Bio-Inspired Artificial Intelligence

Prof. Giovanni Iacca
giovanni.iacca@unitn.it

Week 11 Lab Exercises

Introduction

Goal. The goal of this lab is to study the application of Genetic Programming (GP) to various kinds of problems. We will also investigate the parametrization of the algorithm and its effect on the algorithmic performance.

Getting started. Download the file `11.Exercises.zip` from Moodle and unzip it. This lab continues the use of the *deap*[1] framework for the Python programming language we have seen in the previous lab. All the exercises are based on examples taken from the *deap* tutorial on Genetic Programming[2]. If you did not participate in the previous lab, you may want to look that over first and then start this lab's exercises.

As usual, each exercise has a corresponding `.py` file. To solve the exercises, you will have to open, edit, and run these `.py` files.

Exercise 1

In this exercise we will use GP as a *supervised* Machine Learning (ML) technique for solving one of the classical problems where GP has shown the most straightforward and successful applications: symbolic regression. Symbolic regression is a form of *function approximation* that consists in fitting some data in the form of input-output samples to determine an expression (a symbolic formula) that approximates those samples. Assuming a set S of N samples ($S = \{s_1, s_2, \ldots, s_N\}$), each composed of an n-dimensional input and a scalar output y:

$$
\begin{aligned}
&s_1 : [x_1, x_2, \ldots, x_n] \rightarrow y_1 \\
&s_2 : [x_1, x_2, \ldots, x_n] \rightarrow y_2 \\
&\ldots \\
&s_N : [x_1, x_2, \ldots, x_n] \rightarrow y_N
\end{aligned}
$$

the goal is to find an expression $f(\cdot)$ such that $y = f(x_1, x_2, \ldots, x_n)\ \forall s \in S$. Usually, an error metric such as the MSE (Mean Squared Error), MSE$=\frac{1}{N}\sum_{i \in N}[y - f(x_1, x_2, \ldots, x_n)]^2$, is used to measure an individual's fitness in terms of approximation error.

[1] Distributed Evolutionary Algorithms in Python: `https://github.com/DEAP/deap`
[2] `https://deap.readthedocs.io/en/master/tutorials/advanced/gp.html`

Here, we will consider a 1-dimensional input ($n = 1$), such that the set of samples is composed of couples (x, y). By default, $N = 20$ equidistant samples are generated in the range $[-1, 1]$ by means of the polynomial function:

$$y = f^*(x) = x^4 + x^3 + x^2 + x.$$

These samples are then used to evaluate the fitness of the evolving GP trees. The goal is to find an optimal tree describing a formula that approximates $f^*(\cdot)$ such that the MSE is minimized[3]. Open the script `exercise_symbreg.py` and spend some time to understand its main steps.

One of the most important aspects is the definition of the *primitive set* (also called non-terminal or function set), which in this case contains the four basic arithmetic operations, the trigonometric functions `cos` and `sin`, and the `neg` operator (such that `neg(x)=-x`). Note that the standard division operator is replaced by a custom function named `protectedDiv`, which handles divisions by zero without returning errors (this detail is often very important in GP-based symbolic regression, where it's not possible to check a priori all possible inputs to a division node). Note also that in *deap* the configuration of the Evolutionary Algorithm (in this case, Genetic Programming) is obtained by means of a series of function hooks registered in a `creator` object (that handles the initialization of the individual genotypes with their fitness values) and a `toolbox` object (that handles all the other methods that are called during the algorithm execution: evaluation, selection, mutation, crossover, etc.). All function hooks come with a user-defined alias (the first argument of the methods `create`, `register` and `decorate`). The evaluation of the fitness function (in this case, the MSE) is implemented in the method `evalSymbReg`, that calls the method `generatorFunction` (the only part of code you should change if you want to test a different generator function).

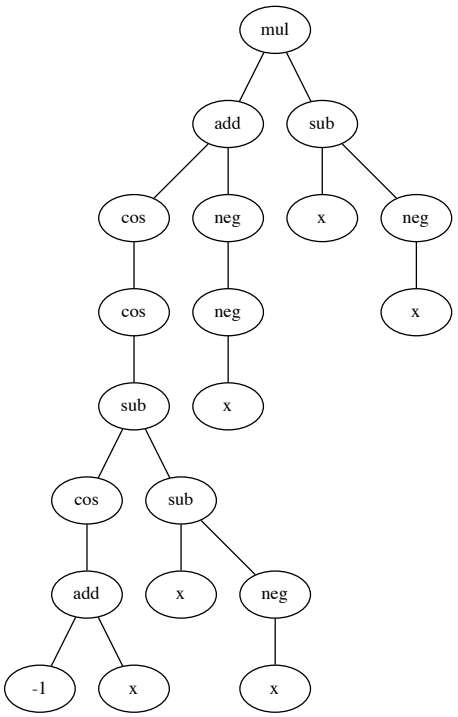

Figure 1: An example of tree evolved by GP in the symbolic regression experiments, showing the use of different operators in the primitive set.

The relevant parameters of the Genetic Programming algorithm can be found at the beginning of the script. In particular, consider the parameters `GP_POP_SIZE`, `GP_NGEN`, `GP_CXPB`, `GP_MUTPB`, `GP_TRNMT_SIZE`, and `GP_HOF_SIZE`. These parameters represent, respectively: the population size (`GP_POP_SIZE`), the number of generations (`GP_NGEN`), the crossover probability (`GP_CXPB`), the mutation probability (`GP_MUTPB`), the tournament size (`GP_TRNMT_SIZE`) and the size of the Hall-

[3]Note that while in this synthetic example we know that samples are generated by a certain *generator function* $f^*(\cdot)$, in practical applications we don't know if this function even exists -or has a meaningful physical/mathematical formulation. In other words, the goal of GP is to approximate the mapping between inputs and outputs in a purely data-driven approach.

of-Fame (`GP_HOF_SIZE`)[4]. See the documentation[5] for further explanations on the implementation details of this exercise.

To start the experiments, from a command prompt run[6]:

```
$>python exercise_symbreg.py
```

At the end of the run, the script will show you (and save on files, in the `results` folder) a graphical representation[7] of the best evolved tree (similar to that shown in Figure 1), as well as the evolutionary trends of the trees' fitness and size (i.e., number of nodes) across generations, and a comparison between the values of the real and GP-approximated data, see Figure 2.

- Is the GP algorithm able to approximate the given polynomial, with the standard configuration? What happens when you run the script multiple times? Do you always obtain the same results, or not? Why?
- Try to change the generator function (e.g. to include trigonometric functions) defined in the method `generatorFunction`. Is the GP algorithm able to approximate more complicated generator functions? Which parameters can you change to improve the results?

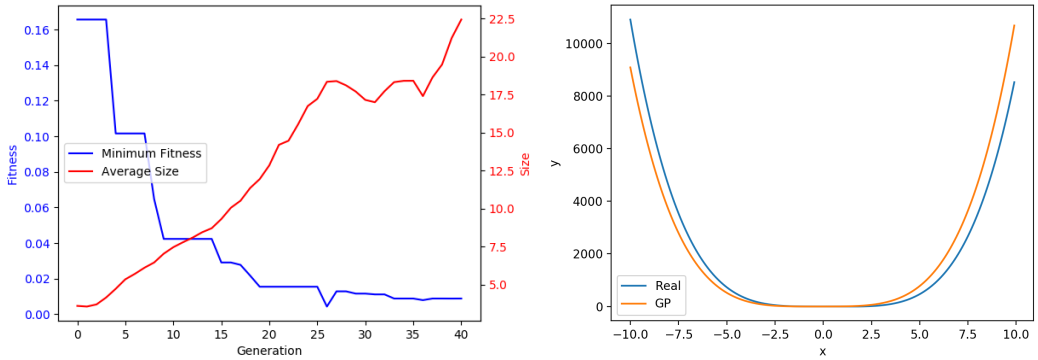

Figure 2: Output of the symbolic regression experiments: fitness/size trends (left); comparison between real and GP-approximated values (right).

Exercise 2

In this exercise we will use GP for solving different kinds of problems. In the `src` folder you will find various other examples of GP applications. Through these examples, we will see how GP can be flexibly applied to solve a broad range of problems beyond function approximation, such as classification and evolution of computer programs, with different features and levels of complexity. In particular, consider the following four scripts:

[4]Note that *deap* uses a structure called Hall-of-Fame to store the best individual(s) found during the evolutionary process. The size of this archive is determined by the parameter `GP_HOF_SIZE`. This structure is only used to report the best solutions at the end of the evolutionary process, and it is not to be confused with the Hall-of-Fame used in competitive co-evolution.

[5]`https://deap.readthedocs.io/en/master/examples/gp_symbreg.html`

[6]For all exercises in this lab you may follow the name of the `.py` file with an integer value, which will serve as the seed for the pseudo-random number generator. This will allow you to reproduce your results. Also, please note that in this document `$>` represents your command prompt, do not re-type these symbols.

[7]The Python packages `pygraphviz` and `networkx` are required for this.

1. `exercise_parity.py`: the even parity problem[8]. This is a classical benchmark problem used in the GP literature: the goal is to *evolve a Boolean expression* that produces the correct value of the even parity bit, given as input an array of n bits. For instance, for an input size $n = 6$ (Parity-6 problem), the goal is to match the parity bit value for each of the $2^6 = 64$ possible combinations of 6 bits[9]. Thus the fitness of an individual (i.e., a candidate Boolean expression) can be calculated simply as the number of successful cases (to be maximized), such that the maximum attainable fitness is 64 in the case of 6 bits inputs. Note that since in this case all inputs/outputs are bits (Boolean values), the primitive set contains only Boolean operators (`and`, `or`, `xor`, `not`).

2. `exercise_multiplexer.py`: the multiplexer problem[10]. This is another classical benchmark problem used in the GP literature. Similarly to the even parity problem, this problem deals with Boolean values: the goal is *evolve a Boolean expression* that is capable to reproduce the behavior of an electronic multiplexer (mux)[11]. In the default configuration, a 3-8 multiplexer is used (3 select entries, labeled from A0 to A2, and 8 data entries, from D0 to D7), for a total number of $2^3 \times 2^8 = 2048$ possible combinations. Thu the fitness is the number of correct outputs over all 2048 cases, and its maximum value is 2048.

3. `exercise_spambase.py`: the "spam base" problem[12]. This problem consists in *evolving a classifier* that is able to distinguish as spam/non-spam a given database of emails (saved in `data/spambase.csv`). The database consists of 4601 *labeled* samples (as such, GP is used here as a supervised ML algorithm), each corresponding to a different email and comprising 57 mixed integer/real-valued features describing various lexicographic/syntactic characteristics of the email[13], in addition to a binary label indicating if the email is spam (1) or not (0). The evolved classifier must then return a Boolean value which must be `True` if the email is spam, `False` otherwise. To avoid overfitting, each tree generated during the GP run is evaluated on a set of 400 samples randomly selected from the database (so that the maximum achievable fitness, i.e. number of correct classifications, is 400). Note that since data of different kinds must be manipulated by the classifier (float and Boolean values), in this case the GP algorithm is configured to handle *strongly typed* operations, i.e. for each function in the primitive set the type of inputs and outputs is specified (e.g. to avoid calling a Boolean function with float values as inputs).

4. `exercise_ant.py`: the "artificial ant" problem[14]. This problem consists in *evolving a program* that can successfully control an artificial ant so that it can find and "eat" all the food located in a certain environment (i.e., placed along a twisting trail on a square toroidal grid). The program can use three operations, `move_forward`, `turn_right` and `turn_left`, to move the ant forward one cell, turn right or turn left. Each of these operations takes one timestep. The sensing function `if_food_ahead` looks into the cell the ant is currently facing and then executes one of its two arguments, depending upon whether that cell contains food or is empty. Two additional custom operations, `prog2` and `prog3` are provided to facilitate the evolution of more complex behaviors. These custom operations take two and three other operations as arguments, respectively, and execute them in sequence. The evaluation function uses an instance of a simple simulator to evaluate a given individual

[8] `https://deap.readthedocs.io/en/master/examples/gp_parity.html`
[9] `https://en.wikipedia.org/wiki/Parity_bit` for more details.
[10] `https://deap.readthedocs.io/en/master/examples/gp_multiplexer.html`
[11] See `https://en.wikipedia.org/wiki/Multiplexer` for more details.
[12] `https://deap.readthedocs.io/en/master/examples/gp_spambase.html`
[13] See `http://archive.ics.uci.edu/ml/datasets/Spambase` for a complete description of the features.
[14] `https://deap.readthedocs.io/en/master/examples/gp_ant.html`

(i.e., a candidate program). Each individual is given 600 timesteps to navigate a virtual map obtained from an external file (see the file `data/santafe_trail.txt`, where "`#`", "`.`" and "`S`" indicate, respectively, a cell with food, an empty cell, and the starting cell). The fitness of each individual corresponds to the number of pieces of food picked up. In this example, the trail contains 89 pieces of food in total. Therefore, an optimal individual would achieve a fitness of 89.

Since these scripts have all the same structure as the one used in the first exercise (besides obviously the details of the specific problem, especially the evaluator function, and the definition of the primitive set), we won't go into the details of each of them. You can spend some time having a look at the code and trying to get the main steps of each of the four examples.

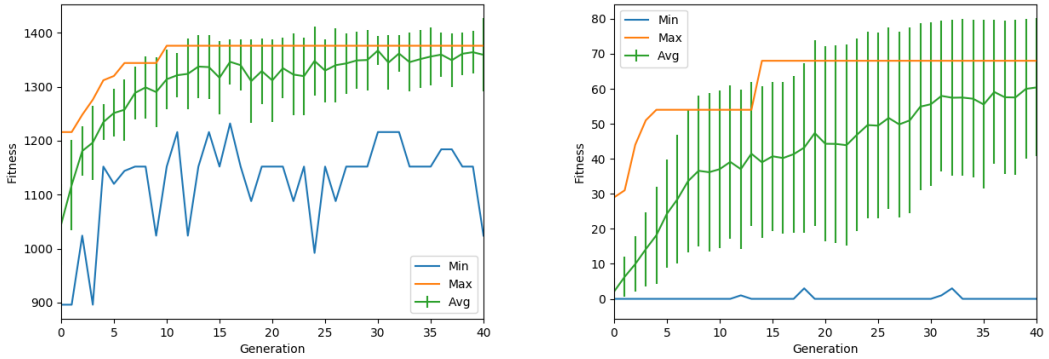

Figure 3: Example fitness trends for the 3-8 multiplexer problem (left) and the "artificial ant" problem (right).

Try to run some experiments with one or more of these four scripts. As in the previous exercise, at the end of the run each script will show you (and save on files, in the `results` folder) a graphical representation of the best evolved tree, as well as the evolutionary trends of the trees' fitness across generations, see Figure 3.

- In the case of the multiplexer problem there is an additional parameter (`MUX_SELECT_LINES`) that allows you to scale the problem, increasing or decreasing its dimensionality. Similarly, for the parity problem there is a parameter (`PARITY_FANIN_M`) that allows you to change the problem dimensionality. This in turn will make the problem easier, or harder. If you tested one of these two problems, consider changing these parameters and observe the GP's behavior. Note, however, that these problems (especially the multiplexer) become computationally very complex when the dimensionality increases, so that experiments can be quite time-consuming!
- What kind of performance do you get, in general, on the tested problems? What happens when you change the parametrization of GP?

Instructions and questions

Concisely note down your observations from the previous exercises (follow the bullet points) and think about the following questions.

- What are the main strengths and limitations of GP, in your opinion?

- In which kind of applications do you think that GP could be more useful than other kinds of black-box Machine Learning techniques, such as Neural Network? Why?

BONUS: In the `src` folder there is a variant of the first exercise, `exercise_symbreg_coev.py`. The main difference between this and the original exercise is that in this variant a *competitive co-evolutionary approach* is used for solving the symbolic regression problem. Rather than determining *a priori* the samples that are used for evolving the GP trees, this approach uses a Genetic Algorithm (GA) to *evolve* a fix-sized set of samples that is fed to a tree evolved by GP. The goal of GP is to *minimize* the MSE, while the goal of GA is to *maximize* it, i.e. to generate a set of samples that produces the largest fitting error in the evolving GP trees.

To do so, at the initial generation a random GA (GP) individual is sampled from the corresponding initial population and tested against all individuals in the initial population of GP (GA). Then at each subsequent generation the best GA individual (i.e., the set of samples that produced the largest MSE so far) is fed to all the trees in the current GP population. Vice versa, the best GP tree (i.e., the one that showed the lowest MSE so far) is fed with all the individuals (i.e., sets of samples) in the current GA population. This form of *arms race* forces the GP trees to become more adaptive, avoiding overfitting to a particular set of samples, thus also becoming better at generalizing and extrapolating data outside the specific set of samples that was used for fitting. Note, however, that in some cases this approach may lead to excessively large trees that are even impossible to evaluate[15].

Run some tests with the co-evolutionary variant of the symbolic regression experiment. At the end of the evolutionary process, you should see a separate fitness trend for the GA and the GP evolution.

- Is this approach better (in terms of time to converge, approximation results, and robustness across multiple runs) than the one based on GP alone? Why?
- Try to change the parameters of the co-evolutionary scheme to test how the system behaves under different configurations. Focus in particular on the population size used in the two algorithms (`GA_POP_SIZE` and `GP_POP_SIZE`), and the number of samples contained in each GA individual (`GA_REP_IND`).

[15] This is the typical effect of *bloat*. In the most extreme cases, this could even cause an internal Python `MemoryError`. Indeed, due to an implicit limitation of Python, trees deeper than 90 layers cannot be evaluated. In these cases the script gives the following error:
`MemoryError: DEAP : Error in tree evaluation : Python cannot evaluate a tree higher than 90. To avoid this problem, you should use bloat control on your operators. See the DEAP documentation for more information. DEAP will now abort.`