COMPUTATIONAL INTELLIGENCE

Short Introduction to Objectives, Deep Learning, Frameworks and Notebooks

Adrian Horzyk

horzyk@agh.edu.pl

AGH

AGH University of Science and Technology Krakow, Poland

Short Introduction

What do you mean by:

- Intelligence
- Artificial Intelligence
- Computational Intelligence

Question

What would you like to learn during this course?

Scope

We will address the following topics:

- Models and methods of Computational Intelligence.
- Frameworks and notebooks for the development of Computational Intelligence (CI) solutions in Python.
- Tuning, regularization, optimization of models and learning strategies.
- Deep learning of classic, convolutional and recurrent models.
- Classification, detection, regression, and clustering.
- Associative and knowledge-based systems and memories.

Notebooks and Frameworks

Popular Notebooks:

Jupyter Notebook

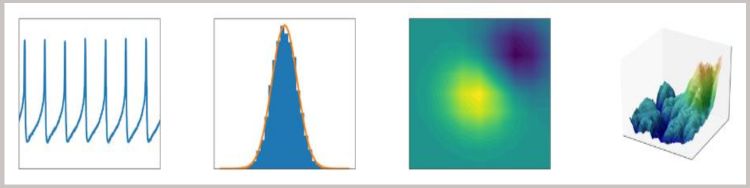
Google Colab

Popular frameworks:

- Tensorflow 2.0
- Keras
- PyTorch
- Caffe2
- Theano
- Lasagne
- DL4J (deep learning for Java)
- CNTK (MS Cognitive Toolkit)
- PaddlePaddle

Popular libraries:

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



Jupyter Notebook

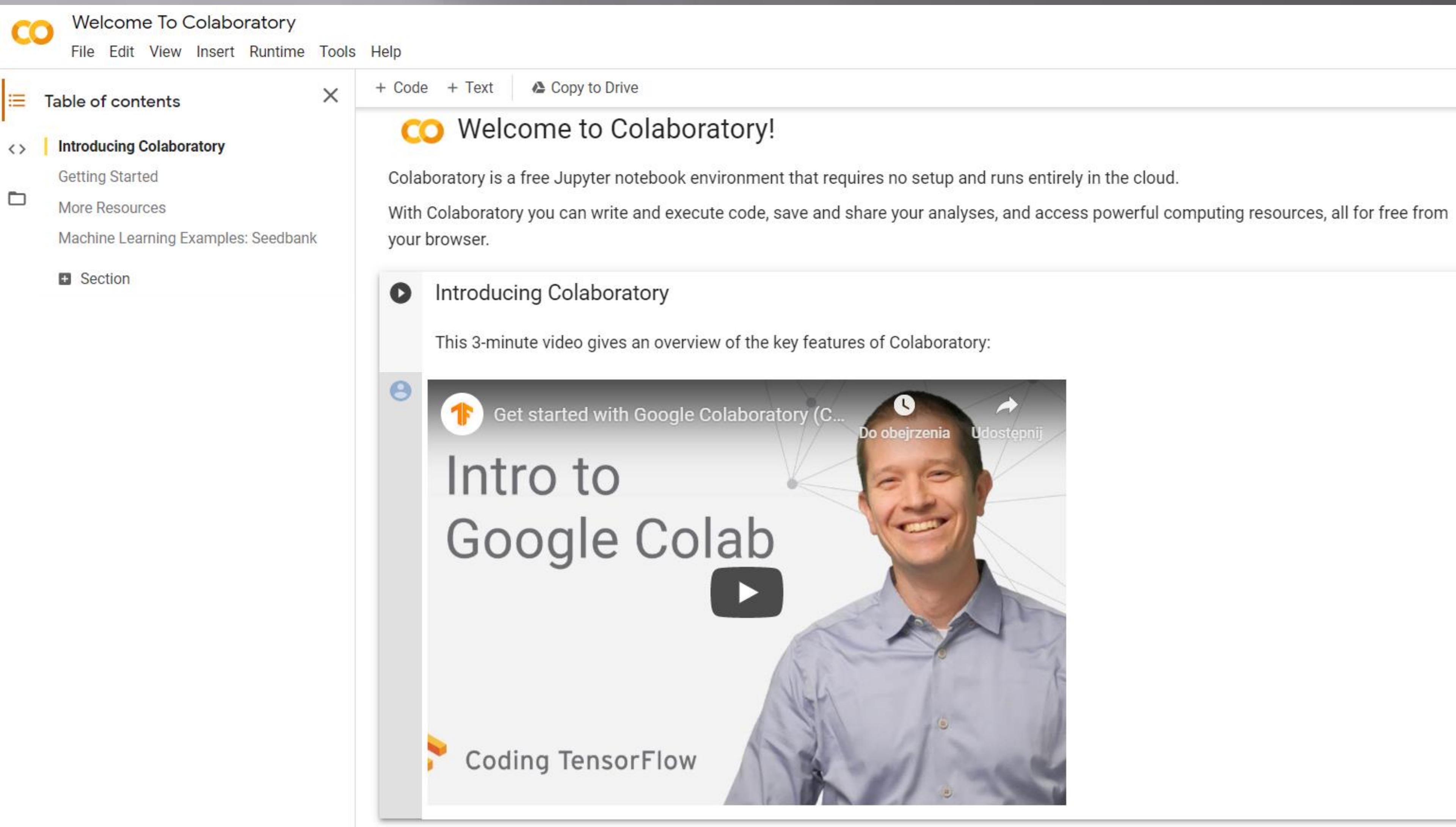
Jupyter is open, free and very popular:

jupyter Install About Us Community Documentation NBViewer JupyterHub Widgets Blog

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):



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

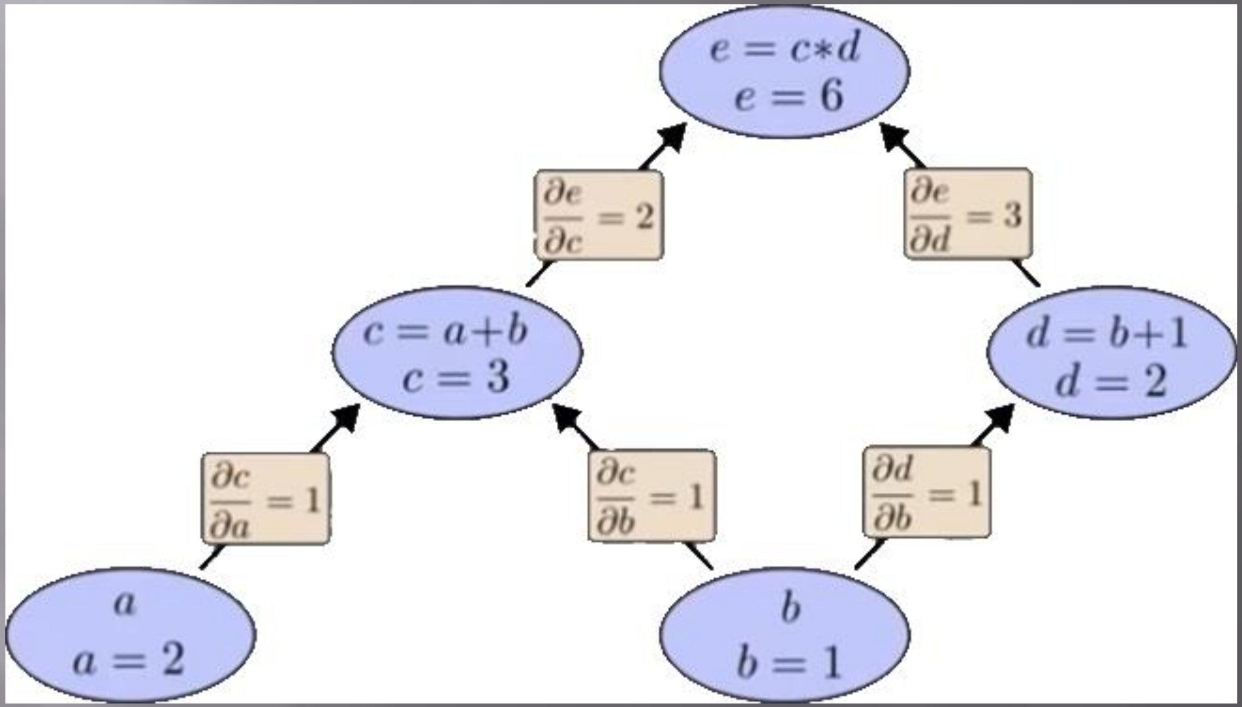
KERAS POSITIVE USER EXPERIENCE

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:

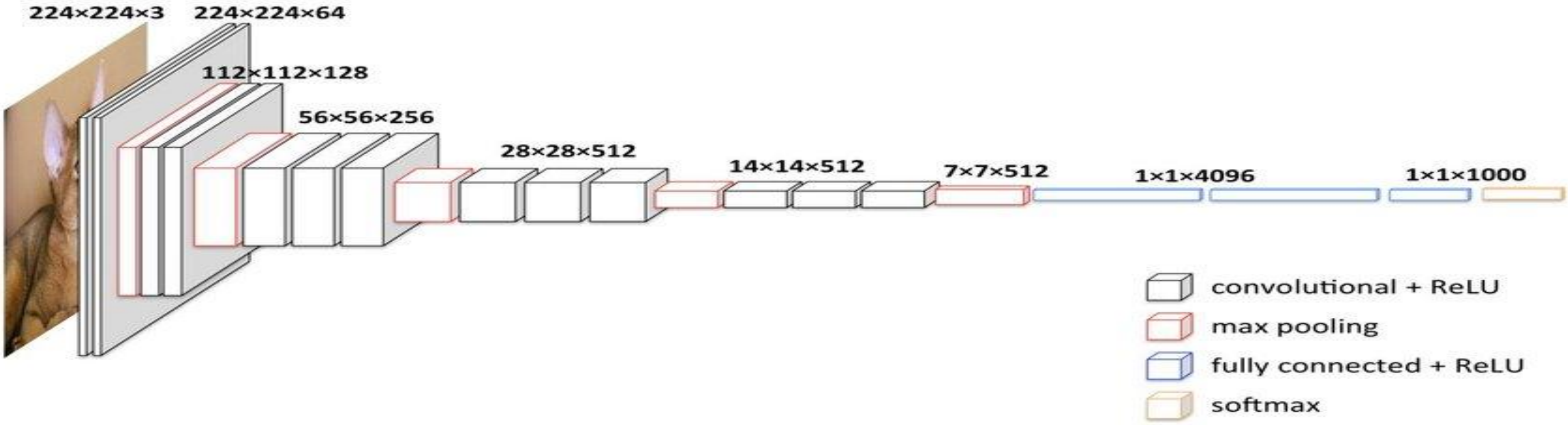


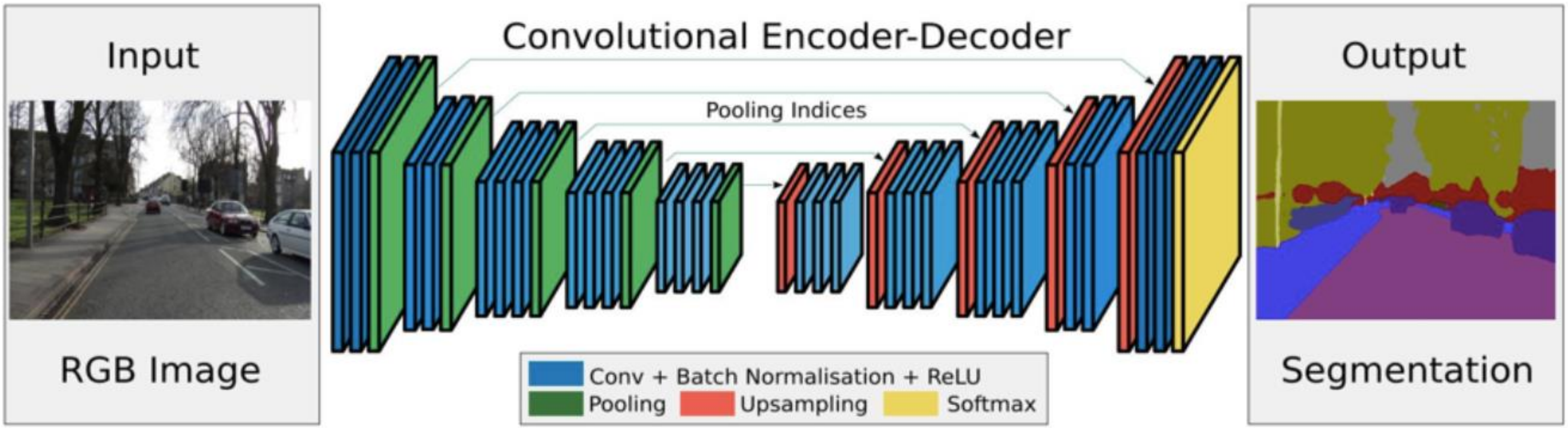
Where “a” and “b” are inputs used to compute “e” as an output using intermediate variables “c” and “d”.

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:

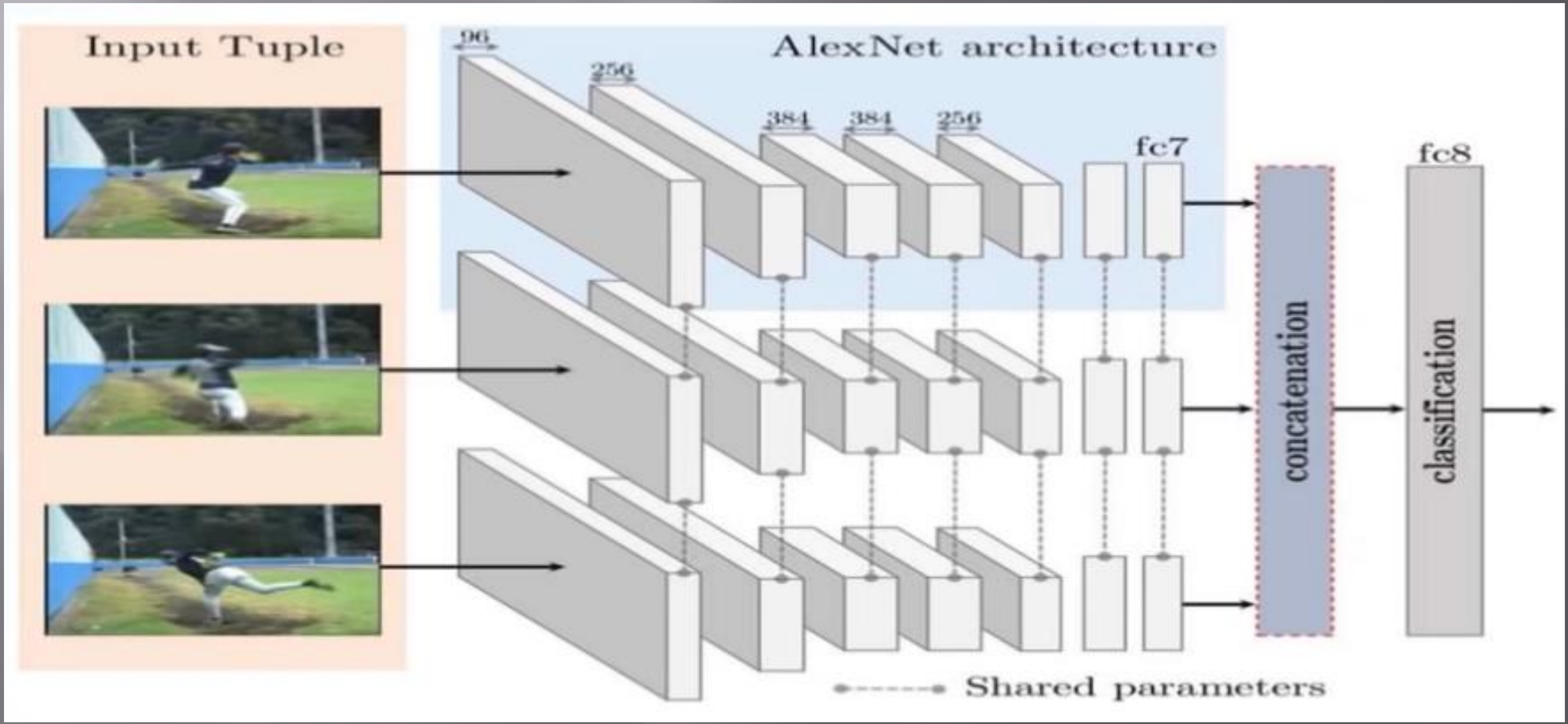




KERAS FUNCTIONAL MODELS

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

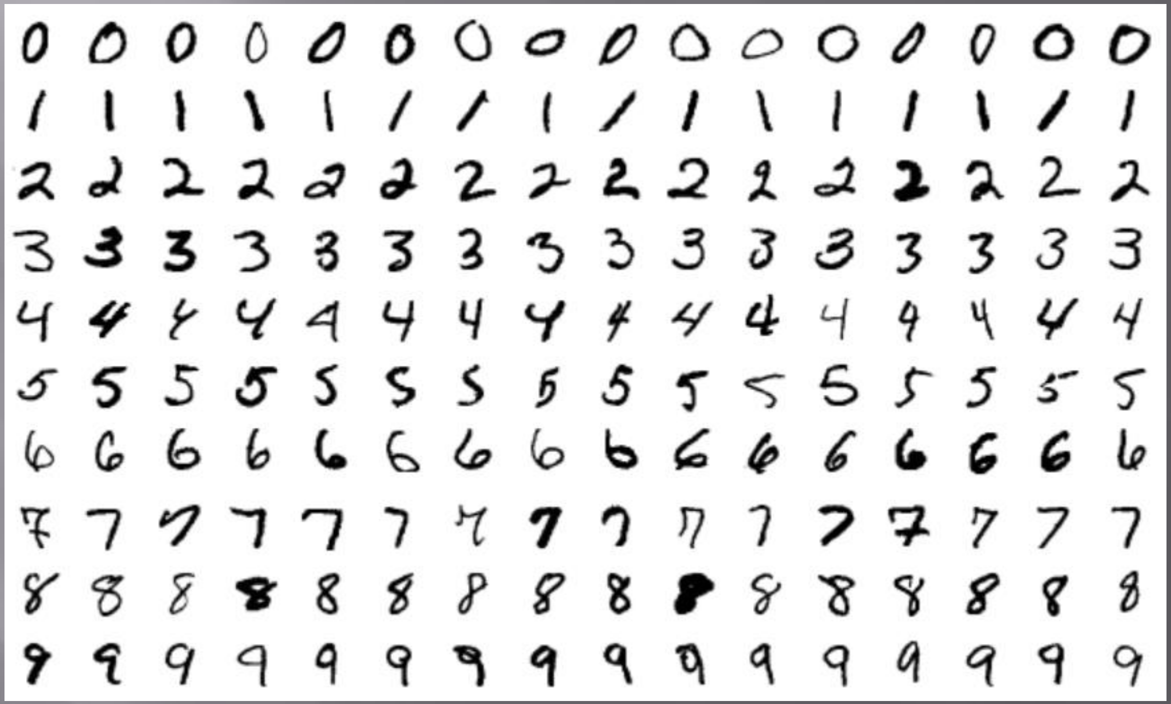
If successfully installed check in Jupyter Notebook the version of the TensorFlow using:

```
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

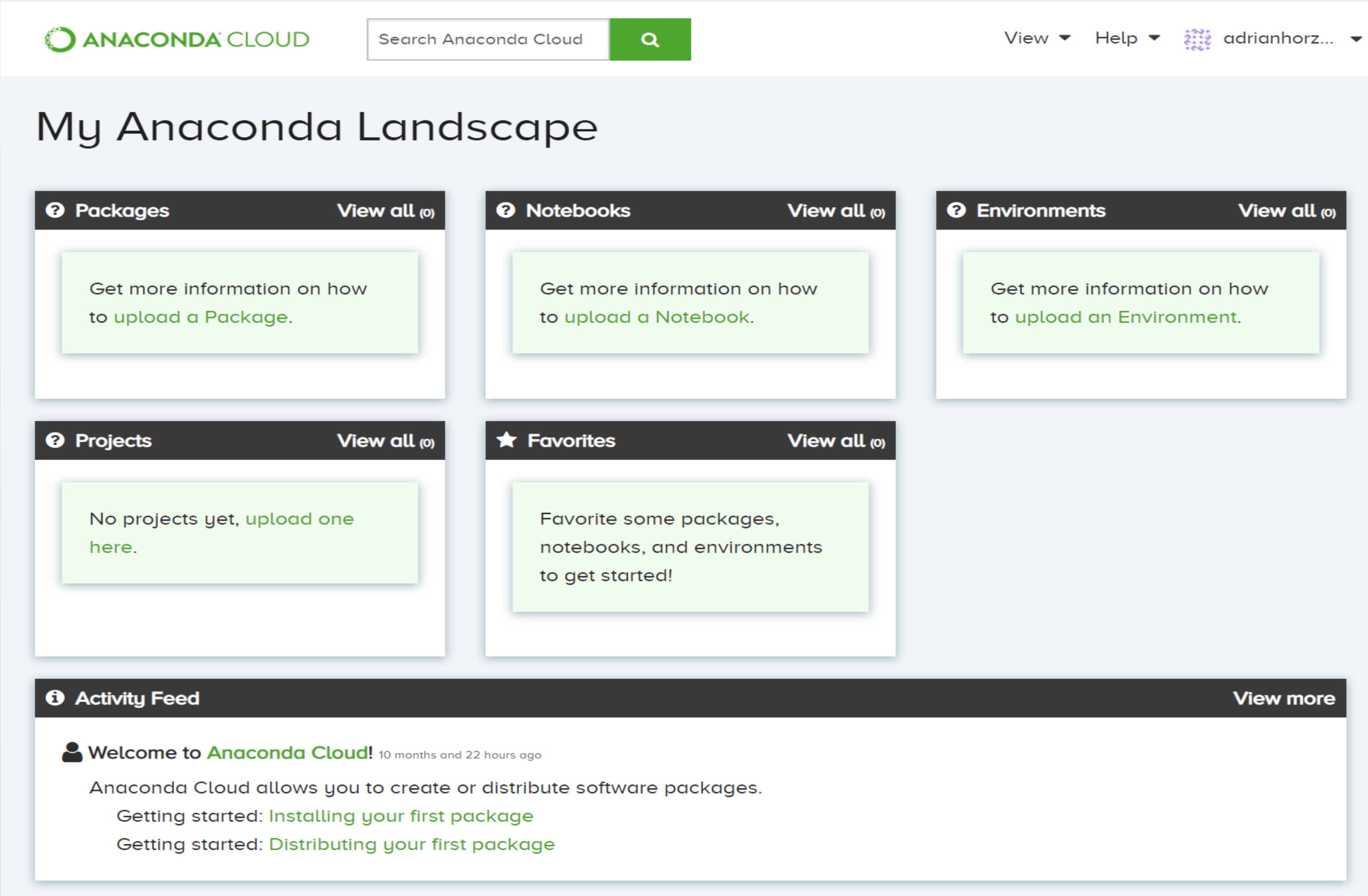
Install Jupyter using Anaconda with built in Python 3.7+

- 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.

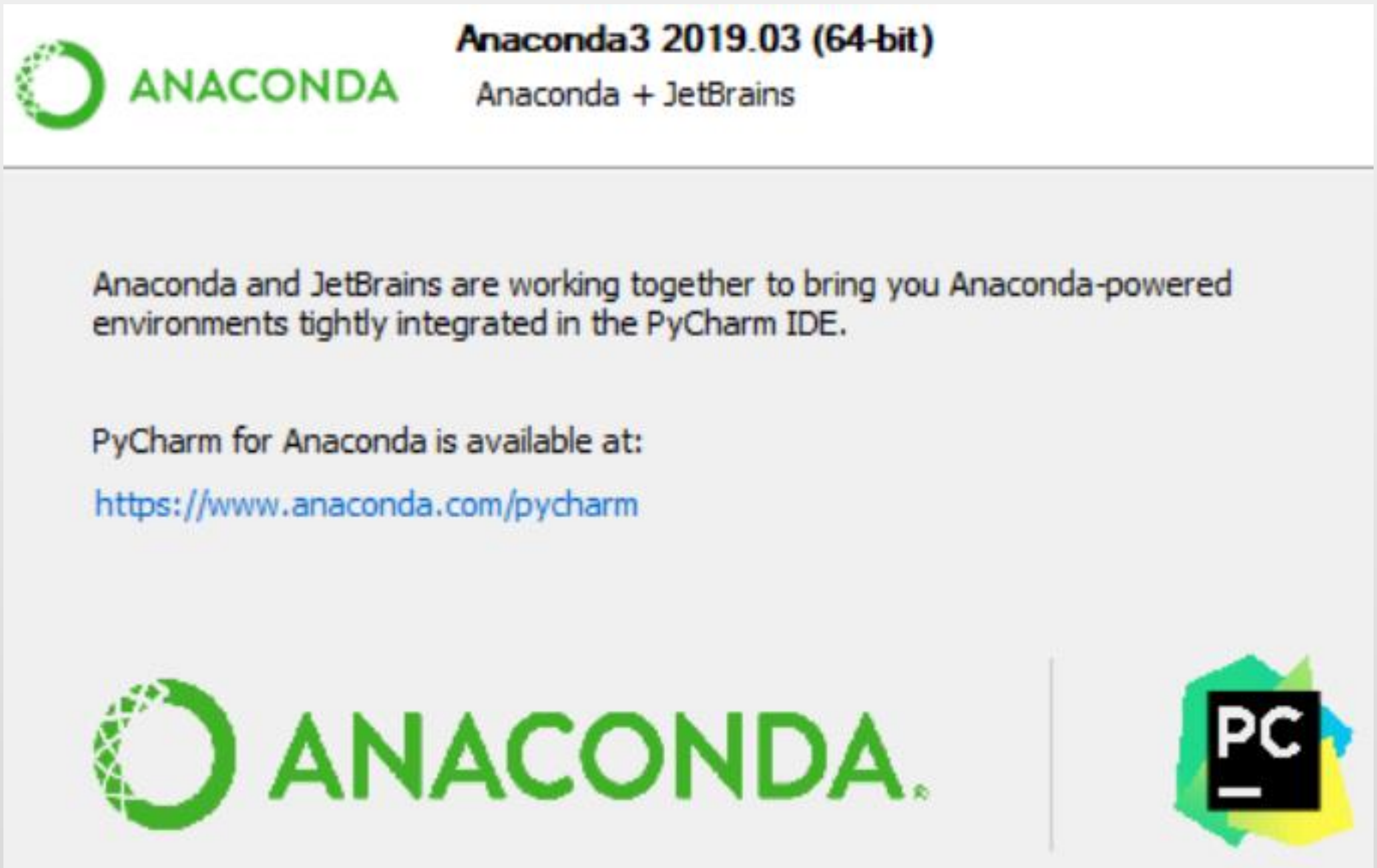
And run it at the Terminal (Mac/Linux) or Command Prompt (Windows):

```
jupyter notebook
```

Anaconda Cloud



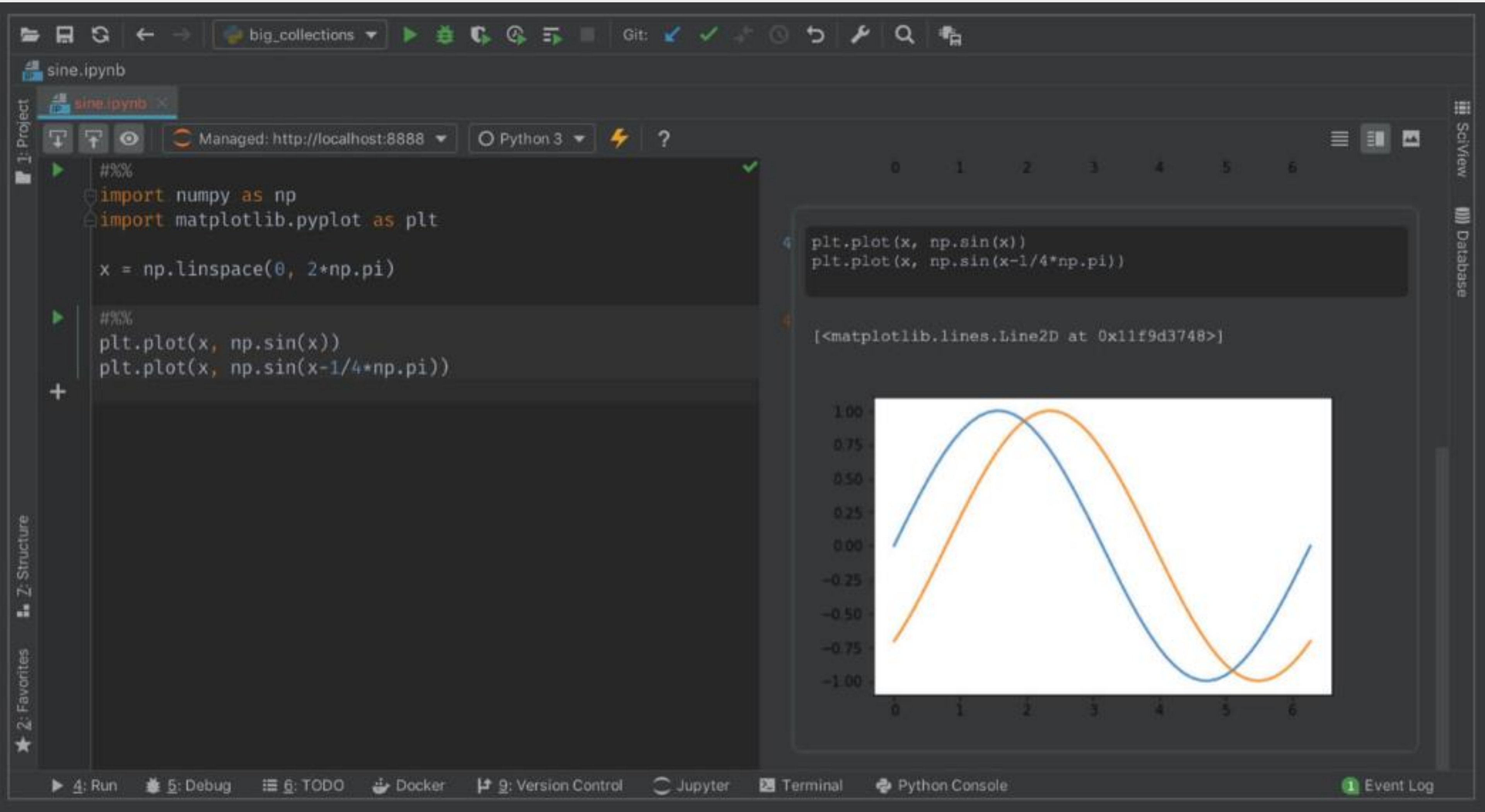
It is recommended to install PyCharm for Anaconda:



Jupyter Notebook

PyCharm is a python IDE for Professional Developers

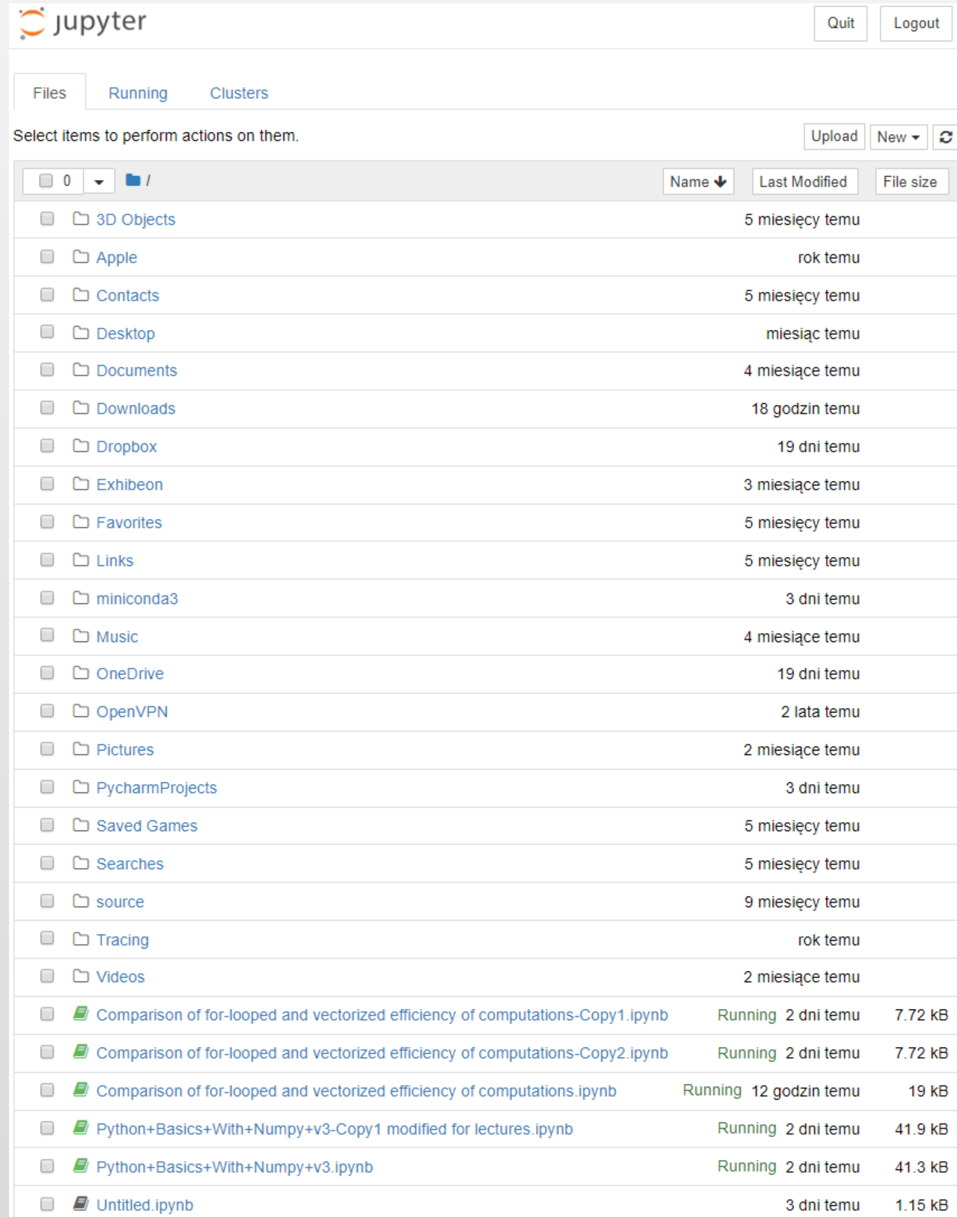
- It includes scientific mode to interactively analyze your data.



Jupyter Notebook Dashboard

Running a Jupyter Notebook in your browser:

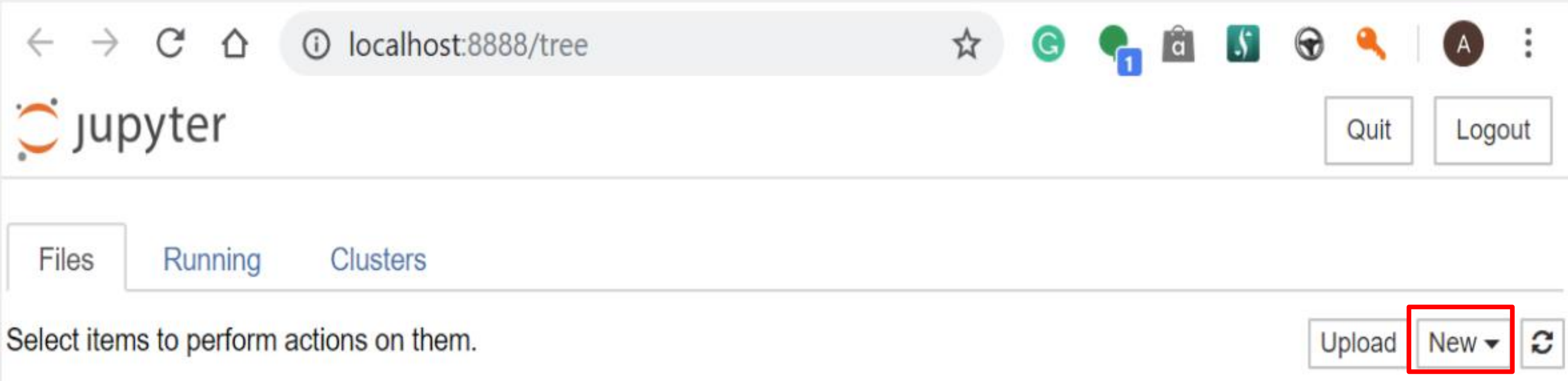
- When the Jupyter Notebook opens in your browser, you will see the Jupyter Notebook Dashboard, which will show you a list of the notebooks, files, and subdirectories in the directory where the notebook server was started by the command line „jupyter notebook”.
- Most of the time, you will wish to start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



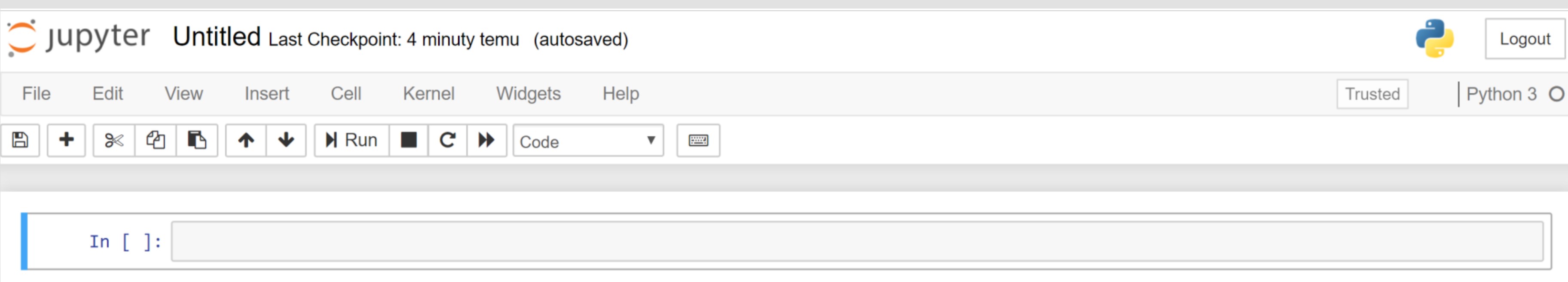
Starting a new Python notebook

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]:

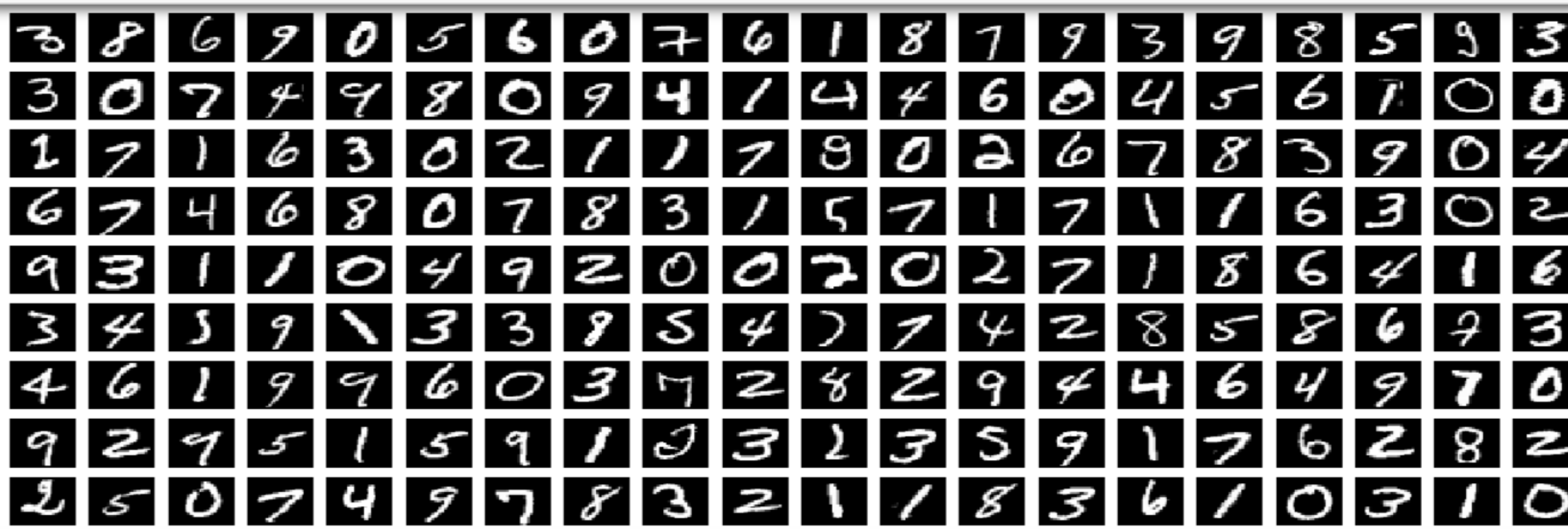
```
# Define hyperparameters
batch_size = 512   # size of mini-baches
num_classes = 10   # number of classes/digits: 0, 1, 2, ..., 9
epochs = 3         # how many times all traing examples will be used to train the model

# 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 example
```

MNIST Classification in Jupyter Notebook

Sample training examples from MNIST set (handwritten digits):

In [4]:

```python
# 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 c
```

```
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, in
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/#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
                                            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])
```

jupyter

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]:

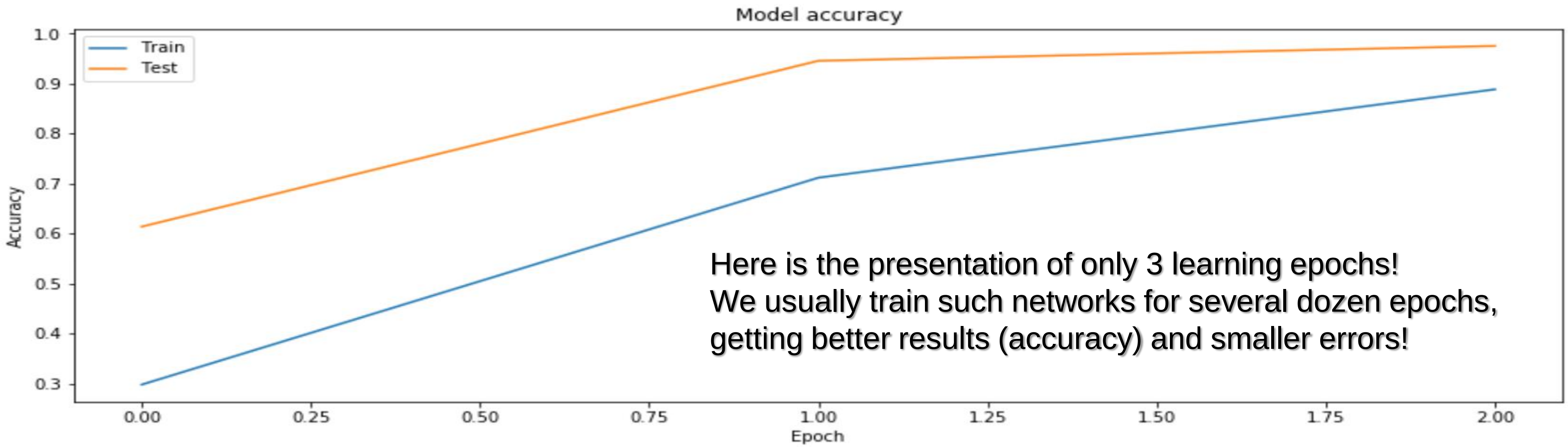
```python
# Evaluate the model and print out the final scores for the test set
score = model.evaluate(x_test, y_test, verbose=0)  # evaluate the model on the test set
print('Test loss:', score[0])      # print out the loss = score[0] (generalization error)
print('Test accuracy:', score[1])  # print out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

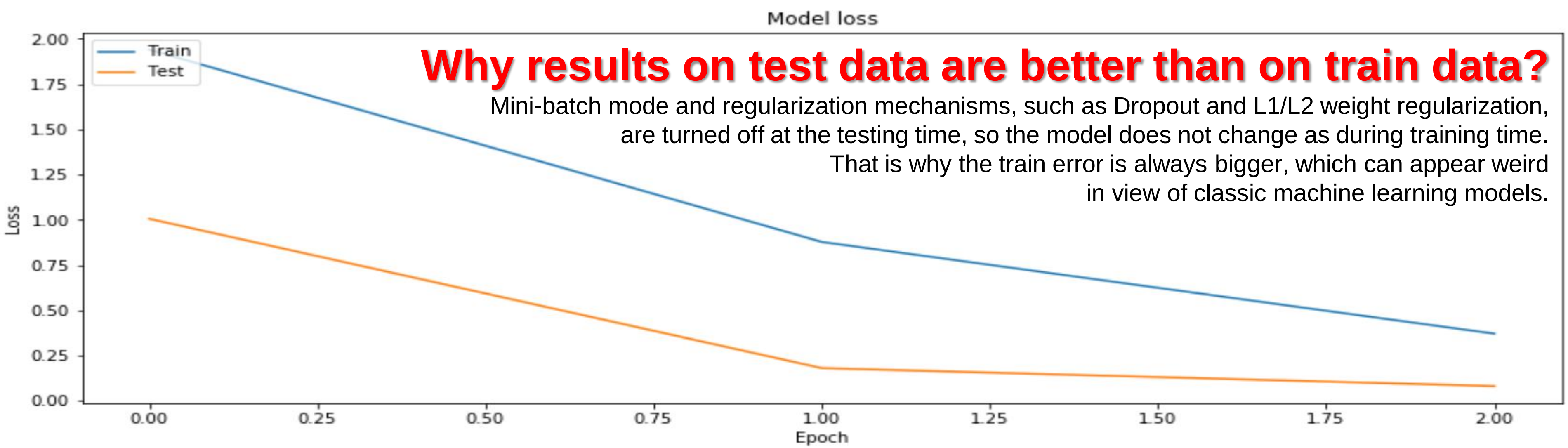
MNIST Classification in Jupyter Notebook

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

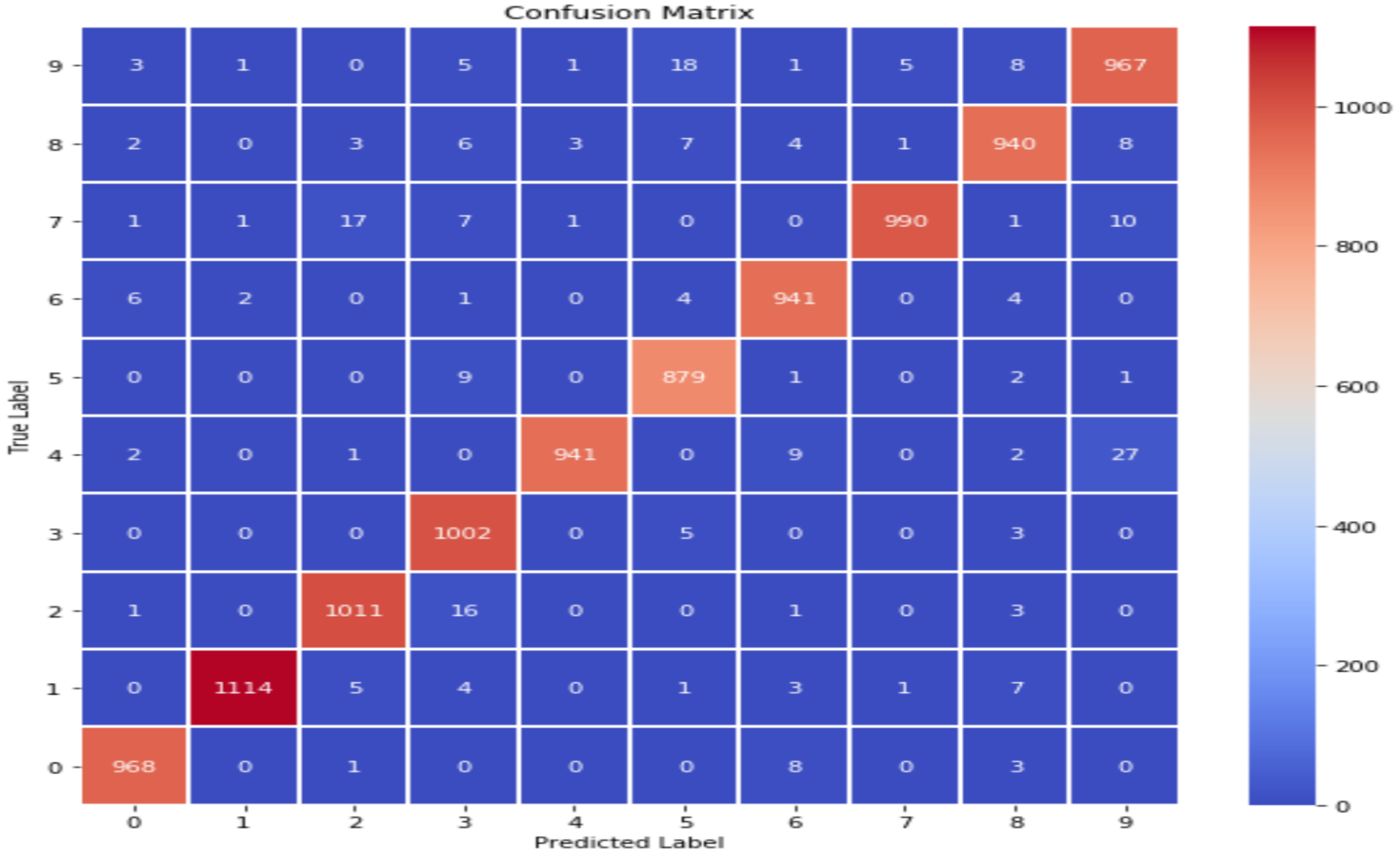
In [11]:

```python
# 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:



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]
```

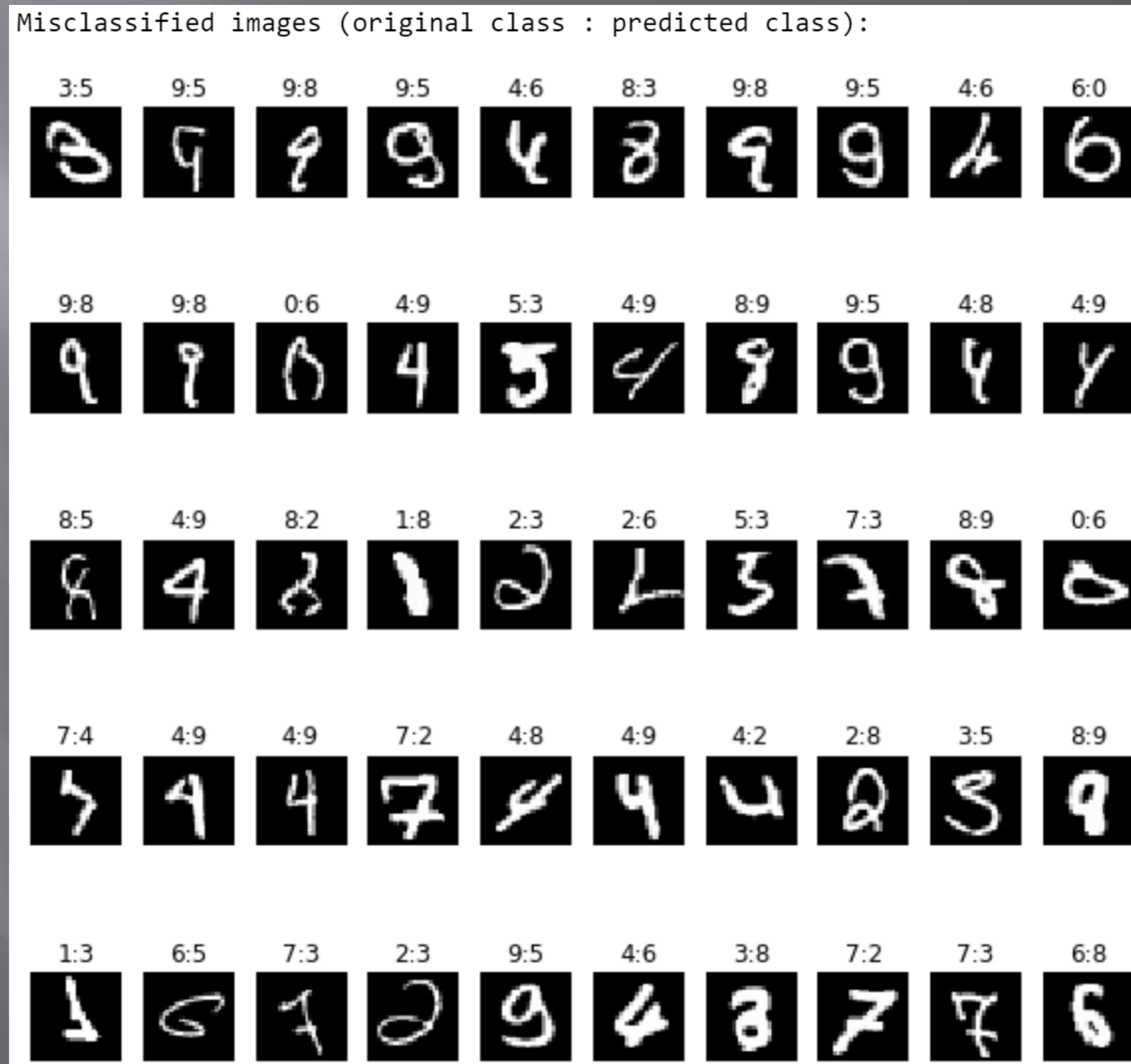
MNIST Classification in Jupyter Notebook

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!



MNIST Classification in Jupyter Notebook

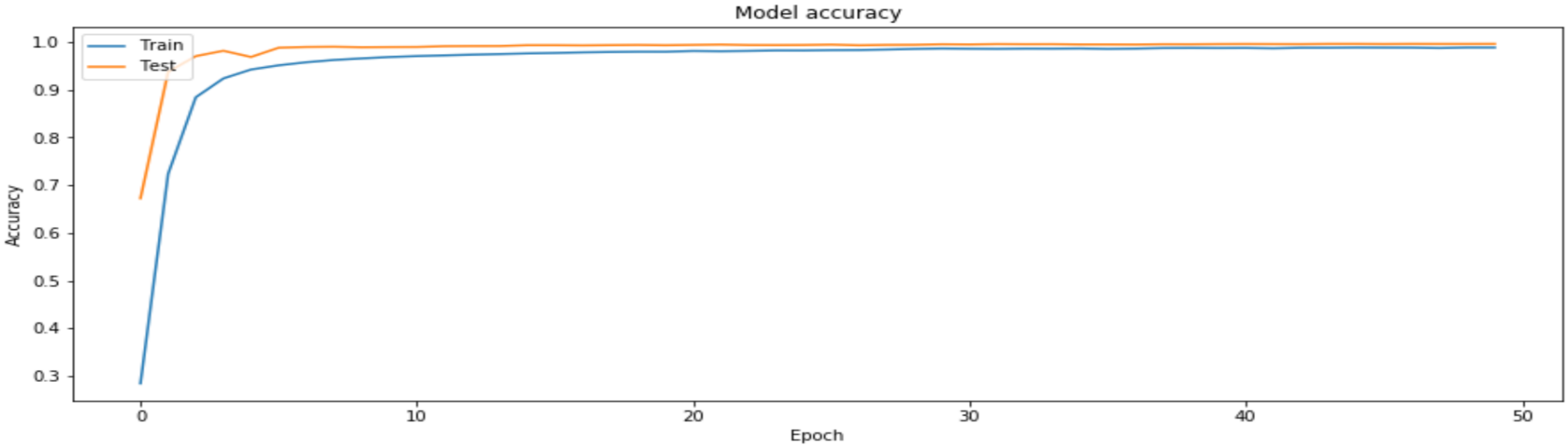
Now, let's try to train the network for 50 epochs:

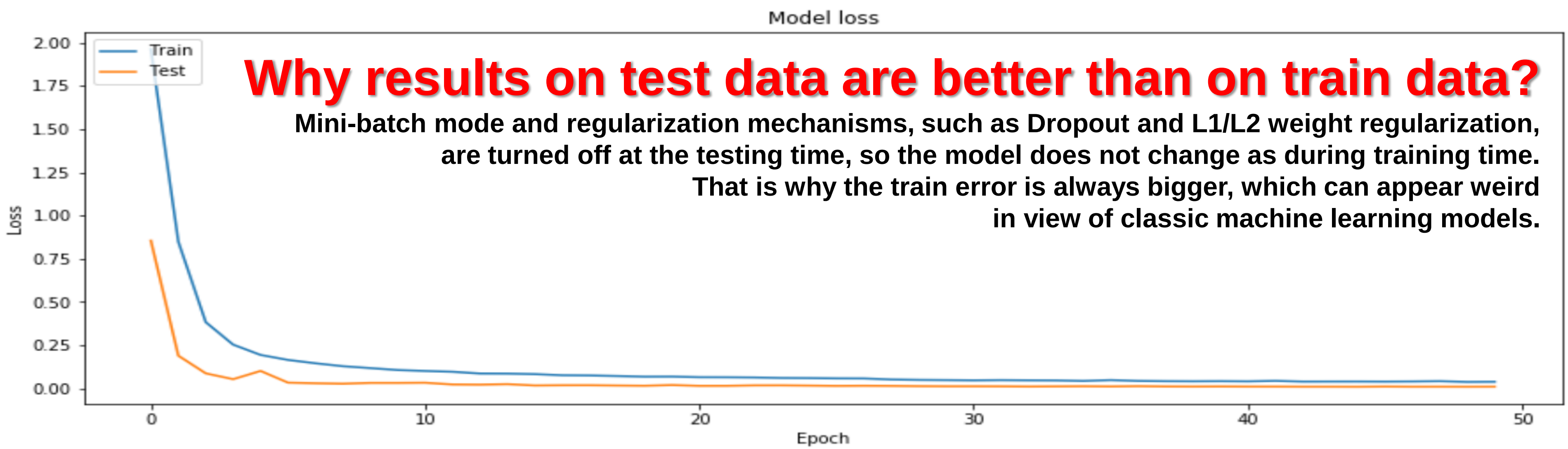
```
Epoch 1/50
117/117 [==============================] - 271s 2s/step - loss: 1.9644 - acc: 0.2841 - val_loss: 0.8554 - val_acc: 0.6723
Epoch 2/50
117/117 [==============================] - 270s 2s/step - loss: 0.8482 - acc: 0.7236 - val_loss: 0.1902 - val_acc: 0.9377
Epoch 3/50
117/117 [==============================] - 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
```



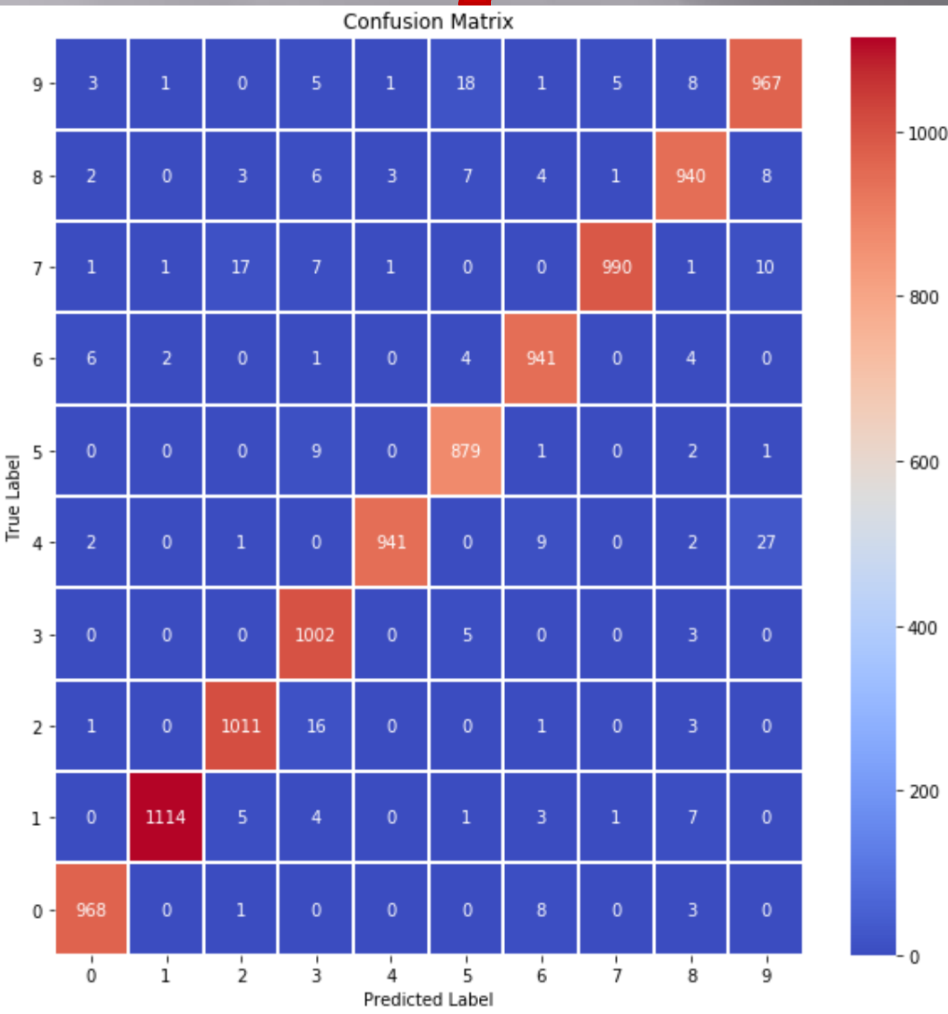


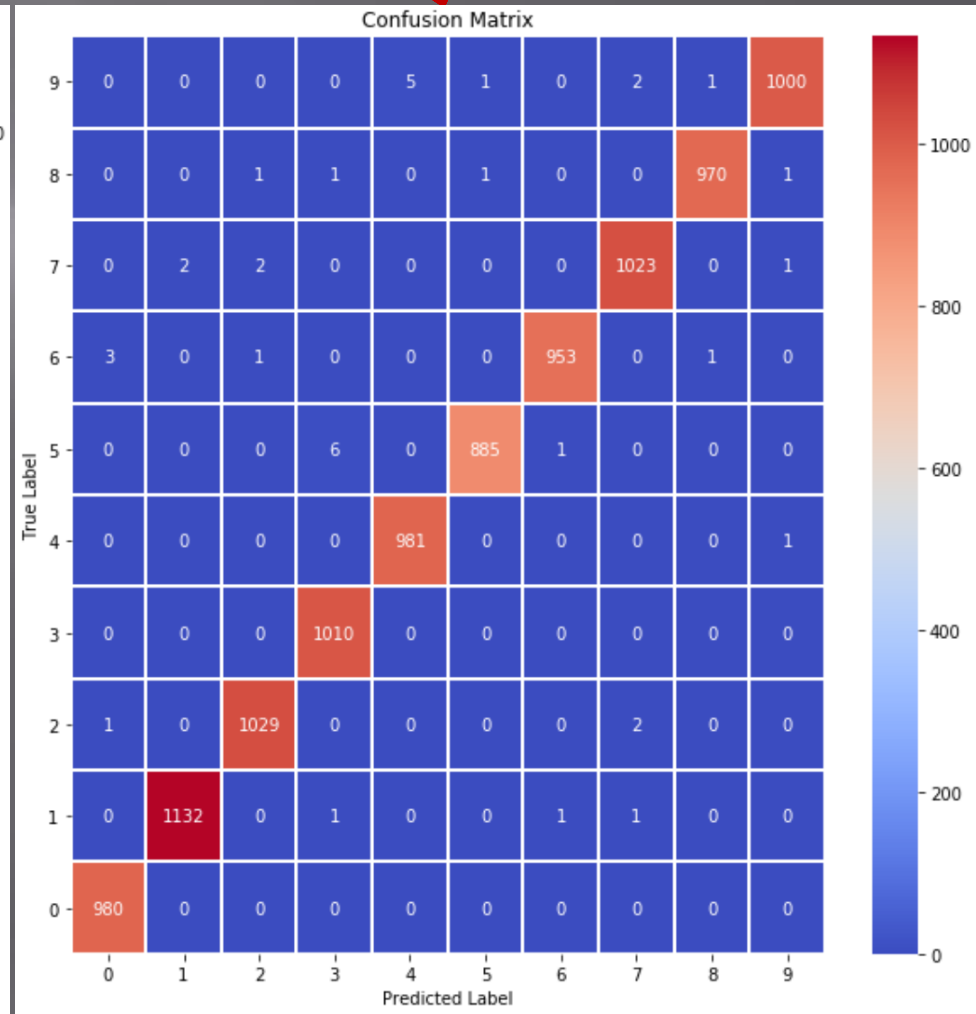
Why results on test data are better than on train data?

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:





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

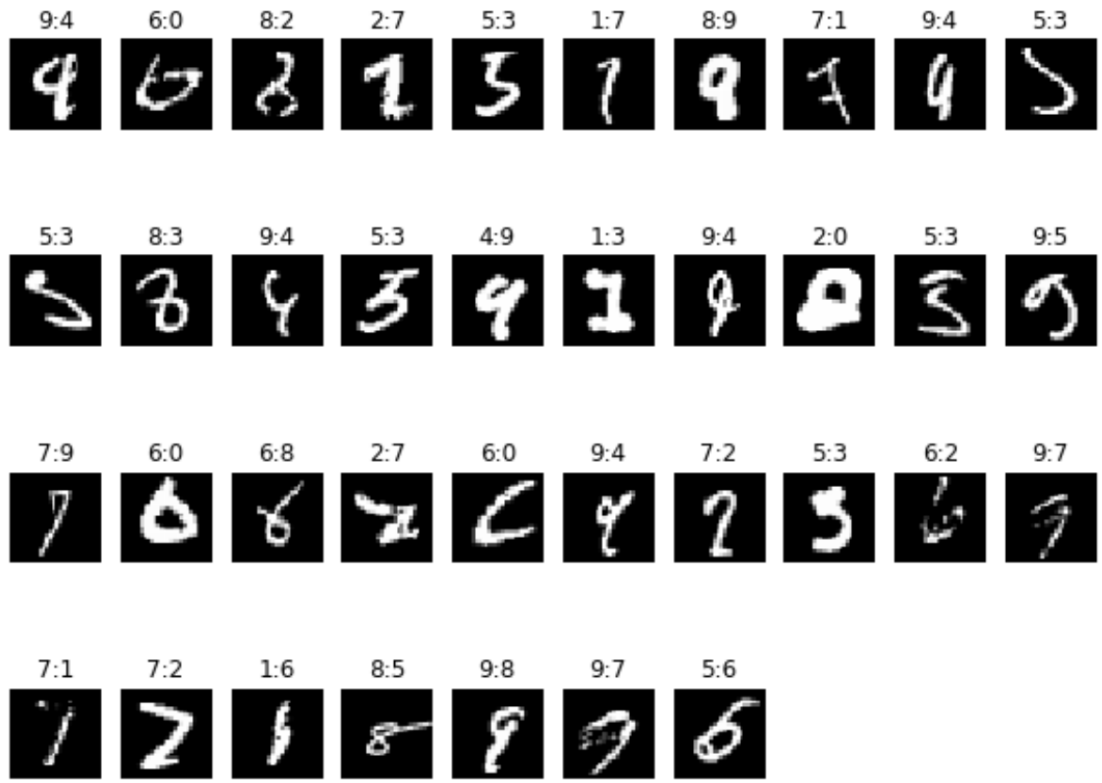
	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	1.00	1.00	1.00	1032
3	0.99	1.00	1.00	1010
4	0.99	1.00	1.00	982
5	1.00	0.99	0.99	892
6	1.00	0.99	1.00	958
7	1.00	1.00	1.00	1028
8	1.00	1.00	1.00	974
9	1.00	0.99	0.99	1009
accuracy			1.00	10000
macro avg	1.00	1.00	1.00	10000
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

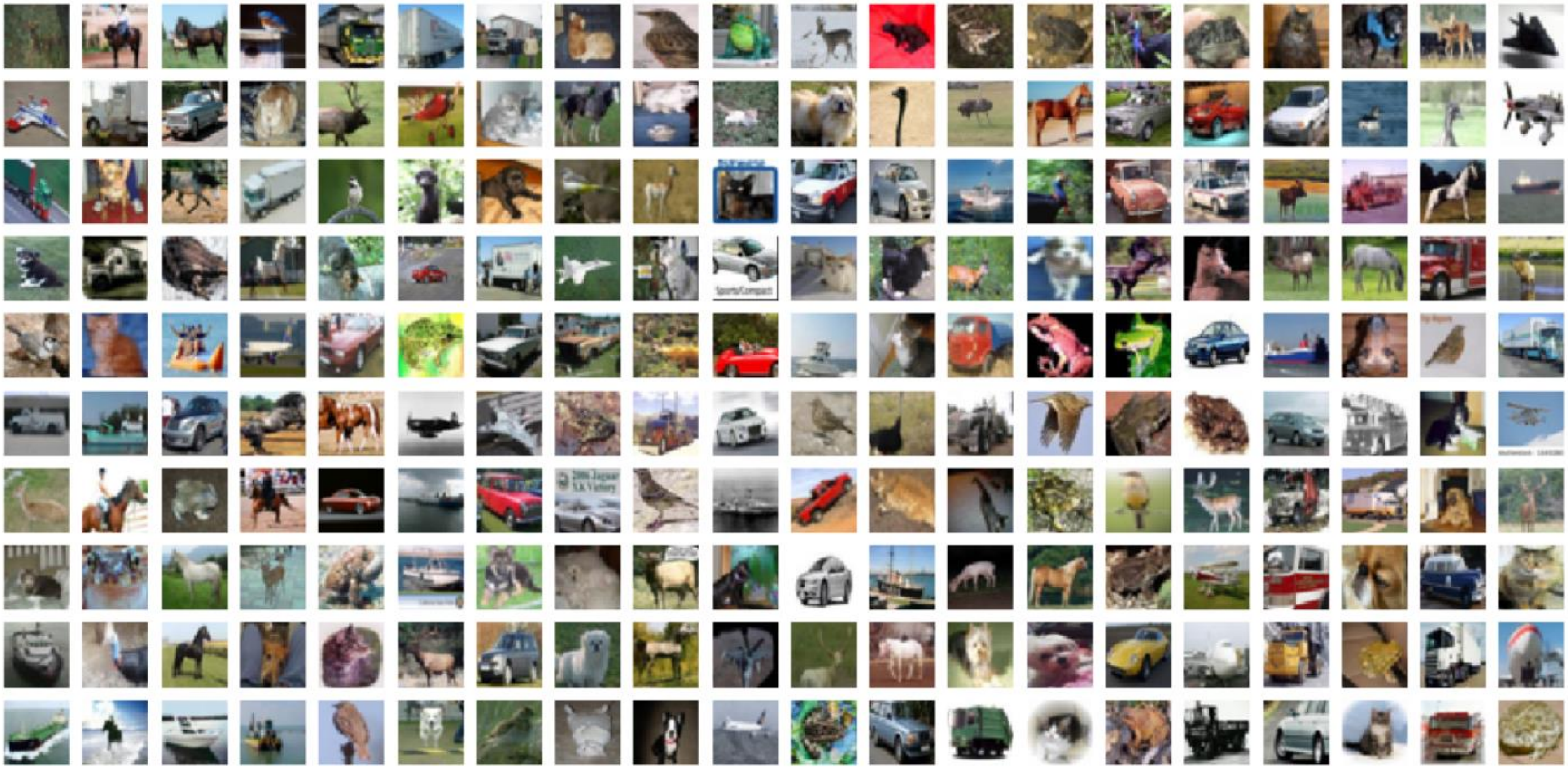
The number of misclassified examples after 50 epochs compared to 3 epochs has dropped from 247 to 37 out of 10,000 test examples, resulting in an error of 0.37%. Here are the misclassified examples:

```
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

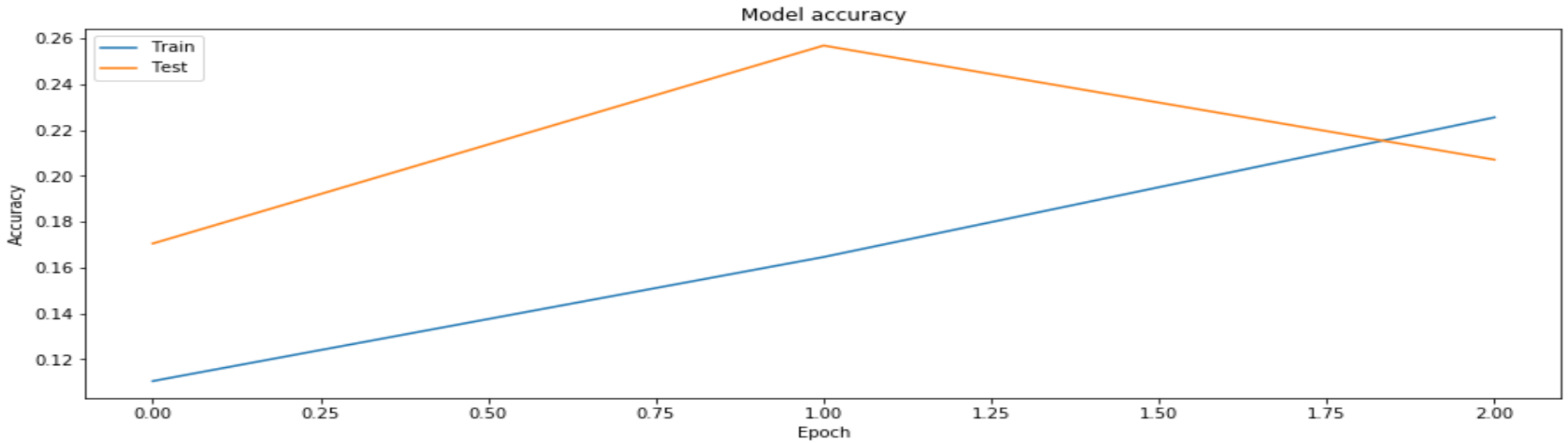
In [7]:

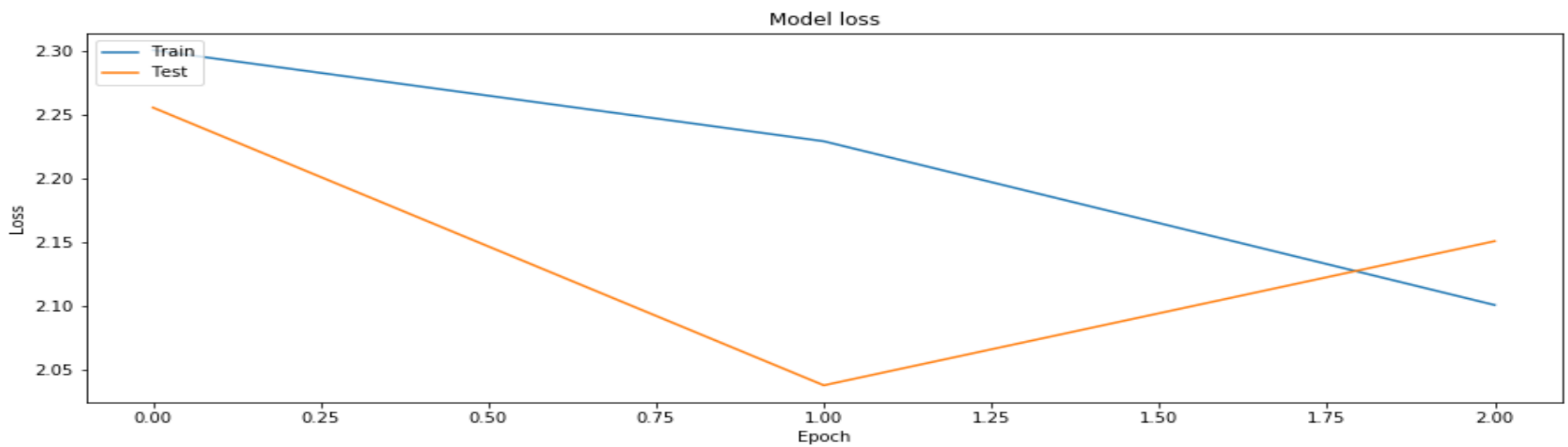
```
# Compile the model using optimizer
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['acc']) # List of metrics to be evaluated by the model during training and testing: https://keras.io/m

# 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
```



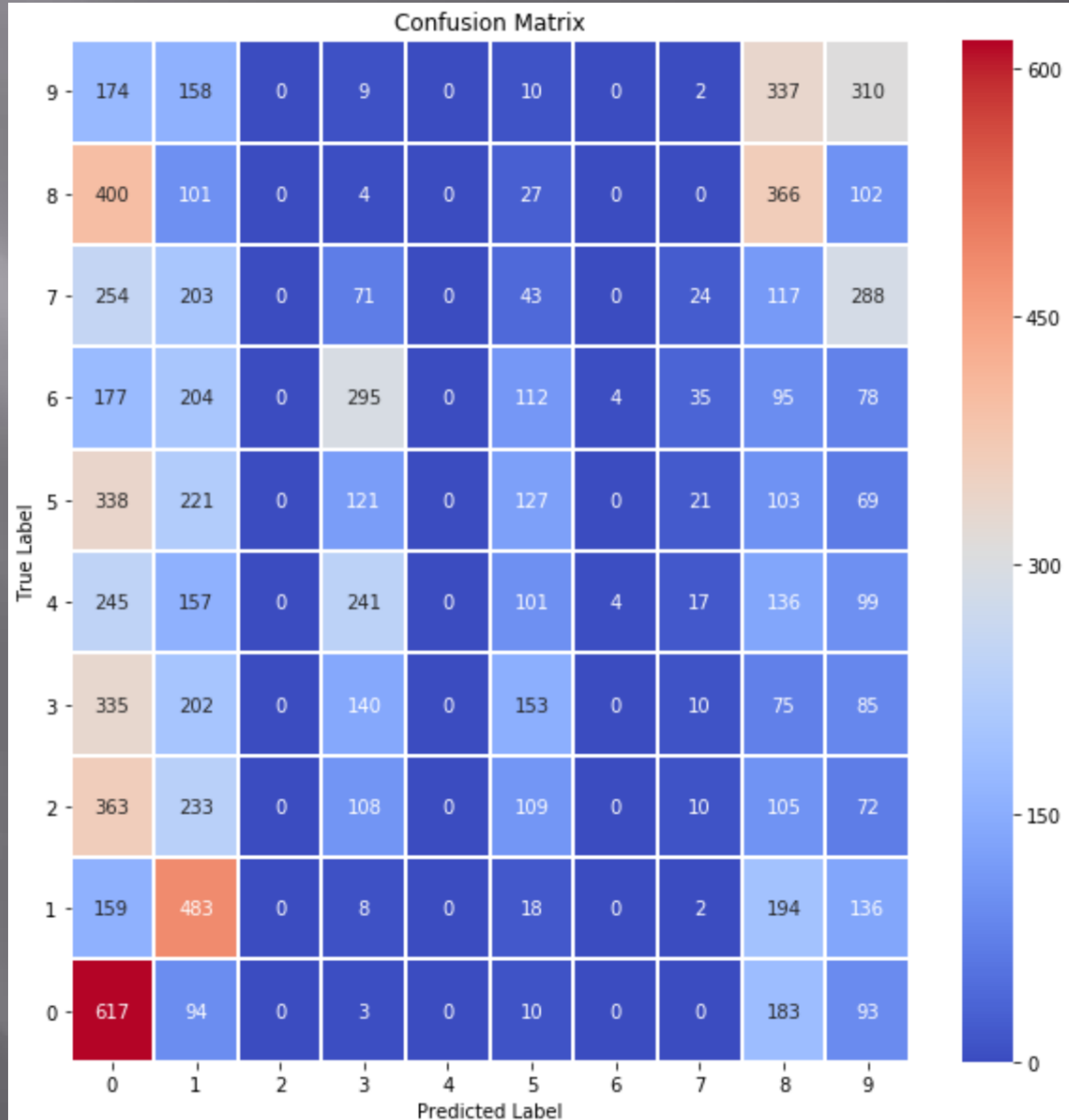


CIFAR-10 Classification in Jupyter

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]
```



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

CIFAR-10 Classification in Jupyter

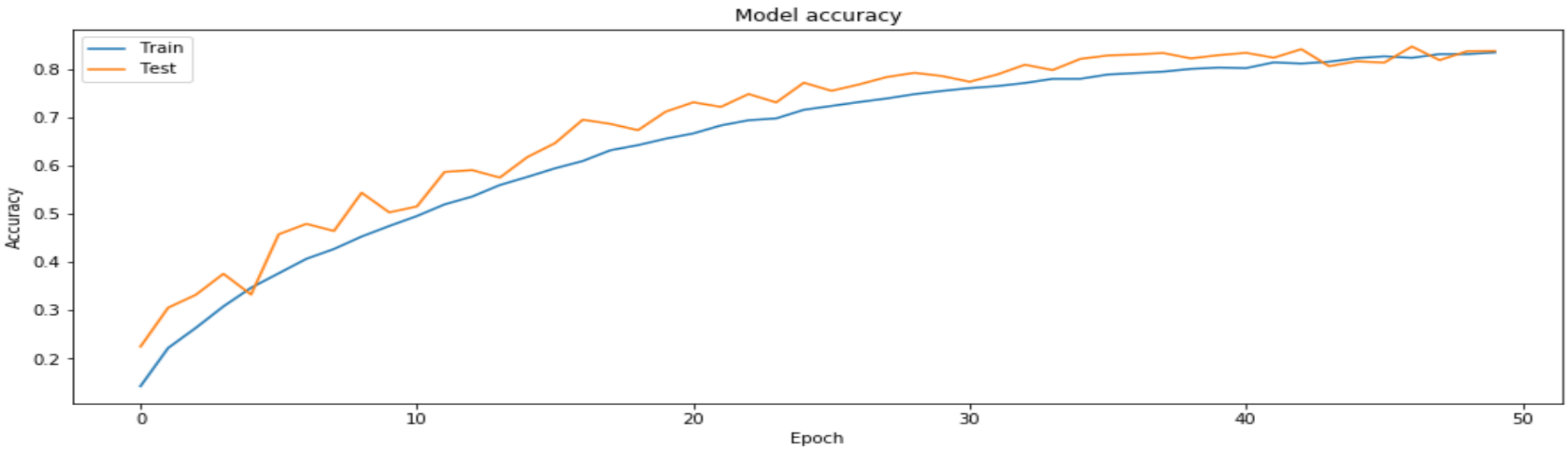
Let's train the network longer (50 epochs, a few hours) and as you can see the error (val_loss) systematically decreases, and the accuracy (val_acc) increases:

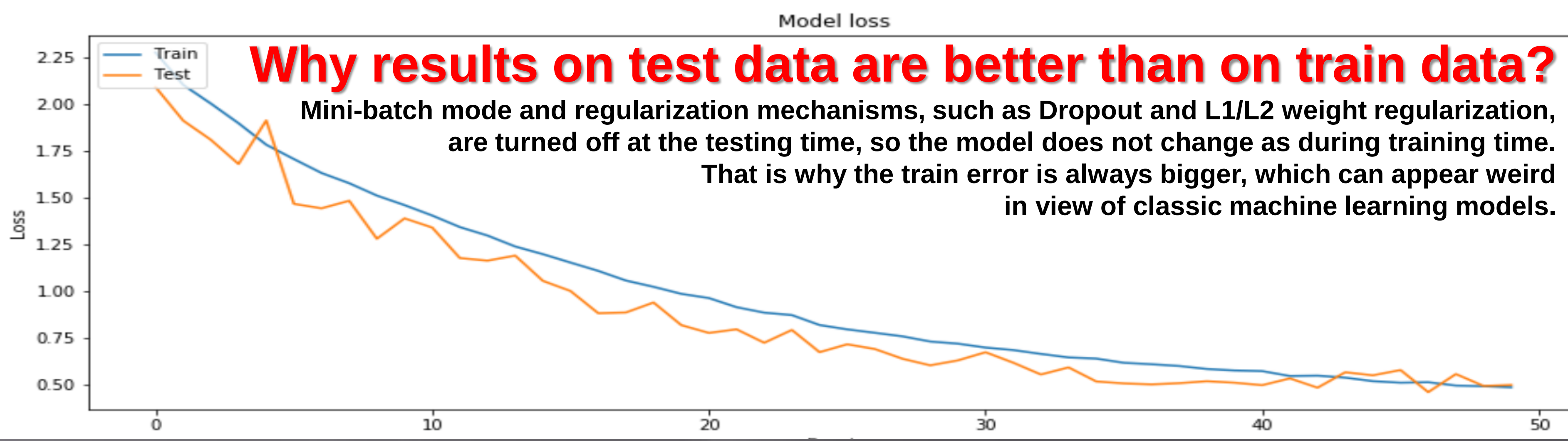
```
Epoch 1/50
97/97 [==============================] - 955s 10s/step - loss: 2.2744 - acc: 0.1426 - val_loss: 2.0892 - val_acc: 0.2247
                                                                                                  ⋮                 ⋮
Epoch 36/50
97/97 [==============================] - 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: 0.5103 - 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
```



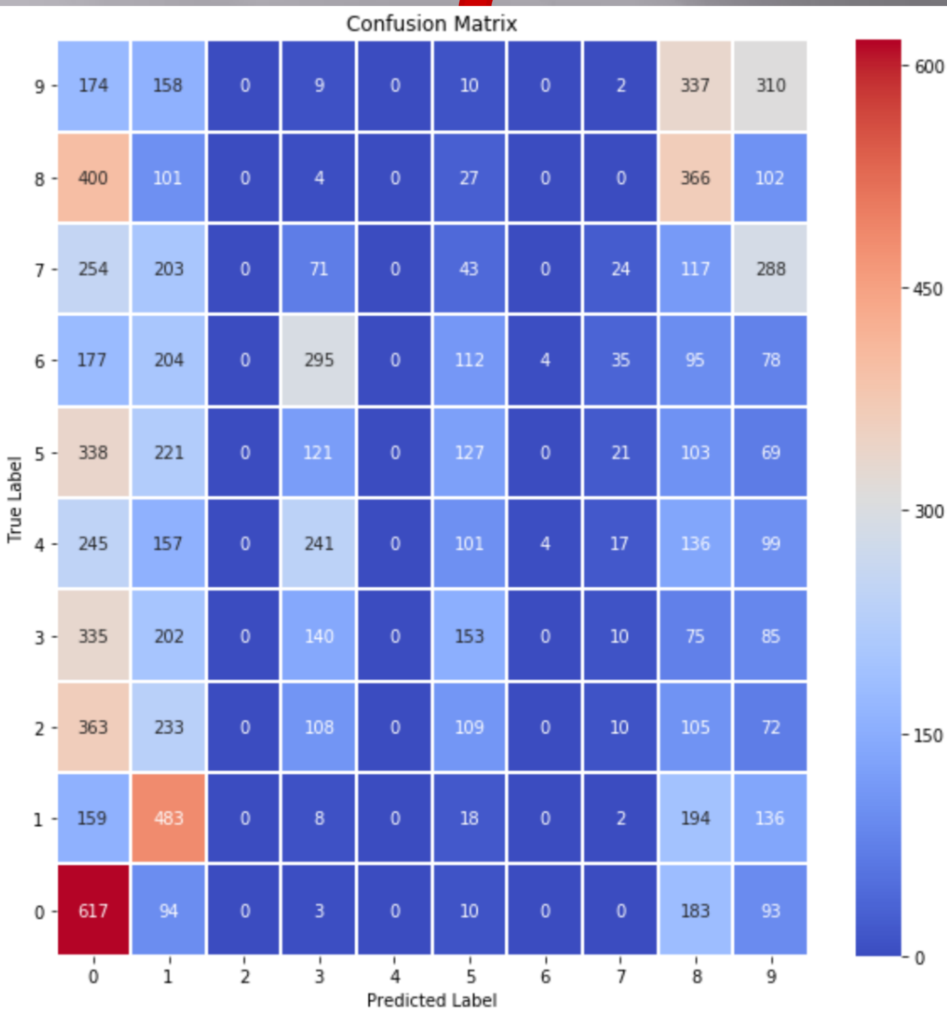


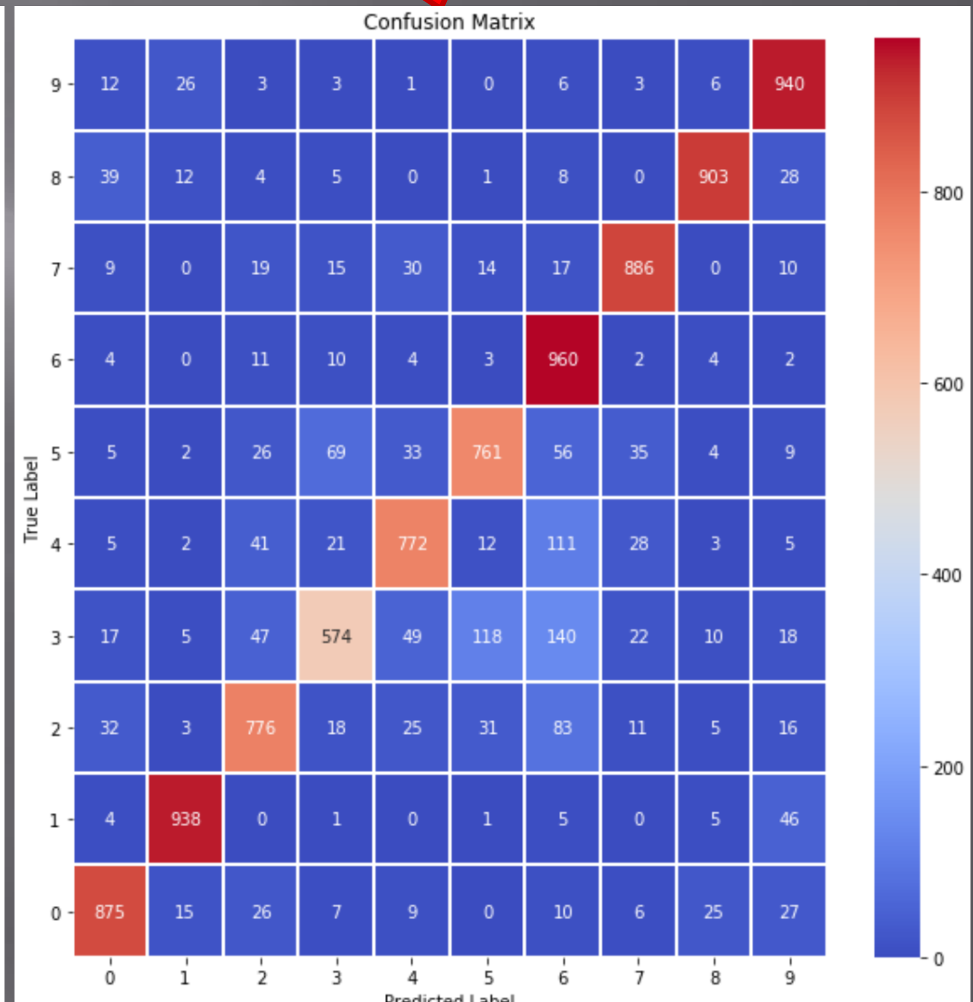
Why results on test data are better than on train data?

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:





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.

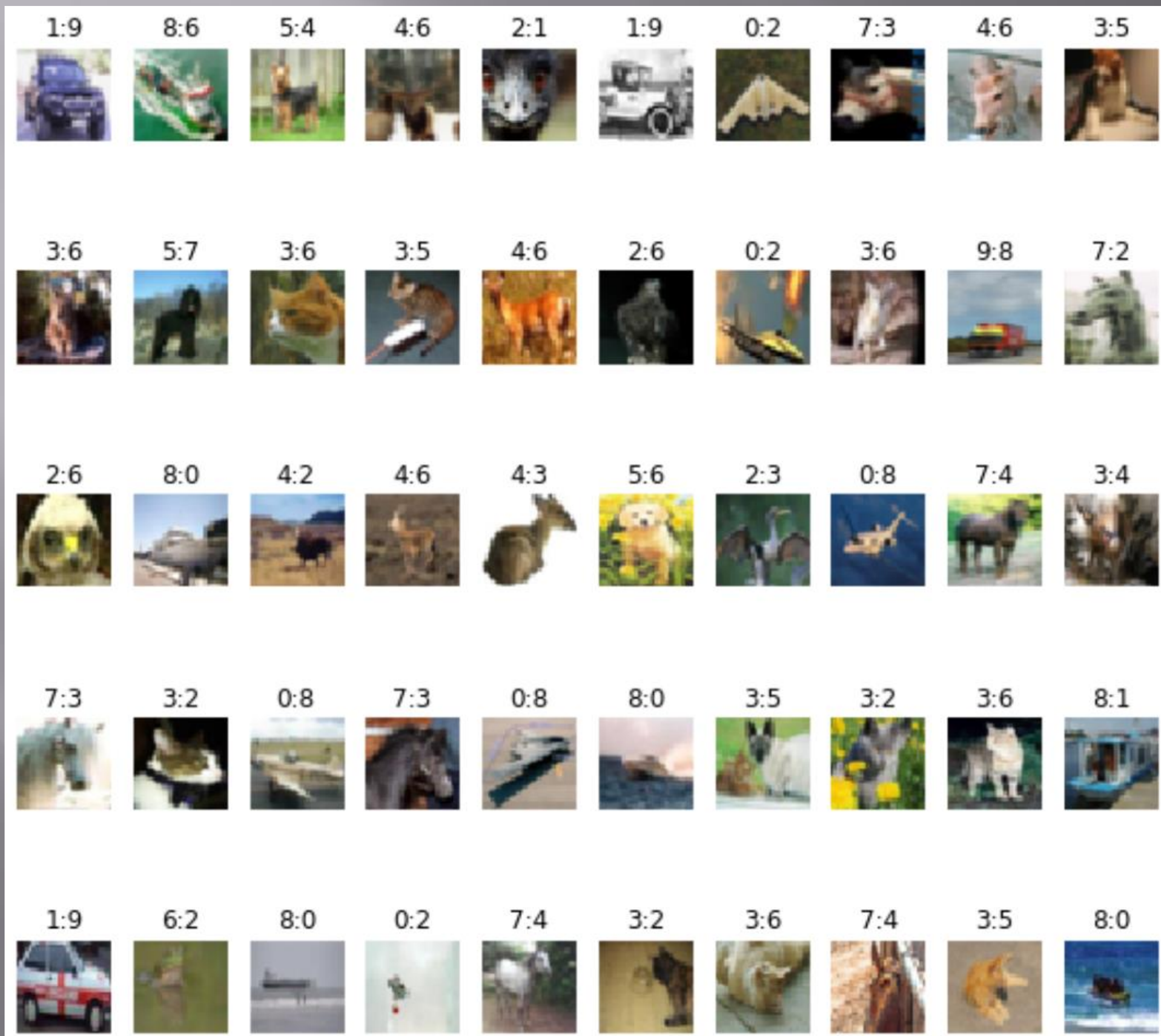
```
Number of misclassified examples:  7929
Misclassified examples:
[   0    3    4 ... 9994 9995 9999]
```

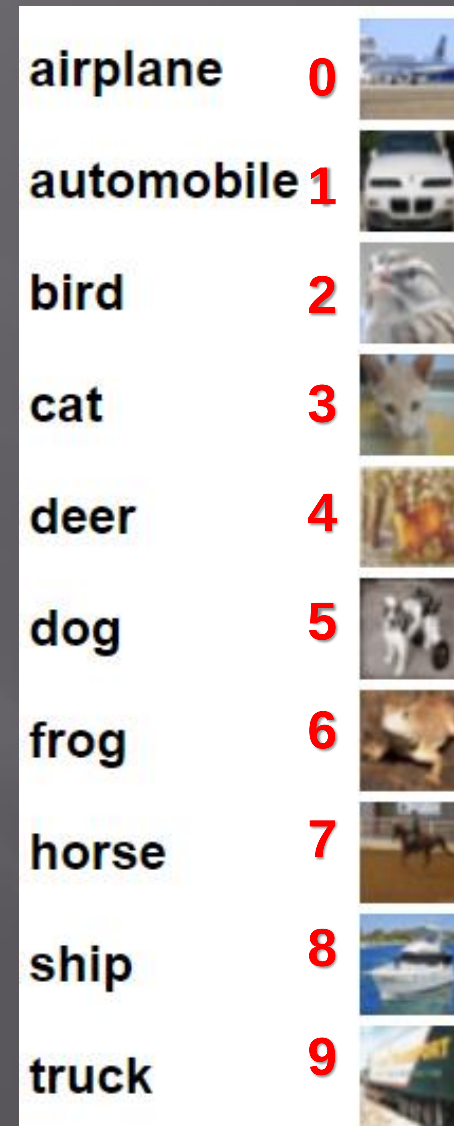
```
Number of misclassified examples:  1615
Misclassified examples:
[   9   15   24 ... 9982 9985 9996]
```

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:





CIFAR-10 Classification in Jupyter

Sample misclassified examples:

Let’s start with powerful computations!

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

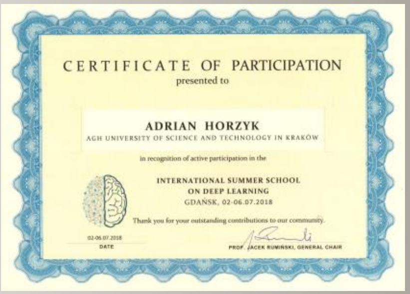
Bibliography and Literature

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



AGH

University of Science and Technology in Krakow, Poland