ARTIFICIAL AND COMPUTATIONAL INTELLIGENCE AND KNOWLEDGE ENGINEERING

Cutting-Edge Python Computational Tools for AI, CI, KE and Data Mining

Adrian Horzyk

horzyk@agh.edu.pl

AGH

AGH University of Science and Technology Krakow, Poland

Scope

- ✓ Python – a modern language often used for AI, CI, KE, and DM computing
- ✓ Jupyter Notebook – a modern and intuitive programming environment with linked libraries (like Tensorflow, Keras) that allow to effectively process deep learning algorithms and show results quickly.
- ✓ Tensorflow and Keras libraries produced by leading IT companies, like Google.

Jupyter Notebook

The Jupyter Notebook:

- is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text;
- includes data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

We will use it to demonstrate various algorithms, so you are asked to install it.

Jupyter in your browser

Install a Jupyter Notebook

Jupyter Notebook & Anaconda

Install Jupyter using Anaconda with built in Python 3.7+

- It includes many other commonly used packages for scientific computing, data science, machine learning ,and computational intelligence libraries.
- It manages libraries, dependencies, and environments with Conda.
- It allows developing and training various machine learning and deep learning models with scikit-learn, TensorFlow, and Theano etc.
- It supplies us with data analysis including scalability and performance with Dask, NumPy, pandas, and Numba.
- It quickly visualizes results with Matplotlib, Bokeh, Datashader, and Holoviews.

And run it at the Terminal (Mac/Linux) or Command Prompt (Windows):

```
jupyter notebook
```

Anaconda Cloud

ANACONDA CLOUD | Search Anaconda Cloud | View | Help | adrianhorz...

My Anaconda Landscape

Packages — View all (0)

Get more information on how to upload a Package.

Notebooks — View all (0)

Get more information on how to upload a Notebook.

Environments — View all (0)

Get more information on how to upload an Environment.

Projects — View all (0)

No projects yet, upload one here.

Favorites — View all (0)

Favorite some packages, notebooks, and environments to get started!

Activity Feed — View more

Welcome to Anaconda Cloud! 10 months and 22 hours ago

Anaconda Cloud allows you to create or distribute software packages.

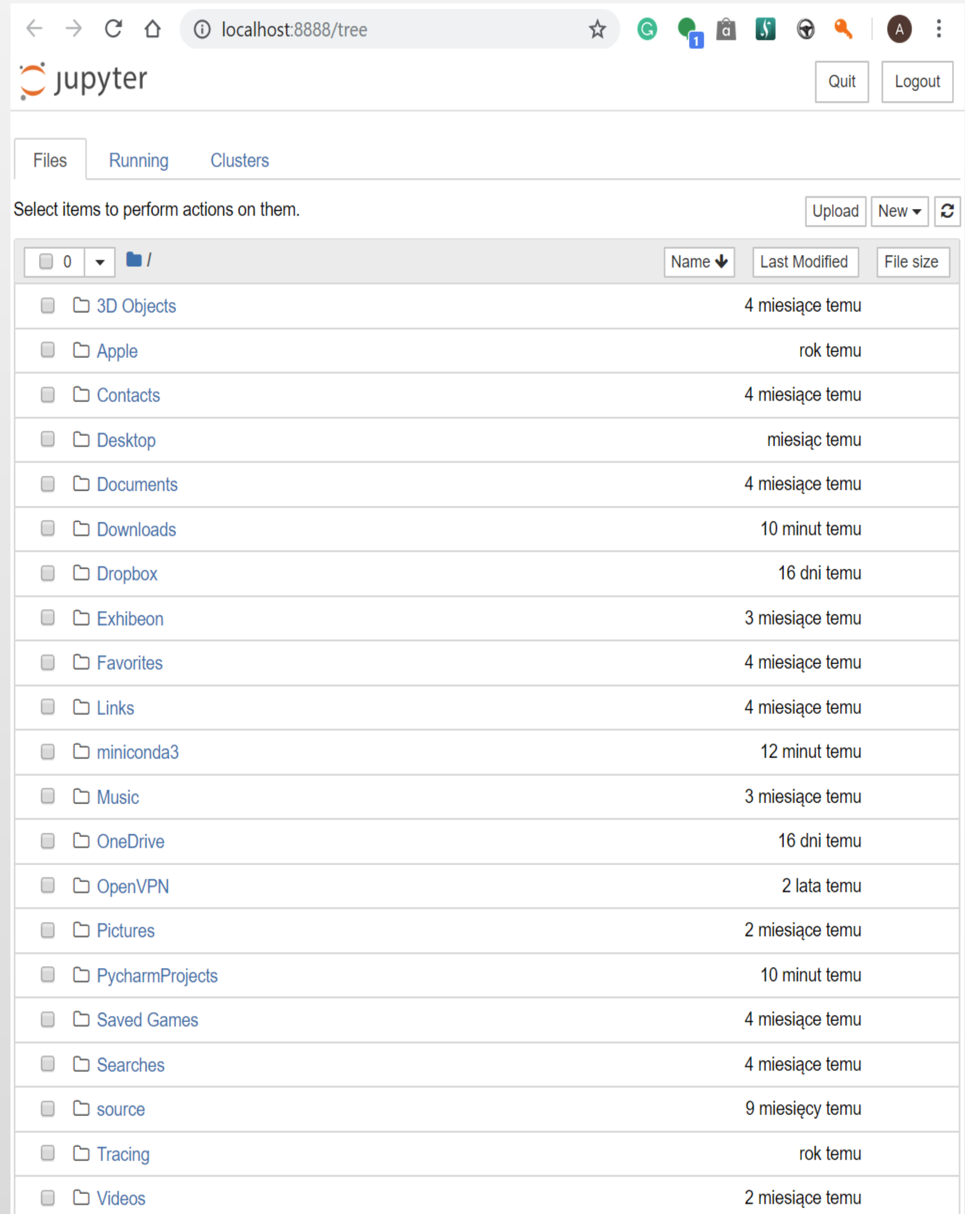
Getting started: Installing your first package

Getting started: Distributing your first package

Jupyter Notebook Dashboard

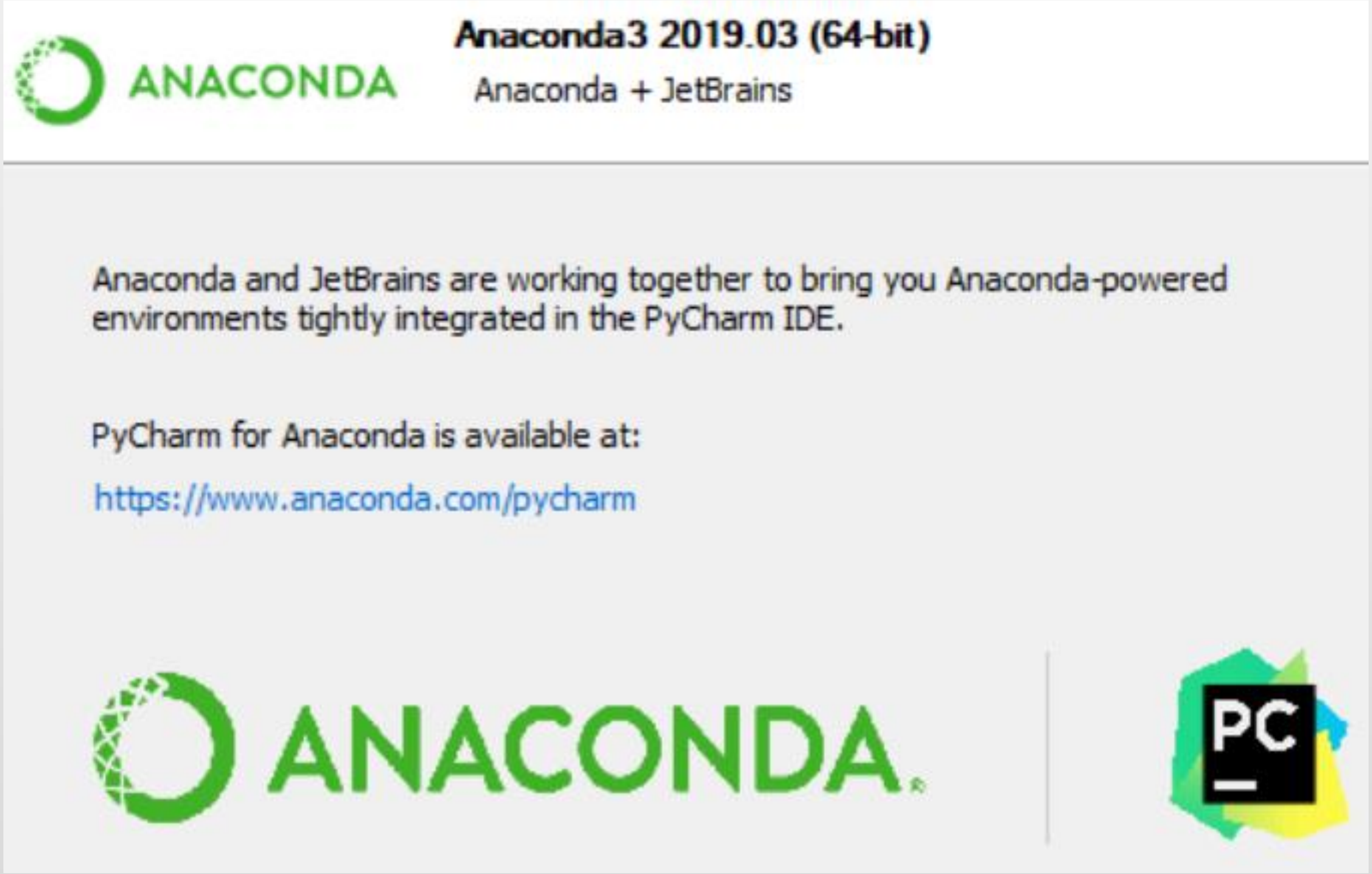
Running a Jupyter Notebook in your browser:

- When the Jupyter Notebook opens in your browser, you will see the Notebook Dashboard, which will show you a list of the notebooks, files, and subdirectories in the directory where the notebook server was started by the command line „jupyter notebook”.
- Most of the time, you will wish to start a notebook server in the highest level directory containing notebooks. Often this will be your home directory.



Jupyter Notebook & PyCharm

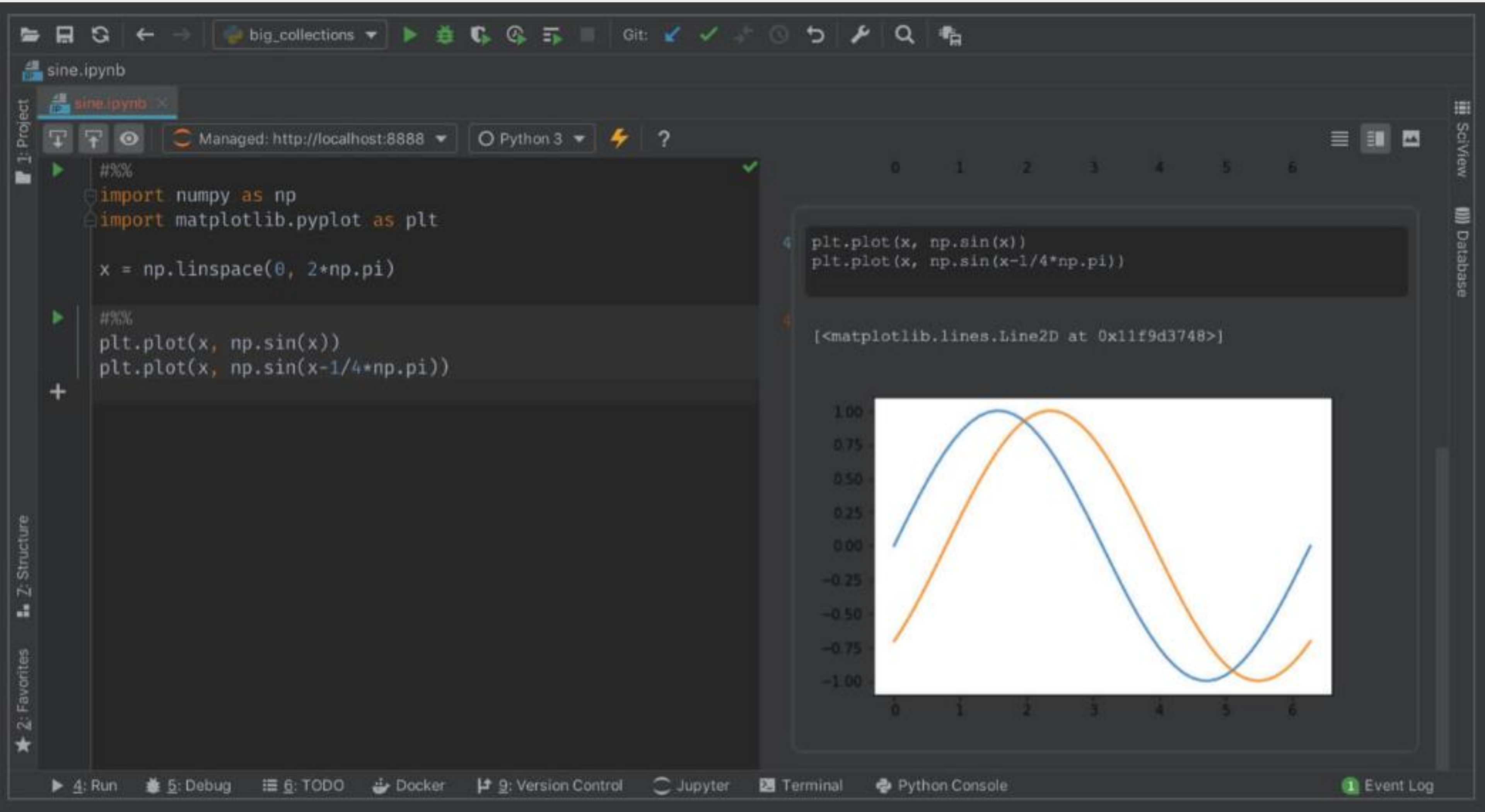
It is recommended to install PyCharm for Anaconda:



Jupyter Notebook

PyCharm is a python IDE for Professional Developers

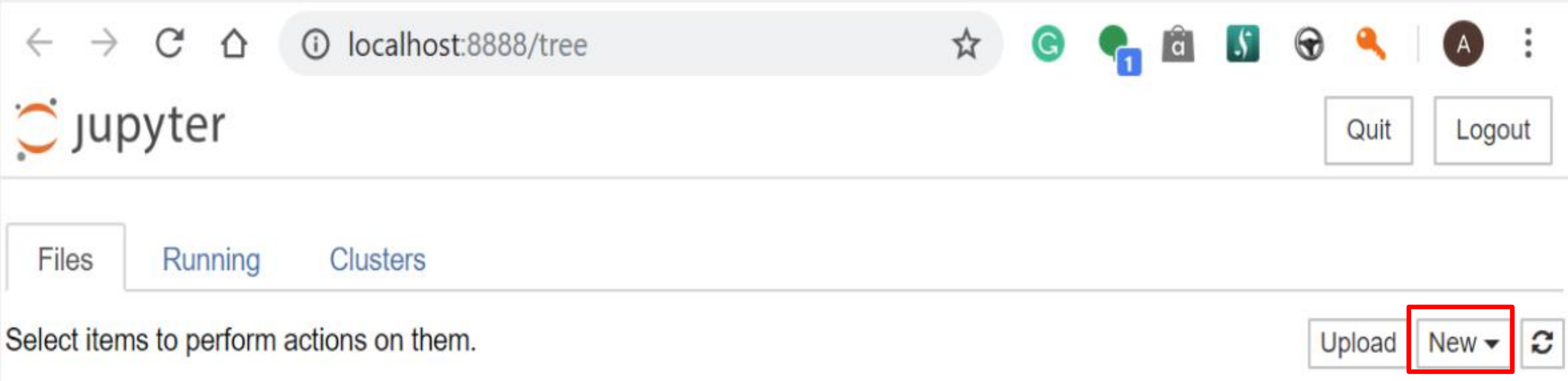
- It includes scientific mode to interactively analyze your data.



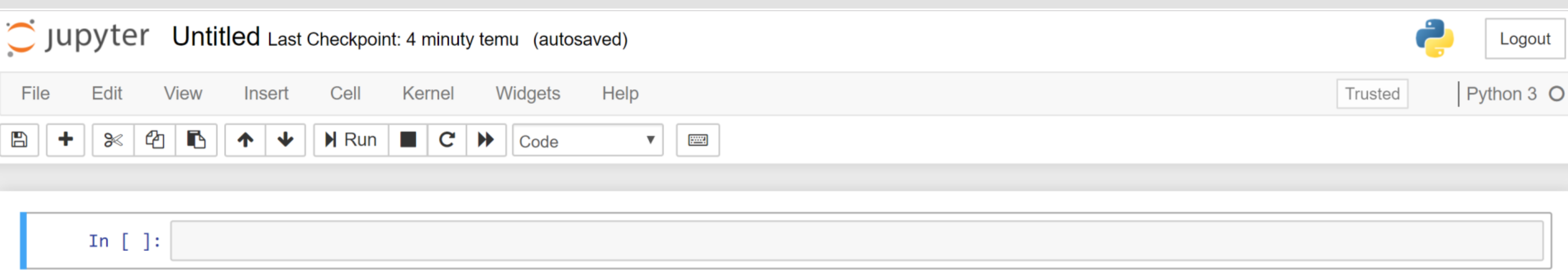
Starting a new Python notebook

Start a new Python notebook:

- **Clicking New → Python 3**



- **And a new Python project in the Jupyter Notebook will be started:**



Efficiency of Vectorization

When dealing with big data collections and big data vectors, we should use vectorization to proceed computations faster:

jupyter Comparison of for-looped and vectorized efficiency of computations Last Checkpoint: 3 godziny temu (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

In [4]:
```python
import numpy as np

a = np.array([1,2,3,4])
print (a)

import time

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

tic = time.time()
c = np.dot(a,b)
toc = time.time()

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

tic = time.time()
for i in range(1000000):
    d += a[i]*b[i]
toc = time.time()

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

```
[1 2 3 4]
Vectorized dot product computation time:0.9996891021728516ms
For-Looped dot product computation time:420.8984375ms
```

Conclusion:
Whenever possible, avoid explicit for-loops and use vectorization:
np.dot(A,v), np.dot(x,w), np.log(v), np.abs(v), np.zeros(v), np.sum(v) etc.

In [ ]:

Computing Sigmoid Function

We use numpy vectorization to compute sigmoid(x) and sigmoid_derivative for any input vector x:

$$\text{For } x \in \mathbb{R}^n \text{, } sigmoid(x) = sigmoid\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} \frac{1}{1+e^{-x_1}} \\ \frac{1}{1+e^{-x_2}} \\ \dots \\ \frac{1}{1+e^{-x_n}} \end{pmatrix} \tag{1}$$

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

In [14]:
```
import numpy as np # You can access numpy functions by writing np.function() instead of numpy.function()

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

def sigmoid_derivative(x):
    s = sigmoid(x)
    ds = s * (1 - s) # It computes the gradient (slope or derivative) of the sigmoid function with respect to its input x.
    return ds
```

In [15]:
```
x = np.array([-2,-1,0,1, 2])
print ("sigmoid(x) = " + str(sigmoid(x)))
print ("sigmoid_derivative(x) = " + str(sigmoid_derivative(x)))
```

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

Normalization for Efficiency

We use normalization 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]]
```

Broadcasting in numpy

Broadcasting is very useful for performing mathematical operations between arrays of different shapes.

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:

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

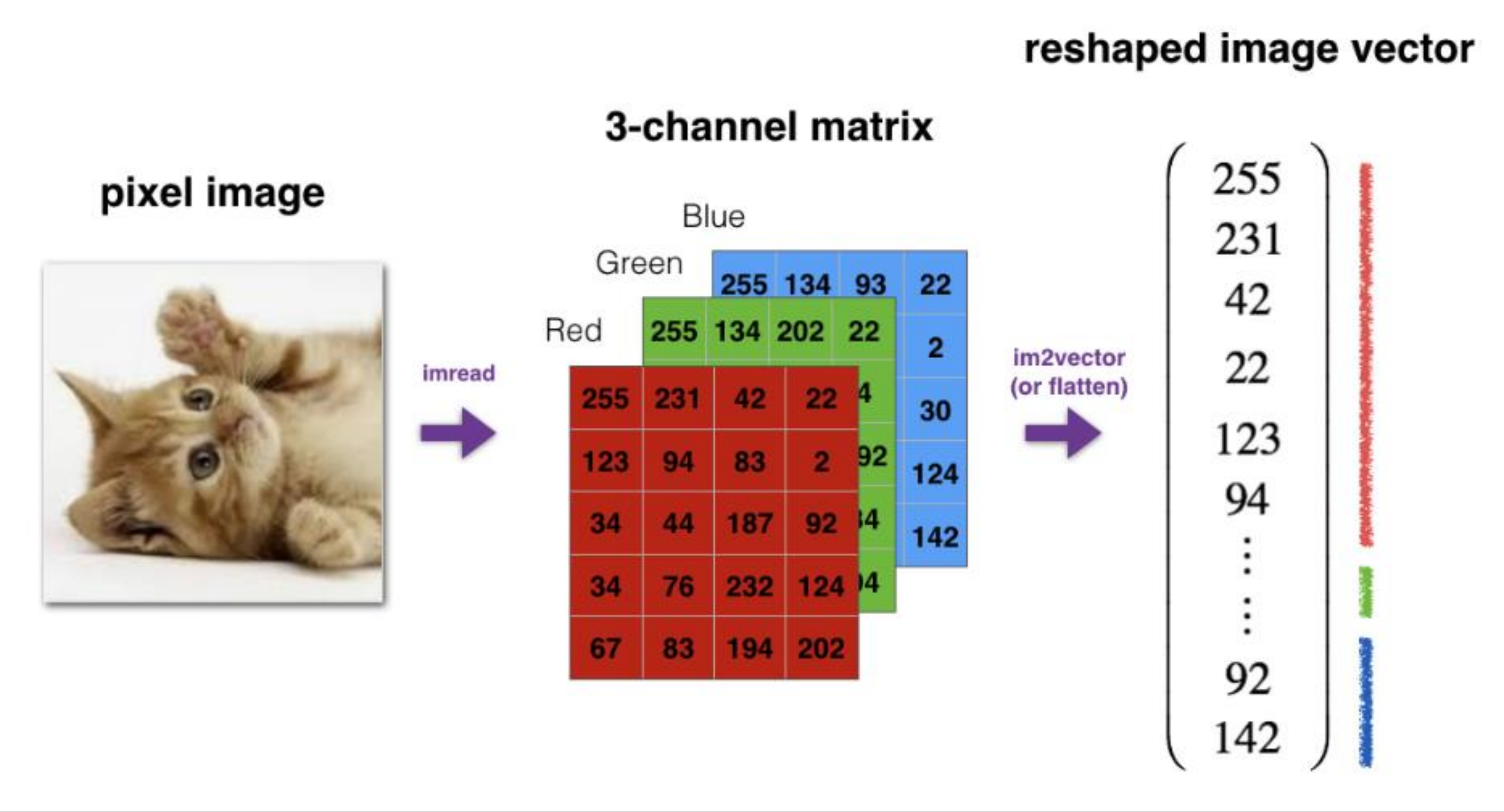
In [27]:
```
def softmax(x):
    # This function calculates the softmax for each row of the input x, where x is a row vector or a matrix of shape (n, m).
    x_exp = np.exp(x)
    x_sum = np.sum(x_exp,axis=1,keepdims=True)
    s = x_exp/x_sum  # It automatically uses numpy broadcasting.
    return s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

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]]
```

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]]
```

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:

In [4]:

```
import time
import numpy as np

x1 = [2, 6, 3, 9, 1, 0, 2, 0, 6, 1, 9, 3, 5, 2, 1]
x2 = [6, 3, 1, 1, 7, 8, 2, 5, 2, 4, 0, 2, 3, 0, 9]

### 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 ("dot = " + str(dot))
print ("Computation time of the classic dot product of vector implementation = " + str(1000 * (toc - tic)) + "ms")

### VECTORIZED DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("dot = " + str(dot))
print ("Computation time of the vectorized dot product of vectors implementation = " + str(1000 * (toc - tic)) + "ms")
```

```
dot = 99
Computation time of the classic dot product of vector implementation = 0.0ms
dot = 99
Computation time of the vectorized dot product of vectors implementation = 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:

In [5]:
```python
import time
import numpy as np

x1 = [2, 6, 3, 9, 1, 0, 2, 0, 6, 1, 9, 3, 5, 2, 1]
x2 = [6, 3, 1, 1, 7, 8, 2, 5, 2, 4, 0, 2, 3, 0, 9]

### 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 ("outer = " + str(outer))
print ("Computation time of the classic outer product of vector implementation = " + str(1000 * (toc - tic)) + "ms")

### VECTORIZED OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
print ("outer = " + str(outer))
print ("Computation time of the vectorized outer product of vectors implementation = " + str(1000 * (toc - tic)) + "ms")
```

```
outer = [[12.  6.  2.  2. 14. 16.  4. 10.  4.  8.  0.  4.  6.  0. 18.]
 [36. 18.  6.  6. 42. 48. 12. 30. 12. 24.  0. 12. 18.  0. 54.]
 [18.  9.  3.  3. 21. 24.  6. 15.  6. 12.  0.  6.  9.  0. 27.]
 [54. 27.  9.  9. 63. 72. 18. 45. 18. 36.  0. 18. 27.  0. 81.]
 [ 6.  3.  1.  1.  7.  8.  2.  5.  2.  4.  0.  2.  3.  0.  9.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [12.  6.  2.  2. 14. 16.  4. 10.  4.  8.  0.  4.  6.  0. 18.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [36. 18.  6.  6. 42. 48. 12. 30. 12. 24.  0. 12. 18.  0. 54.]
 [ 6.  3.  1.  1.  7.  8.  2.  5.  2.  4.  0.  2.  3.  0.  9.]
 [54. 27.  9.  9. 63. 72. 18. 45. 18. 36.  0. 18. 27.  0. 81.]
 [18.  9.  3.  3. 21. 24.  6. 15.  6. 12.  0.  6.  9.  0. 27.]
 [30. 15.  5.  5. 35. 40. 10. 25. 10. 20.  0. 10. 15.  0. 45.]
 [12.  6.  2.  2. 14. 16.  4. 10.  4.  8.  0.  4.  6.  0. 18.]
 [ 6.  3.  1.  1.  7.  8.  2.  5.  2.  4.  0.  2.  3.  0.  9.]]
Computation time of the classic outer product of vector implementation = 0.0ms
```

```
outer = [[12  6  2  2 14 16  4 10  4  8  0  4  6  0 18]
 [36 18  6  6 42 48 12 30 12 24  0 12 18  0 54]
 [18  9  3  3 21 24  6 15  6 12  0  6  9  0 27]
 [54 27  9  9 63 72 18 45 18 36  0 18 27  0 81]
 [ 6  3  1  1  7  8  2  5  2  4  0  2  3  0  9]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [12  6  2  2 14 16  4 10  4  8  0  4  6  0 18]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [36 18  6  6 42 48 12 30 12 24  0 12 18  0 54]
 [ 6  3  1  1  7  8  2  5  2  4  0  2  3  0  9]
 [54 27  9  9 63 72 18 45 18 36  0 18 27  0 81]
 [18  9  3  3 21 24  6 15  6 12  0  6  9  0 27]
 [30 15  5  5 35 40 10 25 10 20  0 10 15  0 45]
 [12  6  2  2 14 16  4 10  4  8  0  4  6  0 18]
 [ 6  3  1  1  7  8  2  5  2  4  0  2  3  0  9]]
Computation time of the vectorized outer product of vectors implementation = 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:

In [8]:

```python
import time
import numpy as np

x1 = [2, 6, 3, 9, 1, 0, 2, 0, 6, 1, 9, 3, 5, 2, 1]
x2 = [6, 3, 1, 1, 7, 8, 2, 5, 2, 4, 0, 2, 3, 0, 9]

### CLASSIC ELEMENTWISE MULTIPLICATION 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 ("elementwise multiplication = " + str(mul))
print ("Computation time of the classic element-wise implementation = " + str(1000 * (toc - tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION IMPLEMENTATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("elementwise multiplication = " + str(mul))
print ("Computation time of the vectorized element-wise implementation = " + str(1000 * (toc - tic)) + "ms")
```

```
elementwise multiplication = [12. 18.  3.  9.  7.  0.  4.  0. 12.  4.  0.  6. 15.  0.  9.]
Computation time of the classic element-wise implementation = 0.0ms
elementwise multiplication = [12 18  3  9  7  0  4  0 12  4  0  6 15  0  9]
Computation time of the vectorized element-wise implementation = 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:

In [7]:

```
import time
import numpy as np

x1 = [2, 6, 3, 9, 1, 0, 2, 0, 6, 1, 9, 3, 5, 2, 1]
x2 = [6, 3, 1, 1, 7, 8, 2, 5, 2, 4, 0, 2, 3, 0, 9]

### 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 ("gdot = " + str(gdot));
print ("Computation time of the classic general dot product of vector implementation = " + str(1000 * (toc - tic)) + "ms")

### VECTORIZED GENERAL DOT PRODUCT IMPLEMENTATION ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("gdot = " + str(dot))
print ("Computation time of the vectorized general dot product of vectors implementation = " + str(1000 * (toc - tic)) + "ms")
```

```
gdot = [29.7697031  21.47839099 29.65064192]
Computation time of the classic general dot product of vector implementation = 0.0ms
gdot = [29.7697031  21.47839099 29.65064192]
Computation time of the vectorized general dot product of vectors implementation = 0.0ms
```

Loss Functions

The loss functions are used to evaluate the performance of your 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 function using absolute distance of y and $\hat{y}$ is defined as:

$$L_1(\hat{y}, y) = \sum_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}| \tag{6}$$

L2 loss function using squared distance of y and $\hat{y}$ is defined as:

$$L_2(\hat{y}, y) = \sum_{i=0}^{m} (y^{(i)} - \hat{y}^{(i)})^2 \tag{7}$$

In [40]:
```
def L1(yhat, y):
    # This function returns the the L1 loss function where
    # yhat - vector of size m (predicted labels)
    # y - vector of size m (true labels)
    loss = np.sum(np.abs(y-yhat))
    return loss

def L2(yhat, y):
    # This function returns the the L2 loss function where
    # yhat - vector of size m (predicted labels)
    # y - vector of size m (true labels)
    loss = np.sum(np.dot(y-yhat,y-yhat))
    return loss
```

In [41]:
```
yhat = np.array([.34, 0.98, 0.21, .15, .12])
y = np.array([0, 1, 0, 0, 0])
print("L1 = " + str(L1(yhat,y)))
print("L2 = " + str(L2(yhat,y)))
```

```
L1 = 0.8400000000000001
L2 = 0.19700000000000004
```

Let’s start with powerful computations!

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

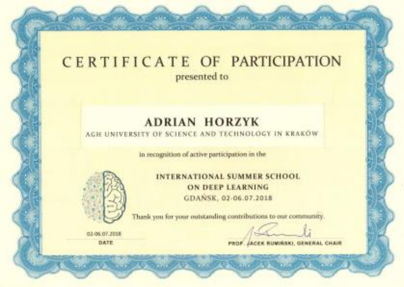
Bibliography and Literature

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