ACV – Applied Computer Vision

Bachelor Medientechnik & Creative Computing

Matthias Zeppelzauer
matthias.zeppelzauer@fhstp.ac.at

Djordje Slijepcevic
djordje.slijepcevic@fhstp.ac.at

Roadmap

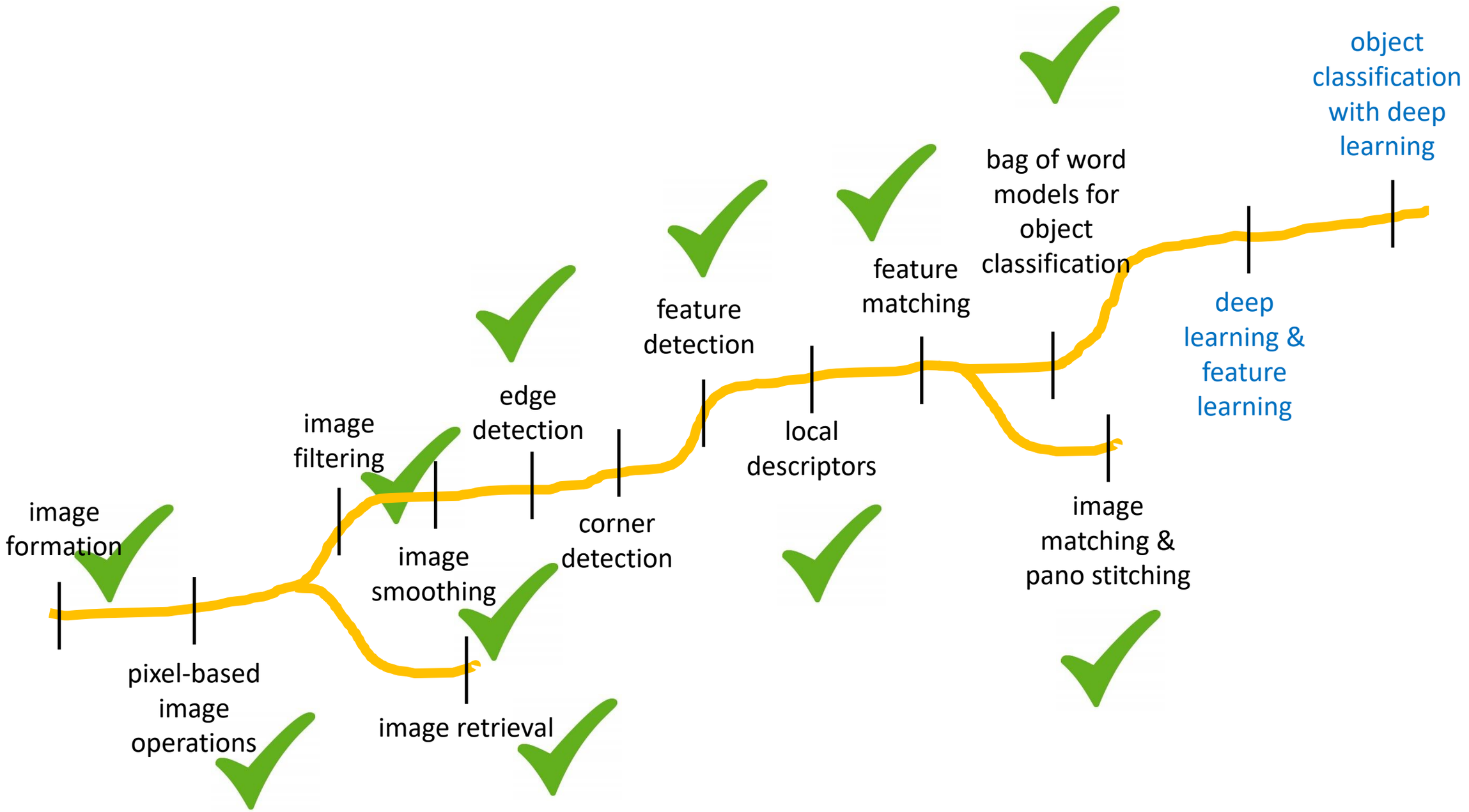


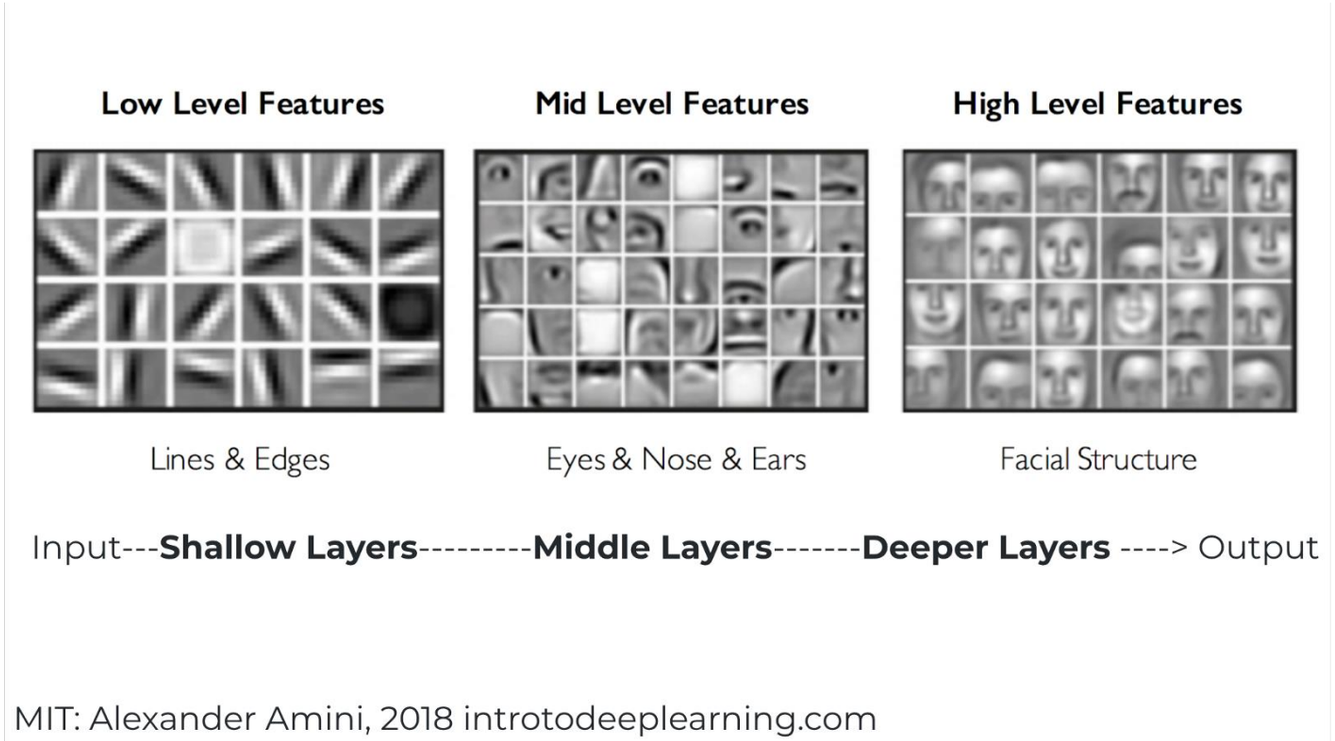
image formation
pixel-based image operations
image filtering
image smoothing
edge detection
corner detection
feature detection
image retrieval
feature matching
local descriptors
bag of word models for object classification
image matching & pano stitching
deep learning & feature learning
object classification with deep learning

Digitale Medientechnologien

Neural Network Training

Motivation: End-to-end learning

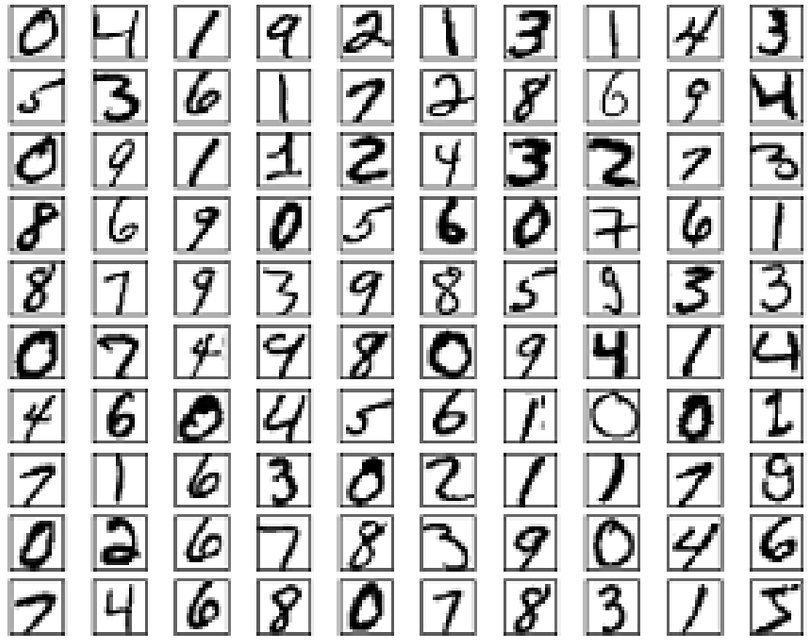
- Basic Concepts / Ideas:
  - Learn from input directly the output (e.g. input = image, output = class label)
  - Disentangle the underlying factors in the input data that make up a pattern of interest
    - Humans are (as shown in the past) not so well-suited for this task (feature engineering)
  - Learn a **representation** that is
    - hierarchical (simple structures build upon each other to model more complex structures)
    - and where concepts are shared between different classes



MIT: Alexander Amini, 2018 introtodeeplearning.com

Recap: Example on Designing a Network

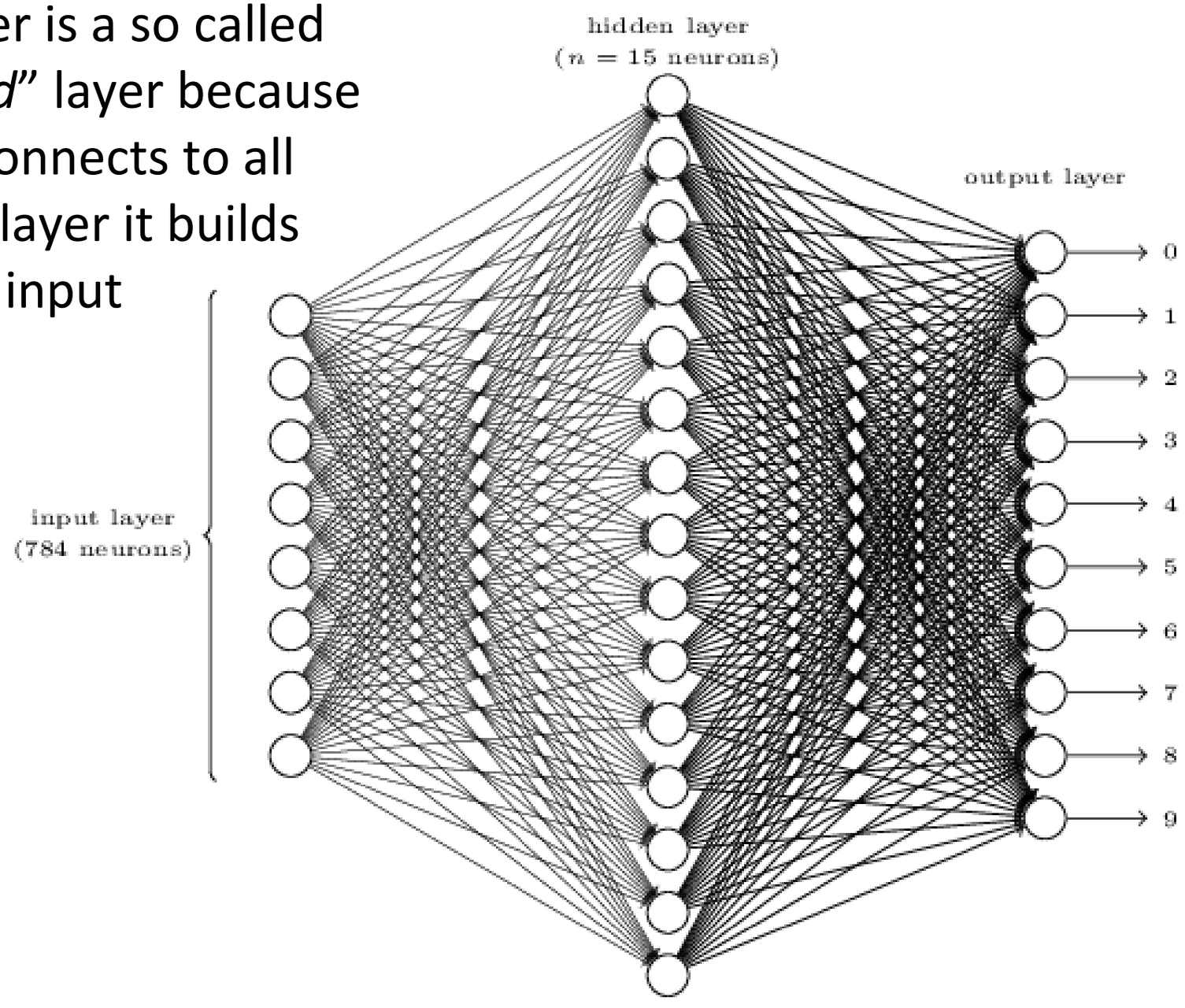
- Input handwritten digit images of size 28x28 pixels
- Output: the digit in the image (0..9)?
- How would the network look like?



- Solution: 784 input nodes, “some” hidden layers, one output neuron per digit

Designing a network continued

- Example network for digit classification
- The hidden layer is a so called "*fully connected*" layer because every neuron connects to all neurons of the layer it builds upon (here the input layer)

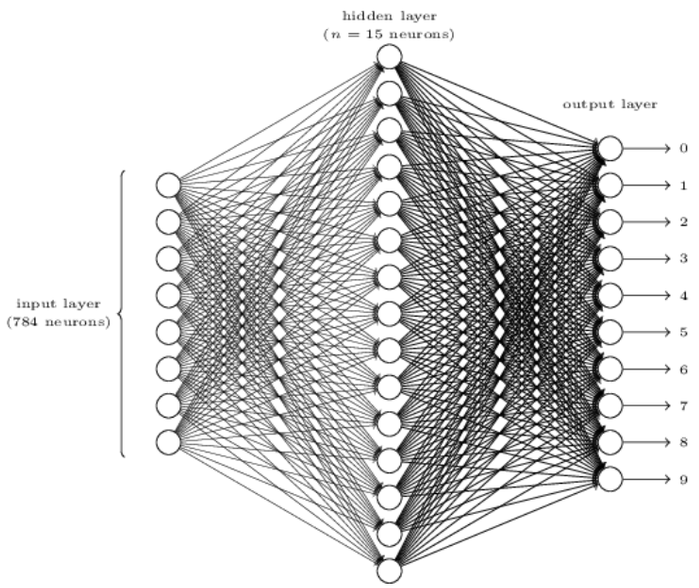


How to train a network?

- Training a network means estimating weights and biases so that the desired outputs are achieved.
- In our example:
  - Input x: 784 gray value pixels of input: 
  - Output y: 10 values (a 10-element vector). If digit is “6”, output should be:

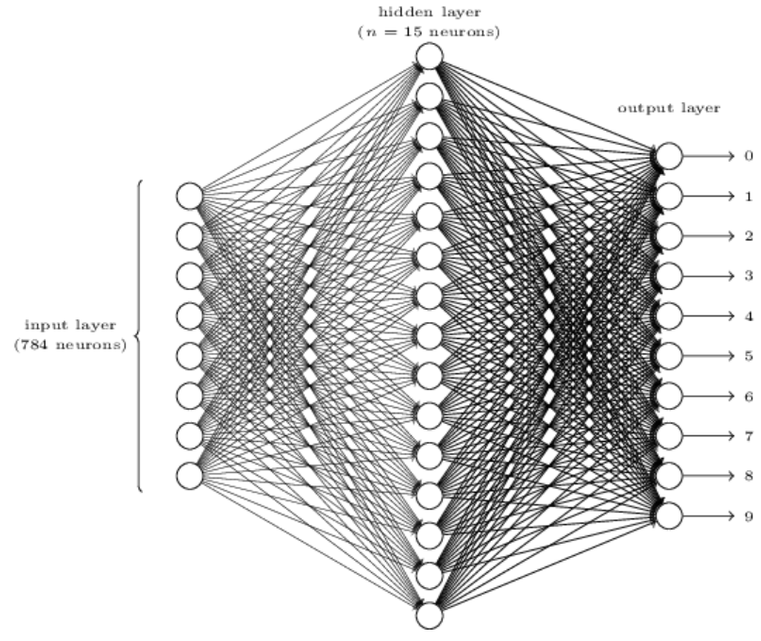
$$= (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$$

- Find weights and biases so that network f(.) produces correct outputs for the inputs: y=f(x) for all training inputs x_i



How to learn a network?

- Find weights and biases so that network f(.) produces outputs y for the inputs x: y=f(x) for all training inputs x_i where y = y' (y' is the correct output)
- Given: x and y
- Unknown: f(.)
  - What determines f(.)?
  - f(.) depends on weights w and biases b! → y = f(w,b,x)
- How to measure the quality of prediction?
  - $C(w,b) = \frac{1}{2n}\sum_x \|y' - f(w,b,x)\|^2$, n = #training samples
  - C ... "cost function" / "objective function" / "loss function"



Cost function

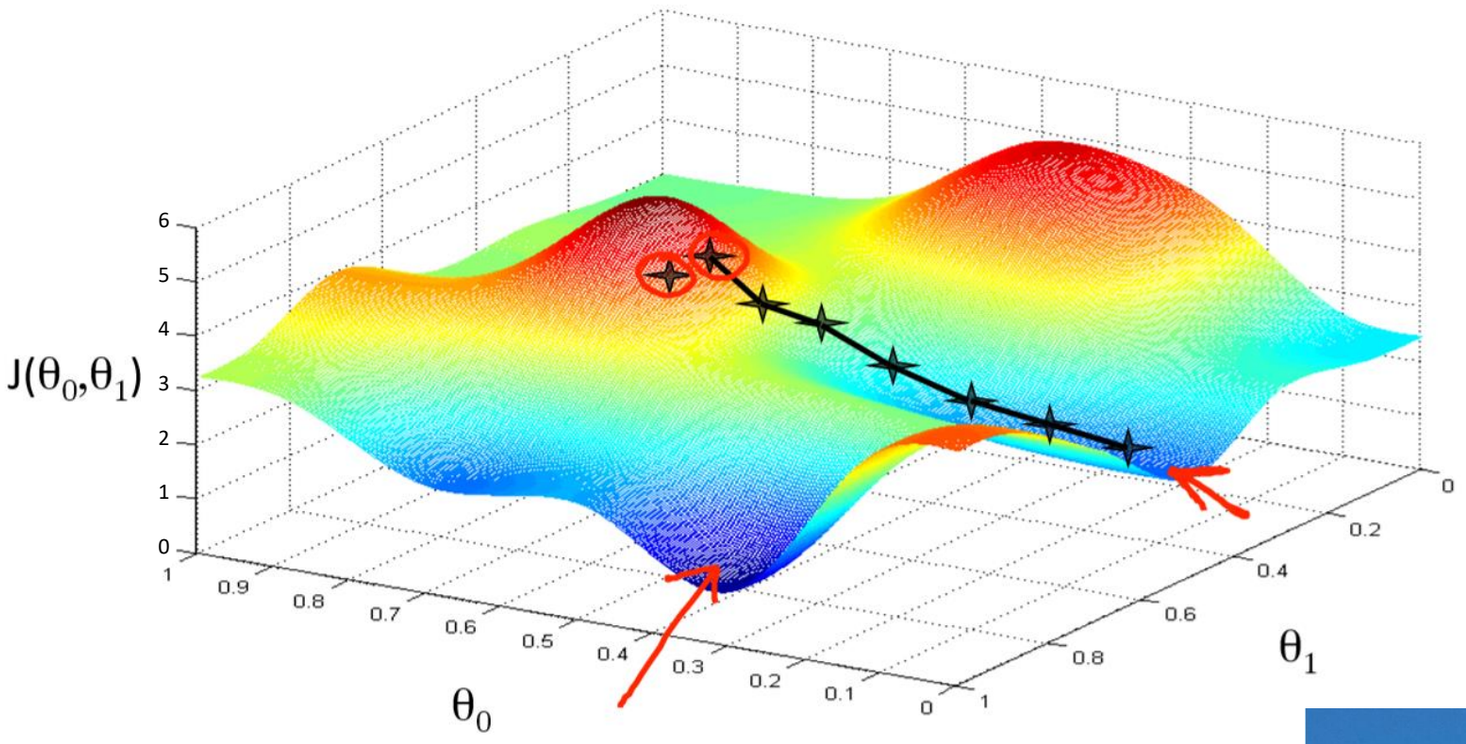
- C(w,b) is non-negative
- C(w,b) = 0 if all the outputs of the network correspond to the true output y’
  - Otherwise C(w,b) > 0
- The *better* the network the *smaller* C!

- For learning we want to find a set of weights and biases which make the cost as small as possible → minimize the cost functions
- Solution for this: the *gradient descent algorithm*!

- Why not just maximize the number of correctly classified images rather than C?

Gradient Descent

Intuition



Consider C(w,b) a multidimensional function (surface) and try to find its minimum (works in 2-D same as in N-D)

Is the concept of functions in multiple variables clear?

Gradient Descent

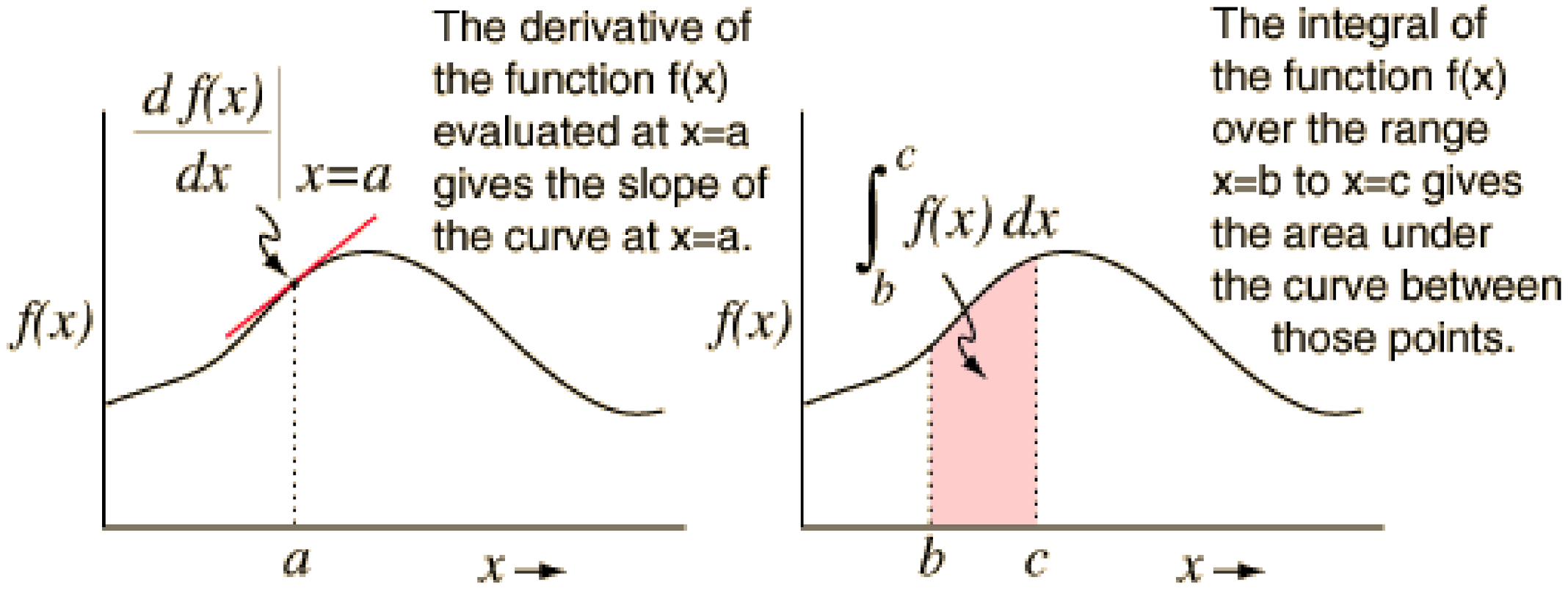
Heavily relies on derivation and partial derivatives...

Recap: Derivatives

- What is the derivative?

$$\frac{d\,f(x)}{dx}$$

$$\int f(x)\,dx$$

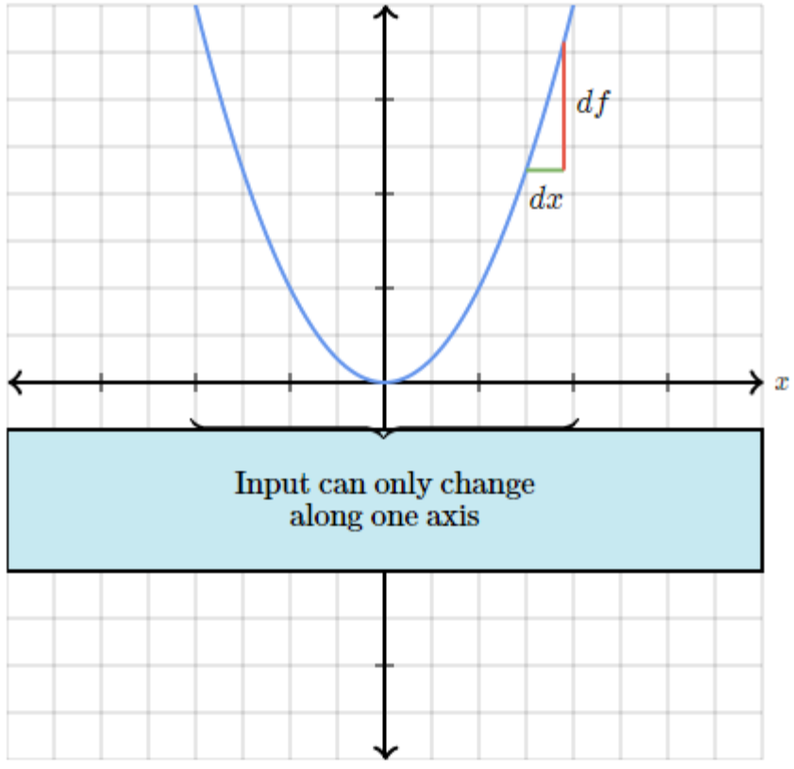


credit: http://hyperphysics.phy-astr.gsu.edu/hbase/math/derint.html

Recap: Derivatives

$$f(x)' = \frac{df}{dx}$$

- What does this notation mean?
- dx is a very small change in x
- df is a very small change in the output of function f
- the change of df follows from the very small change in x!
- Thus: “df/dx” means how much does the output of f change when we change x?



credit: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/introduction-to-partial-derivatives

Recap: Derivatives - Rules

Power Rule	$\frac{d}{dx}\left(x^a\right) = a \cdot x^{a-1}$
Derivative of a constant	$\frac{d}{dx}(a) = 0$
Sum Difference Rule	$(f \pm g)' = f' \pm g'$
Constant Out	$(a \cdot f)' = a \cdot f'$
Product Rule	$(f \cdot g)' = f' \cdot g + f \cdot g'$
Quotient Rule	$\left(\frac{f}{g}\right)' = \frac{f' \cdot g - g' \cdot f}{g^2}$
Chain rule	$\frac{df(u)}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$

credit: https://www.symbolab.com/cheat-sheets/Derivatives#

The Chain Rule

$$\frac{df(u)}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

- So, f is a function of u and u a function of x, i.e. u = g(x)
- This means y = f(u) and u = g(x), thus we can also say: $f(g(x))$
  This is a nested function (remember: *"innere Ableitung"* vs. *"äußere Ableitung"*)
- The chain rule says: compute simply the product of both derivatives:

$$f(g(x))' = g'(x)*f'(g(x))$$

- This is: $g'(x) = u' = \frac{dg(x)}{dx} = \frac{du}{dx}$
- And: $f'(x) = \frac{df(g(x))}{dg(x)} = \frac{df(u)}{du}$

$$\Big\} \quad f\big(g(x)\big)' = \frac{df}{du} * \frac{du}{dx}$$

credit: https://www.symbolab.com/cheat-sheets/Derivatives#

The Chain Rule - Example

- $y = (x^3 + 4x)^{15}$
- inner function: $u = g(x) = x^3 + 4x$
- outer function: $y = f(u) = u^{15}$
- Chain rule: $y' = f'(u) = f\big(g(x)\big)' = g'(x) * f'(g)$

- $g'(x) = \frac{du}{dx} = 3x^2 + 4$
- $f'\big(g(x)\big) = \frac{df}{du} = f'(g) = f'(u) = 15u^{14} = 15(x^3 + 4x)^{14}$
- $y' = f'(x) = \frac{df}{dx} = \frac{df}{du} * \frac{du}{dx} = (15(x^3 + 4x)^{14}) * (3x^2 + 4)$

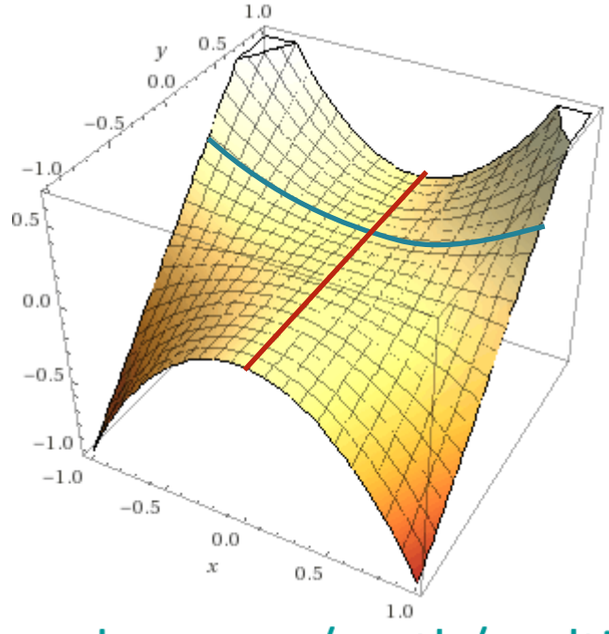
credit: https://www.symbolab.com/cheat-sheets/Derivatives#

Partial Derivatives (Partielle Ableitungen)

- Given: a function in several variables, e.g. $z = f(x, y) = x^2 y$
- You can compute the "full" derivative of "f derived by x and y", i.e. $\frac{df(x,y)}{dxdy}$
- Or the partial derivatives: $\frac{\partial f(x,y)}{\partial x}$ and $\frac{\partial f(x,y)}{\partial y}$

$$\frac{\partial f}{\partial x} = \underbrace{\frac{\partial}{\partial x} x^2 y}_{\text{Treat } y \text{ as constant; take derivative.}} = 2xy$$

$$\frac{\partial f}{\partial y} = \underbrace{\frac{\partial}{\partial y} x^2 y}_{\text{Treat } x \text{ as constant; take derivative.}} = x^2 \cdot 1$$



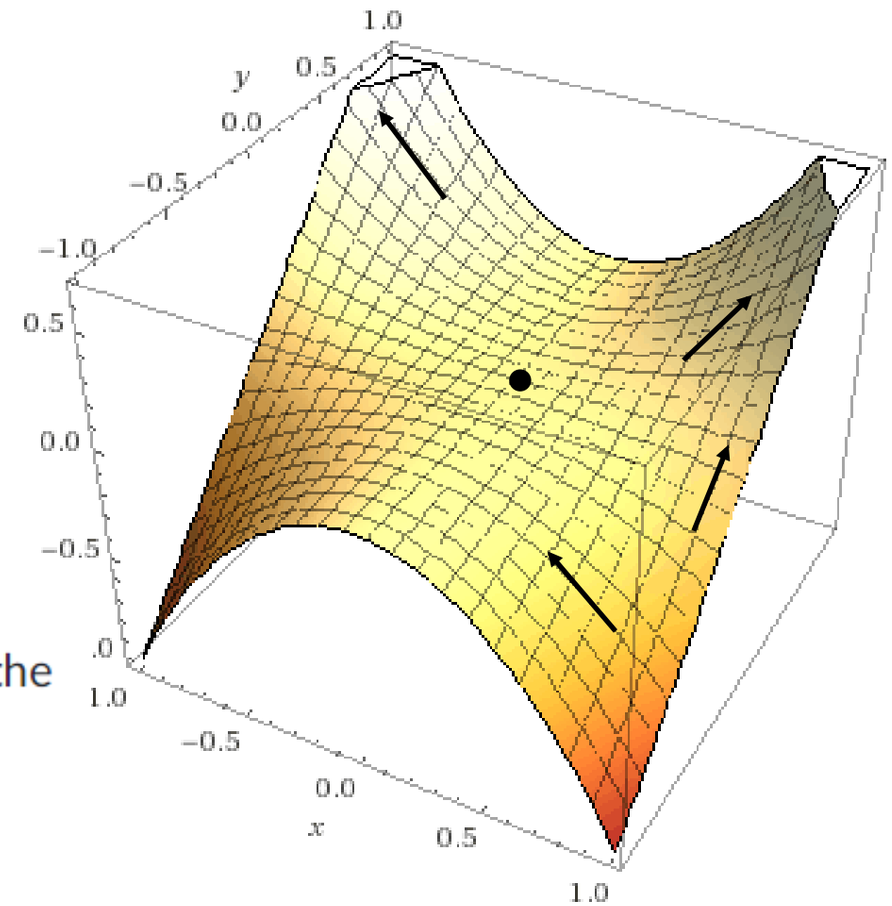
Meaning: how does the function z change as we let just one of its variables change while holding the other constant?

- Credit: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/introduction-to-partial-derivatives

The Gradient

- Partial derivatives provide only an incomplete picture (only the change in one direction)
- To get the full picture (the „full“ derivative), we can take all partial derivatives together in one vector, this is the gradient:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \vdots \end{bmatrix}$$



If you imagine standing at a point $(x_0, y_0, \ldots)$ in the input space of f, the vector $\nabla f(x_0, y_0, \ldots)$ tells you which direction you should travel to increase the value of f most rapidly.

Credit: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/the-gradient

The Gradient

Think of the graph of f as a hilly terrain. If you are standing on the part of the graph directly above—or below—the point (x_0, y_0), the slope of the hill depends on which direction you walk. For example, if you step straight in the positive x direction, the slope is $\frac{\partial f}{\partial x}$; if you step straight in the positive y-direction, the slope is $\frac{\partial f}{\partial y}$. But most directions are some combination of the two. [Display image showing this concept.]

The most important thing to remember about the gradient: The gradient of f, if evaluated at an input (x_0, y_0), points in the direction of steepest ascent.

So, if you walk in the direction of the gradient, you will be going straight up the hill. Similarly, *the magnitude of the vector* $\nabla f(x_0, y_0)$ *tells you what the slope of the hill is* in that direction.

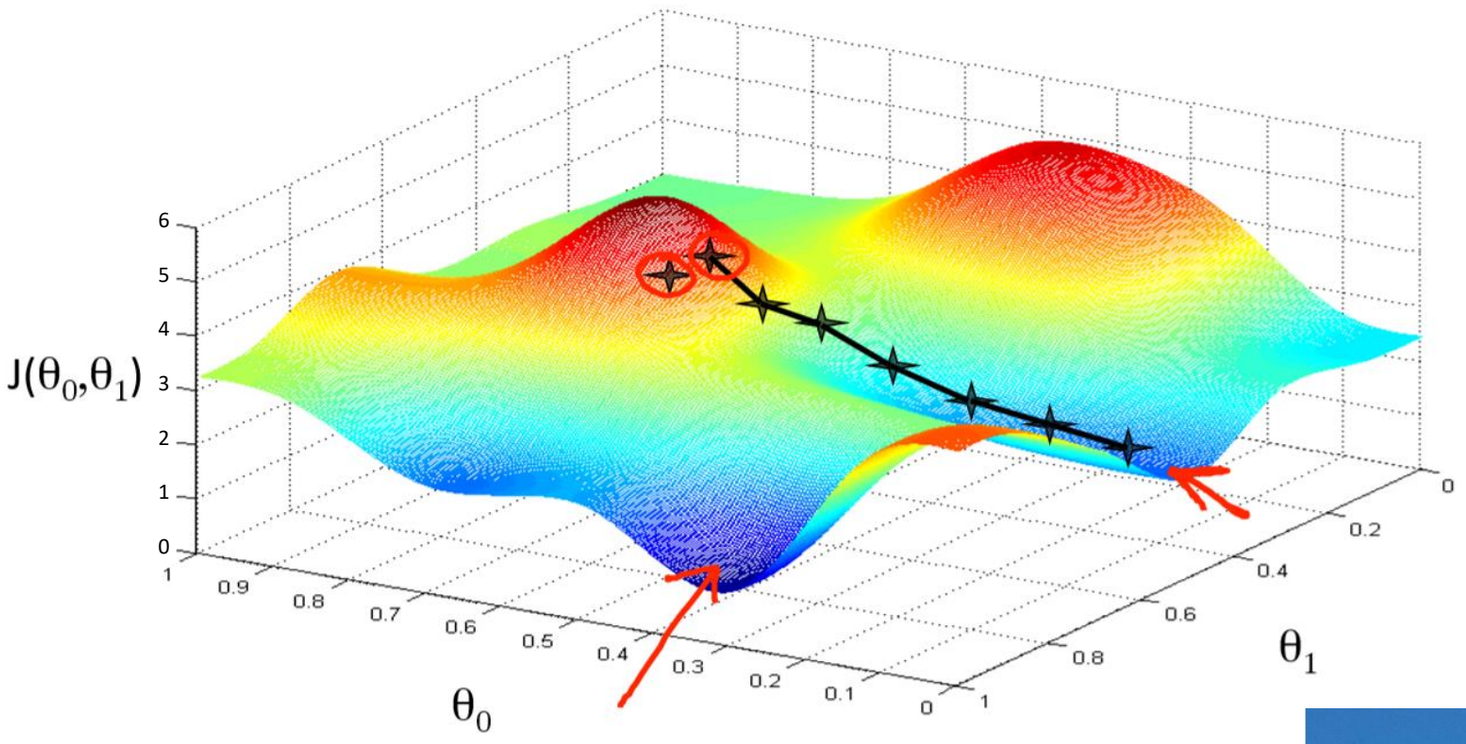
Credit: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/the-gradient

Take Home Message

- The gradient shows in the direction of the **steepest ascent**
- The negative gradient shows in the direction of the **steepest descent**
- We can compute the gradient (for arbitrary dimensional functions f(x,y,z...) by
  - computing the **partial derivatives** df/dx, df/dy, df/dz ...
  - and stacking them into a **vector**: [df/dx, df/dy, df/dz, ...]
- For the gradient we use the symbol ∇, i.e.: ∇f

Recap: Gradient Descent

Intuition

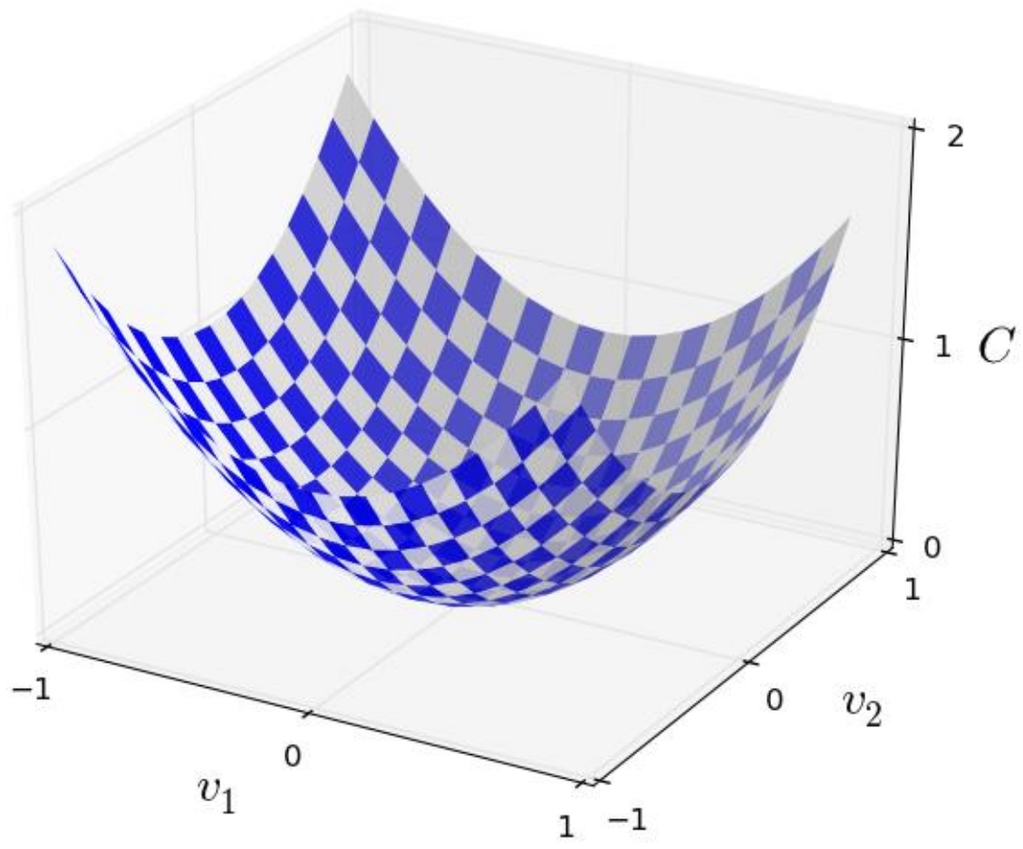


Consider C(w,b) a multidimensional function (surface) and try to find its minimum (works in 2-D same as in N-D)

Is the concept of functions in multiple variables clear?

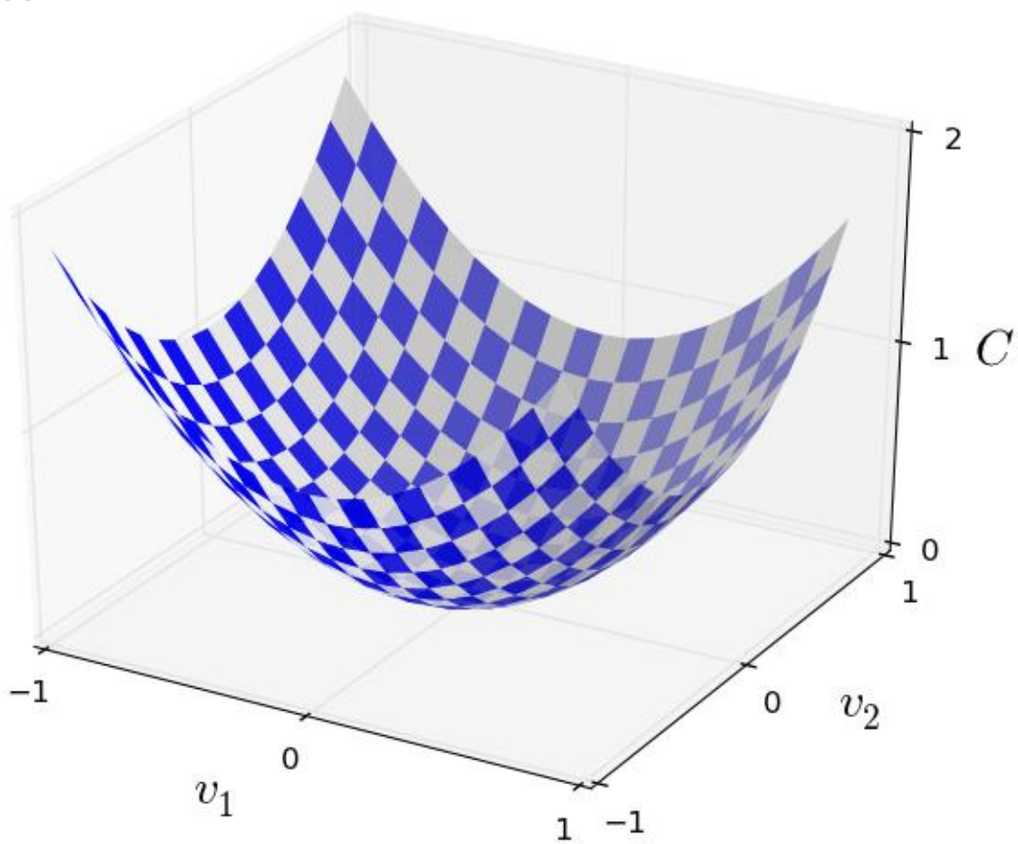
Gradient Descent - Intuition

- C(w,b) is a multi-dimensional function (many weights w and biases b)
  - Note: w and b are actually large vectors
- For simplicity, imagine C would depend only on two variables: a,b, i.e. $C(v_1,v_2)$
- Task: find the minimum!



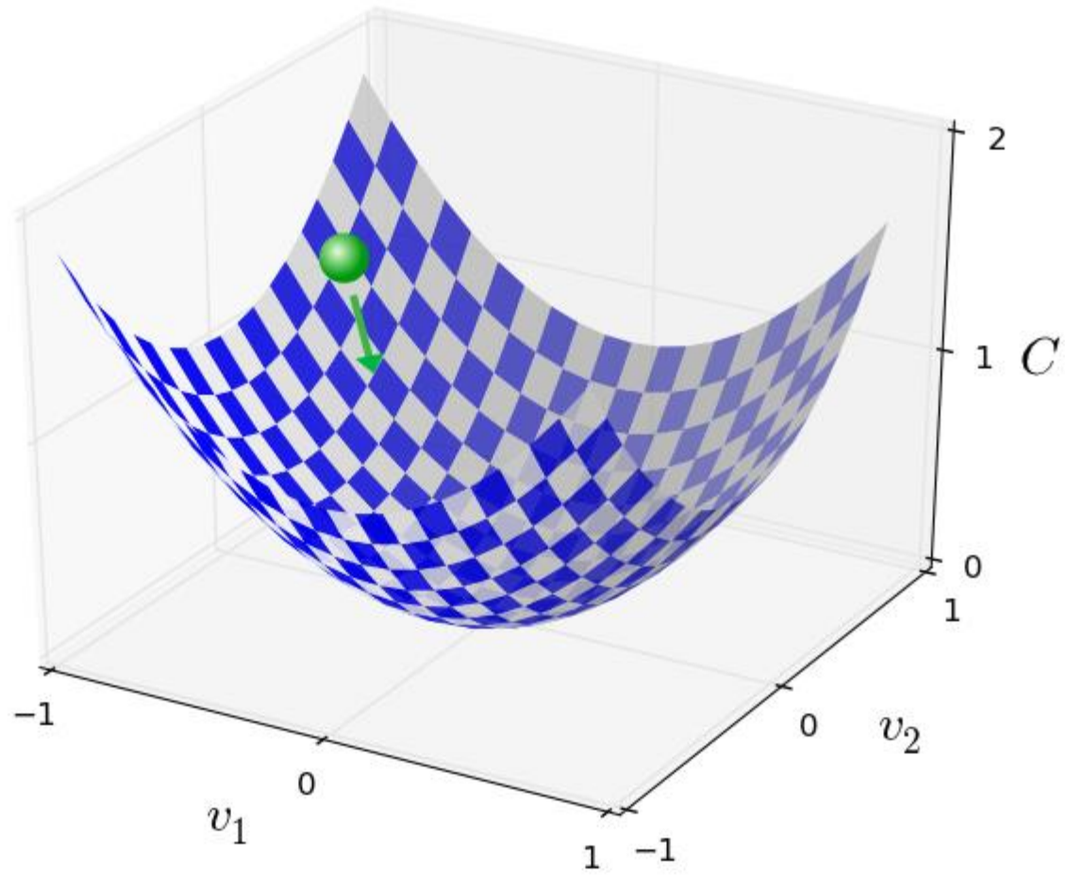
Gradient Descent - Intuition

- Finding minimum analytically would be possible for 2 dimensions but does not work for hundreds of dimensions
- Gradient descent provides a solution!
- Think of the function as a valley and put an imaginary ball onto the surface and let it roll → it rolls towards the minimum
- In which direction does the ball roll?
  - Hint: remember “edge detection”



Gradient Descent - Intuition

- Summing up, the way the gradient descent algorithm works is to repeatedly compute the gradient ∇C, and then to move in the opposite direction, "falling down" the slope of the valley.



Gradient Descent - Intuition

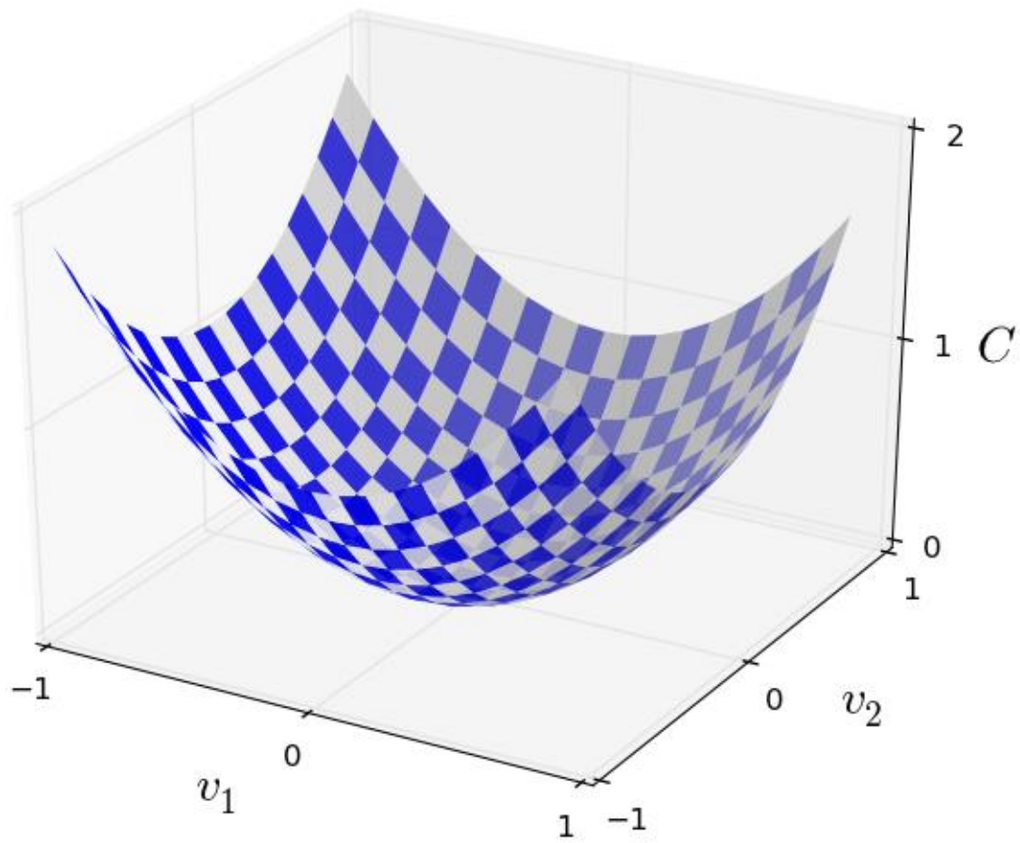
- The ball moves along a certain *distance* Δv along the *gradient direction* ∇C!
- The gradient is the direction provided by the partial derivatives along v_1 and v_2
- For our two-dimensional function, the gradient direction is:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)$$

- The change in function value C is:

$$\Delta C \approx \nabla C \cdot \Delta v$$

i.e. move a certain distance Δv *along the gradient*



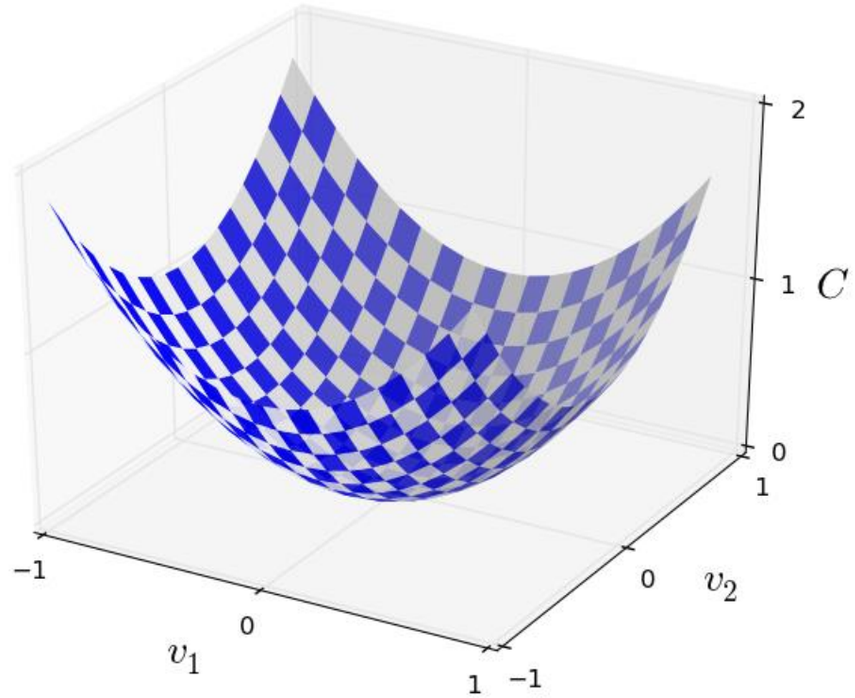
Gradient Descent - Intuition

- The change in function value C is: $\Delta C \approx \nabla C \cdot \Delta v$

- Goal: make C decrease → make ΔC negative so that: $C_{new} = \mathrm{C} + \Delta C$
- Set: $\Delta v = -\eta \nabla C$
  - Parameter η is the learning rate!
- It follows:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

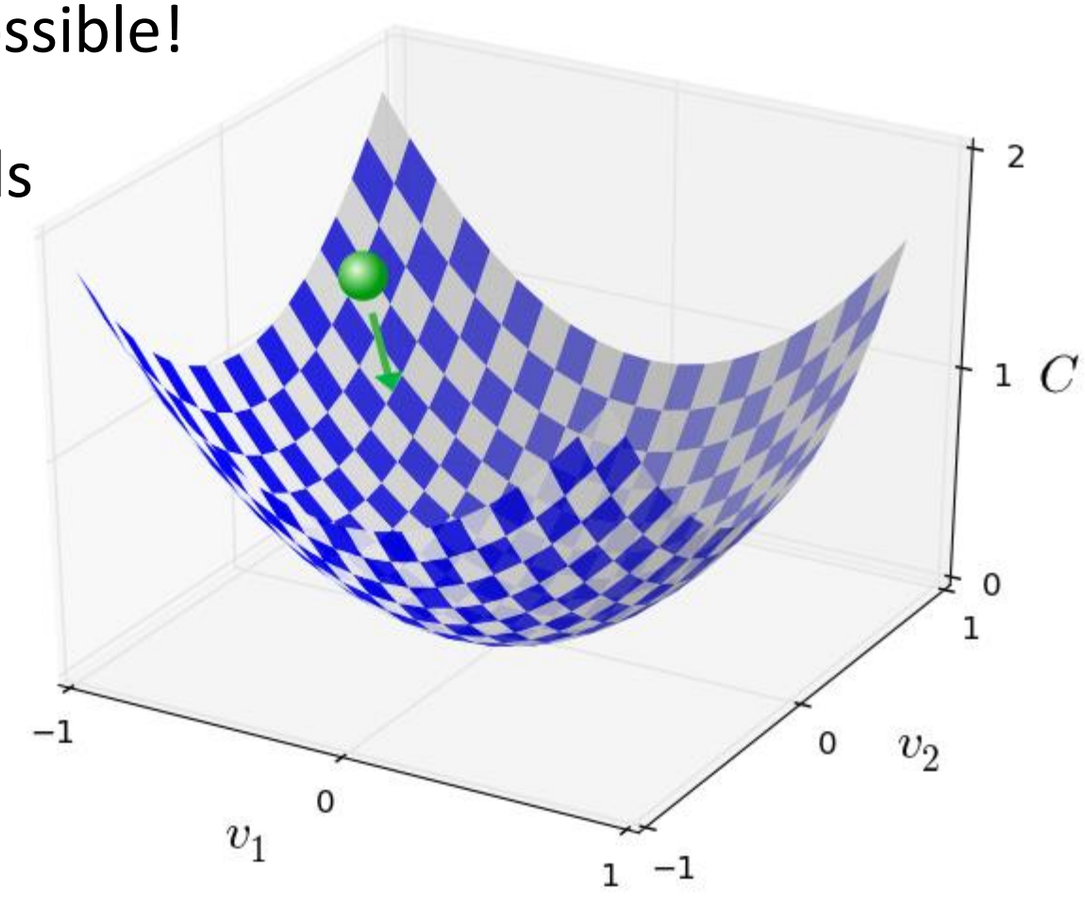
- Thus ΔC is always negative!
- Thus: $C_{new} = \mathrm{C} + \Delta C$ will always decrease



$$v \rightarrow v' = v - \eta \nabla C$$

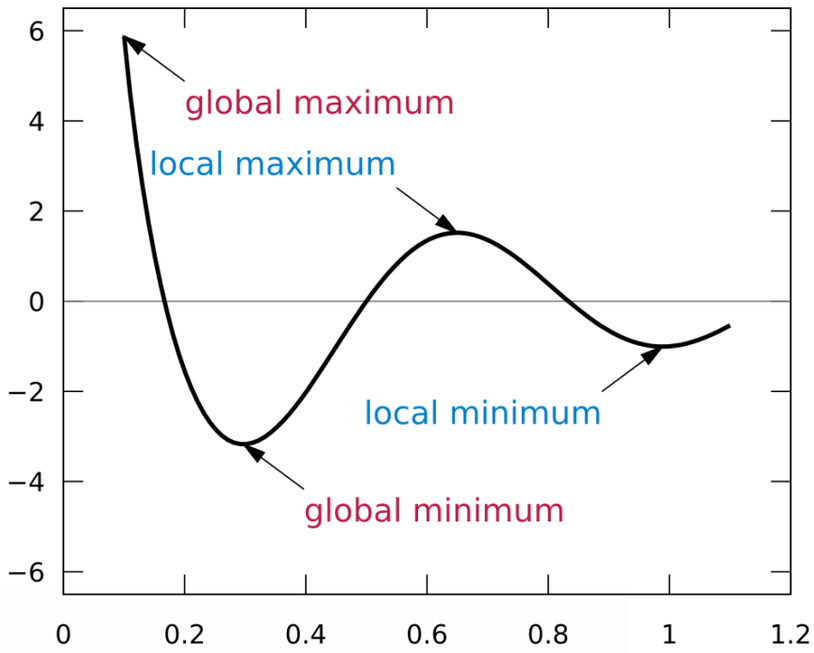
Gradient Descent - Intuition

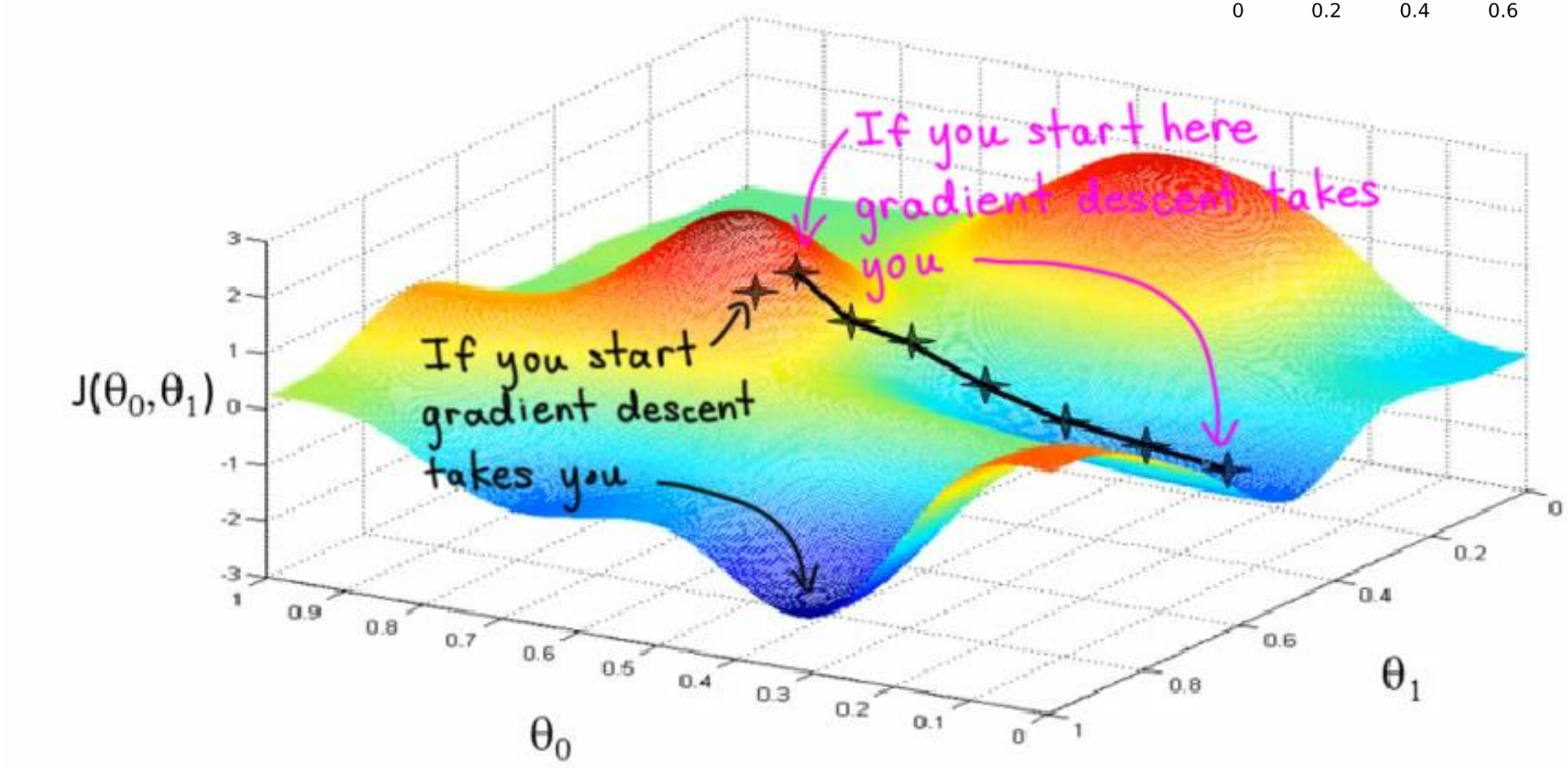
- What works in two dimensions works also in N dimensions, remember our function C(w,b)
- Gradient descent will continuously decrease the value of C (the cost and thereby the number of false classifications)
- Gradient descent is optimal in the sense that it always moves the ball into the direction where C decreases a much as possible!
- BUT: gradient descent not necessarily finds the optimal minimum. It may get stuck in a ***local minimum***!



Gradient Descent - Intuition

- Local minima:





Gradient Descent at a certain Neuron

- Remember our cost function that we want to minimize:

$$C(w,b) = \frac{1}{2n}\sum_x \|y' - f(w,b,x)\|^2$$

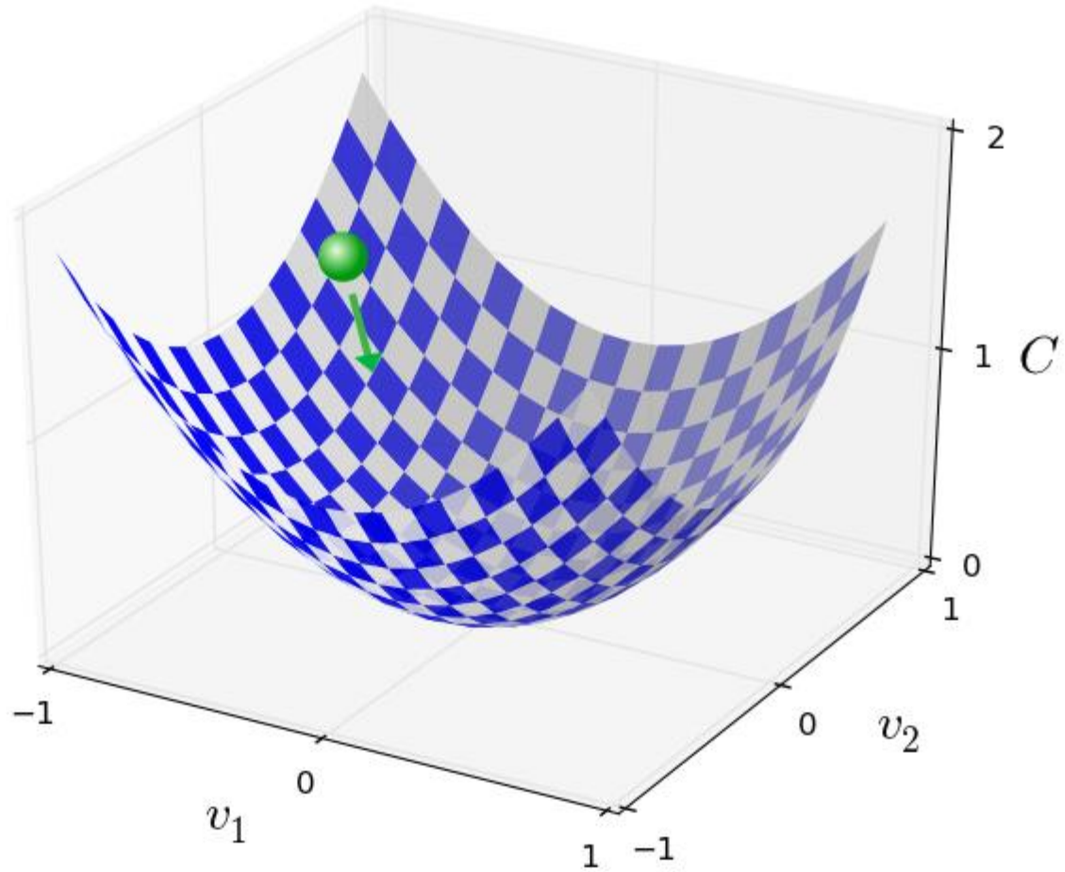
- w is a vector of all weights of the network w=(w_1, w_2, w_3, ... w_W) and b a vector of biases b=b(b_1, b_2, b_3, ... b_B). Thus, there are W*B free parameters in C

- This results in a W*B-dimensional surface

- Now again, find the direction of steepest descent (as in the 2-dimensional case)
  - replace v_1 and v_2 from before by our weights and biases
  - update rule is:

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$



Gradient Descent for an entire Network

- Previous slide: if we know the partial derivatives at a certain neuron with respect to the network output, we can optimize its weights.

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

- How to get the partial derivative at each neuron?
  - They **cannot be computed independently** (because they depend on each other)
  - But: all neurons contribute to *one* output.
  - Thus: we can compute them in a *reverse manner* starting from the output and computing the partial derivative of the second last neuron. Knowing the partial derivative of the second last neuron, we can compute that of the third last neuron using *the chain rule of derivation*.
  - This can continue until we are at the inputs.
  - The algorithm that does this is the ***Backpropagation Algorithm***

Remember: The Chain Rule

$$\frac{df(u)}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

- So, f is a function of u and u a function of x, i.e. u = g(x)
- This means y = f(u) and u = g(x), thus we can also say: $f(g(x))$
  This is a nested function (remember: *"innere Ableitung"* vs. *"äußere Ableitung"*)
- The chain rule says: compute simply the product of both derivatives:

$$f(g(x))' = g'(x)*f'(g(x))$$

- This is: $g'(x) = u' = \frac{dg(x)}{dx} = \frac{du}{dx}$
- And: $f'(x) = \frac{df(g(x))}{dg(x)} = \frac{df(u)}{du}$

$$\Big\} \; f(g(x))' = \frac{df}{du} * \frac{du}{dx}$$

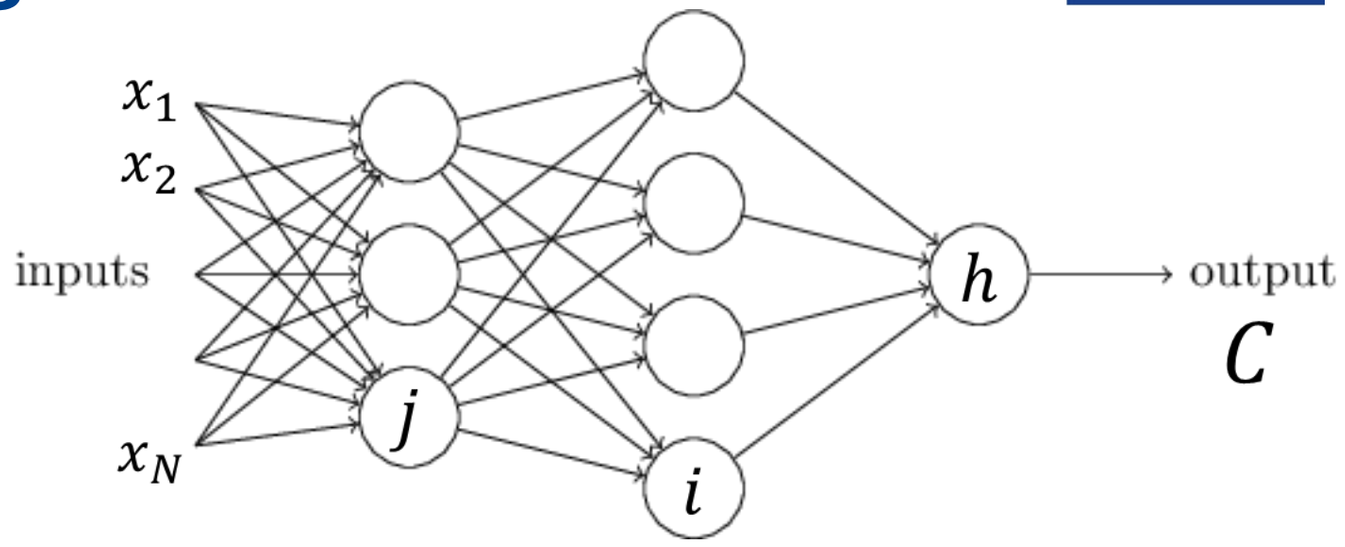
credit: https://www.symbolab.com/cheat-sheets/Derivatives#

Training a network with gradient decent

- For the training of a neural network the gradient descent algorithm is applied for all neurons of the network (from output to input) recursively.
- This process is called *back-propagation*
- The result of back-propagation is a network with minimized cost function, i.e. a network whose predictions y = f(x) are closer to the true values y'!
  - Remember that the result we get is usually not the global minimum but a **local minimum**! Different strategies exist to avoid getting stuck in local minima.
- If you want to know how exactly that works, see https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ for a nice hands-on example

The Backpropagation Algorithm

- A neural network is a very deeply nested function, i.e. the cost function C is a multiply nested function C(h(i,....(j(x_1,x_2,...,x_N)



- To estimate the derivative of C we can successively use the *chain rule of derivation* (note: we could also compute the derivative analytically but this would be enormously expensive for e.g. 60M parameters as in todays networks)

- *Backpropagation* solves this step-by-step by starting at the output (h) and moving recursively back towards the inputs.

- Backpropagation estimates the derivation at each node. All derivations together are the *gradient*, which tells us where to move on the high-dimensional surface

- **Intuition behind: the "derivative at a node" tells us how a change of this node changes C in the end, e.g. how much does a change in x_2 affect the output C**.

The Backpropagation Algorithm

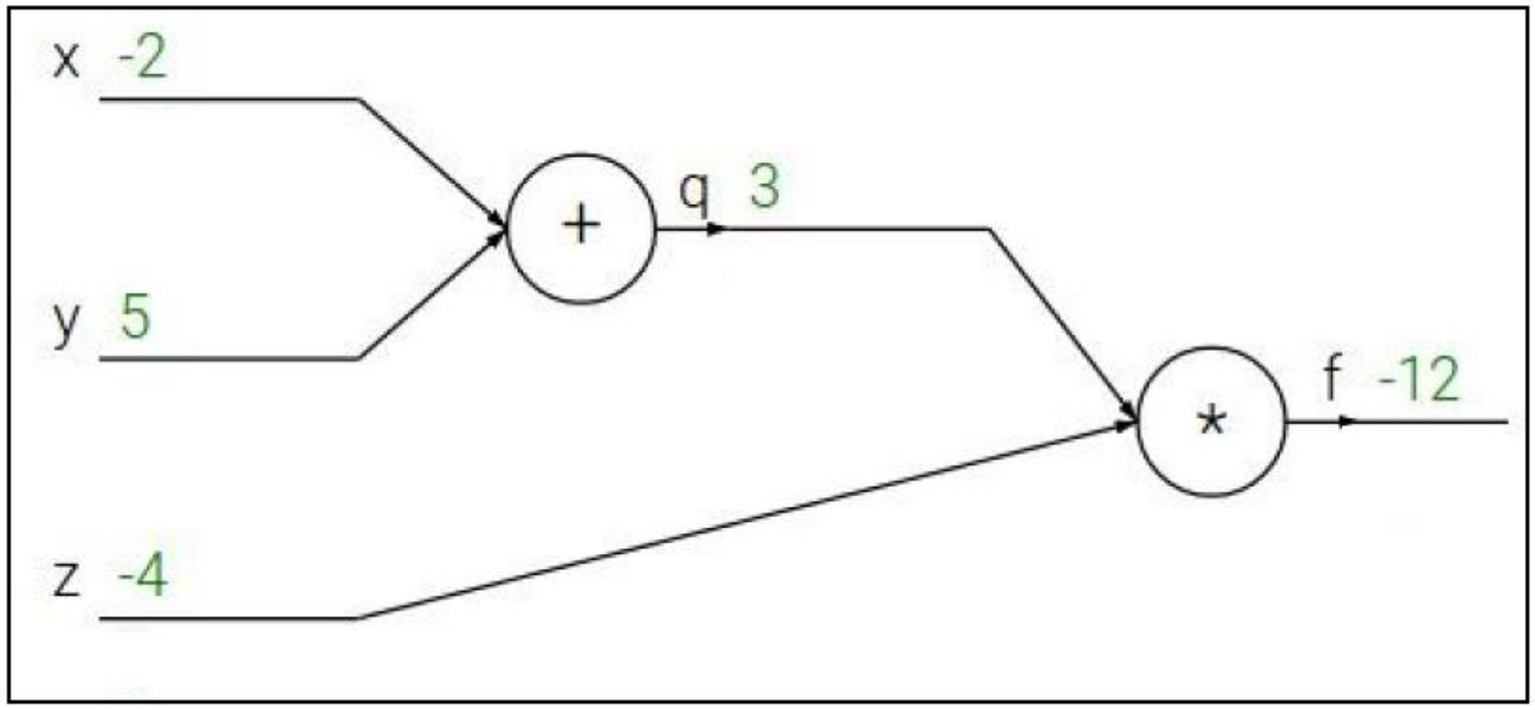
1. Attach input to the network (compute activations of the input nodes)
2. Perform a feedforward pass through the entire network (compute activations of all following nodes)
3. Compute the loss / cost function for the output (use the labels from the ground-truth as reference)
4. Back-propagate the loss recursively from output to the inputs → for each weight and bias get the derivative of the loss with respect to each parameter
5. Update the weights (and biases) by **adding** the derivative multiplied by the *negative* learning rate to them (at each node separately)

Example:
Computing the partial derivatives of the loss function

The Backpropagation Algorithm

Computational Graphs

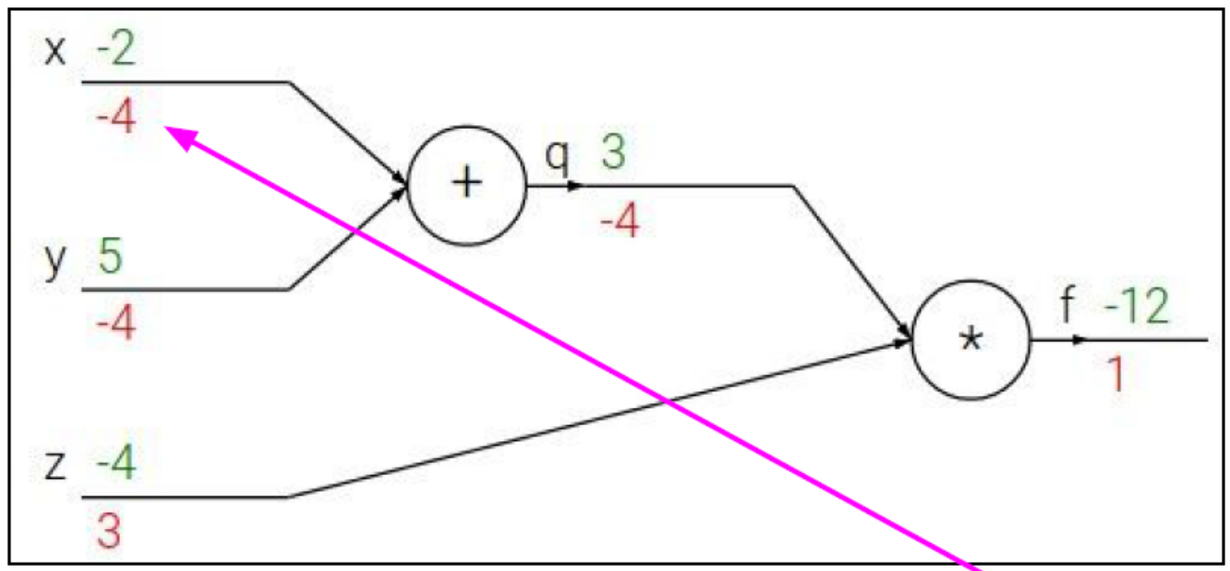
- An effective helper..



Let’s try it out!

Backpropagation - Example

- Backpropagation is a recursive application of the chain rule!
- Step 1: Do a forward pass to fill the graph with values (green values)
- Step 2: Start from the end of the graph and go backwards step by step and compute gradients (= partial derivatives), here the red values

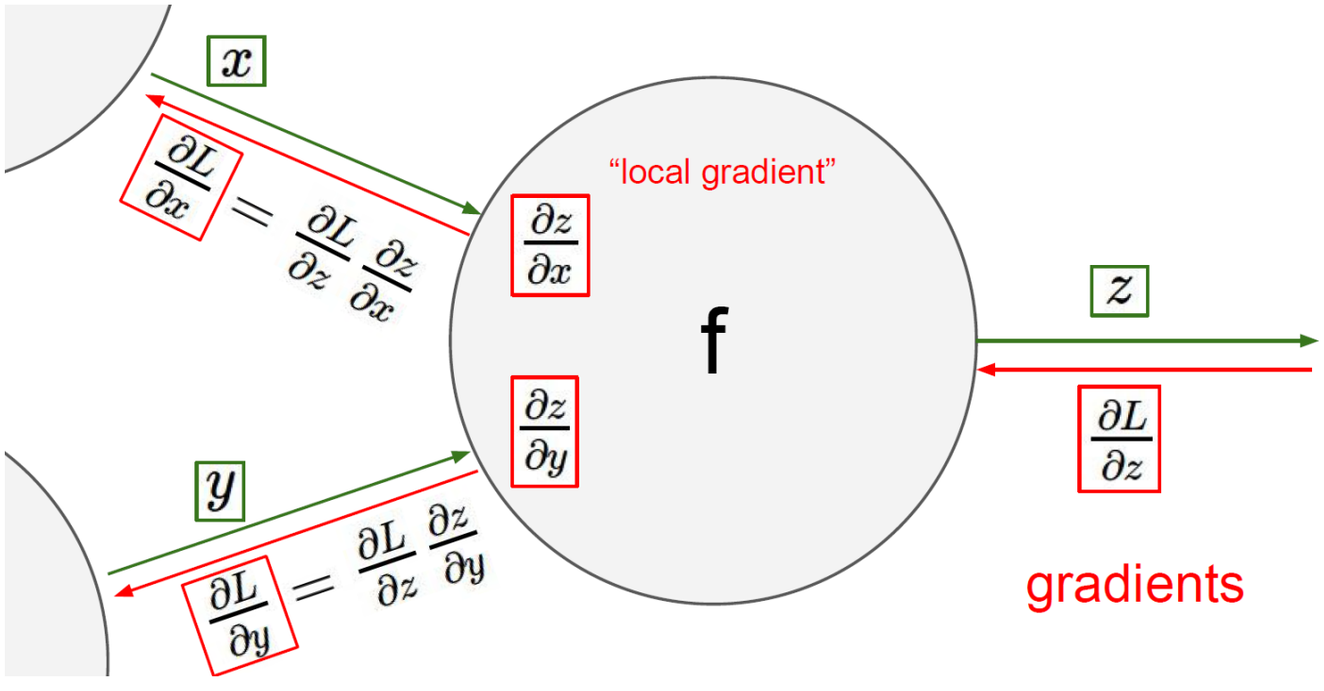


Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Backpropagation

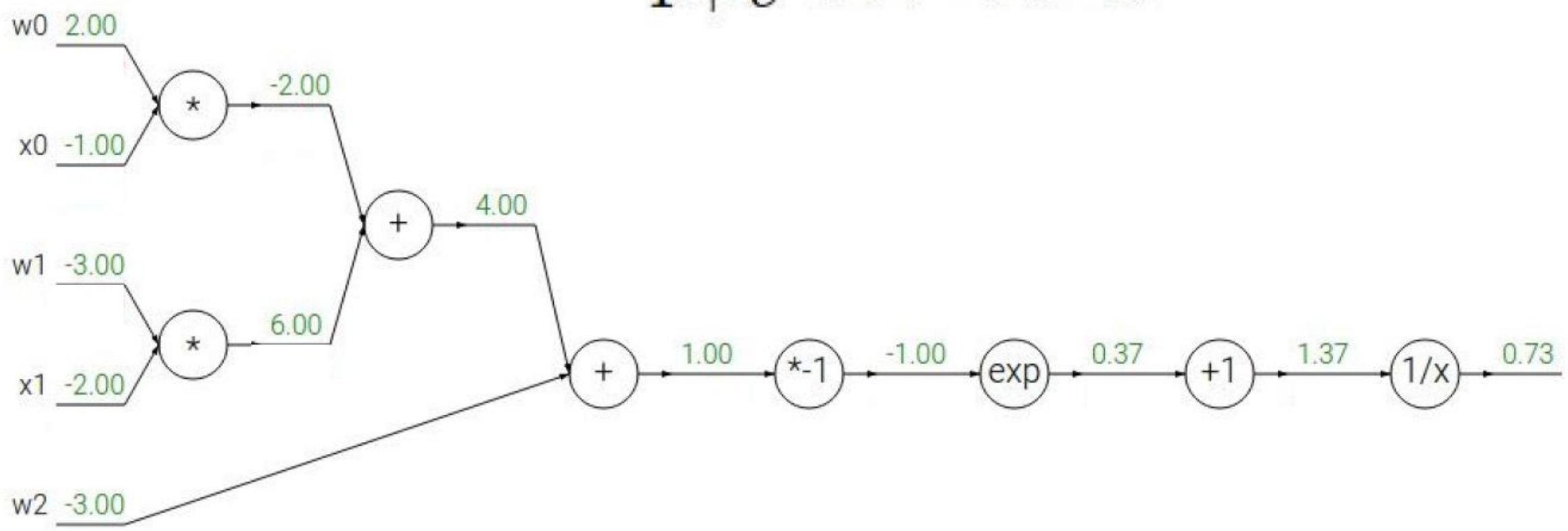
- Calculate for each node based on the result of successive node
- At each node we have the (relatively simple) local gradient.
- Multiply it with the global gradient and go on with the next node
- Overall, we get the gradient for the entire computational graph, step-by-step



- Backpropagation provides all partial derivatives of L with respect to any weight and bias term in the network. The partial derivative tells us: "how much does L change, when I change the current weight a little bit"?

Backpropagation – Another Example

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

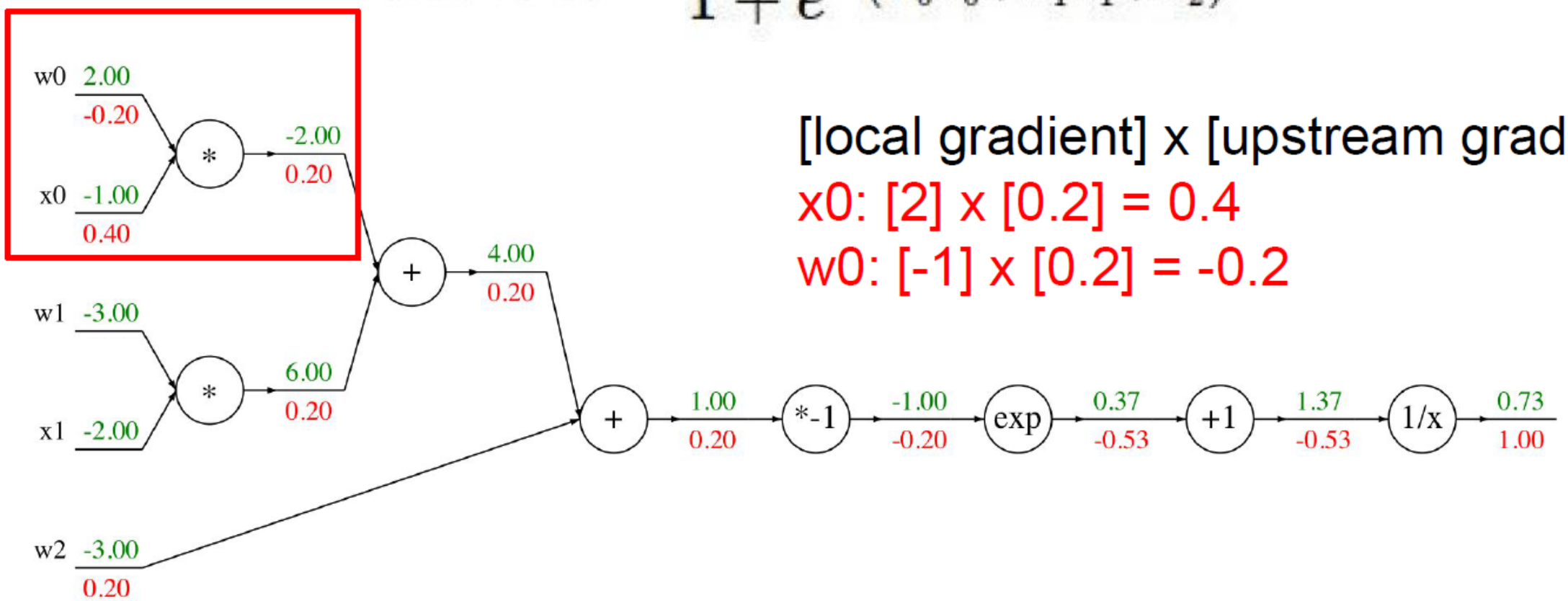
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Backpropagation – Result

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



[local gradient] x [upstream gradient]

x0: [2] x [0.2] = 0.4

w0: [-1] x [0.2] = -0.2

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

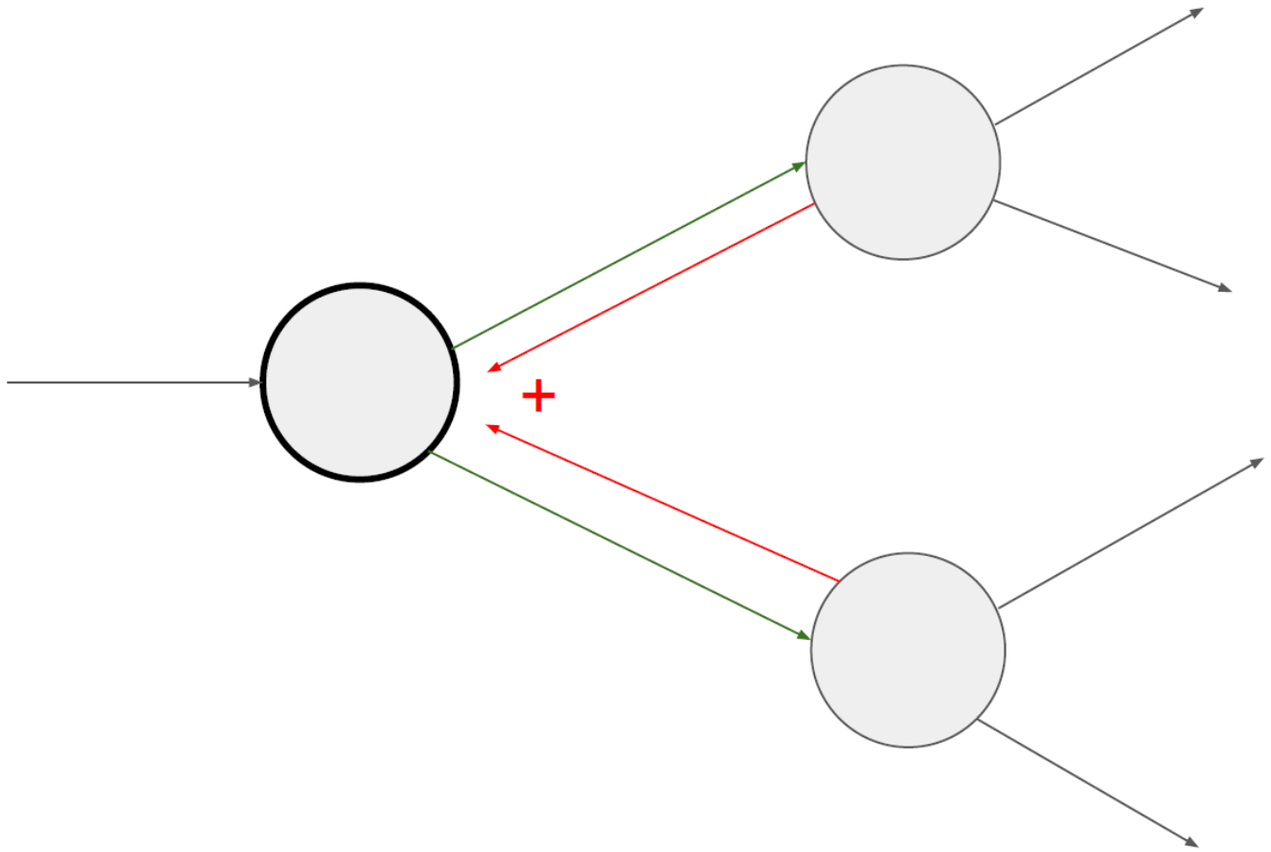
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

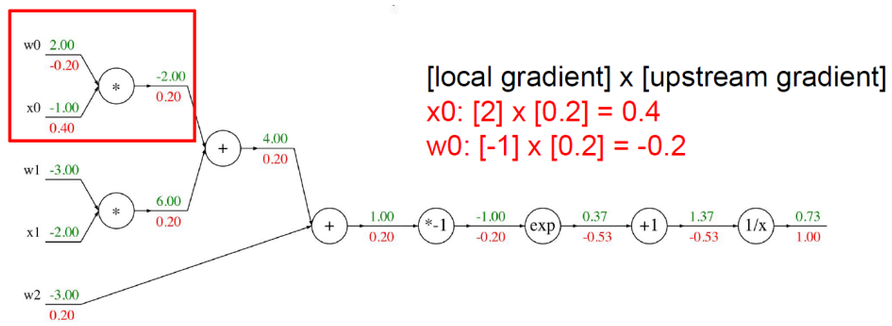
Backpropagation at Branches

- Add the two upstream gradients together to get the new upstream gradient!



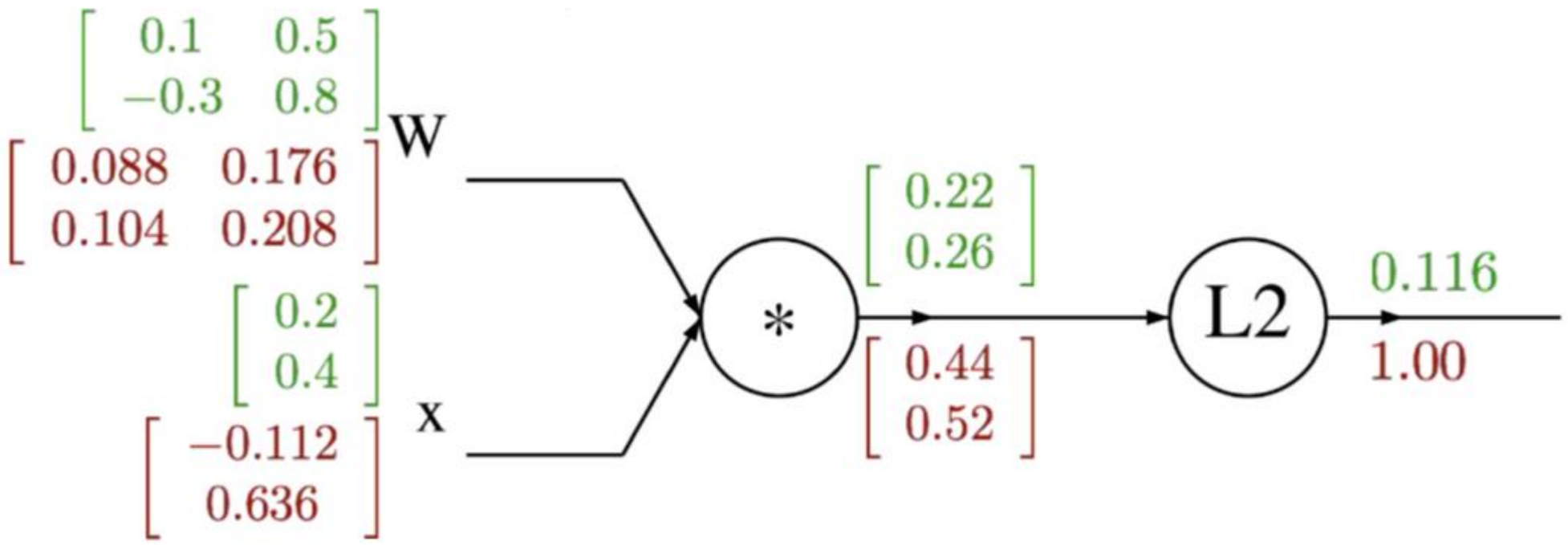
Backpropagation on Matrices

- In our example, we worked only with scalars. This corresponds to computing the back propagation for each weight / bias separately



- In practice, we compute backpropagation for entire matrices, e.g. for an entire neuron with W weights and B biases at once.

- E.g.:



Backpropagation on Matrices

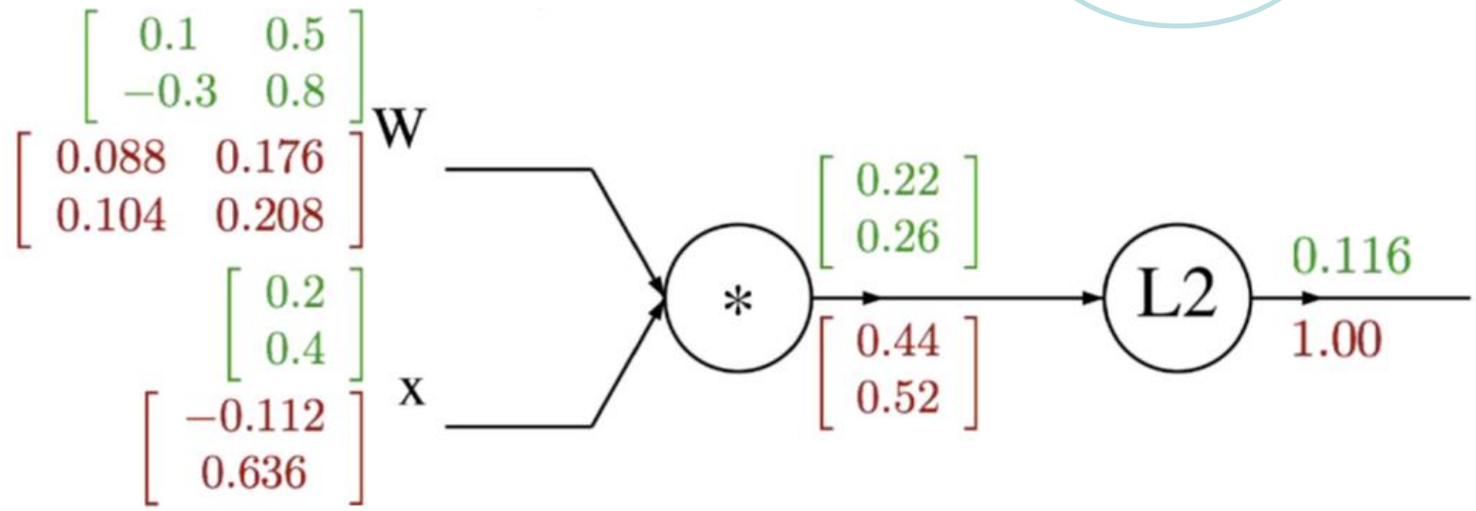
- How to differentiate this?
- Principle is the same: chain rule!
- Chain rule can be applied to matrices and vectors as well!

vector/ matrix

$$\frac{df}{dx} = \frac{df}{du} * \frac{du}{dx}$$

vector/ matrix

vector/ matrix



Backpropagation on Matrices

- Chain rule reduces to a dot product (for vectors) or a matrix multiplication for matrices
- Works even for arbitrary high-dimensional matrices (e.g. 4D-matrices). Such matrices we call **Tensors**!
- Chain rule in this form is extremely efficient!

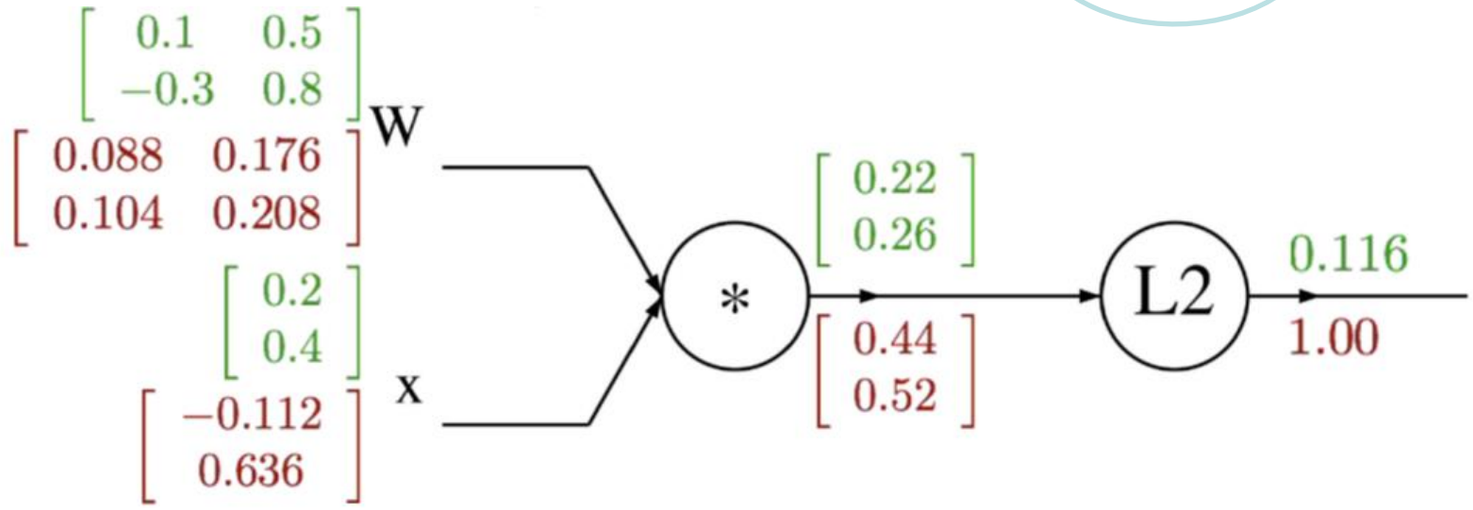
that's the reason why graphics card work so nicely with deep learning

vector/ matrix

$$\frac{df}{dx} = \frac{df}{du} * \frac{du}{dx}$$

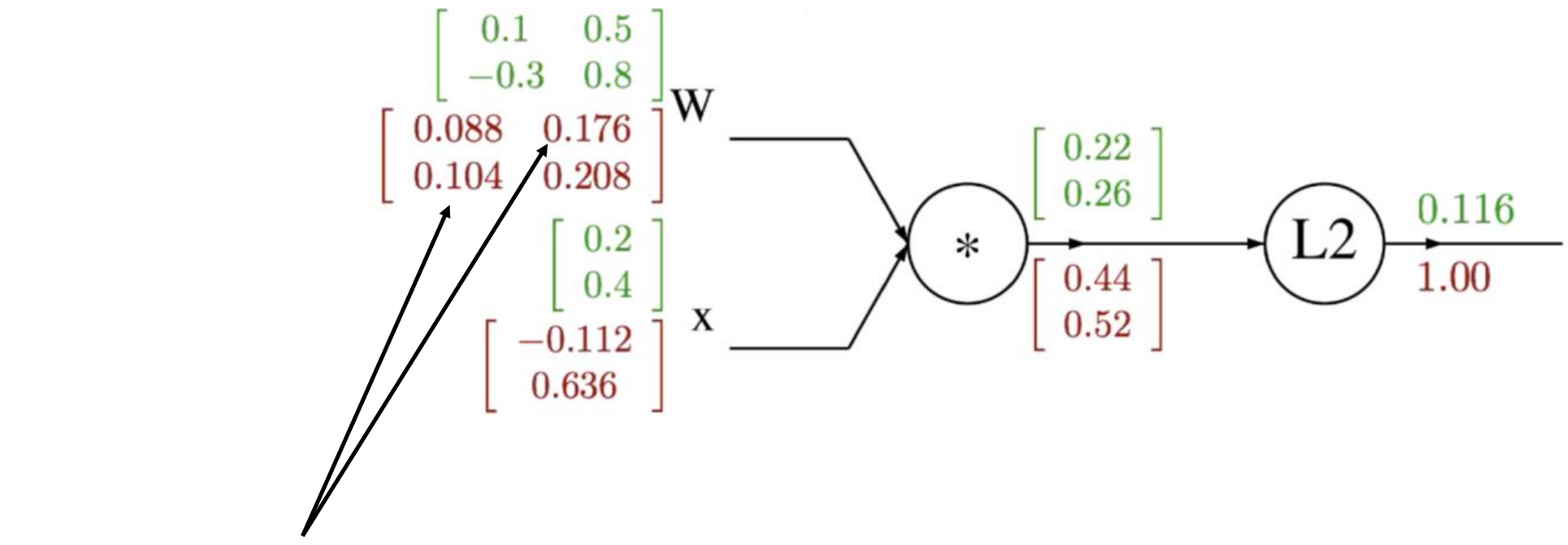
vector/ matrix

vector/ matrix



Backpropagation on Matrices

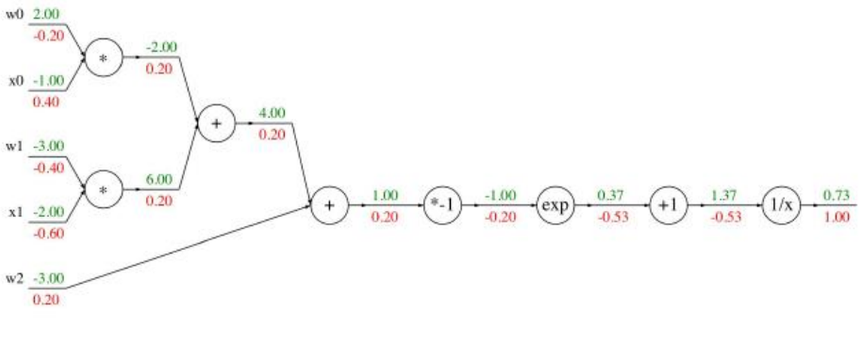
- Example: online lecture 4: https://www.youtube.com/watch?v=d14TUNcbn1k&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=4
- But beware: there is a typo in the video of year 2017
- The right solution is this one:



these two values are mixed up in the video, the rest is ok!

Backpropagation - Implementation

Modularized implementation: forward / backward API

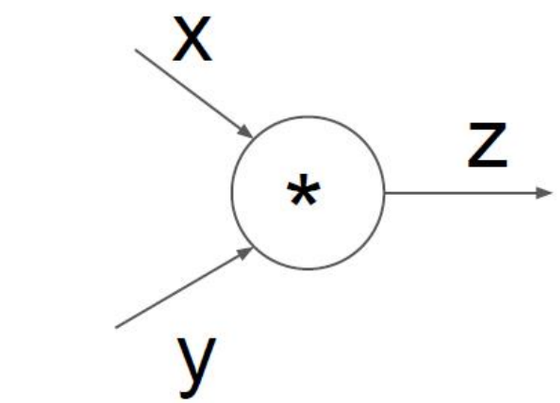


Graph (or Net) object *(rough psuedo code)*

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Backpropagation - Implementation

Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$\frac{\partial L}{\partial z}$

$\frac{\partial L}{\partial x}$

Summary on Backpropagation

- Backprop can estimate the gradient for an entire network
  - step-by-step
  - in a recursive manner
  - from the outputs to the input
- For every weight and bias in the network, we get it's influence on the output (the loss). Knowing this influence, we can change the weights accordingly to minimize the output (the loss)
- **Back propagation is the standard approach for gradient descent today!**
- The whole story: http://neuralnetworksanddeeplearning.com/chap2.html
- If you want to know how exactly that works, see https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ for a nice hands-on example

So far: Stochastic Gradient Descent

- Solution: apply back propagation for a single input (e.g. image), i.e.:
  1. feed **one** image into the network,
  2. compute back propagation for it,
  3. update weights,
  4. iterate
- What is the problem with that?
  - e.g. input cat image → learn → input dog image → learn → input ship image → learn ...
  - learning too sensitive
  - danger of unlearning

Alternative: Batch Gradient Descent

- Compute gradient for **all** images in training set
- Perform **one** update after all images have been processed
- Advantage:
  - Optimal in learning efficiency (the global gradient for the entire training set can be computed – and not only the gradient for one image)
- Disadvantage
  - Requires huge memory!
  - Would be nice but infeasible in practice

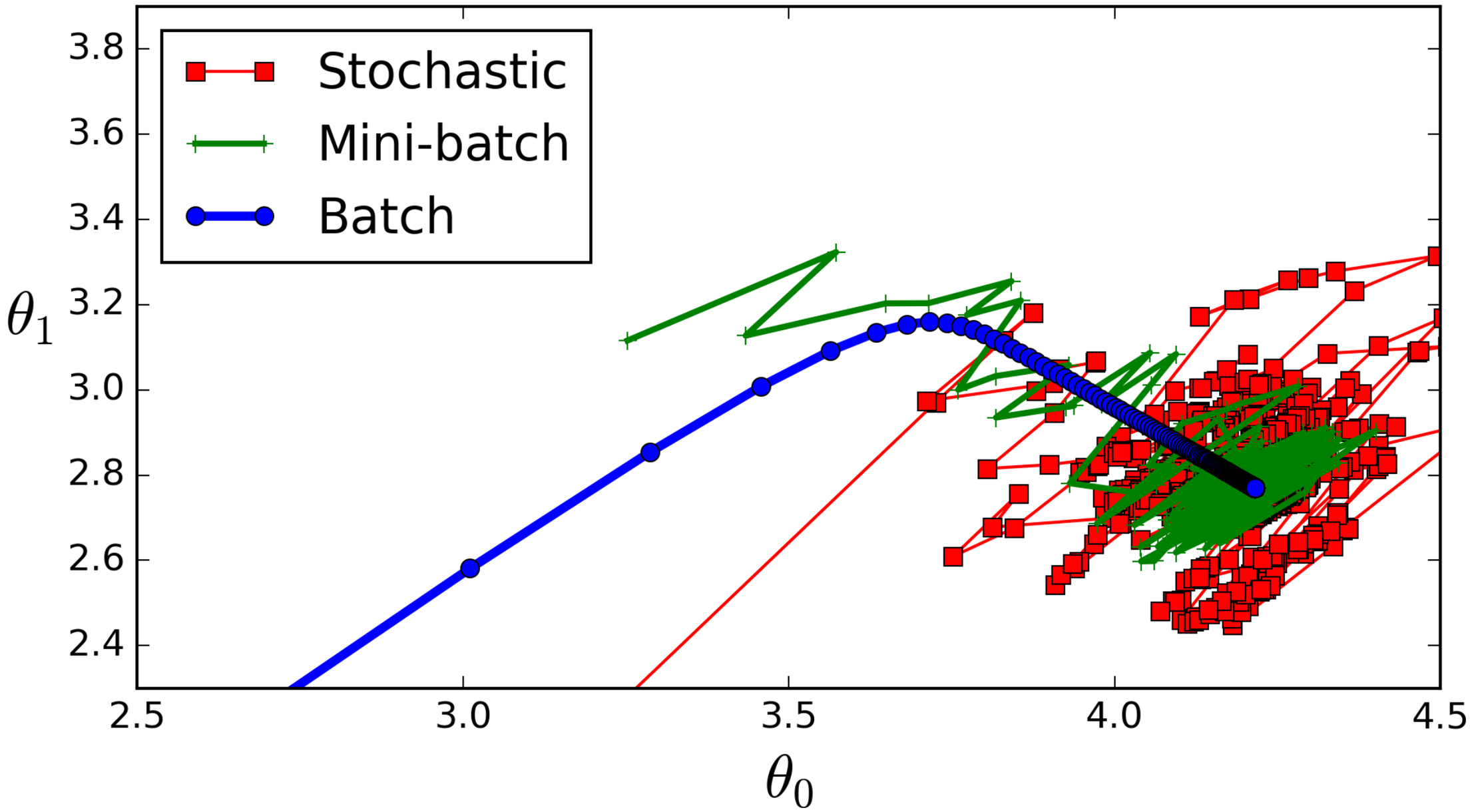
Tradeoff: Mini-batch Gradient Descent

- A tradeoff:
- Take **several** input images (a so called **mini-batch**)
- Feed in each image, compute loss and do backpropagation to get the gradient for **each** image
- **Store** the gradient for each image
- After all images have been processed, **average the gradient** (see also equation 19 here: http://neuralnetworksanddeeplearning.com/chap1.html#implementing_our_network_to_classify_digits)
- **Update** network with the average gradient
- The *batch size* parameter specifies the size of a mini batch

Mini-batch Gradient Descent

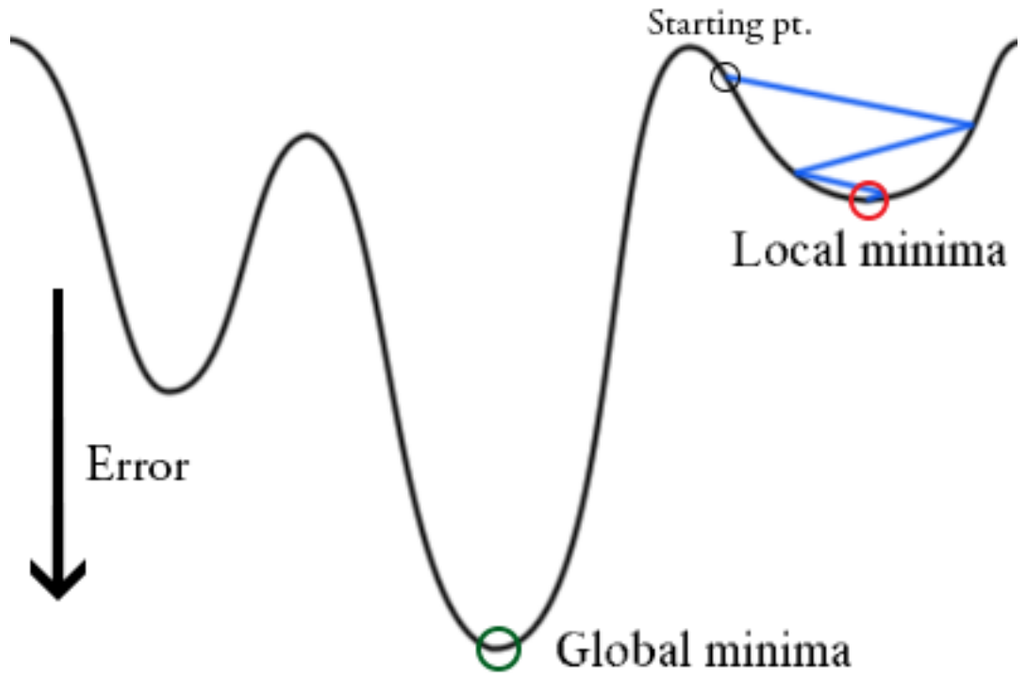
- Advantages
  - More robust convergence (avoids local minima)
  - Less updates → faster
  - Memory efficient
- Disadvantages
  - additional hyperparameter
  - tradeoff between batch size (more global learning) and memory consumption
- Today: mini-batch gradient descent is the standard method in deep learning!
- What's a good batch size? The larger the better usually. Set it according to your available memory. A good starting point is a batch size of e.g. 32.

Batch vs. Mini-batch vs. Stochastic Gradient Descent



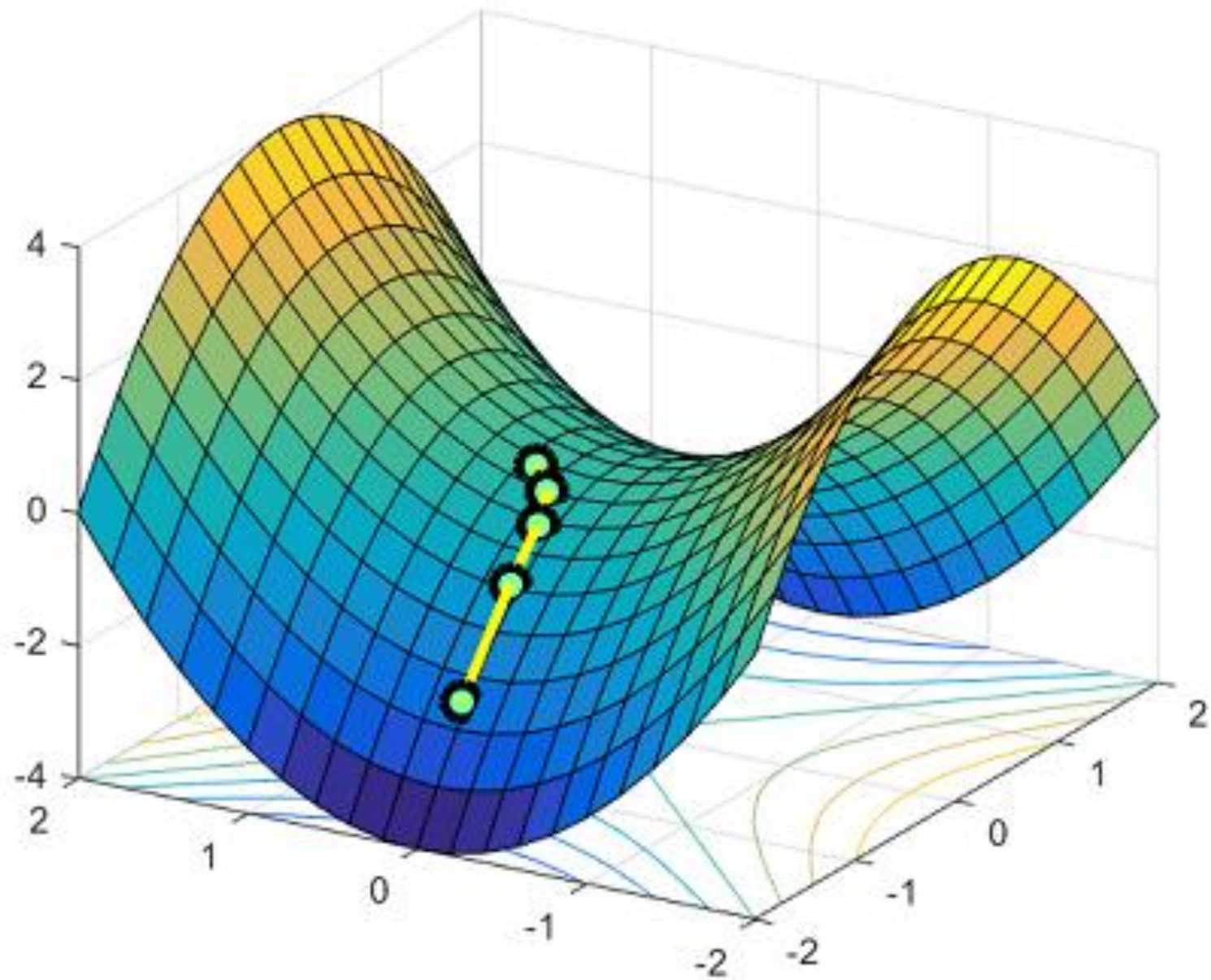
credit: http://deeplearning.buzz/2017/06/01/what-is-batch-size-and-epoch-in-neural-network/

Local Minima



- Do we find the global optimum?
- NO! Gradient descent finds a local optimum (minimum)
- What happens in a local minima? Steepest descent goes upwards → we cannot reduce the loss anymore! → we get STUCK!
- BUT: recent research has shown that the local minima found are almost all very good minima!
- WHY? Most local minima (where we could get trapped) are “saddle points”. This means that there is at least one direction to escape
- Intuition: the higher dimensional we get, the less likelihood is there for a “real” minima without a saddle, i.e. a minimum where all directions show in upwards direction is very rare (there is at least one direction where we can escape)

Local Minima



credit: http://www.offconvex.org/assets/saddle/escapesmall.png

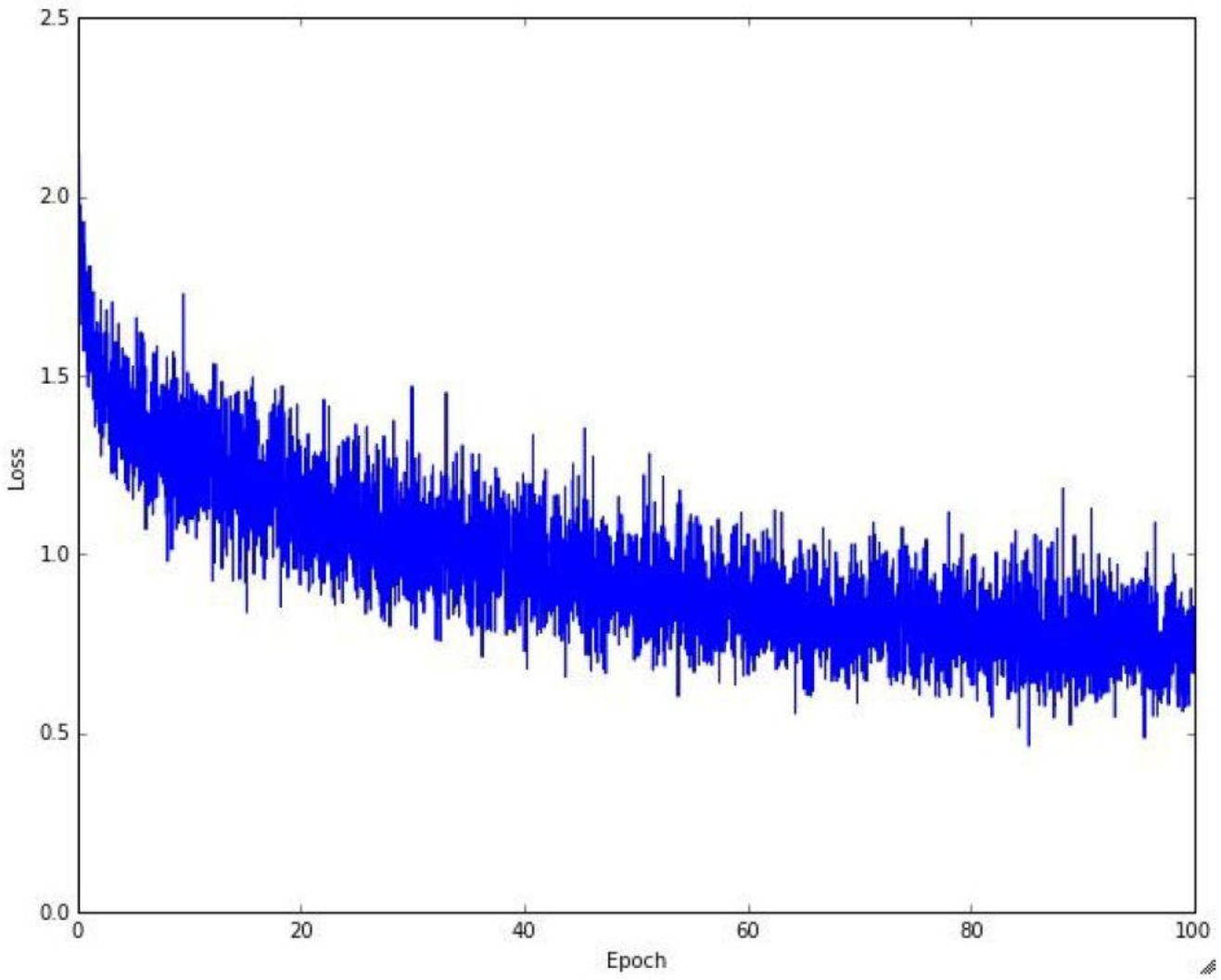
Hyperparameters

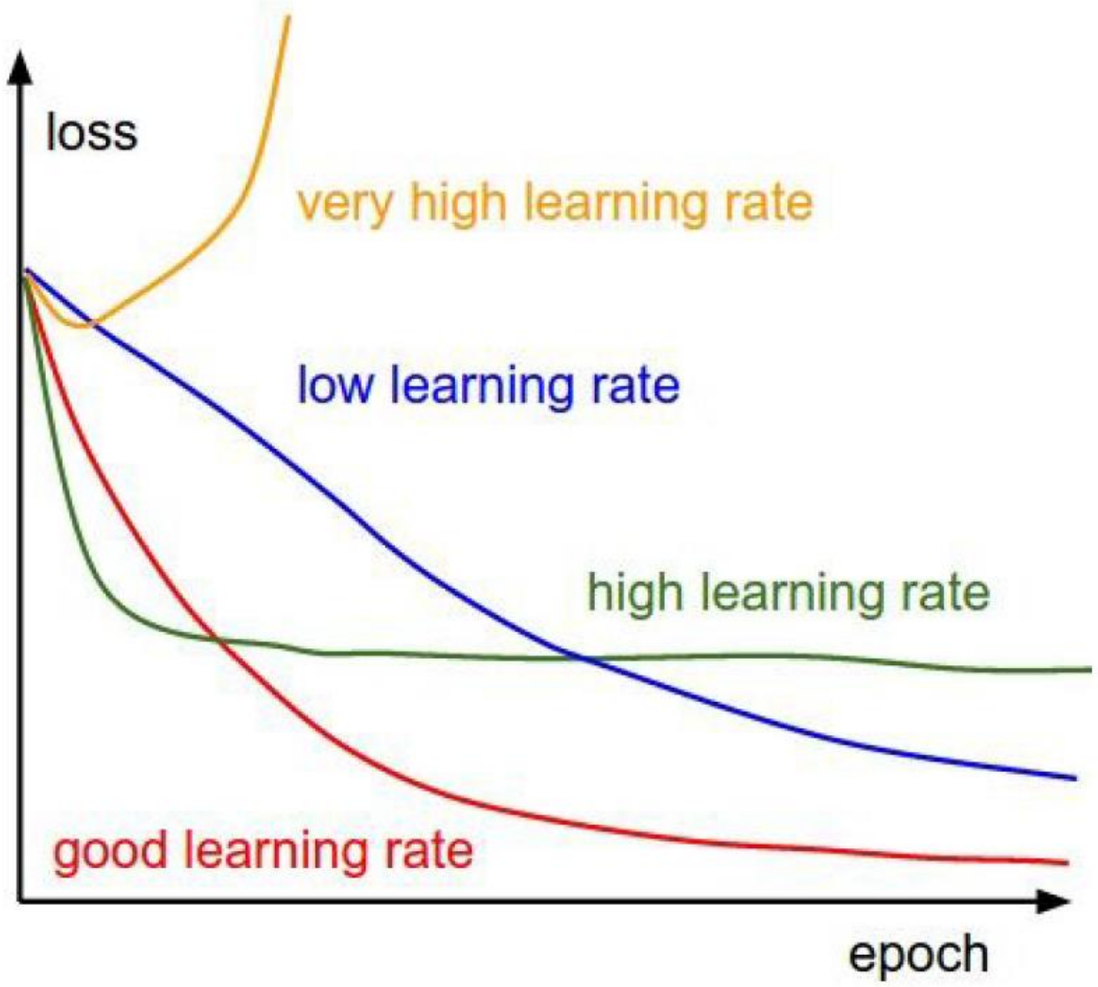
- Learning rate (most important): good starting point 0.01, if loss does no go down or explodes, decrease (e.g. 0,001)

credit: https://towardsdatascience.com/hyperparameters-in-deep-learning-927f7b2084dd

Learning rate

Monitor and visualize the loss curve





Hyperparameters

- Decay:
  - reduce learning rate over time
  - start with bigger steps (faster) and then reduce stepsize successively to support convergence
  - $learning\_rate = \frac{1}{1+decayRate*epochNum} * initial\_learning\_rate$

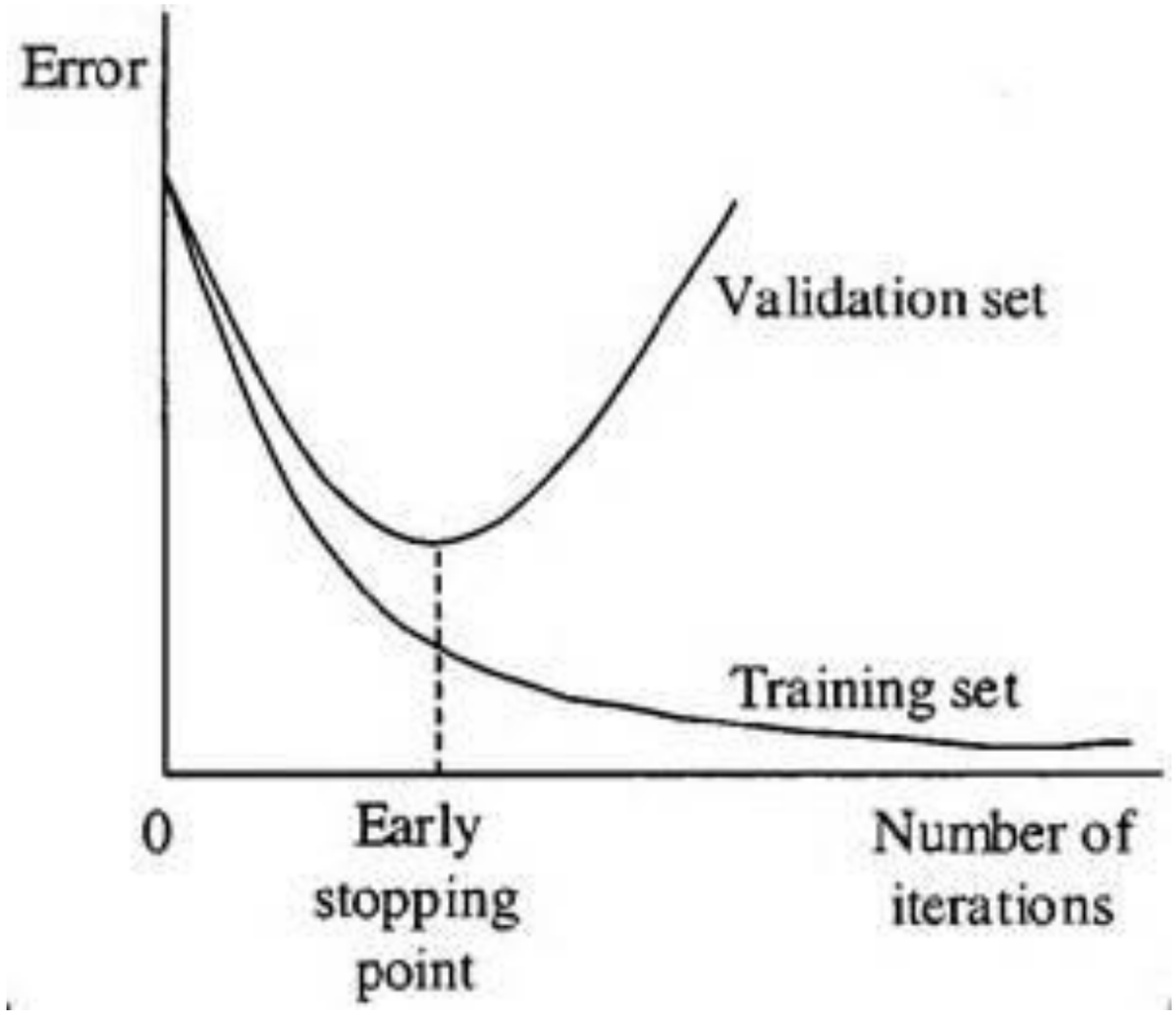
credit: https://towardsdatascience.com/hyperparameters-in-deep-learning-927f7b2084dd

Hyperparameters + When to stop training?

- Number of **epochs**

- Note on terminology
  - **iterations** = number of batches shown to the nework
  - Example: 1000 samples, batch size = 100 → 1 epoch has 1000/100 = 10 iterations

- Different **early stopping criteria** exist, e.g. *stop training if validation loss does not improve for 20 epochs*



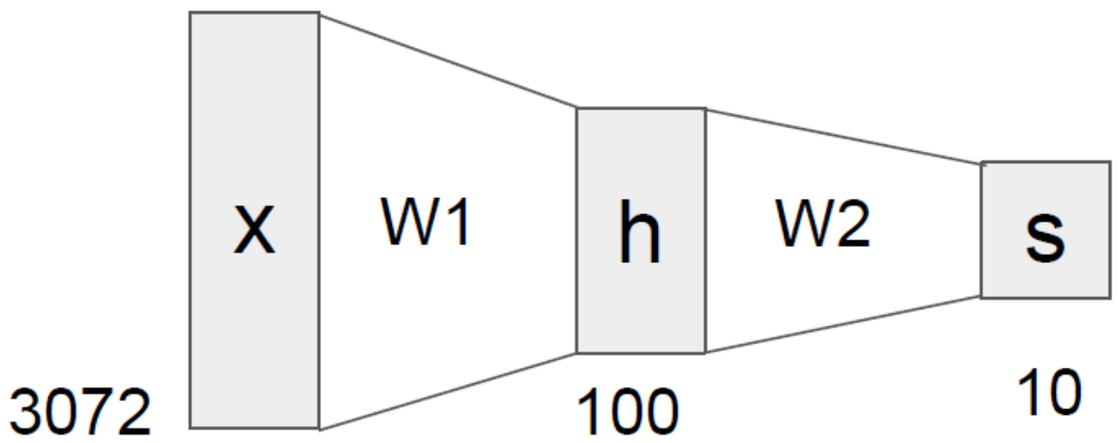
credit: https://towardsdatascience.com/hyperparameters-in-deep-learning-927f7b2084dd

Digitale Medientechnologien

Network Implementation and Training

Neural Networks as Matrices

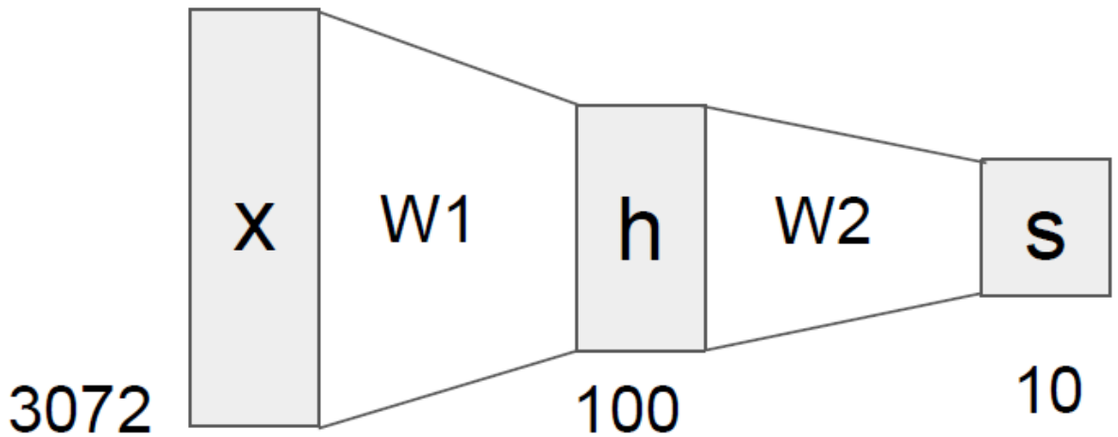
- The weights and biases of a neural network are stored numerically as vectors and matrices.
- A fully connected layer, for example, with 10 neurons and 5 inputs per neuron are stored as:
  - ???
  - 10x5 Matrix for the weights plus 10x1 vector for the biases



- What is the size of W1 and W2?
- Answer: W1: 3072x100 weights, W2: 100x10

Neural Networks as Matrices

- Computing the forward pass can be vectorized (parallelized) well:

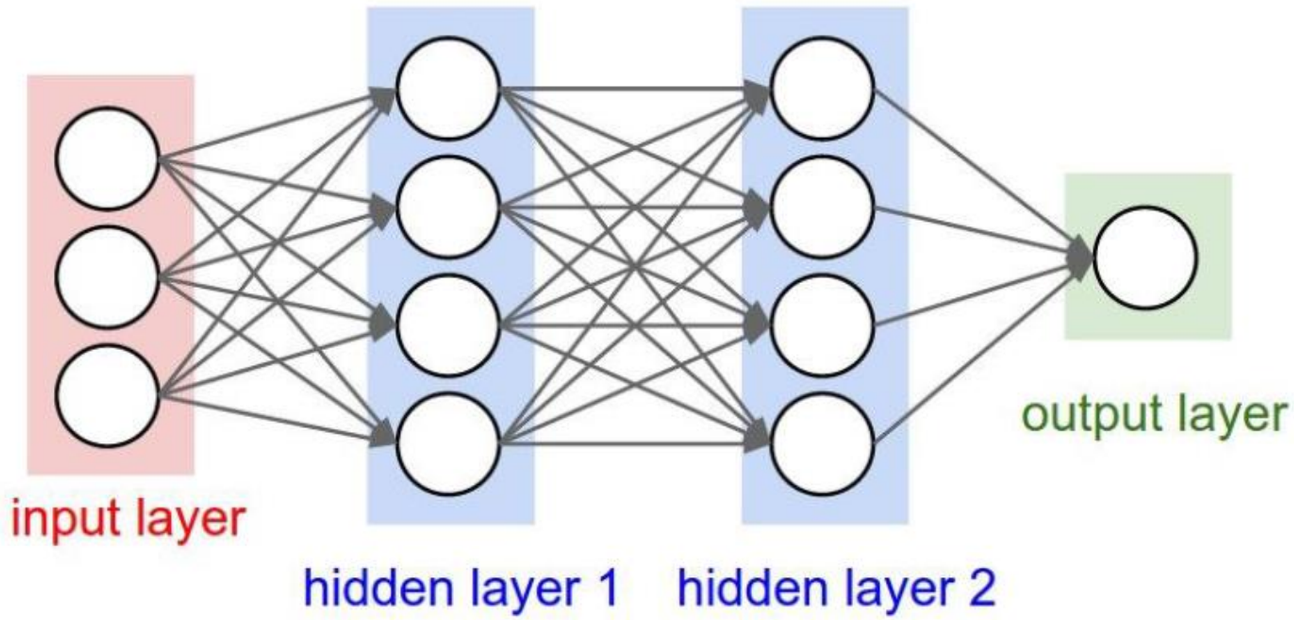


- for a linear layer: $h = W1x$
- for a layer with activation function:
  - $h = \sigma(W1x)$
- for the full network depicted above (with some activation function σ), e.g.
  - $s = \sigma(W2(\sigma(W1x))$
  - Example with ReLU ($\sigma(u) = \max(u, 0)$): $s = max(W2(max(W1x, 0), 0)$

note that this is a matrix multiplication!

Implementation in Python

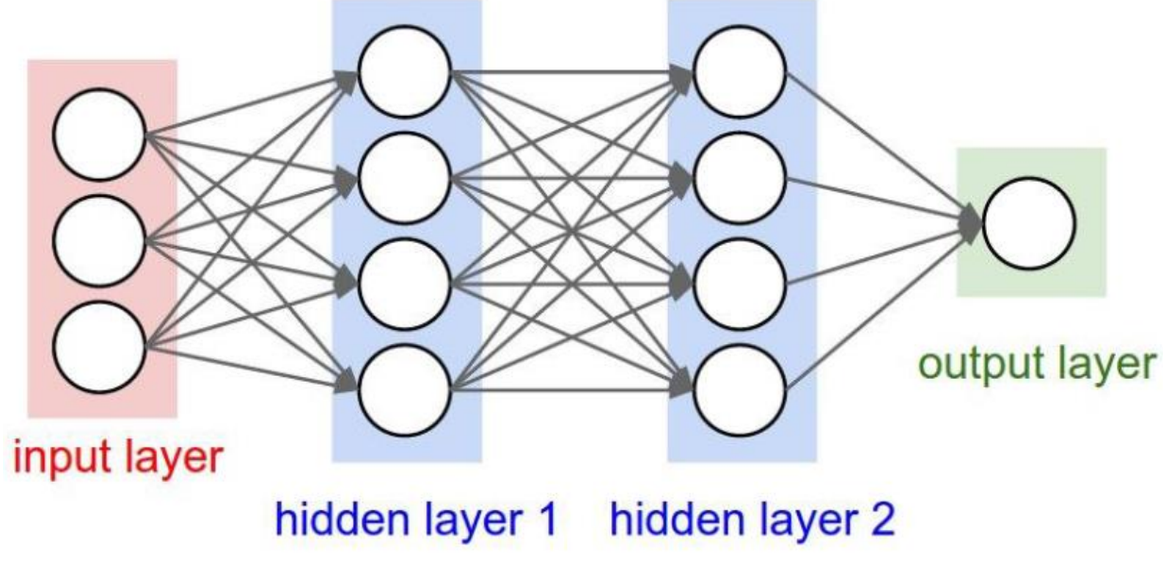
Example feed-forward computation of a neural network




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Terminology (often confused)

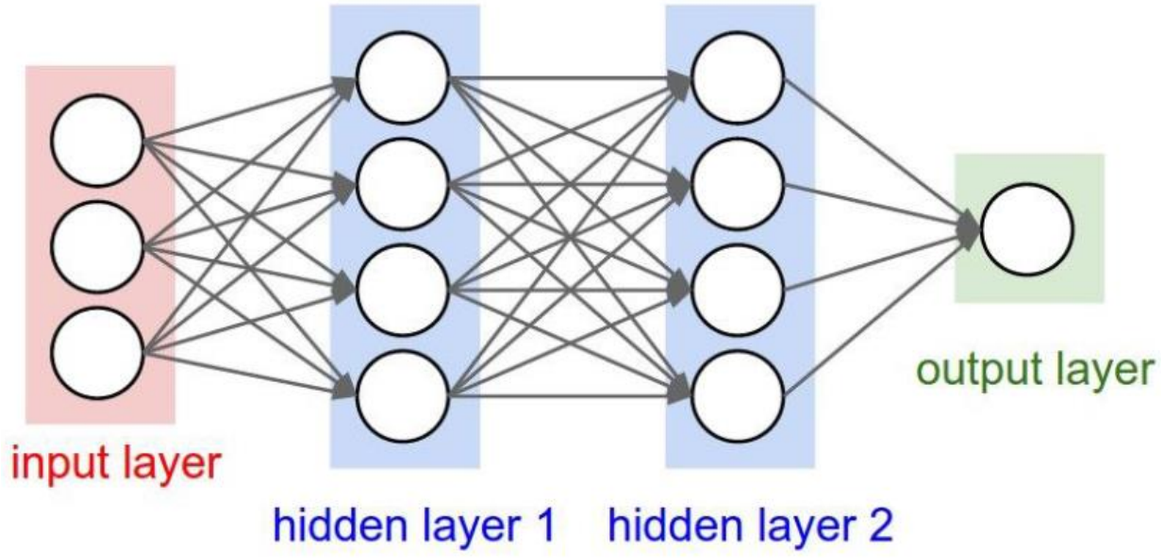
- Weights / Biases
  - Parameters that are **learned**
  - 1 value per weight / bias
- Activations
  - Values obtained at a neuron in the **forward pass** when feeding some input in
  - Activation = input * weights
  - One value per neuron
- Gradients
  - Values obtained at each neuron during the **backwards pass** (backpropagation)
  - One value per weight / bias

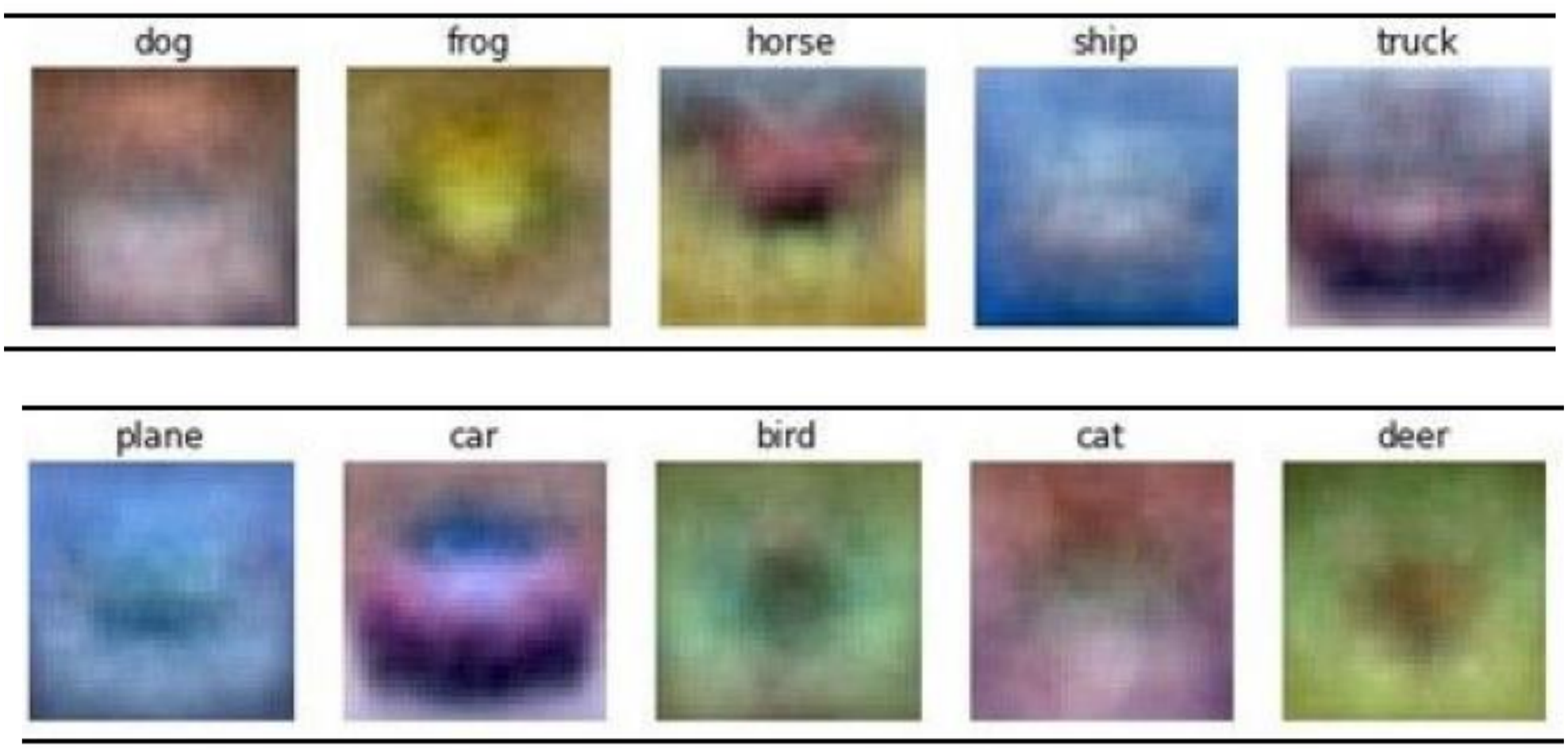


all these values are stored as matrices / vectors

Terminology (continued)

- Activation maps
  - the matrix of activations is often visualized as matrix for images





Digitale Medientechnologien

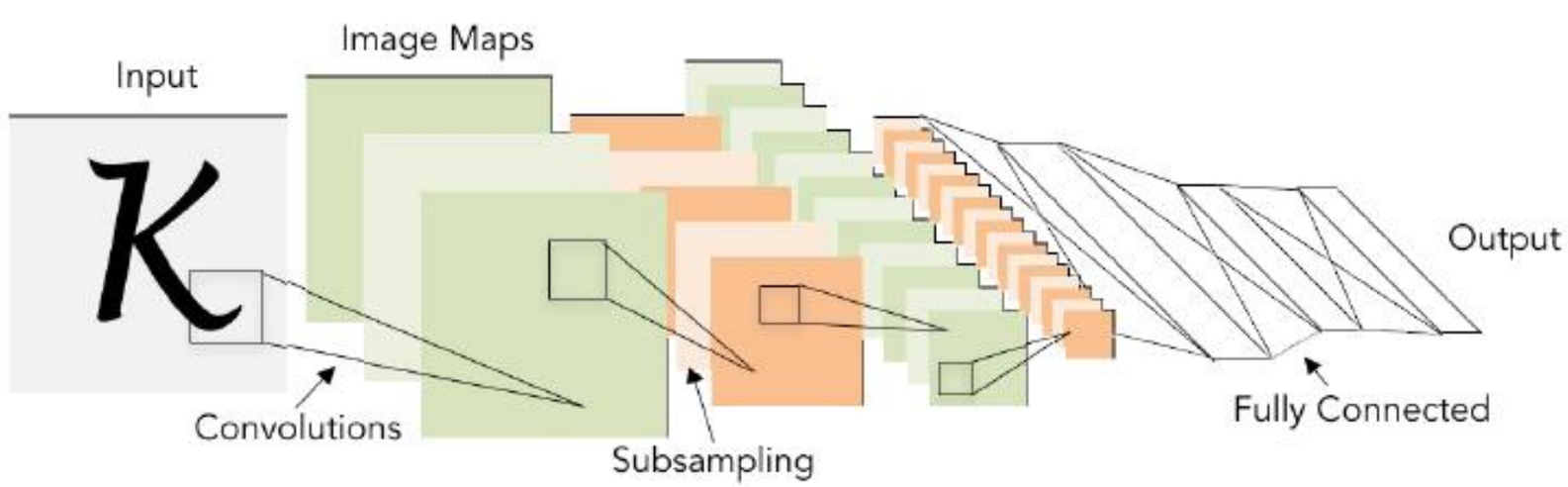
Convolutional Neural Networks

First CNN

A bit of history:

Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



LeNet-5

Breakthrough: AlexNet

A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

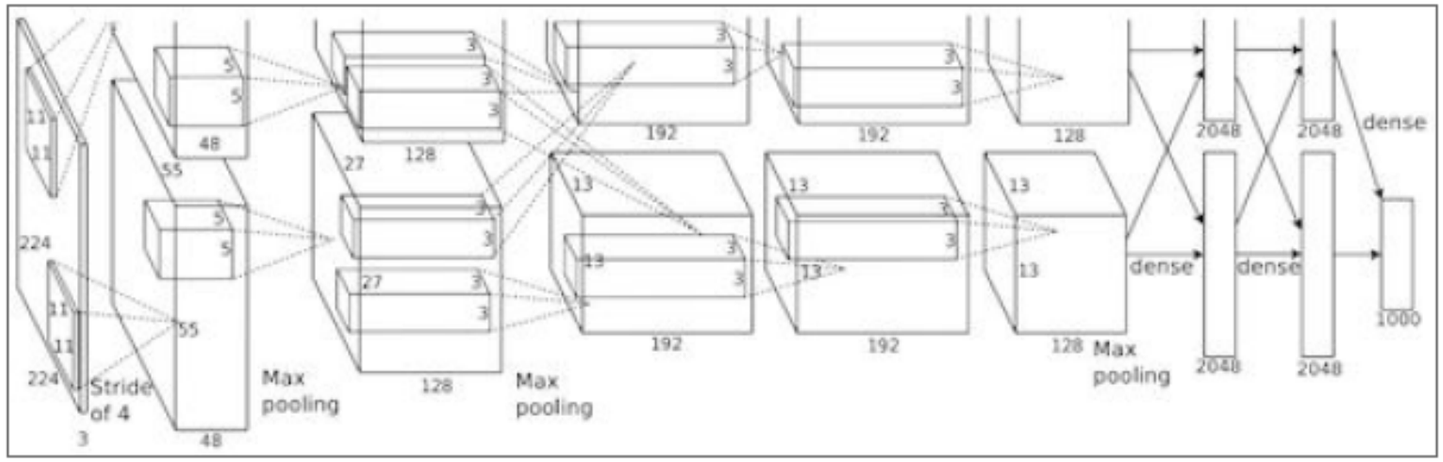
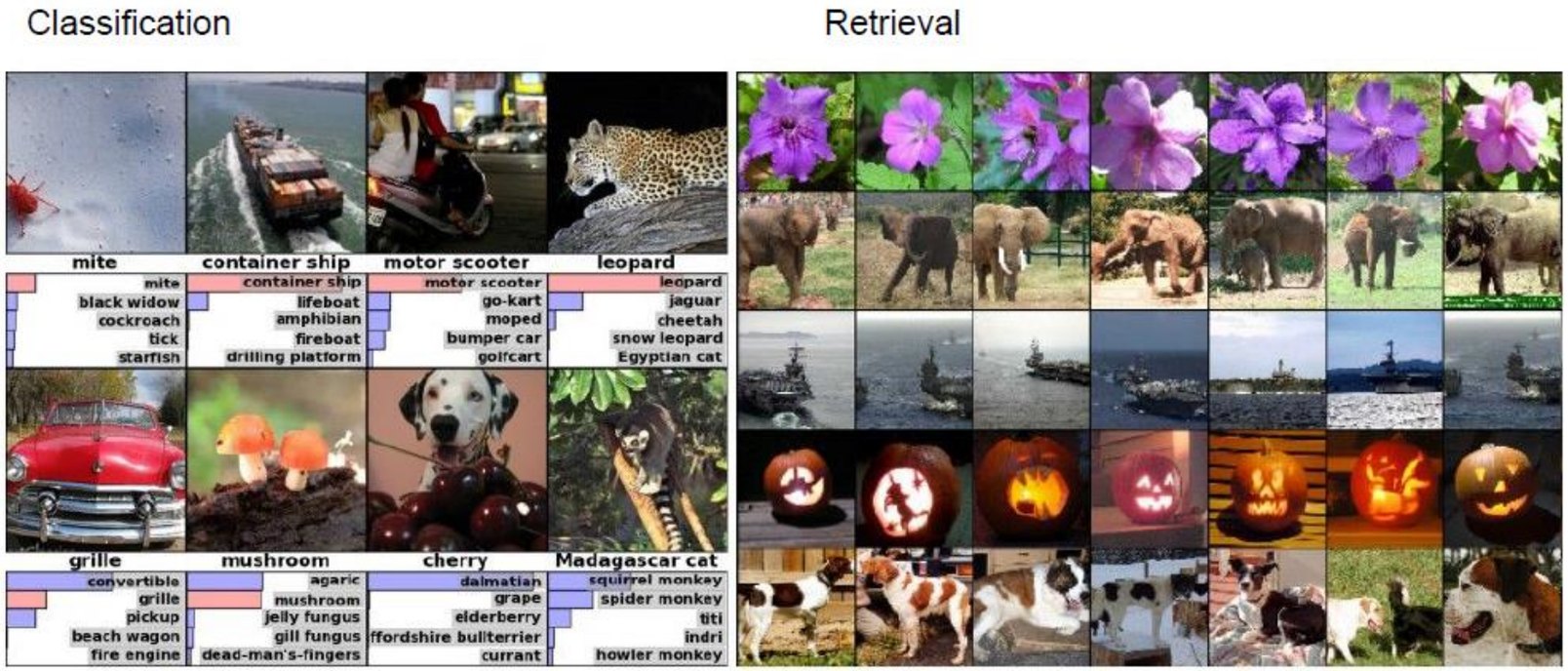


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

"AlexNet"

Today

- CNNs = basis for almost everything in computer vision!

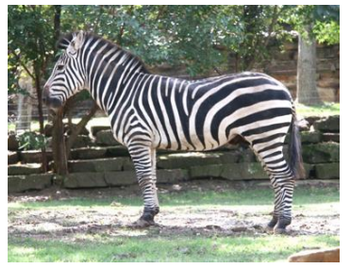


- More examples in online lecture

Why CNNs?

Remember: Similarity Retrieval by Histograms

- Q: What happens if we reshuffle all pixels within the images?



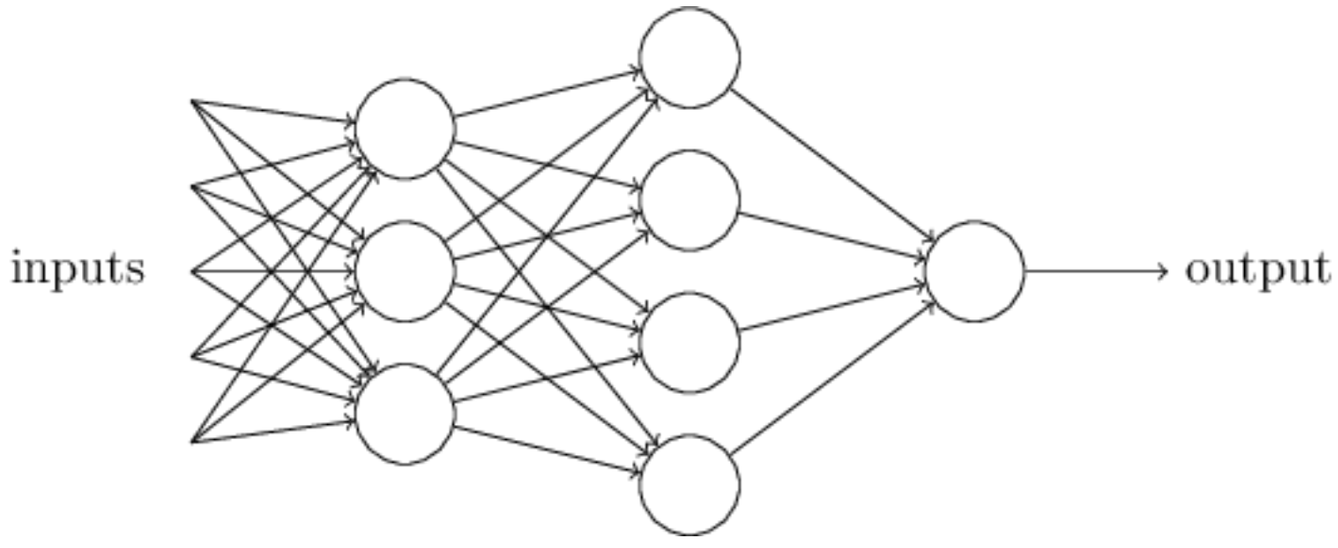

- A: Its histogram won't change.
  Point-wise processing unaffected.
- Similar for MLP: it does not explicitly model neighborhoods!
- Need to measure properties relative to small ***neighborhoods*** of pixels

Slide credit: Travor Darrel

Recap: Example from Image Classification

- Task: predict class of image (e.g. hand written digits)
  - Input: image (size w x h), here w=28, h=28
  - Output: class (0,1,...,9)

- MLP Solution:
  - Architecture:
    - w*h inputs (=768)
    - 10 Outputs
    - H hidden layers



Shortcomings of Example 2

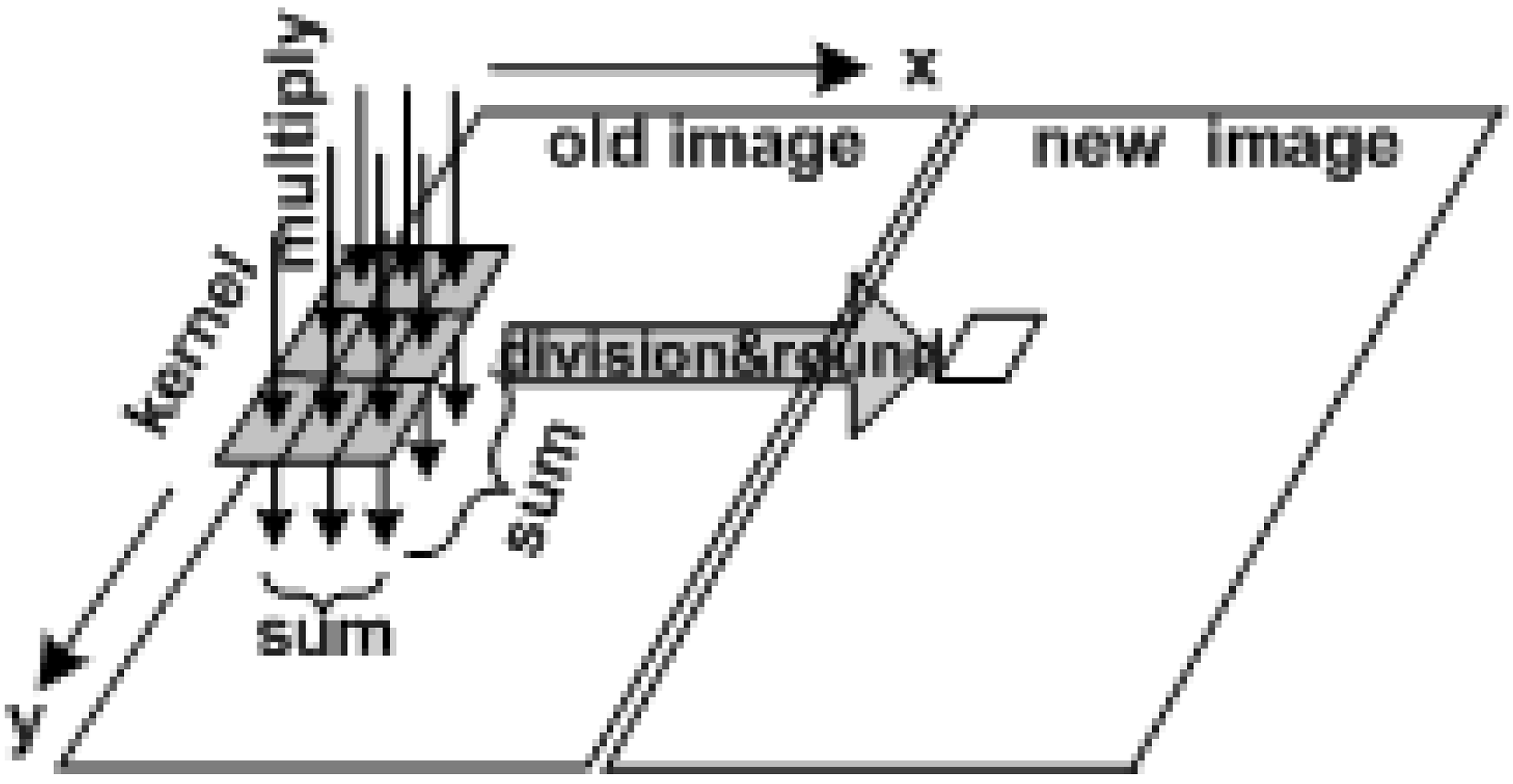
- MLP architecture
  - treats all inputs independent
  - order of pixels is irrelevant to the network
  - hidden layer has a global receptive field (each neuron sees all pixels)
  - Number of neurons and weights scale directly with image size (number of pixels)

- Can we consider all pixels of an image independent?
- Problem:
  - *Neighborhood relations* not preserved
  - Spatial relations would have to be learned by the network itself (fully data-driven) → would require exorbitant number of training data
  - Similar situation for learning of structures at *multiple scales*
  - Scales badly with image size (e.g. 1000x1000 pixels → 1M neurons
- Conclusion: **MLPs are not optimal for images**!

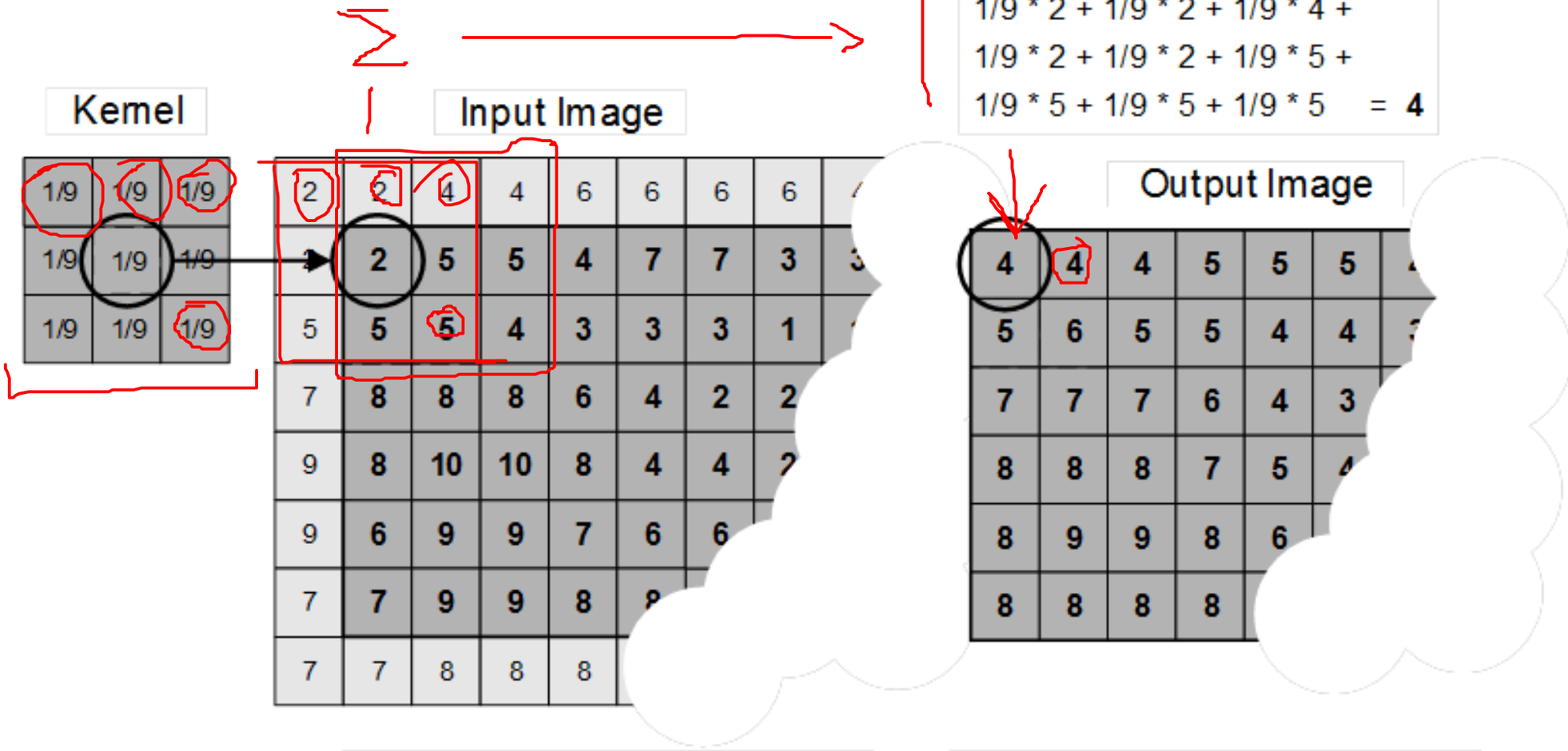
Basics:
The Convolution

Remember

- Go from individual pixels to entire neighborhoods
- This led us to the convolution!



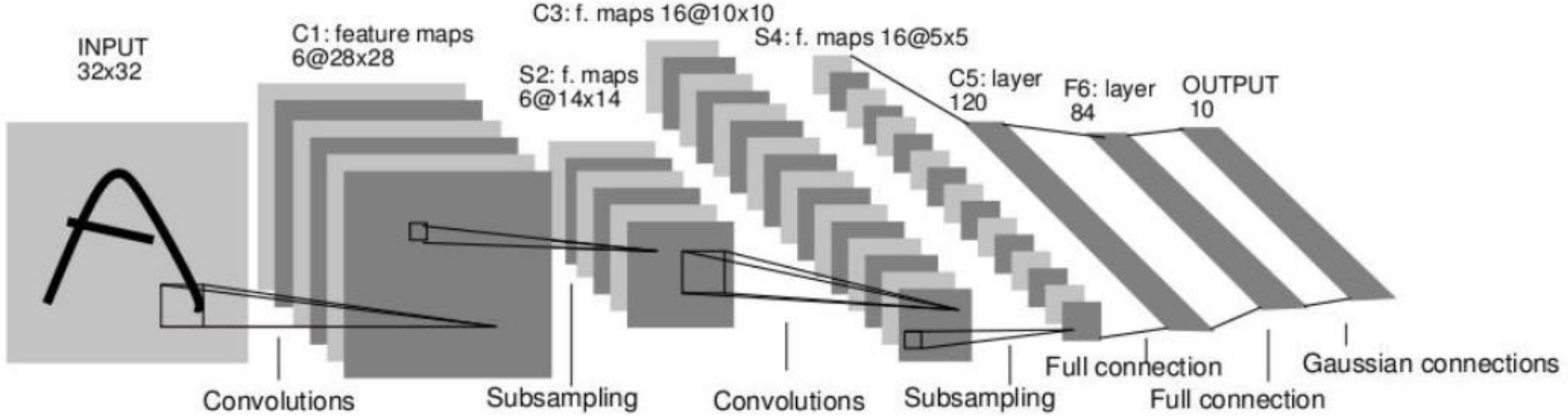
Convolution: Example



credit: https://www.imgtec.com/blog/heterogeneous-compute-case-study-image-convolution-filtering/

Convolutional Neural Networks

- Similar idea for neural networks!
- Instead of processing individual pixels in a neuron, model groups of pixels

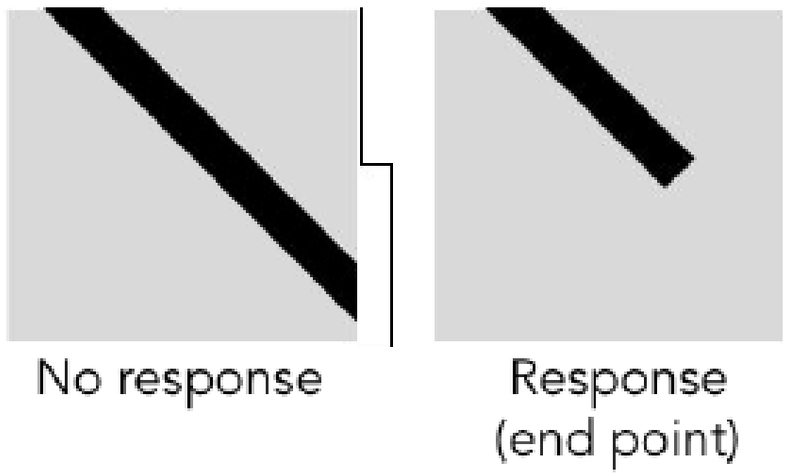


- More information: http://neuralnetworksanddeeplearning.com/chap6.html

Biological Motivation of Convolutional Neural Networks

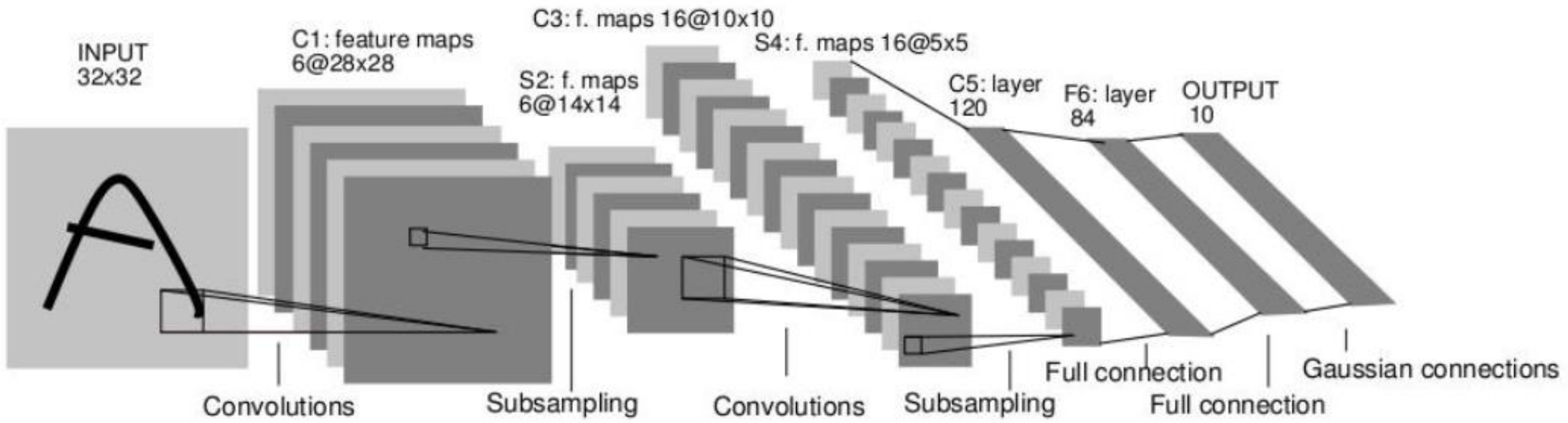
- Motivation
  - Nearby neurons in visual cortex represent nearby regions in the visual field
  - Neurons responsible for visual perception are hierarchically stacked
    - Simple cells (orientation)
    - Complex cells (orientation + movement)
    - Hypercomplex cells (orientation + movement + end point, i.e. corners)

- Both concepts are mapped to some degree in CNNs



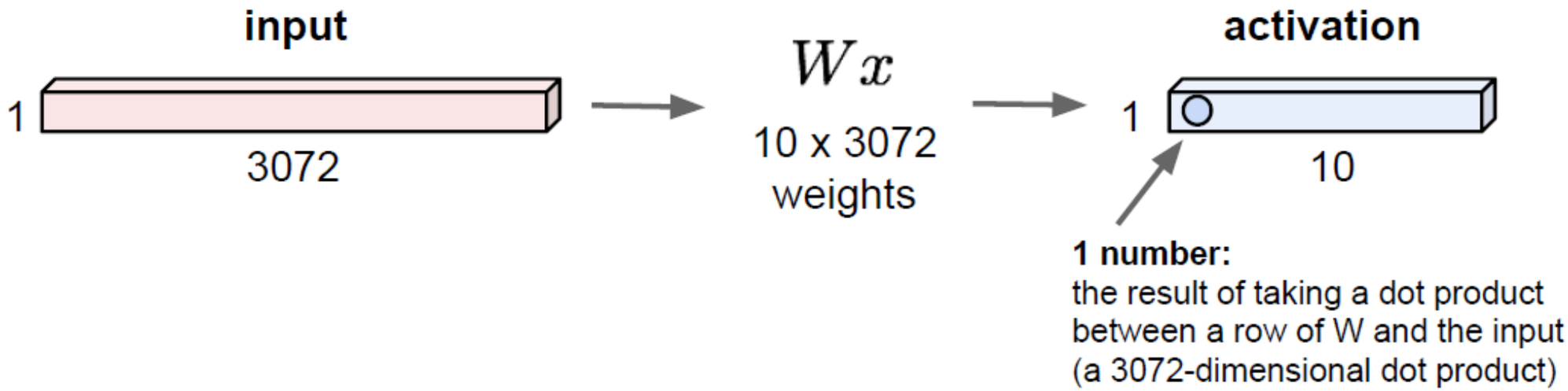
The CNN Architecture

Convolutional Neural Networks

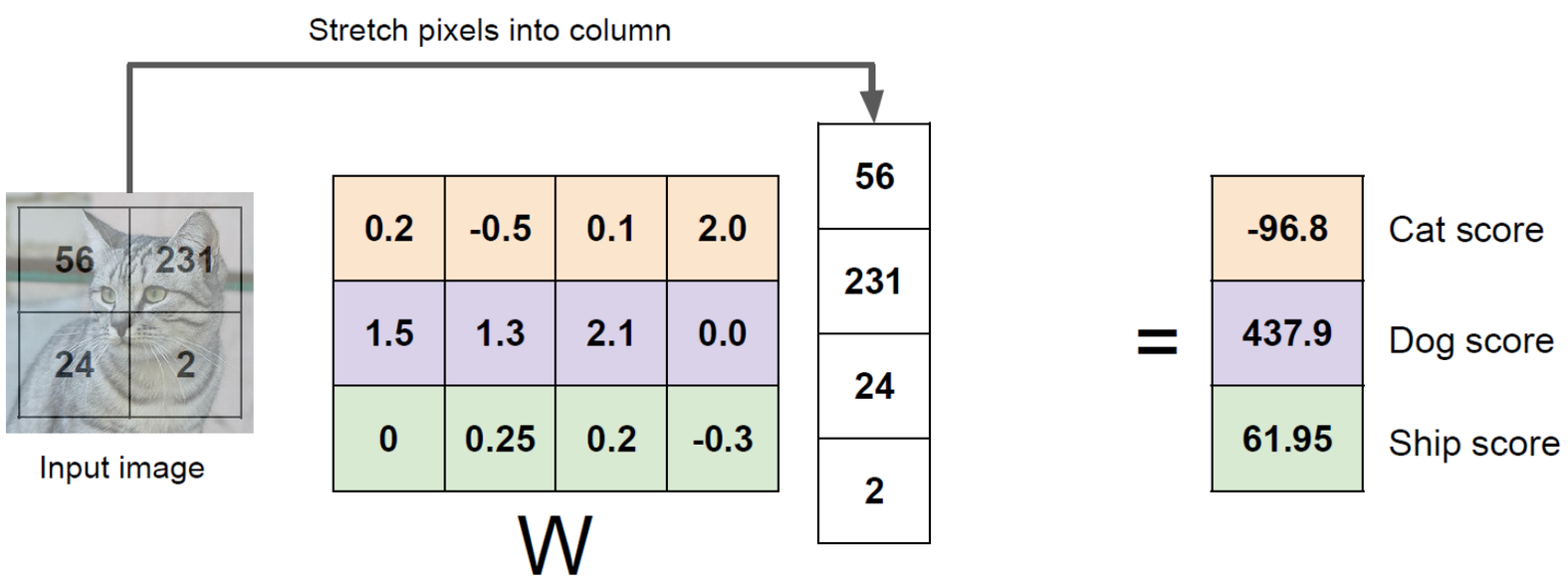
- Similar to other neural networks: weights and biases as trainable parameters
- Non-linearity in activation function
- Fully differentiable
- Can be combined with fully-connected layers to build e.g. classification nets
- 

- See also: http://neuralnetworksanddeeplearning.com/chap6.html and https://cs231n.github.io/convolutional-networks/

So far: fully connected layers



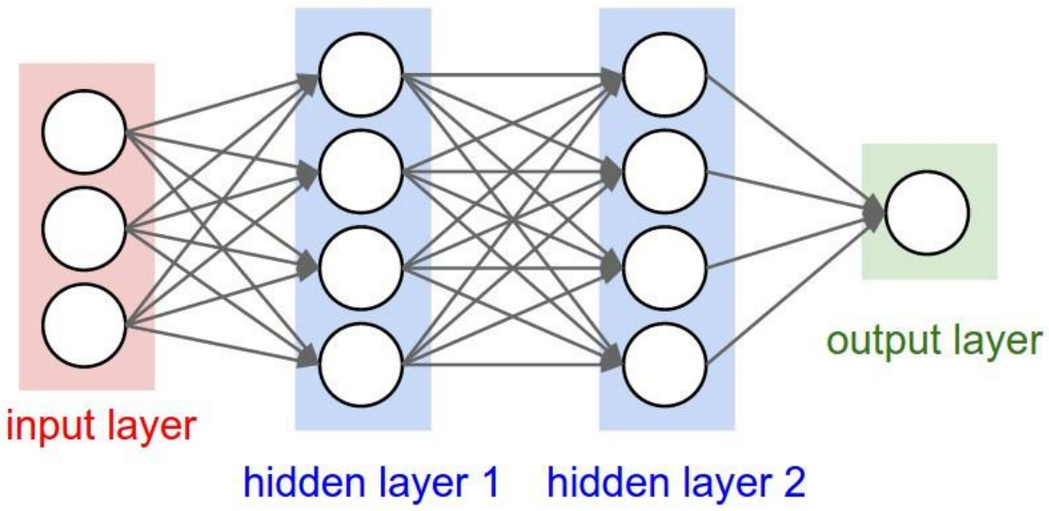
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



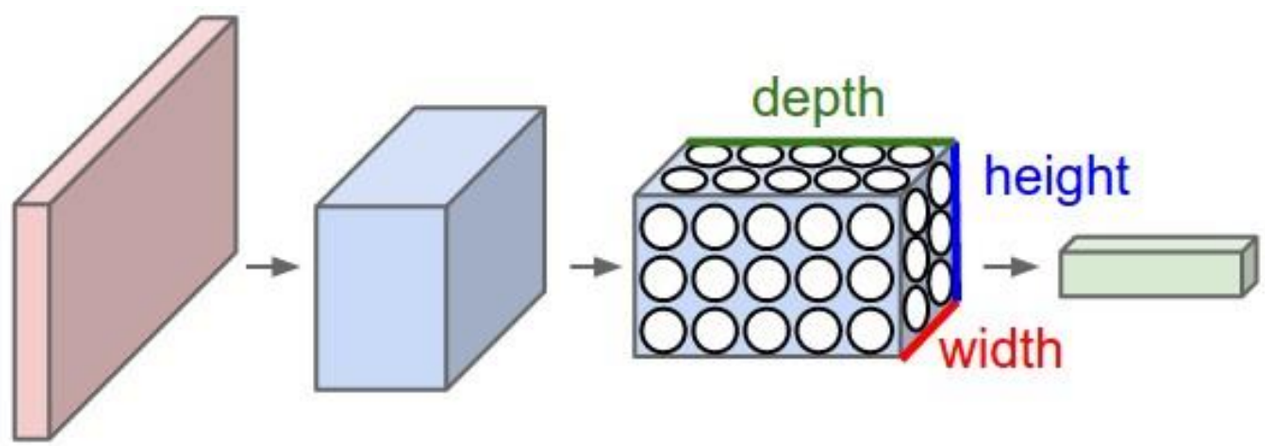
Convolutional Architecture

- Layers have 3 dimensions: width, height and depth
- Every node in a layer is connected to only a small region in the layer before

MLP

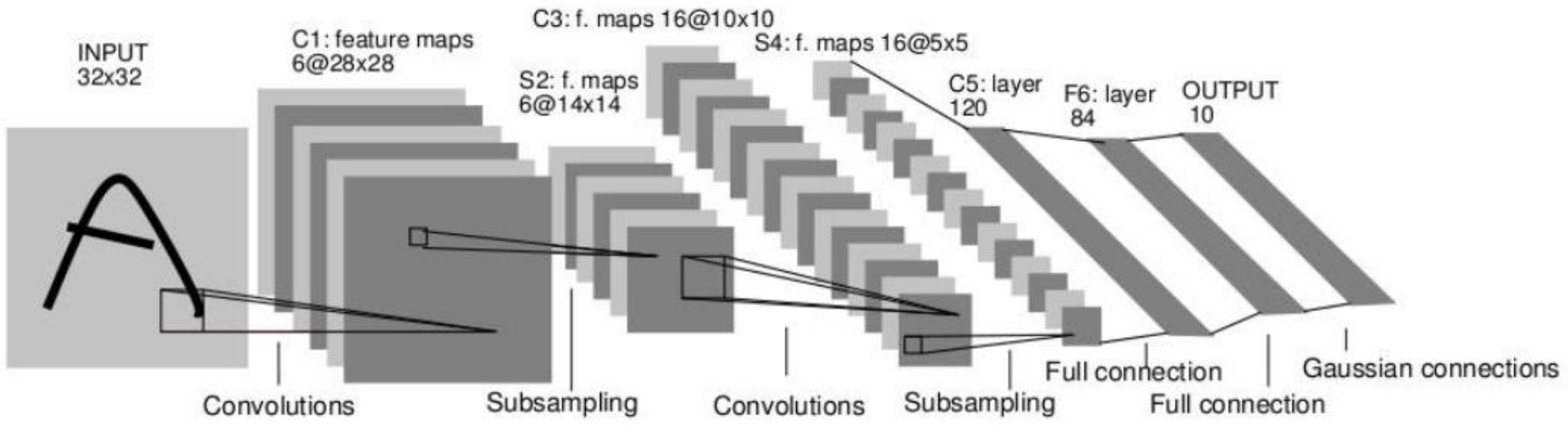


CNN



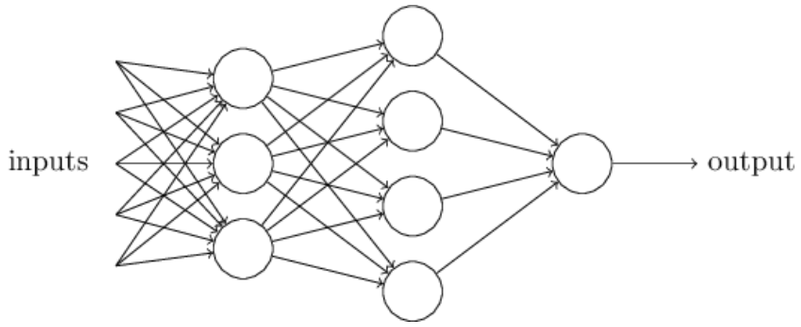
Convolutional Neural Networks

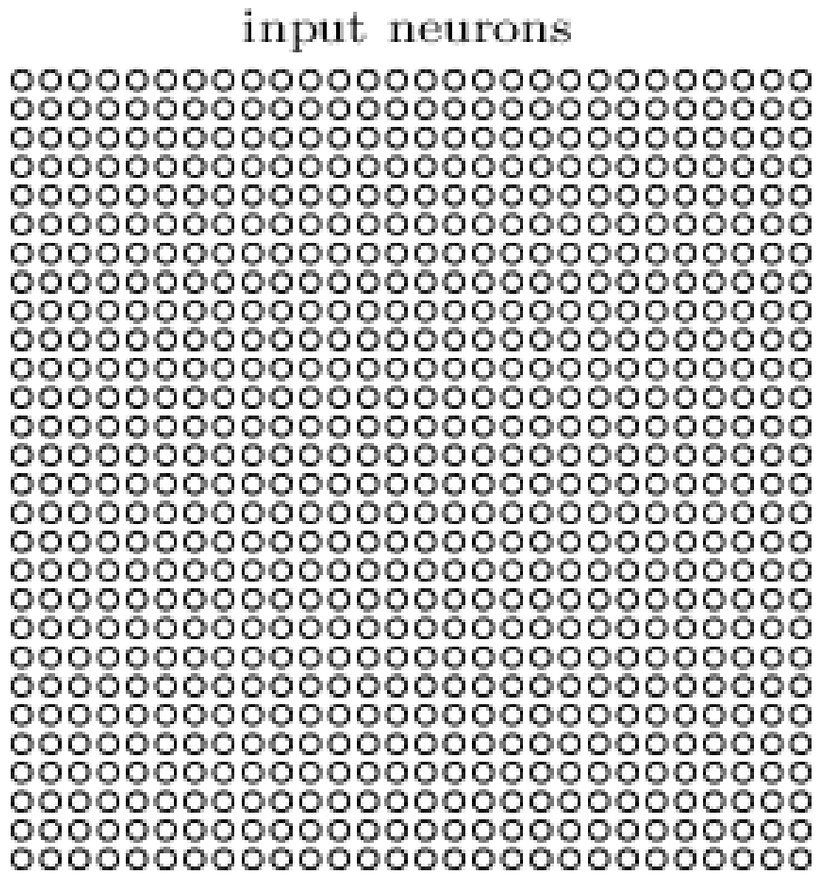
- What is new?
- 3 basic concepts:
  - **Local receptive fields** → preserve neighborhood information
  - **Shared weights** → get global / comparable representations, reduce number of parameters
  - **Pooling** → multi-scale analysis



Local receptive fields

- Think 2-dimensional
- Instead of connecting every neuron to every neuron in the next layer (as before)

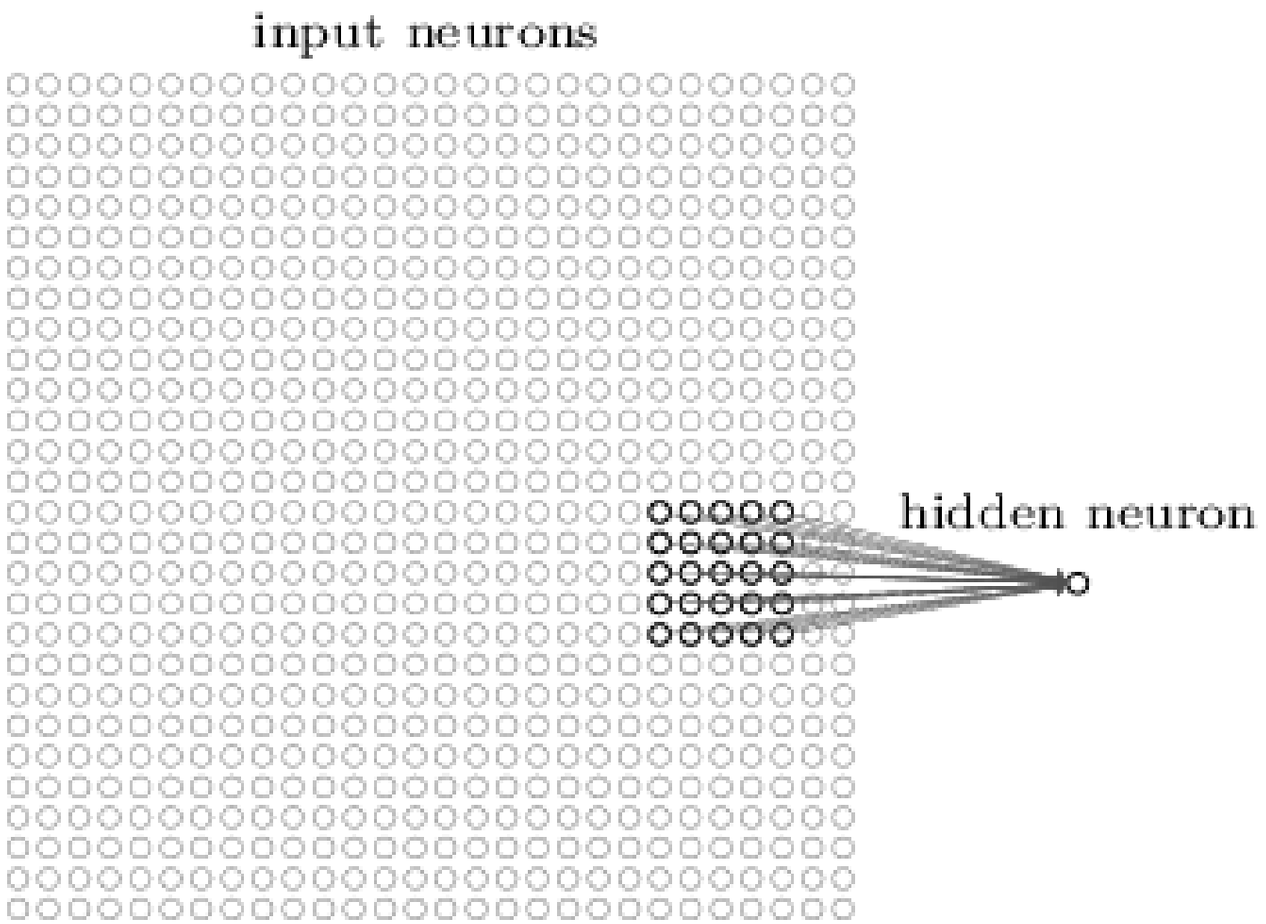




- Only make connections in small, localized regions of the input image

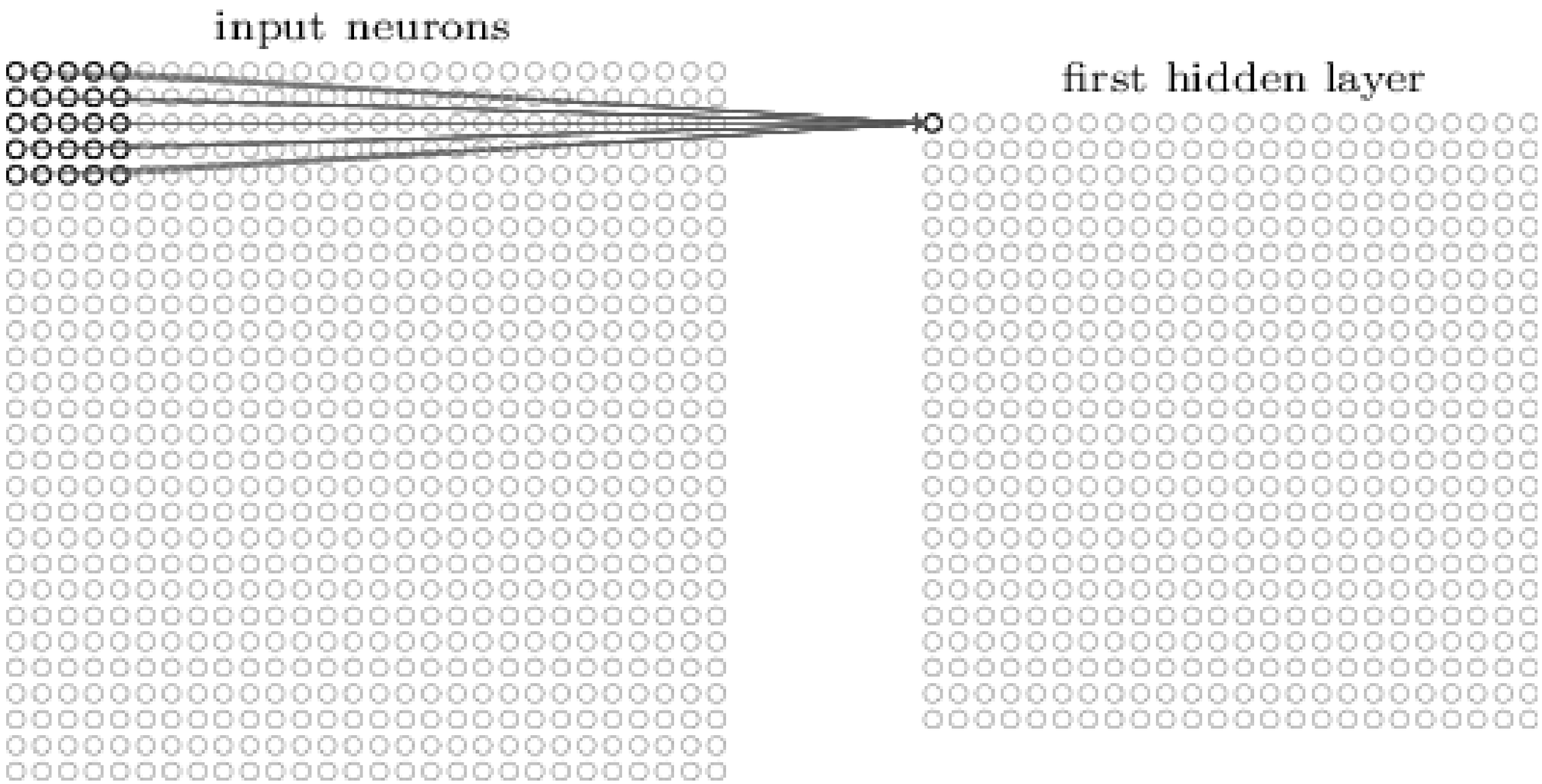
Local receptive fields

- E.g. each neuron in the first hidden layer will be connected to a small region of the input neurons, say, for example, a 5 $\times$ 5 region, corresponding to 25 input pixels
- This region is the receptive field of the hidden neuron



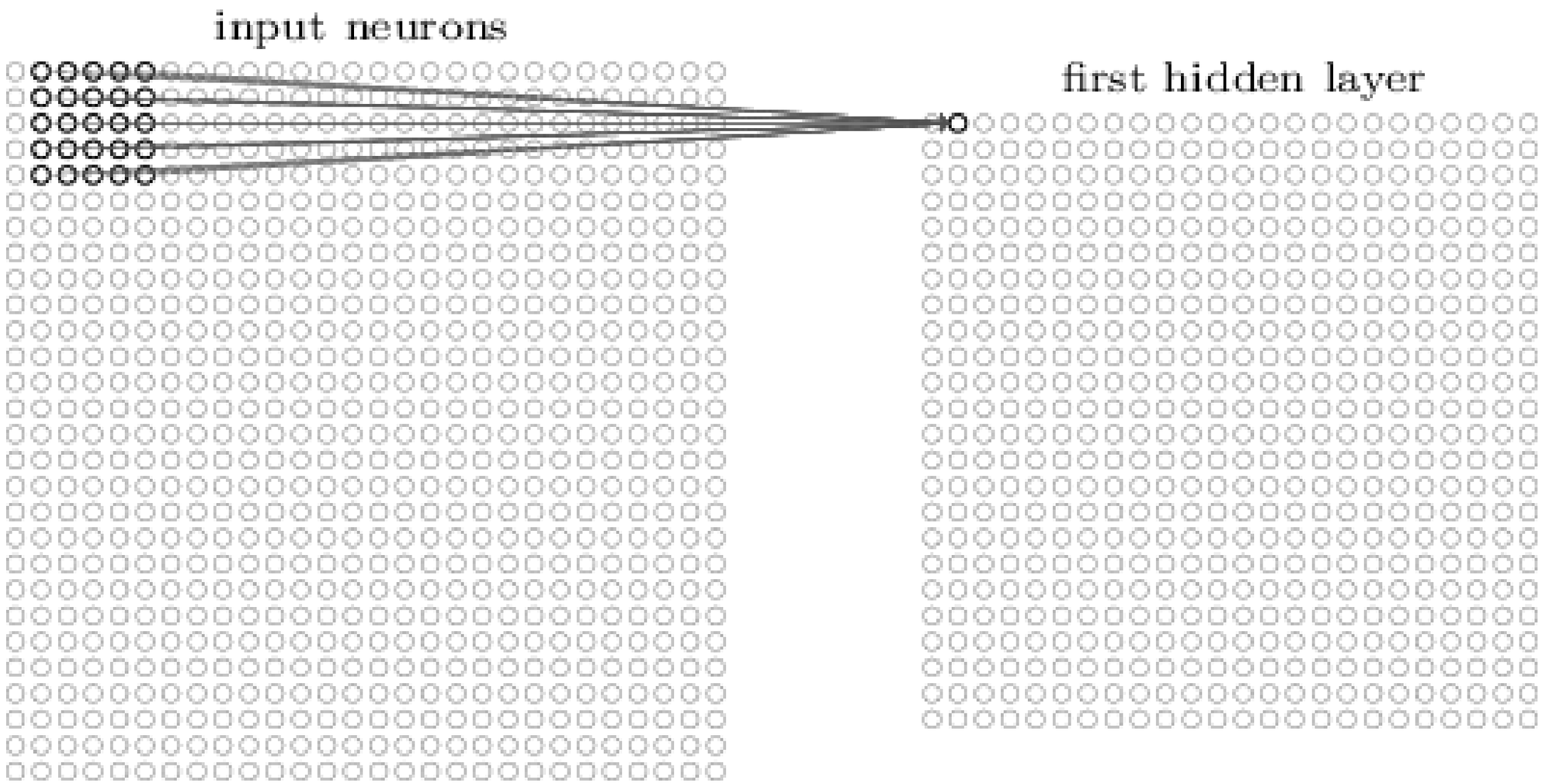
Local receptive fields

- Now, let's build the first hidden layer by sliding the window!



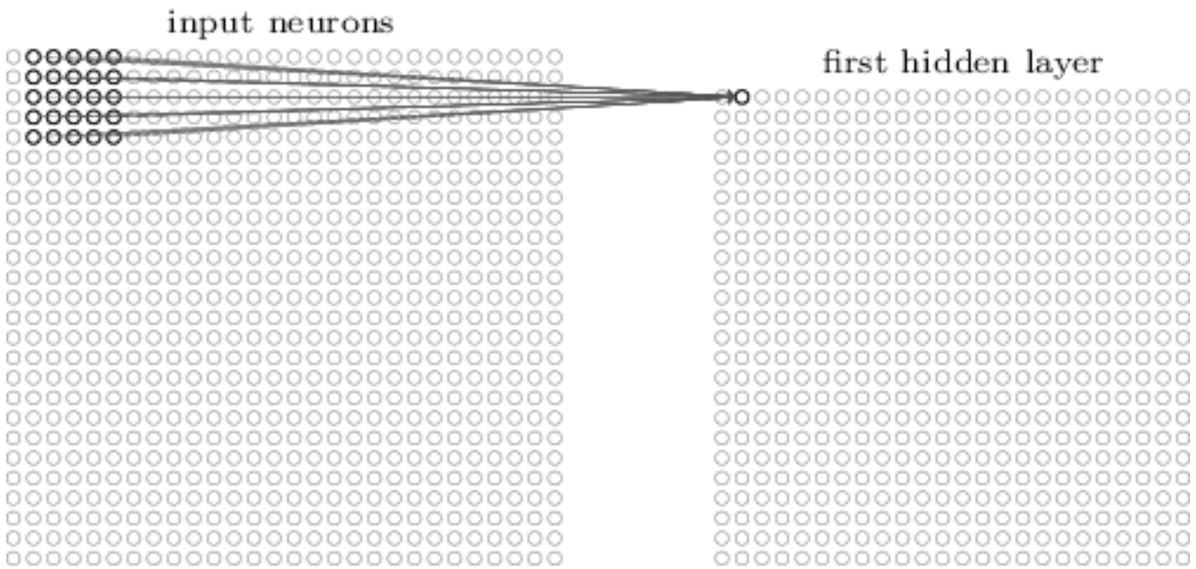
Local receptive fields

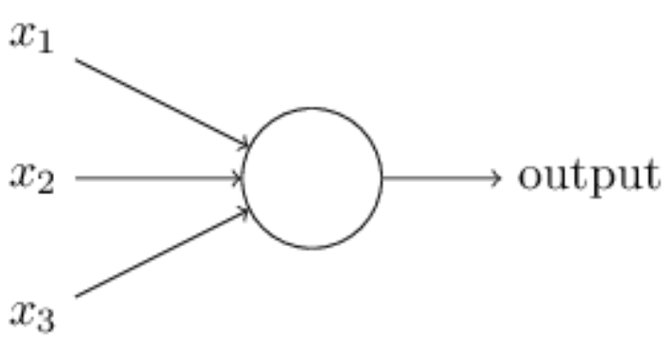
- Now, let's build the first hidden layer by sliding the window!



Shared weights

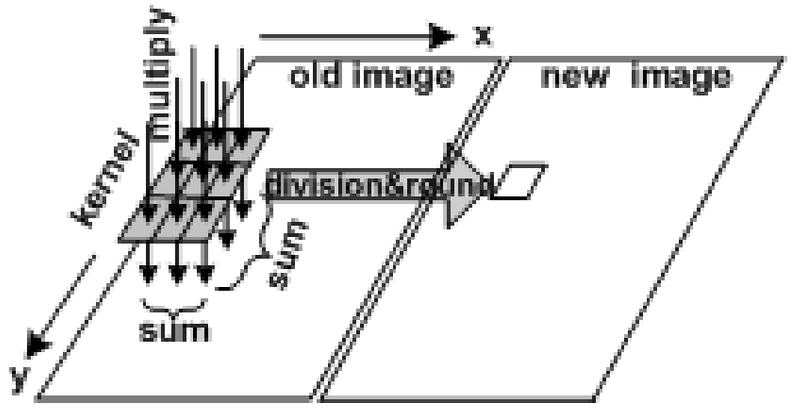
- Each hidden neuron has 5x5 weights and 1 bias
- Keep the weights and bias **constant** for all neurons in the hidden layer (weight sharing)
- What happens in the neuron?
  Remember:





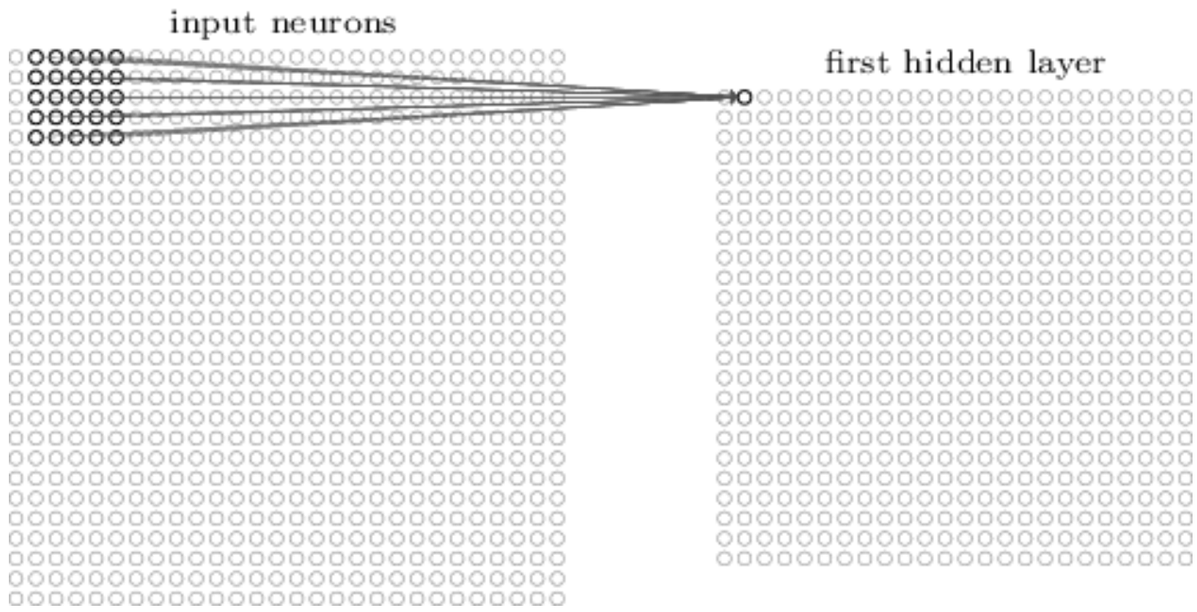
$$\text{output} = \sigma(w \cdot x + b)$$

- The dot product w*x is the same as $\sum_{i=1}^{25} w_i * x_i$



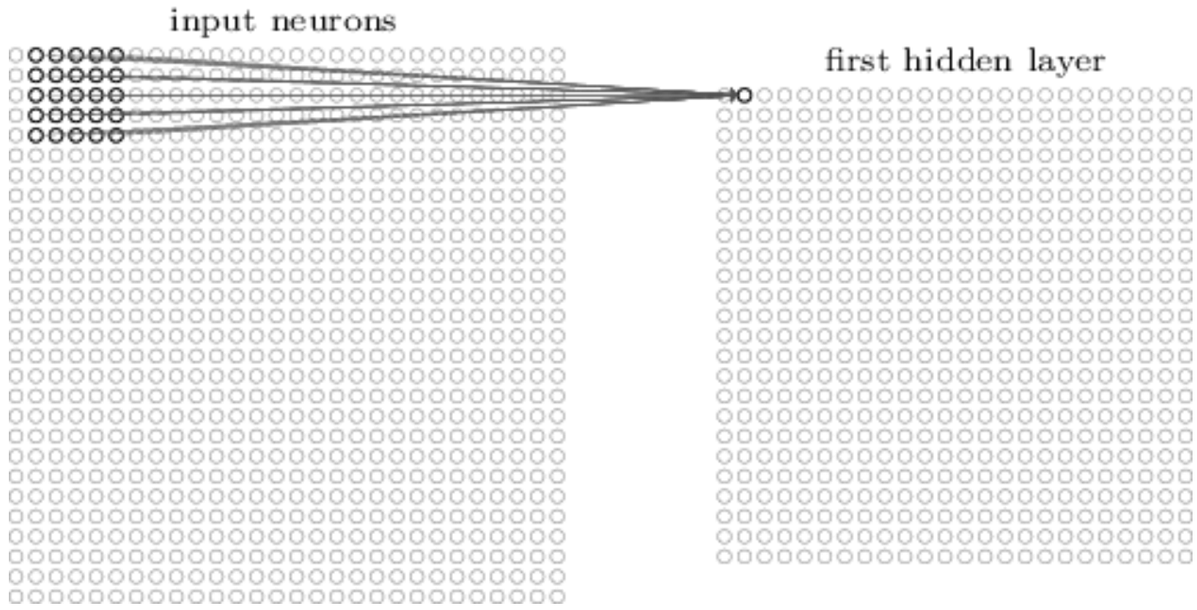
Shared weights

- The weights of the neuron are weights of a convolution **filter** which is sled across the image!
- The weights and biases define a “*kernel*” or “*filter*”
- The result of the convolution is stored in the output of the first hidden layer! This output forms the a “*feature map*”
- The feature map represents the response of the filter to the input image



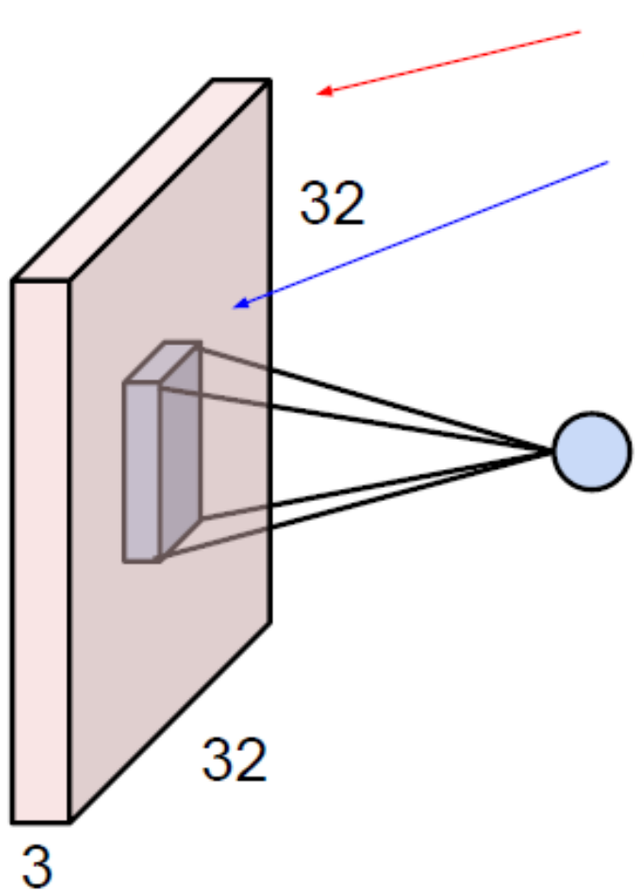
Shared weights

- The weights of the neuron are weights of a convolution **filter** which is sled across the image!
- The weights and biases define a “*kernel*” or “*filter*”
- The result of the convolution is stored in the output of the first hidden layer! This output forms the a “*feature map*”
- The feature map represents the response of the filter to the input image
- One feature map can represent one structure *everywhere* in the image → *location invariance*!



- But: one feature map can represent only one structure → for more structures: simply add more feature maps!

Convolutional Layers

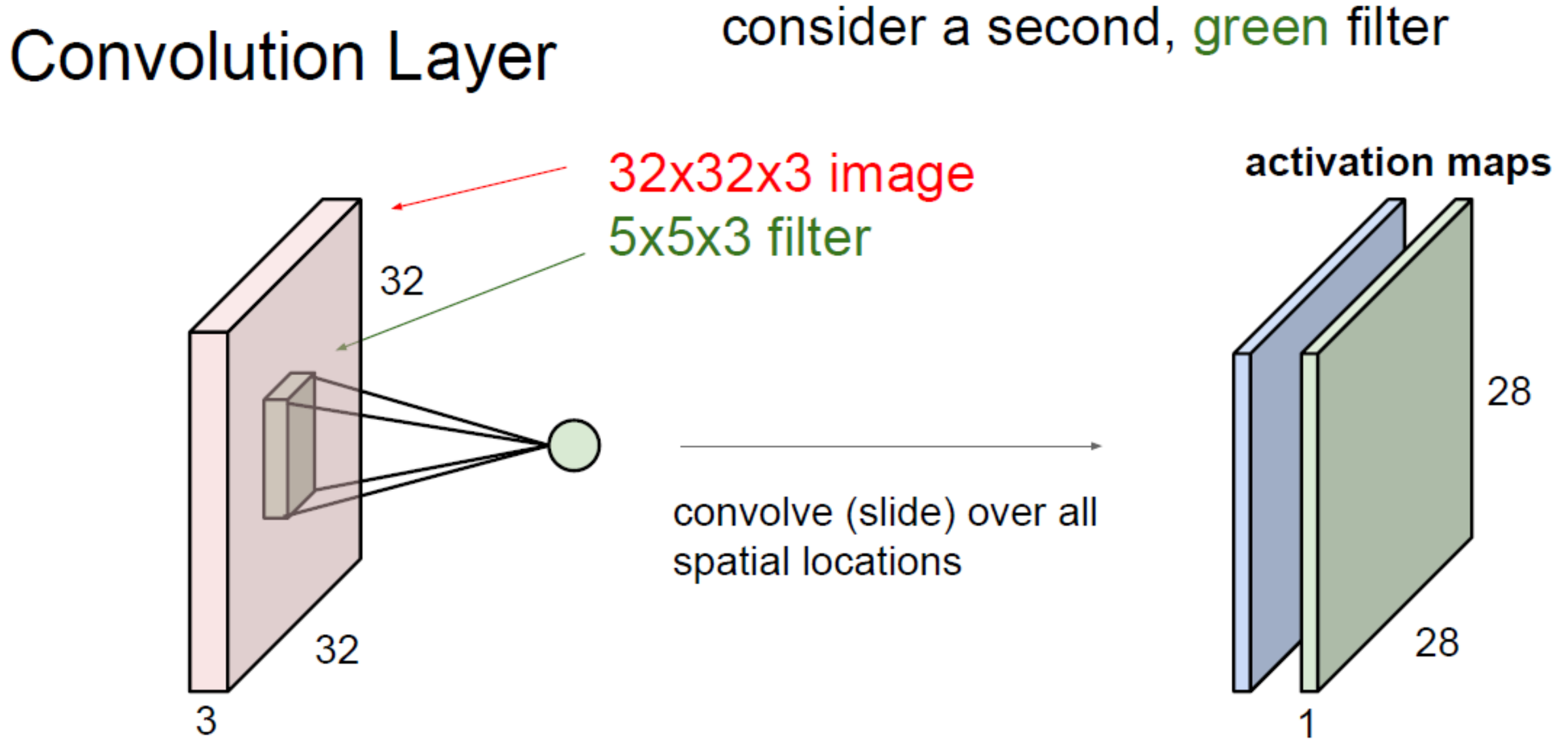


1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

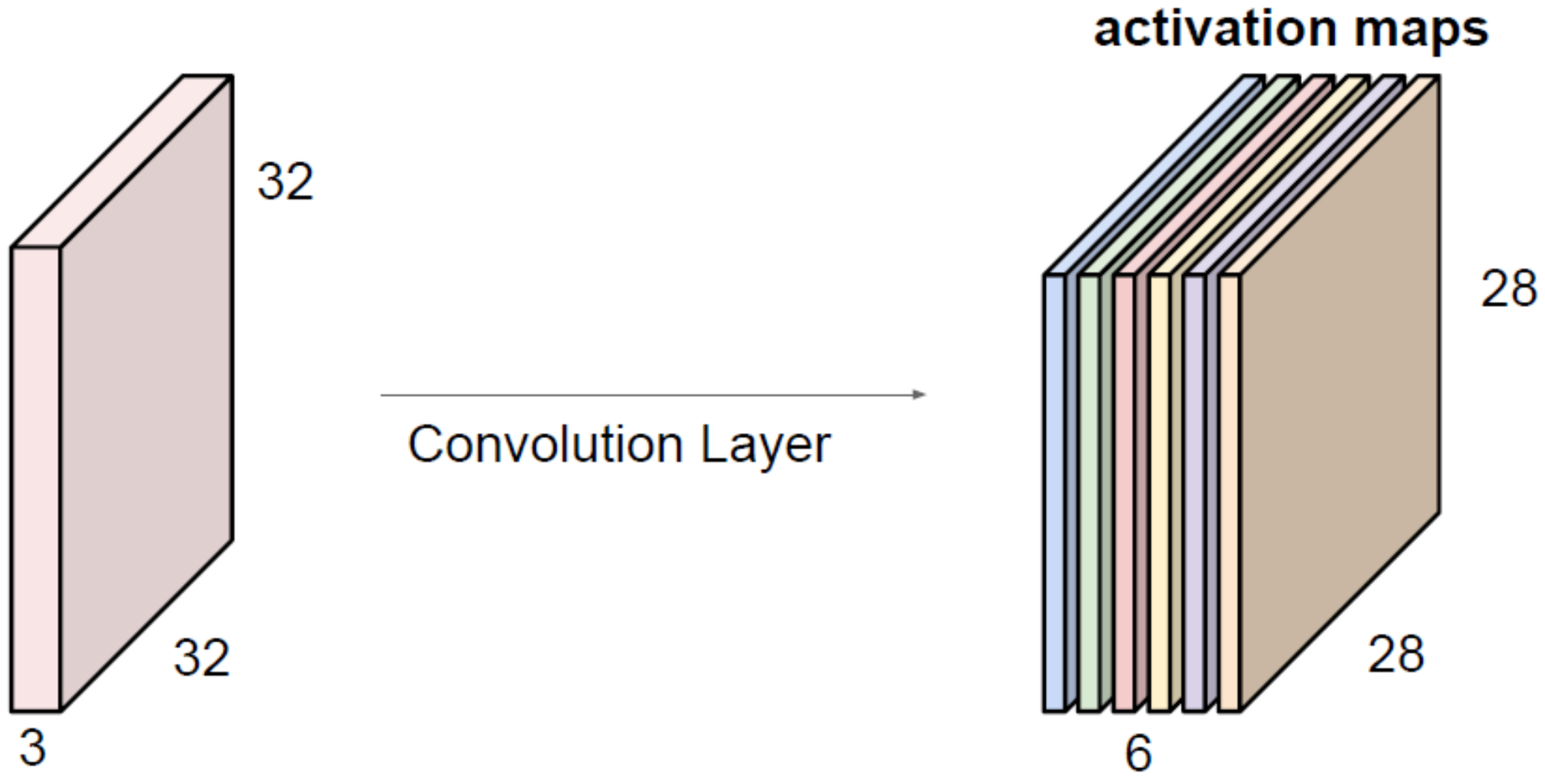
- Note: w is a column vector, w^T is a row vector with 5x5x3=75 elements, x is a column vector (also with 75 elements of corresponding pixels), b is a scalar
- The inner product (=dot product) does the same as the convolution at a certain position. It retrieves how well the filter and the image match in one number (scalar)
- Do this at every location, to do a convolution across the entire image

Feature maps (=activation maps)



Convolutional Layers

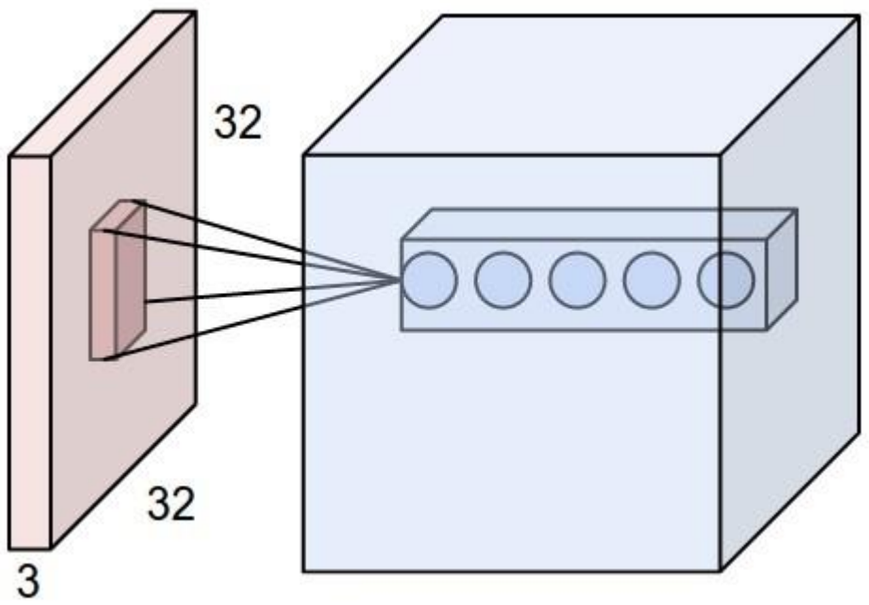
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

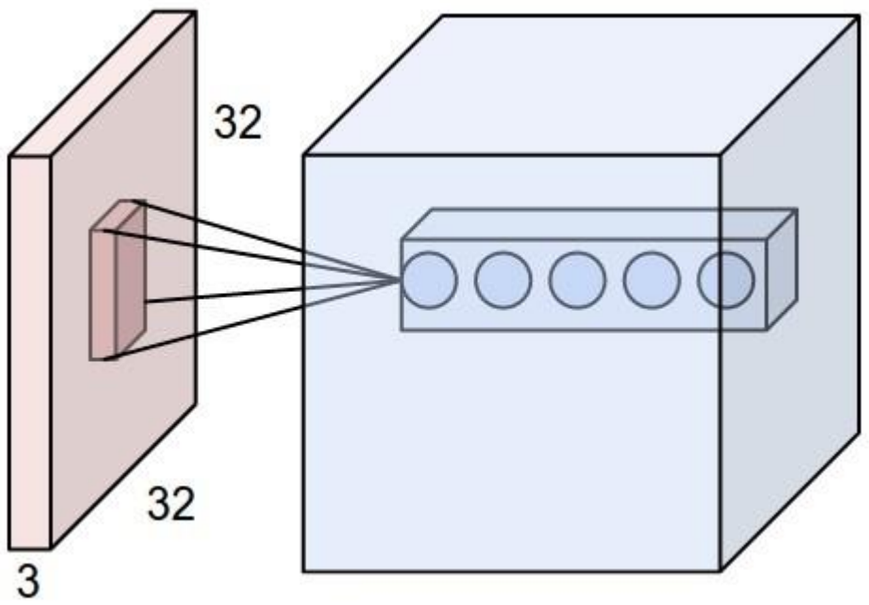
- We can stack also convolutional layers! Here, e.g. use a 5x5x6 filter on the 28x28x6 activation map!

Connectivity



- Connectivity is **sparse** across with and height of the image / feature map
- Connectivity is **dense** across the depth dimension (different feature maps are densely connected), i.e. each layer is connected to all feature maps of the underlying layer.
- Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels).

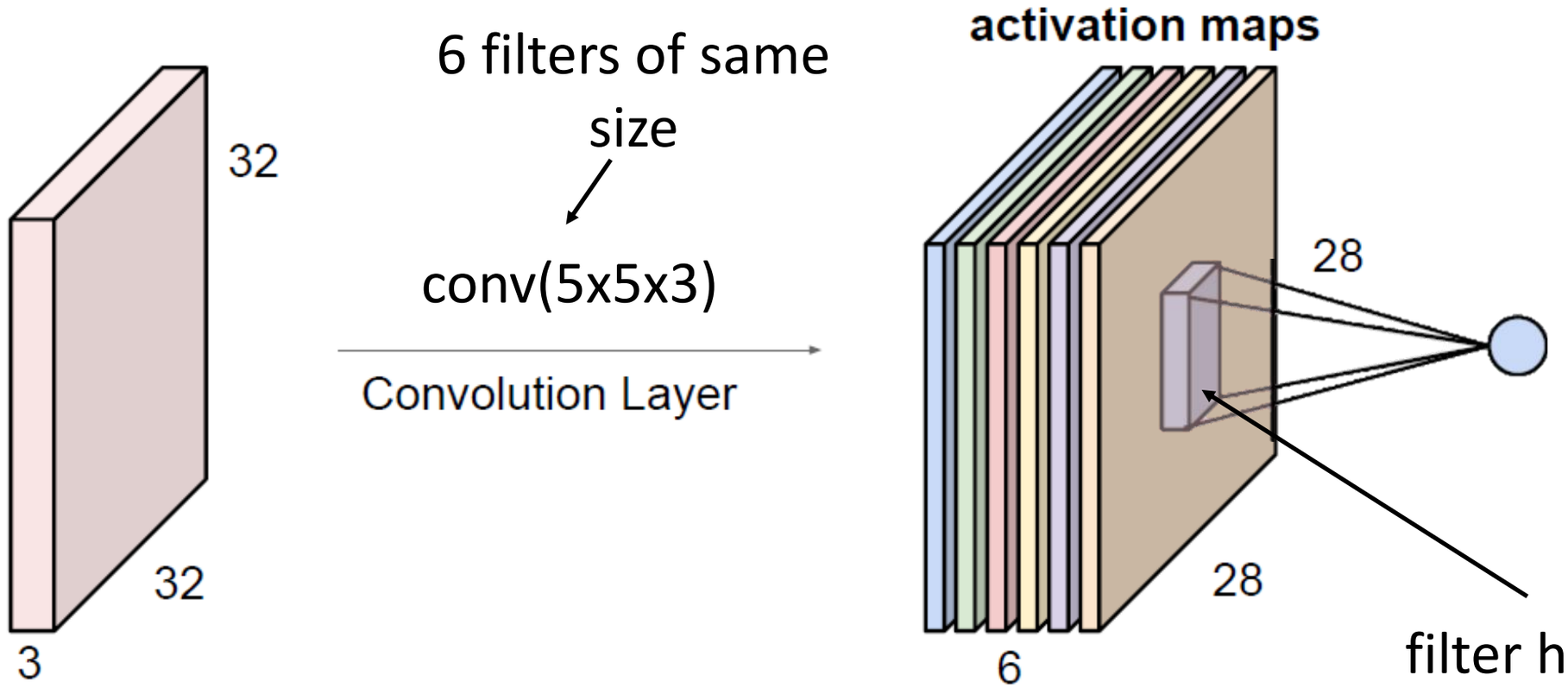
Coping with color



- Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels).
- First convolutional filter learns 3-channel filters (color patterns)
- Higher convolutional filters may learn even higher-dimensional filters (with arbitrary deep volumes / numbers of channels)

The “Depth”-Dimension

- What happens in higher layers? E.g. input consists of 6 activation maps?
- Filter has a “depth” of 6



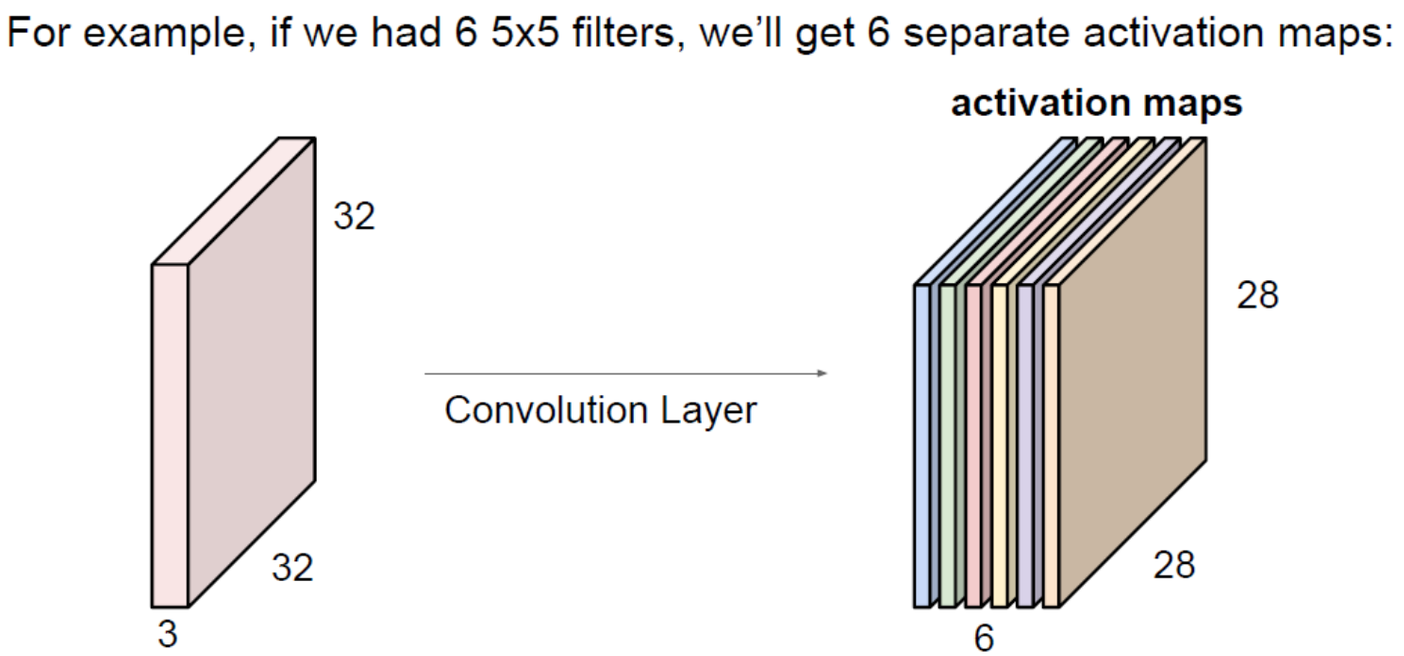
- Note: each filter from the first layer represents a different pattern in the input image!

Local connectivity & weight sharing

- Sharing weights, reduces the number of weights and biases to determine during learning *significantly*! E.g. in this case: for each feature map we need 5x5 weights + 1 bias = 26 parameters, for 6 feature maps this results in 26*6=156 parameters.

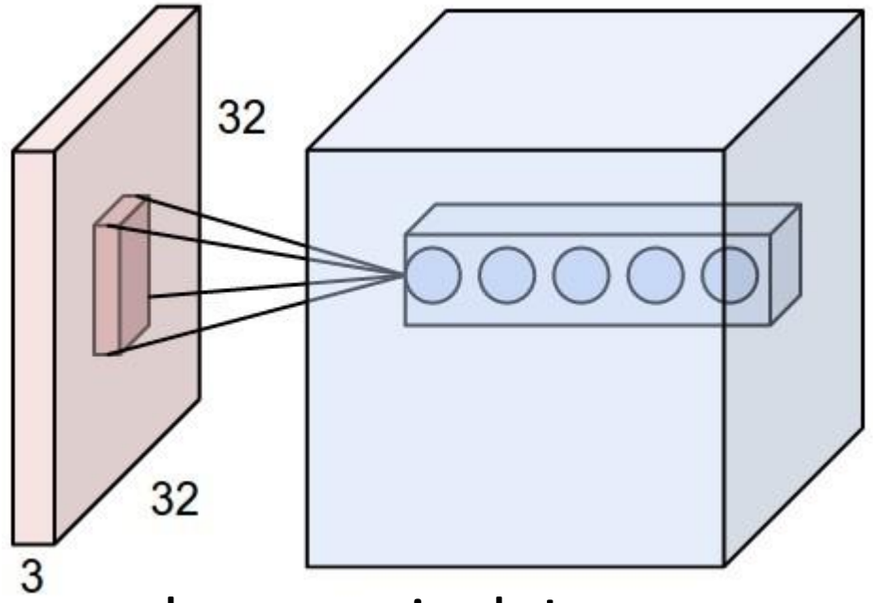
- Compare to fully-connected net with 784 inputs and 30 hidden neurons

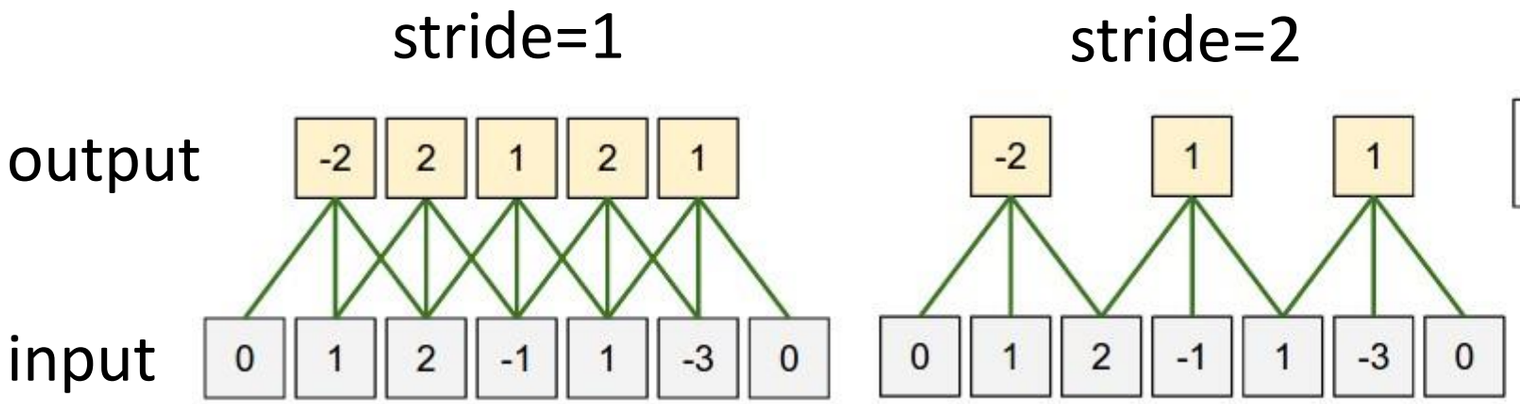
→ 784 * 30 weights + 30 biases = 23.550 parameters



Output side of the convolutional layer

- Output is three dimensional
- Depth = number of feature maps (hyper-parameter)
- Width / height of output volume depends on:
  - Stride: step size of convolution, stride=1 → move by one pixel. Larger strides reduce the width and height
  - Padding: pad zeros at boundary to avoid loosing spatial resolution through convolution

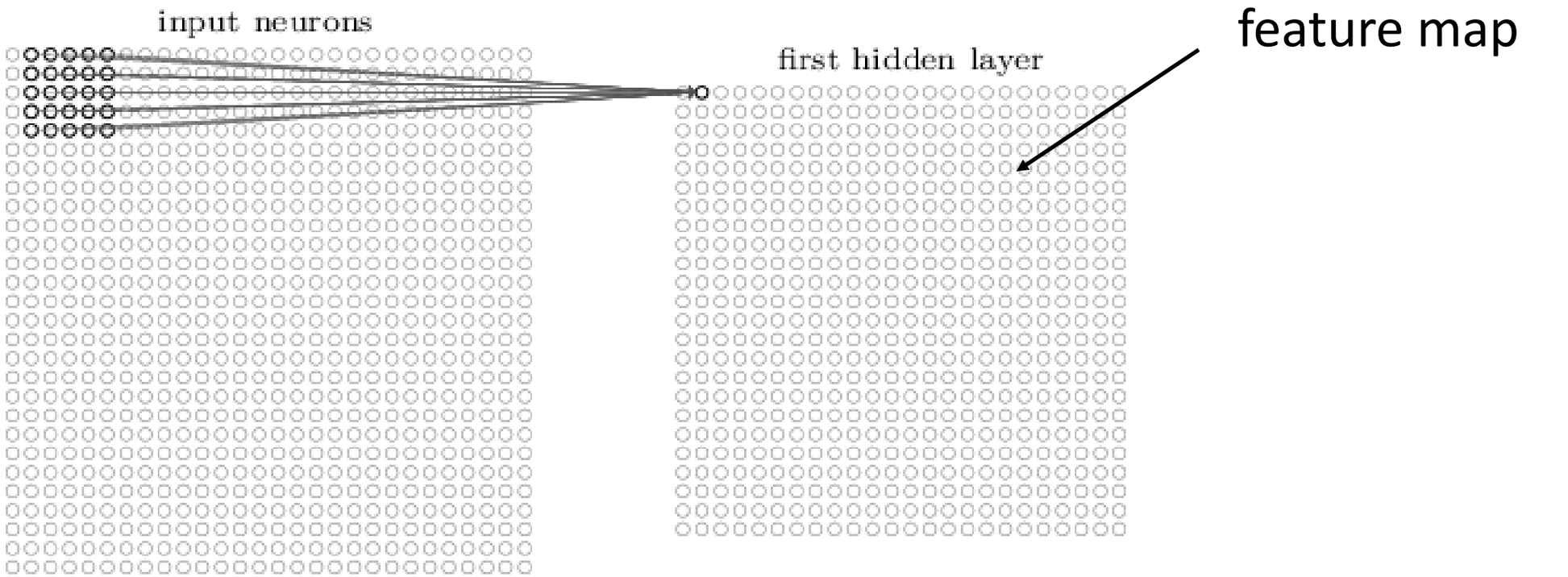




Filter: 1x3
Input: 1x5
Zero padding=1
No bias (for simplicity)

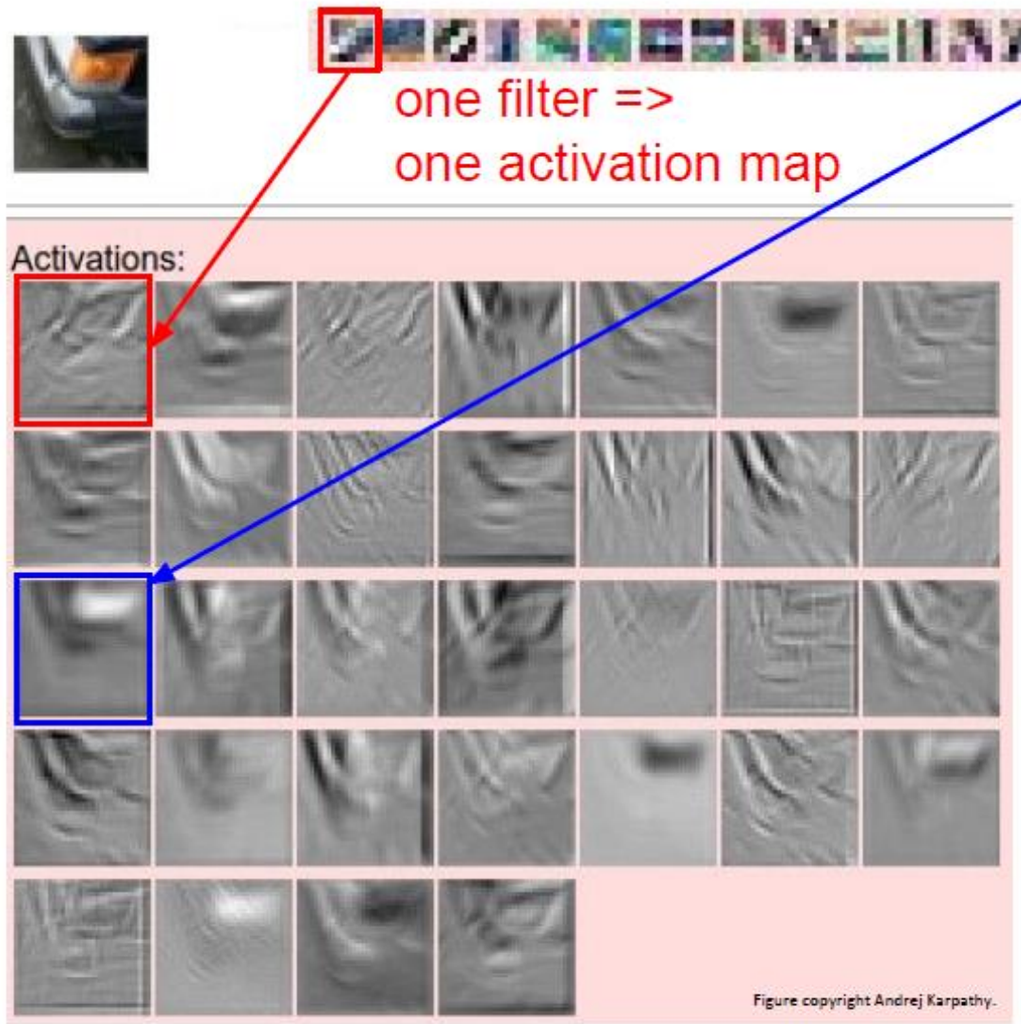
Terminology

- Each convolutional layer defines a number of **filters**
  - first layer typically: **N** filters of size 5x5x3 (because of 3 color channels)
  - second layer: e.g. M filters of size 3x3x**N** (this means each of the M filters combines data from all the N feature maps of the previous layer)
  - filters are the same everywhere in the image (**weight sharing**)
  - the values of the filters (**weights** of the filters) are learned
- Filters are also called **kernels**



Terminology

- When applying a filter to the input image, we get an **activation map**, i.e. each neuron in the next layer is activated somehow by the filter



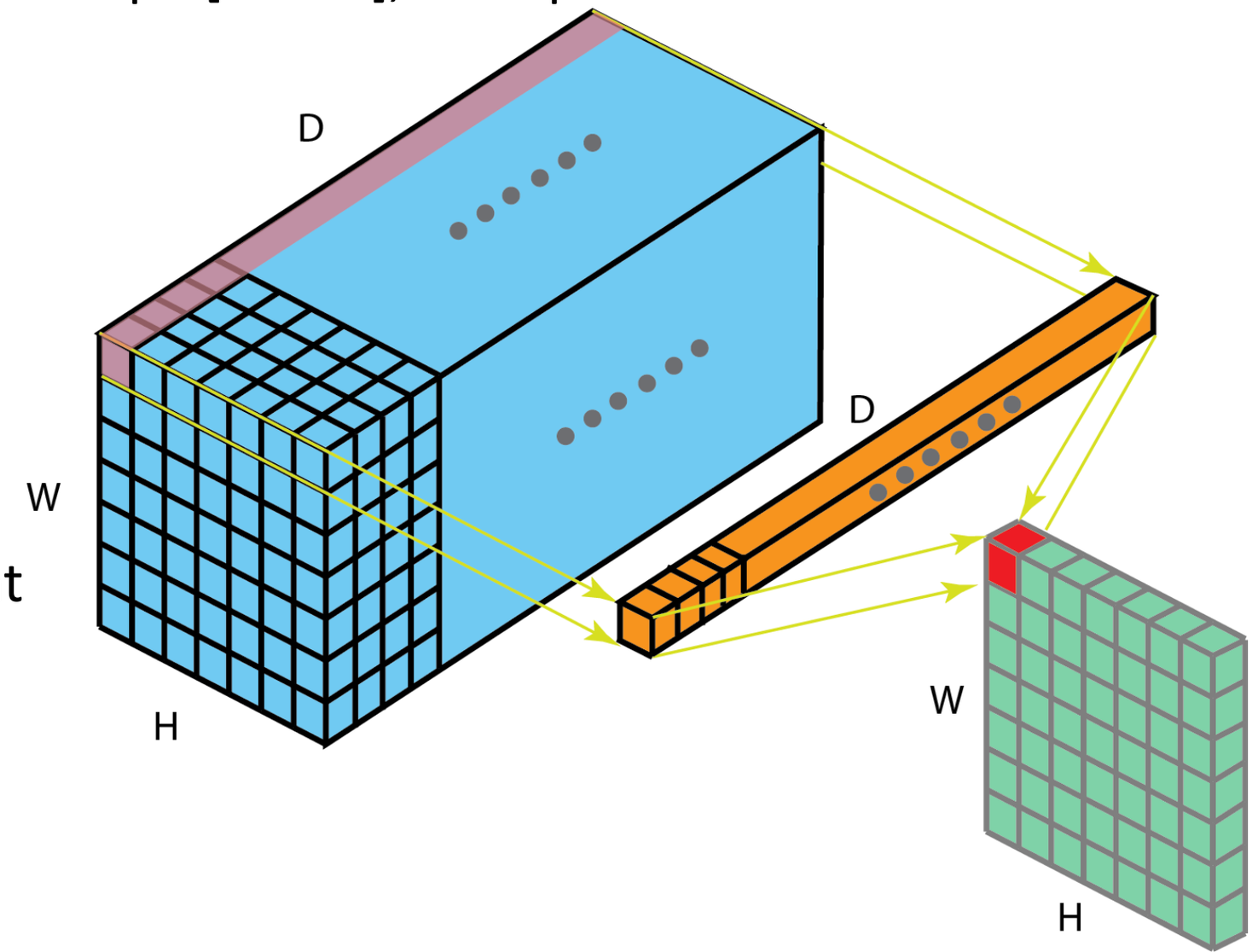
We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

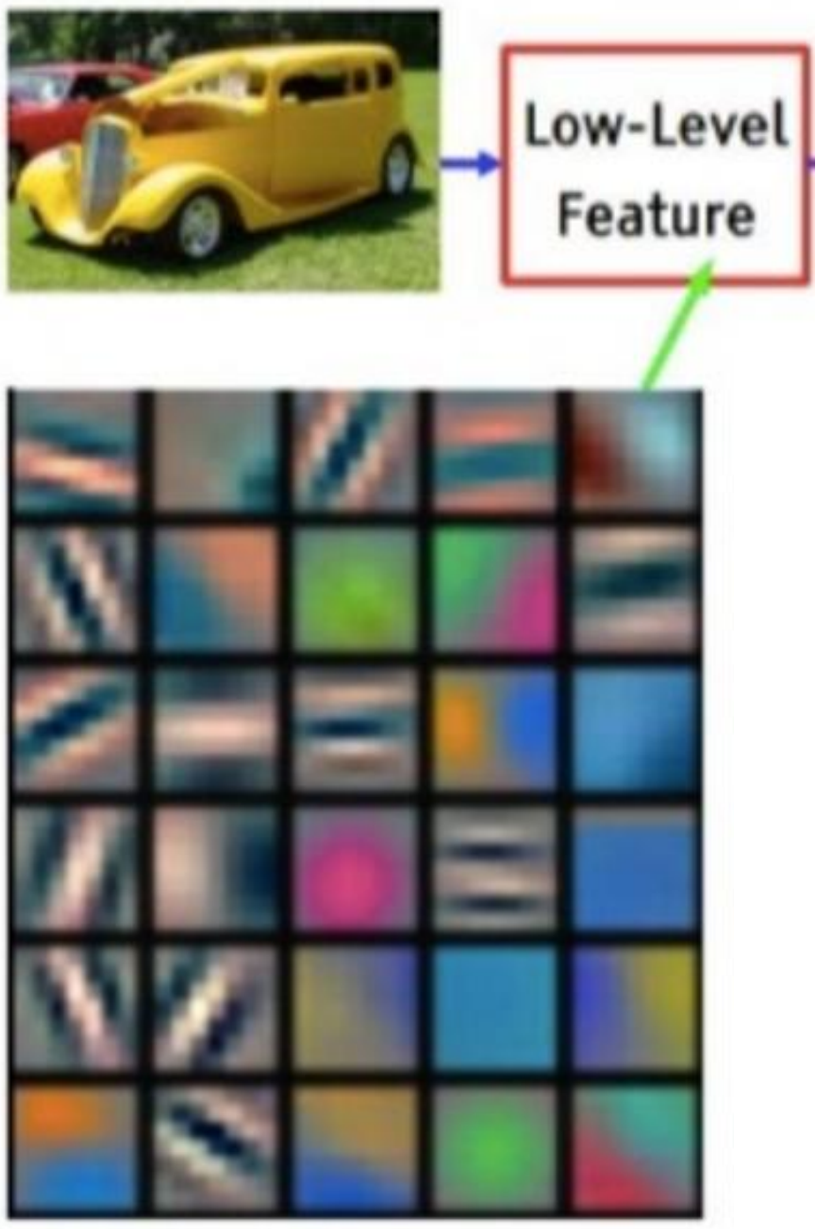
Note on 1x1 convolutions

- Does a 1x1 convolution make sense?
- Yes because convolutions operate across the depth channel as well
- A 1x1 convolution has filters of shape [1x1xD], D=depth
- Meaning: 1x1 convolutions operate only across the channels. They compute a linear combination across all channels for one spatial location.
- Resolution in width and height stays the same



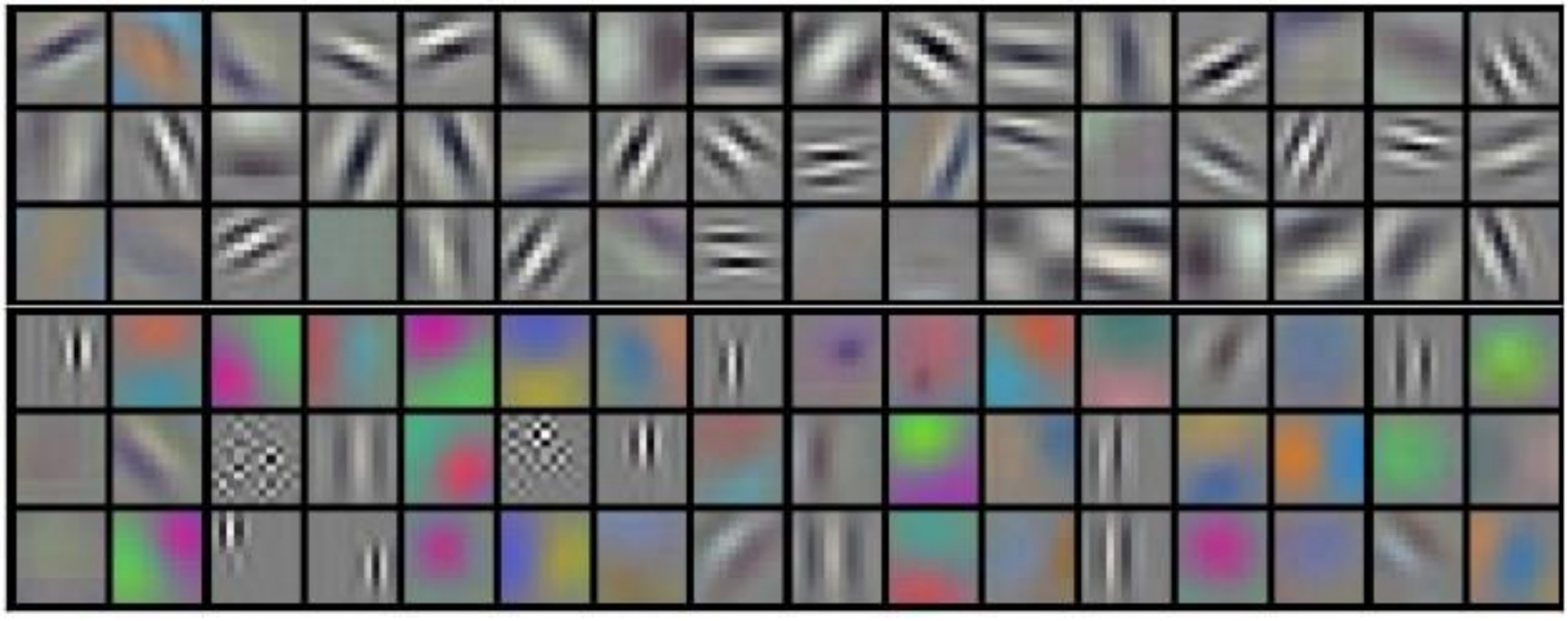
What features does the network learn?

- Typical features learned by one convolutional layer:



What features does the network learn?

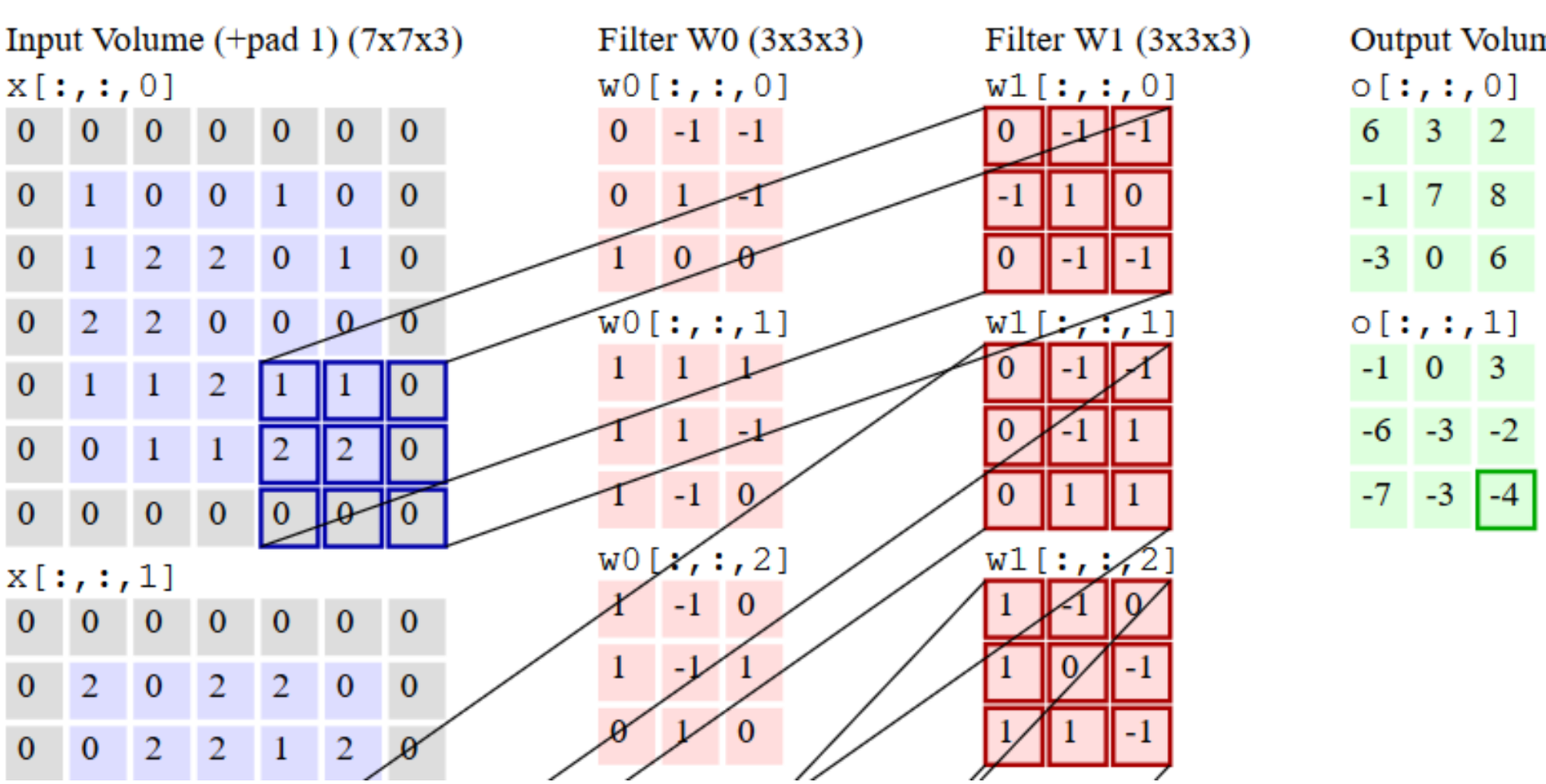
- Filers learned by AlexNet (first layer): size: 11x11x3



- Idea behind weight sharing: a pattern learned in a certain area of an image (e.g. a horizontal edge) may be useful in another area as well → reuse the filter everywhere on the image

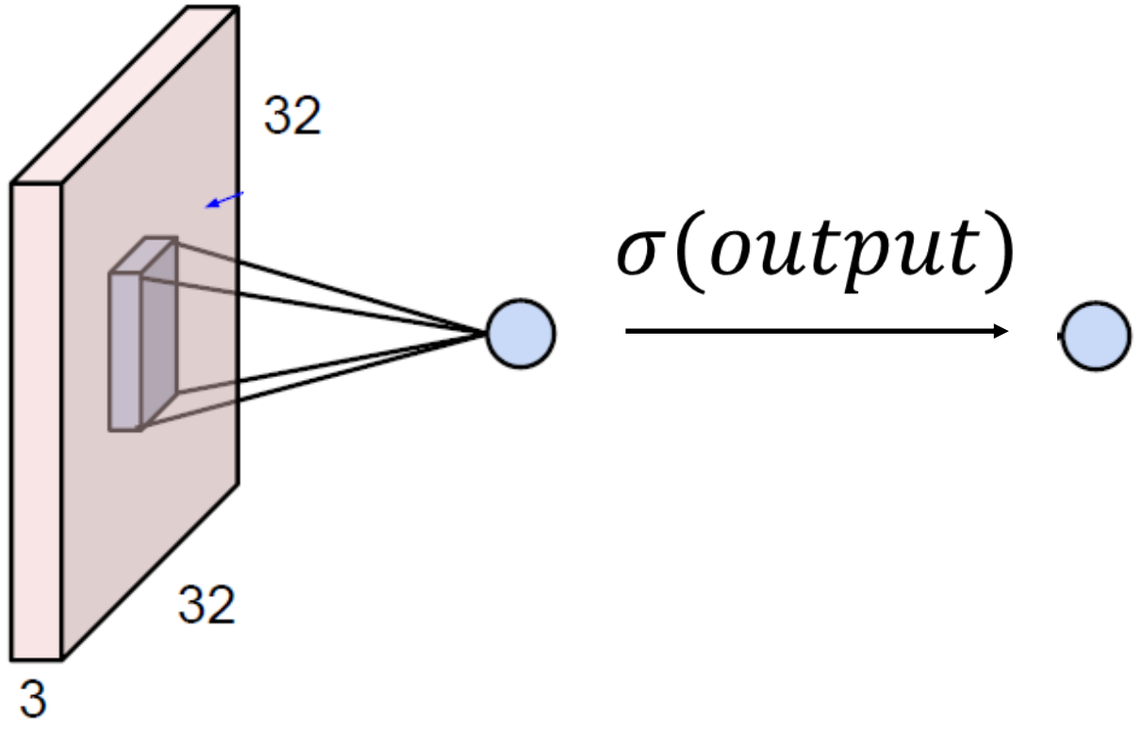
Further reading

- A very comprehensive summary is available here: https://cs231n.github.io/convolutional-networks/
- This includes also a live animation that explains the convolution process in a Conv-Layer



Where is the activation function gone?

- Activation functions are applied on the convolutional layers exactly the same way as for fully connected layers
- Just apply activation function to every element in the activation map!

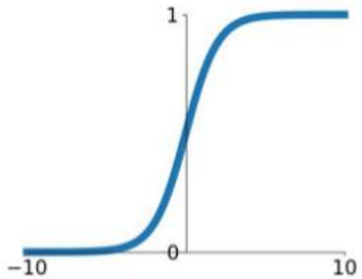


Note on activation function

- Convolutional layers usually use ReLU activation instead of sigmoid activation function, becasues it is much faster to compute and speeds up training!

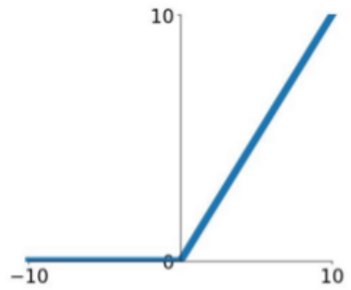
Sigmoid

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



ReLU

$$\max(0, x)$$



- Sometimes the activation function is considered a separate layer stacked on top of a convolutional layer

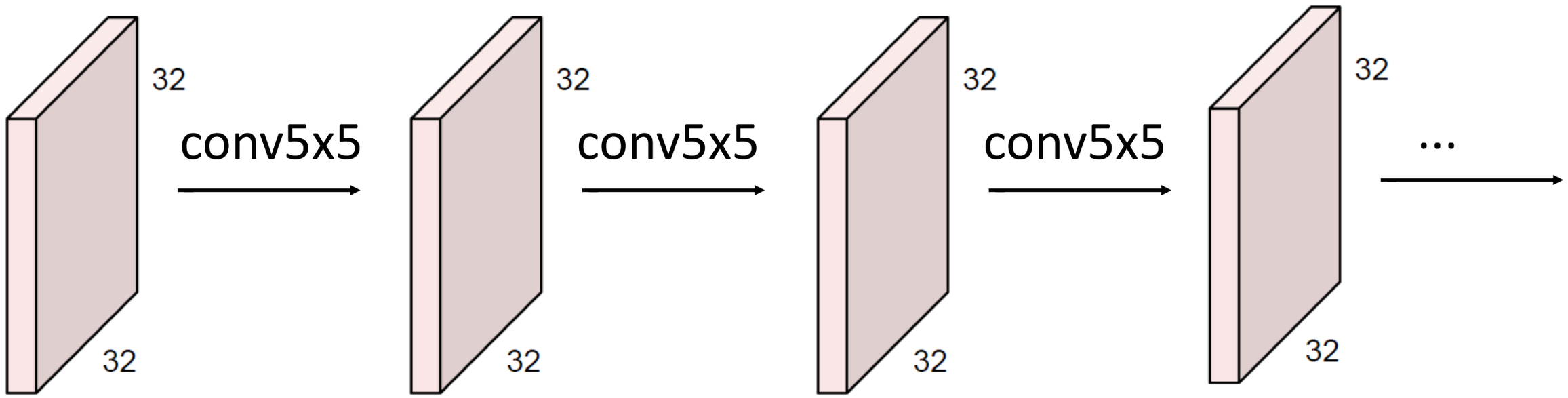
Summary on Convolutional Layers

- Main advantages (over dense layers)
  - The spatial structure is retained!
  - Weight sharing reduces parameters significantly
  - Neurons have local receptive field → enables to localize information in an image!

- By stacking them in a hierarchy, we can learn hierarchical representations that first learn simple patterns (e.g. edges) and then combine them to more complex patterns (e.g. contours).
  - But: for this, we need another concept: pooling

Stacking several Convolutional Layers to get a Convolutional Network (CNN)

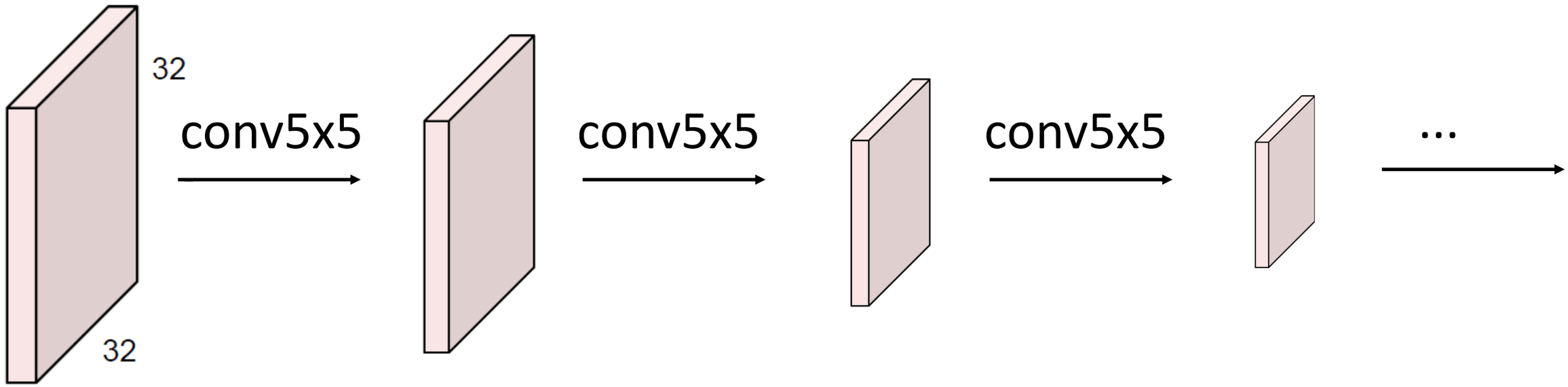
- Does this really make sense?, e.g.:



- Here: receptive fields remain constant
- To capture larger (more meaningful structures), the receptive field should increase!
- Down sample activation maps after each convolutional layer → **Pooling**

Stacking several Convolutional Layers to get a Convolutional Network (CNN)

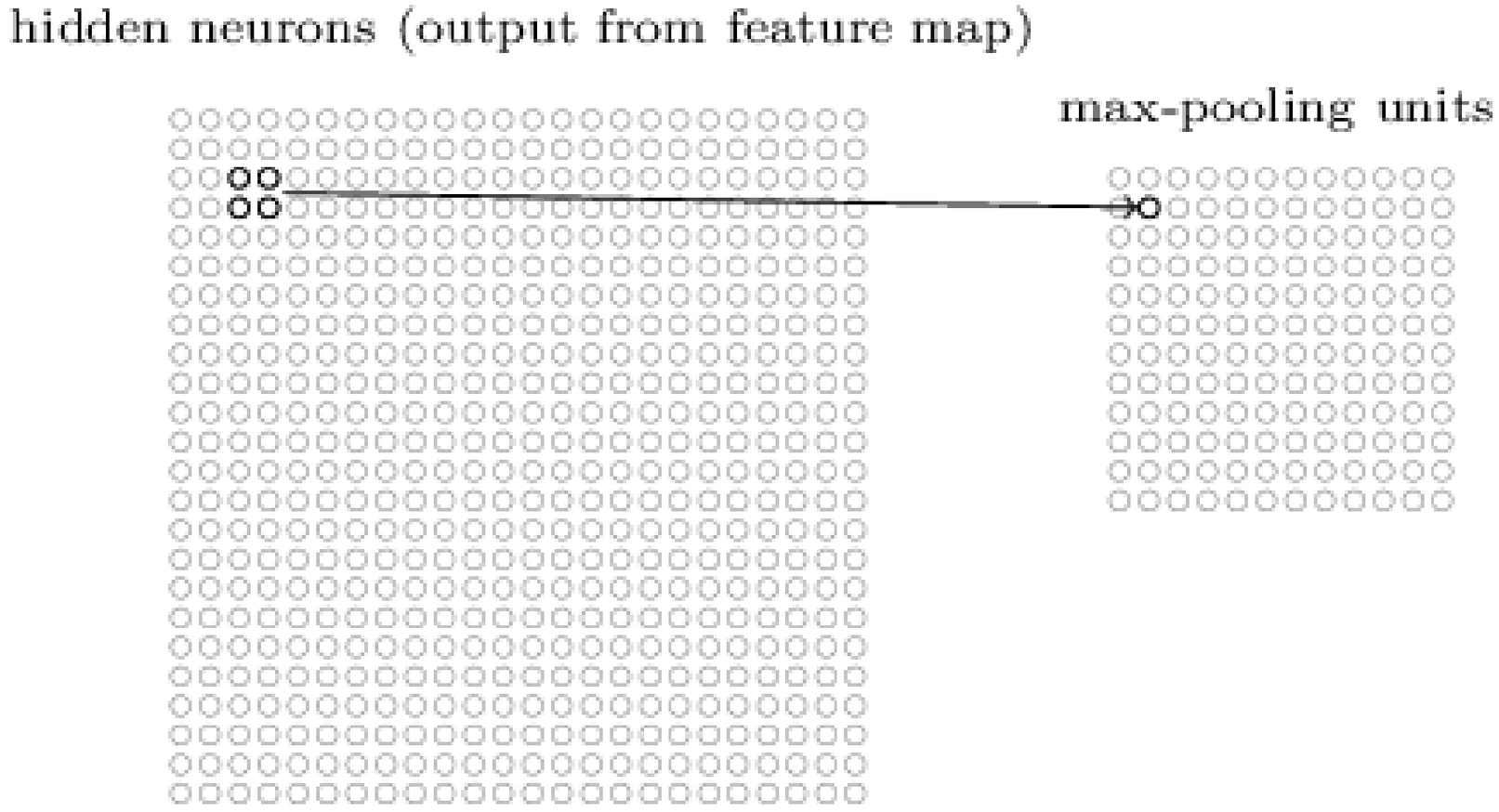
- Idea:



Pooling

- A convolutional layer operates on one scale of the input volume only
- Stacking several convolution layers of the same spatial resolution produces many parameters and impedes abstraction of the data
- Goal: aggregate information to abstract from the input

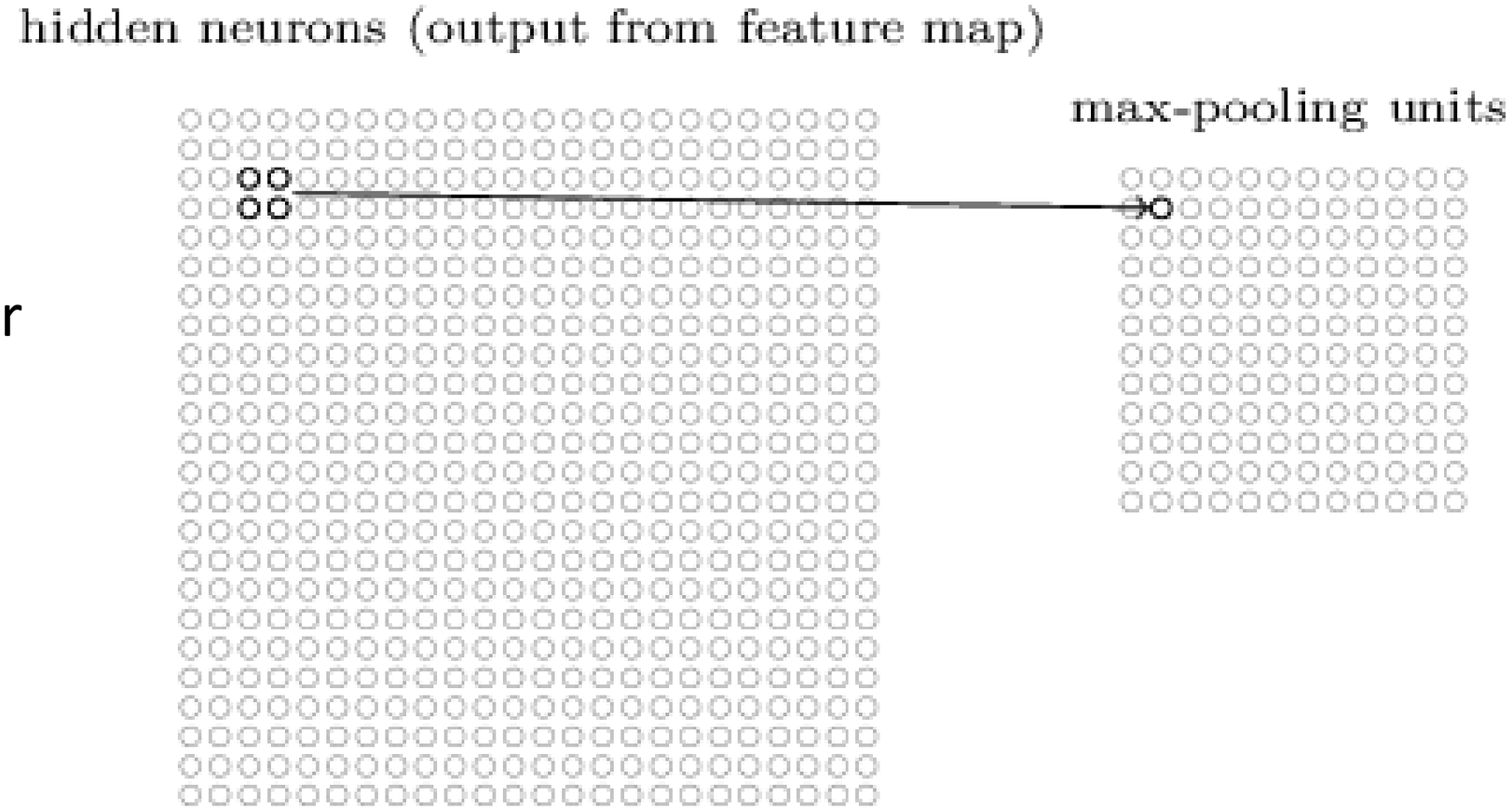
- Idea: simplify the information from the convolutional layer
- Use pooling to down-scale the data.



Pooling

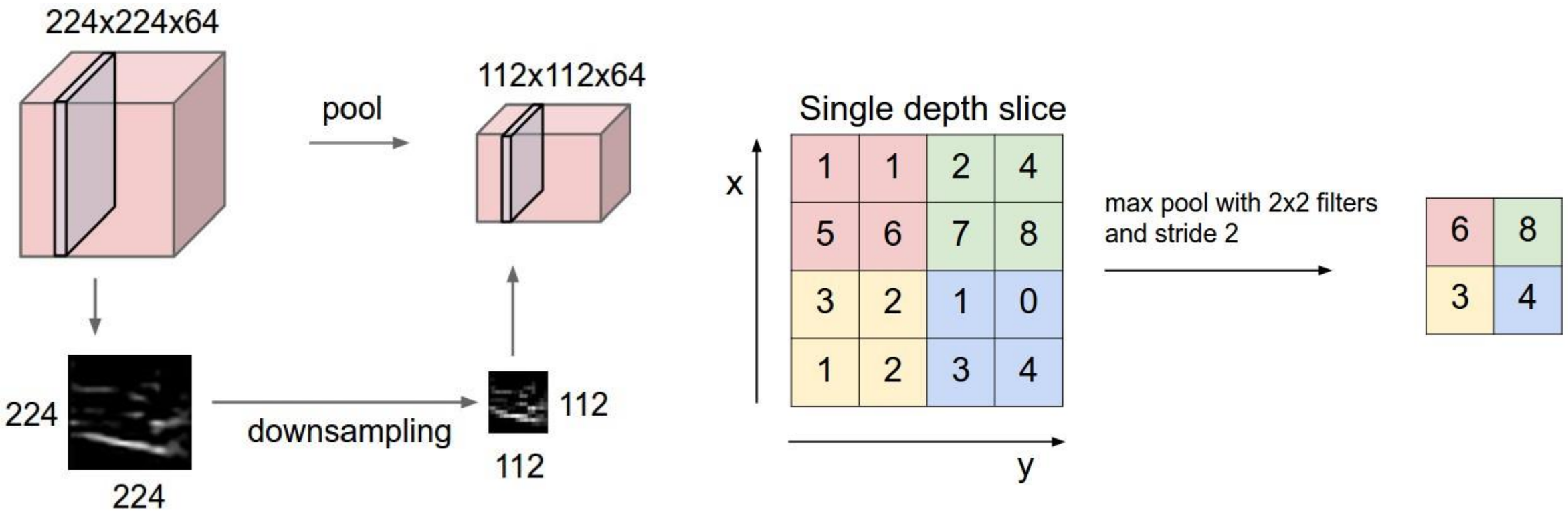
- Take output from convolution layer (=feature map) → reduce in size (subsample)
- Most-frequently used: max-pooling: take the maximum of the (here) four input values
- Convolutional layers are usually followed by “pooling layers”

- Example: 24x24 convolutional layer becomes a 12x12 layer



Pooling

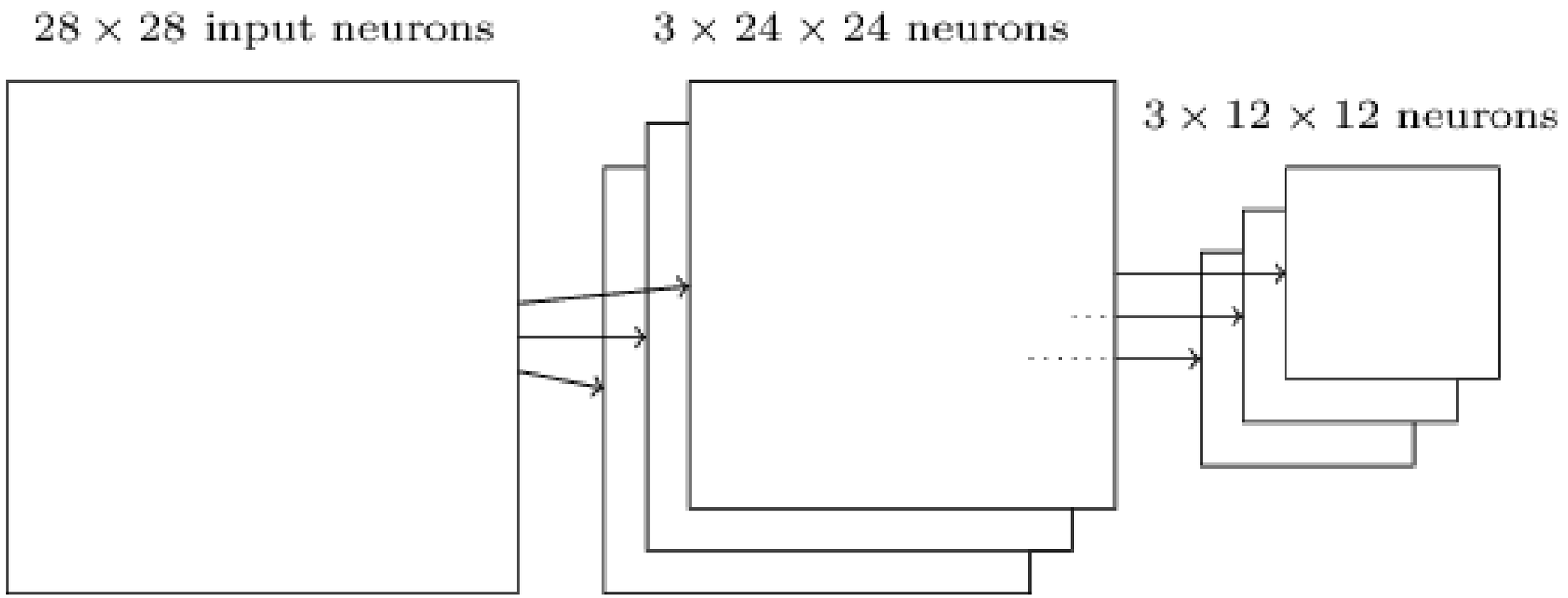
- Example:
  - 2x2 filter with stride (step size) = 2



- Overlap in pooling is possible, e.g. 3x3 filter with stride = 2
- Most common: 2x2 pooling without overlap
- Larger pooling filters destroy too much information

Pooling

- Apply pooling to all 3 feature maps



- Pooling operates independently on all feature maps

Pooling

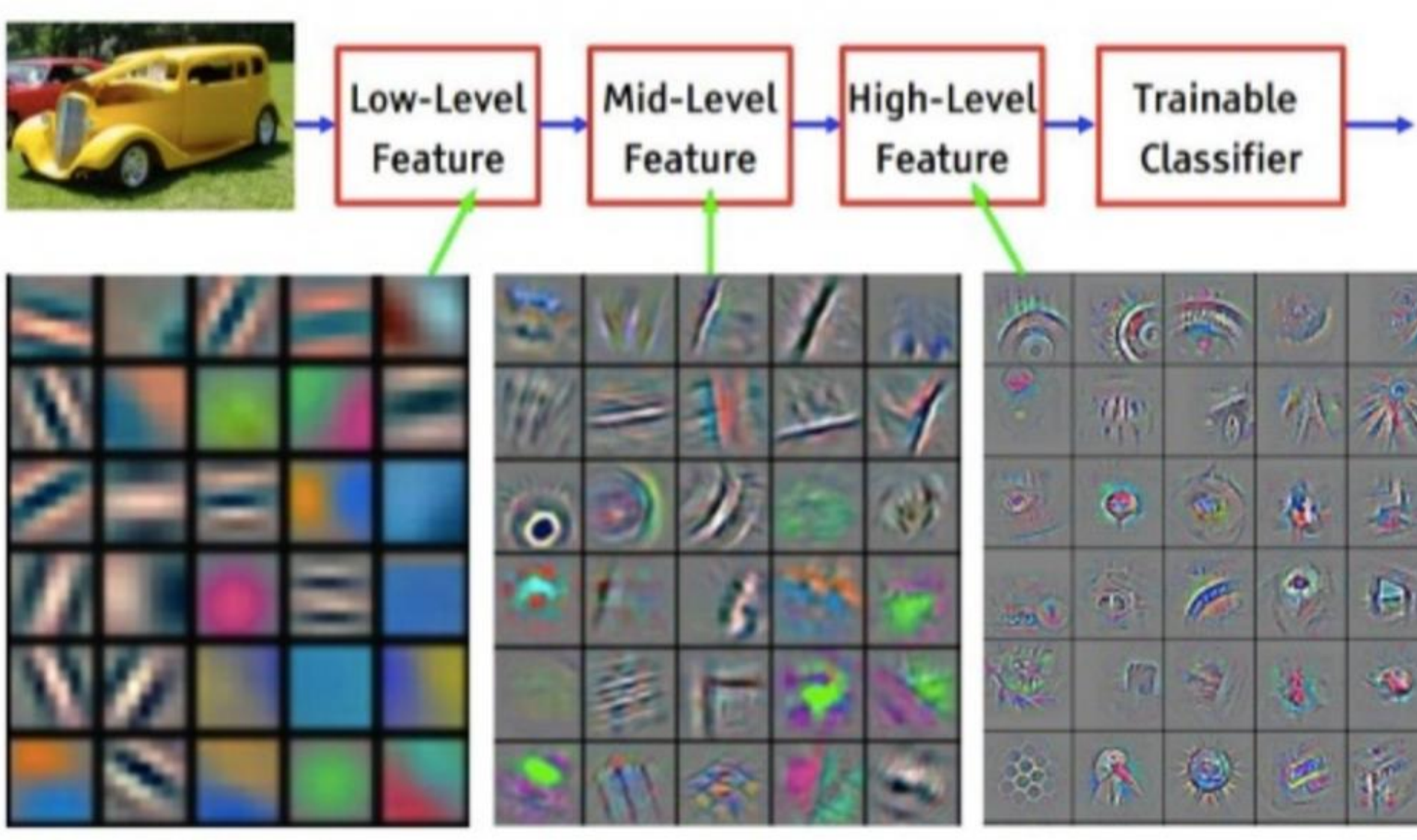
- Reduces the size of the subsequent layers (reduces number of weights and parameters)
- Has no free parameters (no learning necessary)
- We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image
- Spatial information is thrown away, only the information on the presence of the feature is retained. This introduces a certain translation invariance.
- Different types of pooling functions exist e.g.
  - max-pooling
  - average pooling

Pooling

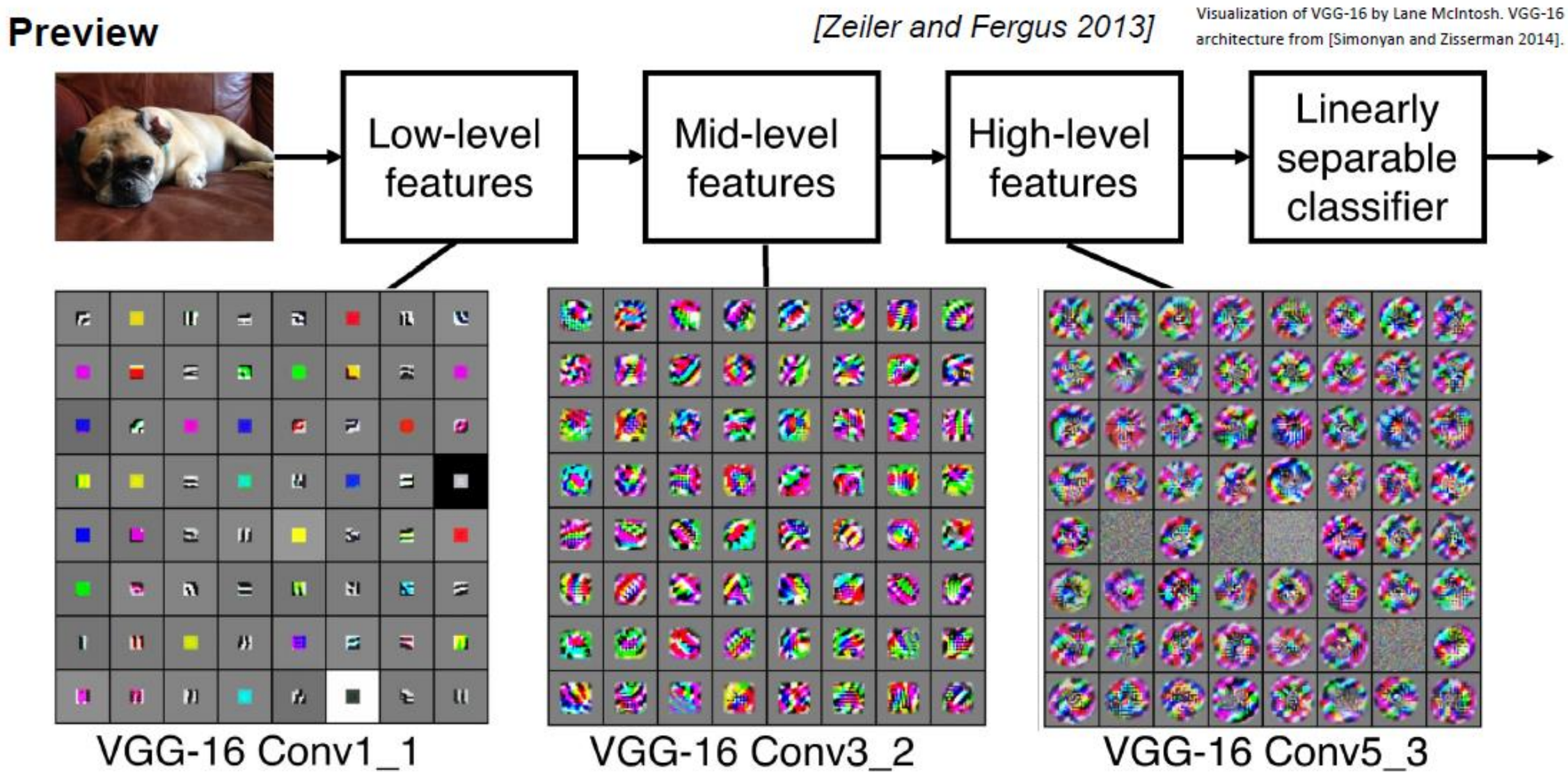
- Why not average pooling?
  - Keep the strongest activations in a local region → introduces some kind of location invariance
  - But: average pooling also possible, but in practice often max pooling better
- Why not using just stride in the convolutional layer to reduce size?
  - this is also a possibility. Also stride can be used to directly reduce the size of the input layers
- What does the Max Pooling layer learn?
  - Nothing. It has no trainable parameter
  - Just computes a fixed function for an input (similarly to an activation function)
- Typical values: filter size of 2x2, 3x3, stride: 2 or 3

Effect of multiple convolutional layers

- Learn hierarchical image representations that build upon shared visual primitives!



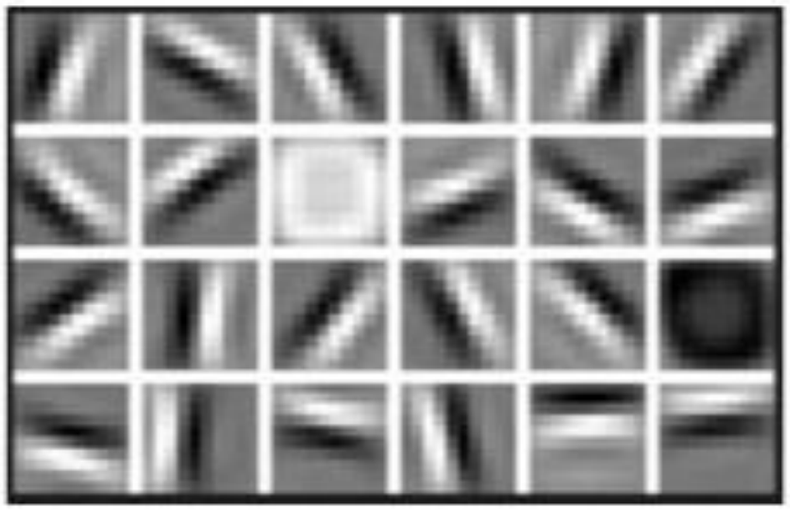
A hierarchy of filters



Note: each grid = 1 neuron, the content represents what type of input maximizes the activation at this neuron?

Effect of multiple convolutional layers

- Learn hierarchical image representations that build upon shared visual primitives!



First Layer Representation

Second Layer Representation

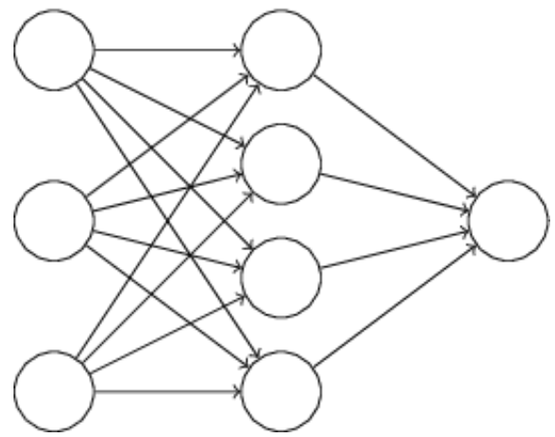
Third Layer Representation

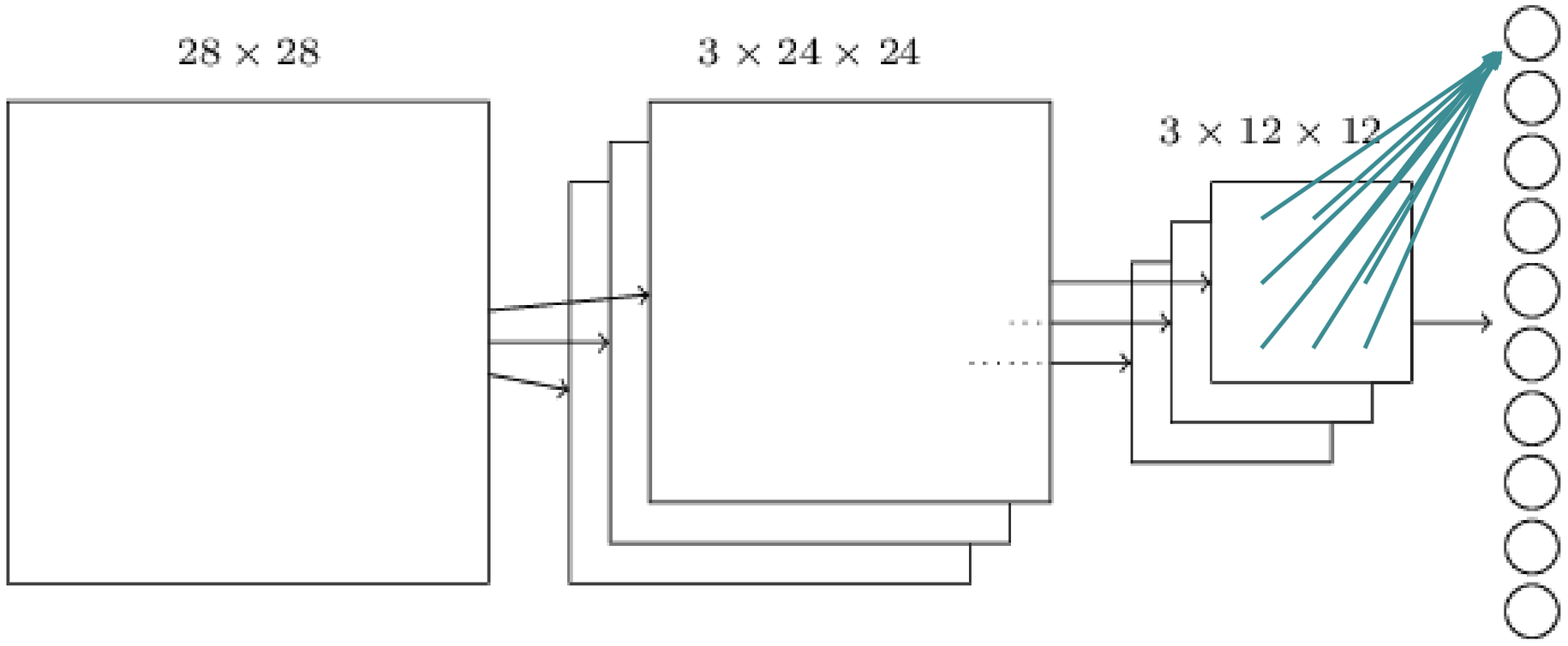
Pooling - Remarks

- For backpropagation: store for each pooling window which input was selected by the pooling operation.
- Enables to backpropagate the gradient to this specific input → more efficient and accurate
- Pooling can be replaced by convolution layers with strides>1, e.g. stride =2 halves the size of the input volume, see e.g. https://arxiv.org/abs/1412.6806

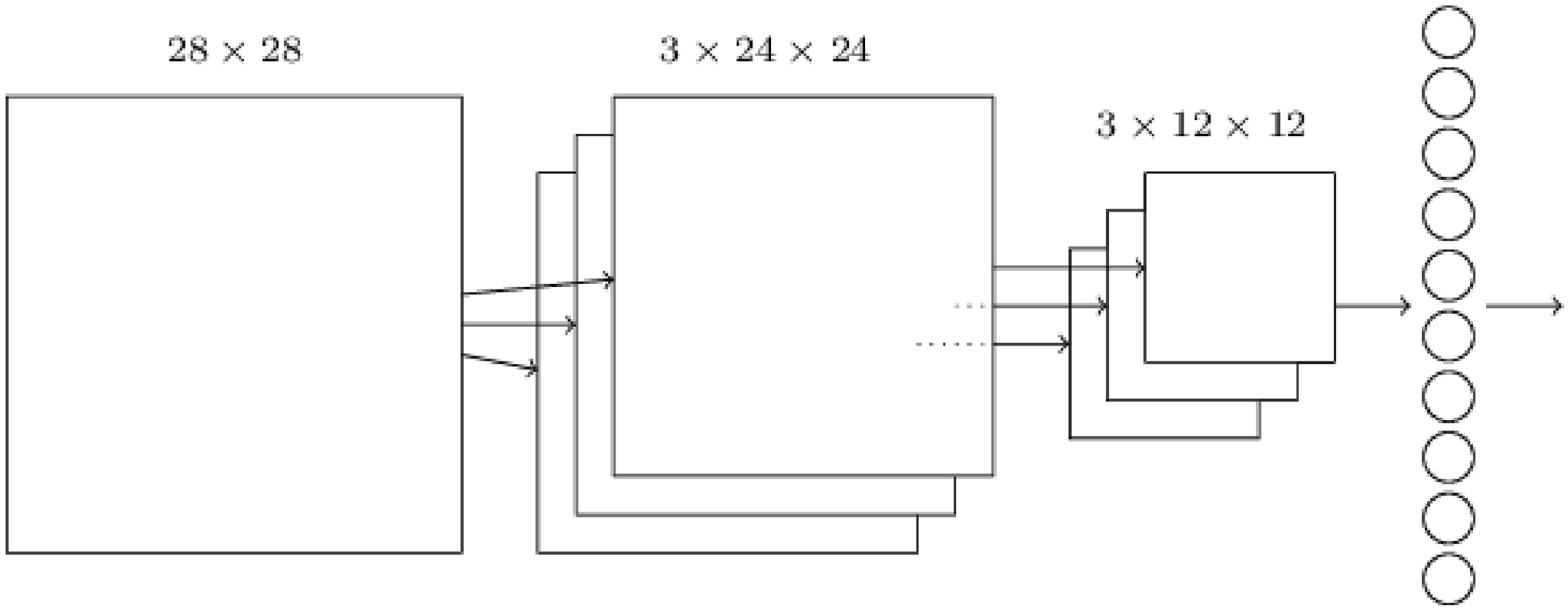
Fully-connected layer

- All neurons in the layer are connected to all neurons of the underlying layer
- One or more fully-connected layers often forms the output of a CNN
- Simply connect all neurons of the last feature map to a linear layer of neurons





An example convolutional neural network for handwritten digits

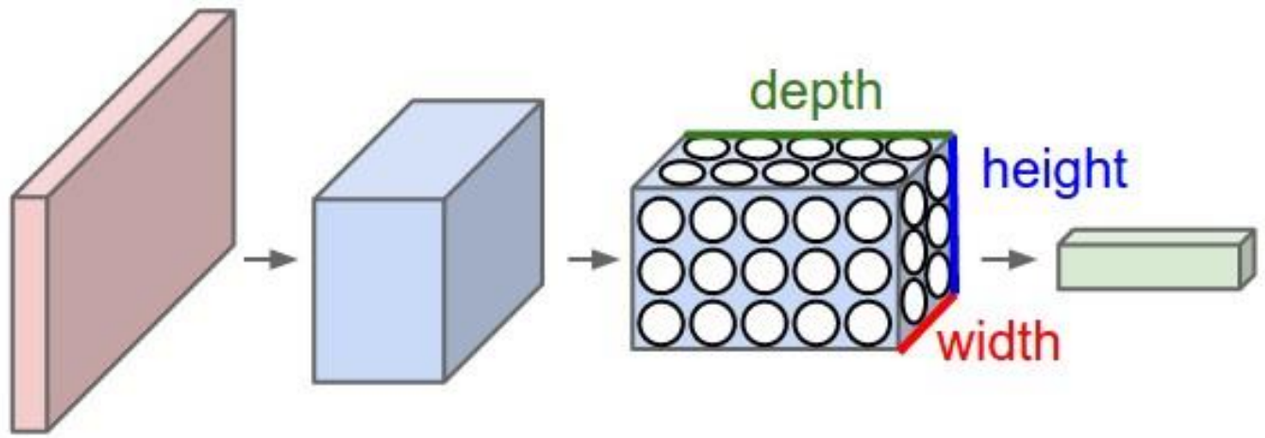


- Last layer is a “fully connected” layer that generates the 10 outputs that we need (for digits 0..9), i.e. every neuron connects to all neurons of the 3 12x12 feature maps, i.e. each neuron connects to 3x12x12 = 432 neurons (total number of connections: 10x432=4320)

Convolutional architecture

- Layer types
  - Convolutional layers
  - ReLU activation
  - Pooling layers
  - Fully-connected layers

CNN



- Typical sequence:

  INPUT → [[CONV → RELU]*N → POOL?]*M → [FC → RELU]*K → FC

- More details here: https://cs231n.github.io/convolutional-networks/

Considerations on filter size:

- Example:
  - A: Conv7x7 vs.
  - B: Conv3x3 -> ReLU -> Conv3x3 -> ReLU -> Conv3x3 (stride = 1)
- Both filters have receptive field of 7x7
- BUT: variant B has:
  - more flexibility due to non-linear units → more powerful features
  - less parameters (A: 3x7x7xC=147C vs. B: (3x3x3xC)x3=81C)

→ **replace larger convolutions by stacks of smaller ones**

Convolutional Layer in Keras

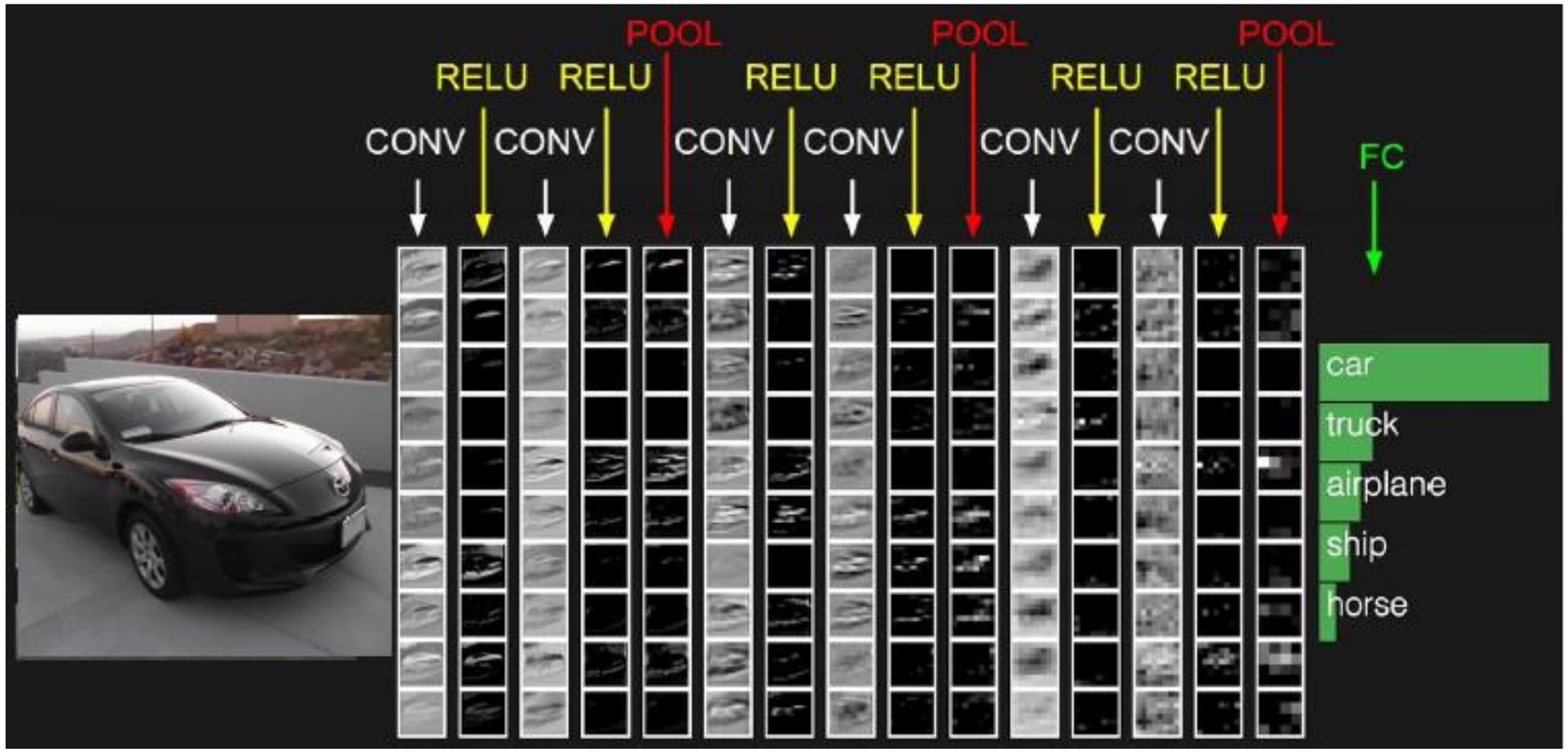
check:
https://keras.io/layers/convolutional/

```
model = Sequential()

model.add(Conv2D(n_filters=16, size=(5, 5), padding='same', strides=1,
input_shape=(640,480,3)))
# convolutional filter with size 3x3, stride 1 and 16 filters.
# input_shape is required in the first conv layer (connect to input img.)

model.add(Activation('relu'))  # ReLu activation

model.add(Conv2D(n_filters=32, size=(3, 3), padding='same', strides=1))
# convolutional filter with size 3x3, stride 1 and 32 filters.

model.add(Activation('relu'))  # ReLu activation
```

- How large is the activation map of the first convolutional layer?
  - 16 maps of size 640x480 (note that the filters are automatically made size 5x5x**3**)
- How large is the activation map of the second convolutional layer?
  - 32 maps 640x480 (filters are automatically extended to 3x3x16)

Typical architecture

- Several layer groups of the form: conv layer + ReLU
- After a couple of conv layers: pooling
- Last (and often also second last) layer: fully connected layers (we can think of this like a classifier that is put on top of the activation map of the last convolutional layer.



Popular architectures

- **LeNet**: first successful application of CNNs by Yann LeCun: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf
- **AlexNet**: made the breakthrough of CNNs in 2012 in the ILSVRC Challenge: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks
- **ZFNet**: ILSVRC 2013 winner which improves AlexNet slightly: https://arxiv.org/abs/1311.2901
- **GoogLeNet**: introduces the inception architecture which strongly reduces number of parameters: https://arxiv.org/abs/1409.4842
- **VGGNet**: demonstrates that deeper architectures are often better, uses only conv3x3 and pool2x2 and fully connected layers: http://www.robots.ox.ac.uk/~vgg/research/very_deep/
- **ResNet**: residual networks enable very deep architectures through skip connections: https://arxiv.org/abs/1512.03385

A typical image classification network

- LeNet [LeCun et al. 1998]

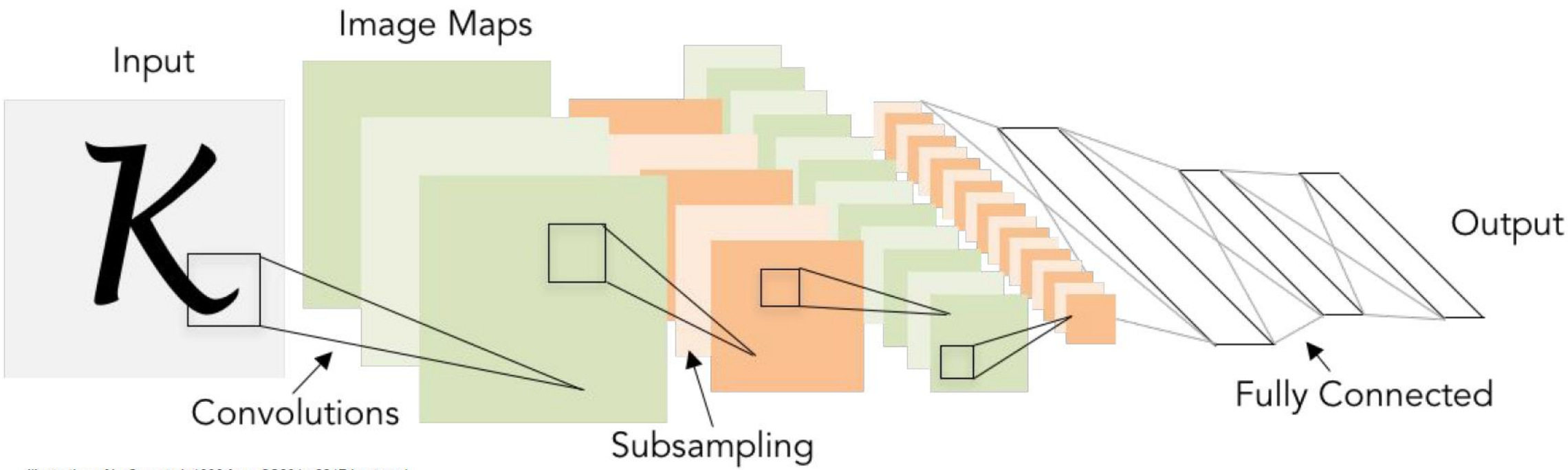
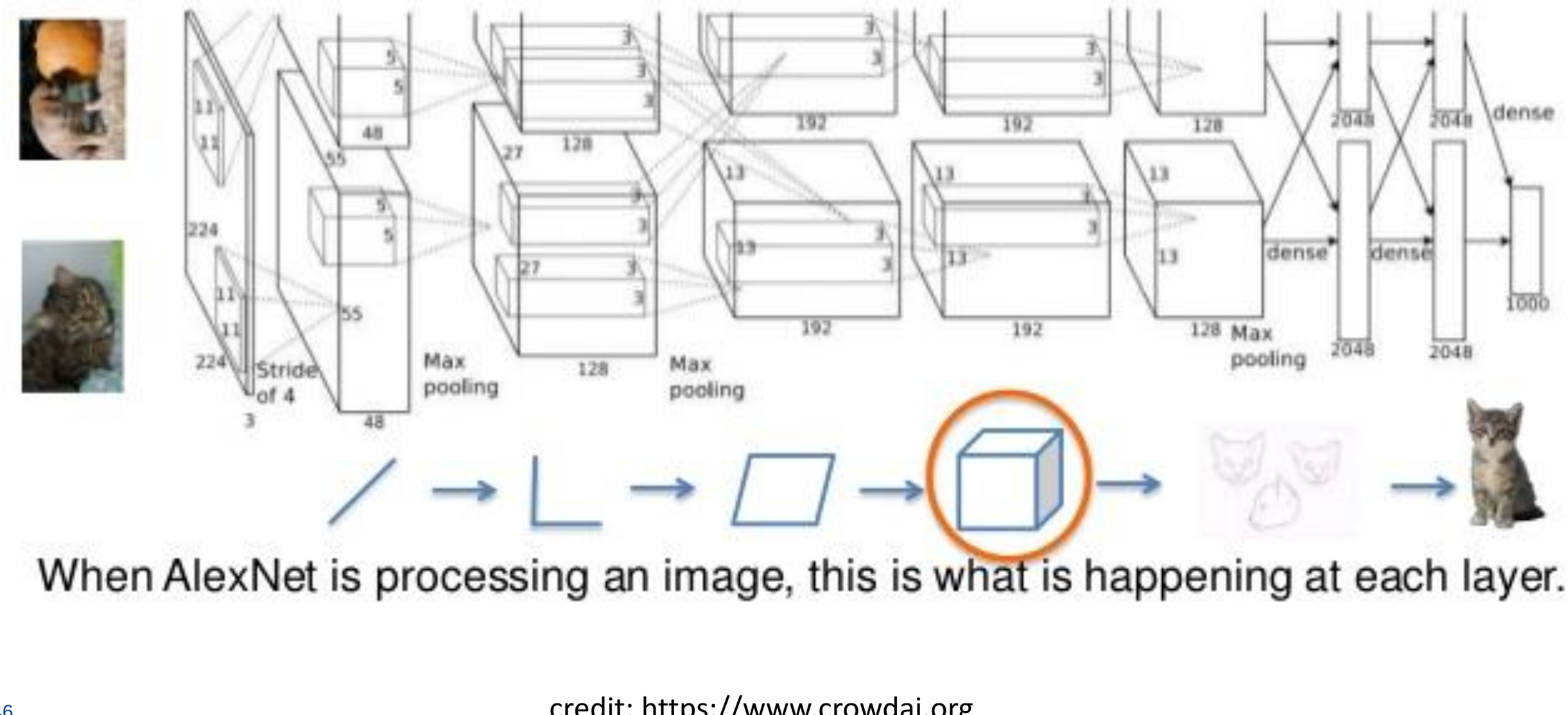


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

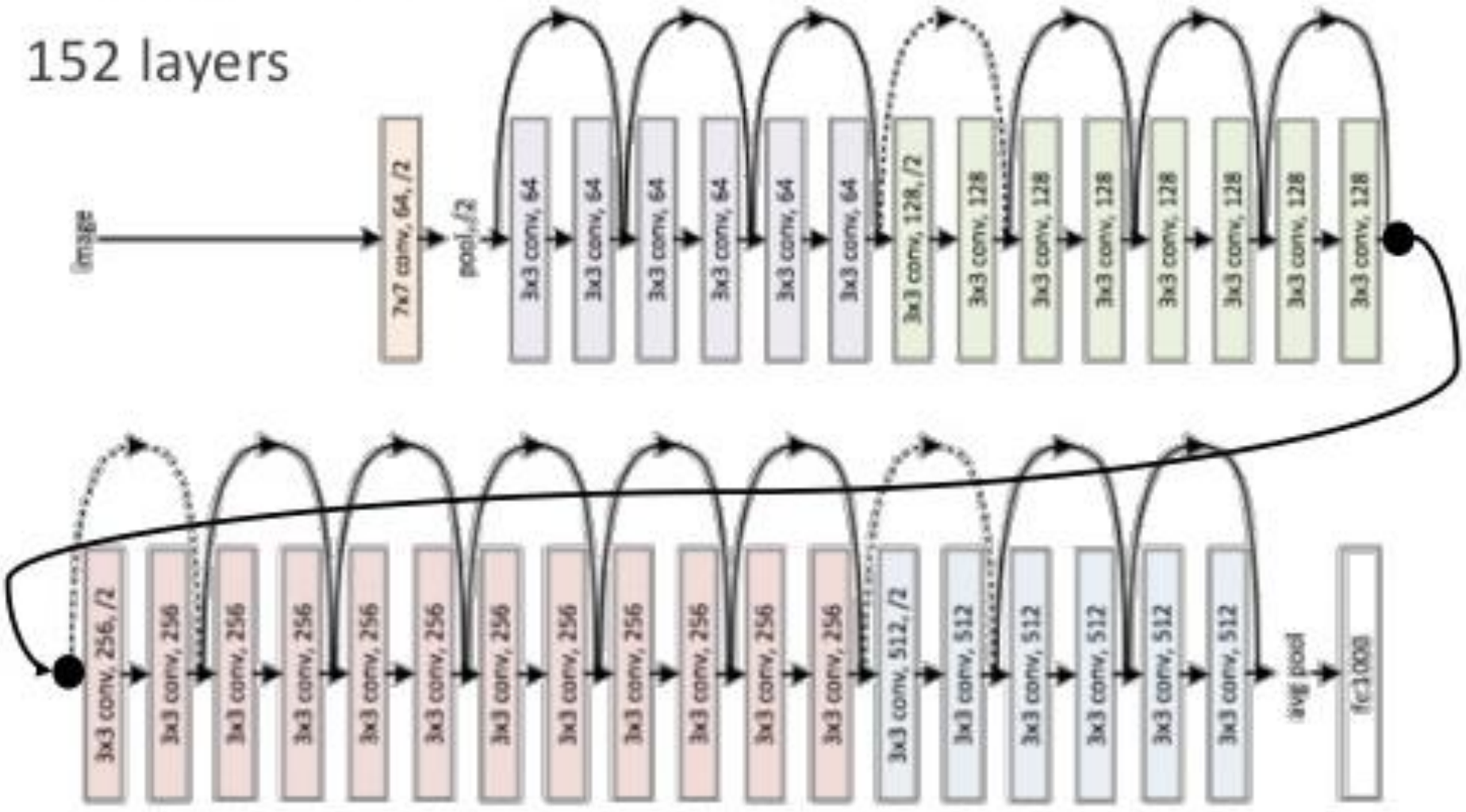
AlexNet



credit: https://www.crowdai.org

Residual Networks (2015)

- Paper: https://arxiv.org/abs/1512.03385
- Residual Networks
- 152 layers



credit: Mark Chang, https://www.slideshare.net/ckmarkohchang/applied-deep-learning-1103-convolutional-neural-networks

What is the downside?

- Hyperparameters / Architecture
  - It is not clear which architecture is the best
  - How many Conv-Layers to use?
  - Which filter sizes, strides, paddings?
  - How long to perform pooling (downsampling)?
  - Which activation function?
  - How many nodes in the dense layer?
- Try out different architectures, run cross-validation
- Always first analyze: which pattern do I want to detect? What is its size? How different can it look?
- Run network visualizations to see what the network actually learns

Outlook

- Setting up an experiment
- Preparing the data
  - Generating data splits (train, test, validation set)
  - Normalizing the input data (zero-mean data)
  - Data augmentation
- Prepare network
  - Load pre-trained weights
  - Batch normalization
  - Define loss function
- Define training hyper-parameters (learning rate, decay, momentum etc.)
- Regularization (dropout, regularization term)
- Understanding backpropagation through CNNs
- Run training / fine-tuning
  - Monitor training process
  - Detecting overfitting

State of the art

EE Times

HOME NEWS PERSPECTIVES DESIGNLINES VIDEOS RADIO EDUCATION IOT TIME

DESIGNLINES | INDUSTRIAL CONTROL DESIGNLINE

Microsoft, Google Beat Humans at Image Recognition

Deep learning algorithms compete at ImageNet challenge

By R. Colin Johnson, 02.18.15 14

Share Post Share on Facebook Share on Twitter G+ in

PORTLAND, Ore. -- First computers beat the best of us at chess, then poker, and finally Jeopardy. The next hurdle is image recognition — surely a computer can't do that as well as a human. Check that one off the list, too. Now Microsoft has programmed the first computer to beat the humans at image recognition.

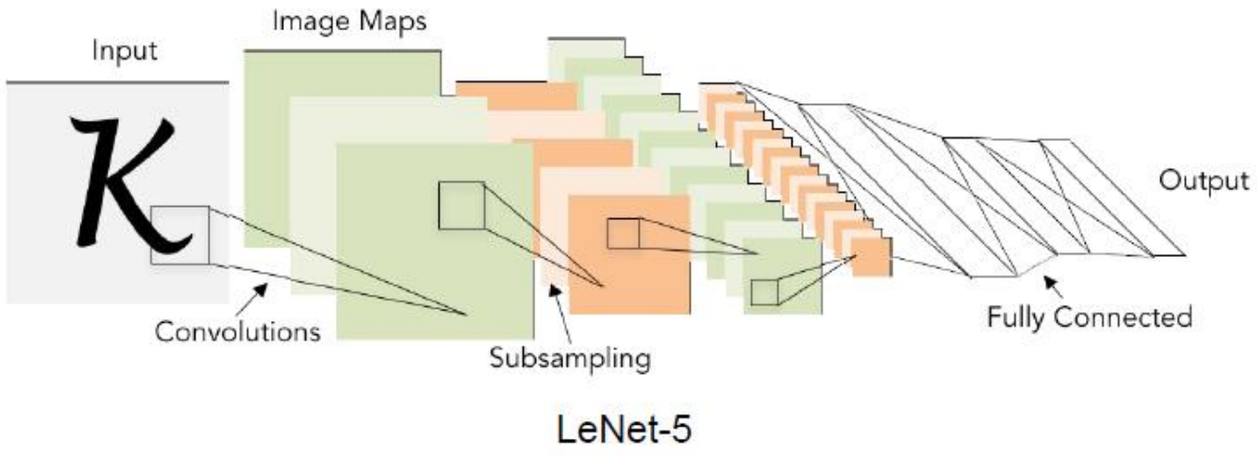
The competition is fierce, with the ImageNet Large Scale Visual Recognition Challenge doing the judging for

Exam

- Oral exam
- 2-3 questions from the topic areas of the course (see next slide)
- Time slots posted in eCampus
- Lecture contributes 50% to the final grade
- Remaining 50%: Labs + lab presentations + active cooperation
- Both parts must be positive for a positive grade

Topic Areas for Exam

- Challenges of CV
- Image Formation
  - Pinhole camera, aperture
  - How is an image generated
  - Color, color spaces
- Basic image operations
  - Arithmetic & Boolean operations
  - Image averaging
  - (Color-) Histograms
  - Thresholding, Contrast enhancement
  - Content-based image retrieval / Similarity Retrieval
- Convolution
  - Principle, applications, calculate a simple convolution, design a simple filter
- Smoothing filters, median filter
- Edge detection
  - Gradients, edge filters, Canny
- Local features
  - Corner detection (Harris)
  - Feature descriptors (SIFT)
  - Feature Matching
- ~~Panorama stitching~~
  - ~~Transforms, homography estimation, image warping~~
- Object recognition with BoW
  - Idea behind, sketch process, basics of machine learning
- Deep learning
  - Perceptron, sigmoid neuron, activation functions, multi-layer perceptron
  - Training a network, most important hyperparameters
  - CNNs, concepts, convolutional layers, pooling



LeNet-5

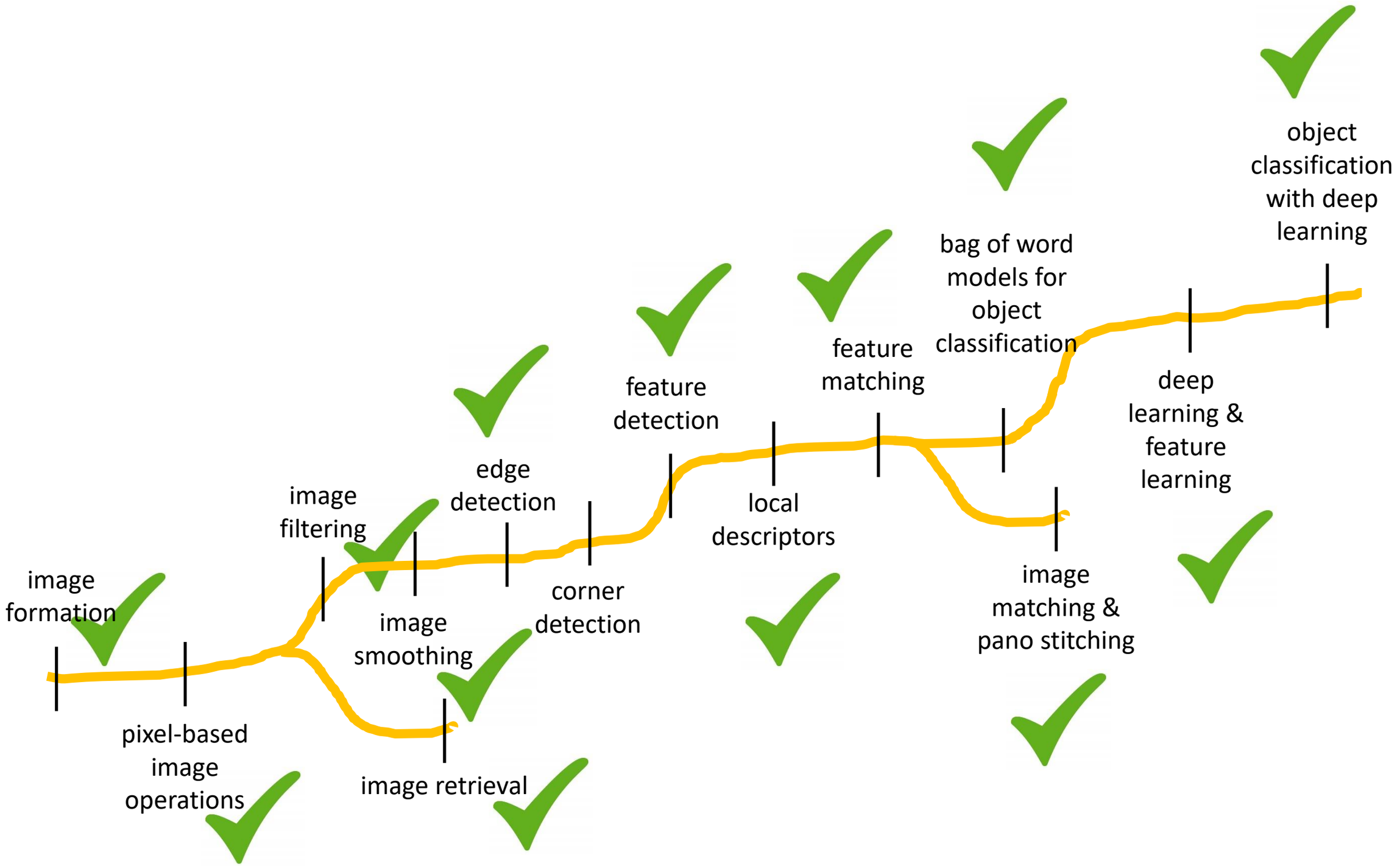
The End!

Questions → eCampus

Feedback → now | LV evaluation

More CV? → contact me
(studentische Mitarbeit, Master thesis)

Roadmap – We made it! ☺



The End!

Questions → eCampus

Feedback → now | LV evaluation

More Computer Vision? → contact me
(Berufspraktikum, studentische Mitarbeit...)