

ANN: PRINCIPLES AND COMMON ARCHITECTURES

S. Giagu - 1st ML_INFN Hackathon

Online/INFN Cloud - 07.06.2021



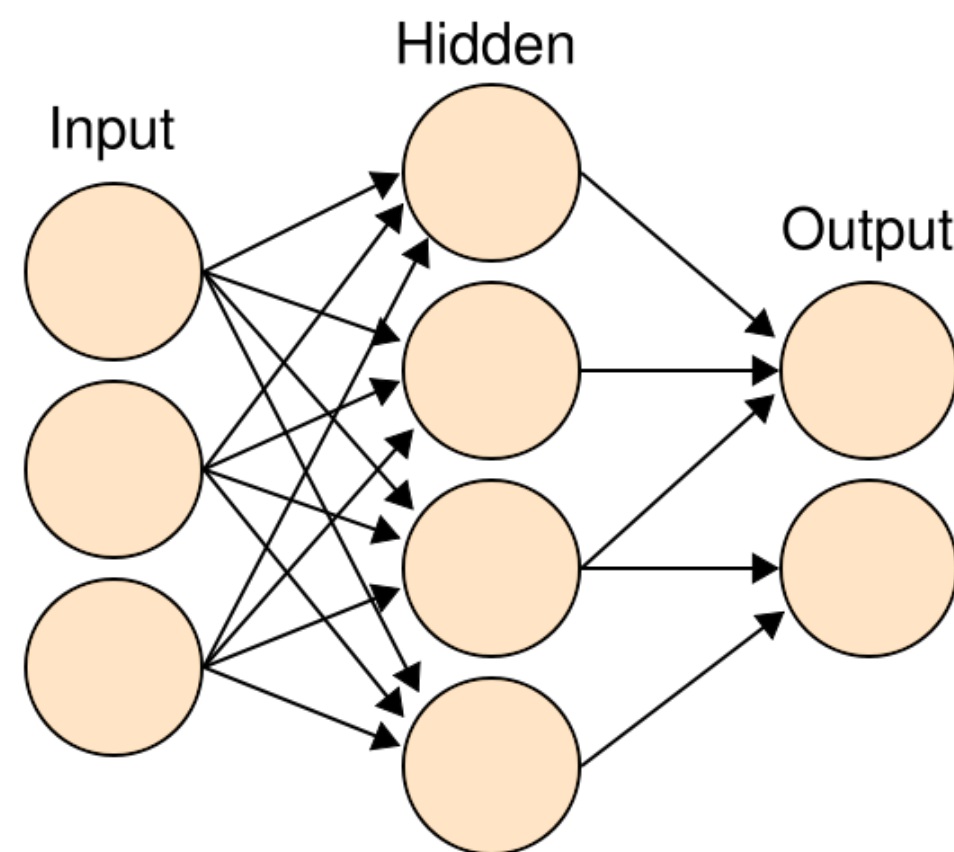
Istituto Nazionale di Fisica Nucleare

ARTIFICIAL NEURAL NETWORKS

- the most popular approach to deep learning to date
- ANN is a mathematical model able to approximate with high precision a generic functional form:

$$f : R^n \rightarrow R^m: y = f(x) \longrightarrow \text{ANN } F: \hat{y} = F(x)$$

- shallow analogy with **biological neural networks**, and in a more precise mathematical / computational language is a **composition of functions connected in chains described by graphs** (eg Feed-Forward ANN can be represented as direct acyclic graph)



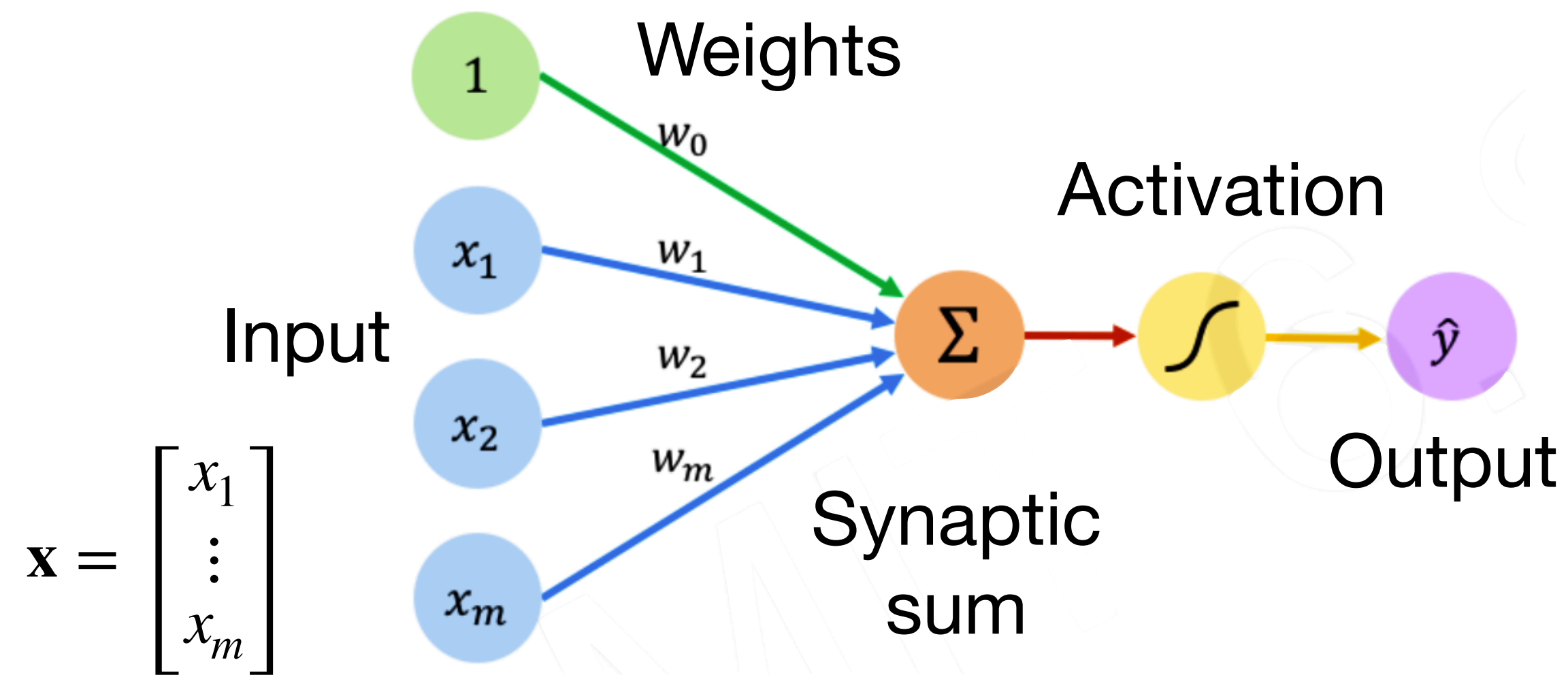
- **architecture**: interconnected group of simple identical computational units (**neurons**)
- **processing**: input information analysed according to a **connectionist computational approach**
→ collective actions performed in parallel by the neurons
- **learning**: behave as an **adaptive system**, the network structure dynamically change during the learning phase based on a set of examples that flow through the network during the training step

Properties:

- **non linear response** obtained by non linear neuron outputs
- **hierarchic representation learning** obtained by implementing complex multilayered topologies

BASIC ARTIFICIAL NEURON MODEL: TRESHOLD LINEAR UNIT

- artificial neuron (McCulloch-Pitts (1943) and Rosenblatt (1962)):



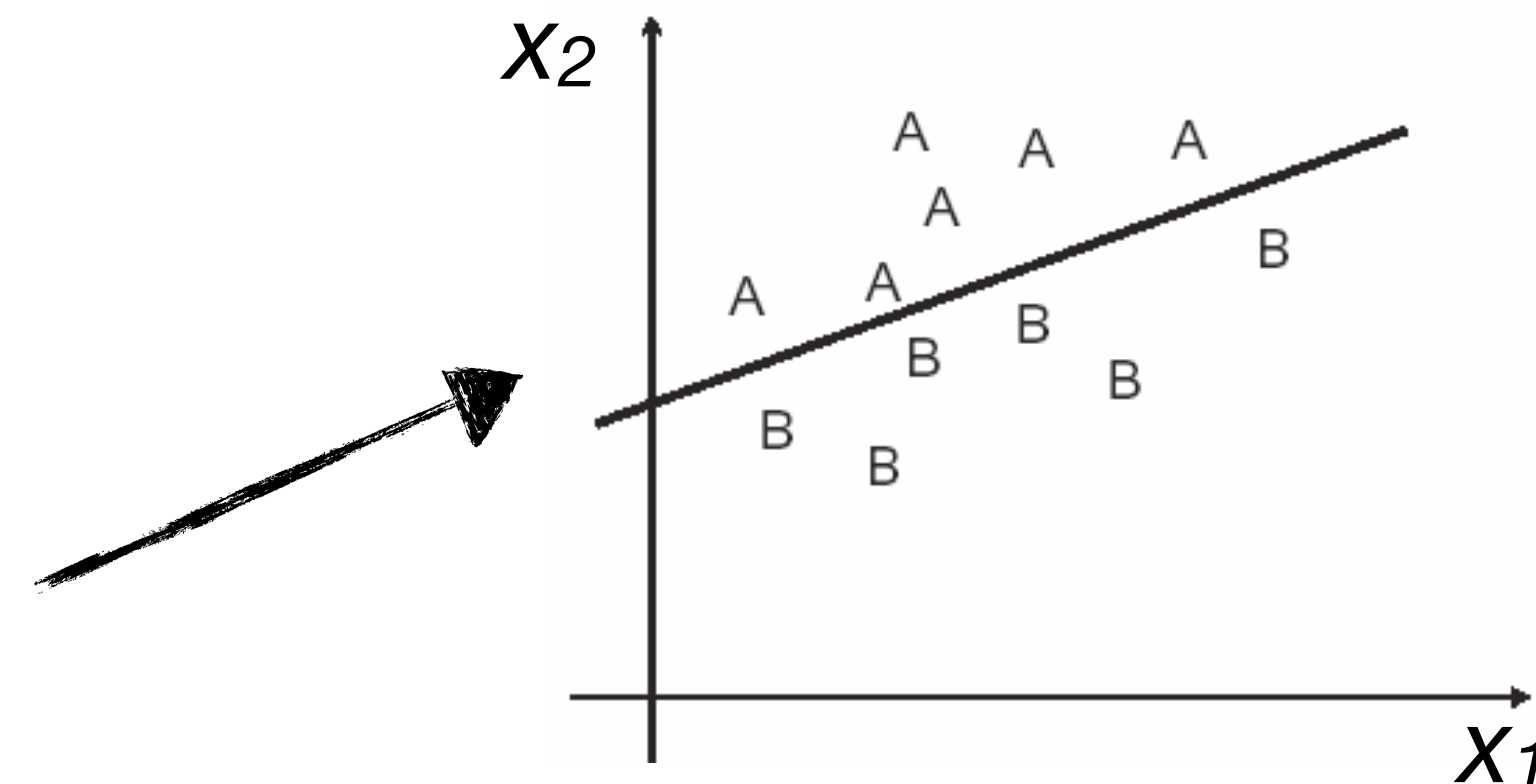
- receives in input n signals x_i and outputs y given as composition of a **synaptic function**:

$$z = w_0 + \sum_{i=1}^n w_i x_i = w_0 + \mathbf{w}^t \mathbf{x} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

- and an **activation function** (for example a step function): $a(z) = \begin{cases} 1 & \text{if } \mathbf{w}^t \mathbf{z} \geq -w_0 \\ 0 & \text{if } \mathbf{w}^t \mathbf{z} < -w_0 \end{cases}$

$$\hat{y} = a(z) = a(w_0 + \mathbf{x}^t \mathbf{w})$$

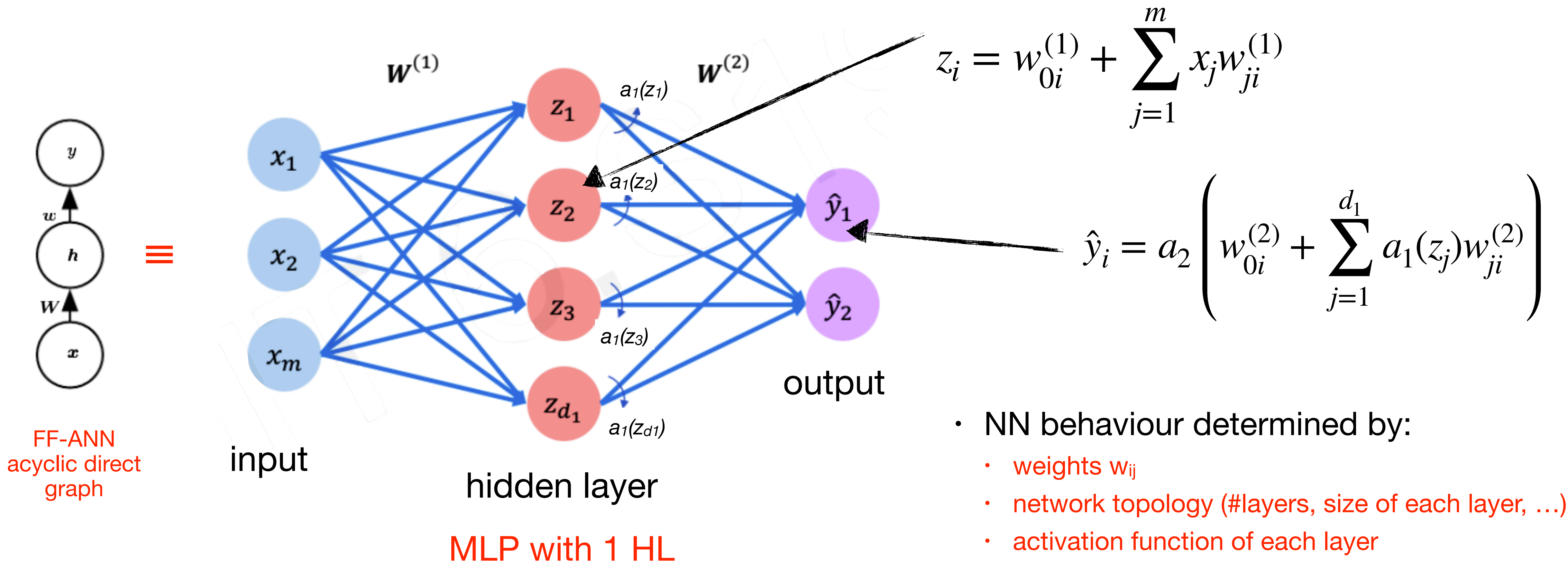
a single layer of TLU with step activation can only learn to solve problems with linearly separable classes



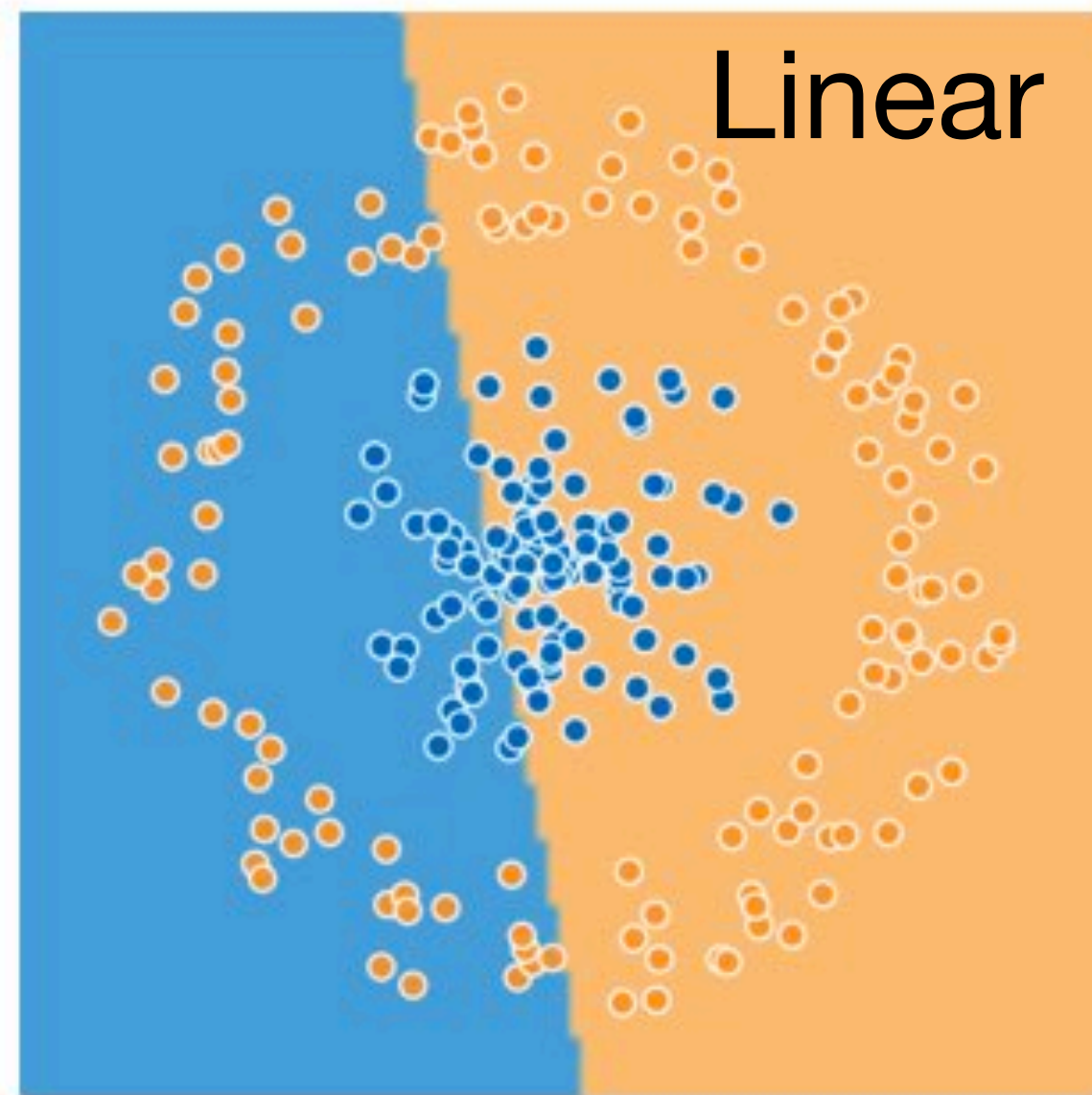
extension to multilayers with non-linear activations allows to effectively learn complex hypersurfaces 3

MULTILAYER PERCEPTRONS OR FEED-FORWARD NN

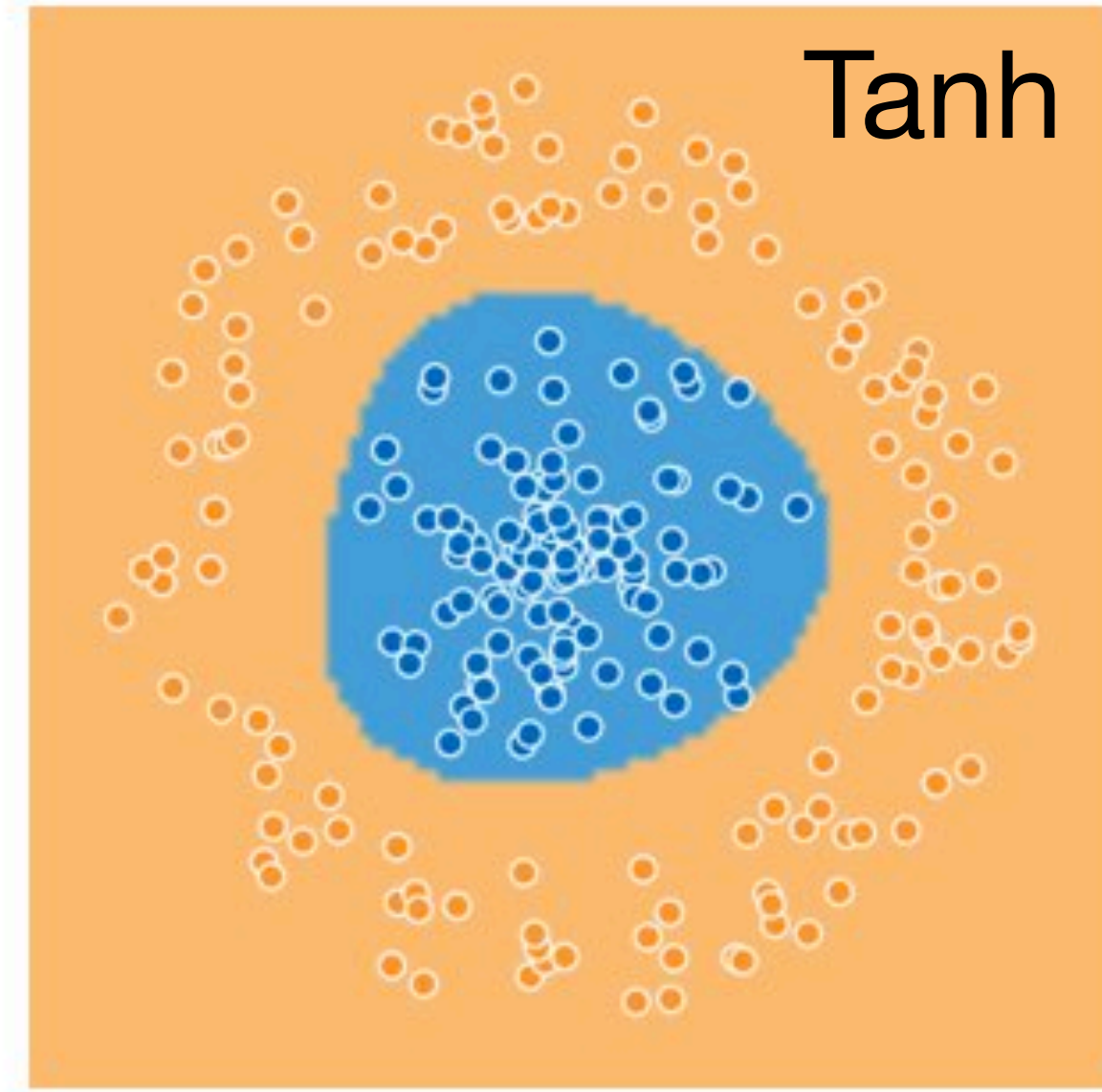
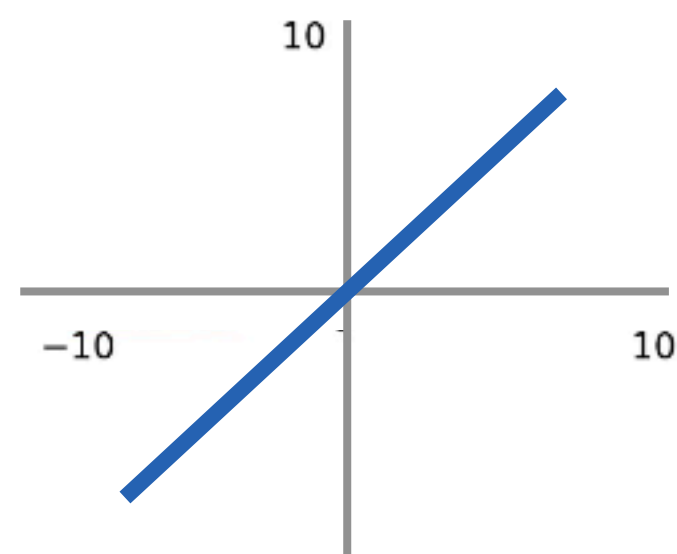
- one of the most used ANN architectures is the so called **Feed-Forward NN**
 - neurons organised in layers: **input, hidden-1, ... , hidden-K, output**
 - possible only connections of neurons of a given layer towards the next: **acyclic direct graph**
 - all possible connections are present (**dense layers**)



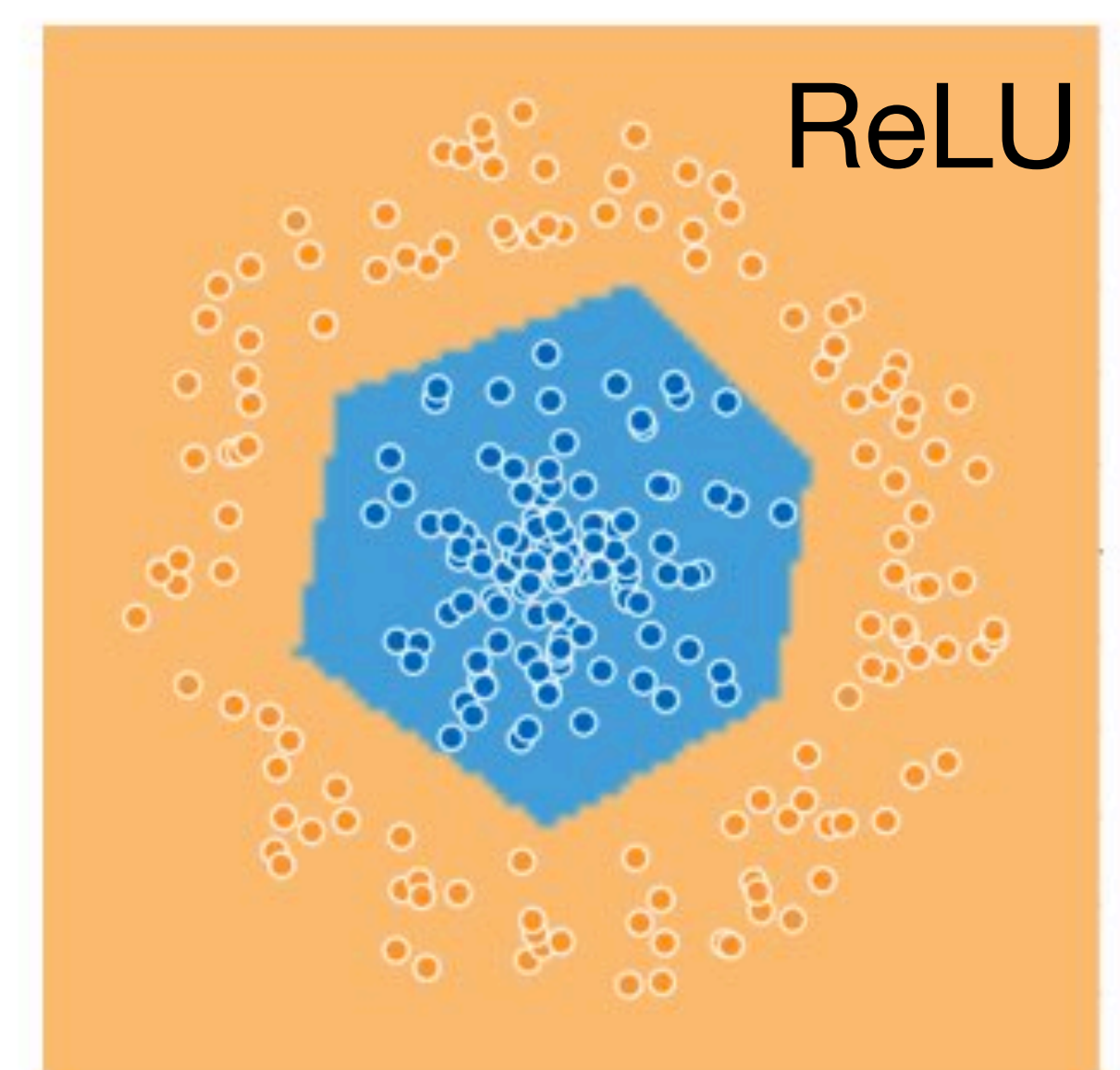
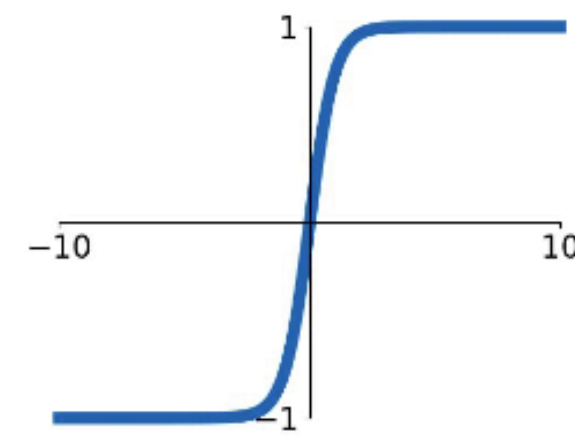
non-linear activations allows to learn complex and non linear patterns ...



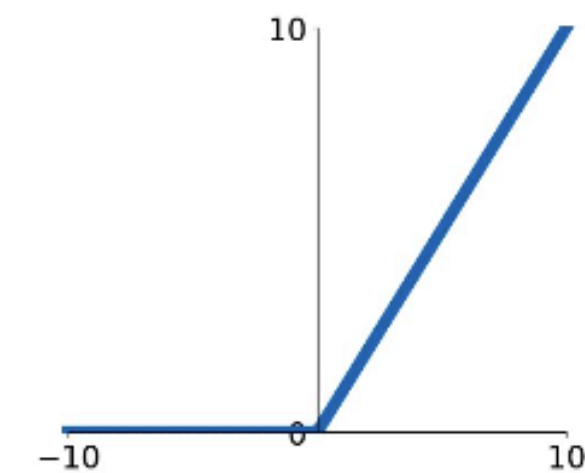
$$a(z) = z$$



$$a(z) = \tanh[z]$$



$$a(z) = \max[0, z]$$



ANN ARE UNIVERSAL APPROXIMATORS

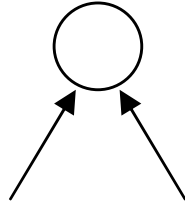
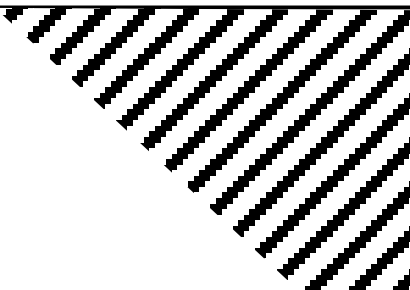
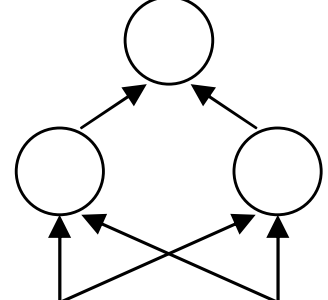
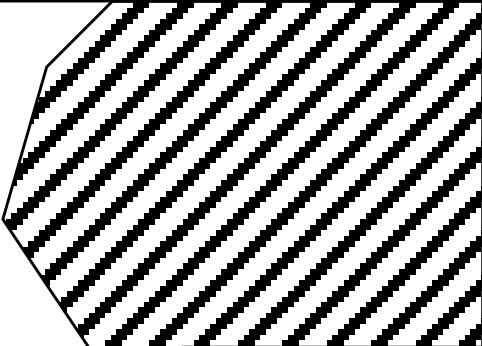
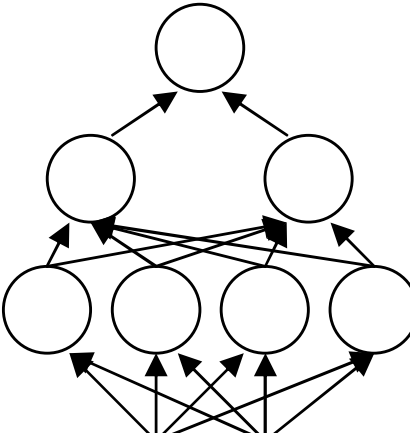
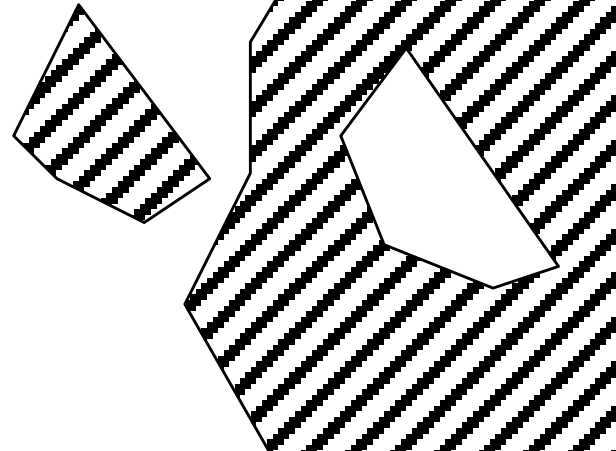
it can be demonstrated that a feed-forward network with a single hidden layer containing a finite number of neurons with non linear activations can approximate continuous functions on compact subsets of \mathbb{R}^n , under mild assumptions on the activation function

$$F(x) = \sum c_i a(w_{0i} + \mathbf{w}^t \mathbf{x})$$

$$\int_{\mathbb{R}^n} ||f(x) - F(x)||_p dx < \epsilon$$

Universal approximation theorem proof:

- unbounded, sigmoid: [here](#)
- bounded, ReLU, arbitrary depth: [here](#)

Structure	Decision regions	Shapes
	sub-spaced delimited by hyperplanes	
	convex regions	
	arbitrary shaped regions	

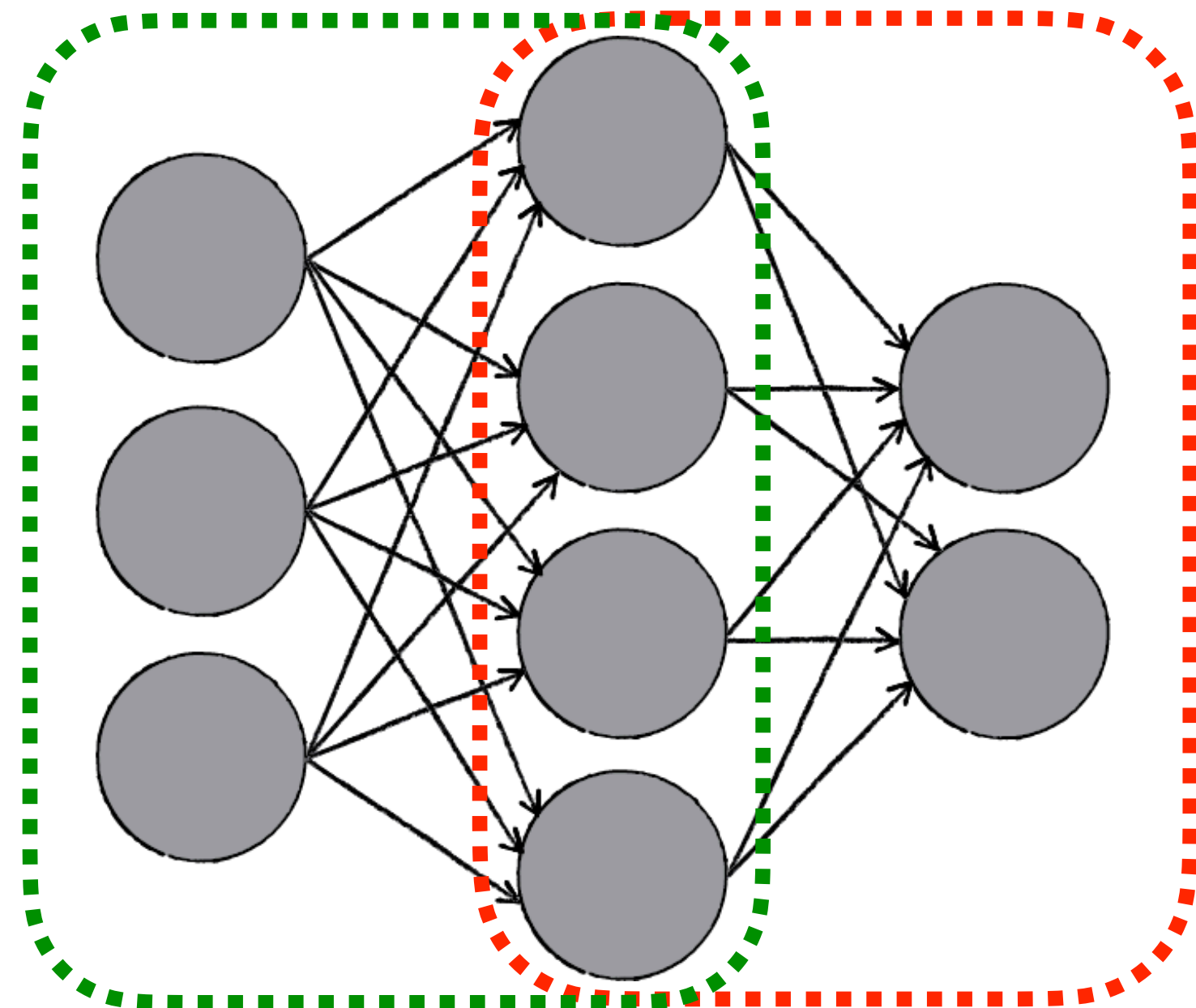
IMPORTANT: the theorem doesn't say nothing about the effective possibility to learn in a simple way the parameters of the model, all the DNN practice boils down in finding optimal and efficient techniques to do that ...

ANN AS NON LINEAR MAPPING ALGORITHM

- An ANN with non linear activations can be thought as an algorithm that learn two tasks at the same time:

LEARN A NON LINEAR
MAPPING OF THE INPUT

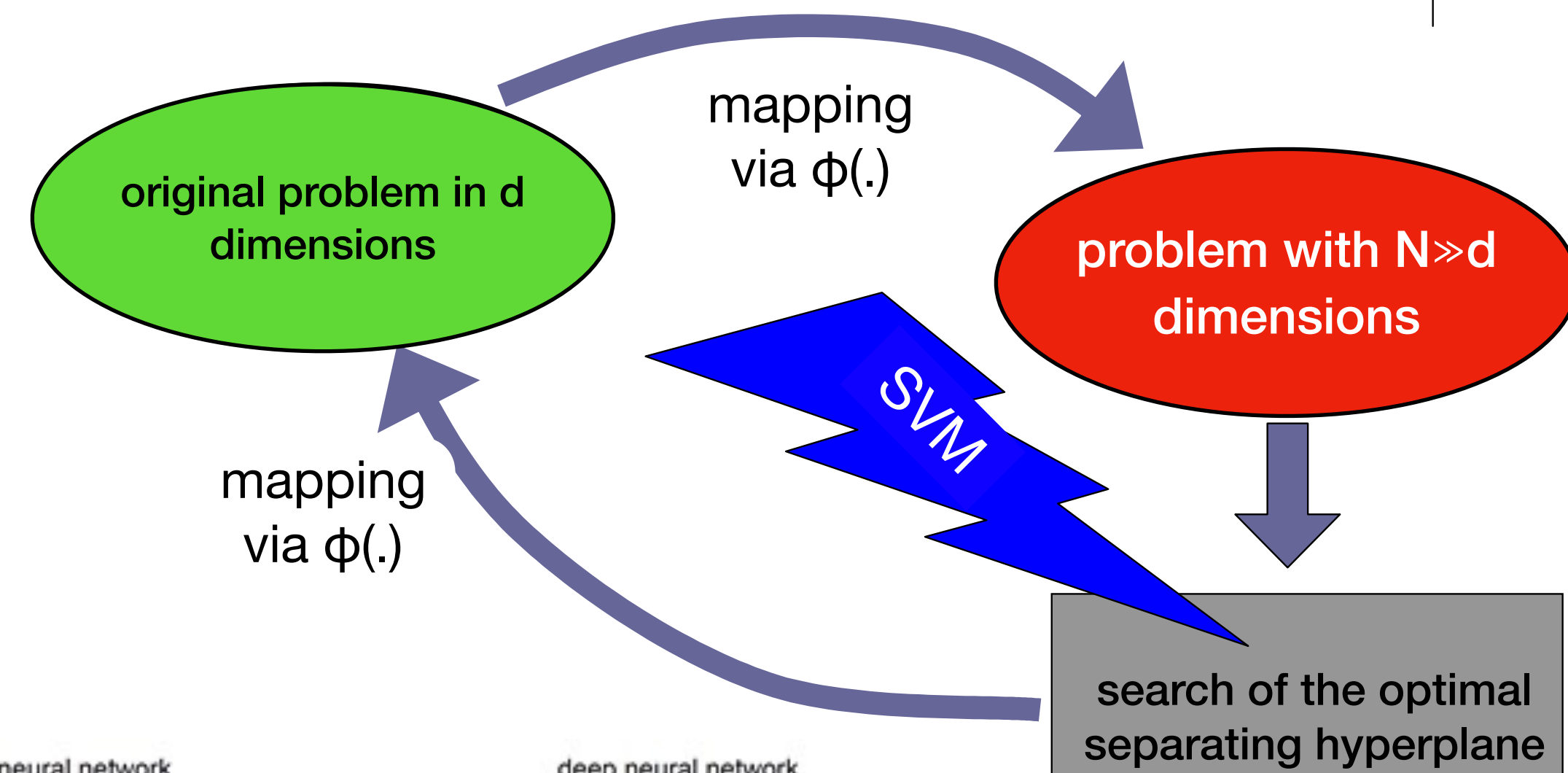
LEARN A (LINEAR) MAPPING
BETWEEN LATENT
REPRESENTATION AND TARGET



a similar approach as in other
classical ML techniques: like SVM

$$\phi: \mathbb{R}^d \rightarrow \mathbb{R}^\infty$$

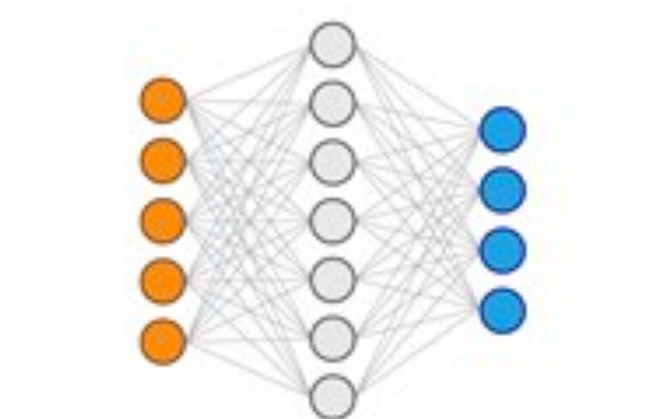
$$g(\mathbf{x}) = \mathbf{w}^t \phi(\mathbf{x}) + w_0$$



evolution of this approach: Deep-NN

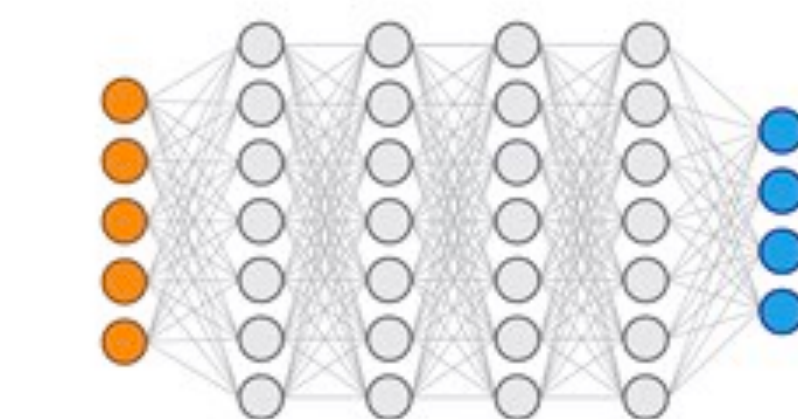
a DNN is a ANN with >1 hidden layer ...

shallow neural network



input layer hidden layer output layer

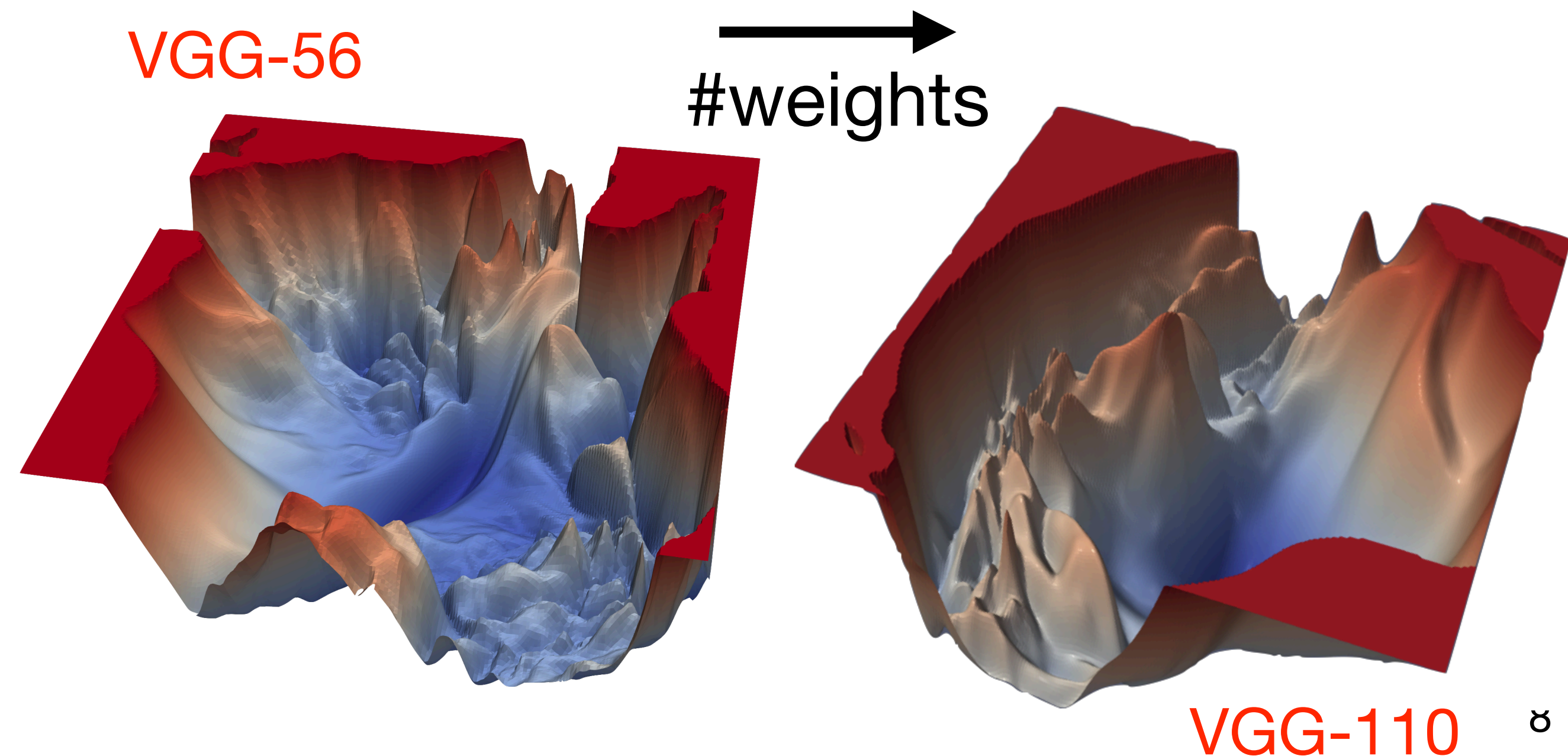
deep neural network



input layer hidden layers output layer

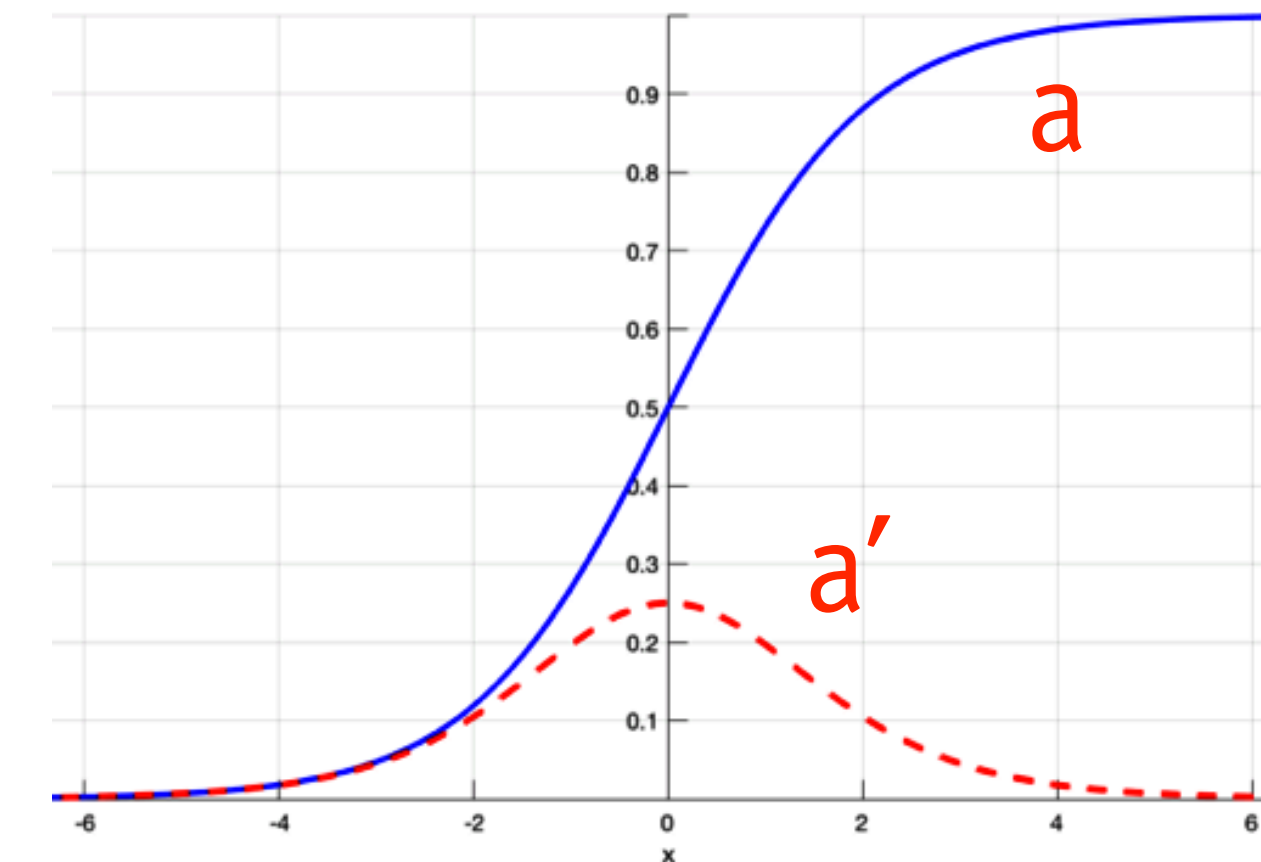
WHY GOING DEEP WORKS?

- the universal approximation theorem tells us that already a FFNN with one hidden layer can approximate any function with arbitrary precision
- however deep architectures are much more efficient at representing a larger class of mapping functions:
 - problems that can be represented with a polynomial number of neurons in k layers require an exponential number of neurons in a shallow network (Hastad et al (86), Y.Bengio (2007))
- other advantages:
 - sub-features (intermediate representations) can be used in parallel for multiple tasks performed with the same model
 - overparametrization in very deep NN seems to have beneficial effects in smoothing the loss function landscape



WHY GOING DEEP IS DIFFICULT: VANISHING GRADIENT

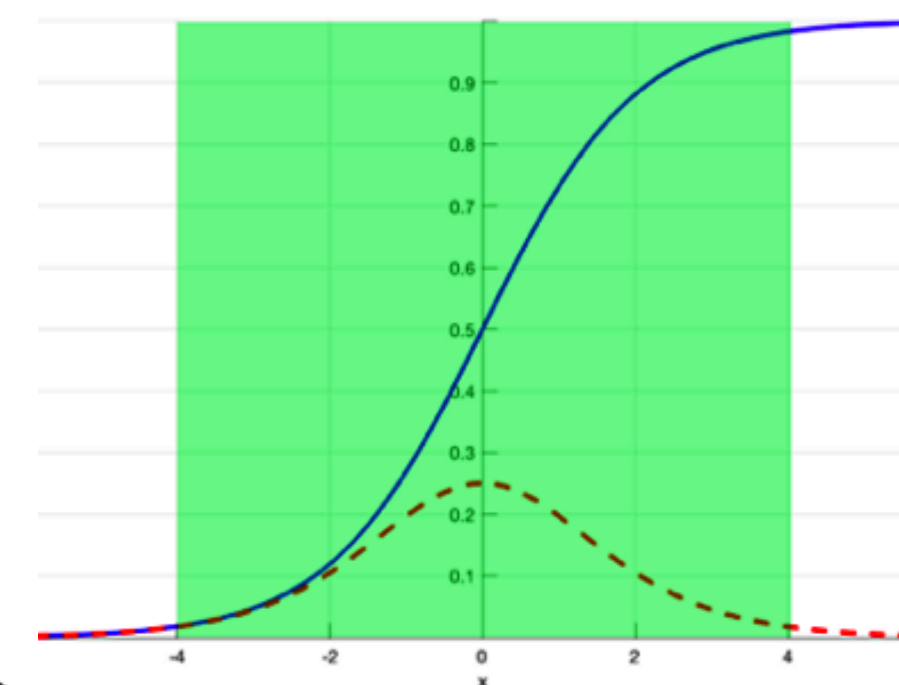
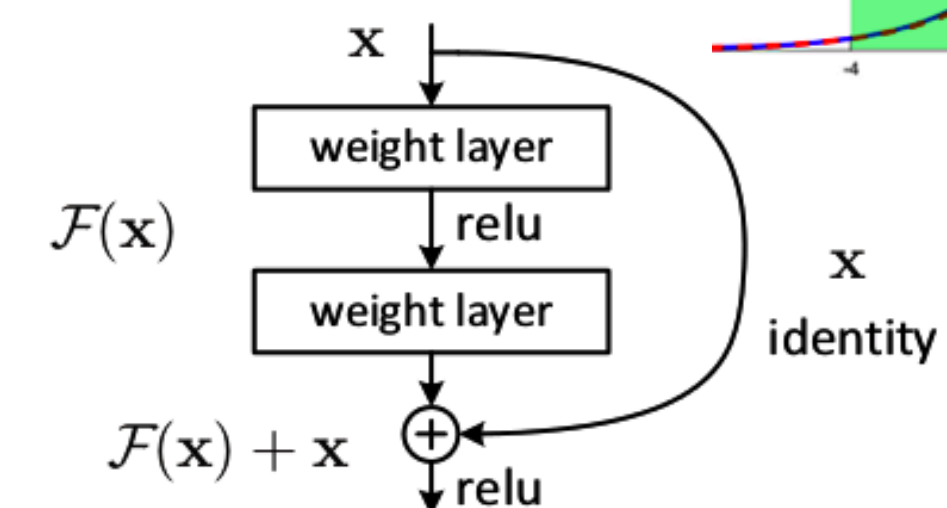
- the main problem in the use of DNN architectures is related to the vanishing gradient
- the first layers of a deep NN fail to learn efficiently
 - reason: during backprop in a network of n hidden layers, n derivatives of the activation functions will be multiplied together. If the derivatives are small then the gradient will decrease exponentially as we propagate through the model until it eventually vanishes



- **SOLUTIONS:**

1. use activation functions which do not produce small derivatives: i.e. **ReLU**, LeakyReLU, Selu, ...
2. use **batch normalisation** layers: in which the input is normalised before to be processed by the layer in order to constraint it to not reach regions of the activation function where derivatives are small (additional advantage: prevent the target of each layer from moving continuously during the training (internal covariate shift))
3. use **residual networks**: in which (residual) connections that do not pass through the activation functions and propagate information to subsequent layers

$$x_i \leftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$



LEARN THE PARAMETERS (I.E. TRAINING OF THE ANN)

- training consists in adjusting the weights according to a given cost function (loss) in order to optimise the performance of the model wrt a specific task
 - weights and biases: **stochastic gradient descent with back-propagation**
 - hyperparameters (parameters whose values are fixed before the learning process begins) optimisation: **heuristic approaches** (autoML: grid or random search, bayesian-opt, ...)

Start by observing that a NN model can be always expressed as a composition of functions associate at each layer

Example: for a deep-NN with d hidden layers: $y = \text{ANN}(x) = f^{(\text{output})}(f^{(d)}(\dots f^{(2)}(f^{(1)}(x))))$

Example: a MLP with:

- a single hidden layer with a: tanh
- an output layer with a: linear
- no bias weights

$$\hat{y} = \sum_{j=1}^{n_h} \tanh[z_j] w_{j1}^{(2)} = \sum_{j=1}^{n_h} \tanh\left[\sum_{i=1}^{n_{\text{feat}}} x_i w_{ij}^{(1)}\right] w_{j1}^{(2)}$$

n_h : number of neurons in the hidden layer

n_{var} : number of input features

weight assigned to the connection between the j-th neuron of the hidden layer and the output neuron

weight assigned to the connection between the i-th neuron of the input layer and the j-th neuron of the hidden layer

TRAINING AND GRADIENT COMPUTATION

- during the training N examples are presented to the network: $T\{x^{(i)}, y^{(i)}\}$ ($i=1, \dots, N$)
- weights are initialised to random values (small and around zero): for example $\sim N(0, \sigma)$
- for each event the output of the model $\hat{y}(x^{(i)})$ is calculated and compared with the expected target $y^{(i)}$ by means of an appropriate loss function that measures the "distance" between $\hat{y}(x^{(i)})$ and $y^{(i)}$:

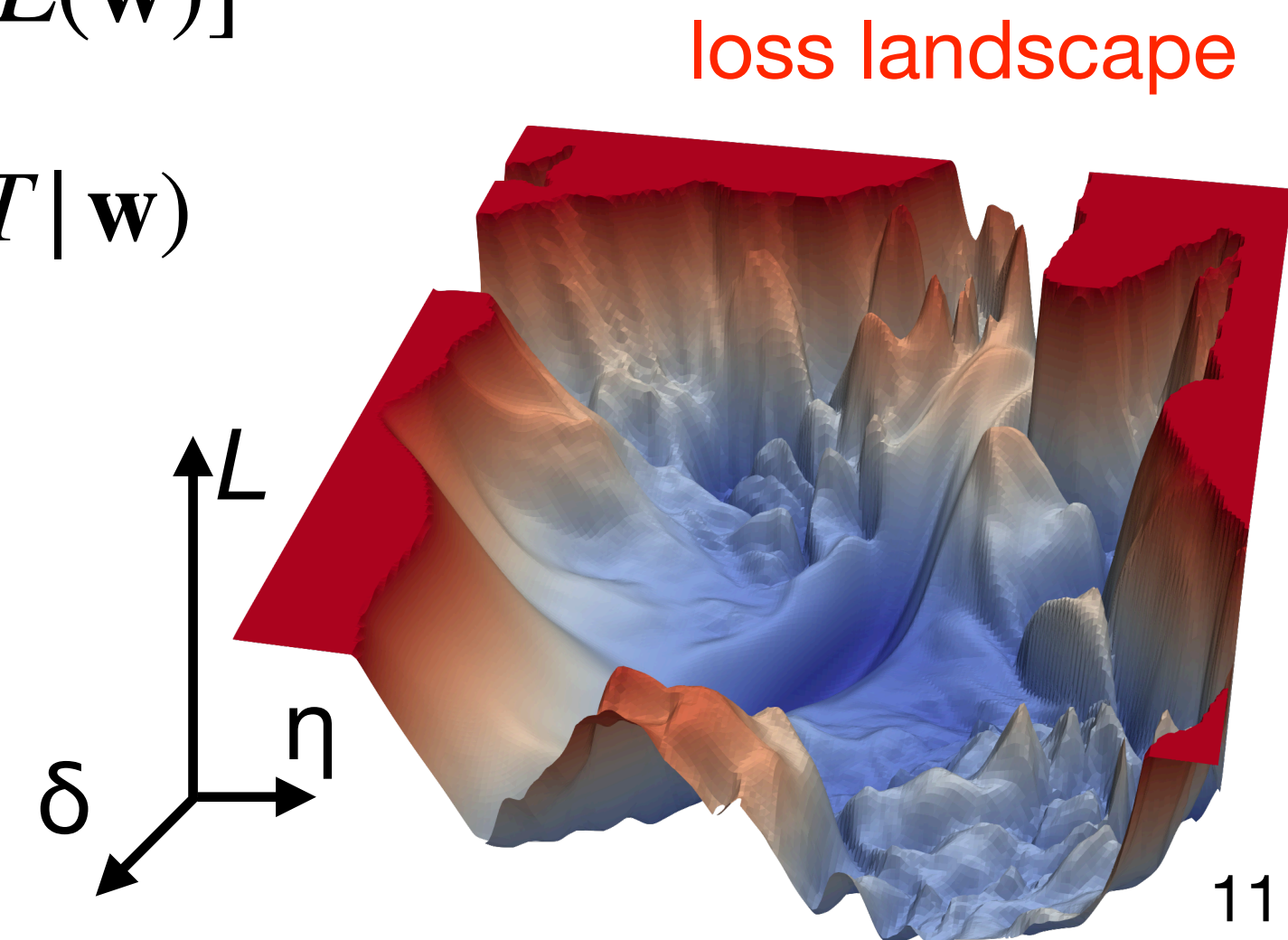
$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N L_i(y^{(i)}, \hat{y}^{(i)}(x^{(i)} | \mathbf{w}))$$

example: MSE

$$L_i(y^{(i)}, \hat{y}^{(i)}(x^{(i)} | \mathbf{w})) = \frac{1}{2} (y^{(i)} - \hat{y}^{(i)}(x^{(i)} | \mathbf{w}))^2$$

- the vector of weights is chosen as the one that minimizes L : $\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}}[L(\mathbf{w})]$
- the minimum is sought with GD / SGD techniques ... $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} L(T | \mathbf{w})$

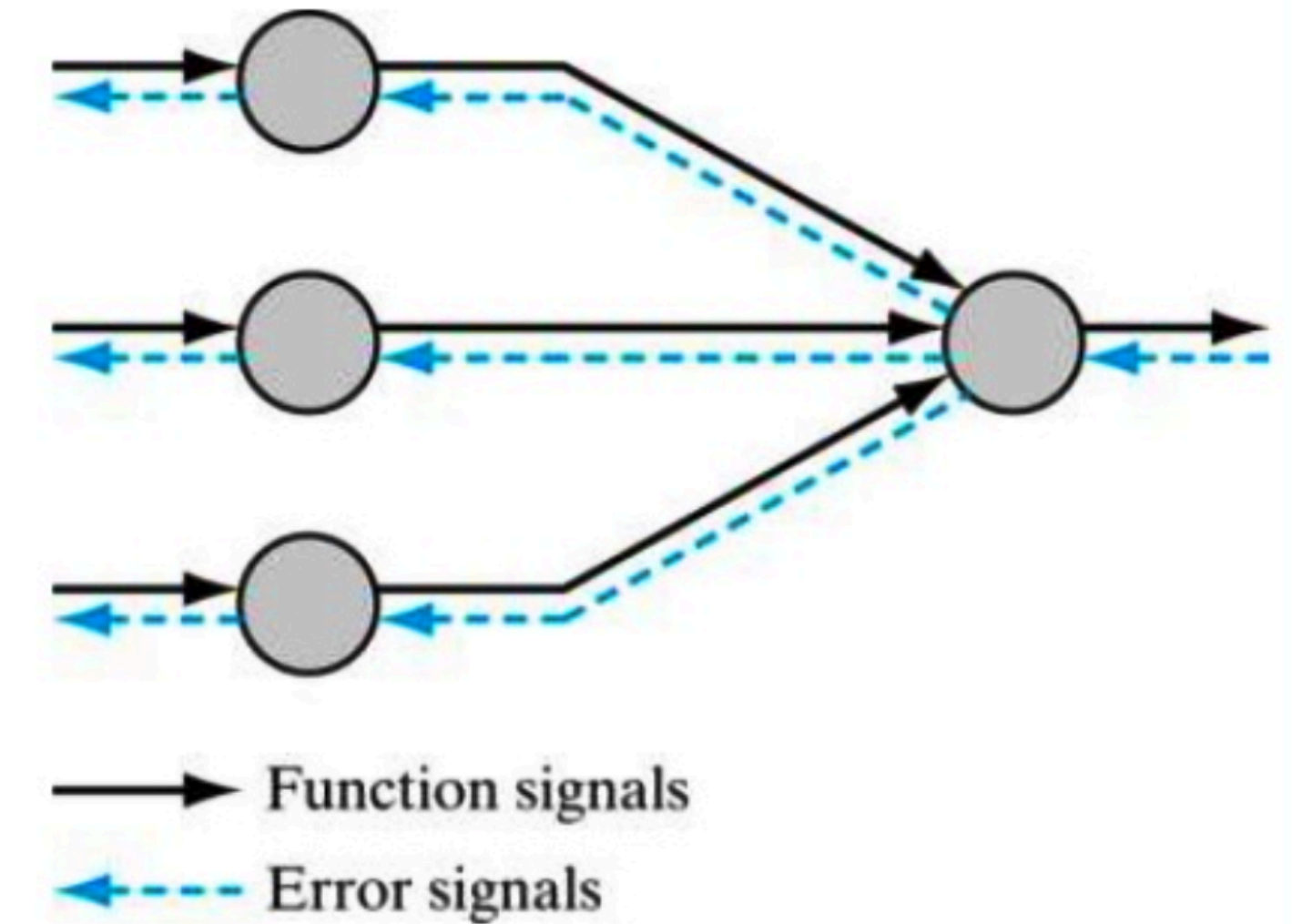
to update the weights of all the layers of the network is necessary to calculate the gradient of complicated **non convex** functions with respect each weight, and to evaluate its numerical value. Doing it in a simple and efficient way is called **Backpropagation** or Backprop procedure



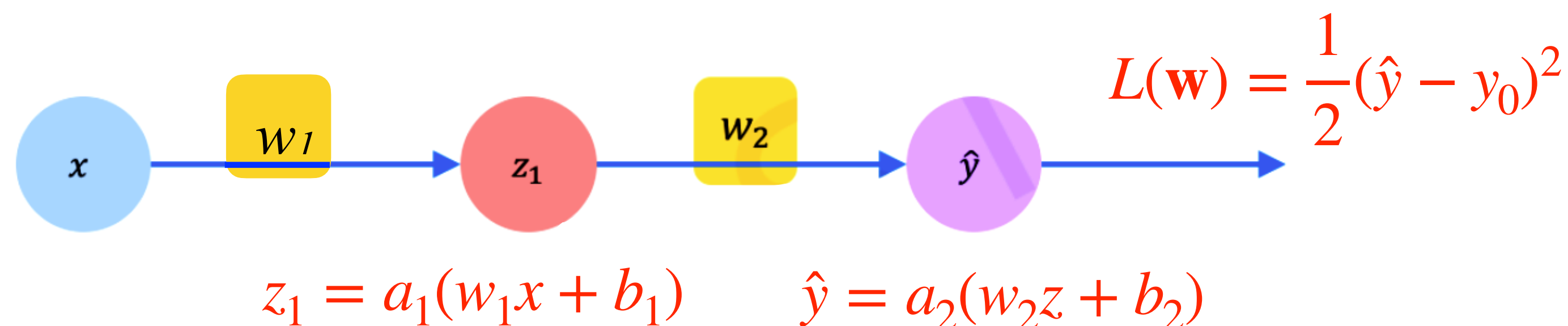
BACKPROPAGATION

- the training of an NN takes place in two distinct phases which are repeated at each iteration:

- Forward phase:** the weights are fixed and the input vector is propagated layer by layer up to the output neurons (**function signal**)
- Backward phase:** the Δ error is calculated by comparing the output with the target y and the result is propagated back, again layer by layer (**error signal**)



- each neuron (hidden or output) receives and compares the function and error signals
- back-propagation consists of a simplification of the gradient calculation obtained by applying recursively the rule of derivation of compound functions (chain rule)



$$\frac{\partial L(\mathbf{w})}{\partial w_1} = \frac{\partial L(\mathbf{w})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_1} = \frac{\partial L(\mathbf{w})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}$$

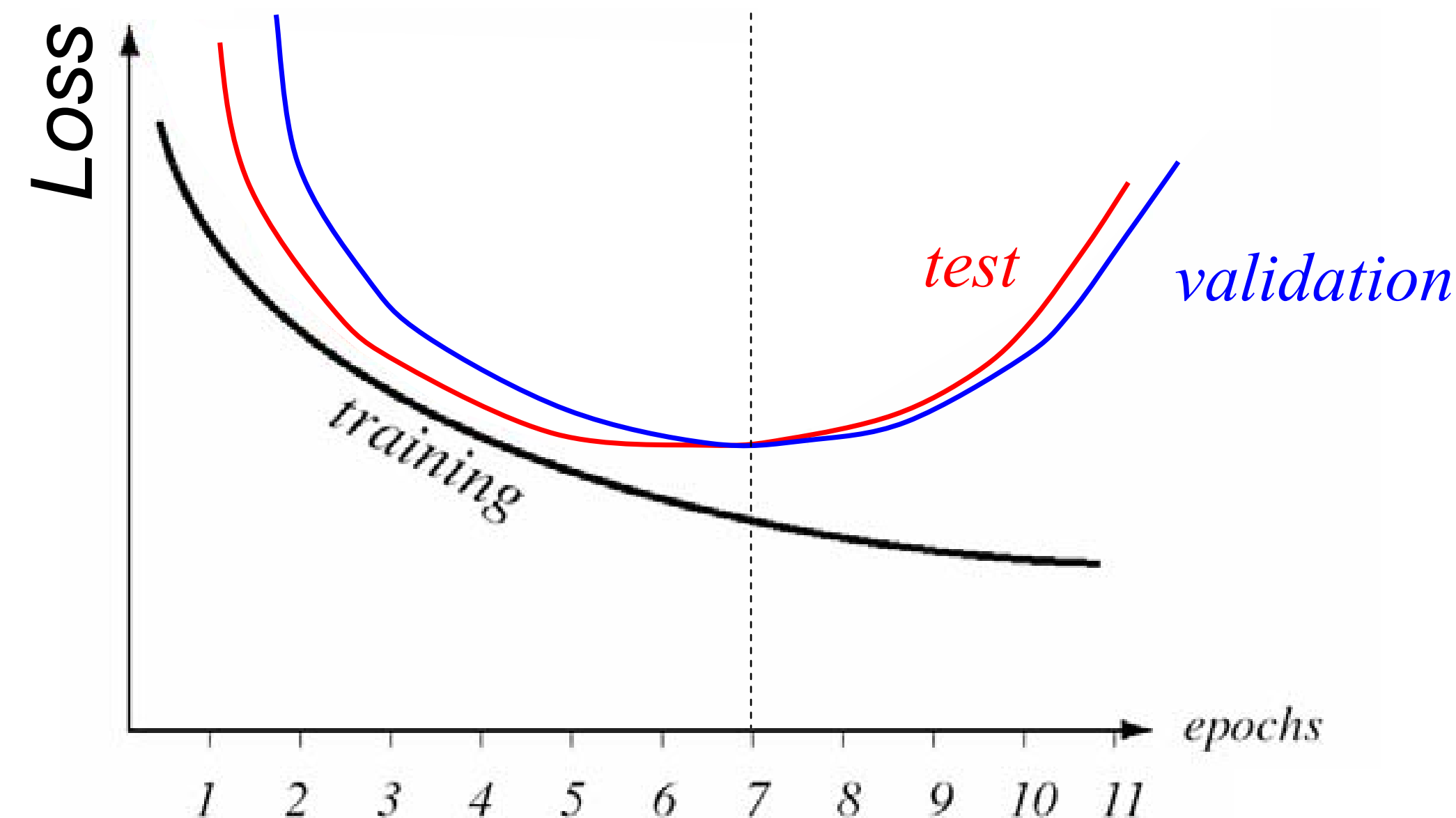
available at the output \nearrow

analytically calculable \nearrow

LEARNING CURVES

- at the start of the training phase when the network weights have been initialised randomly (with small random values) the error on the training set (the loss value) is typically large
- with the iterations (epochs) the error tend to decrease until it reach (typically) a plateau value that depends on:
the size of the training set, the NN architecture, initial value of the weights, the hyper-parameters ...
- training progress is visualized with the learnign curves (loss or accuracy or any useful metrics vs epochs)

- as usual in ML multiple datasets (and/or cross validation) are needed:
 - for the training phase
 - for the optimisation of hyper-parameters and the training stop criteria
 - to evaluate the performances of the trained model



NOTE: WEIGHT INITIALISATION

- general criterion: **the initial weights must break the symmetry of the system**
- if two hidden neurons with the same activation function are connected to the same input then they must have different initial weights, otherwise they will be identically updated by a deterministic algorithm
- **standard rule:**
 - biases fixed to a fixed value (ex. zero)
 - weights randomly initialised around zero (example $N(0, \sigma)$ or $U[-\epsilon, \epsilon]$)
 - large values for σ bring strong symmetry breaking and helps minimise redundancies in the network architecture, but may generate numerical instabilities in deep networks

Most popular heuristics:

m: # inputs of the layer
n: # outputs of the layer

$$N \left[0, \sigma = \frac{1}{\sqrt{m}} \right]$$

normal

$$U \left[-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}} \right]$$

uniform (for dense layers)

$$U \left[-\sqrt{\frac{6}{n+m}}, \sqrt{\frac{6}{n+m}} \right]$$

Glorot

STOCHASTIC GRADIENT DESCENT WITH MOMENTUM

- minimisation of the loss function is performed by gradient descent (GD) techniques
- for large datasets GD becomes computationally inefficient and it is replaced by a stochastic implementation:
- weights are updated after having presented to the model sub-sets (**batches**) of the entire dataset T :

- T is divided in m sub-samples (batches) $T_1 \dots T_m$

- weights are updated using each subset T_i :

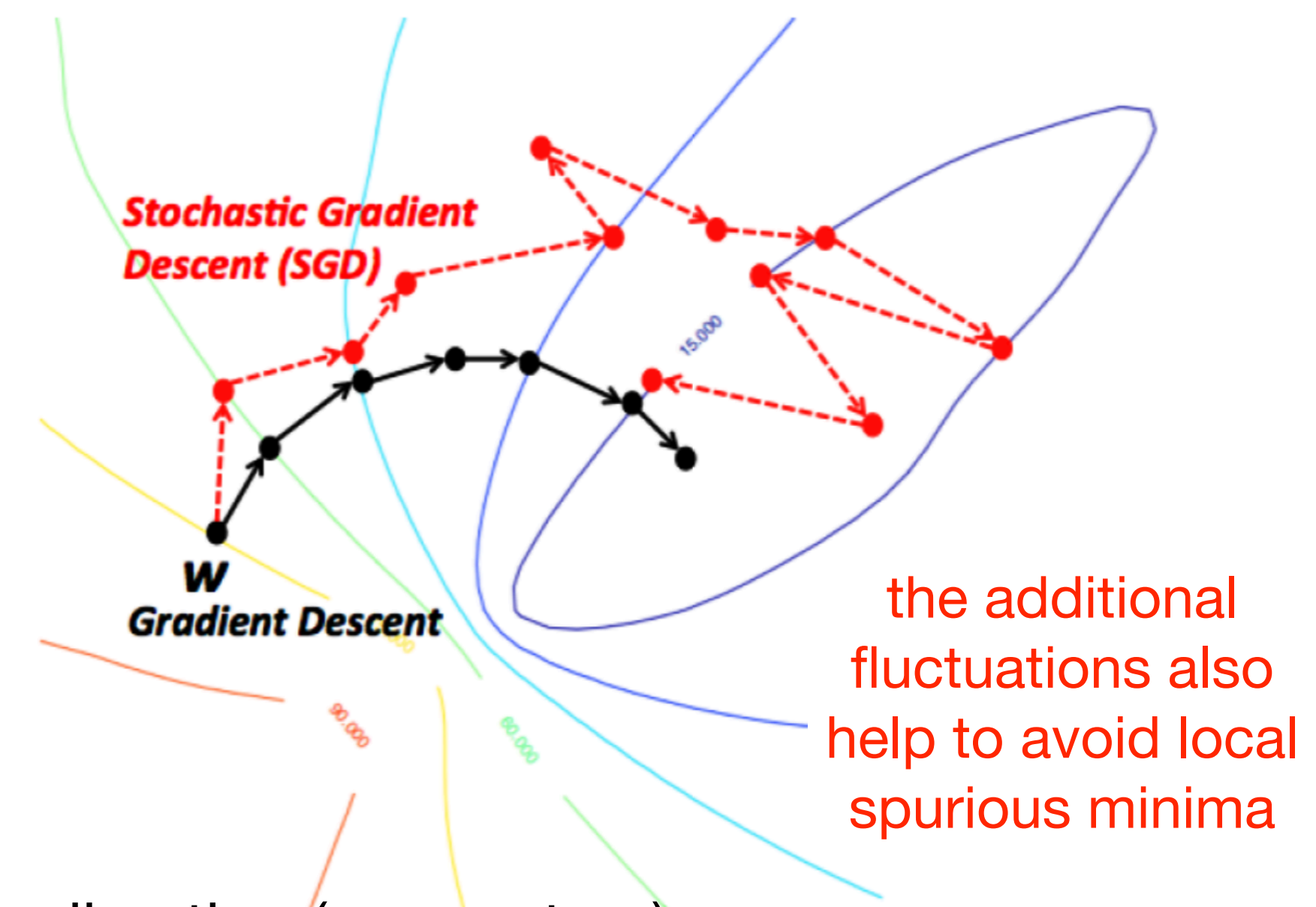
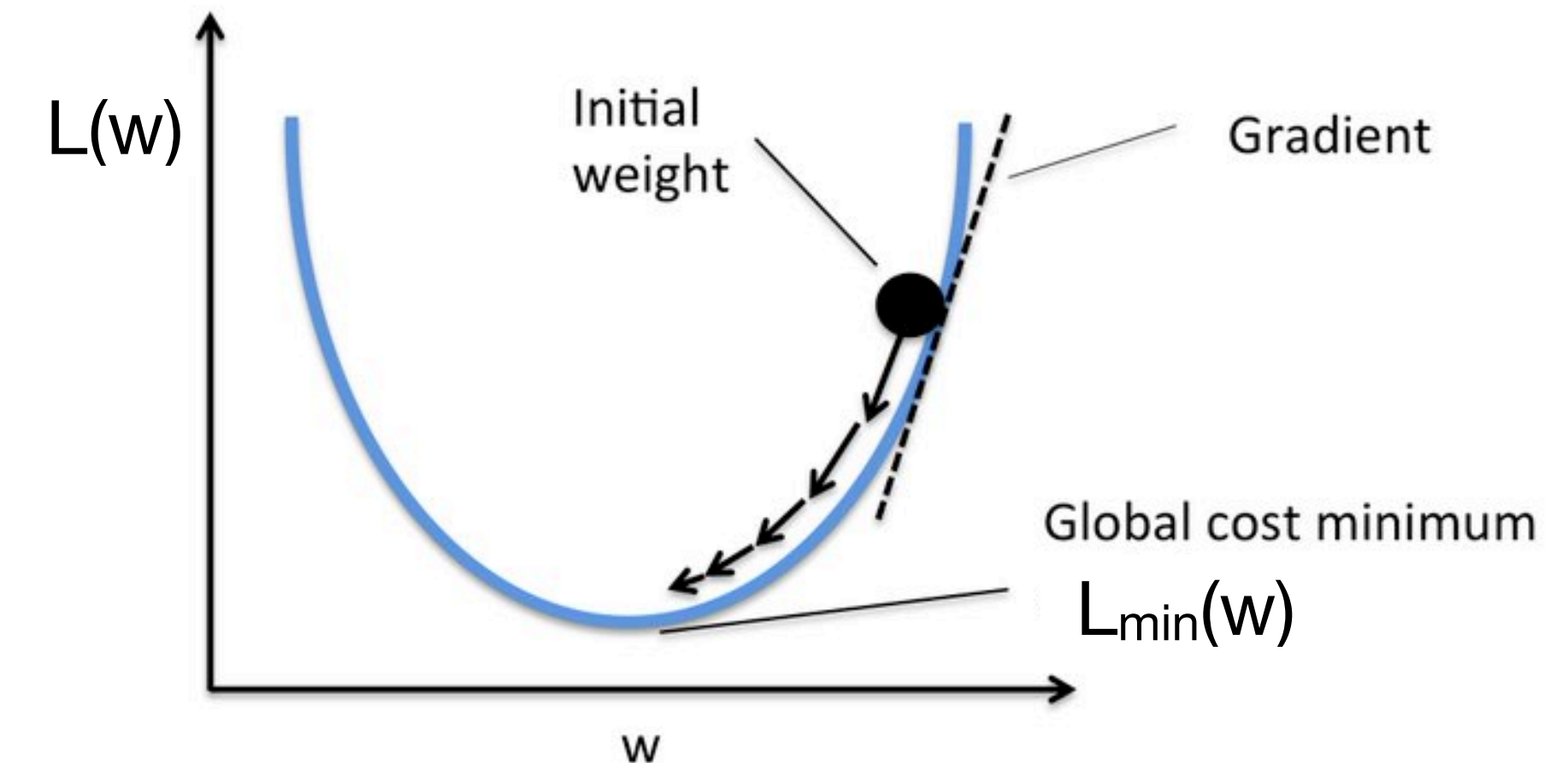
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \mathbf{v}^{(k+1)}$$

$$\mathbf{v}^{(k+1)} = \alpha \mathbf{v}^{(k)} + (1 - \alpha) \nabla L_i(\mathbf{w}^{(k)})$$

previous step
direction

gradient
direction

$$\begin{aligned} \nabla L_i(\mathbf{w}^{(k)}) &= \\ &= \frac{1}{N_i} \nabla \sum_{k \in T_i} L(\mathbf{x}_k, \mathbf{y}_k, \mathbf{w}) \end{aligned}$$

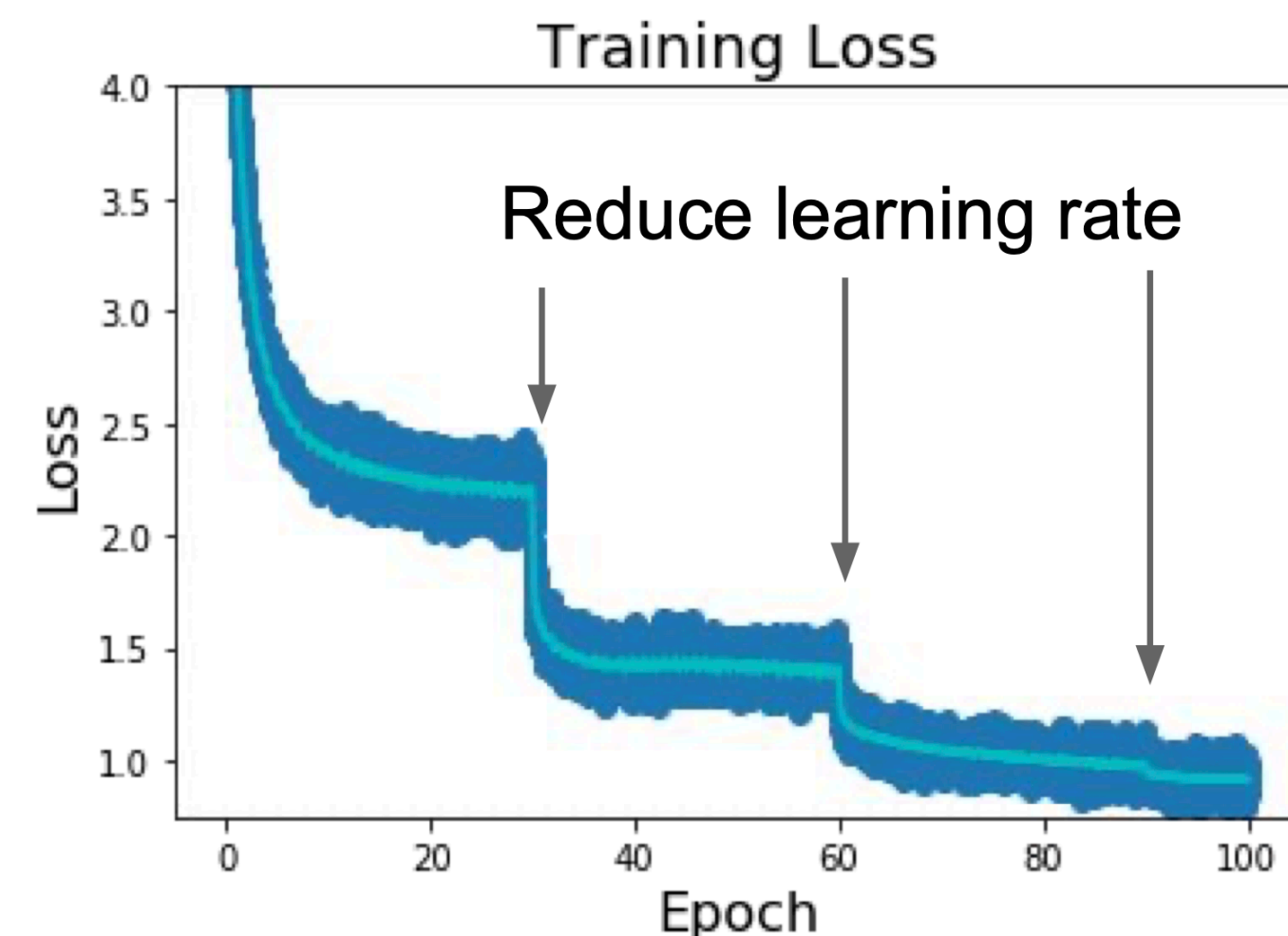
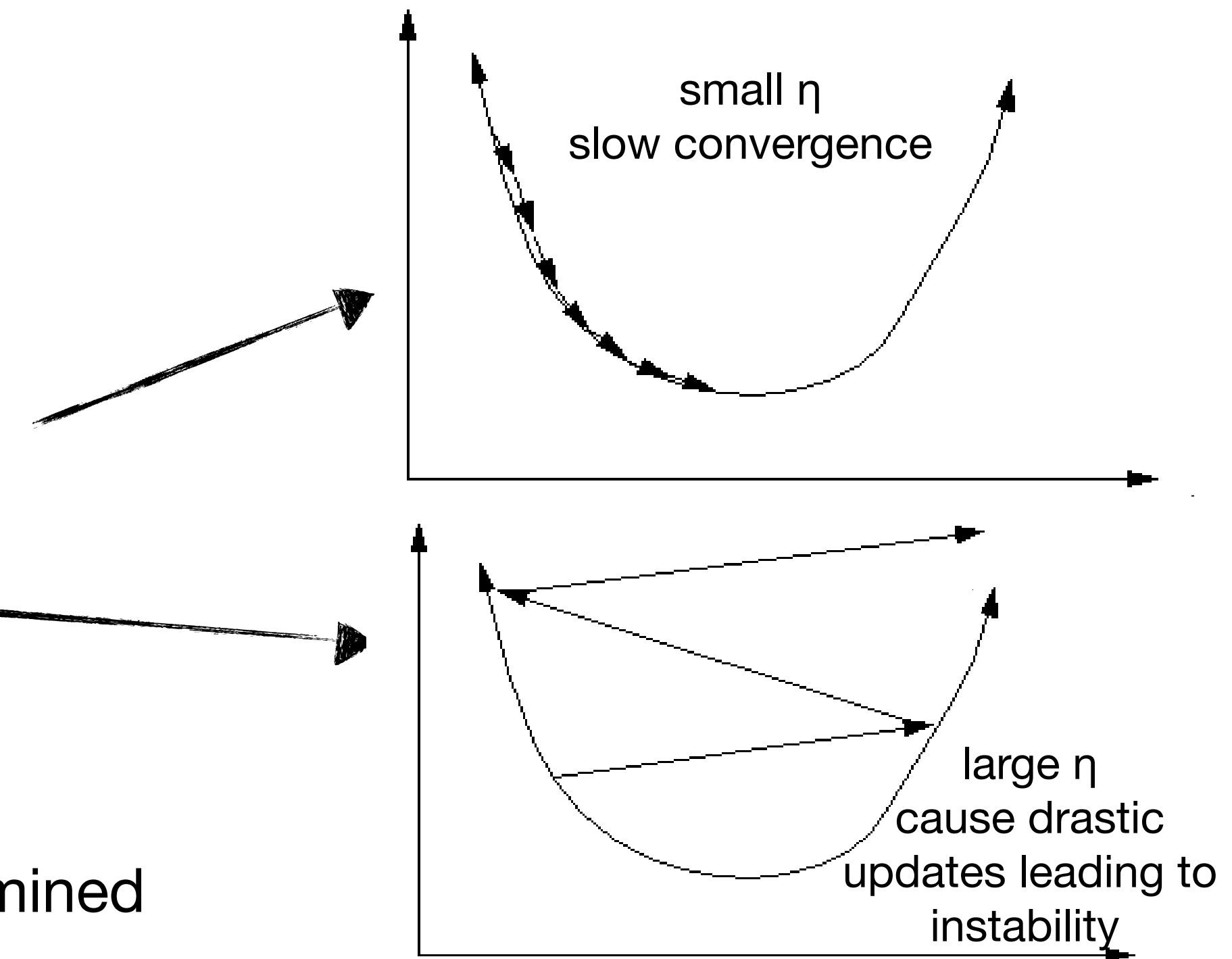


MOMENTUM:

- for $\alpha = 0$ we have classic GD/SGD
- for $\alpha = 1$ the gradient descent is ignored and the weight update follows the previous direction (momentum)
- typically: $\alpha \sim 0.9-0.99$

VARIABLE LEARNING RATE

- η affects the speed of convergence:
 - a small value can result in excessive slowness and an increase in the probability of being trapped in local minima
 - a large value can cause the algorithm to diverge
- solution: **Variable Learning Rate and Adaptive Learning Rate Optimizers**
 - during the iterations the learning rate decrease according to a predetermined schedule or adapt following a specific strategy



ADaptive grad: the learning rate associated with each weight is individually scaled inversely proportional to the root of the historical sum of squares of the gradients for that parameter:

- weight associated to features that appears with high frequency: reduce η
- weight associated to features that appears with low frequency: increase η

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \frac{\partial L}{\partial w}$$

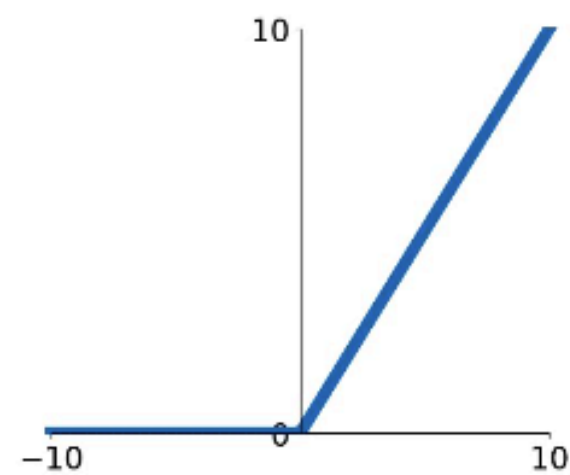
$$G_t = \sum_{\tau=1}^t \left[\frac{\partial L}{\partial w} \right]^2$$

several implementations:
Adadelta, RMSProp, Adam, ...

CHOICE OF ACTIVATION FUNCTIONS FOR THE HIDDEN LAYERS

In general, any function that met the universal approximation theorem conditions is fine. In practice some of them work better for specific NN architectures ...

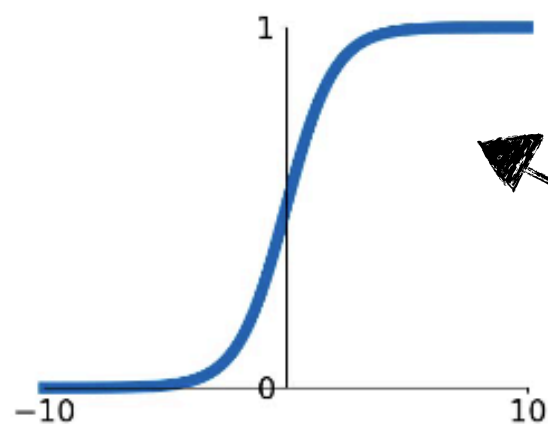
ReLU
 $\max(0, x)$



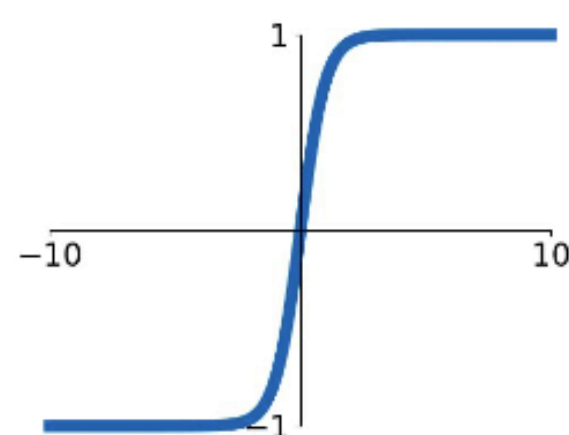
the most popular:

- allows non linear dynamics
- faster convergence of the NN because doesn't saturate
- no vanishing gradient problem
- induce gradient sparsity (0 output for negative values, i.e. fewer active neurons). This can be an advantage or an issue depending on the specific ANN architecture. Needs to be monitored and in case of problems replaced with alternatives

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

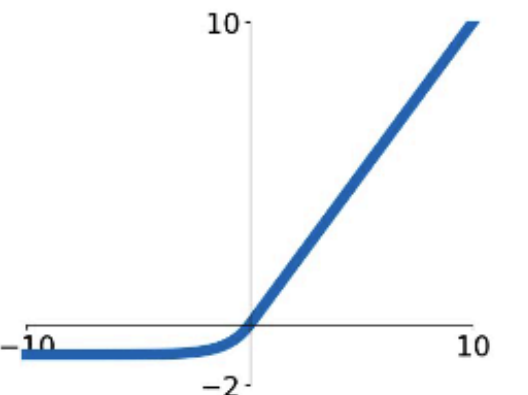


tanh
 $\tanh(x)$



ELU

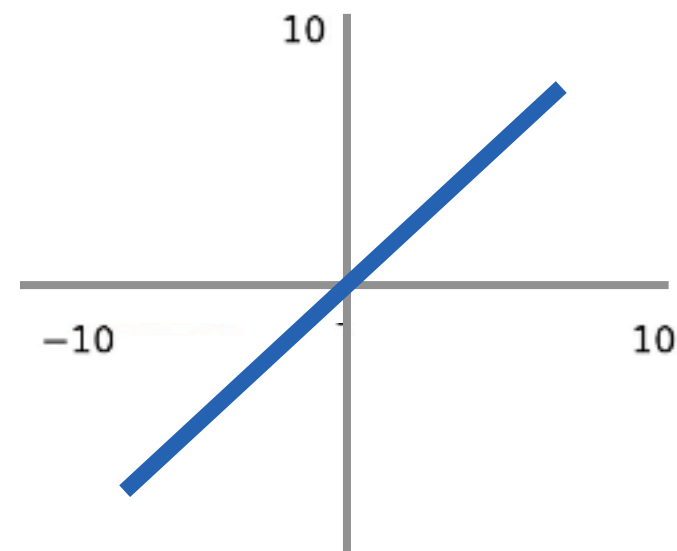
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



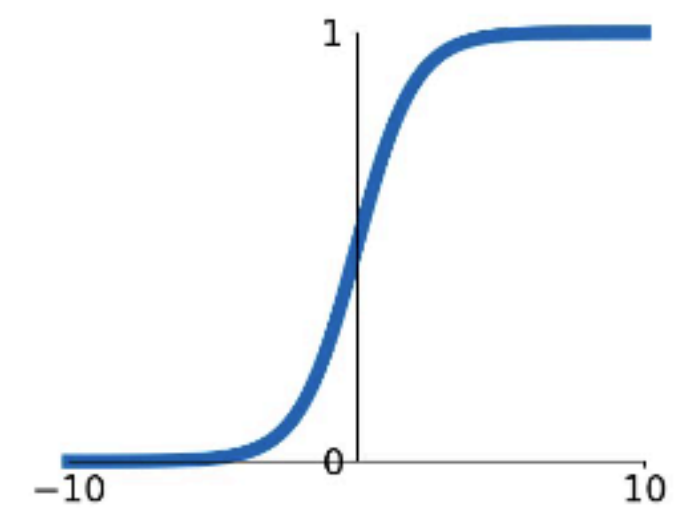
should not be used in dense and convolutional layers:

- gradient vanishes away from $x=0 \rightarrow$ **vanishing gradient problem**
- sigmoid has output not centered in zero \rightarrow affects SGD dynamic (**zig-zag instabilities**)
- **used in RNN to control gated I/O and often in dense layers in GAN to avoid sparsity**

POPULAR ACTIVATION FUNCTIONS FOR THE OUTPUT LAYER



Identity (linear): standard choice for regression tasks



Sigmoid: used in binary classification problems (2 classes) with single output neuron or multilabel (multiple mutually inclusive classes) or when the output features are numbers in (0,1)

$$y_i = \frac{e^{z_j}}{\sum_{j=1}^n e^{z_j}}$$

Softmax: $\mathbb{R}^n \rightarrow [0,1]^n$

- soft version of the argmax output
- often used in multi-class classification tasks with one-hot encoded labels
- output of each neuron $\in (0,1)$ and interpretable as a probability ($\sum y_i = 1$)

LOSS FUNCTIONS

Modern ANNs are **trained using the maximum likelihood principle**, consequently the most used loss functions are simply equivalent expressions/approximations of the negative log-likelihood:

$$L(\mathbf{w}) = -\mathbb{E}_{\mathcal{T}}[\log p_{model}(y | x, \mathbf{w})]$$

most popular forms:

MSE

$$MSE = ||y - \hat{y}||_2 = \frac{1}{N} \sum_{i=1}^N (y - \hat{y})^2$$

for regression problems
(also MAE, UberLoss, ...)

binary cross-entropy

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

p = predicted probability (0,1)
y = label (0 or 1)

given two distributions p and q, $H_p(q)$ measures the average number of bits needed to identify an event extracted from the set, when the p model is used for the probability distribution, rather than the "true" distribution q. **It is usually the best loss function to train ANNs that output probabilities (example: softmax)**

NOTE: generalisation for multi class problems

- categorical cross-entropy (one-hot encoded label)
- sparse categorical cross-entropy (integer labels)

MOST CRITICAL ASPECTS IN THE TRAINING OF ANNs

- training speed:

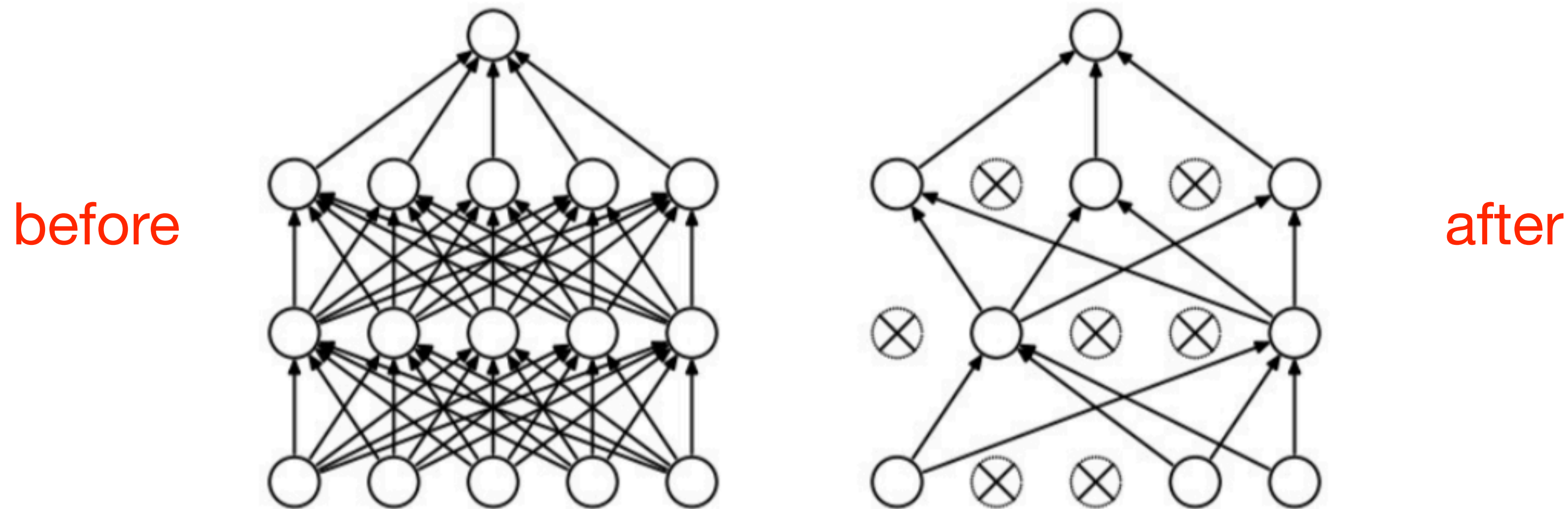
- mitigated by using stochastic-learning, momentum, adaptive learning rate (Adam or RMSProp), non saturating activation functions (ReLU, ...), smart weight initialisation, and scaling of the input features
- but most of all by using dedicated coprocessors (GPUs, TPUs, ACAPs, SOCs, FPGAs, ...)

- hardcore overfitting:

- the main problem for overparametrized models as deep neural networks with $O(M)$ of weights
- inevitable consequence of the trade-off between variance (large expressive power) and bias (generalization)
- issue controlled by applying a set of **regularization techniques aimed at reducing the error on the test set** (typically at the expense of error on the training set)
- regularisation techniques impose constraints on different aspects of the NN model such as the complexity of the NN architecture, the error reduction on the training set, the representation of the loss function landscape, the size of weights, etc... so that will be **more difficult for the model to learn characteristic that are specific of the training set**

DROPOUT

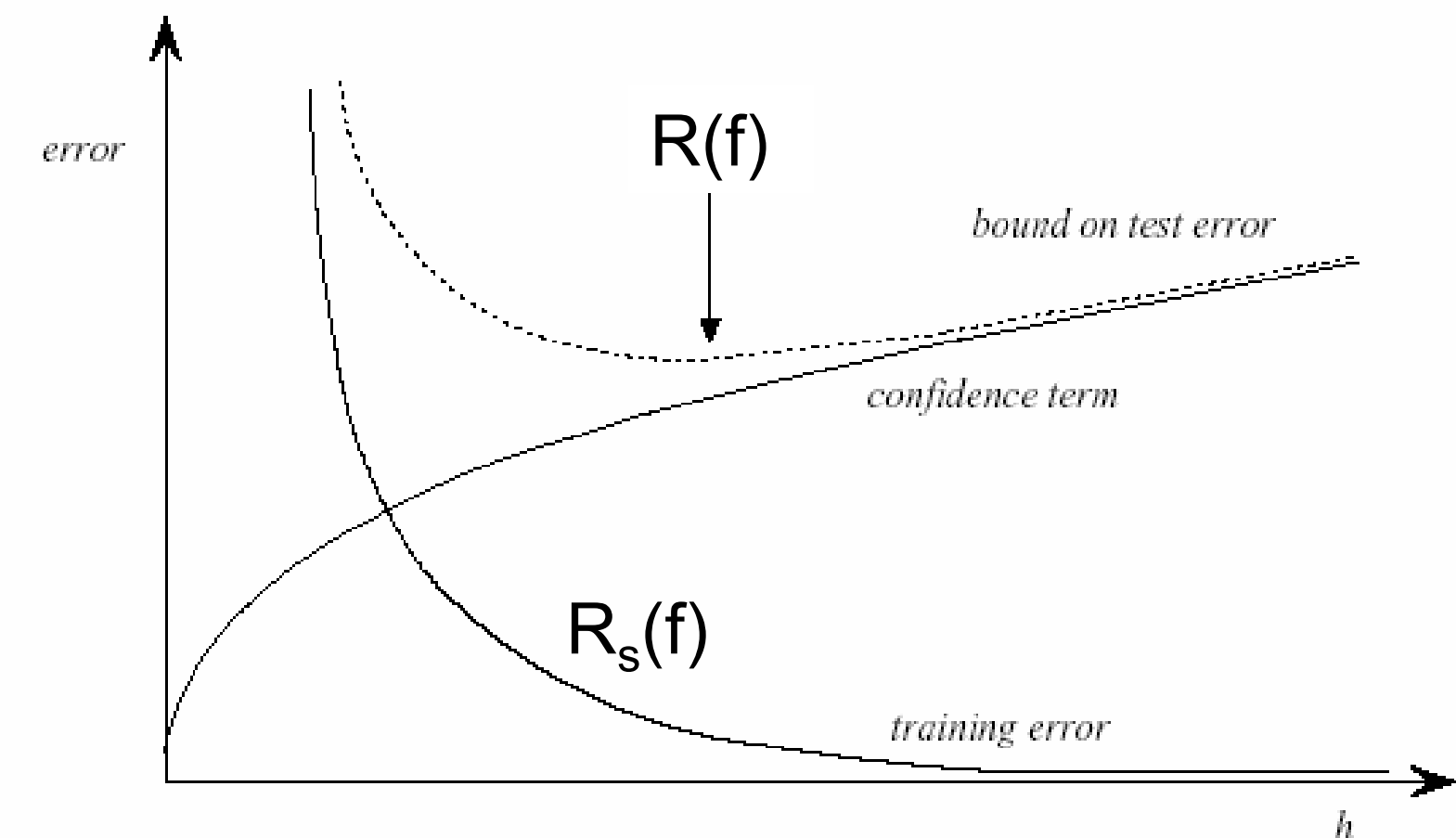
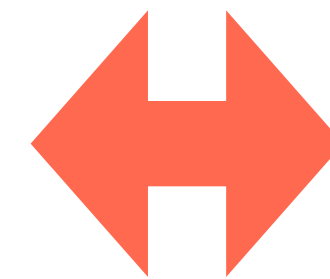
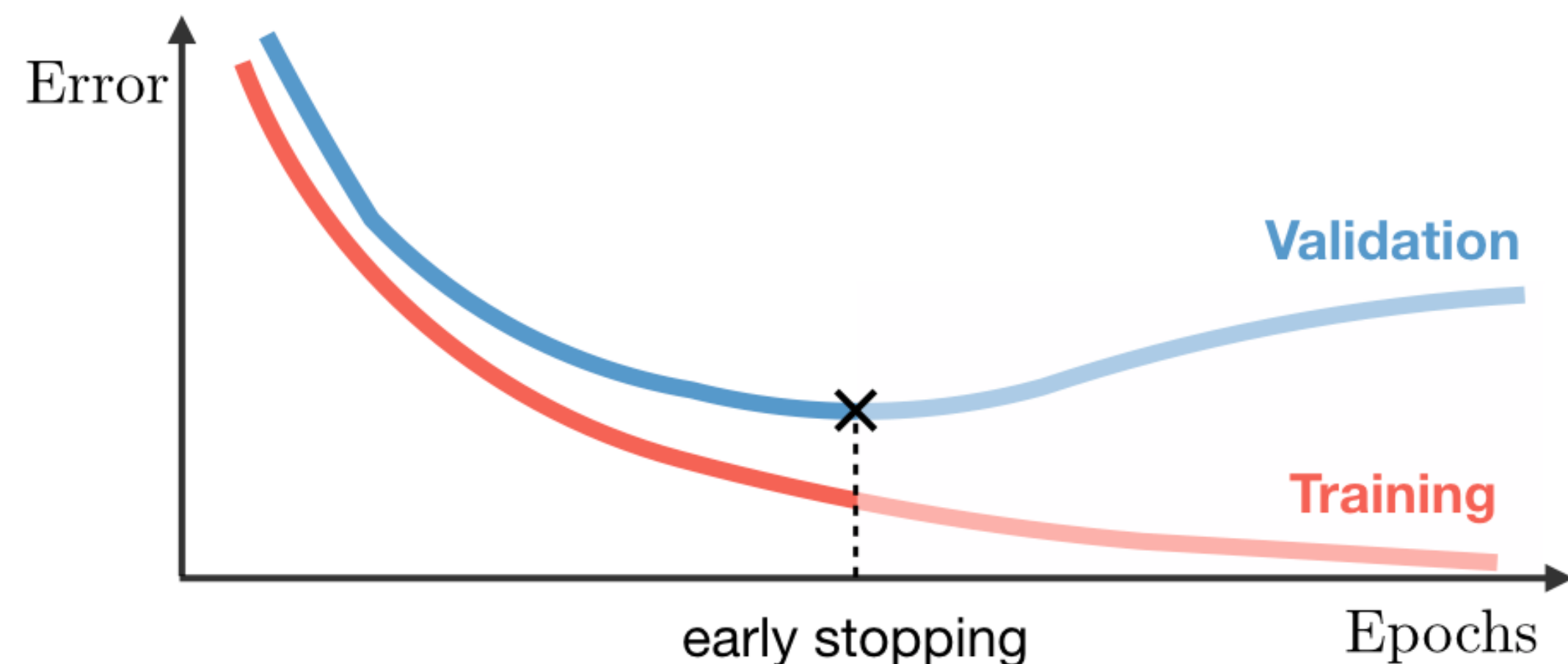
- very popular and powerful technique to prevent overfitting in architecture of deep neural network
 - imposes constraints on the complexity of the Neural Network architecture
 - neuron connections are eliminated based on a defined probability
 - forces the model to not rely excessively on particular sets of features



used routinely in the context of ConvNETs in which it can sensibly increase performance on the test set

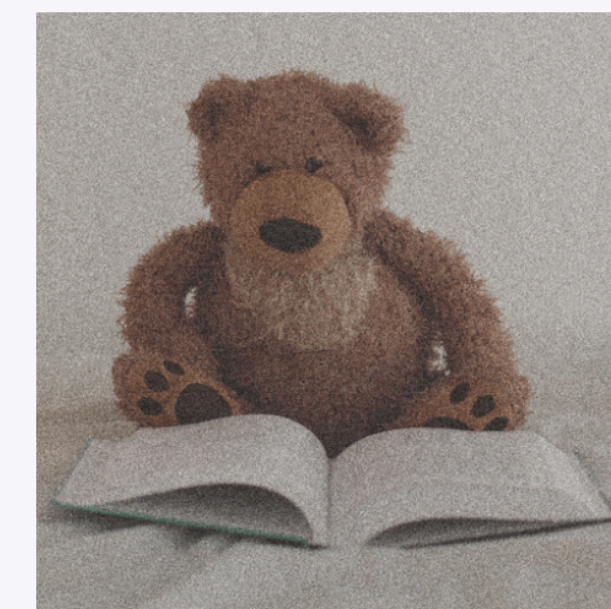
EARLY STOPPING & NOISE INJECTION

- **early stopping**: imposes constraints on the error reduction on the training set
- the training process is stopped as soon as the loss on the validation sample reaches a plateau or start to increase



- **noise injection/information loss**: makes it more difficult for the network to learn specific characteristics of the input features
 - random flip of labels
 - random occlusion of pixels or feature bits
 - adding white/colored/gaussian noise to the features
 - ...

Noise addition



- Addition of noise
- More tolerance to quality variation of inputs







Information loss



- Parts of image ignored
- Mimics potential loss of parts of image

DATA AUGMENTATION

- the **best way** to make an ML algorithm **to generalize better** is to train it on larger and more expressive data
- but having more data is normally the real issue in ML/DL → solution: artificially increase the dimension of the training set

Original	Flip	Rotation	Random crop	Color shift	Contrast change
					
<ul style="list-style-type: none">• Image without any modification	<ul style="list-style-type: none">• Flipped with respect to an axis for which the meaning of the image is preserved	<ul style="list-style-type: none">• Rotation with a slight angle• Simulates incorrect horizon calibration	<ul style="list-style-type: none">• Random focus on one part of the image• Several random crops can be done in a row	<ul style="list-style-type: none">• Nuances of RGB is slightly changed• Captures noise that can occur with light exposure	<ul style="list-style-type: none">• Luminosity changes• Controls difference in exposition due to time of day

+ modern approaches are based on generative DL (GAN, VAE, ...)

IMPORTANT: the applied transformation should not bias the relevant “physics” of your data

ARCHITECTURES FOR VISION: CONVNET/CNN

- CNN are specific DNN designed to excel in image recognition tasks
 - operate directly on the images (raw “pixel” information organised in a fixed mesh)
 - the expressive power of the model is constrained based on assumptions on the properties of the input:
 - **symmetrical spatial structure of the input:** pixels organised in a fixed size mesh
 - **translation invariance (equivariance):** sub-features in the image remain the same in different points of the image
 - **self-similarity:** two or more identical sub-features present in the image can be recognized with a single filter that identifies one of the sub-features
 - **compositionality:** a complex feature made of several sub-features can be recognized by identifying only few sub-features
 - **locality of the features:** to identify a sub-feature it often takes just a few pixels concentrated in a small portion of the image itself
- implementation: apply layers called **convolutional filters** that operate on the input by recognizing the local sub-features present there
 - the same filters use **shared parameters** (weights) and **sequentially analyse all portions of the image**
 - **weights** of the filters are not fixed but **are learned**
 - CNN learns from the training data sample the best set of weights to solve the task given the chosen architecture

HOW A NN “SEE” AN IMAGE ...

images for a computer are essentially meshes (tensors) of numbers



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
206	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
206	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

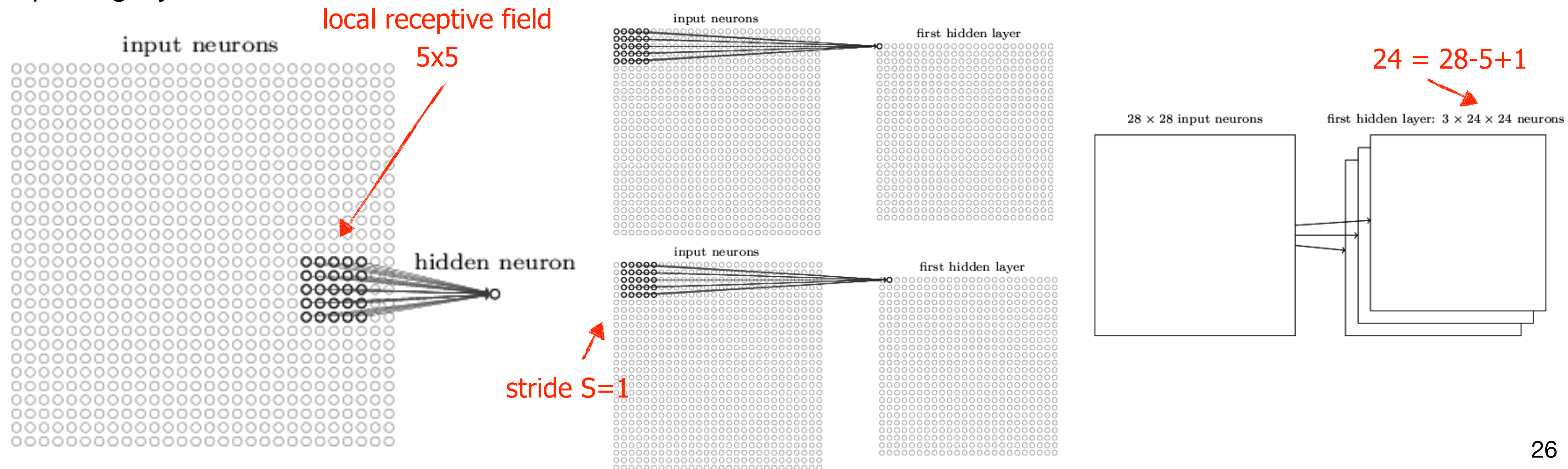
gray scale image with 8bit depth: $12 \times 16 \times 1$ intensity $\in [0, 256]$

color image with n-bit depth: $m_1 \times m_2 \times 3$ with each RGB intensity $\in [0, 2^n]$

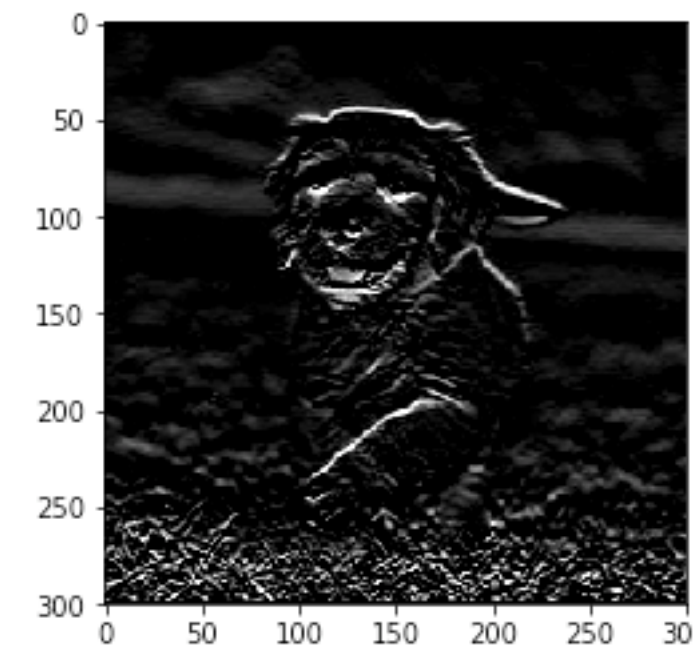
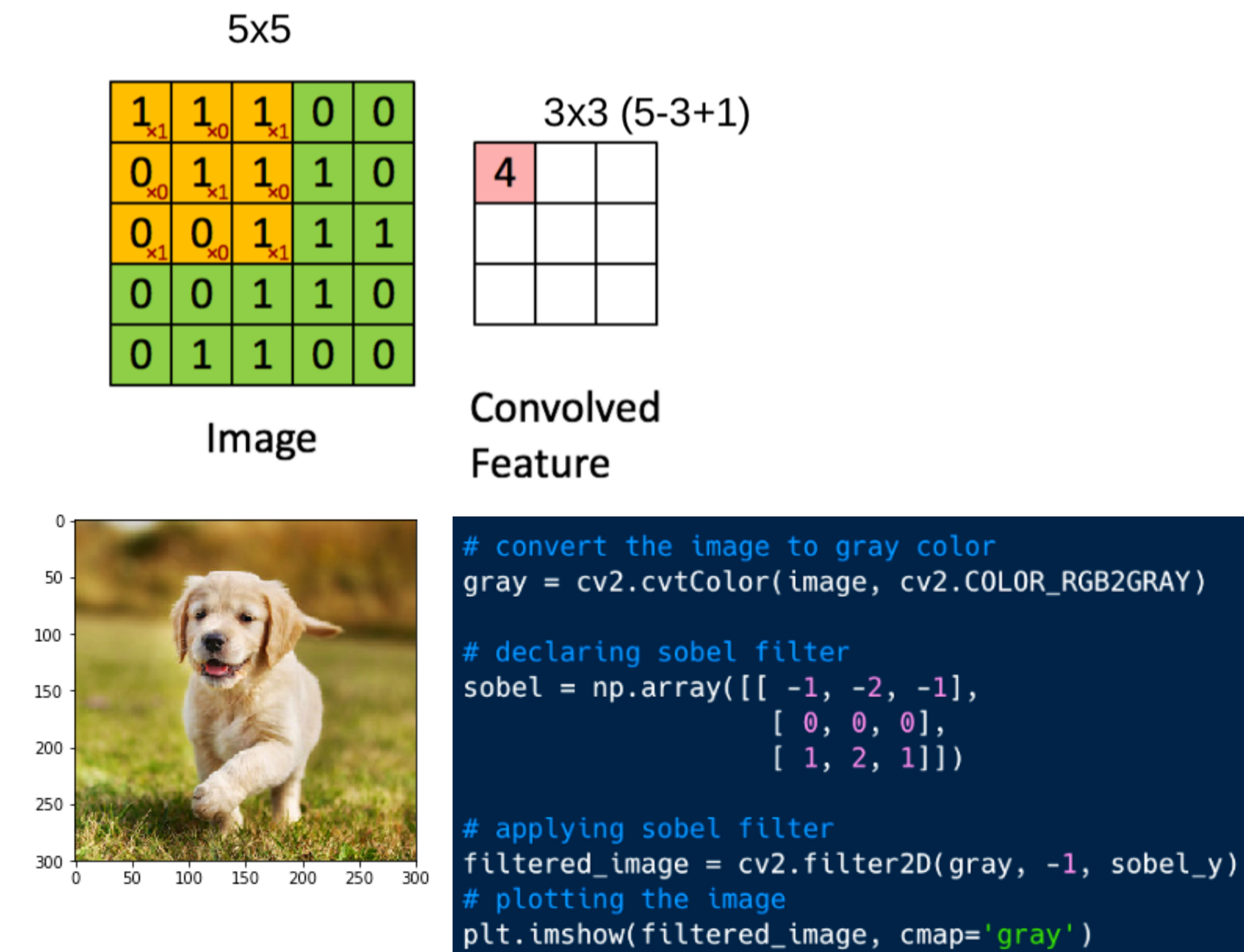
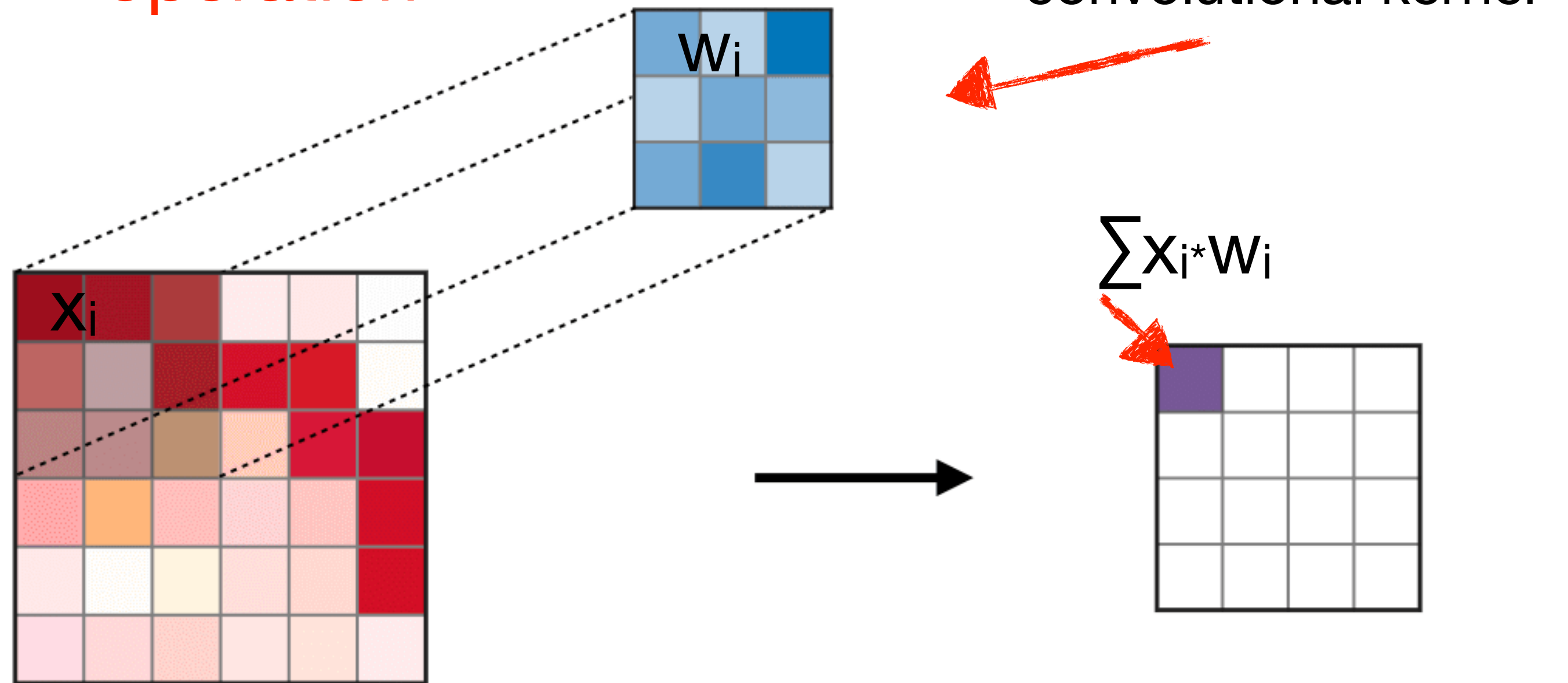
credit MIT AI course

CONVOLUTIONAL FEATURE EXTRACTION LAYER

- used to identify similar features that are present in different position of the image
- based on three basic ideas:
 - local receptive field
 - shared-weights (kernels)
 - pooling layers
- input neurons (one for each NxN pixels of the image) are NOT fully connected with all the neurons of the first hidden layer. Connections exist only for localised and small regions of the image called local receptive fields
- the local receptive field is shifted through the whole image: for each shifted receptive field there will be an hidden neuron in the hidden layer



convolution operation



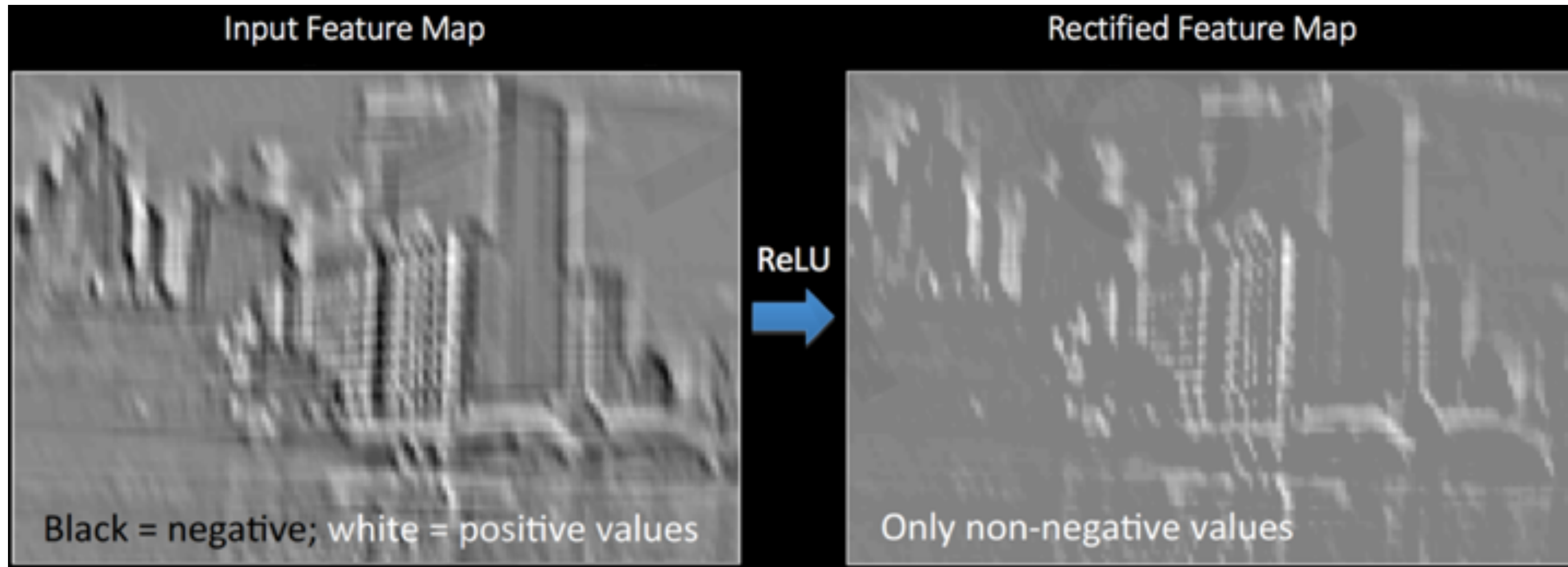
• shared-weights:

- all the hidden neurons of a given hidden layer share the same weights → all neurons of the hidden layer detect the same sub-feature, only in different regions of the image
- as the CNN has to identify many sub-features: there are many convolutional kernels each one with an associated hidden layer: input image (n,m,3) → output (k,l,d)
- huge advantage wrt DNN: much smaller number of weights to learn ...

NON LINEARITY

after the convolution operation, an activation function is applied to each (neuron) of the filtered image (ex. ReLU: all negative values are set to zero)

- emphasize only some of the dominant characteristics of the sub-features selected by the filter

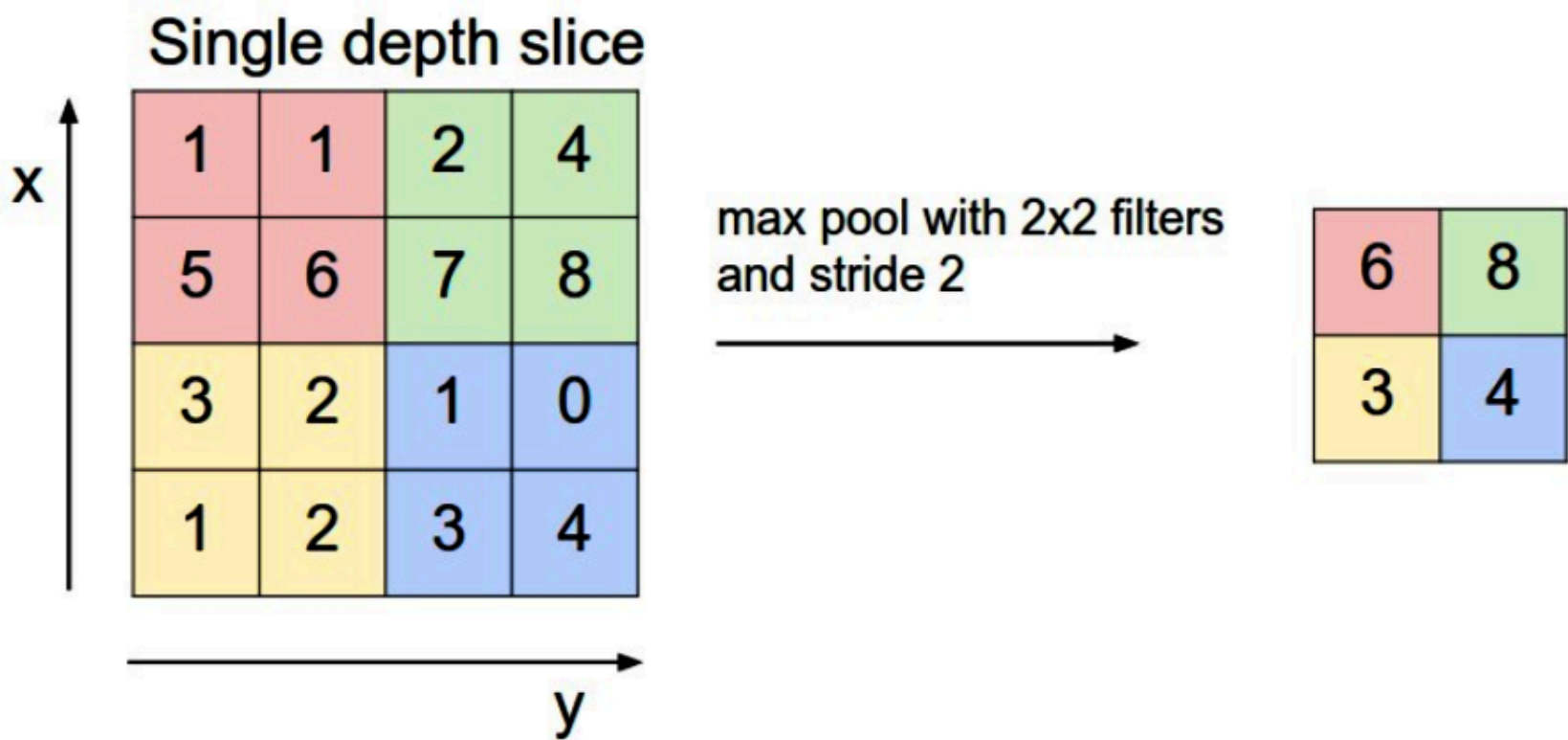


before ReLU

after ReLU

- **pooling layers:**
 - in addition of the convolution layers a CNN has also other layers called **pooling layers**, usually used after each convolution layer. They performs a **downsampling operation**: simplifying the information in output from the convolutional layer (less weights) and making the NN less sensitive to small translations of the image
 - motivated on the fact that once a sub-feature is found, to know the exact position is not as important as to know the relative position wrt the other sub-feature in the image

Type	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	<ul style="list-style-type: none"> • Preserves detected features • Most commonly used 	<ul style="list-style-type: none"> • Downsamples feature map • Used in LeNet

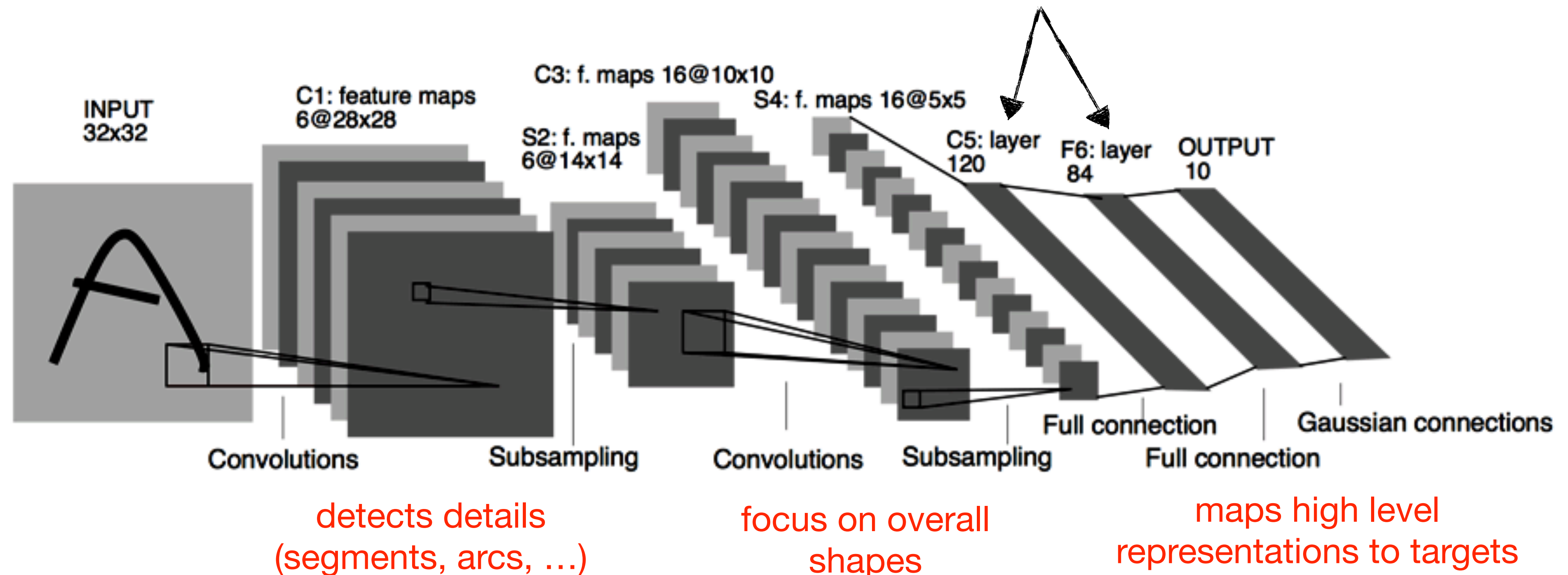


FULL CNN: CONV BLOCKS + DENSE MLP STAGE

- generally after the convolutions the output of the convolutional layers is connected via a flattening layer with one or more dense layers (DNN), that are used to optimize objectives: class scores (classification), mapping (regression), etc...

Example: **LeNet** (Yan LeCun)

multi-staged CNN for classification: (Conv2D+MaxPooling)x2 + **2xDense** + output layer (soft max)



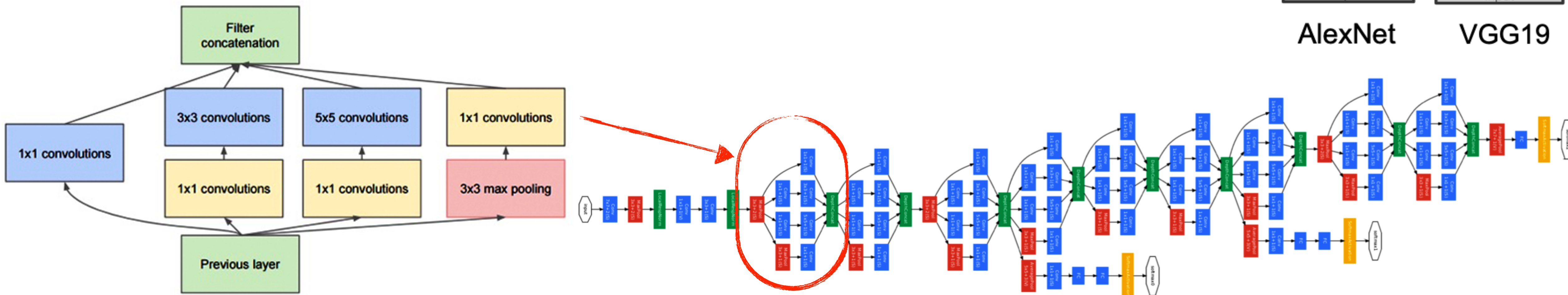
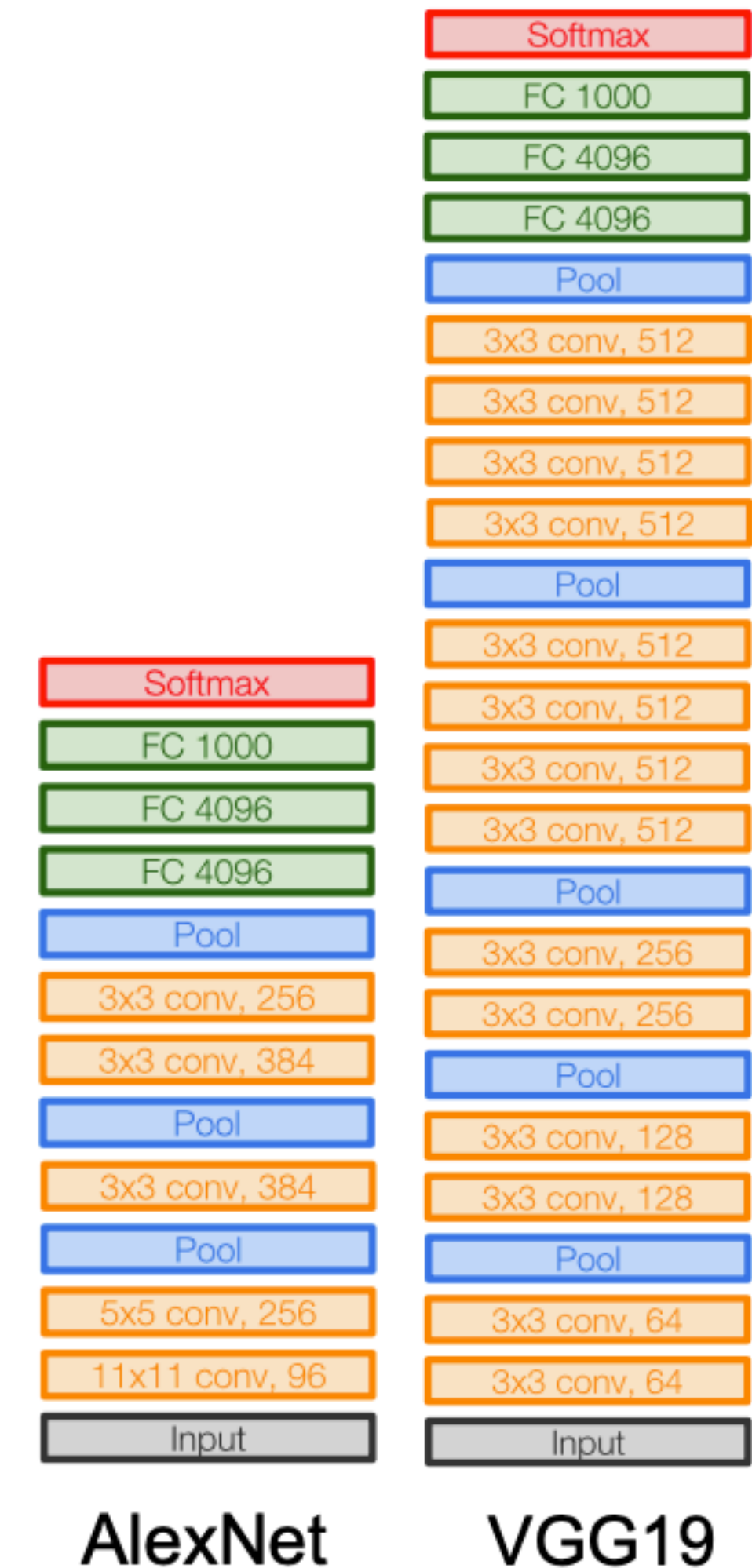
MODERN CNNs

philosophy: deeper is better ...

- **AlexNet**: better backdrop via ReLU, dropout, batch normalisation, data augmentation
- **VGG**: smaller 2D kernels(3x3) with more convolutional blocks to induce more non-linearity and so more degree of freedom for the network
- **GoogleNet** (Inception):

Inception module:

- 2D convolutions with different kernel sizes process the same input and then are concatenated
- multi-level feature extraction at each step: general features captured by 5x5 at the same time with local ones captured by 3x3
- additional intermediate classification tasks to inject gradient in intermediate layers ...

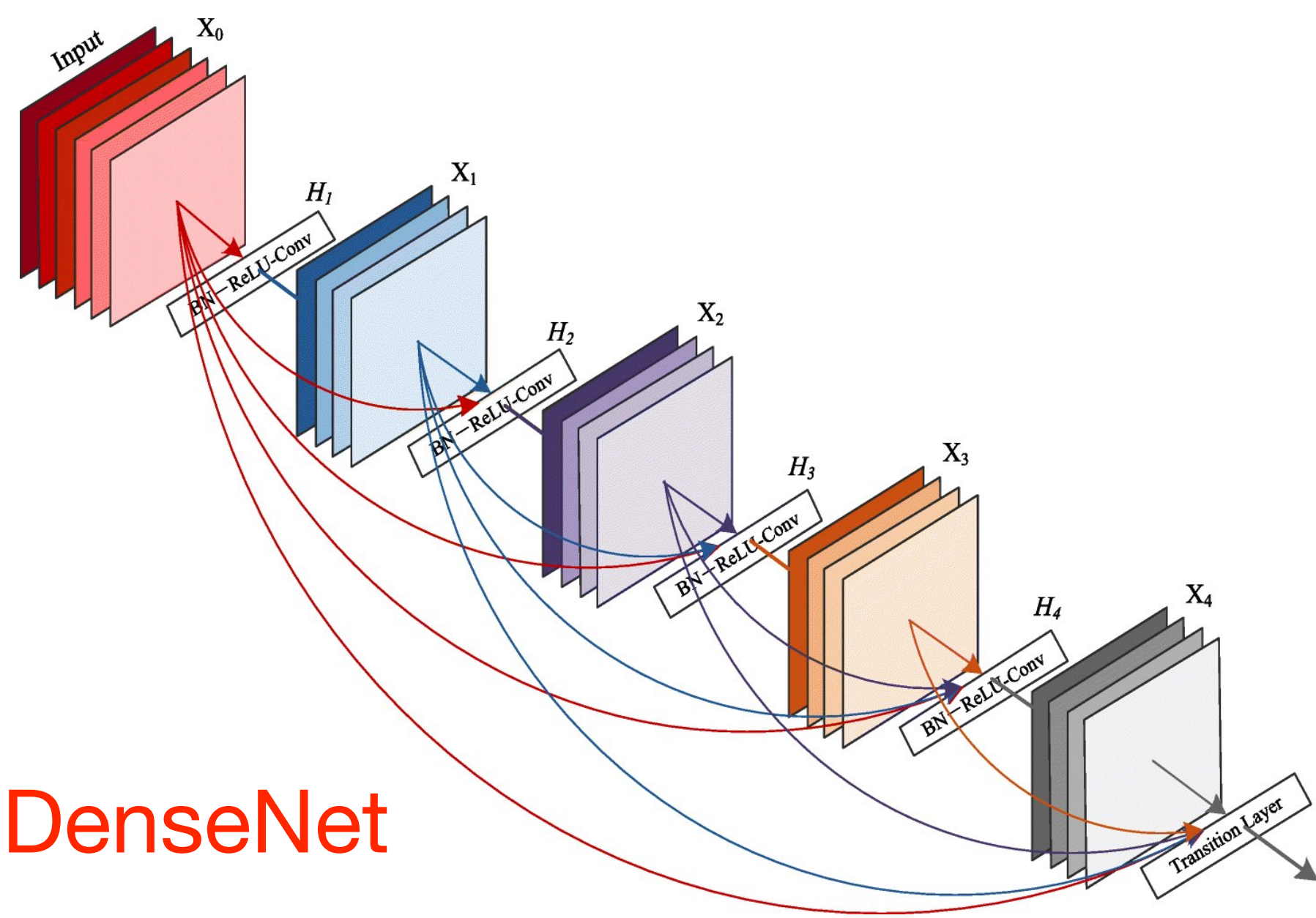


RESNET, DENSENET, XCEPTION

going deeper increase the vanishing gradient problem residual learning in **ResNet** help avoiding it, moreover each block learns the residual wrt the identity (easier task)

evolutions of the idea:

DenseNet: connect entire blocks of layers to one another helps in identifying and use of diverse representations as we go deeper ...

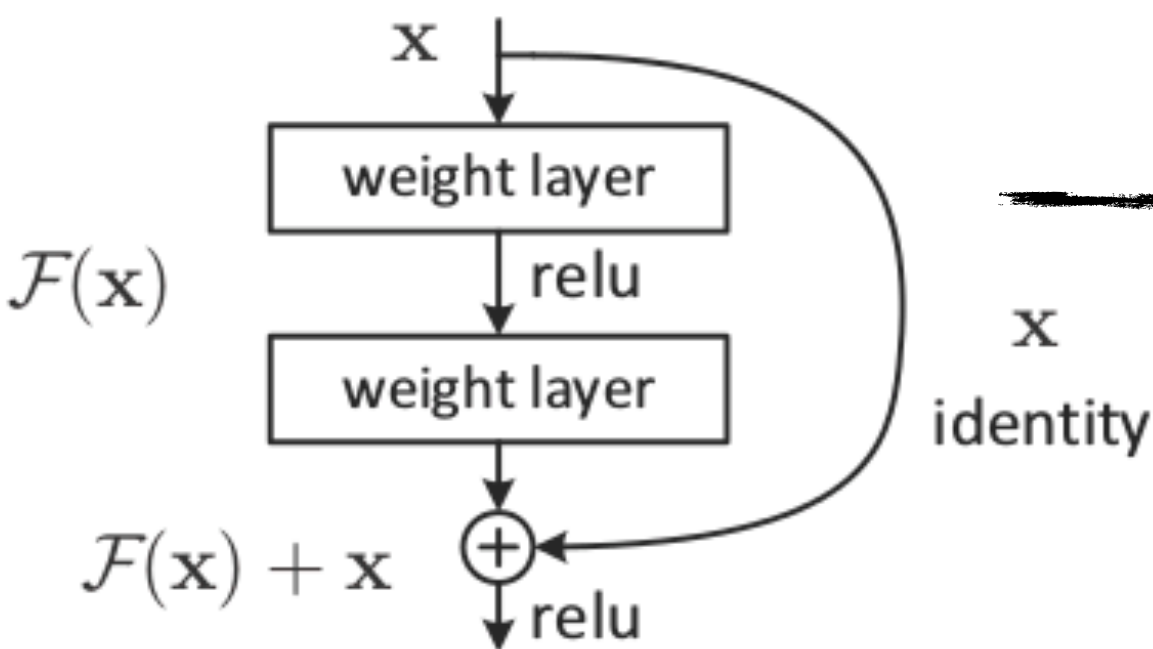
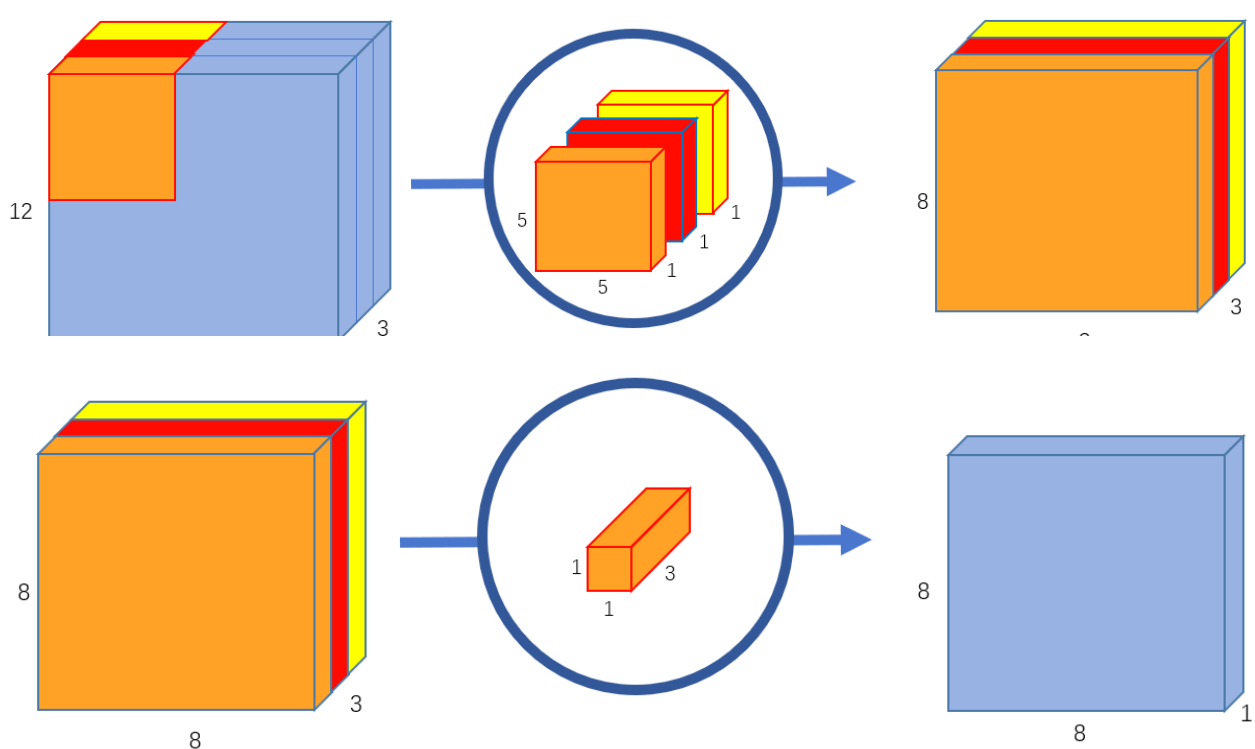


DenseNet

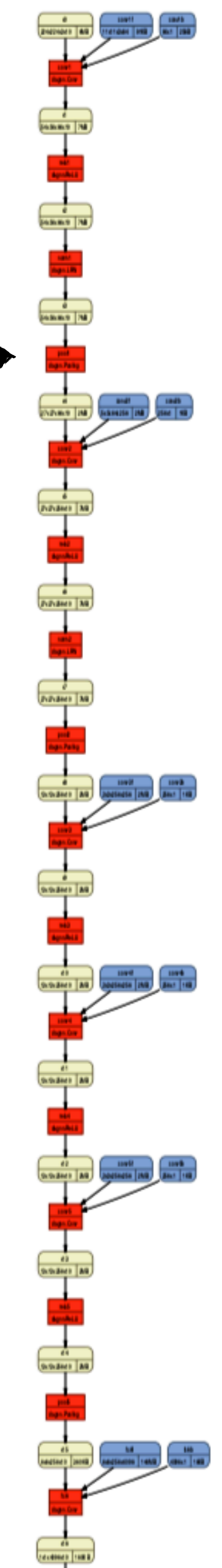
Xception = Inception + ResNet: same parameters as InceptionV3 but better performance ...

leverage Depthwise Separable Convolutions

Conv2D (3x3) \rightarrow DSConv2D: 3x1 + 1x3



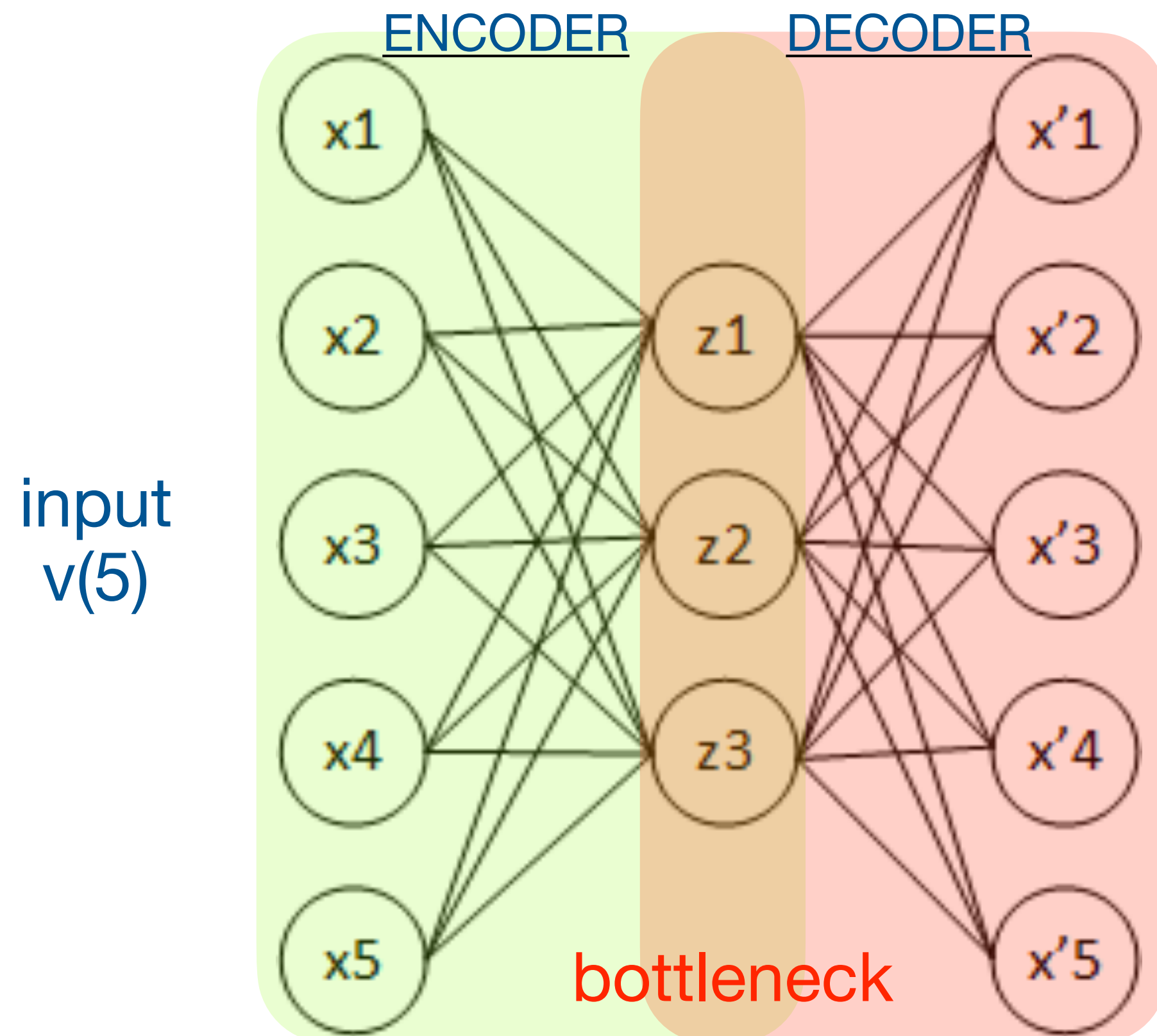
ResNet-152
60 MPar



... 152 layers
32

ANN ARCHITECTURES FOR UNSUPERVISED REPRESENTATION LEARNING: AUTOENCODERS

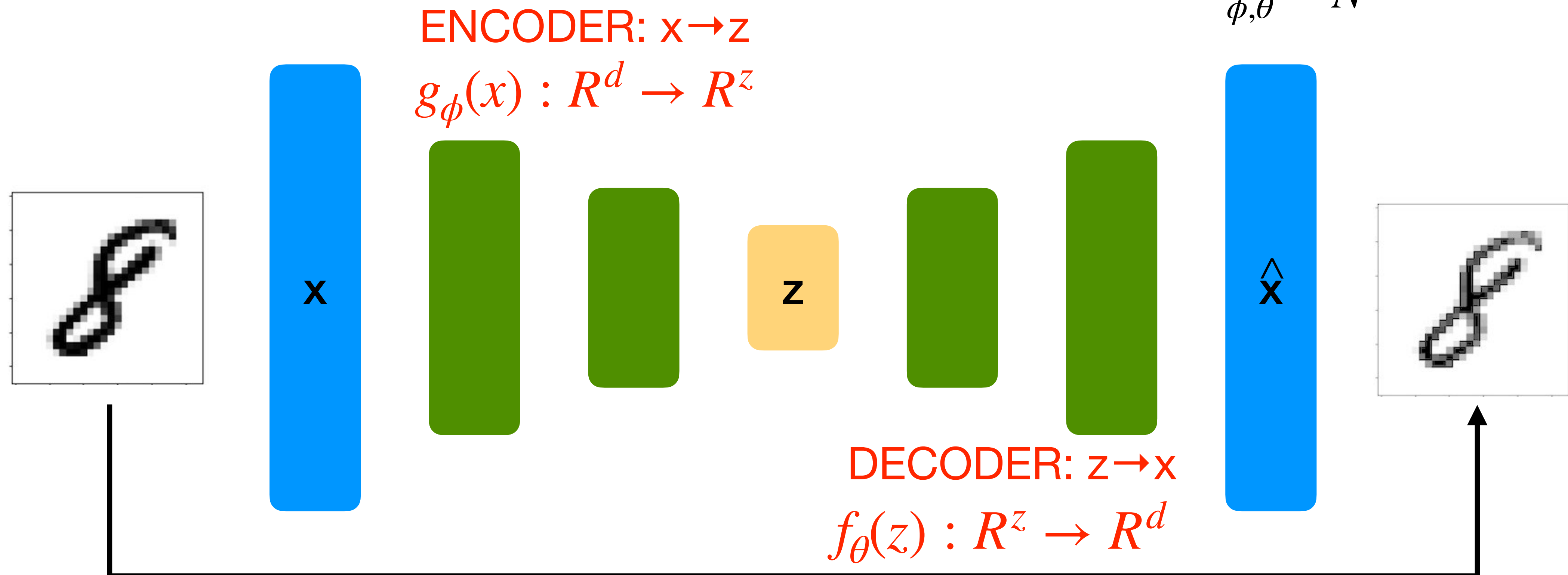
- non-supervised algorithm that try to identify common and fundamental characteristic in the input data
- combines and **encoder** that converts input data in a different representation, with a **decoder** that converts the new representation back to the original input
- trained to output something as close as possible to the input (i.e to learn the identity function)



- “trivial” unless to constrain the network to have the hidden representation with a smaller dimension of the input/output
- in such case the network build (learn) “compressed” representations of the input features: $x \in \mathbb{R}^5 \rightarrow z \in \mathbb{R}^3$

AUTO-ENCODER IMPLEMENTATION

$$\begin{aligned}\phi^*, \theta^* &= \arg \max_{\phi, \theta} \frac{1}{N} \sum L(x^{(i)}, \hat{x}^{(i)}) = \\ &= \arg \max_{\phi, \theta} \frac{1}{N} \sum L(x^{(i)}, f_{\theta}(g_{\phi}(x)))\end{aligned}$$



trained so that: Output \simeq Input

$$L(x, \hat{x}) = ||x - \hat{x}||^2 \quad \text{or} \quad L(x, \hat{x}) = - \sum_D [x_k \log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k)]$$

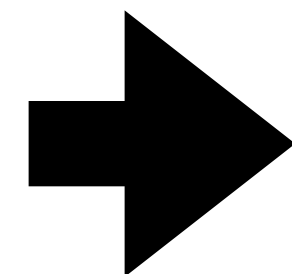
NOTE: L do not depends on dataset labels (unsupervised learning)

AE: RECONSTRUCTION QUALITY

Original images
(ground truth)

2D latent space

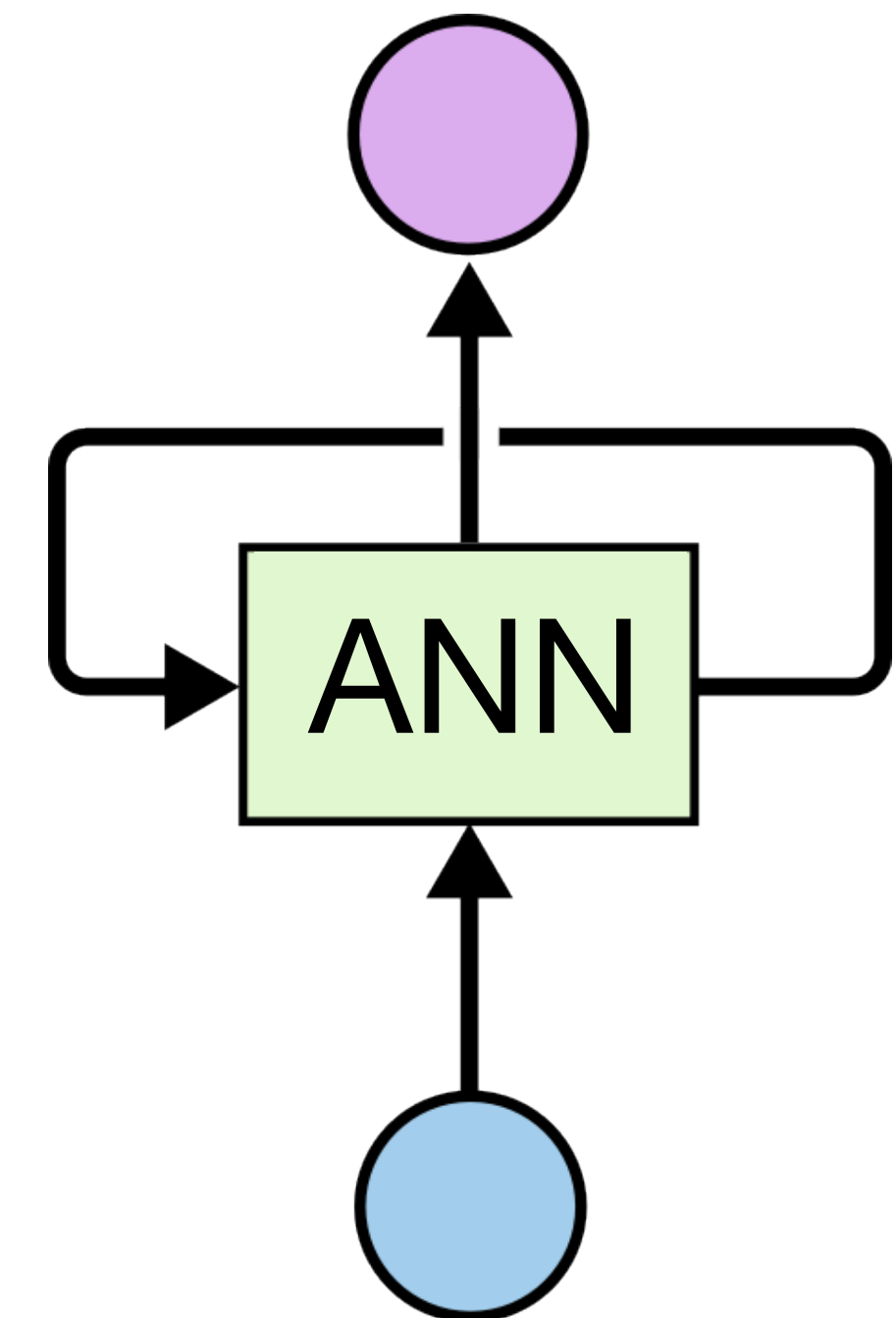
5D latent space



- latent space acts as a "compressor" of information, a certain level of smoothing (inform. loss) is inevitable
- most important limitation: the learned latent space is a non-continuous representation and does not allow interpolations and / or to structure the space appropriately, i.e. cannot be used to generate events (for this scope there are specific generative architectures VAE, GAN, Normalizing Flows, Invertible-nets, etc...)

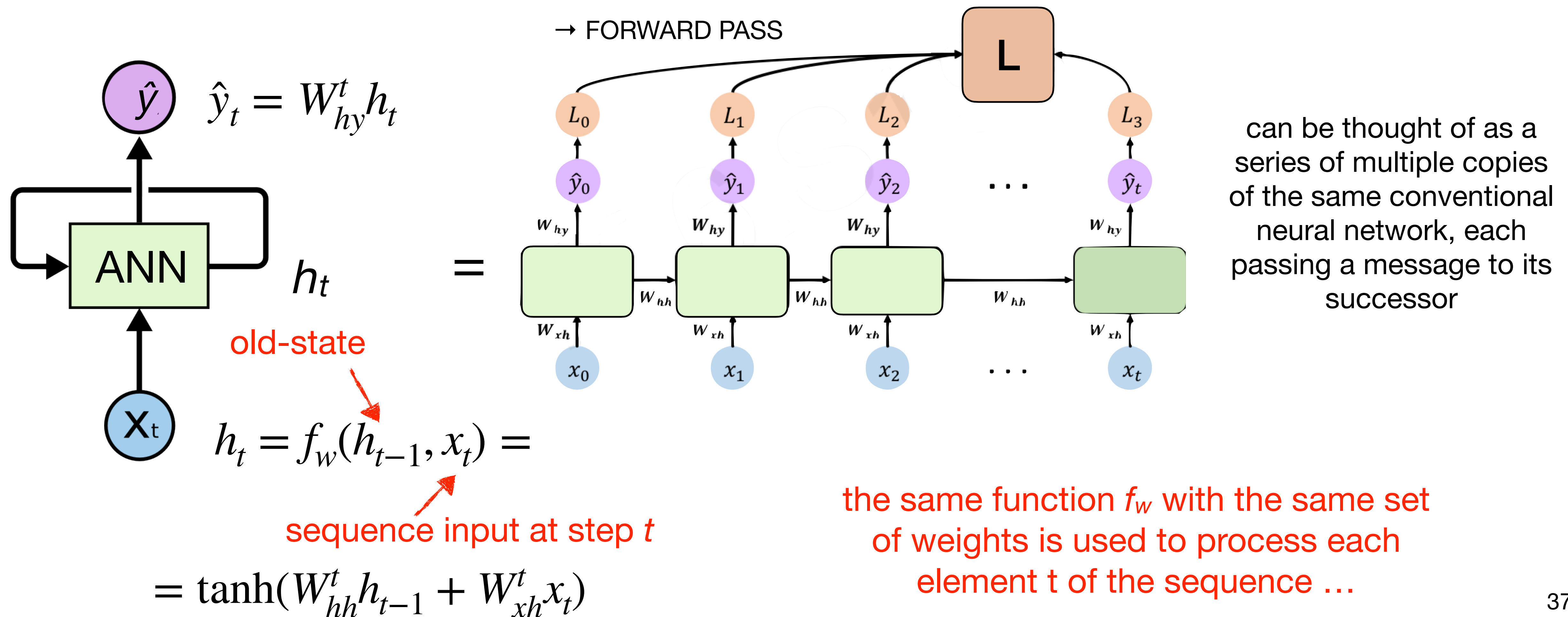
ANN ARCHITECTURES FOR SEQUENCES: RECURRENT NEURAL NETWORKS

- RNN are specific ANN architectures optimised to identify long-term correlations in sequence of informations of variable lengths (example: natural language processing, signal processing, time series forecasting, etc...)
- typical task for a RNN: given a sequence of features (text, music, ... a list of charged tracks parameters), predict one or more targets (the next word on a phrase, the weather in the next 24h, the flavour of a hadron jet in an hep experiment, ...)
- a RNN should be able to:
 - take in input sequences of variable length
 - be able to keep track of dependences between elements that are distant in the sequence
 - be able to keep information on the order between the elements of the sequence
 - have shared parameters (weights) so that identified correlations between elements can be transferred in diverse points of the sequence

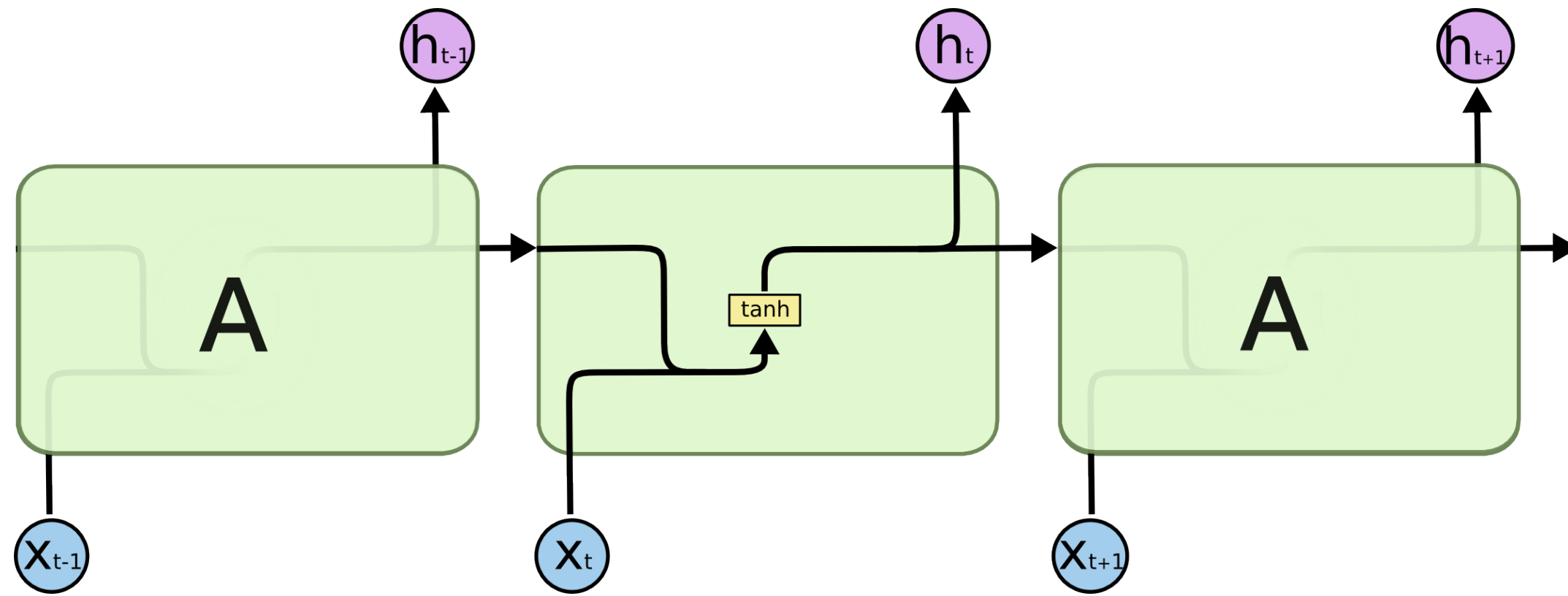


RNN IMPLEMENTATION

- a RNN processes the input in a loop (recurrent connection) that allows the persistence of the informations during the entire processing of the sequence's elements
- base module: A is a NN that analyse the t element of the input sequence x_t and produce the output h_t (**hidden state**). h_t is passed to the same network during the processing of the next element of the sequence

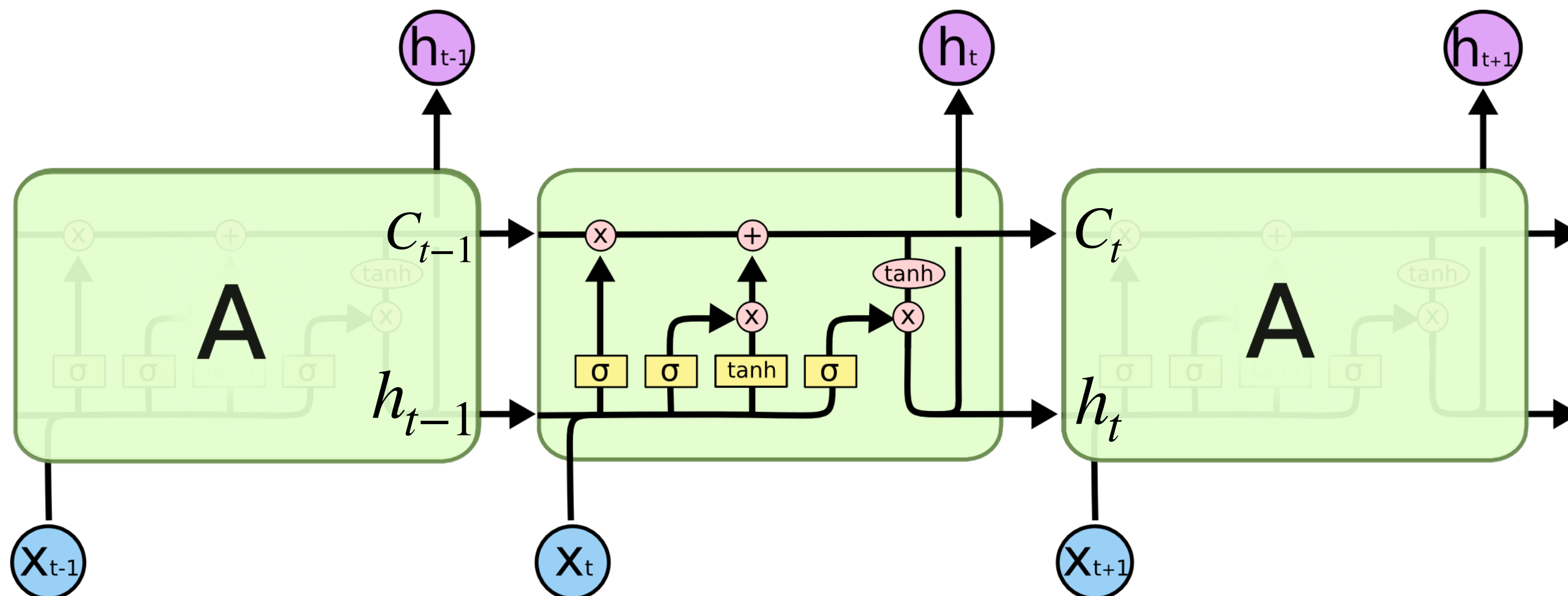


RNN AND LONG TERM DEPENDENCIES



- in RNNs unbounded activations (like ReLU) cannot be used as they create instabilities
- tanh or sigmoid are OK suffer vanishing of the gradient

problem solved in **LSTM** RNN (Hochreiter, '97) with a "software trick": instead of having a single neural layer, it has four, which interact in such a way to implement a sort of parallel data-flow which at each step t makes the previous data available to each layer of the network w/o being affected by gradient dilution

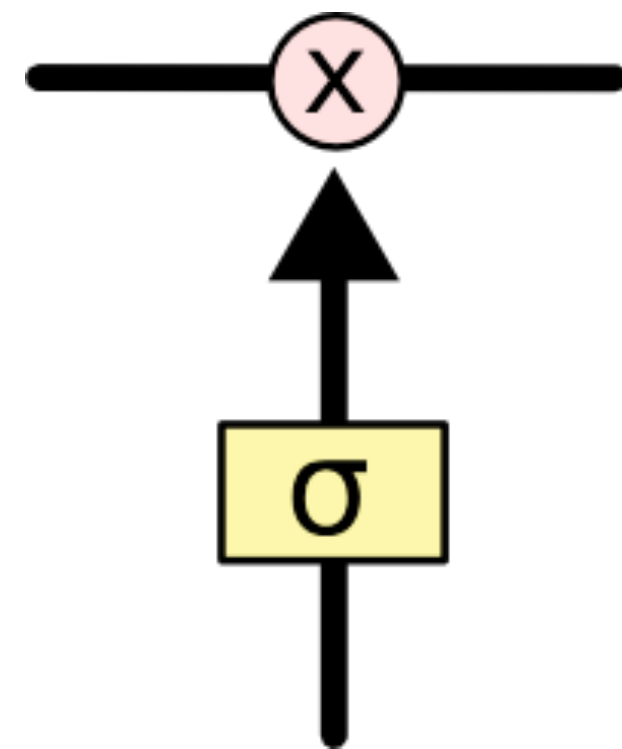
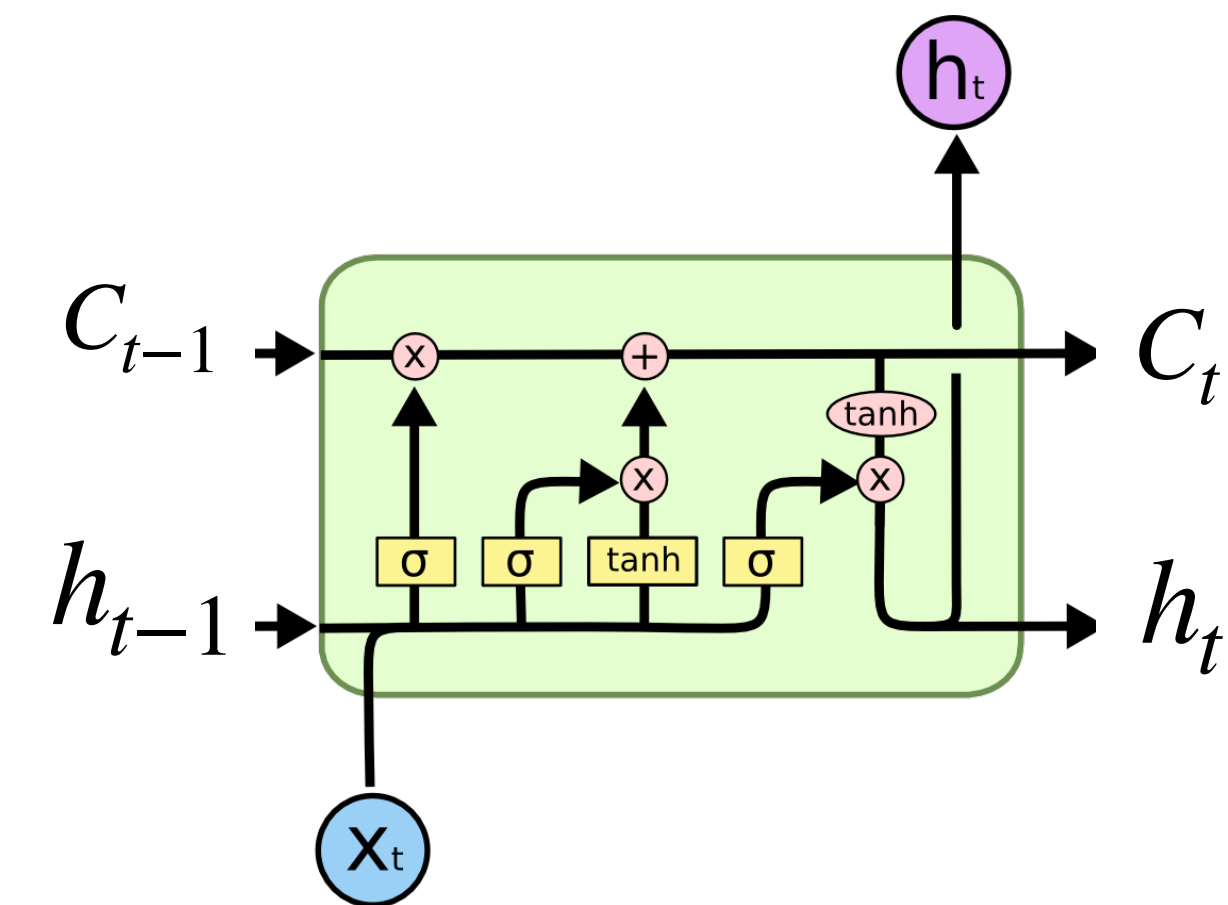


key element: **cell-state C_t**

is a memory units ("conveyor belt") to which is possible to add or subtract information using "gate" structures

LONG SHORT TERM MEMORY NETWORKS

gate: NN-layer with sigmoid activation and a point-wise multiplication



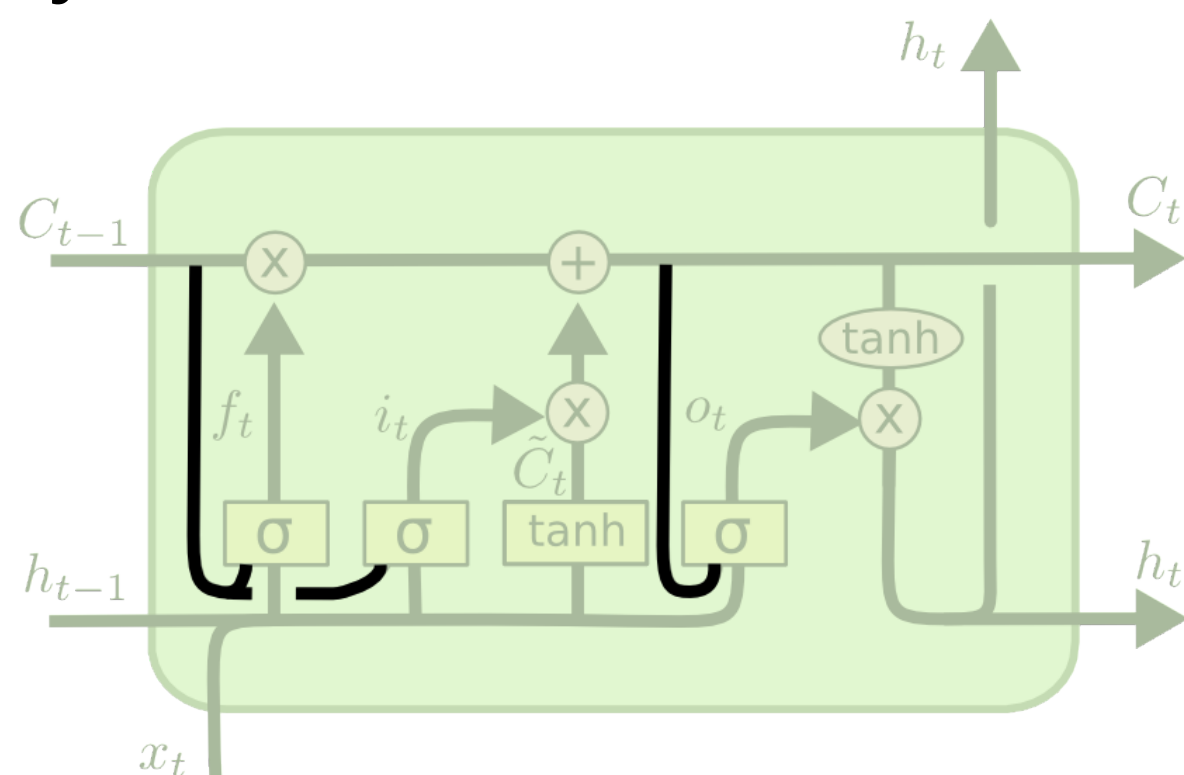
output $\in [0, 1]$:

every LSTM has 3 gates:

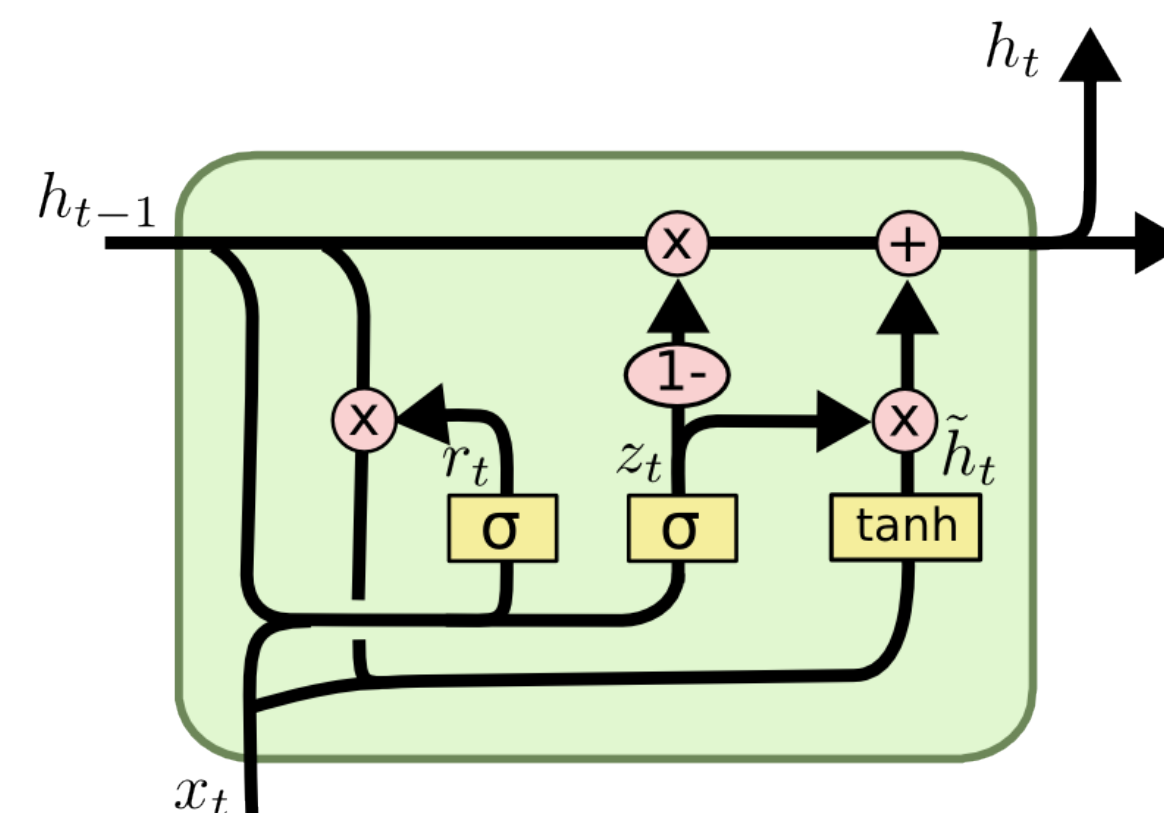
- f: forget gate (controls deleting from the cell-state)
- i: input gate (controls writing on the cell-state)
- o: output gate (controls the output on h_t)

the backprop from $C_t \rightarrow C_{t-1}$ doesn't requires multiplications for tanh/sigmoid \rightarrow no gradient dilution ...

every publication implementing a LSTM has used a slightly different version of the original algorithm, so you'll find it with different names ...



LSTM with "peephole":
gate layers can see the cell-state

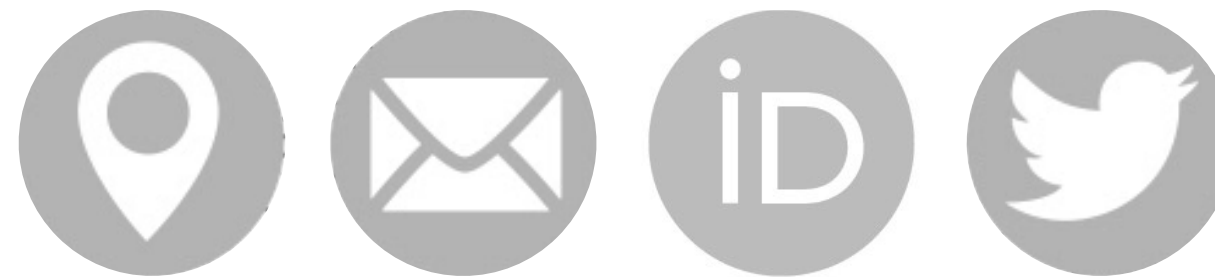


GRU (Gate Recurrent Unit):
combines the gates and unify hidden state with cell-state to simplify model and number of parameters (one of the most used RNNs)

KEEP IN TOUCH ...



SCAN ME



ADDITIONAL MATERIAL

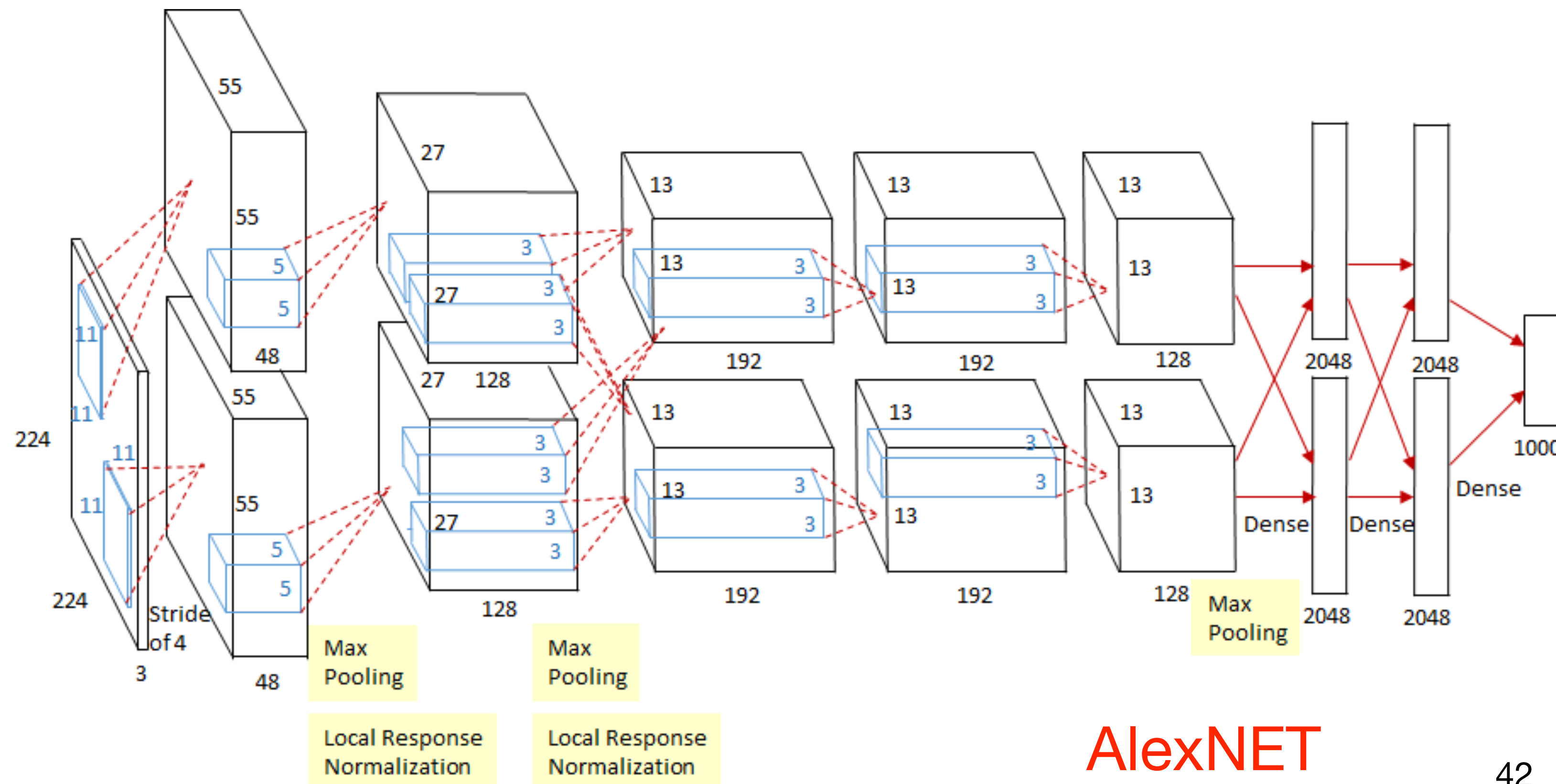
FIRST HIGH PERFORMANCE CNN: ALEXNET

- based on Krizhevsky et Al. architecture winner of the Imagnet 2012 contest
- dev. framework: Caffe (Berkeley Vision Deep Learning framework: <http://caffe.berkeleyvision.org>)

same top-down approach as LeNet with successive filters designed to capture more and more subtle features

+ improvements:

1. better back-propagation via ReLU
2. dropout based regularisation
3. batch normalisation
4. data augmentation: images presented to the NN during training with random translation, rotation, crop
5. deeper architecture: more convolutional layers (7), i.e. more finer features captured



Validation classification

- feature initialised with white gaussian noise
- fully supervised training
- training on GPU NVIDIA for ~1 week
- 650K neurons
- 60M parameters
- 630M connections

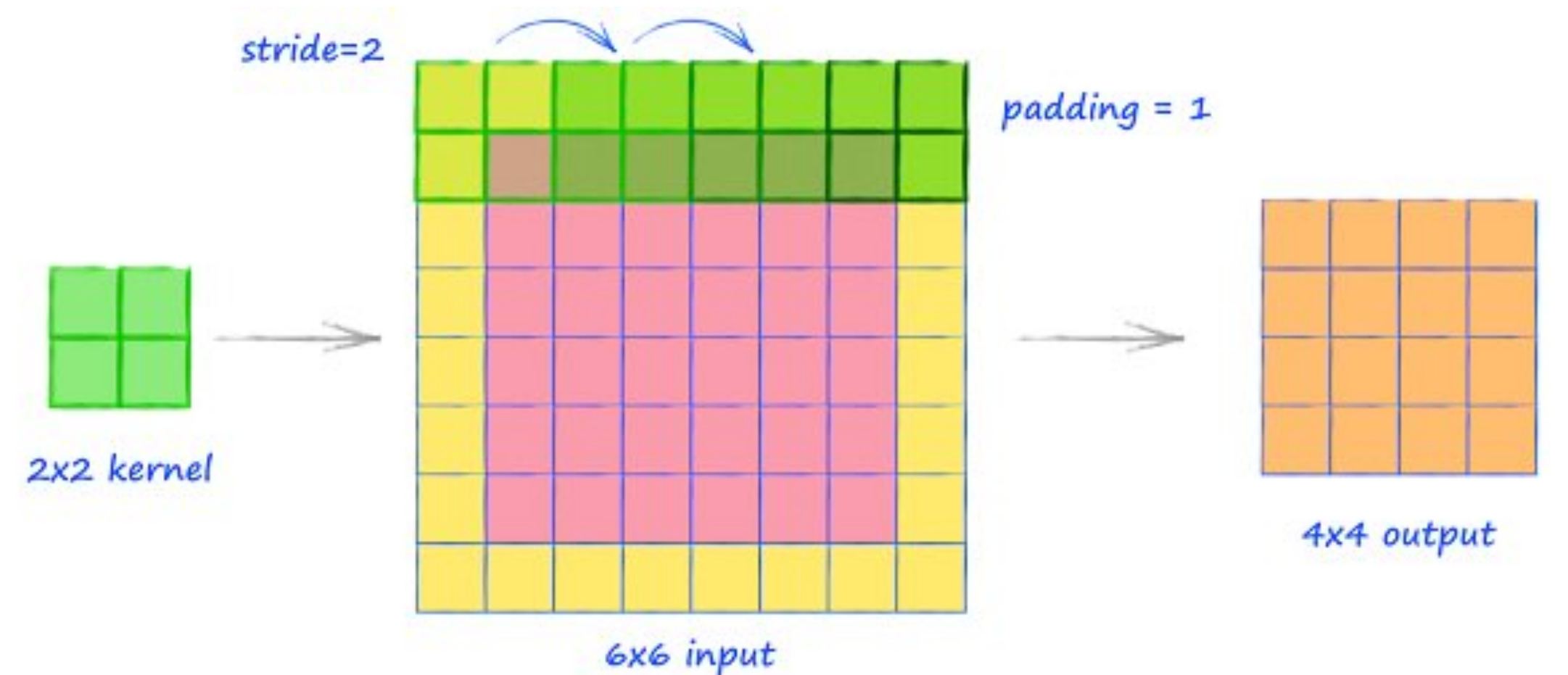
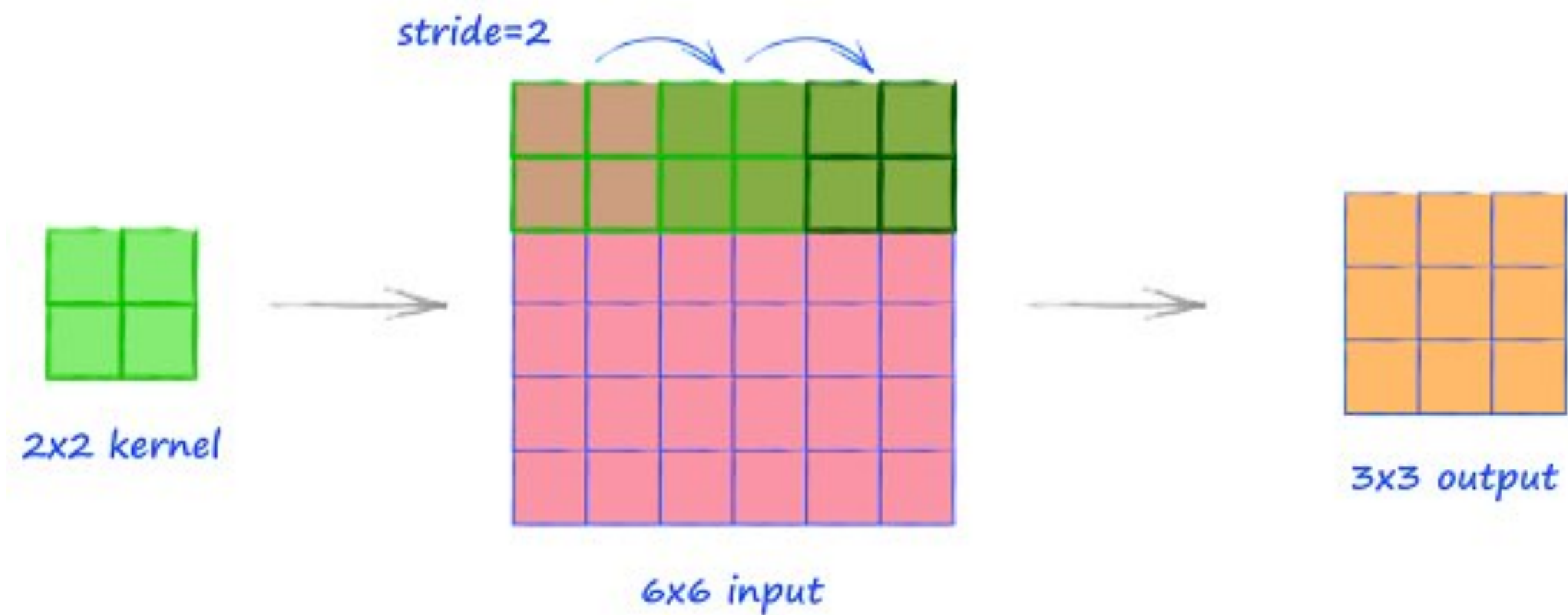
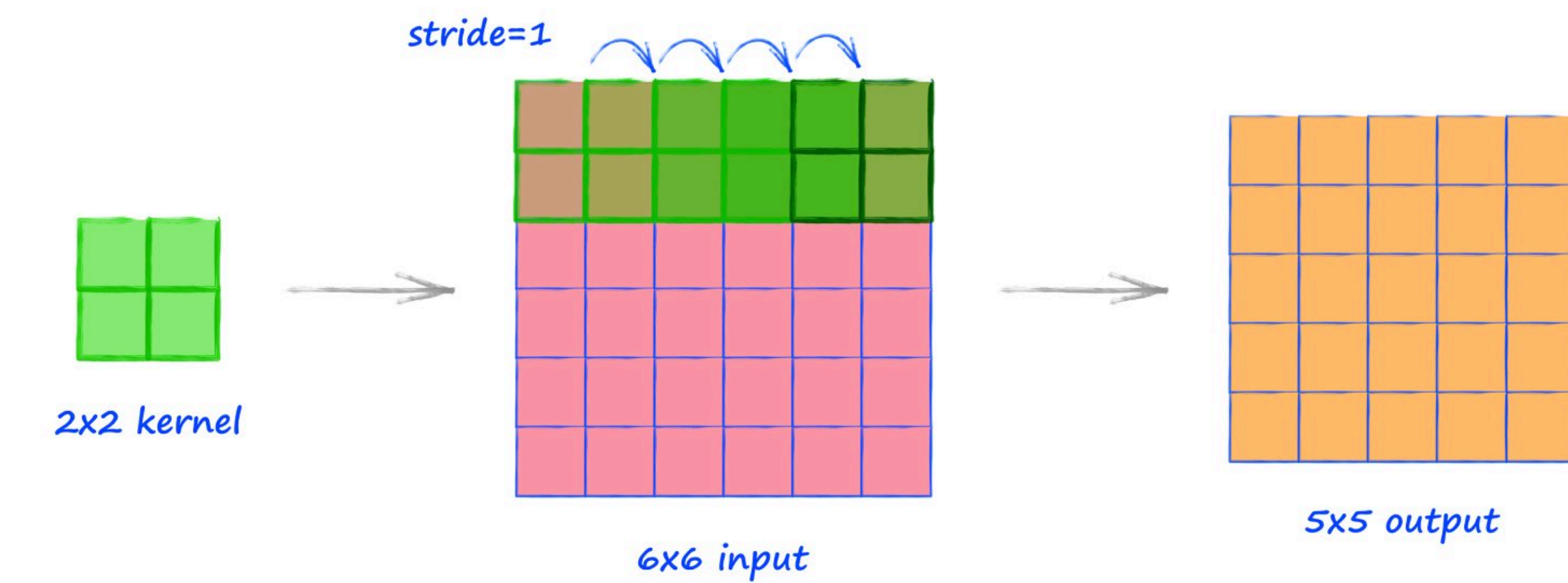


example: standard convolution2D with kernel (2,2)

stride 1, no padding:
input 6x6 \rightarrow output 5x5

stride 2, no padding:
input 6x6 \rightarrow output 3x3

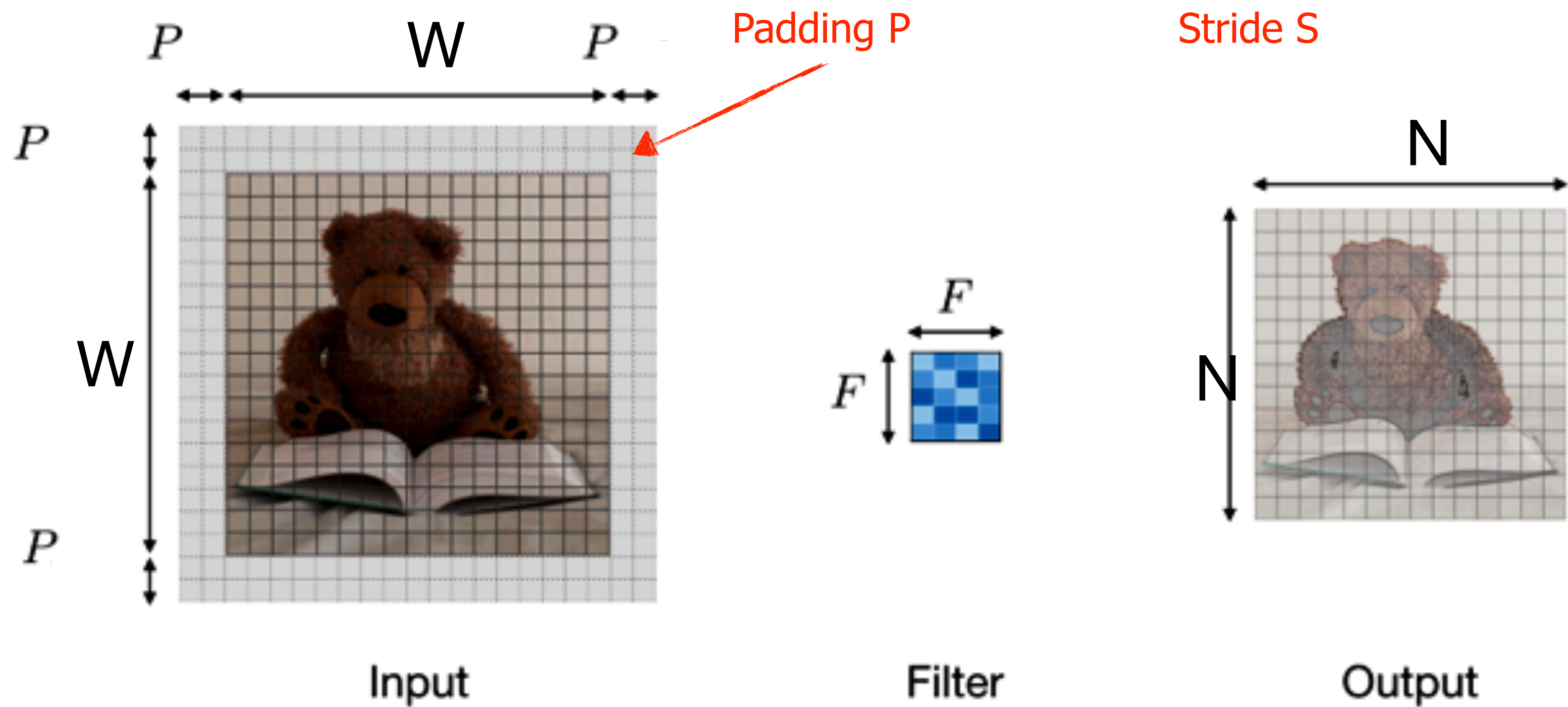
stride 2, padding 1:
input 6x6 \rightarrow output 4x4



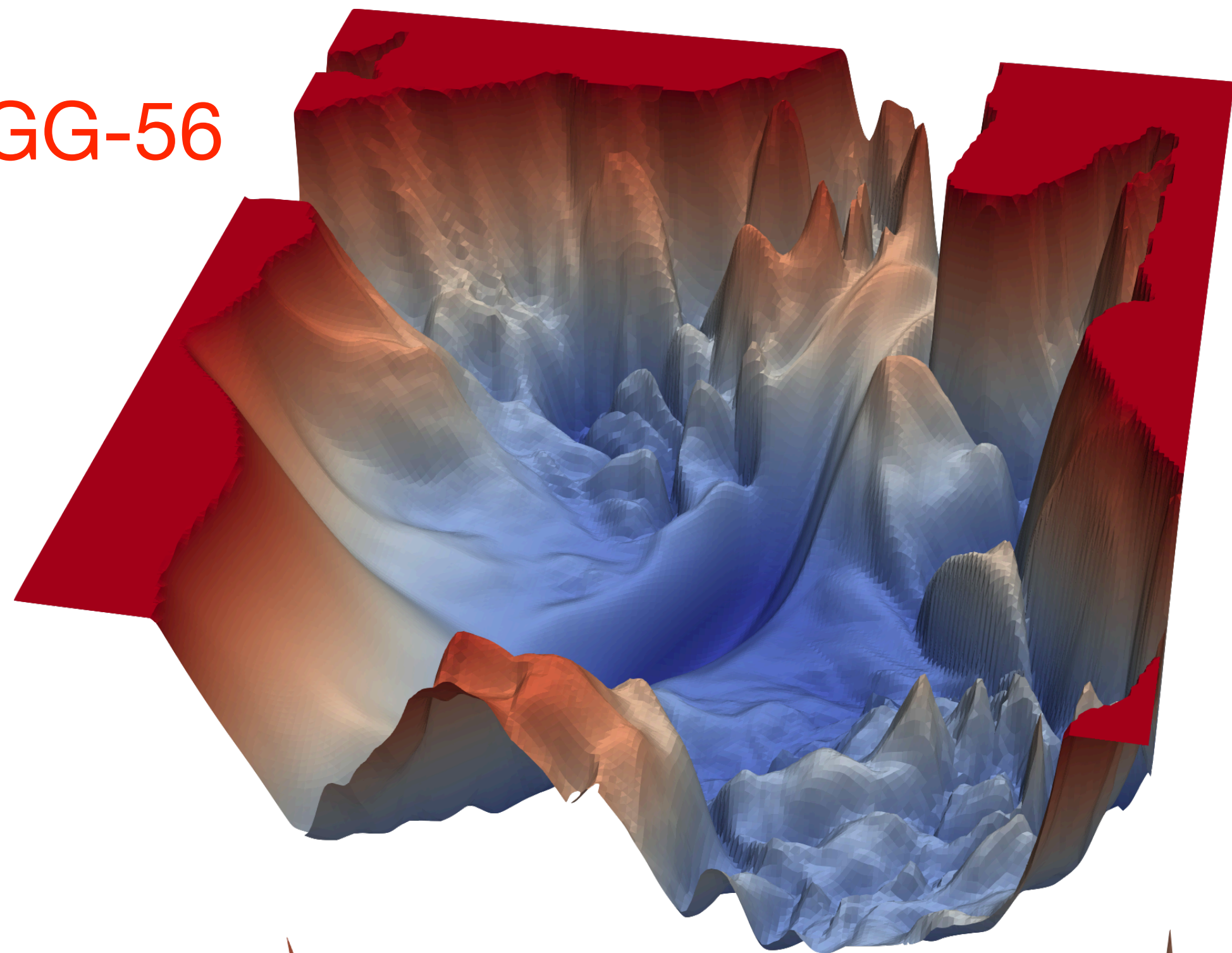
OUTPUT AFTER THE APPLICATION OF THE CONVOLUTIONAL FILTER ...

Size of the output after applying the filter:

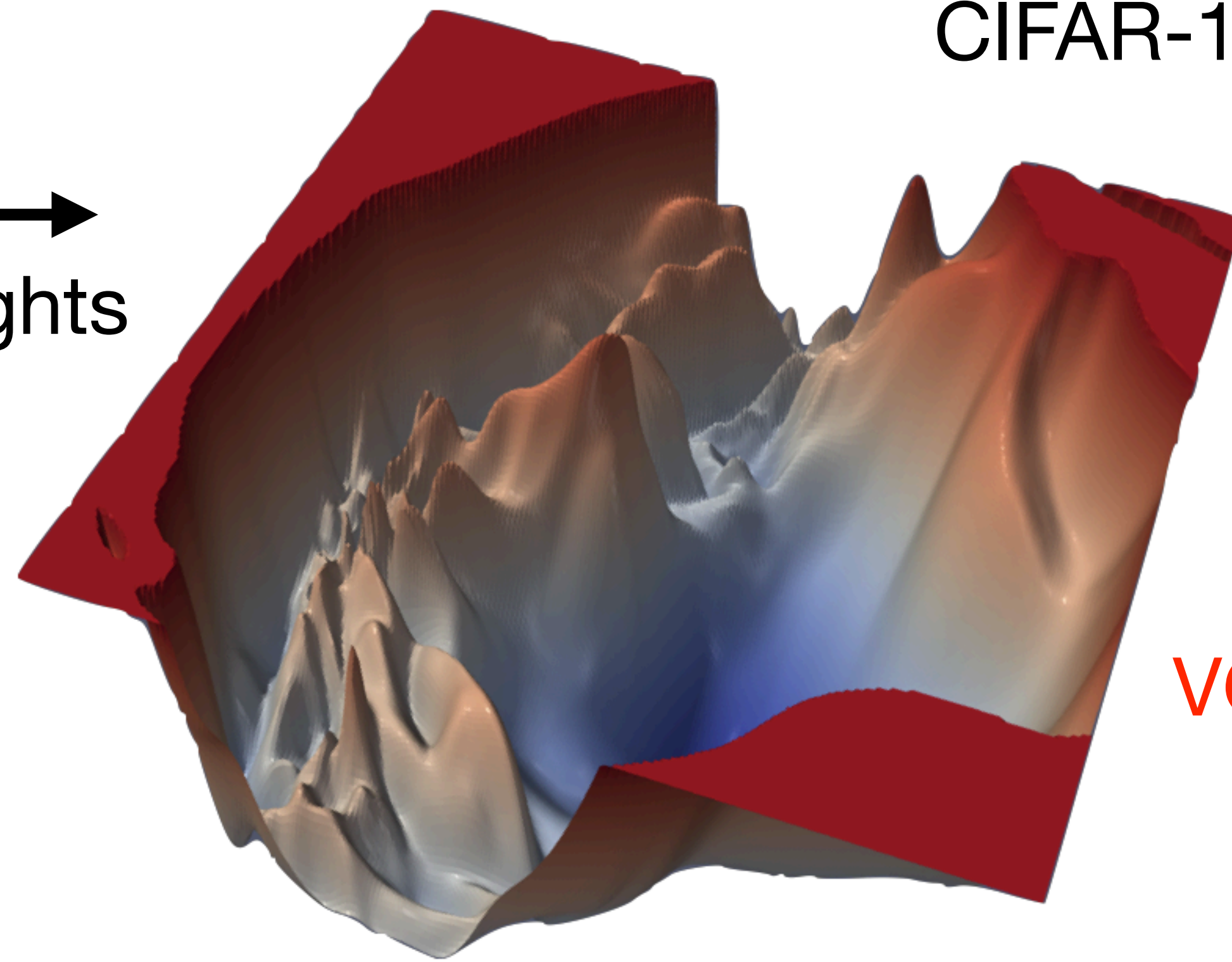
$$N = \frac{W - F + 2P}{S} + 1$$



VGG-56



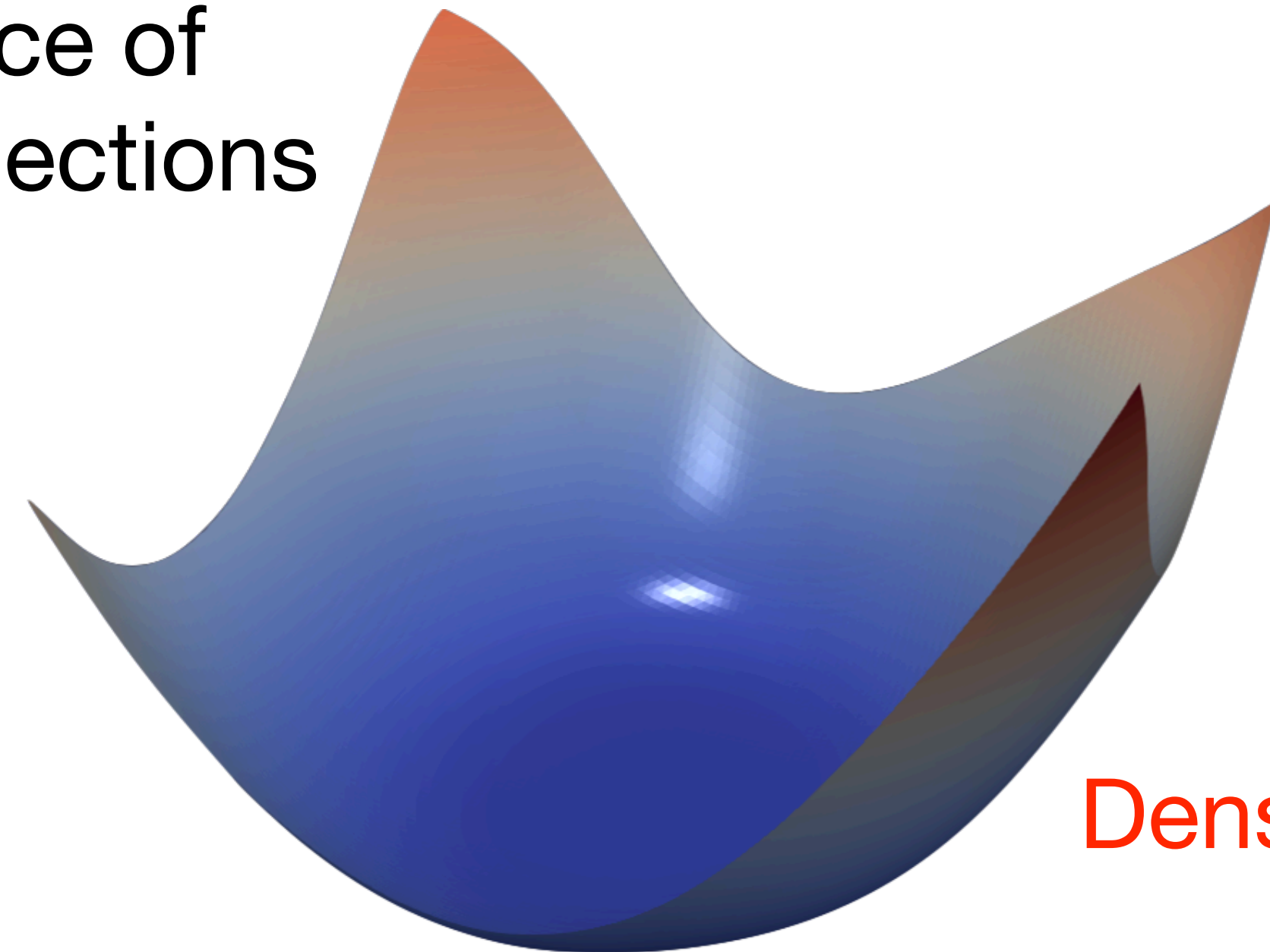
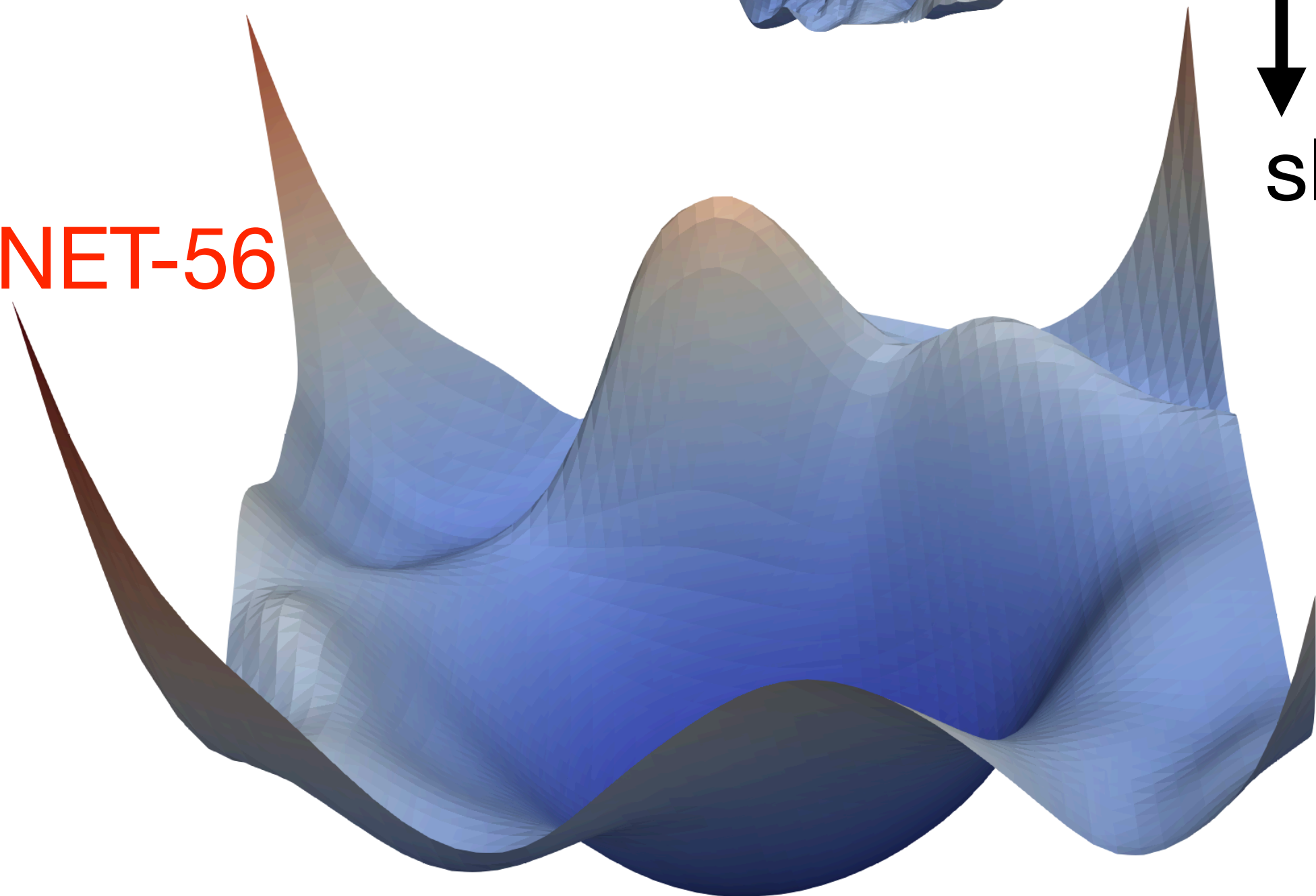
→
#weights



VGG-110

↓
presence of
skip connections

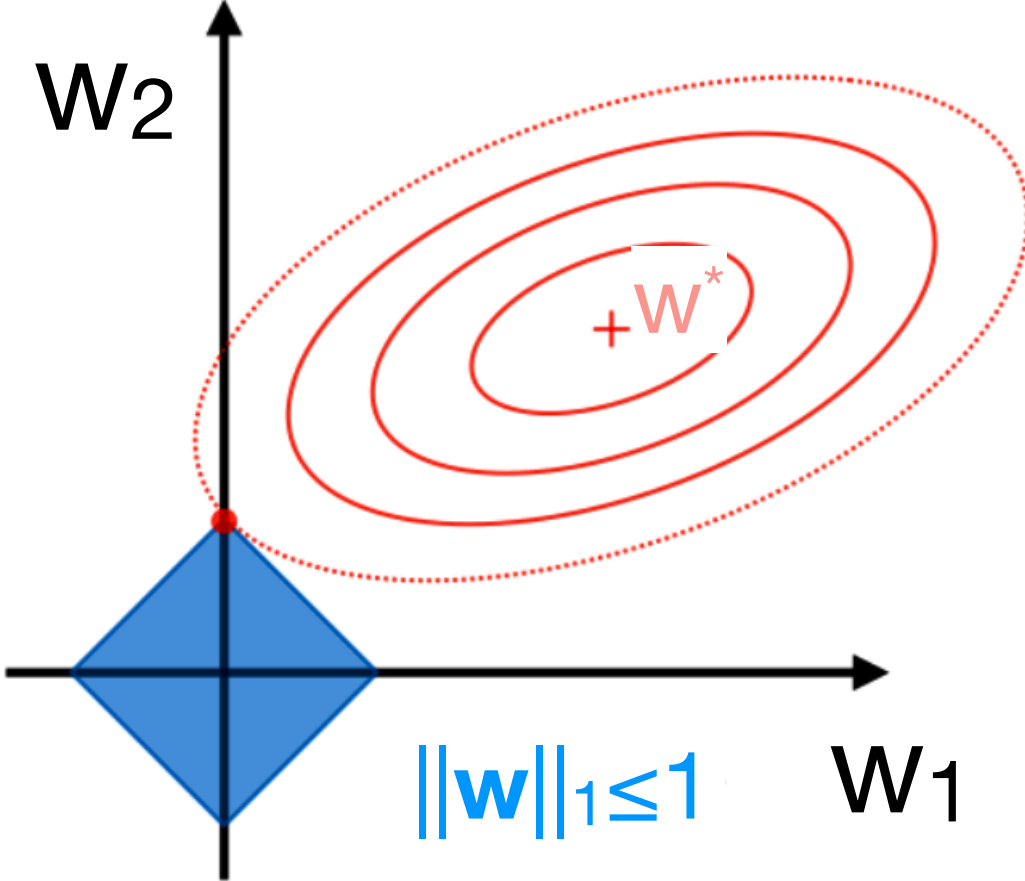
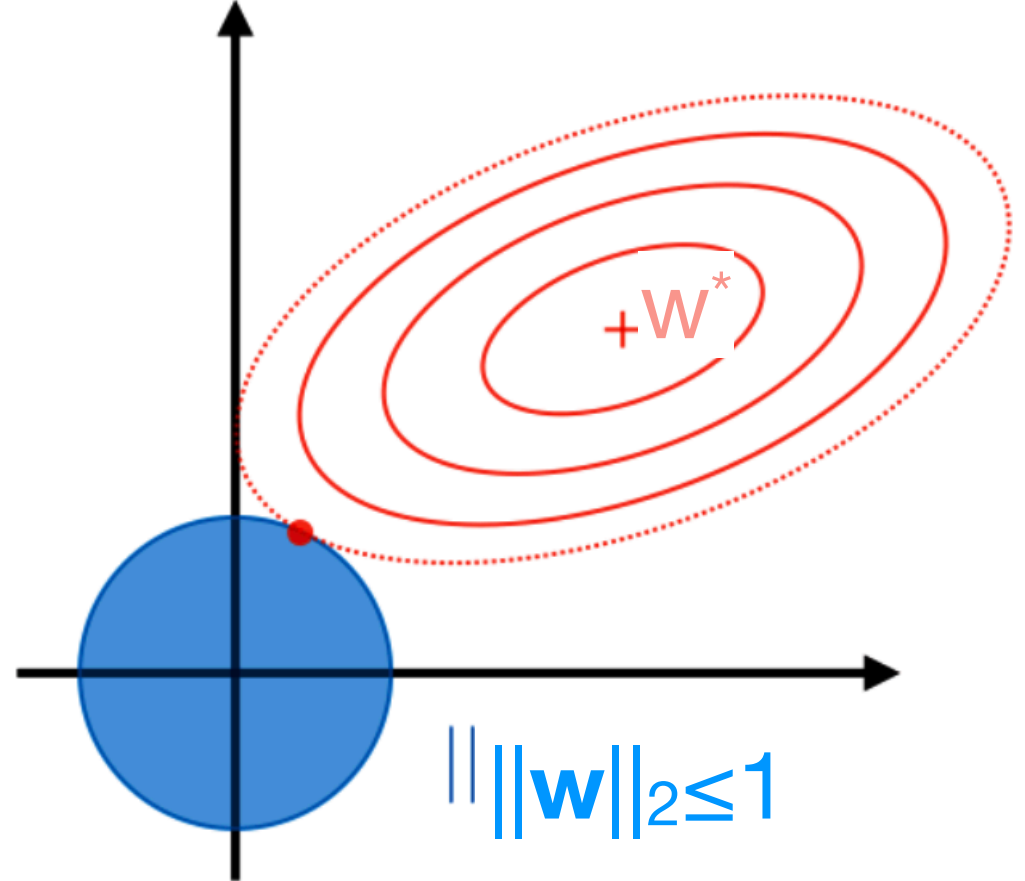
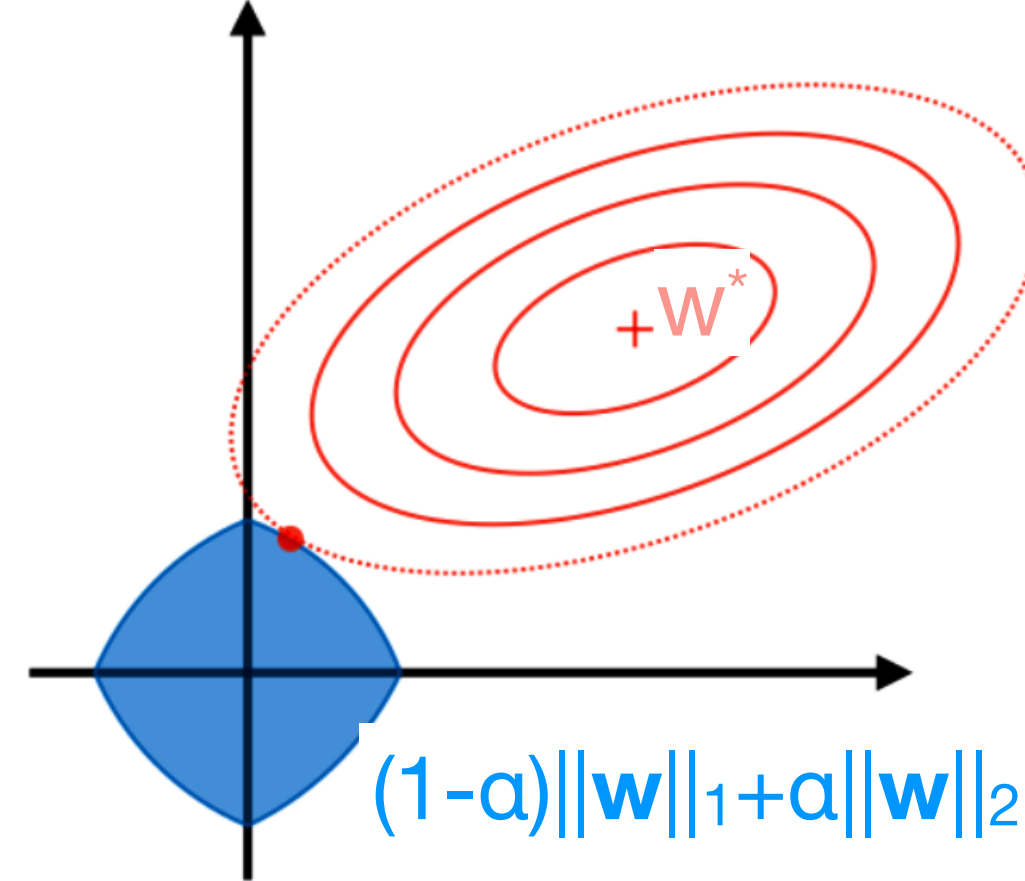
ResNET-56



DenseNET-121

L1/L2/L3 REGULARIZATION

- idea: constrain the complexity of the model by penalizing large values of the weights, unless it is strongly requested by the data itself
- method: a penalty is added to the loss function: $L(\mathbf{w}) \rightarrow L(\mathbf{w}) + \Omega(\mathbf{w})$

L1 LASSO	L2 weight decay	L1+L2 Elastic Net
<ul style="list-style-type: none"> Shrinks coefficients to 0 Good for variable selection 	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		

$$\Omega(\mathbf{w}) = \alpha \|\mathbf{w}\|_1 = \sum |w_k|$$

$$\Omega(\mathbf{w}) = \frac{\alpha}{2} \|\mathbf{w}\|_2^2 = \frac{\alpha}{2} \mathbf{w}^t \mathbf{w} = \sum w_k^2$$

$$\Omega(\mathbf{w}) = \lambda[(1 - \alpha) \|\mathbf{w}\|_1 + \alpha \|\mathbf{w}\|_2^2]$$

EXAMPLE: GD WITH L2 REGULARIZATION

regularised loss:

$$L_R(\mathbf{w}) = L(\mathbf{w}) + \frac{\alpha}{2} \mathbf{w}^t \mathbf{w}$$

gradient:

$$\nabla_{\mathbf{w}} L_R(\mathbf{w}) = \nabla_{\mathbf{w}} L(\mathbf{w}) + \alpha \mathbf{w}$$

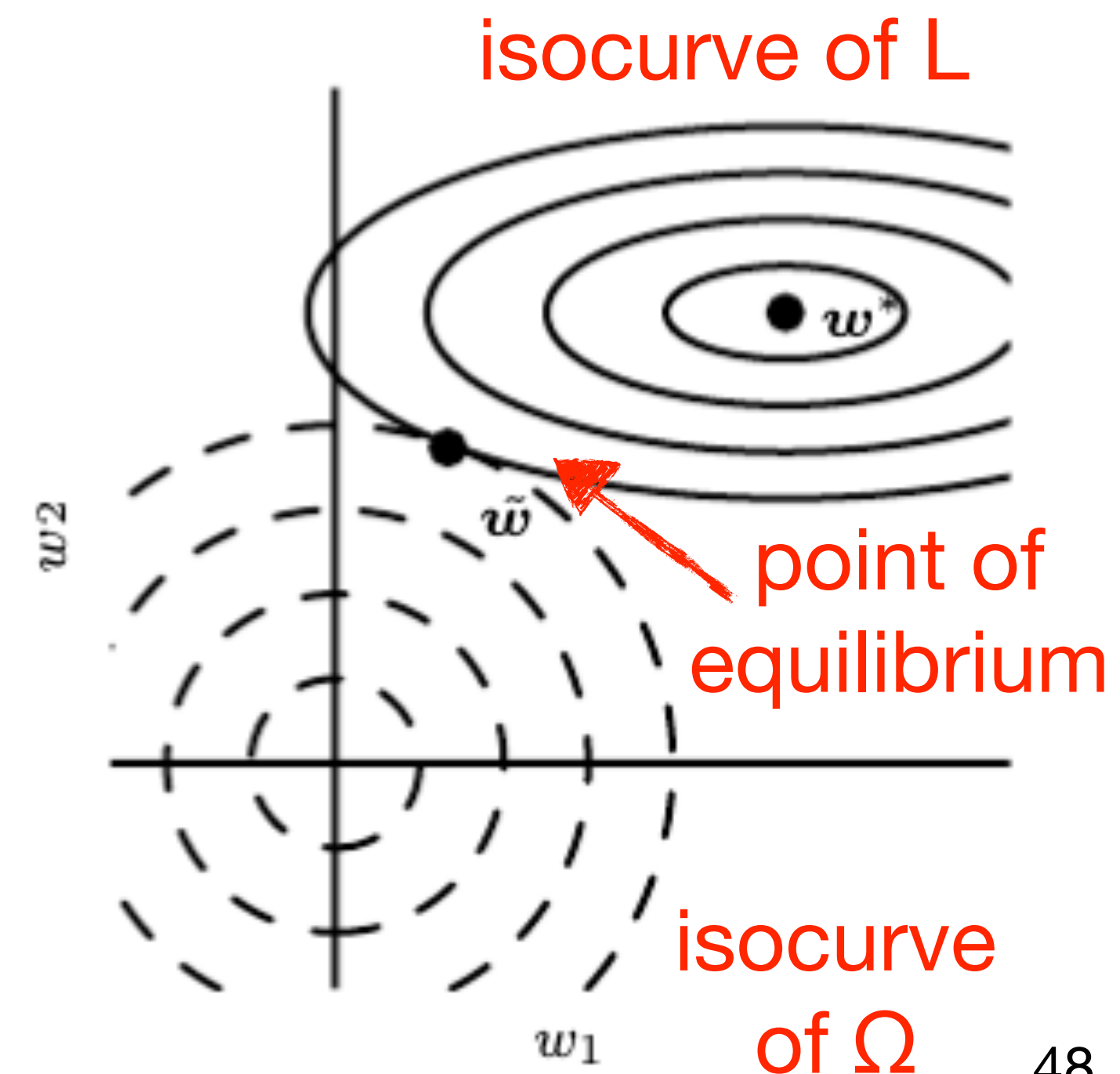
weights update:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\nabla_{\mathbf{w}} L(\mathbf{w}) + \alpha \mathbf{w}] = (1 - \eta\alpha) \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w})$$

the effect of L2 on the single iteration is to reduce the vector of weights for a certain factor

done on all the iterations of the training it can be shown that the overall effect is to rescaling the components of the solution vector \mathbf{w}^* of the non regularised problem proportionally to $\lambda_i/(\lambda_i+\alpha)$, with λ_i the eigenvalues of the Hessian matrix of L

- components in the directions of insensitive \mathbf{w} ($\lambda_i \ll \alpha$) \rightarrow large reducing effect
- components in the sensitive directions ($\lambda_i \gg \alpha$) \rightarrow unchanged



EXAMPLE: GD WITH L1 REGULARIZATION

regularised loss:

$$L_R(\mathbf{w}) = L(\mathbf{w}) + \alpha \sum_i |w_i|$$

gradient:

$$\nabla_{\mathbf{w}} L_R(\mathbf{w}) = \nabla_{\mathbf{w}} L(\mathbf{w}) + \alpha \text{sign}[\mathbf{w}]$$

very different effect wrt the L2 case. Due to the singularity the solution is not easy to find analytically

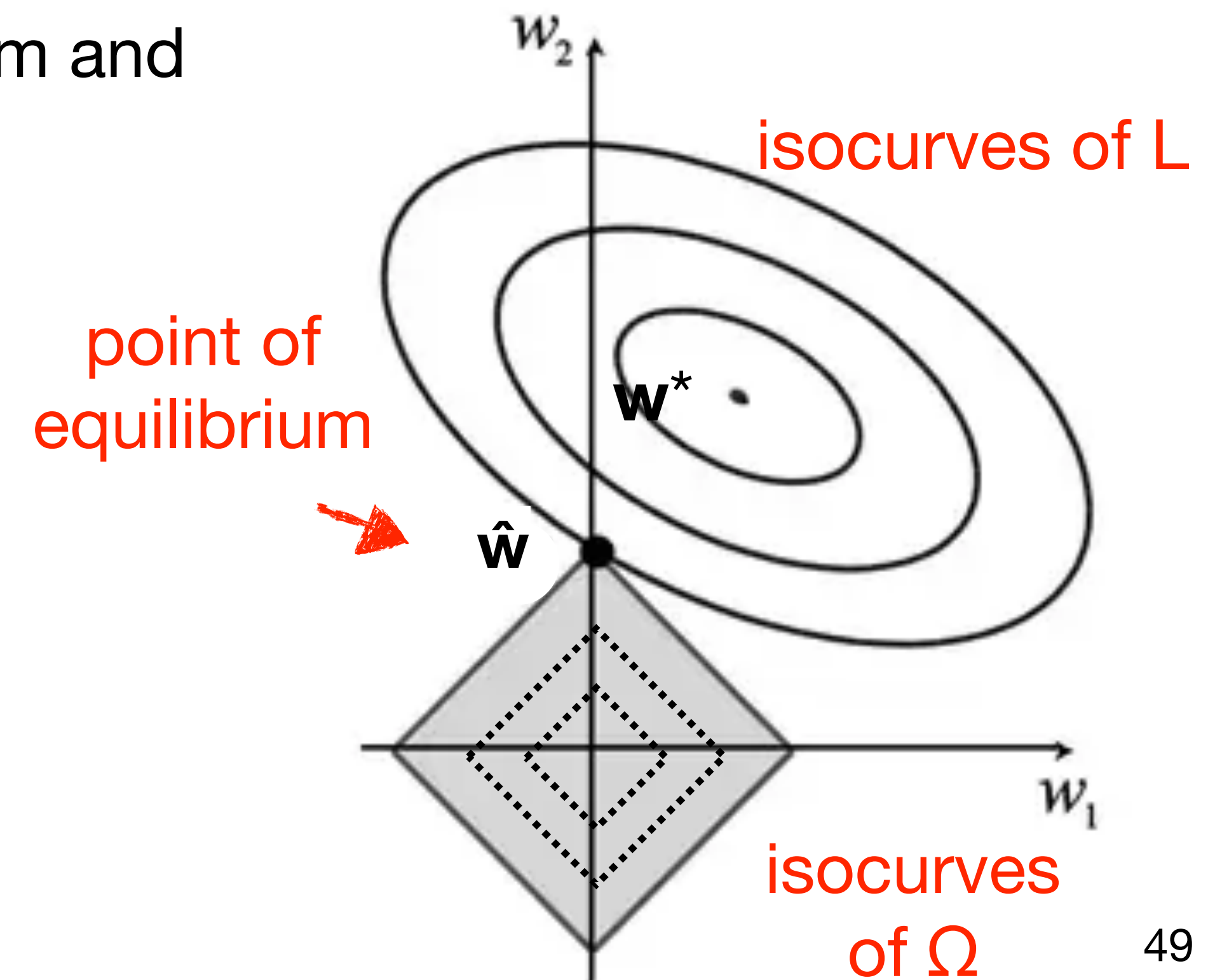
it can be demonstrated that for a loss L with quadratic minimum and diagonal Hessian $H = \text{diag}[\lambda_1 \dots \lambda_n]$:

$$\hat{w}_i = \text{sign}[w_i^*] \max[|w_i^*| - \frac{\alpha}{\lambda_i}, 0]$$

with \mathbf{w}^* solution vector of the non regularised problem

- $w_i^* \leq \alpha/\lambda_i \Rightarrow \hat{w}_i = 0$
- $w_i^* > \alpha/\lambda_i \Rightarrow \hat{w}_i$ scaled by α/λ

L1 acts as a feature selector / sparsifier



GENERALISATION AND BIAS-VARIANCE TRADEOFF

- the choice of the decision boundary is a compromise between:
 - performance of the classifier on the training set ← minimise variance of the training set
 - generalisation capacity of the classifier ← minimise the bias on the test set
- in statistical theory of ML it can be demonstrated the Vapnik inequality:

$$\text{Prob} \left(\underset{\substack{\text{error on an independent dataset (what we} \\ \text{ideally would like to minimize)}}}{R(f)} \leq \underset{\substack{\text{error on training set}}}{R_s(f)} + \sqrt{\frac{h(\log(2N/h) + 1) - \log(\eta/4)}{N}} \right) = 1 - \eta$$

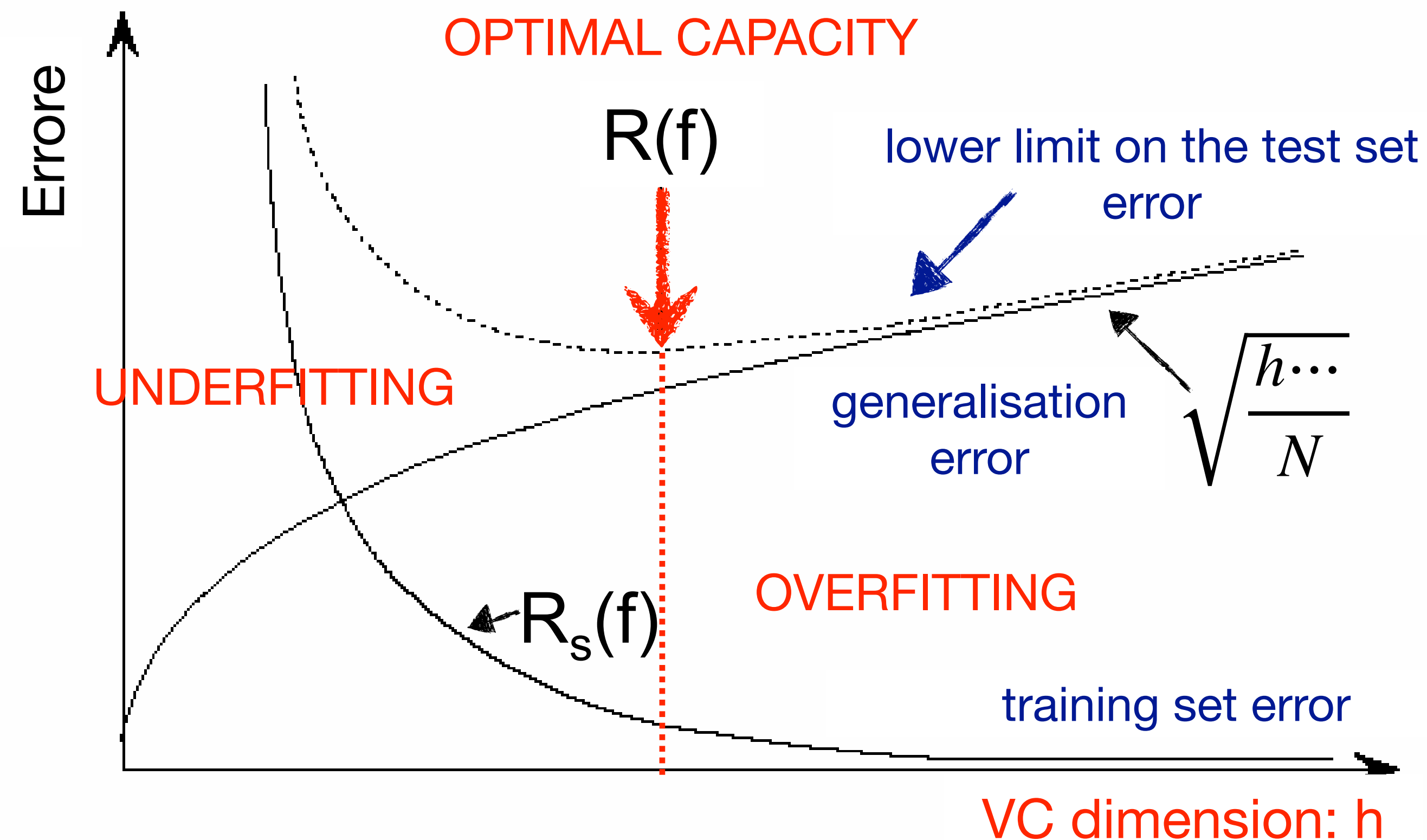
N: dimension of the training set

confidence term (i.e. the generalisation error)

- h: Vapnik-Chervonenkis dimension (VC-dimension)
- is a positive integer that measure the expressive power of the ML model, larger h larger is the capacity of the model to represent complex boundaries

VAPNIK THEORY: MINIMISATION OF STRUCTURAL RISK

bias-variance tradeoff: by using a more complex model (i.e. larger h) able to reduce the variance, we pay this with a larger bias ...



possible
strategies

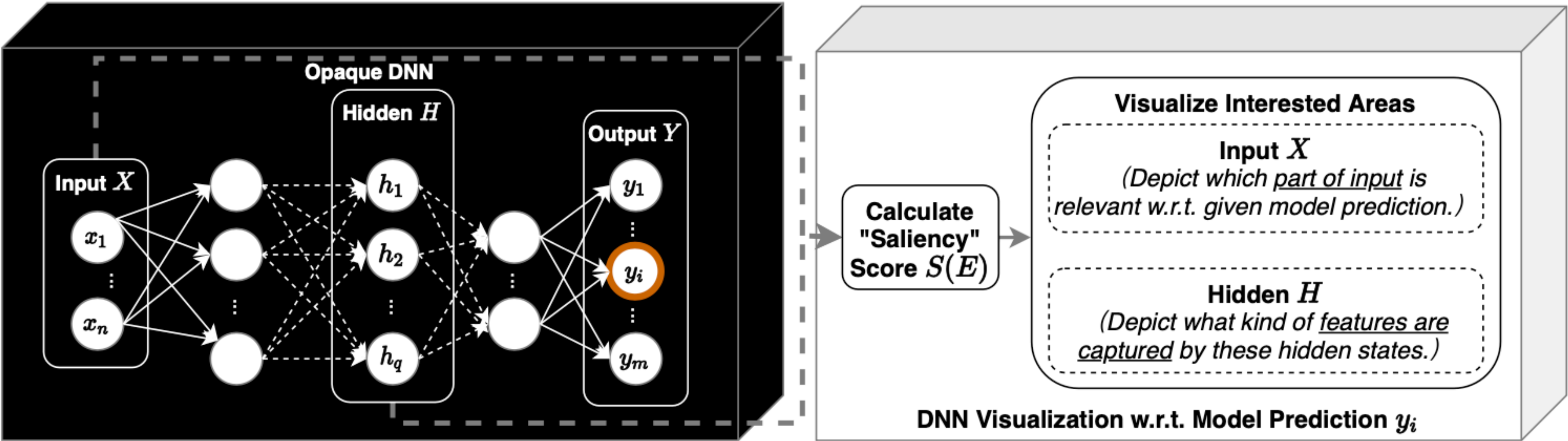
- fix the training set error (for example making it zero), and minimise the confidence term
- choose an appropriate model architectures (i.e. the capacity) and minimise the training set error

SVM, ...
ANN, ...

METHODS TO EXPLAIN DNNs

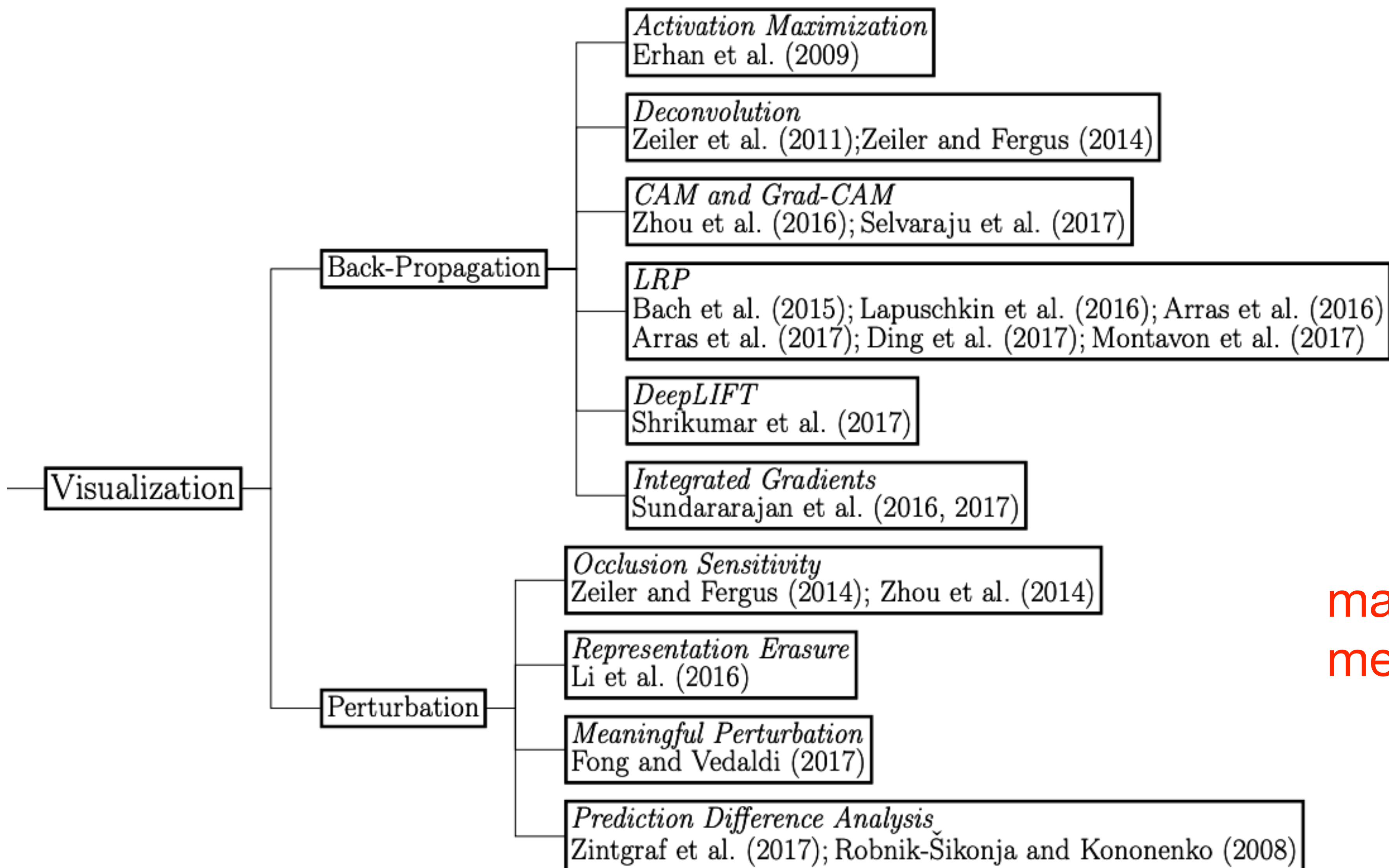
- Due to its apparent black-box nature, it is inherently difficult to understand which aspects of the input data guide the decisions made by a DNN
- There is a research sector expanding these days (xAI / xML where x stands for explainable) whose purpose is to develop useful methodologies to explain the decision-making process of DNNs
- methods for explaining DNNs can be divided into three main groups:
 - **Visualization methods:** they help to understand the correlations between output and input by highlighting, through appropriate methods, the characteristics of the input (of the DNN or of intermediate stages) that strongly influence the output of the network
 - **Synthesis methods:** a separate ML model is developed, a sort of “white box”, trained to mimic the input-output behaviour of the DNN. The white box model, which is intrinsically explainable, aims to identify the decision rules or input characteristics that influence the network outputs
 - **Intrinsic Methods:** they are DNNs created specifically to provide, together with the output, also an explanation of the reason for that output. Intrinsically explainable DNNs simultaneously optimise both model performance and a certain quality of the explanations produced

METODI DI VISUALIZZAZIONE



Visualization methods	Features
Backpropagation-based	visualise the relevance of features based on the magnitude of the gradients flowing through the network layers during training
Perturbation-based	visualise the relevance of the features by comparing the network output for a certain input and for a suitably modified copy of the input

VISUALIZATION METHODS



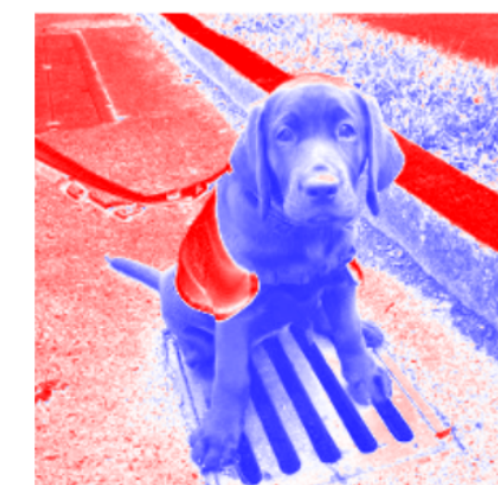
many available
methods ...

EXAMPLES OF BACKPROP-BASED METHODS

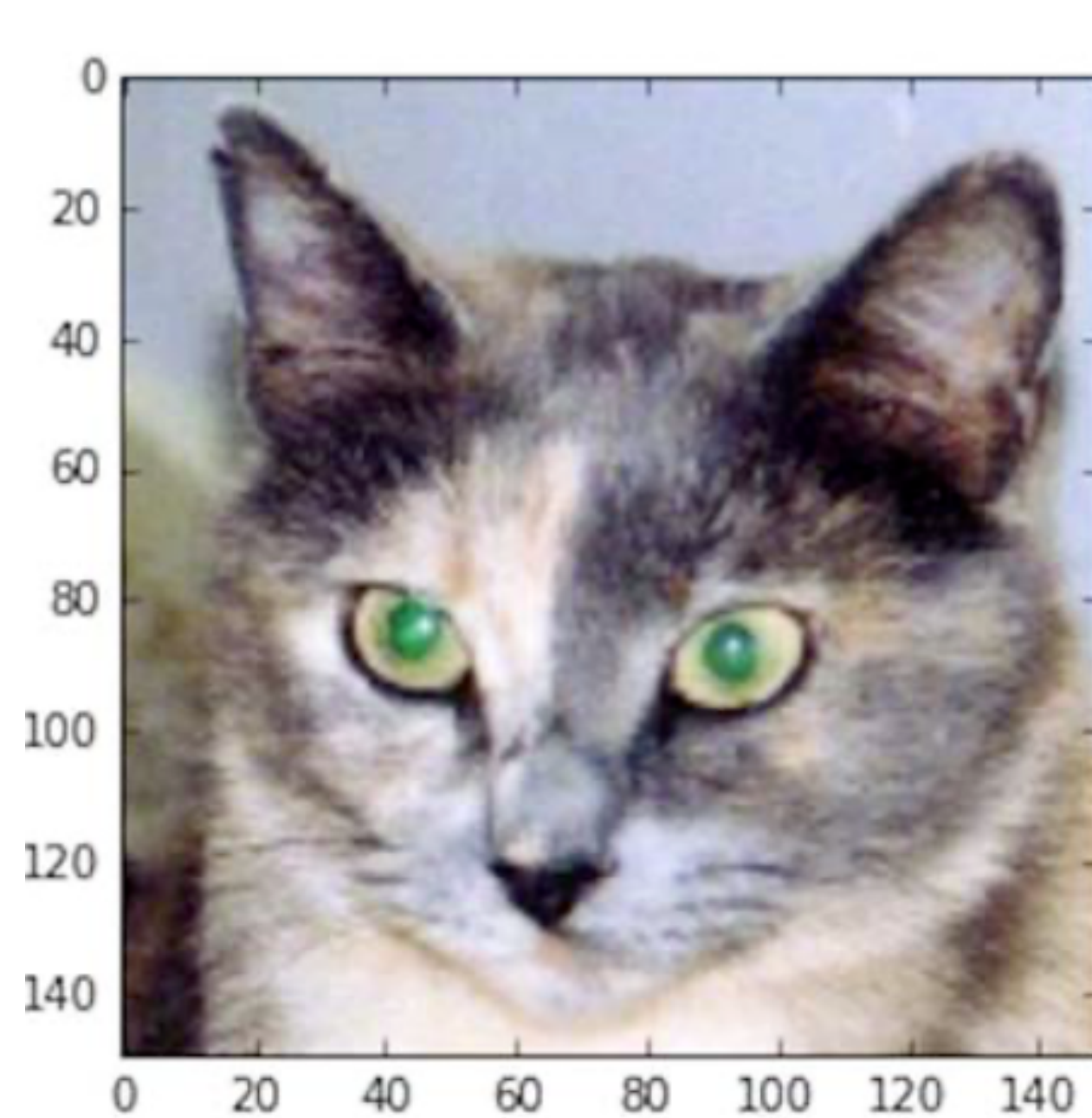
- **ACTIVATION MAXIMIZATION**

- used to display the important features in each layer by optimizing the input x so that the activation a of the neuron considered is maximized (with fixed network weights)
- optimal x obtained using **gradient ascent** of $a(x; w)$

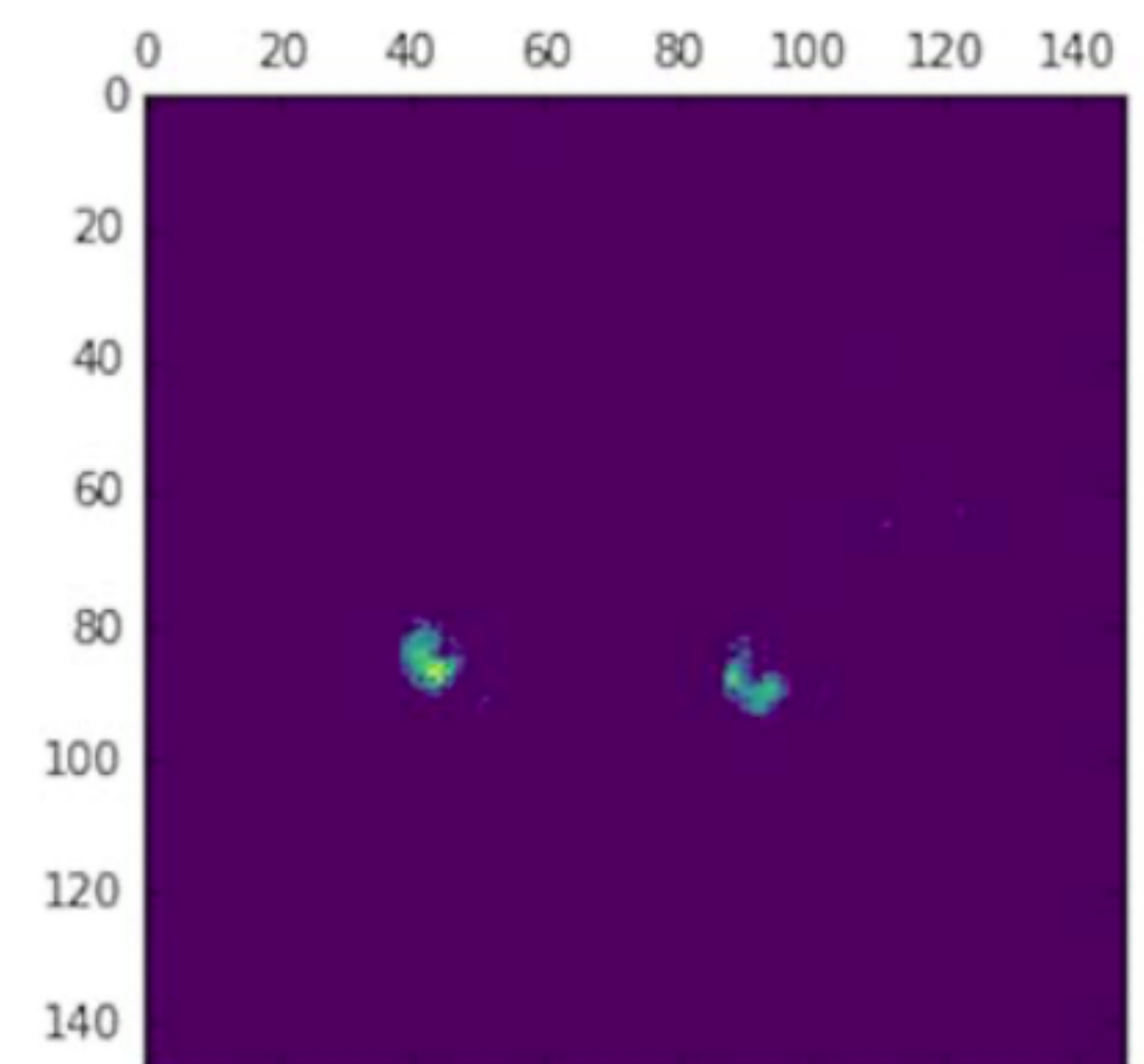
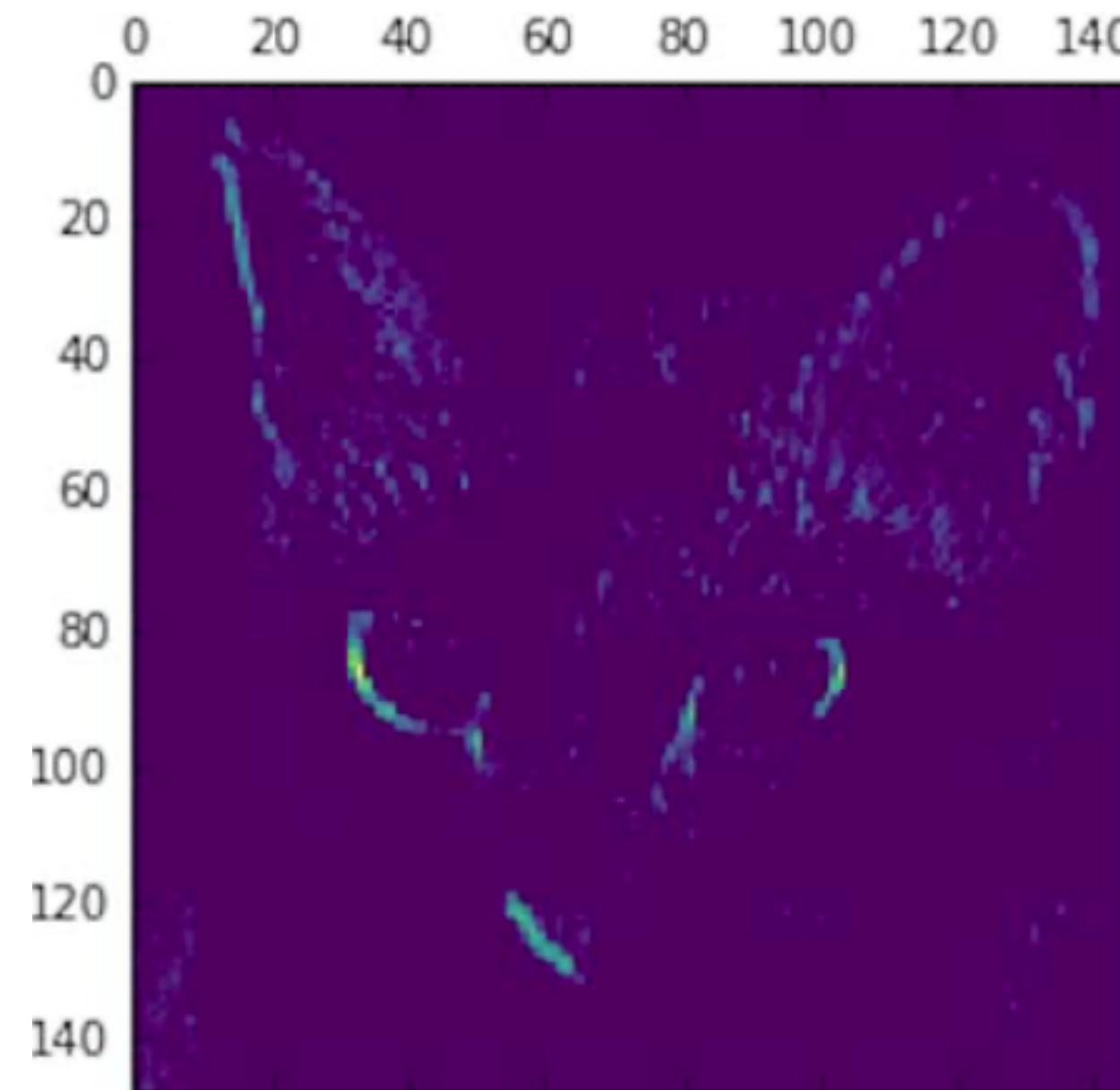
Alexnet



EXAMPLE: VISUALIZATION OF THE OUTPUT OF CNN FILTERS

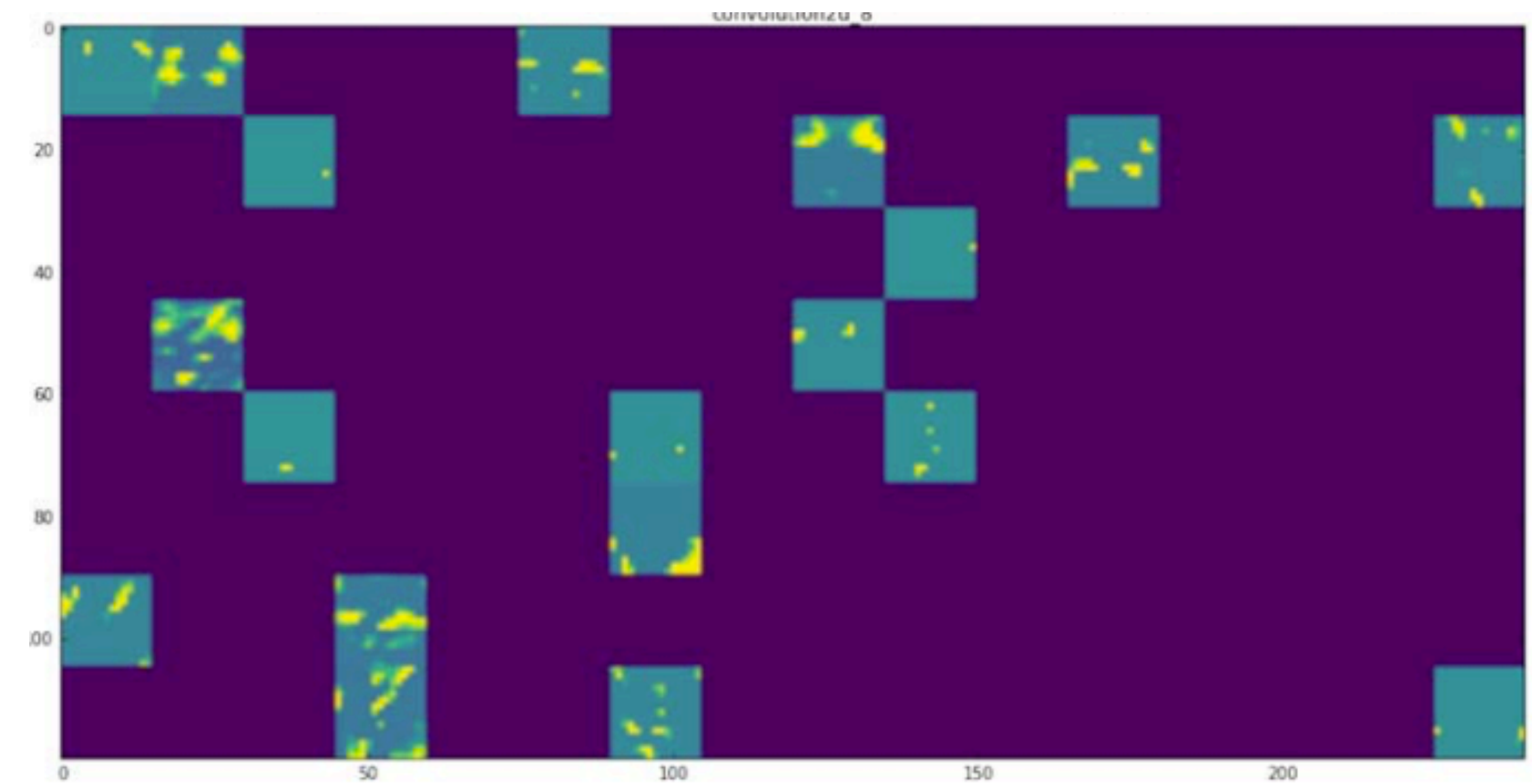
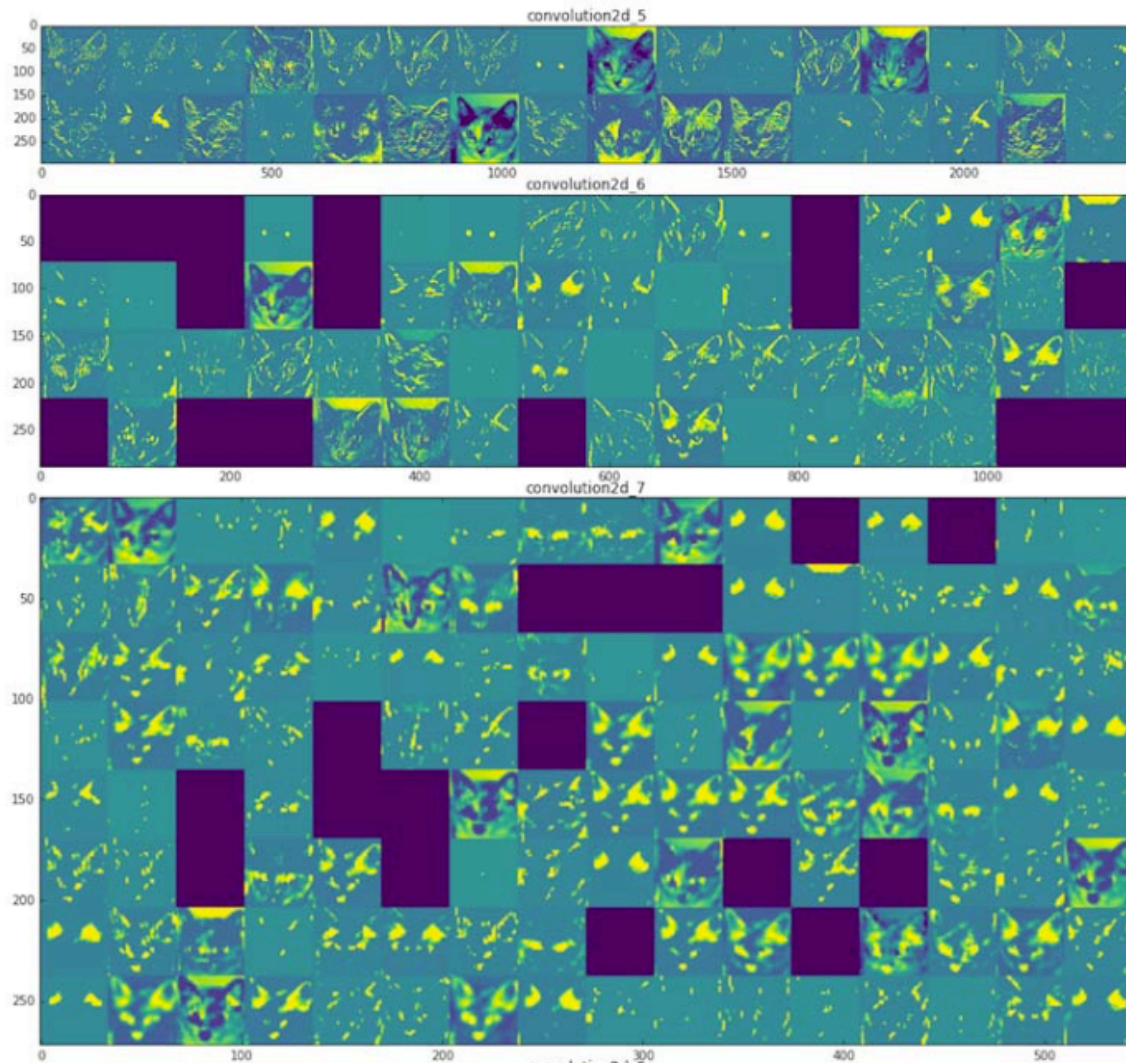


image



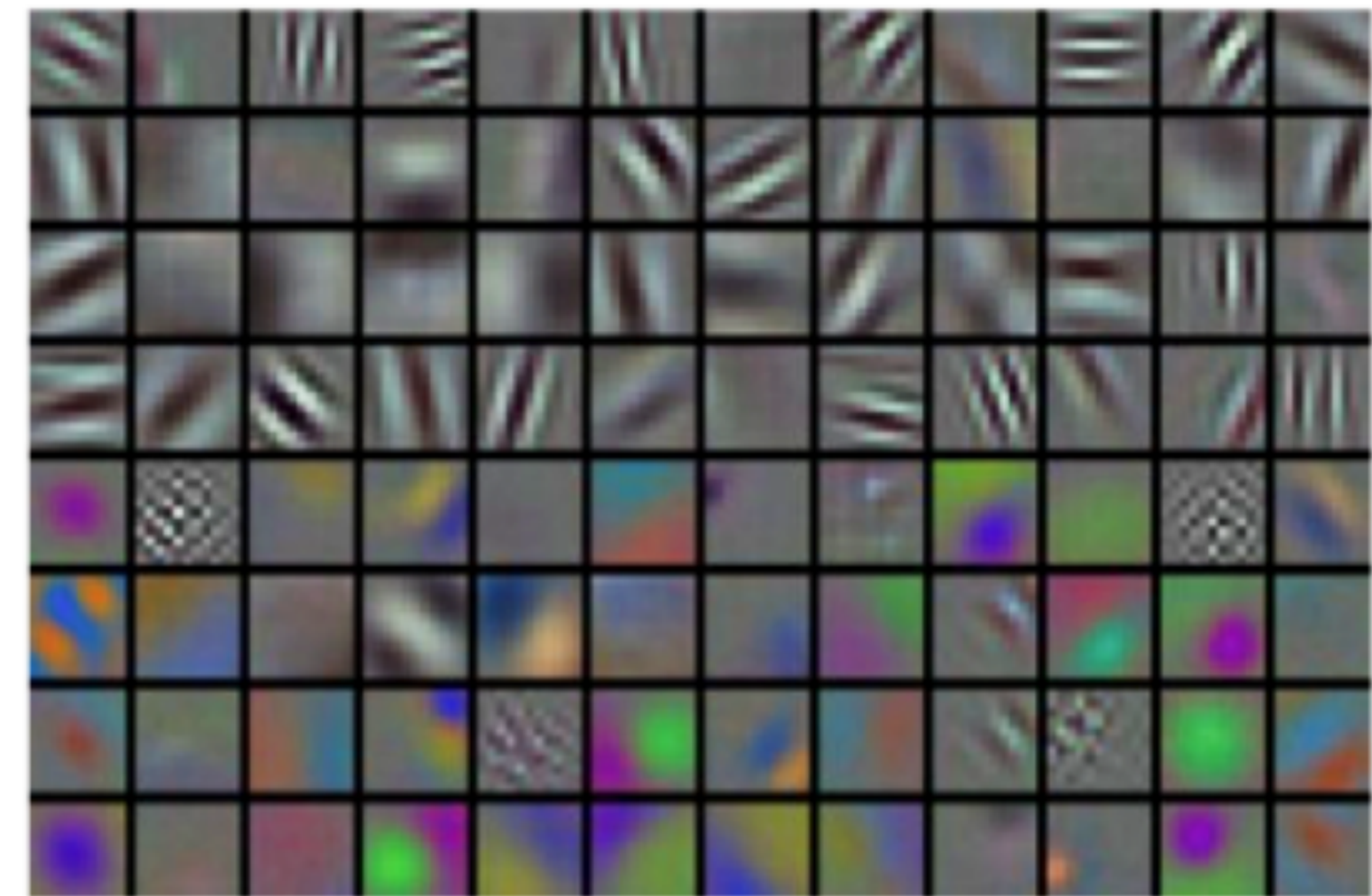
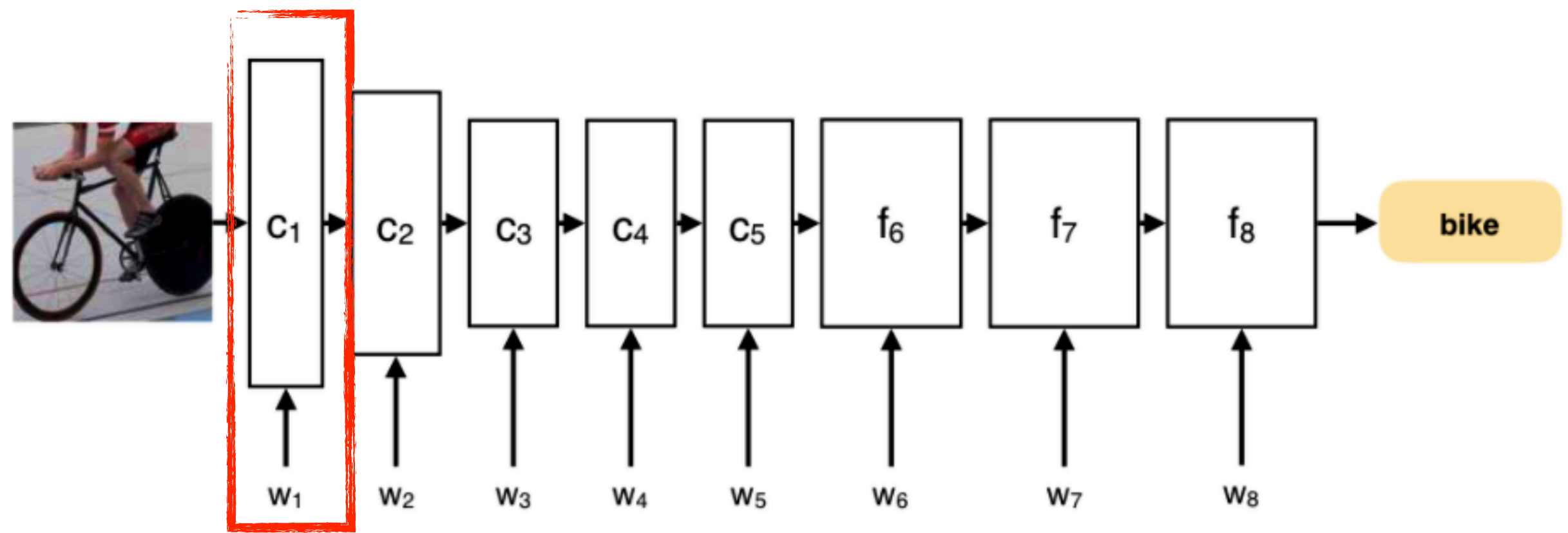
outputs of the activations of two
of the filters of the first layer

useful for understanding how the subsequent layers of convolution transform the input

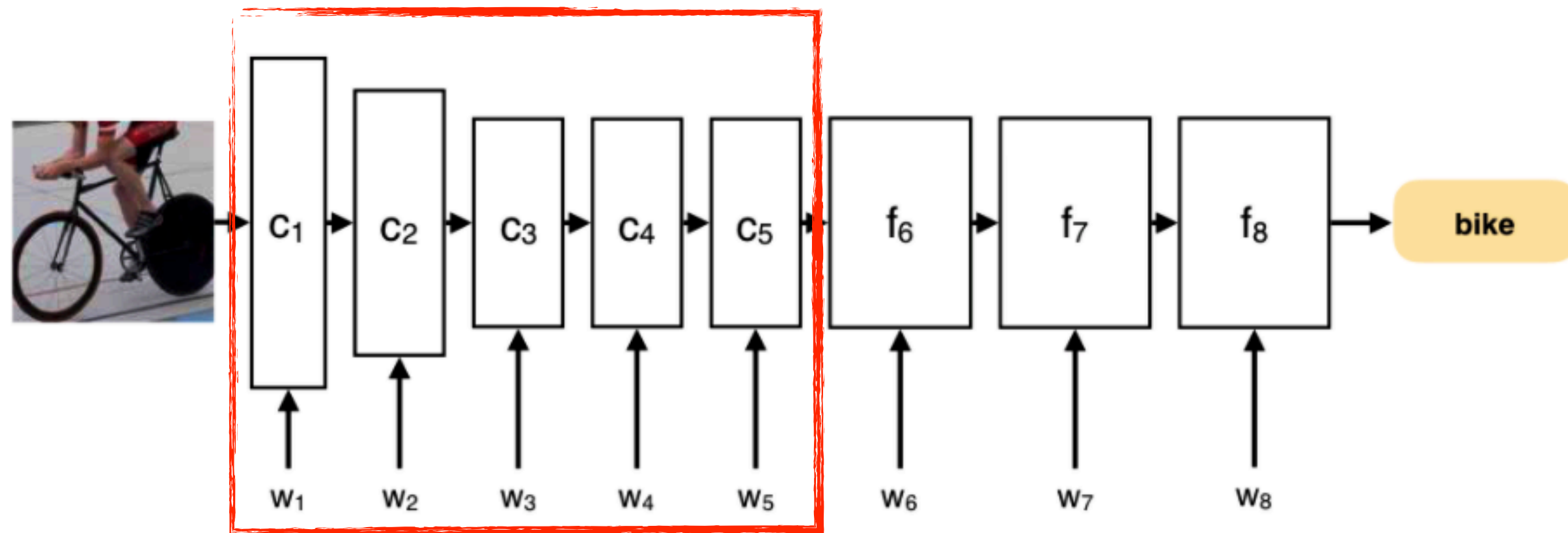


EXAMPLE: VISUALIZATION OF THE OUTPUT OF CNN FILTERS

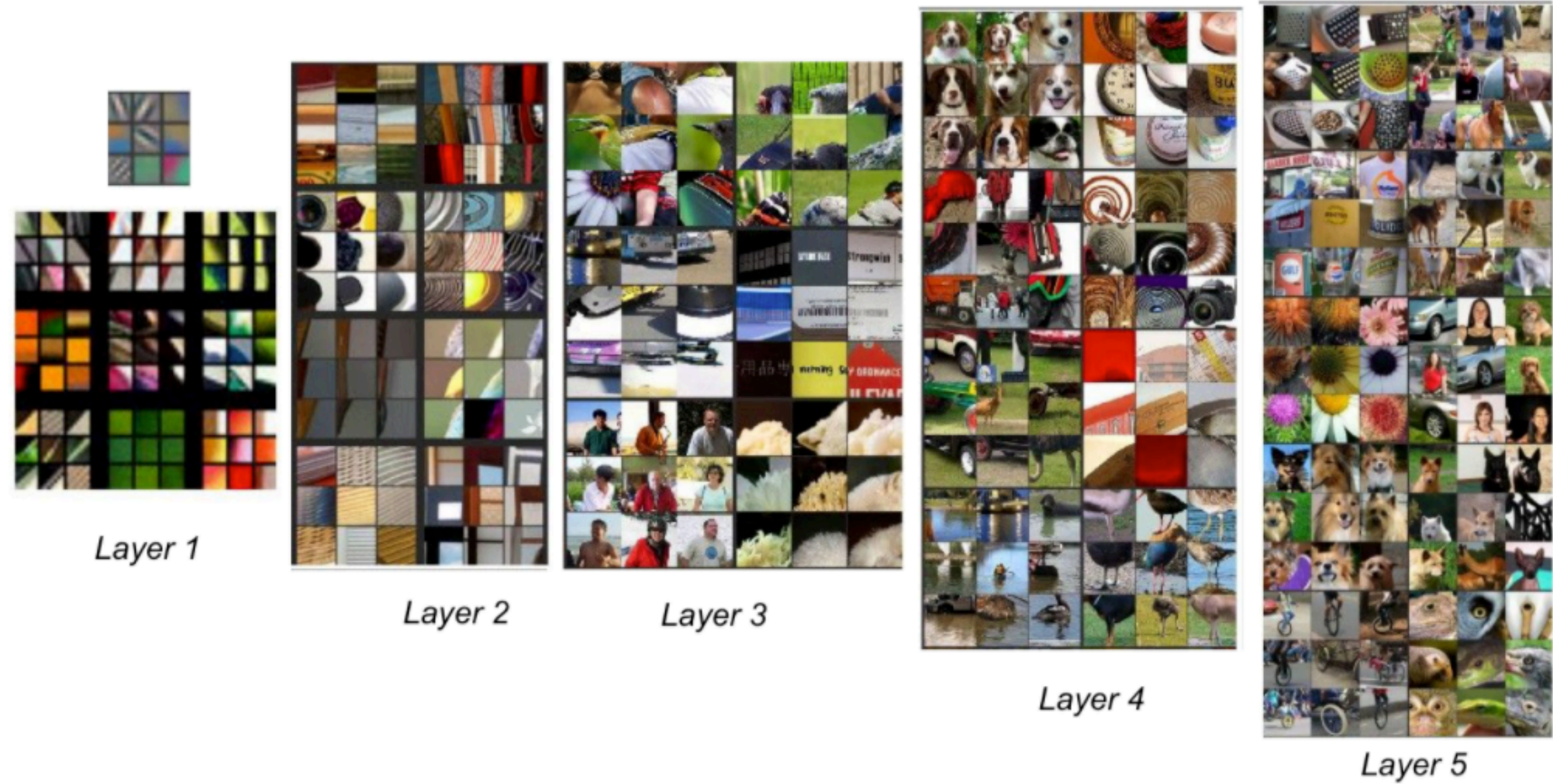
- another very simple way to quickly analyze the shape of the filters learned from the network
- the visual patterns to which each filter is expected to respond are displayed
- technique: ascent along the gradient in the input space → in practice starting from an initial empty image each pixel is varied in order to maximise the response to a specific filter. The image constructed this way will be the one for which the filter has the greatest response
 - a loss is defined that maximize the value of given filtering a given convolutional layer
 - SGD is used to adjust the pixel values in the input image maximizing the loss

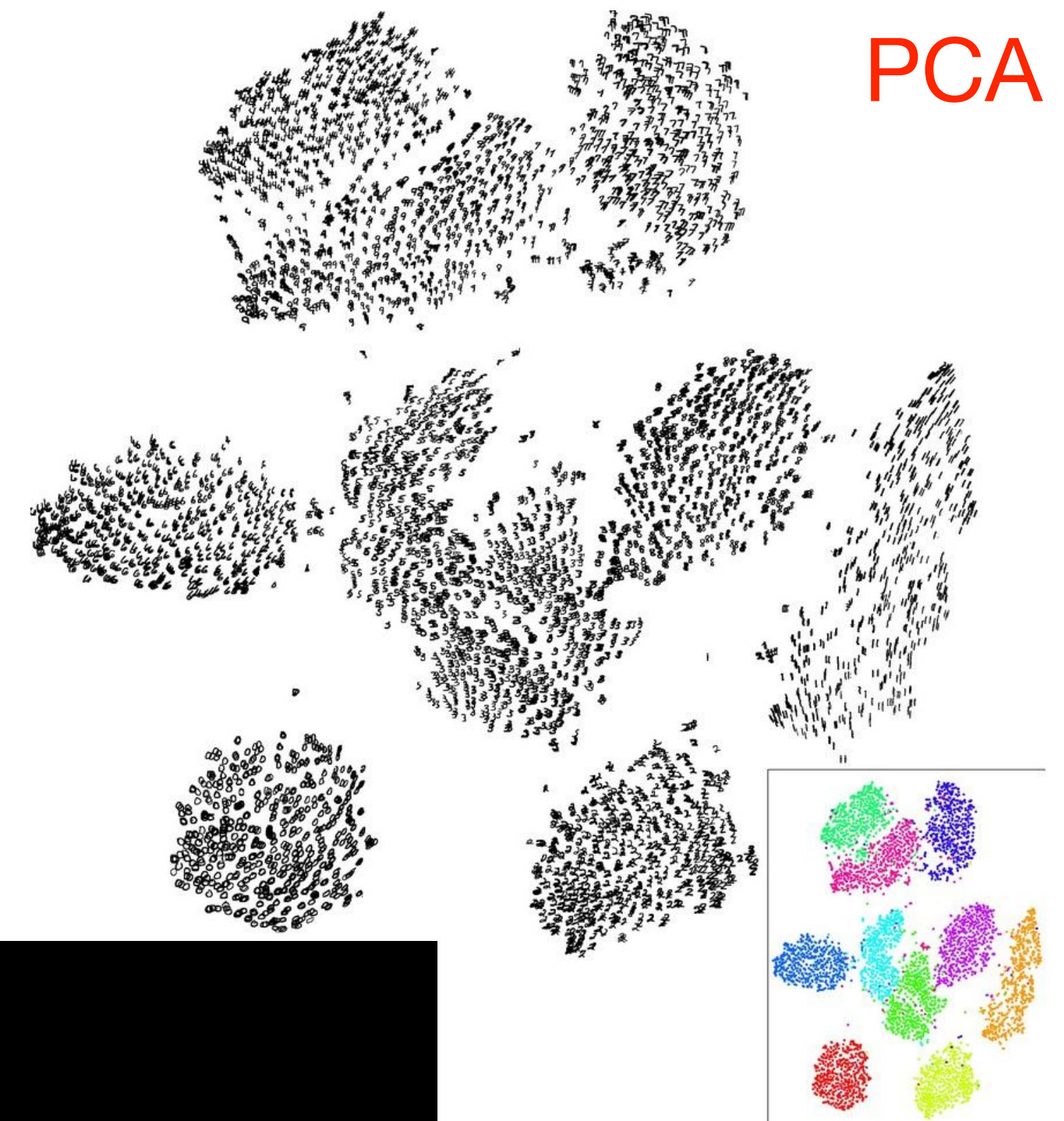
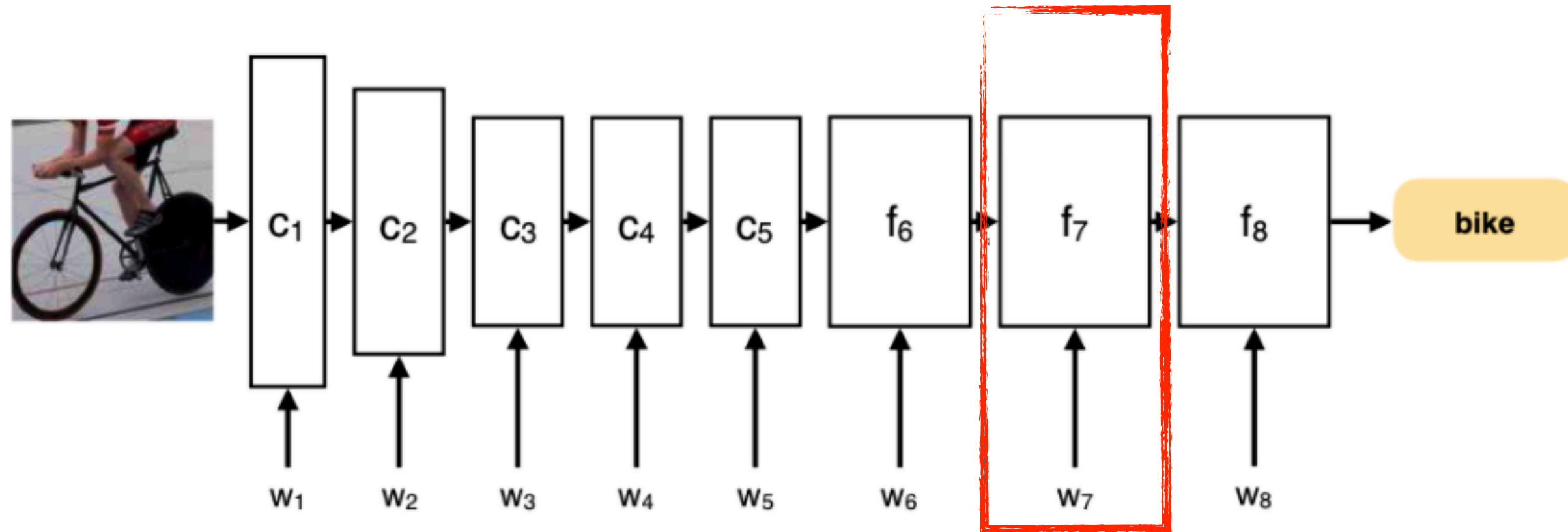


some of the filters of the first layer of VGG16



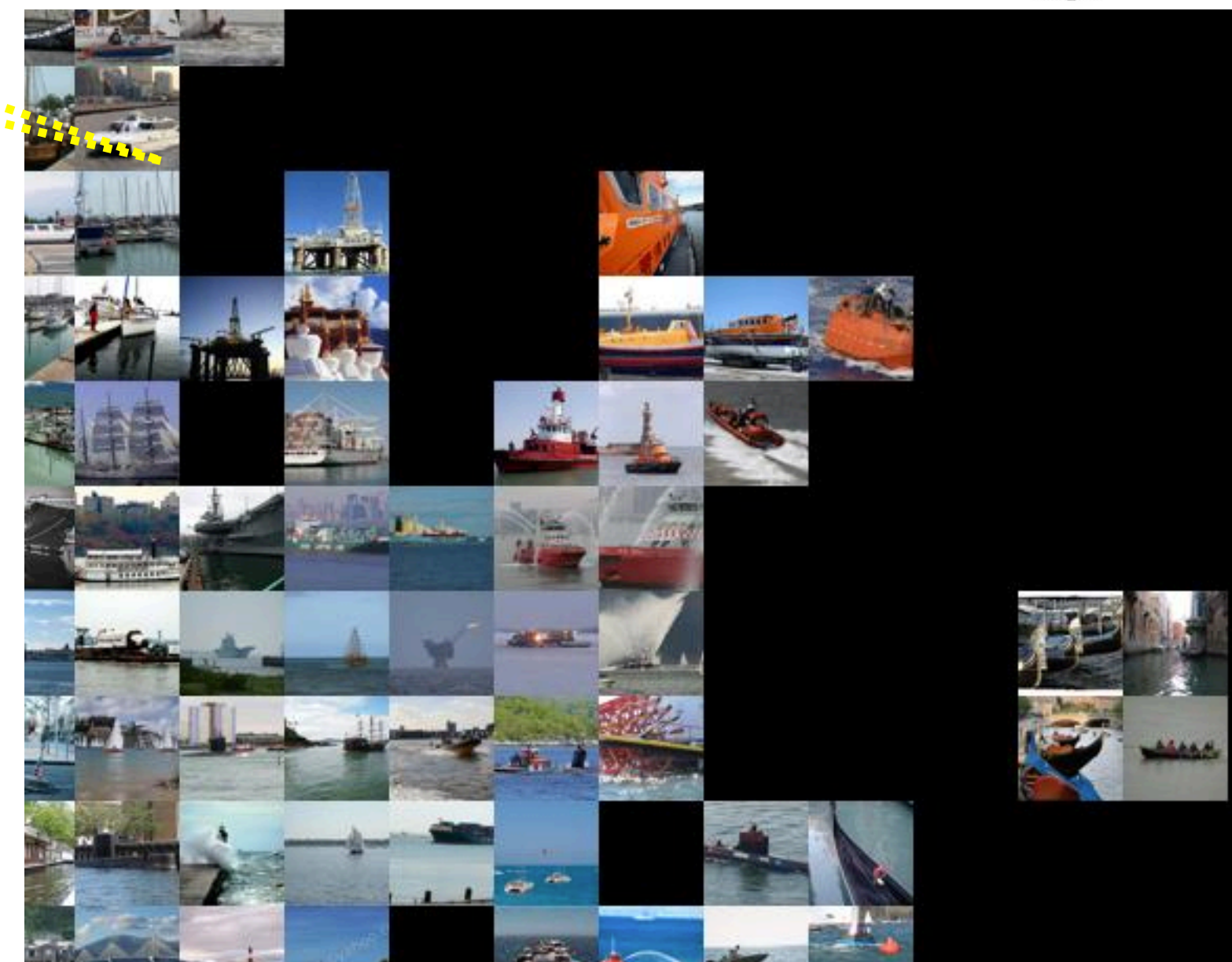
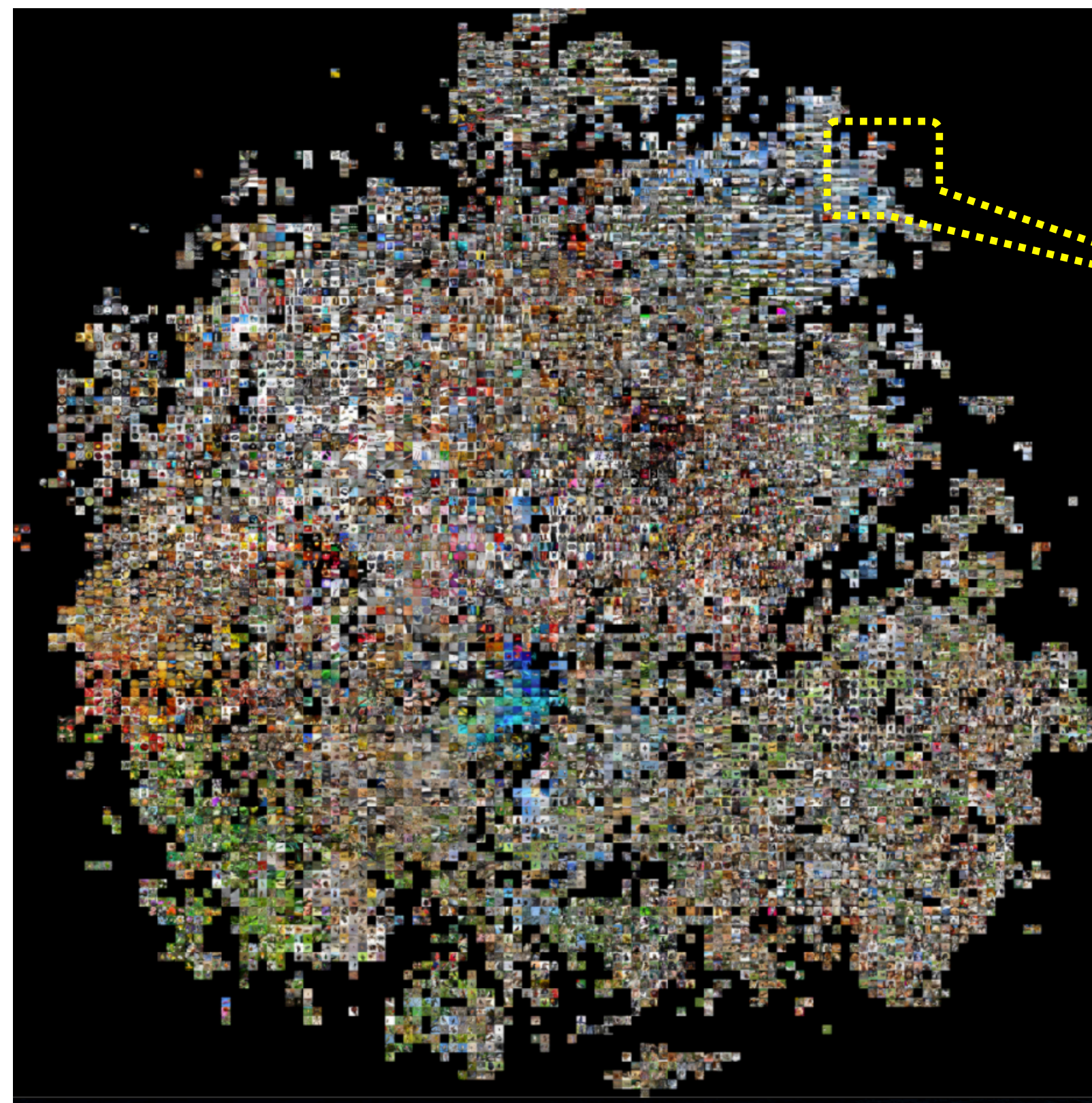
for subsequent convolutional layers the results tend to be less useful ...





for dense layers of extreme dimensionality (ex. alexnet 4096d)
it is advisable to apply methods of size reduction (PCA or tSNE 4096- \rightarrow 2)

tSNE

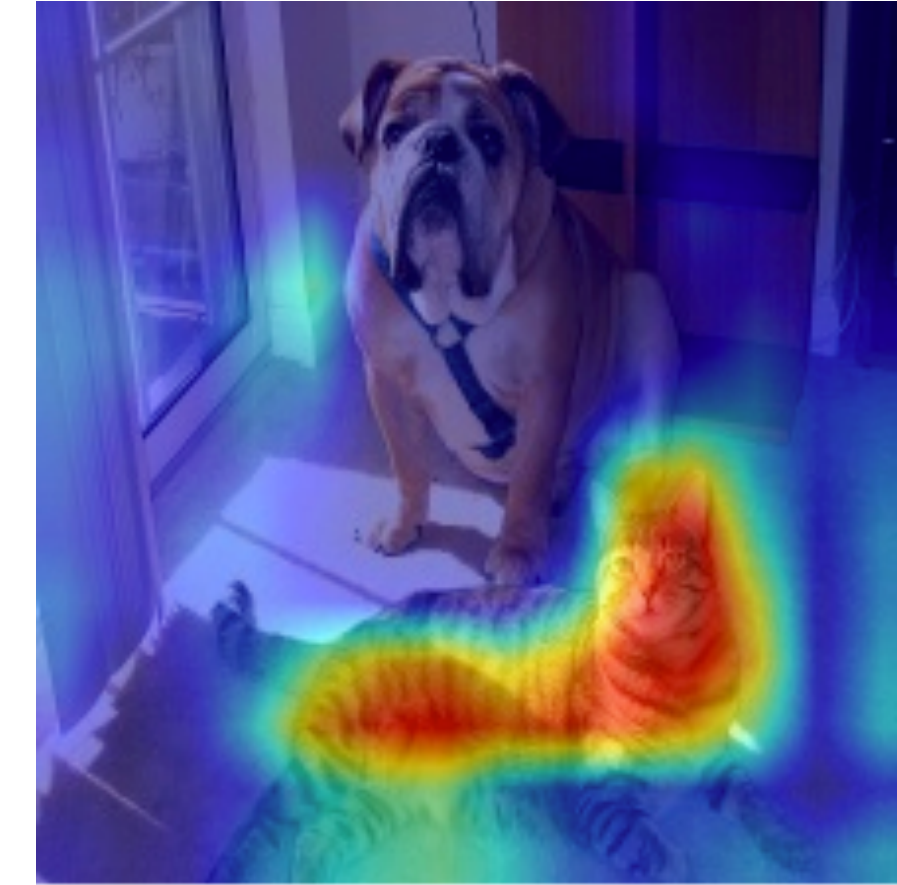


EXAMPLE: HEAT-MAPS (GRAD-CAM)

- useful for understanding which parts of an image have been identified as belonging to a certain class and for locating objects in images
- it takes the feature map output of a convolutional layer produced by a given input images
- each channel in the feature map is weighed through the gradient of the class with respect to the channel (i.e. it is measured how much the input image activates the class)



predicted class:
Indian elephant

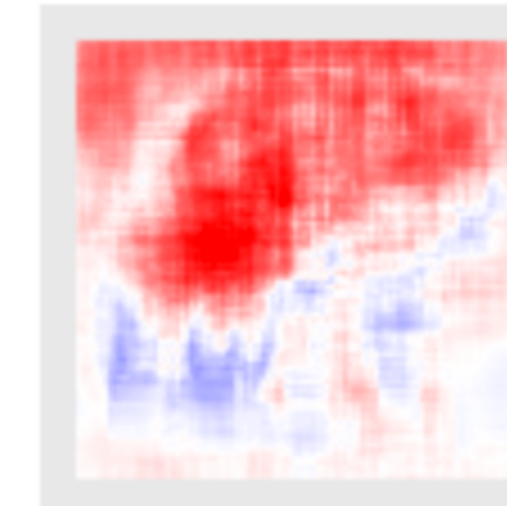
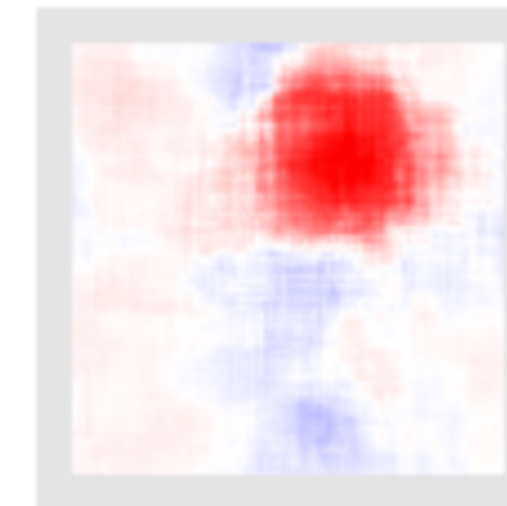


predicted
class: cat

EXAMPLE: PERTURBATION METHODS

- **OCCLUSION SENSITIVITY:**

- a gray patch that hides part of the pixels of the image is slides along the image, looking at how the prediction of the model varies
 - sensitivity maps (heat-maps): difference between the value of the output unit that responds maximally for a given image and the value that the same unit has when a part is occluded
- idea: performance varies significantly when influential elements of the input are masked



original
image

occlusion
mask
32x32

alexnet
stride 2