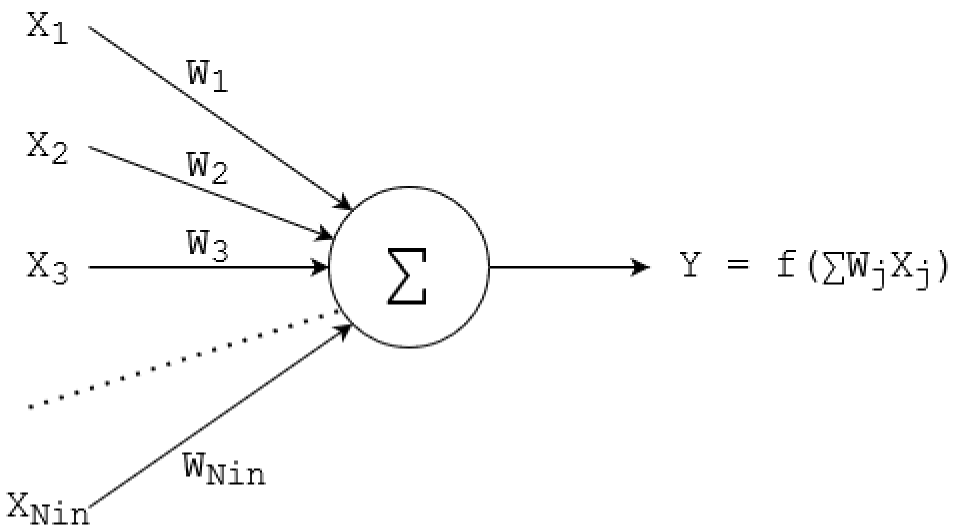
Goal

Implementation (from scratch) of a simple perceptron with training supervised by the gradient method

```
single output
     ^
     |
     O
   / | \    weights
  Nin inputs
```

A bit of theory – a simple perceptron

Simple perceptron (with omitted *bias* – input 0):



f – activation function, usually non-linear, e.g. hyperbolic tangent or sigmoidal function

X_j – inputs, W_j – weights, Y – output

Simple perceptron – supervised learning (1)

- The learning process consists in tuning weights W_j, in such a way that the input data sets $X^T{}_j$ correspond to the desired outputs Y^T
- Set $X^T{}_j, Y^T$ $(j=1 \ \dots \ Nin)$ is a training pair
- We have p such pairs: $X^T{}_{ij}, Y^T{}_i$ $(j=1 \dots Nin, \ i=1 \dots p)$,
- For each pair do:
  - calculate the output from the perceptron: $Y_i = f(\sum W_j X^T{}_{ij})$
  - calculate the contribution to the mean squared error: $MSE_i = (Y_i - Y^T{}_i)^2$
  - change the weights based on the error function gradient:

$$W_j \leftarrow W_j - grad(MSE_i) = W_j + \eta \ f'(\sum W_j X^T{}_{ij})(Y^T{}_i - f(\sum W_j X^T{}_{ij}))X^T{}_{ij}$$

η – learning rate; f' – derivative of the activation function

Simple perceptron – supervised learning (2)

- The data set is usually divided into training and validating
  - The training set is used to calculate the weight change
  - The validating set is used to control the learning process, i.e. when the error on the validating set begins to grow, we are dealing with the so-called network overfitting
- The operations of calculating the error and changing the weights are repeated many times for the training data set, according to the set number of epochs
- For each epoch, we can calculate the mean square error as the root of the sum of the partial errors divided by the number of samples training

General procedure to be followed

1. Loading training data
2. Data normalization
3. Division of data into subsets: training and validation
4. Perceptron training on a training subset and RMSE calculation
   - process control on the validating subset - RMSE calculation on this set
   - drawing a graph of RMSE versus epoch
5. Save the trained model to a file
6. Model test on test data, calculation of basic metrics

Assumptions of the solution (1)

- Implementing functionality in the form of the `simple_perceptron` class
- Use of external libraries only for auxiliary operations (e.g. randomization, graphs, saving / reading files)
- Possibility to select hyperparameters
  - Activation function: `tanh`, `sigmoid` lub `relu`
  - Number of epochs
  - Learning rate
  - Coefficient of division into training / validation set
  - Flag to enable shuffling of the order of training data

Assumptions of the solution (2)

- Loading data from a `CSV` file
- Save the trained model to the `YAML` file
- Reading the saved model from the `YAML` file
- Draw / save RMSE versus epoch plots for training and validation set
- Basic error check
- A program running in the console
- Ability to specify parameters from the command line when starting the program

Assumptions of the solution (3)

Two control programs

- `train.py`
  - training the model on training data, control on validating data
  - saving the trained model to a file
- `predict.py`
  - reading a previously saved model from a file
  - prediction based on test data read from the file
  - calculation of RMSE and R^2 metrics

The skeleton and the implementation of the class `simple_perceptron`

```
class simple_perceptron:
    '''
     Simple perceptron

            single output
                  ^
                  |
                  O
                / | \    weights
              Nin inputs
    '''
```

The following slides will present: a description of the methods and an exemplary implementation of the methods

Imported modules

```
import numpy as np
import csv
import yaml
import matplotlib.pyplot as plt
import sys
from timeit import default_timer as timer
```

- `numpy` – random number generator, mathematical functions
- `csv` – read data from `CSV`
- `yaml` – save / read `YAML`
- `matplotlib` – graphs
- `sys` – termination of the program in the event of an error
- `timeit` – measuring the time of program execution

Constructor

Takes hyperparameters and stores them as class properties, with default values given

```
def __init__(self, epochs=100,
              learning_rate=0.1,
              activation='tanh'):
    '''
    Constructor.
    Parameters:
        epochs - number of epochs (int)
        learning_rate - learning_rate (float)
        activation - activation function,
       (str) of ['tanh', 'sigmoid', 'relu']
    '''
```

Constructor – implementation

```
self.epochs = epochs
self.learning_rate = learning_rate
self.activation = activation
random_seed = 100
np.random.seed(random_seed)
```

- Function `random.seed()` allows to initialize the pseudorandom number generator, which guarantees repeatability of the results
- Of course it was possible `random_seed` treat it as a hyperparameter passed in the constructor

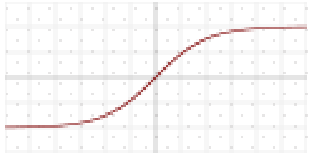
Constructor – alternative implementation

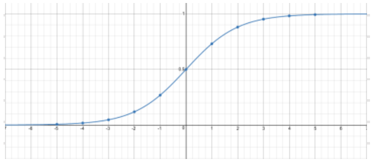
```
def __init__(self, **kwargs):
    self.__dict__.update(kwargs)
    random_seed = 100
    np.random.seed(random_seed)
```

- This method uses an input parameter in the form of a dictionary (`**kwargs`)
- This dictionary is used to automatically set the class properties
- This way, you can avoid specifying / checking optional constructor parameters

Activation function

The function by which the value of the perceptron output will be calculated; will be implemented as one of the three selectable functions: `tanh`, `sigmoid`, `relu`





```
def f(self, x):
    '''
    Calculates activation function at x.
    Parameters:
        x - an argument (float)
    Returns:
        value of the activation function at x (float)
    '''
```

Activation function – implementation

```
if self.activation == 'tanh':
    f = np.tanh(x)
elif self.activation == 'sigmoid':
    f = 1 / (1 + np.exp(-x))
elif self.activation == 'relu':
    f = x if x > 0 else 0
else:
    sys.exit('Error: Unknown activation function.')

return f
```

- The function is selected based on the property set in the constructor
- Function `tanh` could be implemented directly from the formula

$$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

Derivative of the activation function

Necessary in the gradient learning process; will be implemented as a derivative of one of the three functions: `tanh`, `sigmoid`, `relu`

```
def fp(self, x):
    '''
    Calculates derivative of the activation function at x.
    Parameters:
        x - an argument (float)
    Returns:
        value of the derivative to the activation function
        at x (float)
    '''
```

Derivative of the activation function – implementation

```
if self.activation == 'tanh':
    fp = 1 - np.tanh(x)*np.tanh(x)
elif self.activation == 'sigmoid':
    fp = np.exp(-x) / (1 + np.exp(-x))**2
elif self.activation == 'relu':
    fp = 1 if x > 0 else 0
else:
    sys.exit('Error: Unknown activation function.')

return fp
```

- Twin implementation as for the counting of the activation function
- You can easily add new features by adding additional conditions

Loading the training / test set

```
def read_input_data(self, filename, normalize=True):
    '''
    Reads input data (train or test) from the CSV file.
    Parameters:
        filename - CSV file name (string)
            CSV file format:
                input1, input2, ..., output
                                ...
        normalize - flag for data normalization (bool, optional)
    Sets:
        self.Nin = number of inputs of the perceptron (int)
    Returns:
        X - input training data (list)
        Y - output (expected) training data (list)
    '''
```

Loading the training / test set – implementation (1)

```
# Read CSV data
try:
    file = open(filename, 'rt')
except FileNotFoundError:
    sys.exit('Error: data file does not exists.')

dataset = csv.reader(file, delimiter=',',
                     quoting=csv.QUOTE_NONNUMERIC)
```

- A module was used: `csv`
- Alternatively, you can use the function `loadtxt()` from NumPy
- Parameter `quoting=csv.QUOTE_NONNUMERIC` provides numerical data retrieval in numeric rather than text format

Loading the training / test set – implementation (2)

```
# Construct the X and Y lists. This is a simple perceptron with
# only one output, so X should contain all data from all columns
# except the last one, and Y - data from the last column only.
X = []
Y = []
try:
    for line in dataset:
        X.append(line[0:-1])
        Y.append(line[-1])
except ValueError:
    sys.exit('Error: Wrong format of the CSV file.')
file.close()
```

- Writing data to lists `X` i `Y`, vector `Y` consists of only one column

Loading the training / test set – implementation (3)

```
# Store the size of the input vector (Nin) as a class property
self.Nin = len(X[0])

if self.Nin == 0:
    sys.exit('Error: zero-length training vector.')

# Normalize data, if requested
if normalize:
    X,Y = self.normalize(X, Y)

return X,Y
```

- Property assignment `Nin` (for later convenience)
- Optional call to normalization and return from the method

Initialization of connection weights

Initial initialization of weights with random values; in principle, the range should be selected so that the weights are normalized to 1, but in the first approximation we simply use the range [0, 1]

```
def initialize_weights(self):
    '''
    Initialize weights with a random numbers from range [0,1).
    Parameters:
        Nin - number of inputs of the perceptron (int)
    Sets:
        self.weights property (list)
    Returns:
        None
    '''
```

Connection weight initialization – three different implementations

- With NumPy

```
self.weights = np.random.random(self.Nin)
```

- Without NumPy

```
self.weights = [random.random() for i in range(self.Nin)]
```

- Without NumPy, „expanded” version

```
self.weights = []
for i in range(self.Nin):
    self.weights.append(random.random())
```

Division into training and validation set

The default division is 0.8 / 0.2; possibility of random data shuffling

```
def train_validation_split(self, X, Y, split=0.2, shuffle=False):
    '''
    Splits the input vectors into the train and validation ones.
    Parameters:
        X - X-vector to be splitted (list)
        Y - Y-vector to be splitted (list)
        split - splitting factor (float, optional)
        shuffle - data shuffling flag (bool, optional)
    Returns:
        splitted Xtrain, Ytrain, Xvalid, Yvalid (lists)
    '''
```

Division into training and validation set – implementation (1)

```
# Check if split is in a proper range, and adjust it if needed
if split > 0.9:
    print('Warning: Wrong split, adjusting to 0.9.')
    split = 0.9

if split < 0.1:
    print('Warning: Wrong split, adjusting to 0.1.')
    split = 0.1

data_size = len(X)
valid_data_size = int(split*data_size)
```

- Simple input control (thresholds 0.1 and 0.9 were chosen arbitrarily)
- Enumerating the size of files (no control if the validation set is non-empty)

Division into training and validation set – implementation (2)

```
# Generate list of random indexes indicating validation set
valid_random_indexes = sorted(list(np.random.choice(
                                   data_size,
                                   size=valid_data_size,
                                   replace=False)))
# Randomize data if requested
if shuffle:
    randomize = np.arange(len(X))
    np.random.shuffle(randomize)
    X = X[randomize]
    Y = Y[randomize]
```

- Drawing indices for the validating set
- Random data shuffling - optional

Division into training and validation set – implementation (3)

```
Xvalid, Yvalid, Xtrain, Ytrain = [], [], [], []
# Iterate over data and append them to the train or validation set
for i in range(data_size):
    if i in valid_random_indexes:
        Xvalid.append(X[i])
        Yvalid.append(Y[i])
    else:
        Xtrain.append(X[i])
        Ytrain.append(Y[i])

return Xtrain, Ytrain, Xvalid, Yvalid
```

- Joining data to appropriate files

Data normalization

Scale to interval [0, 1]; the scaling parameters will be stored for use in prediction

```
def normalize(self, X, Y):
    '''
    Normalizes the data and stores normalization parameters.
    Parameters:
        X - X-vector to normalize (list)
        Y - Y-vector to normalize (list)
    Sets:
        self.min_val - minimum value used in normalization
        self.max_val - maximum value used in normalization
    Returns:
        normalized vectors X, Y (lists)
    '''
```

Data normalization – implementation

```python
# Find minimum and maximum values of in both vectors.
# In subsequent runs, they will be used.
if not hasattr(self, 'min_val'):
    self.min_val = min(np.amin(X), np.amin(Y))
if not hasattr(self, 'max_val'):
    self.max_val = max(np.amax(X), np.amax(Y))

# Normalization formulas (vector operations)
X = (X - self.min_val) / (self.max_val - self.min_val)
Y = (Y - self.min_val) / (self.max_val - self.min_val)

return X, Y
```

- Implementation of typical scaling formulas, remembering the minimum and maximum value; in principle, scaling should be carried out separately for individual variables, here - for the sake of simplicity - scaling is common for the whole set

Inverse normalization

Rescaling previously normalized data back to the original data – for a clearer presentation of prediction results; the function takes multiple vectors to be denormalized at once

```
def unnormalize(self, *X):
    '''
    "Unnormalizes" vector(s), using previously
    determined minimum and maximum values
    Parameters:
        X - tuple of vector(s) to normalize (lists)
    Returns:
        tuple of vectors of "unnormalized" vector(s) (lists)
    '''
```

Inverse normalization – implementation

```
if hasattr(self, 'min_val') and hasattr(self, 'max_val'):
    Xout = []
    for Xsingle in X:
        # "Unnormalization" formula
        Xout.append([
                  self.min_val + i * (self.max_val - self.min_val)
                  for i in Xsingle])
else:
    # Cannot perform unnormalization, return original data
    print('Warning: Can not "unnormalize" data!')
    Xout = X

return Xout
```

- "Inverse" scaling based on inverted scaling patterns

Perceptron training

The most important method – gradient training based on the training set; computing RMSE for the training and validation set

```
def train(self, Xtrain, Ytrain, Xvalid, Yvalid):
    '''
    Trains the simple perceptron using the gradient method.
    Parameters:
        Xtrain - training (input) vector (list)
        Ytrain - training (output) vector (list)
        Xvalid - validating (input) vector (list)
        Yvalid - validating (output) vector (list)
    Returns:
        None
    '''
```

Perceptron training – implementation (1)

```
start_time = timer()

# Initialize weights
self.initialize_weights()

# Lists for storing RMSE for each epoch
RMSE_train = []
RMSE_valid = []
```

- Start timer to measure training time
- Initialization of empty lists on RMSE (root mean square error)

Perceptron training – implementation (2)

```
# Iterate over epochs
for epoch in range(self.epochs):
    print('Epoch = {}'.format(epoch+1))
    # Calculate output from the perceptron
    sumRMSE_train = 0
    for i in range(len(Xtrain)):
        # i - index of the input training pattern
        sumWeighted = 0
        for j in range(self.Nin):
            # j - index of the weight, for the given input pattern
            sumWeighted += self.weights[j]*Xtrain[i][j]
        Yout = self.f(sumWeighted)
```

- Loop over the epochs, calculating the output from the perceptron (weighted sum of inputs, subjected to the activation function)

Perceptron training – implementation (3)

```
# Calculate weights change
for j in range(self.Nin):
    self.weights[j] += \
        self.learning_rate * \
        self.fp(sumWeighted) * \
        (Ytrain[i]-Yout)*Xtrain[i][j]

# Calculate contribution from the current epoch
# to the RMS on training set
sumRMSE_train += (Yout-Ytrain[i])**2
```

- Continuation of the loop through epochs and training samples
- Calculation of the weight change based on the appropriate formula
- Calculating the RMSE derived from a single sample of the training set

Perceptron training – implementation (4)

```
# Calculate and append RMS on training set
RMSE_train.append(np.sqrt(sumRMSE_train / len(Xtrain)))

# Calculate RMS on validating set
sumRMSE_valid = 0
for i in range(len(Xvalid)):
    sumWeighted = 0
    for j in range(self.Nin):
        sumWeighted += self.weights[j]*Xvalid[i][j]
    Yout = self.f(sumWeighted)
    sumRMSE_valid += (Yout-Yvalid[i])**2
RMSE_valid.append(np.sqrt(sumRMSE_valid / len(Xvalid)))
```

- Continuation of the loop through the epochs, calculation of the cumulative RMSE from all samples in the epoch
- Computation of RMSE derived from validation data

Perceptron training – implementation (5)

```
print('RMSE (training set)   = {}'.format(RMSE_train[epoch]))
print('RMSE (validating set) = {}'.format(RMSE_valid[epoch]))

print('\nTraining completed in {:.2f} seconds.'\
      .format(timer()-start_time))

# Save plot
self.save_plot(RMSE_train, RMSE_valid)
```

- RMSE display at the end of each epoch
- End of training - displaying the time and triggering recording of graphs

Perceptron test

Checking the functioning of the perceptron on test data, i.e. data on which the perceptron has not been learned

```
def test(self, Xtest):
    '''
    Test of the trained perceptron.
    Parameters:
        Xtest - test vector (list)
    Returns:
        Y - output from the perceptron (list)
    '''
```

Perceptron test – implementation

```
Y = []
# Calculate output from the perceptron
for i in range(len(Xtest)):
    # i - index of the input pattern
    sumWeighted = 0
    for j in range(self.Nin):
        # j - index of the weight,
        # for the given input pattern
        sumWeighted += self.weights[j]*Xtest[i][j]
    Y.append(self.f(sumWeighted))

return Y
```

- It is basically a subset of the training method, limited to the perceptron output calculation

Mean squared error plot

Save to a file (and optionally display on the screen) the RMSE graph depending on the epoch (for the training and validation set)

```
def save_plot(self, RMSE_train, RMSE_valid,
                                  filename='loss.png',
                                  show=True):
    '''
    Plots / saves / shows RMSE.
    Parameters:
        RMSE_train - RMSE on training set (list)
        RMSE_valid - RMSE on validating set (list)
        filename - file name for save plot (str, optional)
        show - display or not the plot (bool, optional)
    Returns:
        None
    '''
```

Mean squared error plot – implementation

```
plt.plot(RMSE_train, label='RMS (training set)')
plt.plot(RMSE_valid, label='RMS (validating set)')
plt.legend()
plt.title('Results of training of simple perceptron')
plt.xlabel('Epoch')
plt.ylabel('RMSE')
plt.savefig(filename)
if show:
    plt.show()
print('RMSE plot has been saved to the file', filename)
```

- Standard plot drawing, using `matplotlib.pyplot`
- The format of the saved image file is determined by the file extension

Save model to YAML file

- All hyperparameters used to train the model as well as the trained connection weights will be saved
- `YAML` was chosen as a convenient and modern format for data storage

```
def save_model(self, filename):
    '''
    Saves the perceptron data into a YAML file.
    Parameters:
        filename - YAML file name (str)
    Returns:
        None
    '''
```

Save model to YAML file – implementation

```
data = {'nin':self.Nin,
        'epochs':self.epochs,
        'learning_rate':self.learning_rate,
        'activation':self.activation,
        'min_val':self.min_val,
        'max_val':self.max_val,
        'weights':self.weights}

with open(filename, 'w') as file:
    yaml.dump(data, file, default_flow_style=False)

print('Model has been saved to file', filename)
```

- A dictionary is constructed containing the class's properties, and then written to the YAML file

Read model from YAML file

Read and save as class properties of all hyperparameters used to train the model and weights, previously saved to the YAML file

```python
def load_model(self, filename):
    '''
    Loads the perceptron data from a YAML file.
    Parameters:
        filename - YAML file name (str)
    Returns:
        None
    '''
```

Read model from YAML file – implementation

```
try:
    with open(filename, 'r') as stream:
        data = yaml.load(stream, Loader=yaml.Loader)
except FileNotFoundError:
    sys.exit('Error: Model file does not exists.')
try:
    self.Nin = data['nin']
    self.epochs = data['epochs']
    self.learning_rate = data['learning_rate']
    self.activation = data['activation']
    self.min_val = data['min_val']
    self.max_val = data['max_val']
    self.weights = data['weights']
except KeyError:
    sys.exit('Error: Wrong format of the model file.')
```

Control file `train.py` (1)

```
from perceptron import simple_perceptron
import argparse

if __name__ == '__main__':

    # Hyperparameters defaults
    EPOCHS = 1000
    LEARNING_RATE = 0.1
    ACTIVATION = 'tanh'
    SPLIT = 0.2
    SHUFFLE = False
```

- Imports and default values of hyperparameters
- Modlue `argparse` will be used to read and parse arguments given on the command line

Control file `train.py` (2)

```
# Parse the arguments as parameters which can override defaults.
# They can be also read from a file (@file.par)
ap = argparse.ArgumentParser(fromfile_prefix_chars='@')
ap.add_argument('-d', '--dataset', help='Path to train dataset',
    metavar='filename', required=True)
ap.add_argument('-e', '--epochs', help='Number of epochs',
    type=int, default=EPOCHS, metavar='int')
ap.add_argument('-l', '--learning_rate', help='Learning rate',
    type=float, default=LEARNING_RATE, metavar='float')
ap.add_argument('-a', '--activation', help='Activation function',
    choices=['tanh', 'sigmoid', 'relu'], metavar='function',
    default=ACTIVATION)
```

Control file `train.py` (3)

```
ap.add_argument('-s', '--split', help='Train/validation split',
    type=float, default=SPLIT, metavar='float')
ap.add_argument('-f', '--shuffle', help='Enable data shuffle',
    action='store_true', default=SHUFFLE)
```

- Definition of possible input parameters to the program
- Only `--dataset` parameter is mandatory

Control file `train.py` (4)

```
args = vars(ap.parse_args())

input_filename = args['dataset']
epochs = args['epochs']
activation = args['activation']
learning_rate = args['learning_rate']
split = args['split']
shuffle = args['shuffle']
```

- Parsing the arguments
- More info: https://docs.python.org/3/library/argparse.html

Control file `train.py` (5)

```
# Create instance of the simple_perceptron class
p = simple_perceptron(epochs=epochs,
                      learning_rate=learning_rate,
                      activation=activation)

# Get the data from a CSV file
X,Y = p.read_input_data('train_data.csv')
```

- Create a perceptron object, using hyperparameter values
- Download and normalize data from a CSV file
- Normalization can be disabled by specifying an optional parameter `normalize=False`

Control file `train.py` (6)

```
# Split into the train and validation sets
Xtrain, Ytrain, Xvalid, Yvalid = \
    p.train_validation_split(X, Y, split=split,
                             shuffle=shuffle)

# Train of the perceptron
p.train(Xtrain, Ytrain, Xvalid, Yvalid)

# Save model to a file (with .model extension)
p.save_model(input_filename[:-3] + 'model')
```

- Division of the set into training and validation
- Perceptron training
- Save the trained model to a file

Sample training data – sum of two numbers

```
0,      0,      0
1.1,    1.1,    2.2
2,      2,      4
4,      4,      8
1,      9,      10
0.1,    0.1,    0.2
…
4,      5,      9
0.5,    0.5,    1.0
6,      4,      10
1,      1,      2
3,      5,      8
```

- The first two columns are the components of the sum
- The last column is the expected summation result
- The set consists of 20 data sets
- For clarity, some lines have been omitted

Control file `train.py` – help

```
> python train.py –h

usage: train.py [-h] -d filename [-e int] [-l float]
[-a function] [-s float] [-f]

optional arguments:
  -h, --help            show this help message and exit
  -d filename, --dataset filename      Path to train dataset
  -e int, --epochs int                 Number of epochs
  -l float, --learning_rate float      Learning rate
  -a function, --activation function   Activation function
  -s float, --split float              Train/validation split
  -f, --shuffle                        Enable data shuffle
```

Control file `train.py` – run

- Providing only the training set

  ```
  > python train.py --dataset train_data.csv
  ```

  or

  ```
  > python train.py -d train_data.csv
  ```

- Enter the number of epochs and activation functions

  ```
  > python train.py -d train_data.csv -e 100 -a relu
  ```

- Force data reshuffling

  ```
  > python train.py -d train_data.csv -f
  ```

- Provide the learning coefficient

  ```
  > python train.py -d train_data.csv -l 0.01
  ```

Control file `train.py` – example output

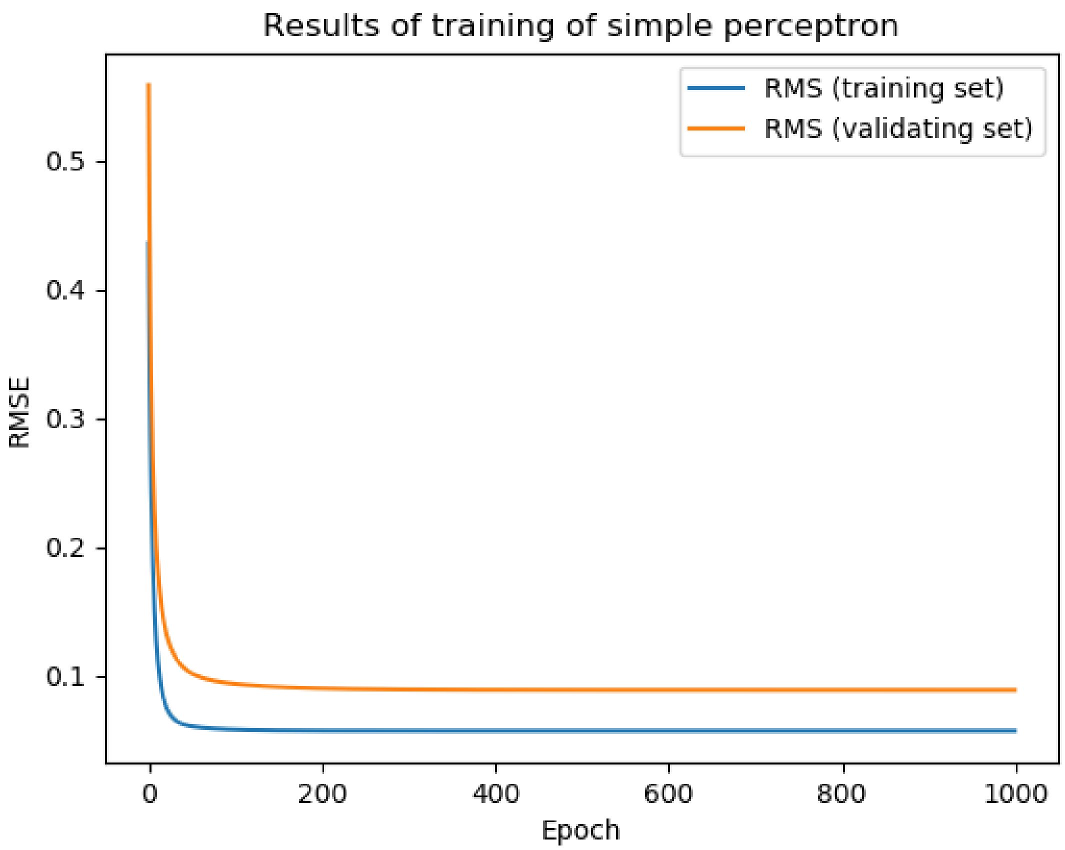
```
Epoch = 1
RMSE (training set)   = 0.4363197708554214
RMSE (validating set) = 0.558988231240701

...

Epoch = 1000
RMSE (training set)   = 0.05751239902930463 4
RMSE (validating set) = 0.08903784956318998

Training completed in 0.31 seconds.
RMSE plot has been saved to the file loss.png
Model has been saved to file train_data.model
```

Control file `train.py` – example graph



oś x: numer epoki
oś y: RMSE
tytuł: Wynik treningu perceptronu prostego
legenda: RMS (zbiór trenujący), RMS (zbiór walidujący)

Control file `predict.py` (1)

```
from perceptron import simple_perceptron
import argparse
import numpy as np

if __name__ == '__main__':
    # Parse the arguments.
    # They can be also read from a file (@file.par)
    ap = argparse.ArgumentParser(fromfile_prefix_chars='@')
    ap.add_argument('-t', '--testset', help='Path to test dataset',
        metavar='filename', required=True)
    ap.add_argument('-m', '--model', help='Path to trained model',
        metavar='filename', required=True)
    args = vars(ap.parse_args())

    test_dataset_filename = args['testset']
    model_filename = args['model']
```

Control file `predict.py` (2)

```
# Create instance of the simple_perceptron class
# (without any parameters, since only prediction is to be done)
p = simple_perceptron()

# Load previously saved model
p.load_model(model_filename)

# Read test data and test the perceptron with the trained weights
Xtest,Yexpected = p.read_input_data(test_dataset_filename)
Yout = p.test(Xtest)

Xtest, Yout, Yexpected = p.unnormalize(Xtest, Yout, Yexpected)
```

- Creating an object (this time without parameters), loading the model, test data and calling the test method; data denormalization

Control file `predict.py` (3)

```
print('Test results:')

for i in range(len(Yout)):
    # For summation only:
    print('{} + {} = {:.3f} (expected {})'
          .format(Xtest[i][0],
                  Xtest[i][1],
                  Yout[i],
                  Yexpected[i]))
```

- In case of a general problem and not a summation, just display the individual lists, for example like this:

```
print('obtained: {}, expected: {}'.format(Yout[i], Yexpected[i]))
```

Control file `predict.py` (4)

- Metrics RMSE and R^2:

```
# Scores: RMSE and R squared score
sse = sum((np.array(Yexpected) - np.array(Yout))**2)
tse = (len(Yexpected) - 1) * np.var(Yexpected, ddof=1)
rmse = np.sqrt(sse / len(Yout))
r2_score = 1 - (sse / tse)
print("\nRMSE score      = {:.2f}".format(rmse))
print("R squared score = {:.2f}".format(r2_score))
```

Sample test data – sum of two numbers

```
1.5,    1.5,    3
2.5,    1.5,    4
1,      6,      7
0.05,   0.1,    0.15
5,      0.1,    5.1
1,      3,      4
2,      4,      6
3,      3,      6
0.9,    5,      5.9
0,      8,      8
2,      0,      2
```

- The first two columns are the components of the sum
- The last column is the expected summation result
- The perceptron was not trained on these data

Control file `predict.py` – run

- Example call:

```
> python predict.py -t test_data.csv -m train_data.model
```

- Example output

```
Test results:
1.5 + 1.5 = 3.763 (expected 3.0)
2.5 + 1.5 = 4.642 (expected 4.0)
1.0 + 6.0 = 7.811 (expected 7.0)
…
3.0 + 3.0 = 6.592 (expected 6.0)
0.0 + 8.0 = 8.497 (expected 8.0)
2.0 + 0.0 = 2.107 (expected 2.0)
RMSE score      = 0.73
R squared score = 0.89
```

Summary

- The class that contains the required functionalities related to supervised learning of a simple perceptron has been successfully implemented
- This class was used to demonstrate learning the addition perceptron
- The learning results turned out to be correct and the learning itself was very fast
- With the *relu* activation function they would be much better (the problem is linear)