Machine Learning: Lecture 2

Jennifer Ngadiuba (Fermilab)
University of Pavia, May 8-12 2023

Overview of the lectures

- **Day 1:**
  - Introduction to Machine Learning fundamentals
  - Linear Models
- **Day 2:**
  - Neural Networks
  - Deep Neural Networks
  - Convolutional Neural Networks
- **Day 3:**
  - Recurrent Neural Networks
  - Graph Neural Networks (part 1)
- **Day 4:**
  - Graph Neural Networks (part 2)
  - Transformers
- **Day 5:**
  - Unsupervised learning
  - Autoencoders
  - Generative Adversarial Networks
  - Normalizing Flows

Hands on sessions each day will closely follow the lectures topics

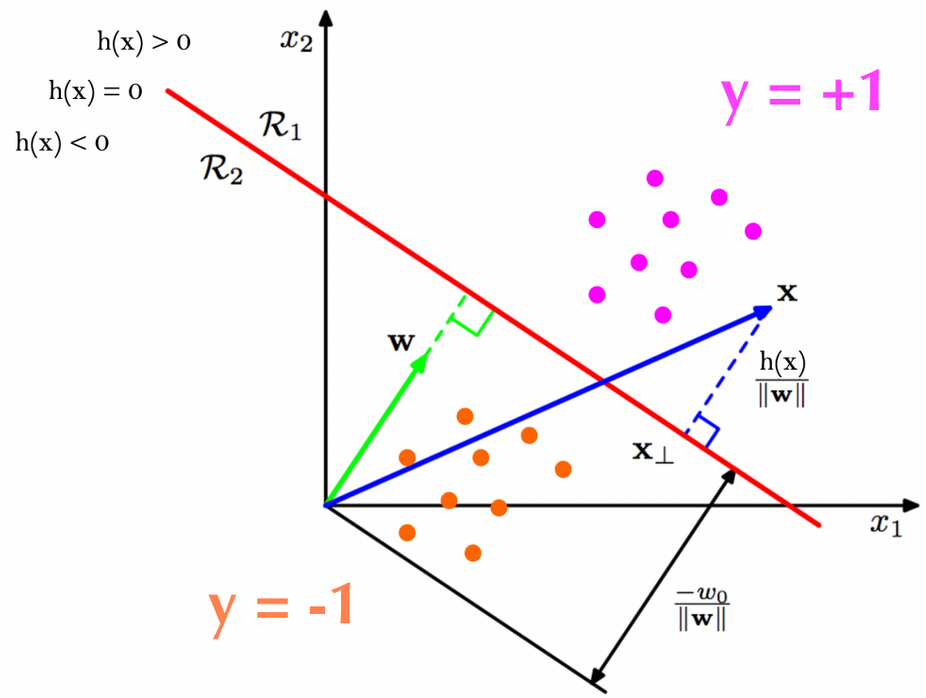
Lectures are adapted from M. Kagan's courses

Table of contents

Intro to neural networks

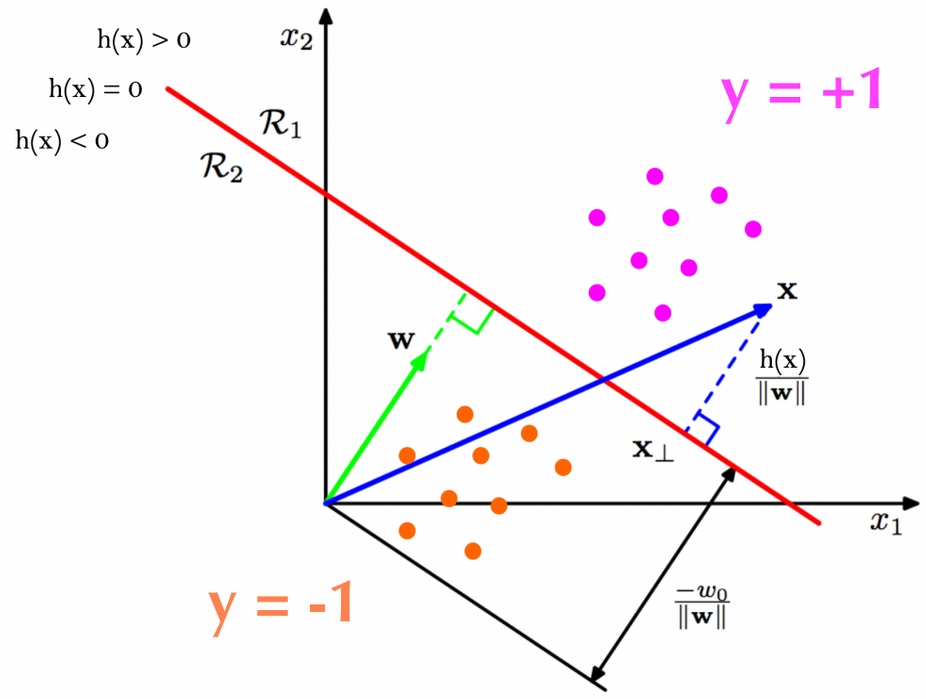
Recap: logistic regression

- Task: learn a function to **separate different classes of data** — for two classes we call this "binary classification":
  - $\mathbf{x}_i \in \mathbb{R}^m$
  - $y_i \in \{-1, 1\}$ — we call these "true labels"
- **Linear decision boundary:** $h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$

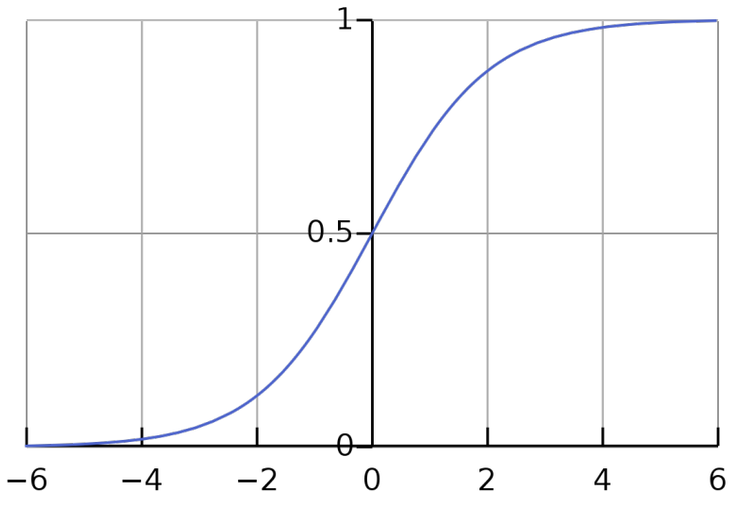


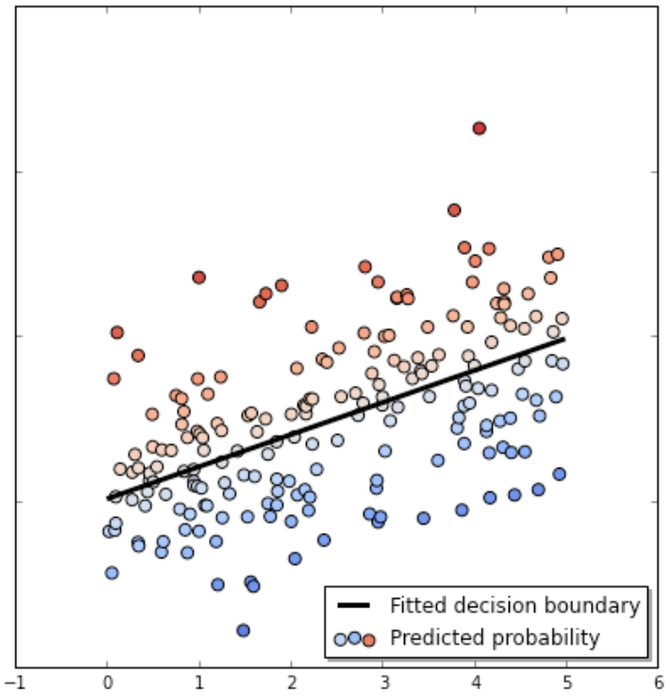
Recap: logistic regression

- Task: learn a function to **separate different classes of data** — for two classes we call this "binary classification":
  - $\mathbf{x}_i \in \mathbb{R}^m$
  - $y_i \in \{-1, 1\}$ — we call these "true labels"
- **Linear decision boundary:** $h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$
- Distance from decision boundary is converted to class probability using **logistic sigmoid function**

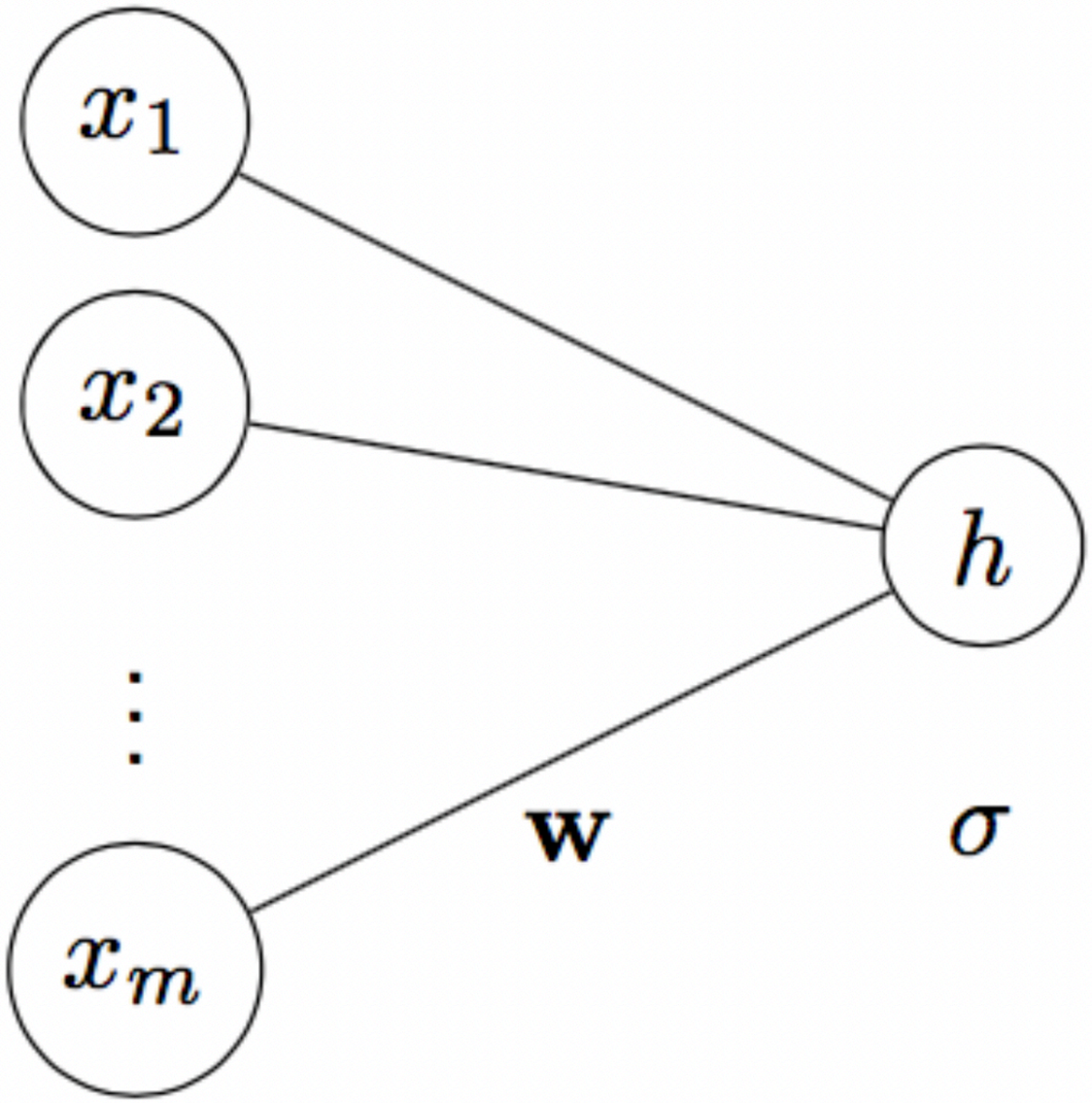


$$p(y = 1 \mid \mathbf{x}) \equiv p_i = \frac{1}{1 + e^{-h(\mathbf{x};\mathbf{w})}} = \frac{1}{1 + e^{-\mathbf{w}^{\mathrm{T}}\mathbf{x}}}$$





Recap: logistic regression



Represent logistic regression as a graph:
this unit is the main building block of a Neural Network!

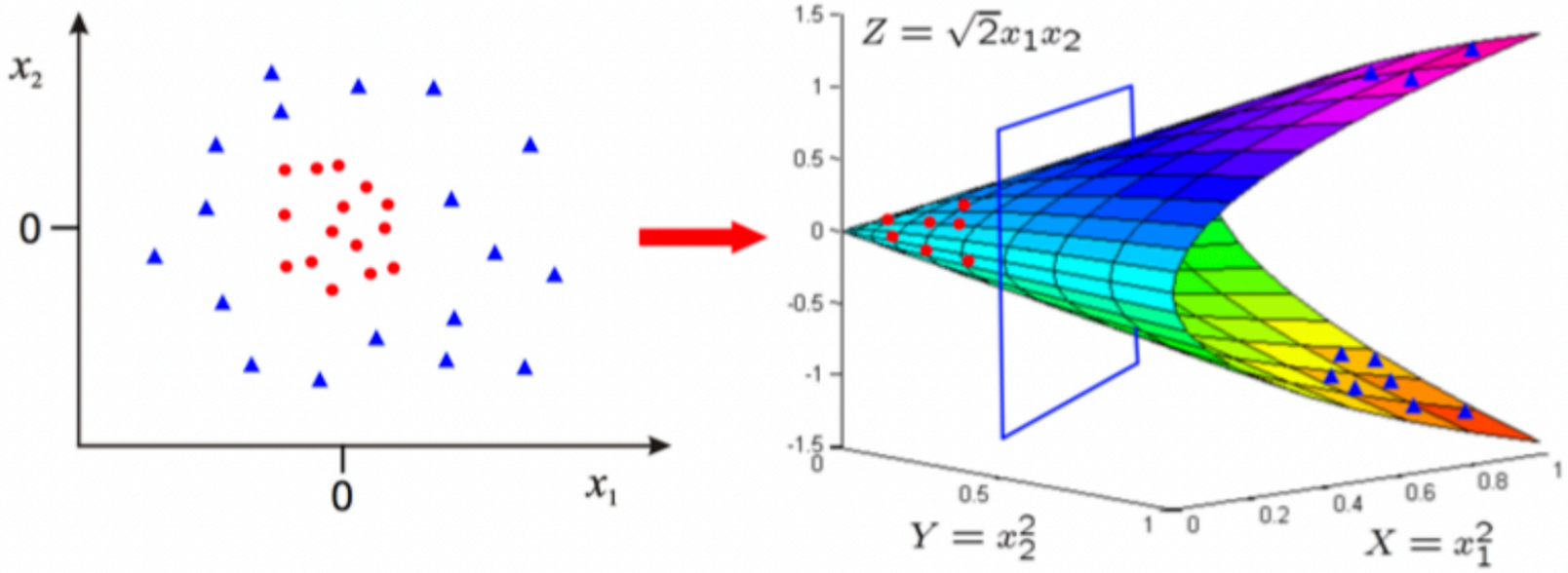
$$p(y = 1 \mid \mathbf{x}) = \sigma(h(\mathbf{x}, \mathbf{w})) = \frac{1}{1 + e^{-h(\mathbf{x};\mathbf{w})}} = \frac{1}{1 + e^{-\mathbf{w}^{\mathrm{T}}\mathbf{x}}}$$

Adding non-linearity

- What if we want a non-linear decision boundary?

  - chose basis, e.g.: $\phi(x) \sim \{x^2, \sin(x), \log(x), \dots\}$

$$\Phi : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \quad \mathbb{R}^2 \rightarrow \mathbb{R}^3$$



Adding non-linearity

- What if we want a non-linear decision boundary?
  - chose basis, e.g.: $\phi(x) \sim \{x^2, \sin(x), \log(x), \dots\}$
- Caveat: how to chose the right basis functions

Adding non-linearity

- What if we want a non-linear decision boundary?
  - chose basis, e.g.: $\phi(x) \sim \{x^2, \sin(x), \log(x), \dots\}$
- Caveat: how to chose the right basis functions
- Learn the basis functions directly from data

$$\phi(\mathbf{x}; \mathbf{u}) : \mathbb{R}^m \longrightarrow \mathbb{R}^d$$

- where **u** is a set of parameters for the transformation

Neural Networks

- Define the basis functions $j = \{1,...,d\}$: $\quad \phi_j(\mathbf{x}; \mathbf{u}) = \sigma(\mathbf{u}_j^{\mathrm{T}}\mathbf{x})$

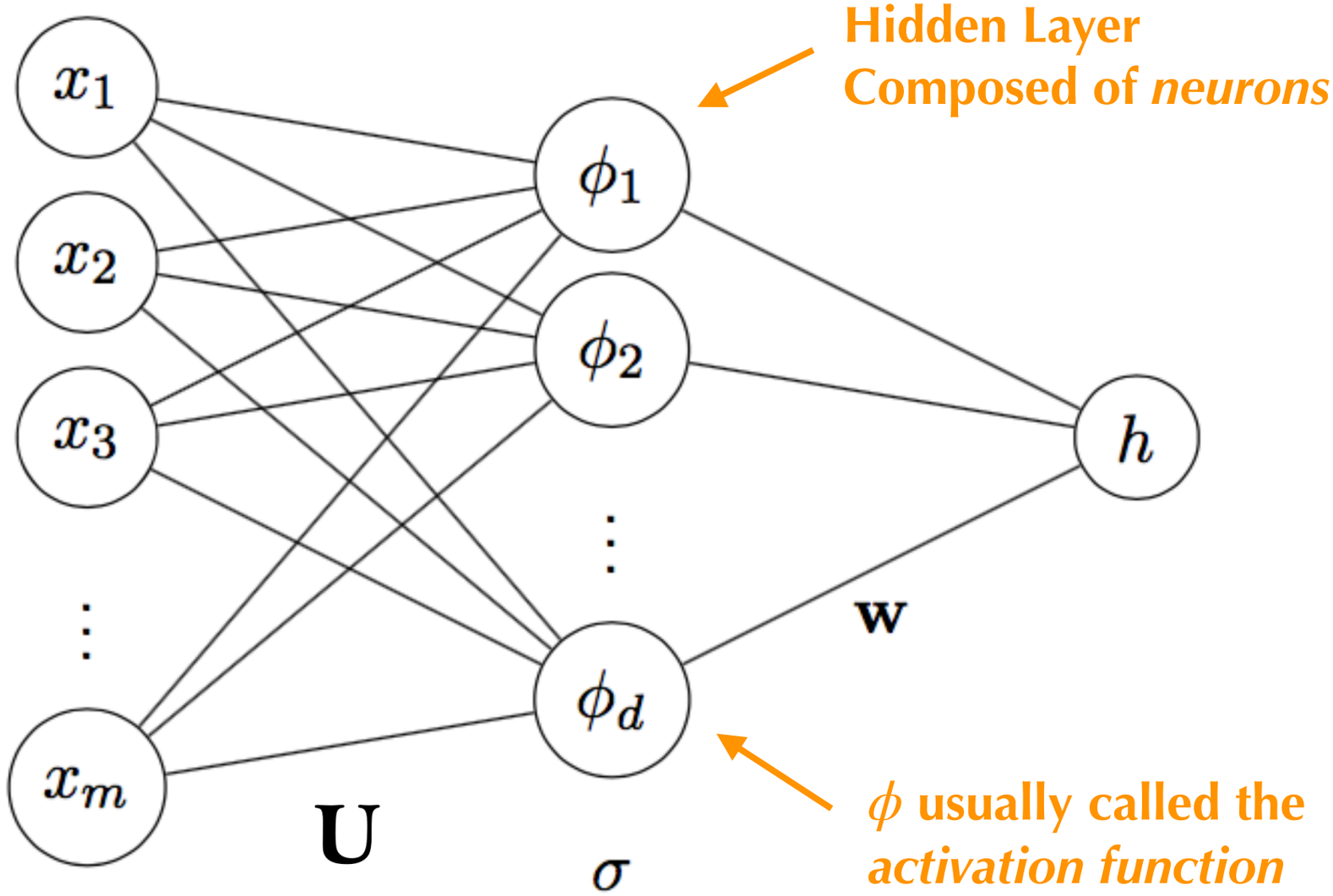
- Put all $\mathbf{u}_j \in \mathbb{R}^m$ vectors into matrix $\mathbf{U}$

$$\phi(\mathbf{x}; \mathbf{U}) = \sigma(\mathbf{U}\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{u}_1^{\mathrm{T}}\mathbf{x}) \\ \sigma(\mathbf{u}_2^{\mathrm{T}}\mathbf{x}) \\ \vdots \\ \sigma(\mathbf{u}_d^{\mathrm{T}}\mathbf{x}) \end{bmatrix} \quad \in \mathbb{R}^d$$

  - σ is a pointwise sigmoid acting on each vector element

- Full model becomes $\quad h(\mathbf{x}; \mathbf{w}, \mathbf{U}) = \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}; \mathbf{U})$

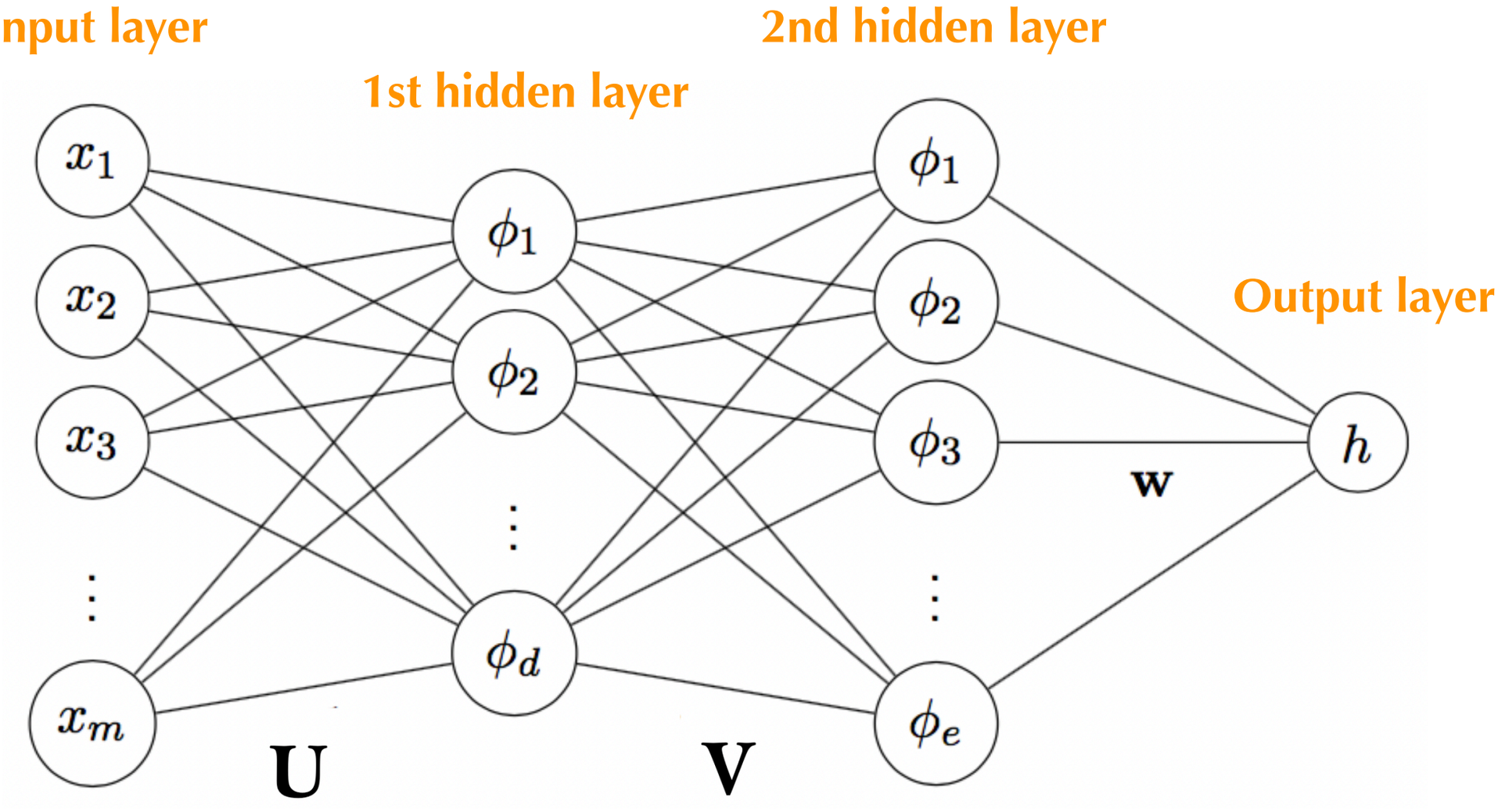
Neural Networks



$$\phi(\mathbf{x}) = \sigma(\mathbf{U}\mathbf{x})$$

$$h(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x})$$

Multi-layer Neural Networks



Each layer adapts basis based on previous layer

These types of networks are called **Multi Layer Perceptrons (MLP)**

Universal approximation theorem

- Feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate continuous functions arbitrarily well on a space $\mathbb{R}^n$
    - only mild assumptions on non-linear activation function needed. Sigmoid functions work, as do others.
- But no other information are added on how many neurons needed, or how much data!
- How to find the parameters, given a dataset, to perform this approximation?

Training

Neural Networks optimization

- Neural Network model: $h(\mathbf{x}; \mathbf{w}, \mathbf{U}) = \mathbf{w}^{\mathrm{T}} \sigma(\mathbf{U}\mathbf{x})$

- **Classification:** cross-entropy loss function

$$\mathscr{L}(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)$$

$$p_i = p(y_i = 1 \,|\, \mathbf{x}_i) = \sigma(h(\mathbf{x}_i))$$

- **Regression:** mean squared error loss function

$$\mathscr{L}(\mathbf{w}, \mathbf{U}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - h(\mathbf{x}_i))^2$$

- **Minimize loss with respect to weights $\mathbf{w}, \mathbf{U}$ with gradient descent**

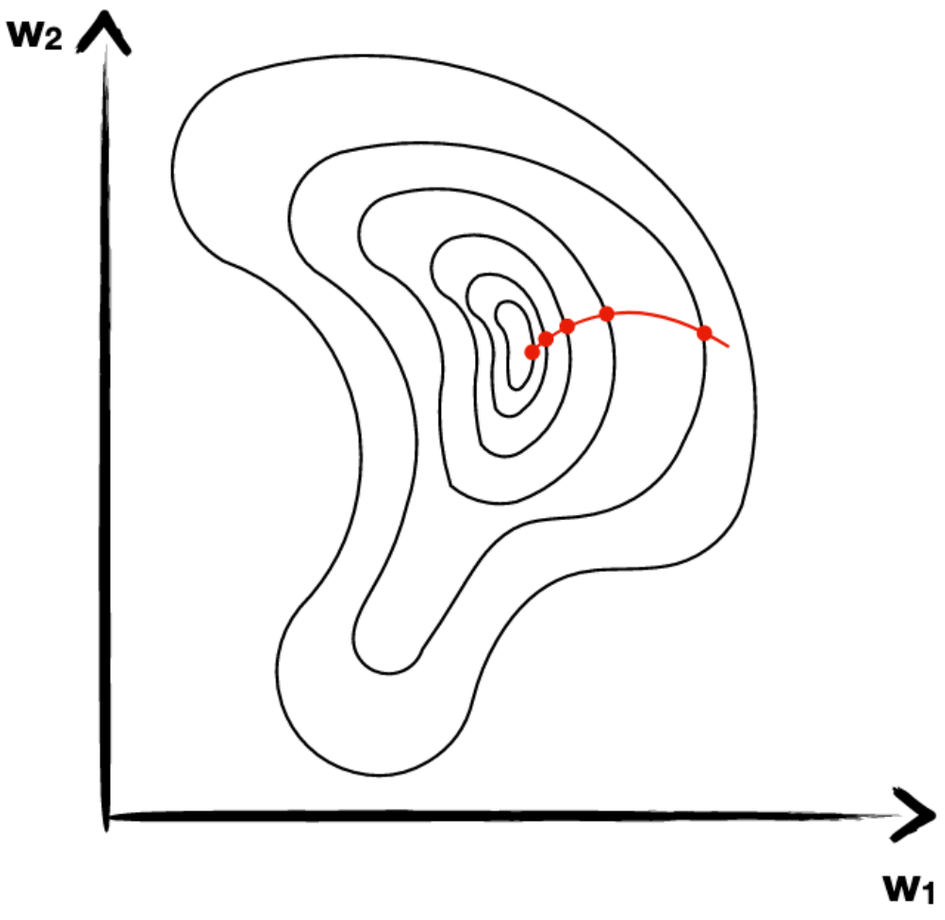
Gradient Descent

- Parameters update:

$$\mathbf{w}' \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}, \mathbf{U})$$

$$\mathbf{U}' \leftarrow \mathbf{U} - \eta \nabla_{\mathbf{U}} \mathscr{L}(\mathbf{w}, \mathbf{U})$$

- **How to compute the gradients?**



Chain rule

- The loss function: $\mathscr{L}(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(\sigma(h(\mathbf{x}_i))) + (1 - y_i)\ln(1 - \sigma(h(\mathbf{x}_i)))$

- Derivative of sigmoid: $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$

- Apply chain rule to compute gradient w.r.t $\mathbf{w}$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \frac{\partial \mathscr{L}}{\partial h}\frac{\partial h}{\partial \mathbf{w}} = \sum_i y_i(1 - \sigma(h(\mathbf{x}_i)))\sigma(\mathbf{U}\mathbf{x}_i) + (1 - y_i)\sigma(h(\mathbf{x}_i))\sigma(\mathbf{U}\mathbf{x}_i)$$

- Apply chain rule to compute gradient w.r.t. $\mathbf{u}_j$

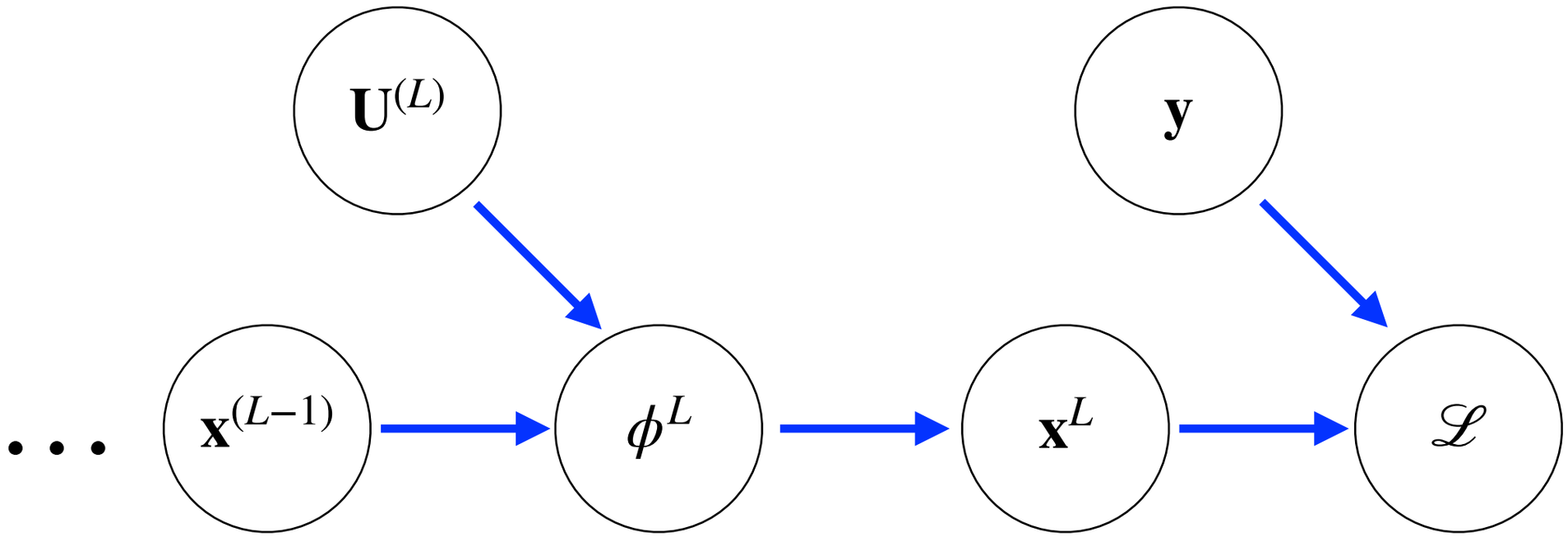
$$\frac{\partial \mathscr{L}}{\partial \mathbf{u}_j} = \frac{\partial \mathscr{L}}{\partial h}\frac{\partial h}{\partial \sigma}\frac{\partial \sigma}{\partial \mathbf{u}_j} = \sum_i y_i(1 - \sigma(h(\mathbf{x}_i)))w_j\sigma(\mathbf{u}_j\mathbf{x}_i)(1 - \sigma(\mathbf{u}_j\mathbf{x}_i))\mathbf{x}_i$$
$$+(1 - y_i)\sigma(h(\mathbf{x}_i))w_j\sigma(\mathbf{u}_j\mathbf{x}_i)((1 - \sigma(\mathbf{u}_j\mathbf{x}_i))\mathbf{x}_i$$

Backpropagation

- Loss function composed of ***L*** layers of non linearity $\mathscr{L}(\phi^{(L)}(\ldots\phi^{(1)}(\mathbf{x})))$

- **Forward step (forward propagation)**

  - compute and save intermediate computations $\phi^{(1)}(\mathbf{x}), \ldots, \phi^{(L)}(\mathbf{x})$



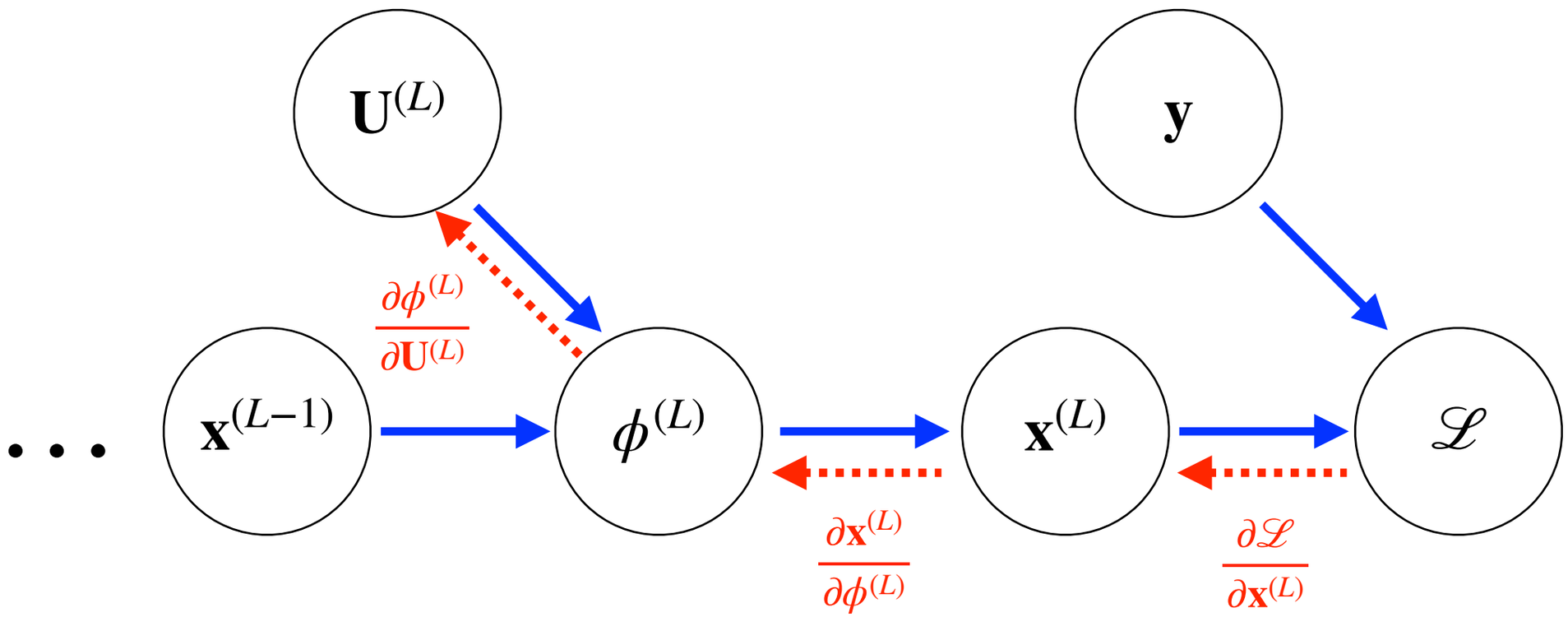
Backpropagation

- Loss function composed of ***L*** layers of non linearity $\mathscr{L}(\phi^{(L)}(\ldots\phi^{(1)}(\mathbf{x})))$

- **Forward step (forward propagation)**

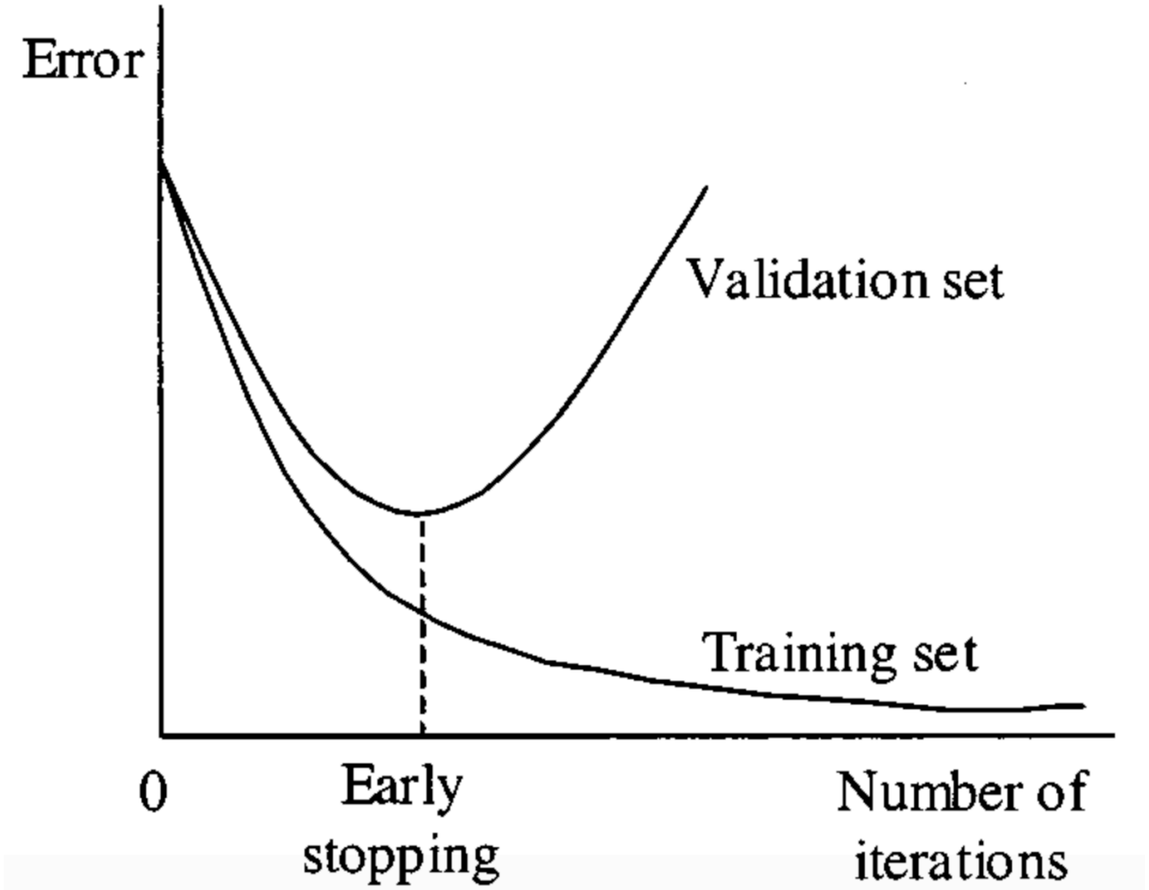
  - compute and save intermediate computations $\phi^{(1)}(\mathbf{x}), \ldots, \phi^{(L)}(\mathbf{x})$

- **Backward step (backpropagation)** → $$\frac{\partial \mathscr{L}}{\partial \mathbf{U}^{(L)}} = \frac{\partial \mathscr{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \phi^{(L)}} \frac{\partial \phi^{(L)}}{\partial \mathbf{U}^{(L)}}$$



Training

- The gradient is typically computed through Automatic Differentiation (fast and exact)
- Repeat gradient update of weights to reduce loss
  - each iteration through the full dataset (in mini batches) is called an epoch
- Use validation set to examine for overtraining, and determine when to stop training

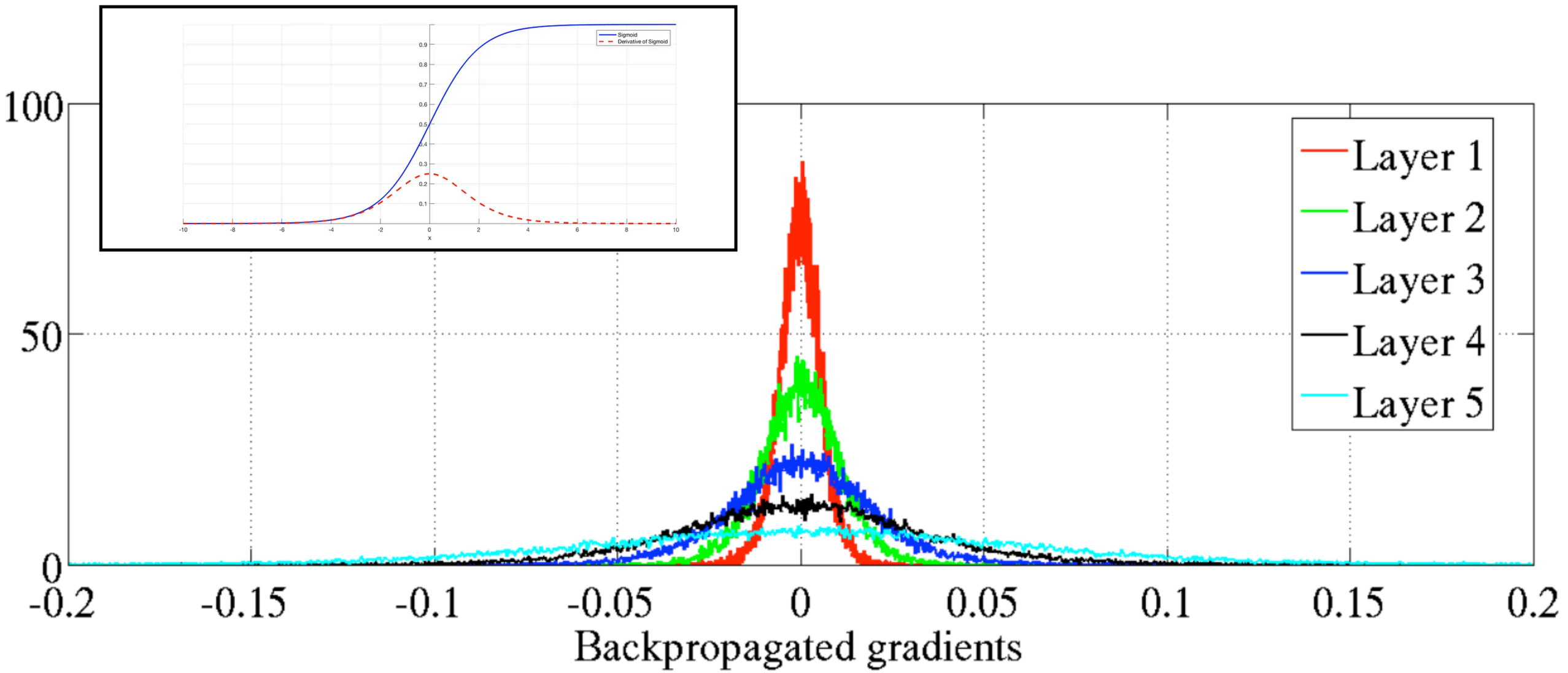


Vanishing gradient

- As more layers using certain activation functions are added to neural networks, the gradients of the loss function might approach zero, making the network hard to train

  - small gradients slow down, and eventually block, stochastic gradient descent

  - this results in a limited capacity of learning

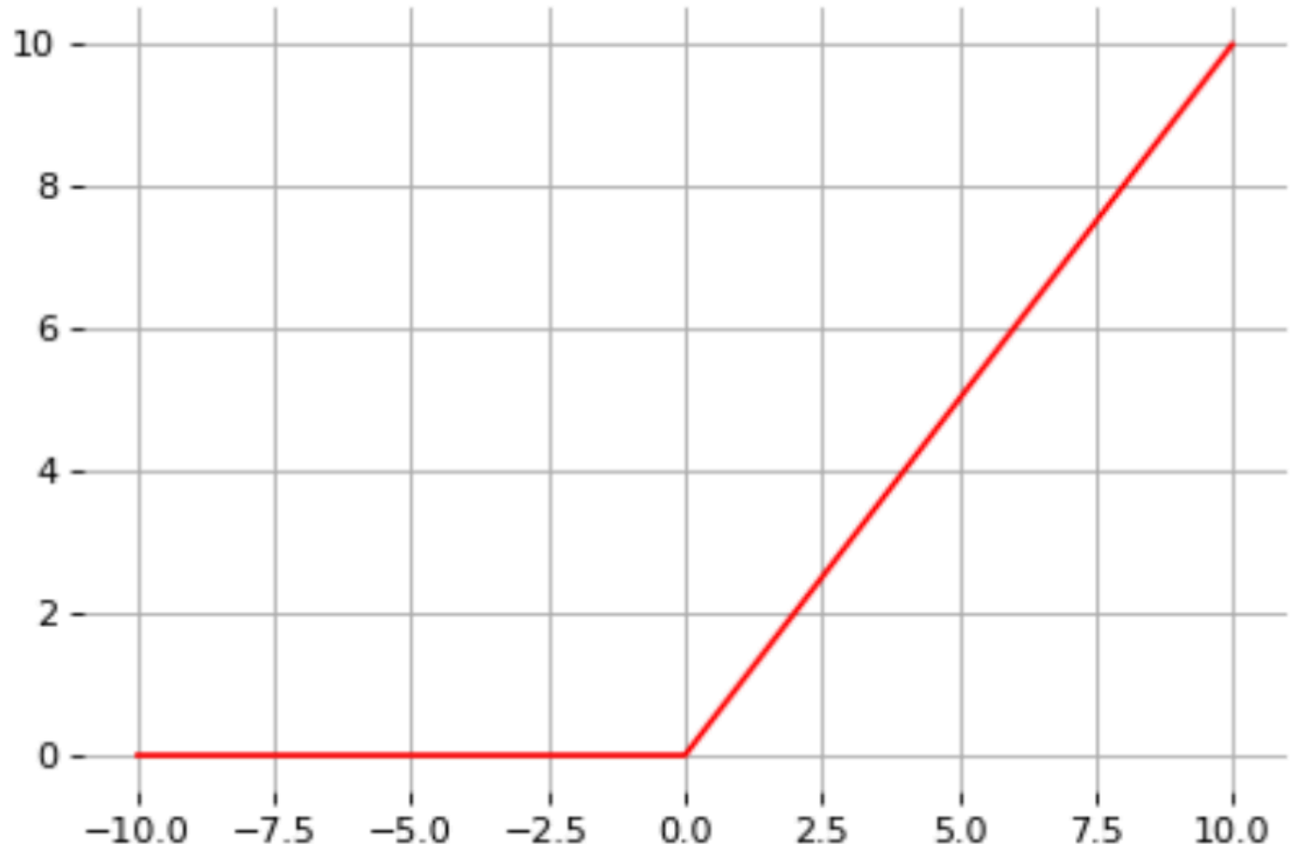


Activation functions

Activation functions

- Instead of the sigmoid (activation) function , modern NN are for most based on **rectified linear units (ReLU)**

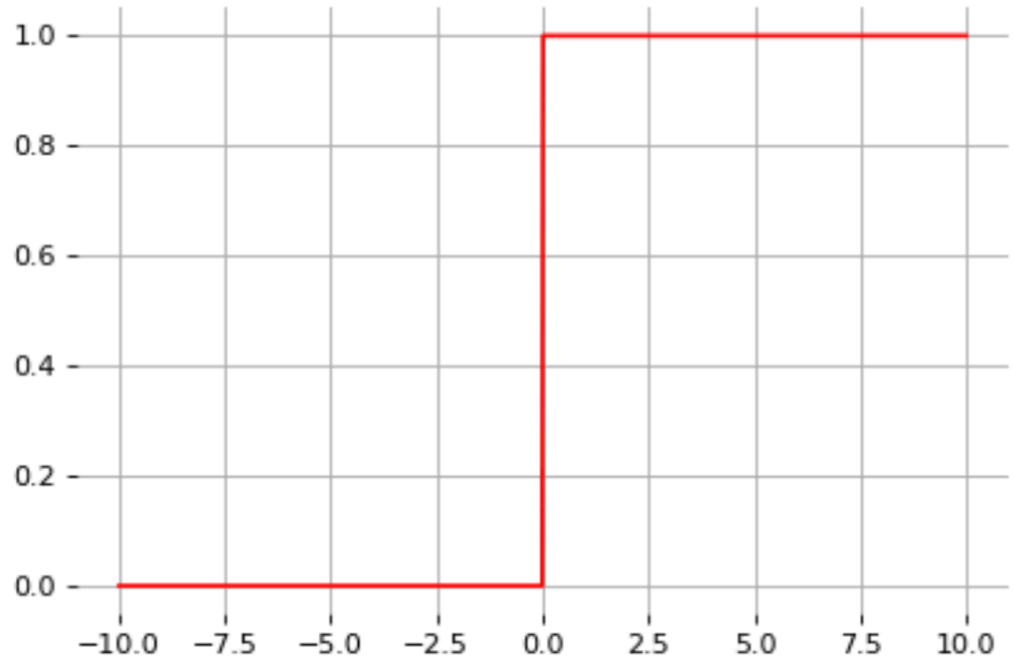
$$\mathrm{ReLU}(x) = \max(0, x)$$



Activation functions

- Note that the derivative of the ReLU function is

$$\frac{\partial}{\partial x}\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$



- For $\mathbf{x = 0}$ the derivative is undefined. In practice, it is set to zero.

Activation functions

- With ReLU you end up with products of ones or zeros rather than products of very small numbers when doing backpropagation

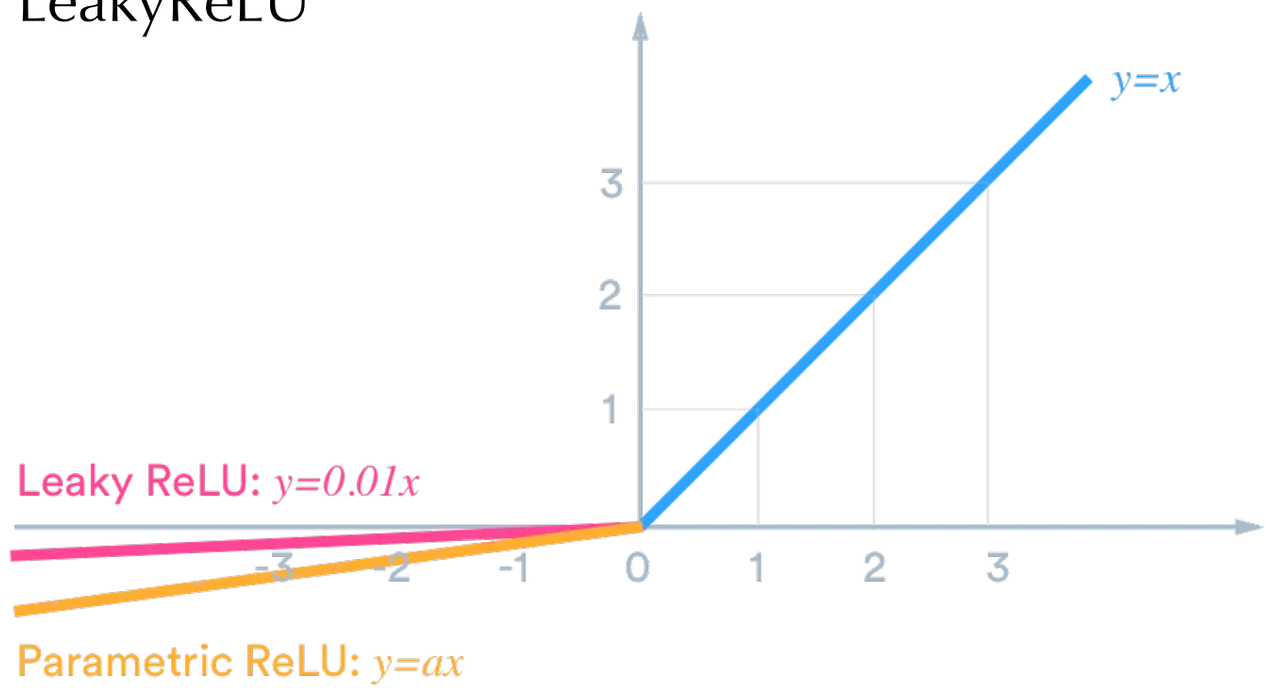
  - the ReLU unit dies when its input is negative

  - this is actually a useful property to induce *sparsity*

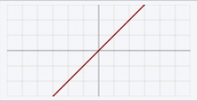
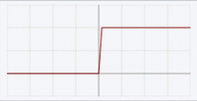
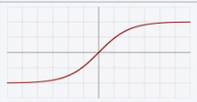
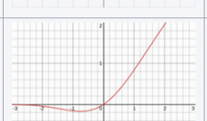
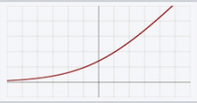
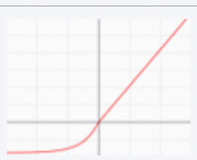
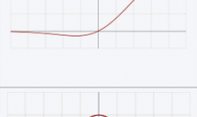
- Drawback: a "dead" neuron is unlikely to recover and if there are too many of these you may end up with a large part of the NN doing nothing

  - in practice this happens if the learning rate is off or if there is a large negative bias

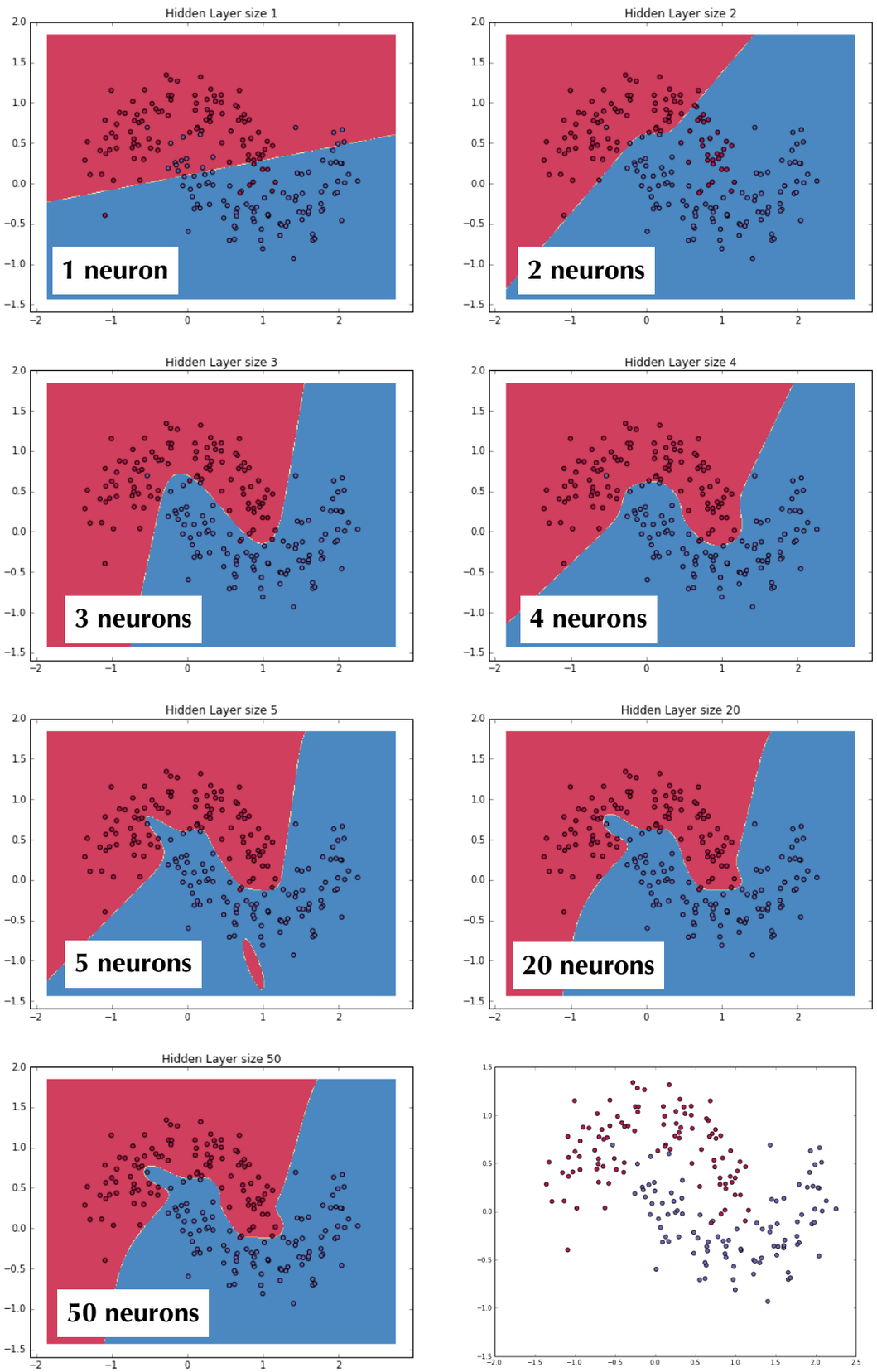
- Several ways to mitigate this, e.g. LeakyReLU

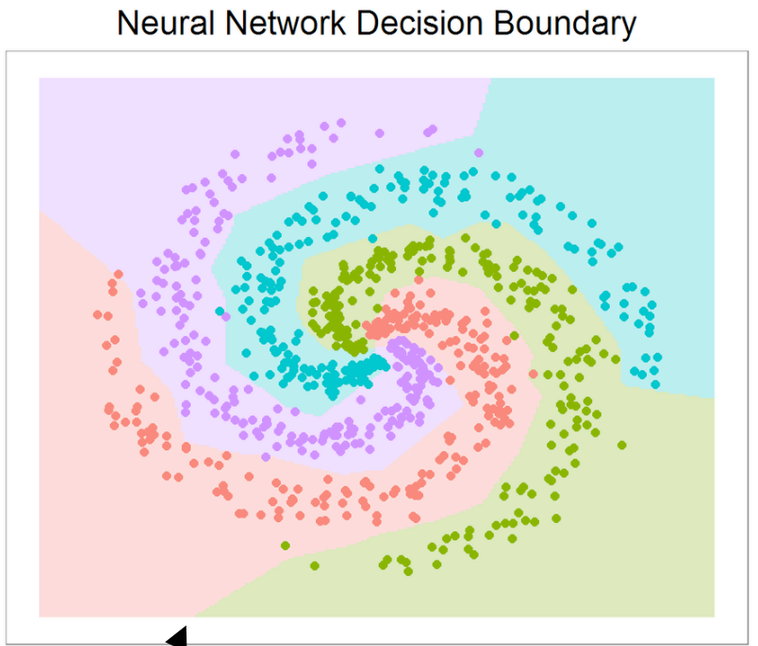


Activation functions

Name	Plot	Function, $g(x)$	Derivative of g, $g'(x)$	Range	Order of continuity
Identity		x	1	$(-\infty, \infty)$	C^∞
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	0	$\{0, 1\}$	C^{-1}
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1 + e^{-x}}$	$g(x)(1 - g(x))$	$(0, 1)$	C^∞
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - g(x)^2$	$(-1, 1)$	C^∞
Rectified linear unit (ReLU)[8]		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = \max(0, x) = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$	C^0
Gaussian Error Linear Unit (GELU)[5]		$\frac{1}{2}x\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = x\Phi(x)$	$\Phi(x) + x\phi(x)$	$(-0.17\ldots, \infty)$	C^∞
Softplus[9]		$\ln(1 + e^x)$	$\frac{1}{1 + e^{-x}}$	$(0, \infty)$	C^∞
Exponential linear unit (ELU)[10]		$\begin{cases} \alpha\left(e^x - 1\right) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ with parameter α	$\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$(-\alpha, \infty)$	$\begin{cases} C^1 & \text{if } \alpha = 1 \\ C^0 & \text{otherwise} \end{cases}$
Scaled exponential linear unit (SELU)[11]		$\lambda\begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameters $\lambda = 1.0507$ and $\alpha = 1.67326$	$\lambda\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\lambda\alpha, \infty)$	C^0
Leaky rectified linear unit (Leaky ReLU)[12]		$\begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$(-\infty, \infty)$	C^0
Parametric rectified linear unit (PReLU)[13]		$\begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameter α	$\begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Sigmoid linear unit (SiLU,[5] Sigmoid shrinkage,[14] SiL,[15] or Swish-1[16])		$\frac{x}{1 + e^{-x}}$	$\frac{1 + e^{-x} + xe^{-x}}{\left(1 + e^{-x}\right)^2}$	$[-0.278\ldots, \infty)$	C^∞
Gaussian		e^{-x^2}	$-2xe^{-x^2}$	$(0, 1]$	C^∞

Non-linear decision boundaries





[Source]

4-class classification
2-hidden layer NN
ReLU activations
L2 regularization

Binary classification
1-hidden layer NN
Varying number of nodes
L2 regularization

[Source]

Universal approximation theorem

- Feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate continuous functions arbitrarily well on a space $\mathbb{R}^n$

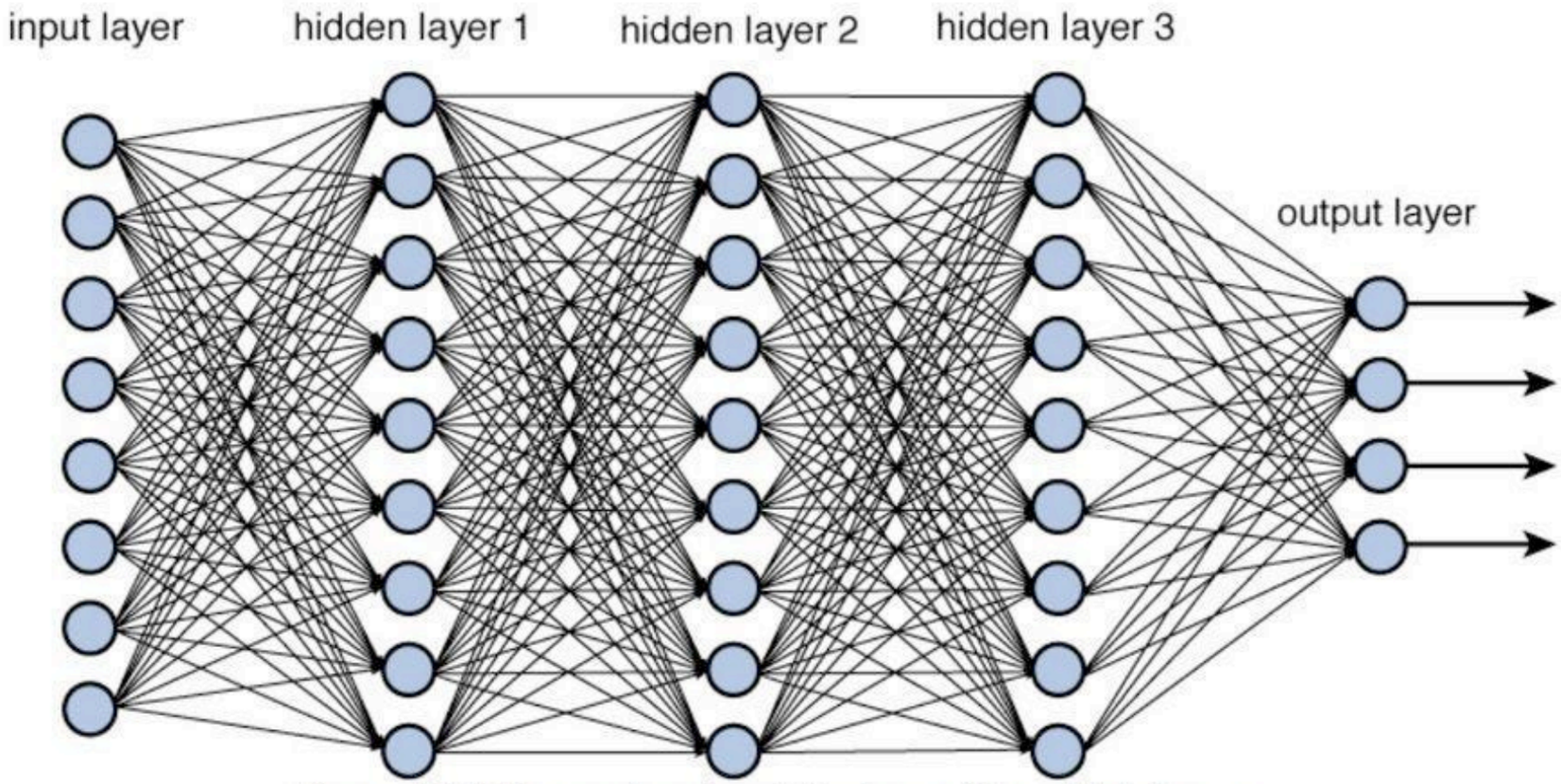
  - only mild assumptions on non-linear activation function needed. Sigmoid functions work, as do others.

- **But no other information are added on how many neurons needed, or how much data!**

- How to find the parameters, given a dataset, to perform this approximation?

Deep Neural Networks

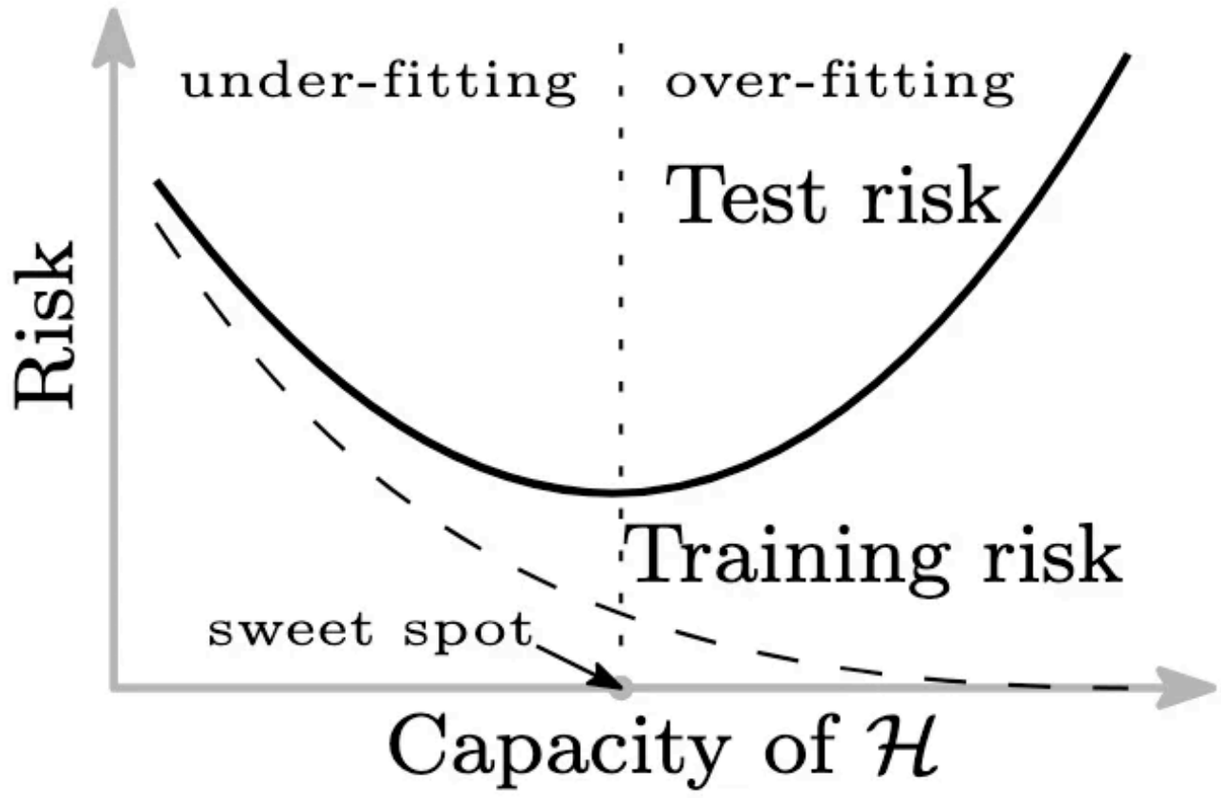
Deep Neural Networks



- As data complexity grows, need exponentially large number of neurons in a single-hidden-layer network to capture all structure in data
- Deep neural networks factorize the learning of structure in data across many layers
- Difficult to train, only recently possible with large datasets, fast computing (GPU/TPU), and new training procedures / NN architectures

Double Descent

- What we learned yesterday: **"complex models are worse"** — the bias-variance trade off



Double Descent

- What we learned yesterday: **“complex models are worse”** — the bias-variance trade off

Not entirely true…

- GPT-3: 175 billion parameters
  - Cost (2020): $4.6 million
- GPT-4 (Human Brain): 100 trillion parameters
  - Cost (2020): $2.6 billion
  - Cost (2024): $325 million
  - Cost (2028): $40 million
  - Cost (2032): $5 million

Double Descent

- What we learned yesterday: **"complex models are worse"** — the bias-variance trade off
- For modern DNN architectures this is not entirely true → so-called **double descent** behaviour observed by Belkin et al. in 2018
- **Double Descent:** over-parametrizing a deep model often improves test performance, contrary to bias-variance tradeoff prediction
- **But we must control that gradient does not vanish or that sparsity is under control**
- Major part of deep learning is to **choose the right function and engineer the data properly** for the architecture
  - instead of trying to improve training with regularization and/or new optimizers
  - need to make gradient descent work, even at the cost of a substantially engineering the model

Neural Network Zoo

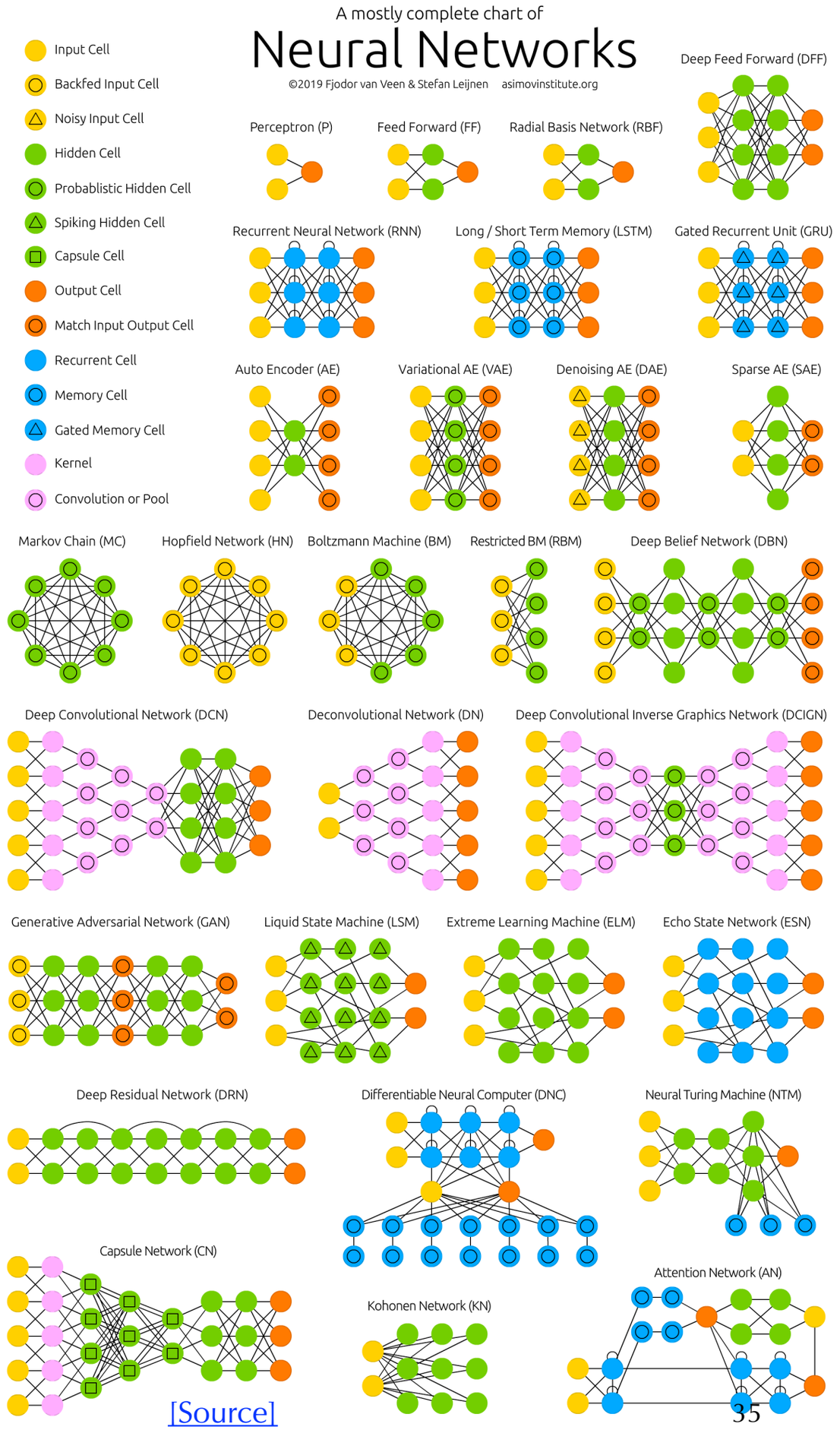
- The NN architecture and the node connectivity can be adapted for problem at hand

- Moving inductive bias from feature engineering to model design

  - *Inductive bias:*
    Knowledge about the problem

  - *Feature engineering:*
    Hand crafter variables

  - *Model design:*
    The data representation and the NN architecture

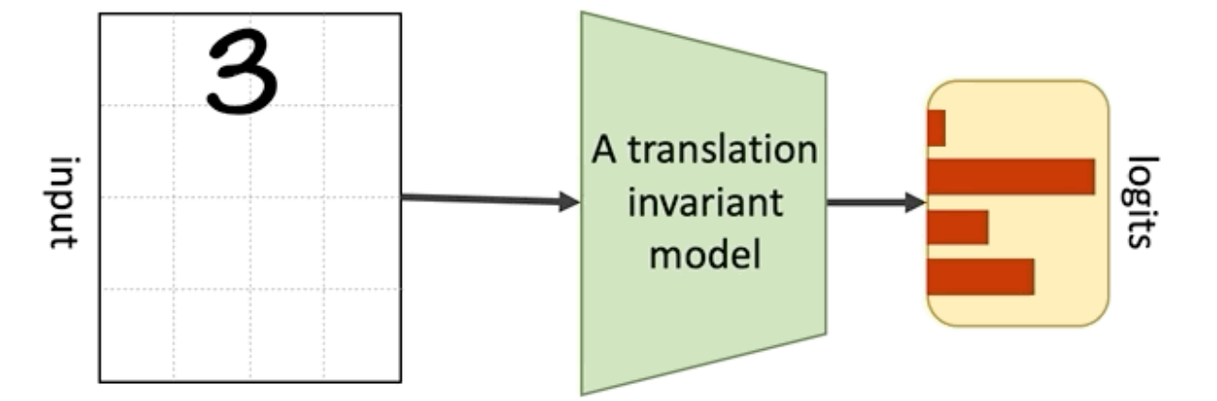


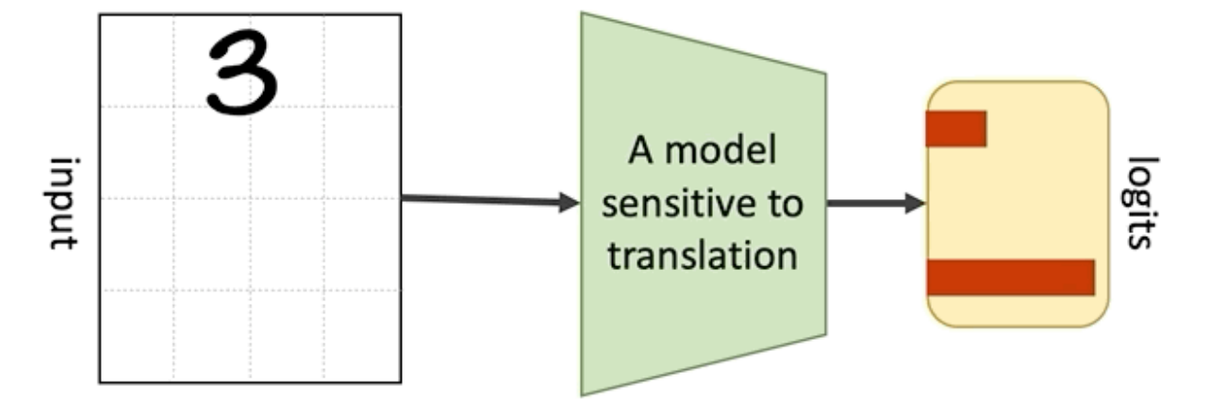
[Source]

Convolutional Neural Networks

Convolutional Neural Networks

- When the structure of data includes **“equivariance to translation”**, a representation meaningful at a certain location can / should be used everywhere

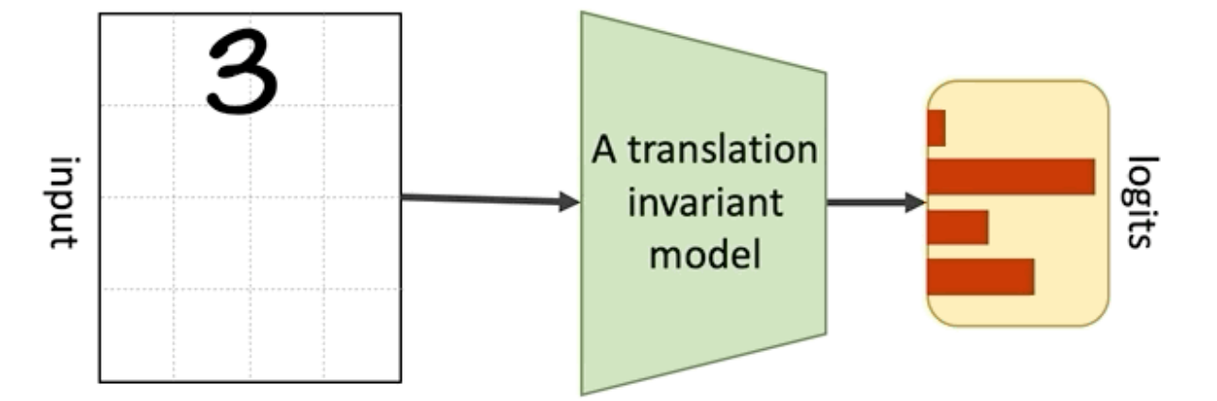


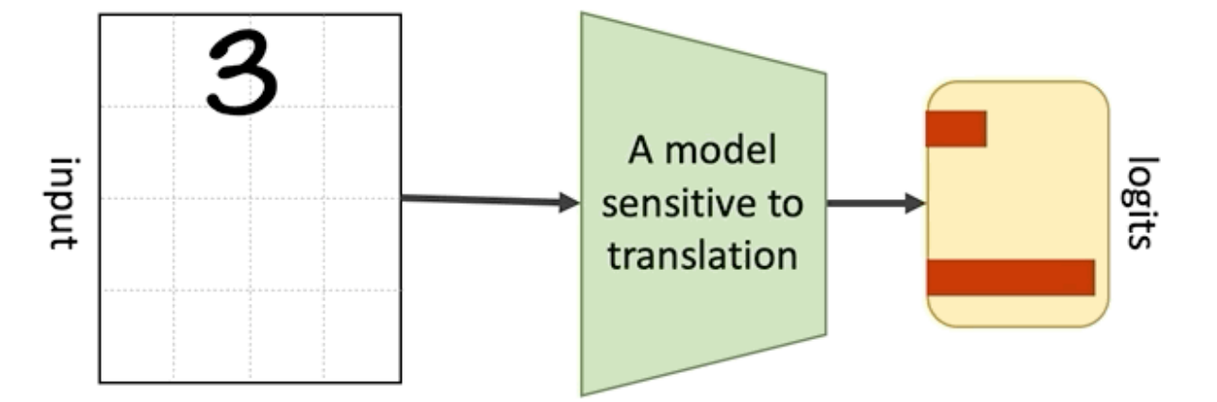


- Convolutional layers build on this idea, that the same “local” transformation is applied everywhere and preserves the final output

Convolutional Neural Networks

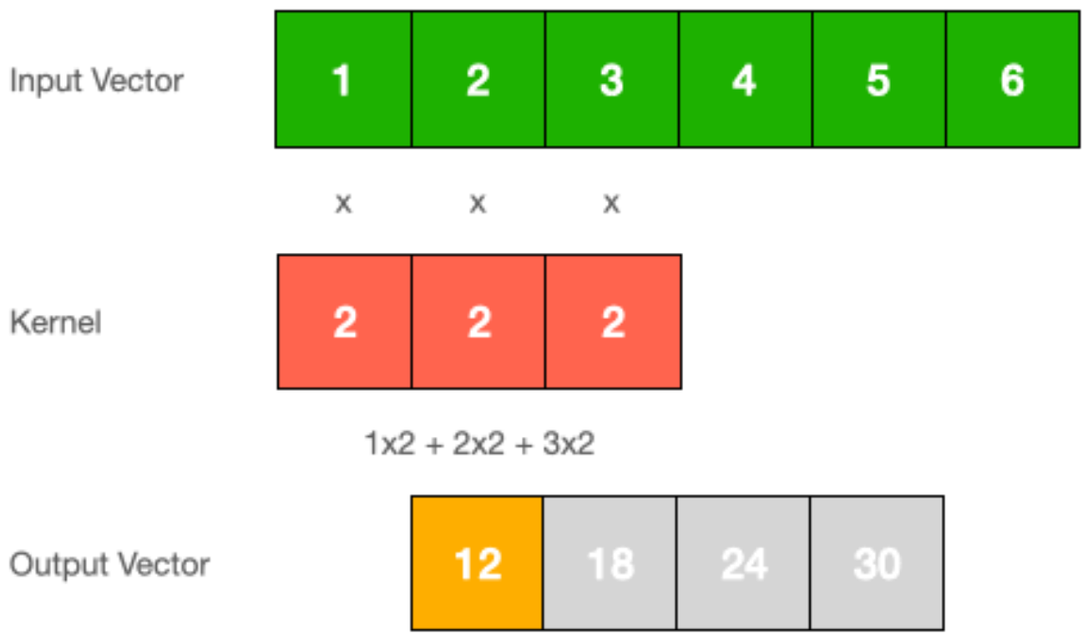
- When the structure of data includes **“equivariance to translation”**, a representation meaningful at a certain location can / should be used everywhere





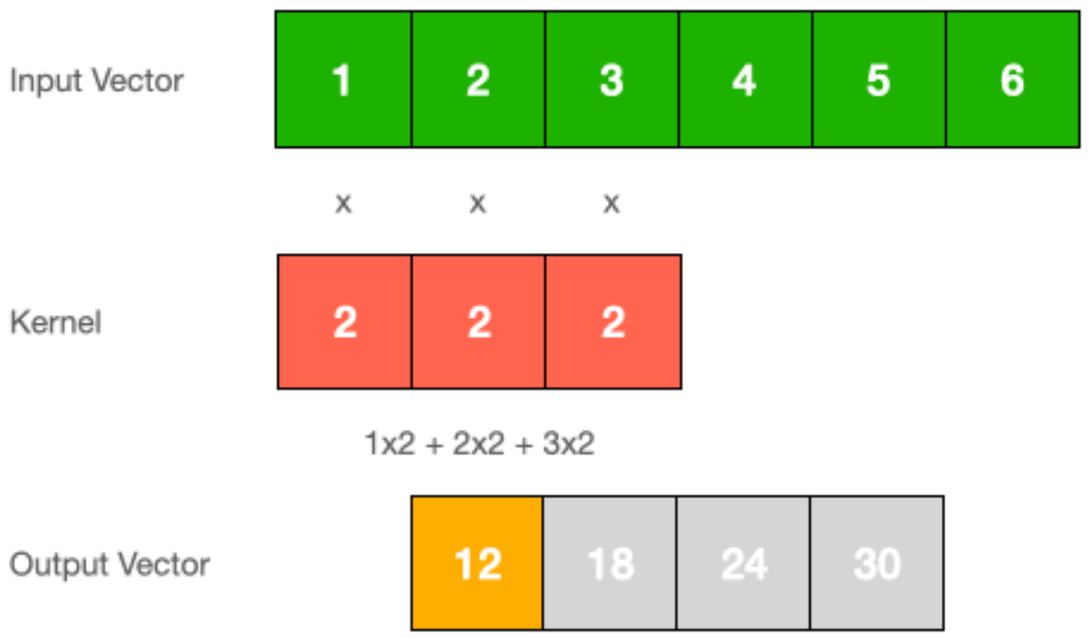
- Convolutional layers build on this idea, that the same “local” transformation is applied everywhere and preserves the final output

1D convolutional layers



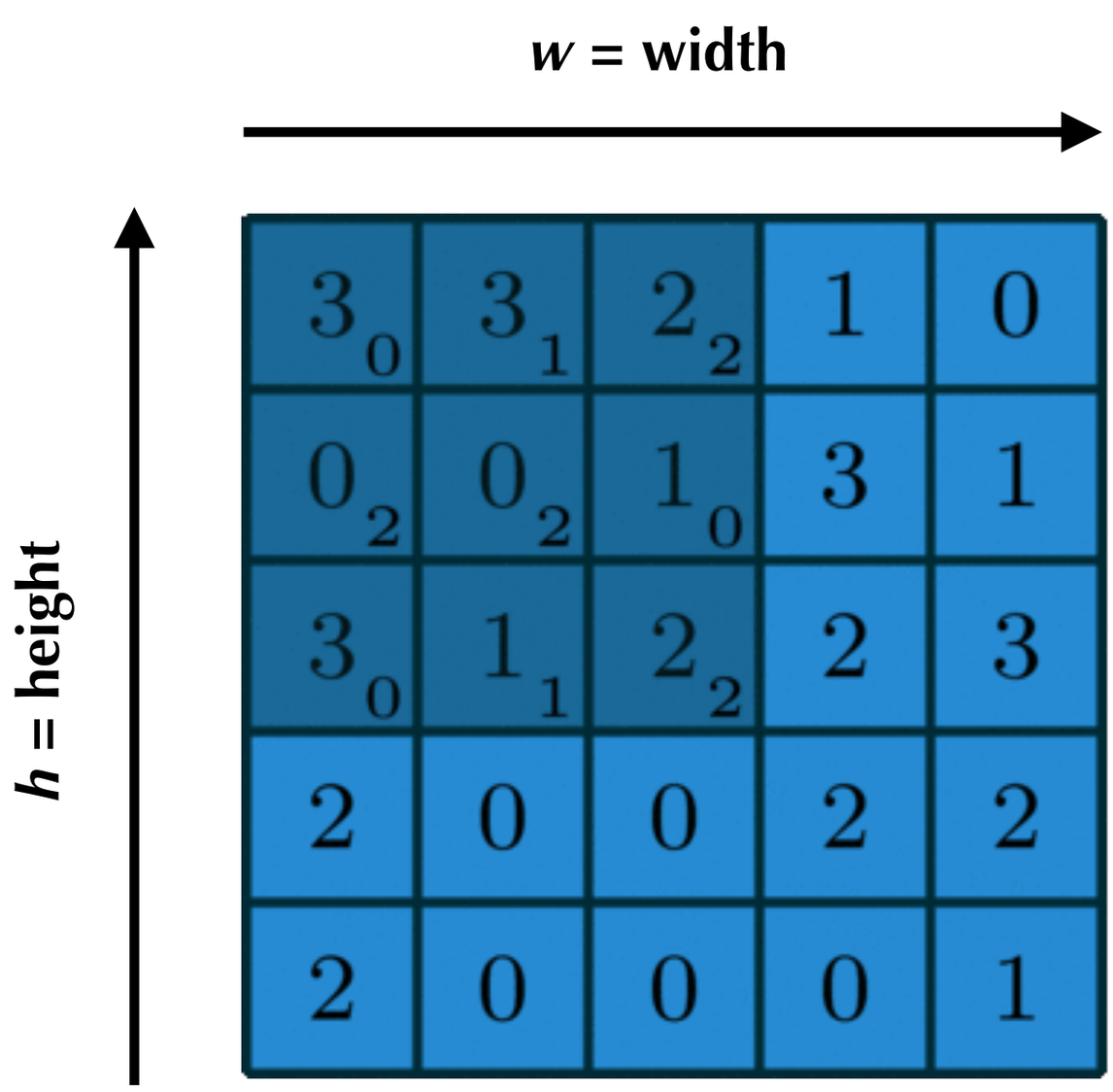
- **Data** (one vector of m features for simplicity): $\mathbf{x} \in \mathbb{R}^m$
- **Convolutional filter of width k:** $\mathbf{u} \in \mathbb{R}^k$
- Convolution $\mathbf{x} \circledast \mathbf{u}$ is a vector of size $m - k + 1$ with element j: $(\mathbf{x} \circledast \mathbf{u})_j = \sum_{b=0}^{k-1} x_{i+b} u_b$
- Scan vector $\mathbf{x}$ by iterating over i to get each output element j

1D convolutional layers



- **Data** (one vector of m features for simplicity): $\mathbf{x} \in \mathbb{R}^m$
- **Convolutional filter of width k:** $\mathbf{u} \in \mathbb{R}^k$
- Convolution $\mathbf{x} \circledast \mathbf{u}$ is a vector of size $m - k + 1$ with element j: $(\mathbf{x} \circledast \mathbf{u})_j = \sum_{b=0}^{k-1} x_{i+b} u_b$
- Scan vector $\mathbf{x}$ by iterating over i to get each output element j

2D convolutional layers



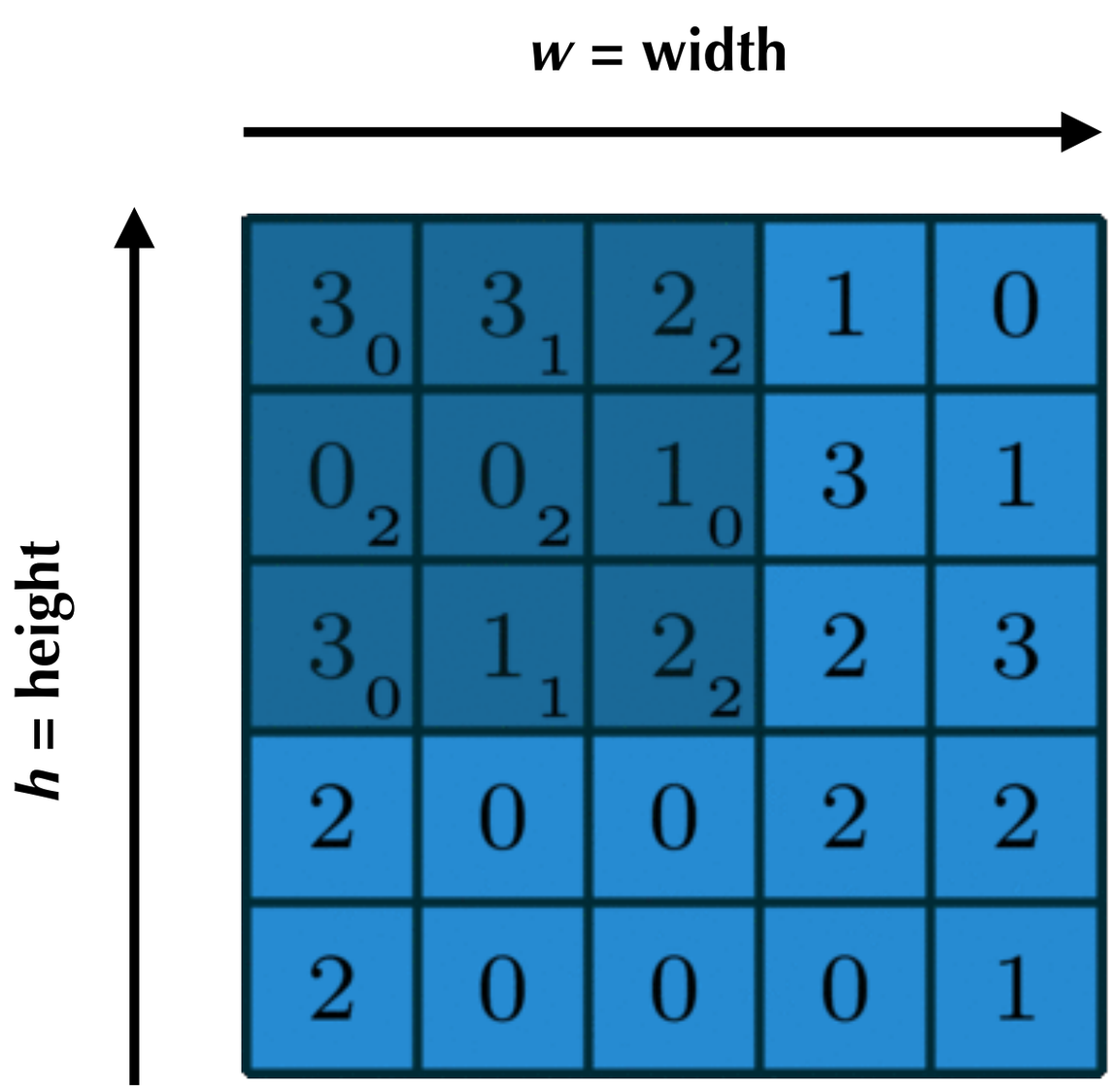
Input image size:
***H x W* = 5 x 5**

Filter size:
height = h_k = 3
width = w_k = 3

Output image size:
height = *h* - h_k + *1* = *3*
width = *w* - w_k + *1* = *3*

Same principle but over a 2D matrix

2D convolutional layers



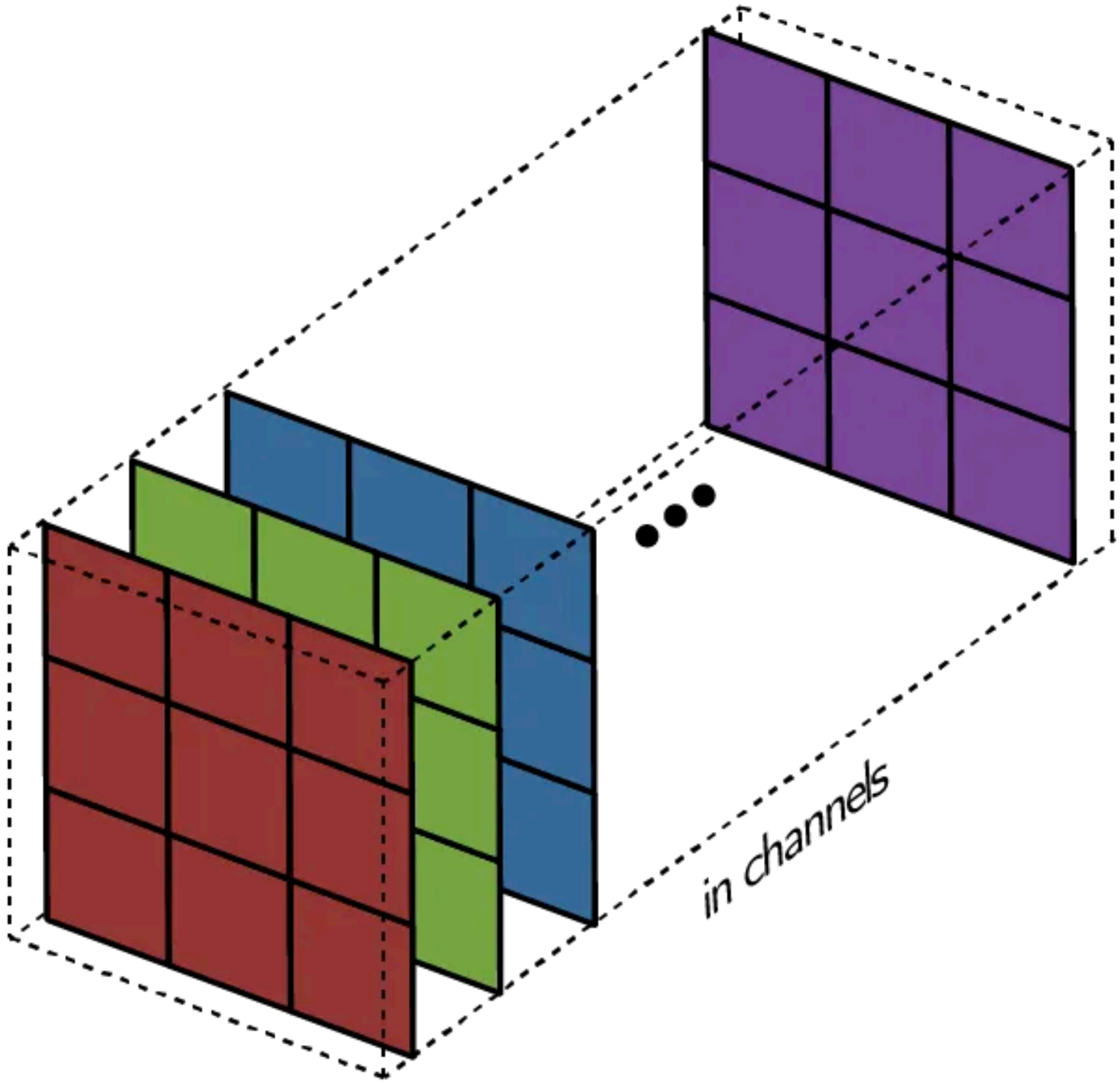
Input image size:
***H x W* = 5 x 5**

Filter size:
height = h_k = 3
width = w_k = 3

Output image size:
height = *h* - h_k + *1* = *3*
width = *w* - w_k + *1* = *3*

Same principle but over a 2D matrix

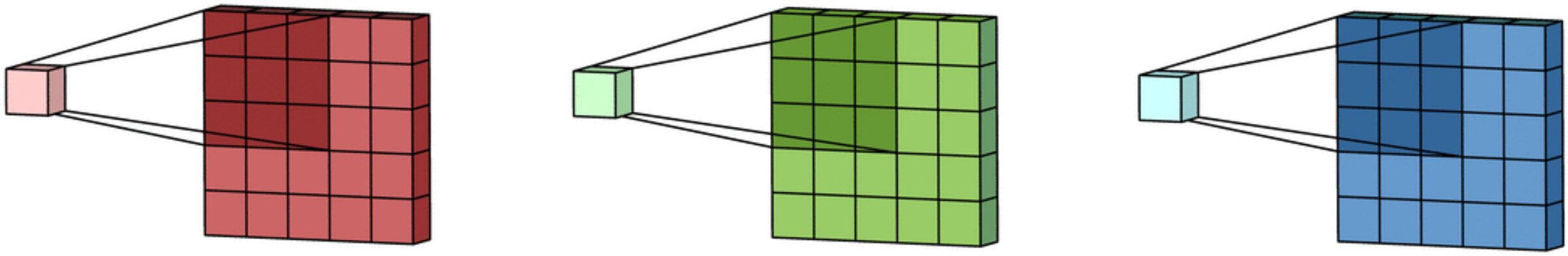
2D CNNs with multiple channels



Many 2D images stacked on top of each others — the third dimension is called “channel”

2D CNNs with multiple channels

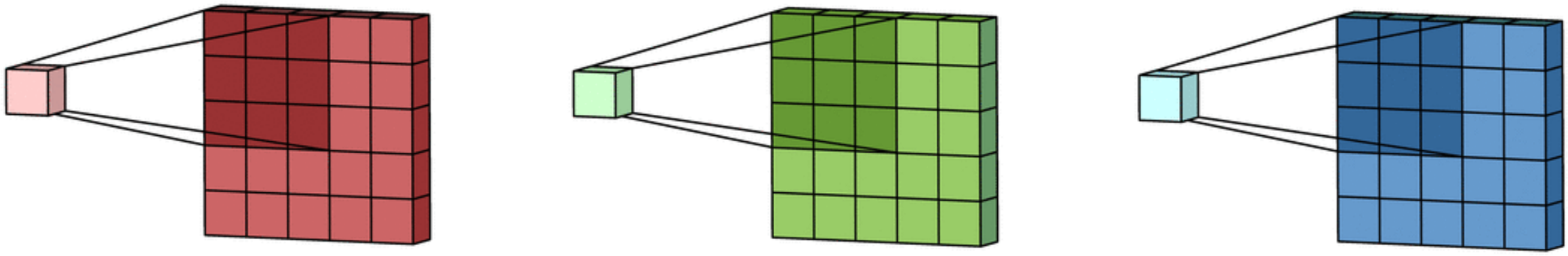
(nb, one filter gets decomposed in three parts)



- The filter has the same depth (or number of channels C) as the input image
- It is decomposed in C sub-filters of same size each acting independently on each of the C channels → **output: three new 2D images (one per channel)**
  - i.e., weights are not shared across channels

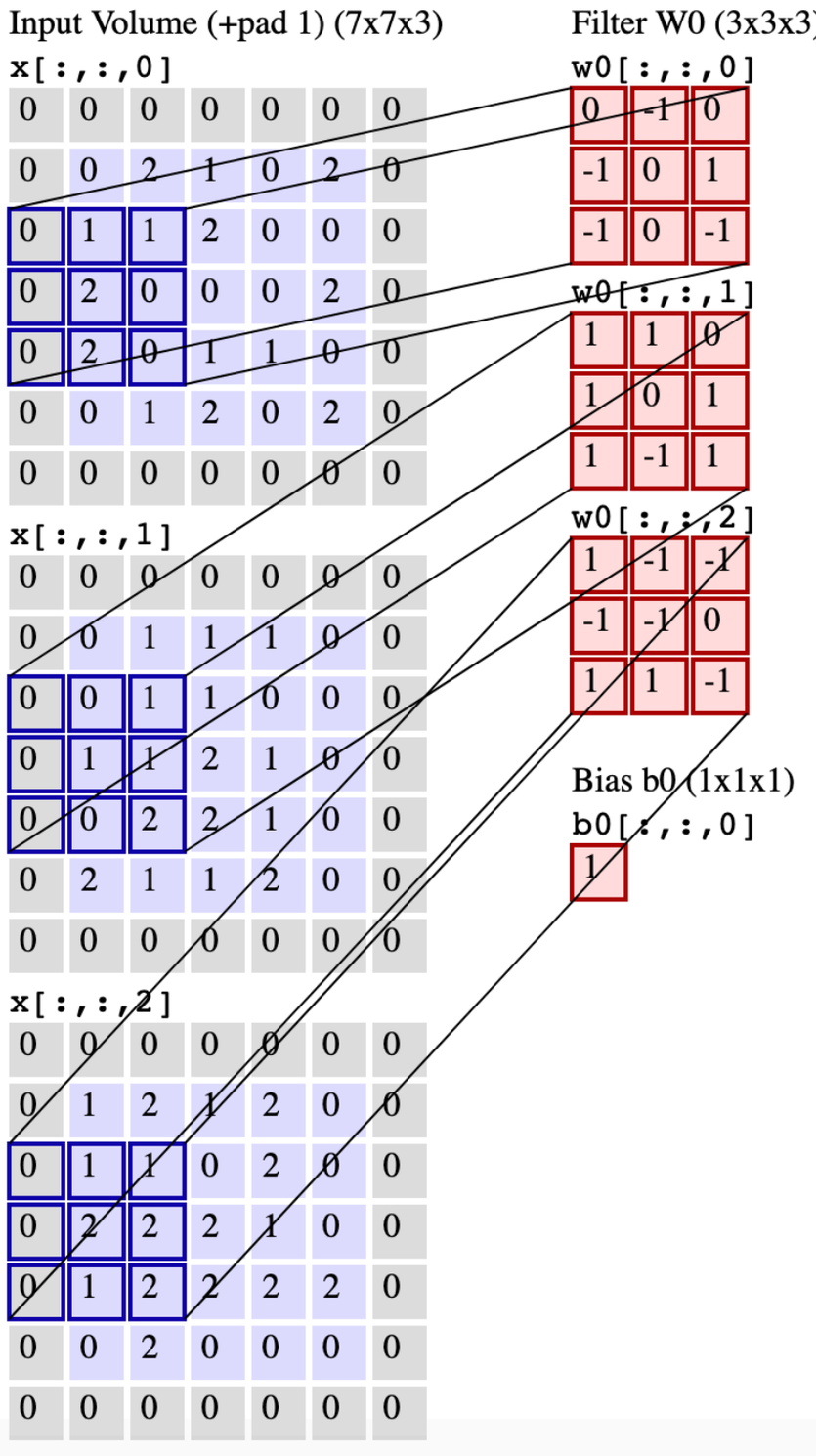
2D CNNs with multiple channels

(nb, one filter gets decomposed in three parts)



- The filter has the same depth (or number of channels C) as the input image
- It is decomposed in C sub-filters of same size each acting independently on each of the C channels → **output: three new 2D images (one per channel)**
  - i.e., weights are not shared across channels

2D CNNs with multiple channels

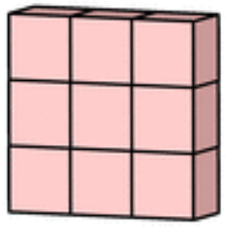


Weights are not shared among channels

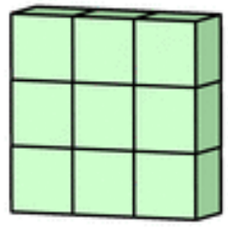
[source]

2D CNNs with multiple channels

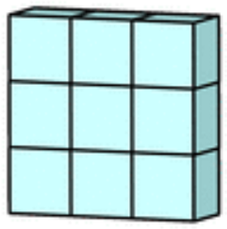
depth = 1



depth = 1



depth = 1

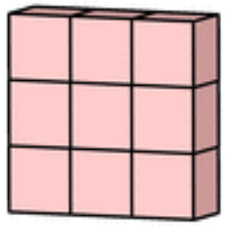


final 2D image:
depth = 1

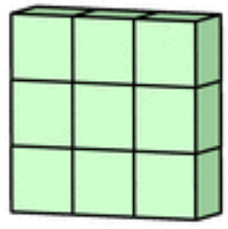
- The filter has the same depth (or number of channels C) as the input image
- It is decomposed in C sub-filters of same size each acting independently on each of the C channels → **output: three new 2D images (one per channel)**
  - i.e., weights are not shared across channels
- The new 2D images are summed pixel-by-pixel to give one new "2D" image

2D CNNs with multiple channels

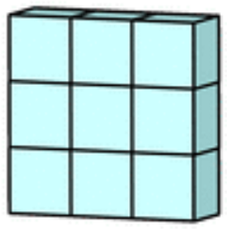
depth = 1



depth = 1



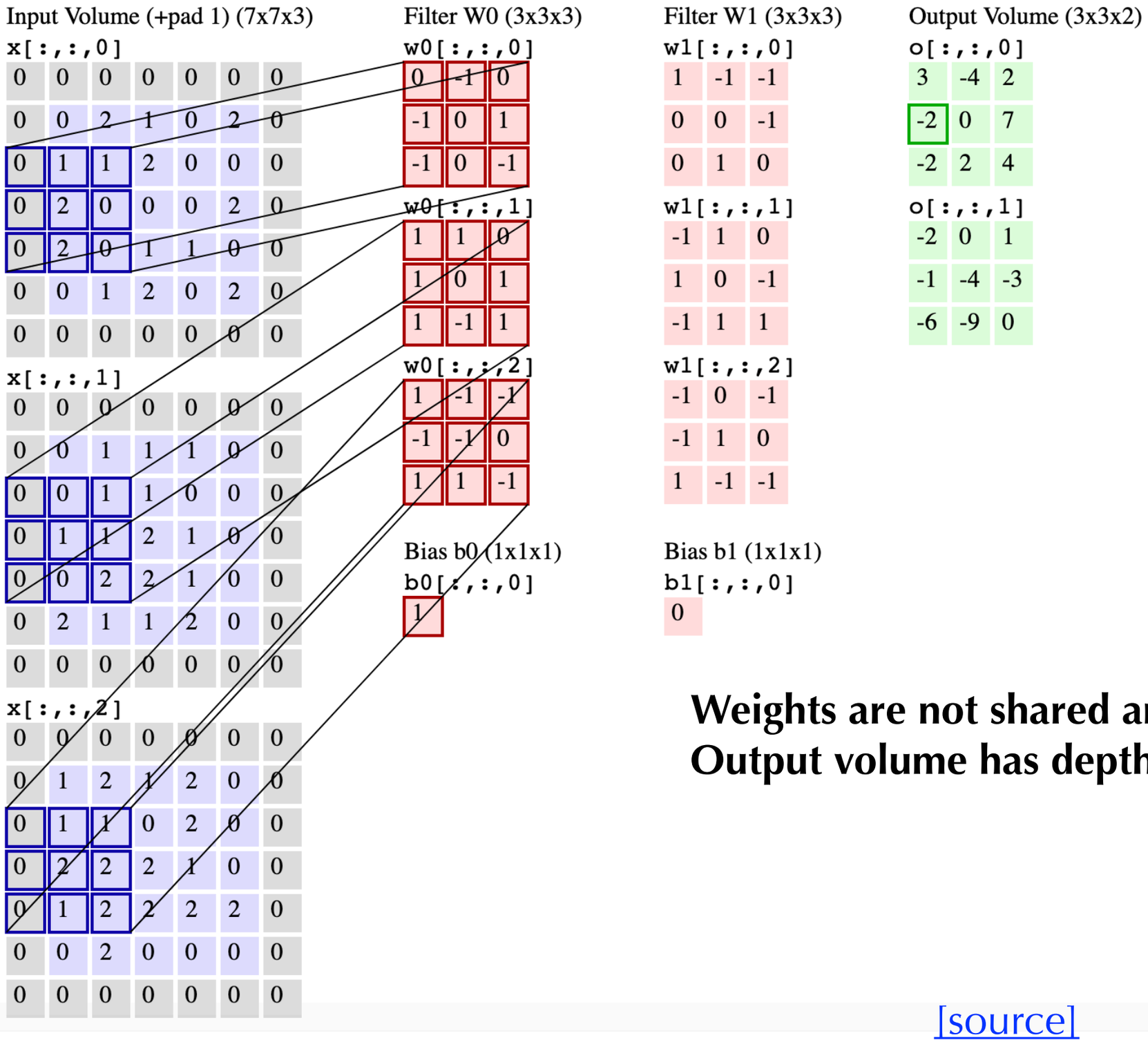
depth = 1



final 2D image:
depth = 1

- The filter has the same depth (or number of channels C) as the input image
- It is decomposed in C sub-filters of same size each acting independently on each of the C channels → **output: three new 2D images (one per channel)**
  - i.e., weights are not shared across channels
- The new 2D images are summed pixel-by-pixel to give one new "2D" image

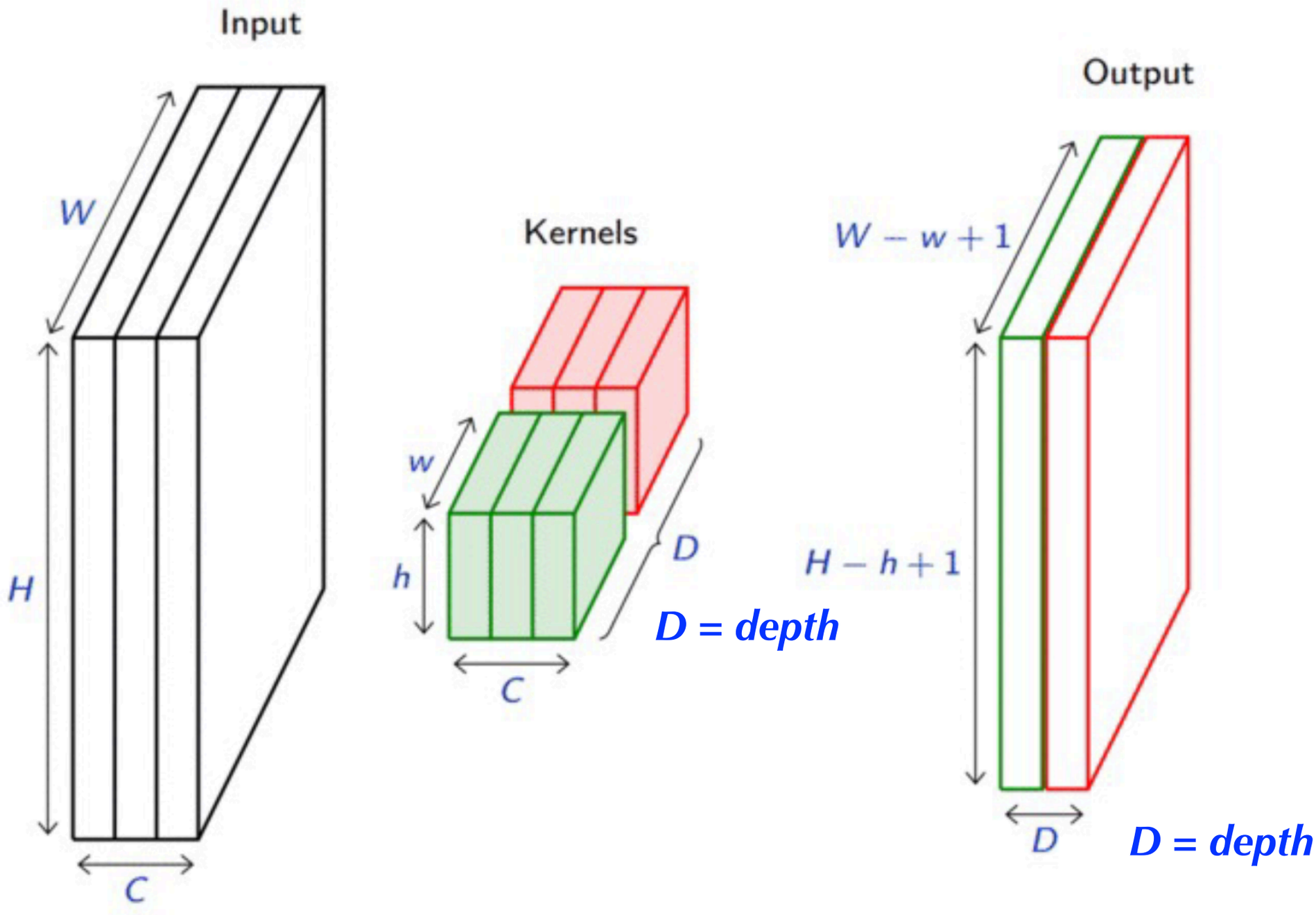
2D CNNs with multiple channels



Weights are not shared among channels
Output volume has depth = number of "3D" filters

[source]

2D CNNs with multiple channels



You can have more than one filter
→ each of them add one more dimension to the output

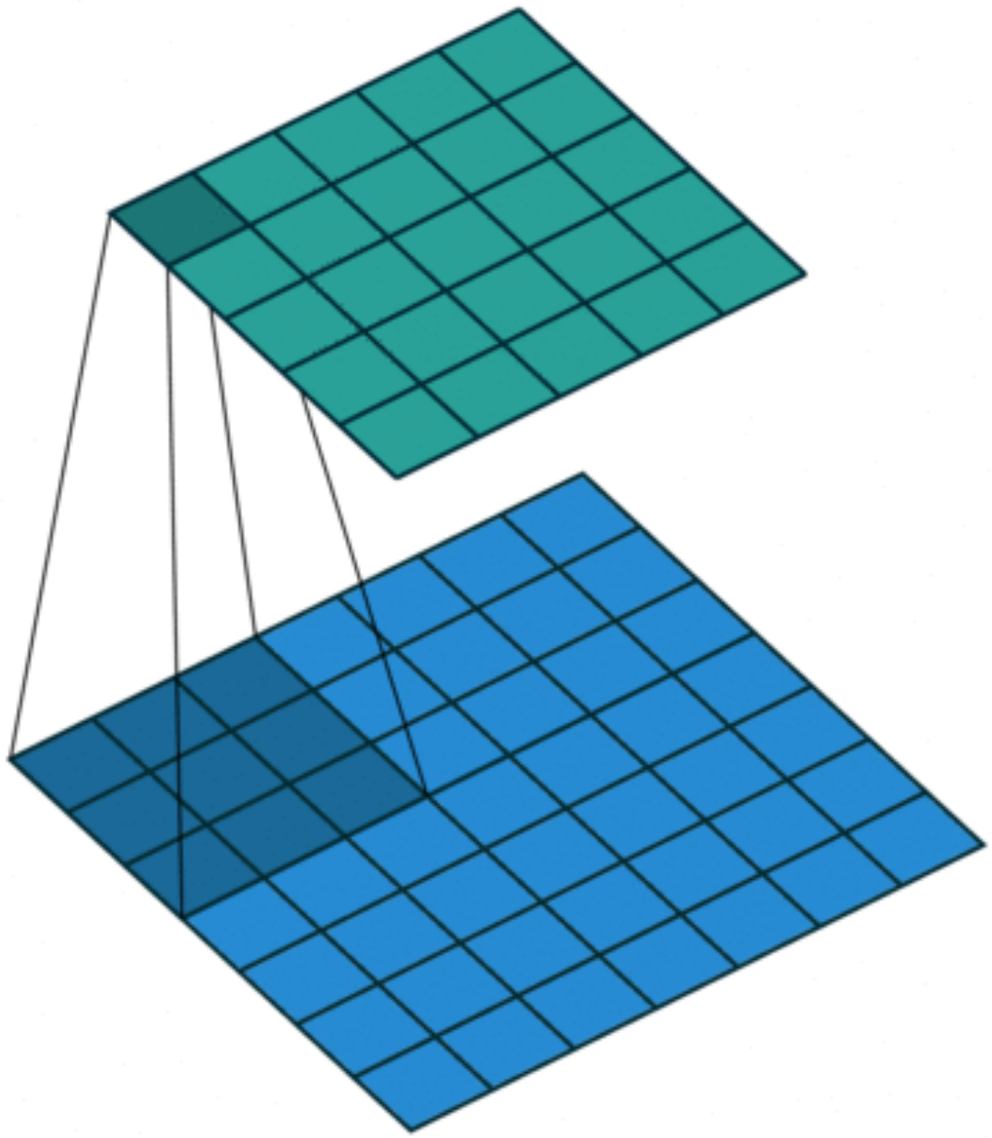
2D convolutional layers

- Input data is a tensor $\mathbf{x}$ of size $C \times h \times w$
- Learnable filter $\mathbf{u}$ of size $C \times h_k \times w_k$
  - the size $h_k \times w_k$ is called the *receptive field*
- Through the convolution operation:

$$(\mathbf{x} \circledast \mathbf{u})_k = \sum_{c=0}^{C-1} (\mathbf{x}_c \circledast \mathbf{u}_c)_k = \sum_{c=0}^{C-1} \sum_{n=0}^{h_k-1} \sum_{m=0}^{w_k-1} x_{c,n+i,m+j} u_{c,n,m}$$

- Scan vector $\mathbf{x}$ by iterating over i and j to get each output element k
- Output size is $(h - h_k + 1) \times (w - w_k + 1)$ for each filter
  - often called *activation map* or *output feature map*

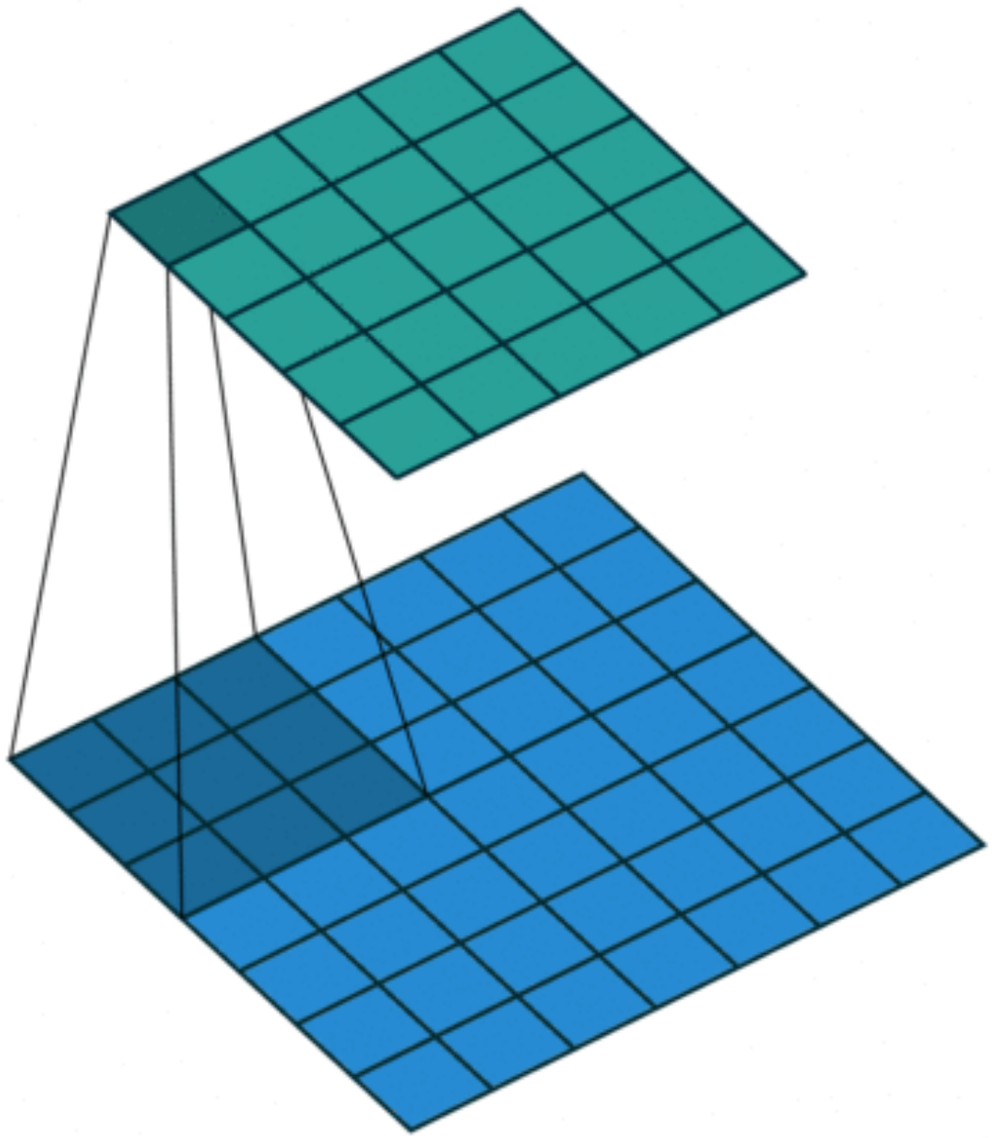
Hyperparameters: stride



Of how many pixels to move the filter
– this case has stride one but could be larger

The stride basically defines how many times you see the same pixels
Caveat: even with stride 1 the corner pixels will be seen only once

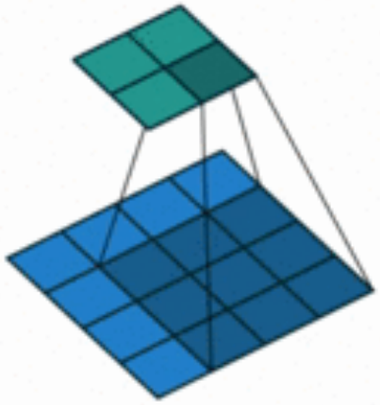
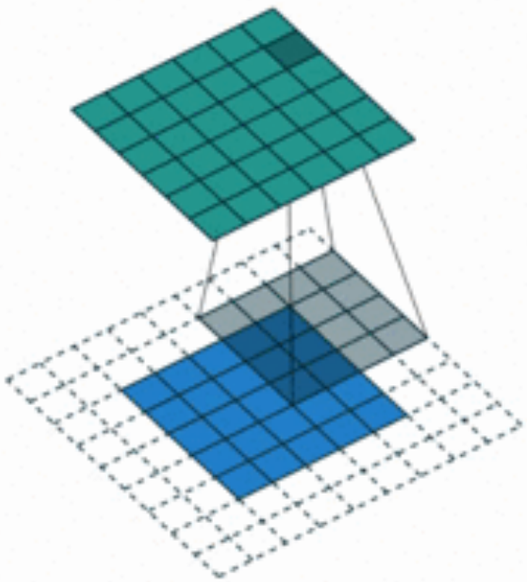
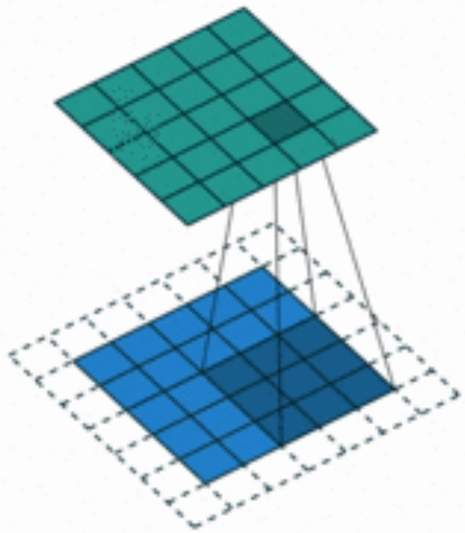
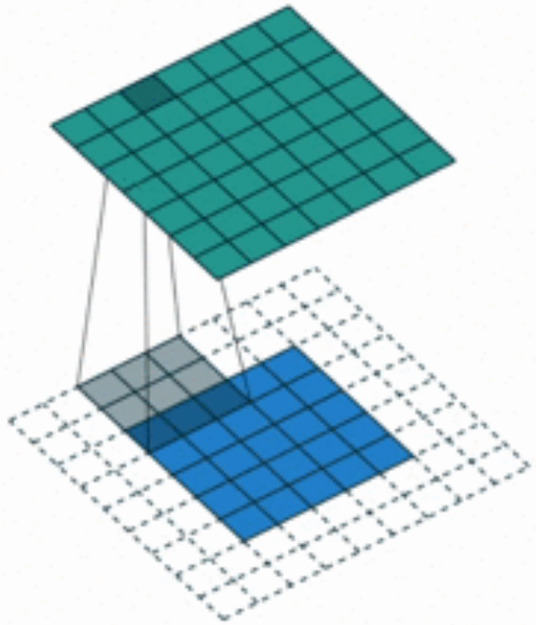
Hyperparameters: stride



Of how many pixels to move the filter
– this case has stride one but could be larger

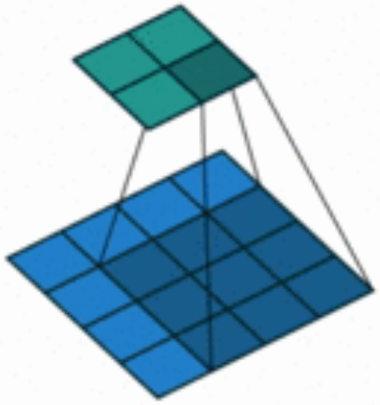
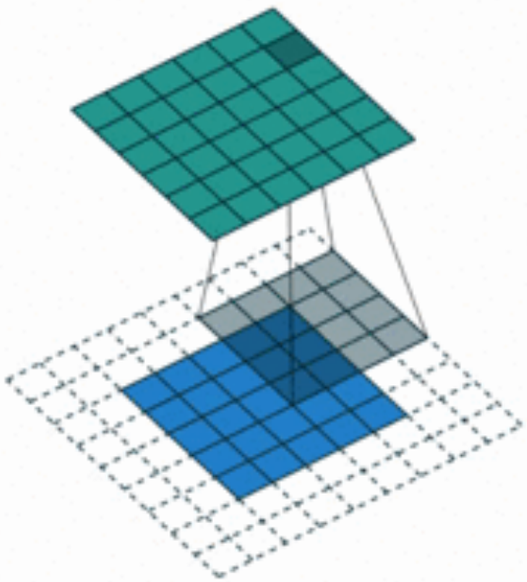
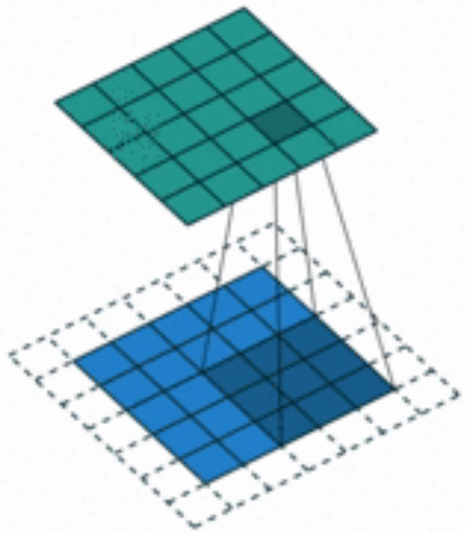
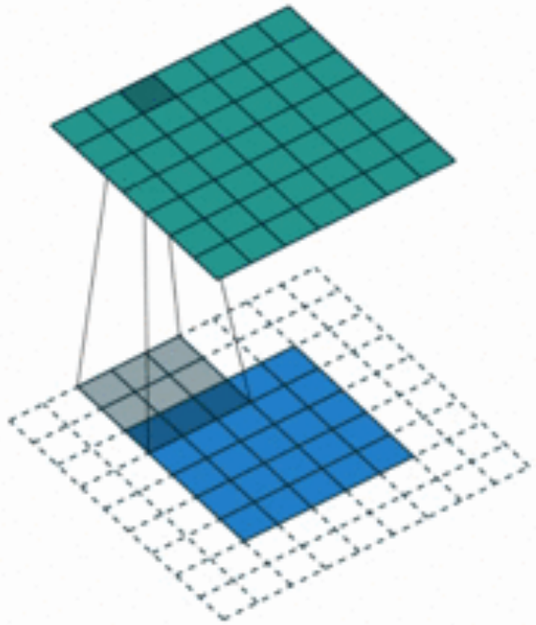
The stride basically defines how many times you see the same pixels
Caveat: even with stride 1 the corner pixels will be seen only once

Hyperparameters: padding

			
No padding, no strides	Arbitrary padding, no strides	Half padding, no strides	Full padding, no strides

How about adding zeros at the end so we can move the window and see every number exactly twice? The adding of zeros is called **padding**.

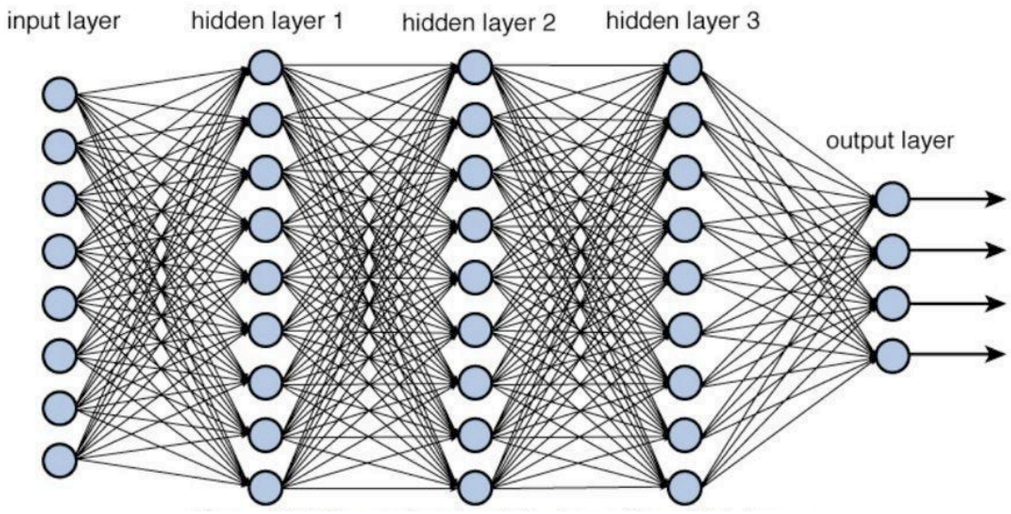
Hyperparameters: padding

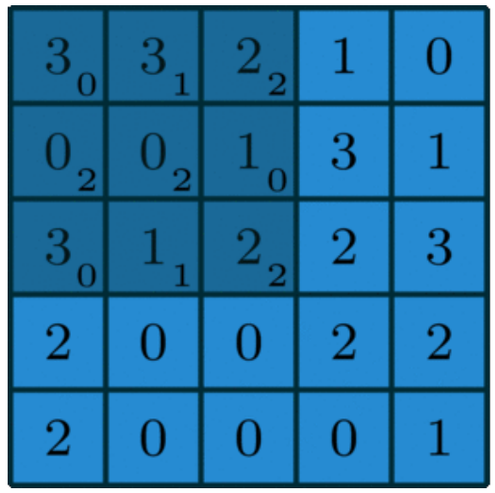
			
No padding, no strides	Arbitrary padding, no strides	Half padding, no strides	Full padding, no strides

How about adding zeros at the end so we can move the window and see every number exactly twice? The adding of zeros is called **padding**.

Shared weights

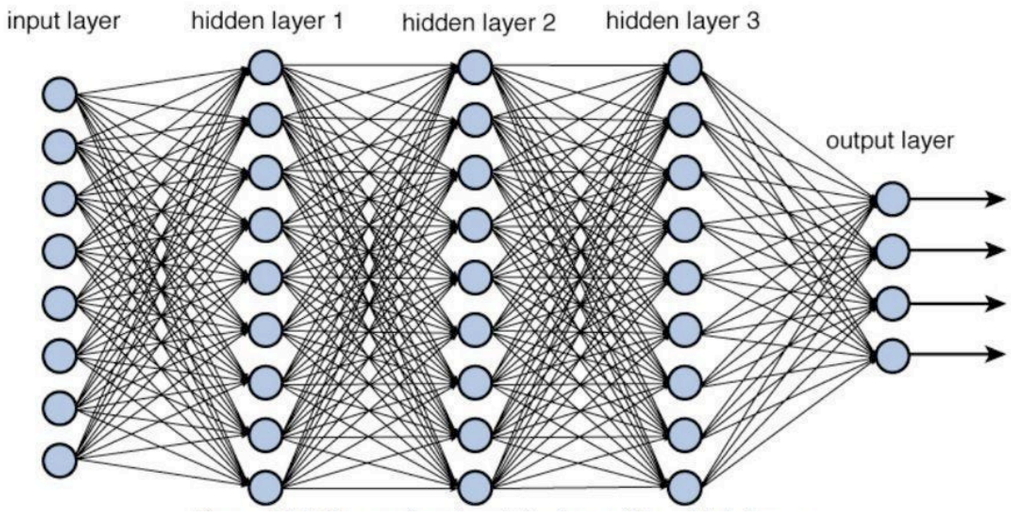
- Parameters are shared by each neuron producing in the activation map
- Dramatically reduces number of weights needed to produce an activation map
  - Data: 256×256×3 RGB image
  - Kernel: 3×3×3 → 27 weights
  - Fully connected layer: 256×256×3 inputs → 256×256×3 neurons → $O(10^{10})$ weights
  - and not translation equivariant!

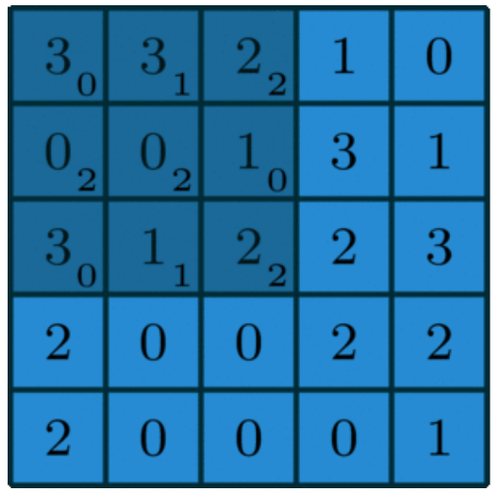




Shared weights

- Parameters are shared by each neuron producing in the activation map
- Dramatically reduces number of weights needed to produce an activation map
  - Data: 256×256×3 RGB image
  - Kernel: 3×3×3 → 27 weights
  - Fully connected layer: 256×256×3 inputs → 256×256×3 neurons → $O(10^{10})$ weights
  - and not translation equivariant!



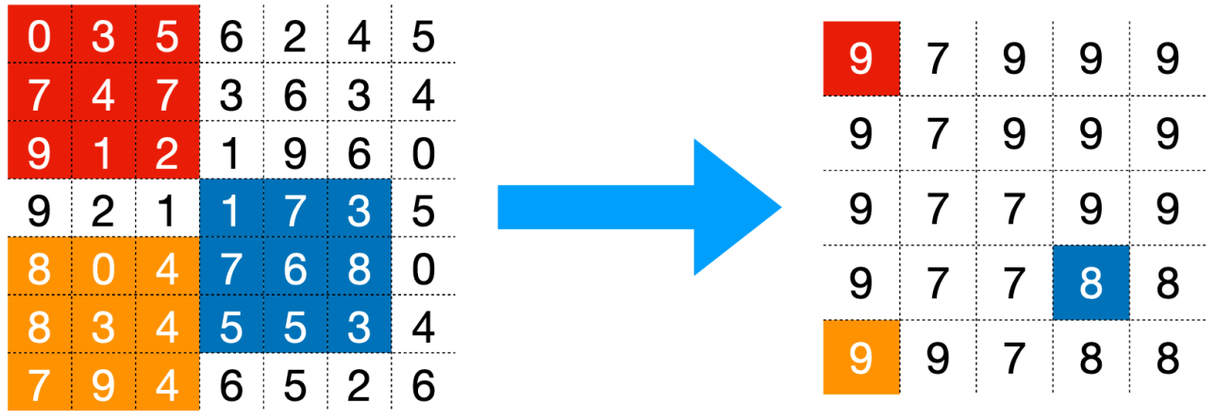


Dimensionality reduction: pooling

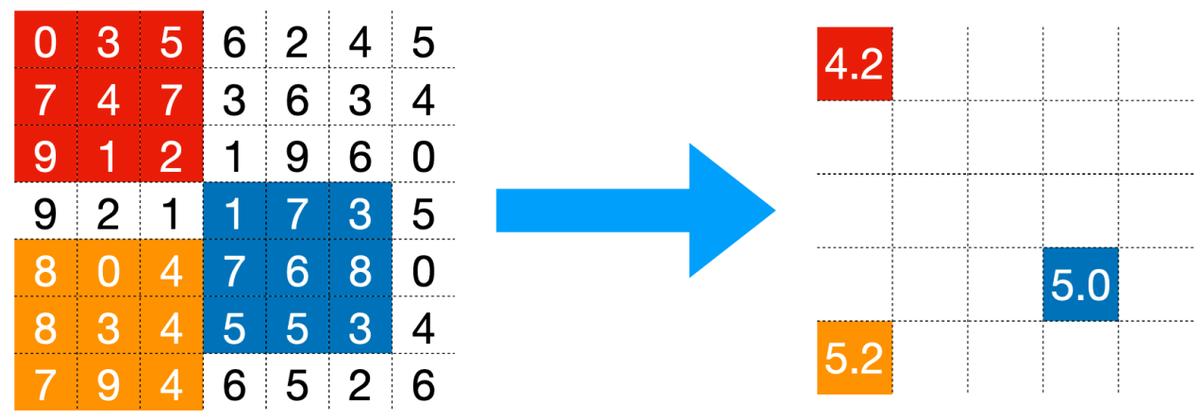
- Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network

- It also makes processing independent to local features (distortion, translation)

- **MaxPooling:** given an image and a filter of size *h x w*, scans the image and replaces each h_p *x* w_p patch with its maximum



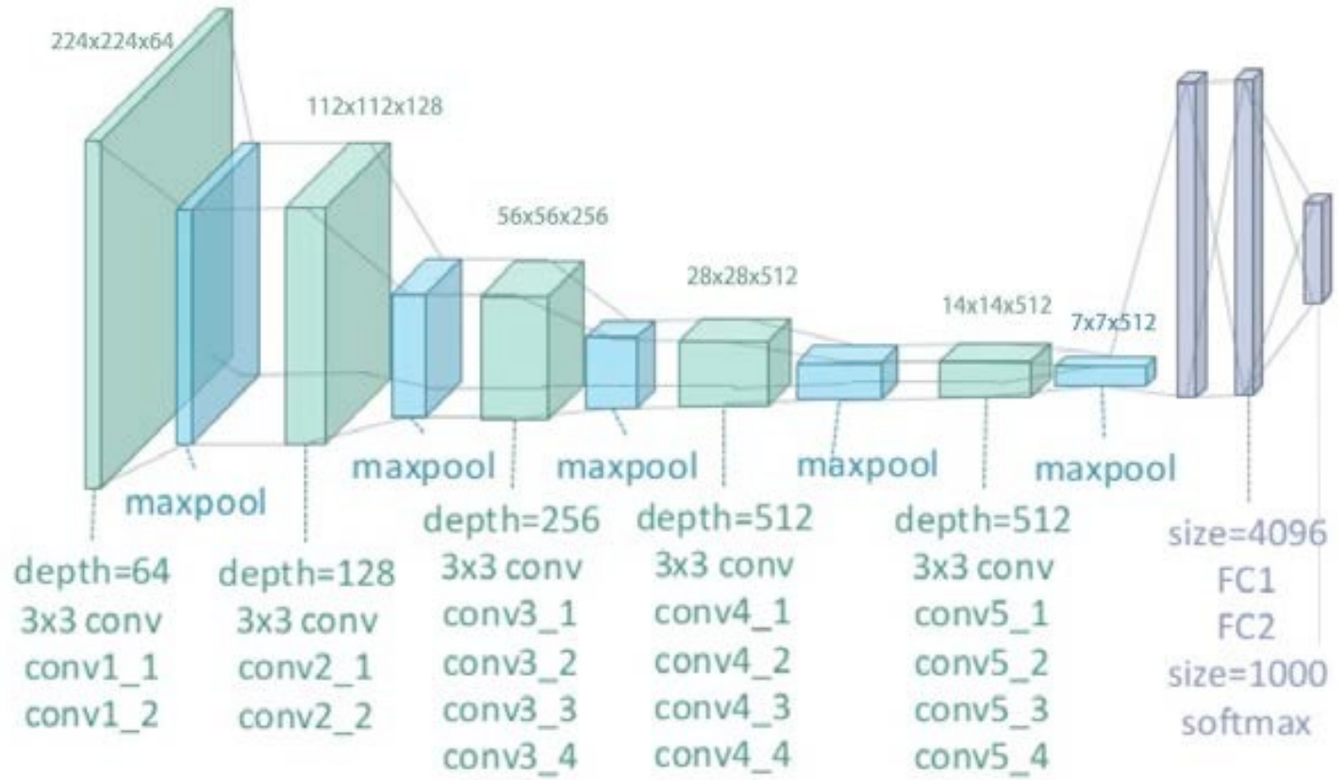
- **AveragePooling:** given an image and a filter of size *h x w*, scans the image and replaces each h_p *x* w_p patch with its average



The final CNN model

- A full 2D CNN is a sequence of 2D CNN+Pooling layers

- The 2D CNN+Pooling layers reduces the 2D image representation

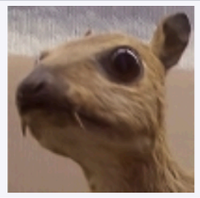
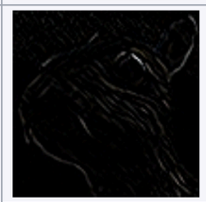
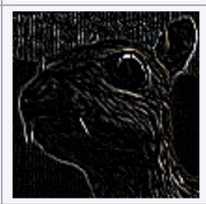
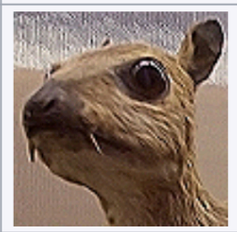
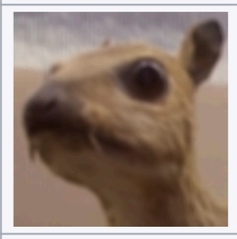
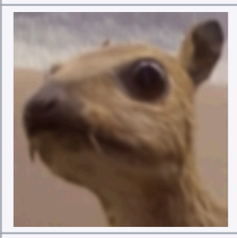


- The use of multiple filters on the image make the output grow on a third dimension

- Eventually, flattening occurs and the result is given to one or two MLP layers (e.g., for classification)

What does a CNN learn?

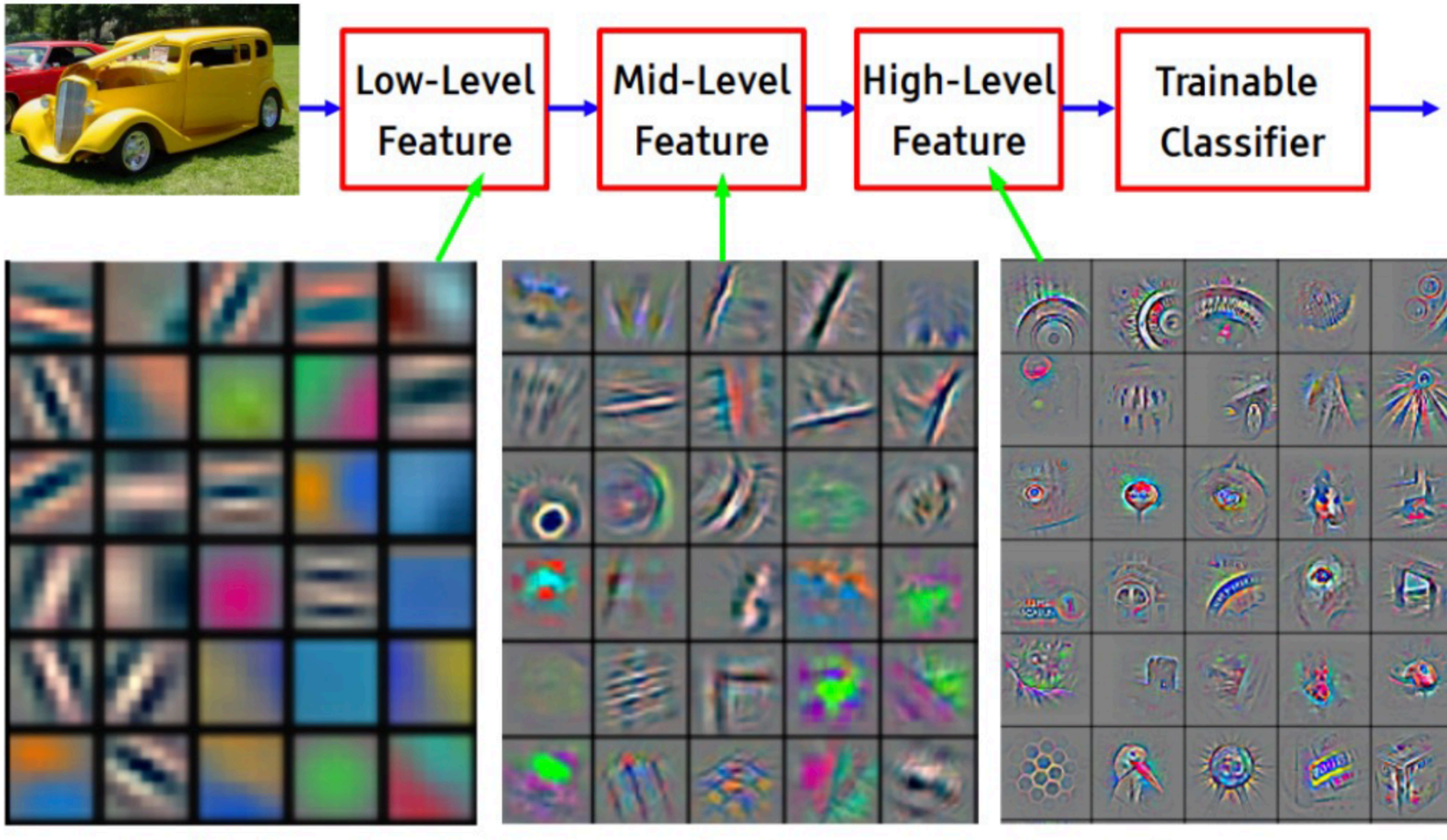
- Each filter alters the image in a different way, picking up different aspects of the image
  - edges oriented in various ways
  - enhancing / blurring of certain features
- In CNNs, filters are not defined: the value of each filter is learned during the training process.
- By being able to learn the values of different filters, CNNs can find more meaning from images that human designed filters might not be able to find

Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

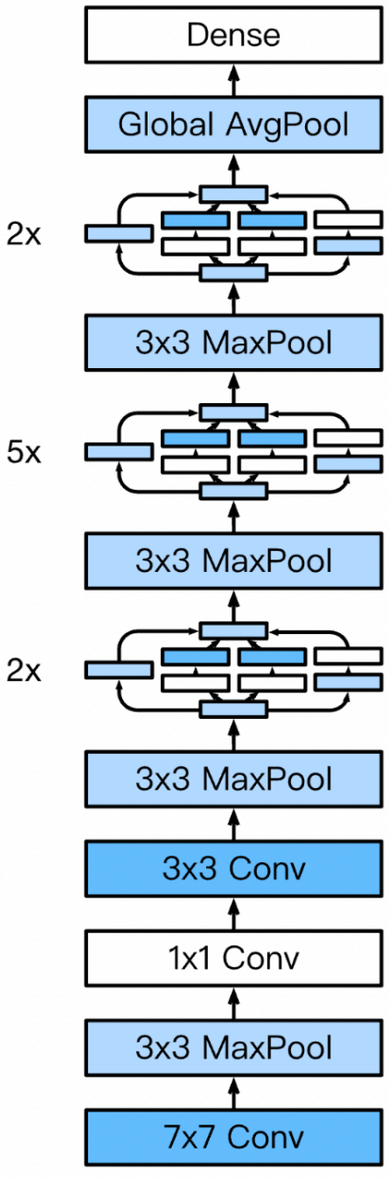
What does a CNN learn?

- The 1st layers appear to encode direction and color
- The direction and color filters get combined into grid and spot textures
- These textures gradually get combined into increasingly complex patterns

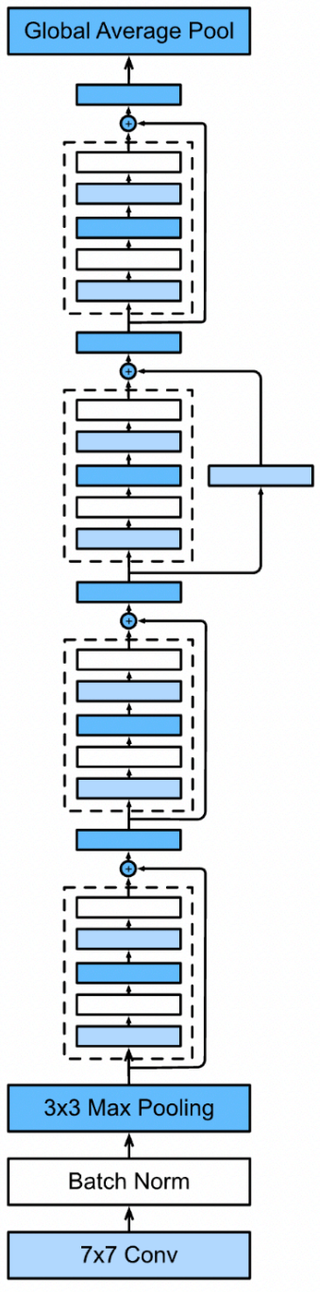
→ The network appears to learn a **hierarchical composition of patterns**



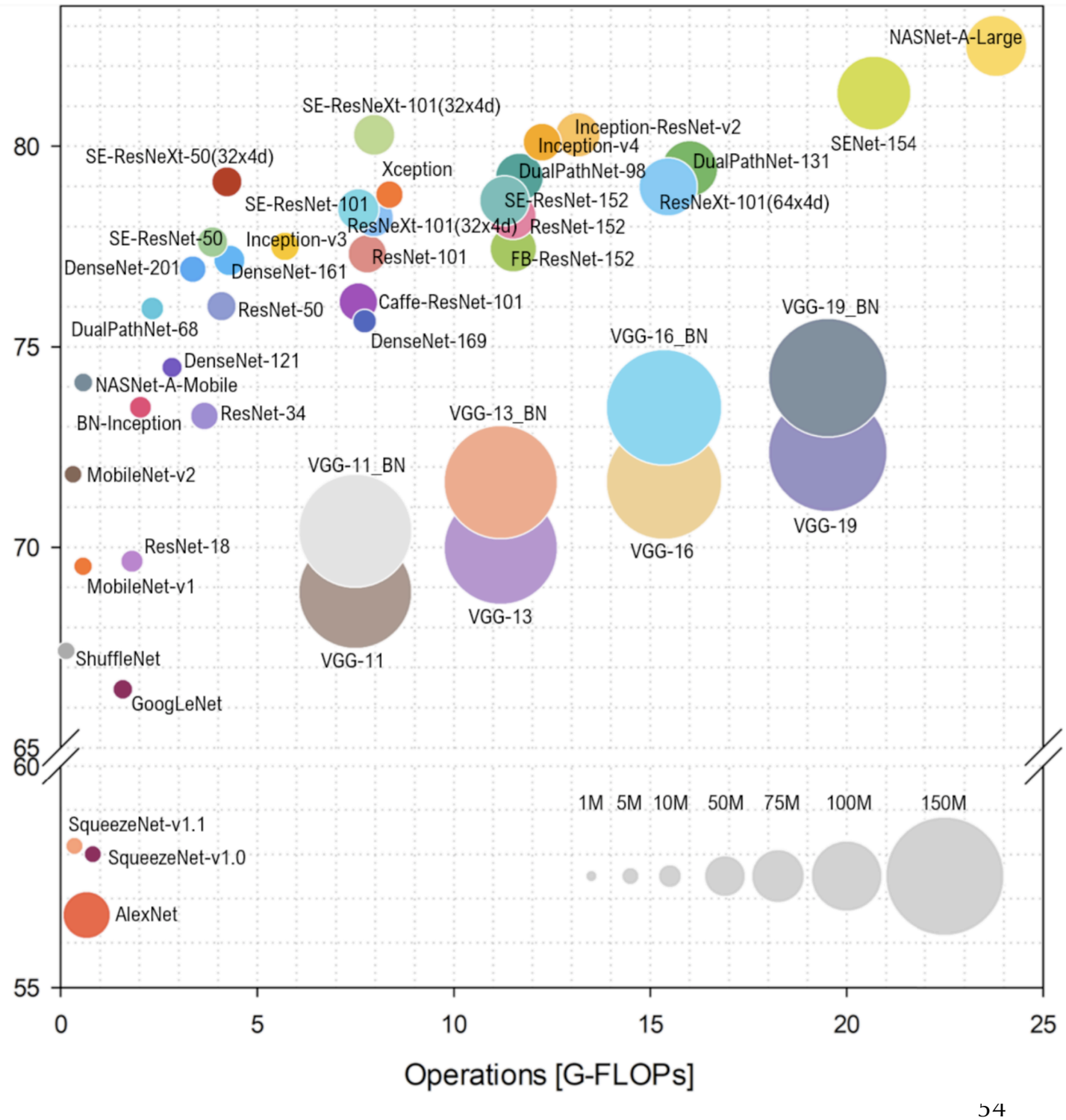
Deep CNNs



GoogLeNet
(Szegedy et al, 2014)



ResNet
(He et al, 2015)



Residual connection

- Observation that deeper models leads to higher training errors → performance degradation problem due to vanishing gradient

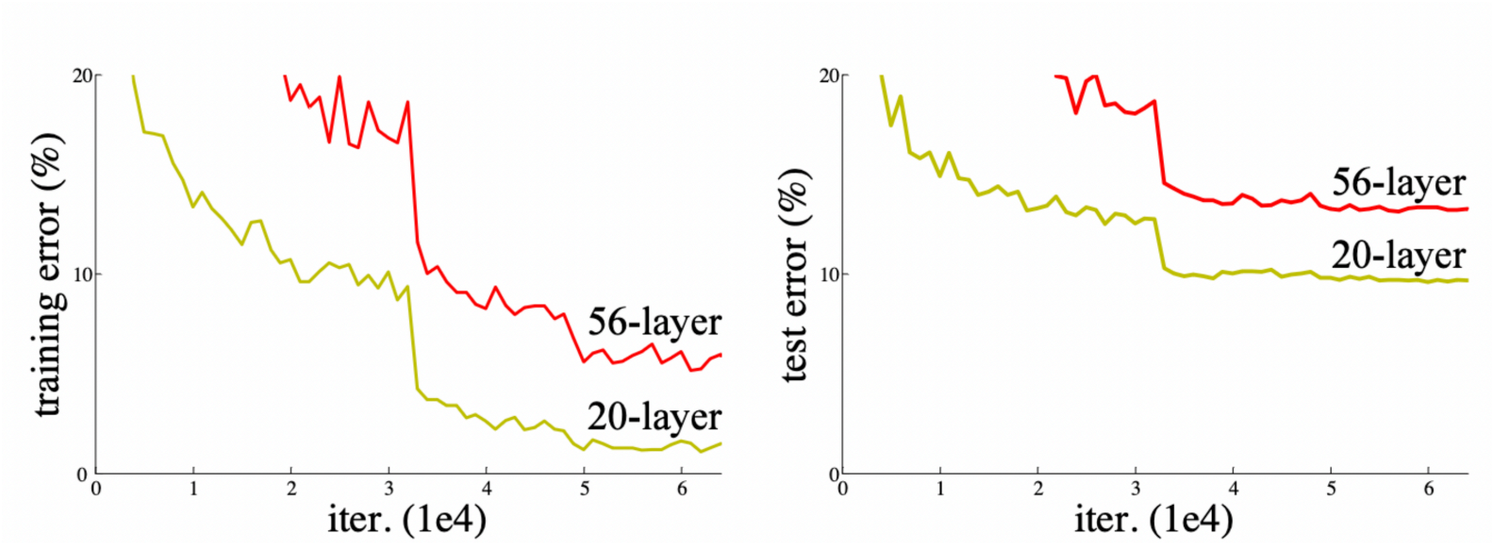


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

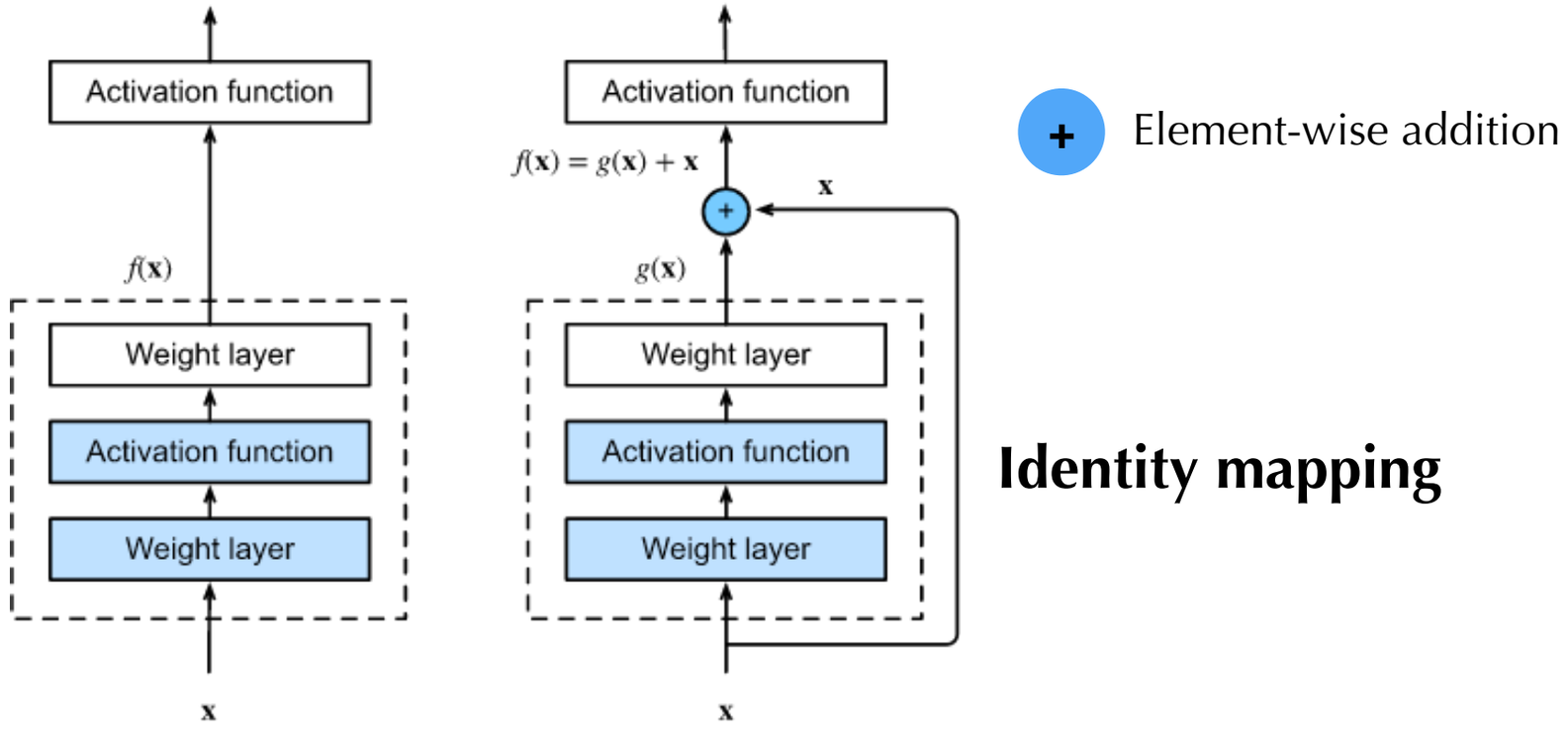
Residual connection

- Observation that deeper models leads to higher training errors → performance degradation problem due to vanishing gradient

- Breakthrough from Microsoft researchers was to introduced **residual connections** (also called *skip connections*):

  - instead of hoping that each stacked layer directly learn the underlying mapping $f(\mathbf{x})$ let them learn the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$

  - intuition: if an identity mapping $f(\mathbf{x}) = \mathbf{x}$ were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of non-linear layers



[Source]

Batch Normalization

Batch Normalization

- Training deep NNs is also complicated by the fact that the distribution of each layer's inputs changes during training as the parameters of the previous layers change
- Solution: during raining **batch normalization** shifts and rescales each layer output according to the mean and variance estimated on the batch

Mean and variance

$$\mu_B = \frac{1}{B}\sum_{b=1}^{B} x_b$$

$$\sigma_B^2 = \frac{1}{B}\sum_{b=1}^{B} (x_B - \mu_B)^2$$

Normalize

$$z_b = \frac{x_b - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}$$

Scale and shift

$$y_b = \gamma \odot z_b + \beta$$

Moving average

$$\mu_{M_b} = \alpha\mu_{M_{b-1}} + (1-\alpha)\mu_b$$

$$\sigma_{M_b}^2 = \alpha\sigma_{M_{b-1}}^2 + (1-\alpha)\sigma_b^2$$

Batch Normalization

- Training deep NNs is also complicated by the fact that the distribution of each layer's inputs changes during training as the parameters of the previous layers change
- Solution: during raining **batch normalization** shifts and rescales each layer output according to the mean and variance estimated on the batch

Mean and variance

$$\mu_B = \frac{1}{B}\sum_{b=1}^{B} x_b$$

$$\sigma_B^2 = \frac{1}{B}\sum_{b=1}^{B} (x_B - \mu_B)^2$$

Normalize

$$z_b = \frac{x_b - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}$$

for numerical stability

Scale and shift

$$y_b = \gamma \odot z_b + \beta$$

learnable parameters

Moving average

$$\mu_{M_b} = \alpha\mu_{M_{b-1}} + (1-\alpha)\mu_b$$

$$\sigma_{M_b}^2 = \alpha\sigma_{M_{b-1}}^2 + (1-\alpha)\sigma_b^2$$

- Parameter α is called momentum (recommended setting is 0.99)
  - acts as a weight factor for the past and the present information
- Final μ_M and σ_M values will be used in the evaluation/test fase (i.e. applied to each example of the dataset)

Batch Normalization

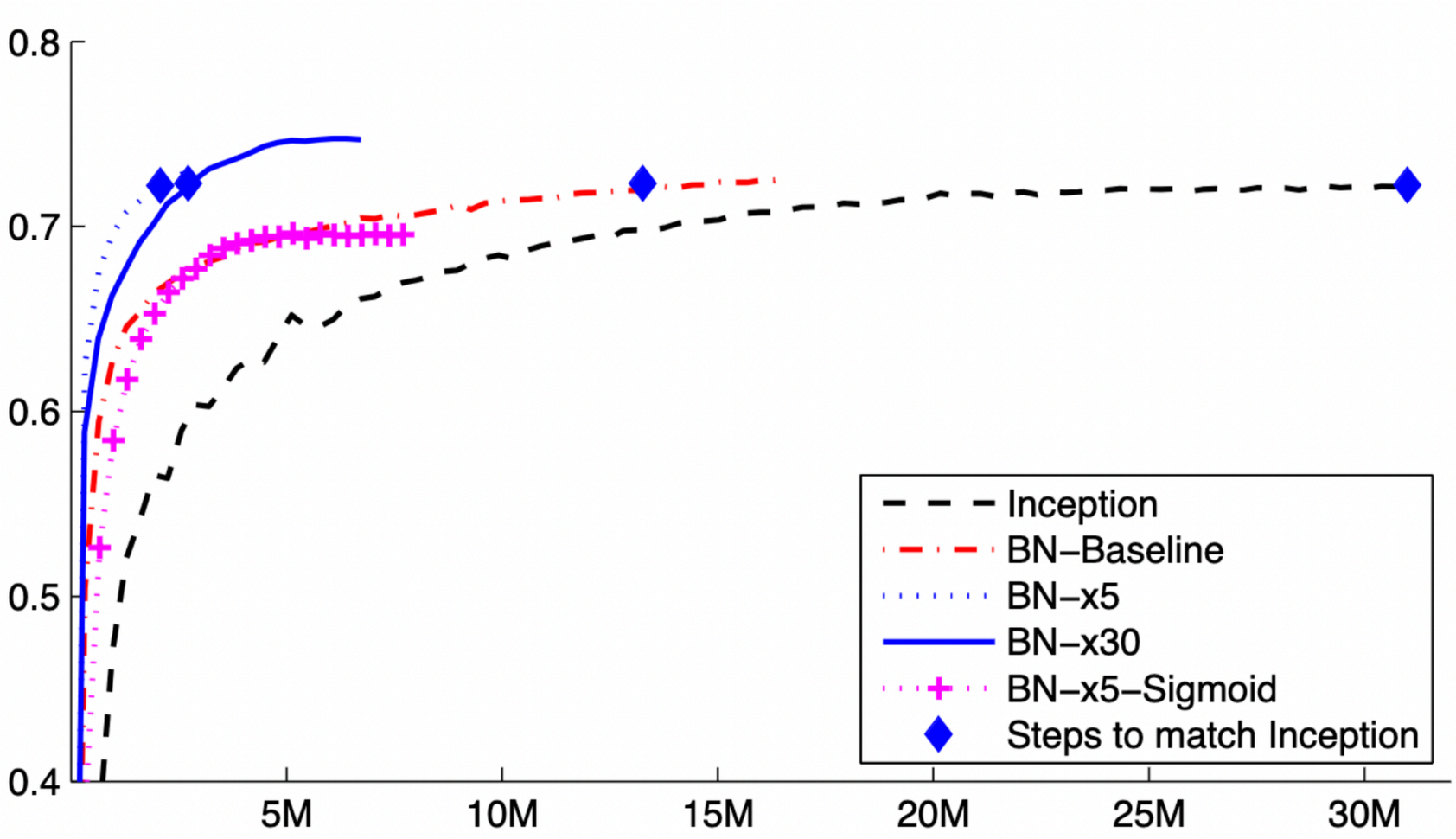


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

[Source]

Summary lecture #2

- Today we discussed Neural Networks as a tool to model the data beyond linear correlations

- We have seen how Deep Neural Networks like Convolutional NN can extract useful representations for the task from raw-level data

- During the course we will see many other architectures and how they adapt to the data and task at hand
  - tomorrow: recurrent and graph neural networks

- This afternoon we will see a first implementation of deep neural networks using Keras for a typical task at LHC experiments