LECTURE UNITS
MAT391
Topics in Machine Learning

Kwai Wong
University of Tennessee, Knoxville

September 19, 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 2: MLP Training

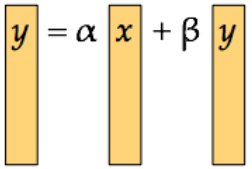
- Linear Algebra Revisit, Performance
- GPU Brief Overview
- Google Colaboratory and python overview
- DNN TensorFlow MNIST review
- DNN Training
- Backpropagation
- Optimization

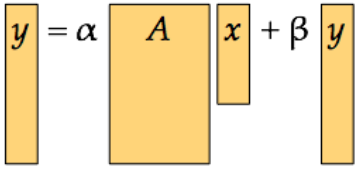
Basic Linear Algebra Subprograms (BLAS)

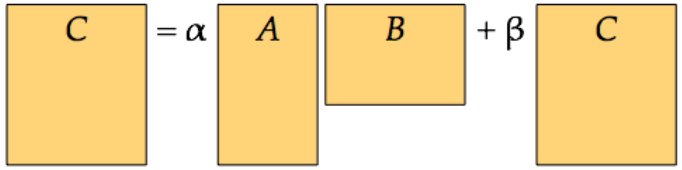
- **BLAS is a library of standardized basic linear algebra computational kernels created to perform efficiently on serial computers taking into account the memory hierarchy of modern processors.**
- **BLAS1 does vectors-vectors operations.**
  - **Saxpy = y(i) = a* x(i) + y(i), ddot= ∑x(i) *y(i)**
- **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







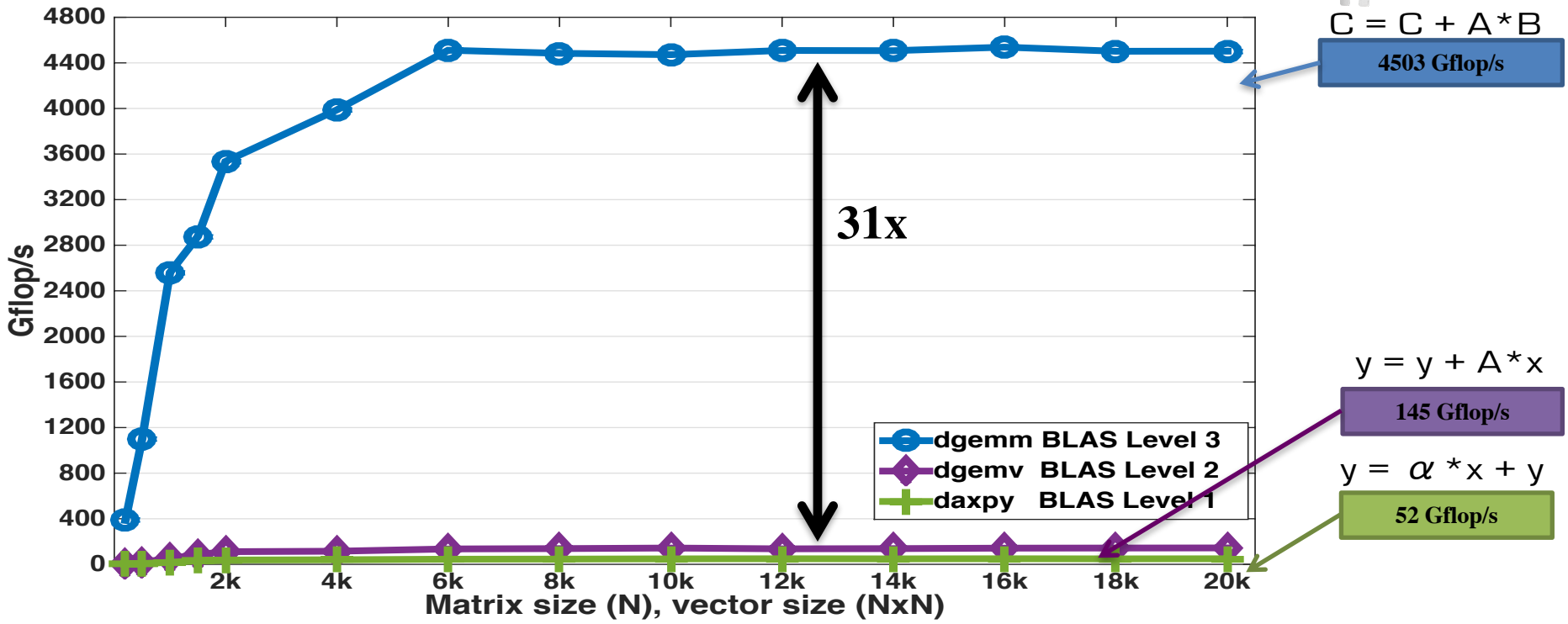
Level 1, 2 and 3 BLAS on GPU

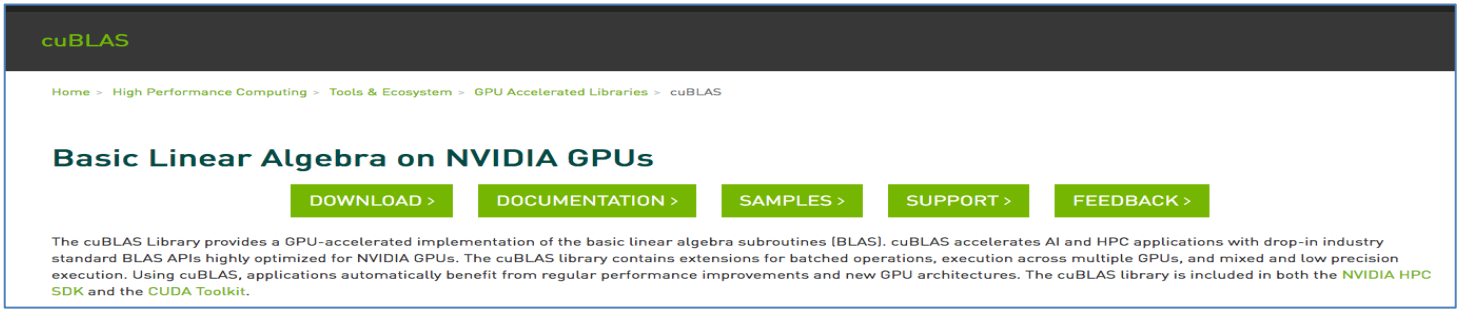
Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

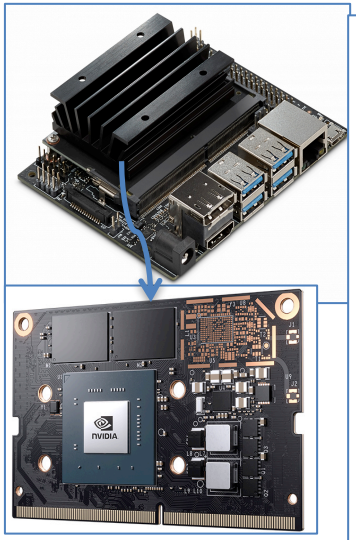
Nvidia P100, The theoretical peak double precision is 4700 Gflop/s, CUDA version 8.0

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





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, ..

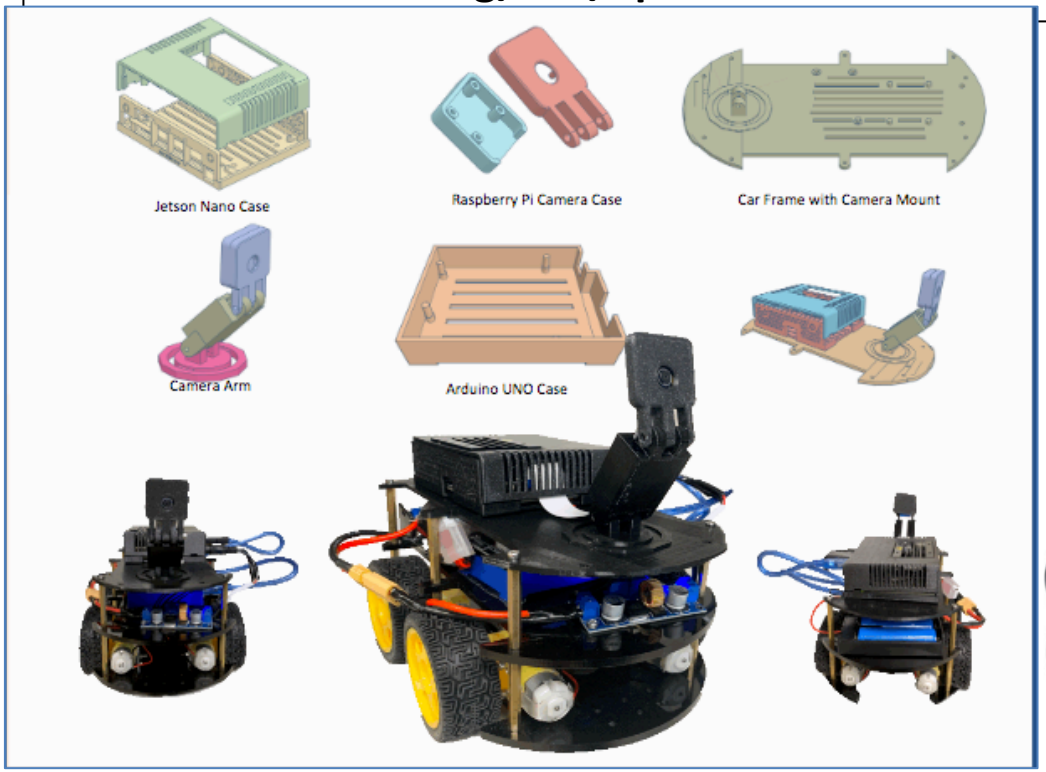
PRODUCT SPECIFICATIONS	
NVIDIA® CUDA Cores	896
Clock Speed	1485 MHz
Boost Speed	1725 MHz
Memory Speed (Gbps)	8
Memory Size	4GB GDDR5
Memory Interface	128-bit
Memory Bandwidth (Gbps)	128

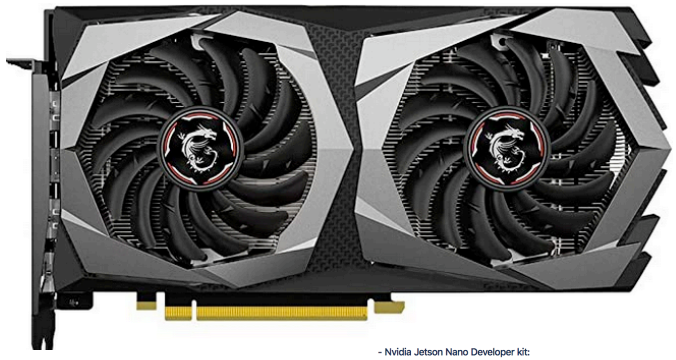
NVIDIA V100 on Summit

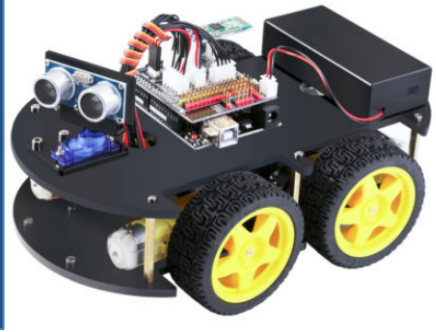
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
Memory Bandwidth	900GB/sec	

Desktop Computers and GPU

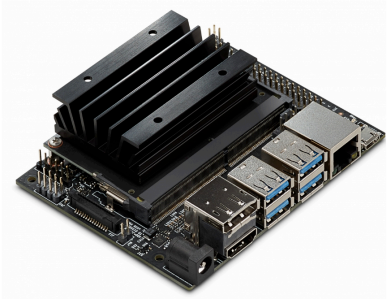
- ✓ **Computer Access, needs of each team, Desktop, laptop, Jetson Nano, Car**
- ✓ **Desktops configuration and deliveries, mid September**
- ✓ **Dell Server Box 12 cores, 48G Ram, 1TB Disk, 1650 super GPU card**
- ✓ **Ubuntu OS, anaconda individual, https://www.anaconda.com/products/individual**
- ✓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



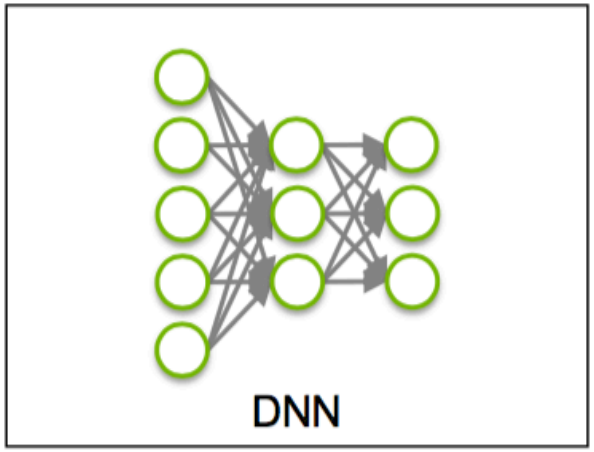


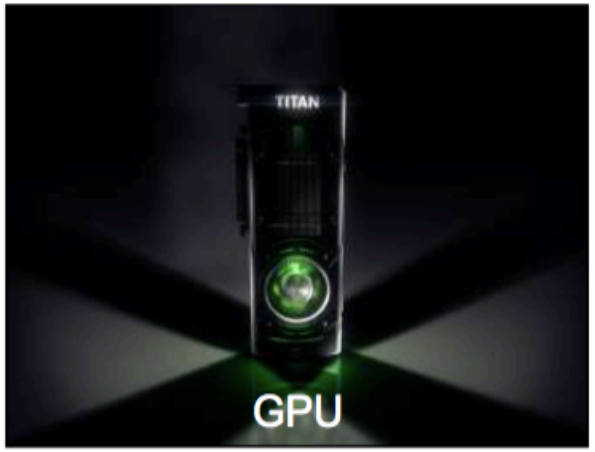


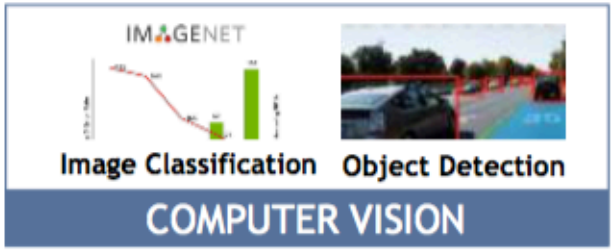
- Nvidia Jetson Nano Developer kit:

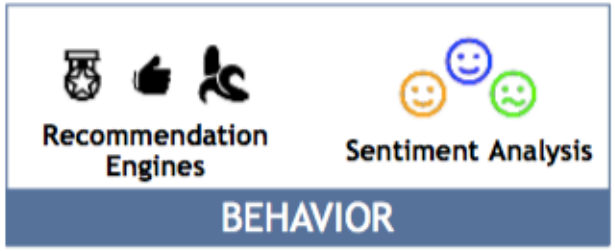


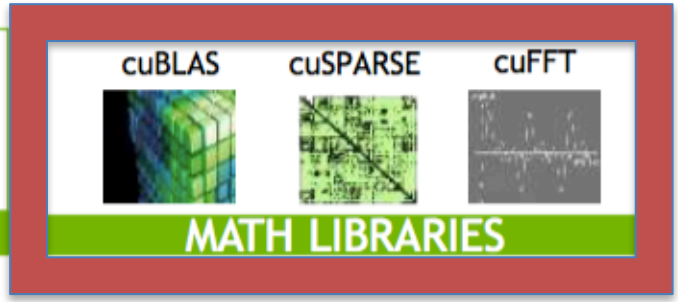
Machine Learning :LA (BLAS3) – GPU acceleration

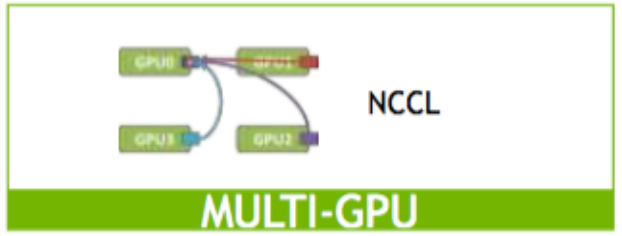












Source captured frin - Julie Bernauer – HPC Advisory Council Stanford Tutorial – 2017/02/07

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"

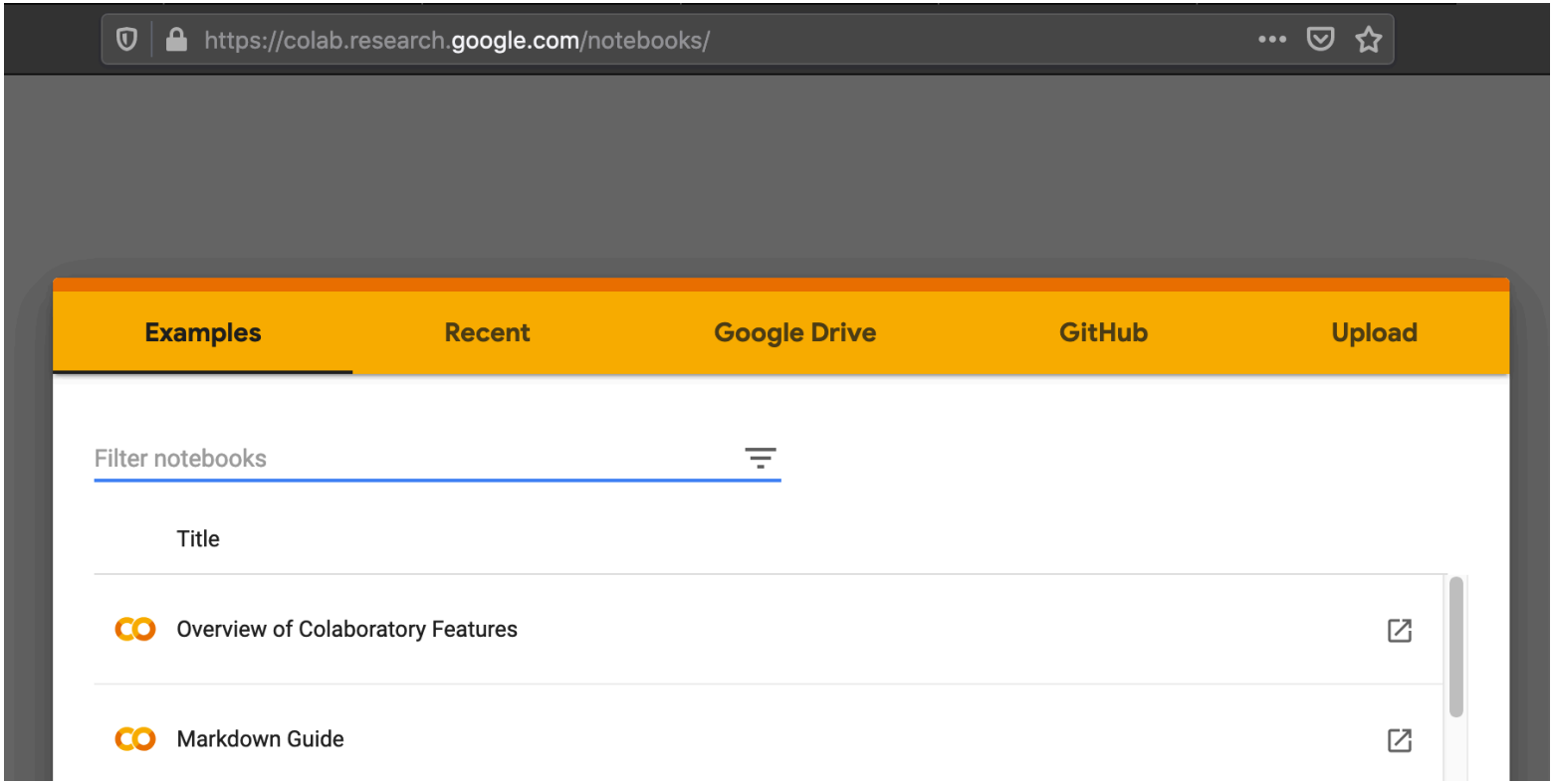
Google Colab, jupyter notebook
python, R tutorials
upload file to /content
mount your google drive

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**

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 :

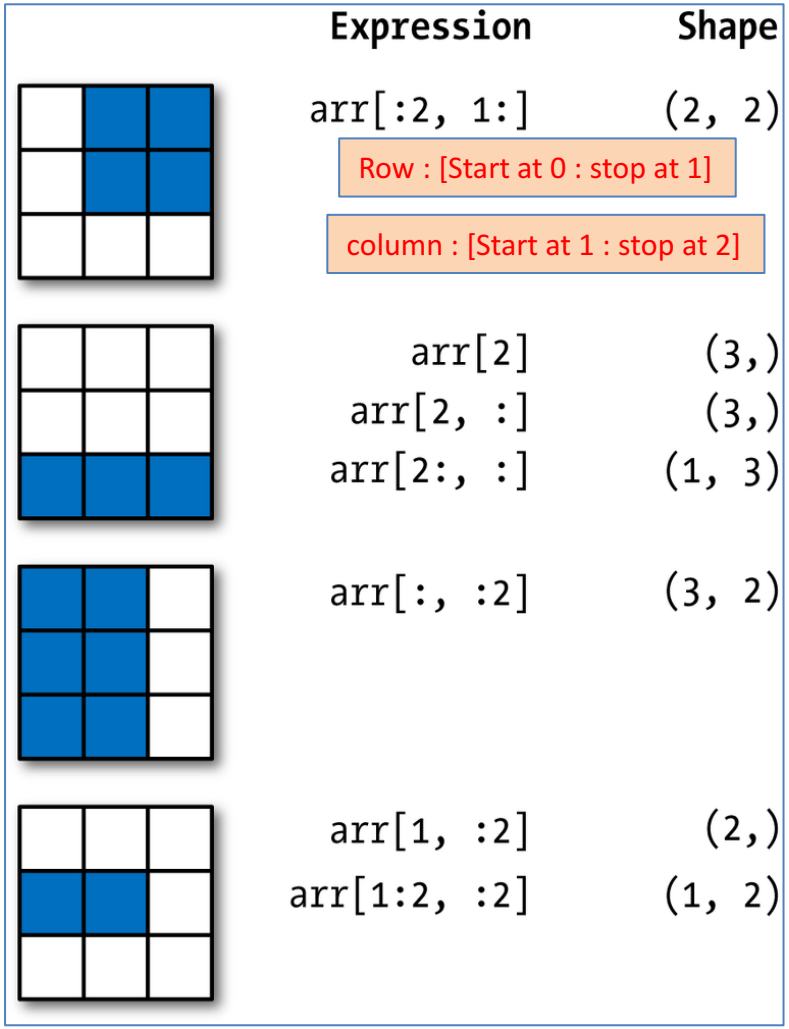
```
b=np.array( [  [1,0,-5] , [-2.5, 7.3, 4.6]  ] )
```

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

Python Numpy – 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!!

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()
```

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
```

Supervised Learning

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

Goal:
Learn a function to map x -> y

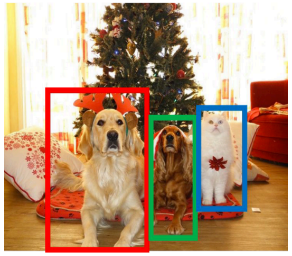
Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

CAT

Classification

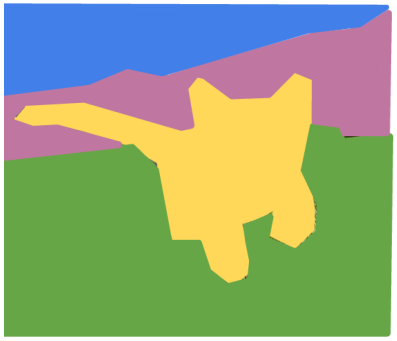
A cat sitting on a suitcase on the floor

Image captioning



DOG, DOG, CAT

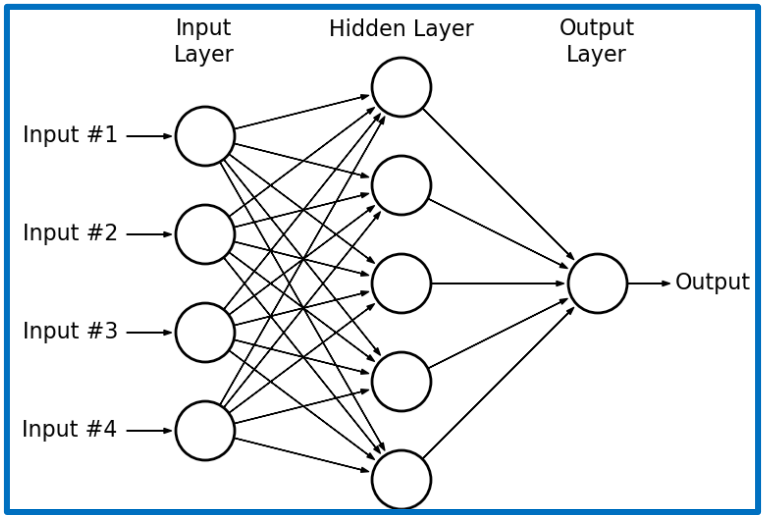
Object Detection

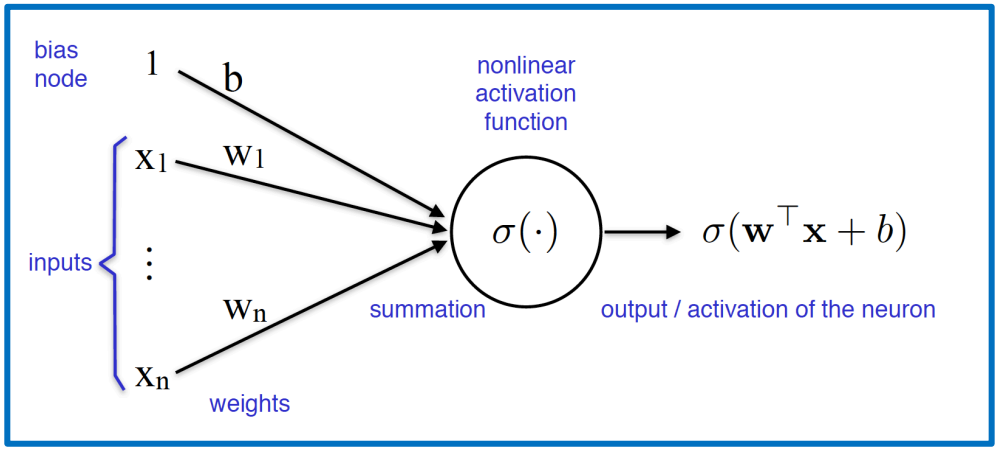


GRASS, CAT, TREE, SKY

Semantic Segmentation

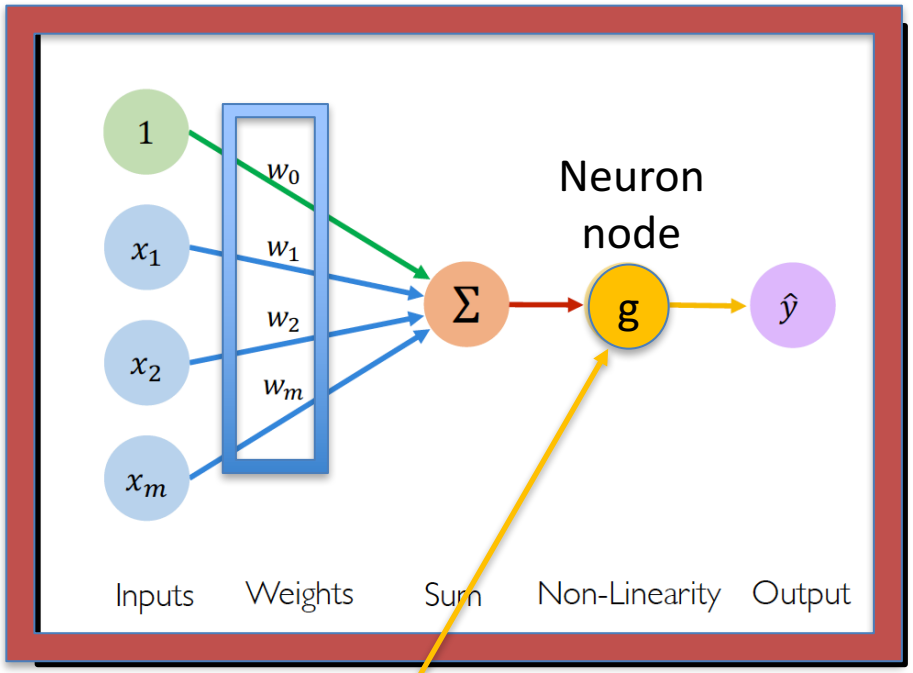
NN Modeling





- ✓ 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.

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

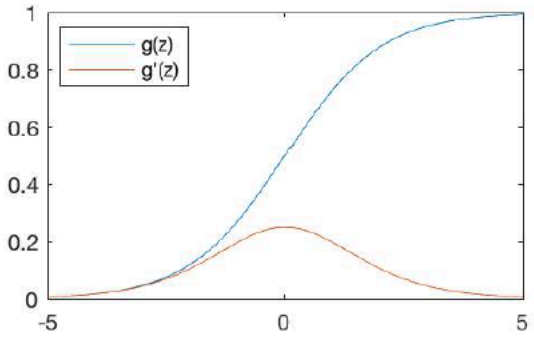
$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

- Example: sigmoid function

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

Nonlinear Activation Function

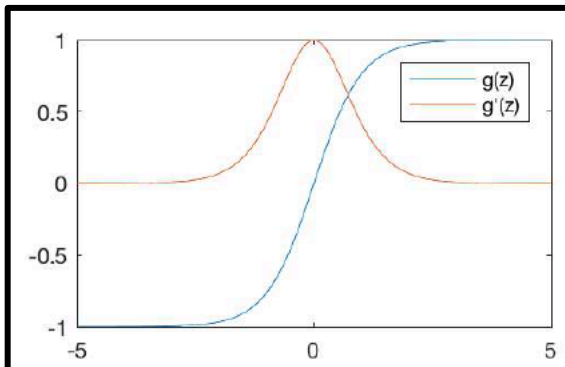
Sigmoid Function



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

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

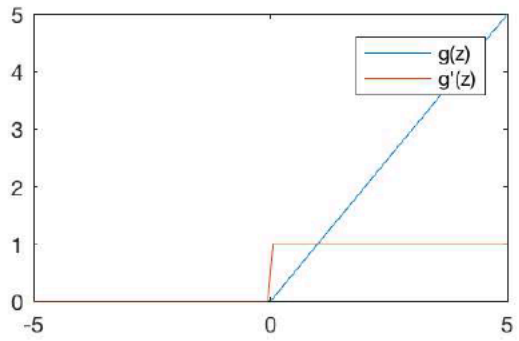
tf.nn.sigmoid(z)



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

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

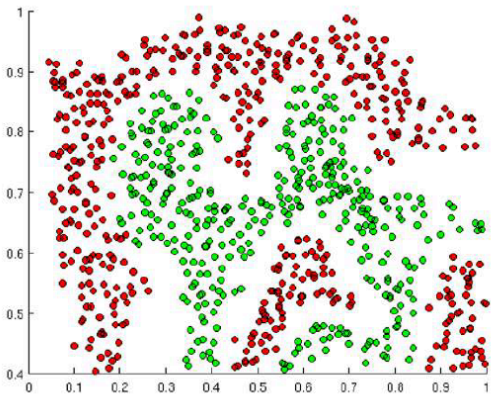
tf.nn.tanh(z)

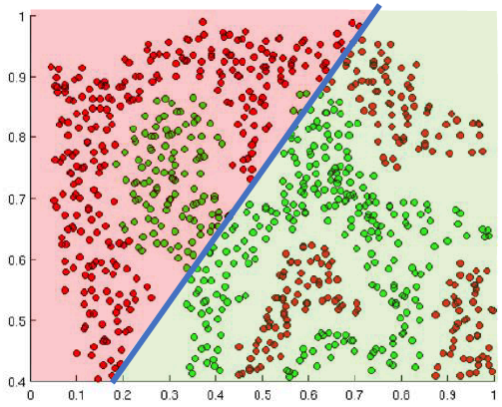


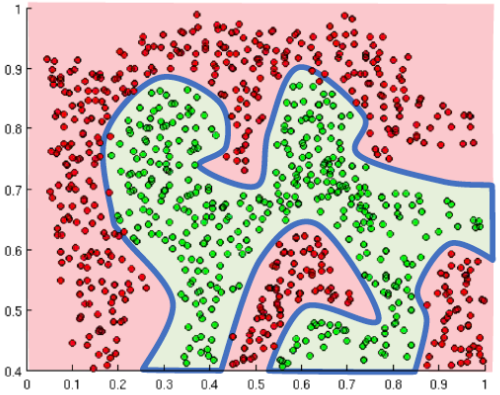
$$g(z) = \max(0, z)$$

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

tf.nn.relu(z)

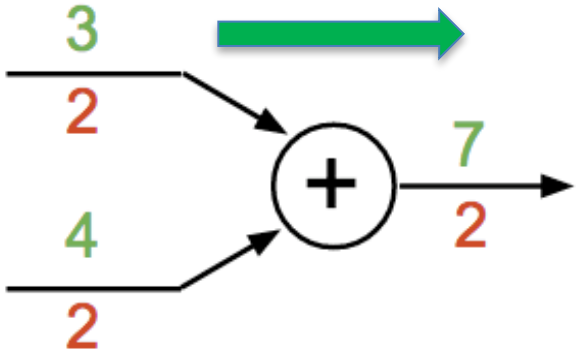




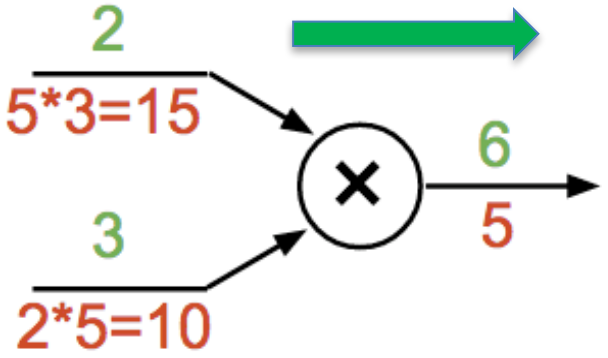


Forward Computing Operators

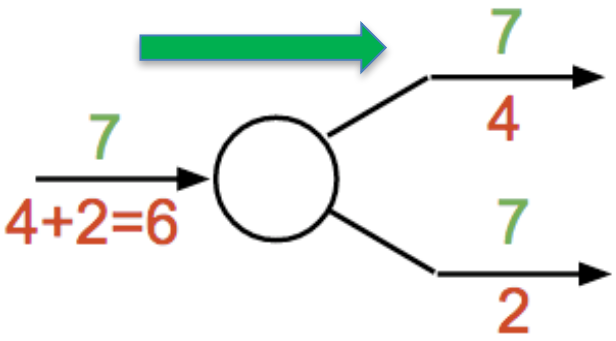
add gate: gradient distributor



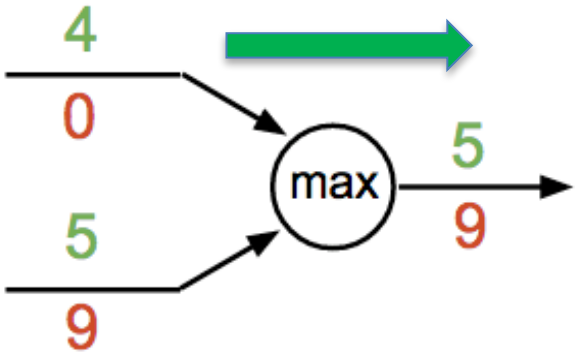
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router



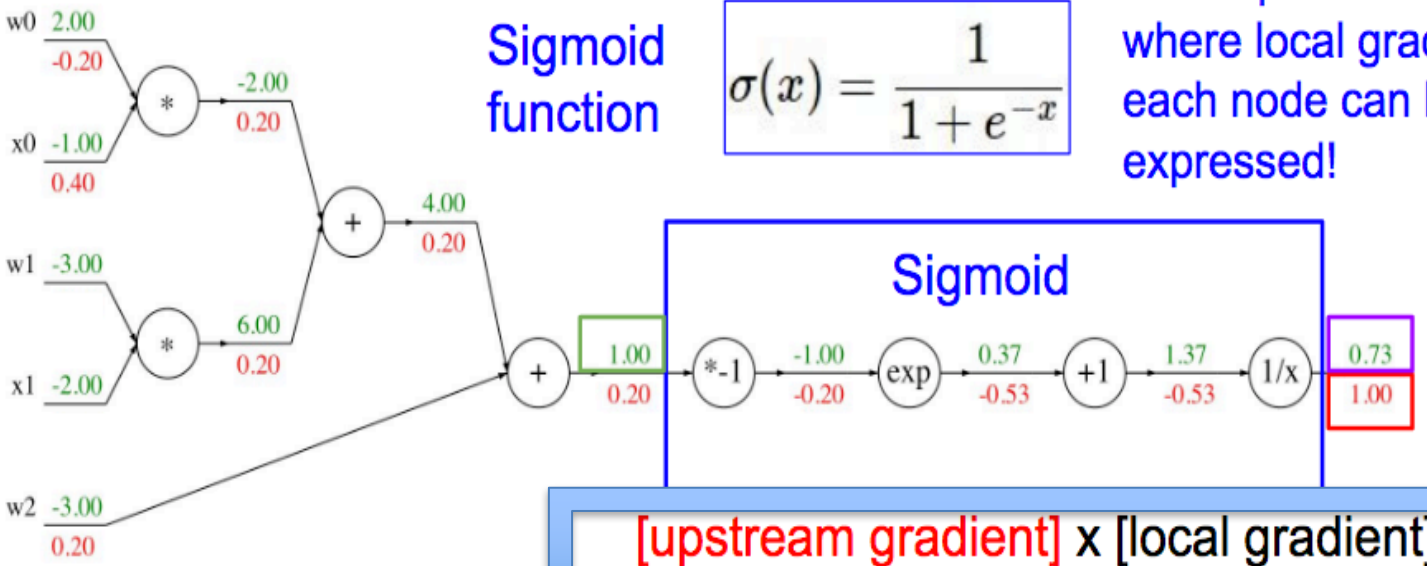
http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

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

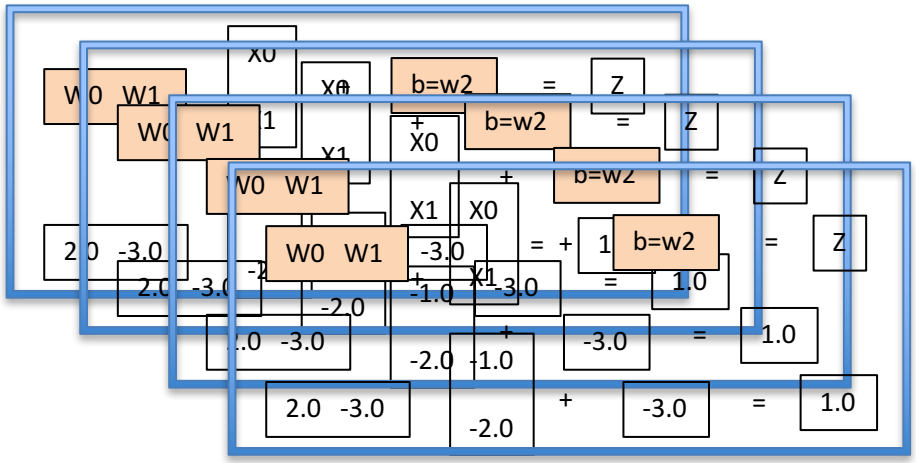
$Y = \text{Activation}(Z) = \sigma(z) = 1 / (1 + e^{-Z}) = 1 / (1 + e^{-1}) = 0.731$

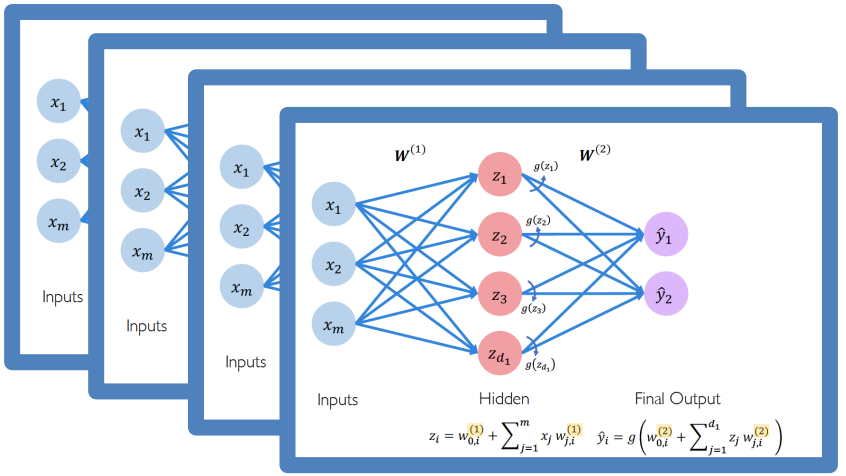
Trainable parameters
W (i) = Weights, b = bias
1 weight per connection (link)
1 bias per neuron
Total number of parameters = 3

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)

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

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

Backpropagation Operators

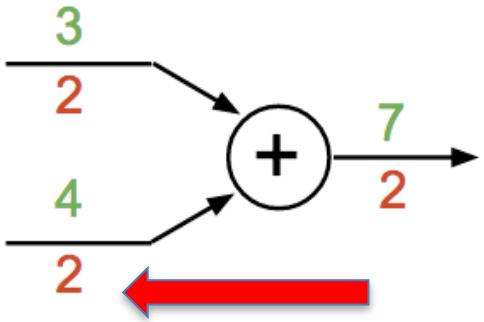
Backward path : Training : using gradient to adjust parameters

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

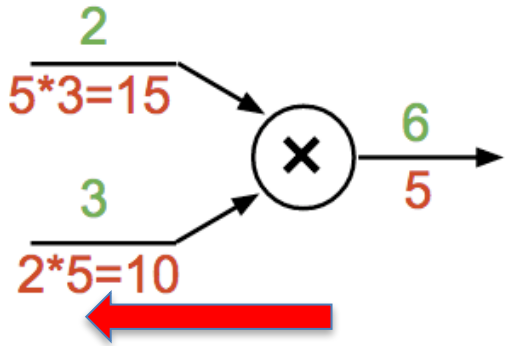
$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

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

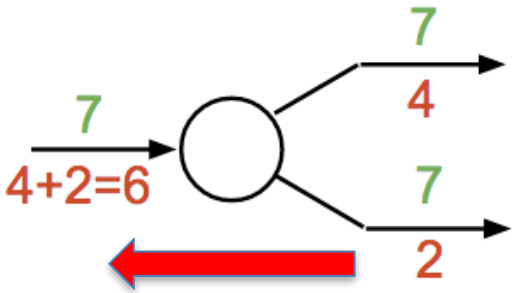
add gate: gradient distributor



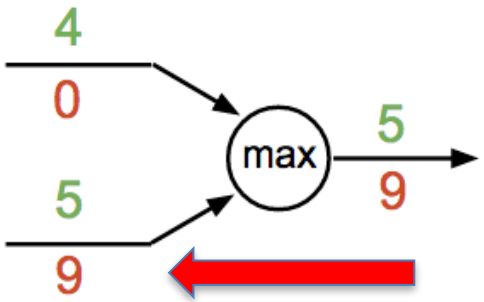
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router

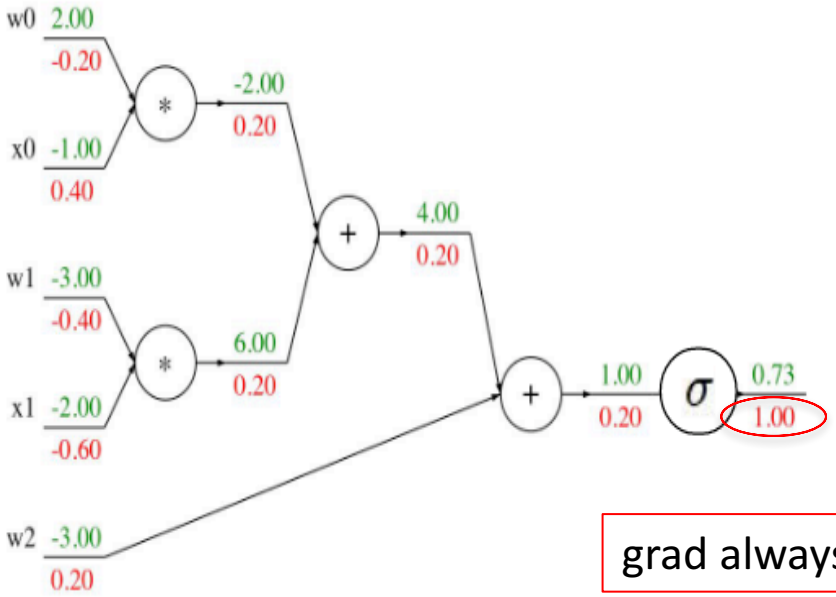


http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

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

#sigmoid function

Import numpy as np

def sigmoid(X):

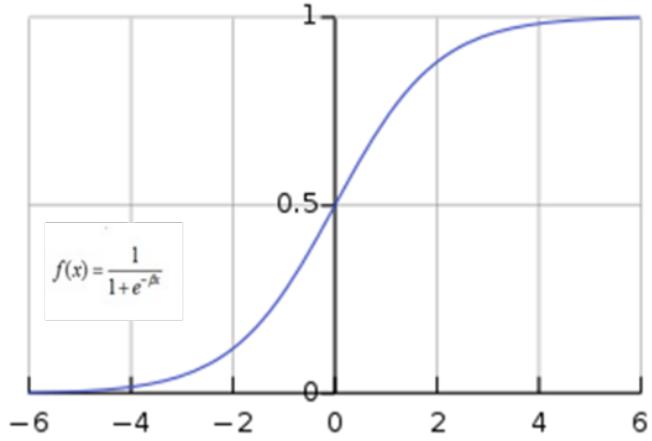
return 1/(1+np.exp(-X))

#Example with mmatrix defined above

sigmoid(mmatrix)

#output:

array([[0.73105858, 0.88079708, 0.95257413],
[0.98201379, 0.99330715, 0.99752738]])



#softmax function

def softmax(X):

expo = np.exp(X)

expo_sum = np.sum(np.exp(X))

return expo/expo_sum

#Example with mmatrix defined above

print (softmax(mmatrix))

#output:

[[0.00426978 0.01160646 0.03154963] [0.08576079
0.23312201 0.63369132]]

#ReLu function

def relu(X):

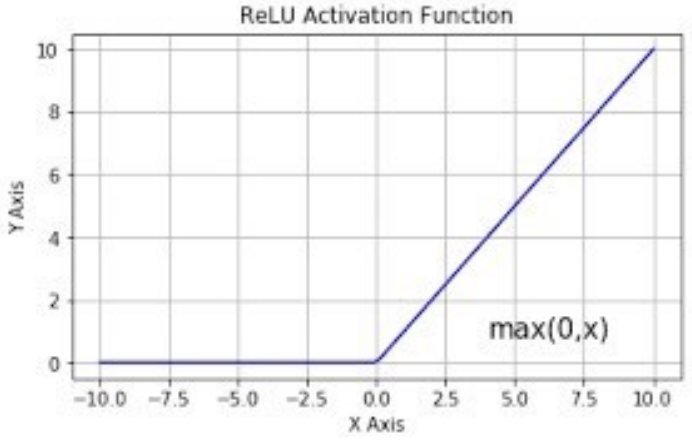
return np.maximum(0,X)

#Example with mmatrix defined above

relu(mmatrix)

#output:

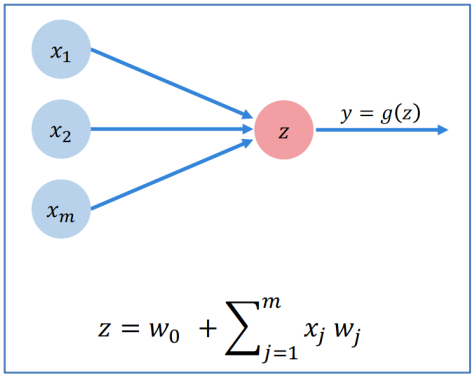
array([[1, 2, 3], [4, 5, 6]])



https://www.codegrepper.com/code-examples/python

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

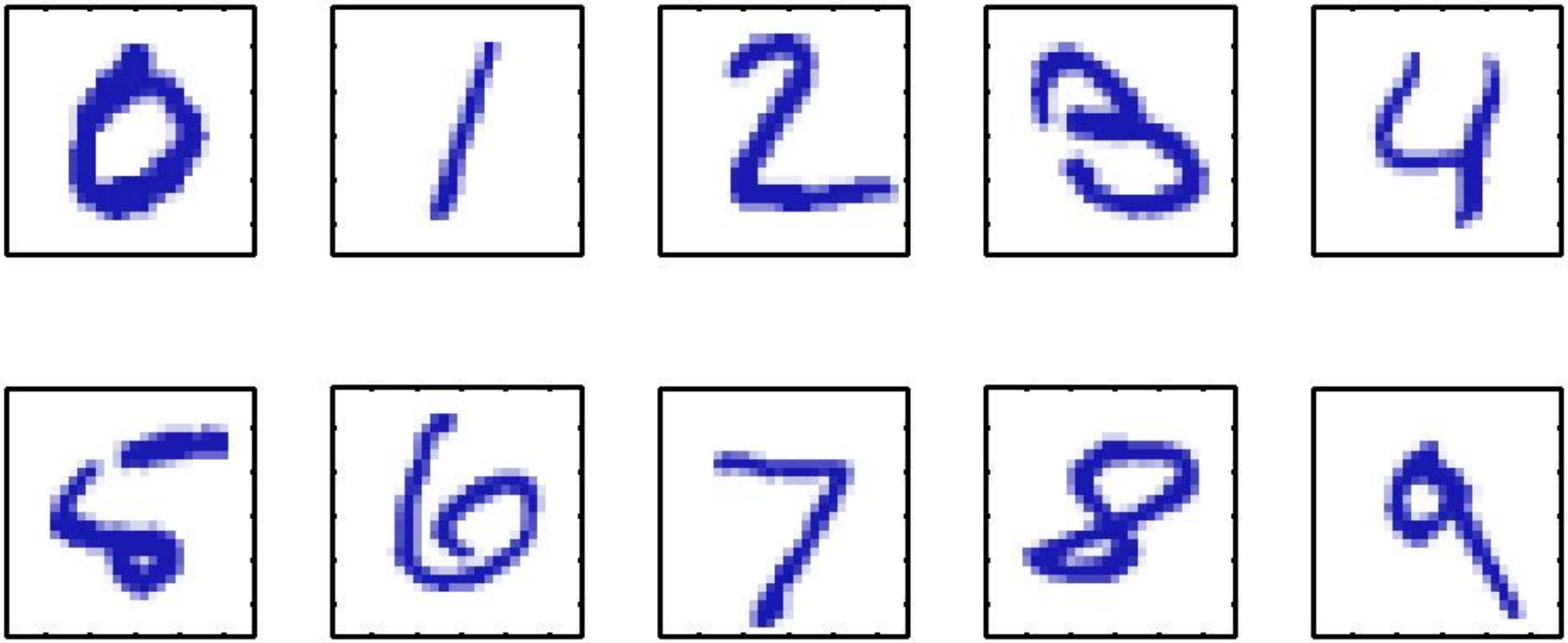
add operator :
Downstream gradient =
upstream gradient

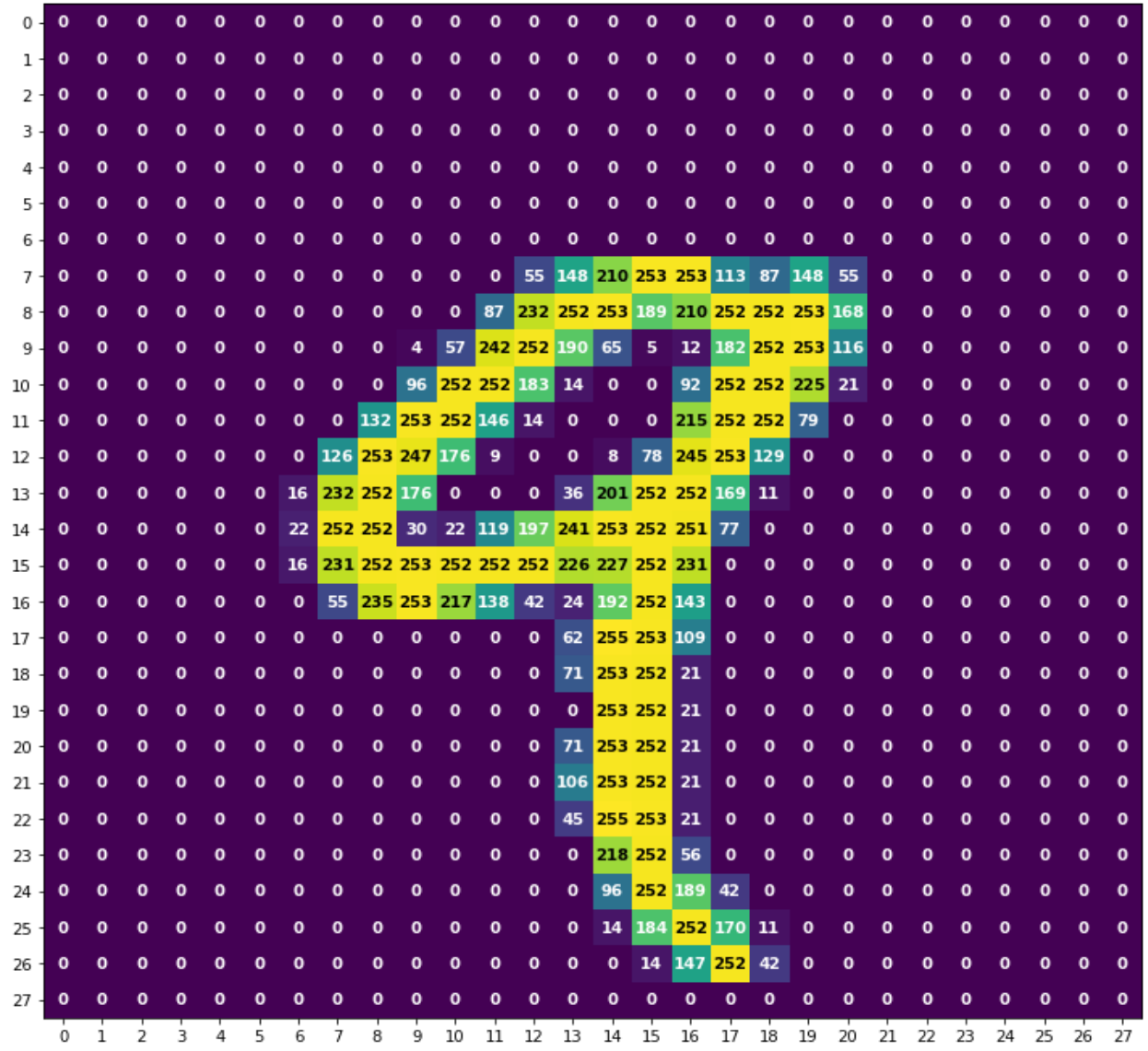
function operator :
downstream gradient =
Upstream gradient * local function gradient

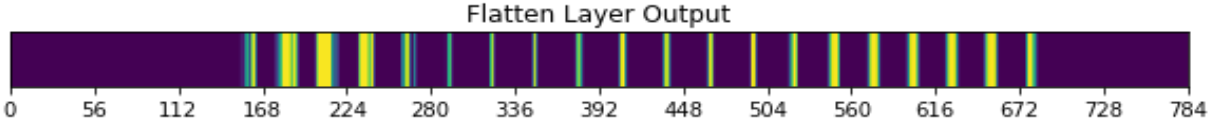
Example: how can computer see images?

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

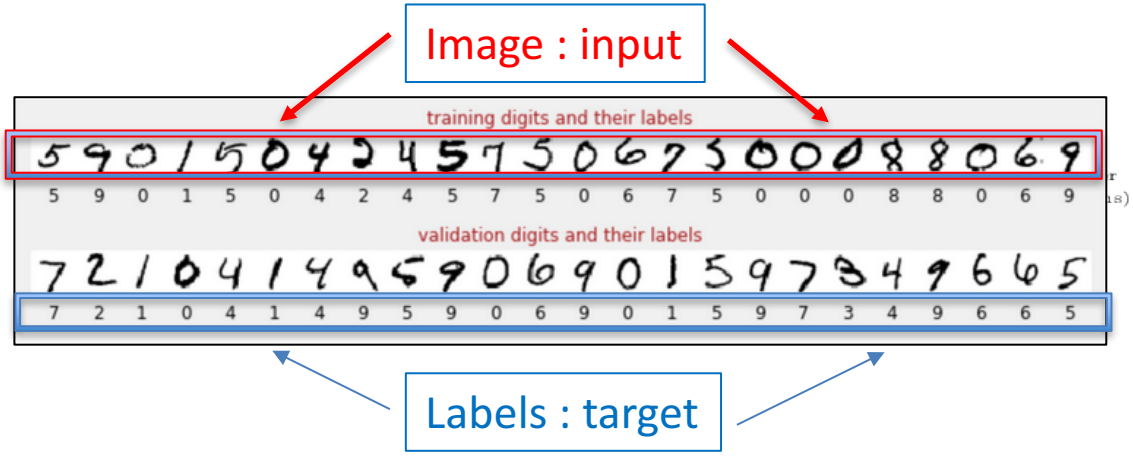
28 x 28 matrix

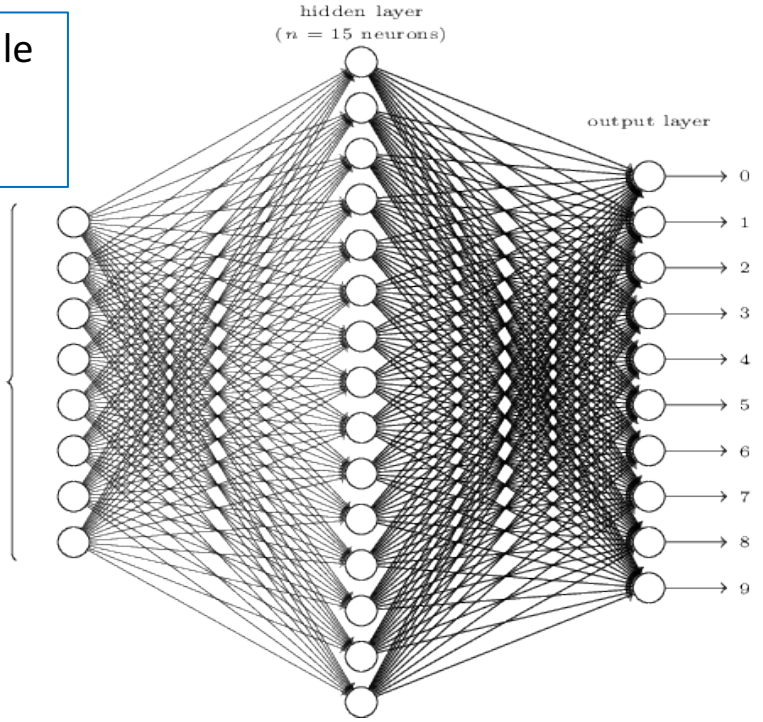
to a vector of
28 x 28 elements

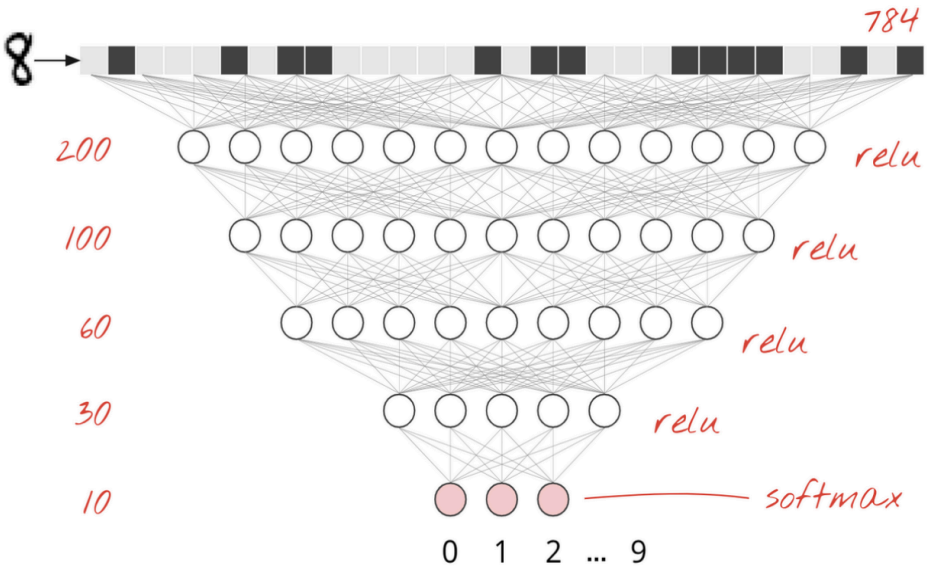
784 elements

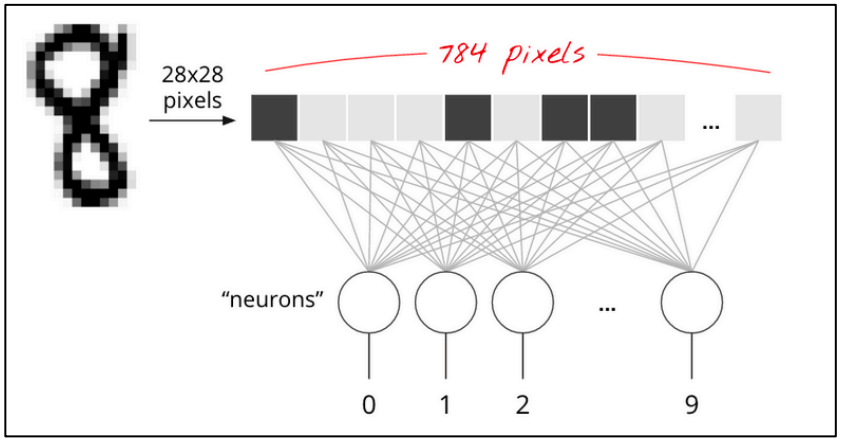
MNIST Example (28x28 pixels)

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.



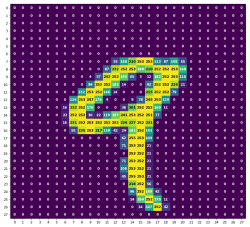






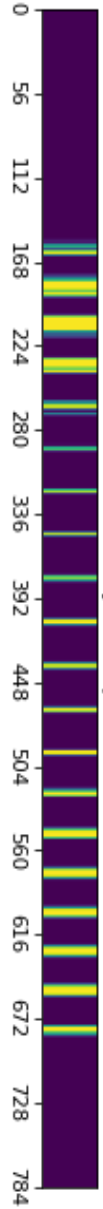
·Flow, Keras and deep learning, without a PhD
https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist#0

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

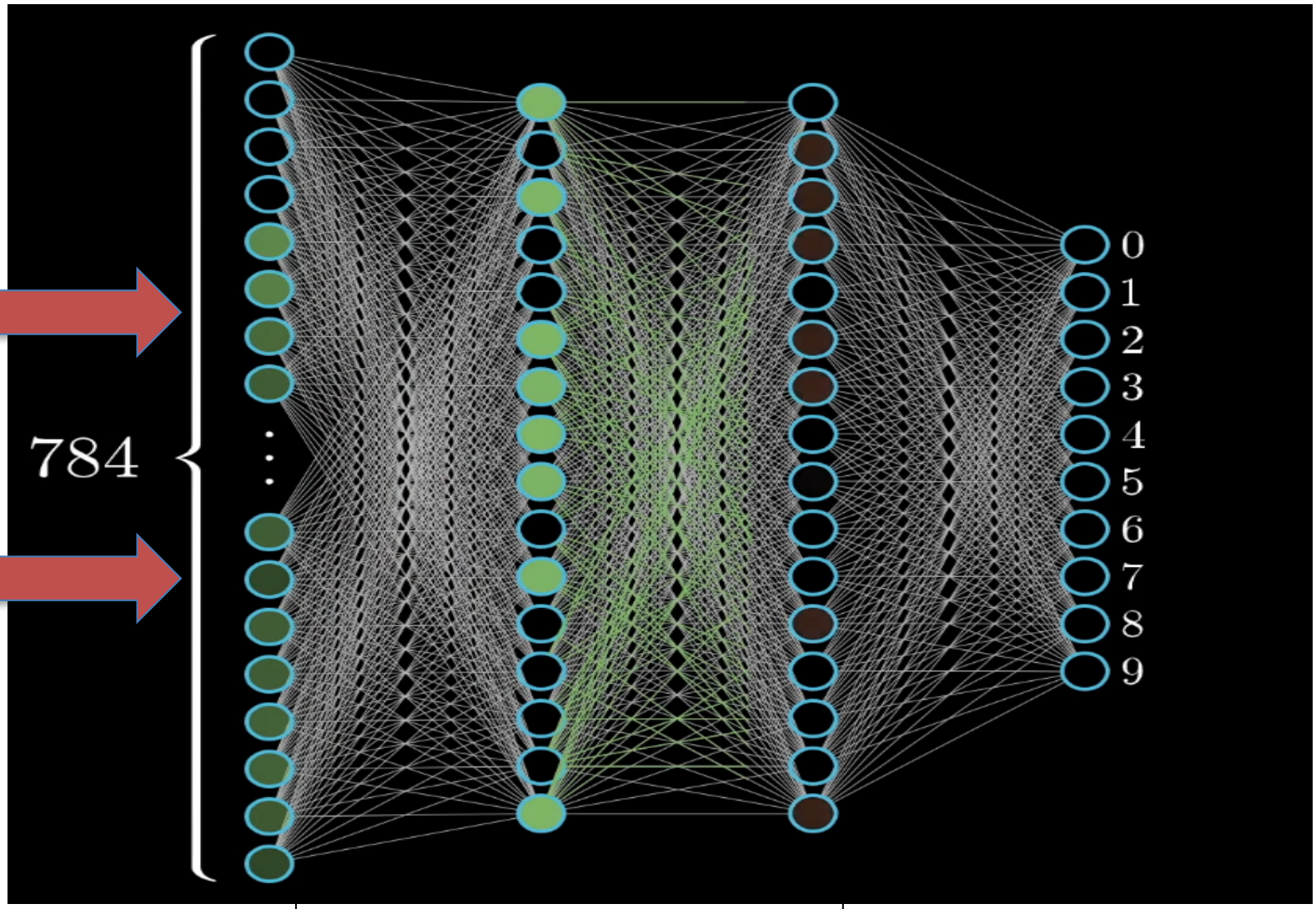


Image:https://www.3blue1brown.com/

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 : Y the end of the NN

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

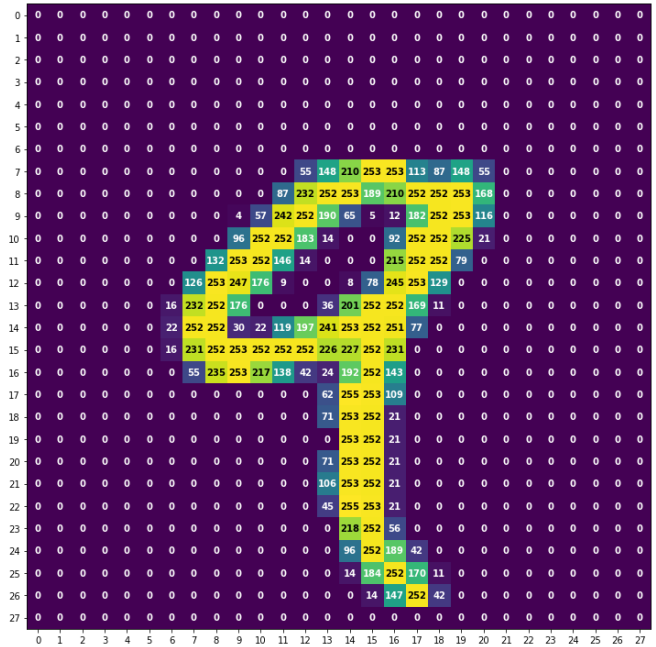
TensorFlow 2.0

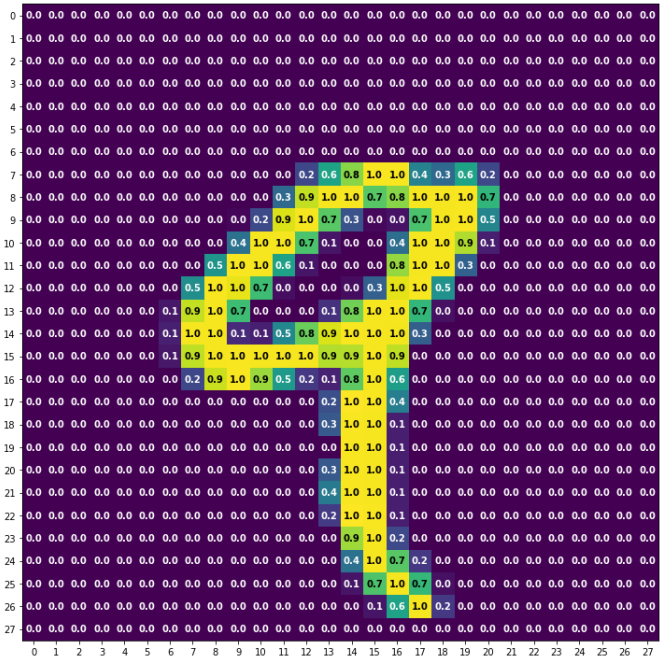
Loading and Preparing the MNIST Dataset

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
#download the images
(x_train, y_train), (x_test, y_test) =
mnist.load_data()
x_train, x_test = x_train / 255.0,
x_test / 255.0 #normalize
```

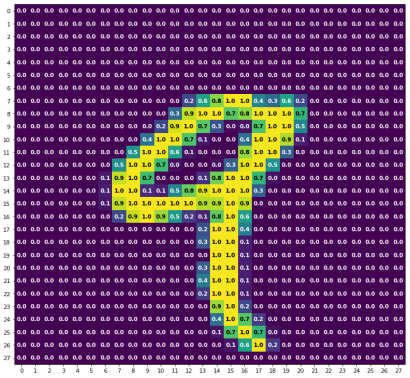
Each Image is a 28x28 image of a handwritten digit

x 60,000 images





TensorFlow 2.0 : Forward Pass



```
tf.reshape(tensor, shape)
#example
tf.reshape(x_train[0], [784])
```

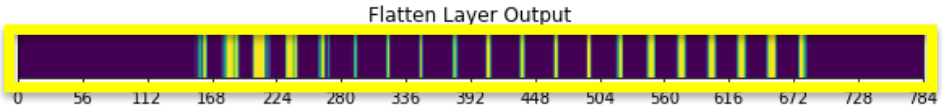
Flatten a single image

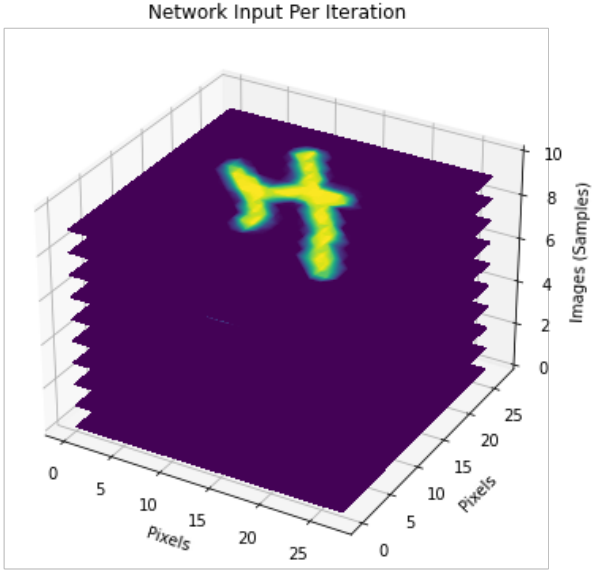
```
tf.reshape(tensor, shape)
#example
tf.reshape(x_train[0:10], [10: 784])
```

Flatten a batch of images

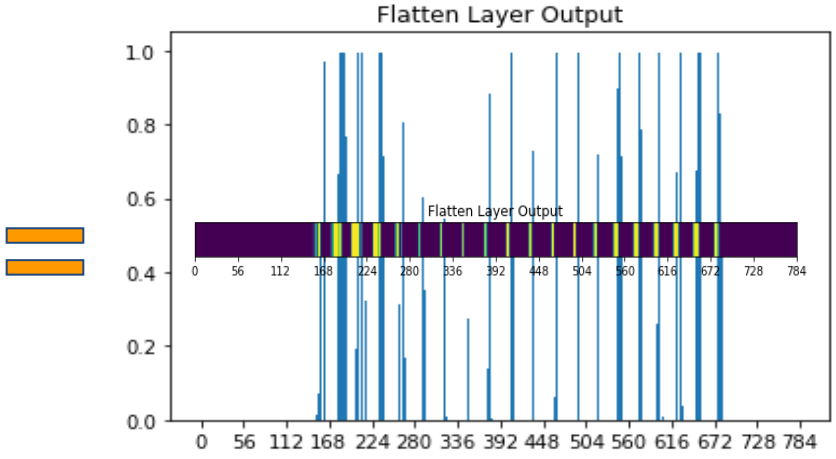
Flatten

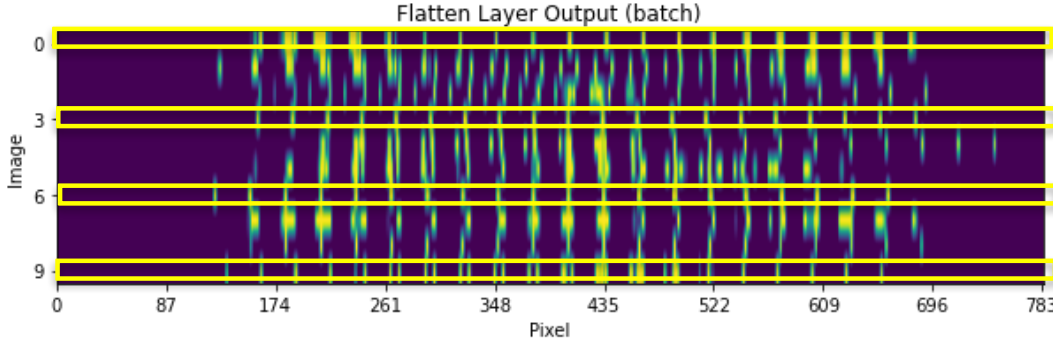
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28))
```





=

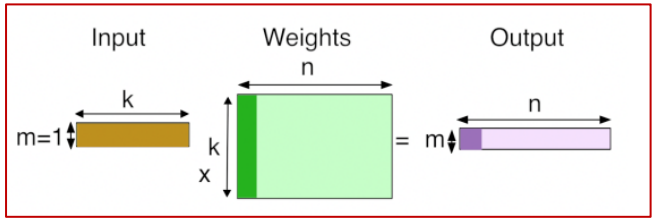




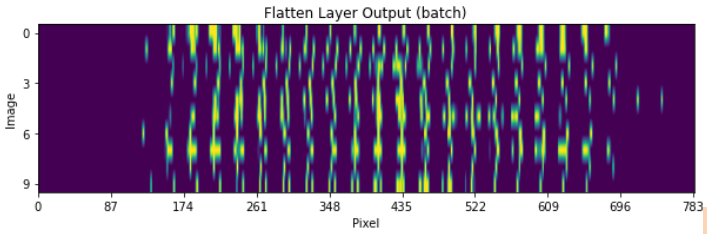
TensorFlow 2.0 : Forward Pass

1ST Fully Connected Layer with 16 neurons
Matrix multiplication

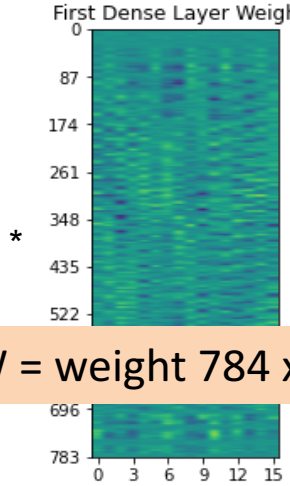
```
...
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(16, activation='relu'),
...
```



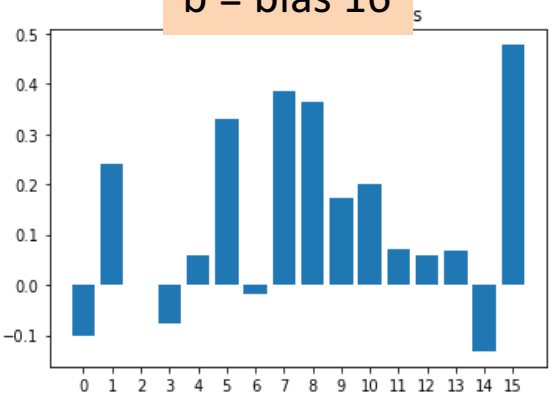
$$a^1 = \text{reLU}(X * W^1 + b^1)$$



X = Input 10 x 784



W = weight 784 x 16

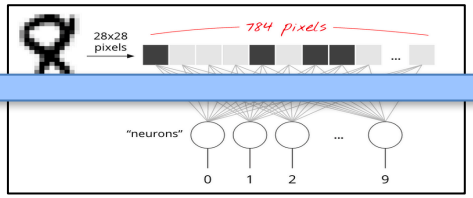


b = bias 16

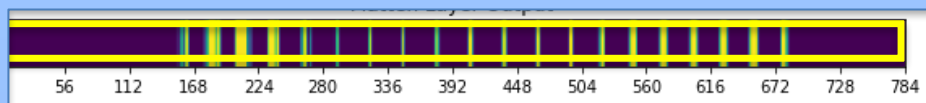
Weights and bias are initialized with random values

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



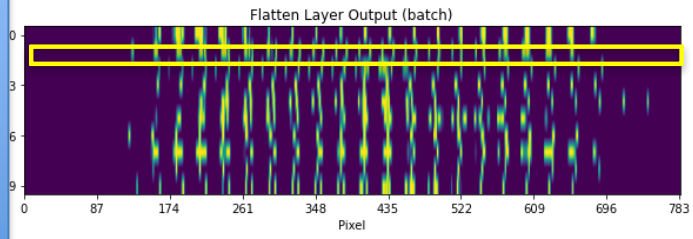
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

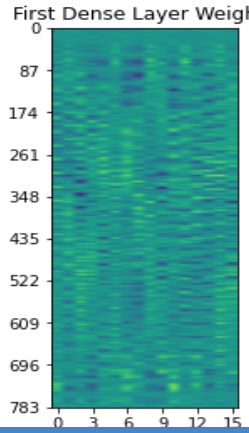
16 neurons
784 x 16 W matrix

10 x 16 output matrix

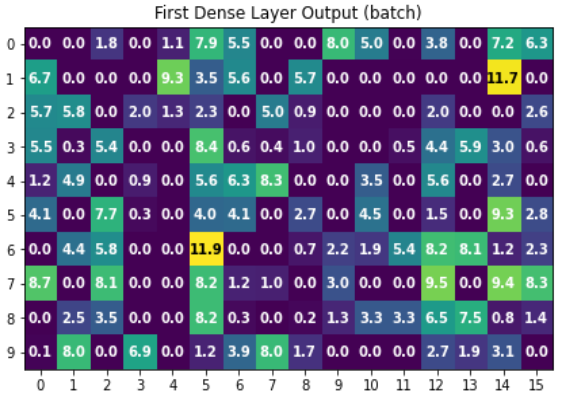
Input to 2nd layer



*



=

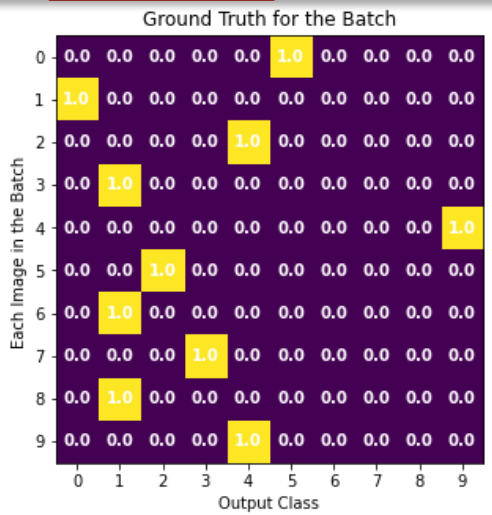


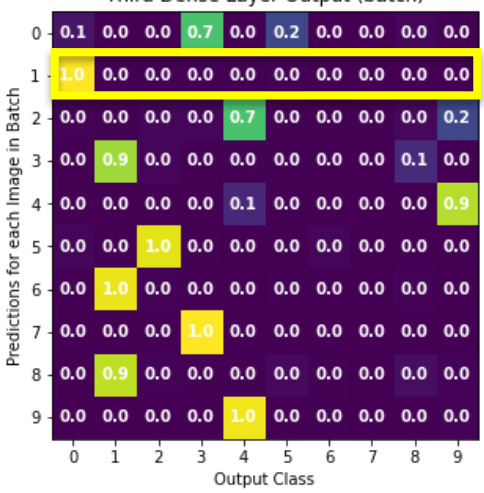
10 x 16 matrix

10 x 10 labeled matrix

10 x 10 output matrix

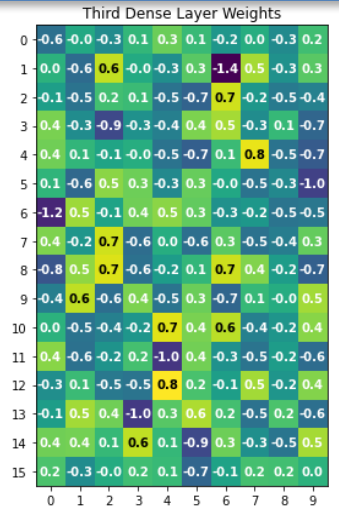
*





=

10 x 10 output matrix



16 neurons
16 x 10
W matrix

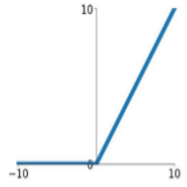
TensorFlow 2.0 : Forward Pass

Second Fully Connected Layer with 16 neurons

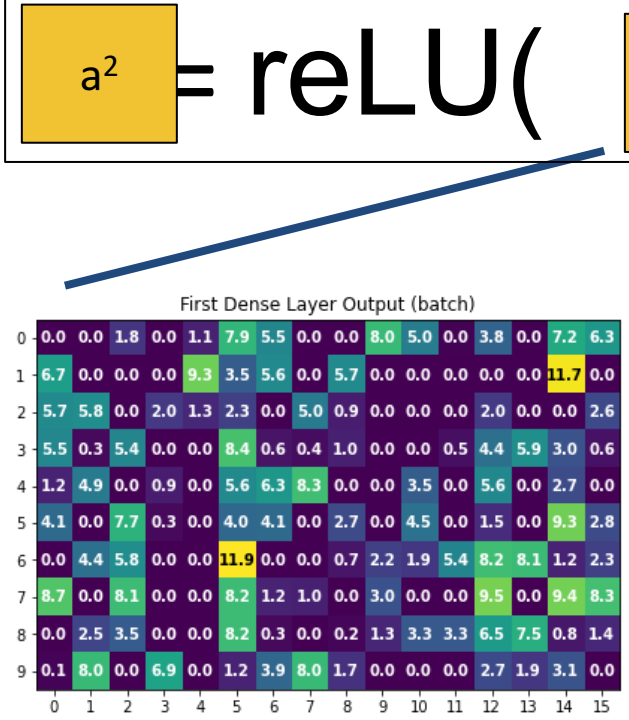
```
model = tf.keras.models.Sequential([
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(16, activation='relu'),
 tf.keras.layers.Dense(16, activation='relu'),
...
```

ReLU

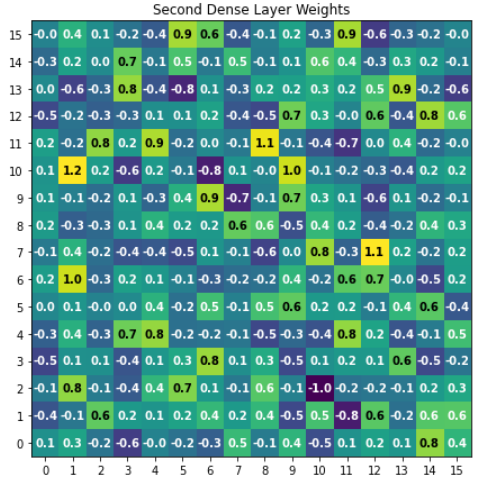
$\max(0, x)$



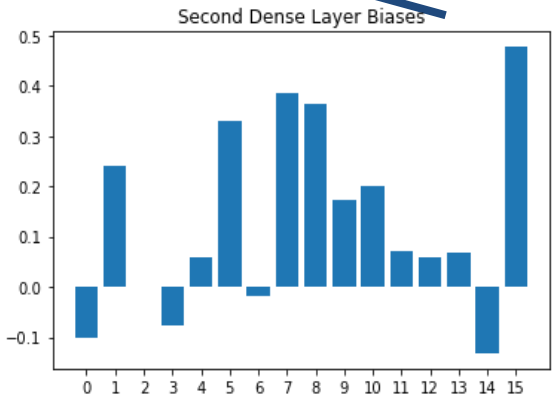
$a^2 = \text{reLU}(a^1 * W^2 + b^2)$



*



+



TensorFlow 2.0 : Forward Pass

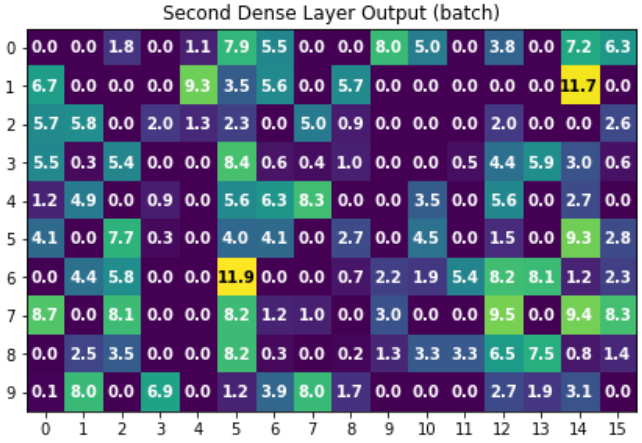
Final Fully Connected Layer with 10 neurons : 10 classes

$$\text{Softmax} = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

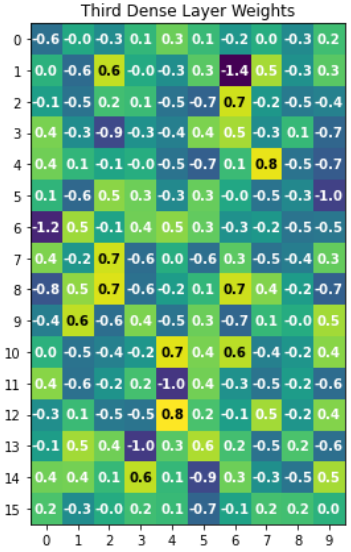
- ✓ The softmax function is a function that turns a vector of K real values into values between 0 and 1, and the values sum up to 1, so that they can be interpreted as probabilities.
- ✓ This is because the softmax is a generalization of logistic regression that can be used for multi-class classification.

https://deepai.org/machine-learning-glossary-and-terms/softmax-layer

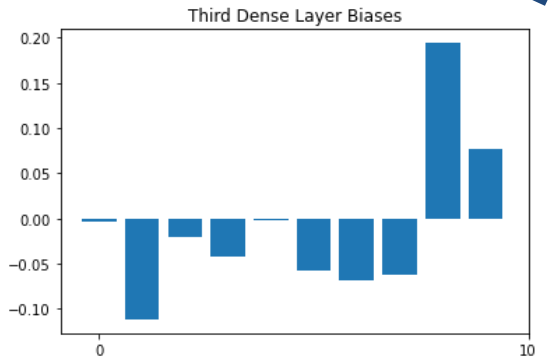
$a^3 = \text{softmax}(a^2 * W^3 + b^3)$



*

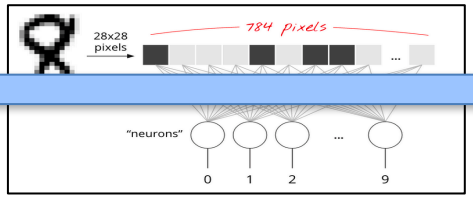


+

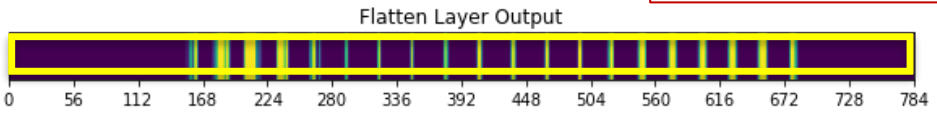


Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



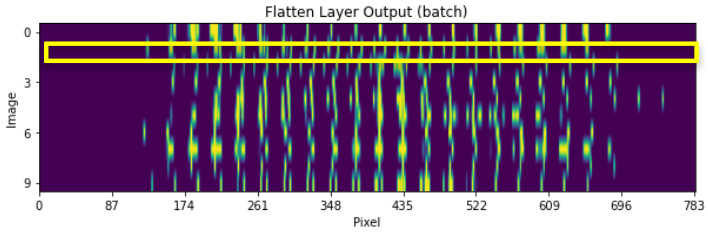
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

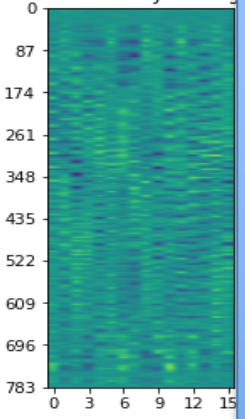
16 neurons
784 x 16 W matr

10 x 16 output matrix

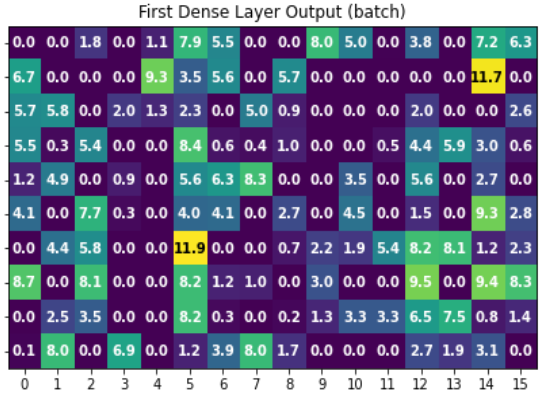
Input to 2nd layer



*



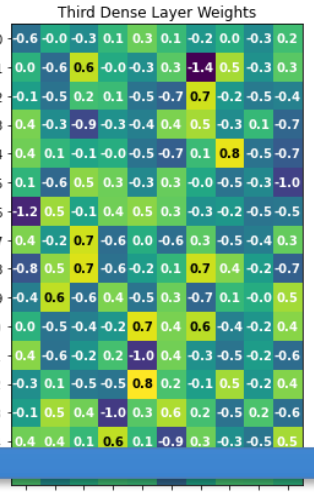
=



Input to 2nd layer

10 x 16 matrix

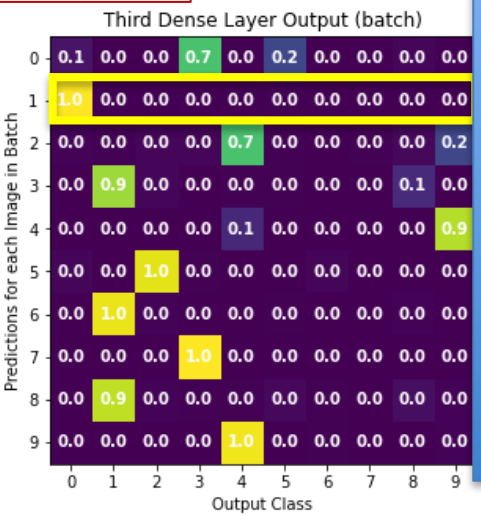
*



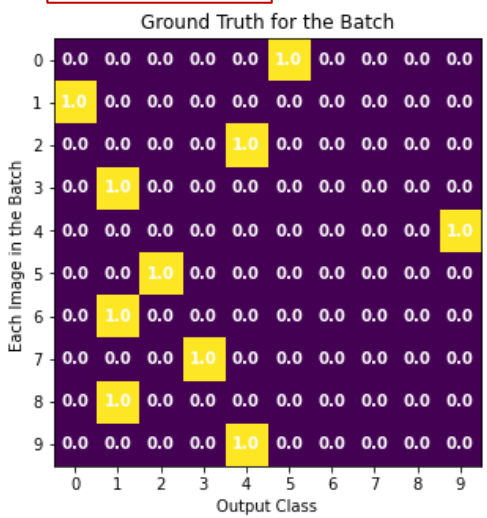
16 neurons
16 x 10 W matrix

=

10 x 10 output matrix

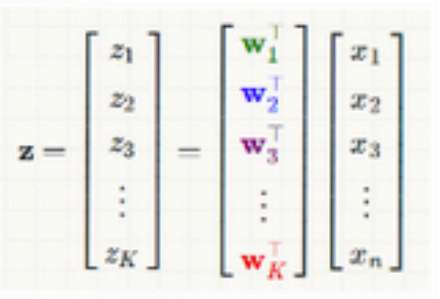


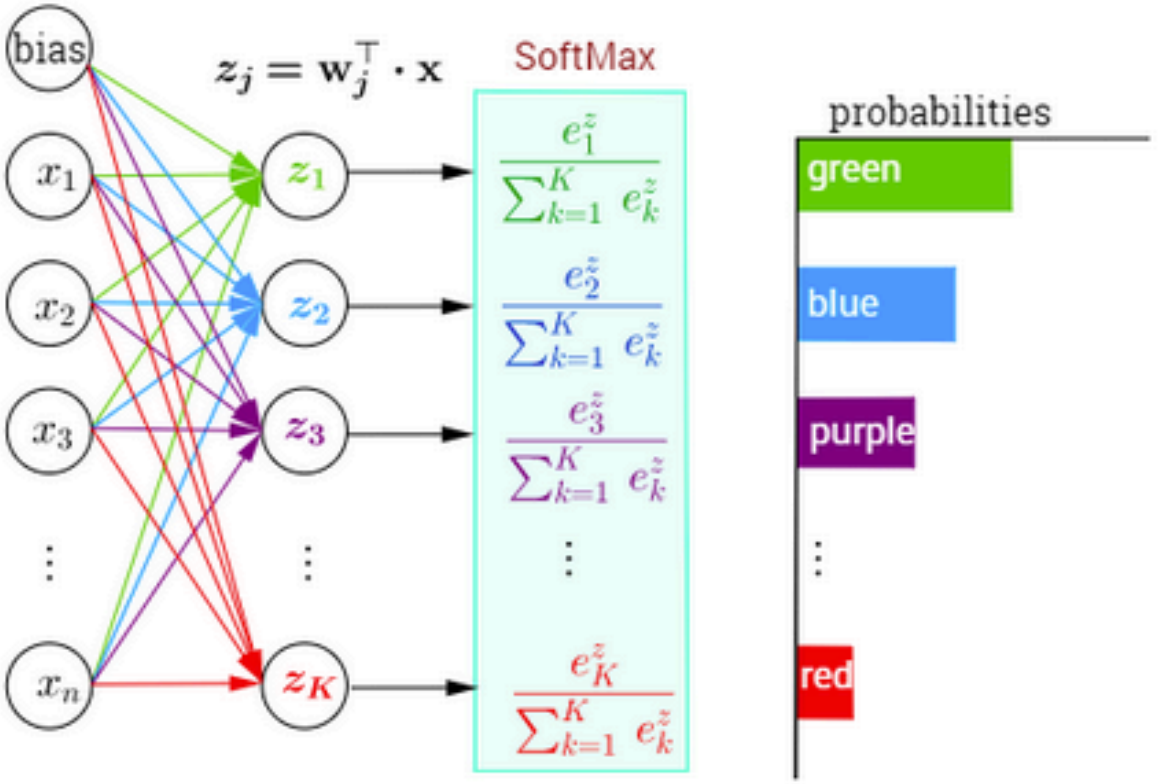
10 x 10 output matrix



10 x 10 labeled matrix

Multi-Class Classification with NN and SoftMax Function

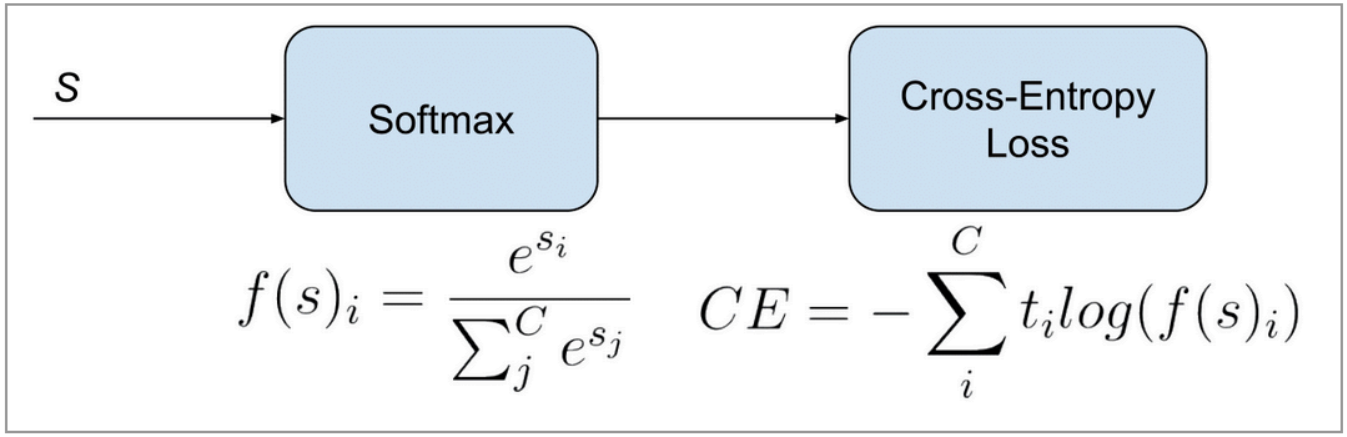




Softmax Function

$$\sigma(j) = \frac{\exp(\mathbf{w}_j^\top \mathbf{x})}{\sum_{k=1}^K \exp(\mathbf{w}_k^\top \mathbf{x})} = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)}$$

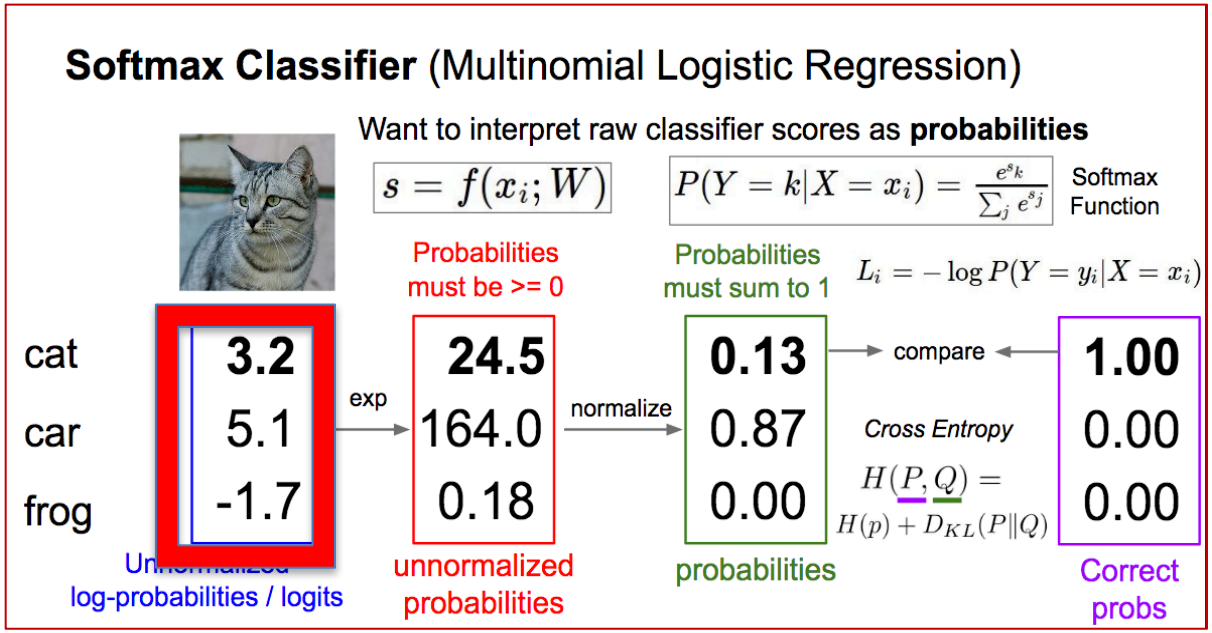
Categorical (Softmax) Cross Entropy Loss (Statistical Learning)



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

T = 1

ti and si are the groundtruth (label) and the computed score for each class i in C.



https://gombru.github.io/2018/05/23/cross_entropy_loss/

e^3.2 = 24.5

e^5.1 = 164.0

e^-1.7 = 0.18

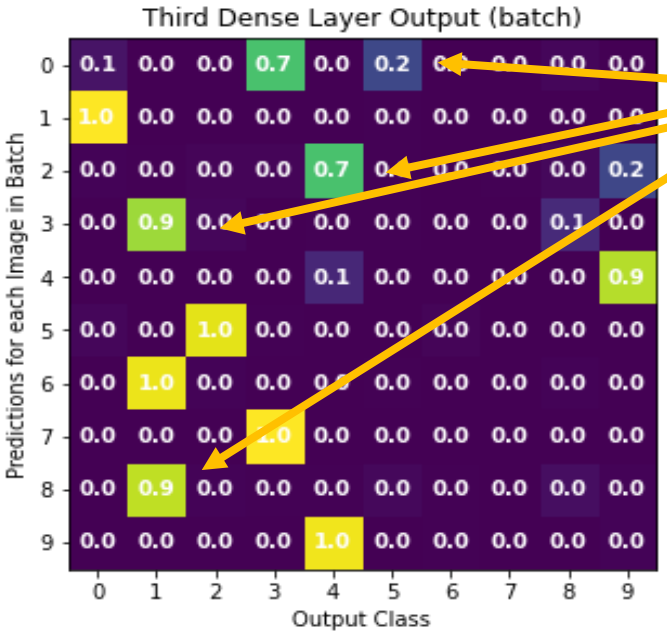
188.68

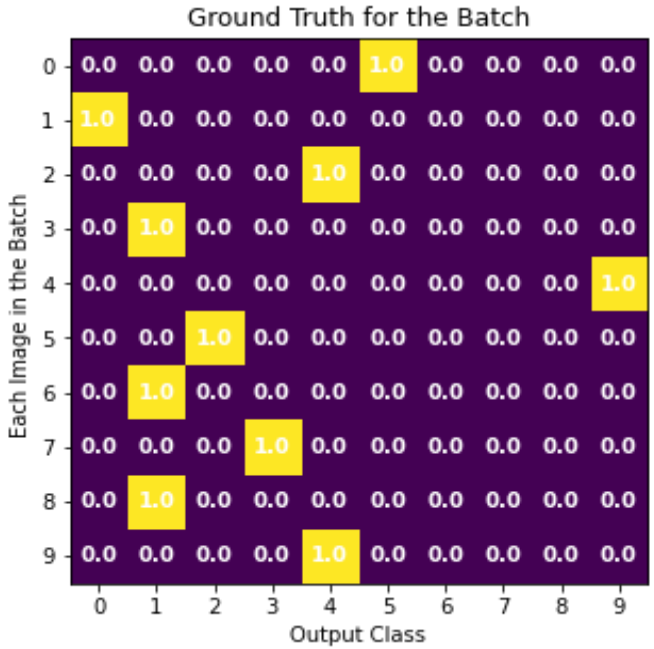
Softmax = 24.5 / 188.68 = 0.13

Softmax = 164 / 188.68 = 0.87

Softax = -1.7 / 188.68 = 0.00

Evaluating the Error





Z

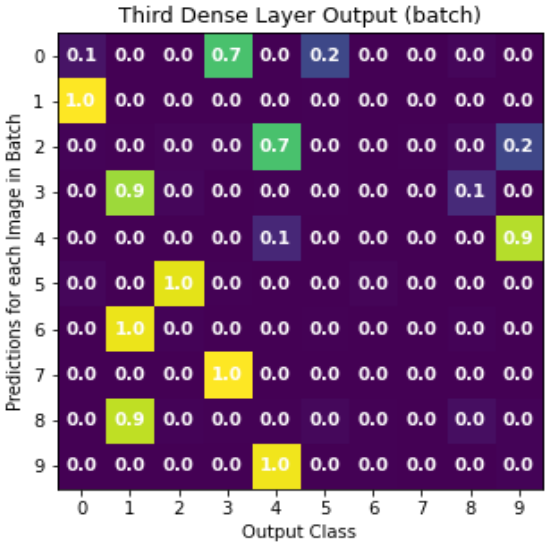
Categorical Cross Entropy Loss

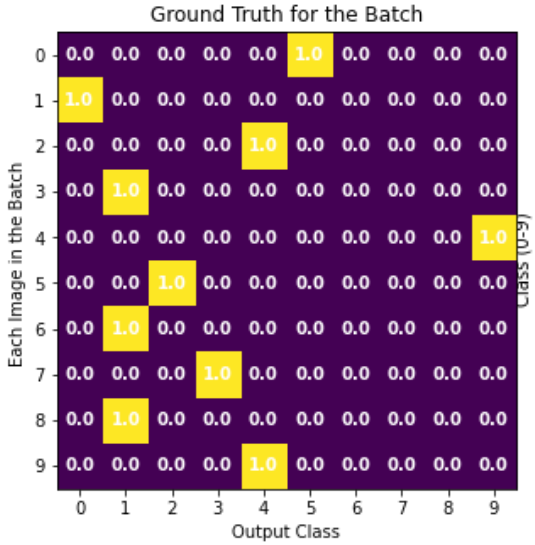
$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

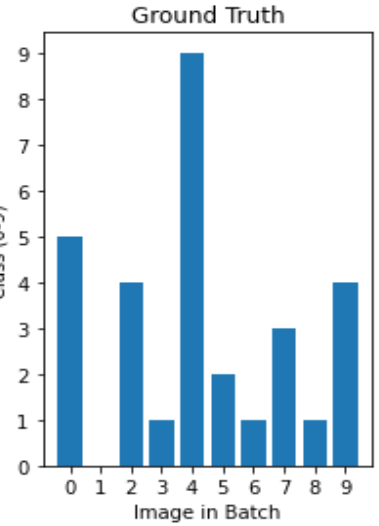
L (i) = - log0.2 = 1.61	Image 0 = 5
- log1.0 = 0	Image 1 = 0
- log0.7 = 0.356	Image 2 = 4
- log0.9 = 0.105	Image 3 = 1
- log0.9 = 0.105	Image 4 = 9
- log1.0 = 0	Image 5 = 2
- log1.0 = 0	Image 6 = 1
- log1.0 = 0	Image 7 = 3
- log0.9 = 0.105	Image 8 = 2
- log1.0 = 0	Image 9 = 4

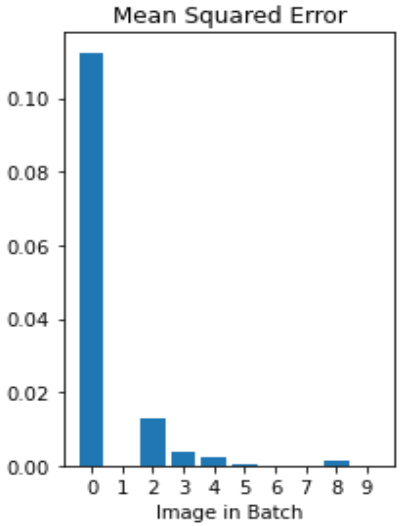
Evaluating the Error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$









Categorical Cross Entropy Loss

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right)$$

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax Function

$$L_i = -\log P(Y = y_i|X = x_i)$$

	Probabilities must be >= 0	exp	Probabilities must sum to 1	normalize		compare	
cat	**3.2**		**24.5**		**0.13**		**1.00**
car	5.1		164.0		0.87		0.00
frog	-1.7		0.18		0.00		0.00
	Unnormalized log-probabilities / logits		unnormalized probabilities		probabilities		Correct probs

Cross Entropy

$$H(P, Q) = H(p) + D_{KL}(P\|Q)$$

L (i) = - log0.2 = 1.61
- log1.0 = 0
- log0.7 = 0.356
- log0.9 = 0.105
- log0.9 = 0.105
- log1.0 = 0
- log1.0 = 0
- log1.0 = 0
- log0.9 = 0.105
- log1.0 = 0

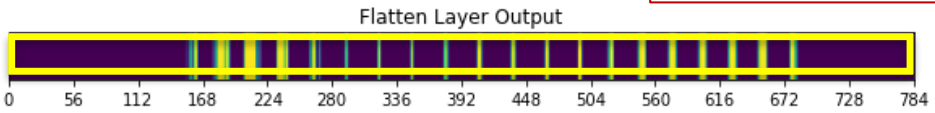
Tensorflow Example

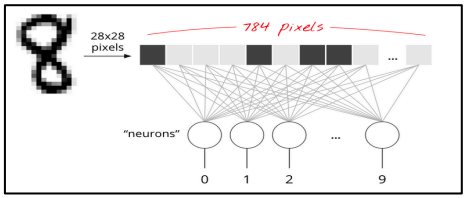
The model is trained in about 10 seconds completing 5 epochs with a Nvidia GTX 1080 GPU and has an accuracy of around 97-98%

```
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
```

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784

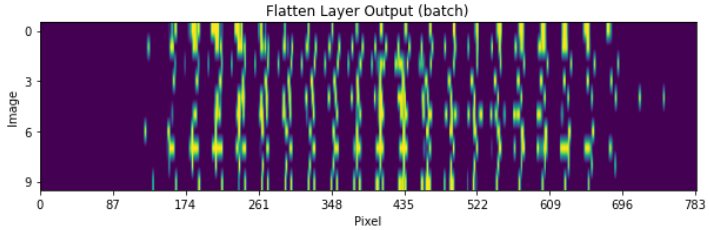




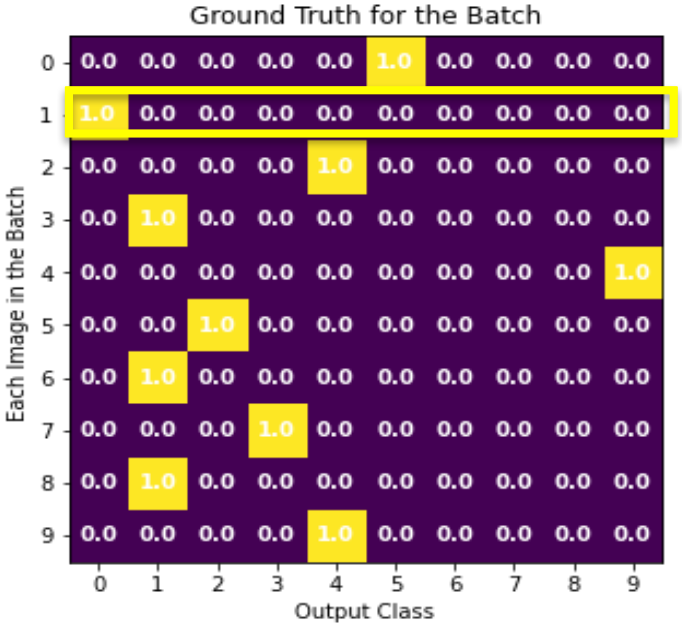
An image of 28x28 pixels

Input to 1st layer

10 images : batch of 10
10 x 784 matrix

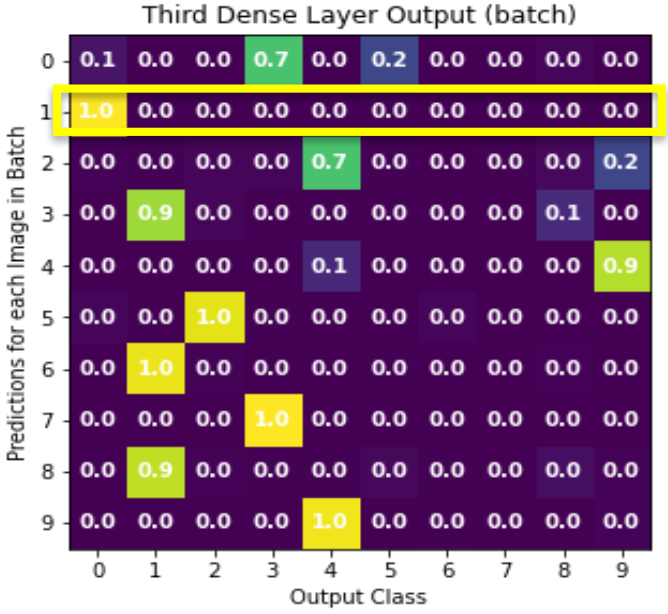


10 x 10 labeled matrix



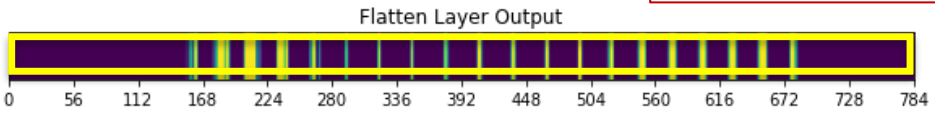
Comparing Computed NN results To Labeled results (target)

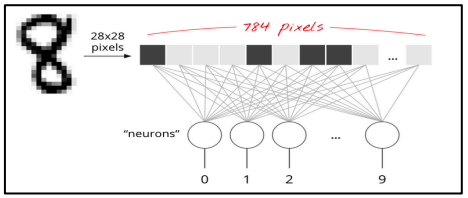
10 x 10 output matrix



Summary : Flow of MLP Dense layer NN

1 image = a vector of 784





An image of 28x28 pixels

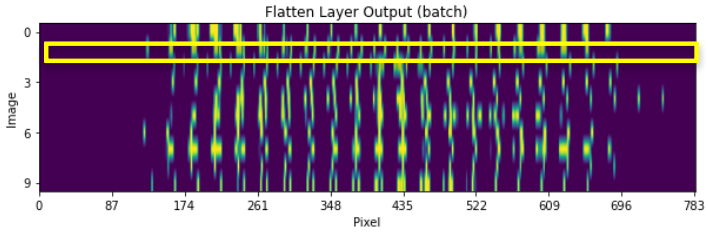
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

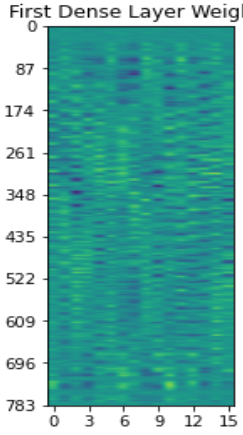
16 neurons
784 x 16 W matrix

10 x 16 output matrix

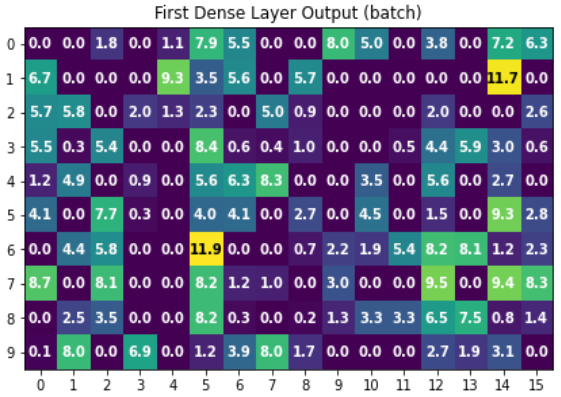
Input to 2nd layer



*

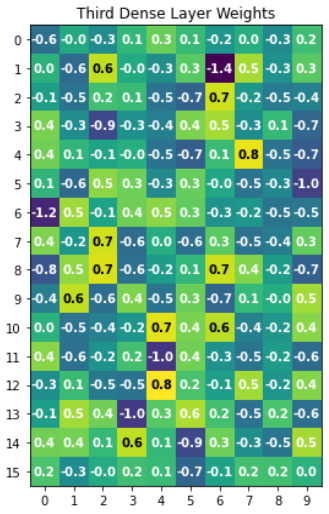


=



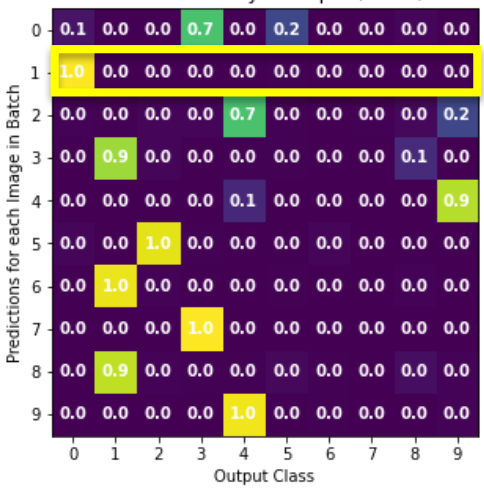
10 x 16 matrix

*

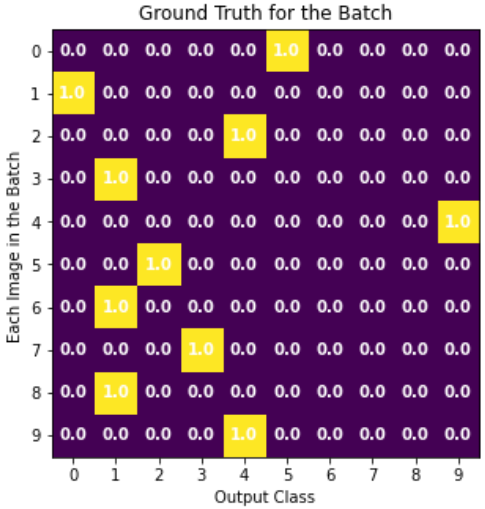


16 neurons
16 x 10
W matrix

=

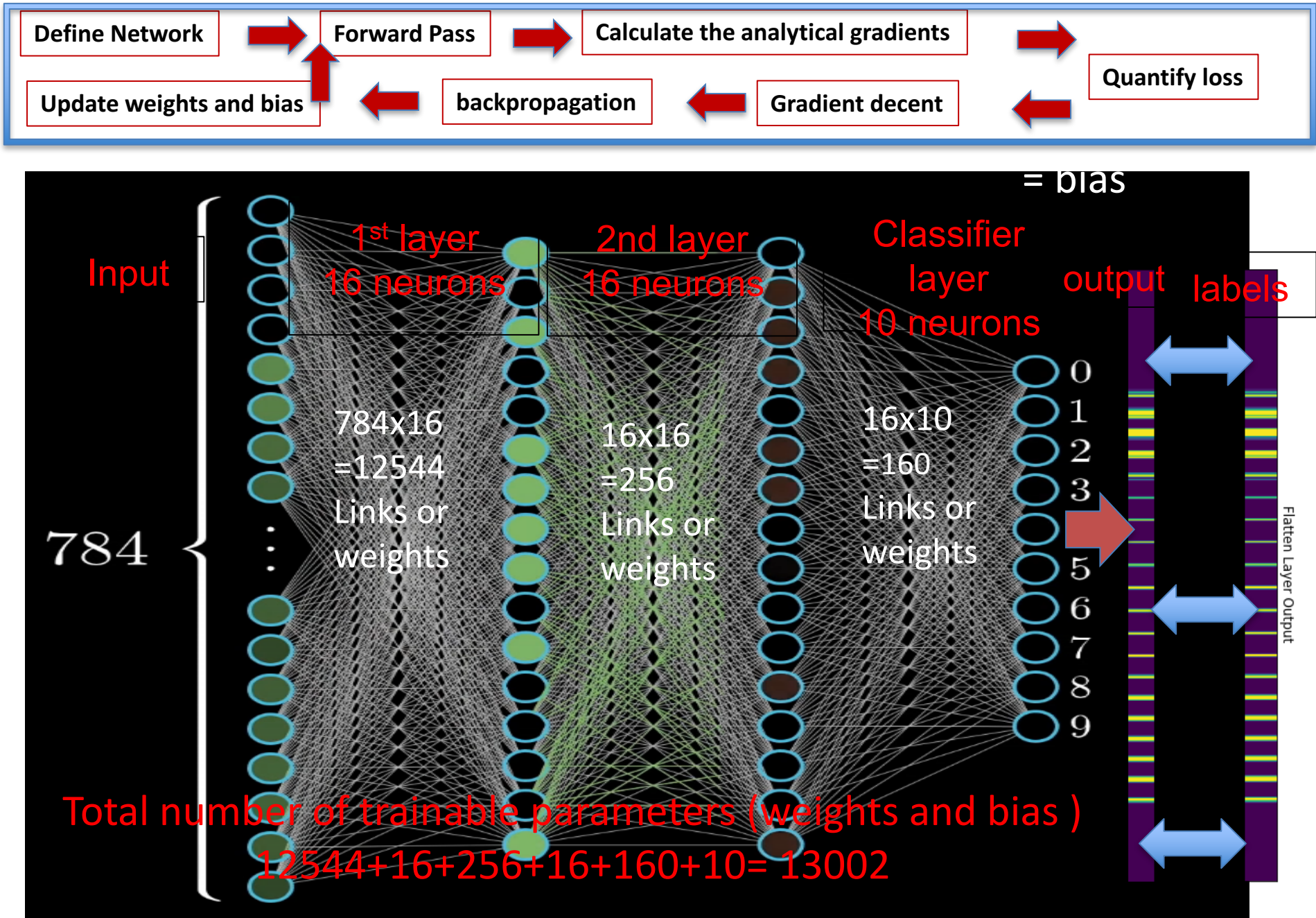


10 x 10 output matrix



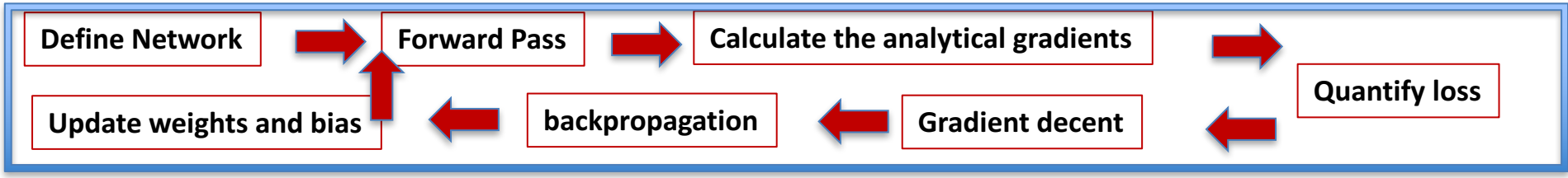
10 x 10 labeled matrix

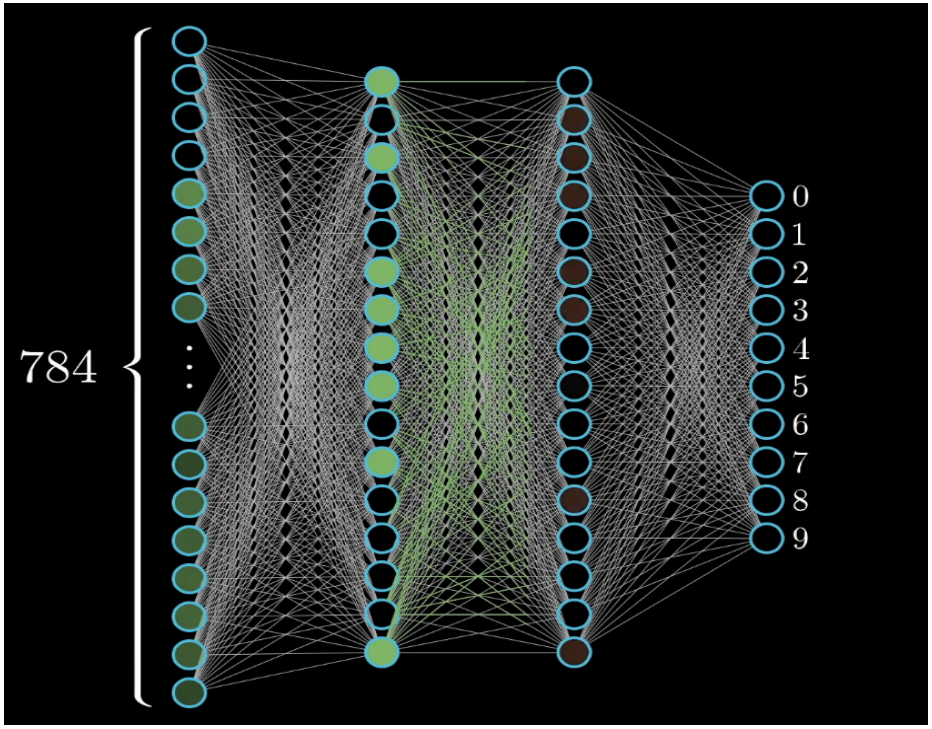
```
model.fit(x_train, y_train, epochs=3, batch_size=10)
```



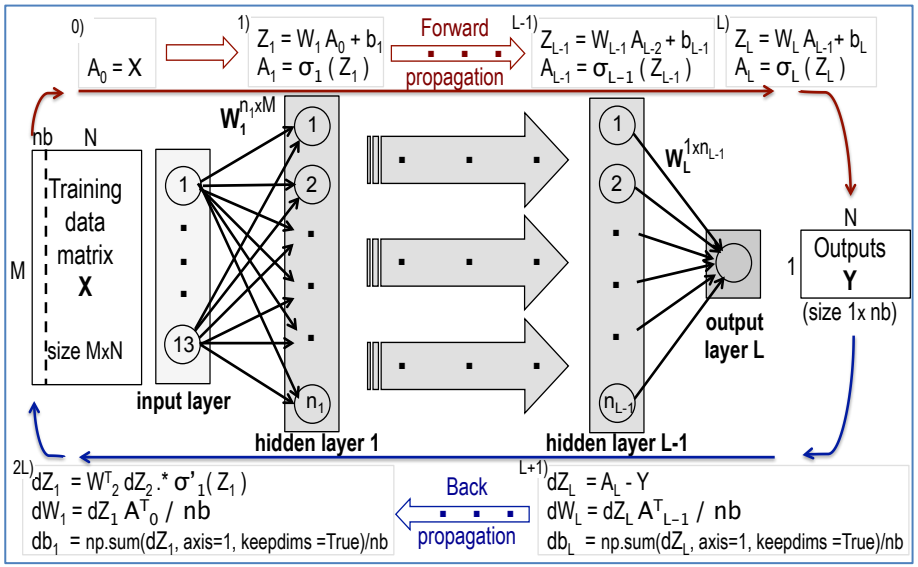
Summary : Flow of NN

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





- ✓ **Batch Size**: how many images do we pass through the network for each iteration (10)
- ✓ **Epochs**: how many times we pass over the entire dataset (3)
- ✓ **# Iterations per Epoch** = number of images / batch size (60,000/10=6,000)



https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0

Training a Neural Network

"Training a neural network"
What does it mean?
What does the network train?
What are the training parameters
How does it work?

- ✓ Neural network is composed of many unknown parameters (weights and bias), initialize them randomly in the beginning
- ✓ Perform forward path computations
- ✓ Computed results are compared to ground truth (labels) and errors are calculated → Need to have a cost (loss) function for error evaluation
- ✓ Use the error to adjust the weights and bias (backpropagation)
- ✓ Repeated the process (training) until the model is good enough

Basic Ideas

Typical Neural Network – MLP

Convolutional Neural Network (Unit 3)

STEP 1 : Model Definition

STEP 2 : Cost Function

Step 3 : Optimization Scheme

Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

Training

"Training a neural network"

What does it mean?

What does the network train?

minimize error of the computed values against the labeled values (loss function)

What are the training parameters

W and b

How does it work?

optimization based on gradient descent algorithm

go back and forth in the NN model to adjust for w and b => need derivatives w.r.t. w and b

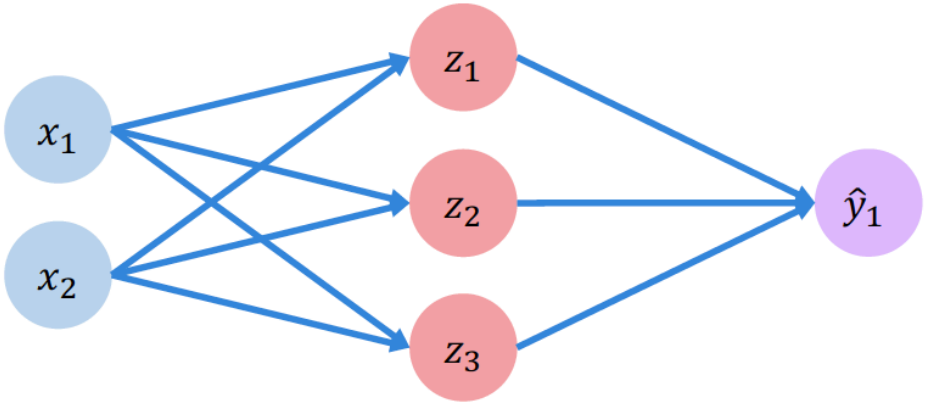
Go through forward calculation

training with backpropagation

Mean Squared Error Loss

Mean squared error loss *can be used with regression models that output continuous real numbers*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$	y
30	90
80	20
85	95
⋮	⋮

Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

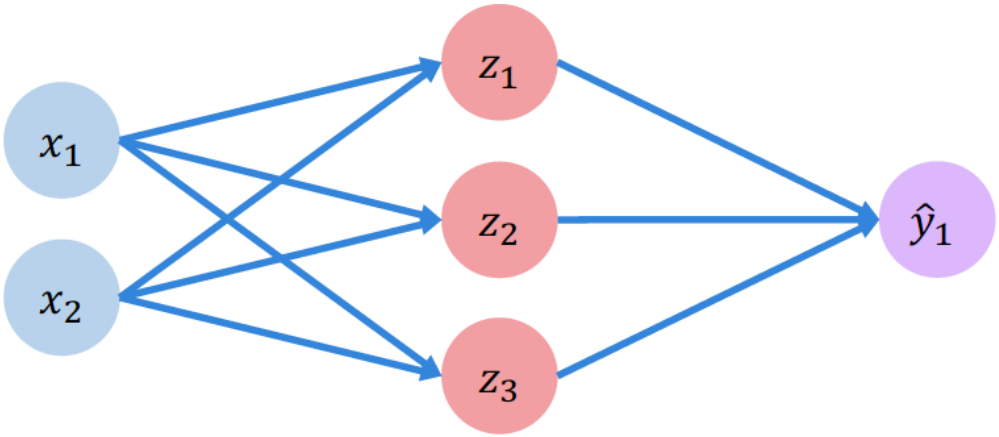
```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Derivatives and Gradient descent

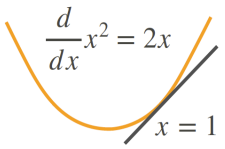
Review Scalar Derivative

y	a	x^n	$\exp(x)$	$\log(x)$	$\sin(x)$
$\frac{dy}{dx}$	0	nx^{n-1}	$\exp(x)$	$\frac{1}{x}$	$\cos(x)$

a is not a function of x

y	$u+v$	uv	$y=f(u),\ u=g(x)$
$\frac{dy}{dx}$	$\frac{du}{dx}+\frac{dv}{dx}$	$\frac{du}{dx}v+\frac{dv}{dx}u$	$\frac{dy}{du}\frac{du}{dx}$

Derivative is the slope of the tangent line



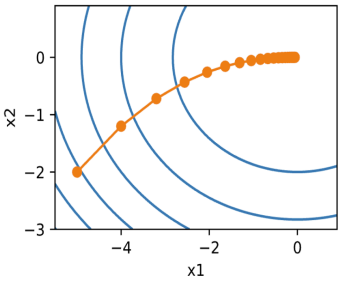
The slope of the tangent line is 2

Algorithm

- Choose initial $\mathbf{x}_0$
- At time $t = 1, \ldots, T$

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

- η is called learning rate



Generalize to Vectors

- Chain rule for scalars:

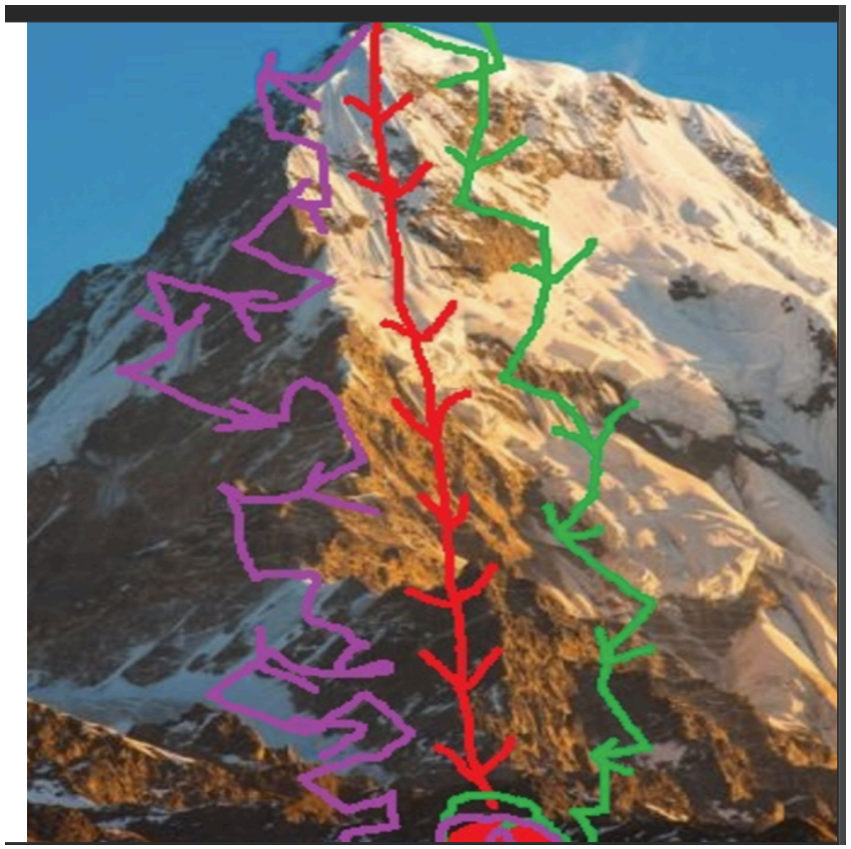
$$y = f(u),\ u = g(x) \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

- Generalize to vectors straightforwardly

$$\frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial \mathbf{x}} \qquad \frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}} \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

$$(1,n)\quad (1,)\ (1,n) \qquad (1,n)\quad (1,k)\ (k,n) \qquad (m,n)\quad (m,k)\ (k,n)$$

Optimization : Derivatives and Gradient Descent



Batch Gradient Descent
Mini-batch Gradient Descent
Stochastic Gradient Descent

Simple story

Derivative is the slope of the tangent line

$$\frac{d}{dx}x^2 = 2x$$

$x = 1$

The slope of the tangent line is 2

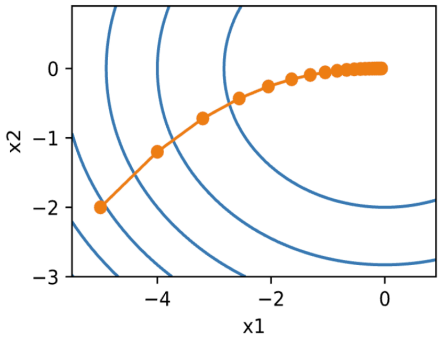
$x)$

Algorithm

- Choose initial $\mathbf{x}_0$
- At time $t = 1, \ldots, T$

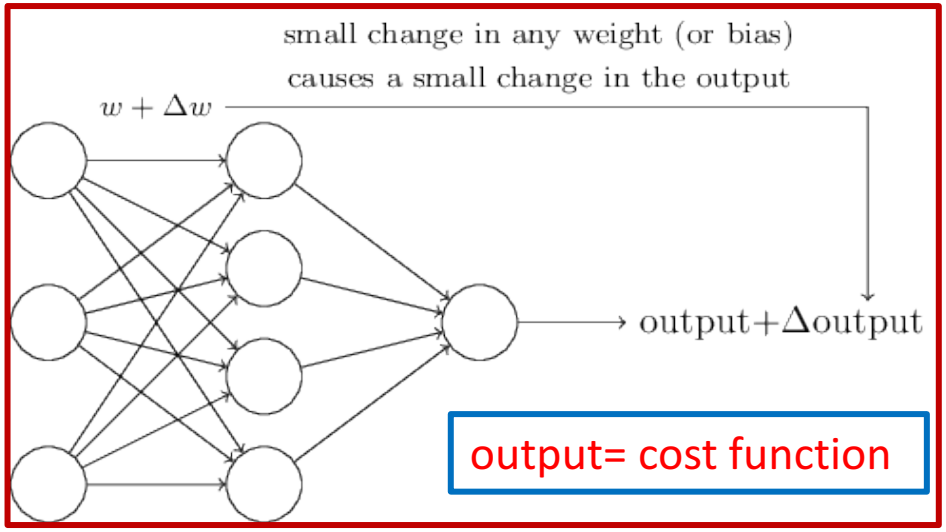
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

- η is called learning rate



Representation of NN Model – Math Story

neuralnetworksanddeeplearning.com



- ✓ Output = cost function = C = evaluator of the model = labeled value - computed value
- ✓ The neural network model is defined by the its connection and the set of parameters, w and b.
- ✓ Small change in the model parameters cause small change in the output (cost) =>
- ✓ Small changes in any weights (Δwj) and bias (Δb) cause a small change in the output (Δoutput).
- ✓ So Δoutput is a *function* of the changes in Δwj (weights) and Δb *(bias), assume it to be a linear function, so*

- ✓ Single variable function, if C depends on only one w, so, output = C = C (w)
- ✓ Average rate of change (slope) = $\Delta C / \Delta w$
- ✓ Instantaneous rate of change (slope) = dC/dx
- ✓ Linear assumption => a straight line so
- ✓ Average slope = instantaneous slope =>
- ✓ $\Delta C / \Delta w = dC / dw$ =>
- ✓ $\Delta C = (dC / dw) * \Delta w$

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b,$$

Error and Scheme

neuralnetworksanddeeplearning.com

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Define a ***cost function***, ***C* is the *quadratic* cost function** also referred as the *mean squared error*(*MSE*) .

Find a scheme to minimize the cost *C*(*w*,*b*) as a function of the weights and biases, casting it as an optimization problem using the ***gradient descent algorithm***.

Find a way of iterating Δ*w*j and Δ*b* so as to make Δ*C* (Δ*output*) negative, Pushing MSE (C) smaller and smaller, ideally to zero

Be negative
Drive the cost to zero

=

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b,$$

For output = C ; wj = v1 ; b = v2

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

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

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

Be negative

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

Just Choose

$$\Delta v = -\eta \nabla C,$$

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

$$v \to v' = v - \eta \nabla C.$$

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

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

Mini-Batch Stochastic Gradient Descent

- ✓ Neural network is an optimization process to find a set of parameters (W, weights) that ensure the prediction function f(x, W) to be as close as possible to the true solution (labeled input) y for any input x. We use gradient descent to find the weights *w* and biases *b* which minimize the cost function, C.
- ✓ To compute the gradient ∇C we need to compute the gradients ∇Cx separately for each training input, *x*, and then average them, $\nabla C=1/n\sum\nabla Cx$. Unfortunately, when the number of training inputs is very large this can take a long time.
- ✓ A way is to use mini-batched ***stochastic gradient descent*** to speed up learning. The idea is to estimate the gradient ∇C by computing a small sample of randomly chosen training inputs, refer to as a ***mini-batch*** *of input* (mini-batched SGD).
- ✓ By averaging over a small sample, we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning, provided the sample size *m* is large enough that the average value of the ∇CXj roughly equals to the average over all ∇Cx.
- ✓ Then, another randomly chosen mini-batch are selected and trained, until all the training inputs are used. It is said to complete an *epoch* (iteration) of training.

$$\frac{\sum_{j=1}^{m}\nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

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

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

DNN Forward Backward Calculation : Weights and bias

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

In one epoch

1. Pick a mini-batch
2. Feed it to Neural Network
3. Forward path calculation
4. Calculate the mean gradient of the mini-batch
5. Use the mean gradient calculated in step 4 to update the weights
6. Repeat steps 1–5 for the mini-batches we created

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

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

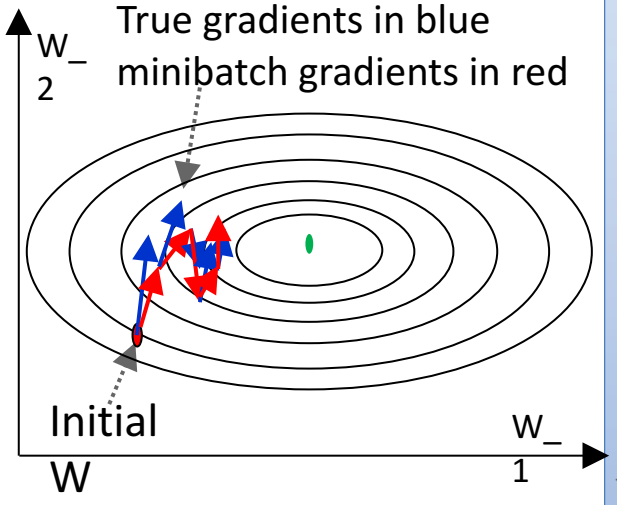
$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Optimization: Batch SGD

Instead of computing a gradient across the entire dataset with size N, divides the dataset into a fixed-size subset ("minibatch") with size m, and computes the gradient for each update on this smaller batch:

(N is the dataset size, m is the minibatch size)

$$\mathbf{g} = \frac{1}{m'}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} E_i(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}),$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{g},$$



Given: training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
Initialize all $\Theta^{(l)}$ randomly (NOT to 0!)
Loop // each iteration is called an epoch
  Loop // each iteration is a mini-batch
    Set $\Delta_{i,j}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)
    Sample m training instances $\mathcal{X} = \{(\mathbf{x}'_1, y'_1), \ldots, (\mathbf{x}'_m, y'_m)\}$ without replacement
    For each instance in $\mathcal{X}$, $(\mathbf{x}_k, y_k)$:
      Set $\mathbf{a}^{(1)} = \mathbf{x}_k$
      Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
      Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_k$
      Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
      Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
    Compute mini-batch regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$
    Update weights via gradient step $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$
  Until all training instances are seen
Until weights converge or max #epochs is reached

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 : Y the end of the NN

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

GD Optimization Algorithms

Gradient Decent (full batch)

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

Stochastic Gradient Decent

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

Mini-batch Stochastic Gradient Decent

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

- ✓ Choosing a proper learning rate can be difficult, slow convergence or diverge!!
- ✓ Learning rate have to be defined in advance and are thus unable to adapt.
- ✓ Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.

https://sites.google.com/view/mlss-2019/lectures-and-tutorials

Momentum

A method which is supposed to help learn faster in the face of noisy gradients (as in our case)

We introduce a new variable which can be thought of as the velocity of learning.

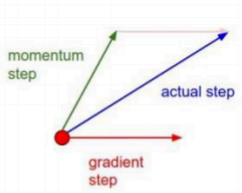
The velocity is basically an exponentially decaying average of the step size:

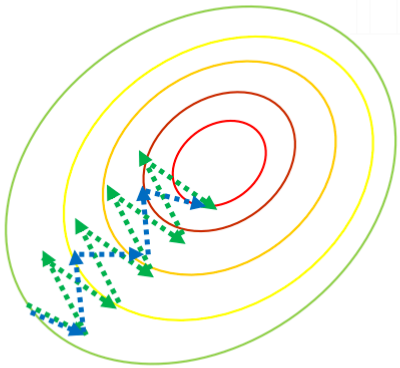
$$v_{j_t} = \gamma v_{j_{t-1}} + \alpha \frac{\partial E_t}{\partial w_j}$$
$$w_j = w_j - v_{j_t}$$

This is similar to a ball rolling down the gradient, and having momentum in a certain direction.

Size of steps depends on size of gradients, but also on how aligned they are.

Common value for $\gamma = 0.9$, which can be thought of as friction.





Global loss function

Mini-Batch Gradient Descent

Mini-Batch Gradient Descent with momentum

Adaptive learning rates + momentum

1. ADAM (Adaptive Moment Estimation):

$$v_{j_t} = \beta_1 v_{j_{t-1}} + (1-\beta_1)\frac{\partial E}{\partial w_j}$$
$$m_{j_t} = \beta_2 m_{j_{t-1}} + (1-\beta_2)\frac{\partial E^2}{\partial w_j}$$
$$w_j = w_j - \frac{\alpha}{\sqrt{m_{j_t}}} v_{j_t}$$

The hyperparameter in general:
- ✓ *beta1* = 0.9
- ✓ *beta_2* = 0.99.

2. NADAM(Nesterov-accelerated Adaptive Moment Estimation):
   1. Use ADAM with Nestrov momentum

https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

Basic Ideas

Typical Neural Network – MLP

Convolutional Neural Network (Unit 3)

STEP 1 : Model Definition

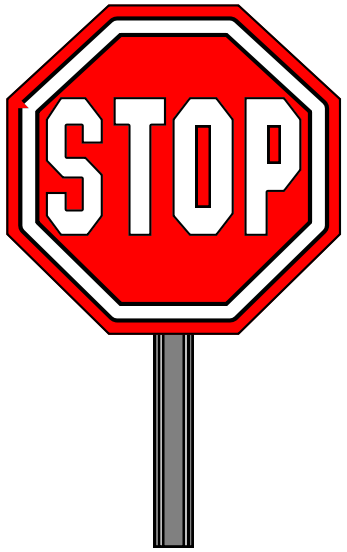
STEP 2 : Cost Function

Step 3 : Optimization Scheme

Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

The End



- The End!