Pick some flowers

- **Topic:** machine learning, neural networks, supervised learning
- **Task A:**
  1. Implement the method to make the network learn. You already have a method for back-propagation, so you don't have to implement any math.
  2. Train the network using `iris-trainingdata.csv`. It contains real data on measurable characteristics of three different species of iris flowers. Then have the network recognize the species in `iris-testdata.csv`. How well does it do?
  3. See if changing the number of nodes or layers affects performance.
- **Task B:**
  1. Train the network to mimic a mathematical function. Data is provided for XOR and sin functions.
- **Template:** neuralnetwork.py
- **Data:** The 'iris' dataset contains measured features from 150 real iris plants of 3 diffrent species (50 each). The first 4 columns of data contain (in cm) sepal length, sepal width, petal length and petal width. The three last columns specify the species. If there is a 1 in the 5th column, the plant is Iris Setosa. The 6th column denotes Iris Versicolour and the 7th Iris Virginica.
  - iris-trainingdata.csv
  - iris-testdata.csv
  - xor-trainingdata.csv
  - xor-testdata.csv
  - sin-trainingdata.csv
  - sin-testdata.csv
- **Further reading:**
  - https://en.wikipedia.org/wiki/Artificial_neural_network
  - https://en.wikipedia.org/wiki/Iris_flower_data_set
  - https://archive.ics.uci.edu/ml/datasets/Iris

neuralnetwork.py

class neuralnetwork.**NeuralNetwork**(*inputsize*, *layersizes*, *outputsize*, *learning_rate=0.5*)

An artificial neural network. [source]

On creation, all weights are given random values between 0 and 1.

Parameters:
- **inputsize** (*int*) – number of input nodes
- **layersizes** (*array*) – Lists the number of nodes in each hidden layer. For example, [5, 6] will result in two hidden layers where the first one has 5 and the second 6 nodes.
- **outputsize** (*int*) – number of output nodes
- **learning_rate** (*float*) – the learning rate, should be between 0 and 1

activation(*signal*) [source]

The activation function.

Neural networks can use different types of activation functions. This function implements the sigmoid function

$$\varphi(x) = \frac{1}{1+e^{-x}}.$$

Parameters: **signal** (*array*) – input x either as a float or an array of floats
Returns: output $\varphi(x)$
Return type: float or array

activation_derivative(*signal_out*) [source]

Derivative of the `activation()` function.

The derivative of the sigmoid, $\varphi(x) = \frac{1}{1+e^{-x}}$ is

$$\varphi'(x) = -\frac{e^{-x}}{(1+e^{-x})^2}.$$

However, since $1 - \varphi(x) = -\frac{e^{-x}}{1+e^{-x}}$, the derivative can also be written nicely in terms of the output value φ instead of the input x as

$$\varphi'(x) = \varphi(x)[1 - \varphi(x)].$$

Parameters: **signal_out** (*array*) – sigmoid value $\varphi(x)$ either as a float or an array of floats
Returns: sigmoid derivative $\varphi'(x)$
Return type: float or array

backpropagate(*target*, *output*) [source]

Compares the output to the target and adjusts weights to drive the output towards the target value.

When this function is called, the weights of the network are slightly adjusted so that the output of the network will resemble the given target somewhat better. When this function is repeatedly called with different learning samples, the network gradually adjusts to reproduce the wanted results.

Mathematically, backpropagation is a one-step gradient search for optimal weights $w_{i,j}^{n\to n+1}$. If E is the error between the network output and the known result, the function calculates the derivatives $\frac{\partial E}{\partial w_{i,j}^{n\to n+1}}$ and adjusts the weights by

$$\Delta w_{i,j}^{n\to n+1} = -\eta \frac{\partial E}{\partial w_{i,j}^{n\to n+1}}.$$

This means the weights are all adjusted in the direction that makes the error diminish.

Here η is the learning rate which controls how much the weights are adjusted. Typically, it should be between 0 and 1.

Parameters:
- **target** (*array*) – the known correct answer to some input
- **output** (*array*) – the answer the network gives for the same input

feedforward(*input*) [source]

Sends the signal through the network.

In other words, produces output for the given input.

The neurons in the input layer receive the given input x as their activation signal. If the signal a neuron receives is strong enough, the neuron activates and sends a new signal y to the neurons in the next layer.

To simulate the strength of the connection between neurons, the signal a neuron sends is multiplied by a coupling factor called a weight, w. (If a weight is 0, there is no connection.) Neurons in layers other than the input layer receive signals from several neurons, and so for them the total activation signal is the sum of the weighted signals. If this sum of signals is strong enough, this neuron activates and sends a signal forward, etc.

In this manner, the signal proceeds through the network. The signal sent by the final layer is the final output of the whole network.

To be more precise, let us write the activation signal for neuron i in layer n as x_i^n. Activation of this neuron is represented by the `activation()` function, which changes rapidly from 0 to 1 as the signal goes from negative to positive values. (So if $x_i^n > 0$, the neuron activates.) The activation output of this neuron is therefore

$$y_i^n = \varphi(x_i^n).$$

The signal that is sent to neuron j in layer $n+1$ is this output multiplied by the weight that connects the two neurons,

$$w_{i,j}^{n\to n+1} y_i^n.$$

The total activation signal for neuron j is the sum of all signals it receives from layer n,

$$x_j^{n+1} = \sum_i w_{i,j}^{n\to n+1} y_i^n.$$

This summation can be written efficiently with matrices. Define

- input vector to layer n as $X^n = [x_0^n, x_1^n, \ldots]^T$
- output vector from layer n as $Y^n = [y_0^n, y_1^n, \ldots]^T$
- weight matrix $W^{n\to n+1}$ with elements $w_{i,j}^{n\to n+1}$.

Then neutron activation in layer n is calculated with

$$Y^n = \varphi(X^n)$$

and the activation signals for layer $n+1$ are obtained with

$$X^{n+1} = W^{n\to n+1} Y^n.$$

Parameters: **input** (*array*) – input (for the input layer)
Returns: output (from the output layer)
Return type: array

nudge(*amount*) [source]

Randomly change weights.

If learning gets stuck in a local optimum, one can try this to escape.

Parameters: **amount** (*float*) – the maximum change allowed in each weight.

read_weights(*filename='weights.txt'*) [source]

Reads network weights from a file.

Parameters: **filename** (*str*) – name of the file to read

save_weights(*filename='weights.txt'*) [source]

Print the current network weights in a file.

Parameters: **filename** (*str*) – name of the file to write

train(*input*, *target*) [source]

Trains the network.

The network takes the given input, calculates an output and compares the result to the given target output using `NeuralNetwork.backpropagate()`.

Calling this function several times with a large group of input – target pairs will make the network learn to reproduce the given target results.

> **Note**
> This function is incomplete!

Parameters:
- **input** (*array*) – input to evaluate
- **target** (*array*) – the correct answer

visualize() [source]

Draws a visual representation of the network.

Each node is represented as a circle and each layer as a row of circles. Input nodes are on the left, and output nodes are on the right.

Weights between nodes are represented by arrows. Positive weights are red while negative ones are blue. The thicker the arrow, the larger the absolute value of the weight.

neuralnetwork.**check_performance**(*nn*, *inputs*, *targets*, *plot=False*, *printout=False*, *classify=False*) [source]

Checks how well a neural network has been trained.

The inputs are given to the neural network and the results are compared to the target results (correct answers).

The function may print or plot the results if required. It always returns the sum of squares error

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j} - t_{i,j})^2$$

where

- N is the amount of test data (number of inputs and targets),
- M is the length of the output vector,
- $y_{i,j}$ is the jth component of the ith output and
- $t_{i,j}$ is the jth component of the ith target

Parameters:
- **nn** (*NeuralNetwork*) – the network to evaluate
- **inputs** (*list*) – inputs to test
- **targets** (*list*) – target outputs to compare to
- **plot** (*bool*) – If True, the results are visualized.
- **printout** (*bool*) – If True, the results are printed on screen.
- **classify** (*bool*) – If True, the network is used for classifying results using `pick_class()`.

Returns: the sum of squares error
Return type: float

neuralnetwork.**main**(*input_size*, *output_size*, *layers=[5]*, *traincycles=5000*, *trainfile='trainingdata.csv'*, *testfile='testdata.csv'*, *classify=False*) [source]

The main program.

Creates a network, trains is using training data, and tests the performance against separate test data.

Parameters:
- **input_size** (*int*) – number of input neurons
- **output_size** (*int*) – number of output neurons
- **layers** (*list*) – number of neurons in each hidden layer
- **traincycles** (*int*) – how many times the training data is fed to the network
- **trainfile** (*str*) – name of the file containing the training data
- **testfile** (*str*) – name of the file containing the test data
- **classify** (*bool*) – If True, the network is used for classifying results using `pick_class()`.

neuralnetwork.**pick_class**(*output*) [source]

Chooses the most likely class from the given output.

Neural networks are often used to classify data. For instance, if we want to sort data instances in three classes, we can use a network with three outputs. Each output corresponds to a class and the output value (between 0 and 1) represents how likely the instance is from that class, according to the network. If the output is [1,0,0], the instance is certainly from the 1st class. If the output is [0.1, 0.7, 0.1], the instance is likely from the 2nd class.

This function looks at an output vector and gives the index of the class with the highest value. For [1,0,0], the function return 0. For [0.1, 0.7, 0.1], the function return 1. If there is a tie, the function returns the smallest of the tied indices.

Parameters: **output** (*array*) – neural network output
Returns: index of the most likely class
Return type: int

neuralnetwork.**print_progress**(*step*, *total*) [source]

Prints a progress bar.

Parameters:
- **step** (*int*) – progress counter
- **total** (*int*) – counter at completion