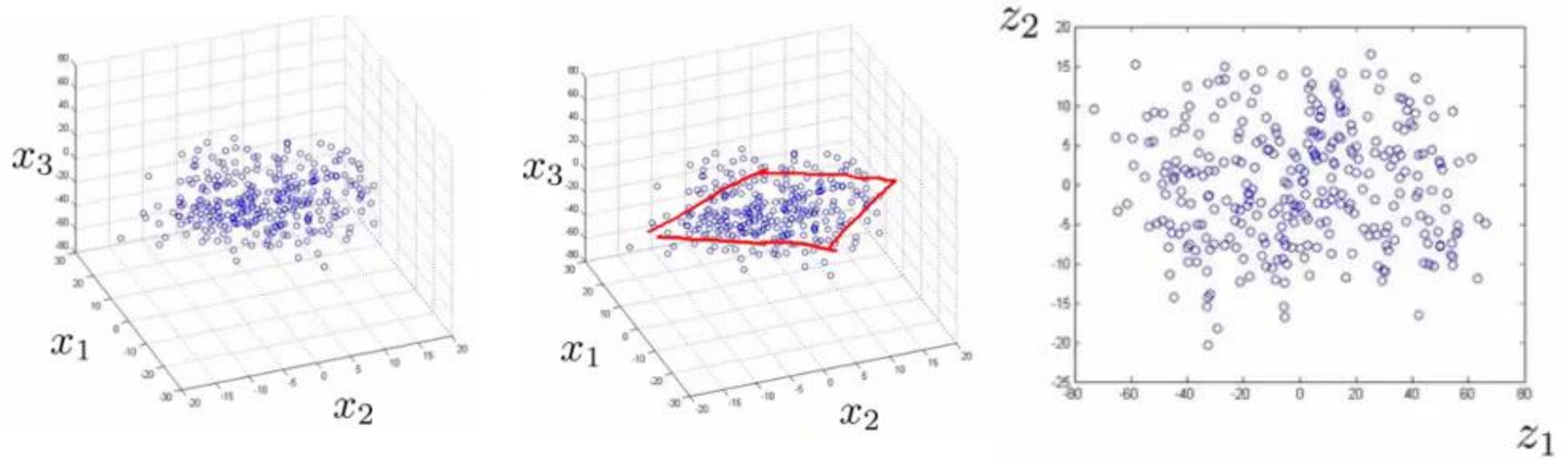
Example – reducing data from 2d to 1d



- X1 and x2 are pretty redundant. We can reduce them to 1d along the green line
- This is done by projecting the points to the line (some information is lost, but not much)

Example – 3D to 2D

- Despite having 3D data most of it lies close to a plane



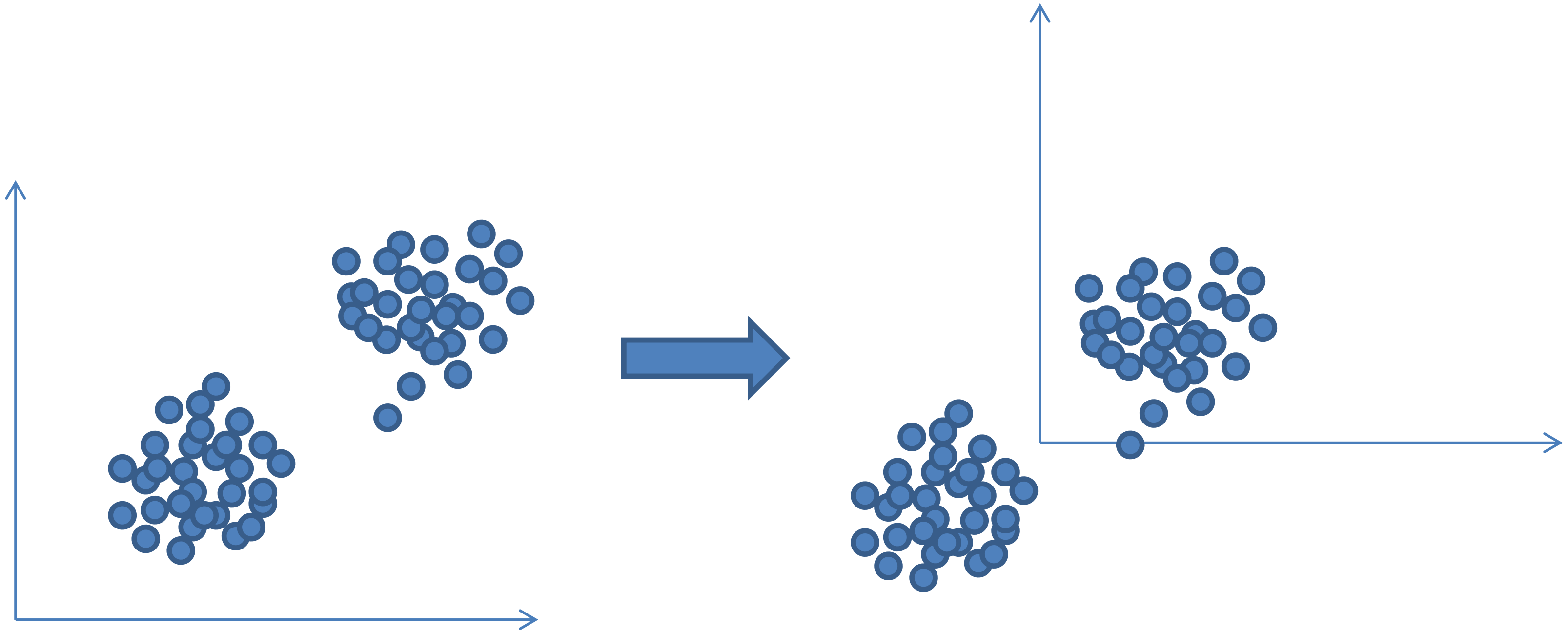
- If we were to project the data onto a plane we would have a more compact representation
- So how do we find that plane without loosing too much of the **variance** in our data? → PCA

Principal component analysis (PCA)

- Technique for dimensionality reduction
- Invented by Karl Pearson (1901)
- Linear coordinate transformation
  - converts a set of observations of possibly correlated variables
  - into a set of values of linearly uncorrelated orthogonal variables called principal components
- Deterministic algorithm

PCA algorithm

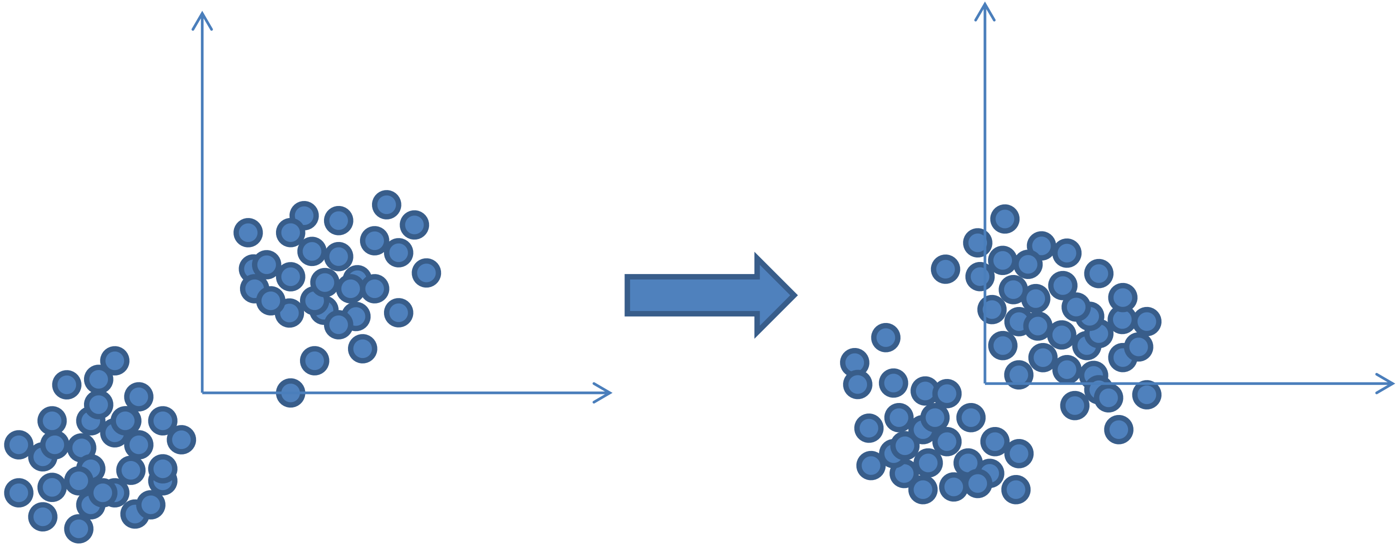
1. **Mean normalization:** For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.



https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

PCA algorithm

1. **Mean normalization:** For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.
2. **Standardization (optional):** Do it, if you want to have each of your features the same variance.



https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

PCA algorithm

1. **Mean normalization:** For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.
2. **Standardization (optional):** Do it, if you want to have each of your features the same variance.
3. **Covariance matrix:** Calculate the covariance matrix

https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

Covariance (formal definition)

- Assume that $\mathbf{x}$ are random variable vectors
- We have *n* vectors

$$\text{Variance(x)} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$$= \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})$$

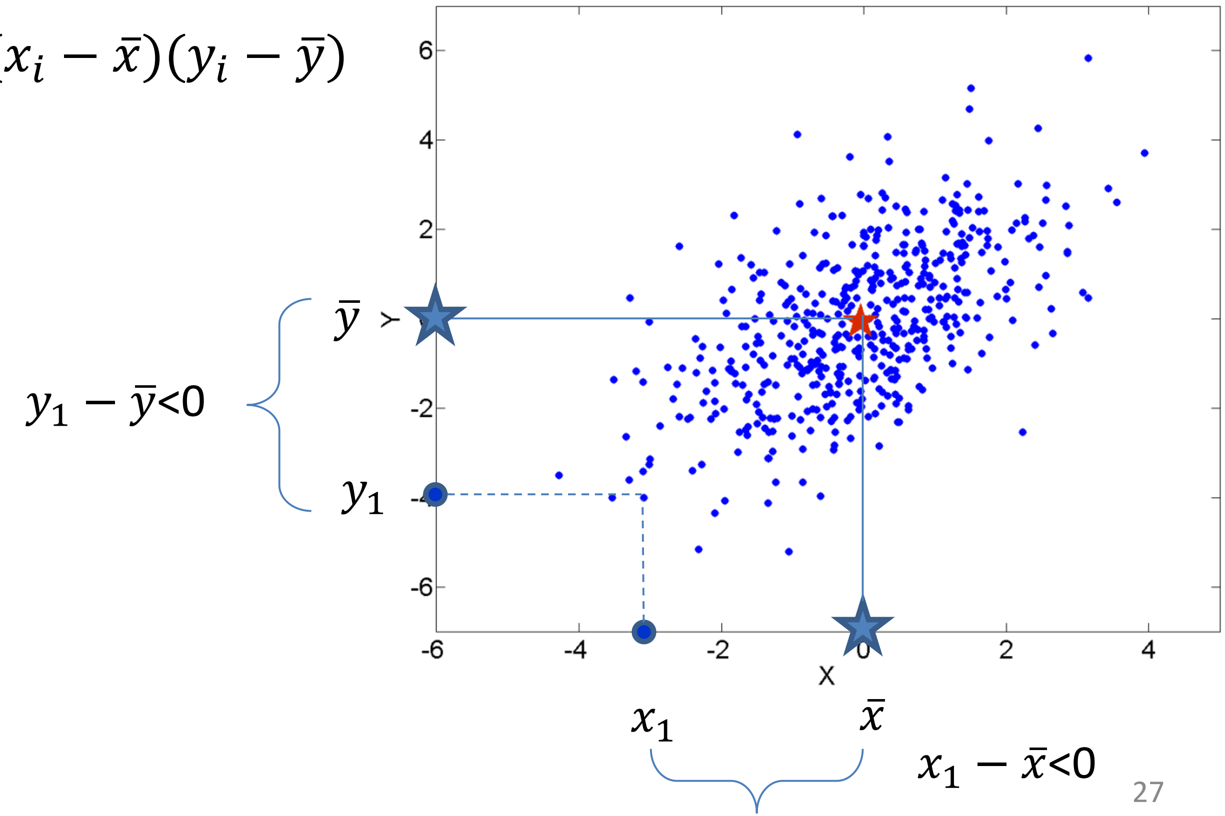
$$\text{Covariance}(\mathrm{x}, \mathrm{y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- $\text{Covariance}(\mathrm{x}, \mathrm{x}) = \text{var}(\mathrm{x})$
- $\text{Covariance}(\mathrm{x}, y) = \text{Covariance}(\mathrm{y}, \mathrm{x})$

Covariance example for 2D

$$\text{Covariance}(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

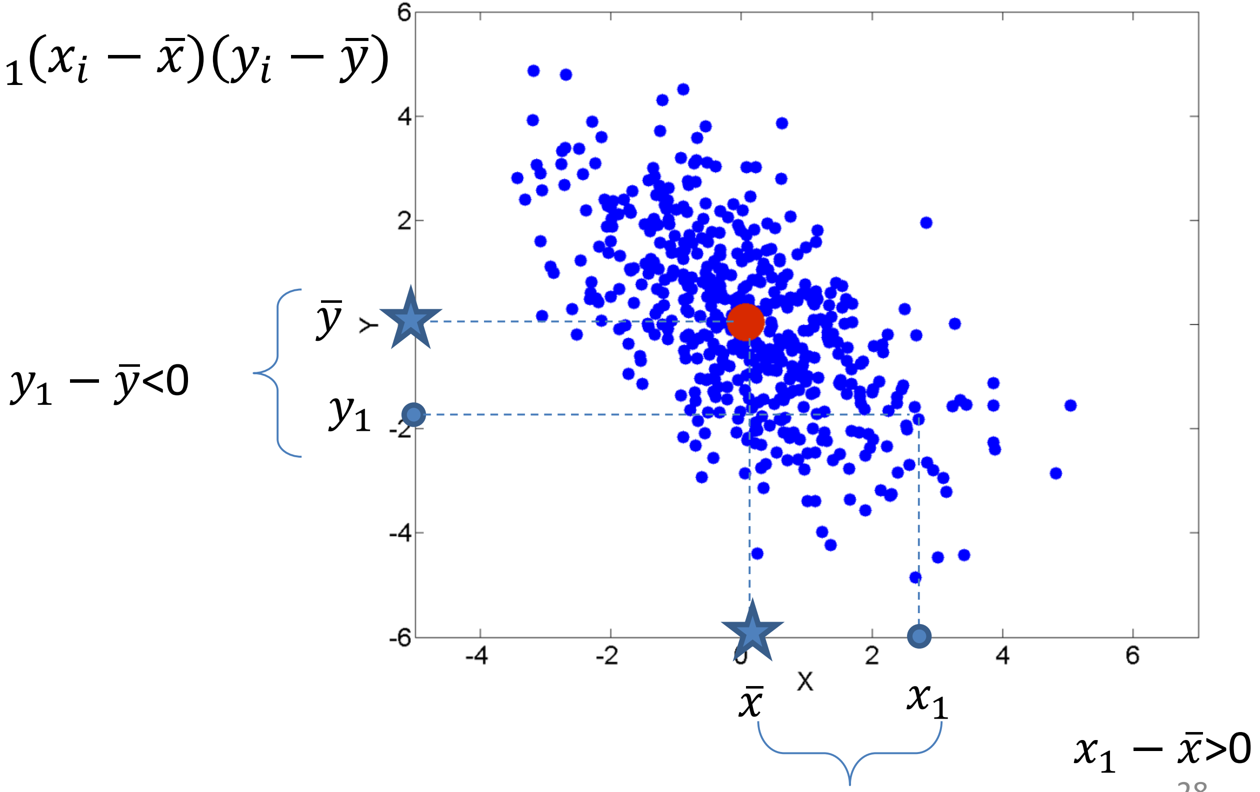
- Positive covariance between the two dimensions



Covariance example for 2D

$$\text{Covariance(x, y)} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

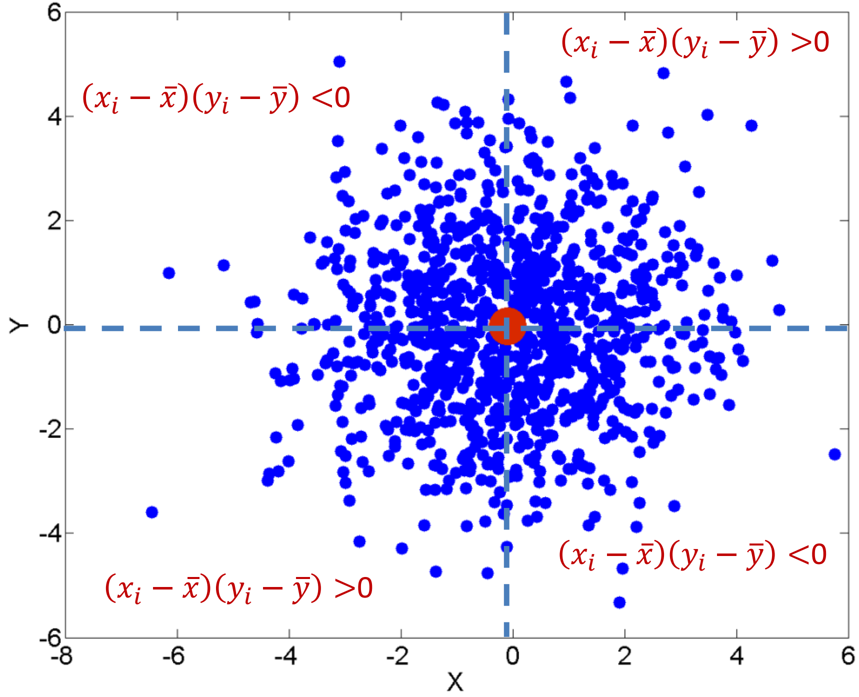
- Negative covariance between the two dimensions



Covariance example for 2D

$$\text{Covariance}(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- No covariance between the two dimensions



Covariance matrix

- Diagonal elements are variances, i.e. Cov(x, x)=$var(x)$
  - n is the number of the vectors
  - m is the dimension

$$Cov\ (\Sigma) = \begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & \cdots & cov(x_1, x_m) \\ cov(x_2, x_1) & cov(x_2, x_2) & \cdots & cov(x_2, x_m) \\ \vdots & \vdots & \vdots & \vdots \\ cov(x_m, x_1) & cov(x_m, x_2) & \cdots & cov(x_m, x_m) \end{bmatrix}$$

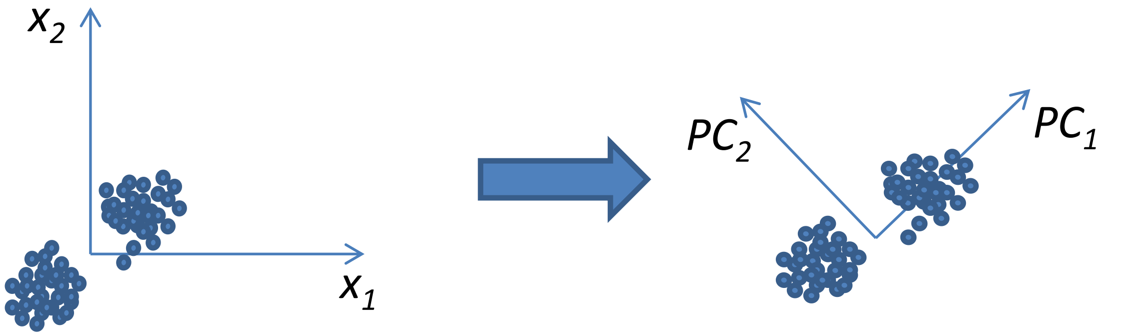
$$Cov\ (\Sigma) = \frac{1}{n}(X - \bar{X})(X - \bar{X})^T; where\ X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

- Covariance Matrix is symmetric
  - commutative

$$Cov\ (\Sigma) = \begin{bmatrix} var(x_1, x_1) & cov(x_1, x_2) & \cdots & cov(x_1, x_m) \\ cov(x_2, x_1) & var(x_2, x_2) & \cdots & cov(x_2, x_m) \\ \vdots & \vdots & \vdots & \vdots \\ cov(x_m, x_1) & cov(x_m, x_2) & \cdots & var(x_m, x_m) \end{bmatrix}$$

PCA algorithm

1. Mean normalization: For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.
2. Standardization (optional): Do it, if you want to have each of your features the same variance.
3. Covariance matrix: Calculate the covariance matrix
4. Eigenvectors and eigenvalues of the covariance matrix
   – Note: Each new axis (PC) is an eigenvector of the data. The standard deviation of the data variance on the new axis is the eigenvalue for that eigenvector.



Principal components will be orthogonal. Uncorrelated, independent!

https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

PCA algorithm

1. Mean normalization: For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.
2. Standardization (optional): Do it, if you want to have each of your features the same variance.
3. Covariance matrix: Calculate the covariance matrix
4. Eigenvectors and eigenvalues of the covariance matrix
   – Note: Each new axis (PC) is an eigenvector of the data. The standard deviation of the data variance on the new axis is the eigenvalue for that eigenvector.
5. Rank eigenvectors by eigenvalues
6. Keep top k eigenvectors and stack them to form a feature vector
7. Transform data to PCs:
   – New data = feature vectors (transposed) * original data

https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

From k original variables: $x_1, x_2, \ldots, x_k$:

Produce k new variables: $y_1, y_2, \ldots, y_k$:

$$\left.\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \ldots + a_{1k}x_k \\ y_2 &= a_{21}x_1 + a_{22}x_2 + \ldots + a_{2k}x_k \\ &\ldots \\ y_k &= a_{k1}x_1 + a_{k2}x_2 + \ldots + a_{kk}x_k \end{aligned}\right\}$$

y_k's are **Principal Components**

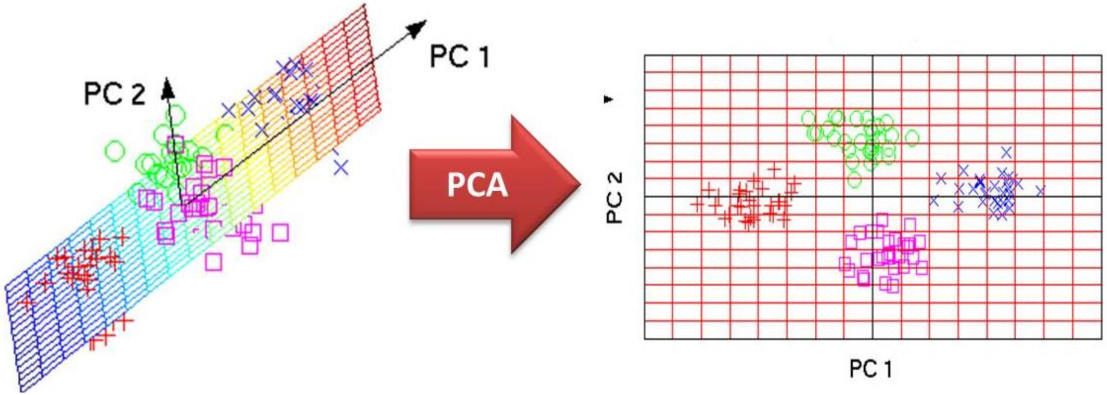
$\{a_{11}, a_{12}, \ldots, a_{1k}\}$ is 1st **Eigenvector** of of first principal component

$\{a_{21}, a_{22}, \ldots, a_{2k}\}$ is 2nd **Eigenvector** of of 2nd principal component

$\{a_{k1}, a_{k2}, \ldots, a_{kk}\}$ is kth **Eigenvector** of of kth principal component

Principal Component Analysis (PCA)

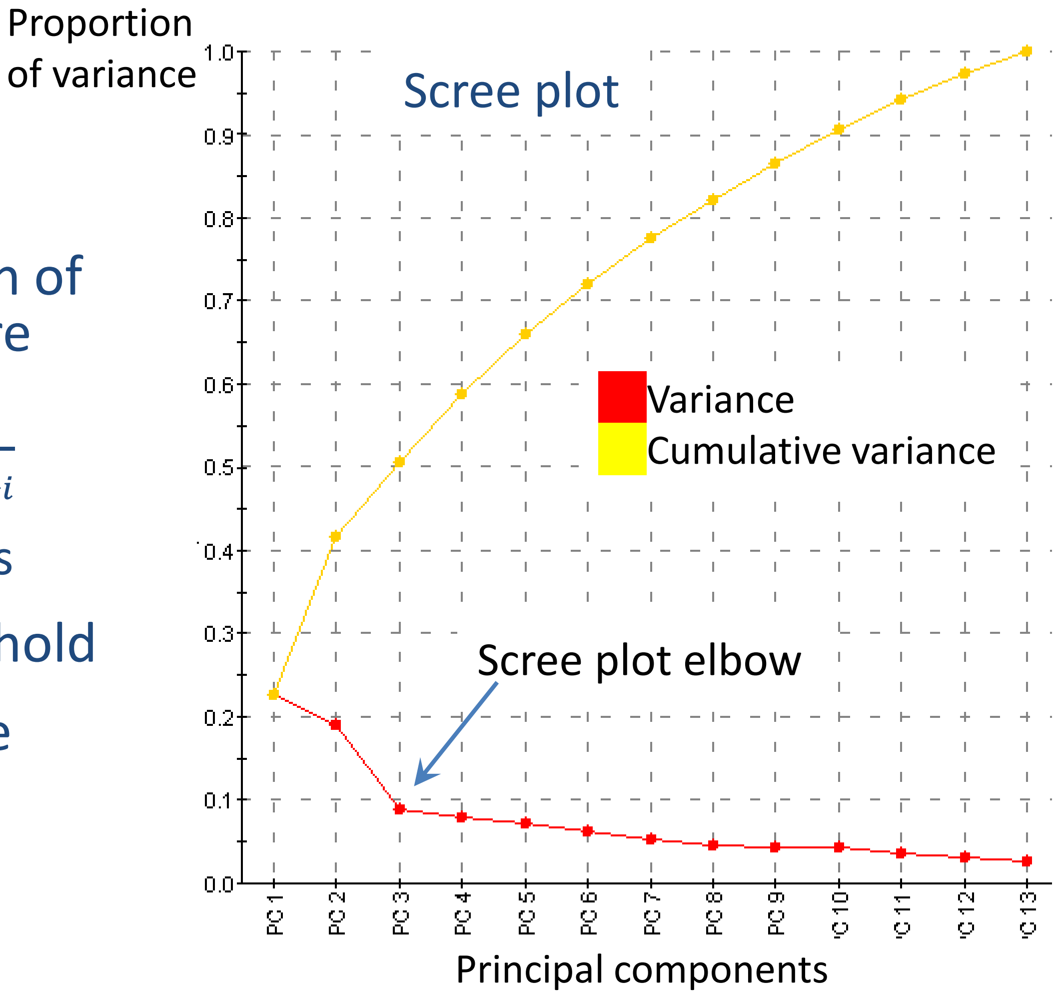
- The idea is to project the data onto a subspace which compresses most of the variance in as little dimensions as possible.
- Each new dimension is a **principle component**
- The principle components **are ordered** according to how much **variance in the data** they capture
  - Example:
    - PC1 – 55% of variance
    - PC2 – 22% of variance
    - PC3 – 10% of variance
    - PC4 – 7% of variance
    - PC5 – 2% of variance
    - PC6 – 1% of variance
    - PC7 - ….



We have to choose how many PCs to use from the top

How many PCs to use?

- Calculate the proportion of variance for each feature
  - $prop.\,of\ var. = \frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i}$
  - λ_i are the eigen values
- Rich a predefined threshold
- Or find the elbow of the Scree plot



PCA Example

- Weekly food consumption of the four countries
  - food types: variables
  - countries: observations
- Clustering the countries:
  - Needs visualization in 17 dimension
- PCA: reduce dimensionality

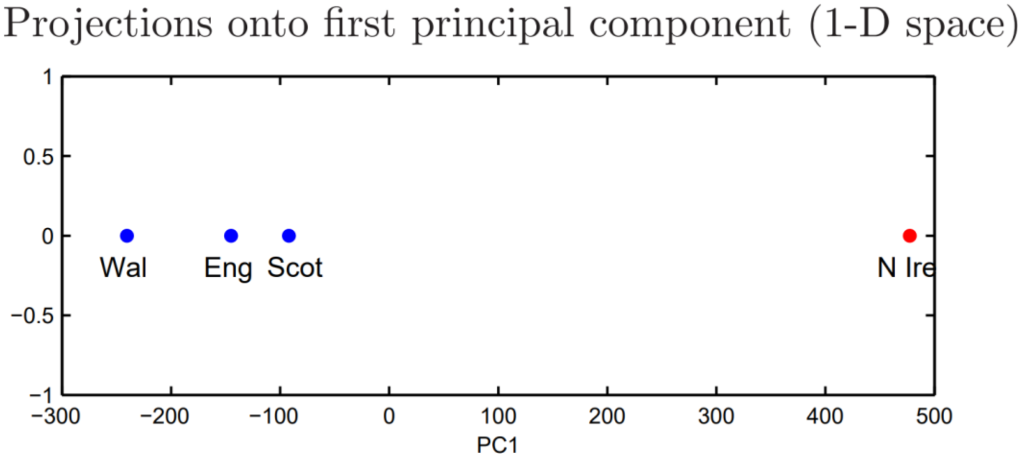
http://www.sdss.jhu.edu/~szalay/class/2016-oldold/SignalProcPCA.pdf

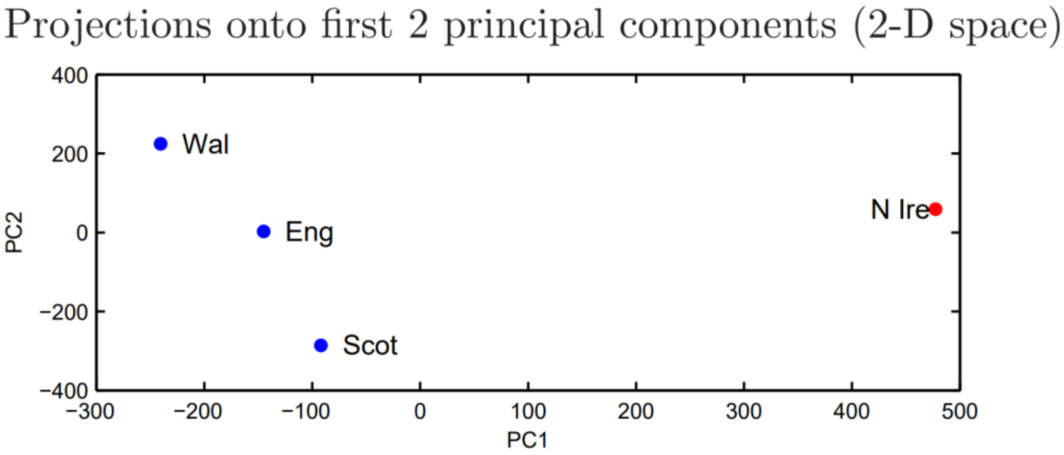
	England	Wales	Scotland	N Ireland
Cheese	105	103	103	66
Carcass meat	245	227	242	267
Other meat	685	803	750	586
Fish	147	160	122	93
Fats and oils	193	235	184	209
Sugars	156	175	147	139
Fresh potatoes	720	874	566	1033
Fresh Veg	253	265	171	143
Other Veg	488	570	418	355
Processed potatoes	198	203	220	187
Processed Veg	360	365	337	334
Fresh fruit	1102	1137	957	674
Cereals	1472	1582	1462	1494
Beverages	57	73	53	47
Soft drinks	1374	1256	1572	1506
Alcoholic drinks	375	475	458	135
Confectionery	54	64	62	41

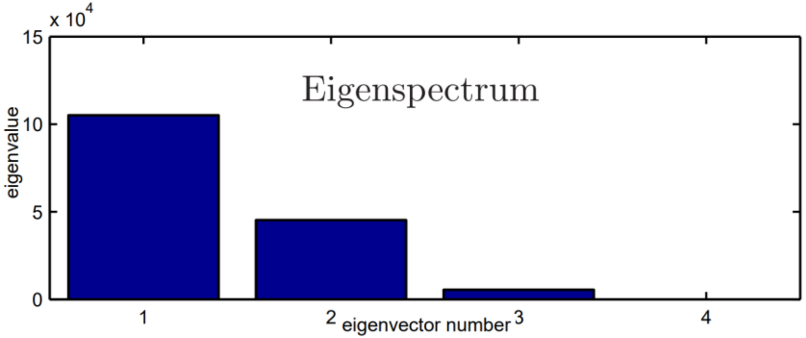
UK food consumption in 1997 (g/person/week). Source: DEFRA

PCA Example

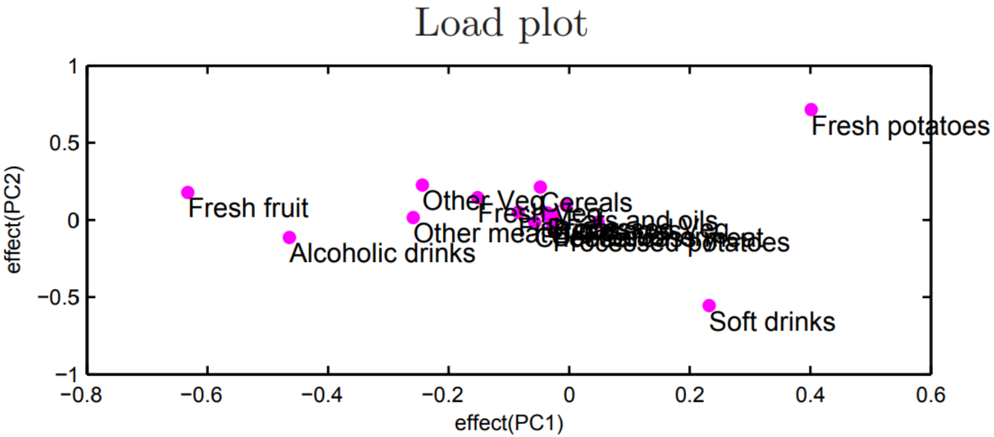
- From PC1, two clusters are well separable
- Including PC2, the four clusters can be well separated

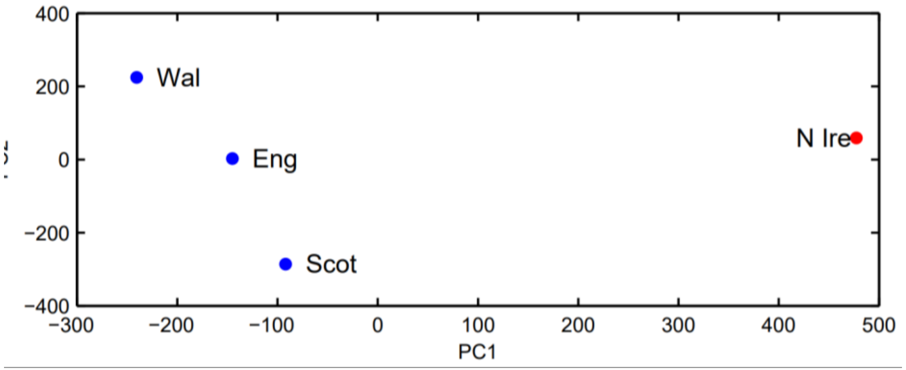






Coefficients of the Principal Components





Load plot shows the coefficients of the original feature vectors to the principal components

t-Distributed Stochastic Neighbor Embedding

(t-SNE)

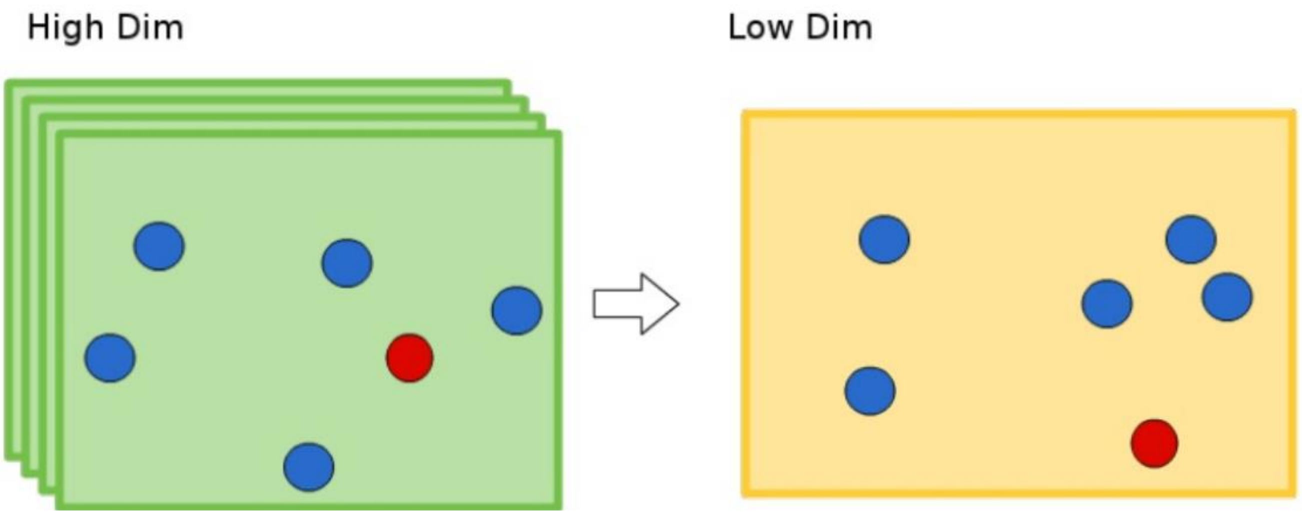
t-Distributed Stochastic Neighbor Embedding (t-SNE)

- Introduced by Laurens Van Der Maaten (2008)
- Generates a low dimensional representation of the high dimensional data set iteratively
- Aims to minimize the divergence between two distributions
  - Pairwise similarity of the points in the higher-dimensional space
  - Pairwise similarity of the points in the lower-dimensional space
- Output: original points mapped to a 2D or a 3D data space
  - similar objects are modeled by nearby points and
  - dissimilar objects are modeled by distant points with high probability
- Unlike PCA, it is stochastic (probabilistic)

t-SNE implementation I

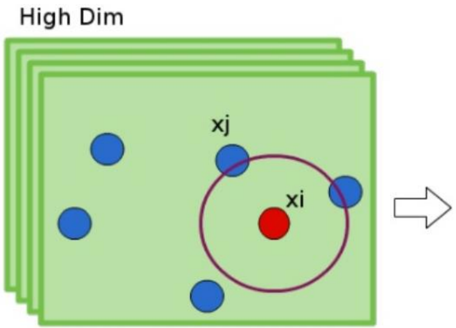
Step 1: *Generate the points in the low dimensional data set (2D or 3D)*

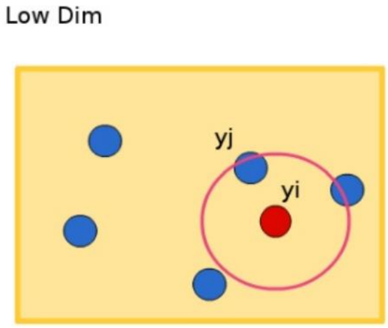
- random initialization
- First two or three components of PCA



t-SNE implementation II

Step 2: *Calculate the pair-wise similarities measures between data pairs (probability measure)*





The similarity of datapoint x_j to datapoint x_i means the conditional probability p_{ji} that x_i would pick x_j as its nearest neighbor.

$$p_{ij} = \frac{exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k\neq l} exp(-||x_l - x_k||^2/2\sigma^2)}$$

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k\neq l}(1 + ||y_k - y_l||^2)^{-1}}$$

Exponential normalization of the Euclidian distances are needed due to the high dimensionality.
(Curse of dimensionality)

t-SNE implementation III

Step 3: *Define the cost function*

- Similarity of data points in High dimension: $$p_{ij} = \frac{exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k \neq l} exp(-||x_l - x_k||^2/2\sigma^2)}$$
- Similarity of data points in Low dimension: $$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}}$$
- Cost function (called Kullback-Leiber divergence between the two distributions): $$C = KL(P||Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}}$$
  - Large p_{ji} modeled by small q_{ji} → <u>Large penalty</u>
  - Large p_{ji} modeled by large q_{ji} → <u>Small penalty</u>
  - Local similarities are preserved

t-SNE implementation IV

Step 4: *Minimize the cost function using gradient descent*

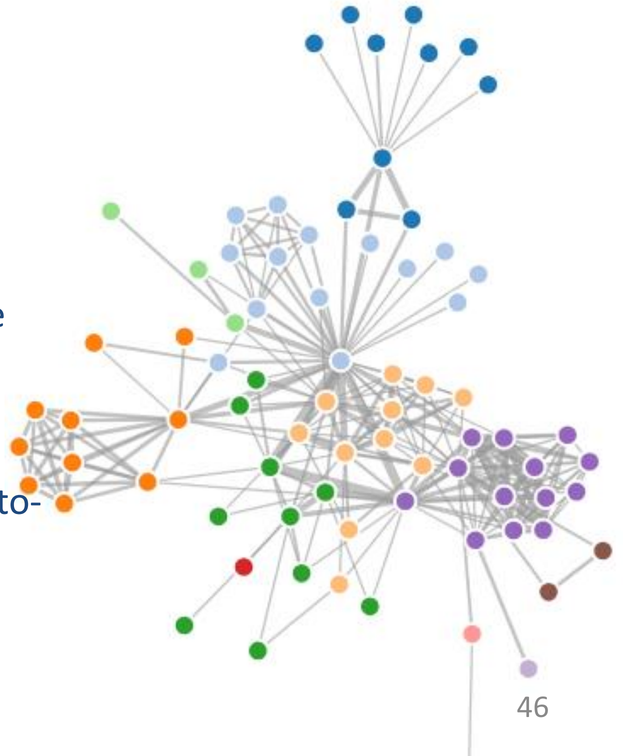
- Gradient has a surprisingly simple form:

$$\frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij})(1 + ||y_i - y_j||^2)^{-1}(y_i - y_j)$$

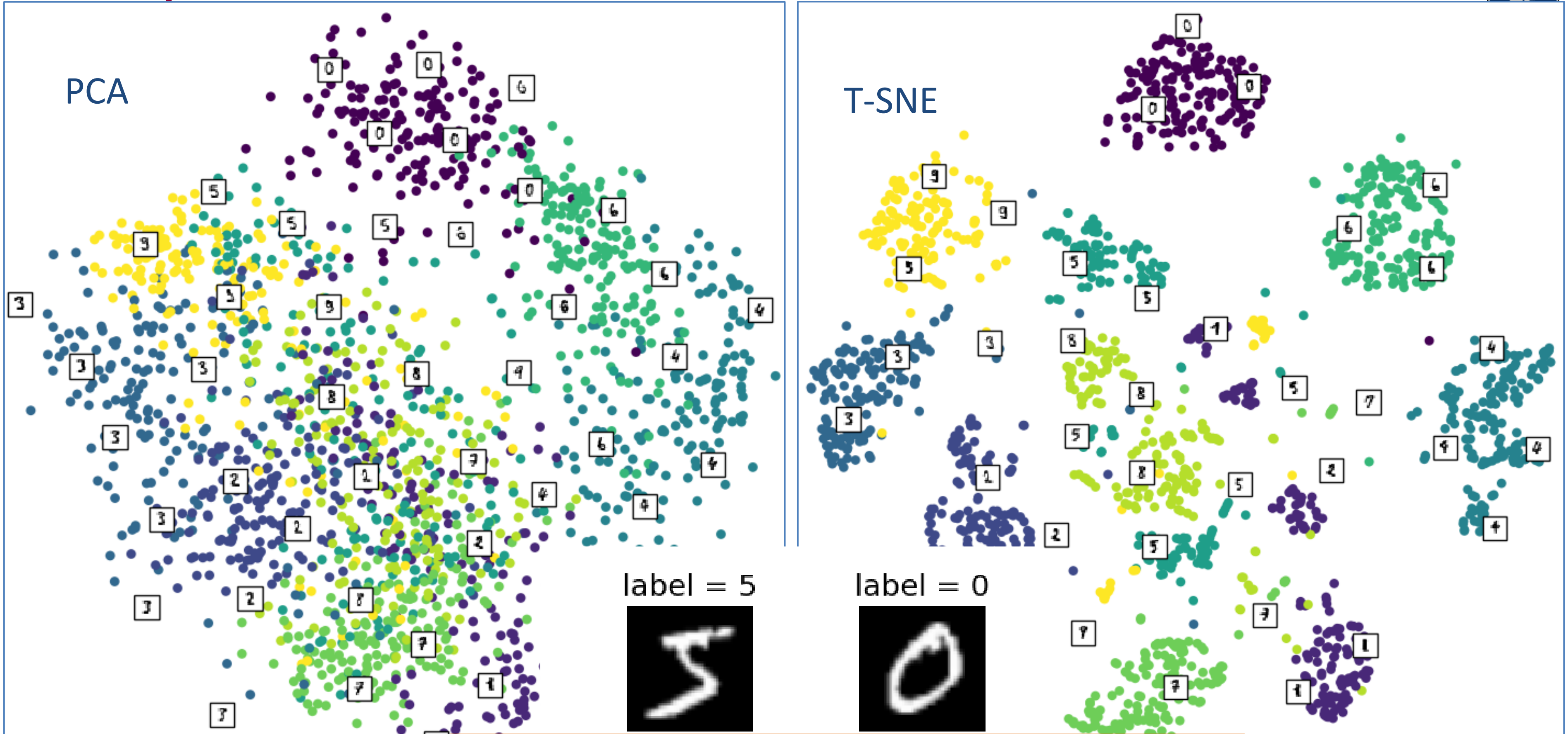
- Optimization can be done using momentum method

Physical analogy

- Our map points are all connected with springs in the low dimensional data map
- Stiffness of the springs depends on $p_{j|i}$ - $q_{j|i}$
- Let the system evolve according to the laws of physics
  - If two map points are far apart while the data points are close, they are attracted together
  - If they are nearby while the data points are dissimilar, they are repelled.
- Illustration (live)
  - https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm



Comparison of PCA and t-SNE on MNIST database

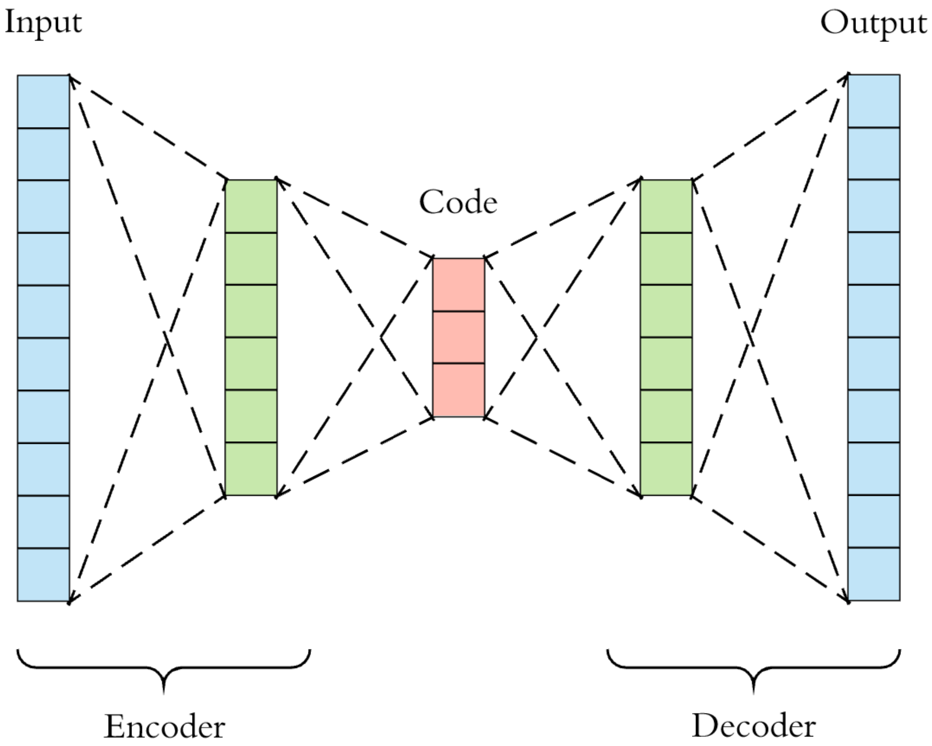


28x28 (784) dimensions → 2 dimensions

Autoencoder

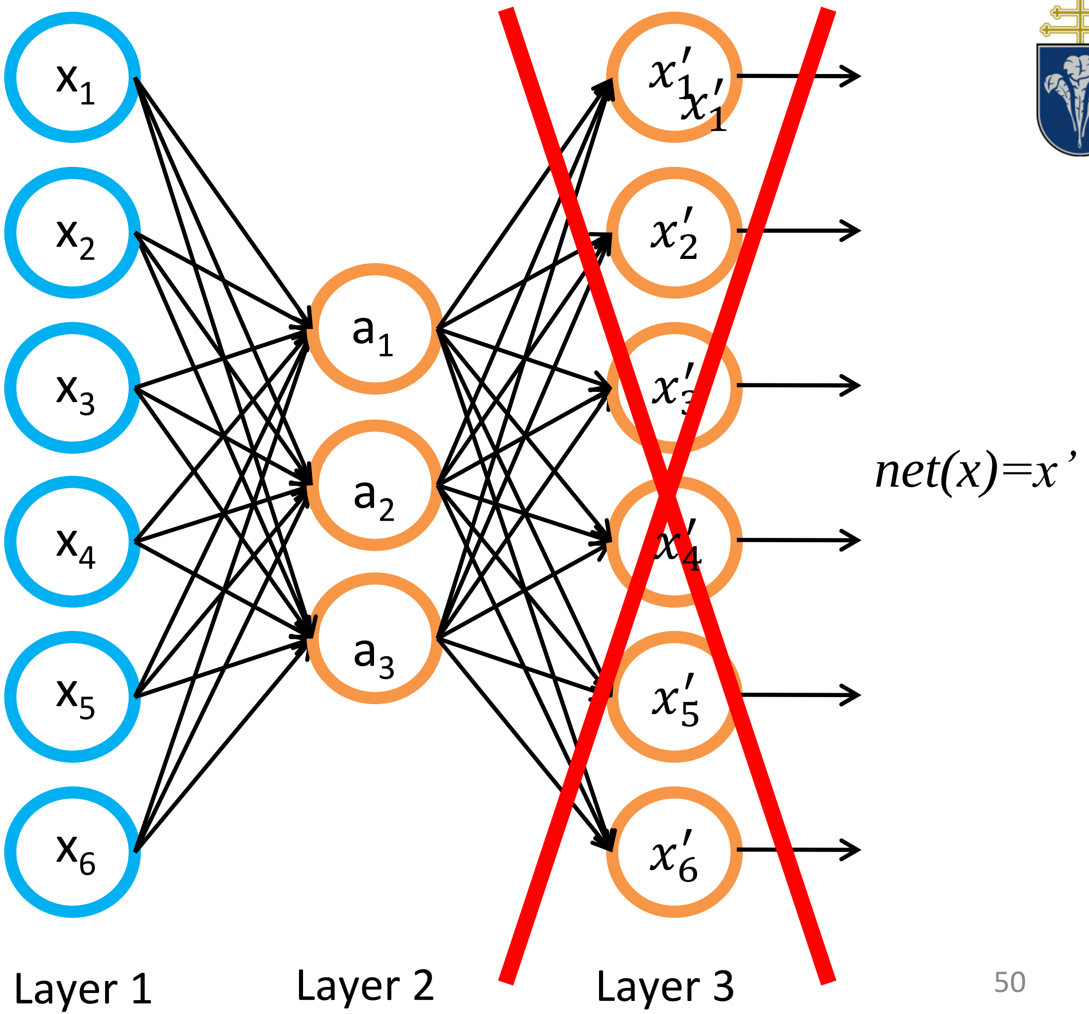
Autoencoder

- Neural network used for efficient data coding
- Uses the same vector for the input and the output
  - No labelled data set is needed
  - Unsupervised learning
- Two parts
  - Encoder: reduces data dimension
  - Decoder: reconstructs data
  - Middle layer: code



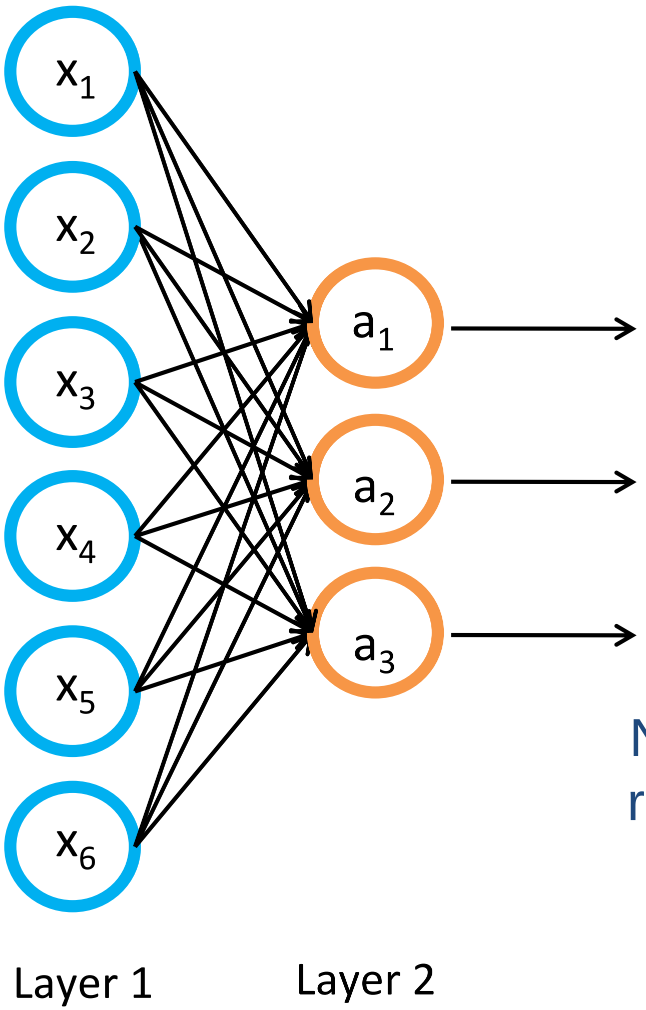
Operation

- The network is trained with the same input-output pairs
- Loss function:
  - MSE
  - Cross Entropy
- After network is trained, remove decoder part



Operation

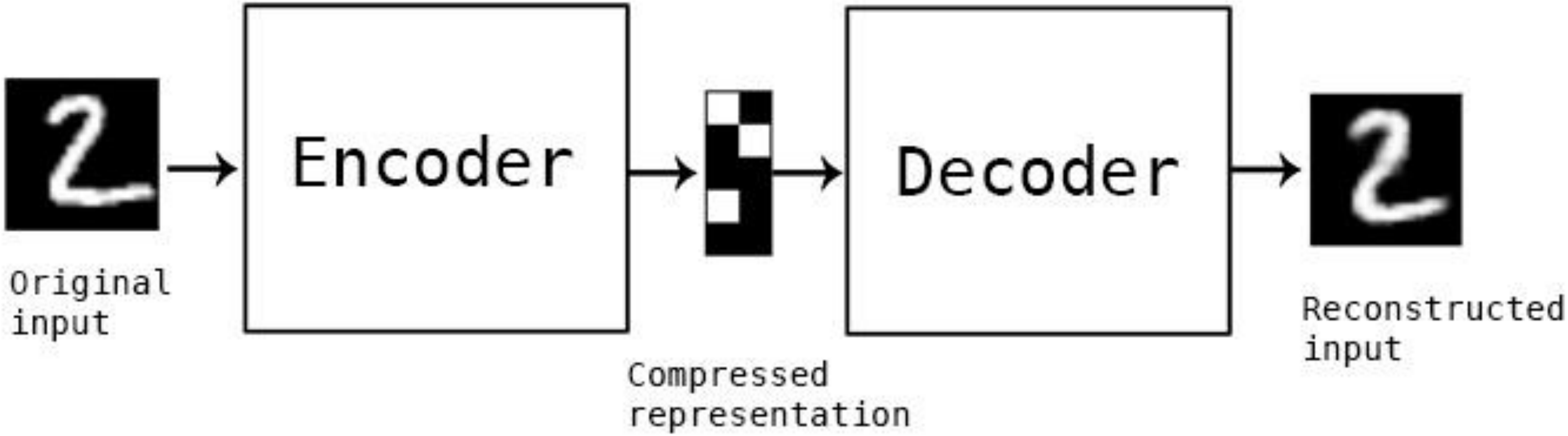
- The network is trained with the same input-output pairs
- Loss function:
  - MSE
  - Cross Entropy
- After network is trained, remove decoder part



$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

New compressed representation for input.

Example



- Coding MNIST data base
- 28x28 (784 dimensions) → 2x5 (10 dimensions)
- 78 times compression

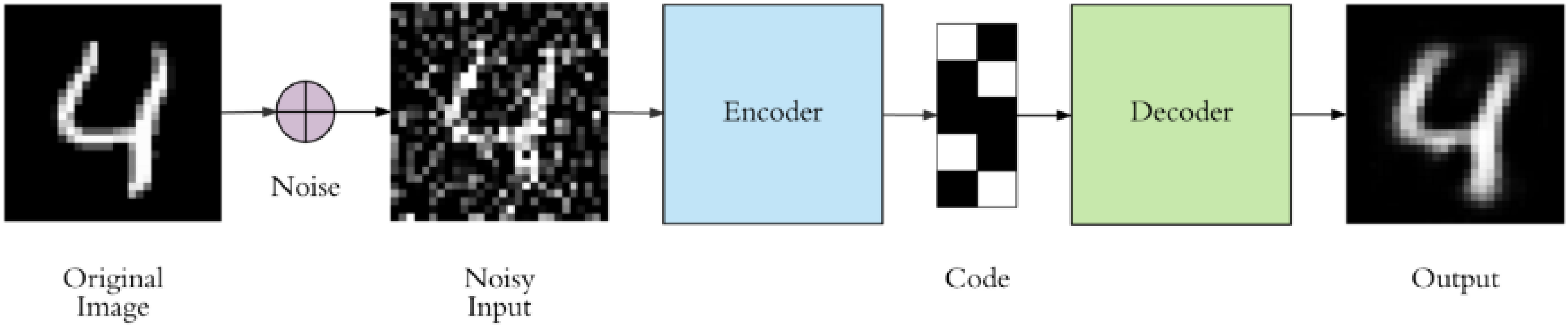
Autoencoder vs PCA

- Undercomplete autoencoder with
  - one hidden layer
  - linear output function
  - MSE loss
- Projects data on subspace of first K principal components

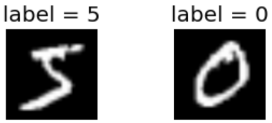
Undercomplete: width (dimension) of
hidden layer is smaller than
width input/output layer

Denoising

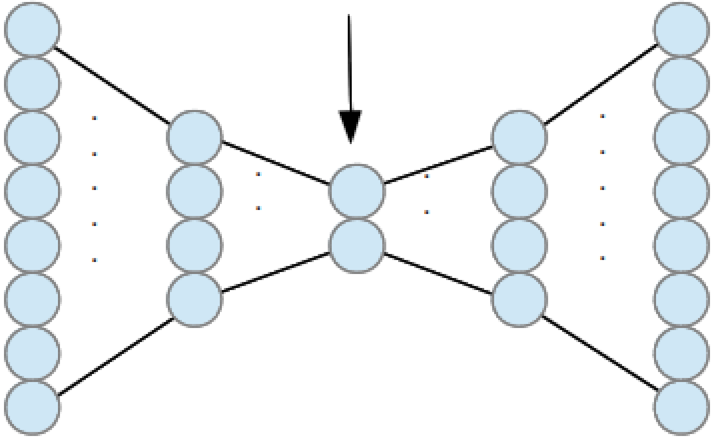
- Trick:
  - Adding noise to the input
  - The desired output is the original input

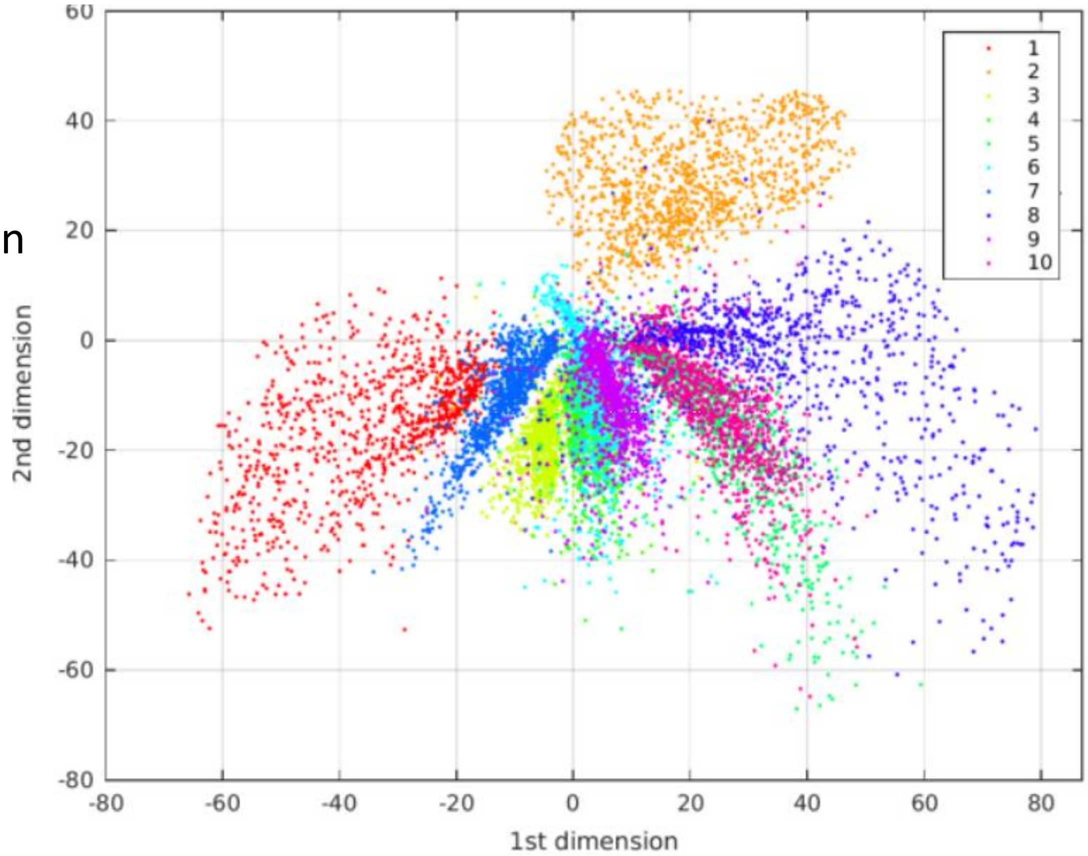


MNIST database coding to two dimension

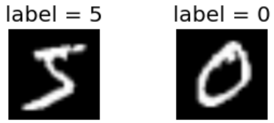


Two neurons in the coding hidden layer

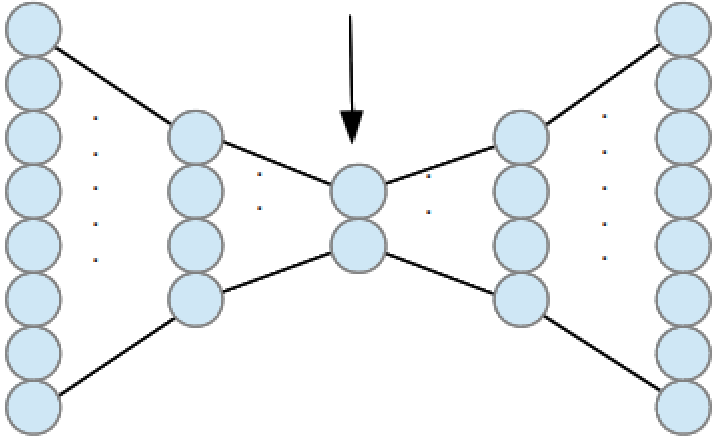


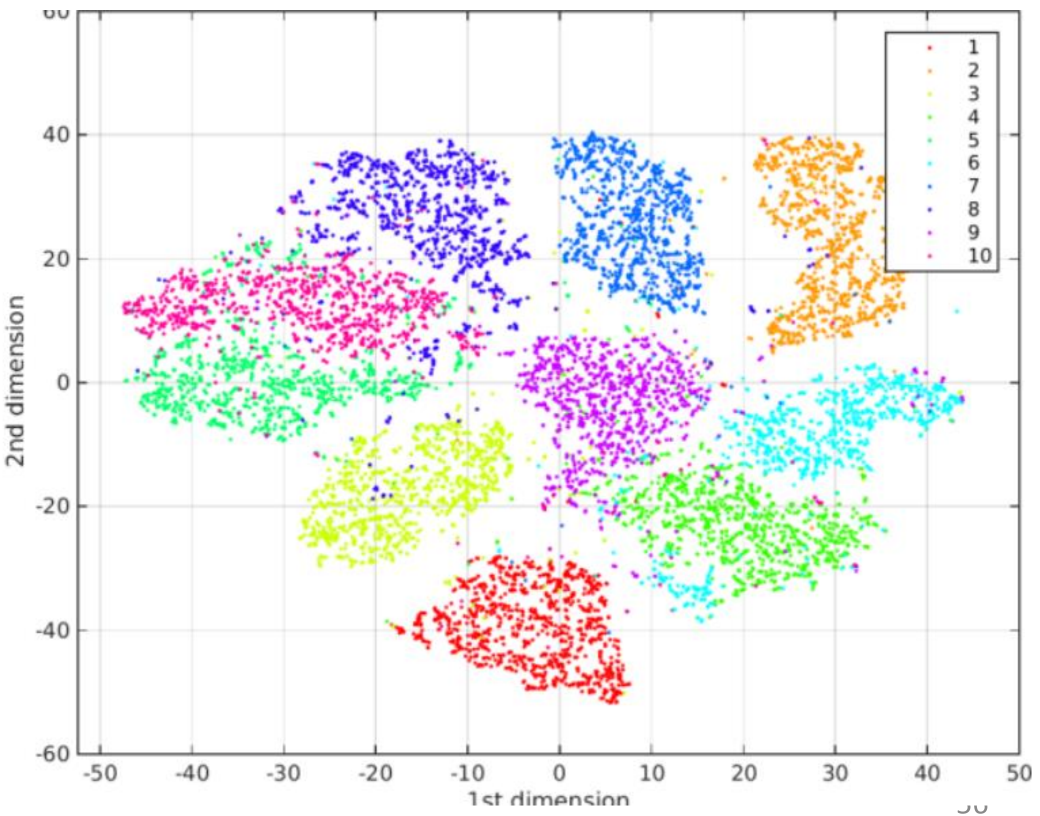


Autoencoder + t-SNE



Two neurons in the coding hidden layer





Recurrent Neural Networks

- How to handle sequential signals with Neural Networks?
- General Architecture of the Recurrent Networks

Static samples vs Data signal flow

AlexNet could recognize 1000s of images.
ResNet could reach better then human performance.

- Though human can recognize
  - Single letters
  - Single sounds
  - Single tunes
  - Single pictures

- But in real life we handle
  - Texts
  - Speech
  - Music
  - Movies

Story

(temporal analysis of sequential data)

Can feed-forward neural networks (perceptrons, conv. nets) solve these problems?

DATA MEMORY

Memory

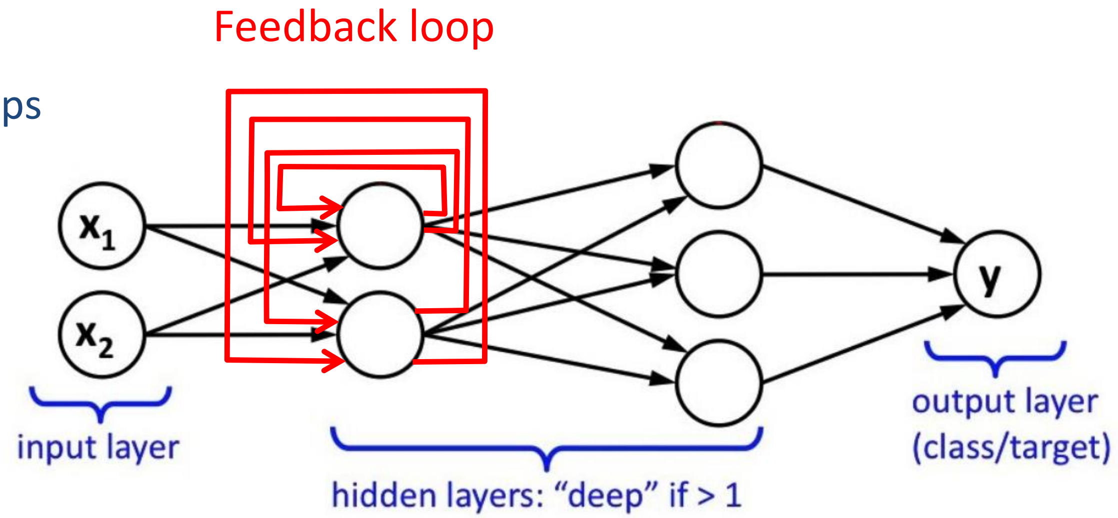
- Our feed-forward nets had so far
  - Program memory (for the weights)
  - Registers
    - For storing data temporally due to implementation and not matematical resasons
    - Registers were not part of the networks
- After each inferences the net was reset
  - All registers were deleted
  - No information remained in the net after processing an input vector
  - Therefore the order of a test sequence made no difference

Recurrent networks (RNN)

- Unlike traditional neural networks, the output of the RNN depends on the previous inputs
  - State
- RNN contains feedback
- Theoretically:
  - Directed graph with cyclic loops
- From now, time has a role in execution
  - Time steps, delays

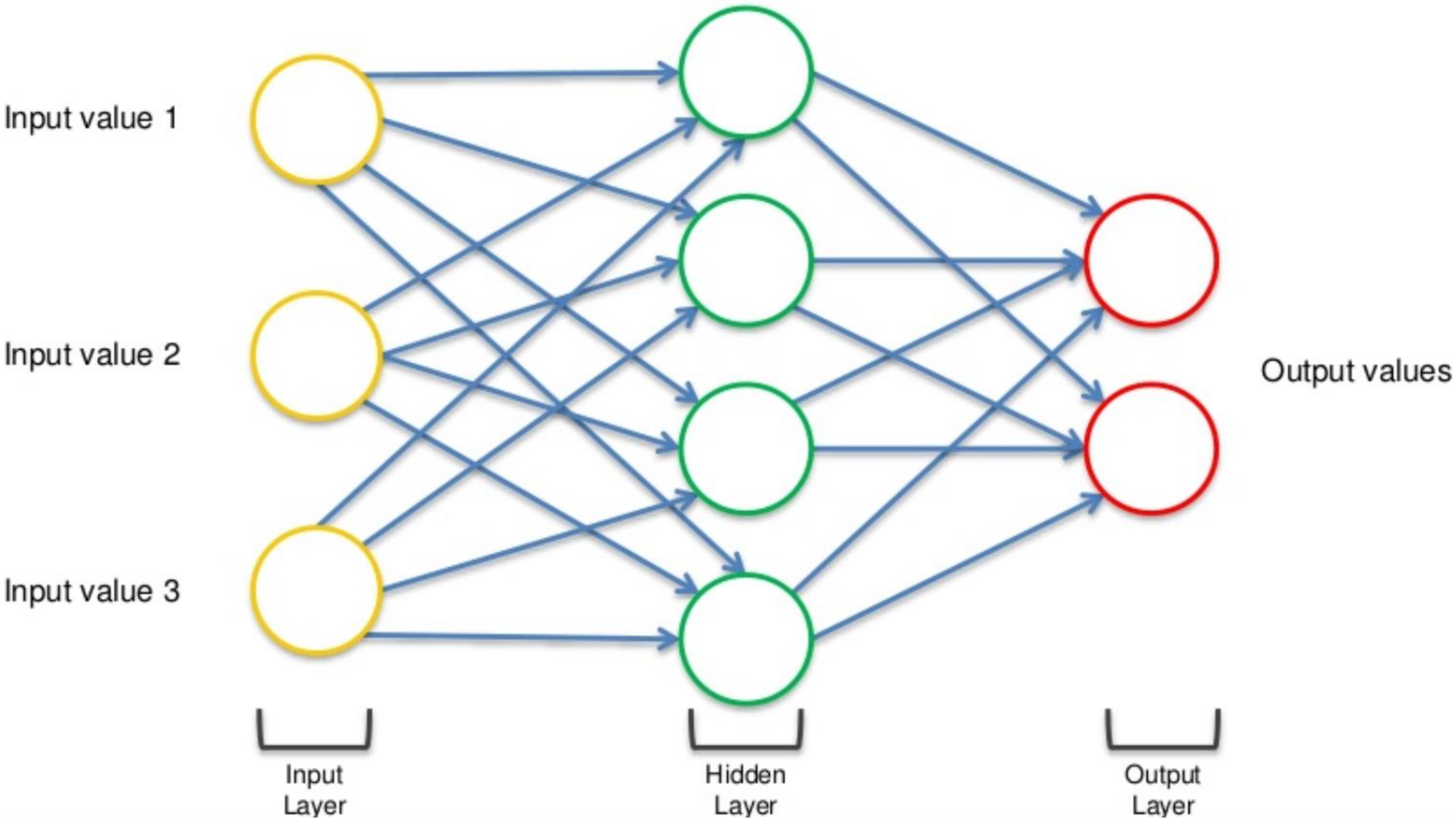
Jürgen lives in Berlin.

He speeks ....



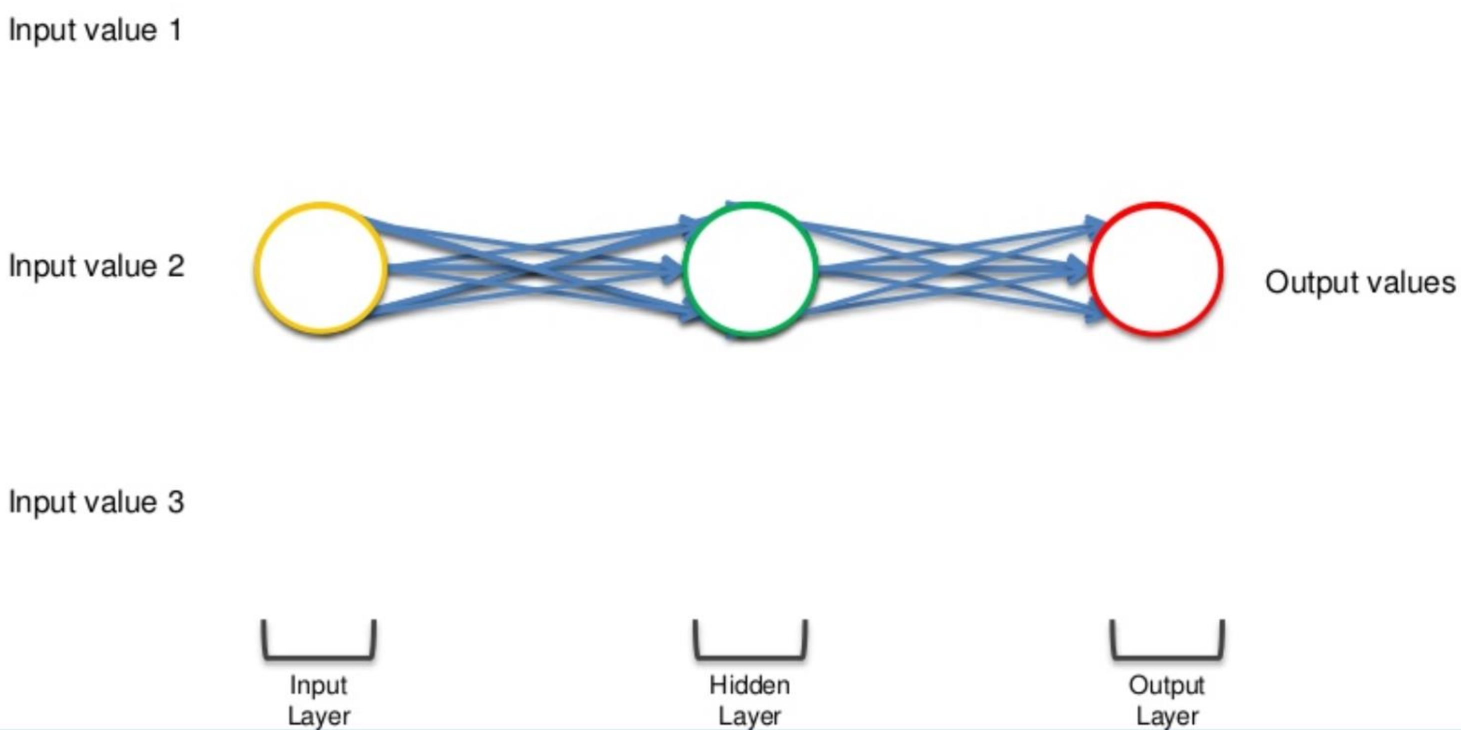
Steps towards vectorized data and parameters

- Weights (multiple arrows)



Steps towards vectorized data and parameters

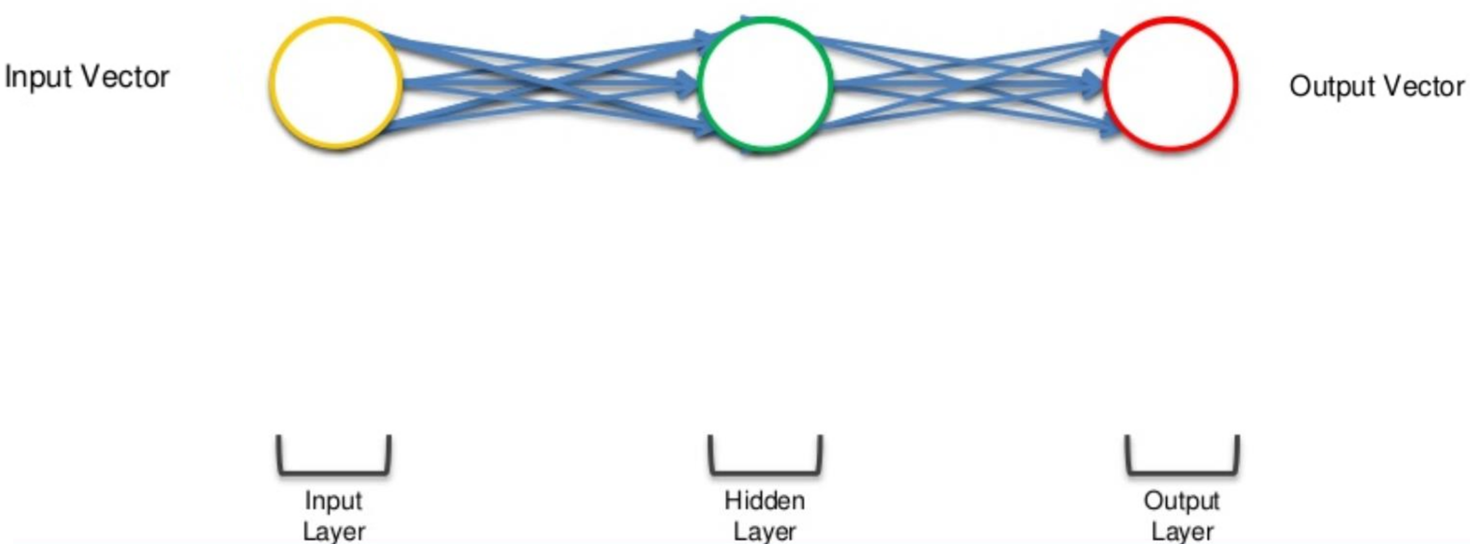
- Weights (multiple arrows)



Steps towards vectorized data and parameters

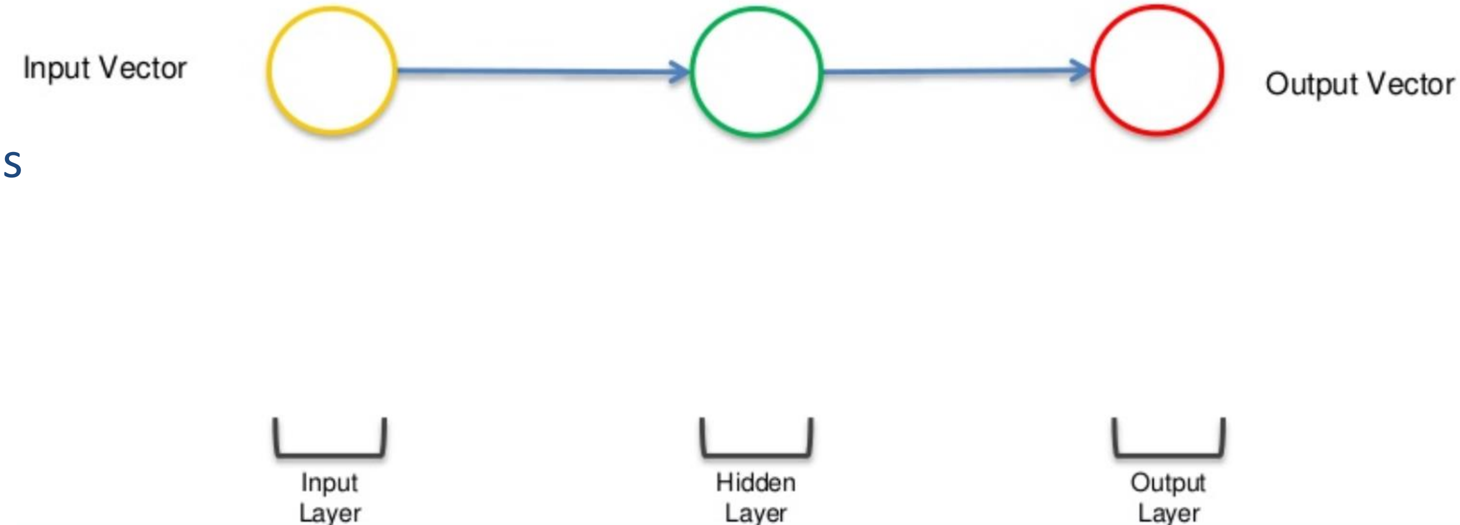
- Weights (multiple arrows)

  replaced with vectors (single arrows)



Steps towards vectorized data and parameters

- Single arrows indicate all interconnections between layers
- $\boldsymbol{w}_{ij}$ matrix matematically



Introducing feedback loop

$$h(0) = \begin{bmatrix} h_1(0) \\ \cdot \\ \cdot \\ \cdot \\ h_l(0) \end{bmatrix}$$

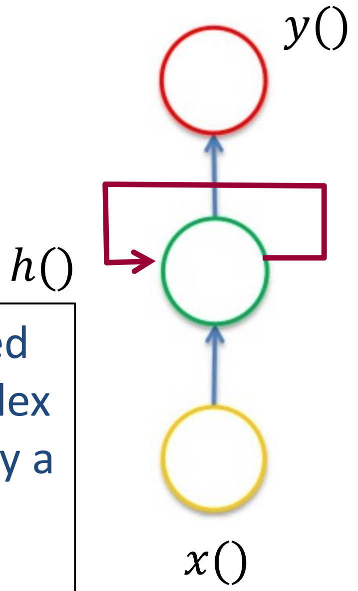
$$x(1) = \begin{bmatrix} x_1(1) \\ \cdot \\ \cdot \\ \cdot \\ x_k(1) \end{bmatrix}$$

$$\xrightarrow{\text{concatenation}} \quad c(1) = \begin{bmatrix} h_1(0) \\ \cdot \\ \cdot \\ h_l(0) \\ x_1(1) \\ \cdot \\ \cdot \\ x_k(0) \end{bmatrix}$$

$$h(1) = f\big(h(0), x(1)\big) = \mathbf{W}_{\boldsymbol{h}} c(1)$$

w: $l \times (k + l)$ *sized weight matrix*

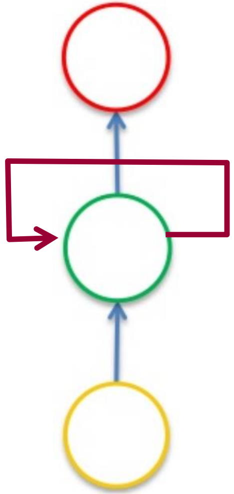
f() can be defined as a more complex function not only a matrix vector multiplication.

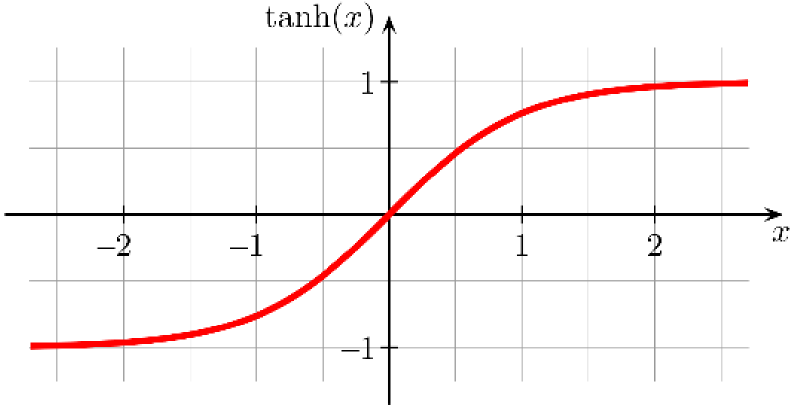


$$h(0) = 0$$

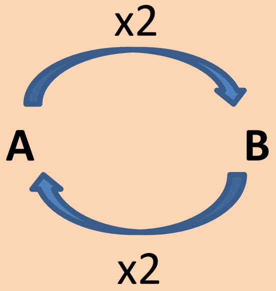
Activation function in feedback loop

- Activation function of the hidden layers is typically hyperbolic tangent
- It avoids large positive feedback
  - Keeps the output between -1 and +1
  - Avoids exploding the loop calculation
  - Gain should be smaller than 1 in the loop!





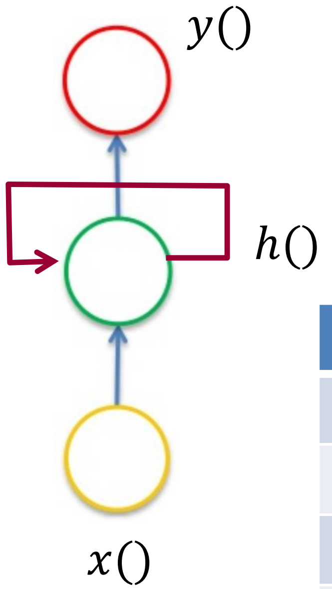
Positive feedback in a loop:
A *produces more of* ***B*** *which in turn produces more of* ***A.***
It leeds to increase beyond any limit.



Timing of the RNN

- Discrete time steps are used
- Input vector sequence to apply
- Signals are calculated in a node, when all inputs exist
- State machine

Time	Input	State	output
t=1	$x(1)$	$h(1) = f\big(h(0), x(1)\big)$	$y(1) = g\big(h(1)\big)$
t=2	$x(2)$	$h(2) = f\big(h(1), x(2)\big)$	$y(2) = g\big(h(2)\big)$
t=3	$x(3)$	$h(3) = f\big(h(2), x(3)\big)$	$y(3) = g\big(h(3)\big)$
t=4	$x(4)$	$h(4) = f\big(h(3), x(4)\big)$	$y(4) = g\big(h(4)\big)$
		. . .	

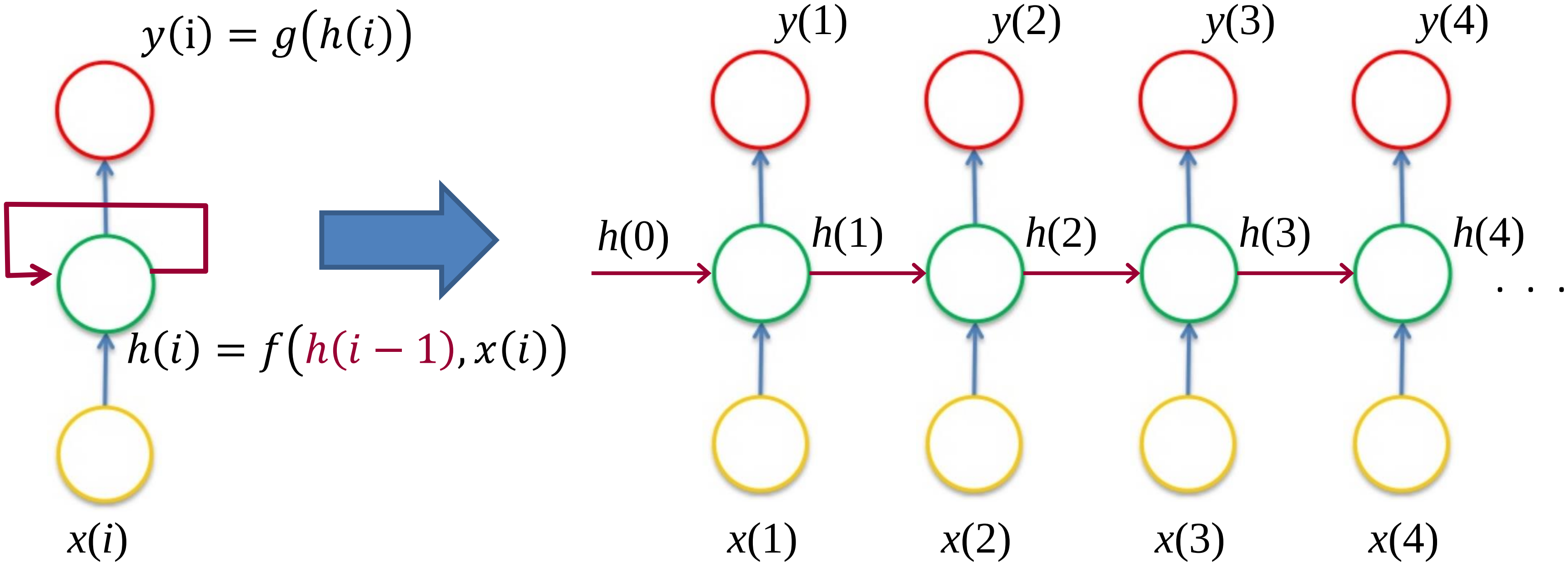


$h(0) = 0$

input
$x(1)$
$x(2)$
$x(3)$
$x(4)$
…

How to calculate back propagation?

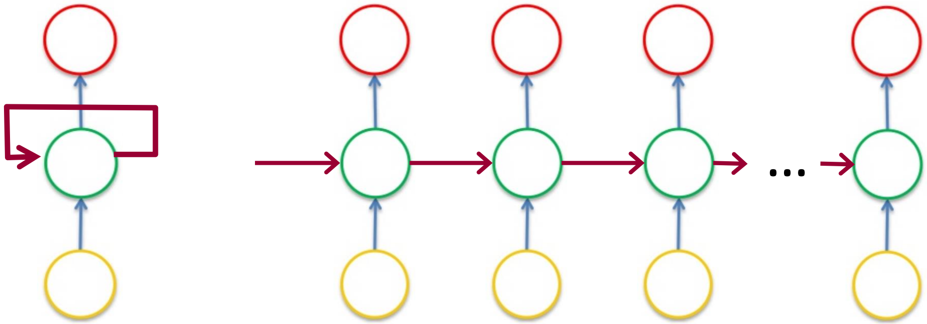
Unrolling



$y(\mathrm{i}) = g\big(h(i)\big)$
$h(i) = f\big(h(i-1), x(i)\big)$
$x(i)$
$y(1)$
$y(2)$
$y(3)$
$y(4)$
$h(0)$
$h(1)$
$h(2)$
$h(3)$
$h(4)$
$. \; . \; .$
$x(1)$
$x(2)$
$x(3)$
$x(4)$

Unrolling

- Unrolling generates an acyclic directed graph from the original cyclic directed graph structure
- It generates a final impulse response (FIR) filter from the original infinite impulse response (IIR) filter
- Dynamic behavior



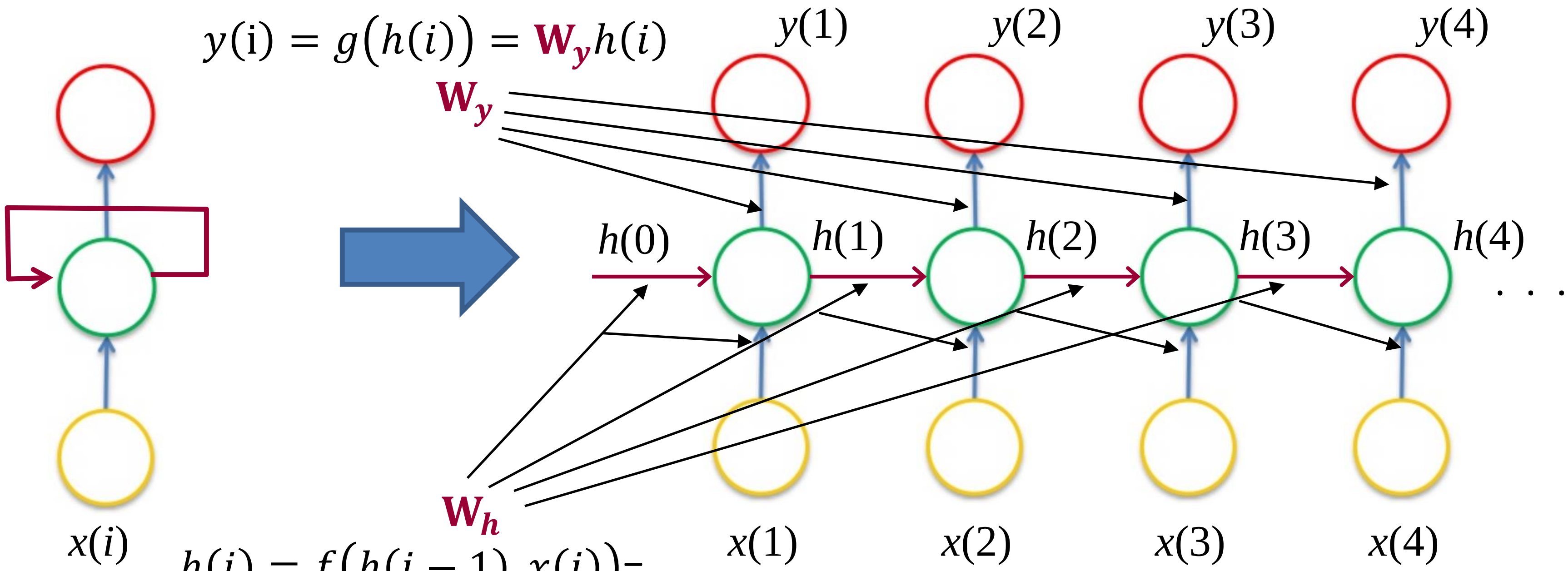
IIR filters may response to any finite length input with a infinite (usually decaying) response, due to their internal loop.

FIR filters response to any finite length input with a final response.

Weight matrix sharing

RNN re-uses the same weight matrix in every unrolled steps.

$$y(\mathrm{i}) = g\big(h(i)\big) = \mathbf{W_y} h(i)$$



$$h(i) = f\big(h(i-1), x(i)\big) = \\ = \mathbf{W_h}\, c(1)$$

Neural Networks

Recurrent Neural networks, LSTM

(P-ITEEA-0011)

Akos Zarandy
Lecture 10
November 26, 2019

Contents

- How to handle sequential signals with Neural Networks?
- Recurrent Networks
  - Training
  - Examples
  - Vanishing gradient problem
- Long Short Term Memory (LSTM)
  - LSTM versions

Static samples vs Data signal flow

AlexNet could recognize 1000s of images.
ResNet could reach better then human performance.

- Though human can recognize
  - Single letters
  - Single sounds
  - Single tunes
  - Single pictures

- But in real life we handle
  - Texts
  - Speech
  - Music
  - Movies

Story
(temporal analysis of sequential data)

Can feed forward neural networks (perceptrons, conv. nets) solve these problems?

Naturally, we can extend the data dimension with the time, but this leads to data size and computational load explosion .

DATA MEMORY

Memory

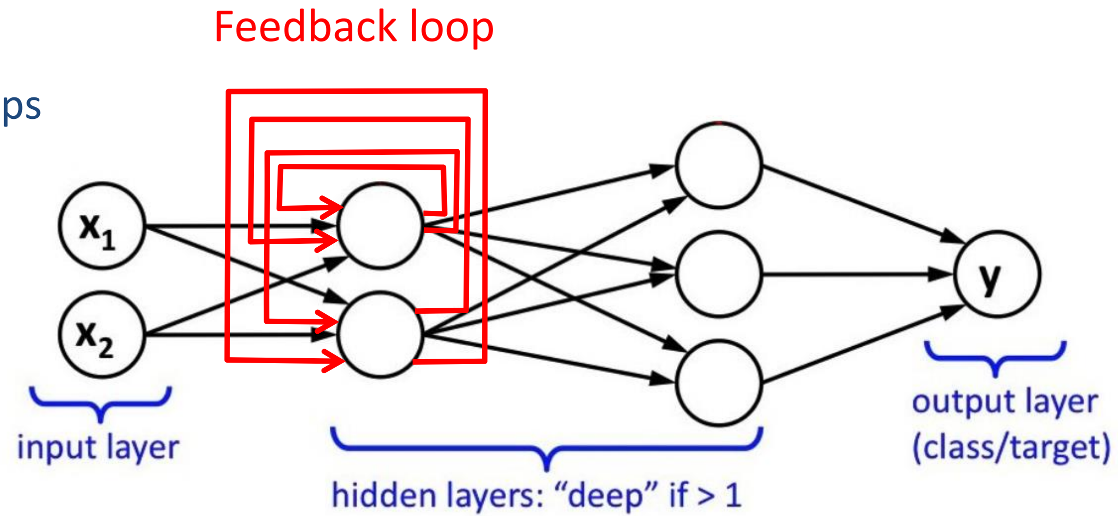
- Our feed-forward nets had so far
  - Program memory (for the weights)
  - Registers
    - For store temporally due to implementation and not matematical resasons
    - Registers were not part of the networks
- After each inferences the net was reset
  - All registers were deleted
  - No information remained in the net after processing an input vector
  - Therefore the order of a test sequence made no difference

Recurrent networks (RNN)

- Unlike traditional neural networks, the output of the RNN depends on the previous inputs
  - State
- RNN contains feedback
- Theoretically:
  - Directed graph with cyclic loops
- From now, time has a role in execution
  - Time steps, delays

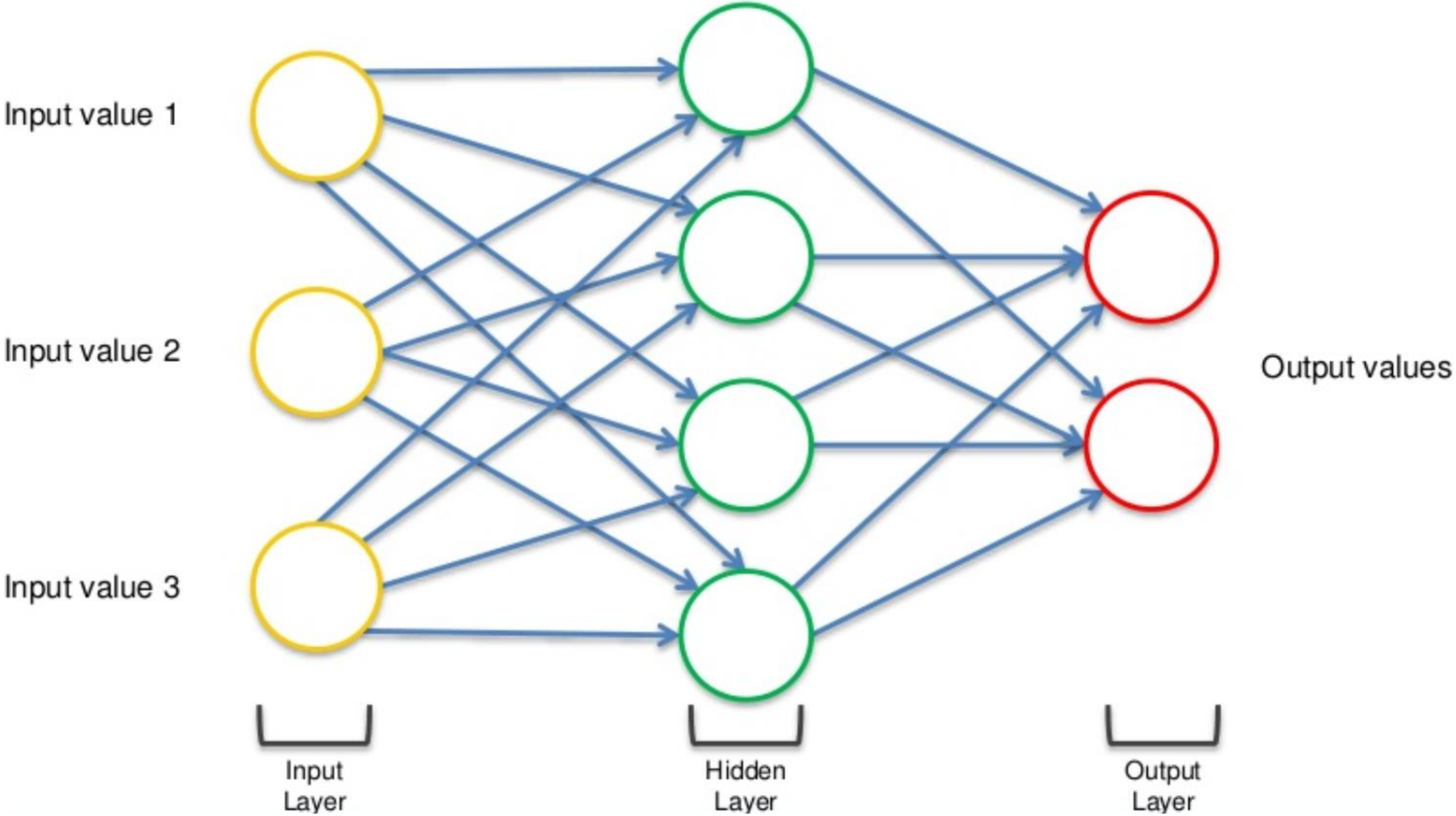
Jürgen lives in Berlin.

He speeks ....



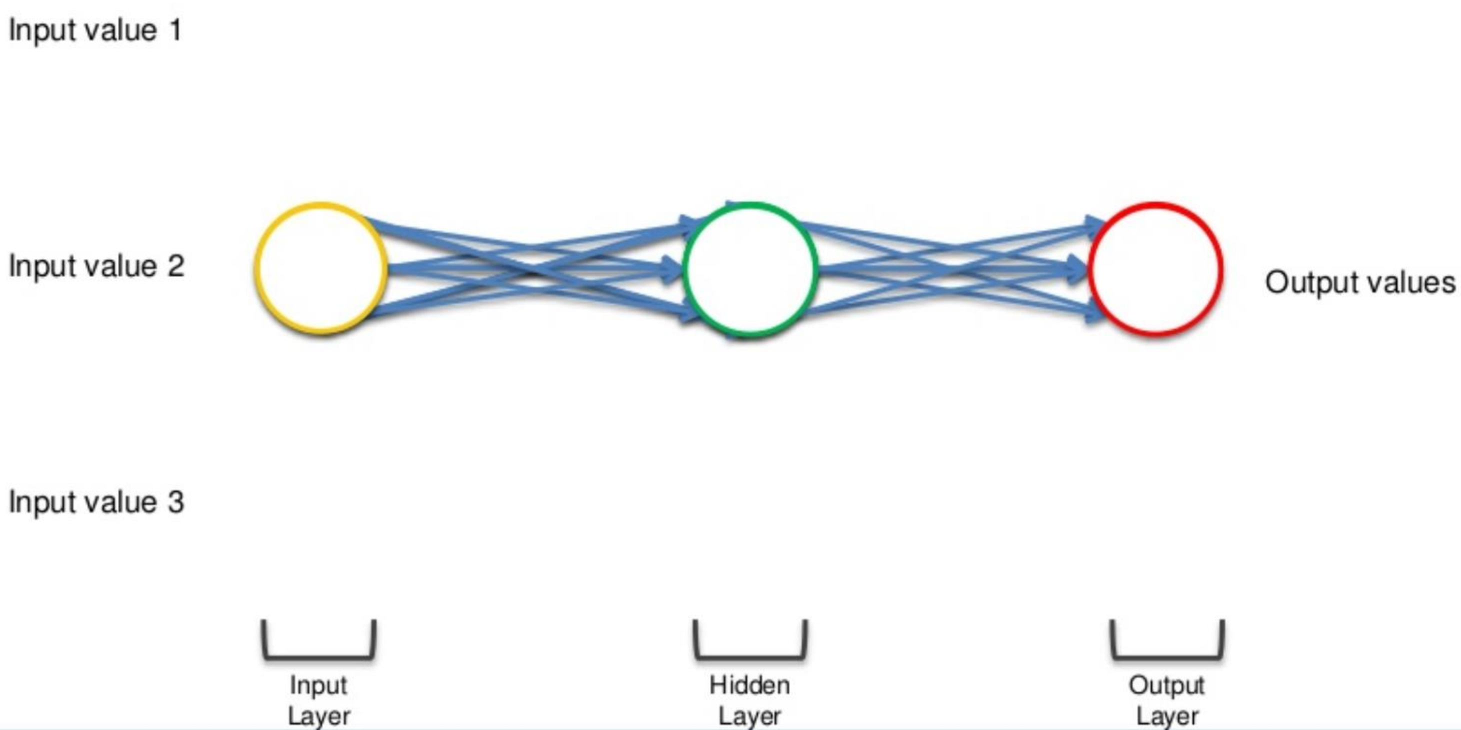
Vectorized presentation of neurons and parameters

- Weights (multiple arrows)



Vectorized presentation of neurons and parameters

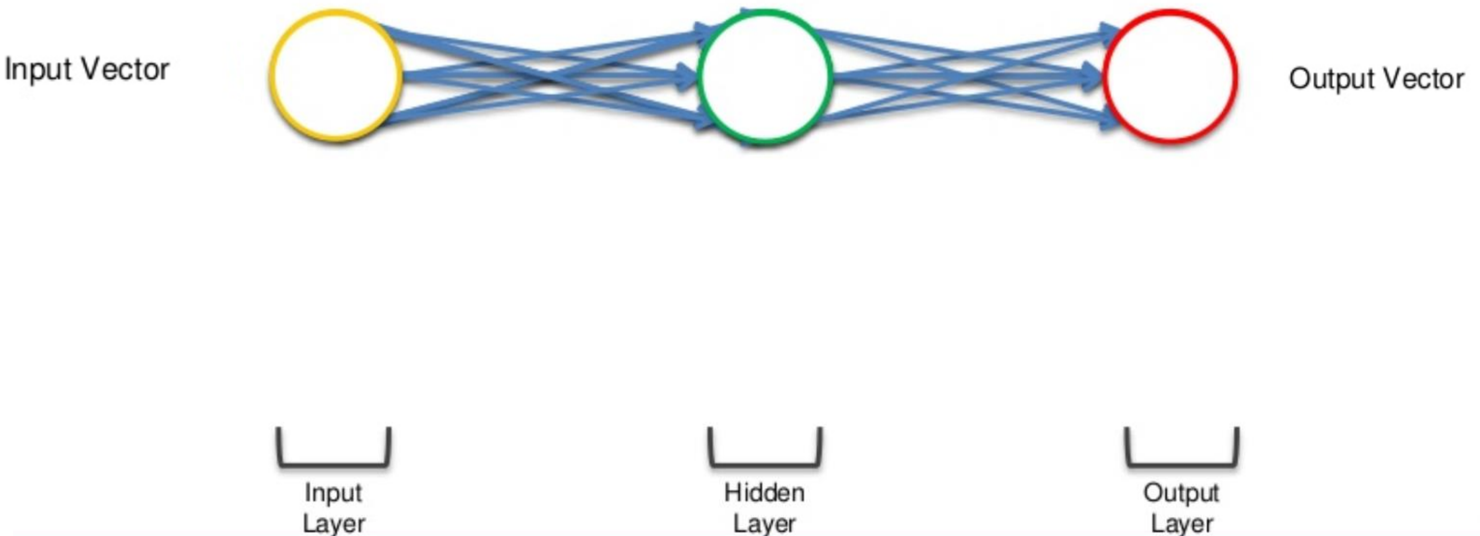
- Weights (multiple arrows)



Vectorized presentation of neurons and parameters

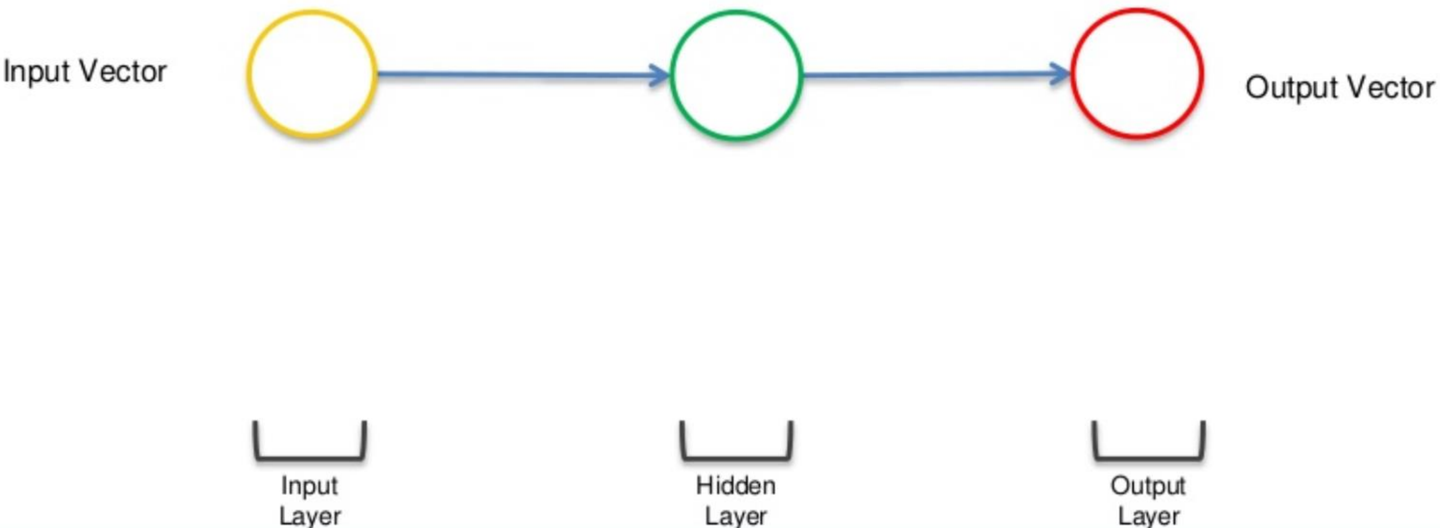
- Weights (multiple arrows)

  replaced with vectors (single arrows)



Vectorized presentation of neurons and parameters

- Single arrows indicate all interconnections between layers
- $\boldsymbol{w}_{ij}$ matrix matematically



Introducing feedback loop

$$h(0) = \begin{bmatrix} h_1(0) \\ \cdot \\ \cdot \\ \cdot \\ h_l(0) \end{bmatrix}$$

$$x(1) = \begin{bmatrix} x_1(1) \\ \cdot \\ \cdot \\ \cdot \\ x_k(1) \end{bmatrix}$$

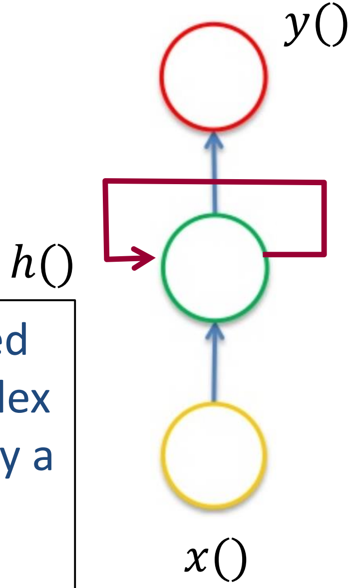
$\xrightarrow{\text{concatenation}}$

$$c(1) = \begin{bmatrix} h_1(0) \\ \cdot \\ \cdot \\ h_l(0) \\ x_1(1) \\ \cdot \\ \cdot \\ x_k(0) \end{bmatrix}$$

$$h(1) = f\big(h(0), x(1)\big) = \mathbf{W}_{\boldsymbol{h}} c(1)$$

$$w:\ l \ \times\ (k+l)\ \textit{sized weight matrix}$$

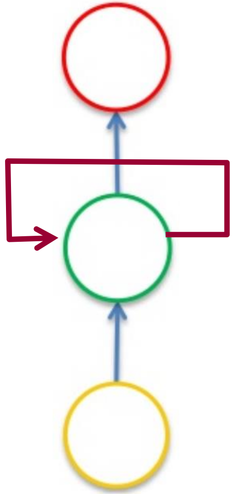
f() can be defined as a more complex function not only a matrix vector multiplication.

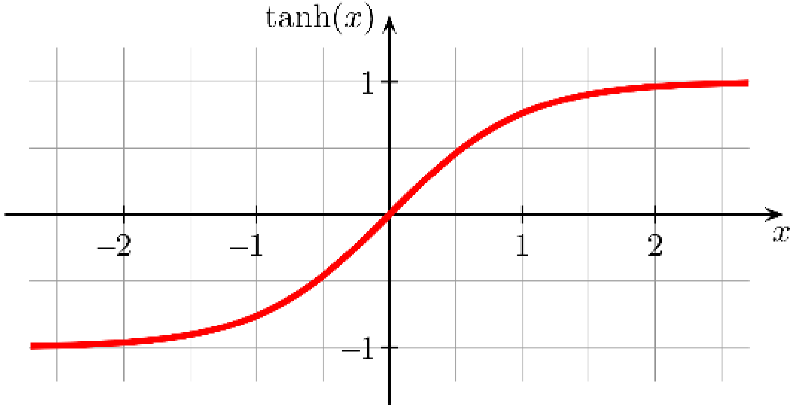


$$h(0) = 0$$

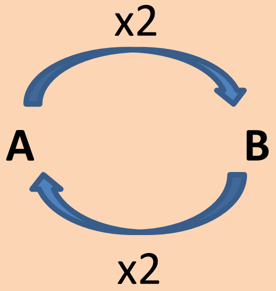
Activation function in feedback loop

- Activation function of the hidden layers is typically hyperbolic tangent
- It avoids large positive feedback
  - Keeps the output between -1 and +1
  - Avoids exploding the loop calculation
  - Gain should be smaller than 1 in the loop!





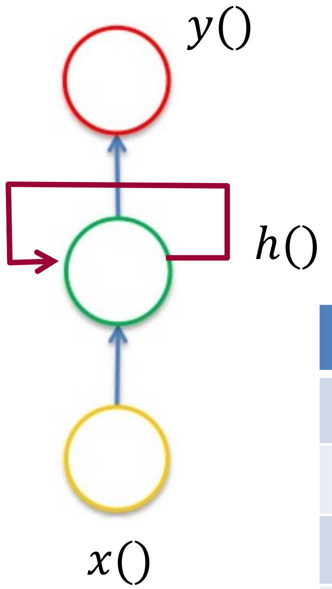
Positive feedback in a loop:
***A** produces more of **B** which in turn produces more of **A**.*
It leeds to increase beyond any limit.



Timing of the RNN

- Discrete time steps are used
- Input vector sequence to apply
- Signals are calculated in a node, when all inputs exist
- State machine

Time	Input	State	output
t=1	$x(1)$	$h(1) = f\big(h(0), x(1)\big)$	$y(1) = g\big(h(1)\big)$
t=2	$x(2)$	$h(2) = f\big(h(1), x(2)\big)$	$y(2) = g\big(h(2)\big)$
t=3	$x(3)$	$h(3) = f\big(h(2), x(3)\big)$	$y(3) = g\big(h(3)\big)$
t=4	$x(4)$	$h(4) = f\big(h(3), x(4)\big)$	$y(4) = g\big(h(4)\big)$
		. . .	

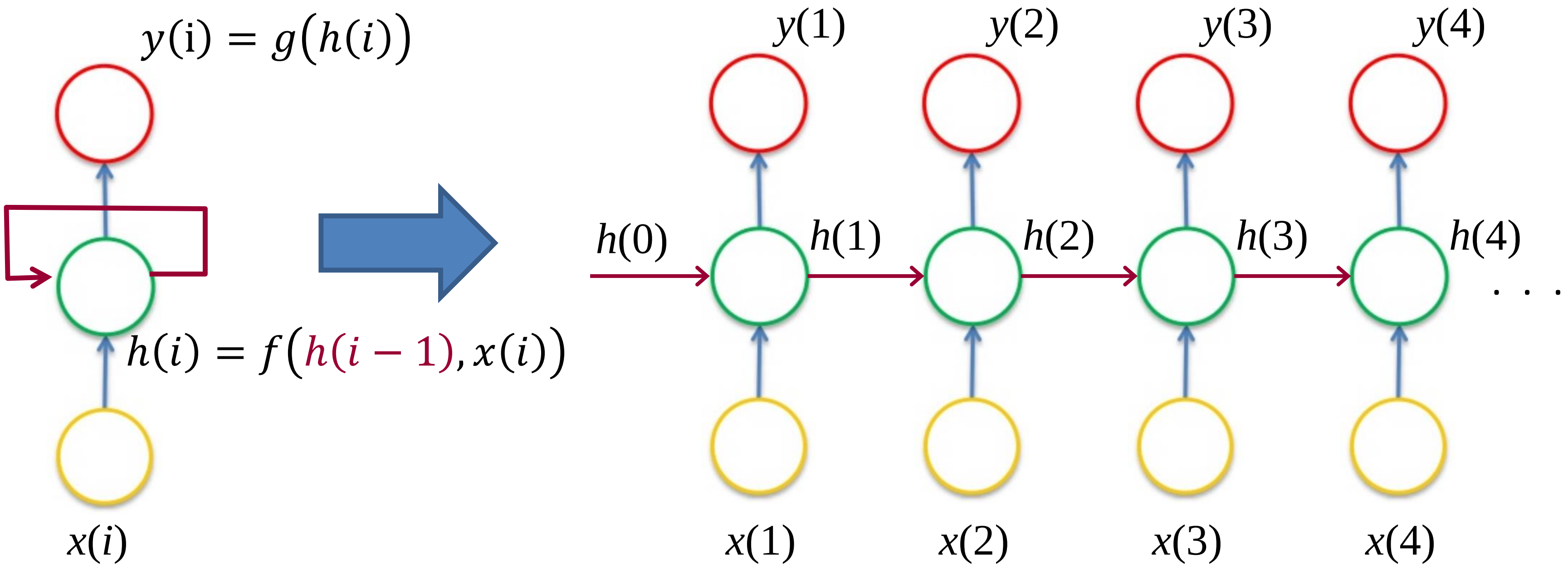


$h(0) = 0$

input
$x(1)$
$x(2)$
$x(3)$
$x(4)$
…

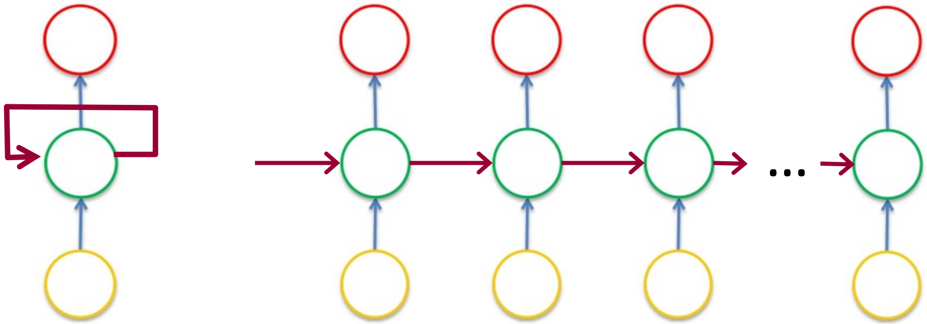
How to calculate back propagation?

Unrolling


$y(\mathrm{i}) = g\big(h(i)\big)$
$h(i) = f\big(h(i-1), x(i)\big)$
$x(i)$
$y(1)$
$y(2)$
$y(3)$
$y(4)$
$h(0)$
$h(1)$
$h(2)$
$h(3)$
$h(4)$
. . .
$x(1)$
$x(2)$
$x(3)$
$x(4)$

Unrolling

- Unrolling generates an acyclic directed graph from the original cyclic directed graph structure
- It generates a final impulse response (FIR) filter from the original infinite impulse response (IIR) filter
- Dynamic behavior

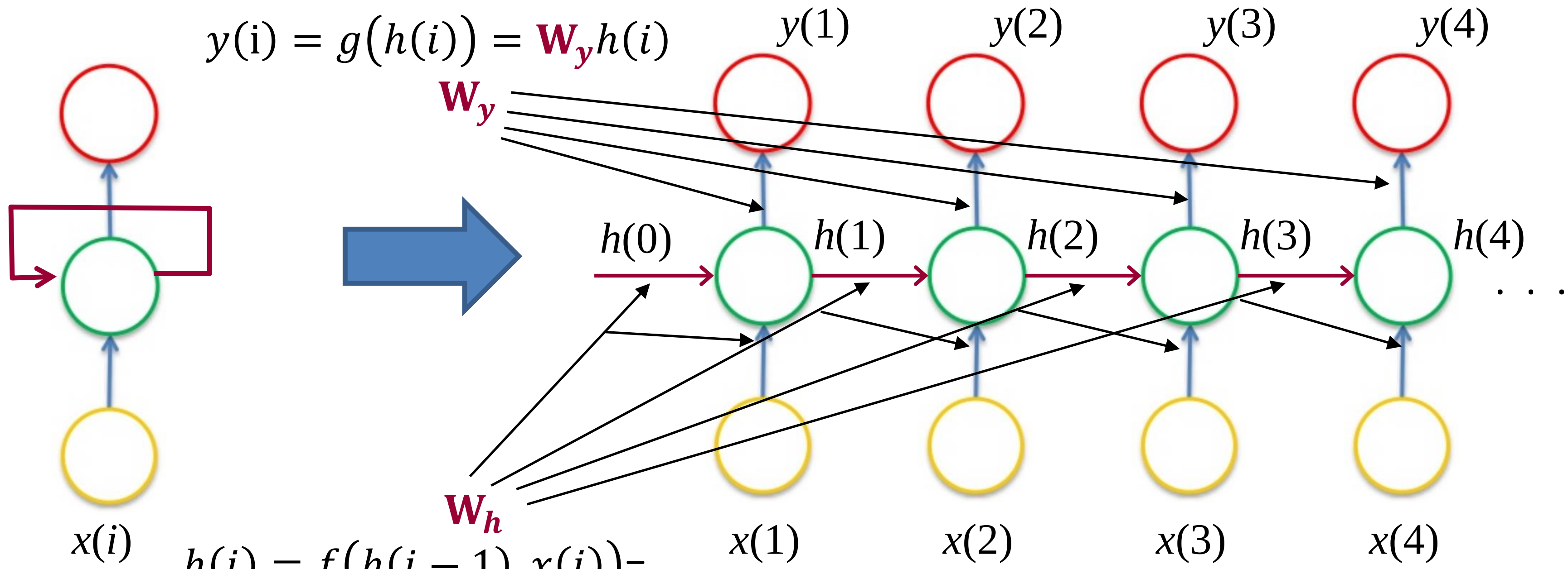


IIR filters may response to any finite length input with a infinite (usually decaying) response, due to their internal loop.

FIR filters response to any finite length input with a final response.

Weight matrix sharing

RNN re-uses the same weight matrix in every unrolled steps.



$$h(i) = f\big(h(i-1), x(i)\big) =$$
$$= \mathbf{W_h}\, c(1)$$

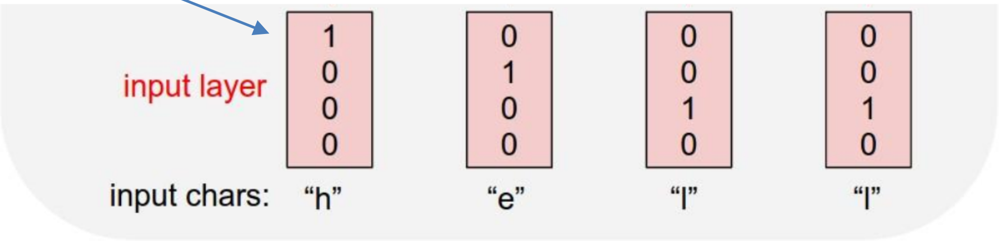
Simple RNN Training Example: Predicting the next letter

Example: Character-level Language Model

Vocabulary: [h,e,l,o]

One-hot encoding

Example training sequence: **“hello”**



http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

Simple RNN Training Example: Predicting the next letter

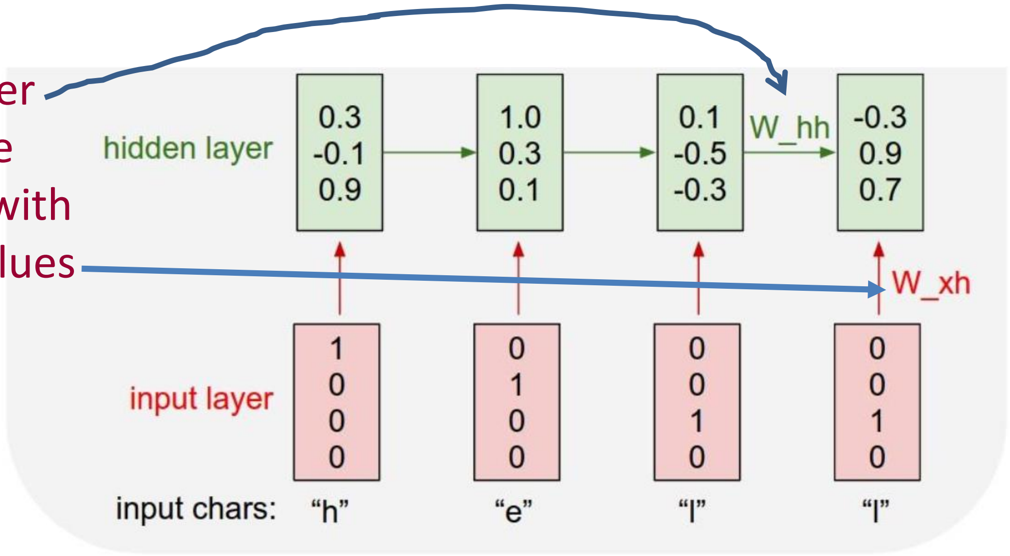
Example: Character-level Language Model

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Hidden layer weights are initialized with random values



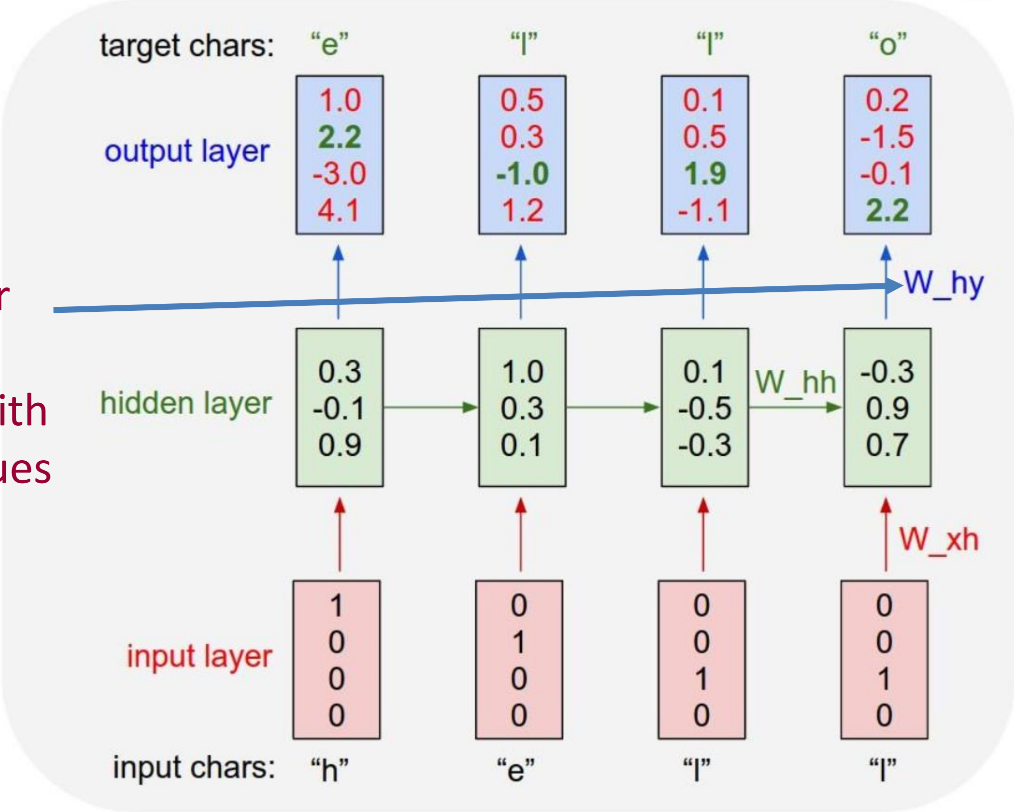
Simple RNN Training Example: Predicting the next letter

Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

Output layer weights are initialized with random values

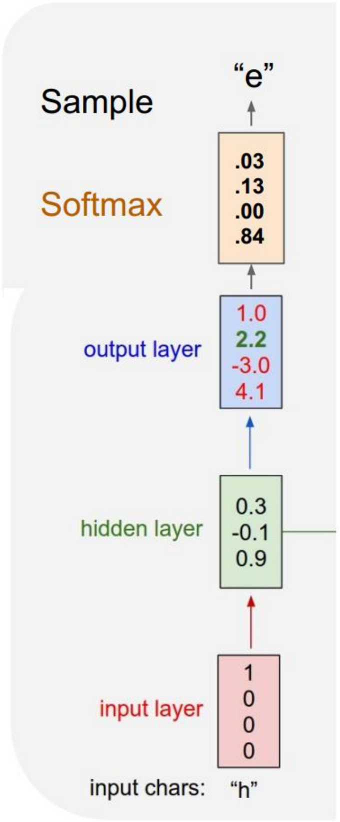


Simple RNN Training Example: Predicting the next letter

Example:
Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

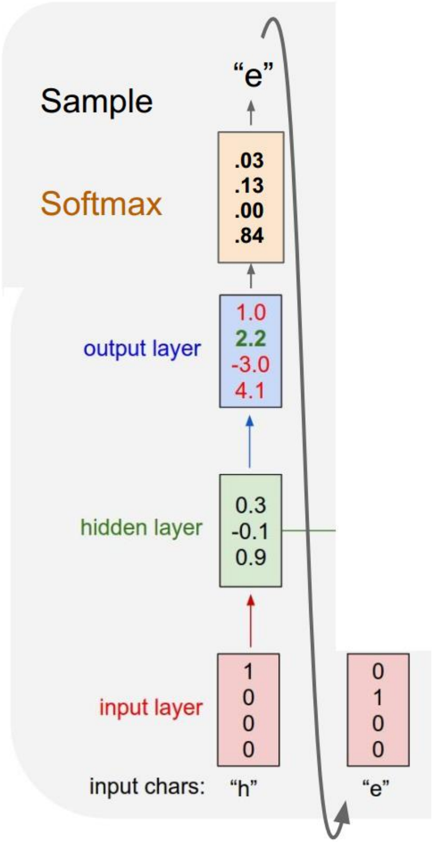


Simple RNN Training Example: Predicting the next letter

Example:
Character-level
Language Model
Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

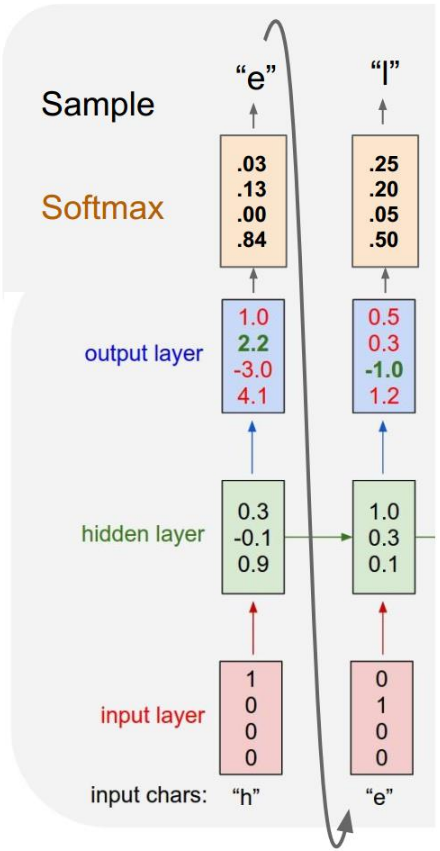


Simple RNN Training Example: Predicting the next letter

Example: Character-level Language Model Sampling

Vocabulary: [h,e,l,o]

At test-time sample characters one at a time, feed back to model



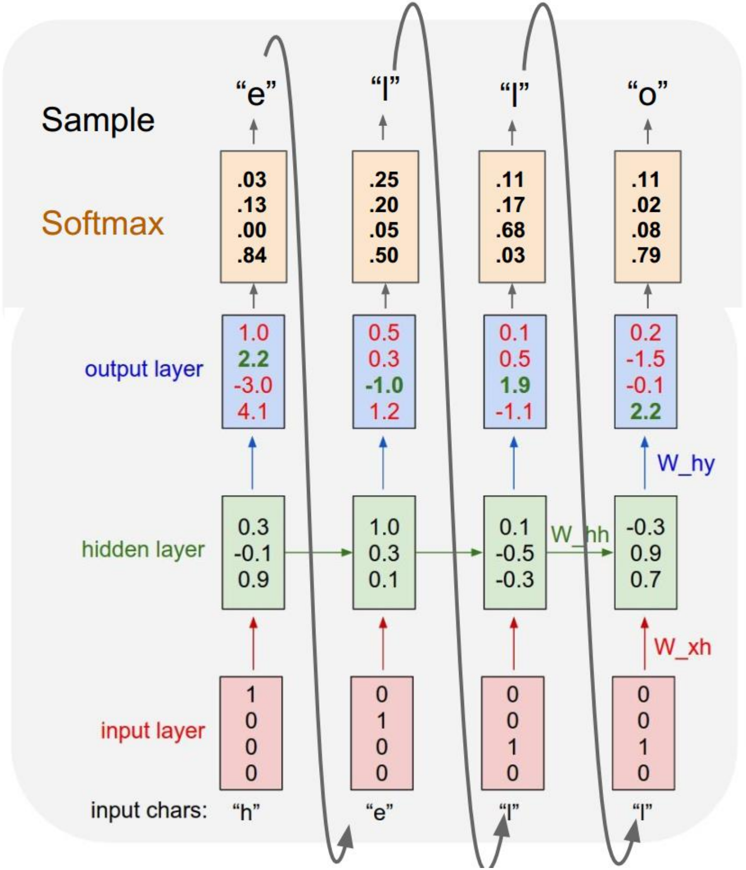
Simple RNN Training Example: Predicting the next letter

Example:
Character-level
Language Model
Sampling

Vocabulary:
[h,e,l,o]

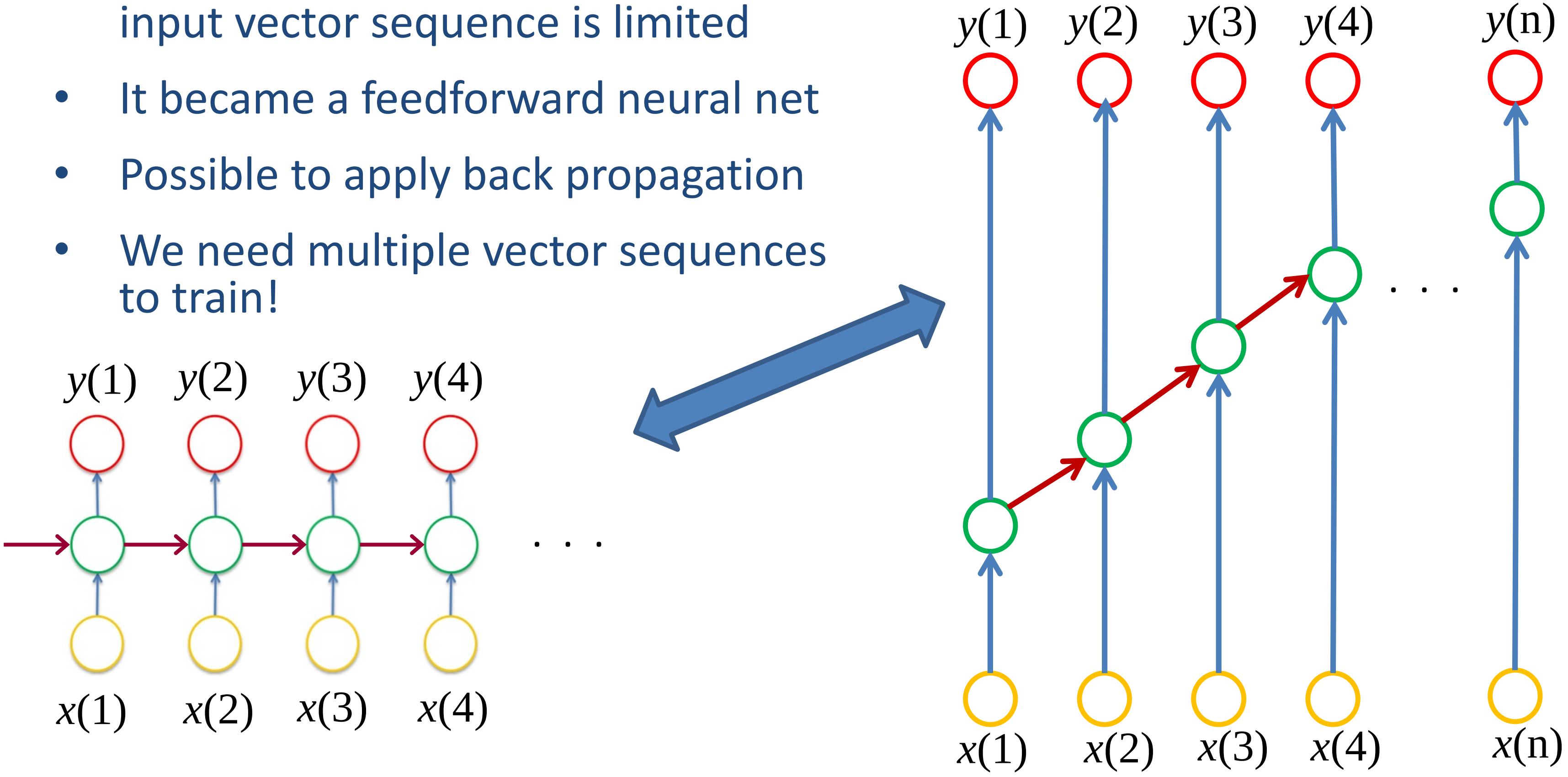
At test-time sample characters one at a time, feed back to model

Backpropagation can be started using negative log likelihood cost function



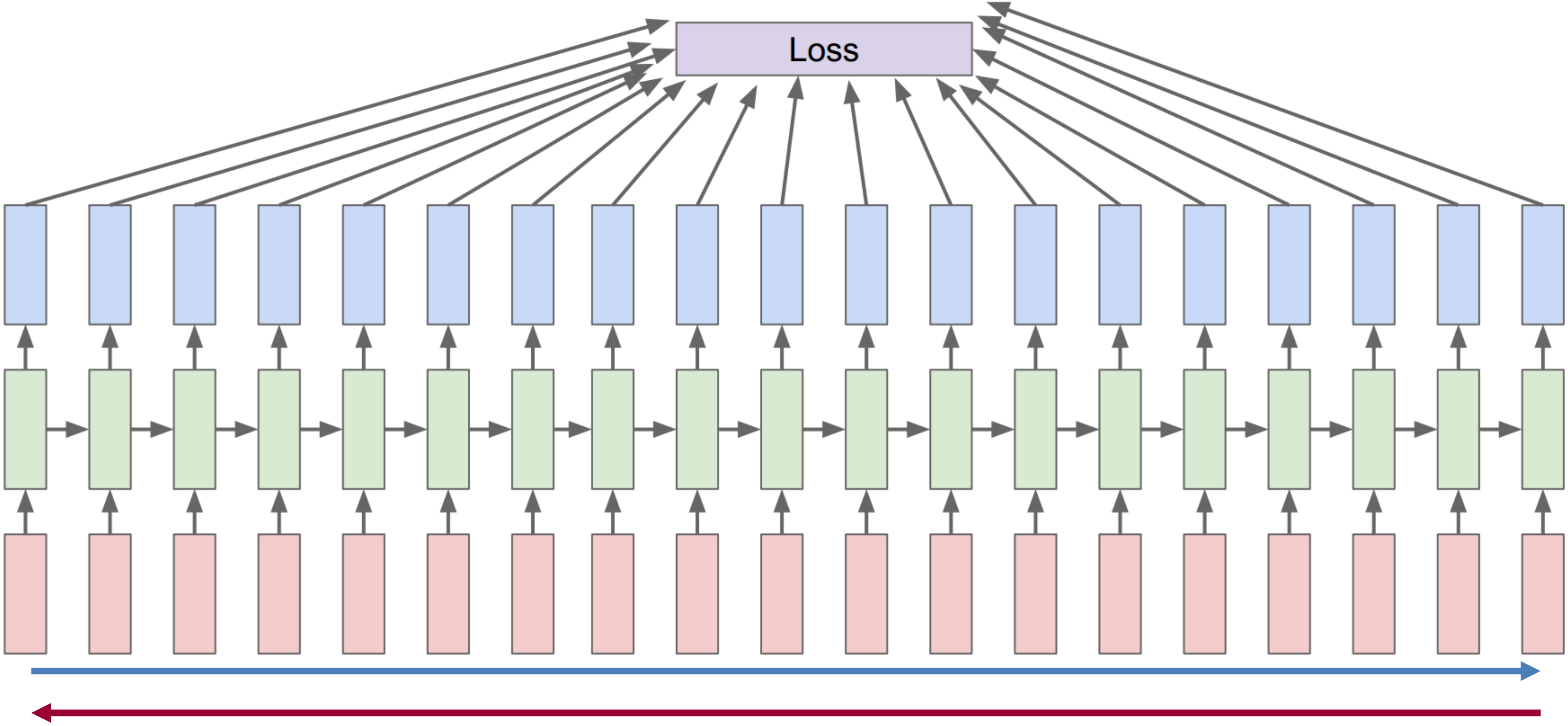
Back propagation through time

- Assuming that the length of the input vector sequence is limited
- It became a feedforward neural net
- Possible to apply back propagation
- We need multiple vector sequences to train!

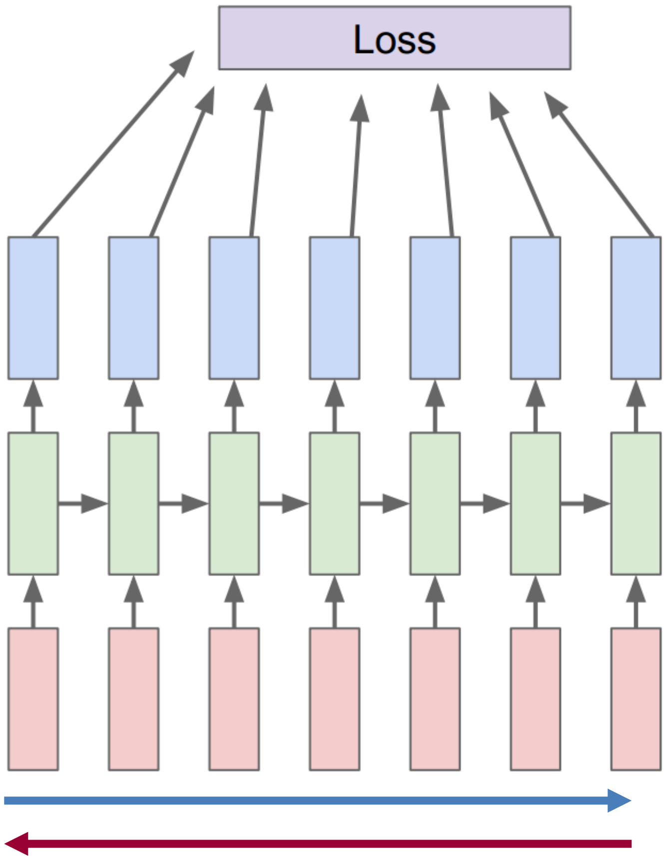


Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

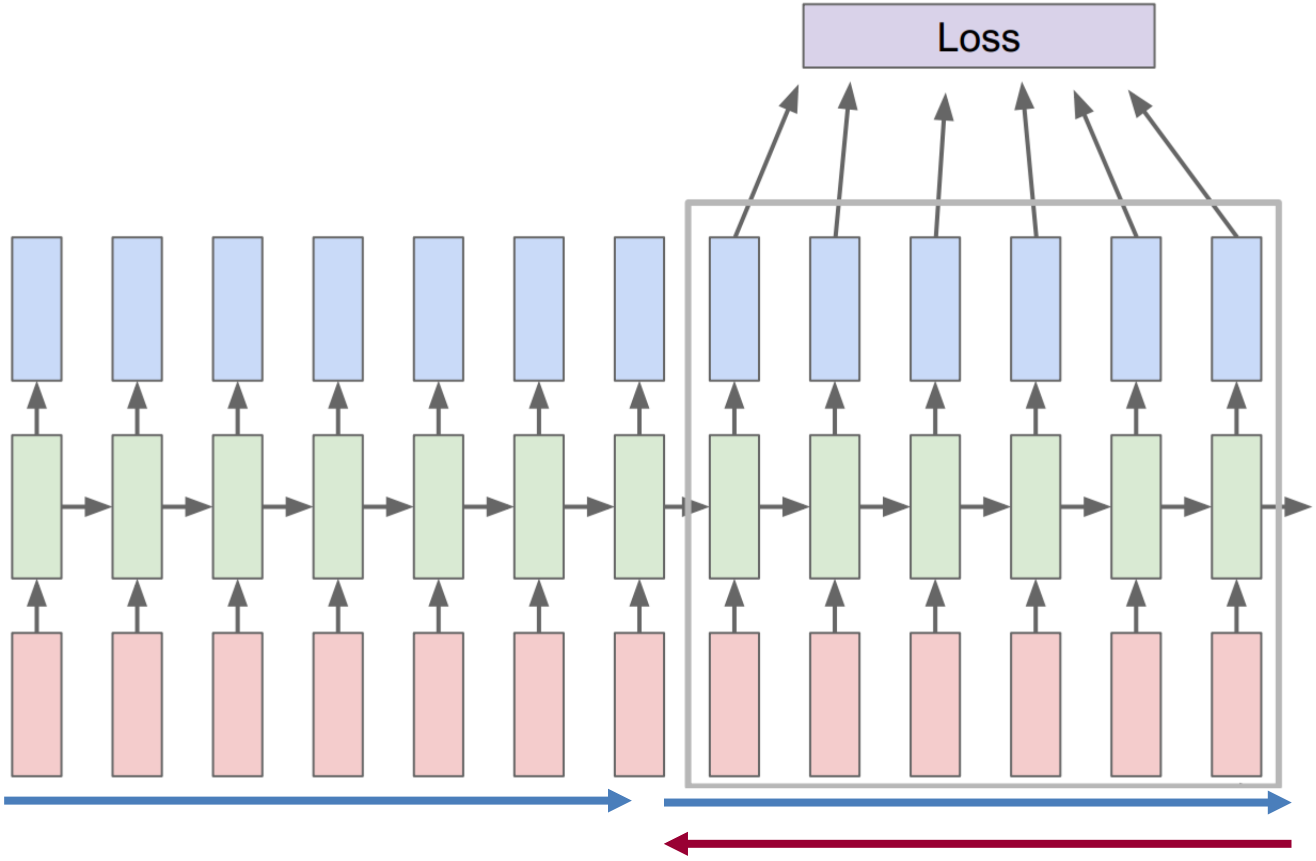


Truncated Backpropagation through time



Run forward and backward through chunks of the sequence instead of whole sequence

Truncated Backpropagation through time



Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps!

Truncated Backpropagation through time

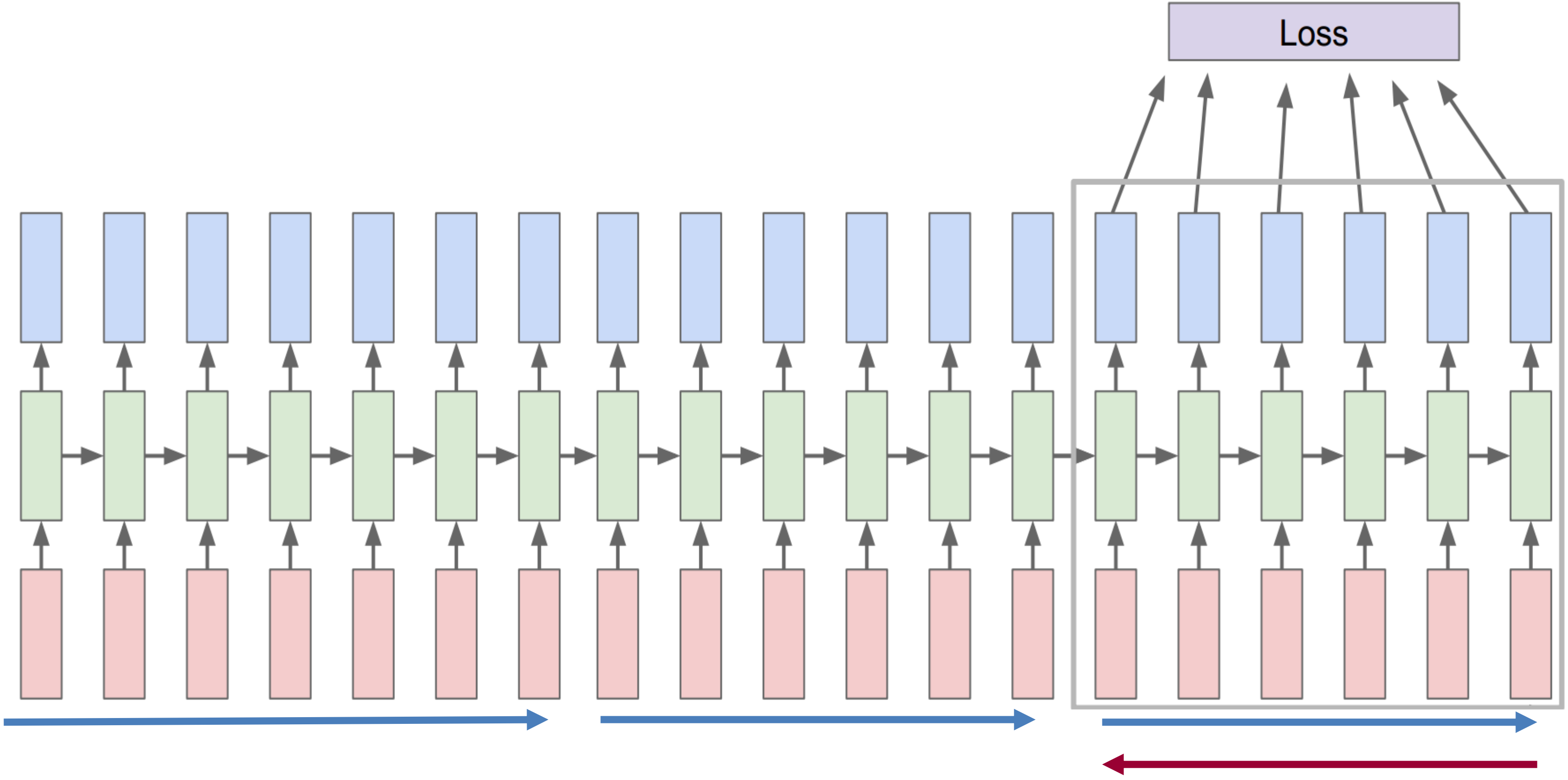


Image captioning example

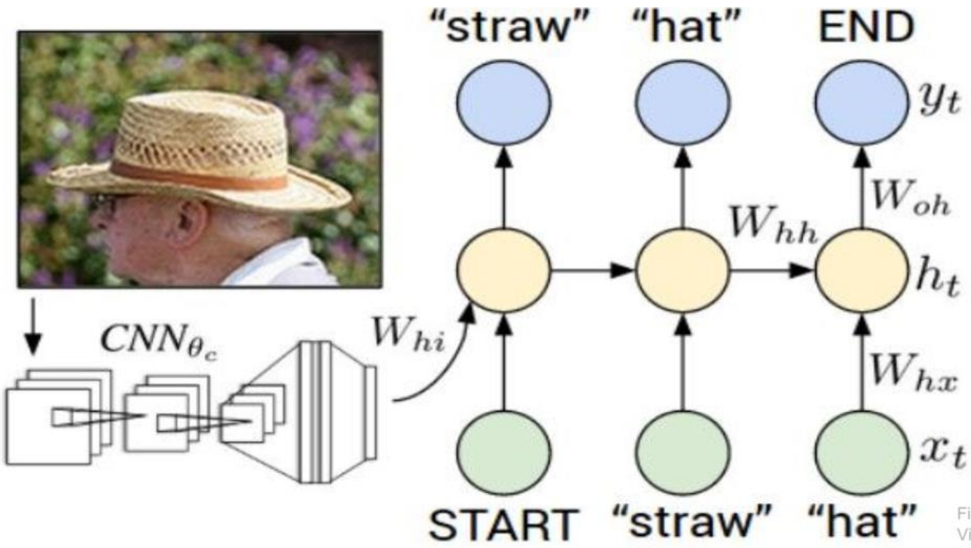
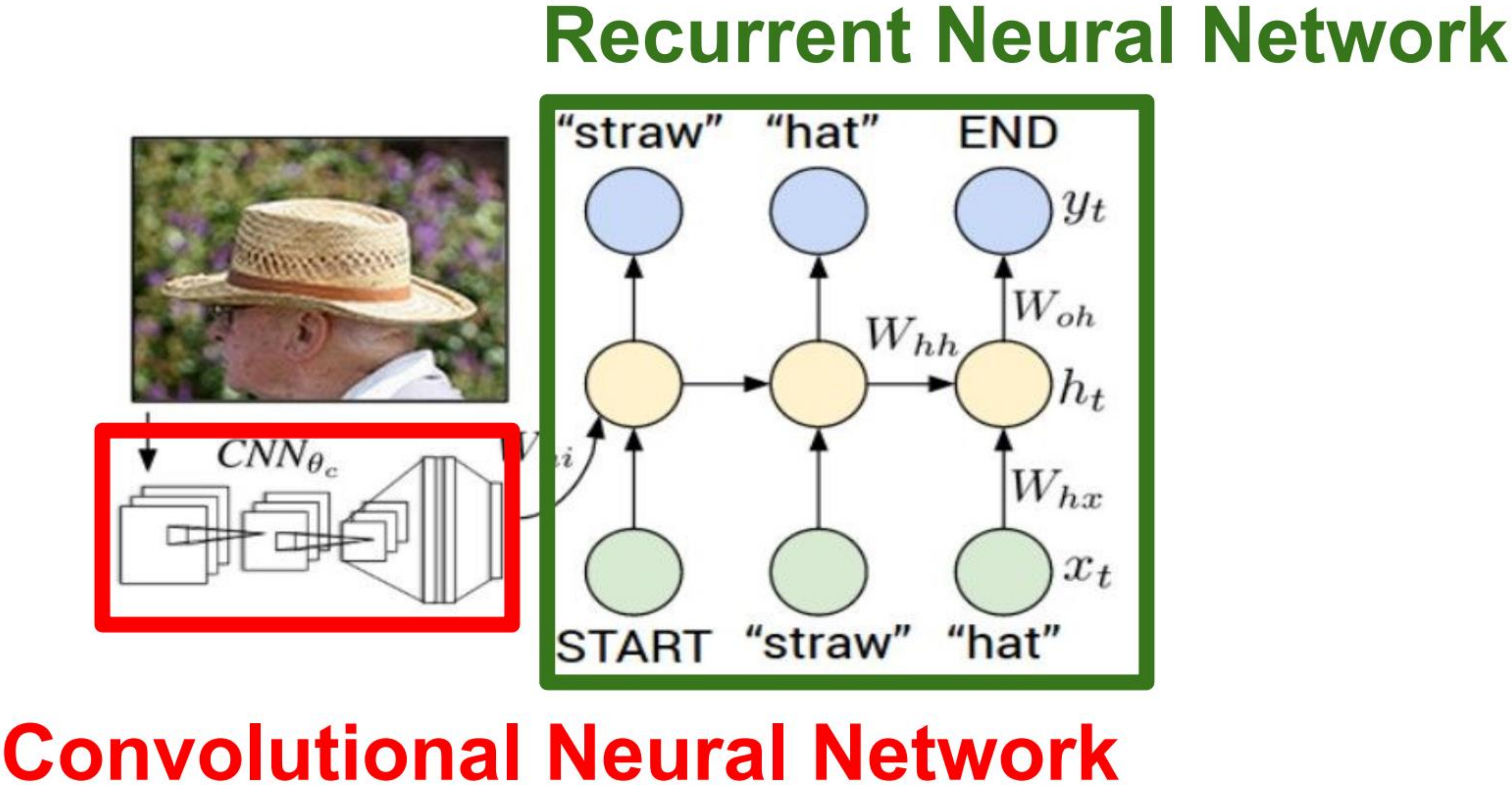


Figure from Karpathy et a, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015. Reproduced for educational purposes.

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Image captioning example



test image

test image

Alexnet: scored 5 best guesses

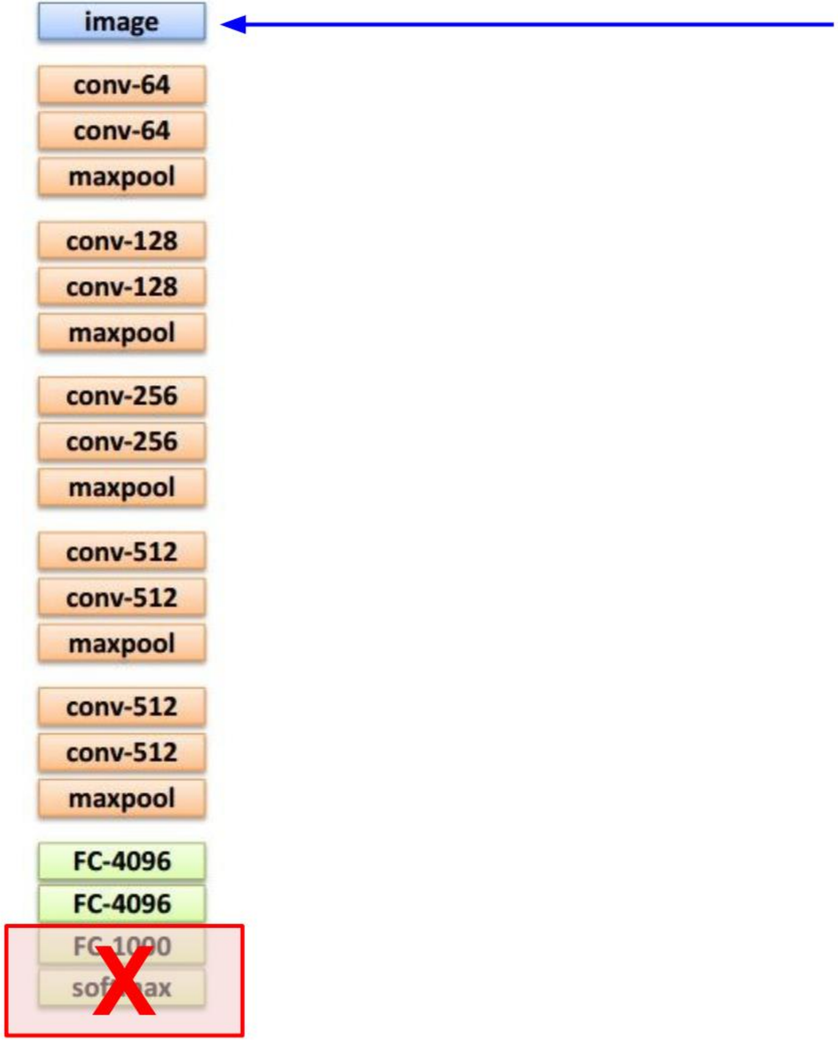


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
FC-1000
softmax
X

test image

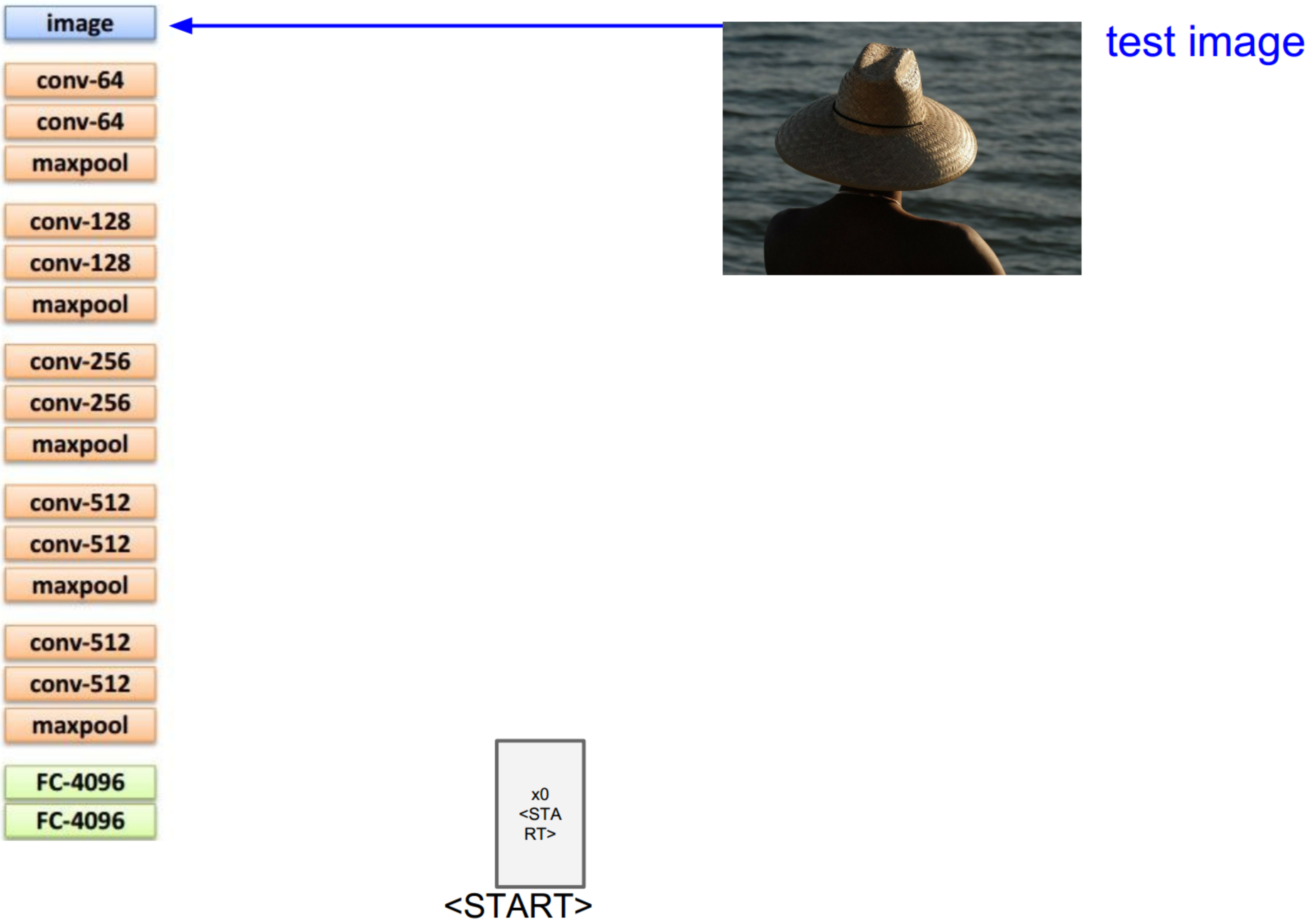
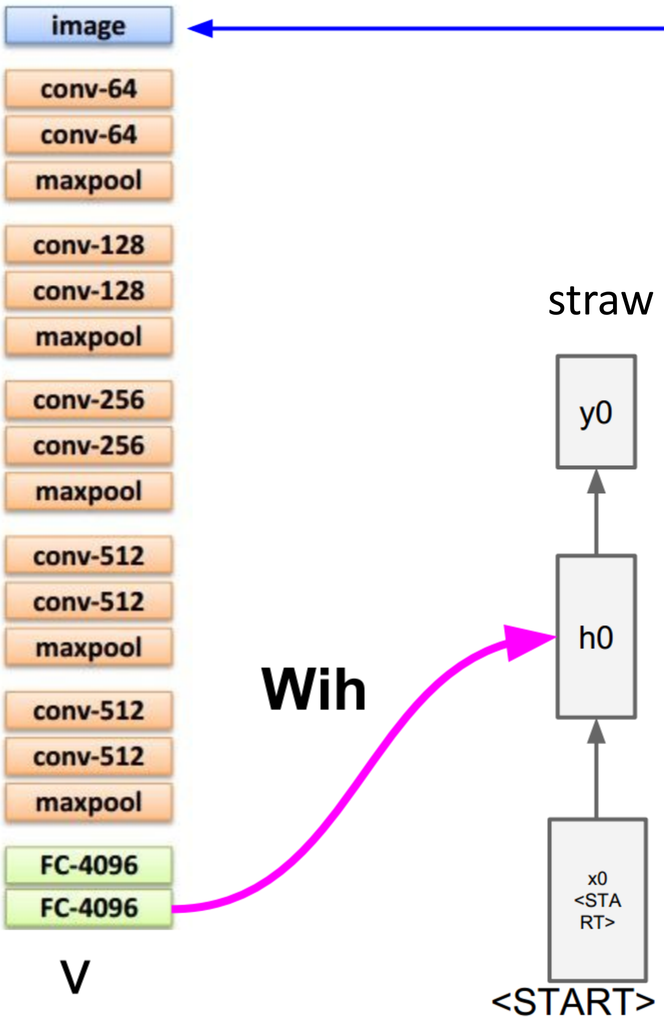


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
x0
<STA
RT>
<START>



test image

$$h = \tanh(W_{xh} * x + W_{hh} * h + W_{ih} * v)$$

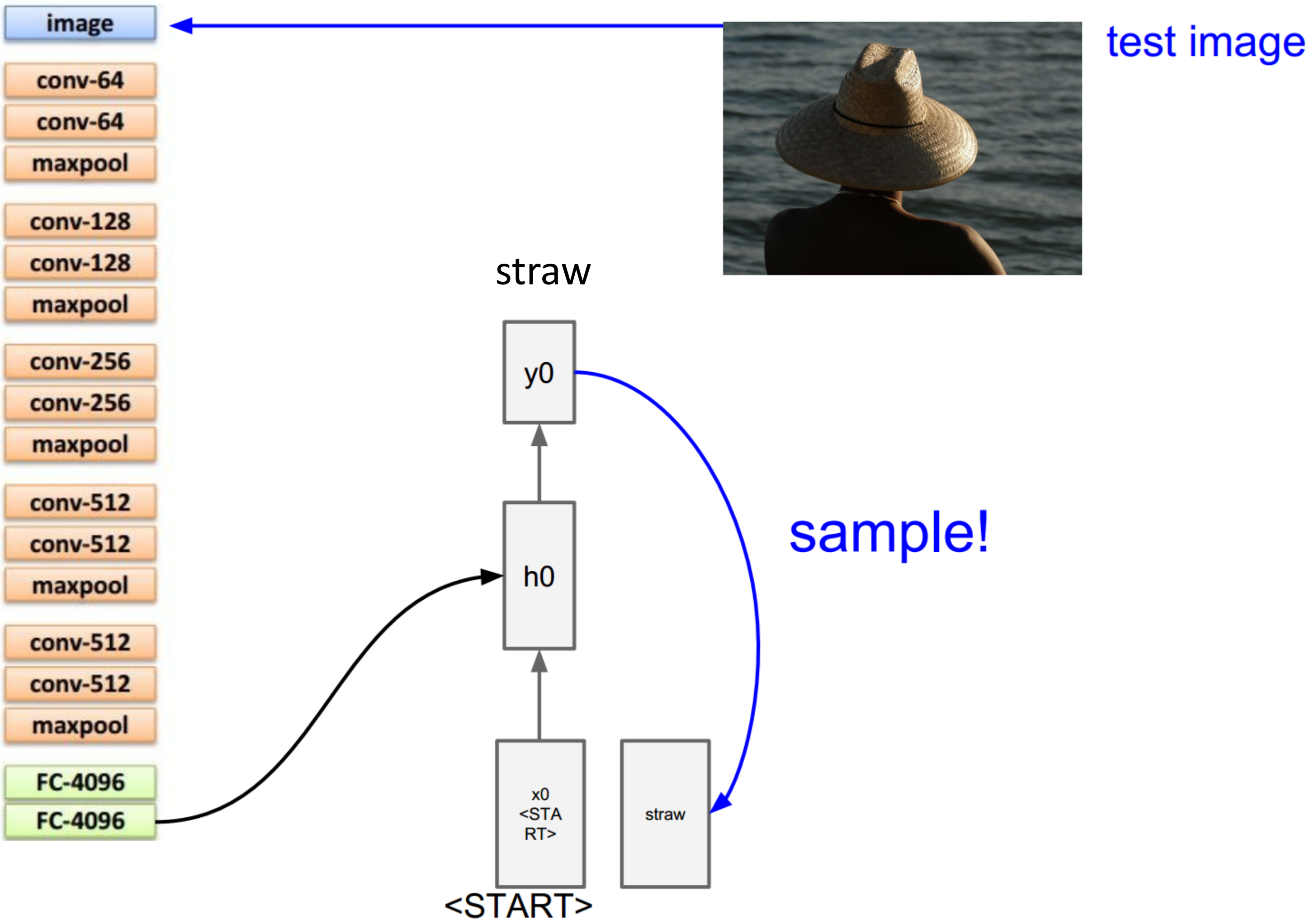


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
y0
h0
sample!
x0
<START>
straw
<START>

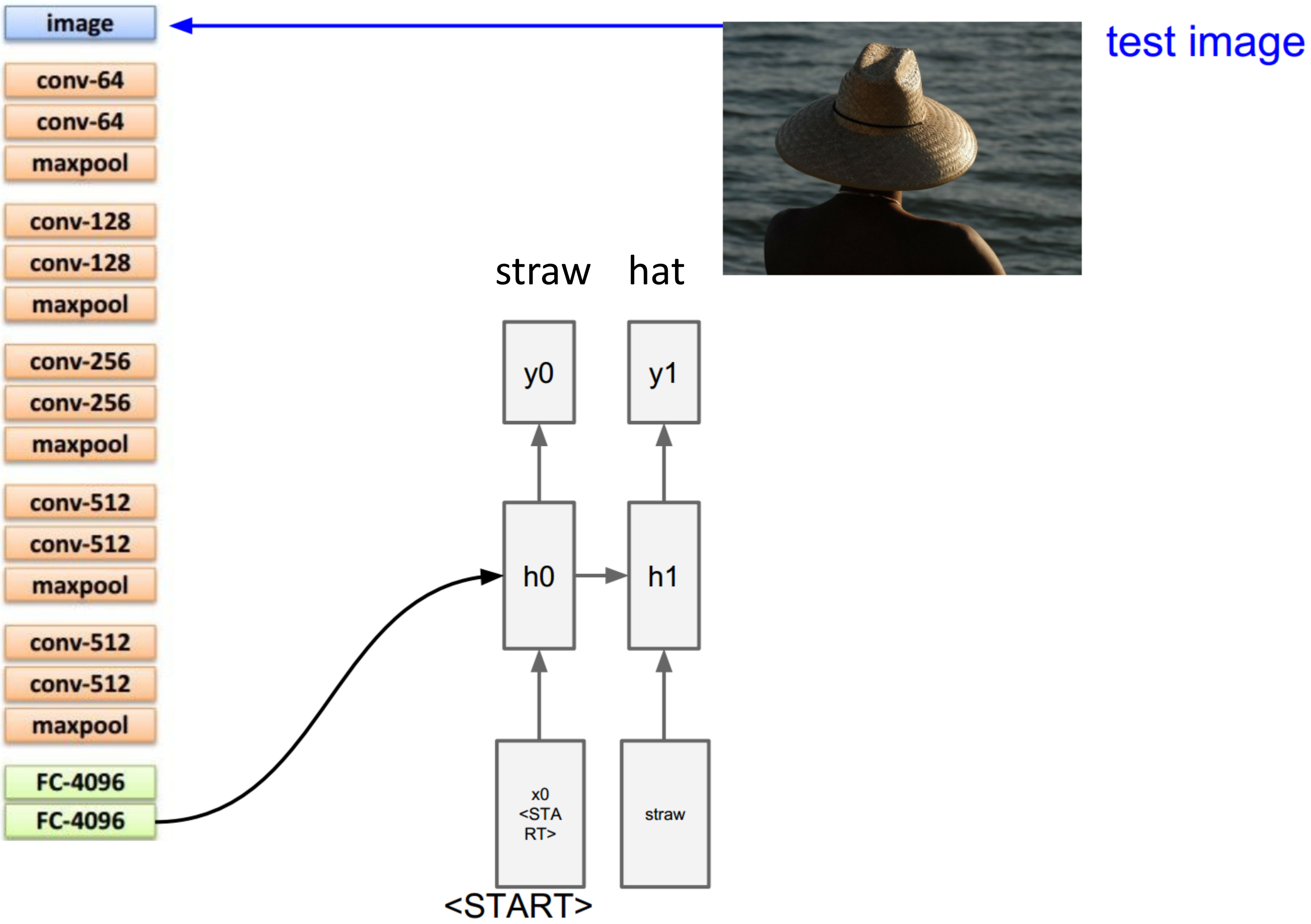


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
y0
y1
h0
h1
x0
<STA
RT>
straw
<START>

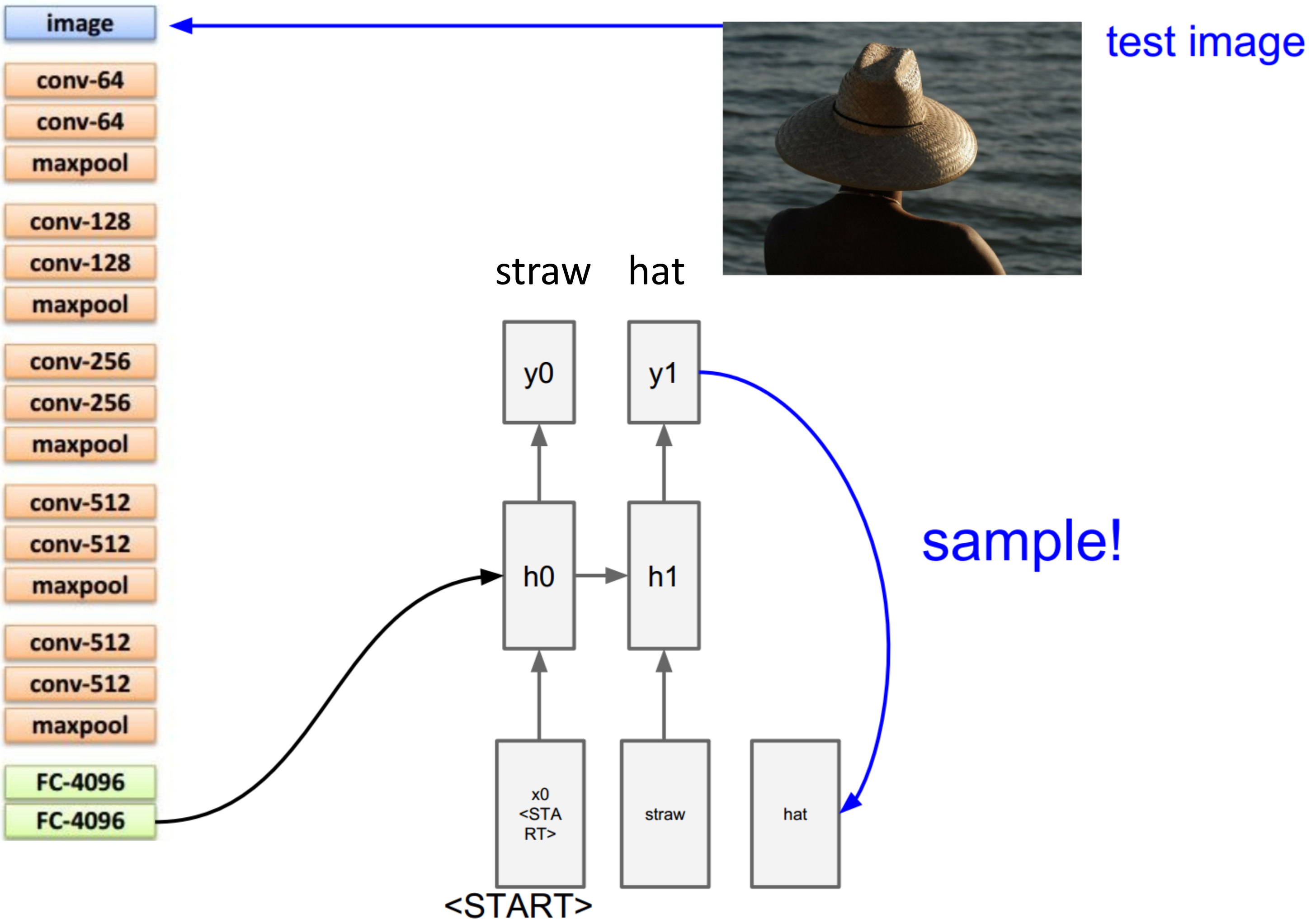
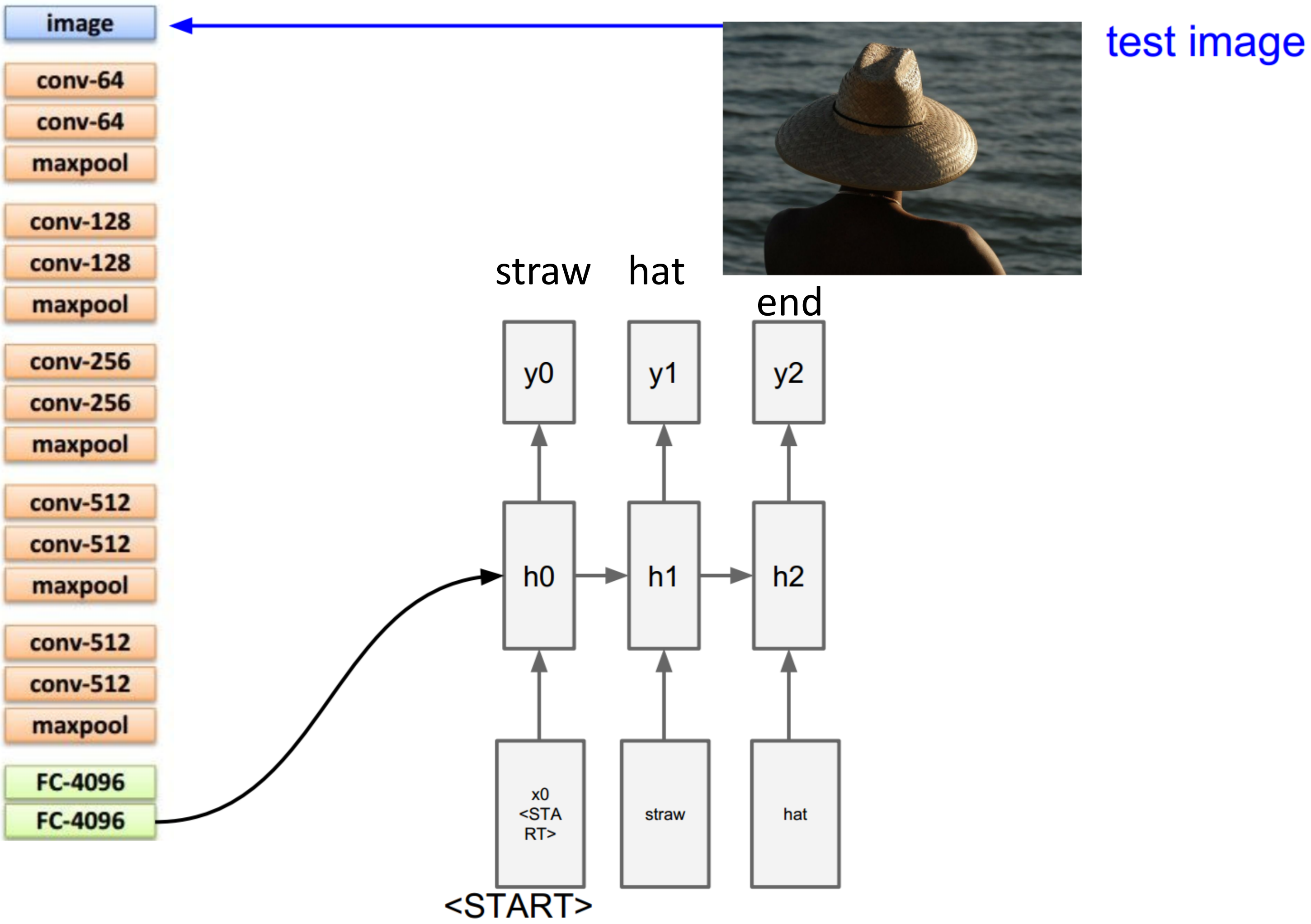


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
y0
y1
h0
h1
sample!
x0
<STA
RT>
straw
hat
<START>



10.00 X 5.63 inch

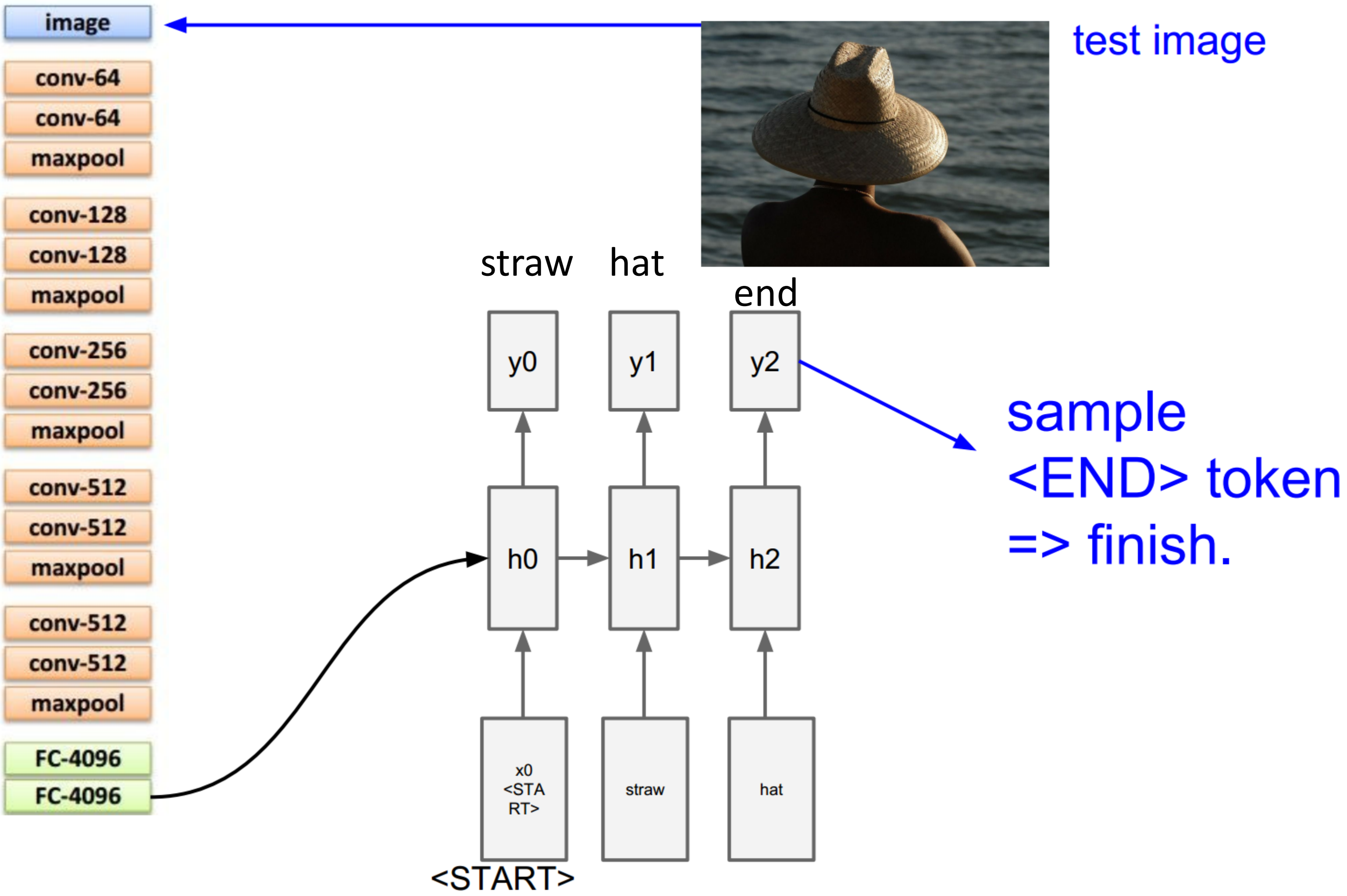


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
end
y0
y1
y2
h0
h1
h2
x0 <STA RT>
straw
hat
<START>
sample <END> token => finish.

Image captioning Example: Results

Captions generated using neuraltalk2
All images are CC0 Public domain:
cat suitcase, cat tree, dog, bear,
surfers, tennis, giraffe, motorcycle

A cat sitting on a suitcase on the floor

A cat is sitting on a tree branch

A dog is running in the grass with a frisbee

A white teddy bear sitting in the grass

Two people walking on the beach with surfboards

A tennis player in action on the court

Two giraffes standing in a grassy field

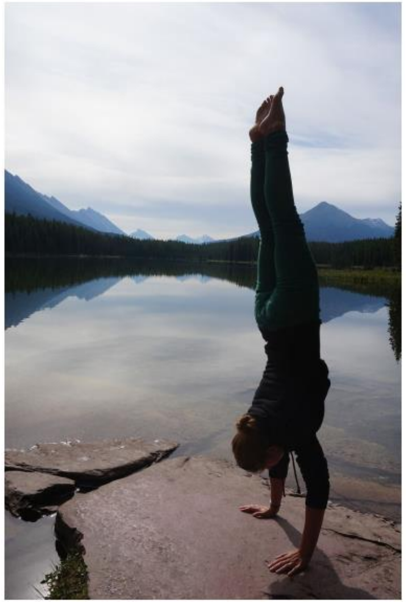
A man riding a dirt bike on a dirt track

Image captioning: Failure cases

Captions generated using neuraltalk2
All images are CC0 Public domain: fur coat, handstand, spider web, baseball

A woman is holding a cat in her hand

A person holding a computer mouse on a desk



A woman standing on a beach holding a surfboard

A bird is perched on a tree branch

A man in a baseball uniform throwing a ball

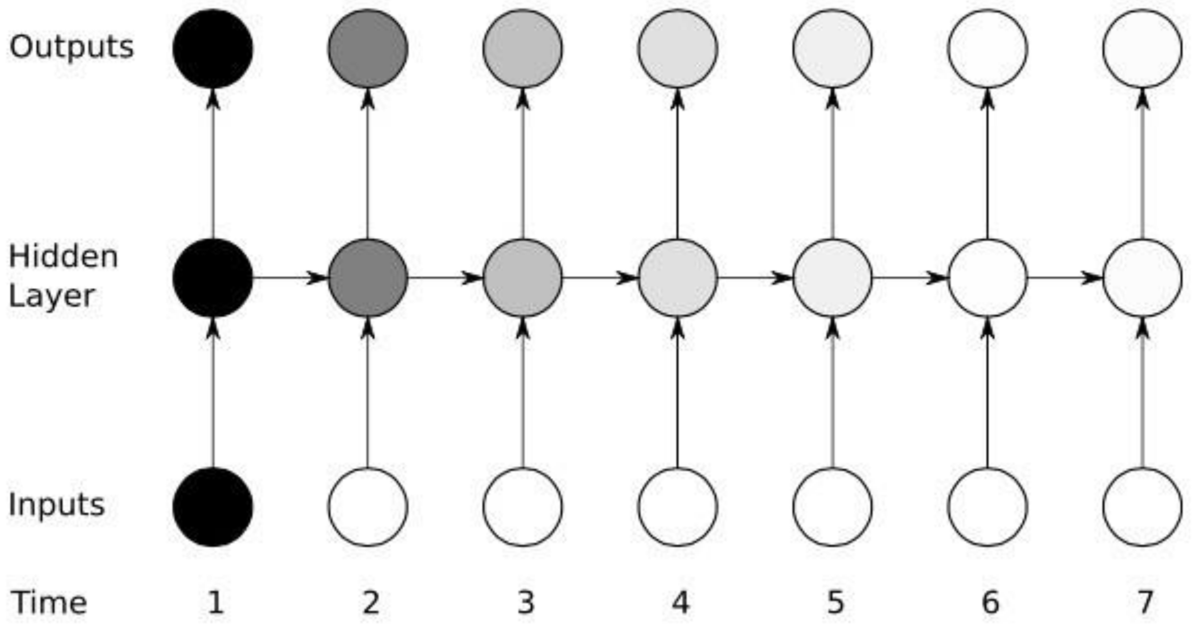
Problem

- What happens if the input sequence is too long?

Vanishing gradient!

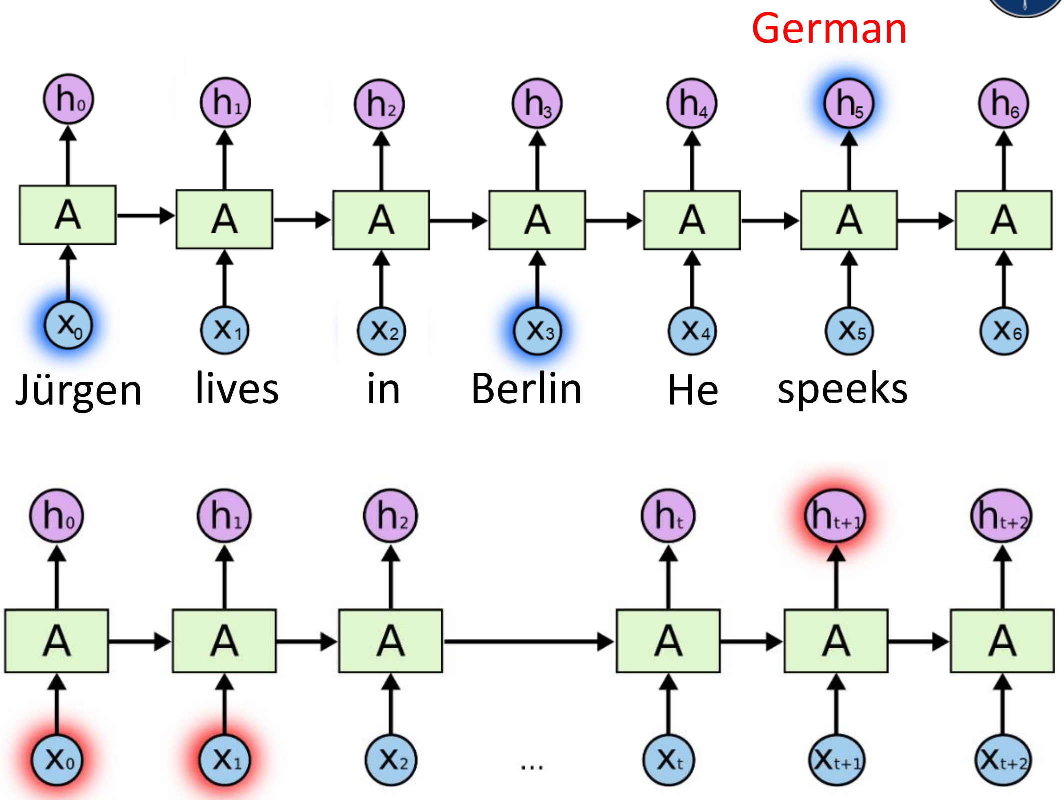
Vanishing Gradient Problem

- In case of long input vector sequencies, the old vectors has a strongly fading effect in inference phase
- In training phase, the stacked gradient functions will be very small



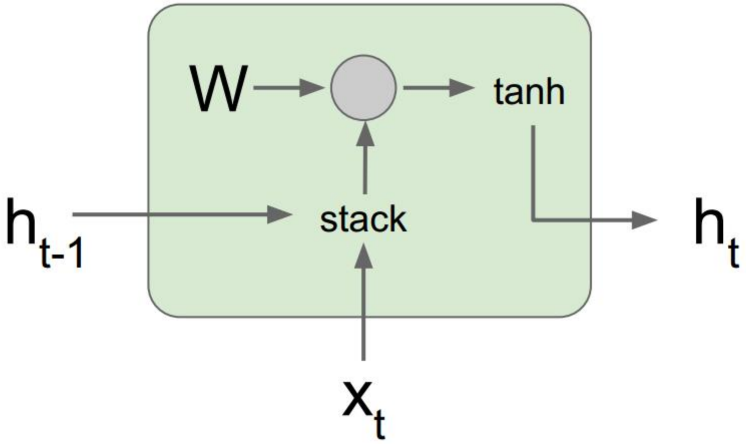
Practical problem of long term dependences

- Consider a network which predicts the next word in a text
  - If the information needed to predict is close, it can be successfully trained
  - If required information is far, the training will be difficult



RNN Gradient flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

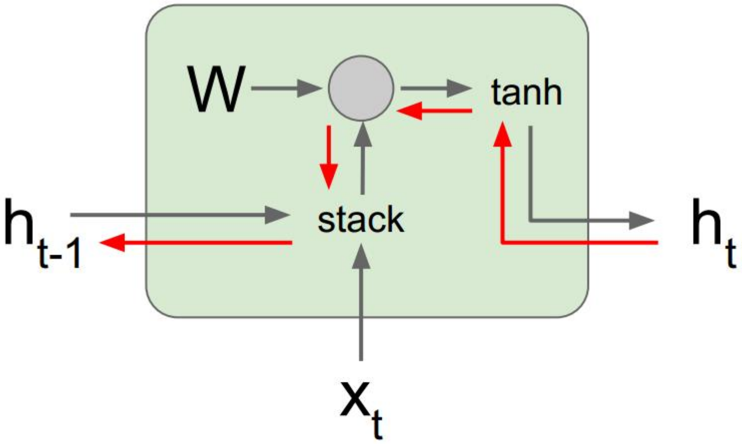


$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

RNN Gradient flow

Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013

Backpropagation from h_t to h_{t-1} multiplies by W (actually W_{hh}^T)

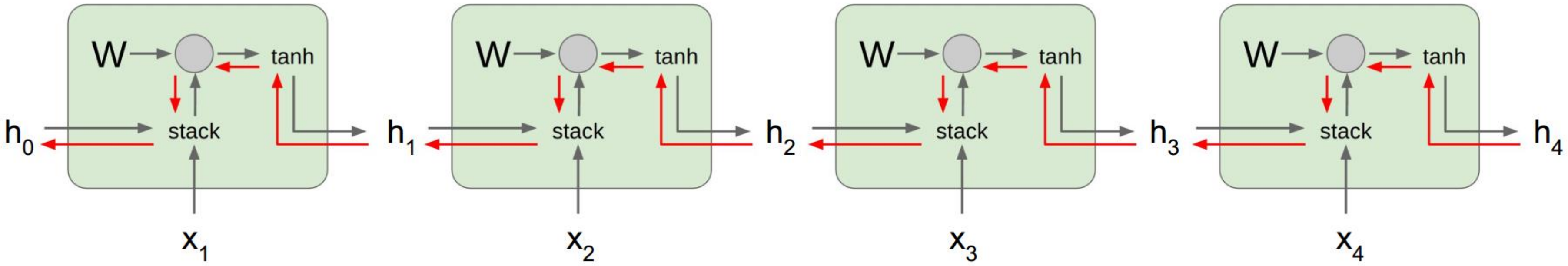


$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$
$$= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

$$h_{t+1} = tanh\left(W\begin{pmatrix} h_t \\ x_{t+1} \end{pmatrix}\right) = tanh\left(W\begin{pmatrix} tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\ x_{t+1} \end{pmatrix}\right)$$

RNN Gradient flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

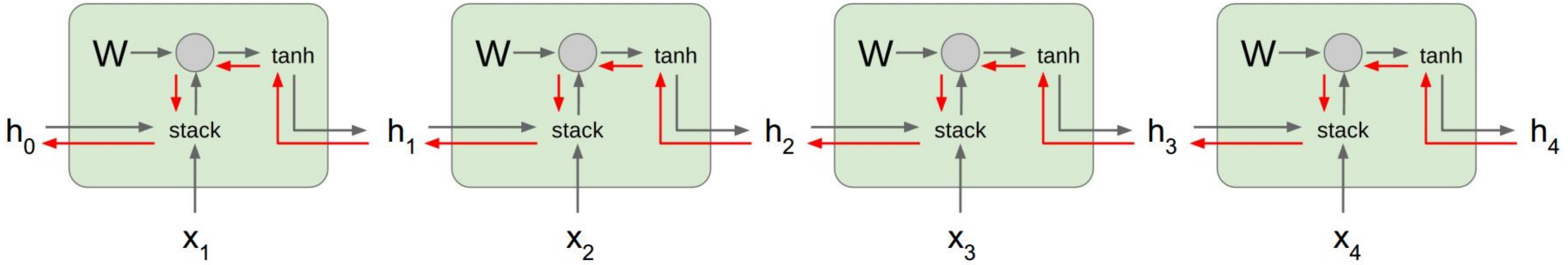
Largest singular value > 1:
Exploding gradients

→ **Gradient clipping**: Scale gradient if its norm is too big

Largest singular value < 1:
Vanishing gradients

RNN Gradient flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

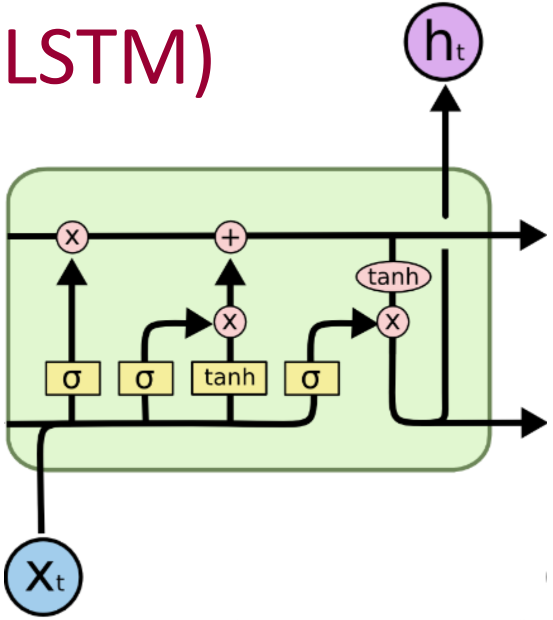
Largest singular value < 1:
Vanishing gradients

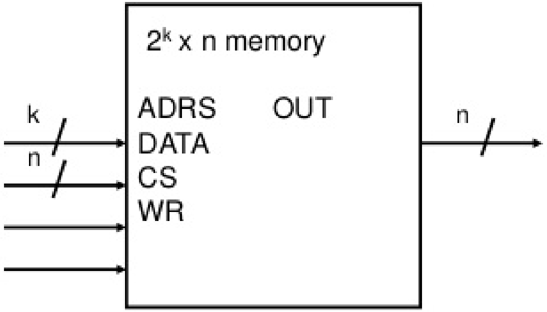
→ Change RNN architecture

Introduction of Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM)

- Was originally introduced Hochreiter & Schmidhuber (1997)
- Idea:
  - To be able to learn long term dependences
  - Collects data when the input is considered to be relevant
  - Keeps it as long as it considers to be important
  - Technique:
    - Handle the state as a memory with minor modifications
      - No matrix multiplication
      - No tanh
      - Apply memory handling kind signals
        - » data in, data out, write, enable

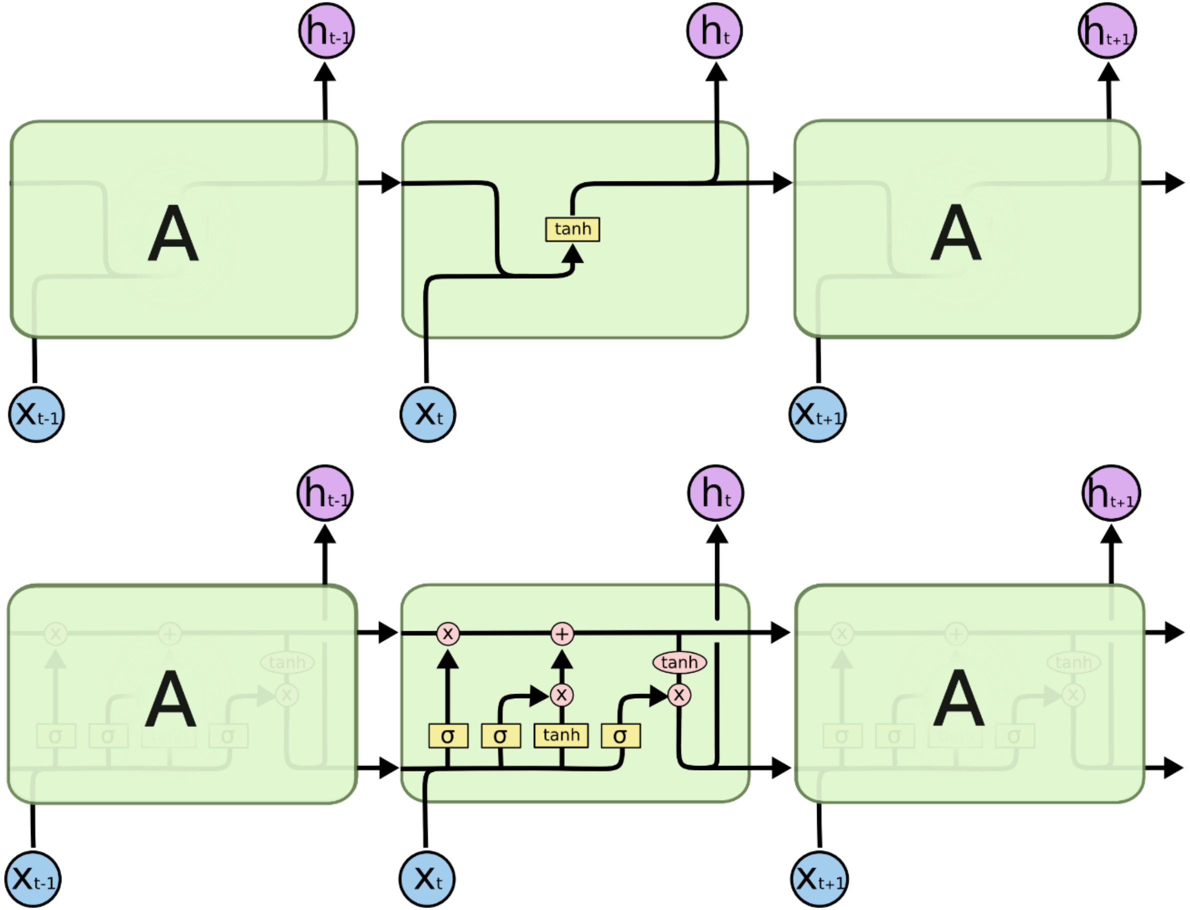




http://colah.github.io/posts/2015-08-Understanding-LSTMs/

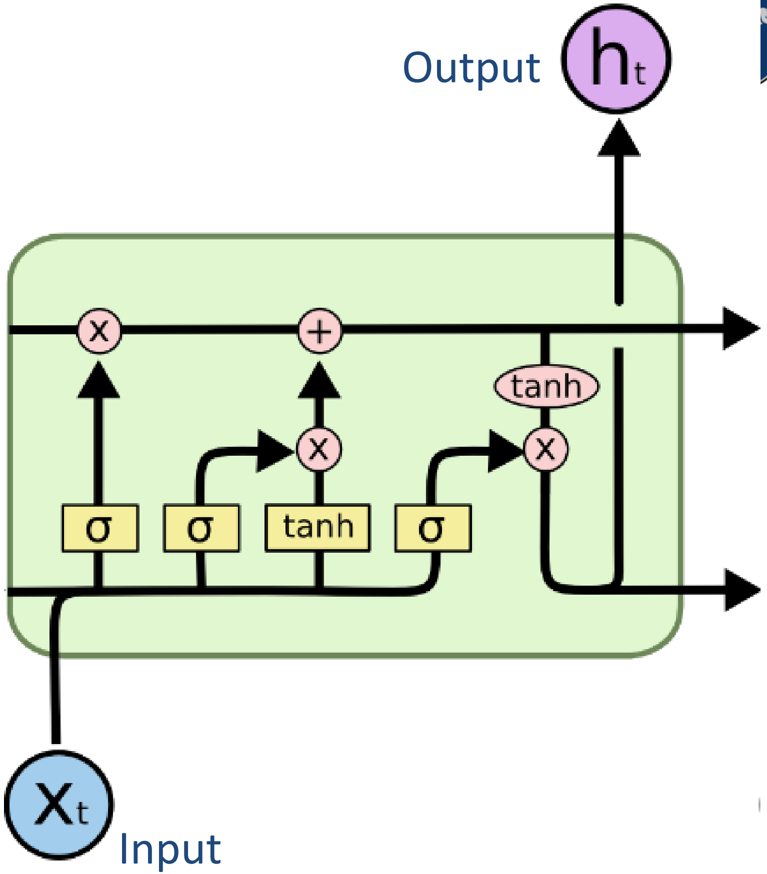
Derivation of LSTM

- Repeating module in Normal RNN
  - concatenates the input and the state
  - A neural network with tanh output and repeats the result
- LSTM
  - Uses the state as a memory
  - Uses 4 neural nets to control the memory
    - Forget_gate, Input_gate, State_update, Output_gate



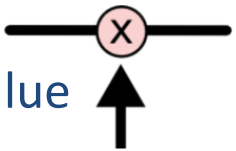
Components of LSTM I

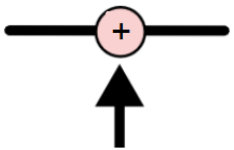
- All wires represents vector
  - Vector transfer
  - Vector concatenation
  - Vector copy
- Neural nets with (yellow boxes)
  - **Multi-layer NN** with *tanh* activation function used for update value calculation tanh
  - **Multi-layer NN** with *logistic* activation function (sigmoid) used for value selection (kind of addressing) σ
- Pointwise operation (pink circles)
  - Pointwise multifaction
  - Pointwise addition

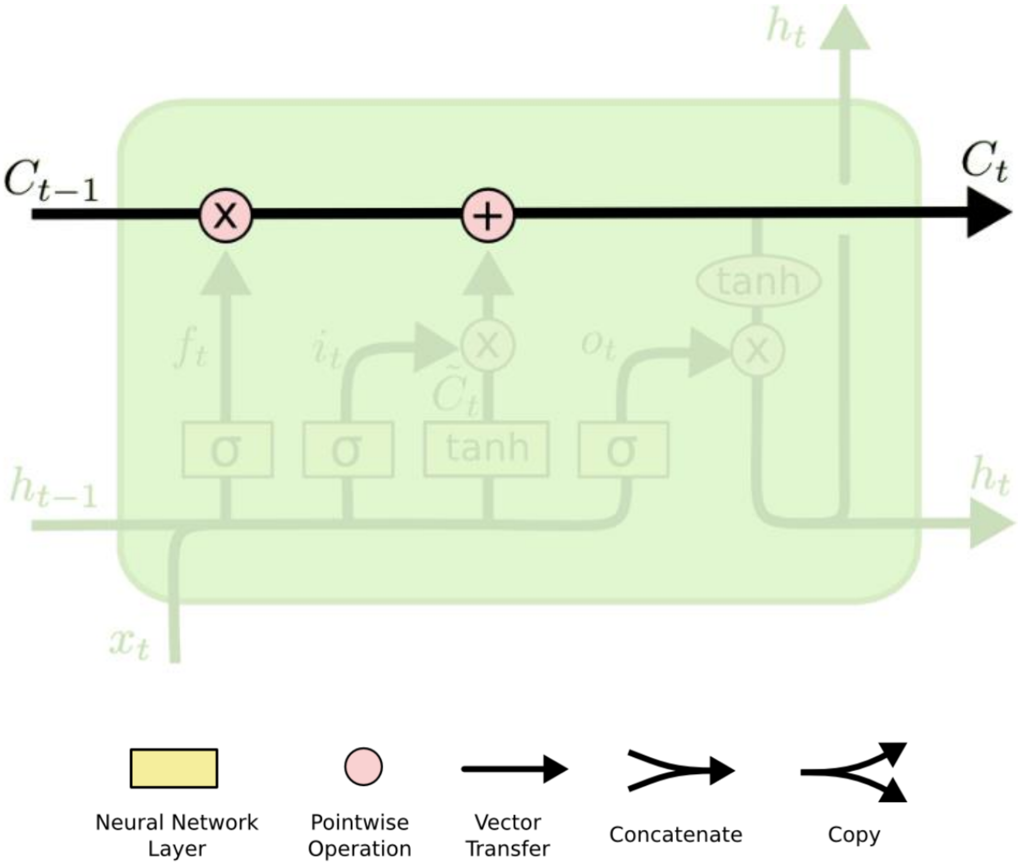


Components of LSTM II

- State of the LSTM
  - This is the actual memory,
  - It can pass the previous values with or without update
  - Represented by the upper black line
  - Indicated with C_t
- Old content can be removed value-by-value
- New content can be added

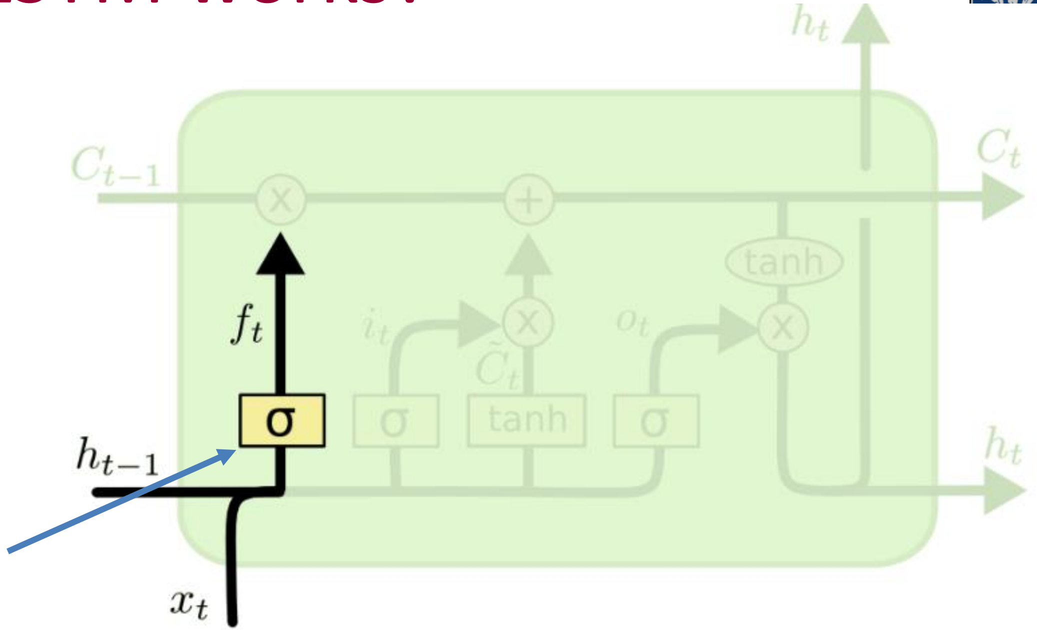






How LSTM works?

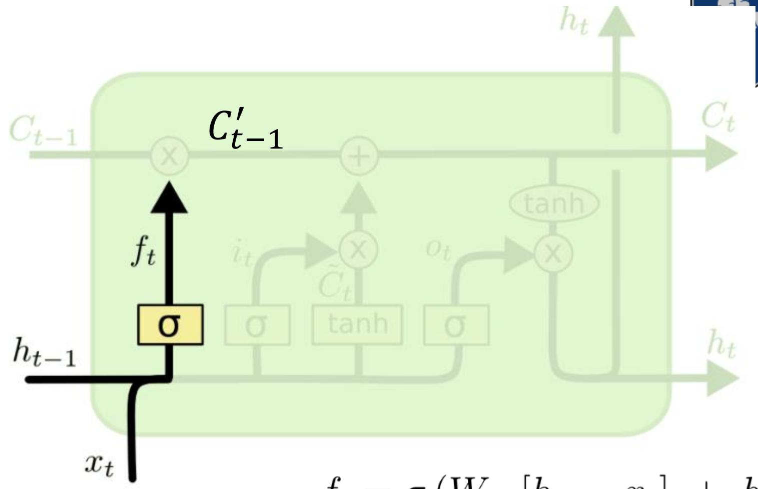
- Step 1
  - Combines input and previous output (concatenation)
  - Selects which values to forget
    - Sort of addressing
    - Done by the **"Forget Gate"**
    - Neural net with sigmoid output



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

Updating state memory (Example)

- Input: “James”
- **Forget Neural** network figures out:
  - Analyzes the concatenated vector
  - Name, Subject of a sentence, Male
- Selects which values to forget and how much
  - Position and weight
- Task:
  - Update gender of the subject (forget the old value)
  - Gender might be represented with a variable
    - c_1: value proportional with the probability that the subject is a male
    - c_2: represents weather
  - Calculate the forget factor of the gender memories
    - 0 completely get rid of it
    - 1 keep the previous value
    - 0 .. 1 partial forget
    - Adressing and suppressing!!!



James

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

c_1: subject’s gender

f1: forget factor of c_1

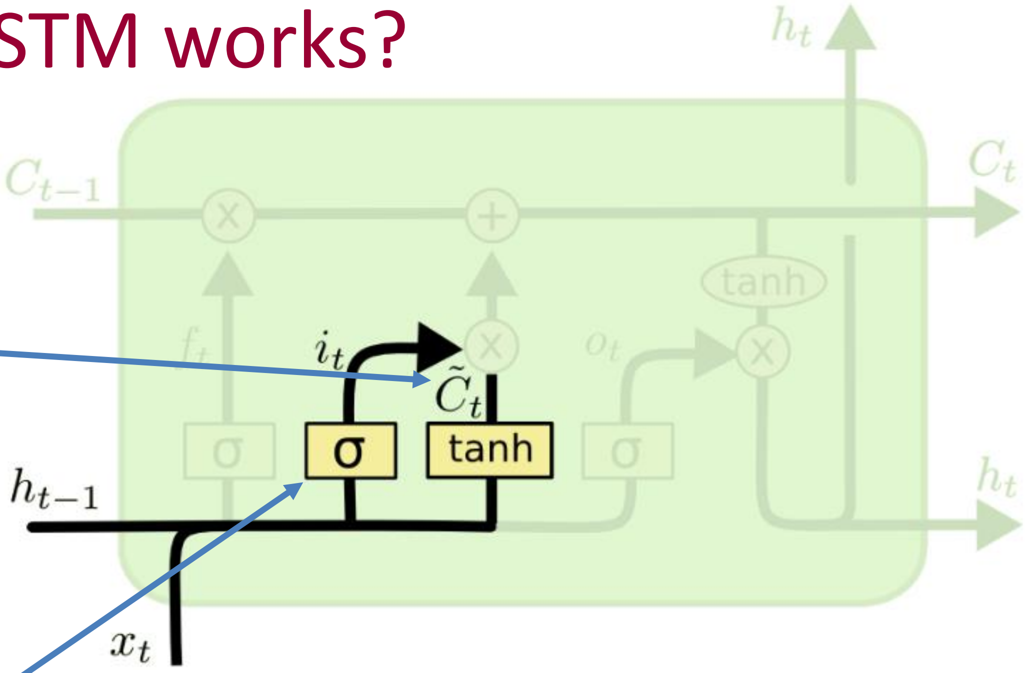
c1 value after partial forget

$$C_{t-1} = \begin{bmatrix} -0.5 \\ 0.2 \\ \vdots \end{bmatrix} \qquad f_t = \begin{bmatrix} 0.1 \\ 1 \\ \vdots \end{bmatrix} \qquad C'_{t-1} = \begin{bmatrix} -0.05 \\ 0.2 \\ \vdots \end{bmatrix}$$

Not to forget c_2

How LTSM works?

- Step 2
  - Calculation of the state update
    - Done by the **"Cell Network"**
    - Not yet the new value, only the update value
    - Neural Net with *tanh*
  - Selection of the state values to be updates (Addressing)
    - Done by the **"Input Gate"**
    - Neural Net with sigmoid
  -

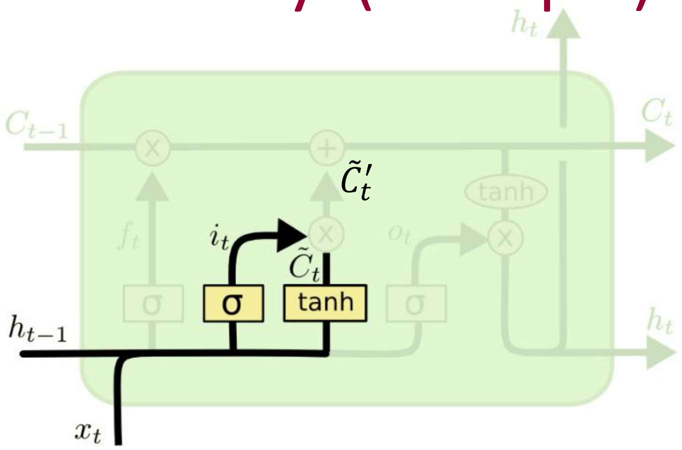


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \;+\; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \;+\; b_C)$$

Updating state memory (Example)

- Input: “James”
- Input Gate figures out:
  - Analyze the concatenated vector
  - Select which values to update (ENABLE!!!)
  - Calculate the update weights
- Cell Network calculates:
  - The update values
- Task:
  - Update gender of the subject (calculate the update value)
  - Gender might be represented with a variable
    - c_1: value proportional with the probability that the gender is male
    - c_2: represents weather
  - Calculate the update factor of the gender memories
    - 0 not to update
    - 1 fully update
    - 0 .. 1 partial update
    - ADRESSING!!!



James

c_1: subject gender estimate value

f1: update factor of c_1

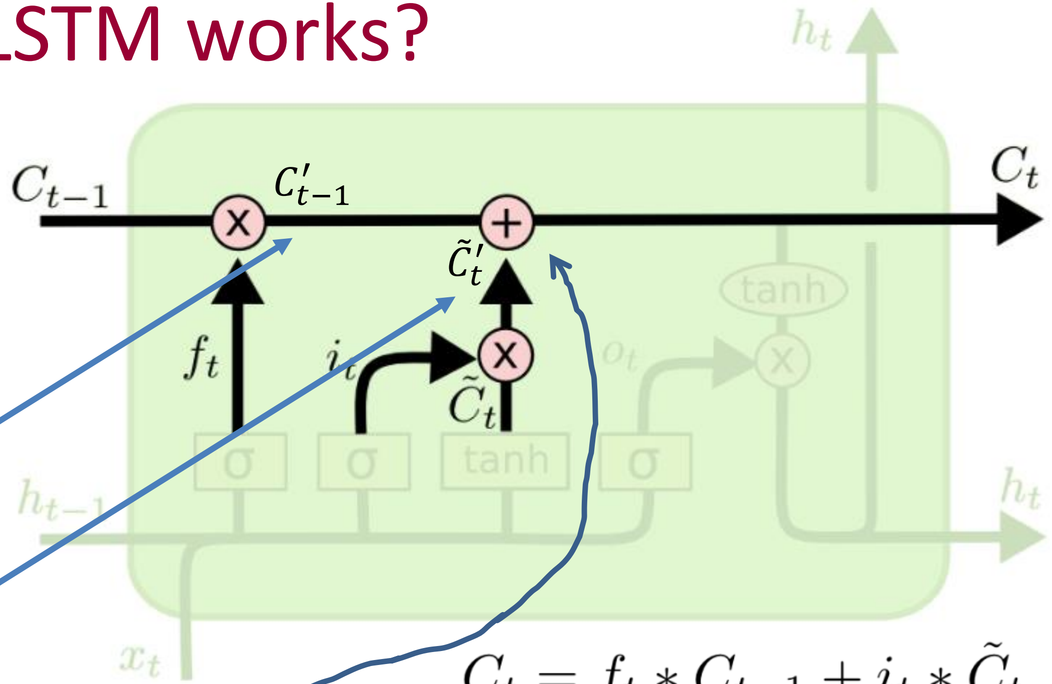
c1 update value

$$\tilde{C}_t = \begin{bmatrix} 0.9 \\ -0.75 \\ \vdots \end{bmatrix} \qquad i_t = \begin{bmatrix} 0.8 \\ 0 \\ \vdots \end{bmatrix} \qquad \tilde{C}'_t = \begin{bmatrix} 0.72 \\ 0 \\ \vdots \end{bmatrix}$$

Not to modify c_2

How LSTM works?

- Step 3
  - Calculation of the state update
    - The old state
      - With the forgotten values in the vector
    - And the state update
      - With update vector
    - **Are added up**



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

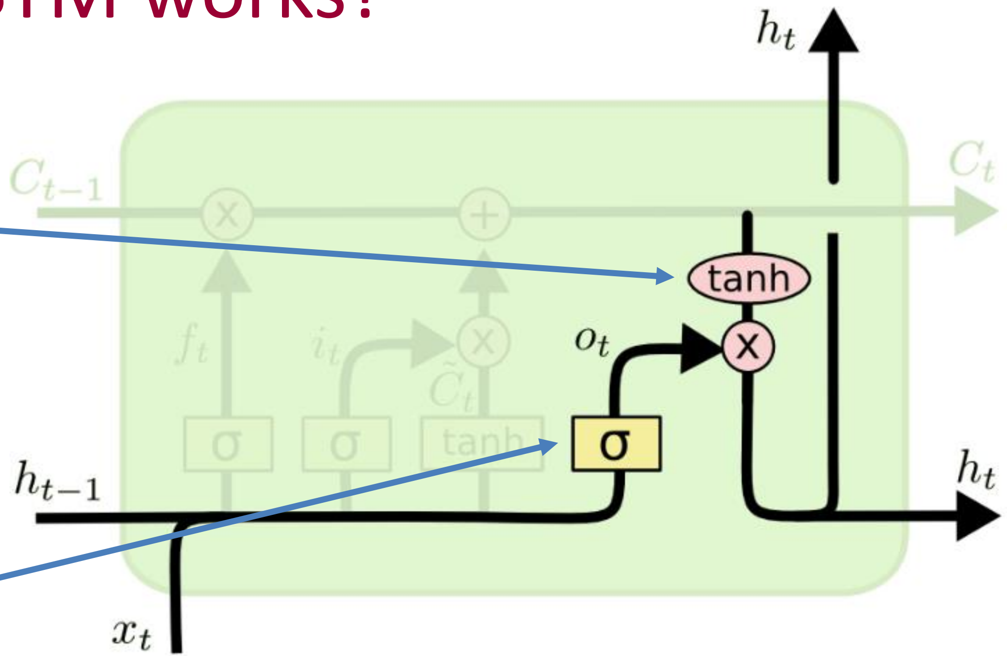
$$C_t = C'_{t-1} + \tilde{C}'_t = \begin{bmatrix} -0.05 \\ 0.2 \\ \vdots \end{bmatrix} + \begin{bmatrix} 0.72 \\ 0 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0.67 \\ 0.2 \\ \vdots \end{bmatrix}$$

c_1: subject gender's estimate value update

c_2: (weather) unchaged

How LTSM works?

- Step 4
  - Apply activation function to the output
    - Squeeze the values between -1 and +1
    - Done by *tanh* activation function
  - Selection of the new output values (Addressing)
    - Done by the **"Output Gate"**
    - Not all the state value is released in each step
    - **Output Gate** decides which values are relevant in this step



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Output vector can be sparse

- Output gate might enables
  - All values of C_t
  - Fraction of C_t (sparse)
  - None of C_t
- C_t can be sparse

$$tanh(C_t) = \begin{bmatrix} 0.2 \\ 0.98 \\ -0.97 \\ -0.1 \\ 0.98 \\ 0.8 \\ 0.2 \\ 0.3 \\ -0.99 \\ 0.8 \\ 0.7 \end{bmatrix}$$

Values are bounded

$$o_t = \begin{bmatrix} 0.01 \\ 0.85 \\ 0.75 \\ 0.1 \\ 0.2 \\ 0.8 \\ 0.1 \\ 0.1 \\ 0.02 \\ 0.9 \\ 0.8 \end{bmatrix}$$

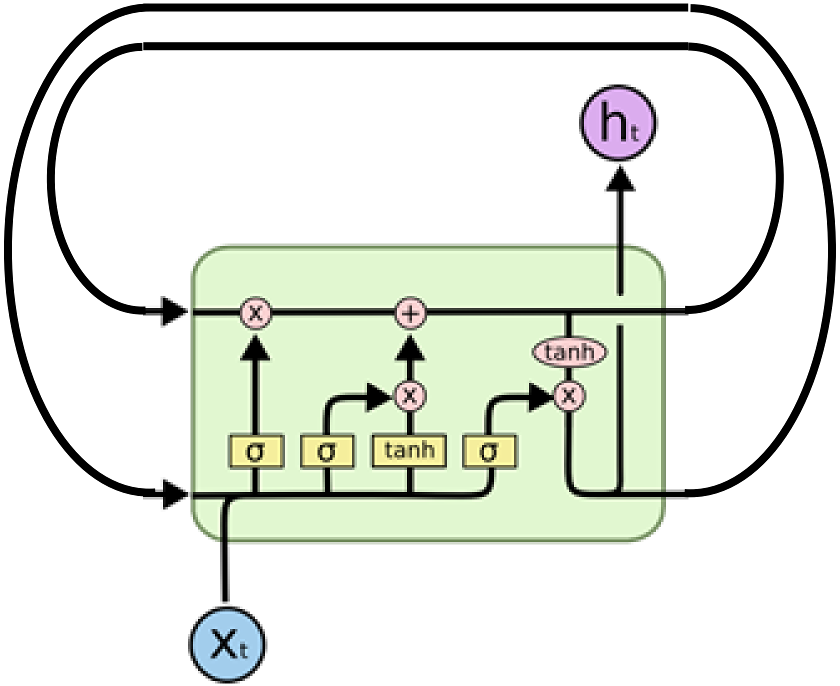
Enabling factor: Enabled values are red

$$h_t = o_t * tanh(C_t) = \begin{bmatrix} 0.002 \\ 0.83 \\ -0.73 \\ -0.01 \\ 0.2 \\ 0.64 \\ 0.02 \\ 0.03 \\ -0.02 \\ 0.72 \\ 0.63 \end{bmatrix}$$

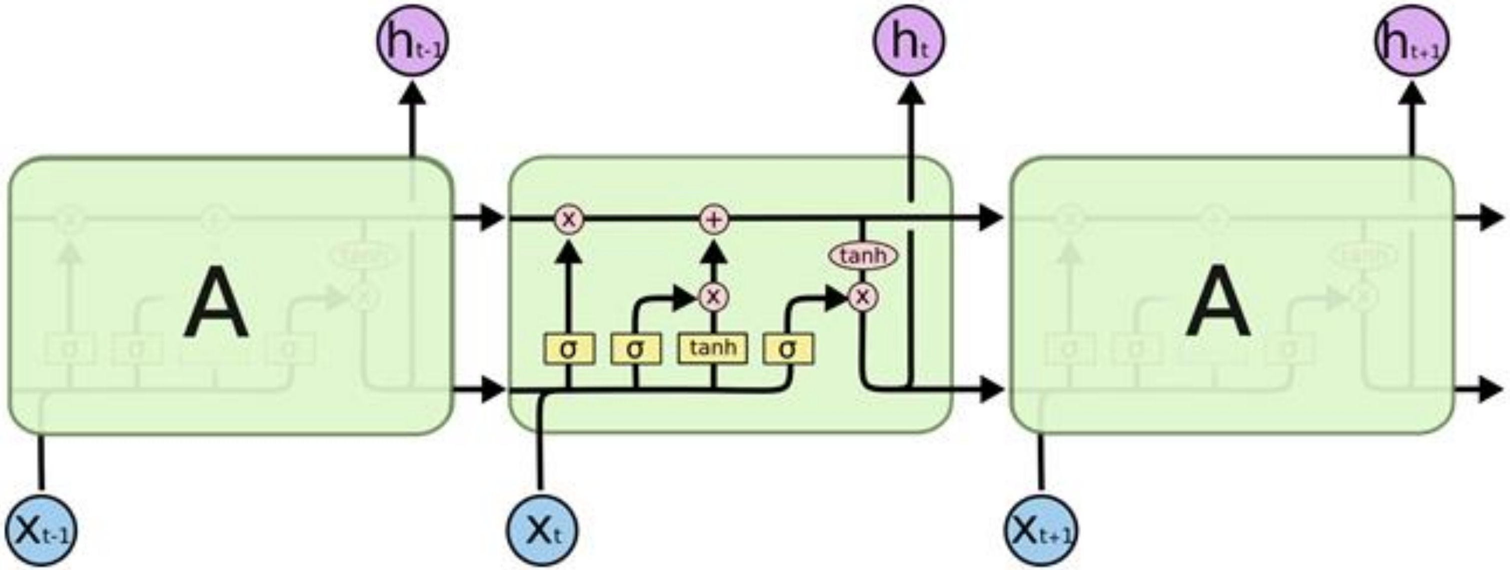
Output vector:
Enabled values are red
Disabled values (gray) will appear on the output, but with reduced values

LSTM network

- General form of an LSTM network

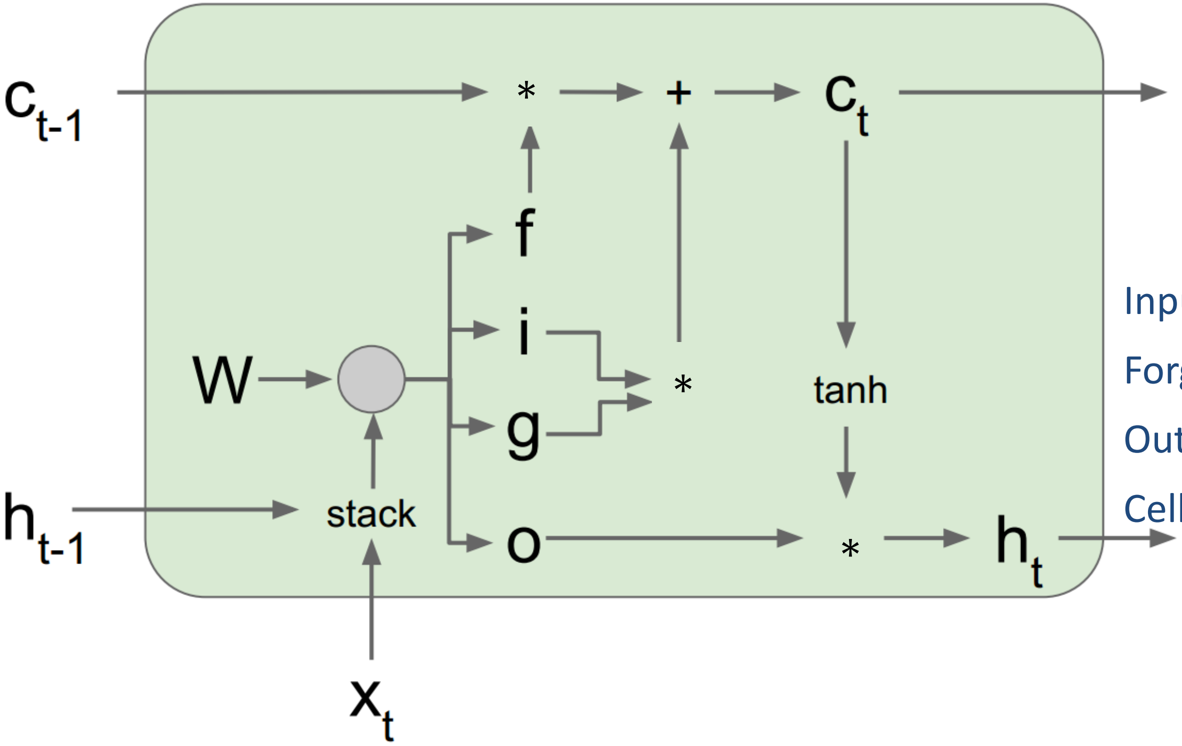


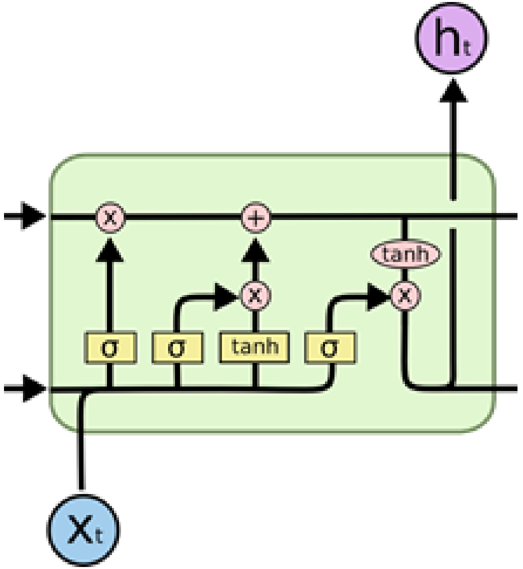
Unrolling LSTM network



Gradient calculation in LSTM

Reformulating equations





$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

Input

Forget

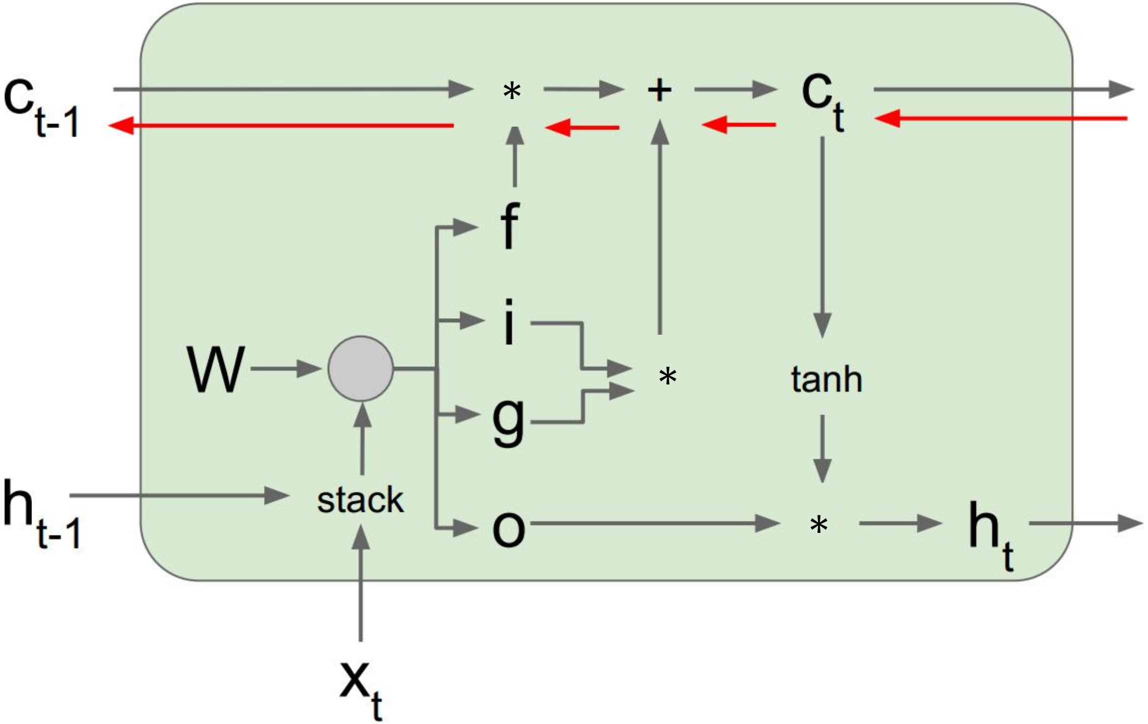
Output

Cell Net

$$c_t = f * c_{t-1} + i * g$$

$$h_t = o * \tanh(c_t)$$

Gradient calculation in LSTM



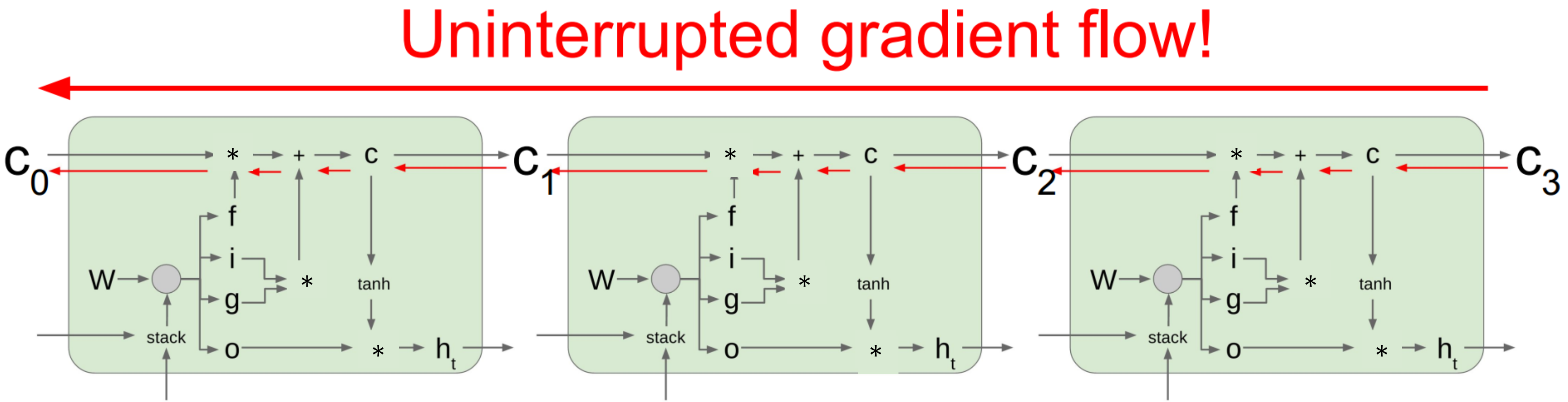
Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \ast c_{t-1} + i \ast g$$

$$h_t = o \ast \tanh(c_t)$$

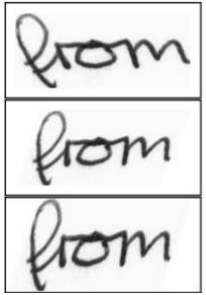
Gradient calculation in LSTM



- Though we multiply the memory content with a smaller than 1 number
- And the W matrix is part of the memory update
- But it still preserves the content for longer time
- As it comes from the name: It is a elongated time **short term** memory

Achevements with LSTM networks

- Record results in natural language text compression
- Unsegmented connected handwriting recognition

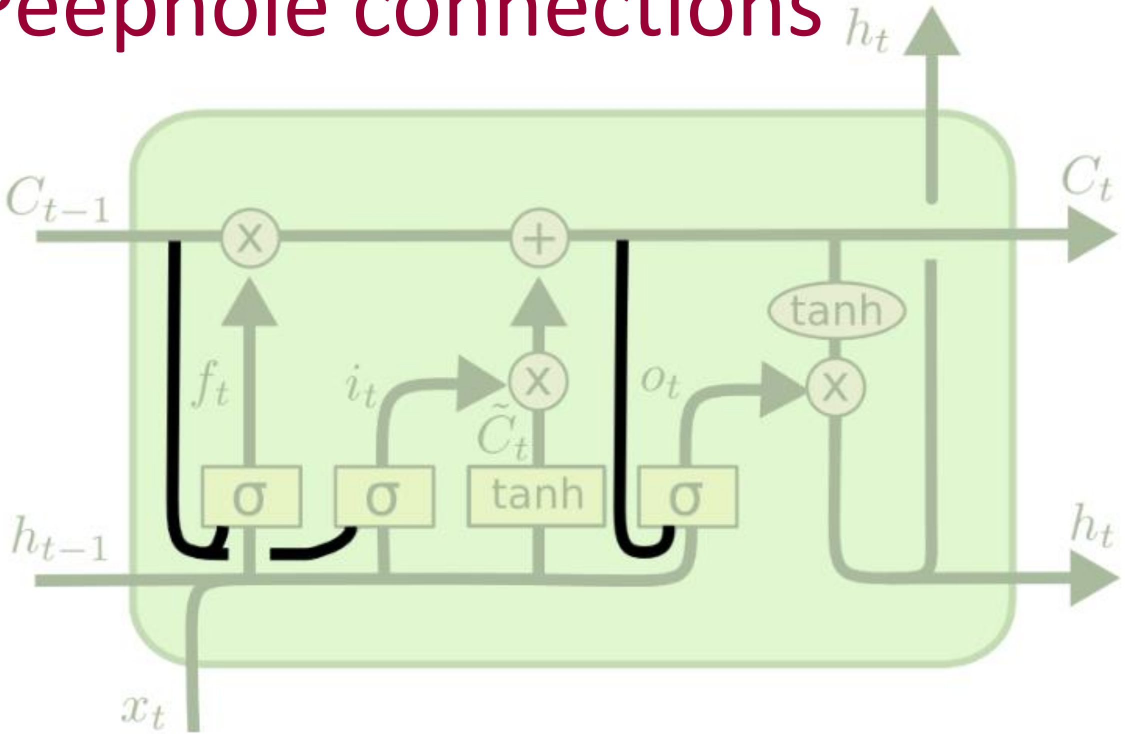


- Natural speech recognition
- Smart voice assistants
  - Google Translate
  - Amazon Alexa
  - Microsoft Cortana
  - Apple Quicktype

- 95.1% recognition accuracy on the Switchboard corpus, incorporating a vocabulary of 165,000 words
  - Continuous spontaneous English native speech

Variants of LSTM I : Peephole connections

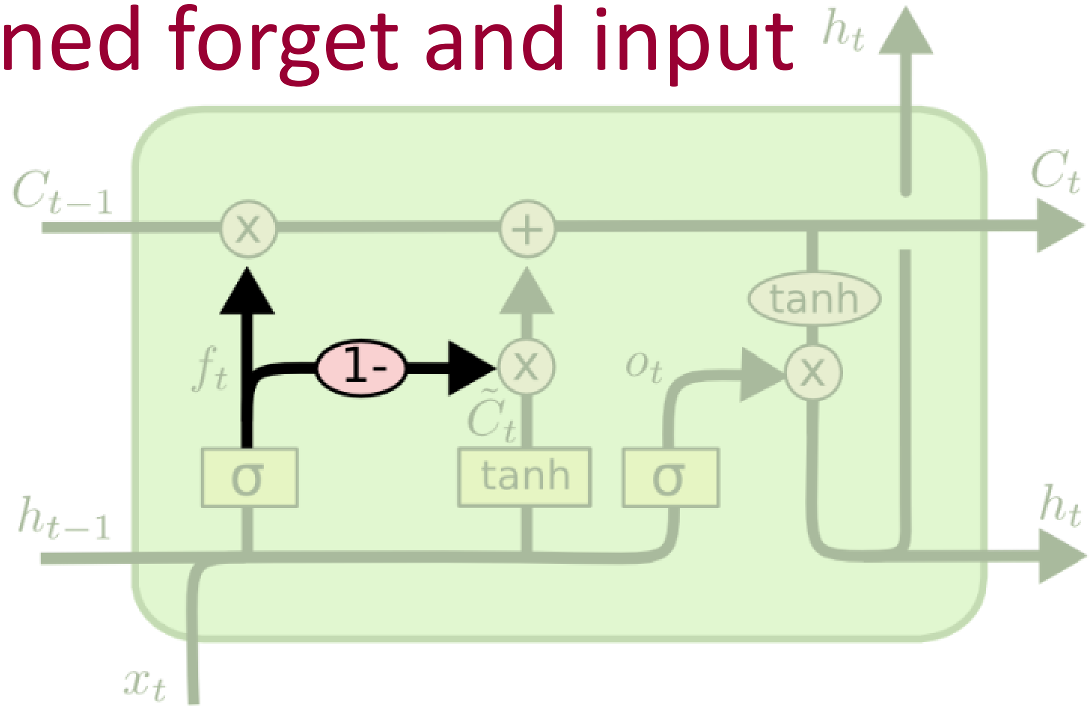
- Introduced by Gers & Schmidhuber (2000)
- All the three gates receives input from the previous state and the input
- Since output can be sparse this version has more information for gating
  - addressing and weighting



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)$$

Variants of LSTM II : Joined forget and input

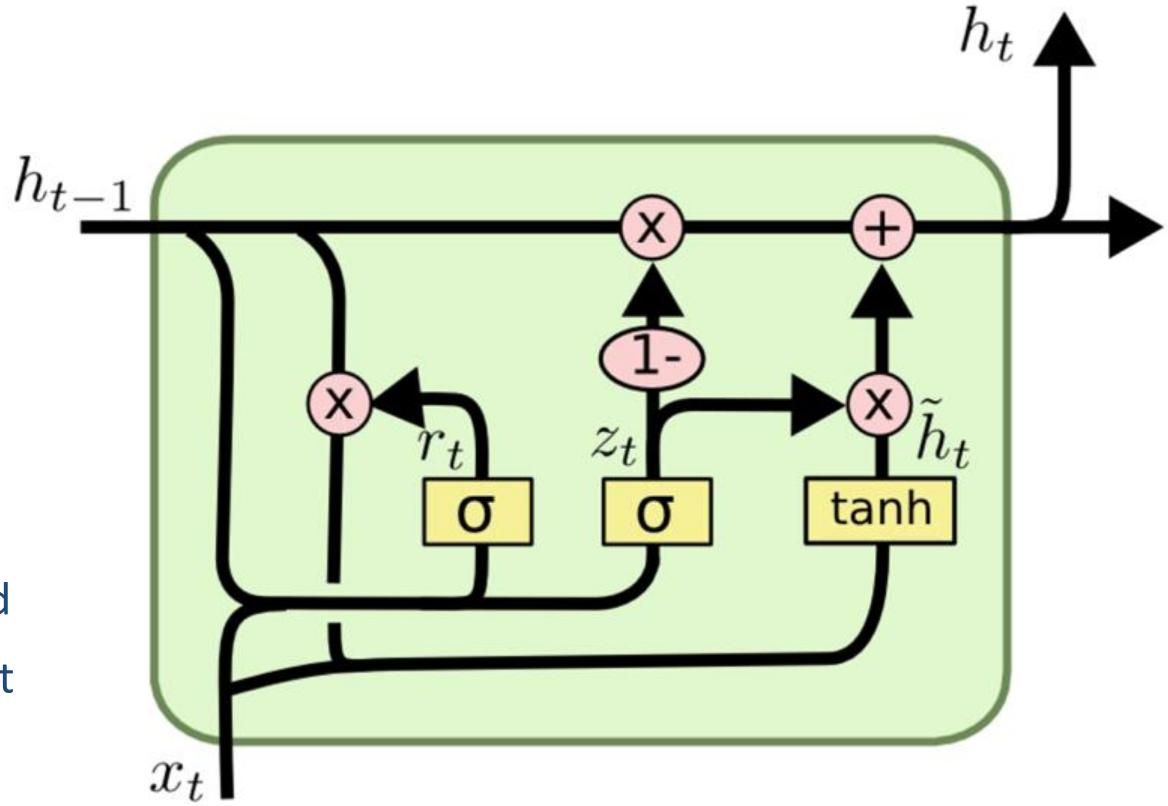
- Input and forget gates has practically the same role
- Why not to join them?



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

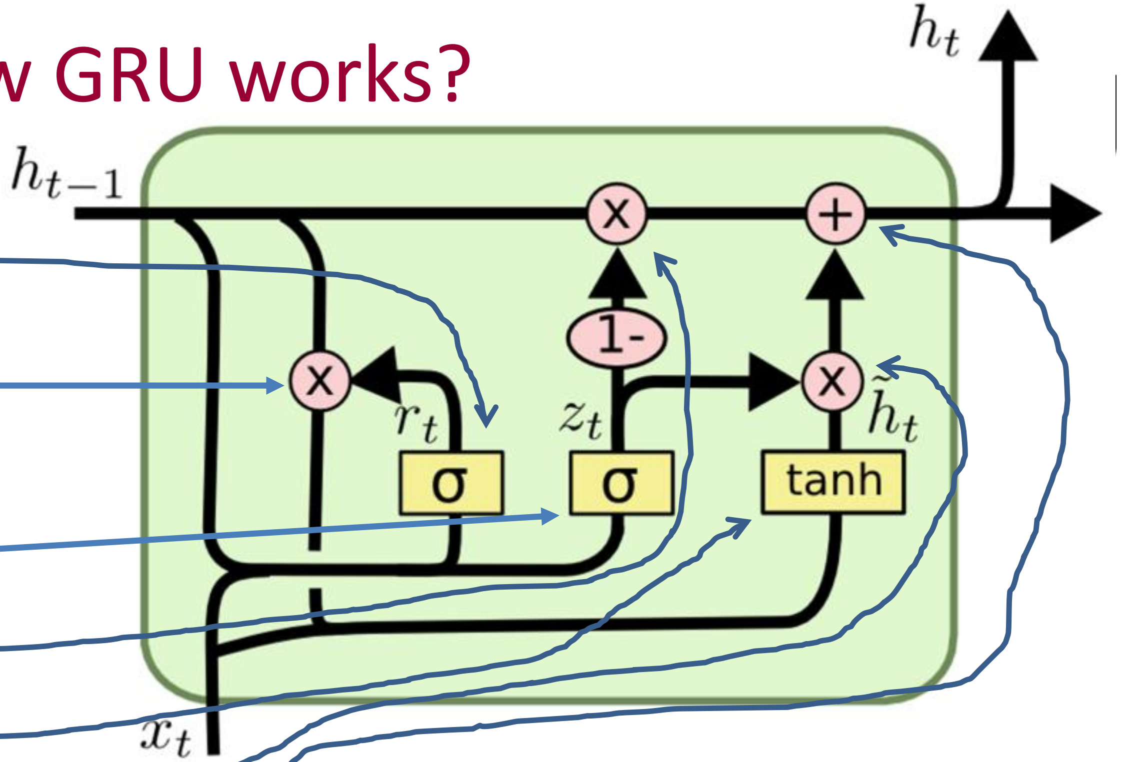
Gated Recurrent Unit (GRU)

- Another variant of LSTM
- Introduced by Kyunghyun Cho (2014)
- There is no separate State and Output
- Only three neural nets
- At GRU the output will not be sparse (not gated)
- Similar performance in music and speech signal modelling and
- Learns faster for smaller data set



How GRU works?

- Concatenate h_{t-1} and x_t
- Calculate the Input Gate
- Suppress the values to be forgotten in h_{t-1} (get sparse memory vector)
- Calculate the joint Forgot and output Gates
- Gate h_{t-1}
- Calculate function of the Cell Network
- Gate $\tilde{h}_{t-1}$
- Calculate the new output (h_t)



$$r_t = \sigma(W_r[h_{t-1}, x_t])$$

$$z_t = \sigma(W_z[h_{t-1}, x_t])$$

$$\tilde{h}_t = tanh(W_c[r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Neural Networks
(P-ITEEA-0011)

Famous architectures

András Horváth,   *Ákos Zarándy*

Budapest, 2019.12.03

Administrative announcements

- Replacement paper-based test 17. 12. 9:00, Room 418
  - papíros pót ZH - dec. 17 9:00, 418-as terem
- Early exam 17. 12. 9:00, Room 419
  - The invited students will be emailed acknowledged this week

  Early exam - dec. 17 9:00, 419-es terem,
  érintettek a héten megtudják meg
- Project presentation - 17. 12. 11:00, Room 418
  Projekt bemutatás - dec. 17 11:00, 418-as terem
-
- Computer-based test - 19. 12. 9:00
  Géptermi ZH - dec. 19 9:00
-
- Computer-based replacement test TBA, early January
  Géptermi pót TBA, ~január eleje
-
- Oral Exams are already in the Neptun system
  Vizsgaidőpontok a Neptunban

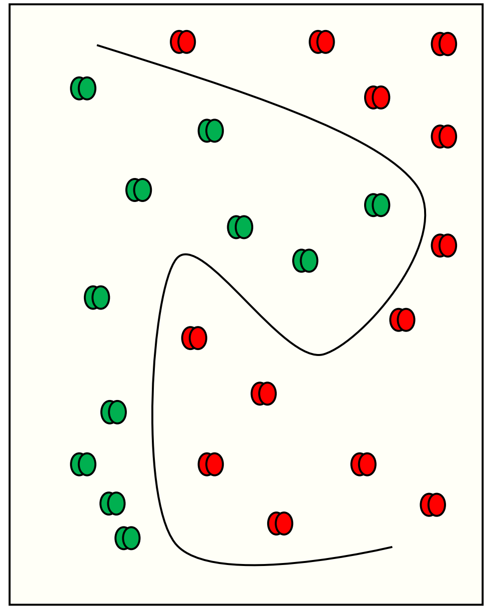
We are considering to create a list of the participants, to reduce waiting time for the oral exam.

Neural Networks

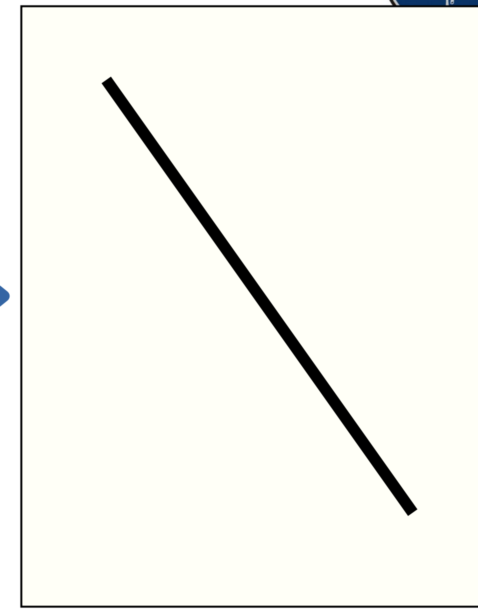
- Classification - decision
- FNN, SVM – linear classification
- Is X larger than a limit? X>k?

- Finding a good feature representation:
  - Meaningful
  - Sparse - low dimensions
  - Ensures easy separation

- Finding the representation with the help of machine learning



- Input *space*

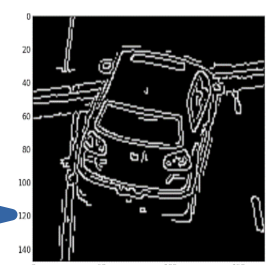


Feature space

Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

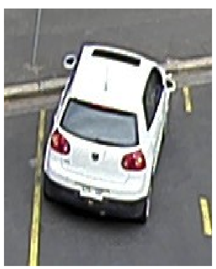
Input Image



Feature Image

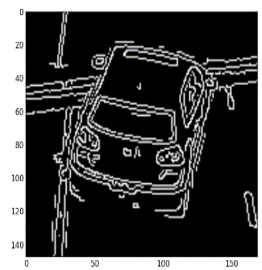
Convolutional neural networks

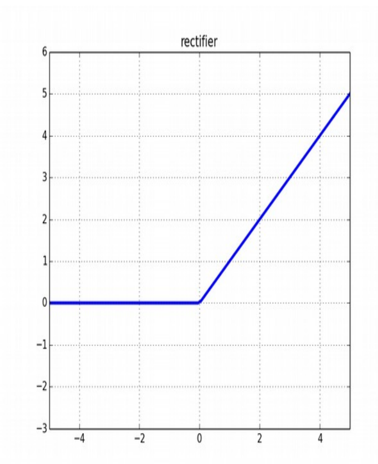
- A network of simple processing elements
  - Elements:
    - Convolution
    - ReLU
    - Pooling



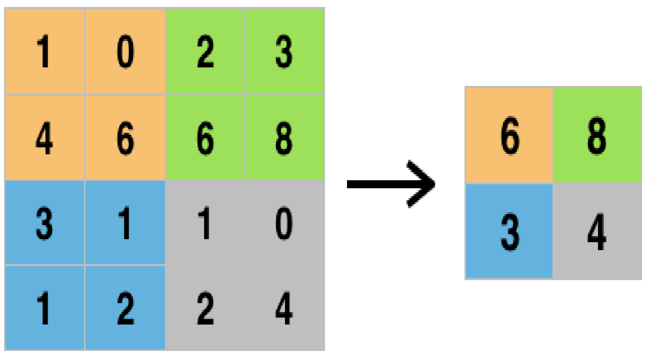
Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

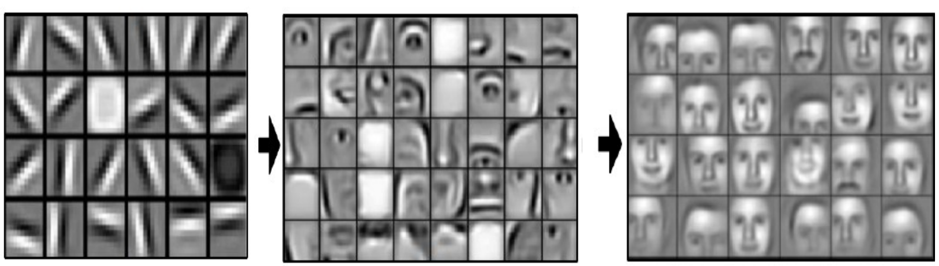




Thresholding all values below zero



Selection of the maximal response in an area



Low layers *Middle layers* *High layers*

Convolutional networks

Assume, I have a problem to solve.

Ok, but how many layers do we need?

How many features should be in each layer?

What should be the network architecture?

Convolutional networks

Assume, I have a problem to solve.

Ok, but how many layers do we need?

How many features should be in each layer?

What should be the network architecture?

These are called hyper-parameters:

Along with: non-linearity type, batch-norm, dropout etc.

Convolutional networks

Assume, I have a problem to solve.

Ok, but how many layers do we need?

How many features should be in each layer?

What should be the network architecture?

These are called hyper-parameters:

Along with: non-linearity type, batch-norm, dropout etc.

We can use a network which performed fairly well on an other dataset

It will probably work well on our task too

Alexnet

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012)

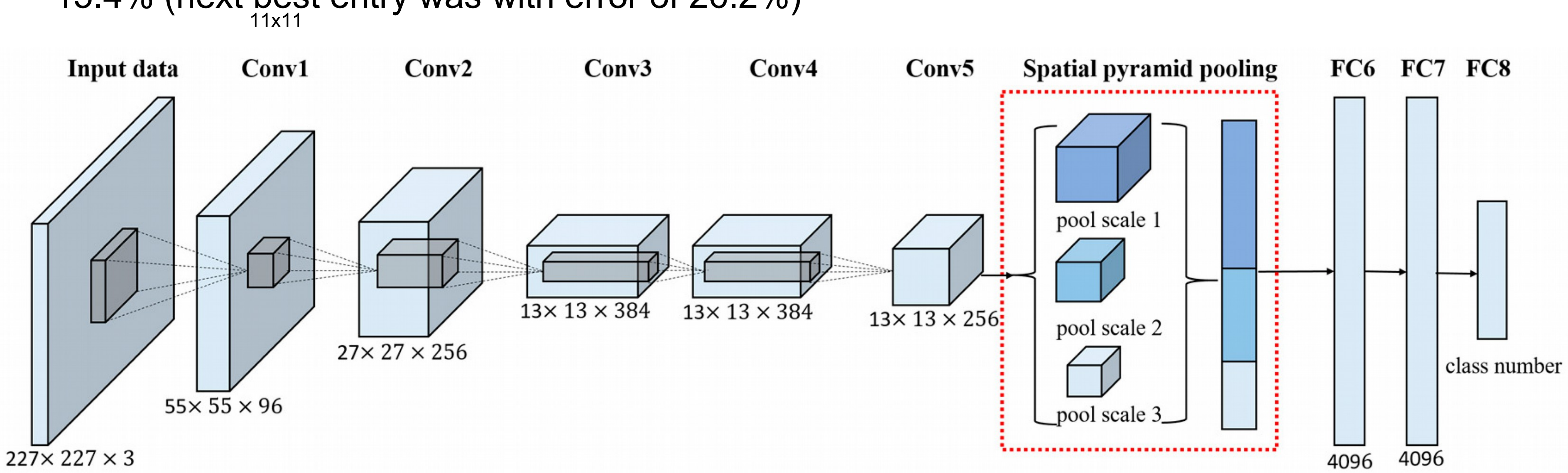
Trained whole ImageNet (15 million,22,000 categories)

Used data augmentation (image translations, horizontal reflections, and patch extractions)

Used ReLU for the nonlinearity functions (Decreased training time compared to tanh) - Trained on two GTX 580 GPUs for six days

Dropout layers

2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4% (next best entry was with error of 26.2%)



VGG - 16/19

Karen Simonyan and Andrew Zisserman of the University of Oxford, 2014 Visual Geometry Group

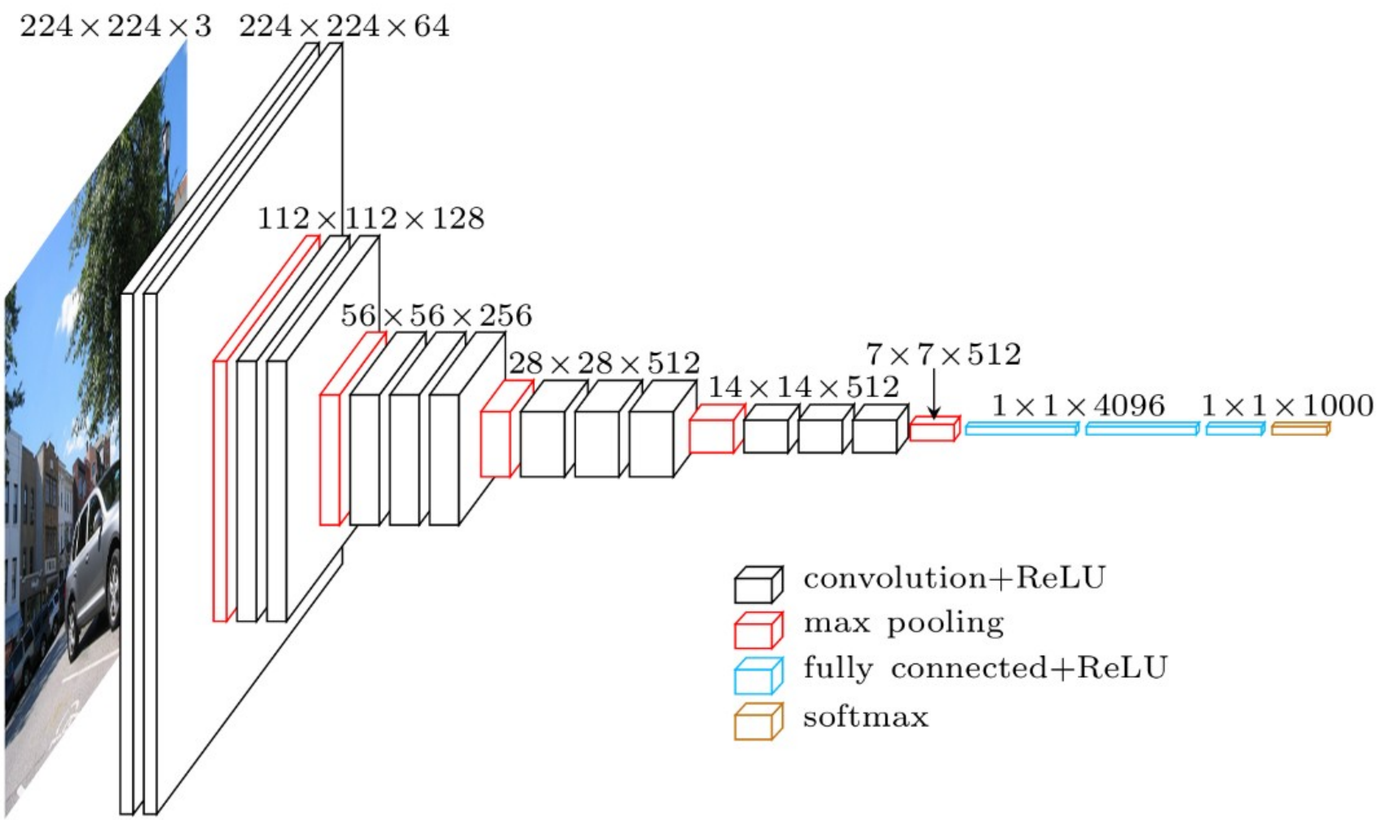
As the spatial size of the input volumes at each layer decrease (result of the conv and pool layers), the depth of the volumes increase due to the increased number of filters as you go down the network.

Shrinking spatial dimensions but grwoing depth

3x3 filters with stride and pad of 1, along with 2x2 maxpooling layers with stride 2

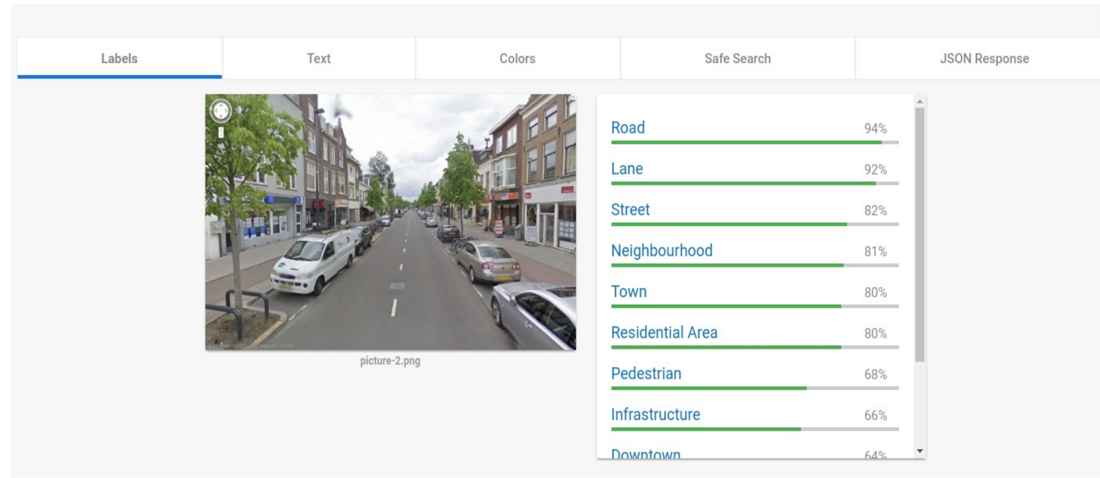
7.3% error rate

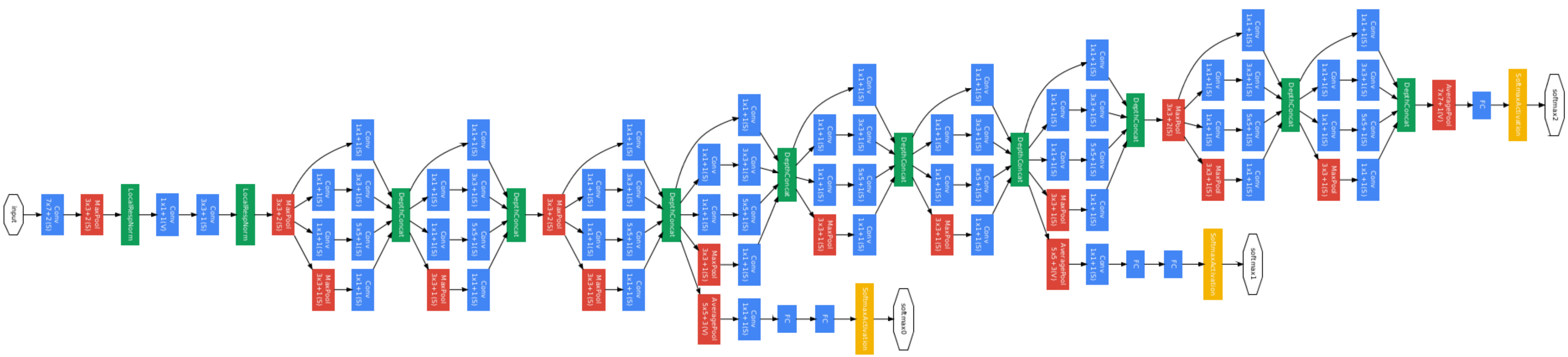
Simple architecture, still the swiss knife of deep learning



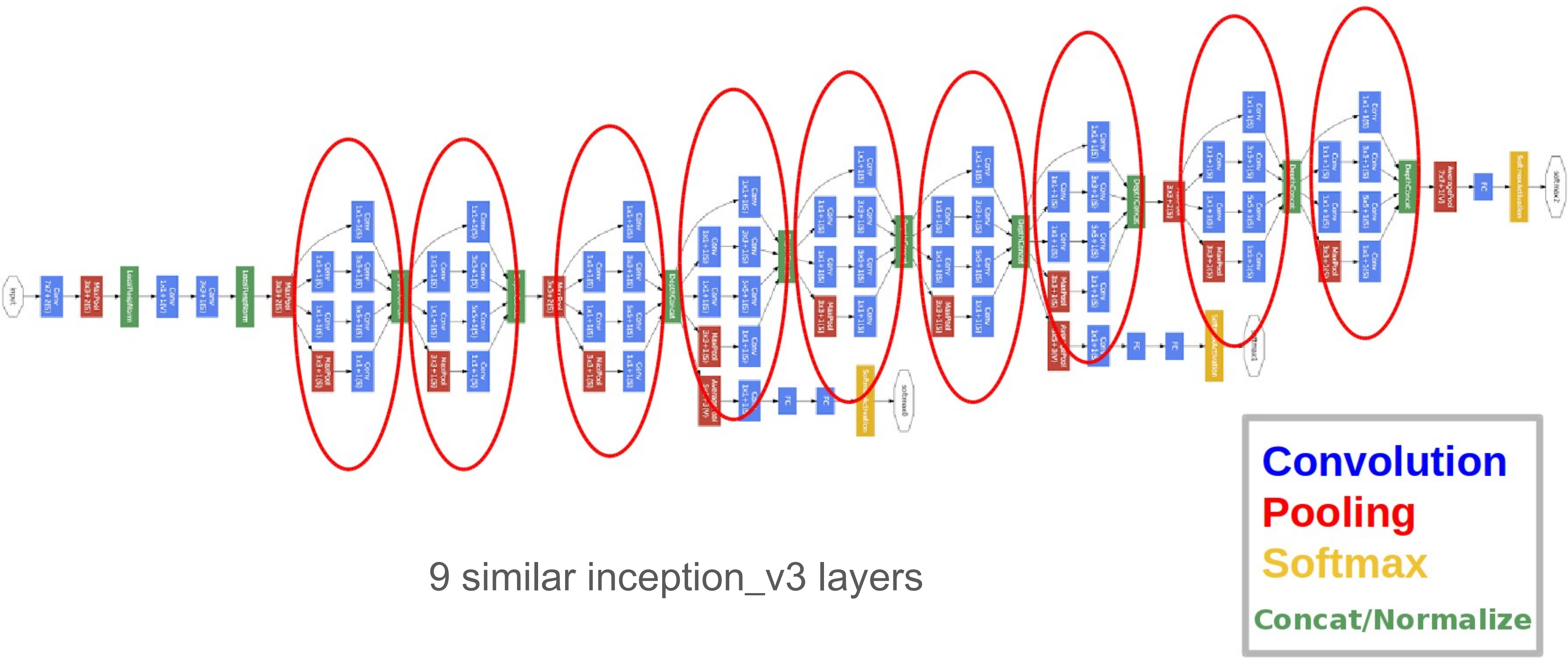
Google - Inception arhcitecture

- GoogLeNet:
22/42 layers (9 inception_v3 layers)
5 million free parameters
~1.5B operations/evaluations
Demo:https://cloud.google.com/vision/





Inception module



9 similar inception_v3 layers

Inception

Google, Christian Szegedy

2014 with a top 5 error rate of 6.7%

This can be thought of as a “pooling of features” because we are reducing the depth of the volume, similar to how we reduce the dimensions of height and width with normal maxpooling layers.

Idea:

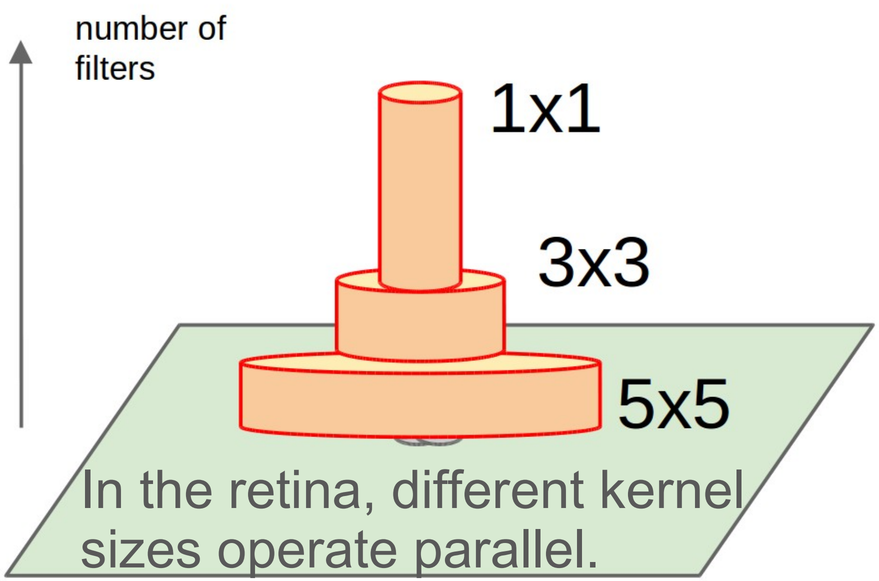
Not to introduce different size kernels in different layers, but introduce 1x1, 3x3, 5x5 in each layers, and let the Neural Net figure out, what representation is the most useful, and use that!

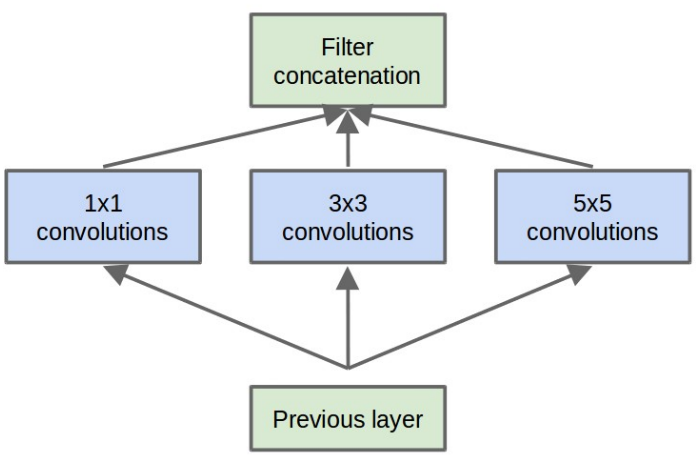
Parallel multi-scale approach.

AlexNet: 60 million param

VGGNet: 1800 million

GoogLeNet / Inception-v1





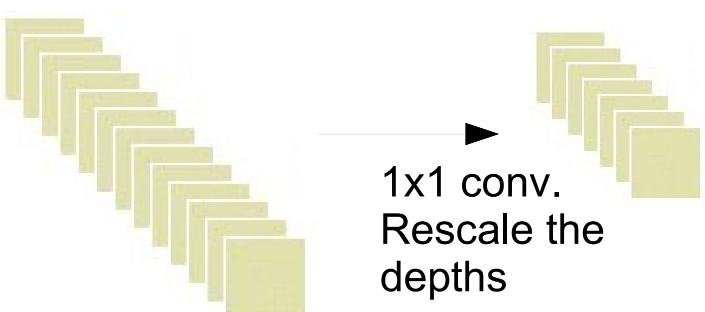
Rethinking Inception

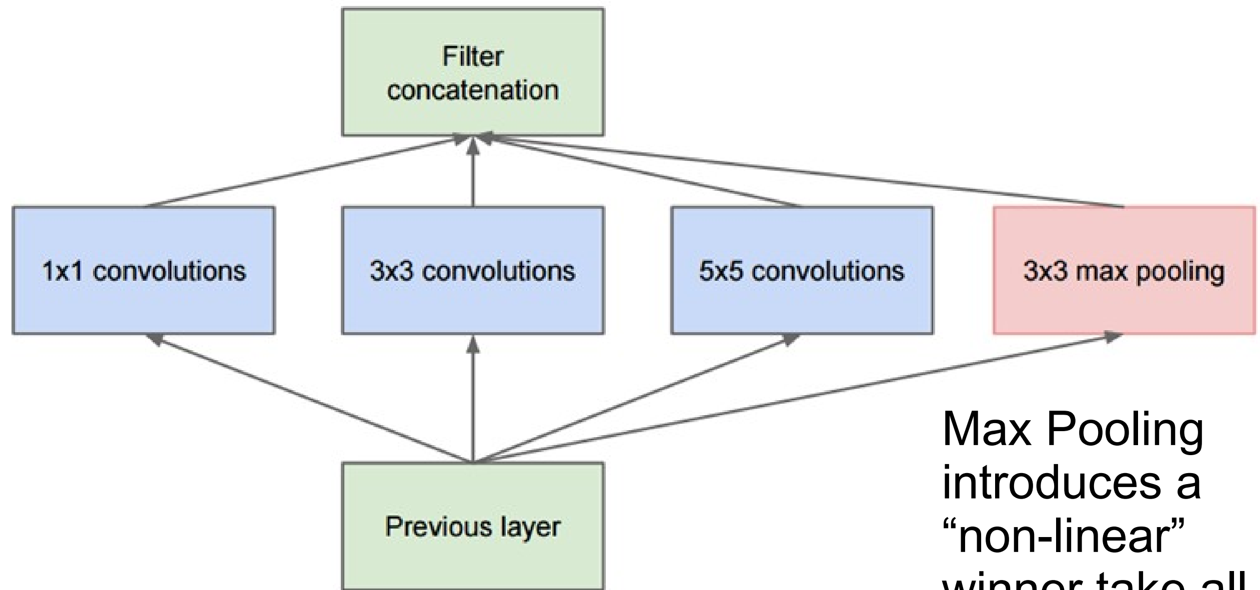
Squeezing the number of channels for each kernel

With the concatenations, the number of features increased in each layers, which introduced too many convolution.

To reduce these numbers, they introduced the 1x1 layer.

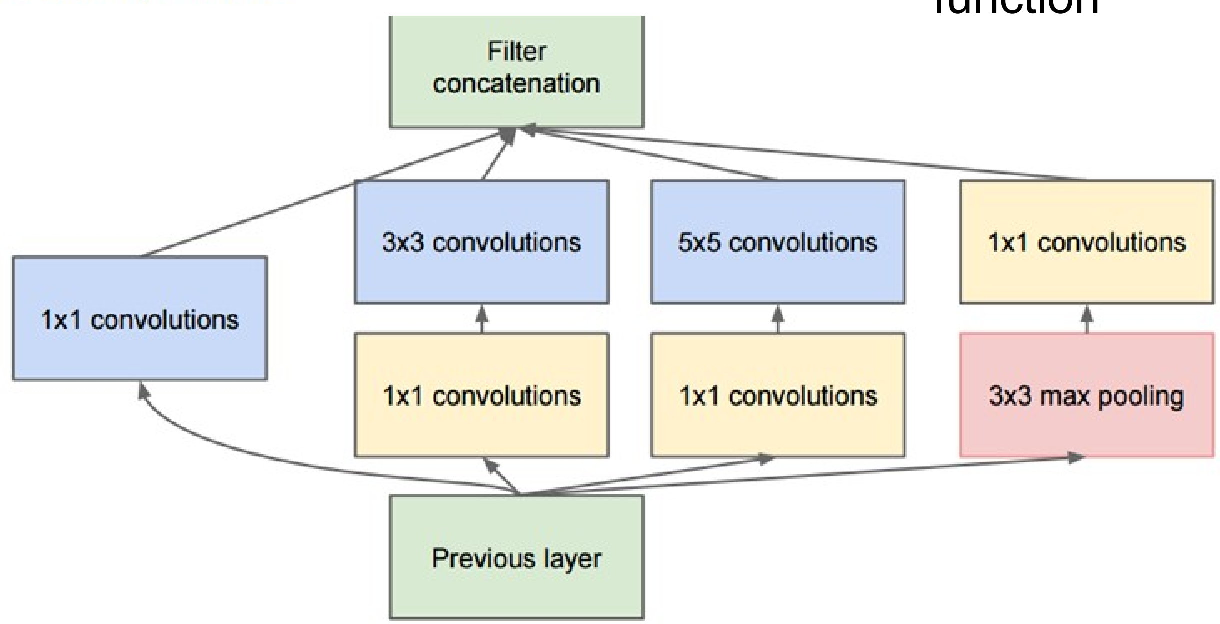
It can generate e.g. 16 feature maps from 64 feature maps





Max Pooling introduces a “non-linear” winner take all function

Naïve idea of an Inception module

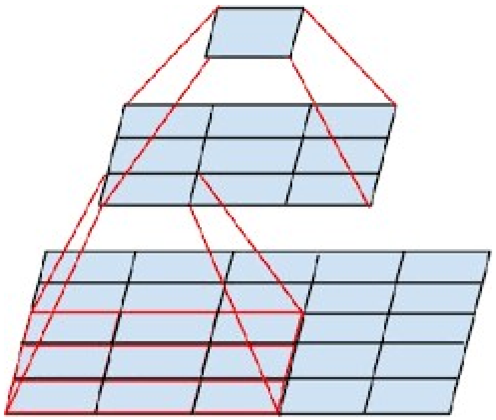


Full Inception module

Rethinking Inception

Larger (5x5) convolutions were substituted by series of 3x3 convolutions

Advantages:
1. Reduction of number of parameters,
2. Additional non-linearities (RELUs) can be introduced



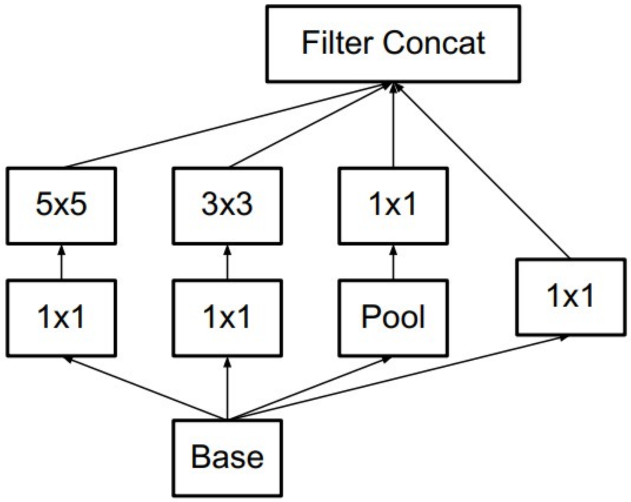
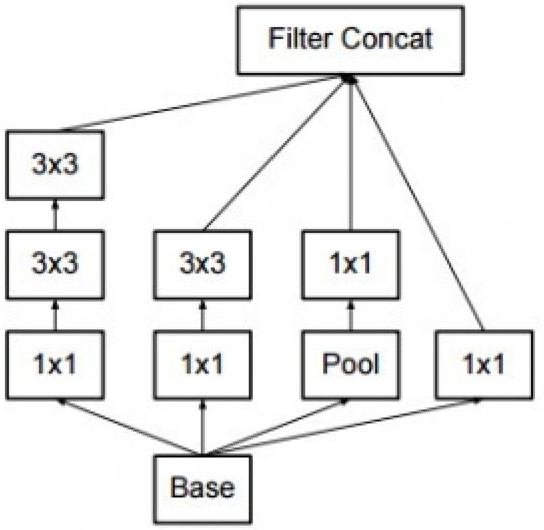
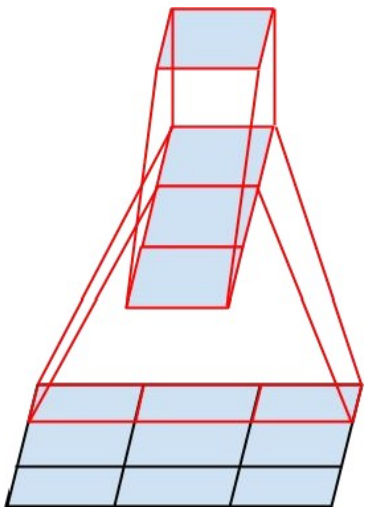


Figure 4. Original Inception module as described in [20].



Rethinking Inception



Larger convolutions were substituted by series of 3x3 convolutions

2D convolution were substituted by two 1D convolutions

AlexNet: 60 million parameters
VGGNet :180 million parameters
GoogLeNet / Inception-v3: 7 million parameters

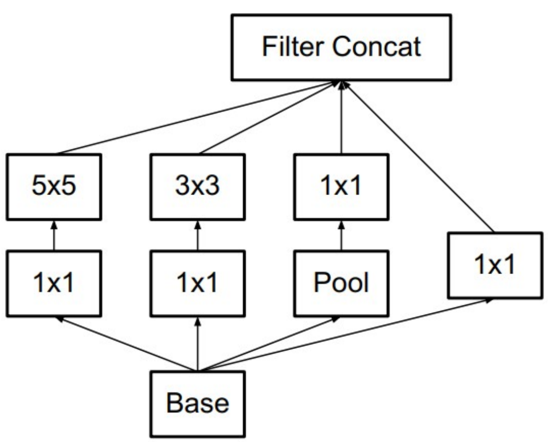
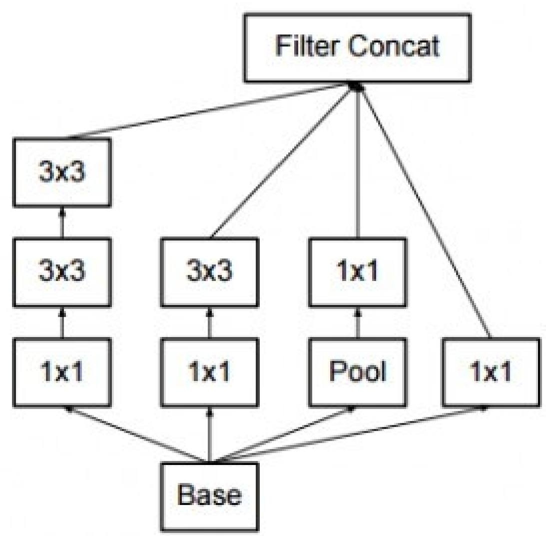
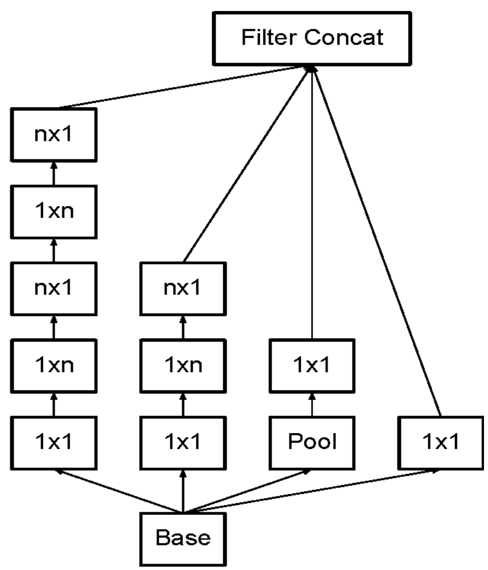
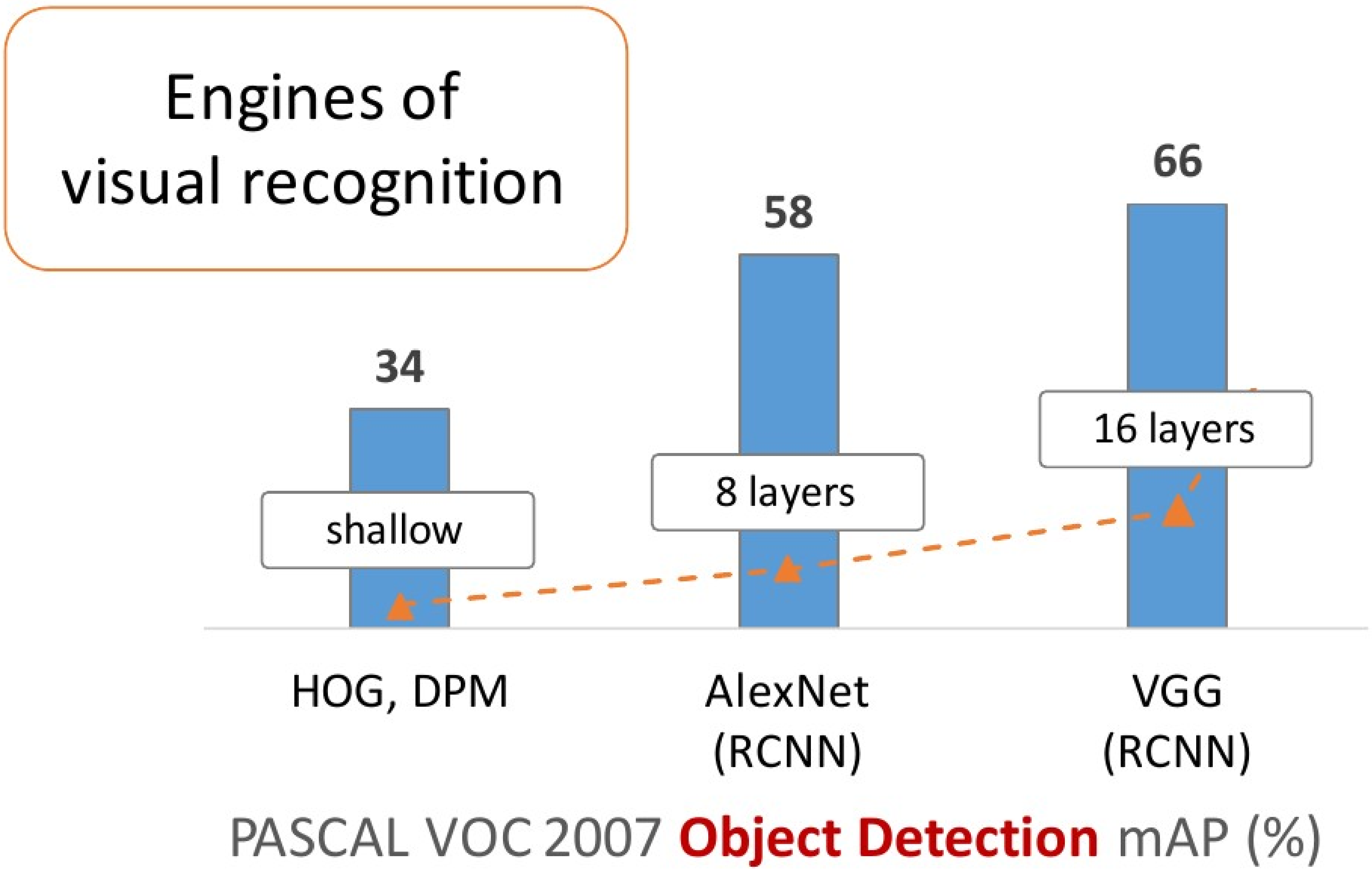


Figure 4. Original Inception module as described in [20].





Revolution of Depth



How deep could/should a network be?

History of network depth

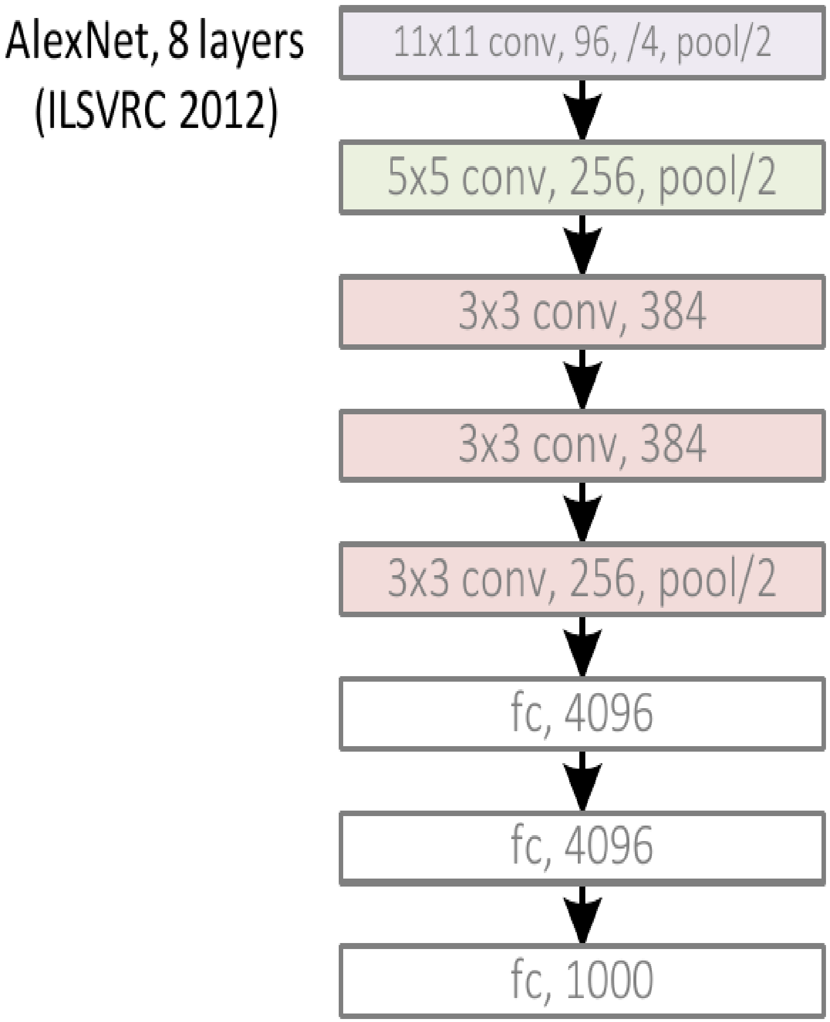
Before 2012: four layers

How deep could/should a network be?

History of network depth

Before 2012: four layers

2012: 8layers



How deep could/should a network be?

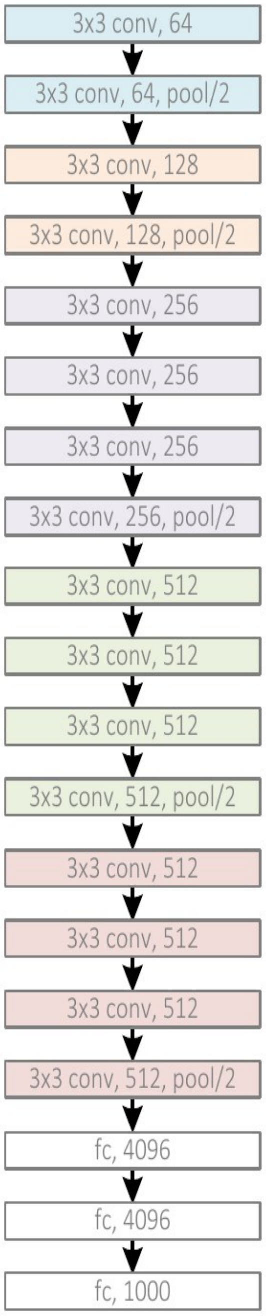
History of network depth

Before 2012: four layer

2012: 8layers

2014: 19 layers

VGG, 19 layers
(ILSVRC 2014)



How deep could/should a network be?

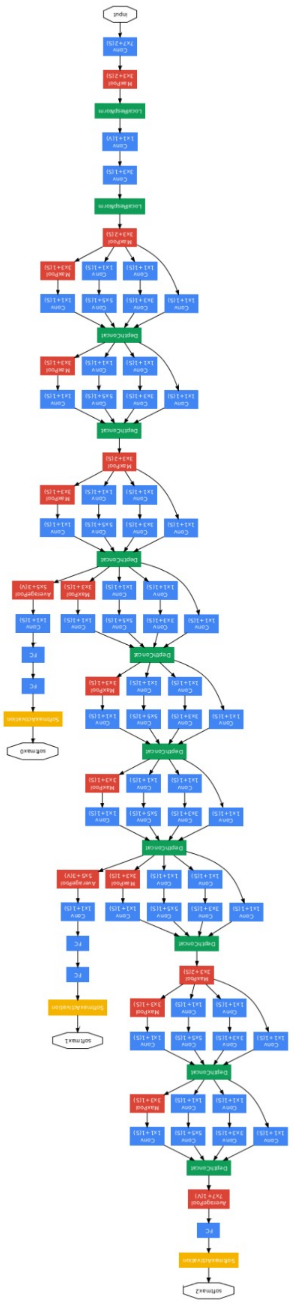
History of network depth

Before 2012: four layer

2012: 8layers

2014: 19 layers

2016: 19-22 layers



How deep could/should a network be?

History of network depth

Before 2012: four layer

2012: 8layers

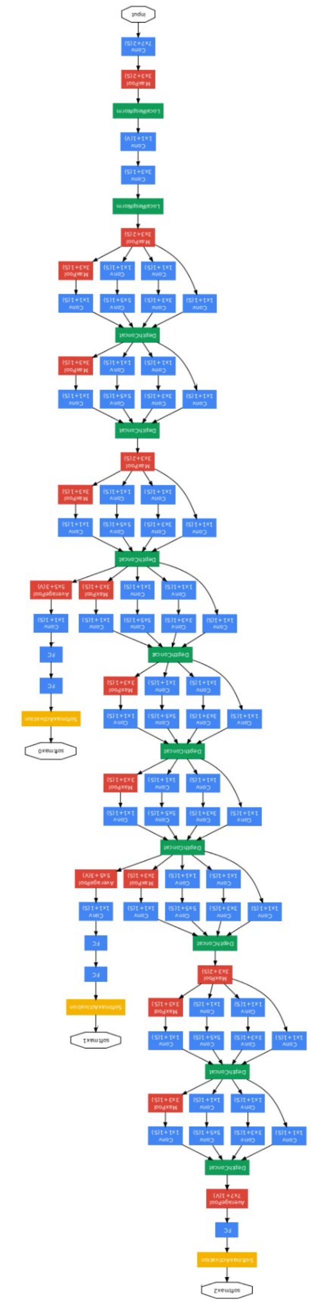
2014: 19 layers

2016: 19-22 layers

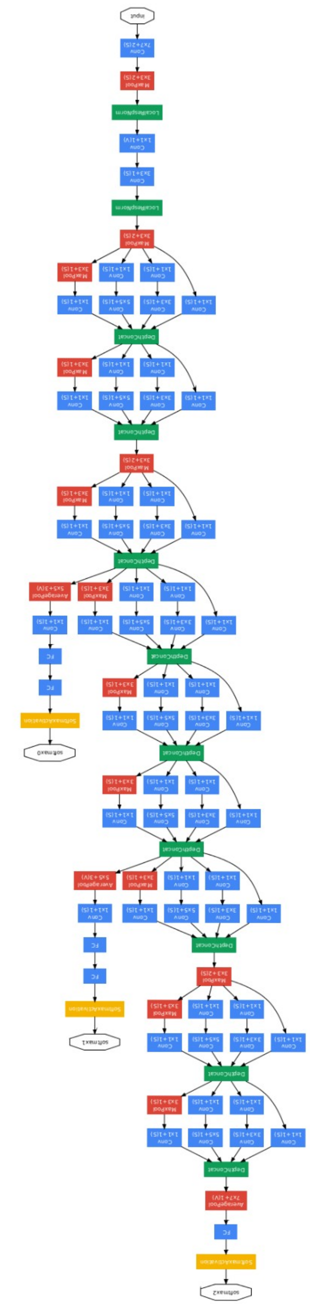
Deeper network:

Possibility to approximate more complex functions

Higher number of parameters



How deep could/should a network be?



History of network depth

Before 2012: four layer

2012: 8layers

2014: 19 layers

2016: 19-22 layers

Deeper network:

Possibility to approximate more complex functions

Higher number of parameters

There are no convolutional networks with more than 30 layers. Why?

The amount of transfered data is decreased from layer to layer

Training becomes difficult

Is a deeper network always better?

A deeper network would have higher approximation power

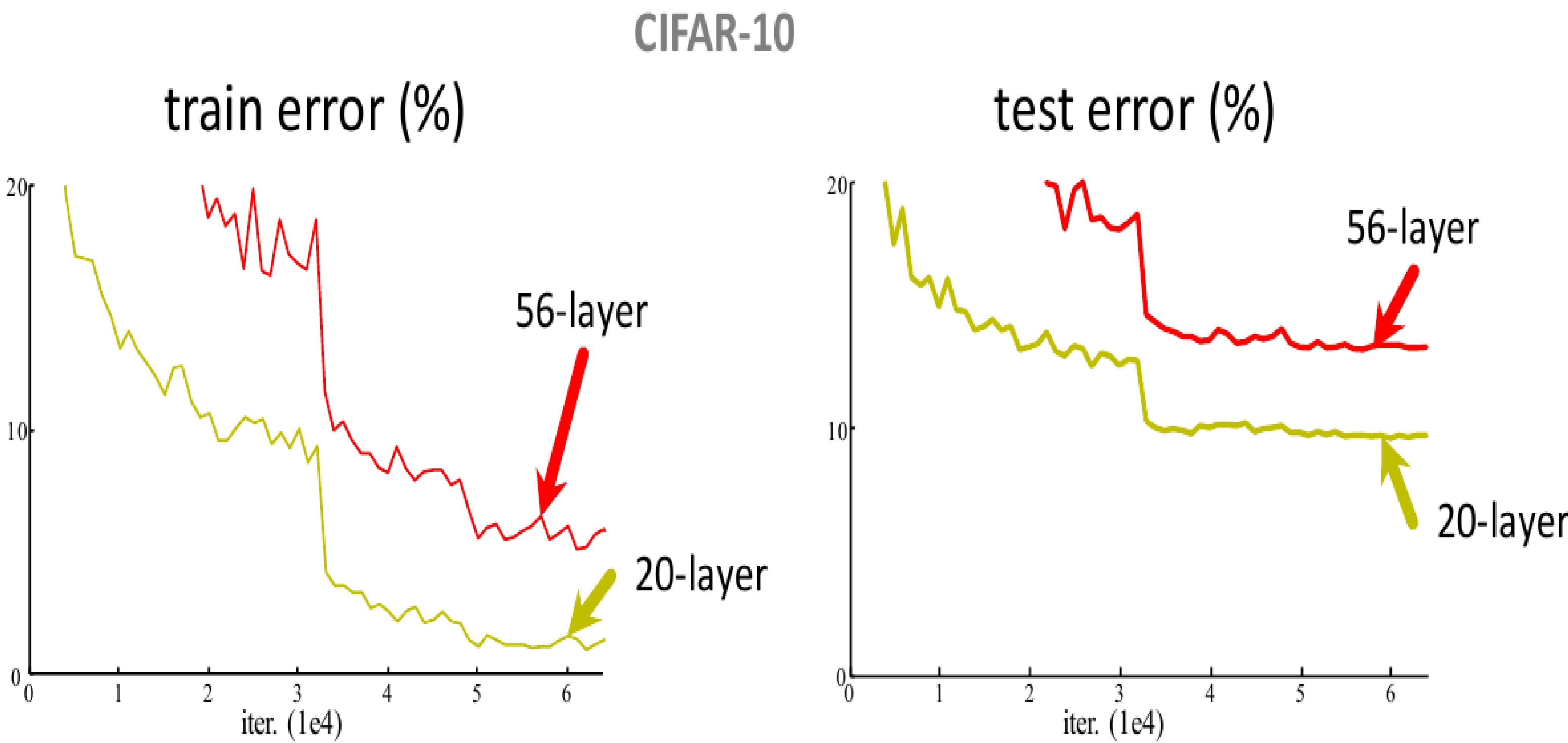
Higher number of parameters (both advantageous and disadvantageous)

Difficult to train the network

Is a deeper network always better?

A deeper network always has the potential to perform better, but training becomes difficult

After a given depth, the same network with the same training on the same data,



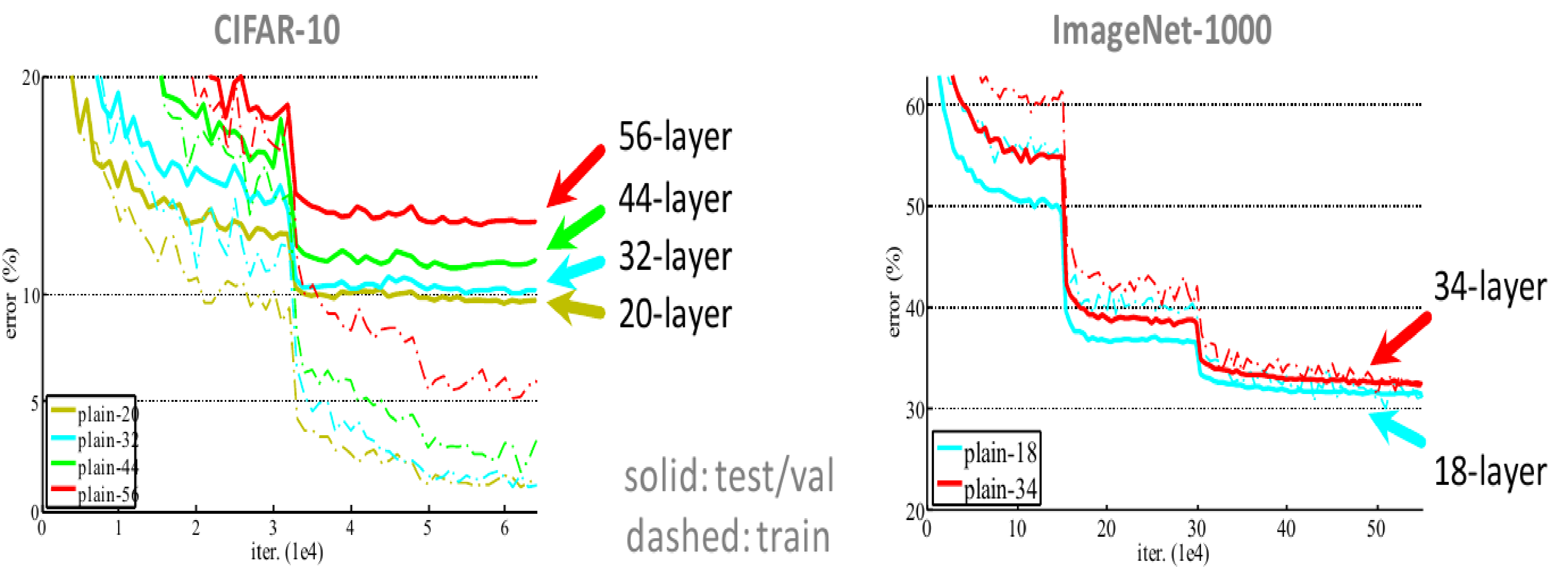
Is a deeper network always better?

A deeper network always have the potential to perform better, but training becomes difficult

We can not just simply stack convolutional layers to increase accuracy

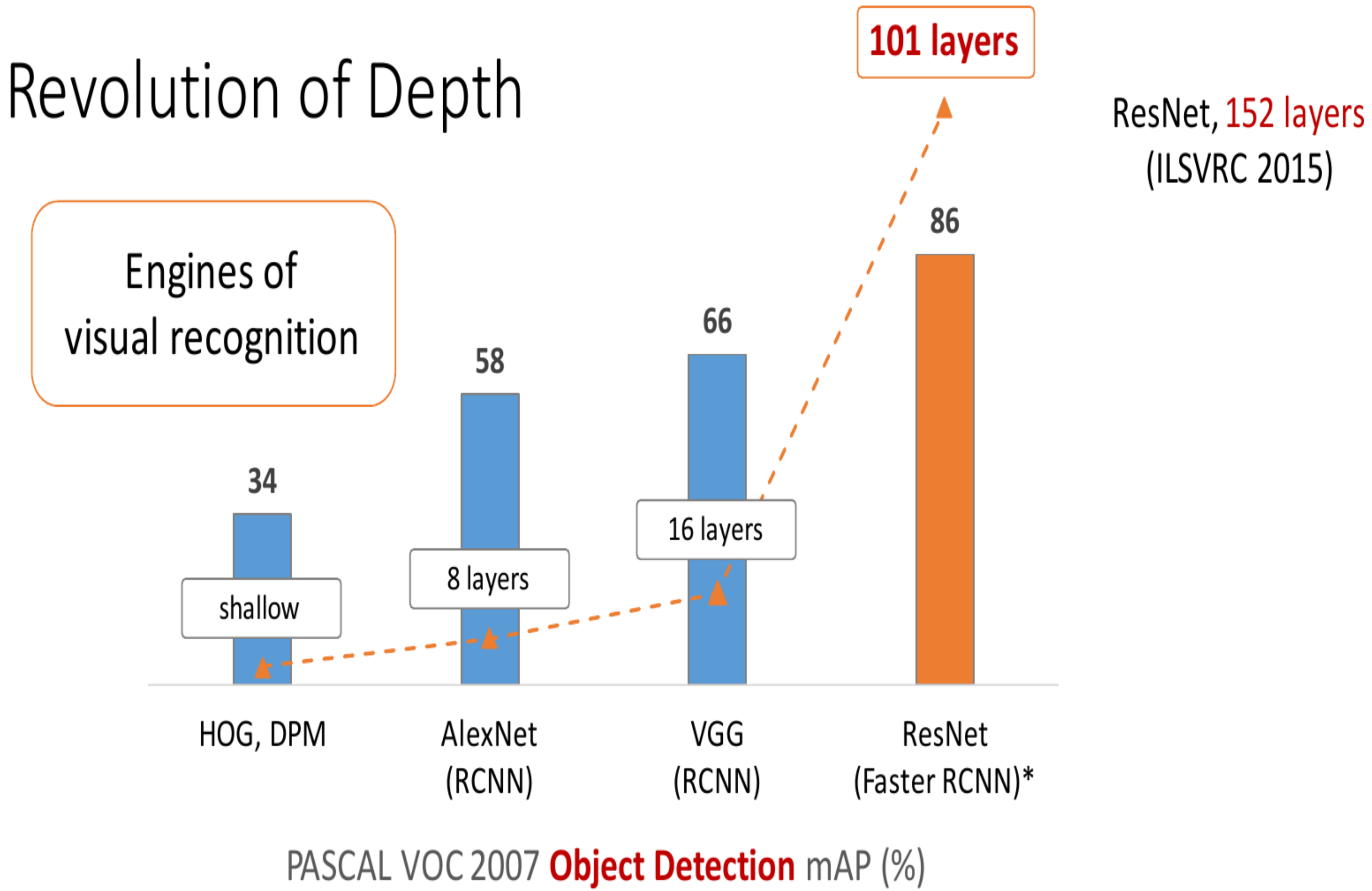
The backpropagated error will be smaller than the floating point accuracy limit.

The gradient will be disappear. The information will not pass the first layers, because there will be random noises on the weights, and they will not be trained.



How deep could a network be?

Residual networks provide an answer to these questions

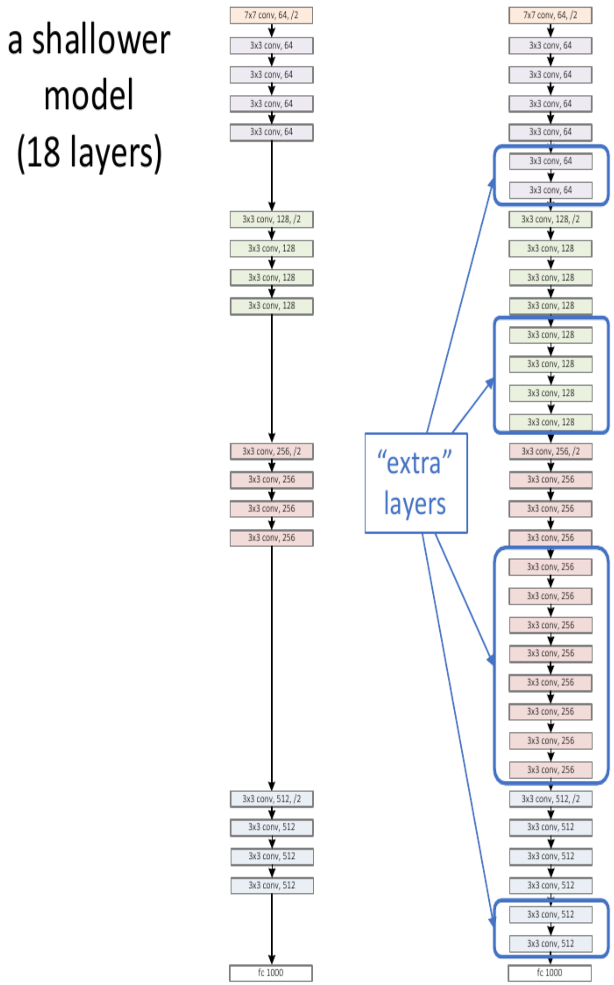


How could we create deeper networks?

A deeper network always have the potential to perform better, but training becomes difficult

How could we ensure that additional layers will not decrease accuracy (might even increase it)?

Let's start with a shallow model (18 layers) and add some extra layers (which we hope could increase accuracy)

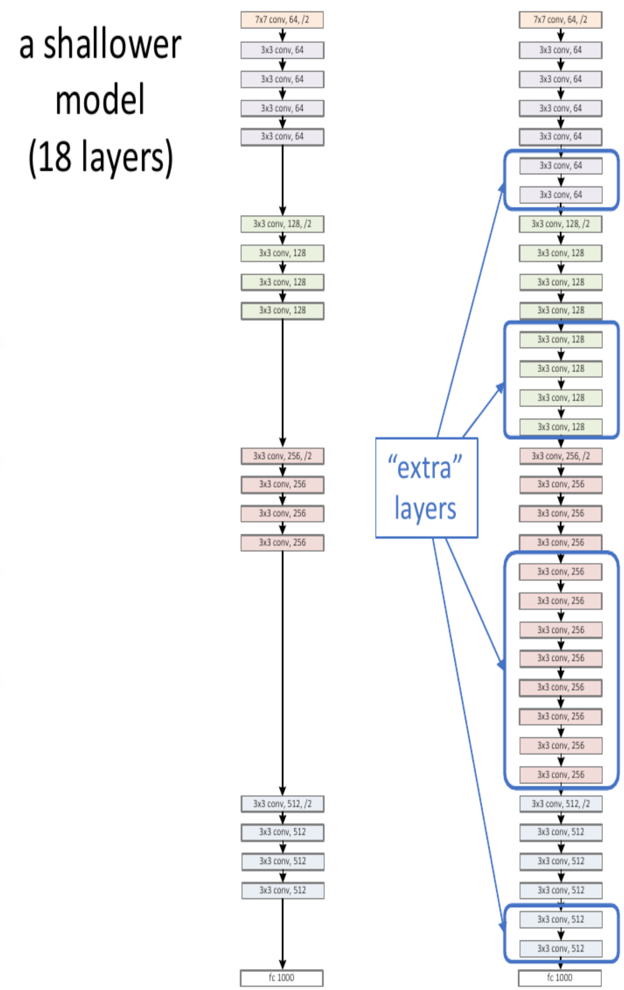


How could we create deeper networks?

A deeper network always have the potential to perform better, but training becomes difficult

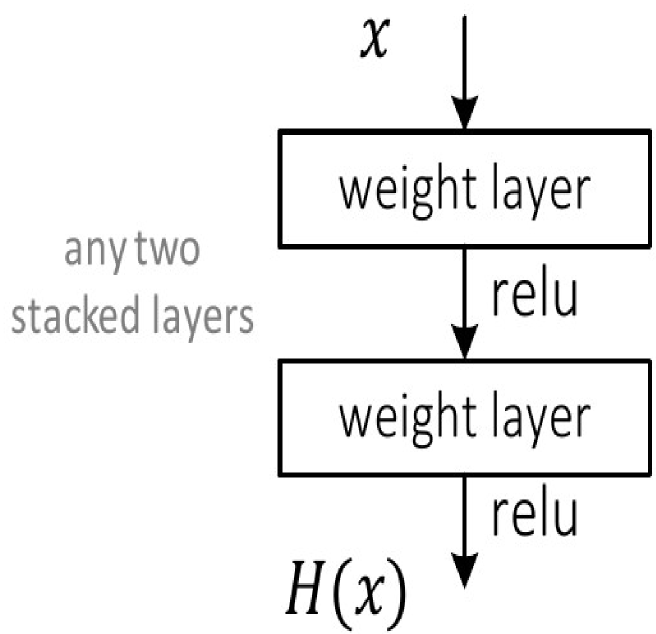
How could we ensure that additional layers will not decrease accuracy (might even increase it)?

Let's start with a shallow model (18 layers) and add some extra layers (which we hope could increase accuracy)



Our aim is to add "useful" operations H(x)

 The problem is that H(x) can ruin our accuracy because vanishing gradients, overfit - extra parameters



How could we create deeper networks?

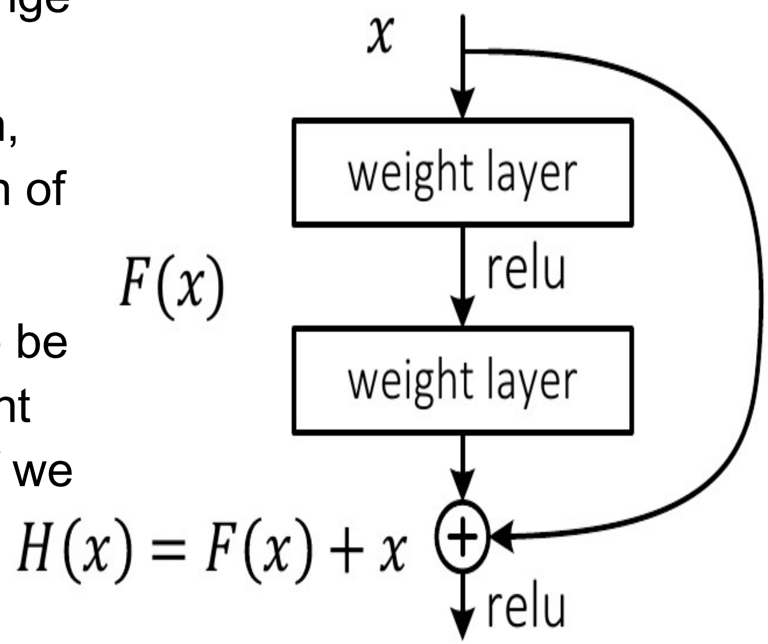
A deeper network always have the potential to perform better, but training becomes difficult

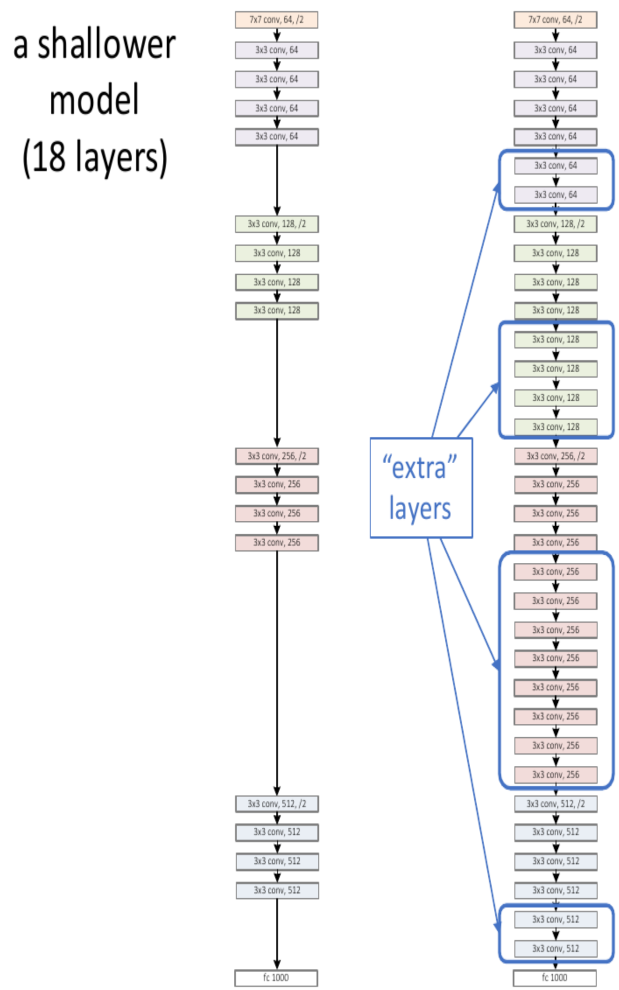
How could we ensure that additional layers will not decrease accuracy (might even increase it)?

The trick is to use residual connection and as a starting point F(x) could be zero, and H(x) becomes the dientity mapping

So H(X) will not change our performance, gradients will remain, because the addition of x

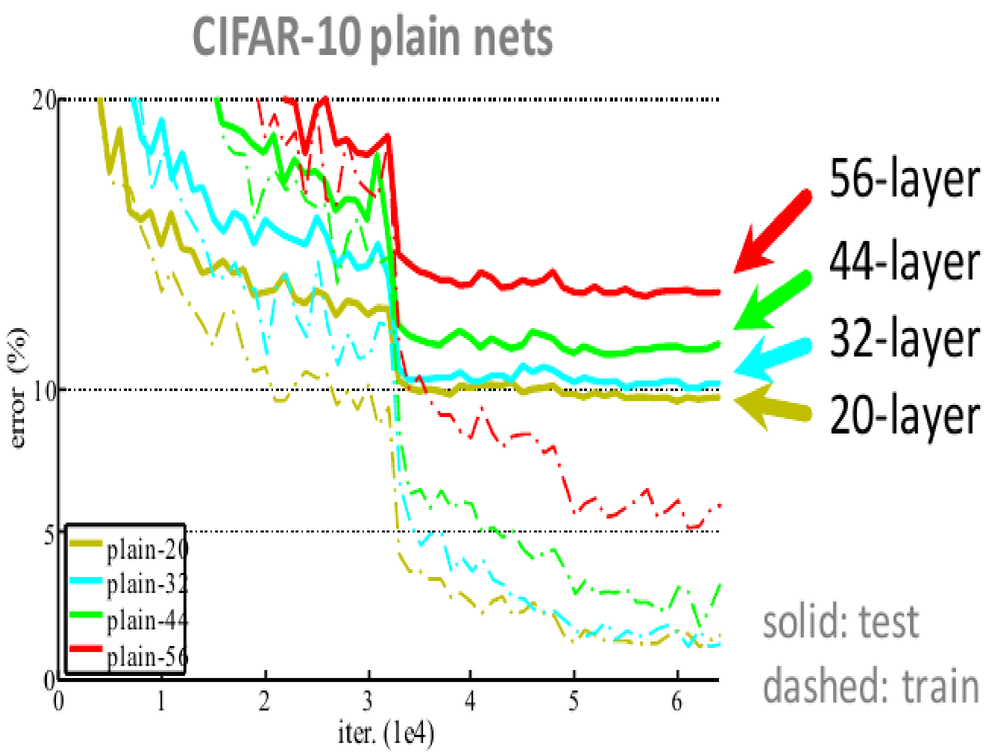
Our accuracy will no be decreased, and might even be increased if we find a proper F(x)

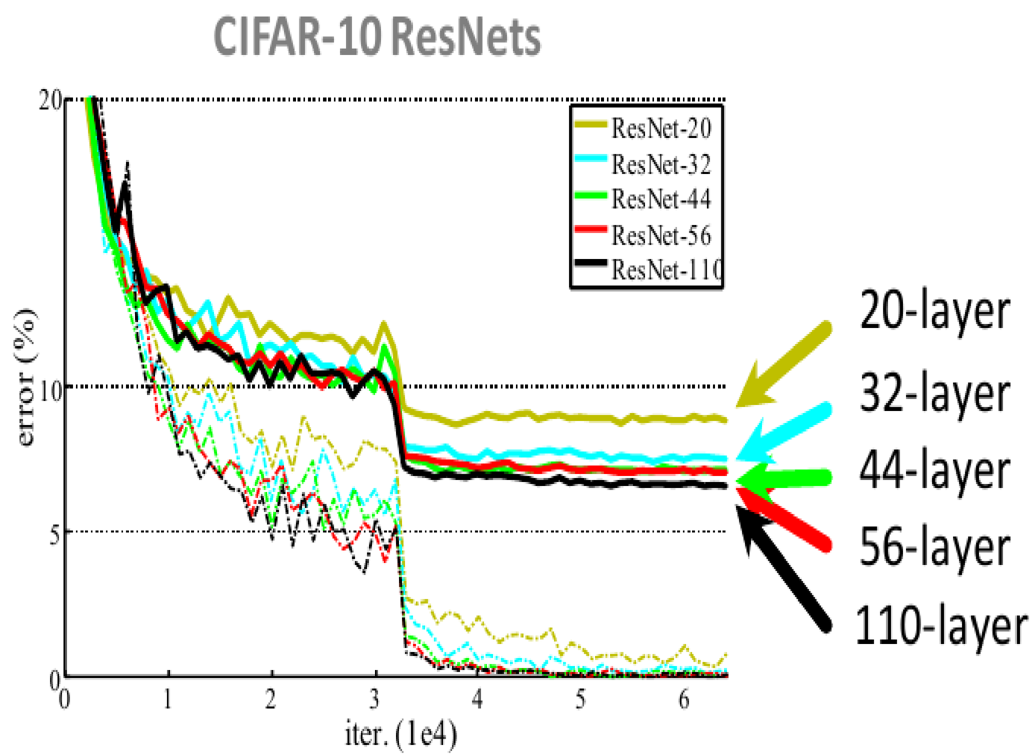




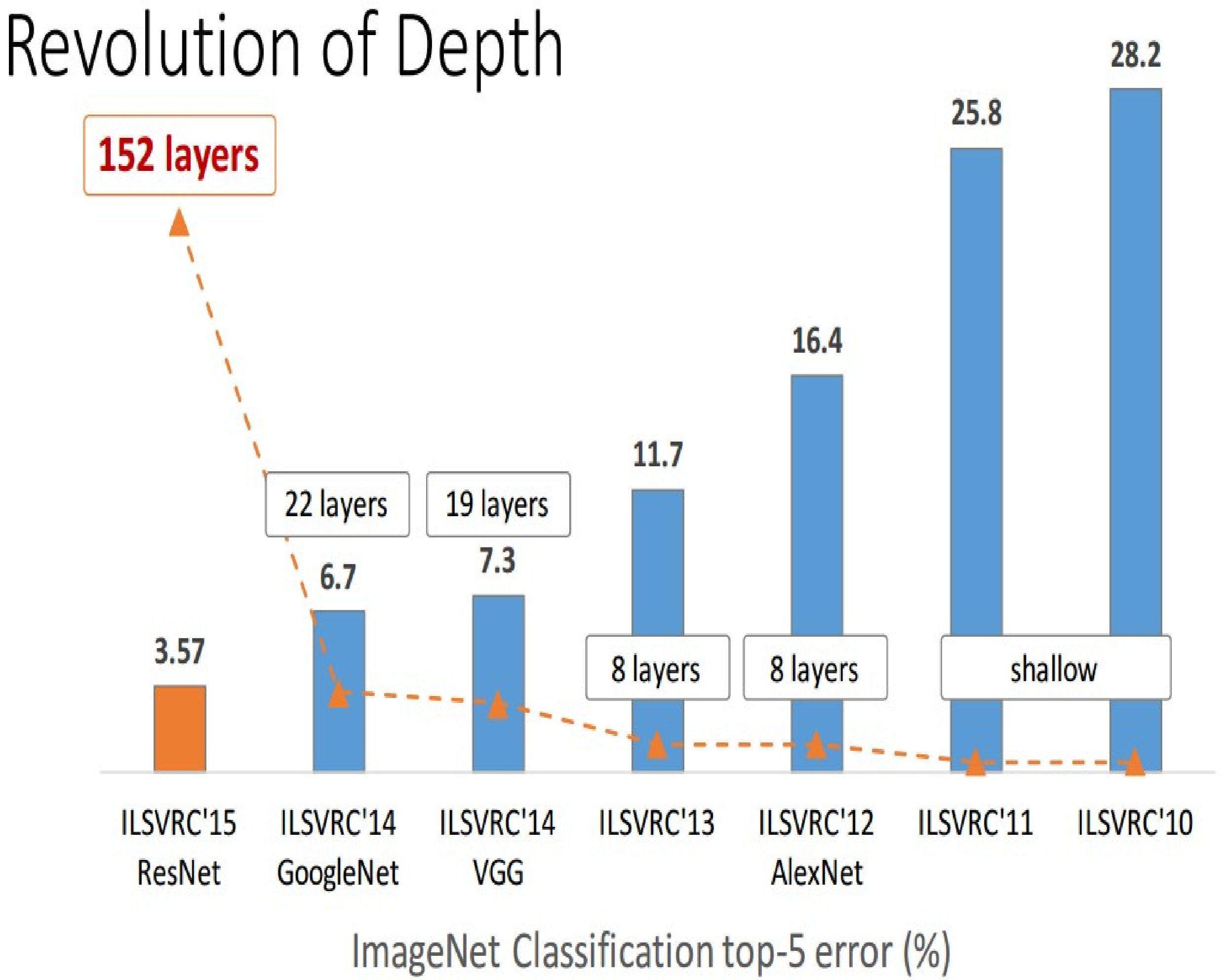
Residual networks

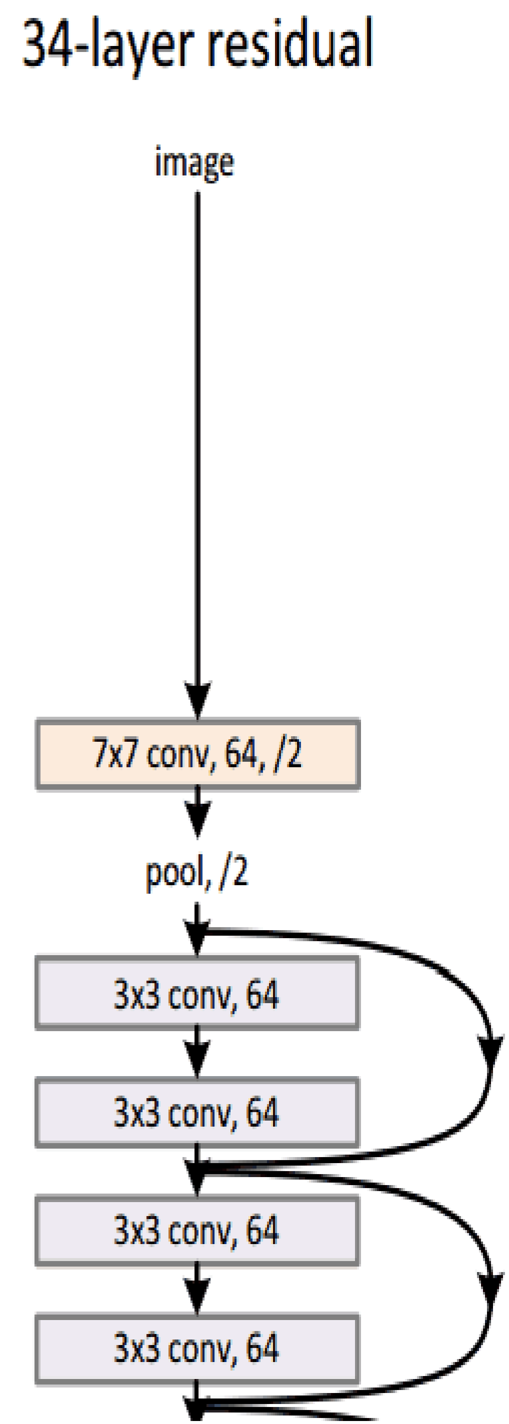
Results: Deeper residual networks result higher accuracy





Results with ResNets





Results with ResNets

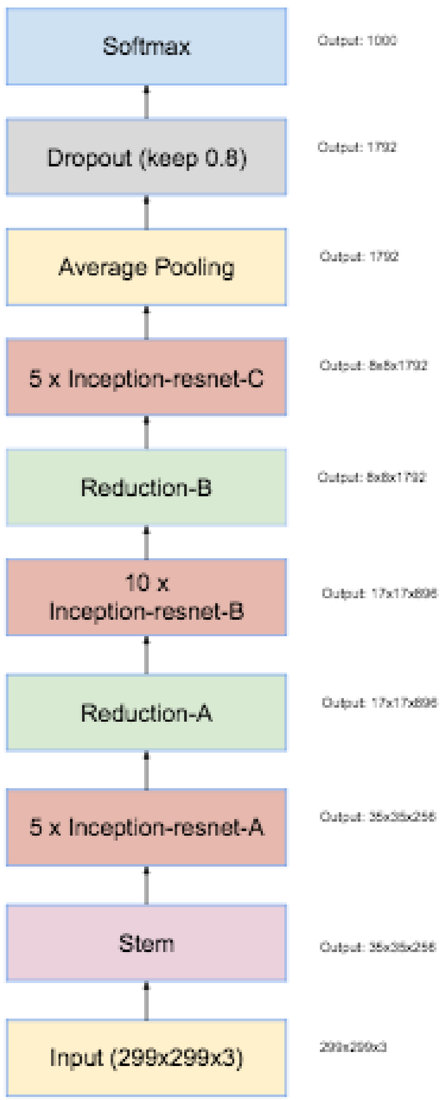
ResNets had the lowest error rate at most competitions since 2015

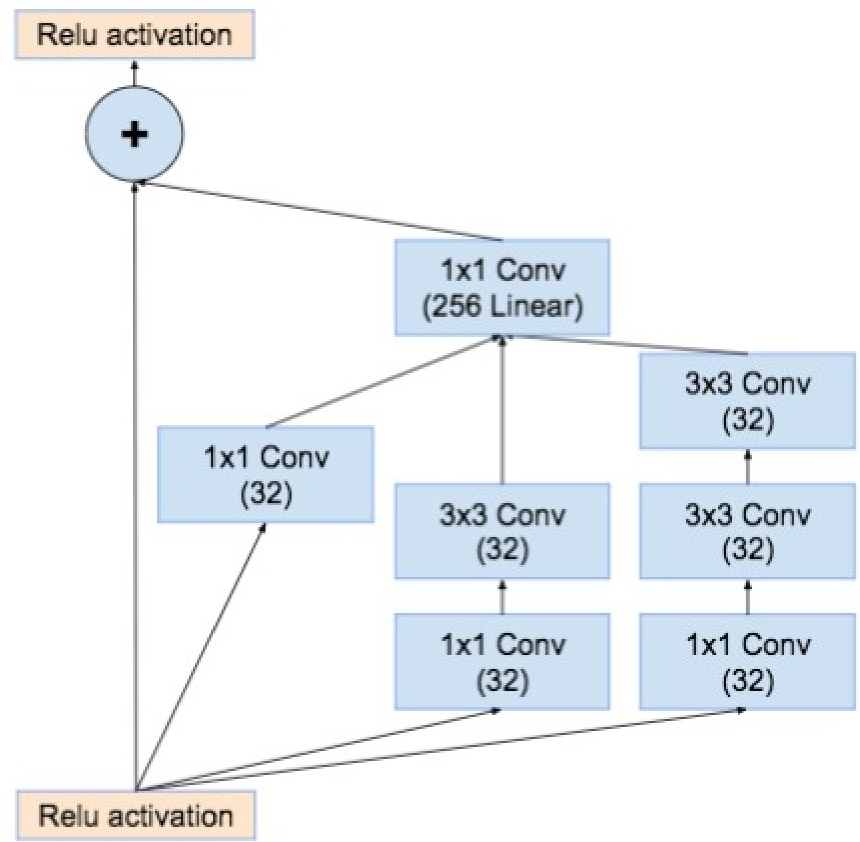
1st places in all five main tracks

- ImageNet Classification: “Ultra-deep” 152-layer nets
- ImageNet Detection: 16% better than2nd
- ImageNet Localization: 27% better than2nd
- COCO Detection: 11% better than2nd
- COCO Segmentation: 12% better than2nd

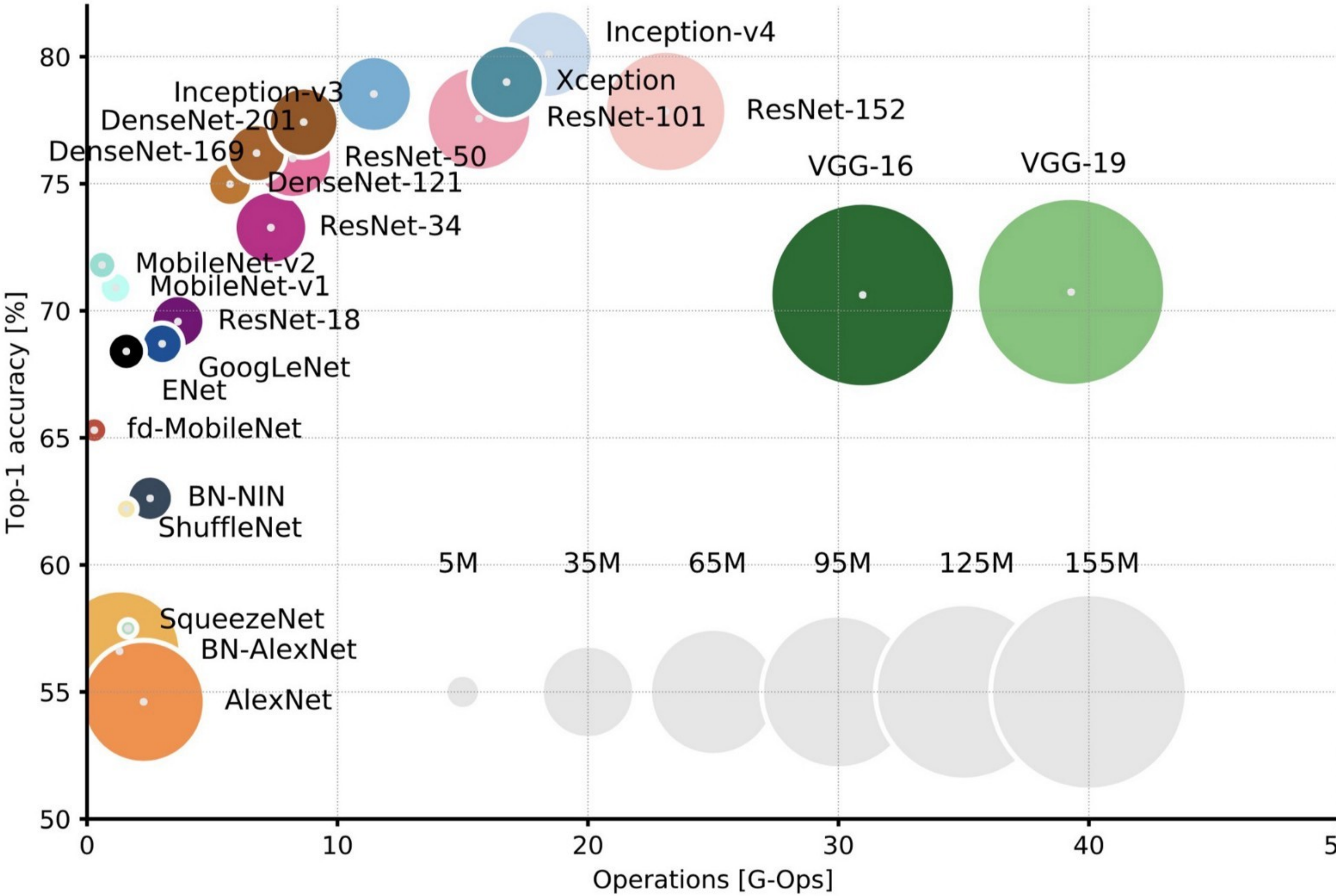
GoogleNet Inception v4

Inception architecture applied to residual networks





Efficiency of Neural Networks



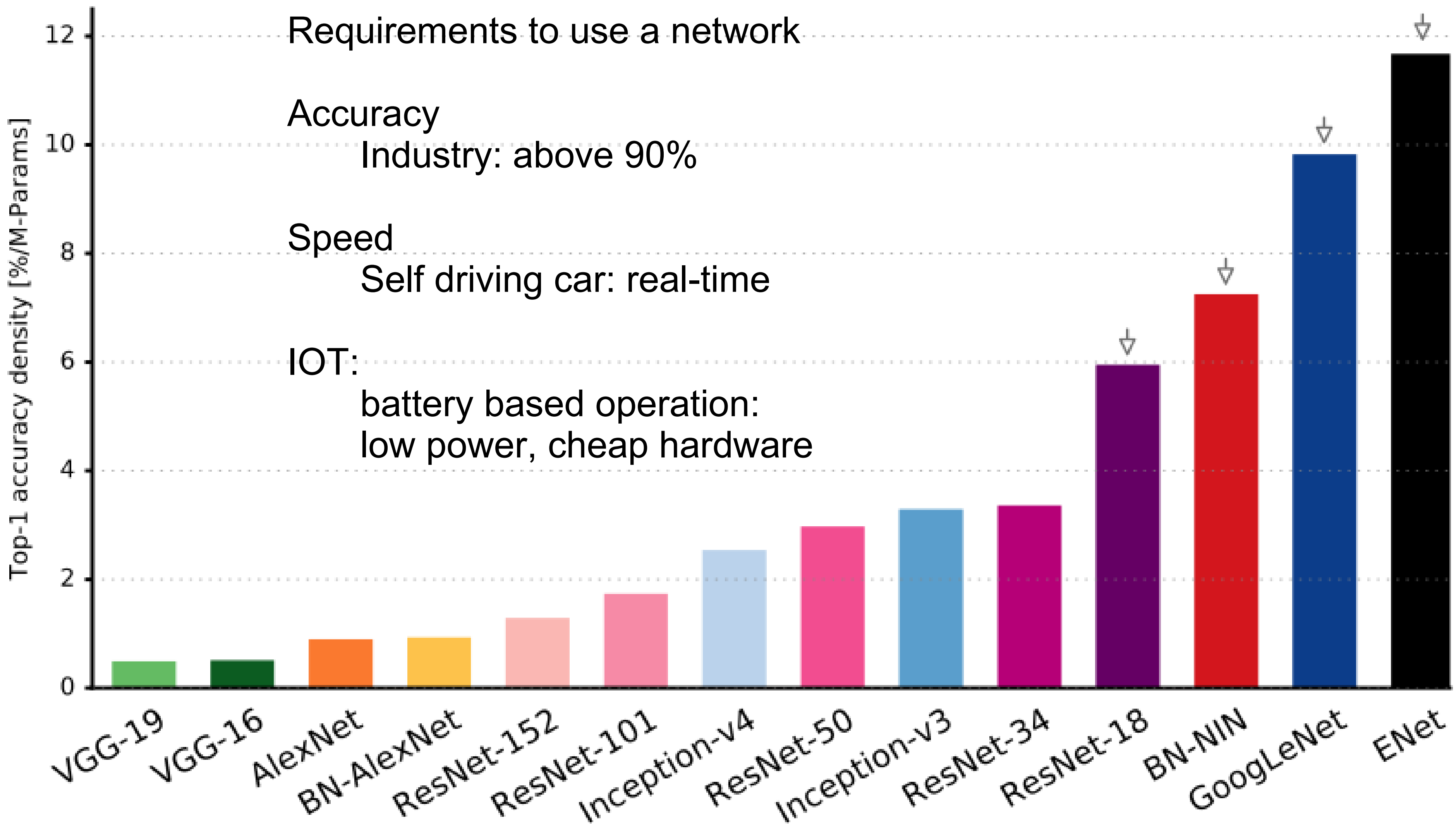
Efficiency of Neural Networks

Requirements to use a network

Accuracy
    Industry: above 90%

Speed
    Self driving car: real-time

IOT:
    battery based operation:
    low power, cheap hardware



MobileNet

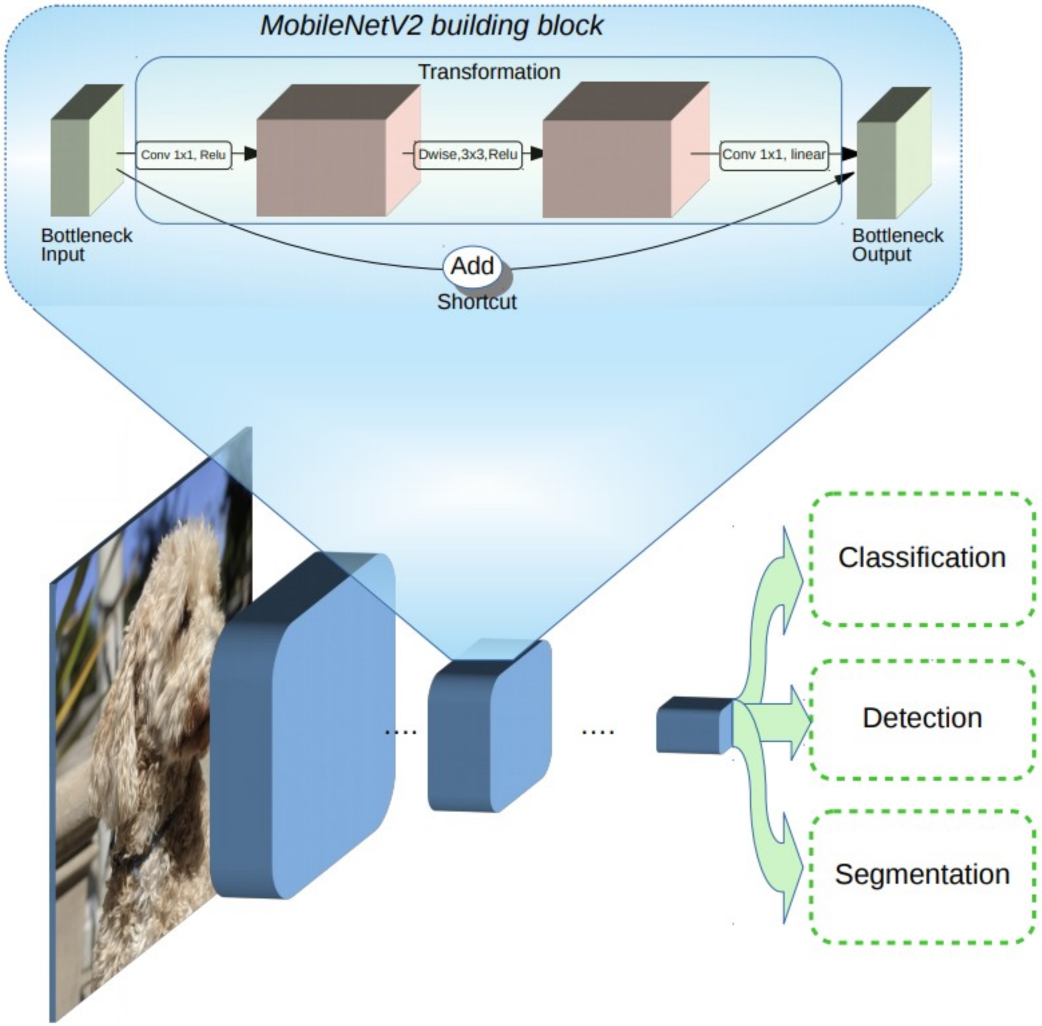
Scaling in feature map depths.

In this arhcitecture feature depths are squeezed before each operation

In a squeezed architecture we will use downscale the 128 feature maps to 16, using a linear combination (1x1 convolution)

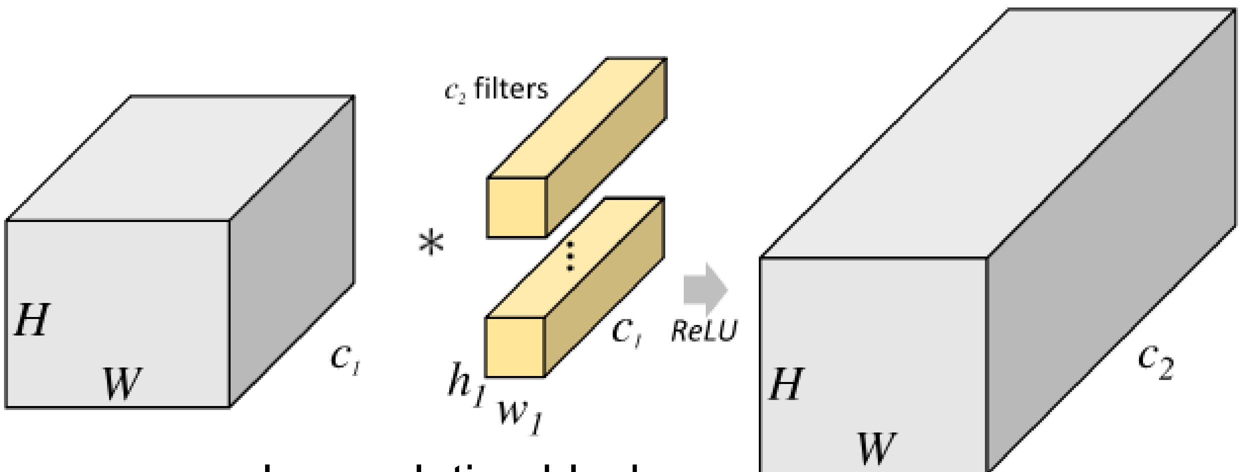
After the 3x3 covolutions, we expanded back to 128 layers by 1x1 convolution again

From the linear combination of these elements the new maps are created



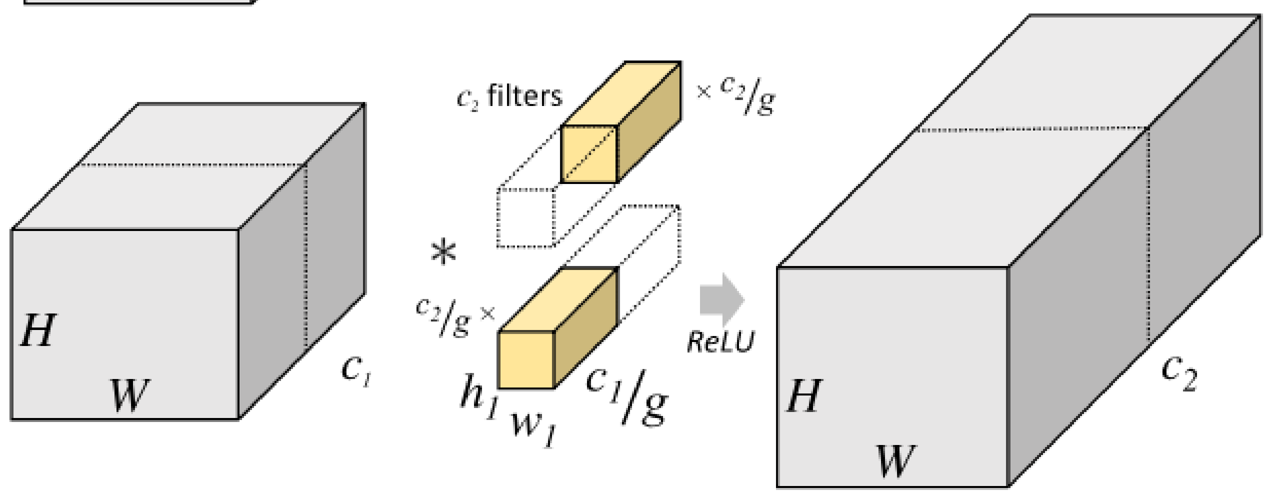
ResNext

- **Group convolution**:
  - Dividing the feature mapes into two groups, and apply the convolutions to each groups separately
  - The number of convolutions will be halved

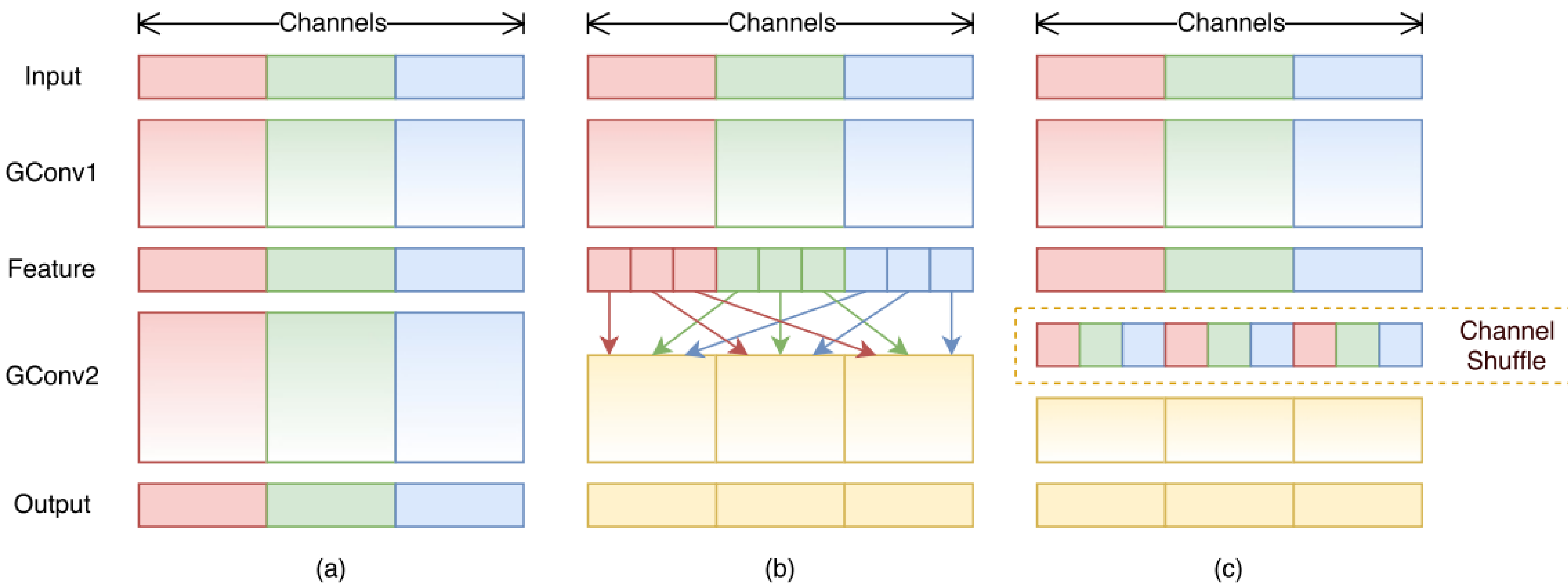


- normal convolution block:
-
- c_1 inputs, c_1 outputs
  - $c_1c_2$number of kernels

- group convolution block:
-
- $2x(c_1/2)$ inputs, $2x(c_2/2)$ output
  - $2x(c_1/2\ c_2/2\ ) = c_1c_2/2$ number of kernels



ShuffleNet

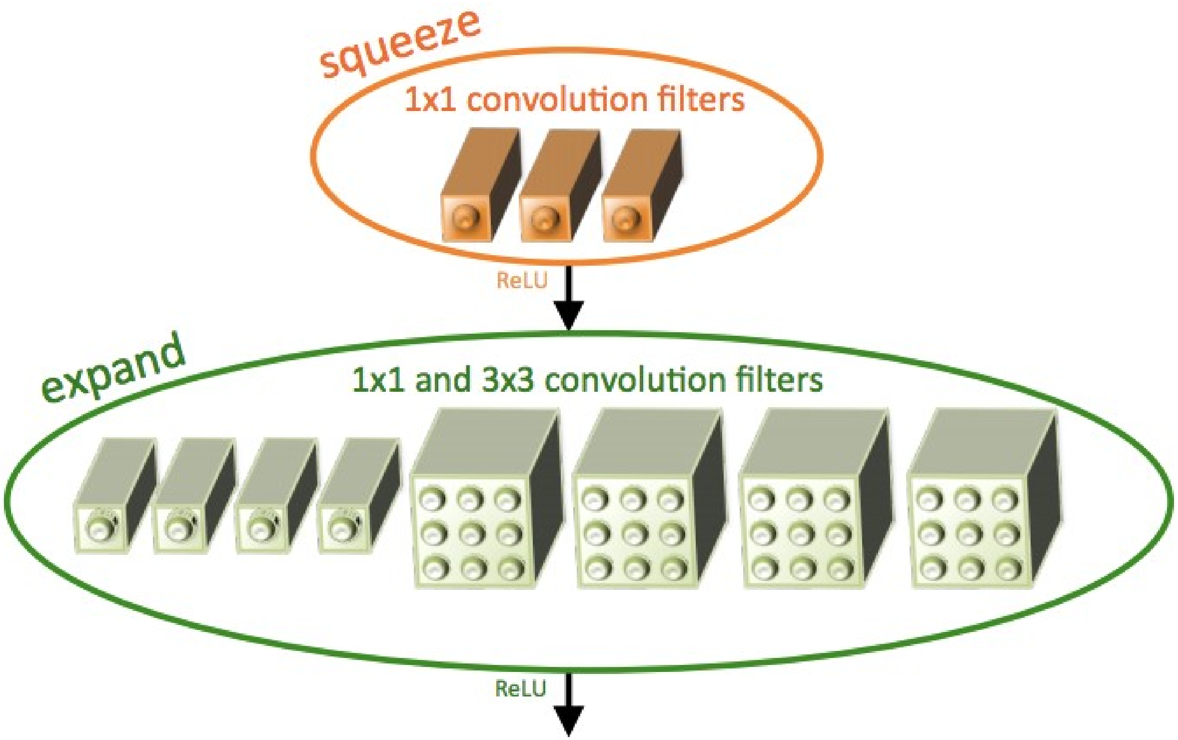

Channels
Channels
Channels
Input
GConv1
Feature
GConv2
Output
Channel Shuffle
(a)
(b)
(c)

SqueezeNet

In this arhcitecture depths are squeezed before each operation

The expand is done by the concatenation of the 1x1 and the 3x3 convolutions.

Advantage: the expand layer is saved.



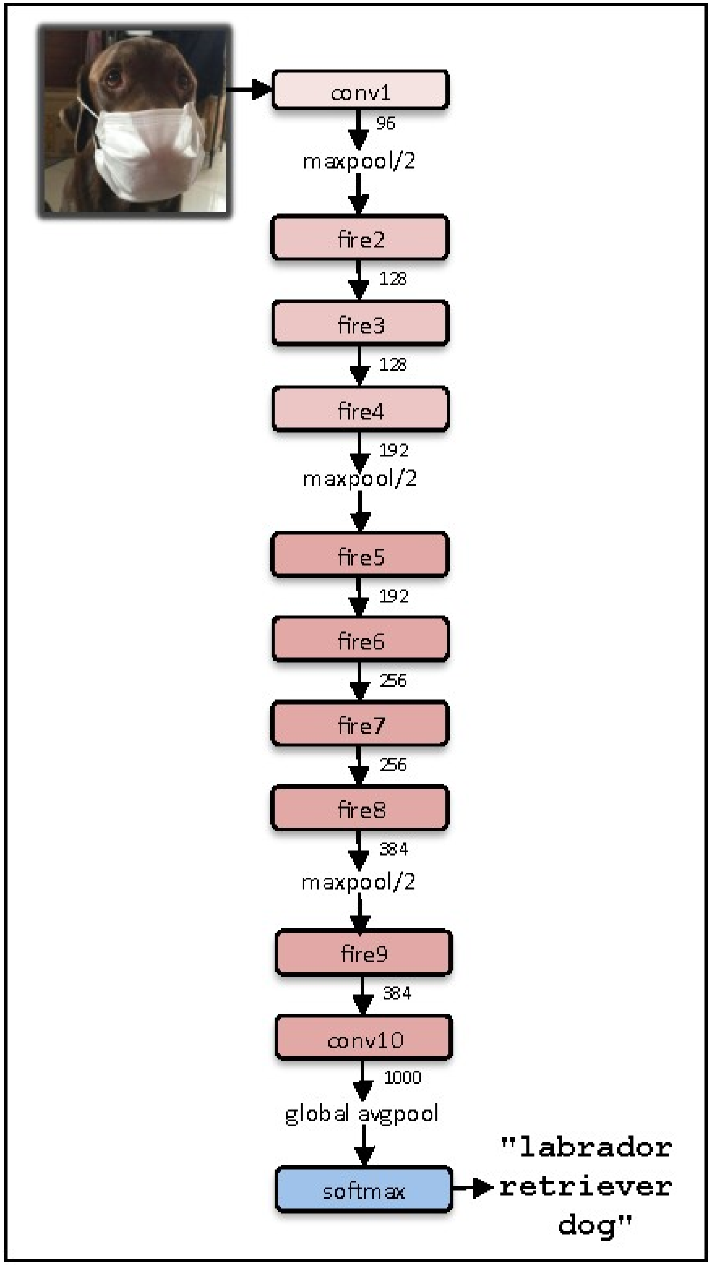
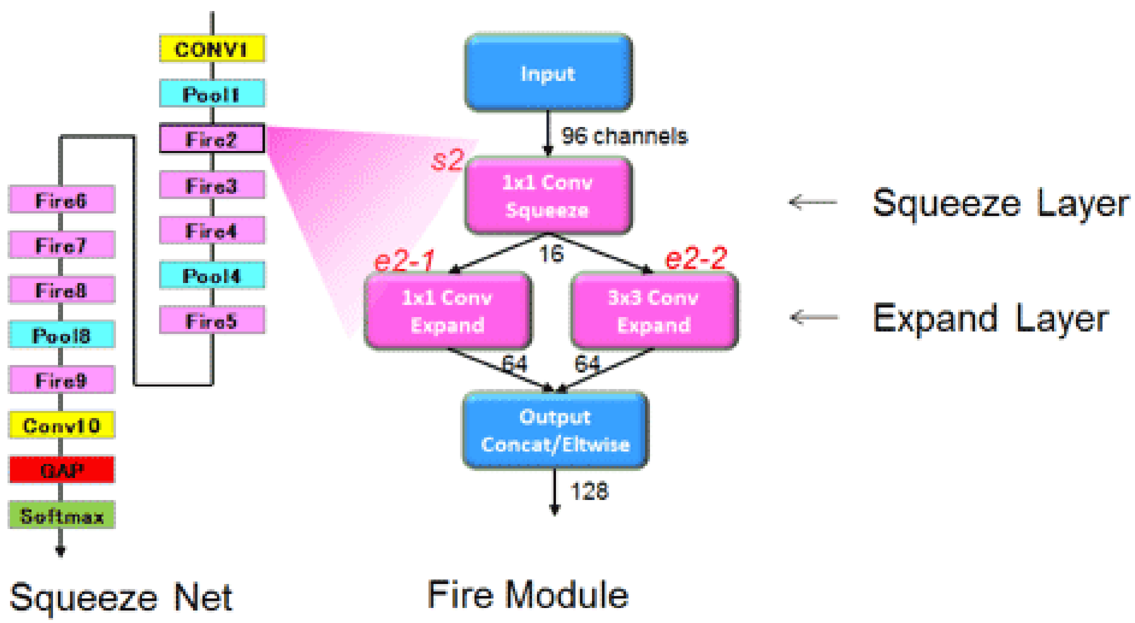


Figure 2. The SqueezeNet architecture

SqueezeNet

In this arhcitecture depths are squeezed before each operation



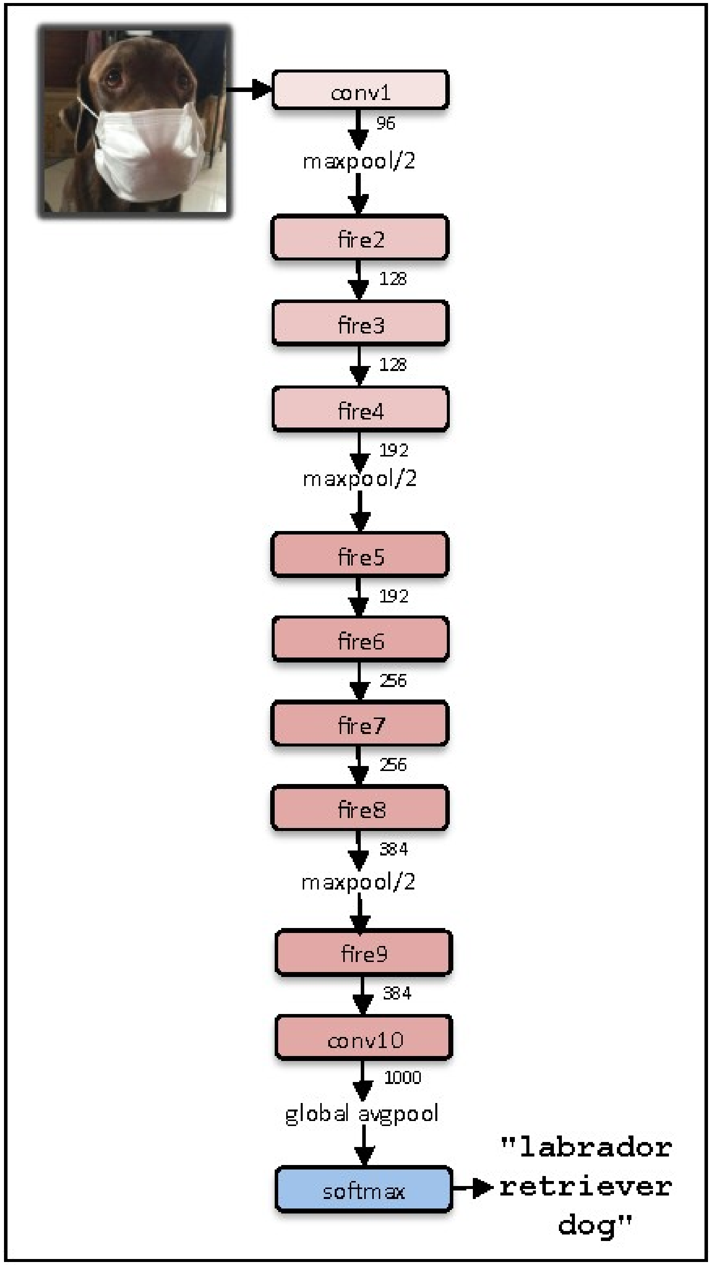


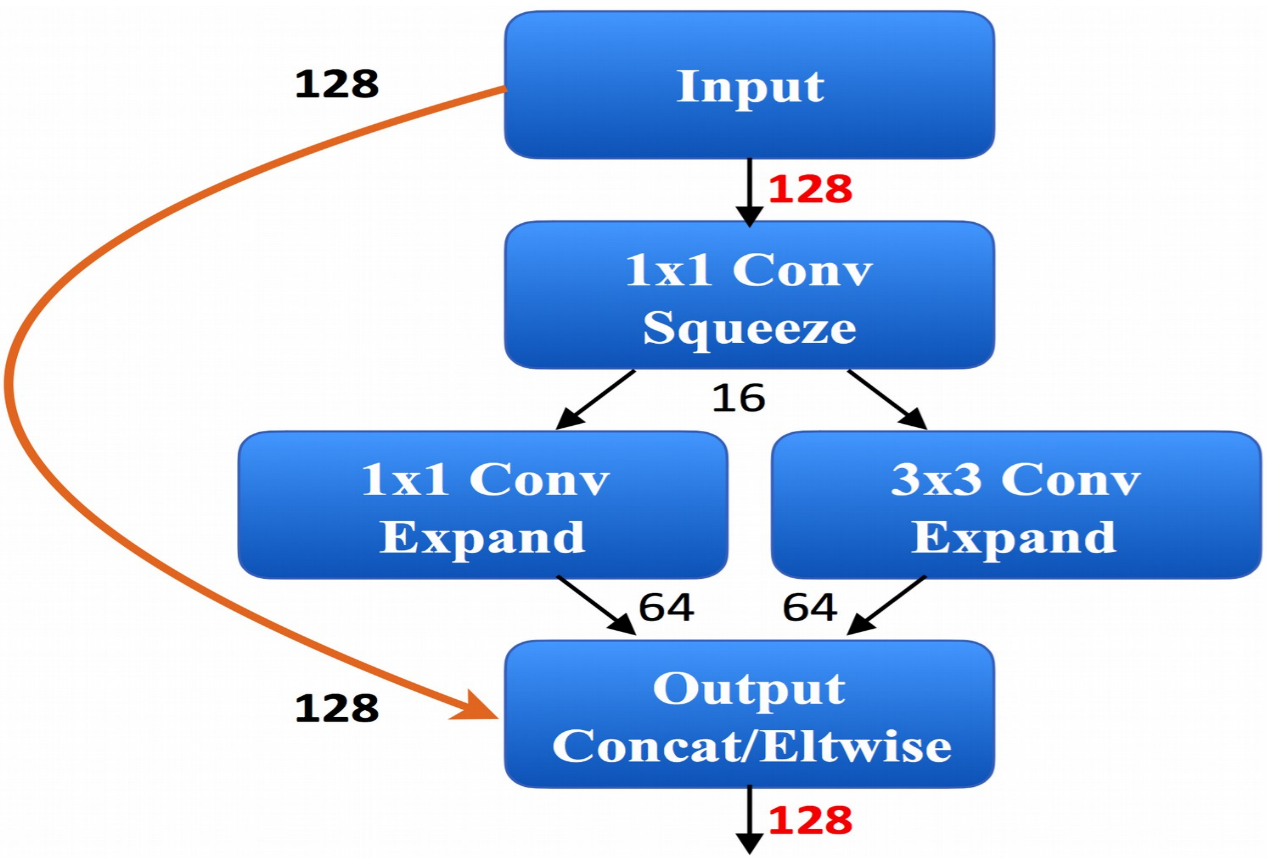
Figure 2. The SqueezeNet architecture

SqueezeNext

In this arhcitecture depths are squeezed before each operation

In a SqueezeNext architecture we will use a linear approximatine of 128 feature maps, using 16 independent feature maps

From the linear combination of these elements the new maps are created



Neural networks for regression

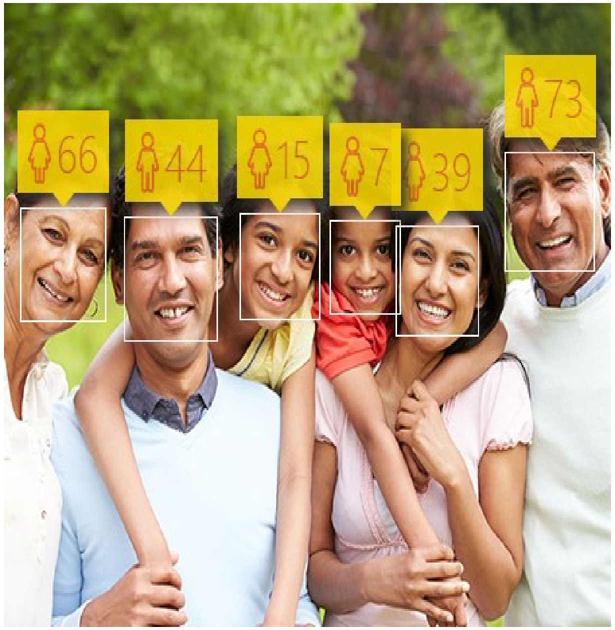
Age estimation

The output is not discreet classes or pixels, but continuous values

The network structure can remain the same but a different loss function and differently annotated dataset is needed.

Hard to interpret the error in common tasks.

E.G: Age estimation on images:

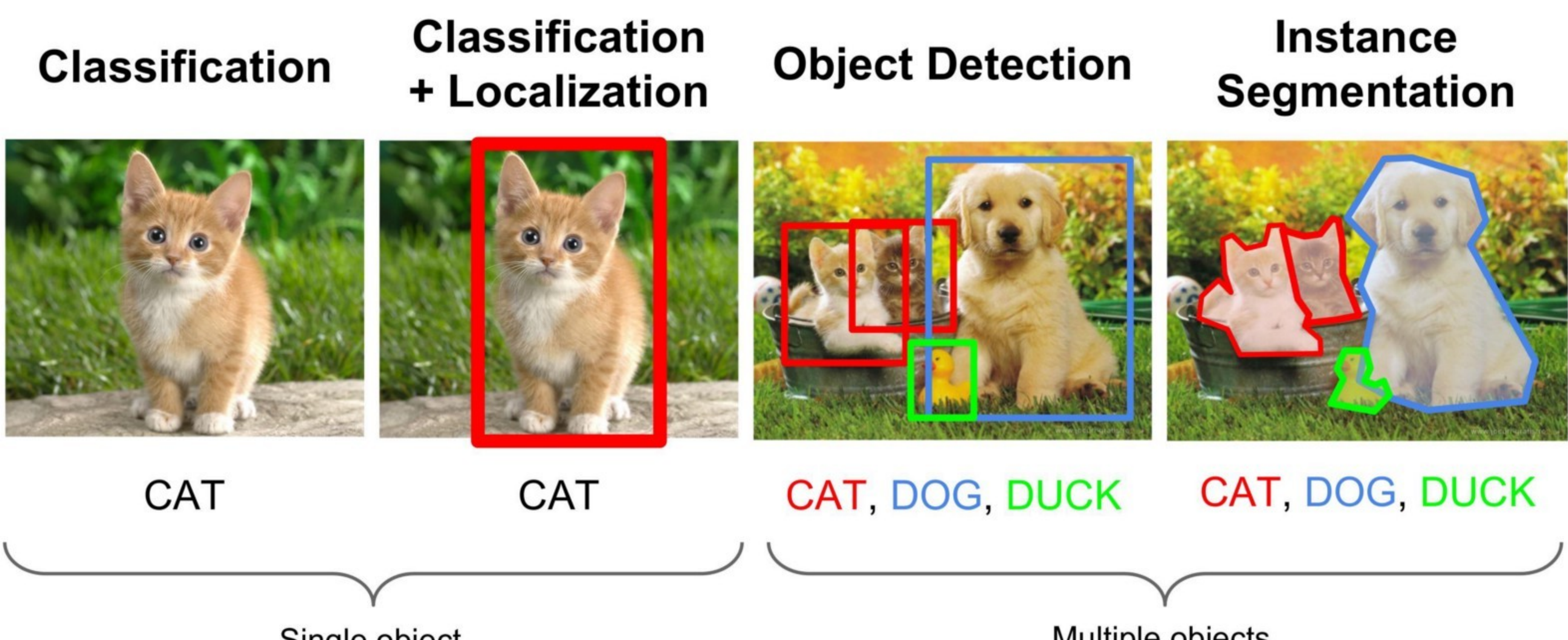


Neural networks for regression

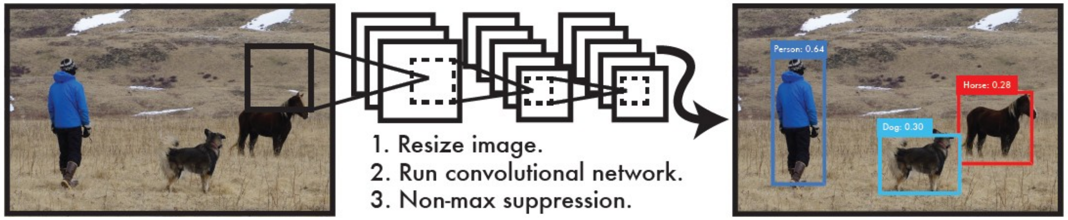
Multiple object detection on a single image

Classification is good for a single object (can be extended for k objects – top k candidates)

How could we detect objects in general, when the number of objects is unknow



Traditional method



Sliding window over the image

We might have objects in different scales

Slidign windowds in different scales, aspect ratios

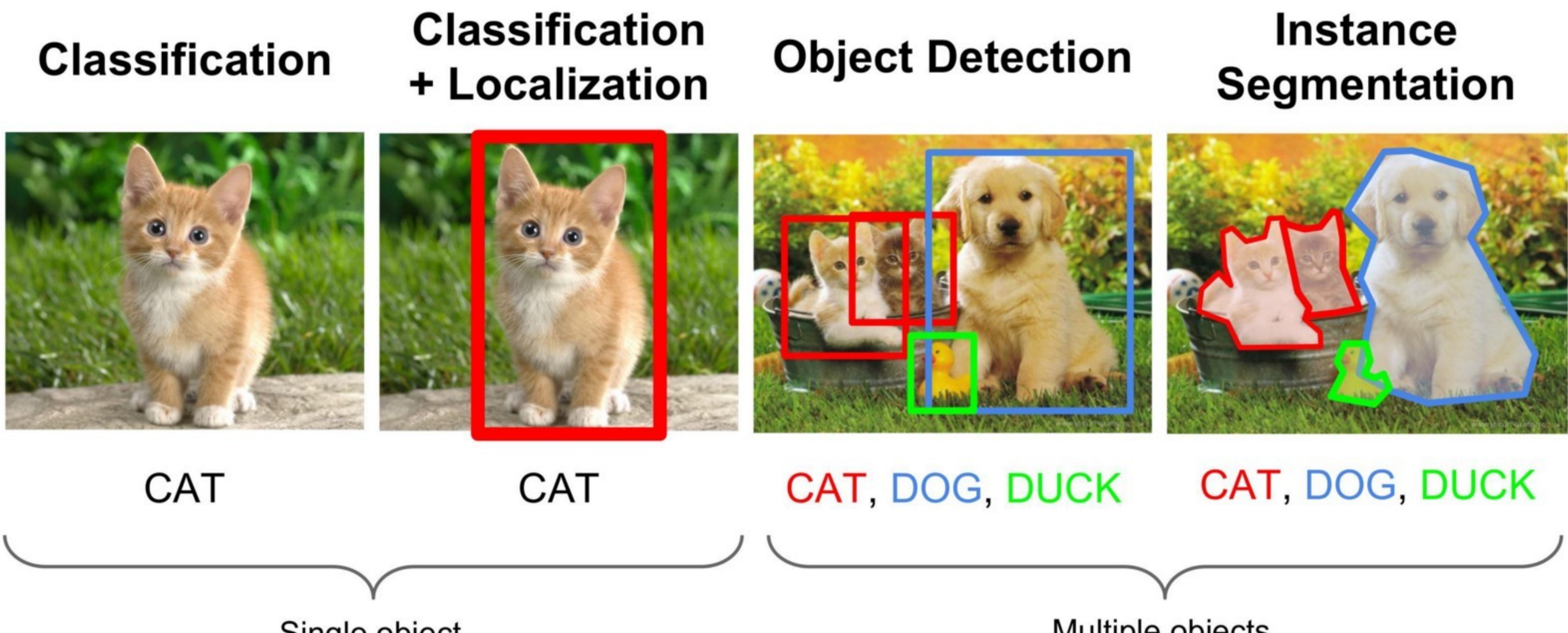
Resutls a heat map → detect the objects: non-maximum suppression

Object detection as regression

RCNN

Single Shot Object Detector (SSD) (2016 March)

You Only Look Once YOLO (2016 May)



R-CNN

Region proposal CNN network

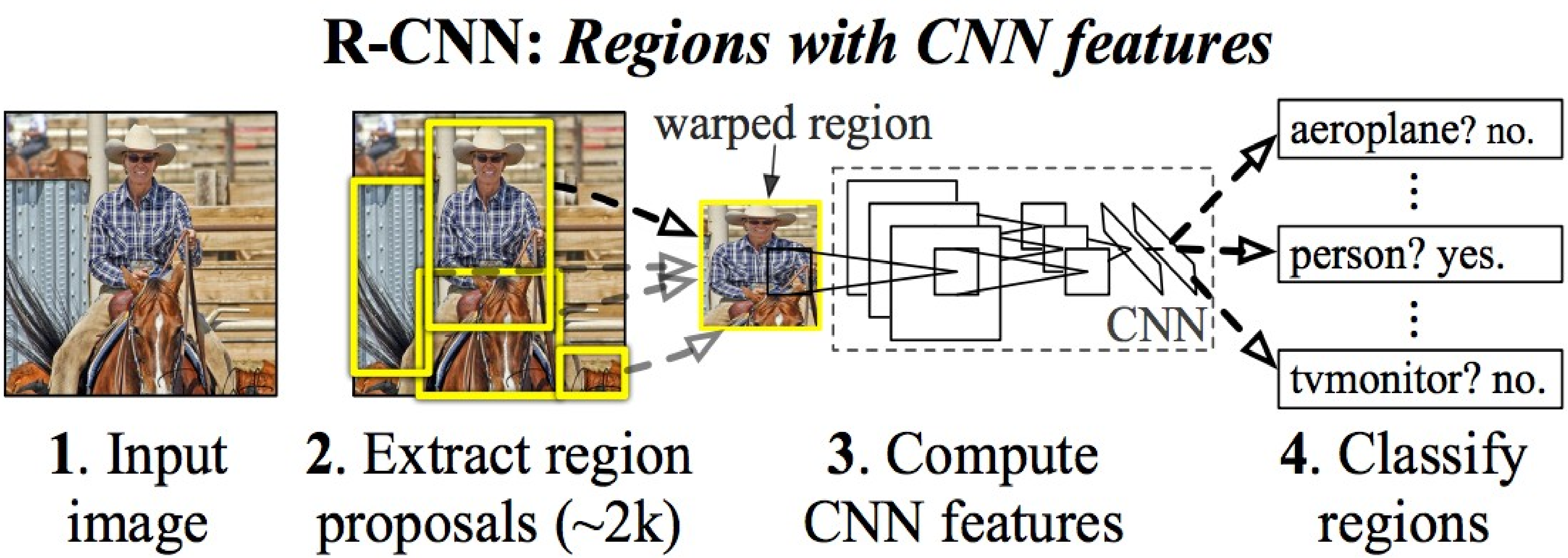
Separate the problem of object detection and calssification

It consists of three modules.

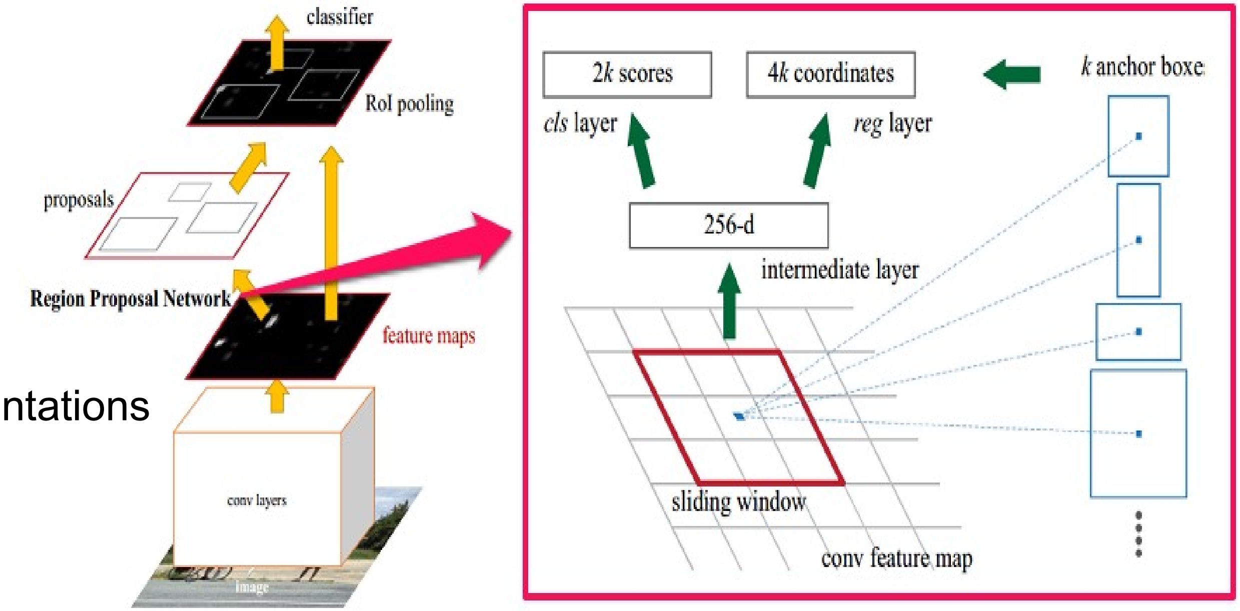
The first generates category-independent region proposals. These proposals define the set of candidate detection avail-able to detector.

· The second module is a large convolutional neural network that extracts a fixed-length feature vector from each region.

The third module is a set of class- specific linear SVMs



Faster R-CNN

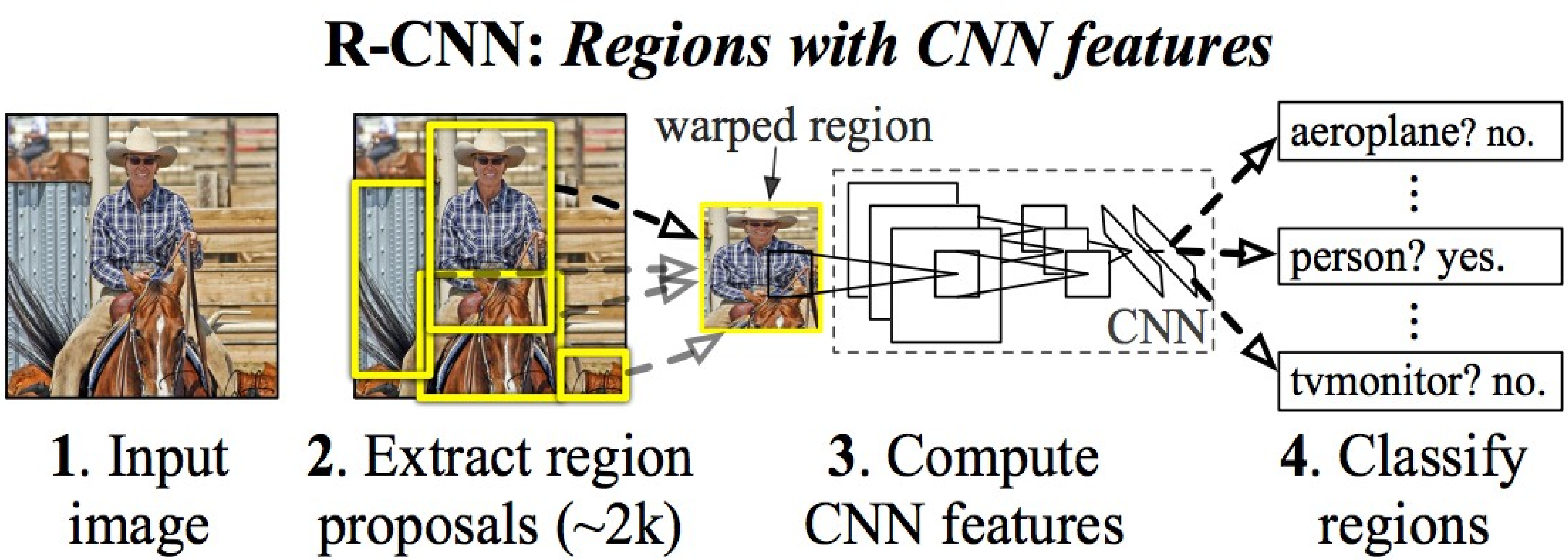


Region proposal from a network

Step 3 and 4 are standard CNN implementations

Extra layers for region proposals

Possible region refinement at the end



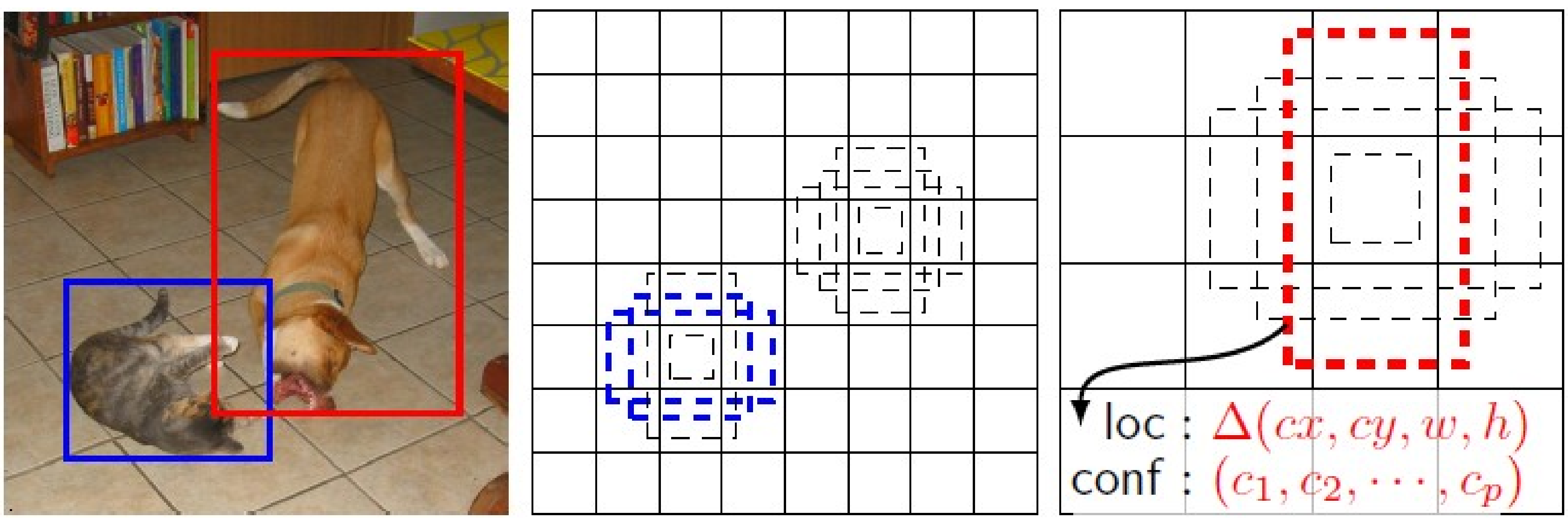
SSD

Single shot object detector SSD (2016 March)

Has a fixed resultion and the last feature maps (with different scales) can be considered as maps of bounding boxes

On these maps each pixel represent a fixed size bounding boxes. (Each feature map represents a certain box size.

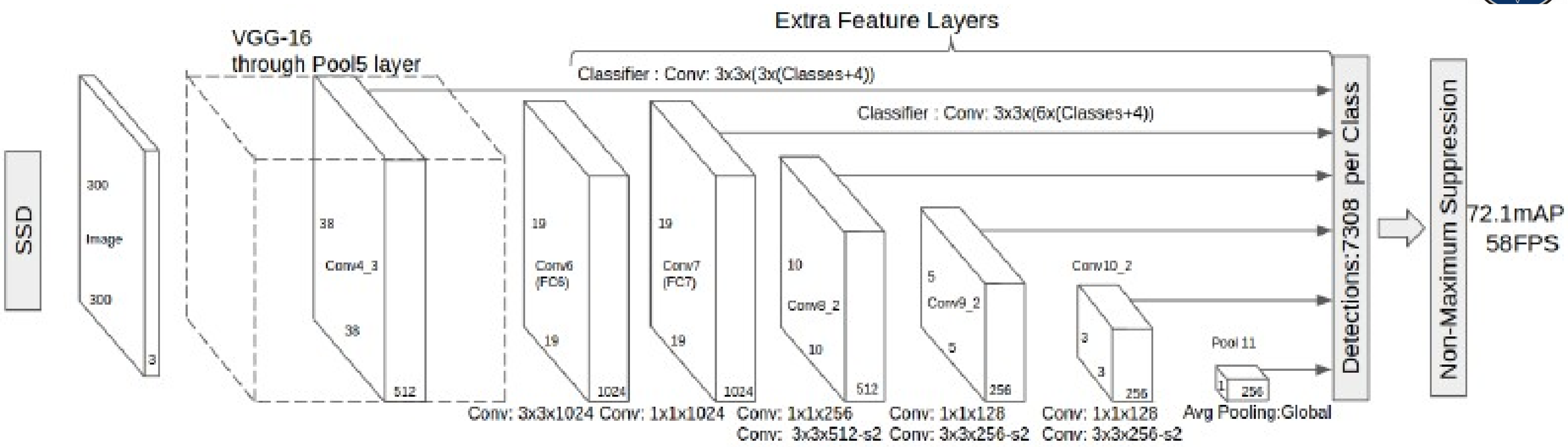
A high pixel value represent high probability of the centerpoint of a detected object.

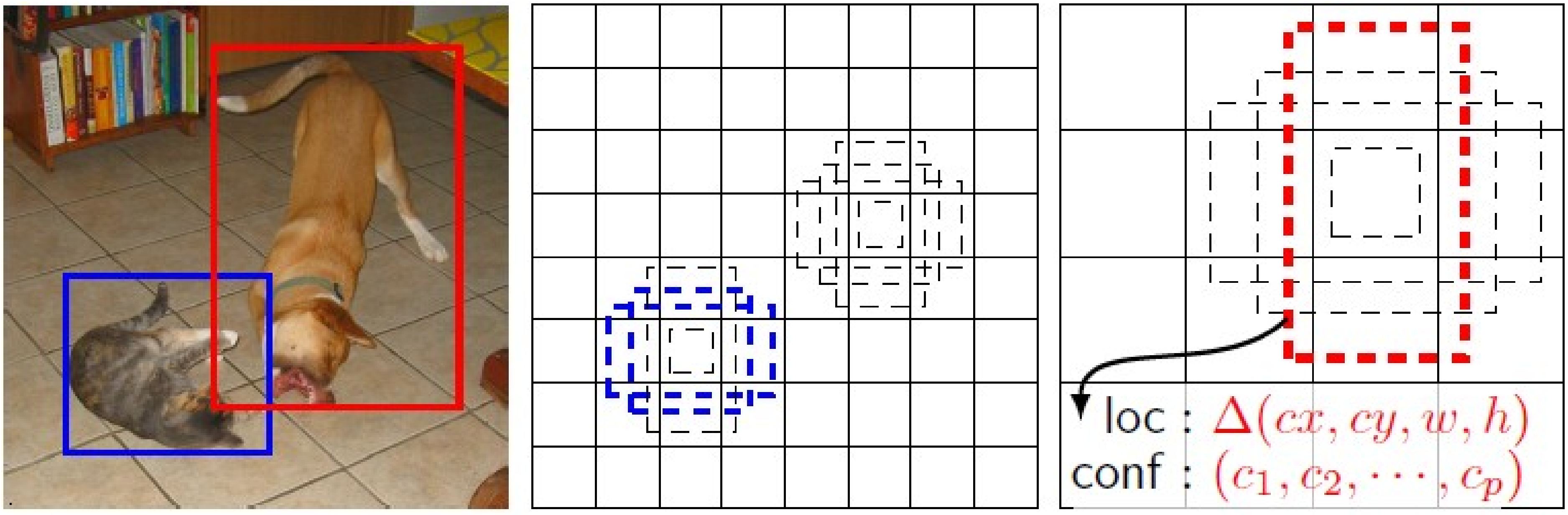


(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Problem: Unlike at R-CNN, the boundix boxes have fixed scale and positions, no fine turning in the last step.

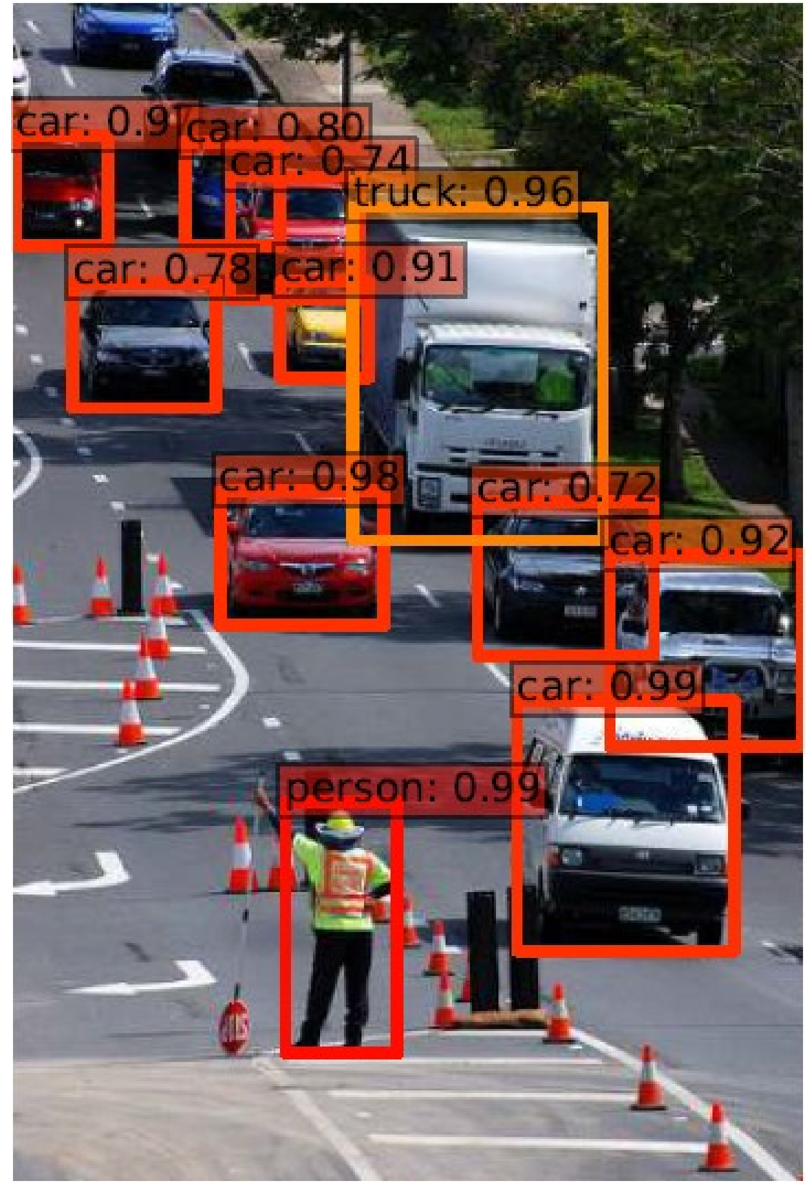
SSD arhcitecture





(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map


truck: 0.60
person: 0.83
person: 0.86
dog: 1.00


car: 0.80
car: 0.74
truck: 0.96
car: 0.78
car: 0.91
car: 0.98
car: 0.72
car: 0.92
car: 0.99
person: 0.99

YOLO, Detectnet

Models detection as a regression problem:

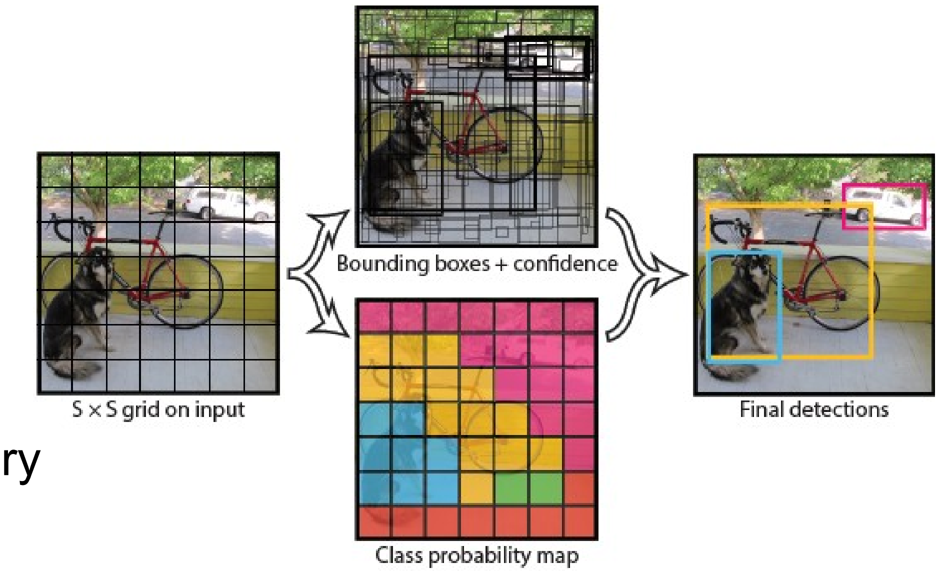
Divide the image into a grid and each cell can vote for the bounding box position of possible object. (Four output per cell for the corner positions.)

Boxes can have arbitrary sizes

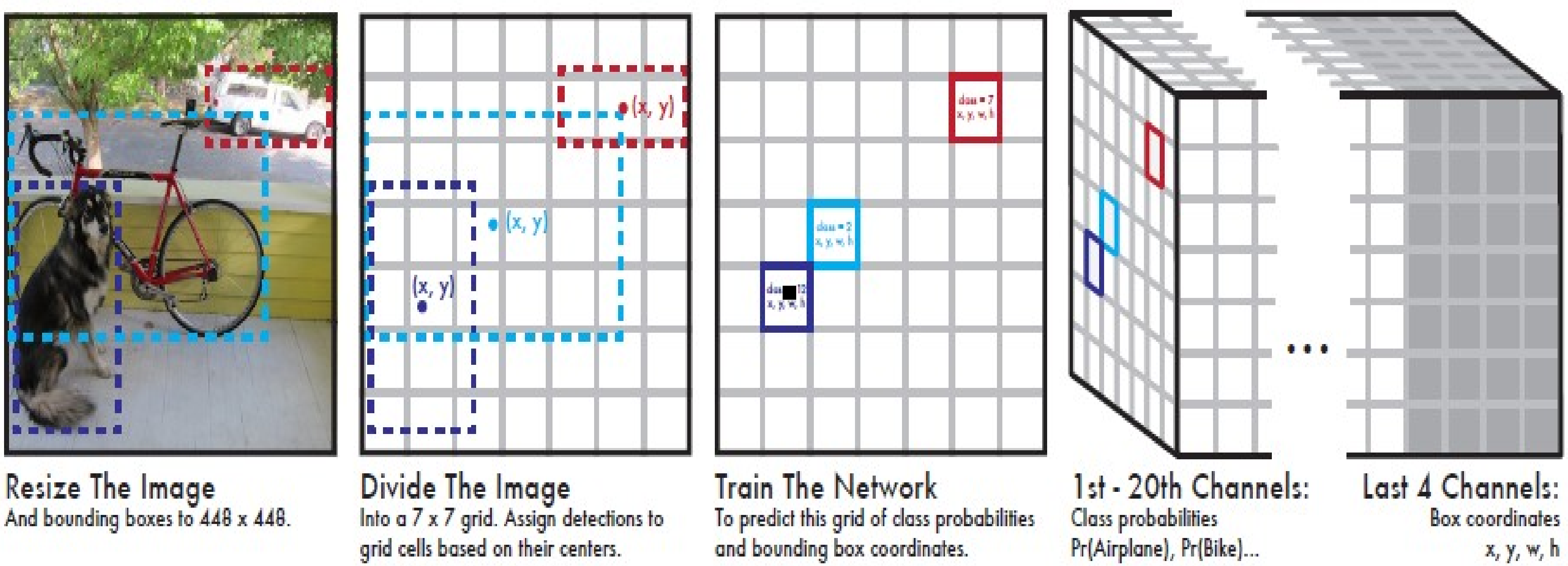
Each cell can proposes a bounding box one category (more layers, more categories per position).

Non-suppression on the boxes

No need for scale search, the image is processed once and objects in different scales can be detected

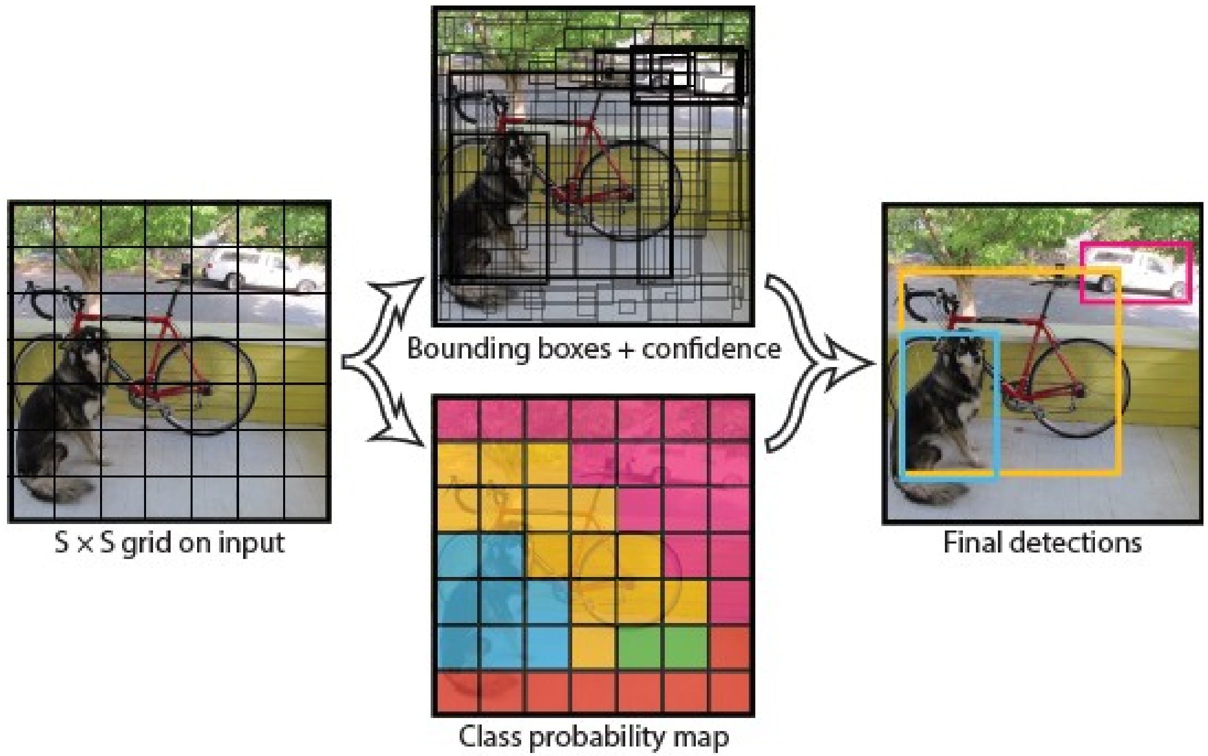


Handles oclusion



Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.

How unified detection works?



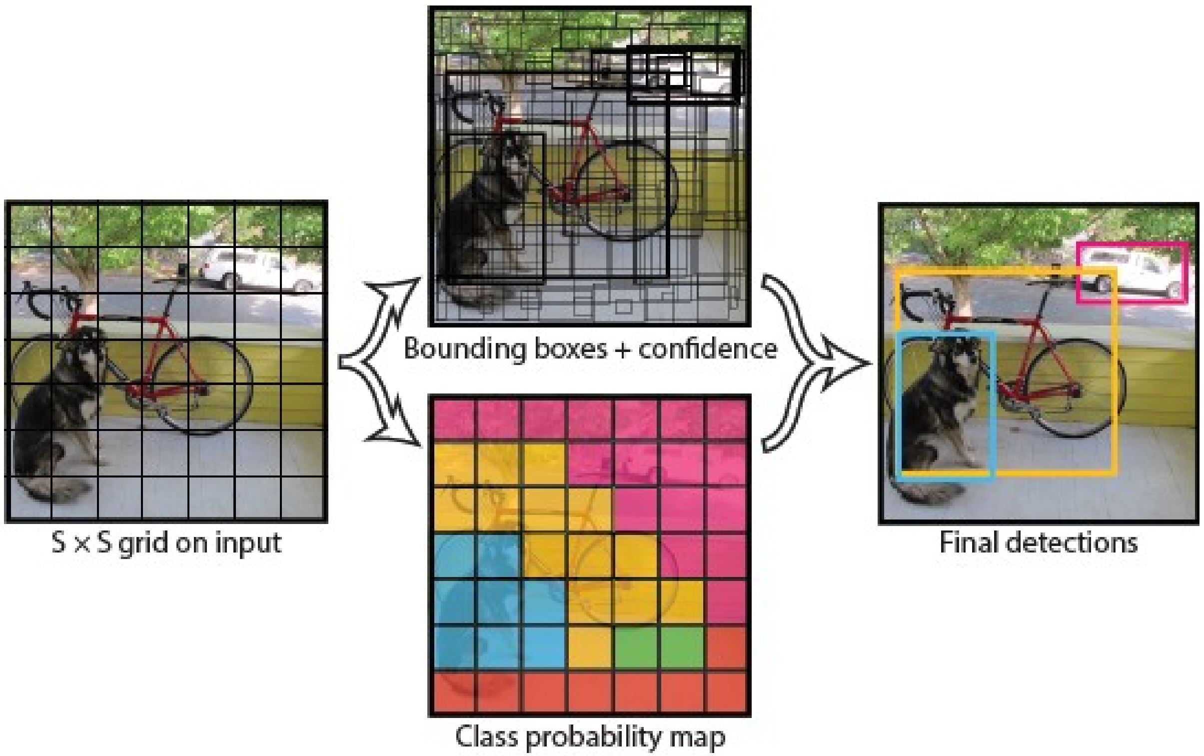
<u>confidence scores</u>: reflect how confident is that the box contains an object+how accurate the box is .

$$\Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

conditional class probabilities: conditioned on the grid cell containing an object

$$\Pr(\text{Class}_i|\text{Object})$$

How unified detection works?



$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

- At test time, multiply the conditional class probabilities and the individual box confidence predictions
- giving class-specific confidence scores for each box
- Showing both the probability of that class appearing in the box and how well the predicted box fits the object

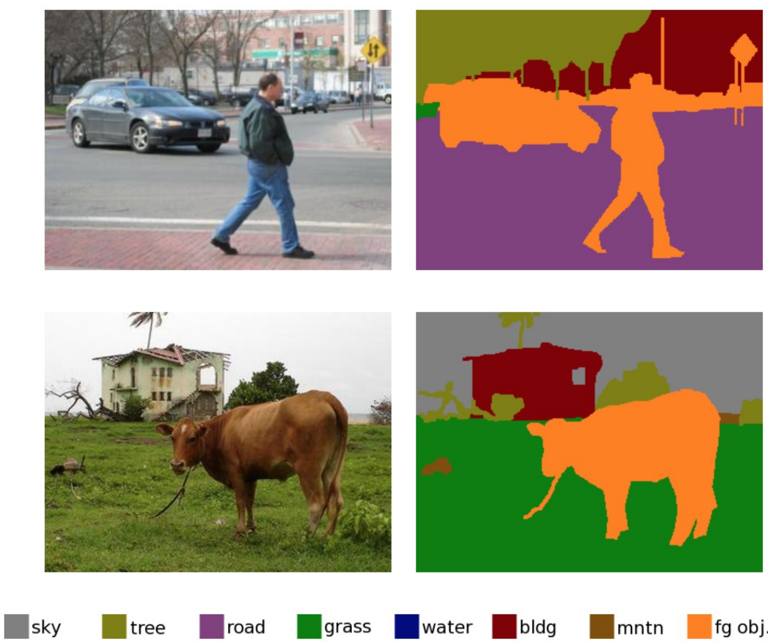
Pixel level segmentation

The expected output of the network is not a class, but a map representing the pixels belonging to a certain class.

Creation of a labeled dataset (handmade pixel level mask) is a tedious task

More complex architectures are needed (compared to classification)

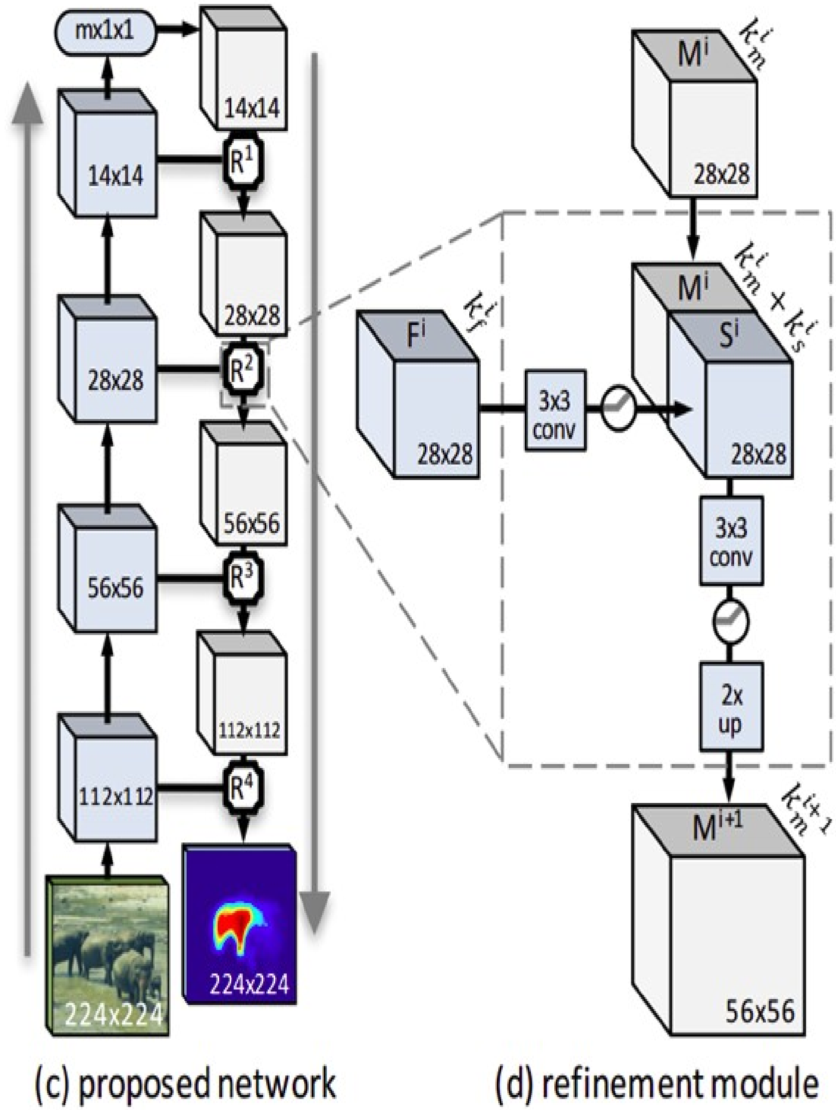
Popular architectures (Sharpmask, U-NET ...)



SharpMask: Learning to Refine Object Segments. Pedro O. Pinheiro, Tsung-Yi Lin, Ronan Collobert, Piotr Dollàr (ECCV 2016)

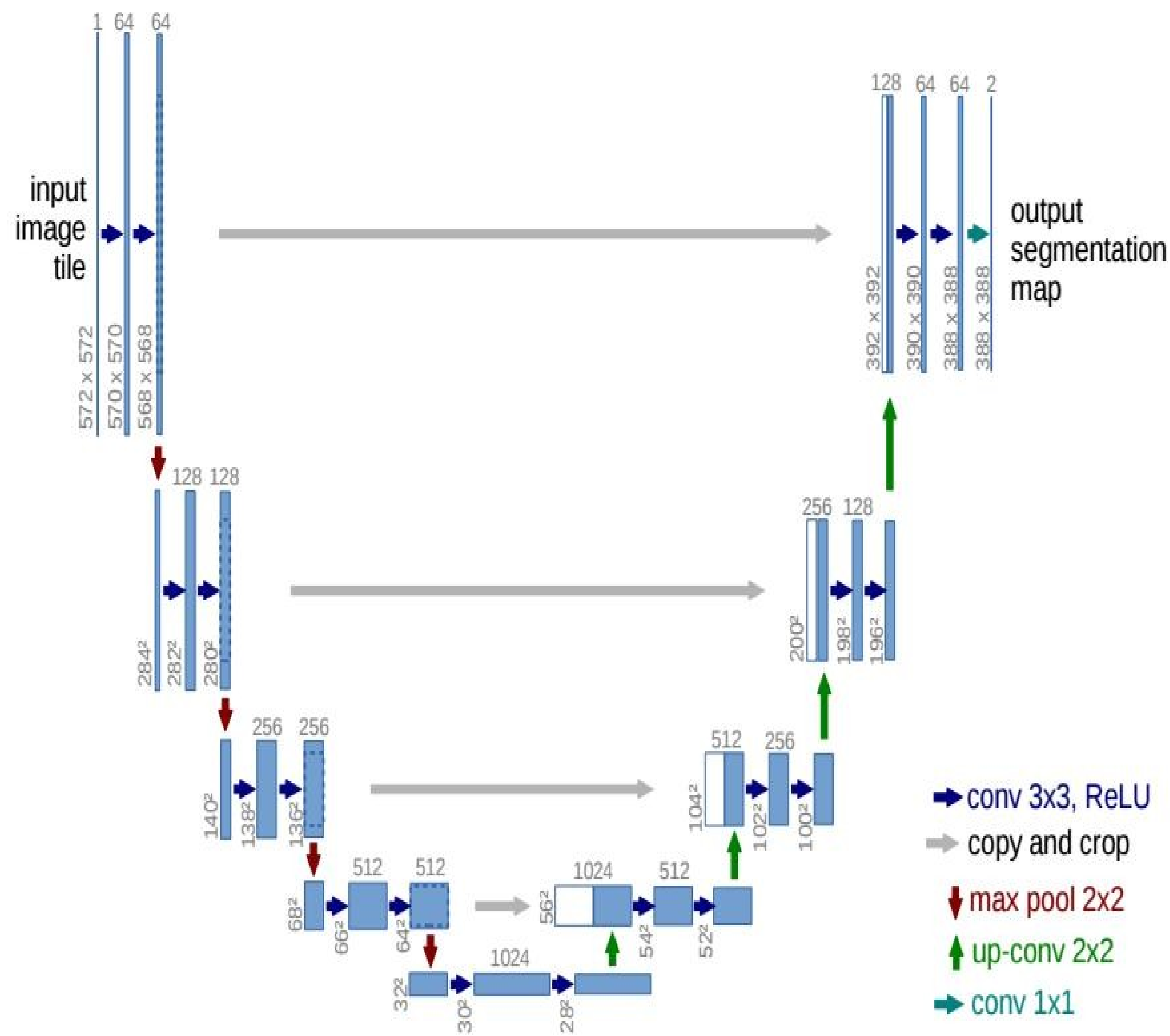
SEMANTIC IMAGE SEGMENTATION WITH DEEP CONVOLUTIONAL NETS AND FULLY CONNECTED CRFS Liang-Chieh Chen et al. ICLR 2015

Sharpmask



SharpMask network architecture

U-net

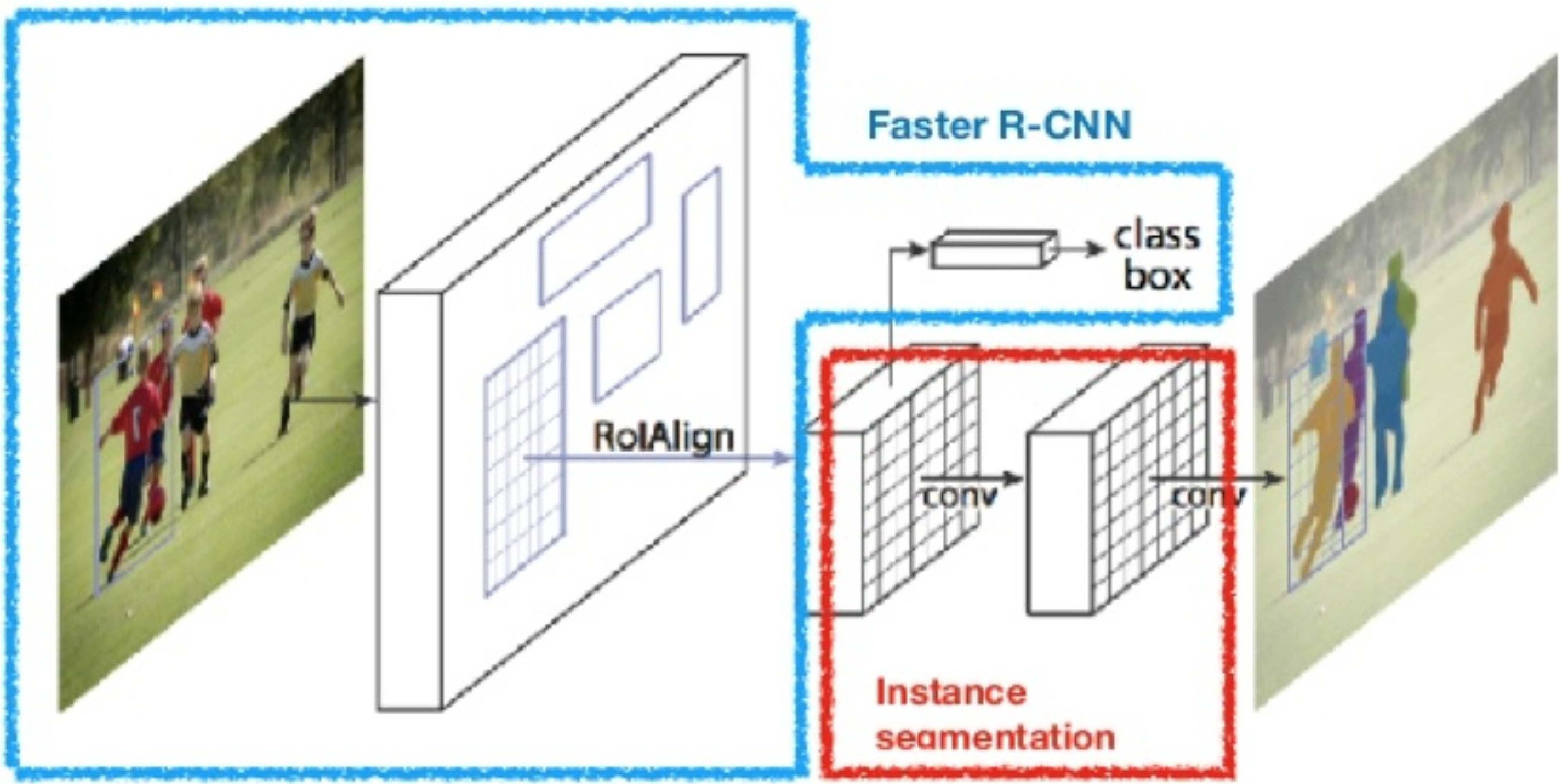


1 64 64
input
image
tile
572 x 572
570 x 570
568 x 568
128 128
284²
282²
280²
256 256
140²
138²
136²
512 512
68²
66²
64²
1024
32²
30²
28²
1024 512
56²
54²
52²
512 256
104²
102²
100²
256 128
200²
198²
196²
128 64 64 2
output
segmentation
map
392 x 392
390 x 390
388 x 388
388 x 388
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

Mask RCNN, RetinaNet

These networks generate bounding boxes and sematnic segmentation maps simultanously

They can be trained on images having lables for only one or both types of output



Mask RCNN, RetinaNet

These networks generate bounding boxes and sematnic segmentation maps simultanously

They can be trained on images having lables for only one or both types of output

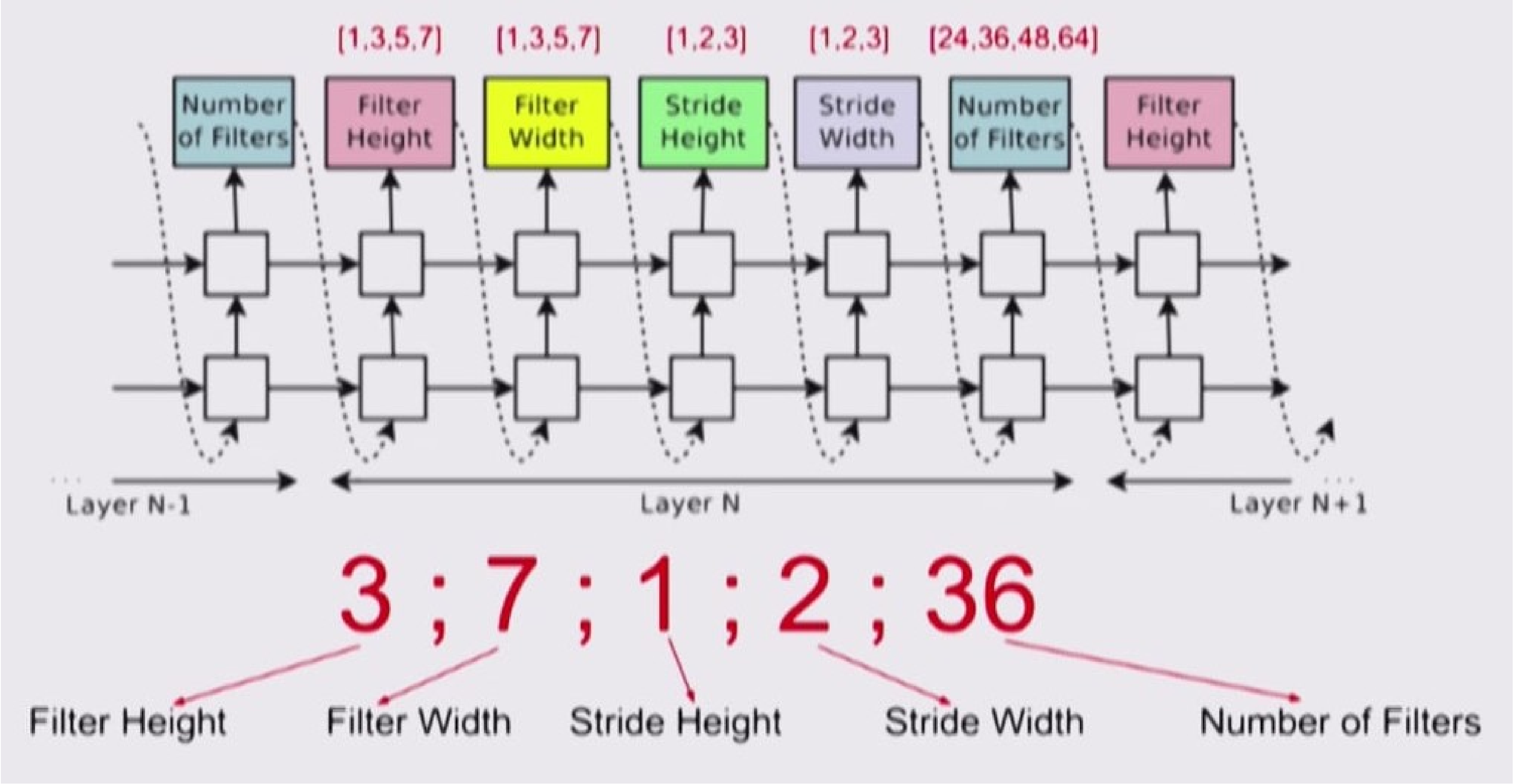
Starting from scratch
(if you do not want to use one of the famous networks)

Neural architecture search:

Networks can be described as a series of operations

As series of words → text

We can feed a Recurrent network with this data series



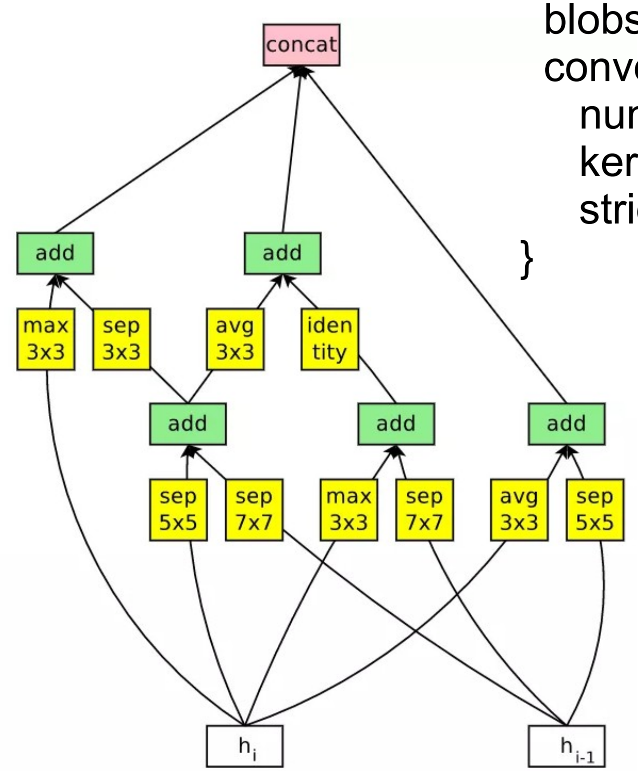
Starting from scratch

Neural architecture search:
Networks can be described as a series of operations
As series of words → text

The parameters of each layer can be described as numbers The input(s)/outputs(s) of the layer can be Ids

The whole network can be described as a graph

```
layers {
  bottom: "conv1"
  top: "conv1"
  name: "relu0"
  type: RELU
}
layers {
  bottom: "conv1"
  top: "cccp1"
  name: "cccp1"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  convolution_param {
    num_output: 96
    kernel_size: 1
    stride: 1
}
```



Starting from scratch

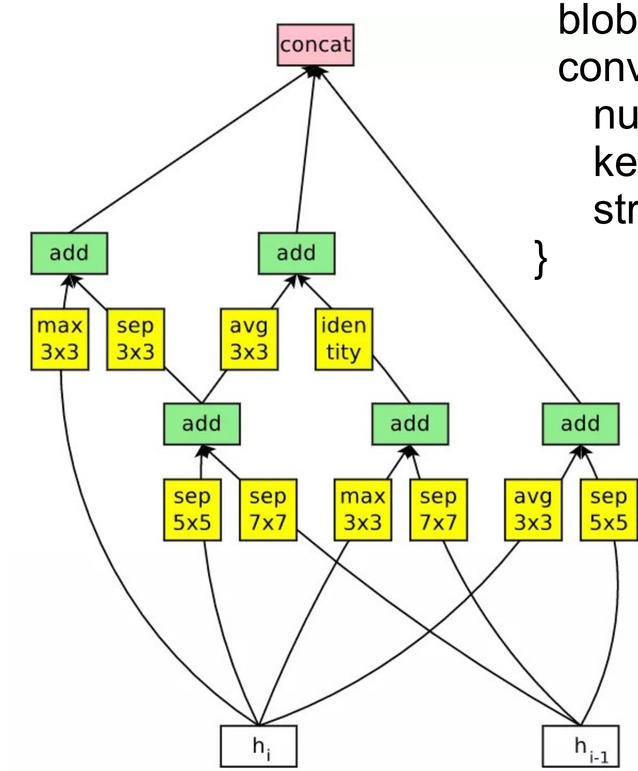
Neural architecture search:
Networks can be described as a series of operations
As series of words → text

The parameters of eahc layer can be described as numbers The input(s)/outputs(s) of the layer can be Ids

The whole network can be described as a graph

We have a problem space where we have text as an input and an accuracy number as an output

```
layers {
  bottom: "conv1"
  top: "conv1"
  name: "relu0"
  type: RELU
}
layers {
  bottom: "conv1"
  top: "cccp1"
  name: "cccp1"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  convolution_param {
    num_output: 96
    kernel_size: 1
    stride: 1
}
```



Starting from scratch

Neural architecture search:
Networks can be described as a series of operations
As series of words → text

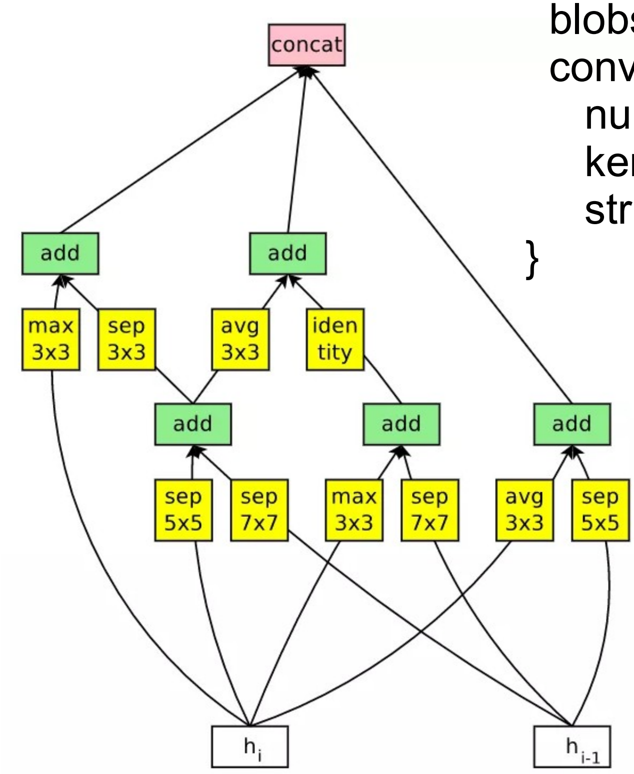
The parameters of eahc layer can be described as numbers The input(s)/outputs(s) of the layer can be Ids

The whole network can be described as a graph

We have a problem space where we have text as an input and an accuracy number as an output

We can train an RNN for regression, which approximates the accuracy of a given network

```
layers {
  bottom: "conv1"
  top: "conv1"
  name: "relu0"
  type: RELU
}
layers {
  bottom: "conv1"
  top: "cccp1"
  name: "cccp1"
  type: CONVOLUTION
  blobs_lr: 1
  blobs_lr: 2
  convolution_param {
    num_output: 96
    kernel_size: 1
    stride: 1
}
```



Starting from scratch

Neural architecture search:
Networks can be described as a series of operations
As series of words → text

We can turn the problem around:

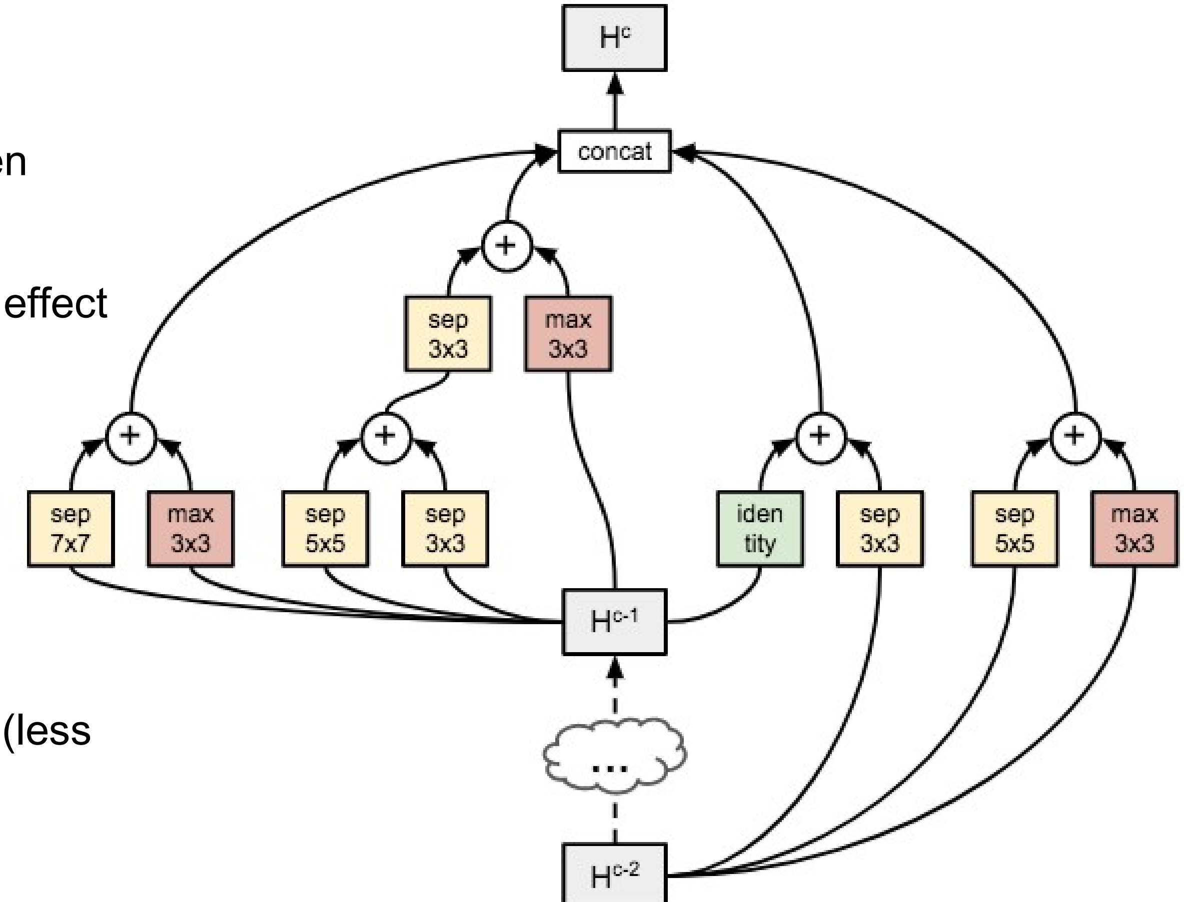
A recurrent network can be trained with reinforcement learning which can train a network with predifined accuracy on a given dataset.

This recurrent network will understand the effect of the elements on this dataset

Test accuracy On CIFAR-10:
96.35%

Best pervious accuracy:
96.26

This architecture os also 1.05 times faster (less computations)



Starting from scratch

Neural architecture search:
Networks can be described as a series of operations
As series of words → text

We can turn the problem around:

A recurrent network can be trained with reinforcement learning which can train a network with predifined accuracy on a given dataset.

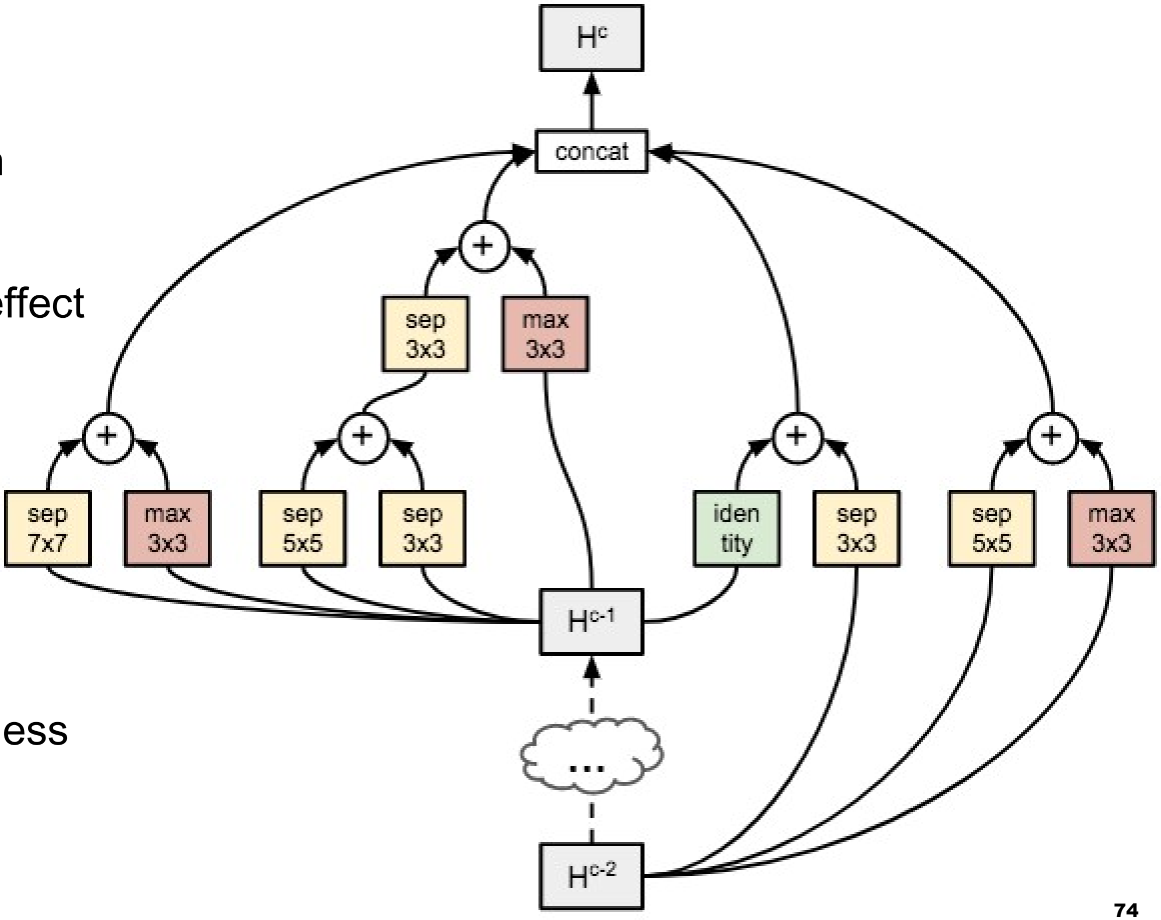
This recurrent network will understand the effect of the elements on this dataset

Test accuracy On CIFAR-10:
96.35%

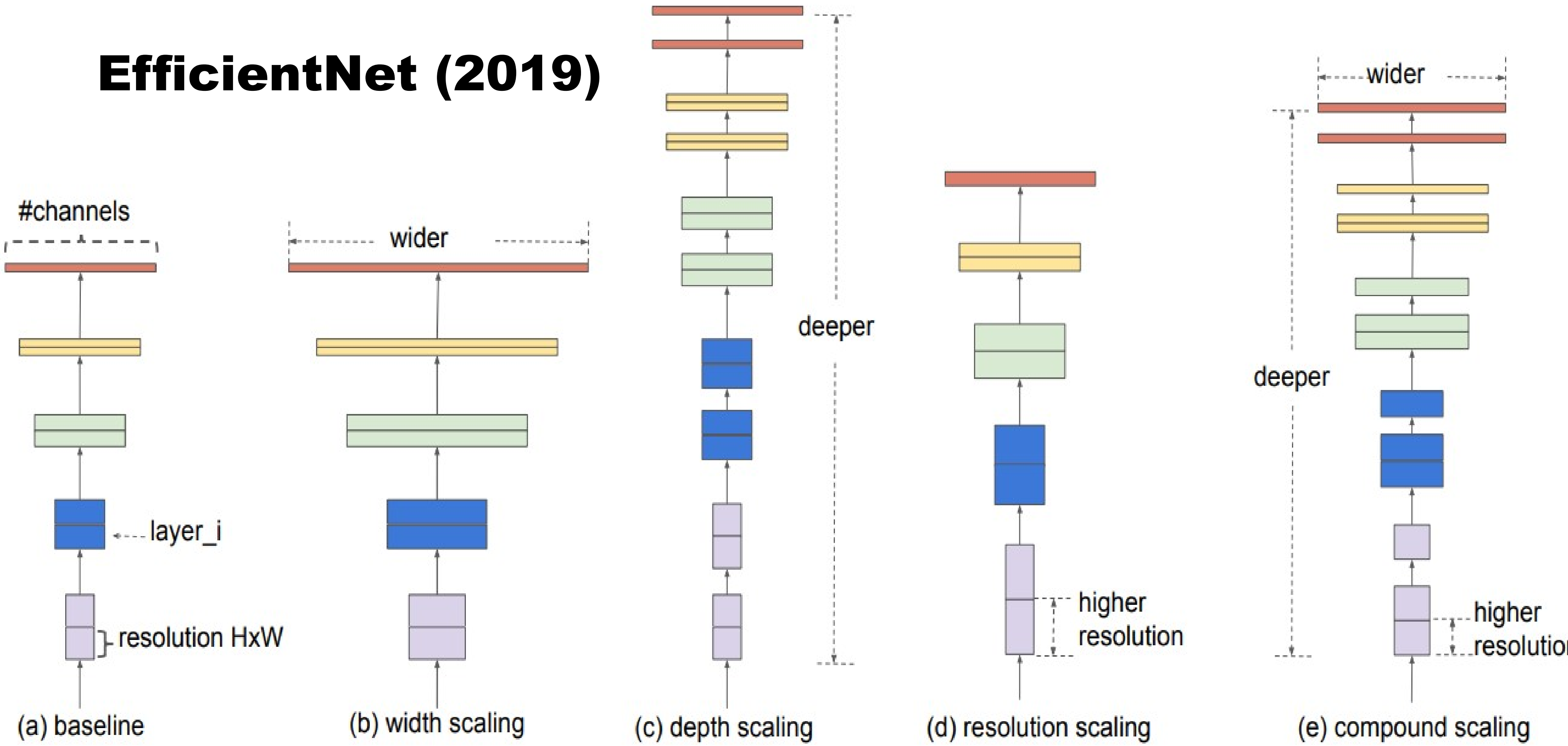
Best pervious accuraccy:
96.26

This architecture os also 1.05 times faster (less computations)

The important in this is that a network could design another network, and could reach as good performance as human.



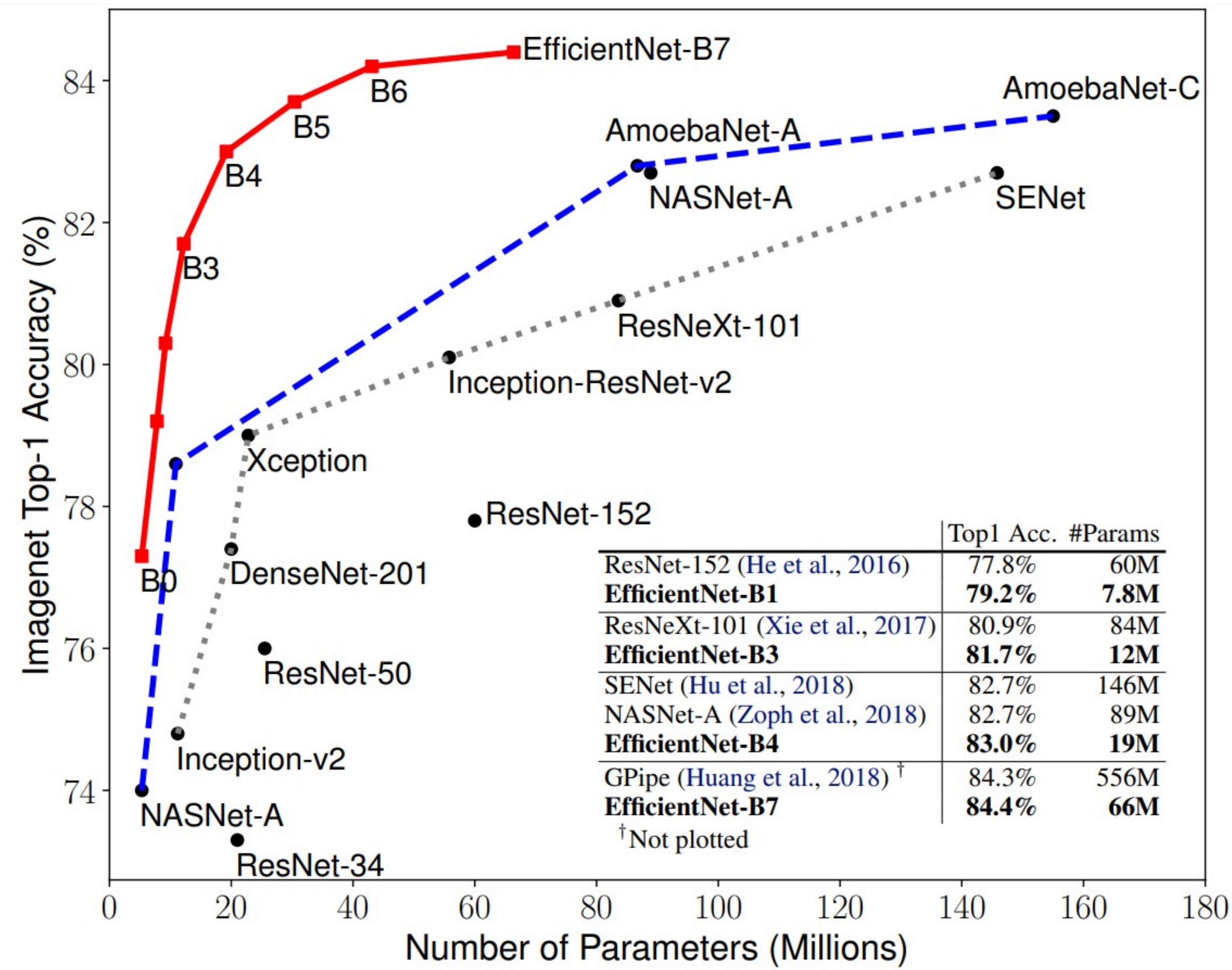
EfficientNet (2019)



- Scale the width, the depth, and the resolution uniformly!
- Can be used for any existing architecture, and the efficiency will be significantly better with the same performance
    - EfficientNet-B7 achieves stateof-the-art 84.4% top-1 / 97.1% top-5 accuracy on ImageNet, while being 8.4x smaller (number of parameters) and 6.1x faster on inference than the best existing ConvNet.
- Best performance can be reached by using NN to generate the optimal baseline ConvNet.

EfficientNet (2019)

7 different scaled version of EfficientNet. (B0, B1, … B7)



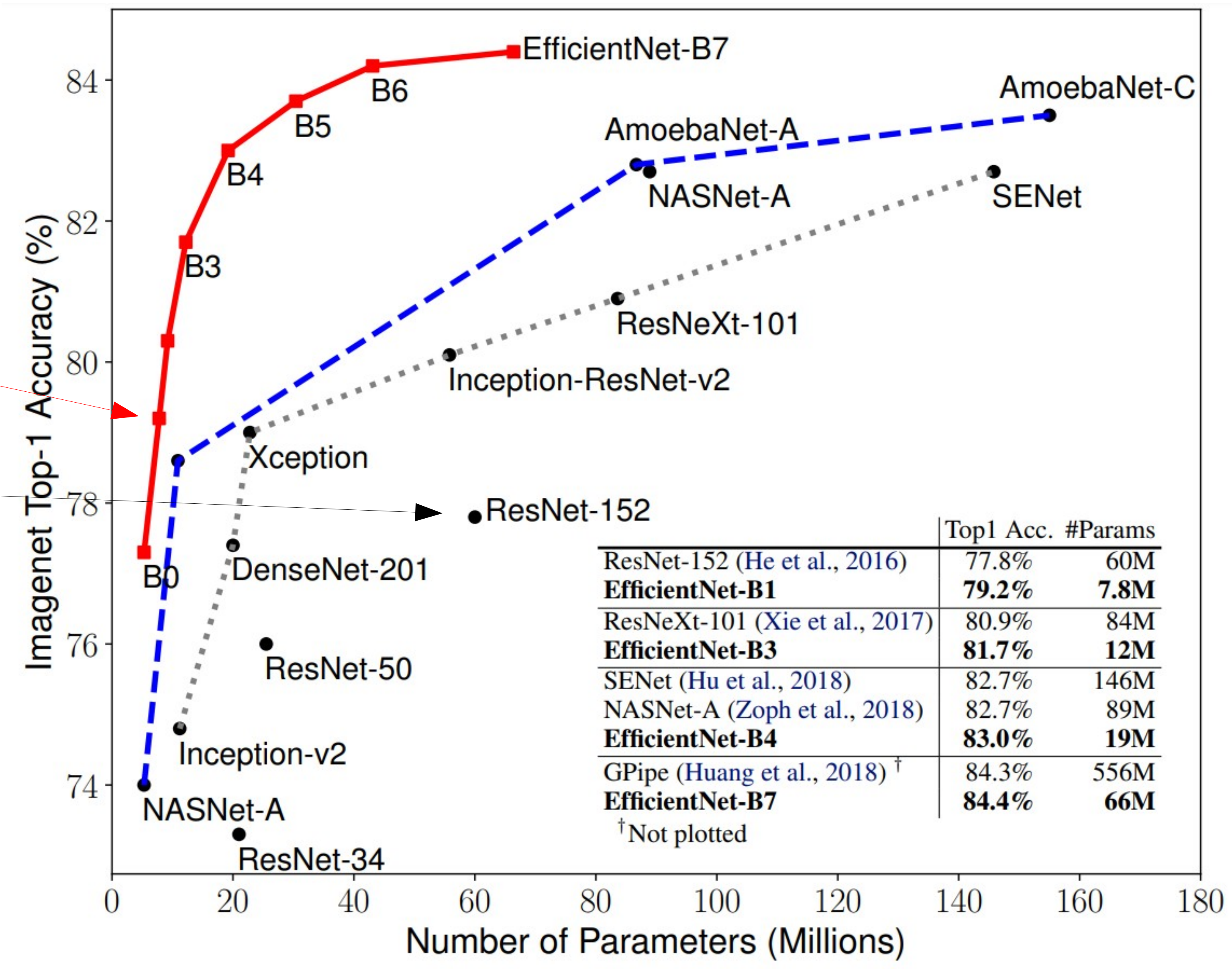
	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.2%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.7%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**83.0%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

EfficientNet (2019)

7 different scaled version of EfficientNet. (B0, B1, … B7)

EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.2%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.7%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**83.0%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

† Not plotted

Visualizing the Decision of Neural Networks

Soma Kontár & András Horváth

Budapest, 2019.12.10

Administrative details

The replacement paper-based test will be on 17 December

The midterm project code submission deadline is Friday, 13 Dec 23:59 via uploading to a shared Google Drive folder (the link will be posted later on the course website)

The midterm project presentations will also be on 17 December

The computer based test will be on 19 December

We will discuss the details of the early exam with the participants in the break

Disclaimer

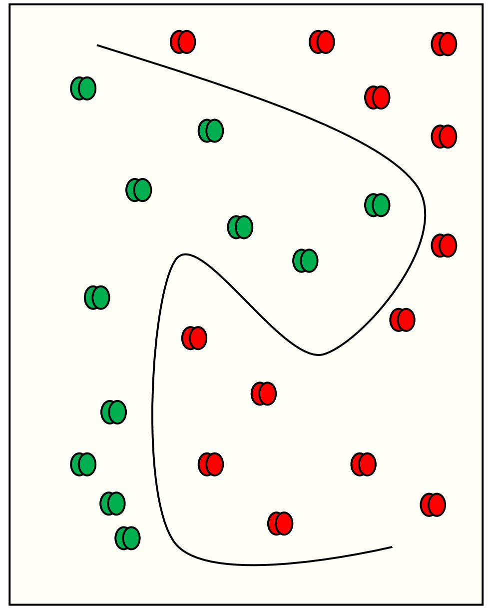
The slides are based on the lectures titled visaulizing and understanding Neural Networks at Stanford. Created by Justin Johnso, Andrej Karpathy and Fei-Fei Li.

Neural Networks

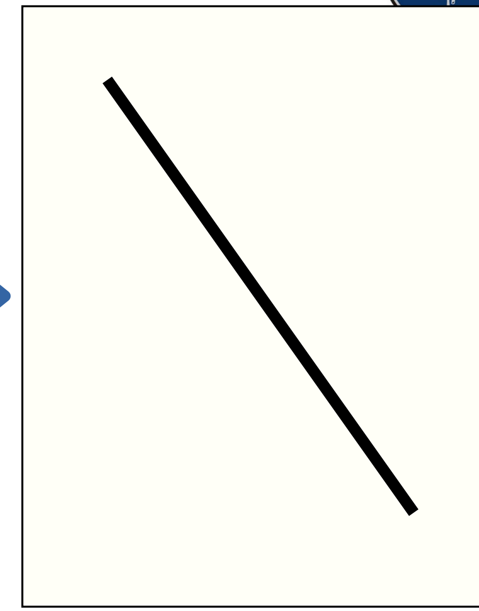
- Classification - decision
- FNN, SVM – linear classification

Is X larger than a limit? X>k?

- Finding a good feature representation:
  - Meaningful
  - Sparse - low dimensions
  - Ensures easy separation



Input space



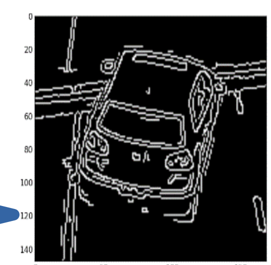
Feature space

Finding the representation with the help of machine learning

Input Image

Convolution Kernel

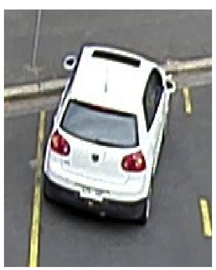
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



Feature Image

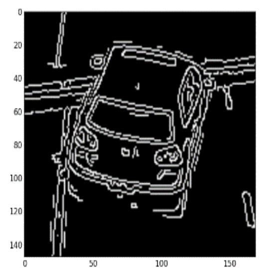
Convolutional neural networks

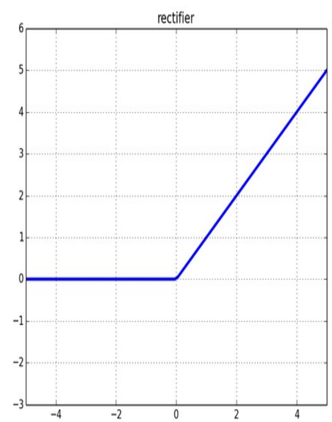
- A network of simple processing elements
  - Elements:
    - Convolution
    - ReLU
    - Pooling



Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$





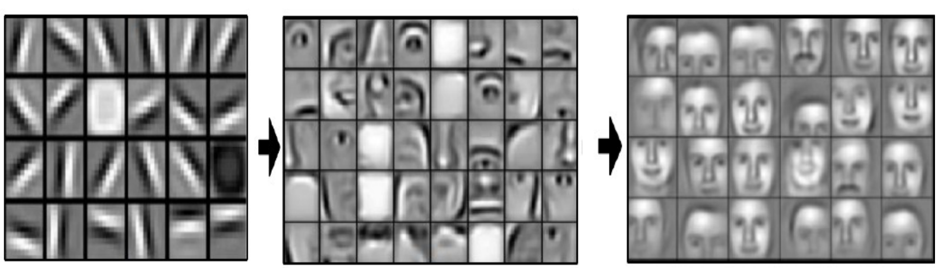
Thresholding all values below zero

1	0	2	3
4	6	6	8
3	1	1	0
1	2	2	4

→

6	8
3	4

Selection of the maximal response in an area



Low layers *Middle layers* *High layers*

Conquest of neural networks

Neural networks work great in various problems

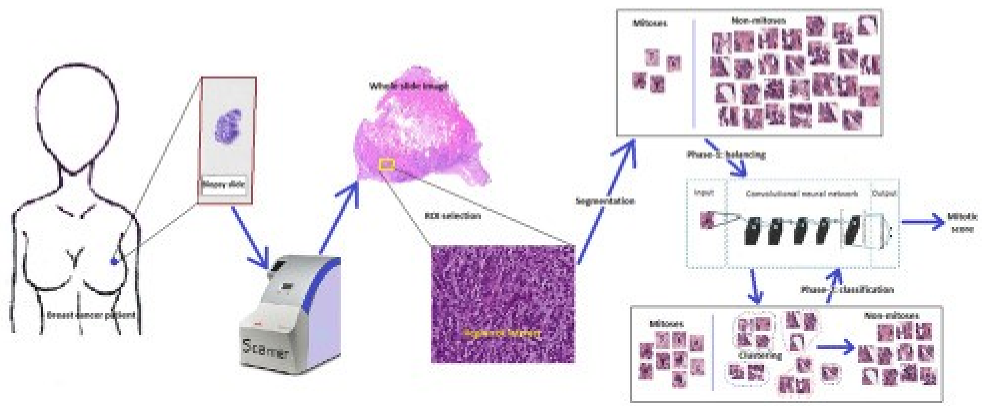
They are capable of solving complex practical tasks

Classification

Segmentation

Reinforcement learning

Image captioning



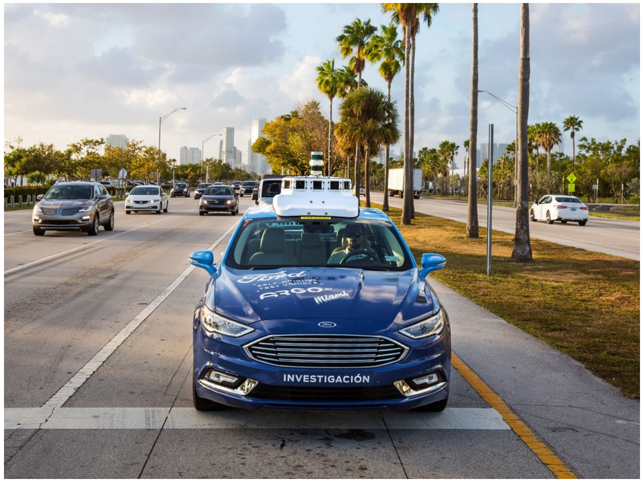


Image captioning

"A train is on the tracks at a station"

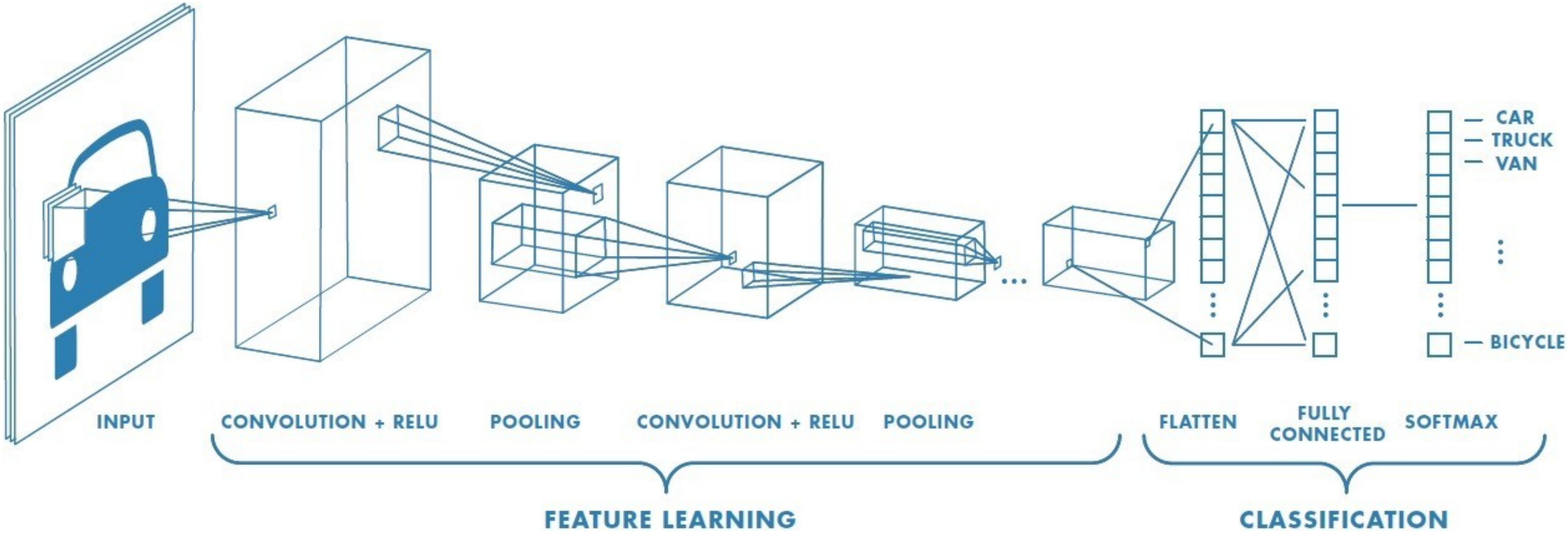
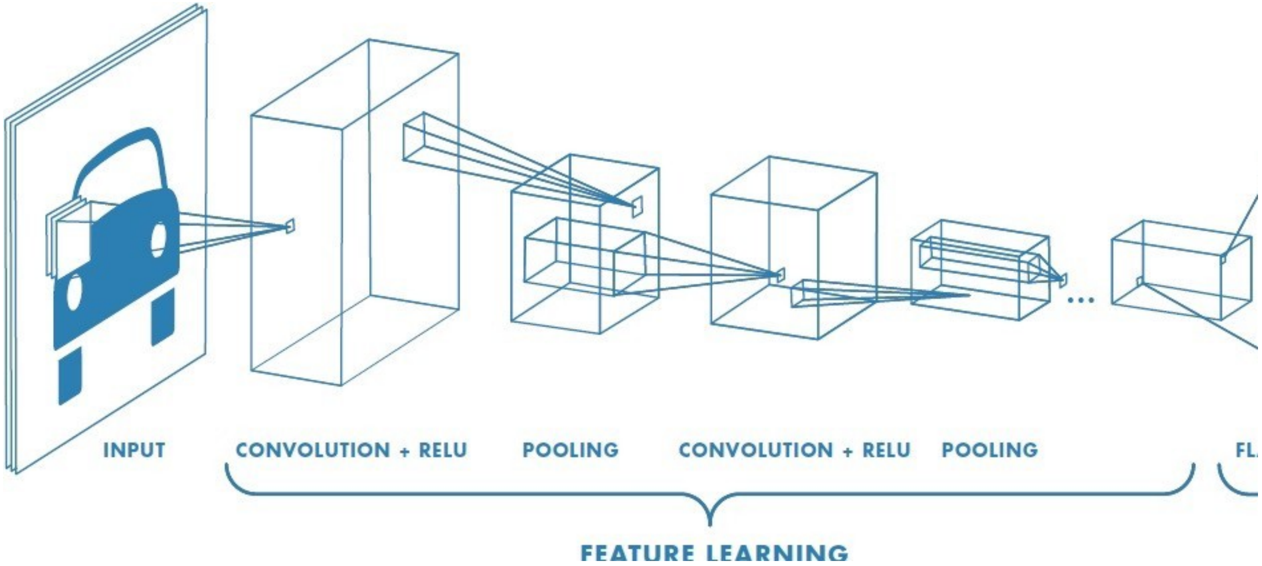
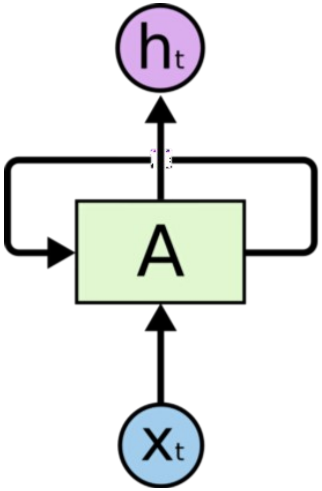


Image captioning

“A train is on the tracks at a station”





MSCOCO

a snowboarder jumping over snow indoors with the coca-cola logo in the background.

person on a snow board up in the air inside of a building

a man is jumping over two coca cola signs.

a room filled with fake white snow under stickers.

fake snow inside a snowboarding facility of some sort

MSCOCO

a picture of a computer screen featuring the face of a movie actor.

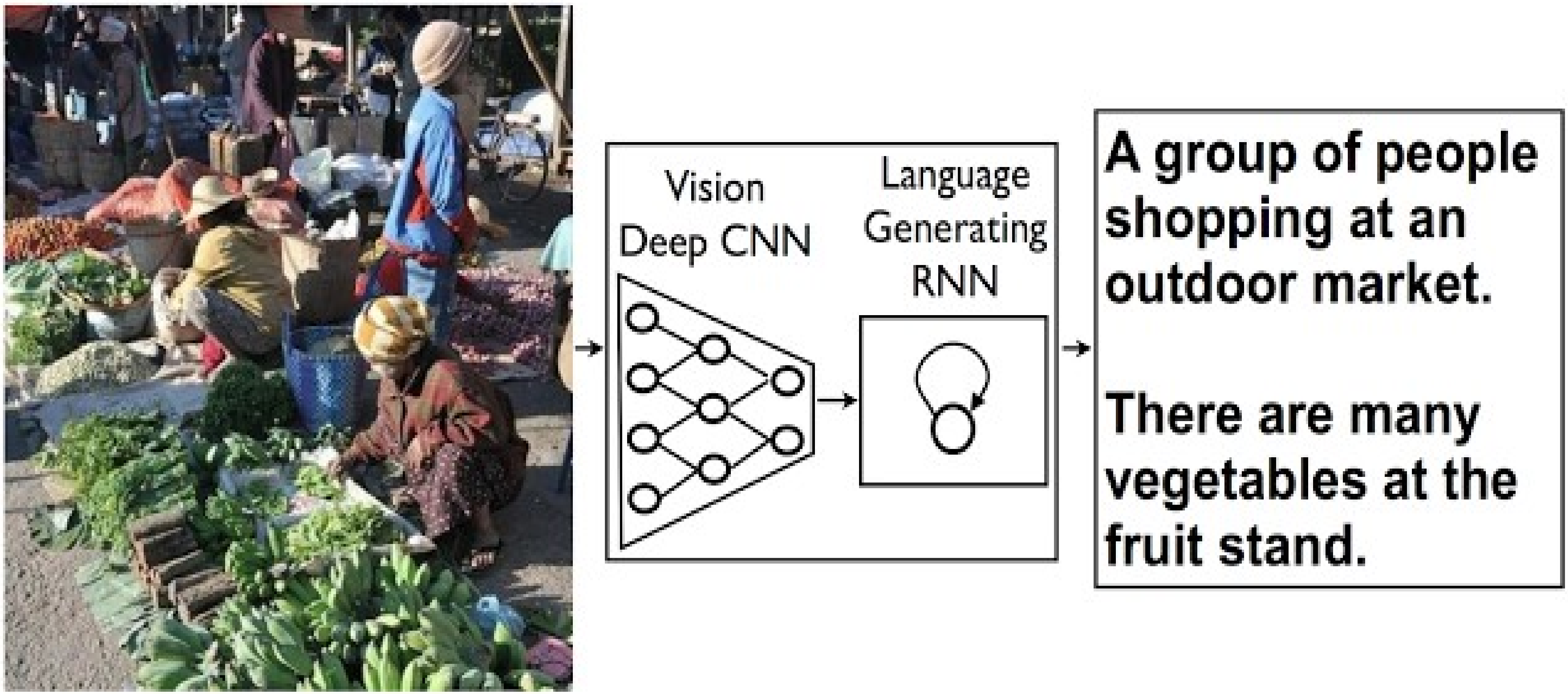
a computer screen on a table showing a man's face.

here is actor mark wahlberg on skype with someone at a home laptop.

a laptop computer with marky mark on it's screen.

a laptop is open and the screen shows mark wahlberg.

Neural Network results



https://arxiv.org/pdf/1411.4555.pdf

Neural Network results

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."

"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

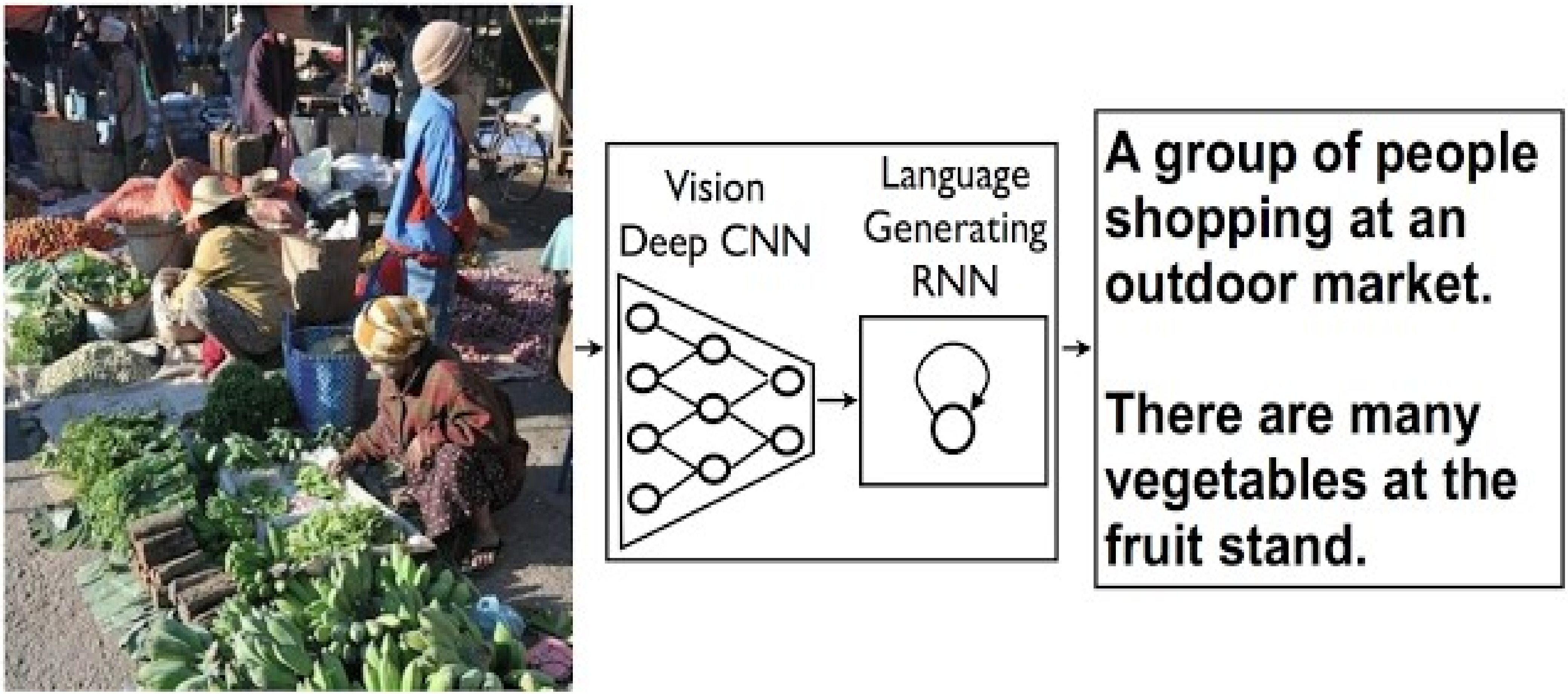
"young girl in pink shirt is swinging on swing."

"man in blue wetsuit is surfing on wave."

Credits: Fei-Fei Li, andrej Karpathy

A refrigerator filled with lots of food and drinks

Not so good...



https://arxiv.org/pdf/1411.4555.pdf

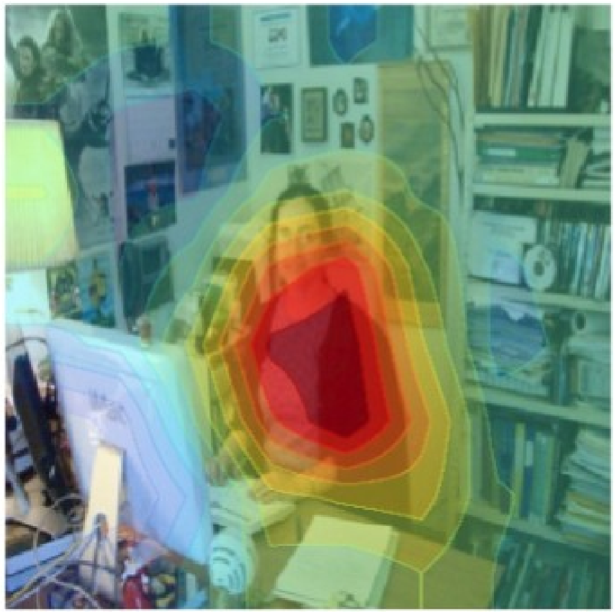
Understanding decisions

If we can understand (or even trace back) network decision we will be able to see if the network managed to grasp the important features in the dataset

Wrong

Baseline:
*A **man** sitting at a desk with a laptop computer.*

Right for the Right Reasons



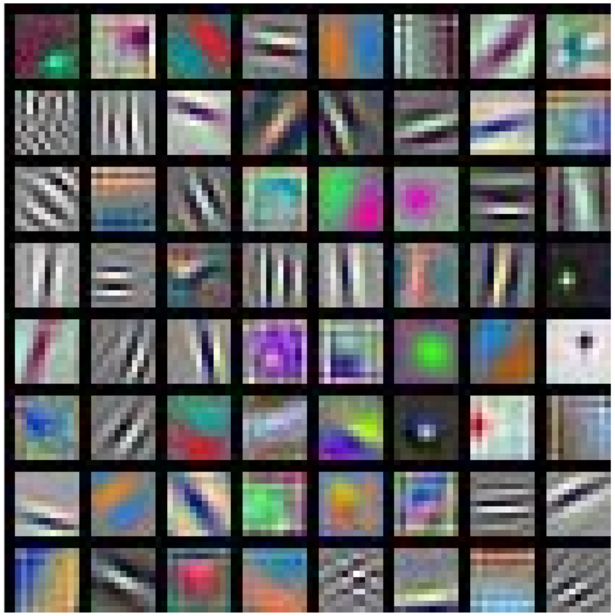
Our Model:
*A **woman** sitting in front of a laptop computer.*

Lisa Anne Hendricks*, Kaylee Burns*, Kate Saenko, Trevor Darrell, Anna Rohrbach:Women also Snowboard: Overcoming Bias in Captioning Models

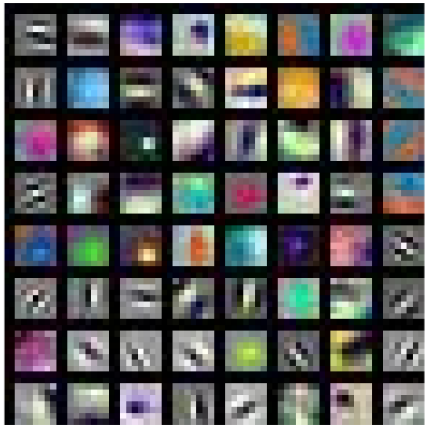
What is going on inside a convnet?

Filter visualization

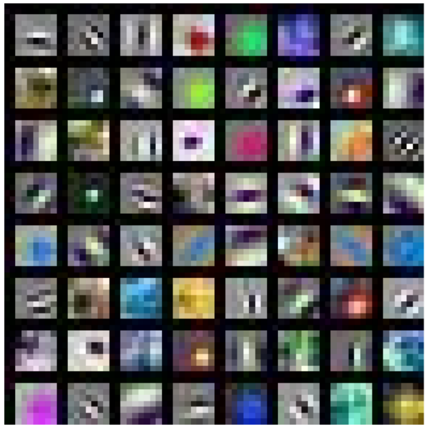
Display the filters what the network has learned



AlexNet:
64 x 3 x 11 x 11



ResNet-18:
64 x 3 x 7 x 7



ResNet-101:
64 x 3 x 7 x 7

DenseNet-121:
64 x 3 x 7 x 7

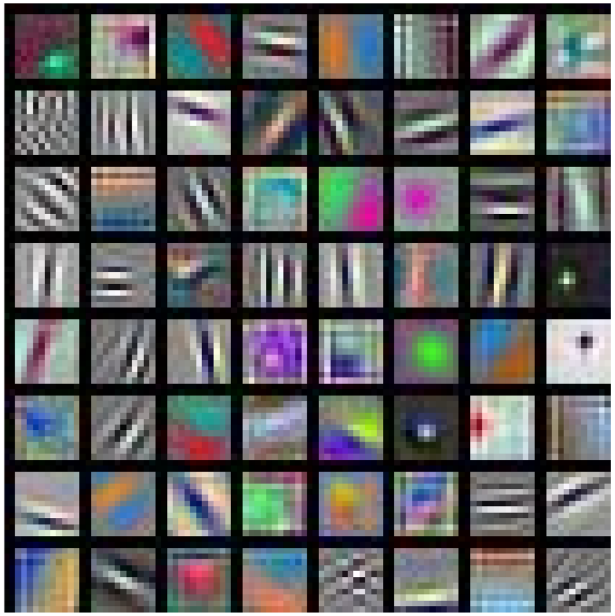
Good to display the first layer(s)

The functionality of higher layer kernels is difficult to see

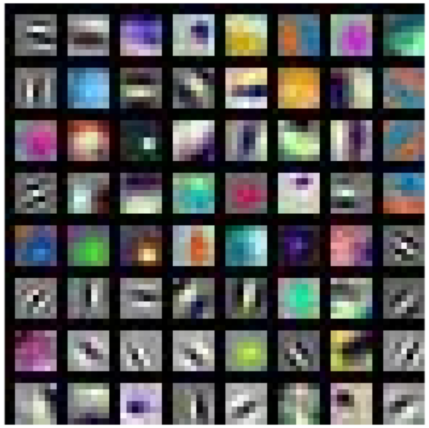
What is going on inside a convnet?

Filter visualization

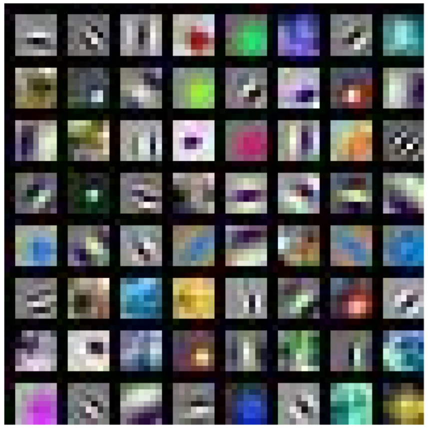
Display the filters what the network has learned



AlexNet:
64 x 3 x 11 x 11



ResNet-18:
64 x 3 x 7 x 7



ResNet-101:
64 x 3 x 7 x 7

DenseNet-121:
64 x 3 x 7 x 7

http://users.itk.ppke.hu/~horan/CNN/convnetjs/convnet.html

Good to display the first layer(s)

The functionality of higher layer kernels is difficult to see

Displaying the decision space of the network

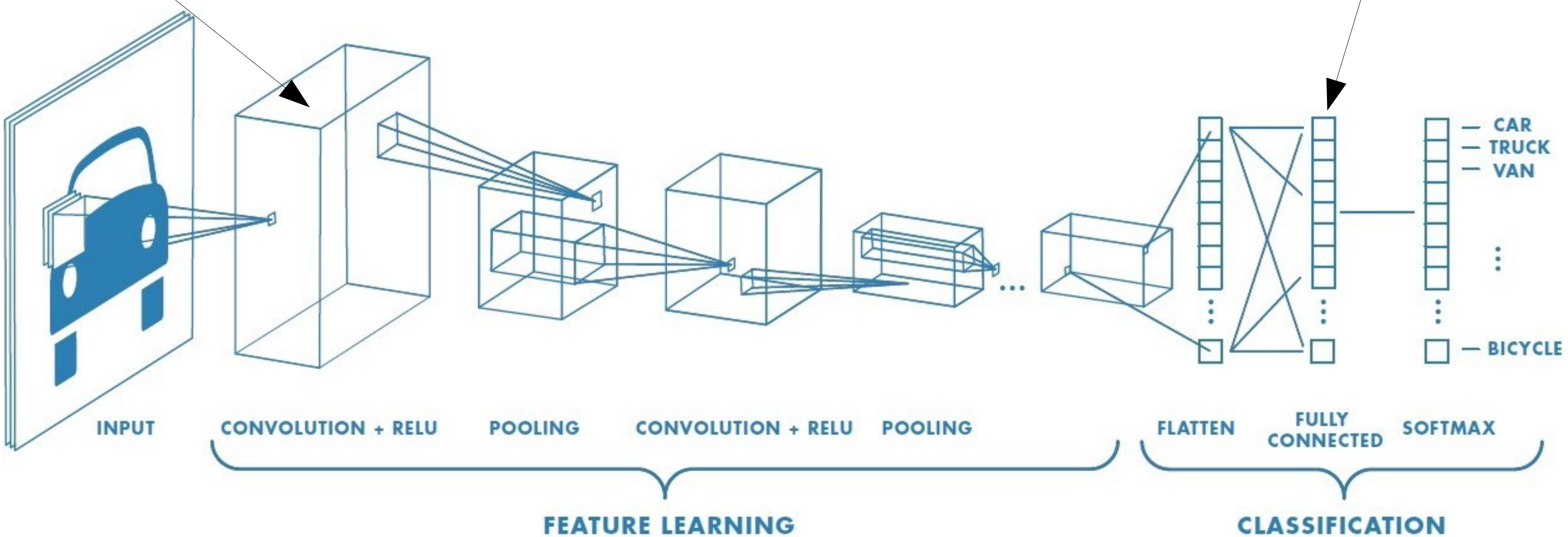
In higher layer kernels work in an abstract spaces

We can not really understand functionality just by visualizing the kernels

Unfortunately these kernels are closer, more determining in the decision than the first layers

Work closer to the image space
Kernel visualization is good

Really important in decision, kernels work in an abstract space

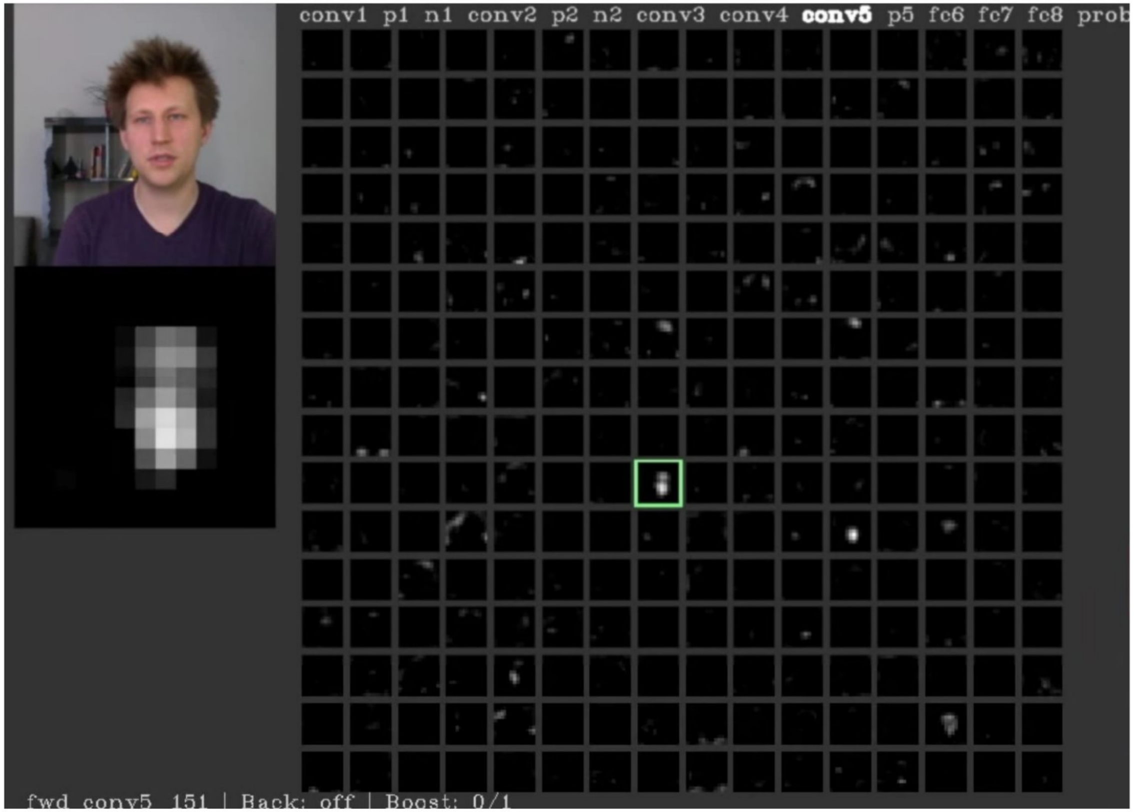


Finding activations

Visualizing activations

Instead of visaulizing the kernels we could visualize activations

Kernel visualization is good, because it is input independent. For this we need an input image



Yosinski et al, “Understanding Neural Networks Through Deep Visualization”, ICML DL Workshop 2014.

Finding activations

Visualizing activations

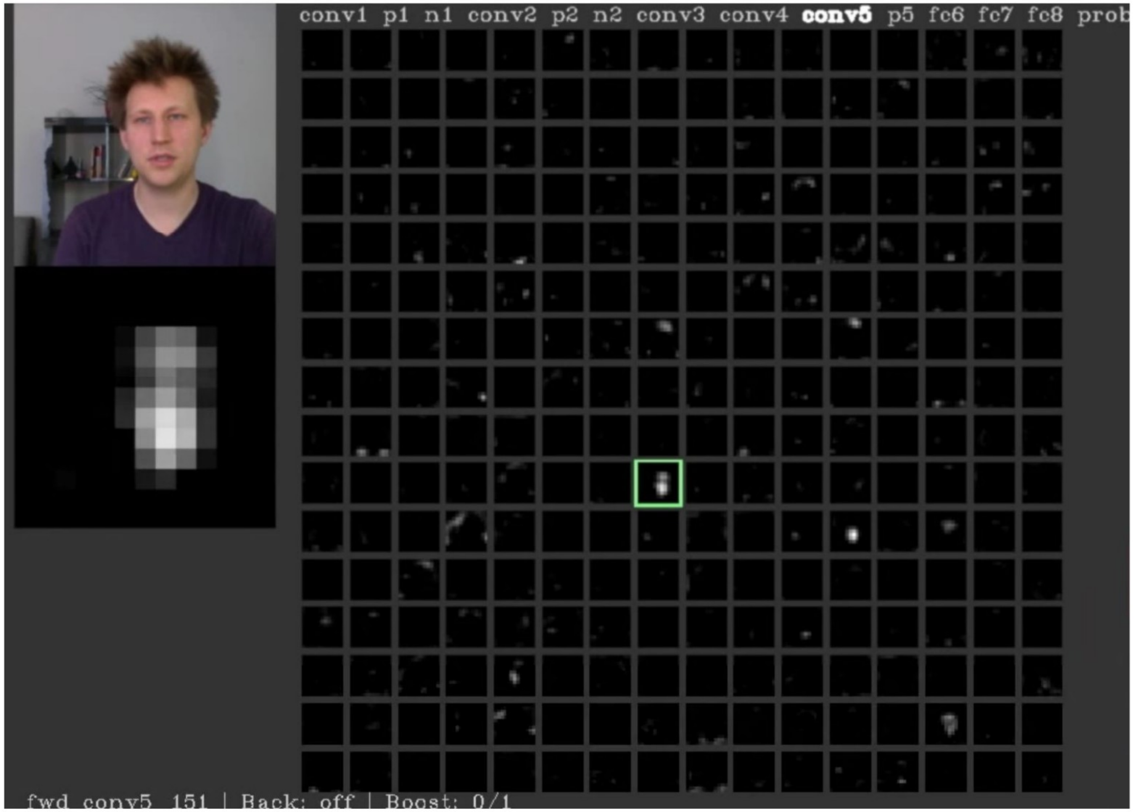
Instead of visaulizing the kernels we could visualize activations

Kernel visualization is good, because it is input independent. For this we need an input image

Activations should be sparse in a high layer

If a neuron is never/always active, it is not good
Responses should be specific

The same neuron should fire for similar inputs



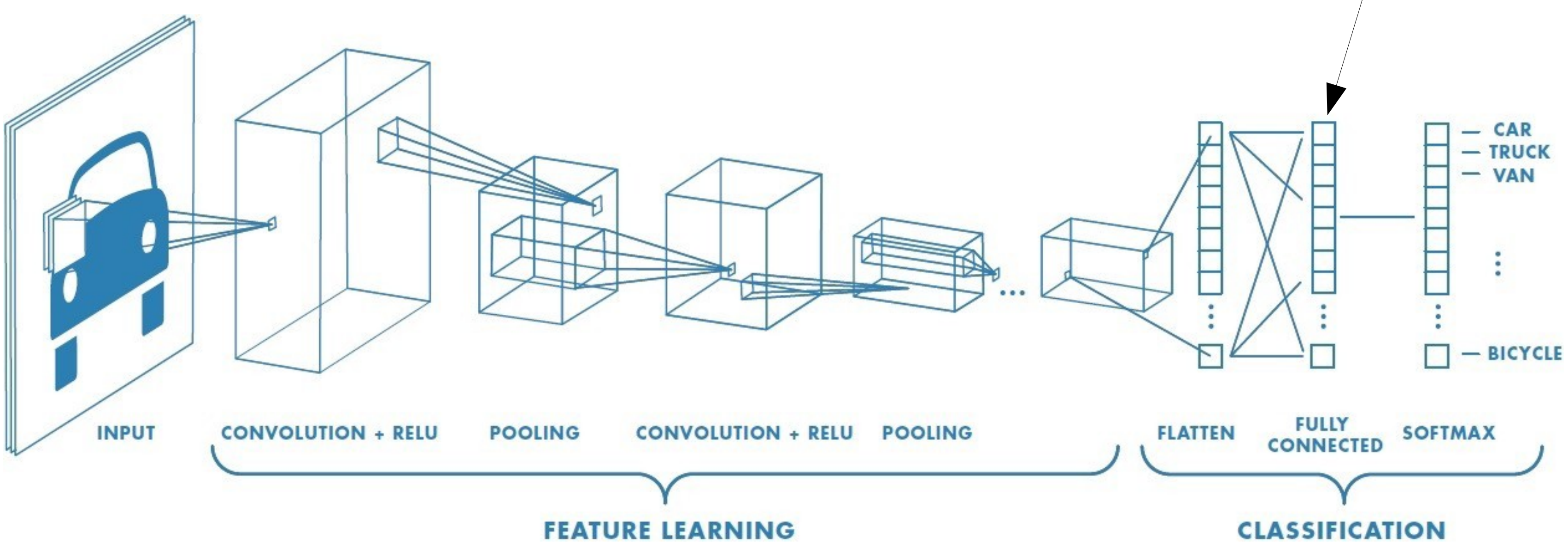
Yosinski et al, “Understanding Neural Networks Through Deep Visualization”, ICML DL Workshop 2014.
Figure copyright Jason Yosinski, 2014. Reproduced with permission.

Displaying the decision space of the network

We focus on the last feature layer (one before the logit layer)

This is usually a non-topographical vector. Every input is a point in a high-dimensional (e.g.: 1024) space

Really important in decision, kernels work in an abstract space



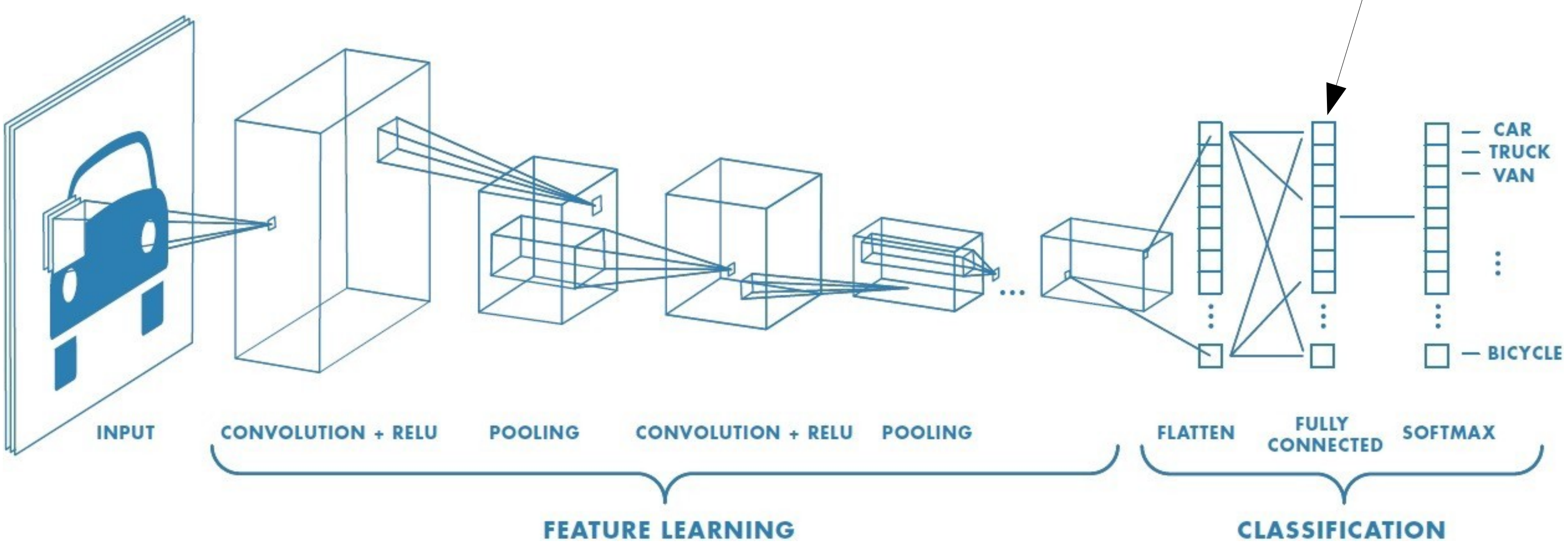
Displaying the decision space of the network

We focus on the last feature layer (one before the logit layer)

This is usually a non-topographical vector. Every input is a point in a high-dimensional (e.g.: 1024) space

We can not plot this high-dimensional space, but:

Really important in decision, kernels work in an abstract space

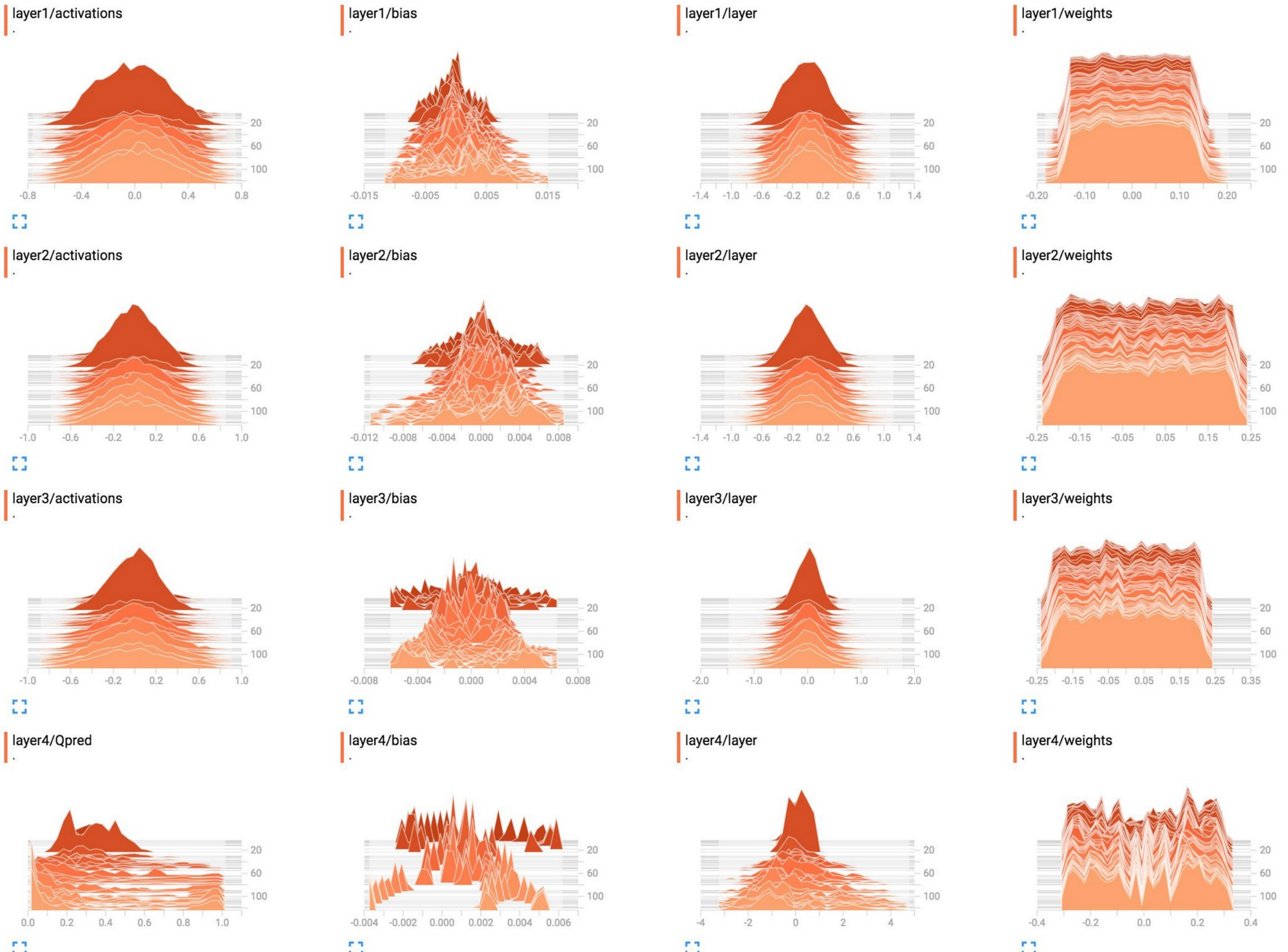


Finding activations

Visualizing activations

Instead of visaulizing the kernels we could visualize activations

Tensorboard is a great tool to display activations/weights

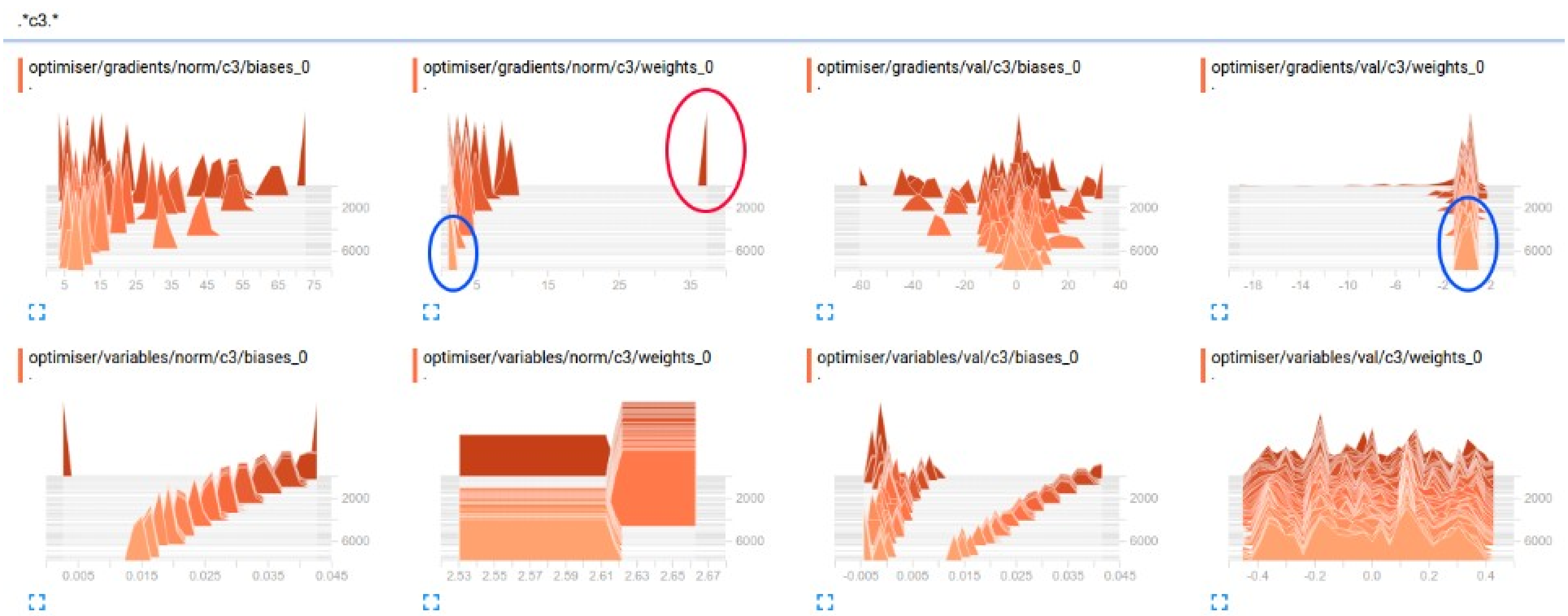


Finding activations

Visualizing activations

Instead of visaulizing the kernels we could visualize activations

Tensorboard is a great tool to display activations/weights



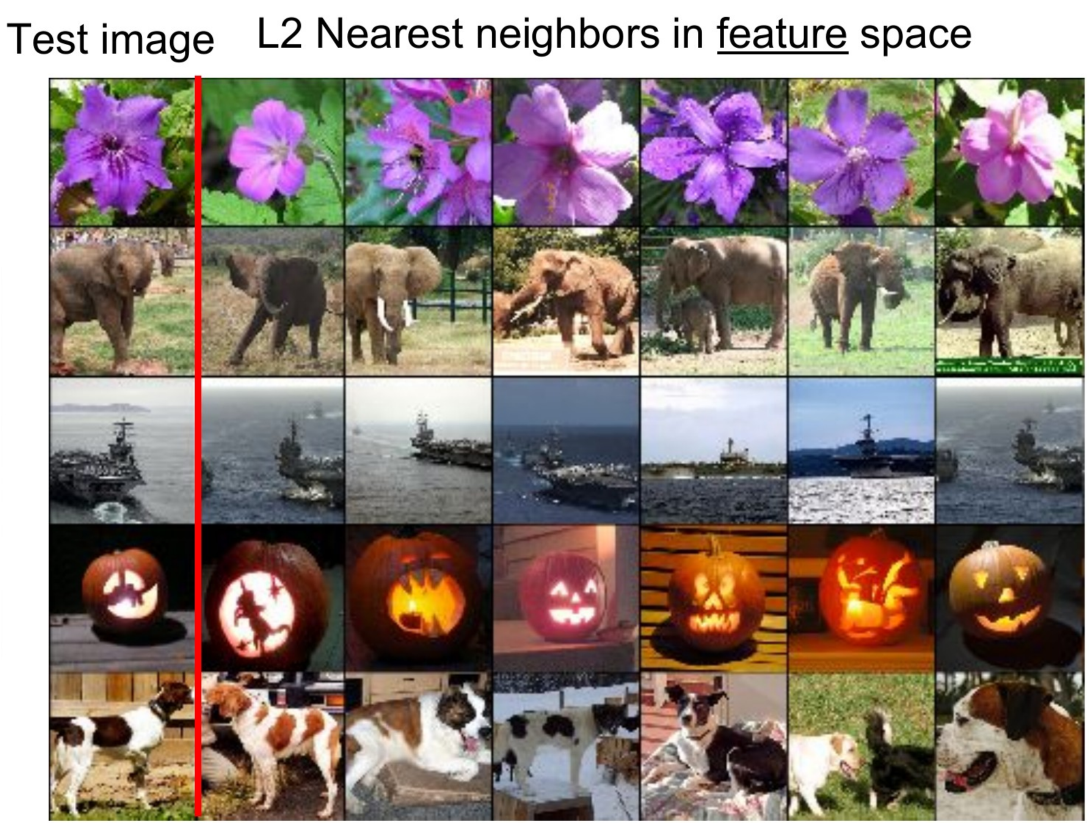
Displaying the decision space of the network

We focus on the last feature layer (one before the logit layer)

This is usually a non-topographical vector. Every input is a point in a high-dimensional (e.g.: 1024) space

We can not plot this high-dimensional space, but:

We can plot nearest neighbours: Select an input image, and find the closest n image in this space (if they are similar the network grasped something important)



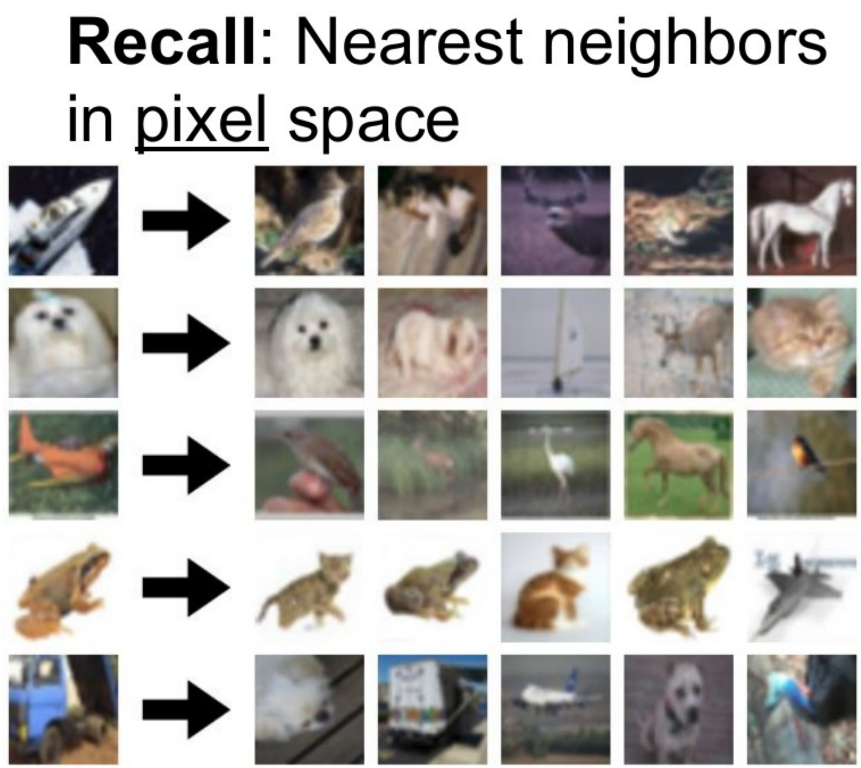
Displaying the decision space of the network

We focus on the last feature layer (one before the logit layer)

This is usually a non-topographical vector. Every input is a point in a high-dimensional (e.g.: 1024) space

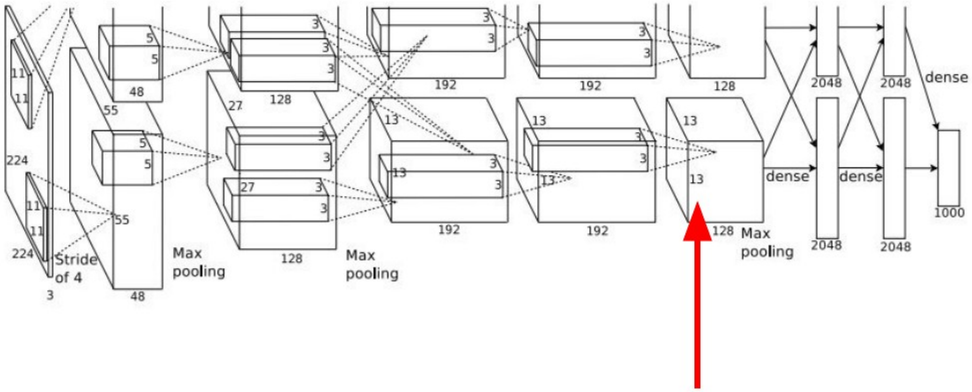
We can not plot this high-dimensional space, but:

We can plot nearest neighbours: Select an input image, and find the closest n image in this space (if they are similar the network grasped something important)



Finding activations

We can find those images in the dataset which will maximize its activation



Springenberg et al, “Striving for Simplicity: The All Convolutional Net”, ICLR Workshop 2015

Finding activations

We can combine the two previous methods and select a kernel/filter (a neuron representing it)

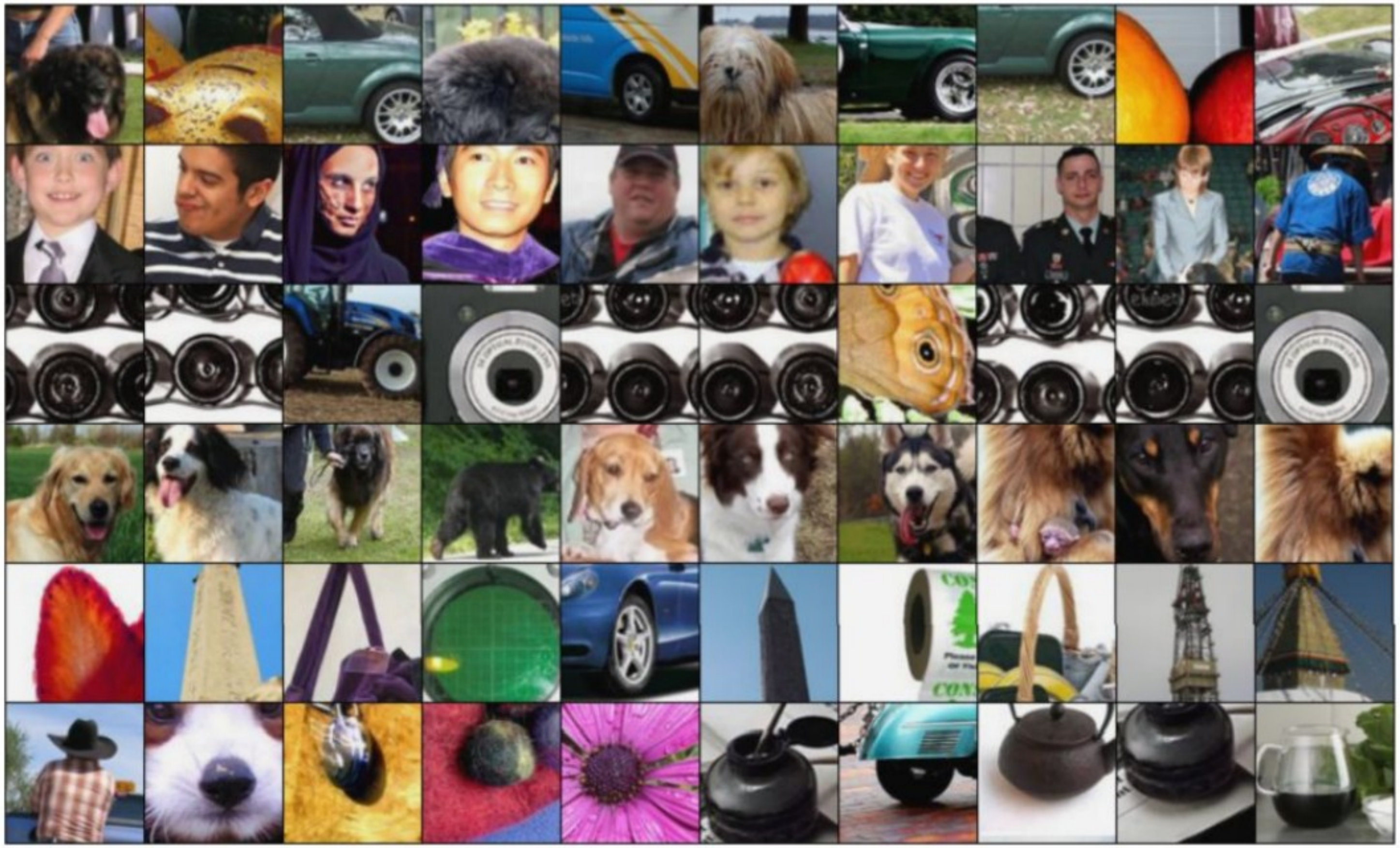
And find those images in the dataset which will maximize its activation

Springenberg et al, “Striving for Simplicity: The All Convolutional Net”, ICLR Workshop 2015

Finding activations

We can combine the two previous methods and select a kernel/filter (a neuron representing it)

And find those images in the dataset which will maximize its activation



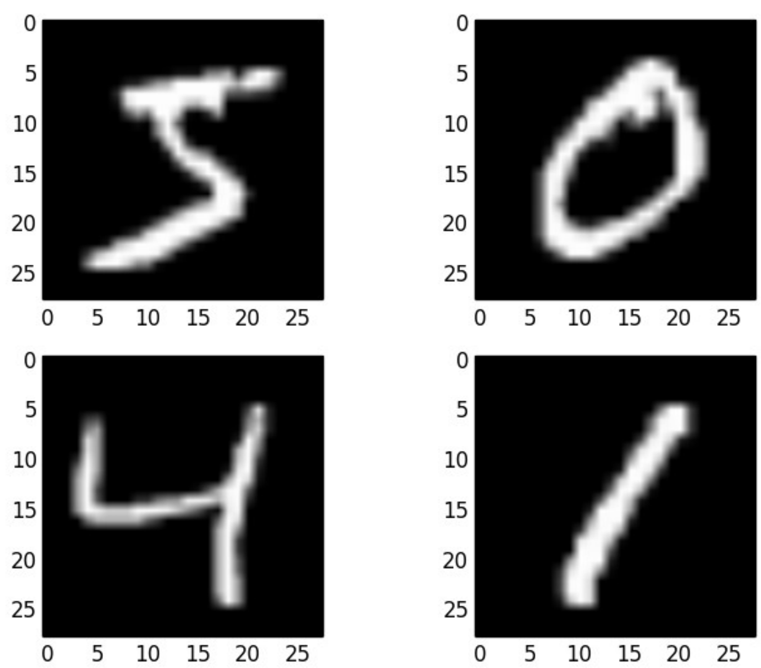
Springenberg et al, “Striving for Simplicity: The All Convolutional Net”, ICLR Workshop 2015

Finding activations

We can combine the two previous methods and select a kernel/filter (a neuron representing it)

And find those images in the dataset which will maximize its activation

With this method one can easily find the typical element for a class



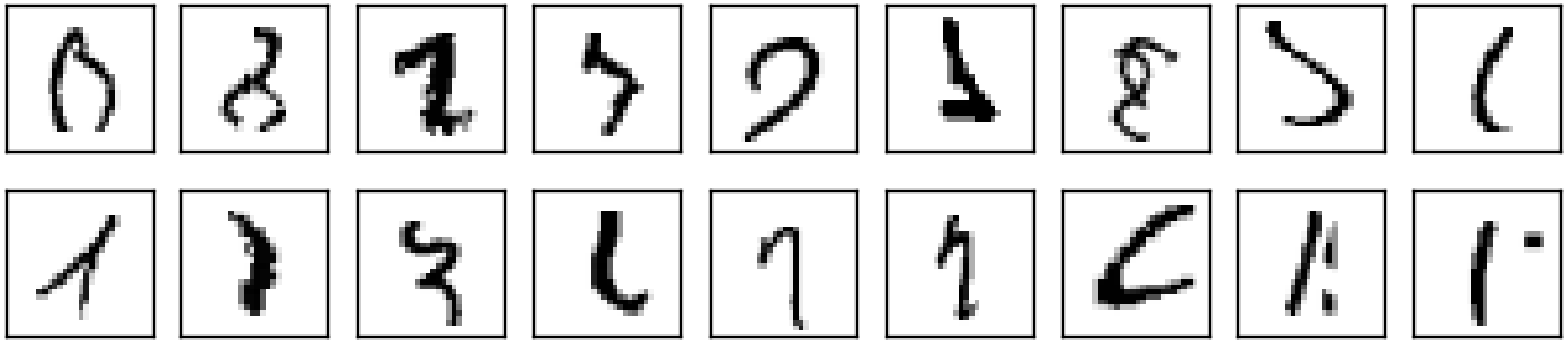
Finding activations

We can combine the two previous methods and select a kernel/filter (a neuron representing it)

And find those images in the dataset which will maximize its activation

With this method one can easily find the typical element for a class

Or find those elements where the classifier was “uncertain”



Displaying the decision space of the network

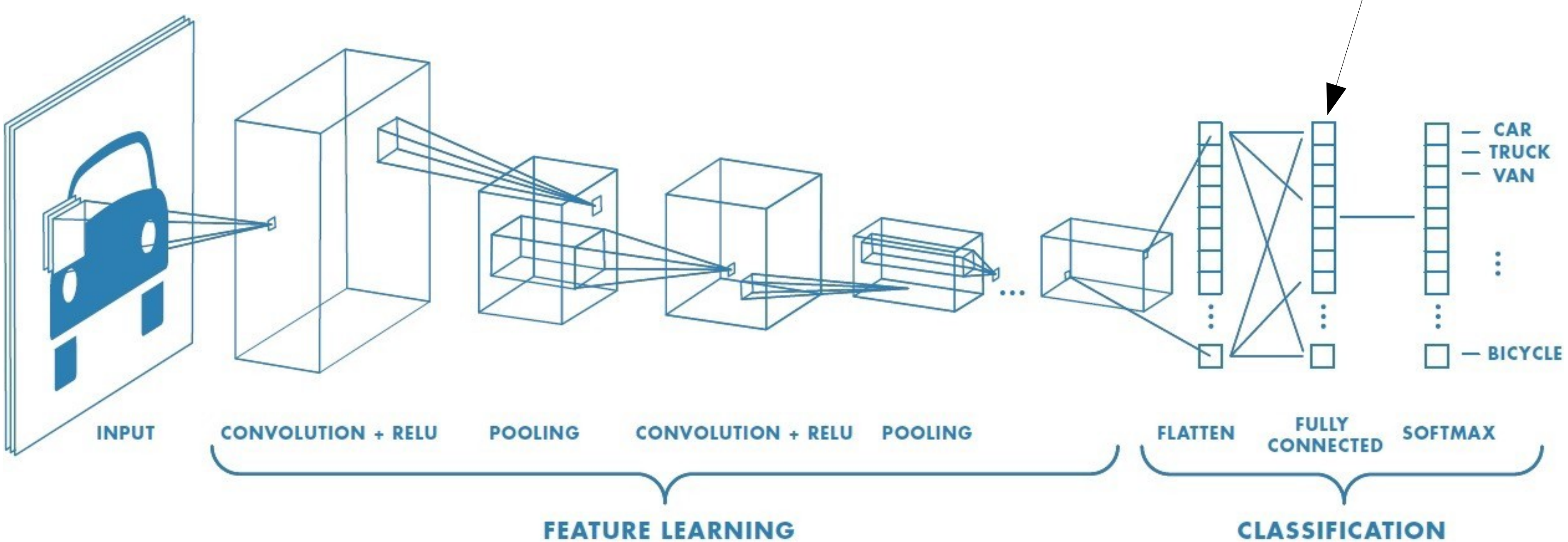
We focus on the last feature layer (one before the logit layer)

This is usually a non-topographical vector. Every input is a point in a high-dimensional (e.g.: 1024) space

We can not plot this high-dimensional space, but:

We could project this data into a lower-dimensional subspace

Really important in decision, kernels work in an abstract space



Dimension reduction

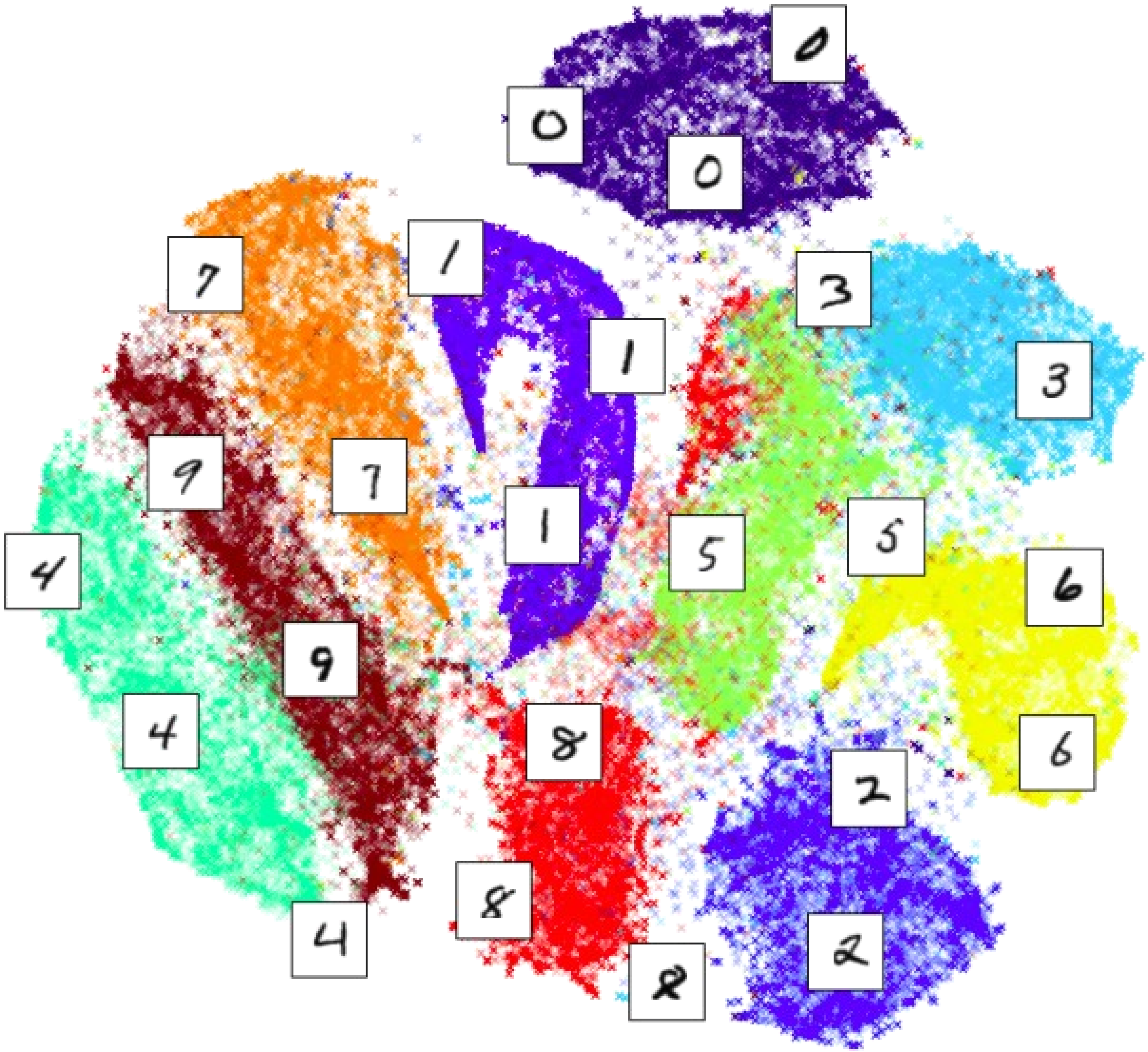
PCA/Autoencoder

T-SNE

Dimension reduction

PCA/LDA/Autoencoder

T-SNE

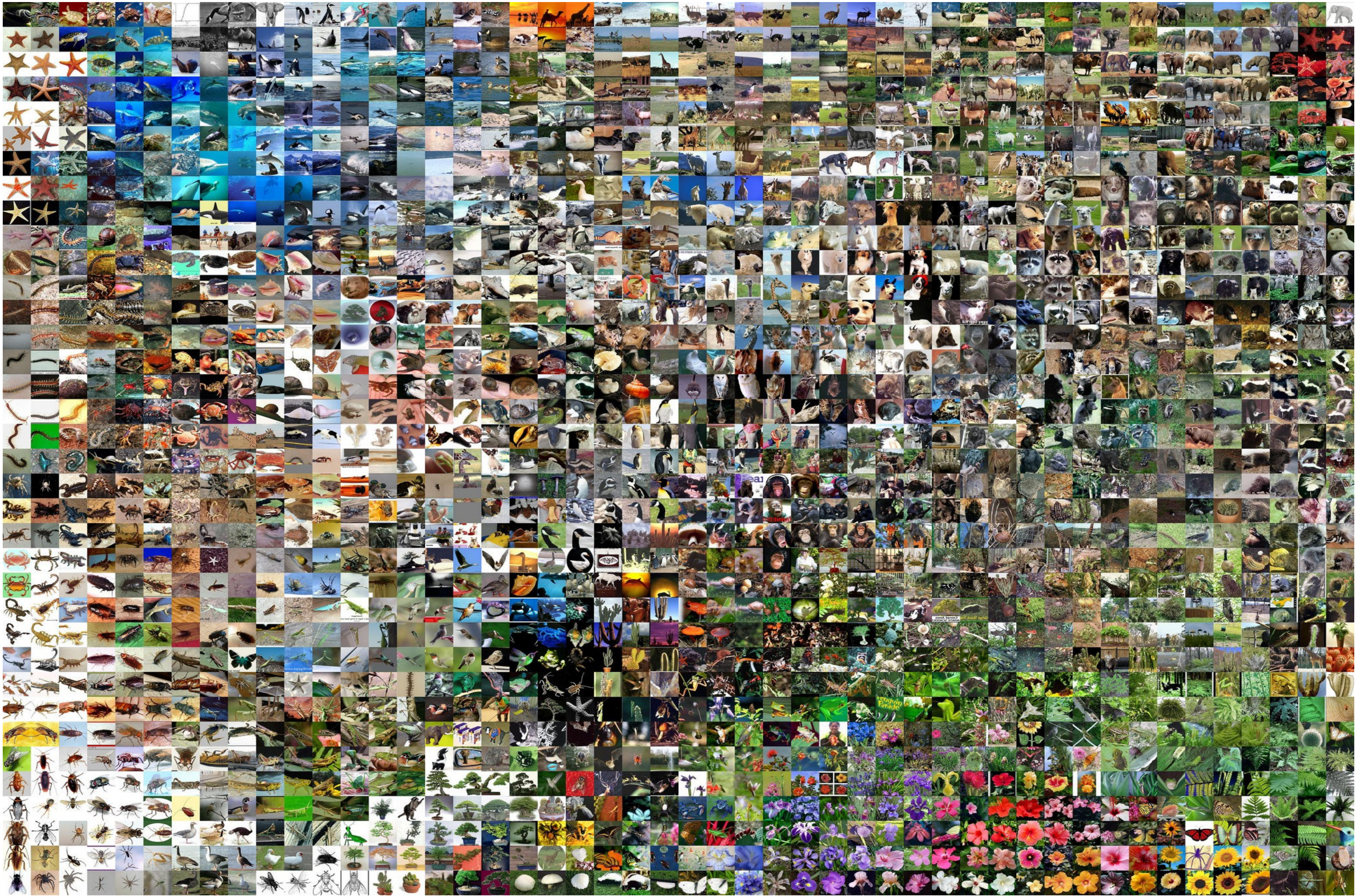


Dimension reduction

PCA/LDA/Autoencoder

T-SNE

T-SNE



Dimension reduction

PCA/LDA/Autoencoder

T-SNE

https://cs.stanford.edu/people/karpathy/tsnejs/

Typical examples

We could search in our database and find typical samples.

It helps, but usually the network is good on this set (train accuracy). We are curious about those images which the network has not seen.

Could we generate and ideal image for a given class?

Gradient Ascent

We could search in our database and find typical samples.

It helps, but usually the network is good on this set (train accuracy). We are curious about those images which the network has not seen.

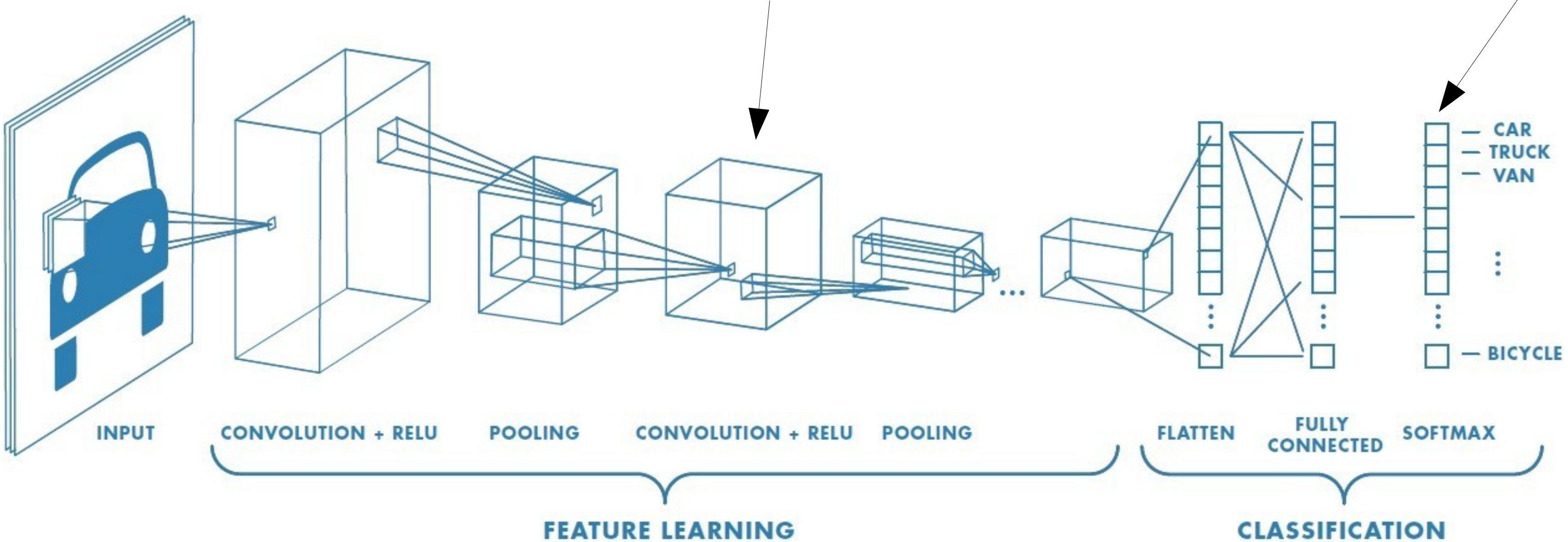
Could we generate and ideal image for a given class?

Normal training

Input Image - Given

Network parameters - Variables

Expected Label - Given



Gradient Ascent

We could search in our database and find typical samples.

It helps, but usually the network is good on this set (train accuracy). We are curious about those images which the network has not seen.

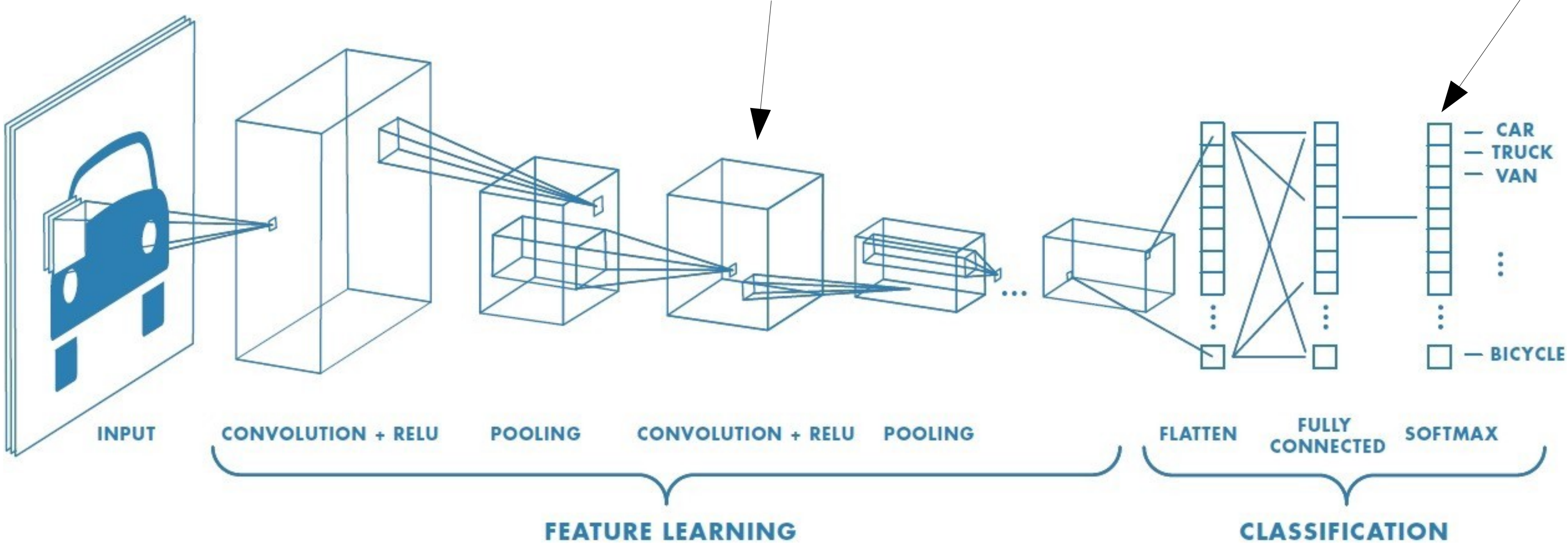
Could we generate and ideal image for a given class?

The gradient ascent method

Input Image - Variable

Network parameters - Given

Expected Label - Given

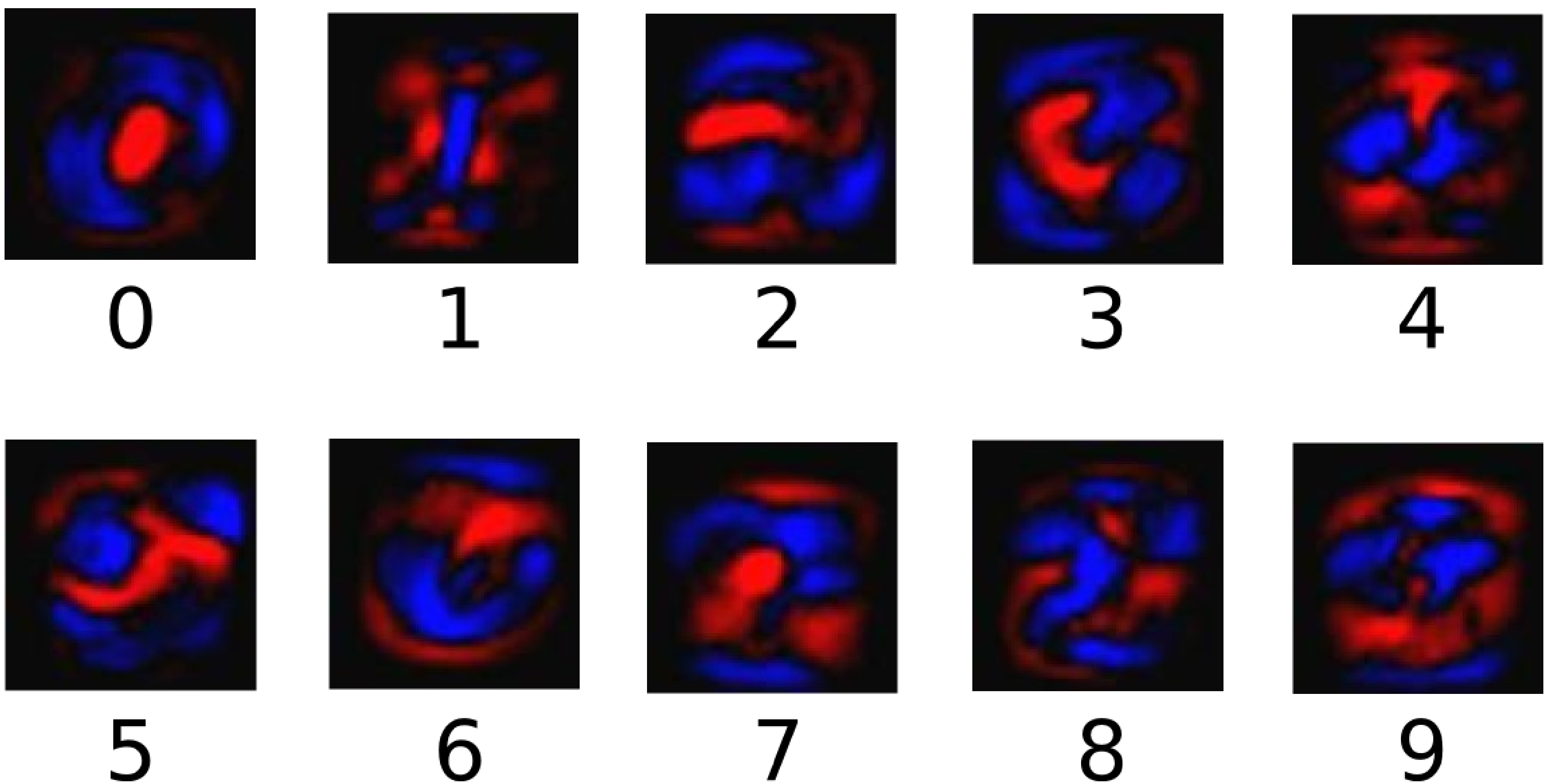


Gradient Ascent – activation maximization

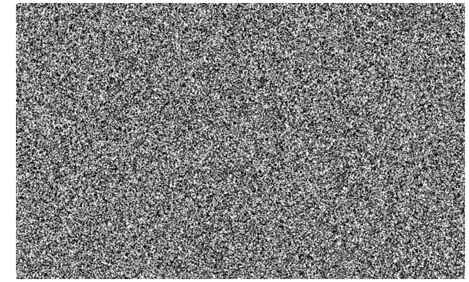
We could search in our database and find typical samples.

It helps, but usually the network is good on this set (train accuracy). We are curious about those images which the network has not seen.

Could we generate and ideal image for a given class?



Gradient Ascent



Generate a synthetic image that maximizes the response of a neuron.

This image has to be „natural”. The response should not depend on pixels and can not have arbitrary values

- Guassian blur on the image

- Clipping image values

- Clipping small gradients to 0

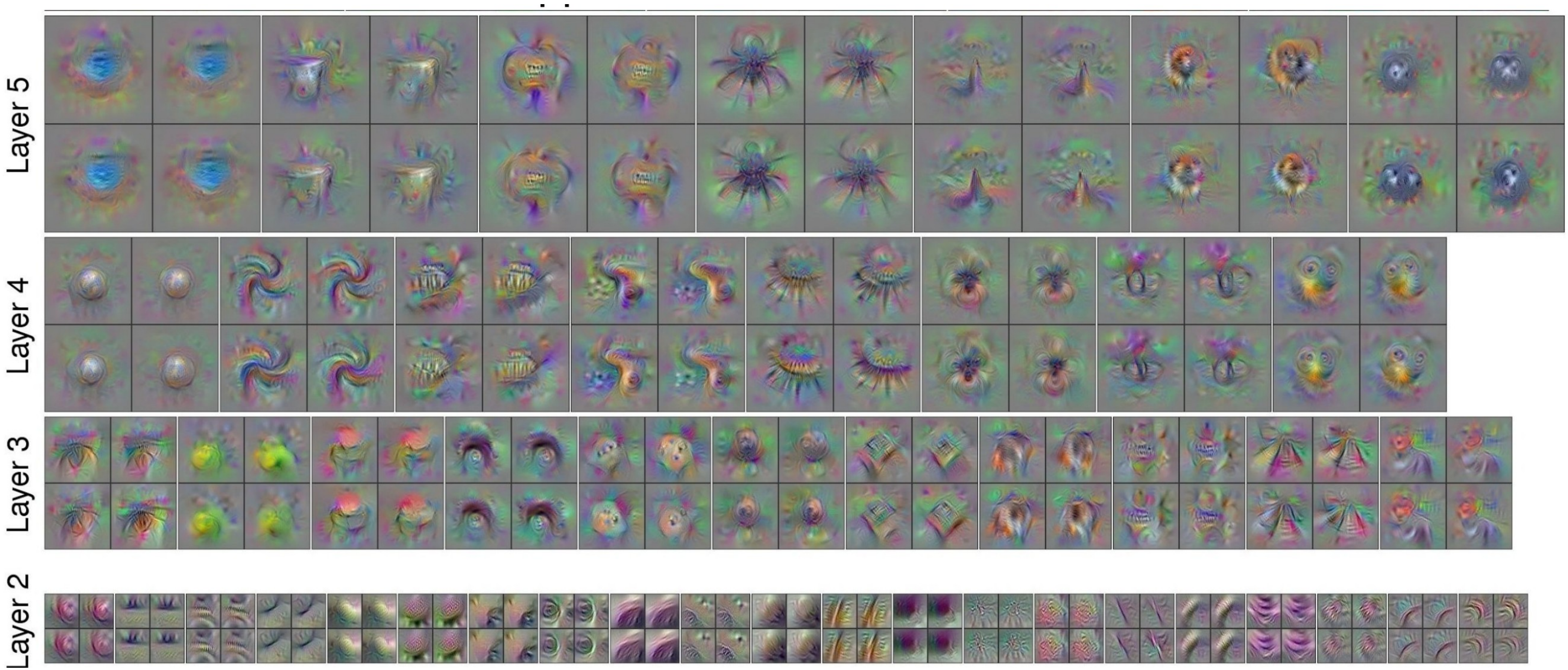
$$I^* = \arg\max_I \boxed{f(I)} + \boxed{R(I)}$$

Neuron value

Natural image regularizer

Gradient Ascent

Intermediate Layers

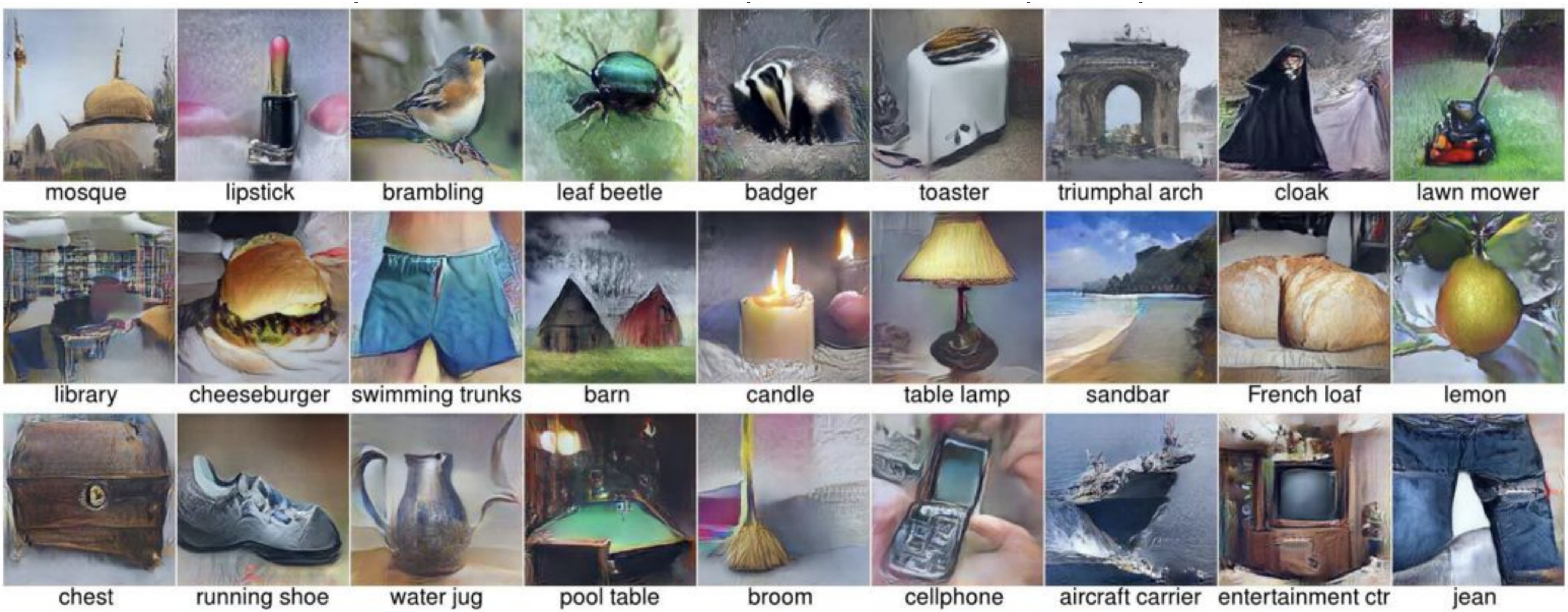


Gradient Ascent

Classes

Gradient Ascent

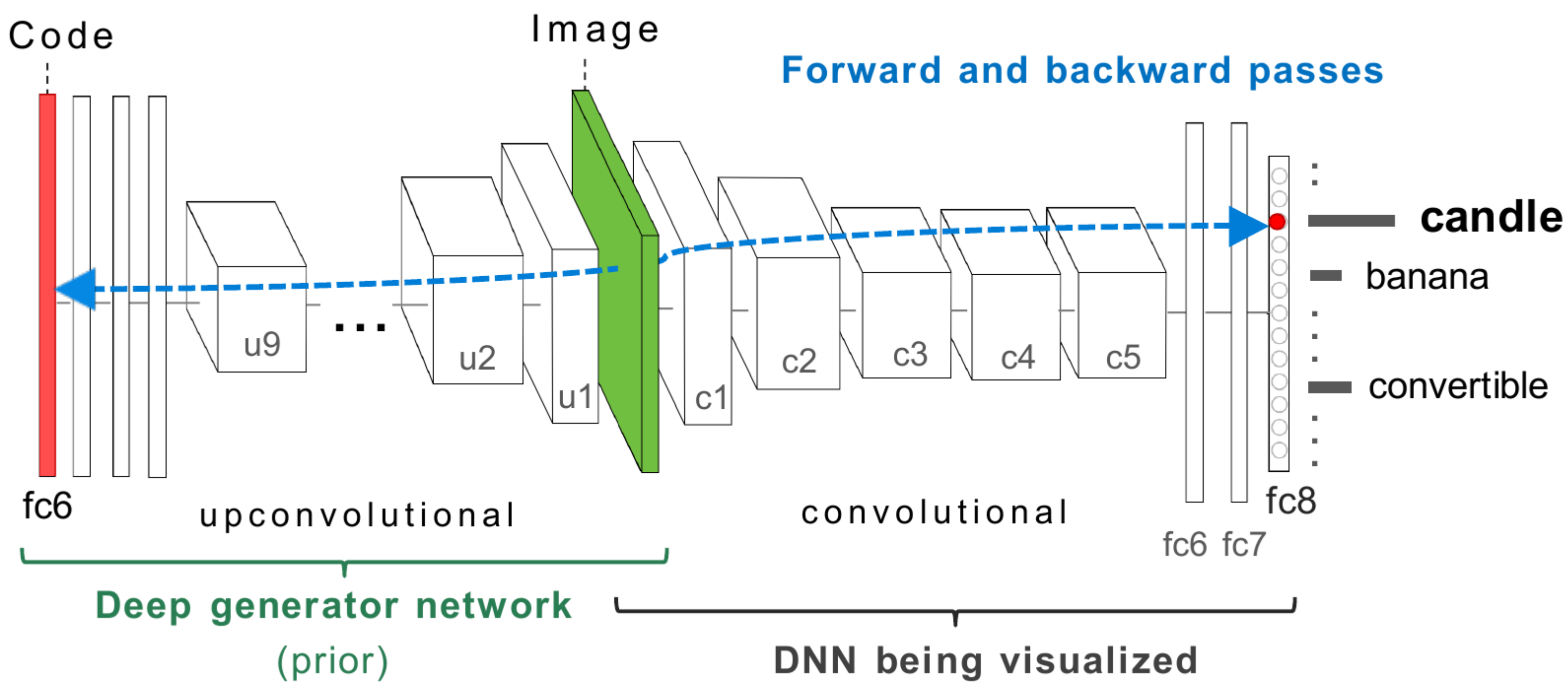
Using a network which can learn feature inversion



Nguyen et al, "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks," NIPS 2016

Gradient Ascent

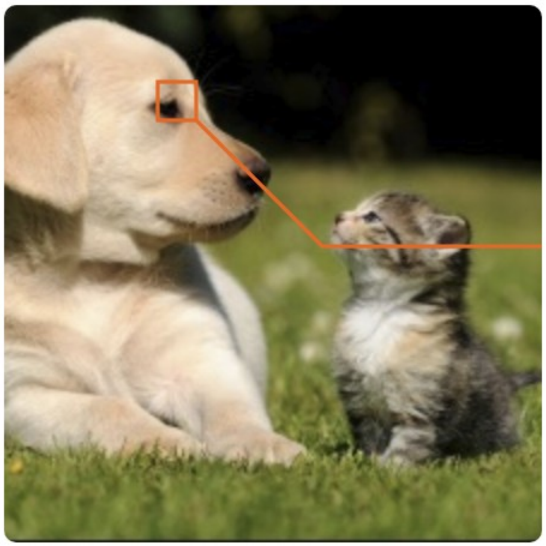
Using a network which can learn feature inversion



Nguyen et al, "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks," NIPS 2016
Figure copyright Nguyen et al, 2016;

Gradient Ascent

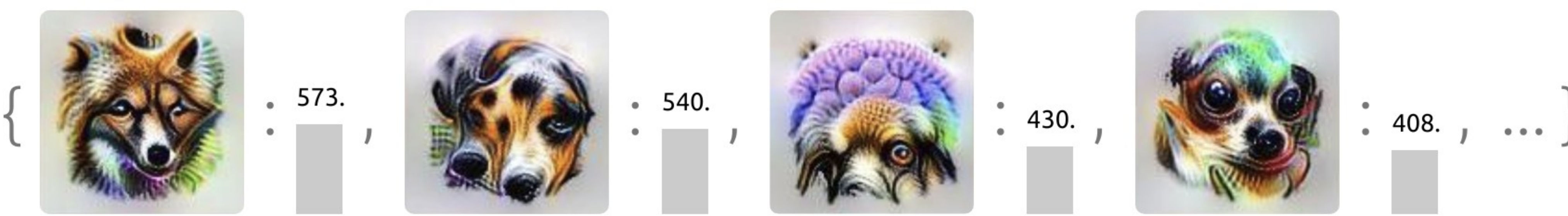
Finding the maximizing patterns for each kernel



Making sense of these activations is hard because we usually work with them as abstract vectors:

$$a_{2,4} = [0, 0, 0, 0, 31.4, 0, 0, 0, 49.0, 0, 0, 0, ...]$$

With feature visualization, however, we can transform this abstract vector into a more meaningful "semantic dictionary".



https://distill.pub/2018/building-blocks/

Deep Dream

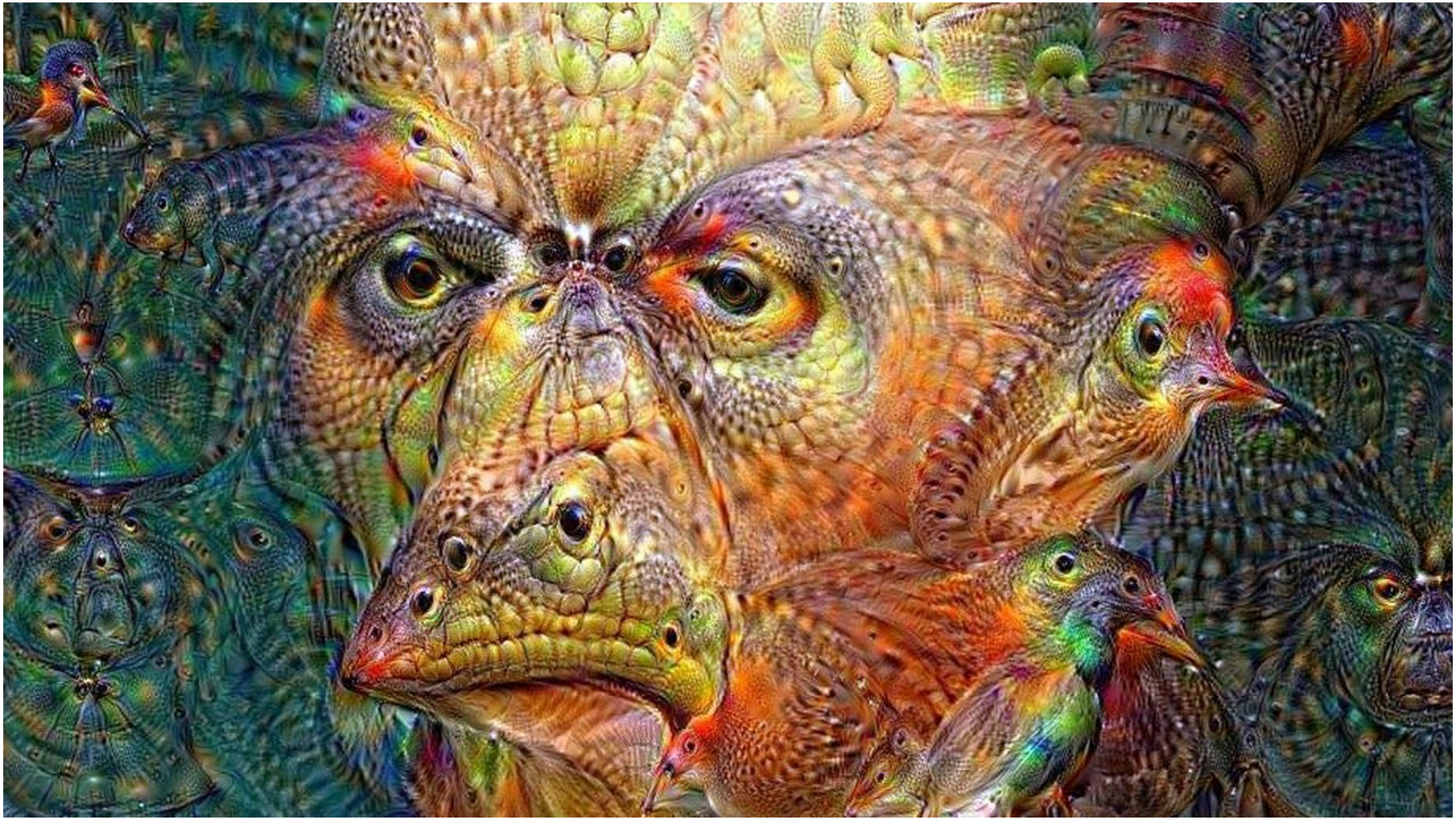
Deep dream does the same, but uses image transformation.

It amplifies, transforms existing features (noise) on the image

Deep Dream

Deep dream does the same, but uses image transformation.

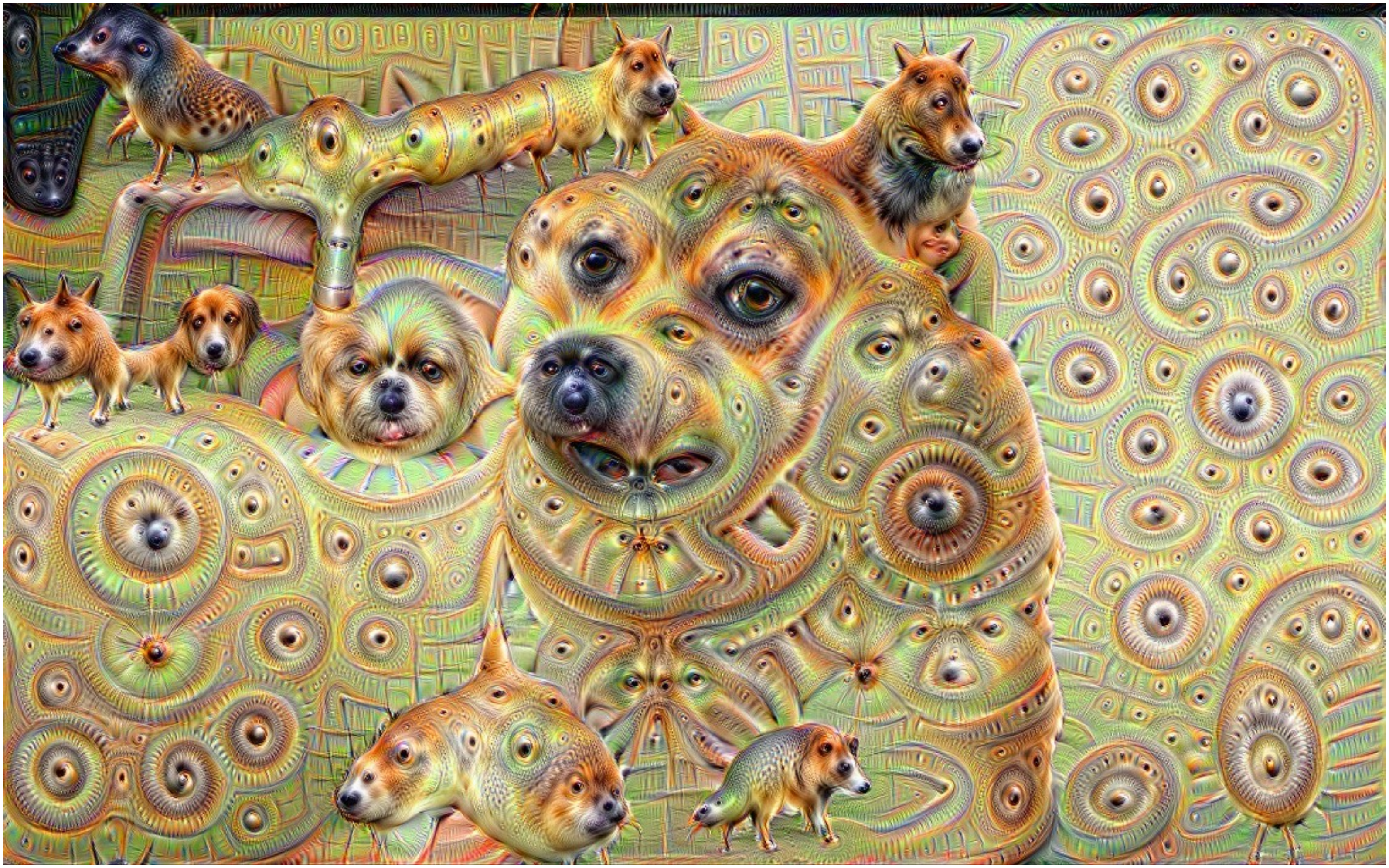
It amplifies, transforms existing features (noise) on the image



Deep Dream

Deep dream does the same, but uses image transformation.

It amplifies, transforms existing features (noise) on the image



Neural Style Transfer

An interesting application of the gradient ascent method is neural style transfer

Could we use an input image and transform it into the style of an other input image?

https://demos.algorithmia.com/deep-style/

Neural Style Transfer

Could we use an input image and transform it into the style of an other input image?

Gradient ascent transforms the image according to a loss function.

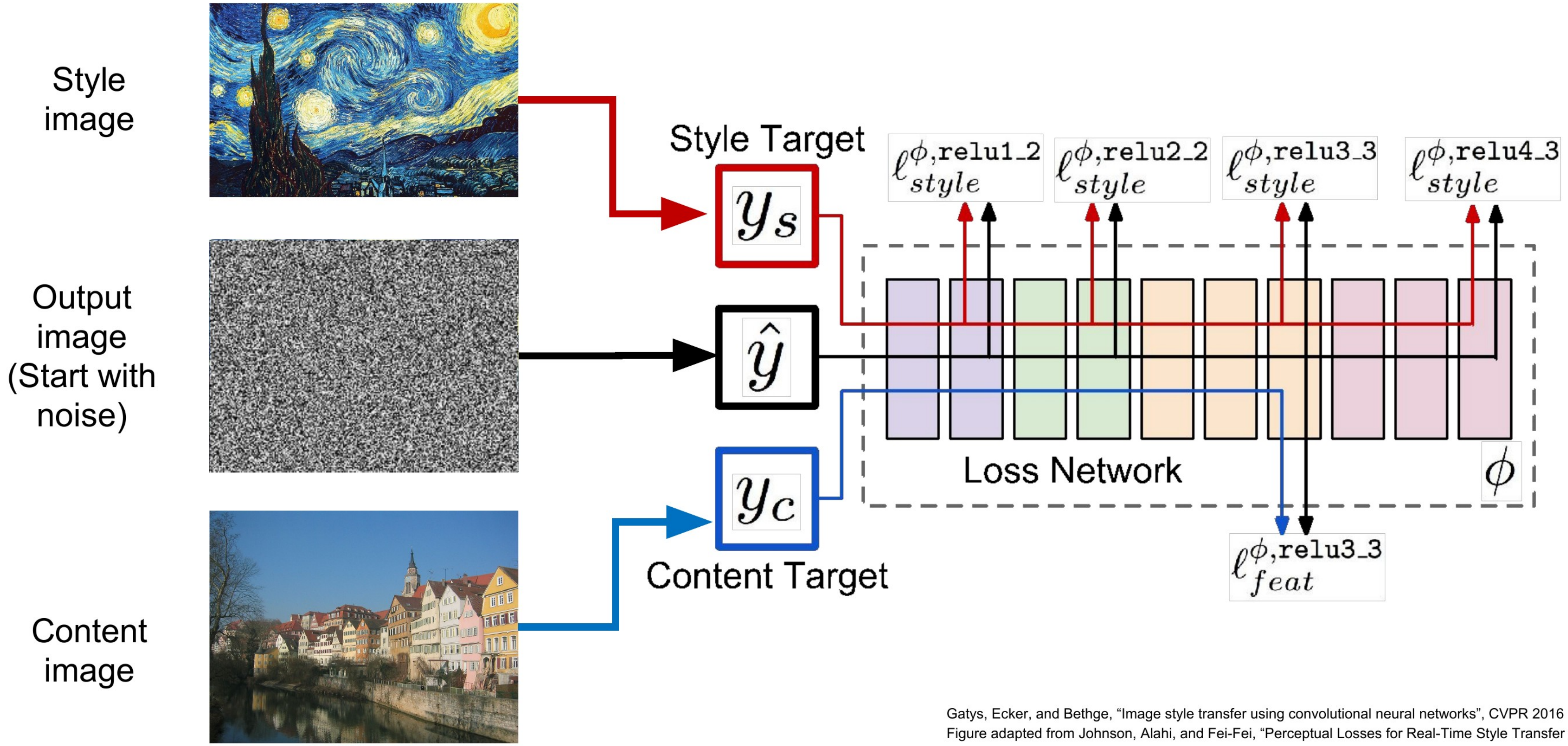
Can we find a loss function, which would preserve objects and another which preserves features connected to style?

Neural Style Transfer

https://tenso.rs/demos/fast-neural-style/

Style transfer works, but It requires a lot of time, to generate an image.

Many forward and backward passes are needed.



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

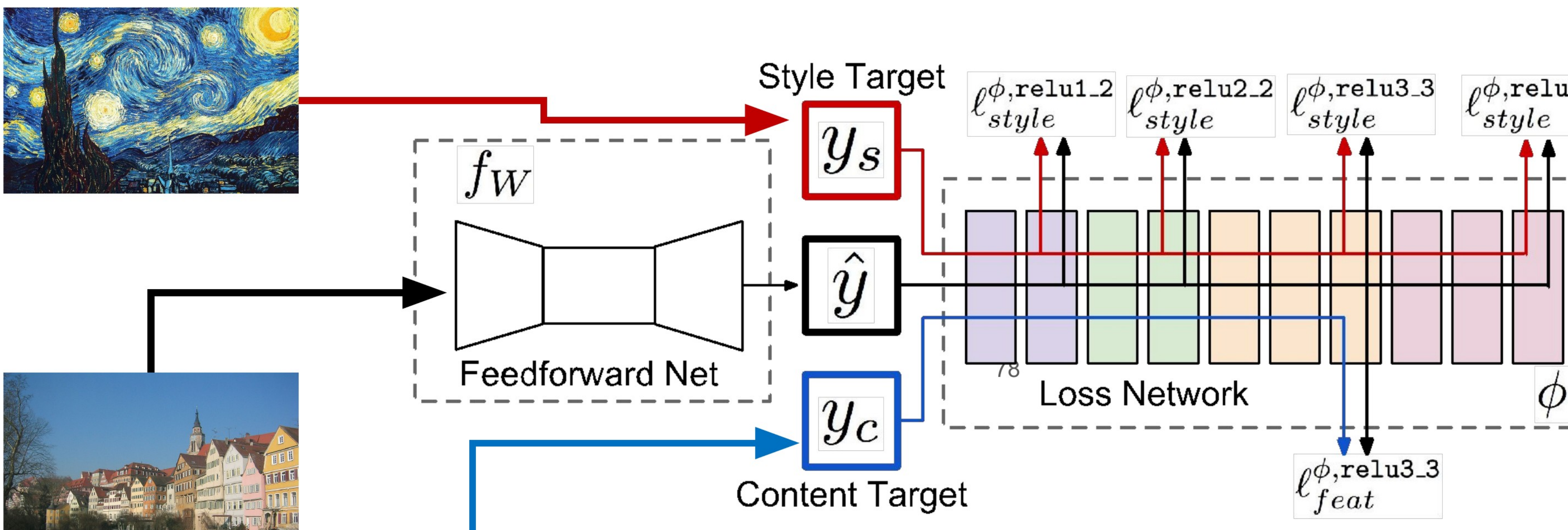
Neural Style Transfer

https://tenso.rs/demos/fast-neural-style/

Style transfer works, but It requires a lot of time, to generate an image.

Many forward and backward passes are needed.

We could train a network that learns the result of this iterative transformation, and tries to predict it. Only a single pas is needed.



Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016
Figure copyright Springer, 2016. Reproduced for educational purposes.

Neural Style Transfer

We have a loss function for content:

Can the same objects be found on both images?

Content loss, Perceptual loss: this is a distance between the two embedded image vectors in the last features layers

Style loss:

Can the same low level features, edges structures, simple patterns be found on both images

Style loss: Distances between lower level representations of the images

Neural Style Transfer

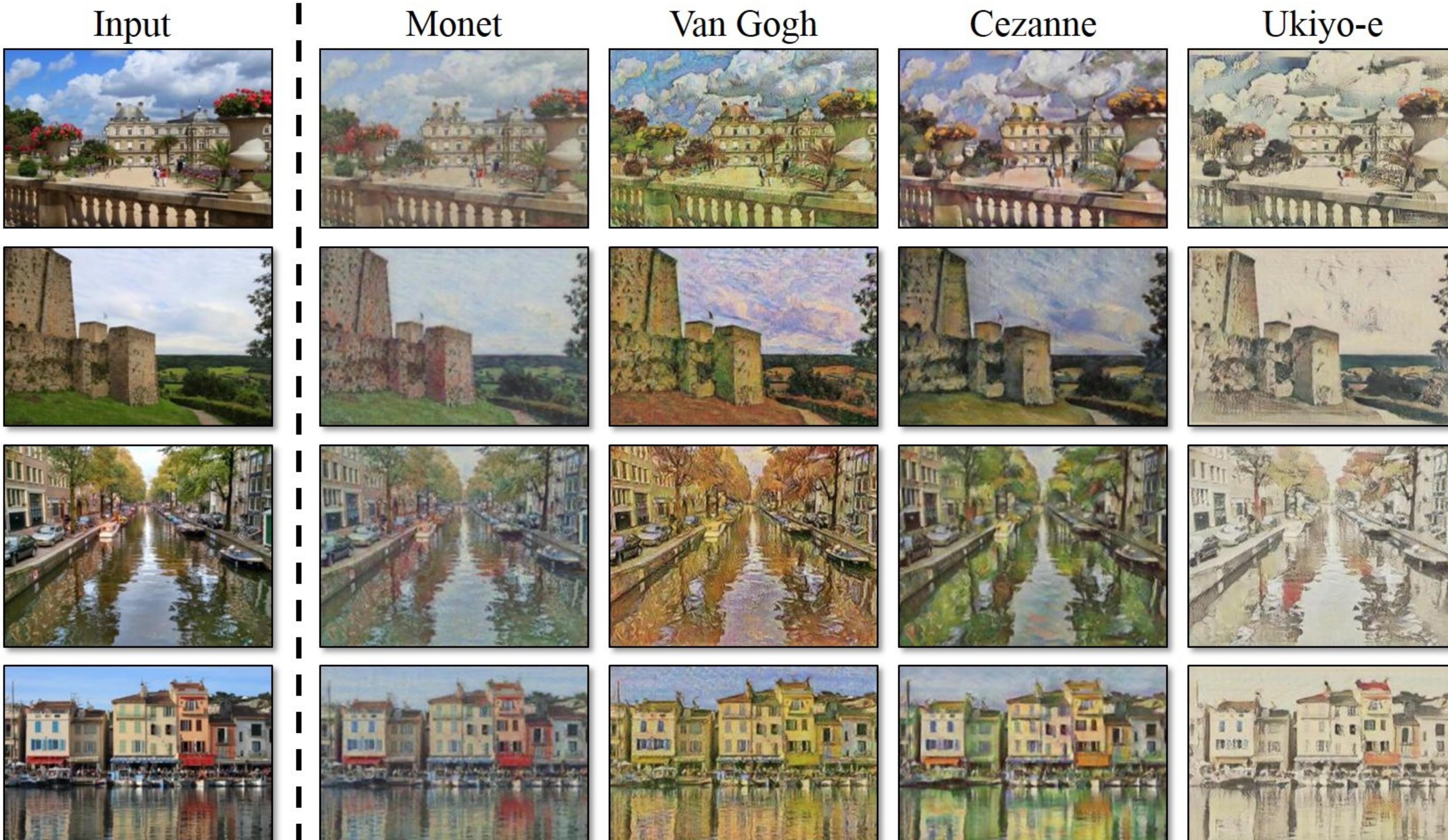
Could we use an input image and transform it into the style of an other input image?

Gradient ascent transforms the image according to a loss function.

Can we find a loss function, which would preserve objects and another which preserves features connected to style?

More weight to content loss ⟷ More weight to style loss

Neural style transfer with Cycle Consistent GANs



Fast Neural Style Transfer

Could we use an input image and transform it into the style of an other input image?

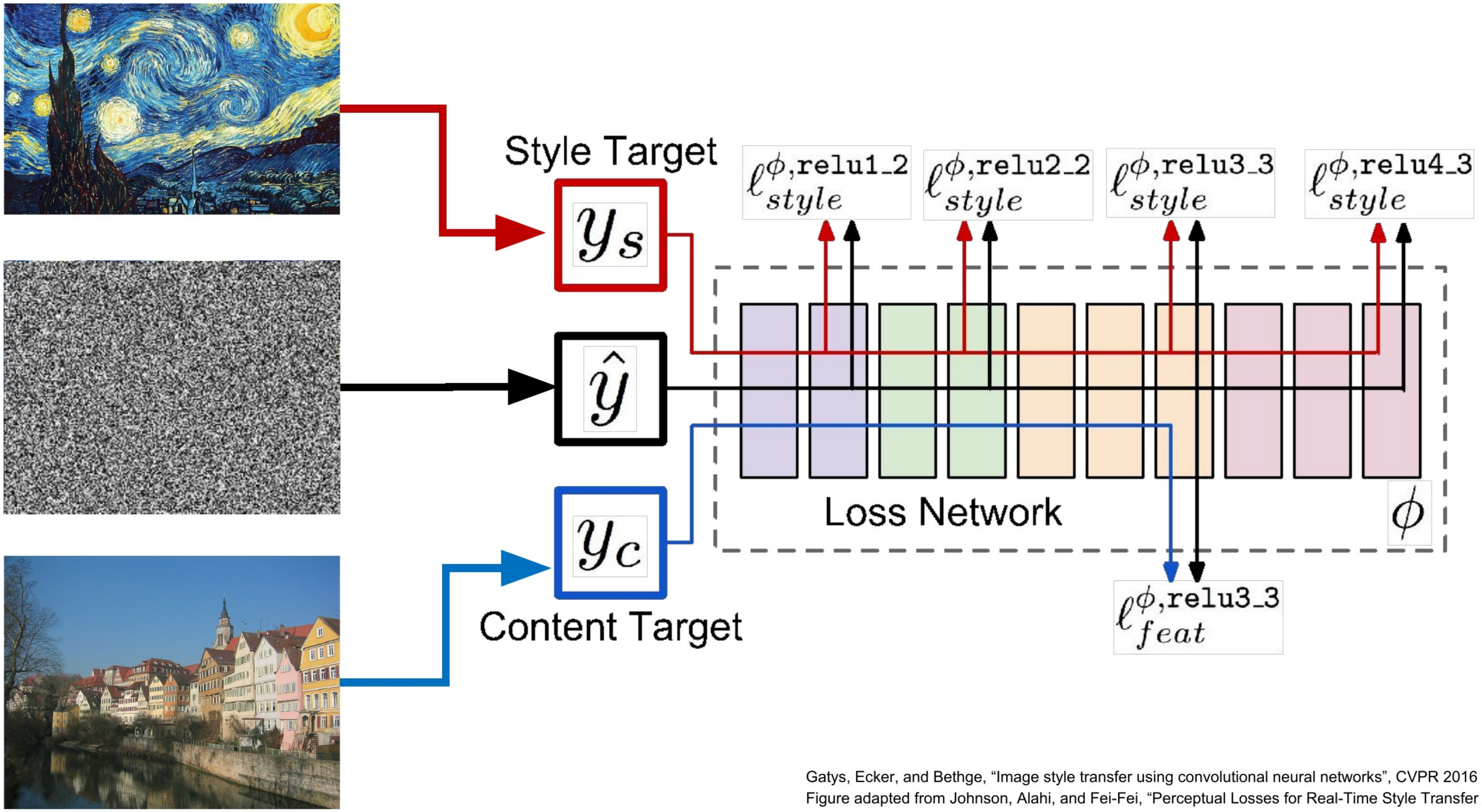
Gradient ascent transforms the image according to a loss function.

Can we find a loss function, which would preserve objects and another which preserves features connected to style?

Style image

Output image (Start with noise)

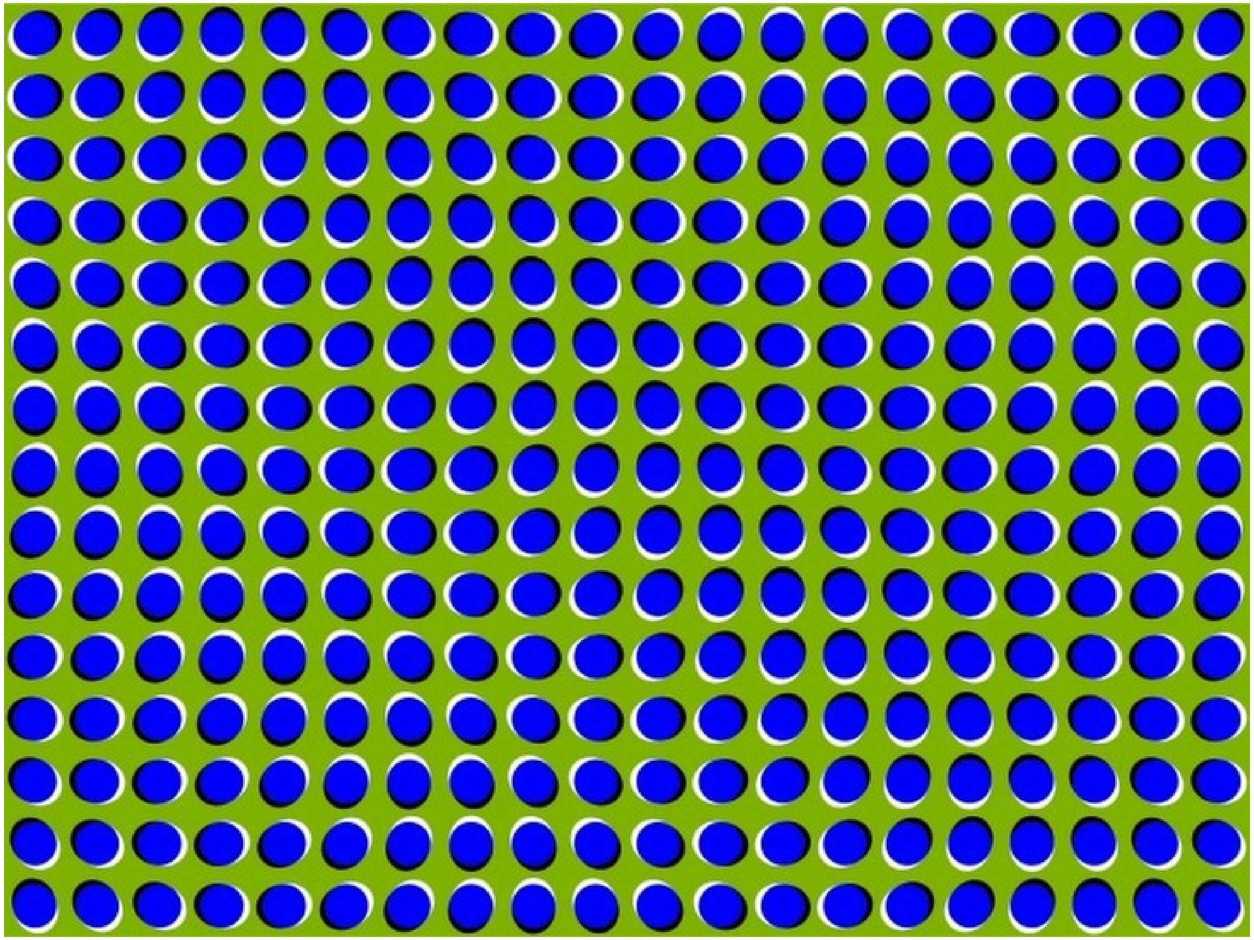
Content image



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

Adversarial Samples for Neural Networks

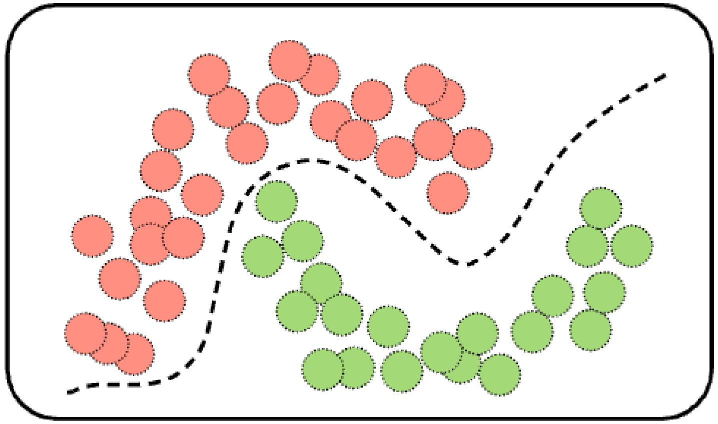
Optical Illusions for neural networks
Special, constructed elements, which can not be found in the normal input set



Adversarial attacks

We have a high number of parameters to be optimized

An even higher-dimensional input

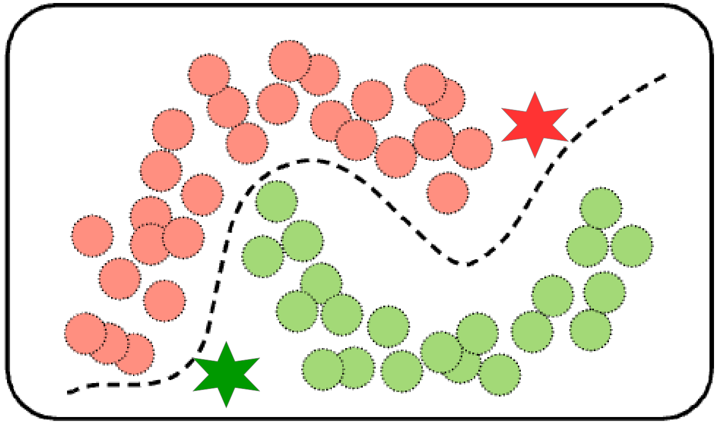


The network works well in practice, but can not cover all the possible inputs

Adversarial attacks

We have a high number of parameters to be optimized

An even higher-dimensional input

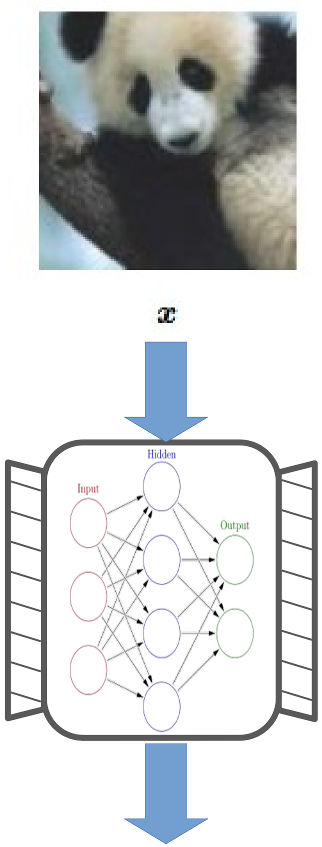


The network works well in practice, but can not cover all the possible inputs

One can exploit that there will be regions in the input domain, which were not seen during training

Adversarial noise

I have a working well-trained classifier:

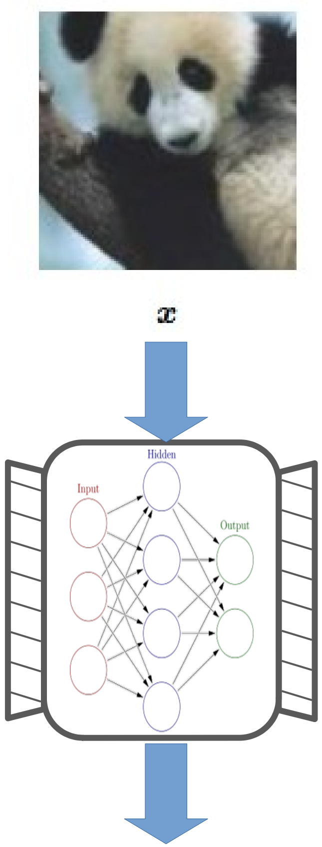


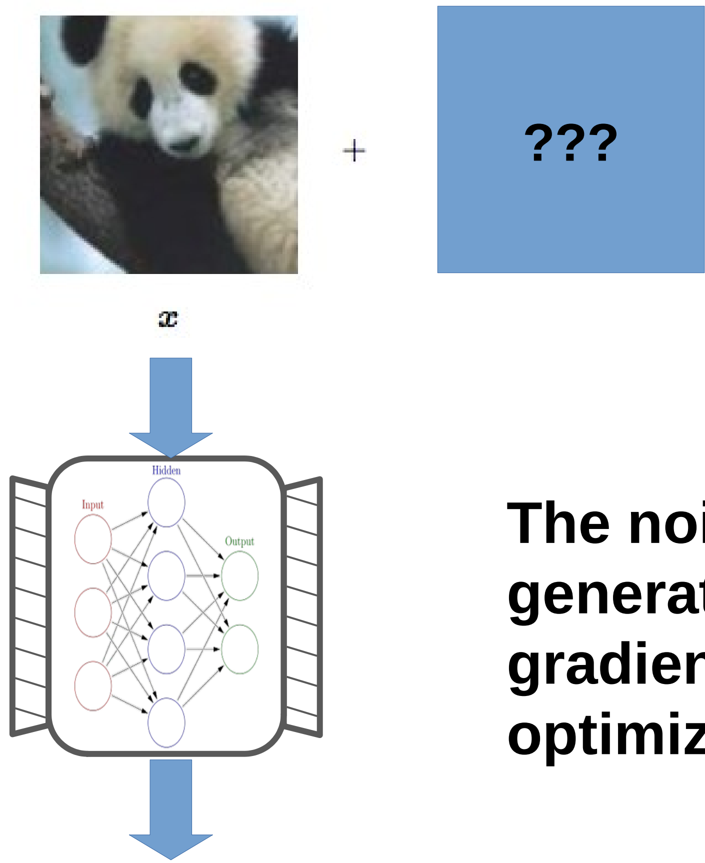
Panda

[Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessi examples. *arXiv preprint arXiv:1412.6572*

Adversarial noise

What should I add to the input to cause misclassification:





Panda

Gibbon

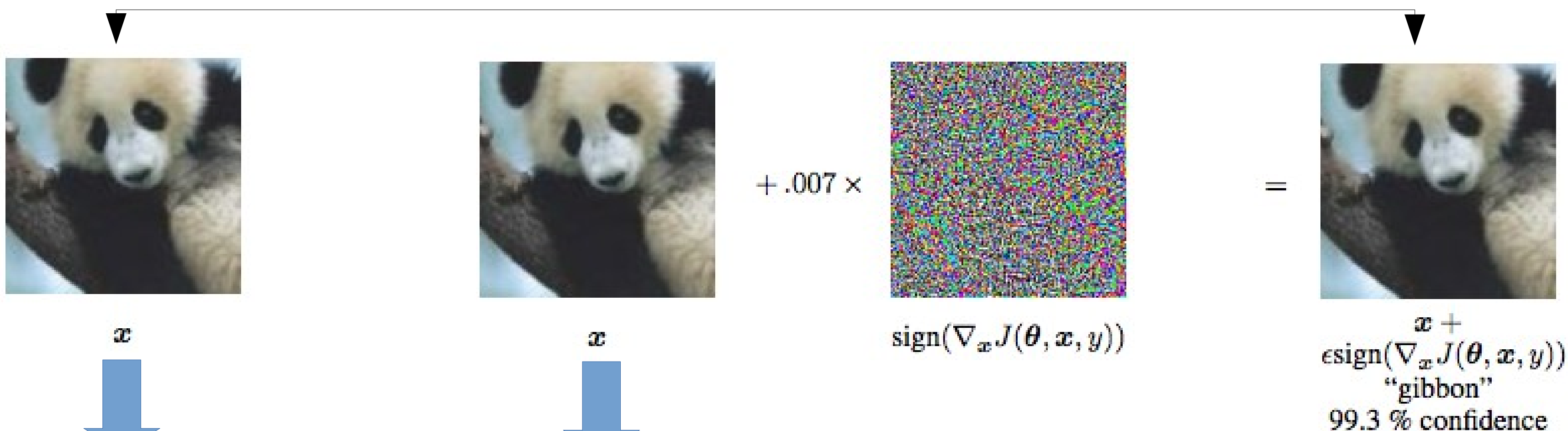
The noise is generated by gradient descent optimization

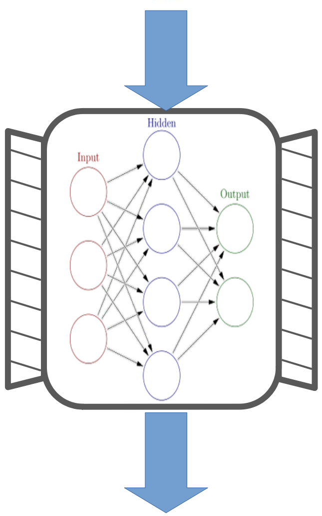
[Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessi examples. *arXiv preprint arXiv:1412.6572*

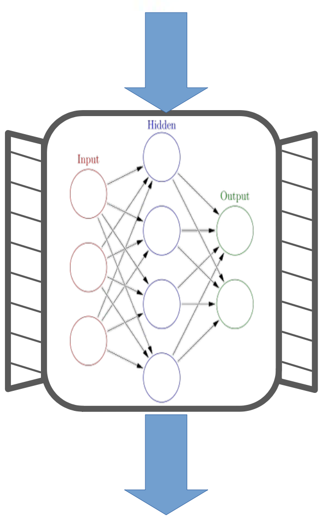
Adversarial noise

A special, low amplitude additive noise:

The two images are the same for human perception







Panda

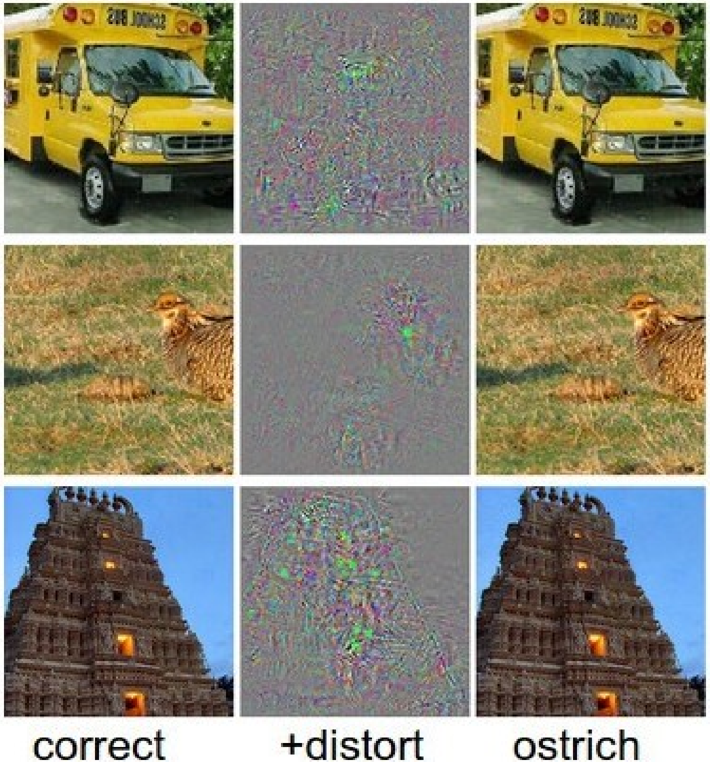
Gibbon

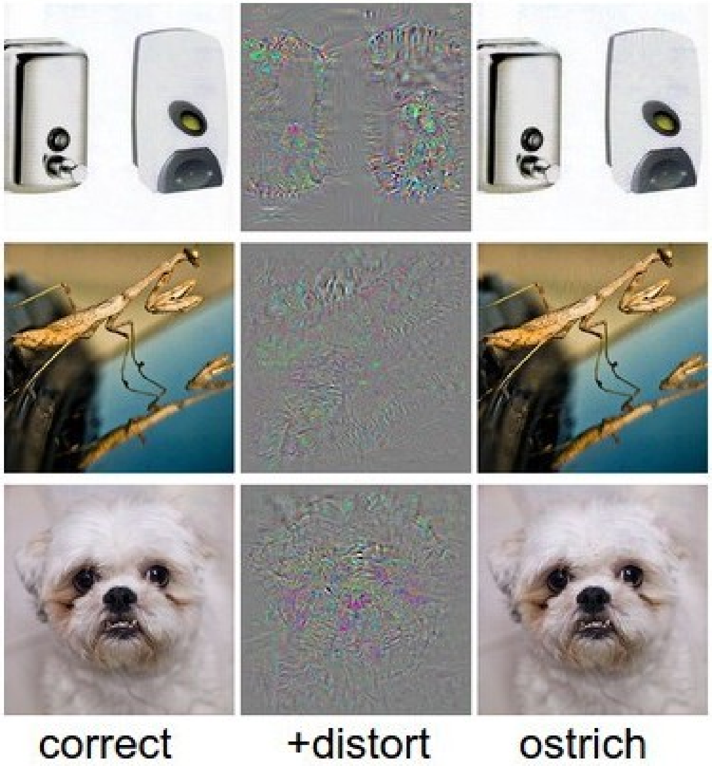
The noise is generated by gradient descent optimization

[Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessi examples. *arXiv preprint arXiv:1412.6572*

Adversarial noise

Knowing a trained network one can identify modifications (which does not happen in real life), which change the network output completely



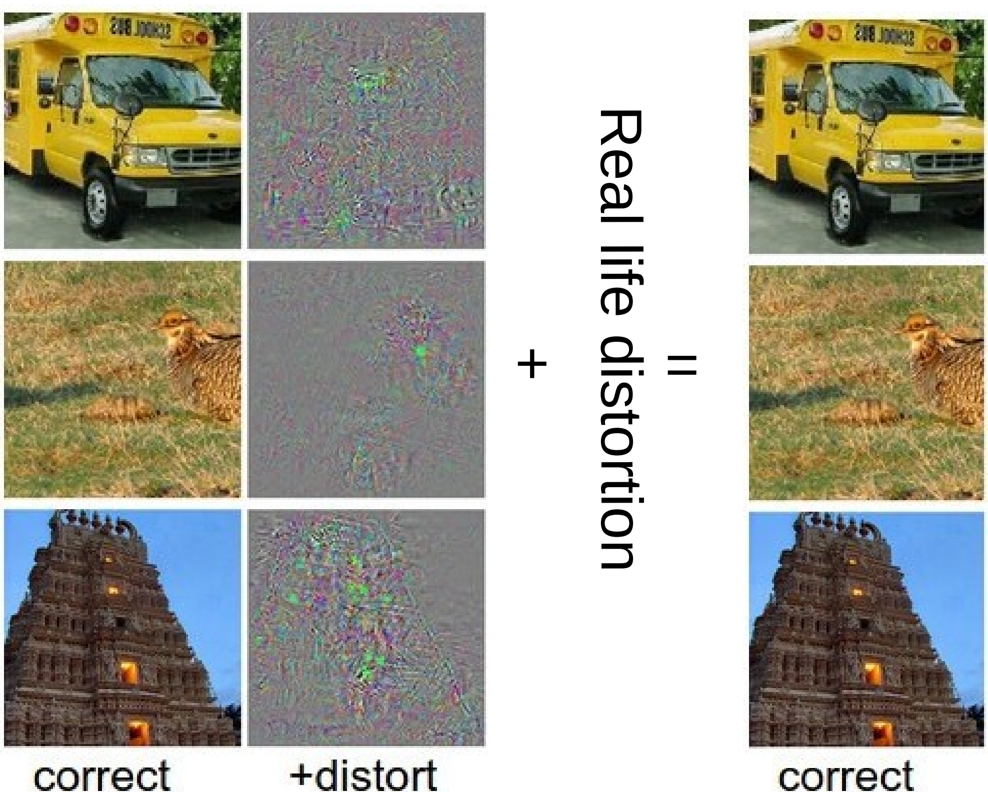


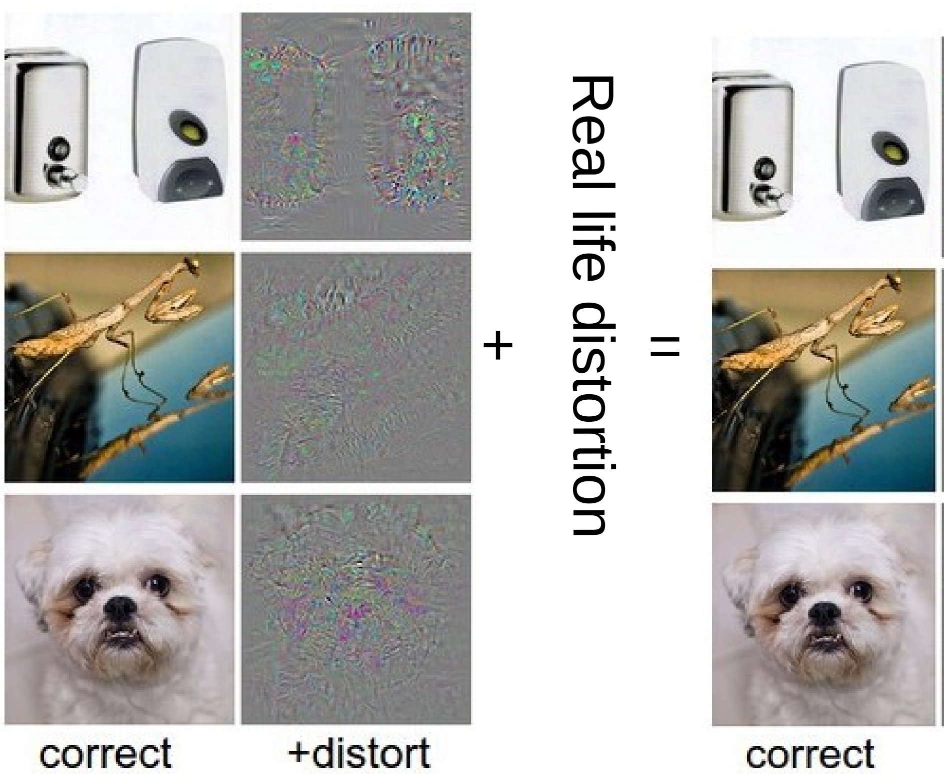
Adversarial noise – does not work in practice

Knowing a trained network one can identify modifications (which does not happen in real life), which change the network output completely

Luckily this low amplitude noise is not robust enough in real life (lens distortion and other additive noises)

Jiajun Lu, Hussein Sibai, Evan Fabry, and David Forsyth. No need to worry about adversarial examples in object detection in autonomous vehicles. 2017. URL https://arxiv.org/ abs/1707.03501.g





Sticker based adversarial attacks

High intensity noise concentrated on a small region of the input image:

$$C_d = N\left(I + \sum_{i=1}^{k} St_i\left(x_i, y_i, w_i, h_i\right) + \sum_{j=1}^{l} St_j\left(x_j, y_j, w_j, h_j\right)\right)$$

Parameters are the positions (x,y) and size (w,h) of the stickers

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Song, D., Kohno, T., ... & Tramer, F. (2017). Note on Attacking Object Detectors with Adversarial Stickers. arXiv preprint arXiv:1712.08062.

Sticker based adversarial attacks

High intensity noise concentrated on a small region of the input image:

$$C_d = N\left(I+\sum_{i=1}^{k} St_i\left(x_i,y_i,w_i,h_i\right)+\sum_{j=1}^{l} St_j\left(x_j,y_j,w_j,h_j\right)\right)$$

Parameters are the positions (x,y) and size (w,h) of the stickers

It was shown that these attacks are **robust** enough to be applied in practical applications

Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., ... & Song, D. (2017). Robust physical-world attacks on machine learning models. arXiv preprint arXiv:1707.08945.

Sticker based adversarial attacks

High intensity noise concentrated on a small region of the input image:

$$C_d = N\left(I + \sum_{i=1}^{k} St_i\left(x_i, y_i, w_i, h_i\right) + \sum_{j=1}^{l} St_j\left(x_j, y_j, w_j, h_j\right) \right)$$

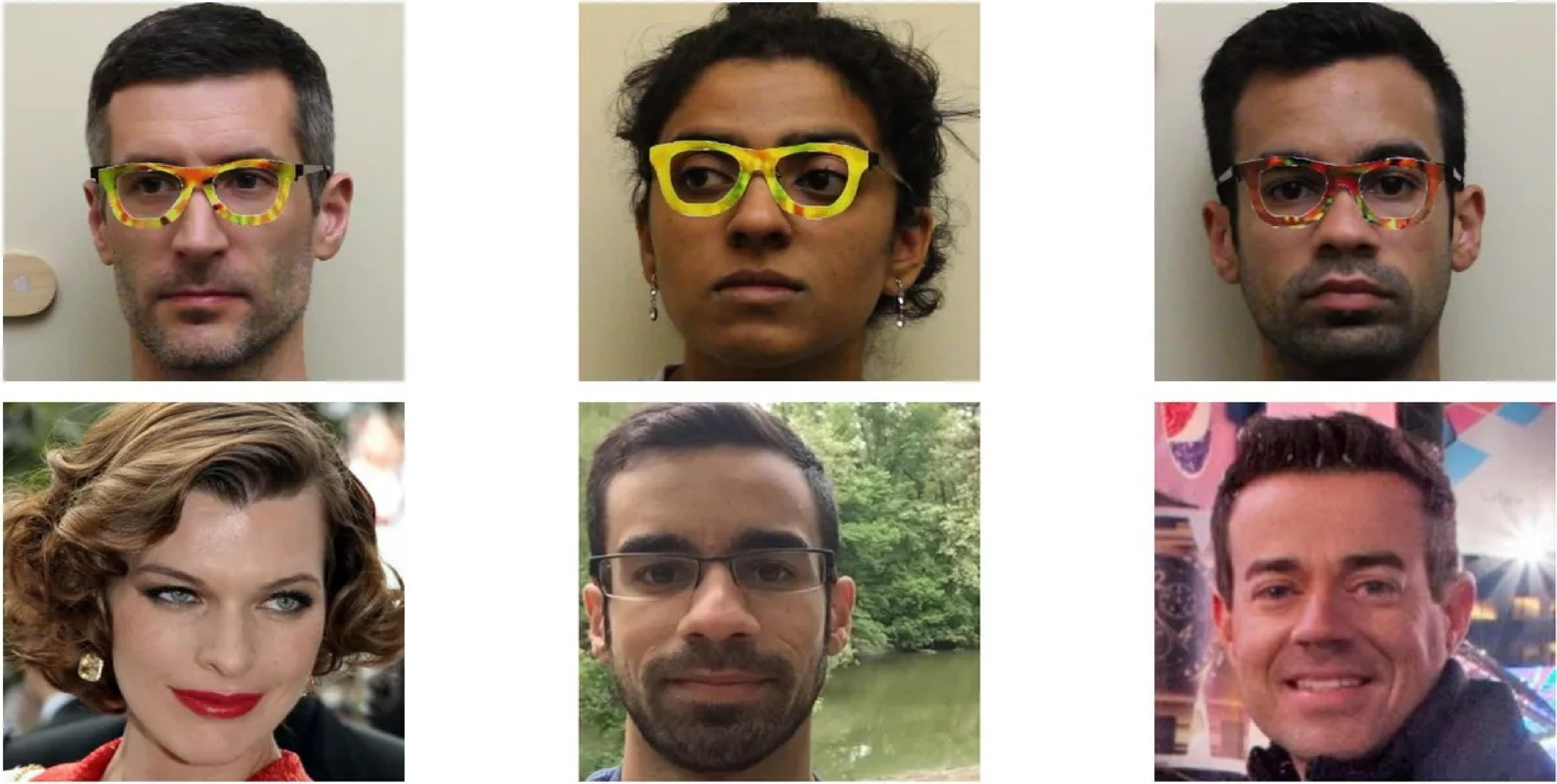
Parameters are the positions (x,y) and size (w,h) of the stickers

It was shown that these attacks are **robust** enough to be applied in practical applications

Does this mean that convolutional neural networks can not be used in critical problem in practice anymore?

Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., ... & Song, D. (2017). Robust physical-world attacks on machine learning models. arXiv preprint arXiv:1707.08945.

Sticker based adversarial attacks



(b) (c) (d)

Understanding decisions

We might be interested in case of a single sample, what triggered the decision of the network

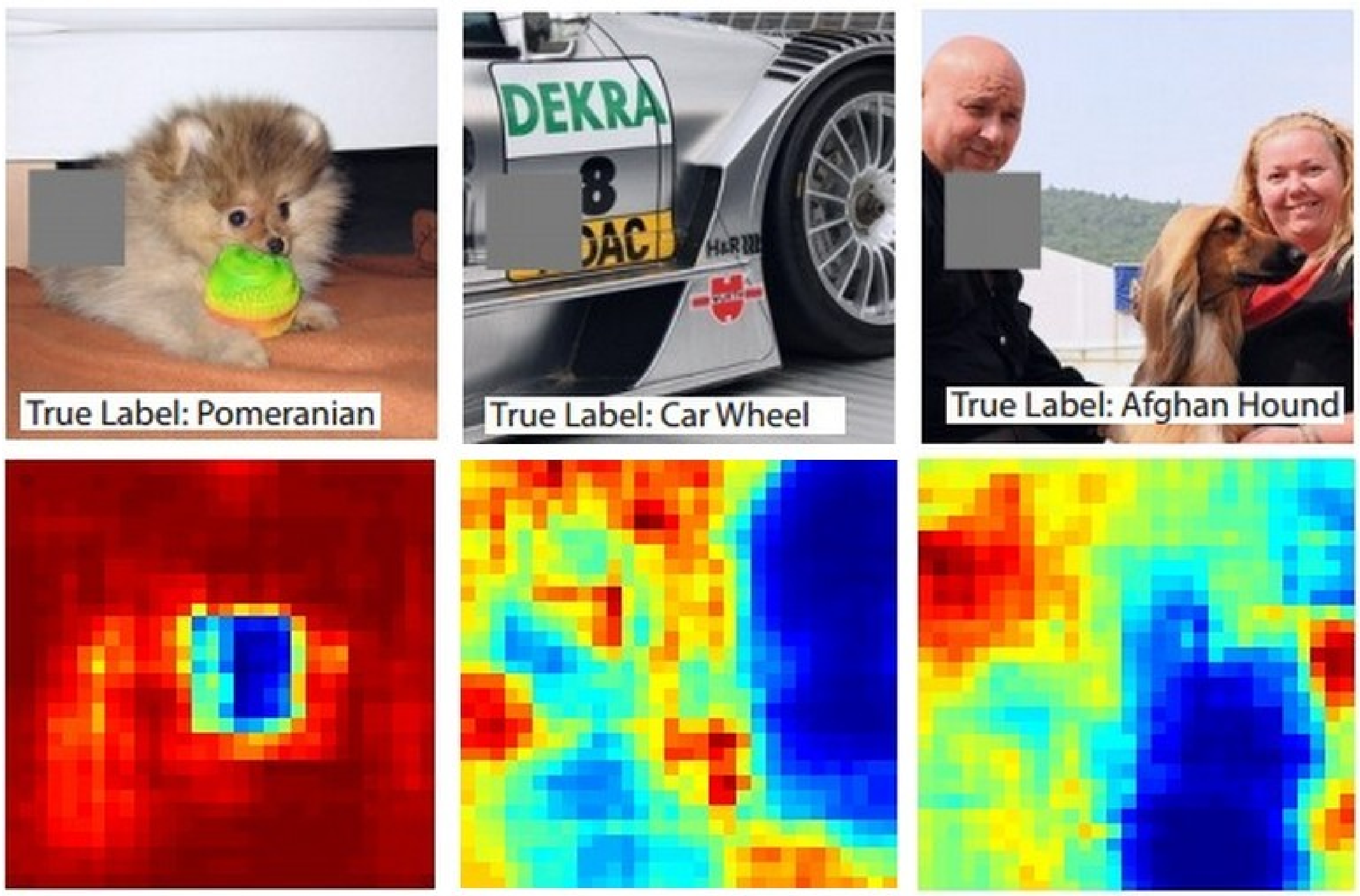
The network only outputs probabilities. Could we display why the network made this decision?

Reasoning by occlusion

We might occlude part of the input image.

If the decision does not change → the occluded part was unimportant

If the decision changes → the part was important, The importance of the part is proportional with the change



Reasoning by importance

Occlusion maps are good

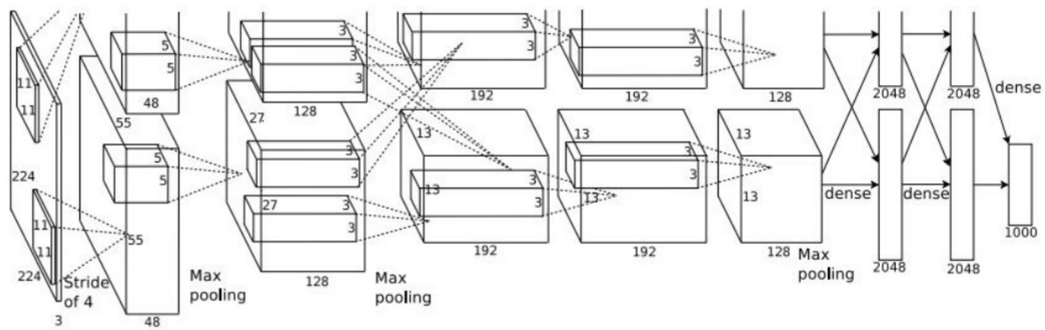
Calculating an occlusion map takes a lot of time

Could we calculate the importance of each pixel in the decision?

Reasoning by importance

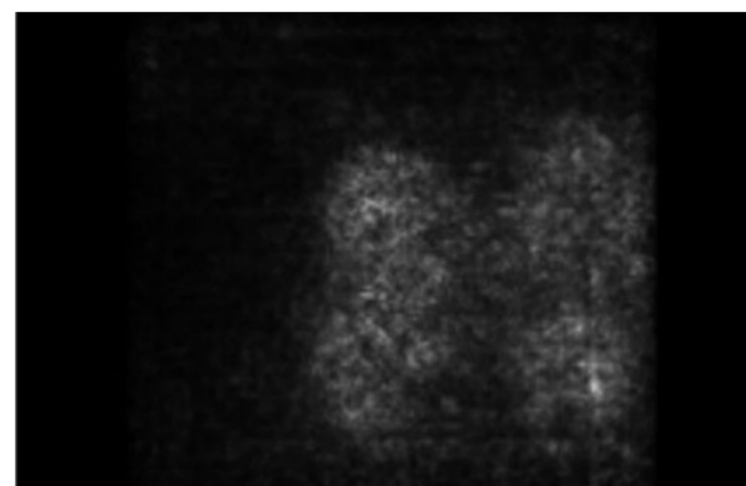
Could we calculate the importance of each pixel in the decision?

Forward pass: regular computation



Dog

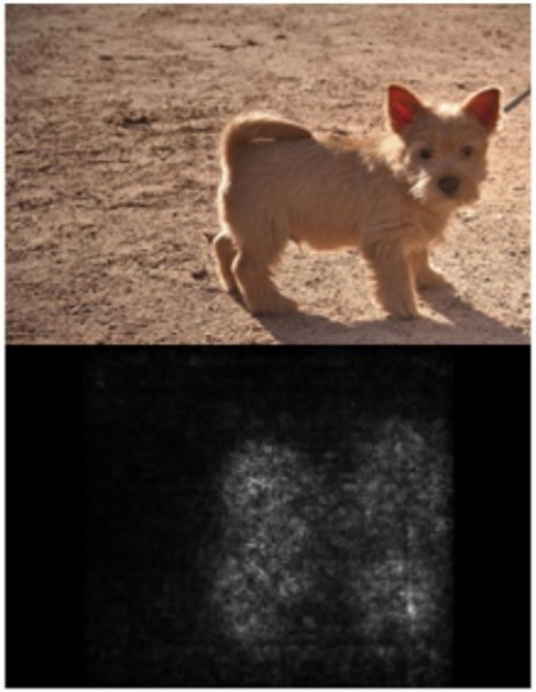
Backward pass: Computing the gradient
of (unnormalized) class score
Taking their absolute value and max over
RGB channels

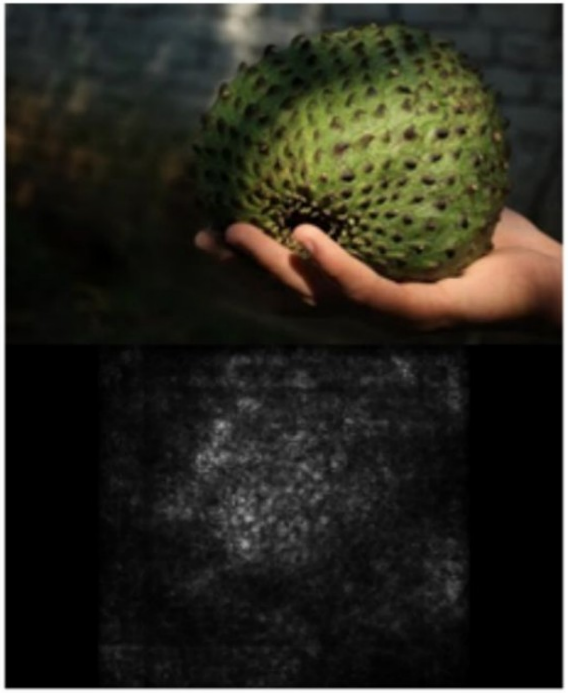


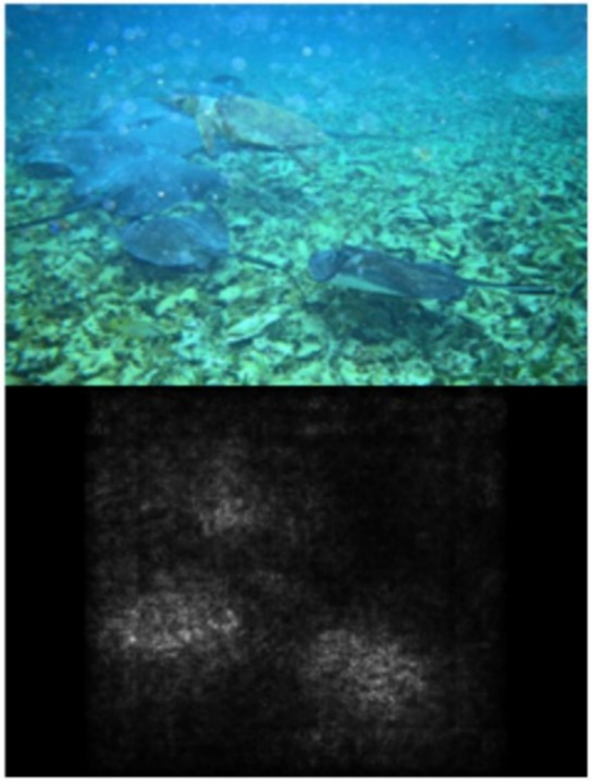
Reasoning by importance

Calculating e the importance of each pixel in the decision?

Right for the right reasons



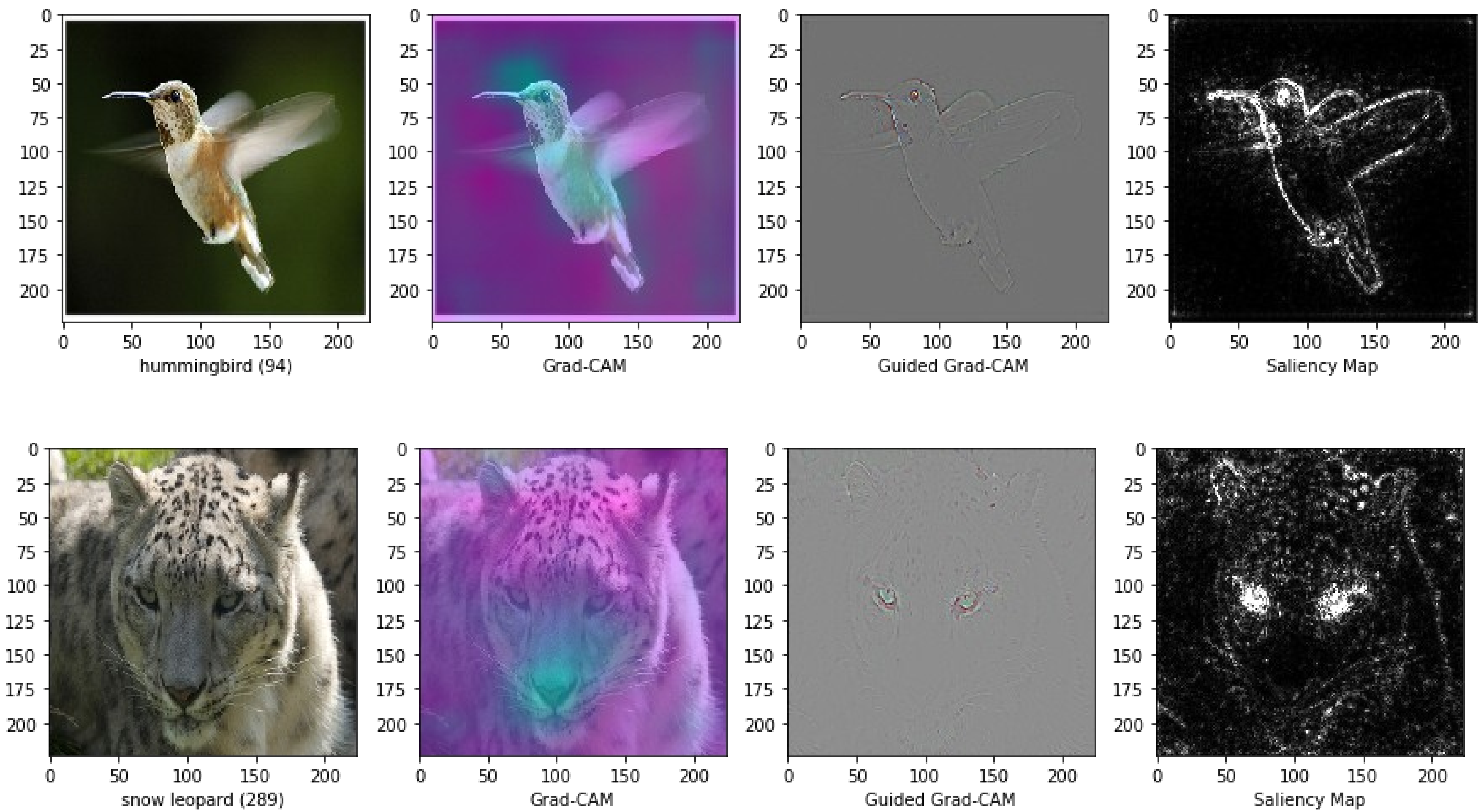




Reasoning by importance

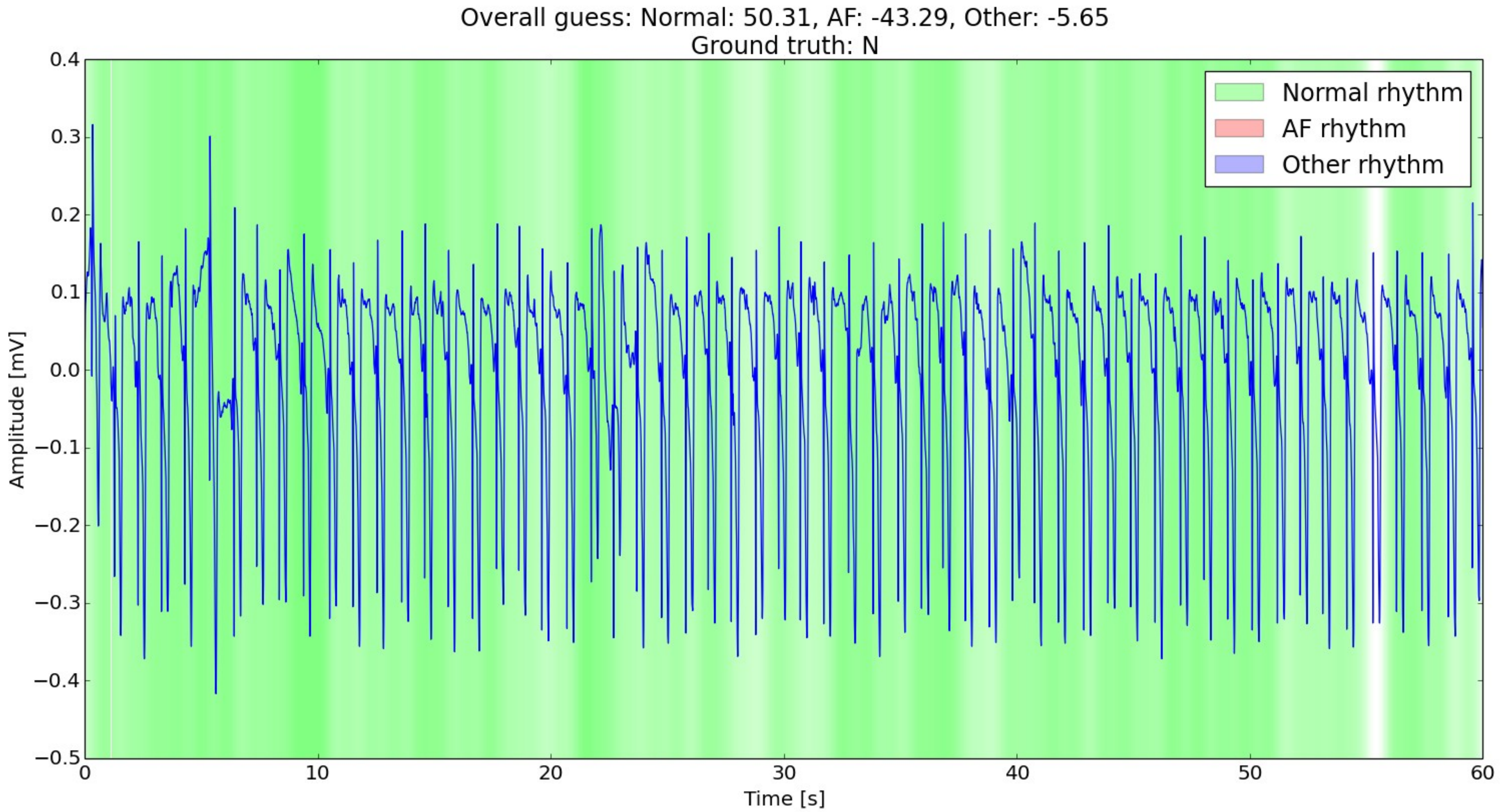
Calculating the importance of each pixel in the decision?

Right for the right reasons



Reasoning by importance – in practice

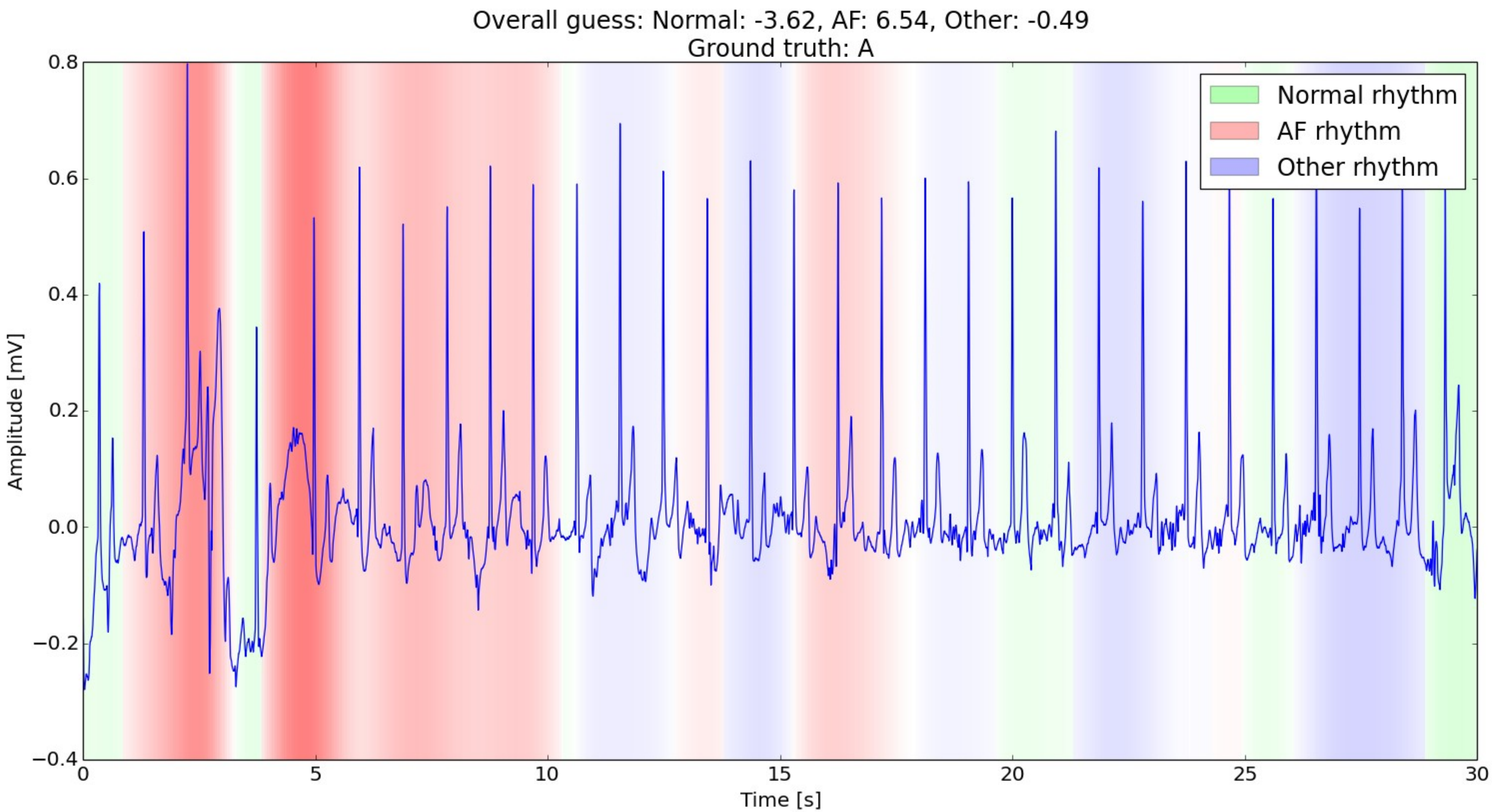
It can help people to show them why the network made such a decision



http://physionet.itk.ppke.hu/

Reasoning by importance – in practice

It can help people to show them why the network made such a decision



http://physionet.itk.ppke.hu/