Artificial Intelligence – Lab assignment 4

UNIZG FER, academic year 2018/19

Handed out: 28.5.2019. Due: 9.6.2019. at 23:59

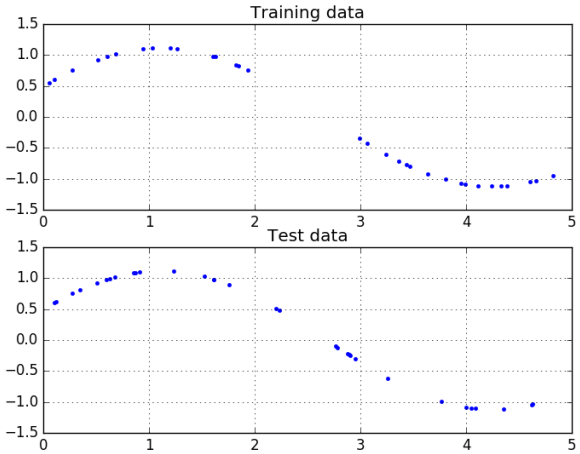
1 Introduction

The topic of this lab assignment are neural networks and the genetic algorithm. The task is to write a program for training a neural network (i.e., determining its weights) using a genetic algorithm.

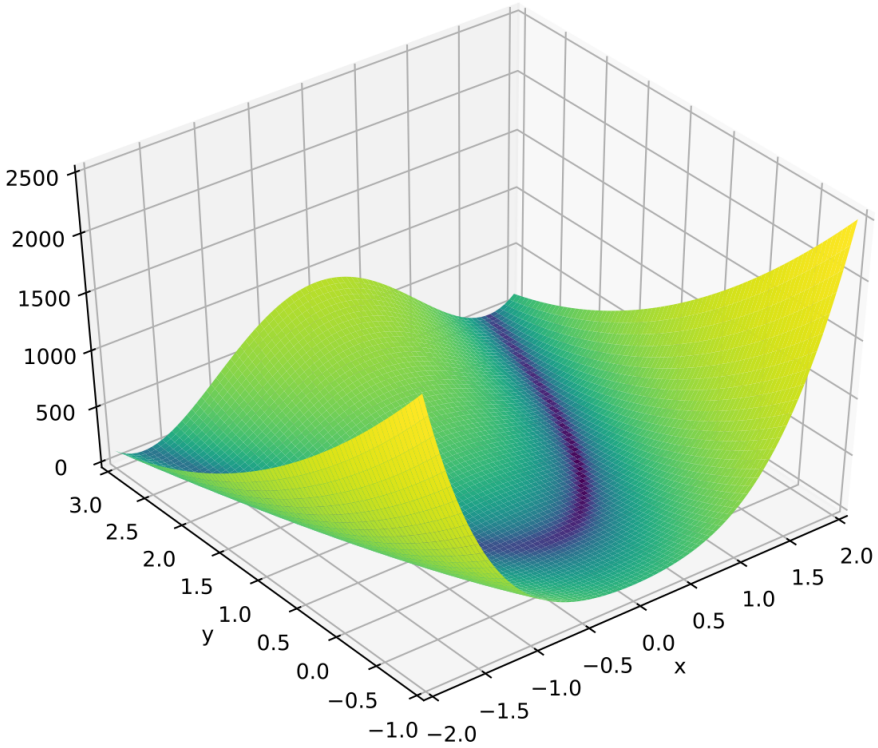
A neural network of the type used in this assignment is typically trained with the backpropagation algorithm. However, training a network using a genetic algorithm is applicable to various types of networks and has additional advantages, such as the wider breadth of space exploration than the standard backpropagation. The main idea is to build a population of neural network instances (each individual corresponds to a single neural network instance and compactly encodes all its weights). We then use the genetic algorithm (which applies the selection, crossover, and mutation operators) to optimize the weights with respect to the error of the network on the training set.

In this assignment the network architecture is fixed throughout the optimization with the genetic algorithm; in a more general case the procedure may also be used to optimize the network structure along with its weights. After the network has been trained, we must check how well it performs on an unseen sample. To this end we will use a test set with different samples than he ones present in the training set.

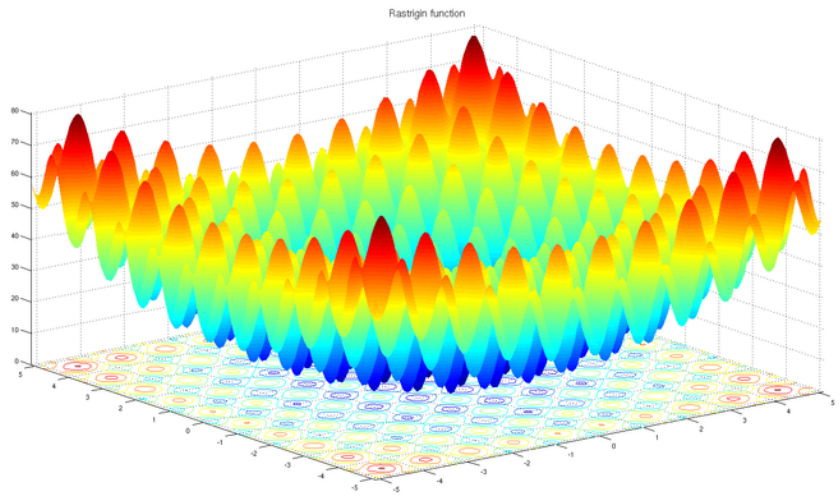
You will solve a regression problem of approximating an unknown function given a sample of its variables and values. The first function you will optimize is the sine curve $f(x) = sin(\alpha x + \phi) \cdot \beta + \gamma$, in the interval of values from 0 to 5. The graph of sample values of the sine function found in the train and test sets can be seen as follows:



The other functions we will approximate are the Rosenbrock function:



and the Rastrigin function:



As input for the training set you will not receive the whole area of definition of the functions, but just a small manageable samle.

2 Code organisation

As in the previous lab assignments, a part of pre-written code is available. In the case of this lab assignment, the code is split in seven python files which represent different modules of the final implementation of the solution.

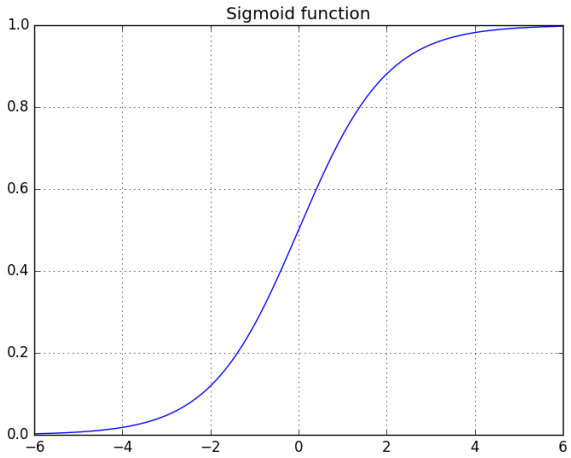
There are two modules you don't need to read - one of them being the dataLoader.py, which simply serves the purpose of loading the data from the text files containing the data in the format of the lab assignment. The module contains just one function - *loadFrom*, which for a given path to a text file returns the sampled function values.

The second module you can ignore is the plotter.py, which uses the matplotlib library to plot the itermediate results of the optimisation by the genetic algorithm. You should install the matplotlib package in order to visualise the process of learning. The package is available on the following address: matplotlib, but it is a part of most standard Python distributions - so you should have it installed already.

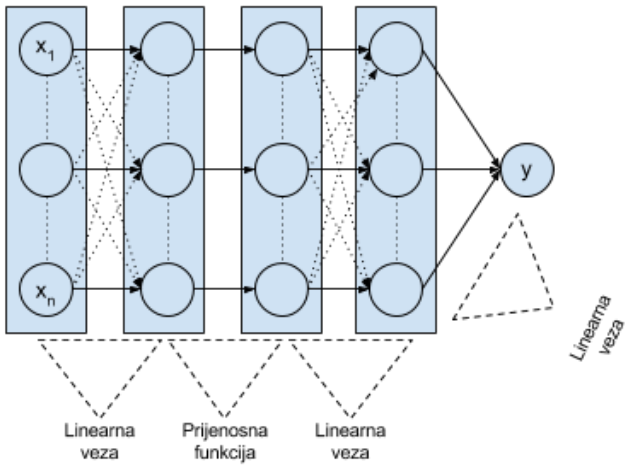
The remaining modules serve for the implementation of the neural network, the genetic algorithm and the main function that conects and runs everything. The idea is that the implementation of the genetic algorithm and the neural network are clearly separated, and that the both of them can function fully independently of one another - never knowing the other one exists.

The implementation of the neural network is divided between three files. transferFunctions.py, contains various standard transfer functions for neural networks. You are not obligated to use any of them (except the sigmoid), however we urge you to explore the differences while using them.

The transfer functions keep the dimensionality of the input ($|\vec{x}| = |f(\vec{x})|$), but they modify the values of the elements of the vectors in a non-linear fashion. During the lectures, we covered the sigmoid transfer function: $sigm(x) = \frac{1}{1+e^{-x}}$, whose function graph you can see as follows:



A fully conected neural network can be viewed as a series of layers, where each of the layers is a series of neurons. An example of this can be seen in the following image:



The layers are interchangeably connected by a linear combination of the input values and the according weights (which take into consideration all the inputs of the previous layer and project to all the values of the following layer) and a non-linear transfer layer, exemplified by the sigmoid transfer function.

In the networkLayers.py file an abstract class *NetworkLayer* is defined by some general methods which are necessary for a layer of a neural network. As we don't use backpropagation in the scope of this lab assignment, we will just define the forward pass in the *output* method.

The rest of the method serves as a simplification for the training of the genetic algorithm - the neural network is basically a list of weights, and our task is to find the optimal combination of weights given a certain target function. The implementation of this optimization will more or less be changing the weight vector with the genetic algorithm, feeding it to the neural network to calculate the error, and using the error to evaluate the fitness of the weights.

In order to evaluate each unit from the genetic algorithm, we need to be able to set the weights to our network architecture, which is done with the method *setWeights*, which distributes the weights in the layer. In order to initialize the units of the genetic algorithm, we need to know the amount of weights, and the amount of weights for each layer of the neural network is defined in the method *size*.

In the neuralNet.py you will need to complete the code for the implementation of the neural network. In the geneticAlgorithm.py you will write the training code for the genetic algorithm. In the runner.py you will tweak the hyperparameters and initialize the architecture of the neural network to your liking. A more detailed description follows in the respective sections for each subproblem.

In addition with the matplotlib library, we also use the numpy library - an extremely powerful tool for matrix computation, linear algebra and randomized operations among others. A short introduction to the numpy library and its sufficient functionalities in order to complete the lab assignment is given as an Ipython notebook along with the lab assignment here. It is important to note that most of the vectors in code are supposed to be of the numpy multidimensional array type.

3 Tasks

Problem 1: Neural Network (20% points)

The first subtask in the lab assignment is to complete the implementation of the neural network in neuralNet.py. The methods you need to complete implement the basic logic of forward propagation of the neural network and the calculation of the error depending on the actual values of the target function.

As a reminder, the training set is split as a series of input values X, and a series of target values $y = f(X)$. A forward pass is a series of operations where from the set of input values X with respect to the weights W of the neural network, we calculate a predicted estimate of the target function $y^{predicted}$ for each instsance X of the input.

When calculatng the error, we compare the vecor of predicted values with the vector of real values given in the training set in order to see how wrong we were in our predictions. In the context of the lab assignment, the error funcion you should implement is the MSE (Mean Square Error), which we calculate as follows:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i^{predicted} - y_i^{true})^2$$

Where $y^{predicted}$ is the aforementioned approximated vector of function values, while y^{true} is the vector containing the true values from the training set.

The *forwardStep* is supposed to combine forward propagation and error calculation and for a given vector of input values X produce a single floating point error (MSE) with respect to the true target values Y.

Problem 2: Genetic algorithm (40% points)

The second subtask in the lab assignment is the implementation of the genetic algorithm. In the geneticAlgorithm.py a constructor of the *GenericAlgorithm* is already imple- mented for you, but there is still a lot of work to do!

Learning of the neural network is supposed to be implemented with a generational genetic algorithm with built in elitism, which selects the best (or the first few best) units and places them in the next generation. As a representation of the chromosome you should use a numpy array of floating point values with the length equal to the amount of weights in the neural network.

- **The crossover operator** should be implemented as the mean of the parent vectors' values.
- **The mutation operator** should be implemented by adding a random Gaussian noise $N(0, K)$, where K is the standard deviation of the Gaussian distribution, set by the *mutationScale* hyperparameter. You should only mutate p percent weights at once (where p is an additional hyperparameter).
- The **selection operator** should be the fitness proportional selection strategy.

Problem 3: Function optimization and hyperparameter exploration (40% points)

After you've successfully implemented the genetic algorithm and the neural network, you should complete the definition of the neural network architecture in the runner.py file by

adding layers to the neural network. Start with a single layer neural network ($1 \times n \times 1$ architecture), where the dimensions of the input and output layer are predefined for you.

In the file, you will also find predefined paths to various training and testing datasets of various functions. For starters, your genetic algorithm should be able to relatively easily solve the sine problem. However - before you do that you will still need to tweak the hyperparameters (as well as the number of neurons in the hidden layer).

Think about how each of the hyperparameters influences the execution of your optimization. **The solution for this part of the assignment that doesn't adequatly approximate the sine function won't be awarded any points in this problem.** A sample of an adequate sine wave approximation, as well as samples for other approximated functions can be seen at the end of the instructions.

After approximating the sine function well enough, try your luck with the two-dimensional Rastrigin and Rosenbrock functions.

- How does the optimization behave when you increase the number of layers of the neural network?
- How does it behave when you increase the depth of the network?
- Which, according to you, would be the optimal network architecture? Why?
- How does each of the genetic algorithm hyperparameters affect the optimisation process?

Answer these questions and elaborate your answers at the submission of the lab assignment. Keep track of the errors you get while optimizing all of the functions, and the hyperparameters you got those errors for so you can reproduce your best results at the submission.

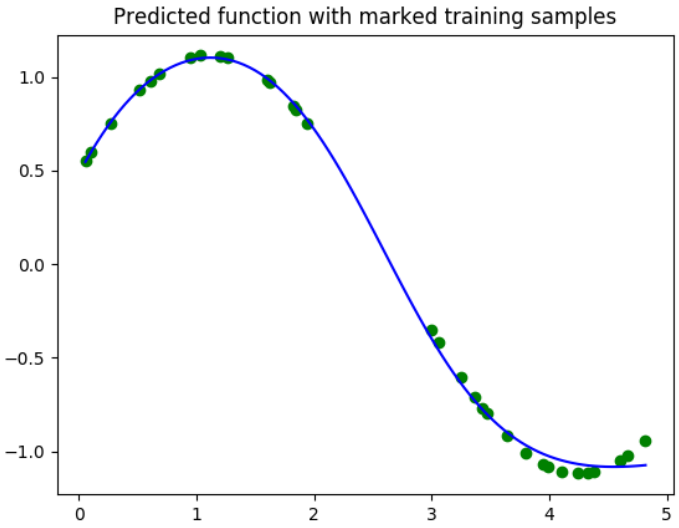
During the training of the algorithm, occasionaly you will see plots of the visualisation of your current best approximation and the true values of the functions. You can control the frequency of the printed and plotted values by setting the variables *print_every* and *plot_every*. You need to close the plots to continue execution of the algorithm.

4 Approximation samples

After a fruitful hyperparameter modification, you can see printouts and plots (on the training set) of adequately optimized functions as follows:

Sine function

```
Error at iteration 1000 = 0.003803
Error at iteration 2000 = 0.002901
Error at iteration 4000 = 0.002151
Error at iteration 6000 = 0.001792
Error at iteration 8000 = 0.001636
Error at iteration 10000 = 0.001443
Error on test set:  0.00092272888201
```



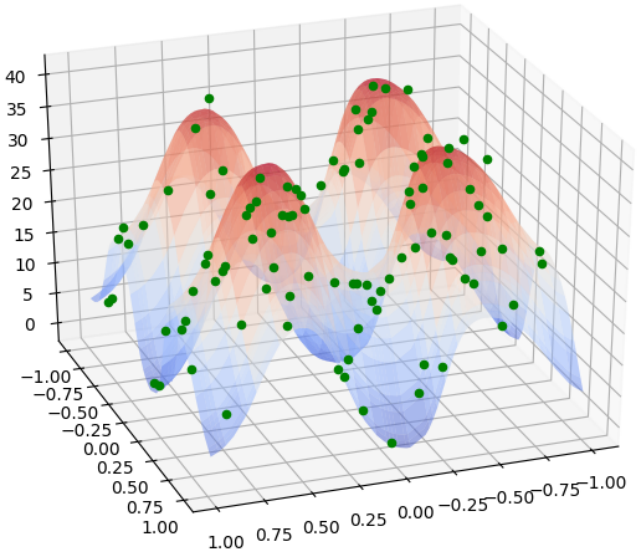
Slika 1: Sine function

Rastrigin function

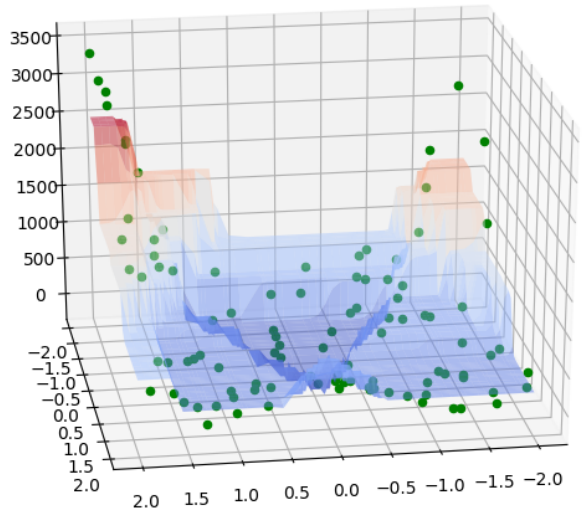
```
Error at iteration 1000 = 56.064648
Error at iteration 2000 = 20.680325
Error at iteration 4000 = 2.596907
Error at iteration 6000 = 1.697376
Error at iteration 8000 = 1.581420
Error at iteration 10000 = 1.457434
Error on test set:  2.06605975288
```

Rosenbrock function

```
Error at iteration 1000 = 146096.470522
Error at iteration 2000 = 132394.253083
Error at iteration 4000 = 80059.521281
Error at iteration 6000 = 73484.380331
Error at iteration 8000 = 67925.648229
Error at iteration 10000 = 59211.889461
Error on test set:  90720.1330711
```



Slika 2: Rastrigin function



Slika 3: Rosenbrock function