LAPENNA Program
LECTURE 4

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

University of Tennessee, Knoxville

February 24, 2022

Acknowledgements:

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

INNOVATIVE COMPUTING LABORATORY

Overview of LAPENNA

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

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

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

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

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

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

Teams and Participants of LAPENNA

- ✓ **PIs : Kwai Wong (JICS), Stan Tomov (ICL), University of Tennessee, Knoxville**
  - **Stephen Qiu, Julian Halloy, Eric Zhao (Students)**

- ✓ **Team : Clemson University**
- ✓ **Teams : University of Arkansas**
- ✓ **Team : University of Houston, Clear Lake**
- ✓ **Team : Miami University, Ohio**
- ✓ **Team : Boston University**
- ✓ **Team : West Virginia University**
- ✓ **Team : Louisiana State University, Alexandria**
- ✓ **Teams : Jackson Laboratory**
- ✓ **Team : Georgia State University**
- ✓ **Teams : University of Tennessee, Knoxville**
- ✓ **Teams : Morehouse College, Atlanta**
- ✓ **Team : North Carolina A & T University**
- ✓ **Team : Clark Atlanta University, Atlanta**
- ✓ **Team : Alabama A & M University**
- ✓ **Team : Slippery Rock University**
- ✓ **Team : University of Maryland, Baltimore County**

Meeting Time

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

Topic: LAPENNA Spring 2022 Webinar
Time: Feb 3, 2022 08:00 PM Eastern Time (US and Canada)
Every week on Thu, 12 occurrence(s)
Feb 3, 2022 07:30 PM
Feb 10, 2022 07:30 PM
Feb 17, 2022 07:30 PM
Feb 24, 2022 07:30 PM
Mar 3, 2022 07:30 PM
Mar 10, 2022 07:30 PM
Mar 17, 2022 07:30 PM
Mar 24, 2022 07:30 PM
Mar 31, 2022 07:30 PM
Apr 7, 2022 07:30 PM
Apr 14, 2022 07:30 PM
Apr 21, 2022 07:30 PM

Join from PC, Mac, Linux, iOS or Android: https://tennessee.zoom.us/j/94140469394
Password: 708069

Schedule of LAPENNA Spring 2022
Thursday 8:00pm -10:00pm Eastern Time

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

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

Computational Ecosystem

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

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

DNN Model

Applications

+

Data Ensemble + Input

+

It's all about
linear algebra calculations

+

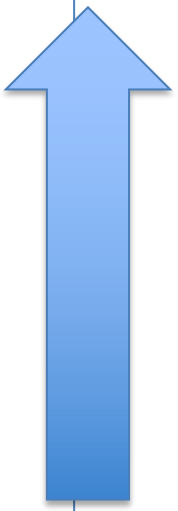
Algorithms, Software

+

Output + Data Analysis

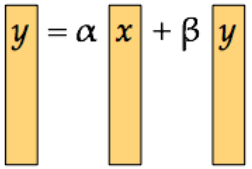
+

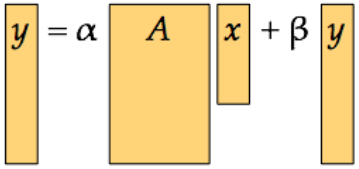
Hardware

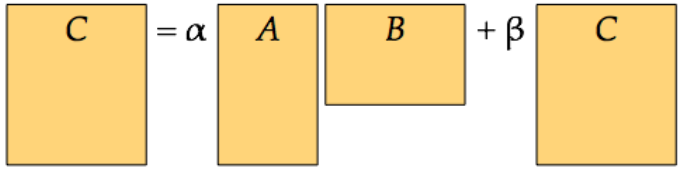


Computational Intensity = FLOPS/ Memory Access

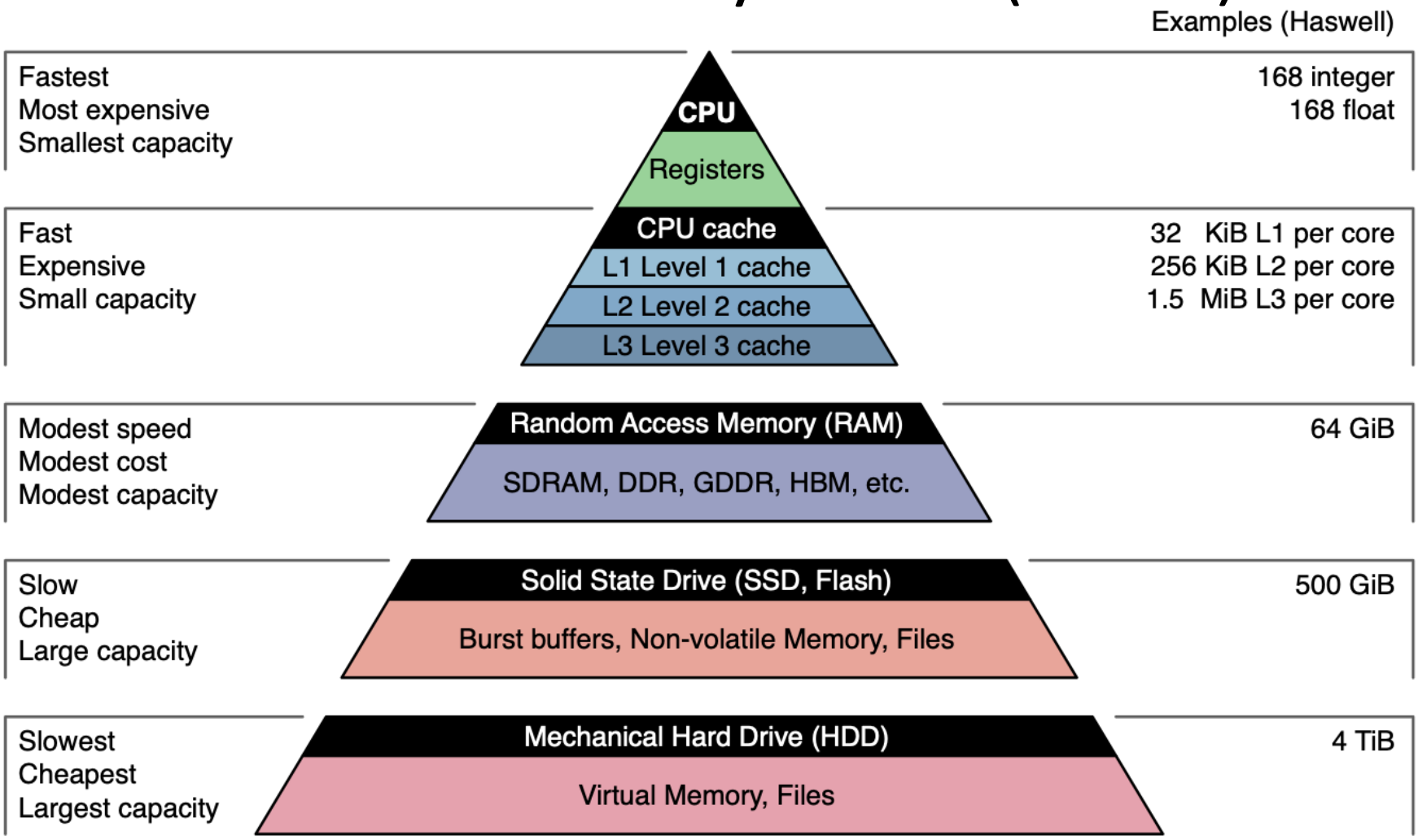
- ✓ Level 1 BLAS — vector operations
  - ✓ $O(n)$ data and flops (floating point operations)
  - ✓ Memory bound:
    $O(1)$ flops per memory access
- ✓ Level 2 BLAS — matrix-vector operations
  - ✓ $O(n^2)$ data and flops
  - ✓ Memory bound:
    $O(1)$ flops per memory access
- ✓ Level 3 BLAS — matrix-matrix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







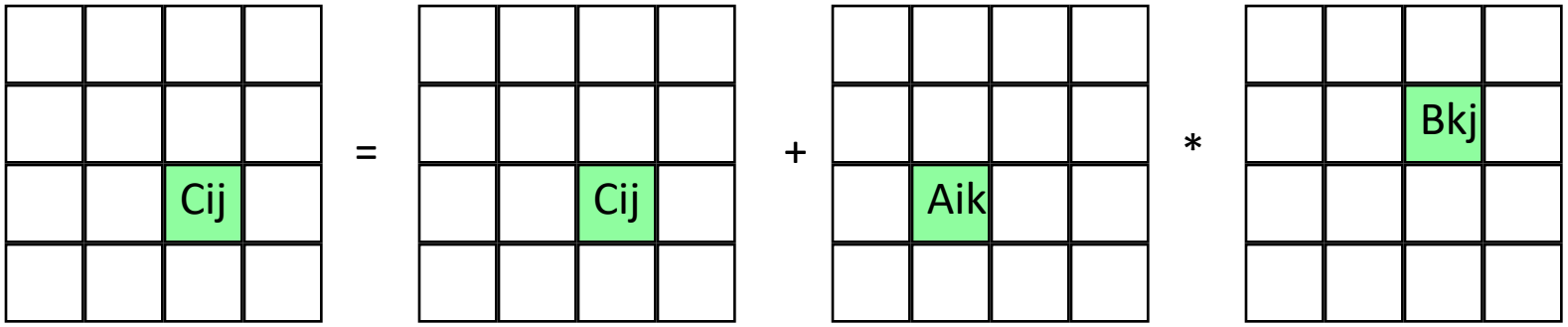
Memory hierarchy
It’s all about memory access (data!!)



Adapted from illustration by Ryan Leng

Block MM

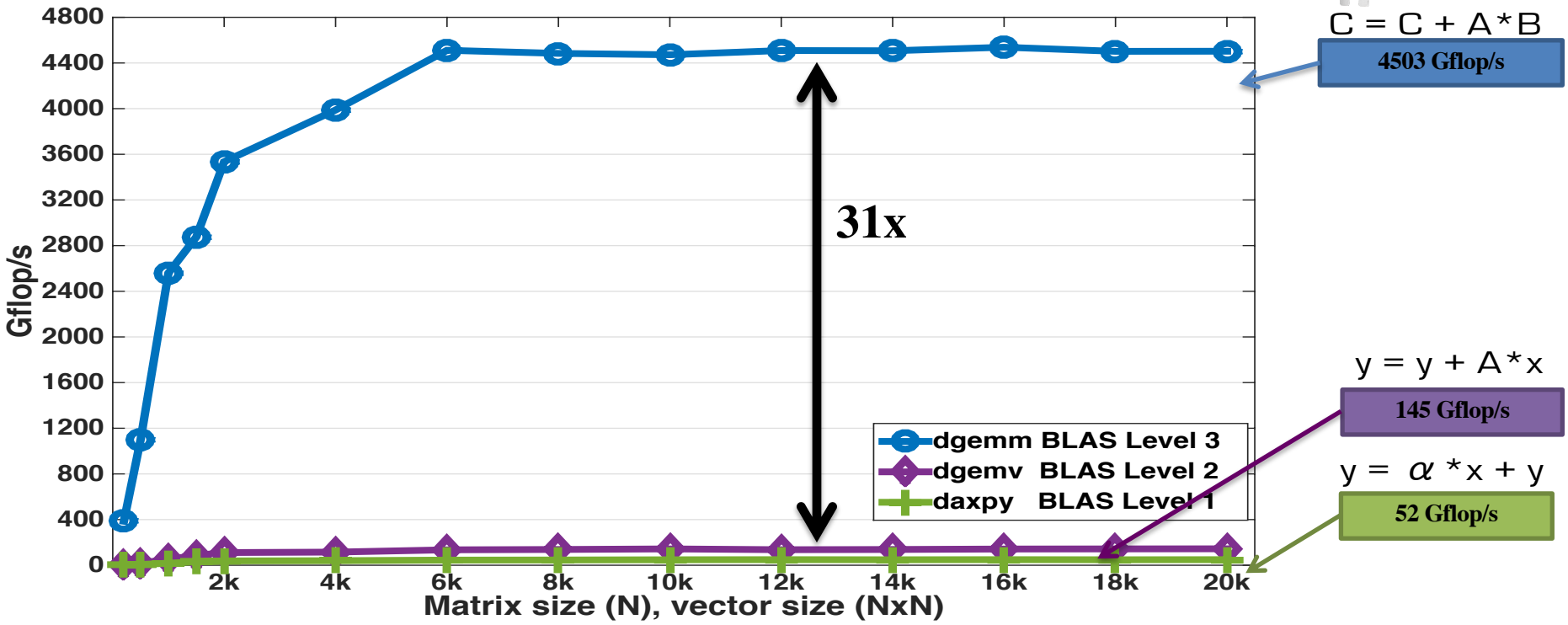
- q = f/m = (2*n^3) / ((2*N + 2) * n^2) ~ n / N
- If N is equal to 1, the algorithm is ideal. However, N is bounded by the amount of fast cache memory. However, N can be taken independently to the size of matrix, n.
- The optimal value of N = sqrt (size of fast memory / 3)



$$C_{ij} = C_{ij} + \sum_{k=1}^{n} A_{ik} * B_{kj}$$

Level 1, 2 and 3 BLAS

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



C = C + A*B

4503 Gflop/s

y = y + A*x

145 Gflop/s

y = α *x + y

52 Gflop/s

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

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

```
[18] import numpy as np

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

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

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

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

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

LAB 2
Problem 4

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

```
import numpy as np
import time

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

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

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

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

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

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

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

Topics of Unit Five

- ✓ **Unit 4 : Multilayer Perceptron**
  - ➢ **Overview, Activation function**
  - ➢ **Overview MLP, Forward path, example**
  - ➢ **computational graph**
  - ➢ **Backpropagation, example**
  - ➢ **Complete the cycle**

Supervised Learning

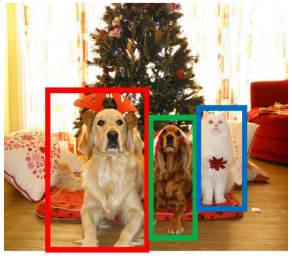
Supervised Learning Data:
(x, y) x is data, y is label

Goal:
Learn a function to map x -> y

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

CAT

Classification

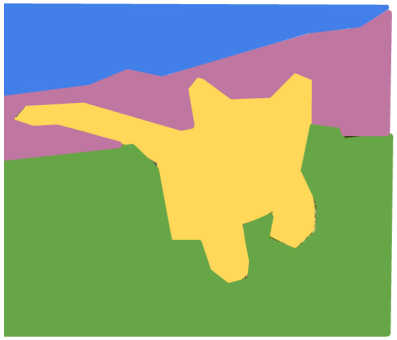


DOG, DOG, CAT

Object Detection

A cat sitting on a suitcase on the floor

Image captioning



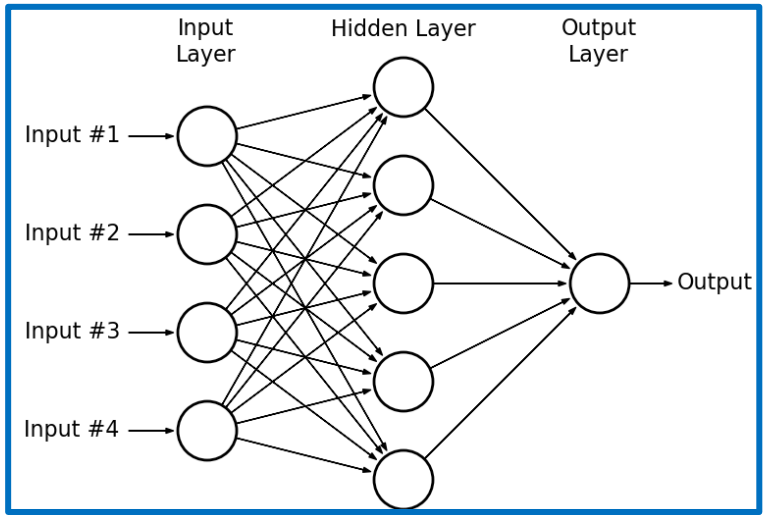
GRASS, CAT, TREE, SKY

Semantic Segmentation

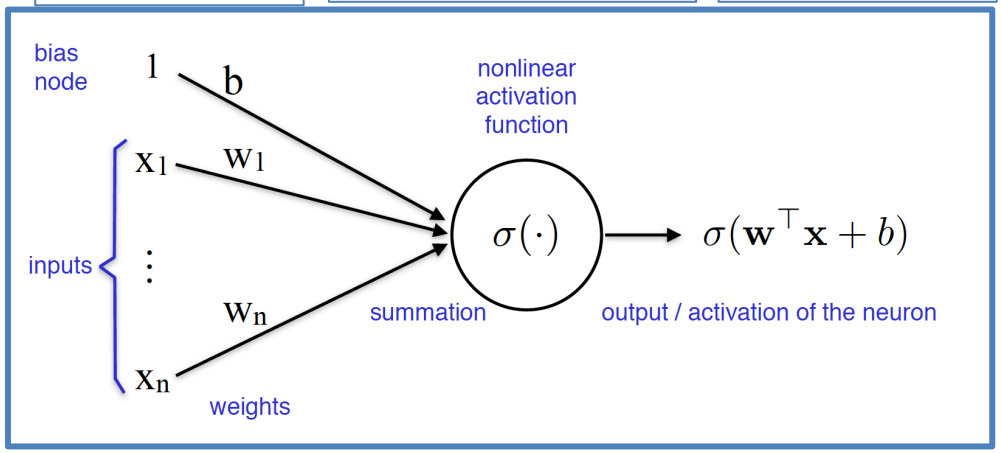
Neural Network Modeling

A Neural network is a mesh of links with connecting nodes

A node in a NN Model is a nonlinear activation function which maps an input vector to an output value

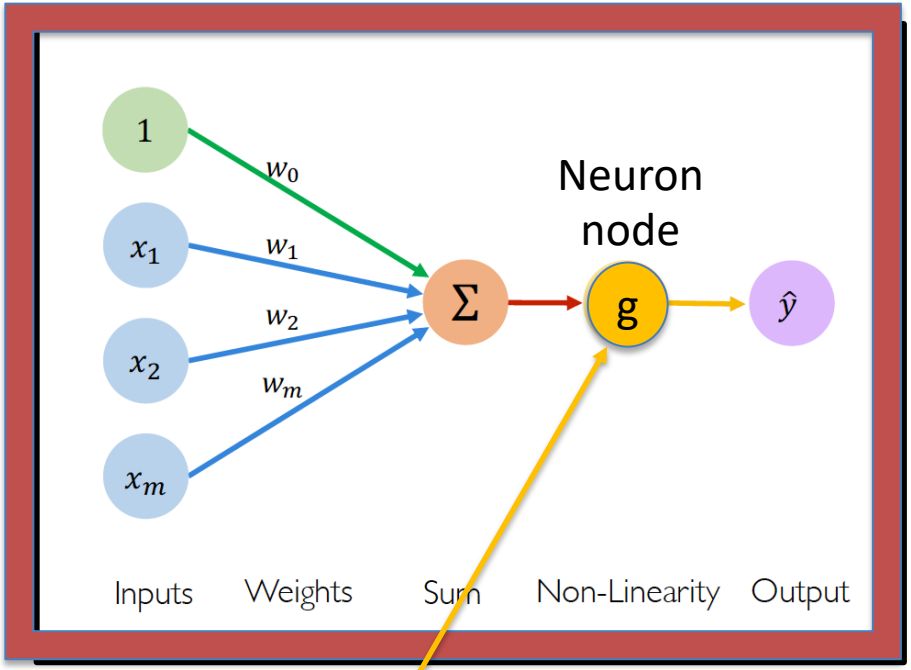


Input vectors | Nonlinear function | output values



- ✓ A neural network is a parametric model.
- ✓ Activation function (f or σ), is generally a nonlinear data operator which facilitates identification of complex features.
- ✓ Input vectors are composed of weights (w) and bias (b). They are the sets of parameters to be determined. Each link represents an unknow w. Each node has a bias. W and b are initiated “randomly“ to start. It‘s more efficient to use a batch of data vectors as input.
- ✓ Each batch of data vectors are chosen randomly from the training dataset. Computation continues with new batches of data vectors until everyone of them is used. (iteration).
- ✓ An epoch is defined as a “time step“ after every data vector is used in the training dataset.

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

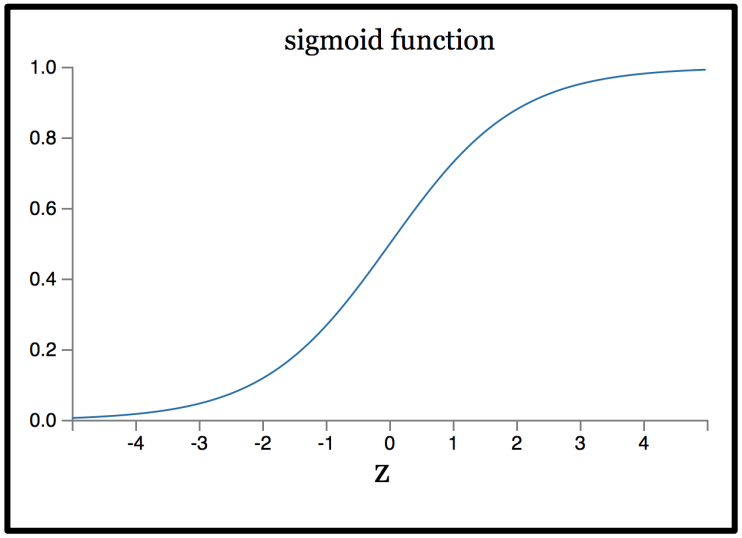
Bias

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

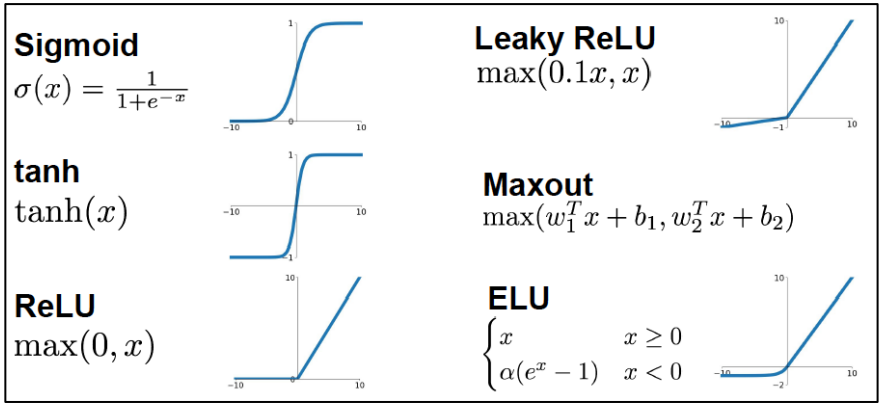
- Example: sigmoid function

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

Activation Function (neuron node function , g)



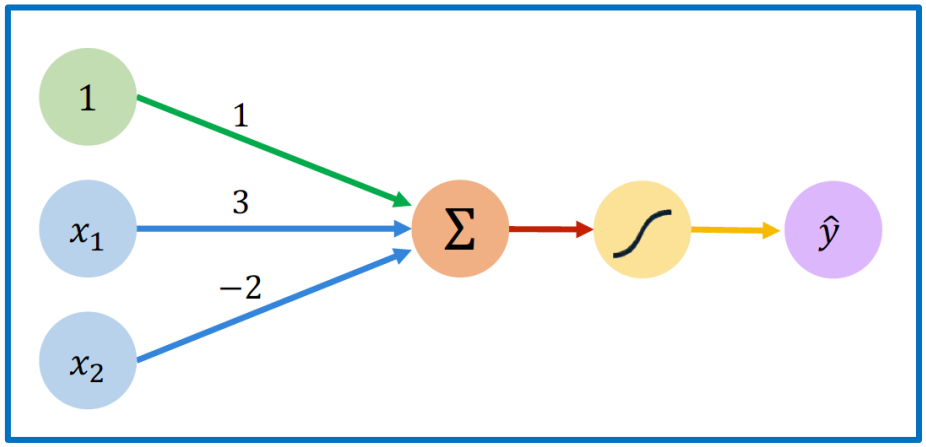
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$



- ✓ 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

Activation Function Example



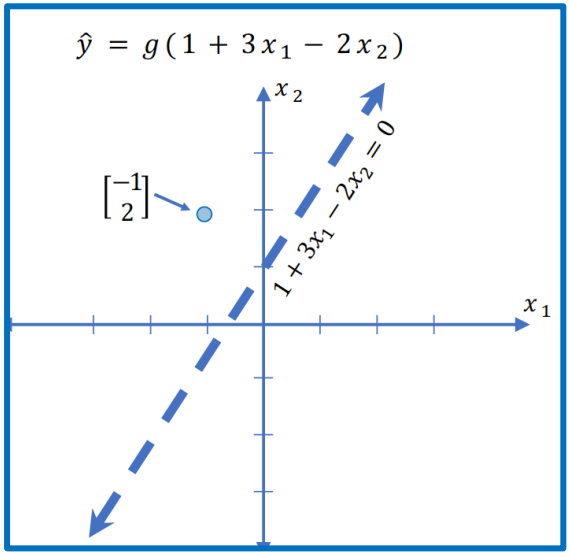
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

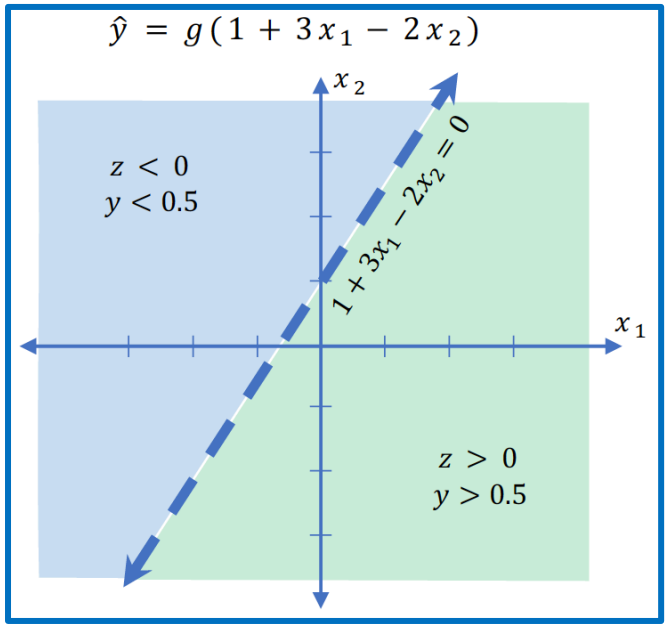
$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

This is just a line in 2D!

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

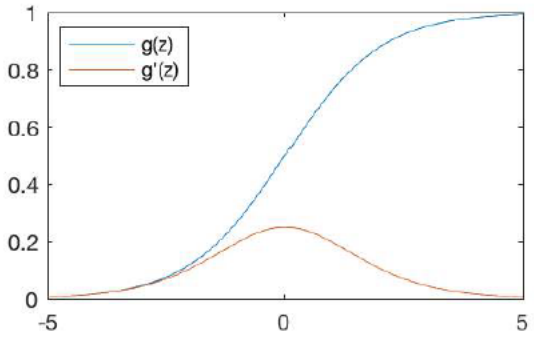
$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$





Nonlinear Activation Function

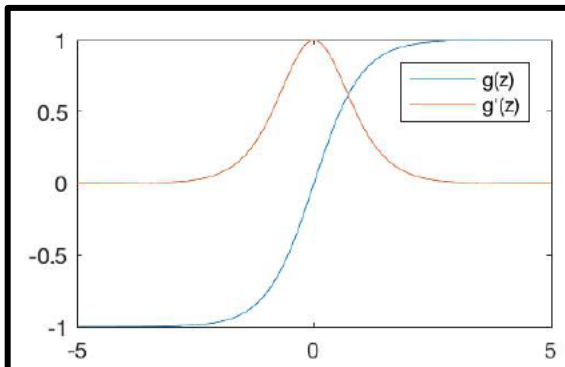
Sigmoid Function



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

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

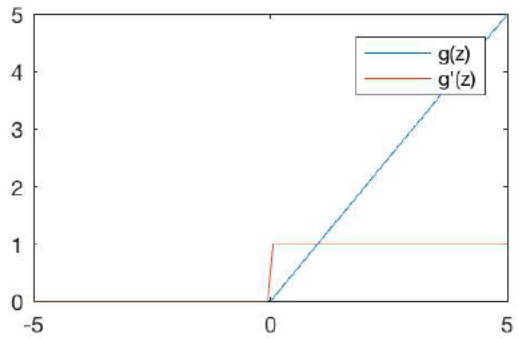
tf.nn.sigmoid(z)



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

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

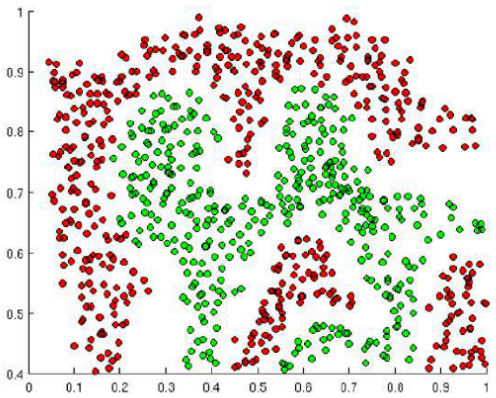
tf.nn.tanh(z)

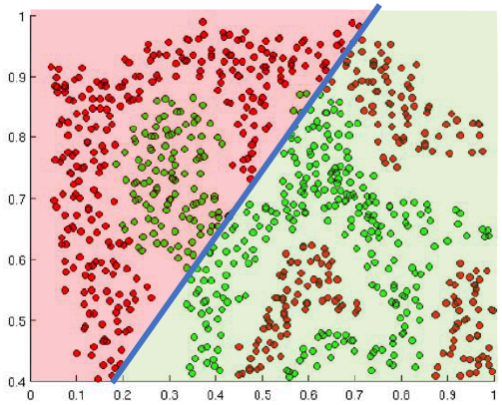


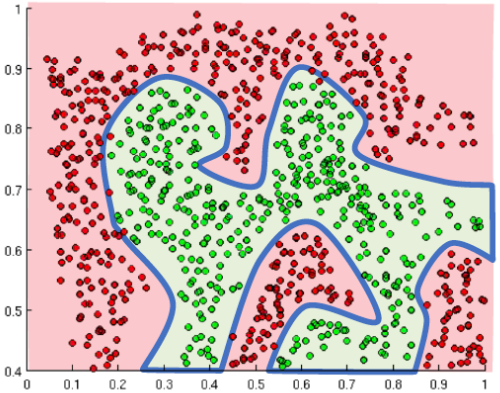
$$g(z) = \max(0, z)$$

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

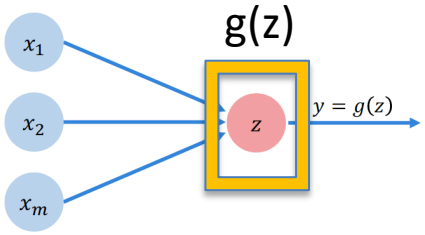
tf.nn.relu(z)



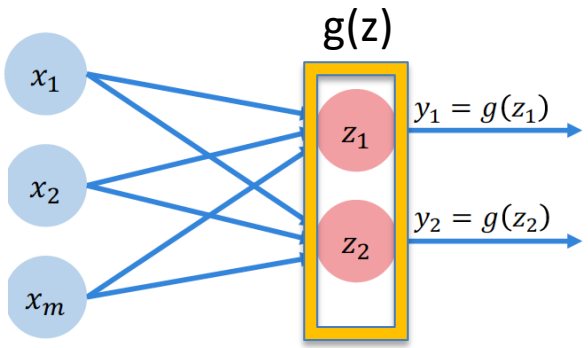




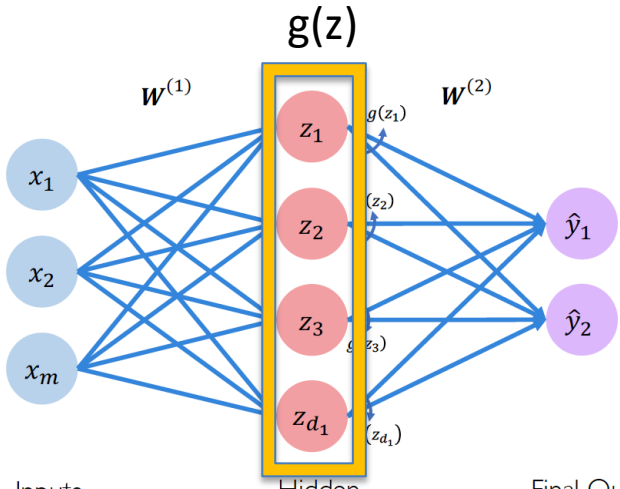
Multilayer Perceptron : Forward Path Calculation



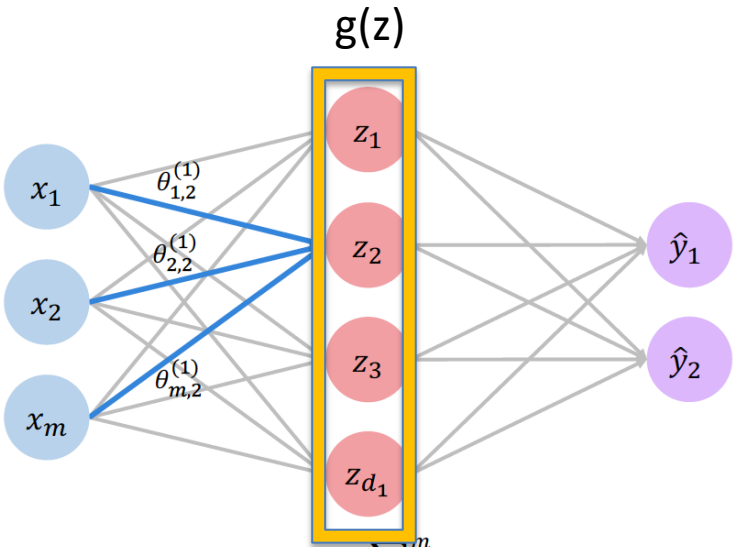
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$



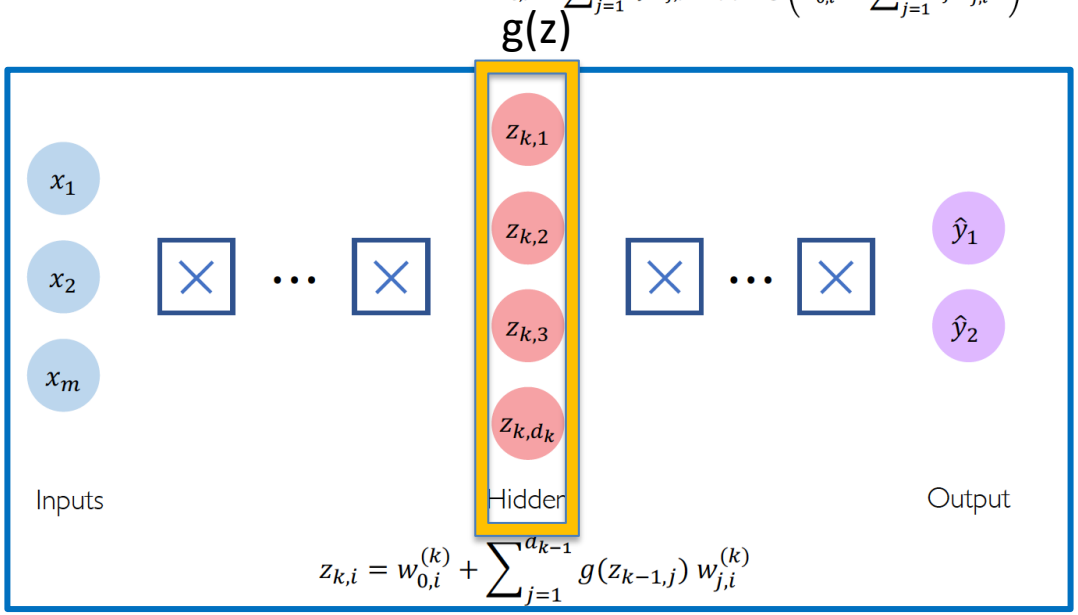
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$



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

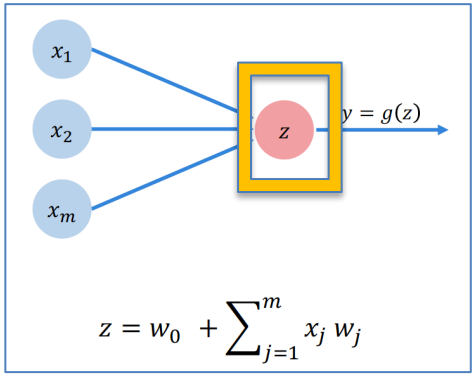


$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j w_{j,2}^{(1)}$$
$$= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)}$$



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

Forward Calculation represented as a graph



Sigmoid function

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

X1=-1.0

w1=-2.0

X2=-2.0

w2=3.0

b=-3.0

Z

g(z)

σ (z)

Y

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

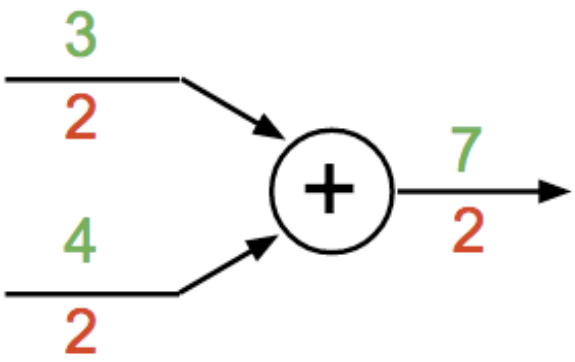
Z = (x1*w1 + x2*w2) + b = (-1.0*-2.0 + -2.0*3.0) + -3.0 = 4.0 – 3.0 = 1.0

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

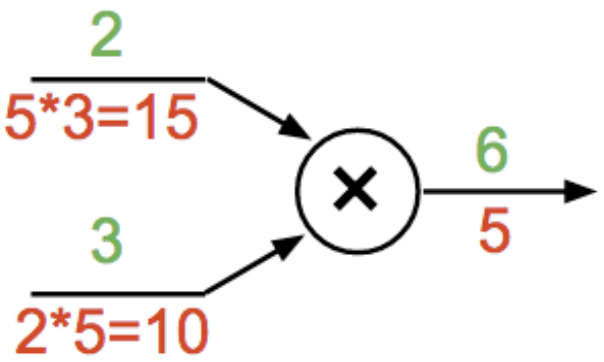
Forward computational Operators

Forward path (Green) : backward path

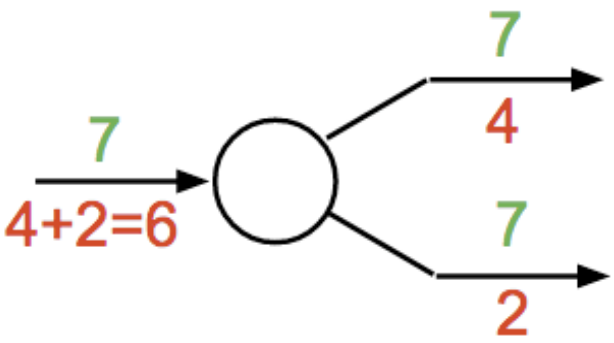
add gate: gradient distributor



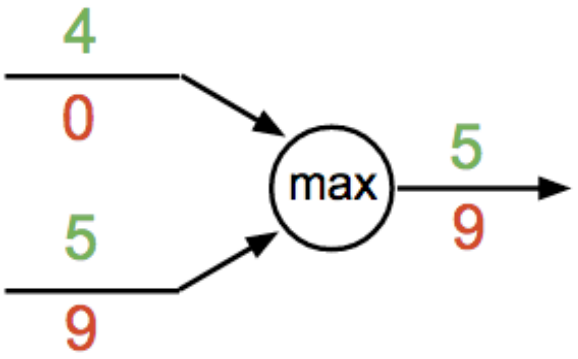
mul gate: "swap multiplier"



copy gate: gradient adder



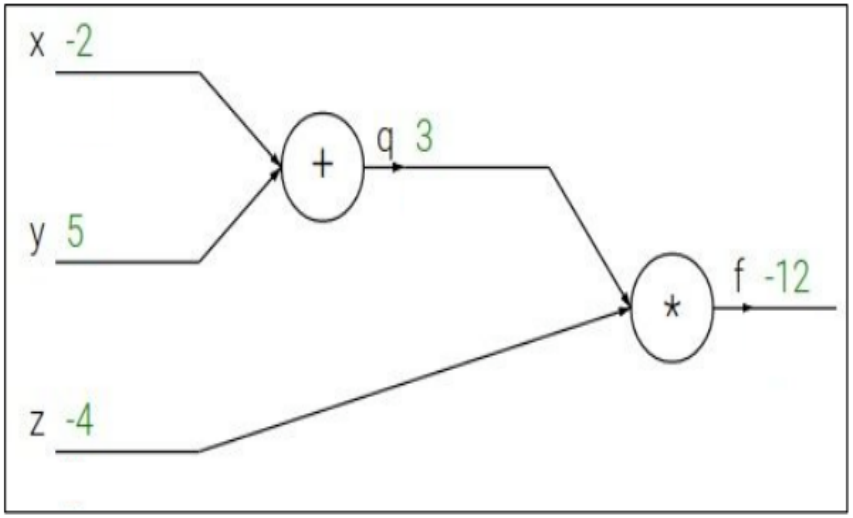
max gate: gradient router



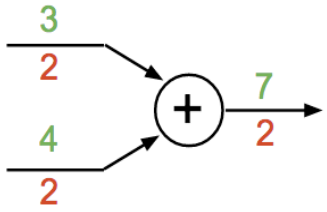
http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

Computational Graph

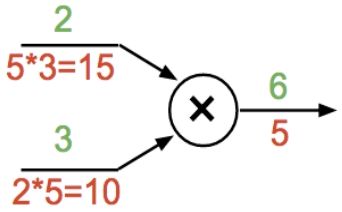
http://cs231n.stanford.edu/syllabus.html



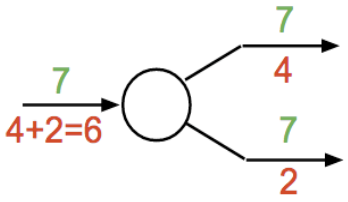
add gate: gradient distributor



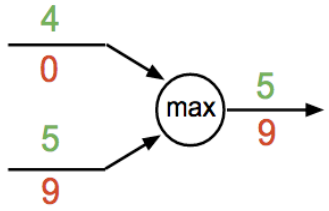
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router



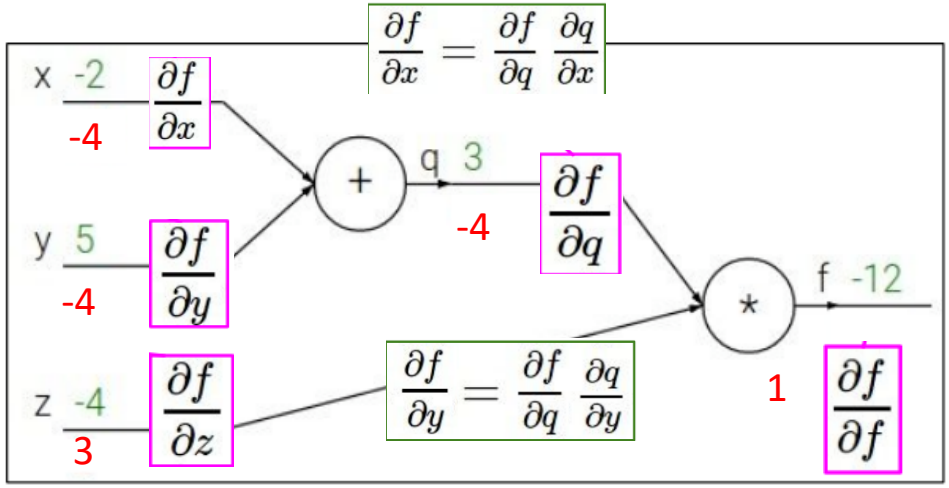
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

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

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

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

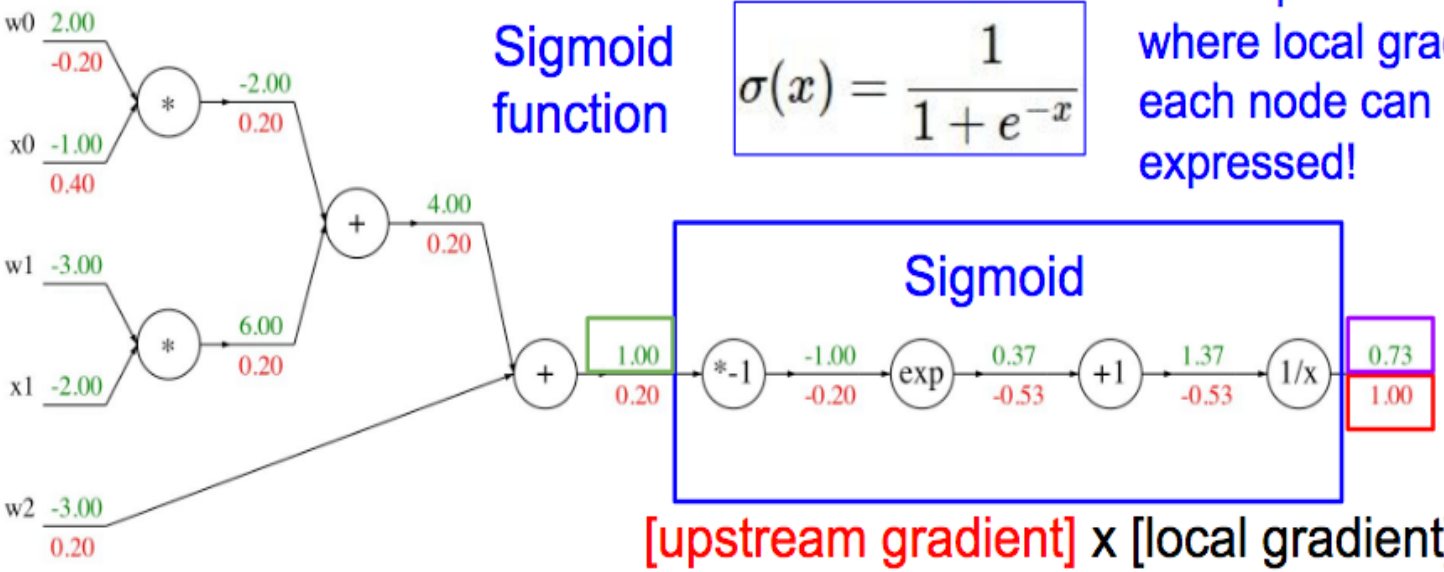
$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$



Computational Graph

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

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



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

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

[upstream gradient] x [local gradient]

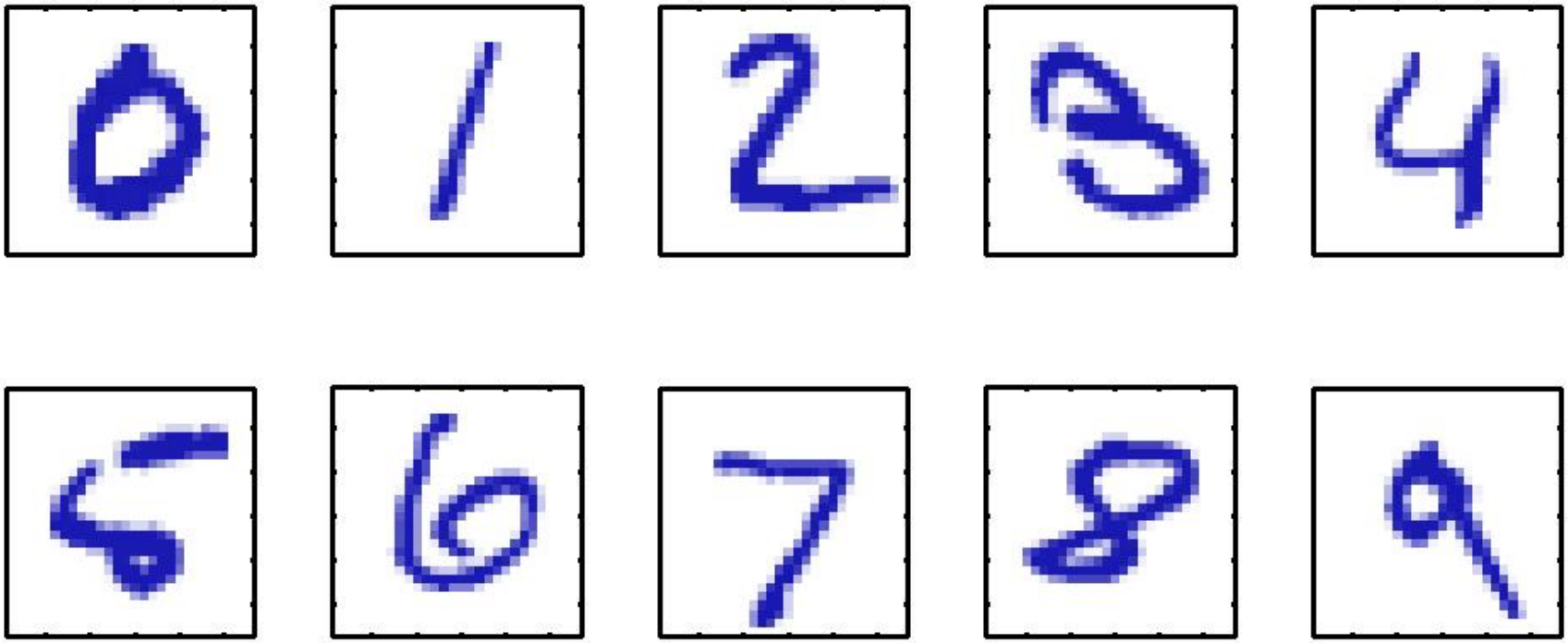
[1.00] x [(1 - 0.73) (0.73)] = 0.2

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

Example: how can computer see images?

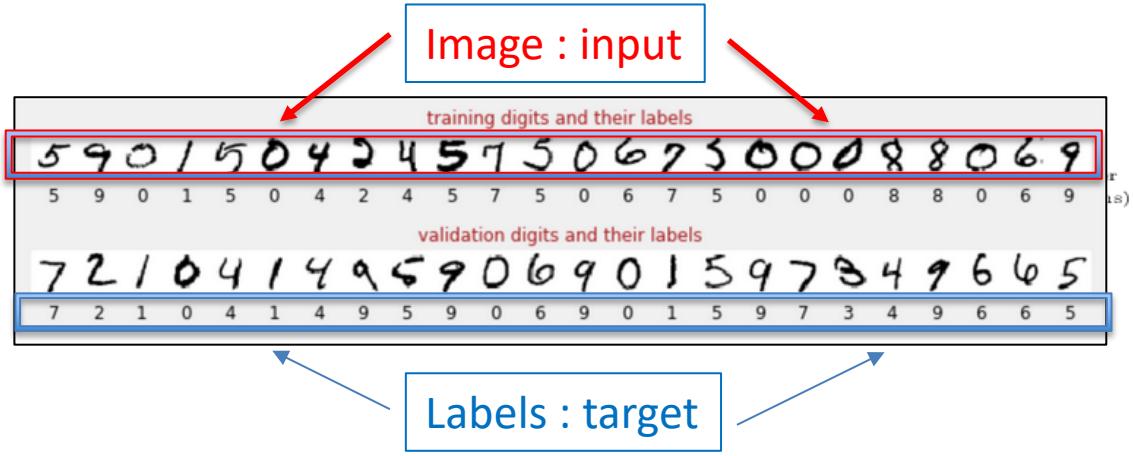
Handwritten Digit Recognition (MNIST data set)

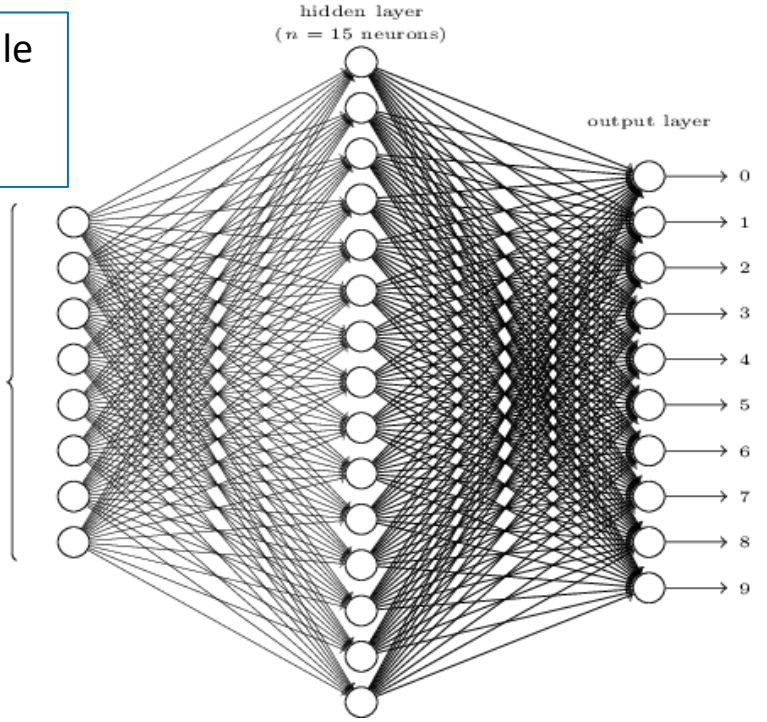
The **MNIST database** (*Modified National Institute of Standards and Technology database*) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.[4][5] 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

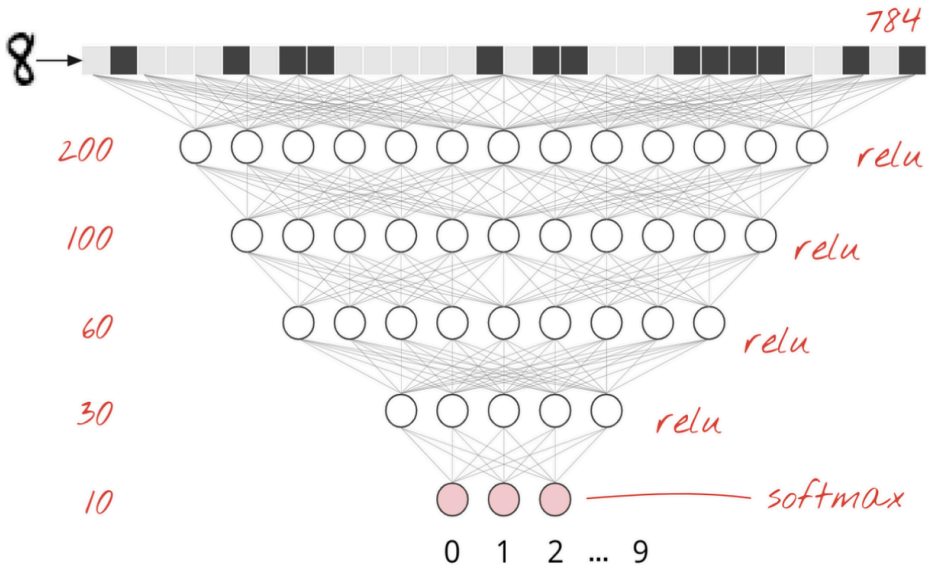


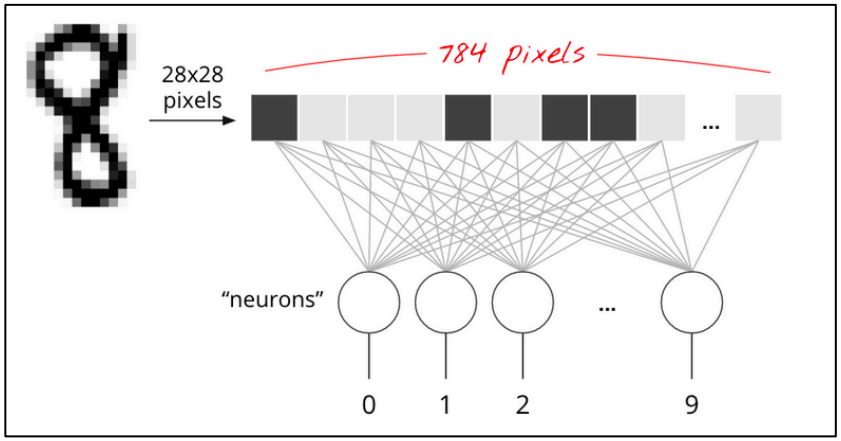
MNIST Example (28x28 pixels)

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

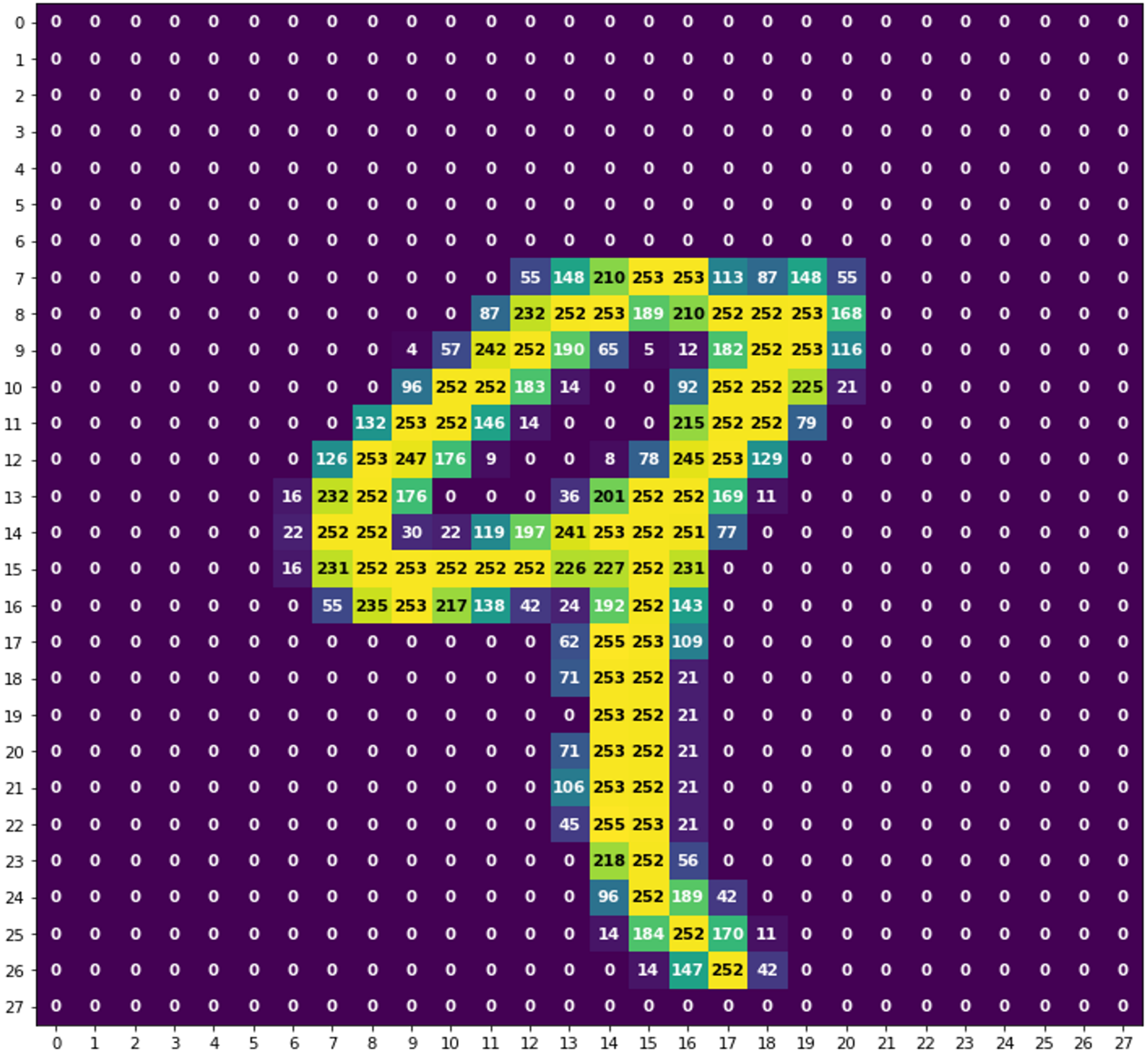


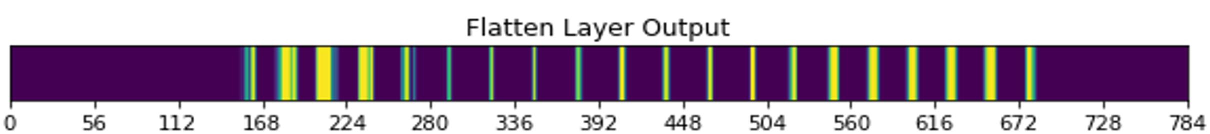






·Flow, Keras and deep learning, without a PhD
https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist#0





Grayscale image
of 28 x 28 pixels

same as a
28 x 28 matrix

values of
0 - 255

Flatten the

28 x 28 matrix

to a vector of
28 x 28 elements

784 elements

Simple MNIST MLP Neural Network

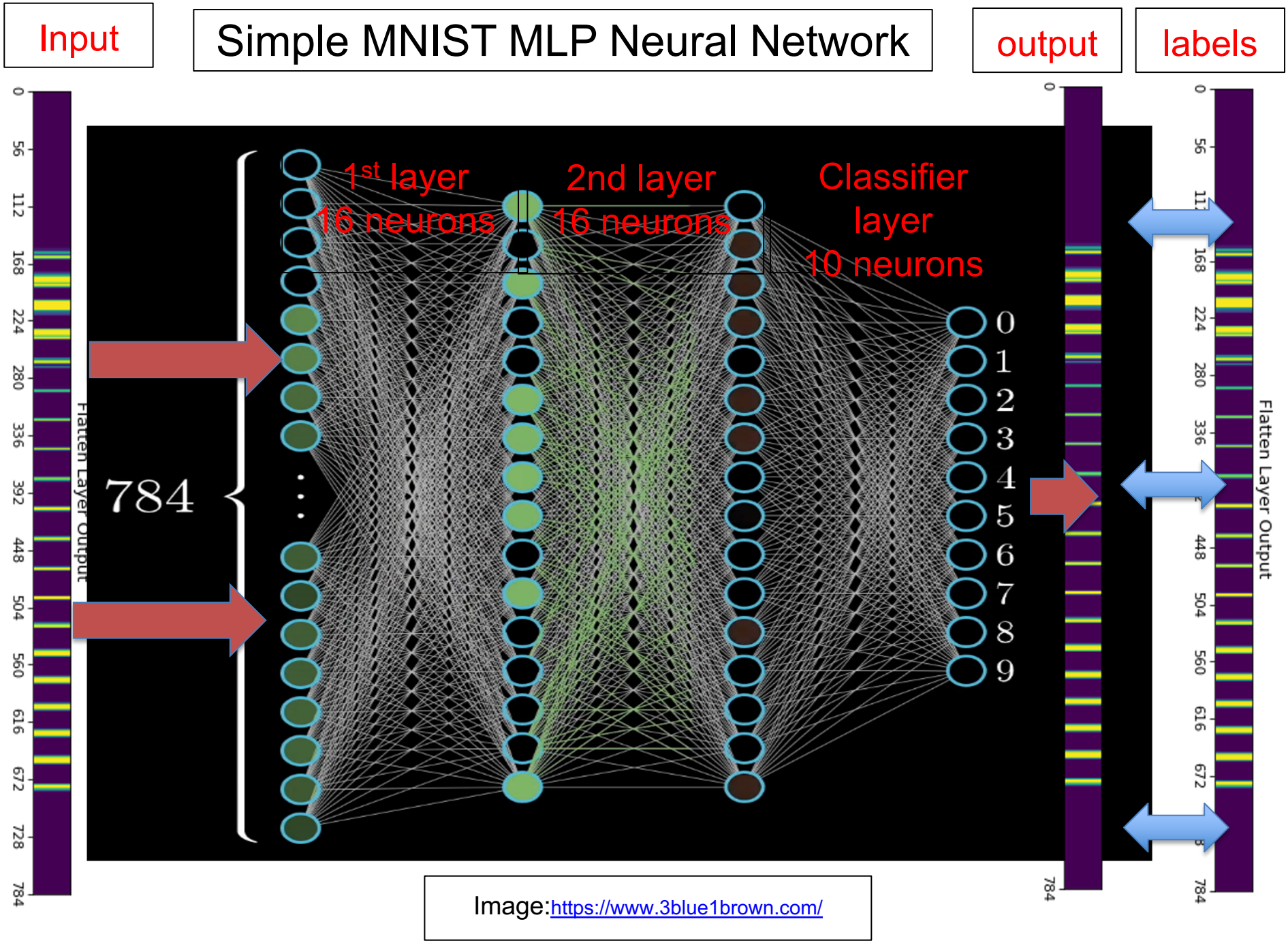


Image:https://www.3blue1brown.com/

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

Tensorflow Example

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

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

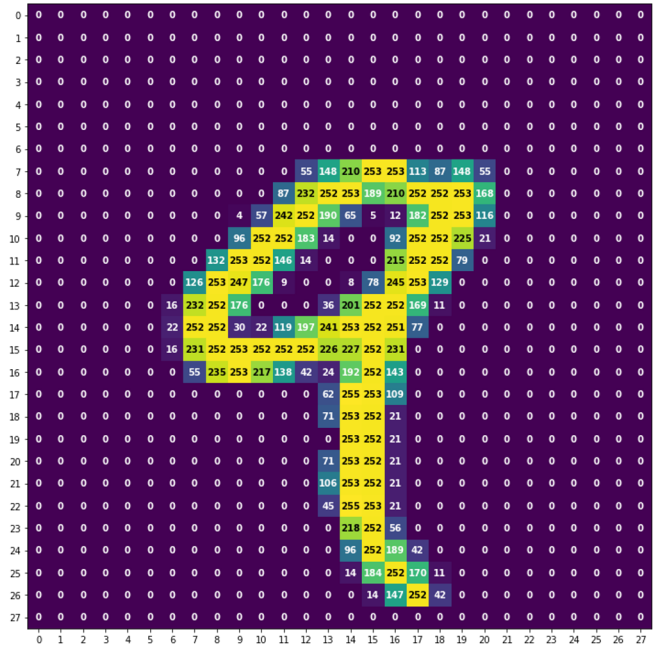
TensorFlow 2.0

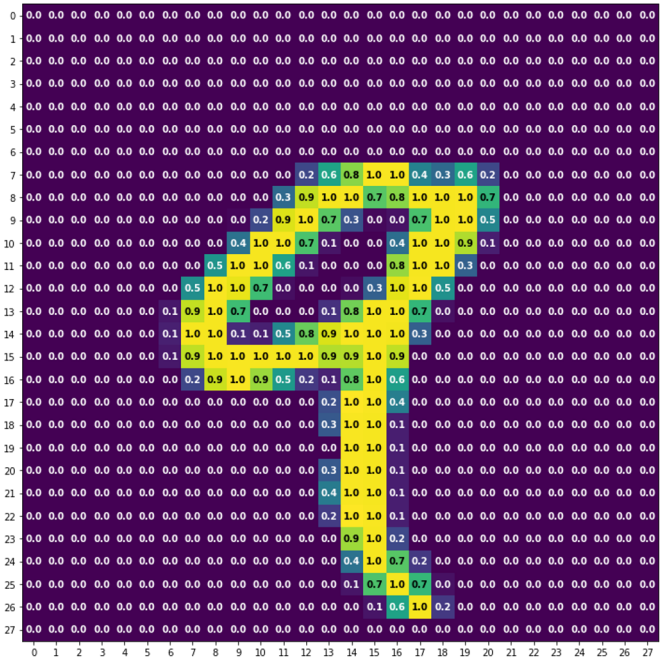
Loading and Preparing the MNIST Dataset

```
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

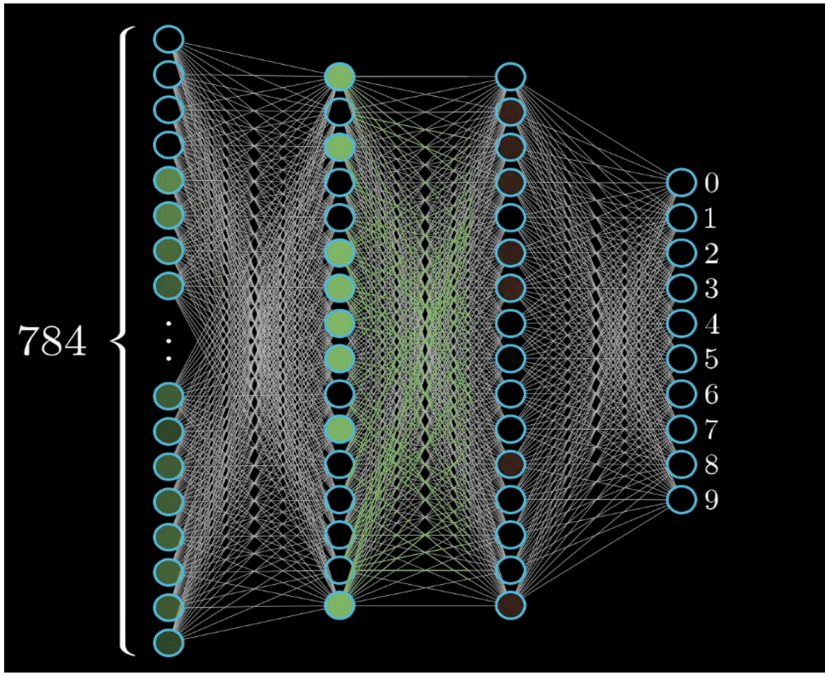


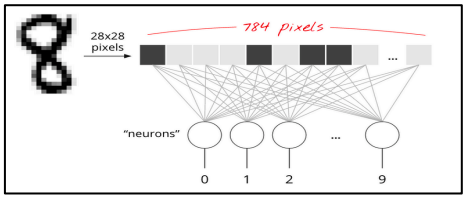
Image:
https://www.3blue1brown.com/

- ✓ **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)

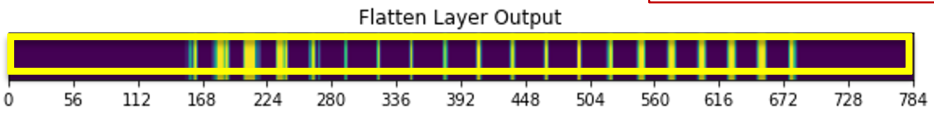
https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



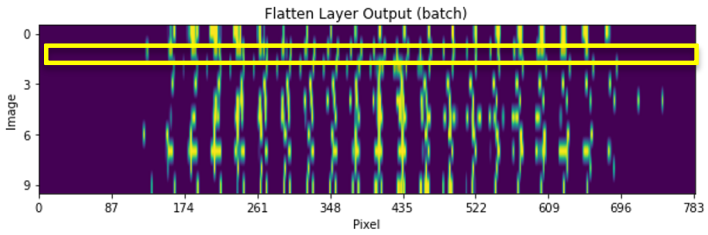
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

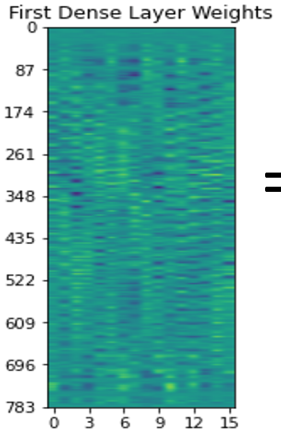
16 neurons
784 x 16 W matrix

10 x 16 output matrix

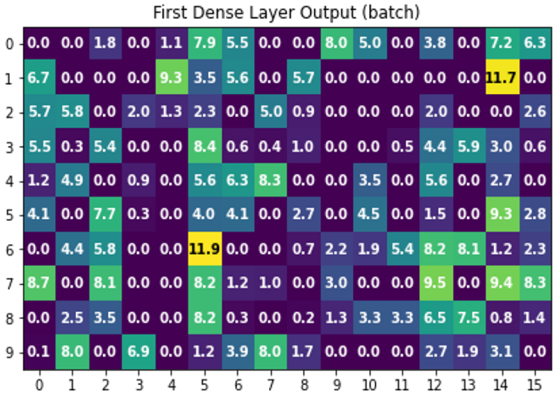
Input to 2nd layer



*

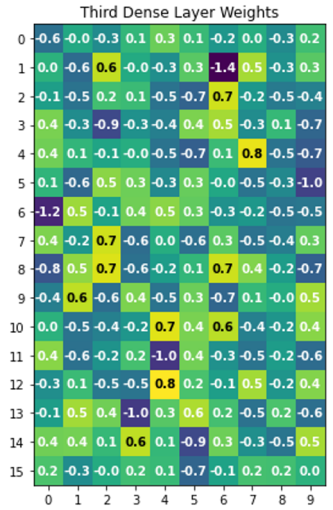


=



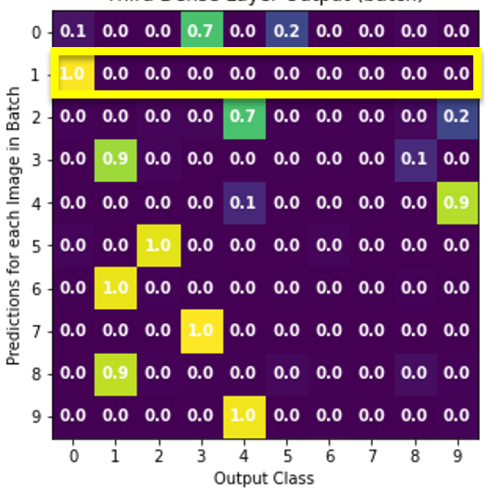
10 x 16 matrix

*

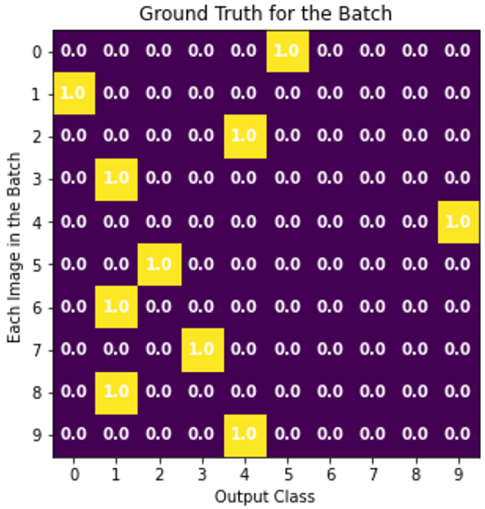


16 neurons
16 x 10
W matrix

=

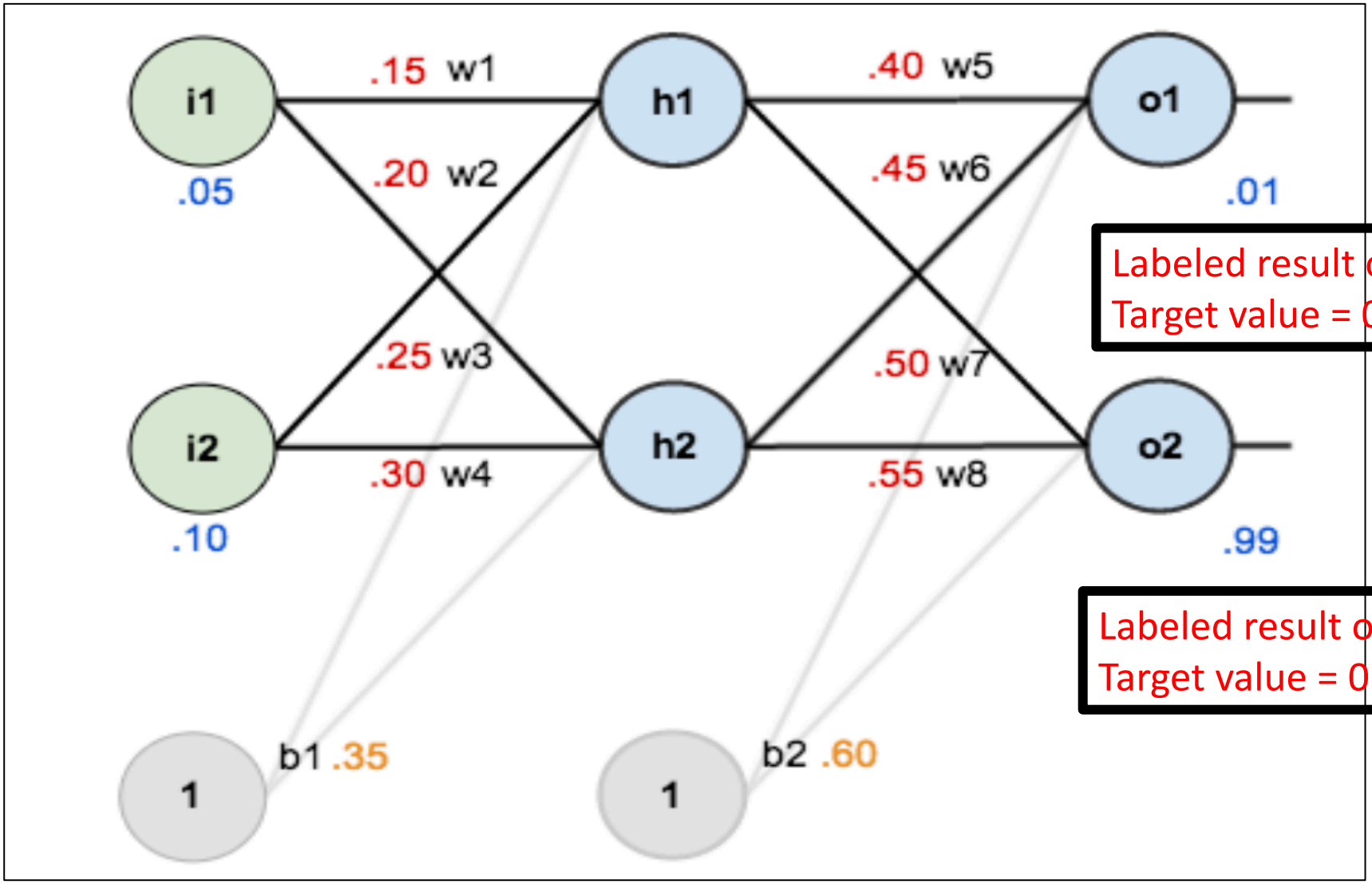


10 x 10 output matrix

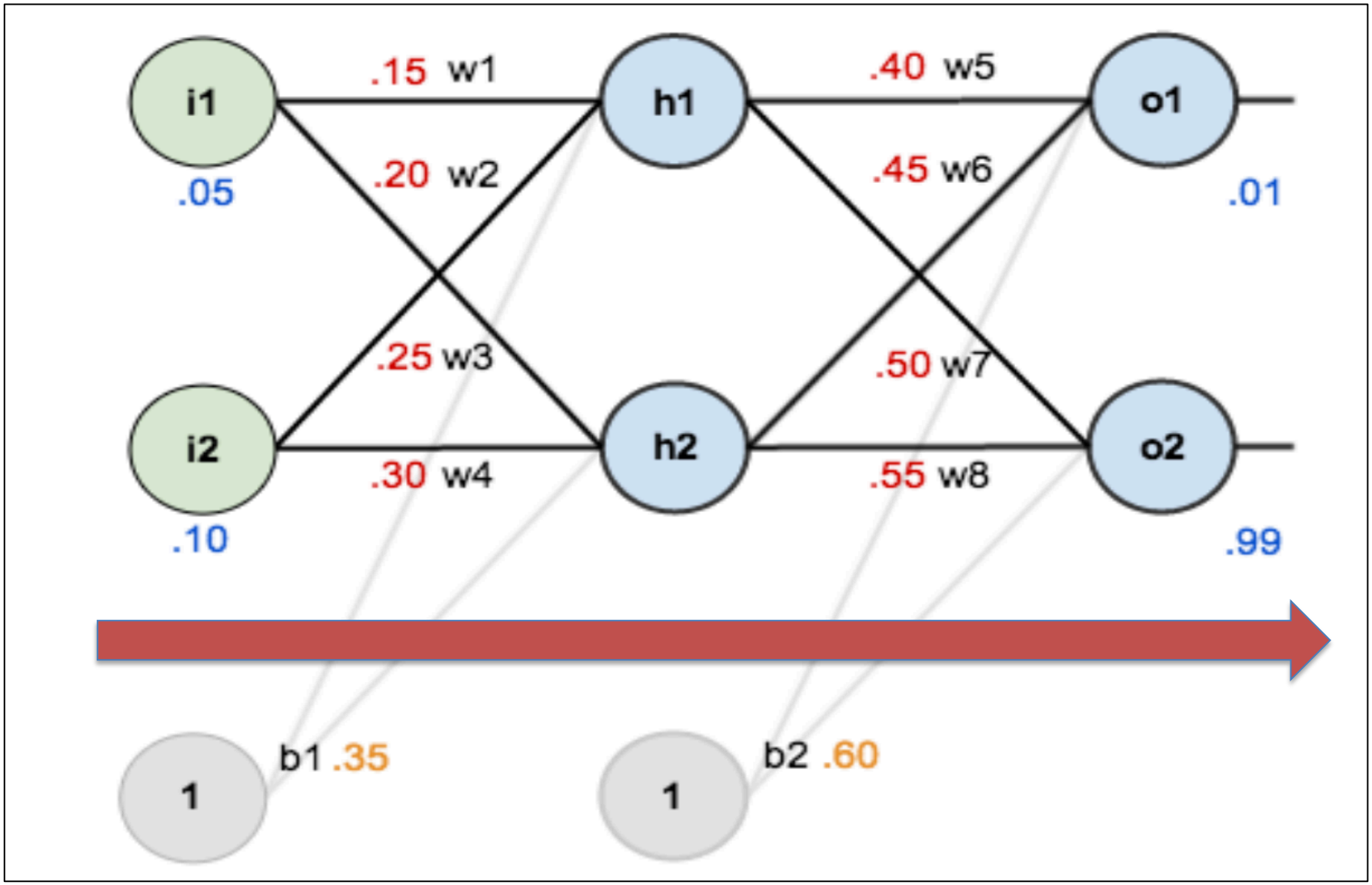


10 x 10 labeled matrix

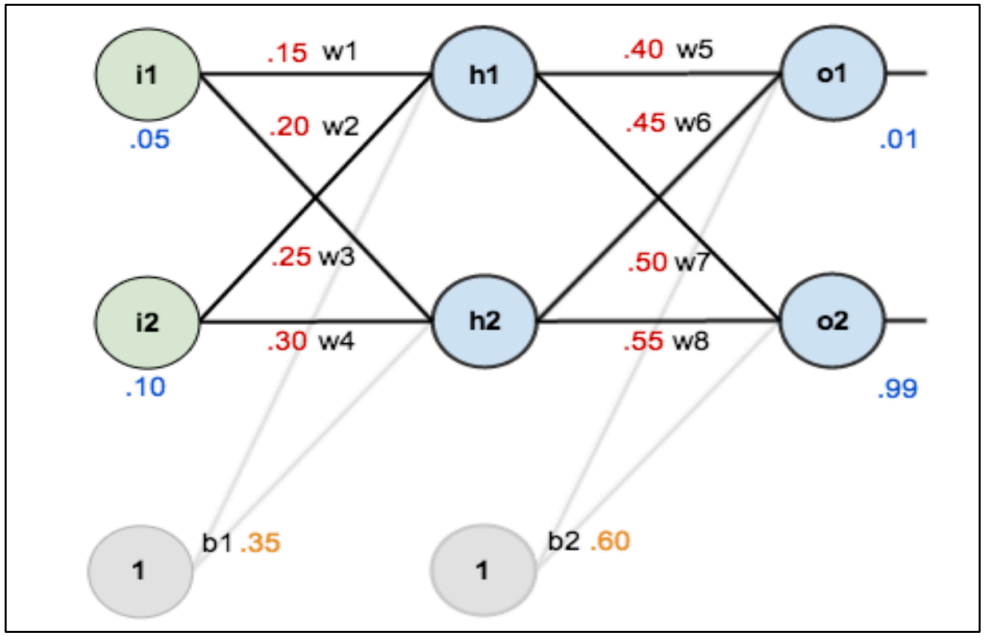
DNN Example


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

Multilayer Perceptron : Forward Path Calculation



DNN Example (1)



w1=0.15

i1 = 0.05

*

w2=0.2

i2 = 0.1

*

+

+

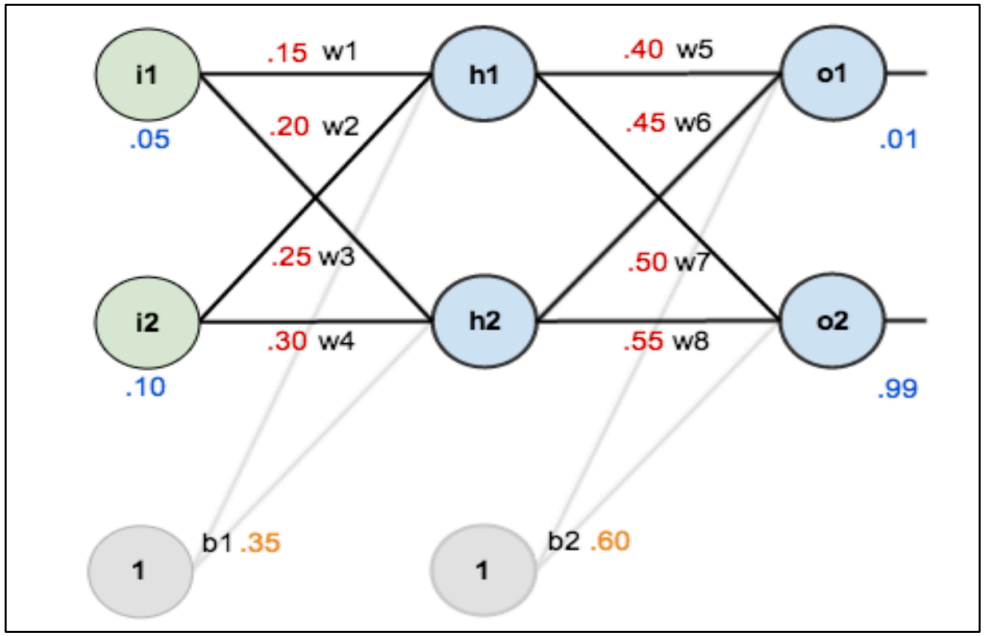
σ

h1 = 0.59326992

b1=0.35

1

DNN Example (2)



w3=0.25

i1 = 0.05

*

w4=0.3

i2 = 0.1

*

+

+

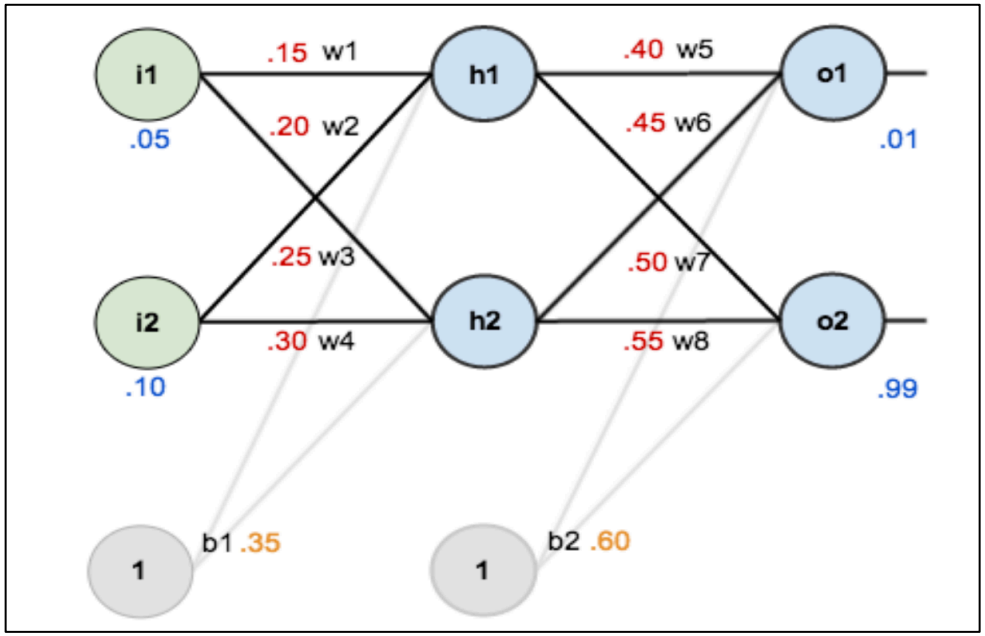
σ

h2 = 0.596884378

b1=0.35

1

DNN Example (3)



w5=0.4

h1= 0.59327

*

w6=0.45

h2 = 0.59688

*

+

+

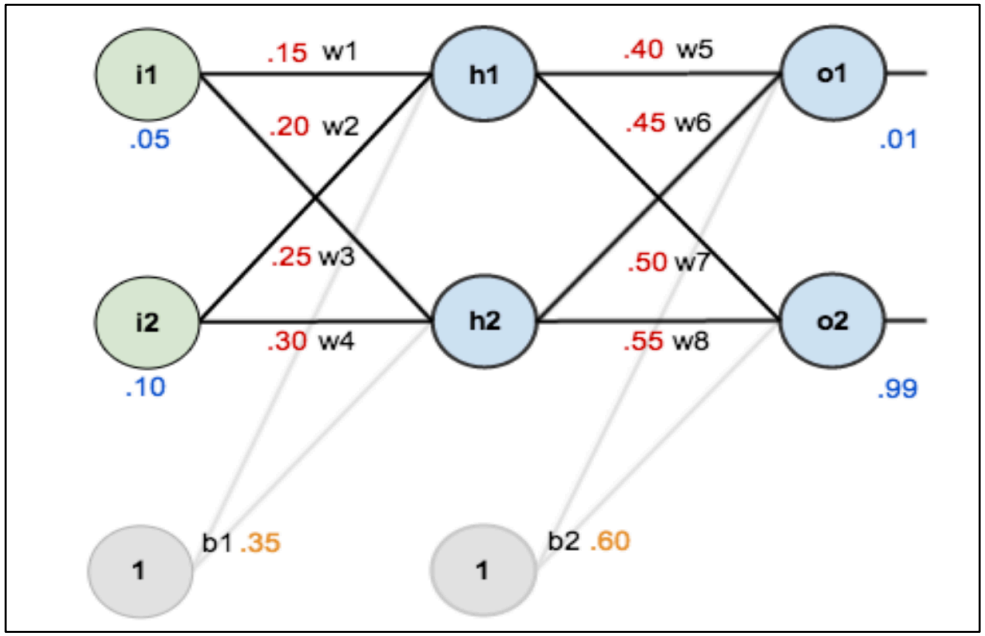
σ

o1 = 0.75136

b2=0.6

1

DNN Example (4)



w7=0.5

h1= 0.59327

*

w8=0.55

h2 = 0.59688

*

+

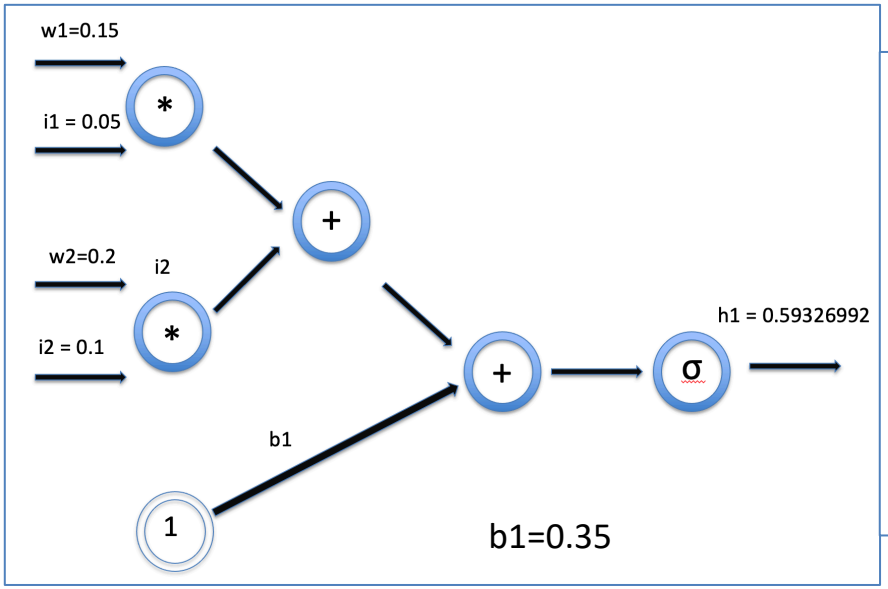
+

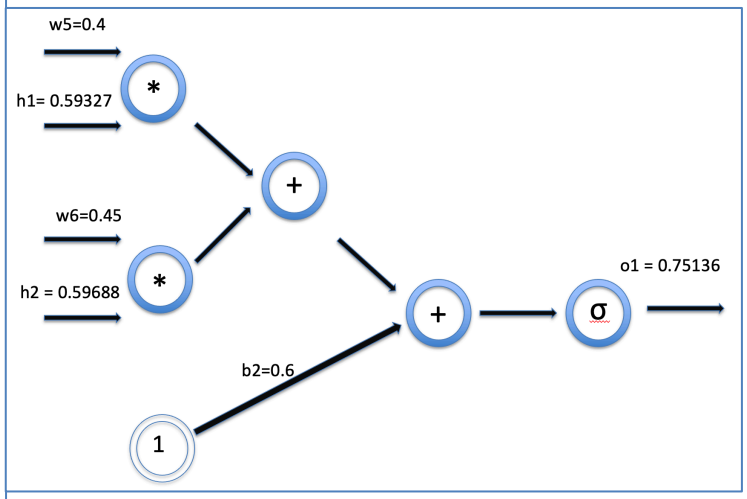
σ

o2 = 0.77293

B2=0.6

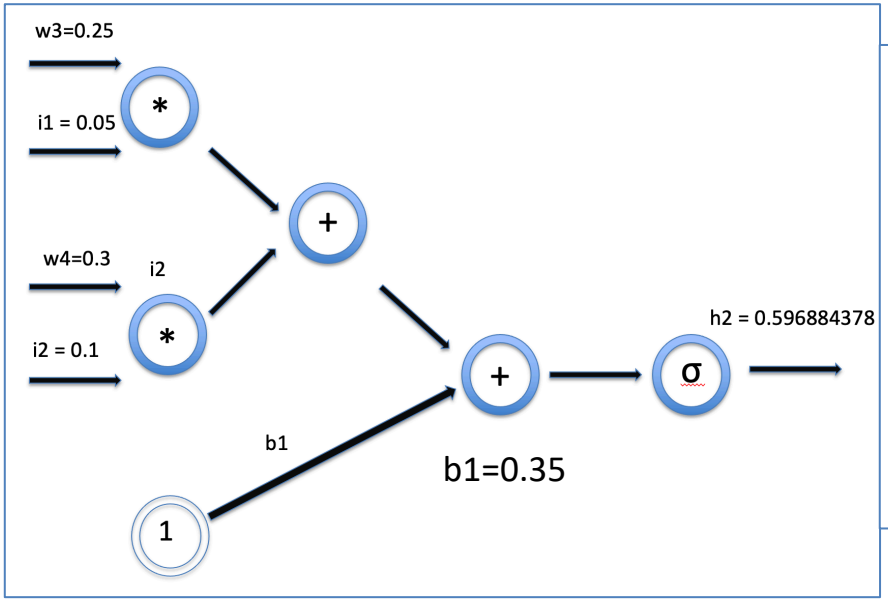
1

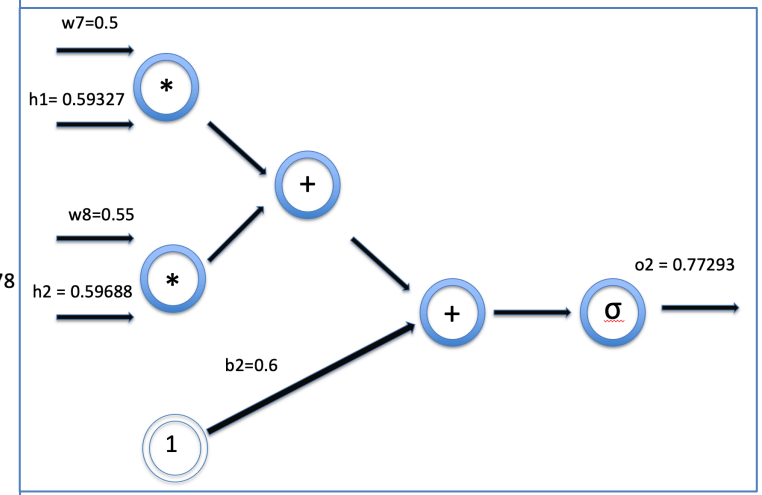




target=0.01

o1=0.75136



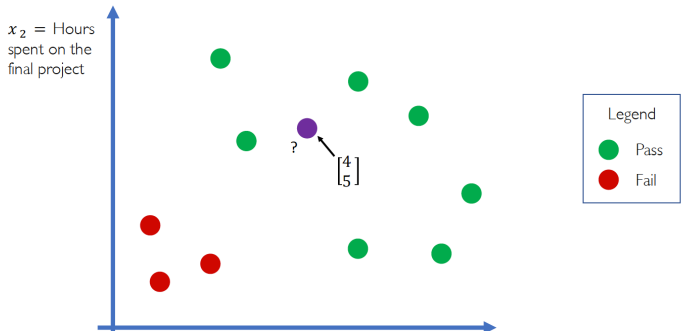


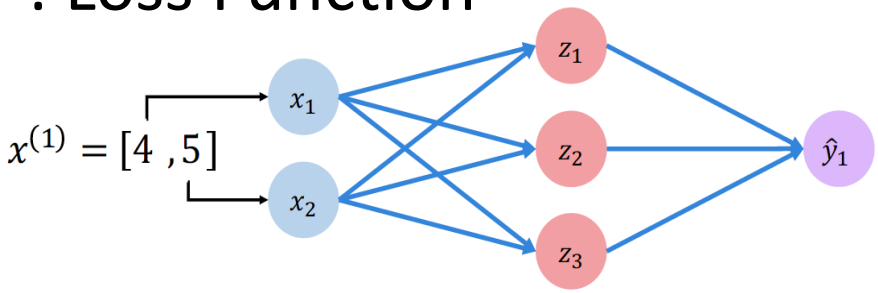
o2=0.77293

target=0.99

Quantify Error : Loss Function

Example Problem: Will I pass this class?



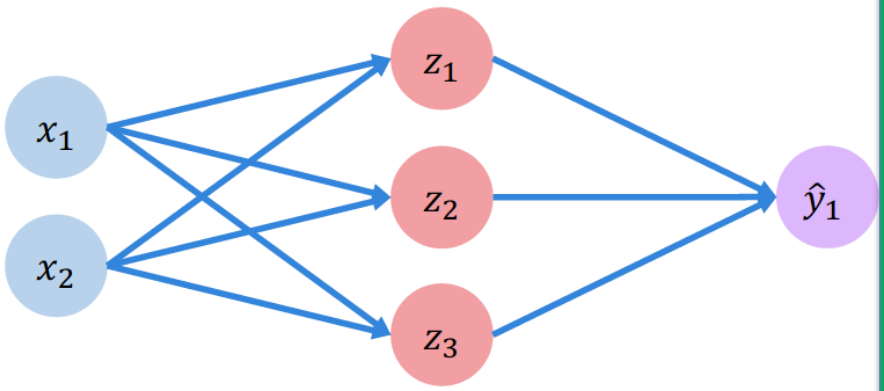


$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

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

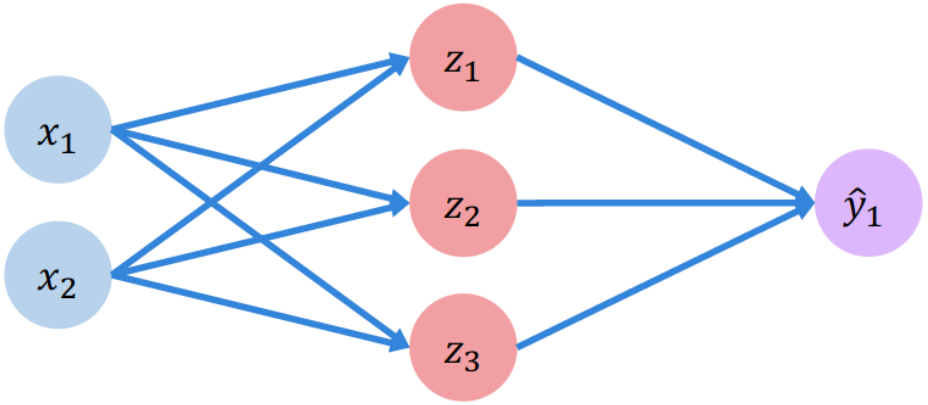
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

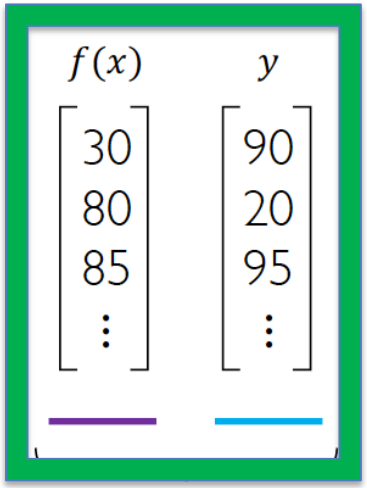
Predicted Actual

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)

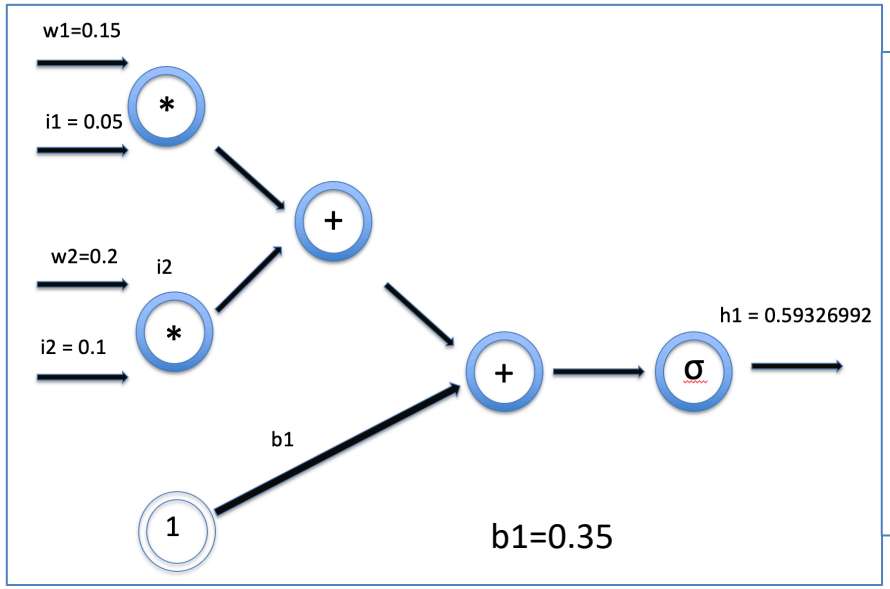
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

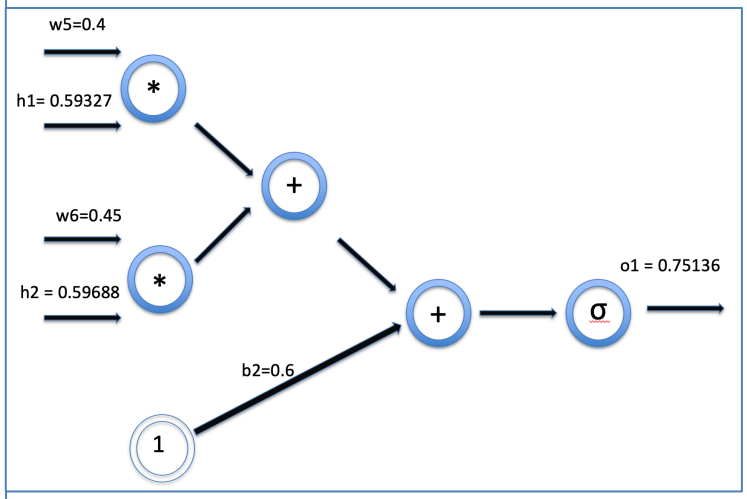
Actual Predicted

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

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

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





target=0.01

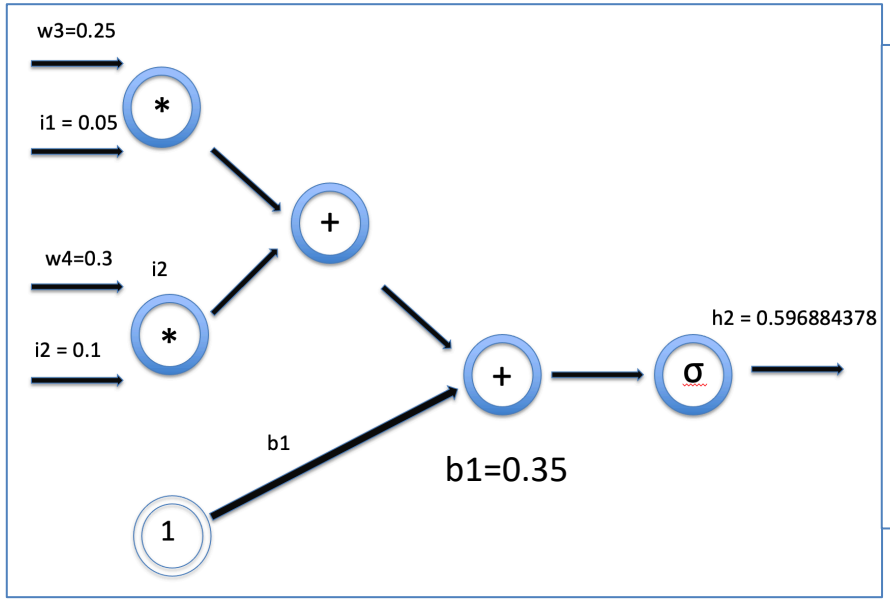
o1=0.75136

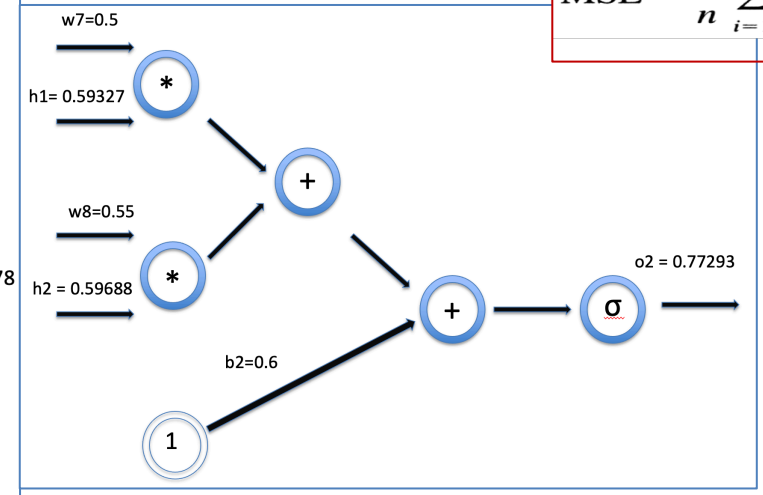
Eo1=0.27481

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

Etotal = Eo1 + Eo2 = 0.298371109

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





Eo2=0.02356

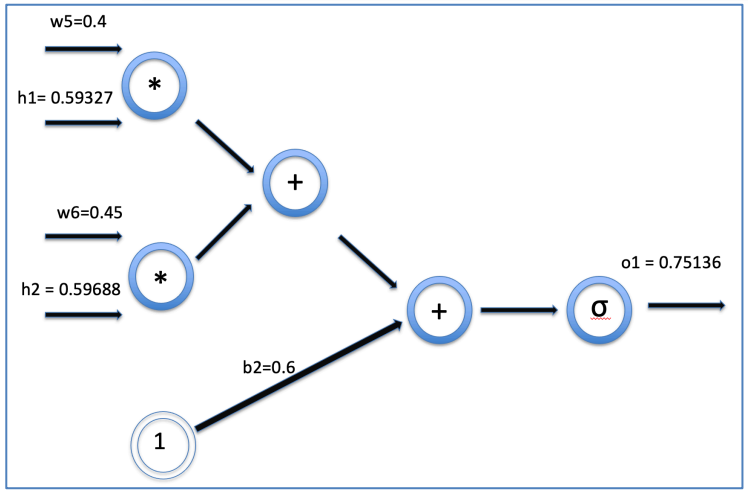
o2=0.77293

target=0.99

$$E_{o2} = 0.023560026$$

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

target=0.01 | 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

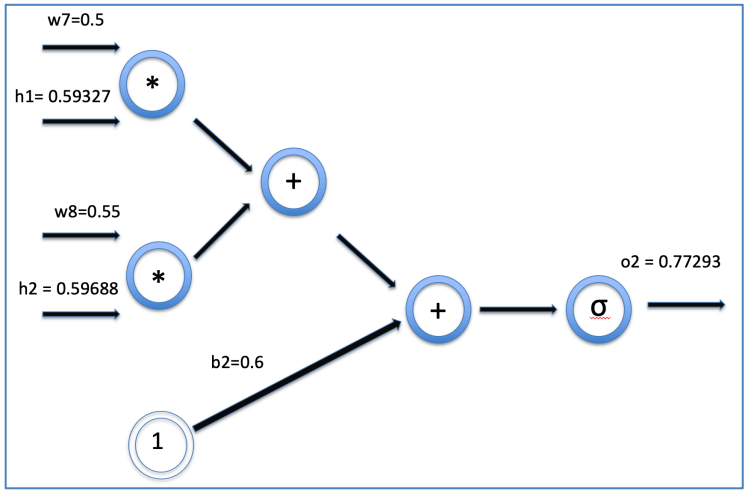
o2 = 0.77293

Cost f

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

target2 = 0.99

target=0.99 | o2=0.77293 | Eo2=0.02356



```
# 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))
```

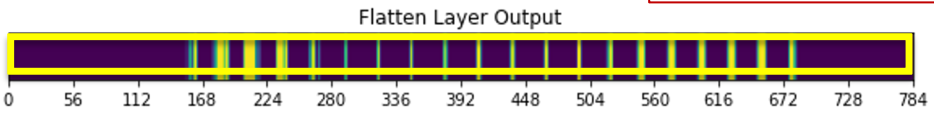
```
h1: 0.5932699921052087 h2: 0.5968843782577157
o1: 0.7513650695475076 o2: 0.772928465316421
```

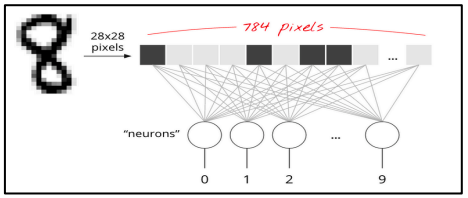
```
# Error
Eo1 = 0.5*(To1-o1)**2
print("Error o1: " + str(Eo1))
Eo2 = 0.5*(To2-o2)**2
print("Error o2: " + str(Eo2))

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

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

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784

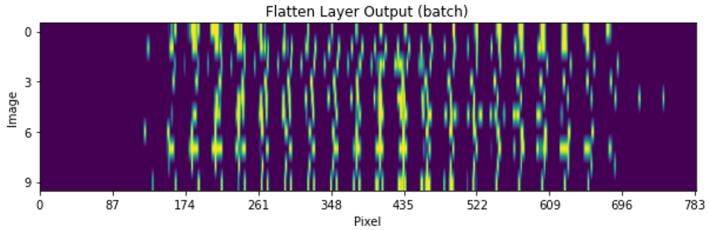




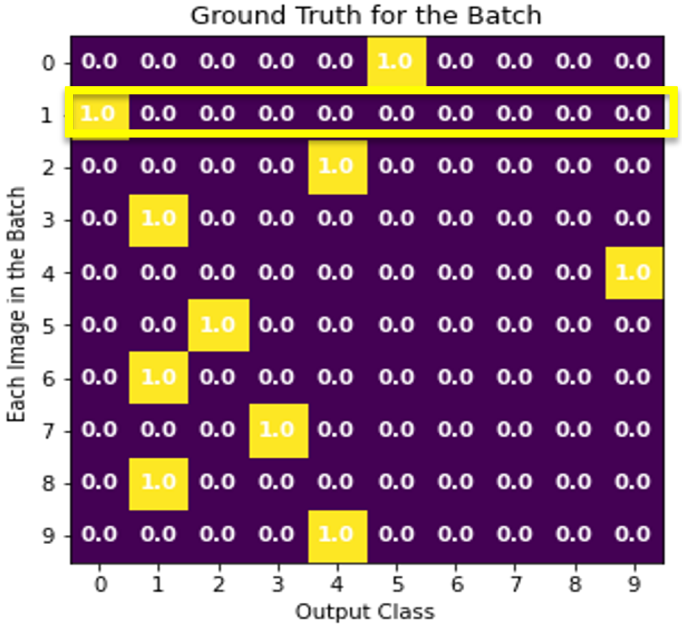
An image of 28x28 pixels

Input to 1st layer

10 images : batch of 10
10 x 784 matrix

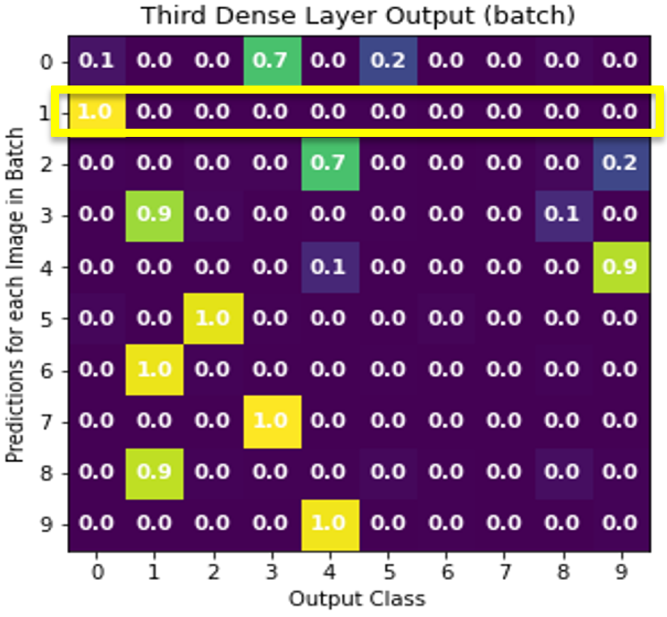


10 x 10 labeled matrix



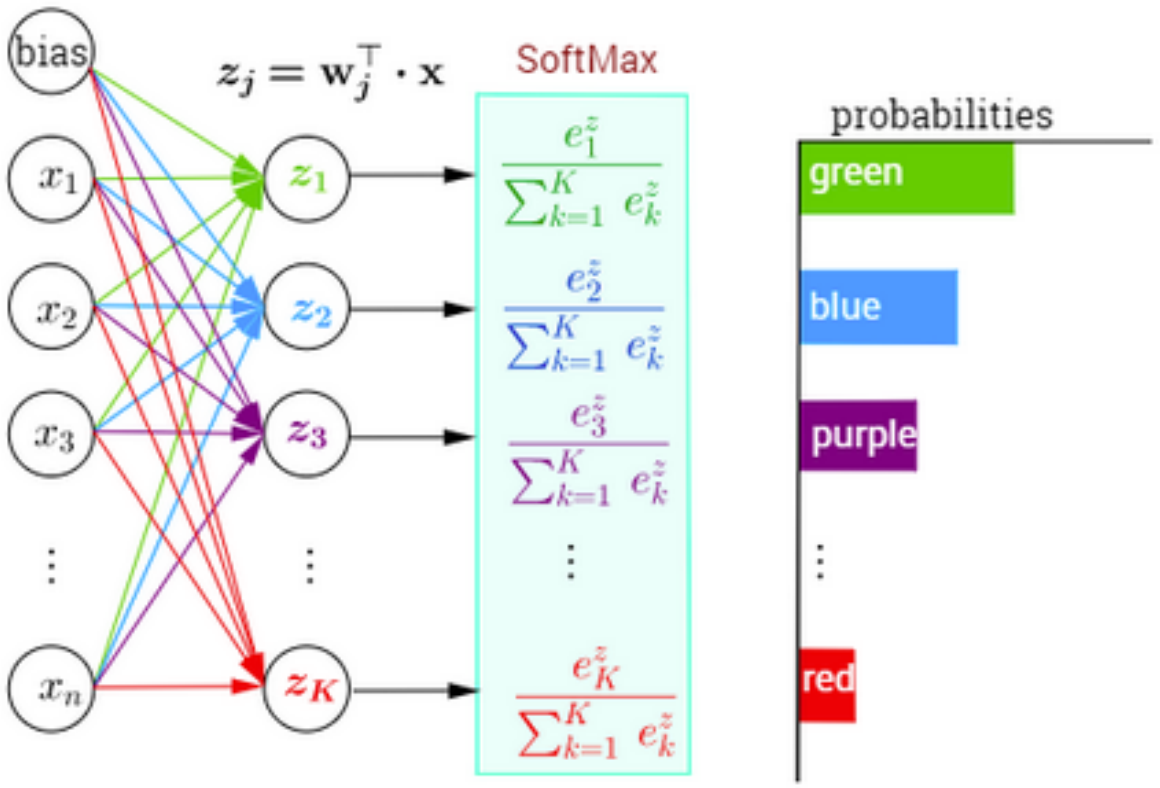
Comparing Computed NN results To Labeled results (target)

10 x 10 output matrix



Multi-Class Classification with NN and SoftMax Function

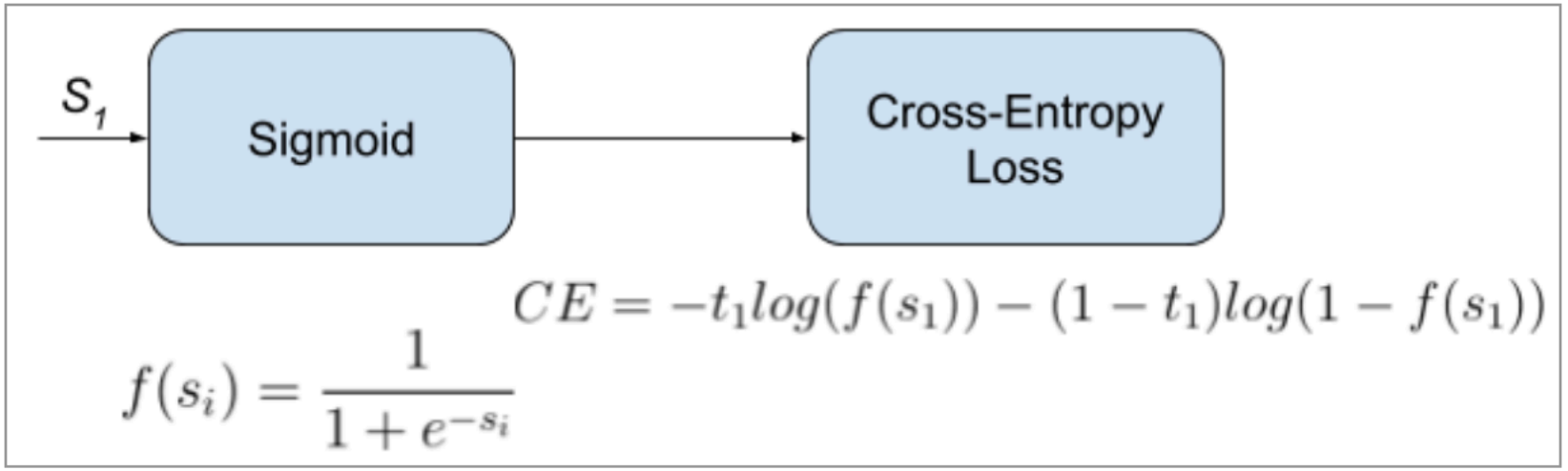
$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \mathbf{w}_3^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



Softmax Function

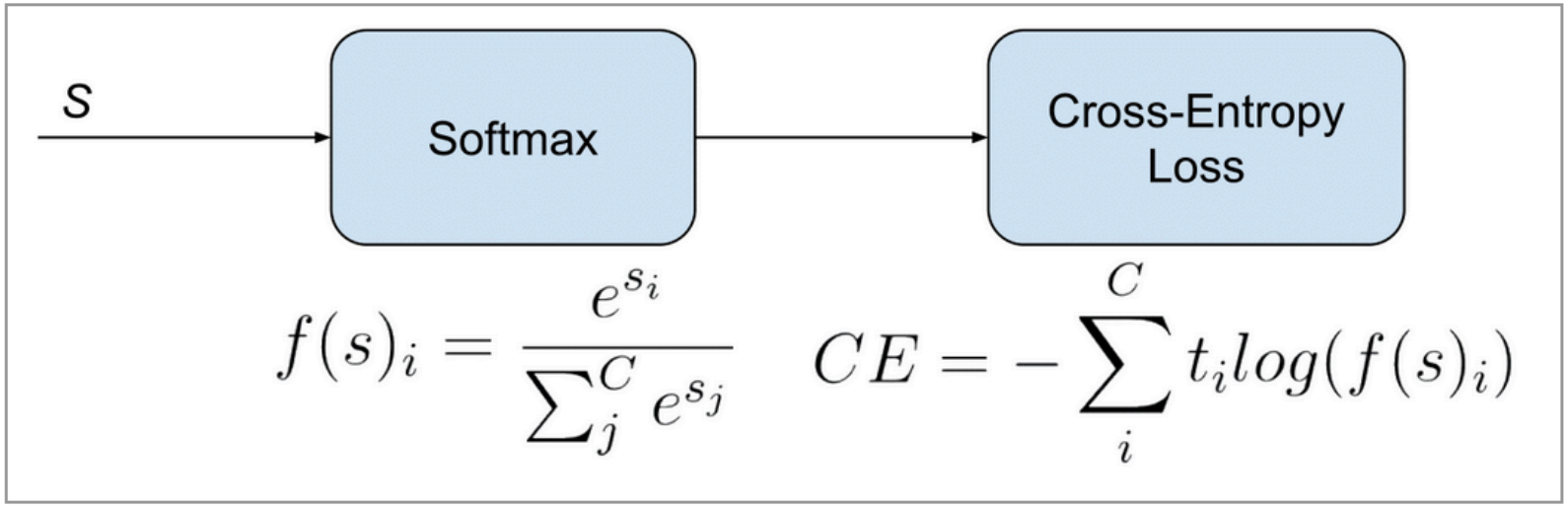
$$\sigma(j) = \frac{\exp(\mathbf{w}_j^\top \mathbf{x})}{\sum_{k=1}^{K} \exp(\mathbf{w}_k^\top \mathbf{x})} = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

Binary (Sigmoid) Cross Entropy Loss (Statistical Learning)



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

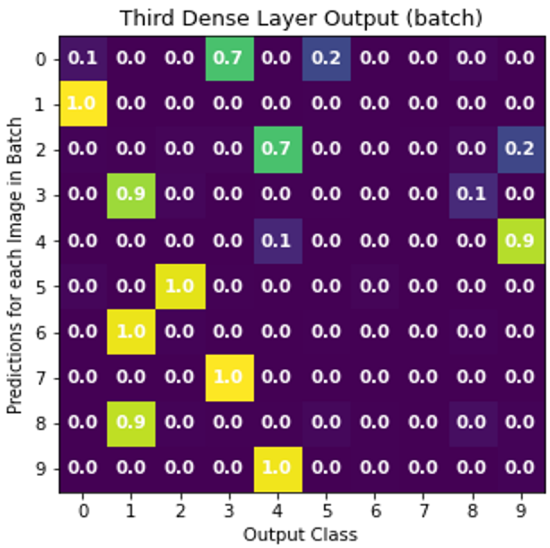
Categorical (Softmax) Cross Entropy Loss (Statistical Learning)

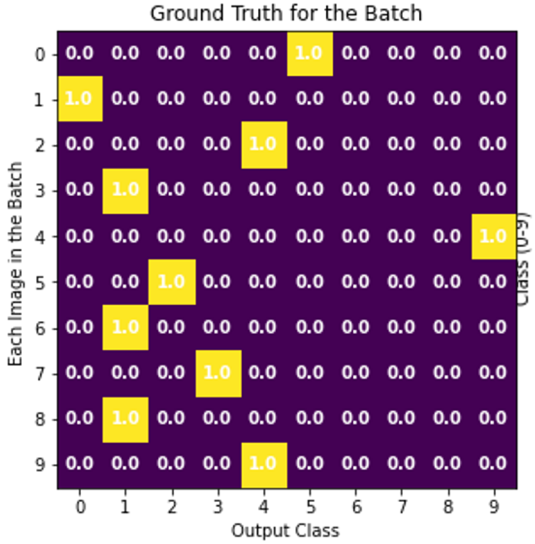


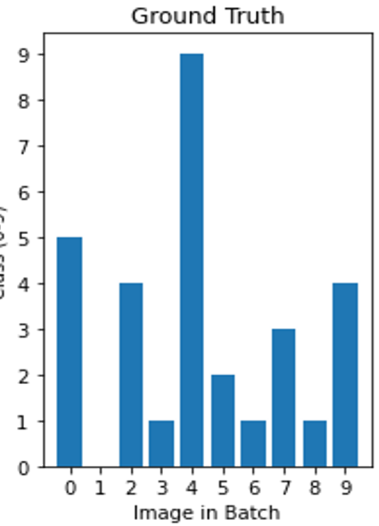
https://gombru.github.io/2018/05/23/cross_entropy_loss/

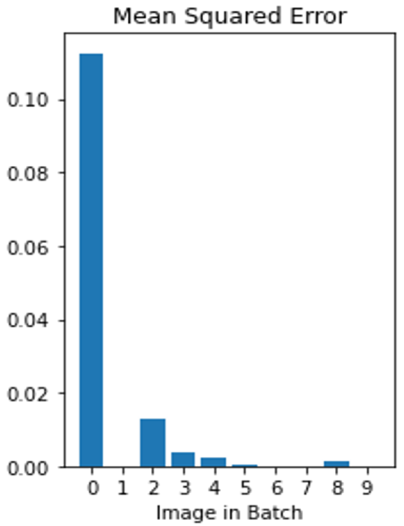
Evaluating the Error

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









Categorical Cross Entropy Loss

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

Softmax Classifier (Multinomial Logistic Regression)

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

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

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

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

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

Cross Entropy

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

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

Training

"Training a neural network"

What does it mean?

What does the network train?

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

What are the training parameters

W and b

How does it work?

optimization based on gradient descent algorithm

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

Go through forward calculation

training with backpropagation

Derivatives and Gradient descent

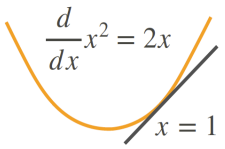
Review Scalar Derivative

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

a is not a function of x

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

Derivative is the slope of the tangent line



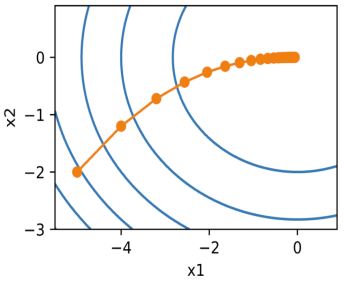
The slope of the tangent line is 2

Algorithm

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

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

- η is called learning rate



Generalize to Vectors

- Chain rule for scalars:

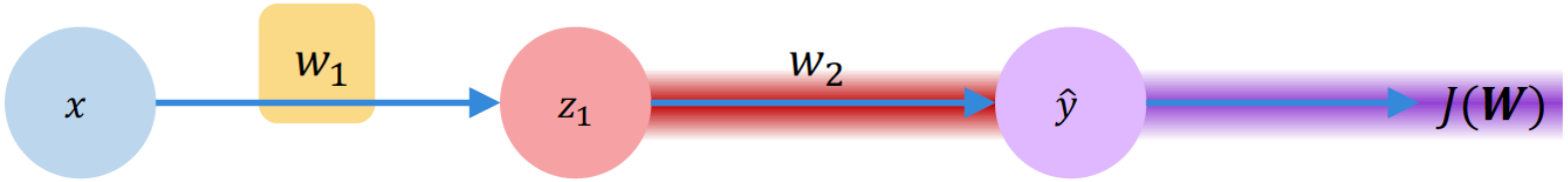
$$y = f(u),\ u = g(x) \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

- Generalize to vectors straightforwardly

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

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

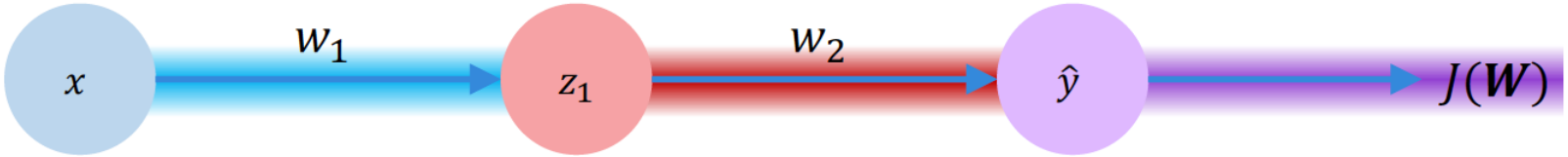
DNN MLP LA Kernels



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

Apply chain rule!

Apply chain rule!



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

Composite Differentiation

$$z = f(w_1(\theta_1, \theta_2), w_2(\theta_1, \theta_2))$$

$$\frac{\partial f}{\partial \theta_1} = \frac{\partial f}{\partial w_1} \cdot \frac{\partial w_1}{\partial \theta_1} + \frac{\partial f}{\partial w_2} \cdot \frac{\partial w_2}{\partial \theta_1}$$

Calculus background: how to compute gradients?

- Finite Difference Method
- Symbolic Differentiation
- **Automatic Differentiation in Reverse Mode (called Backpropagation when applied to Neural Networks)**
- Automatic Differentiation in Forward Mode

$$z = f(w_1, w_2) = \frac{w_1^2}{a^2} + \frac{w_2^2}{b^2}$$

$$\frac{\partial f}{\partial w_1} = \frac{2w_1}{a^2}$$

$$\frac{\partial f}{\partial w_2} = \frac{2w_2}{b^2}$$

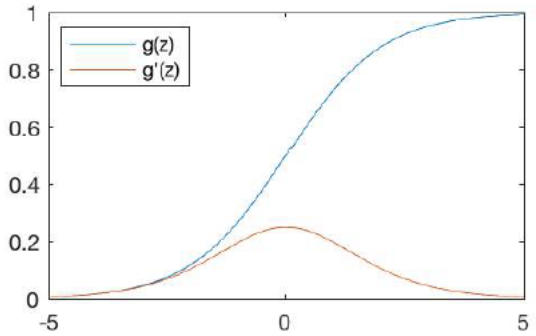
$$\nabla_{\mathbf{w}} f = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \frac{2w_1}{a^2} \\ \frac{2w_2}{b^2} \end{bmatrix}$$

Forward	Backward
$J = y^* \log y + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
$y = \frac{1}{1 + \exp(-b)}$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$
$b = \sum_{j=0}^{D} \beta_j z_j$	$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$ $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$
$z_j = \frac{1}{1 + \exp(-a_j)}$	$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$ $\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$

- Each hidden node j is responsible for some fraction of the error δj(l) in each of the output nodes to which it connects
- δj(l) is divided according to the strength of the connection between hidden node and the output node
- Then, the "blame" is propagated back to provide the error values for the hidden layer

Nonlinear Activation Function

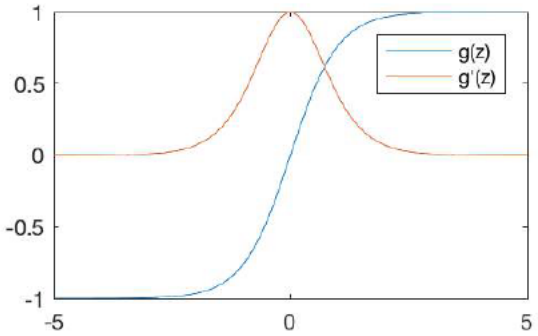
Sigmoid Function



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

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

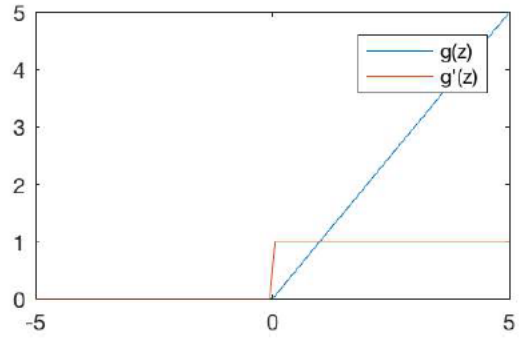
`tf.nn.sigmoid(z)`



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

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

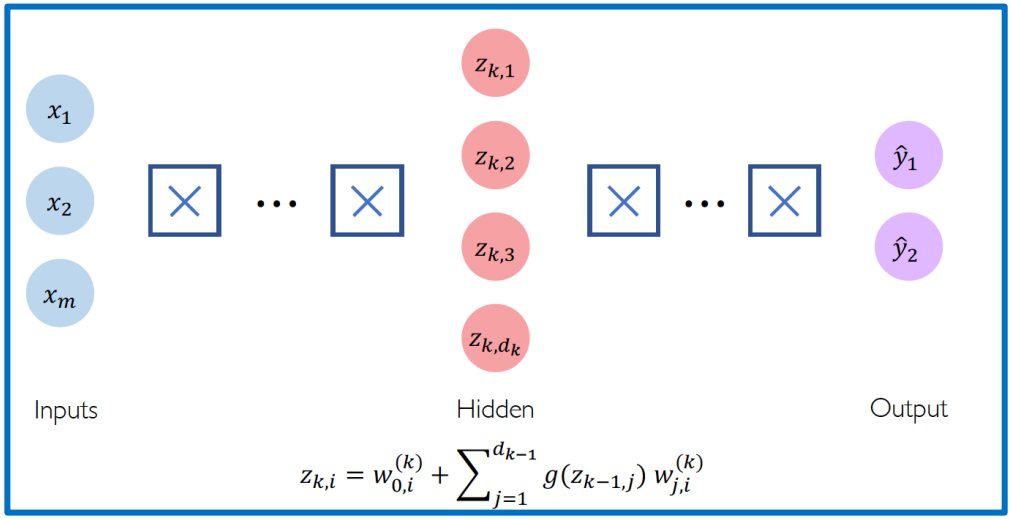
`tf.nn.tanh(z)`



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

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

`tf.nn.relu(z)`

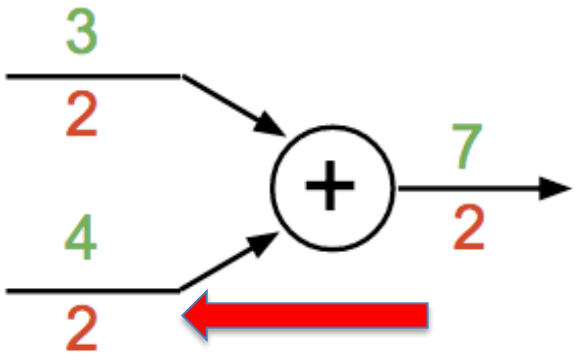


MIT, 6.S191 Introduction to Deep Learning,
introtodeeplearning.com

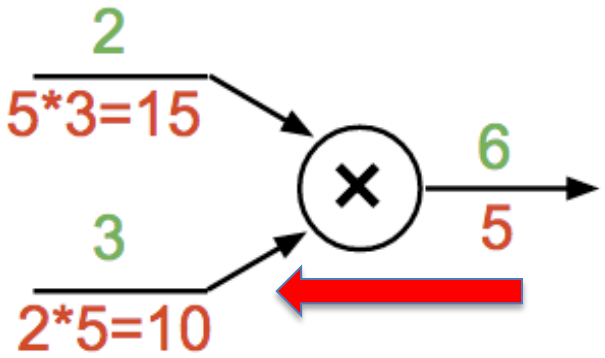
Forward and Backpropagation Operators

Forward path : backward path

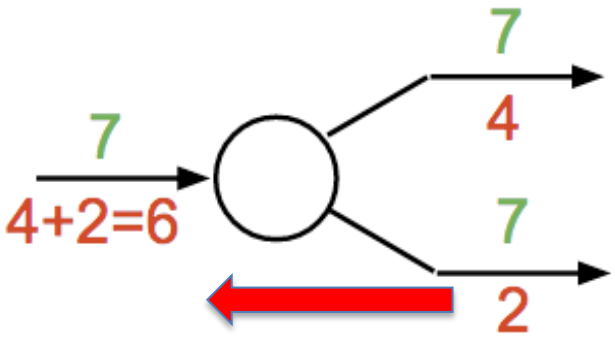
add gate: gradient distributor



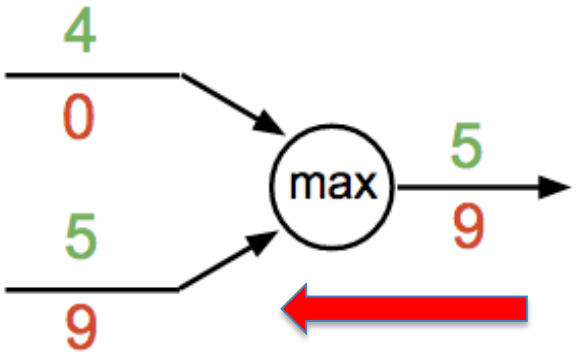
mul gate: "swap multiplier"



copy gate: gradient adder



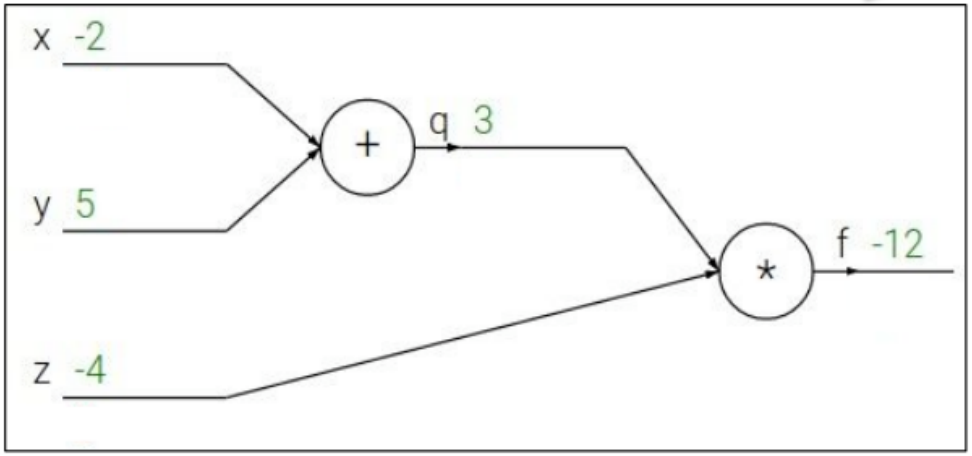
max gate: gradient router



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

Computational Graph

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



Forward path calculation

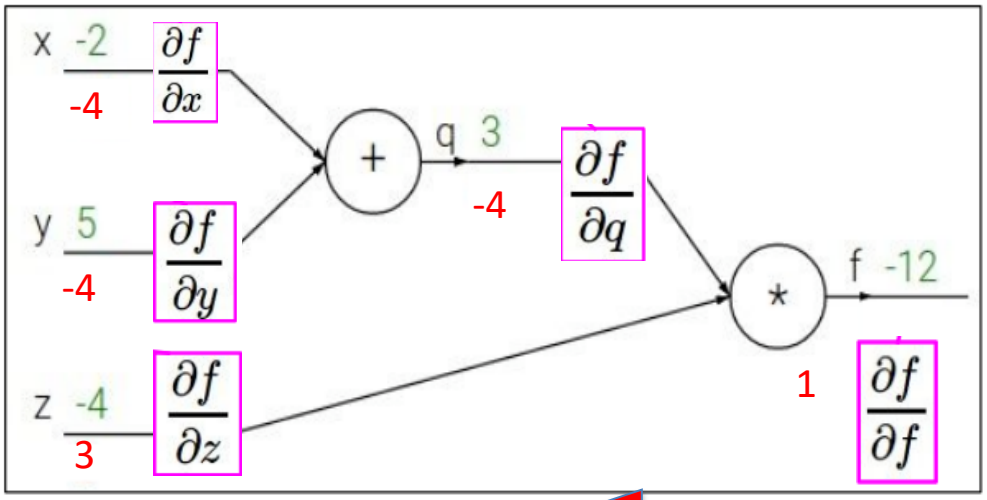
$$f(x,y,z) = (x+y)z$$

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 inputs



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

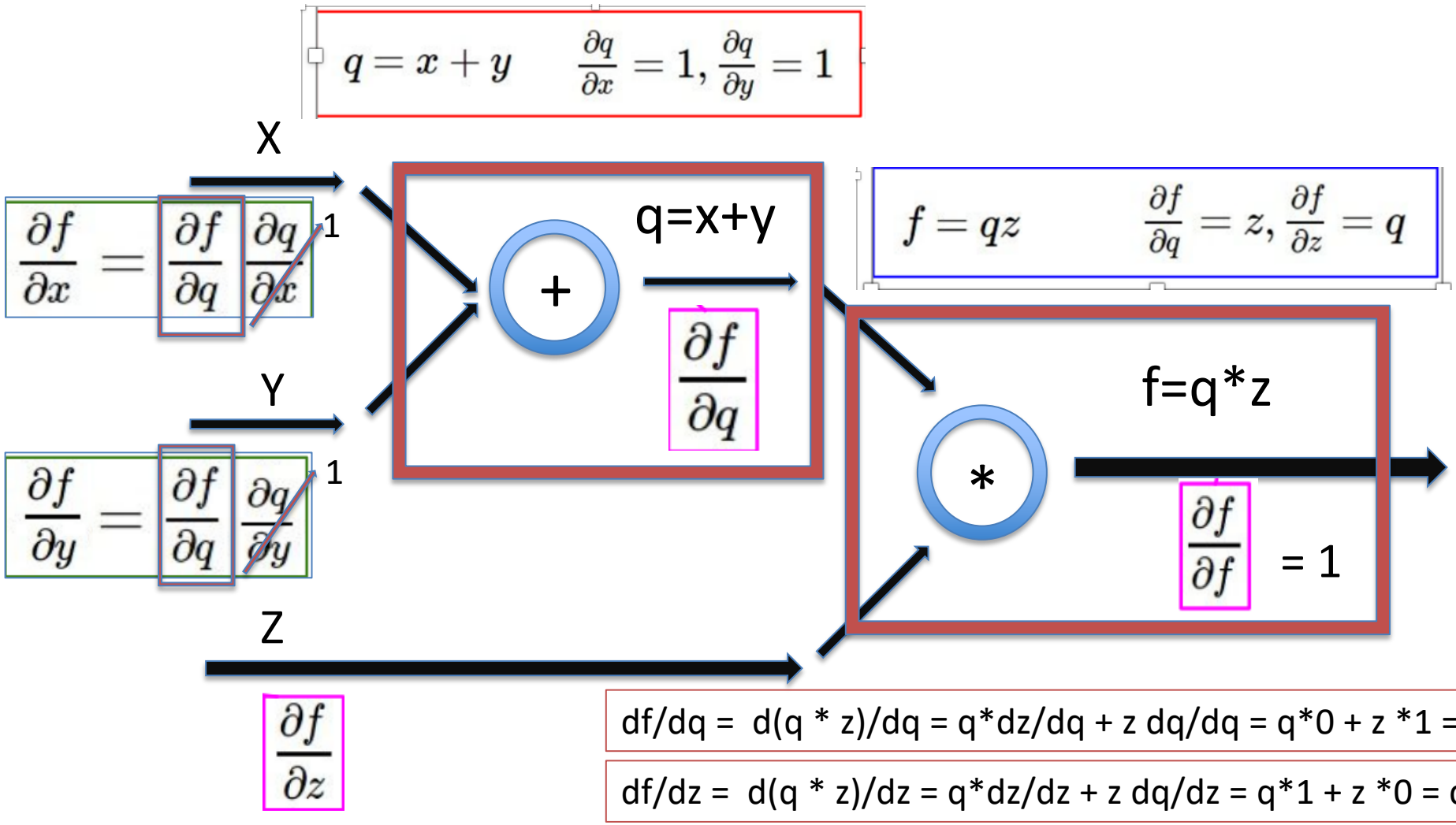
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$ = (z)(1) = -4

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

Computational Graph : backward path calculation

Need Gradient of f wrt to the variable

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



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

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

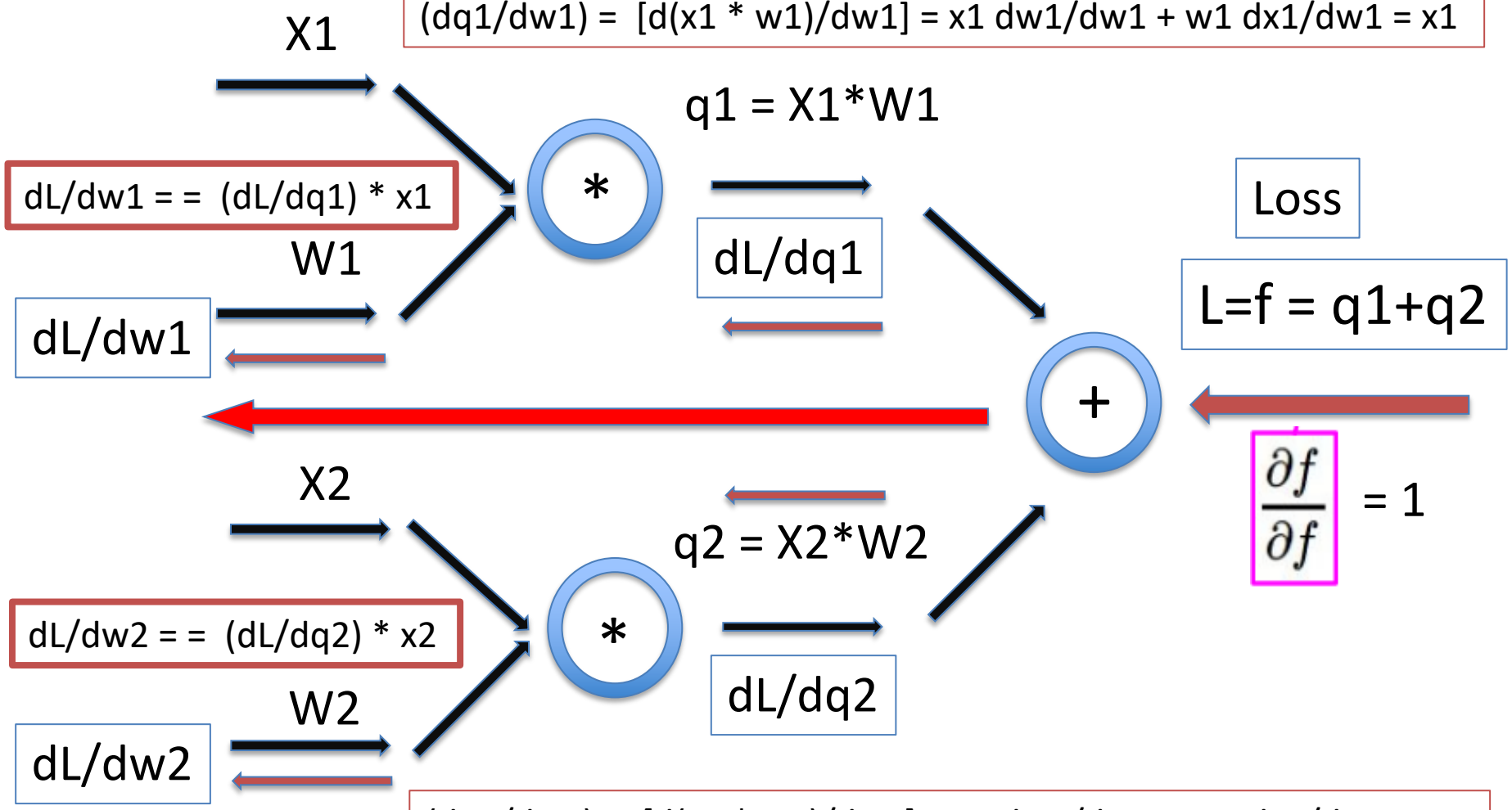
Computational Graph Backpropagation

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

qi = Xi*Wi

dL/dw1 = (dL/dq1) * (dq1/dw1) = [dL/dq1] * [d(x1 * w1)/dq1] = (dL/dq1) * x1

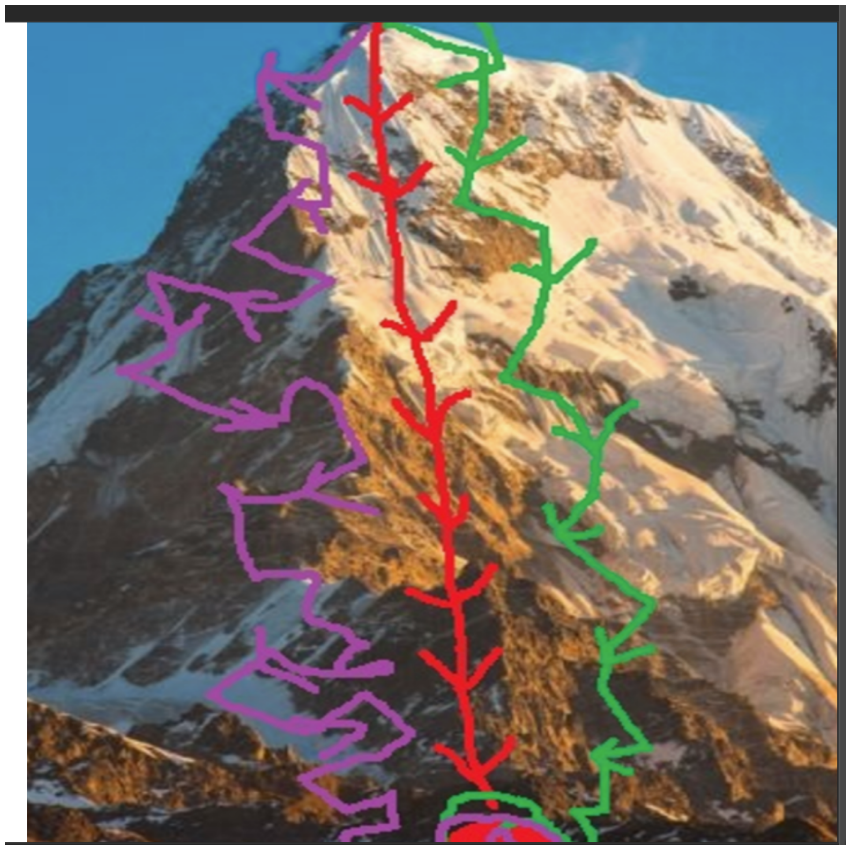
(dq1/dw1) = [d(x1 * w1)/dw1] = x1 dw1/dw1 + w1 dx1/dw1 = x1



(dq2/dw2) = [d(x2 * w2)/dw2] = x2 dw2/dw2 + w2 dx2/dw2 = x2

dL/dw2 = (dL/dq2) * (dq2/dw2) = [dL/dq2] * [d(x2 * w2)/dq2] = (dL/dq2) * x2

Optimization : Derivatives and Gradient Descent



Batch Gradient Descent
Mini-batch Gradient Descent
Stochastic Gradient Descent

Simple story

Derivative is the slope of the tangent line

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

$x = 1$

The slope of the tangent line is 2

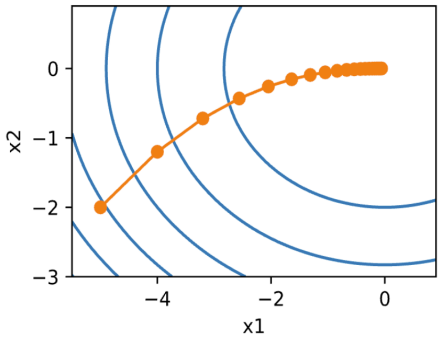
$x)$

Algorithm

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

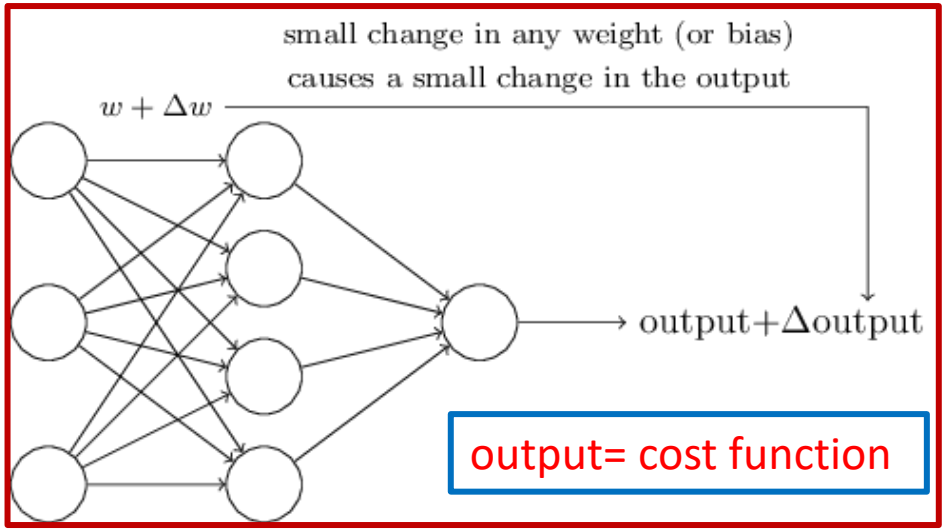
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

- η is called learning rate



Representation of NN Model – Math Story

neuralnetworksanddeeplearning.com



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

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

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

Error and Scheme

neuralnetworksanddeeplearning.com

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

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

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

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

Be negative
Drive the cost to zero

=

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

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

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

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

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

Be negative

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

Just Choose

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

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

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

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

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

Mini-Batch Stochastic Gradient Descent

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

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

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

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

Stochastic Gradient Descent

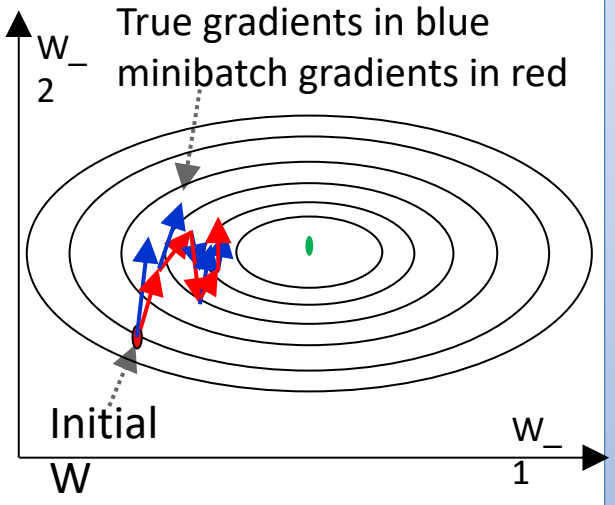
- ✓ Neural network is an optimization process to find a set of parameters (W, weights) that ensure the prediction function f(x, W) to be as close as possible to the true solution (labeled input) y for any input x
- ✓ Use gradient descent to find the weights *w* and biases *b* which minimize the cost function, C.
- ✓ To compute the gradient ∇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, and thus learning occurs slowly.
- ✓ A way is to use ***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 this small sample, it turns out that 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.
- ✓ Another randomly chosen mini-batch are selected and trained, until all the exhausted the training inputs are used. It is said to complete an *epoch* (iteration) of training, then more iteration.

Optimization: Batch SGD

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

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

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



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

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

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

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

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

GD Optimization Algorithms

Gradient Decent (full batch)

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

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

Stochastic Gradient Decent

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

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

Mini-batch Stochastic Gradient Decent

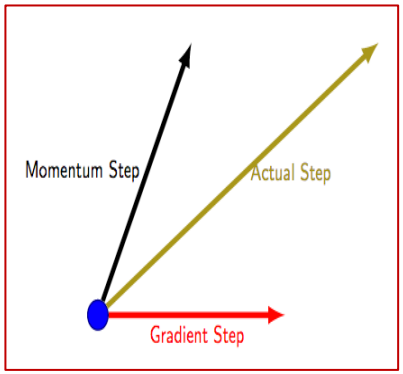
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

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

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

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

Gradient-based Optimization with Momentum



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

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

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

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

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

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

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

Algorithm 2 Stochastic Gradient Descent with Momentum

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

Momentum

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

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

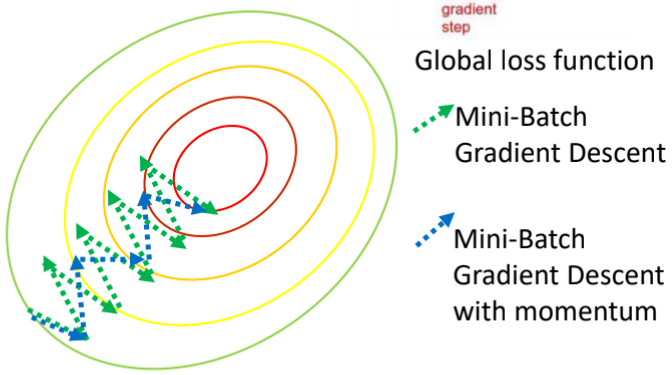
The velocity is basically an exponentially decaying average of the step size:

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

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

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

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



Adaptive learning rates + momentum

1. ADAM (Adaptive Moment Estimation):

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

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

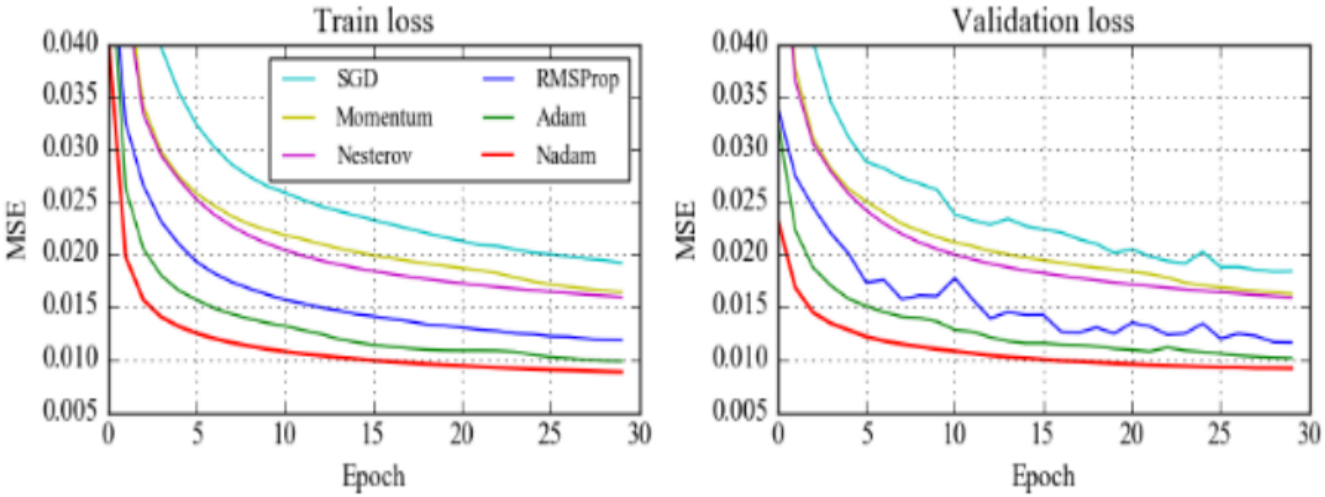
2. NADAM(Nesterov-accelerated Adaptive Moment Estimation):
   1. Use ADAM with Nestrov momentum

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

Optimizer : ADAM

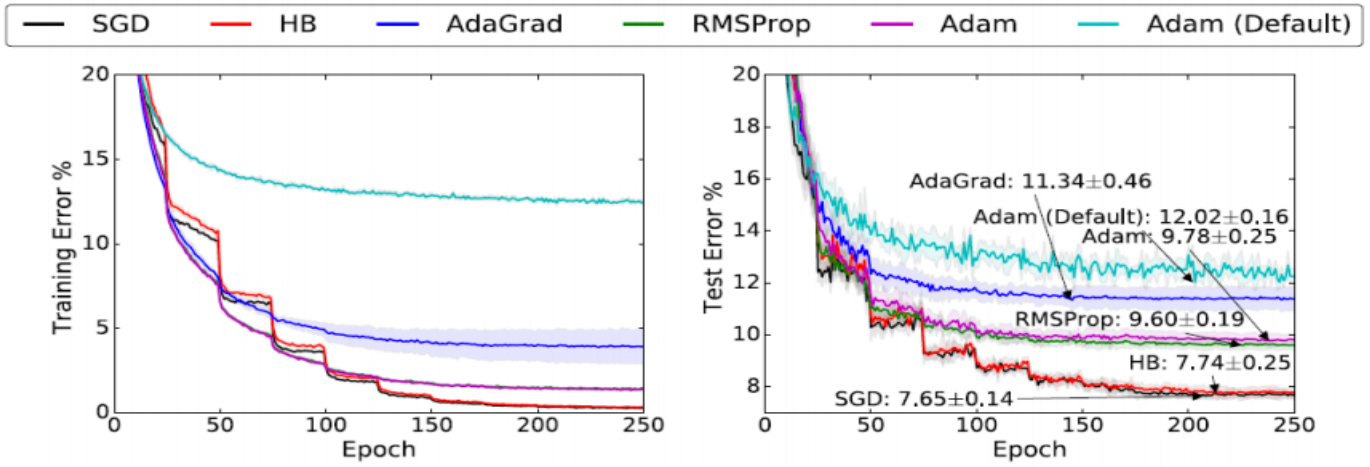
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

https://mlfromscratch.com/optimizers-explained/#/



2016

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

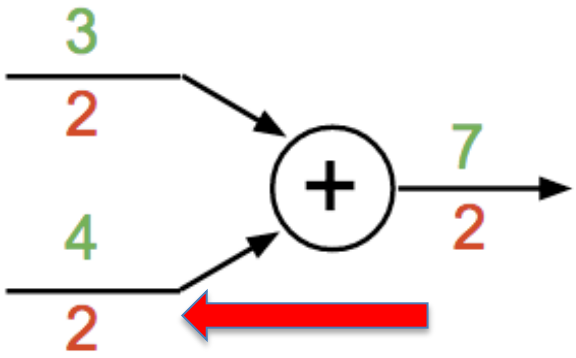


Wilson, Ashia C., et al. "The marginal value of adaptive gradient methods in machine learning." *arXiv preprint arXiv:1705.08292* (2017).

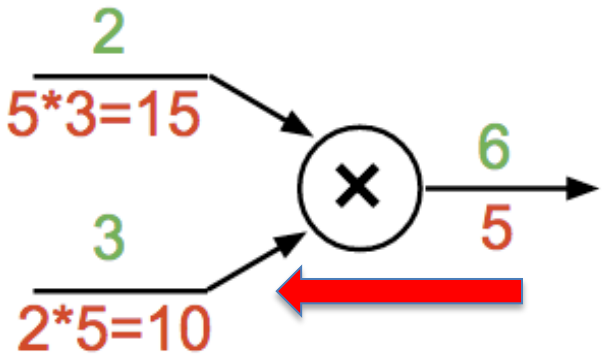
Forward and Backpropagation Code

Forward path : backward path

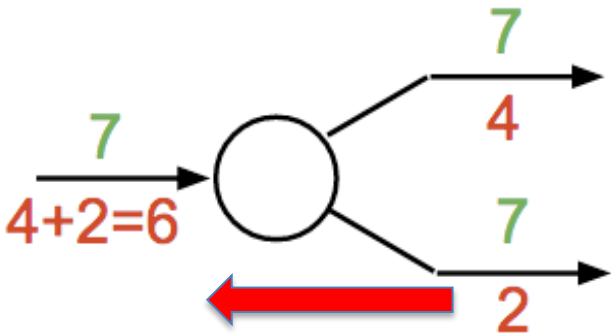
add gate: gradient distributor



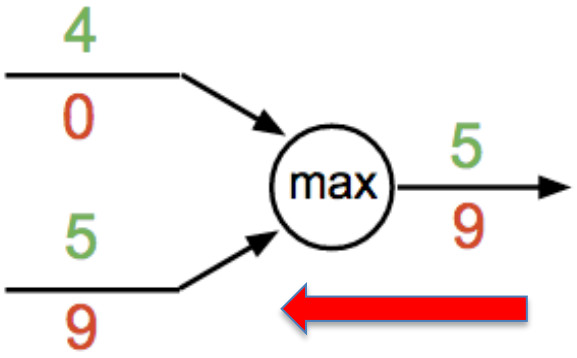
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router



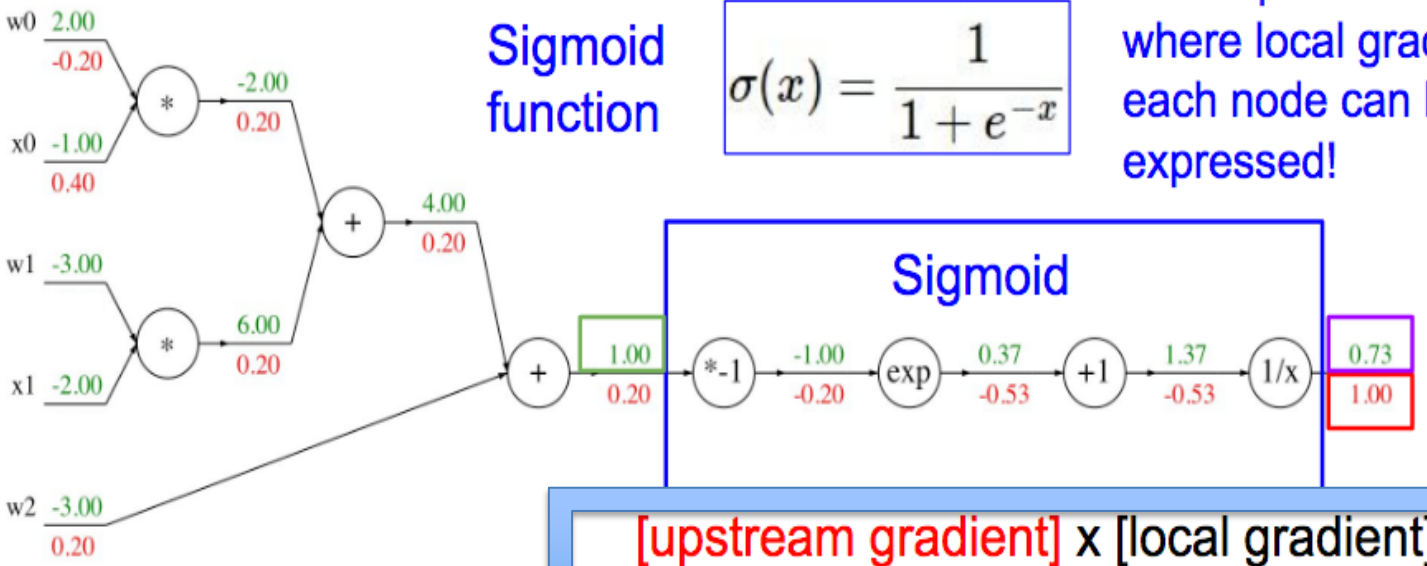
http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

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

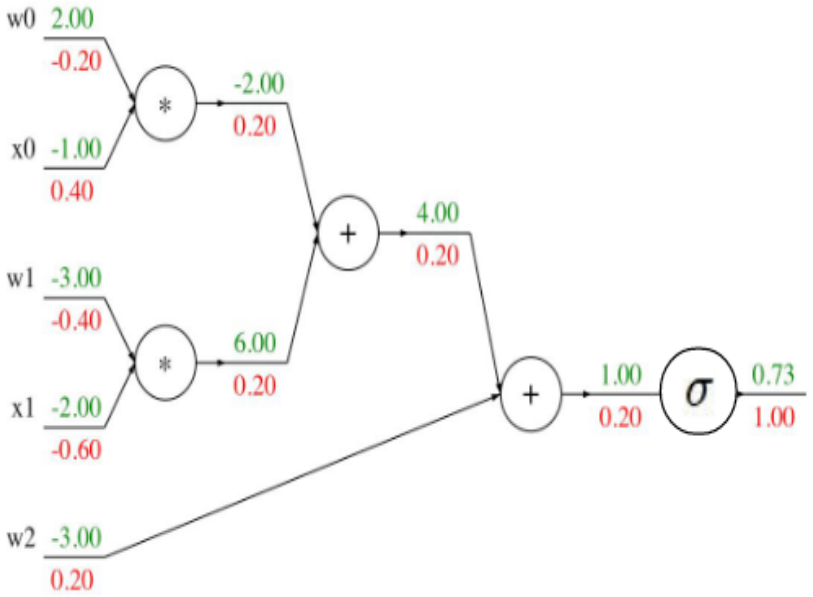
Sigmoid local gradient:

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

Forward and Backpropagation Code

Forward path : backward path

Backprop Implementation: "Flat" code



Forward pass:
Compute output

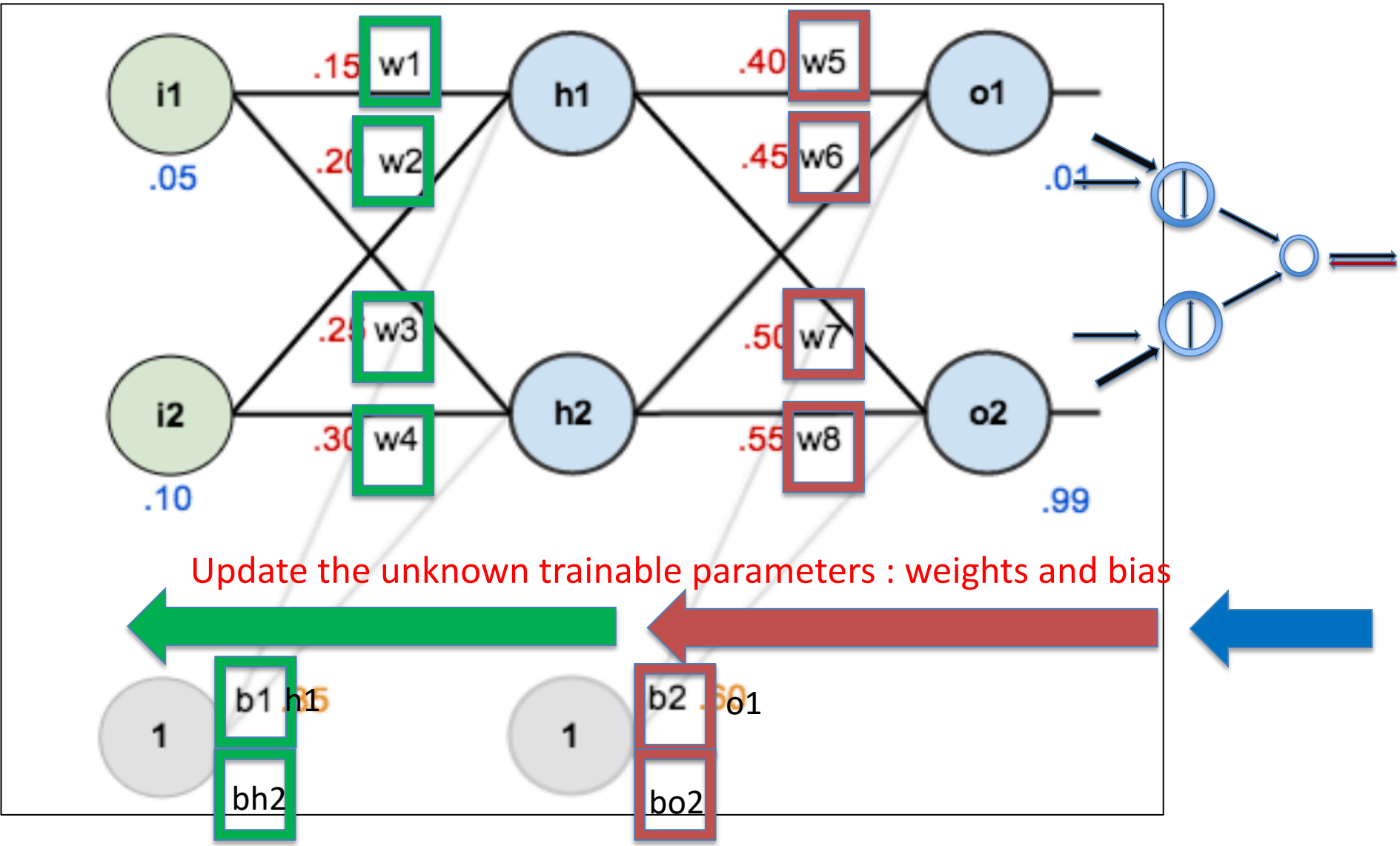
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

Backward pass:
Compute grads

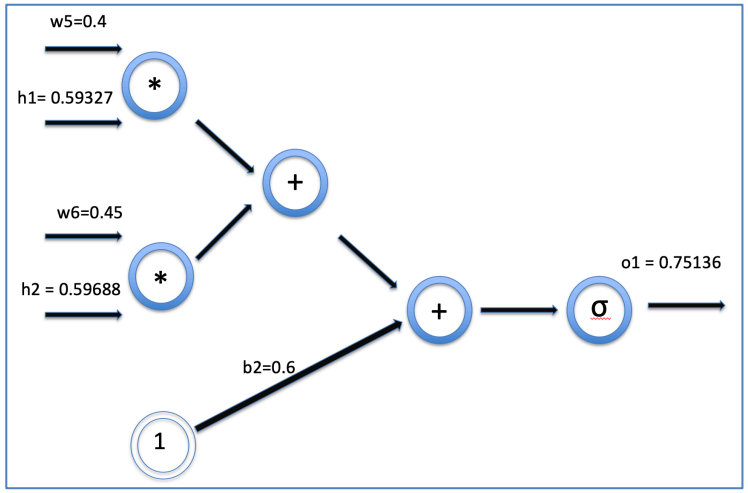
```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

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

Multilayer Perceptron : Backward Path Calculation



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



target=0.01 | o1=0.75136 | Eo1=0.27481

Forward Path to Objective Function

target1 = 0.01

o1 = 0.75136

Cost f

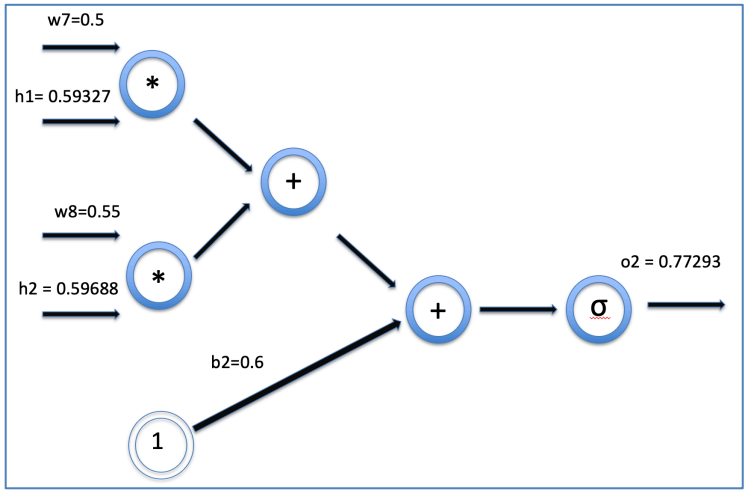
Eo1 = 0.27481

ET = 0.298371

grad ET = 1.0

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

Etotal = Eo1 + Eo2 = 0.298371109



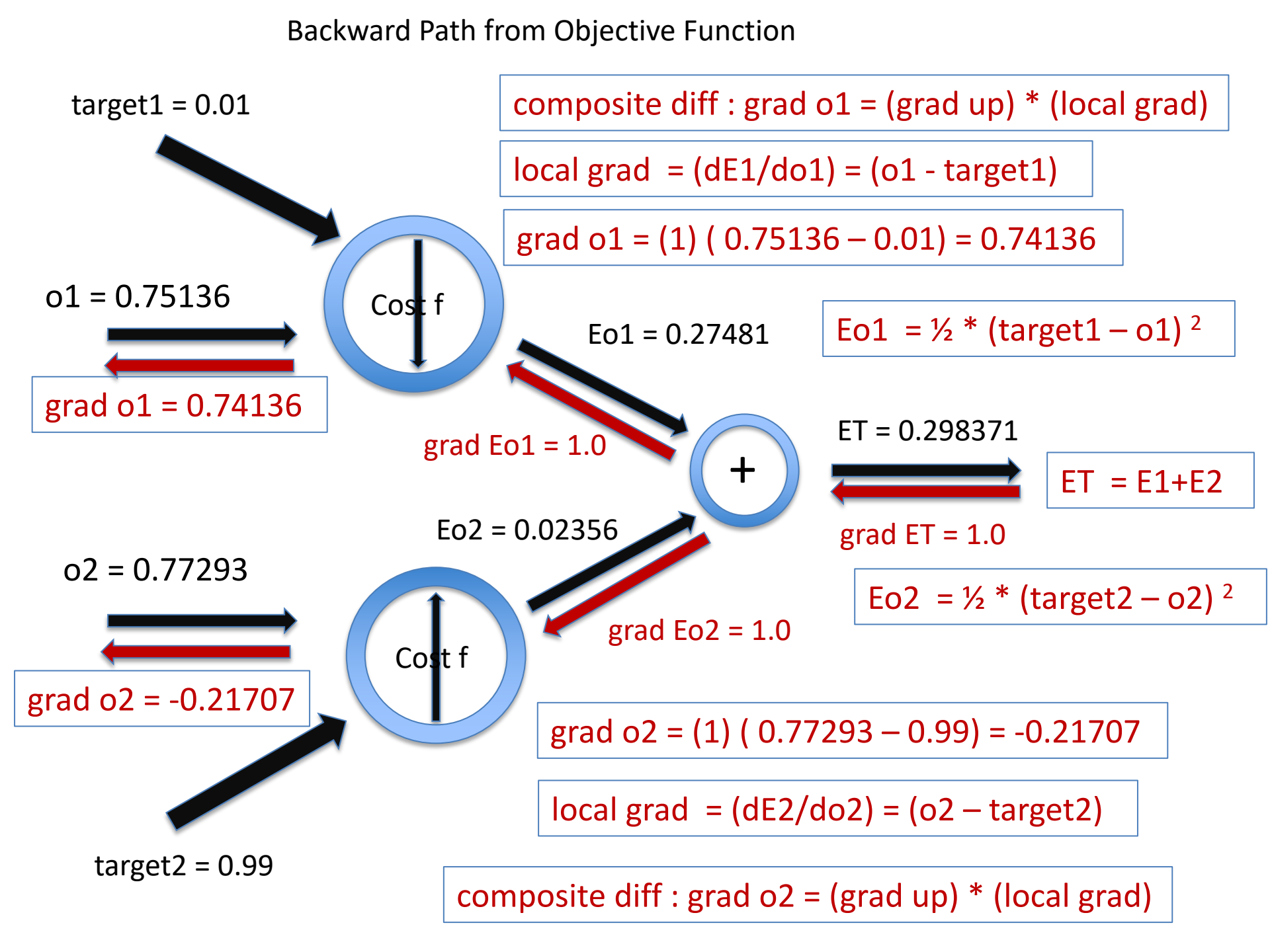
o2 = 0.77293

Cost f

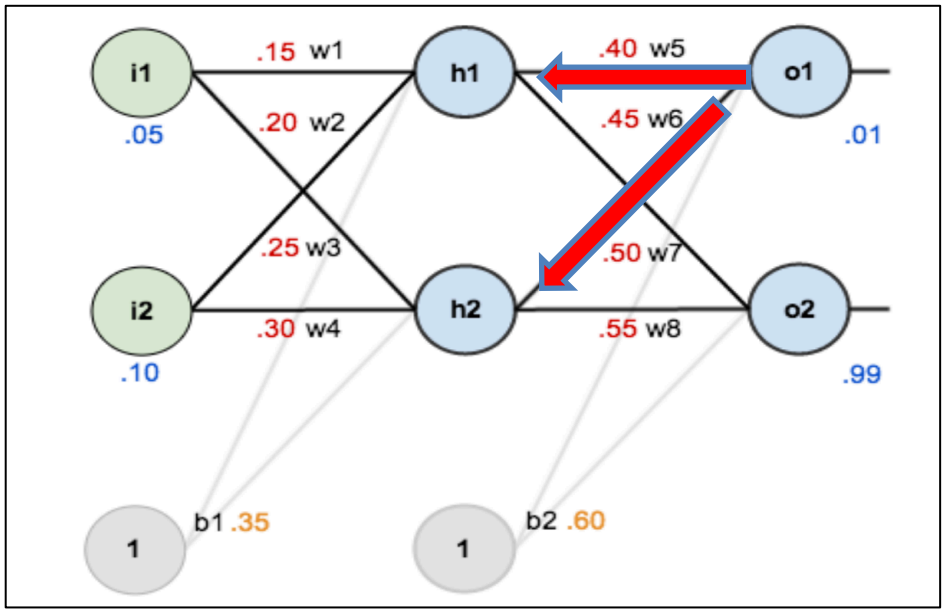
Eo2 = 0.02356

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

*

+

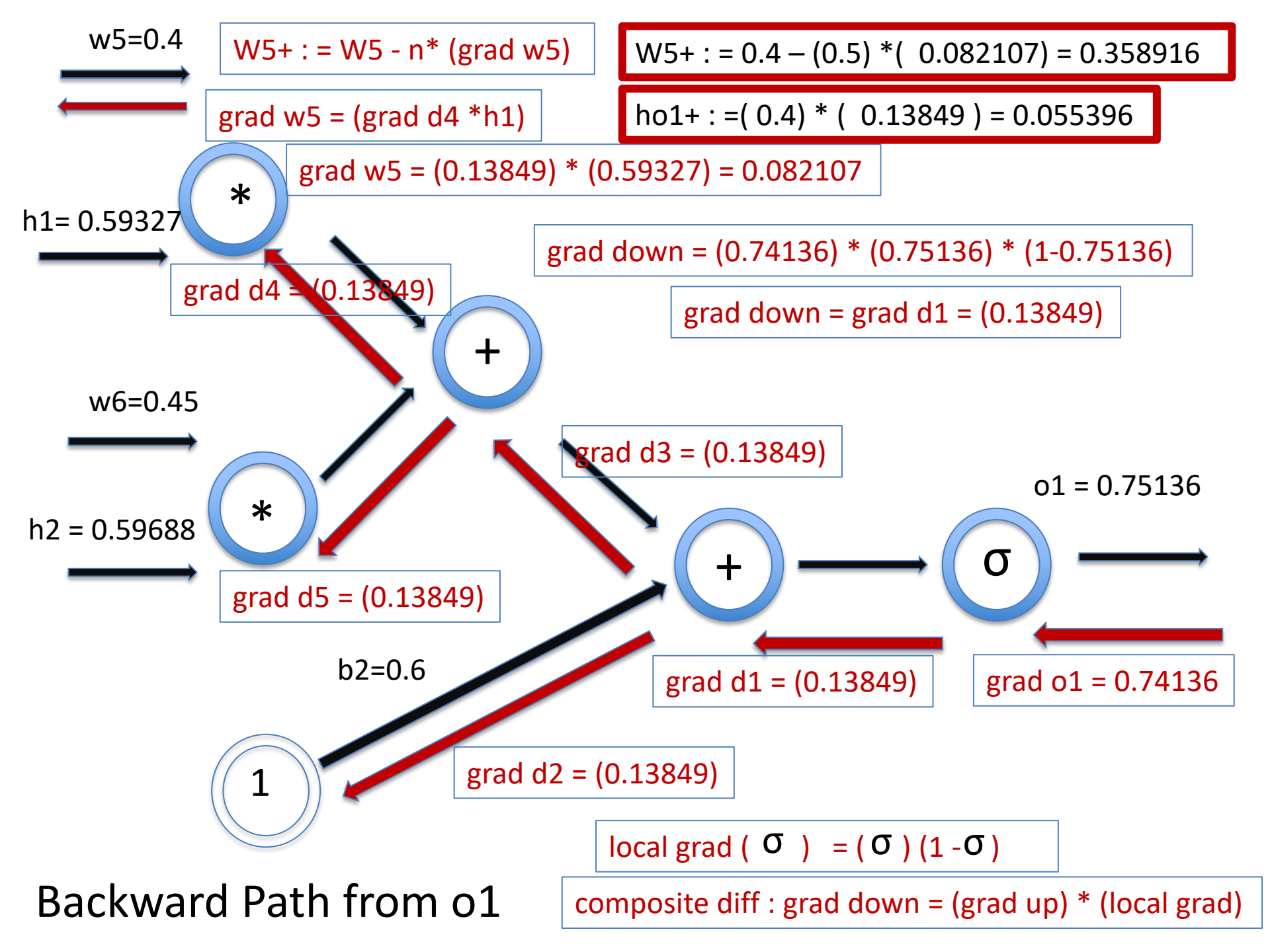
+

σ

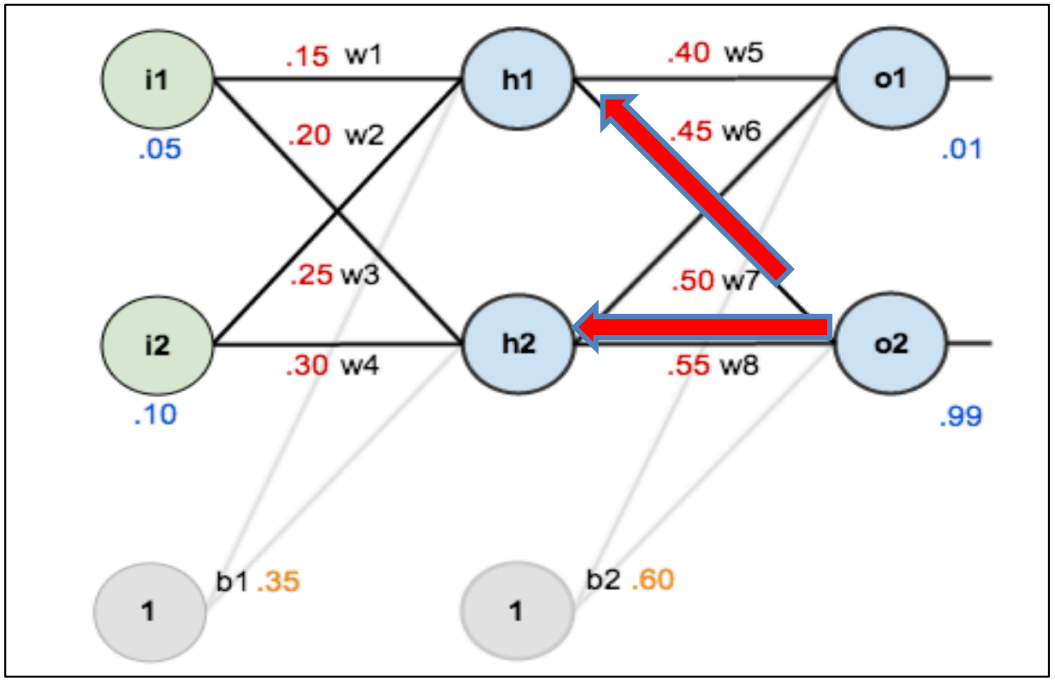
o1 = 0.75136

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

*

+

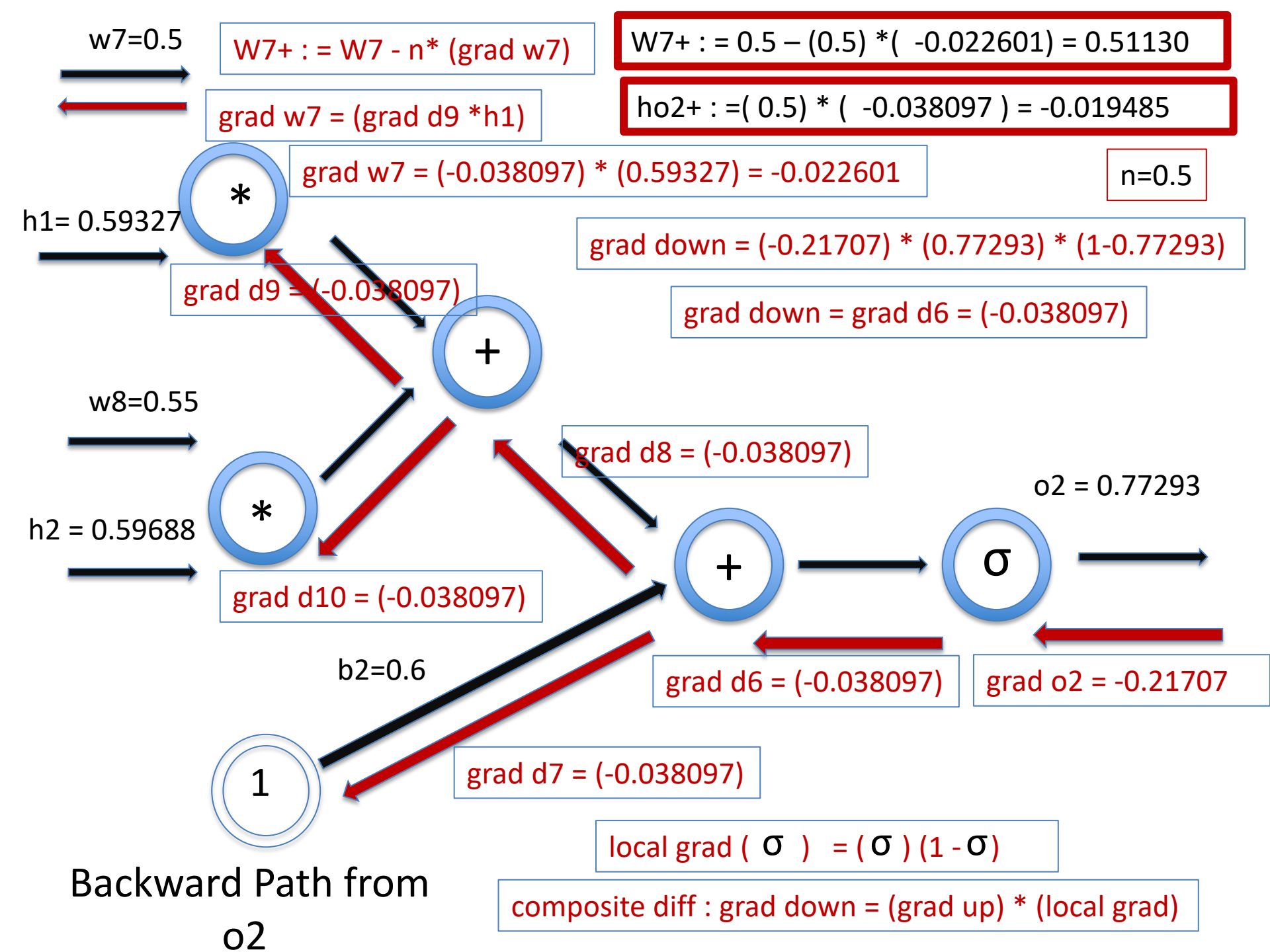
+

σ

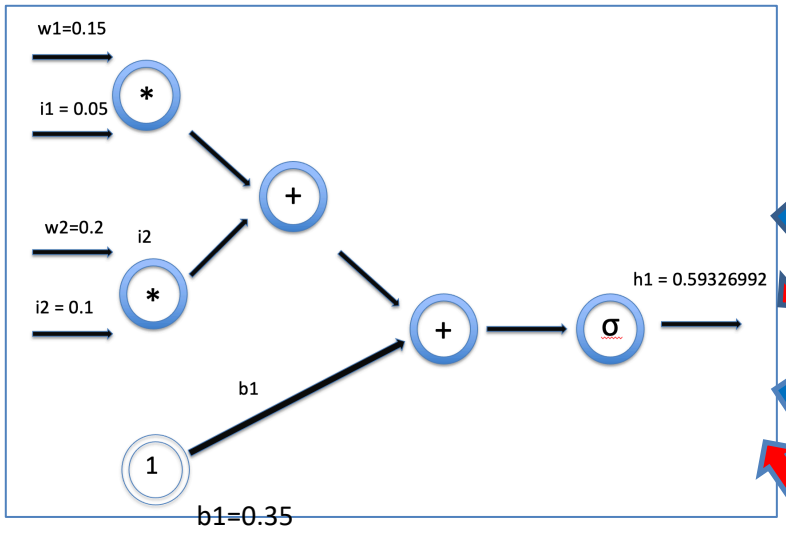
o2 = 0.77293

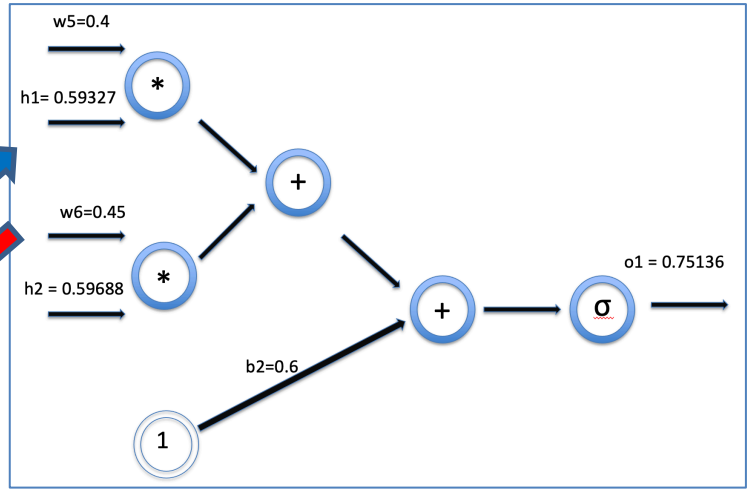
B2=0.6

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



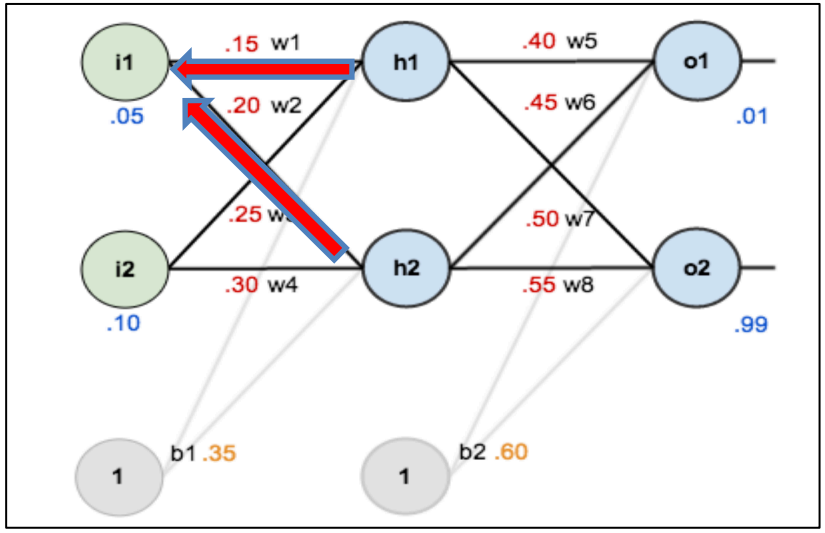


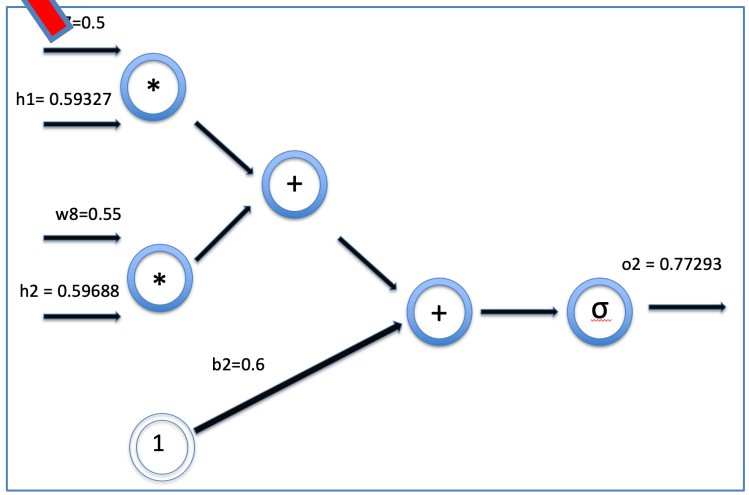
target=0.01

o1=0.75136

Eo1=0.27481

Etotal = Eo1 + Eo2 = 0.298371109

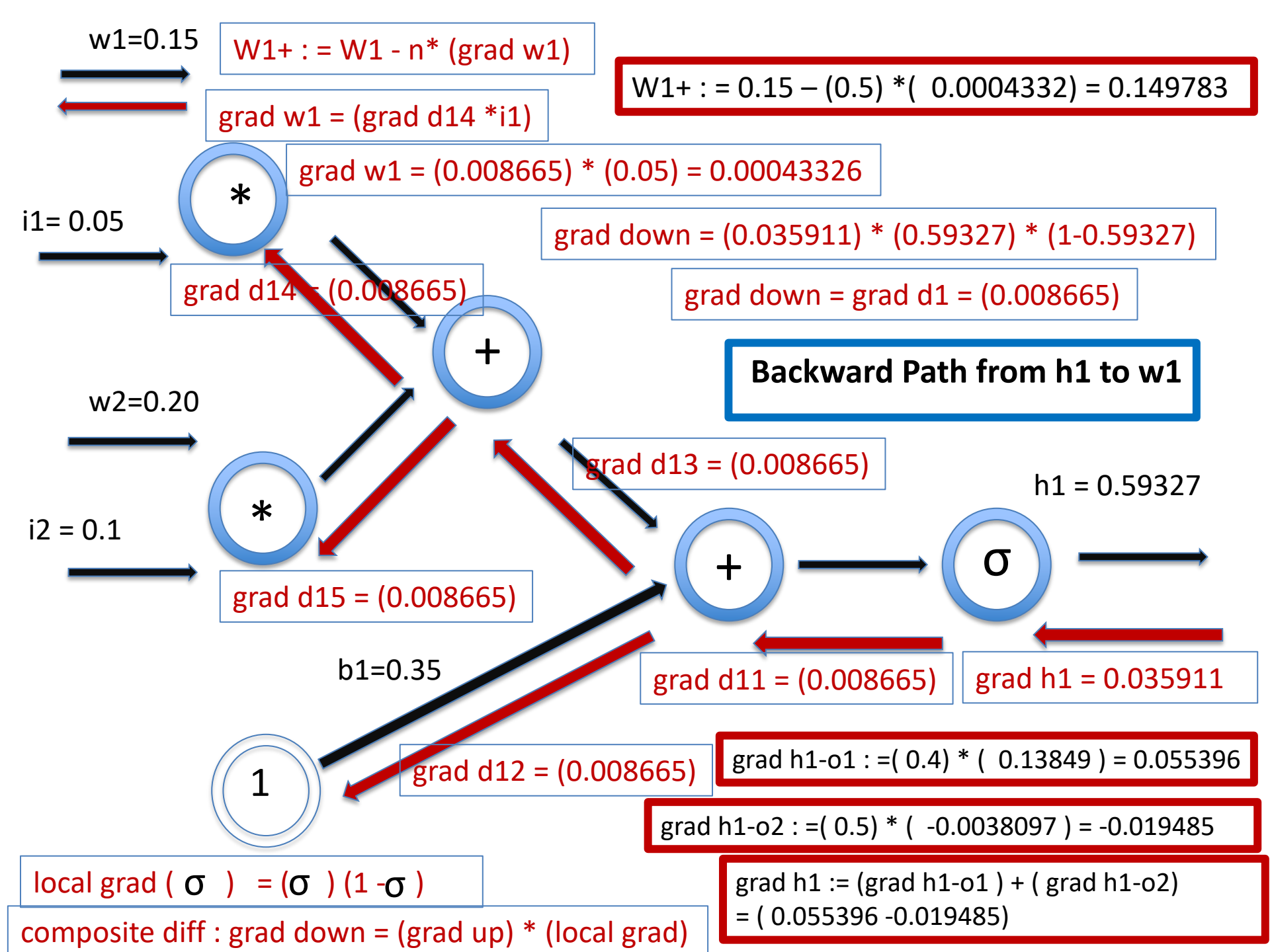




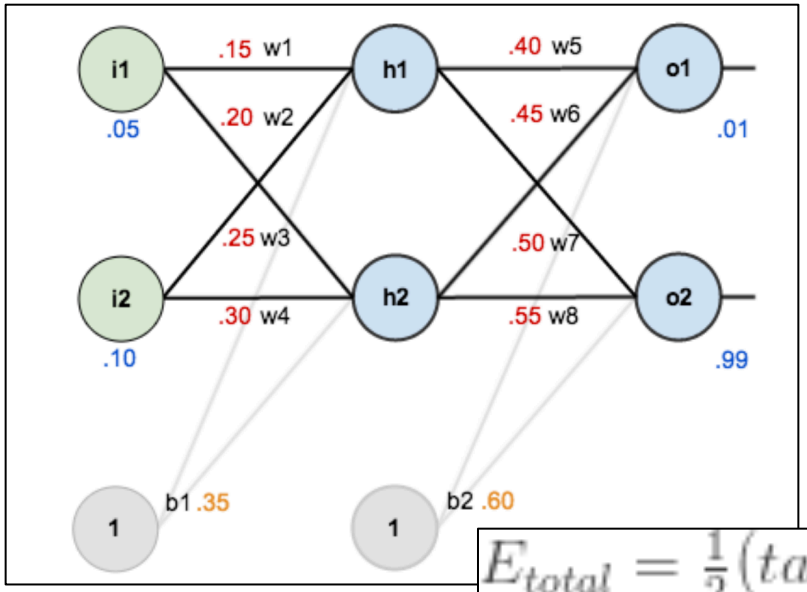
Eo2=0.02356

o2=0.77293

target=0.99

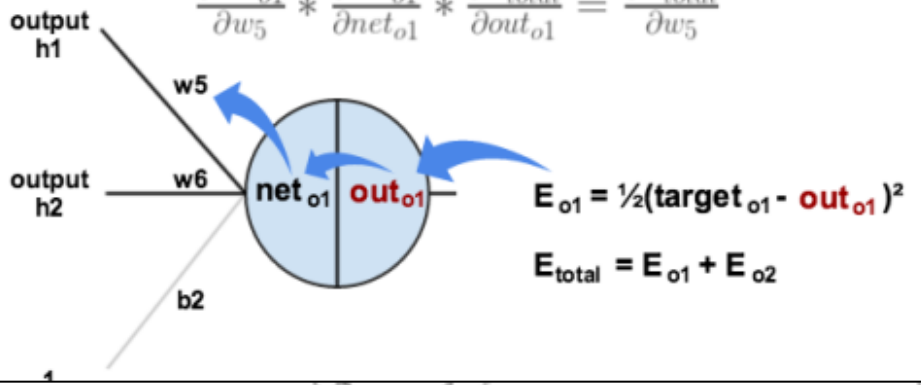

w1=0.15
W1+ : = W1 - n* (grad w1)
W1+ : = 0.15 – (0.5) *(0.0004332) = 0.149783
grad w1 = (grad d14 *i1)
grad w1 = (0.008665) * (0.05) = 0.00043326
*
i1= 0.05
grad down = (0.035911) * (0.59327) * (1-0.59327)
grad d14 = (0.008665)
grad down = grad d1 = (0.008665)
+
Backward Path from h1 to w1
w2=0.20
grad 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



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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

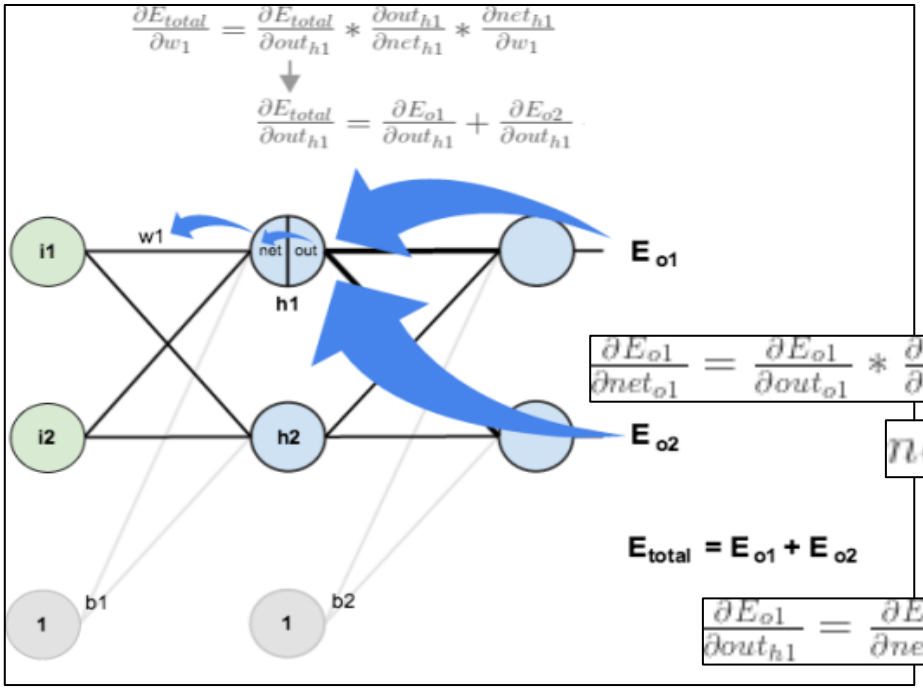
$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

Backpropagation Example

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$w_2^+ = 0.19950143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

```
# 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
```

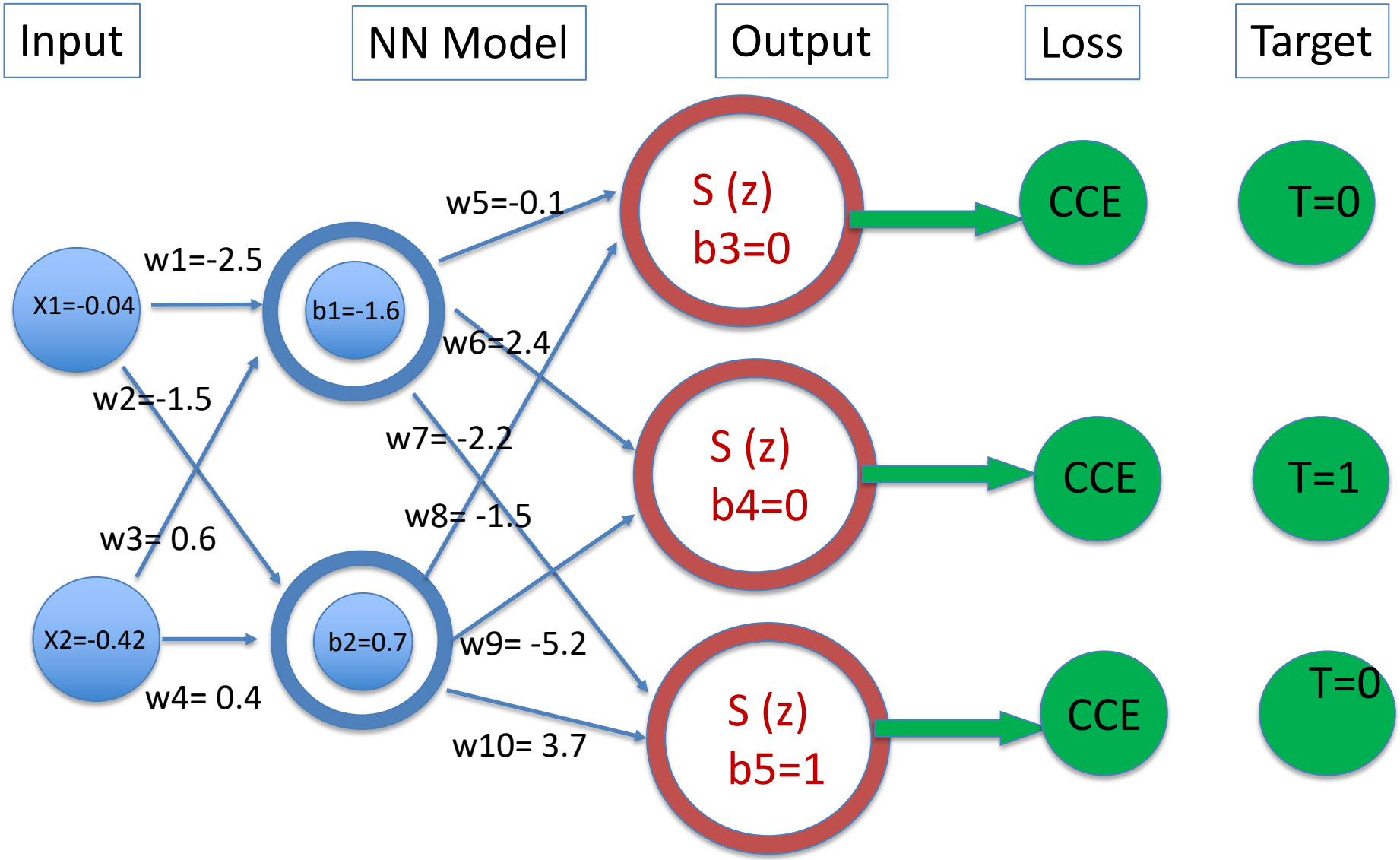
```
grad_h1 = w5*grad_d1 + w7*grad_d2
print("grad_h1: "+str(grad_h1))
grad_d11 = grad_h1*h1*(1-h1)
print("grad_d11: "+str(grad_d11))

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

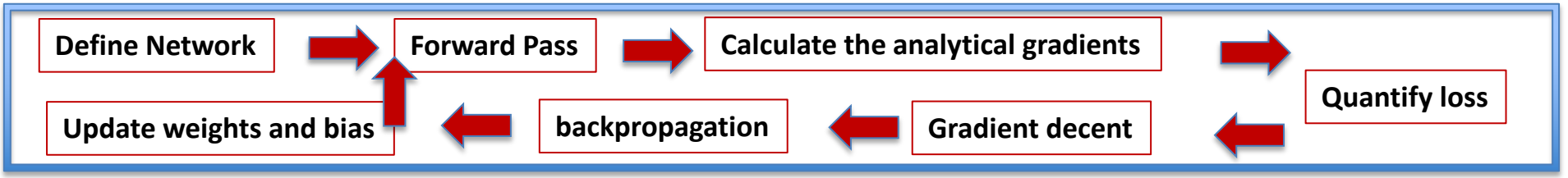
Exercises : Represent the following NN model as a computational graph and build a python code to update w and b



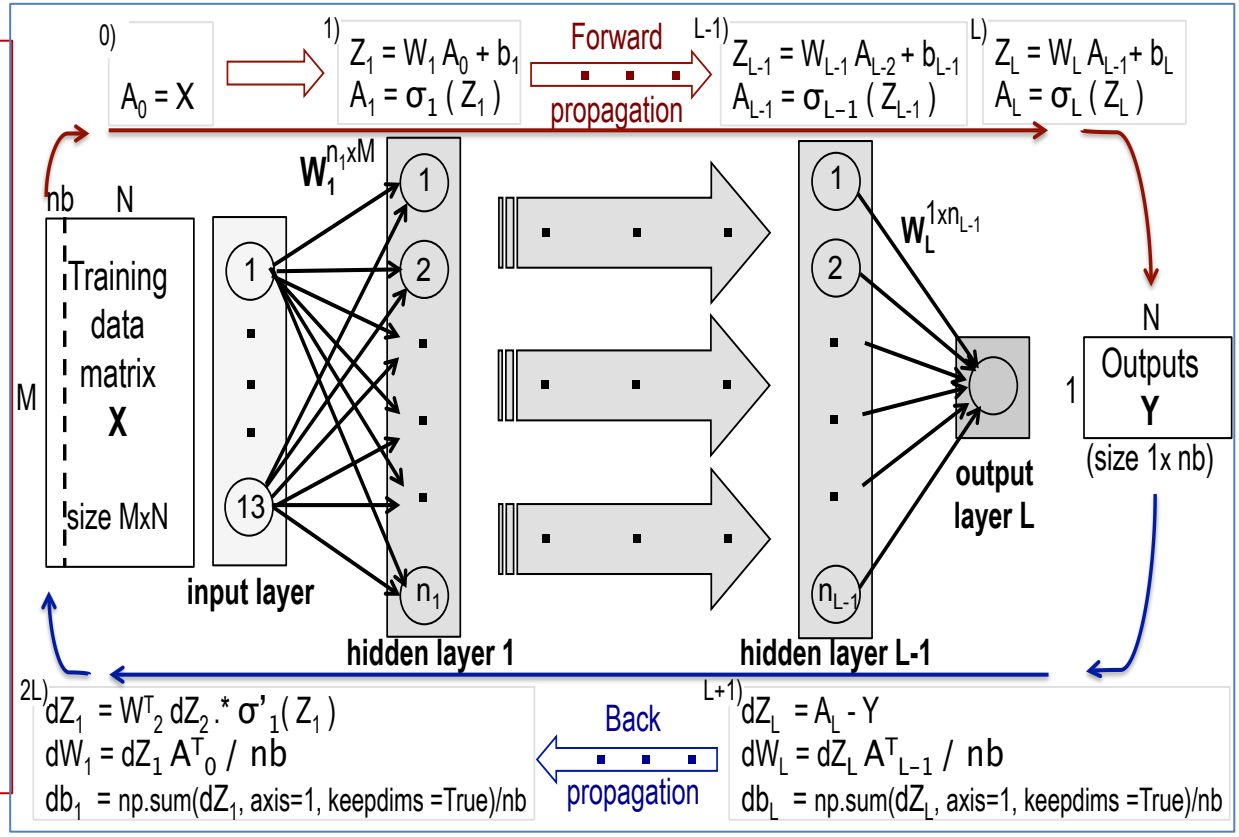
Categorical (Softmax) Cross Entropy Loss

Summary : Flow of NN

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

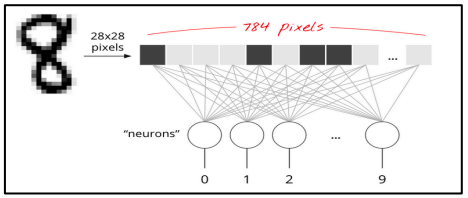



```python
import numpy as np
from numpy.random import randn

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
```

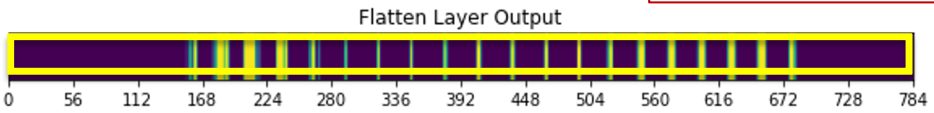


Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



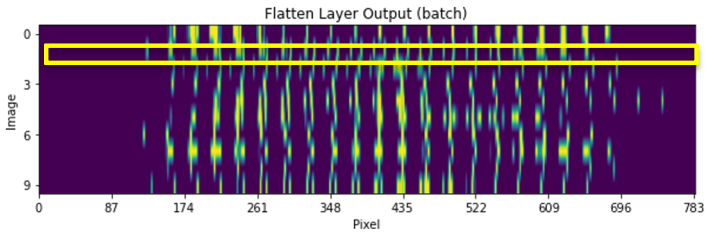
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

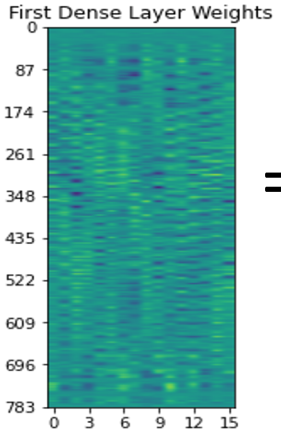
16 neurons
784 x 16 W matrix

10 x 16 output matrix

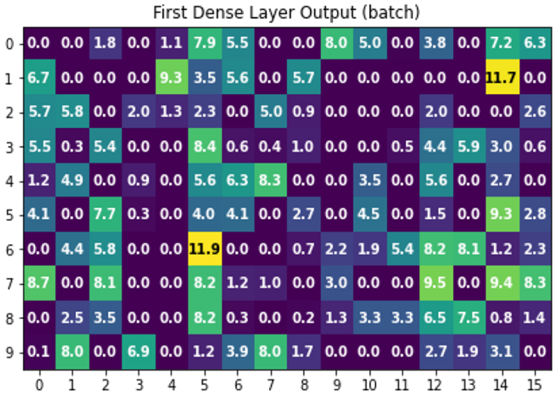
Input to 2nd layer



*

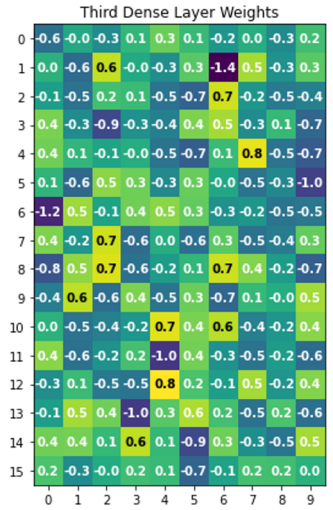


=



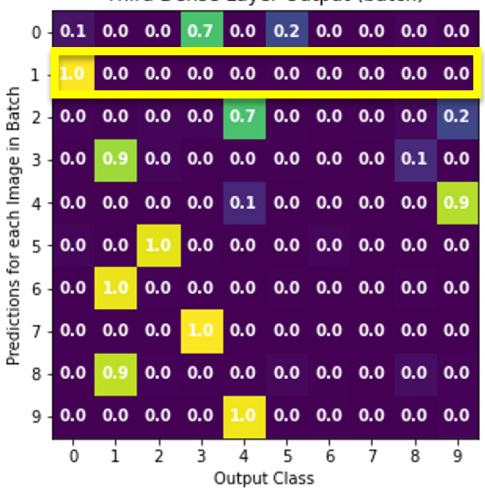
10 x 16 matrix

*

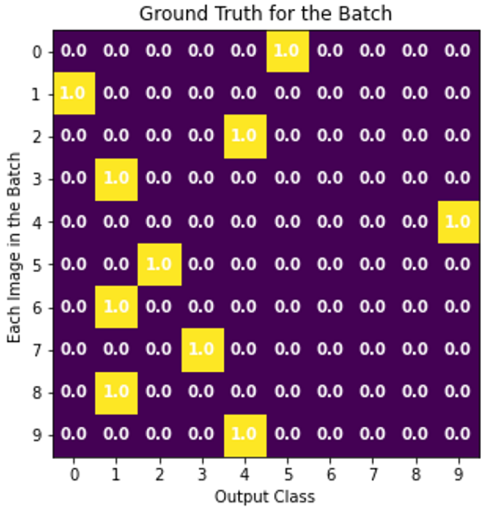


16 neurons
16 x 10
W matrix

=



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/

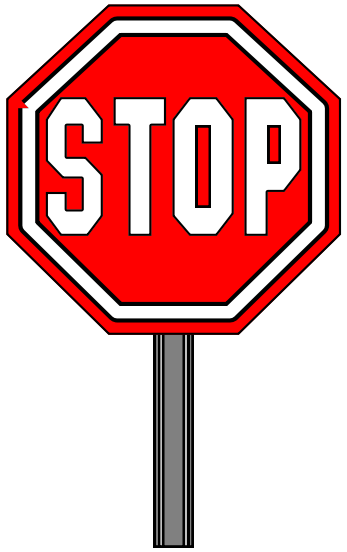
10) https://oneapi-src.github.io/oneDNN/

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!