KNOWLEDGE-BASED COMPUTATIONAL INTELLIGENCE AND DATA MINING AND BIOMEDICINE

Deep Learning of Convolutional Neural Networks in Python Frameworks and Notebooks

Adrian Horzyk

horzyk@agh.edu.pl

AGH University of Science and Technology Krakow, Poland

Your Current Understanding

What do you mean by:

- Knowledge?
- Intelligence?
- Artificial Intelligence?
- Computational Intelligence?
- Data Mining?

Important Question

What would you like to learn during this course?

Scope

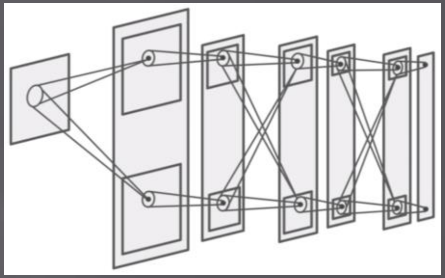
We will address the following topics:

- Models and methods of Computational Intelligence for classification, detection, regression, and clustering.
- Frameworks and notebooks for the development of Computational Intelligence (CI) and Data Mining (DM) solutions in Python.
- Deep learning of classic and convolutional models with tuning, regularization, optimization of models and learning strategies.
- Classic data mining models for frequent pattern search.
- Associative knowledge-based systems for similarity search and recommendation systems.

Deep Learning

Deep learning (also known as hierarchical learning) is a class of machine learning algorithms and learning strategies that:

- ✓ Develop hierarchical deep structures and representation of primary and secondary (derived) features, representing different levels of abstraction.
- ✓ Use a cascade of many layers of neurons (or other processing units) of various kinds for gradual feature extraction and their transformation to achieve a hierarchy of secondary, derived features which can led to better final results of such constructed neural network. In this way, they try to determine higher level features which are derived from lower level features.
- ✓ Apply various supervised and unsupervised
- ✓ learning strategies to various layers.
- ✓ Gradually upgrade and develop a structure until
- ✓ significant improvement in performance is achieved.

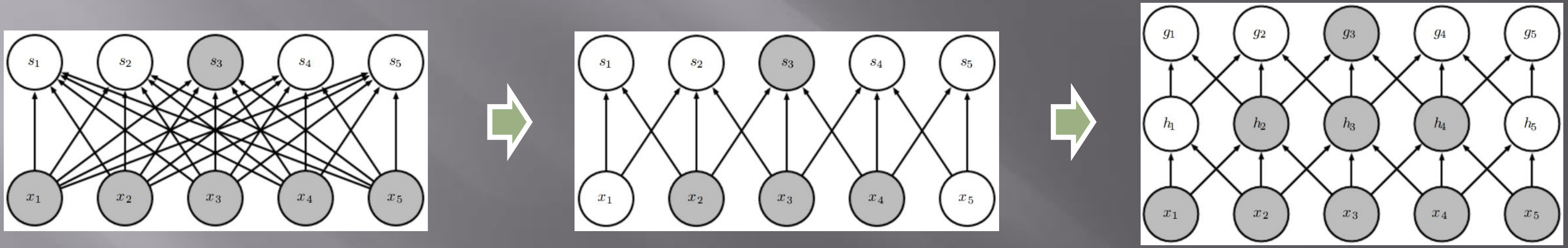


Deep learning Convolutional Neural Networks are mostly popular today because they allow achieving high-quality results. They were inspired by biological retina and proposed by Yann LeCun in 1998 using Fukushima’s Cognitron and Neocognitron (a model of neurons).

CNN to Biomedical Image Classification

Deep learning strategies assume the ability to:

- ✓ update only a selected part of neurons that respond best to the given input data, so the other neurons and their parameters (e.g. weights, thresholds) are not updated,
- ✓ avoid connecting all neurons between successive layers, so we do not use all-to-all connection strategy known and commonly used in MLP and other networks, but we try to allow neurons to specialize in recognizing of subpatterns that can be extracted from the limited subsets of inputs,



s_1 s_2 s_3 s_4 s_5
x_1 x_2 x_3 x_4 x_5
s_1 s_2 s_3 s_4 s_5
x_1 x_2 x_3 x_4 x_5
g_1 g_2 g_3 g_4 g_5
h_1 h_2 h_3 h_4 h_5
x_1 x_2 x_3 x_4 x_5

- ✓ create connections between various layers and subnetworks, not only between successive layers
- ✓ use many subnetworks that can be connected in different ways in order to allow neurons from these subnetworks to specialize in defining or recognizing of a limited subsets of features or subpatterns,
- ✓ let neurons specialize and not overlap represented regions and represent the same features or subpatterns.

Convolutional Neural Networks (CNN)

For classification of images where objects can be located in different places of the image, Convolutional Neural Networks are especially useful because their convolutional layers are insensitive for shifting the objects in the image, and they still work correctly.

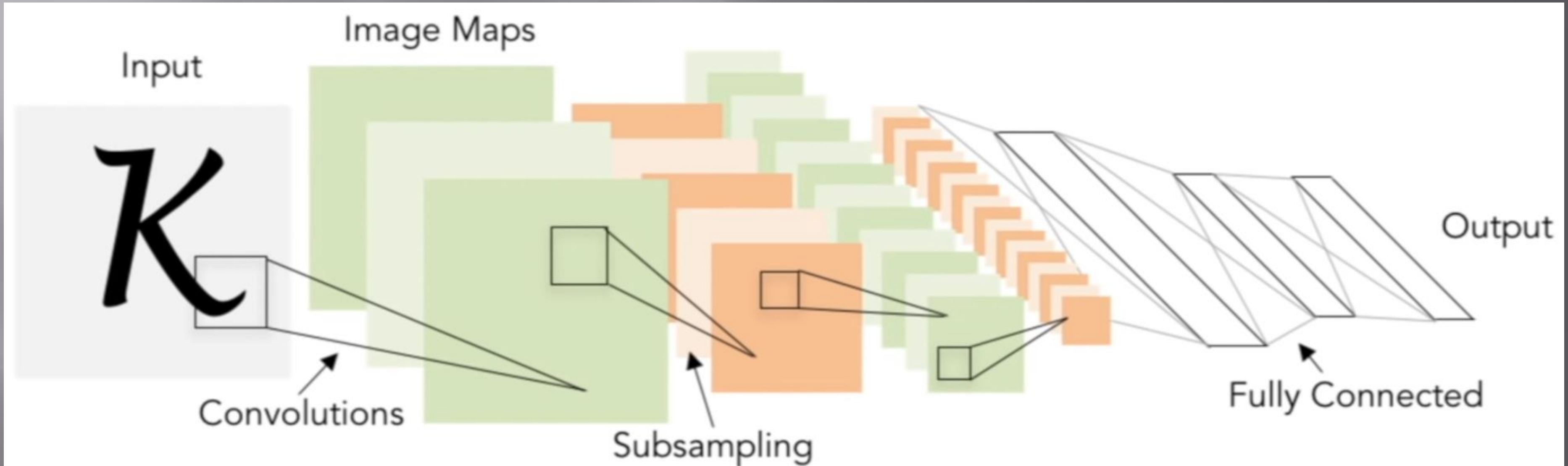


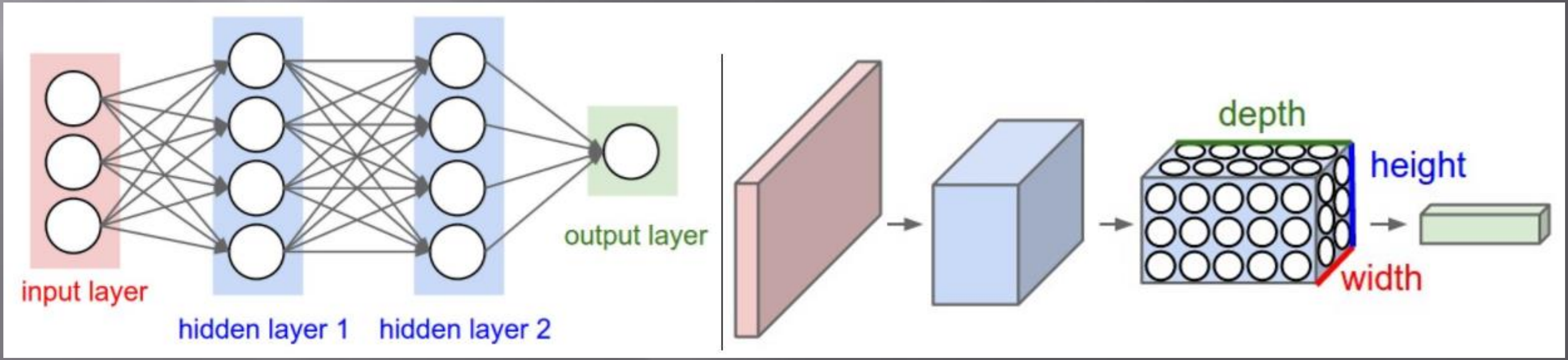
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Convolutional Neural Networks (CNN)

Convolutional Neural Networks arrange computational units (neurons) in 3D (**width, height, and depth**). The neurons in each layer are only connected to a small region of the previous layer instead of all-to-all (fully-connected) met in typical artificial neural networks.

Moreover, CNNs (e.g. CIFAR-10) reduces full images to a single output vector of class scores, arranged along the depth dimension as shown in the figure below.

The following figure presents the comparison of typical and a deep convolutional architecture:



Convolutional Neural Networks (CNN)

Convolutional Neural Networks consist of (sample):

1. Input image [32x32x3] where the third parameter codes colors from R, G, and B channels separately.
2. **Convolutional layer (CONV)** computes the output of neurons that are connected to local regions
   in the input image, each layer computes a **dot product** between their weights and a small region.
   This may result in volume such as [32x32x8] if we decide to use 8 convolutional filters.
3. **ReLU layer (RELU)** applies an elementwise activation function (such as the max(0,x) introduced before) thresholding at zero. This layer leaves the size of the volume unchanged [32x32x8].
4. **Pooling layer (POOL)** performs a downsampling operation along the spatial dimension (width x height), resulting in the volume such as [16x16x8]
5. **Fully connected layer** of a selected artificial neural network (FCNN) computes the **class scores** (classification), resulting in volume of size [1x1x5], where each individual output corresponds to one of
   5 classes (scores, categories). This layer is fully connected to all outputs of the previous layer and is trained using a gradient descent method.

Convolutional Layers Compute DotProduct

A **dot product** (called also a **scalar product**) is an algebraic operation that takes two equal-length sequences of numbers (usually vectors, however matrices can be used as well) and returns a single number that is computed as a sum of products of equivalent values from these two sequences (vectors or matrices):

Suppose, we have two vectors:

$$A = [a_1, a_2, \dots, a_n] \quad \text{and} \quad B = [b_1, b_2, \dots, b_n]$$

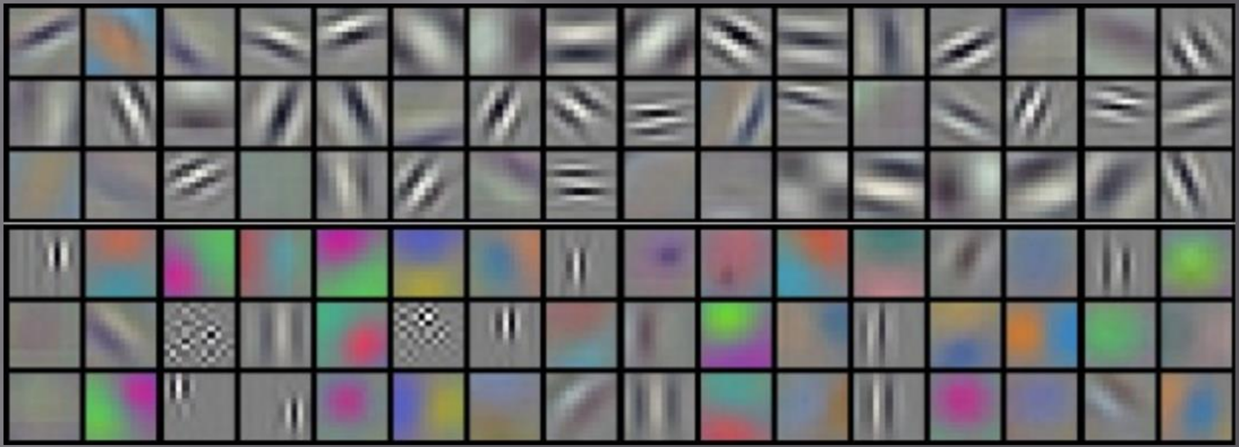
The dot product of these two vectors is defined as:

$$A \cdot B = \sum_{i=1}^{n} a_i \cdot b_i$$

Shared Weights and Biases

Each depth slice uses the same weights and bias for all neurons. In practice, every neuron in the volume will compute the gradient for its weights during backpropagation, but these gradients will be added up across each depth slice and only update a single set of weights per slice. Thus, all neurons in a single slice are using the same weight vector. The convolutional layer using this vector computes a convolution of the neuron's weights with the input volume. Because the same set of weights is used it can be treated as an adaptive filter convolving the input into the output scalar value.

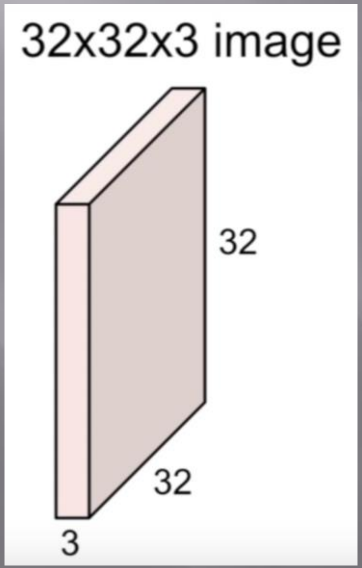
Example of 96 filters [11x11x3] learned by Krizhevsky at al. Each filter is shared by 55x55 neurons in one depth slice.
If detecting e.g. vertical line at some location in the image, it should be useful at some other location as well
due to the translationally-invariant structure of images.

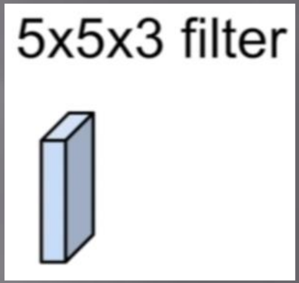


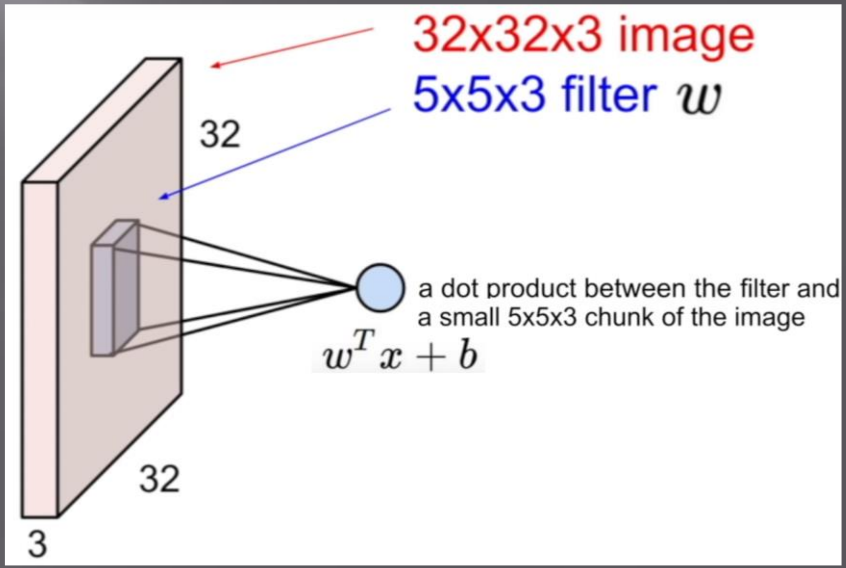
Therefore, **we do not need to relearn to detect a vertical line at every one** of the 55x55 distinct locations in the convolutional layer output volume.

Convolutional Layers

- ✓ Preserve a spatial structure of the image and its depth usually defined by the color components.
- ✓ Convolve the filter (weight matrix) with the image, sliding the filter over the image spatially computing dot products as a result of the convolution (we call it a feature map).
- ✓ Such filters extend the full depth (here 3) of the input volume.






32x32x3 image
5x5x3 filter w
32
a dot product between the filter and a small 5x5x3 chunk of the image
$w^T x + b$
32
3

Adaptive Filtering

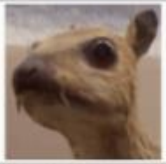
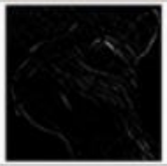
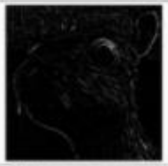
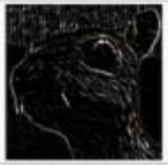
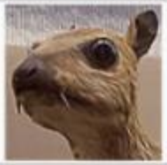
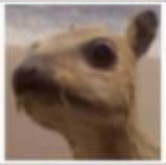
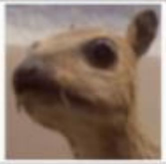
A convolutional layer works as an **adaptive filter** that allow to set values in such matrices:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

Using the other well-known filters we can convolve an input image as shown on the right.

We call the layer convolutional because it is related to
the convolution of two signals,
i.e. a filter and the signal:

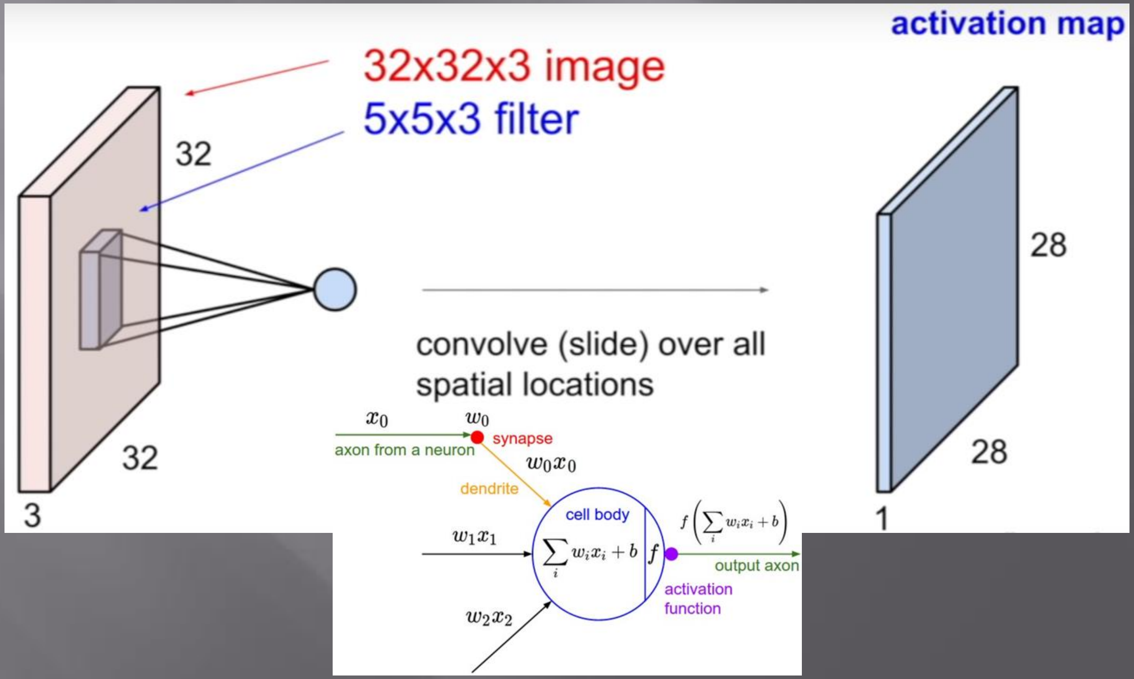
$$f[x,y] * g[x,y] \quad = \quad \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Convolution and Sliding Filters

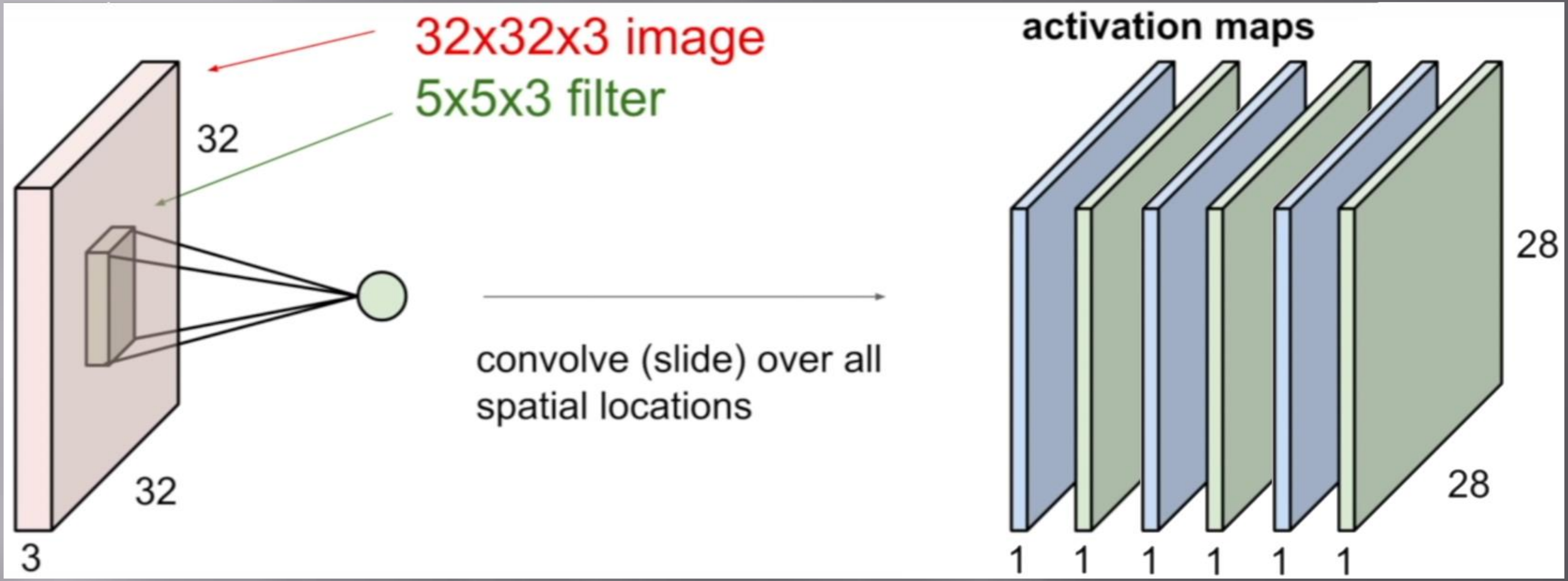
Sliding a filter over the image:

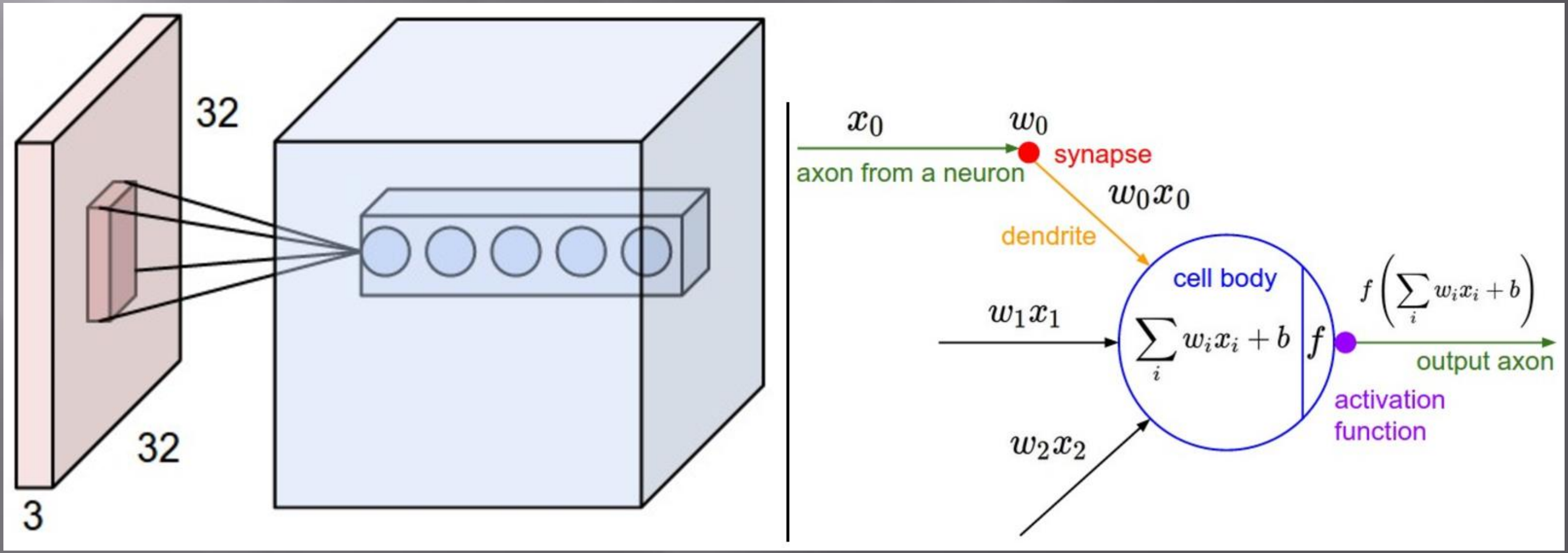
- ✓ When sliding the filter over the image, we always use the same filter for a given slice of neurons.
- ✓ The resultant matrix consisting of the dot products of the filter and the chunks of the image is called an activation map or a feature map.
- ✓ Its dimension can be smaller due to the size of the filter, used boarder and stride that control the way how we slide the filter over the image.



Many Adaptive Filters in Conv Layers

We use many filters in each convolutional layer represented by slices of neurons:





Example of Convolution

In this example, 3 separate tables are used to visualize 3 slice of the 3D input volume [5x5x3].

The input volume is in blue, the weight volumes are in red, and the output volume is in green.

In this convolutional layer we will use the following parameters:

K = 2 (number of filters),
F = 3 (filter size 3x3 in green),
S = 2 (stride),
P = 1 (padding), which makes the outer border of the input volume zero (in grey).

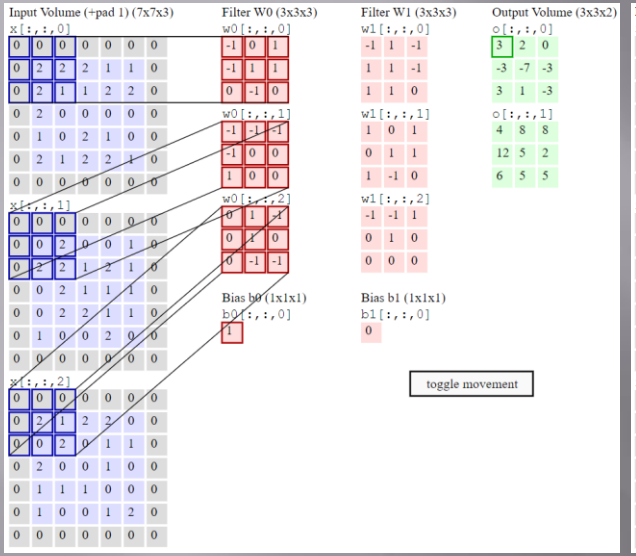
0	0	0	0	0	0	0
0	🟩	🟩				0
0	🟩	🟩				0
0						0
0						0
0						0
0	0	0	0	0	0	0

Hence, the output volume size is equal (5 - 3 + 2 ·1) / 2 + 1 = 3

The following visualization iterates over the green output activations, and shows that each element is computed by elementwise multiplying the highlighted blue input with the red filter, summing it up, and then offsetting the result by the bias.

How Does Convolution Work?

The weights are shared during the dot product computation:



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

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

x[:,:,1]

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

x[:,:,2]

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

Filter W0 (3x3x3)

w0[:,:,0]

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

w0[:,:,1]

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

w0[:,:,2]

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

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

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

w1[:,:,1]

1	0	1
0	1	1
1	-1	0

w1[:,:,2]

-1	-1	1
0	1	0
0	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

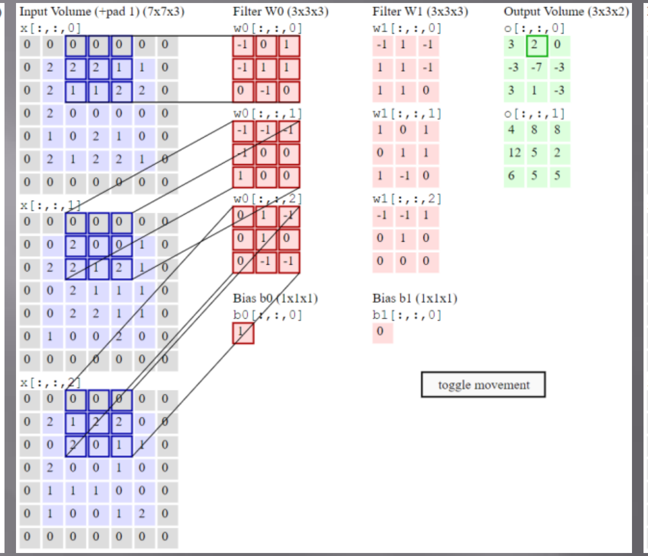
o[:,:,0]

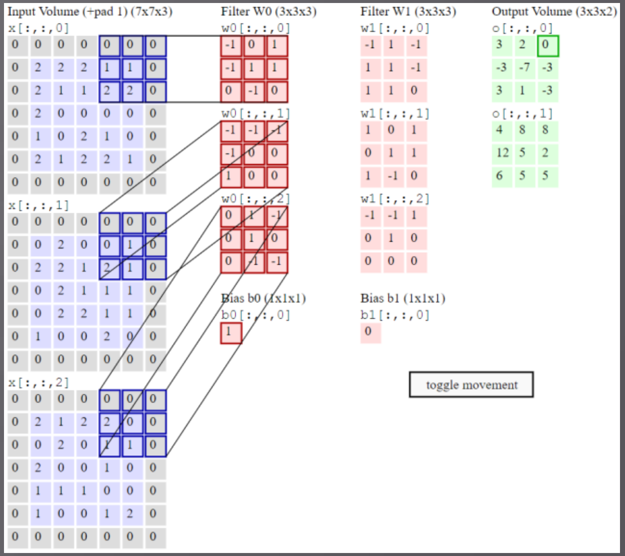
3	2	0
-3	-7	-3
3	1	-3

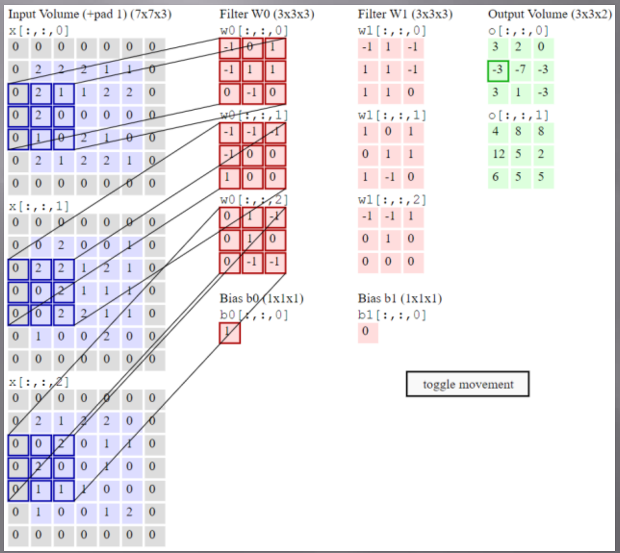
o[:,:,1]

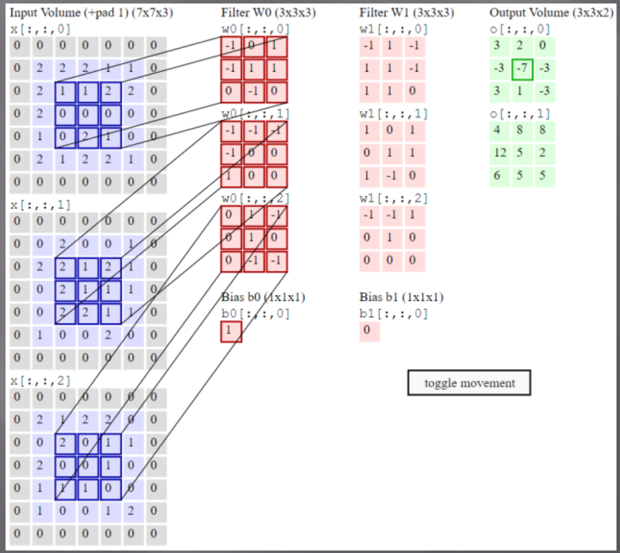
4	8	8
12	5	2
6	5	5

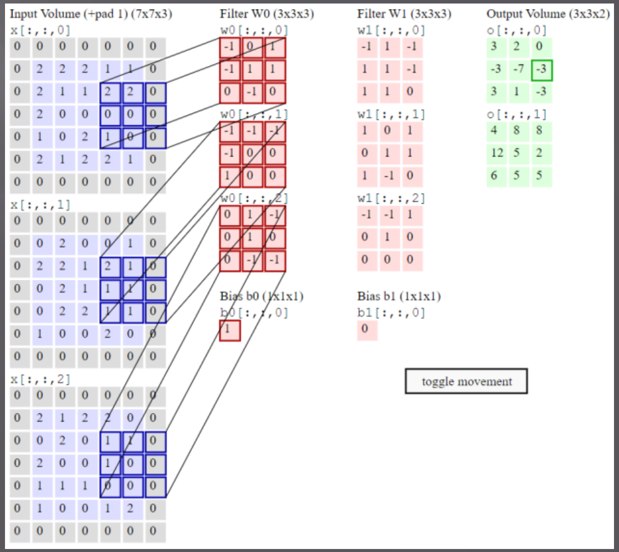
toggle movement





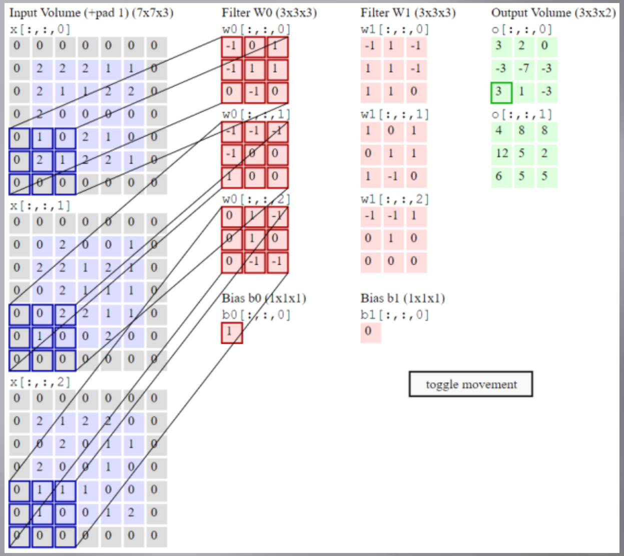
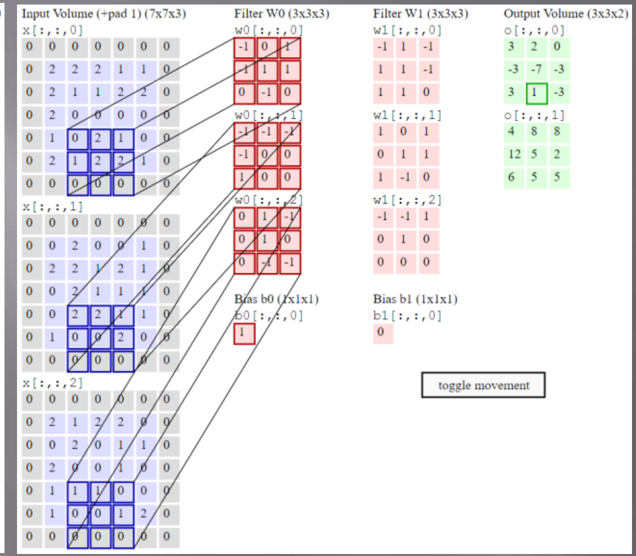
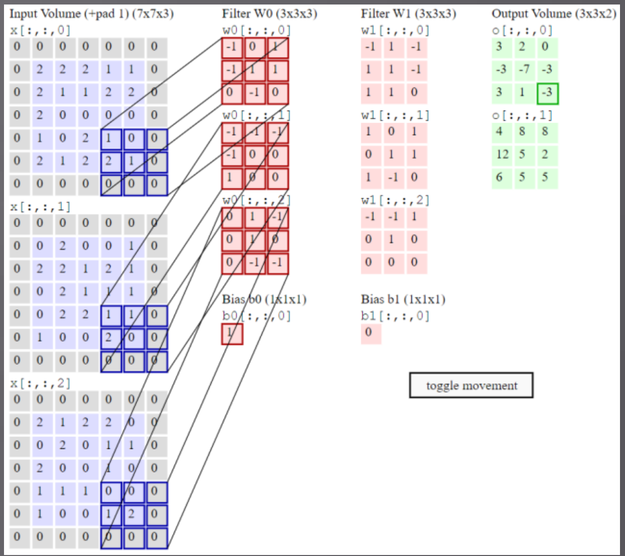
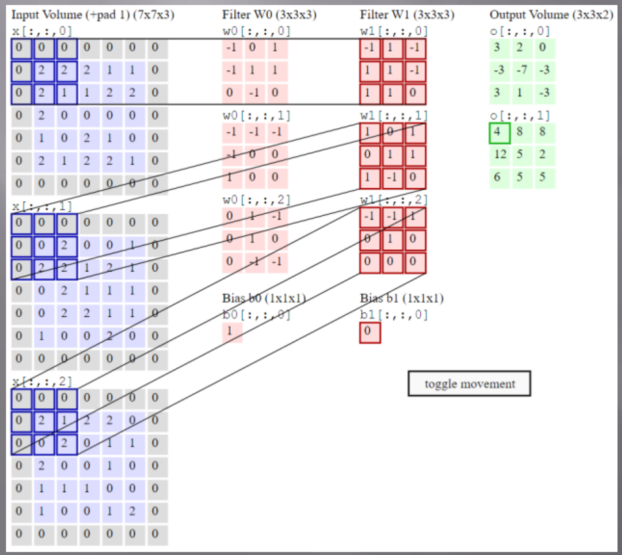
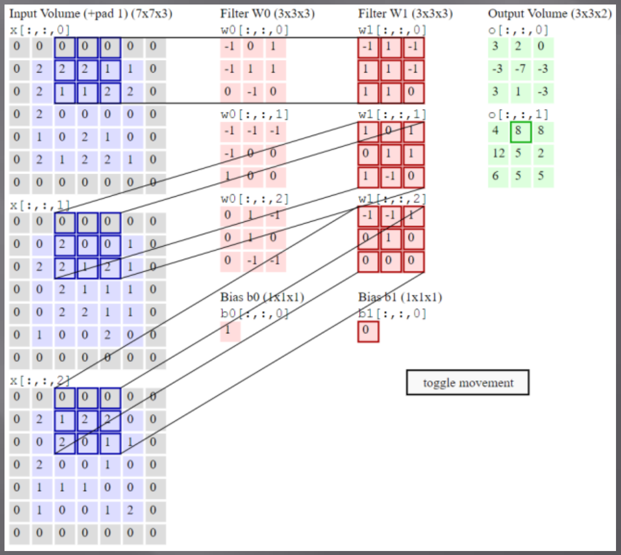
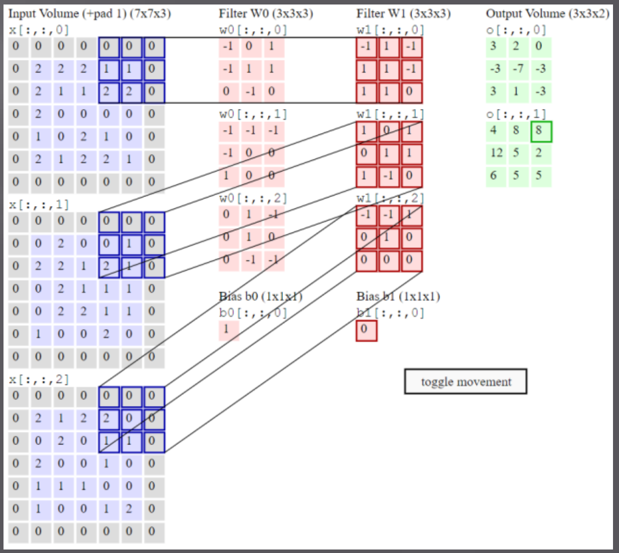






How Does Convolution Work?

The weights are shared during the dot product computation:

Input Volume (+pad 1) (7x7x3)

x[:,:,0]

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

x[:,:,1]

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

x[:,:,2]

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

Filter W0 (3x3x3)

w0[:,:,0]

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

w0[:,:,1]

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

w0[:,:,2]

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

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

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

w1[:,:,1]

1	0	1
0	1	1
1	-1	0

w1[:,:,2]

-1	-1	1
0	1	0
0	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

3	2	0
-3	-7	-3
3	1	-3

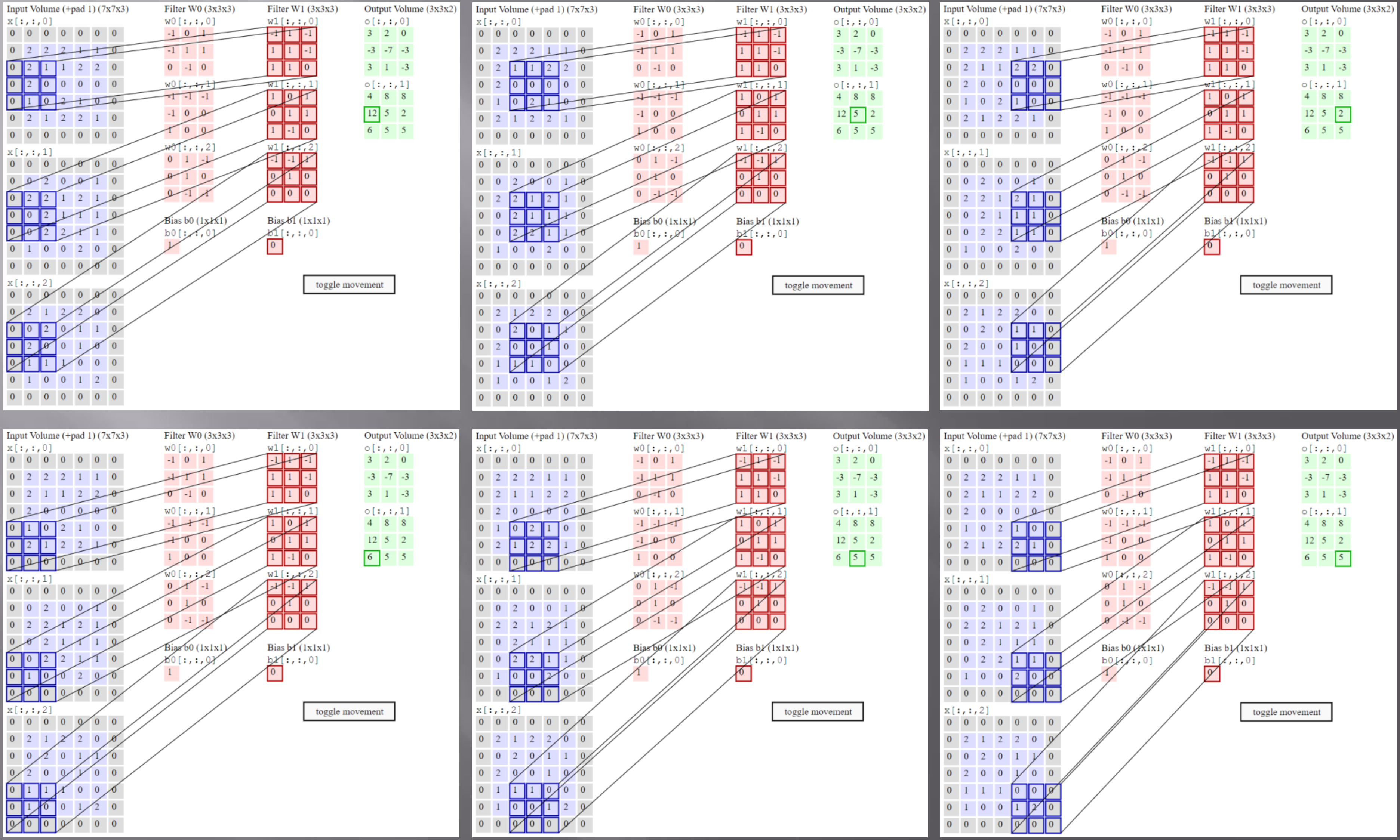
o[:,:,1]

4	8	8
12	5	2
6	5	5

toggle movement

How Does Convolution Work?

The weights are shared during the dot product computation:



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

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

x[:,:,1]

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

x[:,:,2]

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

Filter W0 (3x3x3)

w0[:,:,0]

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

w0[:,:,1]

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

w0[:,:,2]

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

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

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

w1[:,:,1]

1	0	1
0	1	1
1	-1	0

w1[:,:,2]

-1	-1	1
0	1	0
0	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

3	2	0
-3	-7	-3
3	1	-3

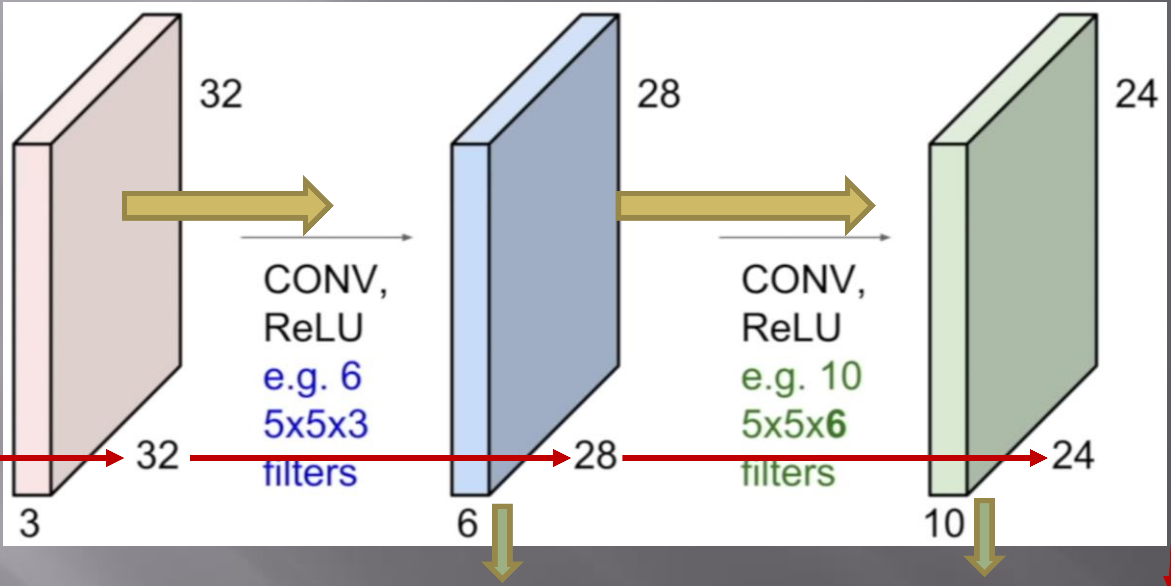
o[:,:,1]

4	8	8
12	5	2
6	5	5

toggle movement

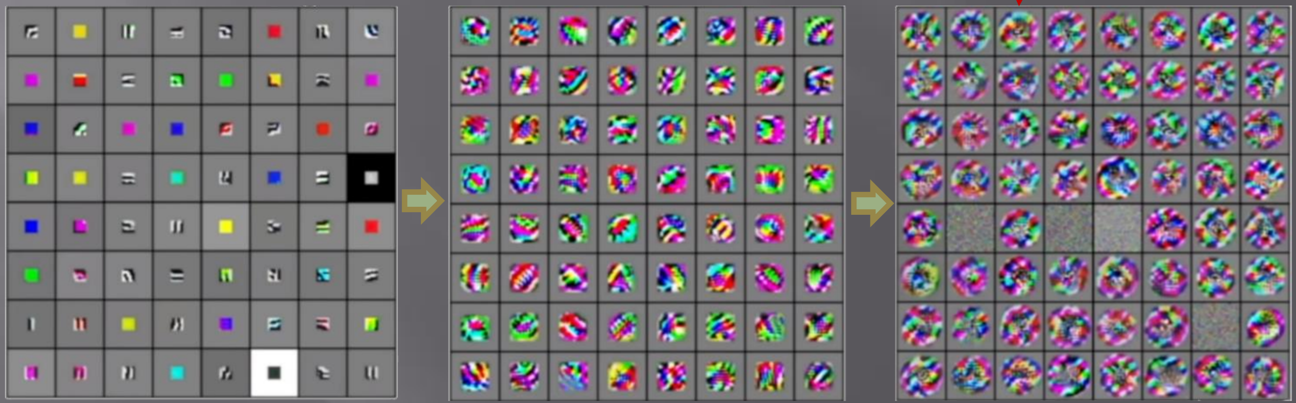
Convolutional Network Construction

The popular ConvNets are constructed as a sequence of many convolutional layers that represent still more abstract features starting from low-level (primary, simpliest) features, through mid-level (secondary) features, to high-level (more complex) features which are finally used by dense layers (softmax) for classification.



Each neuron shows the average picture generated from all the same chunks of different training images to which it reacts the strongest (wins the competition).

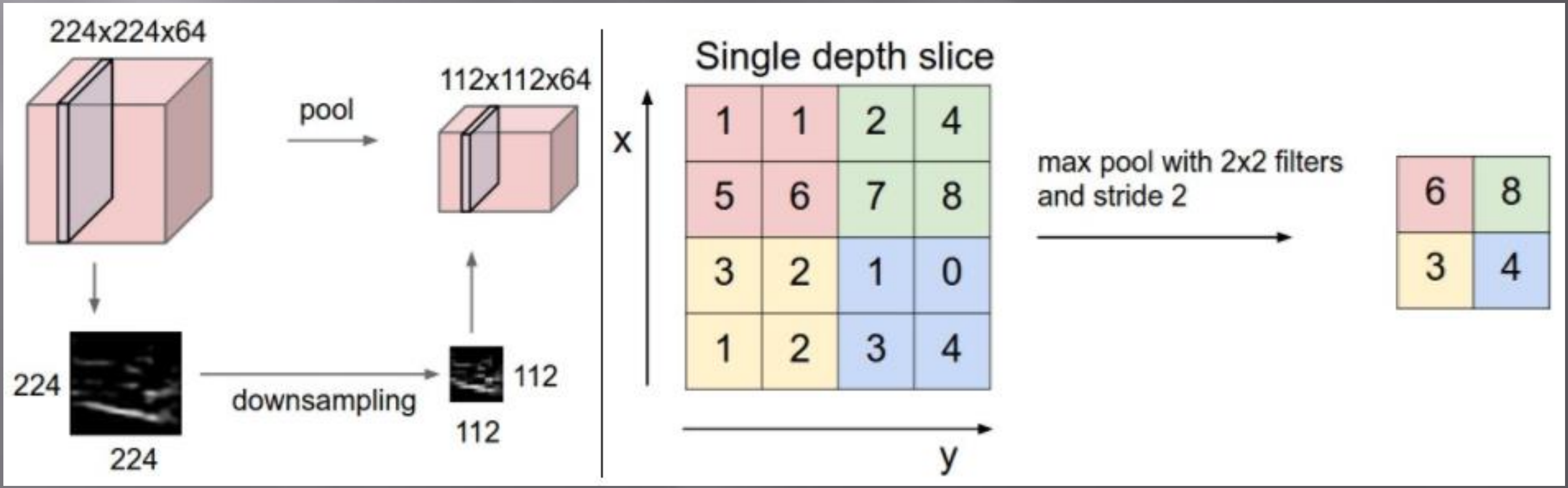
Be careful about shrinking the filter sizes too fast because it does not work well!



Max Pooling

The pooling layer typically uses **MAX operation** independently on every depth slice of the input and **resizes it spatially**.

The most common form of pooling is to use filters of size 2x2 applied with the stride 2, downsampling every depth slice in the input by 2 along both width and height, discarding 75% of the activations, because we always choose 1 maximum activation from four activations in the region 2x2 in each depth slice. The depth is always preserved.

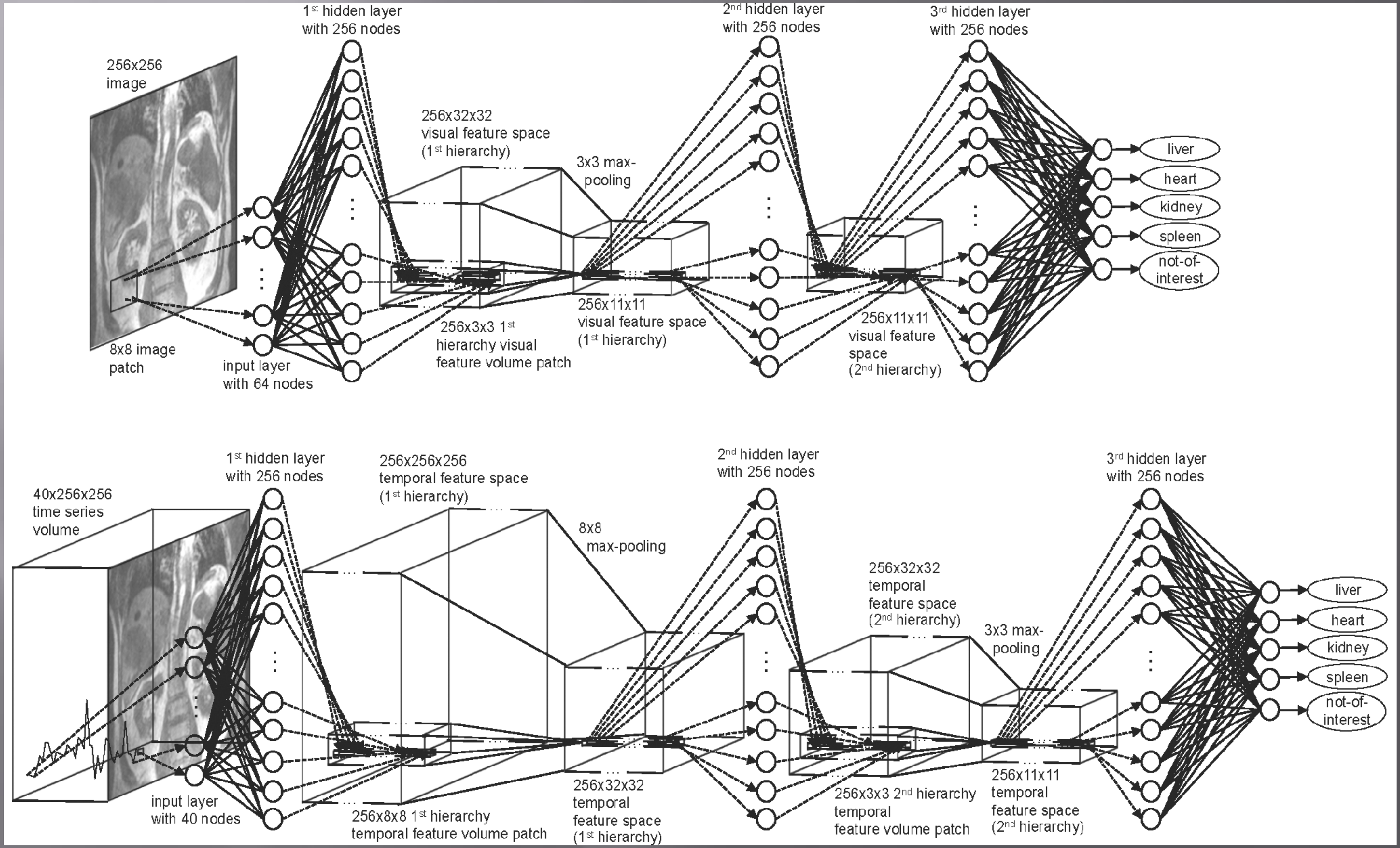


1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

CNN to Biomedical Image Classification

Example of the recognition of human organs on RT images:



256x256 image
8x8 image patch
input layer with 64 nodes
1st hidden layer with 256 nodes
256x32x32 visual feature space (1st hierarchy)
256x3x3 1st hierarchy visual feature volume patch
3x3 max-pooling
256x11x11 visual feature space (1st hierarchy)
2nd hidden layer with 256 nodes
256x11x11 visual feature space (2nd hierarchy)
3rd hidden layer with 256 nodes
liver
heart
kidney
spleen
not-of-interest
40x256x256 time series volume
input layer with 40 nodes
1st hidden layer with 256 nodes
256x256x256 temporal feature space (1st hierarchy)
256x8x8 1st hierarchy temporal feature volume patch
8x8 max-pooling
256x32x32 temporal feature space (1st hierarchy)
2nd hidden layer with 256 nodes
256x32x32 temporal feature space (2nd hierarchy)
256x3x3 2nd hierarchy temporal feature volume patch
3x3 max-pooling
256x11x11 temporal feature space (2nd hierarchy)
3rd hidden layer with 256 nodes
liver
heart
kidney
spleen
not-of-interest

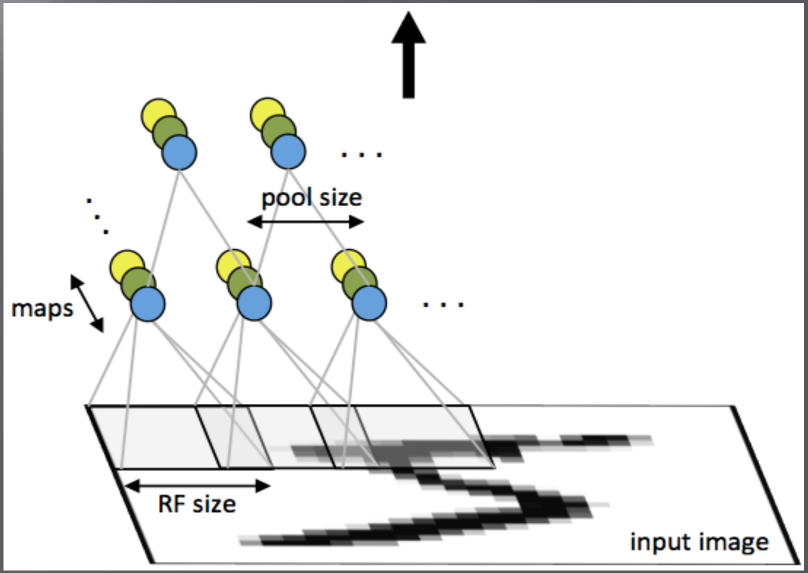
CNN to Biomedical Image Classification

A Convolutional Neural Network (CNN) comprises of one or more convolutional layers (typically with a subsampling step) and then followed by one or more fully connected layers
as in a standard multilayer neural network (e.g. MLPs), SVM, SoftMax etc.

A Deep CNN consists of more layers. The CNNs are easier to train and have many fewer parameters (using the same weights) than typical neural networks with regards to the number of convolutional layers and their size.

This kind of networks are naturally suited to perform computations on 2D structures (images).

In the figure, the first layer of a convolutional neural network with pooling. Units of the same color have tied weights and units of different color represent different filter maps:



http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

Notebooks and Frameworks

Popular Notebooks:

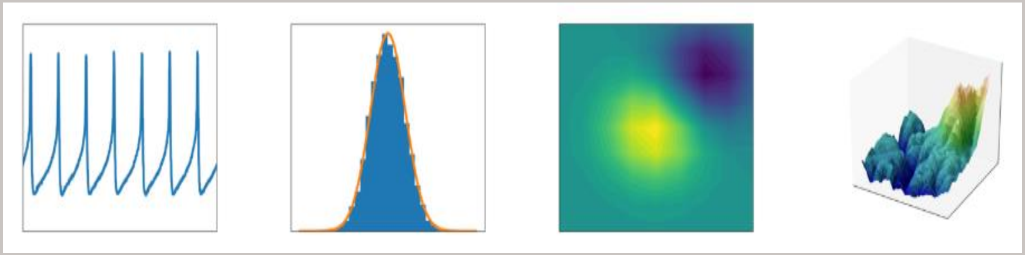
Jupyter Notebook Google Colab

Popular frameworks:

- Tensorflow 2.0
- Keras

Popular libraries:

- numpy
- scikit-learn
- pandas
- mxnet
- matplotlib



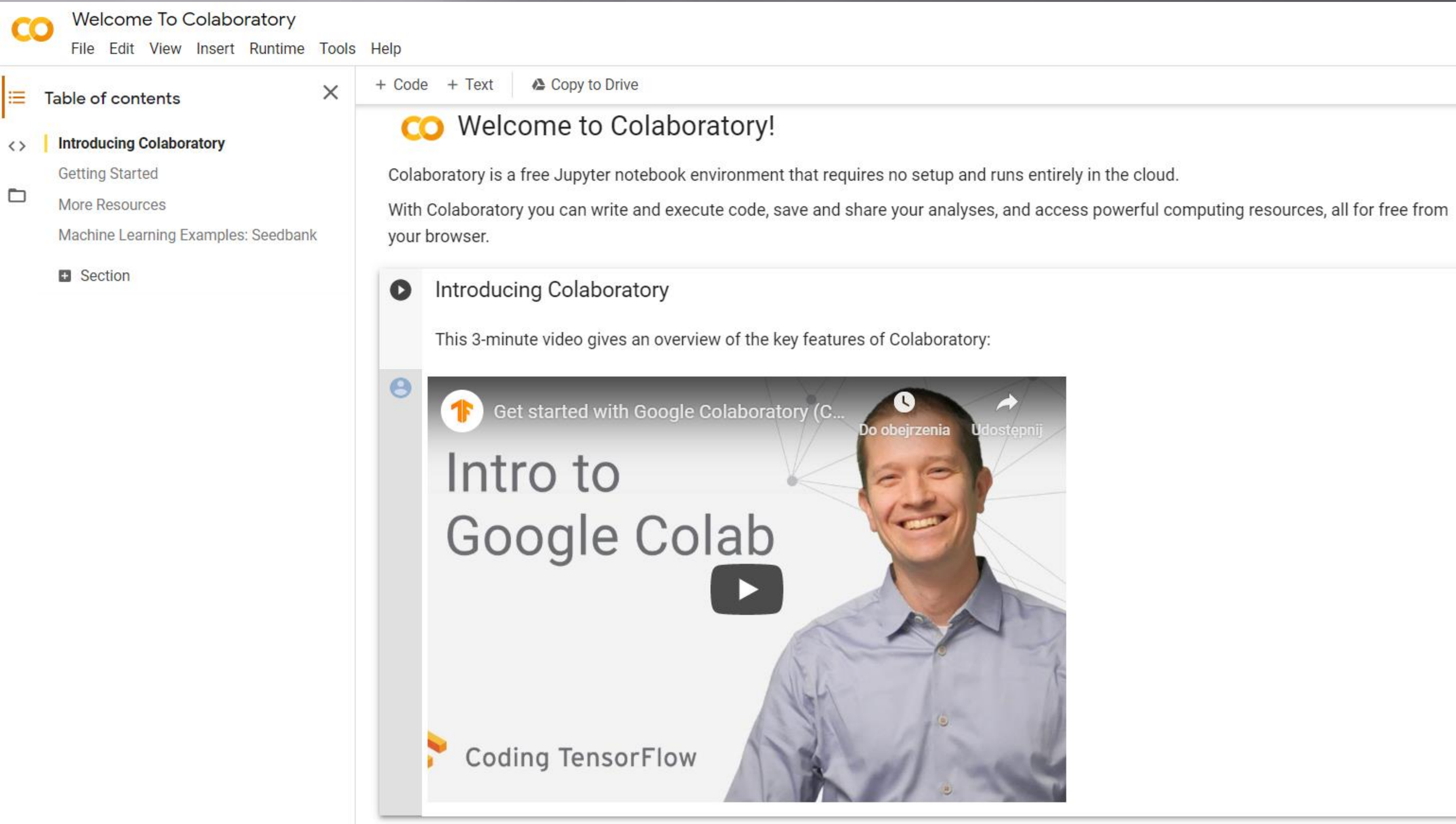
Jupyter Notebook

Jupyter is open, free and very popular:

Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

Google Colaboratory

Google Colab is an alternative notebook supported by Google using a Google cloud where the computation can be executed (< 8 hours for free):



Welcome to Colaboratory!

Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud.

With Colaboratory you can write and execute code, save and share your analyses, and access powerful computing resources, all for free from your browser.

Introducing Colaboratory

This 3-minute video gives an overview of the key features of Colaboratory:

WHAT IS KERAS?

Keras developed by François Chollet:

- Is an official high-level and high-performing API of TensorFlow used to specify and train different programs.
- Runs on top of TensorFlow, Theano, MXNet, or CNTK.
- Builds models by stacking layers and connecting graphs.
- Is actively developed by thousands of contributors across the world, e.g. Microsoft, Google, Nvidia, AWS.
- Is used by hundred thousands of developers, e.g. NetFlix, Uber, Google, Huawei, NVidia.
- Has a good amount of documentation and easy to grasp all concepts.
- Supports GPU both of Nvidia and AMD and runs seamlessly on CPU and GPU.
- Is multi-platform (Python, R) and multi-backend.
- Allows for fast prototyping and leaves freedom to design and architecture

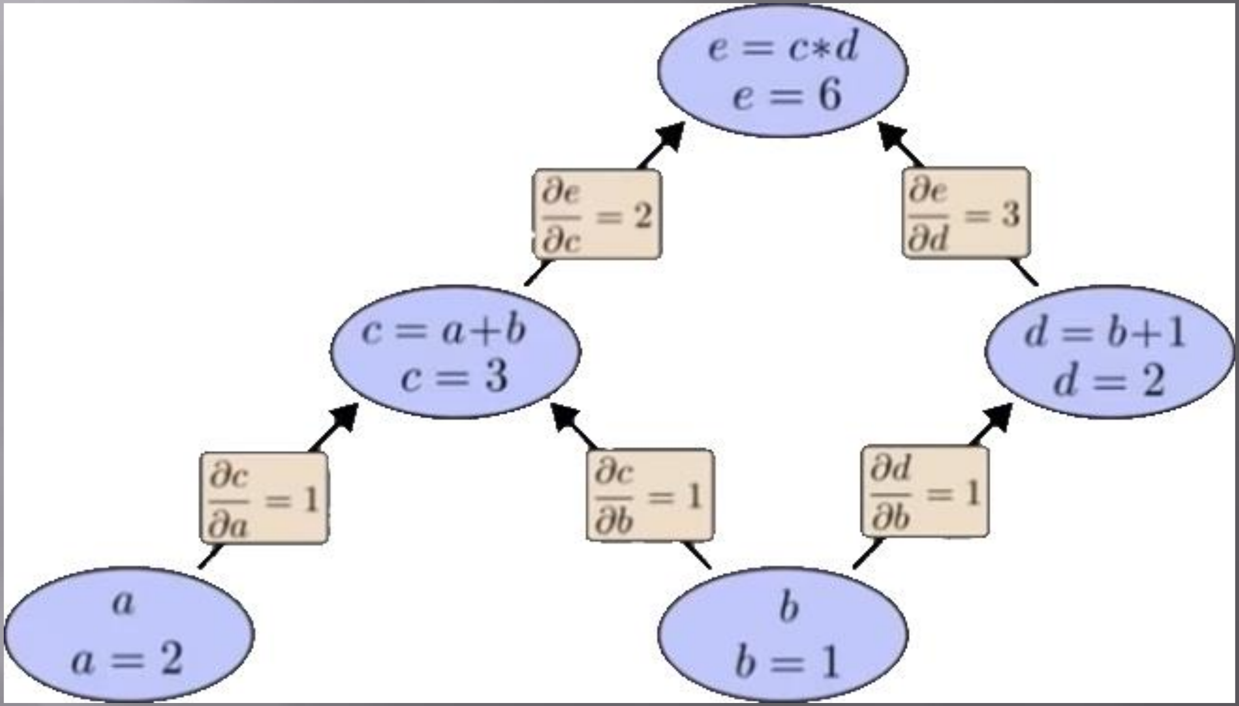
KERAS POSITIVE USER EXPERIENCE

Keras:

- **Follows best practices for reducing cognitive load**
- **Offers consistent and simple APIs.**
- **Minimizes the number of user actions required for common use cases.**
- **Provides clear feedback upon user errors.**
- **More productive than many other frameworks.**
- **Integrates with lower-level Deep Learning languages like TensorFlow or Theano.**
- **Implements everything which was built-in the base language, i.e. TensorFlow.**
- **Produces models using GPU acceleration for various systems like Windows, Linux, Android, iOS, Raspberry Pi.**

WORKING PRINCIPLE

Keras is based on Computational Graphs like:



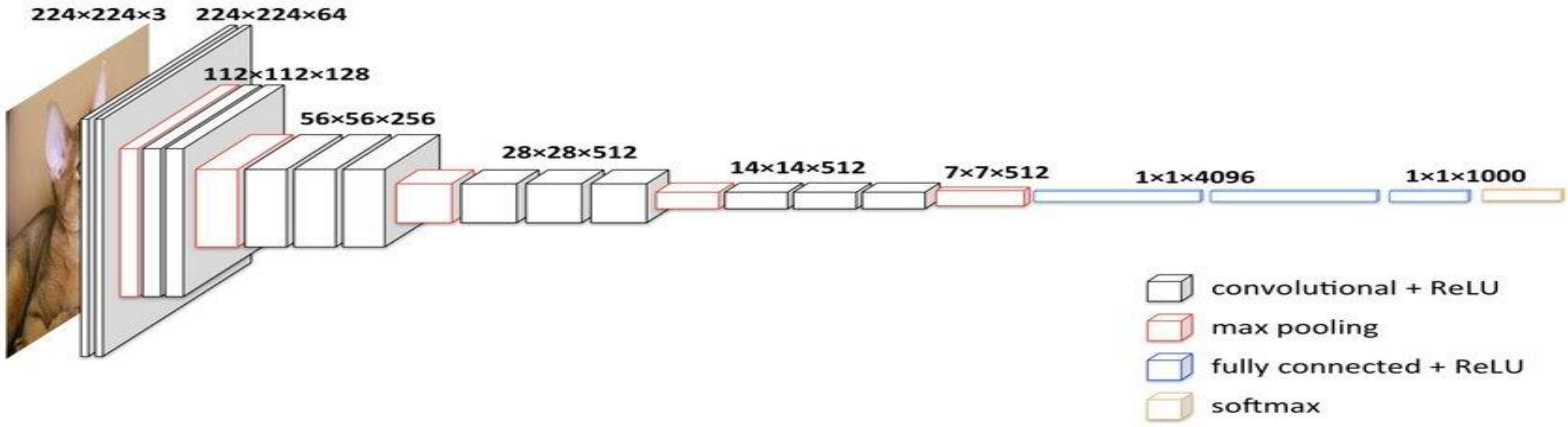
$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

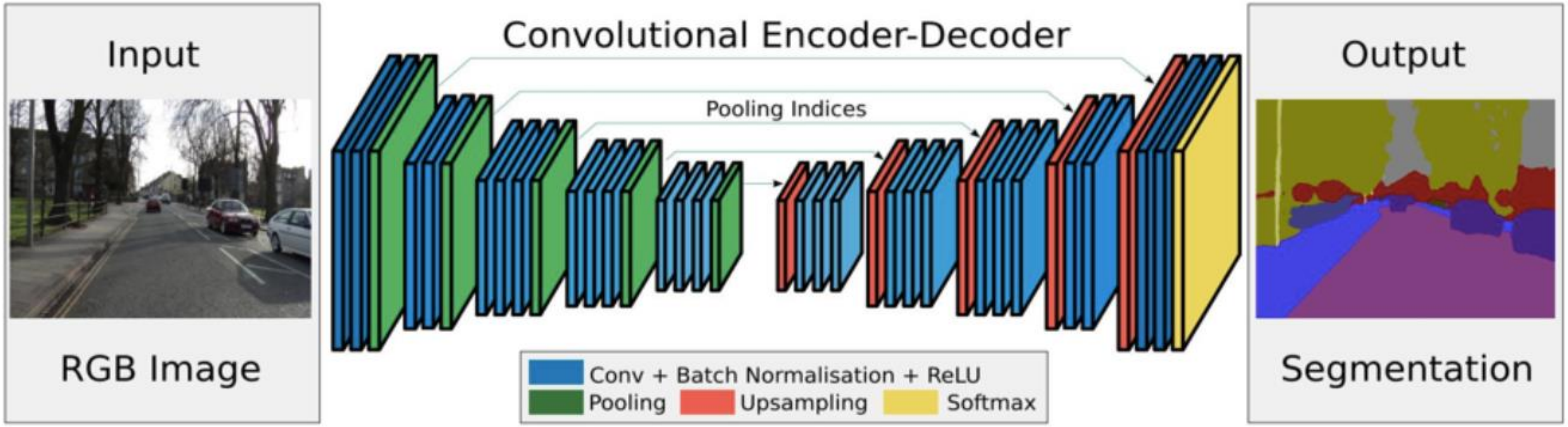
Where “a” and “b” are inputs used to compute “e” as an output using intermediate variables “c” and “d”.

Computational Graphs allow expressing complex expressions as a combination of simple operations.

KERAS SEQUENTIAL MODELS

We can create various sequential models which linearly stack layers and can be used for classification networks or autoencoders (consisting of encoders and decoders) like:

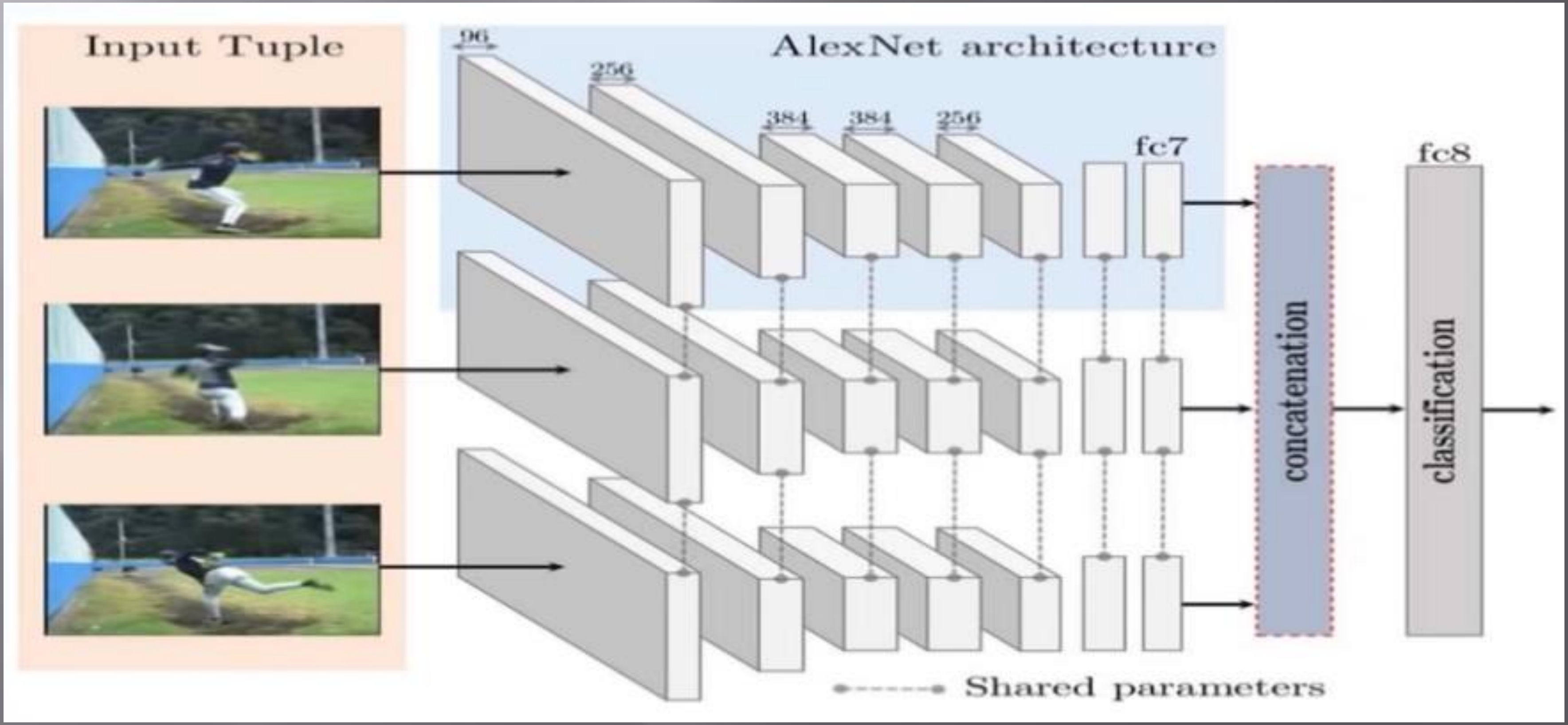




KERAS FUNCTIONAL MODELS

Keras models can:

- **Use multi-input, multi-output and arbitrary static graph topologies,**
- **Branch into two or more submodels,**
- **Share layers and/or weights.**



TWO TYPES OF EXECUTION OF KERAS MODELS

We can execute Keras model in two ways:

1. **Deferred (symbolic)**
   - Using Python to build a computational graph, next compiling and executing it.
   - Symbolic tensors **don't have a value** in the Python code.
2. **Eager (imperative)**
   - Here the Python runtime is the execution runtime, which is similar to the execution with Numpy.
   - Eager tensors **have a value** in the Python code.
   - With the eager execution, **value-dependent dynamic topologies** (tree-RNNs) can be constructed and used.

5 STEPS TO IMPLEMENT A NEURAL NETWORK

1. **Prepare Input (e.g. text, audio, images, video)** and specify the input dimension (size).
2. **Define the Model:** its architecture, build the computational graph, define the sequential or functional style of the model and the kind of the network (MLP, CNN, RNN etc.).
3. **Specify the Optimizers** (Stochastic Gradient Descent (SGD), Root Mean Square (RMSprop), Adam etc.) to configure the learning process.
4. **Define the Loss Function (e.g. Mean Square Error (MSE), Cross Entropy, Hinge)** for checking the accuracy of the achieved prediction to adapt and improve the model.
5. **Train using training data, Test using testing/validation data, and Evaluate the Model.**

INSTALLING TENSORFLOW AND KERAS

To start working with TensorFlow and Keras in Jupyter Notebook, you have to install them using the following commands in the Anaconda Prompt window:

conda install pip   # install pip in the virtual environment

pip install --upgrade tensorflow   # for python 2.7

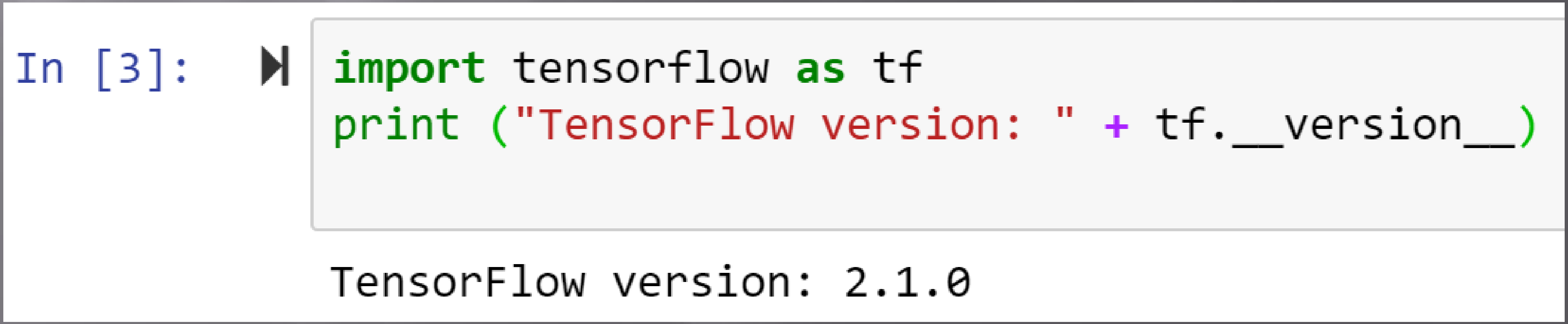
pip3 install --upgrade tensorflow   # for python 3.*

It is recommended to install tensorflow with parameter –gpu to use GPU unit and make computations faster:

pip install tensorflow-gpu

$ pip install Keras

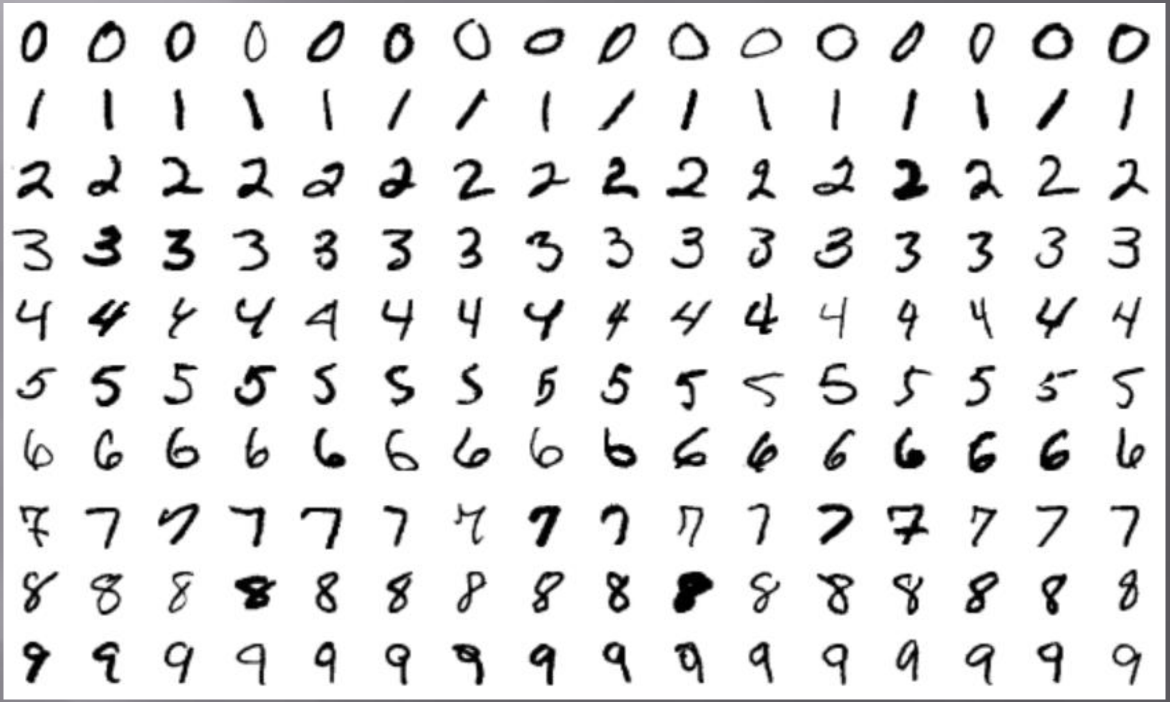
If successfully installed check in Jupyter Notebook the version of the TensorFlow using:



```
In [3]:  import tensorflow as tf
          print ("TensorFlow version: " + tf.__version__)

          TensorFlow version: 2.1.0
```

IMPLEMENTING A CNN USING KERAS

We will try to create and train a simple Convolutional Neural Network (CNN) to tackle with handwritten digit classification problem using MNIST dataset:



Each image in the MNIST dataset is 28x28 pixels and contains a centred, grayscale digit form 0 to 9. Our goal is to classify these images to one of the ten classes using ten output neurons of the CNN network.

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

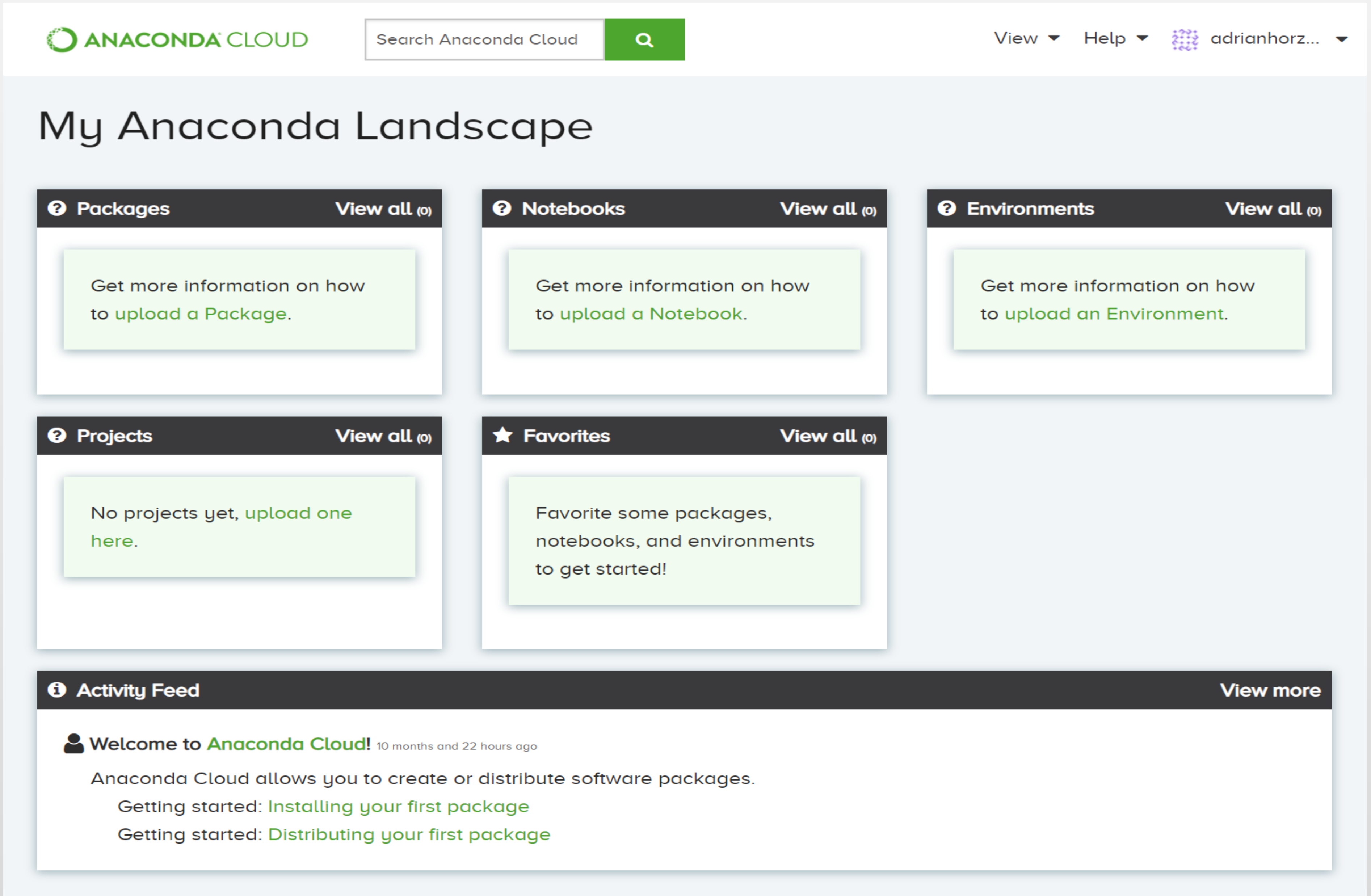
Install Jupyter using Anaconda with built in Python 3.7+

- 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, Keras, 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



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

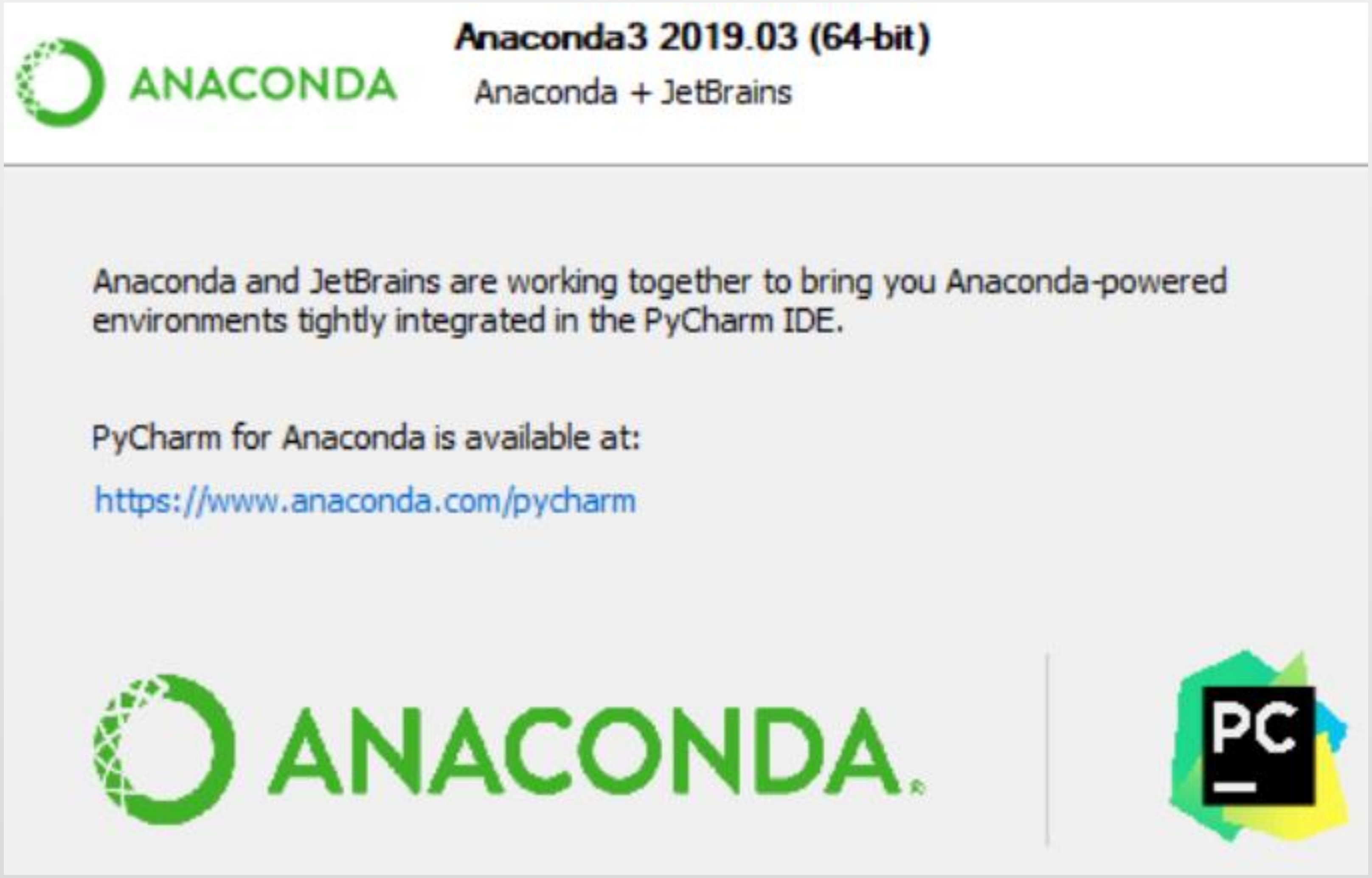
Anaconda Cloud allows you to create or distribute software packages.

Getting started: Installing your first package

Getting started: Distributing your first package

Jupyter Notebook & PyCharm

It is recommended to install PyCharm for Anaconda:



Anaconda3 2019.03 (64-bit)

Anaconda + JetBrains

Anaconda and JetBrains are working together to bring you Anaconda-powered environments tightly integrated in the PyCharm IDE.

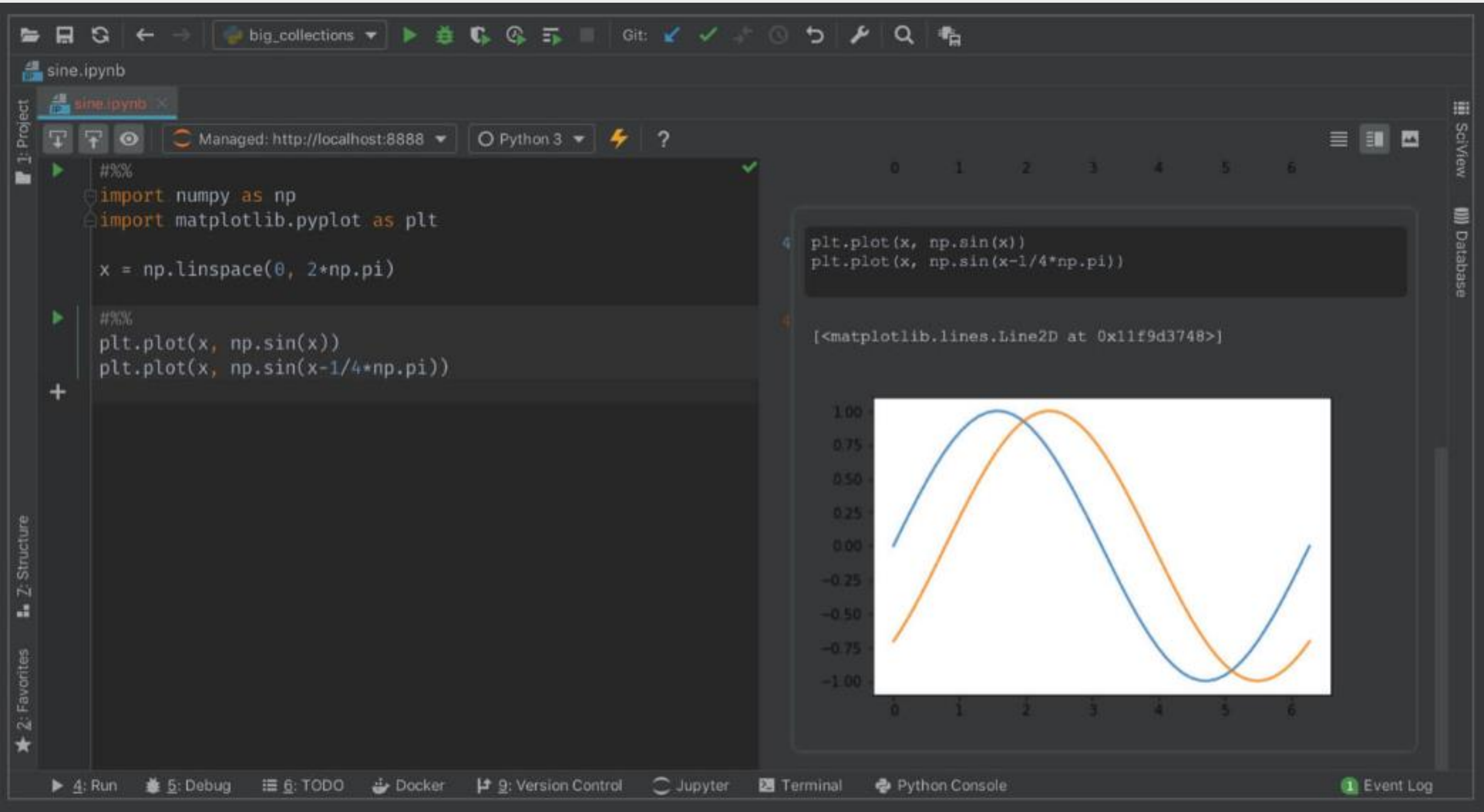
PyCharm for Anaconda is available at:

https://www.anaconda.com/pycharm

Jupyter Notebook

PyCharm is a python IDE for Professional Developers

- It includes scientific mode to interactively analyze your data.



```
#%%
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi)
```

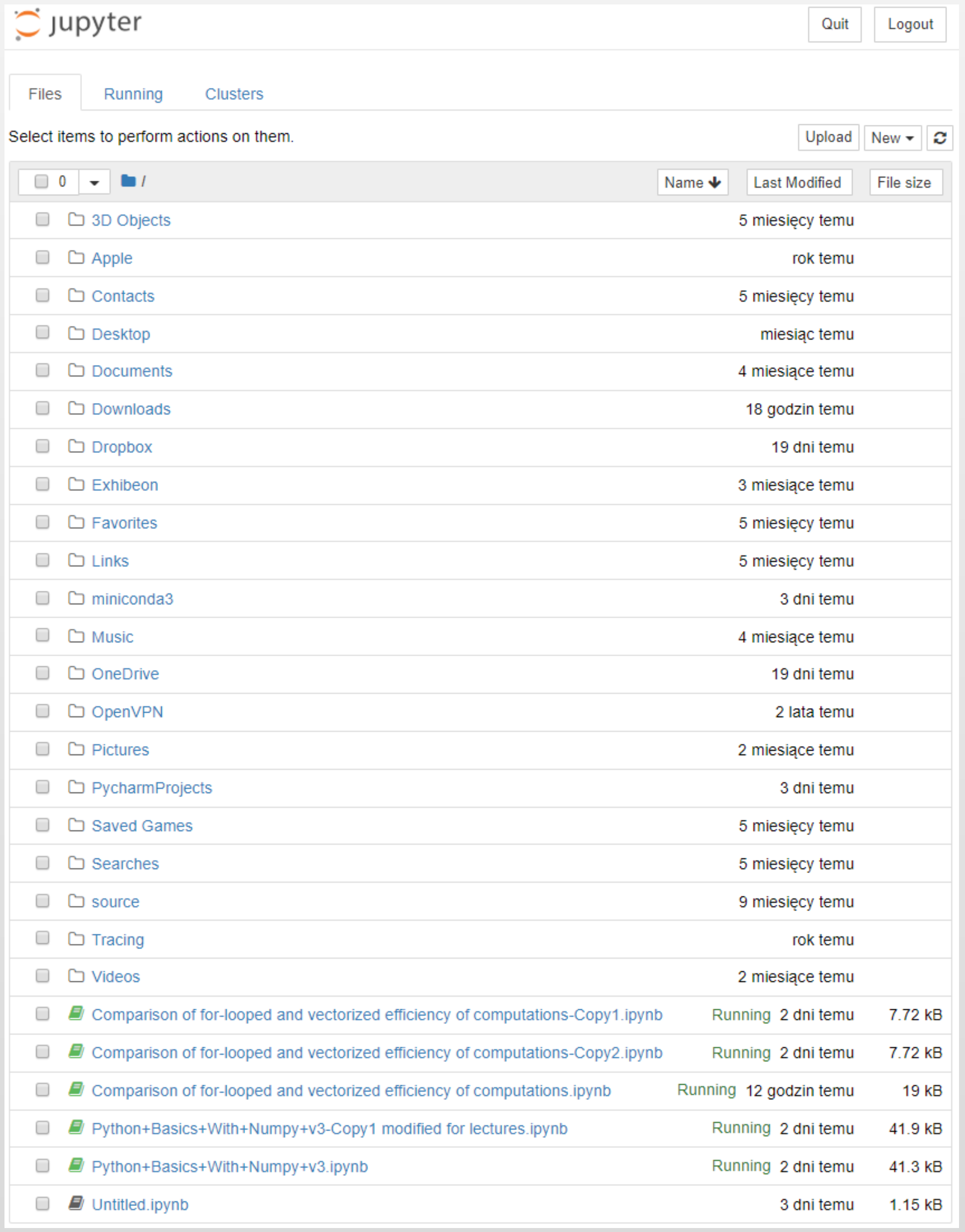
```
#%%
plt.plot(x, np.sin(x))
plt.plot(x, np.sin(x-1/4*np.pi))
```

```
[<matplotlib.lines.Line2D at 0x11f9d3748>]
```

Jupyter Notebook Dashboard

Running a Jupyter Notebook in your browser:

- When the Jupyter Notebook opens in your browser, you will see the Jupyter 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.



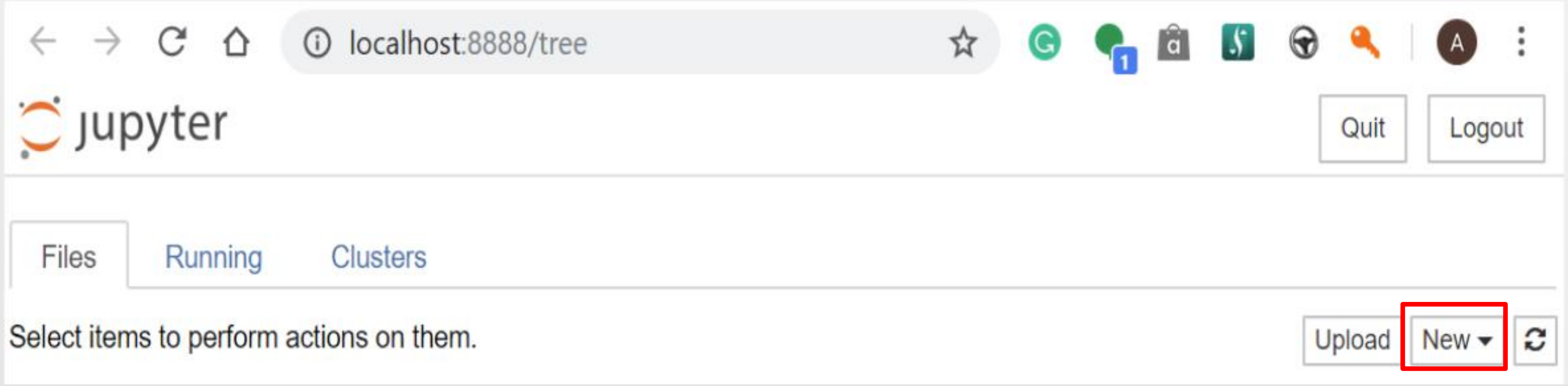
Select items to perform actions on them.

0 /		Name	Last Modified	File size
3D Objects			5 miesięcy temu	
Apple			rok temu	
Contacts			5 miesięcy temu	
Desktop			miesiąc temu	
Documents			4 miesiące temu	
Downloads			18 godzin temu	
Dropbox			19 dni temu	
Exhibeon			3 miesiące temu	
Favorites			5 miesięcy temu	
Links			5 miesięcy temu	
miniconda3			3 dni temu	
Music			4 miesiące temu	
OneDrive			19 dni temu	
OpenVPN			2 lata temu	
Pictures			2 miesiące temu	
PycharmProjects			3 dni temu	
Saved Games			5 miesięcy temu	
Searches			5 miesięcy temu	
source			9 miesięcy temu	
Tracing			rok temu	
Videos			2 miesiące temu	
Comparison of for-looped and vectorized efficiency of computations-Copy1.ipynb	Running		2 dni temu	7.72 kB
Comparison of for-looped and vectorized efficiency of computations-Copy2.ipynb	Running		2 dni temu	7.72 kB
Comparison of for-looped and vectorized efficiency of computations.ipynb	Running		12 godzin temu	19 kB
Python+Basics+With+Numpy+v3-Copy1 modified for lectures.ipynb	Running		2 dni temu	41.9 kB
Python+Basics+With+Numpy+v3.ipynb	Running		2 dni temu	41.3 kB
Untitled.ipynb			3 dni temu	1.15 kB

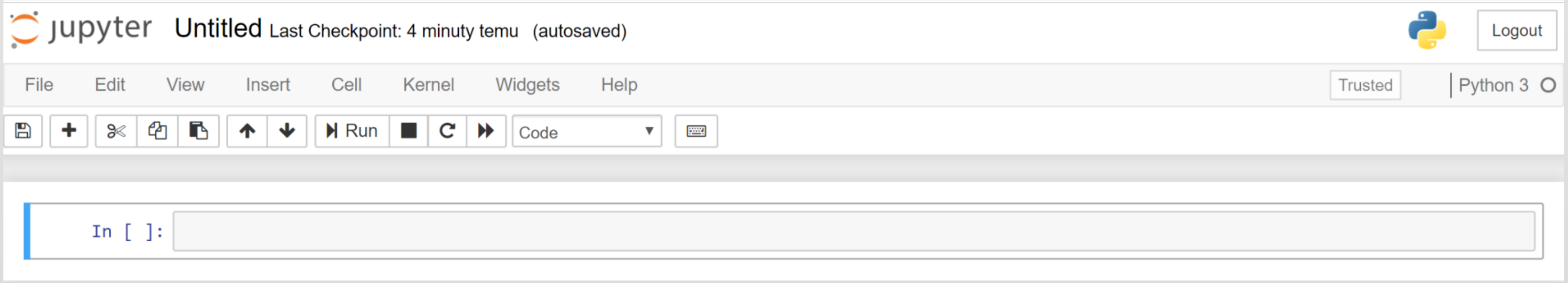
Starting a new Python notebook

Start a new Python notebook:

- **Clicking New → Python 3**



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

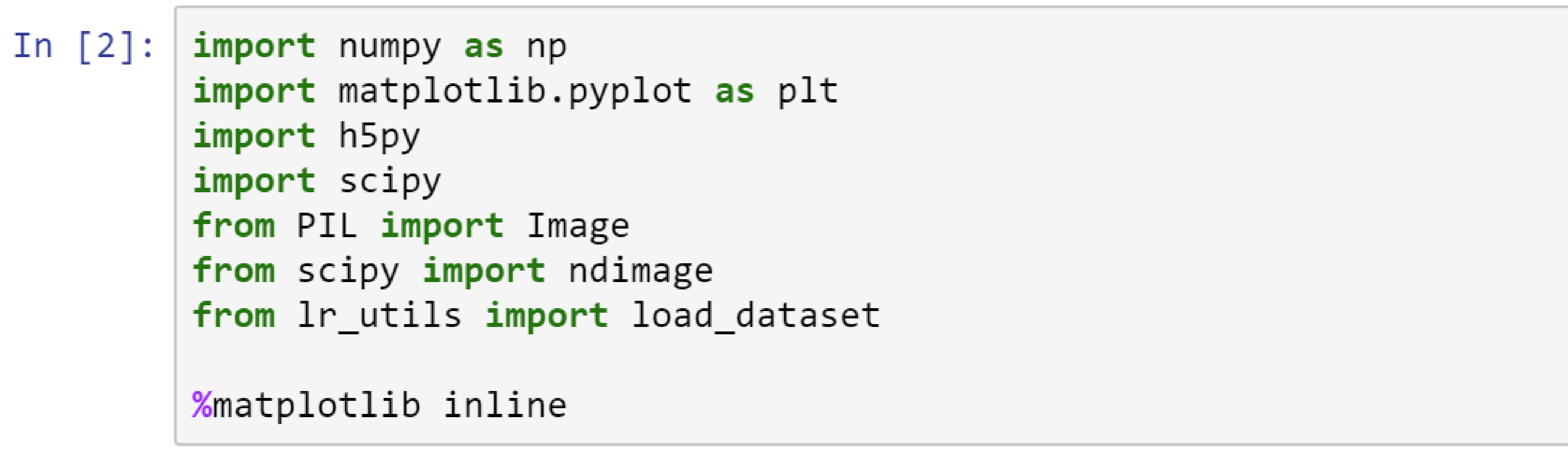


Useful Packages and Libraries

In the next assignments and examples, we well use the following packages:

- **numpy is the fundamental package for scientific computing with Python.**
- **h5py is a common package to interact with a dataset that is stored on an H5 file.**
- **matplotlib is a famous library to plot graphs in Python.**
- **PIL and scipy are used here to test your model with your own picture at the end.**

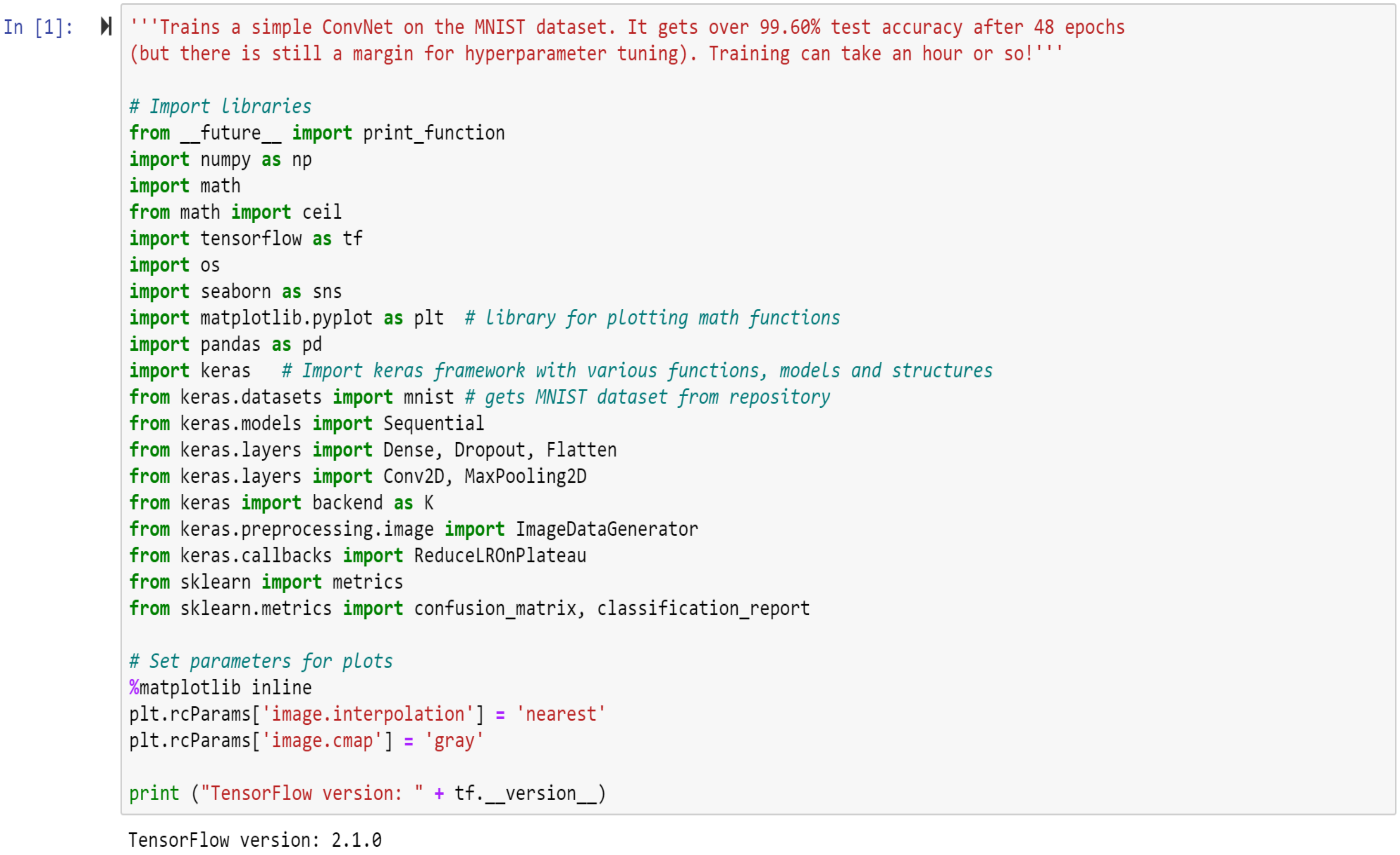
They must be imported:



```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import h5py
        import scipy
        from PIL import Image
        from scipy import ndimage
        from lr_utils import load_dataset

        %matplotlib inline
```

MNIST Classification in Jupyter Notebook

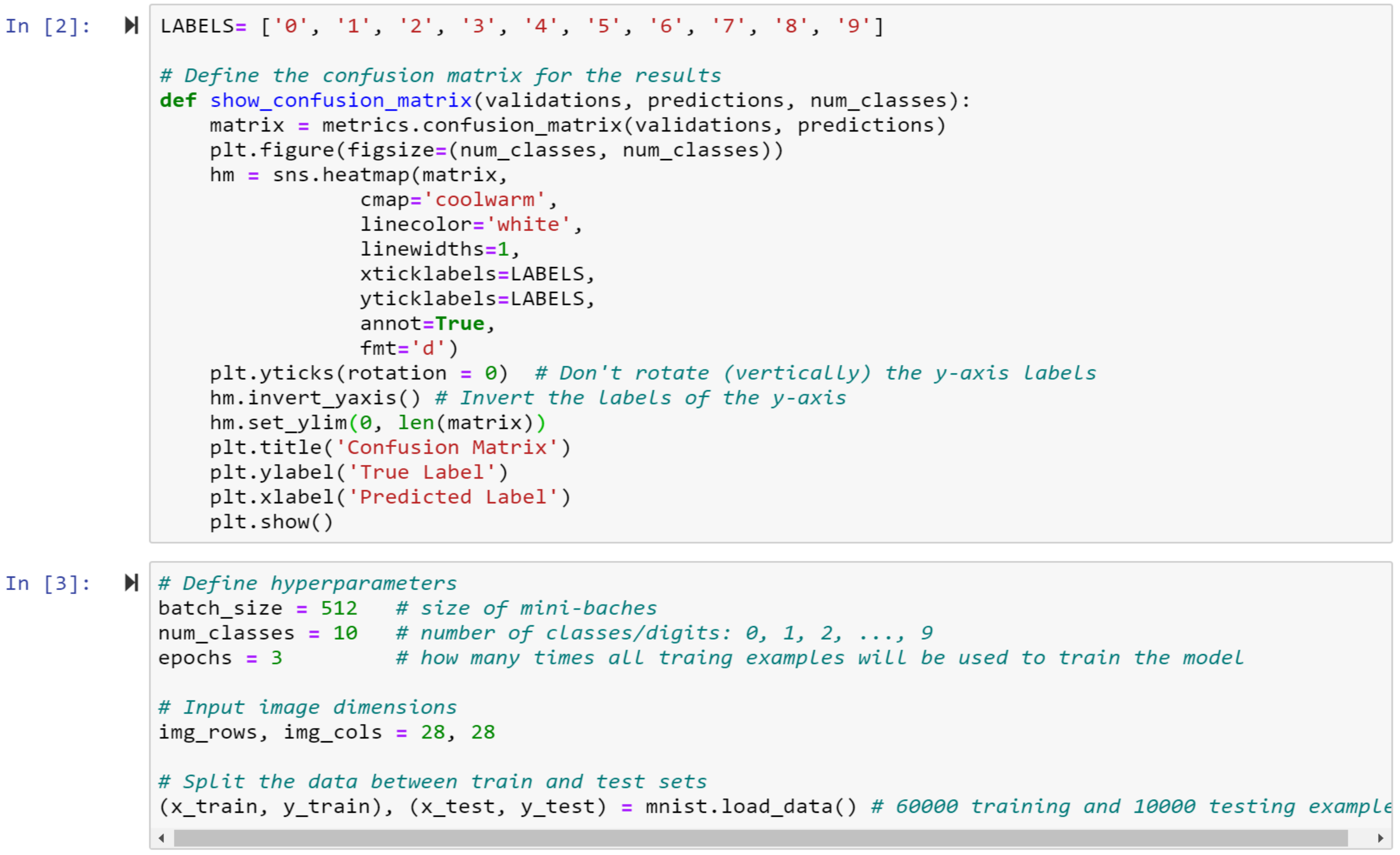
Import of libraries and setting of the parameters:



```
In [1]:  '''Trains a simple ConvNet on the MNIST dataset. It gets over 99.60% test accuracy after 48 epochs
         (but there is still a margin for hyperparameter tuning). Training can take an hour or so!'''

         # Import libraries
         from __future__ import print_function
         import numpy as np
         import math
         from math import ceil
         import tensorflow as tf
         import os
         import seaborn as sns
         import matplotlib.pyplot as plt  # library for plotting math functions
         import pandas as pd
         import keras   # Import keras framework with various functions, models and structures
         from keras.datasets import mnist # gets MNIST dataset from repository
         from keras.models import Sequential
         from keras.layers import Dense, Dropout, Flatten
         from keras.layers import Conv2D, MaxPooling2D
         from keras import backend as K
         from keras.preprocessing.image import ImageDataGenerator
         from keras.callbacks import ReduceLROnPlateau
         from sklearn import metrics
         from sklearn.metrics import confusion_matrix, classification_report

         # Set parameters for plots
         %matplotlib inline
         plt.rcParams['image.interpolation'] = 'nearest'
         plt.rcParams['image.cmap'] = 'gray'

         print ("TensorFlow version: " + tf.__version__)

         TensorFlow version: 2.1.0
```

MNIST Classification in Jupyter Notebook

Defining of hyperparameters and the function presenting results:



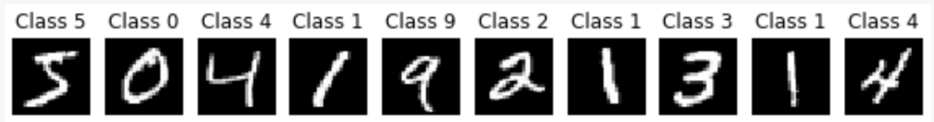
```
In [2]:
LABELS= ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# Define the confusion matrix for the results
def show_confusion_matrix(validations, predictions, num_classes):
    matrix = metrics.confusion_matrix(validations, predictions)
    plt.figure(figsize=(num_classes, num_classes))
    hm = sns.heatmap(matrix,
                cmap='coolwarm',
                linecolor='white',
                linewidths=1,
                xticklabels=LABELS,
                yticklabels=LABELS,
                annot=True,
                fmt='d')
    plt.yticks(rotation = 0)  # Don't rotate (vertically) the y-axis labels
    hm.invert_yaxis() # Invert the labels of the y-axis
    hm.set_ylim(0, len(matrix))
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()
```

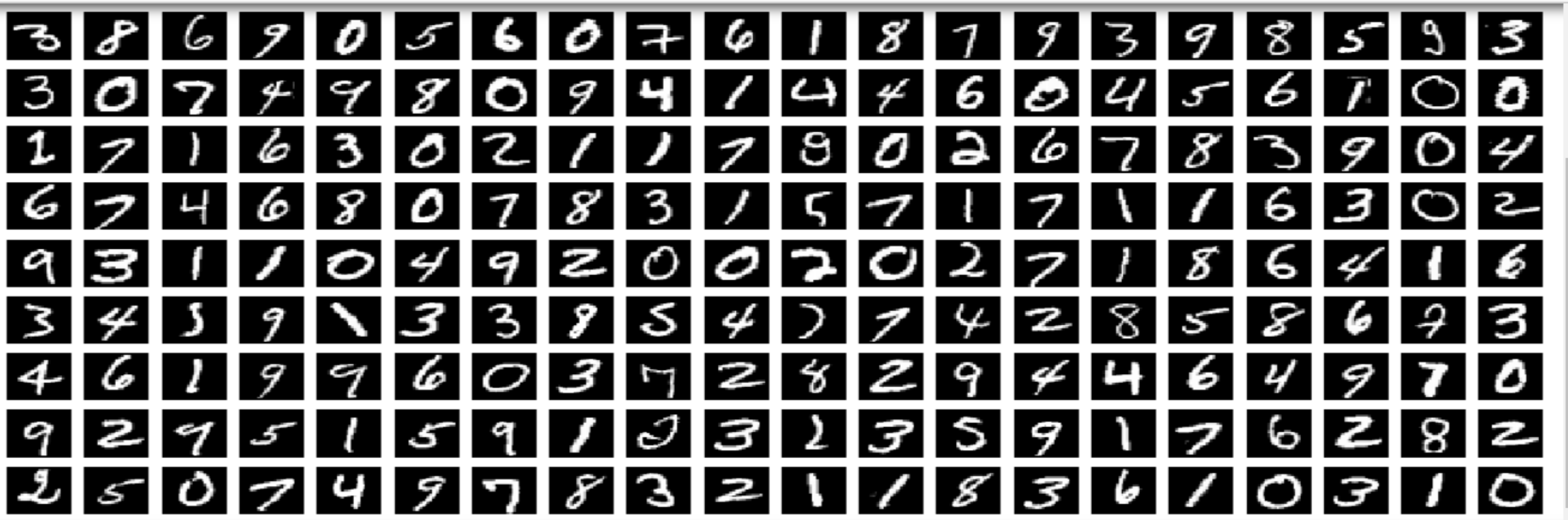
```
In [3]:
# Define hyperparameters
batch_size = 512   # size of mini-baches
num_classes = 10   # number of classes/digits: 0, 1, 2, ..., 9
epochs = 3         # how many times all traing examples will be used to train the model

# Input image dimensions
img_rows, img_cols = 28, 28

# Split the data between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data() # 60000 training and 10000 testing exampl
```

MNIST Classification in Jupyter Notebook

Sample training examples from MNIST set (handwritten digits):

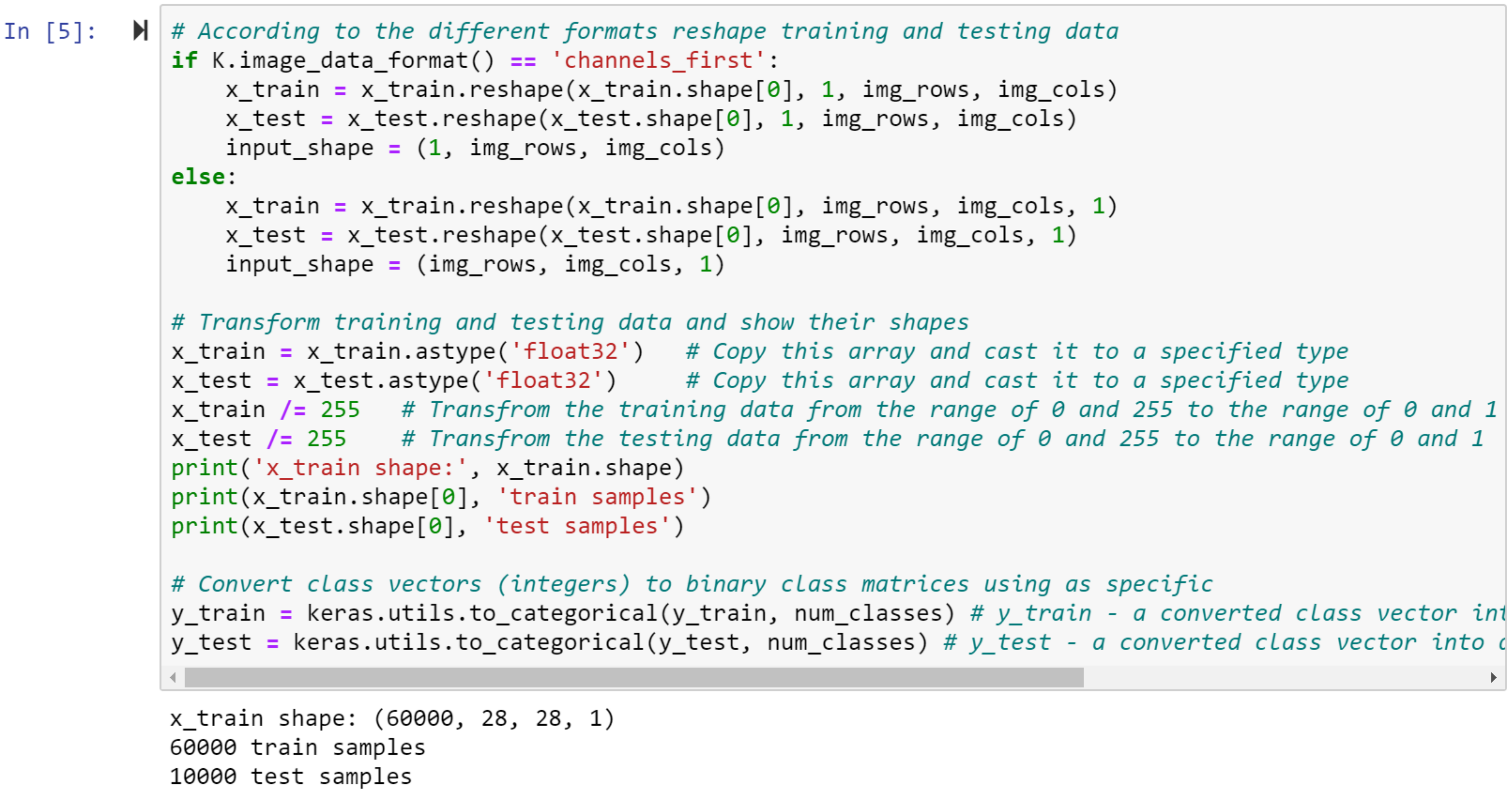
```
In [4]:  # Show a few sample digits from the training set
         plt.rcParams['figure.figsize'] = (2.5, 2.5) # set default size of plots
         col1 = 10
         row1 = 1
         fig = plt.figure(figsize=(col1, row1))
         for index in range(0, col1*row1):
             fig.add_subplot(row1, col1, index + 1)
             plt.axis('off')
             plt.imshow(x_train[index]) # index of the sample picture
             plt.title("Class " + str(y_train[index]))
         plt.show()

         # Show a few sample digits from the training set
         plt.rcParams['figure.figsize'] = (1.0, 1.0) # set default size of plots
         col2 = 20
         row2 = 10
         fig = plt.figure(figsize=(col2, row2))
         for index in range(col1*row1, col1*row1 + col2*row2):
             fig.add_subplot(row2, col2, index - col1*row1 + 1)
             plt.axis('off')
             plt.imshow(x_train[index]) # index of the sample picture
         plt.show()
```





MNIST Classification in Jupyter Notebook

Loading training data, changing the shapes of the matrices storing training and test data, transformation of the input data from [0, 255] to [0.0, 1.0] range, and conversion of numeric class names into categories:

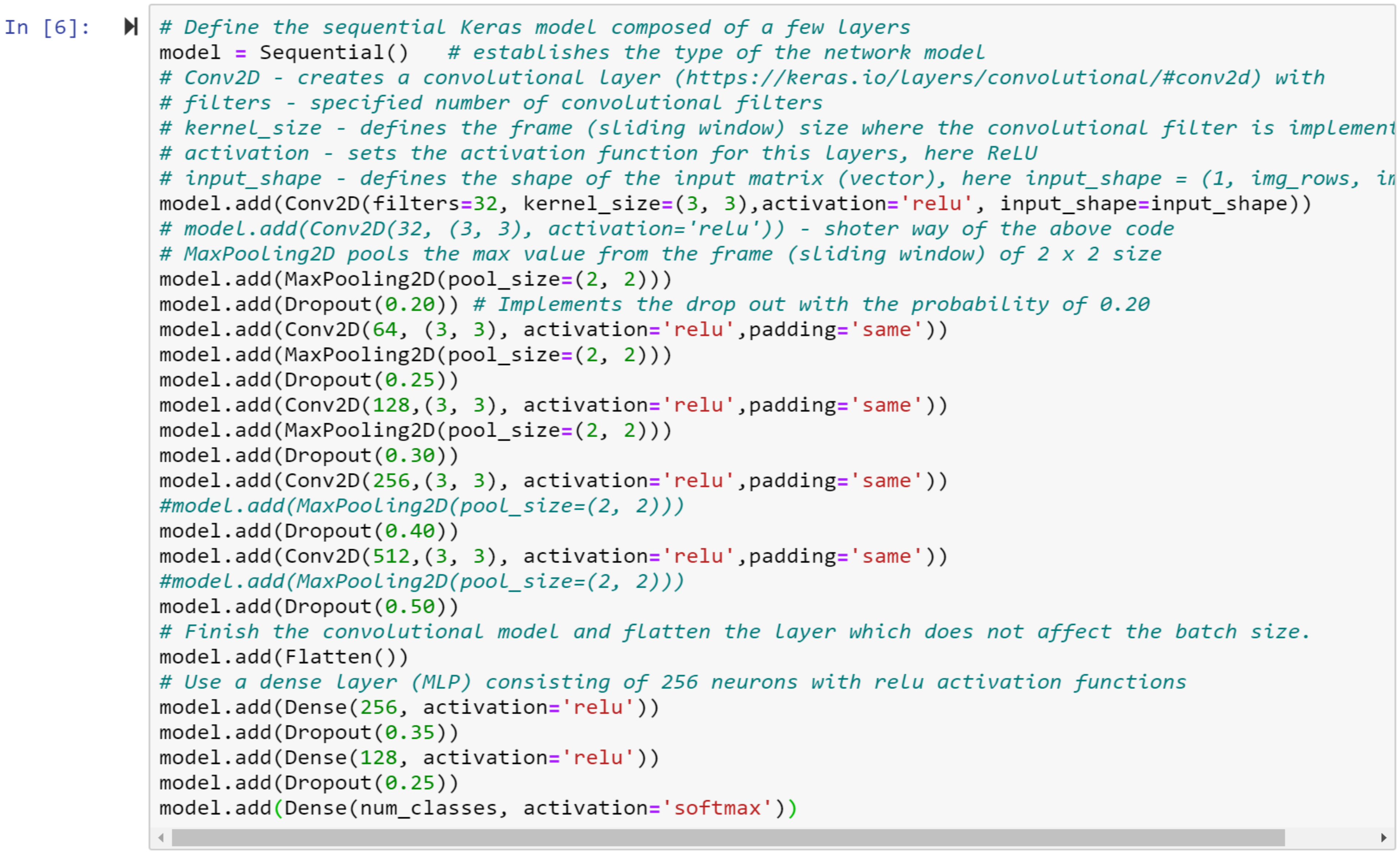


```
In [5]:
# According to the different formats reshape training and testing data
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# Transform training and testing data and show their shapes
x_train = x_train.astype('float32')   # Copy this array and cast it to a specified type
x_test = x_test.astype('float32')     # Copy this array and cast it to a specified type
x_train /= 255   # Transfrom the training data from the range of 0 and 255 to the range of 0 and 1
x_test /= 255    # Transfrom the testing data from the range of 0 and 255 to the range of 0 and 1
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors (integers) to binary class matrices using as specific
y_train = keras.utils.to_categorical(y_train, num_classes) # y_train - a converted class vector int
y_test = keras.utils.to_categorical(y_test, num_classes) # y_test - a converted class vector into 
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

MNIST Classification in Jupyter Notebook

Building a neural network structure (computational model):

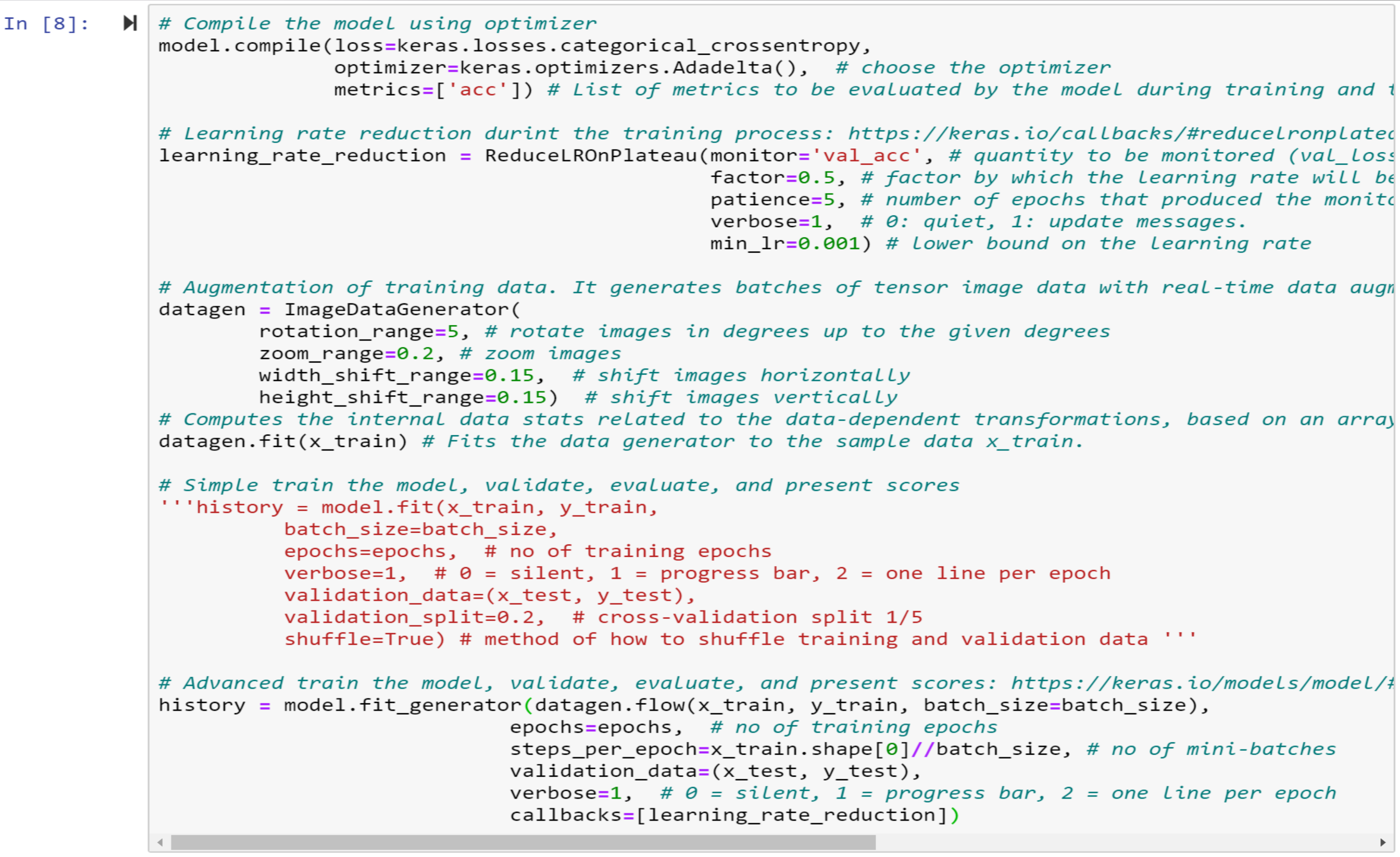


In [6]:

```
# Define the sequential Keras model composed of a few layers
model = Sequential()   # establishes the type of the network model
# Conv2D - creates a convolutional layer (https://keras.io/layers/convolutional/#conv2d) with
# filters - specified number of convolutional filters
# kernel_size - defines the frame (sliding window) size where the convolutional filter is implement
# activation - sets the activation function for this layers, here ReLU
# input_shape - defines the shape of the input matrix (vector), here input_shape = (1, img_rows, im
model.add(Conv2D(filters=32, kernel_size=(3, 3),activation='relu', input_shape=input_shape))
# model.add(Conv2D(32, (3, 3), activation='relu')) - shoter way of the above code
# MaxPooling2D pools the max value from the frame (sliding window) of 2 x 2 size
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.20)) # Implements the drop out with the probability of 0.20
model.add(Conv2D(64, (3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128,(3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.30))
model.add(Conv2D(256,(3, 3), activation='relu',padding='same'))
#model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.40))
model.add(Conv2D(512,(3, 3), activation='relu',padding='same'))
#model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.50))
# Finish the convolutional model and flatten the layer which does not affect the batch size.
model.add(Flatten())
# Use a dense layer (MLP) consisting of 256 neurons with relu activation functions
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(num_classes, activation='softmax'))
```

MNIST Classification in Jupyter Notebook

Compilation, optimization, data generation, augmentation and learning:

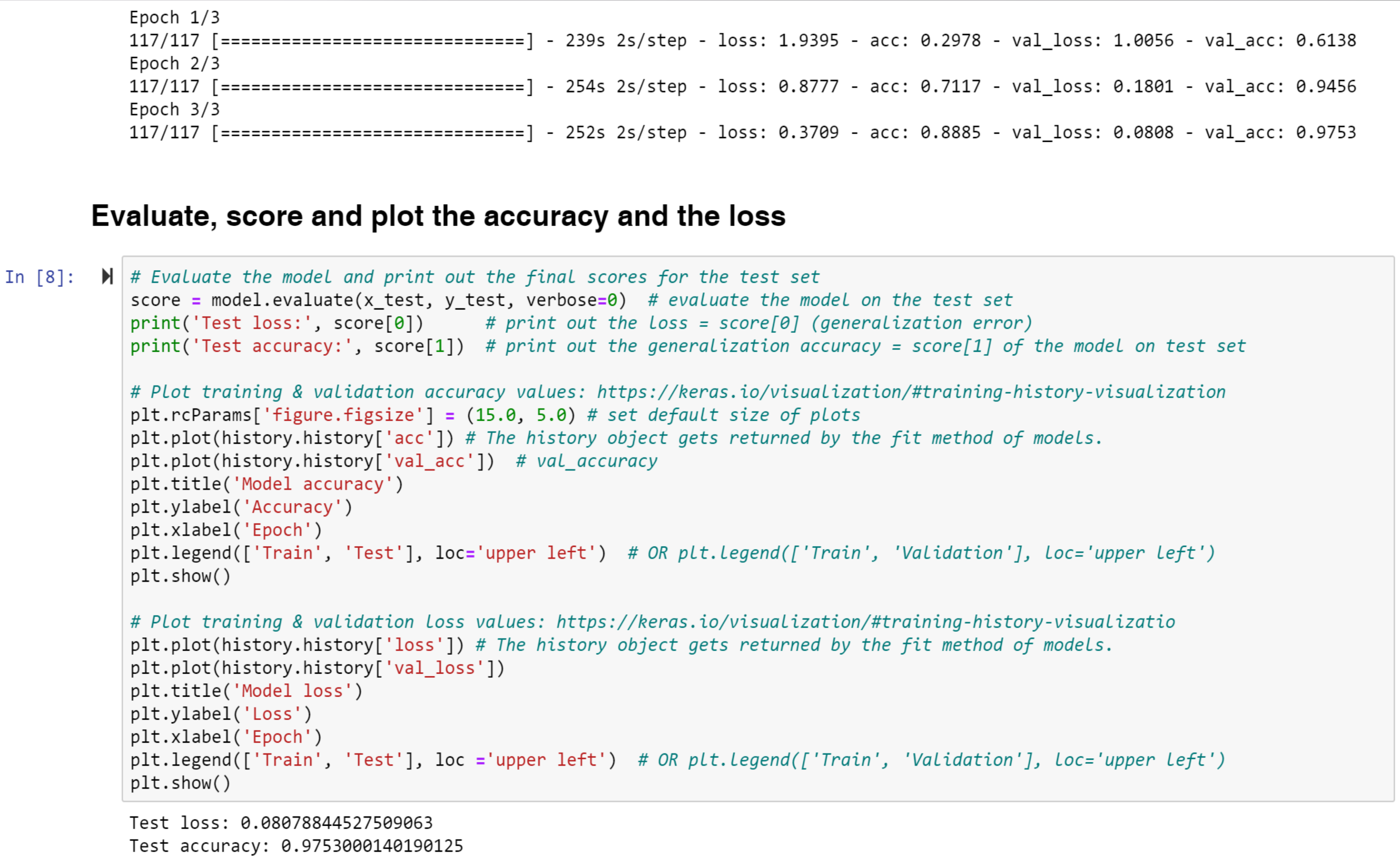


In [8]:

```
# Compile the model using optimizer
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),  # choose the optimizer
              metrics=['acc']) # List of metrics to be evaluated by the model during training and t

# Learning rate reduction durint the training process: https://keras.io/callbacks/#reducelronplatea
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc', # quantity to be monitored (val_loss
                                            factor=0.5, # factor by which the learning rate will be
                                            patience=5, # number of epochs that produced the monito
                                            verbose=1,  # 0: quiet, 1: update messages.
                                            min_lr=0.001) # lower bound on the learning rate

# Augmentation of training data. It generates batches of tensor image data with real-time data augm
datagen = ImageDataGenerator(
        rotation_range=5, # rotate images in degrees up to the given degrees
        zoom_range=0.2, # zoom images
        width_shift_range=0.15,  # shift images horizontally
        height_shift_range=0.15)  # shift images vertically
# Computes the internal data stats related to the data-dependent transformations, based on an array
datagen.fit(x_train) # Fits the data generator to the sample data x_train.

# Simple train the model, validate, evaluate, and present scores
'''history = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,  # no of training epochs
          verbose=1,  # 0 = silent, 1 = progress bar, 2 = one line per epoch
          validation_data=(x_test, y_test),
          validation_split=0.2,  # cross-validation split 1/5
          shuffle=True) # method of how to shuffle training and validation data '''

# Advanced train the model, validate, evaluate, and present scores: https://keras.io/models/model/#
history = model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                            epochs=epochs,  # no of training epochs
                            steps_per_epoch=x_train.shape[0]//batch_size, # no of mini-batches
                            validation_data=(x_test, y_test),
                            verbose=1,  # 0 = silent, 1 = progress bar, 2 = one line per epoch
                            callbacks=[learning_rate_reduction])
```

MNIST Classification in Jupyter Notebook

Model evaluation, convergence drawing and error charts:



```
Epoch 1/3
117/117 [==============================] - 239s 2s/step - loss: 1.9395 - acc: 0.2978 - val_loss: 1.0056 - val_acc: 0.6138
Epoch 2/3
117/117 [==============================] - 254s 2s/step - loss: 0.8777 - acc: 0.7117 - val_loss: 0.1801 - val_acc: 0.9456
Epoch 3/3
117/117 [==============================] - 252s 2s/step - loss: 0.3709 - acc: 0.8885 - val_loss: 0.0808 - val_acc: 0.9753
```

Evaluate, score and plot the accuracy and the loss

In [8]:

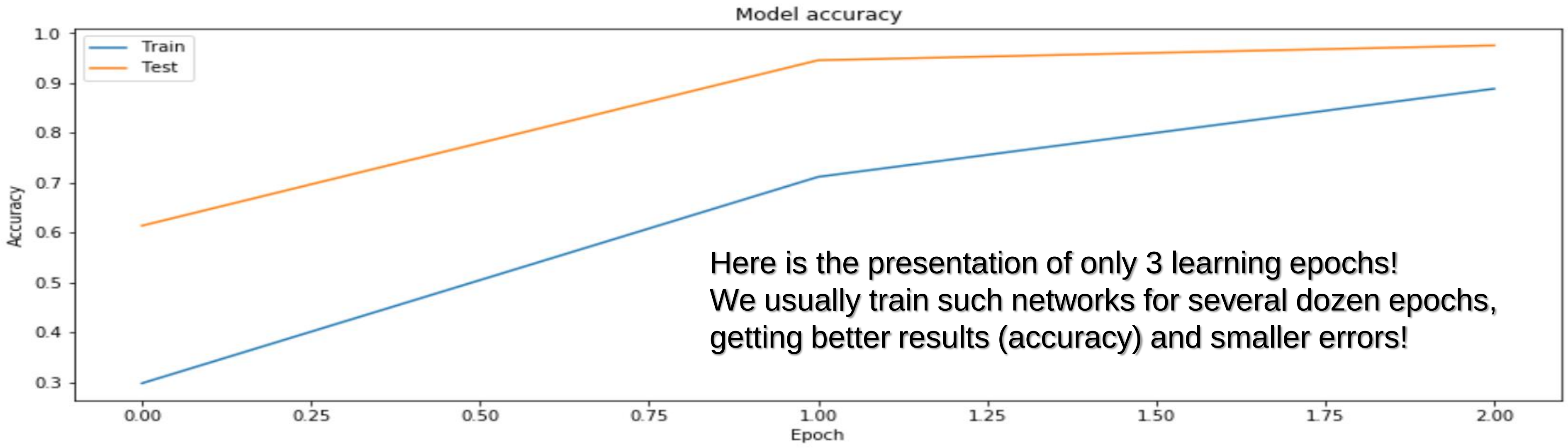
```
# Evaluate the model and print out the final scores for the test set
score = model.evaluate(x_test, y_test, verbose=0)  # evaluate the model on the test set
print('Test loss:', score[0])      # print out the loss = score[0] (generalization error)
print('Test accuracy:', score[1])  # print out the generalization accuracy = score[1] of the model on test set

# Plot training & validation accuracy values: https://keras.io/visualization/#training-history-visualization
plt.rcParams['figure.figsize'] = (15.0, 5.0) # set default size of plots
plt.plot(history.history['acc']) # The history object gets returned by the fit method of models.
plt.plot(history.history['val_acc'])  # val_accuracy
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')  # OR plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values: https://keras.io/visualization/#training-history-visualizatio
plt.plot(history.history['loss']) # The history object gets returned by the fit method of models.
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc ='upper left')  # OR plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

```
Test loss: 0.08078844527509063
Test accuracy: 0.9753000140190125
```

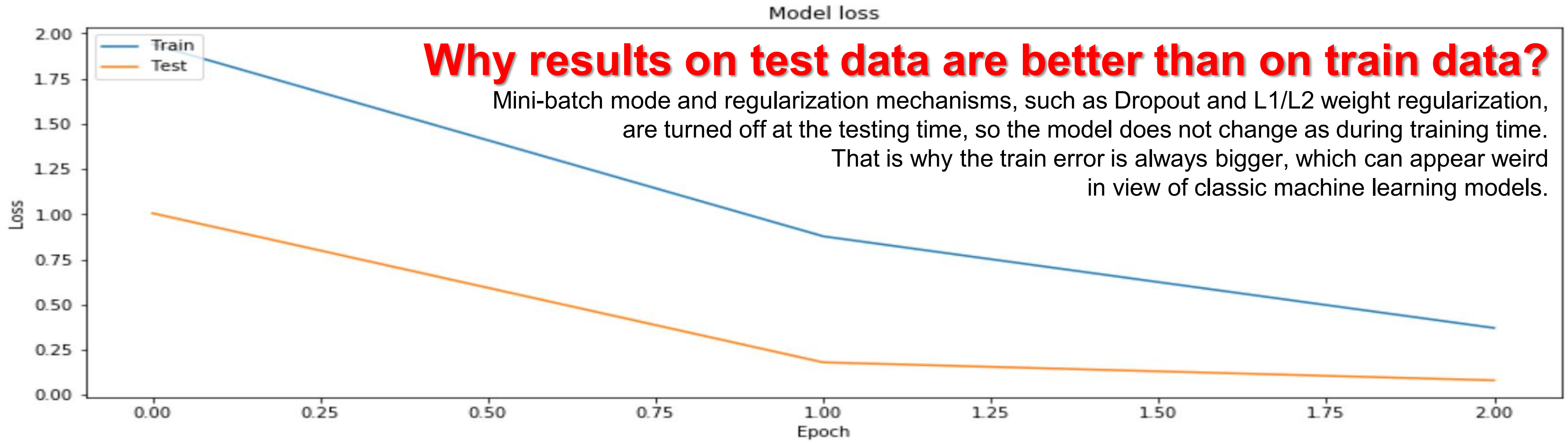
MNIST Classification in Jupyter Notebook

Model evaluation, convergence drawing and error charts:

```
Test loss: 0.08078844527509063
Test accuracy: 0.9753000140190125
```



Here is the presentation of only 3 learning epochs!
We usually train such networks for several dozen epochs,
getting better results (accuracy) and smaller errors!

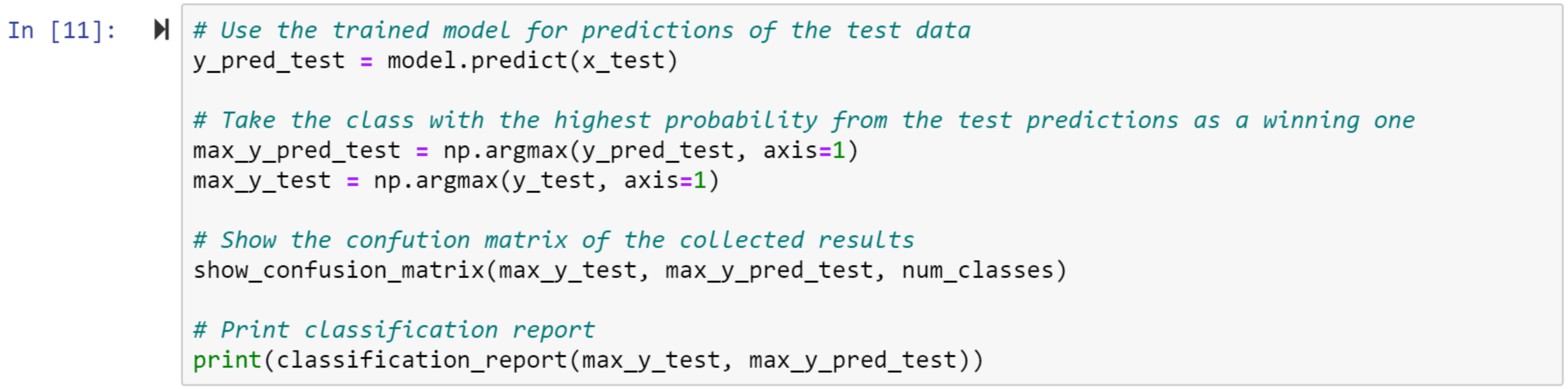


Why results on test data are better than on train data?

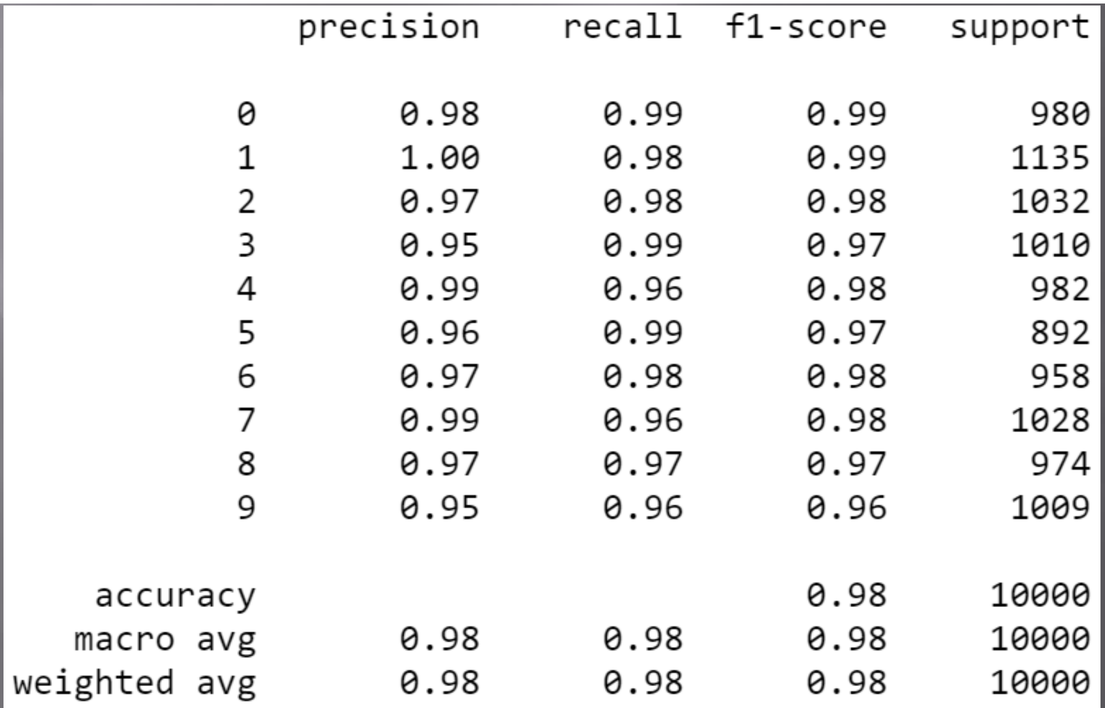
Mini-batch mode and regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at the testing time, so the model does not change as during training time. That is why the train error is always bigger, which can appear weird in view of classic machine learning models.

MNIST Classification in Jupyter Notebook

Generation of summaries of the learning process



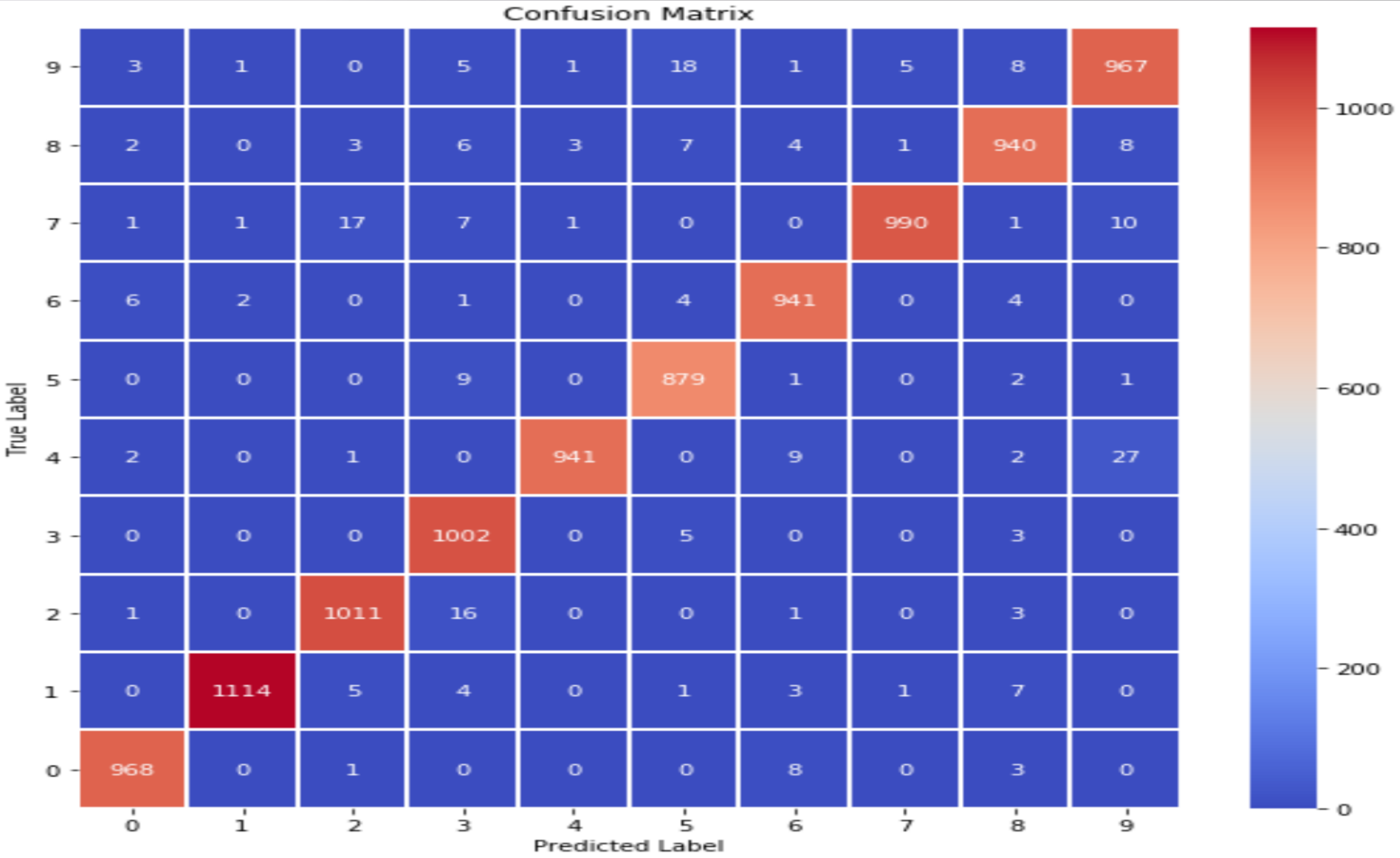
```
In [11]:
# Use the trained model for predictions of the test data
y_pred_test = model.predict(x_test)

# Take the class with the highest probability from the test predictions as a winning one
max_y_pred_test = np.argmax(y_pred_test, axis=1)
max_y_test = np.argmax(y_test, axis=1)

# Show the confution matrix of the collected results
show_confusion_matrix(max_y_test, max_y_pred_test, num_classes)

# Print classification report
print(classification_report(max_y_test, max_y_pred_test))
```



```
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       1.00      0.98      0.99      1135
           2       0.97      0.98      0.98      1032
           3       0.95      0.99      0.97      1010
           4       0.99      0.96      0.98       982
           5       0.96      0.99      0.97       892
           6       0.97      0.98      0.98       958
           7       0.99      0.96      0.98      1028
           8       0.97      0.97      0.97       974
           9       0.95      0.96      0.96      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

MNIST Classification in Jupyter Notebook

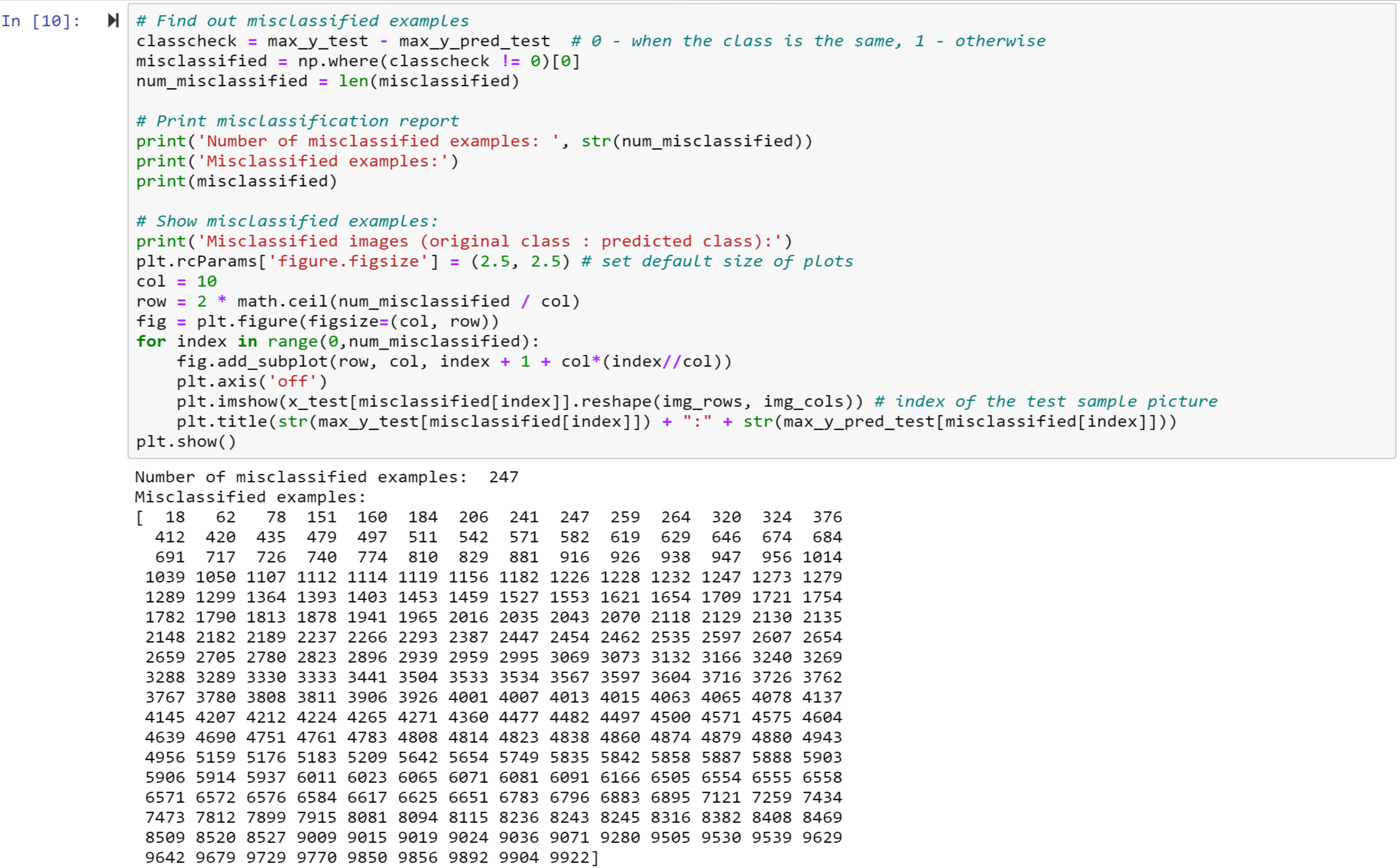
Generation of a confusion (error) matrix in the form of a heat map:



True \ Predicted	0	1	2	3	4	5	6	7	8	9
9	3	1	0	5	1	18	1	5	8	967
8	2	0	3	6	3	7	4	1	940	8
7	1	1	17	7	1	0	0	990	1	10
6	6	2	0	1	0	4	941	0	4	0
5	0	0	0	9	0	879	1	0	2	1
4	2	0	1	0	941	0	9	0	2	27
3	0	0	0	1002	0	5	0	0	3	0
2	1	0	1011	16	0	0	1	0	3	0
1	0	1114	5	4	0	1	3	1	7	0
0	968	0	1	0	0	0	8	0	3	0

MNIST Classification in Jupyter Notebook

Counting and filtering incorrectly classified test data:



In [10]:

```
# Find out misclassified examples
classcheck = max_y_test - max_y_pred_test  # 0 - when the class is the same, 1 - otherwise
misclassified = np.where(classcheck != 0)[0]
num_misclassified = len(misclassified)

# Print misclassification report
print('Number of misclassified examples: ', str(num_misclassified))
print('Misclassified examples:')
print(misclassified)

# Show misclassified examples:
print('Misclassified images (original class : predicted class):')
plt.rcParams['figure.figsize'] = (2.5, 2.5) # set default size of plots
col = 10
row = 2 * math.ceil(num_misclassified / col)
fig = plt.figure(figsize=(col, row))
for index in range(0,num_misclassified):
    fig.add_subplot(row, col, index + 1 + col*(index//col))
    plt.axis('off')
    plt.imshow(x_test[misclassified[index]].reshape(img_rows, img_cols)) # index of the test sample picture
    plt.title(str(max_y_test[misclassified[index]]) + ":" + str(max_y_pred_test[misclassified[index]]))
plt.show()
```

```
Number of misclassified examples:  247
Misclassified examples:
[  18   62   78  151  160  184  206  241  247  259  264  320  324  376
  412  420  435  479  497  511  542  571  582  619  629  646  674  684
  691  717  726  740  774  810  829  881  916  926  938  947  956 1014
 1039 1050 1107 1112 1114 1119 1156 1182 1226 1228 1232 1247 1273 1279
 1289 1299 1364 1393 1403 1453 1459 1527 1553 1621 1654 1709 1721 1754
 1782 1790 1813 1878 1941 1965 2016 2035 2043 2070 2118 2129 2130 2135
 2148 2182 2189 2237 2266 2293 2387 2447 2454 2462 2535 2597 2607 2654
 2659 2705 2780 2823 2896 2939 2959 2995 3069 3073 3132 3166 3240 3269
 3288 3289 3330 3333 3441 3504 3533 3534 3567 3597 3604 3716 3726 3762
 3767 3780 3808 3811 3906 3926 4001 4007 4013 4015 4063 4065 4078 4137
 4145 4207 4212 4224 4265 4271 4360 4477 4482 4497 4500 4571 4575 4604
 4639 4690 4751 4761 4783 4808 4814 4823 4838 4860 4874 4879 4880 4943
 4956 5159 5176 5183 5209 5642 5654 5749 5835 5842 5858 5887 5888 5903
 5906 5914 5937 6011 6023 6065 6071 6081 6091 6166 6505 6554 6555 6558
 6571 6572 6576 6584 6617 6625 6651 6783 6796 6883 6895 7121 7259 7434
 7473 7812 7899 7915 8081 8094 8115 8236 8243 8245 8316 8382 8408 8469
 8509 8520 8527 9009 9015 9019 9024 9036 9071 9280 9505 9530 9539 9629
 9642 9679 9729 9770 9850 9856 9892 9904 9922]
```

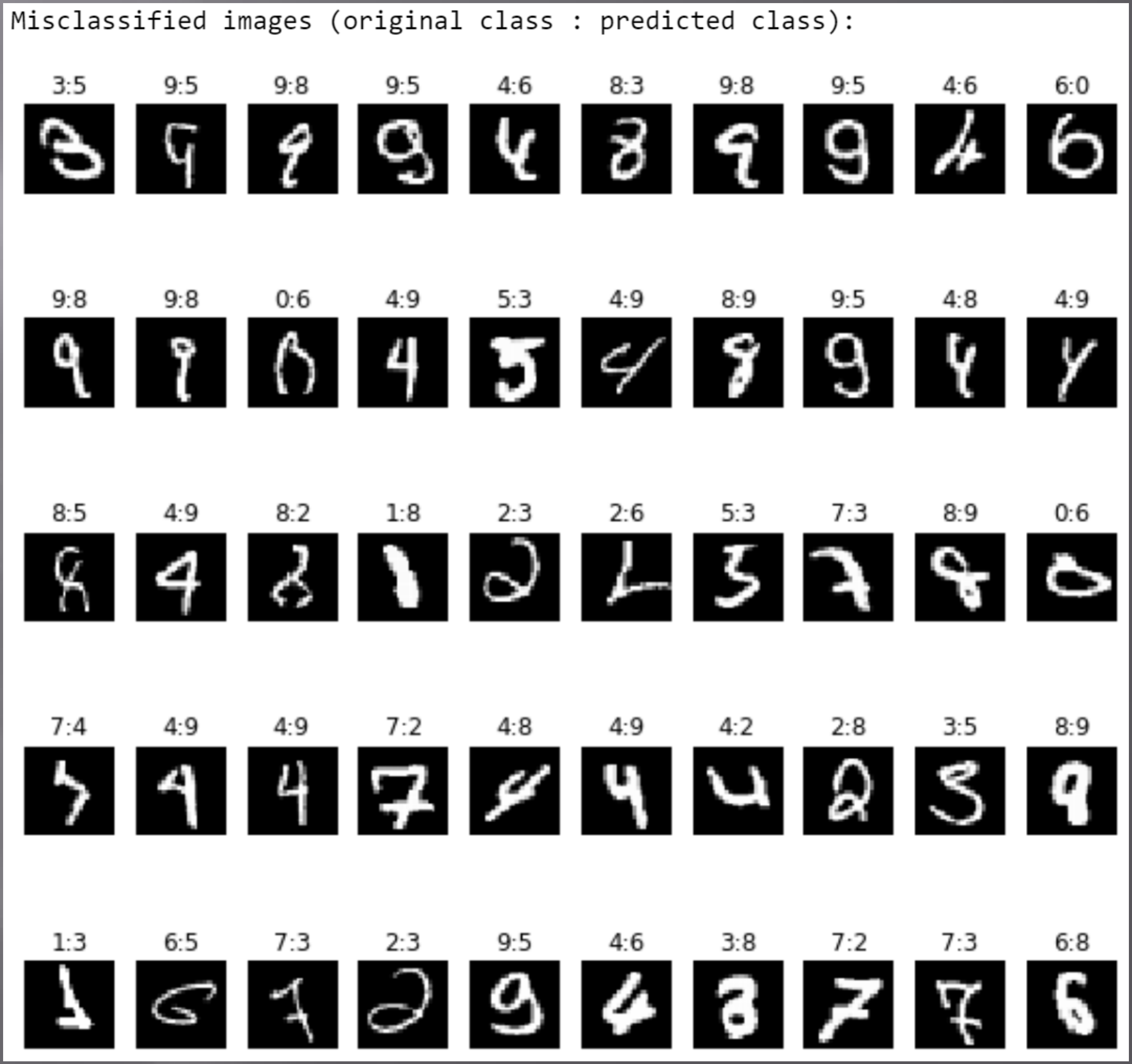
MNIST Classification in Jupyter Notebook

247 out of 10,000 incorrectly classified test patterns:

One might wonder why the network had difficulty in classifying them?

Of course, such a network can be taught further to achieve a smaller error!

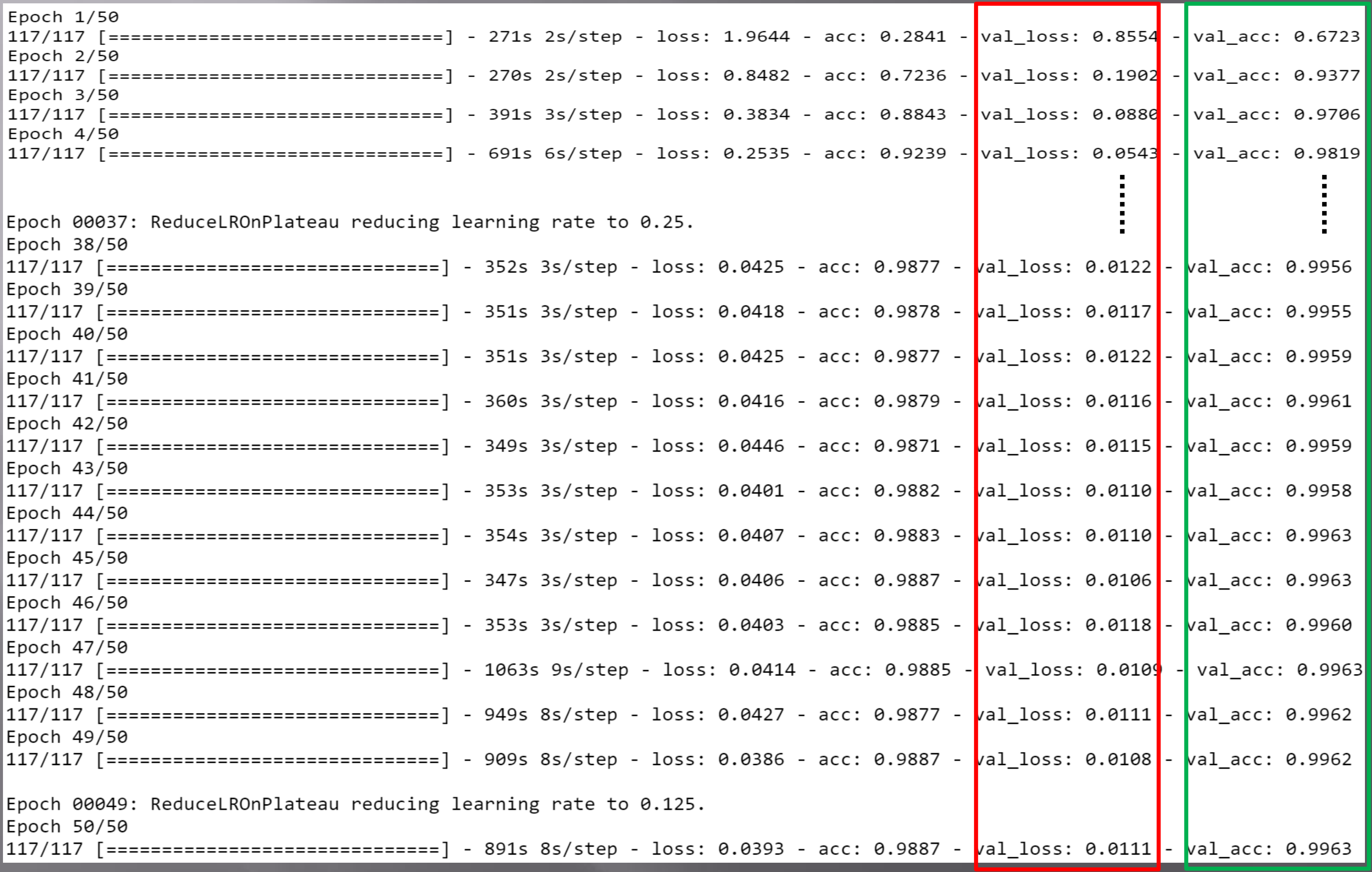
This network was taught only for 3 epochs!



```
Misclassified images (original class : predicted class):

 3:5   9:5   9:8   9:5   4:6   8:3   9:8   9:5   4:6   6:0

 9:8   9:8   0:6   4:9   5:3   4:9   8:9   9:5   4:8   4:9

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

 7:4   4:9   4:9   7:2   4:8   4:9   4:2   2:8   3:5   8:9

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

MNIST Classification in Jupyter Notebook

Now, let's try to train the network for 50 epochs:

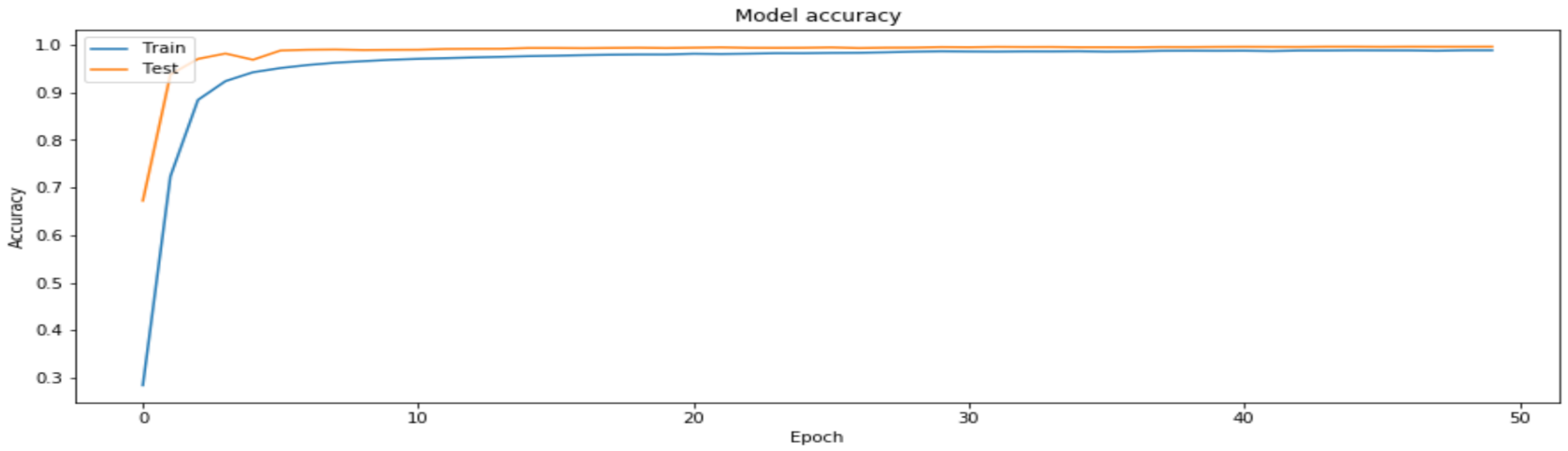


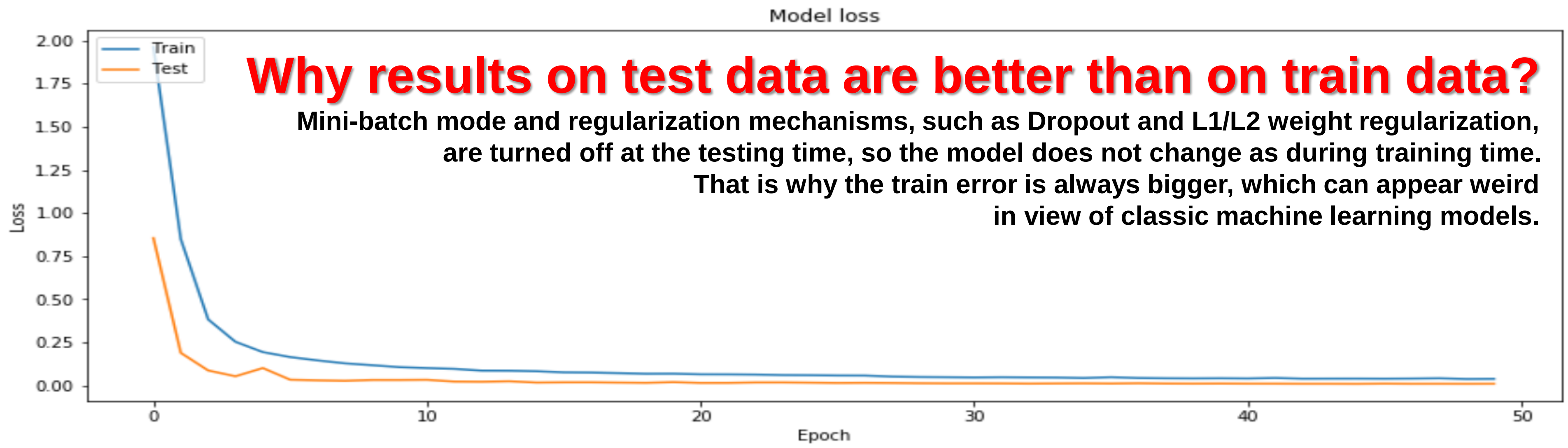
```
Epoch 1/50
117/117 [==============================] - 271s 2s/step - loss: 1.9644 - acc: 0.2841 - val_loss: 0.8554 - val_acc: 0.6723
Epoch 2/50
117/117 [==============================] - 270s 2s/step - loss: 0.8482 - acc: 0.7236 - val_loss: 0.1902 - val_acc: 0.9377
Epoch 3/50
117/117 [==============================] - 391s 3s/step - loss: 0.3834 - acc: 0.8843 - val_loss: 0.0880 - val_acc: 0.9706
Epoch 4/50
117/117 [==============================] - 691s 6s/step - loss: 0.2535 - acc: 0.9239 - val_loss: 0.0543 - val_acc: 0.9819


Epoch 00037: ReduceLROnPlateau reducing learning rate to 0.25.
Epoch 38/50
117/117 [==============================] - 352s 3s/step - loss: 0.0425 - acc: 0.9877 - val_loss: 0.0122 - val_acc: 0.9956
Epoch 39/50
117/117 [==============================] - 351s 3s/step - loss: 0.0418 - acc: 0.9878 - val_loss: 0.0117 - val_acc: 0.9955
Epoch 40/50
117/117 [==============================] - 351s 3s/step - loss: 0.0425 - acc: 0.9877 - val_loss: 0.0122 - val_acc: 0.9959
Epoch 41/50
117/117 [==============================] - 360s 3s/step - loss: 0.0416 - acc: 0.9879 - val_loss: 0.0116 - val_acc: 0.9961
Epoch 42/50
117/117 [==============================] - 349s 3s/step - loss: 0.0446 - acc: 0.9871 - val_loss: 0.0115 - val_acc: 0.9959
Epoch 43/50
117/117 [==============================] - 353s 3s/step - loss: 0.0401 - acc: 0.9882 - val_loss: 0.0110 - val_acc: 0.9958
Epoch 44/50
117/117 [==============================] - 354s 3s/step - loss: 0.0407 - acc: 0.9883 - val_loss: 0.0110 - val_acc: 0.9963
Epoch 45/50
117/117 [==============================] - 347s 3s/step - loss: 0.0406 - acc: 0.9887 - val_loss: 0.0106 - val_acc: 0.9963
Epoch 46/50
117/117 [==============================] - 353s 3s/step - loss: 0.0403 - acc: 0.9885 - val_loss: 0.0118 - val_acc: 0.9960
Epoch 47/50
117/117 [==============================] - 1063s 9s/step - loss: 0.0414 - acc: 0.9885 - val_loss: 0.0109 - val_acc: 0.9963
Epoch 48/50
117/117 [==============================] - 949s 8s/step - loss: 0.0427 - acc: 0.9877 - val_loss: 0.0111 - val_acc: 0.9962
Epoch 49/50
117/117 [==============================] - 909s 8s/step - loss: 0.0386 - acc: 0.9887 - val_loss: 0.0108 - val_acc: 0.9962

Epoch 00049: ReduceLROnPlateau reducing learning rate to 0.125.
Epoch 50/50
117/117 [==============================] - 891s 8s/step - loss: 0.0393 - acc: 0.9887 - val_loss: 0.0111 - val_acc: 0.9963
```

MNIST Classification in Jupyter Notebook

Graphs of learning convergence (accuracy) and error minimization (loss):

```
Test loss: 0.011101936267607016
Test accuracy: 0.9962999820709229
```



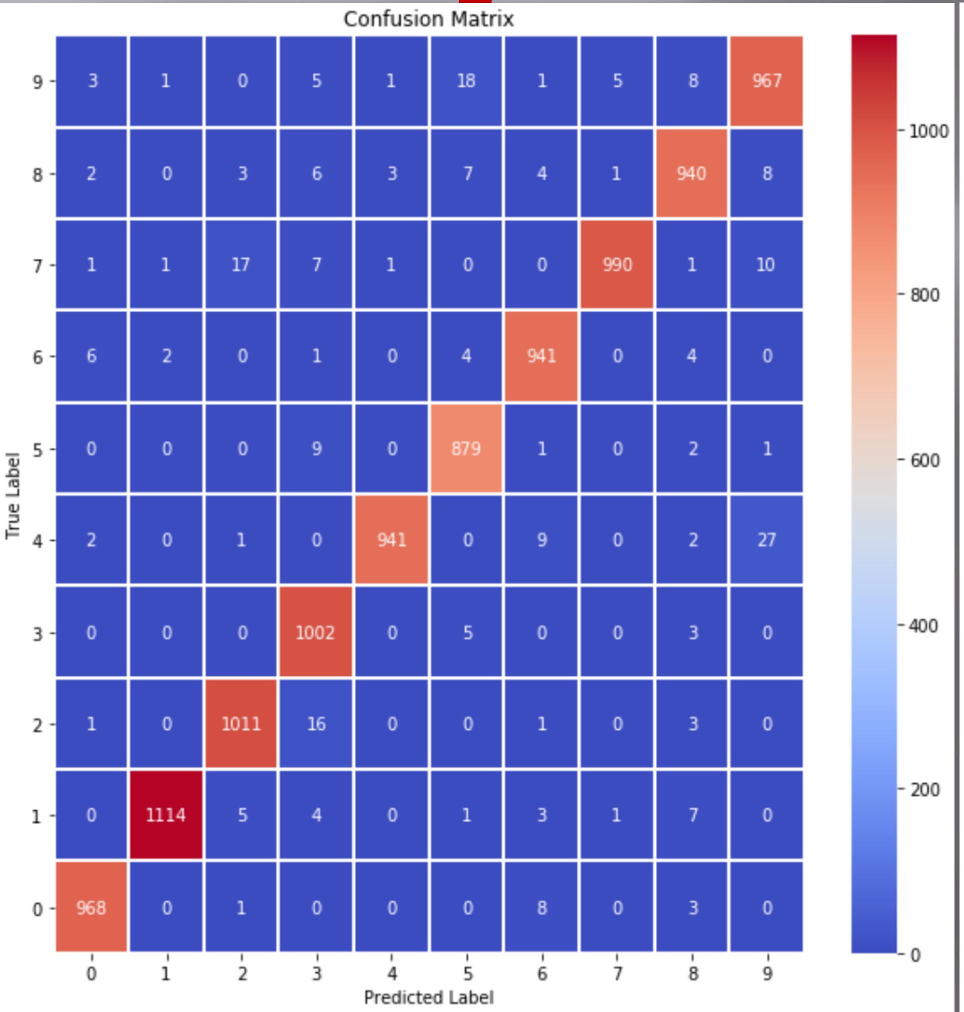


Why results on test data are better than on train data?

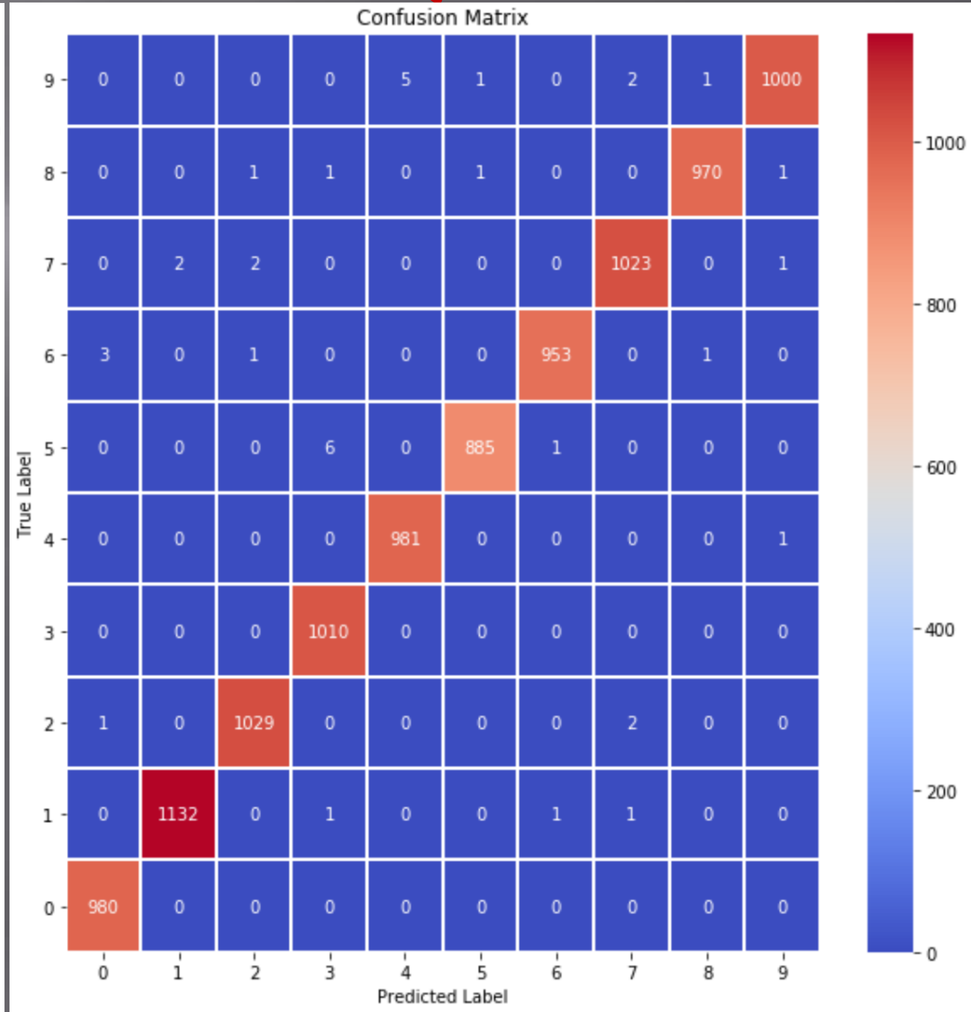
Mini-batch mode and regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at the testing time, so the model does not change as during training time. That is why the train error is always bigger, which can appear weird in view of classic machine learning models.

MNIST Classification in Jupyter Notebook

The confusion matrix has also improved: more patterns migrate towards the diagonal (correct classifications) from other regions:



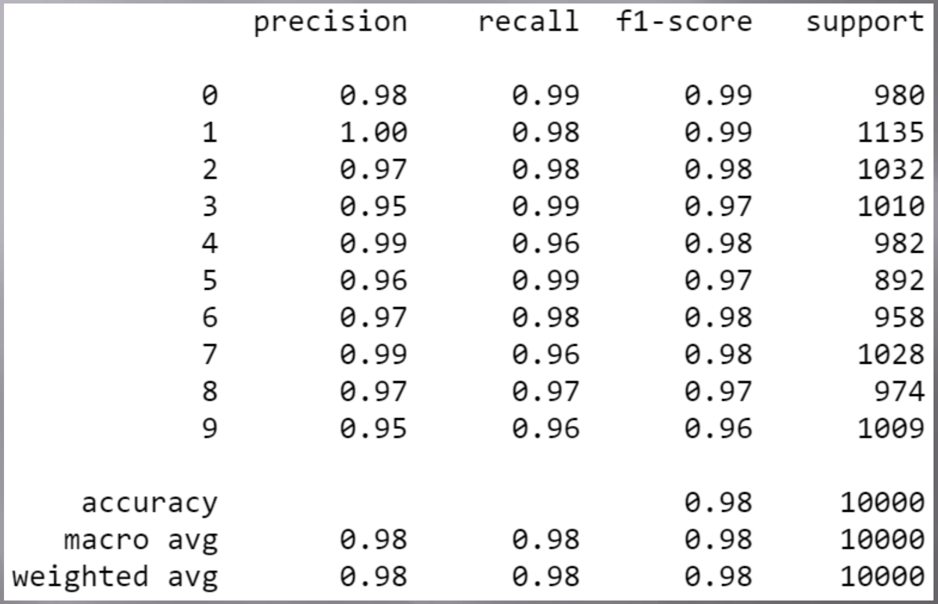
True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	3	1	0	5	1	18	1	5	8	967
8	2	0	3	6	3	7	4	1	940	8
7	1	1	17	7	1	0	0	990	1	10
6	6	2	0	1	0	4	941	0	4	0
5	0	0	0	9	0	879	1	0	2	1
4	2	0	1	0	941	0	9	0	2	27
3	0	0	0	1002	0	5	0	0	3	0
2	1	0	1011	16	0	0	1	0	3	0
1	0	1114	5	4	0	1	3	1	7	0
0	968	0	1	0	0	0	8	0	3	0



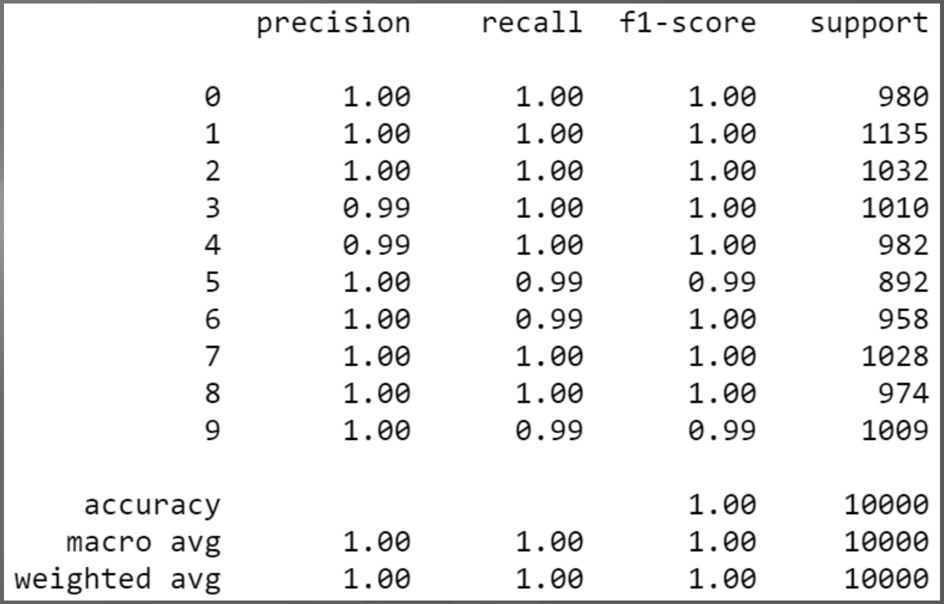
True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	0	0	0	0	5	1	0	2	1	1000
8	0	0	1	1	0	1	0	0	970	1
7	0	2	2	0	0	0	0	1023	0	1
6	3	0	1	0	0	0	953	0	1	0
5	0	0	0	6	0	885	1	0	0	0
4	0	0	0	0	981	0	0	0	0	1
3	0	0	0	1010	0	0	0	0	0	0
2	1	0	1029	0	0	0	0	2	0	0
1	0	1132	0	1	0	0	1	1	0	0
0	980	0	0	0	0	0	0	0	0	0

MNIST Classification in Jupyter Notebook

The number and the accuracy of correctly classified examples for all individual classes increase:



```
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       1.00      0.98      0.99      1135
           2       0.97      0.98      0.98      1032
           3       0.95      0.99      0.97      1010
           4       0.99      0.96      0.98       982
           5       0.96      0.99      0.97       892
           6       0.97      0.98      0.98       958
           7       0.99      0.96      0.98      1028
           8       0.97      0.97      0.97       974
           9       0.95      0.96      0.96      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

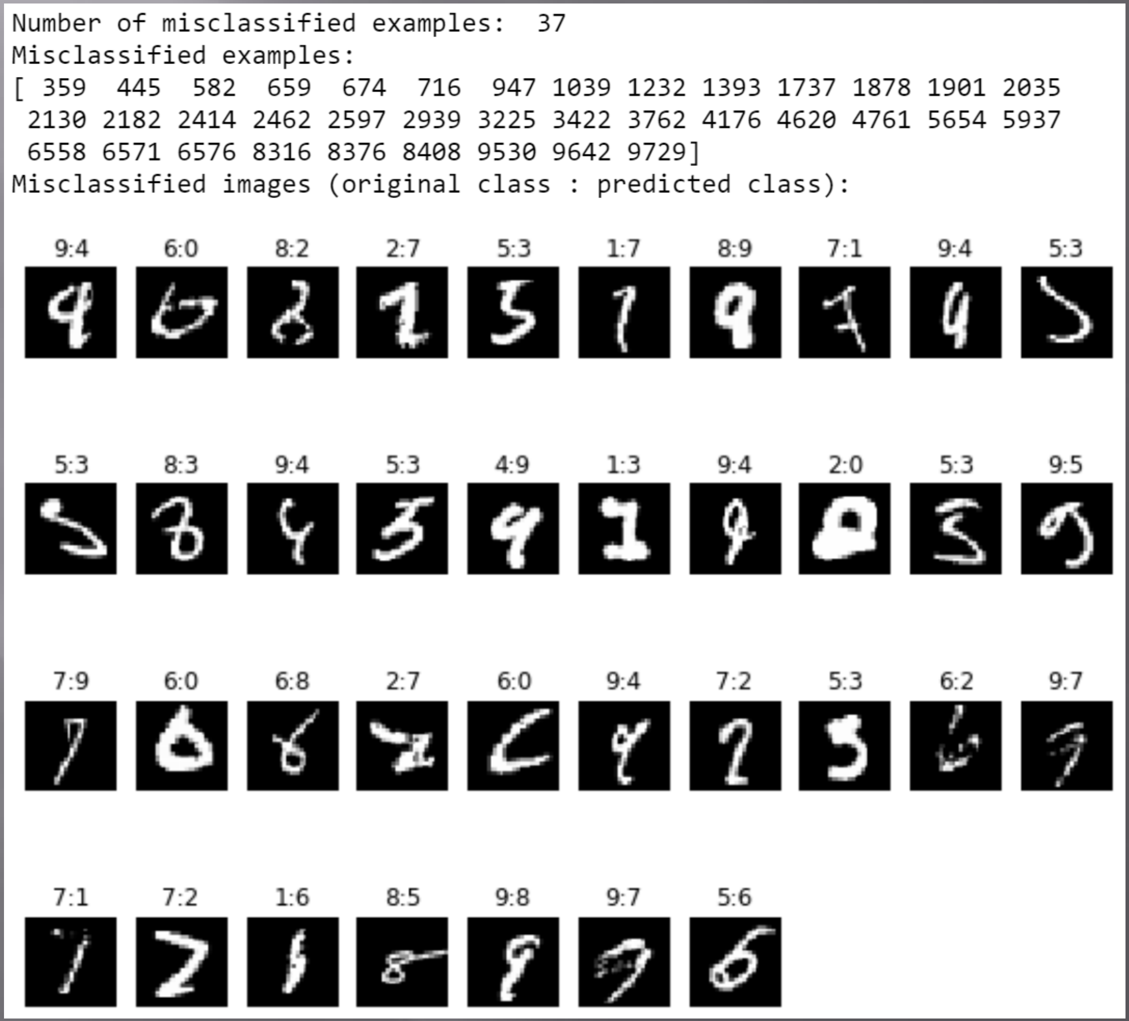


```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       980
           1       1.00      1.00      1.00      1135
           2       1.00      1.00      1.00      1032
           3       0.99      1.00      1.00      1010
           4       0.99      1.00      1.00       982
           5       1.00      0.99      0.99       892
           6       1.00      0.99      1.00       958
           7       1.00      1.00      1.00      1028
           8       1.00      1.00      1.00       974
           9       1.00      0.99      0.99      1009

    accuracy                           1.00     10000
   macro avg       1.00      1.00      1.00     10000
weighted avg       1.00      1.00      1.00     10000
```

However, we can see that the process of network training is not over yet and should be continued for several dozen epochs.

MNIST Classification in Jupyter Notebook

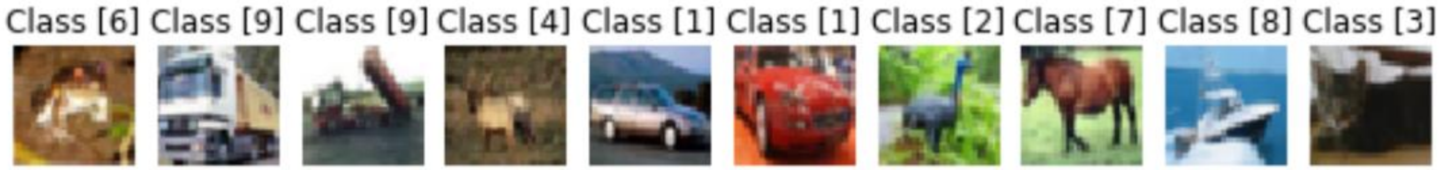
The number of misclassified examples after 50 epochs compared to 3 epochs has dropped from 247 to 37 out of 10,000 test examples, resulting in an error of 0.37%. Here are the misclassified examples:

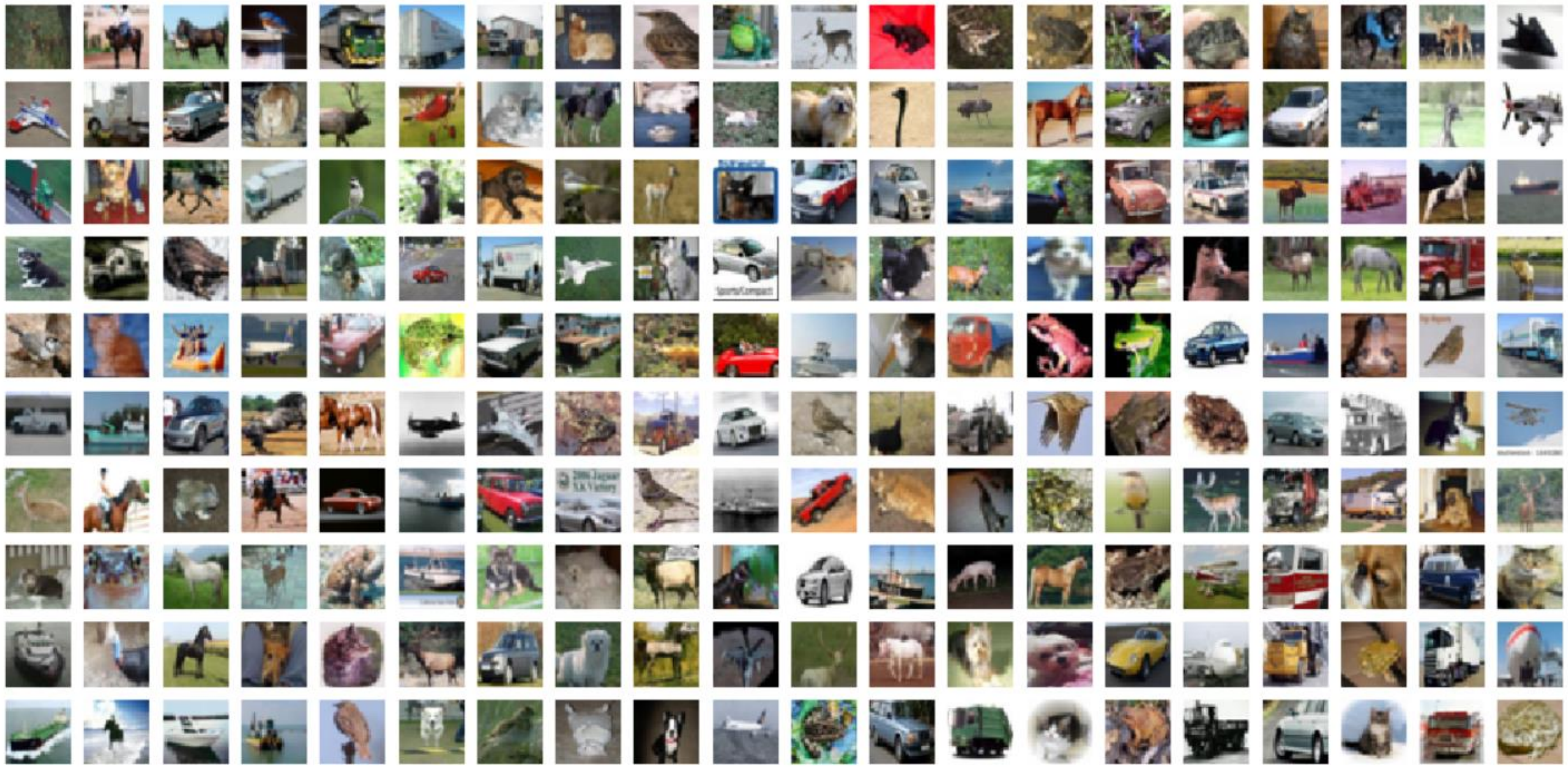


```
Number of misclassified examples:  37
Misclassified examples:
[ 359  445  582  659  674  716  947 1039 1232 1393 1737 1878 1901 2035
 2130 2182 2414 2462 2597 2939 3225 3422 3762 4176 4620 4761 5654 5937
 6558 6571 6576 8316 8376 8408 9530 9642 9729]
Misclassified images (original class : predicted class):
```

CIFAR-10 Classification in Jupyter

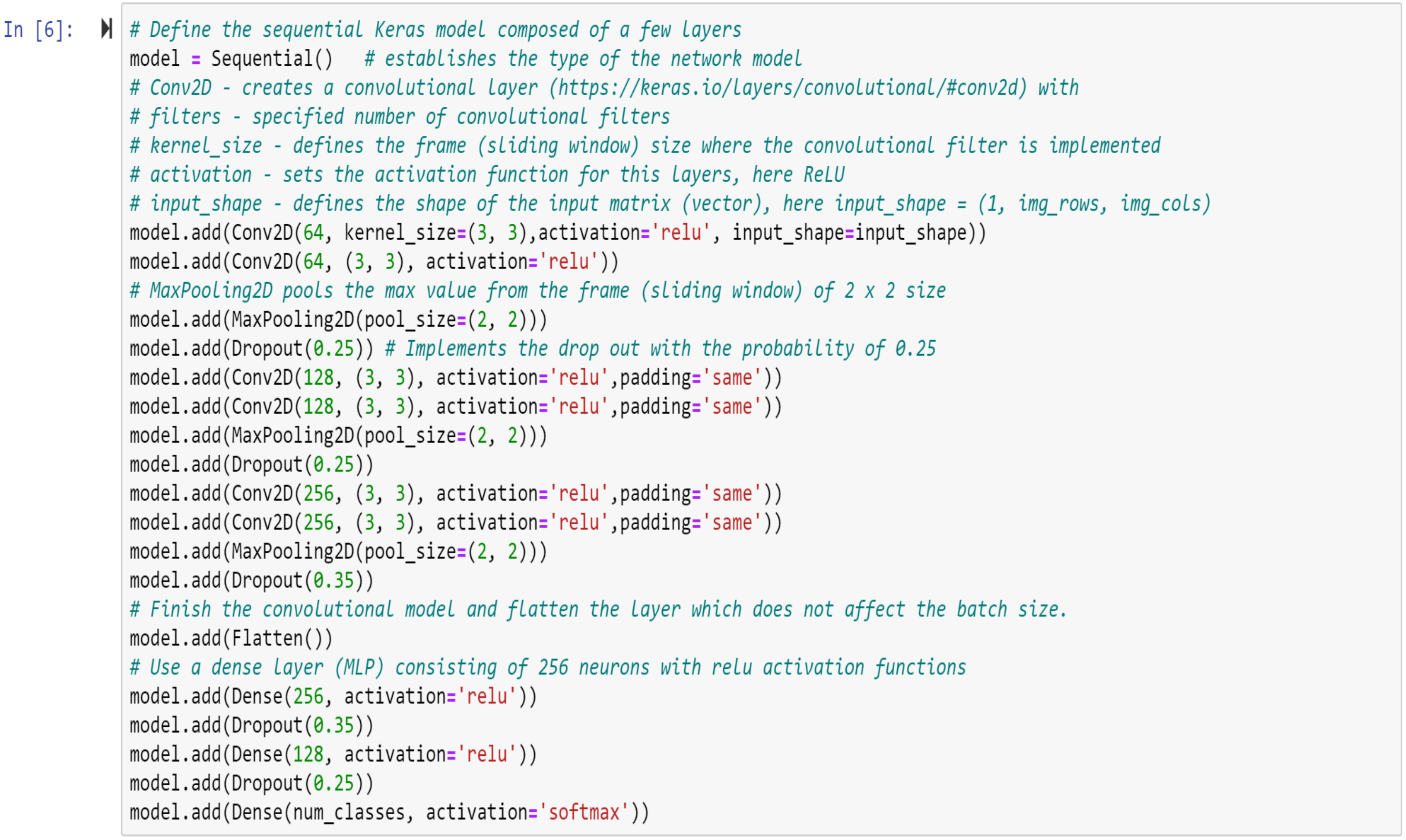
Classification of images 32 x 32 pixels to 10 classes (3 learning epochs):





CIFAR-10 Classification in Jupyter

Create the network structure

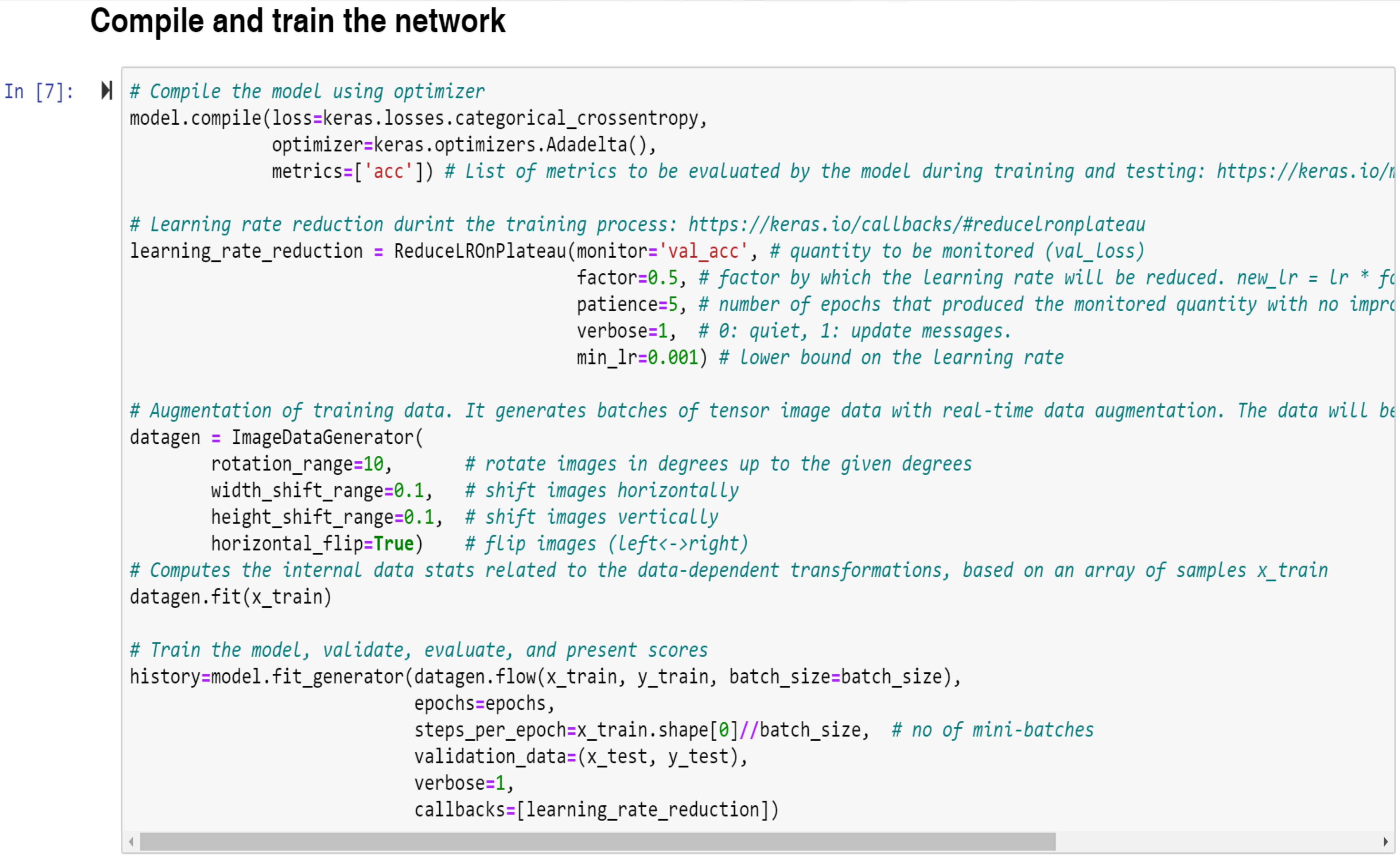


In [6]:

```
# Define the sequential Keras model composed of a few layers
model = Sequential()   # establishes the type of the network model
# Conv2D - creates a convolutional layer (https://keras.io/layers/convolutional/#conv2d) with
# filters - specified number of convolutional filters
# kernel_size - defines the frame (sliding window) size where the convolutional filter is implemented
# activation - sets the activation function for this layers, here ReLU
# input_shape - defines the shape of the input matrix (vector), here input_shape = (1, img_rows, img_cols)
model.add(Conv2D(64, kernel_size=(3, 3),activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
# MaxPooling2D pools the max value from the frame (sliding window) of 2 x 2 size
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Implements the drop out with the probability of 0.25
model.add(Conv2D(128, (3, 3), activation='relu',padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(256, (3, 3), activation='relu',padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.35))
# Finish the convolutional model and flatten the layer which does not affect the batch size.
model.add(Flatten())
# Use a dense layer (MLP) consisting of 256 neurons with relu activation functions
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(num_classes, activation='softmax'))
```

CIFAR-10 Classification in Jupyter

Compilation, optimization , data augmentation (generation) and training:



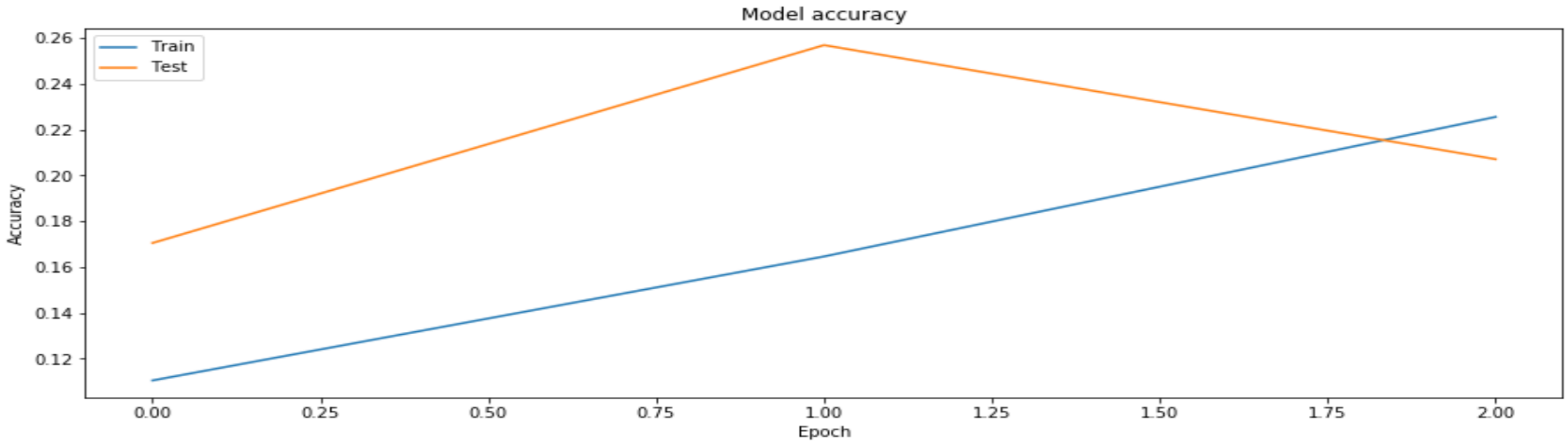
Compile and train the network

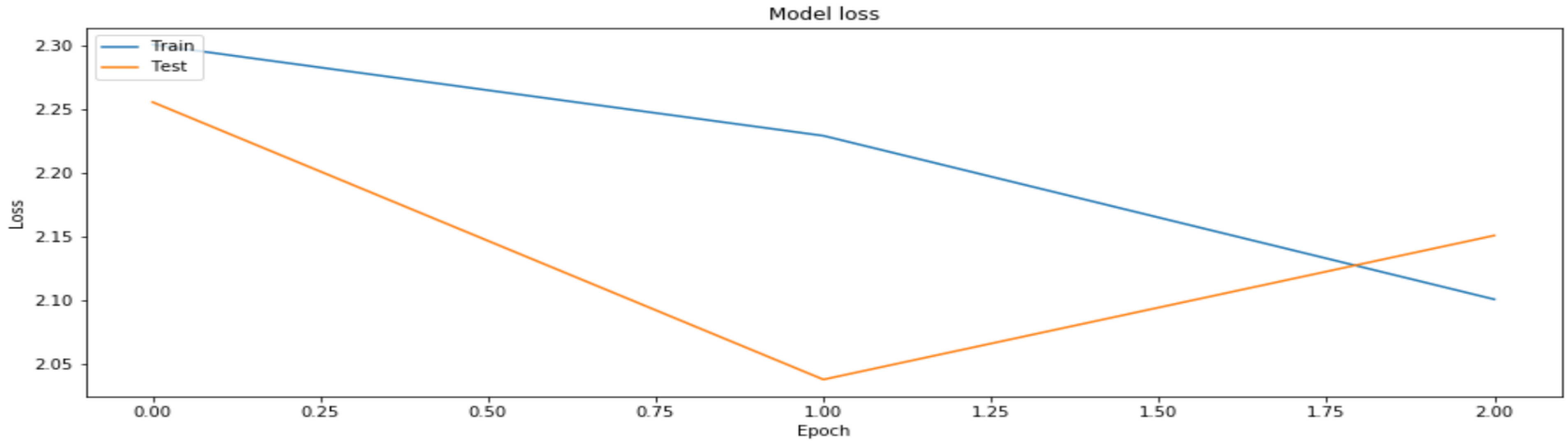
```
In [7]:  # Compile the model using optimizer
         model.compile(loss=keras.losses.categorical_crossentropy,
                       optimizer=keras.optimizers.Adadelta(),
                       metrics=['acc']) # List of metrics to be evaluated by the model during training and testing: https://keras.io/m

         # Learning rate reduction durint the training process: https://keras.io/callbacks/#reducelronplateau
         learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc', # quantity to be monitored (val_loss)
                                                     factor=0.5, # factor by which the learning rate will be reduced. new_lr = lr * fa
                                                     patience=5, # number of epochs that produced the monitored quantity with no impro
                                                     verbose=1,  # 0: quiet, 1: update messages.
                                                     min_lr=0.001) # lower bound on the learning rate

         # Augmentation of training data. It generates batches of tensor image data with real-time data augmentation. The data will be
         datagen = ImageDataGenerator(
                 rotation_range=10,       # rotate images in degrees up to the given degrees
                 width_shift_range=0.1,   # shift images horizontally
                 height_shift_range=0.1,  # shift images vertically
                 horizontal_flip=True)    # flip images (left<->right)
         # Computes the internal data stats related to the data-dependent transformations, based on an array of samples x_train
         datagen.fit(x_train)

         # Train the model, validate, evaluate, and present scores
         history=model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                                     epochs=epochs,
                                     steps_per_epoch=x_train.shape[0]//batch_size,  # no of mini-batches
                                     validation_data=(x_test, y_test),
                                     verbose=1,
                                     callbacks=[learning_rate_reduction])
```

CIFAR-10 Classification in Jupyter

Results of training after tree training epochs:

```
Test loss: 2.1507028507232664
Test accuracy: 0.2071000039577484
```



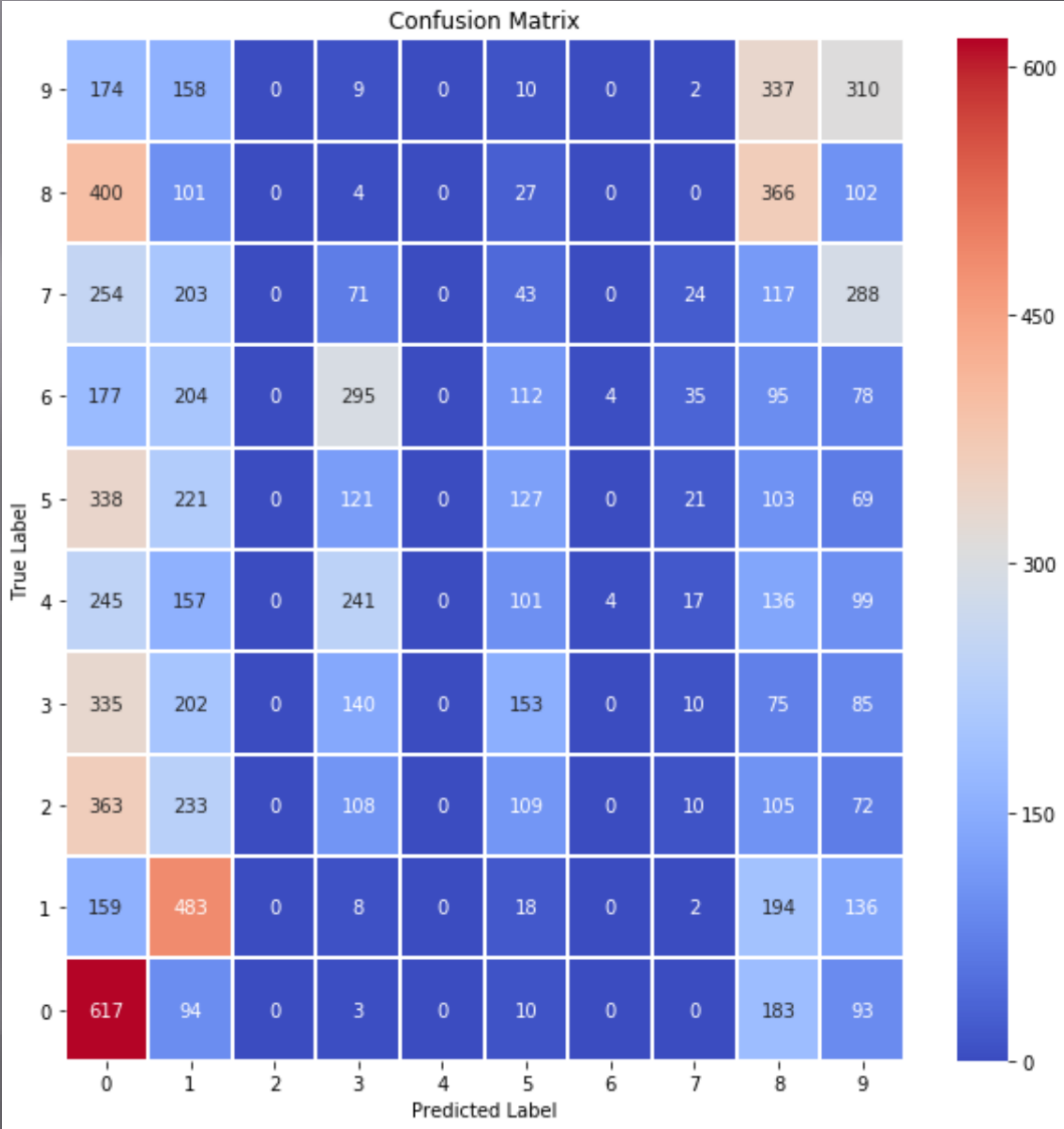


CIFAR-10 Classification in Jupyter

Confusion (error) martrix after three training epochs:

```
              precision    recall  f1-score   support

           0       0.20      0.62      0.30      1000
           1       0.23      0.48      0.32      1000
           2       0.00      0.00      0.00      1000
           3       0.14      0.14      0.14      1000
           4       0.00      0.00      0.00      1000
           5       0.18      0.13      0.15      1000
           6       0.50      0.00      0.01      1000
           7       0.20      0.02      0.04      1000
           8       0.21      0.37      0.27      1000
           9       0.23      0.31      0.27      1000

    accuracy                           0.21     10000
   macro avg       0.19      0.21      0.15     10000
weighted avg       0.19      0.21      0.15     10000
```

```
Number of misclassified examples:  7929
Misclassified examples:
[   0    3    4 ... 9994 9995 9999]
```

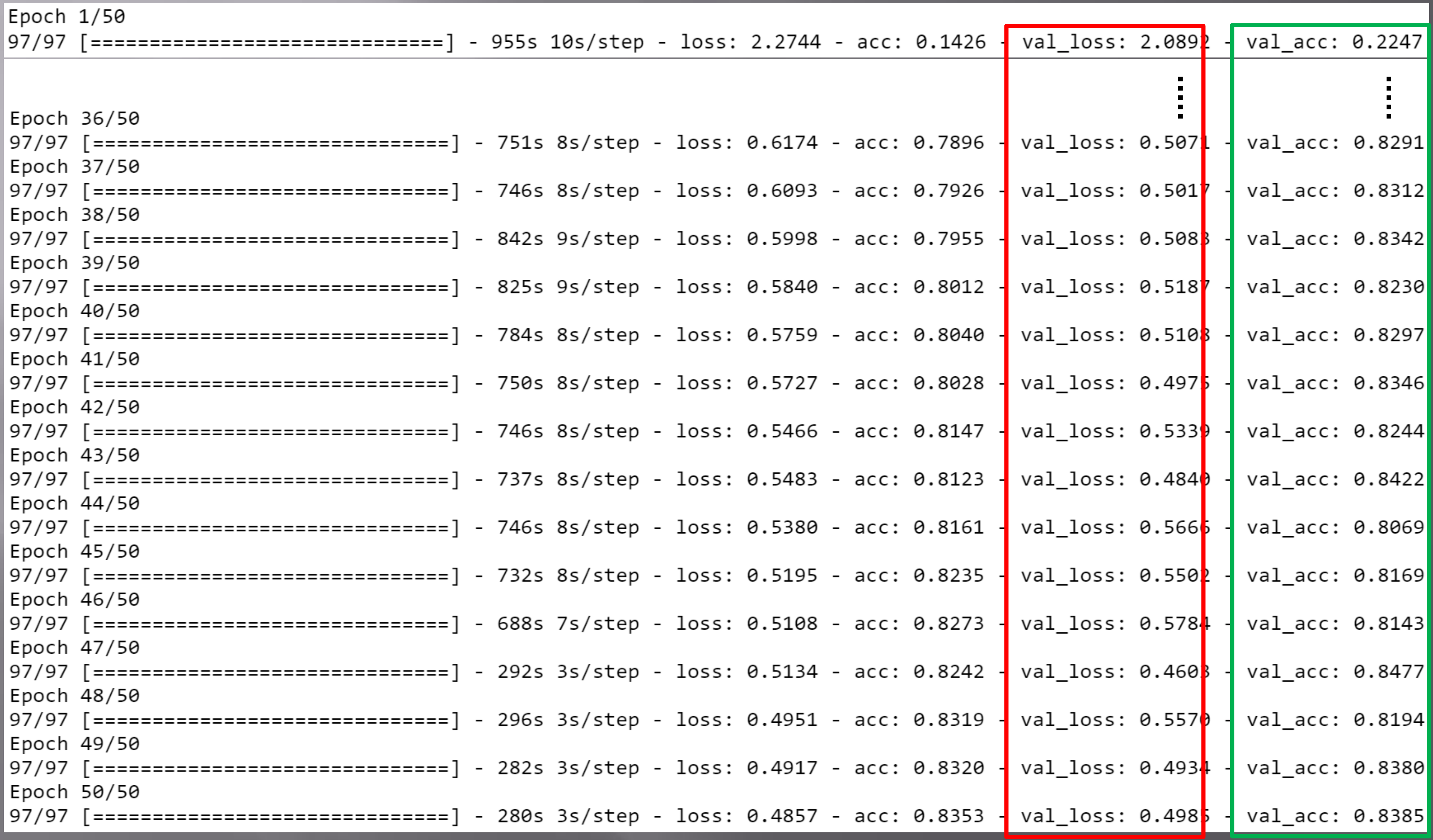


True \ Predicted	0	1	2	3	4	5	6	7	8	9
9	174	158	0	9	0	10	0	2	337	310
8	400	101	0	4	0	27	0	0	366	102
7	254	203	0	71	0	43	0	24	117	288
6	177	204	0	295	0	112	4	35	95	78
5	338	221	0	121	0	127	0	21	103	69
4	245	157	0	241	0	101	4	17	136	99
3	335	202	0	140	0	153	0	10	75	85
2	363	233	0	108	0	109	0	10	105	72
1	159	483	0	8	0	18	0	2	194	136
0	617	94	0	3	0	10	0	0	183	93

We usually train such networks for min. a few dozens of epochs to get satisfying results ...

CIFAR-10 Classification in Jupyter

Let's train the network longer (50 epochs, a few hours) and as you can see the error (val_loss) systematically decreases, and the accuracy (val_acc) increases:

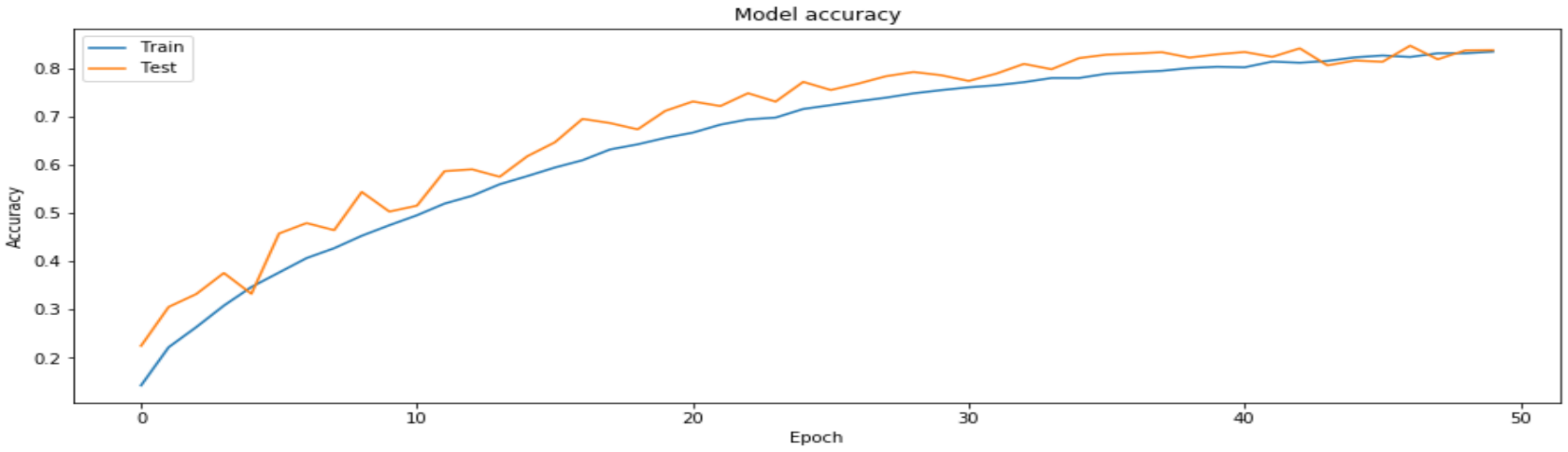


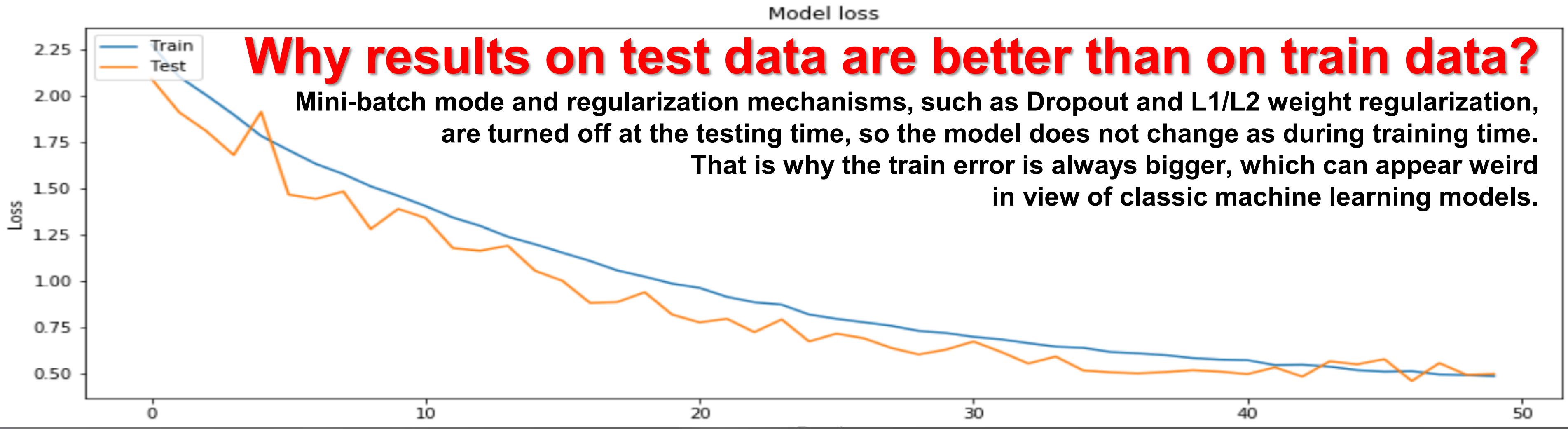
```
Epoch 1/50
97/97 [==============================] - 955s 10s/step - loss: 2.2744 - acc: 0.1426 - val_loss: 2.0892 - val_acc: 0.2247

                                                                                                       ⋮                  ⋮

Epoch 36/50
97/97 [==============================] - 751s 8s/step - loss: 0.6174 - acc: 0.7896 - val_loss: 0.5071 - val_acc: 0.8291
Epoch 37/50
97/97 [==============================] - 746s 8s/step - loss: 0.6093 - acc: 0.7926 - val_loss: 0.5017 - val_acc: 0.8312
Epoch 38/50
97/97 [==============================] - 842s 9s/step - loss: 0.5998 - acc: 0.7955 - val_loss: 0.5083 - val_acc: 0.8342
Epoch 39/50
97/97 [==============================] - 825s 9s/step - loss: 0.5840 - acc: 0.8012 - val_loss: 0.5187 - val_acc: 0.8230
Epoch 40/50
97/97 [==============================] - 784s 8s/step - loss: 0.5759 - acc: 0.8040 - val_loss: 0.5108 - val_acc: 0.8297
Epoch 41/50
97/97 [==============================] - 750s 8s/step - loss: 0.5727 - acc: 0.8028 - val_loss: 0.4975 - val_acc: 0.8346
Epoch 42/50
97/97 [==============================] - 746s 8s/step - loss: 0.5466 - acc: 0.8147 - val_loss: 0.5339 - val_acc: 0.8244
Epoch 43/50
97/97 [==============================] - 737s 8s/step - loss: 0.5483 - acc: 0.8123 - val_loss: 0.4840 - val_acc: 0.8422
Epoch 44/50
97/97 [==============================] - 746s 8s/step - loss: 0.5380 - acc: 0.8161 - val_loss: 0.5666 - val_acc: 0.8069
Epoch 45/50
97/97 [==============================] - 732s 8s/step - loss: 0.5195 - acc: 0.8235 - val_loss: 0.5502 - val_acc: 0.8169
Epoch 46/50
97/97 [==============================] - 688s 7s/step - loss: 0.5108 - acc: 0.8273 - val_loss: 0.5784 - val_acc: 0.8143
Epoch 47/50
97/97 [==============================] - 292s 3s/step - loss: 0.5134 - acc: 0.8242 - val_loss: 0.4603 - val_acc: 0.8477
Epoch 48/50
97/97 [==============================] - 296s 3s/step - loss: 0.4951 - acc: 0.8319 - val_loss: 0.5570 - val_acc: 0.8194
Epoch 49/50
97/97 [==============================] - 282s 3s/step - loss: 0.4917 - acc: 0.8320 - val_loss: 0.4934 - val_acc: 0.8380
Epoch 50/50
97/97 [==============================] - 280s 3s/step - loss: 0.4857 - acc: 0.8353 - val_loss: 0.4985 - val_acc: 0.8385
```

CIFAR-10 Classification in Jupyter

The graphs also show this convergence process:

```
Test loss: 0.4984995872974396
Test accuracy: 0.8385000228881836
```



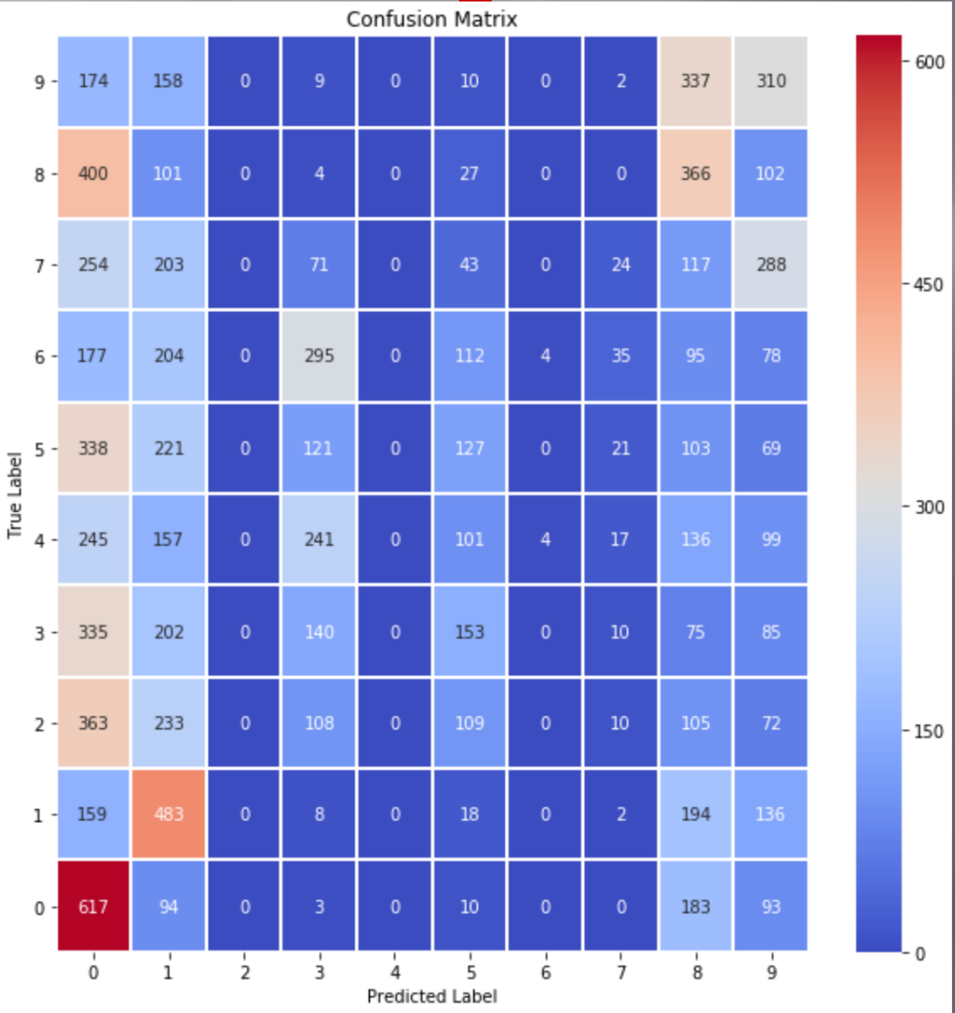


Why results on test data are better than on train data?

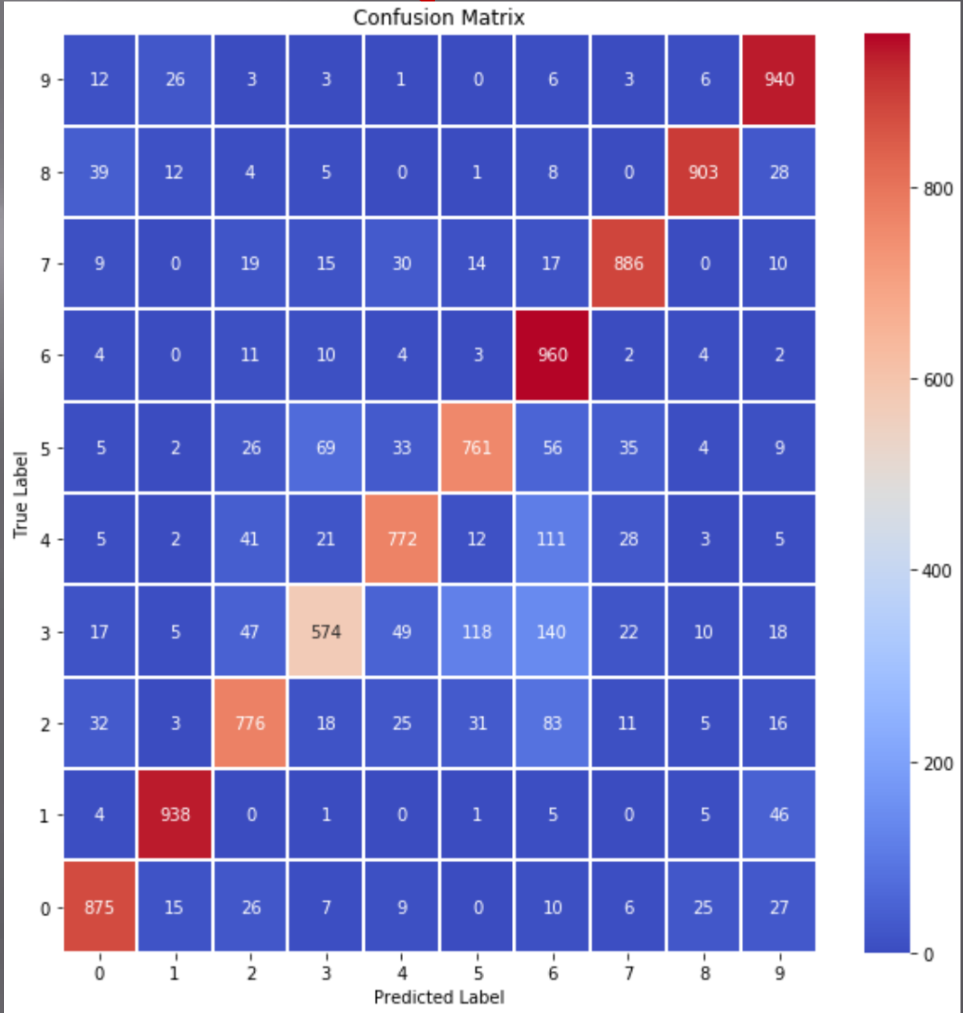
Mini-batch mode and regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at the testing time, so the model does not change as during training time. That is why the train error is always bigger, which can appear weird in view of classic machine learning models.

CIFAR-10 Classification in Jupyter

The confusion matrix has also improved: more examples migrate towards the diagonal (correct classifications) from other regions:



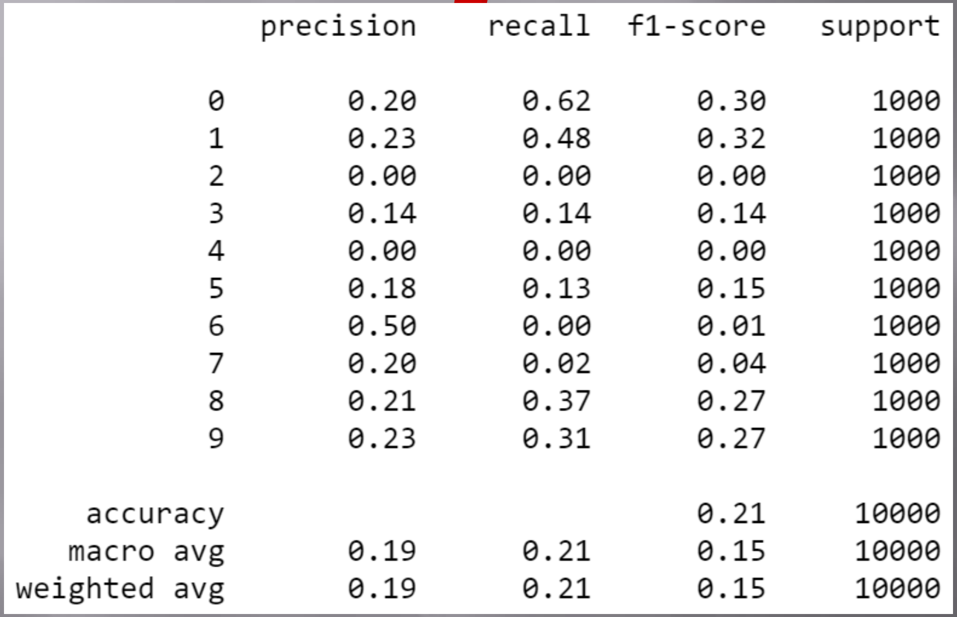
True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	174	158	0	9	0	10	0	2	337	310
8	400	101	0	4	0	27	0	0	366	102
7	254	203	0	71	0	43	0	24	117	288
6	177	204	0	295	0	112	4	35	95	78
5	338	221	0	121	0	127	0	21	103	69
4	245	157	0	241	0	101	4	17	136	99
3	335	202	0	140	0	153	0	10	75	85
2	363	233	0	108	0	109	0	10	105	72
1	159	483	0	8	0	18	0	2	194	136
0	617	94	0	3	0	10	0	0	183	93



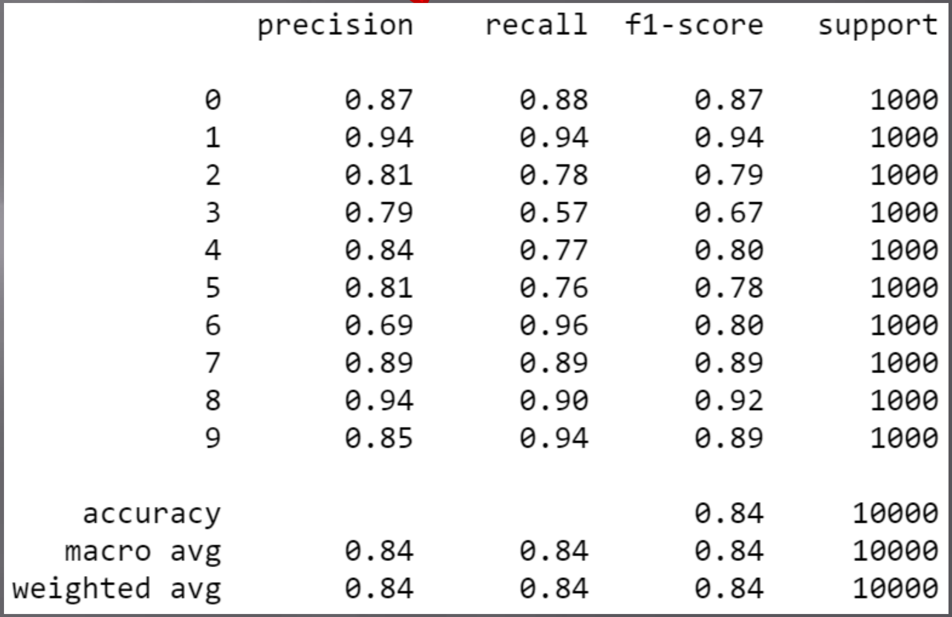
True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	12	26	3	3	1	0	6	3	6	940
8	39	12	4	5	0	1	8	0	903	28
7	9	0	19	15	30	14	17	886	0	10
6	4	0	11	10	4	3	960	2	4	2
5	5	2	26	69	33	761	56	35	4	9
4	5	2	41	21	772	12	111	28	3	5
3	17	5	47	574	49	118	140	22	10	18
2	32	3	776	18	25	31	83	11	5	16
1	4	938	0	1	0	1	5	0	5	46
0	875	15	26	7	9	0	10	6	25	27

CIFAR-10 Classification in Jupyter

The number and the accuracy of correctly classified examples for all individual classes increase:



```
              precision    recall  f1-score   support

           0       0.20      0.62      0.30      1000
           1       0.23      0.48      0.32      1000
           2       0.00      0.00      0.00      1000
           3       0.14      0.14      0.14      1000
           4       0.00      0.00      0.00      1000
           5       0.18      0.13      0.15      1000
           6       0.50      0.00      0.01      1000
           7       0.20      0.02      0.04      1000
           8       0.21      0.37      0.27      1000
           9       0.23      0.31      0.27      1000

    accuracy                           0.21     10000
   macro avg       0.19      0.21      0.15     10000
weighted avg       0.19      0.21      0.15     10000
```

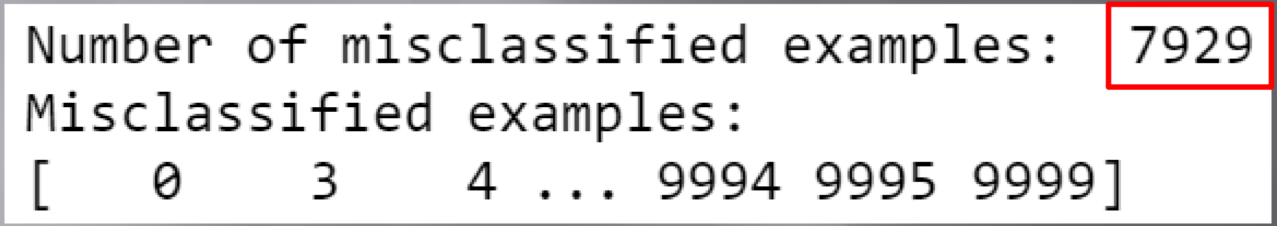


```
              precision    recall  f1-score   support

           0       0.87      0.88      0.87      1000
           1       0.94      0.94      0.94      1000
           2       0.81      0.78      0.79      1000
           3       0.79      0.57      0.67      1000
           4       0.84      0.77      0.80      1000
           5       0.81      0.76      0.78      1000
           6       0.69      0.96      0.80      1000
           7       0.89      0.89      0.89      1000
           8       0.94      0.90      0.92      1000
           9       0.85      0.94      0.89      1000

    accuracy                           0.84     10000
   macro avg       0.84      0.84      0.84     10000
weighted avg       0.84      0.84      0.84     10000
```

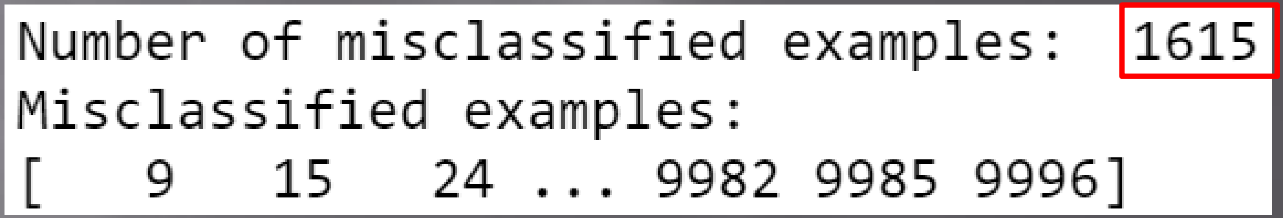
However, we can see that the process of network training is not over yet and should be continued for several dozen epochs.

CIFAR-10 Classification in Jupyter

Examples of misclassifications after 50 training epochs for a test set of 10,000 examples: The number of misclassifications decreased from 7929 after 3 epochs to 1615 after 50 epochs.



```
Number of misclassified examples:  7929
Misclassified examples:
[   0    3    4 ... 9994 9995 9999]
```

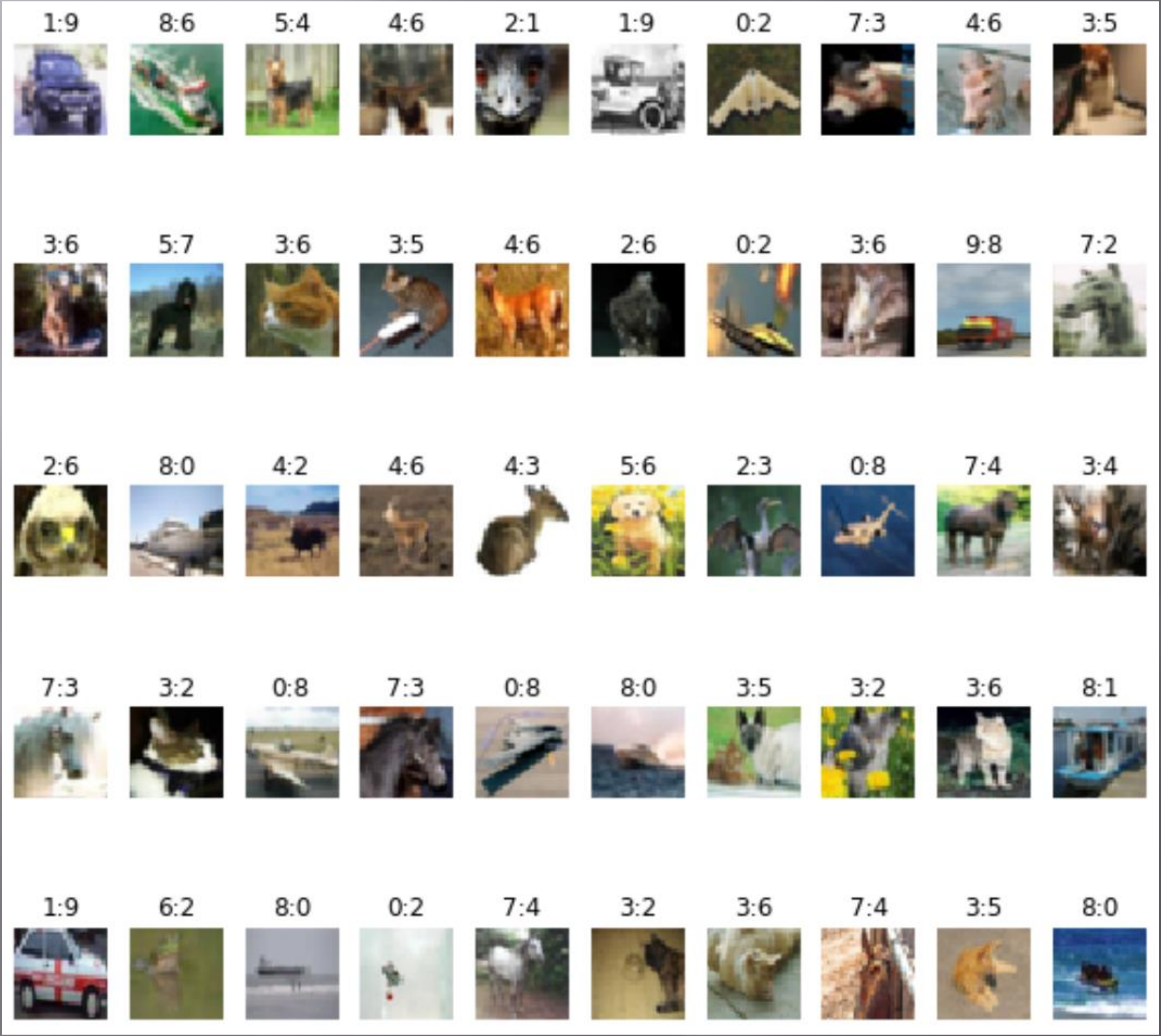


```
Number of misclassified examples:  1615
Misclassified examples:
[   9   15   24 ... 9982 9985 9996]
```

We can see that in the case of this training set, the convolution network should be taught much longer (16.15% of incorrect classifications remain) or the structure or the hyperparameters of the model should be changed.

CIFAR-10 Classification in Jupyter

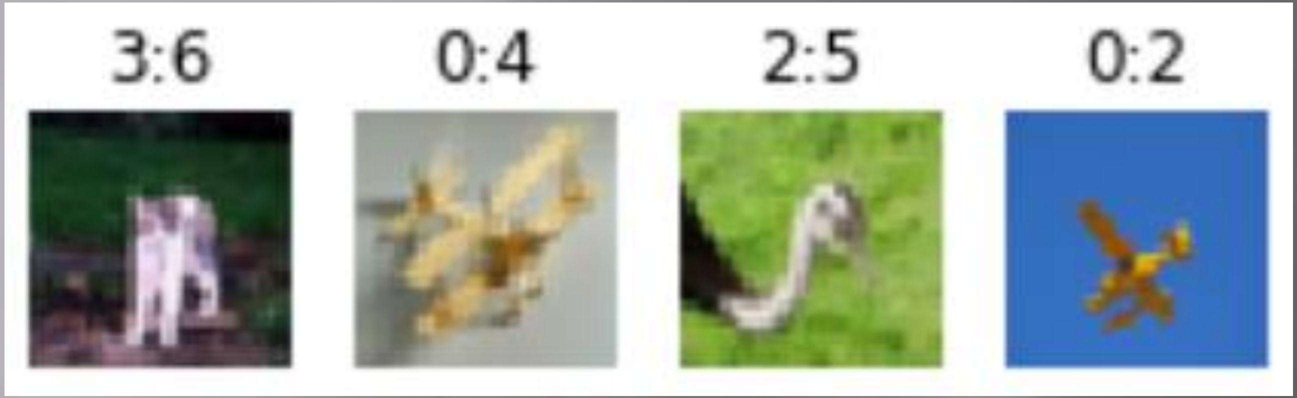
Sample misclassified examples:

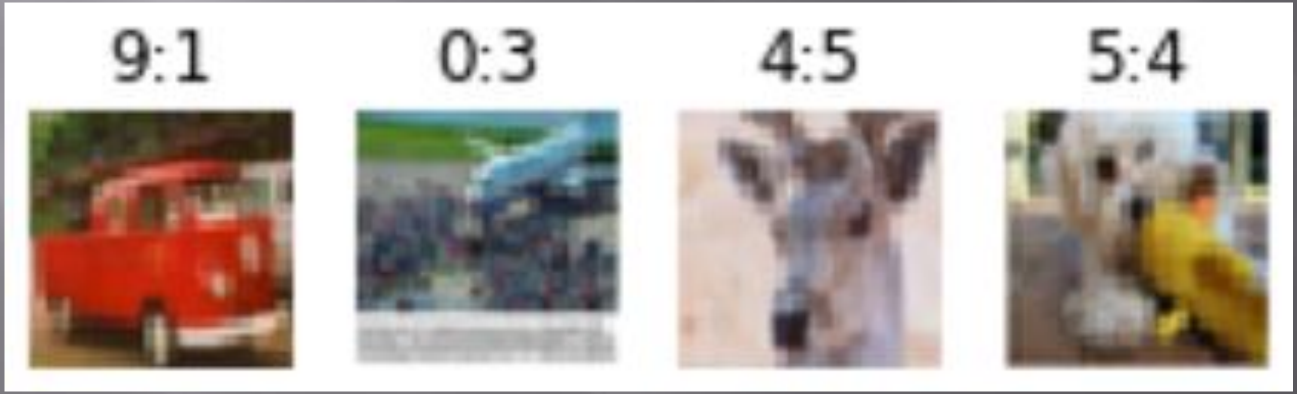


airplane	0
automobile	1
bird	2
cat	3
deer	4
dog	5
frog	6
horse	7
ship	8
truck	9

CIFAR-10 Classification in Jupyter

Sample misclassified examples:





airplane	0	
automobile	1	
bird	2	
cat	3	
deer	4	
dog	5	
frog	6	
horse	7	
ship	8	
truck	9	

CNN to Biomedical Image Classification

We will try to use CNNs to biomedical data, e.g. medical image classification.

Search for some free-available medical images and try to adapt the described model into them.

Let's start with powerful computations!

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

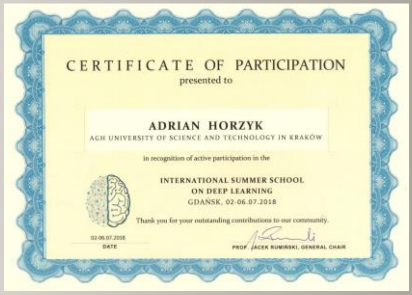
Bibliography and Literature

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/
10. https://www.youtube.com/watch?v=XNKeayZW4dY
11. https://victorzhou.com/blog/keras-cnn-tutorial/
12. https://github.com/keras-team/keras/tree/master/examples
13. https://medium.com/@margaretmz/anaconda-jupyter-notebook-tensorflow-and-keras-b91f381405f8
14. https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html
15. http://coursera.org/specializations/tensorflow-in-practice
16. https://udacity.com/course/intro-to-tensorflow-for-deep-learning
17. MNIST sample: https://medium.com/datadriveninvestor/image-processing-for-mnist-using-keras-f9a1021f6ef0
18. Heatmaps: https://towardsdatascience.com/formatting-tips-for-correlation-heatmaps-in-seaborn-4478ef15d87f

Adrian Horzyk

horzyk@agh.edu.pl

Google: Horzyk



CERTIFICATE OF PARTICIPATION
presented to

ADRIAN HORZYK
AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY IN KRAKÓW

in recognition of active participation in the

INTERNATIONAL SUMMER SCHOOL
ON DEEP LEARNING
GDAŃSK, 02-06.07.2018

Thank you for your outstanding contributions to our community.

02-06.07.2018
DATE

PROF. JACEK RUMIŃSKI, GENERAL CHAIR

AGH

University of Science and Technology in Krakow, Poland