

# LAPENNA Program

## LECTURE 5

**Kwai Wong, Stan Tomov**  
**Julian Halloy, Stephen Qiu, Eric Zhao**

**University of Tennessee, Knoxville**

**March 3, 2022**

# Acknowledgements:

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, [www.jics.utk.edu/lapenna](http://www.jics.utk.edu/lapenna), NSF award #202409
- [www.icl.utk.edu](http://www.icl.utk.edu), [cfdlab.utk.edu](http://cfdlab.utk.edu), [www.xsede.org](http://www.xsede.org),  
[www.jics.utk.edu/recsem-reu](http://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](http://www.bitbucket.org/icl/magmadnn)
- [www.bitbucket.org/cfdl/opendnnwheel](http://www.bitbucket.org/cfdl/opendnnwheel)



INNOVATIVE  
COMPUTING LABORATORY

THE UNIVERSITY OF  
TENNESSEE  
KNOXVILLE

**JICS**  
Joint Institute for  
Computational Sciences  
ORNL  
Computational  
Sciences

OAK  
RIDGE  
National Laboratory

**The major goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem. This program aims to prepare college faculty, researchers, and industrial practitioners to design, enable and direct their own course curricula, collaborative projects, and training programs for in-house data-driven sciences programs. The LAPENNA program focuses on delivering computational techniques, numerical algorithms and libraries, and implementation of AI software on emergent CPU and GPU platforms.**

Ecosystem Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Modeling, Numerical Linear Algebra, Data Analytics, Machine Learning, DNN, GPU, HPC

Session 1	Session 2	Session 3	Session 4
7/2020 - 12/2020	1/2021- 6/2021	7/2021 - 1/2022	1/2022 - 6/2022
16 participants	16 participants	16 participants	16 participants
Faculty/Students	Faculty/Students	Faculty/Students	Faculty/Students
10 webinars	10 webinars	10 webinars	10 webinars
4 Q & A	4 Q & A	4 Q & A	4 Q & A

Colleges courses, continuous integration, online courses, projects, software support

Web-based resources, tutorials, webinars, training, outreach

- ✓ PIs : **Kwai Wong (JICS), Stan Tomov (ICL), University of Tennessee, Knoxville**
  - Stephen Qiu, Julian Halloy, Eric Zhao (Students)
  
- ✓ Team : Clemson University
- ✓ Teams : University of Arkansas
- ✓ Team : University of Houston, Clear Lake
- ✓ Team : Miami University, Ohio
- ✓ Team : Boston University
- ✓ Team : West Virginia University
- ✓ Team : Louisiana State University, Alexandria
- ✓ Teams : Jackson Laboratory
- ✓ Team : Georgia State University
- ✓ Teams : University of Tennessee, Knoxville
- ✓ Teams : Morehouse College, Atlanta
- ✓ Team : North Carolina A & T University
- ✓ Team : Clark Atlanta University, Atlanta
- ✓ Team : Alabama A & M University
- ✓ Team : Slippery Rock University
- ✓ Team : University of Maryland, Baltimore County

- ✓ **Webinar Meeting time. Thursday 8:00 – 10:00 pm ET,**
- ✓ **Tentative schedule, [www.jics.utk.edu/lapenna](http://www.jics.utk.edu/lapenna) --> Spring 2022**

Topic: LAPENNA Spring 2022 Webinar

Time: Feb 3, 2022 08:00 PM Eastern Time (US and Canada)

Every week on Thu, 12 occurrence(s)

Feb 3, 2022 07:30 PM

Feb 10, 2022 07:30 PM

Feb 17, 2022 07:30 PM

Feb 24, 2022 07:30 PM

Mar 3, 2022 07:30 PM

Mar 10, 2022 07:30 PM

Mar 17, 2022 07:30 PM

Mar 24, 2022 07:30 PM

Mar 31, 2022 07:30 PM

Apr 7, 2022 07:30 PM

Apr 14, 2022 07:30 PM

Apr 21, 2022 07:30 PM

Join from PC, Mac, Linux, iOS or Android: <https://tennessee.zoom.us/j/94140469394>

Password: 708069

# Schedule of LAPENNA Spring 2022

## Thursday 8:00pm -10:00pm Eastern Time



The goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem for data-driven applications.

<b>Month</b>	<b>Week</b>	<b>Date</b>	<b>Topics</b>
<b>February</b>	Week 01	3	Logistics, High Performance Computing
	Week 02	10	Computational Ecosystem, Linear Algebra
	Week 03	17	Introduction to DNN, Forward Path (MLP)
	Week 04	24	Backward Path (MLP), Math, Example
<b>March</b>	Week 05	3	Backpro (MLP), CNN Computation
	Week 06	10	CNN Backpropagation, Example
	Week 07	17	CNN Network, Linear Algebra
	Week 08	24	Segmentation, Unet,
<b>April</b>	Week 09	31	Object Detection, RC vehicle
	Week 10	7	RNN, LSTEM, Transformers
	Week 11	14	DNN Computing on GPU
<b>June or July</b>	Week 12	21	Overview, Closing
	Workshop	To be arranged	4 Days In Person at UTK

## ✓ **Unit 5 : Multilayer Perceptron**

- Overview, Activation function, MLP, Forward path, example, computational graph
- Backpropagation, example
- Complete the cycle
- CNN

# DNN Model

Applications

+

Data Ensemble + Input

+

**It's all about  
linear algebra calculations  
(particularly in NN)**

+

Algorithms, Software

+

Output + Data Analysis

+

Hardware

# 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

$$\begin{matrix} \textcolor{orange}{y} \\ \vdots \end{matrix} = \alpha \begin{matrix} \textcolor{orange}{x} \\ \vdots \end{matrix} + \beta \begin{matrix} \textcolor{orange}{y} \\ \vdots \end{matrix}$$

## ✓ Level 2 BLAS — matrix-vector operations

- ✓  $O(n^2)$  data and flops
- ✓ Memory bound:  
 $O(1)$  flops per memory access

$$\begin{matrix} \textcolor{orange}{y} \\ \vdots \end{matrix} = \alpha \begin{matrix} \textcolor{orange}{A} & | & \textcolor{orange}{x} \\ \vdots & & \vdots \end{matrix} + \beta \begin{matrix} \textcolor{orange}{y} \\ \vdots \end{matrix}$$

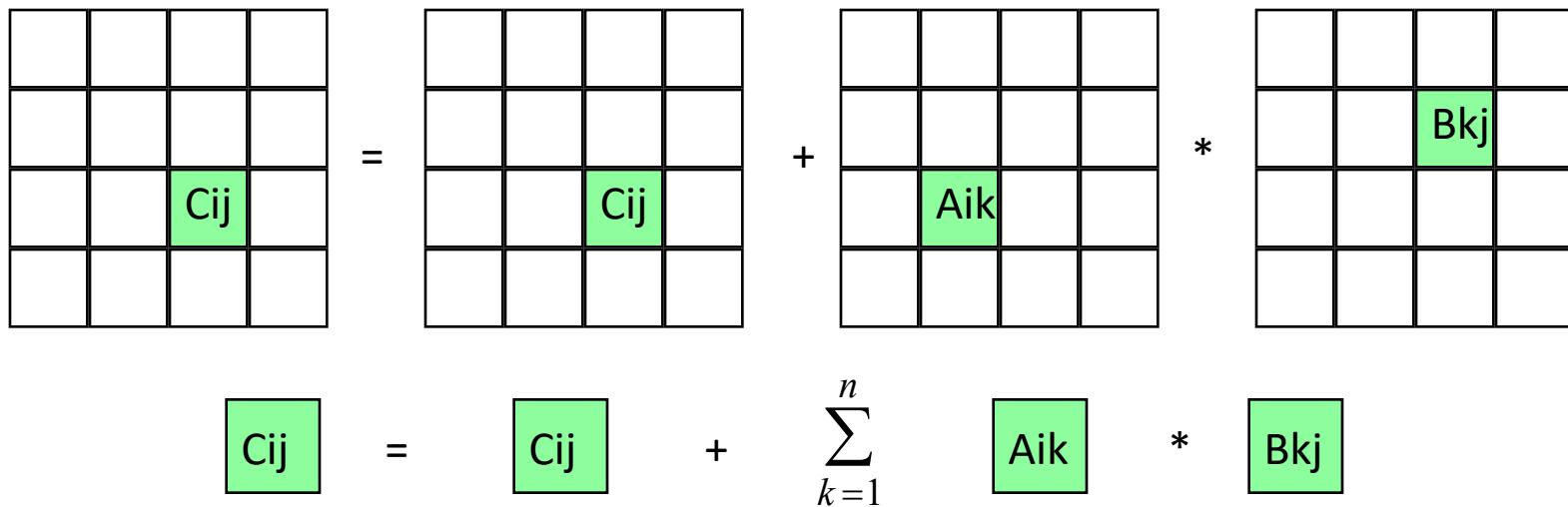
## ✓ Level 3 BLAS — matrix-matrix operations

- ✓  $O(n^2)$  data,  $O(n^3)$  flops
- ✓ Surface-to-volume effect
- ✓ Compute bound:  
 $O(n)$  flops per memory access

$$\begin{matrix} \textcolor{orange}{C} \\ \vdots \end{matrix} = \alpha \begin{matrix} \textcolor{orange}{A} & | & \textcolor{orange}{B} \\ \vdots & & \vdots \end{matrix} + \beta \begin{matrix} \textcolor{orange}{C} \\ \vdots \end{matrix}$$

# It's about Block matrix access

- $q = f/m = (2*n^3) / ((2*N + 2) * n^2) \sim n / N$
- 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  $N = \sqrt{\text{size of fast memory} / 3}$



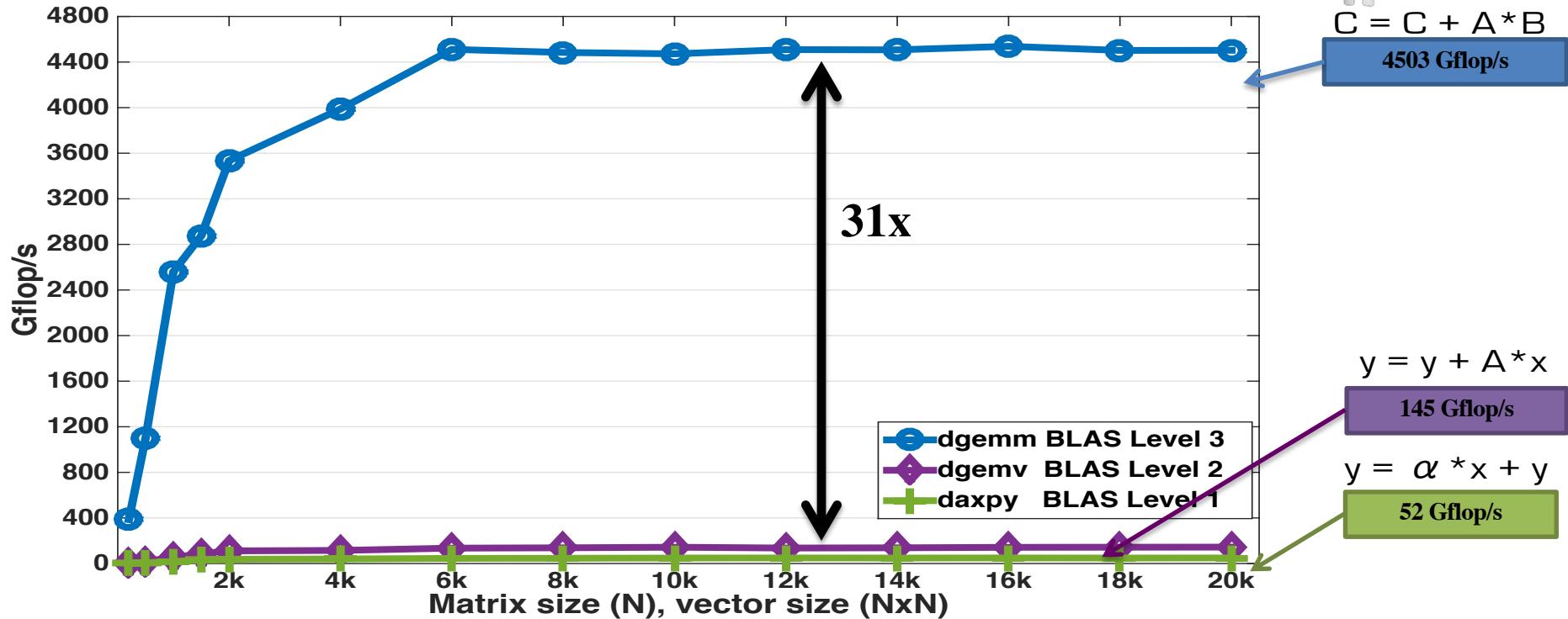
# Level 1, 2 and 3 BLAS

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



$$C = C + A * B$$

4503 Gflop/s



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

## LAB 2

### Problem 4

Write a python code to compute  $C = A \times B$  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)
```

```
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
```

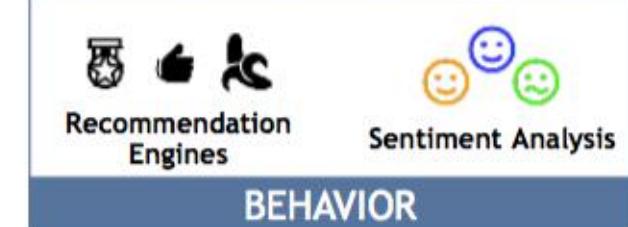
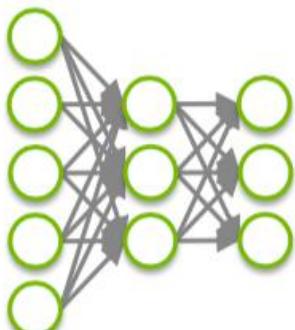
```
1 loop, best of 5: 214 ms per loop
time = 0.22870469093322754
GFLOPS = 69.95921218192831
time = 1.8288803100585938
GFLOPS = 69.98817762759944
time = 6.098201036453247
GFLOPS = 70.84056386754575
```

```
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)
```

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

# Machine Learning – GPU acceleration



cuDNN

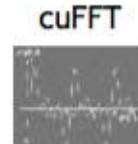
DEEP LEARNING



cuBLAS



cuSPARSE



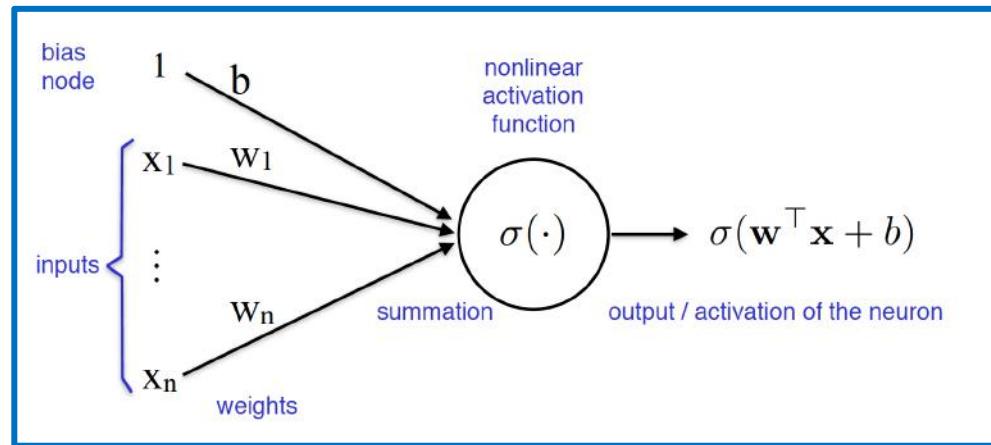
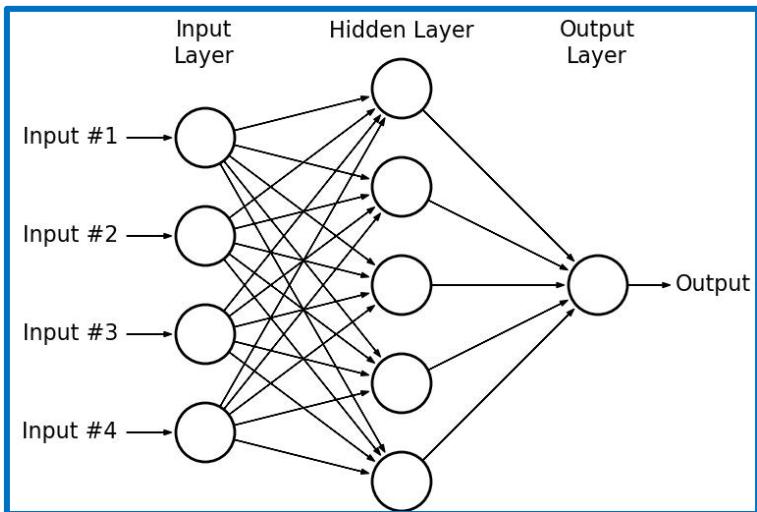
cuFFT

MATH LIBRARIES



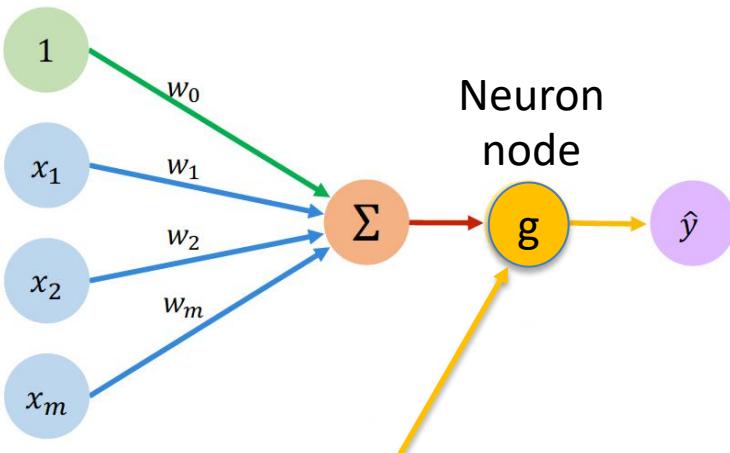
MULTI-GPU

# NN Modeling



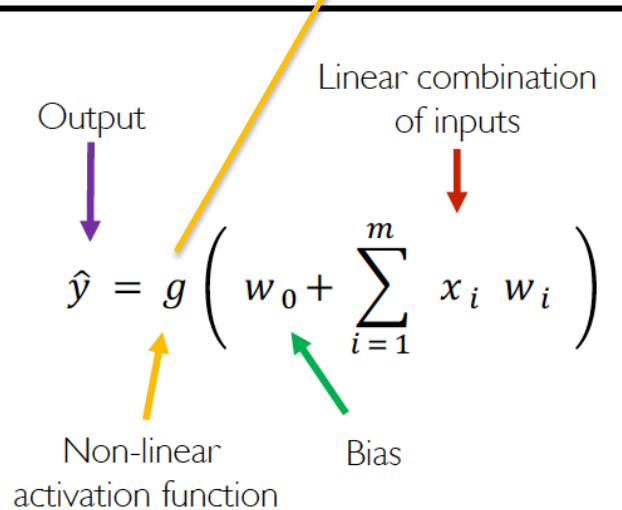
- ✓ A node in the neural network is a mathematical function or activation function which maps input to output values.
- ✓ Inputs represent a set of vectors containing weights ( $w$ ) and bias ( $b$ ). They are the sets of parameters to be determined.
- ✓ Many nodes form a **neural layer**, **links**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

# DNN MLP Forward Steps



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

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

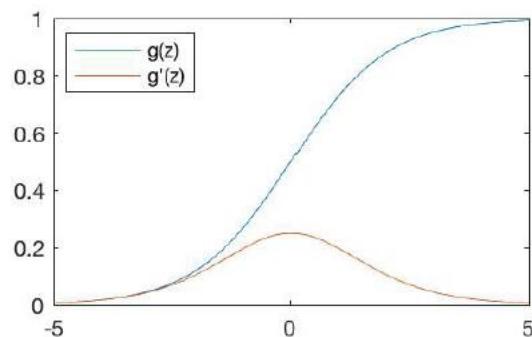


$$\hat{y} = g(w_0 + X^T W)$$

- Example: sigmoid function

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

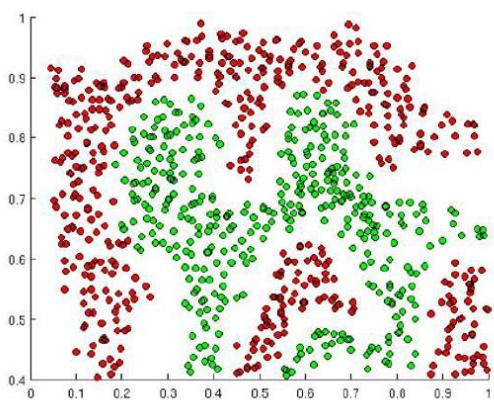
## Sigmoid Function



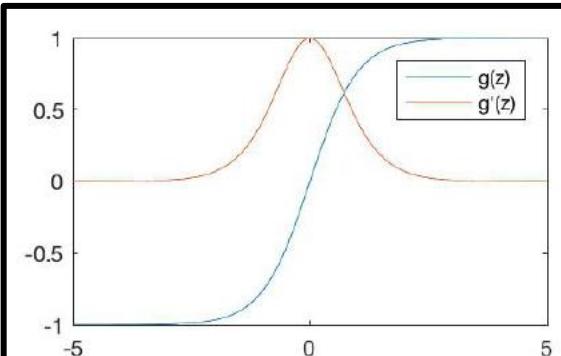
$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.nn.sigmoid(z)`



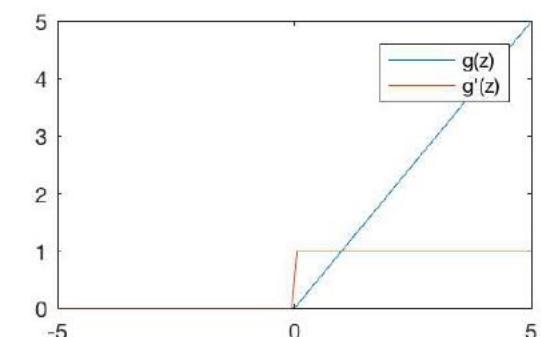
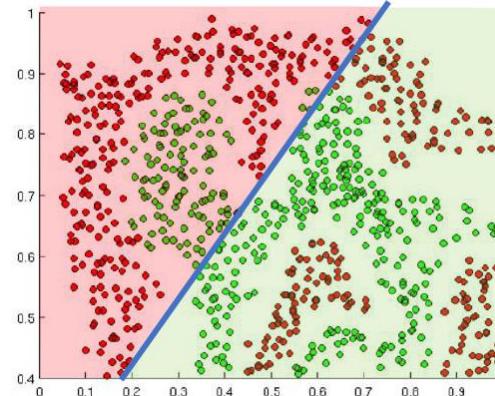
## Nonlinear Activation Function



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

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

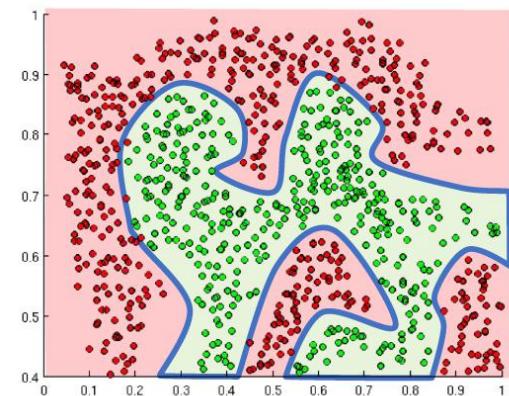
`tf.nn.tanh(z)`



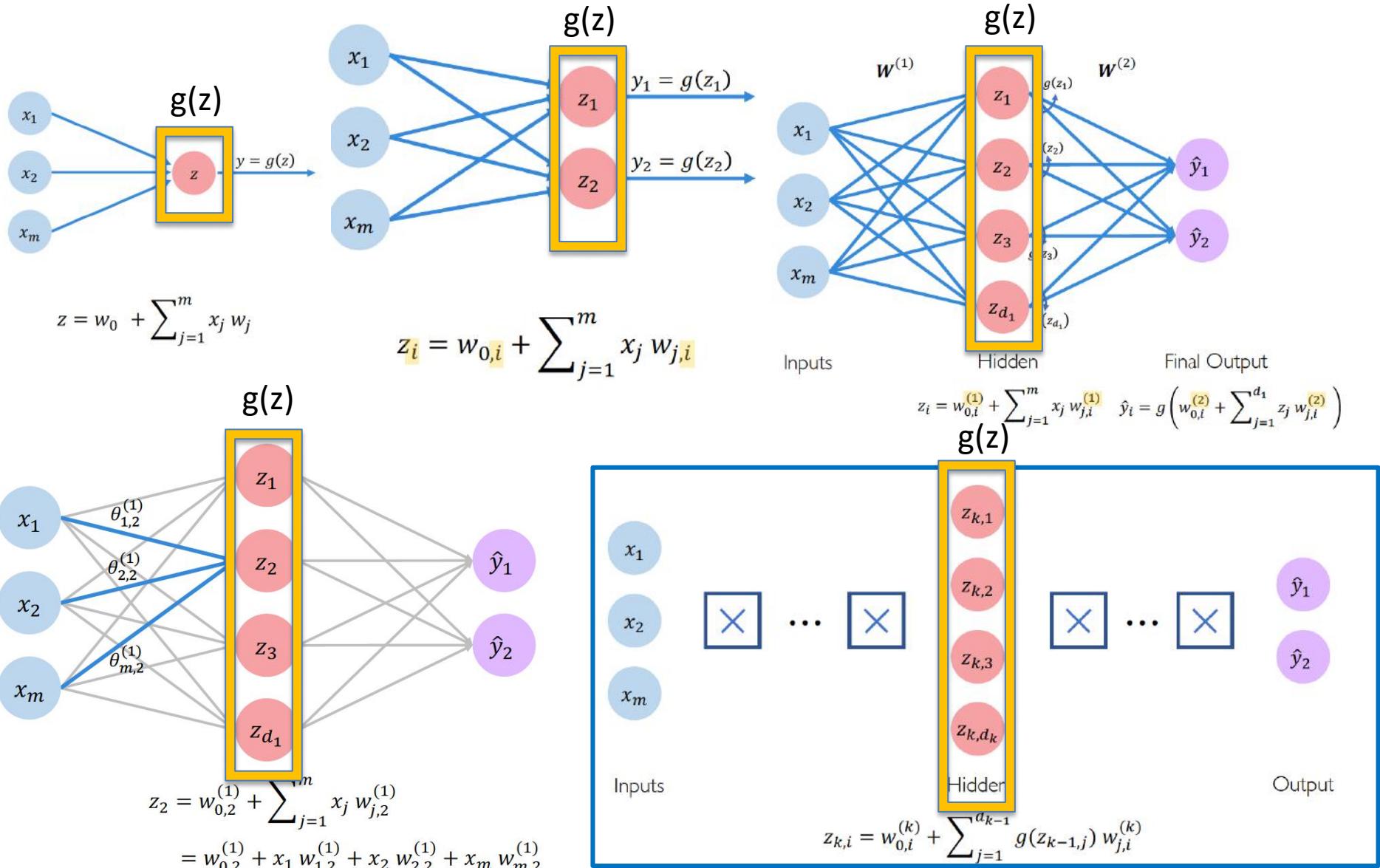
$$g(z) = \max(0, z)$$

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

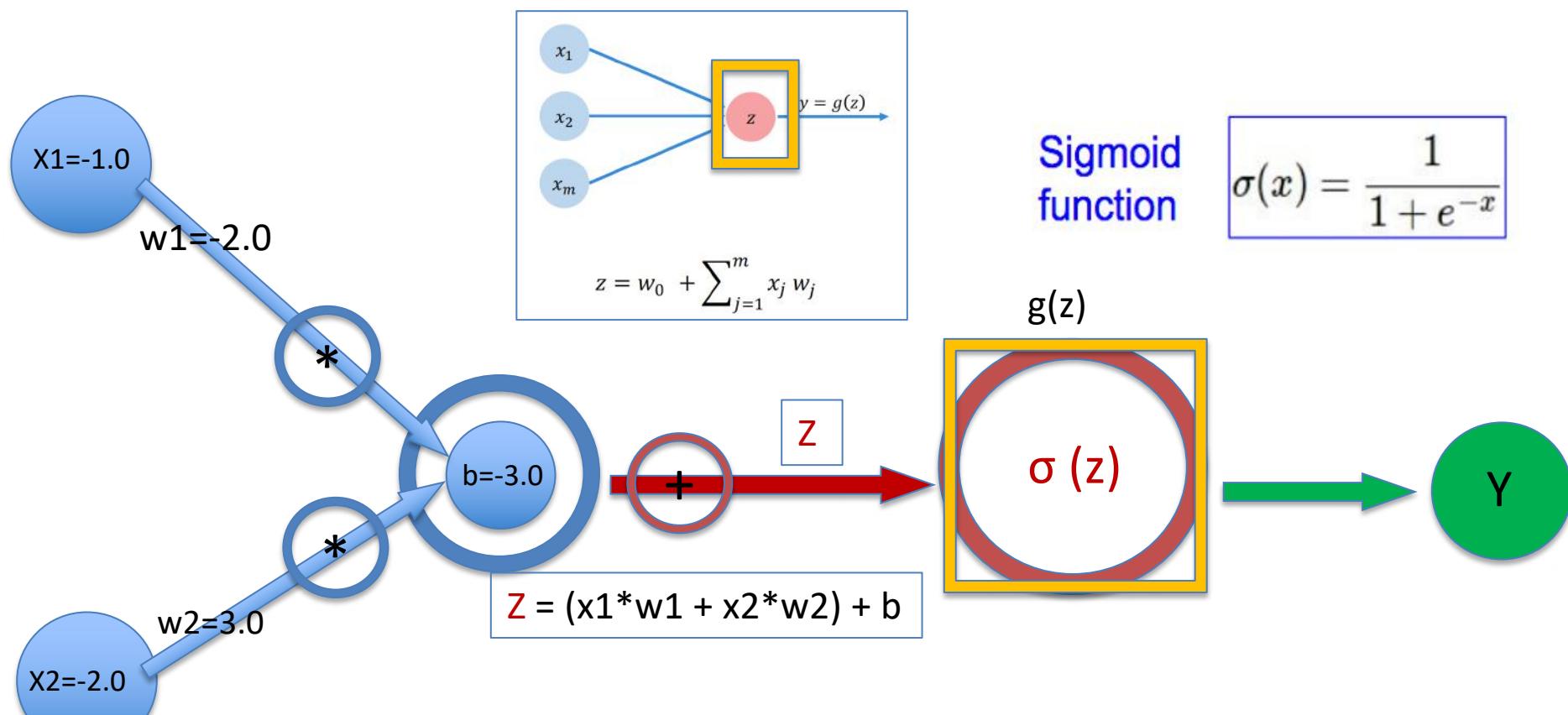
`tf.nn.relu(z)`



# Multilayer Perceptron : Forward Path Calculation



# Forward Calculation represented as a graph



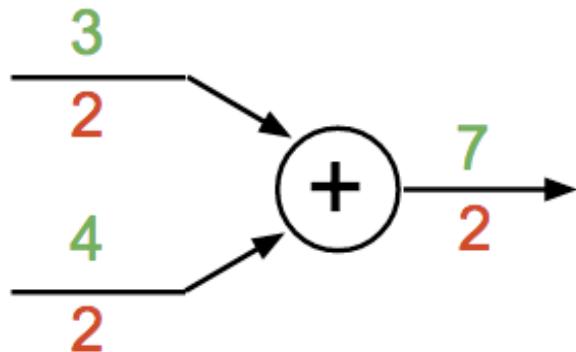
$$Z = (x_1 * w_1 + x_2 * w_2) + b = (-1.0 * -2.0 + -2.0 * 3.0) + -3.0 = 4.0 - 6.0 = -2.0$$

$$Y = \sigma(Z) = \frac{1}{1 + e^{-Z}} = \frac{1}{1 + e^{-(-2.0)}} = \frac{1}{1 + e^{-1}} = 0.731$$

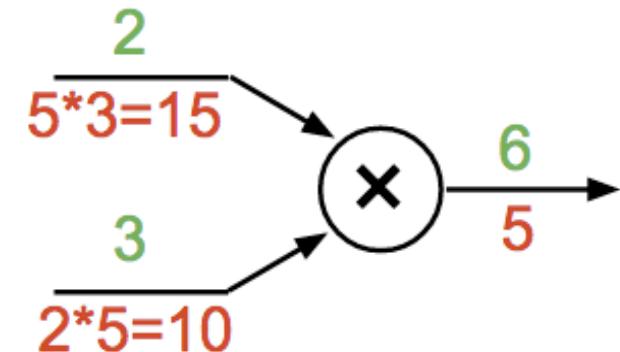
# Forward computational Operators

Forward path (Green) : backward path (Red)

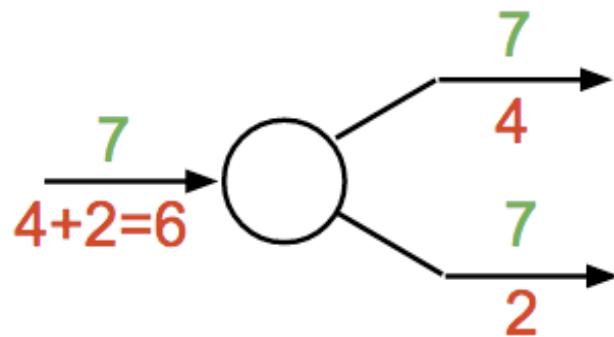
**add** gate: gradient distributor



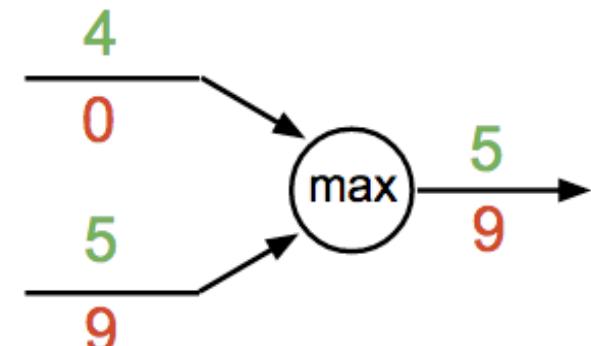
**mul** gate: “swap multiplier”



**copy** gate: gradient adder



**max** gate: gradient router

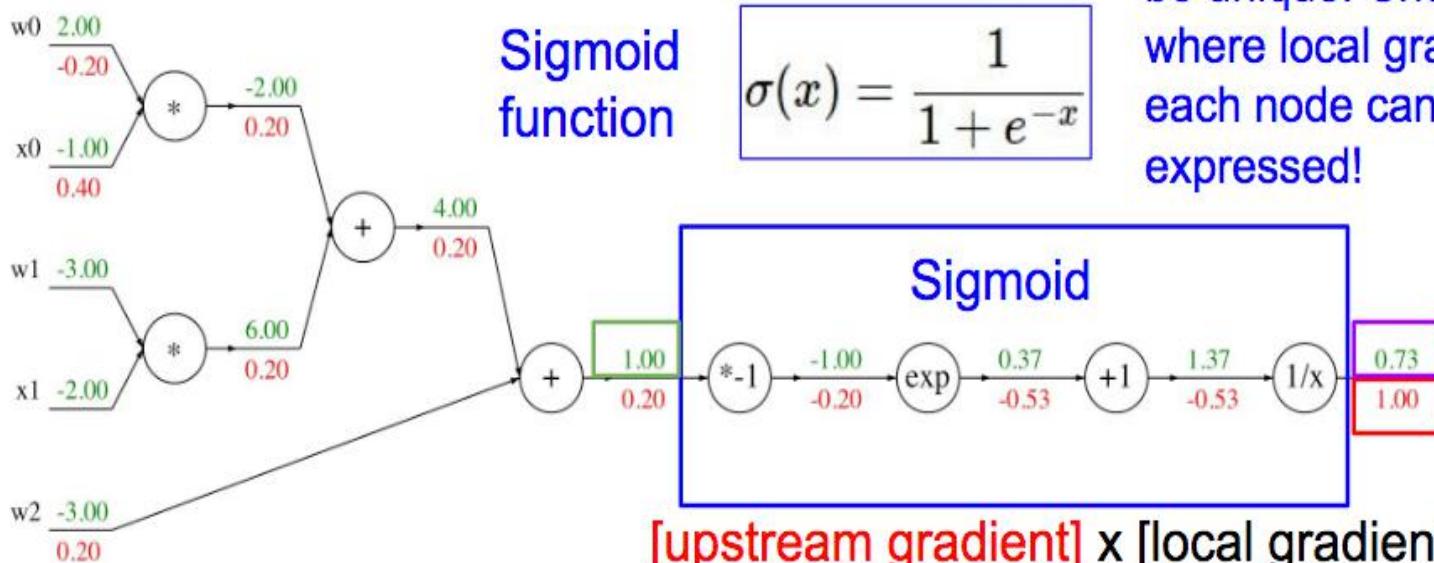


# Computational Graph

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

Another example:

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

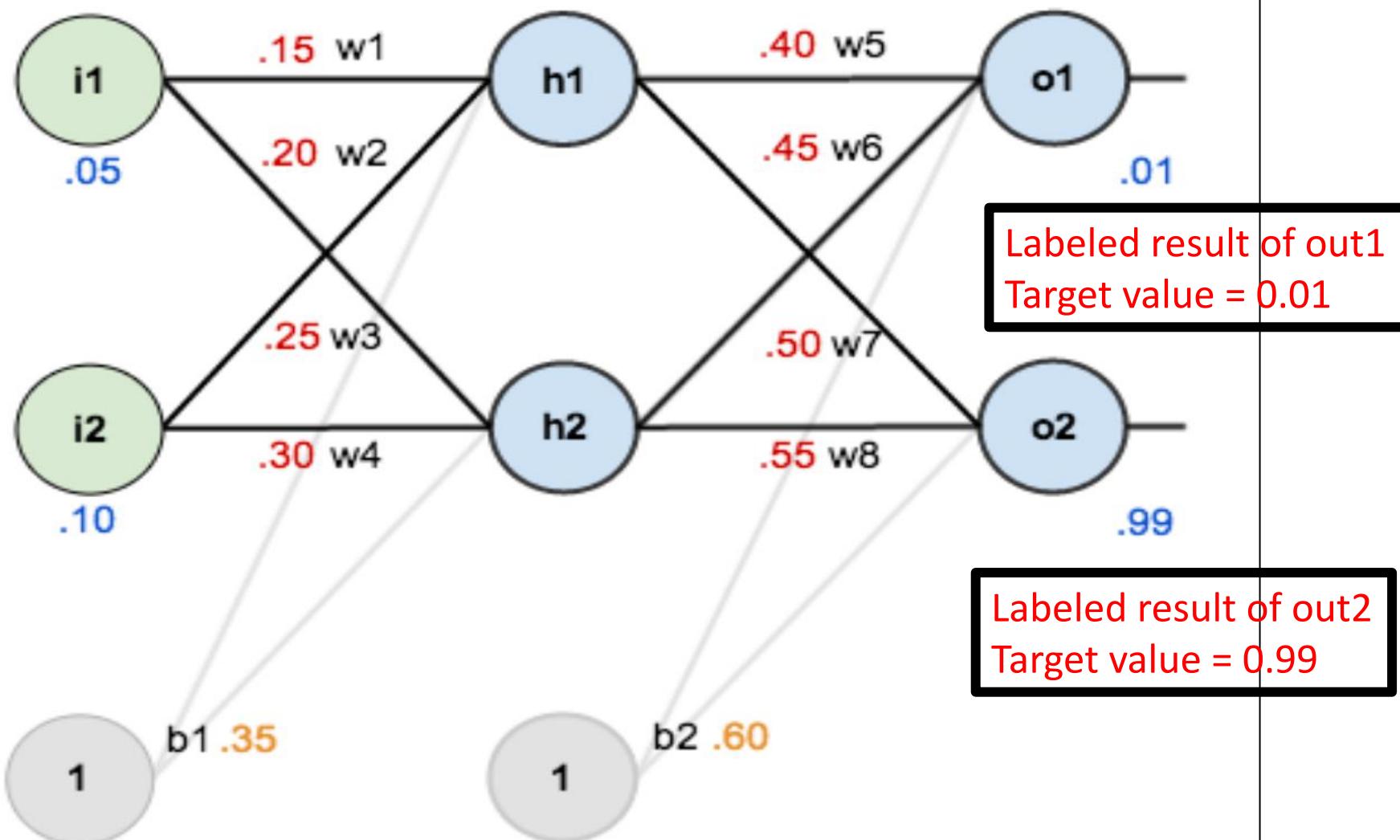


Sigmoid local  
gradient:

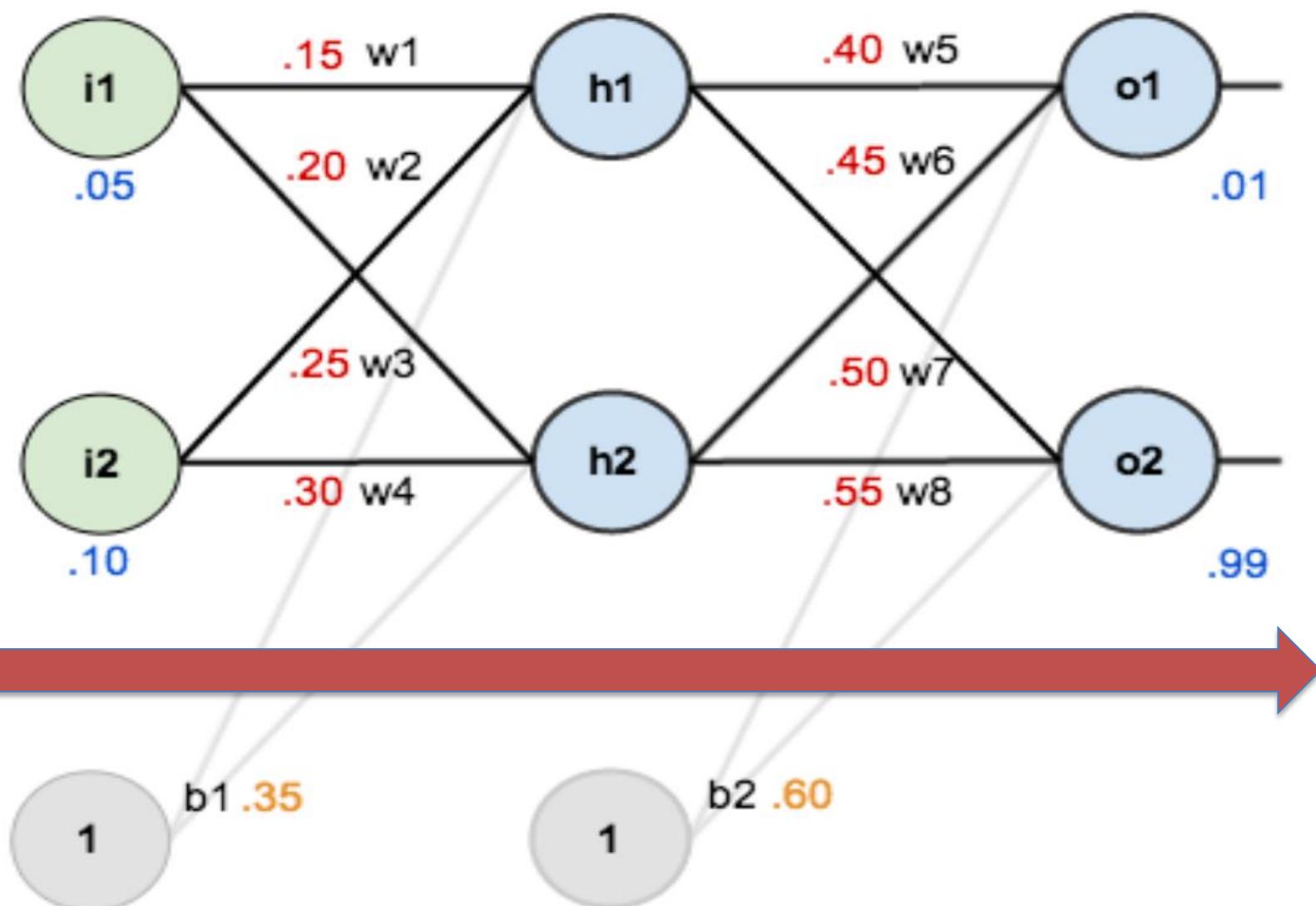
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$

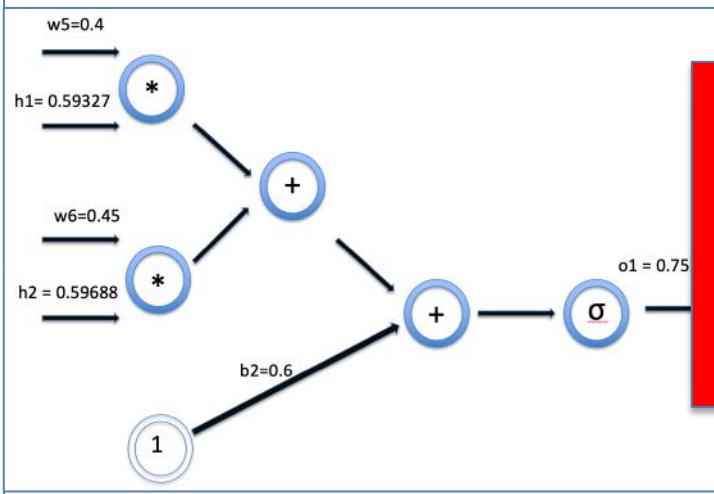
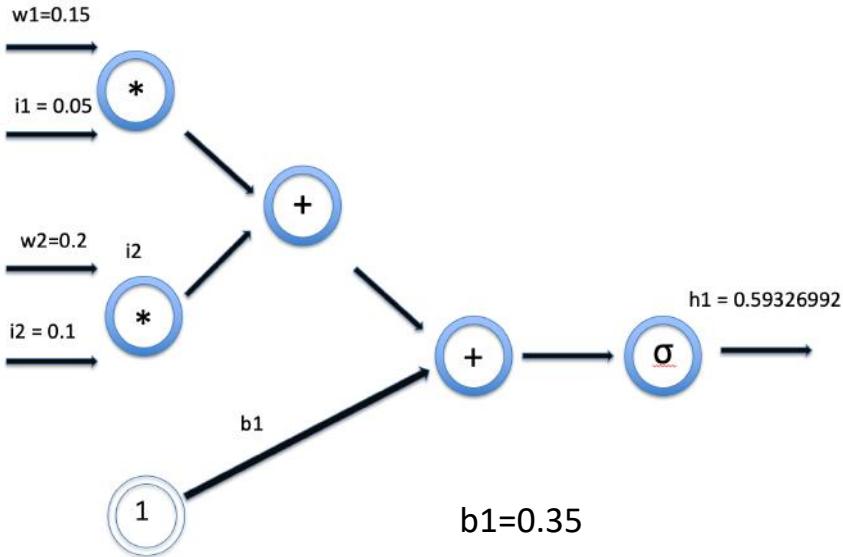
Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

# DNN Example



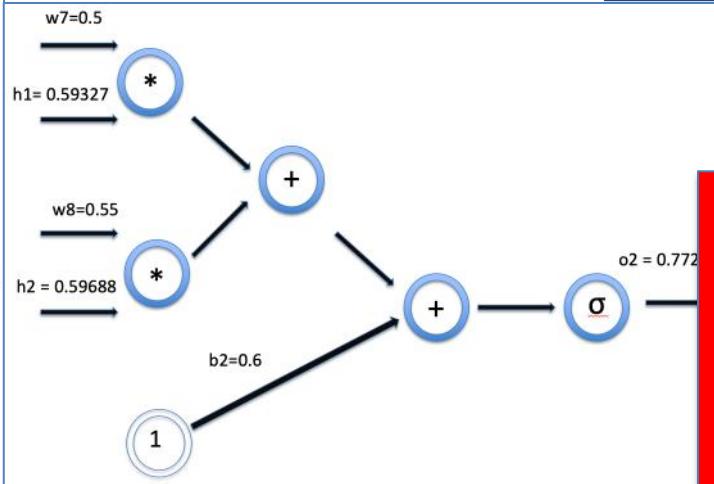
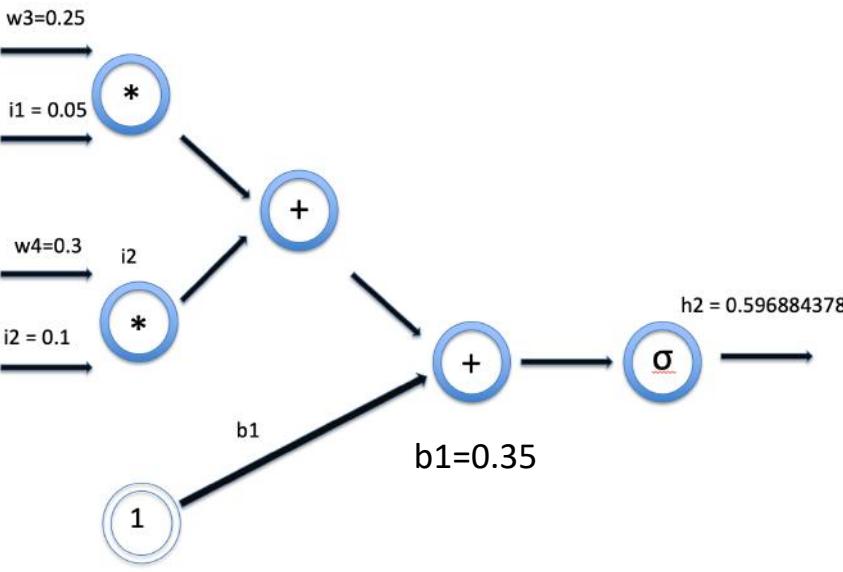
# Multilayer Perceptron : Forward Path Calculation





target=0.01  
o1=0.75136

Evaluate Error Loss Function

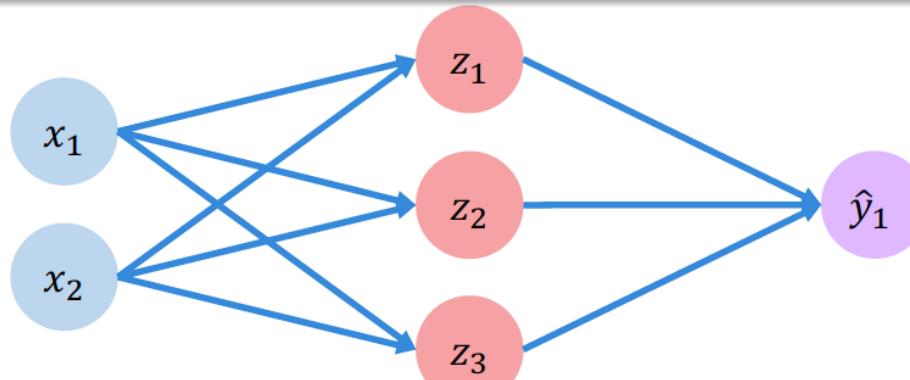


o2=0.77293  
target=0.99

# Mean Squared Error Loss

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

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



$f(x)$	$y$
30	90
80	20
85	95
$\vdots$	$\vdots$

Final Grades  
(percentage)

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

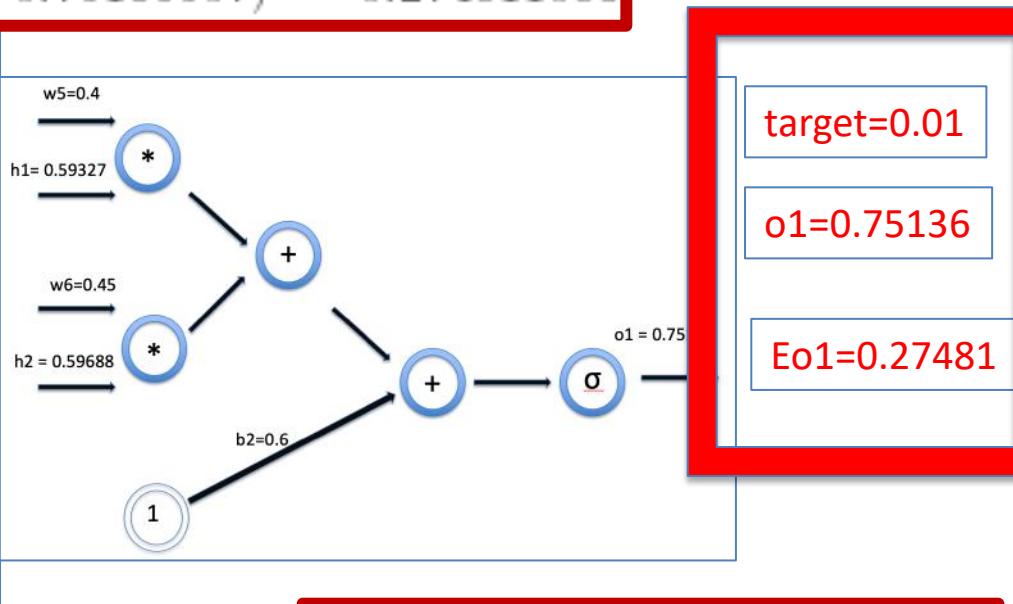
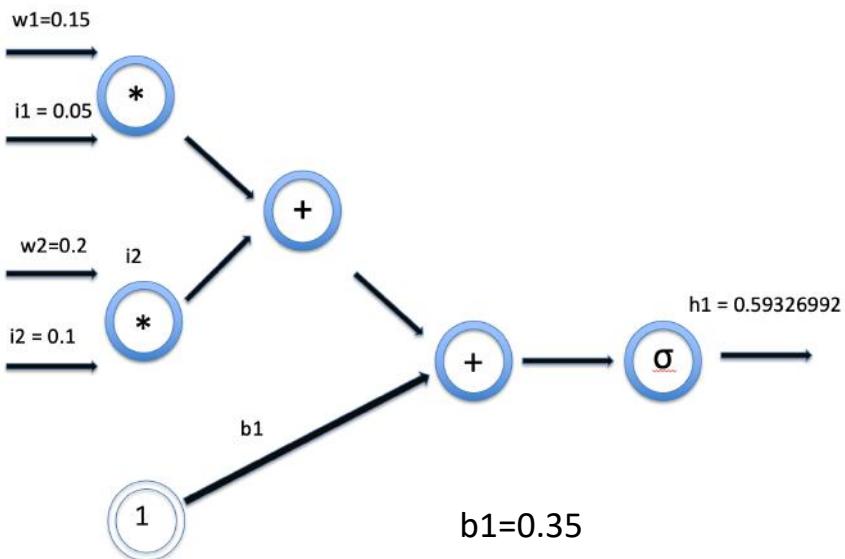
Actual      Predicted



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

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

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$



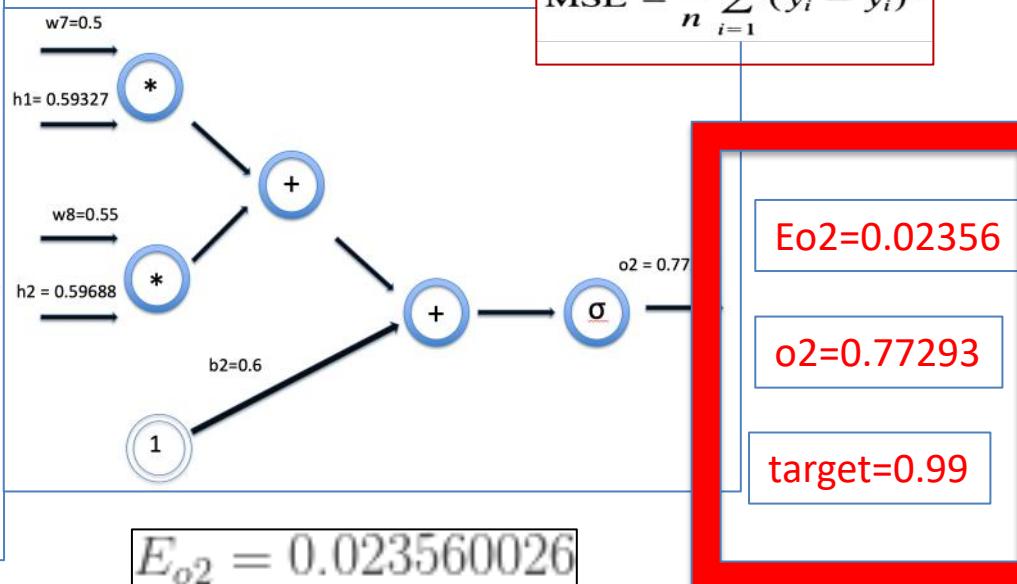
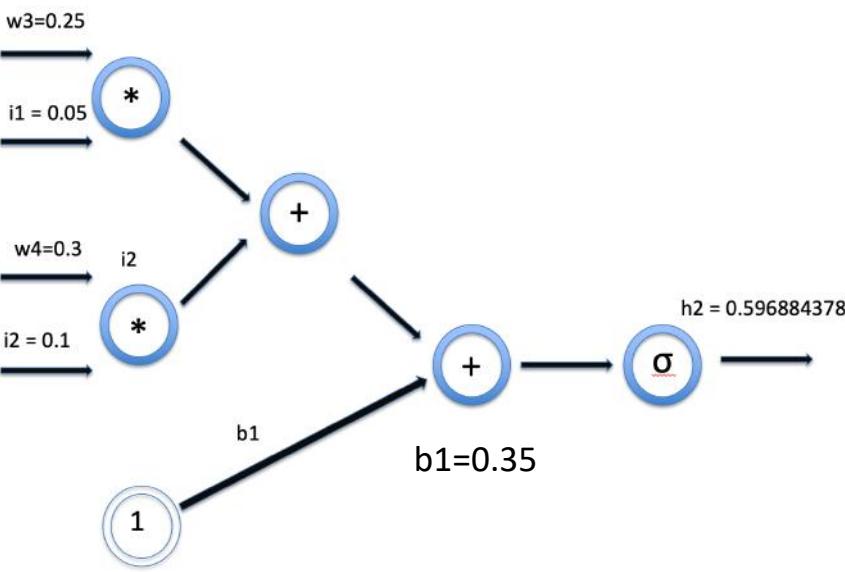
target=0.01

$o1=0.75136$

$E_{o1}=0.27481$

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{total} = E_{o1} + E_{o2} = 0.298371109$$



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

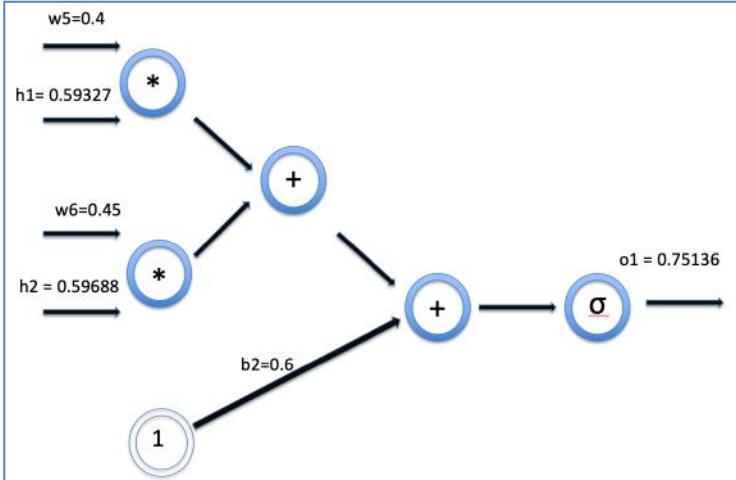
$E_{o2}=0.02356$

$o2=0.77293$

target=0.99

$$E_{o2} = 0.023560026$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$



target=0.01

$o_1=0.75136$

$E_{o1}=0.27481$

Forward Path to Objective Function

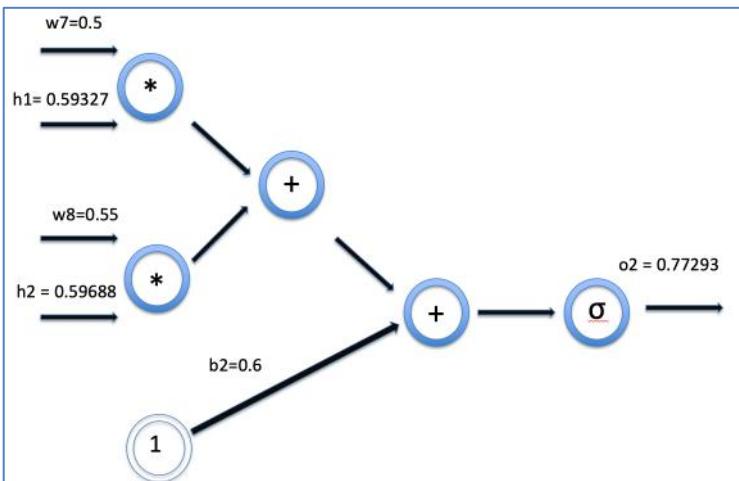
target1 = 0.01

$o_1 = 0.75136$

Cost f

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{total} = E_{o1} + E_{o2} = 0.298371109$$



target2 = 0.99

target=0.99

$o_2=0.77293$

$E_{o2}=0.02356$

$o_2 = 0.77293$

Cost f

$E_{o2} = 0.02356$

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

$E_{o2}=0.02356$

$o_1 = 0.75136$

$E_{o1}=0.27481$

$ET = 0.298371$

grad ET = 1.0

```

# e constant
e = 2.7182818284
# initial values
i1 = 0.05
i2 = 0.10
# initial weights
w1 = 0.15
w2 = 0.20
w3 = 0.25
w4 = 0.30
w5 = 0.40
w6 = 0.45
w7 = 0.50
w8 = 0.55
# bias
b1 = 0.35
b2 = 0.60
# targets
To1 = 0.01
To2 = 0.99

```

```

# forward propagation
h1 = 1/(1+e**(-(w1*i1 + w2*i2+b1)))
print("h1: " + str(h1))

h2 = 1/(1+e**(-(w3*i1 + w4*i2+b1)))
print("h2: " + str(h2))

o1 = 1/(1+e**(-(w5*h1 + w6*h2+b2)))
print("o1: " + str(o1))

o2 = 1/(1+e**(-(w7*h1 + w8*h2+b2)))
print("o2: " + str(o2))

```

```

h1: 0.5932699921052087 h2: 0.5968843782577157
o1: 0.7513650695475076 o2: 0.772928465316421

```

```

# Error
Eo1 = 0.5*(To1-o1)**2
print("Error o1: " + str(Eo1))
Eo2 = 0.5*(To2-o2)**2
print("Error o2: " + str(Eo2))

E = Eo1 + Eo2
print("Total Error: " + str(E))

```

```

Error o1: 0.2748110831725904
Error o2: 0.023560025584942117
Total Error: 0.29837110875753253

```

“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

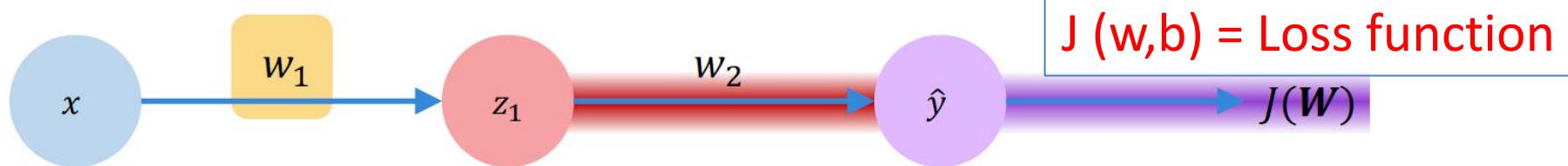
$W$  and  $b$

How does it work?

optimization based on gradient descent algorithm  
go back and forth in the NN model to adjust for  
 $w$  and  $b \Rightarrow$  need derivatives w.r.t.  $w$  and  $b$

Go through forward calculation  
training with backpropagation

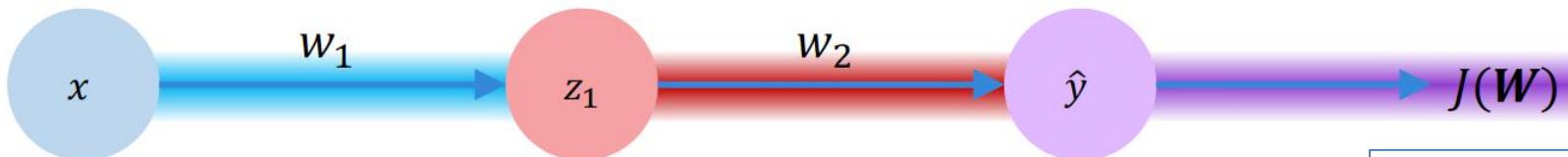
# DNN MLP LA Kernels: Update W and b, Chain rule



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!                      Apply chain rule!

Update W  
and b with  
respect to  
 $J(w, b)$



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Update W  
and b with  
respect to  
 $J(w, b)$



## Optimization : Derivatives and Gradient Descent : Simple Story

— Batch Gradient Descent  
— Mini-batch Gradient Descent  
— Stochastic Gradient Descent

Derivative is the slope of the tangent line

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

$x = 1$

The slope of the tangent line is 2

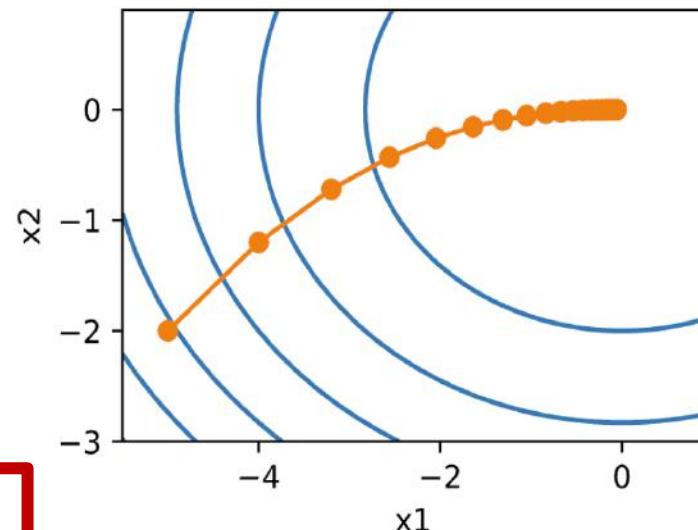
## Algorithm

- Choose initial  $\mathbf{x}_0$  ( $\mathbf{X} = \mathbf{W}$ , weight)
- At time  $t = 1, \dots, T$

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

- $\eta$  is called learning rate

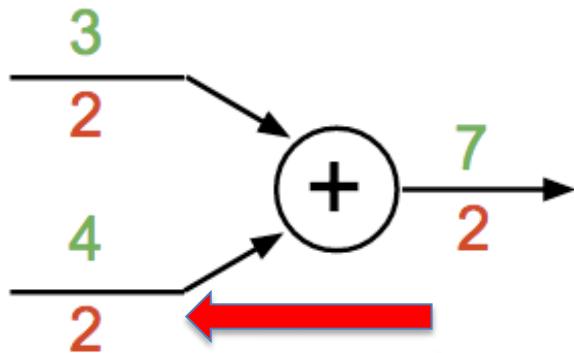
$$\mathbf{W}+ = \mathbf{W} - (\text{learning rate}) * \frac{\partial J}{\partial \mathbf{w}}$$



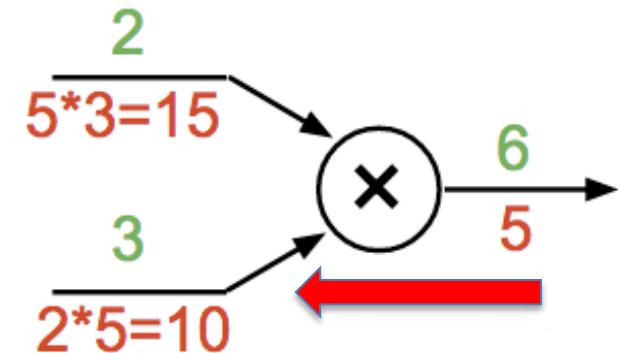
# Forward and Backpropagation Operators

## Forward path : backward path

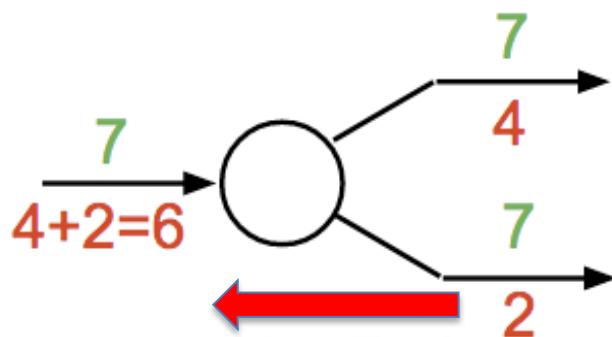
**add** gate: gradient distributor



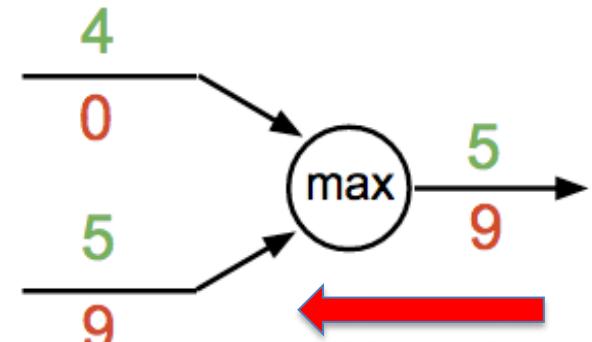
**mul** gate: “swap multiplier”



**copy** gate: gradient adder

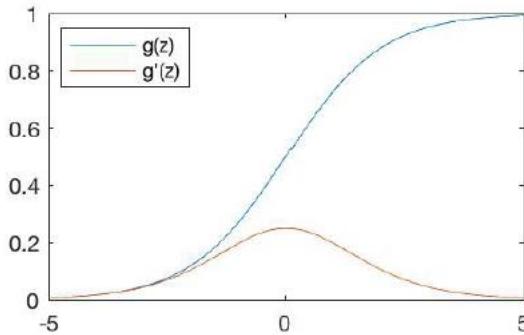


**max** gate: gradient router



# Nonlinear Activation Function

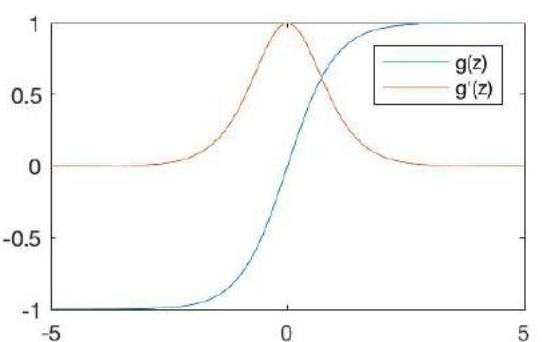
Sigmoid Function



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

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

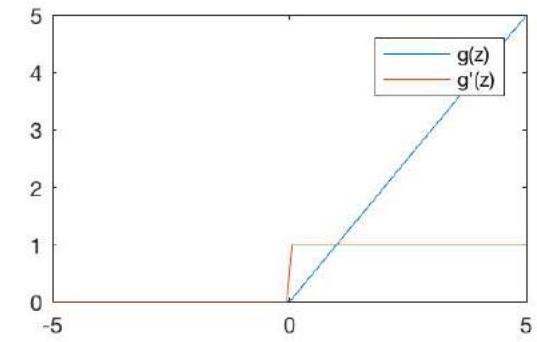
`tf.nn.sigmoid(z)`



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

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

`tf.nn.tanh(z)`

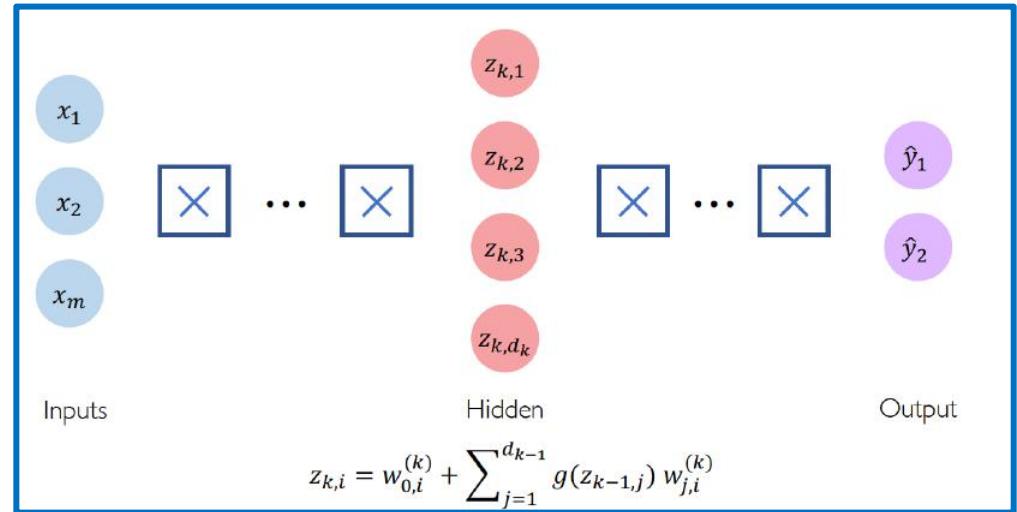


$$g(z) = \max(0, z)$$

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

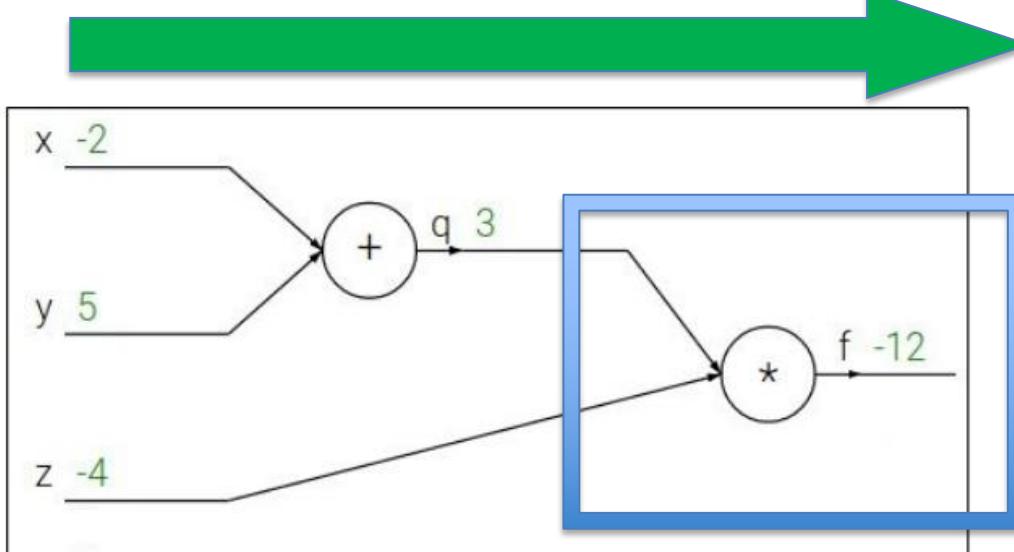
`tf.nn.relu(z)`

MIT, 6.S191 Introduction to  
Deep Learning,  
[introtodeeplearning.com](http://introtodeeplearning.com)



# Computational Graph

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



## Forward path calculation

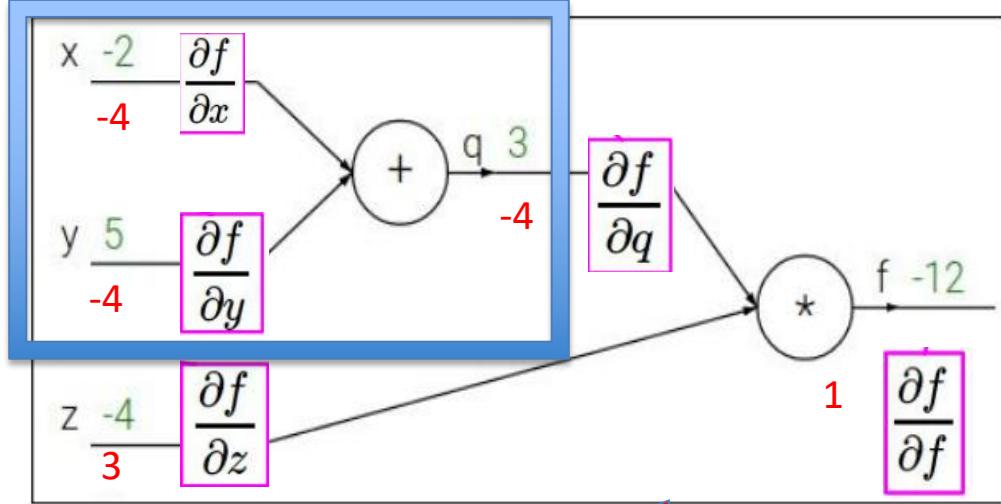
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Need Gradient of  $f$  wrt inputs



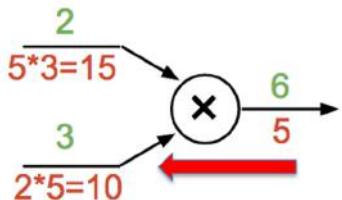
Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = (z)(1) = -4$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = (z)(1) = -4$$

## MUL GATE ( \* )

mul gate: "swap multiplier"



# Computational Graph : backward path calculation

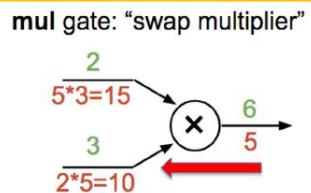
Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$$df/dq = d(q * z)/dq = q * dz/dq + z dq/dq = q * 0 + z * 1 = z$$

$$q = x + y$$

$$f = qz$$

$$\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

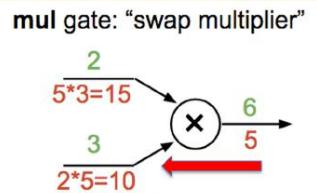


$$\frac{\partial f}{\partial q} = z * \frac{\partial f}{\partial f}$$

$$f = q * z$$

$$z$$

$$\frac{\partial f}{\partial f} = 1$$

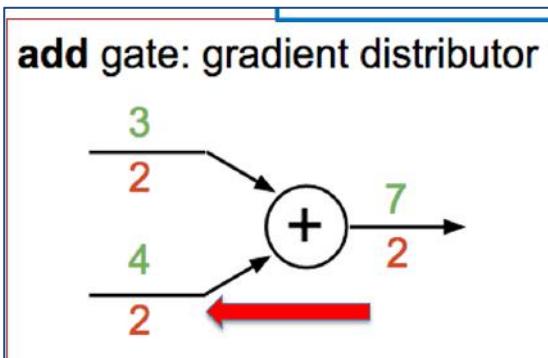


$$\frac{\partial f}{\partial z} = q * \frac{\partial f}{\partial f}$$

$$\begin{aligned} df/dz &= d(q * z)/dz \\ &= q * dz/dz + z dq/dz \\ &= q * 1 + z * 0 = q \end{aligned}$$

# Computational Graph : backward path calculation

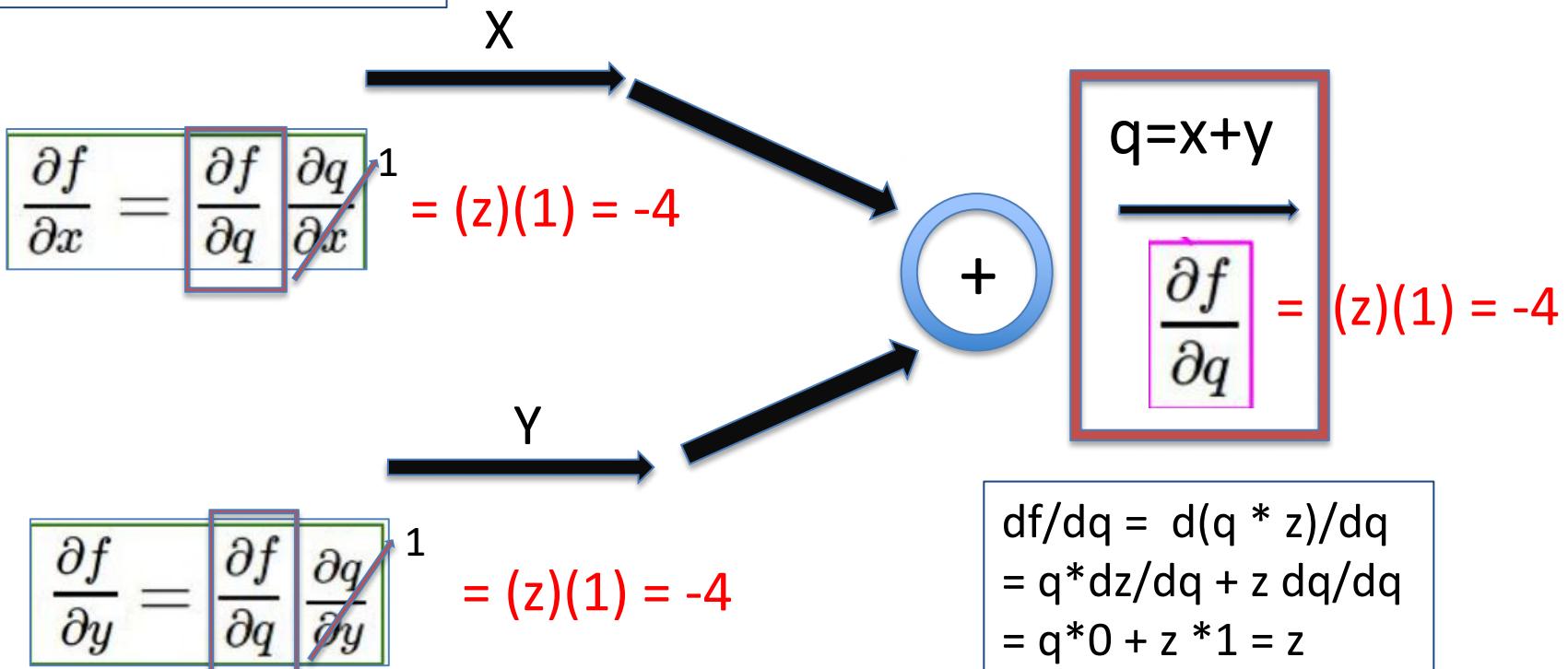
## ADD GATE



Need Gradient of  $f$  wrt to the variable

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$



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

$$q_i = X_i * W_i$$

# Computational Graph Backpropagation

$$dL/dw_1 = (dL/dq_1) * (dq_1/dw_1) = [dL/dq_1] * [d(x_1 * w_1)/dq_1] = (dL/dq_1) * x_1$$

$x_1$

$$(dq_1/dw_1) = [d(x_1 * w_1)/dw_1] = x_1 dw_1/dw_1 + w_1 dx_1/dw_1 = x_1$$



$$q_1 = X_1 * W_1$$

$w_1$

\*

$$dL/dw_1 = (dL/dq_1) * x_1$$

$$dL/dq_1$$

Loss

$$dL/dw_1$$

$$L = f = q_1 + q_2$$



+

$x_2$



$$q_2 = X_2 * W_2$$

$w_2$

\*

$$dL/dw_2 = (dL/dq_2) * x_2$$

$$dL/dq_2$$

$$\frac{\partial f}{\partial f} = 1$$

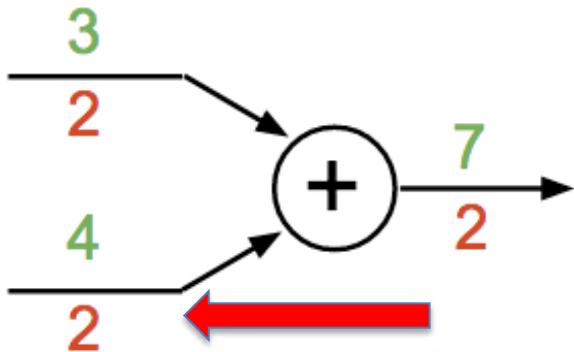
$$dL/dw_2$$

$$(dq_2/dw_2) = [d(x_2 * w_2)/dw_2] = x_2 dw_2/dw_2 + w_2 dx_2/dw_2 = x_2$$

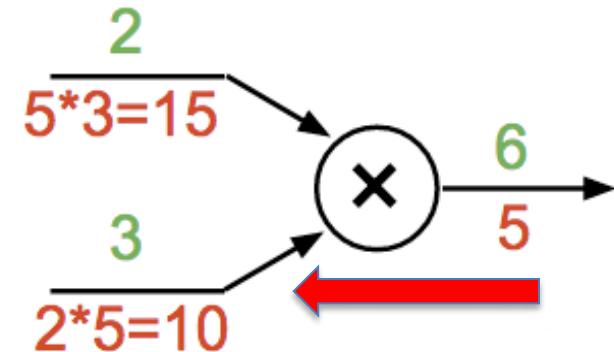
$$dL/dw_2 = (dL/dq_2) * (dq_2/dw_2) = [dL/dq_2] * [d(x_2 * w_2)/dq_2] = (dL/dq_2) * x_2$$

## Forward path : backward path

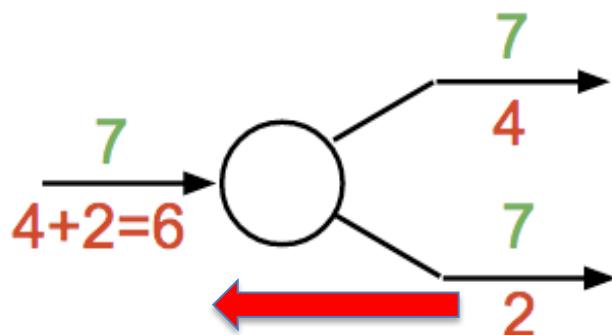
**add** gate: gradient distributor



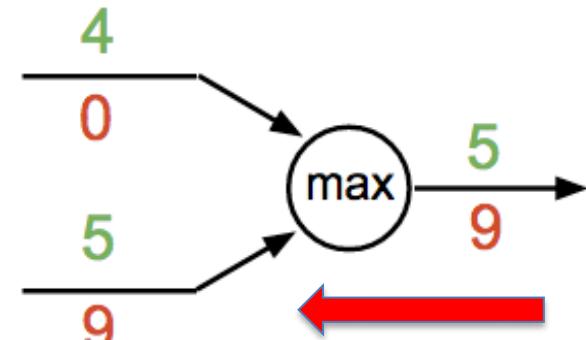
**mul** gate: “swap multiplier”

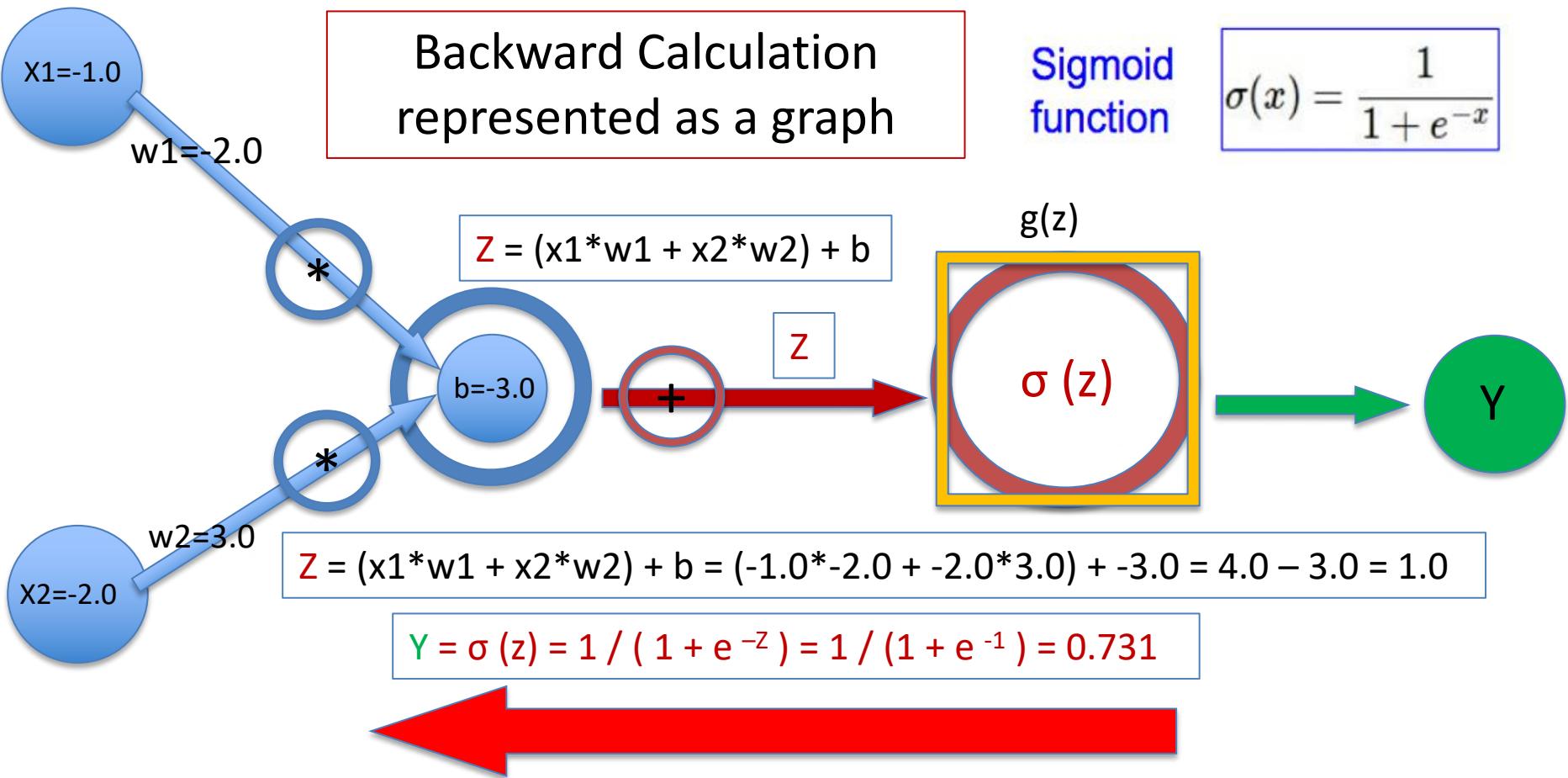


**copy** gate: gradient adder



**max** gate: gradient router





**Sigmoid local gradient:**

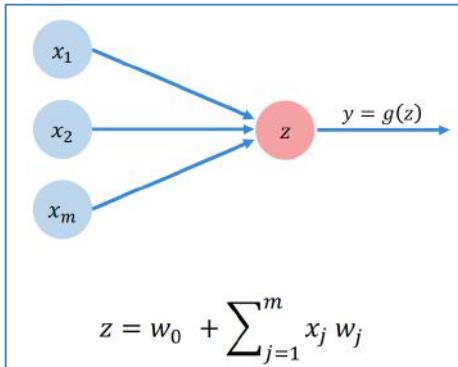
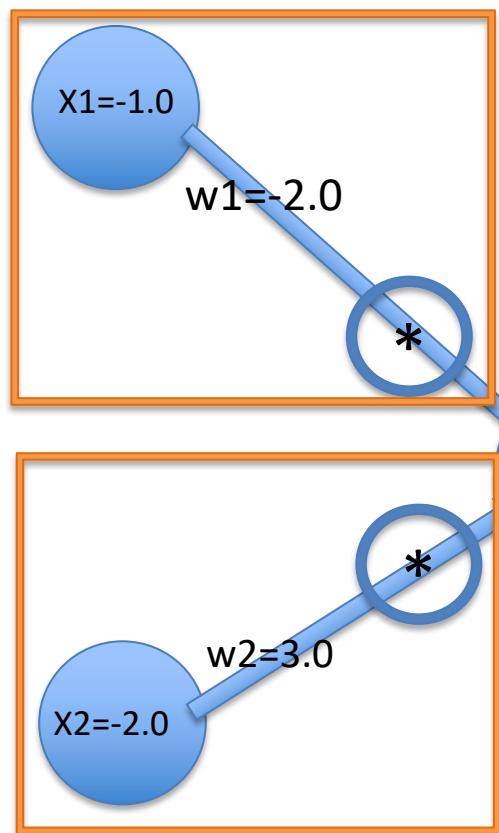
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

**Downstream gradient =**  
 $\text{grad } z = d Y/dz$

[upstream gradient] x [local gradient]  
 $[1.00] \times [(1 - 0.73)(0.73)] = 0.2$

# Forward and backward computation operators

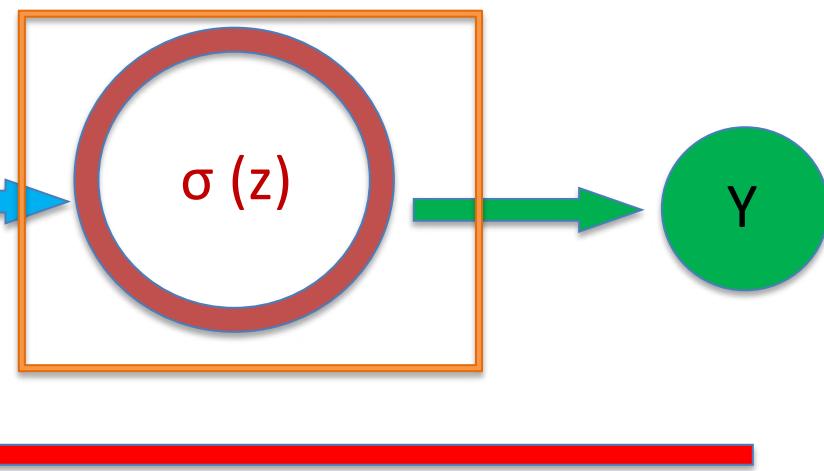
$$Z = (x_1 * w_1 + x_2 * w_2) + b = (-1.0 * -2.0 + -2.0 * 3.0) + -3.0 = 4.0 - 3.0 = 1.0$$



Sigmoid function

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

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



multiple operator :  
downstream gradient =  
upstream gradient \* input value

add operator :  
Downstream gradient =  
upstream gradient

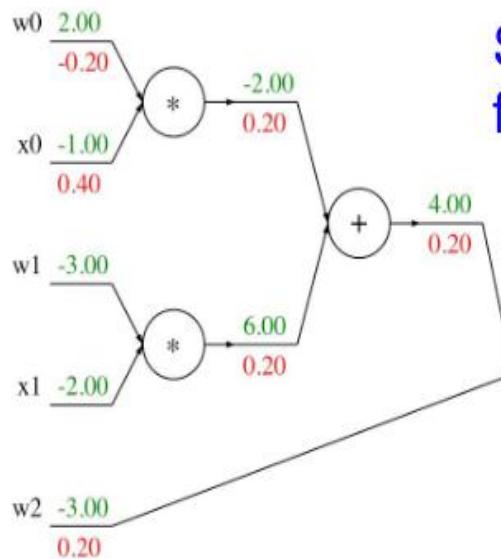
function operator :  
downstream gradient =  
Upstream gradient \* local function gradient

# Computational Graph

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

Another example:

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



Sigmoid function

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

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

Sigmoid

[upstream gradient]  $\times$  [local gradient]  
[1.00]  $\times$  [(1 - 0.73) (0.73)] = 0.2

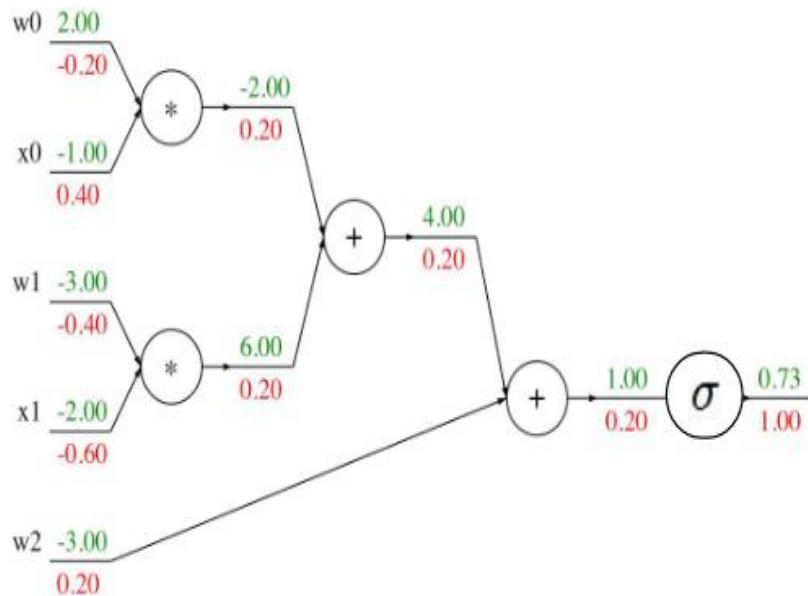
Sigmoid local gradient:

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

# 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)
```

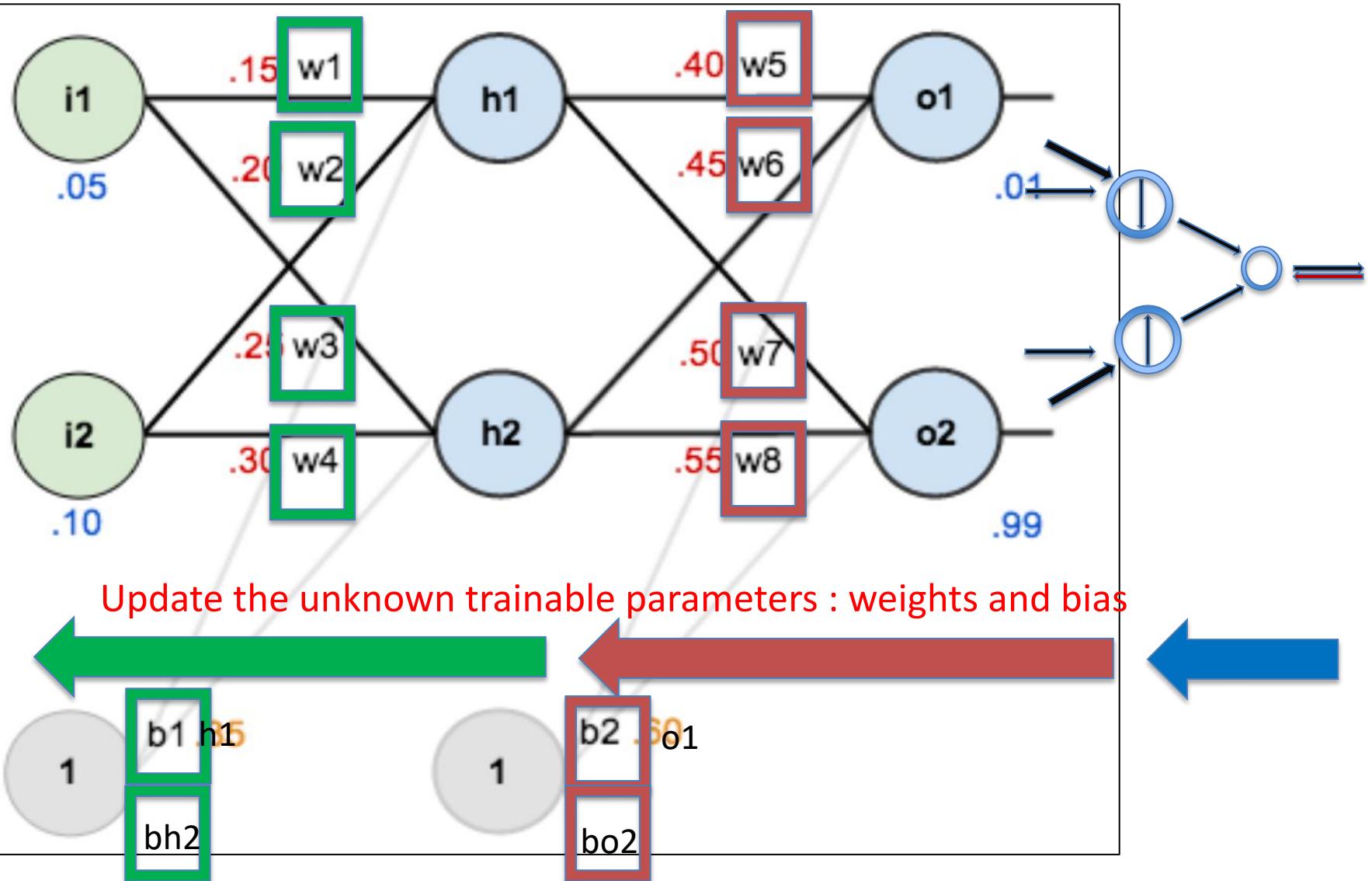
Backward pass:  
Compute grads

```
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
```

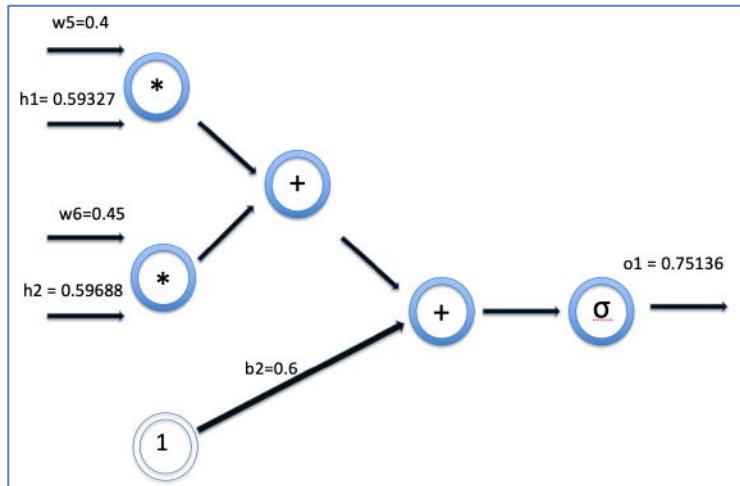
<http://cs231n.stanford.edu/syllabus.html>



# Multilayer Perceptron : Backward Path Calculation

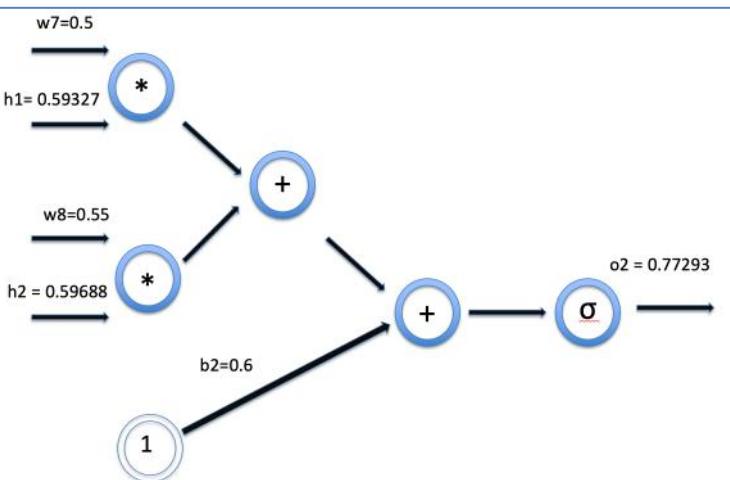
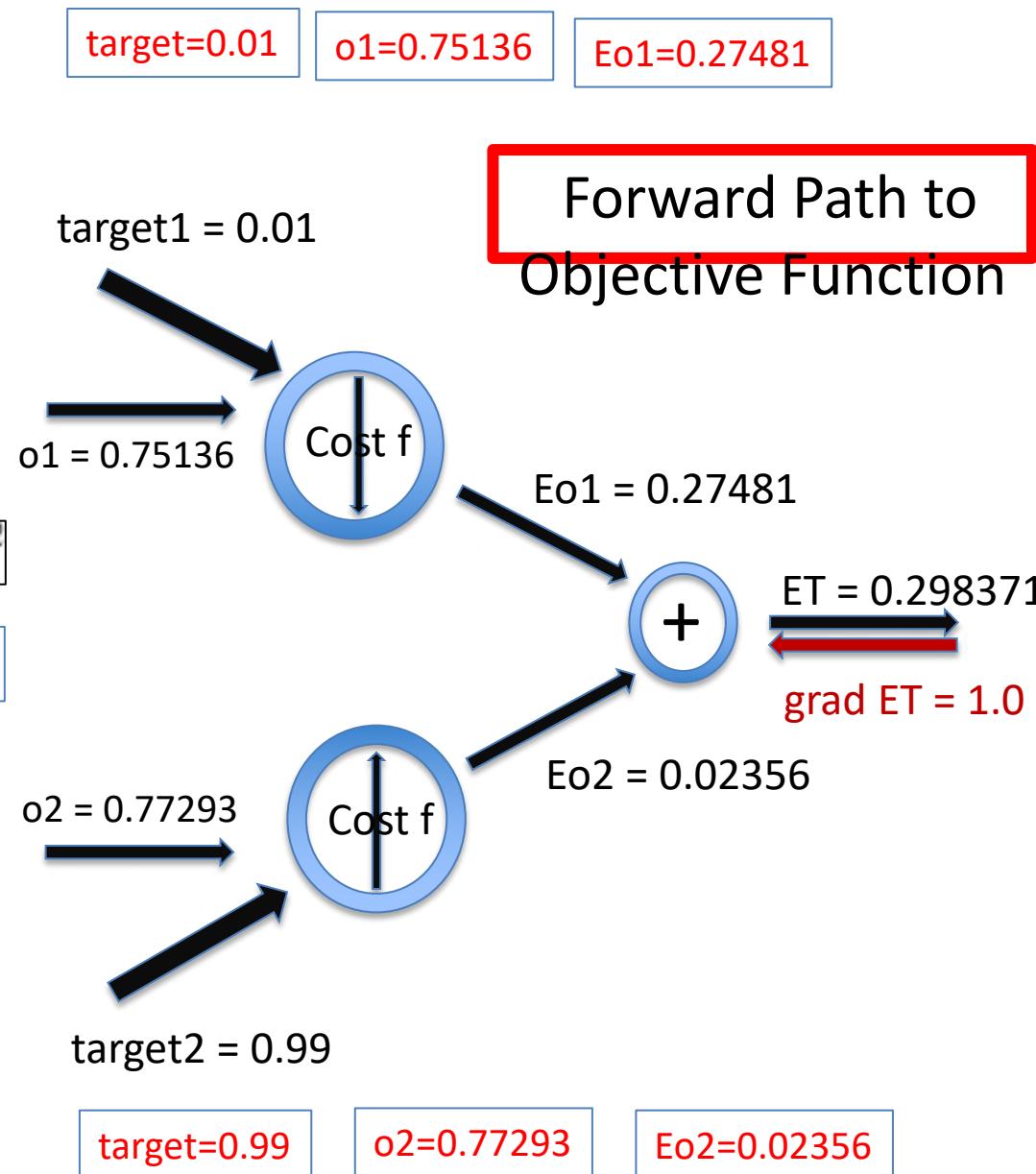


$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$



$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{total} = E_{o1} + E_{o2} = 0.298371109$$



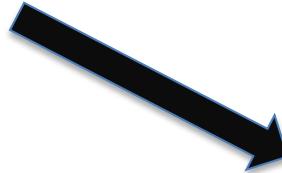
$$target=0.99$$

$$o2=0.77293$$

$$Eo2=0.02356$$

## Backward Path from Objective Function

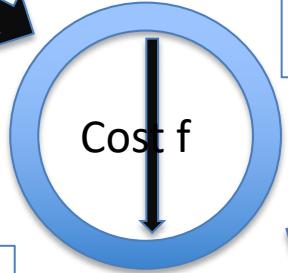
target1 = 0.01



$o_1 = 0.75136$



$\text{grad } o_1 = 0.74136$



composite diff :  $\text{grad } o_1 = (\text{grad up}) * (\text{local grad})$

local grad =  $(dE_1/do_1) = (o_1 - \text{target}_1)$

$\text{grad } o_1 = (1) (0.75136 - 0.01) = 0.74136$

$E_{o1} = \frac{1}{2} * (\text{target}_1 - o_1)^2$

$E_{o1} = 0.27481$

$\text{grad } E_{o1} = 1.0$



$ET = 0.298371$

$ET = E_1 + E_2$

$\text{grad } ET = 1.0$

$E_{o2} = \frac{1}{2} * (\text{target}_2 - o_2)^2$

$E_{o2} = 0.02356$

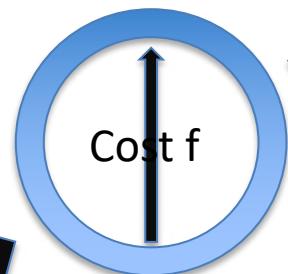
$\text{grad } E_{o2} = 1.0$

$\text{grad } o_2 = (1) (0.77293 - 0.99) = -0.21707$

local grad =  $(dE_2/do_2) = (o_2 - \text{target}_2)$

$\text{target}_2 = 0.99$

composite diff :  $\text{grad } o_2 = (\text{grad up}) * (\text{local grad})$



# Backward Path from o1

$$w5=0.4$$



$$h1 = 0.59327$$



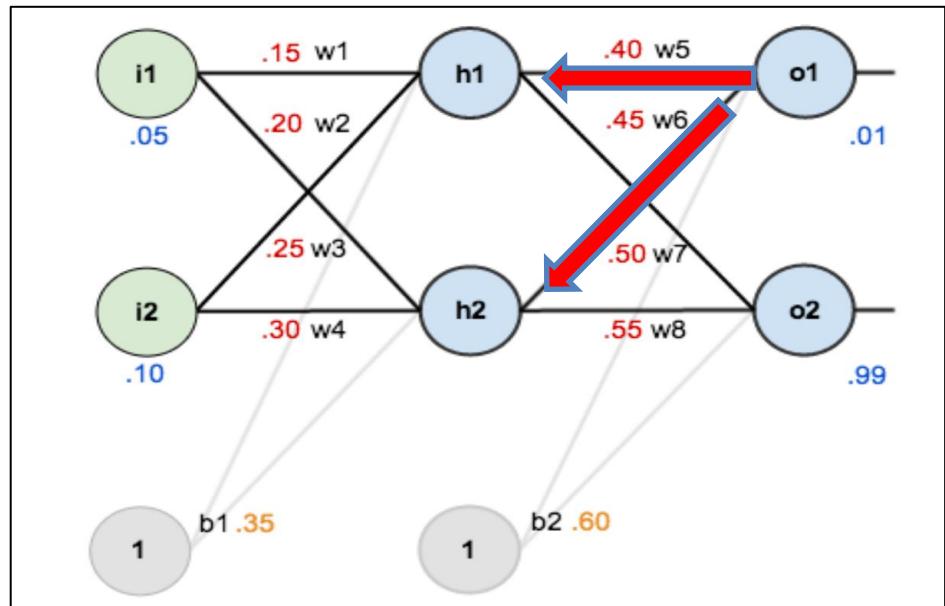
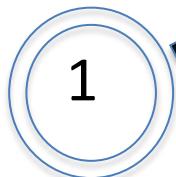
$$w6=0.45$$



$$h2 = 0.59688$$

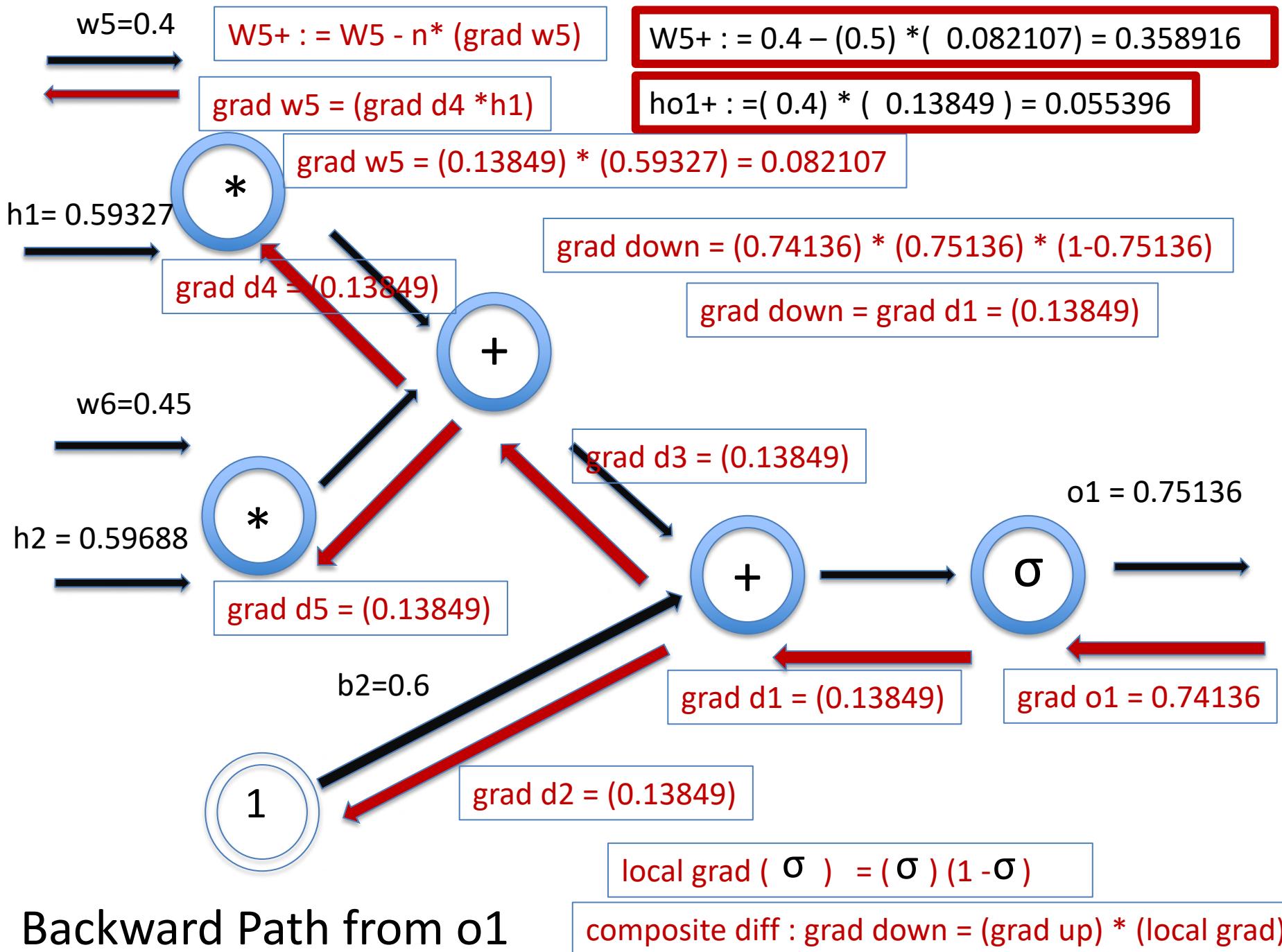


$$b2=0.6$$



$$o1 = 0.75136$$





# Backward Path from

$o_2$

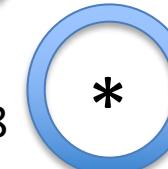
$w_7=0.5$



$h_1 = 0.59327$



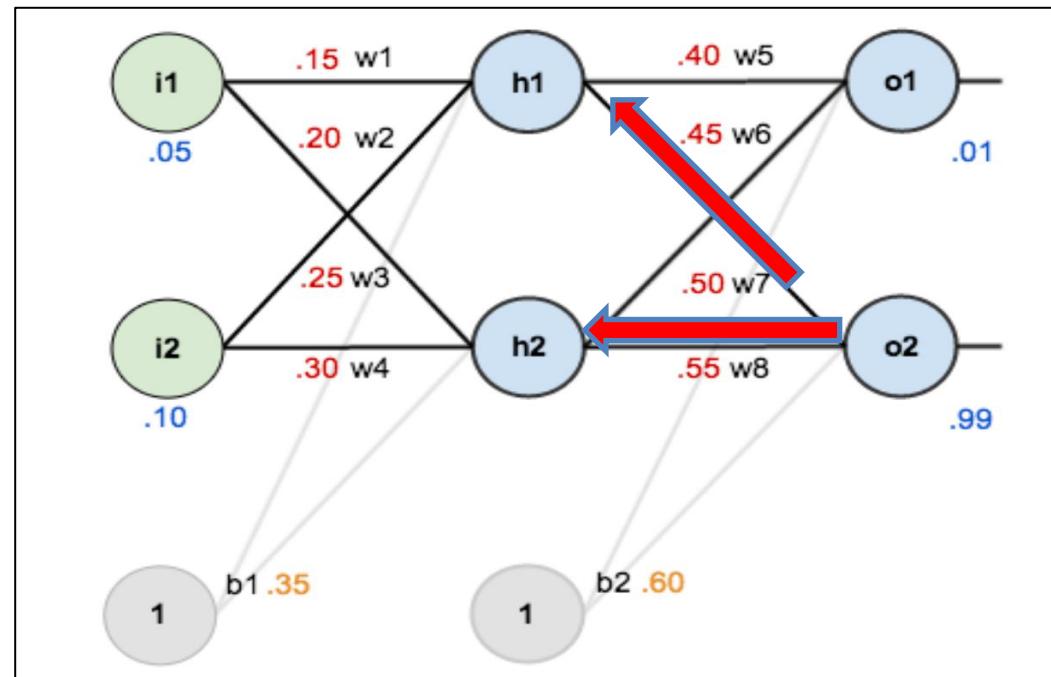
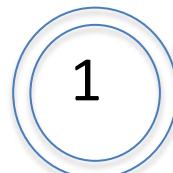
$w_8=0.55$



$h_2 = 0.59688$



$B_2=0.6$



$+$

$+$



$o_2 = 0.77293$



$w7=0.5$

$W7+ := W7 - n * (\text{grad } w7)$

$W7+ := 0.5 - (0.5) * (-0.022601) = 0.51130$

$\text{grad } w7 = (\text{grad } d9 * h1)$

$h02+ := (0.5) * (-0.038097) = -0.019485$

 $*$ 

$\text{grad } w7 = (-0.038097) * (0.59327) = -0.022601$

$n=0.5$

$h1 = 0.59327$

$\text{grad } d9 = (-0.038097)$

$\text{grad down} = (-0.21707) * (0.77293) * (1-0.77293)$

$\text{grad down} = \text{grad } d6 = (-0.038097)$

$w8=0.55$

 $+$ 

$\text{grad } d8 = (-0.038097)$

$o2 = 0.77293$

$h2 = 0.59688$

$\text{grad } d10 = (-0.038097)$

$b2=0.6$

 $+$ 

$\text{grad } d6 = (-0.038097)$

 $\sigma$ 

$\text{grad } o2 = -0.21707$

 $1$ 

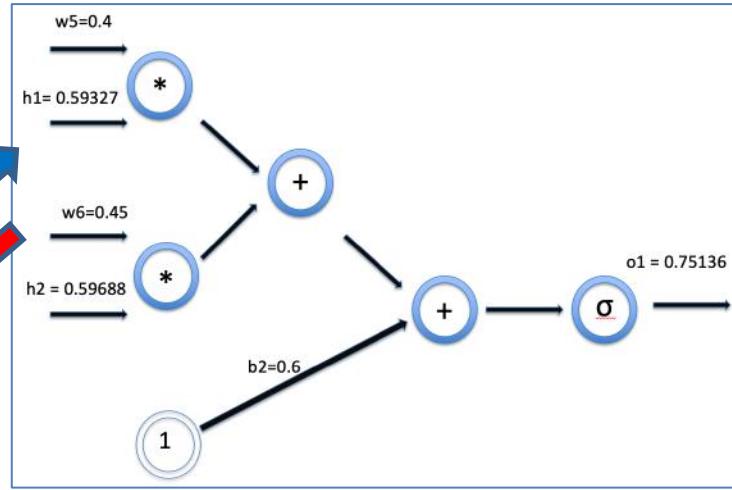
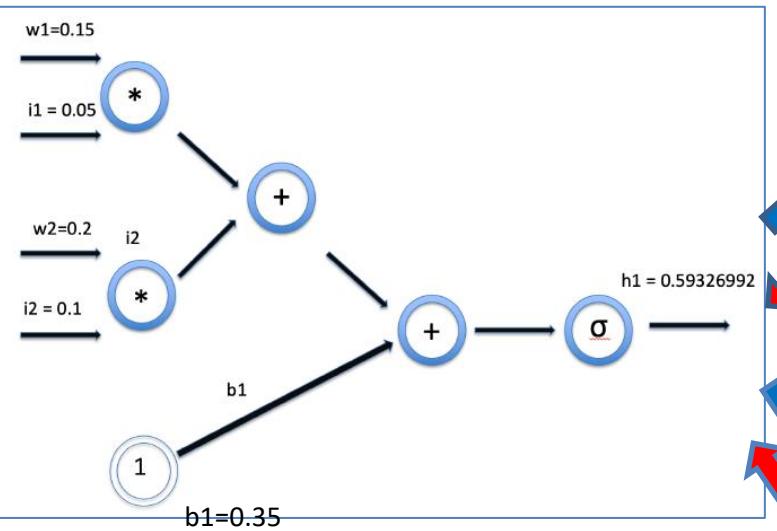
$\text{grad } d7 = (-0.038097)$

$\text{local grad } (\sigma) = (\sigma)(1-\sigma)$

Backward Path from  
 $o2$

$\text{composite diff : grad down} = (\text{grad up}) * (\text{local grad})$

# Backward Path from o1 and o2 to h1, from h1 to w1

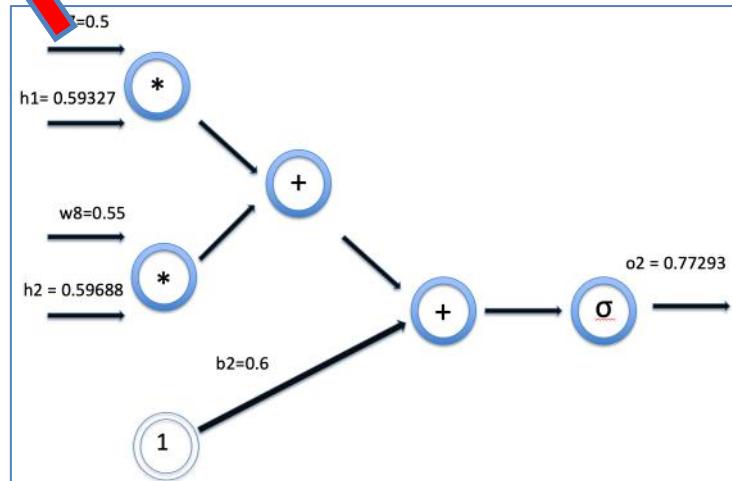
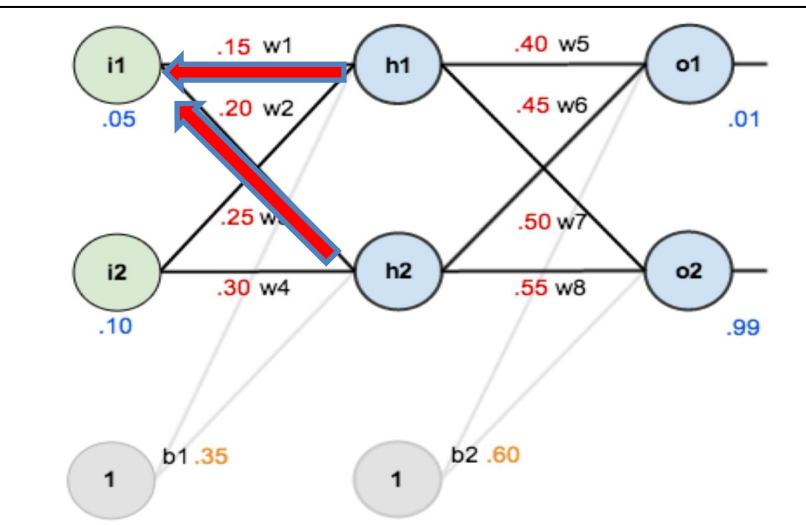


target=0.01

$o_1=0.75136$

$E_{o1}=0.27481$

$$E_{\text{total}} = E_{o1} + E_{o2} = 0.298371109$$



$E_{o2}=0.02356$

$o_2=0.77293$

target=0.99

$w1 = 0.15$

$W1+ := W1 - n * (\text{grad } w1)$

$W1+ := 0.15 - (0.5) * (0.0004332) = 0.149783$

$\text{grad } w1 = (\text{grad } d14 * i1)$

$\text{grad } w1 = (0.008665) * (0.05) = 0.00043326$

$i1 = 0.05$

$\text{grad } d14 = (0.008665)$

$\text{grad down} = (0.035911) * (0.59327) * (1 - 0.59327)$

$\text{grad down} = \text{grad } d1 = (0.008665)$

$w2 = 0.20$

$i2 = 0.1$

$\text{grad } d15 = (0.008665)$

### Backward Path from $h1$ to $w1$

$h1 = 0.59327$

 $\sigma$ 

$b1 = 0.35$

 $+$ 

$\text{grad } d11 = (0.008665)$

$\text{grad } h1 = 0.035911$

 $1$ 

$\text{grad } d12 = (0.008665)$

$\text{grad } h1-o1 := (0.4) * (0.13849) = 0.055396$

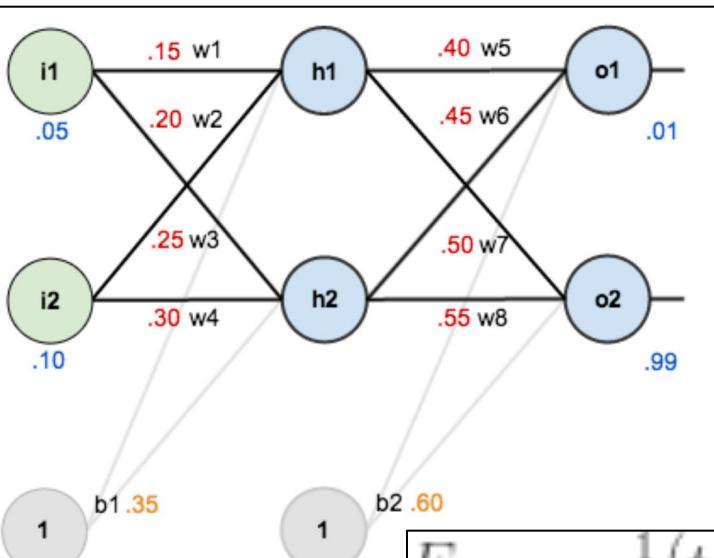
$\text{grad } h1-o2 := (0.5) * (-0.0038097) = -0.019485$

$\text{local grad } (\sigma) = (\sigma)(1 - \sigma)$

$\text{grad } h1 := (\text{grad } h1-o1) + (\text{grad } h1-o2) = (0.055396 - 0.019485)$

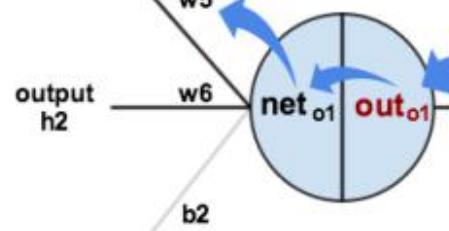
$\text{composite diff : grad down} = (\text{grad up}) * (\text{local grad})$

# Backpropagation Example



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

$$\frac{\partial E_{total}}{\partial w_5} = -\delta_{o1} out_{h1}$$

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

# Backpropagation Example

<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

$$\frac{\partial E_{total}}{\partial w_1} = \left( \sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}} \right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

$$w_2^+ = 0.19950143$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial E_{total}}{\partial w_1} = \left( \sum_o \delta_o * w_{ho} \right) * out_{h1}(1 - out_{h1}) * i_1$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

```

# Backpropagation
n = 0.5

grad_o1 = o1 - To1
grad_o2 = o2 - To2

print("grad_o1: "+str(grad_o1))
print("grad_o2: "+str(grad_o2))

grad_d1 = (grad_o1)*o1*(1-o1)
grad_d2 = (grad_o2)*o2*(1-o2)

print("grad_d1: "+str(grad_d1))
print("grad_d2: "+str(grad_d2))

grad_w5 = grad_d1*h1
print("grad_w5: "+str(grad_w5))

w5_final = w5 - n*grad_w5

grad_w6 = grad_d1*h2
print("grad_w6: "+str(grad_w6))
w6_final = w6 - n*grad_w6

grad_w7 = grad_d2*h1
print("grad_w7: "+str(grad_w7))
w7_final = w7 - n*grad_w7

grad_w8 = grad_d2*h2
print("grad_w8: "+str(grad_w8))
w8_final = w8 - n*grad_w8

```

```

grad_h1 = w5*grad_d1 + w7*grad_d2
print("grad_h1: "+str(grad_h1))
grad_d11 = grad_h1*h1*(1-h1)
print("grad_d11: "+str(grad_d11))

grad_w1 = grad_d11*i1
print("grad_w1: "+str(grad_w1))
w1_final = w1 - n*grad_w1

grad_w2 = grad_d11*i1
print("grad_w2: "+str(grad_w2))
w2_final = w2 - n*grad_w2

grad_h2 = w6*grad_d1 + w8*grad_d2
print("grad_h2: "+str(grad_h2))
grad_d22 = grad_h2*h2*(1-h2)
print("grad_d22: "+str(grad_d22))

grad_w3 = grad_d22*i1
print("grad_w3: "+str(grad_w3))
w3_final = w3 - n*grad_w3

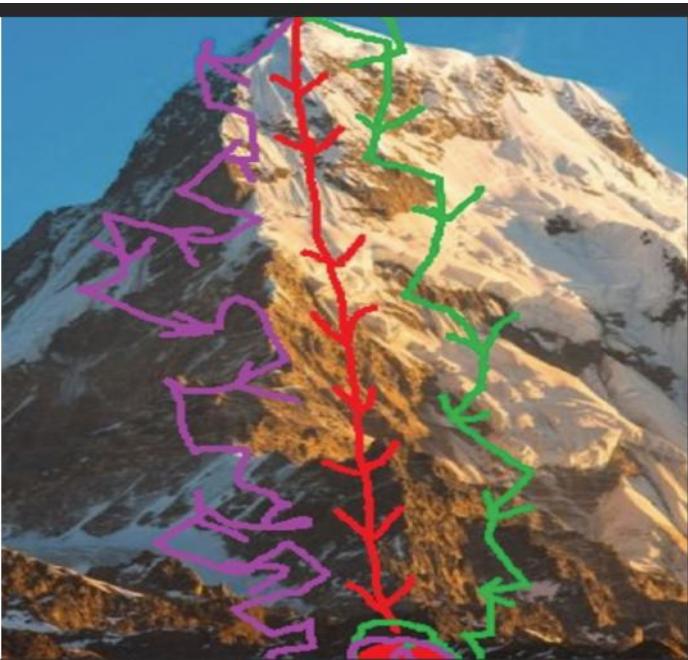
grad_w4 = grad_d22*i2
print("grad_w4: "+str(grad_w4))
w4_final = w4 - n*grad_w4

```

```
print("w1+: "+str(w1_final))
print("w2+: "+str(w2_final))
print("w3+: "+str(w3_final))
print("w4+: "+str(w4_final))
print("w5+: "+str(w5_final))
print("w6+: "+str(w6_final))
print("w7+: "+str(w7_final))
print("w8+: "+str(w8_final))
```

w1+: 0.1497807161327648  
w2+: 0.19978071613276482  
w3+: 0.24975114363237164  
w4+: 0.29950228726474326  
w5+: 0.35891647971775653  
w6+: 0.4086661860761087  
w7+: 0.5113012702391395  
w8+: 0.5613701211083925

grad\_o1: 0.7413650695475076  
grad\_o2: -0.21707153468357898  
grad\_d1: 0.13849856162945076  
grad\_d2: -0.03809823651803844  
grad\_w5: 0.08216704056448701  
grad\_w6: 0.08266762784778263  
grad\_w7: -0.02260254047827904  
grad\_w8: -0.02274024221678477  
grad\_h1: 0.036350306392761086  
grad\_d11: 0.00877135468940779  
grad\_w1: 0.0004385677344703895  
grad\_w2: 0.0004385677344703895  
grad\_h2: 0.04137032264833171  
grad\_d22: 0.009954254705134271  
grad\_w3: 0.0004977127352567136  
grad\_w4: 0.0009954254705134271



## Optimization : Derivatives and Gradient Descent : Simple Story

— Batch Gradient Descent  
— Mini-batch Gradient Descent  
— Stochastic Gradient Descent

Derivative is the slope of the tangent line

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

$x = 1$

The slope of the tangent line is 2

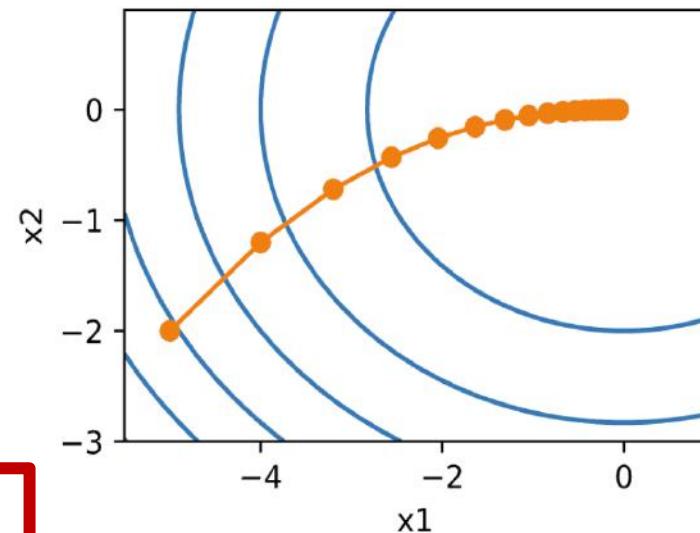
## Algorithm

- Choose initial  $\mathbf{x}_0$  ( $\mathbf{X} = \mathbf{W}$ , weight)
- At time  $t = 1, \dots, T$

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

- $\eta$  is called learning rate

$$\mathbf{W}+ = \mathbf{W} - (\text{learning rate}) * \frac{\partial J}{\partial \mathbf{w}}$$



# 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,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$



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

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



```
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

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

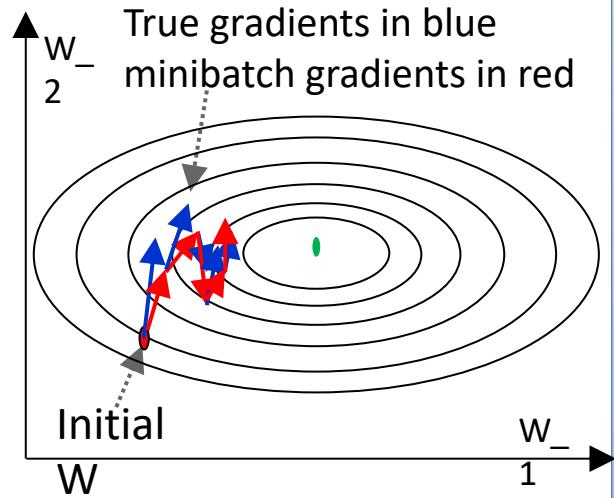
$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

# 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)

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



Given: training set  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$

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  $\mathcal{X} = \{(\mathbf{x}'_1, y'_1), \dots, (\mathbf{x}'_m, y'_m)\}$  without replacement

For each instance in  $\mathcal{X}$ ,  $(\mathbf{x}_k, y_k)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_k$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  via forward propagation

Compute  $\delta^{(L)} = \mathbf{a}^{(L)} - y_k$

Compute errors  $\{\delta^{(L-1)}, \dots, \delta^{(2)}\}$

Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute mini-batch regularized gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{m} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

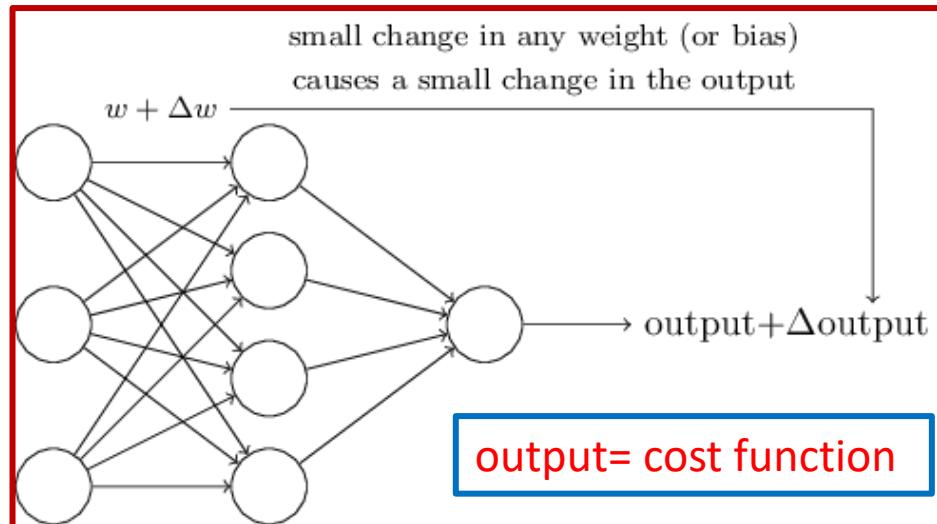
Update weights via gradient step  $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$

Until all training instances are seen

Until weights converge or max #epochs is reached

# Representation of NN Model – Math Story

neuralnetworksanddeeplearning.com



- ✓ 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 =>
- ✓  $\Delta C / \Delta w = dC / dw \Rightarrow$
- ✓  $\Delta C = (dC / dw) * \Delta w$

- ✓ 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 ( $\Delta w_j$ ) and bias ( $\Delta b$ ) cause a small change in the output ( $\Delta output$ ).
- ✓ So  $\Delta output$  is a *function* of the changes in  $\Delta w_j$  (weights) and  $\Delta b$  (bias), assume it to be *a linear function*, so

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

# 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  $\Delta w_j$  and  $\Delta b$  so as to make  $\Delta C$  (**Δoutput**) 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 =  $C$  ;  $w_j = v_1$  ;  $b = v_2$

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

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

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

Be negative

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

Just Choose

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

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

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

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

# 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  $\nabla C$  we need to compute the gradients  $\nabla C_x$  separately for each training input,  $x$ , and then average them,  $\nabla C = 1/n \sum \nabla C_x$ . 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  $\nabla C$  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  $\nabla C$ , and this helps speed up gradient descent, and thus learning, provided the sample size  $m$  is large enough that the average value of the  $\nabla C_{X_j}$  roughly equals to the average over all  $\nabla C_x$ .
- ✓ 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.

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

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

# 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  $x$
- ✓ Use gradient descent to find the weights  $w$  and biases  $b$  which minimize the cost function,  $C$ .
- ✓ To compute the gradient  $\nabla C$  we need to compute the gradients  $\nabla C_x$  separately for each training input,  $x$ , and then average them,  $\nabla C = 1/n \sum \nabla C_x$ . Unfortunately, when the number of training inputs is very large this can take a long time, and thus learning occurs slowly.
- ✓ A way is to use **stochastic gradient descent** to speed up learning. The idea is to estimate the gradient  $\nabla C$  by computing a small sample of randomly chosen training inputs, refer to as a **mini-batch** of input (mini-batched SGD).
- ✓ By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient  $\nabla C$ , and this helps speed up gradient descent, and thus learning, provided the sample size  $m$  is large enough that the average value of the  $\nabla C_{X_j}$  roughly equals to the average over all  $\nabla C_x$ .
- ✓ Another randomly chosen mini-batch are selected and trained, until all the exhausted the training inputs are used. It is said to complete an **epoch (iteration)** of training, then more iteration.

# 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
```

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

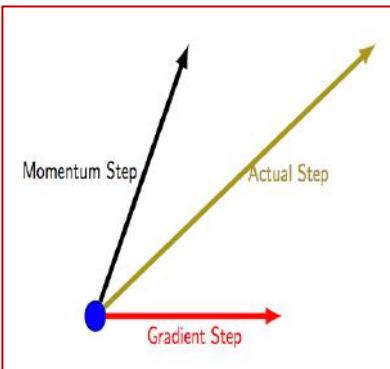
## Mini-batch Stochastic Gradient Decent

```
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.

# Gradient-based Optimization with Momentum



- ✓ Introduce a new variable  $v$ , the velocity:
- ✓ of  $v$  as the direction and speed by which the parameters move as the learning dynamics progresses
- ✓ Velocity is an exponentially decaying moving average of the negative gradients

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

<https://ruder.io/optimizing-gradient-descent/index.html>

<https://www.seas.upenn.edu/~cis519/spring2019/>

- How do we try and solve this problem?
- Introduce a new variable  $v$ , the velocity
- We think of  $v$  as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an **exponentially decaying moving average of the negative gradients**

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left( L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- $\alpha \in [0, 1]$  Update rule:  $\theta \leftarrow \theta + \mathbf{v}$

## Algorithm 2 Stochastic Gradient Descent with Momentum

**Require:** Learning rate  $\epsilon_k$

**Require:** Momentum Parameter  $\alpha$

**Require:** Initial Parameter  $\theta$

**Require:** Initial Velocity  $\mathbf{v}$

- 1: **while** stopping criteria not met **do**
- 2:     Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set
- 3:     Compute gradient estimate:
- 4:      $\hat{\mathbf{g}} \leftarrow +\nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
- 5:     Compute the velocity update:
- 6:      $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$
- 7:     Apply Update:  $\theta \leftarrow \theta + \mathbf{v}$
- 8: **end while**

# 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.

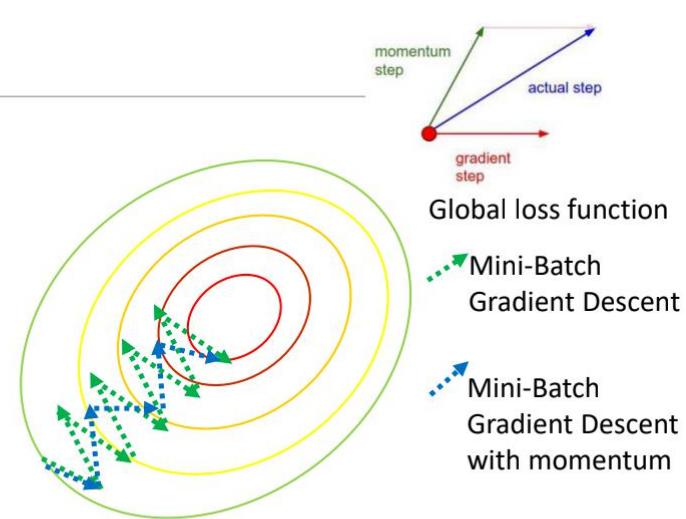
The velocity is basically an exponentially decaying average of the step size:

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

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.



## 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:  
✓  $\text{beta1} = 0.9$   
✓  $\text{beta\_2} = 0.99$ .

### 2. NADAM(Nesterov-accelerated Adaptive Moment Estimation):

#### 1. Use ADAM with Nesterov momentum

# Optimizer : ADAM

**Require:**  $\alpha$ : Stepsize

**Require:**  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates

**Require:**  $f(\theta)$ : Stochastic objective function with parameters  $\theta$

**Require:**  $\theta_0$ : Initial parameter vector

$m_0 \leftarrow 0$  (Initialize 1<sup>st</sup> moment vector)

$v_0 \leftarrow 0$  (Initialize 2<sup>nd</sup> moment vector)

$t \leftarrow 0$  (Initialize timestep)

**while**  $\theta_t$  not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)

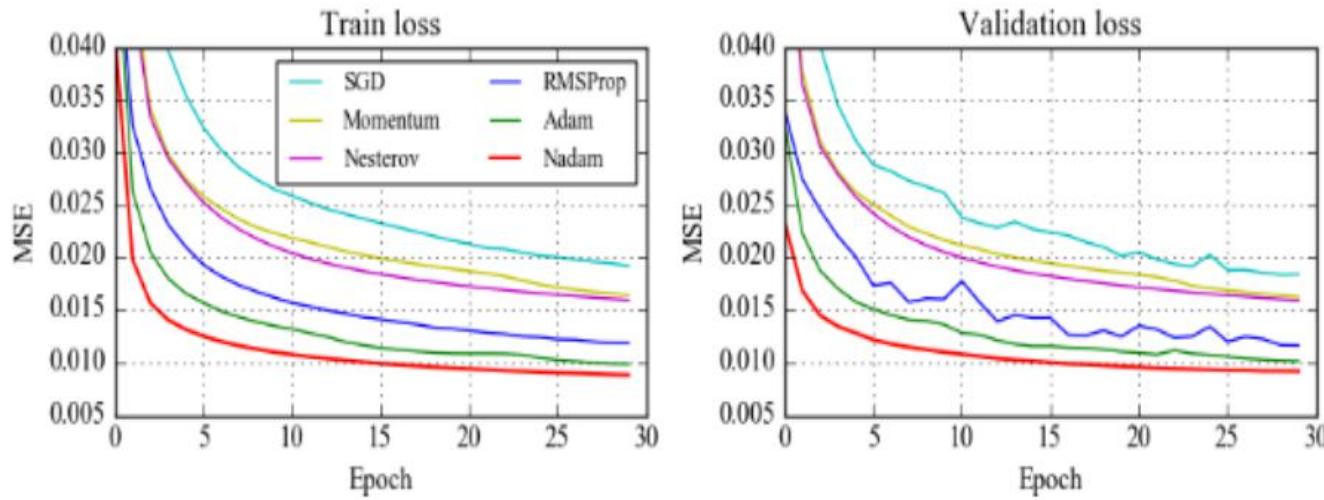
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

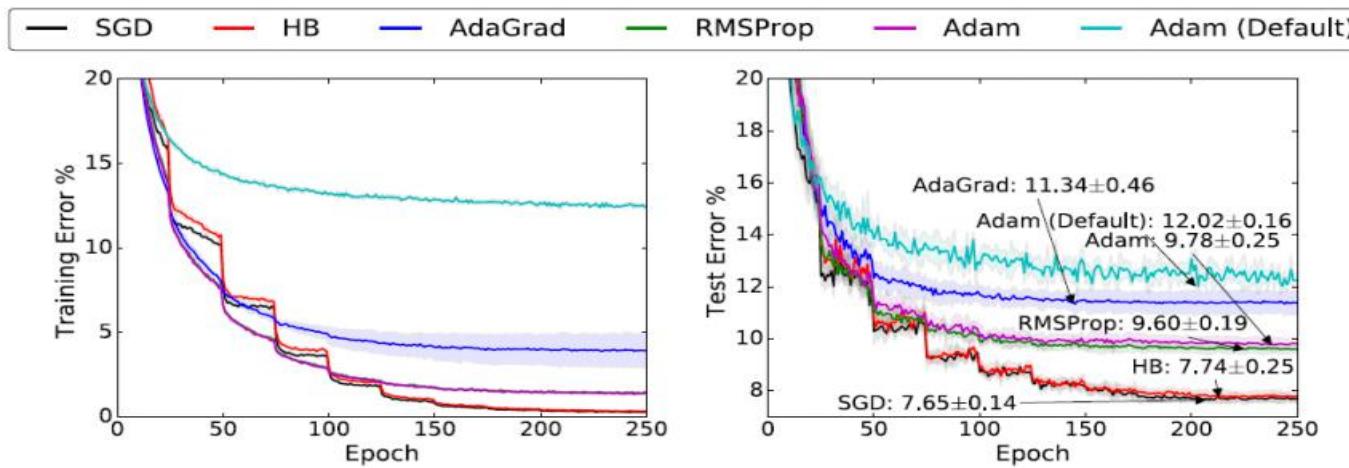
**end while**

**return**  $\theta_t$  (Resulting parameters)



2016

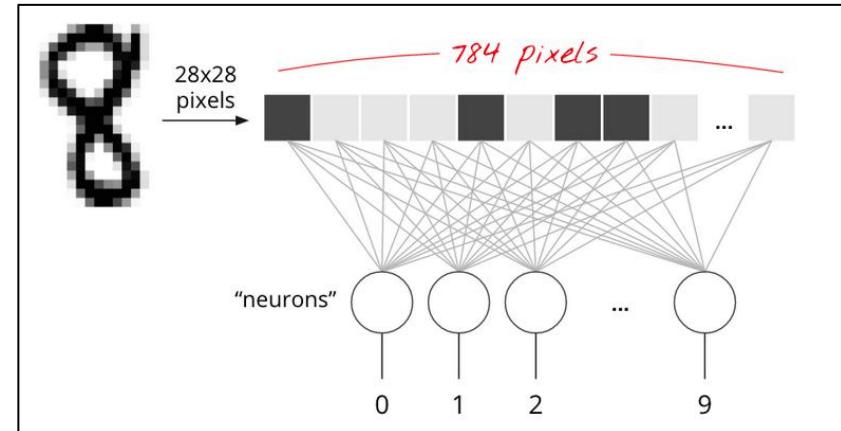
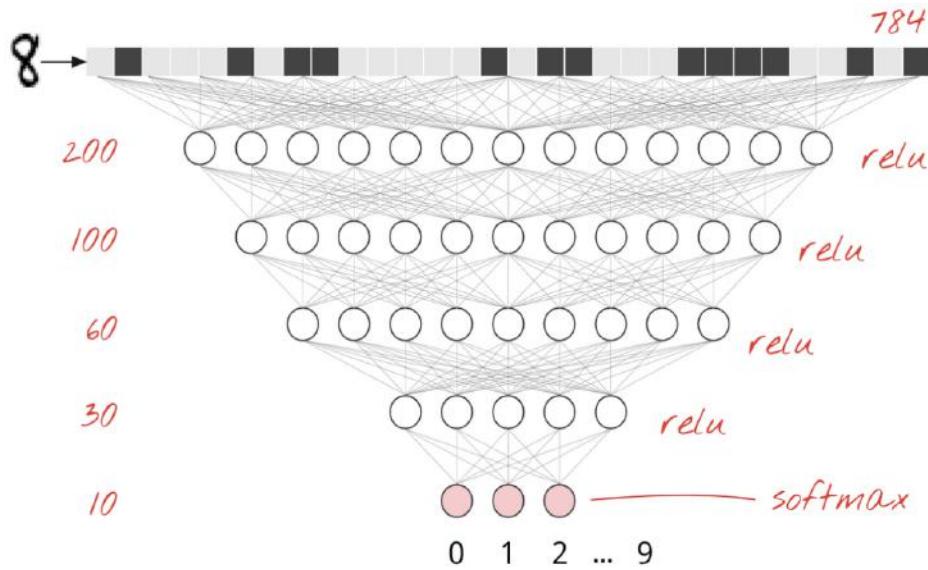
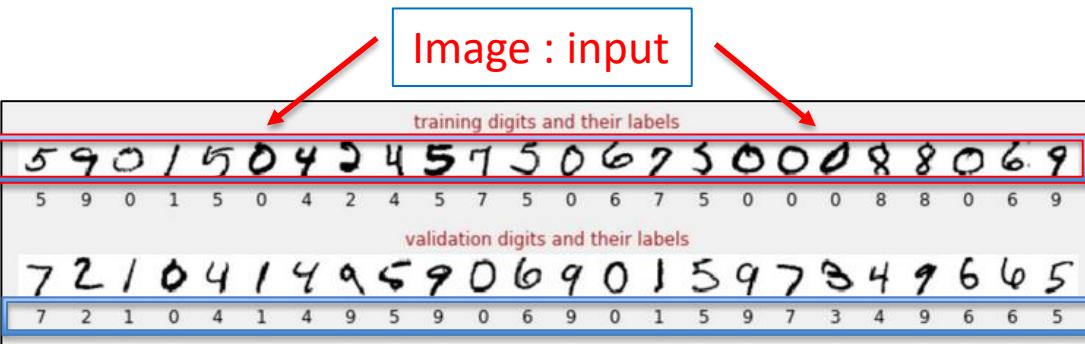
Dozat, Timothy. "Incorporating Nesterov Momentum into Adam." (2016).



Wilson, Ashia C., et al. "The marginal value of adaptive gradient methods in machine learning." *arXiv preprint arXiv:1705.08292* (2017).

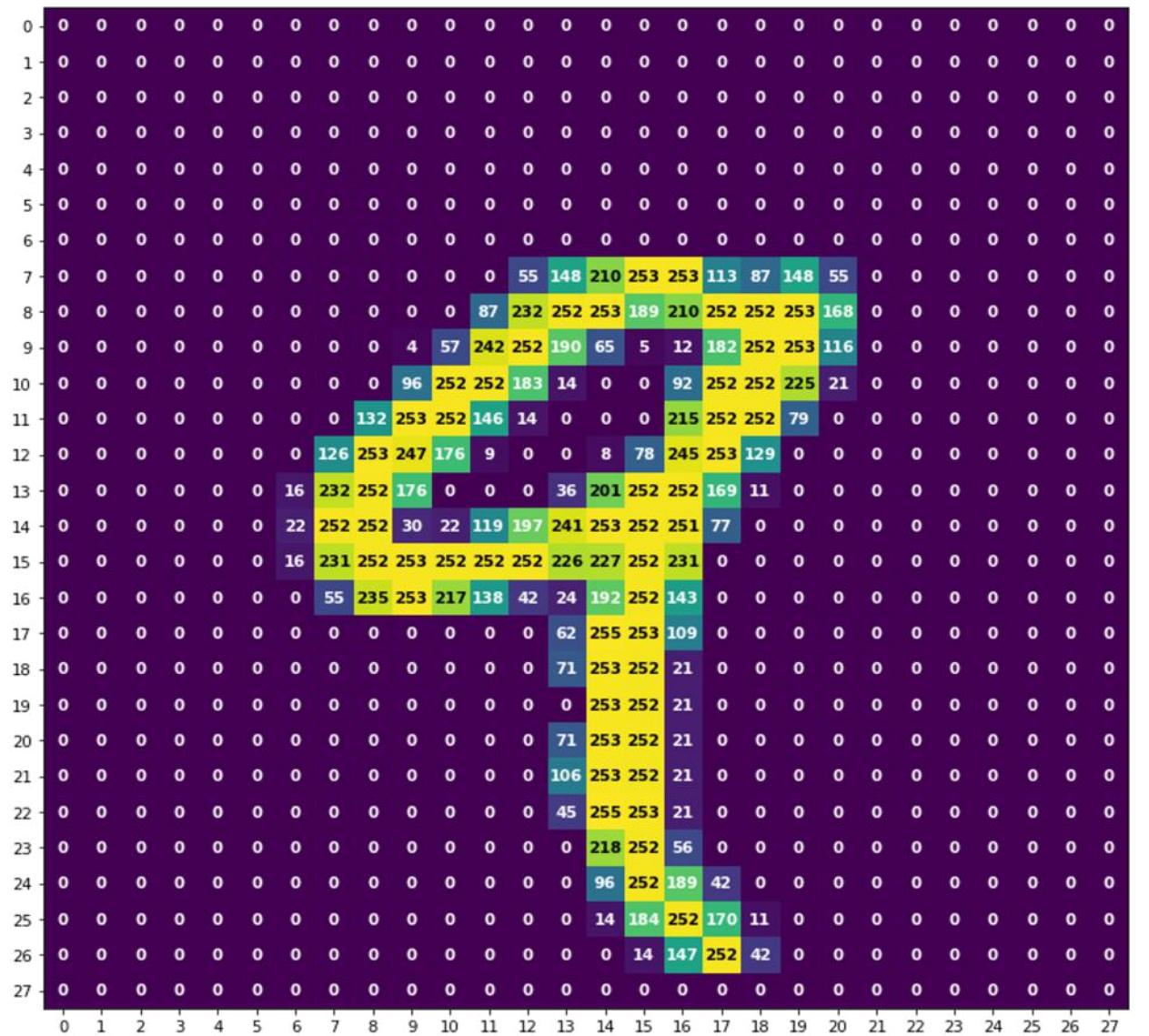
# MNIST Example (28x28 pixels)

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images. The simplest approach for classifying them is to use the  $28 \times 28 = 784$  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>



## 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

Input

# Simple MNIST MLP Neural Network

output

labels

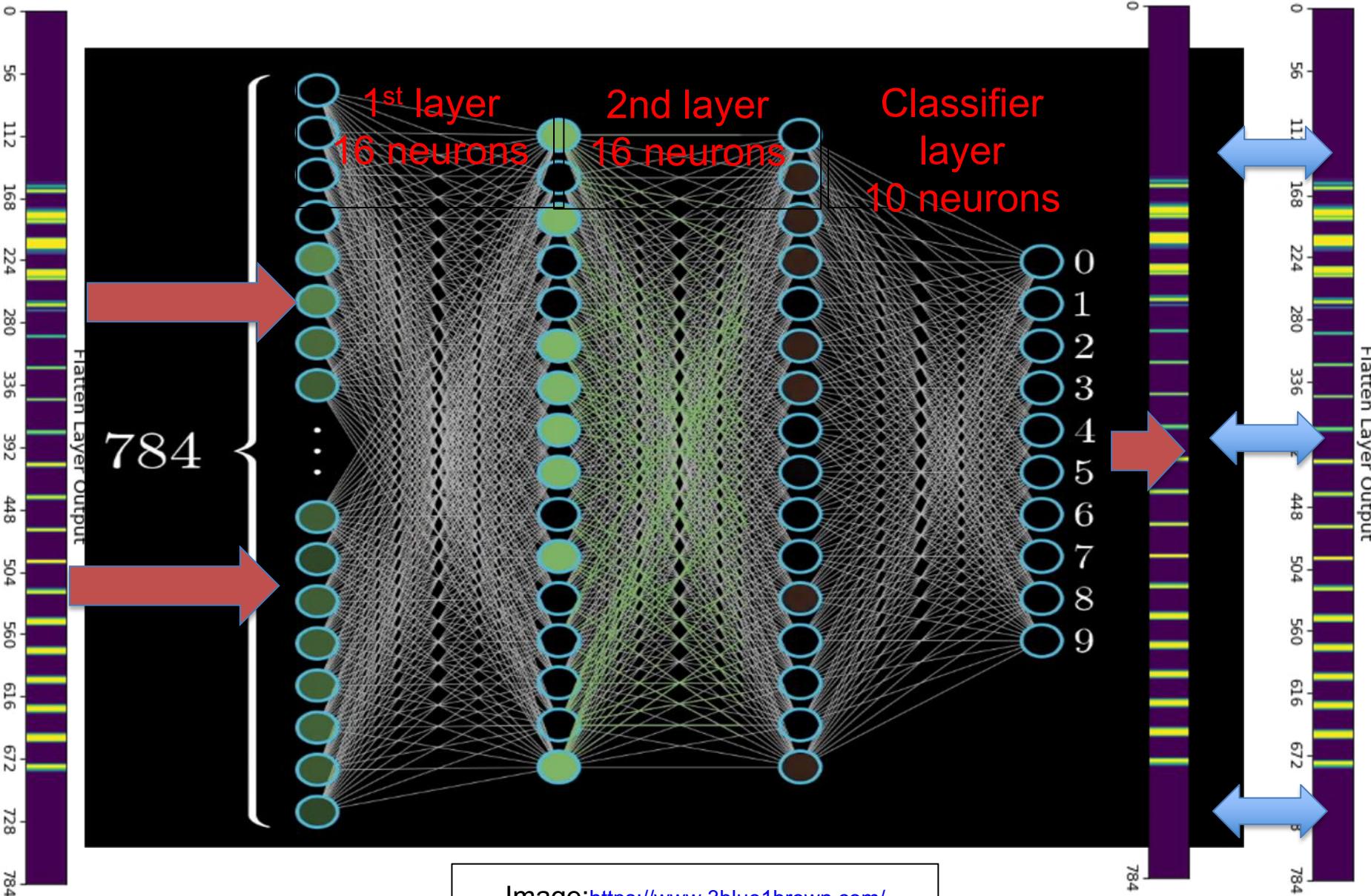


Image:<https://www.3blue1brown.com/>

# 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'])
```

Loss Function

```
model.summary()
```

Compiles the model with the SGD optimizer

```
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)
```

Print summary

Use tensorborad

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

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

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

**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:localhost/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 for 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
```

```

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)

model.fit(x_train, y_train, epochs=20, batch_size=50, validation_data=(x_test, y_test),
          callbacks=[tensorboard_callback])

model.evaluate(x_test, y_test, verbose=2)

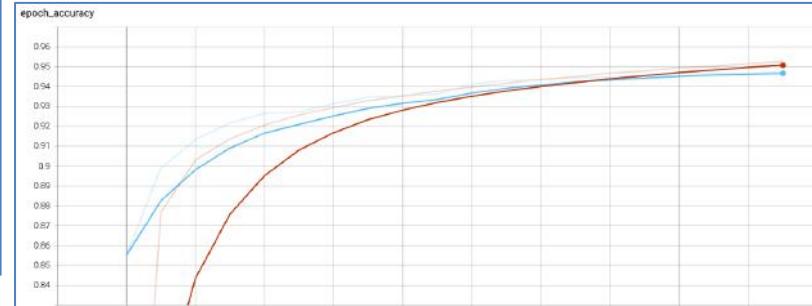
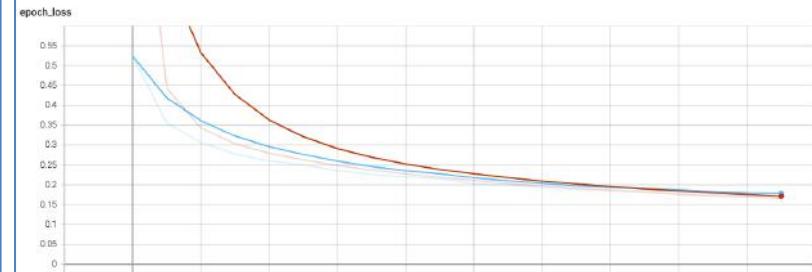
%tensorboard --logdir logs

```

2.4.1  
The tensorboard extension is already loaded. To reload it, use:  
%reload\_ext tensorboard  
Model: "sequential\_1"

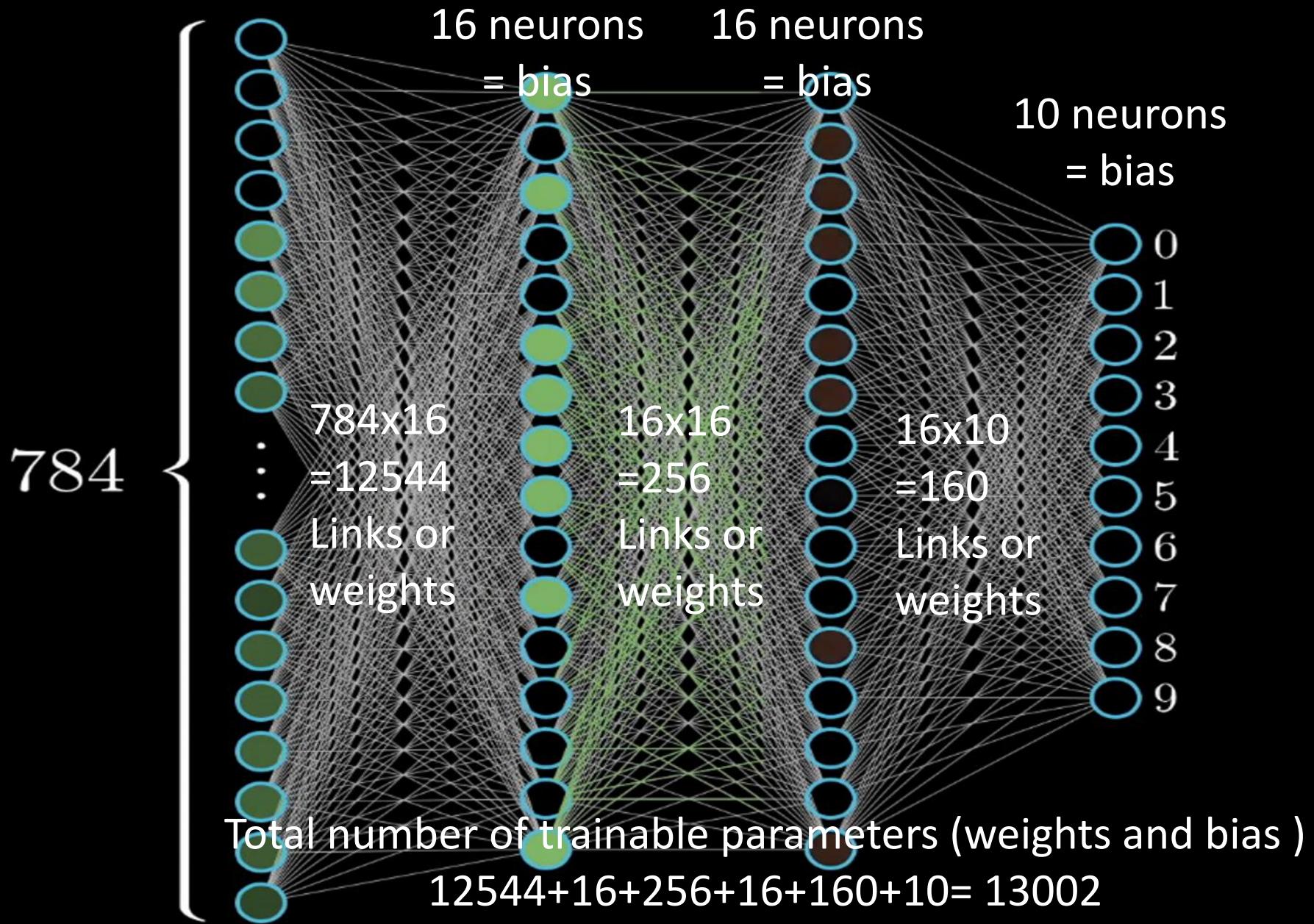
Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_3 (Dense)	(None, 16)	12560
dense_4 (Dense)	(None, 16)	272
dense_5 (Dense)	(None, 10)	170

Total params: 13,002  
Trainable params: 13,002  
Non-trainable params: 0

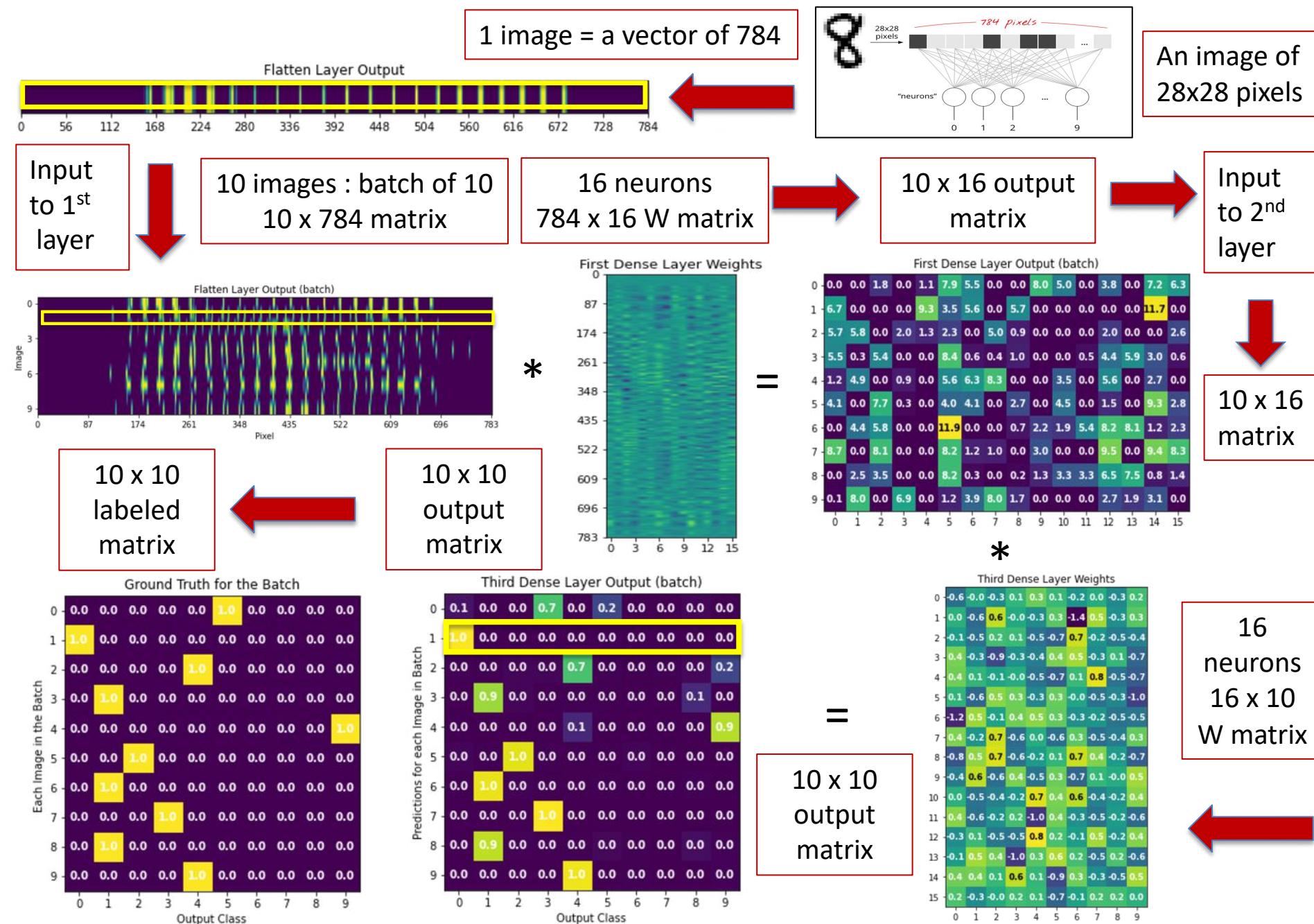


## Simple MLP Network

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

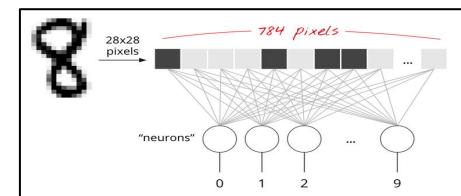


# Summary : Flow of MLP Dense layer NN



# Summary : Flow of MLP Dense layer NN

1 image = a vector of 784

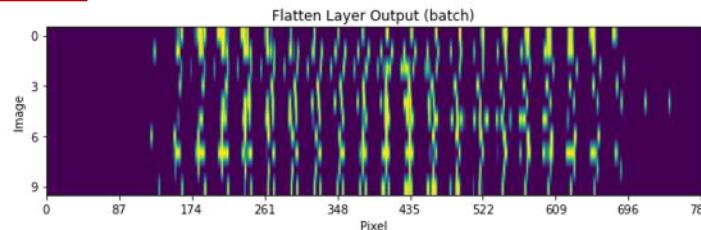


An image of 28x28 pixels

Input to 1<sup>st</sup> layer

10 images : batch of 10  
10 x 784 matrix

10 x 10 labeled matrix



10 x 10 output matrix

Ground Truth for the Batch

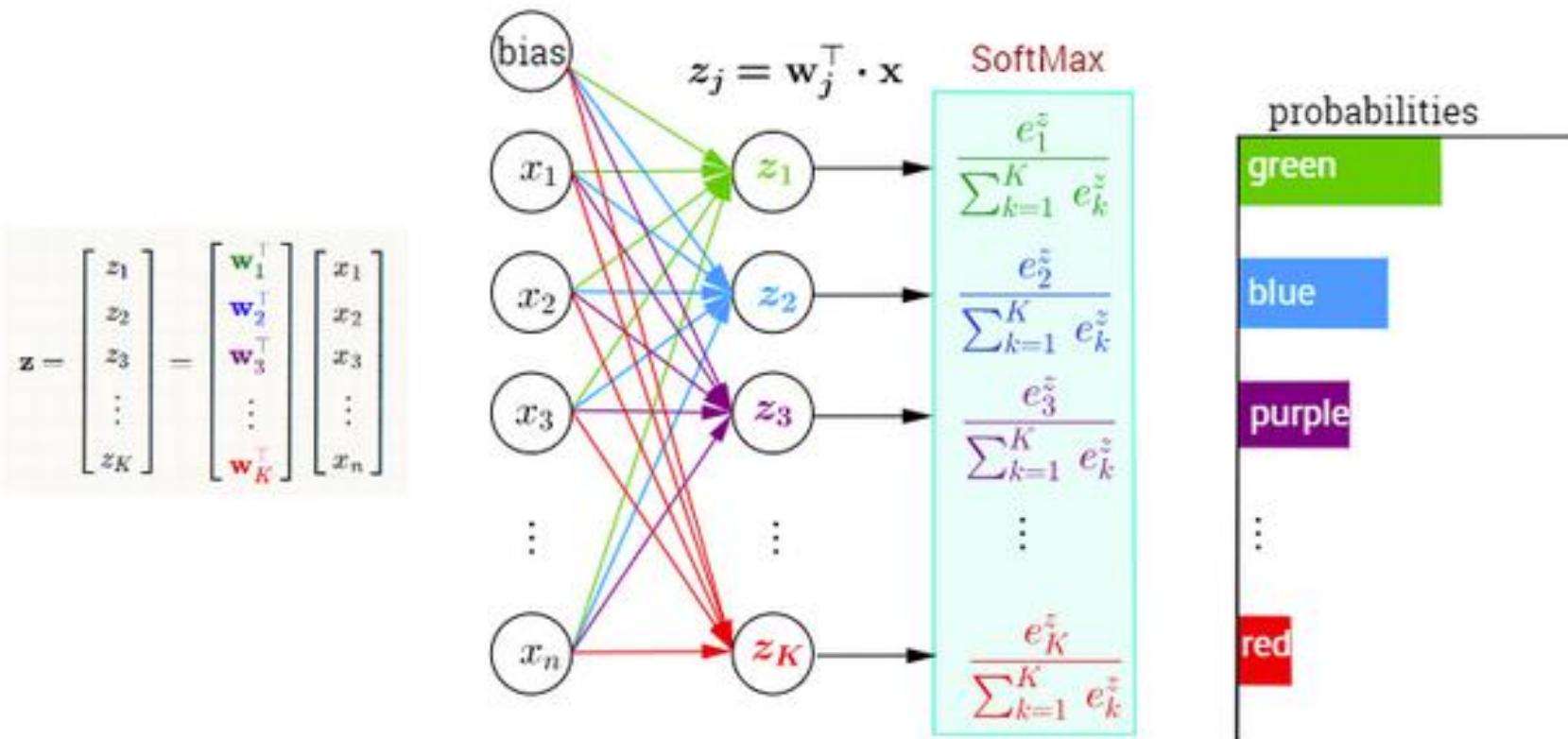
Each Image in the Batch	0	1	2	3	4	5	6	7	8	9	Output Class
0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	4
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
2	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	3
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	9
5	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	2
6	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
7	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	3
8	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
9	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	4

Comparing  
Computed NN  
results  
To  
Labeled results  
(target)

Third Dense Layer Output (batch)

Predictions for each Image in Batch	0	1	2	3	4	5	6	7	8	9	Output Class
0	0.1	0.0	0.0	0.7	0.0	0.2	0.0	0.0	0.0	0.0	3
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
2	0.0	0.0	0.0	0.0	0.0	0.7	0.0	0.0	0.0	0.2	2
3	0.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0
4	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.9	4
5	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
6	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
7	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	3
8	0.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
9	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	4

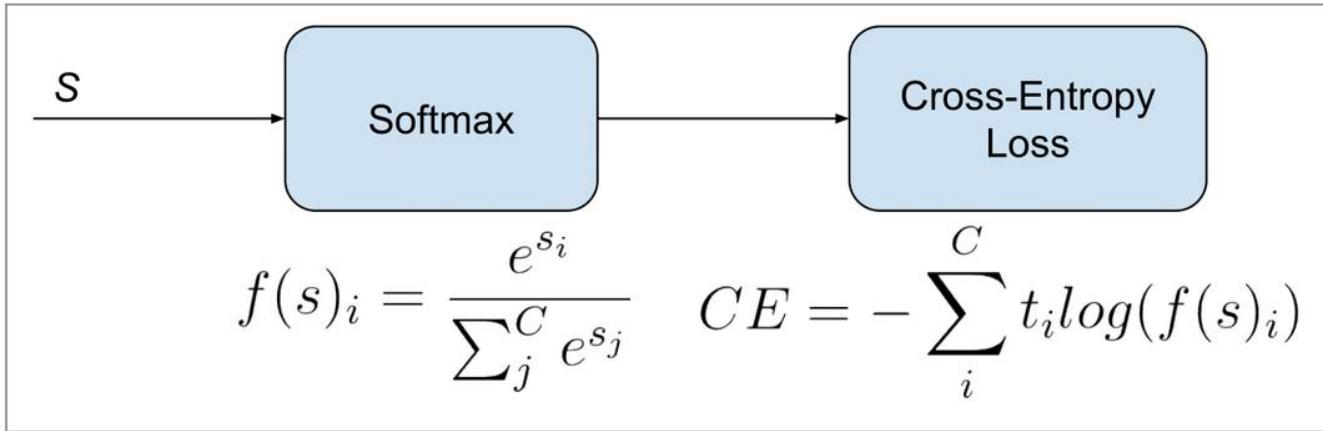
# Multi-Class Classification with NN and SoftMax Function



## Softmax Function

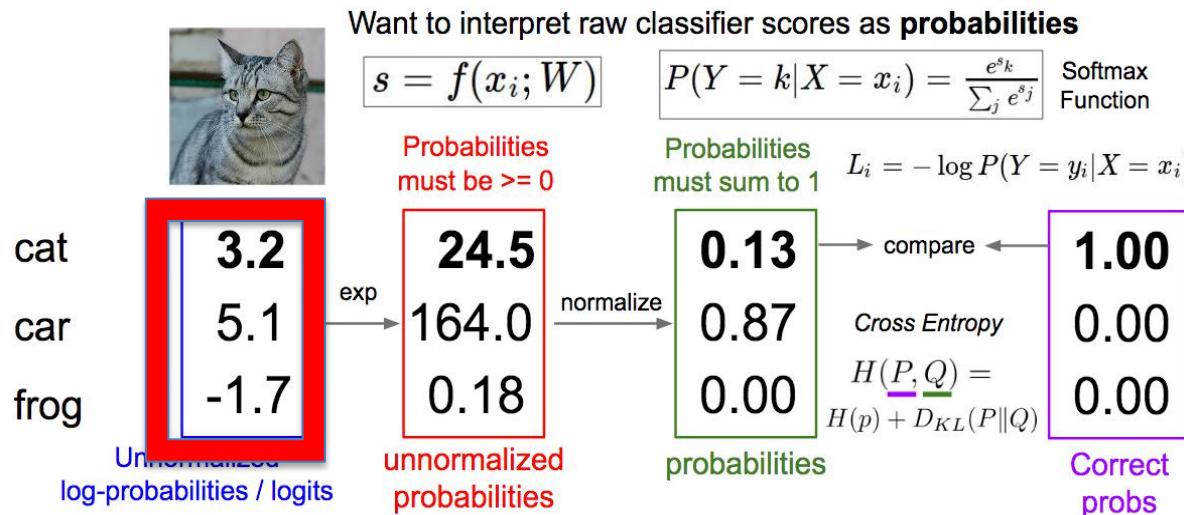
$$\sigma(j) = \frac{\exp(\mathbf{w}_j^\top \mathbf{x})}{\sum_{k=1}^K \exp(\mathbf{w}_k^\top \mathbf{x})} = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)}$$

# Categorical (Softmax) Cross Entropy Loss (Statistical Learning)



$t_i$  and  $s_i$  are the groundtruth (label) and the computed score for each class  $i$  in  $C$ .

## Softmax Classifier (Multinomial Logistic Regression)



$$e^{3.2} = 24.5$$

$$e^{5.1} = 164.0$$

$$e^{-1.7} = 0.18$$

---

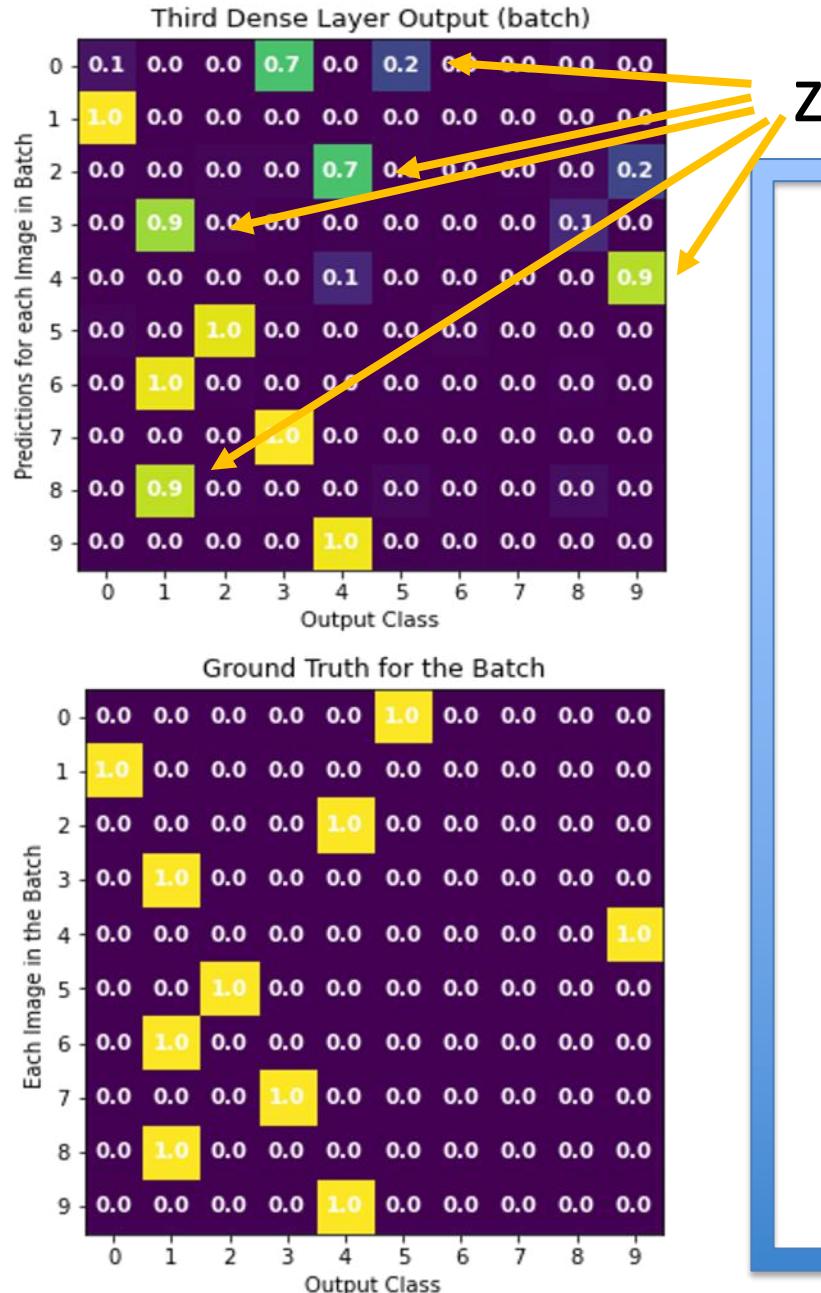
$$188.68$$

Software =  $24.5 / 188.68 = 0.13$

Software =  $164 / 188.68 = 0.87$

Software =  $-1.7 / 188.68 = 0.00$

# Evaluating the Error



# Categorical Cross Entropy Loss

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

$$L(i) = -\log 0.2 = 1.61$$

Image 0 = 5

$$-\log 1.0 = 0$$

Image 1 = 0

$$-\log 0.7 = 0.356$$

Image 2 = 4

$$-\log 0.9 = 0.105$$

Image 3 = 1

$$-\log 0.9 = 0.105$$

Image 4 = 9

$$-\log 1.0 = 0$$

Image 5 = 2

$$-\log 1.0 = 0$$

Image 6 = 1

$$-\log 1.0 = 0$$

Image 7 = 3

$$-\log 0.9 = 0.105$$

Image 8 = 2

$$-\log 1.0 = 0$$

Image 9 = 4

# Summary : Flow of NN

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

Define Network

Forward Pass

Calculate the analytical gradients

Update weights and bias

backpropagation

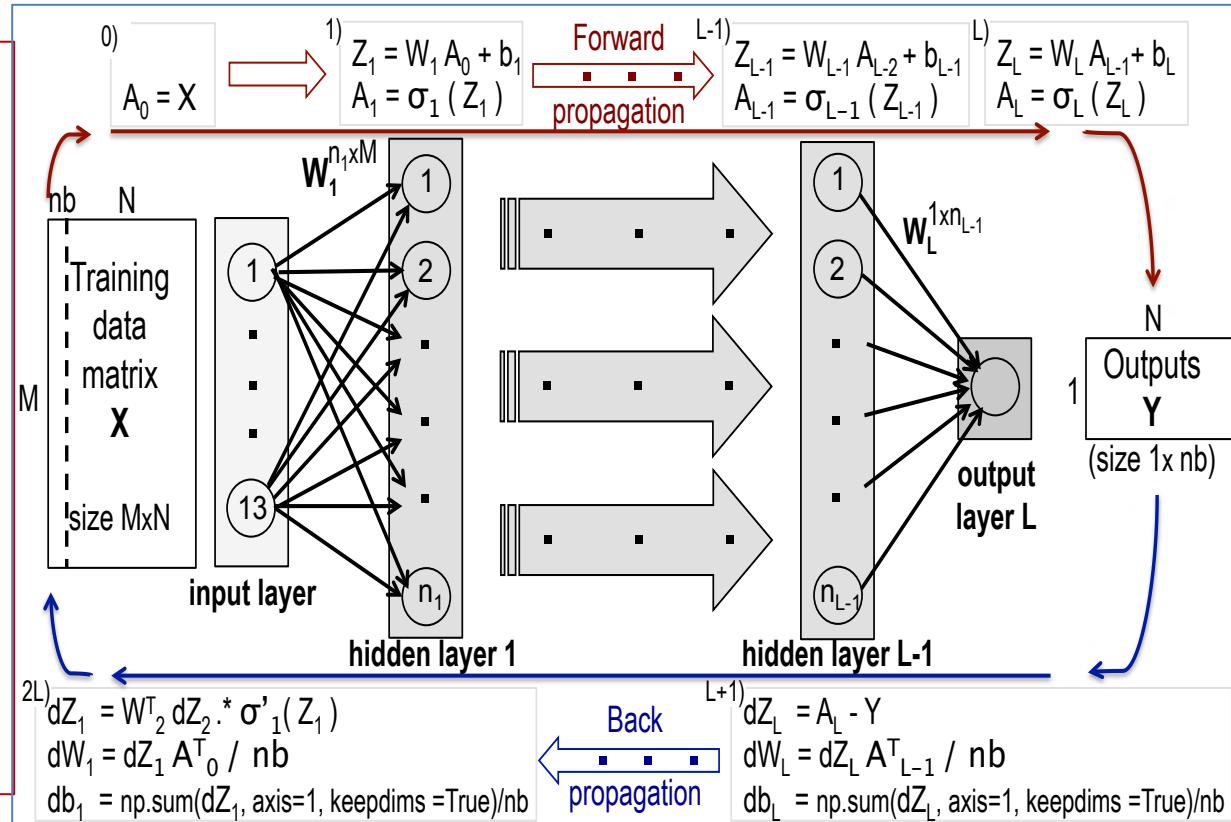
Gradient decent

Quantify loss

```

1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2

```



Exercise : Represent the following NN model as a computational graph and build a python code to update w and b

Input

$x_1 = -0.04$

$x_2 = -0.42$

NN Model

$b_1 = -1.6$

$b_2 = 0.7$

Output

$S(z)$   
 $b_3 = 0$

$S(z)$   
 $b_4 = 0$

$S(z)$   
 $b_5 = 1$

Loss

CCE

CCE

CCE

Target

$T = 0$

$T = 1$

$T = 0$

Categorical (Softmax) Cross Entropy Loss

$w_1 = -2.5$

$w_2 = -1.5$

$w_3 = 0.6$

$w_4 = 0.4$

$w_5 = -0.1$

$w_6 = 2.4$

$w_7 = -2.2$

$w_8 = -1.5$

$w_9 = -5.2$

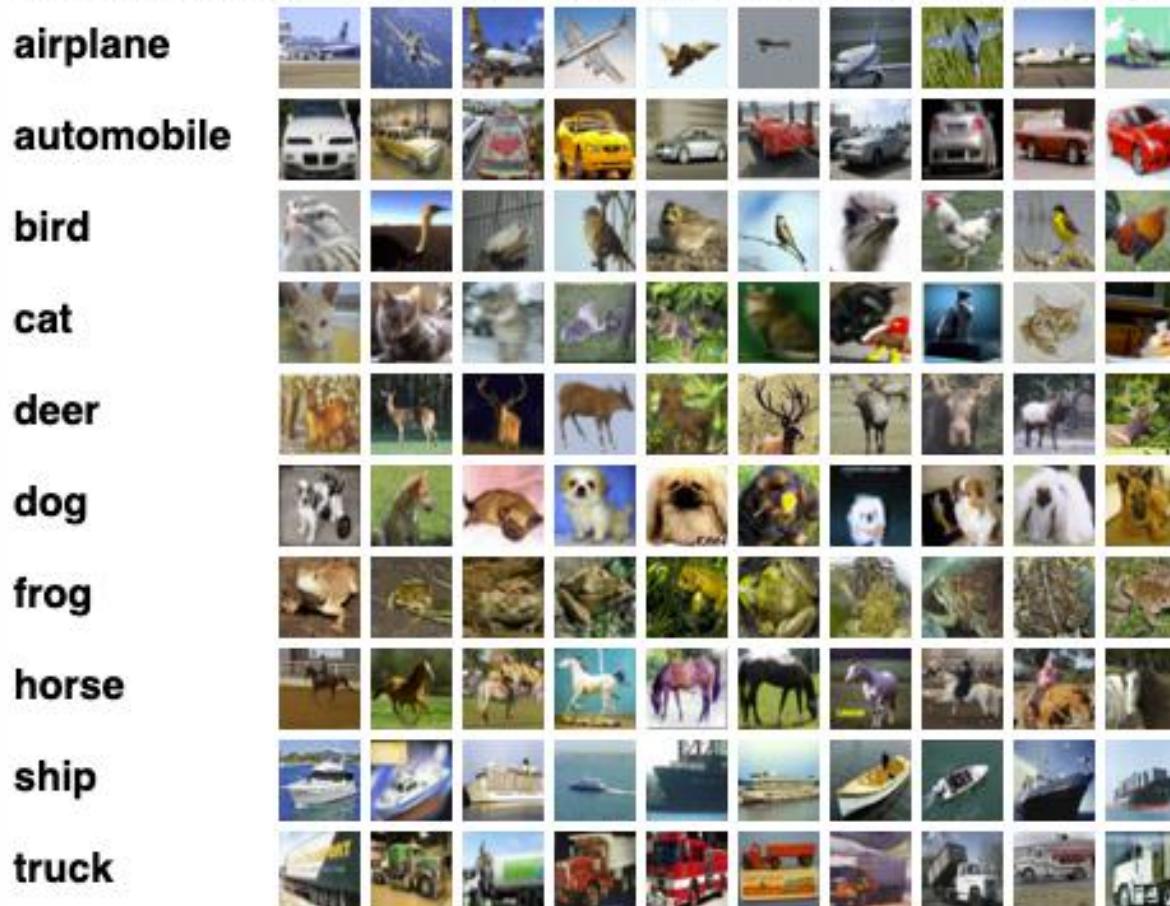
$w_{10} = 3.7$

CE : Categorical Cross Entropy Loss  
Classification  
0, 1, 2, ...., 9 MNIST Example  
Image Recognition

Convolutional Neural Network

The **CIFAR-10 dataset** ([Canadian Institute For Advanced Research](#)) is a collection of images that are commonly used to train [machine learning](#) and [computer vision](#) algorithms. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes. There are 6,000 images of each class. Since the images in CIFAR-10 are low-resolution (32x32), this dataset can allow researchers to quickly try different algorithms to see what works. CIFAR-10 is a labeled subset of the [80 million tiny images](#) dataset. When the dataset was created, students were paid to label all of the images.

Here are the classes in the dataset, as well as 10 random images from each:

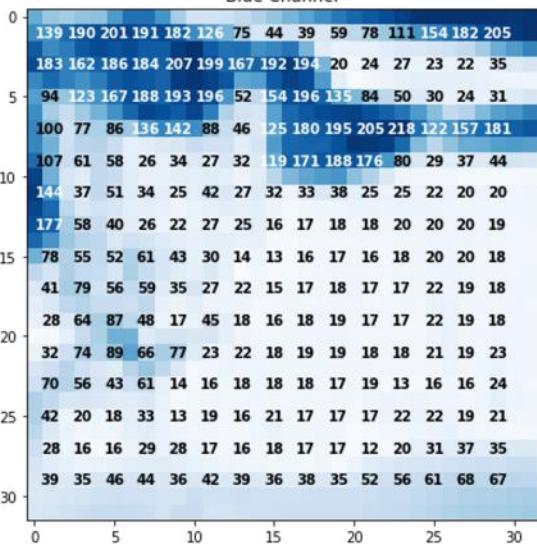
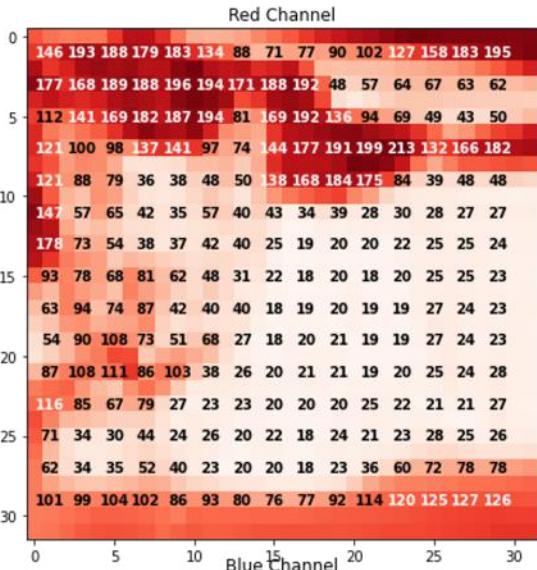
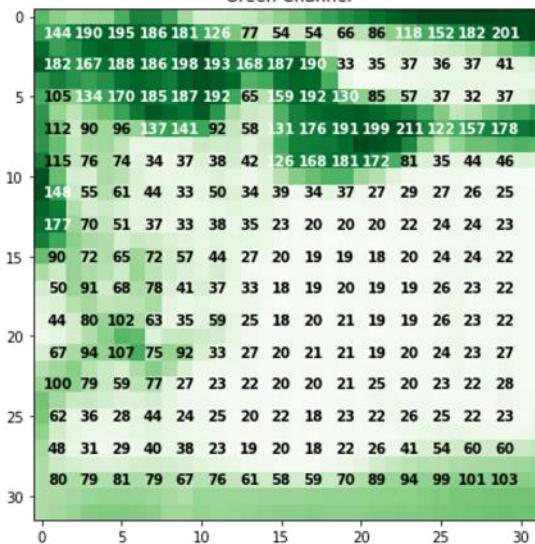
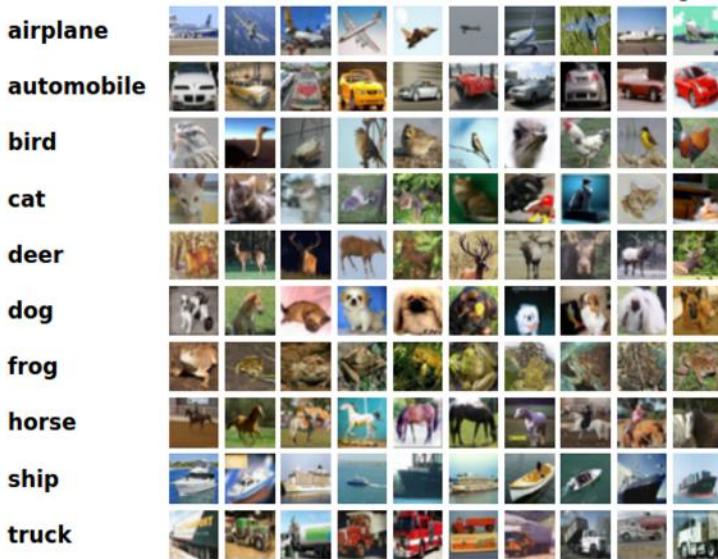


# TensorFlow 2.0 : CIFAR 10 Dataset

## Loading the CIFAR10 Dataset

```
import tensorflow as tf  
  
cifar10 = tf.keras.datasets.cifar10  
  
#download the images  
  
(x_train, y_train), (x_test, y_test)  
= cifar10.load_data()
```

- ✓ Each Image is a 32x32 image of one of 10 objects
- ✓ There are 50,000 training images and 10000 testing images



```
#normalize
```

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

# TensorFlow 2.0

## The CNN Network

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(4, (3, 3), activation='relu',
    input_shape=(32, 32, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(4, (3, 3), activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

```
loss_fn =
tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

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

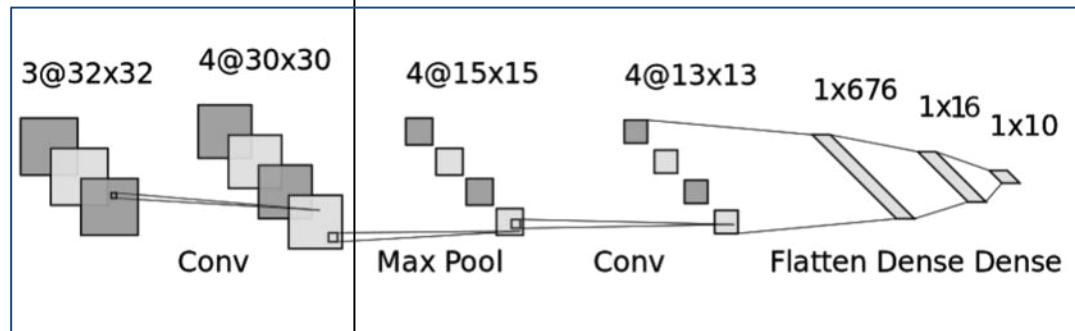
Let's use this simple network to demonstrate what TensorFlow is doing behind the scenes

**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 ( $50,000/10=5,000$ )

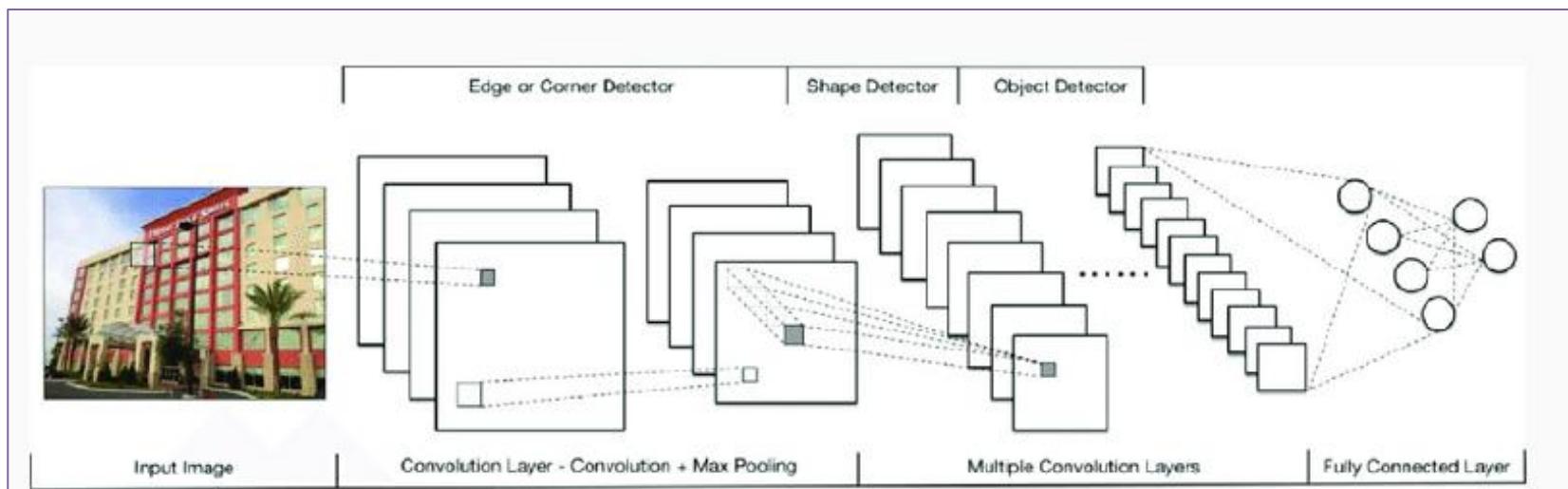
### Simple CNN Network



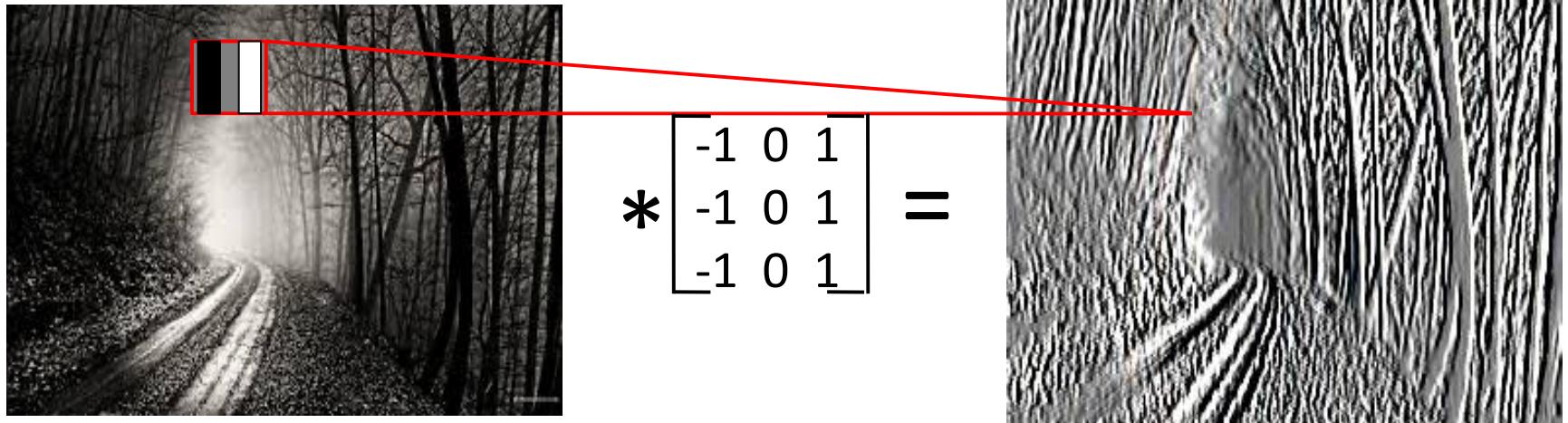
# Convolutional Neural Network

- ✓ **Convolutional Neural Network Introduction**
- ✓ **CNN example, Convolution operations**
- ✓ **MM in convolution operations**

- ✓ **CNN framework** :a class of neural network where the input is modified by a filter
- ✓ Multiple filters in convolutional networks are applied to different slices of an image, mapping them one by one, and identifying different features of an input image



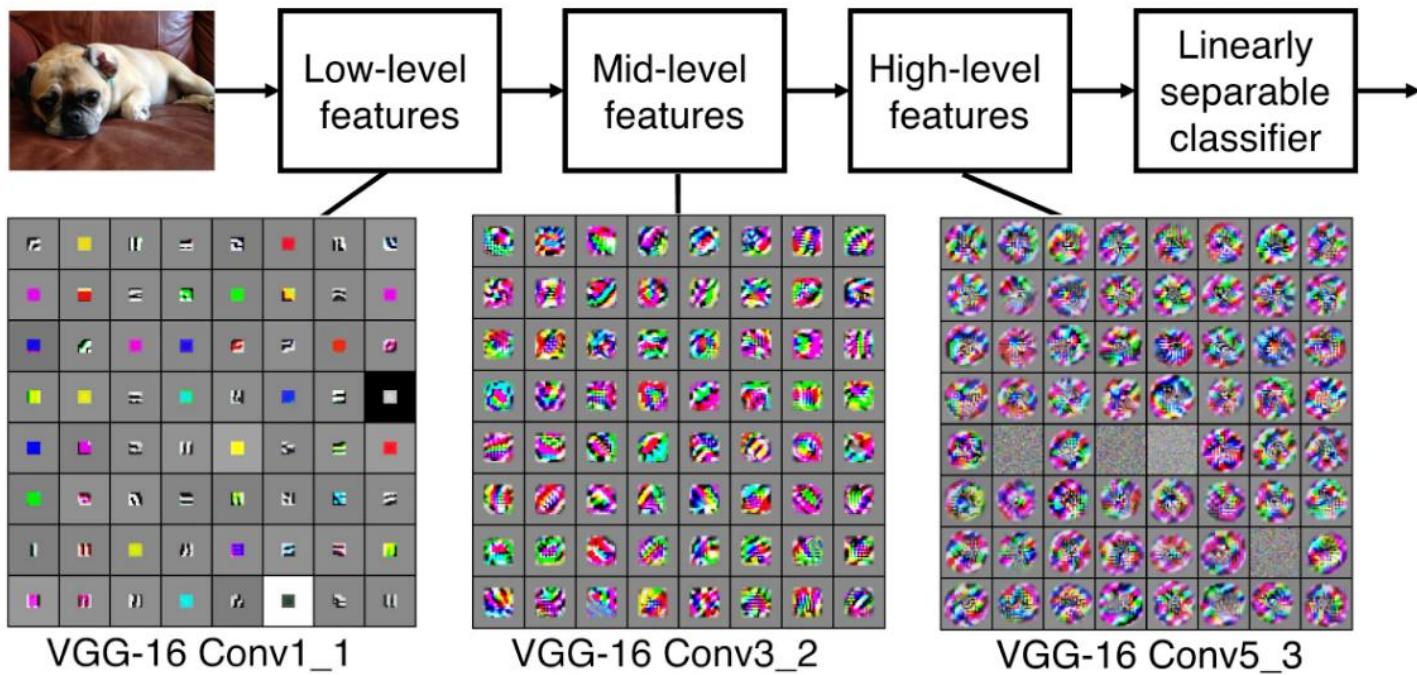
# CNN filters



## Preview

[Zeiler and Fergus 2013]

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



# Convolutional NN

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

\*

Horizontal

1	1	1
0	0	0
-1	-1	-1

=

0	0	0	0
30	10	-10	-30
30	10	-10	-30
0	0	0	0

Vertical

1	0	-1
1	0	-1
1	0	-1

$$(10,10,10)*(1,1,1)+(10,10,10)*(0,0,0)+(10,10,10)*(-1,-1,-1)=0 , \text{ element (1,1)}$$

Elementwise multiplication

Softmax function : Recognizing cats, dogs, and baby chicks



3



1



2



0



3



2



0



1

<https://www.youtube.com/watch?v=FmpDlaiMleA>

How Convolutional Neural Networks work by Brandon Rohrer

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

$$2*1 + 4*2 + 9*3 + 2*(-4) + 1*7 + 4*4 + 1*2 + 1*(-5) + 2*1 = 51$$

X                          Filter / Kernel                  =                  Feature

1	2	3
-4	7	4
2	-5	1

51		

One filter

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

$$4*1 + 9*2 + 1*3 + 1*(-4) + 4*7 + 4*4 + 1*2 + 2*(-5) + 9*1 = 66$$

X                          Filter / Kernel                  =                  Feature

1	2	3
-4	7	4
2	-5	1

51	66	

Input (image) has only one channel (grey scale), so one filter!

Image

$$\text{Feature size} = ((\text{Image size} - \text{Kernel size}) / \text{Stride}) + 1$$

$$\text{Feature size} = ((5 - 3) / 1) + 1 = 3$$

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

X

1	2	3
-4	7	4
2	-5	1

Filter / Kernel

=

51	66	20
31	49	101
15	53	-2

Feature

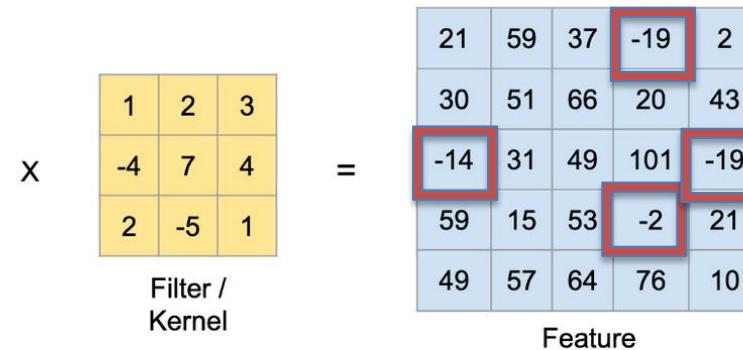
Image

$$\text{Feature size} = ((\text{Image size} + 2 * \text{Padding size} - \text{Kernel size}) / \text{Stride}) + 1$$

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

Image

$$\text{Feature size} = ((5 + 2 * 1 - 3) / 1) + 1 = 5$$



21	59	37	-19	2
30	51	66	20	43
-14	31	49	101	-19
59	15	53	-2	21
49	57	64	76	10

Convolved Feature

21	59	37	-19	2
30	51	66	20	43
-14	31	49	101	-19
59	15	53	-2	21
49	57	64	76	10

Convolved Feature

21	59	37	-19	2
30	51	66	20	43
-14	31	49	101	-19
59	15	53	-2	21
49	57	64	76	10

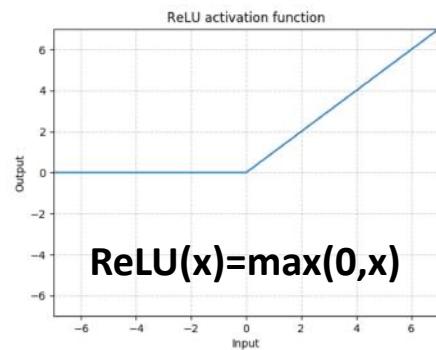
Feature

59	66		

Max Pooled Feature

59	66	66	43
51	66	101	101
59	53	101	101
59	64	76	76

Max Pooled Feature



$$\text{MaxPool}(\text{ReLU}(x)) = \text{ReLU}(\text{MaxPool}(x))$$

The procedure satisfies the communicative property and can be used either way. In practice ReLU activation function is applied right after a convolution layer and then that output is max pooled. Change negative numbers to zero, positive numbers stay the same

-?

0

# CNN Arithmetic, multiple input channels (RGB image)

Input : W=5, H=5, D=3 (Channel)

Filter : K=2 (number), F=3 x3 (size), S (Stride)=2, P (Zero Padding) = 1

Input Volume (+pad 1) (7x7x3)

$x[:, :, 0]$

0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	1	2	0	1	0	0
0	1	0	1	1	2	0
0	0	0	2	1	2	0
0	1	2	2	1	1	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

$w0[:, :, 0]$

0	-1	1
0	1	1
-1	0	1

$w0[:, :, 1]$

$w0[:, :, 2]$

$w0[:, :, 3]$

$b0[:, :, 0]$

$b0[:, :, 1]$

$b0[:, :, 2]$

Filter W1 (3x3x3)

$w1[:, :, 0]$

-1	-1	1
1	1	0
1	-1	1

$w1[:, :, 1]$

$w1[:, :, 2]$

$w1[:, :, 3]$

$b1[:, :, 0]$

$b1[:, :, 1]$

$b1[:, :, 2]$

Output Volume (3x3x2)

$o[:, :, 0]$

7	0	0
2	6	0
5	5	0
-1	1	-2
0	-5	-5
-4	-10	-9

$o[:, :, 1]$

$(W-F+S)/3 + P = \text{output volume size}$

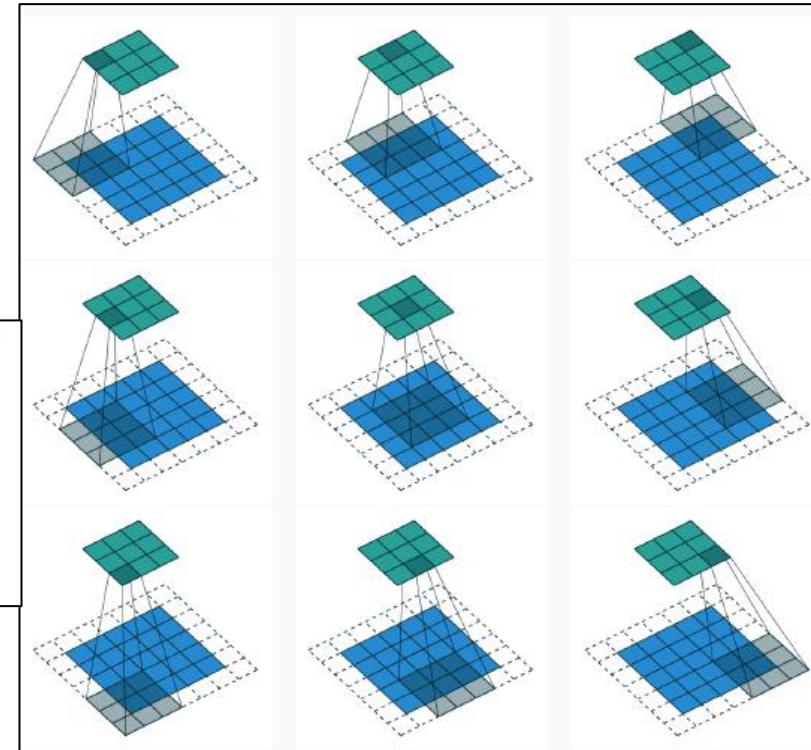
Output : W=3, H=3, D=2 (Channel)

Single depth slice

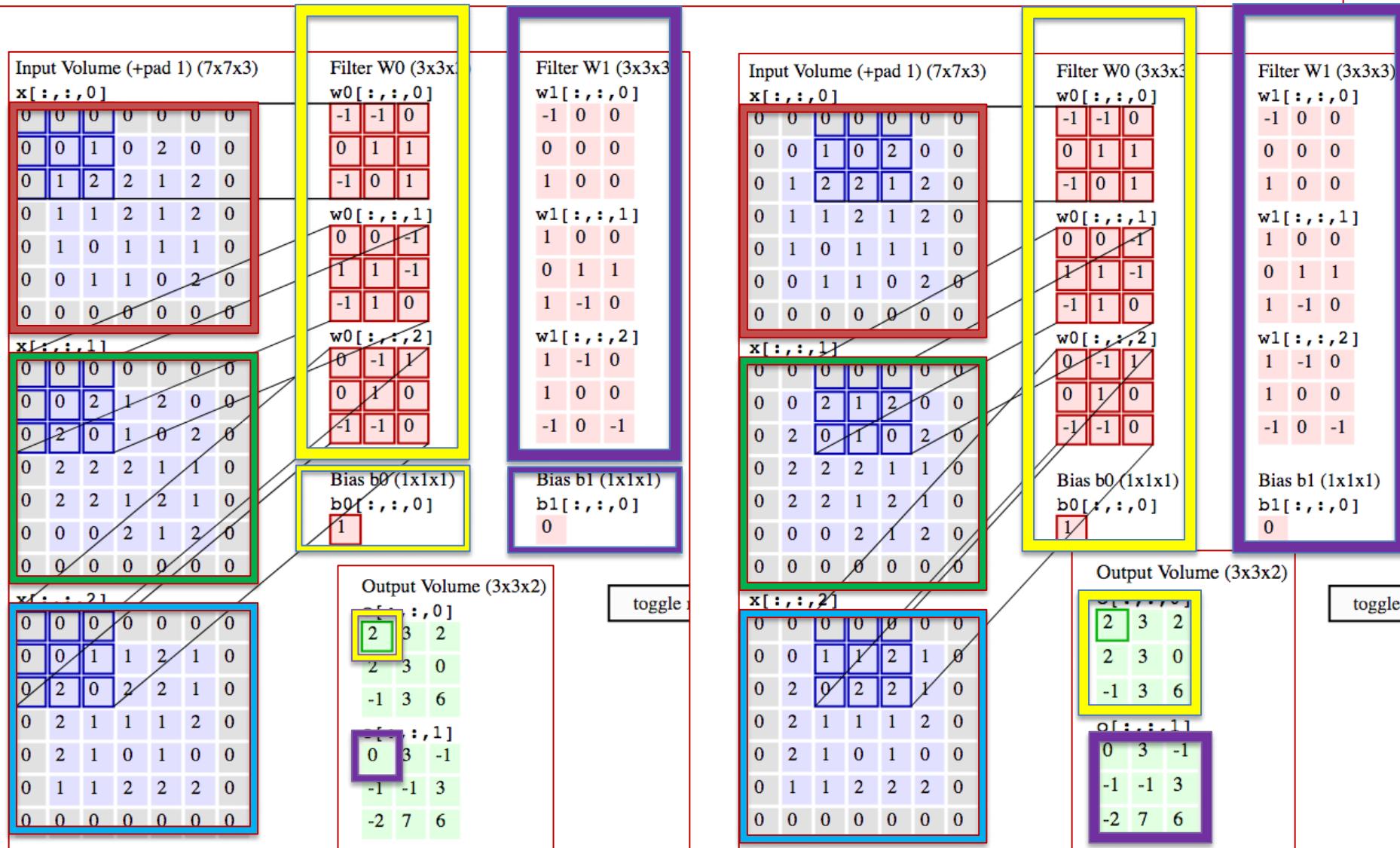
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4



Input volume :  $W1=5, H1=5, D1=3$ , CONV layer parameters =  $K=2, F=3, S=2, P=1$ .



$$(0 \times -1 + 0 \times -1 + 0 \times 0) + (0 \times 0 + 0 \times 1 + 1 \times 1) + (0 \times -1 + 1 \times 0 + 2 \times 1) = 3$$

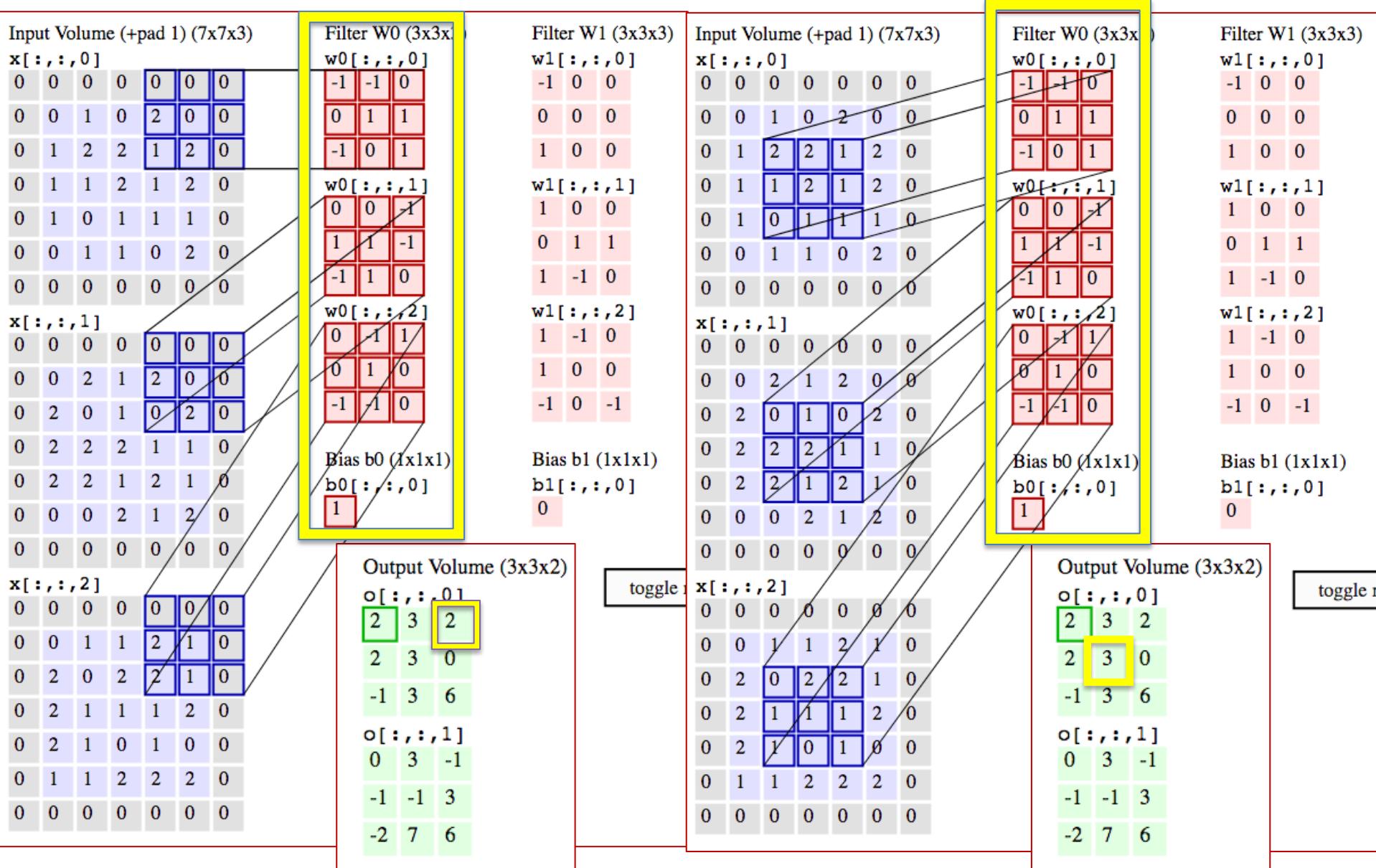
$$(0 \times 0 + 0 \times 0 + 0 \times -1) + (0 \times 1 + 0 \times 1 + 2 \times -1) + (0 \times -1 + 2 \times 1 + 0 \times 0) = 0$$

$$(0 \times 0 + 0 \times -1 + 0 \times -1) + (0 \times 0 + 0 \times 1 + 1 \times 0) + (0 \times -1 + 2 \times -1 + 0 \times 0) = -2$$

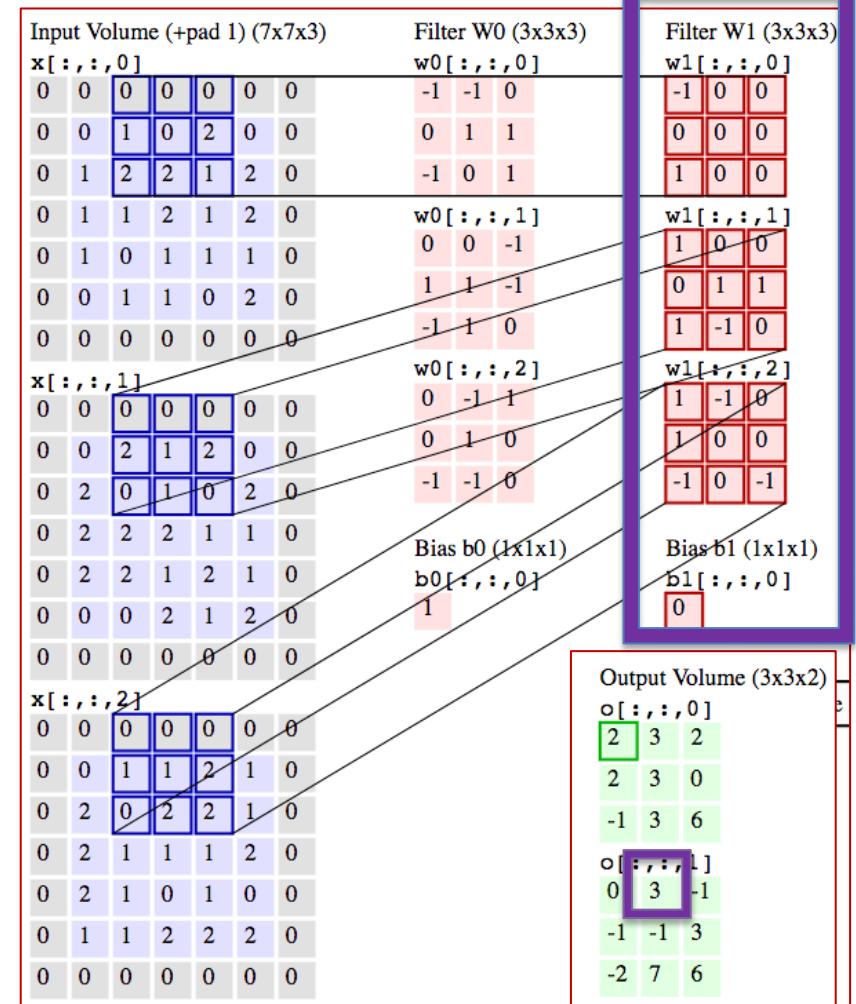
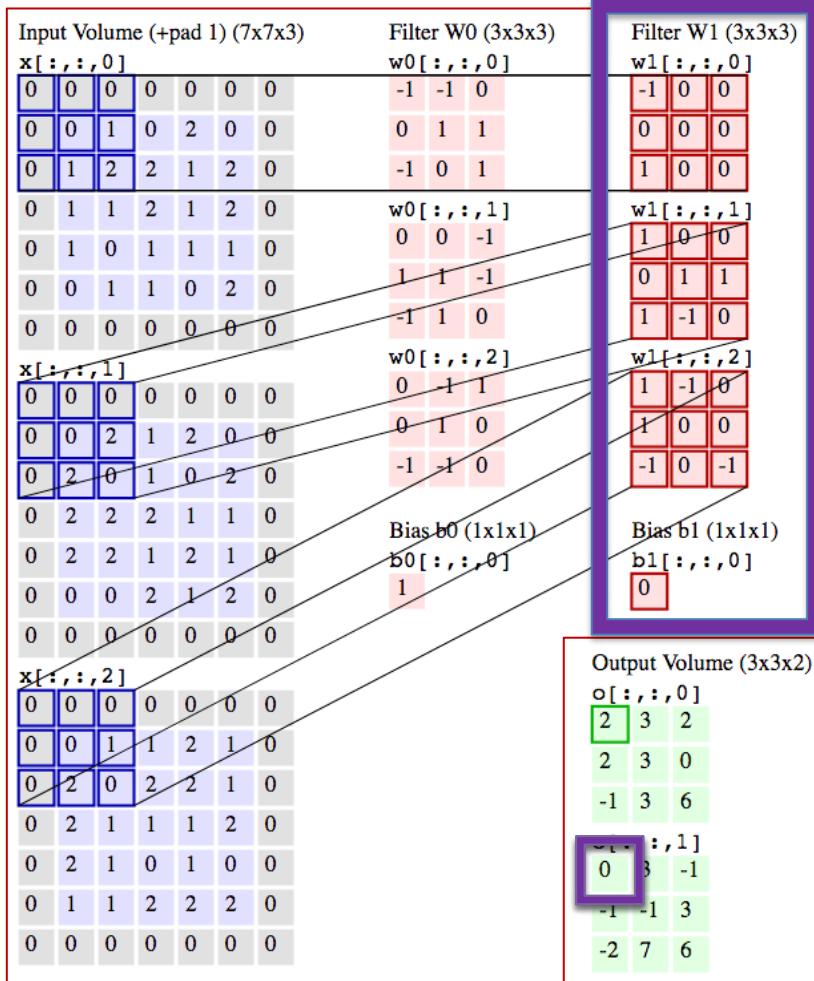
1 + b0 (1) =

2

# CNN Filter Operations - RGB



# CNN Filter Operations - RGB



$$(0 \times -1 + 0 \times 0 + 0 \times 0) + (0 \times 0 + 0 \times 0 + 1 \times 0) + (0 \times 1 + 1 \times 0 + 2 \times 0) = 0$$

$$(0 \times 1 + 0 \times 0 + 0 \times 0) + (0 \times 0 + 0 \times 1 + 2 \times 1) + (0 \times 1 + 2 \times -1 + 0 \times 0) = 0$$

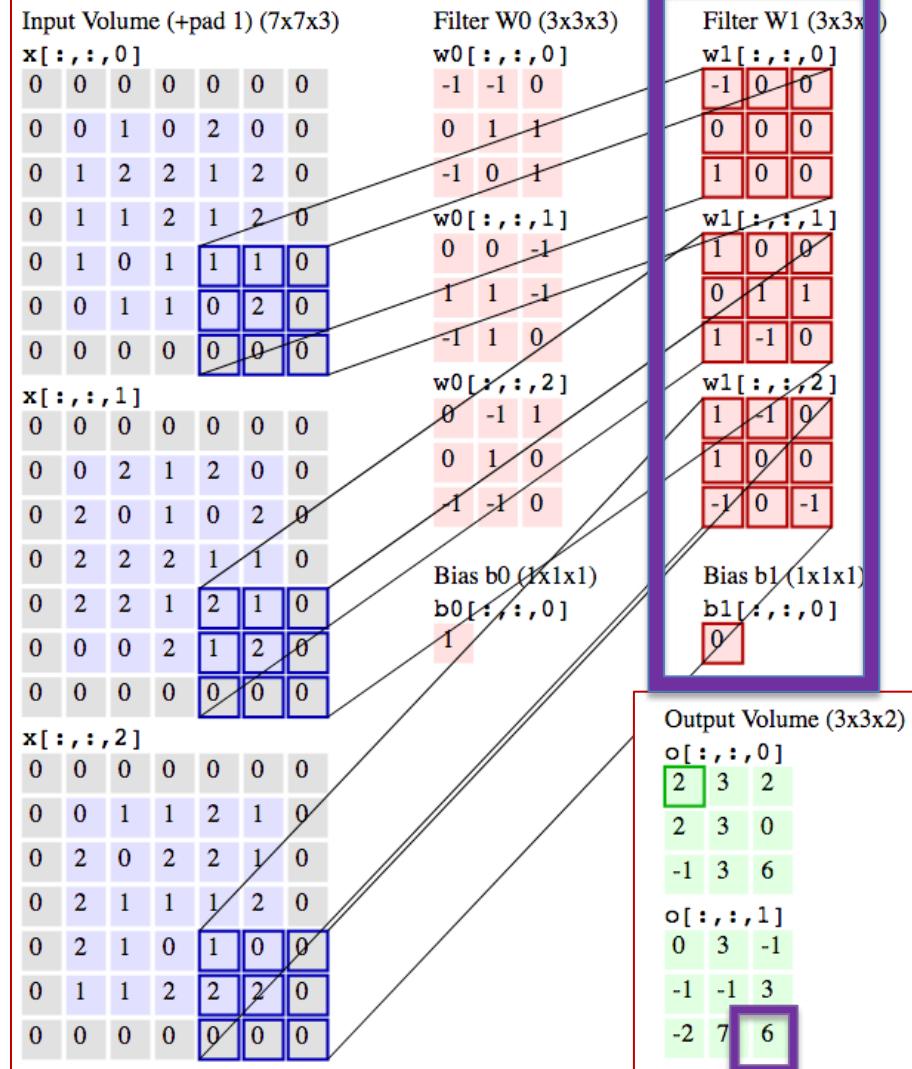
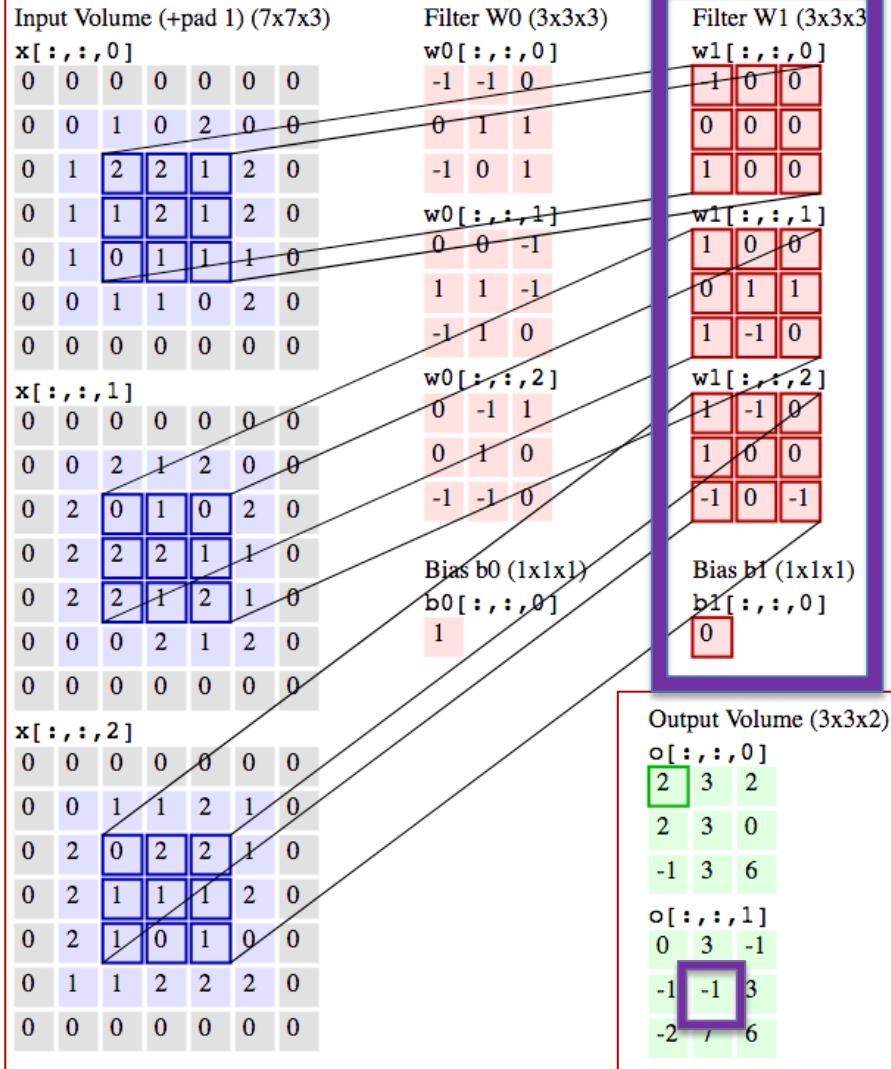
$$(0 \times 1 + 0 \times -1 + 0 \times 0) + (0 \times 1 + 0 \times 0 + 1 \times 0) + (0 \times -1 + 2 \times 0 + 0 \times -1) = 0$$

}

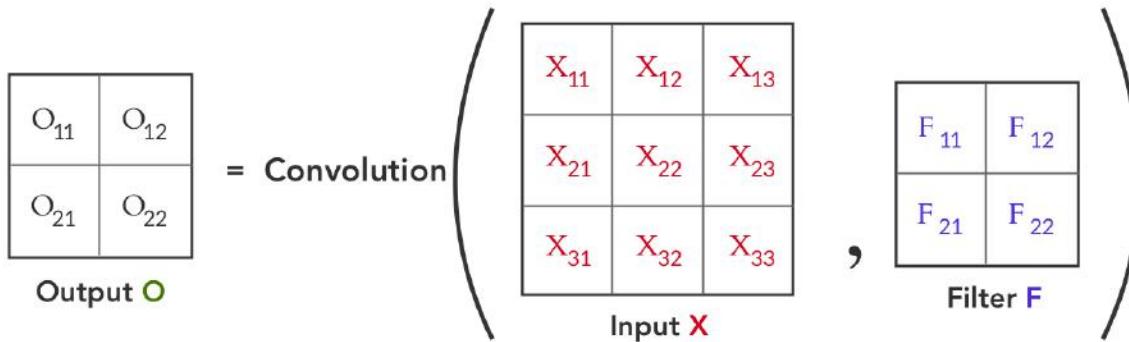
$$0 + b1(0) =$$

$$0$$

# CNN Filter Operations - RGB



# Forward path



X <sub>11</sub>	X <sub>12</sub>	X <sub>13</sub>
X <sub>21</sub>	X <sub>22</sub>	X <sub>23</sub>
X <sub>31</sub>	X <sub>32</sub>	X <sub>33</sub>

Input X



F <sub>11</sub>	F <sub>12</sub>
F <sub>21</sub>	F <sub>22</sub>

Filter F

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

X <sub>11</sub>	X <sub>12</sub>	X <sub>13</sub>
X <sub>21</sub>	X <sub>22</sub>	X <sub>23</sub>
X <sub>31</sub>	X <sub>32</sub>	X <sub>33</sub>

Input X

$\otimes$

F <sub>11</sub>	F <sub>12</sub>
F <sub>21</sub>	F <sub>22</sub>

Filter F

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

X <sub>11</sub>	X <sub>12</sub>	X <sub>13</sub>
X <sub>21</sub>	X <sub>22</sub>	X <sub>23</sub>
X <sub>31</sub>	X <sub>32</sub>	X <sub>33</sub>

Input X

$\otimes$

F <sub>11</sub>	F <sub>12</sub>
F <sub>21</sub>	F <sub>22</sub>

Filter F

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

$$O_{21} = X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22}$$

X <sub>11</sub>	X <sub>12</sub>	X <sub>13</sub>
X <sub>21</sub>	X <sub>22</sub>	X <sub>23</sub>
X <sub>31</sub>	X <sub>32</sub>	X <sub>33</sub>

Input X

$\otimes$

F <sub>11</sub>	F <sub>12</sub>
F <sub>21</sub>	F <sub>22</sub>

Filter F

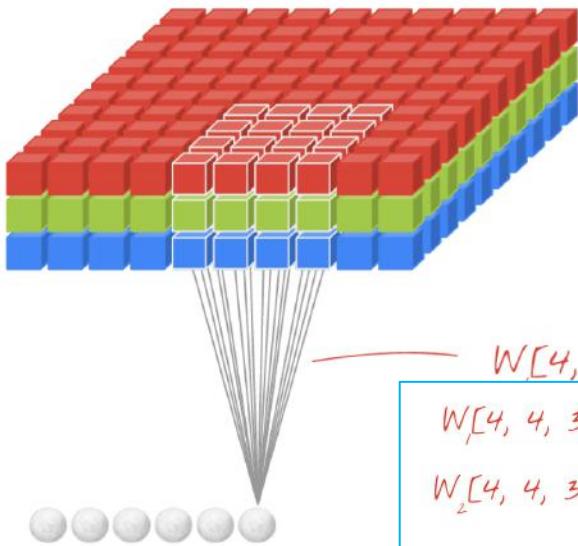
$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

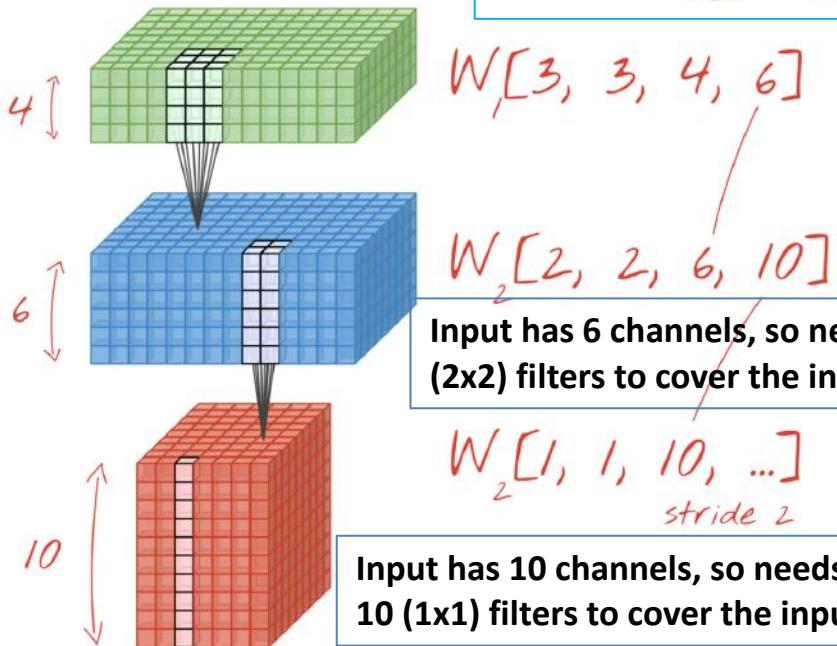
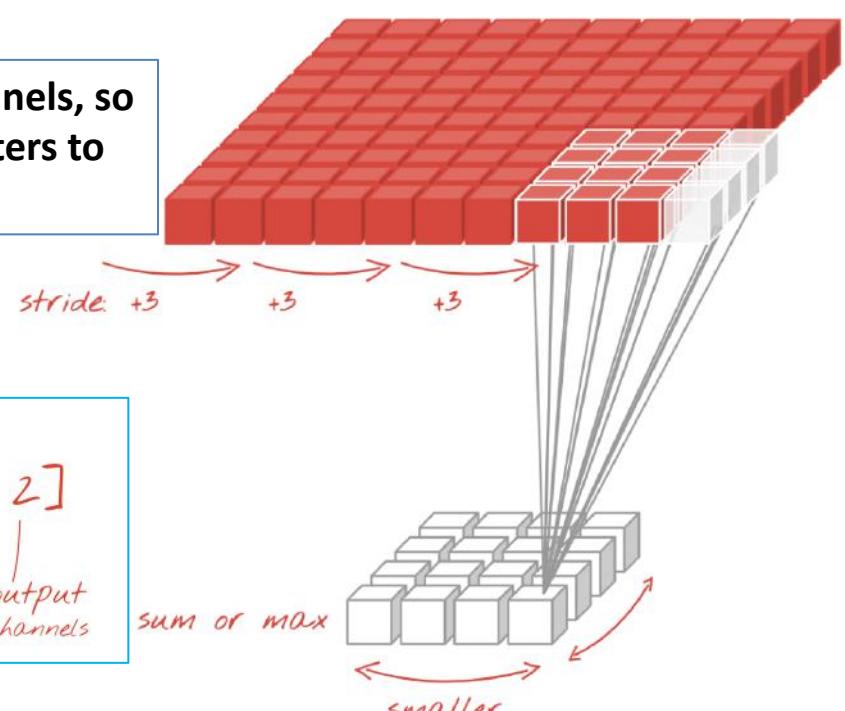
$$O_{21} = X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22}$$

$$O_{22} = X_{22}F_{11} + X_{23}F_{12} + X_{32}F_{21} + X_{33}F_{22}$$

# Multiple channels in inputs (outputs from previous convolution steps)



**Input has 3 channels, so needs 3 (4x4) filters to cover the input!**

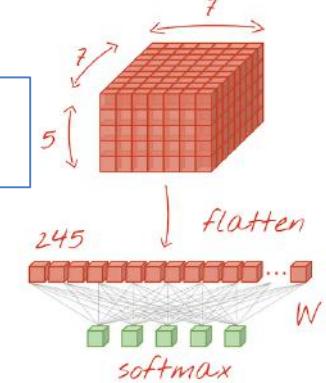


**Input has 6 channels, so needs 6 (2x2) filters to cover the input!**

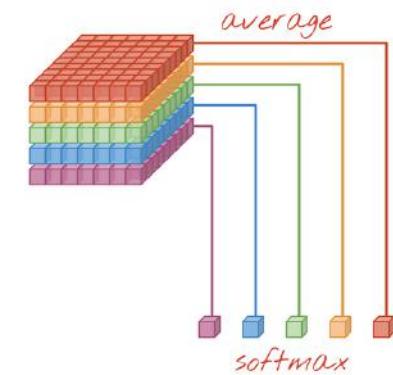
$W[2, 2, 6, 10]$   
stride 2

**Input has 10 channels, so needs 10 (1x1) filters to cover the input!**

Fully connected layer



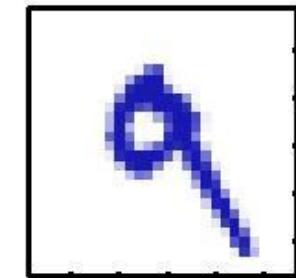
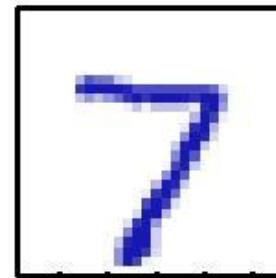
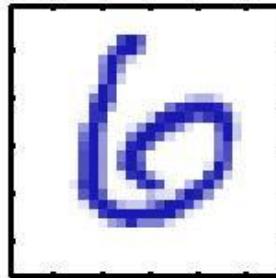
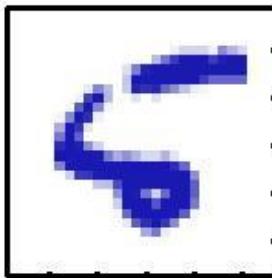
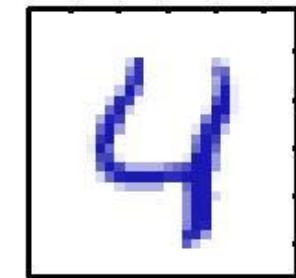
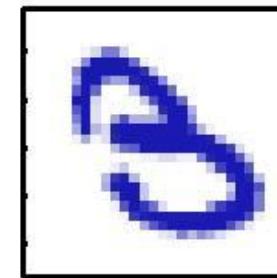
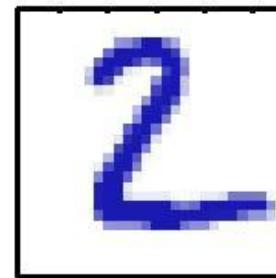
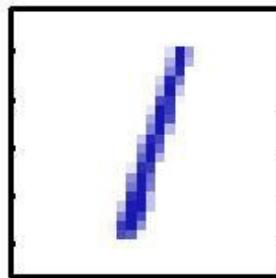
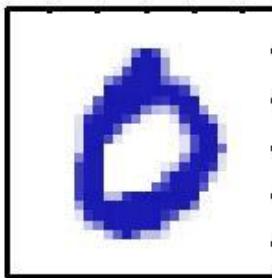
Global average pooling



# 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](#).<sup>[4][5]</sup> 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



# MNIST Example (28x28 pixels image)

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images, use the original image (28x28 matrix).

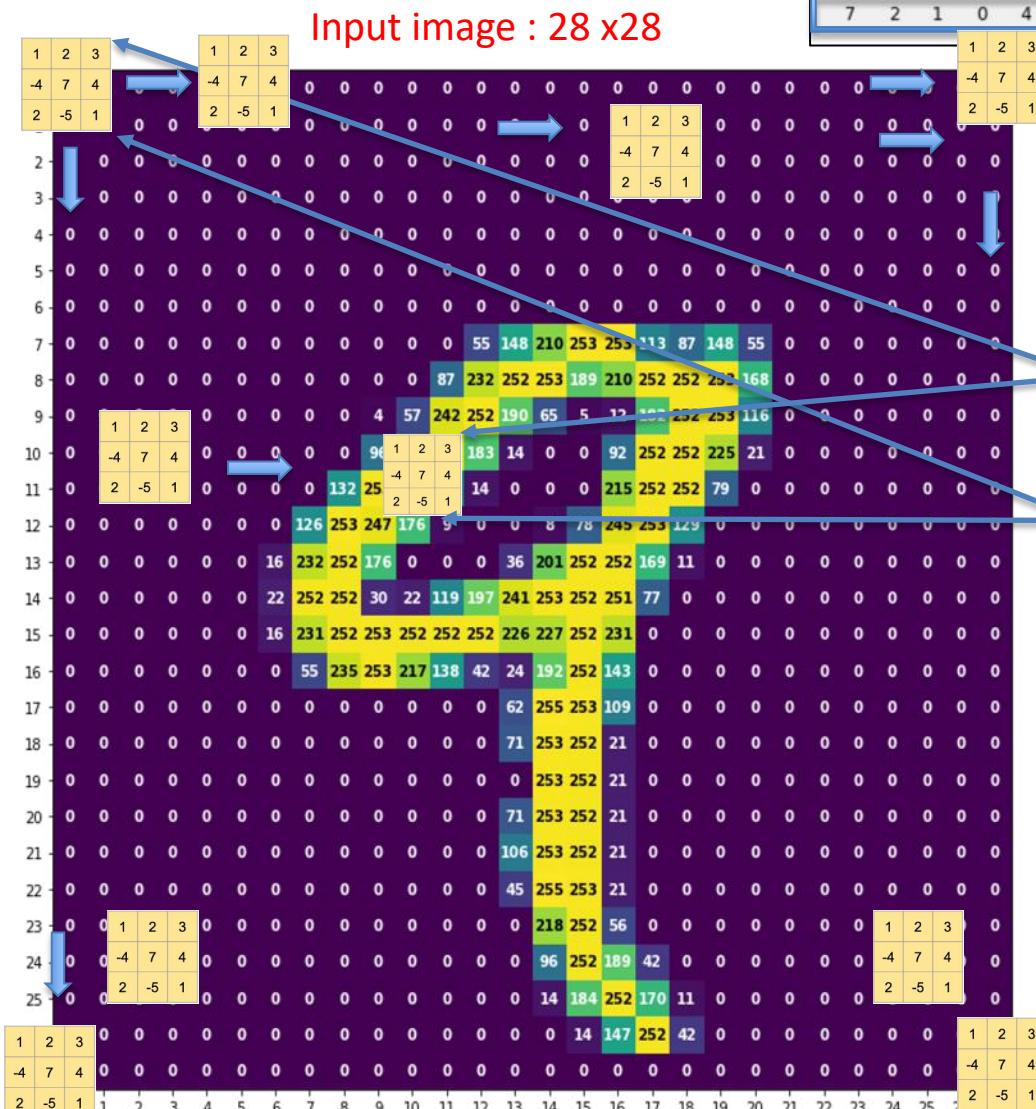


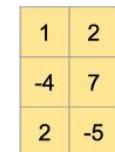
Image : input

training digits and their labels									
5	9	0	1	5	0	4	2	4	5
validation digits and their labels									
7	2	1	0	4	1	4	9	5	9

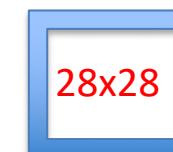
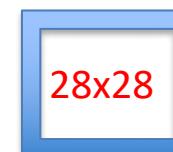
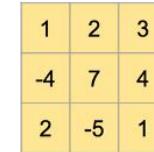
Labels : target

12 filters, each one acts on an image until it is fully covered.

3 x 3 filter (kernel)



12 3x3 filters  
Same padding



12 Output features : each 28 x 28 matrix

# Convolution NN : MNIST 28x28 Pixels

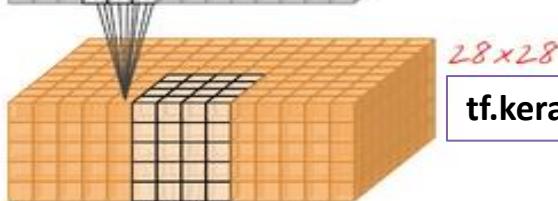
[FW, FH, C, H] = [ filter size ? X ?, input channel (Depth), output channel (no. of filter) ]

`tf.keras.layers.Reshape(input_shape=(28*28,), target_shape=(28, 28, 1))`



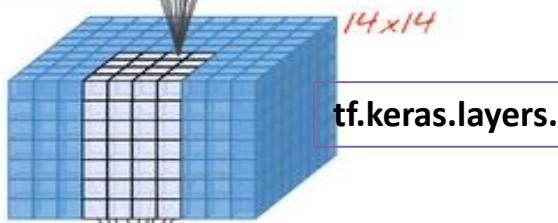
*Convolutional 3x3 filters=12  
W<sub>1</sub>[3, 3, 1, 12]*

`tf.keras.layers.Conv2D(kernel_size=3, filters=12, padding='same', activation='relu')`



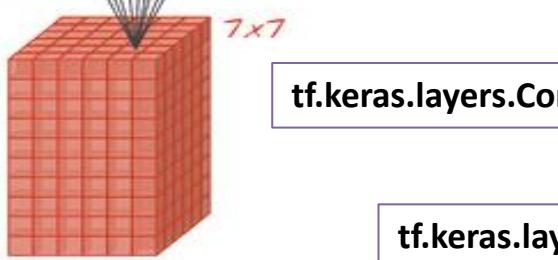
*Convolutional 6x6 filters=24  
W<sub>2</sub>[6, 6, 12, 24] stride 2*

`tf.keras.layers.Conv2D(kernel_size=6, filters=24, padding='same', activation='relu', strides=2)`



*Convolutional 6x6 filters=32  
W<sub>3</sub>[6, 6, 24, 32] stride 2*

`tf.keras.layers.Conv2D(kernel_size=6, filters=32, padding='same', activation='relu', strides=2)`



`tf.keras.layers.Flatten()`

*flatten*

*Dense layer*

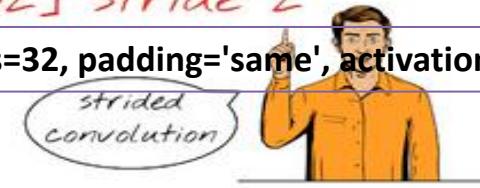
*W<sub>4</sub>[1568, 200]*

`tf.keras.layers.Dense(200, activation='relu')`

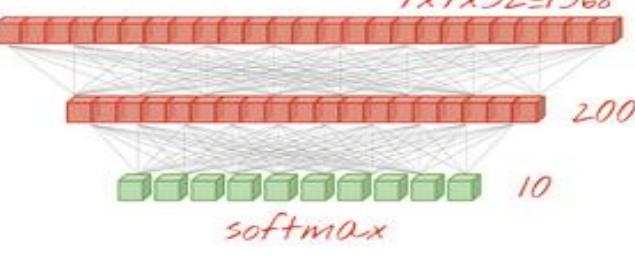
*Softmax dense layer*

*W<sub>5</sub>[200, 10]*

`tf.keras.layers.Dense(10, activation='softmax')`



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



# CNN : example – MNIST

There are three main types of layers in a CNN: Convolutional Layer, Pooling Layer, and Fully-Connected Layer.

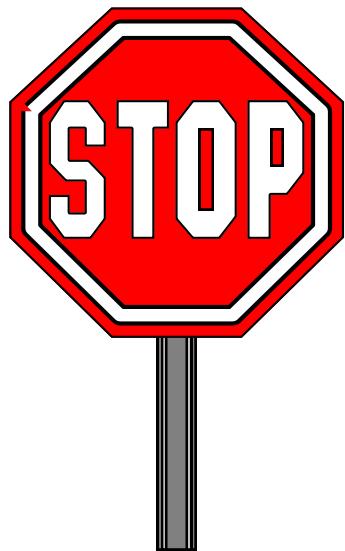
- ✓ INPUT [28x28x1] will hold the raw pixel values of the image, in this case an image of width 28, height 28 in grey scale (0-255)
- ✓ Apply twelve (12) 3x3 filters to one layer (channel) - W[3,3,Channel=1,number of filters=12]
- ✓ CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume of [28x28x12] when we use 12 filters. Volume size = [ 28 width x height = 28 , 12 channels (depth) ]
- ✓ RELU layer will apply an elementwise activation function, such as the  $\max(0,x)$
- ✓ Apply 24 6x6 filters to 12 layers (channel) with stride 2 - W[6,6,12,24] stride 2
- ✓ This may result in volume of [14x14x24] when we use 24 filters, stride 2.
- ✓ RELU layer will apply an elementwise activation function, such as the  $\max(0,x)$
- ✓ Apply 32 6x6 filters to 24 layers (channel) with stride 2 - W[6,6,24,32] stride 2
- ✓ This may result in volume of [7x7x32] when we use 32 filters, stride 2.
- ✓ RELU layer will apply an elementwise activation function, such as the  $\max(0,x)$
- ✓ Flatten the volume (tensor) to be a vector of  $7 \times 6 \times 32 = 1568$  elements
- ✓ Use 200 neurons with the input vector, Fully connected layer, W [1558, 200], output=1 x 200
- ✓ FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score.
- ✓ In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores.

# Acknowledgements and References

This portion of the tutorial contains many extracted materials from many online websites and courses. Listed below are the major sites. This portion of the materials is not intended for public distribution. Please visit the sites websites for detail contents. If you are beginner users of DNN, I suggest to read the following list of websites in its order.

- 1) <http://neuralnetworksanddeeplearning.com>
- 2) <https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist>
- 3) <https://www.deeplearning.ai/deep-learning-specialization/>
- 4) <http://cs231n.stanford.edu/>
- 5) MIT 6.S191, <https://www.youtube.com/watch?v=njKP3FqW3Sk>
- 6) <https://livebook.manning.com/book/deep-learning-with-python/about-this-book/>
- 7) <https://www.fast.ai/>
- 8) <https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/>
- 9) <https://www.nersc.gov/users/training/gpus-for-science/gpus-for-science-2020/>
- 10) <https://oneapi-src.github.io/oneDNN/>
- 11) <http://www.cs.cornell.edu/courses/cs4787/2020sp/>
- 12) More sites and free books are listed in [www.jics.utk.edu/actia](http://www.jics.utk.edu/actia) → ML
- 13) <https://machinelearningmastery.com>

# The End



- The End!

