Robotics, Artificial Intelligence and Embedded Systems

Cognitive Systems: Neural Networks and Applied AI

Exercise 4
Deep Learning

Prof. Dr.-Ing. habil. Alois Knoll
Florian Walter, M.Sc.

Summer Semester 2019

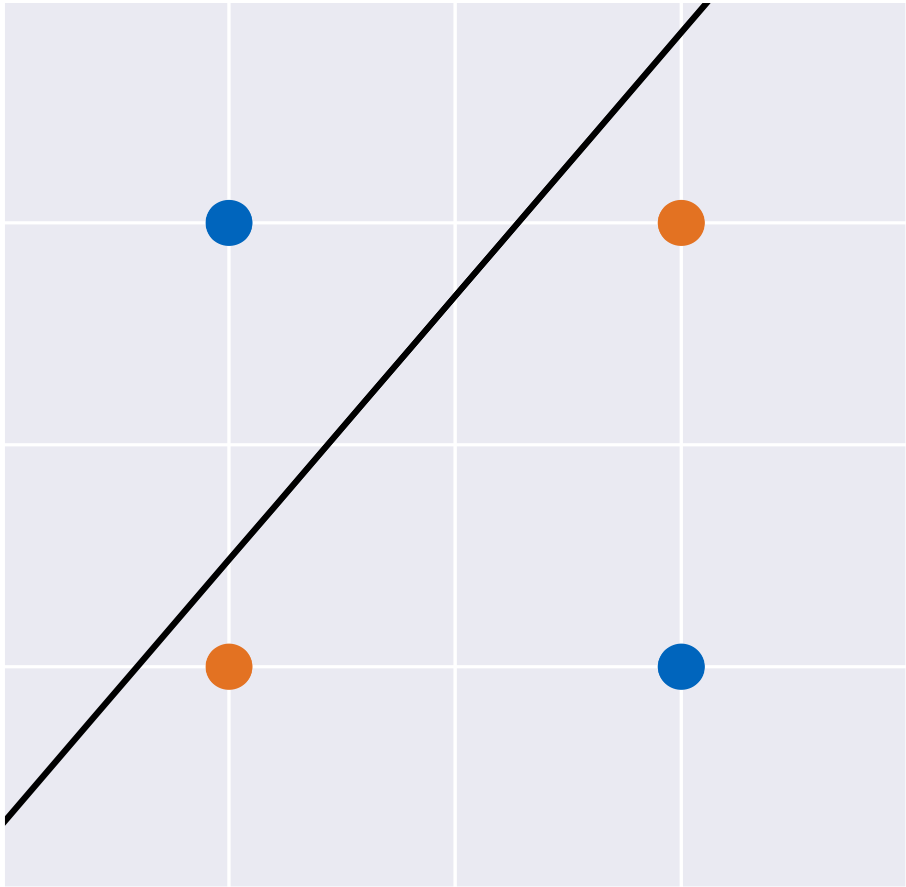
Topics in this Exercise

How do deep neural networks work?

- How to deep neural network learn?
- How is learning in artificial neural networks related to the brain?
- What do deep neural networks learn?

Recap: The XOR Problem

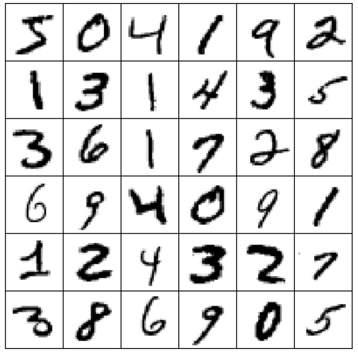
- The Perceptron learning rule only converges for linearly separable data sets
- It therefore cannot classify the XOR dataset correctly
- This finding led to an AI winter and the widespread abandonment of connectionism for almost two decades



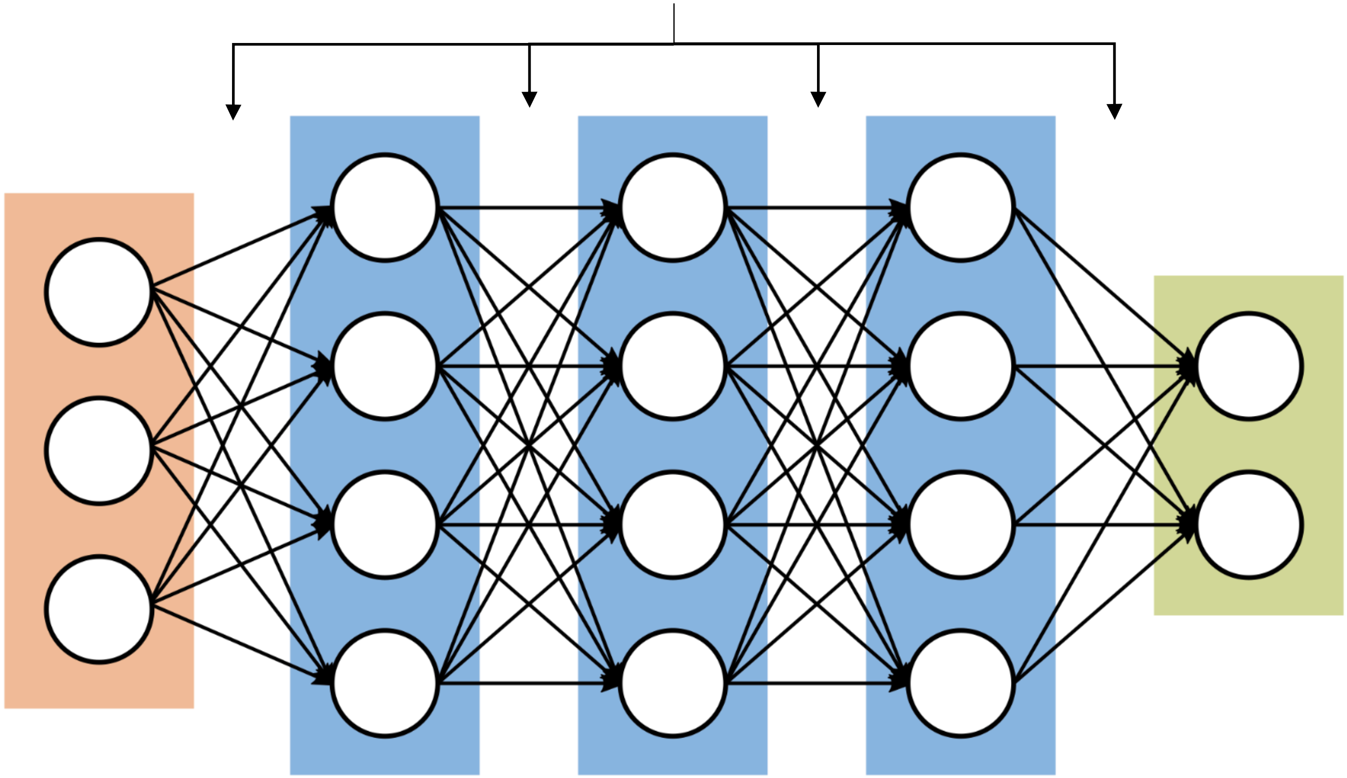
Neural Networks with Hidden Layers

Feedforward Neural Networks

Data



Synaptic Weights



Input Layer **Hidden Layers** **Output Layer**

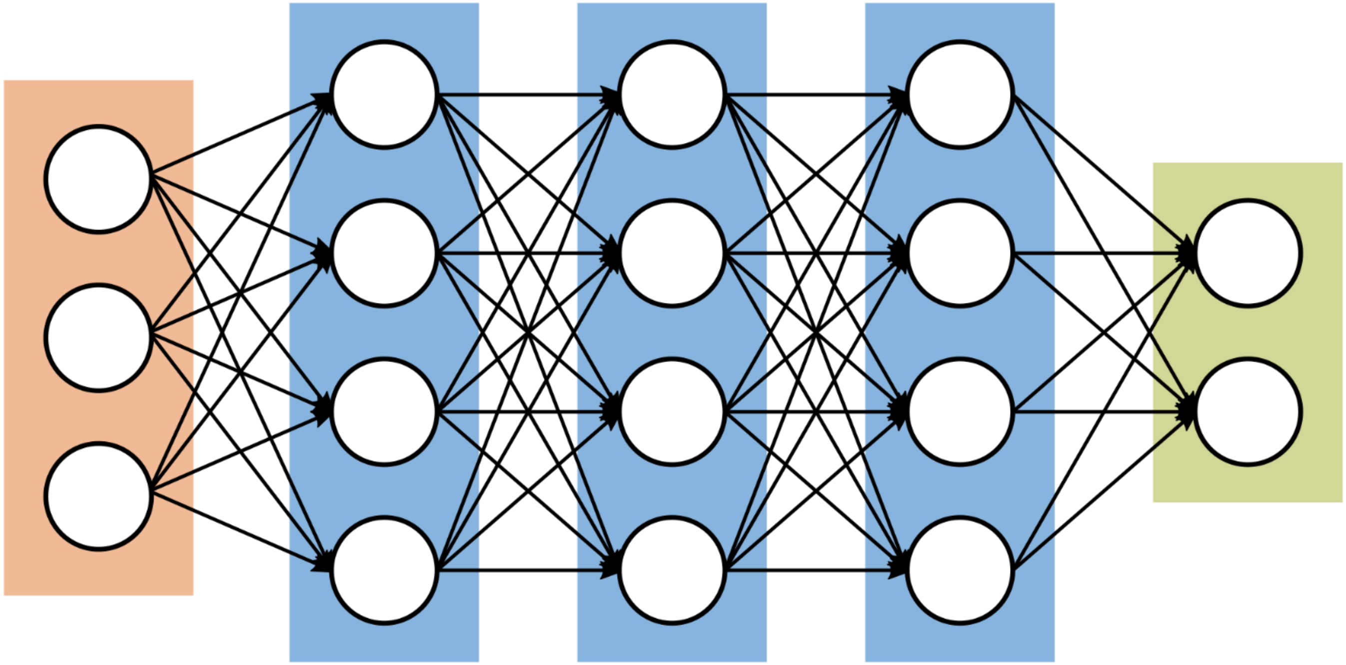
Labels / Predictions

5	0	4	1	9	2
1	3	1	4	3	5
3	6	1	7	2	8
6	9	4	0	9	1
1	2	4	3	2	7
3	8	6	9	0	5

Computing the Network Output

Synaptic Weights W_0 W_1 W_2 W_3

Input I



Network Output

$$O_4 = A(W_3^T O_3)$$

Layer Output

$O_1 = A(W_0^T I)$ $O_2 = A(W_1^T O_1)$ $O_3 = A(W_2^T O_2)$

The XOR Problem Revisited

- The proof of the limited computational capabilities of the Perceptron is only valid for single units
- It does not apply to cascades of Perceptrons that are organized in layered feedforward networks
- The two main questions therefore are:

What is the computational power of neural networks with hidden layers?

Which learning rules are available for these networks?

Universal Function Approximation

Feedforward networks with only a single hidden layer and appropriate activation functions are universal function approximators.

However, the universal function approximation theorem does not make any statement about …

- … which activation function works best
- … the number of hidden units required to get a sufficiently good approximation
- … how to set the weights

Neural Networks, Vol. 4, pp. 251–257, 1991
Printed in the USA.

0893-6080/91 $3.00 + .00

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Technische Universität Wien, Vienna, Austria

(*Received* 30 *January* 1990; *revised and accepted* 25 *October* 1990)

Abstract—*We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to* $L^p(\mu)$ *performance criteria, for arbitrary finite input environment measures* μ*, provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonconstant, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.*

Keywords—Multilayer feedforward networks, Activation function, Universal approximation capabilities, Input environment measure, $L^p(\mu)$ approximation, Uniform approximation, Sobolev spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural network architectures have recently been investigated by many authors, including Carroll and Dickinson (1989), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989, 1990), Irie and Miyake (1988), Lapedes and Farber (1988), Stinchcombe and White

measured by the uniform distance between functions on X, that is,

$$\rho_{\mu,X}(f, g) = \sup_{x \in X} |f(x) - g(x)|.$$

In other applications, we think of the inputs as random variables and are interested in the *average performance* where the average is taken with respect to the input environment measure μ, where $\mu(R^k) < \infty$

The Computational Power of Neural Networks

- The universal function approximation theorem guarantees that a simple feedforward neural network of appropriate size is in principle capable of computing any computable function
- This means that, in general, there is no need for a specific network architectures (we will see that this statement is of rather theoretical nature) …
- … and that neural networks can in principle replace many other AI models and algorithms

However, the theorem does not state how a neural network that approximates a certain function can be constructed

The Backpropagation Algorithm

Computing the Network Output Error

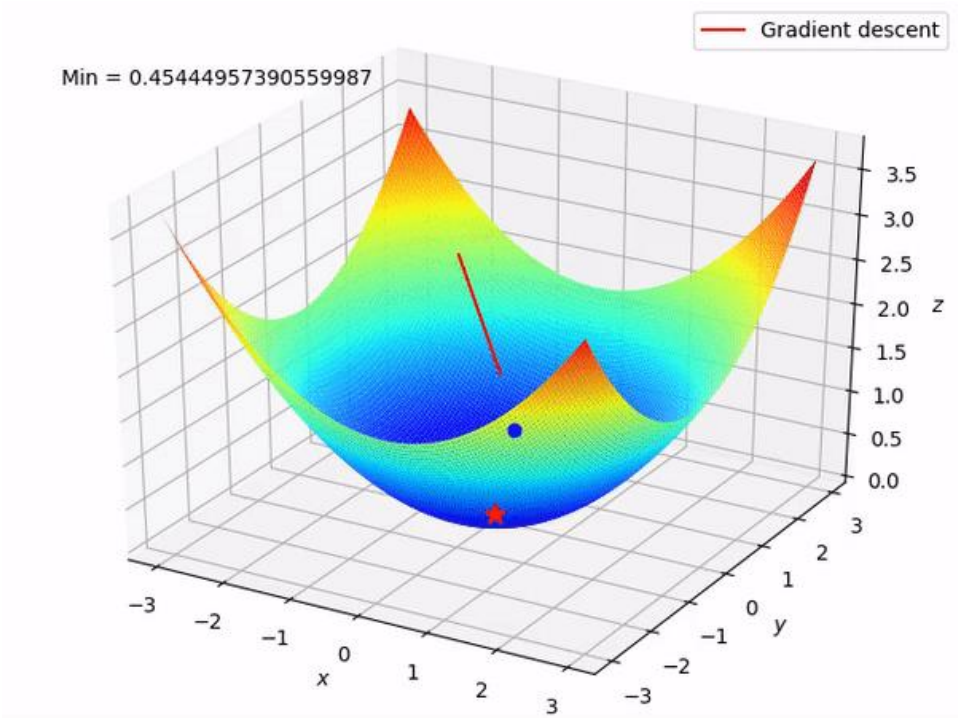
A first step to assess the performance of a given neural network N with weights W on a dataset is to compute an error function such as the sum of squared errors:

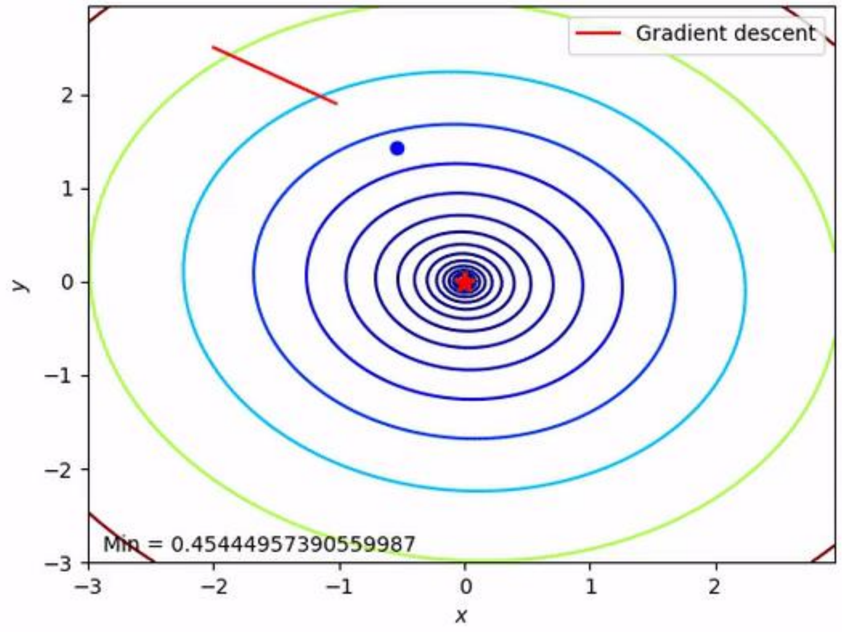
$$\mathcal{L}_{\mathcal{S}}(W) = \frac{1}{2}\sum_{n=1}^{N}(N_W(x_n) - y_n)^2$$

- $\mathcal{L}_{\mathcal{S}}(W)$ directly relates prediction errors to synaptic weights
- $\mathcal{L}_{\mathcal{S}}(\cdot)$ can be minimized by adjusting the synaptic weights W with standard optimization methods such as gradient descent
- However, the computational complexity becomes very high for large networks with many connections

Gradient Descent – A Simple Example

Finding the Minimum of a Quadratic Function





Most optimization methods in machine learning search for local minima

Animations from https://jed-ai.github.io/py1_gd_animation/

The Challenge

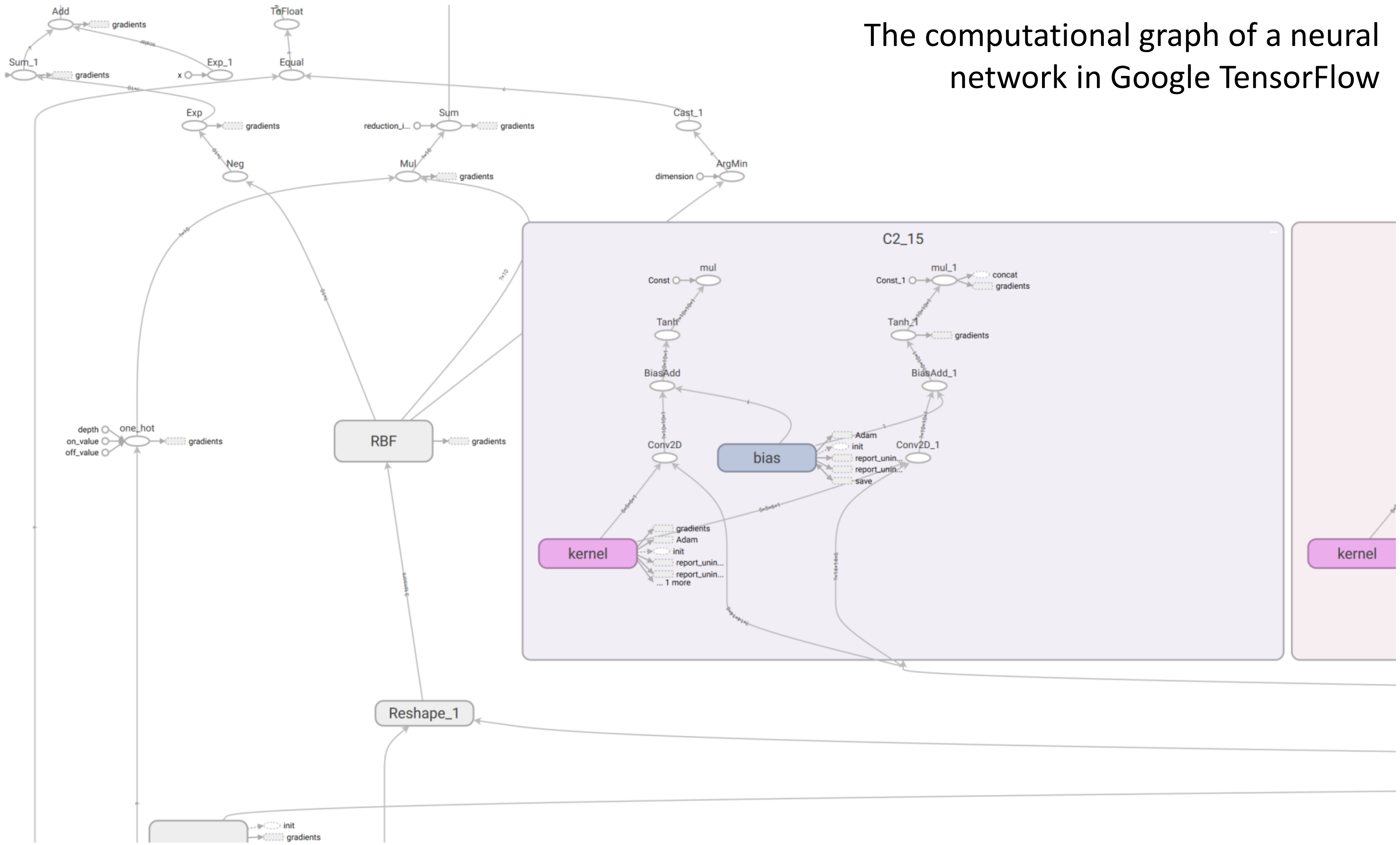
- State-of-the-art deep neural networks have up to millions of synaptic weights
- Computing the gradient means computing the partial derivative for every single weight

Naïve approaches for gradient computation will not scale to practically relevant problem sizes!

Neural Networks as Computational Graphs

The data flow and the computations in a neural network can be represented as a computational graph.

This graph representation is the basis for simple and efficient network specification and training.



The computational graph of a neural network in Google TensorFlow

A Simple Example

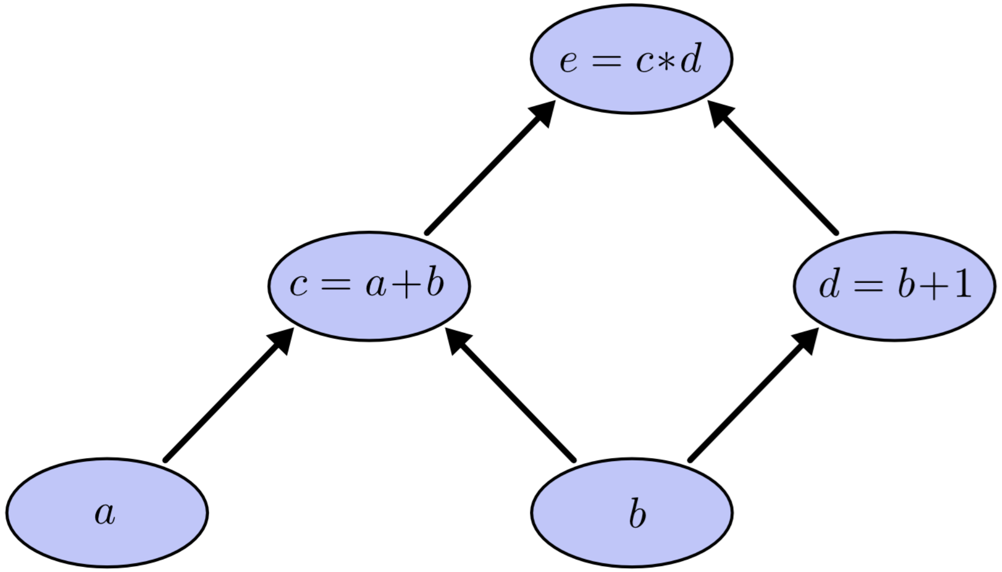
Consider the following formula:

$$e = (a + b) * (b + 1)$$

After decomposition into individual terms…

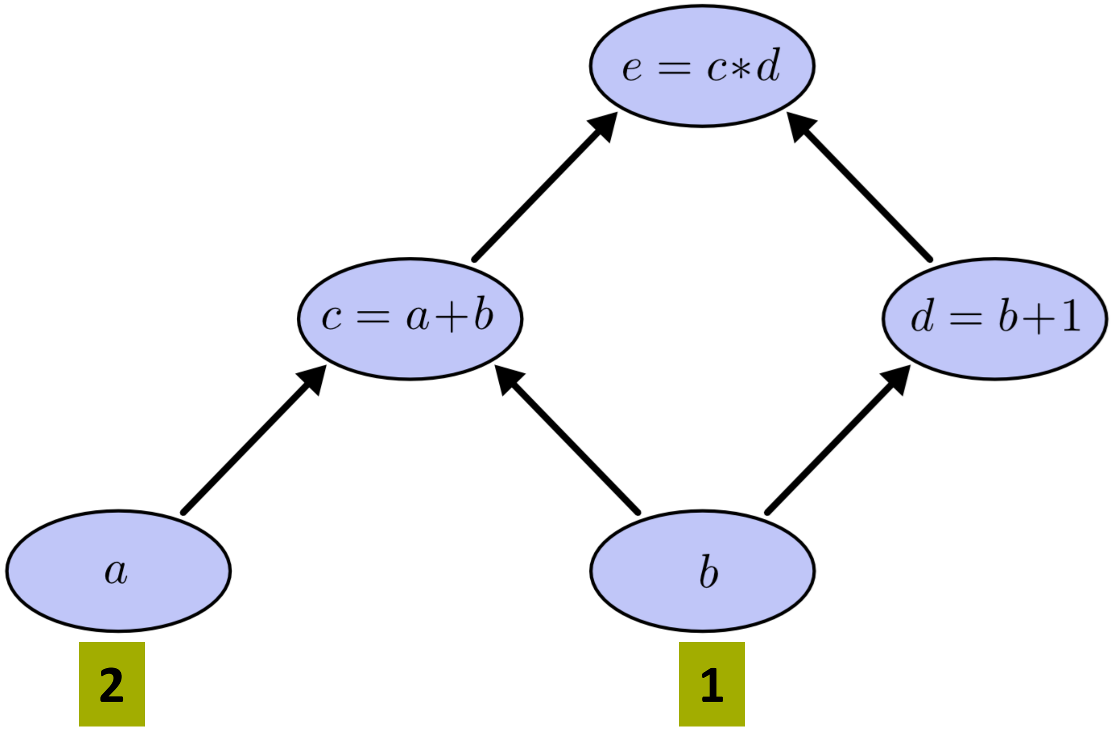
$$c = a + b$$
$$d = b + 1$$
$$e = c * d$$

… the arithmetic operations can be structured as shown in the graph on the right.



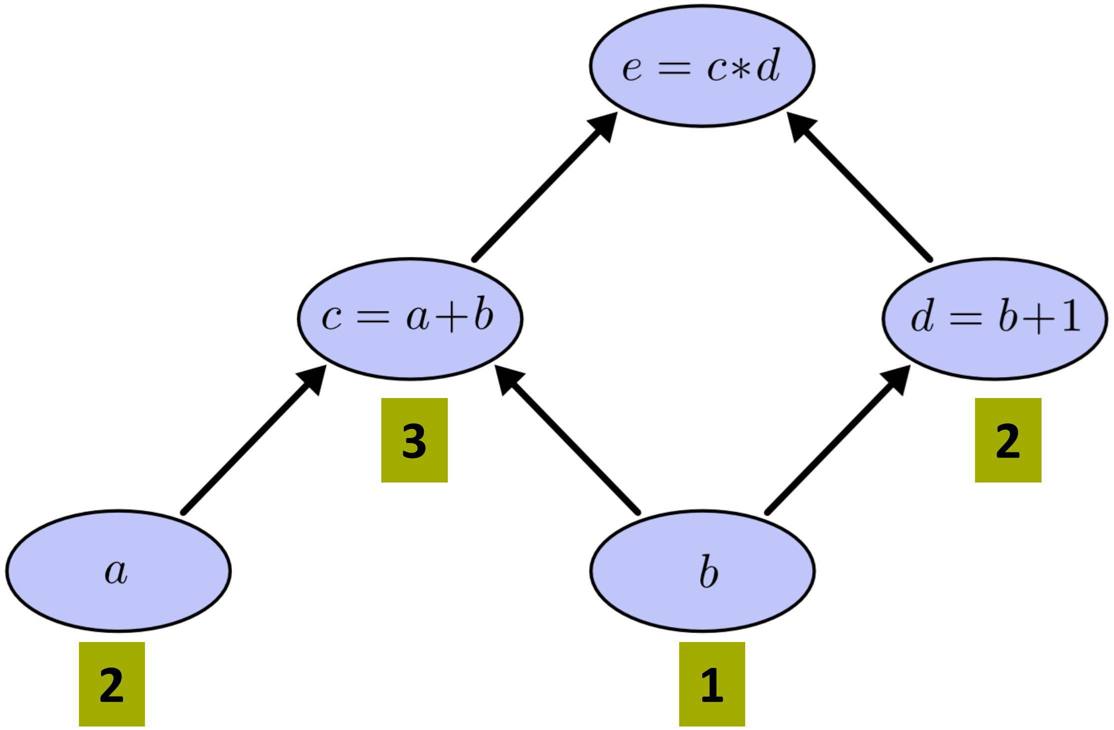
Examples and illustrations from http://colah.github.io/

Graph Evaluatioin



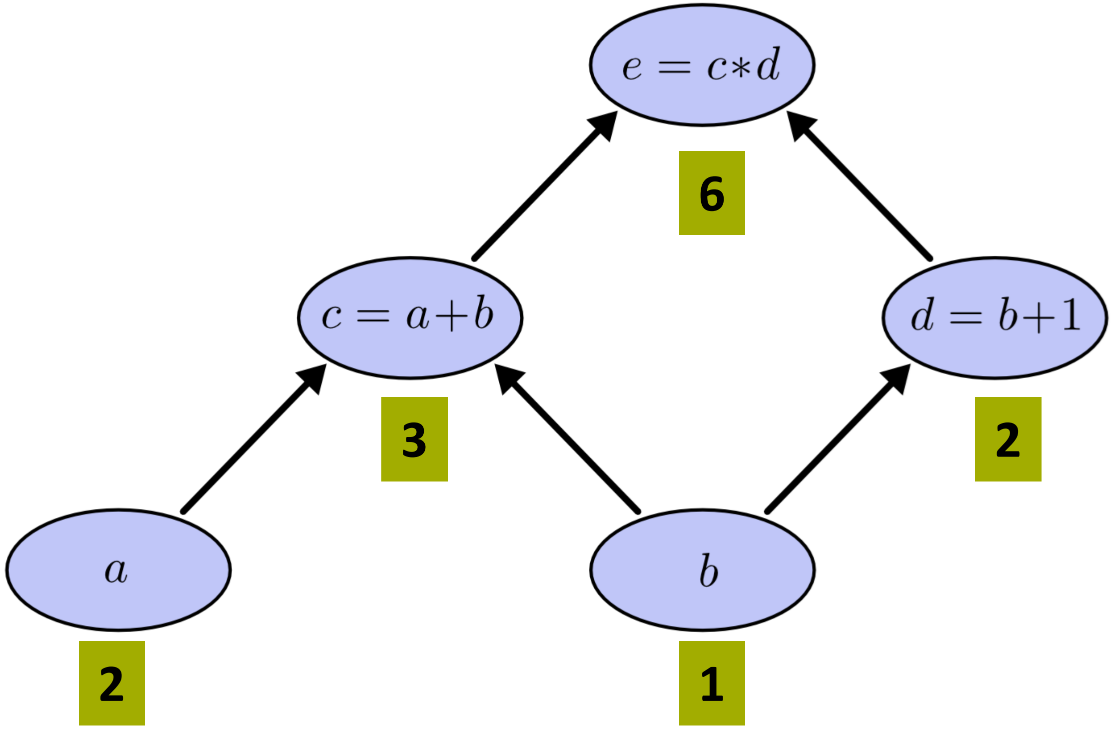
Examples and illustrations from http://colah.github.io/

Graph Evaluation



Examples and illustrations from http://colah.github.io/

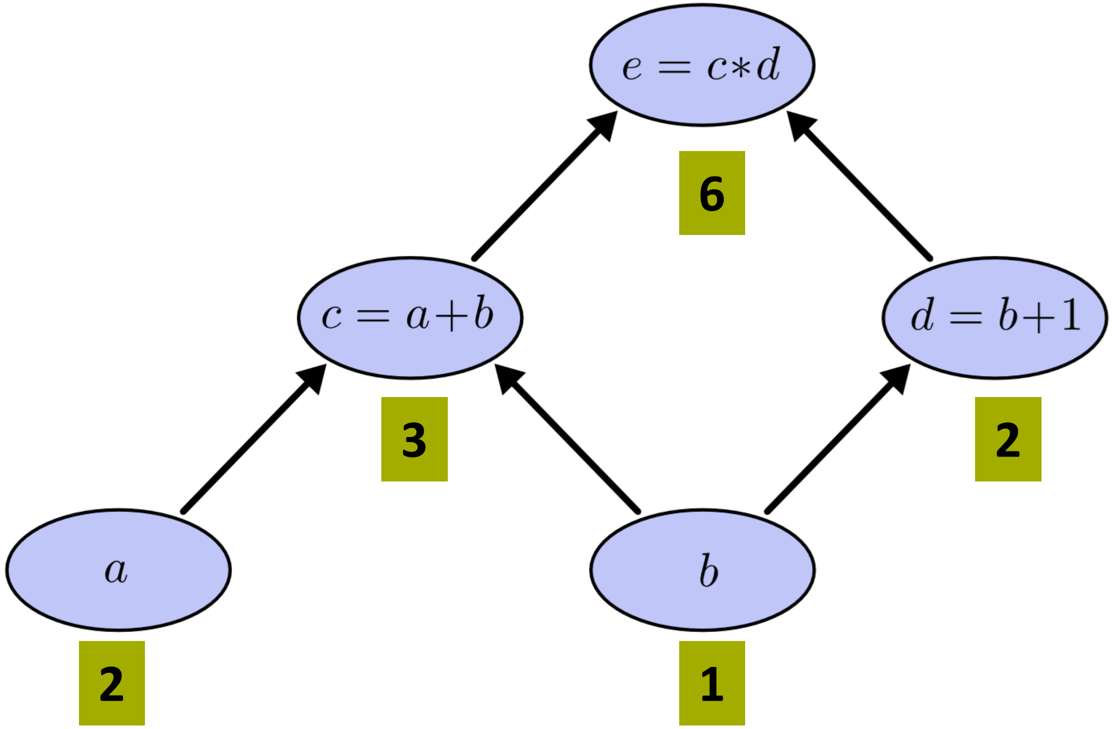
Graph Evaluation



Examples and illustrations from http://colah.github.io/

Graph Evaluation

Data “flows” through the computational graph



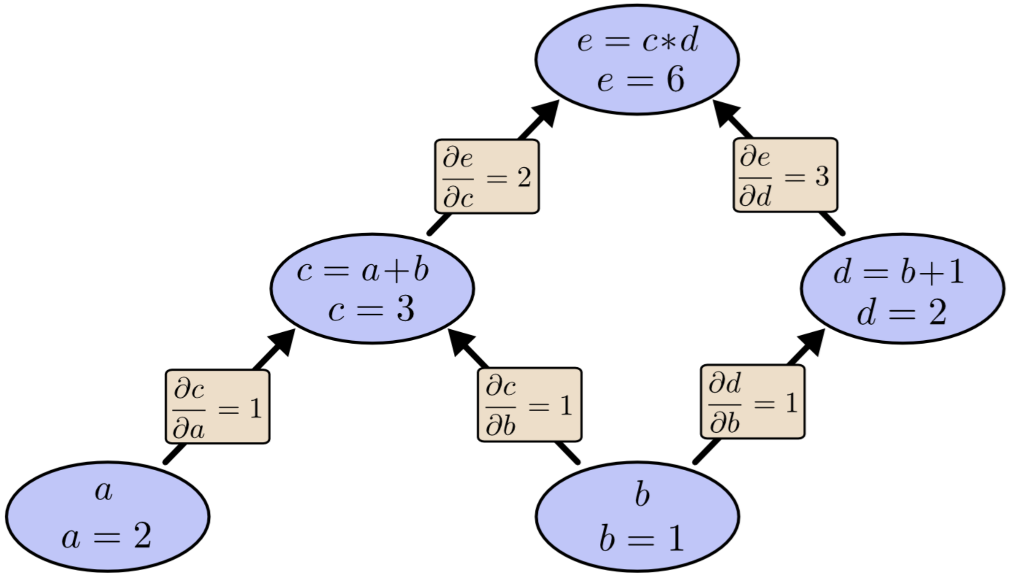
Examples and illustrations from http://colah.github.io/

Task 1: The Chain Rule in Computational Graphs

Blackboard

Derivatives in Computational Graphs

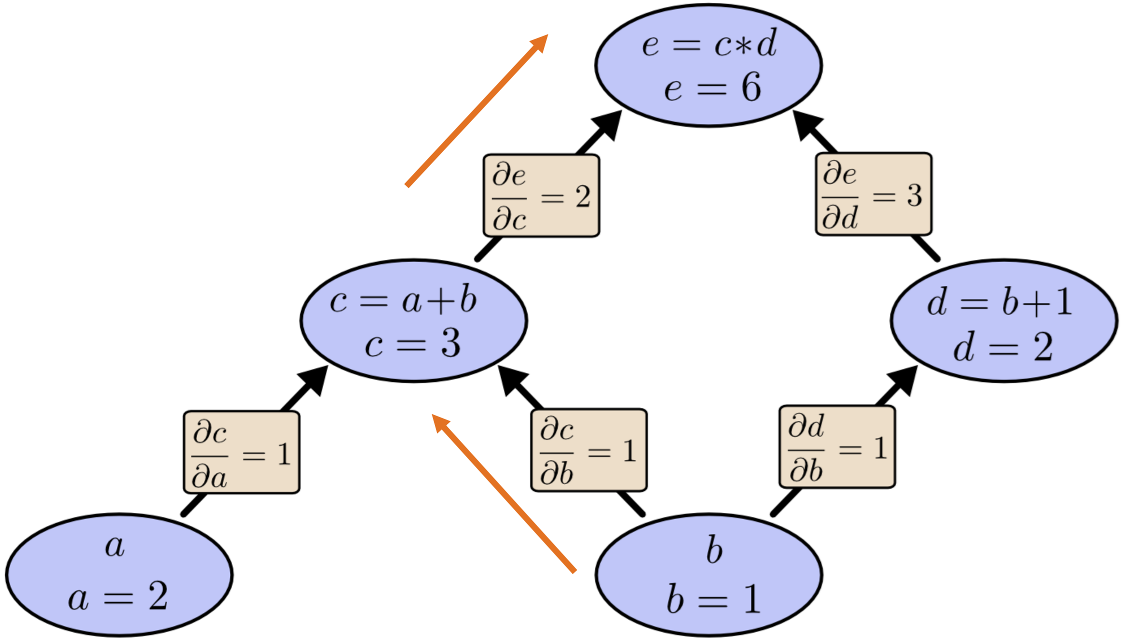
- Computing the gradient in the graphs can be done bottom-up by computing the partial derivatives with respect to the node variables
- The partial derivatives of the complete formula with respect to a variable can then be computed by following all possible paths the leaf node to the root node (follows from the rules for computing multivariate derivatives)



Examples and illustrations from http://colah.github.io/

Forward-Mode Differentiation

$$\frac{\partial e}{\partial b} = 1 * 2 +$$

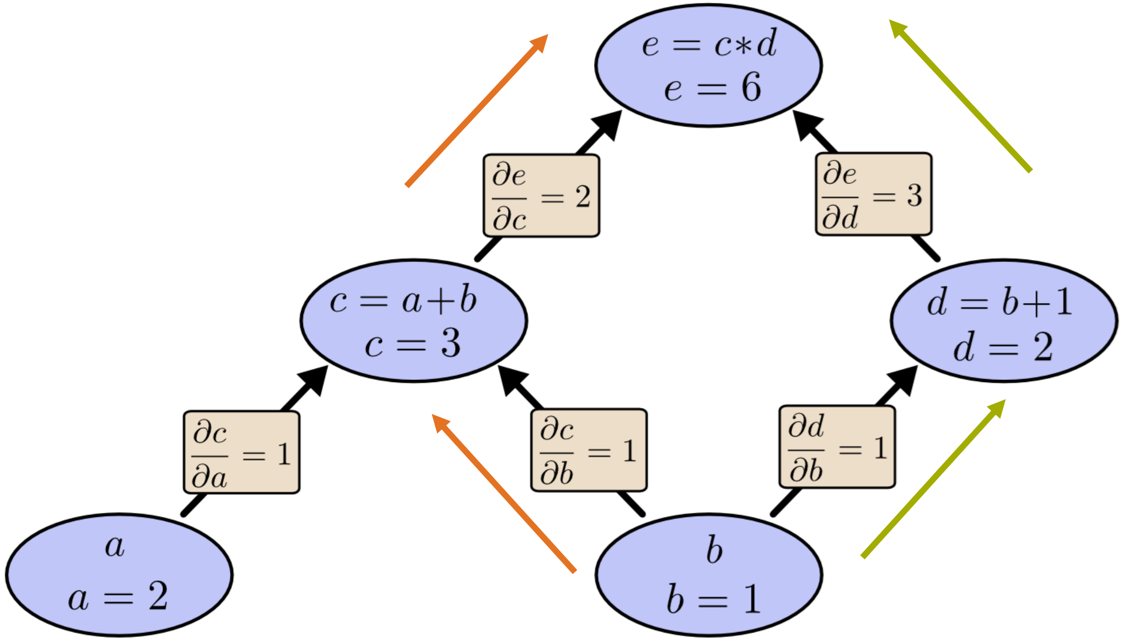


Examples and illustrations from http://colah.github.io/

Forward-Mode Differentiation

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

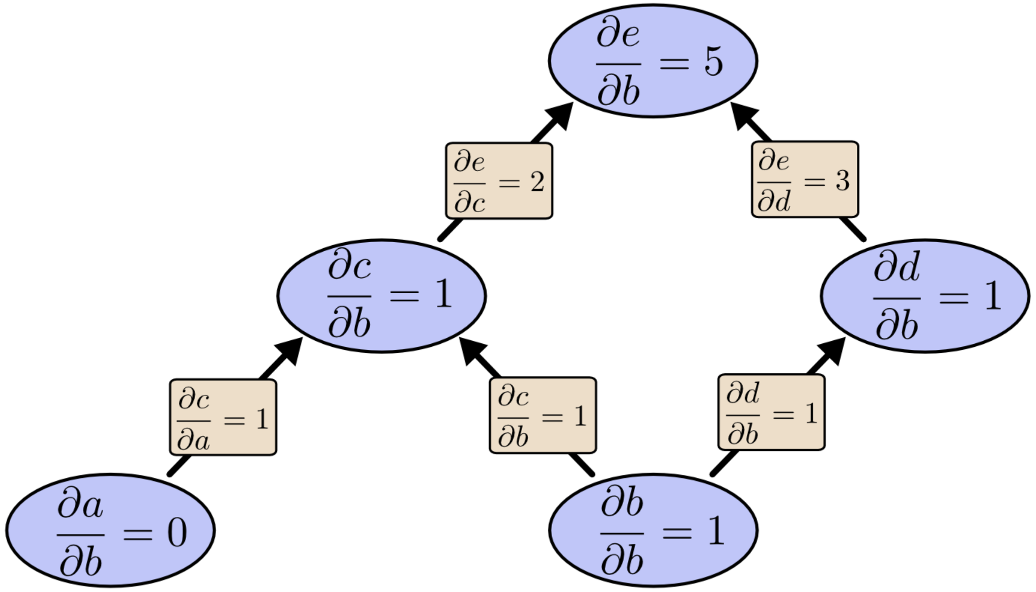
The partial derivatives at the edges indicate the sensitivity of a child not to changes in its parent.



Examples and illustrations from http://colah.github.io/

Forward-Mode Differentiation

- The number of possible paths increases exponentially with the number of layers
- Computational effort can be considerably reduced by storing intermediate results at every node in the graph
- With forward-mode differentiation, a single pass through the graph is sufficient



Examples and illustrations from http://colah.github.io/

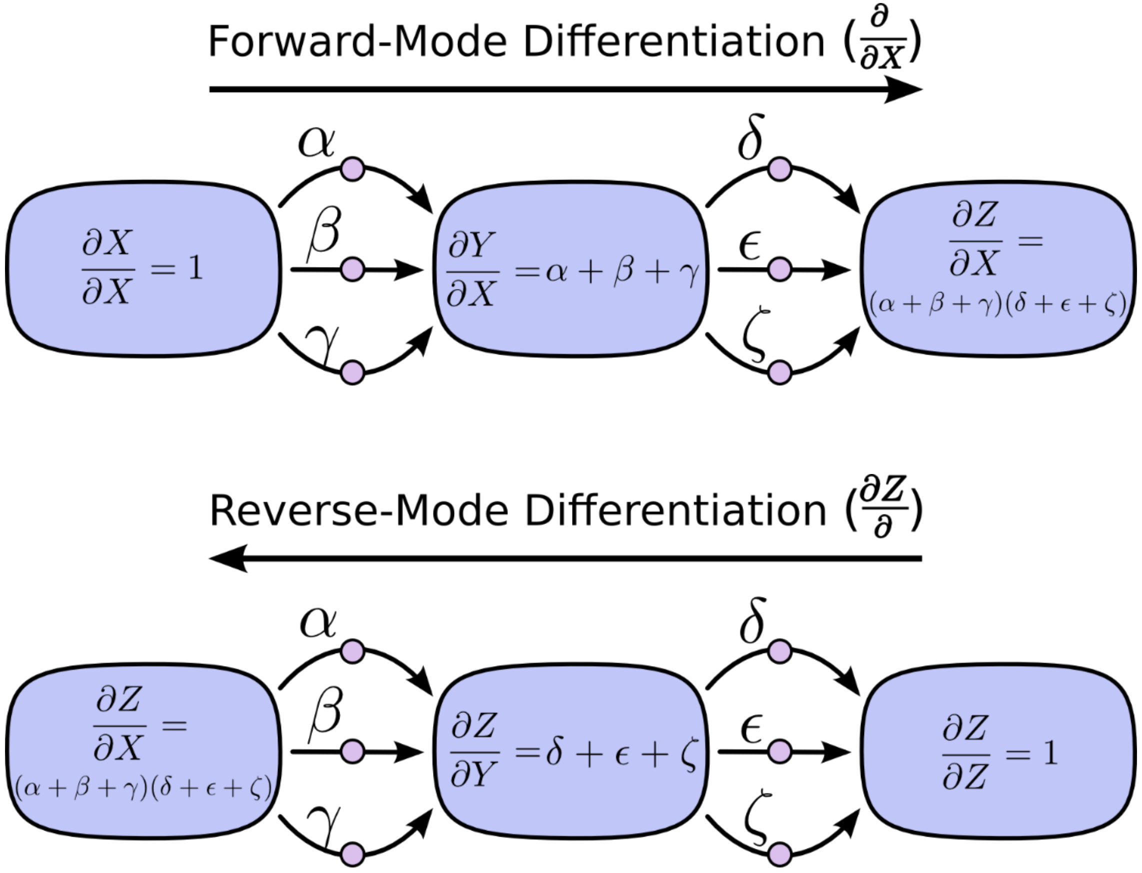
The Backpropagation Algorithm

At every graph node, forward-mode differentiation computes the partial derivative $\frac{\partial}{\partial X}$ with respect to one variable (X is a single synaptic weight).

→ Graph must be computed for possible millions of synaptic weights

Reverse-mode differentiation (also known as backpropagation) computes the partial derivative $\frac{\partial Z}{\partial}$ at every node.

→ After a single pass of the graph, every leaf nodes contains the partial derivative of its variable

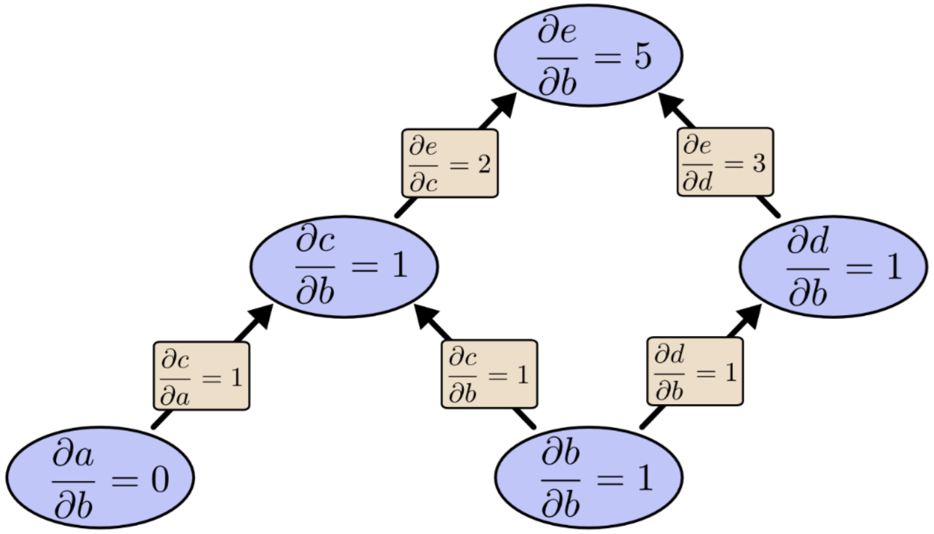


Examples and illustrations from http://colah.github.io/

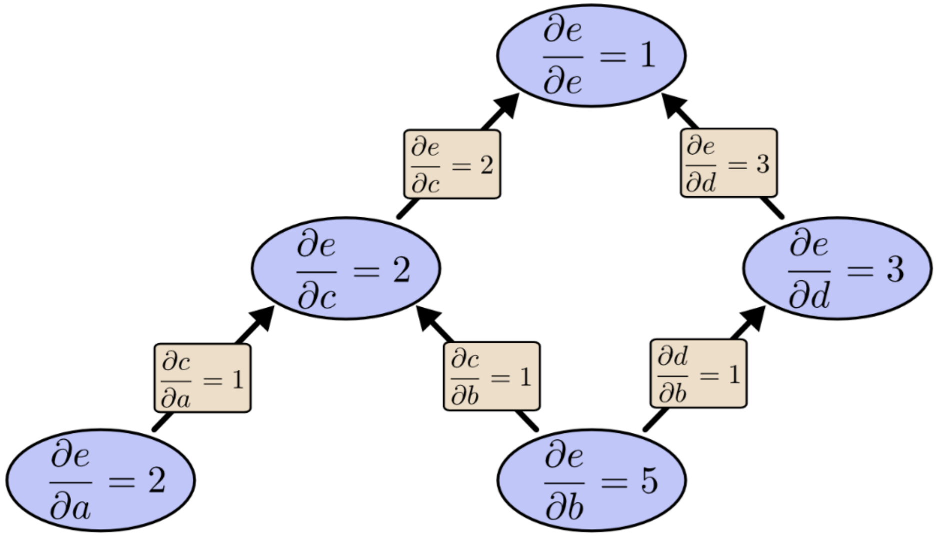
Task 2: Executing the Algorithm

Blackboard

Example



Forward-Mode Differentiation



Backpropagation

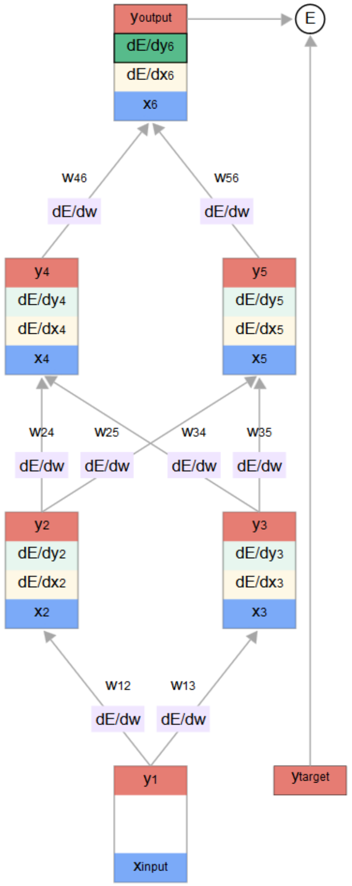
Examples and illustrations from http://colah.github.io/

Demo

Back propagation

Let's begin backpropagating the error derivatives. Since we have the predicted output of this particular input example, we can compute how the error changes with that output. Given our error function $E = \frac{1}{2}(y_{output} - y_{target})^2$ we have:

$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target}$$



https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/

Some Remarks on Backpropagation

- Backpropagation has become the arguably most important method for machine learning in neural networks
- It enables the efficient computation of loss function gradients even in large networks with many parameters
- The actual behavior of the trained network is encoded in the loss function and many loss functions correspond to classic machine learning algorithms
- The representation of neural networks as computational graphs enables the automatic calculation of partial derivatives (→ automatic differentiation)
- Backpropagation is directly available in all modern machine learning frameworks for neural networks

However…

- Backpropagation requires a global error signal and a central execution mechanism for computing weight updates
- Neurons in the brain operate independently – there is no centralized control!
- The backpropagation algorithm in its original form is therefore not biologically plausible
- There is research in computational neuroscience with the goal of mapping the algorithm to a biologically plausible implementation

Neural Network Architectures

Why is there a Need for Application-Specific Neural Network Architectures?

- In general, the universal function approximation theorem guarantees that a neural network with a sufficiently wide single hidden layer can approximate any function
- However, the number of required hidden neurons might grow very large
- Deep neural networks with multiple hidden layers are a first example of how optimized architectures can make neural networks more efficient (less parameters, faster training, better performance etc.)

Optimized Neural Network Architectures

Optimized neural network architectures...

... enable the modeling of prior knowledge about the problem

... impose constraints on how the network represents knowledge

... support specific types of input data (multiple dimensions, sequences etc.)

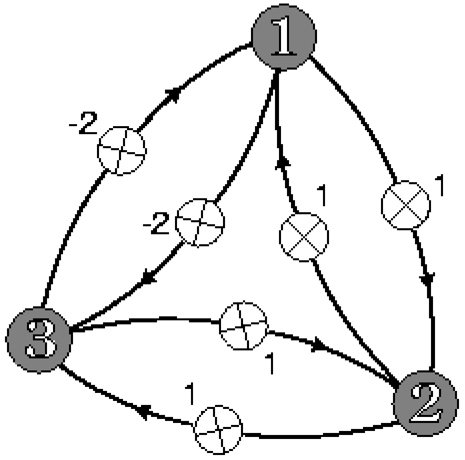
... can reduce the number of network weights that need to be trained

... can speed up training and improve model performance

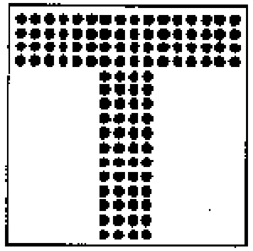
1982: Hopfield Networks

Hopfield net are fully connected neural networks (no autapses, i.e. self-connections) with symmetric weights:

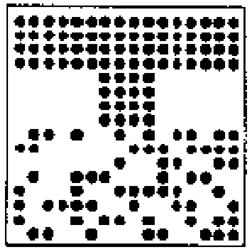
- Every neuron is a Perceptron (but without the Perceptron learning rule)
- Input data is provided through the bias parameters of all neurons
- The resulting network is a dynamical system with attractor states that can be controlled by setting appropriate synaptic weights
- Hopfield nets can be used as associative memories
- Adding stochastic activation dynamics to the neurons results in a new network called Boltzmann Machine



3 node Hopfield net



Original 'T'



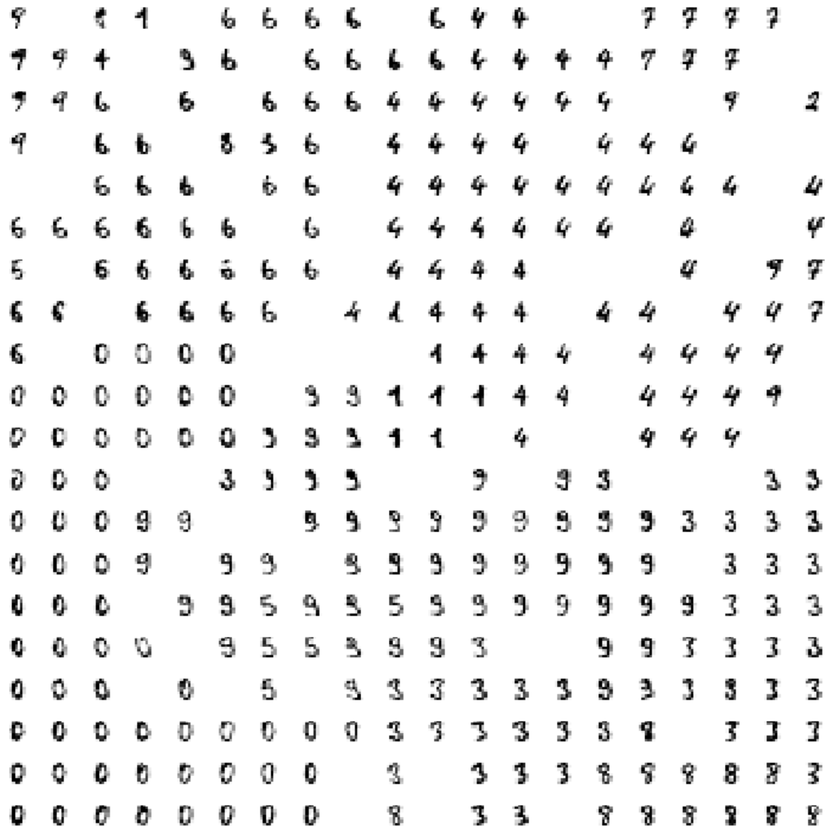
half of image corrupted by noise

From http://web.cs.ucla.edu/~rosen/161/notes/hopfield.html

1982: Kohonen Networks / Self-Organizing Maps

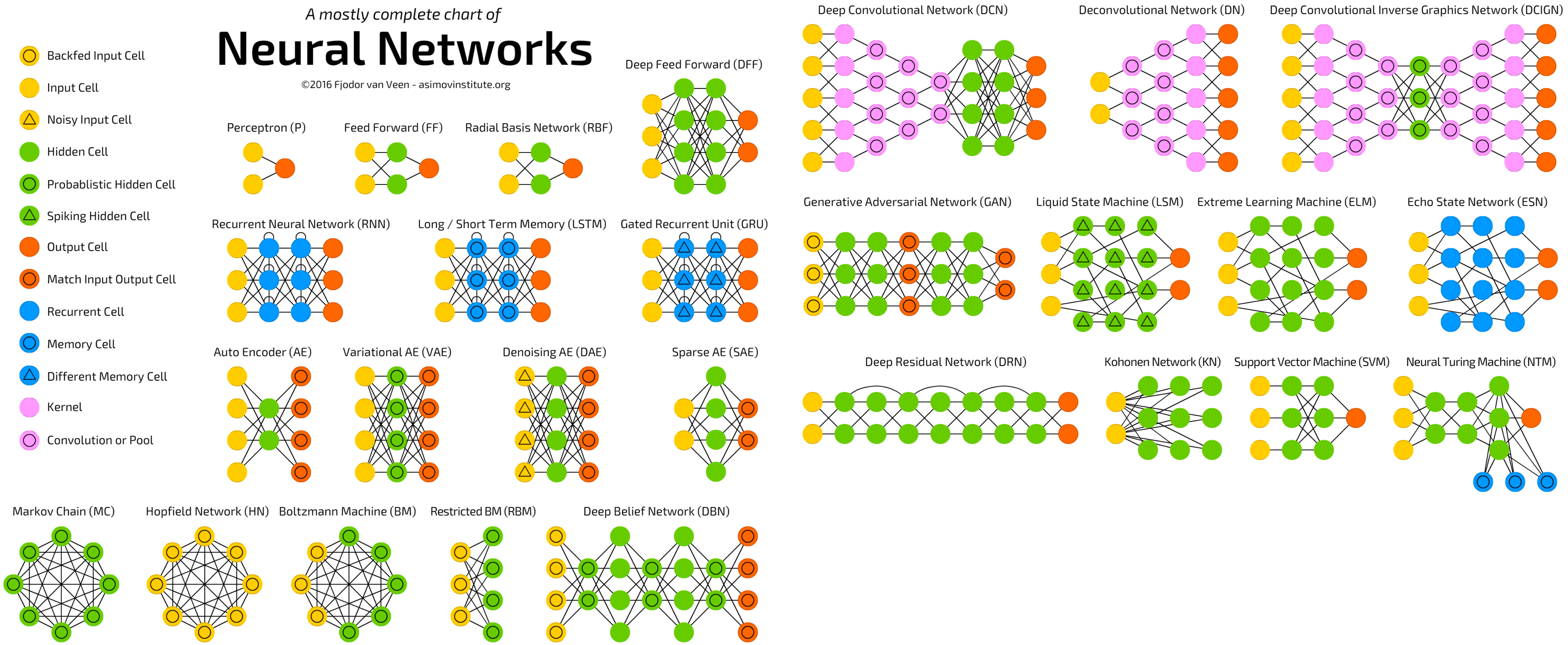
Self-organizing maps are a dimensionality reduction method for the visualization and analysis of complex datasets:

- Neurons are arranged on a grid (typically two-dimensional); there are no synapses
- Every neuron on the grid represents a prototype data element (a model)
- During learning, the model of every neuron is adjusted through a winter-take-all mechanism
- Neighboring neurons represent similar models

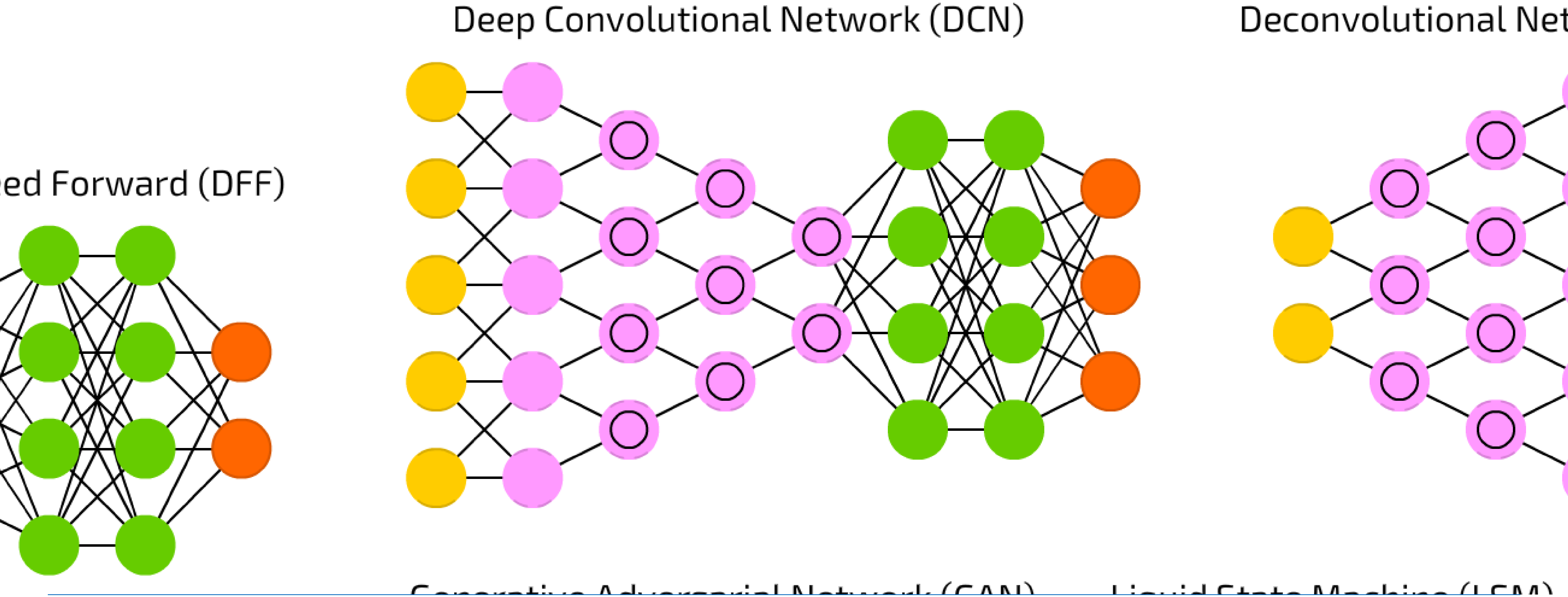


Self-Organizing Map for Handwritten Digits

Today: A Zoo of Neural Network Architectures



In this Session: Convolutional Neural Networks



Convolutional Neural Networks

Recap: Feature Selection and Feature Vectors

- There are many different types of features that can be used to describe the data at hand (shapes, colors, histograms, filters etc.)
- The selection of the right features (feature engineering) is critical for the performance of the machine learning model – the features must contain the information required for predictions
- All selected features are grouped into a feature vector:

$$\longrightarrow \begin{pmatrix} colors \\ edges \\ location \\ \vdots \\ dimensions \end{pmatrix}$$

Image Filters

Natural images contain different types of features …

- Colors
- Edges in multiple directions
- Shapes
- Texture patterns

… and undesired artifacts such as noise and distortions.

Filters extract relevant features and discard undesired artifacts

Convolutions

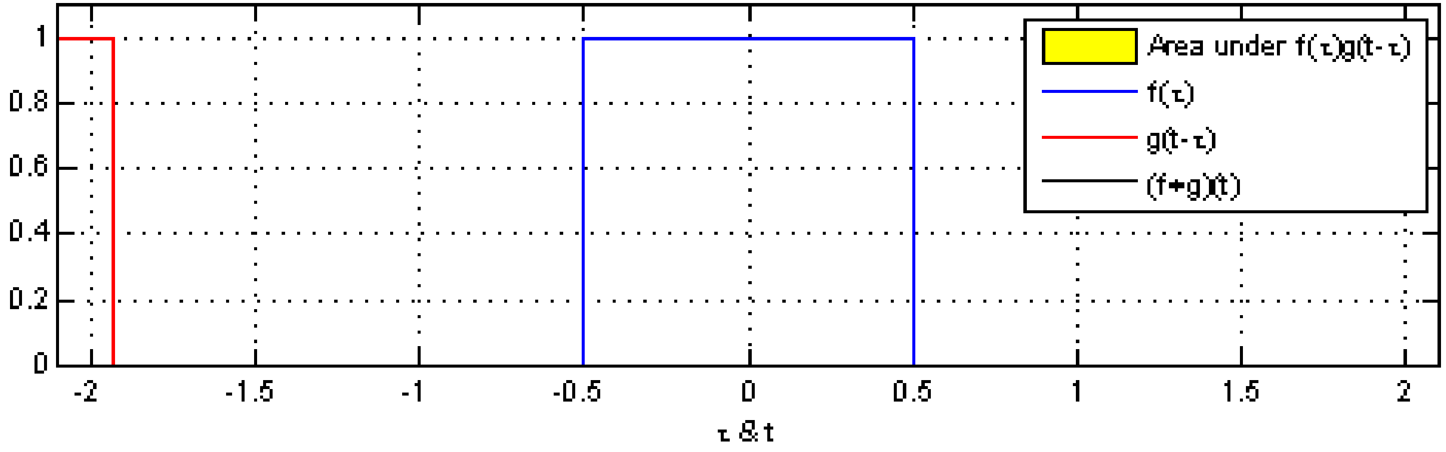
Filters on signals (images, audio, sensor data etc.) are typically implemented as convolutions of a filter f with the signal s:

$$(f * s)(x) = \underbrace{\sum_{t \in T}}_{\text{Sum over all kernel entries}} \underbrace{f(t)}_{\text{Kernel}} \underbrace{s(x - t)}_{\text{Input image pixel shifted to the current position of the kernel}}$$

In continuous systems, the sum operator becomes an integral

$$(f * s)(x) = \int_{\mathbb{R}} f(\tau)s(x - \tau)d\tau$$

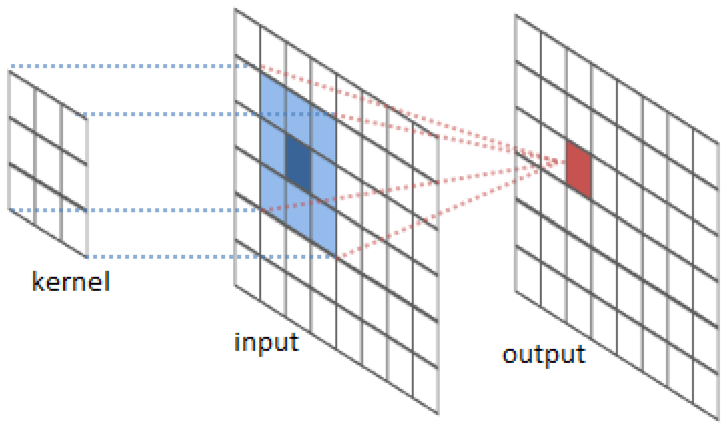
Example

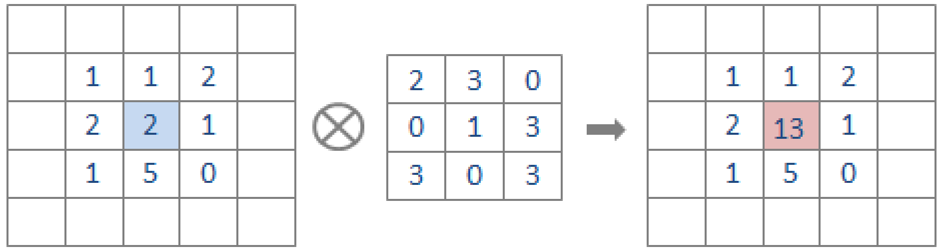


https://en.wikipedia.org/wiki/Convolution

Filtering Images with Kernels

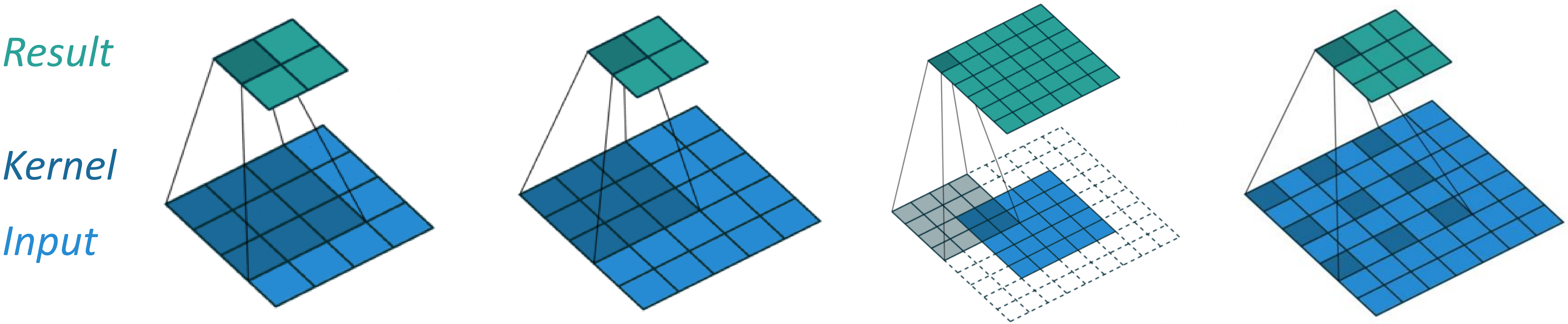
- A kernel is a matrix that defines a signal filter
- The entries of the kernel determine how the influence of neighboring data points (e.g. pixels) of a sample on each other
- The filtered data sample is computed by sliding the kernel along the input sample (e.g. an image) and computing new data points by adding up neighboring data points (pixels) with the weighting defined by the kernel





From http://intellabs.github.io/RiverTrail/tutorial/

Example



No Padding
No Strides

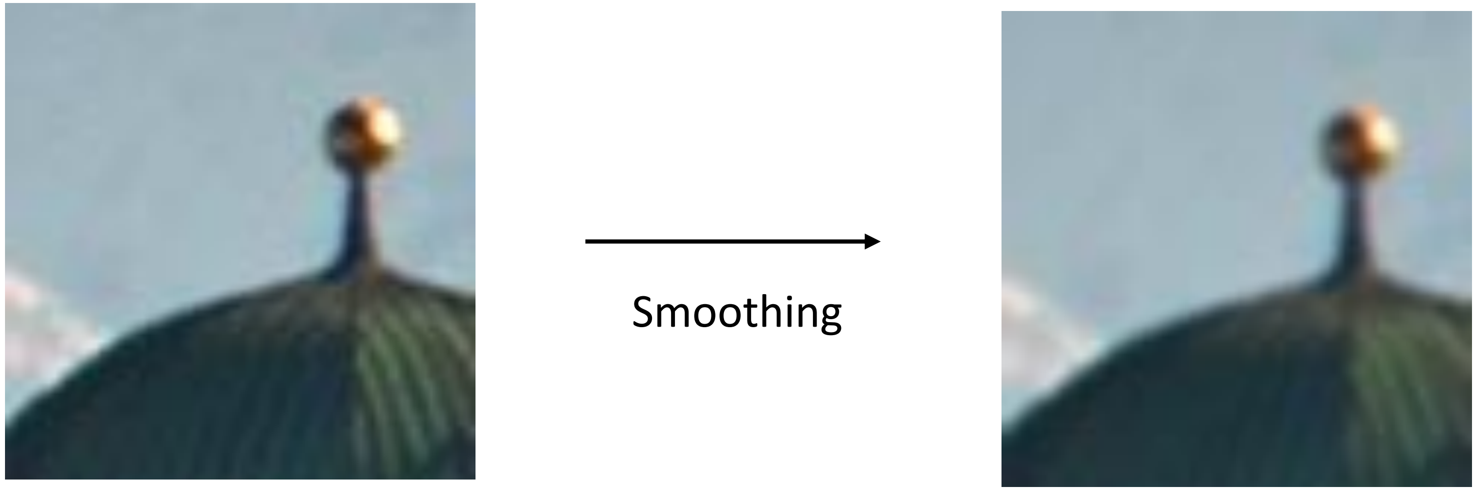
No Padding
Strides

Padding
No Strides

No Padding
Strides
Dilated Kernel

https://github.com/vdumoulin/conv_arithmetic

Task 3: Implement a Basic Smoothing Filter

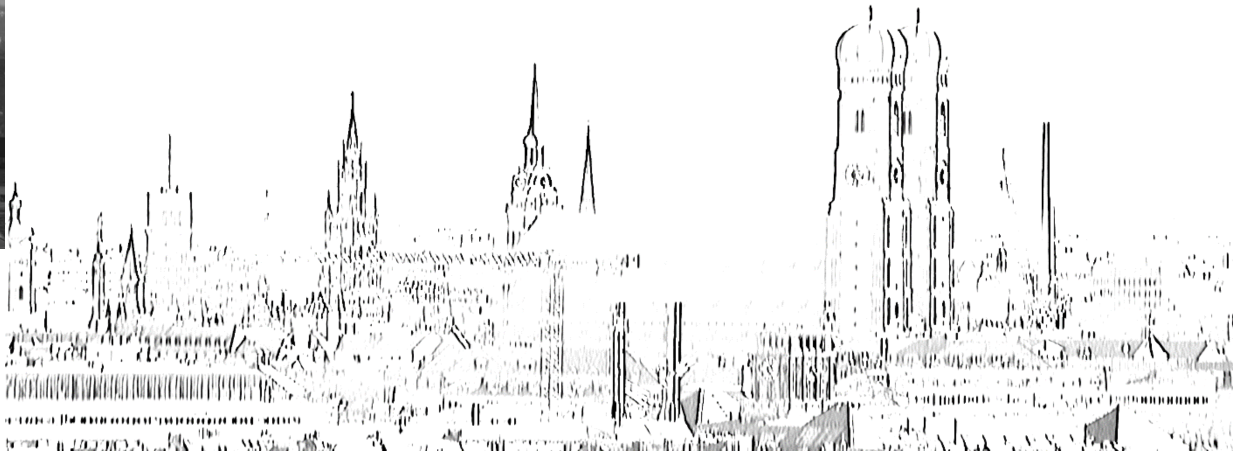


Task 4: Implement a Simple Edge Detector

- Explain your choice of the filter matrix
- How do changes along the axes of the kernel matrix change the result?

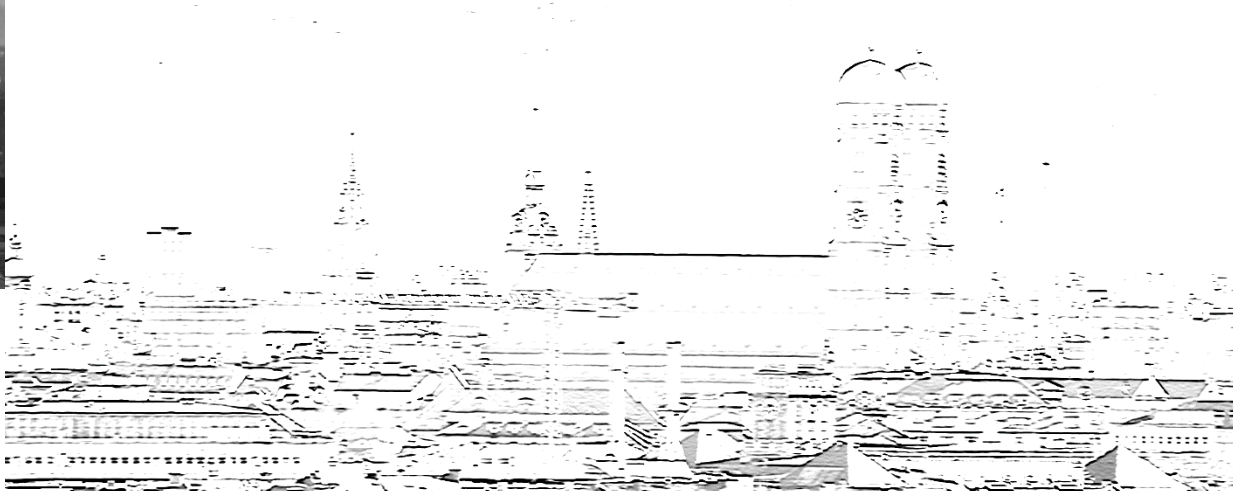
Edge Detection

Texture Analysis with Gabor Filters



The filter was adjusted to identify vertical edges

Texture Analysis with Gabor Filters



The filter was adjusted to identify horizontal edges

Typical Layout of a Convolutional Neural Network

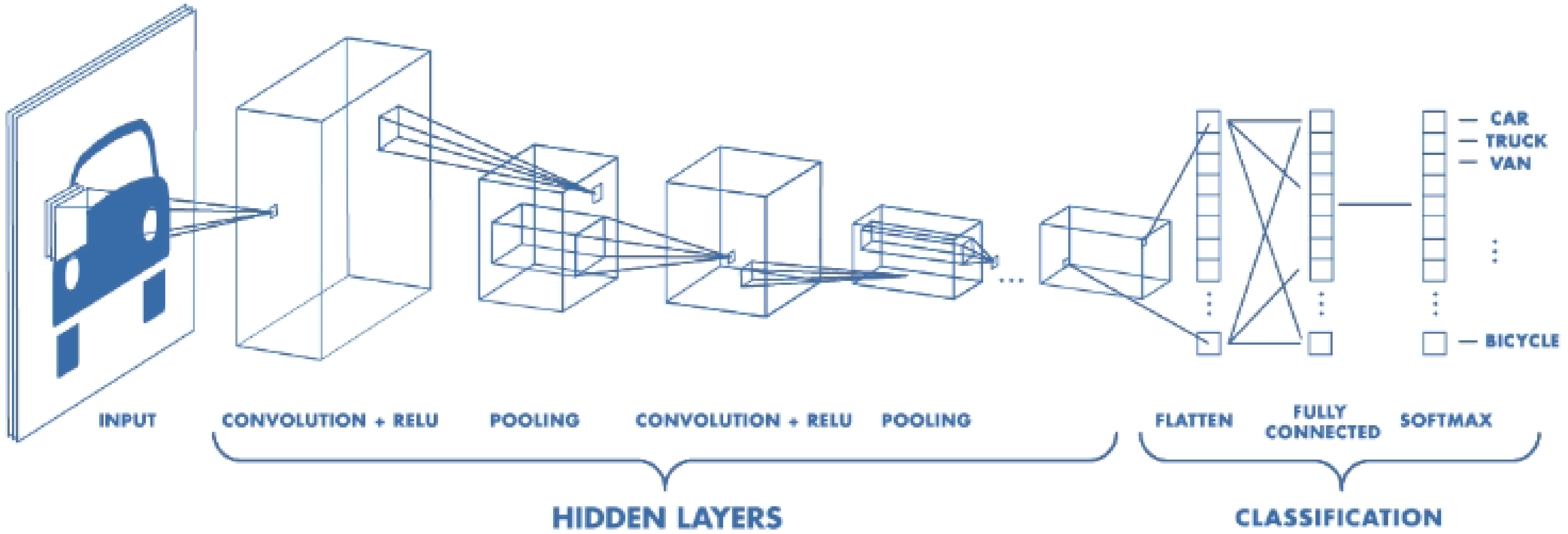


Image from https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html

Task 4: Counting Network Parameters

Consider an input image with a size of 256x256. What is the size of the output image after applying a convolution of size 3x3 with a stride width of 2 and padding 0?