HARDWARE ACCELERATORS FOR MACHINE LEARNING
***
PROGRAMMING FRAMEWORKS FOR MACHINE LEARNING

Christoph Kessler
IDA, Linköping University

LINKÖPINGS UNIVERSITET

Guest lecture 3 feb. 2020 in LiU course
"Hardware for Machine Learning"

Contents

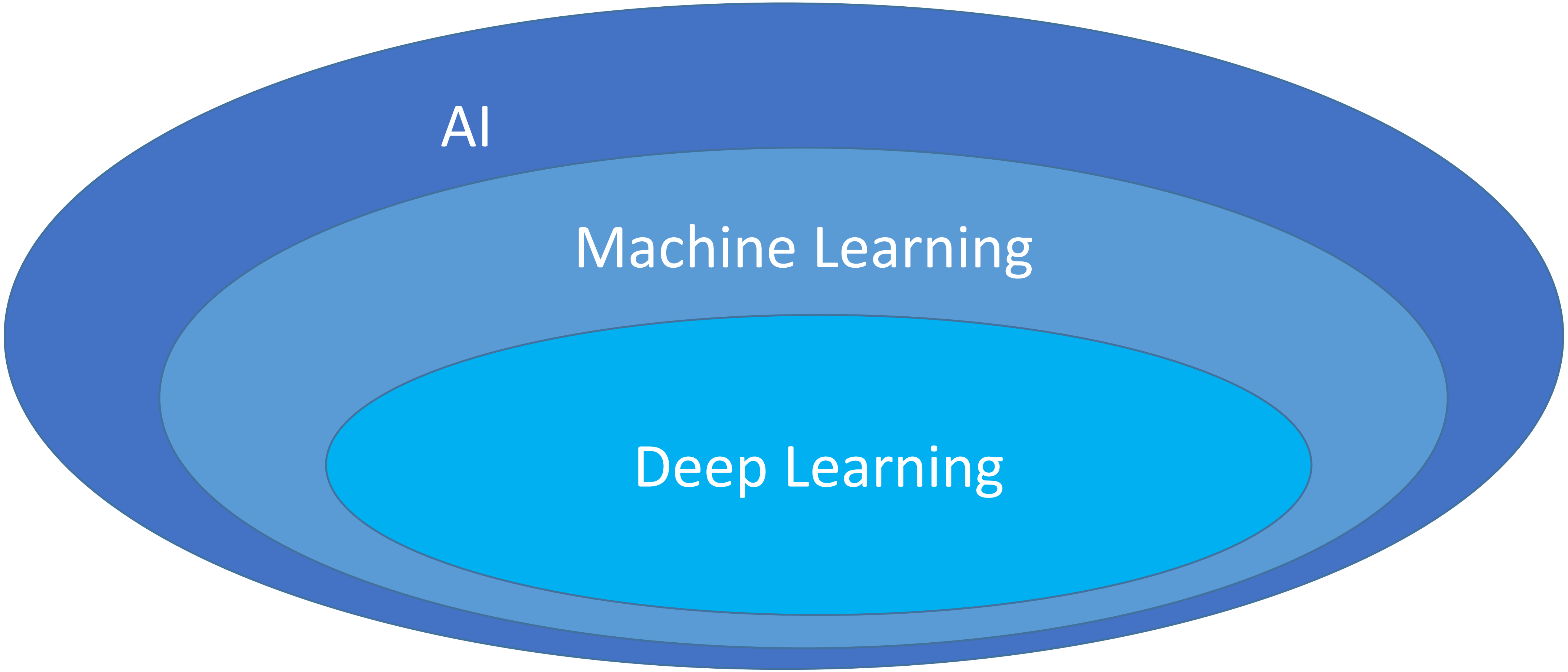
1. Motivation and short overview of ANN and Deep Learning

2. Hardware Platforms for Acceleration of Deep Learning

3. Overview of programming frameworks for Deep Learning
   - TensorFlow
   - Keras
   - …

AI/ML/DL

(much simplified...)

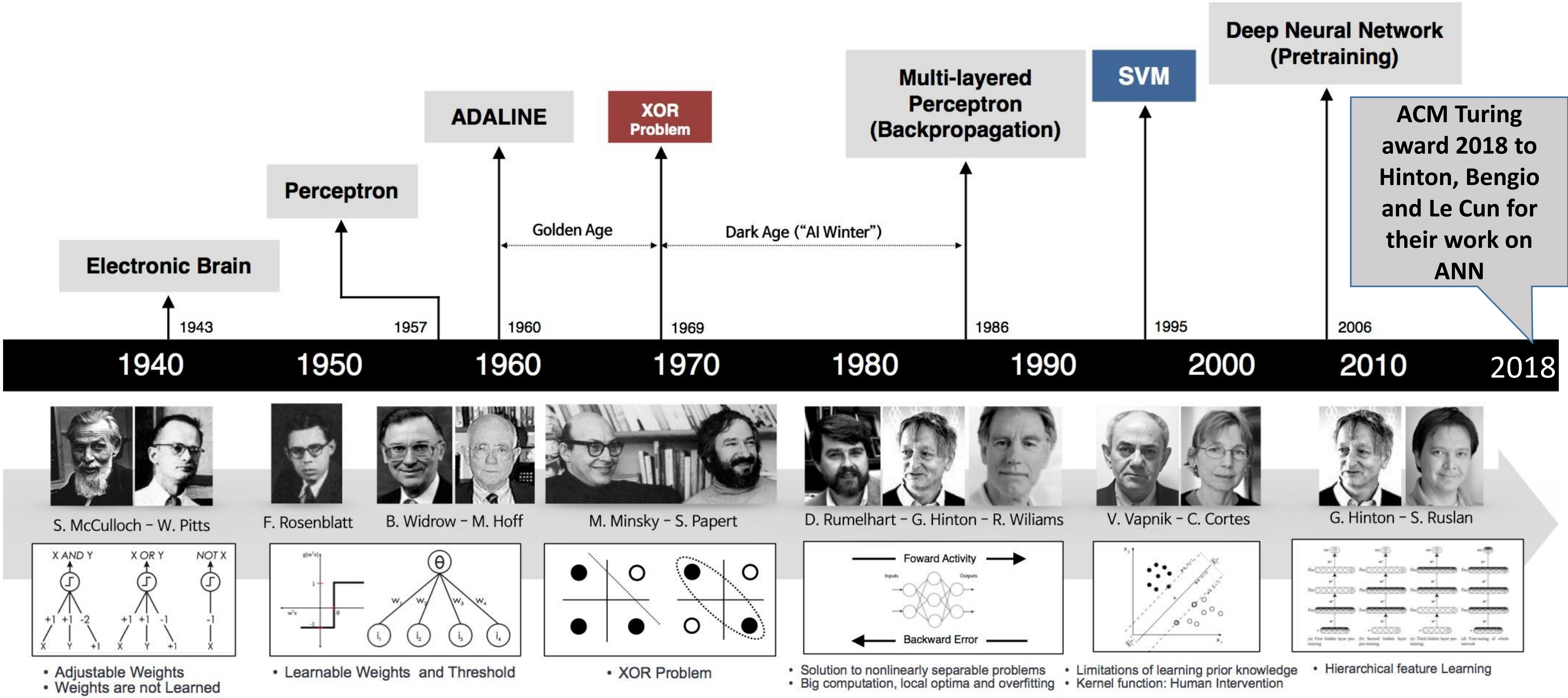


Machine Learning – A Definition

"[Machine] *learning* is the process of [automatically] constructing, from training data, a fast and/or compact surrogate function that *heuristically* solves a decision, prediction or classification problem for which only expensive or no *algorithmic* solutions are known. It automatically abstracts from sample data to a total decision function."

- [Danylenko, Kessler, Löwe, "Comparing Machine Learning Approaches...", *Software Composition* (SC'2011), LNCS 6708]

Major Milestones in Neural Networks and ML

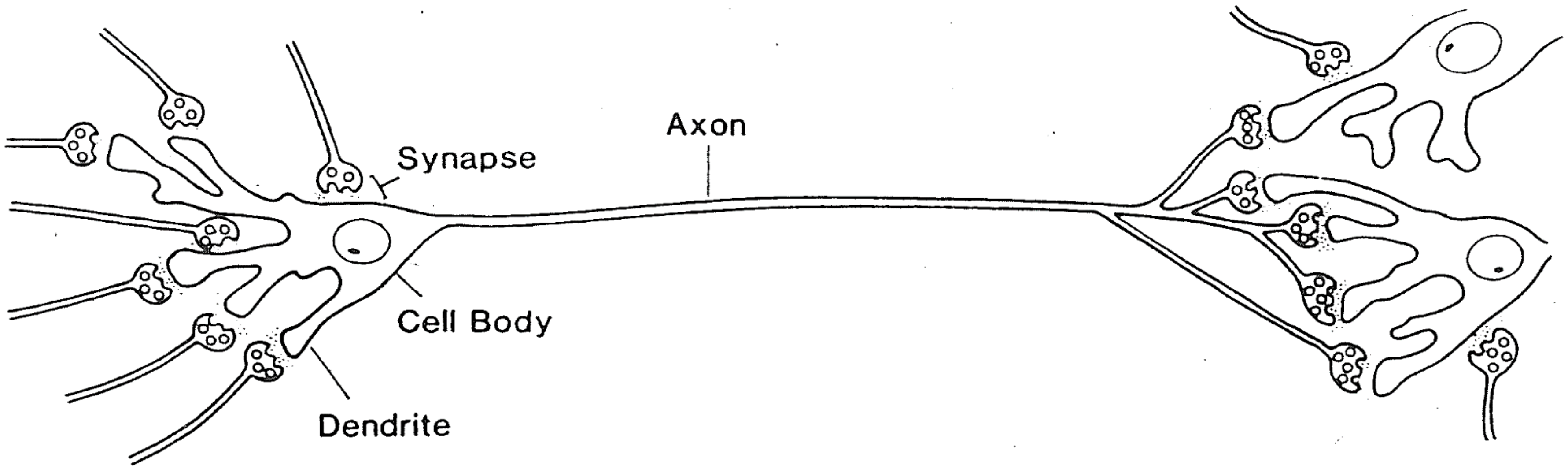


Source: https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

Idea (old!): Artificial Neural Networks

- Understand structure and functionality of the human brain
  → Biology / neurology, since ca. 1900
- Develop a simplified mathematical model,
  an artificial neural network (ANN)
  → Mathematics / CS, since 1943
- Simulate the model on a digital computer → CS
- Identify (commercial) application areas, e.g. → since ca. 1985
  - Pattern recognition, classification
  - Function approximation
  - Optimization, planning
  - Prediction
  - Content-addressable (associative) memory
  - Brain-Machine coupling, prothese control, …

Biological Neural Networks

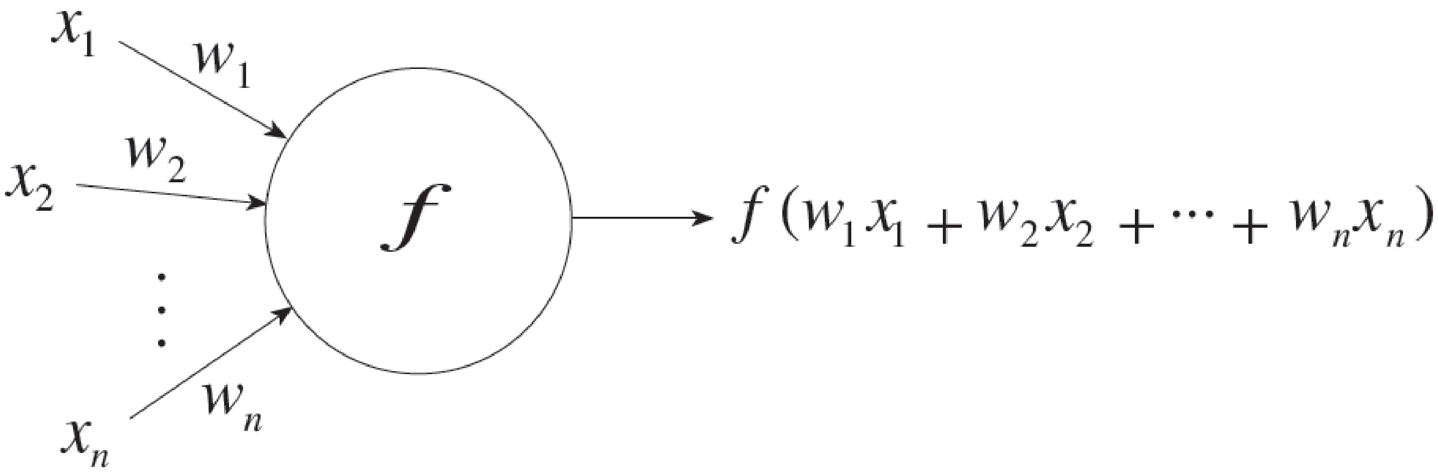


Neuron (neural cell, ganglion)

- main building block of the neural system
  - Human neural system has ca. 2.5 10^{10} neurons
- **Soma / cell body** (cell membrane, cytoplasma, cell core, …)
- **Axon**: connection to other neurons or other (eg. muscle) cells
- **Dendrites**: tree-shaped connection of synapses to soma
- **Synapse**: contact point to (axons of) other neurons to take up neural (electrical) signals. Human brain: ca. 10^5 synapses per neuron

Generic Model of a Neuron

- McCulloch and Pitts 1943:



where *f* calculates function

$$y = \theta\,\left(\Sigma_{j=1,\dots,n}\, w_j x_j - u \right)$$

with $\theta\,(h) = 1$ if $h > 0$, and
0 otherwise

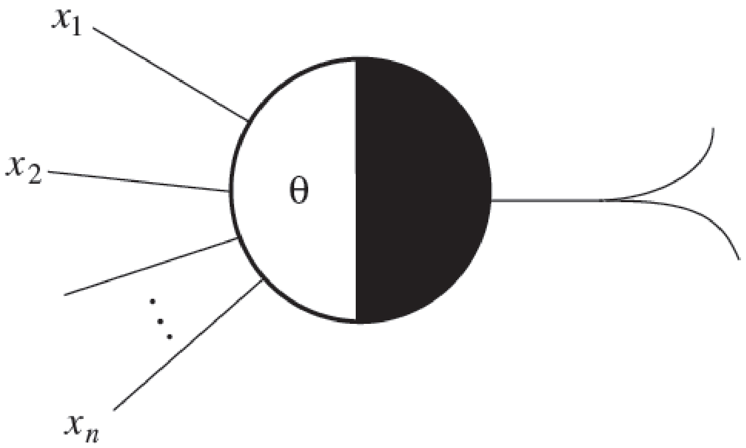
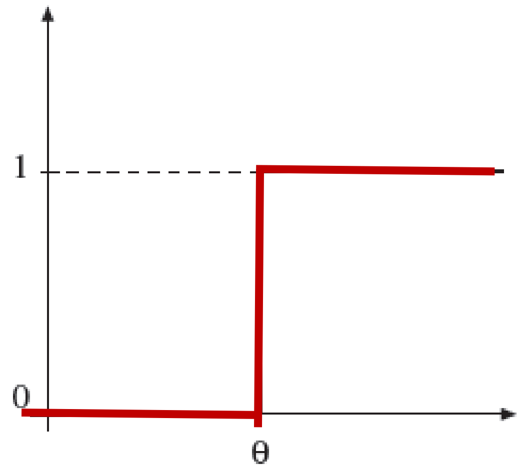


Fig. 2.6. Diagram of a McCulloch–Pitts unit

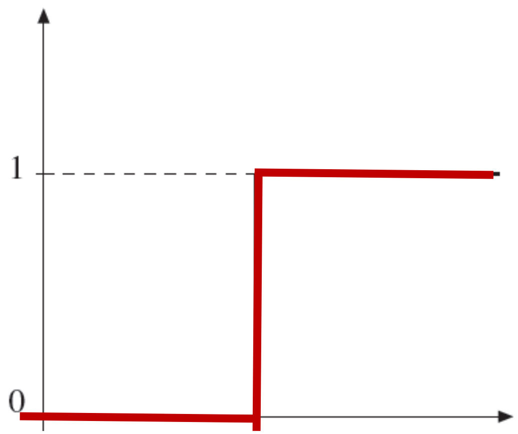
Remarks



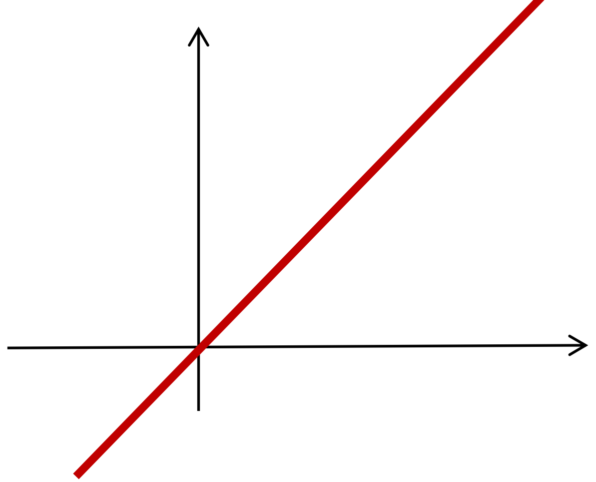
- θ is called the **activation function**
  - Step function is the most common one
  - All input signals x_i and the output signal y are then binary.
- Threshold value u can be integrated into the summation:
  - Set $x_0 = 1$ (constant)
  - Set $w_0 = -u$
  - Then $y = \theta\left(\sum_{j=0,\ldots,n} w_j x_j \right)$
- For now, no switching time delay assumed

Alternative Activation Functions

- By now: Step function
  → output signal is binary

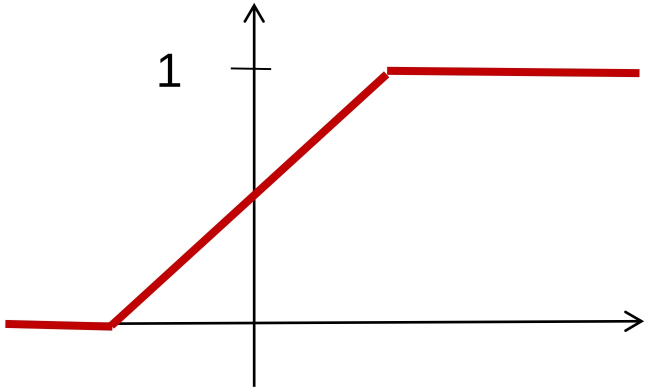


Identity = no activation function
(OK for regression)

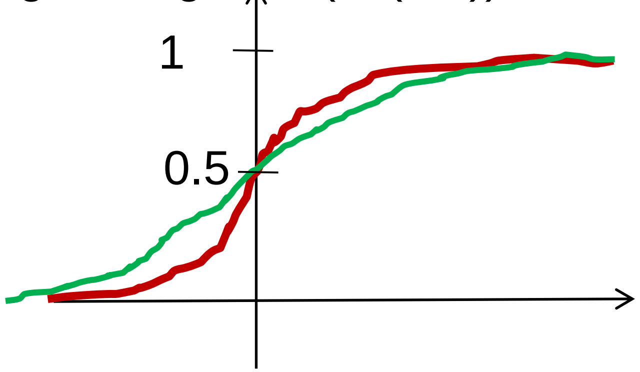


- Some alternatives:

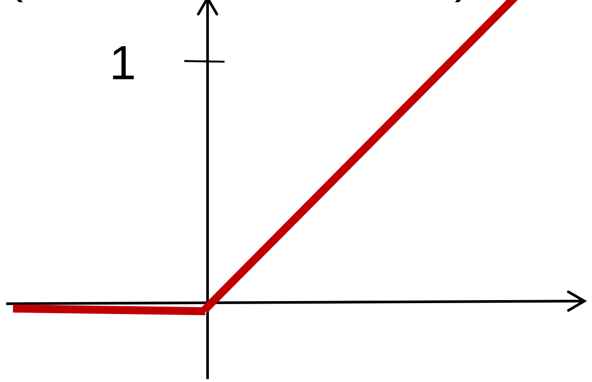
Piecewise linear function



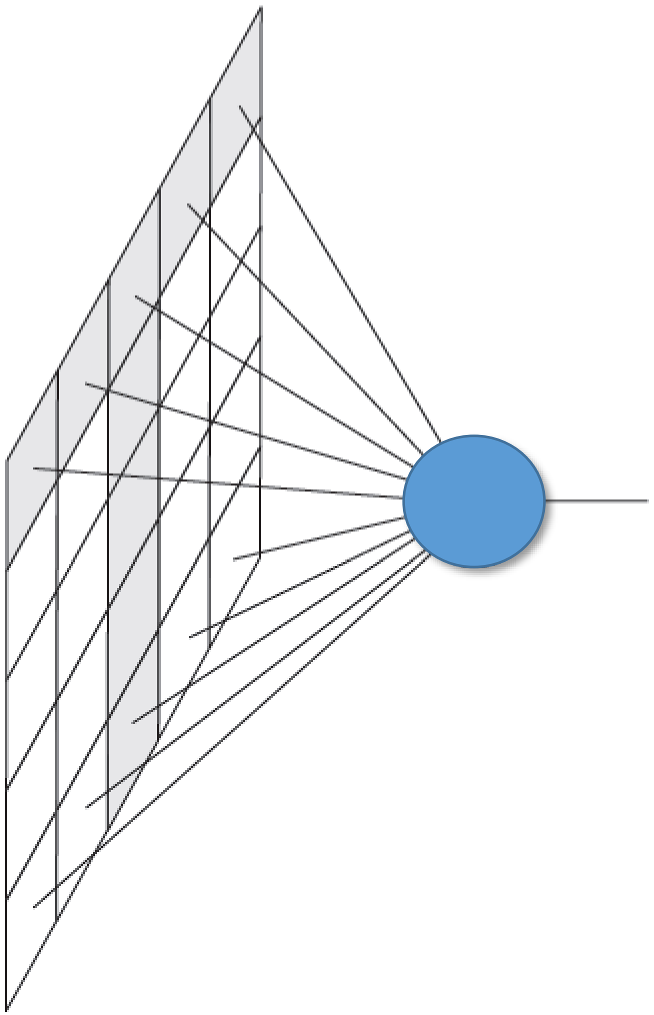
Sigmoid function e.g. tanh (in (-1,1)),
logistic sigmoid (in (0,1))



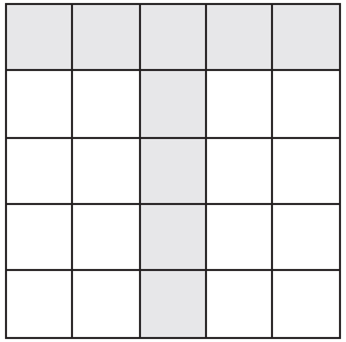
ReLU(x) = max(0, x)
(Rectified Linear Unit)



Feature detection by Perceptron



pattern



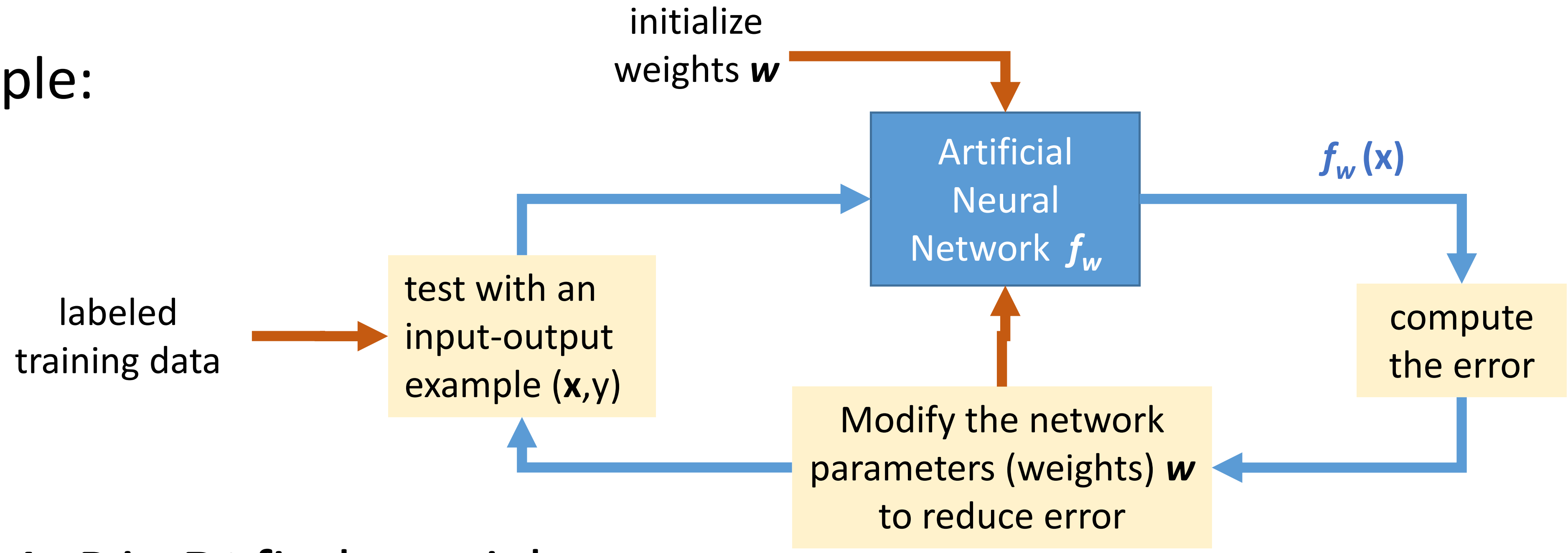
weights

1	1	1	1	1
–1	–1	1	–1	–1
–1	–1	1	–1	–1
–1	–1	1	–1	–1
–1	–1	1	–1	–1

. Feature detector for the pattern T

Learning Algorithms for Perceptron

- General principle:



- For given sets A, B in $\mathbf{R}^n$ find a weight vector $\boldsymbol{w}$ such that the perceptron computes a function $f_{\boldsymbol{w}}(\boldsymbol{x}) \sim 1$ if $\boldsymbol{x}$ in A, and 0 if $\boldsymbol{x}$ in B (classification)
- **Error (loss) function** = # wrong classifications for a given $\boldsymbol{w}$

$$E(\boldsymbol{w}) = \sum_{\boldsymbol{x} \text{ in } A} (1 - f_{\boldsymbol{w}}(\boldsymbol{x})) + \sum_{\boldsymbol{x} \text{ in } B} f_{\boldsymbol{w}}(\boldsymbol{x}) \quad >= 0$$

"Zero-One Loss"

- **Learning = Minimizing the error function**

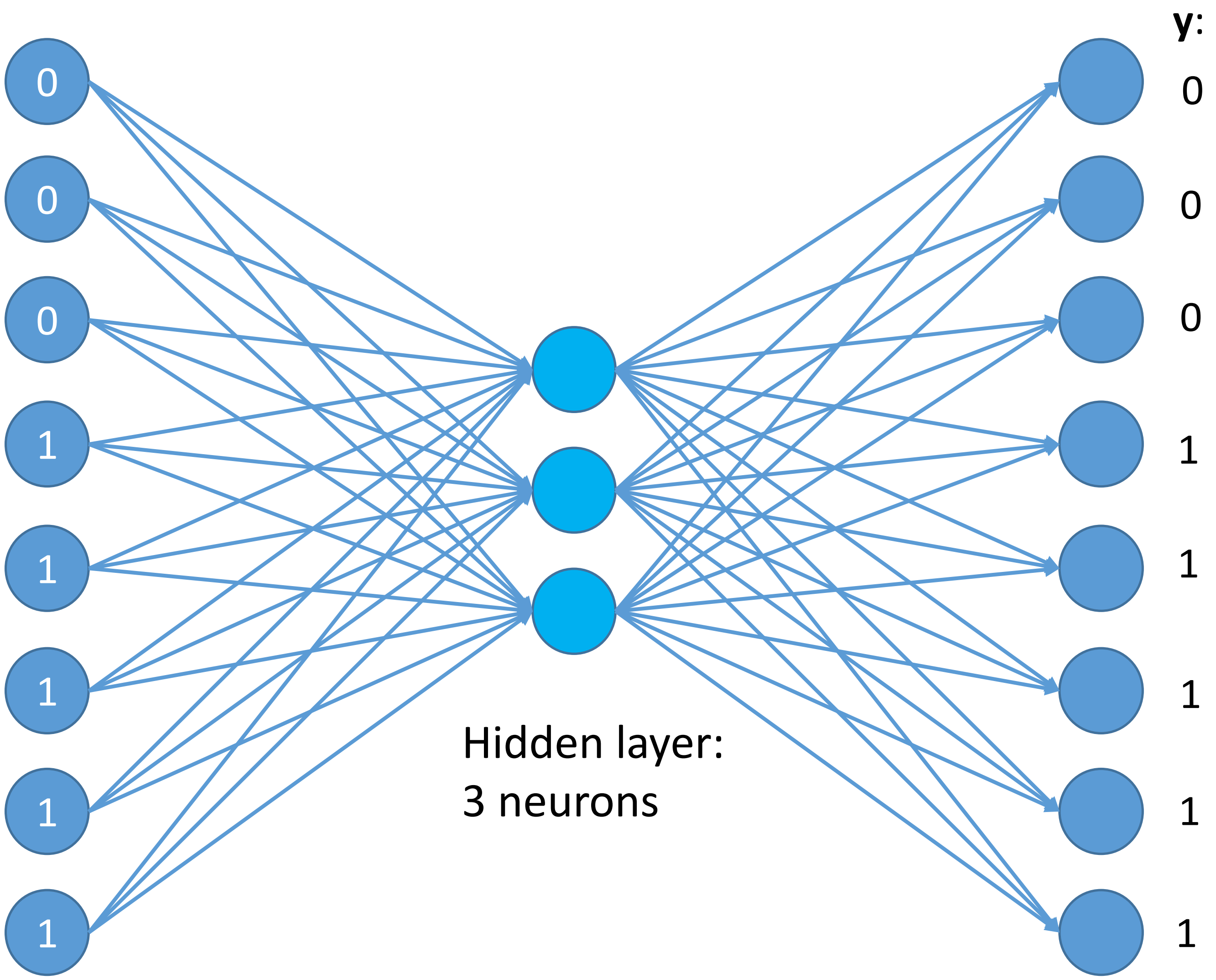
Error (Loss) Functions

(Image removed)

Source: H. Huttonen: “Deep Neural Networks: A Signal Processing Perspective”. In S. Bhattacharyya et al.: *Handbook of Signal Processing*, Third Edition, Springer, 2019.

Towards Deep Learning:
Example: 8-3-8 Auto-Encoder Problem

Input layer:
8 neurons,
getting unary
inputs **x** from
(00000000,
00000001,
00000011,
00000111,
...
11111111)

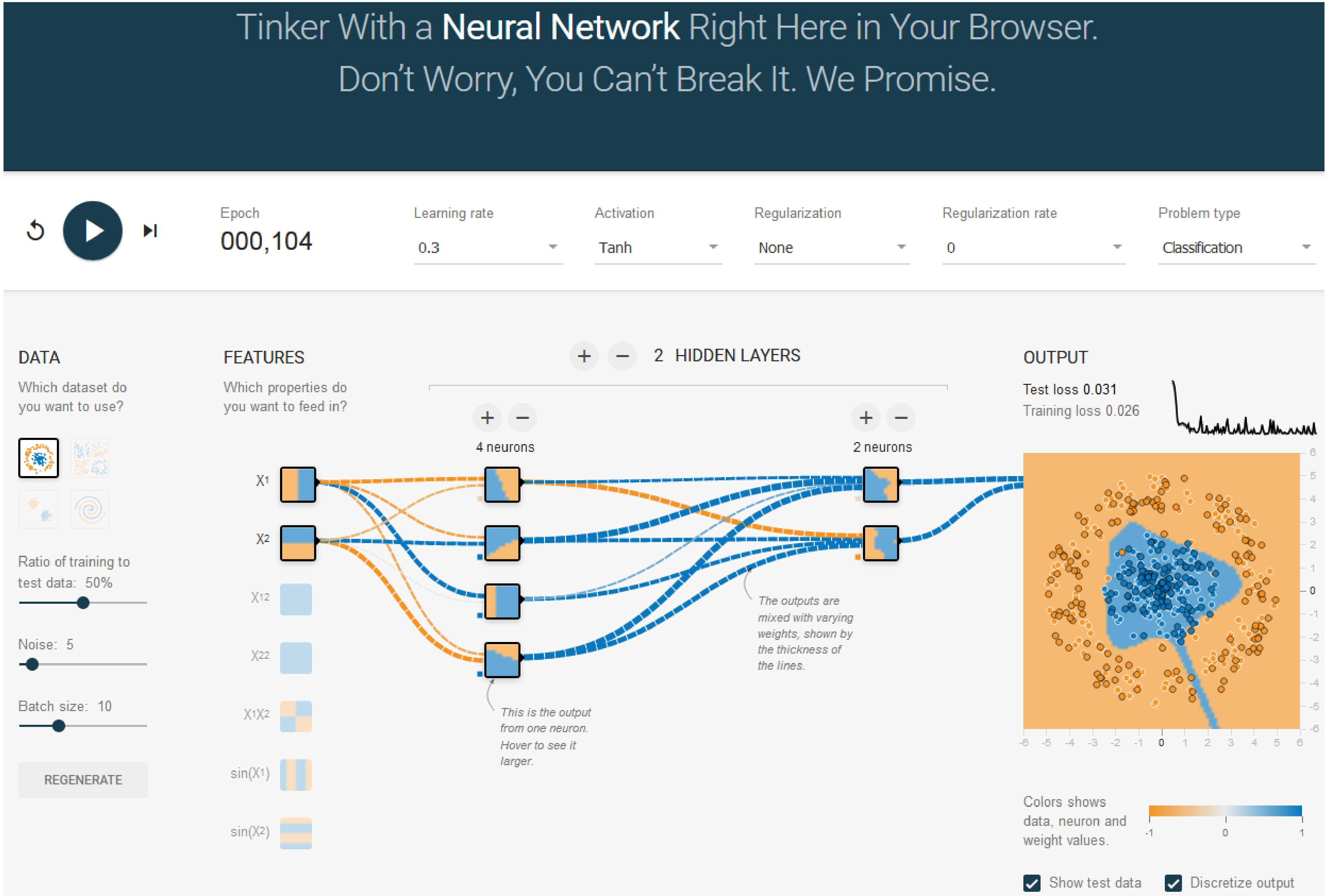


With BP algorithm:
Hidden layer neurons learn (some permutation of) binary encoding of unary input

Output layer:
8 neurons,
desired output y = input x

TensorFlow Playground

http://playground.tensorflow.org



Each successive layer in a neural network uses features from the previous layer to learn more complex features.

(Image removed)

Convolutional Neural Networks (CNN)

A class of *deep*, feed-forward artificial neural networks

- most commonly applied to analyzing images
- use a variation of multilayer perceptrons designed to require minimal preprocessing.
- include **convolution layers** (implementing filters over each pixel and *nearest* neighbors (→sparsely locally connected) in the predecessor layer resp. input image)
  - producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input
- combined with **pooling layers** (sampling/reduction for coarsening the resolution to next layer)
- and with **ReLU** layers (thresholding) and **fully-connected** layers and more ...

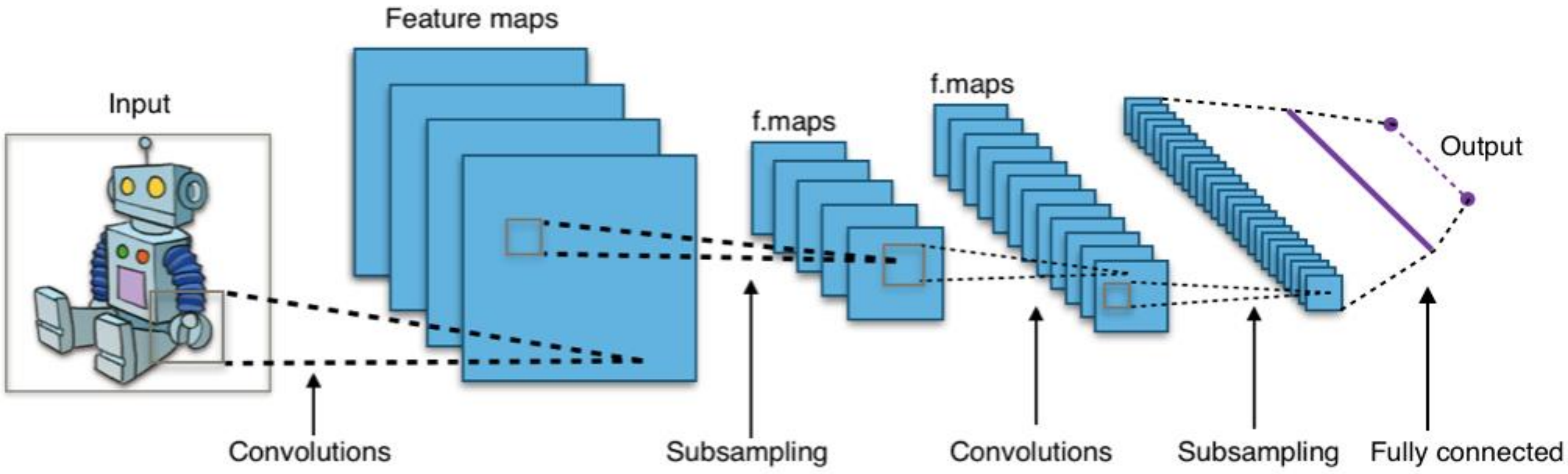
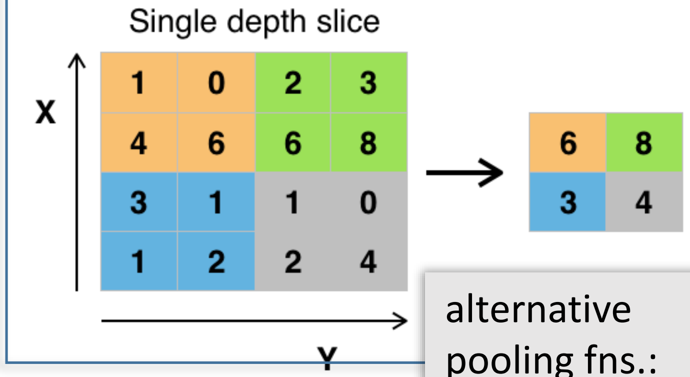


Image source: Wikipedia, By Aphex34 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45679374

Pooling

(Ex.: Max-pooling, 2x2, stride 2):



alternative pooling fns.: average, L2, ...

Example: AlexNet

Convolutional layer 5: Output matrix has dimensionality (Nx13x13) x (128), where N is the batch size

(Image removed)

Image source: Nvidia, Jetson^{TM} TX1 White Paper, https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf

The Resurgence of Deep Learning since ~2010

- Deep Learning (based on deep/convolutional neural networks) is a *subset* of Machine Learning using Artificial Neural Networks
- Excellent recognition accuracy for deep/convolutional neural networks
  - Automatic feature extraction
  - More self-organizing and robust against translation/rotation/scaling – Less dependent on proper manual image preprocessing (engineering effort)

- Everything was basically there since the 1980s, except for the "computability of DNNs". Then, DL boosted by **3 enabling factors**:

1. Public availability of versatile datasets like MNIST, CIFAR, and ImageNet
2. Widespread popularity of accelerators e.g. GPUs – training can be done offline
3. Sensors and cameras everywhere → new applications
   - Automated image classification needed for important commercial applications, such as assisted / autonomous driving, video surveillance, X-ray diagnostics, ...
   - And countless other application areas
     - Some might be ethically questionable

- Much hype ...

(Open) Labeled Datasets

Examples:

- MNIST (handwritten digits) http://yann.lecun.com/exdb/mnist/
- CIFAR10 https://www.cs.toronto.edu/~kriz/cifar.html →
- ImageNet https://www.image-net.org
- Street View House Numbers (SVHN)
  http://ufldl.stanford.edu/housenumbers/
- Several others...

Note: Most commercial datasets are *not* open
(this is the real IP of a DL-based product, not the ML methods/code)

Example: CIFAR-10

https://www.cs.toronto.edu/~kriz/cifar.html

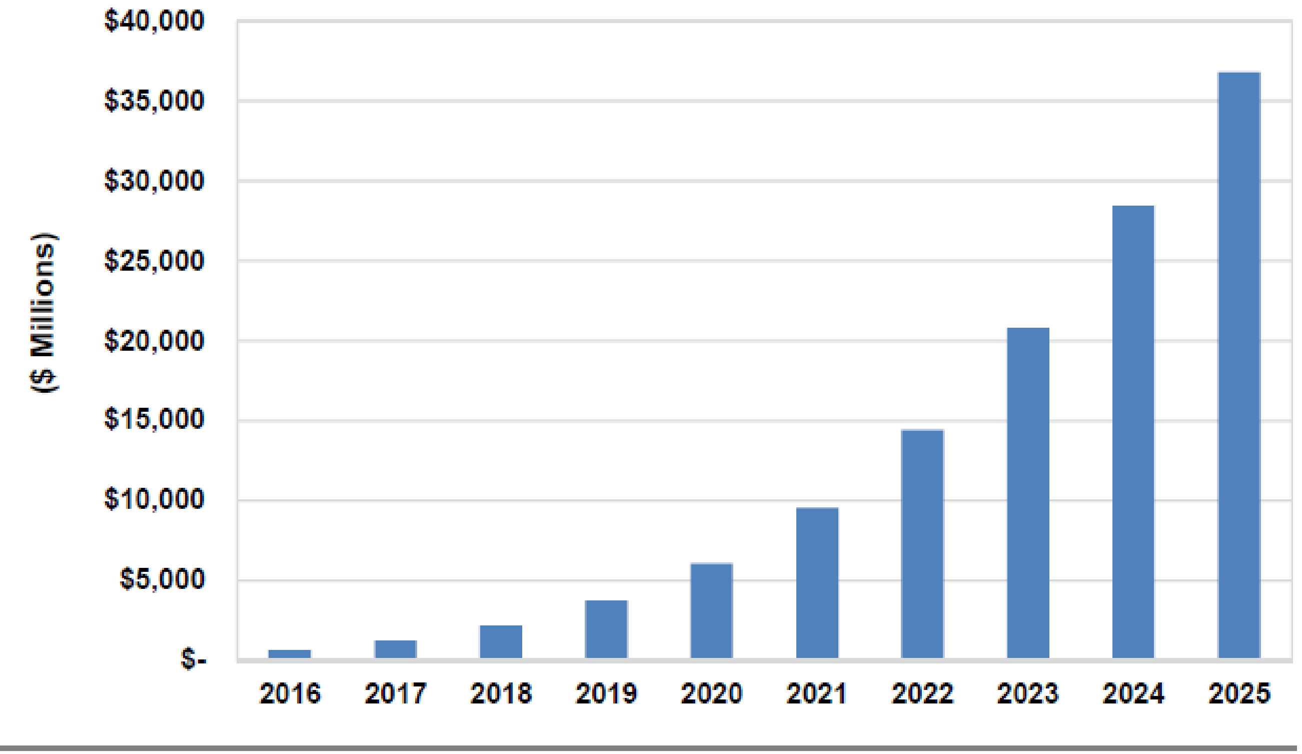
CIFAR-10 dataset

- 60000 32x32 colour images in 10 classes →
  - 6000 images per class
- 50000 training images and 10000 test images.

(image removed)

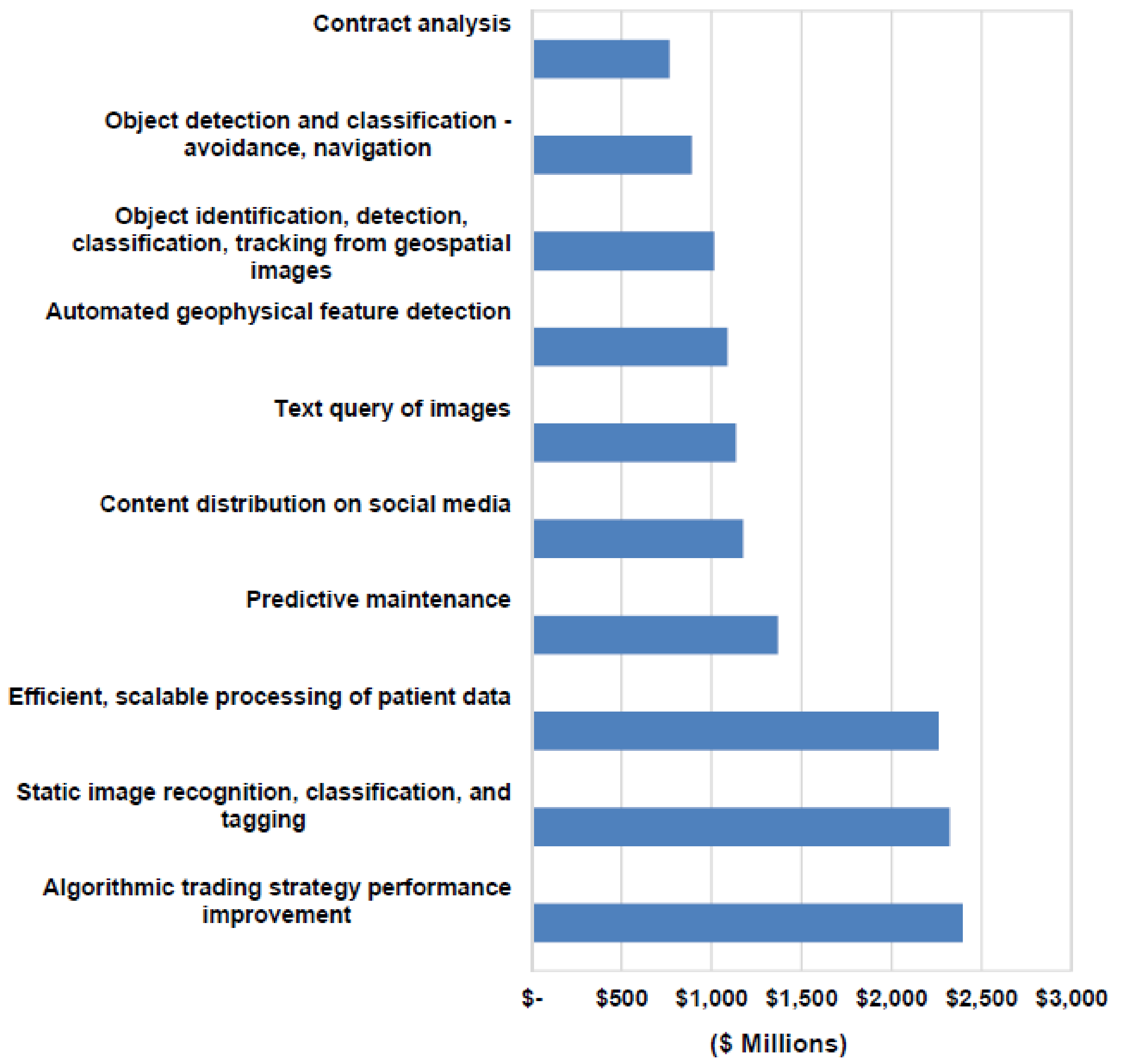
AI/ML Market Prognosis

t 1.1 *Artificial Intelligence Revenue, World Markets: 2016-2025*



(Source: Tractica)

Chart 1.2 *Artificial Intelligence Revenue, Top 10 Use Cases, World Markets: 2025*



(Source: Tractica)

Source: https://www.top500.org/news/market-for-artificial-intelligence-projected-to-hit-36-billion-by-2025/

Applications of Deep Learning

- Vision
  - Image Classification
  - Object Recognition
  - Style Transfer
  - Caption Generation
- Speech
  - Speech Recognition
  - Real-time Translation
- Text
  - Sequence Recognition and Generation
  - Machine Translation
- Medtech
  - Disease discovery
  - Cancer Detection
- Assisted / Autonomous Driving
  - Combination of multiple areas
    like Image/Object Detection and classification, Text Recognition, etc.
- ...

Example: Cancer Detection

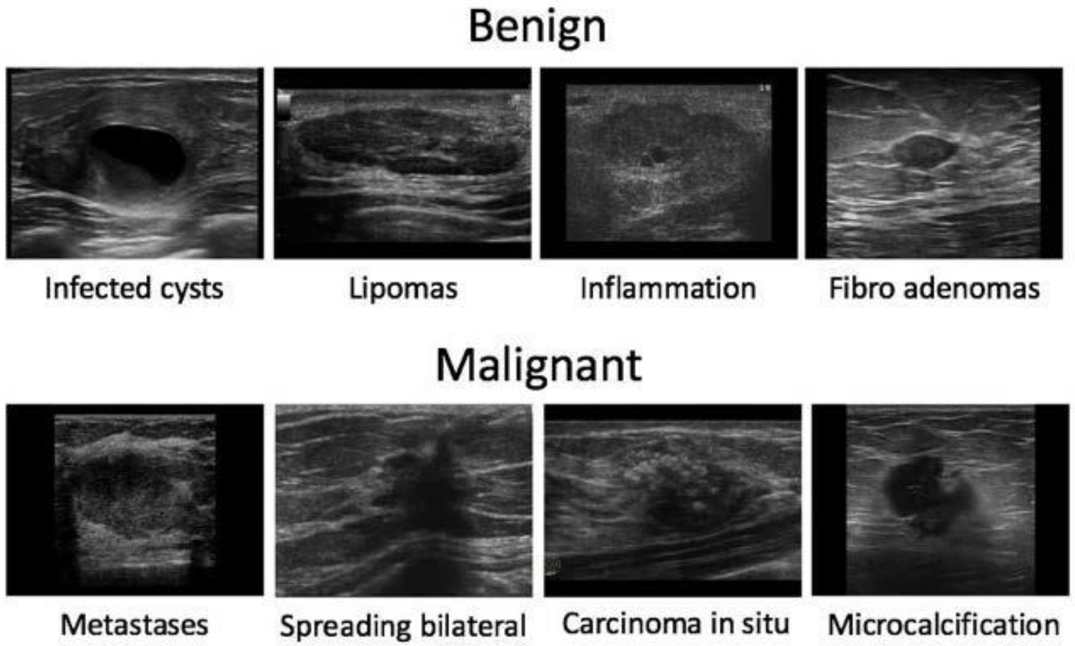
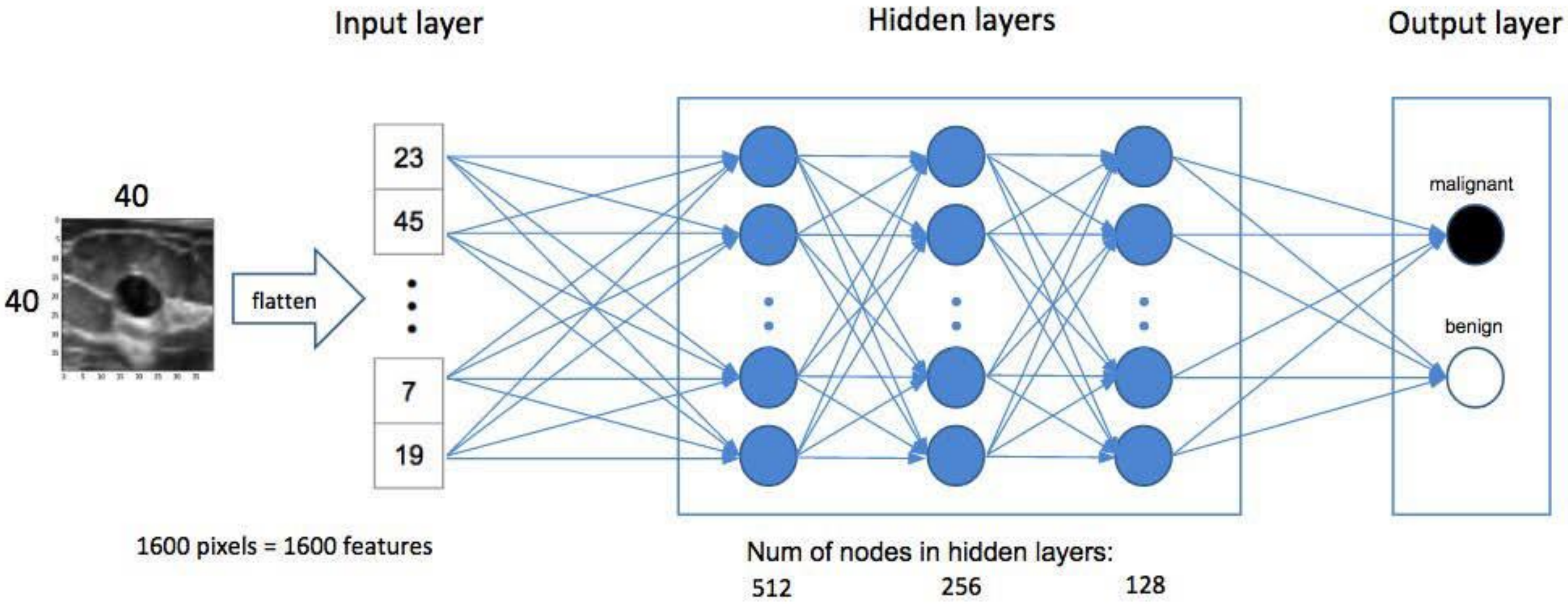


Image source: https://blog.insightdatascience.com/automating-breast-cancer-detection-with-deep-learning-d8b49da17950



Training Data Labeling and Augmentation

- **Where do we get labeled training data** for new problems?
  - Examples: Frame drivable area, bridges, motorcycles, humans on the road, traffic lights, car plates, ...
  - Usually need **human** labelers
    - expensive – this training data is the real IP of the companies, not the software
    - crowdsourcing in some cases, e.g. Oxford cats-and-dogs dataset [Parkhi et al. 2012] → →

(image removed)

Training Data Labeling and Augmentation (cont.)

- **Where do we get labeled training data** for new problems?
  - Examples: Frame drivable area, bridges, motorcycles, humans on the road, traffic lights, car plates, ...
  - Usually need **human** labelers
    - expensive – this training data is the real IP of the companies, not the software
    - crowdsourcing in some cases, e.g. Oxford cats-and-dogs dataset [Parkhi et al. 2012]

- Risk with large DNNs and (too) few labeled training images: **Overfitting**
  - Overfitting = the DNN just memorizes the training set
    but it does not do a good job in generalizing classifications for previously unseen input

- **Training Data Augmentation**
  - applies scaling, rotation, translation, distortion, and other modifications to the set of available labeled training images

    → more training data, better generalization (and more work...)

    → more robust inference

Deep Learning – Non-functional requirements

Deep Learning has two major tasks

- **Training** of the Deep Neural Network, using labeled training data (often, images)
  - → Result: set of weight vectors for all layers
- **Inference** (or deployment) that uses a trained DNN to classify new data

DNN Training

- Training is a compute/communication intensive process –can take days to weeks
- Inference should have short latency – esp. for realtime use, e.g. in assisted / autonomous driving
- Latency lower bound given by number of layers, e.g. ResNet-152 has 152 layers

Faster training can be achieved by

- Optimized numerical libraries, esp. BLAS and convolution
- Parallelization and more special-purpose hardware
- esp., using GPUs (currently e.g. Nvidia DGX-1 with 8 V100 GPUs is a typical platform)
  - Power-hungry (ca. 300W each GPU – not suitable for mobile devices or automotive on-board use)
    → do training off-line or offload training to the cloud

Acceleration of DNNs

LINKÖPINGS UNIVERSITET

Recall: Main Enabling Factors of Deep Learning …

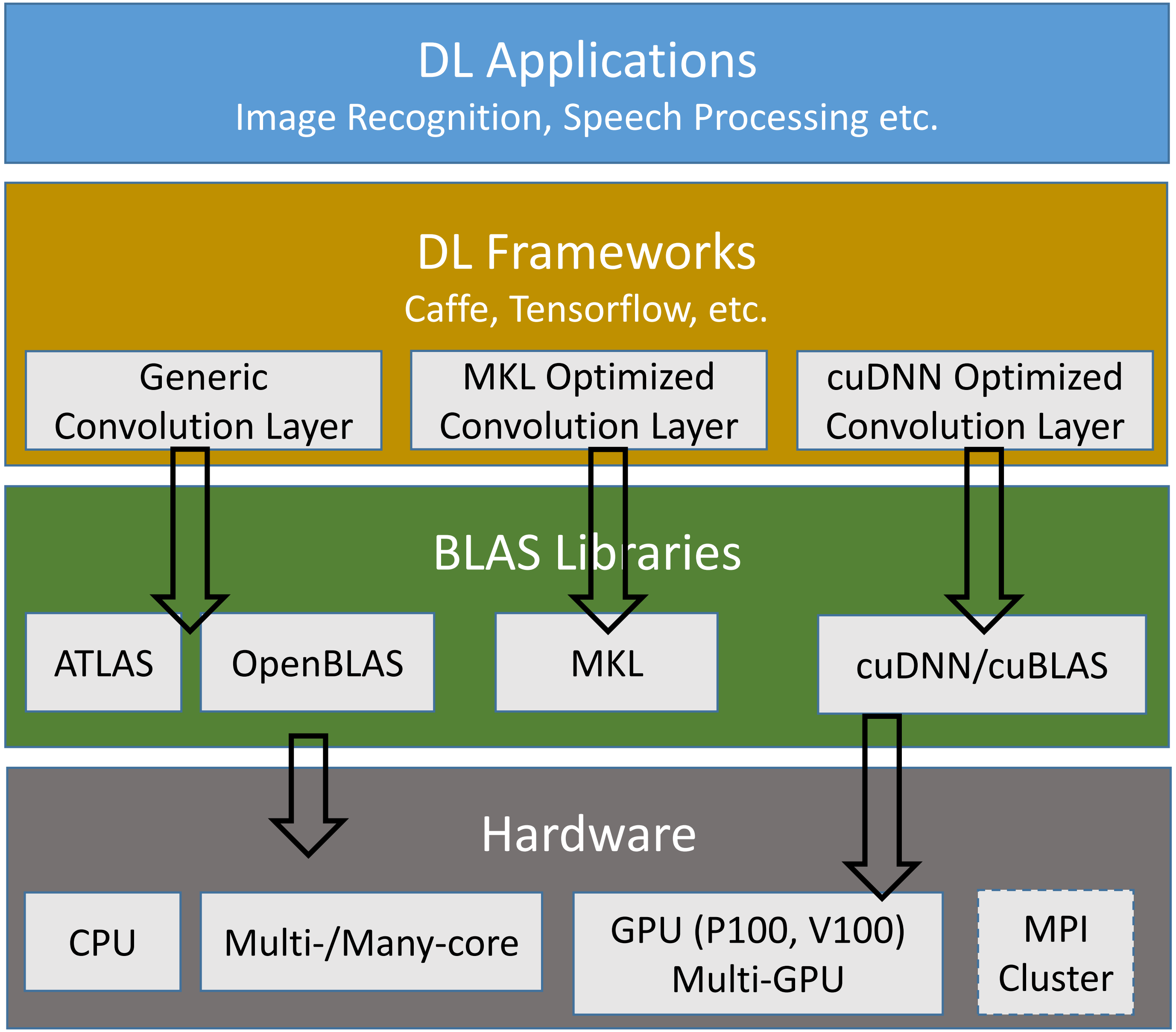
Computability of DNNs was made possible by modern and efficient hardware

- Mostly, based on dense/sparse **linear algebra** (BLAS2, BLAS3) computations
- GPUs enabled DNN **training** performance required for practical problems and realistic data sizes
  - massive data parallelism
  - *throughput* computing
  - learning is done off-line
- Modern CPUs, mobile GPUs and TPUs for low-latency DNN **inference**

Acceleration of DNN

- Requires efficient BLAS (Basic Linear Algebra Subroutines) Implementations
  - GEMM, SpMV, Dot product, ...
- Performance depends on the full software/hardware stack
  - Isolated analysis/optimization is not helpful

A. Awan, H. Subramoni, and D. K. Panda. "An In-depth Performance Characterization of CPU-and GPU-based DNN Training on Modern Architectures", Proc. Machine Learning on HPC Environments (MLHPC'17). ACM, New York, NY, USA, Article 8.

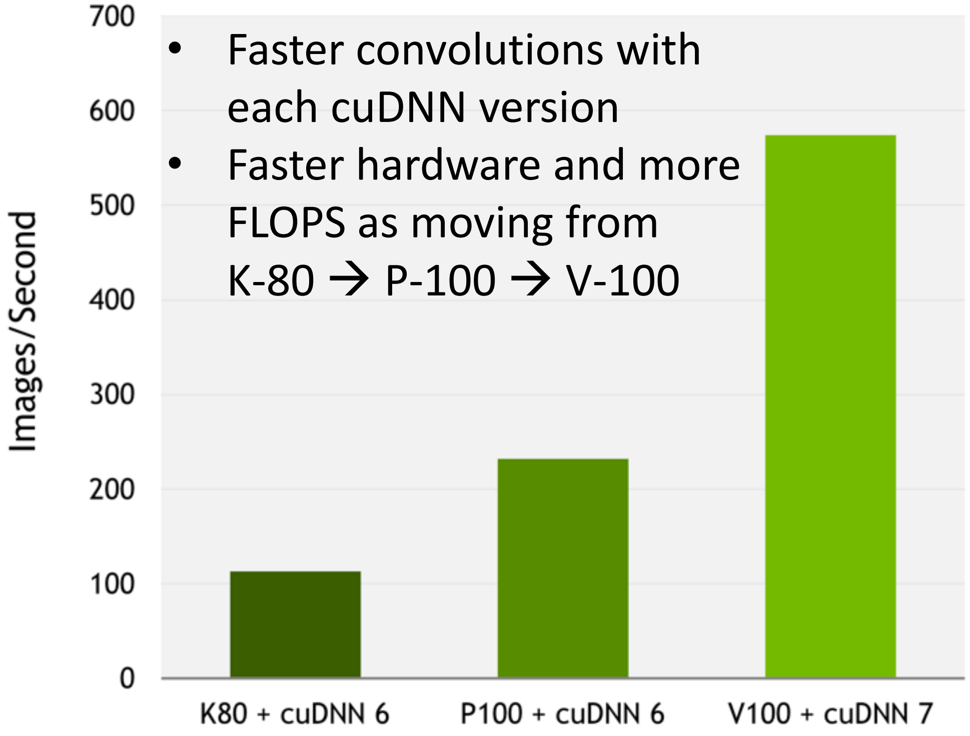


BLAS and DNN Libraries

- BLAS Libraries
  - Atlas/OpenBLAS (cf. TDDC78)
  - NVIDIA cuBLAS
  - Intel Math Kernel Library (MKL)

- Most compute-intensive layers generally optimized for a specific hardware
  - Convolution Layer, Pooling Layer, etc.

- DNN Libraries
  - Computational core: Convolutions
  - NVIDIA cuDNN (current: cudnn-v7) →
  - Intel MKL-DNN (MKL 2017)



Caffe2 performance (images/sec), Tesla K80 + cuDNN 6 (FP32), Tesla P100 + cuDNN 6 (FP32), Tesla V100 + cuDNN 7 (FP16, pre-release H/W and S/W). ResNet50, Batch size: 64

Image source: https://developer.nvidia.com/cudnn

Use of GPUs for Deep Learning

Nvidia GPUs are the main driving force for faster training of DL models

- The ImageNet Challenge (ILSVRC) →
- 90% of the ImageNet teams used GPUs in 2014
  - https://blogs.nvidia.com/blog/2014/09/07/imagenet/
- Used with Deep Neural Networks (DNNs) like AlexNet, GoogLeNet, and VGG
- A natural fit for DL due to their throughput-oriented, data-parallel architecture

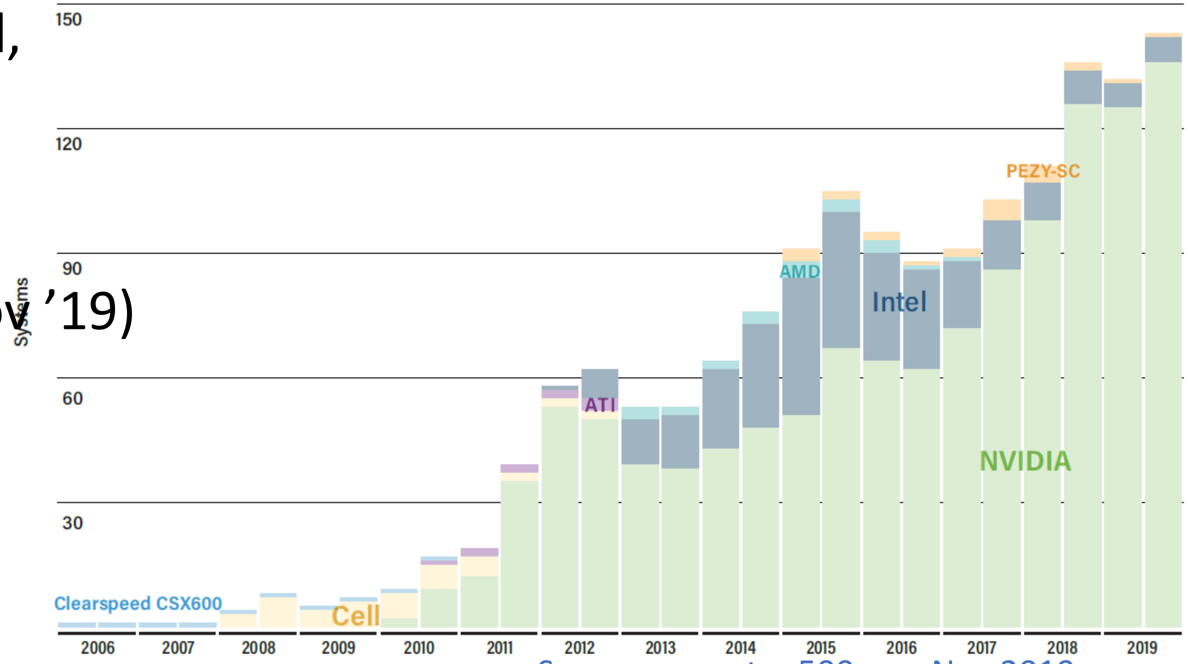
(images removed)

HPC systems

- >135 of TOP-500 HPC systems use NVIDIA GPUs (Nov ’19)
- CUDA-Aware Message Passing Interface (MPI)
- NVIDIA Fermi, Kepler, and Pascal architecture

NVIDIA DGX-1 and DGX1-V (Volta architecture)

- Dedicated DL super-computers



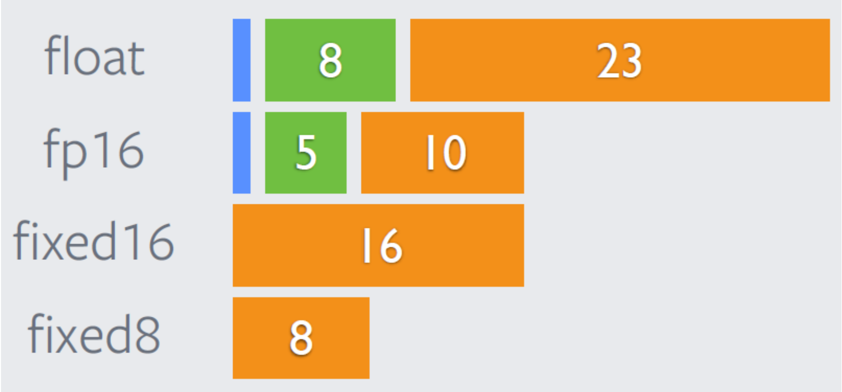
Source: www.top500.org, Nov 2019

→More about GPU architecture
in Ingemar Ragnemalm’s guest lecture

More ML Power by More Parallelism

Reduced Precision

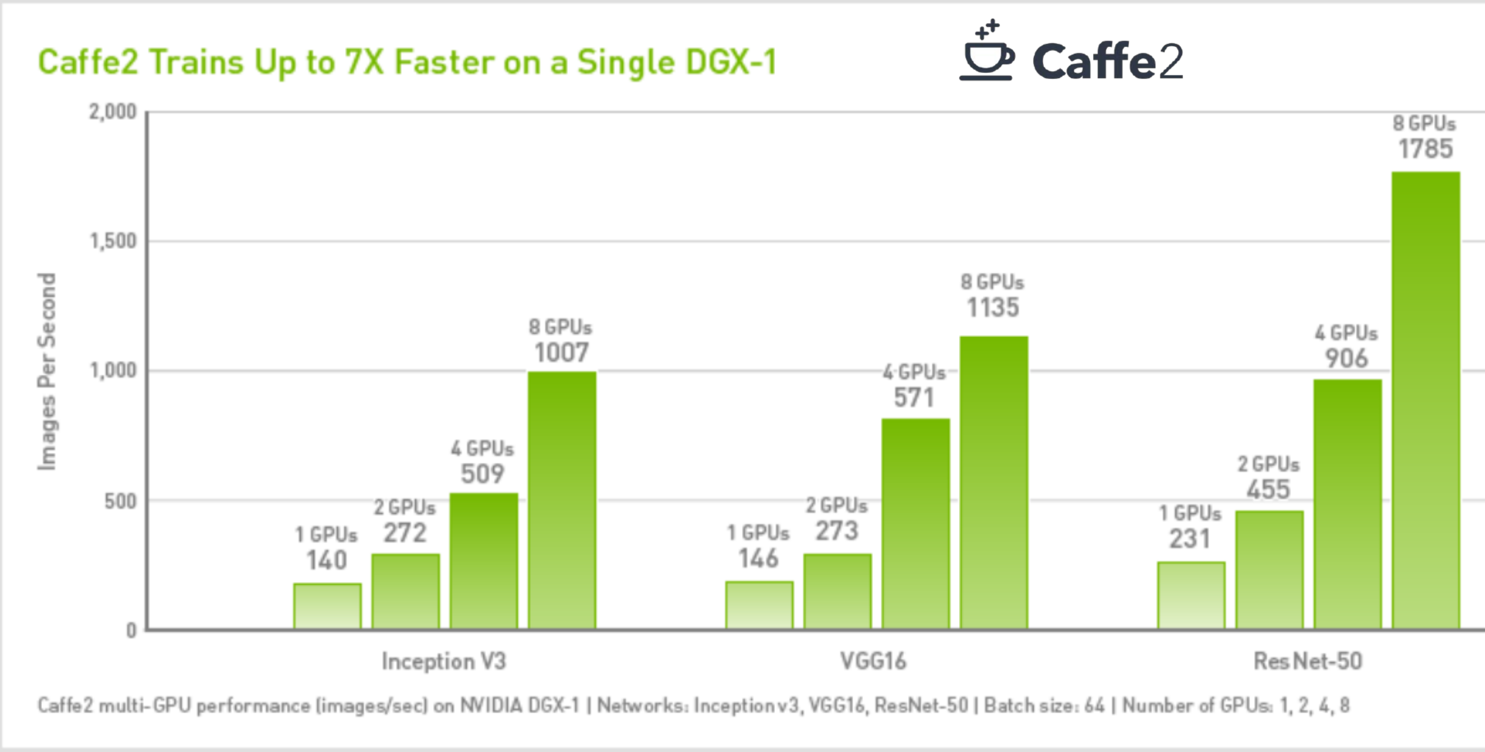
Essential for performance on modern DL-optimized GPUs (e.g. V100): Support for **reduced precision data types**



fp16b yields basically as good DL accuracy as fp32:

S. Gupta et al.: Deep Learning with Limited Numerical Precision. ArXiv 1502.02551v1, 2015.

Multi-GPU Computing



(Example code at caffe2/python/examples/resnet50_trainer.py)

Image source: Yangqing Jia, GTC-2017

A single server may not be enough

- Larger and deeper models are being proposed
  - AlexNet → ResNet → Neural Machine Translation (NMT)
  - Increasing #layers, complexity, training data
- DNNs require a lot of memory
  - Larger models cannot fit a GPU's device memory
- Single GPU training became a bottleneck
- Community has already moved to multi-GPU training, e.g. DGX-1 and similar multi-GPU servers
  - There is a limit to scale-up (8 GPUs)
- Possible direction currently being explored: **Multi-node (distributed parallel) training on GPU clusters**

DNN Distributed Parallel Training Strategies

applicable for both **multi-GPU** and **multi-node** scenarios

- **Data Parallelism** (most common)
  - **Intra-operator data parallelism**: parallelize calls to matmul, convolution etc. internally – usually exploited *within* one node/GPU, matrix sizes too small for distribution
  - **Intra-batch data parallelism**: replicate the network, partition the batch of (input,output) training items, train locally and reduce over the partial gradients computed by different workers (mapreduce pattern)

- **Model Parallelism**
  - **(intra-batch) Task parallelism** between independent BLAS/convolution calls:
  - The operators in the DNN network (model) are partitioned and mapped to the available workers.
  - Each worker evaluates and performs updates for only a subset of the model's parameters for ***all*** inputs.
  - Intermediate outputs (forward sweep) and corresponding gradients (backward sweep) need be communicated between workers.

- **Hybrid** Model and Data Parallelism

- **Inter-batch Parallelism by Pipelining**
  - Pipelining over the network layers
  - D. Narayanan *et al*.: PipeDream - Generalized Pipeline Parallelism for DNN Training. SOSP'19, ACM. https://cs.stanford.edu/~matei/papers/2019/sosp_pipedream.pdf

Automatic Selection of Parallelization Strategy

(image removed)

Image source:
http://on-demand.gputechconf.com/gtc/2017/presentation/s7724-minjie-wong-tofu-parallelizing-deep-learning.pdf

M. Wang: “Tofu: Parallelizing Deep Learning Systems with Automatic Tiling.” GTC 2017

Google TPU

cf. systolic matrix-multiply algorithm by Kung/Leiserson 1980, see also TDDC78

Tensor Processing Unit

V1 - for inference in the cloud

V2, V3, Edge-TPU announced (2018)

(images removed)

- CISC style instruction set
- Uses 256x256 8b MAC systolic arrays in multiply unit

https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

https://www.nextplatform.com/2017/04/05/first-depth-look-googles-tpu-architecture/

NB:

- Google TPU should not be confused with Nvidia's Tensor cores

Nvidia Tensor Core

- 4x4 Matrix-Matrix multiply in 1 clock cycle
- Systolic array of multipliers
- 16b x 16b operands (half-precision) → 32b result (single precision IEEE754)
- Deployed in Nvidia Volta GPGPU series since 2017
  - e.g. 640 Tensor cores in V100
    → for "AI" acceleration
  - Complement the 2,560 CUDA cores (64bit) + 5,120 CUDA cores (32bit)
    → for HPC acceleration
- Used via intrinsics in CUDA9, via a CUDA template include-only MM library, or via cuBLAS library

S. Markidis *et al.*: NVIDIA tensor core programmability, performance & precision. IPDPS Workshops 2018, IEEE.

Intel® Nervana™ Neural Network Processor (NNP)

- Formerly known as “Lake Crest”
- Recently announced as part of Intel’s strategy for next-generation AI systems
- Architecture targeted for deep learning
  - NNP-T1000 for training
  - NNP-I1000 for inference
- 1 TB/s High Bandwidth Memory (HBM)
- Spatial Architecture
- FlexPoint format
  - Similar performance (in terms of accuracy) to FP32 while using 16 bits of storage

Image source: https://ai.intel.com/intel-nervana-neural-network-processor-architecture-update/

Other Domain-Specific Architectures for DL

- Intel Nervana TPU
- GraphCore IPU
  - UK-based startup
  - Early benchmarks show 10-100x speedup over GPUs
- IBM TrueNorth (2014)
  - 4096 cores each simulating 256 neurons with 256 synapses each
  - Low-power, only 70mW
  - DARPA SyNAPSE with 16 TrueNorth chips →
- Intel Loihi (Spiking NN neuromorphic chip) (2017)
- Movidius Myriad-2 / Myriad-X VPU (Vision Processing Unit)

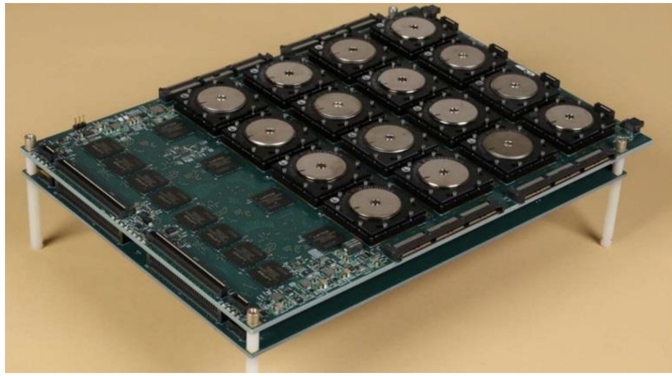


Image source: DARPA SyNAPSE, http://www.darpa.mil/NewsEvents/Releases/2014/08/07.aspx, Public Domain, https://commons.wikimedia.org/w/index.php?curid=34614979

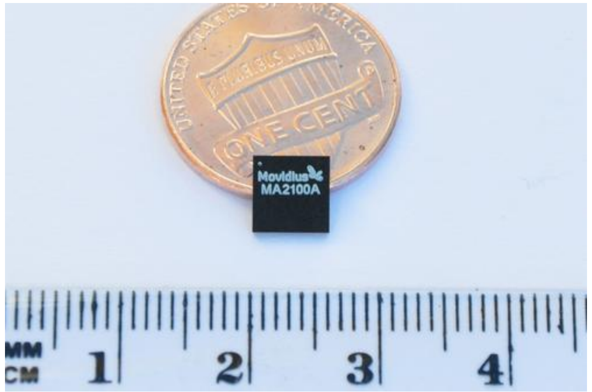
Cluster-class architectures:

- SpiNNaker
  - "Spiking Neural Network Architecture", U. Manchester (S. Furber)
  - http://apt.cs.manchester.ac.uk/projects/SpiNNaker/
  - 57,600 ARM9 processors (1M cores, 7TB RAM) - oct. 2018
  - "Models 1% of the human brain"

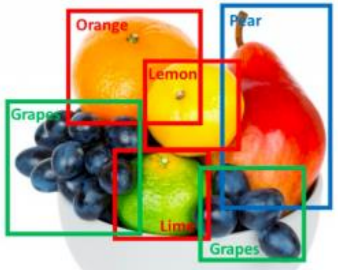
... (NB list is not complete, esp. some academic projects omitted)

Myriad 2

Image source:
Movidius / Intel

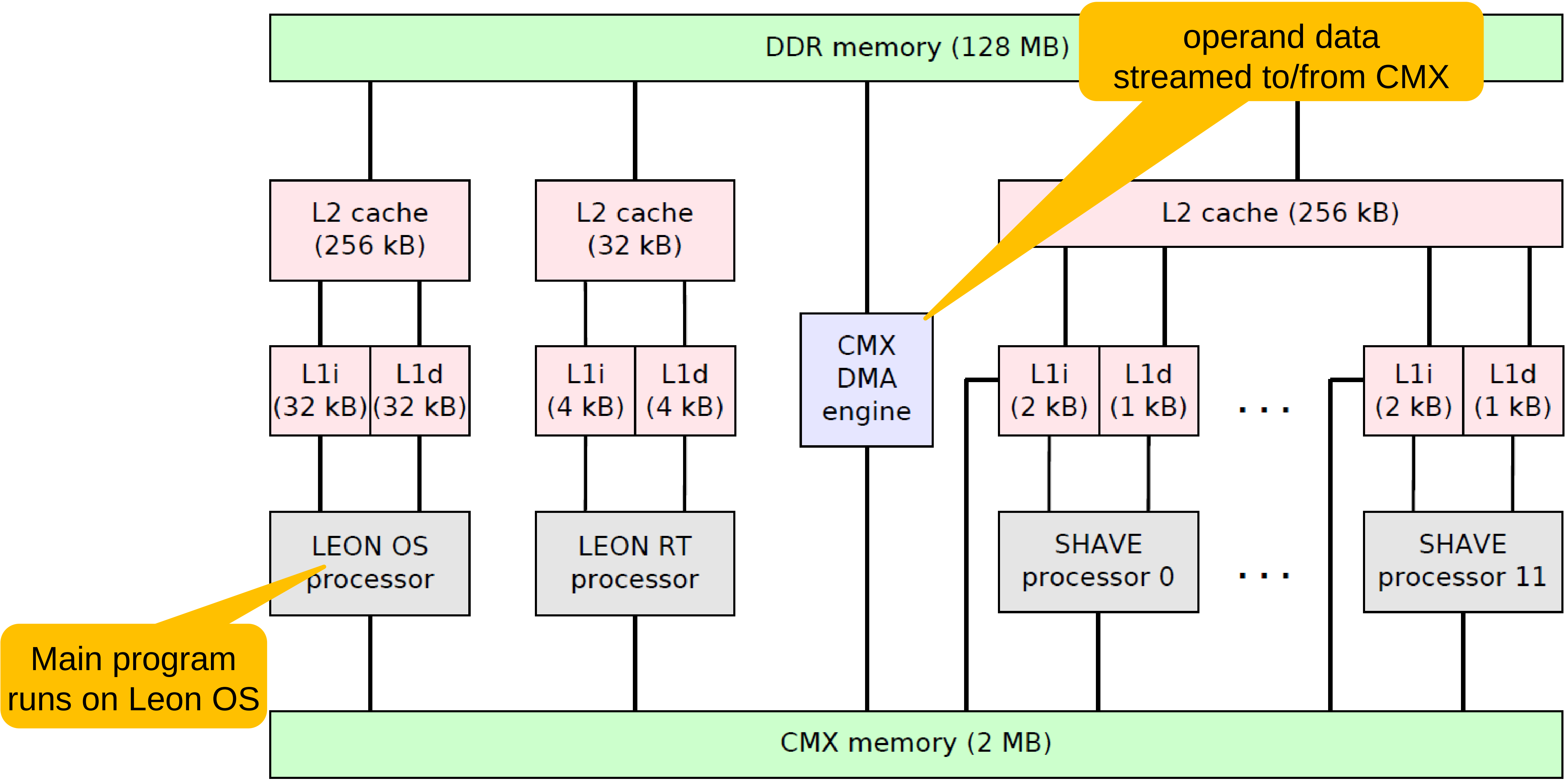


- Low-power ”Vision processor” (VPU) from Intel / Movidius, introduced 2015/2016
- 2 RISC cores (LEON)
- 12 VLIW SIMD cores (SHAVE)
- 2MB on-die scratchpad memory (CMX)
- L1, L2 caches (non-coherent)
- 128MB stacked LPDDR2 DRAM
- High performance per watt
  - Using SHAVEs up to 150 Gflops @ 1.2W
  - With built-in HW accelerators (SIPP) up to 2 Tops_{16} @ 0.5W
- For Vision, Linear Algebra, AR/VR, CNN Deep Learning
- Next generation VPU expected for spring 2020

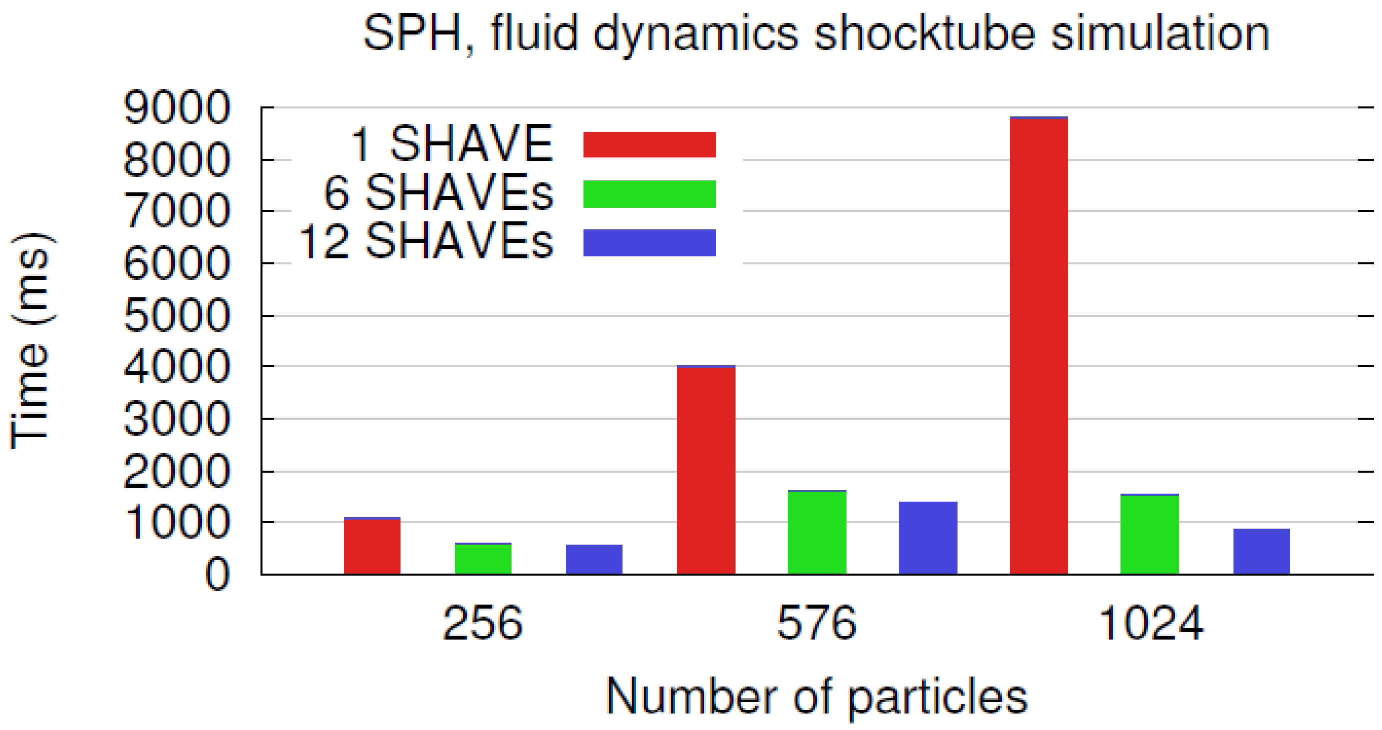


B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, V. Toma: Always-on Vision Processing Unit for Mobile Applications. *IEEE Micro* 35(2):56-66, 2015.

Myriad 2 Processor and Memory Structure



Example: SPH Application in **SkePU** running on **Myriad-2**



S. Thorarensen, R. Cuello, C. Kessler, L. Li and B. Barry: Efficient Execution of SkePU Skeleton Programs on the Low-Power Multicore Processor Myriad2. Proc. 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'16), Heraklion, Feb. 2016, pp. 398-402. IEEE. DOI: 10.1109/PDP.2016.123

- Same application was run on a GPU (Nvidia K20c)
  - Energy-efficiency calculated with $\frac{1}{time \cdot power}$
  - 33 times as energy-efficient when run on Myriad 2

SkePU documentation/download:
www.ida.liu.se/labs/pelab/skepu
(Myriad2 backend not included)

Challenge: Migrating ML to the Edge

- Machine learning is usually very energy-costly
  - Example: Autonomous driving uses ca. 2500 W*, the human brain uses ca. 12 W
- Background:
  Global ICT energy consumption (currently 5...9%) is expected to reach up to 20% of the world's total energy consumption by 2030

(image removed)

Image source: A. Andrae, T. Elder, "On Global Electricity Usage of Communication Technology: Trends to 2030", *Challenges* 6:117-157; doi:10.3390/challe6010117, 2015

O. Mitchell: "Self-Driving Cars Have Power Consumption Problems". *The Robot Report*, 26 Feb. 2018, reporting from CES'18. https://www.therobotreport.com/self-driving-cars-power-consumption/

Challenge: Migrating Learning to the Edge

- In the Cloud?
  - Recall: cloud = someone else's server farms offering storage and processing for hire
  - Can run the learning on relatively power-hungry high-end GPUs (e.g. Nvidia Xavier platform)
  - → offload learning work (and my data!) to the cloud
  - privacy concerns
- At the Edge?
  - cloud-connected devices, e.g. smart cameras, other sensors, smartphones, cars ...
  - mobile CPUs / GPUs still too weak for learning (OK for inference)
  - battery driven

Goal: drastically reduce energy consumption of machine learning

→Both at algorithmic level (e.g., low precision),
through code generation (e.g., SIMD), and hardware support

→could allow machine learning to run on edge devices, keep private data locally

→Domain-specific accelerators have a role to play here!

Challenges: Programmability, Portability, Performance Portability

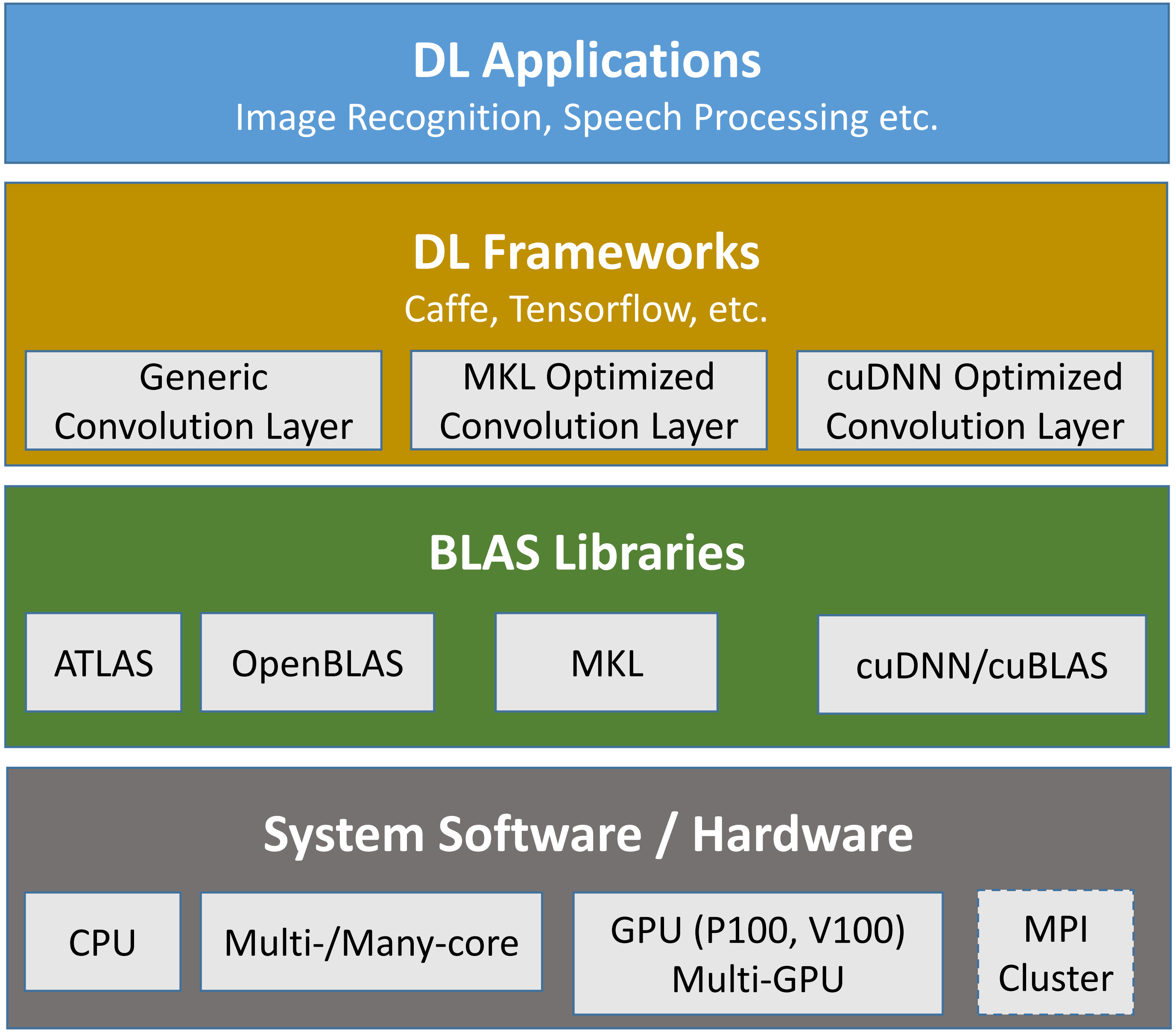
- Avoid hardcoding platform-specific optimizations (e.g., use of SIMD instructions, accelerators, multithreading, stream buffer sizes, ...) in the source code
- Use high-level / domain-specific constructs for abstraction and portability (e.g. SkePU skeletons, TensorFlow)
- Expose options to a separate autotuning toolchain (e.g. SkePU tuner)
- Runtime management of memory and data transfers

- Algorithmic improvements for energy efficiency still involves human effort ...

Programming Frameworks for Machine Learning

Here: Focus on Deep Learning / ANN

Software/Hardware Stack



A. Awan, H. Subramoni, and D. K. Panda. "An In-depth Performance Characterization of CPU-and GPU-based DNN Training on Modern Architectures", Proc. Machine Learning on HPC Environments (MLHPC'17). ACM, New York, NY, USA, Article 8.

Why do we need Deep Learning Programming Frameworks?

***Domain-specific* programming frameworks**

- hide most of the *nasty mathematics*
    - provide most common structures and functionalities ready to use
      → high programmer productivity
- and implementation details
    - e.g., memory management, data locality optimization, data transfers, parallelization, GPU/accelerator use
      → portability, programmability, performance
- focus on the *design* of neural networks
    - declarative, not imperative
      → portability, abstraction

Frameworks for DNN/CNN Programming

- Caffe (Berkeley)
- Caffe-2 (Facebook)
- Deeplearning4j
- TensorFlow (Google)
- Keras
- MatConvNet (MATLAB)
- MXNet
- Neon (Intel/Nervana)
- Theano
- Torch (Lua) / PyTorch (Python) (Facebook)
- Chainer
- Dlib
- Microsoft Cognitive Toolkit (Microsoft)
- TinyDNN
- ...

Open Neural Net eXchange (ONNX) Format

Caffe

http://caffe.berkeleyvision.org

- UC Berkeley BVLC Caffe (PhD thesis Yangqing Jia), open source (BSD)
- One of the most popular DL frameworks (#2 in 2017)
  - Winner of the ACM MM open source award 2014
  - Nearly 4,000 citations, usage by award papers at CVPR/ECCV/ICCV, and tutorials at ECCV'14 and CVPR'15
  - Adopted by industry
- 2017: Caffe2 by Facebook,
  - which was merged into PyTorch in 2018
- CaffeOnSpark by Yahoo!

- C++ and Python frontends
- Written in C++, with modular C++ backend
- Caffe is a single-node, multi-GPU framework
  - supports CUDA, cuDNN and Intel MKL
- Several efforts towards parallel/distributed training
  - OSU-Caffe -http://hidl.cse.ohio-state.edu/overview/
  - Intel-Caffe -https://github.com/intel/caffe
  - NVIDIA-Caffe -https://github.com/nvidia/caffe

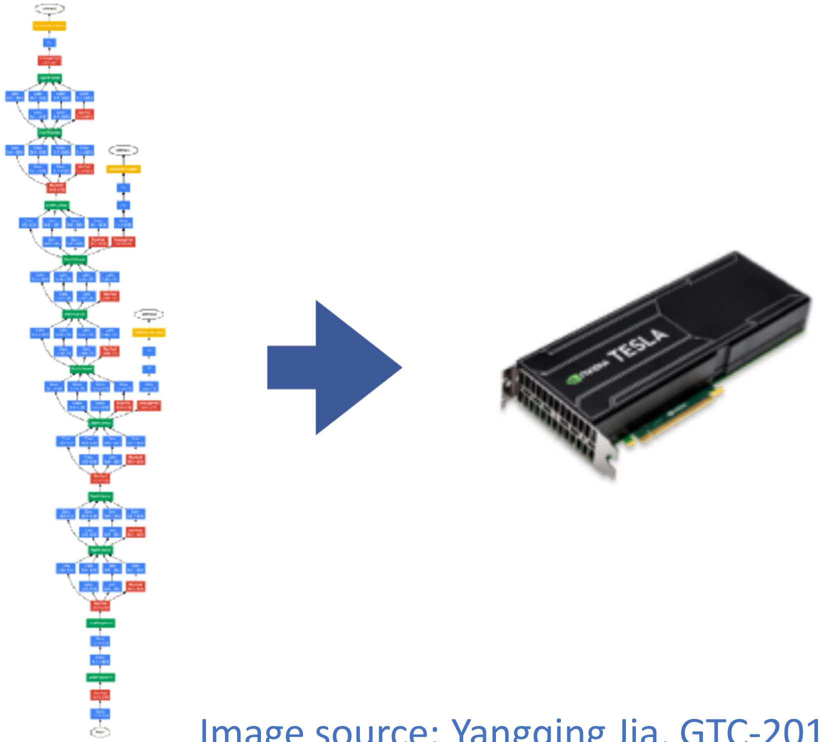
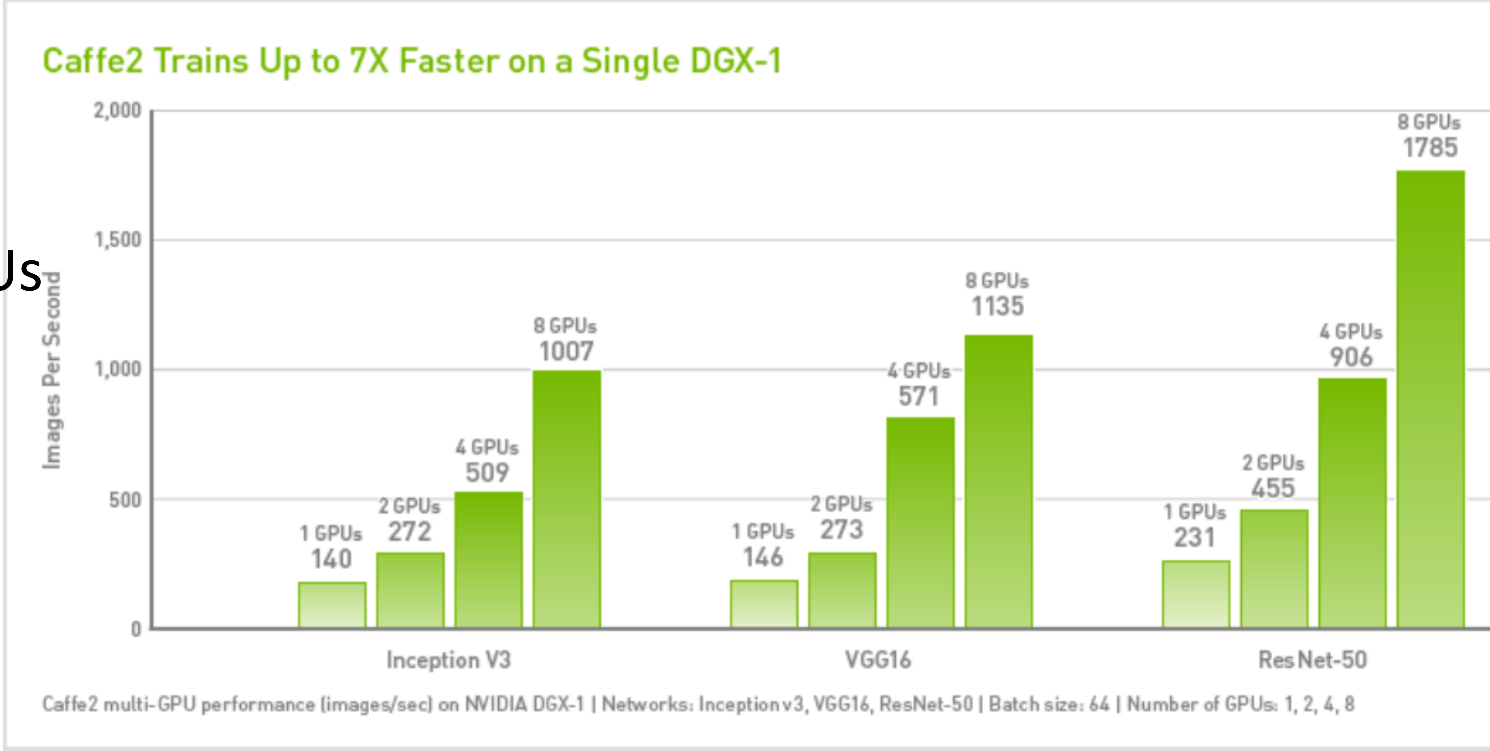


Image source: Yangqing Jia, GTC-2017

Caffe-2

https://github.com/caffe2/caffe2

- Symbolic differentiation
- Recurrent NNs supported
- Support for multi-GPU and distributed training
- Support for reduced precision data types on modern DL-optimized GPUs
- Cross-platform
- Extensible
- Applications in CV, AR, NLP, Speech



(Example code at caffe2/python/examples/resnet50_trainer.py)

Image source: Yangqing Jia, GTC-2017

Introduction to TensorFlow

TensorFlow https://tensorflow.org, https://github.com/tensorflow/tensorflow

- Today the most widely used framework
- Open-sourced by Google
  - Introduced 2015,
    replaced Google's *DistBelief* framework
    - J. Dean et al., "Large Scale Distributed Deep Networks", NIPS-2012
- Very flexible, but performance has been an issue
- Certain Python peculiarities like *variable_scope* etc.

- Runs on almost all execution platforms available
  (CPU, GPU, TPU, Mobile, etc.)
- Parallel/Distributed learning
  - Official support through gRPC library (Google 2015, open source, high-performance RPC)
  - Several community efforts (TensorFlow/contrib)
    - MPI version by PNNL: https://github.com/matex-org/matex
    - MPI version by Baidu: https://github.com/baidu-research/tensorflow-allreduce
    - MPI+gRPC version by Minds.ai: https://www.minds.ai

Tensors

- In TensorFlow, a **tensor** is an abstraction of a multidimensional (rectangular) array.
  - **Scalar** = 0-dimensional tensor
  - **Vector** = 1-dimensional tensor
  - **Matrix** = 2-dimensional tensor

- **Rank** = number of dimensions
- **Shape** = vector of extents
  - [] – scalar
  - [5] – vector containing 5 values
  - [3,4] – 3x4 matrix
- **Generic** in the element type
  - Must be a basic data type: bool, uint8, uint16, int8, int16, int32, int64, ..., float16, float32, float64, complex64, complex128, string

Tensor initializers

- **constant** (value, dtype=None, shape = None, name='Const', verify_shape=False)
  - returns a tensor containing the given value
- **zeros** (shape, dtype=tf.float32, name=None)
  - returns a tensor filled with zeros
- **ones** (shape, dtype=tf.float32, name=None)
- **fill** (dims, value, name=None)
  - returns a tensor filed with the given value (only float32)
  - ft1 = tf.fill ([1, 2, 3], 17.0) yields a 3D tensor (shape 1 x 2 x 3), all elements set to 17.0
- **linspace** (start, stop, num, name=None)
  - e.g., tf.linspace(5., 9., 5) yields [5. 6. 7. 8. 9.]
- **range** (start, limit, delta=1, dtype=None, name='range')
  - e.g. tf.range (3., 5., delta=0.5) yields [3.0 3.5 4.0 4.5 5.0]
- **random_normal**(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)
  - creates a tensor with normally distributed values
- **random_uniform**(shape, minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)
  - also: truncated_normal(), random_shuffle(), set_random_seed()

Tensor transformations

- **cast** (tensor, dtype, name=None)
  - changes the tensor's (element) data type to the given type
- **reshape** (tensor, shape, name=None)
  - returns a tensor with same elements as the given tensor with the given shape (only shape cast, same data layout – no copying of data)
- **squeeze**(tensor, axis=None, name=None)
  - removes dimensions of size 1
- **reverse**(tensor, begin, size, name=None)
  - extracts a portion of a tensor
- **stack** (tensors, axis=0, name='stack')
  - combines a list of tensors into a tensor of higher rank
  - e.g.: tf.stack (tf.constant([1.,2.]), tf.constant([3.,4.])) yields [[1. 2.][3. 4.]]
- **unstack** (tensor, num=None, axis=0, name='unstack')
  - splits a tensor into a list of tensors of lower rank

Tensor operations (type Map)

- **add** (x, y, name=None)
  - elementwise adds two tensors
  - similar: subtract, multiply, divide, div, mod, maximum, minimum, square_difference, pow
- **abs** (x, name=None)
  - elementwise absolute value
  - similar: negative, sign, reciprocal, scalar_mul, square, sqrt, rsqrt round, rint, ceil, floor, exp, log
- Could likewise be done using regular Python operators, i.e.,
  - ta1 = tf.add(a, b)
  - ta2 = a + b

  are equivalent.

Tensor operations (type Reduce / MapReduce)

- **argmax**(x, axis=None, name=None, dimension=None)
  - returns the index of the greatest element in the tensor
  - similar: argmin
- **tensordot**(a, b, axes, name=None)
  - returns the dot product of a, b along the given axes
  - similar: norm

Matrix computations

- **diag**, **trace**, **transpose**, **eye** (identity matrix),
- **matmul**, **matrix_solve**, **qr**, **svd**,
- **einsum** (equation, *inputs)
  - generic polyhedral tensor operation using Einstein notation
  - e.g. for m1=tf.constant([[1, 2],[3, 4]]),
tf.einsum(‘ij->ji’, m1) yields [[1 3] [2 4]]

Graphs and Tensors

Example:

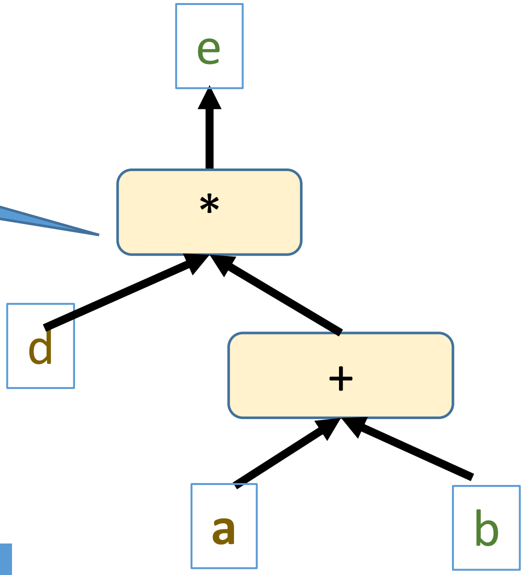
Internal graph-based representation is built by *lazy execution* of the calls to tensor constructors and operations:

import tensorflow as tf

c = tf.**add**(a, b)
e = tf.**multiply**(c, d)

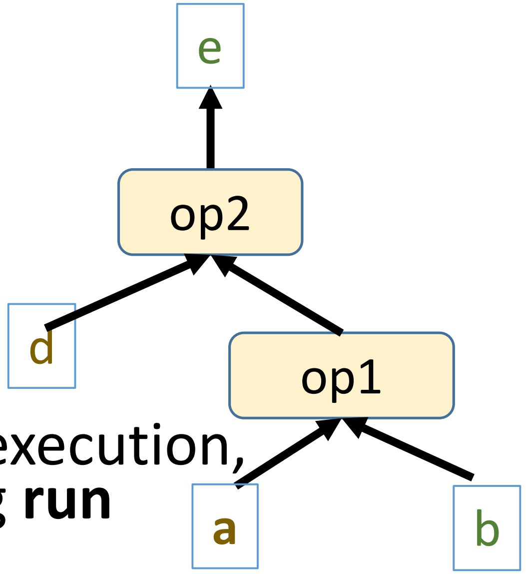
# current graph is implicit (context), can be retrieved:
tf.get_detfault_graph().**get_operations**()

New tensor and operation nodes are automatically built into the current graph (runtime representation).



The constructed graph is executed only when the Session.**run**() method is invoked.

Graphs



- Through operand tensor data flow we can chain multiple tensor constructors and operations on tensors into expression trees/DAGs → **graphs** (= containers for *code* computing on tensors)
- **Lazy execution** – tensor constructors and operations just recorded for execution, really executed (in data flow order) only in a **session** by explicitly calling **run**
  - Cf. the *lineages* in Apache *Spark* [Zaharia *et al*. 2010]
- Graphs can be serialized and exported to a file or launched on a remote system
  - GraphDef (binary or JSON text format) – basically an AST IR as known from compilers
- Graphs cannot be nested
- Encountered tensor constructors and operators are automatically added to the current (default) graph
  - Can traverse and compute over Graphs,
    e.g. print (tf.get_default_graph().get_operations())
    print (tf.get_default_graph().get_tensor_by_name('first_val:0'))
  - Can create new graphs and change default graph to new one (using newgraph.as_default())
- Graphs can hold some additional information beyond tensors and operations.
- **Automatic symbolic differentiation** of graphs (needed for gradient-based training) is possible as the graph structure is given and the operations' semantics are known

Tensors
vs. Variables
vs. Placeholders

"**Placeholders**" are tensor variables (here, x) created by

tf.**placeholder**(<elementtype>, <nrows>, <rowsize>)

Serve as symbolic input variables in the ANN function
Holds a batch of input data in training

"**Variables**" are tensor-*like* variables (here, **W**, b) created by

tf.**Variable**(<initializer>).

Serve as symbolic solution variables for the **training** process (i.e., the weights of the ANN)

Example: h = ReLU (**W** x + b)

Internal graph-based repr. of ANN is built by lazy execution:

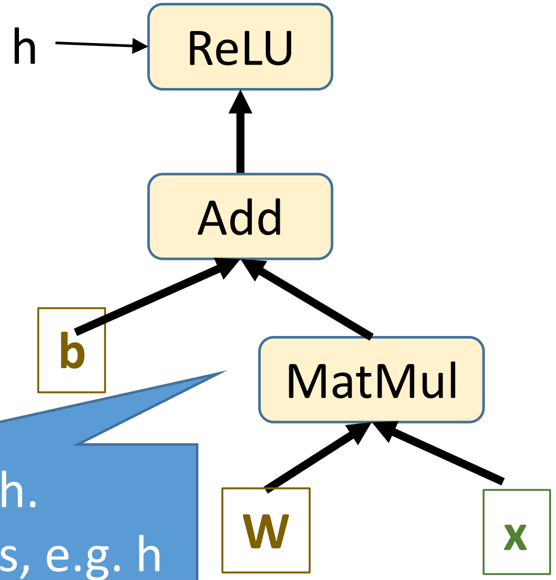
```
import numpy as np
import tensorflow as tf

b = tf.Variable( tf.zeros ((100, )))
W = tf.Variable( tf.random_uniform((784,100), -1, 1)
x = tf.placeholder( tf.float32, (100, 784))
h = tf.nn.relu( tf.matmul( x, W ) + b )
```

b = < 0,...,0 >

Initializer with entries of W in Uniform(-1,1)

100 x 784 tensor



The current graph.
Node object references, e.g. h

Sessions

- Create a session by calling tf.Session
  - 3 optional arguments: target execution engine, the graph, and target configuation info
- **run** method of Session kicks off the execution
  - Arguments: fetches, feeds, options, run_metadata
  - Variables (weights) must be initialized before starting training (bulk initialization support is available)

Deploy the graph in a session (for execution on CPU, GPU or TPU)

```
sess = tf.Session()
```

Usage: sess.run (fetches, feeds)

```
sess.run( tf.initialize_all_variables() )
```

Batch (lazy) execution:

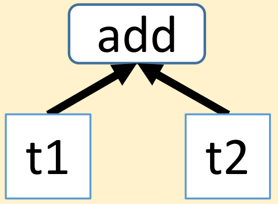
```
sess.run( h, { x: np.random.random( 100, 784) } )
```

Map Iterator: Initialize tensor placeholder x with 100 random images of 784 pixels each, and apply each to graph h

→ produces a new tensor of 100 output signals

- **Fetches**: the first argument of run(): (list of) graph nodes (operations, tensors) – *what* to execute. Return outputs of these nodes (evaluate where necessary).
  - Example:

```
t1 = tf.constant(3)
t2 = tf.constant(4)
with tf.Session() as sess:
        res = sess.run( t1 + t2 )   # fetches assigned to an operation (graph)
        print( res )        # prints 7
```



  - Example 2:

```
with tf.Session() as sess:
        res1, res2 = sess.run( [t1, t2] )   # fetches assigned to a list of code items
        print( res1 )       # prints 3
        print( res2 )       # prints 4
```

- **Feeds**: dictionary mapping from graph nodes to concrete (training) input values. Specifies the (desired) value of each graph node given in the dictionary.
  - Important for defining batches of training data

```
sess = tf.Session()
sess.run( tf.initialize_all_variables() )
for i in range(1000):
    batch_x, batch_label  =  data.next_batch()
    sess.run( train_step, feed_dict = { x: batch_x, label: batch_label } )
```

Training in batches

Compute entropy (loss, energy) and gradient

```
prediction = tf.nn.softmax( ... )      # output tensor of neural network
label = tf.placeholder ( tf.float32, [100,10] )    # expected output data
cross_entropy = - tf.reduce_sum( label * tf.log(prediction), axis = 1 )
```

sum up over the rows
of this tensor

```
#alternatively:
cross_entropy = tf.reduce_mean( - tf.reduce_sum( label * tf.log(prediction))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize( cross_entropy)
```

Optimizer object:
adds optimization operation to the computation graph

Alternative optimizers to GradientDescentOptimizer:

- MomentumOptimizer
- AdagradOptimizer
  - Adaptive gradient descent
  - works on subgradients
  - applicable to non-differentiable functions
- AdamOptimizer
- adaptive moment estimation, similar to Adagrad

All TensorFlow graph nodes have attached gradient operations computing the gradient w.r.t. parameters (here, W and b). The gradient operations are needed by the backpropagation algorithm used in training.

Training in Tensorflow – Overview

1. Construct a **graph** (mathematical expression) for the general model (e.g., a feed-forward ANN)
2. Declare **variables** to be updated as training is performed (weights, parameters)
3. Obtain an expression for the **loss** (error function) describing the difference between the model and the observation
4. Create an Optimizer with the loss function of Step 3, and call its **minimize**() method
5. (Optional) Configure the second argument of the session's run method to **feed** batches of data to the session
6. Execute the session by calling its **run**() method.

Linear Regression Example

```
def linear_regression():
   x = tf.placeholder( tf.float32, shape=(None, ), name='x')
   y = tf.placeholder( tf.float32, shape=(None, ), name='y')
   with tf.variable_scope('linreg') as scope:
       w = tf.Variable( np.random.normal(), name='w' )
       y_pred = tf.mul( w, x )
       error = tf.reduce_mean( tf.square( y_pred – y ))
   return x, y, y_pred, error
```

```
def run():
    x_batch, y_batch = generate_dataset()
    x, y, y_pred, error = linear_regression()
    optimizer = tf.train.GradientDescentOptimizer(0.1).minimize( error )
    init = tf.global_variables_initializer();
    with tf.Session() as session:
        session.run( init )
        feed_dict = { x: x_batch, y: y_batch }
        for _ in range(30):
            error, val, _ = session.run( [error, optimizer], feed_dict )
            print( 'error:', error.val.mean() )
        y_pred_batch = session.run( y_pred, { x: x_batch } )
```

Eager Mode

- Imperative code, like Python
- Debugging with breakpoints, step through like Python code
    - Can even step into the TensorFlow source code (is open-source)

Additional features in TensorFlow

- Generating summary data (graph metadata)
- TensorBoard – tool for visualization of summary data
- Logging
- Importing and exporting graphs
- Storing and loading models
- Interactive sessions
- Session hooks
- Session configuration (e.g. GPU usage)
- Weight initialization functions
- Dataset operations (concatenate, shuffle, shard, cache, filter, map, flat_map, zip, ...) for training/testing data e.g. from file
- Iterators
- Batching support functions
- Batch normalization functions
- Variable scopes, name scopes, ...
- DNN layer constructor library (tf.contrib.layers.fully_connected, ...)
- Convolution operator library (tf.layers.conv2d, tf.layers.max_pooling2d, ...)
- Image operations and conversions (tf.image)
- Support for RNNs (Recurrent ANNs)
- ...

Acceleration in Tensorflow

- Multicore CPU (default: 1 worker thread per CPU core)
  - Default execution mode is 1 thread per CPU core, using a thread pool.
  - Can set #threads (actually, tasks, partitions) for each operation, e.g. for Dataset.map()
- GPU
  - CUDA (for Nvidia GPUs)
  - OpenCL only if ComputeCpp is installed
    - www.codeplay.com/products/computesuite/computecpp
- Cluster (distributed runtime system, RPC, ClusterSpec)
- **config** parameter in tf.Session() should refer to a ConfigProto buffer with proper configuration settings
  - device_count, intra_op_parallelism_threads (max. #tasks), inter_op_parallelism_threads, session_inter_op_thread_pool, placement_period, device_filters, gpu_options (e.g. GPU device memory pre-allocation), allow_soft_placement, graph_options, operation_timeout_in_ms, rpc_options, cluster_def
  - conf = tf.**ConfigProto**(intra_op_parallelism_threads=6, inter_op_parallelism_threads=8)
  - also additional configuration options to Session.**run**() call possible

Colab

- colab.research.google.com
  - Research project by Google
- Google-docs-like notebook for zero-install-Tensorflow
  - runs in a virtual machine in the Google cloud
  - including access to GPU
  - includes a Jupyter notebook for Python
  - Python 2 and Python 3 supported
  - notebooks can be saved to Google Drive and shared

Keras

- tf.keras
- High-level API for TensorFlow, lego-like
- concept-heavy but code-light
- Many parameters, but good defaults
- 5 steps
  1. collect a data set (most of the work)
  2. build the model (few lines of code)
  3. train (1 line)
  4. evaluate (1 line)
  5. predict (1 line)

MNIST: 28x28 = 784 pixels per image
Training: 60,000 images
Testing: 10000 images

Example: Download a dataset for training and testing:

```
(train_images, train_labels), (test_images, test_labels)
   = tf.keras.datasets.mnist.load_data()
....  (reformat the images)
```

Example: NN model with 3 layers of 512, 256 and 10 neurons

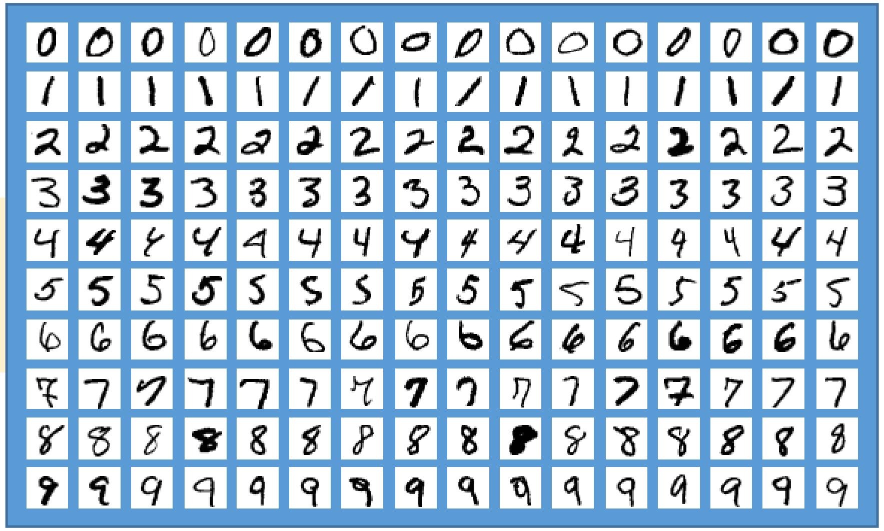
```
model = tf.keras.Sequential()
model.add( tf.keras.layers.Dense( 512, activation = tf.nn.relu,
                                   input_shape=(784, )))
model.add( tf.keras.layers.Dense( 256, activation = tf.nn.relu )
model.add( tf.keras.layers.Dense( 10, activation = tf.nn.softmax))
model.compile( error = ... , optimizer = ... )
```

```
model.fit( train_images, train_labels, epochs = 5)
```

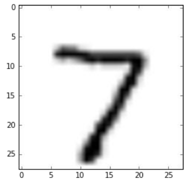
```
error, accuracy = model.evaluate( test_images, test_labels)
```

Keras example: Prediction / Inference

```
scores = model.predict( test_images[0] )
print( np.argmax( scores ))
```



first test image in MNIST:



Output layer: 10 neurons

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

Evidence (scores):

0.0 0.2 0.0 0.0 0.0 0.0 0.0 **0.7** 0.0 0.0

For large input data sets (> MNIST): stream the input data set.

Keras Example

(image removed)

Source: H. Huttonen: “Deep Neural Networks: A Signal Processing Perspective”. In S. Bhattacharyya et al.: *Handbook of Signal Processing*, Third Edition, Springer, 2019.

Keras Example

Keras code for creating a small convolutional network with random weights.

(images removed)

References (TensorFlow and Keras)

- Google: Machine Learning Crash Course
  - g.co/machinelearningcrashcourse
  - takes a few days fulltime studies
- Book:
  F. Chollet (= the author of Keras):
  Deep Learning with Python (Manning, 2017)
- Book:
  M. Scarpino: *Tensorflow for dummies*. Wiley, 2018
  - Available as electronic copy in the LiU library
- Web resources:
  - colab.research.google.com
  - github.com/tensorflow/workshops
  - Keras-compatible API with Tensorflow.js: js.tensorflow.org
- More on Machine learning: ai.google/education

More DL Programming Frameworks …

- Facebook Torch / PyTorch
- Microsoft Cognitive Toolkit
- Chainer / ChainerMN
  https://chainer.org
- MXNet
  http://mxnet.io
- Theano
  http://deeplearning.net/software/theano/
- Blocks
  https://blocks.readthedocs.io/en/latest/
- Intel Neon
- Intel BigDL
  https://software.intel.com/en-us/articles/bigdl-distributed-deep-learning-on-apache-spark
- Livermore Big Artificial Neural Network Toolkit (LBANN)
  https://github.com/LLNL/lbann
- Deep Scalable Sparse Tensor Network Engine (DSSTNE)
  https://github.com/amzn/amazon-dsstne
- …

Facebook Torch, PyTorch

https://pytorch.org

- Torch was written in Lua
  - No wide-spread adoption
- PyTorch is a Python adaptation of Torch
  - Gaining lot of attention
- Several contributors
  - Largest support by Facebook
  - Very active development
- PyTorch and Caffe2 were merged in March 2018
- Key selling point:
  ease of expression and “define-by-run” approach
- Recently got distributed training support:
  http://pytorch.org/docs/master/distributed.html

Microsoft Cognitive Toolkit https://github.com/microsoft/cntk

- Formerly CNTK, now called the Cognitive Toolkit
- C++ and Python frontend
- C++ backend
- ASGD (averaged stochastic gradient descent), SGD, and several other choices for solvers/optimizers
- Constantly evolving support for multiple platforms
- Focus on performance
- Parallel and Distributed Training
  - MPI and NCCL2 support
  - Community efforts

Neon

- Neon is a Deep Learning framework by Intel/Nervana
- Works on CPUs as well as GPUs
- https://github.com/NervanaSystems/neon

- Nervana Graph IR:
  - https://github.com/NervanaSystems/ngraph
  - www.ngraph.ai
  - open source C++ library, compiler and runtime for Deep Learning

(image removed)

Image source: https://ai.intel.com/intel-nervana-graph-preview-release/

Open Neural Network eXchange (ONNX) Format

- Not a Deep Learning framework but an open format to exchange “**trained**” networks across different frameworks
- Currently supportedFrameworks:
  Caffe2, Chainer, CNTK, MXNet, PyTorch
- Converters: CoreML, TensorFlow
- Runtimes: NVIDIA
- https://onnx.ai
- https://github.com/onnx

Programming Frameworks for Deep Learning
2 Main Variants

Define-and-Run:
Theano, Tensorflow,
Caffe, Torch, and most others

Construct a computational graph in advance of training. **Declarative.**

(image removed)

Define-by-Run:
PyTorch, Chainer
TensorFlow 1.5+ has an *eager* mode

Build the computational graph "on-the-fly" *during* training. **Imperative.**
More appropriate for recurrent and stochastic neural networks

(image removed)

Popularity of DL Programming Frameworks

(image removed)

(image removed)

Questions?

Acknowledgments

- Image sources: see slide annotations
- Some slides adapted from a tutorial at PPoPP’18 by D. K. Panda, Ohio State University
- Google online video lectures on Tensorflow

LINKÖPINGS
UNIVERSITET