Master in Artificial Intelligence (CI-MAI) Evolutionary Computation practical work

Lluís Belanche

10 oct. 2024

Training Neural Networks with Evolutionary Algorithms

The application of evolutionary search procedures to artificial neural networks is a long-standing research topic. Literally hundreds of works have been published in the matter, with different approaches and levels of sophistication. The topic is still active, so I give you (as a zipped folder) several classical and recent references if you want to get more inspiration or delve deeper in the subject.

We can distinguish five main kinds of evolution in artificial neural networks:

- evolution of connection weights (their values);
- evolution of architectures (number of layers, neurons per layer and connectivity pattern);
- evolution of hyper-parameters;
- evolution of activation functions;
- evolution of learning rules.

These topics can be carried out simultaneously, although only the first two or three are combined. Finding a suitable way of using evolutionary algorithms for optimizing artificial neural networks is by no means a trivial matter. However, they bring several clear advantages over traditional, derivative-based methods:

- the objective (error) function need not be continuous nor differentiable
- the objective (error) function can incorporate discrete information (like the number of neurons)
- the objective (error) function can be noisy (as is typically the case in machine learning)

On the other hand, the error landscape is complex and noisy since the mapping from an architecture to its performance is indirect and dependent on the evaluation method used. Moreover,

1. The error landscape is deceptive since similar architectures may have quite different performance.[1]
2. The error landscape is multimodal since different architectures may have similar performance.[2]

All these features make the matter very appealing. Arguably the worse drawback is the computational cost.

The exercise consists in creating a full method for training (i.e., *learning*) an artificial neural network (ANN) as an alternate method to backprop-based techniques. To make the exercise feasible for the amount of time given, the following decisions must be adopted:

1. Use a MLP or a RBF network, with a single layer of hidden neurons
2. Use synthetic data (the precise problem is left to you); this allows to be in control of:

- the sample size of the datasets used for training, validation and testing (I suggest varying only the first of these and keep the others constant, to a large number)
- the true generalization error of a model (approximated by its error on a large test set)
- the amount of noise (controlled by a single hyperparameter)

[1]This is caused by the fact that a single neuron can have an important impact on *predictive* performance.

[2]This is caused by the presence of symmetries: exchanging hidden neurons, input weights, etc, which will not affect performance.

- the problem hardness (controlled by a single hyperparameter) [3]

3. Use only one problem and one kind of ANN
4. You can set a maximum number of neurons and weight values
5. You can create either a classification or a regression problem, and then choose a suitable error function
6. Use a single validation set (no need for cross-validation)

Regarding regularization, you can choose one or both commonly used strategies:

- Avoid the use of a regularization parameter (also called weight decay, in the neural network context) and search among a predefined set of numbers of hidden neurons
- Fix a maximum number of hidden neurons (such that the network overfits[4]), and then search search among a predefined set of values for the regularization parameter

In either case, notice that the validation error of a neural network depends on the initial values for the weights, which are typically initialized randomly. In the evolutionary setting, this is not a problem, since the network comes from a population individual. A crucial issue is how to communicate the EA and the ANN: it must be flexible and efficient; make good use of the given packages.

The (strongly) suggested R packages to be used are: **GA, cmaesr, nnet**. Other packages are up to you. The goal of the work is to compare:

1. Genetic algorithms against derivative methods
2. Evolution Strategies against derivative methods
3. All three

What to report in the comparison?

- execution time (in R, see the *tic* and *toc* functions)
- estimate of true error of the chosen model
- size (number of neurons or decay parameter) of the chosen model

Optimizing a difficult function with Evolution Strategies

The application of formal tools to the minimization/maximization of multivariate functions is a long-standing problem in mathematics. It is not possible to compile the numerous approaches existing in the literature, mostly based on computing first order (the gradient) and/or second order (the Hessian) derivatives to get approximate information of the local behaviour of the function to be optimized.

Literally hundreds of works have been published in the matter, with different approaches and levels of sophistication. The best approach is to consult good specialized books on the matter, like those written by Dimitri P. Bertsekas (*Convex Optimization Theory*, *Convex Analysis and Optimization*)[5] or by Boyd and Vandenberghe (*Convex Optimization*)[6].

Moreover, constraints on the function can also be added to the fitness function. In mathematical optimization, this leads to Linear Programming, Quadratic Programming, and the like.

You are required to select a problem (function to be minimized/maximized, root finding, etc).

Ideally, you may bring your own problem, but there are many (mostly artificial) problems designed as a benchmark suite; a couple of starting webs are:

- https://www.sfu.ca/~ssurjano/optimization.html
- https://en.wikipedia.org/wiki/Test_functions_for_optimization

[3] Again, the theoretically optimal generalization error can be estimated as the error on a large test set of the non-noisy data generator.

[4] How do we do this? One simple way is to find the simplest network that is clearly overfitting the data.

[5] Course notes (that cover the same material as the textbook) freely available on his website: http://web.mit.edu/dimitrib/www/home.html

[6] Freely available at https://web.stanford.edu/ boyd/cvxbook/bv_cvxbook.pdf

The exercise consists in creating or choosing a problem like the ones described above and attack it using an Evolution Strategies (ESs). The requirement for this practical work is that the search space must be a (preferably compact) subset of R^n.

The use of synthetic data allows to be in control of:

- the dimension of the problem n
- the way you represent mutation in the ES (as seen in class)
- the possible present of noise (controlled by a single hyperparameter)
- the problem hardness (controlled by a single hyperparameter) [7]

All other conditions, settings, hyperparameters, etc, are left to your decision. You can also choose any other (evolutionary or non evolutionary) method that you want to test in comparison to the ES. The problem hardness should not be extremely difficult or trivial, otherwise the study is pointless.

What to report in the comparison? An obvious choice is to set a predefined performance and then report the (average):

- execution time needed to achieve a given performance
- number of generations needed to achieve a given performance
- number of fitness function calls needed to achieve a given performance
- Fraction of times the algorithm reaches a given performance

  ... all as a function of population size, n or problem hardness. Alternatively, one can let the algorithm run for a predefined number of generations and study all the other quantities. The study is quite flexible and left to your criterion.

Optimizing a difficult function with Genetic Algorithms

The application of formal tools to the minimization/maximization of multivariate binary functions is a long-standing problem in mathematics. Combinatorial optimization problems can be viewed as searching for the best element of some set of discrete items. Perhaps the most universally applicable approaches are branch-and-bound (an exact algorithm which can be stopped at any point in time to serve as heuristic), branch-and-cut (uses linear optimisation to generate bounds), dynamic programming (a recursive solution construction with limited search window) and tabu search (a greedy-type swapping algorithm)[8].

As in the continuous case, it is not possible to compile the numerous approaches existing in the literature. Again, arguably the best approach is to consult good specialized books on the matter, like those written by Schrijver[9] or the now classic by Papadimitriou and Steiglitz *Combinatorial Optimization: Algorithms and Complexity.*

Some examples of combinatorial optimization problems that are covered by this framework are shortest paths and shortest-path trees, flows and circulations, spanning trees, matching, and matroid problems. Famous examples include the travelling salesman problem (TSP), the minimum spanning tree problem (MST), and the knapsack problem.

These are not the only problems that may have a natural binary representation. Many other problems in search, scheduling, planning, etc, have been tackled by Genetic Algorithms, to an astonishing number of applications (in the thousands!). Even discrete or integer optimization is possible (by using Gray codes or by using alphabets of cardinality larger than two).

You are required to select a problem. Ideally, you may bring your own problem, but there are many (mostly artificial) problems designed as a benchmark suite; a couple of starting papers are:

- https://www.sciencedirect.com/science/article/abs/pii/0303264796016218

[7] It is desirable that you know the theoretically optimal solution.
[8] This last method could be considered an EA, under a wide sense of the definition.
[9] Freely available at http://homepages.cwi.nl/~lex/files/dict.pdf

- https://dl.acm.org/doi/abs/10.1145/3205651.3208251

The exercise consists in creating or choosing a problem like the ones described above and attack it using GAs. The requirement for this practical work is that the search space must be a hypercube of dimension n (the number of bits and also the chromosome length, of your choice).

You are then required to choose appropriate:

- representation for the candidate solutions
- selection mechanism
- crossover operator(s)
- mutation operator(s)
- stopping criteria

The use of synthetic data allows to be in control of:

- the dimension of the problem n
- the way you represent the individuals
- the amount of available data for the fitness function
- the problem hardness (controlled by a single hyperparameter) [10]

All other conditions, settings, hyperparameters, etc, are left to your decision. You can also choose any other (evolutionary or non evolutionary) method that you want to test in comparison to the GAs. The problem hardness should not be extremely difficult or trivial, otherwise the study is pointless.

What to report in the comparison? An obvious choice is to set a predefined performance and then report the (average):

- execution time needed to achieve a given performance
- number of generations needed to achieve a given performance
- number of fitness function calls needed to achieve a given performance
- Fraction of times the algorithm reaches a given performance

  … as a function of population size, n or problem hardness. Alternatively, one can let the algorithm run for a predefined number of generations and study all the other quantities. The study is quite flexible and left to your criterion.

Tackling the Traveling Salesman Problem with Genetic Algorithms

The Traveling Salesman Problem or TSP is probably onoe of the most studied problem in combinatorial optimization. Specifically for the TSP, there are good web resources like this[11], and many others.

Still the problem is very interesting by itself, given that it is an NP-hard problem in combinatorial optimization and has inspired dozens of new algorithms and applications.

In the context of this practical work, the aim is to devise "suboptimal" or heuristic algorithms, i.e., algorithms that deliver approximated solutions in a reasonable time, a perfect setting for a genetic algorithm. More specifically, the idea is to research the literature to find workable[12] proposals for specialized chromosomal representations, crossover operators and mutation operators.

TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types. As an example, Homer's Odyssey of Ulysses[13] can be used for this task, make sure to download the 22-city version (the 16-city one is probably too easy). A nice read is also The Optimized Odyssey.

What to study and report is left to you.

[10]It is desirable that you know the theoretically optimal solution.

[11]https://www.math.uwaterloo.ca/tsp/optimal/index.html

[12]Meaning "amenable to implementation and experimentation".

[13]It follows the Greek hero Odysseus, king of Ithaca, and his journey home after the Trojan War.

Important information (mandatory)

- Deliver a brief pdf document (10 sheets maximum, including everything) that describes only the relevant information (problem setup, previous work, work done by you, discussion and conclusions)
- Delivery date: no later than **November 10, 2024**, via the Racó
- To be done in groups of 2 students
- Please include all names in the final document and upload only one document per group
- Add a plain text file named "README.txt" with complete instructions about how to obtain your results
- If you use ChatGPT (or another similar tool) in the document, indicate it every time it is used. We want to evaluate your work, not someone else's! You can find information on how to cite it in Citing Generative AI.

Suggestions (your choice)

- There is *no obligation* to use any particular programming language; but I ask you to limit the choice to R, Julia, MATLAB and python.
- If you use R, a good tool is to use *Rmarkdown* to produce the document, integrating LaTeX code and R code. If you do, have a look at https://bookdown.org/yihui/rmarkdown/. There are other similar tools for other languages: for example, R Markdown and knitr do support Python. You can add a Python code chunk to an R Markdown document seamlessly. Moreover, the **reticulate** package includes a Python engine for R Markdown that enables easy interoperability between Python and R chunks.
- Similar IDE tools are available for python: Visual Studio Code, PyCharm, Sublime Text 3, Jupyter, many usable also for R and Julia. For the latter, have a look at Pluto.
- Keep the work problem simple; once the whole thing works, you can complicate it.