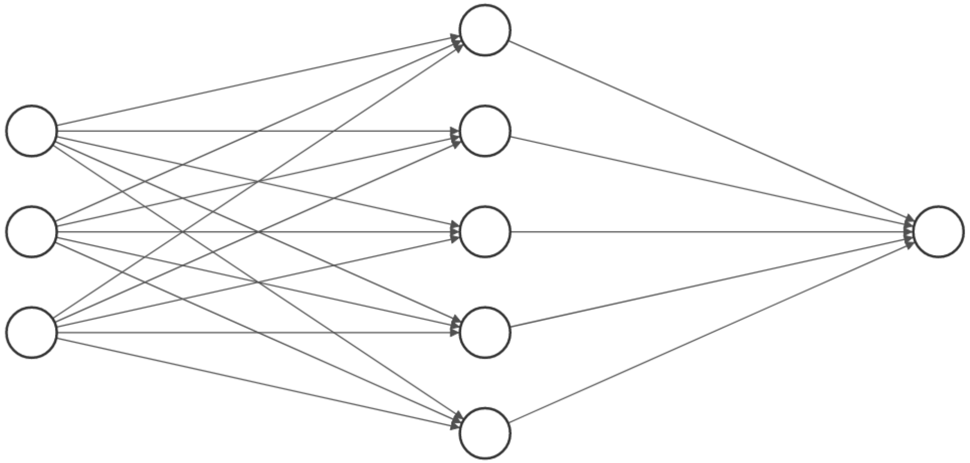
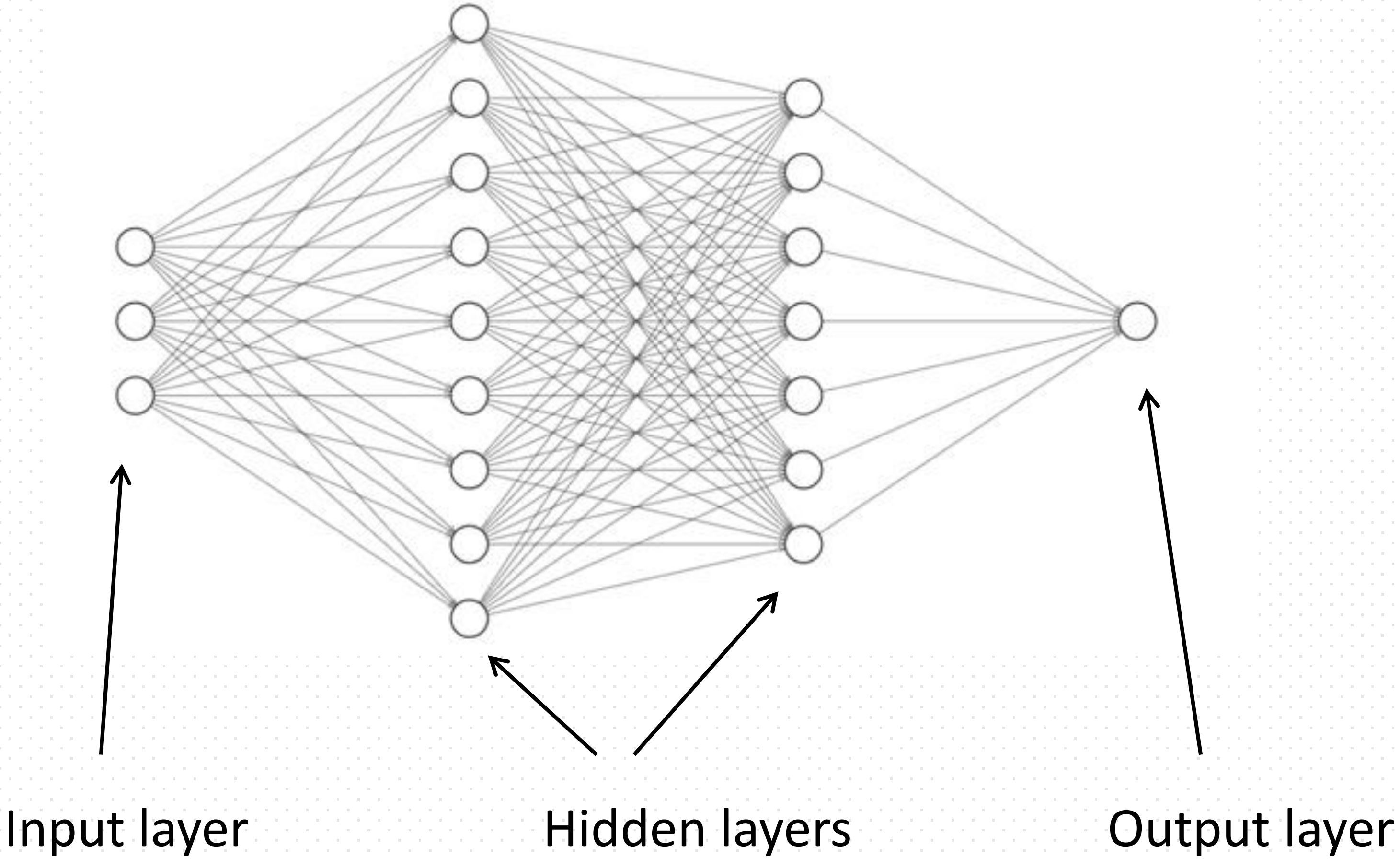
Goal

The use of selected programming libraries (TensorFlow / Keras) for the implementation of a one-way, multi-layer neural network



One-directional multi-layer neural network



The number of neurons in the layers

- In input layer - equal to the size of the input vector (`Nin`)
- In output layer - equal to the size of the output vector (`Nout`)
- Examples:
  - Approximation of the function of one variable: `Nin=1`, `Nout=1`
  - Sum (one output variable) of two numbers (two input variables): `Nin=2`, `Nout=1`
  - Classification of images from the MNIST set – handwritten digits, black and white images (color depth 1) with a size of 28x28 pixels: `Nin=28x28x1=784`, `Nout=10`

Hyperparameters

- Selected network parameters that are not subject to training:
  - The number of hidden layers and the number of neurons in the layers
  - activation functions in individual layers and their parameters
  - cost (error) function
  - optimizer: type and parameters (e.g. learning rate)
  - number of learning epochs
  - division coefficient into training / validation set
  - … and many others
- Hyperparameters are selected based on the type of problem to be solved, the number and type of training data, the analyst's experience / intuition, etc.

Supervised learning – backpropagation method

- Basic gradient learning is similar to the simple perceptron, but with many layers and many neurons in the layers.
- In the output layer, we calculate the error (eg RMSE) – total for all neurons
- We use the value of this error to modify the weights of neuron connections between the last hidden layer and the output layer, based on the appropriate formulas
- Then we modify the weights between successive layers back to the weights between the input layer and the first hidden layer

Programming of a multi-layer one-directional neural network

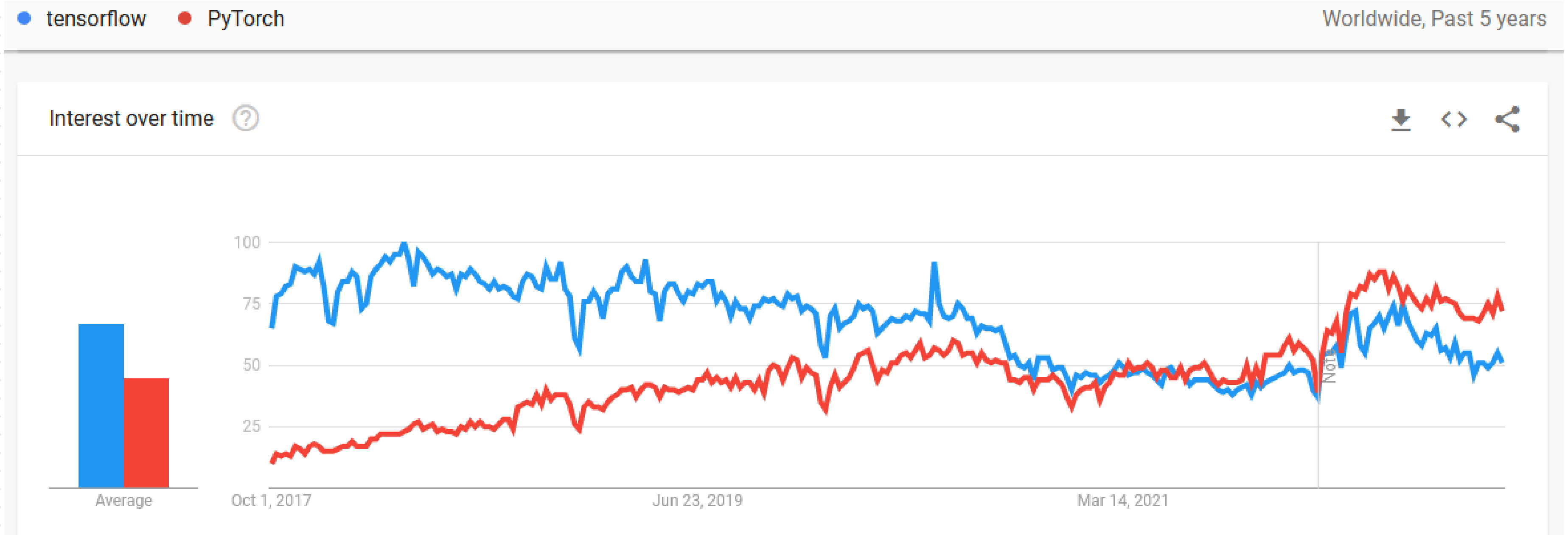
- Writing a program that implements the learning process from scratch in a selected language (especially in Python, for example) is not difficult, even without the use of additional libraries
- However, there are ready-made libraries, developed for years by development groups, which – in addition to basic functionalities – offer advanced possibilities (additional network layers, various optimizers, activation functions, error functions, etc.)
- Examples of such libraries: TensorFlow (with the Keras interface), PyTorch, NeuroLab, CAFFE, scikit-neuralnetwork

Tensorflow

- An open source library that helps to create and train machine learning models, e.g. in Python
- Initially developed by the Google Brain team, it is now used by many companies, including commercial ones
- Enables calculations on one or more processors (CPU) or graphics cards (GPU) on a personal computer or a dedicated computing machine, etc. without the need for (major) code modifications
- It presents calculations in the form of graphs - data flow diagrams
- It is quite extensive, but thanks to additional high-level APIs (e.g. Keras) its use is quite easy

Tensorflow vs Pytorch

- The Pytorch library has similar features and capabilities to TensorFlow, but is preferred by some as slightly easier to use and slightly faster; made by Facebook
- They enjoy similar popularity and basically do not have much competition



Installation of the TensorFlow

https://www.tensorflow.org/install

```
pip install tensorflow
```

- The latest version (2.x) for CPU and GPU is installed automatically (even if the machine does not have supported graphics cards)
- Be careful! TensorFlow is often adapted to the latest versions of Python with a long delay
- Older versions (1.x) had separate packages for CPU and GPU
- One can use a ready-made Docker container
- Installation is not required on Google Colab
- Keras interface is part of the library (from version 2.0)

Interface (API) Keras

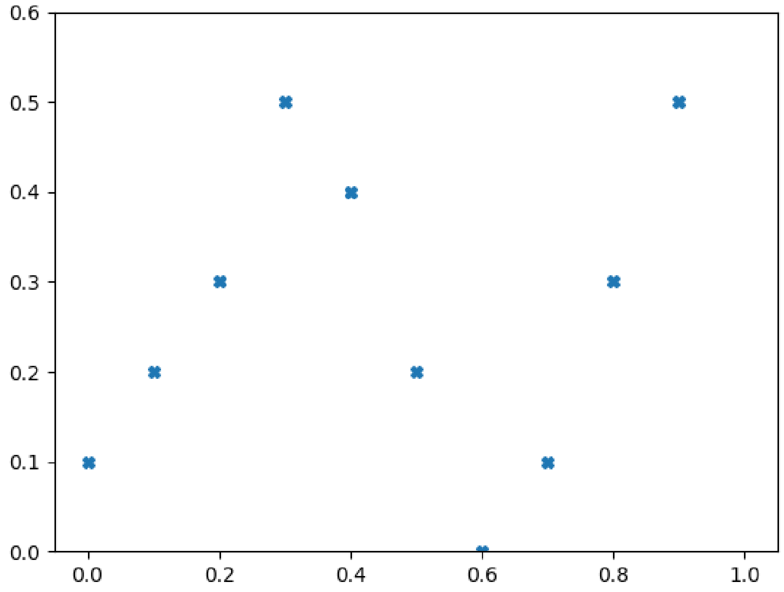
- High-level machine learning interface (including deep learning)
- Written in Python
- It works on the basis of the TensorFlow library and (from version 2.0 TensorFlow) is its integral part
- It allows you to quickly define and train machine models composed of various layers of neural networks
- In addition to the basic functionalities, it has an entire ecosystem, which includes, for example, Keras Tuner for the automatic search for hyperparameter values

https://keras.io/

Example: one variable function approximation

```
0.0, 0.1
0.1, 0.2
0.2, 0.3
0.3, 0.5
0.4, 0.4
0.5, 0.2
0.6, 0.0
0.7, 0.1
0.8, 0.3
0.9, 0.5
```

- We have points (X, Y)



- We want to find approximate values of these points for any values on the abscissa axis (i.e. make an approximation) – regression problem

Solution design

- We will design and train a one-directional, multi-layer neural network (file `train.py`)
- File `predict.py` will do approximation
- We will use the TensorFlow library with the Keras interface
- Selection of hyperparameters :
  - network architecture: 1 - 3 - 5 - 1 (two hidden layers with 3 and 5 neurons respectively, 1 input and output neurons)
  - lepochs: 10000
  - activation function: `tanh` in hidden layers, `sigmoid` in output layer
  - Adam optimizer (https://arxiv.org/abs/1412.6980)
  - Loss: RMSE

Imports

- NumPy (ndarray arrays) and Matplotlib (charts)

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

- TensorFlow can be imported in its entirety

```
import tensorflow as tf
```

- Or (for convenience) only the functions that will be used, e.g.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

(you can also combine both methods)

Input file and input parameters

```
# CSV file name with train data
DATAFILE = "train.csv"

# No of epochs for training
EPOCHS = 10000

# Validation / train split
SPLIT = 0.2

# Size of input vector (number of neurons in the input layer)
Nin = 1
```

- We will write the program so that it can be easily adapted to other problems, so the number of neurons in the input layer is a parameter
- The number of neurons in the output layer will be calculated automatically

Load data

```
# Load train data from CSV file
data = np.loadtxt(DATAFILE, delimiter=",")

# Size of the output vector
# (number of neurons in the output layer)
Nout = data.shape[1] - Nin
```

- Loading data from a CSV file
- Calculation of the number of neurons in the output layer as the difference between the total number of rows and the number of neurons in the input layer
- One could of course use the parameter `Nout` directly, as for `Nin`

Division of data into vectors X i Y

```
# Divide into X and Y vectors
X = data[:, :Nin]
Y = data[:, Nin:]
```

- The vector `X` will be the `Nin` values, starting with the first (index zero) – for all rows from the loaded set
- The `Y` vector is the rest of the data on each line read

- In such code snippets, you can clearly see the advantage of dealing specifically with array ranges in Python

One-directional network model

```
# Create model
# One-directional network
model = tf.keras.models.Sequential()
```

- The sequential model is a simple model composed of successive layers
- These layers can be specified immediately in the constructor
- In the vast majority of cases, it is sufficient to obtain good machine learning results
- Keras has an additional functional API that allows you to model any complex models based on various architectures of neural networks

List of available models: https://keras.io/api/models/

Input and first hidden layer

```
# Input layer of dimension Nin
model.add(tf.keras.layers.InputLayer(input_shape=(Nin, )))
# Hidden layer: 3 neurons
model.add(tf.keras.layers.Dense(3, activation='tanh'))
```

- In this way, the input layer of the size `Nin` was defined (and added to the model) (there is no computational process in this layer, so you do not need to provide e.g. the activation function)
- Then the first hidden layer was defined, which consists of 3 neurons, and the activation function is the hyperbolic tangent
- `Dense` means a fully connected layer (each neuron from the previous layer with each neuron of the current layer)

Input and first hidden layer (alternative)

- You can also define the input layer and the first hidden layer with one statement:

```
model.add(tf.keras.layers.Dense(3,
                                input_dim=Nin,
                                activation='tanh'))
```

- And if you used an alternate import of a single Dense function, the notation would simplify to

```
model.add(Dense(3, input_dim=Nin, activation='tanh'))
```

- This shows how flexible the Keras interface and Python import system are

Second hidden layer and output layer

```
# Hidden layer: 5 neurons
model.add(tf.keras.layers.Dense(5, activation='tanh'))
# Output layer: Nout neurons
model.add(tf.keras.layers.Dense(Nout, activation='sigmoid'))
```

- The second hidden layer consists of 5 neurons and the activation function is a hyperbolic tangent (similar to the first layer)
- The output layer has of course the `Nout` of neurons, in our case of the approximation of the function of one variable it will be the value 1
- The sigmoid function was chosen as the activation function

Other layers: https://keras.io/api/layers/

Other activation functions: https://keras.io/api/layers/activations/

Build (compile) the model

```
# Compile model
model.compile(loss='mean_squared_error', optimizer='adam')
```

- We compile the model, most often specifying a cost (error) function and an optimizer that will be used to minimize this function
- Common cost functions (https://keras.io/losses/):
  - MSE – `mean_squared_error`
  - cross entropy – `binary/categorical_crossentropy`
- Common optimizers (https://keras.io/optimizers/):
  - stochastic gradient – `sgd` (equivalent to back propagation)
  - Adaptive moment estymation (Adam) – `adam`
  - RMS propagator – `rmsprop`

Model training

```
# Fit the model
H = model.fit(X, Y, epochs=EPOCHS, validation_split=SPLIT)
```

- `X`, `Y` – training data, from which the validating subset will be automatically separated in accordance with the parameter value `SPLIT`
- The number of epochs is also given
- Other common parameters:
  - `batch_size` – input sample size for gradient calculation (default 32)
  - `shuffle` – a logical variable deciding whether training data will be shuffled before each epoch
  - `callbacks` – list of callback functions used during training
- Data about the course of training are saved to the facility „History”

More about `batch_size`

- Case `batch_size = 1`
  - a gradient is computed for each training sample and the weight change is immediately calculated
  - computationally expensive (frequent weight changes and limited use of vector calculations)
  - uses little operating memory
- Case `batch_size = Nsamples # Number of samples`
  - all samples are passed through the net, the mean gradient is calculated and only then is the weight changed
  - possibility of vector calculations (passing many samples through the network simultaneously)
  - less computationally expensive, but much more memory is required
- Case `batch_size = N # 1 < N < Number of samples`
  - a compromise between the above

Alternative division into training and validation set

- If we want to divide the data set into trainers and validators ourselves, the *sklearn* package can be used

```
from sklearn.model_selection import train_test_split
# Split into train and validation sets
(Xtrain, Xvalid, Ytrain, Yvalid) = train_test_split(
      X, Y, test_size=SPLIT)
```

- And then trigger the learning process a little differently:

```
H = model.fit(Xtrain, Ytrain, epochs=EPOCHS,
              validation_data=(Xvalid, Yvalid))
```

Thanks to this, we get a little more control over the training process

Model evaluation

```
# Evaluate the model
print("Evaluate model:")
model.evaluate(X, Y)
```

- In our case, the evaluation result is the value of the error function on a given set
- Of course, it would be better to evaluate the model on test data, but we do not have such data available
- Alternative display of the evaluation result:

```
test_score = model.evaluate(X, Y)
print("Loss on complete training set = {:.15f}".format(test_score))
```

Save the model to a file

```
# Save model to file
model.save("model.h5")
```

- The model is saved to a file in HDF5 format (default extension .h5)
- It is a standard that describes the structure of a file, into which you can save arrays of data in a way that preserves the hierarchy and structure of the data
- If we had not given the .h5 extension, the model would have been saved in the internal Keras package format (which usually has a larger volume)

Graphs of the error as a function of the epoch

```
# Plot the train and validation losses
plt.style.use("ggplot")
plt.figure()
plt.plot(range(EPOCHS), H.history["loss"], label="Train loss")
plt.plot(range(EPOCHS), H.history["val_loss"],
                        label="Validation loss")
plt.title("Train / validation loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.legend(loc="lower left")
plt.savefig("model.png")
```

- The data on the learning process is saved in the form of a dictionary with an appropriate key `loss` / `val_loss`

Sample script invocation and its result (1)

`> python train.py`

Epoch 1/10000

1/1 [==============================] - ETA: 0s - loss: 0.1165
1/1 [==============================] - 1s 1s/step - loss: 0.1165 - val_loss: 0.0315
...
Epoch 10000/10000

1/1 [==============================] - ETA: 0s - loss: 8.0519e-04
1/1 [==============================] - 0s 22ms/step - loss: 8.0519e-04 - val_loss: 0.0210
Evaluate model:

1/1 [==============================] - ETA: 0s - loss: 0.0048
1/1 [==============================] - 0s 17ms/step - loss: 0.0048

Sample script invocation and its result(2)

You can see the model overtraining (the error on the validation set increases)

Script for prediction (approximation)

```
import numpy as np
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt

# CSV file name with test data
DATAFILE = "test.csv"

# Load test data from CSV file
X = np.loadtxt(DATAFILE, delimiter=",")
```

- There is a single data column in the test.csv file containing numbers from the range [0, 1) with a step of 0.01, for which the prediction (approximation) will be made
- Instead of writing these numbers to a file, you could generate them "on the fly"

Model loading and prediction

```
# Load model and print it's summary
model = load_model("model.h5")
model.summary()

# Calculate predictions
Ypredicted = model.predict(X)
```

- After loading model, `summary()` displays basics information about the model
- Method `predict()` performs prediction on `X` data

The graph of the results of the prediction

```
# Plot results
xmin, xmax, ymin, ymax = plt.axis([-0.05, 1.05, 0, 0.8])
plt.plot(X, Ypredicted, ".")

# Add traing data to a plot
data = np.loadtxt('train.csv', delimiter=",")
Xt = data[:,0]
Yt = data[:,1]
plt.plot(Xt, Yt, "X")

plt.show()
```

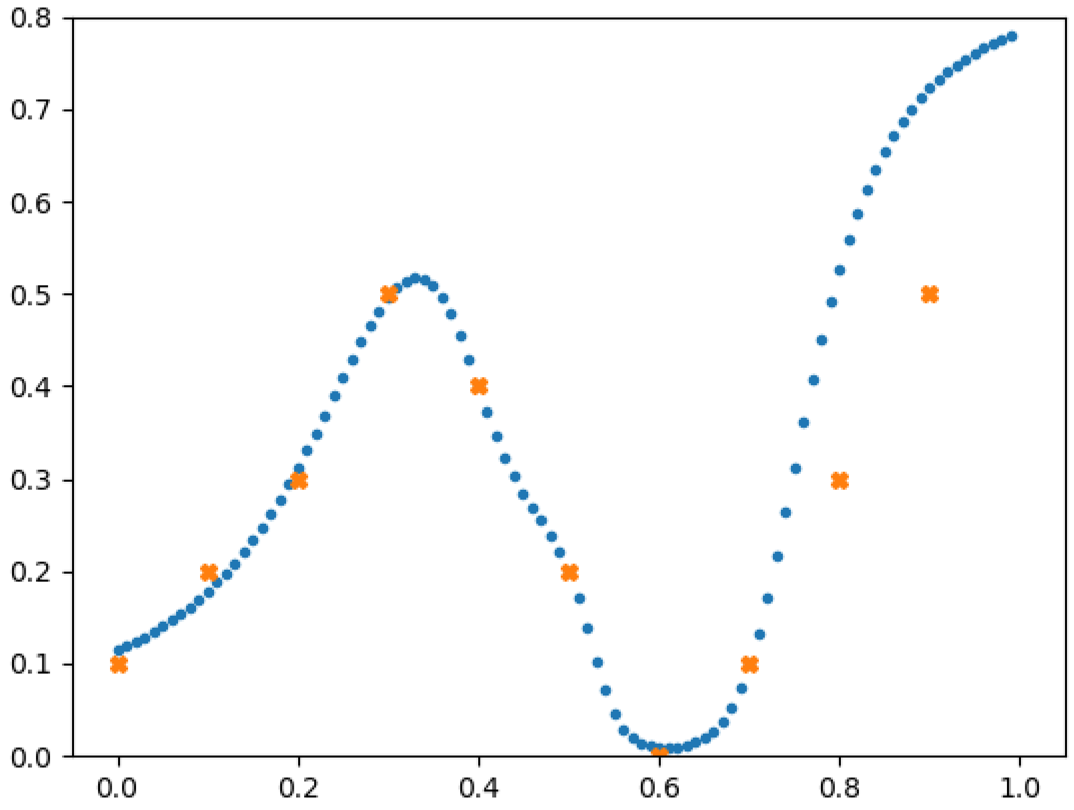
- A graph is drawn from the prediction results, along with training data (to better visualize the approximation result)

Sample script invocation and its result(1)

```
> python predict.py
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)              Output Shape              Param #
=================================================================
dense (Dense)             (None, 3)                 6
_________________________________________________________________
dense_1 (Dense)           (None, 5)                 20
_________________________________________________________________
dense_2 (Dense)           (None, 1)                 6
=================================================================
Total params: 32
Trainable params: 32
Non-trainable params: 0
```

Sample script invocation and its result(2)



- We see the correct (though not perfect) result of the approximation

Summary

- We managed to implement a simple, one-way, multi-layer neural network, train it and use it to solve the problem of approximation of one variable
- Thanks to the use of the TensorFlow / Keras library, the code was closed in (almost) a dozen lines of code and the results turned out to be correct
- In some sense, however, we lose (despite the open code) the possibility of direct insight into how the functionality is implemented ("black box")

More examples

- Examples of the attached source codes (apart from the one discussed in the presentation) demonstrate the use of the TensorFlow / Keras library for:
  - the summation problem, solved with a simple perceptron (similar to the presentation, where the problem was solved from scratch, without the use of machine learning libraries)
  - the problem of image classification – handwritten digits available in the MNIST file (without deep learning / using convolutional networks)

Final remarks (1)

- TensorFlow displays a lot of warnings and information (e.g. when it cannot find the GPU on the system, when the optimal compilation is not used, etc.); they can be turned off by setting the appropriate environment variable and calling dedicated methods, for example:

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.get_logger().setLevel('ERROR')
```

- One should remember about data normalization (in our case it was not necessary)

Final remarks (2)

- TensorFlow / Keras provides easy access to several data sets, e.g. MNIST, CIFAR10 etc.

  Lista: https://keras.io/api/datasets/

- A layer `Flatten` is a very useful layer, which allows you to "flatten" multidimensional inputs
- When the output layer consists of many neurons, the prediction results in a vector with a size corresponding to the number of output neurons

Final remarks (3)

- The cost function and the optimizer can be passed in the `fit()` method as a class, not a function name, so you can pass parameters to them
- Example:

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.losses import CosineSimilarity
model.compile(loss=CosineSimilarity(axis=1),
              optimizer=SGD(learning_rate=0.1))
```

(we provide the parameter specifying the axis against which the cost function will be calculated and the learning coefficient for the gradient method)

Final remarks (4)

- Multi-threaded and GPU computing
  - TensorFlow (from version 2.x) will detect the supported GPUs by itself (Nvidia with CUDA drivers) and will use them for calculations, without having to change the code
  - Since using the GPU often even slows down computation with simple problems, you can turn it off :

```
os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
```

- To force multithreading on the CPU, an additional parameter must be passed to the method `fit()`:

```
H = model.fit(X, Y, validation_split=0.2, epochs=100,
              use_multiprocessing=True)
```

Final remarks (5)

- Pseudorandom number generator initialization
  - Thanks to this, the reproducibility of the results will be ensured
  - Unfortunately, it works on CPU only (not GPU)
  - This has to be done separately for NumPy and separately for TensorFlow, for example like this:

```
RANDOM_SEED = 100
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
```

(assuming appropriate imports)

  - In newer versions of TensorFlow / Keras (replaces both of the above):

```
tf.keras.utils.set_random_seed(RANDOM_SEED)
```

Final remarks (6)

- Alternative builds
  - TensorFlow is compiled by default for a specific version of Python, for CPU + GPU (CUDA) and to use some advanced processor instructions to increase computing performance (e.g. AVX2)
  - In some cases (especially when we run scripts on a virtual private server, VPS) there is a need to use an alternative compilation, e.g. with disabled support for additional instructions or (to limit the disk space taken) with disabled GPU support, or for a specific version of Python / CUDA
  - In such cases, alternative builds (created by the community) can be used:

    https://github.com/davidenunes/tensorflow-wheels

    https://github.com/fo40225/tensorflow-windows-wheel