Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)
MLP Example

Kwai Wong, Stan Tomov
Rocco Febbo, Julian Halloy

University of Tennessee, Knoxville

Feb. 27, 2021

Acknowledgements:

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, www.jics.utk.edu/lapenna, NSF award #202409
- www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,
- Autonomous vehicle, www.bitbucket.org/cfdl/opendnnwheel
- Parallel workflow framework, www.bitbucket.org/cfdl/opendiel
- magmaDNN, www.bitbucket.org/icl/magmadnn
- These projects come out from the RECSEM REU Summer program supported under NSF award #1659502

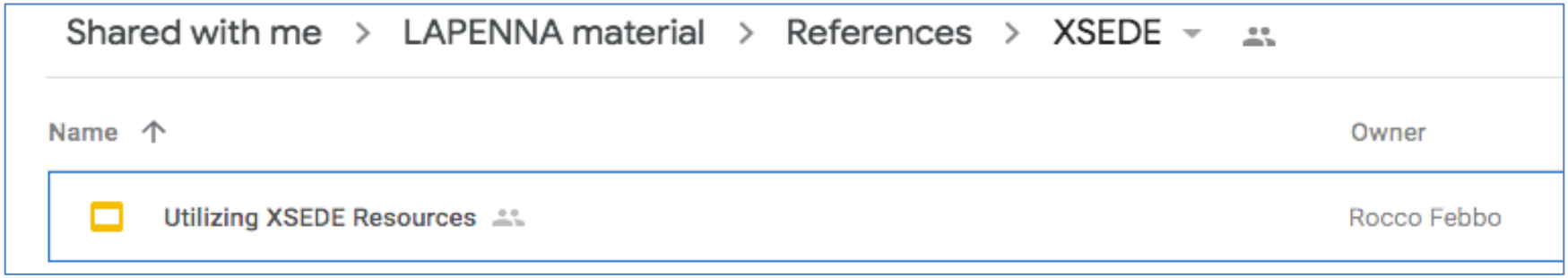
INNOVATIVE COMPUTING LABORATORY

Topics of Webinar Two

- ✓ **Introduction to LAPENNA Program**
- ✓ **Section 2 : Machine Learning and DNN**
  - ➢ **Overview**
  - ➢ **MNIST Test case , TensorFlow example**
  - ➢ **Neural Network Computation, Forward path**
  - ➢ **Example of Forward path calculation**
  - ➢ **Backward path**
  - ➢ **Fundamental Mathematical Theory**
  - ➢ **Optimization**
  - ➢ **Complete the cycle**

XSEDE Access

✓ **XSEDE user code, login, general operations**



Ensure you have an XSEDE account and have setup 2 factor authentication.

Obtain access to an SSH terminal. If you're on Windows you can use MobAXterm. (https://mobaxterm.mobatek.net/)

Type:

ssh login.xsede.org

Logging In

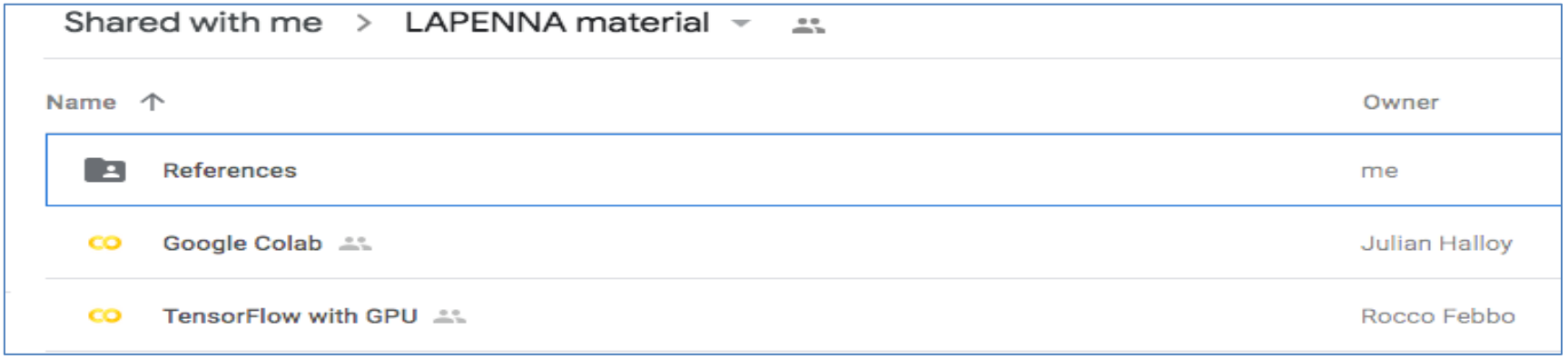
Enter your password and select 2 factor authentication option. (either phone Duo app)
Once logged in, you will be greeted by the Single Sign on Hub message.

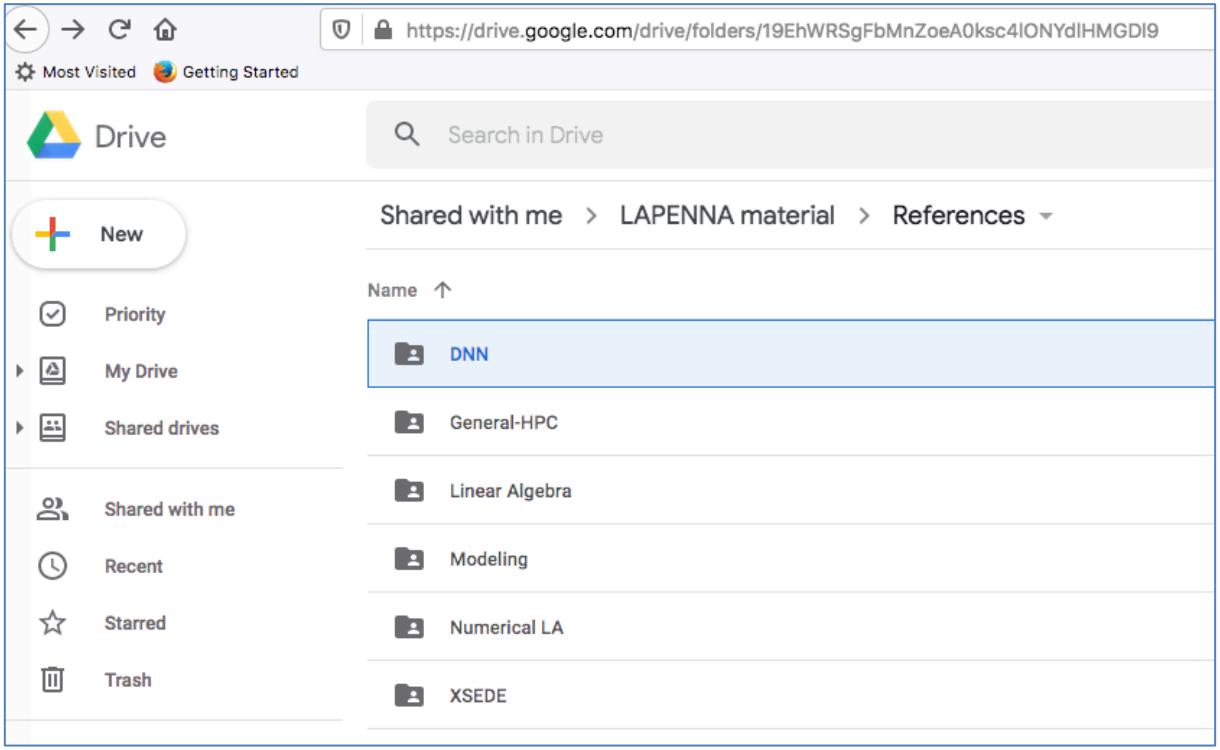
$> gsissh bridges

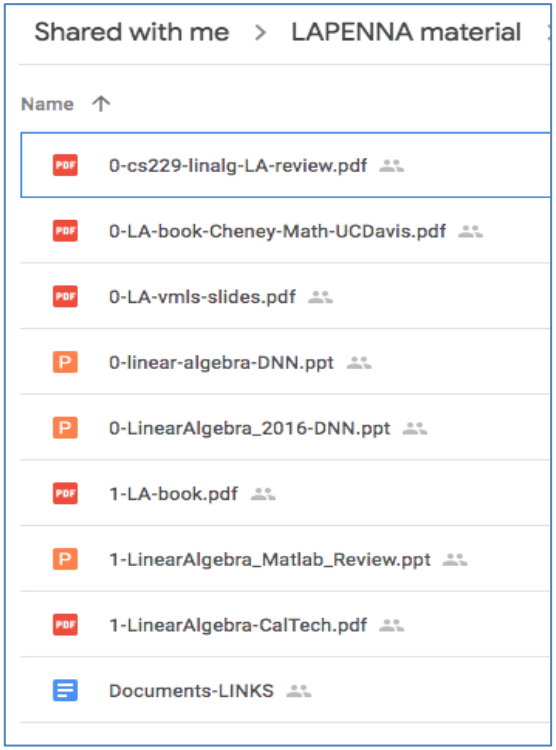
XSEDE Access : need to sign in to www.xsede.org

Google Drive, Shared Space

✓ **Google drive share space, google colab, colab.research.google.com**







Google Colab, jupyter notebook

" To learn data science is to do data science"

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

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

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

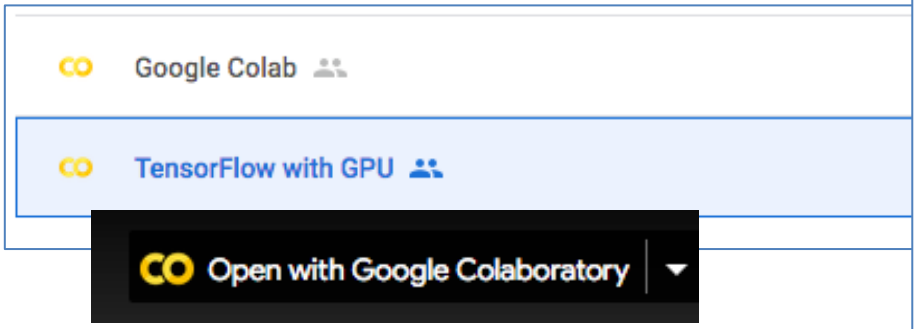
Use this link to run R code on Google COLAB
https://colab.research.google.com/notebook#create=true&language=r

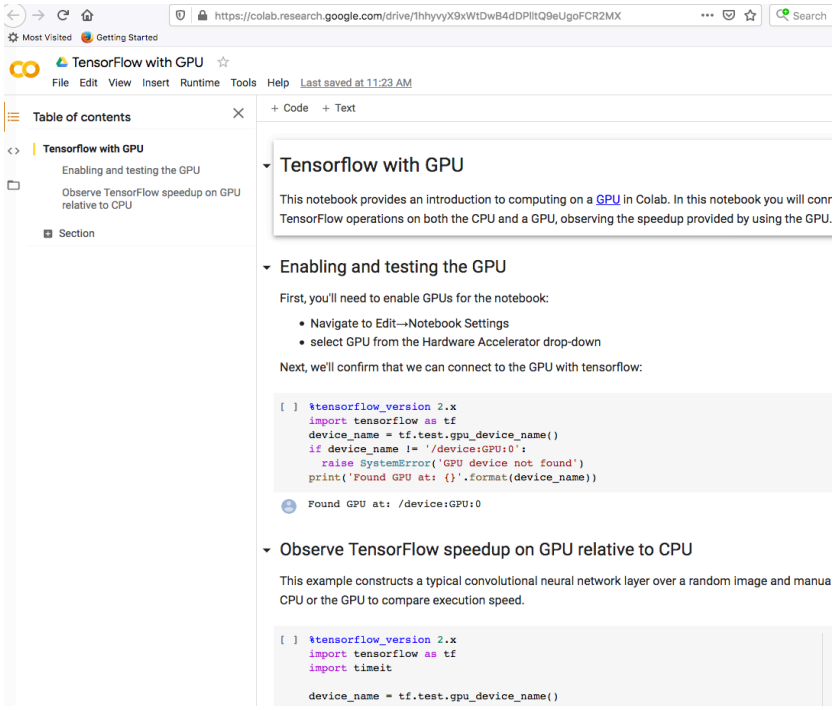
Google Colab, jupyter notebook

Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

✓ **colab.research.google.com,**






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

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

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

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

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

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

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

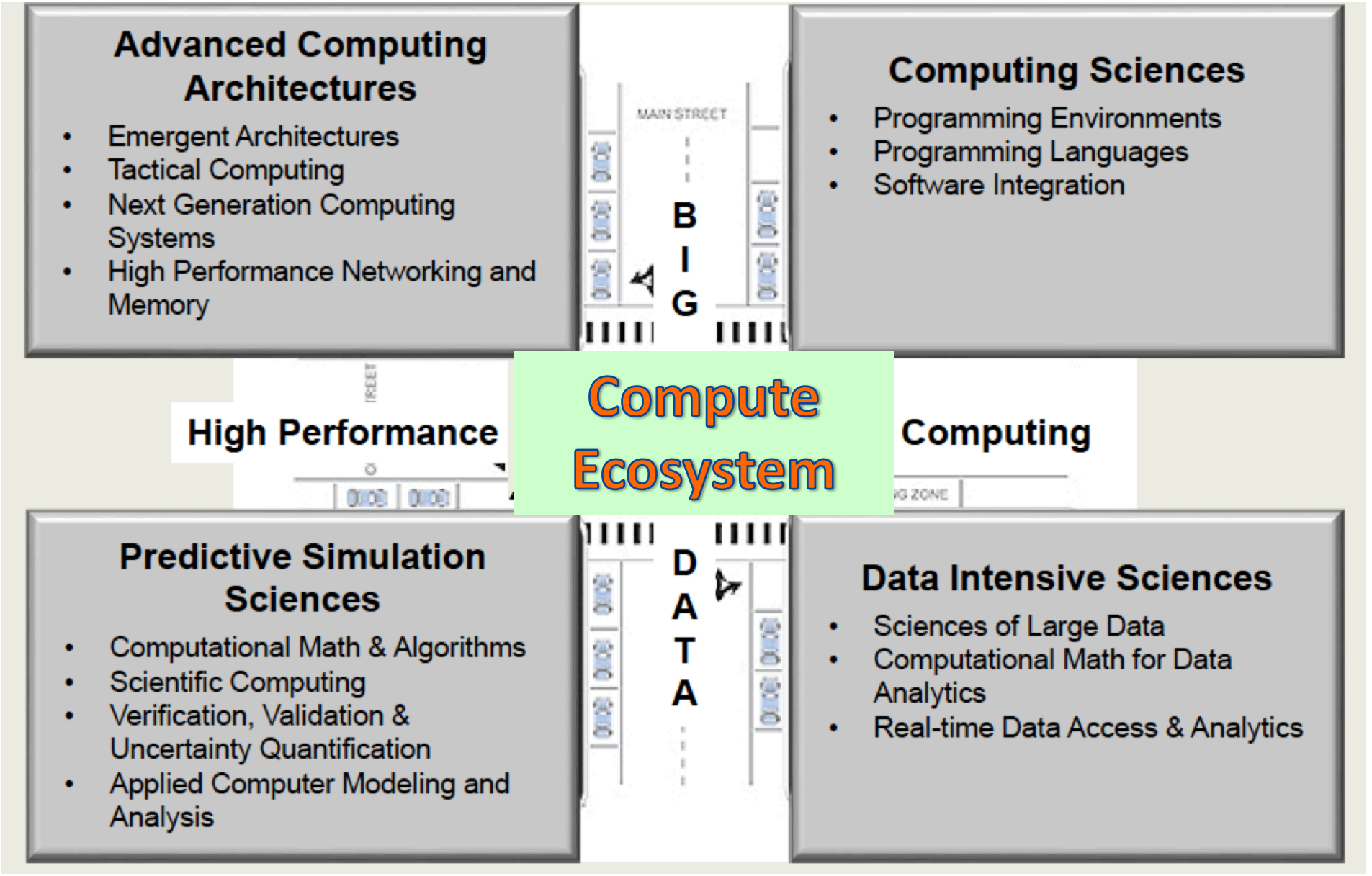
✓ **COLAB Introduction http://www.youtube.com/watch?v=vVe648dJOdI**

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

List of references: Linear Algebra

- ✓ https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/
- ✓ Deep Learning UC Berkeley STAT-157 2019, https://www.youtube.com/watch?v=Va8WWRfw7Og&list=PLZSO_6-bSqHQHBCoGaObUljoXAyyqhpFW
- ✓ **Linear Algebra – Gilbert Strang:** https://www.youtube.com/watch?v=dtX8iQGQQkI&list=PLZSO_6-bSqHQHBCoGaObUljoXAyyqhpFW&index=4
- ✓ http://www-math.mit.edu/~gs/ **, video course**
- ✓ **Matrix Applied Linear Algebra-Jim Demmel** https://people.eecs.berkeley.edu/~demmel/ma221_Spr20/Lectures/index.html
- ✓ https://www.3blue1brown.com/
- ✓ https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab
- ✓ https://www.math.ucdavis.edu/~linear/linear-guest.pdf
- ✓ https://comp-think.github.io/
- ✓ https://edu.google.com/resources/programs/exploring-computational-thinking/
- ✓ **https://www.youtube.com/watch?v=Nc-V948dXWI**
- ✓ **Google drive → Linear Algebra**
- ✓ **Many more ….**

Four Tiers – Computational Ecosystem



Menu

Modeling Cycle and Emergent Technology

Big Data → Deep Learning → Computing capacity
Combining the Learned the pattern and behavior hidden in sensor or computed data to predict complicated phenomena

A growing field with evolving technology
Data- + model-based simulation --> Supercomputer

Fast enough to get approximate results (time)
Formulation, Algorithm, linear algebra

Enough memory to process the huge amount of data (length scale)
Length, size, reduction, representation, projection

Exercises

1. A desktop computer has the following specification: quad-core, 24GB RAM, 2.5 GHz, and each core has 8 floating units (perform 8 DP operations in one clock cycle). What is the theoretical double precision peak performance of the desktop in FLOPS?

2. Given A is a M x K matrix, B is a K x N matrix, what is the number of floating point operation (FLOP) of A x B?

3. Given M=K=N for A and B, what is the maximum dimension N that the computer can hold?

4. Based on the answer above, what is the time needed to compute C=A x B if 90% of the theoretical rate of the desktop can be attained for the BLAS3 operation?

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

- Core : computing unit : processor
- Dual core machine (Intel or AMD CPU) : a CPU with 2 cores, each core is a 2.4 GHz computing unit with 2GB of RAM (memory in the processor not disk space)
- Binary bits (b) : "0" or "1" , 1 Byte (B) = 8 bits
- Binary number : 11111111= $(2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^{0)} = (2^8 - 1) = 255$!!
- 32 bits machine or operating system => largest integer (all positive) = $(2^{32} - 1)$ = (4,294,967,296 -1) or range of integer = $-(2^{31})$ to $(2^{31} - 1)$
- 64 bits machine or operating system => range of integer = $-(2^{63})$ to $(2^{63} - 1)$
- Kilo (K) = 10^3 (or 2^{10}) ; Mega (M) = 10^6 (or 2^{20}); Giga (G) = 10^9 (or 2^{30}); Tera (T billion) = 10^{12} (or 2^{40}) ; Peta (P) = 10^{15} (or 2^{50})
- GB = 1000^3 bytes, GiB = 1000^3 Bytes
- FLoating Point Operation (+, -, / , *) : (10.1 + 0.1) * 1.0 / 2.0 = 5.1 => 3 FLOP
- FLOPS = FLOP per second :: 1 PetaFLOPS (kraken) = 10^{15} FLOP in one second
- FLOPS in a core = (clock rate) x (floating point operation in one clock cycle)
- Peak Rate = (FLOPS in one compute unit, core) x (no. of core)

1. A desktop computer named MA6633DC has the following specification: quad-core, 24GB RAM, 2.5 GHz, and each core has 8 floating units (perform 8 DP operations in one clock cycle). What is the theoretical double precision peak performance of the desktop in FLOPS?

FLOPS in one core= (clock rate) x (floating point operation in one clock cycle)
Peak Rate = (FLOPS in one core) x (no. of core)
Peak FLOPS = 2.5 GHz * 8 * 4 cores = 80 GFLOPS

2. Given A is a M x K matrix, B is a K x N matrix, what is the number of floating point operation (FLOP) of A x B?

C=A x B ; A : A(M , K) ; B : B(K, N) , C = C(M, N) ; one element of C need (2K-1) DP
FLOP = (2K-1) x M x N

3. Given M=K=N for A and B, what is the maximum dimension N that MA6633DC can hold?

For double precision float, we need 8 bytes to store each element.
If M=N=K=N, in total we need 3N^2 elements
3N^2 * 8 = 24 GB, thus **N = floor(sqrt(1e9)) = 31622**

4. Based on the answer above, what is the time needed to compute C=A x B if 90% of the theoretical rate of MA6633DC can be attained for the BLAS3 operation?

90% * 80 GFLOPS = 72 GFLOPS
The FLOP we need to compute A x B is ~2N^3 ; **Time = 2 (31622)^3 FLOP / 72 GFLOPS = 878 seconds**

Question #1

a) A supercomputer is performing a HPL test for submission to the top500 list. The computer has 5000 compute units. Each compute unit has 20 cores, 64 GB RAM, 2.5 GHz, and each core has 8 floating units (performing 8 DP operations in one clock cycle). What is the theoretical double precision peak performance of the supercomputer in TFLOPS?

b) (Assuming solving Ax=b requires 2N^3/3 FLOP, where A is a NxN matrix, what is the maximum size of A (N) that the computer can solve? (only need to store A)

c) Using the N in b), what is the time needed to solve Ax=b based on the peak rate obtained in a)?

Question #1

a) A supercomputer is performing a HPL test for submission to the top500 list. The computer has 5000 compute units. Each compute unit has 20 cores, 64 GB RAM, 2.5 GHz, and each core has 8 floating units (performing 8 DP operations in one clock cycle). What is the theoretical double precision peak performance of the supercomputer in TFLOPS?

b) (Assuming solving Ax=b requires 2N^3/3 FLOP, where A is a NxN matrix, what is the maximum size of A (N) that the computer can solve? (only need to store A)

c) Using the N in b), what is the time needed to solve Ax=b based on the peak rate obtained in a)?

JICS Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

Linear Algebra

David Cherney, Tom Denton,
Rohit Thomas and Andrew Waldron

Introduction to Linear Algebra

Mark Goldman

Emily Mackevicius

Lecture slides for

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

Stephen Boyd Lieven Vandenberghe

Caltech Division of the Humanities and Social Sciences

Quick Review of Matrix and Real Linear Algebra

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

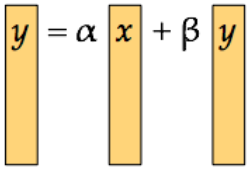
Linear Algebra Review and Reference

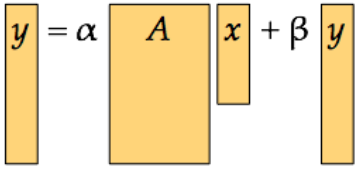
Zico Kolter (updated by Chuong Do)

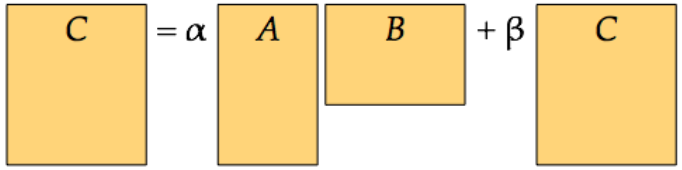
September 30, 2015

BLAS: Basic Linear Algebra Subroutines

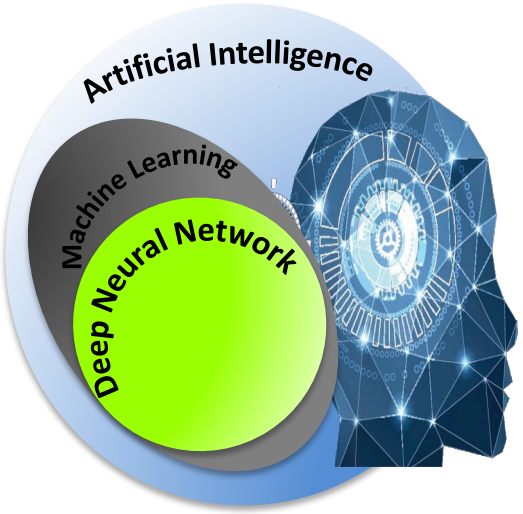
- ✓ Level 1 BLAS — vector operations
  - ✓ $O(n)$ data and flops (floating point operations)
  - ✓ Memory bound: $O(1)$ flops per memory access
- ✓ Level 2 BLAS — matrix-vector operations
  - ✓ $O(n^2)$ data and flops
  - ✓ Memory bound: $O(1)$ flops per memory access
- ✓ Level 3 BLAS — matrix-matrix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

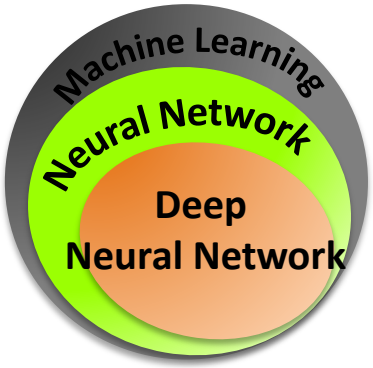


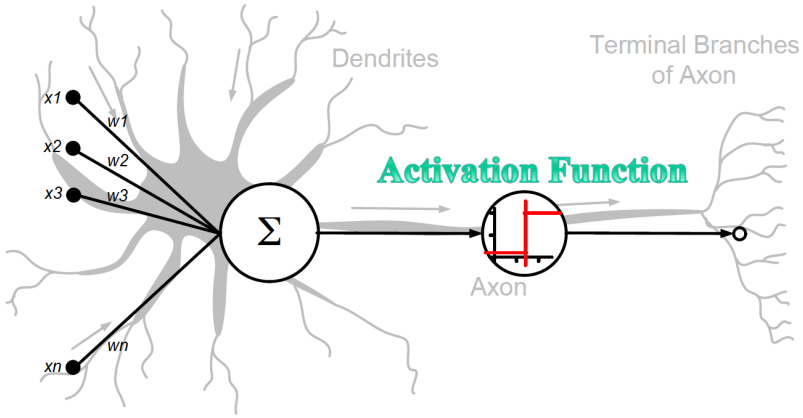




AI, ML, NN, DNN





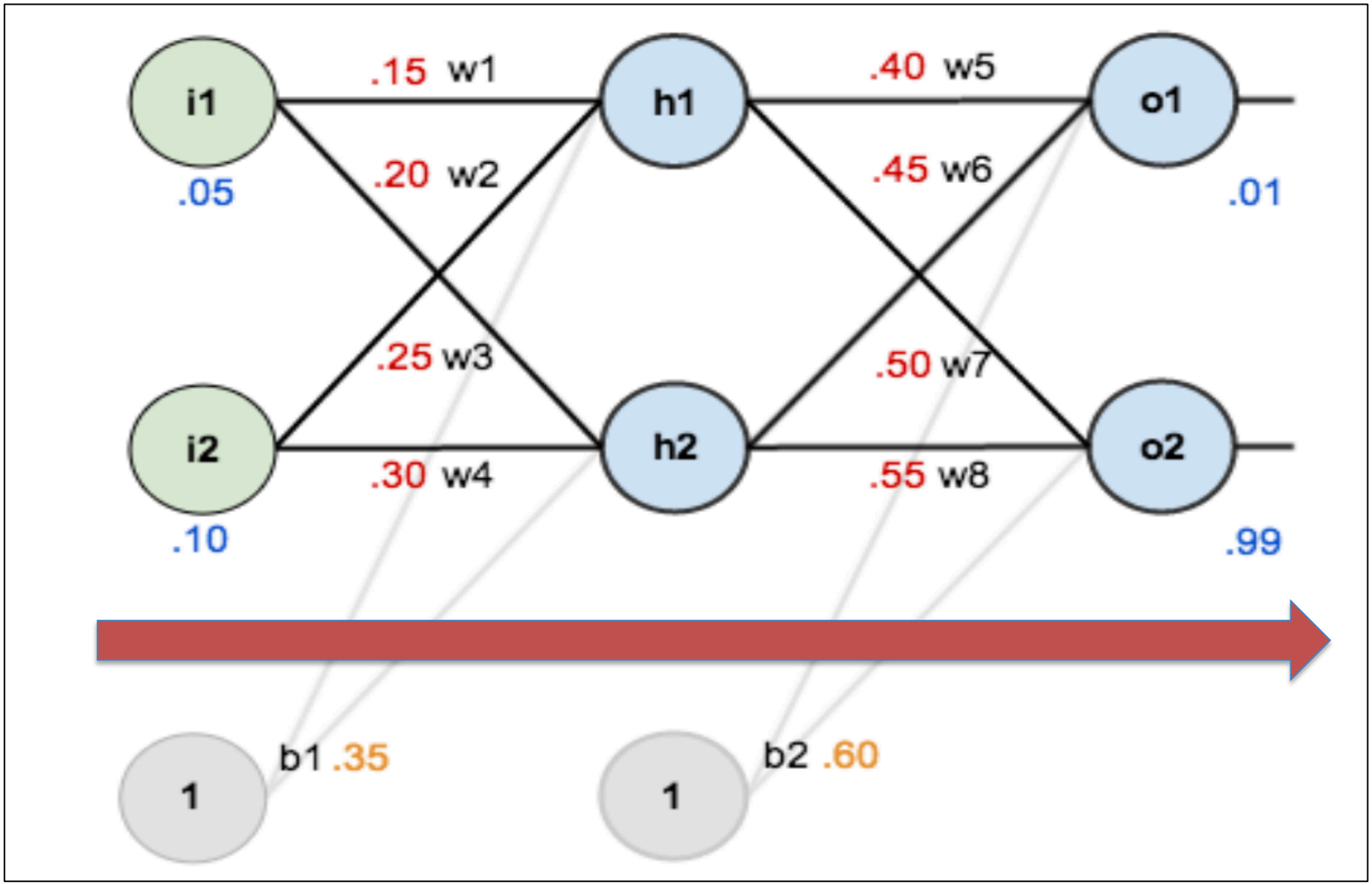


- ✓ **Artificial Intelligence (AI)**: science and engineering of making intelligent machines to perform the human tasks (John McCarthy,1956). AI applications is ubiquitous.
- ✓ **Machine learning (ML) :** A field of study that gives computers the ability to learn without being explicitly programmed (Arthur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

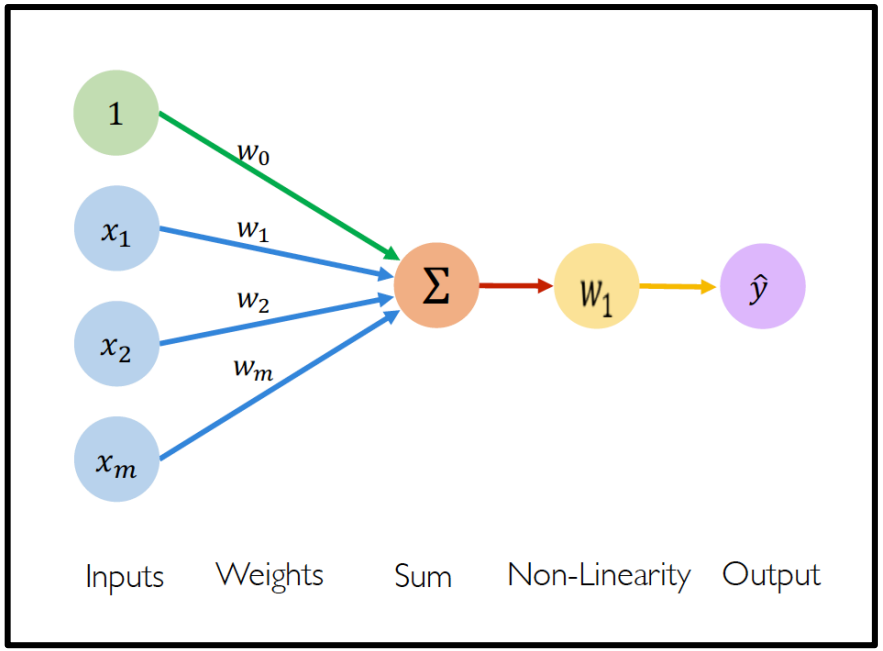
Topics of Webinar Two

- ✓ **Section 2 : Machine Learning and DNN**
  - ➢ **Neural Network Computation, Forward path**
  - ➢ **Example of Forward path calculation**

Multilayer Perceptron : Forward Path Calculation



DNN MLP Forward Steps



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

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

Output

Linear combination of inputs

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

Non-linear activation function

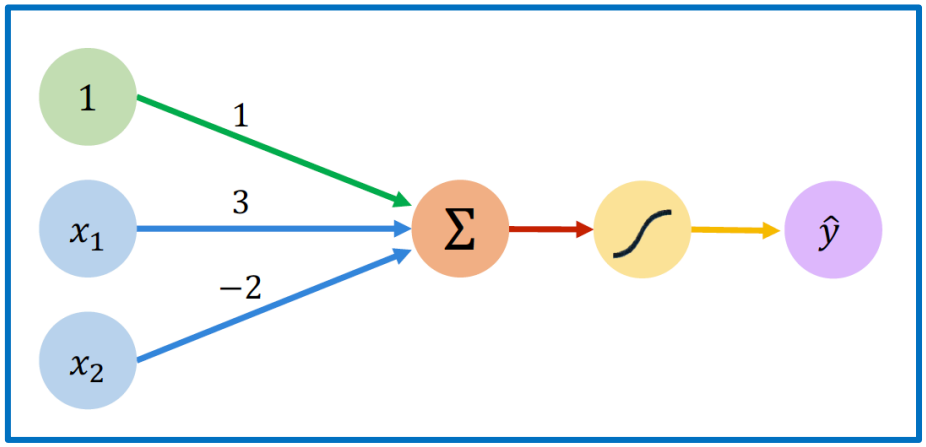
Bias

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

- Example: sigmoid function

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

Activation Function Example



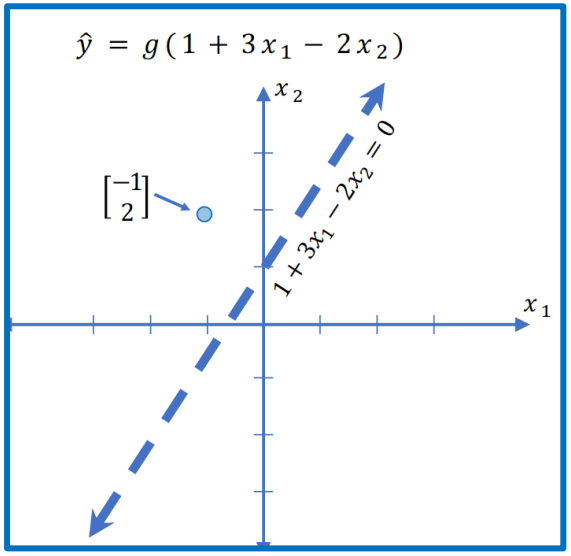
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

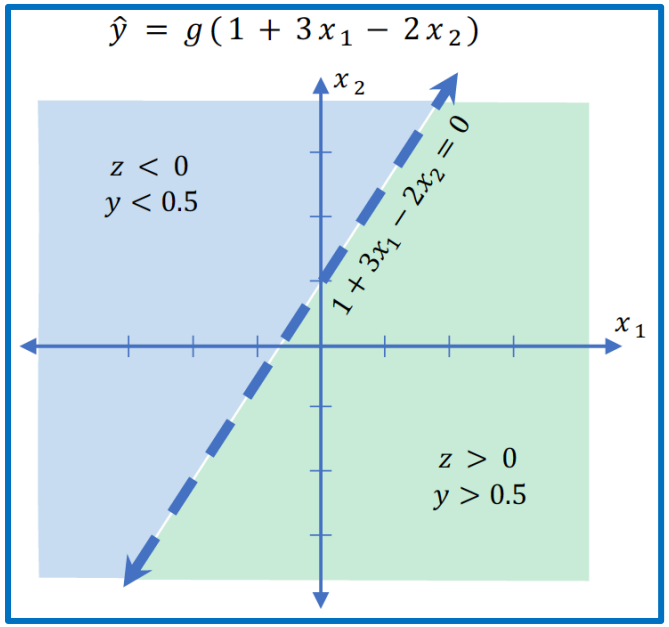
$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

This is just a line in 2D!

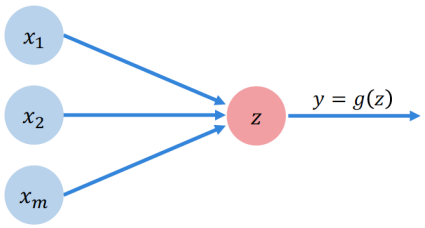
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$

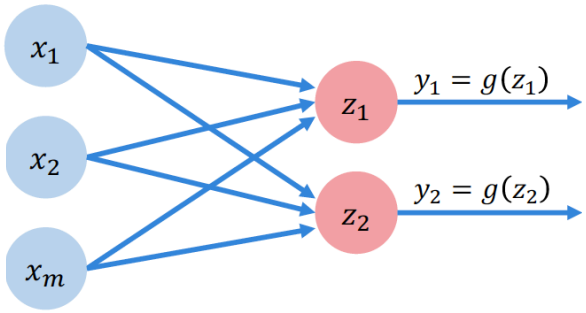




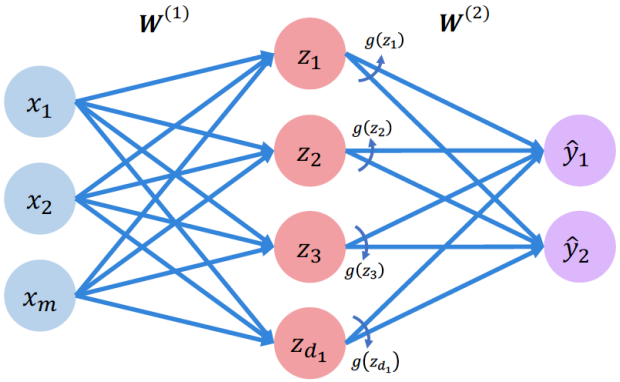
Multilayer Perceptron : Forward Path Calculation



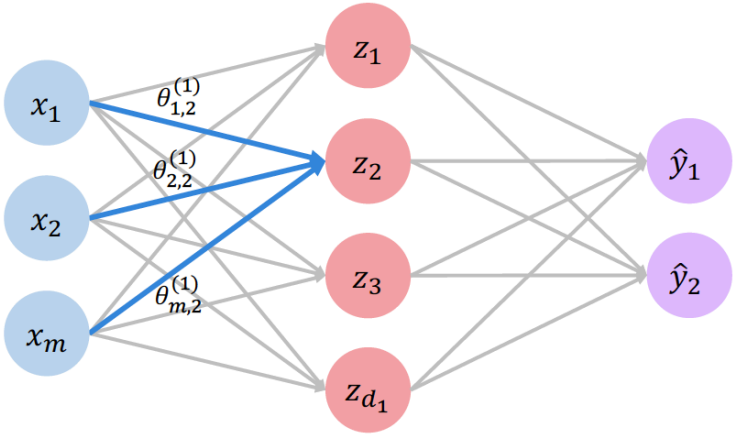
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$



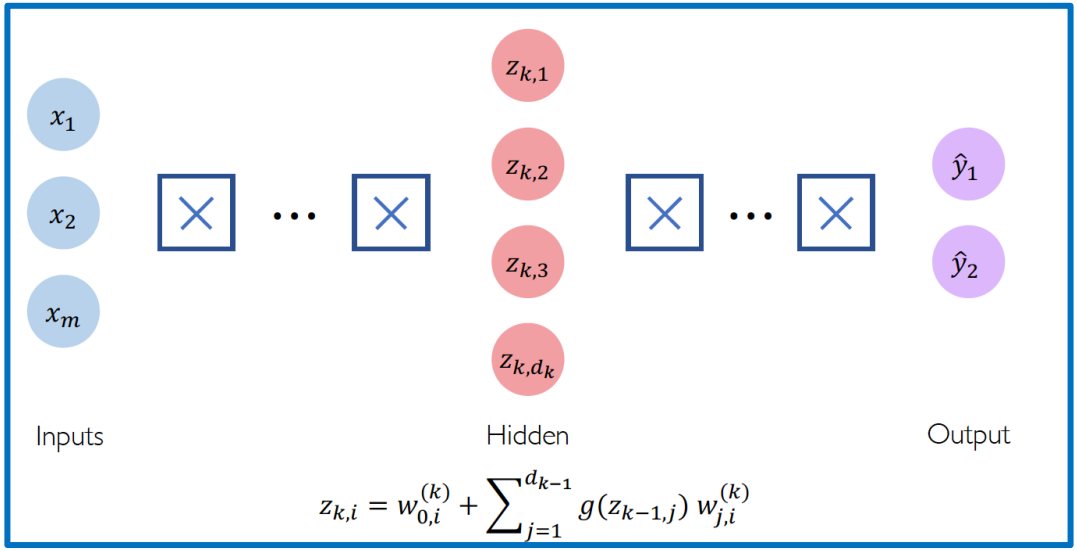
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$



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



$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j w_{j,2}^{(1)}$$
$$= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)}$$

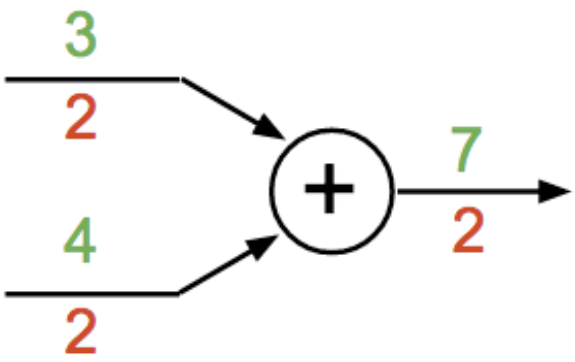


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

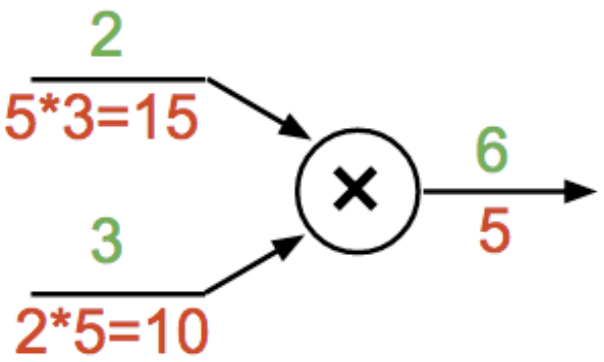
Forward and Backpropagation Code

Forward path : backward path

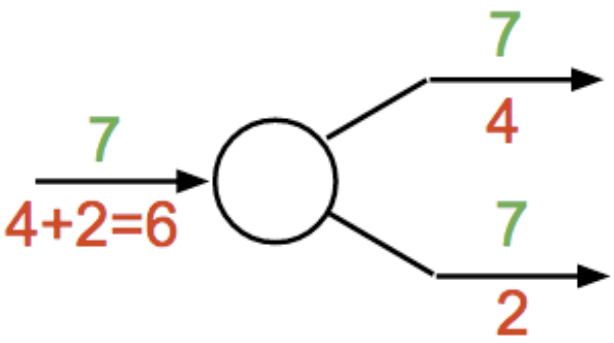
add gate: gradient distributor



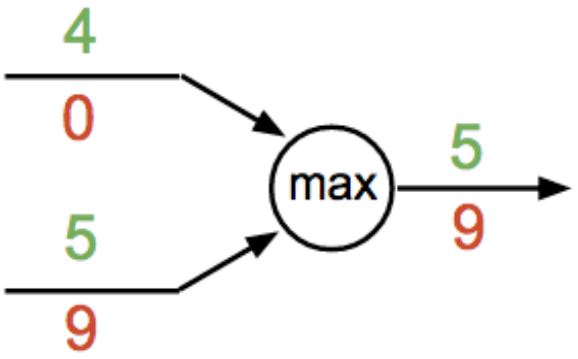
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router



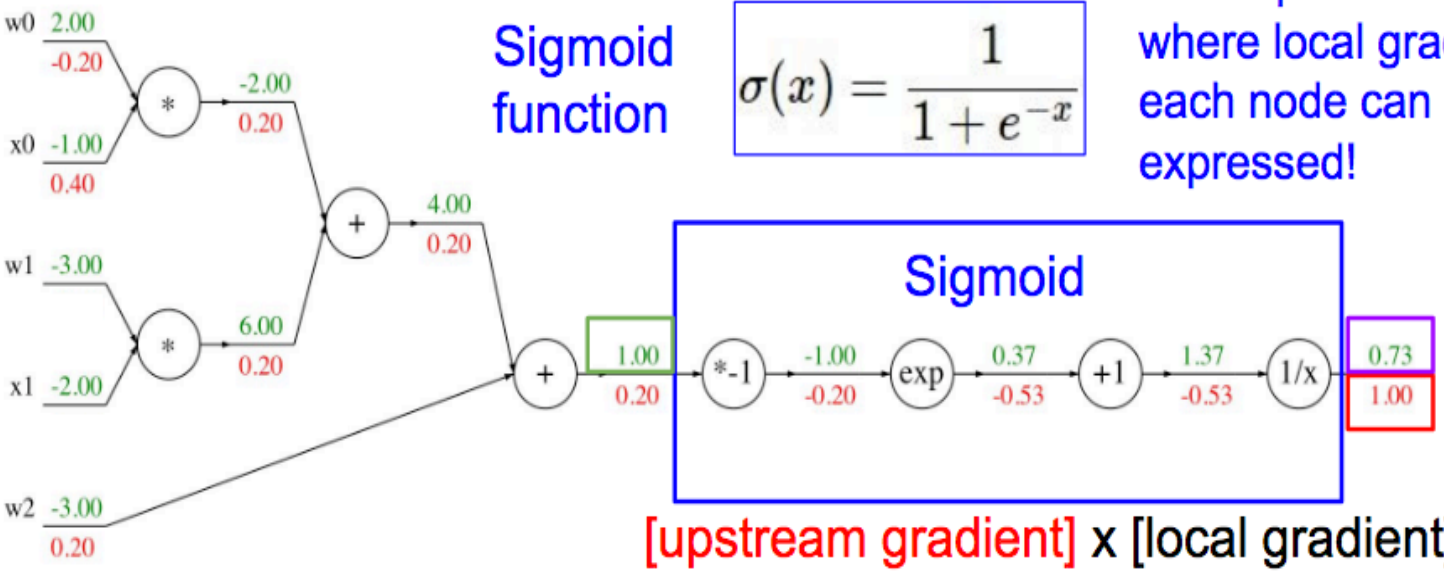
http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

Computational Graph

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

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

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



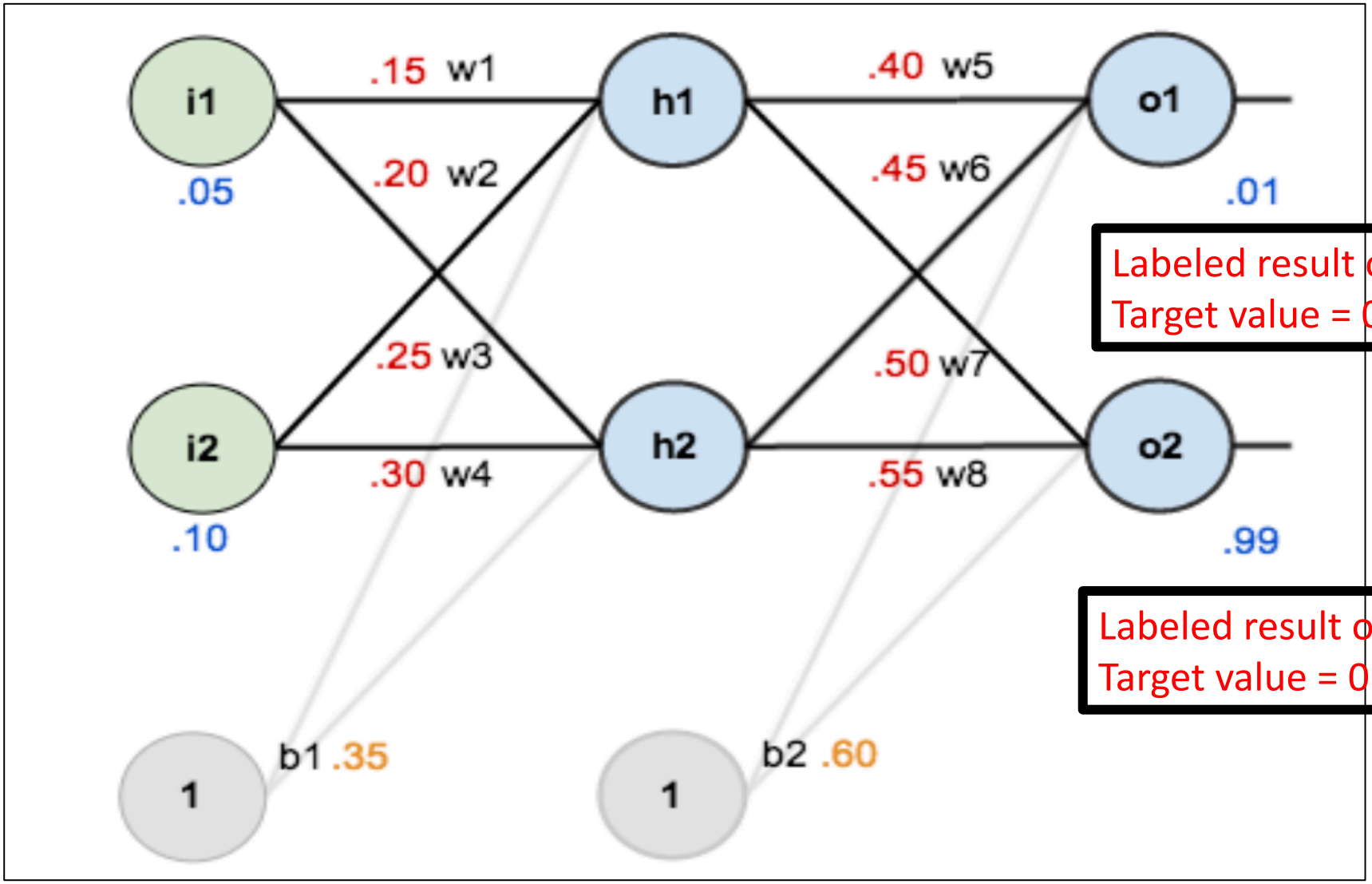
Sigmoid function $$\sigma(x) = \frac{1}{1+e^{-x}}$$

[upstream gradient] x [local gradient]

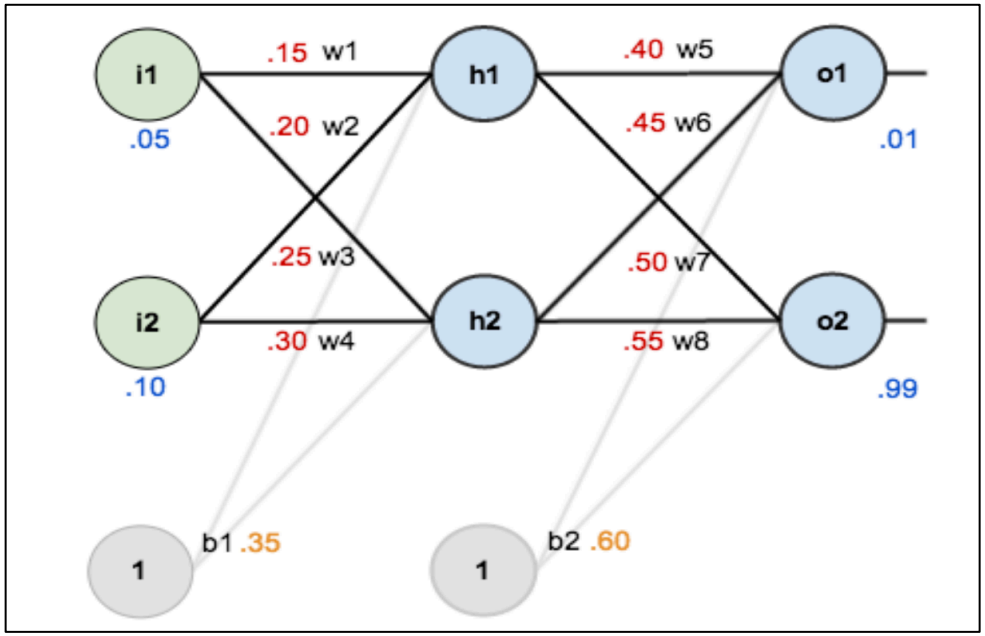
[1.00] x [(1 - 0.73) (0.73)] = 0.2

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

DNN Example


i1
.05
i2
.10
.15 w1
.20 w2
.25 w3
.30 w4
h1
h2
.40 w5
.45 w6
.50 w7
.55 w8
o1
.01
o2
.99
Labeled result of out1
Target value = 0.01
Labeled result of out2
Target value = 0.99
1
b1 .35
1
b2 .60

DNN Example (1)



w1=0.15

i1 = 0.05

*

w2=0.2

i2 = 0.1

*

+

+

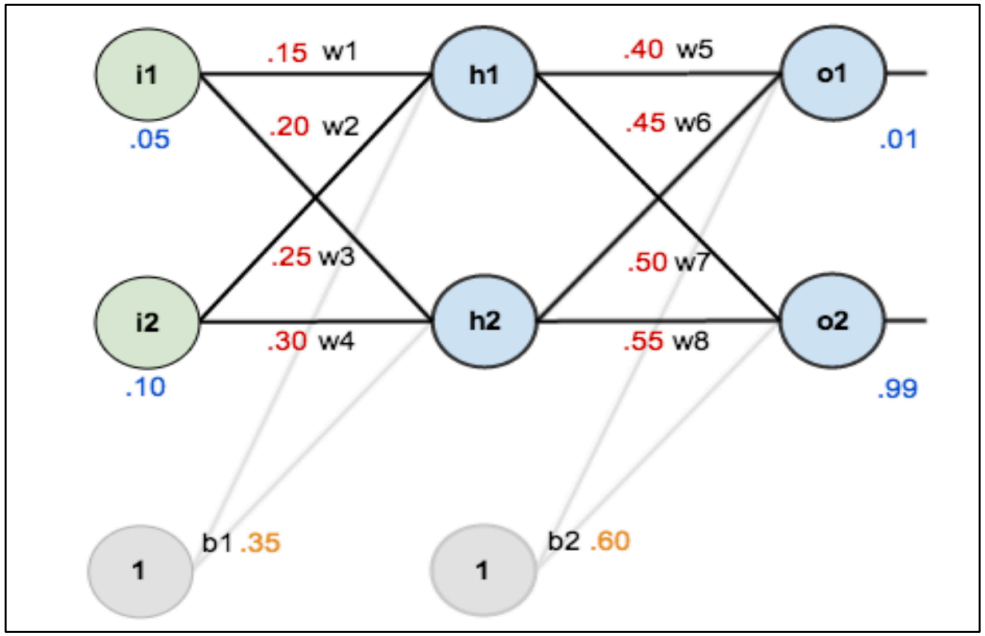
σ

h1 = 0.59326992

b1=0.35

1

DNN Example (2)



w3=0.25

i1 = 0.05

*

w4=0.3

i2 = 0.1

*

+

+

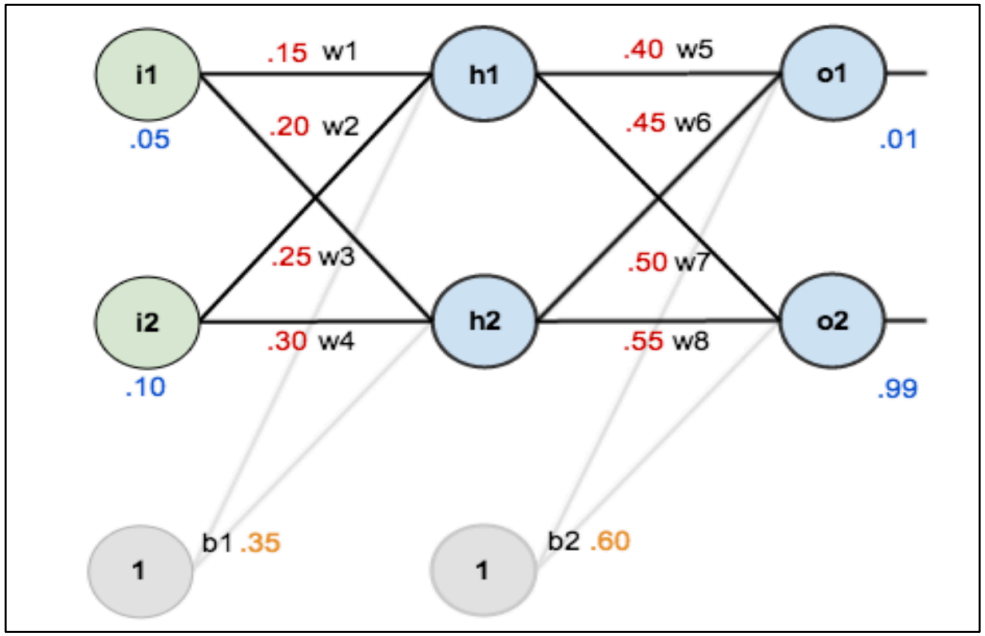
σ

h2 = 0.596884378

b1=0.35

1

DNN Example (3)



w5=0.4

h1= 0.59327

*

w6=0.45

h2 = 0.59688

*

+

+

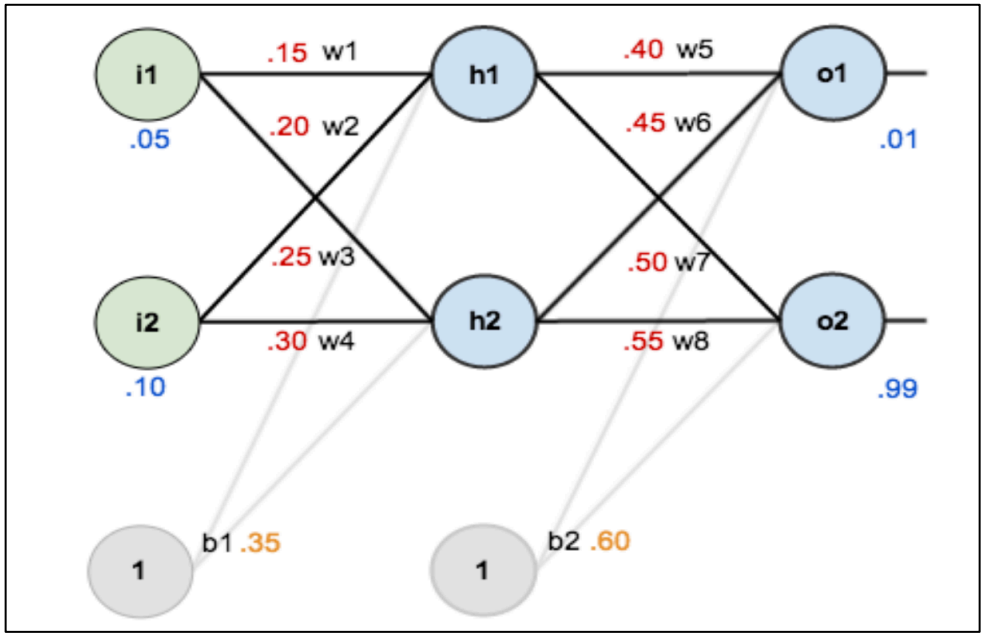
σ

o1 = 0.75136

b2=0.6

1

DNN Example (4)



w7=0.5

h1= 0.59327

*

w8=0.55

h2 = 0.59688

*

+

+

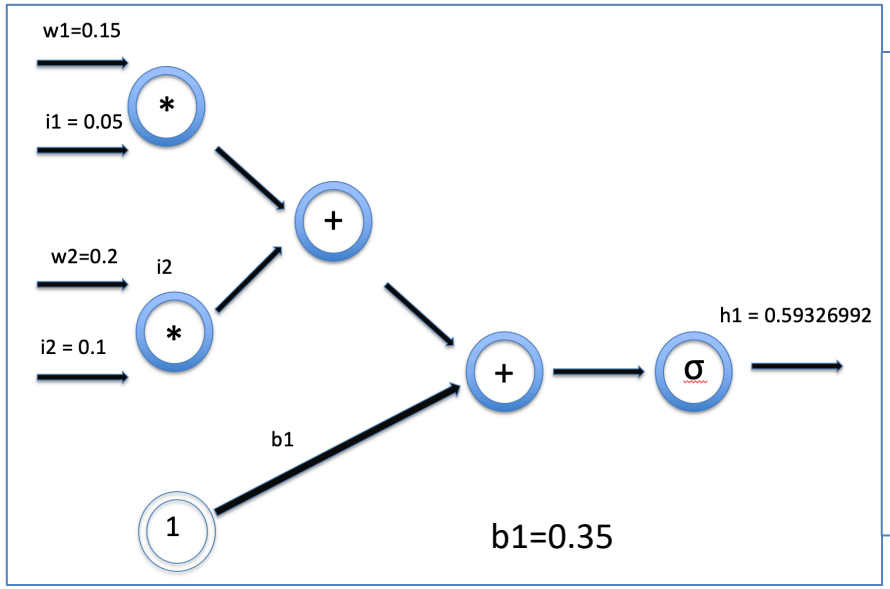
σ

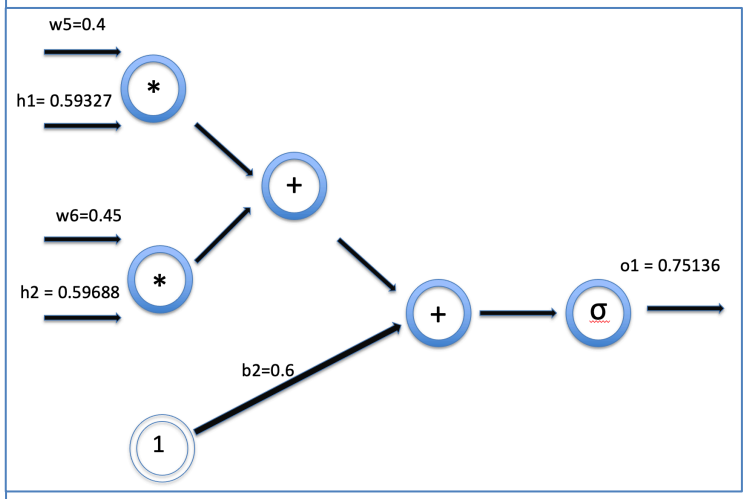
o2 = 0.77293

B2=0.6

1

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





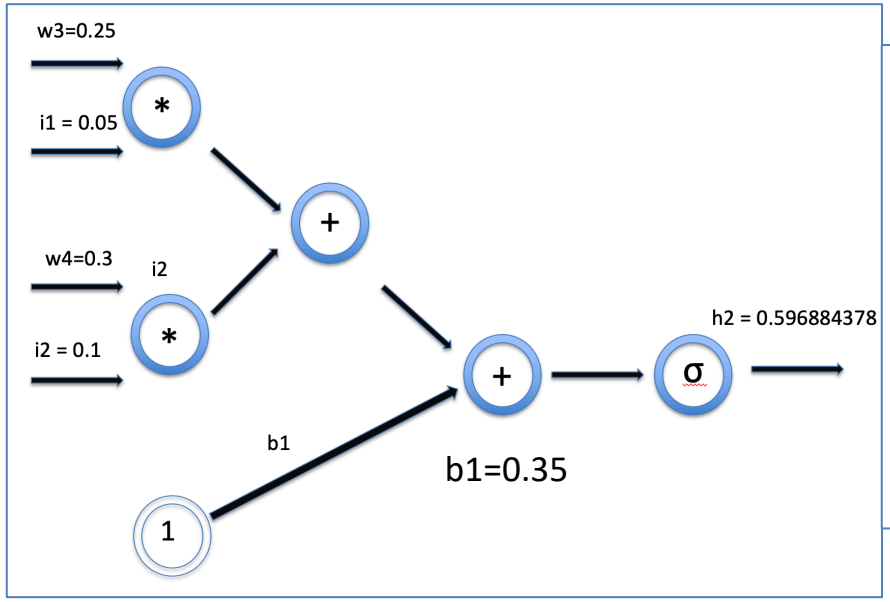
target=0.01

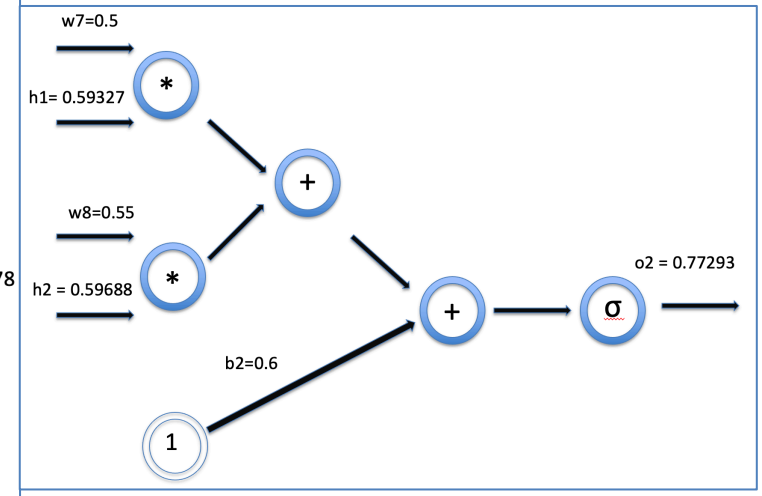
o1=0.75136

Eo1=0.27481

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

Etotal = Eo1 + Eo2 = 0.298371109





Eo2=0.02356

o2=0.77293

target=0.99

$$E_{o2} = 0.023560026$$

```
# 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

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

Training

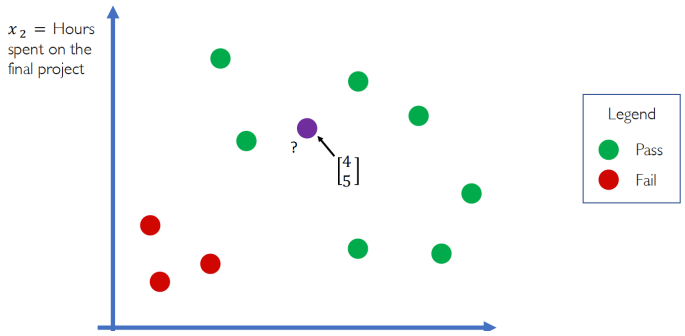
"Training a neural network"

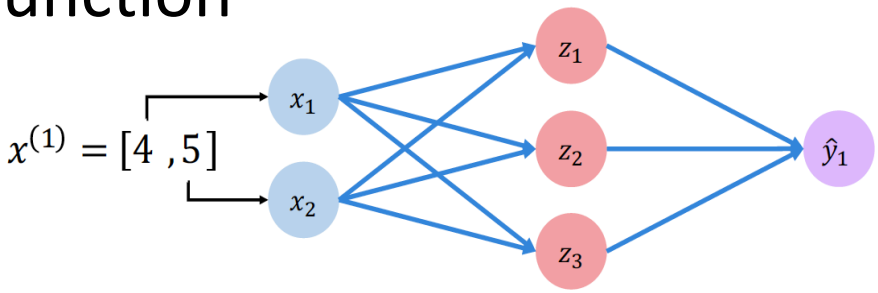
Supervised training
Compare the difference between the computed and the labeled given values

Loss values – loss function

Loss Function

Example Problem: Will I pass this class?



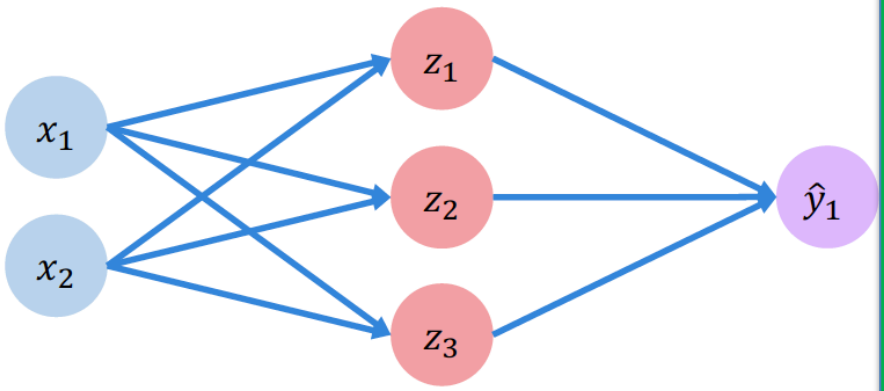


$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

*The **empirical loss** measures the total loss over our entire dataset*

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



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

Also known as:
- Objective function
- Cost function
- Empirical Risk

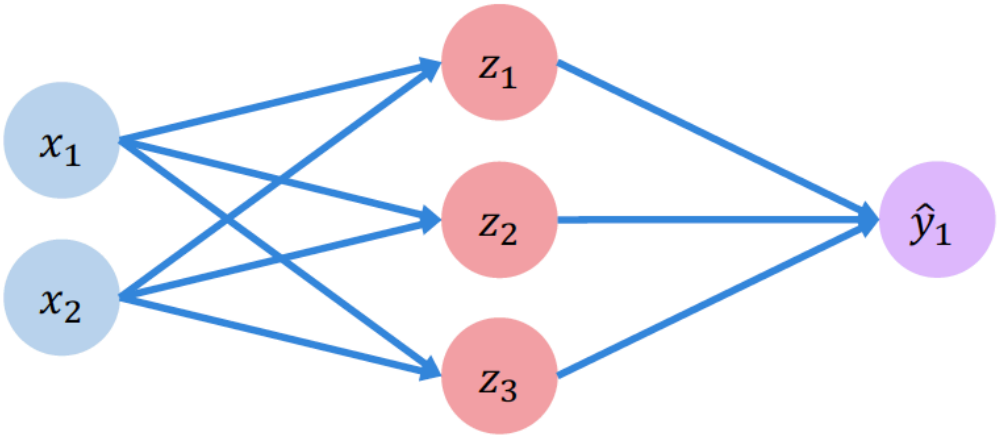
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Cross Entropy Loss

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

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



$f(x)$	y
0.1	1
0.8	0
0.6	1
⋮	⋮

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

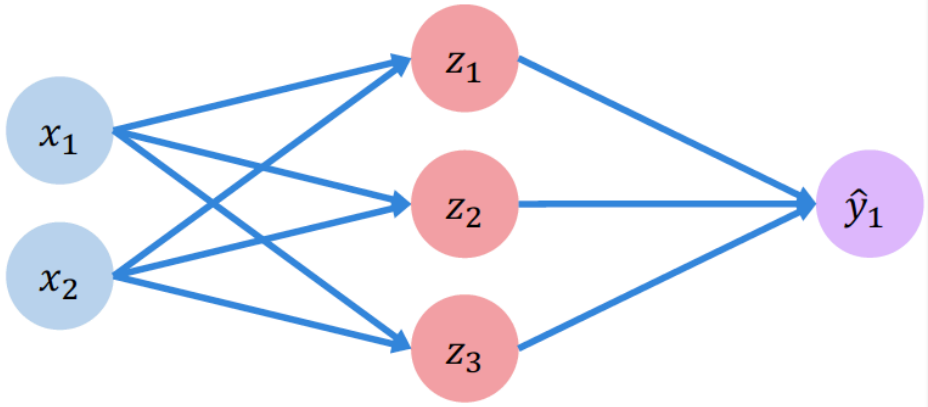
Actual — Predicted — Actual — Predicted

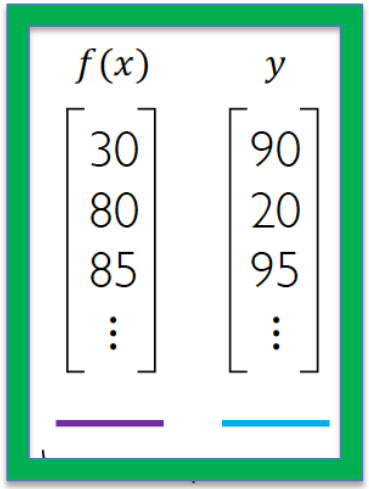
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Mean Squared Error Loss

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

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





Final Grades (percentage)

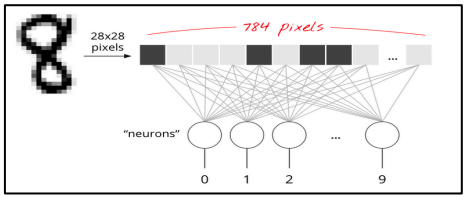
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

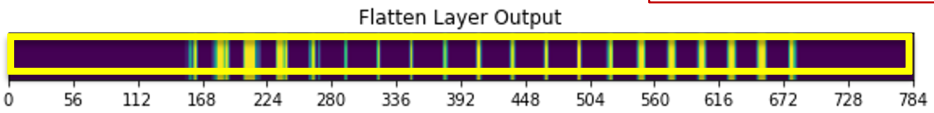
```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



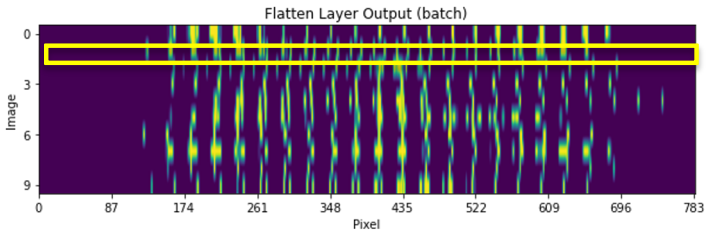
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

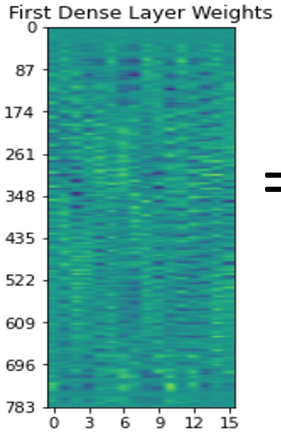
16 neurons
784 x 16 W matrix

10 x 16 output matrix

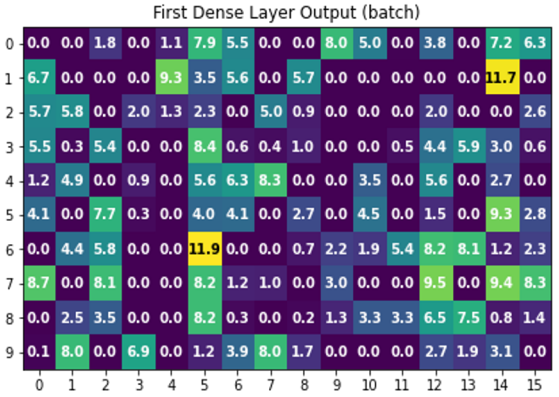
Input to 2nd layer



*

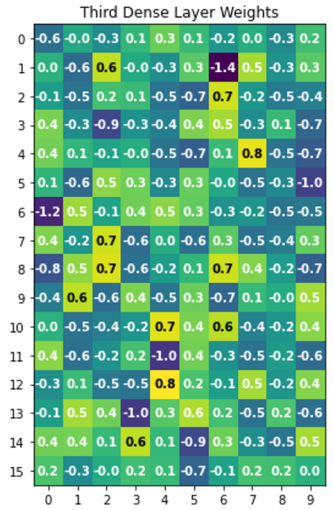


=



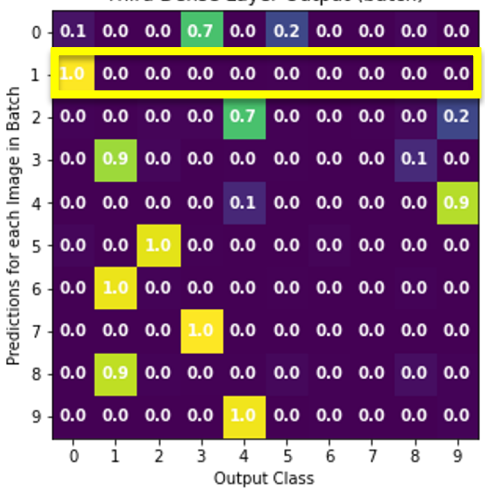
10 x 16 matrix

*

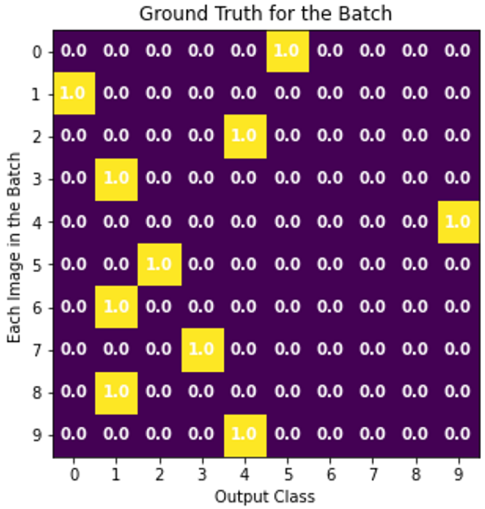


16 neurons
16 x 10
W matrix

=



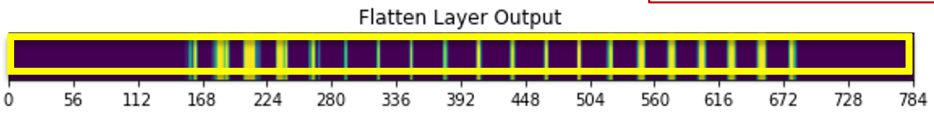
10 x 10 output matrix

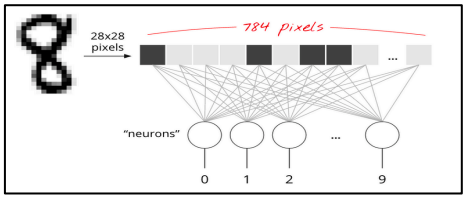


10 x 10 labeled matrix

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784

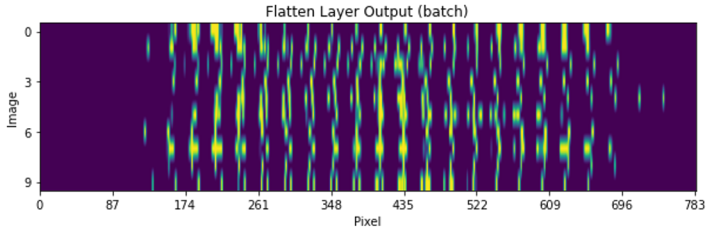




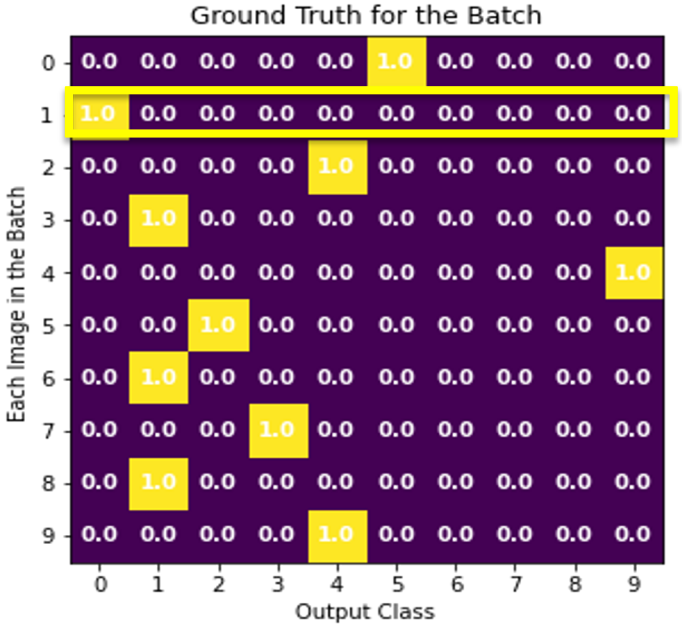
An image of 28x28 pixels

Input to 1st layer

10 images : batch of 10
10 x 784 matrix

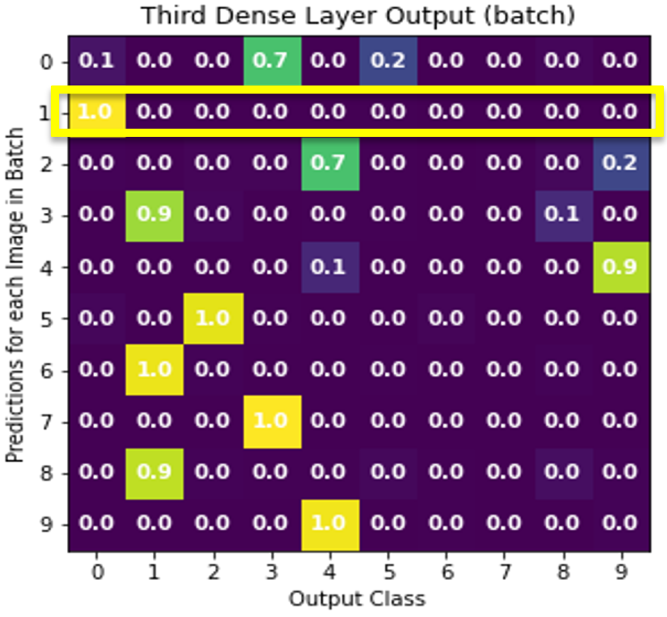


10 x 10 labeled matrix



Comparing Computed NN results To Labeled results (target)

10 x 10 output matrix

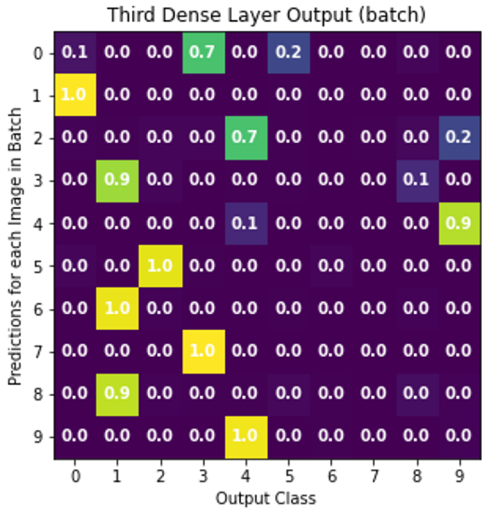


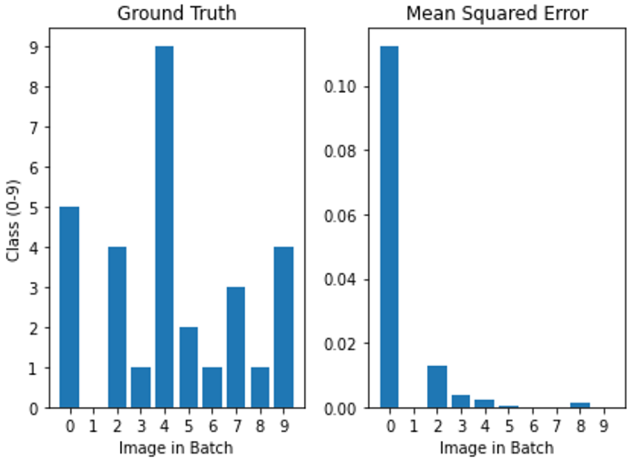
TensorFlow 2.0 : Forward Pass

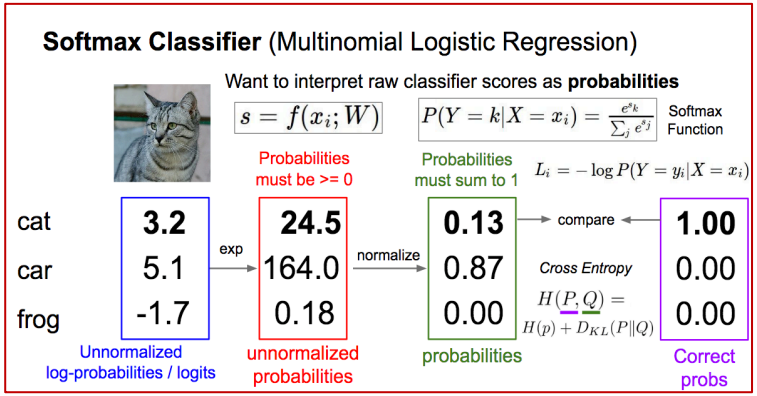
tf.losses.mean_squared_error(y_true_tf, y_pred_tf)

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

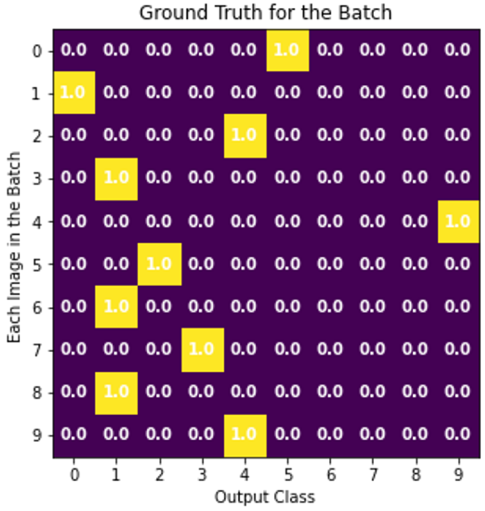
Evaluating the Error

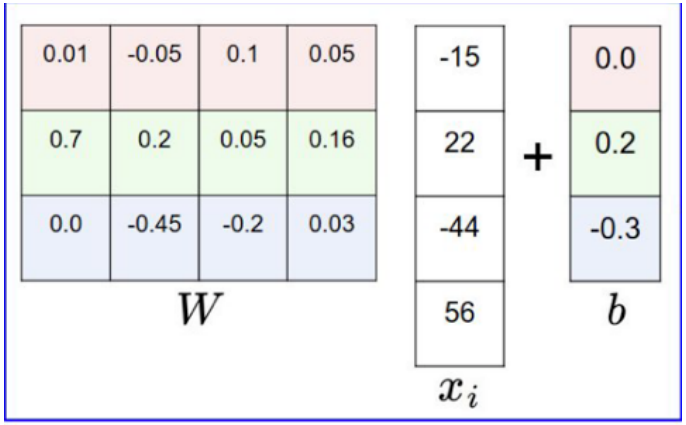


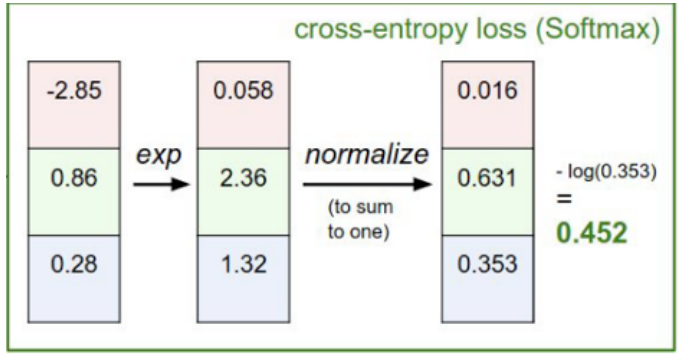




$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

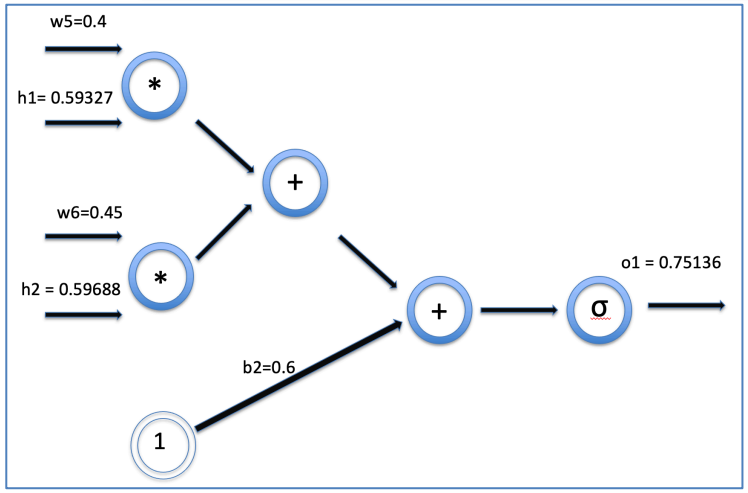






$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

target=0.01 | o1=0.75136 | Eo1=0.27481



Forward Path to Objective Function

target1 = 0.01

o1 = 0.75136

Cost f

Eo1 = 0.27481

+

ET = 0.298371

grad ET = 1.0

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

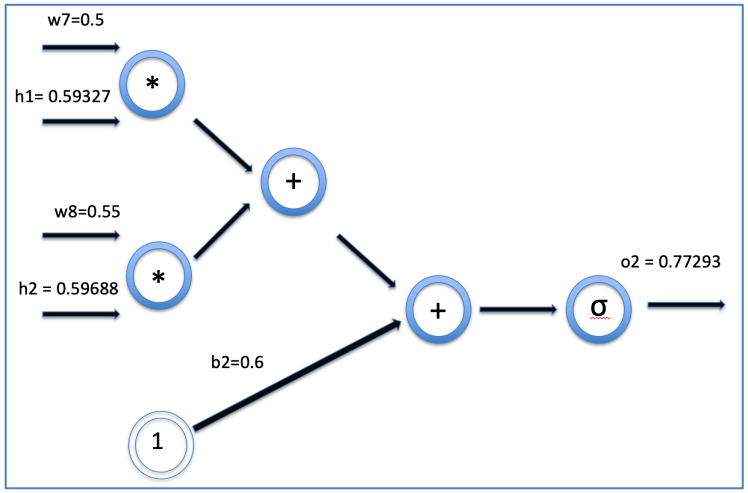
Etotal = Eo1 + Eo2 = 0.298371109

Eo2 = 0.02356

o2 = 0.77293

Cost f

target2 = 0.99



target=0.99 | o2=0.77293 | Eo2=0.02356

Training

What are the training parameters

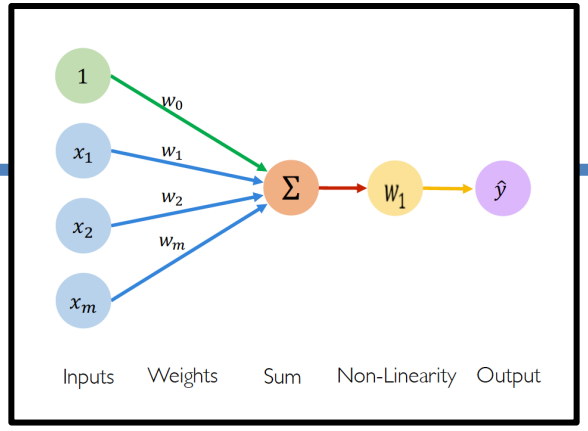
How does it work?

Weights and bias

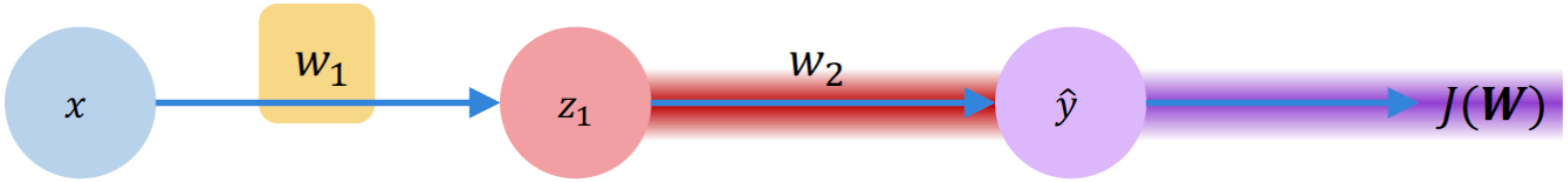
Weights = w1,w2,w3,w4,w5,w6,w7,w8

= number of links connecting different layers of neurons

Bias = b1-h1, b1-h2, b2-o1, b2-o2 = number of neurons



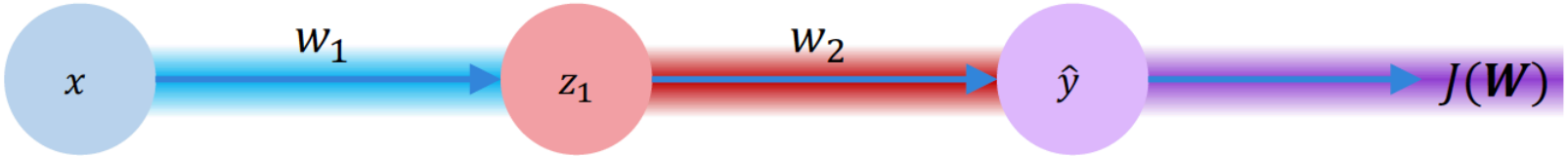
DNN MLP LA Kernels



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

Apply chain rule!

Apply chain rule!



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

DNN Forward Backward Calculation : Weights and bias

Algorithm

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

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

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

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

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

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

5. Return weights

In one epoch

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

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

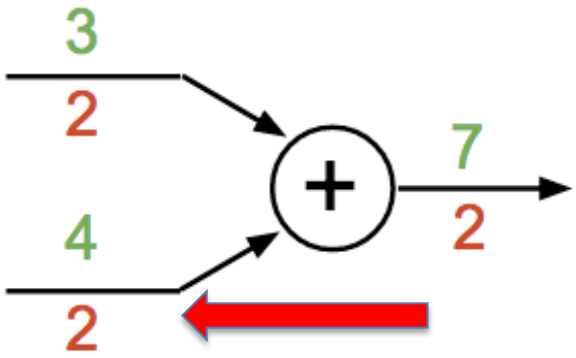
$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

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

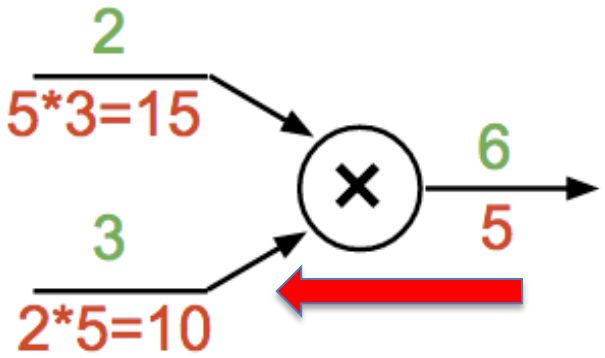
Forward and Backpropagation Code

Forward path : backward path

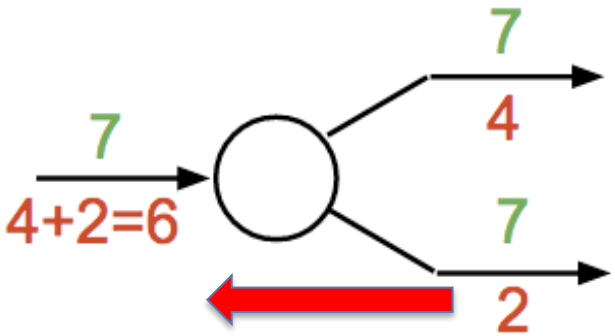
add gate: gradient distributor



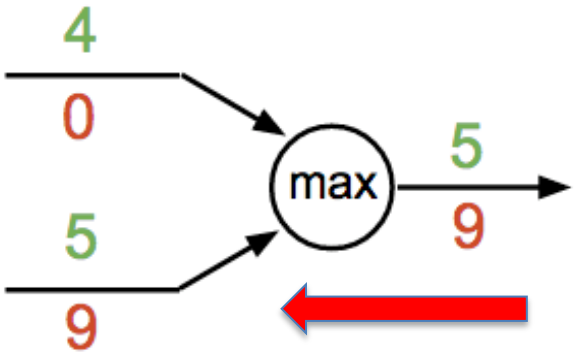
mul gate: "swap multiplier"



copy gate: gradient adder



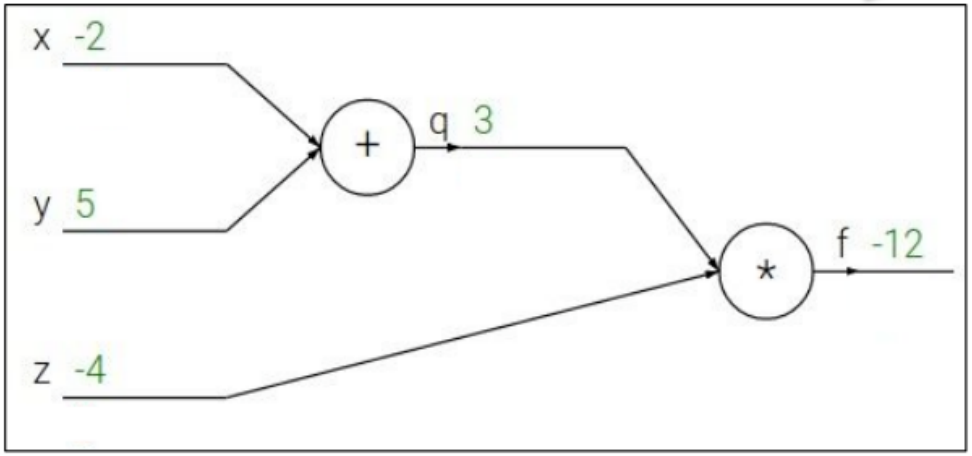
max gate: gradient router



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

Computational Graph

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



Forward path calculation

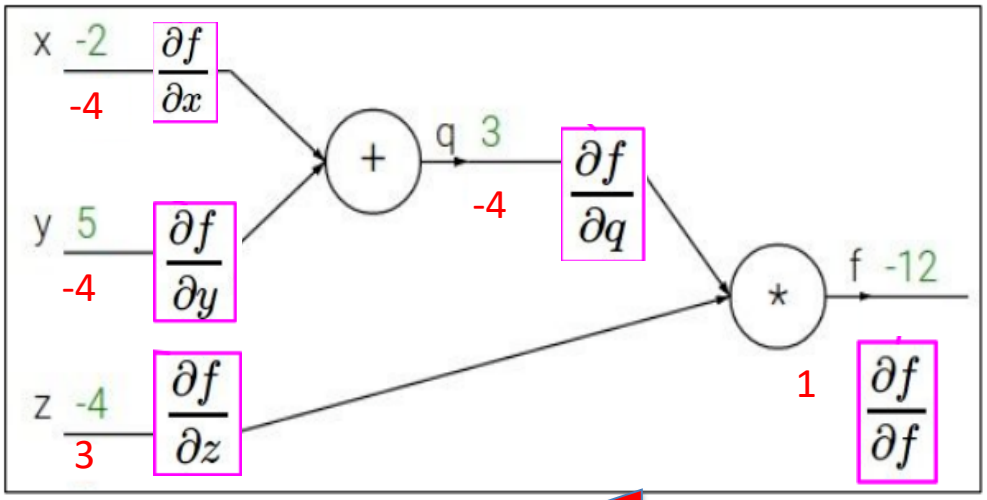
$$f(x,y,z) = (x+y)z$$

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

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

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

Need Gradient of f wrt to inputs



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

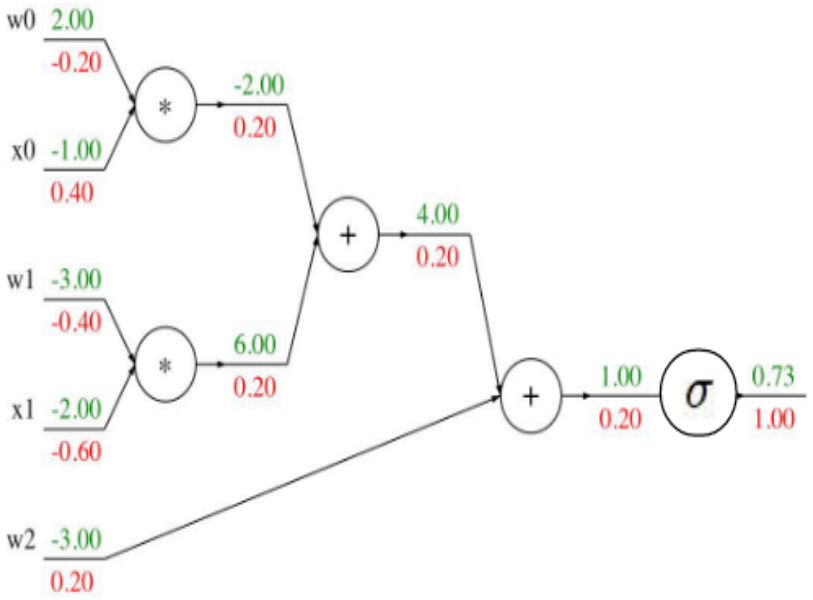
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$ = (z)(1) = -4

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

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

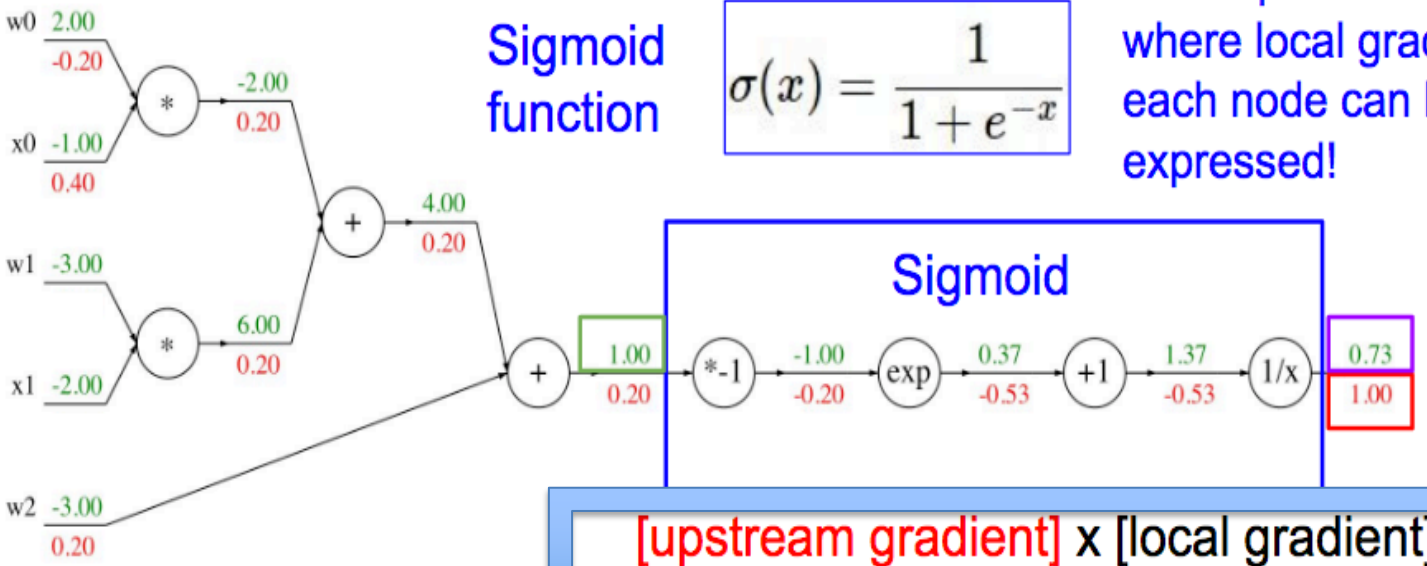
```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Computational Graph

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

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

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



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

[upstream gradient] x [local gradient]

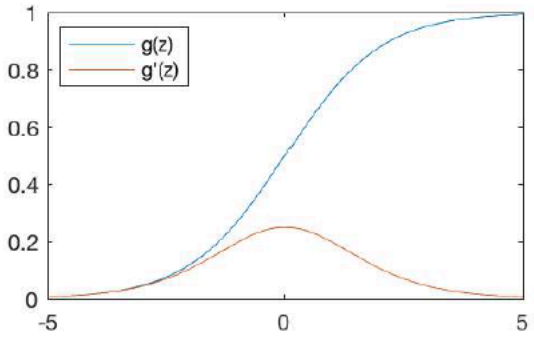
[1.00] x [(1 - 0.73) (0.73)] = 0.2

Sigmoid local gradient:

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

Nonlinear Activation Function

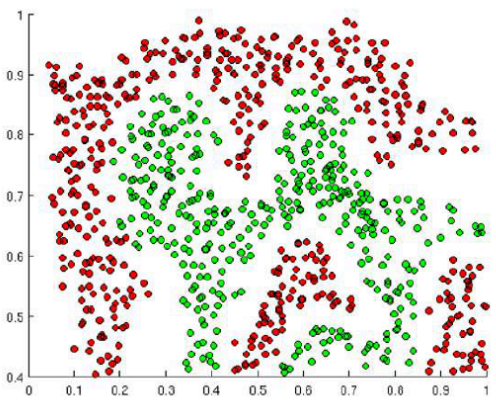
Sigmoid Function

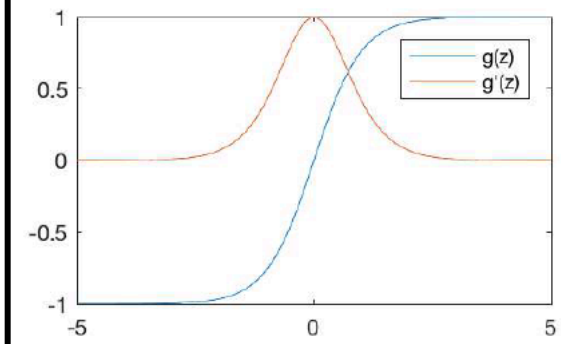


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

tf.nn.sigmoid(z)

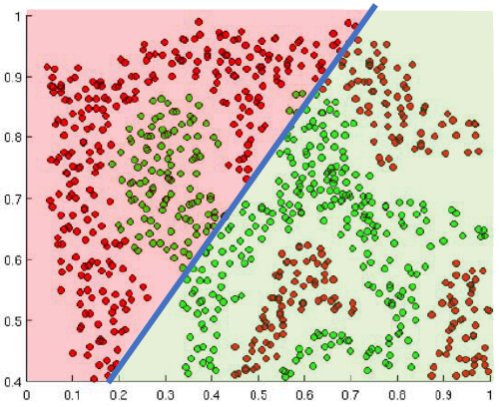


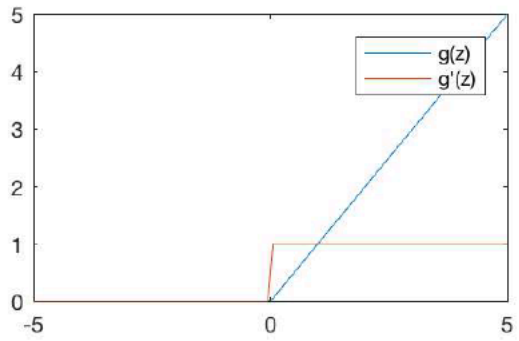


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

tf.nn.tanh(z)

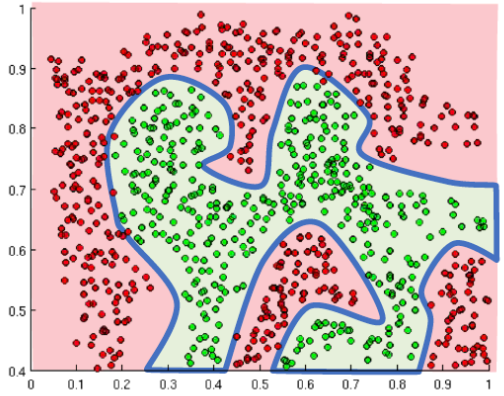




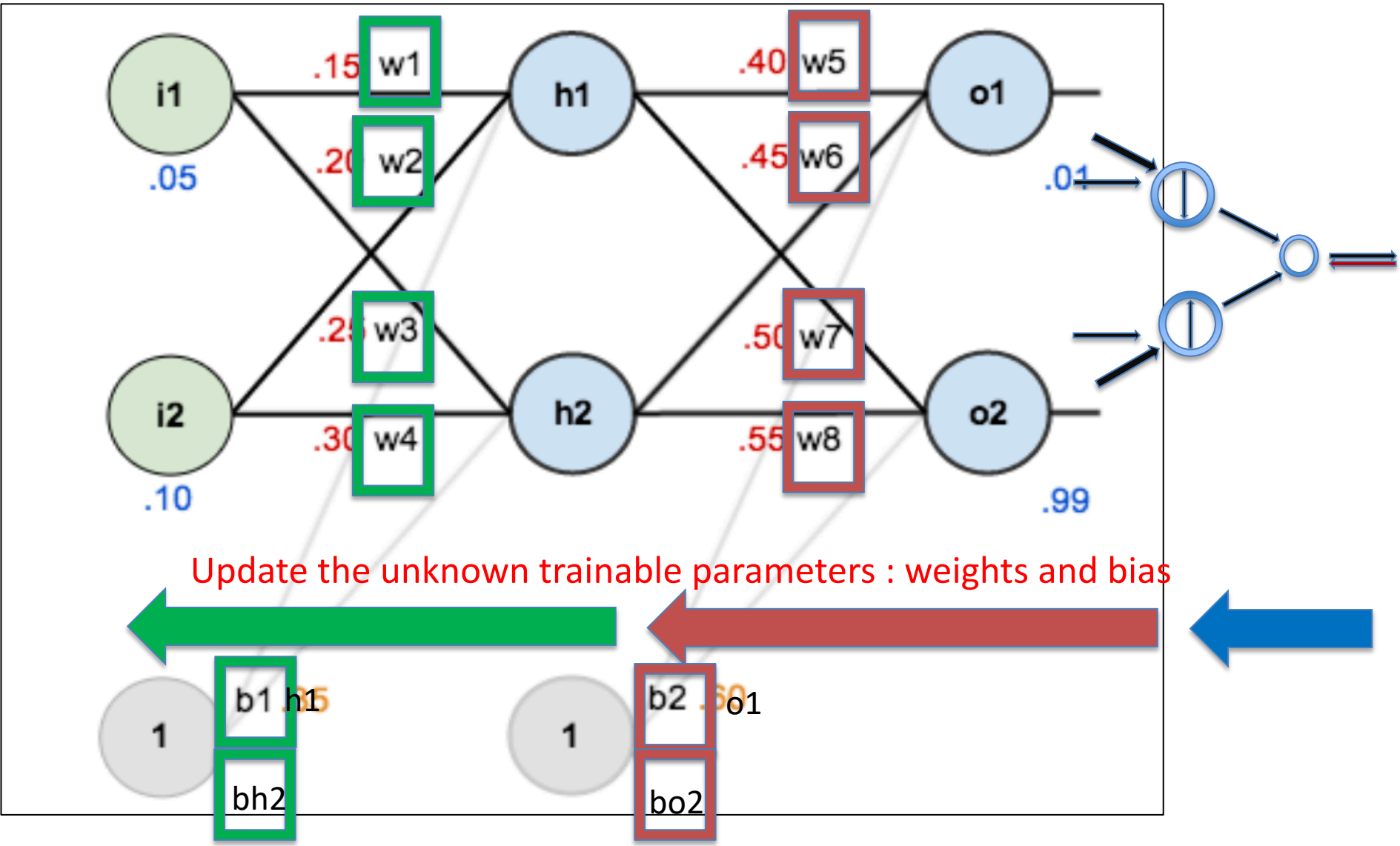
$$g(z) = \max(0, z)$$

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

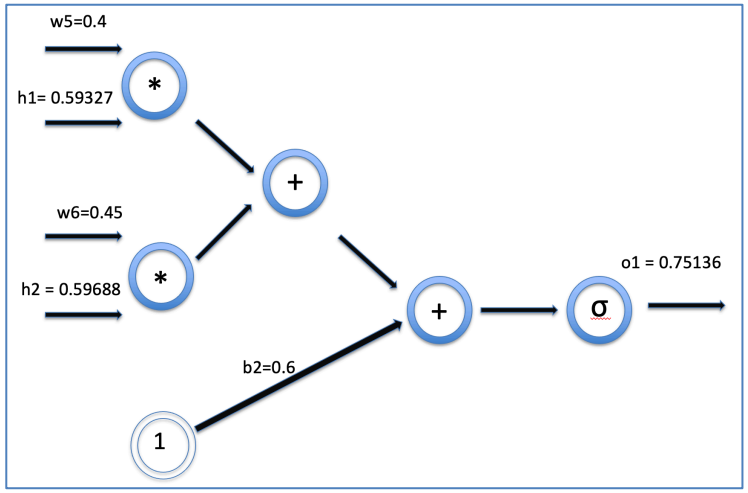
tf.nn.relu(z)



Multilayer Perceptron : Backward Path Calculation



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



target=0.01 | o1=0.75136 | Eo1=0.27481

Forward Path to Objective Function

target1 = 0.01

o1 = 0.75136

Cost f

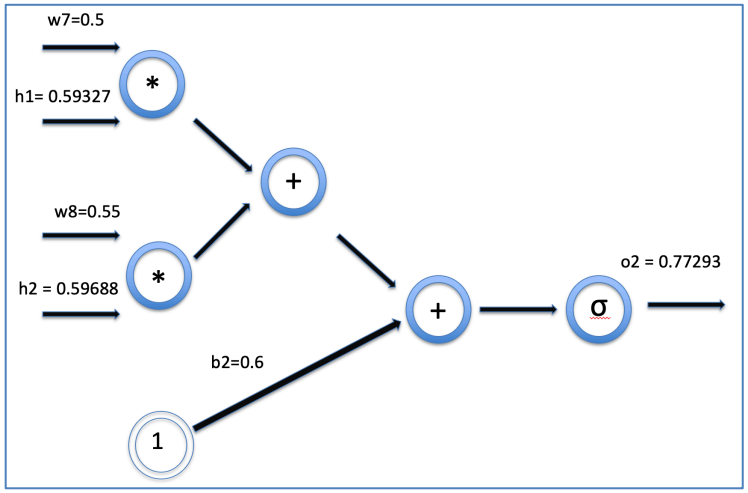
Eo1 = 0.27481

ET = 0.298371

grad ET = 1.0

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

Etotal = Eo1 + Eo2 = 0.298371109



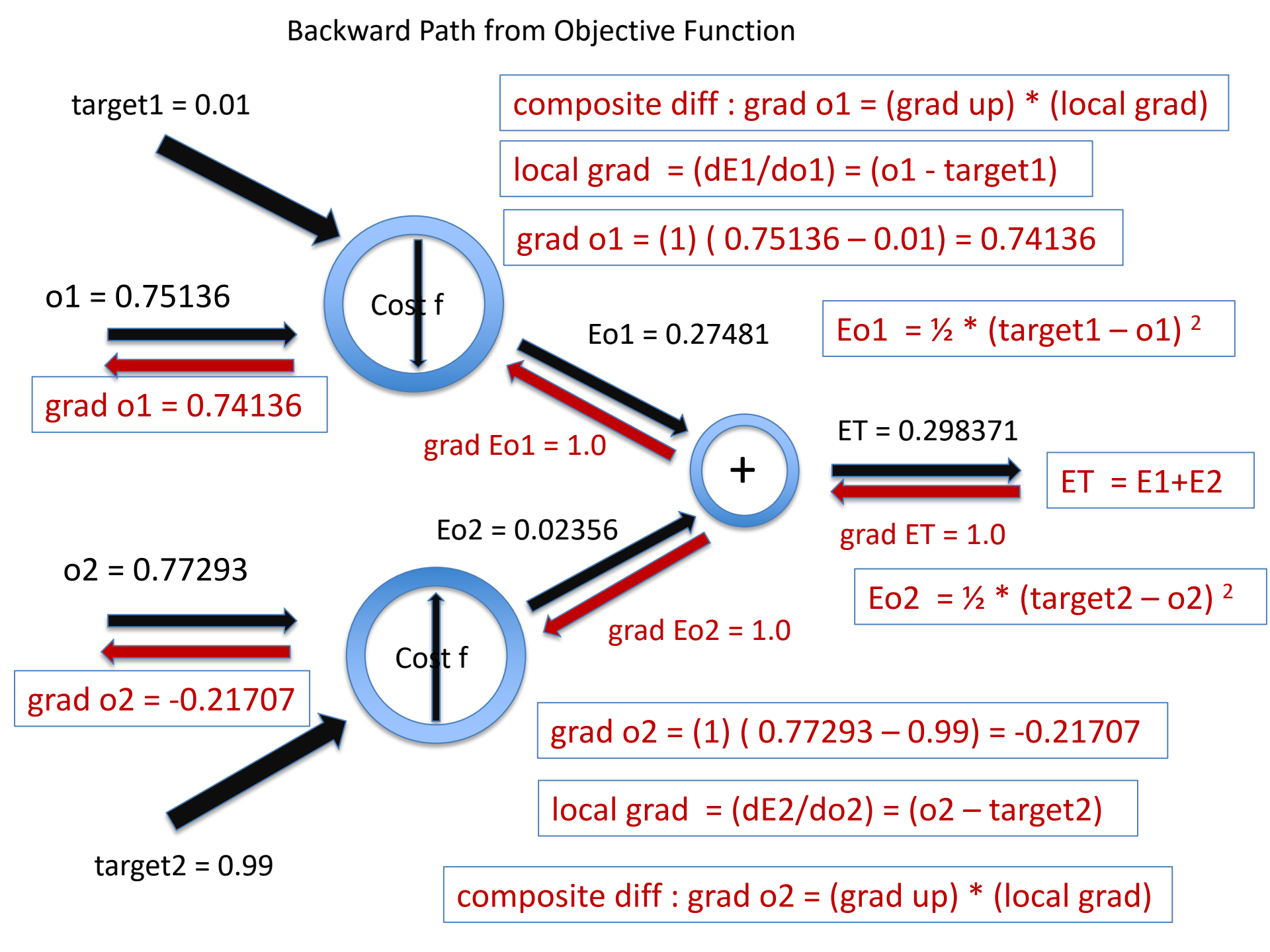
o2 = 0.77293

Cost f

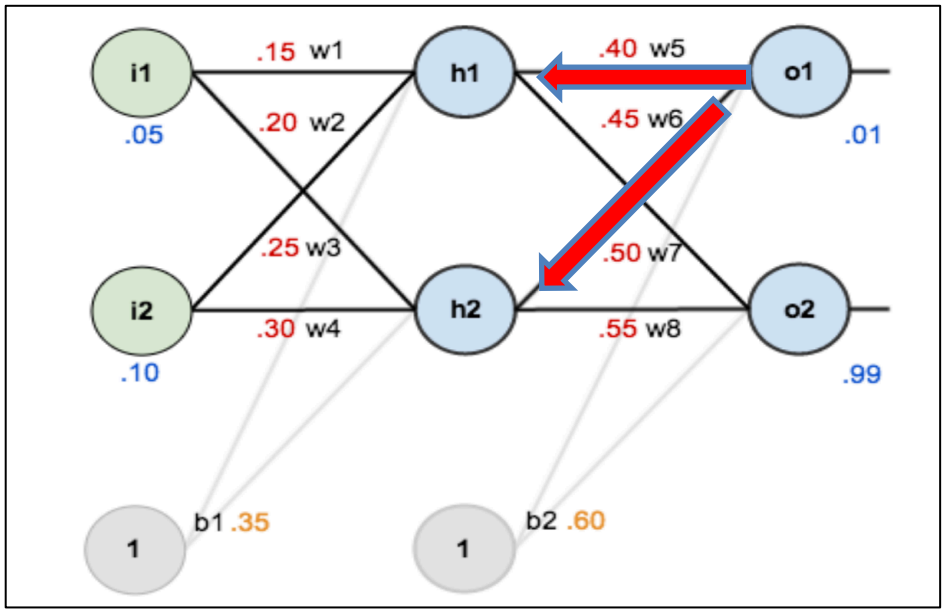
Eo2 = 0.02356

target2 = 0.99

target=0.99 | o2=0.77293 | Eo2=0.02356


Backward Path from Objective Function
target1 = 0.01
composite diff : grad o1 = (grad up) * (local grad)
local grad = (dE1/do1) = (o1 - target1)
grad o1 = (1) (0.75136 – 0.01) = 0.74136
o1 = 0.75136
Cost f
Eo1 = 0.27481
Eo1 = ½ * (target1 – o1) 2
grad o1 = 0.74136
grad Eo1 = 1.0
+
ET = 0.298371
ET = E1+E2
grad ET = 1.0
Eo2 = 0.02356
Eo2 = ½ * (target2 – o2) 2
o2 = 0.77293
grad Eo2 = 1.0
Cost f
grad o2 = -0.21707
grad o2 = (1) (0.77293 – 0.99) = -0.21707
local grad = (dE2/do2) = (o2 – target2)
target2 = 0.99
composite diff : grad o2 = (grad up) * (local grad)

Backward Path from o1



w5=0.4

h1= 0.59327

*

w6=0.45

h2 = 0.59688

*

+

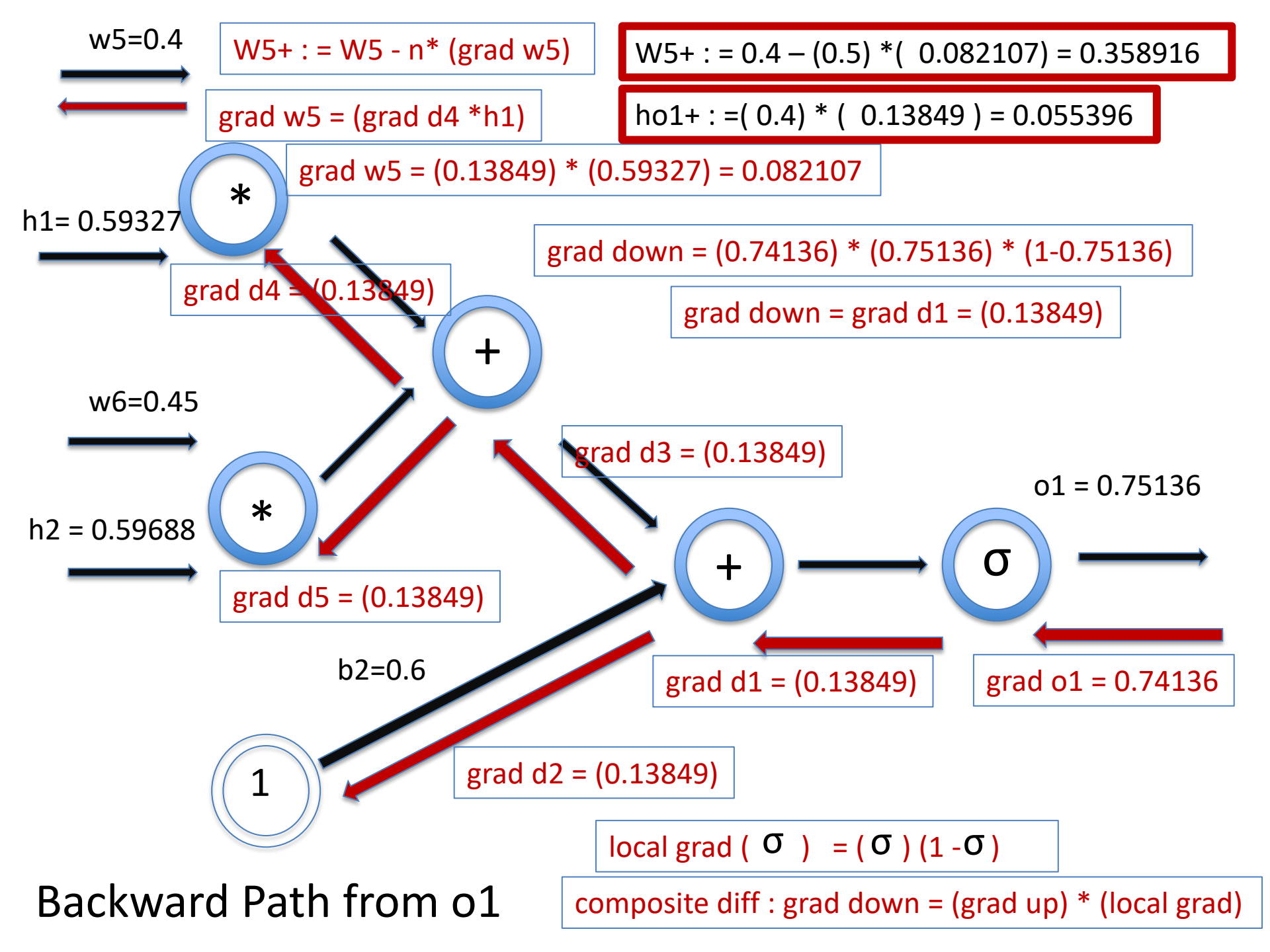
+

σ

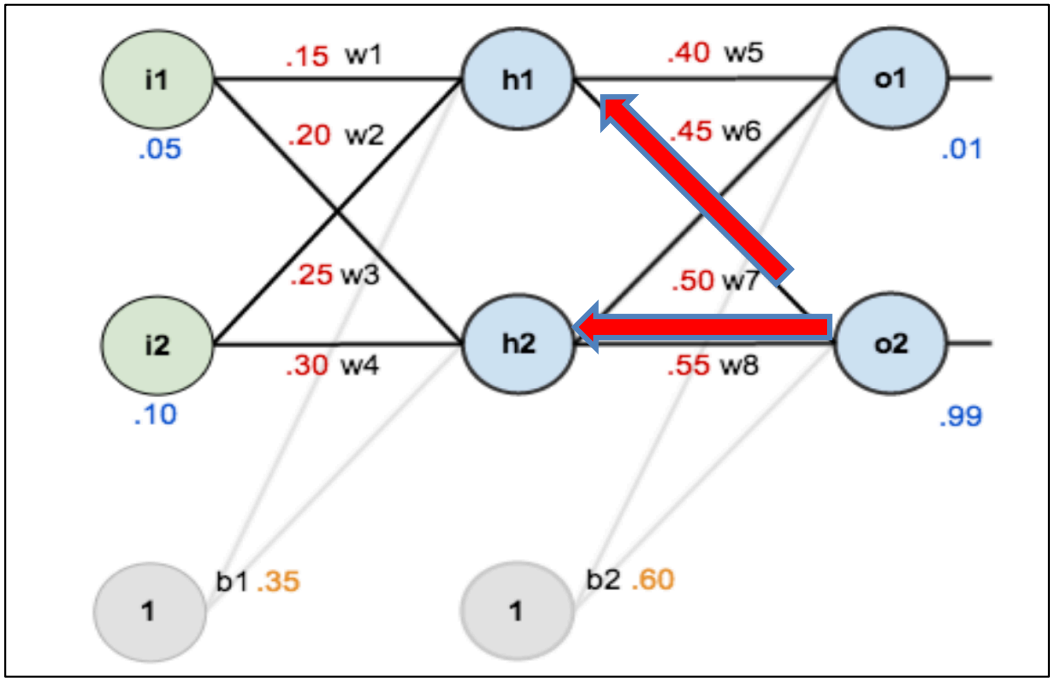
o1 = 0.75136

b2=0.6

1


w5=0.4
W5+ : = W5 - n* (grad w5)
W5+ : = 0.4 – (0.5) *(0.082107) = 0.358916
grad w5 = (grad d4 *h1)
ho1+ : =(0.4) * (0.13849) = 0.055396
*
grad w5 = (0.13849) * (0.59327) = 0.082107
h1= 0.59327
grad down = (0.74136) * (0.75136) * (1-0.75136)
grad d4 = (0.13849)
grad down = grad d1 = (0.13849)
+
w6=0.45
grad d3 = (0.13849)
o1 = 0.75136
*
h2 = 0.59688
+
σ
grad d5 = (0.13849)
b2=0.6
grad d1 = (0.13849)
grad o1 = 0.74136
1
grad d2 = (0.13849)
local grad (σ) = (σ) (1 -σ)
Backward Path from o1
composite diff : grad down = (grad up) * (local grad)

Backward Path from o2



w7=0.5

h1= 0.59327

*

w8=0.55

h2 = 0.59688

*

+

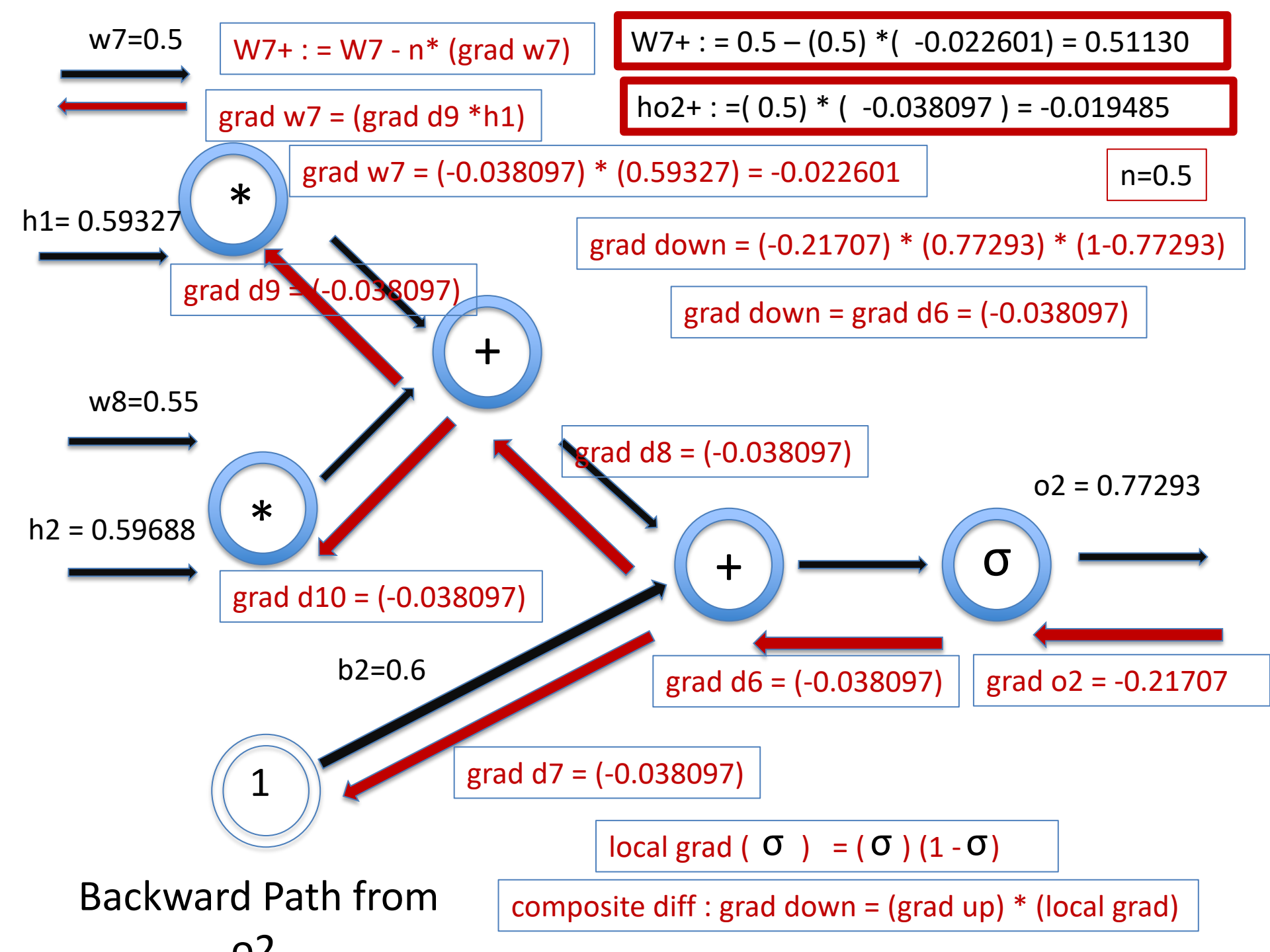
+

σ

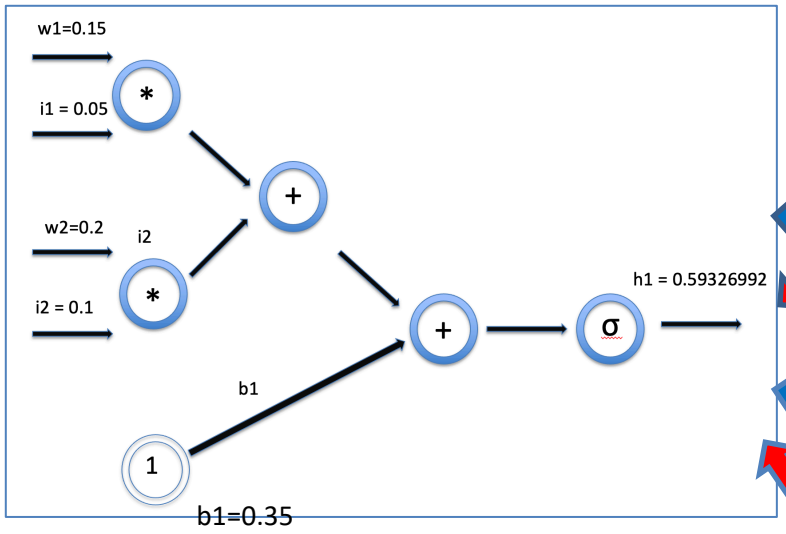
o2 = 0.77293

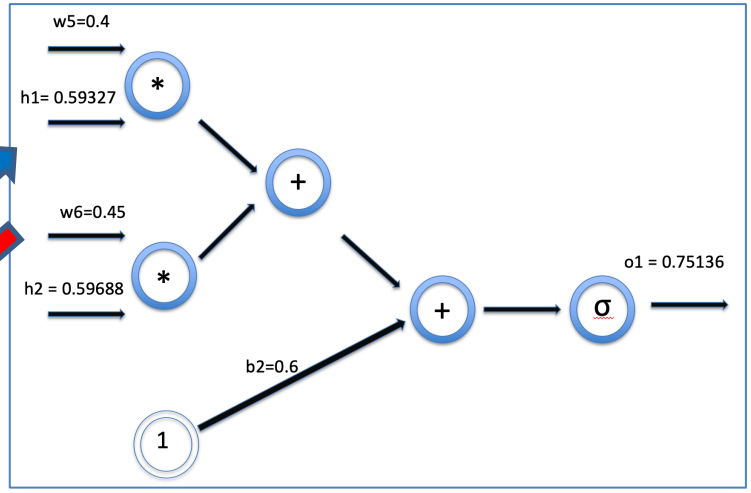
B2=0.6

1


w7=0.5
W7+ : = W7 - n* (grad w7)
W7+ : = 0.5 – (0.5) *(-0.022601) = 0.51130
grad w7 = (grad d9 *h1)
ho2+ : =(0.5) * (-0.038097) = -0.019485
grad w7 = (-0.038097) * (0.59327) = -0.022601
n=0.5
h1= 0.59327
*
grad down = (-0.21707) * (0.77293) * (1-0.77293)
grad d9 = (-0.038097)
grad down = grad d6 = (-0.038097)
+
w8=0.55
grad d8 = (-0.038097)
o2 = 0.77293
h2 = 0.59688
*
+
σ
grad d10 = (-0.038097)
b2=0.6
grad d6 = (-0.038097)
grad o2 = -0.21707
1
grad d7 = (-0.038097)
local grad (σ) = (σ) (1 - σ)
composite diff : grad down = (grad up) * (local grad)
Backward Path from o2

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



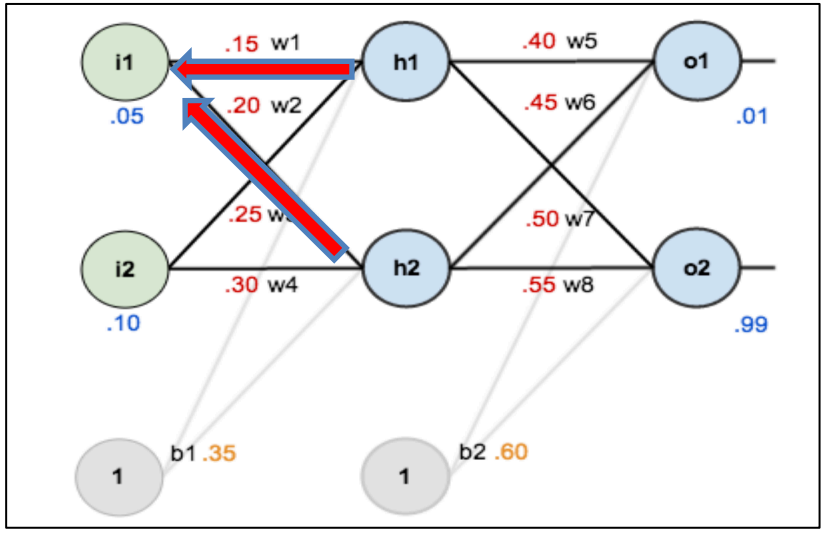


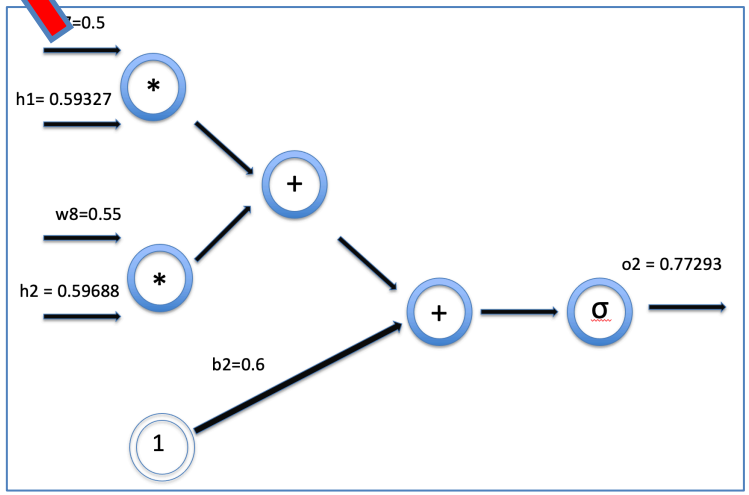
target=0.01

o1=0.75136

Eo1=0.27481

Etotal = Eo1 + Eo2 = 0.298371109

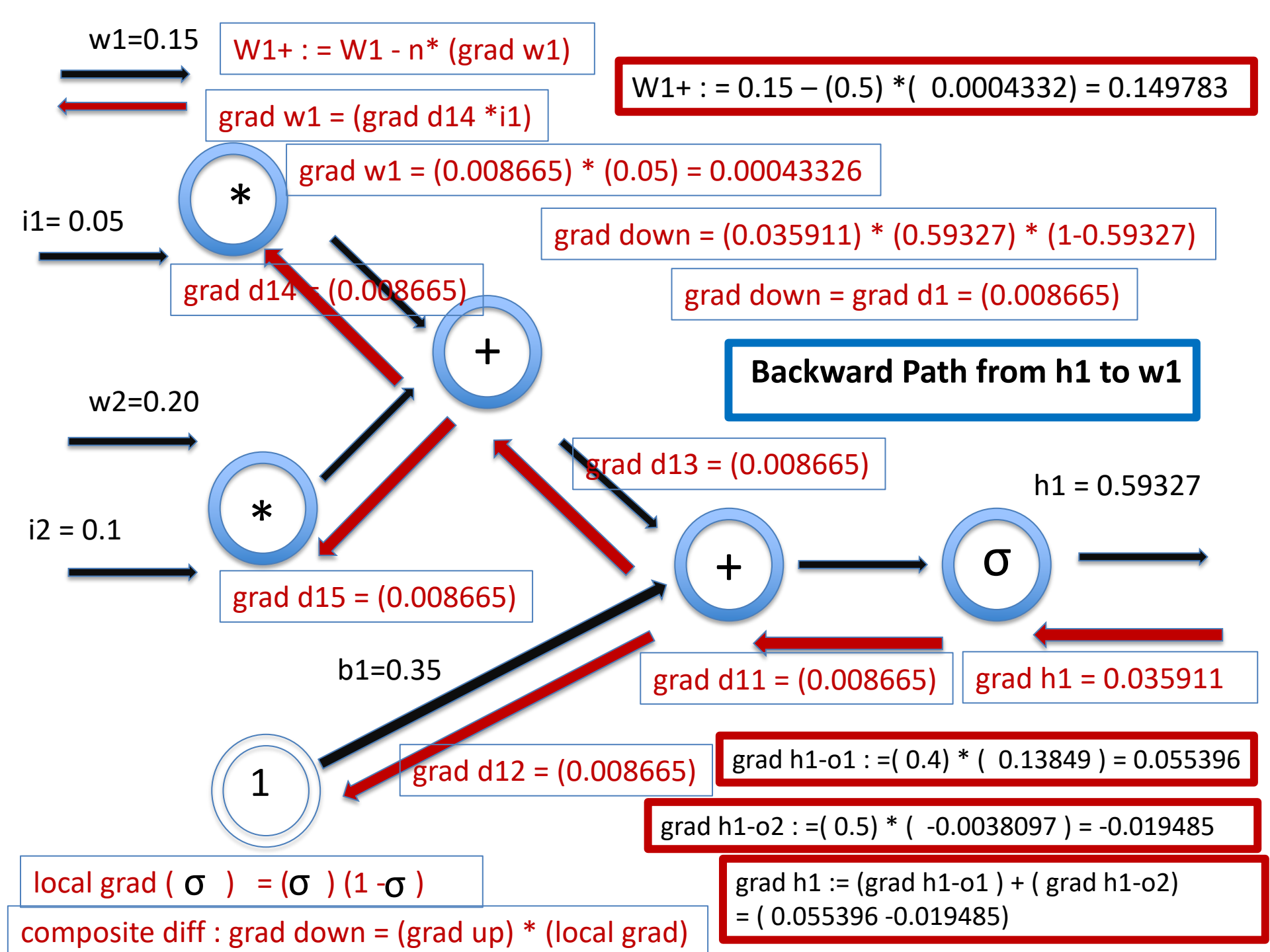




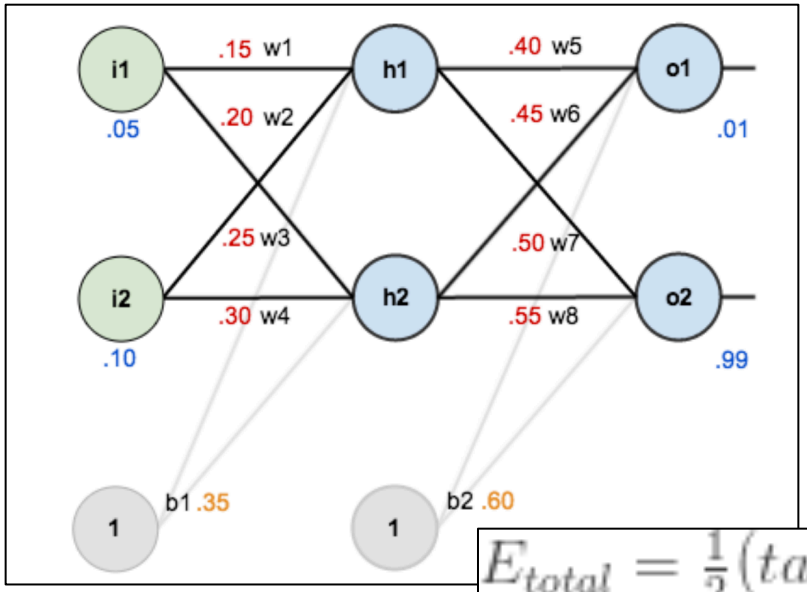
Eo2=0.02356

o2=0.77293

target=0.99

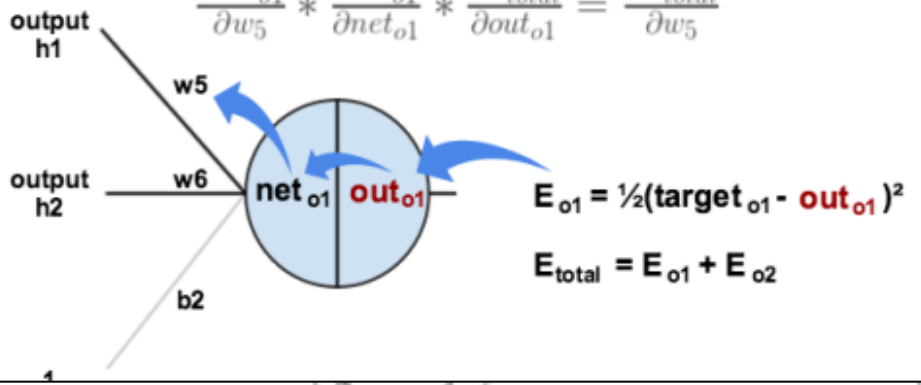

w1=0.15
W1+ : = W1 - n* (grad w1)
W1+ : = 0.15 – (0.5) *(0.0004332) = 0.149783
grad w1 = (grad d14 *i1)
grad w1 = (0.008665) * (0.05) = 0.00043326
*
i1= 0.05
grad down = (0.035911) * (0.59327) * (1-0.59327)
grad d14 = (0.008665)
grad down = grad d1 = (0.008665)
+
Backward Path from h1 to w1
w2=0.20
grad d13 = (0.008665)
h1 = 0.59327
*
i2 = 0.1
+
σ
grad d15 = (0.008665)
b1=0.35
grad d11 = (0.008665)
grad h1 = 0.035911
1
grad d12 = (0.008665)
grad h1-o1 : =(0.4) * (0.13849) = 0.055396
grad h1-o2 : =(0.5) * (-0.0038097) = -0.019485
local grad (σ) = (σ) (1 -σ)
grad h1 := (grad h1-o1) + (grad h1-o2)
= (0.055396 -0.019485)
composite diff : grad down = (grad up) * (local grad)

Backpropagation Example



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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

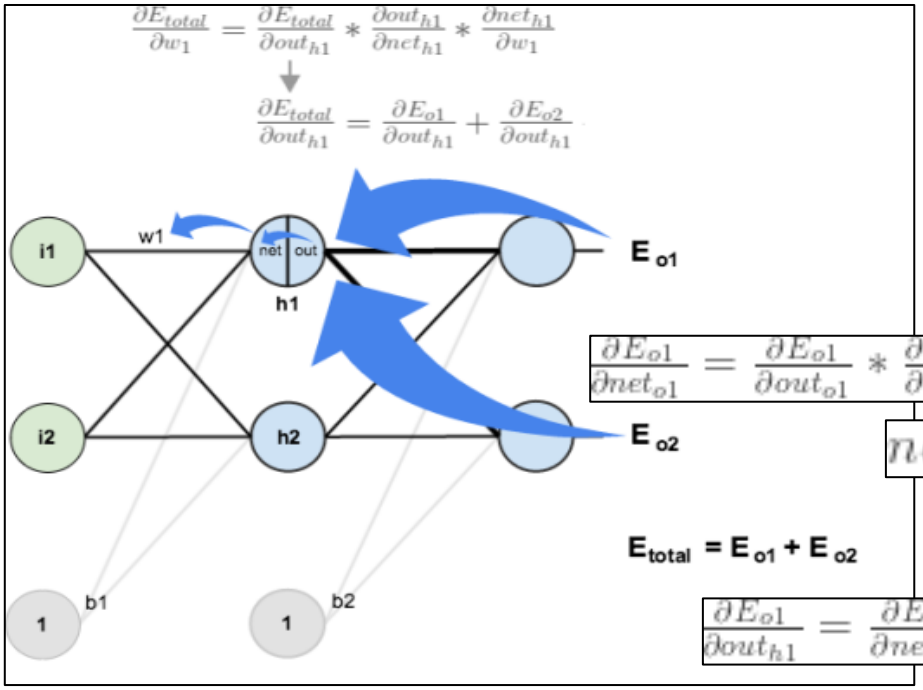
$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

Backpropagation Example



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

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

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

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1$$

$$w_2^+ = 0.19950143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

```
# 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))
```

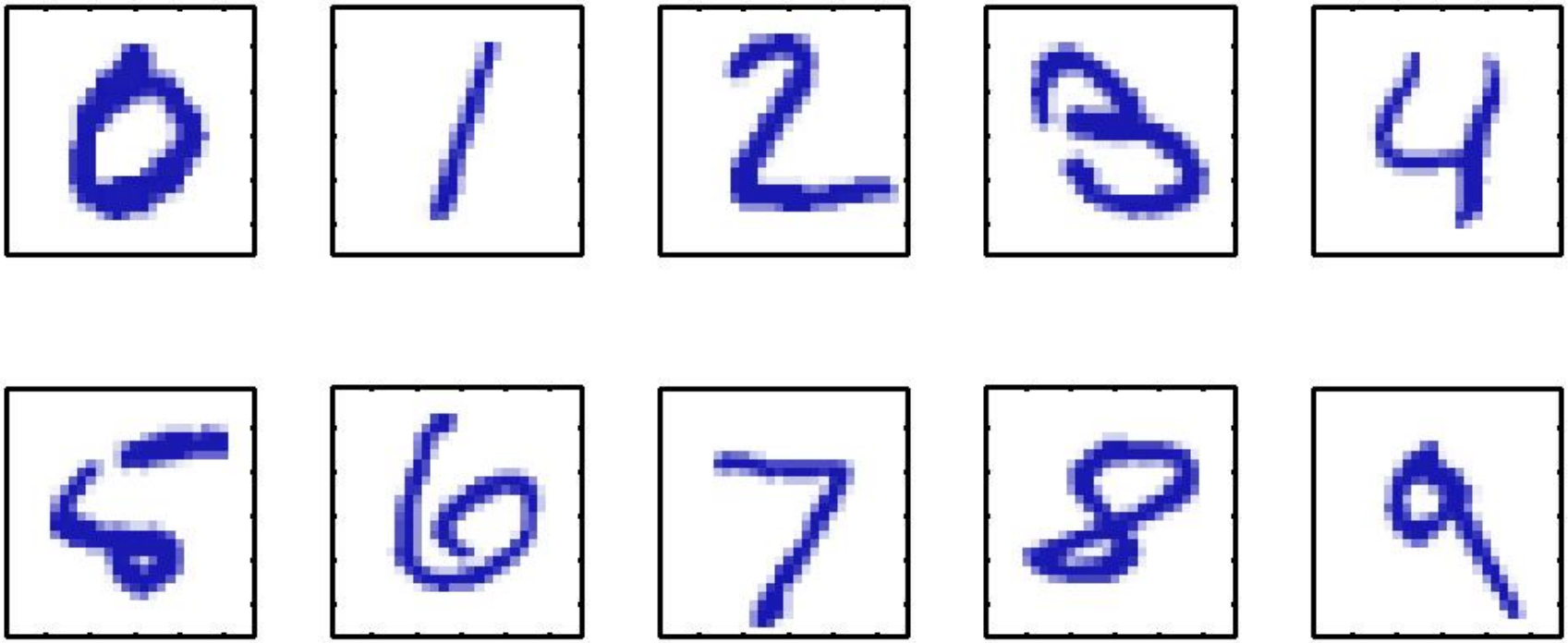
w1+: 0.1497807161327648
w2+: 0.1997807161327648 2
w3+: 0.2497511436323716 4
w4+: 0.2995022872647432 6
w5+: 0.3589164797177565 3
w6+: 0.4086661860761087
w7+: 0.5113012702391395
w8+: 0.5613701211083925

grad_o1: 0.7413650695475076
grad_o2: -0.2170715346835789 8
grad_d1: 0.1384985616294507 6
grad_d2: -0.0380982365180384 4
grad_w5: 0.0821670405644870 1
grad_w6: 0.0826676278477826 3
grad_w7: -0.0226025404782790 4
grad_w8: -0.0227402422167847 7
grad_h1: 0.0363530639276108 6
grad_d11: 0.0087713546894077 9
grad_w1: 0.0004385677344703895
grad_w2: 0.0004385677344703895
grad_h2: 0.0413703226483317 1
grad_d22: 0.0099542547051342 71
grad_w3: 0.0004977127352567136
grad_w4: 0.0009954254705134271

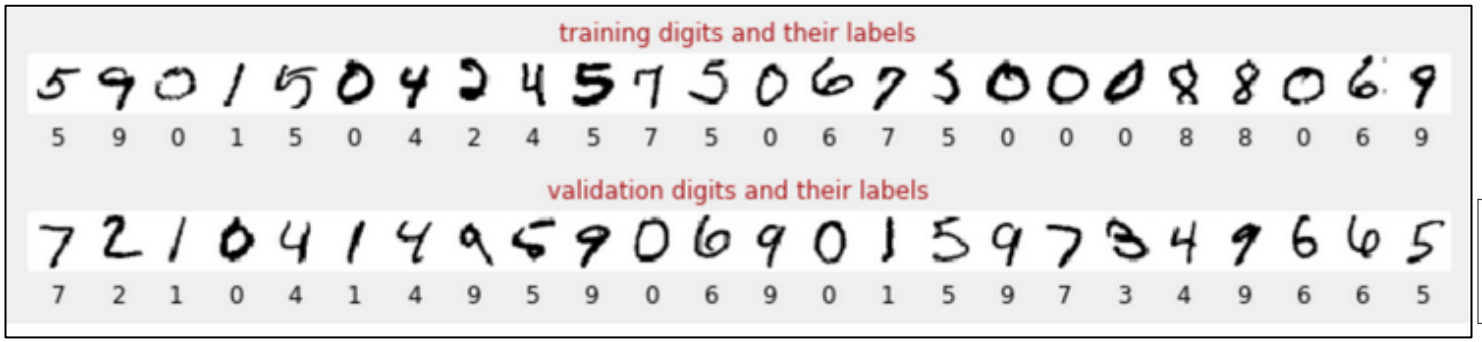
Example: how can computer see images?

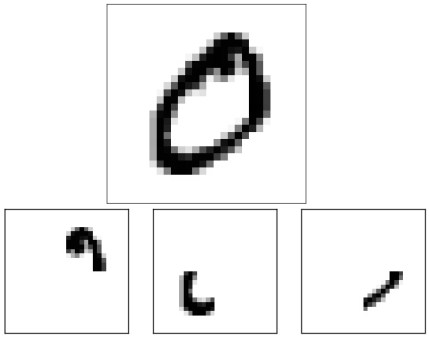
Handwritten Digit Recognition (MNIST data set)

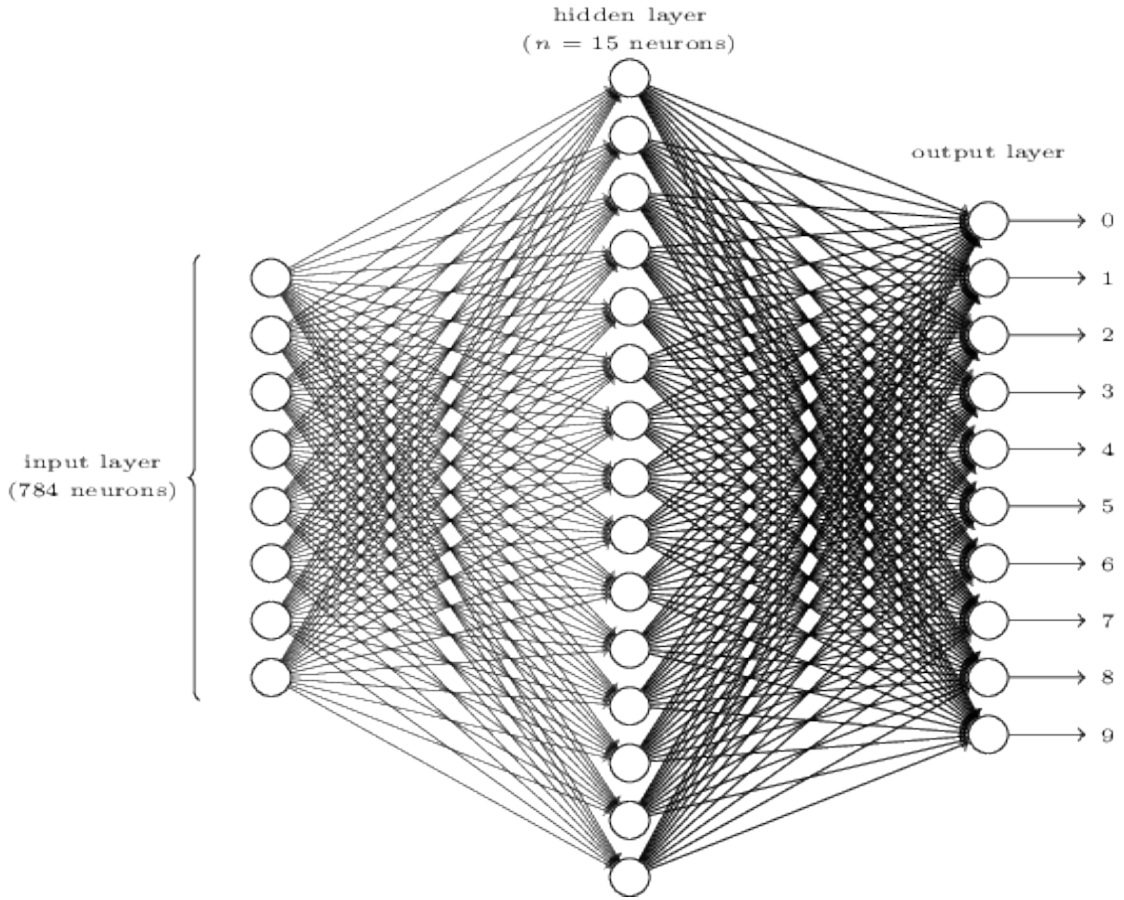
The **MNIST database** (*Modified National Institute of Standards and Technology database*) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.[4][5]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

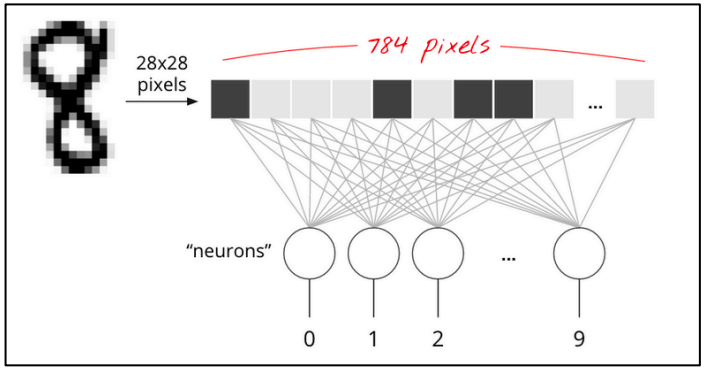


MNIST Example (28x28 pixels)









	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	55	148	210	253	253	113	87	148	55	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	87	232	252	253	189	210	252	252	253	168	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	4	57	242	252	190	65	5	12	182	252	253	116	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	96	252	252	183	14	0	0	92	252	252	225	21	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	132	253	252	146	14	0	0	0	215	252	252	79	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	126	253	247	176	9	0	0	8	78	245	253	129	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	16	232	252	176	0	0	0	36	201	252	252	169	11	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	22	252	252	30	22	119	197	241	253	252	251	77	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	16	231	252	253	252	252	252	226	227	252	231	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	55	235	253	217	138	42	24	192	252	143	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	62	255	253	109	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	71	253	252	21	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	253	252	21	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	71	253	252	21	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	106	253	252	21	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	45	255	253	21	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	218	252	56	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	96	252	189	42	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	184	252	170	11	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	147	252	42	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

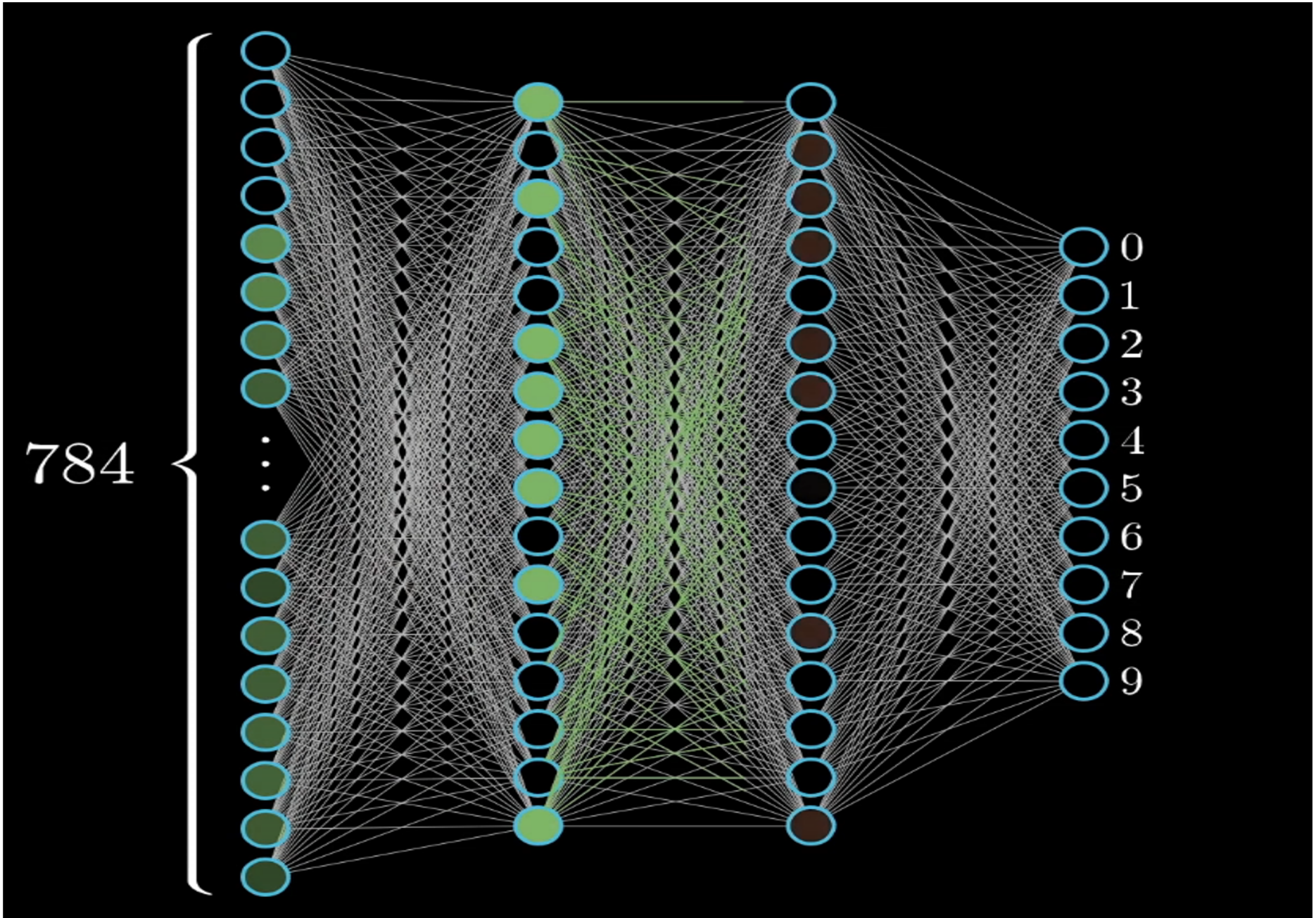
Grayscale image of 28 x 28 pixels

same as a 28 x 28 matrix

values of 0 - 255

Simple MLP Network

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='sdg', loss='SparseCategoricalCrossentropy',
metrics=['accuracy'])

model.summary()

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

Loss Function

Compiles the model with the SGD optimizer

Print summary

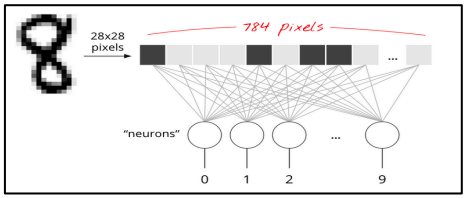
Use tensorborad

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

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

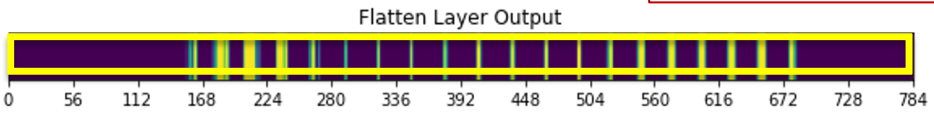
Adjusts model parameters to minimize the loss
Tests the model performance on a test set

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



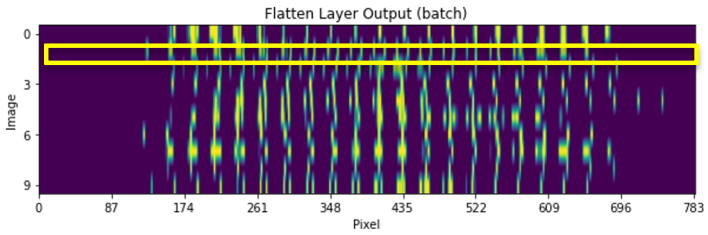
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

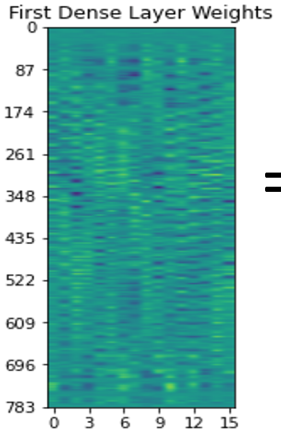
16 neurons
784 x 16 W matrix

10 x 16 output matrix

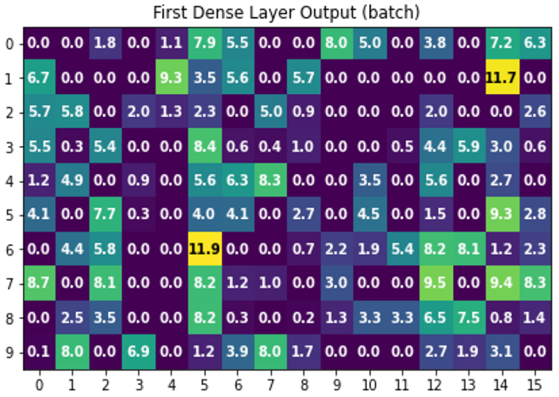
Input to 2nd layer



*

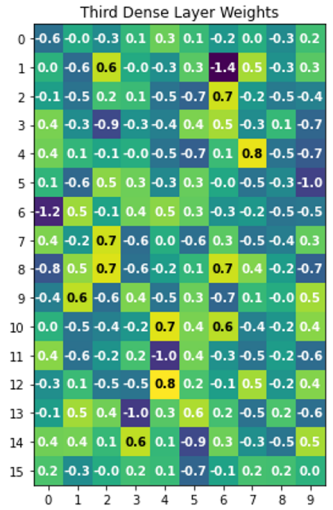


=



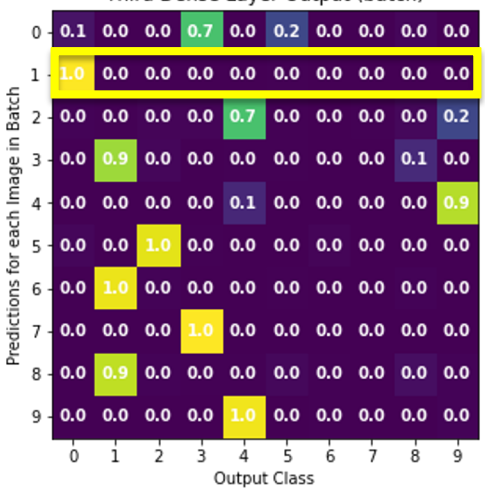
10 x 16 matrix

*

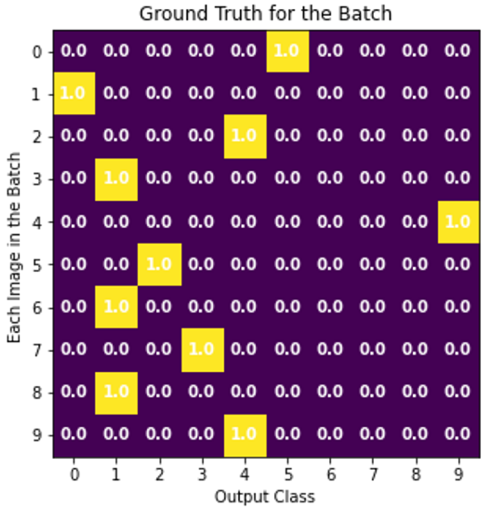


16 neurons
16 x 10
W matrix

=



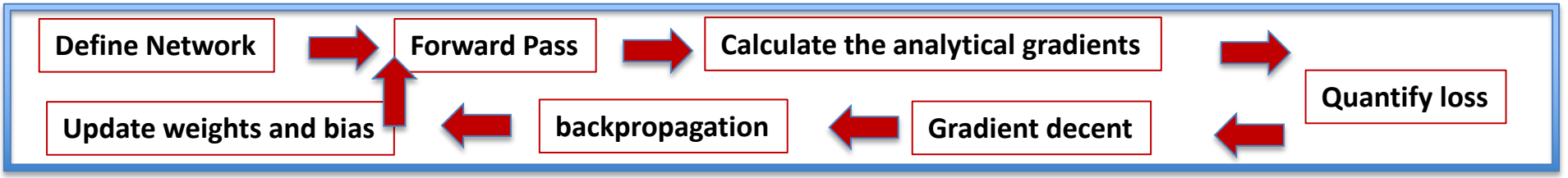
10 x 10 output matrix



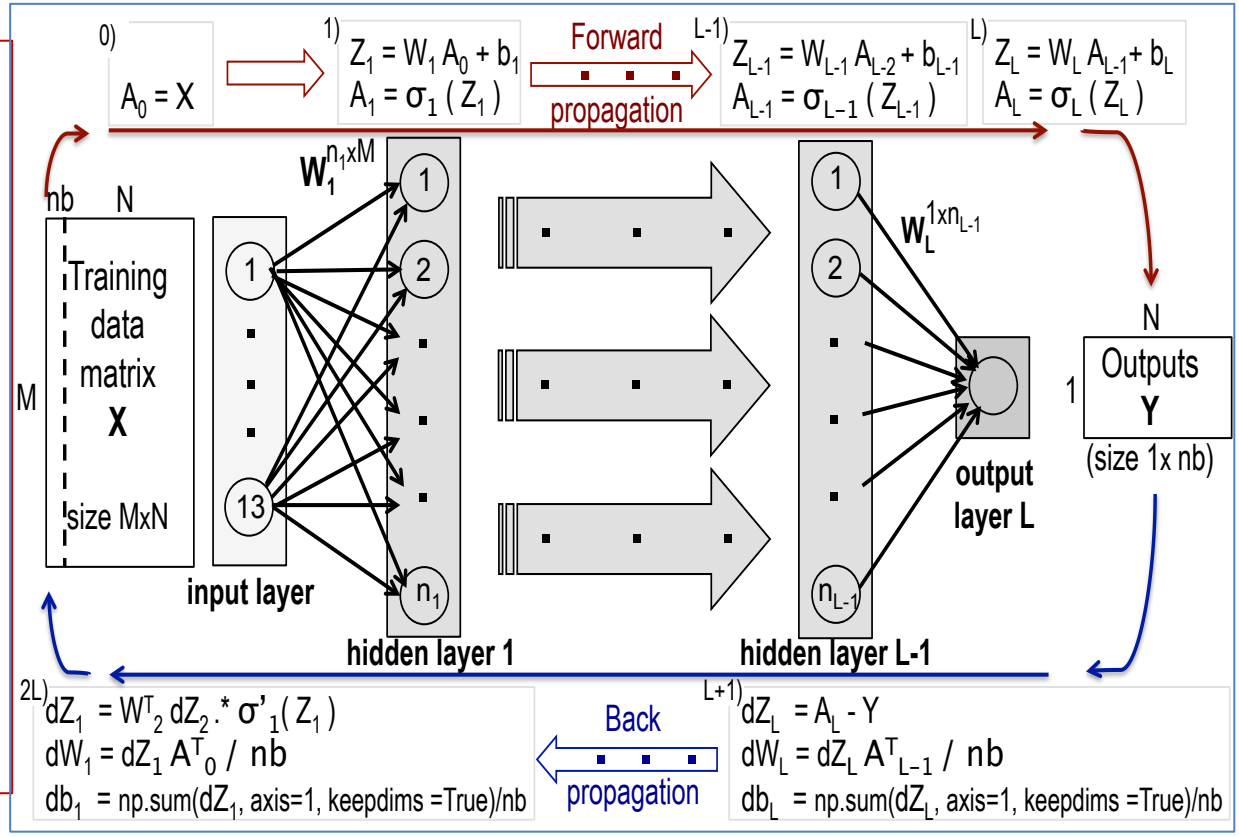
10 x 10 labeled matrix

Summary : Flow of NN

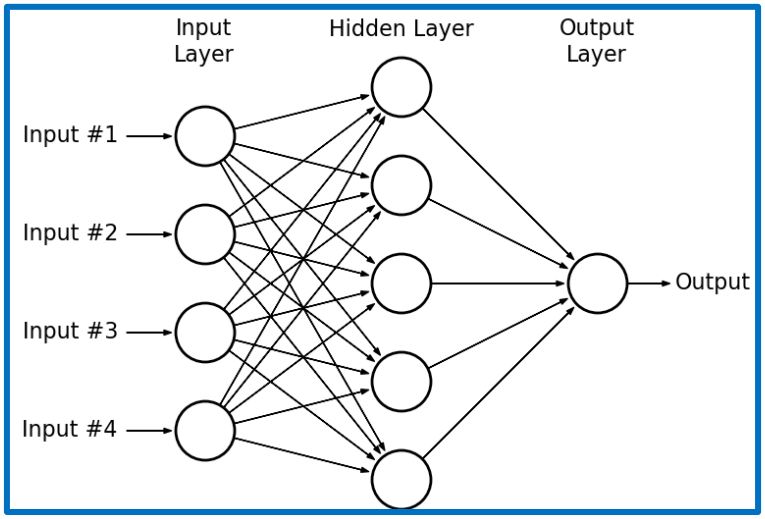
```
model.fit(x_train, y_train, epochs=3, batch_size=10)
```

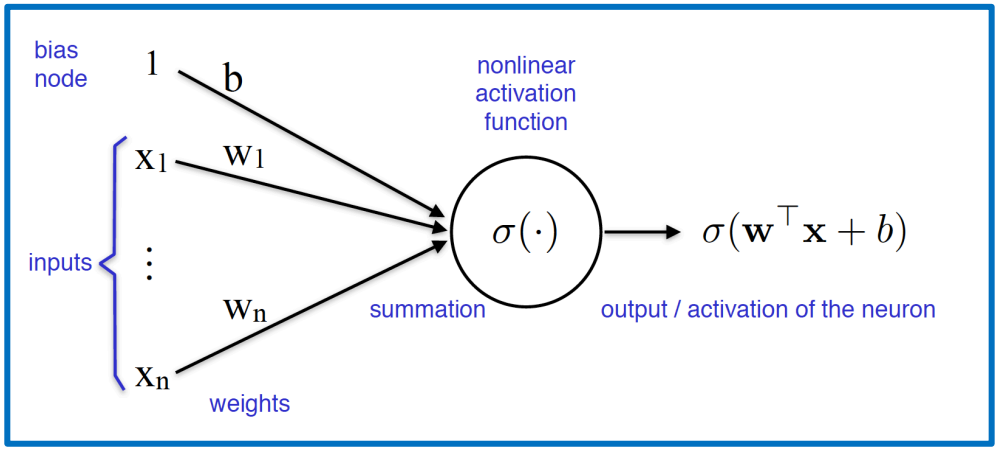



```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```



NN Modeling

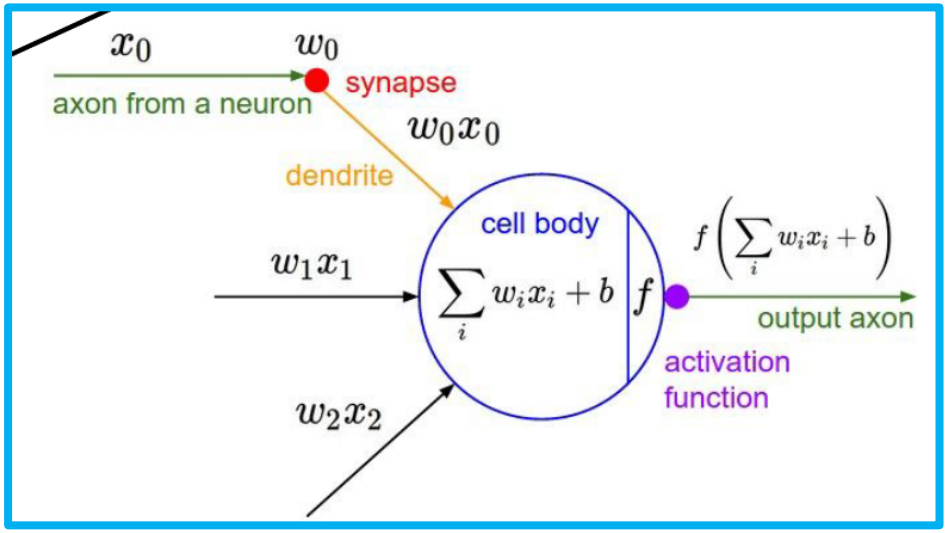




- ✓ A node in the neural network is a mathematical function or activation function which maps input to output values.
- ✓ Inputs represent a set of vectors containing weights (w) and bias (b). They are the sets of parameters to be determined.
- ✓ Many nodes form a **neural layer**, **links** connect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

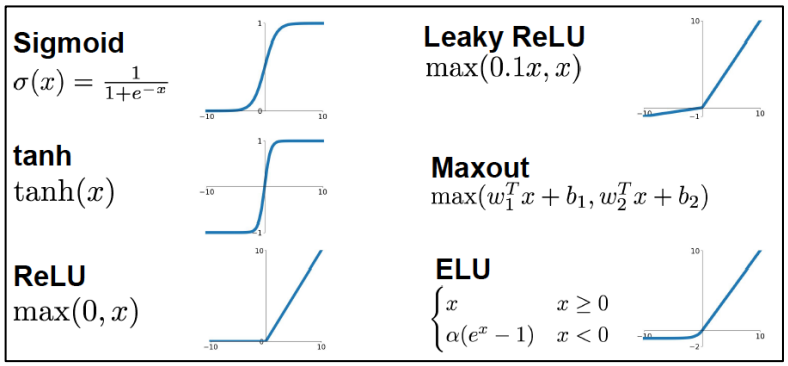
Perceptron

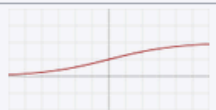
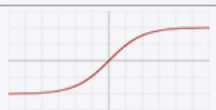
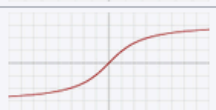
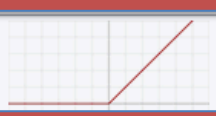
Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt. A perceptron takes several binary inputs, *x*1,*x*2,…, and produces a single binary output. By varying the weights and the threshold, we can get different models of decision-making.



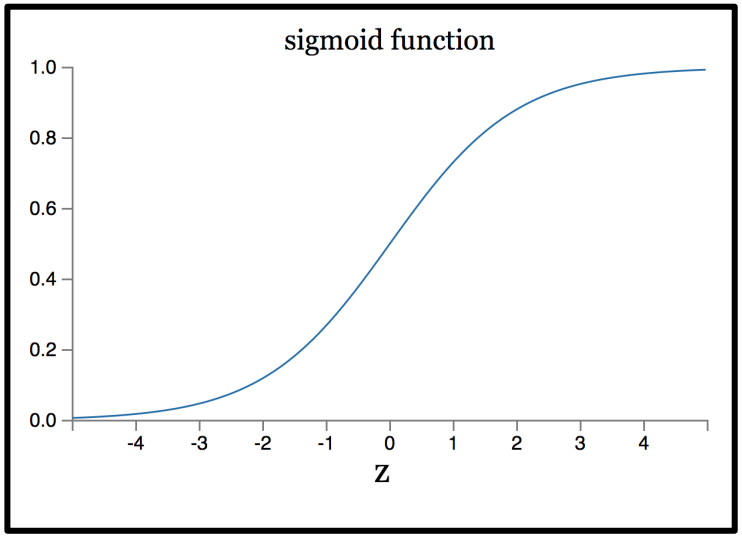
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

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

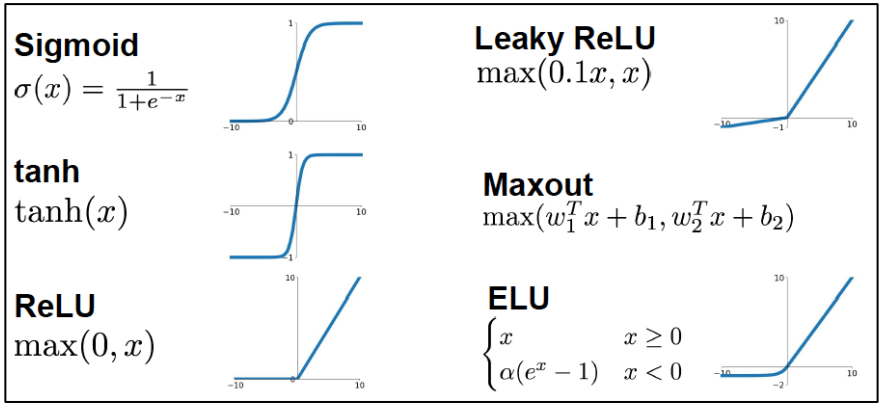


Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign [7][8]		$f(x) = \frac{x}{1+\lvert x\rvert}$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

Activation Function



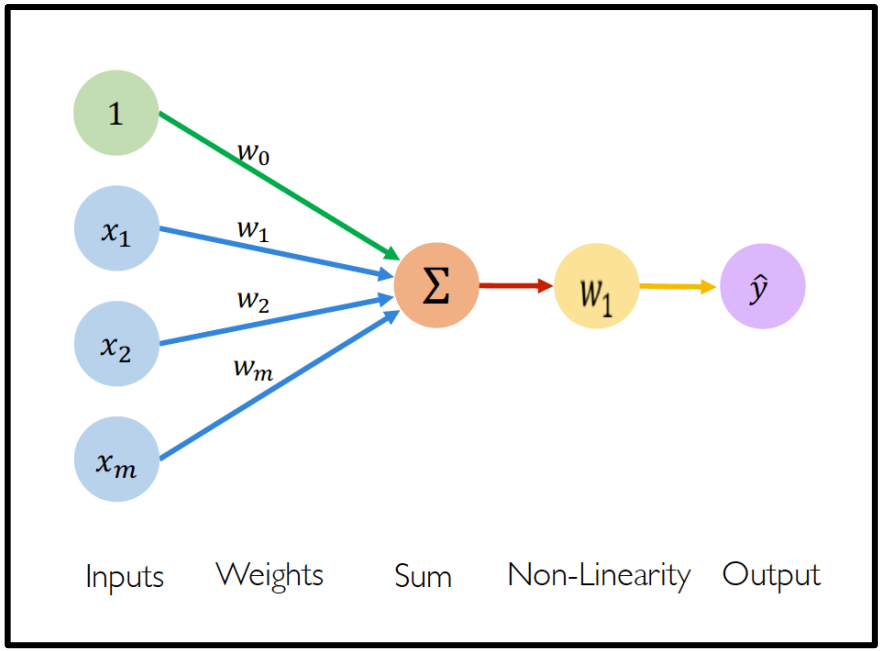
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$



- ✓ In much modern work on neural networks, the main neuron model used is one called the *sigmoid neuron*.
- ✓ Sigmond (logistic) function, σ, is a smooth out preceptron operating on a set of data, $\sigma(w\cdot x+b)$. The smoothness of σ means that small changes Δw in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron.
- ✓ If $z\equiv w\cdot x+b$ is a large positive number, then $exp(-z)\approx 0$ and so $\sigma(z)\approx 1$.
- ✓ If $z=w\cdot x+b$ is very negative, then $exp(-z)\rightarrow\infty$, and $\sigma(z)\approx 0$. So when $z=w\cdot x+b$ is very negative.
- ✓ It's only when $w\cdot x+b$ is of modest size that there's much deviation from the perceptron model.

neuralnetworksanddeeplearning.com

DNN MLP Forward Steps



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

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

Output

Linear combination of inputs

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

Non-linear activation function

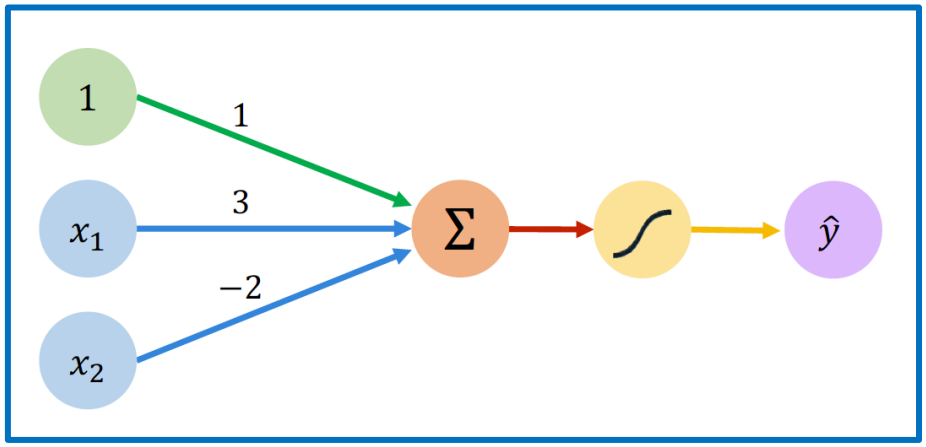
Bias

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

- Example: sigmoid function

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

Activation Function Example



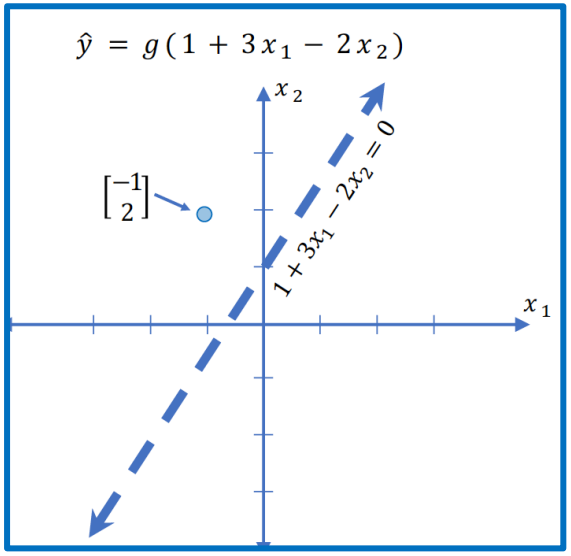
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

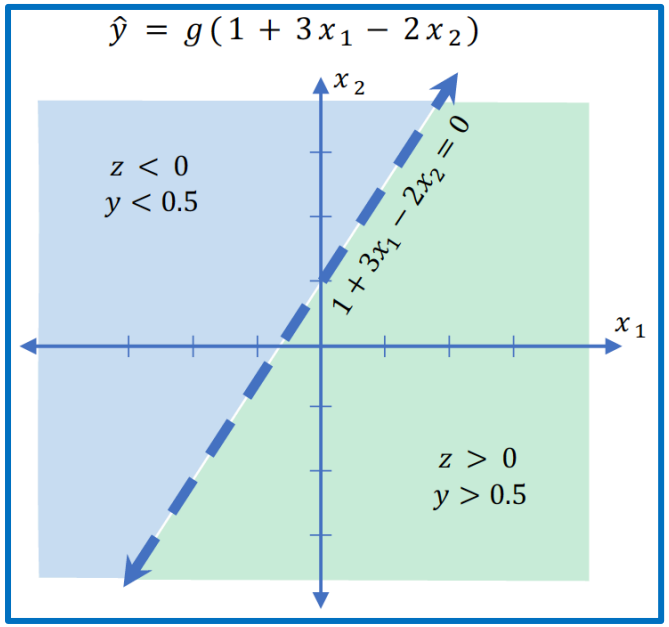
$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

This is just a line in 2D!

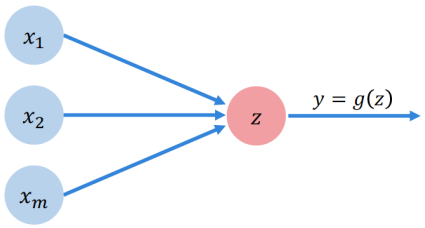
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$

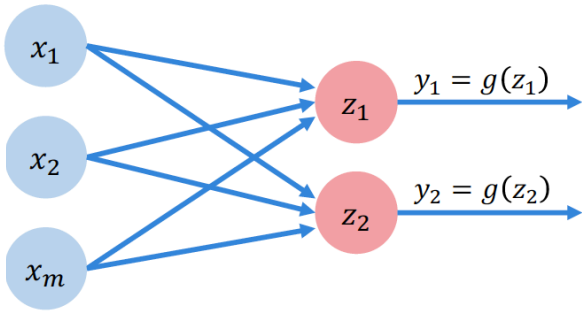




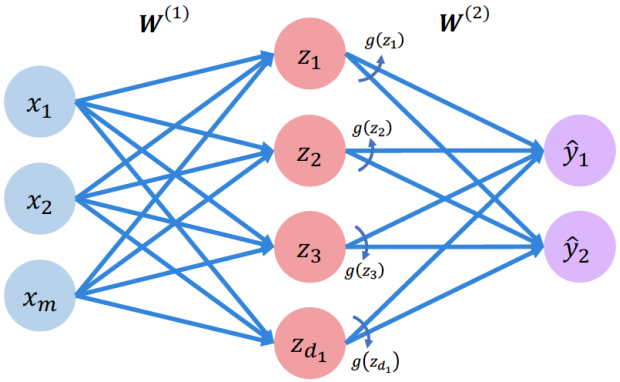
Multilayer Perceptron : Forward Path Calculation



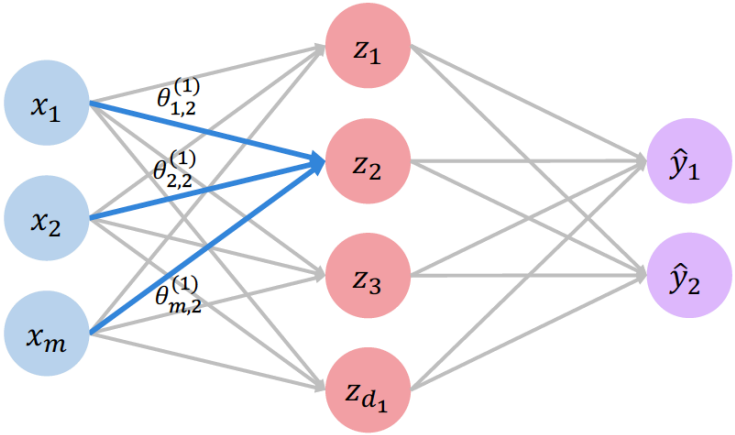
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$



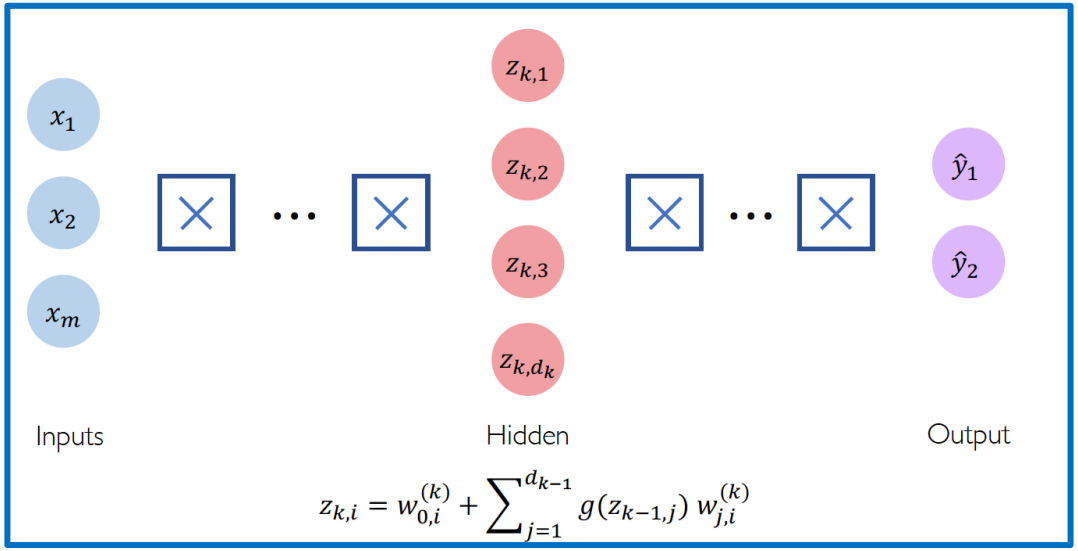
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$



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



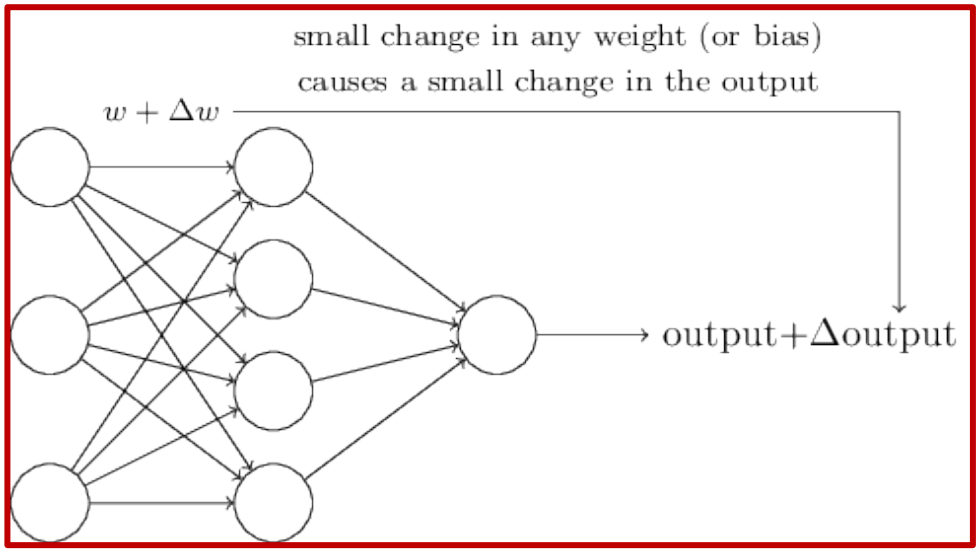
$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j w_{j,2}^{(1)}$$
$$= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)}$$

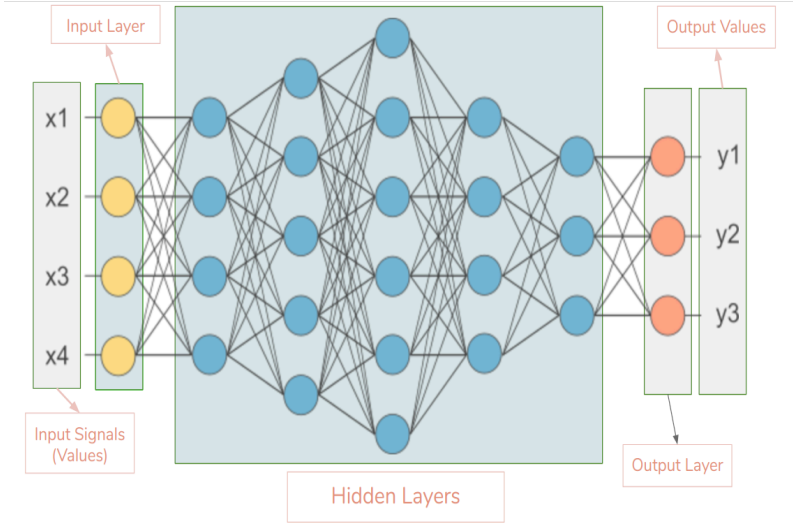


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Representation of NN Model

neuralnetworksanddeeplearning.com





✓ Δoutput is approximated using a *linear function* of the changes Δ*wj* (weights) and Δ*b (bias)*. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output.

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

output= cost function

✓ We then cast the model of the NN model as an optimization problem, minimizing the changes of the cost as functions of changes of weight and bias

Minimization : Iteration Scheme

neuralnetworksanddeeplearning.com

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

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

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

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

Be negative =

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

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

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

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

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

Be negative Choose

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

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

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

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

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

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

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 ∇C we need to compute the gradients ∇Cx separately for each training input, *x*, and then average them, $\nabla C = 1/n \sum \nabla Cx$. Unfortunately, when the number of training inputs is very large this can take a long time.
- ✓ A way is to use mini-batched ***stochastic gradient descent*** to speed up learning. The idea is to estimate the gradient ∇C by computing a small sample of randomly chosen training inputs, refer to as a ***mini-batch*** *of input* (mini-batched SGD).
- ✓ By averaging over a small sample, we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning, provided the sample size *m* is large enough that the average value of the ∇CXj roughly equals to the average over all ∇Cx.
- ✓ Then, another randomly chosen mini-batch are selected and trained, until all the training inputs are used. It is said to complete an *epoch* (iteration) of training.

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

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

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

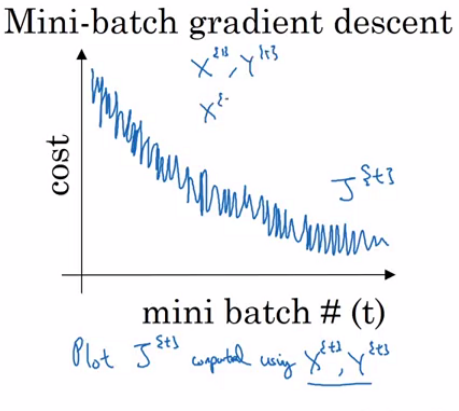
Optimization: Mini-Batch SGD

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)

In one epoch

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



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

DNN Forward Backward Calculation : Weights and bias

Algorithm

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

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

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

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

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

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

5. Return weights

In one epoch

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

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

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

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

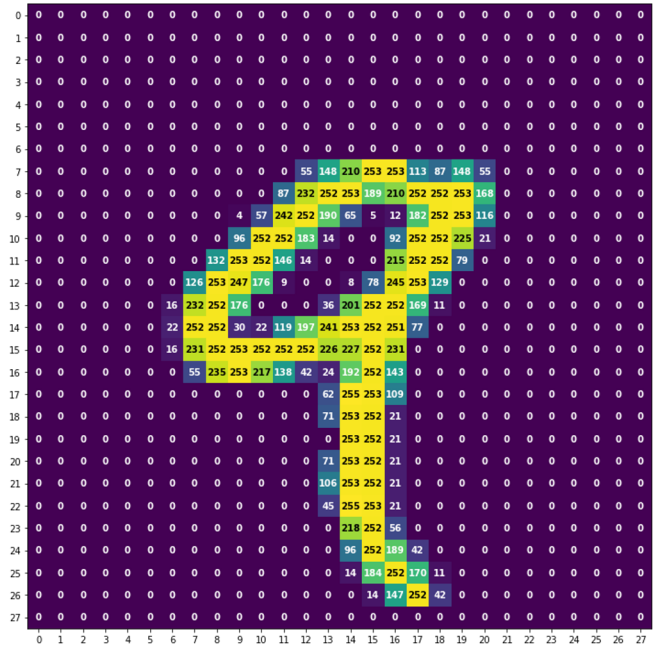
TensorFlow 2.0

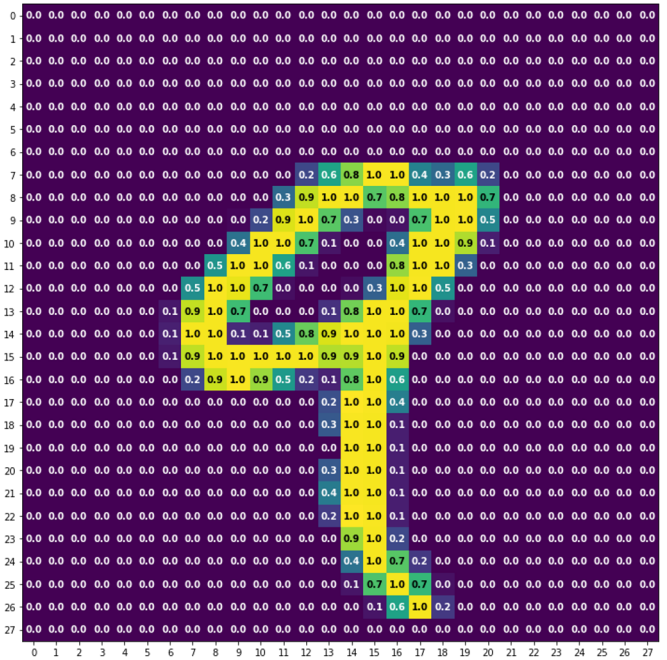
Loading and Preparing the MNIST Dataset

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

Each Image is a 28x28 image of a handwritten digit

x 60,000 images





TensorFlow 2.0 : NN Network

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

```
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')
])
model.compile(optimizer='sgd',
              loss='Catagorical_Crossentropy,
              metrics=['accuracy'])
Model.summary
model.fit(x_train, y_train, epochs=3, batch_size=10)
```

Simple MLP Network

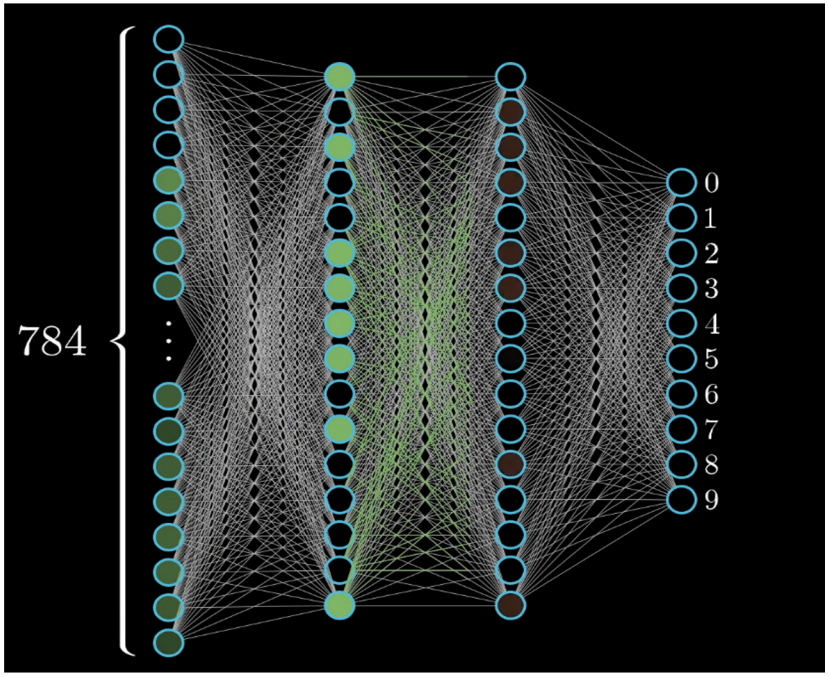
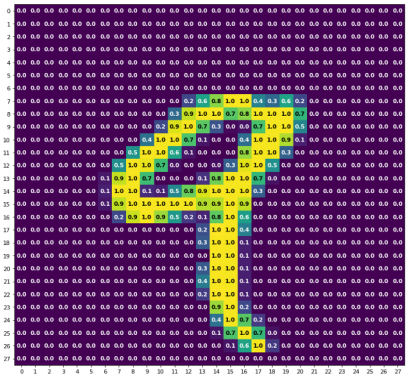


Image:
https://www.3blue1brown.com/

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

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

TensorFlow 2.0 : Forward Pass



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

Flatten a single image

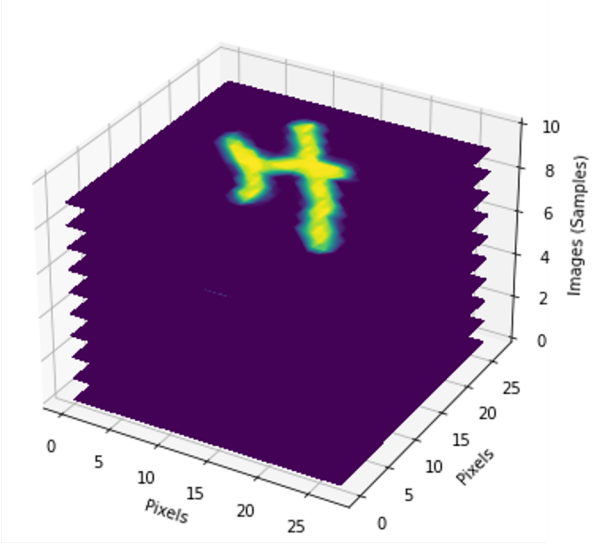
```
tf.reshape(tensor, shape)
#example
tf.reshape(x_train[0:10], [10: 784])
```

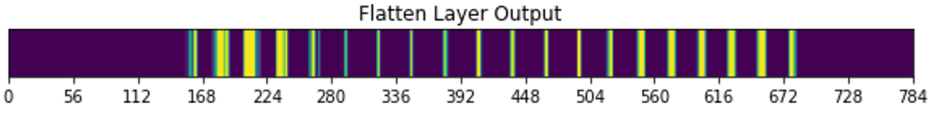
Flatten a batch of images

Flatten

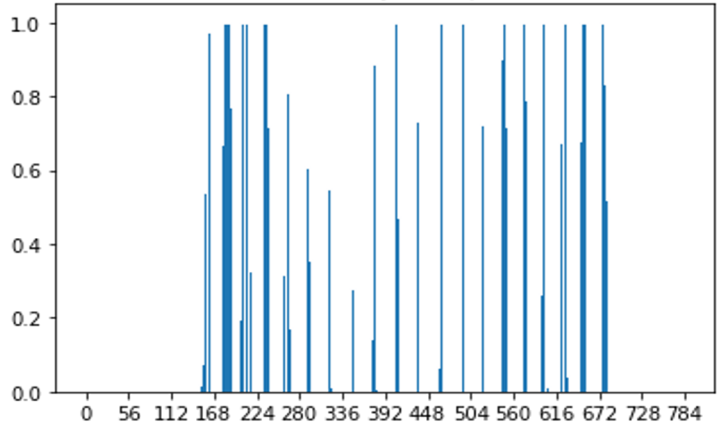
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28))
```

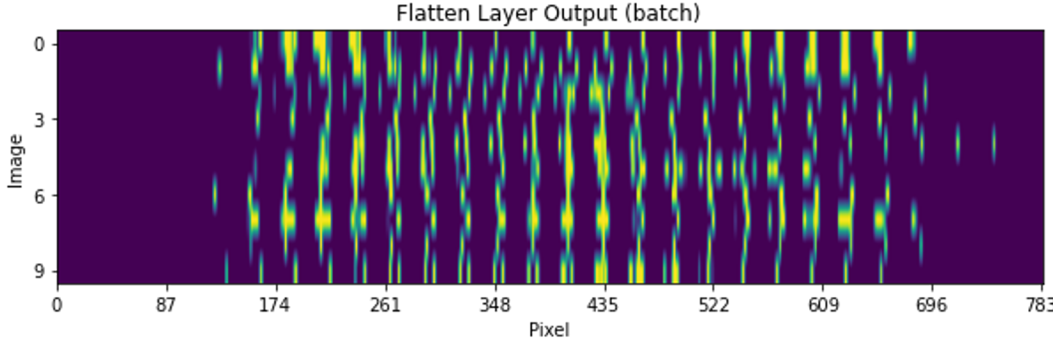
Network Input Per Iteration





=

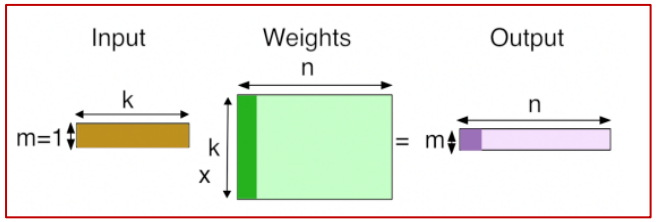




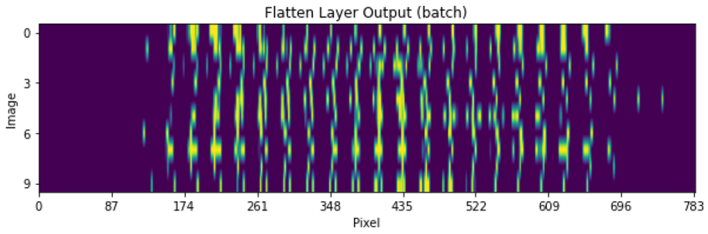
TensorFlow 2.0 : Forward Pass

1ST Fully Connected Layer with 16 neurons
Matrix multiplication

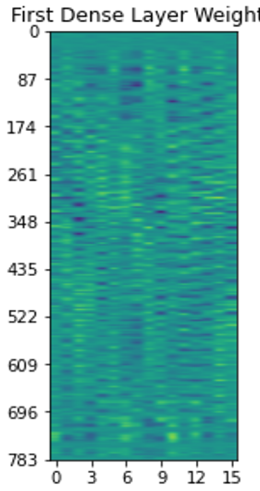
```
...
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(16, activation='relu'),
...
```



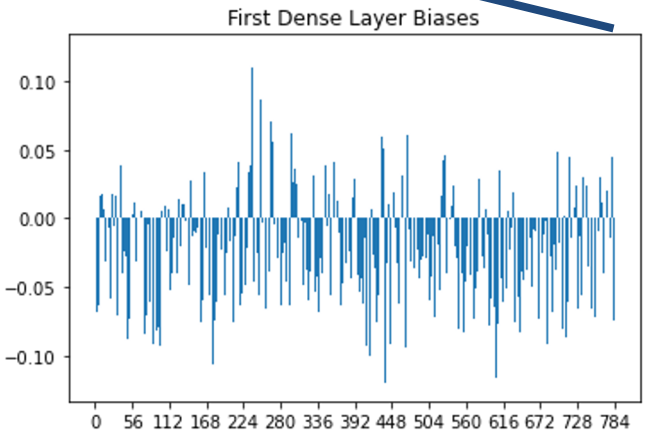
$$a^1 = \text{reLU}(X * W^1 + b^1)$$



*



+



Weights are initialized with random values

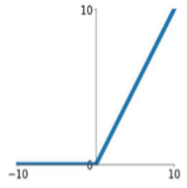
TensorFlow 2.0 : Forward Pass

Second Fully Connected Layer with 16 neurons

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

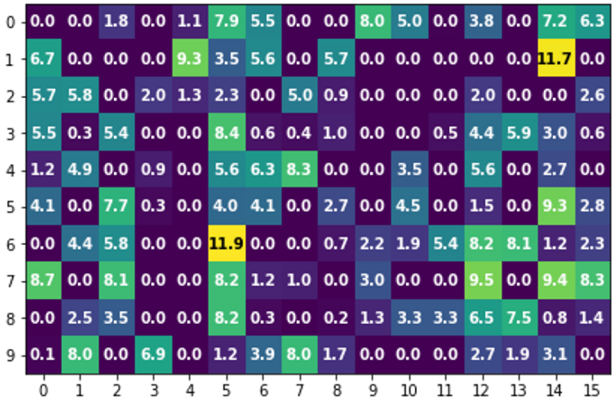
ReLU

$\max(0, x)$



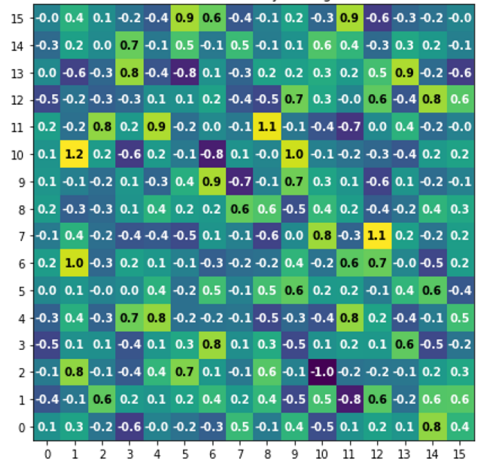
$a^2 = \text{reLU}(a^1 * W^2 + b^2)$

First Dense Layer Output (batch)



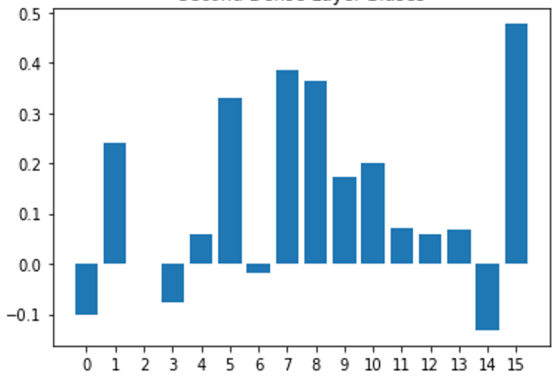
*

Second Dense Layer Weights



+

Second Dense Layer Biases



TensorFlow 2.0 : Forward Pass

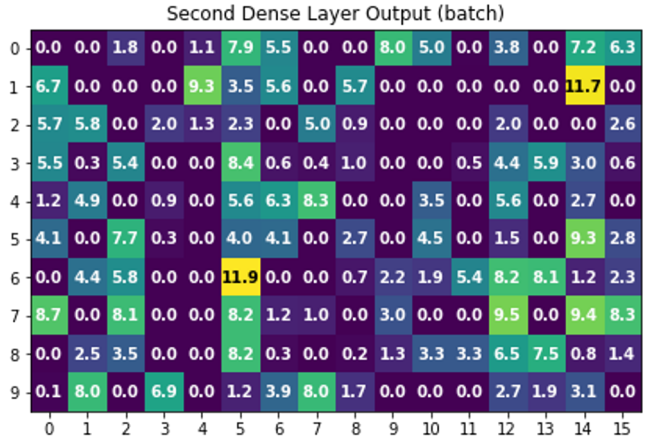
Final Fully Connected Layer with 10 neurons : 10 classes

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

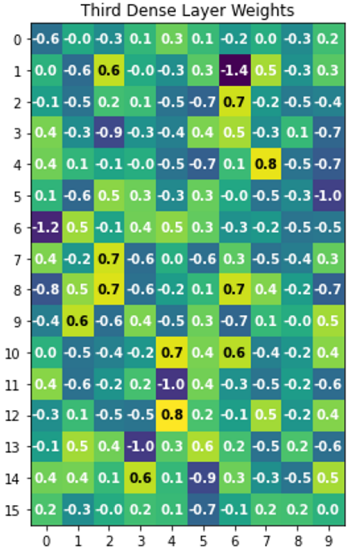
- ✓ The softmax function is a function that turns a vector of K real values into values between 0 and 1, and the values sum up to 1, so that they can be interpreted as probabilities.
- ✓ This is because the softmax is a generalization of logistic regression that can be used for multi-class classification.

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

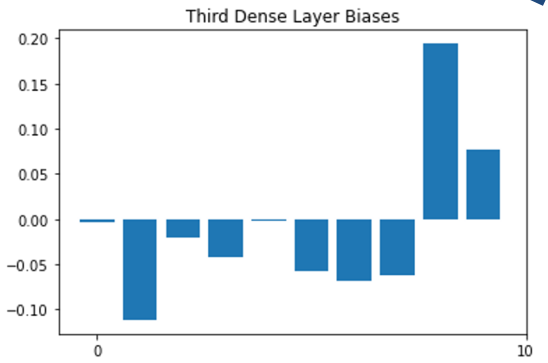
$a^3 = \text{softmax}(a^2 * W^3 + b^3)$



*



+



TensorFlow 2.0 : Forward Pass

`tf.losses.mean_squared_error(y_true_tf, y_pred_tf)`

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

Evaluating the Error

Softmax Classifier (Multinomial Logistic Regression)

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

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

Probabilities must be >= 0

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

Softmax Function

Probabilities must sum to 1

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

cat	3.2	exp	24.5	normalize	0.13	compare 1.00
car	5.1		164.0		0.87	0.00
frog	-1.7		0.18		0.00	0.00

Unnormalized log-probabilities / logits

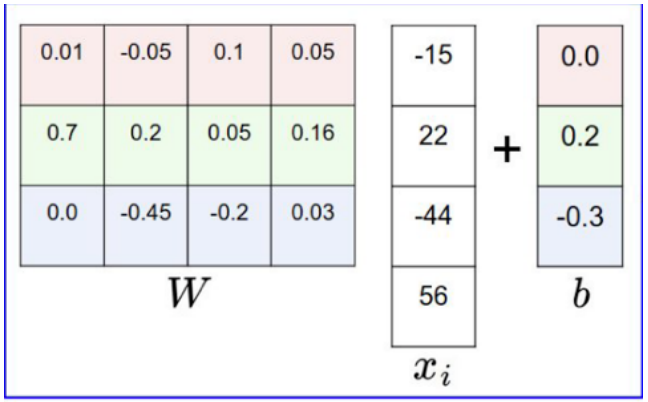
unnormalized probabilities

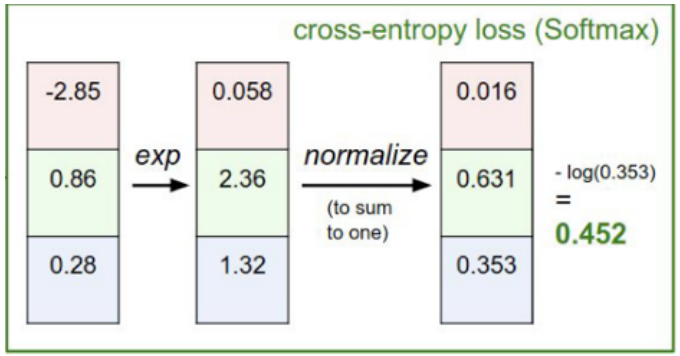
probabilities

Cross Entropy

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

Correct probs



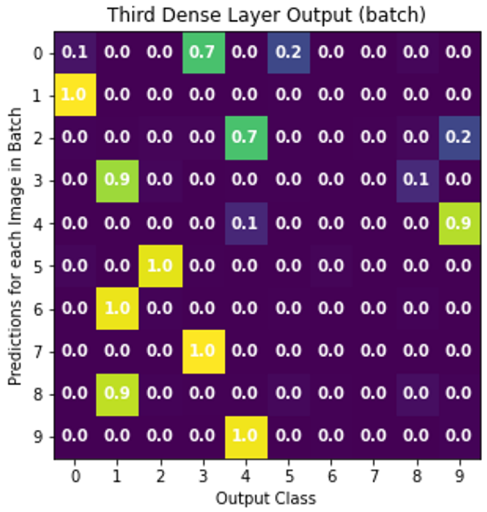


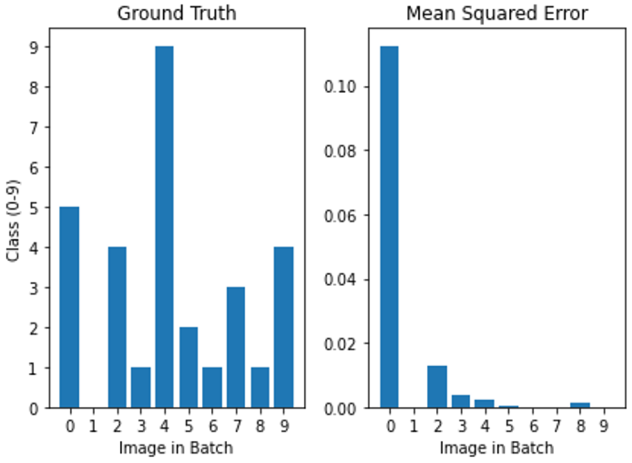
TensorFlow 2.0 : Forward Pass

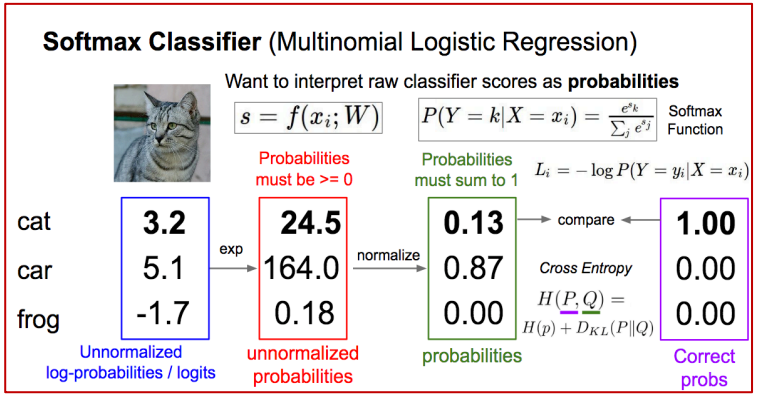
tf.losses.mean_squared_error(y_true_tf, y_pred_tf)

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

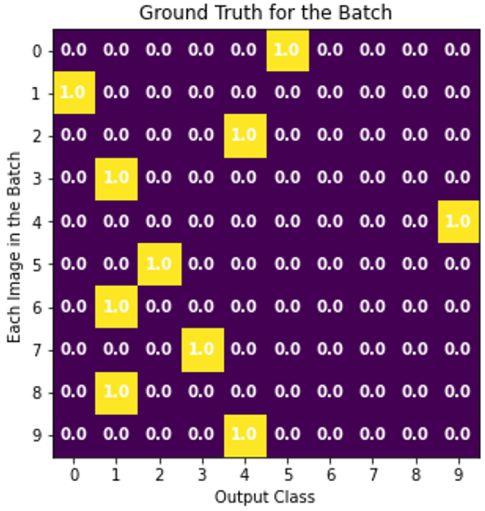
Evaluating the Error

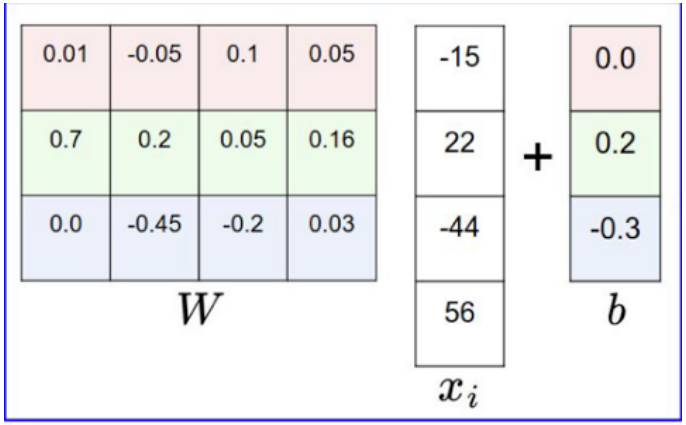


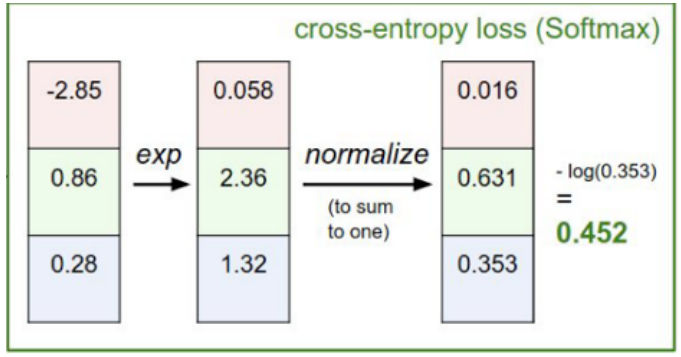




$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

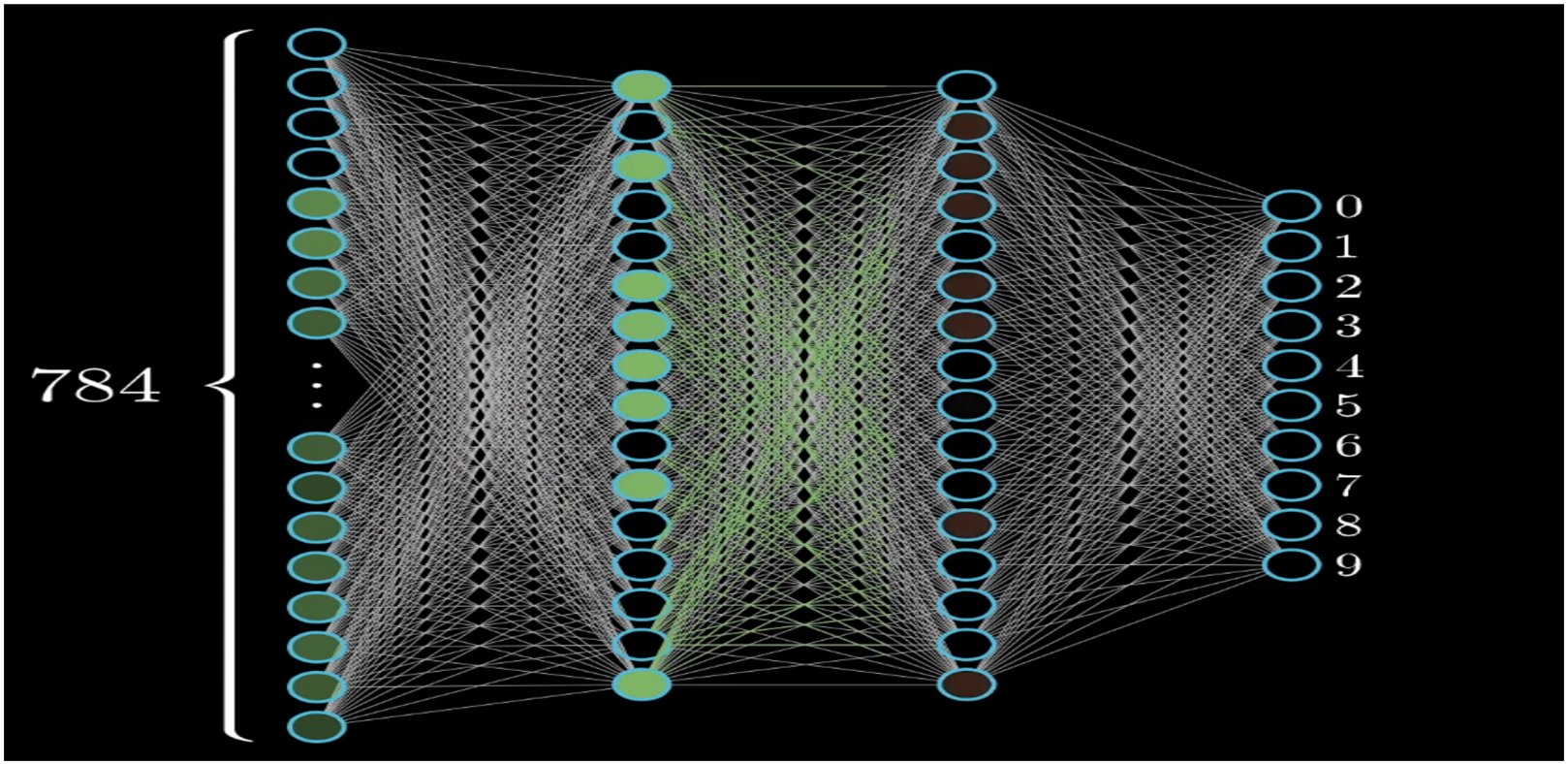






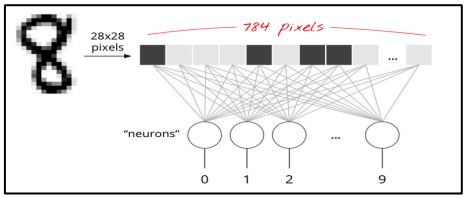
Simple MNIST MLP Network

Google Colab Code

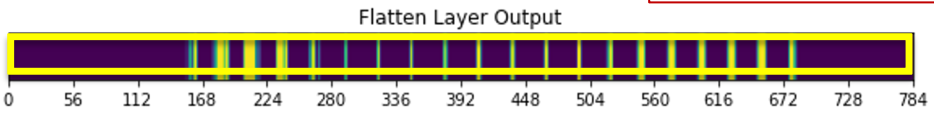


Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



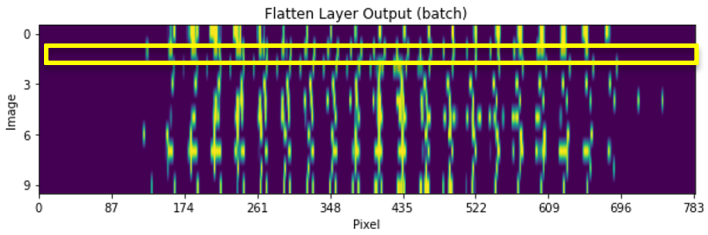
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

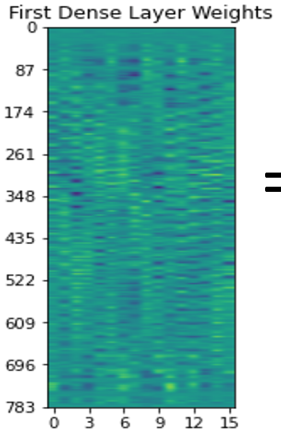
16 neurons
784 x 16 W matrix

10 x 16 output matrix

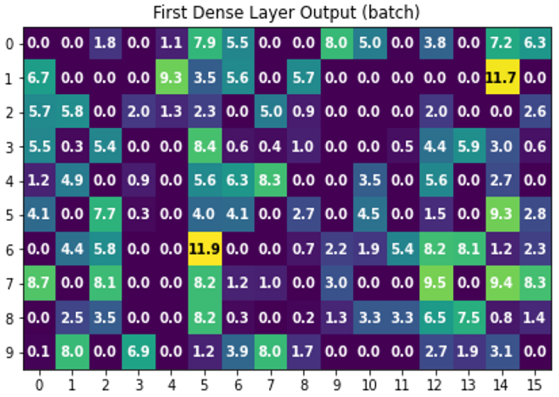
Input to 2nd layer



*

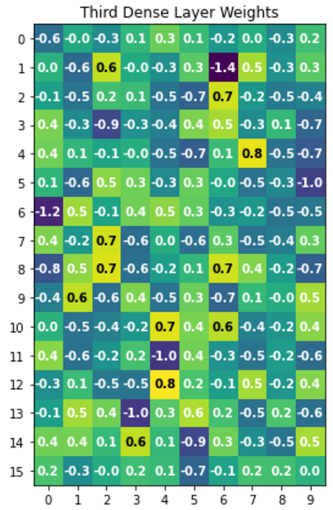


=



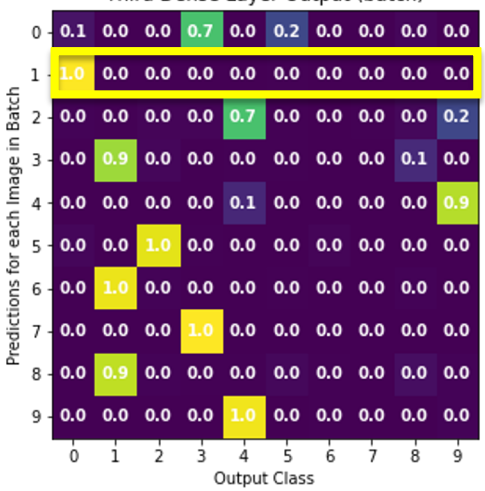
10 x 16 matrix

*

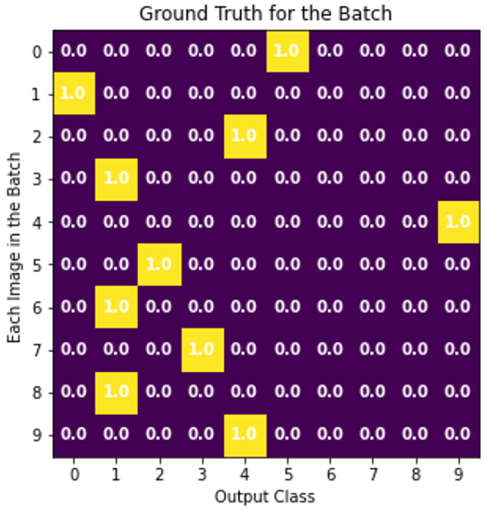


16 neurons
16 x 10
W matrix

=



10 x 10 output matrix



10 x 10 labeled matrix

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/

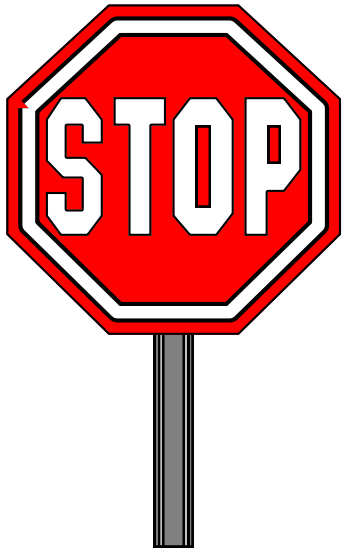
10) https://oneapi-src.github.io/oneDNN/

11) http://www.cs.cornell.edu/courses/cs4787/2020sp/

12) More sites and free books are listed in www.jics.utk.edu/actia → ML

13) https://machinelearningmastery.com

The End



- The End!