DEEP LEARNING

Introduction to Deep Learning and Deep Network Learning Issues

Adrian Horzyk

horzyk@agh.edu.pl

AGH

AGH University of Science and Technology Krakow, Poland

Tasks for Deep Neural Networks

We use Deep Neural Networks for specific group of issues:

- Classification (of images, signals etc.)
- Prediction (e.g. price, temperature, size, distance)
- Recognition (of speech, objects etc.)
- Translation (from one language to another)
- Autonomous behaviors (driving by the autonomous cars, flying of the drones…)
- Clustering of objects (grouping them according to their similarity)
- etc.

using supervised or unsupervised training of such networks.

We have to deal with structures and unstructured data:

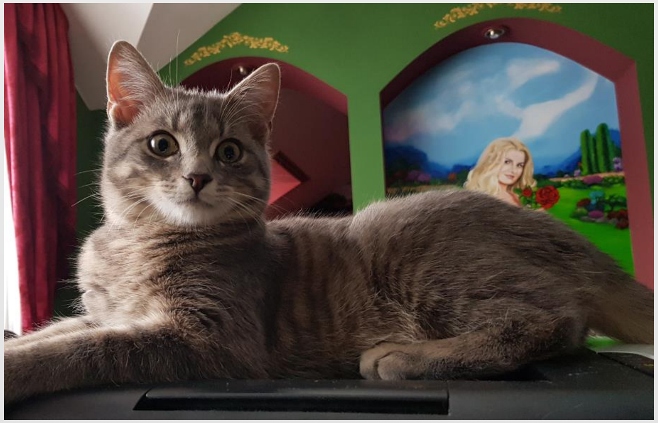
Structured data are usually well-described by the attributes and collected in data tables (relational databases), while **unstructured data** are images, (audio, speech) signals, (sequences of) texts (corpora).

Binary Classification

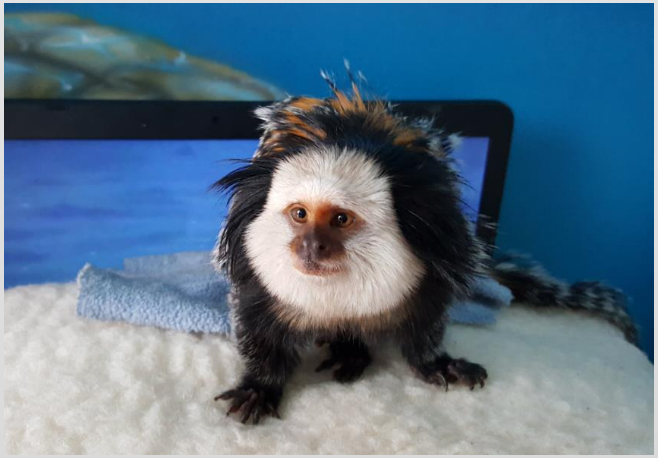
In binary classification, the result is describe by two values:

- **1 – when the object of the class was recognized (e.g. is a cat),**
- **0 – when the object was not recognized as belonging to the given class (e.g. is not a cat).**

Example:



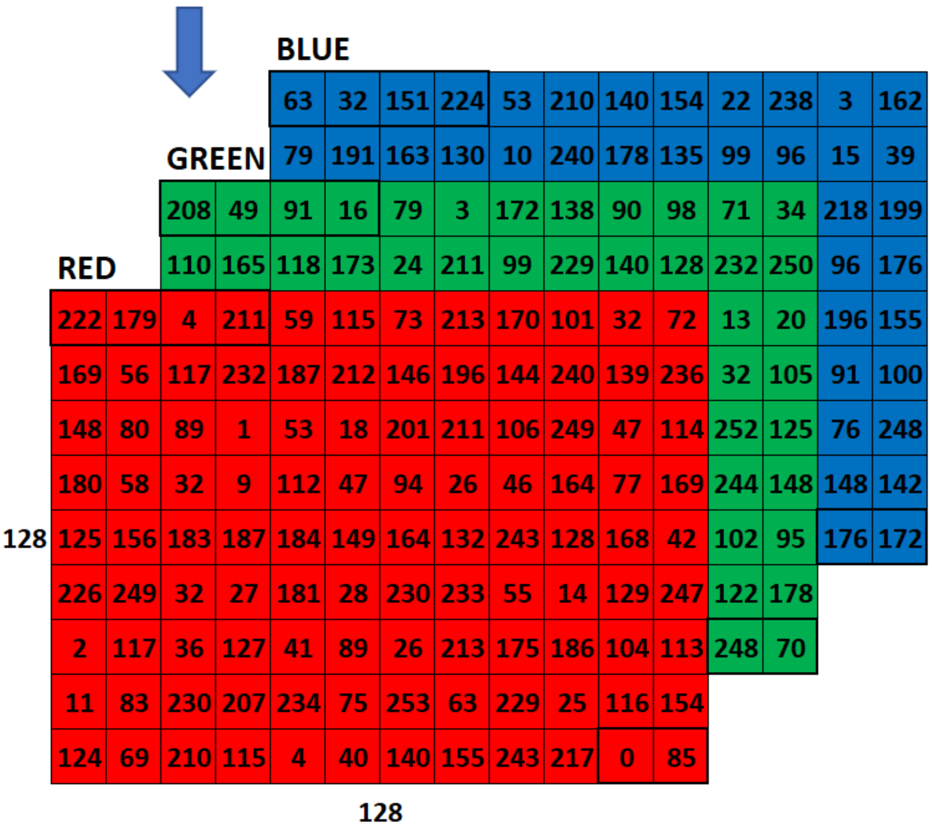
Is a cat (1)



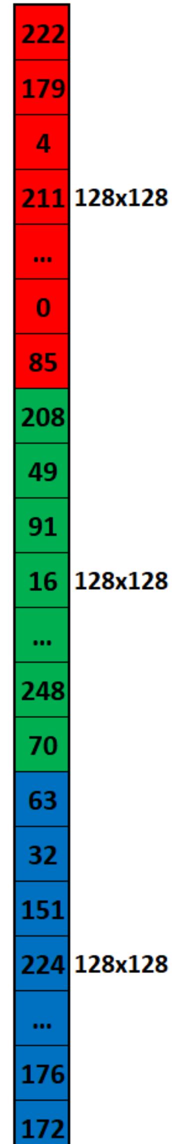
Is not a cat (0)

Image Representation

Images are reprezented as
a combination of three colours
reprezented by three matrices
that store the intensities of
these colours (Red, Green, and Blue):



x n_x = 128x128x3= 49152 is the dimension of vector x



In the binary classification tasks,
input vectors are assigned
to one of the two classes 0 or 1
that is the output value y of
the classification process.

So, we have to create
the transformation
$x \rightarrow y$
and denote the training example
as pairs (x, y)
where
$x \in R^{n_x}$
$y \in \{0, 1\}$

Training Examples

Training examples are represented as a set of m pairs:

$$(\boldsymbol{X}, \boldsymbol{Y}) = \{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \dots, (\boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)})\}$$

where

$\boldsymbol{m}$ – is the number of examples

$\boldsymbol{m}_{train}$ – is the number of training examples

$\boldsymbol{m}_{test}$ – is the number of test examples

For vectorization, we stack the training examples in the matrix X as well as outputs Y:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^{(1)} & \cdots & \boldsymbol{x}_1^{(m)} \\ \vdots & \ddots & \vdots \\ \boldsymbol{x}_{n_x}^{(1)} & \cdots & \boldsymbol{x}_{n_x}^{(m)} \end{bmatrix} \in \mathbb{R}^{\boldsymbol{n_x} \times \boldsymbol{m}} \qquad \boldsymbol{Y} = [\boldsymbol{y}^{(1)} \quad \dots \quad \boldsymbol{y}^{(m)}] \in \mathbb{R}^{\boldsymbol{1} \times \boldsymbol{m}}$$

When we use the Python command to read or set the shape, the notation is:

$$\boldsymbol{X.shape} = (\boldsymbol{n_x}, \boldsymbol{m}) \qquad \boldsymbol{Y.shape} = (\boldsymbol{1}, \boldsymbol{m})$$

Logistic Regression

For the given x, we get the output prediction $\hat{y} = P(y = 1|x)$
where y is the desired output that will be trained using parameters:

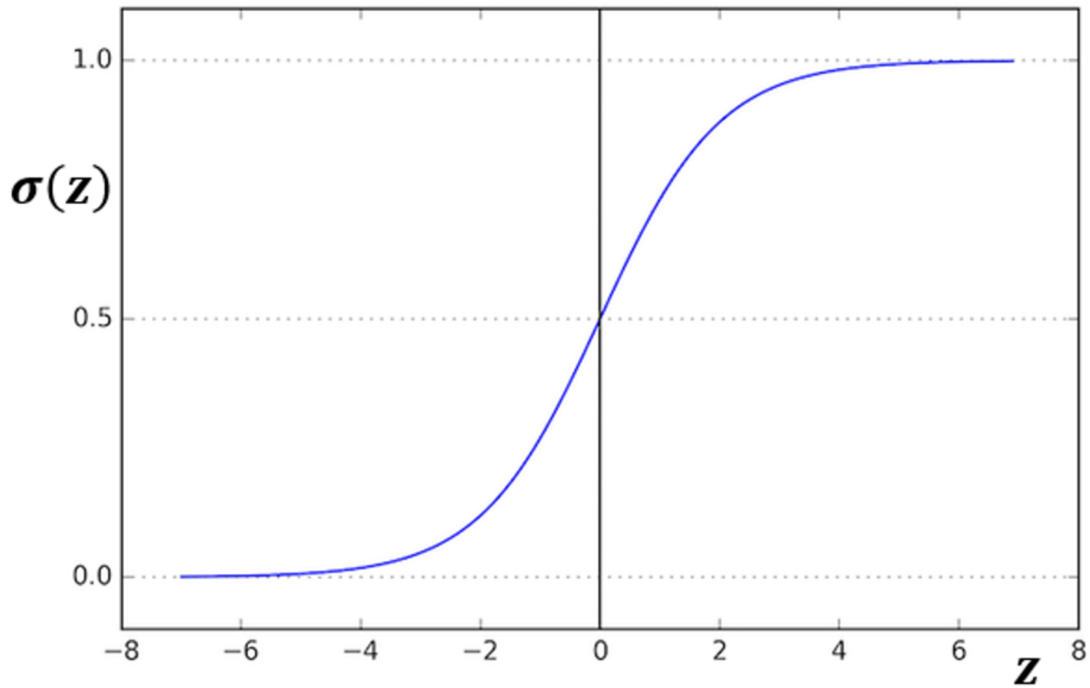
$\boldsymbol{w} \in \mathbb{R}^{n_x}$

$\boldsymbol{b} \in \mathbb{R}$

computing the output in the following way:

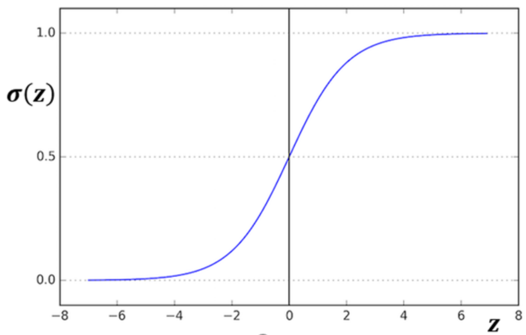
$$\hat{y} = \sigma\left(w^T x + b\right) = \sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1)$$

where σ is a sigmoid function:



Computing Sigmoid Function

We use numpy vectorization to compute sigmoid and sigmoid_derivative for any input vector z:



$$\text{For } z \in \mathbb{R}^n,\ sigmoid(z) = sigmoid\begin{pmatrix} z_1 \\ z_2 \\ \dots \\ z_n \end{pmatrix} = \begin{pmatrix} \frac{1}{1+e^{-z_1}} \\ \frac{1}{1+e^{-z_2}} \\ \dots \\ \frac{1}{1+e^{-z_n}} \end{pmatrix} \tag{1}$$

$$sigmoid\_derivative(z) = \sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{2}$$

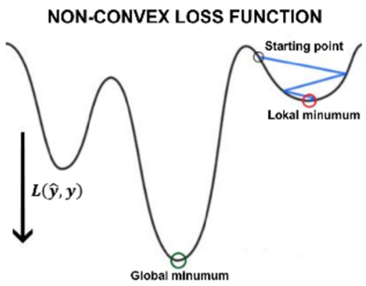
```python
import numpy as np # this means you can access numpy functions by writing np.function() instead of numpy.function()

def sigmoid(z):
    a = 1 / (1 + np.exp(-z)) # Compute the sigmoid of z, where z can be a scalar or numpy array of any size
    return a

def sigmoid_derivative(z):
    a = sigmoid(z)          # Compute the gradient (slope, derivative) of the sigmoid function with respect to its input z.
    dJa = a * (1 - a)
    return dJa
```

```python
z = np.array([-2,-1,0,1, 2])
print ("sigmoid(z) = " + str(sigmoid(z)))
print ("sigmoid_derivative(z) = " + str(sigmoid_derivative(z)))
```

```
sigmoid(z) = [0.11920292 0.26894142 0.5        0.73105858 0.88079708]
sigmoid_derivative(z) = [0.10499359 0.19661193 0.25       0.19661193 0.10499359]
```

Logistic Regression Cost Function

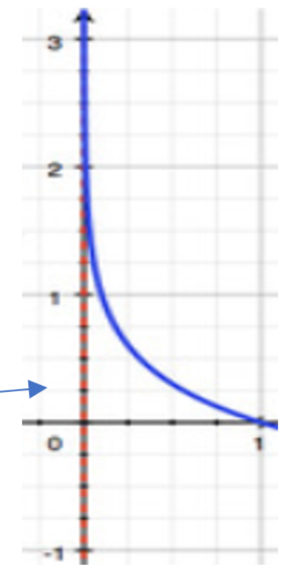
We need to define logistic regression cost function to compute w and b parameters:



For the given training data set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$ we want to get $\forall_i\ \hat{y}^{(i)} \approx y^{(i)}$

where $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$ and $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}} \in (0,1)$ (i) is the notation for i-th example

On this basis, we can define a loss function, called also an error function, for a single example that measures how good the output $\hat{y}$ is when the desired (trained) label is y:



The absolute error function $L_1(\hat{y}, y) = |\hat{y} - y|$ or the squared error function: $L_2(\hat{y}, y) = (\hat{y} - y)^2$ might seem like a good choice for this measure, but today we do not usually do this in this way because the optimization problem for it becomes not convex, so the gradient descent algorithm cannot find the global optimum of such loss functions easily!

We need to define the loss function in such a way that the function will be convex, so we use:

$$L_3(\hat{y}, y) = -(y\,log\,\hat{y} + (1-y)log(1-\hat{y}))$$

Consider two bounding cases:

If $y = 0$ then $L(\hat{y}, y) = -\,log\,(1-\hat{y})$, so to minimize it, $log\,(1-\hat{y})$ must be large and $\hat{y}$ small $(\hat{y} \to 0)$.

If $y = 1$ then $L(\hat{y}, y) = -\,log\,\hat{y}$, so to minimize it, $log\,\hat{y}$ and $\hat{y}$ must be large $(\hat{y} \to 1)$.

Finally, we define a cost function that measures the error on the entire training data set (for all examples):

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\,log\,\hat{y}^{(i)} + (1-y^{(i)})log(1-\hat{y}^{(i)})\right)$$

Loss Functions

The loss functions are used to evaluate the performance of the models. The bigger your loss is, the more different your predictions ($\hat{y}$) are from the true values (y). In deep learning, we use optimization algorithms like Gradient Descent to train models and minimize the cost.

L1 loss is defined as an absolute distance between vectors $\hat{y}$ and y of the size n:

$$L_1(\hat{y}, y) = \sum_{j=0}^{n} |y_j - \hat{y}_j| \tag{1}$$

L2 loss is defined as a square distance between vectors $\hat{y}$ and y of the size n:

$$L_2(\hat{y}, y) = \sum_{j=0}^{n} (y_j - \hat{y}_j)^2 \tag{2}$$

L2 loss is defined between vectors $\hat{y}$ and y of the size n in the following way:

$$L_3(\hat{y}, y) = -\sum_{j=0}^{n} (y_j log(\hat{y}_j) + (1 - y_j)(1 - log(\hat{y}_j))) \tag{3}$$

```python
def L1(yhat, y):
    loss1 = np.sum(np.abs(y-yhat))
    return loss1

def L2(yhat, y):
    loss2 = np.sum(np.dot(y-yhat,y-yhat))
    return loss2

def L3(yhat, y):
    loss3 = - np.sum(y * np.log(yhat) + (1-y) * np.log(1-yhat))
    return loss3
```

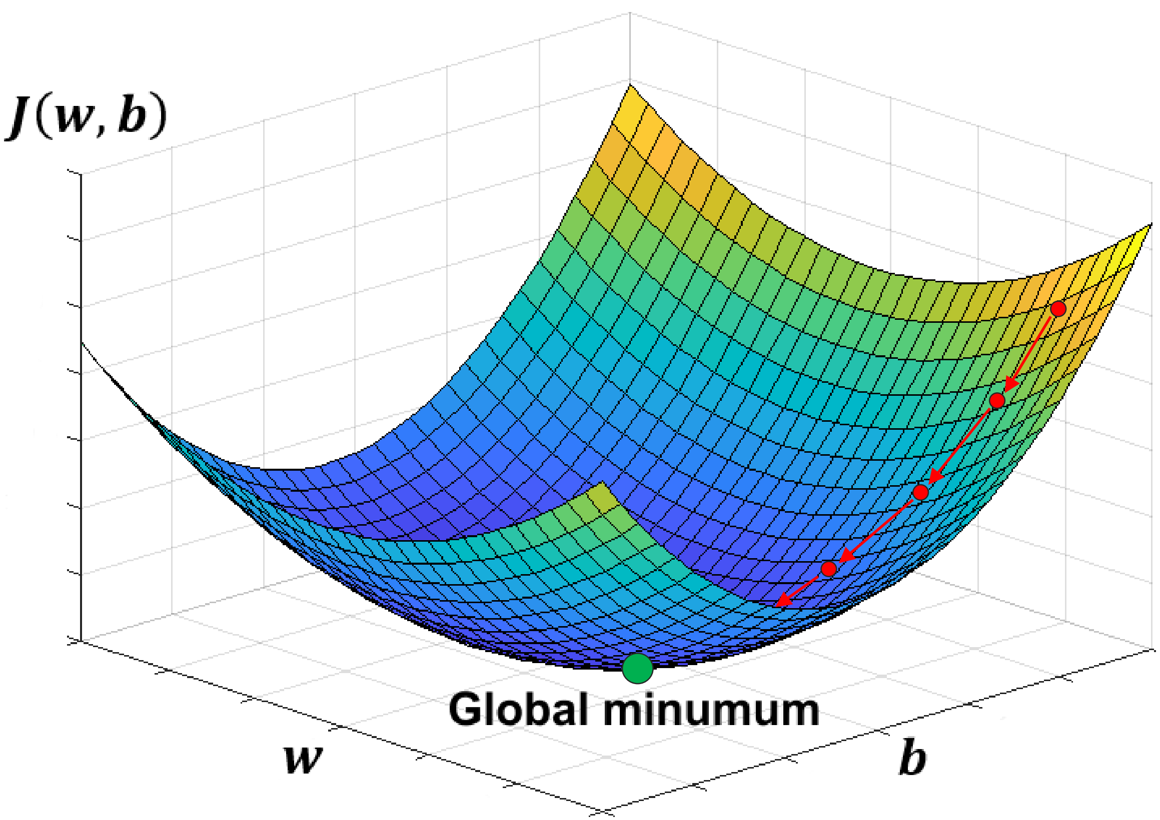
```python
yhat = np.array([.78, .89, .12, .08, .97])
y = np.array([1, 1, 0, 0, 1])
print("Loss1 = " + str(L1(yhat,y)))
print("Loos2 = " + str(L2(yhat,y)))
print("Loos3 = " + str(L3(yhat,y)))
```

```
Loss1 = 0.5599999999999999
Loos2 = 0.0822
Loos3 = 0.6066693634880955
```

Gradient Descent

We have to minimize the cost function *J* for a given training data set to achieve as correct prediction for input data as possible:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\, log\, \hat{y}^{(i)} + \left(1-y^{(i)}\right) log\left(1-\hat{y}^{(i)}\right)\right)$$



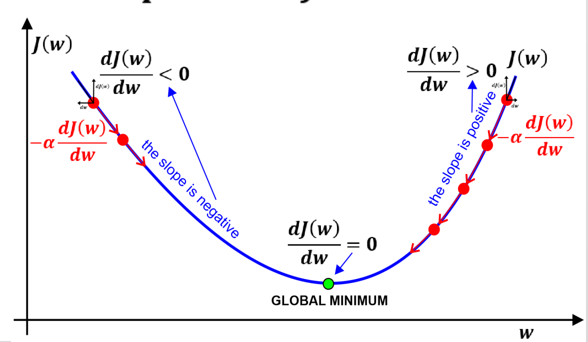
Here, *w* is 1D, but its dimension is bigger in real.

To minimize the cost function we calculate partial derivatives where $\frac{dJ(w,b)}{dw}$ and $\frac{dJ(w,b)}{db}$ of J with respect to parameters w and b and repeatedly use them to update them with a step α – called a learning rate:

$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

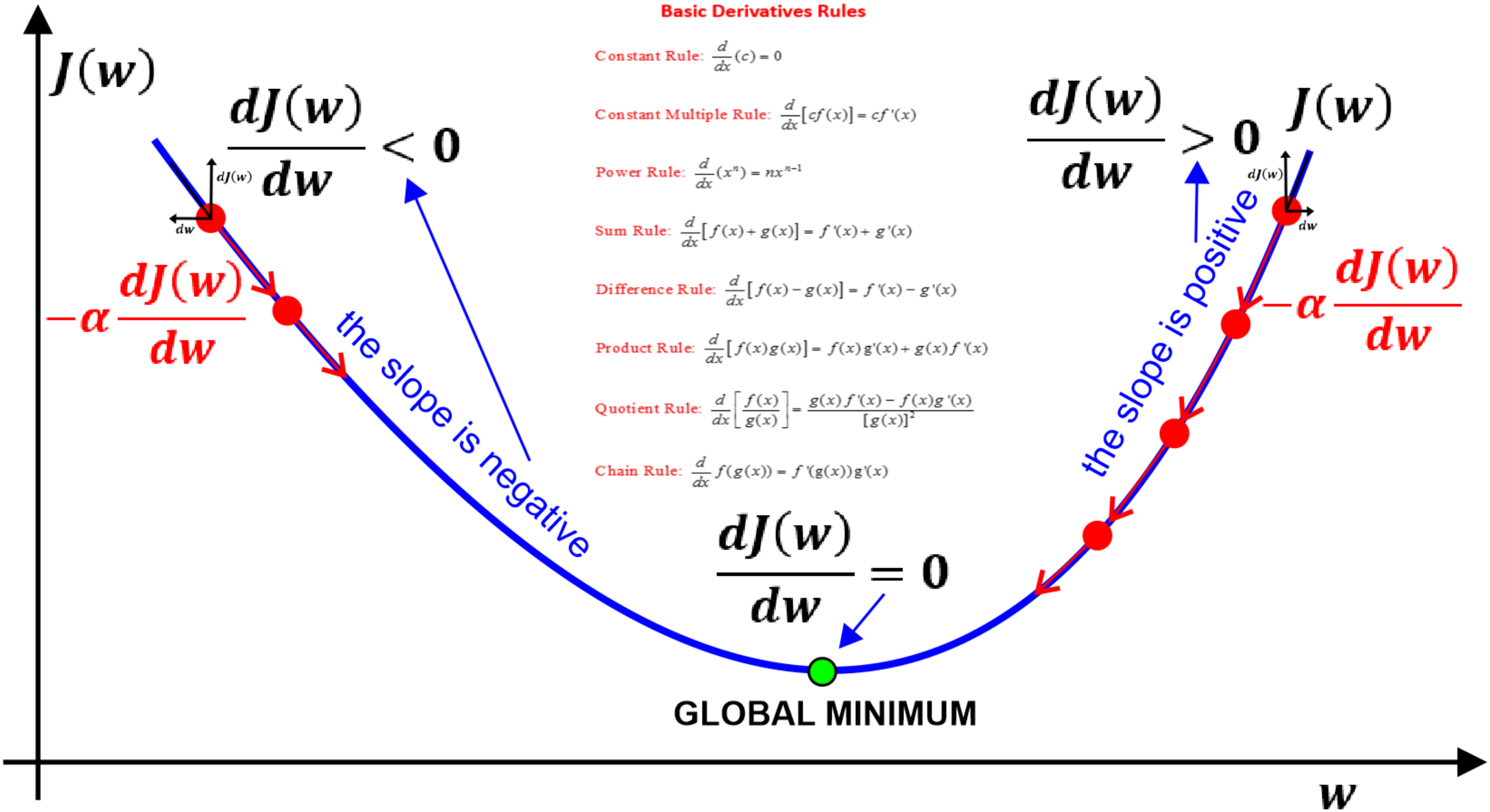
$$b := b - \alpha \frac{dJ(w,b)}{db}$$

Partial derivatives $\frac{dJ(w,b)}{dw} = \frac{\partial J(w,b)}{\partial w}$ and $\frac{dJ(w,b)}{db} = \frac{\partial J(w,b)}{\partial b}$ represent the slopes of the J function:



Calculus of the Gradient Descent

The main idea of the Gradient Descent algorithm is to go in the reverse direction to the gradient (the descent slope):



Derivative Rules

The Gradient Descent algorithm uses partial derivatives calculated after the following rules:

Basic Derivatives Rules

Constant Rule: $\frac{d}{dx}(c) = 0$

Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$

Power Rule: $\frac{d}{dx}(x^n) = nx^{n-1}$

Sum Rule: $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$

Difference Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$

Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$

Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$

Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

Derivative Rules

Exponential Functions

$$\frac{d}{dx}(e^x) = e^x$$
$$\frac{d}{dx}(a^x) = a^x \ln a$$
$$\frac{d}{dx}(e^{g(x)}) = e^{g(x)} g'(x)$$
$$\frac{d}{dx}(a^{g(x)}) = \ln(a)\, a^{g(x)}\, g'(x)$$

Logarithmic Functions

$$\frac{d}{dx}(\ln x) = \frac{1}{x},\ x > 0$$
$$\frac{d}{dx}\ln(g(x)) = \frac{g'(x)}{g(x)}$$
$$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a},\ x > 0$$
$$\frac{d}{dx}(\log_a g(x)) = \frac{g'(x)}{g(x)\ln a}$$

Trigonometric Functions

$$\frac{d}{dx}(\sin x) = \cos x$$
$$\frac{d}{dx}(\cos x) = -\sin x$$
$$\frac{d}{dx}(\tan x) = \sec^2 x$$
$$\frac{d}{dx}(\csc x) = -\csc x \cot x$$
$$\frac{d}{dx}(\sec x) = \sec x \tan x$$
$$\frac{d}{dx}(\cot x) = -\csc^2 x$$

Inverse Trigonometric Functions

$$\frac{d}{dx}(\sin^{-1} x) = \frac{1}{\sqrt{1 - x^2}},\ x \neq \pm 1$$
$$\frac{d}{dx}(\cos^{-1} x) = \frac{-1}{\sqrt{1 - x^2}},\ x \neq \pm 1$$
$$\frac{d}{dx}(\tan^{-1} x) = \frac{1}{1 + x^2}$$
$$\frac{d}{dx}(\cot^{-1} x) = \frac{-1}{1 + x^2}$$
$$\frac{d}{dx}(\sec^{-1} x) = \frac{1}{x\sqrt{x^2 - 1}},\ x \neq \pm 1, 0$$
$$\frac{d}{dx}(\csc^{-1} x) = \frac{-1}{x\sqrt{x^2 - 1}},\ x \neq \pm 1, 0$$

Hyperbolic Functions

$$\frac{d}{dx}(\sinh x) = \cosh x$$
$$\frac{d}{dx}(\cosh x) = \sinh x$$
$$\frac{d}{dx}(\tanh x) = \operatorname{sech}^2 x$$
$$\frac{d}{dx}(\operatorname{csch} x) = -\operatorname{csch} x \coth x$$
$$\frac{d}{dx}(\operatorname{sech} x) = -\operatorname{sech} x \tanh x$$
$$\frac{d}{dx}(\coth x) = -\operatorname{csch} x$$

Inverse Hyperbolic Functions

$$\frac{d}{dx}(\sinh^{-1} x) = \frac{1}{\sqrt{1 + x^2}}$$
$$\frac{d}{dx}(\cosh^{-1} x) = \frac{1}{\sqrt{x^2 - 1}},\ x > 1$$
$$\frac{d}{dx}(\tanh^{-1} x) = \frac{1}{1 - x^2},\ |x| < 1$$
$$\frac{d}{dx}(\operatorname{csch}^{-1} x) = \frac{-1}{|x|\sqrt{1 - x^2}},\ x \neq 0$$
$$\frac{d}{dx}(\operatorname{sech}^{-1} x) = \frac{-1}{x\sqrt{1 - x^2}},\ 0 < x < 1$$
$$\frac{d}{dx}(\coth^{-1} x) = \frac{1}{1 - x^2},\ |x| > 1$$

Gradient Descent for Logistic Regression

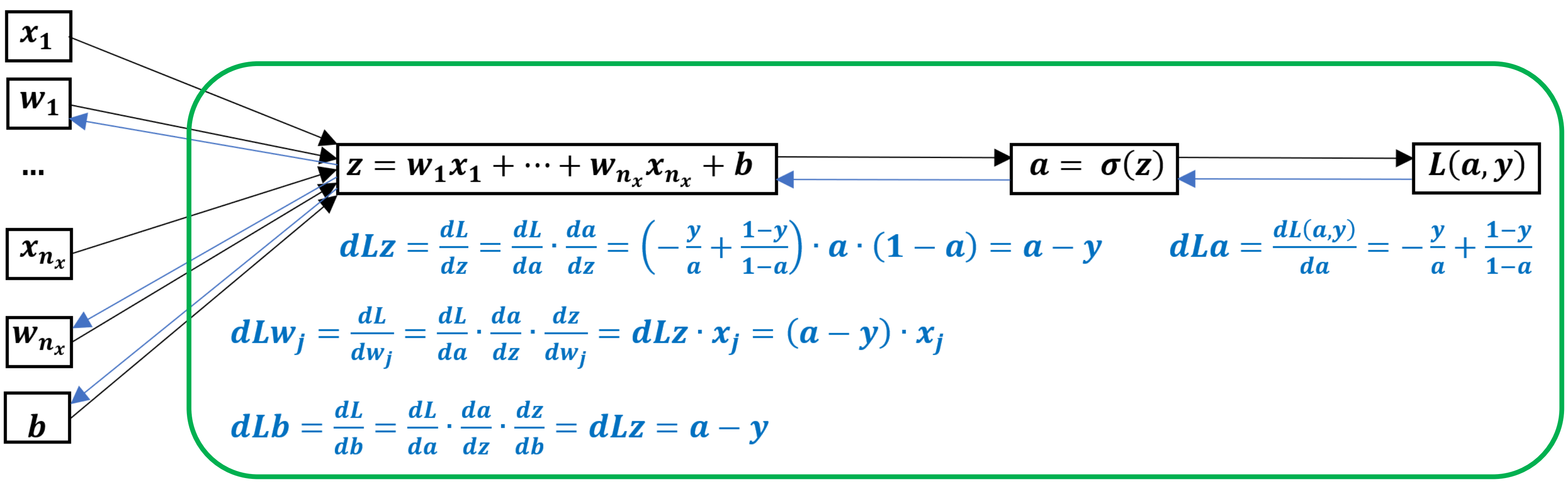
We use a computational graph for the presentation of forward and backward operations for a single neuron implementing logistic regression for the weighted sum of inputs *x*:

Use a computational graph to present operations of computation of the logistic regression and its derivatives:

$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$L(a, y) = -\big(y \log a + (1 - y) \log(1 - a)\big)$$



$$dLz = \frac{dL}{dz} = \frac{dL}{da} \cdot \frac{da}{dz} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \cdot a \cdot (1 - a) = a - y \qquad dLa = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$dLw_j = \frac{dL}{dw_j} = \frac{dL}{da} \cdot \frac{da}{dz} \cdot \frac{dz}{dw_j} = dLz \cdot x_j = (a - y) \cdot x_j$$

$$dLb = \frac{dL}{db} = \frac{dL}{da} \cdot \frac{da}{dz} \cdot \frac{dz}{db} = dLz = a - y$$

Finally, we get the update-rules for the logistic regression using the gradient descent algorithm:

$$w_j := w_j - \alpha \cdot dLw_j = w_j - \alpha \cdot (a - y) \cdot x_j$$

$$b := b - \alpha \cdot dLb = b - \alpha \cdot (a - y)$$

Gradient Descent for Training Dataset

For training dataset consisting of m training examples, we minimize the cost function J:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{(i)}, y^{(i)}\right)$$

$$\hat{y}^{(i)} = a^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(w^T x^{(i)} + b\right)$$

$$\frac{dJ(w,b)}{dw_j} = \frac{1}{m}\sum_{i=1}^{m} \frac{dL\left(a^{(i)}, y^{(i)}\right)}{dw_j} = \frac{1}{m}\sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right) \cdot x_j^{(i)}$$

$$\frac{dJ(w,b)}{db} = \frac{1}{m}\sum_{i=1}^{m} \frac{dL\left(a^{(i)}, y^{(i)}\right)}{db} = \frac{1}{m}\sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right)$$

The final logistic regression gradient descent algorithm will repeatedly go through all training examples updating parameters until the cost function is not small enough:

To speed up computation we should use vectorization instead of for-loops:

repeat

$J = 1$

for $j = 1$ to n_x

$dJw_j = 0$

$dLb = 0$

for $i = 1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma\left(z^{(i)}\right)$

$J += -\left(y^{(i)} \log a^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right)$

$dJz^{(i)} = a^{(i)} - y^{(i)}$

for $j = 1$ to n_x

$dJw_j += x_j^{(i)} \cdot dJz^{(i)}$

$dJb += dJz^{(i)}$

$J /= m$

for $j = 1$ to n_x

$dJw_j /= m$

$w_j -= \alpha \cdot dJw_j$

$dJb /= m$

$b -= \alpha \cdot dJb$

until $J < \varepsilon$

Efficiency of Vectorization

When dealing with big data collections and big data vectors, we definitely should use vectorization (that performs SIMD operations) to proceed computations faster:

```
import numpy as np
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
dot_vec = np.dot(a,b)
toc = time.time()
print ("dot_vec = " + str(dot_vec))

print("Vectorized dot product computation time: " + str(1000 * (toc-tic)) + "ms")

dot_for = 0
tic = time.time()
for i in range(1000000):
    dot_for += a[i]*b[i]
toc = time.time()
print ("dot_for = " + str(dot_for))

print("For-looped dot product computation time: " + str(1000 * (toc-tic)) + "ms")
```

```
dot_vec = 250265.14164263124
Vectorized dot product computation time: 0.9922981262207031ms
dot_for = 250265.1416426372
For-looped dot product computation time: 352.65374183654785ms
```

Compare time efficacies of these two approaches!

Conclusion:

Whenever possible, avoid explicit for-loops and use vectorization: np.dot(w.T,x), np.dot(W,x), np.multiply(x1,x2), np.outer(x1,x2), np.log(v), np.exp(v), np.abs(v), np.zeros(v), np.sum(v), np.max(v), np.min(v) etc.
Vectorization uses parallel CPU or GPU operations (called SIMD – single instruction multiple data) proceed on parallelly working cores.

Vectorization of the Logistic Regression

repeat

$J = 1$

for $j = 1$ to n_x

$dJw_j = 0$

$dLb = 0$

for $i = 1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J += -\left(y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right)$

$dJz^{(i)} = a^{(i)} - y^{(i)}$

for $j = 1$ to n_x

$dJw_j += x_j^{(i)} \cdot dJz^{(i)}$

$dJb += dJz^{(i)}$

$J /= m$

for $j = 1$ to n_x

$dJw_j /= m$

$w_j -= \alpha \cdot dJw_j$

$dJb /= m$

$b -= \alpha \cdot dJb$

until $J < \varepsilon$

Let's vectorize the previous algorithm:

$dJw = np.zeros((n_x, 1))$

broadcasted

$Z = w^T X + b = np.dot(w.T, X) + b$

We use matrices

$X = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$

$Z = [z^{(1)}, z^{(2)}, \dots, z^{(m)}]$

$A = [a^{(1)}, a^{(2)}, \dots, a^{(m)}]$

$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$

$A = \sigma(Z)$

$dJZ = A - Y$

$dJw += x^{(i)} \cdot dJz^{(i)}$

$dJw = \frac{1}{m} X \cdot dJZ^T$

$dJb = \frac{1}{m} \cdot np.sum(dJZ)$

$dJw /= m$

$w -= \alpha \cdot dJw$

$b -= \alpha \cdot dJb$

Broadcasting in Python

BROADCASTING PRINCIPLE:

(m, n)	+	(1, n) → (m, n)	=	(m, n)
(m, n)	-	(1, n) → (m, n)	=	(m, n)
(m, n)	*	(1, n) → (m, n)	=	(m, n)
(m, n)	/	(1, n) → (m, n)	=	(m, n)
(m, n)	+	(m, 1) → (m, n)	=	(m, n)
(m, n)	-	(m, 1) → (m, n)	=	(m, n)
(m, n)	*	(m, 1) → (m, n)	=	(m, n)
(m, n)	/	(m, 1) → (m, n)	=	(m, n)

BROADCASTING SAMPLES:

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + 10 = \begin{bmatrix}11\\12\\13\end{bmatrix}$$

where 10 was broadcasted (1,1) → (4,1)

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + \begin{bmatrix}10\\10\\10\end{bmatrix} = \begin{bmatrix}11\\12\\13\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}10 & 20 & 30\end{bmatrix} = \begin{bmatrix}11 & 22 & 33\\14 & 25 & 36\end{bmatrix}$$

where $\begin{bmatrix}10 & 20 & 30\end{bmatrix}$ was broadcasted (1,3) → (2,3)

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}10 & 20 & 30\\10 & 20 & 30\end{bmatrix} = \begin{bmatrix}11 & 22 & 33\\14 & 25 & 36\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}10\\20\end{bmatrix} = \begin{bmatrix}11 & 12 & 13\\24 & 25 & 26\end{bmatrix}$$

where $\begin{bmatrix}10\\20\end{bmatrix}$ was broadcasted (2,1) → (2,3)

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}10 & 10 & 10\\20 & 20 & 20\end{bmatrix} = \begin{bmatrix}11 & 12 & 13\\24 & 25 & 26\end{bmatrix}$$

Broadcasting in numpy

Broadcasting is very useful for performing mathematical operations between arrays of different shapes. The example below show the normalization of the data.

A softmax function is a normalizing function often used in the output layers of neural networks when you need to classify two or more classes:

- for $x \in \mathbb{R}^{1\times n}$, $softmax(x) = softmax(\begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}) = \begin{bmatrix} \frac{e^{x_1}}{\sum_j e^{x_j}} & \frac{e^{x_2}}{\sum_j e^{x_j}} & \dots & \frac{e^{x_n}}{\sum_j e^{x_j}} \end{bmatrix}$
- for a matrix $x \in \mathbb{R}^{m\times n}$, x_{ij} maps to the element in the i^{th} row and j^{th} column of x, thus we have:

$$softmax(x) = softmax\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} \frac{e^{x_{11}}}{\sum_j e^{x_{1j}}} & \frac{e^{x_{12}}}{\sum_j e^{x_{1j}}} & \frac{e^{x_{13}}}{\sum_j e^{x_{1j}}} & \dots & \frac{e^{x_{1n}}}{\sum_j e^{x_{1j}}} \\ \frac{e^{x_{21}}}{\sum_j e^{x_{2j}}} & \frac{e^{x_{22}}}{\sum_j e^{x_{2j}}} & \frac{e^{x_{23}}}{\sum_j e^{x_{2j}}} & \dots & \frac{e^{x_{2n}}}{\sum_j e^{x_{2j}}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{e^{x_{m1}}}{\sum_j e^{x_{mj}}} & \frac{e^{x_{m2}}}{\sum_j e^{x_{mj}}} & \frac{e^{x_{m3}}}{\sum_j e^{x_{mj}}} & \dots & \frac{e^{x_{mn}}}{\sum_j e^{x_{mj}}} \end{bmatrix} = \begin{pmatrix} softmax\text{(first row of x)} \\ softmax\text{(second row of x)} \\ \dots \\ softmax\text{(last row of x)} \end{pmatrix}$$

In [27]:
```
def softmax(x):
    # This function calculates the softmax for each row of the input x, where x is a row vector or a matrix of shape (n, m).
    x_exp = np.exp(x)
    x_sum = np.sum(x_exp,axis=1,keepdims=True)
    s = x_exp/x_sum  # It automatically uses numpy broadcasting.
    return s
```

In [29]:
```
x = np.array([
    [0, 9, 3, 0],
    [3, 0, 8, 1]])
print("softmax(x) = " + str(softmax(x)))
```

```
softmax(x) = [[1.23074356e-04 9.97281837e-01 2.47201452e-03 1.23074356e-04]
 [6.68456877e-03 3.32805082e-04 9.92077968e-01 9.04658008e-04]]
```

Normalization for Efficiency

We use normalization (np.linalg.norm) to achieve a better performance because gradient descent converges faster after normalization:

Normalization is changing x to $\frac{x}{\|x\|}$ (dividing each row vector of x by its norm), e.g.

If

$$x = \begin{bmatrix} 3 & 2 & 4 \\ 1 & 8 & 2 \end{bmatrix} \tag{3}$$

then

$$\|x\| = np.linalg.norm(x, axis = 1, keepdims = True) = \begin{bmatrix} \sqrt{29} \\ \sqrt{69} \end{bmatrix} \tag{4}$$

and

$$x\_normalized = \frac{x}{\|x\|} = \begin{bmatrix} \frac{3}{\sqrt{29}} & \frac{2}{\sqrt{29}} & \frac{4}{\sqrt{29}} \\ \frac{1}{\sqrt{69}} & \frac{8}{\sqrt{69}} & \frac{2}{\sqrt{69}} \end{bmatrix} \tag{5}$$

In [25]:
```
def normalizeRows(x):
    # This function normalizes each row of the matrix x, where x is a numpy matrix of shape (n, m)
    x_norm = np.linalg.norm(x,ord=2,axis=1,keepdims=True)
    print("x_norm = " + str(x_norm))
    x = x/x_norm
    return x
```

In [26]:
```
x = np.array([
    [3, 2, 4],
    [1, 8, 2]])
print("normalizeRows(x) = " + str(normalizeRows(x)))
```

```
x_norm = [[5.38516481]
 [8.30662386]]
normalizeRows(x) = [[0.55708601 0.37139068 0.74278135]
 [0.12038585 0.96308682 0.24077171]]
```

Lists vs. Vectors and Matrices

```python
import numpy as np

print("List of values:")
a = np.random.randn(6)    # generates list of samples from the normal distribution, while rand from unifrom (in range [0,1))
print(a)
print(a.shape)            # the shape suggest that a is a list
print(a.T)                # the list cannot be transposed because it is not a vector or matrix!
print(np.dot(a,a.T))      # what should it mean?!

print("Vector of values:")
b = np.random.randn(6,1) # generates matrix of samples from the normal distribution
print(b)
print(b.shape)            # the shape suggest that b is a matrix (vector)
print(b.T)                # the vector can be transposed
print(np.dot(b,b.T))      # now we get a matrix as a result of multiplication of the vectors
```

Be careful when creating vectors because lists have no shape and are declared similarly.

```
List of values:
[ 1.63130571  1.30039595 -1.42170758  1.28012586  1.63085575  0.64436582]
(6,)
[ 1.63130571  1.30039595 -1.42170758  1.28012586  1.63085575  0.64436582]
11.08706038339276
Vector of values:
[[-1.2426375 ]
 [-0.54254535]
 [ 0.76000053]
 [-0.83861851]
 [ 0.66463   ]
 [-1.60972555]]
(6, 1)
[[-1.2426375  -0.54254535  0.76000053 -0.83861851  0.66463    -1.60972555]]
[[ 1.54414796  0.6741872  -0.94440516  1.04209881 -0.82589416  2.00030533]
 [ 0.6741872   0.29435546 -0.41233475  0.45498857 -0.36059191  0.87334911]
 [-0.94440516 -0.41233475  0.57760081 -0.63735051  0.50511915 -1.22339227]
 [ 1.04209881  0.45498857 -0.63735051  0.703281   -0.55737102  1.34994564]
 [-0.82589416 -0.36059191  0.50511915 -0.55737102  0.44173303 -1.06987188]
 [ 2.00030533  0.87334911 -1.22339227  1.34994564 -1.06987188  2.59121633]]
```

Column and Row Vectors

```
import numpy as np

C=np.random.randn(5,1)
D=np.random.randn(1,5)
print("We define matrices and vectors using (m, n) where m is a number of rows, and n is a number of columns")
print(C)
print("... is a column vector")
print(D)
print("... is a row vector")
```

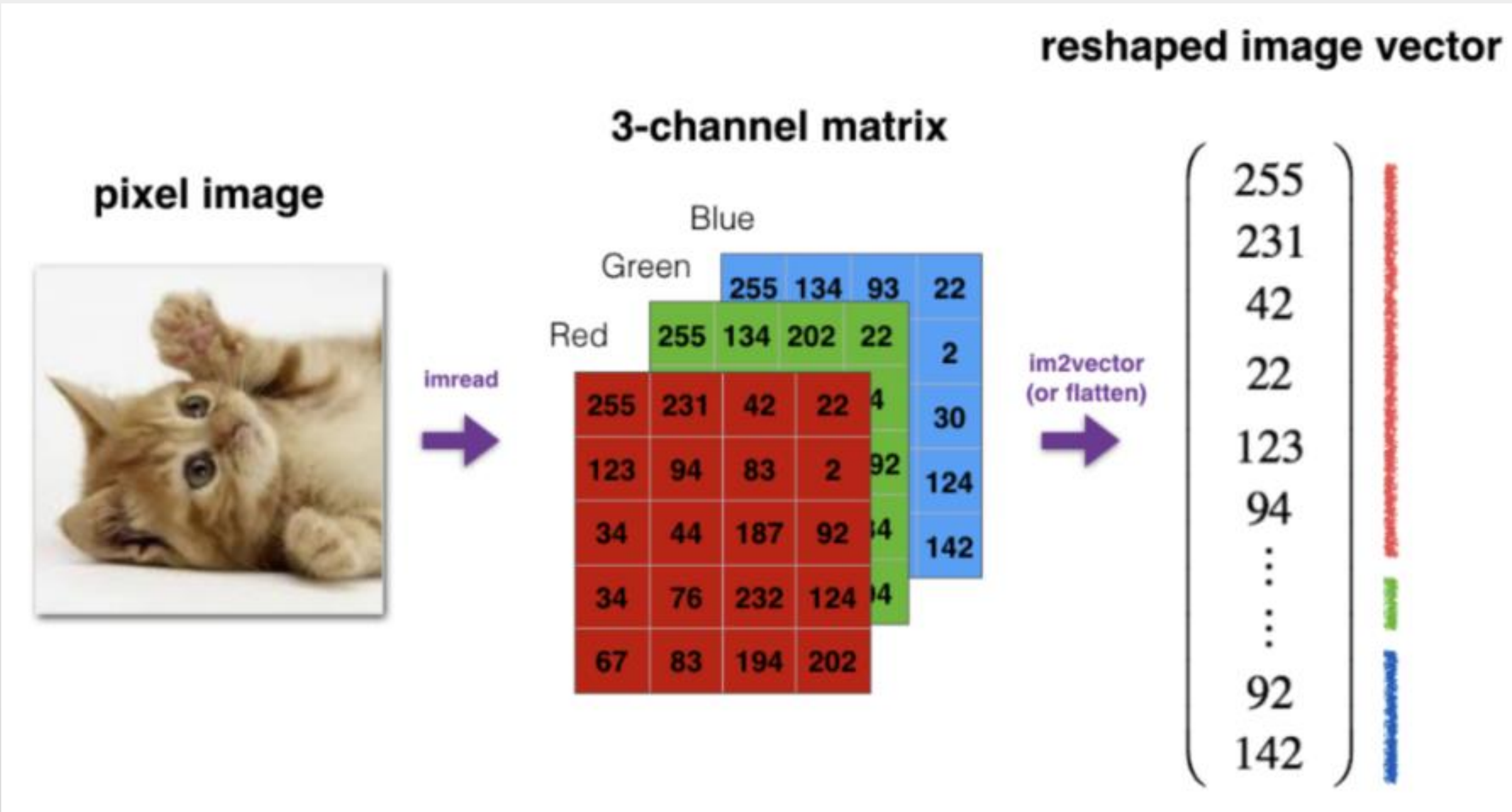
```
We define matrices and vectors using (m, n) where m is a number of rows, and n is a number of columns
[[ 0.23665149]
 [ 0.45132428]
 [-0.89728231]
 [ 0.72912635]
 [-0.92627707]]
... is a column vector
[[ 0.99318971 -0.8439588   1.20413677 -1.00233032 -1.55317979]]
... is a row vector
```

```
import numpy as np

a = np.random.randn(5)     # the list can be reshaped to create a vector
print(a)
print(a.shape)
a = a.reshape((5,1))
print(a)
print(a.shape)

assert(a.shape == (5, 1)) # we can check whether the shape is correct
```

```
[-0.07161977 -2.17009596  0.09644837  0.5044574  -0.04263376]
(5,)
[[-0.07161977]
 [-2.17009596]
 [ 0.09644837]
 [ 0.5044574 ]
 [-0.04263376]]
(5, 1)
```

Reshaping Image Matrices

When working with images in deep learning, we typically reshape them into vector representation using [np.reshape()]():



Shape and Reshape Vectors and Matrices

We commonly use the numpy functions np.shape() and np.reshape() in deep learning:

- **X.shape is used to get the shape (dimension) of a vector or a matrix X.**
- **X.reshape(...) is used to reshape a vector or a matrix X into some other dimension(s).**

Images are usually represented by 3D arrays of shape $(length, height, depth = 3)$. Nevertheless, when you read an image as the input of an algorithm you typically convert it to a vector of shape $(length * height * 3, 1)$, so you "unroll" (reshape) the 3D arrays into 1D vectors for further processing:

Example 1: If you would like to reshape an array v of shape (a, b, c) into a vector of shape (a*b,c) you would do:

```
v = v.reshape((v.shape[0] * v.shape[1], v.shape[2])) # where v.shape[0] = a ; v.shape[1] = b ; v.shape[2] = c
```

Example 2: If you would like to reshape an array v of shape (a, b, c) into a vector of shape (abc) you would do:

```
v = v.reshape((v.shape[0] * v.shape[1] * v.shape[2], 1)) # where v.shape[0] = a ; v.shape[1] = b ; v.shape[2] = c
```

- Never hard-code the dimensions of the image as a constant but use the quantities you need with `image.shape[0]`, etc.

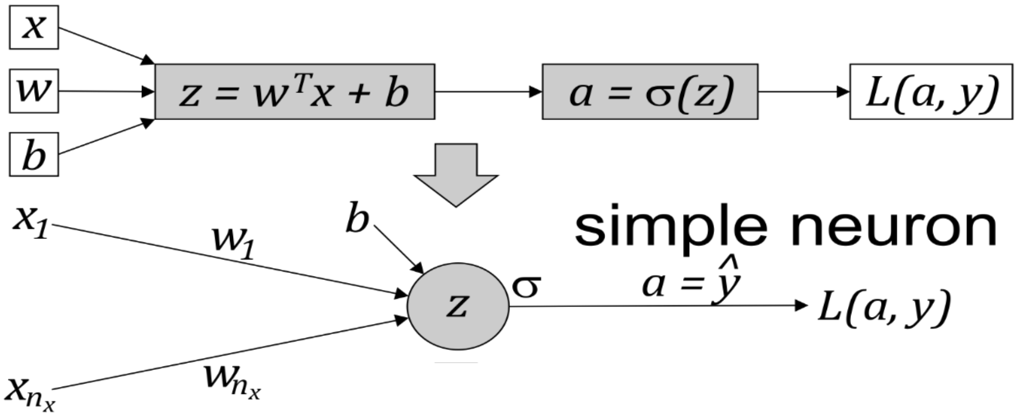
In [30]:
```
def image2vector(image):
    # This function reshapes a numpy array of shape (length, height, depth) to a vector of shape (length*height*depth, 1)
    v = image.reshape((image.shape[0]*image.shape[1]*image.shape[2]),1)
    return v
```

In [33]:
```
# Images usually are (num_px_x, num_px_y, 3) where 3 represents the RGB values: red, green, and blue
# This is an exemplary 3 by 3 by 3 array:
image = np.array([[[ 0.139,  0.381],
        [ 0.982,  0.647],
        [ 0.251 ,  0.551]],
       [[ 0.219,  0.647],
        [ 0.703,  0.845],
        [ 0.397,  0.313]],
       [[ 0.855,  0.165],
        [ 0.313,  0.937],
        [ 0.279,  0.077]]])

print ("image = " + str(image))
print ("image2vector(image) = " + str(image2vector(image)))
```

```
image = [[[0.139 0.381]
  [0.982 0.647]
  [0.251 0.551]]

 [[0.219 0.647]
  [0.703 0.845]
  [0.397 0.313]]

 [[0.855 0.165]
  [0.313 0.937]
  [0.279 0.077]]]
```

```
image2vector(image) = [[0.139]
 [0.381]
 [0.982]
 [0.647]
 [0.251]
 [0.551]
 [0.219]
 [0.647]
 [0.703]
 [0.845]
 [0.397]
 [0.313]
 [0.855]
 [0.165]
 [0.313]
 [0.937]
 [0.279]
 [0.077]]
```

Simple Neuron

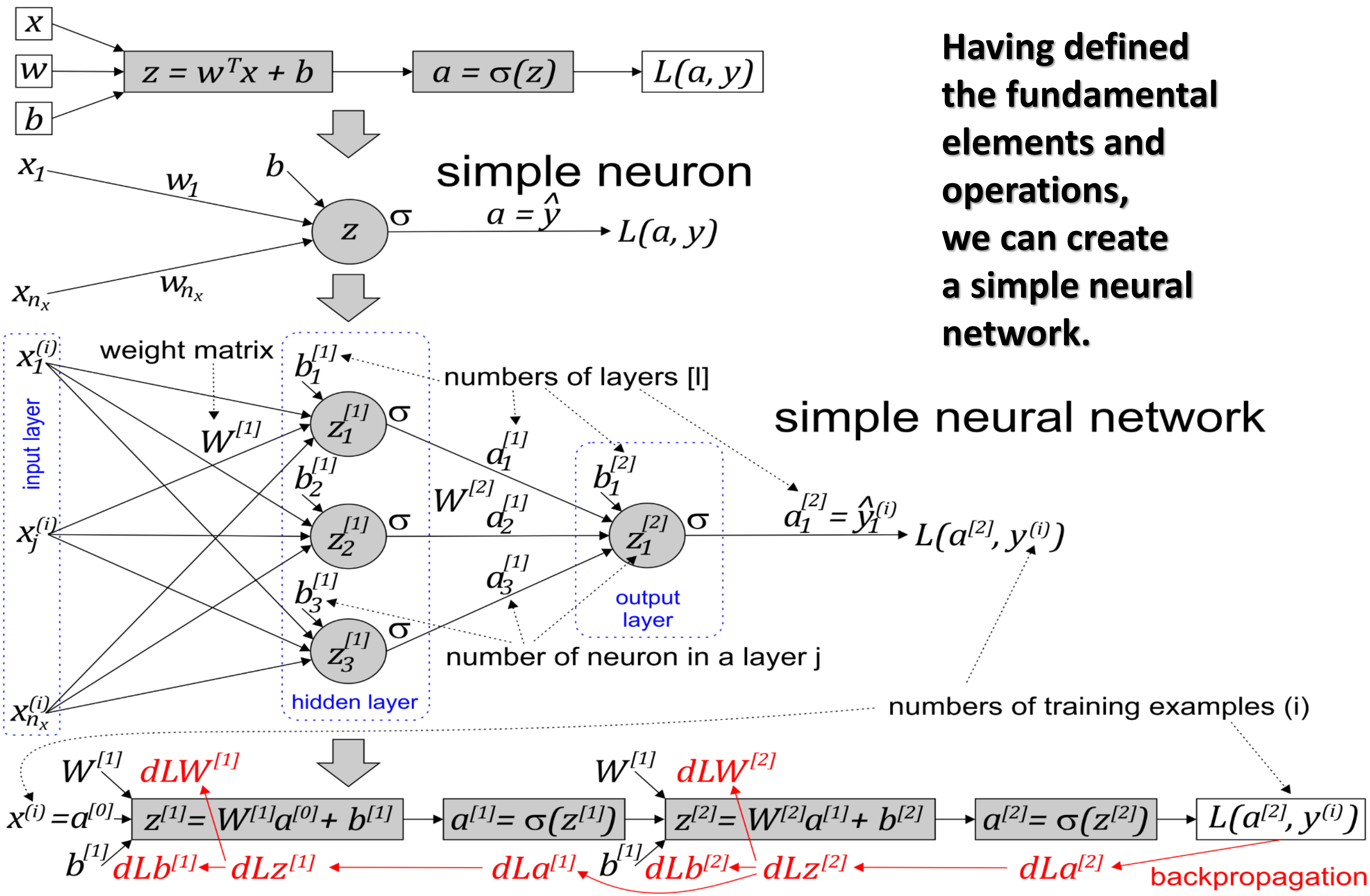


simple neuron

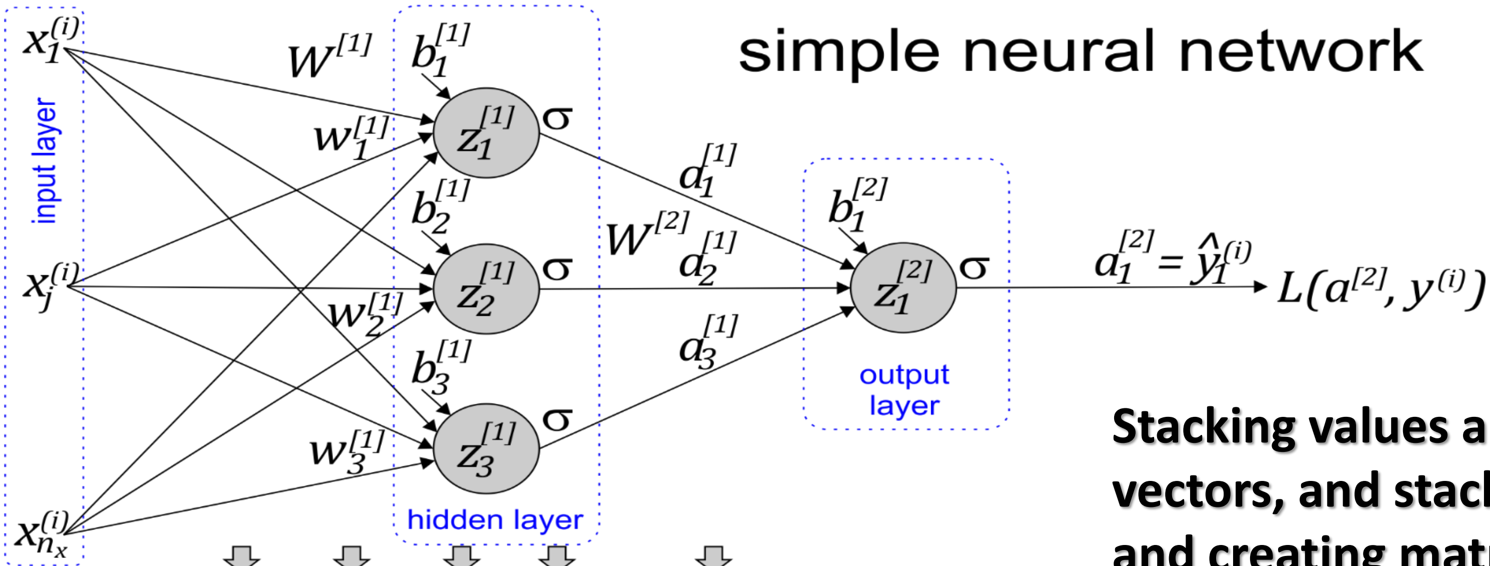
We defined the fundamental elements and operations on a single neuron.

Simple Neural Network

Having defined the fundamental elements and operations, we can create a simple neural network.



Stacking Neurons Vertically and Vectorizing



$$z_1^{[1]} = w_1^{[1]T} a^{[0]} + b_1^{[1]} \longrightarrow a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} a^{[0]} + b_2^{[1]} \longrightarrow a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} a^{[0]} + b_3^{[1]} \longrightarrow a_3^{[1]} = \sigma(z_3^{[1]})$$

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \end{bmatrix} \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ a_3^{[0]} \end{bmatrix} \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} \qquad \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix}$$

$$z^{[1]} \quad W^{[1]} \quad a^{[0]} \quad b^{[1]} \qquad a^{[1]}$$

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]} \longrightarrow a^{[1]} = \sigma(z^{[1]})$$

Stacking values and creating vectors, and stacking vectors and creating matrices is very important from the efficiency of computation point of view!

numbers of layers [l]

numbers of training examples (i)

number of neuron in a layer j

Stacking Examples Horizontally and Vectorizing

No of neurons of (l-1)-th layer

$$W^{[l]} = \begin{bmatrix} — w_1^{[l]T} — \\ — w_2^{[l]T} — \\ \vdots \\ — w_{n^{[l]}}^{[l]T} — \end{bmatrix}$$

No of neurons of l-th layer

$$z_1^{[1](i)} = w_1^{[1]T} a^{[0](i)} + b_1^{[1]} \rightarrow a_1^{[1](i)} = \sigma(z_1^{[1](i)})$$

$$z_2^{[1](i)} = w_2^{[1]T} a^{[0](i)} + b_2^{[1]} \rightarrow a_2^{[1](i)} = \sigma(z_2^{[1](i)})$$

$$z_3^{[1](i)} = w_3^{[1]T} a^{[0](i)} + b_3^{[1]} \rightarrow a_3^{[1](i)} = \sigma(z_3^{[1](i)})$$

Stacking vectors of training examples horizontally creating matrices is very important from the efficiency of computation point of view!

$$\begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \end{bmatrix} = \begin{bmatrix} a_1^{[0](i)} \\ a_2^{[0](i)} \\ a_3^{[0](i)} \end{bmatrix}$$

input examples

$$\begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \\ z_3^{[1](i)} \end{bmatrix} \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \end{bmatrix} \begin{bmatrix} a_1^{[0](i)} \\ a_2^{[0](i)} \\ a_3^{[0](i)} \end{bmatrix} \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} \begin{bmatrix} a_1^{[1](i)} \\ a_2^{[1](i)} \\ a_3^{[1](i)} \end{bmatrix}$$

numbers of layers [l]

numbers of training examples (i)

number of neuron in a layer j

$$x^{(i)} = a^{[0](i)} \qquad z^{[1](i)} \quad W^{[1]} \quad a^{[0](i)} \quad b^{[1]} \quad a^{[1](i)}$$

$$x^{(1)}, \ldots, x^{(i)}, \ldots, x^{(m)}$$

training examples

$$\begin{bmatrix} | & & | & & | \\ x^{(1)} & \ldots & x^{(i)} & \ldots & x^{(m)} \\ | & & | & & | \end{bmatrix}$$

input data dimension

$$z^{[1](1)} = W^{[1]} a^{[0](1)} + b^{[1]} \rightarrow a^{[1](1)} = \sigma(z^{[1](1)})$$

$$\vdots$$

$$z^{[1](i)} = W^{[1]} a^{[0](i)} + b^{[1]} \rightarrow a^{[1](i)} = \sigma(z^{[1](i)})$$

$$\vdots$$

$$z^{[1](m)} = W^{[1]} a^{[0](m)} + b^{[1]} \rightarrow a^{[1](m)} = \sigma(z^{[1](m)})$$

training examples

$$\begin{bmatrix} | & & | & & | \\ a^{[l](1)} & \ldots & a^{[l](i)} & \ldots & a^{[l](m)} \\ | & & | & & | \end{bmatrix}$$

hidden neurons

training examples

$$\begin{bmatrix} | & & | & & | \\ z^{[l](1)} & \ldots & z^{[l](i)} & \ldots & z^{[l](m)} \\ | & & | & & | \end{bmatrix}$$

hidden neurons

$X = A^{[0]}$ **After Vectorizing** $Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]} \rightarrow A^{[1]} = \sigma(Z^{[1]})$

Vectorization of Dot Product

In deep learning, you deal with very large datasets. Non-computationally-optimal functions become a huge bottleneck in your algorithms and can result in models that take ages to run. To make sure that your code is computationally efficient, you should use vectorization. Compare the following codes:

```python
import time

x1 = [5, 1, 0, 3, 8, 2, 5, 6, 0, 1, 2, 5, 9, 0, 7]  # x1 = np.random.rand(1000000)
x2 = [2, 5, 2, 0, 3, 2, 2, 9, 1, 0, 2, 5, 4, 0, 9]  # x2 = np.random.rand(1000000)

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
dot = 0
for i in range(len(x1)):
    dot+= x1[i] * x2[i]
toc = time.process_time()
print ("for-looped dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("vectorized dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
for-looped dot = 235
 ----- Computation time = 0.0ms
vectorized dot = 235
 ----- Computation time = 0.0ms
```

Vectorization of Outer Product

In deep learning, you deal with very large datasets. Non-computationally-optimal functions become a huge bottleneck in your algorithms and can result in models that take ages to run. To make sure that your code is computationally efficient, you should use vectorization. Compare the following codes:

```python
import time

x1 = [5, 1, 0, 3, 8, 2, 5, 6, 0, 1, 2, 5, 9, 0, 7]  # x1 = np.random.rand(1000000)
x2 = [2, 5, 2, 0, 3, 2, 2, 9, 1, 0, 2, 5, 4, 0, 9]  # x2 = np.random.rand(1000000)

### CLASSIC OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.zeros((len(x1),len(x2))) # we create a len(x1)*len(x2) matrix with only zeros
for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i] * x2[j]
toc = time.process_time()
print ("for-looped outer = " + str(outer) + "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
print ("vectorized outer = " + str(outer) + "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
outer = [[81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [63. 14. 14. 63.  0. 63. 14. 35.  0.  0. 63. 14. 35.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
 ----- Computation time = 0.0ms
```

```
outer = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
 ----- Computation time = 0.0ms
```

Vectorization of Element-Wise Multiplication

In deep learning, you deal with very large datasets. Non-computationally-optimal functions become a huge bottleneck in your algorithms and can result in models that take ages to run. To make sure that your code is computationally efficient, you should use vectorization. Compare the following codes:

```
import time

x1 = [5, 1, 0, 3, 8, 2, 5, 6, 0, 1, 2, 5, 9, 0, 7]  # x1 = np.random.rand(1000000)
x2 = [2, 5, 2, 0, 3, 2, 2, 9, 1, 0, 2, 5, 4, 0, 9]  # x2 = np.random.rand(1000000)

### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))
for i in range(len(x1)):
    mul[i] = x1[i] * x2[i]
toc = time.process_time()
print ("for-looped elementwise multiplication = " + str(mul) + "\n ---- Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("vectorized elementwise multiplication = " + str(mul) + "\n ---- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
for-looped elementwise multiplication = [10.  5.  0.  0. 24.  4. 10. 54.  0.  0.  4. 25. 36.  0. 63.]
 ---- Computation time = 0.0ms
vectorized elementwise multiplication = [10  5  0  0 24  4 10 54  0  0  4 25 36  0 63]
 ---- Computation time = 0.0ms
```

Vectorization of General Dot Product

In deep learning, you deal with very large datasets. Non-computationally-optimal functions become a huge bottleneck in your algorithms and can result in models that take ages to run. To make sure that your code is computationally efficient, you should use vectorization. Compare the following codes:

```
import time

x1 = [5, 1, 0, 3, 8, 2, 5, 6, 0, 1, 2, 5, 9, 0, 7]  # x1 = np.random.rand(1000000)

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Random 3*len(x1) numpy array
tic = time.process_time()
gdot = np.zeros(W.shape[0])
for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j] * x1[j]
toc = time.process_time()
print ("for-looped gdot = " + str(gdot) + "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
gdot = np.dot(W,x1)
toc = time.process_time()
print ("vectorized gdot = " + str(gdot) + "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
gdot = [18.62176729 22.85934666 20.59097031]
 ----- Computation time = 0.0ms
gdot = [18.62176729 22.85934666 20.59097031]
 ----- Computation time = 0.0ms
```

Activation Functions of Neurons

We use different activation functions for neurons in different layers:

- **Sigmoid function is used in the output layer:**

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

- **Tangent hyperbolic function is used in hidden layers:**

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

- **Rectified linear unit (ReLu) is used in hidden layers (FAST!):**

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

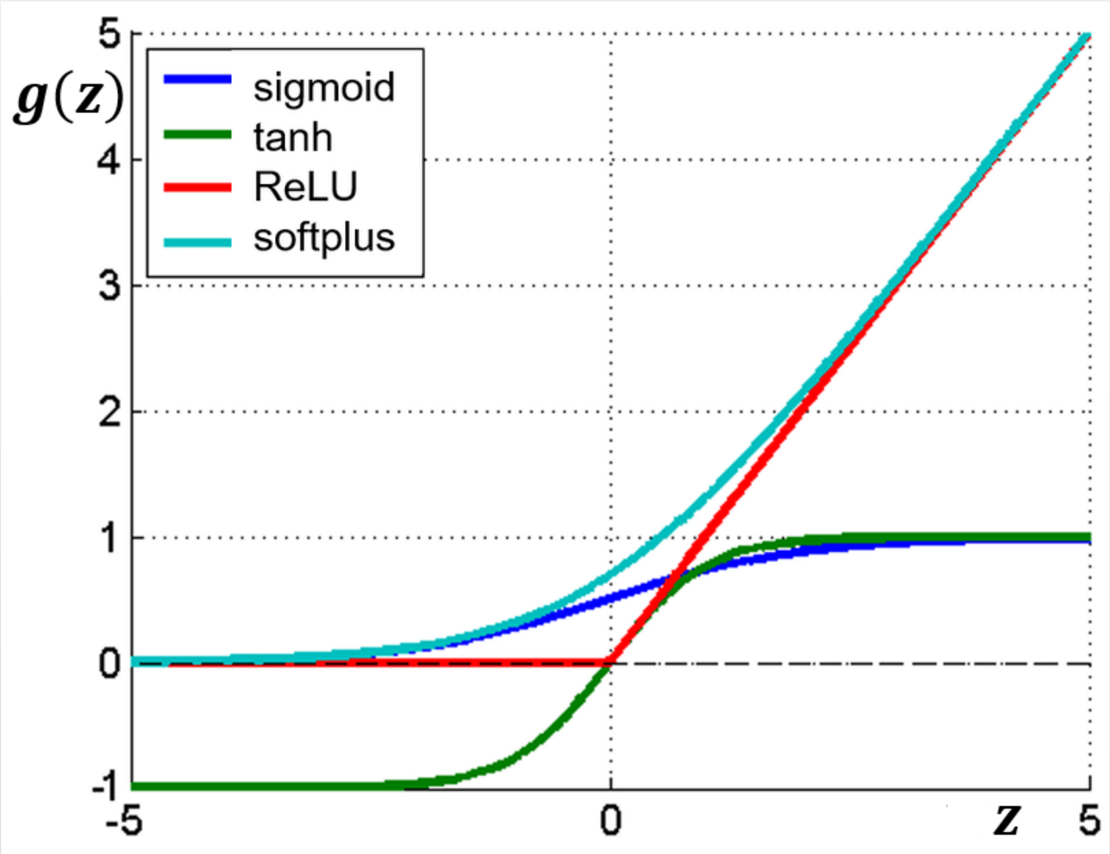
- **Smooth ReLu (SoftPlus) is used in hidden layers:**

  $g(z) = SoftPlus(z) = log(1 + e^{z})$

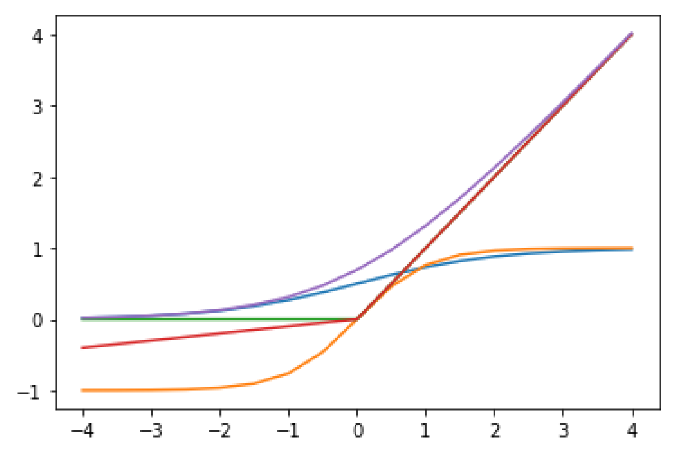
- **Leaky ReLu is used in hidden layers :**

- $g(z) = LeakyReLu(z) = \begin{cases} z & if\ z > 0 \\ 0.01z & if\ z \leq 0 \end{cases}$

COMPARISON OF ACTIVATION FUNCTIONS



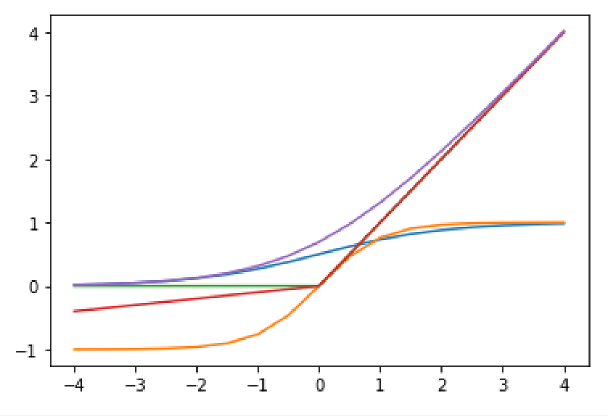
Activation Functions




```python
import numpy as np

def sigmoid(x):
    s = 1 / (1 + np.exp(-x))    # use np.exp to implement sigmoid activation function that works on a vector or a matrix
    return s

def tanh(x):
    t = np.tanh(x)    # np.tanh to implement tanh activation function that works on a vector or a matrix
    return t

def relu(x):
    r = np.maximum(0, x)    # use np.maximum to implement relu activation function that works on a vector or a matrix
    return r

def leakyrelu(x, slope):
    l = np.maximum(x * slope, x)    # use np.maximum to implement leaky relu activation function that works on a vector or a m
    return l

def softplus(x):
    p = np.log(1 + np.exp(x))    # use np.log and np.exp to implement softplus activation function that works on a vector or a
    return p
```

Derivatives of Activation Functions

Derivatives are necessary for the use of gradient descent:



- **Sigmoid function:**

$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}} \qquad g'(z) = \frac{dg(z)}{dz} = g(z)\cdot\big(1-g(z)\big) = a\cdot(1-a)$$

- **Tangent hyperbolic function:**

$$g(z) = tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \qquad g'(z) = \frac{dg(z)}{dz} = 1-\big(g(z)\big)^2 = 1-a^2$$

- **Rectified linear unit (ReLu):**

$$g(z) = ReLu(z) = max(0,z) \qquad g'(z) = \frac{dg(z)}{dz} = \begin{cases} 1 & if\ z > 0 \\ 0 & if\ z \leq 0 \end{cases}$$

- **Smooth ReLu (SoftPlus):**

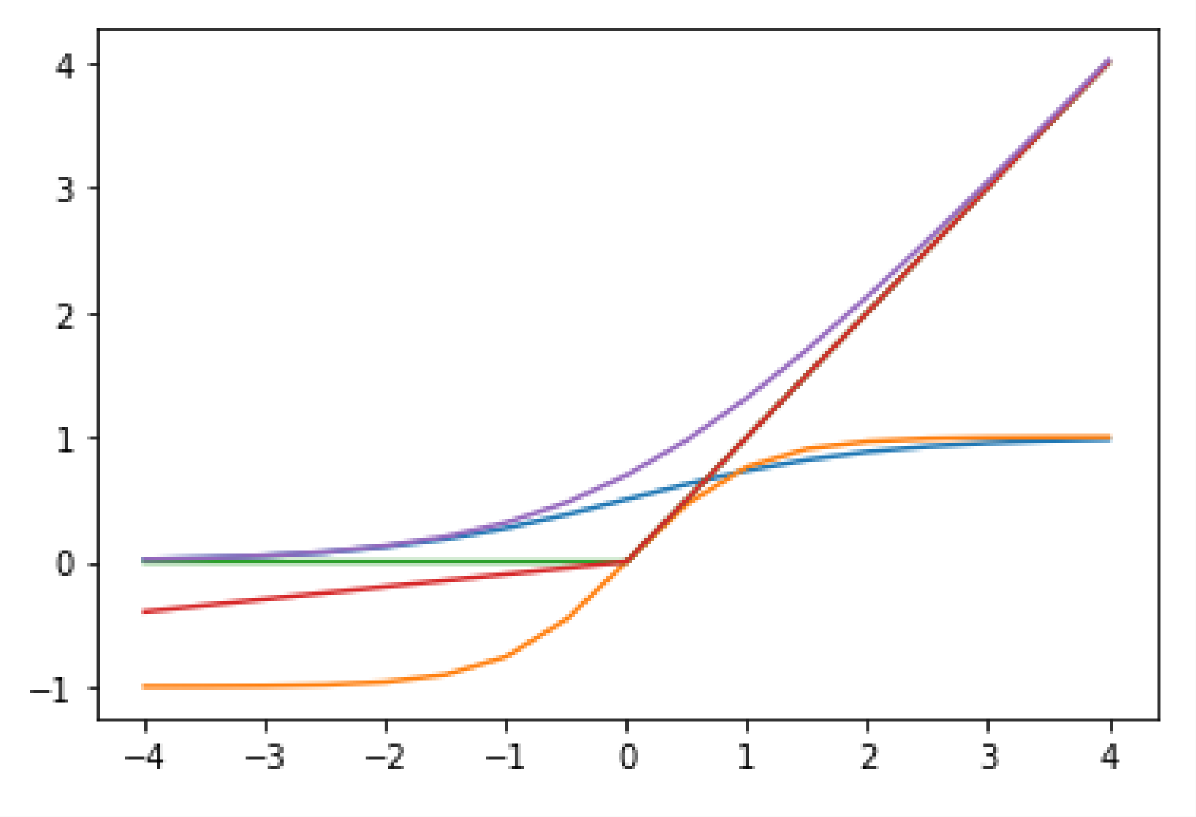
$$g(z) = SoftPlus(z) = ln(1+e^{z}) \qquad g'(z) = \frac{dg(z)}{dz} = \frac{e^{z}}{1+e^{z}} = \frac{1}{1+e^{-z}}$$

- **Leaky ReLu:**

$$g(z) = LeakyReLu(z) = \begin{cases} z & if\ z > 0 \\ 0.01z & if\ z \leq 0 \end{cases} \qquad g'(z) = \frac{dg(z)}{dz} = \begin{cases} 1 & if\ z > 0 \\ 0.01 & if\ z \leq 0 \end{cases}$$

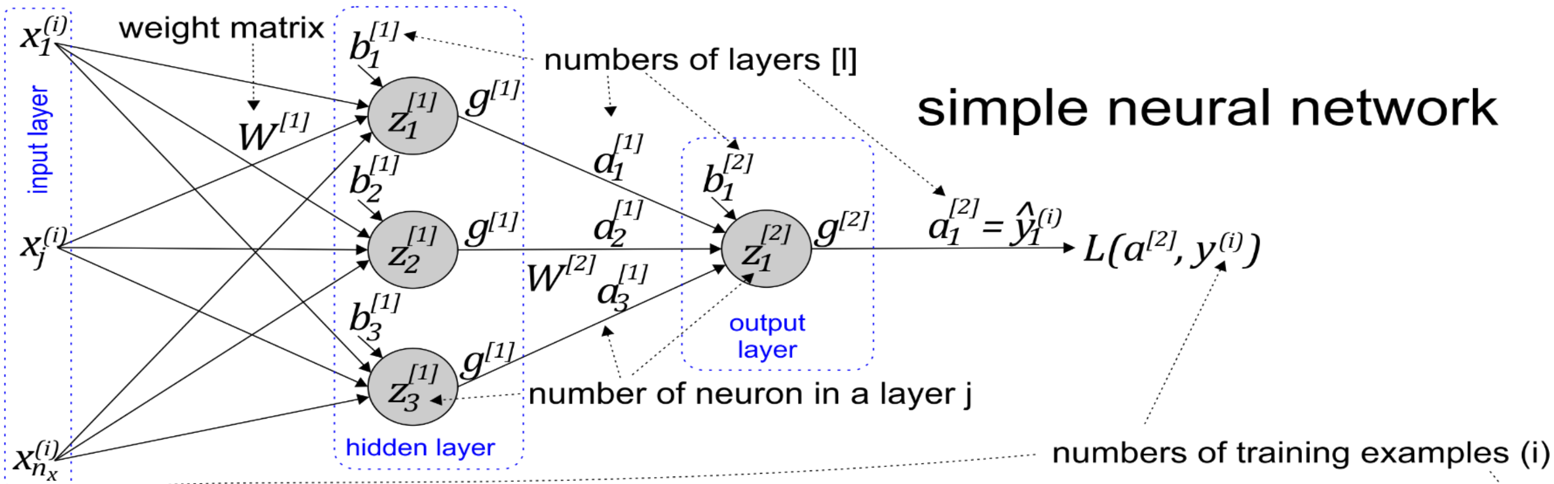
Derivatives of Activation Functions

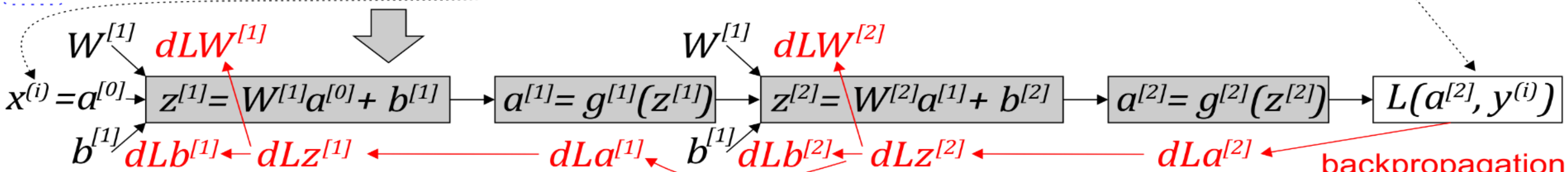
```python
def sigmoid_derivative(x):
    s = sigmoid(x)
    dLs = s * (1 - s)
    return dLs

def tanh_derivative(x):
    t = tanh(x)
    dLt = 1 - t * t
    return dLt

def relu_derivative(x):
    r = relu(x)
    dLr = np.heaviside(x, 0)
    return dLr

def leakyrelu_derivative(x, slope):
    l = leakyrelu(x, slope)
    dLl = np.ones_like(x)
    dLl[x < 0] = slope
    return dLl

def softplus_derivative(x):
    p = softplus(x)
    dLp = 1 / (1 + np.exp(-x))
    return dLp
```



Neural Network Gradients

simple neural network





COLLAPSING COMPUTATION

$dLa^{[1]} = W^{[2]T} \cdot dLz^{[2]}$

$dLz^{[1]} = dLa^{[1]} * g^{[1]\prime}(z^{[1]})$

$dLW^{[1]} = dLz^{[1]} \cdot a^{[0]T}$

$dLb^{[1]} = dLz^{[1]}$

COLLAPSING COMPUTATION

$dLa^{[2]} = -y^{(i)} \log a^{[2]} - (1-y^{(i)}) \cdot \log(1-a^{[2]}) = -y^{(i)}/a^{[2]} + (1-y^{(i)})/(1-a^{[2]})$

$dLz^{[2]} = dLa^{[2]} * g^{[2]\prime}(z^{[2]})$ * element-wise product

$dLW^{[2]} = dLz^{[2]} \cdot a^{[1]T}$

$dLb^{[2]} = dLz^{[2]}$

TRAINING EXAMPLES COLLAPSING AND VECTORIZATION

$dLZ^{[1]} = W^{[2]T} \cdot dLZ^{[2]} * g^{[1]\prime}(Z^{[1]})$

$dLW^{[1]} = \frac{1}{m} dLZ^{[1]} \cdot A^{[0]T}$

$dLb^{[1]} = \frac{1}{m} dLZ^{[1]}$

$dLZ^{[2]} = dLA^{[2]} * g^{[2]\prime}(Z^{[2]})$ * element-wise product

$dLW^{[2]} = \frac{1}{m} dLZ^{[2]} \cdot A^{[1]T}$

$dLb^{[2]} = \frac{1}{m} dLZ^{[2]}$

Random Initialization of Weights

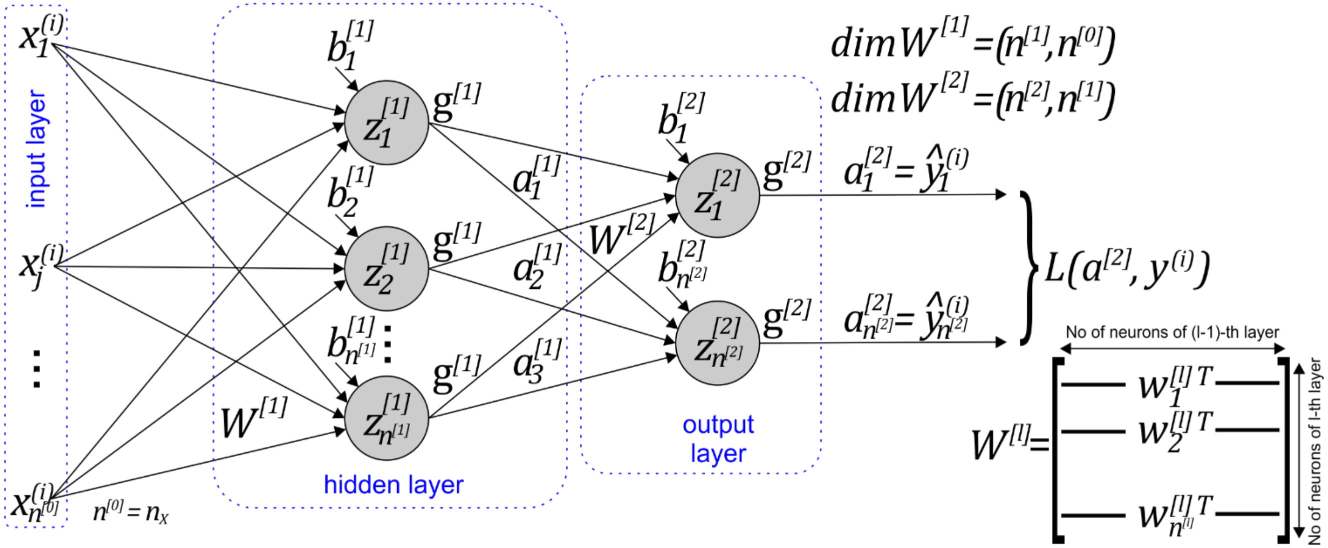
Parameters must be initialized by small random numbers:

- W cannot be initialized to 0:
- $W^{[l]} = np.random.randn\left(\left(n^{[l]}, n^{[l-1]}\right)\right) * 0.01$
- Small random initial weights values of the weights allow for faster training because the activation functions of neurons stimulated by values a little bit greater than 0 usually have the biggest slopes, so each update of weights results in big changes of output values and allows the network to move towards the solution faster.

- b can be initialized to 0:
- $b^{[l]} = np.zero\left(\left(n^{[l]}, 1\right)\right)$

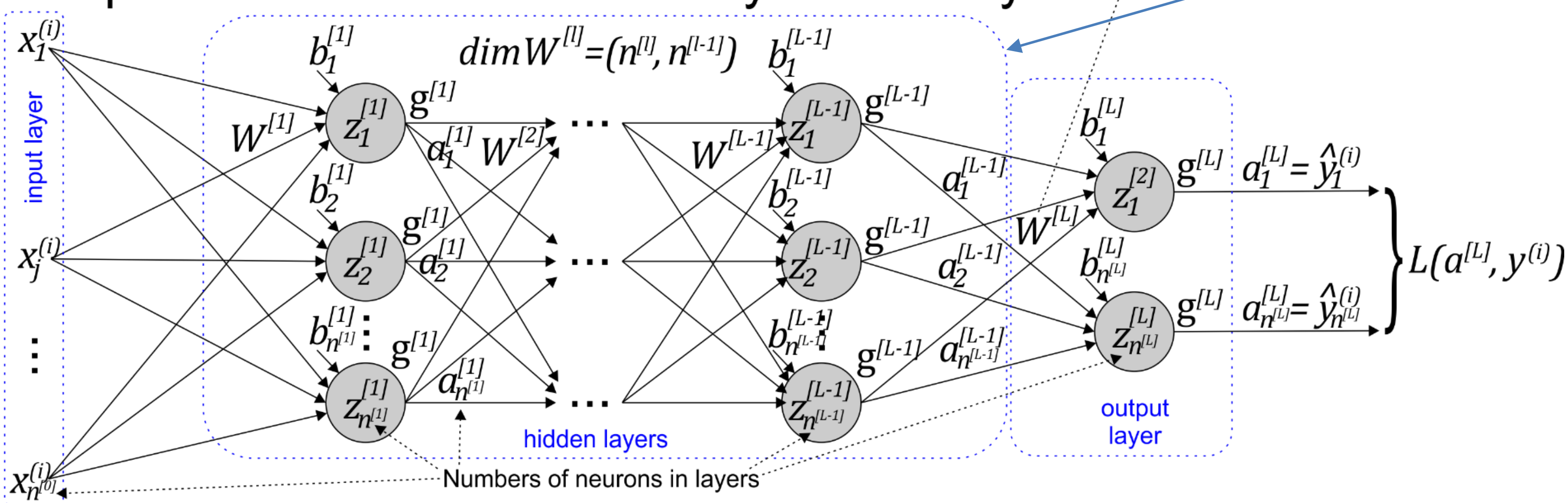
Going to Deeper NN Architectures

Shallow 2-layer NN architecture with 1 hidden layer



Deep neural network architecture means the use of many hidden layers between input and output layers.

Deep NN architecture with many hidden layers



Dimensions of Stacked Matrices

$$\underset{(n^{[l]},m)}{Z^{[l]}} = \underset{(n^{[l]},n^{[l-1]})}{W^{[l]}} \cdot \underset{(n^{[l-1]},m)}{A^{[l-1]}} + \underset{(n^{[l]},1)}{b^{[l]}}$$

broadcasted during addition

$$\underset{(n^{[l]},m)}{A^{[l]}} = g^{[l]}\big(\underset{(n^{[l]},m)}{Z^{[l]}}\big)$$

$$\begin{bmatrix} | & & | & & | \\ z^{[l](1)} & \dots & z^{[l](i)} & \dots & z^{[l](m)} \\ | & & | & & | \end{bmatrix} = \begin{bmatrix} — & w_1^{[l]\,T} & — \\ — & w_2^{[l]\,T} & — \\ & \vdots & \\ — & w_{n^{[l]}}^{[l]\,T} & — \end{bmatrix} \cdot \begin{bmatrix} | & & | & & | \\ a^{[l-1](1)} & \dots & a^{[l-1](i)} & \dots & a^{[l-1](m)} \\ | & & | & & | \end{bmatrix} + \begin{bmatrix} | \\ b^{[l]} \\ | \end{bmatrix}$$

$$\begin{bmatrix} | & & | & & | \\ a^{[l](1)} & \dots & a^{[l](i)} & \dots & a^{[l](m)} \\ | & & | & & | \end{bmatrix} = g^{[l]}\left(\begin{bmatrix} | & & | & & | \\ z^{[l](1)} & \dots & z^{[l](i)} & \dots & z^{[l](m)} \\ | & & | & & | \end{bmatrix}\right)$$

$$\begin{bmatrix} | & & | & & | \\ a^{[0](1)} & \dots & a^{[0](i)} & \dots & a^{[0](m)} \\ | & & | & & | \end{bmatrix} = \begin{bmatrix} | & & | & & | \\ x^{(1)} & \dots & x^{(i)} & \dots & x^{(m)} \\ | & & | & & | \end{bmatrix}$$

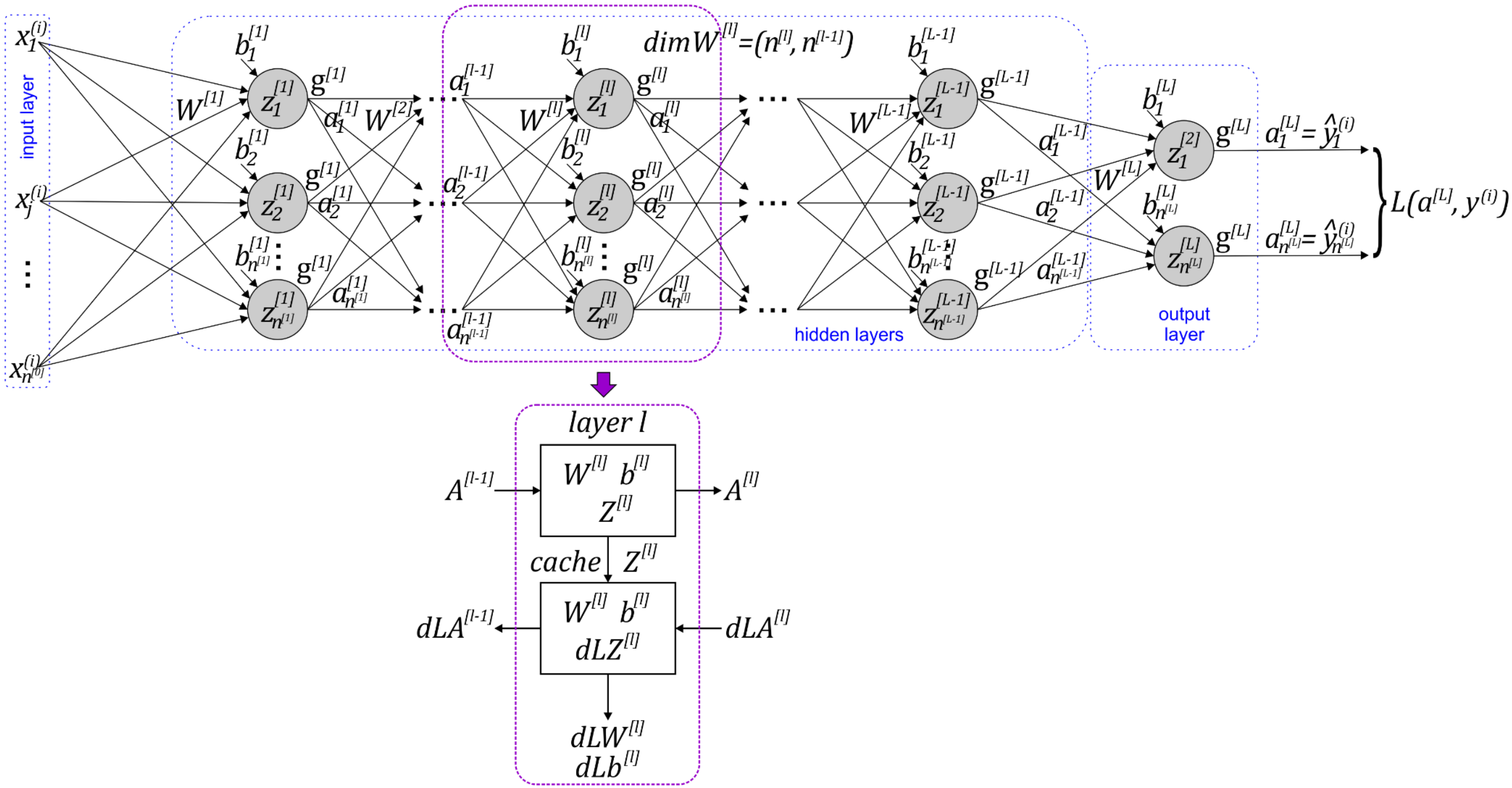
axis 0

$$\begin{bmatrix} | & & | & & | \\ a^{[0](1)} & \dots & a^{[0](i)} & \dots & a^{[0](m)} \\ | & & | & & | \end{bmatrix}$$

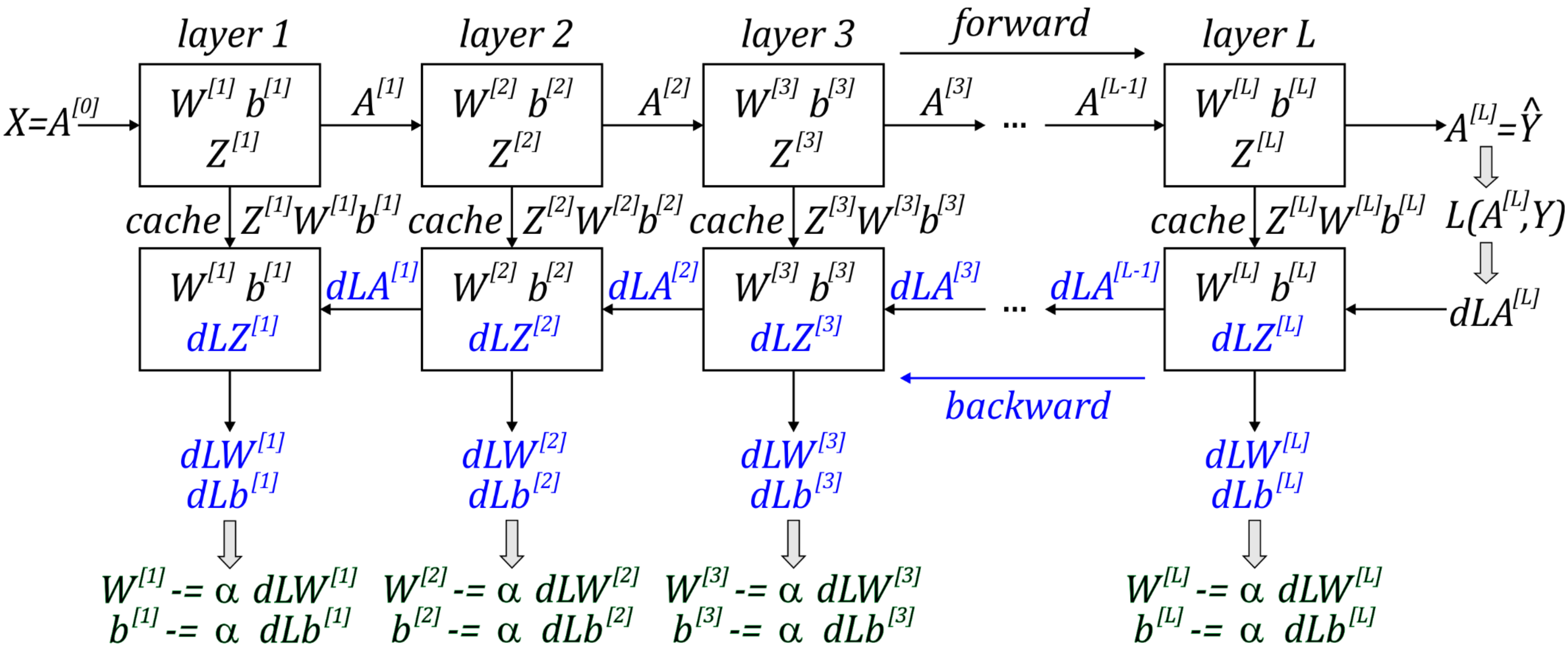
axis 1

$$\underset{(n^{[l]},m)}{dLZ^{[l]}} \qquad \underset{(n^{[l]},m)}{dLA^{[l]}} \qquad \underset{(n^{[l]},n^{[l-1]})}{dLW^{[l]}} \qquad \underset{(n^{[l]},1)}{dLb^{[l]}}$$

Building Blocks of Deep Neural Networks



Stacking Building Blocks Subsequently



$$dLA^{[L]} = \left(-\frac{y^{(1)}}{a^{(1)}} + \frac{1-y^{(1)}}{1-a^{(1)}}, \ldots, \frac{y^{(m)}}{a^{(m)}} + \frac{1-y^{(m)}}{1-a^{(m)}}\right)$$

$$dLA^{[l]} = W^{[l+1]T} \cdot dLZ^{[l+1]}$$

$$dLZ^{[l]} = dLA^{[l]} * g^{[l]\prime}(Z^{[l]})$$

$$dLZ^{[l]} = W^{[l+1]T} \cdot dLZ^{[l+1]} * g^{[l]\prime}(Z^{[l]})$$

* element-wise product

$$dLW^{[l]} = \frac{1}{m} \cdot dLZ^{[l]} \cdot A^{[l-1]T}$$

$$dLb^{[l]} = \frac{1}{m} \cdot np.sum(dLZ^{[l]}, axis=1, keepdims=True)$$

$$dLA^{[l-1]} = W^{[l]T} \cdot dLZ^{[l]}$$

Parameters and Hyperparameters

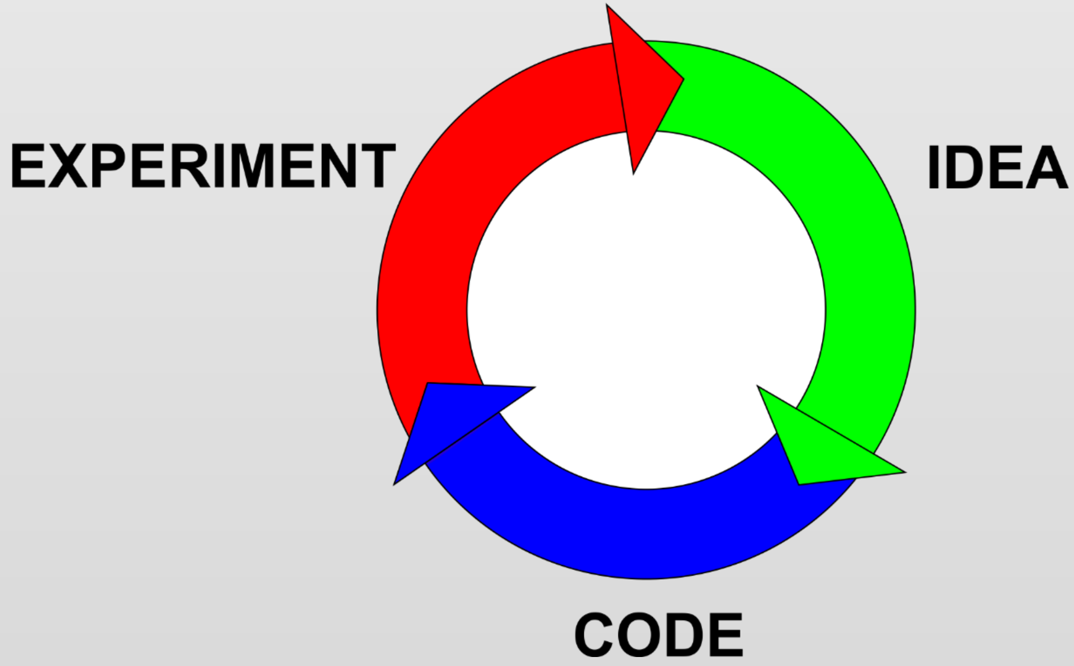
We should distinguish between parameters and hyperparameters:

- Parameters of the model are established during the training process, e.g.:
  - $W^{[l]}, b^{[l]}$.
- Hyperparameters control parameters and are established by the developer of the model, e.g.:
  - α – learning rate,
  - L – number of hidden layers,
  - $n^{[l]}$ - number of neurons in layers,
  - $g^{[l]}$ - choice of activation functions for layers,
  - number of iterations over training data,
  - momentum,
  - minibatch size,
  - regularization parameters,
  - optimization parameters,
  - dropout parameters, …

Iterative Development of DL Solutions

Deep Learning solutions are usually developed in an iterative and empirical process that composes of three main elements:

- **Idea – when we suppose that a selected model, training method, and some hyperparameters let us to solve the problem.**
- **Code – when we try to code and apply the idea in a real code.**
- **Experiment – prove our suppositions and assumptions or not, and allow to update or change the idea until the experiments return satisfactory results.**



Let’s start with powerful computations!

- ✓ **Questions?**
- ✓ **Remarks?**
- ✓ **Suggestions?**
- ✓ **Wishes?**

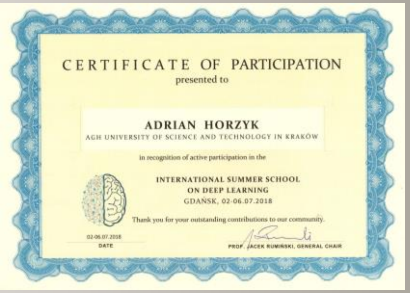
Bibliography and Literature

1. Nikola K. Kasabov, *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016, ISBN 978-1-59327-741-3 or PWN 2018.
3. Holk Cruse, *Neural Networks as Cybernetic Systems*, 2nd and revised edition
4. R. Rojas, *Neural Networks*, Springer-Verlag, Berlin, 1996.
5. *Convolutional Neural Network* (Stanford)
6. *Visualizing and Understanding Convolutional Networks*, Zeiler, Fergus, ECCV 2014
7. IBM: https://www.ibm.com/developerworks/library/ba-data-becomes-knowledge-1/index.html
8. NVIDIA: https://developer.nvidia.com/discover/convolutional-neural-network
9. JUPYTER: https://jupyter.org/

Adrian Horzyk

horzyk@agh.edu.pl

Google: Horzyk



AGH

University of Science and Technology in Krakow, Poland