FUTURE AI RESEARCH

ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

ANN: PRINCIPLES AND COMMON ARCHITECTURES

S. Giagu - 4th ML_INFN Hackathon

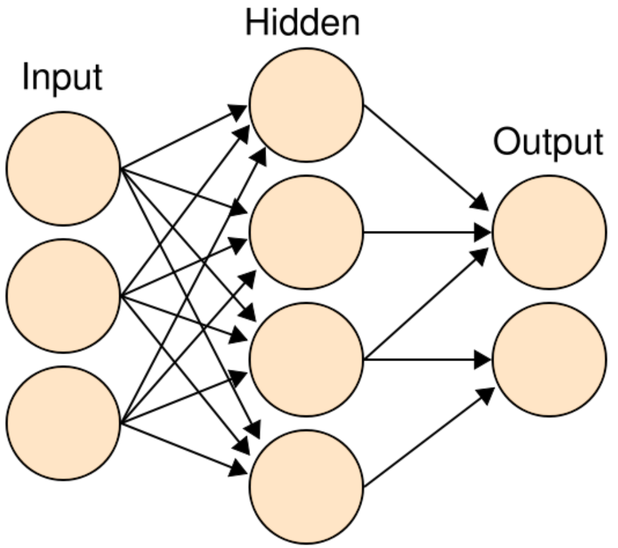
Online/INFN Cloud - 21.6.2023

INFN
Istituto Nazionale di Fisica Nucleare

ARTIFICIAL NEURAL NETWORKS

- the most popular approach to machine learning in the last decade
- an ANN is a mathematical model able to approximate with high precision generic multidimensional functions:

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

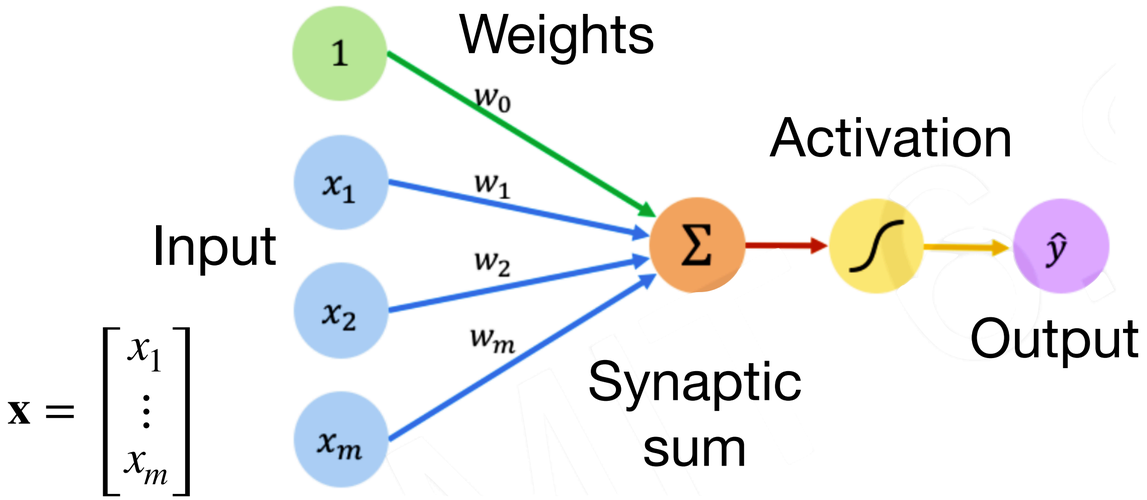


- very shallow analogy with biological neural networks
- more precisely defined as a composition of functions (layers) connected in chains described by a graph (ex: a feed-forward ANN described as a direct acyclic graph)

- connectionist computational approach: collective actions performed in parallel by simple computational units (neurons)
- learns as an adaptive system: the network structure dynamically change during a training phase based on a set of examples that flow through the network during the training steps
- non linear response obtained by non linear neuron outputs
- hierarchic representation learning obtained by implementing complex multilayered topologies (DNN)

ARTIFICIAL NEURON MODEL: TRESHOLD LINEAR UNIT

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



$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

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

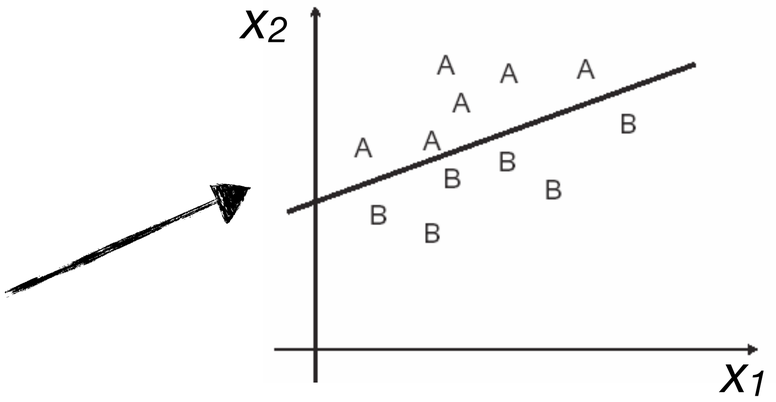
$$z = w_0 + \sum_{i=1}^{n} w_i x_i = w_0 + \mathbf{w}^t \mathbf{x} \qquad \mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

- and an activation function (for example a step function):

$$a(z) = \begin{cases} 1 \text{ if } \mathbf{w}^t \mathbf{z} \geq -w_0 \\ 0 \text{ if } \mathbf{w}^t \mathbf{z} < -w_0 \end{cases}$$

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

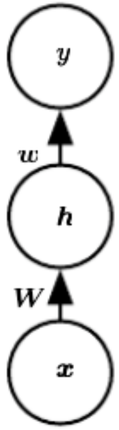
a single layer of TLU with step activation can only learn to solve problems with linearly separable classes



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

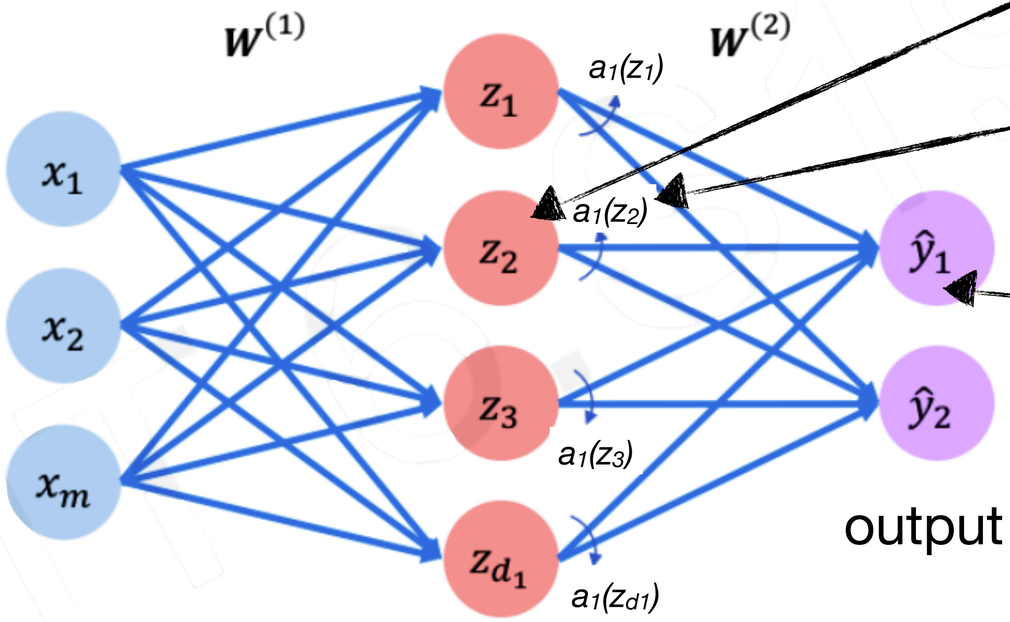
MULTILAYER PERCEPTRON (FEED-FORWARD NN)

- the most classical and simplest DNN architecture (FFNN or MLP)
  - neurons organised in consecutive layers: input, hidden-1, ... , hidden-K, output
  - only connections of neurons of a given layer towards the next are possible: acyclic direct graph
  - all possible connections are present (dense layers)



≡

FFNN
acyclic direct
graph



MLP with 1 hidden layer

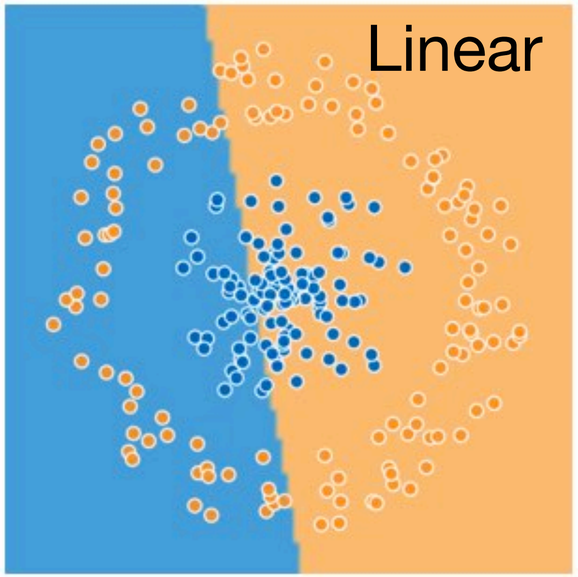
$$z_i = w_{0i}^{(1)} + \sum_{j=1}^{m} x_j w_{ji}^{(1)}$$ synaptic sum

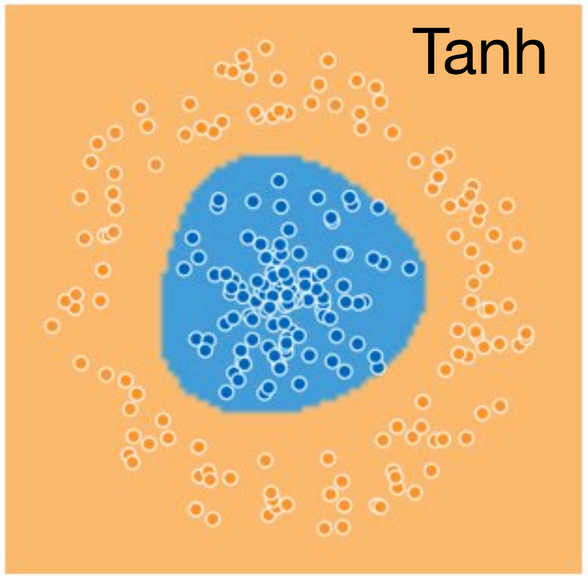
$$a_1(z_i)$$ activation

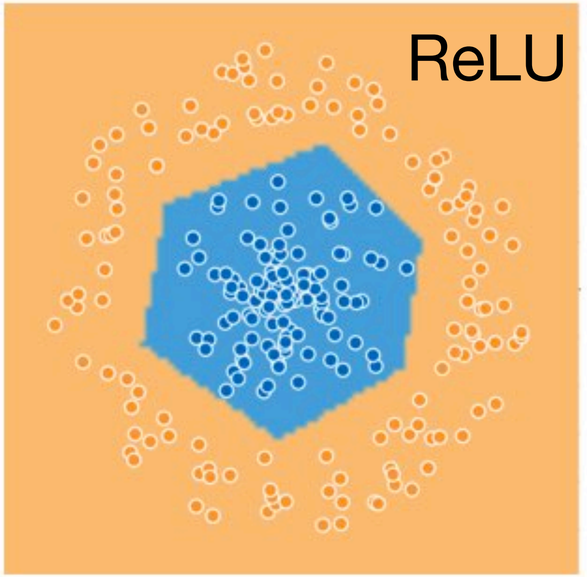
$$\hat{y}_i = a_2\left(w_{0i}^{(2)} + \sum_{j=1}^{d_1} a_1(z_j) w_{ji}^{(2)}\right)$$

- NN behaviour determined by:
  - network topology (#layers, size of each layer, …)
  - weights w_{ij}
  - activation function of each layer

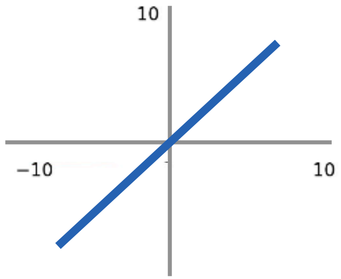
non-linear activations allows to learn complex and non linear patterns …



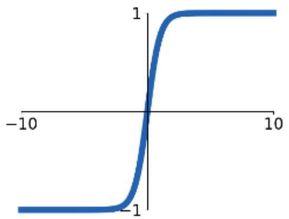




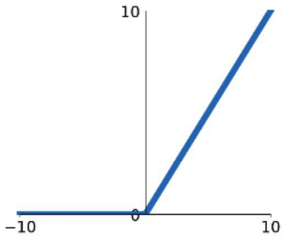
$$a(z) = z$$



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



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

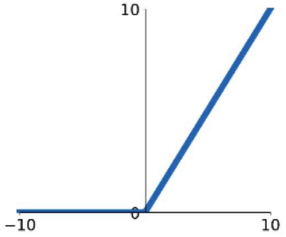


ACTIVATION FUNCTIONS FOR THE HIDDEN LAYERS

In general, any continuous and differentiable function would works. In practice some functions are better than others …

ReLU

$\max(0, x)$

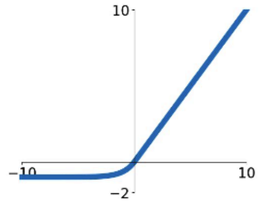


the most popular:

- 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 and task. In case of problems can be replaced with alternatives:

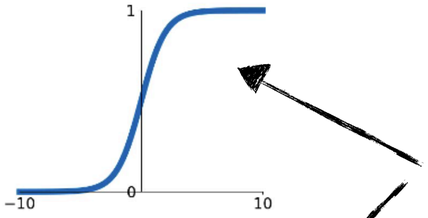
ELU

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



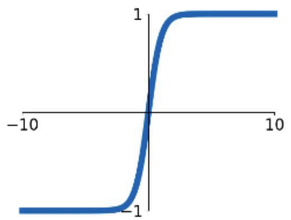
Sigmoid

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



tanh

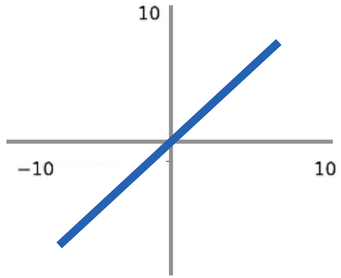
$\tanh(x)$



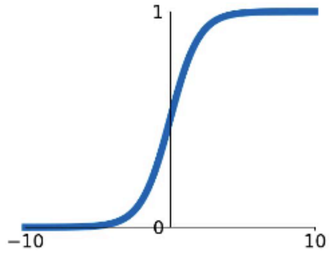
should not be used in general for dense and convolutional layers:

- gradient vanishes away from x=0 → vanishing gradient problem
- sigmoid has output not centered in zero → 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: typically used in binary classification problems (2 classes) with a single output neuron, or multilabel (multiple mutually inclusive classes) or sometime when the output features are numbers in (0,1)

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

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

- soft version of the argmax output ($y = \arg\max[\mathbf{z}]$)
- often used in multi-class classification tasks (with mutually exclusive classes)
- makes output a convex sum: $y_i \in (0,1)$ and $\sum_i y_i = 1$ interpretable as a probability

FFNN AS UNIVERSAL APPROXIMATORS

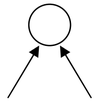
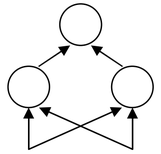
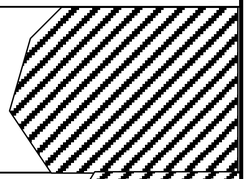
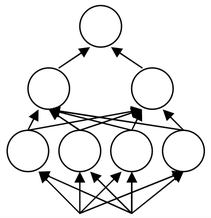
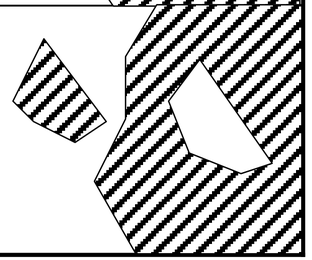
a FFNN with a single hidden layer containing a finite number of neurons with non linear activations can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation function

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

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

Universal approximation theorem proof:

- unbounded, sigmoid: here
- bounded, ReLU, arbitrary depth: here

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

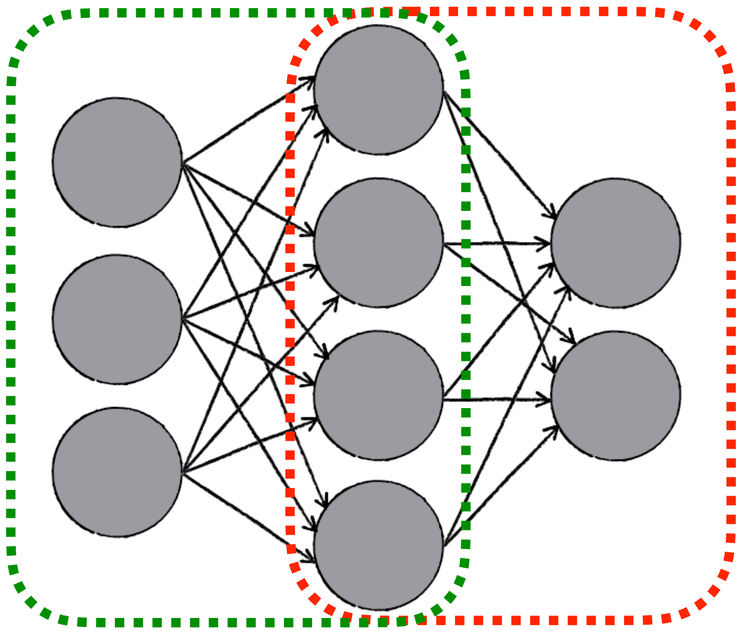
IMPORTANT: the theorem says nothing about the effective possibility to learn in a simple way the parameters of the model, all the DNN practice boils down in finding optimal and efficient techniques to solve this problem …

ANN AS A 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

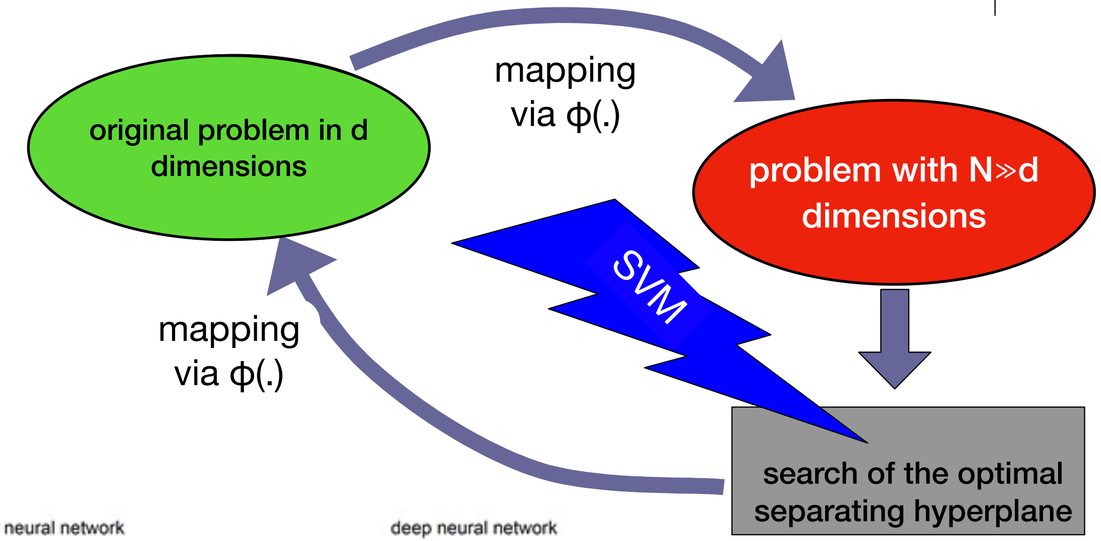
LEARN A (LINEAR) MAPPING BETWEEN LETENT REPRESENTATION AND TARGET



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

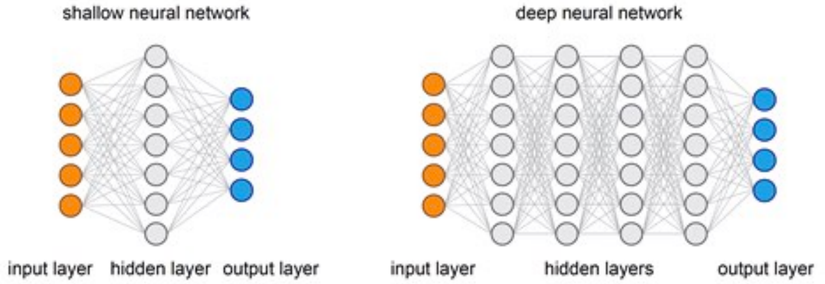
$\Phi: R^d \rightarrow R^\infty$

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



evolution of this approach: Deep-NN

a DNN is a ANN with >1 hidden layer …



DNN: WHY GOING DEEP WORKS?

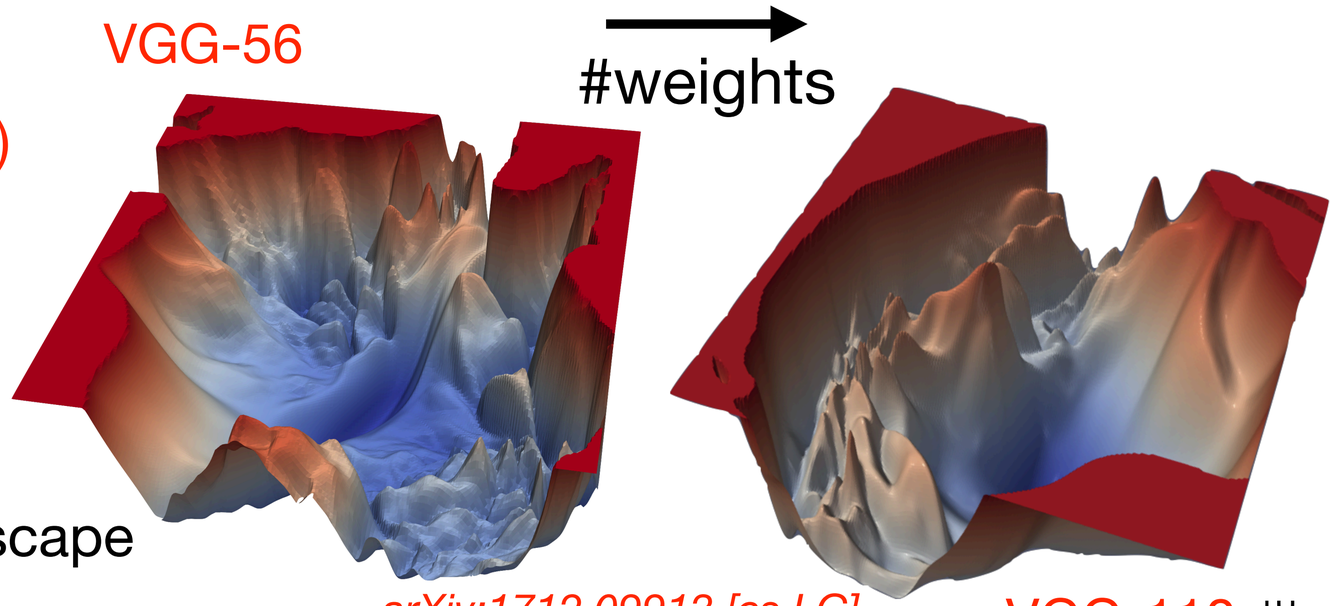
- the universal approximation theorem tells us that already a FFNN with one hidden layer can approximate any function with arbitrary precision

- 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))

  - sub-features (intermediate representations) can be used in parallel for multiple tasks performed with the same model

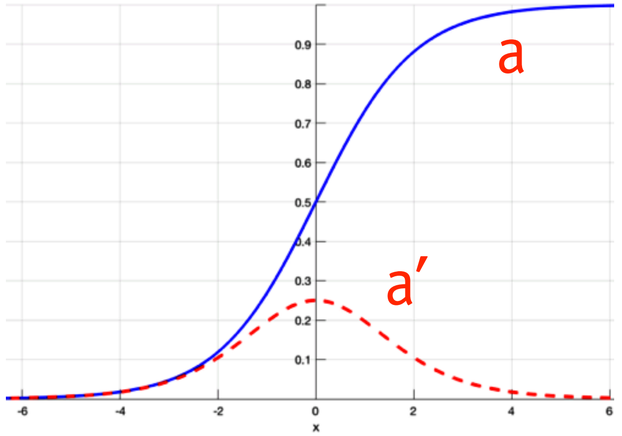
  - overparametrization and skip connections in very deep NN seems to have beneficial effectsin smoothing the loss function landscape



arXiv:1712.09913 [cs.LG]

WHY GOING DEEP IS DIFFICULT: VANISHING GRADIENT

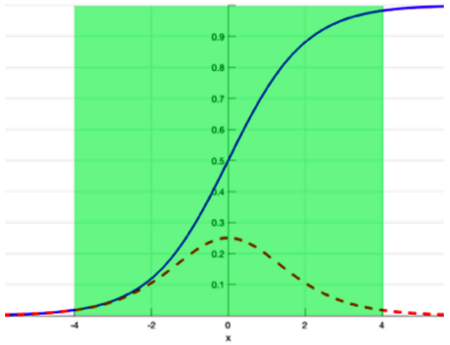
- the main problem in the use of DNN architectures is related to the vanishing gradient
- the first layers of a deep NN fail to learn efficiently
  - reason: during backprop in a network of n hidden layers, n derivatives of the activation functions will be multiplied together. If the derivatives are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



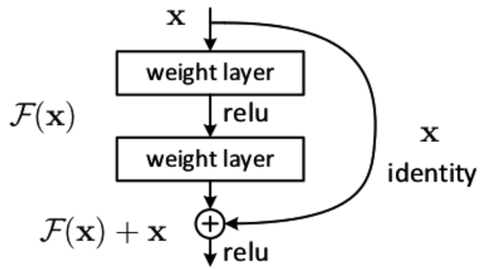
- SOLUTIONS:

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))

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



3. use residual networks: in which skip connections that do not pass through the activation functions and propagate information to subsequent layers (additional advantage : makes learning the layer easier)



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

- training consists in adjusting the parameters according to a given cost function that is a differentiable proxy to the performance of the model wrt the specific task we want to solve
  - weights and biases: “adjusted” using stochastic gradient descent with back-propagation
  - hyperparameters (parameters whose values are fixed before the learning process begins): “adjusted” using heuristic approaches (manual trial&error, grid or random search, bayesian-opt, autoML, …)

Example: supervised training

- during the training N examples are presented to the network: T{$x^{(i)}$, $y^{(i)}$} (i=1,…,N)
- weights are initialised to random values (small and around zero): for example ~N(0,σ) or U[-ε,ε]
- 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 $y^{(i)}$:

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

$$L_i = \frac{1}{2} \left(y^{(i)} - \hat{y}^{(i)}(x^{(i)} \mid \mathbf{w}) \right)^2$$

example: MSE

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:

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

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)

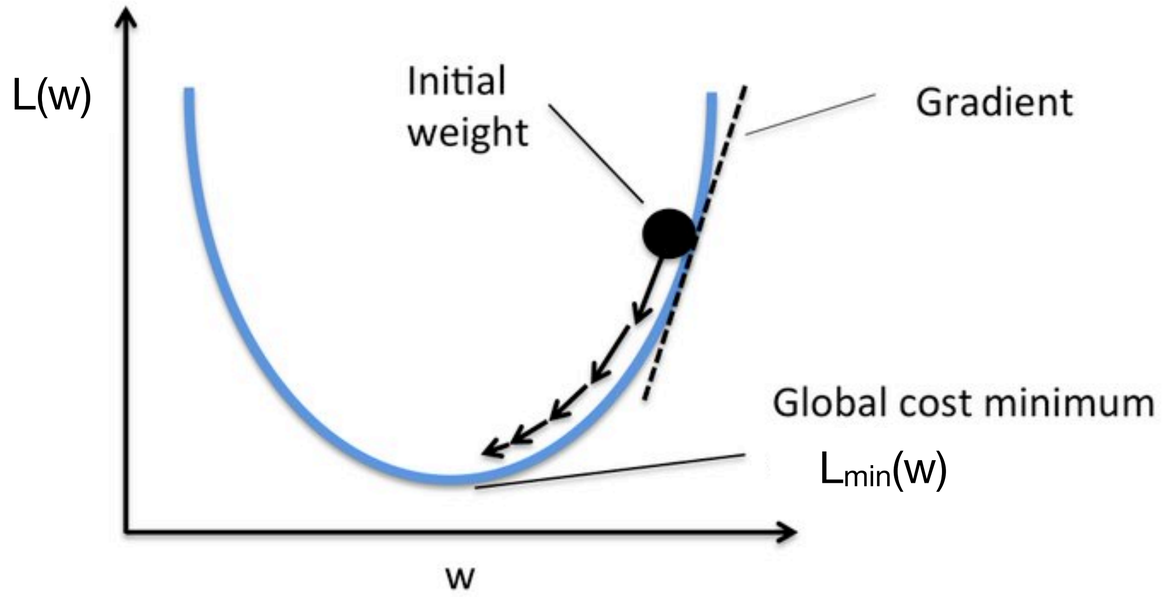
given two distributions p and q, $H_p(q)$ measures the average number of bits needed to identify an event extracted from the set, when the p model is used for the probability distribution, rather than the "true" distribution q. It is usually the best loss function to train ANNs that output probabilities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

NOTE: generalisation for multi class problems

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

LOSS FUNCTION MINIMIZATION

- the vector of weights is chosen as the one that minimizes L: $$\mathbf{w}^* = \underset{\mathbf{w}}{argmin}[L(\mathbf{w})]$$
- the minimum is sought with GD techniques … $$\mathbf{w}_{(t+1)} = \mathbf{w}_{(t)} - \eta \nabla_w L(T \mid \mathbf{w})$$



STOCHASTIC GRADIENT DESCENT (WITH MOMENTUM)

- 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 (mini batches) of the entire dataset T:

  - T is divided in m sub-samples (mini batches) $T_1 \ldots T_m$
  - weights are updated using each subset T_i:

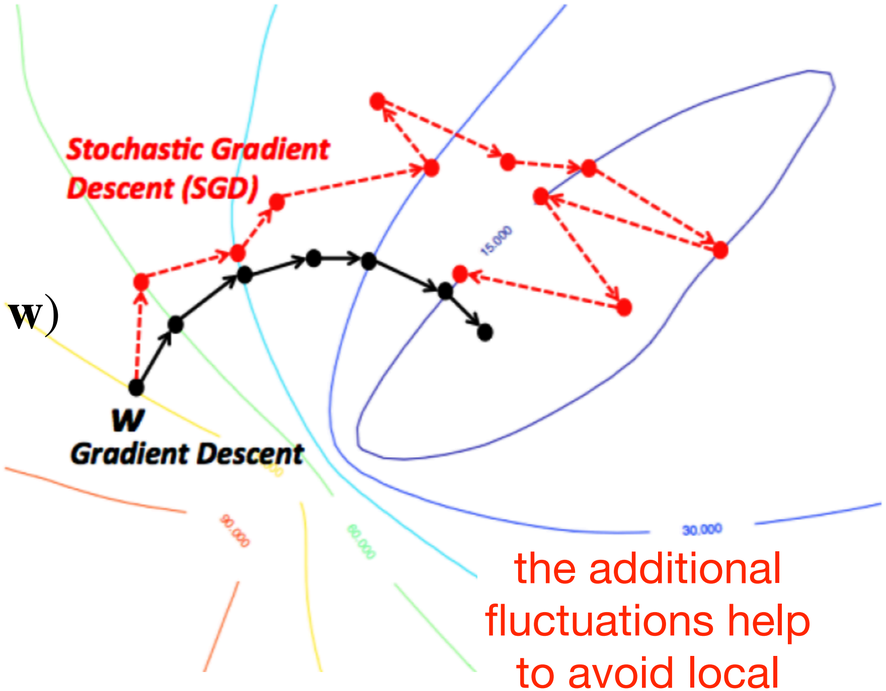
$$\nabla L_i(\mathbf{w}^{(k)}) = \frac{1}{N_i} \nabla \sum_{k \in T_i} L(\mathbf{x}_k, \mathbf{y}_k, \mathbf{w})$$

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

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

previous step direction

gradient direction



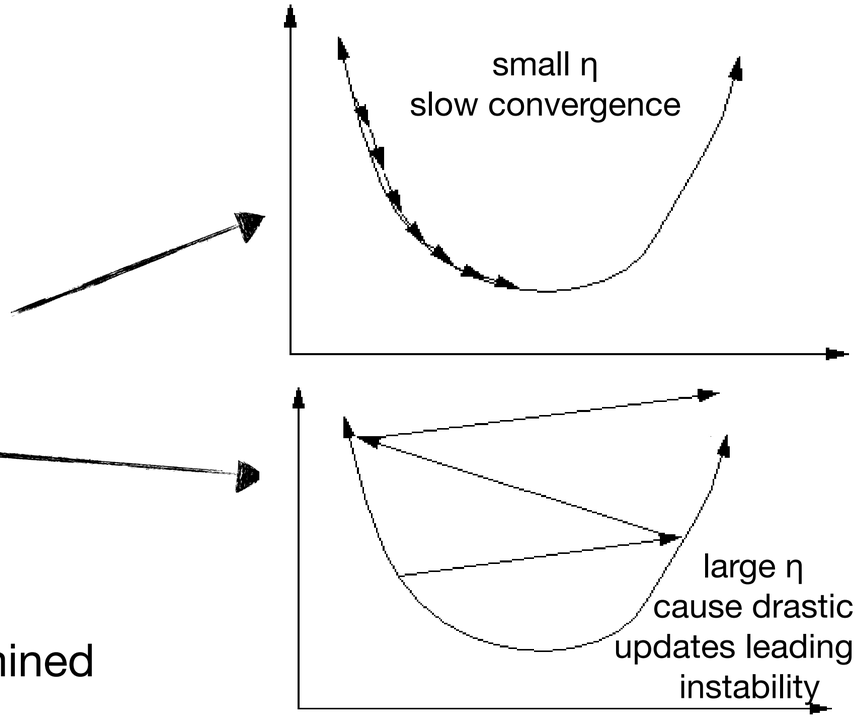
the additional fluctuations help to avoid local spurious minima

MOMENTUM:

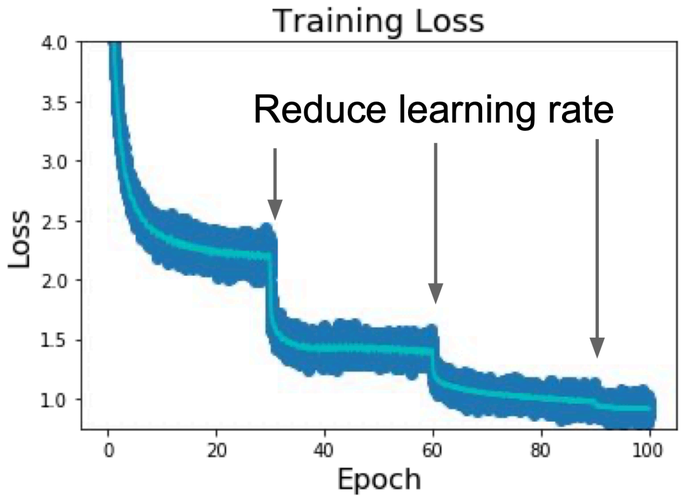
- for α = 0 we have classic GD/SGD
- for α = 1 the gradient descent is ignored and the weight update follows the previous direction (momentum)
- typically: α ~0.9-0.99

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:

- directions associated to relevant features: smaller effective LR

- directions associated to non relevant/low frequency features: larger effective LR

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

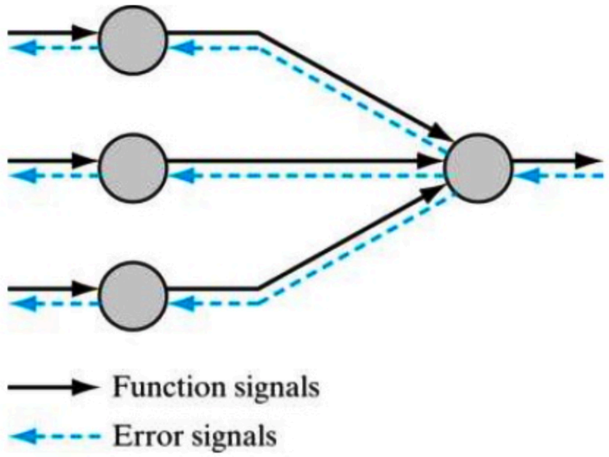
$$G_t = \sum_{\tau=1}^{t} \left[\frac{\partial L}{\partial w} \right]^2$$

several implementations:
Adadelta, RMSProp, Adam, …

BACKPROPAGATION

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 procedure

- 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)

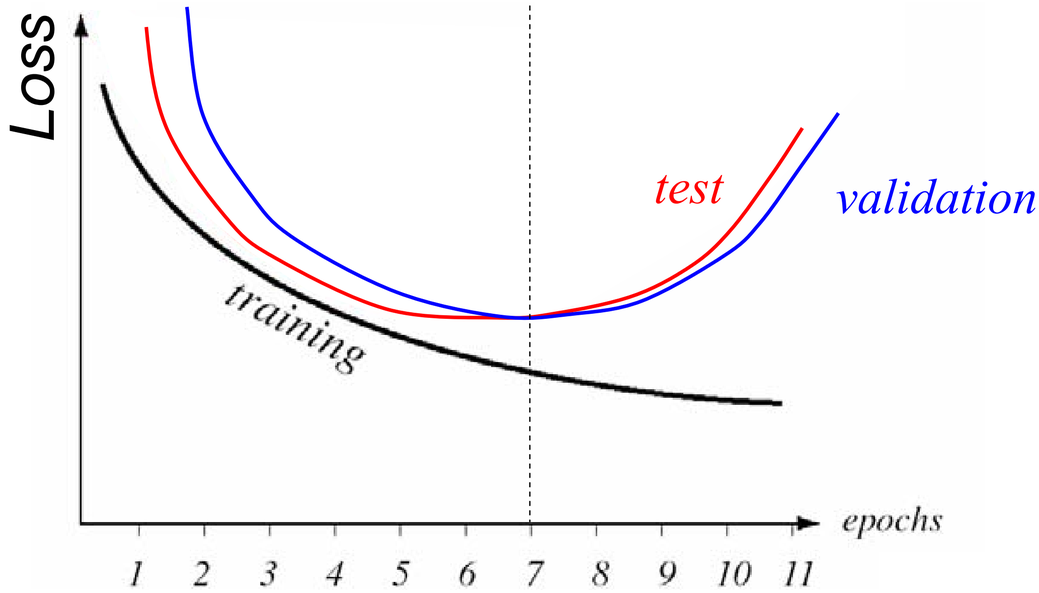


- each neuron (hidden or output) receives and compares the function and error signals
- back-propagation consists of a simplification of the gradient calculation obtained by applying recursively the rule of derivation of compound functions in the backward phase and implementing it as a computational graph to which apply automatic differentiation
- with backprop calculating the gradient of the Loss is as fast as calculating the Loss itself …

MONITORING THE TRAINING: 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 learning curves (loss or accuracy or any useful metrics vs epochs)

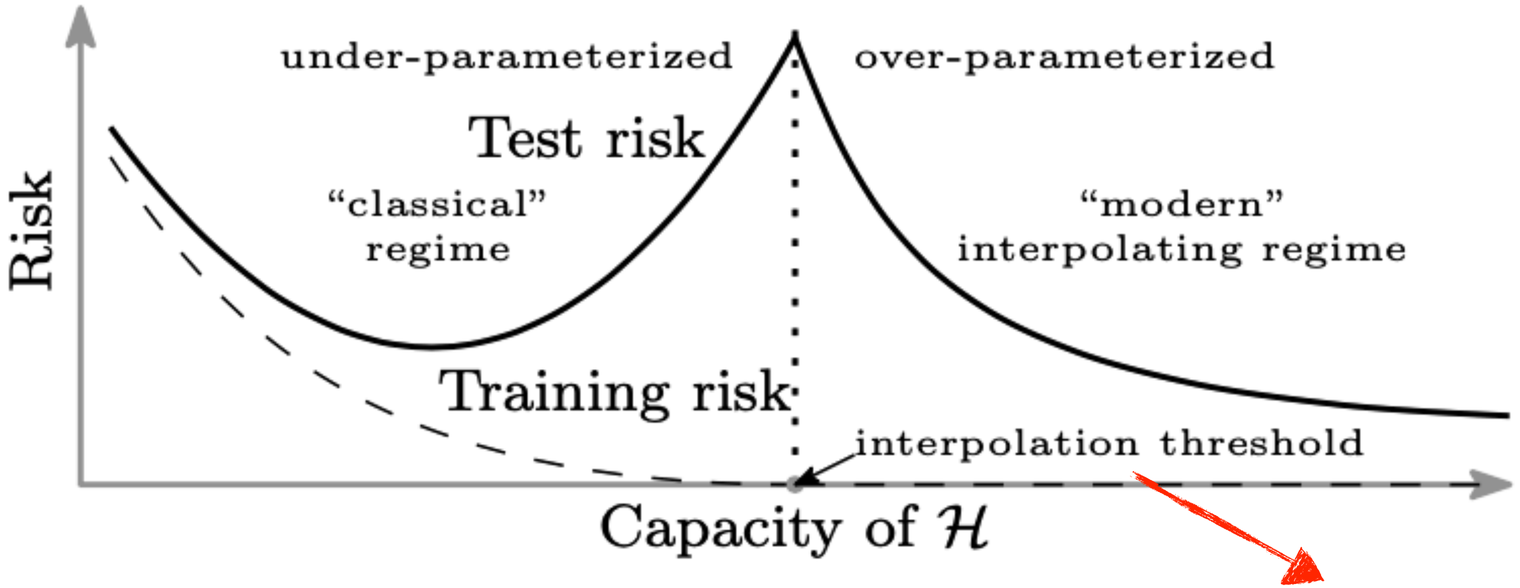
- multiple datasets (or cross validation) are needed to monitor the tradeoff between bias and variance during the training (e.g. undercutting vs overfitting) and to optimise the hyper parameter of the model
  - training set: to update the weights
  - validation set: to choose hyper-parameters and the training stop criteria
  - test set: to evaluate the final performances of the trained model



DOUBLE DESCENT IN DNN

- the bias-variance tradeoff tell us that larger models are worse: a model with zero training error is overfit to the training data and is expected to generalize poorly

- however large overparametrized DNNs, trained with datasets smaller than the number of parameters, exhibit the so called doubled descent phenomena (M.Belkin et al. https://arxiv.org/abs/1812.11118):



interpolation threshold seems to occur when the number of parameters is roughly equal to the number of samples

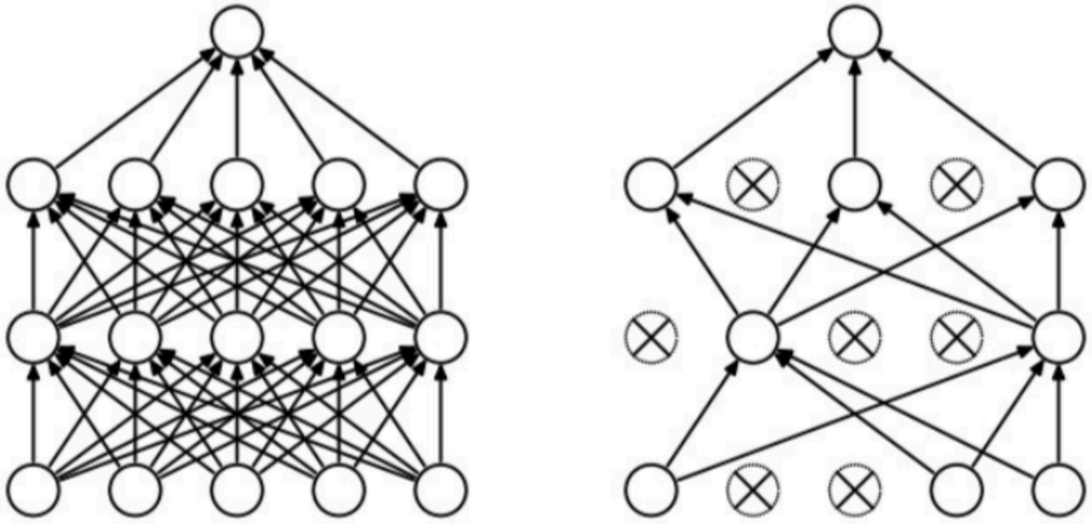
MOST CRITICAL ASPECTS IN THE TRAINING OF ANNs

- training speed:
  - mitigated by using stochastic-learning, momentum, adaptive learning rate (Adam o 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:
  - 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 itself

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 randomly eliminated during training (only on training step non validation) based on a defined probability
  - forces the model to not rely excessively on particular sets of features

before



after

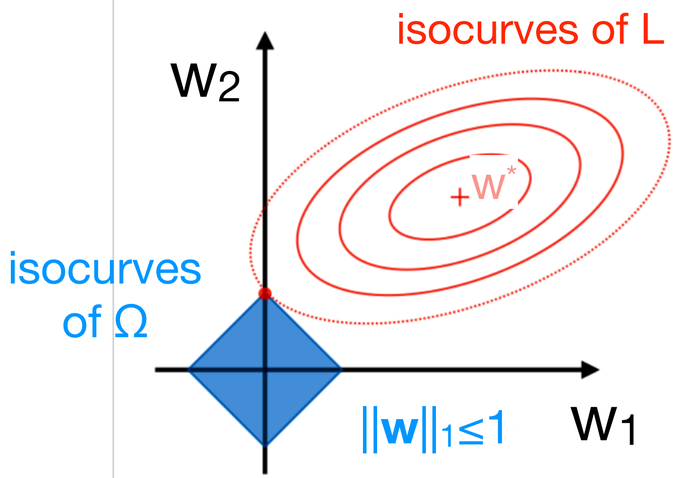
NOTE: random sampling is applied independently for each mini-batch (works like a sort of ensamble method)

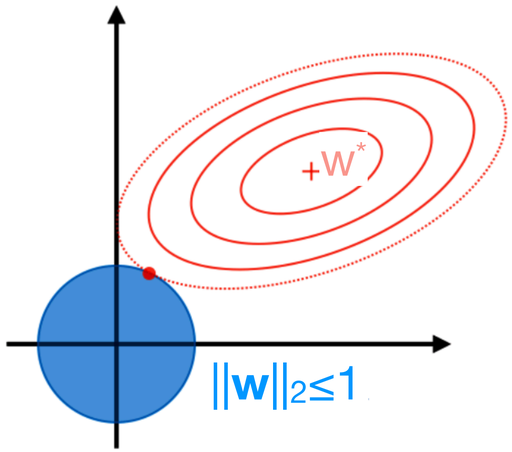
- used routinely in with convolutional-NN where it can sensibly increase performance on the test set
- can be also applied as preprocessing of the input (dropping part of the input, like some nodes in a point cloud or nodes/edges in a graph), or also after training (weight pruning) to improve generalisation and also to compress the model …

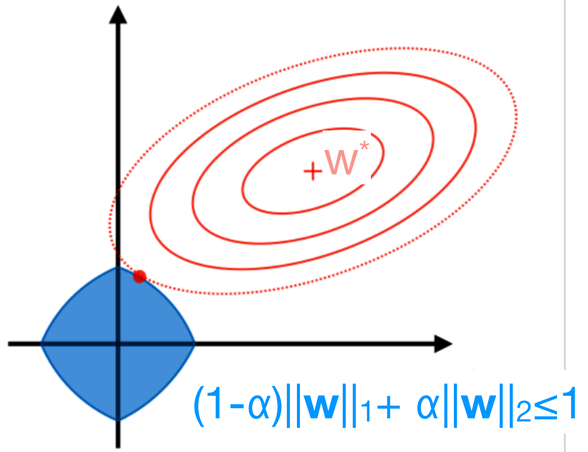
EXPLICIT WEIGHT PENALTIES: L1/L2/L3 REGULARIZATION

- 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(**w**) → L(**w**) + α Ω(**w**)

L1 **LASSO**	L2 **weight decay**	L1+L2 **Elastic Net**
• Shrinks coefficients to 0 • Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients







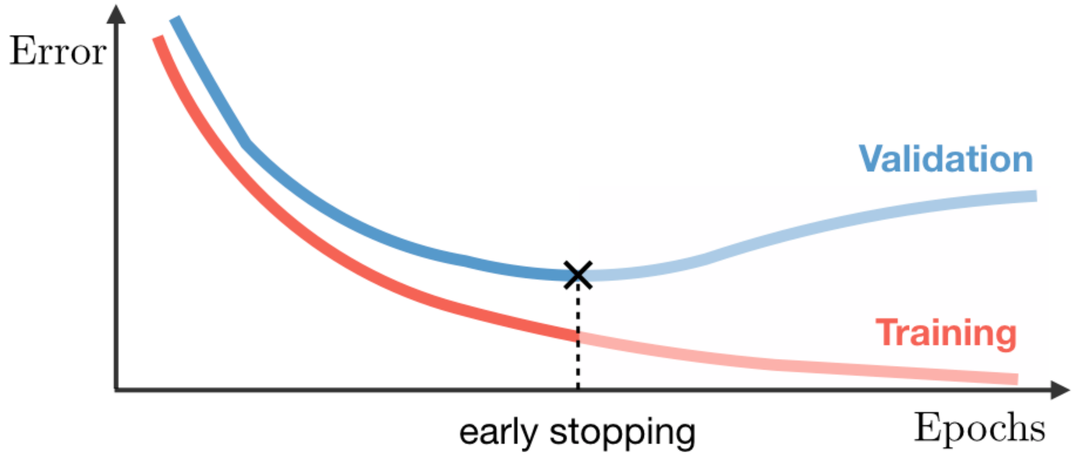
$$\Omega(\mathbf{w}) = \alpha \parallel \mathbf{w} \parallel_1 = \sum |w_k|$$

$$\Omega(\mathbf{w}) = \frac{\alpha}{2} \parallel \mathbf{w} \parallel_2^2 = \frac{\alpha}{2}\mathbf{w}^t\mathbf{w} = \sum w_k^2$$

$$\Omega(\mathbf{w}) = \lambda[(1-\alpha) \parallel \mathbf{w} \parallel_1 + \alpha \parallel \mathbf{w} \parallel_2^2]$$

EARLY STOPPING AND 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



always remember there are regimes in which training more improve generalisation (e.g. double descent)

- 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 withe/colored/gaussian noise to the features
  - …

Noise addition	Information loss
	
• Addition of noise • More tolerance to quality variation of inputs	• Parts of image ignored • Mimics potential loss of parts of image

DATA AUGMENTATION

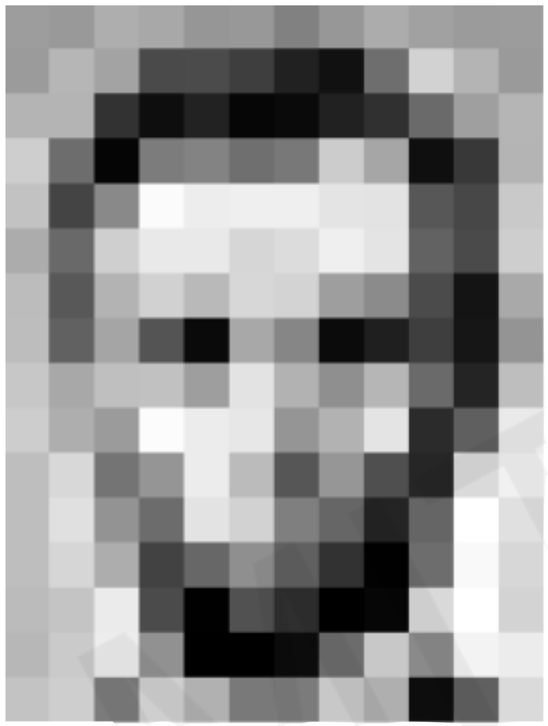
- a good way to make an ML algorithm to generalize better is to train it on larger and more expressive data
- but having more data is normally the real issue in ML/DL → solution: artificially increase the dimension of the training set by applying transformations that preserve the relevant “physics” of the data/problem

Original	Flip	Rotation	Random crop	Color shift	Contrast change
					
• Image without any modification	• Flipped with respect to an axis for which the meaning of the image is preserved	• Rotation with a slight angle • Simulates incorrect horizon calibration	• Random focus on one part of the image • Several random crops can be done in a row	• Nuances of RGB is slightly changed • Captures noise that can occur with light exposure	• Luminosity changes • Controls difference in exposition due to time of day

\+ modern approaches can be also based on data produced with generative models (GAN, VAE, …)

HOW A ANN “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	105	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
205	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	105	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
205	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

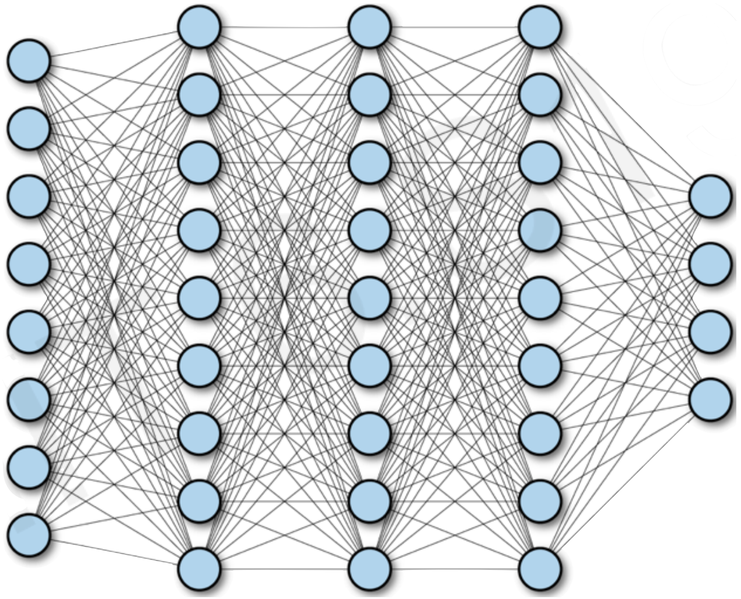
gray scale image with 8bit depth: 12x16x1 intensity $\in$ [0,256]
color image with n-bit depth: $m_1 x m_2 x 3$ with each RGB intensity $\in [0,2^n]$

credit MIT AI course

HOW AN ANN "SEE" AN IMAGE …

we know that a FFNN is able to learn the desired hierarchical representations:

x 1D array of the image pixel values



P(label | **x**)

However:

- it needs a huge number of learnable parameters (weights) → hard to train, overfitting …
- it does not use any local spatial information
- too much flexibility results in arbitrarily complex models for which is very hard to achieve generalisation

A remedy to facilitate the training, is to introduce task independent priors, i.e. parts of the model that are not learnt (called inductive relational biases), motivated by general properties observed in data

ARCHITECTURES FOR VISION: CNN

- Convolutional NN is one of these specific DNN architecture designed to excel in image recognition tasks

- Acts directly on the images (raw “pixel” information organised in a fixed size mesh)
- Implements several inductive biases typical of photographic images:
  - locality of the features: to identify a feature it takes just a few pixels in a small portion of the image
  - translation equivariance: features in the image remain the same in different points of the image
  - self-similarity: identical features can be recognised with a single filter
  - compositionality: a complex feature can be recognised by identifying only few of such sub-features

- Implementation idea: apply layers called convolutional filters that operate on the input by recognising the local 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

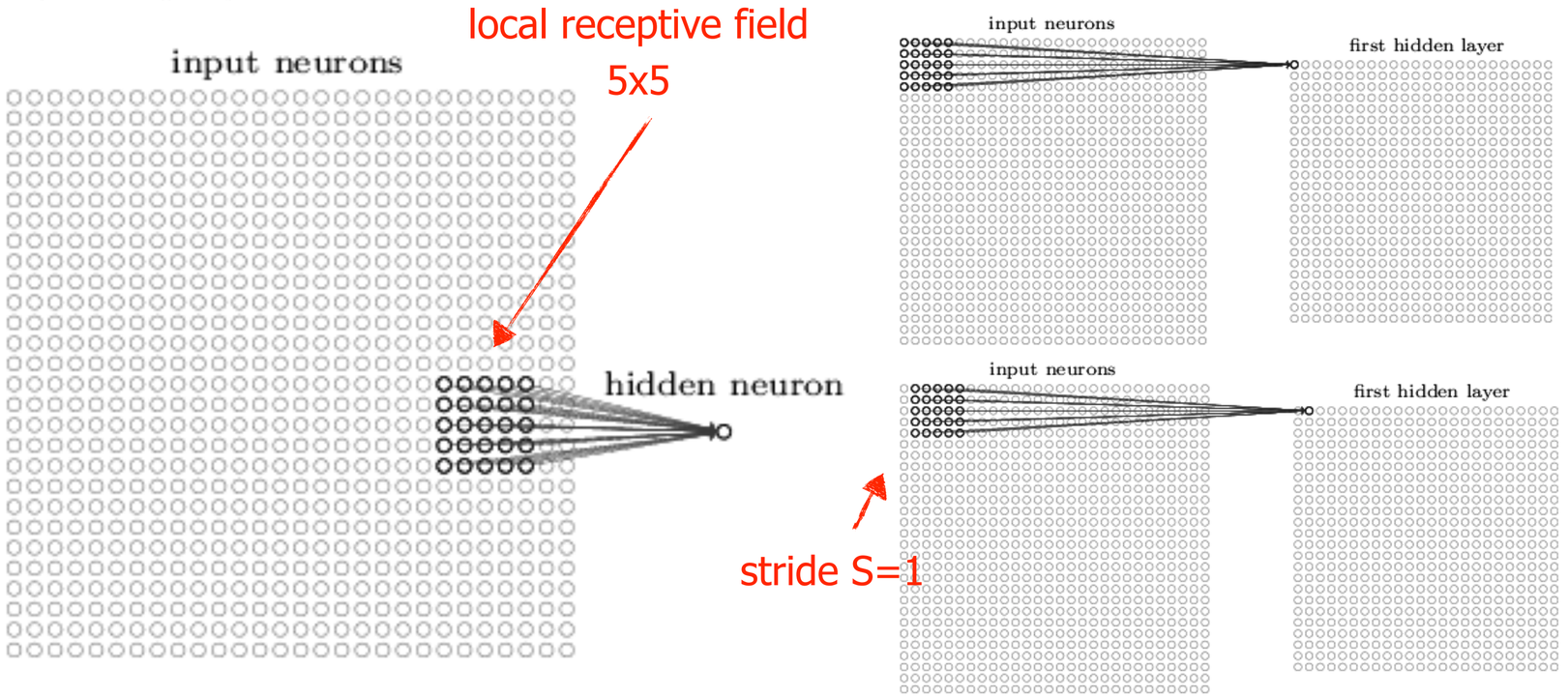
- CNNs learn from the training data sample the best set of filters to solve the task given the chosen architecture

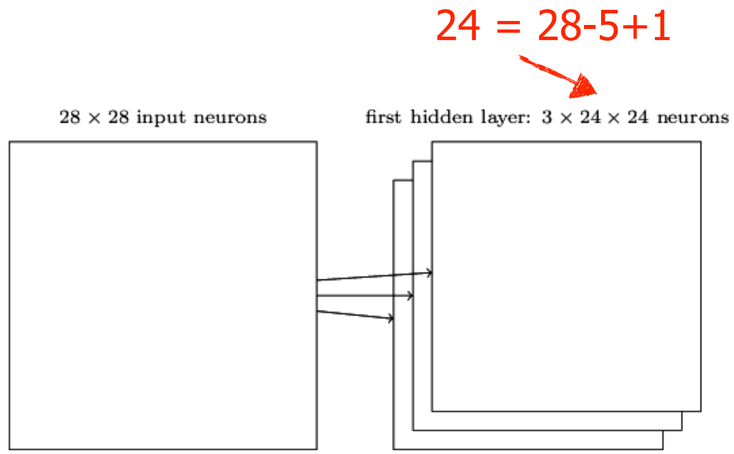
CONVOLUTIONAL FEATURE EXTRACTION LAYER

- used to identify similar features that are present in different position of the image

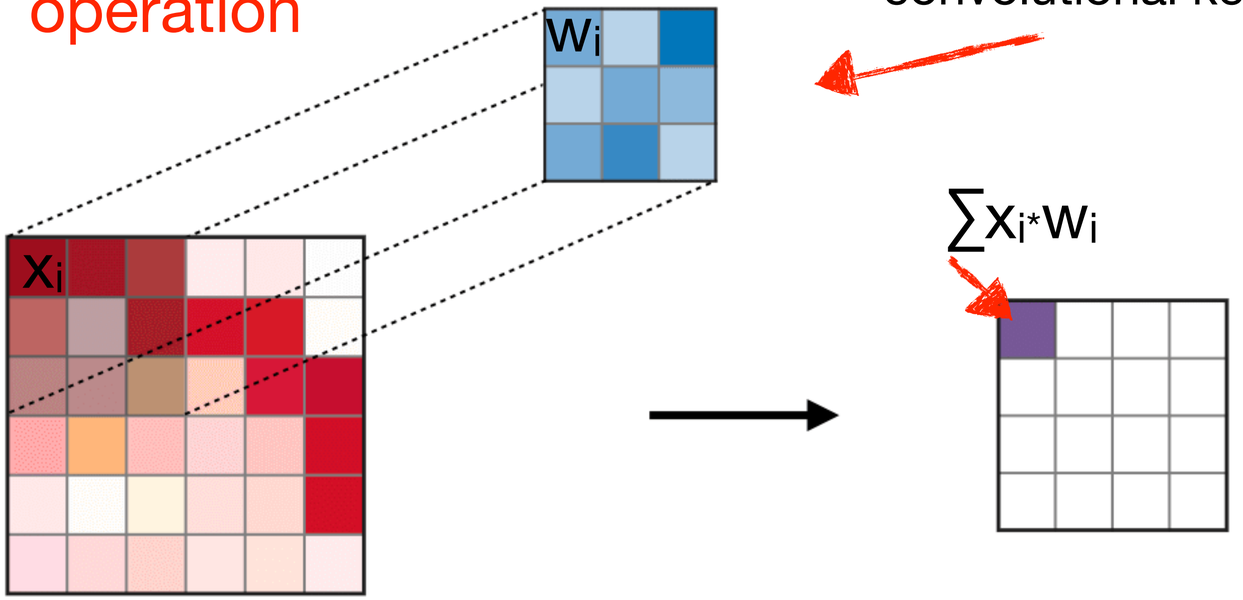
- based on three basic ideas:
  - local receptive field
  - shared-weights kernels
  - pooling layers

- 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

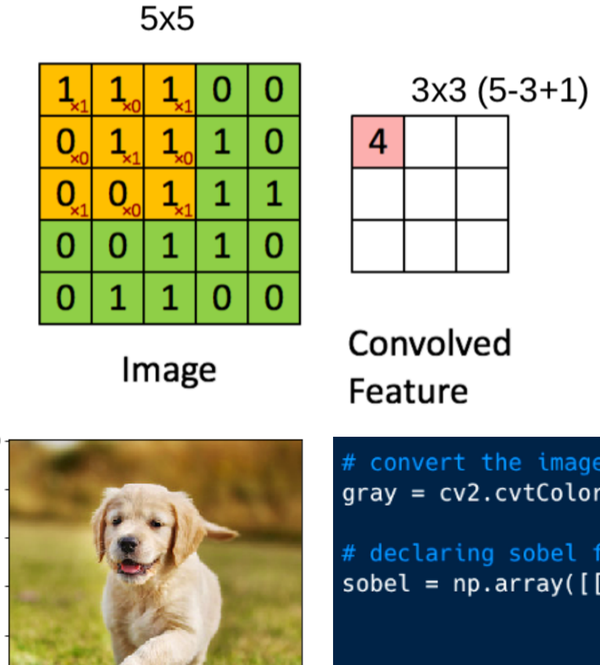




convolution operation



local receptive field or convolutional kernel

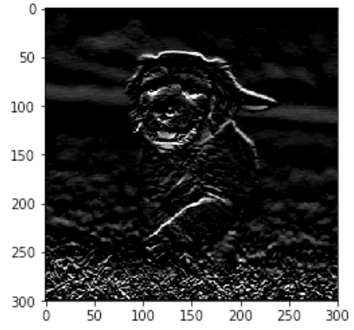



```
# convert the image to gray color
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# declaring sobel filter
sobel = np.array([[ -1, -2, -1],
                  [ 0, 0, 0],
                  [ 1, 2, 1]])

# applying sobel filter
filtered_image = cv2.filter2D(gray, -1, sobel_y)
# plotting the image
plt.imshow(filtered_image, cmap='gray')
```

after the convolution operation, non linearity is applied to each neutron of the filtered image via an activation (ex. ReLU)

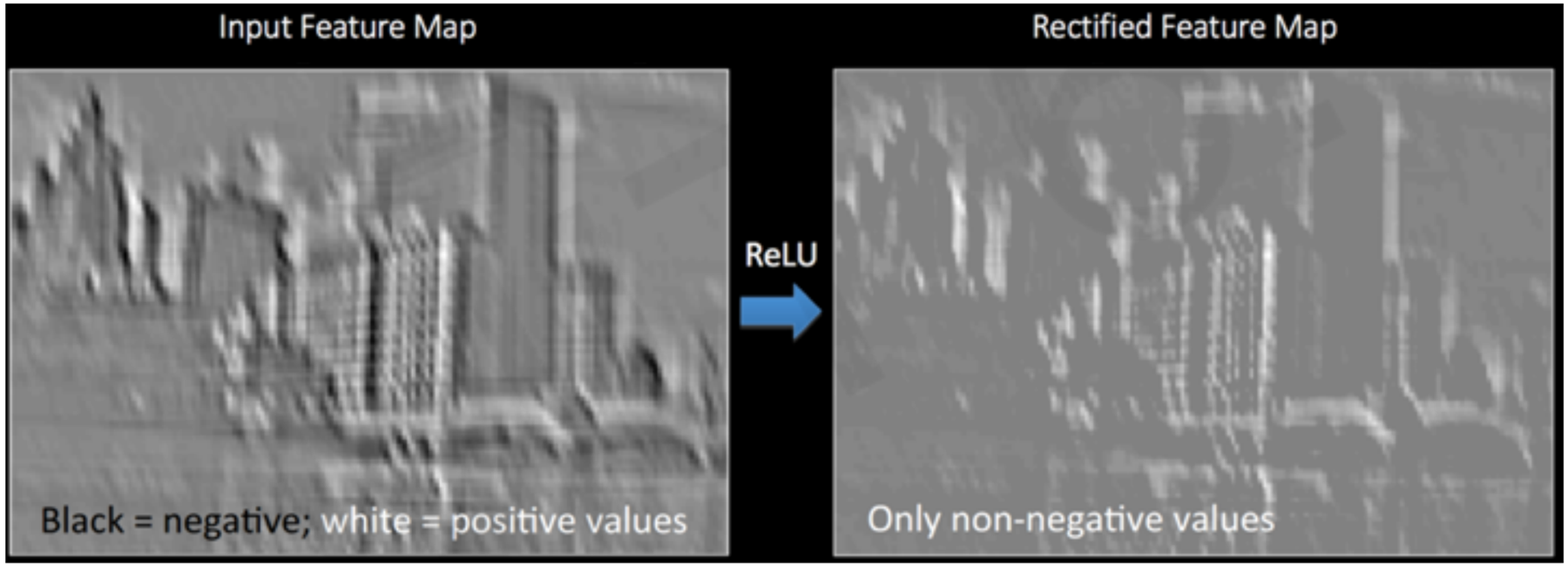


- shared-weights:
  - all the hidden neurons of a given hidden layer share the same weights → all neurons of the hidden layer detect the same sub-feature, only in different regions of the image
  - as the CNN has to identify many sub-features: there can be many convolutional kernels each one with an 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)

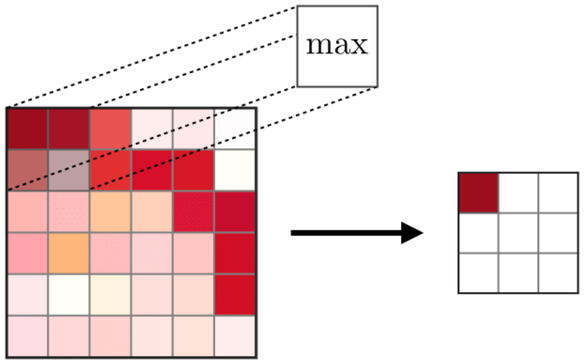
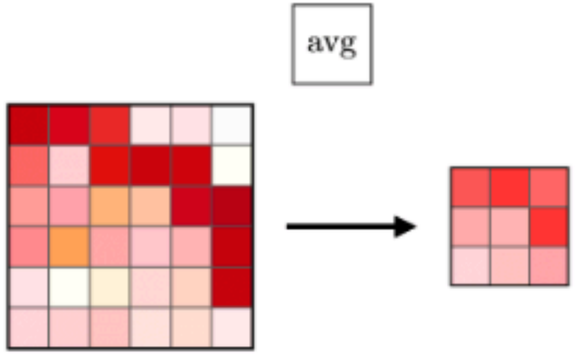
- emphasize only some of the dominant characteristics of the sub-features selected by the filter

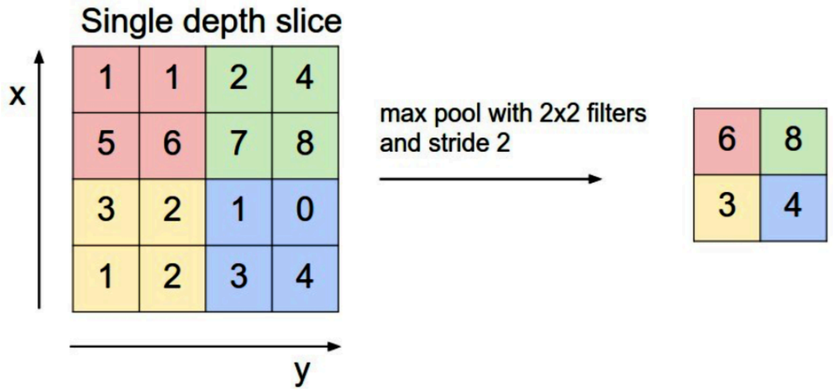


before ReLU

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	• Preserves detected features • Most commonly used	• Downsamples feature map • Used in LeNet

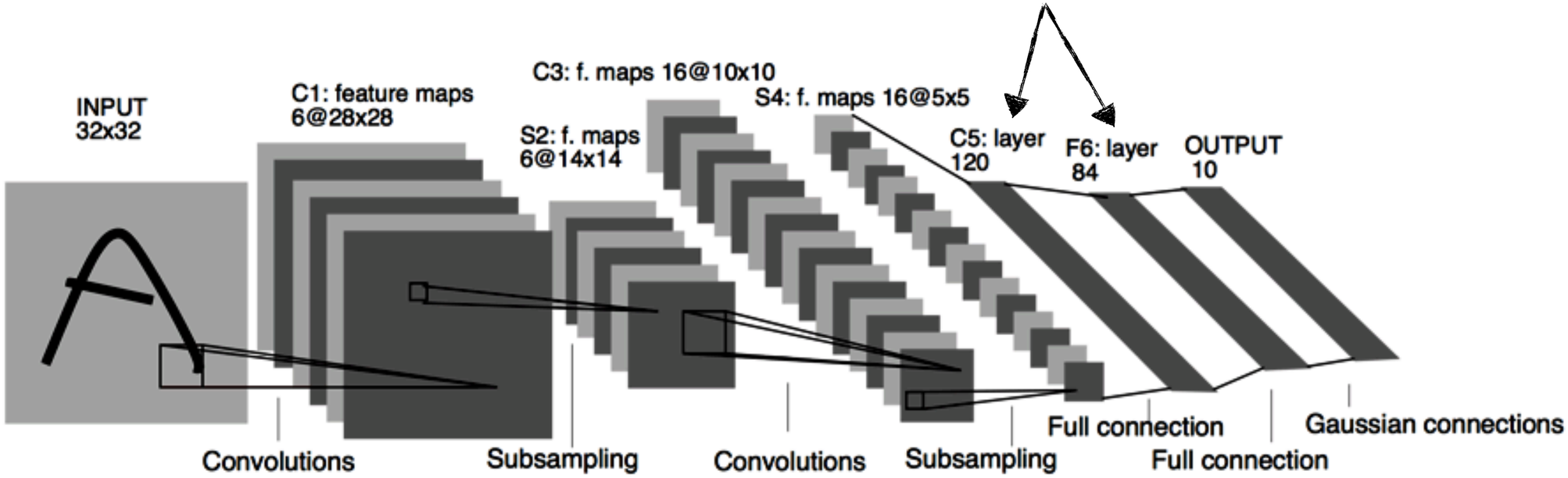


FULL CNN: CONV BLOCKS + DENSE MLP STAGE

- 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 optime objectives: class scores (classification), mapping (regression), etc…

Example: LeNet (Yan LeCun)

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



detects details (segments, arcs, …)

focus on overall shapes

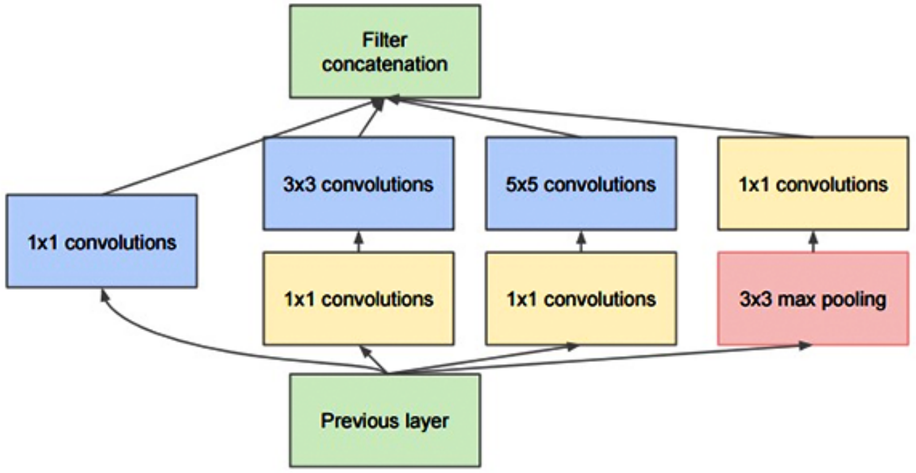
maps high level representations to targets

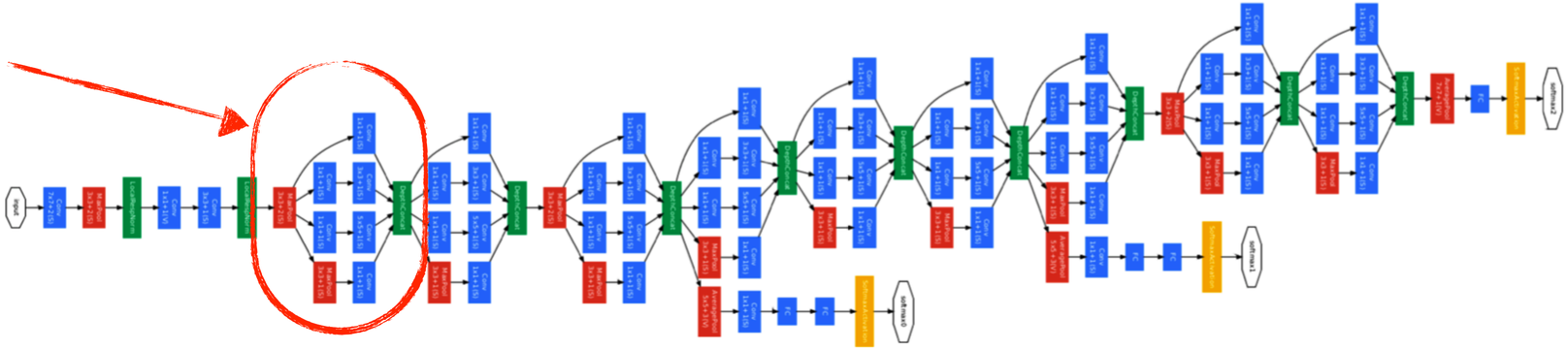
MODERN CNNs

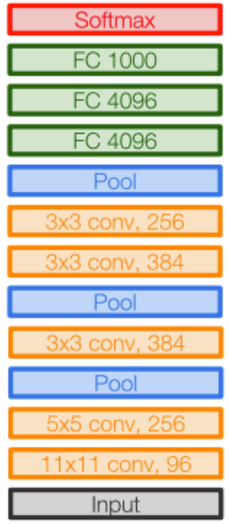
philosophy: deeper is better …

- 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 …





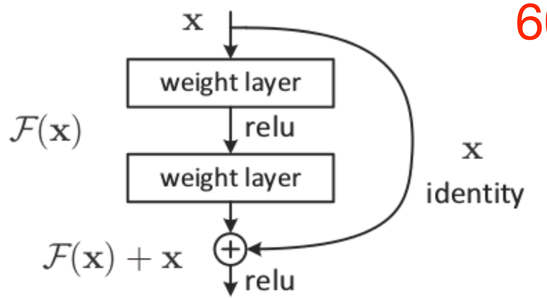


AlexNet

VGG19

ResNet and DenseNet

going deeper increase the vanishing gradient problem
residual learning in ResNet help mitigating it



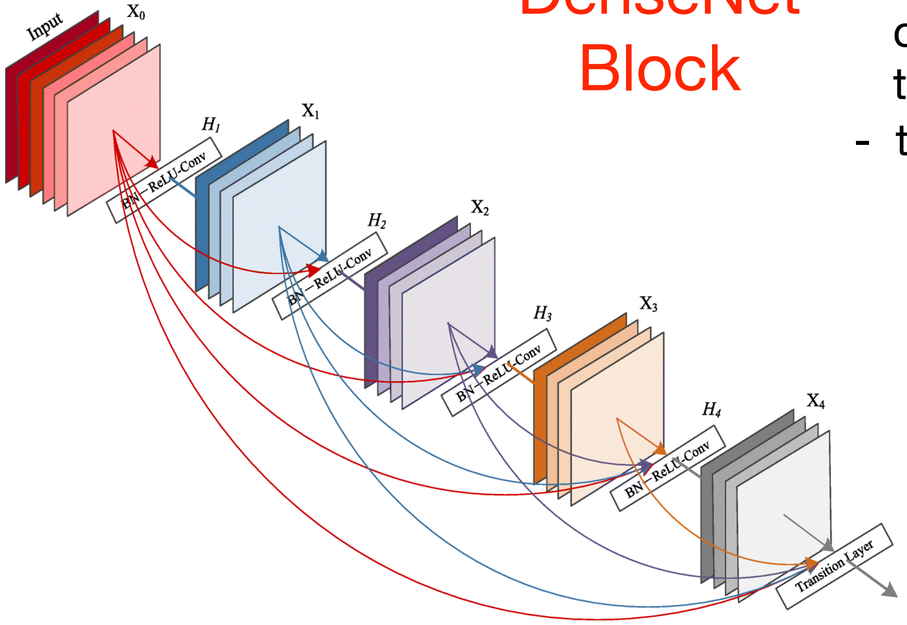
ResNet-152

…. continue for 152 layers

60 MPar

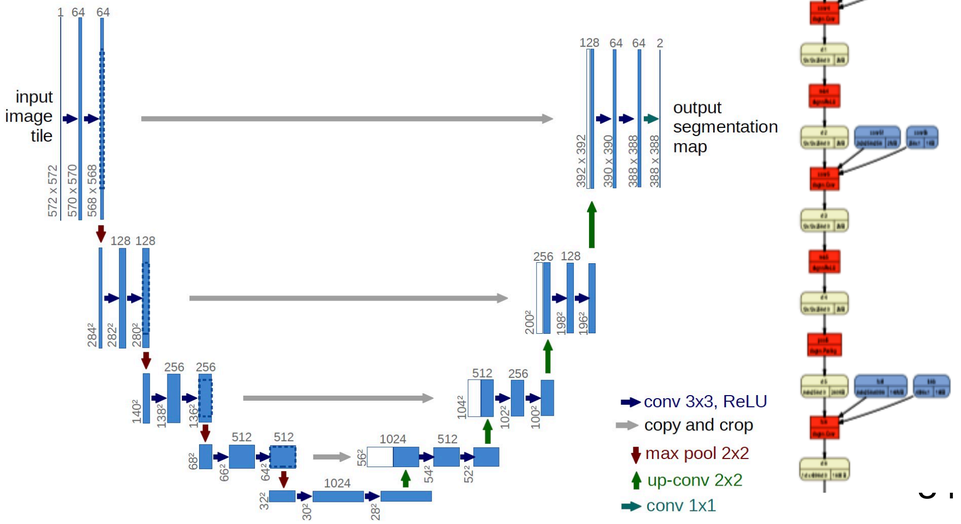
<u>estreme extension of the idea</u>: connect entire blocks of layers to one another
helps in diversifying the features within these blocks

DenseNet Block



- main difference were ResNets is that feature maps are concatenated instead of added (better feature reuse but requires transition downsampling layers to keep same size in feature maps)
- to avoid memory explosion use 1x1 bottlenecks

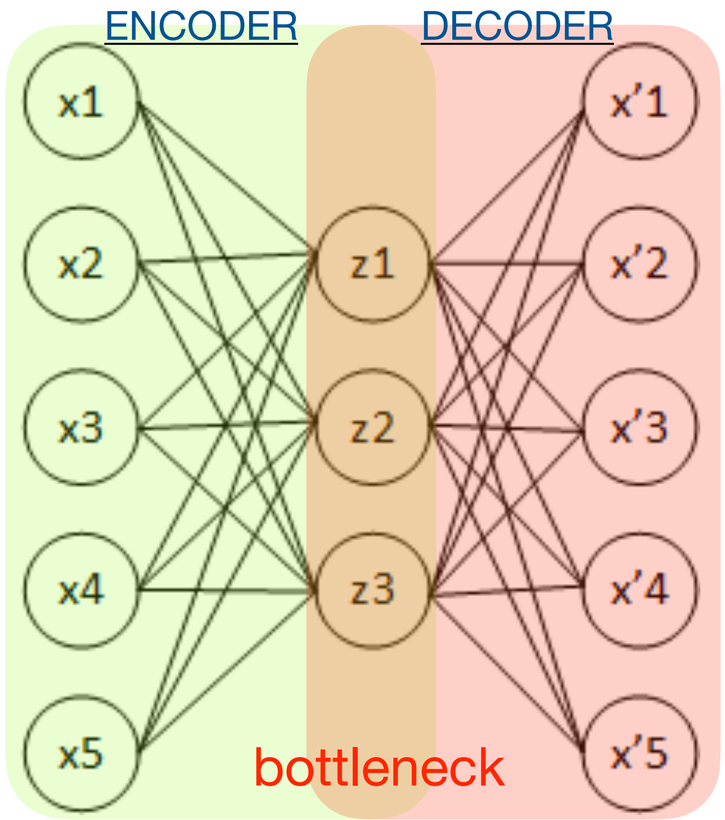
<u>specialized evolution</u>: UNet architecture, CNN for image segmentation and denoising (see specific lectures on object-detection)



ANN ARCHITECTURES FOR UNSUPERVISED REPRESENTATION LEARNING: AUTOENCODERS

- non-supervised algorithm that try to identify common and fundamental characteristic in the input data
- combines an 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)

input
v(5)



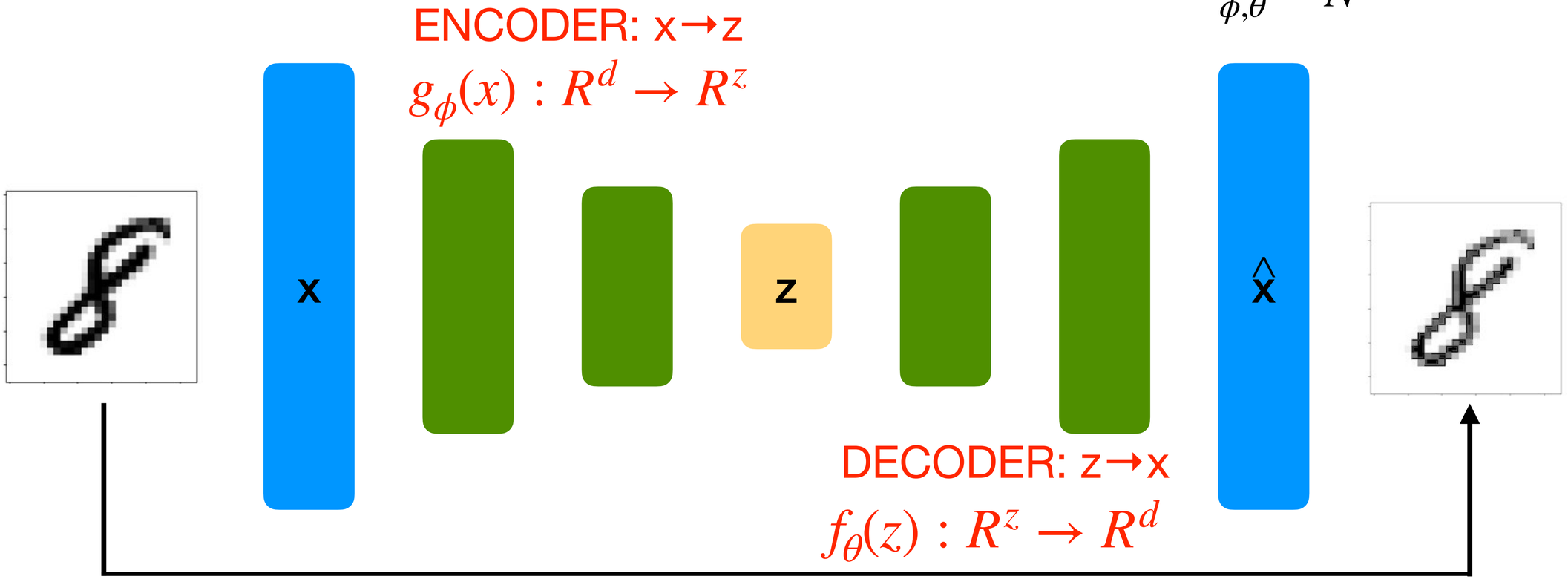
output = input

- "trivial" unless to constrain the network to learn a "compressed" latent representations (in the example: $x \in R^5 \rightarrow z \in R^{3)}$, able to capture "high-level" features of the data

AUTO-ENCODER IMPLEMENTATION

$$\phi^*, \theta^* = \arg\max_{\phi,\theta} \frac{1}{N} \sum L(x^{(i)}, \hat{x}^{(i)}) =$$

$$= \arg\max_{\phi,\theta} \frac{1}{N} \sum L(x^{(i)}, f_\theta(g_\phi(x))$$



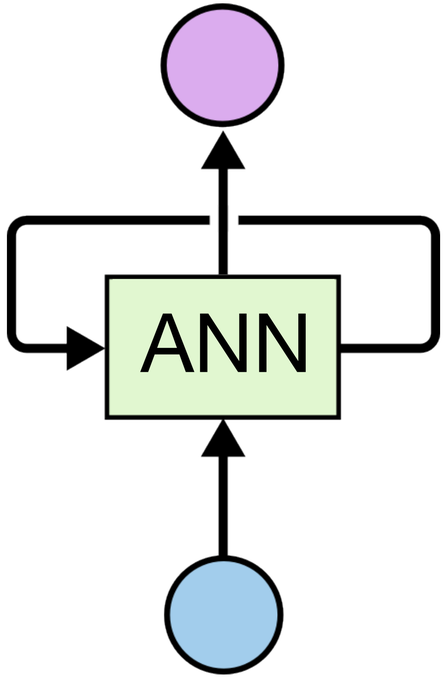
trained so that: Output ≃ Input

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

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

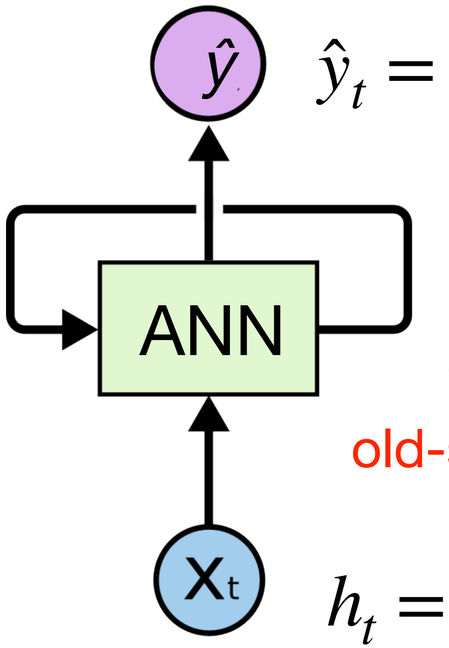
ARCHITECTURES FOR SEQUENCES: RECURRENT NEURAL NETWORKS

- 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 is able to:
  - takes in input sequences of variable length
  - keep track of dependences between elements thatare distant in the sequence
  - keep information on the order between the elements of the sequence
  - use shared weights so that identified correlations between elements can be transferred in diverse points of the sequence



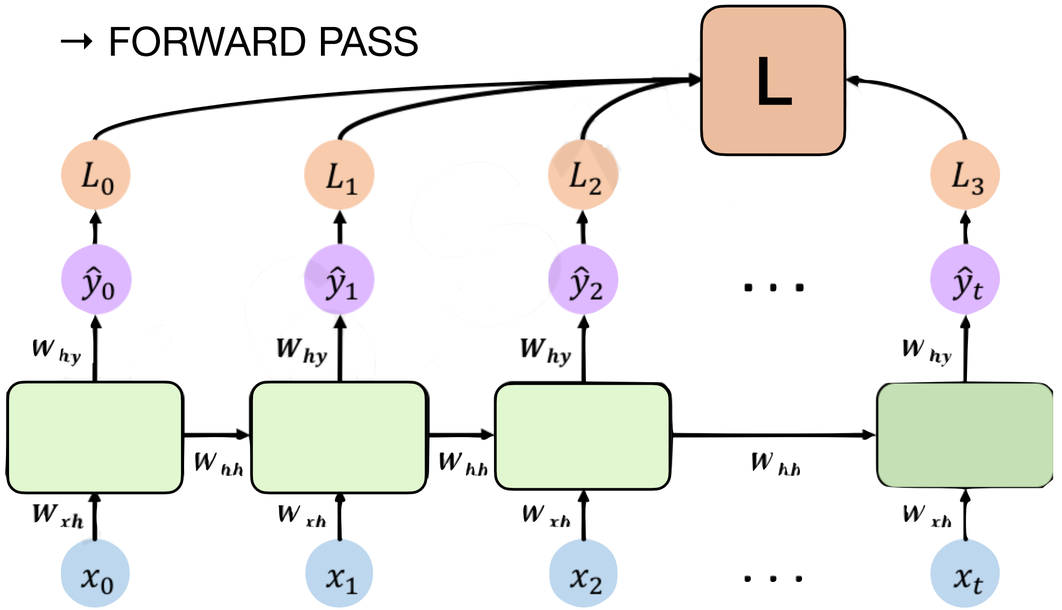
RNN IMPLEMENTATION

- 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



$$\hat{y}_t = W_{hy}^t h_t$$

h_t =



can be thought of as a series of multiple copies of the same conventional neural network, each passing a message to its successor

old-state

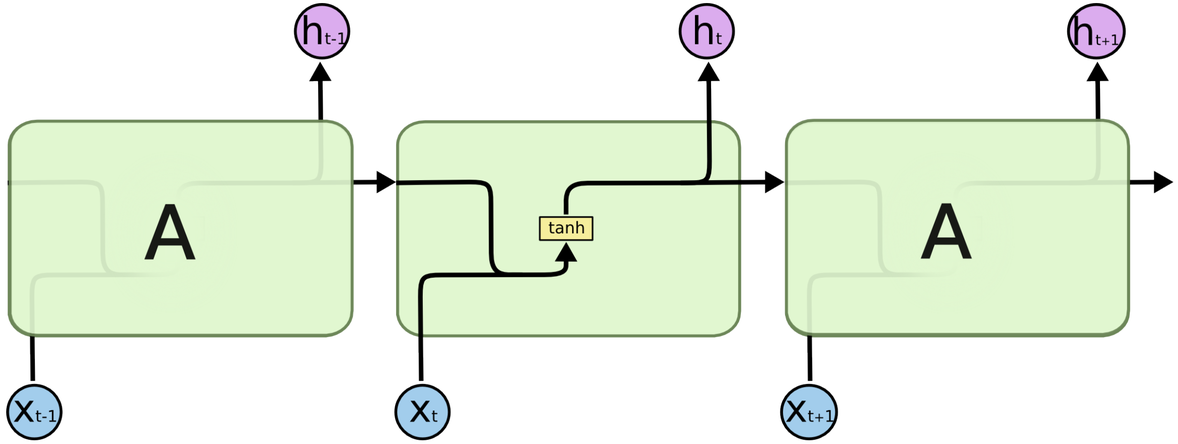
$$h_t = f_w(h_{t-1}, x_t) =$$

sequence input at step *t*

$$= \tanh(W_{hh}^t h_{t-1} + W_{xh}^t x_t)$$

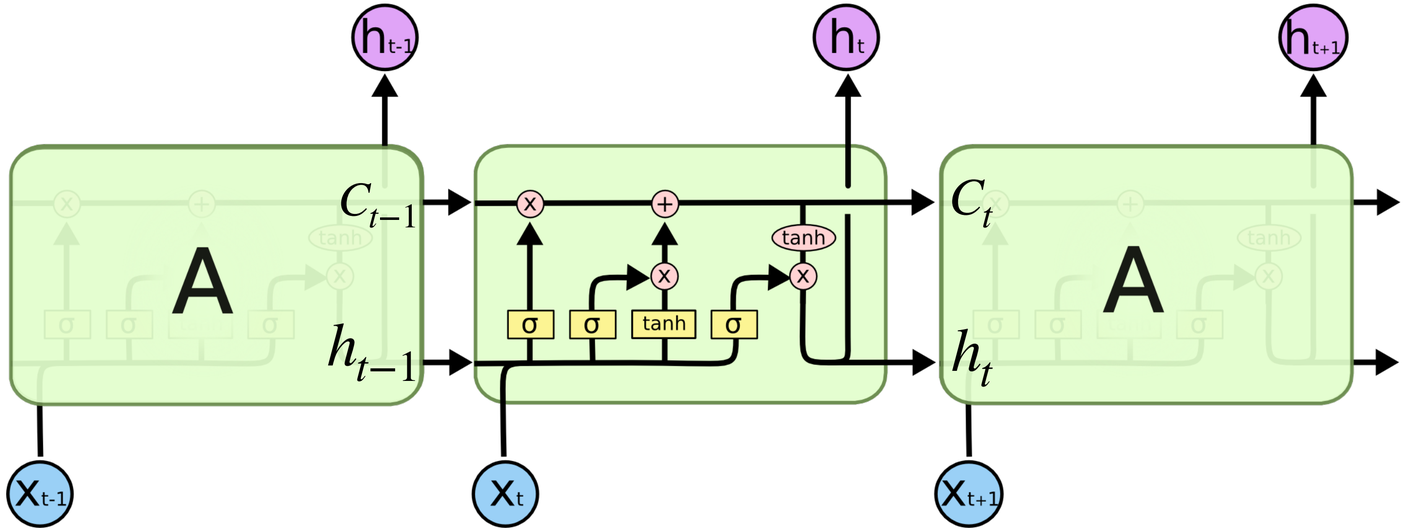
the same function f_w with the same set of weights is used to process each element t of the sequence …

RNN AND LONG TERM DEPENDENCIES



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

problem solved in LSTM RNN (Hochreiter, '97) with a "software trick": uses four neural layers which interact in such a way to implement a sort of parallel data-flow which at each step t makes the information from previous steps available w/o being affected by gradient dilution

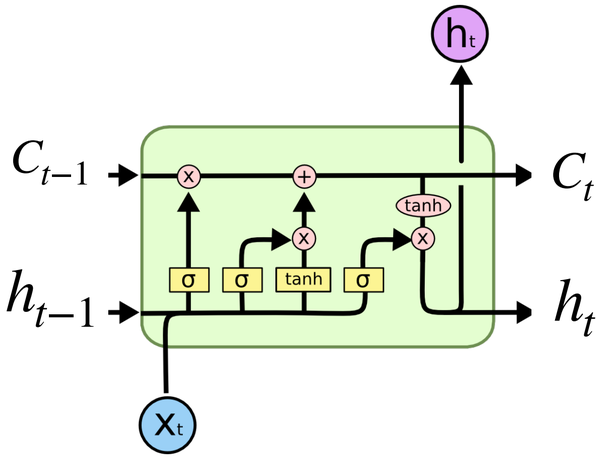


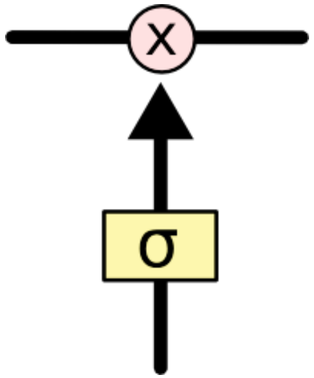
key element: cell-state C_t

is a memory units (“conveyor belt”) to which is possibile to add or subtract information using “gate” structures

LONG SHORT TERM MEMORY NETWORKS

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





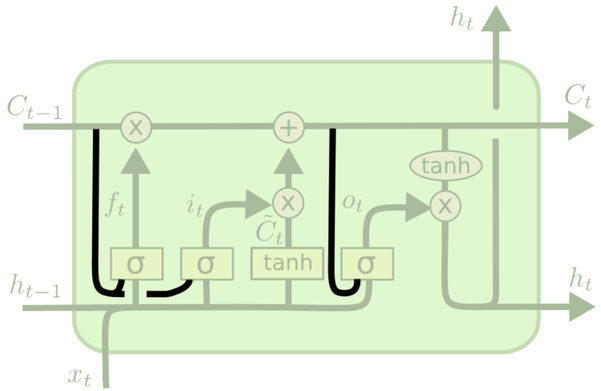
output ∈[0,1]:

every LSTM has 3 gates:

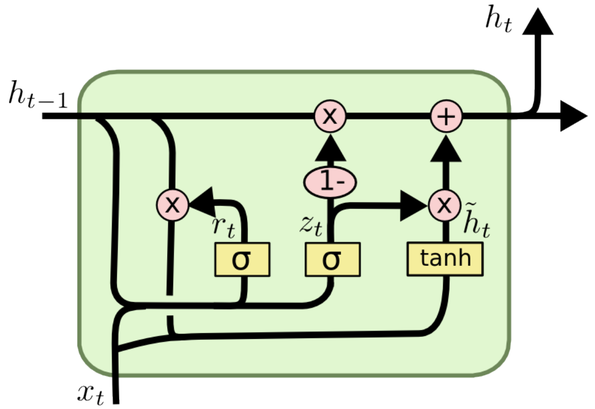
- f: forget gate (controls deleting from the cell-state)
- i: input gate (controls writing on the cell-state)
- o: output gate (controls the output on h_t)

the backprop from $C_t \rightarrow C_{t-1}$ doesn't requires multiplications for tanh/sigmoid → 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

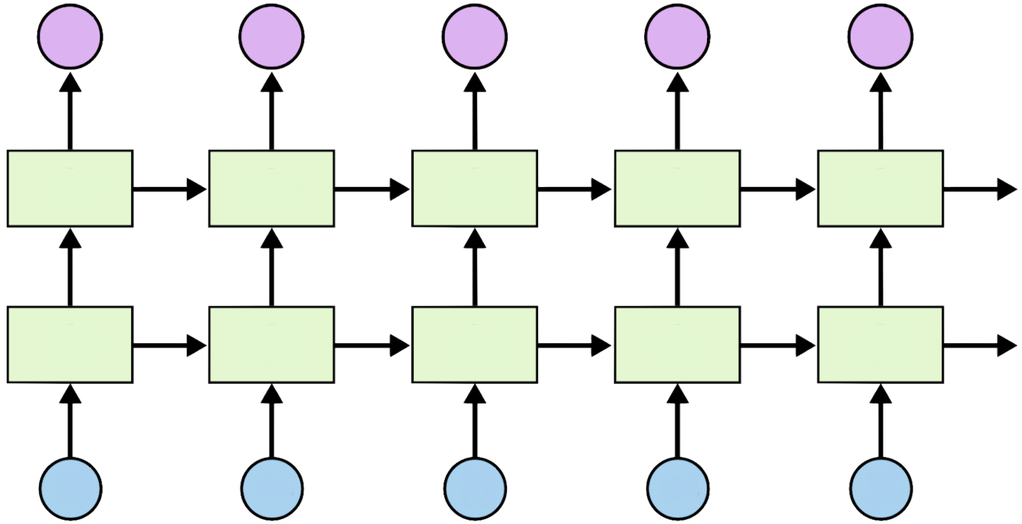


GRU (Gate Recurrent Unit):
combines the gates and unify hidden state with cell-state to simplify model and number of parameters (one of the most used RNNs)

VARIATIONS

Stacked RNNs

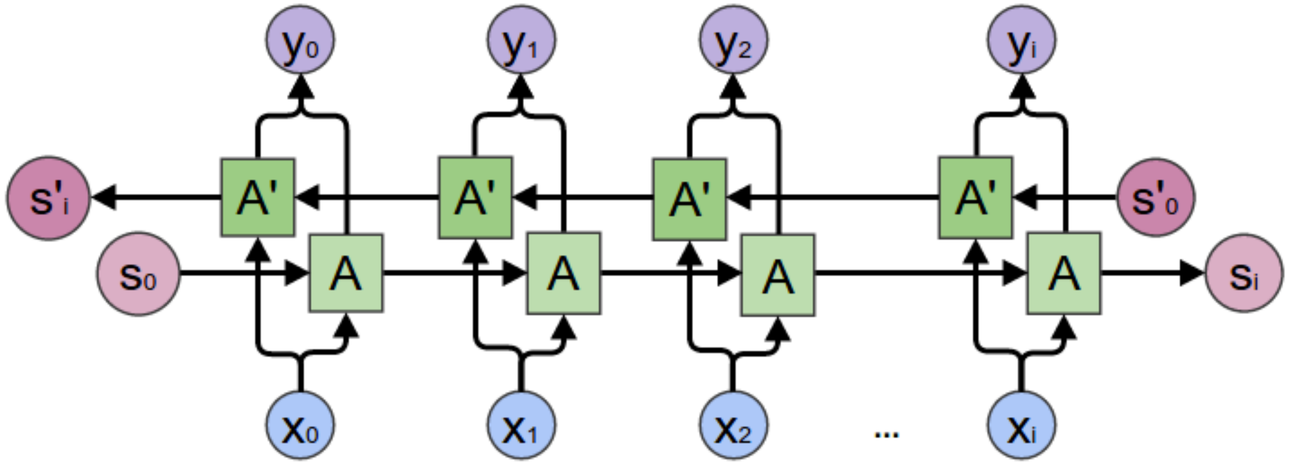
enables more complex representation
capturing information at different scales



```
# Initialize the RNN.
rnn = nn.RNN(input_size, hidden_size, num_layers = 2,
             nonlinearity='tanh', batch_first=True)
```

Bidirectional RNNs

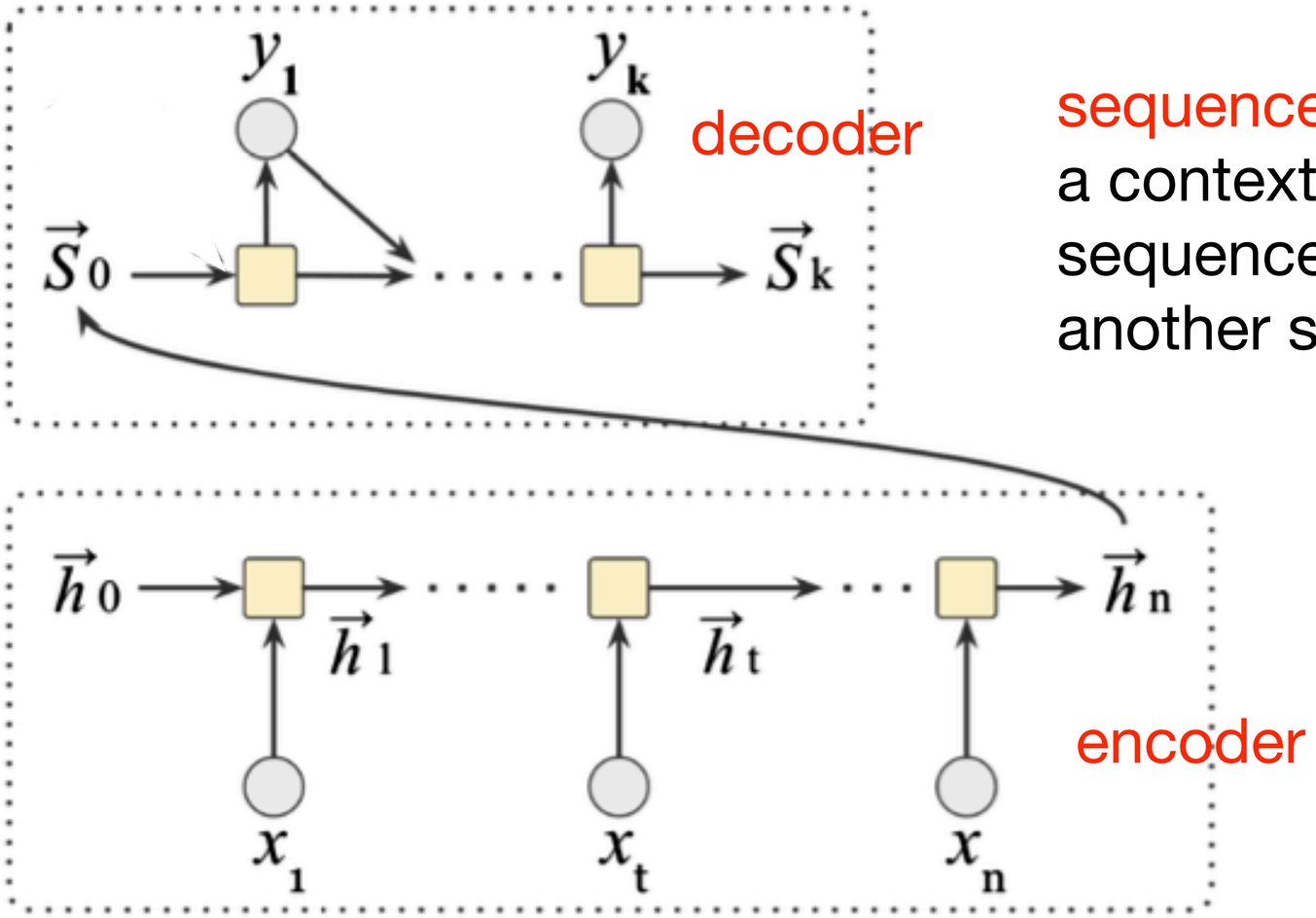
inputs are processed in both forward
and reverse time order
allows a BRNN to look at future
context as well




```
# Initialize the RNN.
rnn = nn.RNN(input_size, hidden_size, num_layers = 1,
             nonlinearity='tanh',
             bidirectional = True, batch_first=True)
```

VARIATIONS

Encoder-Decoder RNNs



sequence2sequence: uses an RNN to generate a context vector that encode the whole input sequence and a separate RNN to generate another sequence from the encoded context

often used in machine translation

KEEP IN TOUCH ...

FUTURE AI RESEARCH

SCAN ME

This activity is partially supported by ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing, funded by European Union – NextGenerationEU, and by PNRR MUR project PE0000013-FAIR