Università di Verona

A.A 2020-2021

Machine Learning & Artificial Intelligence

Introduction to Deep Learning

Neural Networks, Loss functions, Backpropagation and Gradient Descent, Regularization and Dropout, Batch Normalization, Convolutional Neural Networks

Vittorio Murino

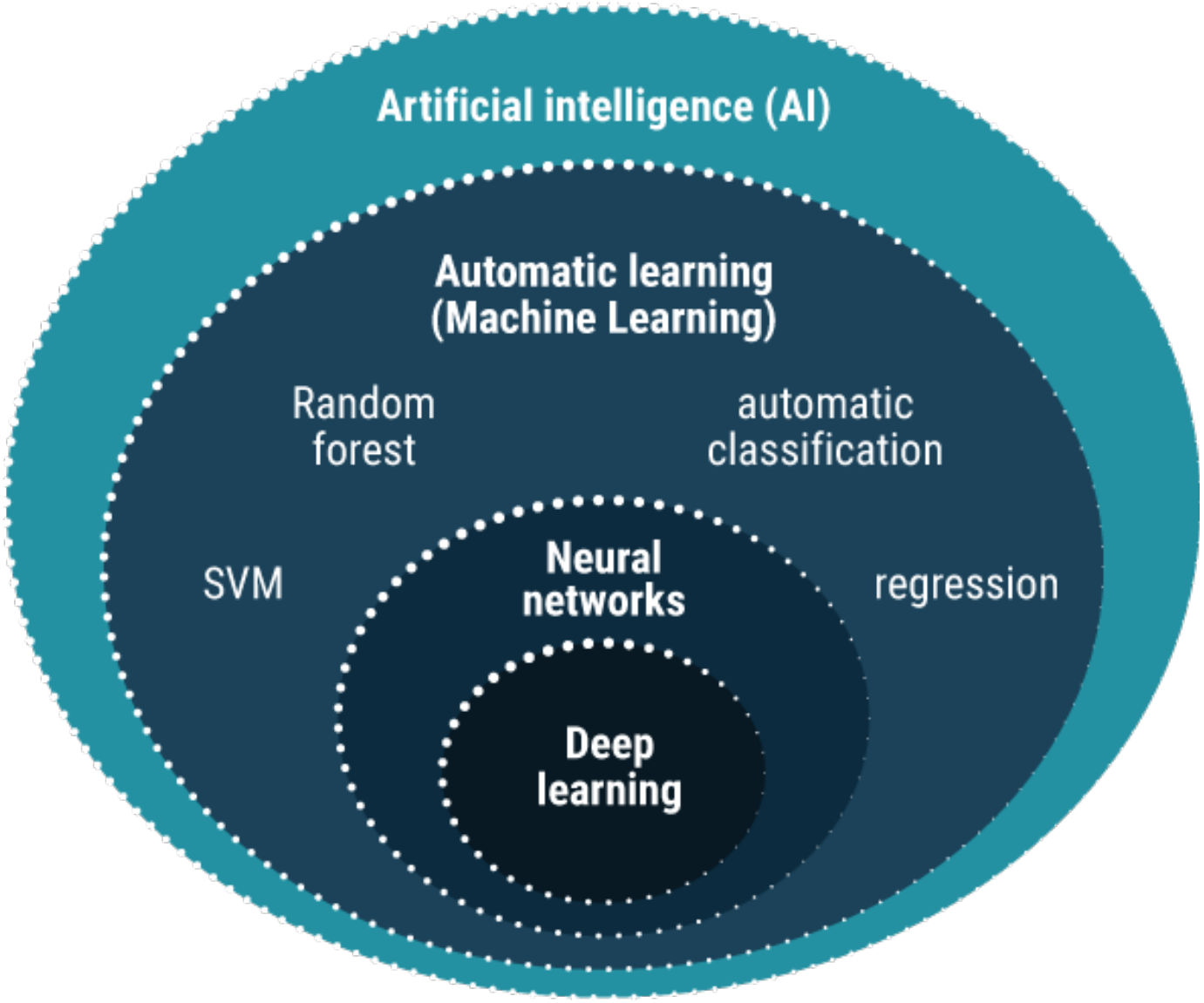
Credits

- Geri Skenderi

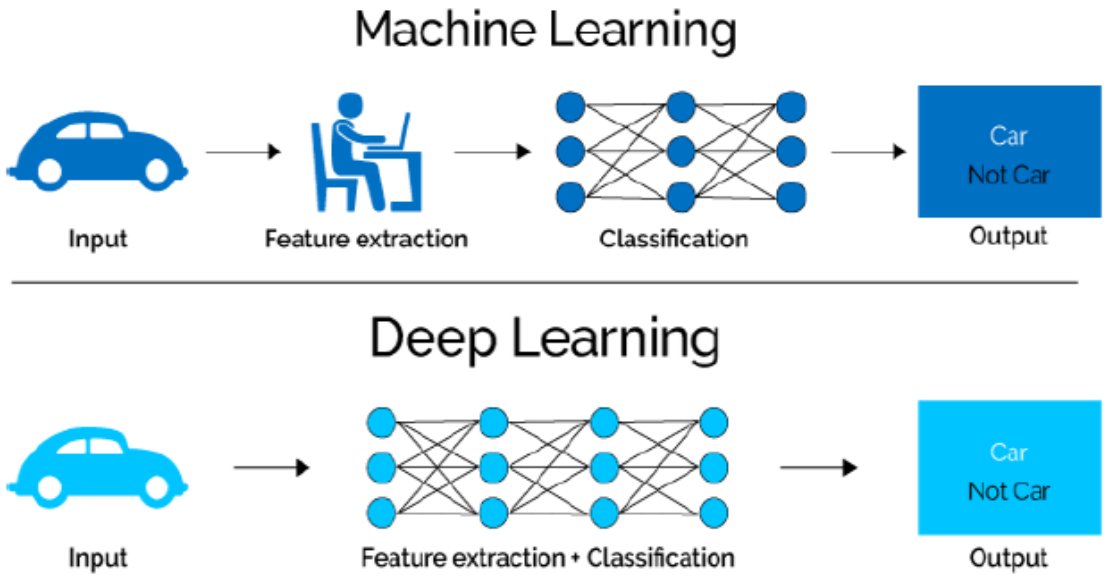
- Deep learning course @ MIT (deeplearning.mit.edu)
- Deep learning course @ Stanford
- Deep learning course @ UPC Barcelona (dlai.deeplearning.barcelona)

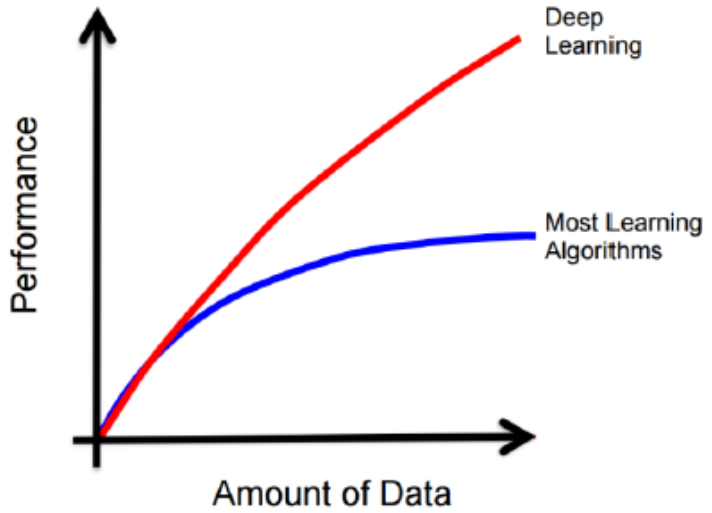
Deep Learning

- Subset of machine learning and representation (feature) learning.
- We wish to *learn underlying features directly from data*.



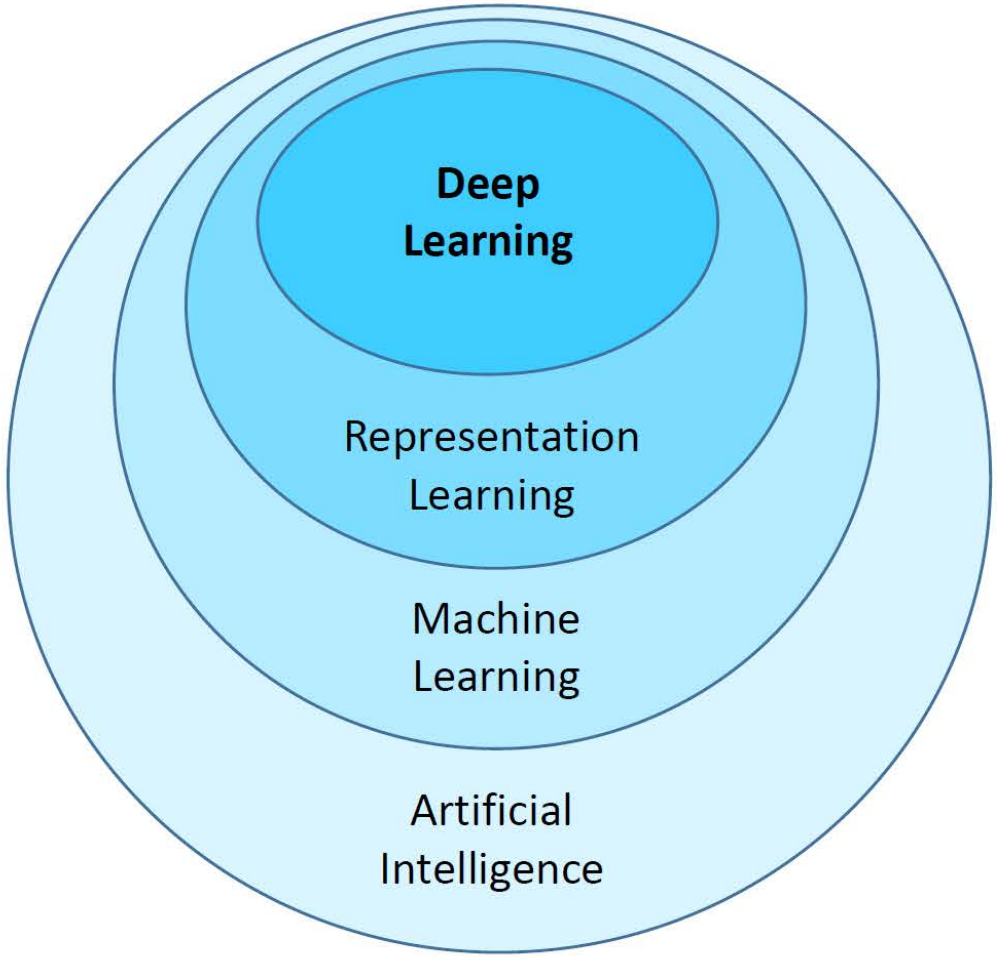
Why?

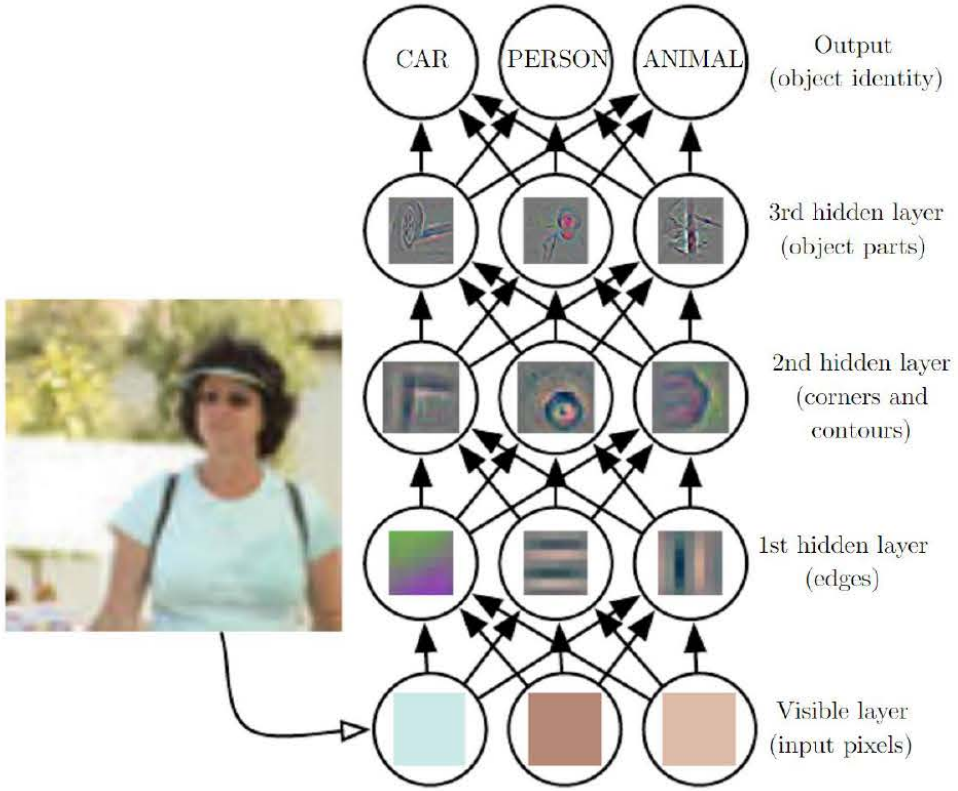

Machine Learning
Input
Feature extraction
Classification
Car
Not Car
Output
Deep Learning
Input
Feature extraction + Classification
Car
Not Car
Output


Performance
Amount of Data
Deep Learning
Most Learning Algorithms

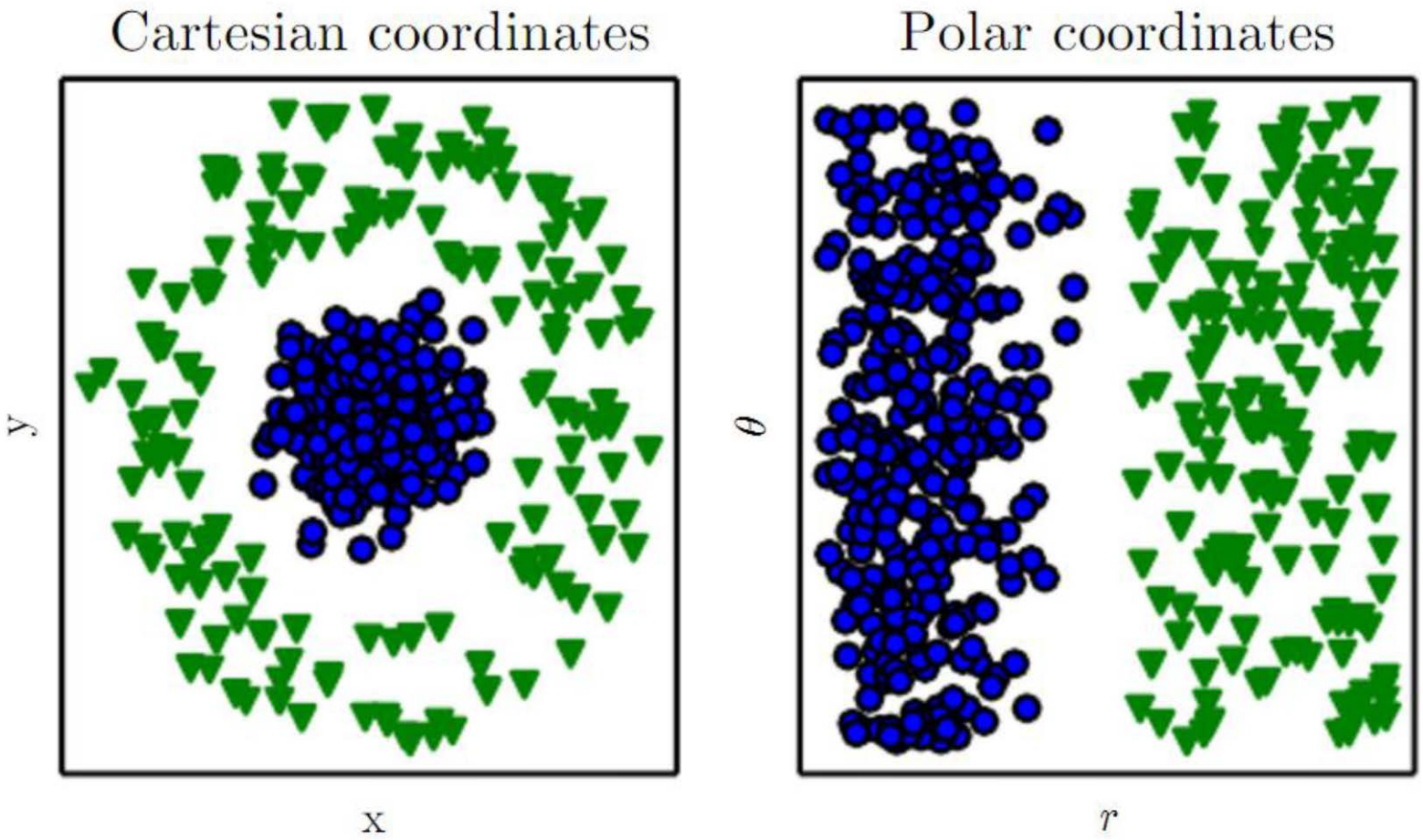
Deep Learning is **Representation Learning**

(aka Feature Learning)





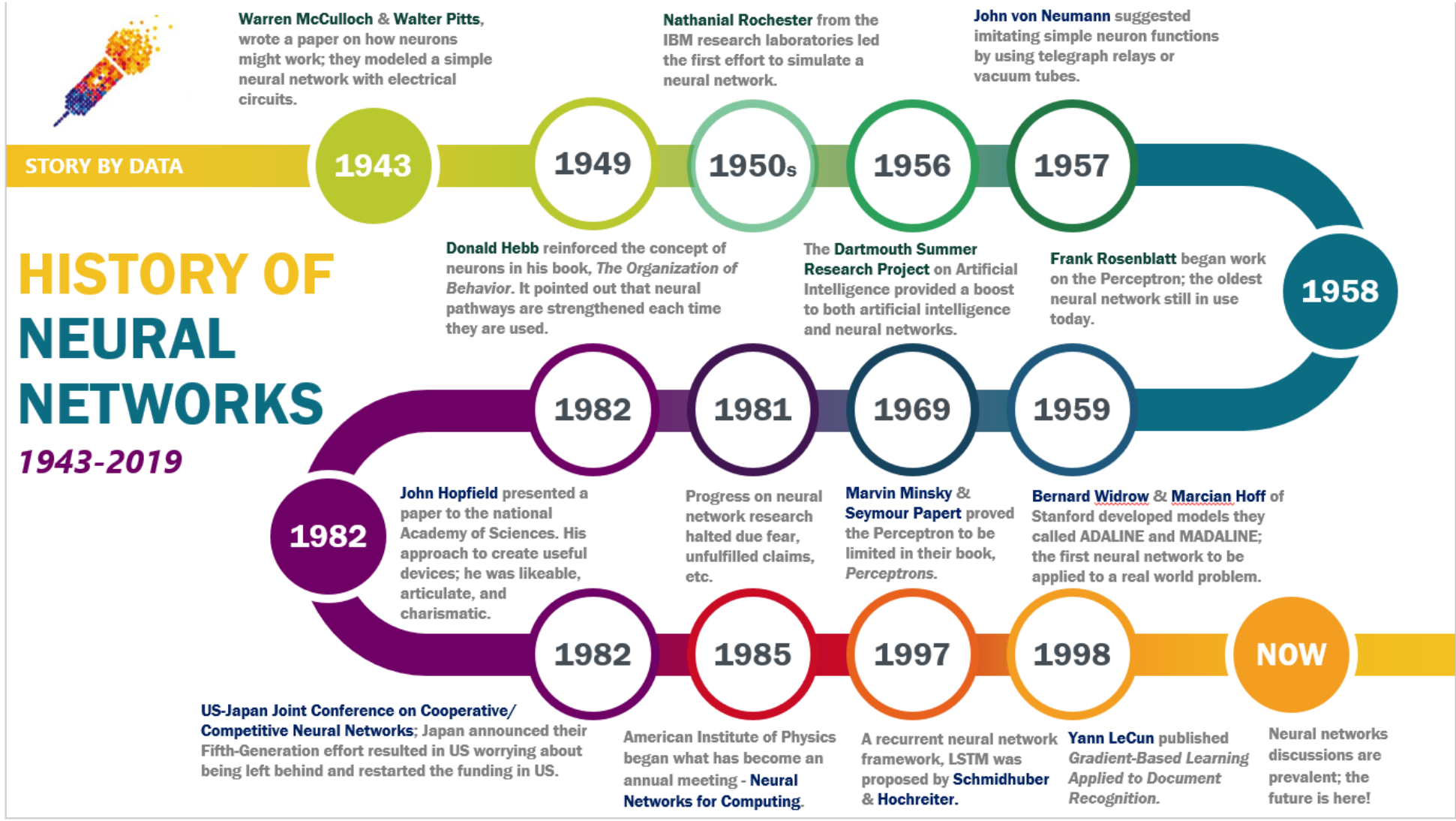
Representation Matters



Task: Draw a line to separate the **green triangles** and **blue circles**.

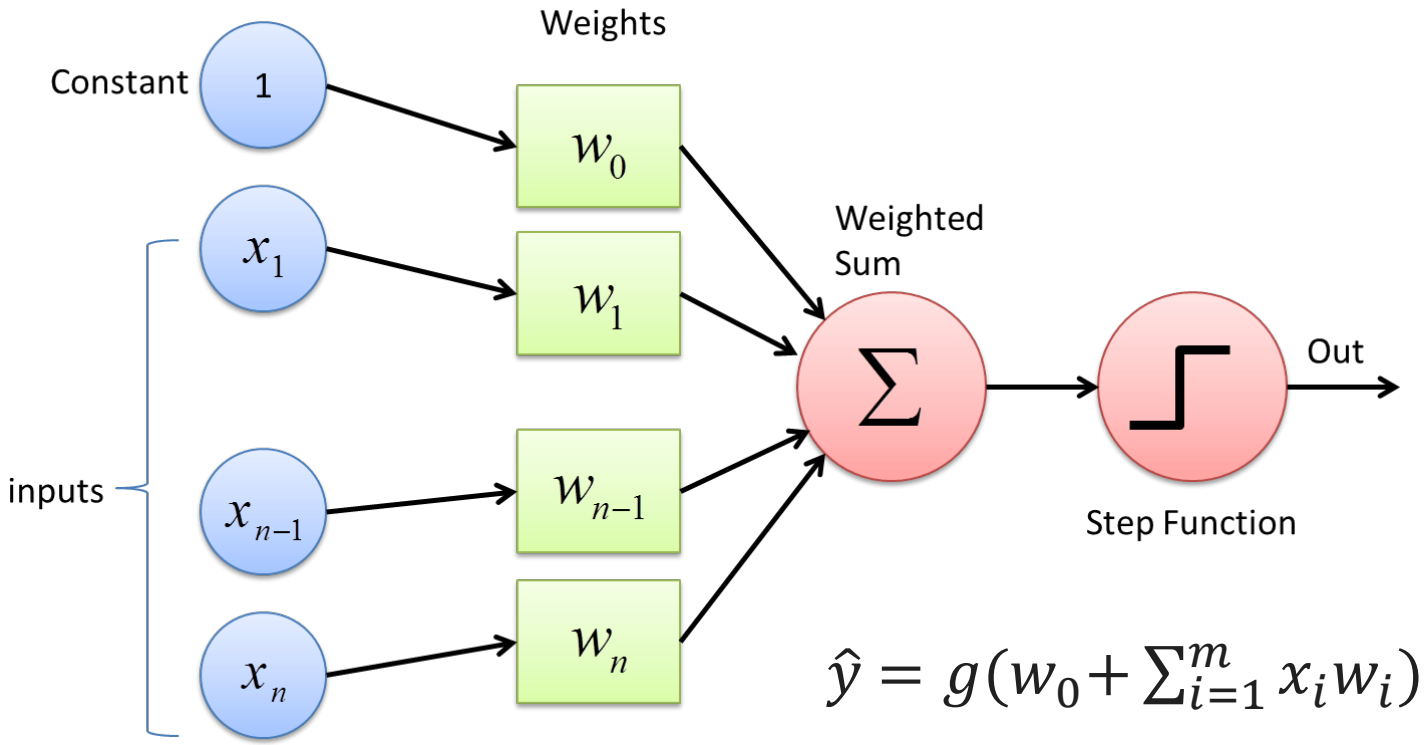
Why now?

1. Big data
2. Improved software libraries
3. Hardware



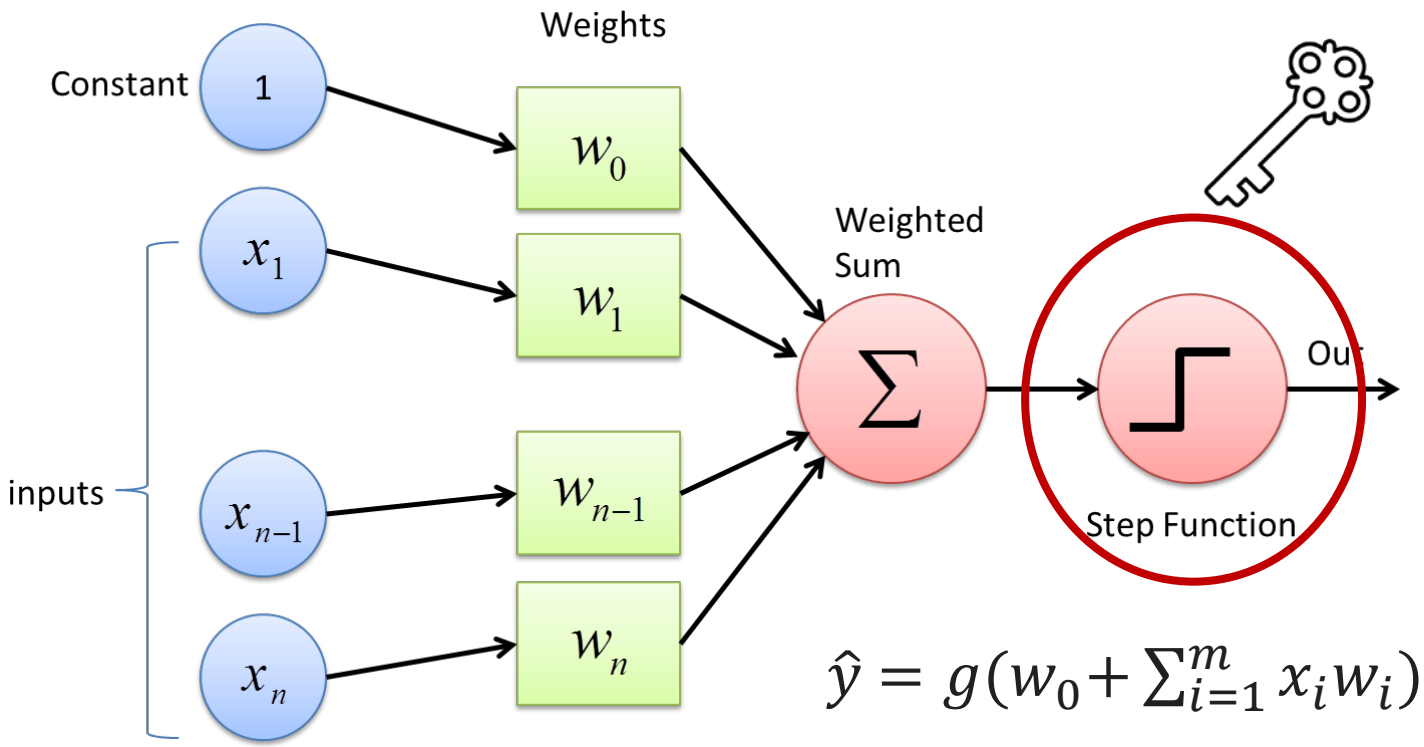
The perceptron

- The basic building block of neural networks.
- Information flows *forward.*



The perceptron

- The basic building block of neural networks.
- Information flows *forward.*

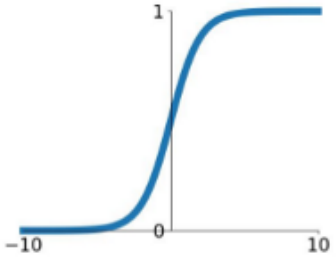


Activation functions

- They provide the non-linearity which makes these methods so powerful
- Crucial element of any architecture.

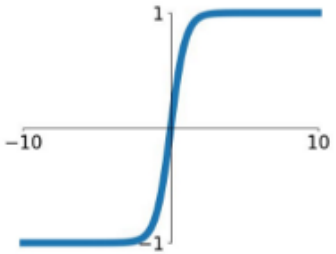
Sigmoid

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



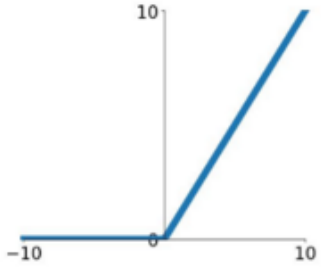
tanh

$$\tanh(x)$$



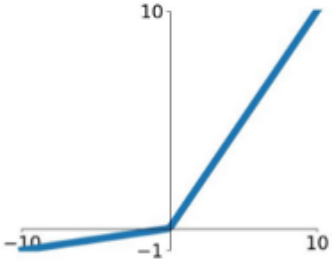
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

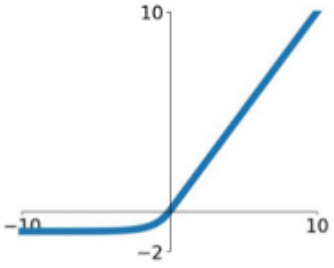


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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

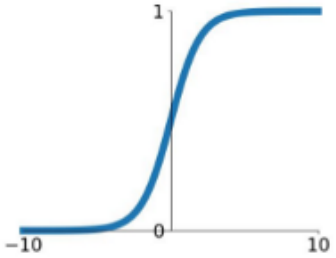


Activation functions

- They provide the non-linearity which makes these methods so powerful

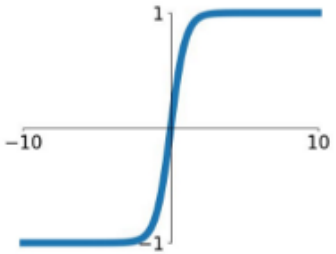
Sigmoid

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



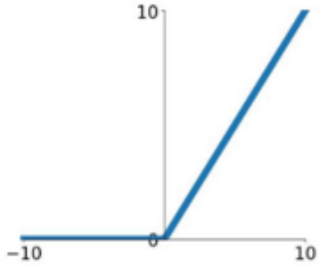
tanh

$$\tanh(x)$$



ReLU

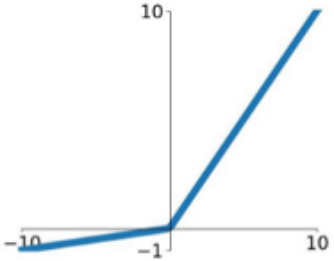
$$\max(0, x)$$



- Crucial element of any architecture.

Leaky ReLU

$$\max(0.1x, x)$$

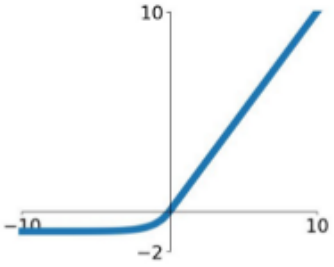


Maxout

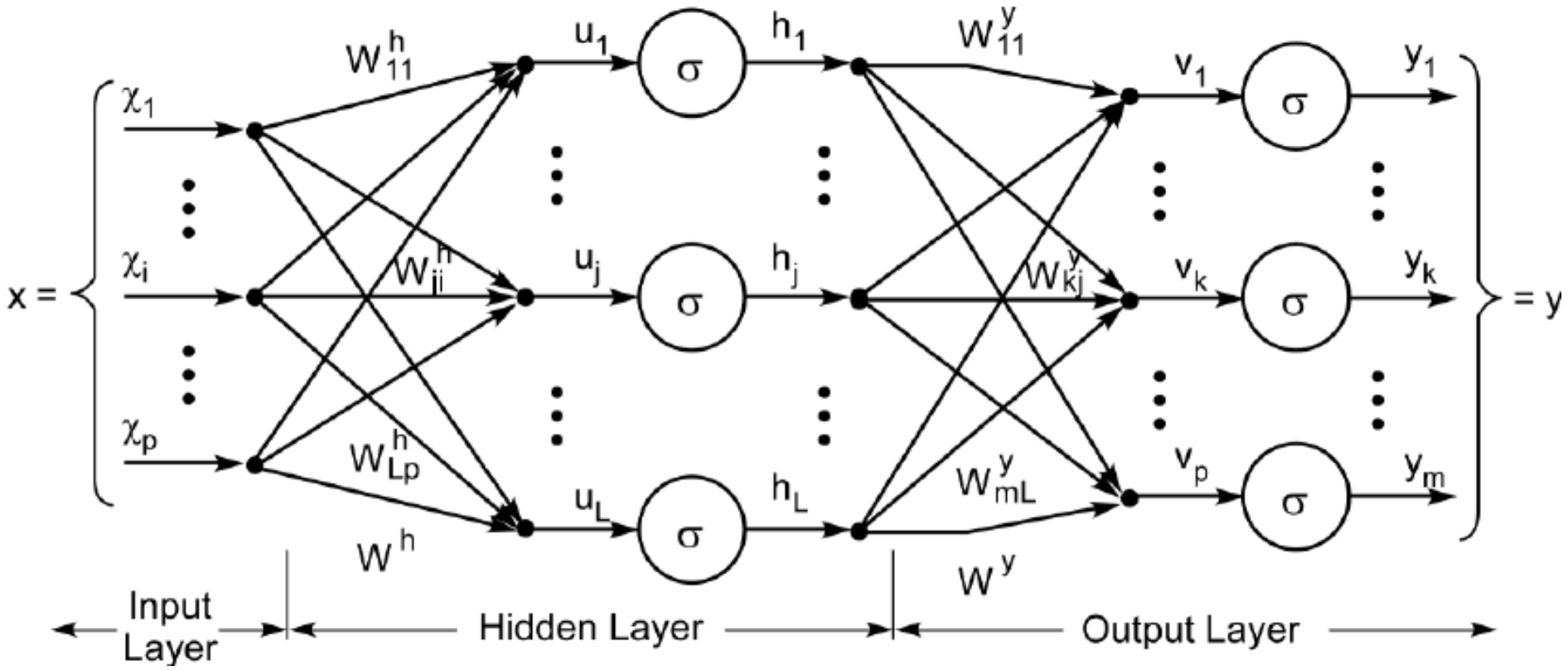
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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

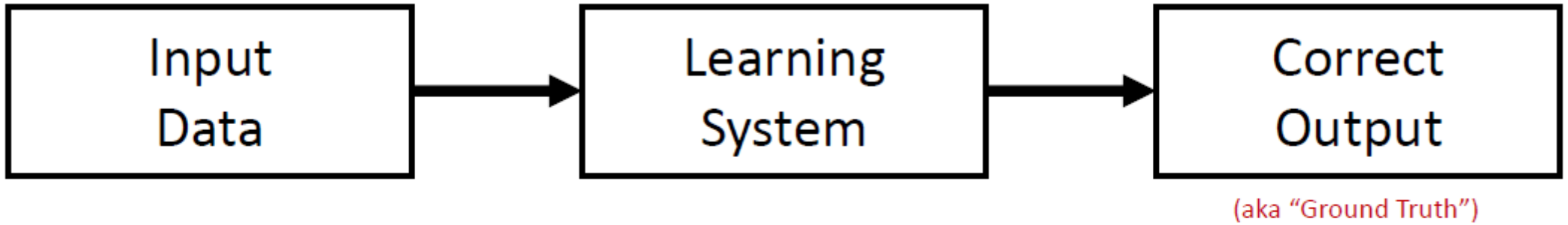


Neural Networks: Multilayer perceptron

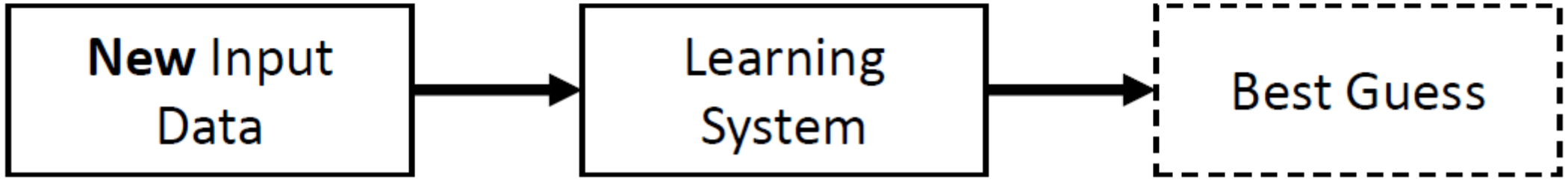


How do Neural Networks learn? (1/2)

Training Stage:



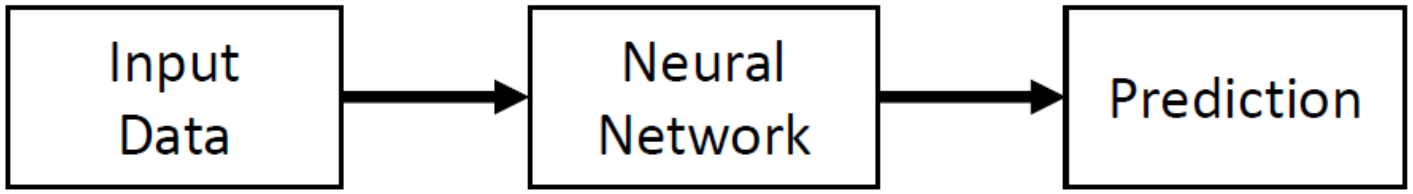
Testing Stage:



How do Neural Networks learn? (2/2)

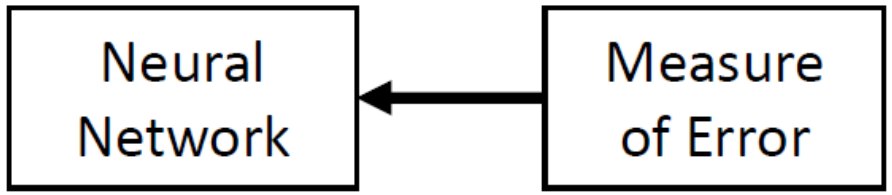
- We perform the forward pass to obtain a prediction.

Forward Pass:



- Based on a ***loss function***, we ***backpropagate to optimize the error***.

Backward Pass (aka Backpropagation):



Adjust to Reduce Error

Loss functions (1/2)

- Quantify the difference between prediction p and ground truth t.
- **Task dependent*:***
  1. *Regression* $MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$
  2. *Binary classification: binary cross entropy loss*

$$BCE\,(t,p) = -(t * \log(p) + (1-t) * \log\,(1-p))$$

  3. *Multiclass classification: cross entropy*

$$CCE(p,t) = -\sum_{c=1}^{C} t_{o,c}\log\,(p_{o,c})$$

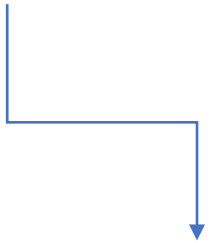
Loss functions (2/2)

- We can make use of these loss functions, which are *differentiable*, to pose our learning task as an *optimization problem*.
- By training the network, **we want to find the networks weights that achieve minimal loss.**
- This is the intuition behind the **backprogation procedure**, and the algorithm used to perform the weight update, called **Gradient Descent.**

Backpropagation (1/7)

- General idea: Update the **weights** and **biases** to decrease **loss function.**

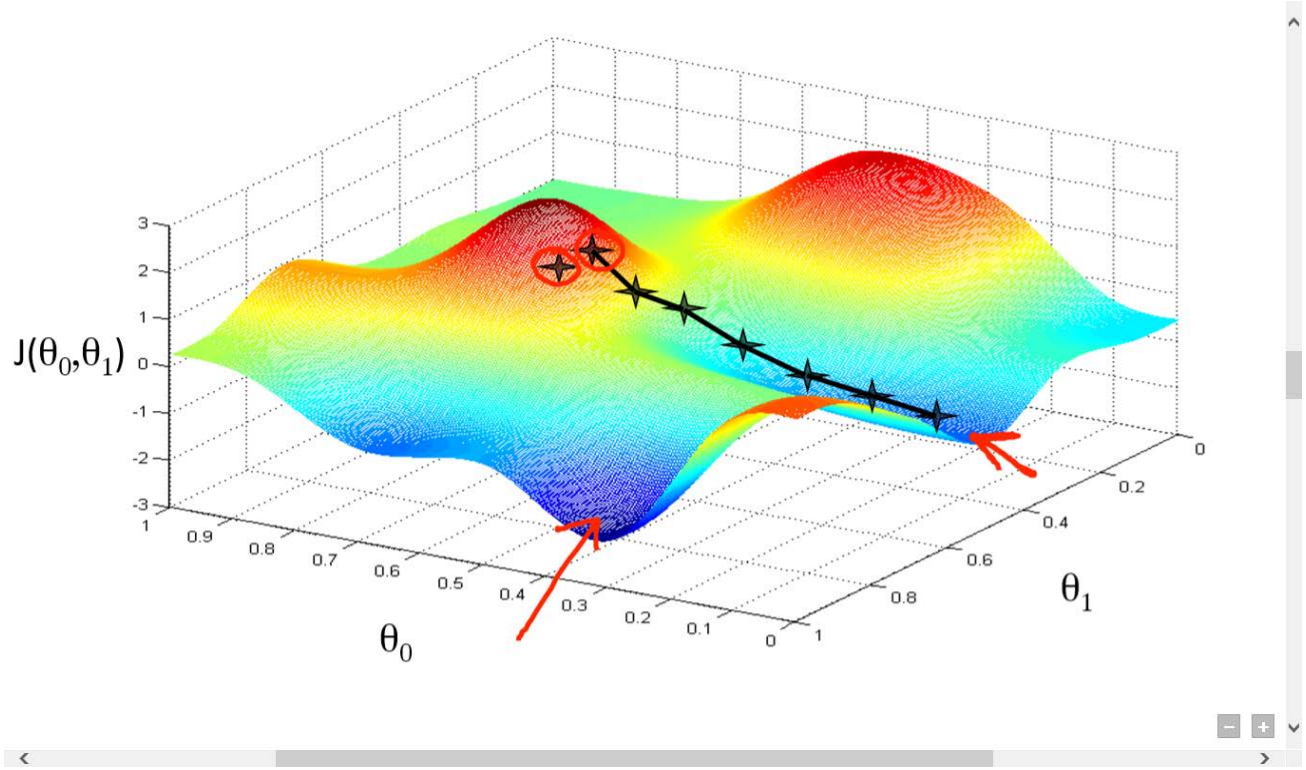
- Sub-tasks:
  1. Forward pass to compute network output and error.
  2. Backward pass to compute gradients.
  3. A fraction of the weight's gradient is subtracted from the weight.

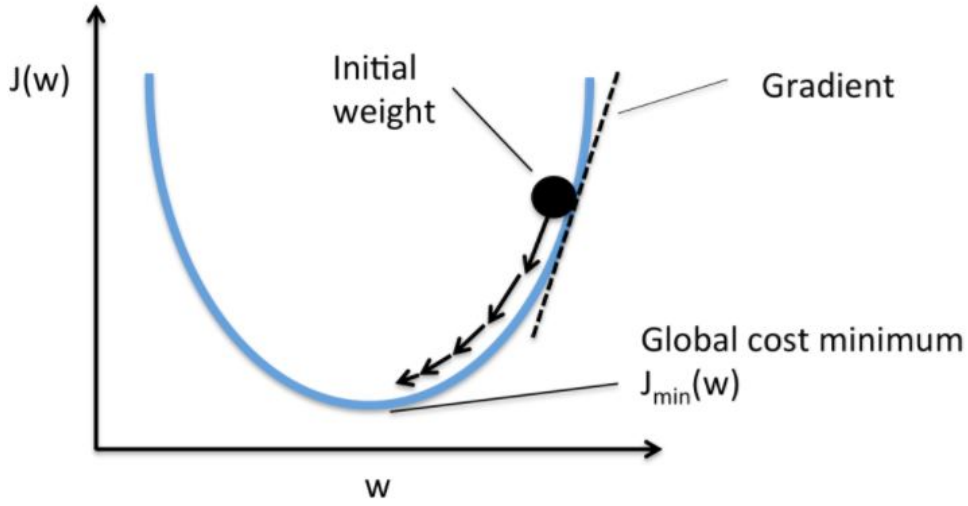


Decided by a hyperparameter called **learning rate.**

Backpropagation (3/7)

Weight update in slightly more mathematical terms: Calculate **the rate of change of the loss function** and **go in the direction which minimizes the loss → Gradient Descent.**

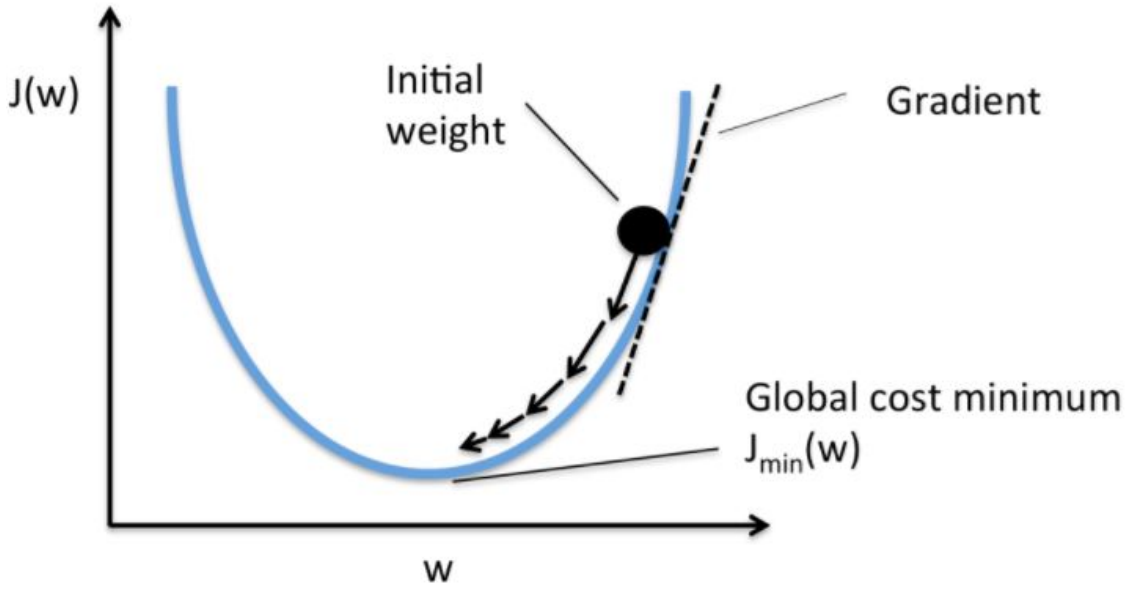




Backpropagation (4/7)

- Gradient Descent Algorithm:
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Compute the gradient $\frac{\partial J(w)}{\partial w}$
     2. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights

Learning rate



Backpropagation (5/7)

- Gradient Descent Algorithm:
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Compute the gradient $\frac{\partial J(w)}{\partial w}$ → Very expensive to compute for all training points (might be millions !)
     2. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights

Learning rate

Improve gradient descent computationally

Backpropagation (6/7)

- **Stochastic** Gradient Descent Algorithm:
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Pick single data point *i*
     2. Compute the gradient $\frac{\partial J_i(w)}{\partial w}$
     3. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$
  3. Return weights

Learning rate

- While this version is much easier to compute, it is very noisy (stochastic) since it relies on single instances.

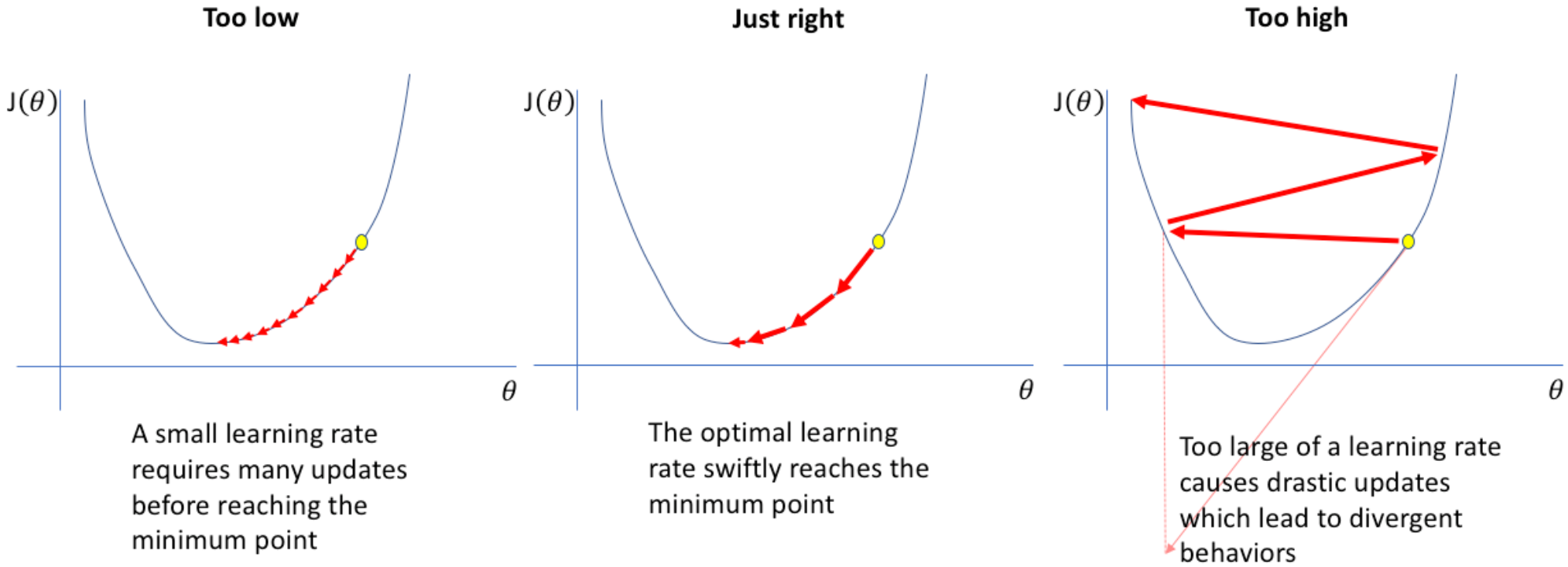
Backpropagation (7/7)

- **Mini-batch** Gradient Descent Algorithm (the happy, middle ground):
  1. Initialize weights randomly $\sim N(0, \sigma^2)$
  2. Repeat until convergence:
     1. Pick a batch *B* of data points
     2. Compute the gradient $\frac{\partial J_B(w)}{\partial w} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(w)}{\partial w}$
     3. Update weights $w \leftarrow w - \eta \frac{\partial J(w)}{\partial w}$

        Learning rate
  3. Return weights
- Mini-batch gradient descent allows for smoother convergence, more stable gradients and larger learning rates.

Learning rate

- Considered the most crucial hyperparameter in neural network training.
- We want to have a stable learning rate which converges smoothly and avoids local minima. Since the learning rate is a hyperparameter, it depends a lot on the problem and the network architecture, so it requires extensive experiments.



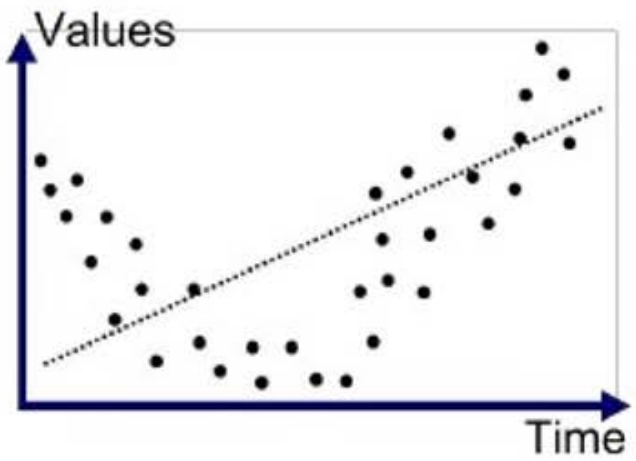
Model fit (1/2)

- After training, as in all machine learning approaches we have seen so far, we want to understand how well our model fits the data.

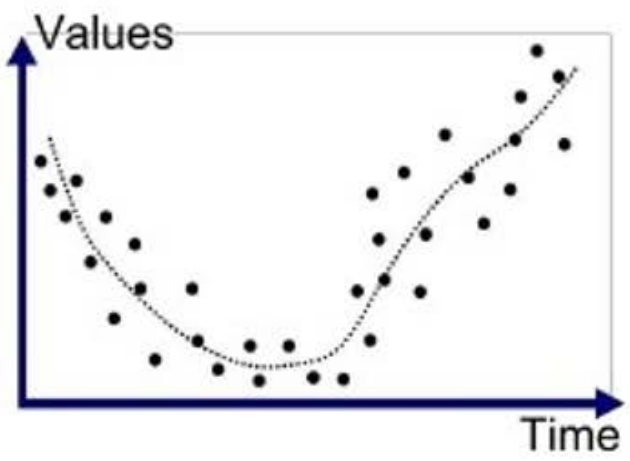
- Deep learning methods are troublesome when it comes to fitting the data, because of their high expressive power and the ability to directly learn underlying (linear and non-linear) relationships.

- In simpler words, this means that a neural network will often tend to memorize the training data, which is not what we want. We want the ability **to generalize,** or even simpler, the ability to perform well on unseen data.

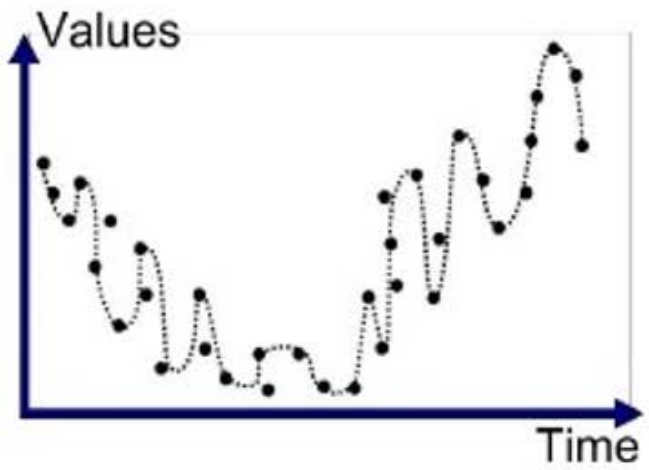
Model fit (2/2)



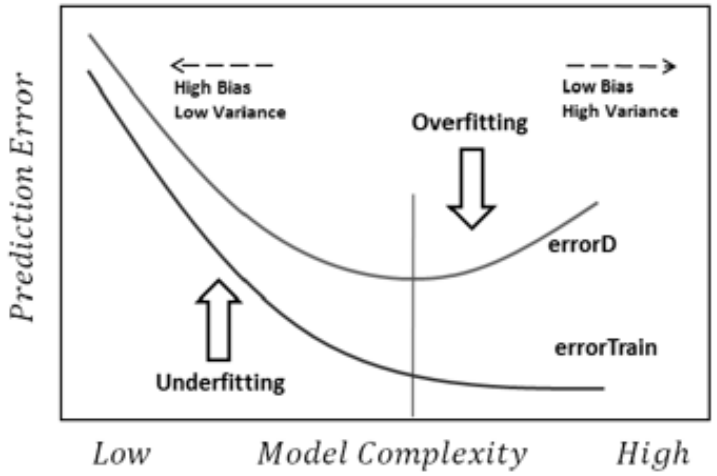
Underfitted



Good Fit/Robust



Overfitted



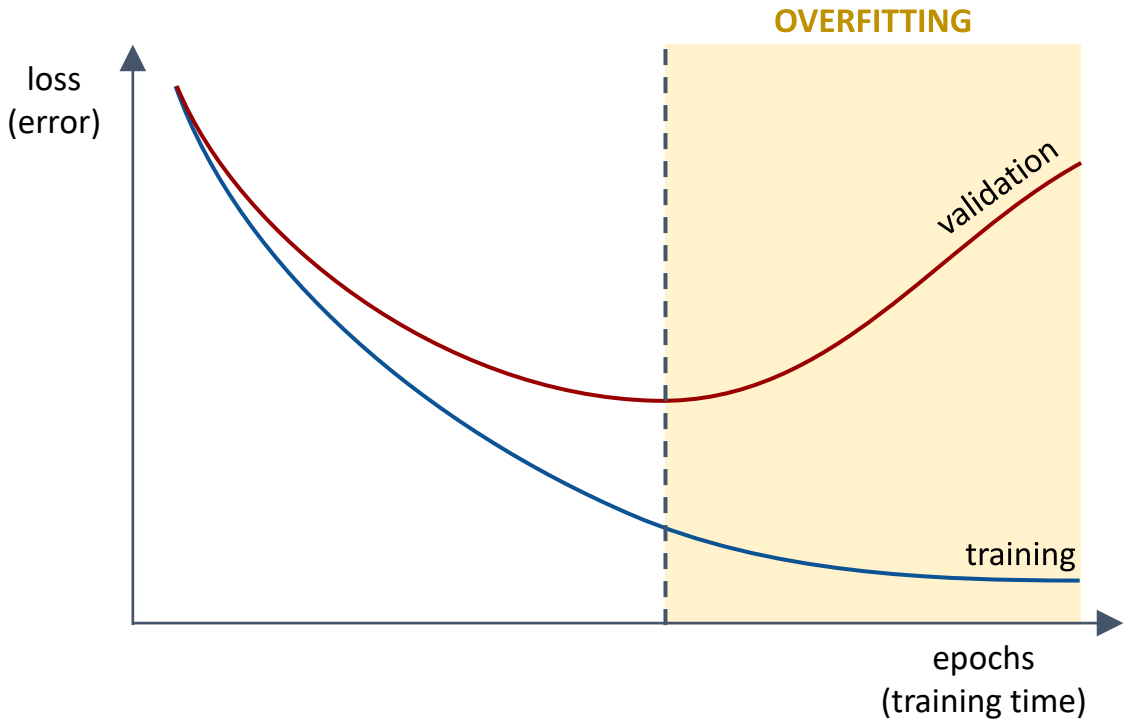
Regularization

- Technique that encourages the model *not to* become complex and overfit the data.
- By constraining the optimization problem in such a way, regularization methods improve the ability of the network to **generalize on unseen data**

- There are several techniques used for regularization. We will take a look at three common ones:
  1. Early termination
  2. Ridge regression (L2 regularization)
  3. Dropout (NN specific)

Regularization - Early termination

- IDEA: stop the training process as soon as the network stops improving the performance of the validation set



Regularization - Ridge regression

IDEA: apply artificial constraints on the network to reduce the number of free parameters

$$\mathcal{L}' = \mathcal{L} + \beta \frac{1}{2} \|W\|_2^2 = \mathcal{L} + \beta \frac{1}{2} \sum_i w_i^2$$

new loss

add **regularization term** (L_2 norm) to the loss function which penalizes large weights

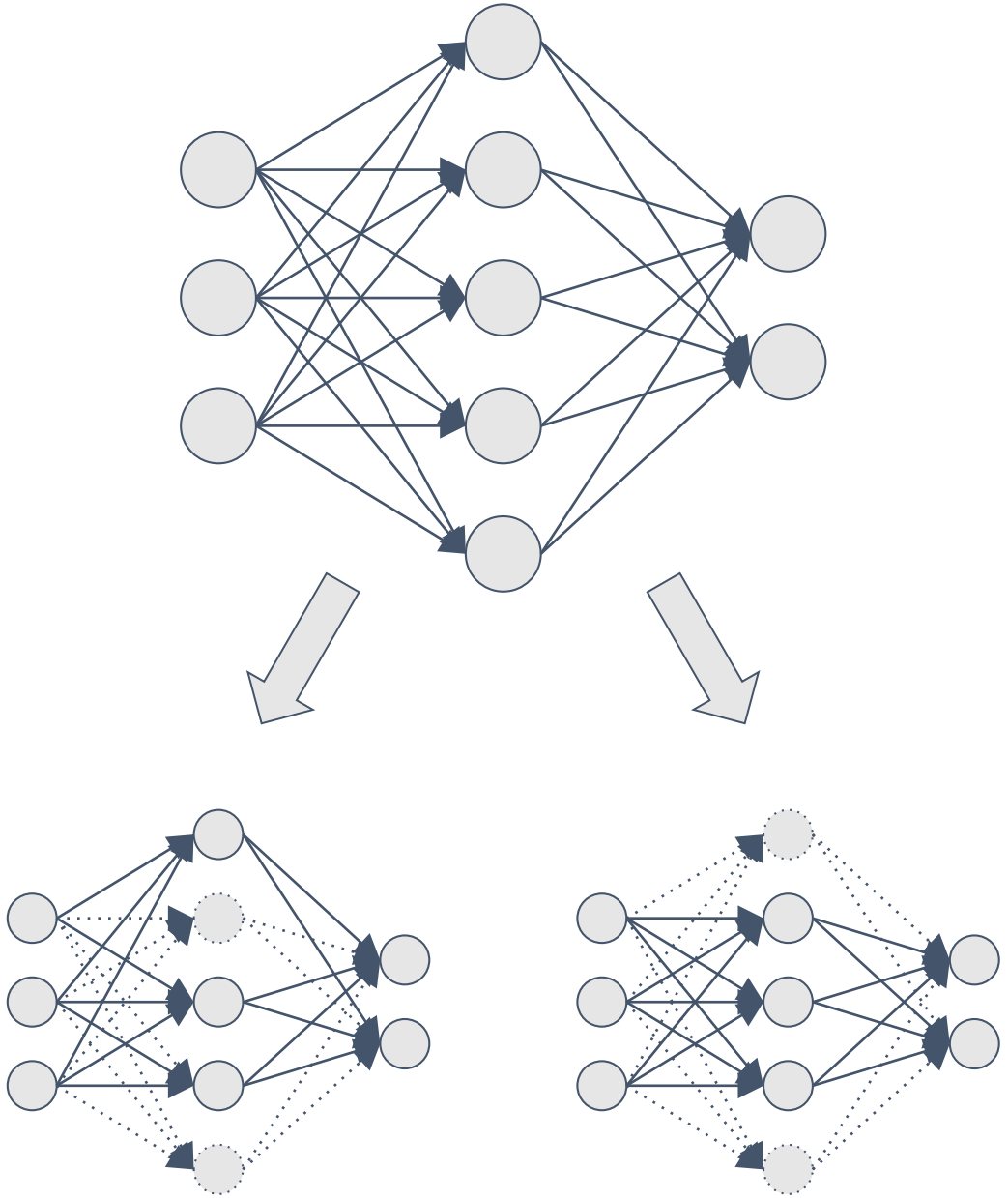
β = *penalty factor (real number)*

In other words, we force the network to prevent big disparity among the (absolute) value of the weights

Regularization - Dropout

- GOAL: prevent the neurons from co-adapting
- IDEA: randomly switch off the neurons (set the weights to zero) for each learning step on one training input.
- Probability of keeping a node is $p >= 0.5$, usually done at the last layers. At deeper and input layers, p should be much higher (and use noise instead of dropout)

Dropout simulates **different** networks that are trained to represent the same input data in many ways (**redundant**).



Normalization

- So far, we have talked a lot about techniques applied to the neural networks themselves, but we can also apply transformations to data.

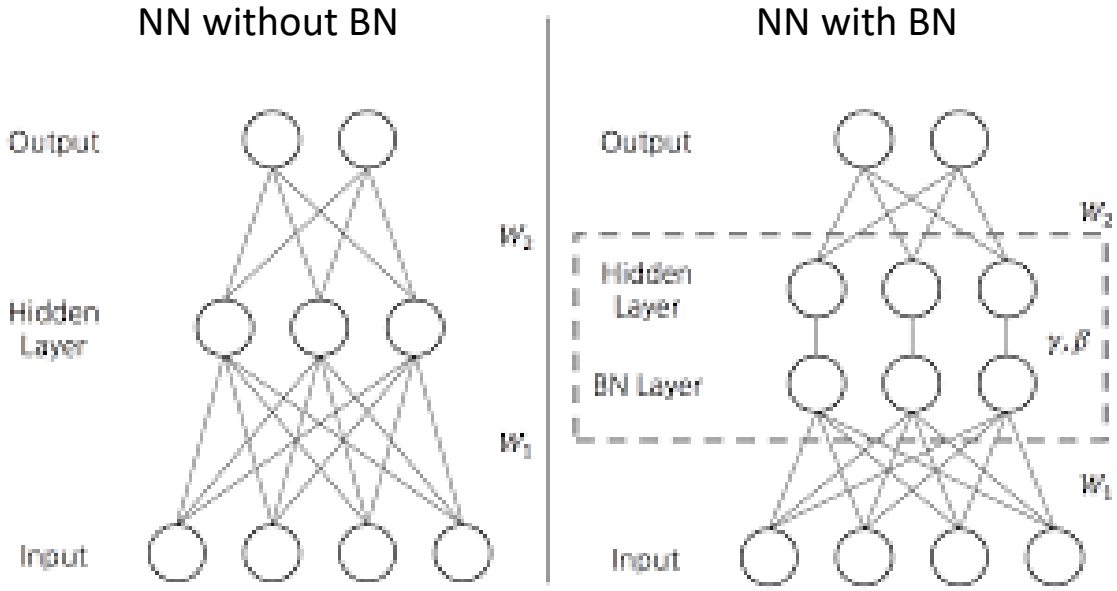
- Normalization is a procedure meant to scale the data points, typically in the range *[-1,1]* or *[0,1]*. This provides many benefits for neural networks, namely:
  - Scaled data provides numerical stability in the forward pass and reduces the chances of enormous weights and hence dominant features.
  - In the backpropagation step, this translates to a smoother and more spherical loss function surface, which makes the gradient descent procedure more stable.

- Common techniques used for normalization are :
  1. Min-Max normalization
  2. Z-score normalization (Standardization)
  3. **Batch Normalization (NN specific)**

Batch Normalization

- IDEA: Normalize the inputs to the hidden layers by the batch mean and variance. This reduces the effect of previous layers into later ones.

- Extremely helpful (and almost necessary) in very deep architectures.



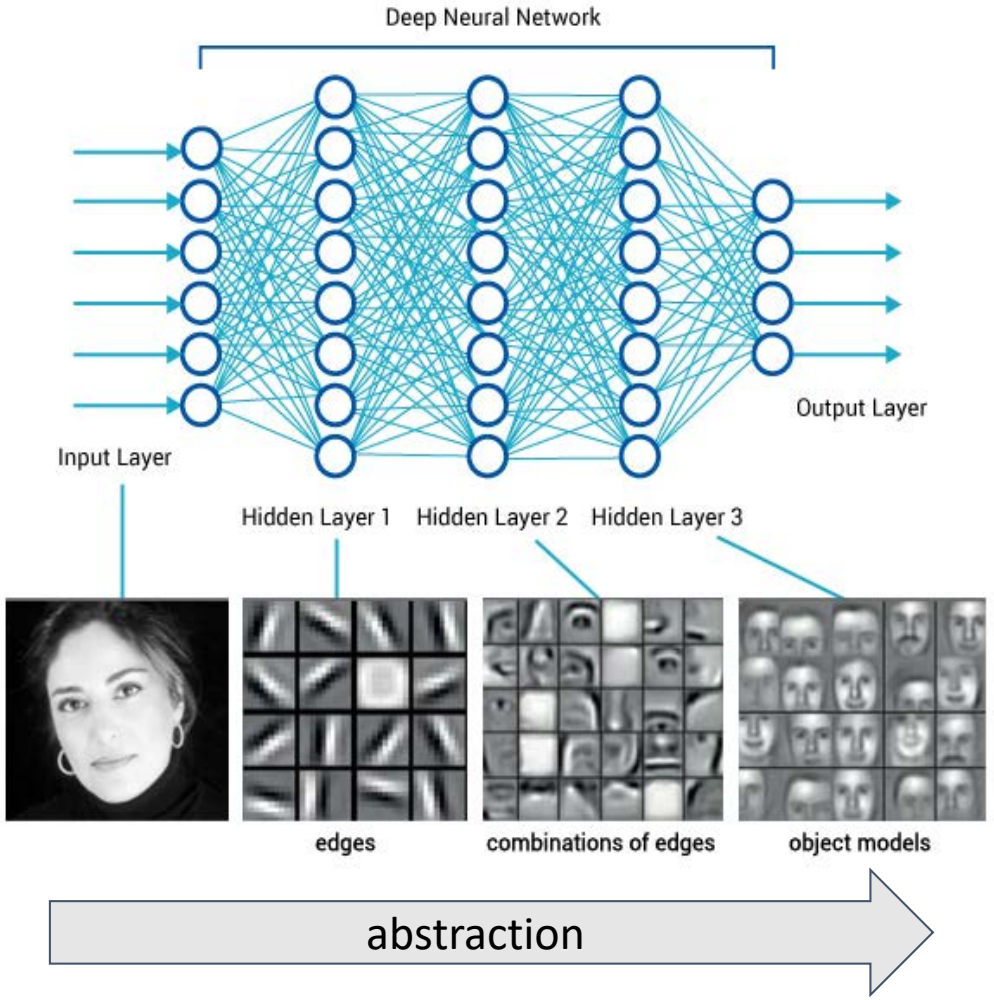
Summary: Deep Neural Networks

Pros:

- **parameter efficiency**: much more performance
- **hierarchical structure**: a lot of natural phenomena could be represented since each layer describes a specific level of abstraction of the input

Cons:

- needs a **huge amount of data**
- needs **regularization**
- **In the case of fully connected architectures, tons of params to tune**



Specialized architectures

- Our brain contains other specialized structures which are used to perceive the environment. Such an example is the visual cortex, where individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field.

- There is a class of neural networks which is loosely inspired by this structure, called **Convolutional Neural Networks** or **CNNs** or **ConvNets**.

- They are architectures with shared weights coming from so-called convolution kernels that shift over input features.

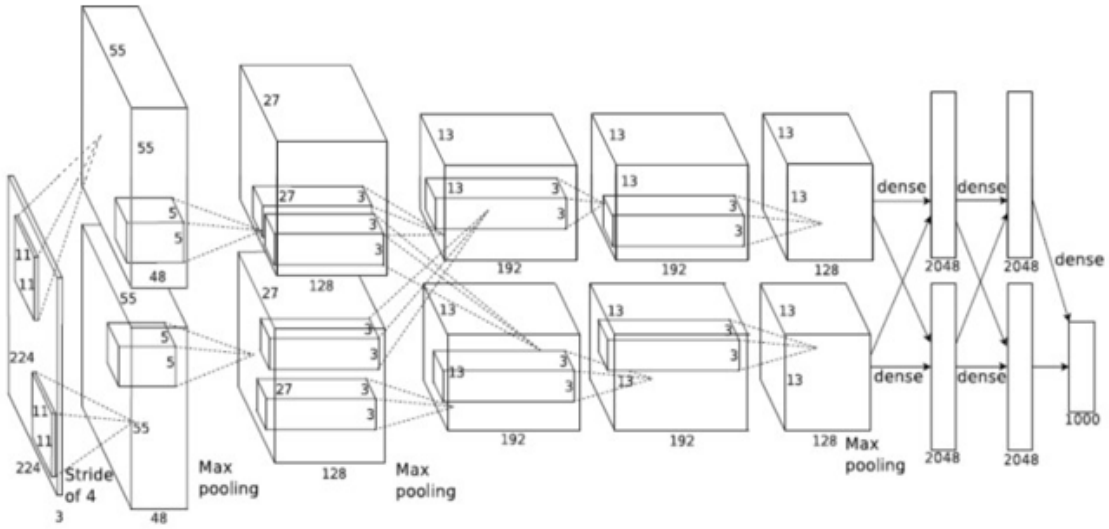
Convolutional Neural Networks

GOAL: Build a space or time invariant model

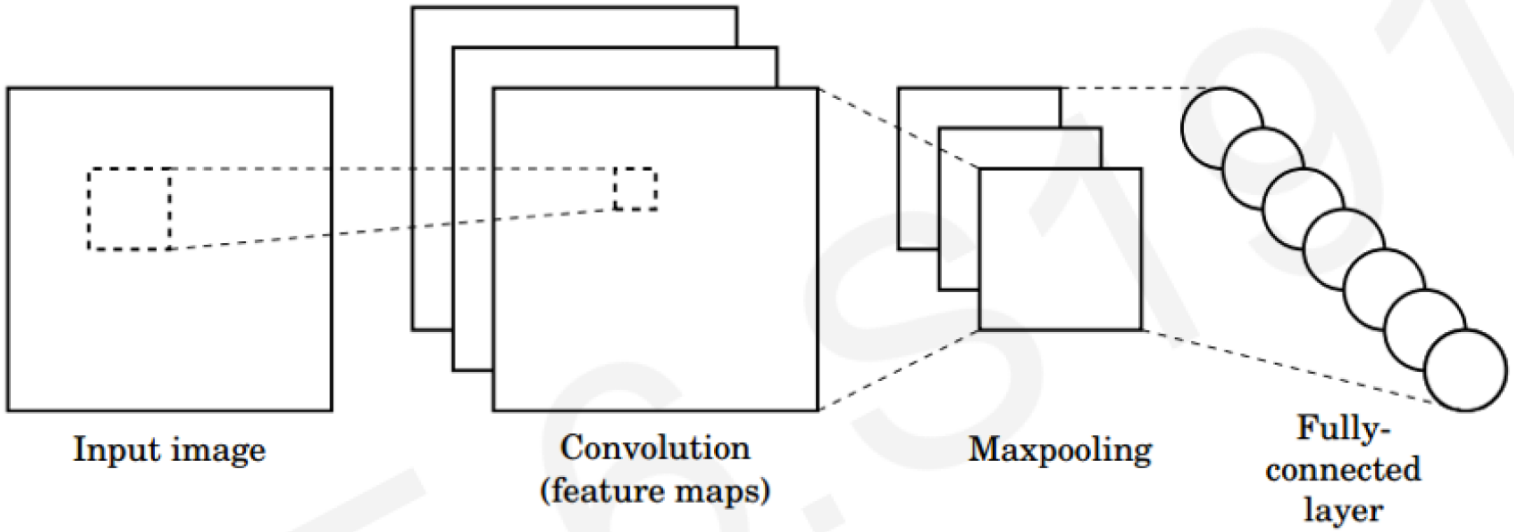
IDEA: Use the same set of weights over different parts of the data, by exploiting the known type of the inputs

weight sharing

Share and train the weights jointly
for those inputs that contain
the same information



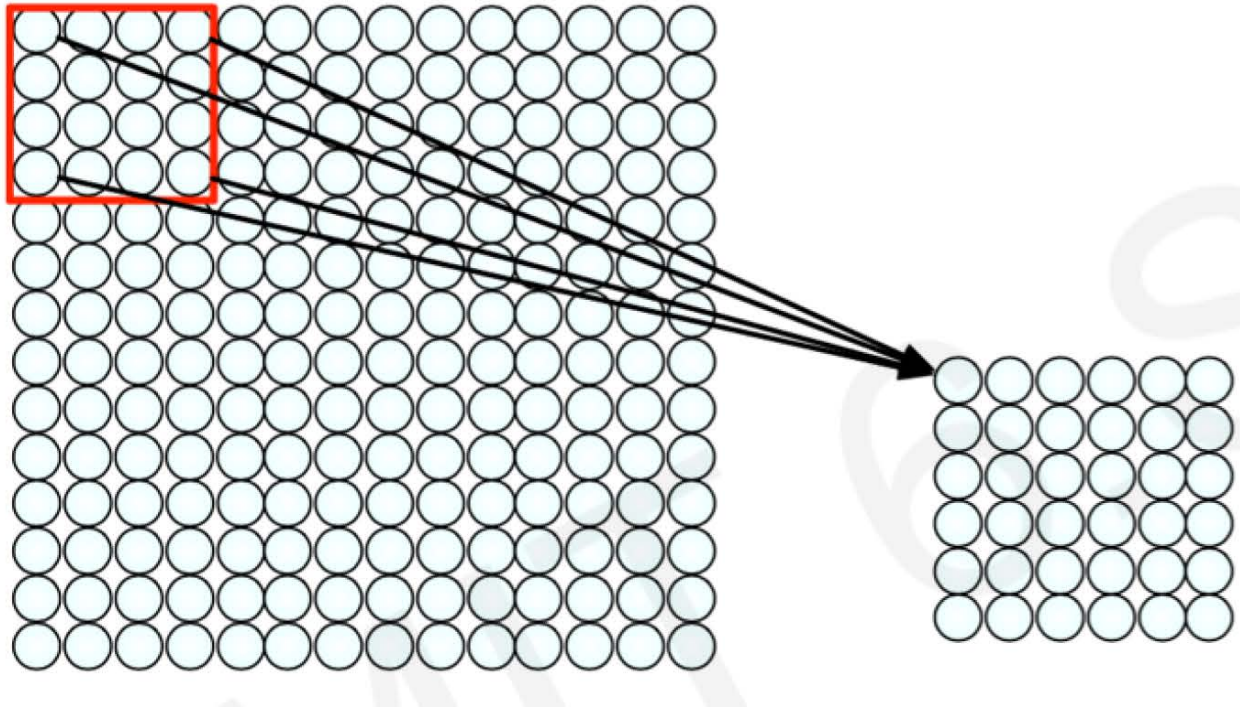
Convolutional Neural Networks



1. **Convolution**: Apply filters to generate feature maps.
2. **Non-linearity**: Often ReLU.
3. **Pooling**: Downsampling operation on each feature map.

Train model with image data.
Learn weights of filters in convolutional layers.

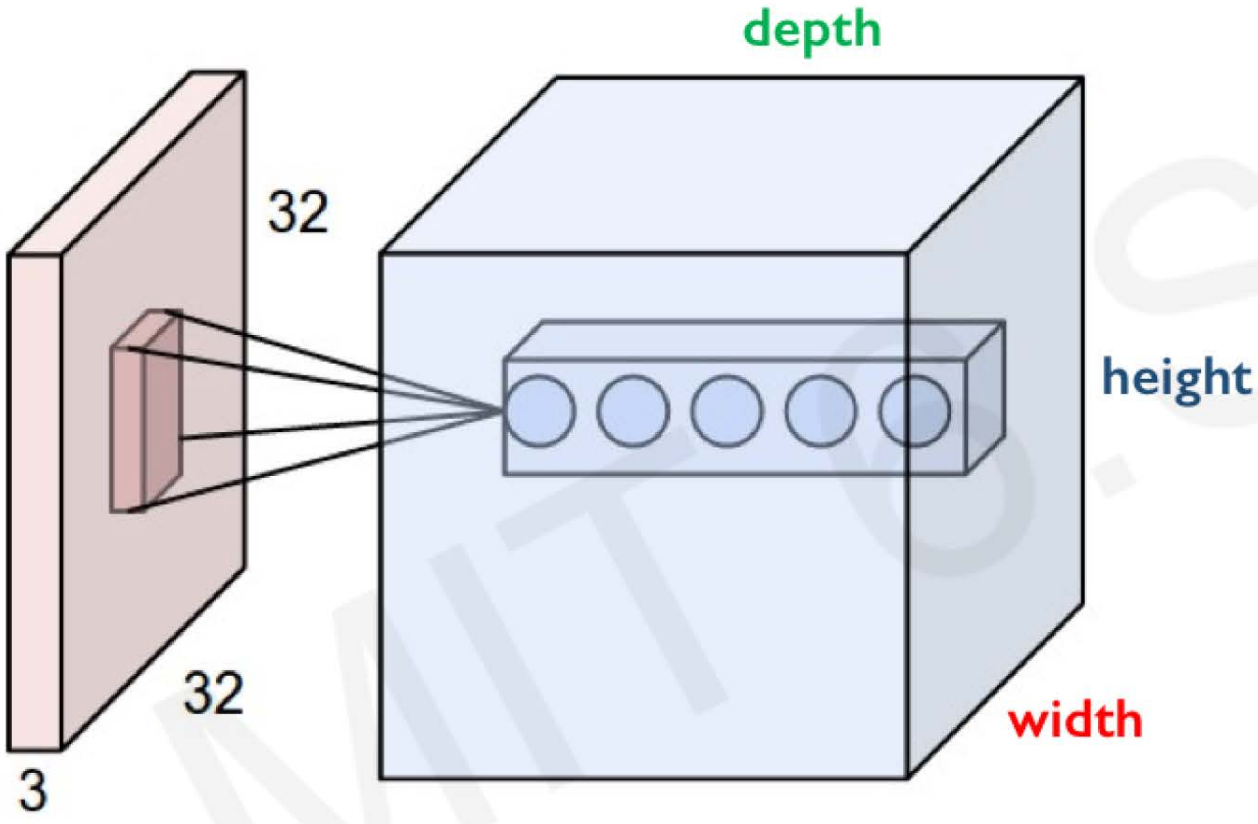
Convolutional Layer: local connectivity



For a neuron in hidden layer:

- Take inputs from patch
- Compute weighted sum
- Apply bias

CNN: spatial arrangement of the output volume



Layer Dimensions:

$h \; x \; w \; x \; d$

where h and w are spatial dimensions
d (depth) = number of filters

Stride:

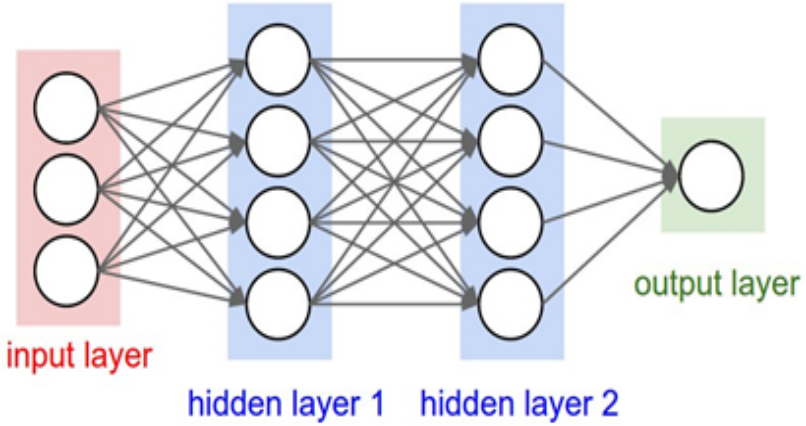
Filter step size

Receptive Field:

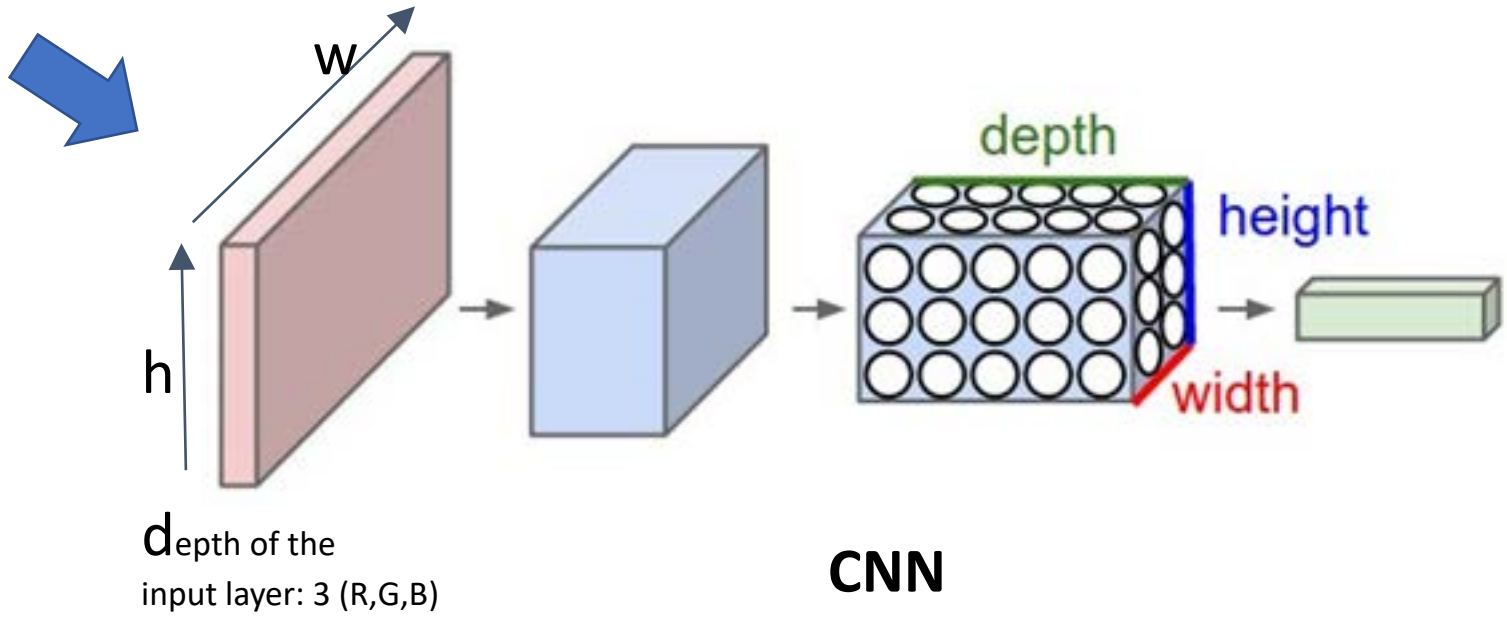
Locations in input image that
a node is path connected to

Convolutional Neural Networks

- CNN have neurons **arranged in 3 dimensions:** width, height, *depth (depth* ≠ depth of the network)
- A **layer** transforms an input 3D volume to an output 3D volume.



NN

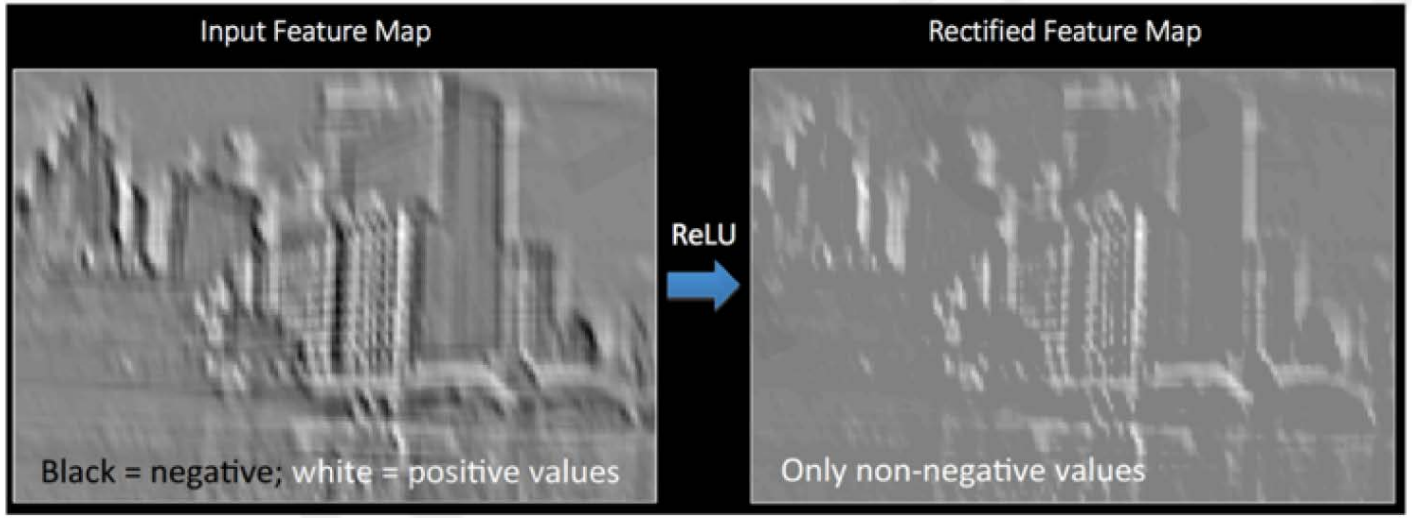


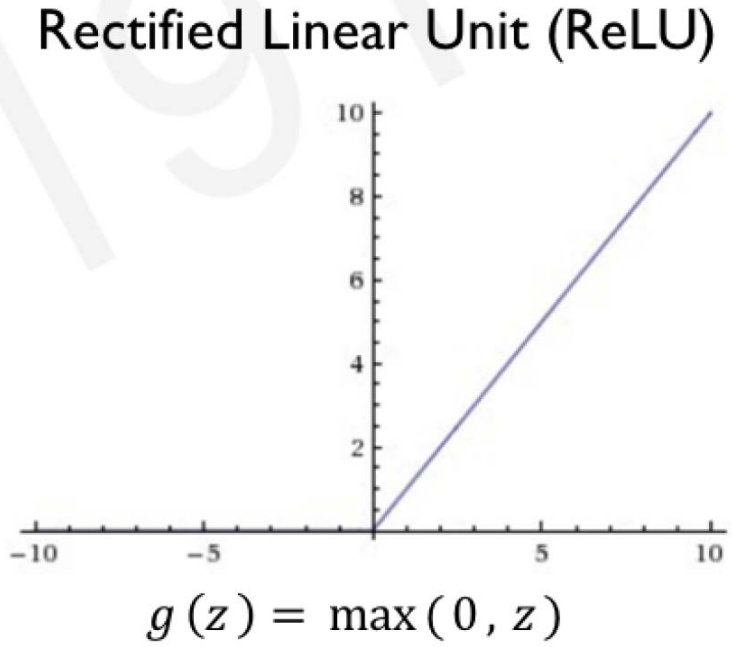
depth of the input layer: 3 (R,G,B)

CNN

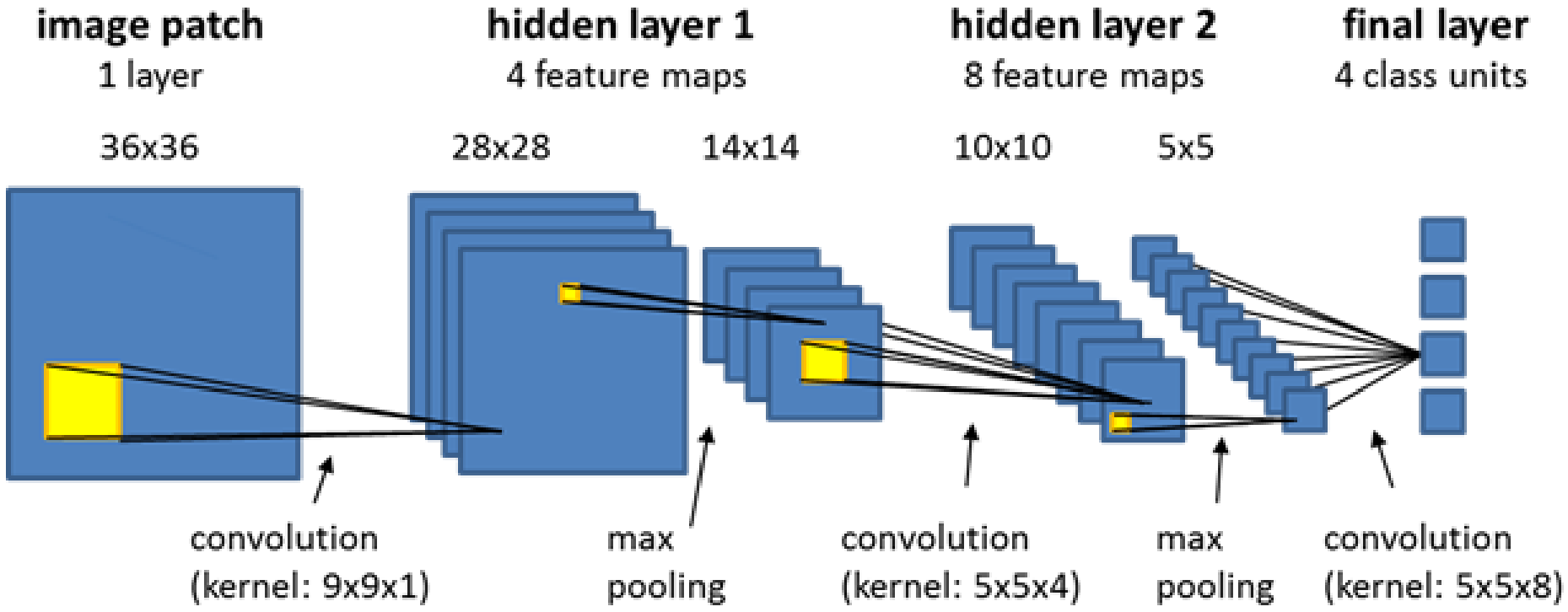
CNN: introducing non-linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**





Convolutional Neural Networks - Example



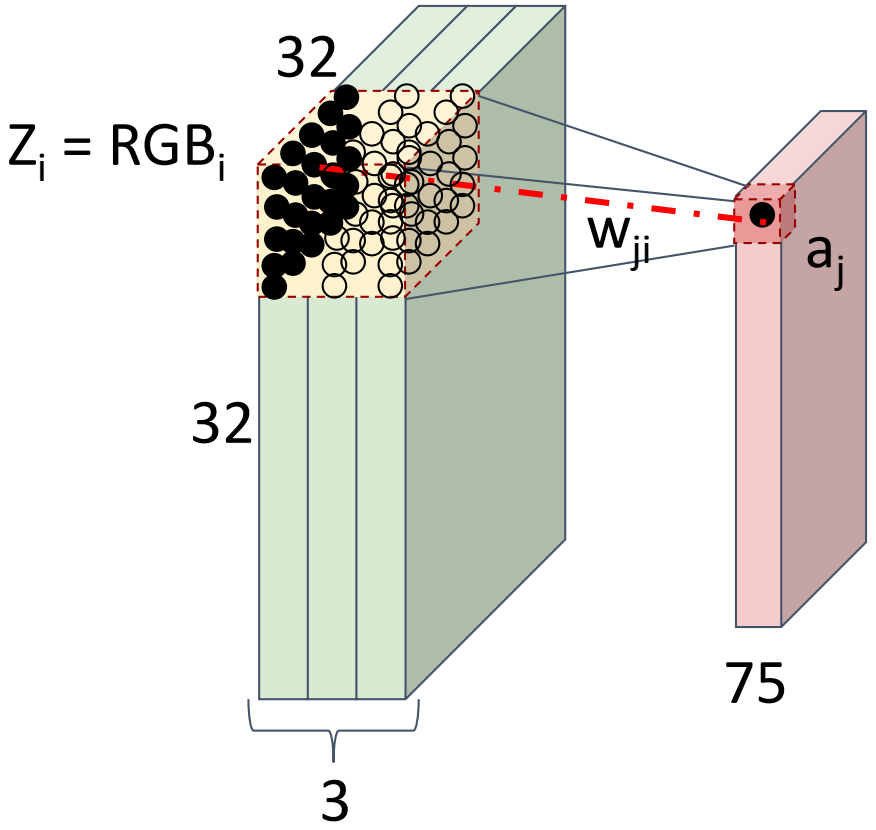
Unique layers of a CNN

- As you can see in the previous example CNN layer may be of 5 different kinds, 2 of them are CNN specific (in **bold**)

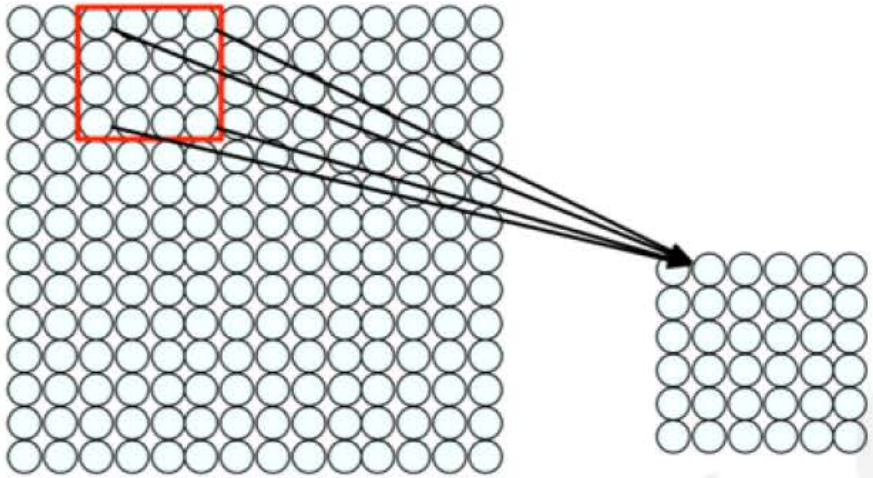
- Input Layer, **Convolutional Layer**, Activation (RELU) Layer, **Pooling Layer**, and Fully-Connected Layer.

Convolutional layer (CONV)

- The reason why we are talking about CNN.
- Highlights low-level - poorly semantic - highly perceptually salient patterns in an economic way (in a parameter cardinality sense).
- The great idea is that **all the weights across one layer are shared.**
- In practice, the output from a CONV slice is like it was processed by an image filter.



Feature extraction with convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

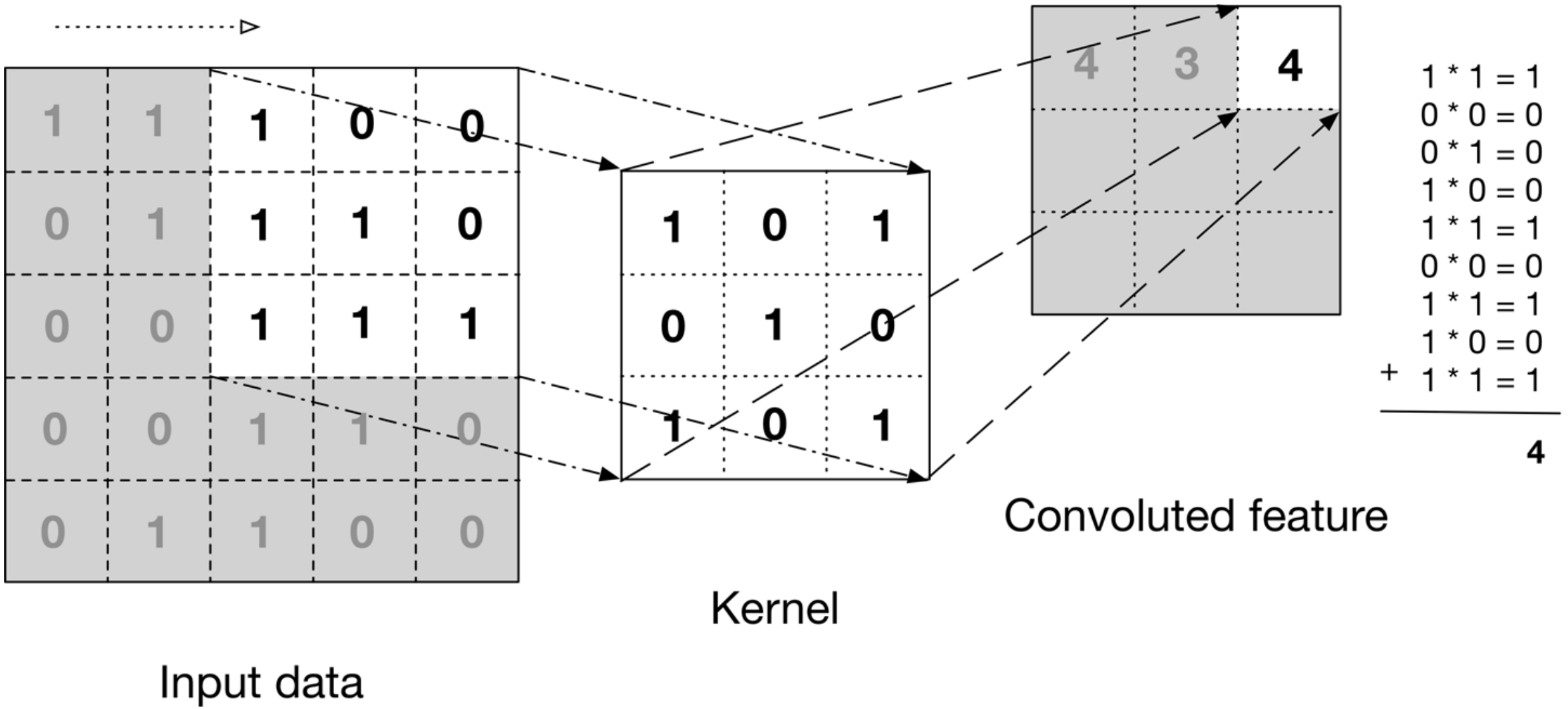
This "patchy" operation is **convolution**

1) Apply a set of weights – a filter – to extract **local features**

2) Use **multiple filters** to extract different features

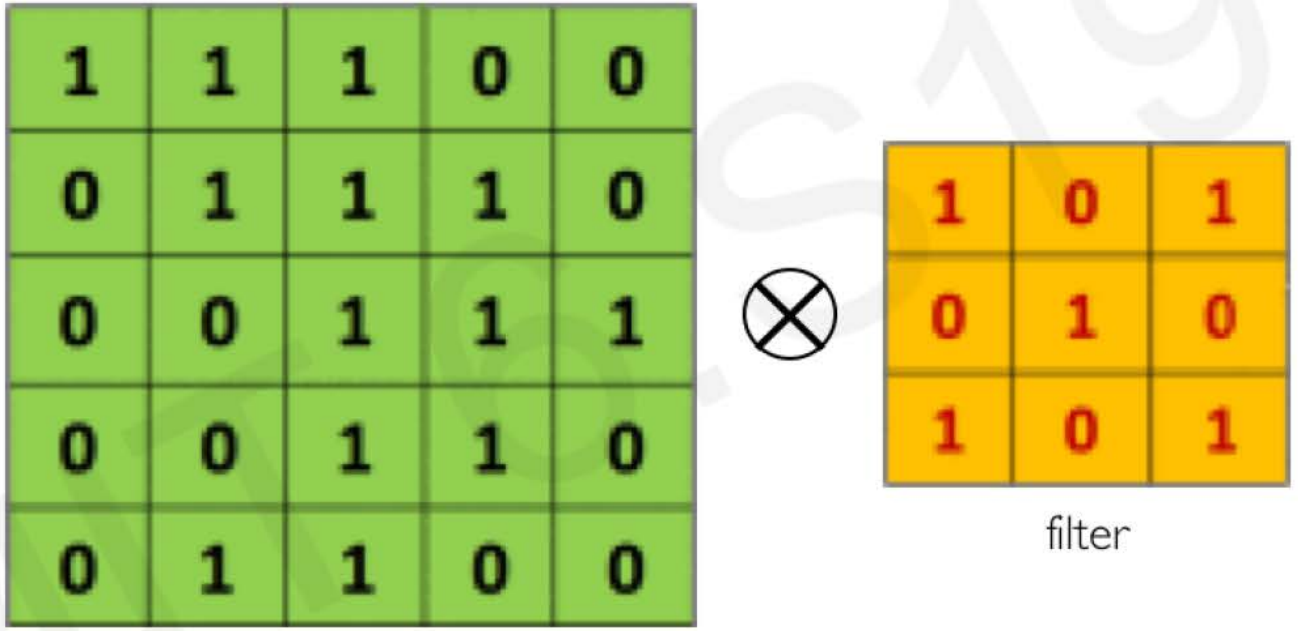
3) **Spatially share** parameters of each filter

Convolutional layer (CONV) - Convolution



The Convolution operation

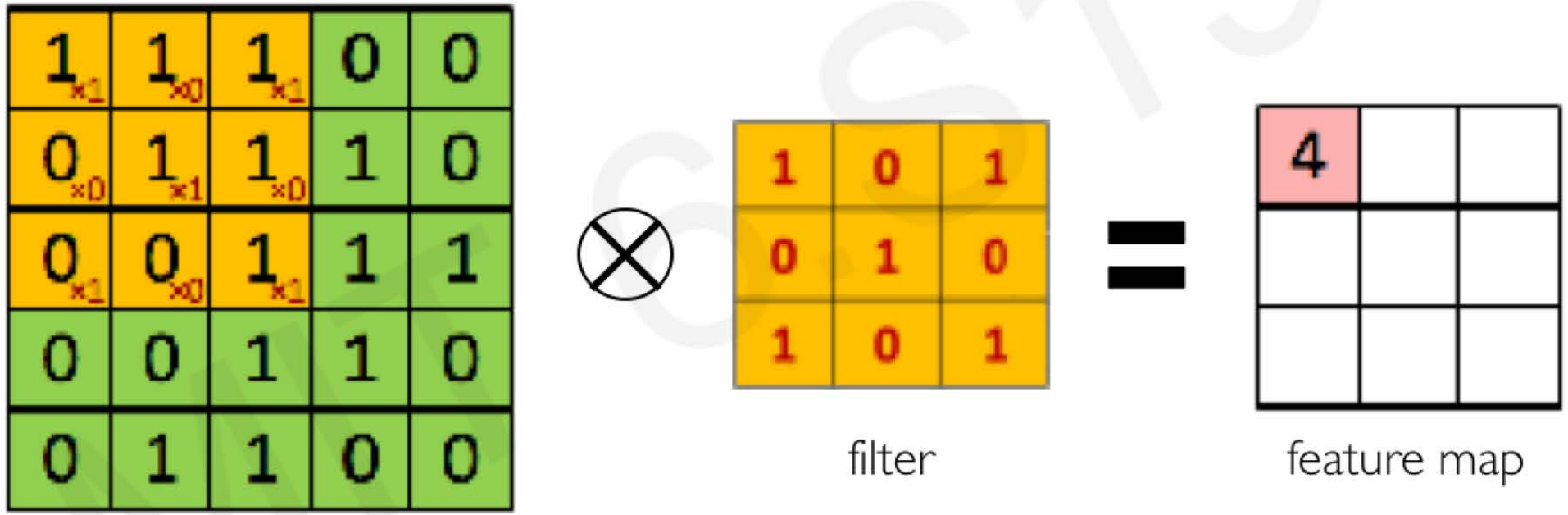
Suppose we want to compute the convolution of a 5x5 image and a 3x3 filter:



We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs…

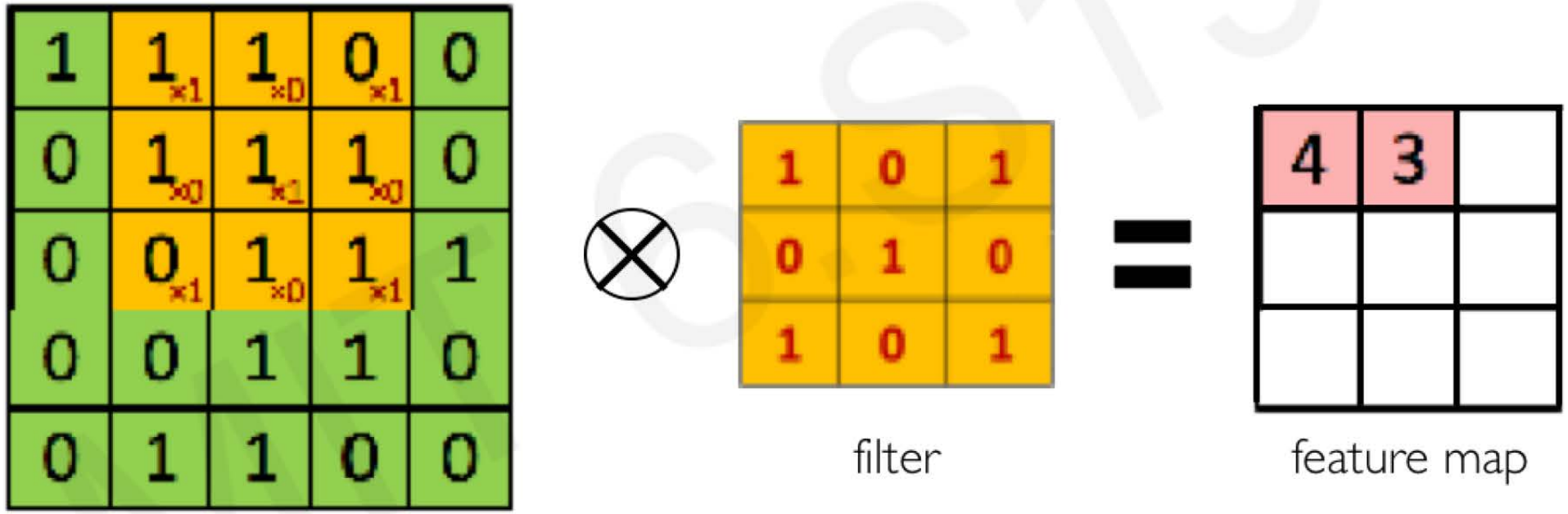
The Convolution operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



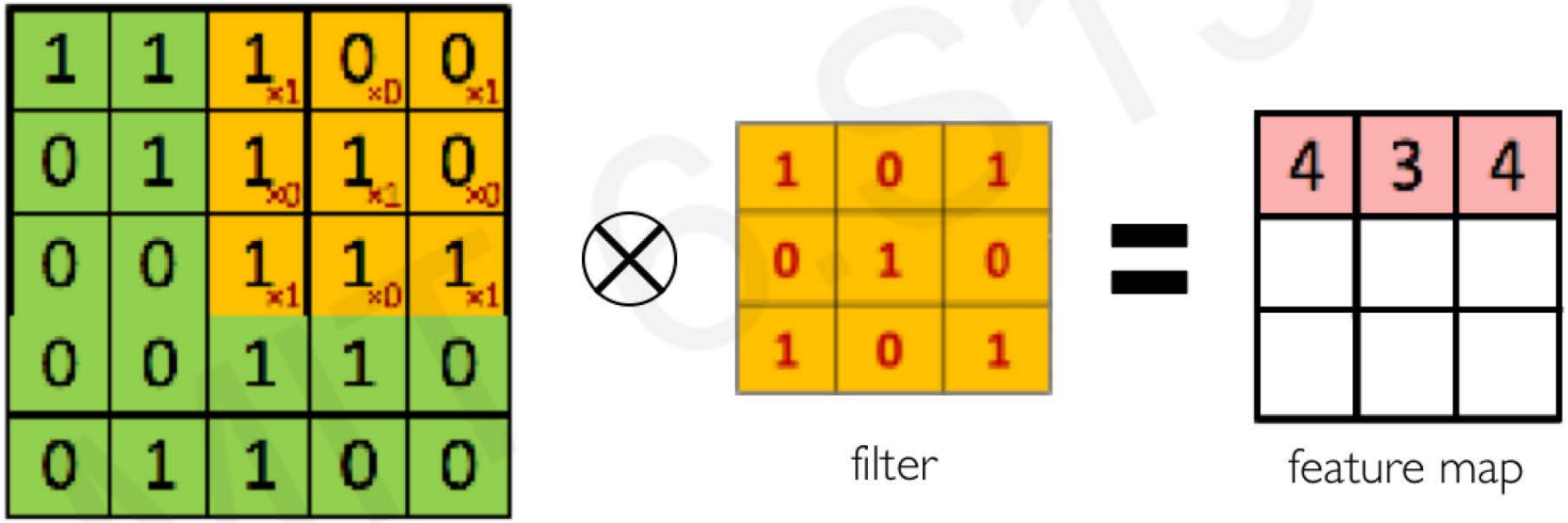
The Convolution operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



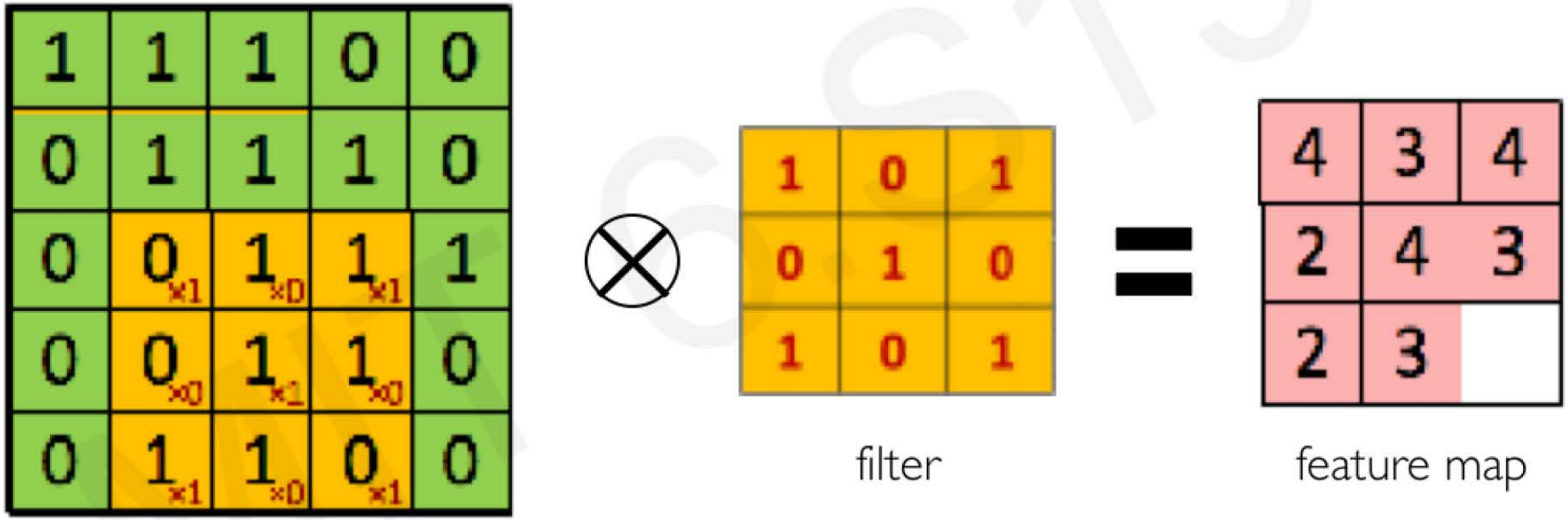
The Convolution operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



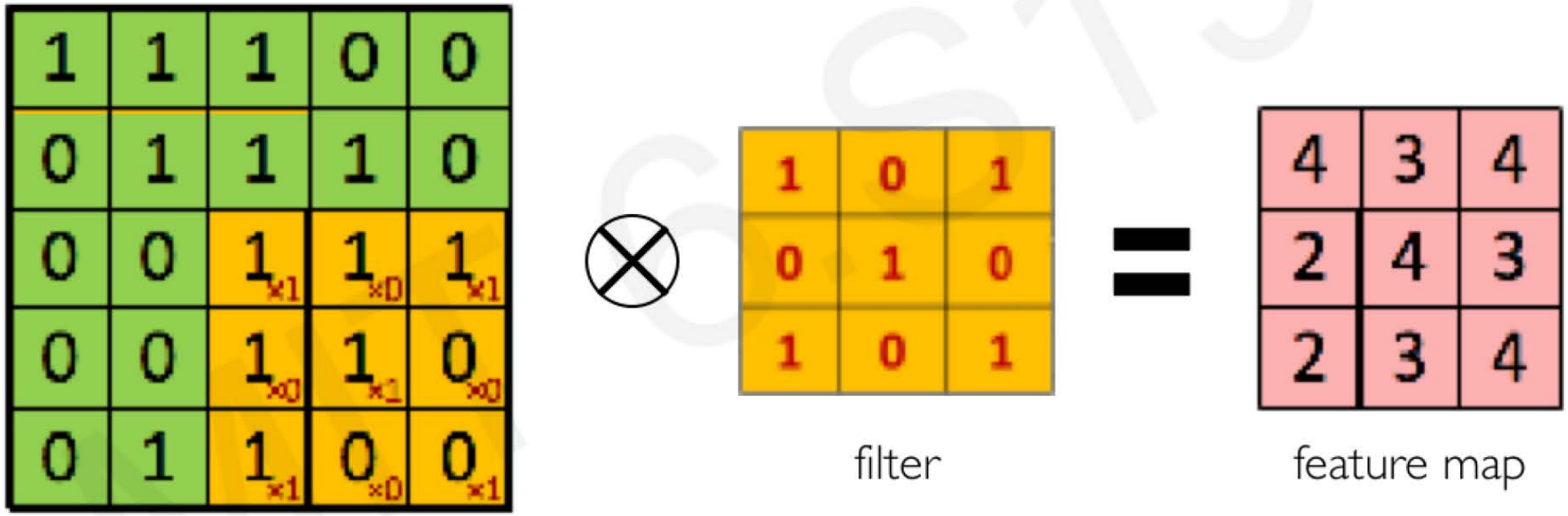
The Convolution operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



The Convolution operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



Producing Feature Maps

Original

Sharpen

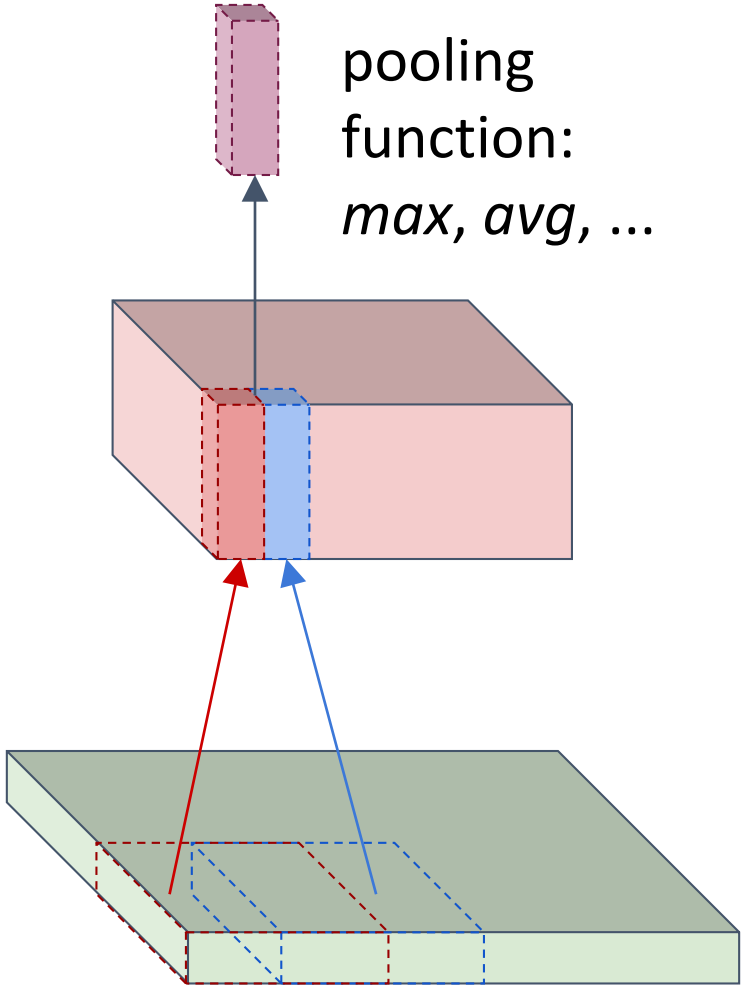
Edge Detect

"Strong" Edge Detect

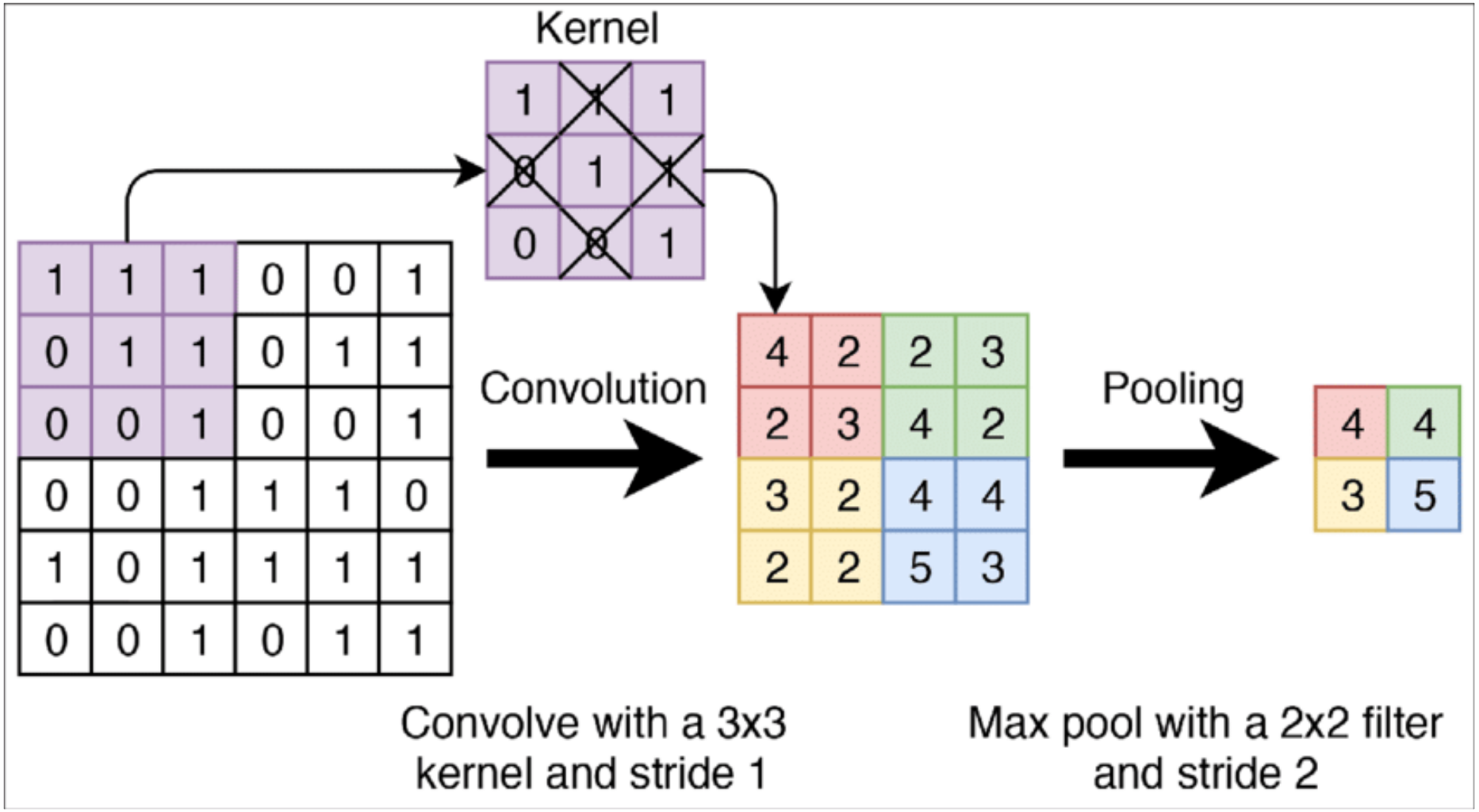
Pooling layer

- Common to periodically insert a Pooling layer in-between successive Conv layers

- Reduces:
  - the spatial size of the representation
  - the amount of parameters and computation, controlling the overfitting.



Pooling layer



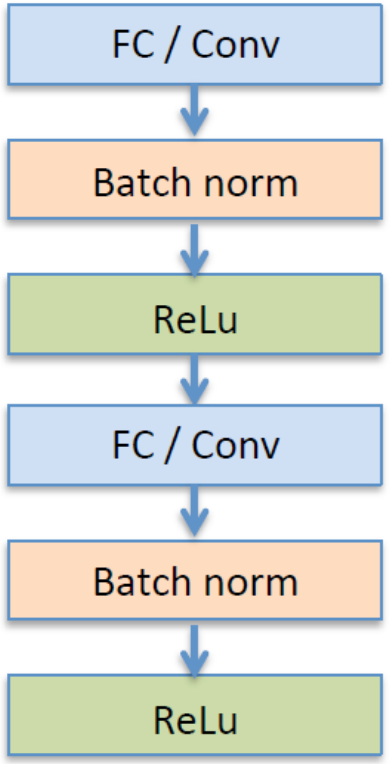
Batch-norm layer

- As learning progresses, the distribution of the layer inputs changes due to parameter updates (internal covariate shift)
- This can result in most inputs being in the non-linear regime of the activation function, slowing down learning
- Bach normalization is a technique to reduce this effect
  - Explicitly force the layer activations to have zero mean and unit variance w.r.t running batch estimates

$$\hat{x}^{(k)} = \frac{x^{(k)} - E(x^{(k)})}{\sqrt{\text{var}(x^{(k)})}}$$

  - Adds a learnable scale and bias term to allow the network to still use the nonlinearity

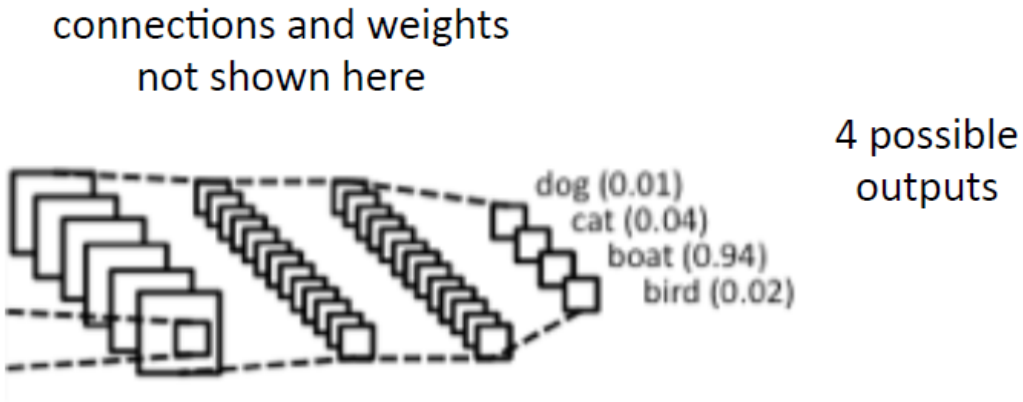
$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$



S. Ioffe & C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." ICML 2015.

Fully connected layer

- In the end it is common to add one or more fully (or densely) connected layers.
- Every neuron in the previous layer is connected to every neuron in the next layer (as in regular neural networks). Activation is computed as matrix multiplication plus bias
- At the output, softmax activation for classification



Softmax

- A special kind of activation layer, usually at the end of FC layer outputs
- Can be viewed as a fancy normalizer (a.k.a. Normalized exponential function)
- Produce a discrete probability distribution vector
- Very convenient when combined with cross-entropy loss

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

Given sample vector input **x** and weight vectors $\{\mathbf{w}_i\}$, the predicted probability of y = j

To wrap up, common CNN building blocks

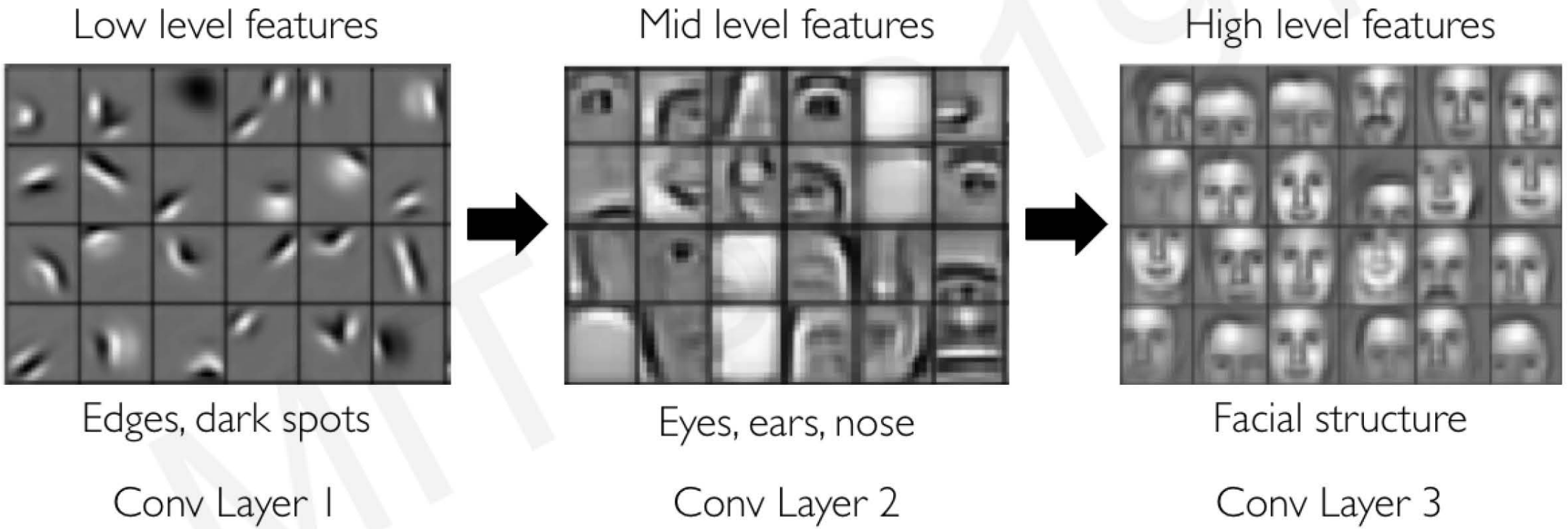
- The most common form of a CNN architecture stacks
  - a few **CONV-RELU** layers,
  - follows them with **POOL** layers, and
  - repeats this pattern until the image has been merged spatially to a small size
  - At some point, it is common to transition to **fully-connected (FC)** layers
  - The last fully-connected layer holds the output, such as the **class scores**

$$\text{INPUT} \rightarrow [[\text{CONV} \rightarrow \text{RELU}] \times N \rightarrow \text{POOL?}] \times M \rightarrow [\text{FC} \rightarrow \text{RELU}] \times K \rightarrow \text{FC}$$

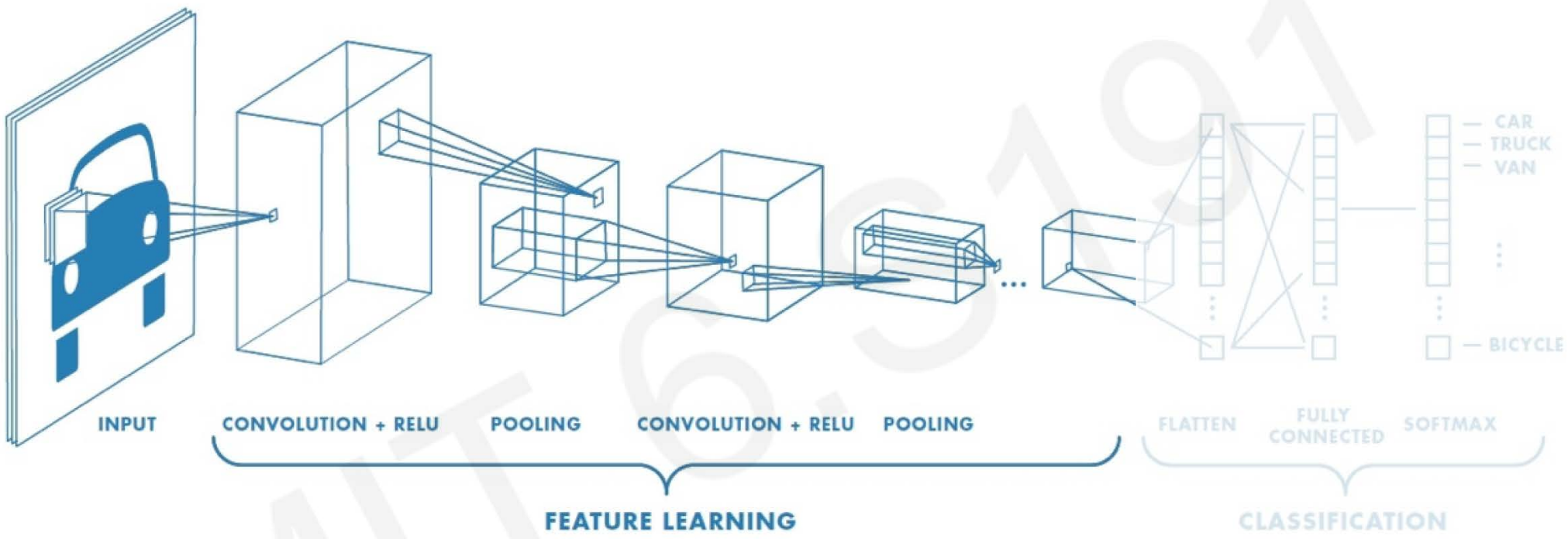
where the $\times$ indicates repetition, and the POOL? indicates an optional pooling layer

- Moreover, $N >= 0$ (and usually $N <= 3$), $M >= 0$, $K >= 0$ (and usually $K < 3$).

Representation Learning in Deep CNN

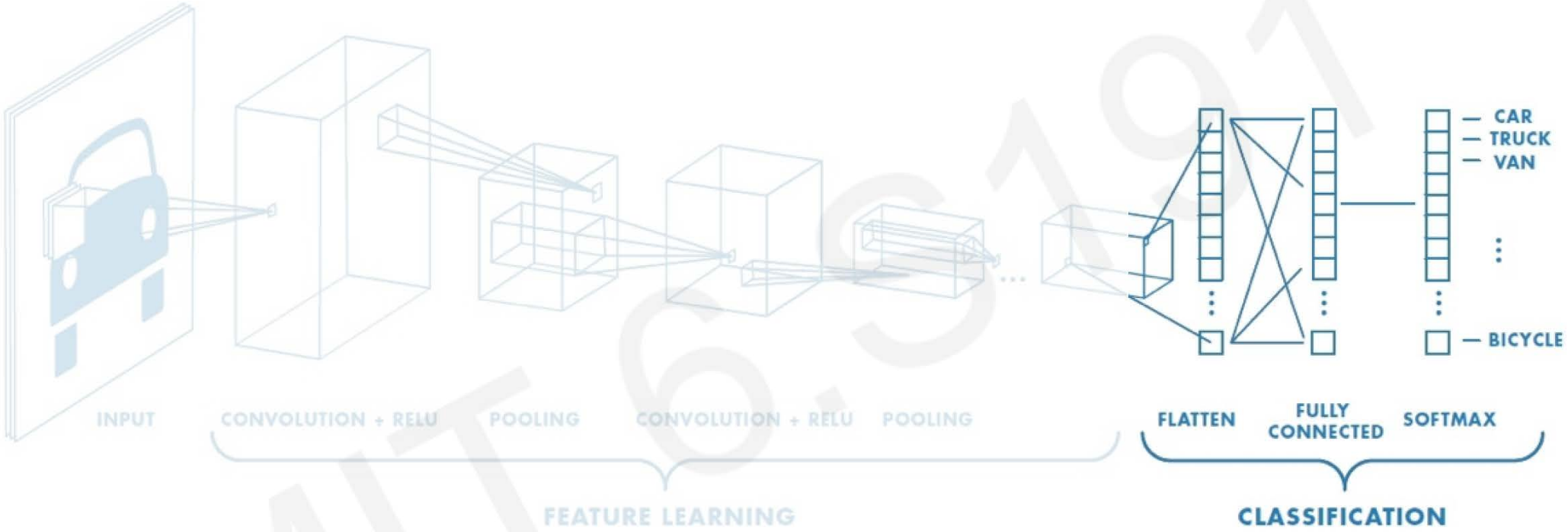


CNN for classification: Feature Learning



1. Learn features in input image through **convolution**
2. Introduce **non-linearity** through activation function (real-world data is non-linear!)
3. Reduce dimensionality and preserve spatial invariance with **pooling**

CNN for classification: Class Probabilities

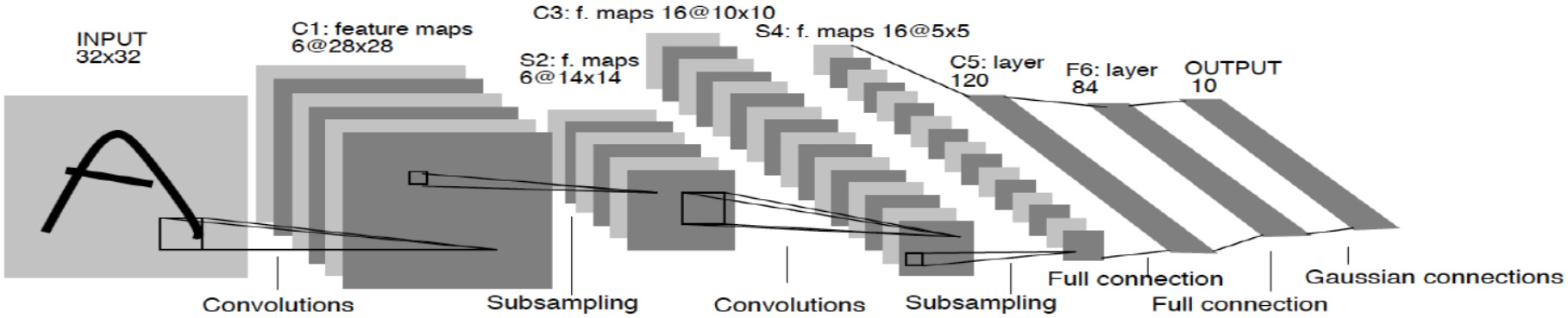


- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Famous CNN architectures

- **LeNet**. The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's. Used to read zip codes, digits, etc.



MNIST digit classification problem
handwritten digits
60,000 training examples
10,000 test samples
10 classes
28x28 grayscale images

Conv filters were 5x5, applied at stride 1
Sigmoid or tanh nonlinearity
Subsampling (average pooling) layers were 2x2 applied at stride 2
Fully connected layers at the end
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

Y. LeCun et al. "Backpropagation applied to handwritten zip code recognition." *Neural Computation*, 1989.

A disruptive benchmark

ImageNet: ILSVRC

IMAGENET

Large Scale Visual Recognition Challenge

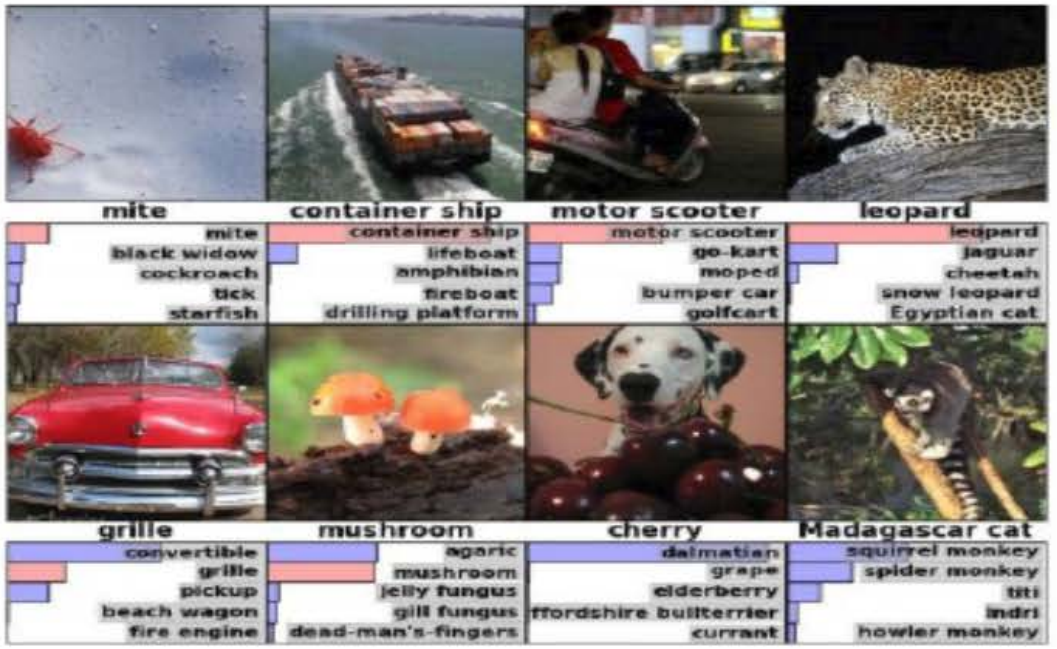
Image Classification

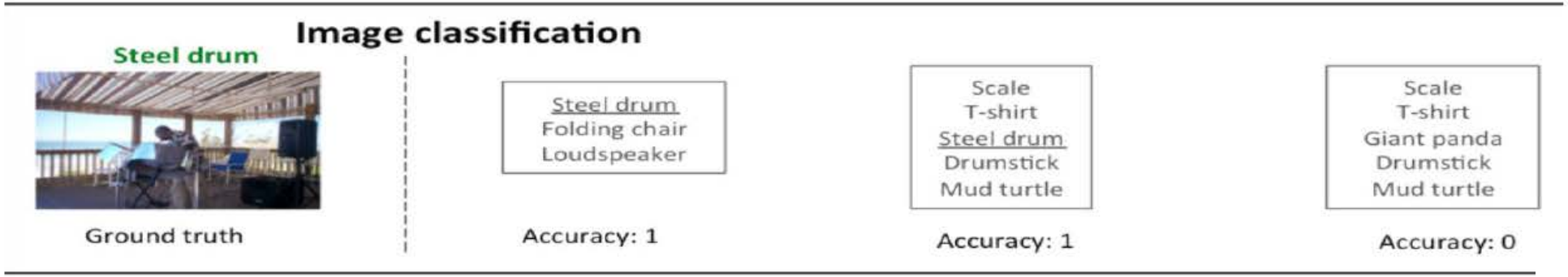
1000 object classes (categories)

Images:

- 1.2 M train
- 100k test.

Metric: top 5 error rate (predict 5 classes)

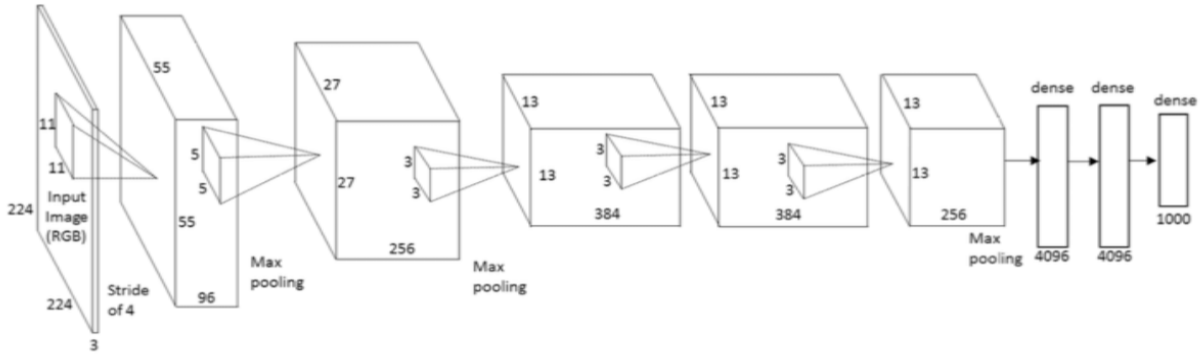




www.image-net.org/challenges/LSVRC/

Famous CNN architectures

- **AlexNet**. Developed by Alex Krizhevsky et al. was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). AlexNet had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).

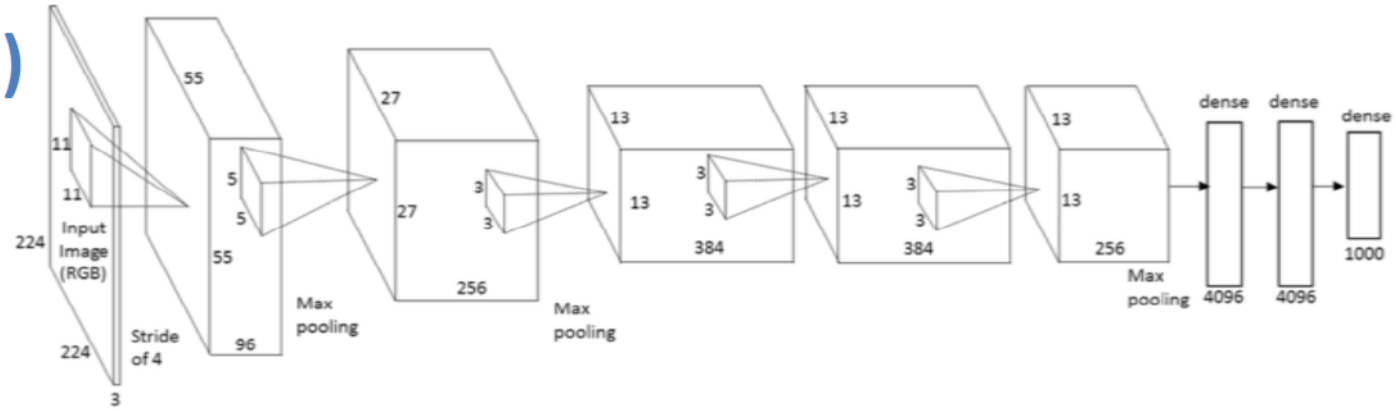


- Similar framework to LeNet:
  - 8 parameter layers (5 convolutional, 3 fully connected)
  - Max pooling, ReLu nonlinearities
  - 650,000 units, 60 million parameters)
  - trained on two GPUs for a week
  - Dropout regularization

Alex Krizhevsky et al. "ImageNet classification with deep convolutional neural networks." *Neural Information Processing Systems NIPS 2012.*

Famous CNN architectures

AlexNet (2012)



Full AlexNet architecture:
8 layers
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

ILSVRC 2012 winner
7 CNN ensemble: 18.2% -> 15.4%

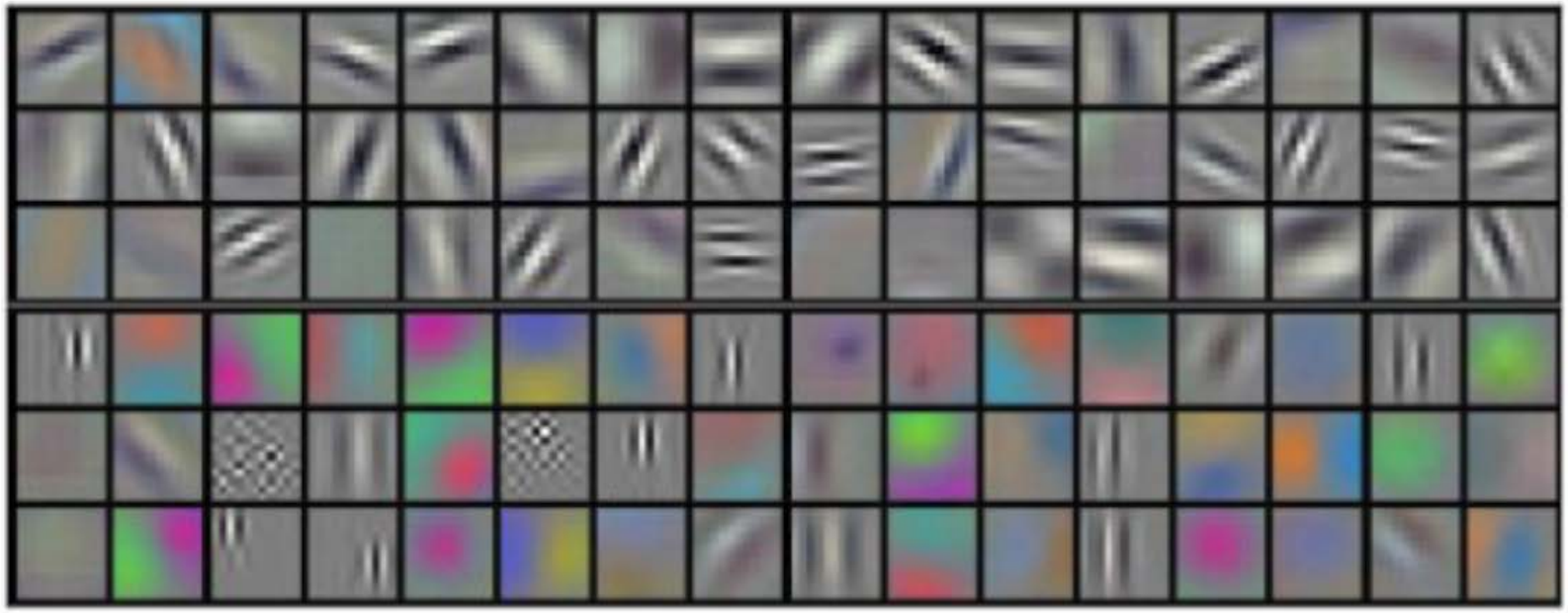
Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4

Alex Krizhevsky et al. "ImageNet classification with deep convolutional neural networks." *Neural Information Processing Systems NIPS 2012.*

Famous CNN architectures

AlexNet (2012)

- Visualization of the 96 11x11 filters learned by the first layer

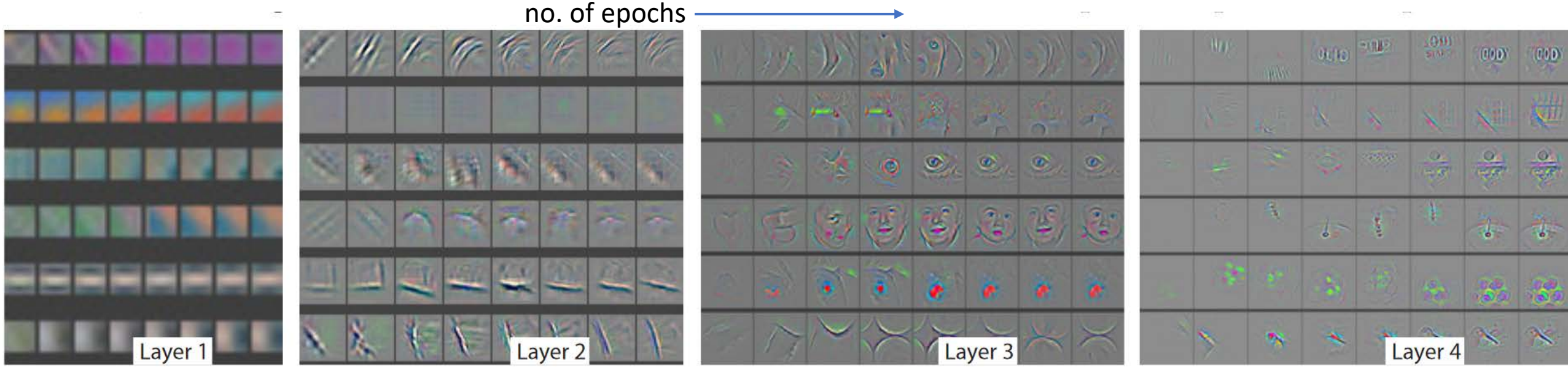


Alex Krizhevsky et al. "ImageNet classification with deep convolutional neural networks." *Neural Information Processing Systems NIPS 2012.*

Famous CNN architectures

- **ZF Net**. The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZF Net (short for Zeiler & Fergus Net).

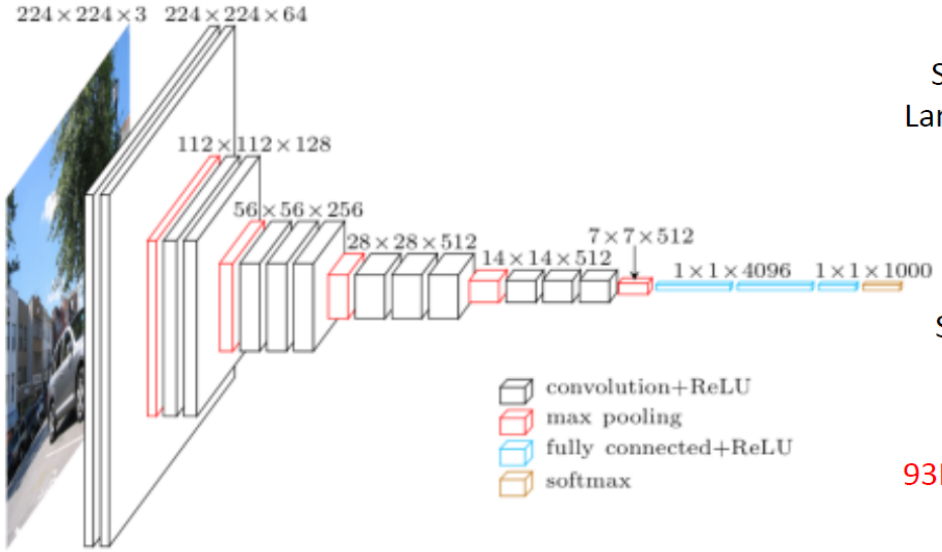
- It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.



D. Zeiler & R. Fergus. "Visualizing and Understanding Convolutional Networks." *European Conf. on Computer Vision ECCV 2014.*

Famous CNN architectures

- **VGGNet**. Runner-up in ILSVRC 2014, its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Downside, is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer



Seq. of deeper nets trained progressively
Large receptive fields replaced by 3x3 conv

Only:
3x3 CONV stride 1, pad 1 and
2x2 MAX POOL stride 2

Shows that depth is a critical component for good performance

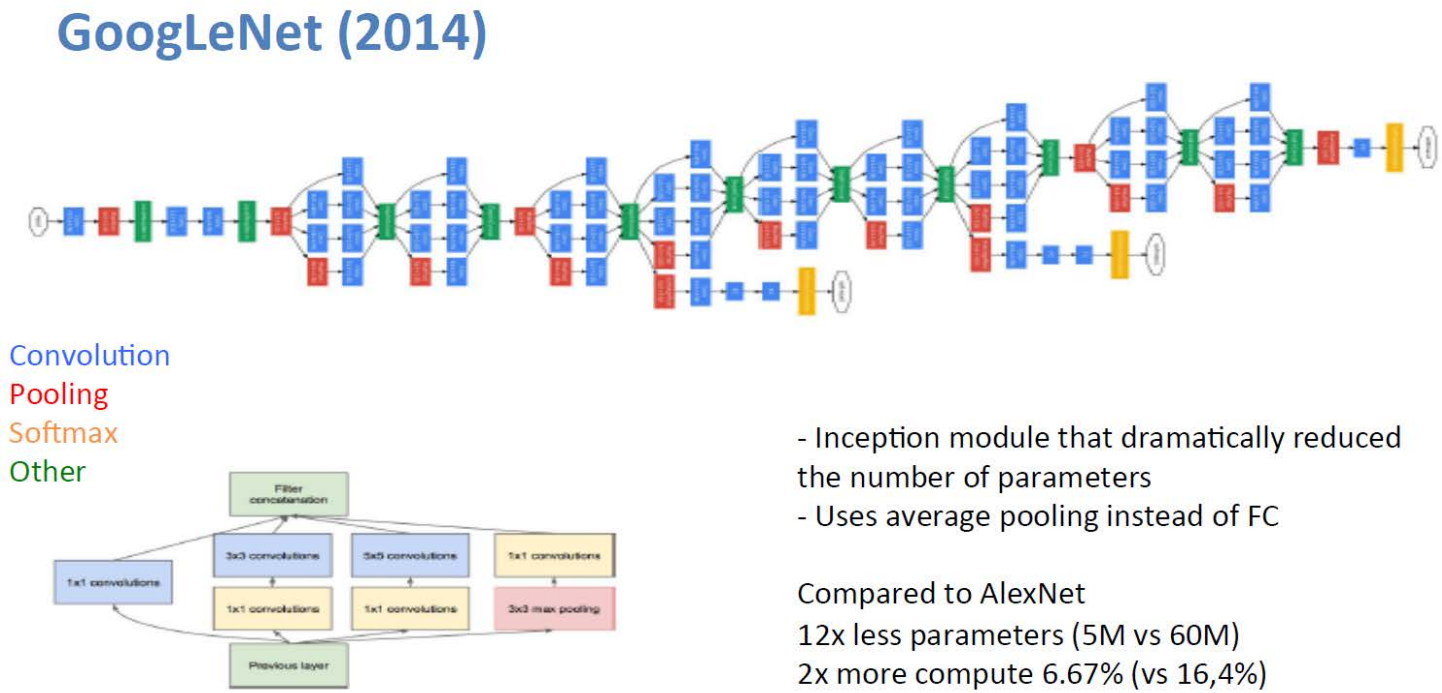
TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
most memory is in the early CONV

TOTAL params: 138M parameters
most parameters are in late FC

K. Simonyan & A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *Int'l Conf. on Learning Representations ICLR 2015*.

Famous CNN architectures

- **GoogLeNet**. The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. There are also several followup versions to the GoogLeNet, most recently Inception-v4.

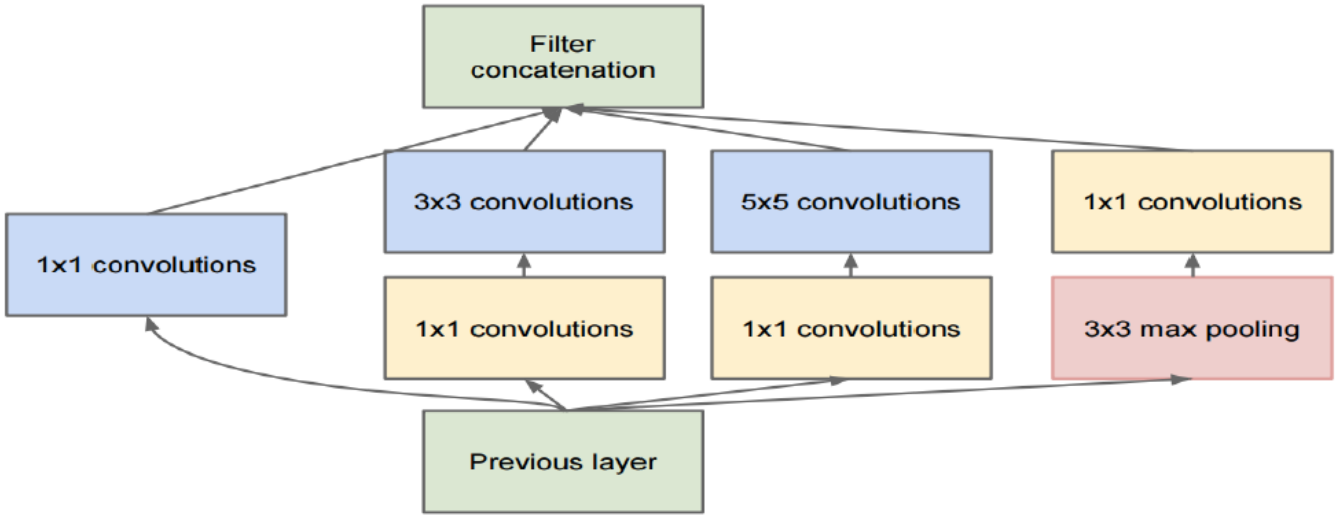


C.Szegedy et al. “Going Deeper with Convolutions.”, *CVPR 2015*.

Famous CNN architectures

GoogLeNet (2014)

- The Inception Module
  - Parallel paths with different receptive field sizes and operations are meant to capture sparse patterns of correlations in the stack of feature maps
  - Use 1x1 convolutions for dimensionality reduction before expensive convolutions

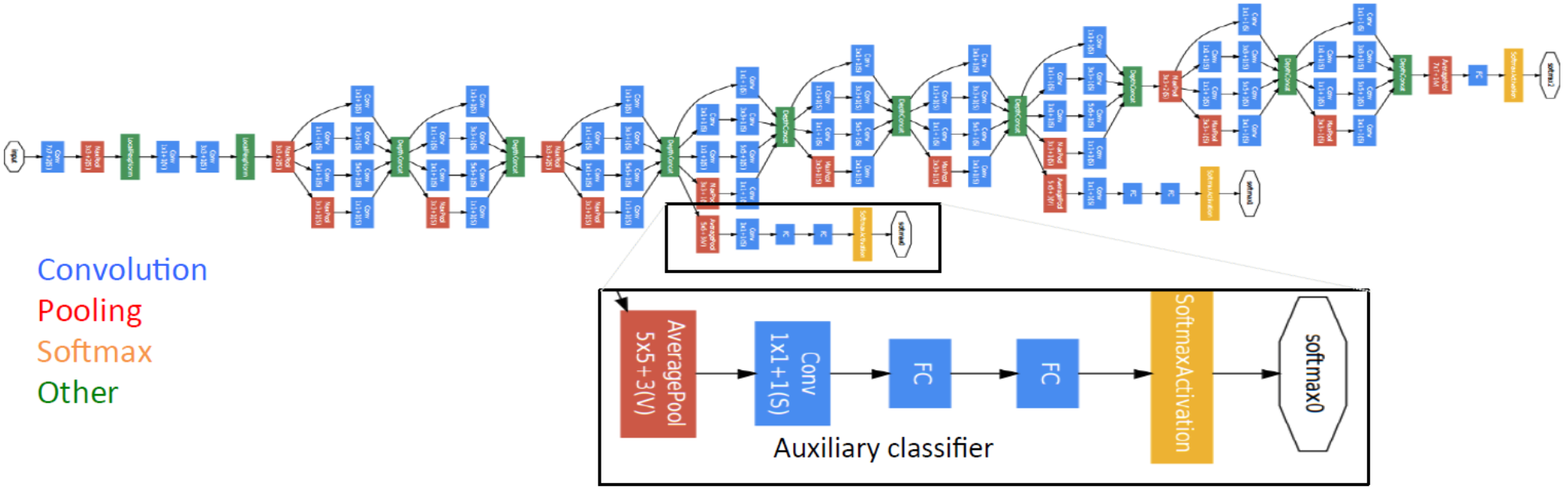


C.Szegedy et al. “Going Deeper with Convolutions.”, *CVPR 2015*.

Famous CNN architectures

GoogLeNet (2014)

- Two Softmax Classifiers at intermediate layers combat the vanishing gradient while providing regularization at training time.



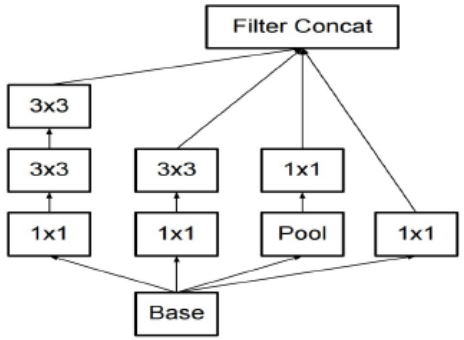
...and no fully connected layers needed !

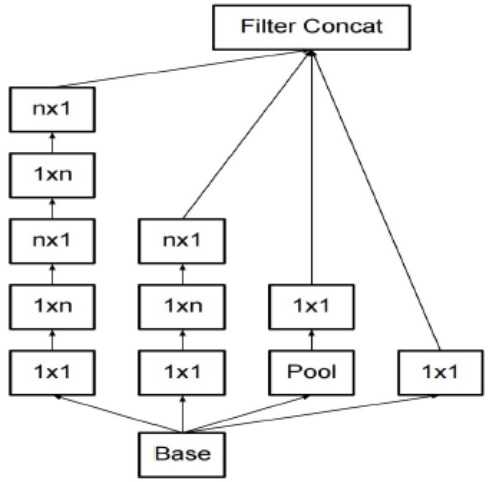
C.Szegedy et al. “Going Deeper with Convolutions.”, *CVPR 2015*.

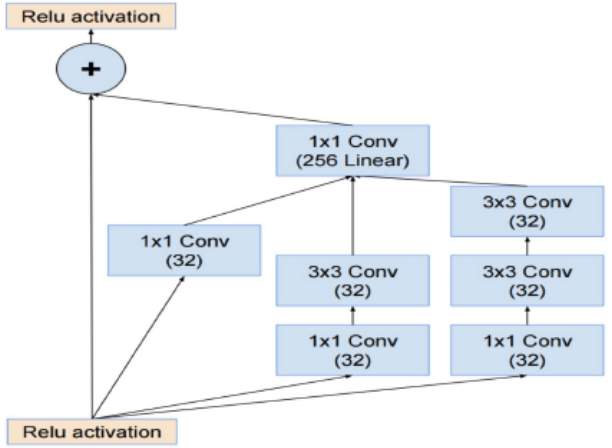
Famous CNN architectures

Inception v2, v3, v4

- Regularize training with batch normalization, reducing importance of auxiliary classifiers
- More variants of inception modules with aggressive factorization of filters





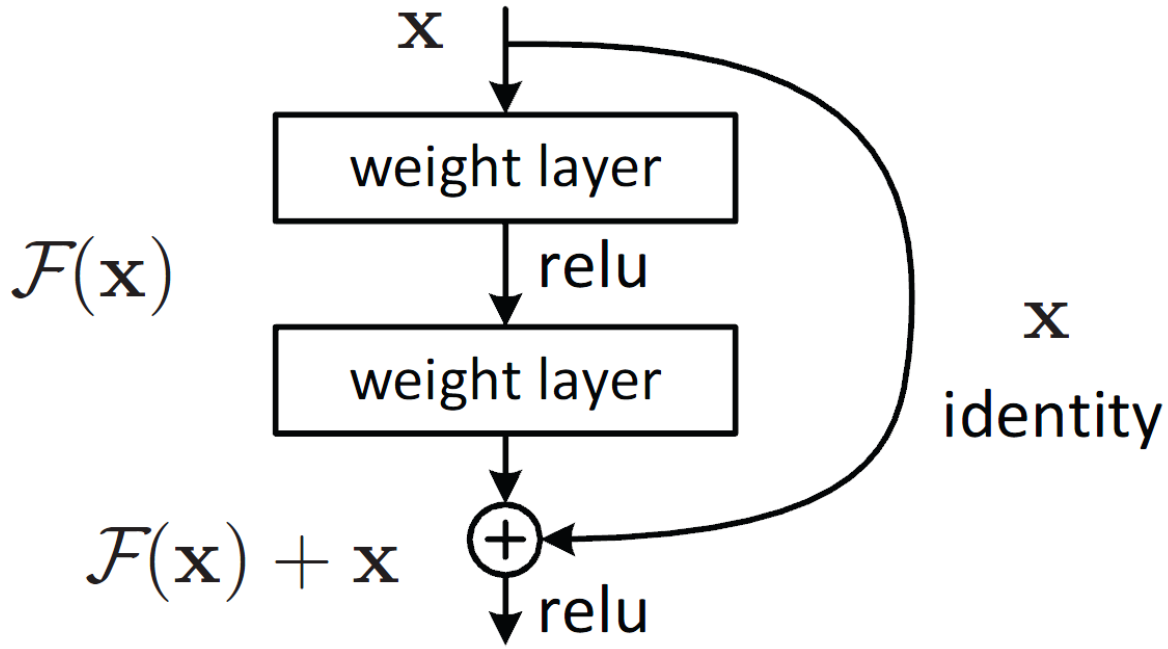


C. Szegedy et al., Rethinking the inception architecture for computer vision, CVPR 2016

C. Szegedy et al., Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, arXiv 2016

Famous CNN architectures: **ResNets**

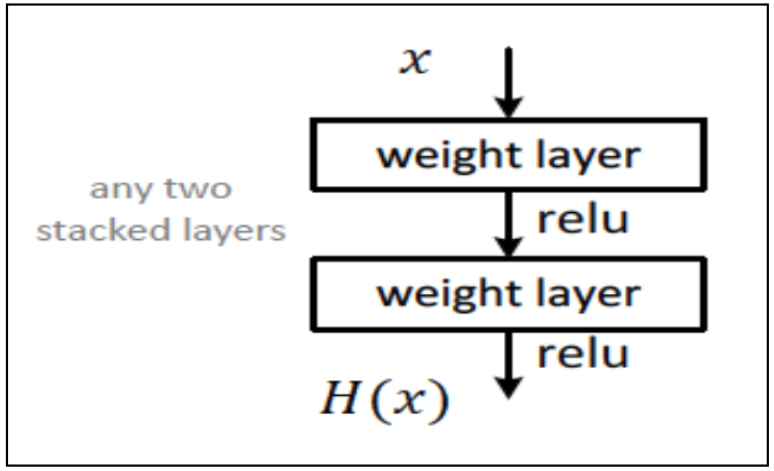
- Residual Networks developed by Kaiming He et al. were the winner of ILSVRC 2015. It features special *skip connections* and a heavy use of *batch normalization*. The architecture is also missing fully connected layers at the end of the network.
- CNN architecture that introduced *residual learning* or *identity* learning. ResNets allow the network to have *skip connections* and allow for much deeper convolutional networks.



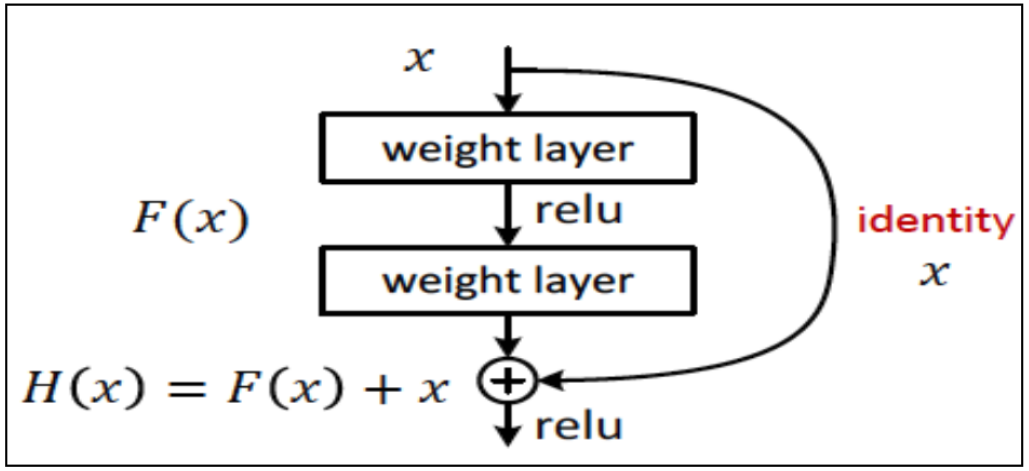
Kaiming He et al. "Deep Residual Learning for Image Recognition." *CVPR 2016*.

ResNet

Plain net



Residual net



Residual learning: reformulate the layers as **learning residual functions** with reference to the layer inputs, instead of learning unreferenced functions

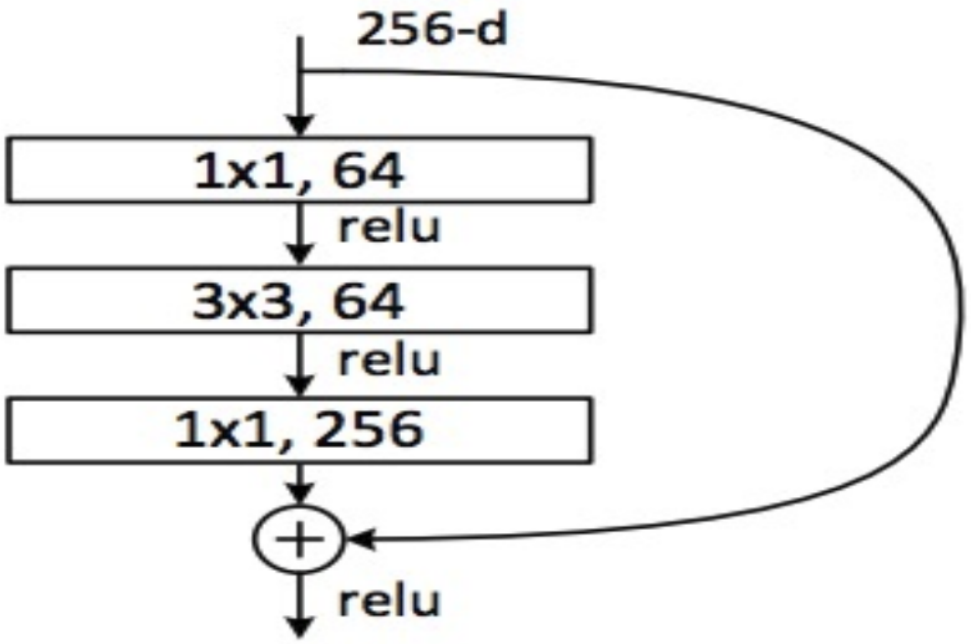
- Introduces skip or shortcut connections (existing before in the literature)
- Make it easy for net layers to represent the identity mapping

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

Kaiming He et al. "Deep Residual Learning for Image Recognition." *CVPR 2016*.

ResNet

- Deeper residual module (bottleneck)



- Directly performing 3x3 convolutions with 256 feature maps at input and output:
  256 x 256 x 3 x 3 ~ 600K operations
- Using 1x1 convolutions to reduce 256 to 64 feature maps, followed by 3x3 convolutions, followed by 1x1 convolutions to expand back to 256 maps:
  256 x 64 x 1 x 1 ~ 16K
  64 x 64 x 3 x 3 ~ 36K
  64 x 256 x 1 x 1 ~ 16K
  Total: ~70K

Kaiming He et al. "Deep Residual Learning for Image Recognition." *CVPR 2016*.

ResNet

ILSVRC 2015 winner (3.6% top 5 error)

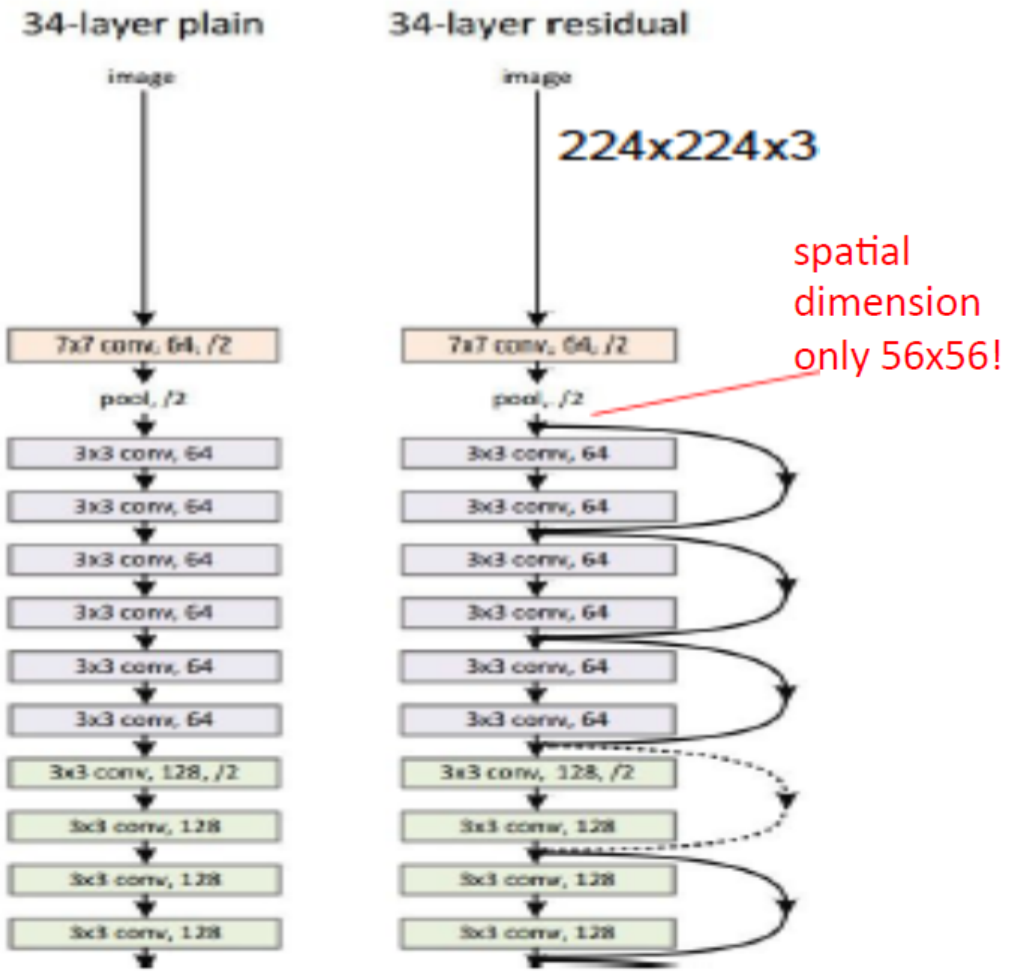
MSRA: ILSVRC & COCO 2015 competitions

- ImageNet Classification: “ultra deep”, 152-leyers
- ImageNet Detection: 16% better than 2nd
- ImageNet Localization: 27% better than 2nd
- COCO Detection: 11% better than 2nd
- COCO Segmentation: 12% better than 2nd

2-3 weeks of training on 8 GPU machine

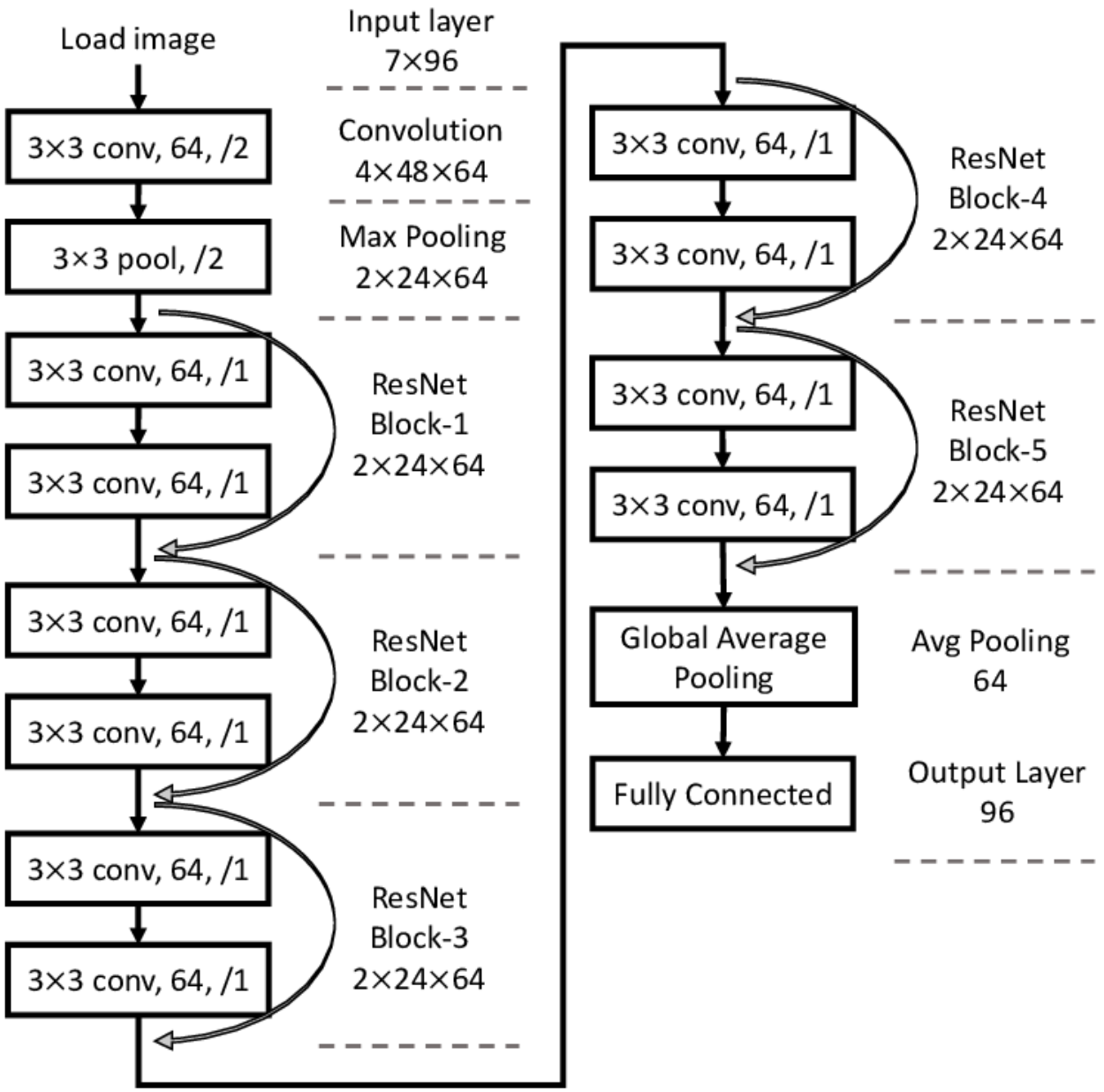
at runtime: faster than a VGGNet!

(even though it has 8x more layers)



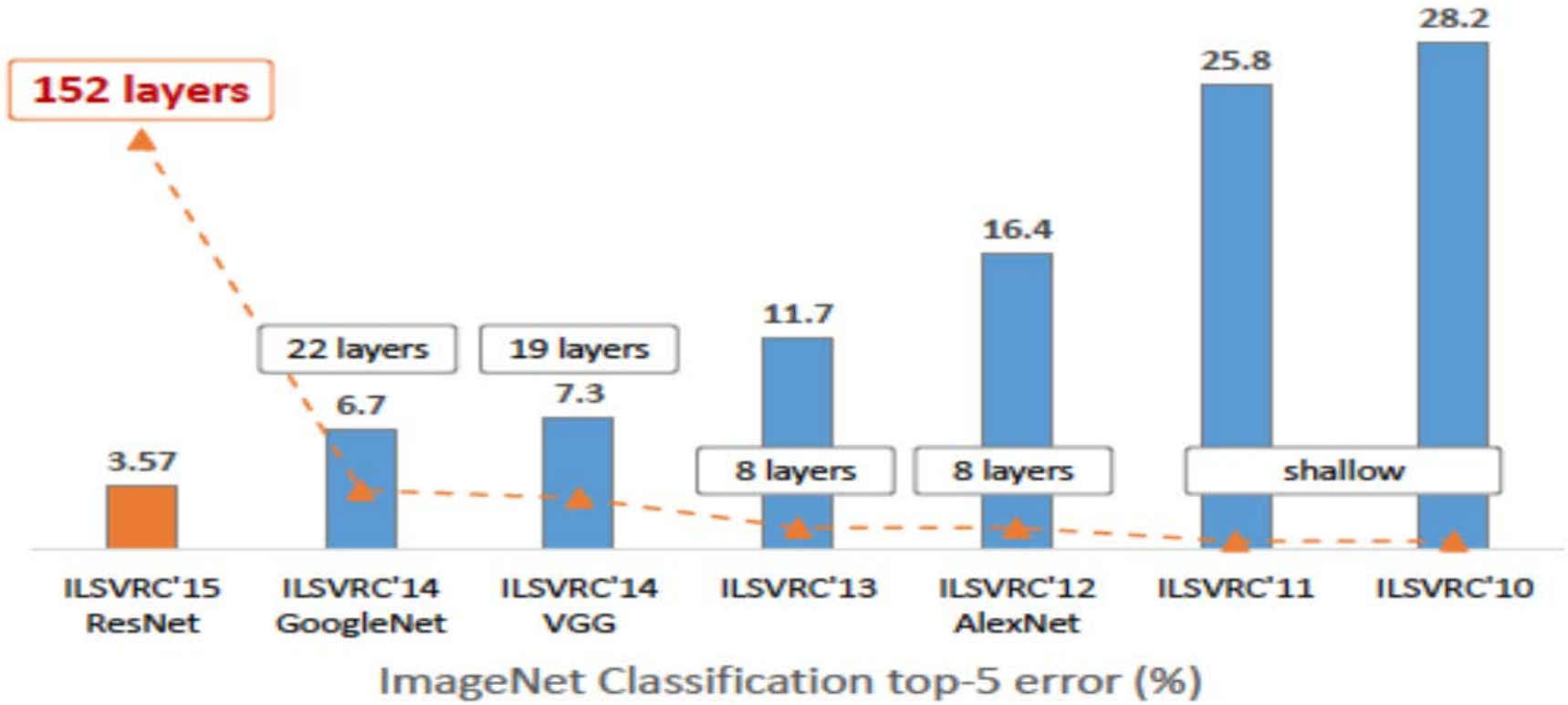
Kaiming He et al. “Deep Residual Learning for Image Recognition.” *CVPR 2016*.

ResNet Walkthrough



Kaiming He et al. “Deep Residual Learning for Image Recognition.” *CVPR 2016*.

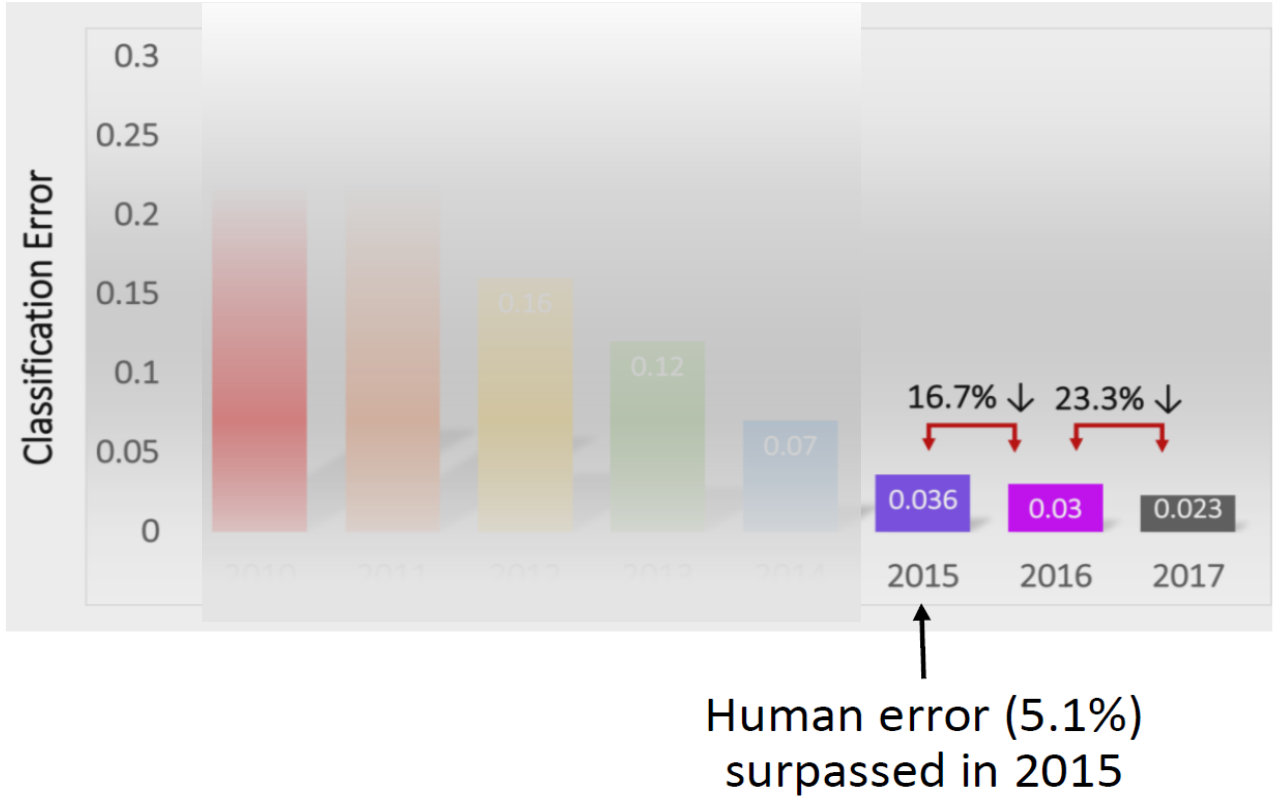
Deeper models



ILSVRC 2012-2015 summary

Team	Year	Place	Error (top-5)	External data
SuperVision – Toronto (**AlexNet**, 7 layers)	2012	-	16.4%	no
SuperVision	2012	1st	15.3%	ImageNet 22k
Clarifai – NYU (7 layers)	2013	-	11.7%	no
Clarifai	2013	1st	11.2%	ImageNet 22k
VGG – Oxford (16 layers)	2014	2nd	7.32%	no
GoogLeNet (19 layers)	2014	1st	6.67%	no
ResNet (152 layers)	2015	1st	3.57%	
Human expert*			5.1%	

Performance summary



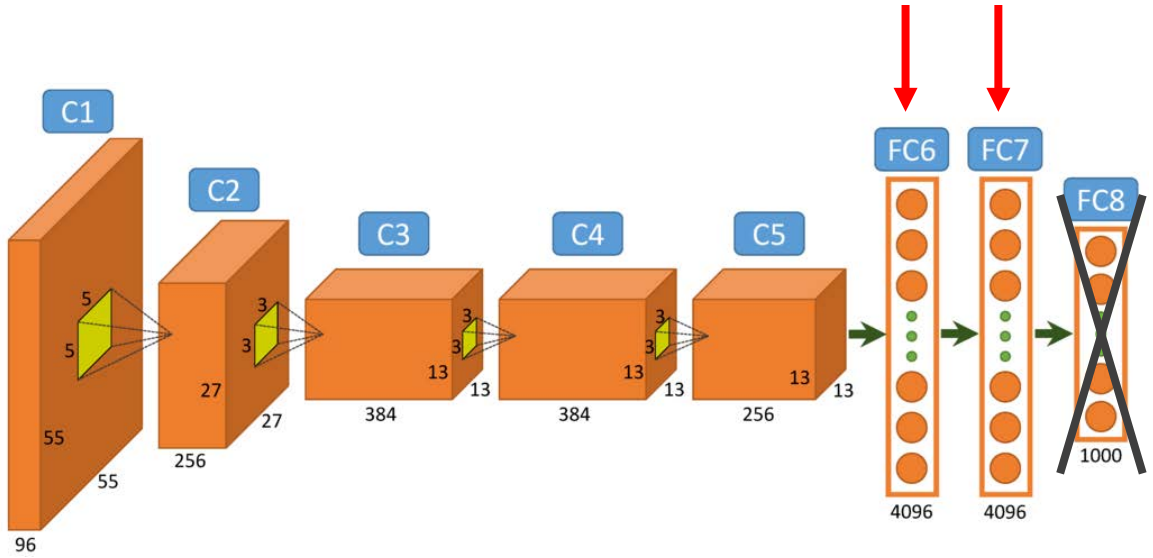
- **AlexNet (2012): First CNN (15.4%)**
  - 8 layers
  - 61 million parameters
- **ZFNet (2013): 15.4% to 11.2%**
  - 8 layers
  - More filters. Denser stride.
- **VGGNet (2014): 11.2% to 7.3%**
  - Beautifully uniform:
    3x3 conv, stride 1, pad 1, 2x2 max pool
  - 16 layers
  - 138 million parameters
- **GoogLeNet (2014): 11.2% to 6.7%**
  - Inception modules
  - 22 layers
  - 5 million parameters
    (throw away fully connected layers)
- **ResNet (2015): 6.7% to 3.57%**
  - More layers = better performance
  - 152 layers
- **CUImage (2016): 3.57% to 2.99%**
  - Ensemble of 6 models
- **SENet (2017): 2.99% to 2.251%**
  - Squeeze and excitation block: network is allowed to adaptively adjust the weighting of each feature map in the convolutional block.

Transfer Learning

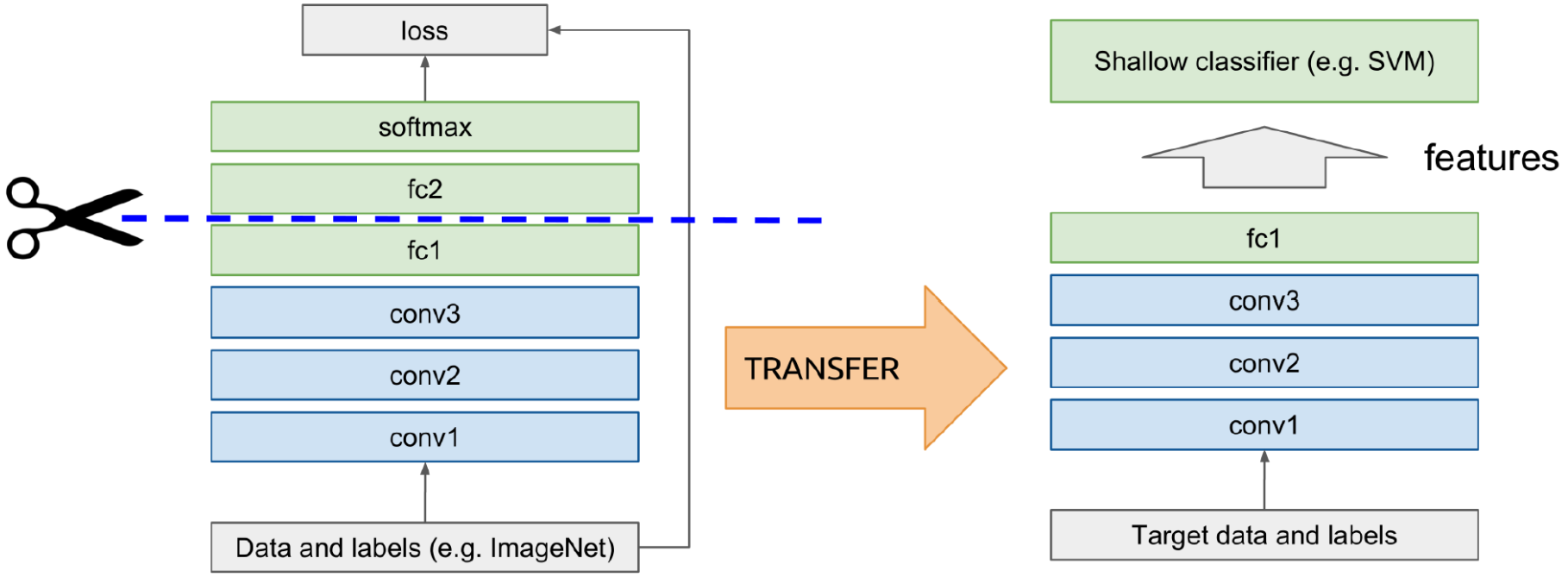
Transfer learning and its strategies – CNN codes

- Very few people train from scratch (with random initialization) a CNN (no data, time - weeks!)
- Instead, it is common to (*let others*) pretrain a ConvNet on a very large dataset (e.g., ImageNet, which contains 1.2 million images with 1000 categories), and then:

1. **ConvNet as fixed feature extractor**:
    a. Take a ConvNet pretrained on ImageNet
    b. Remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet)
    c. Treat the rest of the ConvNet as a fixed feature extractor for the new dataset.
        - In an AlexNet, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier.



Transfer learning – Fine tuning



- Fine-tune a pre-trained model
- Effective in many applications: computer vision, audio, speech, natural language processing

Transfer learning – Fine tuning

1. **Fine tuning**:
    a. Start with an initialization already computed by backpropagation
    b. Do backpropagation on the layers you want
        - Usually, only the last layers are trained, the earlier are more generic and are preferred to be left unchanged

In particular, four scenarios are available:

- *New dataset is small and similar to original dataset (**NO FINE TUNING**)*. Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.

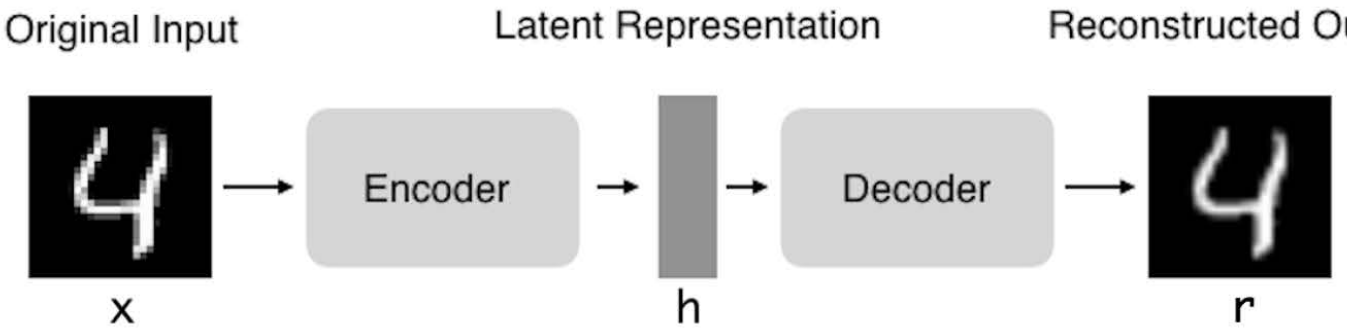
- *New dataset is large and similar to the original dataset*. Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.

Transfer learning – Fine tuning

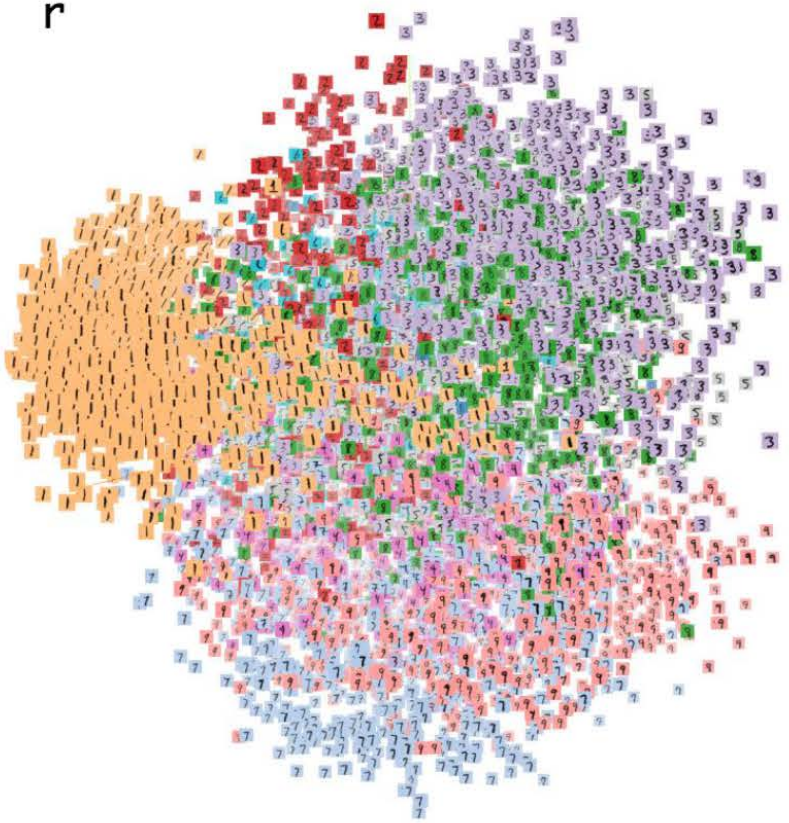
- *New dataset is small but very different from the original dataset*. Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier from the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.

- *New dataset is large and very different from the original dataset*. Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch. However, in practice it is very often still beneficial to initialize with weights from a pre-trained model. In this case, we would have enough data and confidence to fine-tune through the entire network.

Other DL models – Autoencoders

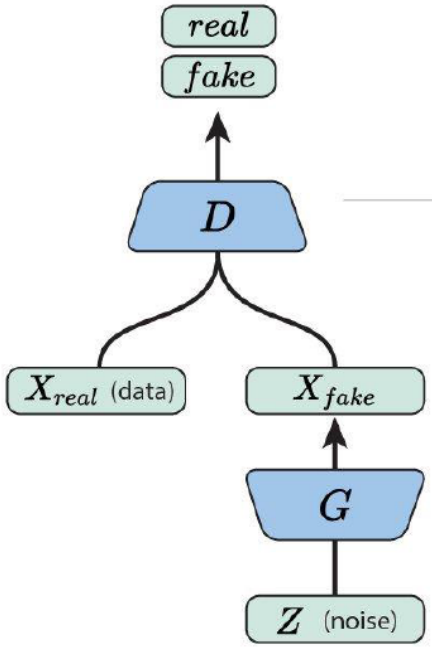


- Unsupervised learning
- Gives embedding
  - Typically better embeddings come from discriminative task



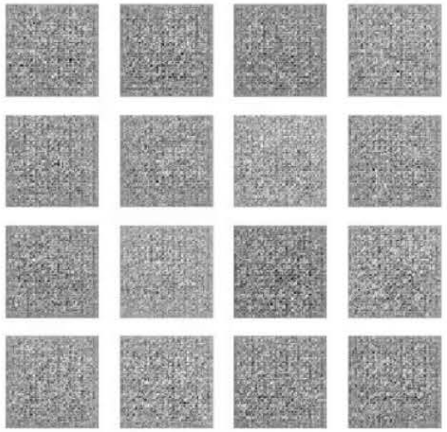
Other DL models – GANs

Generative Adversarial Networks (GANs) are a way to make a generative model by having two neural networks compete with each other.



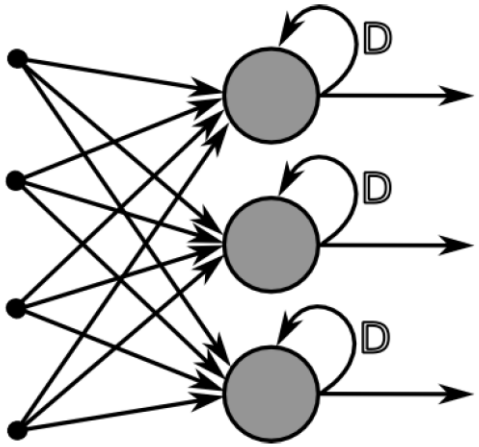
The **discriminator** tries to distinguish genuine data from forgeries created by the generator.

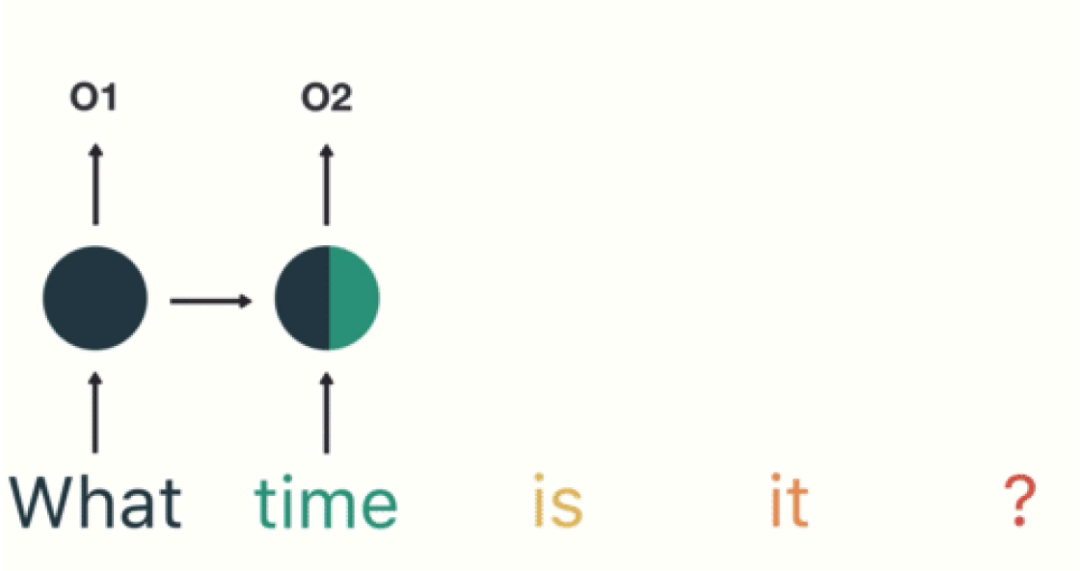
The **generator** turns random noise into immitations of the data, in an attempt to fool the discriminator.



Progressive GAN
10/2017
1024 x 1024

Other DL models – Recurrent NN

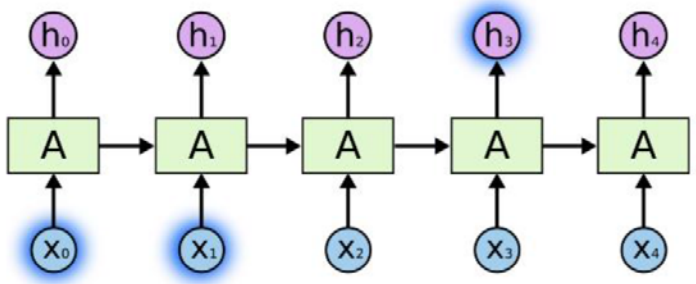




- Applications
  - Sequence Data
  - Text
  - Speech
  - Audio
  - Video
  - Generation

Other DL models – Recurrent NN

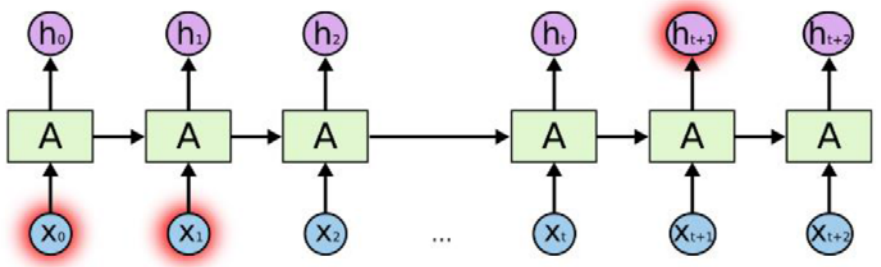
Long-Term Dependency



- Short-term dependence:
  Bob is eating an **apple.**

Context →

- Long-term dependence:
  Bob likes **apples**. He is hungry and decided to have a snack. So now he is eating an **apple.**



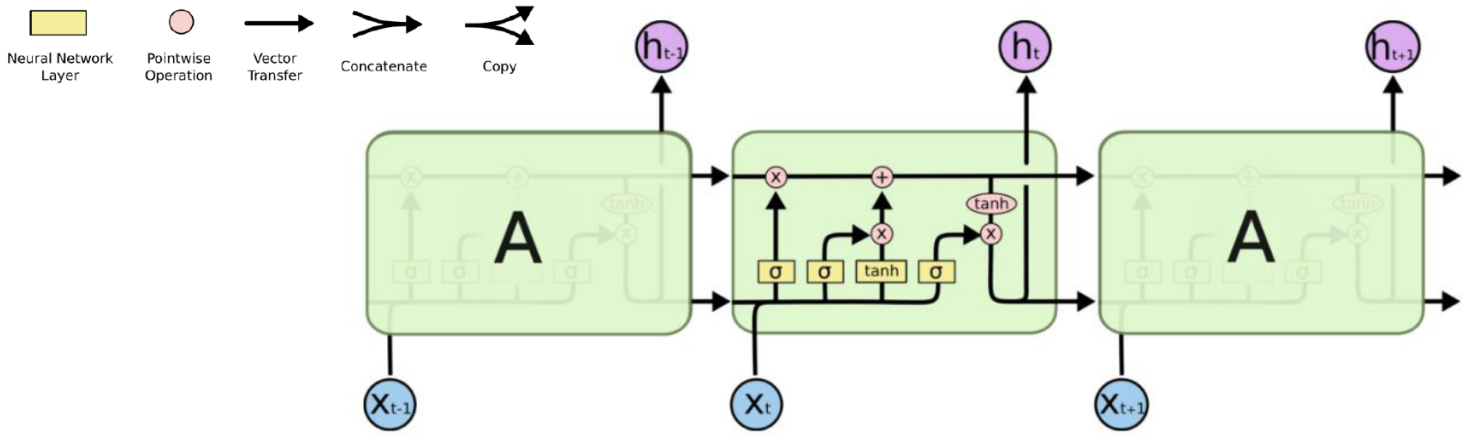
In theory, vanilla RNNs can handle arbitrarily long-term dependence.

In practice, it's difficult.

Other DL models – Long Short-Term Memory nets

LSTM

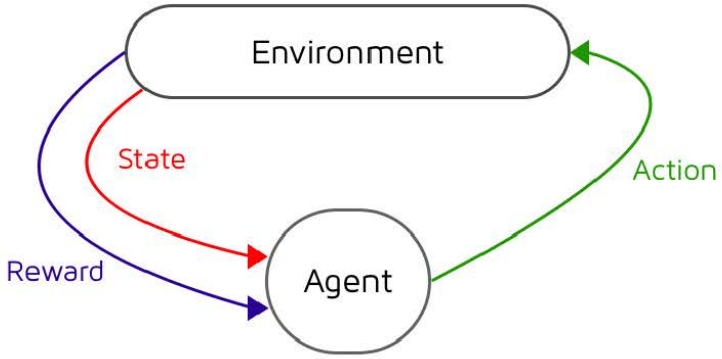
Pick What to Forget and What To Remember

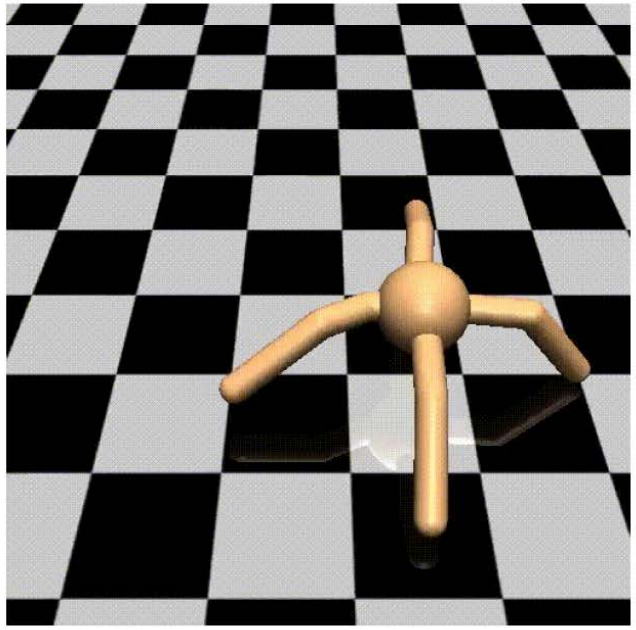


Conveyer belt for **previous state** and **new data**:

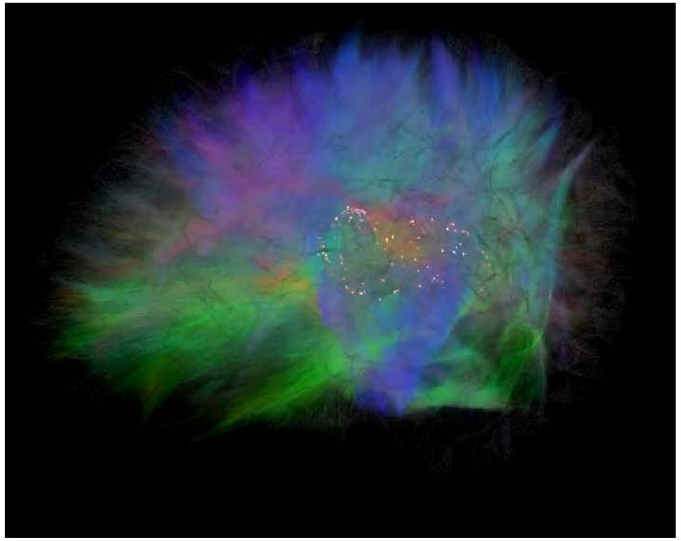
1. Decide what to forget (state)
2. Decide what to remember (state)
3. Decide what to output (if anything)

Deep Reinforcement Learning





Towards *general* AI



- Transfer Learning
- Hyperparameter Optimization
- Architecture Search
- Meta Learning