LECTURE UNITS
MAT391
Topics in Machine Learning

Kwai Wong
University of Tennessee, Knoxville

September 16, 2022

Acknowledgements:

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, www.jics.utk.edu/lapenna, NSF award #202409
- www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,
- MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502
- Source code: www.bitbucket.org/icl/magmadnn
- www.bitbucket.org/cfdl/opendnnwheel

INNOVATIVE COMPUTING LABORATORY

NSF

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Unit 1: Big Data Computing Ecosystem

- Linear Algebra Revisit, Performance
- GPU Brief Overview
- Google Colaboratory
- Introduction to Python
- Data Intensive Sciences,
- Deep Neural Network
- TensorFlow MNIST example

JICS Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

Linear Algebra

David Cherney, Tom Denton,
Rohit Thomas and Andrew Waldron

Introduction to Linear Algebra

Mark Goldman

Emily Mackevicius

Lecture slides for

Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares

Stephen Boyd Lieven Vandenberghe

Caltech Division of the Humanities and Social Sciences

Quick Review of Matrix and Real Linear Algebra

KC Border
Subject to constant revision
Last major revision: December 12, 2016
v. 2019.10.23::14.52

Linear Algebra Review and Reference

Zico Kolter (updated by Chuong Do)

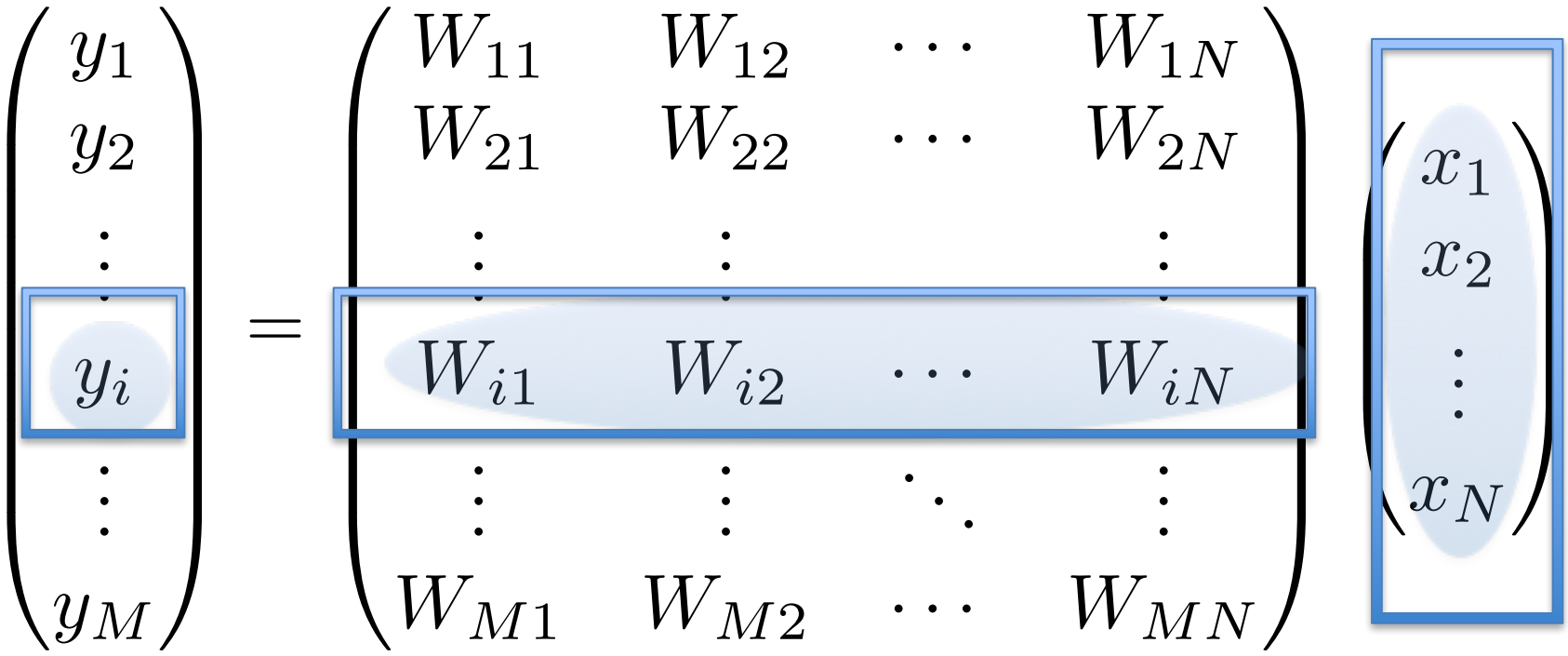
September 30, 2015

Introduction to Linear Algebra Review MV and MM

Mark Goldman

Emily Mackevicius

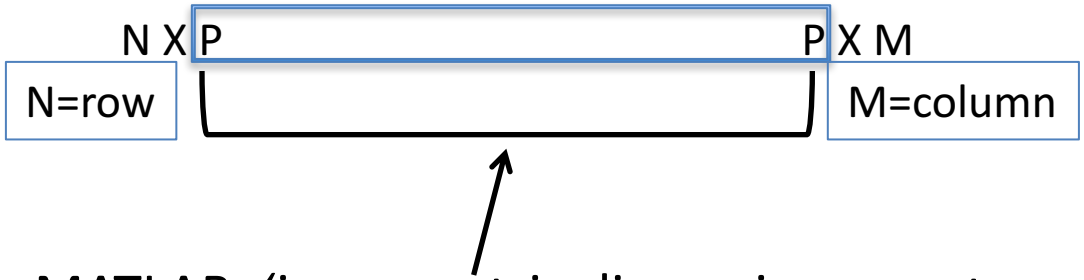
Matrix times a vector: **inner product interpretation**



- Rule: the i^{th} element of **y** is the dot product of the i^{th} row of W with **x**

Product of 2 Matrices

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NP} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1M} \\ B_{21} & B_{22} & \cdots & B_{2M} \\ \vdots & \vdots & & \vdots \\ B_{P1} & B_{P2} & \cdots & B_{PM} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & & \vdots \\ C_{N1} & C_{N2} & \cdots & C_{NM} \end{pmatrix}$$



- MATLAB: 'inner matrix dimensions must agree'
- **Note:** Matrix multiplication doesn't (generally) commute, **AB** ≠ **BA**

Matrix times Matrix:
by inner products

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & & \vdots \\ A_{i1} & A_{i2} & \cdots & A_{iP} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NP} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1j} & \cdots & B_{1M} \\ B_{21} & B_{22} & \cdots & B_{2j} & \cdots & B_{2M} \\ \vdots & \vdots & & \vdots & & \vdots \\ B_{P1} & B_{P2} & \cdots & B_{Pj} & \cdots & B_{PM} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & C_{ij} & \vdots \\ C_{N1} & C_{N2} & & C_{NM} \end{pmatrix}$$

$$C_{ij} = \sum_{k=1}^{P} A_{ik} B_{kj}$$

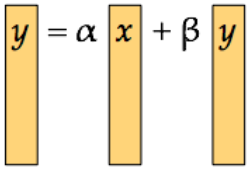
- C_{ij} is the inner product of the i^{th} row of **A** with the j^{th} column of **B**

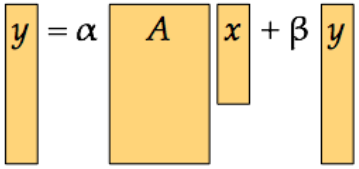
Basic Linear Algebra Subprograms (BLAS)

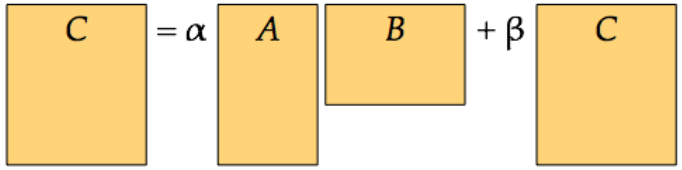
- **BLAS is a library of standardized basic linear algebra computational kernels created to perform efficiently on serial computers taking into account the memory hierarchy of modern processors.**
- **BLAS1 does vectors-vectors operations.**
  - **Saxpy = y(i) = a* x(i) + y(i), ddot= ∑x(i) *y(i)**
- **BLAS2 does matrices - vectors operations.**
  - **MV : y = A x + b (dgemv)**
- **BLAS3 operates on pairs or triples of matrices.**
  - **MM : C = αAB + βC, Triangular Solve : X = $\alpha T^{-1} X$**
- **Level3 BLAS is created to take full advantage of the fast cache memory. Matrix computations are arranged to operate in block fashion. Data residing in cache are reused by small blocks of matrices. dgemm**
- **openBLAS, MKL(Intel), ESSL(IBM), cuBLAS(Nvidia), BLIS(AMD)**

Computational Intensity = FLOPS/ Memory Access

- ✓ Level 1 BLAS — vector operations
  - ✓ $O(n)$ data and flops (floating point operations)
  - ✓ Memory bound:
    $O(1)$ flops per memory access
- ✓ Level 2 BLAS — matrix-vector operations
  - ✓ $O(n^2)$ data and flops
  - ✓ Memory bound:
    $O(1)$ flops per memory access
- ✓ Level 3 BLAS — matrix-matrix operations
  - ✓ $O(n^2)$ data, $O(n^3)$ flops
  - ✓ Surface-to-volume effect
  - ✓ Compute bound:
    $O(n)$ flops per memory access







oneAPI

A New Era of Accelerated Computing

What Is oneAPI? | Get the Toolkits

Say Goodbye to Proprietary Lock-In

View the oneAPI Specification →

Realize All the Hardware Value

Compare CPUs, GPUs & FPGAs for oneAPI →

Search this document

Intel oneAPI Math Kernel Library (oneMKL)

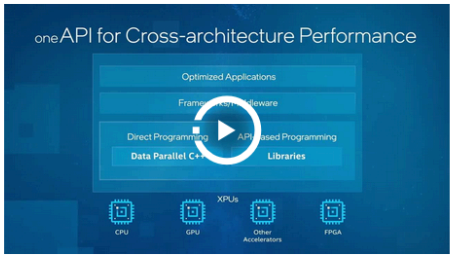
The Intel® oneAPI Math Kernel Library (oneMKL) is a computing math library of highl require maximum performance. oneMKL contains the high-performance optimizatio C/Fortran programming language interfaces) and adds to them a set of SYCL* interfa Graphics Technology for certain key functionalities.

You can use OpenMP* offload to run standard oneMKL computations on Intel GPUs. Fortran interfaces for more information.

The new SYCL interfaces with optimizations for CPU and GPU architectures have bee computation:

- BLAS and LAPACK dense linear algebra routines
- Sparse BLAS sparse linear algebra routines
- Random number generators (RNG)
- Vector Mathematics (VM) routines for optimized mathematical operations on v
- Fast Fourier Transforms (FFTs)

For the complete list of features, documentation, code samples, and downloads, visit oneMKL as part of the oneAPI Base Toolkit, consider that priority support is available For the community-supported open-source version, visit the oneMKL GitHub* page.



What Is oneAPI?

oneAPI is an open, cross-architecture prc single code base across multiple architec vendor lock-in.

Explore the Application Catalog

View the Latest Projects

Bookmark the Tech Library

Read What's New

Multiplying Matrices Using dgemm

Intel MKL provides several routines for multiplying matrices. The most widely used is the *dgemm* routine, which calculates the product of double precision matrices:

$$C \Leftarrow \alpha A * B + \beta C$$

The *dgemm* routine can perform several calculations. For example, you can perform this operation with the transpose or conjugate transpose of *A* and *B*. The complete details of capabilities of the *dgemm* routine and all of its arguments can be found in the *cblas_?gemm* topic in the *Intel Math Kernel Library Developer Reference*.

Use dgemm to Multiply Matrices

This exercise demonstrates declaring variables, storing matrix values in the arrays, and calling *dgemm* to compute the product of the matrices. The arrays are used to store these matrices:

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 & \cdots & 1000.0 \\ 1001.0 & 1002.0 & 1003.0 & \cdots & 2000.0 \\ 2001.0 & 2002.0 & 2003.0 & \cdots & 3000.0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 999001.0 & 999002.0 & 999003.0 & \cdots & 1000000.0 \end{bmatrix} \quad B = \begin{bmatrix} -1.0 & -2.0 & -3.0 & \cdots & -1000.0 \\ -1001.0 & -1002.0 & -1003.0 & \cdots & -2000.0 \\ -2001.0 & -2002.0 & -2003.0 & \cdots & -3000.0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -999001.0 & -999002.0 & -999003.0 & \cdots & -1000000.0 \end{bmatrix}$$

This call to the dgemm routine multiplies the matrices:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A, k, B, n, beta, C, n);
```

The arguments provide options for how Intel MKL performs the operation. In this case:

CblasRowMajor
Indicates that the matrices are stored in row major order, with the elements of each row of the matrix stored contiguously as shown in the figure above.

CblasNoTrans
Enumeration type indicating that the matrices *A* and *B* should not be transposed or conjugate transposed before multiplication.

m, n, k
Integers indicating the size of the matrices:

- *A*: m rows by k columns
- *B*: k rows by n columns
- *C*: m rows by n columns

alpha
Real value used to scale the product of matrices *A* and *B*.

A
Array used to store matrix *A*.

k
Leading dimension of array A, or the number of elements between successive rows (for row major storage) in memory. In the case of this exercise the leading dimension is the same as the number of columns.

B
Array used to store matrix *B*.

n
Leading dimension of array B, or the number of elements between successive rows (for row major storage) in memory. In the case of this exercise the leading dimension is the same as the number of columns.

beta
Real value used to scale matrix *C*.

C
Array used to store matrix *C*.

n
Leading dimension of array C, or the number of elements between successive rows (for row major storage) in memory. In the case of this exercise the leading dimension is the same as the number of columns.

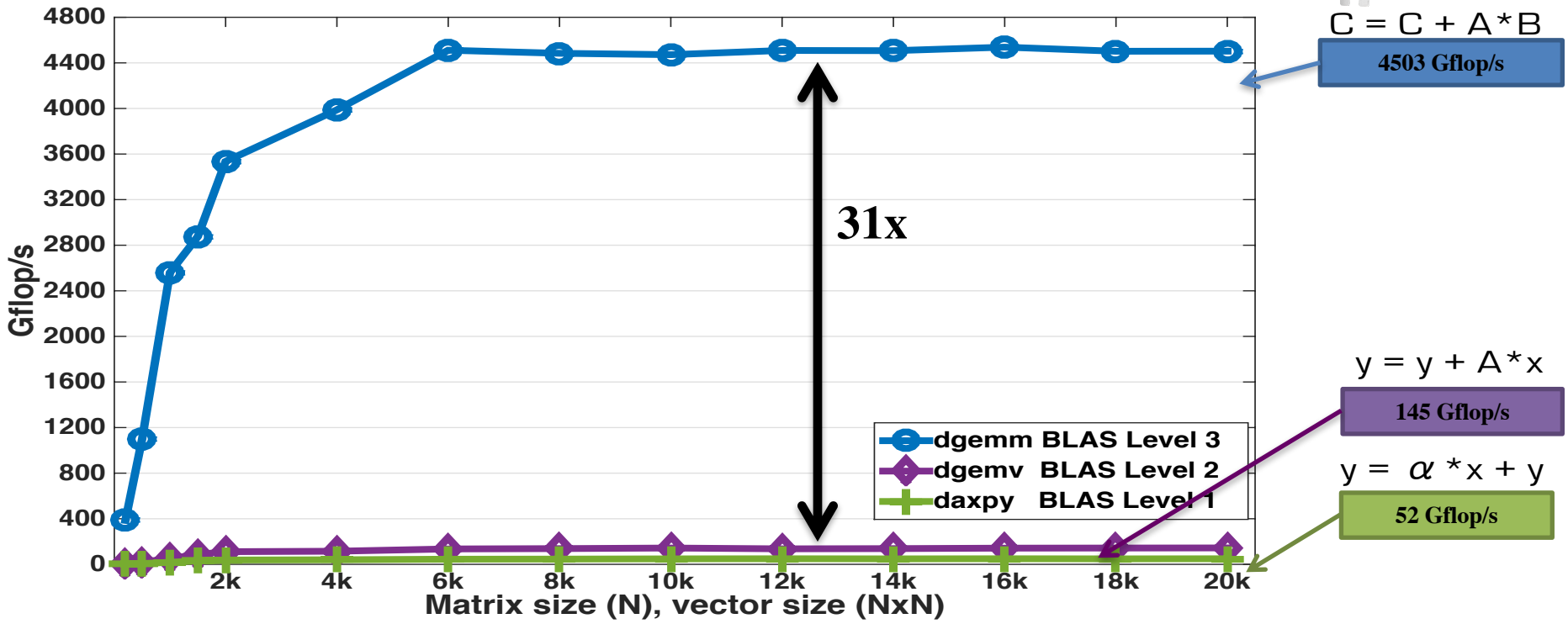
Level 1, 2 and 3 BLAS on GPU

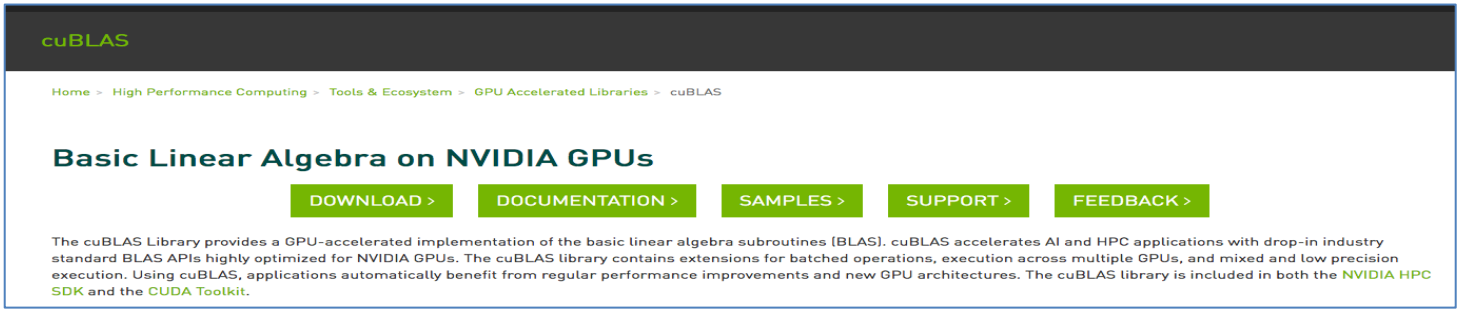
Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

Nvidia P100, The theoretical peak double precision is 4700 Gflop/s, CUDA version 8.0

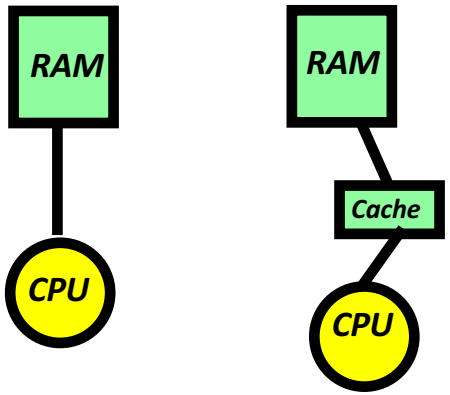
Nvidia **P100**, 1.19 GHz, Peak DP = 4700 Gflop/s

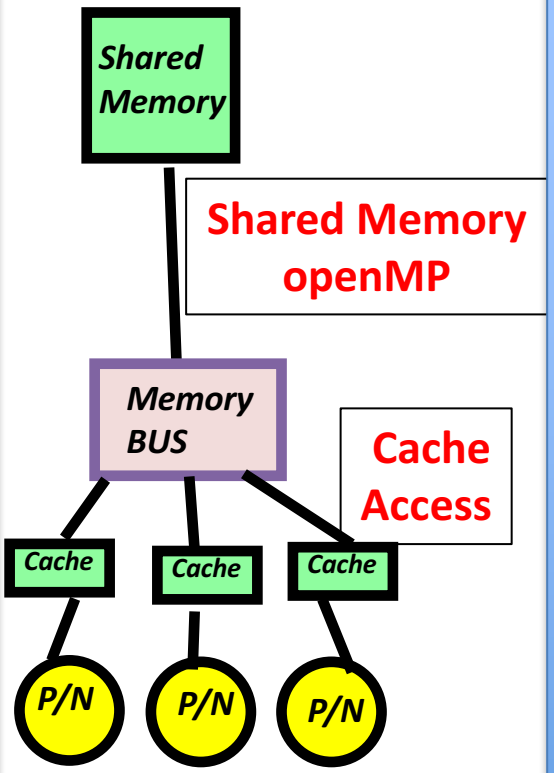


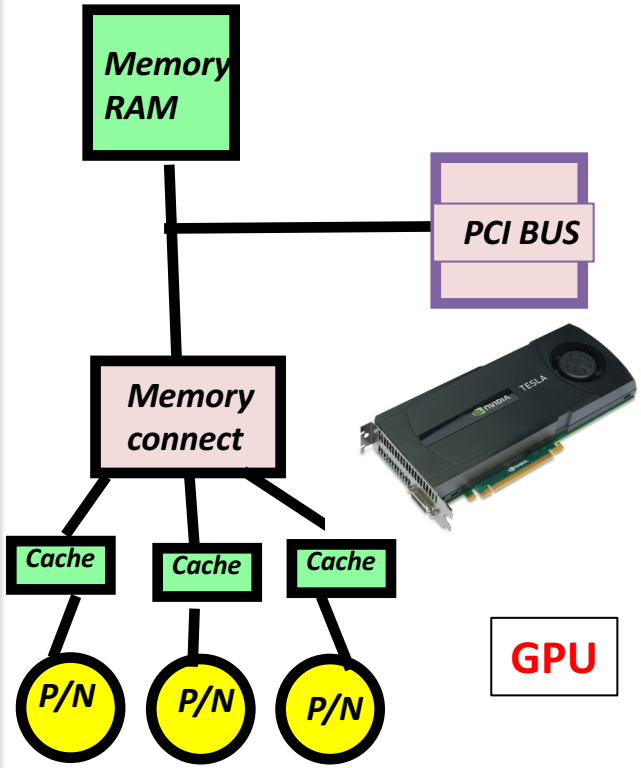


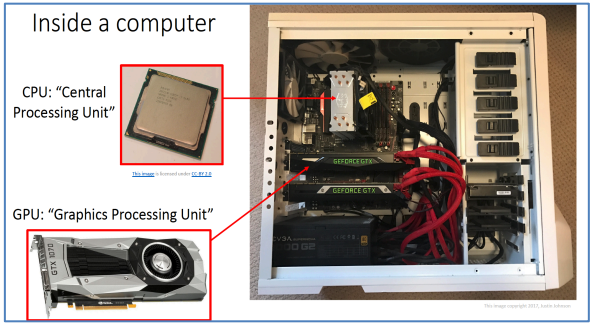
Simple story of Computers

Major Bottleneck - Communication, Memory Access









Numbers : Lots of Them: bit, byte, FLOP(S)

- Core : computing unit : processor
- Dual core machine (Intel or AMD CPU) : a CPU with 2 cores, each core is a 2.4 GHz computing unit with 2GB of RAM (memory in the processor not disk space)
- Binary bits (b) : "0" or "1" , 1 Byte (B) = 8 bits
- Binary number : 11111111= $(2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^{0)} = (2^8 -1) = 255$!!
- 32 bits machine or operating system => largest integer (all positive) = $(2^{32} -1)$ = (4,294,967,296 -1) or range of integer = $-(2^{31})$ to $(2^{31} -1)$
- 64 bits machine or operating system => range of integer = $-(2^{63})$ to $(2^{63} -1)$
- Kilo (K) = 10^3 (or 2^{10}) ; Mega (M) = 10^6 (or 2^{20}); Giga (G) = 10^9 (or GiB = 2^{30}); Tera (T billion) = 10^{12} (or 2^{40}) ; Peta (P) = 10^{15} (or 2^{50})
- GB in base 10 unit 10^9 ,GiB in 2 power unit (2^{30})
- FLoating Point Operation (+, -, / , *) : (10.1 + 0.1) * 1.0 / 2.0 = 5.1 => 3 FLOP
- FLOPS = FLOP per second :: 1 PetaFLOPS (kraken) = 10^{15} FLOP in one second
- FLOPS in a core = (clock rate) x (floating point operation in one clock cycle)
- Peak Rate = (FLOPS in one compute unit, core) x (no. of core)

1. Write a python code to compute C=C+A x B and plot a curve of the FLOPS against the matrix size N when the computation is done on a CPU. Do the same on the GPU.
2. 2 * (N^3) FLOP / time = ? FLOPS, 2 * 5 *5 * 5 /7.4 = 33.8 GFLOPS
3. Run the same problem again using single precision.
4. Repeat question #5 using R

```
[18] import numpy as np

     A = np.random.rand(5000, 5000).astype('float64')
     B = np.random.rand(5000, 5000).astype('float64')

     %timeit np.dot(A, B)
```

```
1 loop, best of 3: 7.39 s per loop
```

```
%%R
library(dplyr)
A<-matrix(runif(25000000),nrow=5000)
B<-matrix(runif(25000000),nrow=5000)
system.time(C<-A%*%B)
```

```
   user  system elapsed
 15.449   0.025   7.840
```

LAB 2
Problem 4

Write a python code to compute C=AXB and plot a curve of the FLOPS against the matrix size N (take N=2000, 4000, 6000) using single precision when the computation is done on a CPU. DO the same on the GPU

```
import numpy as np
import time

A = np.random.rand(2000, 2000).astype('float32')
B = np.random.rand(2000, 2000).astype('float32')
%timeit np.dot(A,B)
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm2000 = timeend - timestt
gf2000 = 2*2*2*2/tmm2000
print('time = ',tmm2000)
print('GFLOPS = ', gf2000)

A = np.random.rand(4000, 4000).astype('float32')
B = np.random.rand(4000, 4000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm4000 = timeend - timestt
gf4000 = 2*4*4*4/tmm4000
print('time = ',tmm4000)
print('GFLOPS = ', gf4000)
```

```
A = np.random.rand(6000, 6000).astype('float32')
B = np.random.rand(6000, 6000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)

import torch
A = torch.randn(6000, 6000).cuda()
B = torch.randn(6000, 6000).cuda()
timestt = time.time()
C=torch.matmul(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)
```

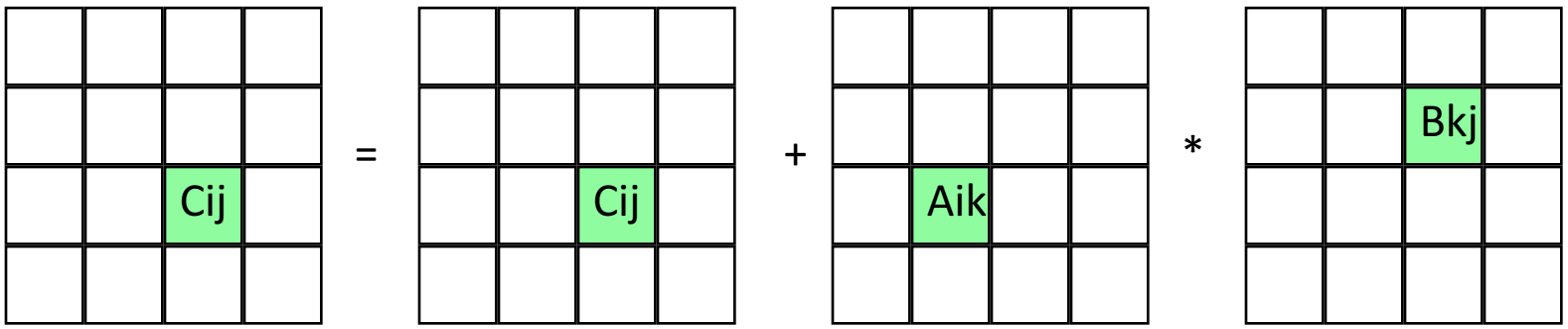
```
1 loop, best of 5: 228 ms per loop
time =  0.2360849380493164
GFLOPS =  67.7722184744277
time =  1.8578176498413086
GFLOPS =  68.8980428251037
time =  6.155170440673828
GFLOPS =  70.18489644824643
```

```
1 loop, best of 5: 214 ms per loop
time =  0.22870469093322754
GFLOPS =  69.95921218192831
time =  1.8288803100585938
GFLOPS =  69.98817762759944
time =  6.09820103645324 7
GFLOPS =  70.84056386754575
```

```
time =  6.029452800750732
GFLOPS =  71.6482928510878
time =  0.06104087829589844
GFLOPS =  7077.224510202169
```

Optimized : Block MM

- q = f/m = (2*n^3) / ((2*N + 2) * n^2) ~ n / N
- If N is equal to 1, the algorithm is ideal. However, N is bounded by the amount of fast cache memory. However, N can be taken independently to the size of matrix, n.
- The optimal value of N = sqrt (size of fast memory / 3)

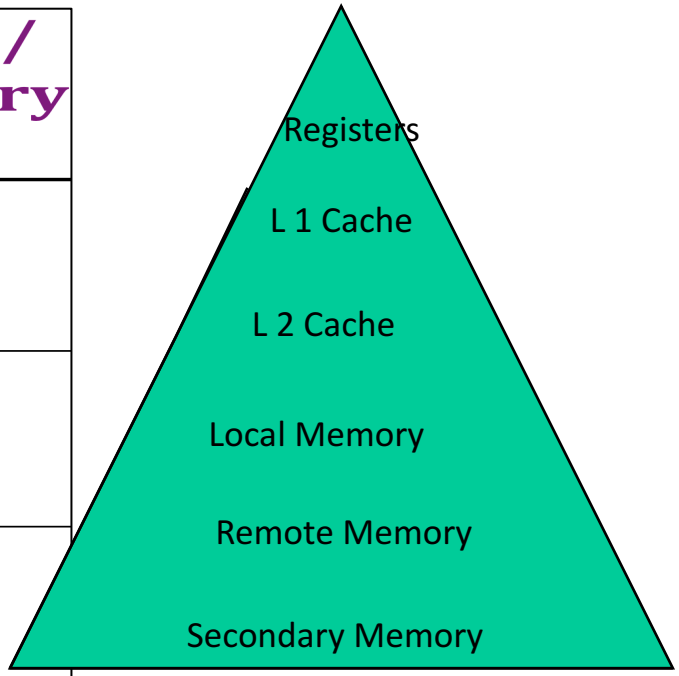


$$Cij = Cij + \sum_{k=1}^{n} Aik * Bkj$$

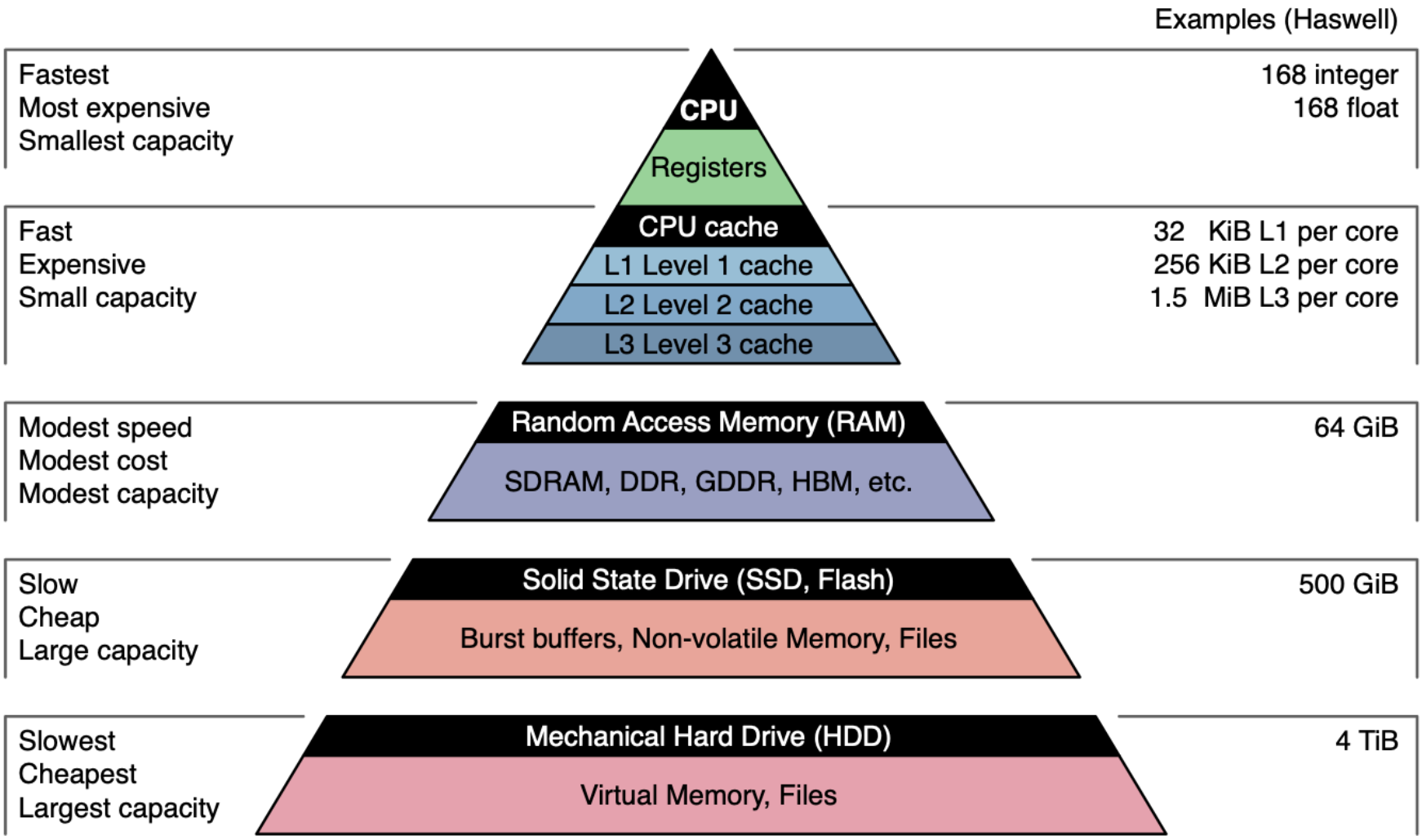
Why Higher Level BLAS?

- ✓ By taking advantage of the principle of locality:
- ✓ Present the user with as much memory as is available in the cheapest technology.
- ✓ Provide access at the speed offered by the fastest technology.
- ✓ Can only do arithmetic on data at the top of the hierarchy
- ✓ Higher level BLAS lets us do this

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y=y+\alpha x$	$3n$	$2n$	$2/3$
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	$n/2$

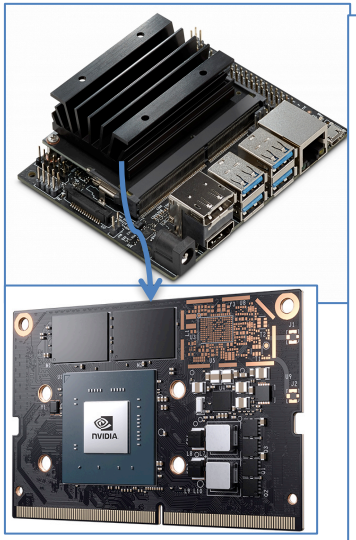


Memory hierarchy



Adapted from illustration by Ryan Leng

GPU architectures



DEVELOPER KIT	
GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	microSD (not included)
Video Encoder	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video Decoder	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30| (H.264/H.265)
Camera	1x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI 2.0 and eDP 1.4
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I²C, I²S, SPI, UART
Mechanical	100 mm x 80 mm x 29 mm

Jetson Nano Developer Kit

Nsight Systems	2019.3
Nsight Graphics	2018.7
Nsight Compute	1.0
Jetson GPIO	1.0
Jetson OS	Ubuntu 18.04
CUDA	10.0.166
cuDNN	7.3.1.28
TensorRT	5.0.6.3

Other NVIDIA GPUs used in this workshop: GTX 1650 , K80, P100, V100, A100, ..

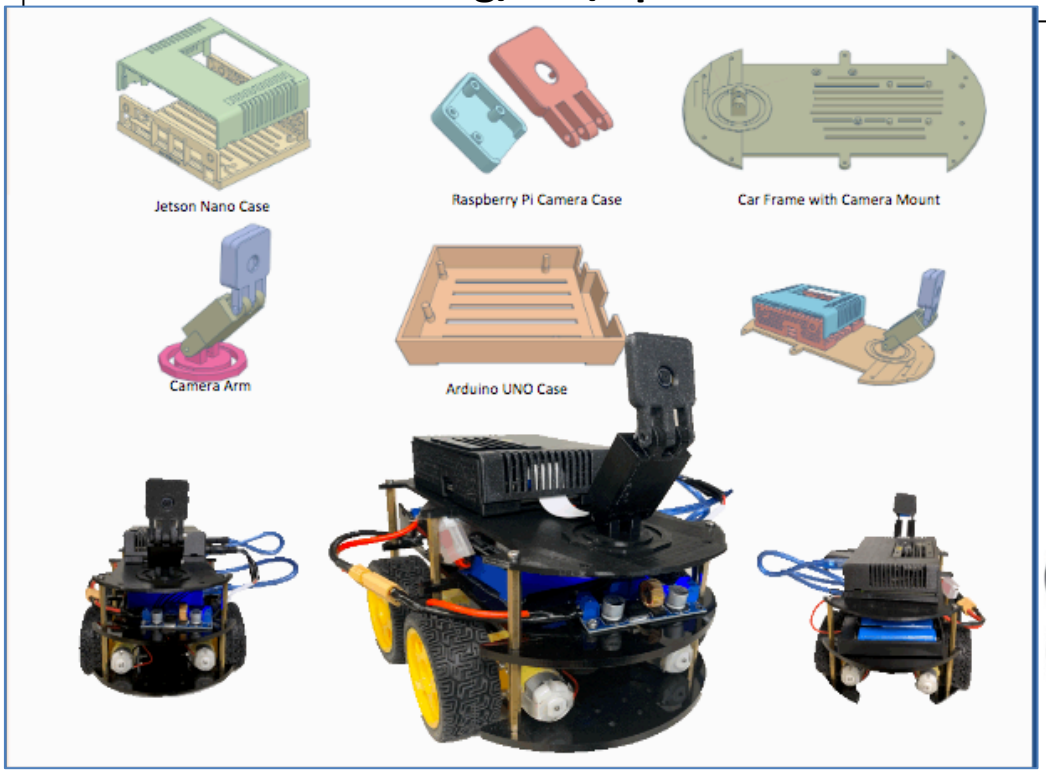
PRODUCT SPECIFICATIONS	
NVIDIA® CUDA Cores	896
Clock Speed	1485 MHz
Boost Speed	1725 MHz
Memory Speed (Gbps)	8
Memory Size	4GB GDDR5
Memory Interface	128-bit
Memory Bandwidth (Gbps)	128

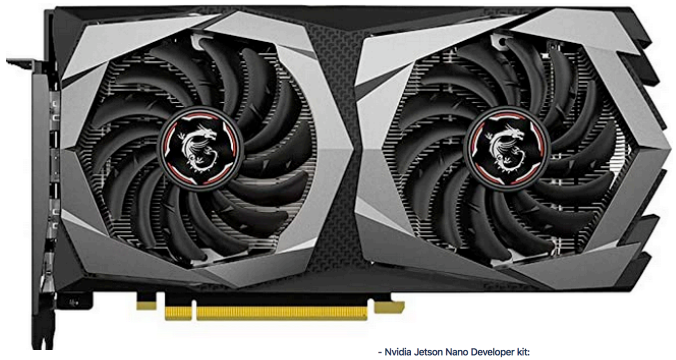
NVIDIA V100 on Summit

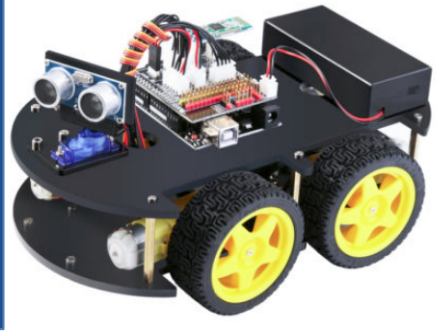
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
Memory Bandwidth	900GB/sec	

Desktop Computers and GPU

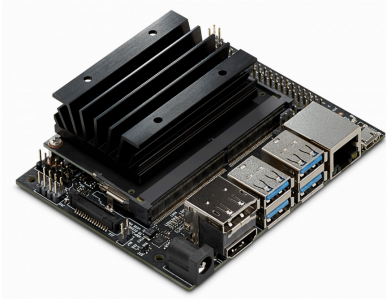
- ✓ **Computer Access, needs of each team, Desktop, laptop, Jetson Nano, Car**
- ✓ **Desktops configuration and deliveries, mid September**
- ✓ **Dell Server Box 12 cores, 48G Ram, 1TB Disk, 1650 super GPU card**
- ✓ **Ubuntu OS, anaconda individual, https://www.anaconda.com/products/individual**
- ✓ **Download, Linux, 64-Bit (x86) Installer (550 MB), python 3.8**
- ✓ **Jetson Nano card and car kit, late October**
- ✓ **www.bitbucket.org/cfdl/opendnnwheel**







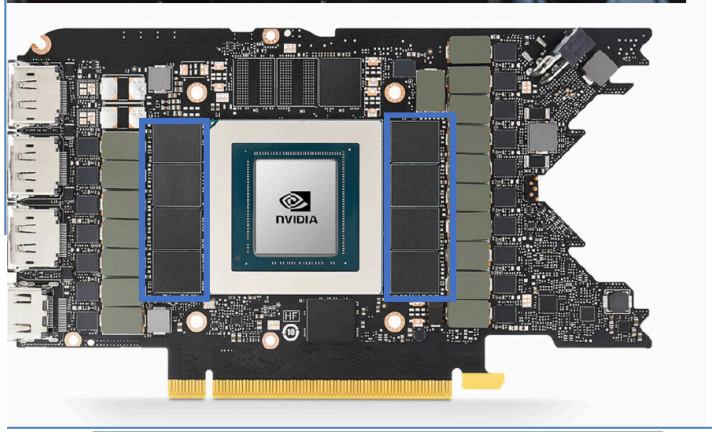
- Nvidia Jetson Nano Developer kit:



Inside a computer

CPU: "Central Processing Unit"

GPU: "Graphics Processing Unit"



CPU vs GPU

Nvidia RTX3090 GPU system

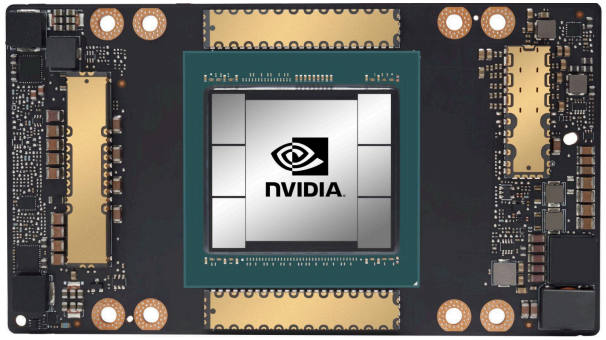
	Cores	Clock Speed (GHz)	Memory	Price	TFLOP/sec
CPU Ryzen Threadripper 3970X	64 (128 threads with hyperthreading)	3.7 (4.5 boost)	System RAM	$1999	~6.9 FP32
GPU NVIDIA RTX 3090	10496	1.4 (1.7 boost)	24 GB GDDR6X	$1499	~35.6 FP32 ~142 TFLOP with Tensor core

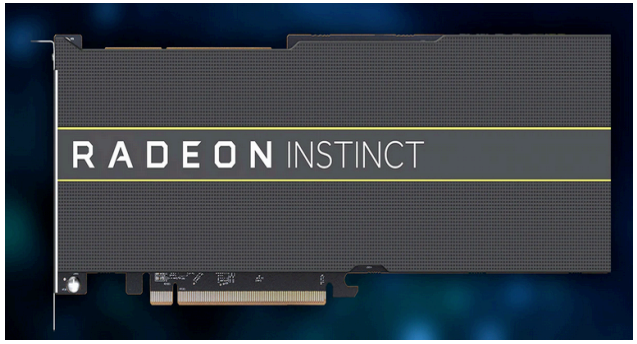
CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

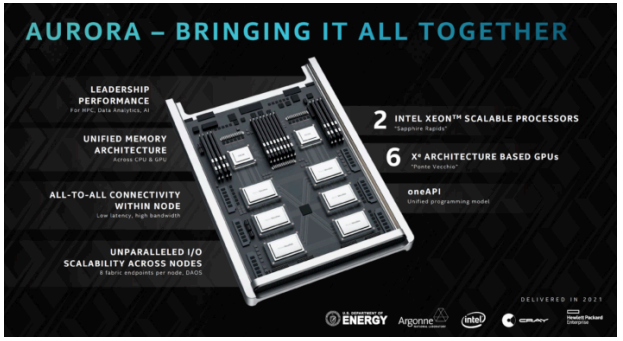
GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/schedule.html

GPU : Software Stack : Single Precision Computation







nvcc, openACC, CUDA
cuBLAS, cuDNN, NSIGHT,
grid, block, thread, warp

hipcc, openMP, HIP, rocprofiler
hipBLAS, Tensile (GEMM), ROCm
Grid, block, thread, wavefront

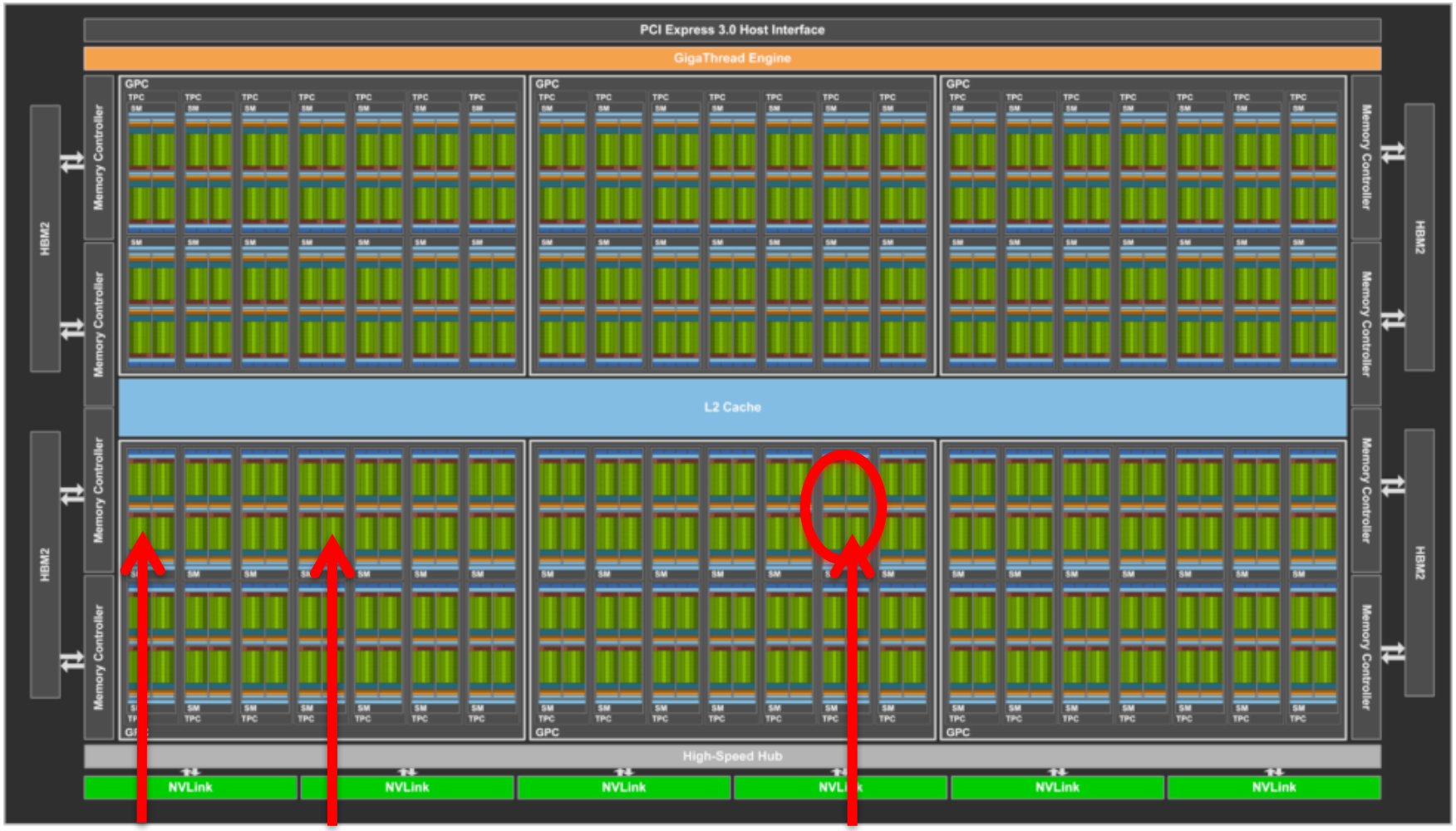
Intel, openMP,DPC++, oneAPI,
Vtune, sycl, oneMKL, , oneDNN.

	bits	precision		epsilon (2 × unit roundoff)	underflow (min normal)	overflow (max)
bfloat16	16	8 bits	≈ 2 digits	7.81×10^{-3}	1.18×10^{-38}	3.40×10^{38}
half	16	11 bits	≈ 3 digits	9.77×10^{-4}	6.10×10^{-5}	65504
single (float)	**32**	**24 bits**	**≈ 7 digits**	$\mathbf{1.19 \times 10^{-7}}$	$\mathbf{1.18 \times 10^{-38}}$	$\mathbf{3.40 \times 10^{38}}$
double	**64**	**53 bits**	**≈ 16 digits**	$\mathbf{2.22 \times 10^{-16}}$	$\mathbf{2.23 \times 10^{-308}}$	$\mathbf{1.79 \times 10^{308}}$
double-double	64x2	107 bits	≈ 32 digits	1.23×10^{-32}	2.23×10^{-308}	1.79×10^{308}
quad	128	113 bits	≈ 34 digits	1.93×10^{-34}	3.36×10^{-4932}	1.19×10^{4932}

Google TPU implements bfloat16, NVIDIA implements half, quad is double double

GPU architectures

nvidia-tesla-v100-gpu-details



Streaming Multiprocessors (SMs) **32 cores**

https://devblogs.nvidia.com/inside-volta/image3-3/

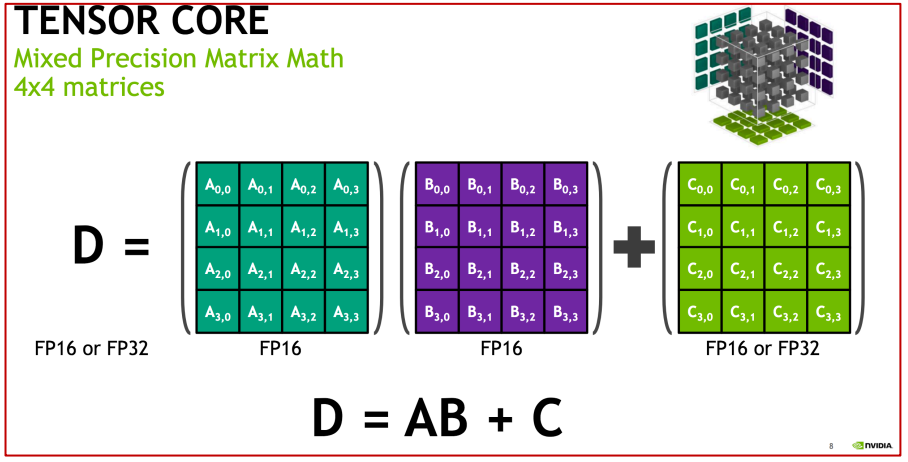
All About MM in SM V100, (A100, H100)

General Matrix-Matrix Multiplication (GEMM) operations are at the core of neural network training and inference, and are used to multiply large matrices of input data and weights in various layers. The GEMM operation computes the matrix product *D = A * B + C*, where *C* and *D* are *m*-by-*n* matrices, *A* is an *m*-by-*k* matrix, and *B* is a *k*-by-*n* matrix. The problem size of such GEMM operations running on Tensor Cores is defined by the matrix sizes, and typically denoted as *m*-by-*n*-by-*k*, (4x4x4) or (8x4x8).

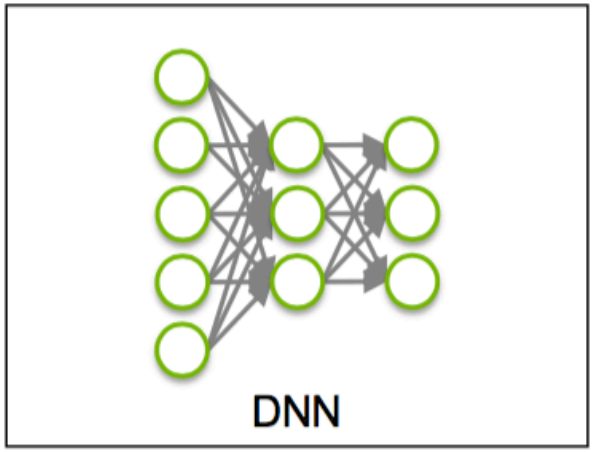
FMA = Fused Multiple-add, compute 2 floating point

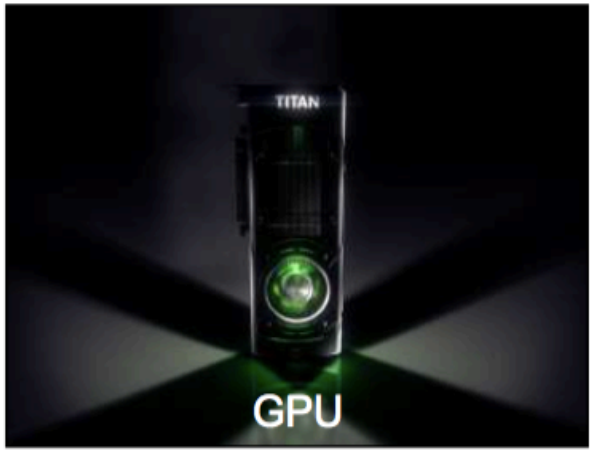
	V100	A100	A100 Sparsity[1]	A100 Speedup	A100 Speedup with Sparsity
A100 FP16 vs V100 FP16	31.4 TFLOPS	78 TFLOPS	NA	2.5x	NA
A100 FP16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 BF16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 FP32 vs V100 FP32	15.7 TFLOPS	19.5 TFLOPS	NA	1.25x	NA
A100 TF32 TC vs V100 FP32	15.7 TFLOPS	156 TFLOPS	312 TFLOPS	10x	20x
A100 FP64 vs V100 FP64	7.8 TFLOPS	9.7 TFLOPS	NA	1.25x	NA
A100 FP64 TC vs V100 FP64	7.8 TFLOPS	19.5 TFLOPS	NA	2.5x	NA
A100 INT8 TC vs V100 INT8	62 TOPS	624 TOPS	1248 TOPS	10x	20x
A100 INT4 TC	NA	1248 TOPS	2496 TOPS	NA	NA
A100 Binary TC	NA	4992 TOPS	NA	NA	NA

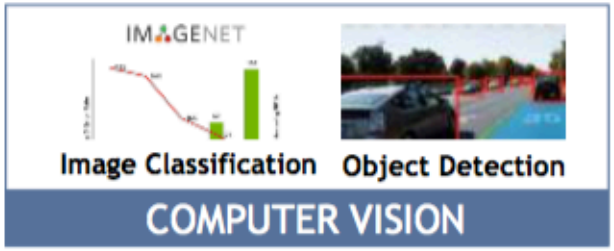
Tensor core is a new type of processing core that performs a type of specialized matrix math, suitable for deep learning and certain types of HPC. Tensor cores perform a fused multiply add, where two 4 x 4 FP16 matrices are multiplied and then the result added to a 4 x 4 FP16 or FP32 matrix. The result is a 4 x 4 FP16 or FP32 matrix; NVIDIA refers to tensor cores as performing mixed precision math, because the inputted matrices are in half precision but the product can be in full precision.

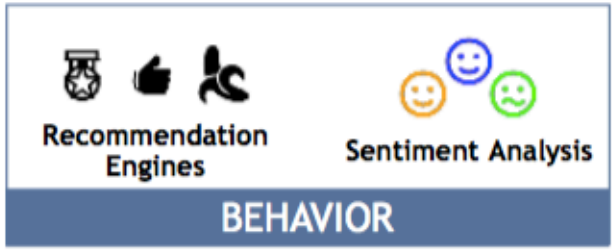


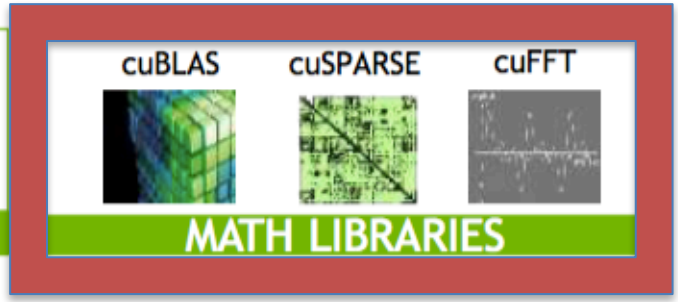
Machine Learning :LA (BLAS3) – GPU acceleration

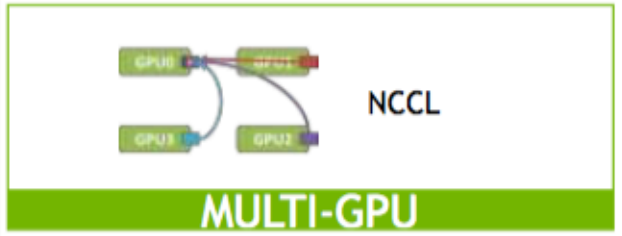












Source captured frin - Julie Bernauer – HPC Advisory Council Stanford Tutorial – 2017/02/07

Nvidia GPU Computing Applications

- ✓ A parallel computing platform that leverages NVIDIA GPUs
- ✓ Accessible through CUDA libraries, compiler directives, application programming interfaces, and extensions to several programming languages (**C/C++**, Fortran, and Python).

GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
				

Google Colab, jupyter notebook
Free Cloud Computing with GPU

" To learn data science is to do data science"

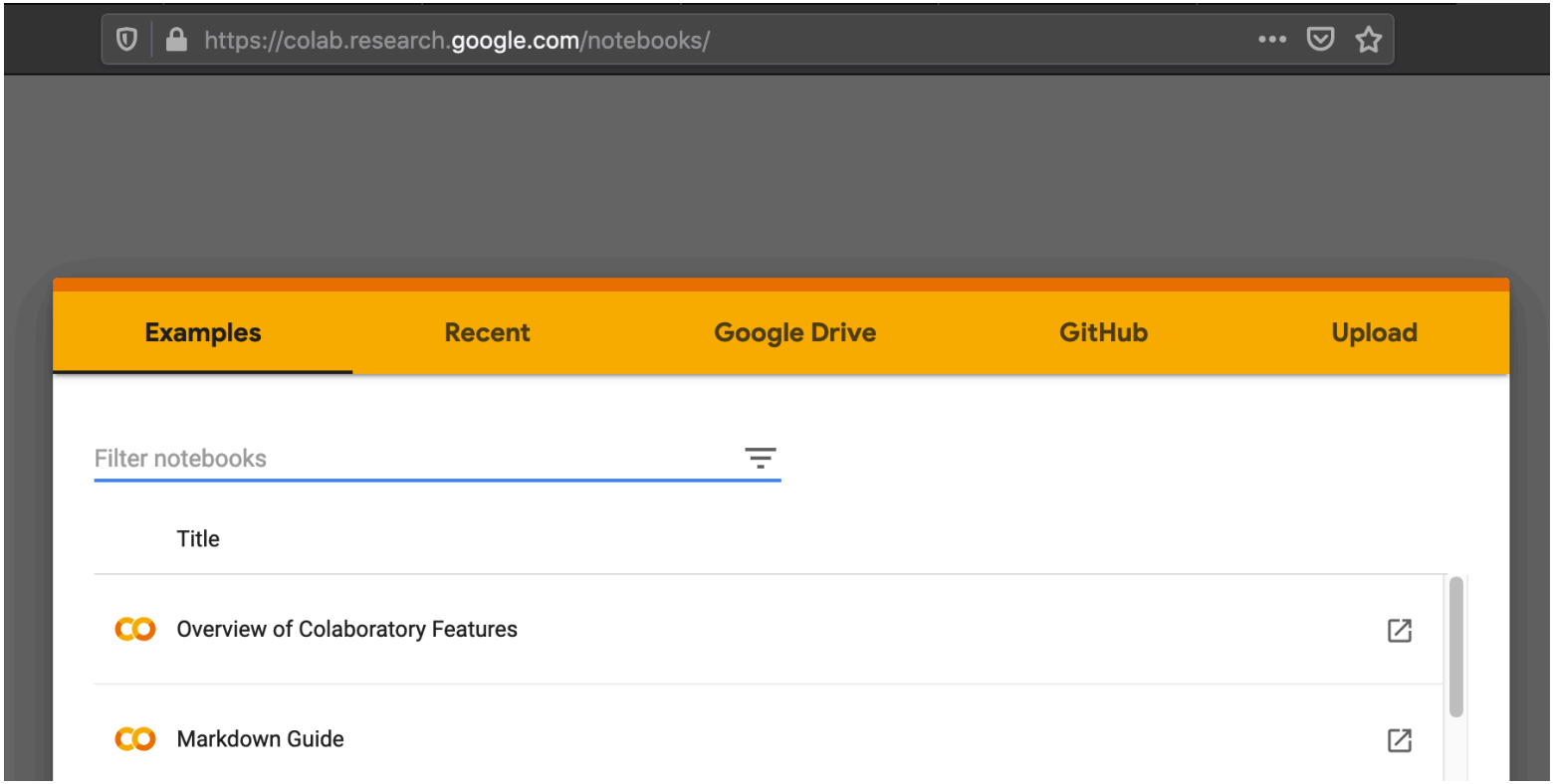
Google Colab, jupyter notebook
python, R tutorials
upload file to /content
mount your google drive

https://colab.research.google.com/notebooks/intro.ipynb
https://www.youtube.com/watch?v=inN8seMm7UI

https://github.com/dataprofessor
https://www.youtube.com/watch?v=huAWa0bqxtA
https://www.youtube.com/watch?v=Ri1MfaSlSW0

Use this link to run R code on Google COLAB
https://colab.research.google.com/notebook#create=true&language=r

https://colab.research.google.com/notebooks/
https://colab.research.google.com/notebook#create=true&language=python



- ✓ **COLAB Introduction http://www.youtube.com/watch?v=vVe648dJOdI**
- ✓ **Google COLAB : https://www.youtube.com/watch?v=inN8seMm7UI**

1. Write a python code to compute C=A x B and plot a curve of the FLOPS against the matrix size N when the computation is done on a CPU. Do the same on the GPU.
2. Repeat question #5 using R

```
[18] import numpy as np

     A = np.random.rand(5000, 5000).astype('float64')
     B = np.random.rand(5000, 5000).astype('float64')

     %timeit np.dot(A, B)
```

```
1 loop, best of 3: 7.39 s per loop
```

```
%%R
library(dplyr)
A<-matrix(runif(25000000),nrow=5000)
B<-matrix(runif(25000000),nrow=5000)
system.time(C<-A%*%B)
```

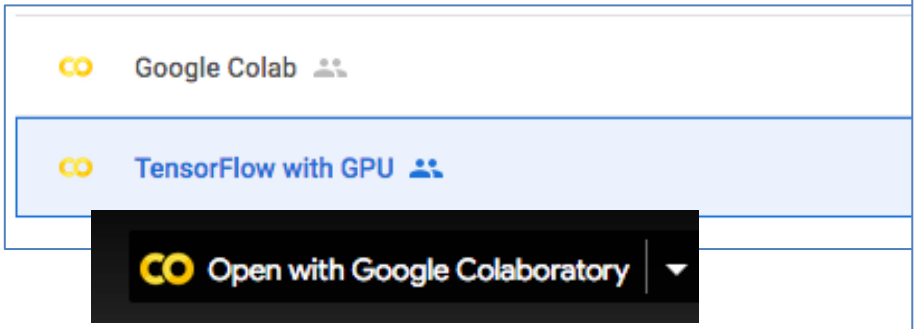
```
   user  system elapsed
 15.449   0.025   7.840
```

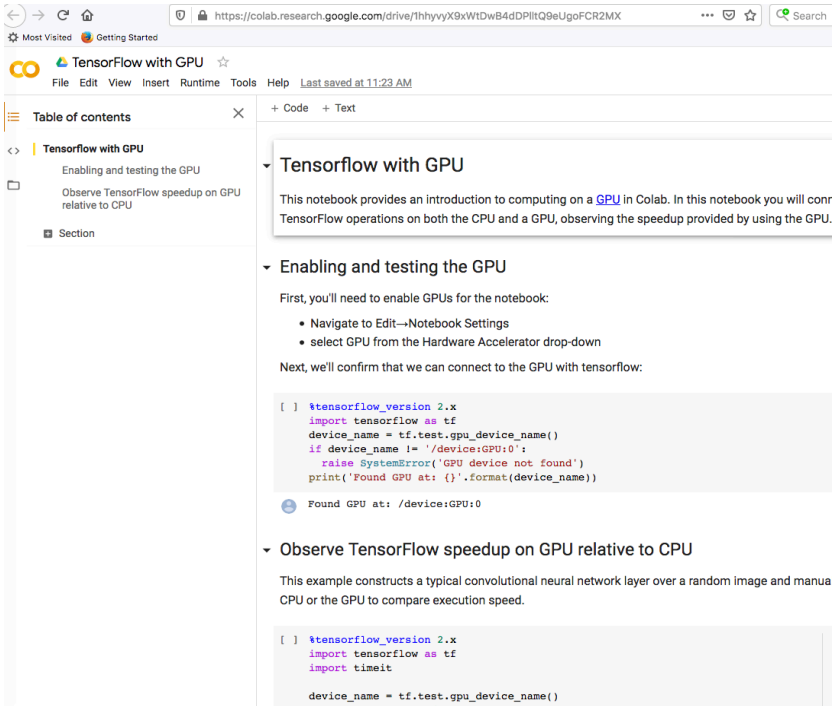
Google Colab, jupyter notebook

Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

✓ **colab.research.google.com,**






```
%tensorflow_version 2.x
import tensorflow as tf
import timeit

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  print(
      '\n\nThis error most likely means that this notebook is not '
      'configured to use a GPU.  Change this in Notebook Settings via the '
      'command palette (cmd/ctrl-shift-P) or the Edit menu.\n\n')
  raise SystemError('GPU device not found')

def cpu():
  with tf.device('/cpu:0'):
    random_image_cpu = tf.random.normal((100, 100, 100, 3))
    net_cpu = tf.keras.layers.Conv2D(32, 7)(random_image_cpu)
    return tf.math.reduce_sum(net_cpu)

def gpu():
  with tf.device('/device:GPU:0'):
    random_image_gpu = tf.random.normal((100, 100, 100, 3))
    net_gpu = tf.keras.layers.Conv2D(32, 7)(random_image_gpu)
    return tf.math.reduce_sum(net_gpu)

# We run each op once to warm up; see: https://stackoverflow.com/a/45067900
cpu()
gpu()

# Run the op several times.
print('Time (s) to convolve 32x7x7x3 filter over random 100x100x100x3 images '
      '(batch x height x width x channel). Sum of ten runs.')
print('CPU (s):')
cpu_time = timeit.timeit('cpu()', number=10, setup="from __main__ import cpu")
print(cpu_time)
print('GPU (s):')
gpu_time = timeit.timeit('gpu()', number=10, setup="from __main__ import gpu")
print(gpu_time)
print('GPU speedup over CPU: {}x'.format(int(cpu_time/gpu_time)))
```

```
Time (s) to convolve 32x7x7x3 filter over random 100x100x100x3 images (batch x h
CPU (s):
3.862475891000031
GPU (s):
0.10837535100017703
GPU speedup over CPU: 35x
```

✓ **COLAB Introduction http://www.youtube.com/watch?v=vVe648dJOdI**

✓ **Google COLAB : https://www.youtube.com/watch?v=inN8seMm7UI**

Python Basic

```
# This is a comment statement
import math
A=10
B=10
C= A*B
print (“ C”)
D = math.sqrt(C)
print (“D”)
```

```
#Timing Example
import time
time.time()
t0=time.time()
Let's test your typing speed!
t1=time.time()
t1-t0
```

Reference – what the exact description of how this function work

Python Basic – Control Flow, Timing

A. if, elif and else

```
if … :
    …
elif …:
    …
else:
    …
```

Example

```
a=5
b=7
c=8
d=4
if a<b and c>d:
    print('correct')
```

B. for loop : for i in range(N)

The command will loop from $i = 0$ to $i = N$-1.

range(N) is a sequence from 0, 1, 2, …, to N-1.

Example

```
total = 0
for i in range(4):
    total = total + i
    print(i)
print(total)
```

Example

```
for i in range(4):
    for j in range(4):
        if j>i:
            break
        print((i,j))
```

C. while loop , Example

```
er=10
count = 0
while er > 1:
    er = er - 0.5
    count = count + 1
print(er)
print(count)
```

Python Basic - List

In Python, a list is a sequence. It is enclosed in a pair of squared brackets '[]'. For example,
a = [1,2,3]

Elements are numbered from '0'. To call for a particular element, we give the index in square bracket:
a[0] = 1
a[1] = 2
a[2] = 3

To add an element, we use the 'append' function

a.append(4), which gives
a = [1,2,3,4]

To insert an element at a specific location, we use the 'insert' function

a.insert(1,4), which gives

a = [1,4,2,3,4]

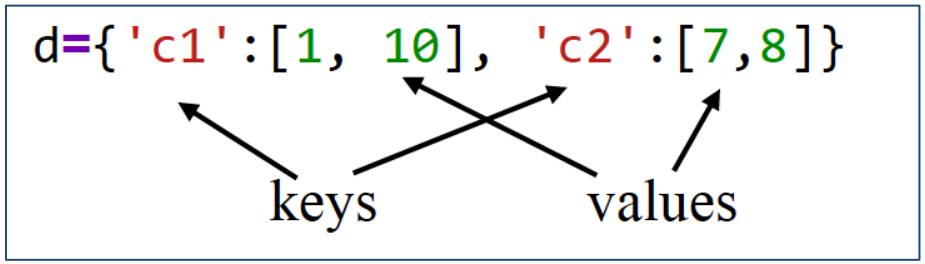
That is, the entry '4' is added into the location with index '1', in other words, the second entry.

Datatype can be different in a list and it is not necessarily numbers.
If we have two lists, we can easily concatenate (combine) the two lists by the '+' sign. Note that this is not numerical addition.

```
In [8]: a
Out[8]: [1, 4, 2, 3, 4]

In [9]: b=[1,3,5,7,9]

In [10]: a+b
Out[10]: [1, 4, 2, 3, 4, 1, 3, 5, 7, 9]
```

Python Basic - Dictionary

Dictionary is an associative array with key and value. To define a dictionary, we enclose the content with a pair of curly brackets '{}'. The names of the keys are enclosed in ''. We can get the values under a particular key by dict['key']




```
In [12]: d['c1']
Out[12]: [1, 10]

In [13]: d['c2']
Out[13]: [7, 8]
```

The keys and values in a dictionary can be obtained by listing them out.

```
In [14]: list(d.keys())
Out[14]: ['c1', 'c2']

In [15]: list(d.values())
Out[15]: [[1, 10], [7, 8]]
```

Entries can easily be added by defining a new key: d['c3']=['Hello','World']

Then, the dictionary is updated to : d={'c1': [1, 10], 'c2': [7, 8], 'c3': ['Hello', 'World']}

On the other hand, we delete entries by 'del' : **del d['c1']**
d={'c2': [7, 8], 'c3': ['Hello', 'World']}

Python – Numpy, Arrays – LA - BLAS

NumPy stands for Numerical Python. It is the package for numerical computation in Python. We will learn about : Arrays, Matrix Operations, Universal Functions, Random number, Statistics, etc. To use the NumPy package, we first import it:

```
import numpy as np
```

We define an array by specifying that it is an NumPy array

```
a=np.array([1,2,3])
```

As in C++ , the indexing starts from 0. That is,
a[0] = 1, a[1]=2 and a[2]=3. a[3] is undefined.

Similarly, we can define a matrix by adding '[]'. For example, to define the following matrix

$$b = \begin{bmatrix} 1 & 0 & -5, \\ -2.5. & 7.3 & 4.6 \end{bmatrix}$$; in Python, we write :

```
b=np.array( [  [1,0,-5] , [-2.5, 7.3, 4.6]  ] )
```

Indexing starts from the outer brackets: b[0,2] = –5, b[1,1] = 7.3.

Python Numpy – Dimension

The dimension or size of an array is given by the 'shape' function.
To get the number of rows in b, we ask for the first entry in its shape

b.shape[0].

Similarly, the number of columns in b is given by

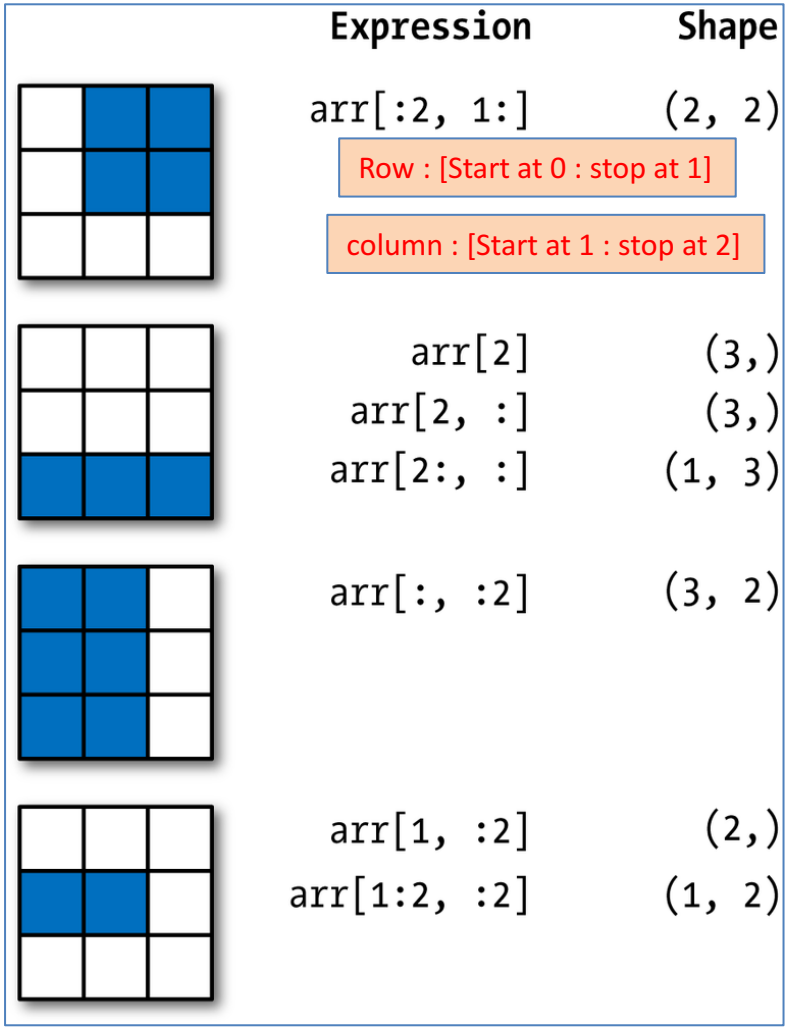
b.shape[1]

```
In [6]: a.shape

Out[6]: (3,)

In [17]: b.shape

Out[17]: (2, 3)
```

Python Numpy – Indexing and Slicing, Example

[start at this index : end before this index]

```
In [10]: b

Out[10]: array([[ 1. ,  0. , -5. ],
                [-2.5,  7.3,  4.6]])

In [8]: b[1:,:2]

Out[8]: array([[-2.5,  7.3]])

In [9]: b[:2,1:]

Out[9]: array([[ 0. , -5. ],
               [ 7.3,  4.6]])

In [11]: b[:,1:]

Out[11]: array([[ 0. , -5. ],
                [ 7.3,  4.6]])
```



Form : Shape of an array; running indices, each index represents a dimension

Extraction and description of an array - functions to get to what you need!!

Numpy – Reshape

'reshape' function allows us to change the dimension of an array. A 1-dimensional array can be reshaped to a 2-dimensiaonl matrix. The reshape function is useful when we deal with image data. An image is usually described as a matrix (2d). By reshape, we can turn it into a row vector. We can then stack all image data into one large matrix. A lazy way to specify the dimension is '-1' when the dimension is inferred from the data.

Example

```
np.arange(15).reshape(3,5)
```

gives

```
array( [ [ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]  ] )

M=np.array ([[1,2],[11,12]] )
M.reshape(1,4) or M.reshape(1,-1)
```

gives

```
array( [[ 1, 2, 11, 12]] )
```

Numpy – Arithmetic

Basics operations are *elementwise*

+ addition
- subtraction
* multiplication
/ division
** power

For example, to take the square of every element in an array u:

```
u**2
```

The square root can be computed in two ways

```
v**0.5
```

or

```
np.sqrt(v)
```

Note that, you can sum over all entries in an array by

```
np.sum(v)
```

To get the real part of an array, we have

```
v.real
```

Empty Array

'empty' creates an array without initializing its values. : np.empty((3,2))

Zero Array

We can define a zero matrix with all entries equal to 0, : d=np.zeros((2,4))

Datatype

The datatype of an array is given by

```
b.dtype
```

The datatype of an array can be converted to another type

```
b.astype(np.int)
```

Say for example, we have an array of true or false, we wish to change the values to 0(false) and 1(true). Example

```
v_boolean=np.array([True,False,False])
v_int = v_boolean.astype(np.int)
v_intequals
array([1, 0, 0])
```

Matrix Operations

A. Matrix Multiplication

To multiply two matrices *x* and *y*, we either use the 'dot' function

```
np.dot(x,y)
```

or simply '@'

```
x @ y
```

B. Transpose :

To compute the transpose of a matrix M,

```
M.T
```

C. Norm

More functions related to linear algebra can be obtained from the 'linalg' library in NumPy. For instance, we compute the norm of a vector by the norm function. Various types of norm can be computed. If we do not specify the choice of norm, the program return the default Euclidean *l*2-norm.

```
from numpy.linalg import norm
norm(v)
```

Universal Functions

Common functions such as the exponential function, logarithmic function, etc. can be computed by the usual command.

```
np.exp(x)
np.log(x)
np.abs(x)
np.sign(x)
np.sqrt(x)
```

Random Number

A random number can be generated by

```
np.random.rand()
```

The above command yields a number draw from a uniform distribution.

```
np.random.randn(5)
```

The above command yields an array of length 5 where each entry is drawn from a normal distribution with mean 0 and standard deviation 1.

Basic Statistics : Mean, Standard Deviation, maximum

Suppose that we have a 2 by 3 matrix,

```
A=np.array( [[1,0,2],[3,5,6]] )
```

the mean of all the entries is given by

```
np.mean(A)
```

or simply

```
A.mean()
```

We can compute the mean along rows or columns by specifying the axis. By specifying 'axis=i', summation is taken w.r.t the *i*th-index. In the above example,

```
A.mean(axis=0)
```

yields the means of the three columns,
i.e. `array([2. , 2.5, 4. ])`
whereas

```
A.mean(axis=1)
```

gives the means of the two rows
i.e. `array([1., 4.66666667])`

Similarly, we can compute the standard deviation and maximum

```
A.std()
A.max()
```

Python DataFrame, Pandas

Pandas is a library in Python for handling data described in tabular form. While there are many functions in pandas, we will just give a brief introduction here. In this chapter, we will learn about DataFrame, importing data, importing data from libraries

```
import pandas as pd
```

DataFrame is a table of data. The rows are called 'index' and columns are simply referred to as 'columns'.

```
from pandas import DataFrame
```

To define a DataFrame, we need to define three objects: values, index, columns; Example

```
df =
pd.DataFrame(np.arange(16).reshape(4,4),
index=[1,2,3,4],columns=['a','b','c','d'])
```

	a	b	c	d
1	0	1	2	3
2	4	5	6	7
3	8	9	10	11
4	12	13	14	15

Python DataFrame, Pandas

```
df = pd.DataFrame(index=[0,1,2,3],columns=['Distance','Age','Area','Price'])
```

A dictionary stores data under specific keys. We can easily define a DataFrame from a dictionary where keys are turned into columns. Suppose that we define the following dictionary:

```
data = {'Distance':[10,0.5,2,6.2],'Age':[5,40,30,10],
'Area':[100,200,500,150],'Price':[20,15,75,34]}
```

A DataFrame can be defined from the above dictionary as a well-organized and easy to read table

```
df = pd.DataFrame(data)
```

	Distance	Age	Area	Price
0	10.0	5	100	20
1	0.5	40	200	15
2	2.0	30	500	75
3	6.2	10	150	34

Pandas : Importing Data From Files

In application, the data is usually given in some files of standard format, say, excel, csv, etc. The data can be imported into Python through the 'read' function in pandas.

```
df = pd.read_csv('filename.csv',sep=';')
df = pd.read_csv('filename.csv',delimiter='\t')
```

Notice that the file should be saved in the same folder as the Jupyter notebook. The regular expression for separating the data is specified in 'sep' or 'delimiter':

',' ';' '\t' tab

Example (wine)

```
df = pd.read_csv('winequality-red.csv',sep=';')
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000

Let's take a look into the 'iris' dataset. The dataset is a classic classification problem where we determine the species of a given iris based on some physical parameters. We first import the data set.

```
from sklearn.datasets import load_iris

iris_dataset = load_iris()
```

By checking the keys in the data set, we understand the major components in the data set.

```
iris_dataset.keys()
```

gives

```
dict_keys(['data', 'target', 'target_names',
'DESCR', 'feature_names', 'filename'])
```

where

'DESCR' gives the description of the dataset,

'target_names' gives the names of the species,

'target' stores the target values (label the species in terms of 0,1,2) of all the given data,

1. Write a python code to compute C=C+A x B and plot a curve of the FLOPS against the matrix size N when the computation is done on a CPU. Do the same on the GPU.
2. 2 * (N^3) FLOP / time = ? FLOPS, 2 * 5 *5 * 5 /7.4 = 33.8 GFLOPS
3. Run the same problem again using single precision.
4. Repeat question #5 using R

```
[18] import numpy as np

     A = np.random.rand(5000, 5000).astype('float64')
     B = np.random.rand(5000, 5000).astype('float64')

     %timeit np.dot(A, B)
```

```
1 loop, best of 3: 7.39 s per loop
```

```
%%R
library(dplyr)
A<-matrix(runif(25000000),nrow=5000)
B<-matrix(runif(25000000),nrow=5000)
system.time(C<-A%*%B)
```

```
   user  system elapsed
 15.449   0.025   7.840
```

LAB 2
Problem 4

Write a python code to compute C=AXB and plot a curve of the FLOPS against the matrix size N (take N=2000, 4000, 6000) using single precision when the computation is done on a CPU. DO the same on the GPU

```
import numpy as np
import time

A = np.random.rand(2000, 2000).astype('float32')
B = np.random.rand(2000, 2000).astype('float32')
%timeit np.dot(A,B)
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm2000 = timeend - timestt
gf2000 = 2*2*2*2/tmm2000
print('time = ',tmm2000)
print('GFLOPS = ', gf2000)

A = np.random.rand(4000, 4000).astype('float32')
B = np.random.rand(4000, 4000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm4000 = timeend - timestt
gf4000 = 2*4*4*4/tmm4000
print('time = ',tmm4000)
print('GFLOPS = ', gf4000)
```

```
A = np.random.rand(6000, 6000).astype('float32')
B = np.random.rand(6000, 6000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)

import torch
A = torch.randn(6000, 6000).cuda()
B = torch.randn(6000, 6000).cuda()
timestt = time.time()
C=torch.matmul(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)
```

```
1 loop, best of 5: 228 ms per loop
time =  0.2360849380493164
GFLOPS =  67.7722184744277
time =  1.8578176498413086
GFLOPS =  68.8980428251037
time =  6.155170440673828
GFLOPS =  70.18489644824643
```

```
1 loop, best of 5: 214 ms per loop
time =  0.22870469093322754
GFLOPS =  69.95921218192831
time =  1.8288803100585938
GFLOPS =  69.98817762759944
time =  6.09820103645324 7
GFLOPS =  70.84056386754575
```

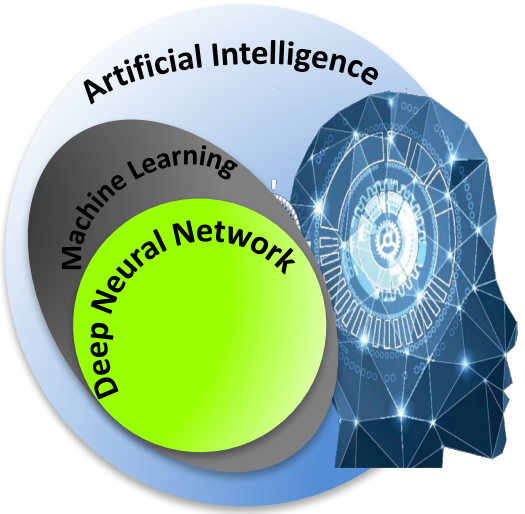
```
time =  6.029452800750732
GFLOPS =  71.6482928510878
time =  0.06104087829589844
GFLOPS =  7077.224510202169
```

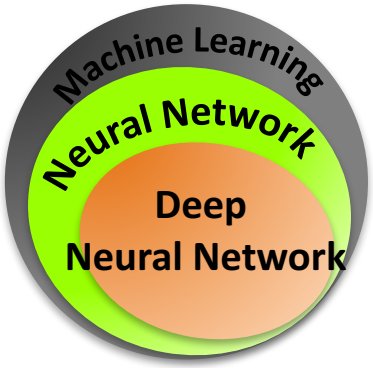
Computational Ecosystem

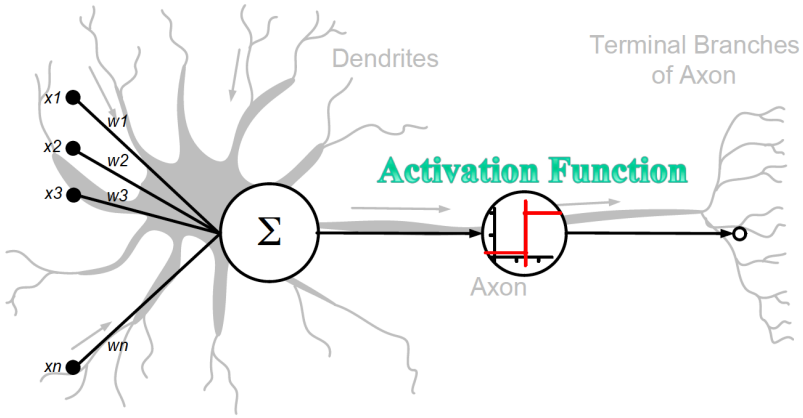
- **Advance Computing system**
  - **Top500, FLOPS and Performance**
  - **DOE Exascale Road Map**
  - **NSF infrastructure**
  - **Desktop, Accelerators, GPUs**
  - **Google COLAB**
- **Predictive Simulation Science**
  - **Equation based simulation sciences**
  - **Mathematics Essentials, Calculus**
  - **Linear Algebra, BLAS**

- **Computer Sciences and Software tools**
  - **Linux OS**
  - **Computational thinking**
  - **Workflow**
  - **Languages**
- **Data Intensive Sciences**
  - **Statistical Learning**
  - **Data mining**
  - **Deep Learning**
  - **Framework**

AI, ML, NN, DNN







- ✓ **Artificial Intelligence (AI)**: science and engineering of making intelligent machines to perform the human tasks (John McCarthy,1956). AI applications is ubiquitous.
- ✓ **Machine learning (ML) :** A field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel, 1959). A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E (Tom Mitchell,1998).
- ✓ **Neural Network (NN) :** Neural Network modeling, a subfield of **ML** is algorithm inspired by structure and functions of biological neural nets
- ✓ **Deep Neural Network (DNN) :** (aka deep learning): an extension of **NN** composed of many layers of functional neurons, is dominating the science of modern **AI** applications
- ✓ **Supervised Learning (SL)** : A class in ML, dataset has labeled values, use to predict output values associated with new input values.

Supervised Learning

Supervised Learning Data:
(x, y) x is data, y is label

Goal:
Learn a function to map x -> y

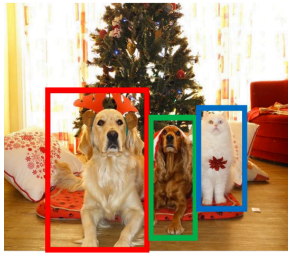
Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

CAT

Classification

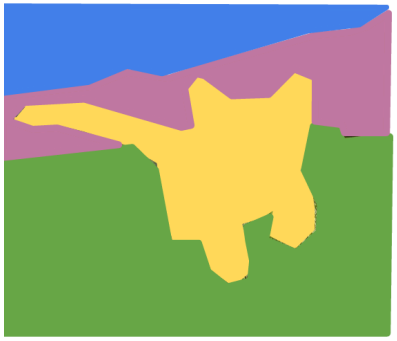
A cat sitting on a suitcase on the floor

Image captioning



DOG, DOG, CAT

Object Detection



GRASS, CAT, TREE, SKY

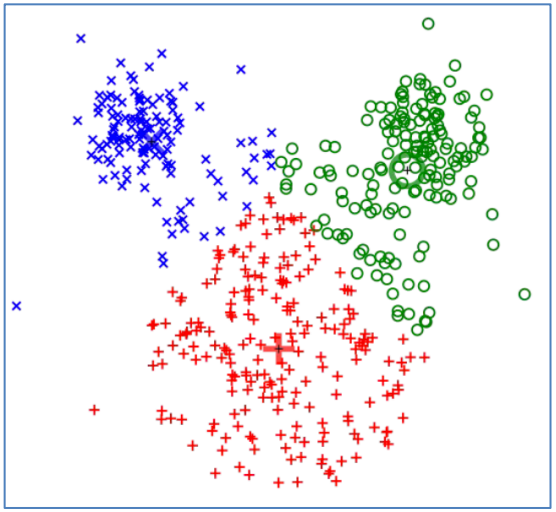
Semantic Segmentation

Unsupervised Learning

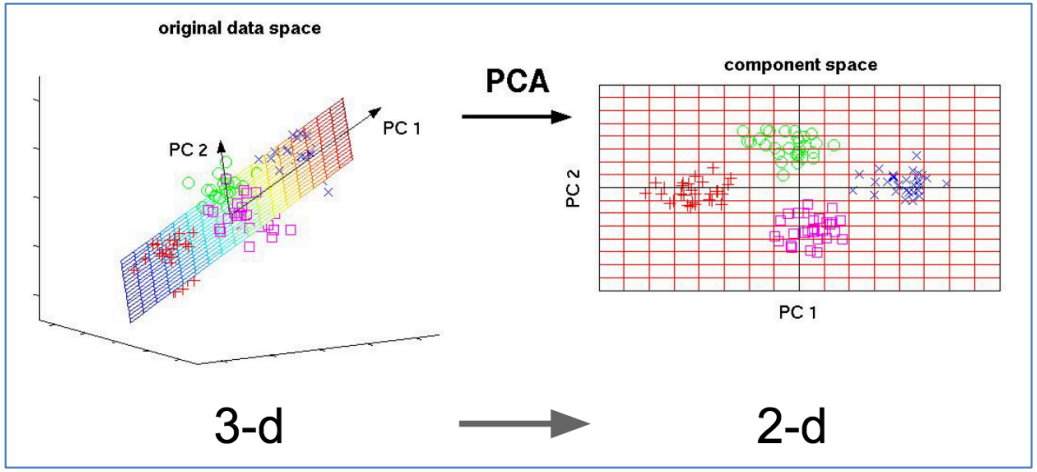
Unsupervised Learning Data:
x Just data, no labels!

Goal:
Learn some underlying hidden structure of the data

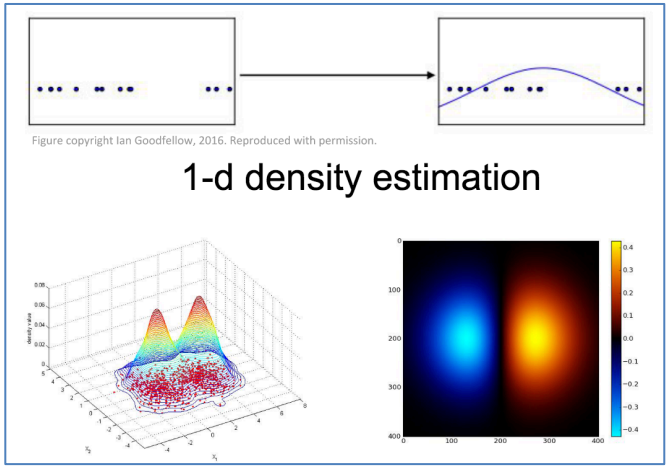
Examples: Clustering, dimensionality reduction, density estimation, etc.



K-means clustering

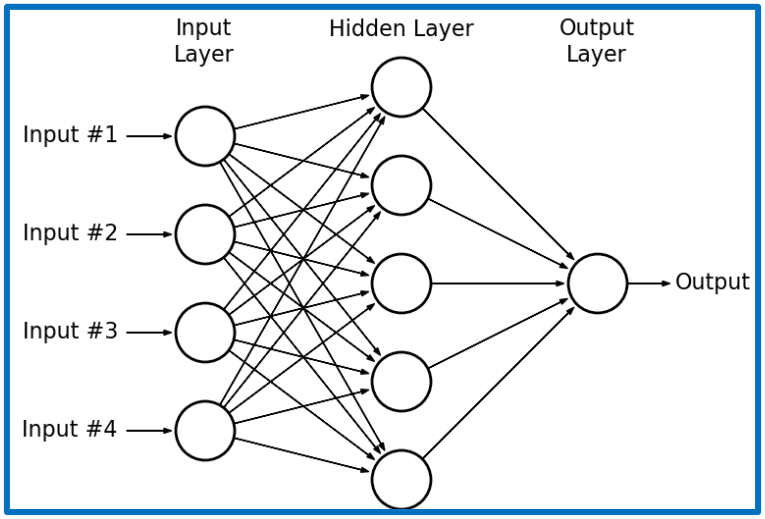


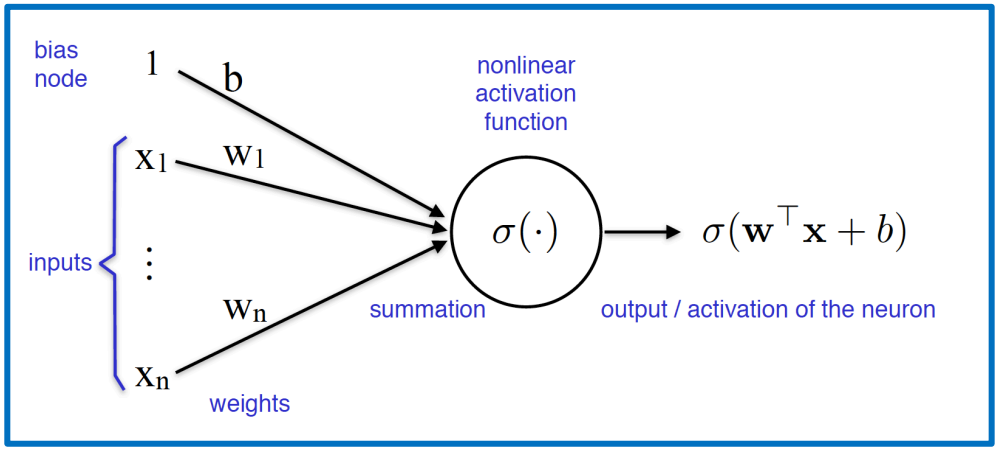
Principal Component Analysis
(Dimensionality reduction)



2-d density estimation

NN Modeling

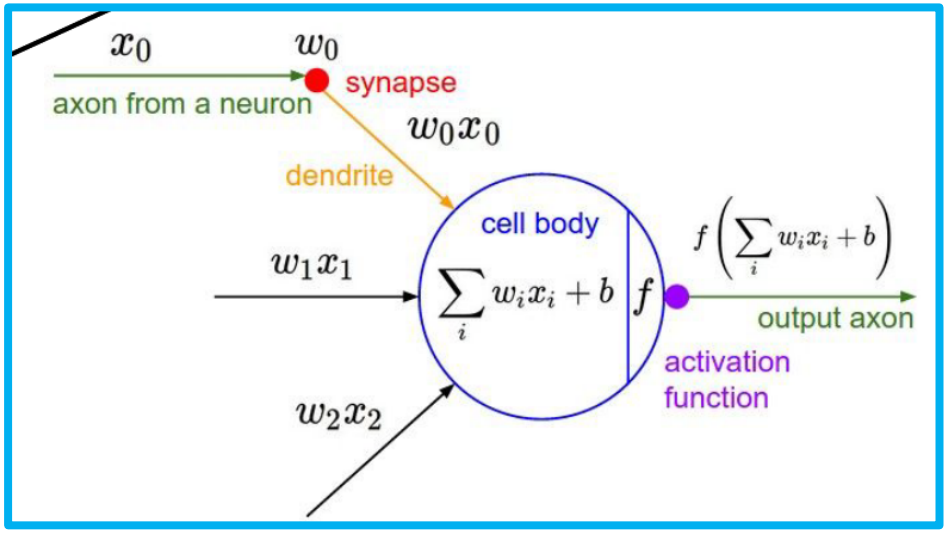




- ✓ A node in the neural network is a mathematical function or activation function which maps input to output values.
- ✓ Inputs represent a set of vectors containing weights (w) and bias (b). They are the sets of parameters to be determined.
- ✓ Many nodes form a **neural layer**, **links** connect layers together, defining a NN model.
- ✓ Activation function (f or σ), is generally a nonlinear data operator which facilitates identification of complex features.

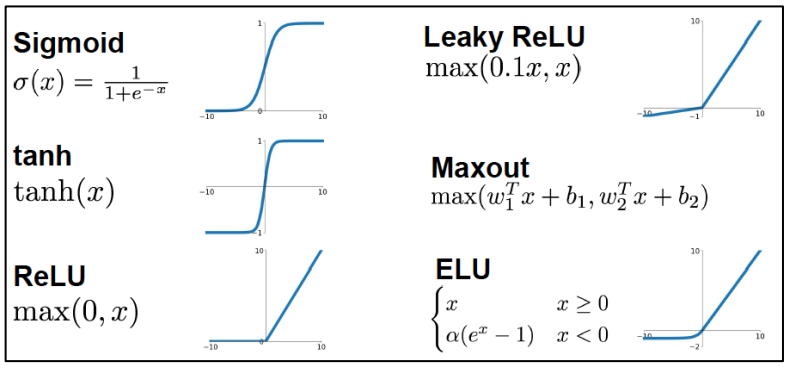
Perceptron

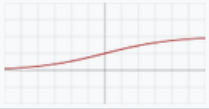
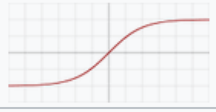
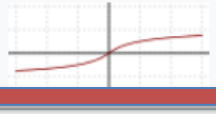
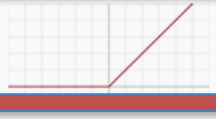
Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt. A perceptron takes several binary inputs, *x*1,*x*2,…, and produces a single binary output. By varying the weights and the threshold, we can get different models of decision-making.



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

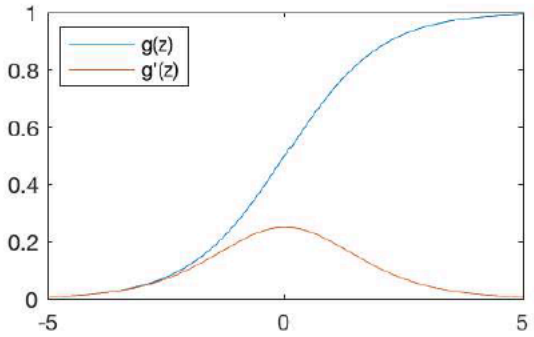
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$



Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign [7][8]		$f(x) = \frac{x}{1+\lvert x\rvert}$
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

Nonlinear Activation Function

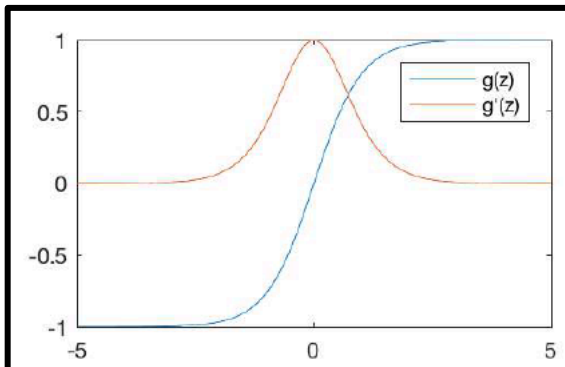
Sigmoid Function



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

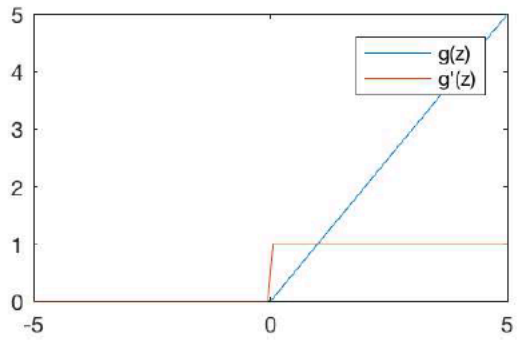
tf.nn.sigmoid(z)



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

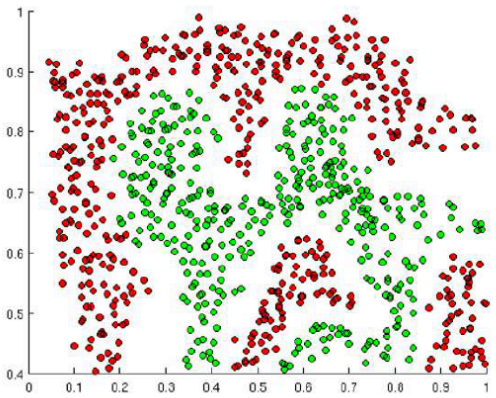
tf.nn.tanh(z)

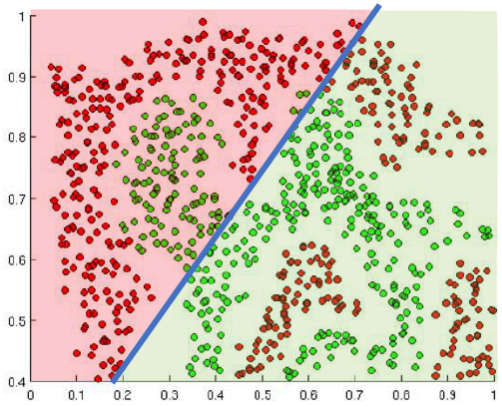


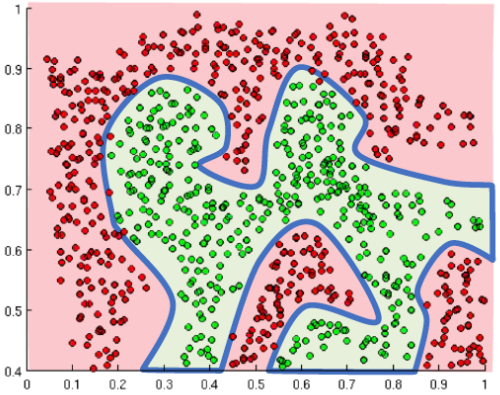
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

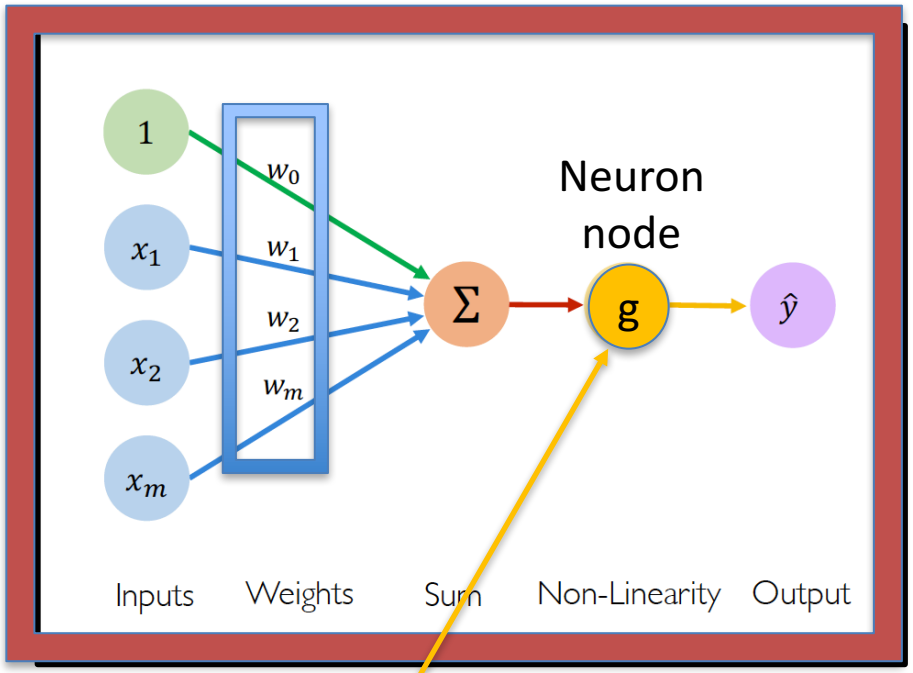
tf.nn.relu(z)







DNN MLP Forward Steps



$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

Output

Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

Non-linear activation function

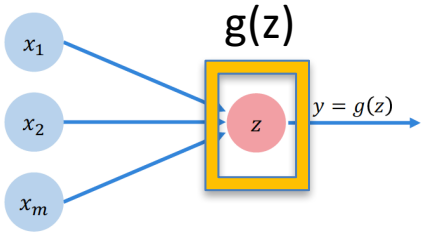
Bias

$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

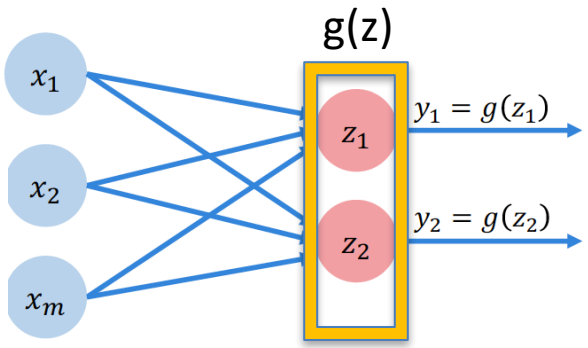
- Example: sigmoid function

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

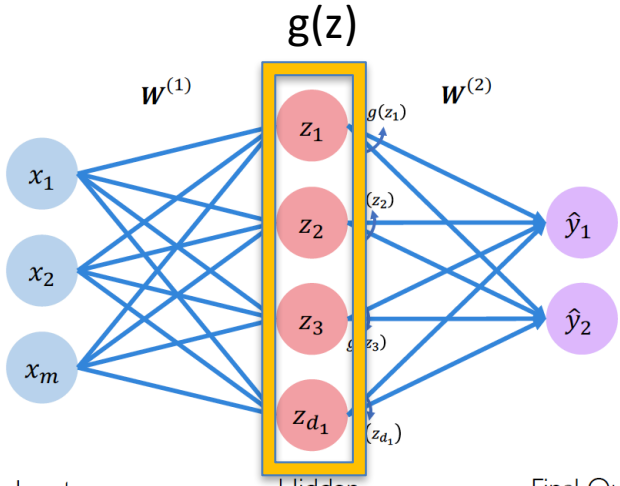
Multilayer Perceptron : Forward Path Calculation



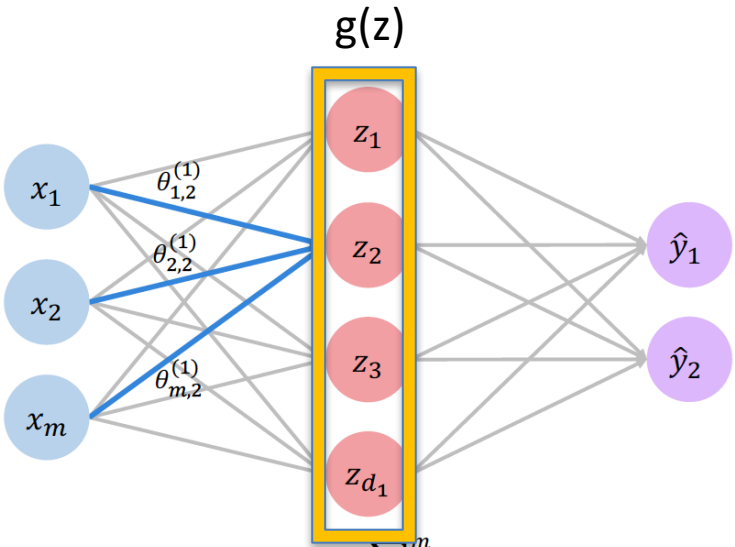
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$



$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

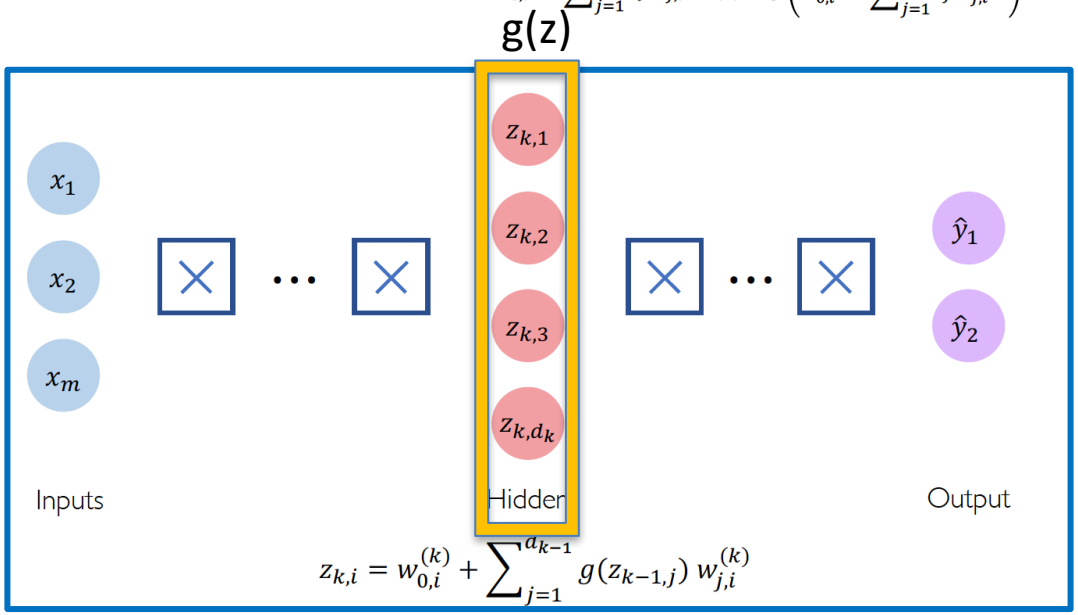


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$



$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j w_{j,2}^{(1)}$$

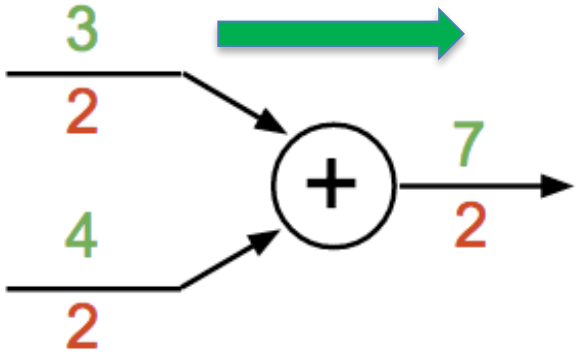
$$= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)}$$



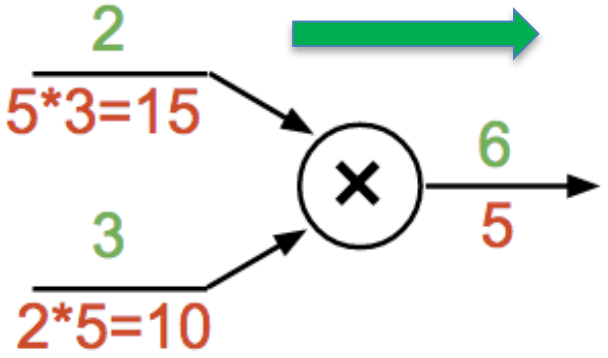
$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Forward Computing Operators

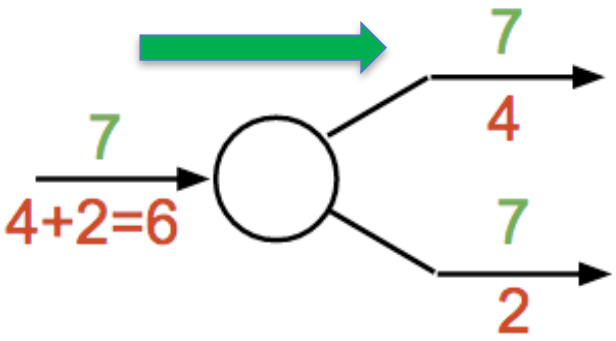
add gate: gradient distributor



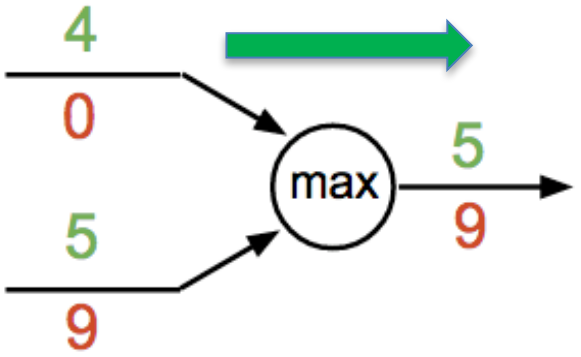
mul gate: "swap multiplier"



copy gate: gradient adder



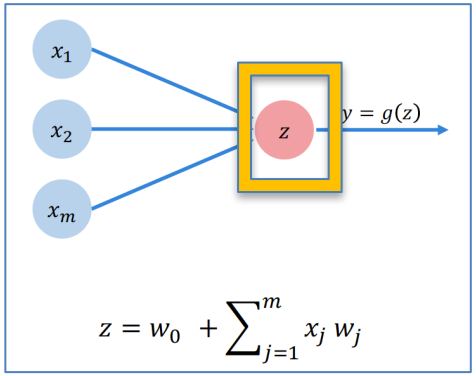
max gate: gradient router



http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

Forward Path Calculation

Input = X(i)



Activation (Neuron)

Sigmoid function

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

X1=-1.0

w0=-2.0

each connection (link) has a weight (w0, w1)

each neuron has a bias (b)

b=w2=-3.0

w1=3.0

X2=-2.0

Z

g(z)

σ (z)

Y

Z = (x1*w1 + x2*w2) + b

output = Y(i)

Z = (x1*w1 + x2*w2) + b = (-1.0*-2.0 + -2.0*3.0) + -3.0 = 4.0 – 3.0 = 1.0

Y = σ (z) = 1 / (1 + e $^{-Z}$) = 1 / (1 + e $^{-1}$) = 0.731

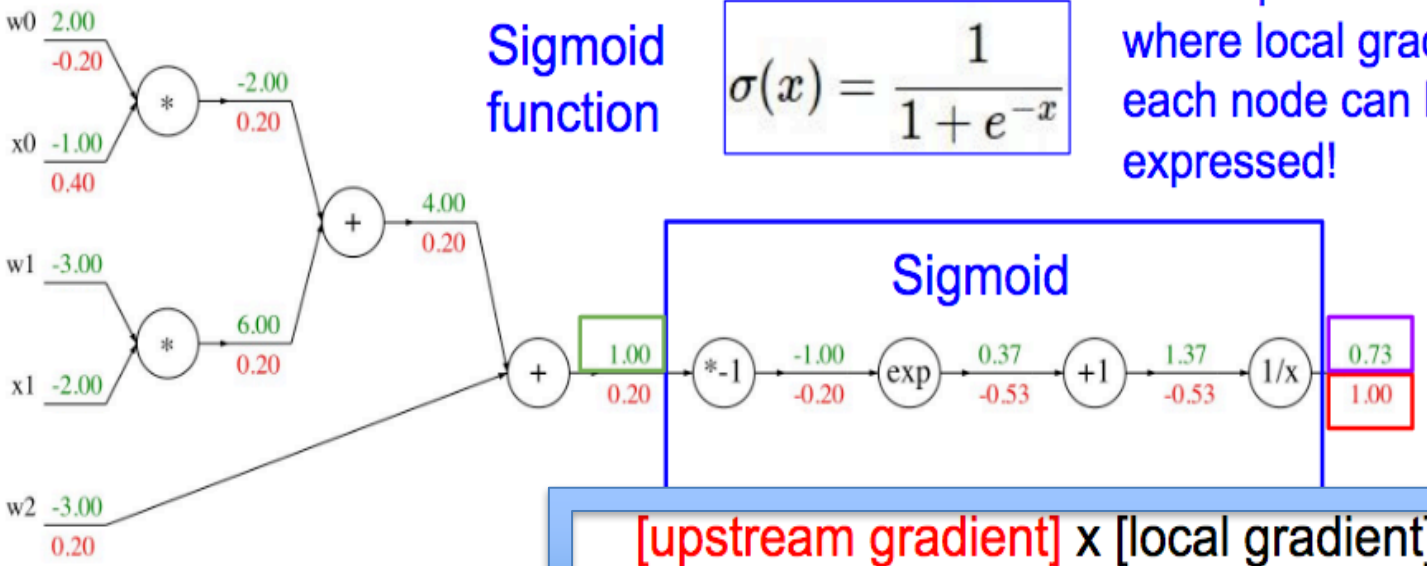
Input (X) → Activation F(X) → Output (Y)

Computational Graph

http://cs231n.stanford.edu/syllabus.html

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



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

[upstream gradient] x [local gradient]

[1.00] x [(1 - 0.73) (0.73)] = 0.2

Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

Input (X) → Activation F(X) → Output (Y)

Input (X) → Activation F(X) → Output (Y)

One data point

One Neuron

Vector vector multiply

W0 W1 | X0, X1 | + b=w2 = Z

each connection (link) has a weight (w0, w1)

each neuron has a bias (b)

2.0 -3.0 | -1.0, -2.0 | + -3.0 = 1.0

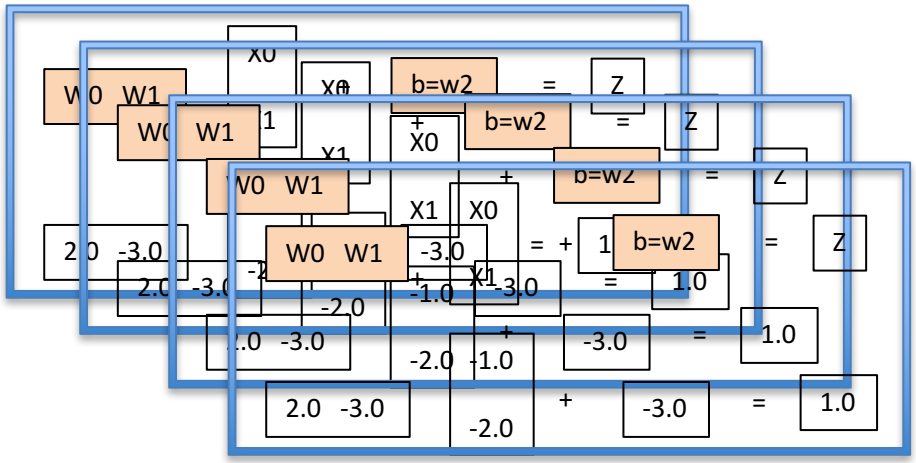
$Y = \text{Activation}(Z) = \sigma(z) = 1 / (1 + e^{-Z}) = 1 / (1 + e^{-1}) = 0.731$

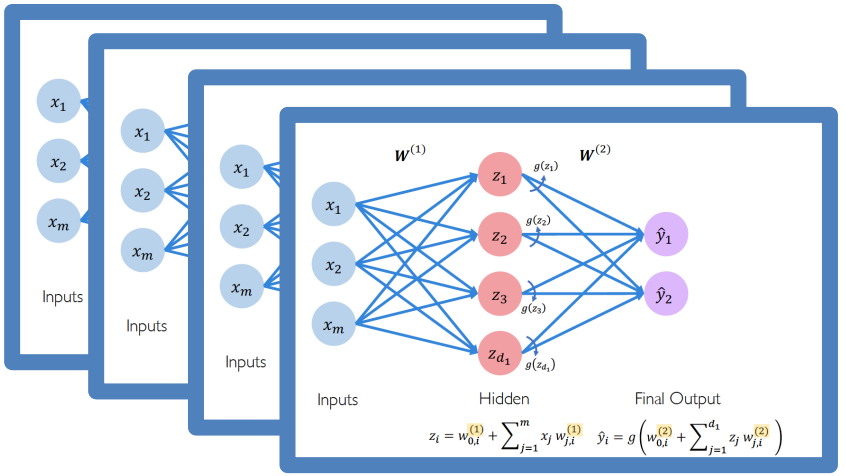
Trainable parameters
W (i) = Weights, b = bias
1 weight per connection (link)
1 bias per neuron
Total number of parameters = 3

Weight and bias are initialized randomly

Goal : Adjust W and b
Training a neural network

Input (X) → Activation F(X) → Output (Y)





many data points; many neurons → matrix matrix multiplication (GEMM)

Y = Activation (Z) = σ (z) = 1 / (1 + e $^{-Z}$) = 1 / (1 + e $^{-1}$) = 0.731

Trainable parameters
W (i) = Weights, b = bias
1 weight per connection (link)
1 bias per neuron
Total number of parameters = 3

Weight and bias
are initialized randomly

Goal : Adjust W and b
Training a neural network

Backpropagation Operators

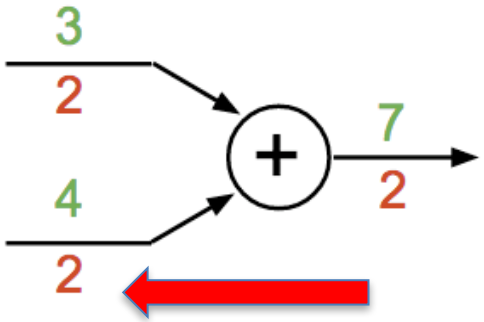
Backward path : Training : using gradient to adjust parameters

new W = old W– learning rate * (grad [Y w.r.t W])

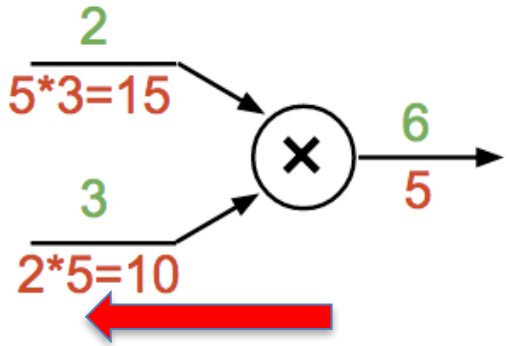
$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

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

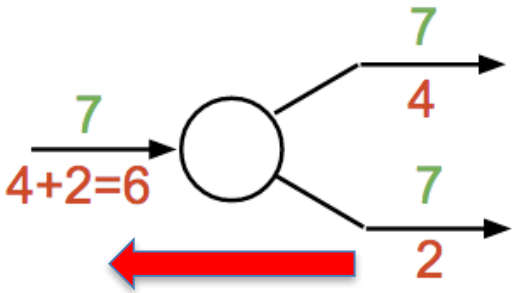
add gate: gradient distributor



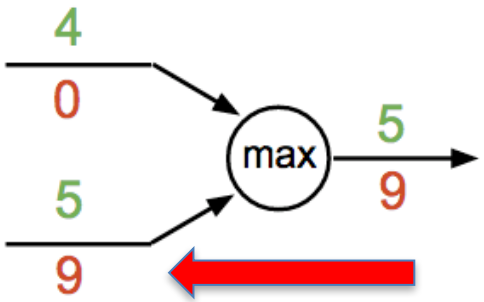
mul gate: "swap multiplier"



copy gate: gradient adder

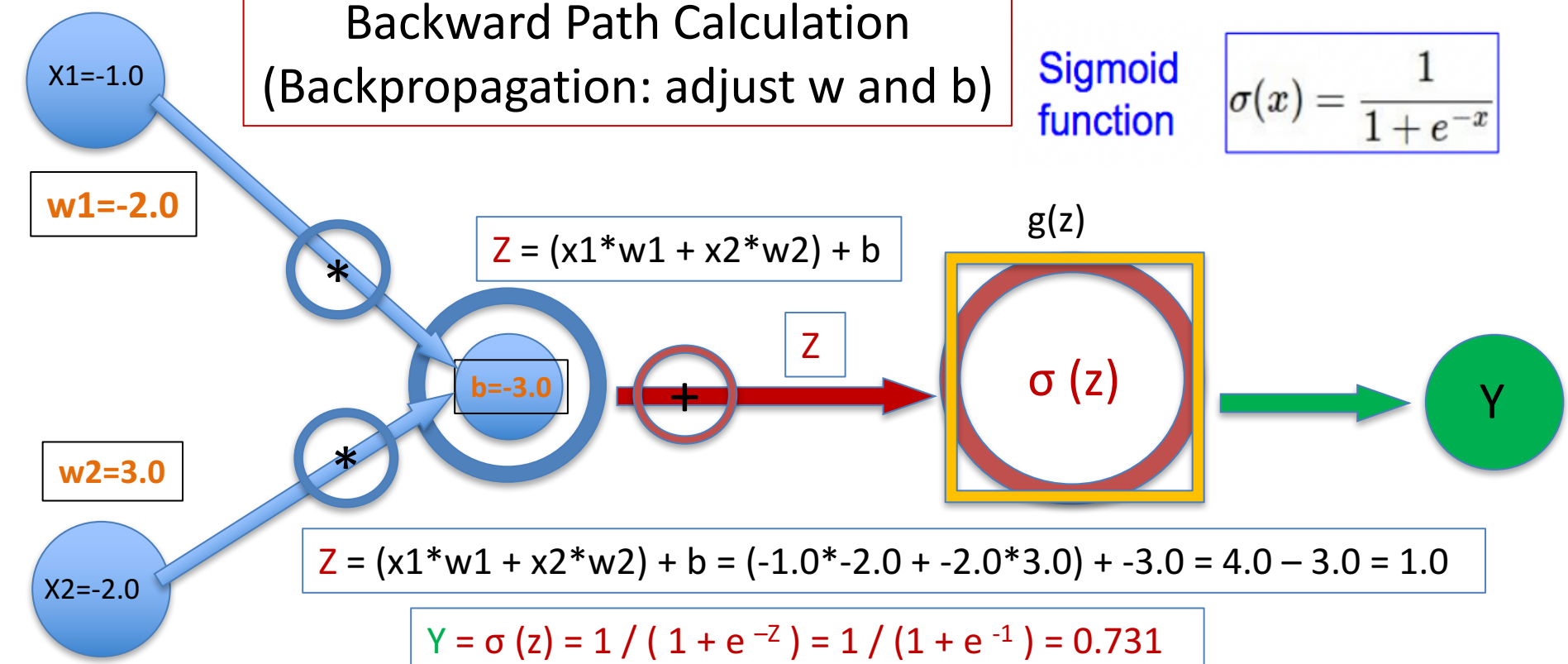


max gate: gradient router



http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

Backward Path Calculation (Backpropagation: adjust w and b)



Sigmoid function

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

Z = (x1*w1 + x2*w2) + b

Z = (x1*w1 + x2*w2) + b = (-1.0*-2.0 + -2.0*3.0) + -3.0 = 4.0 – 3.0 = 1.0

Y = σ (z) = 1 / (1 + e^{-Z}) = 1 / (1 + e^{-1}) = 0.731

Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

Downstream gradient = grad z = d Y/dz

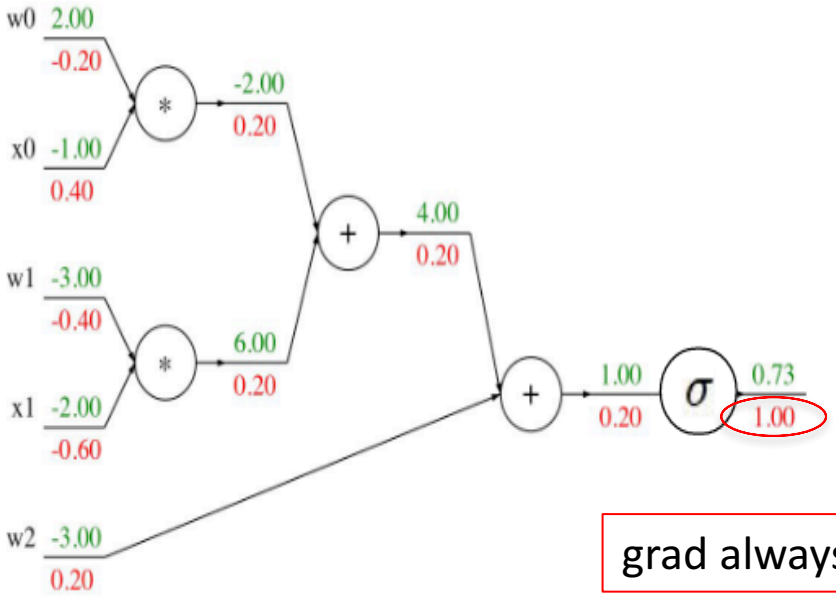
[upstream gradient] x [local gradient]

[1.00] x [(1 - 0.73) (0.73)] = 0.2

Forward and Backpropagation Code

Forward path : backward path

Backprop Implementation: "Flat" code



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

Backward pass:
Compute grads

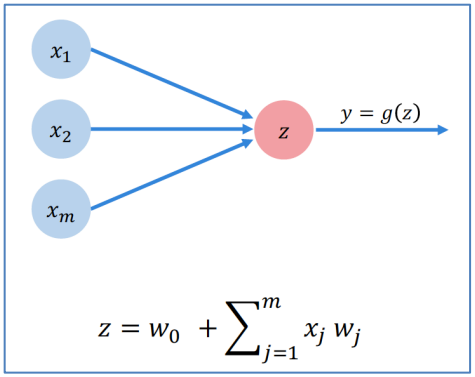
```
  grad_L = 1.0
  grad_s3 = grad_L * (1 - L) * L
  grad_w2 = grad_s3
  grad_s2 = grad_s3
  grad_s0 = grad_s2
  grad_s1 = grad_s2
  grad_w1 = grad_s1 * x1
  grad_x1 = grad_s1 * w1
  grad_w0 = grad_s0 * x0
  grad_x0 = grad_s0 * w0
```

grad always = 1 at the end

http://cs231n.stanford.edu/syllabus.html

Forward and backward computation operators

Z = (x1*w1 + x2*w2) + b = (-1.0*-2.0 + -2.0*3.0) + -3.0 = 4.0 – 3.0 = 1.0



Sigmoid function

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

Y = σ (z) = 1 / (1 + e $^{-Z}$)
= 1 / (1 + e $^{-1}$) = 0.731

multiple operator :
downstream gradient =
upstream gradient * input value

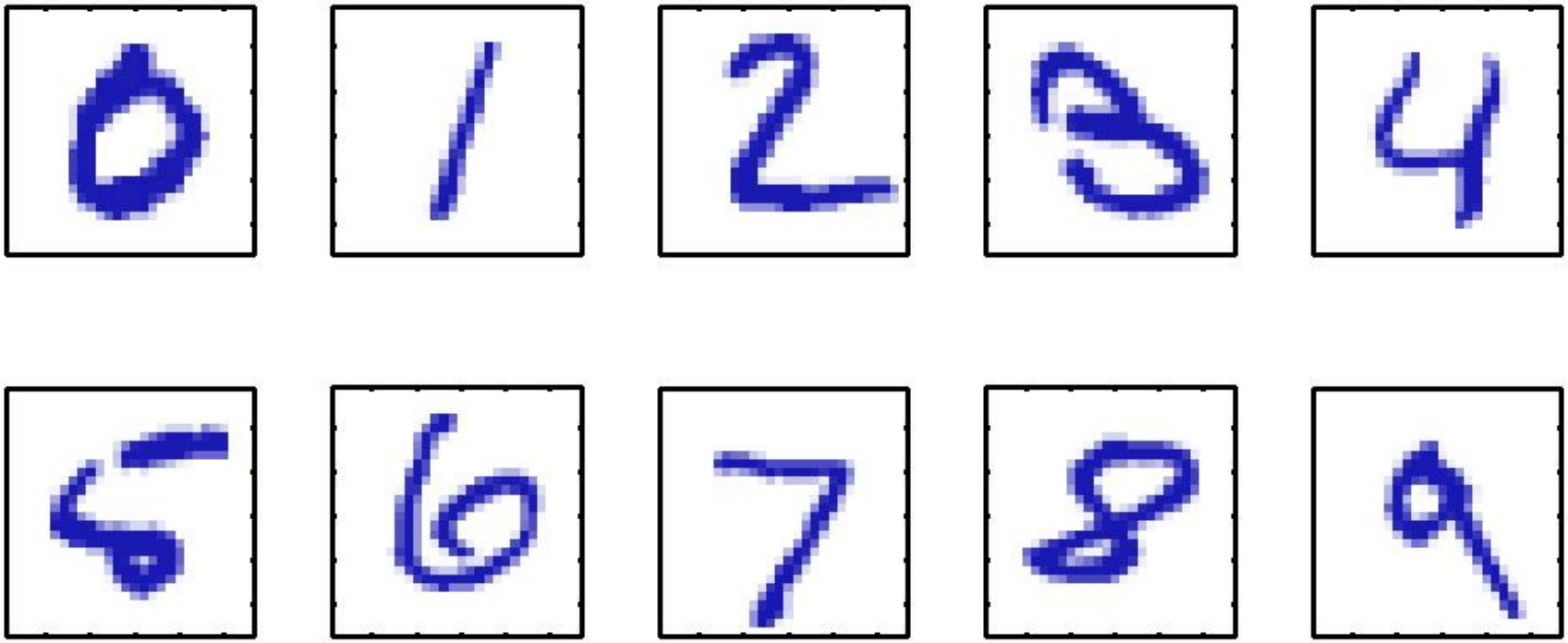
add operator :
Downstream gradient =
upstream gradient

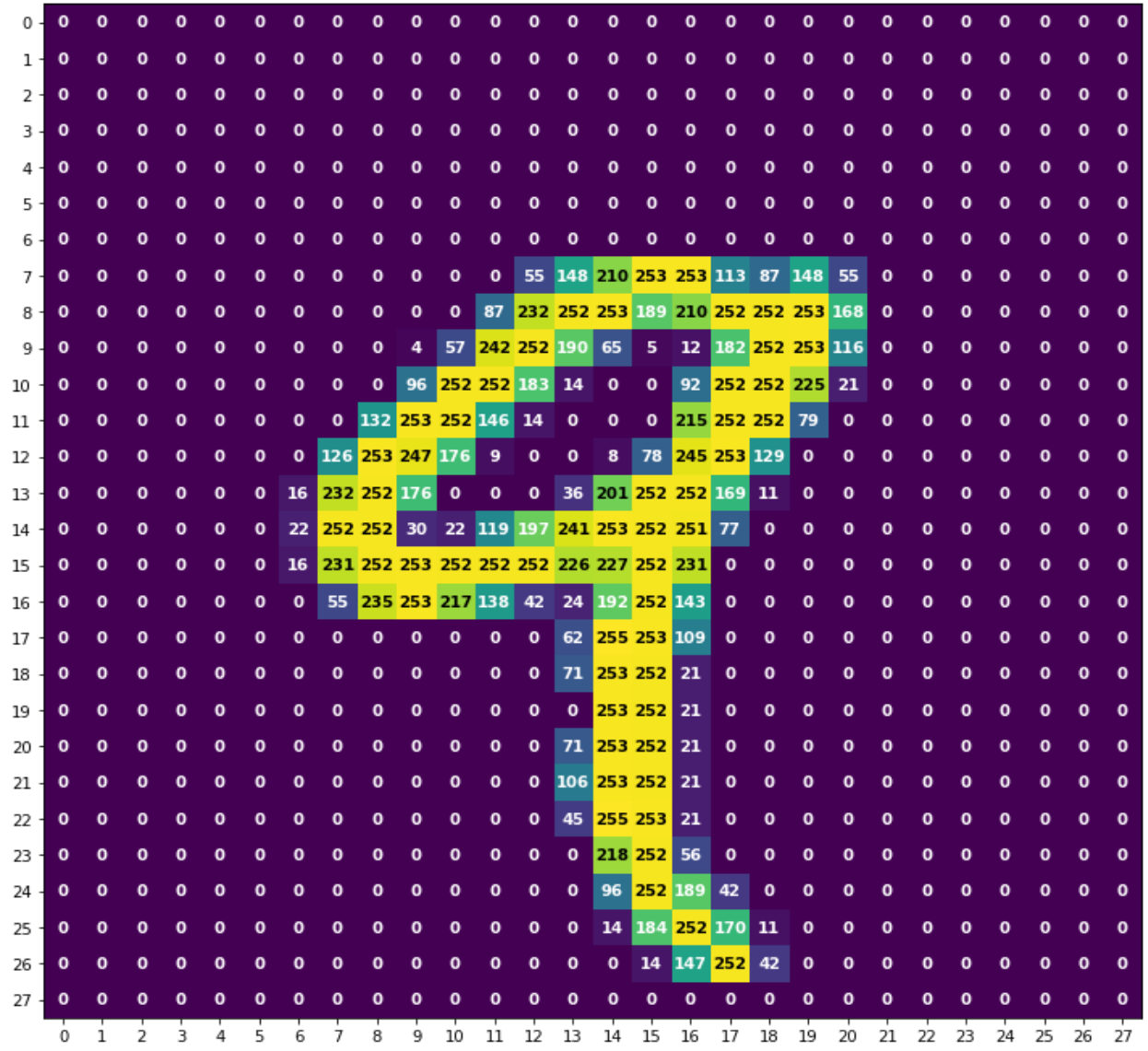
function operator :
downstream gradient =
Upstream gradient * local function gradient

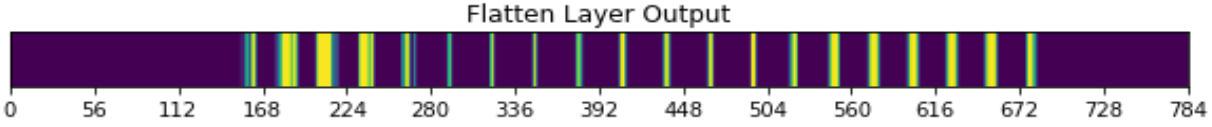
Example: how can computer see images?

Handwritten Digit Recognition (MNIST data set)

The **MNIST database** (*Modified National Institute of Standards and Technology database*) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.[4][5] It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. The MNIST database contains 60,000 training images and 10,000 testing images. – from Wikipedia







Grayscale image
of 28 x 28 pixels

same as a
28 x 28 matrix

values of
0 - 255

Flatten the

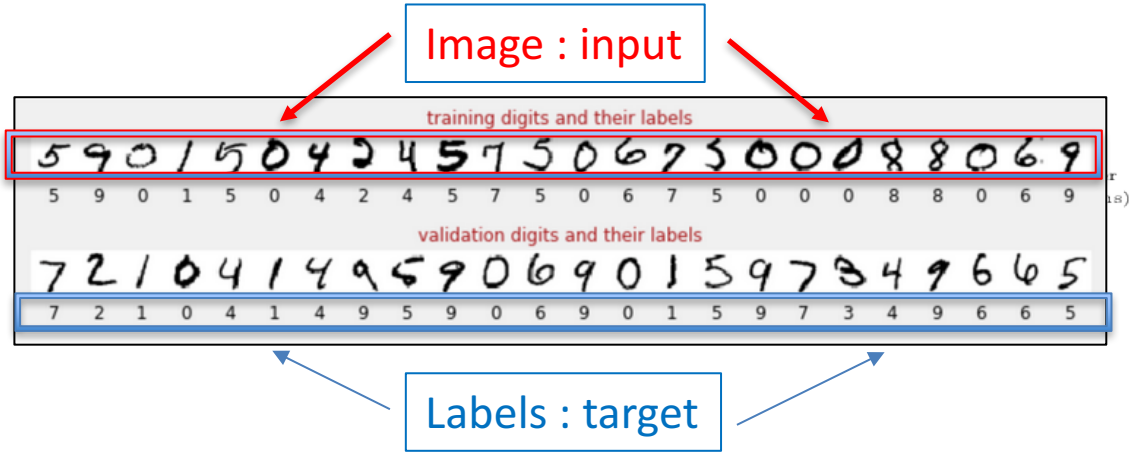
28 x 28 matrix

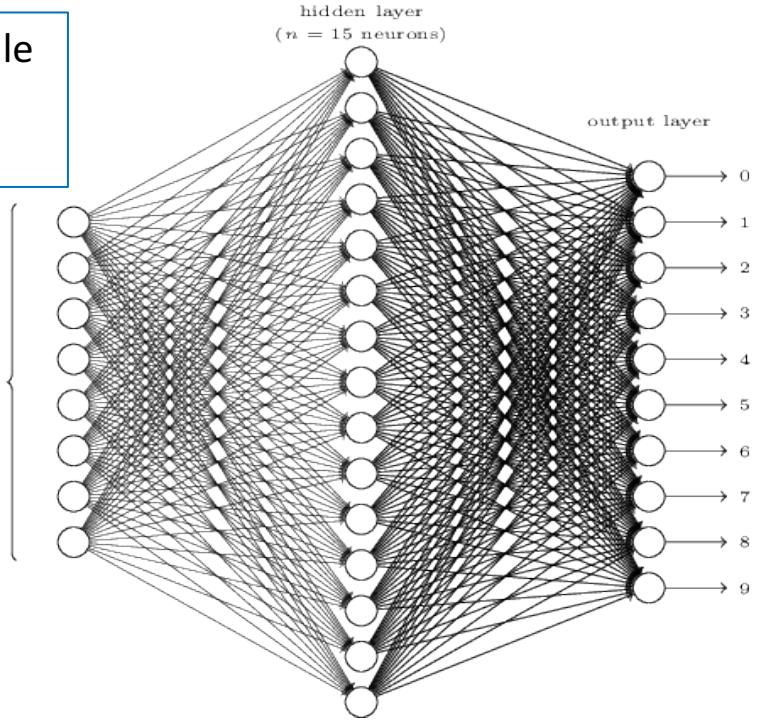
to a vector of
28 x 28 elements

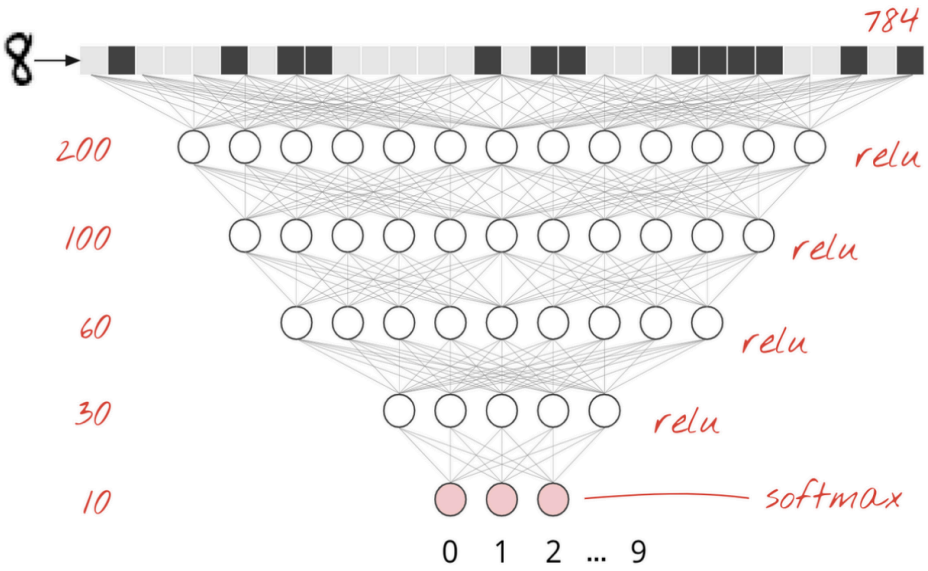
784 elements

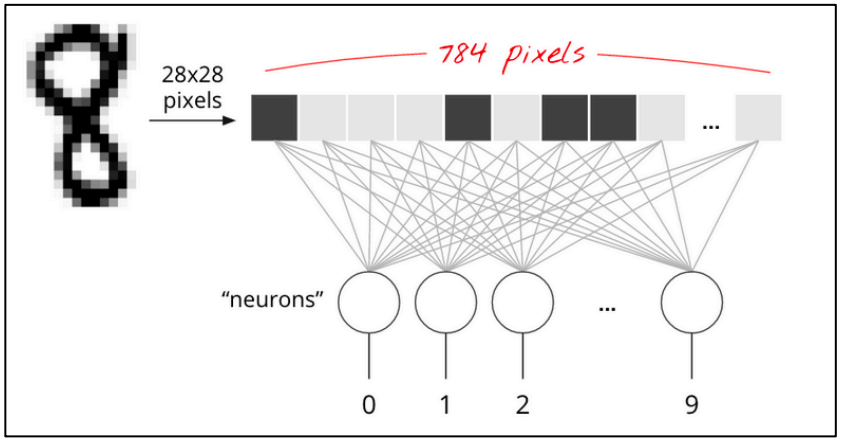
MNIST Example (28x28 pixels)

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images. The simplest approach for classifying them is to use the 28x28=784 pixels as inputs for a 1-layer neural network.



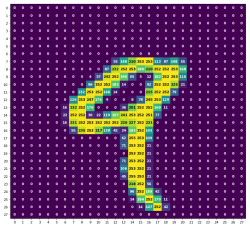






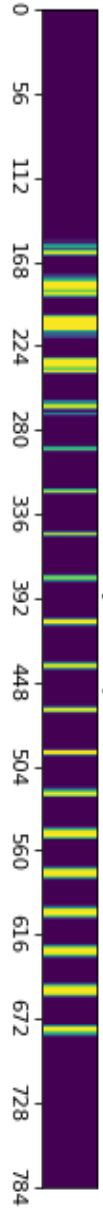
·Flow, Keras and deep learning, without a PhD
https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist#0

Flatten the 28 x 28 matrix to a vector of 28 x 28 of 784 elements vector = Input



Simple MNIST MLP Network

Google Colab Code



Flatten Layer Output

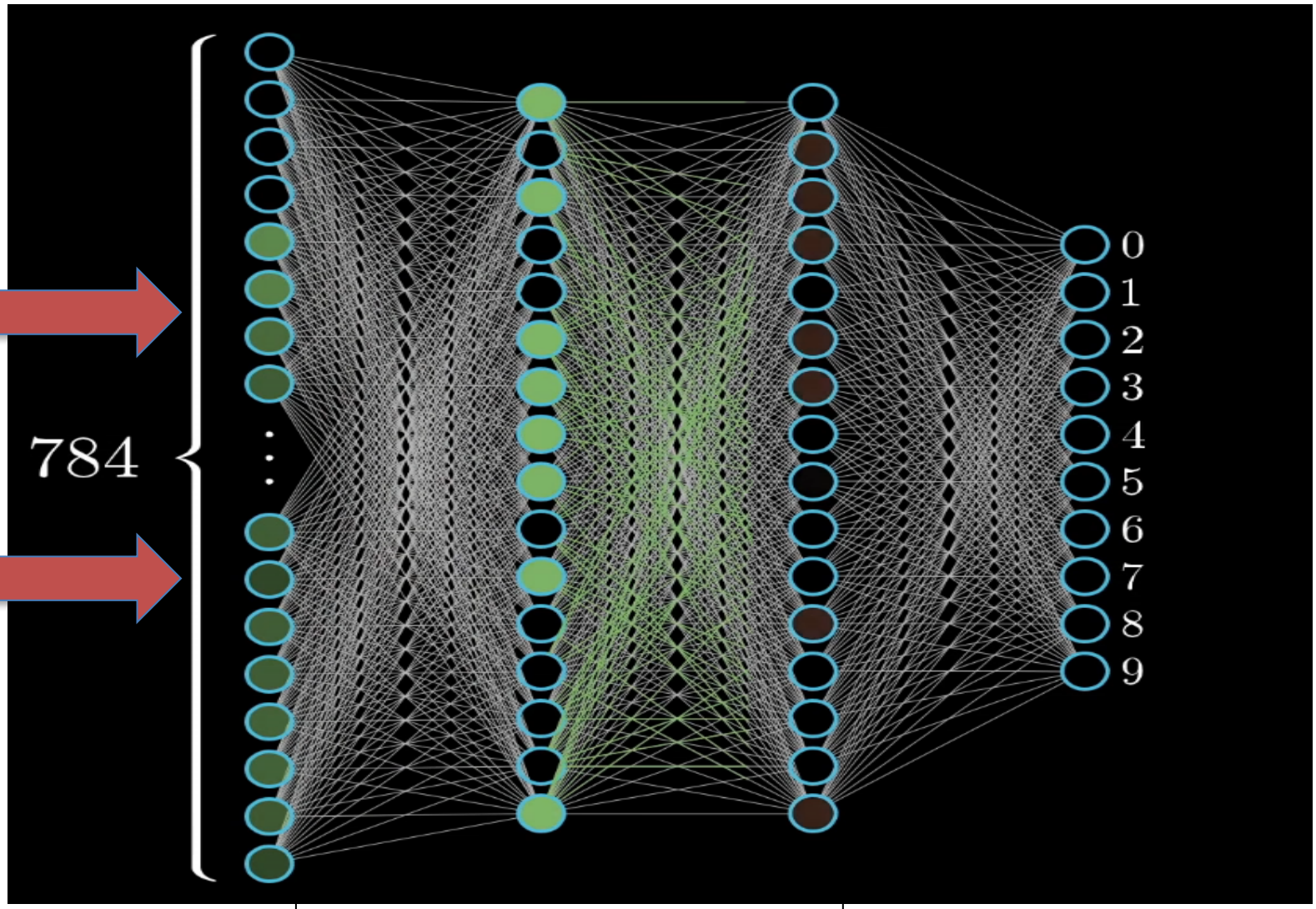


Image:https://www.3blue1brown.com/

Tensorflow Example

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(16, activation='relu'),
 tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Tensorflow is imported

Sets the mnist dataset to variable "mnist"

Loads the mnist dataset

Builds the layers of the model
4 layers in this model

```
model.compile(optimizer='sgd', loss='SparseCategoricalCrossentropy',
metrics=['accuracy'])

model.summary()

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-
%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
```

Loss Function

Compiles the model with the SGD optimizer

Print summary

Use tensorborad

```
model.fit(x_train, y_train, epochs=5, batch_size=10,
validation_data=(x_test, y_test),  callbacks=[tensorboard_callback])

model.evaluate(x_test,  y_test, verbose=2)
%tensorboard --logdir logs
```

Adjusts model parameters to minimize the loss
Tests the model performance on a test set

Tensorflow Example

The model is trained in about 10 seconds completing 5 epochs with a Nvidia GTX 1080 GPU and has an accuracy of around 97-98%

```
2020-07-29 19:53:40.592860: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1108]      0
2020-07-29 19:53:40.592871: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1121] 0:   N
2020-07-29 19:53:40.593073: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.593535: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.593973: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.594387: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1247] Created TensorFlow device (/job:lo
calhost/replica:0/task:0/device:GPU:0 with 7219 MB memory) -> physical GPU (device: 0, name: GeForce GTX 1080, pci bus
id: 0000:26:00.0, compute capability: 6.1)
2020-07-29 19:53:40.623075: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x55d981198b80 initialized fo
r platform CUDA (this does not guarantee that XLA will be used). Devices:
2020-07-29 19:53:40.623096: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): GeForce GTX
1080, Compute Capability 6.1
2020-07-29 19:53:52.215159: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
 library libcublas.so.10
Epoch 1/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2982 - accuracy: 0.9127
Epoch 2/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1433 - accuracy: 0.9571
Epoch 3/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1097 - accuracy: 0.9663
Epoch 4/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0893 - accuracy: 0.9730
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0777 - accuracy: 0.9750
313/313 - 0s - loss: 0.0727 - accuracy: 0.9776
```

Simple MNIST MLP Neural Network

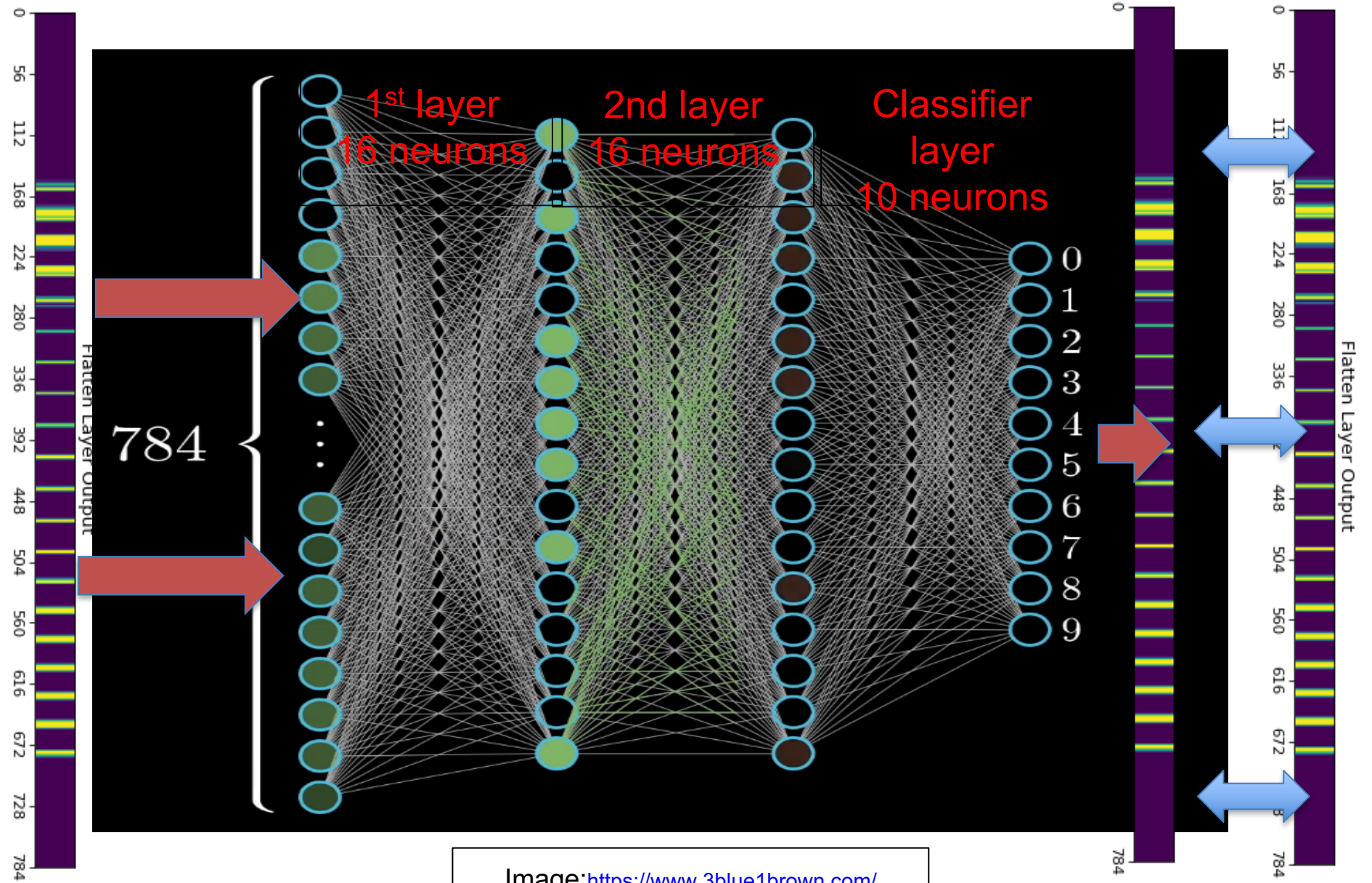
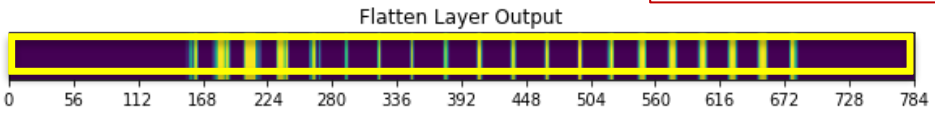
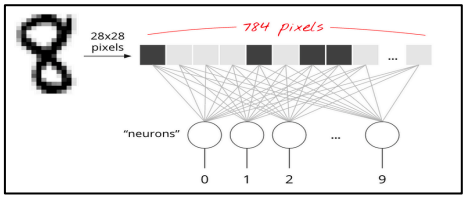


Image:https://www.3blue1brown.com/

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784





An image of 28x28 pixels

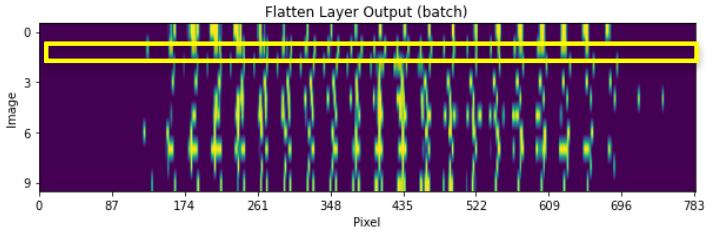
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

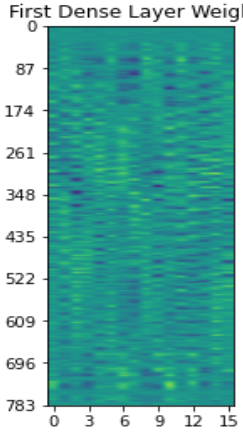
16 neurons
784 x 16 W matrix

10 x 16 output matrix

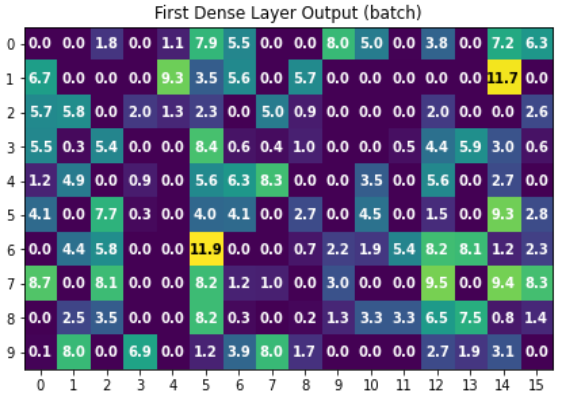
Input to 2nd layer



*

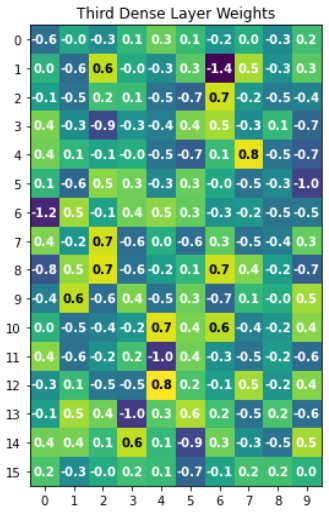


=



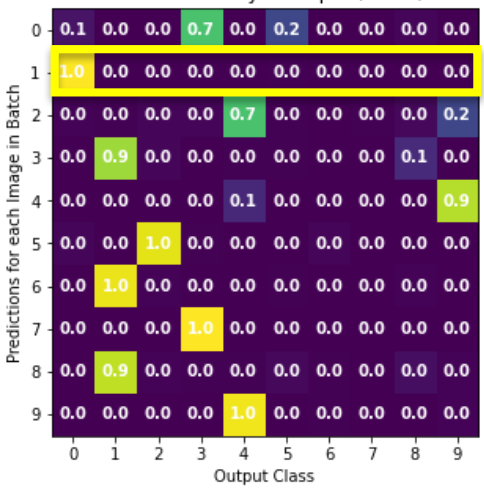
10 x 16 matrix

*

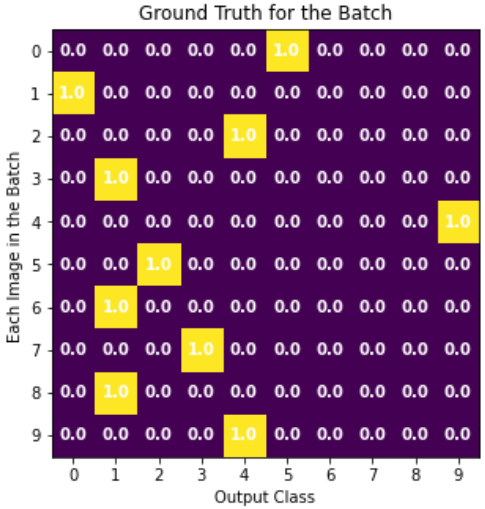


16 neurons
16 x 10
W matrix

=

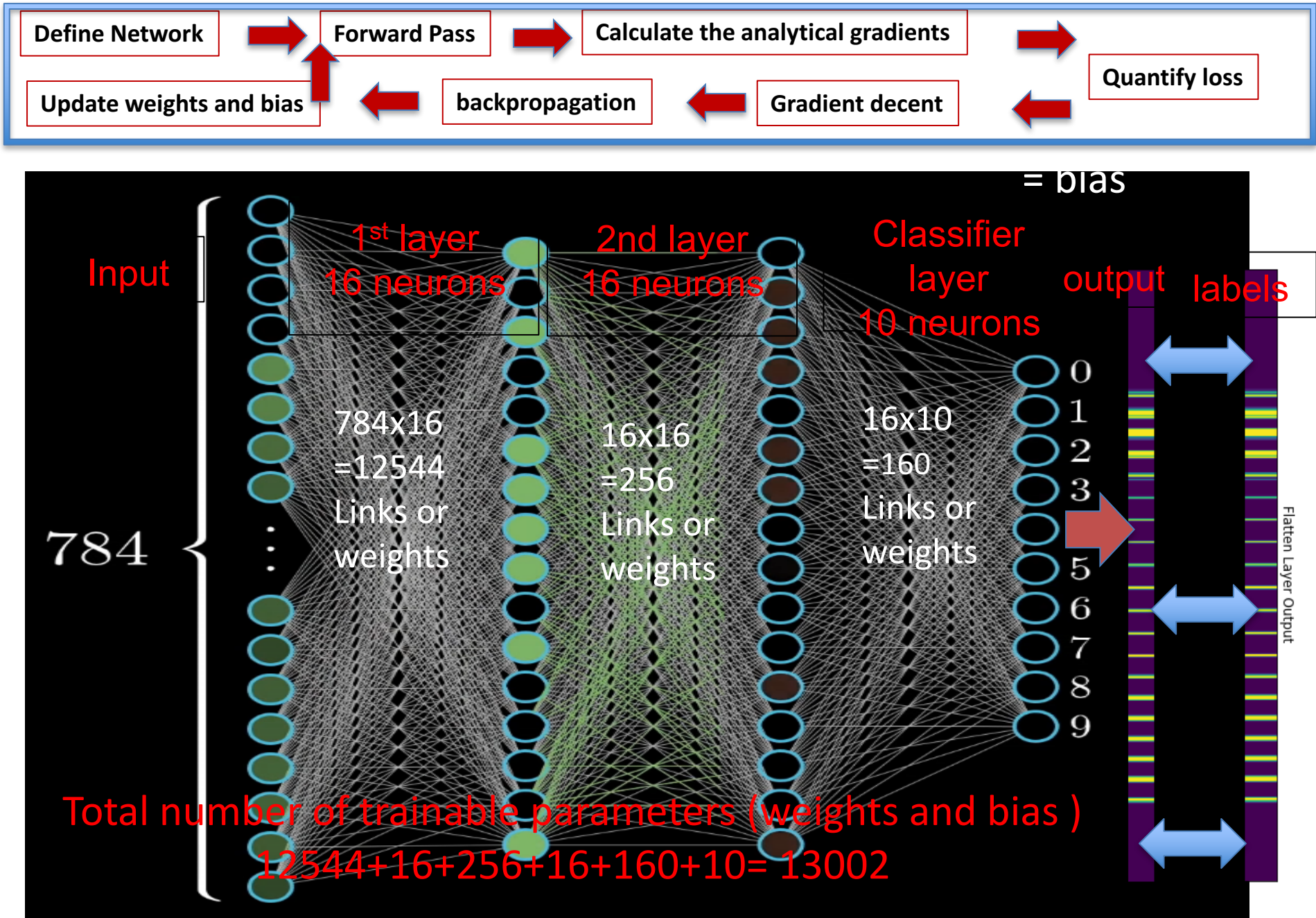


10 x 10 output matrix

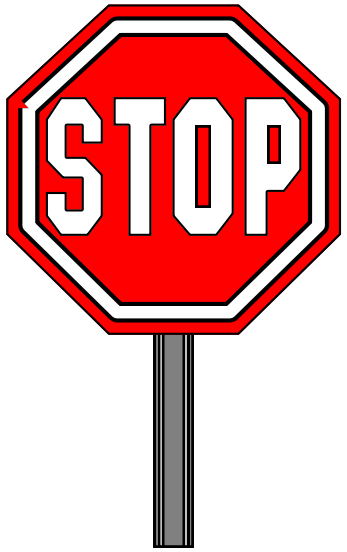


10 x 10 labeled matrix

```
model.fit(x_train, y_train, epochs=3, batch_size=10)
```



The End



- The End!