Introduction to Artificial Intelligence – Programming assignment 4

UNIZG FER, academic year 2022/23.

Handed out: February 27, 2023 Due: June 1, 2023 at 23:59.

Genetic algorithm optimized neural networks (24 points)

The topics of this lab assignment are neural networks and the generational genetic algorithm. The task of the lab assignment is to implement the code for learning an artificial neural network by means of a genetic algorithm. Neural networks are traditionally learned with the backpropagation algorithm. However, learning a neural network is an optimization problem which can also be tackled by a genetic algorithm. The core idea of our approach will be to define a population of neural networks (where each chromosome will be an instsance of a neural network), and then to optimize the network weight values with respect to the error of a neural network on the training set. Once optimized, the learned network should be evaluated on a thus far unseen test set.

Your assignment will be to approximate functions based on a sample of input variables and output values (also known as regression). The first function we will optimize will be a sine wave $f(x) = \sin(\alpha x + \phi) \cdot \beta + \gamma$, on the value interval from 0 to 5. The graph containing samples from the training can be seen in Fig 1.

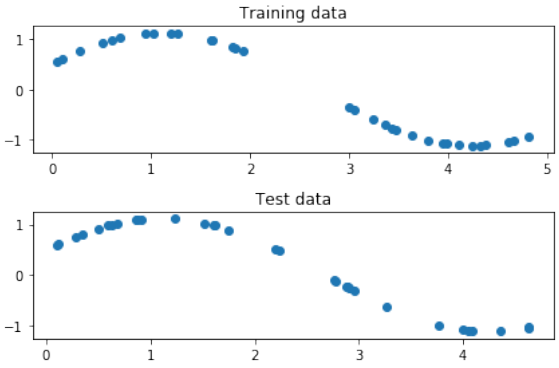

Figure 1: The sine wave dataset

The remaining functions we will optimize are the Rosenbrock function (Fig. 2) and the Rastrigin function (Fig. 3).

Of course, our training and test data won't contain the entire domains of the previous functions but rather a small subset of sampled values.

Exceptionally for this lab assignment, your code **may use** the following external libraries: **numpy** in case you are writing your lab assignment in Pytho, **eigen** in case `c++`

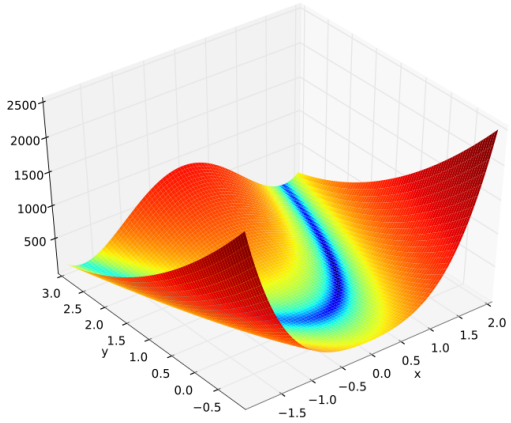

Figure 2: Rosenbrock function

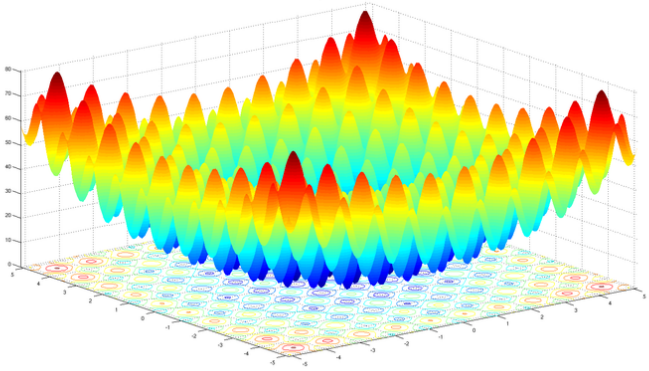

Figure 3: Rastrigin function

is your language of choice and **Apache Commons Math** if you are coding in Java. But, as the mathematical operations you need to implement are only the dot product and the logistic sigmoid, we recommend you try and implement them yourselves without the use of external libraries. For random sampling from the normal distribution use the libraries which are part of the standard libraries or **numpy** in Python. If you aren't sure whether you may use a specific library, check whether it is part of the standard library package for that programming language.

The evaluation of your implementation will be conducted using *autograder*. It is expected that your know how to run your solution with *autograder* in front of the Teaching Assistant during the examinaton. "Autograder instructions" were written by taking into account that **we** will run your solution, which we won't do this year. **You** need to make sure that your solution is runnable and can be evaluated with *autograder* in front of the Teaching Assistant. Together with the lab assignment, you will get access to all test samples.

Deadline for solution submission on Moodle **does not include** the last minute in the day, so make sure that you upload the archive with your solution **before** 23:59. Submissions uploaded exactly at 23:59 or after will be considered as late submissions.

The total number of points in this lab assignment is 24. The number of points that you obtain will be scaled by percentage to 7.5 points.

1. Data loading

So that we can apply our algorithm to various tasks, we will first define the format of the dataset. We will once again use the `csv` format from the previous lab assignment to store our datasets. Files in the `csv` format contain values separated by commas. Each line of the file will contain the same amount of values. In our case, the values are the features and the target for our machine learning algorithm. The first line of the file will always be the `header` containing the names of the values in each column.

An example of the first three rows of the sine wave dataset file can be seen as follows (the first line contains the header):

```
x,y
3.469,-0.795
1.626,0.971
```

As it is often the case in machine learning, the last column will contain the target value (response variable). The remaining columns will contain input features. For this lab assignment, we will always have only one **real valued target variable**. All input features will also be real valued.

2. Neural network (12 points)

In the first part of the lab assignment your task will be to implement the feedforward neural network and its forward pass. In the scope of this lab assignment, you won't need to implement the backpropagation algorithm as the weights (parameters) of our neural network will be learned by the genetic algorithm.

For simplicity, in scope of this lab assignment it is only necessary that you implement the following three neural network configurations:

1. input $\rightarrow$ 5 $\rightarrow$ σ $\rightarrow$ output (`5s`)
2. input $\rightarrow$ 20 $\rightarrow$ σ $\rightarrow$ output (`20s`)
3. input $\rightarrow$ 5 $\rightarrow$ σ $\rightarrow$ 5 $\rightarrow$ σ $\rightarrow$ output (`5s5s`)

Where *input* is the input data dimension, which you need to determine from the input file (number of columns - 1), and *output* is the dimension of the target variable (which will always be 1). Values 5 and 20 indicate the size of the neural network hidden layer, while σ indicates the logistic sigmoid transfer function. The initial values of all weights of the neural network should be sampled from the normal distribution with standard deviation of 0.01. The neural network configuration will be passed to your code through one of the program arguments. The only configurations which will be evaluated by the autograder will be $5s$, $20s$ and $5s5s$.

The network (1) contains two linear transformations – matrix multiplications followed by addition of a bias vector, and one application of a transfer function. The input data is first projected to the hidden state of size 5, then the sigmoid function is applied elementwise and finally the second linear transformation projects the data to the dimensionality of the output value. So, in the last step, after applying the final linear transformation to get the output of the network, there should be **no** sigmoid function. The visualized network (1) can be seen in Fig. 4.

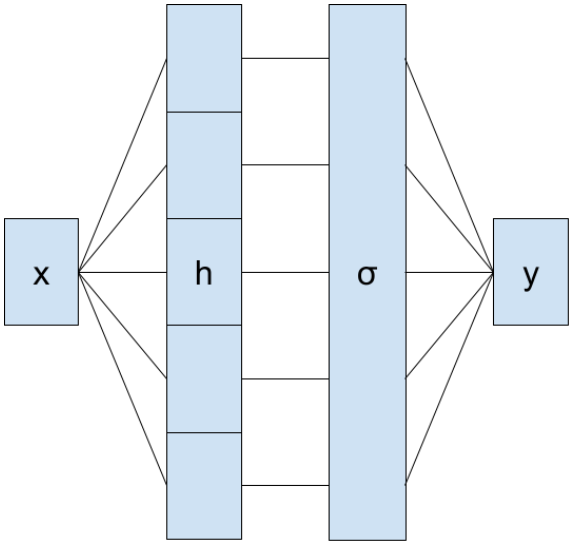

Figure 4: Single layer feedforward network (5s)

The forward pass of your neural network should compute the predicted output values for an input dataset

$$\hat{y} = \text{NN}(X), \qquad X \in \underbrace{\{x_1, x_2, \ldots, x_N\}}_{\text{Dataset instances}}$$

where N is the size of the train or test dataset.

Except the forward pass, your neural network implementation should also be able to compute the mean squared error from the correct values Y:

$$\text{err} = \frac{1}{N}\sum_{i}^{N}(y_i - \hat{y}_i)^2 = \frac{1}{N}\sum_{i}^{N}(y_i - NN(x_i))^2$$

The mean squared error will be used when computing the fitness of each chromosome in the genetic algorithm.

Although in the scope of the lab assignment you need to implement only the previously defined neural network architectures, we suggest you design your implementation in such a way that the number of layers, as well as their hidden dimensions, can be arbitrarily defined so you can investigate the influence of the network architecure on the results your self. When implementing your network, we suggest you design the code in such a way so that it will be easy for you to retrieve and then set the weights of the network, as you will frequently update them through the genetic algorithm.

3. Genetic algorithm (12 points)

Once we've implemented the neural network, our next assignment is to implement the genetic algorithm which will optimize the weights of our neural network. In our case, the chromosomes of the genetic algorithm will be real-valued – they will contain the weights of the neural network. You should define the fitness of each chromosome arbitrarily as a function of the mean squared error of that chromosome's network on the training set. Pay attention to the fact that the fitness of a chromosome should be greater the better the chromosome is, while the reverse is true for the mean squared error.

The components of the genetic algorithm should be implemented as follows:

- **Elitism**: the best (or multiple best) units should be carried over to the next generation;
- **Crossover**: the crossover operator should be the arithmetic mean;
- **Mutation**: the mutation operator should be implemented as Gaussian noise, in a way that Gaussian noise should be added to each weight chromosome. The Gaussian noise should be sampled from the normal distribution with standard deviation K. Each weight of the chromosome should be mutated with probability p.
- **Selection**: the selection operator should be fitness proportional selection.

Genetic algorithms contain a large number of hyperparameters, which will be passed to your implementation via the command line. Concretely, the hyperparameters of the genetic algorithm of interest in scope of the lab assignment are: (1) the **population size** (*popsize*) of the genetic algorithm, (2) **elitism** (*elitism*) – the number of best chromosomes which should be preserved between iterations, (3) the **mutation probability** (p) for each element of the chromosome, (4) the **mutation scale** (K), representing the standard deviation of Gaussian noise and (5) the **number of iterations** (*iter*) of the genetic algorithm. Along with the paths to the train and test datasets and the neural network architecture, these arguments will be passed to your code via the command line.

Every 2000 itierations (generations) your genetic algorithm should print out the mean squared error of the **best** chromosome on the training set. At the end of the optimization process (after *iter* generations), print out the mean squared error of the best chromosome on the test set.

The output should be formatted as in the following example of the $5s$ network on the sine wave dataset:

```
python solution.py --train sine_train.txt --test sine_test.txt --nn 5s --popsize
    10 --elitism 1 --p 0.1 --K 0.1 --iter 10000
[Train error @2000]: 0.002901
[Train error @4000]: 0.002151
[Train error @6000]: 0.001792
[Train error @8000]: 0.001636
[Train error @10000]: 0.001443
[Test error]: 0.000922
```

Autograder instructions

The instructions are given below about uploaded archive structure. Detailed instructions on how to run *autograder* using given structure can be found in `autograder.zip` archive in file `README.md`.

Uploaded archive structure

The archive that you will upload to Moodle **has to** be named `JMBAG.zip`, while the structure of the unpacked archive **has to** be as in the following example (the following example is for solutions written in Python, while examples for other languages are given in subsequent sections):

```
|JMBAG.zip
|-- lab4py
|----solution.py [!]
|----neuralnet.py (optional)
|----...
```

Uploaded archives that are not structured in the given format will **not be graded**. Your code must be able to execute with the following arguments from command line:

1. Path to the training dataset (`--train`)
2. Path to the test dataset (`--test`)
3. The neural network architecture (`--nn`)
4. The population size of the genetic algorithm (`--popsize`)
5. The elitism of the genetic algorithm (`--elitism`)
6. The mutation probability of each chromosome element (`--p`)
7. The standard deviation of the mutation Gaussian noise (`--K`)
8. The number of genetic algorithm iterations (`--iter`)

All arguments will always be passed to each execution of your solution. Due to the stochasticity of the genetic algorithm execution, your output will not be matched exactly. The autograder will only liberally check whether your solution successfully reaches a low enough error on repeated executions.

Your code will be executed on linux. This does not affect your code in any way except if you hardcode the paths to files (which in any way, **you should not do**). Your code **may use** only the aforementioned external libraries. Use the `UTF-8` encoding for all your source code files.

An example of running your code for Python:

```
>>> python solution.py --train sine_train.txt --test sine_test.txt --nn 5s --popsize
10 --elitism 1 --p 0.1 --K 0.1 --iter 10000
```

Instructions for Python

The entry point for your code **must** be in the `solution.py` file. You can structure the rest of your code using additional files and folders, or you can leave all of it inside `solution.py` file. Your code will always be executed from the folder of your assignment (`lab4py`).

The directory structure and execution example can be seen at the end of the previous chapter. Your code will be executed with python version `3.7.4`.

Instructions for Java

Along with the lab assignment, we will publish a project template which should be imported in your IDE. The structure inside your archive `JMBAG.zip` is defined in the template and has to be as in the following example:

```
|JMBAG.zip
|--lab4java
|----src
|------main.java.ui
|--------Solution.java [!]
|--------NeuralNet.java (optional)
|--------...
|----target
|----pom.xml
```

The entry point for your code **must** be in the file `Solution.java`. You can structure the rest of the code using additional files and folders, or you can leave all of it inside the `Solution.java` file. Your code will be compiled using Maven.

An example of running your code with the autograder (from the `lab4java` folder):

```
>>> mvn compile
>>> java -cp target/classes ui.Solution --train sine_train.txt --test
    sine_test.txt --nn 5s --popsize 10 --elitism 1 --p 0.1 --K 0.1 --iter 10000
```

Info regarding the Maven and Java versions:

```
>>> mvn -version
Apache Maven 3.6.3
Maven home: /opt/maven
Java version: 15.0.2, vendor: Oracle Corporation, runtime: /opt/jdk-15.0.2
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-139-generic", arch: "amd64", family: "unix"

>>> java -version
openjdk version "15.0.2" 2021-01-19
OpenJDK Runtime Environment (build 15.0.2+7-27)
OpenJDK 64-Bit Server VM (build 15.0.2+7-27, mixed mode, sharing)
```

Important: please check that your implementation can be compiled with the predefined `pom.xml` file.

Instructions for C++

The structure inside your archive `JMBAG.zip` has to be as in the following example:

```
|JMBAG.zip
|--lab4cpp
|----solution.cpp [!]
|----neuralnet.cpp (optional)
|----neuralnet.h (optional)
|----Makefile (optional)
|----...
```

If your submitted archive does not contain a `Makefile`, we will use the Makefile template available along with the assignment. **If** you submit a `Makefile` in the archive (which we don't suggest, unless you really know what you're doing), we expect it to work.

An example of compiling and running your code with the autograder (from the `lab4cpp` folder):

```
>>> make
>>> ./solution --train sine_train.txt --test sine_test.txt --nn 5s --popsize 10
    --elitism 1 --p 0.1 --K 0.1 --iter 10000
```

Info regarding the gcc version:

```
>>> gcc --version
gcc (Ubuntu 9.3.0-11ubuntu0~18.04.1) 9.3.0
```

Output examples

Each output example will also contain the running command that was used to produce that output. Running command assumes Python implementation, but the arguments will be the same for other languages as well.

1. Sine wave

```
>>> python solution.py --train sine_train.txt --test sine_test.txt --nn 5s
    --popsize 10 --elitism 1 --p 0.1 --K 0.1 --iter 10000

[Train error @2000]: 0.002106
[Train error @4000]: 0.001565
[Train error @6000]: 0.001097
[Train error @8000]: 0.000891
[Train error @10000]: 0.000830
[Test error]: 0.000433
```

```
>>> python solution.py --train sine_train.txt --test sine_test.txt --nn 20s
    --popsize 20 --elitism 1 --p 0.7 --K 0.1 --iter 10000

[Train error @2000]: 0.003323
[Train error @4000]: 0.002699
[Train error @6000]: 0.001903
[Train error @8000]: 0.001898
[Train error @10000]: 0.001898
[Test error]: 0.002190
```

2. Rastrigin function

```
>>> python solution.py --train rastrigin_train.txt --test rastrigin_test.txt
    --nn 5s --popsize 10 --elitism 1 --p 0.3 --K 0.5 --iter 10000

[Train error @2000]: 50.640524
[Train error @4000]: 46.177471
[Train error @6000]: 45.925162
[Train error @8000]: 45.578621
[Train error @10000]: 45.539866
[Test error]: 50.544842
```

```
>>> python solution.py --train rastrigin_train.txt --test rastrigin_test.txt
    --nn 20s --popsize 20 --elitism 1 --p 0.1 --K 0.5 --iter 10000


[Train error @2000]: 62.903413
[Train error @4000]: 15.013800
[Train error @6000]: 1.687832
[Train error @8000]: 1.230139
[Train error @10000]: 0.936814
[Test error]: 1.242374
```

3. Rosenbrock function

```
>>> python solution.py --train rosenbrock_train.txt --test rosenbrock_test.txt
    --nn 5s --popsize 10 --elitism 1 --p 0.5 --K 4. --iter 10000


[Train error @2000]: 287165.467951
[Train error @4000]: 262715.584204
[Train error @6000]: 247707.420406
[Train error @8000]: 230967.797905
[Train error @10000]: 196869.790757
[Test error]: 221803.901110
```

```
>>> python solution.py --train rosenbrock_train.txt --test rosenbrock_test.txt
    --nn 5s5s --popsize 20 --elitism 1 --p 0.5 --K 1. --iter 10000


[Train error @2000]: 158104.691349
[Train error @4000]: 79784.402381
[Train error @6000]: 63715.907020
[Train error @8000]: 63446.270092
[Train error @10000]: 62710.538447
[Test error]: 94489.025575
```

```
>>> python solution.py --train rosenbrock_train.txt --test rosenbrock_test.txt
    --nn 20s --popsize 20 --elitism 3 --p 0.5 --K 10. --iter 10000


[Train error @2000]: 149078.216254
[Train error @4000]: 124099.641035
[Train error @6000]: 107335.449190
[Train error @8000]: 98479.245133
[Train error @10000]: 96613.933070
[Test error]: 125773.981642
```