Natural Computing 2019/2020: Assignment Genetic programming

School of Informatics, University of Edinburgh
s1653529

November 10, 2019

Abstract

With Charles Darwin's publication of The Origin of Species he introduced the basis of a theory that would only gain in influence and importance up until the present time. With the emerging concepts of various algorithms, it also found its place in the field of Computer Science, more specifically, the field of Genetic Programming. This paper puts forward multiple approaches on how Genetic Programming can be used to generate polynomials that produce prime numbers from a given input. It continues with the presentation of different integer functions for said purpose and finishes with an outlook on the use of a GP for time-series prediction.

Introduction

A number p is a Lucky number of Euler if and only if the prime-generating polynomial n^2+n+p is prime for $n = 1, 2, ..., p-1$. It has already been proven that the only lucky numbers are 2, 3, 5, 11, 17, and 41 (WolframMathWorld, 04.11.19). This paper does not attempt to question Heegner and Stark (WolframMathWorld, 04.11.19) through extensive exploration. This paper attempts to explore various polynomials, later more general function-combinations and their ability to produce primes.

A Genetic Program starts with a population of computer programs. These programs can be chosen at random, initialised by another algorithm or initialised according to a specific distribution. It then proceeds to evaluate all programs in the population. The program evaluations are done through a so-called fitness function. Subsequently, the best scoring programs are kept from the population while the other programs are discarded. This step will be referred to as the selection process. From this reduced population, a subset is selected to recombine the programs within and replenish the reduced population by a certain amount. This process is denoted as the replenishing phase. It then continues to iteratively apply these last two steps to the population until the predefined termination condition is met.

This paper starts with an attempt to reproduce the polynomial $x^2 + x + 41$ with a Genetic Algorithm. The results will serve as a baseline model. It will help to establish reasonable bounds for the parameters. The parameters are chosen is such a way that they are comparable. The GA offers a sensible starting point. Followed by that, it will present an implementation of a Genetic Program and the experiments that were carried out with it. Lastly, it will combine the GP with Simulated Annealing to reduce the number of initially useless programs and offer an outlook on time-series prediction with GPs.

Basic Concept

Fitness function

For the evaluation of the GA and the GP a function f is utilized that combines three fitness functions by adding their scores. The scores are not weighted. The first fitness function returns the number of primes that were produced by the polynomial on a given input interval. The second fitness function returns the maximal number of consecutively produced primes on the given input interval $X = [0, 200)$. The last fitness function only returns the number of primes that are consecutively produced when starting at the lower bound of the input interval. A polynomial p is

evaluated via

$$\begin{aligned} f(p) =& f_1(p) + f_2(p) + f_3(p) \\ f_1(p) =& |\{p(x_i) \in \mathbb{P} \mid x_i \in X\}| \\ f_2(p) =& \max_{x_k \in X} |\{x_i \mid x_i, x_{i-1}, ..., x_k \in X, \\ & p(x_i), p(x_{i-1}), ..., p(x_k) \in \mathbb{P}\}| \\ f_3(p) =& |\{x_i \mid x_i, x_{i-1}, ..., x_0 \in X, \\ & p(x_i), p(x_{i-1}), ..., p(x_0) \in \mathbb{P}\}|, \end{aligned}$$

where $\mathbb{P}$ denotes the set of prime numbers.

Populations

The GA/GP starts with a population of polynomials of a given size. Multiple populations can be chosen to evolve in parallel. The populations merge whenever a critical, predefined population size is reached. The GA/GP iteratively goes through the selection process followed by the replenishing of the population. The GA/GP stops and returns its result as soon as the population size stagnates, meaning that it only contains one kind of polynomial.

Selection

In the selection process the polynomials with a fitness score above the population's fitness mean f_μ remain in the population. The rest of the population is discarded. The selection process redefines a population P_r via

$$P_r \leftarrow \{p \in P_r \mid f(p) \geq f_\mu\}.$$

The algorithm handles every polynomial as a unique object even if its composition is not.

Replenishment

Subsequently, the number of discarded polynomials $|P_d|$ is determined. The remaining polynomials P_r are then taken and recombined in the crossover phase. The population is replenished by an amount $\eta \cdot |P_d|$ where η is the replenishing-rate also denoted as the birth-rate here. The replenishing phase is summarized via

$$P_r = P_r \cup \ Crossover\left(\bigcup_{i=1}^{\eta\cdot|P_d|} p_i, \bigcup_{j=1}^{\eta\cdot|P_d|} p_j\right), \quad p_i, p_j \sim D(\epsilon),$$

where $\epsilon \propto 0.1 \cdot |P_r|$ and prevents zero-divisions. $D(\epsilon)$ is a probability distribution over P_r following

$$d(p|\epsilon) = \frac{f(p) + \epsilon}{\sum_{p_k \in P_r}(f(p_k) + \epsilon}.$$

The crossover-function recombines features of the polynomials from the two given input sets. This will be explained in later sections.

Baseline

Concept

As mentioned before, a GA was chosen to represent the baseline for comparisons.
The chromosome of a polynomial in the implemented GA consists of its coefficients incorporated into an array of the size of its order. The bound for the coefficients is predefined. The bound of the order of the polynomial is also set beforehand and can be regarded as an additional feature in the polynomial's chromosome. The coefficients and the order are initialised via a uniform distribution over the given bounds. Firstly, the order is determined. The coefficients follow.
The **crossover-function** takes one polynomial from the first input set and combines it with another polynomial of the second input set with a coinciding index. Hence, each pair produces one offspring polynomial. It is very much possible that a polynomial is recombined with a copy of itself. The recombination works through extending both coefficient arrays with zeros to a length that goes up to the order constraint. The new polynomial is then formed by selecting either of the respective coefficients from the two arrays factor-wise at equal chance.
The resulting array is then passed through mutation function with a predefined mutation probability. The mutation function then selects a coefficient from the zero-extended, resulting array and substitutes it by a new coefficient value selected from the uniform distribution over the coefficient bounds. The leading zeros are then discarded.

Setting the Baseline

For $x^2 + x + 41$ the recorded fitness score is 236, producing 156 primes on the input interval. The starting streak is the longest streak with a length of 40 consecutively produced primes. After several trials the GA that reliably reproduced the

polynomial had one population of 5000 polynomials, a birth-rate of 0.7, a mutation-rate of 0.1, a coefficient bounding of $[1, 41]$, and an order restriction of at most degree 2 (Figure 1). It converged after just under 35 generations (other trials managed to converge close to 25). One can observe that there was a rather big oscillation in fitness at generation 17 and 29. Before generation 17, one can assume that it was stuck in a local maximum as it also seemed to stagnate. The rather high mutation-rate could have contributed to its further improvement.

Genetic Program Implementation

Concept

The chromosome of the function to be produced by the GP is represented by a syntax tree (see Figure 2). The leaf-nodes are either terminal or non-terminal. Terminals are simply integers while non-terminals form variables. The internal nodes make up sets of nested functions. The set of functions to choose from can be passed in as an argument of the population. Previously established coefficient bounds are also applied on the terminals here. The order bound is now implicitly given by limiting the depth of the tree. A tree is initialised by creating internal nodes down to the depth-limit. The function is chosen randomly from the set of passed in functions. The leaves are then randomly filled by either terminal or non-terminal nodes.

The **crossover-function** takes one tree from the first input set and combines it with another tree from the second input set with coinciding index. Each tree pair produces one offspring tree. Both input trees are then dissected. In the first input set, the nodes of all trees that are above half the depth limit are selected and added to a breeder set. In the second input set, the nodes of all trees that are below half the depth limit are selected and added to another breeder set. Any of the child nodes of the nodes in the first breeder set can then be substituted by any node of the second breeder set. This is done in a purely random manner.

A node of the second breeder set has a chance to be substituted by a randomly generated tree of half the depth of the depth limit. The probability of the substitution is determined by the mutation-rate parameter.

Experiments

Using the same parameters as with the GA, i.e. a depth limit of 3, the GP also manages to reproduce the polynomial x^2+x+41. It took around 30 generations longer to converge. Though, the GP produced the polynomial more reliably than the GA. The average fitness oscillated more heavily than with the baseline. Surprisingly, it did not take longer because $x^2 + x + 41$ did not appear in the population until the very end. x^2+x+41 actually, most likely already appeared in the 2nd generation when using the GP. Its converging time depends more on the chosen termination condition. The population utilizing the GA needed 17 generations for that. The GP also seems to provide a greater diversity in the top-performing polynomials (see Figure 4 and Figure 3). Figure 5 shows the effects of the two different phases on the average fitness throughout the generations. The differences become more severe in the final generations, suggesting that the fitness deviation from the mean increases.

Expanding the coefficient bounds from $[1, 41]$ to $[-61, 61]$ and increasing the depth limit from 3 to 5 effectively increases the search space. The consequences are quite evident to observe. While the converging time did not increase, it did not perform well. With a final fitness score of 145, it settled on a significantly worse polynomial than $x^2 + x + 41$ (see Figure 6).

One could counter this by simply increasing the population size. Instead of following that approach, one can introduce multiple populations and evolve them in parallel. In the next experiment three populations developed in parallel until the number of polynomials in the first population dropped to under 40% of its original size. This slightly increased the final fitness score to 154 (see Figure 7).

The next experiment attempts to offer a glance on the direct influence of the mutation-rate on the performance. The low mutation-rate seems to do well and questions the previous assumption that the mutation rate was responsible for the oscillations. Additionally, the low mutation-rate seems to have affected the f_2- and f_3-scores. The population with the low mutation-rate had a prime streak that was longer by 24 consecutive inputs. The fitness comparison graph (Figure 9) suggests that the crossover could not lead to major improvements due to very likely, subsequent mutations. For a comparison of the average fit-

ness, see Figure 8.
Up to this point the GP only made use of two functions, namely addition and multiplication. Three functions that are excellent candidates for expanding the set of possible functions are $\max(a, b)$, $\min(a, b)$ and $a \mod b$. The mod-function defines taking something mod 0 as 0 here. Only using these three functions with previous parameters decreased the final score to 49. The average fitness oscillated substantially more.
Using the function set [addition, multiplication, maximum, mod] with a 40% chance for the maximum function (20% for the other functions respectively) delivered unexpected results. Not only did the final function reach the previous maximum score of 236, with 40 consecutive primes starting at 0, with a total of 156 primes. It managed to do this in just 25 generations, therefore, being even faster than the GA. It should also be noted that the search space was substantially larger than when the GA ran. The final tree most likely appeared first in generation 11. The average fitness oscillated less heavily (see Figure 10). When considering numpy's mod function the tree actually collapses to $x^2 + x + 41$ (see Figure 11). Using the function set [addition, multiplication, maximum, minimum, mod] delivered similar results in 40 generations.

Experiments with Simulated Annealing

Local search methods can be incorporated into the GP to quickly determine the terminal values of the tree. Using Simulated Annealing (Algorithm 1) on the terminal values of a random tree with a fitness score of 0 improved its score to 8(1000 iterations, a temperature of 100000, and a temperature change of 0.98). When combined with the actual GP, using 700 iterations, a temperature of 70000, and a temperature change of 0.98, the GP converged within 26 generations. That's a reduction by more than half compared to the GP presented in Figure 3. The final polynomial most likely already appeared in generation 3 (see Figure 12). The algorithm applied Simulated Annealing to 10% of the population after each replenishing phase. The effect of the local search can be observed in Figure 13 and is essentially depicted by the height difference between a green bar and the respective next blue bar.

Outlook

Genetic Programming concepts have various, possible applications, one of those being time series prediction. This can be to predict stock prices or to give sensible weather forecasts for example. Here, the GP attempts to find a tree that can model the series of input values and true labels.
To enable the GP implementation to give reasonable time series predictions the fitness function needs to be switched out. It must be redefined through a simple mean squared error between prediction and label, i.e.

$$f(p) = \frac{(p(X) - Y)^2}{|Y|},$$

where X is, as usual, the input interval and Y are the true labels for said input interval. The result of an example approximation is displayed in Figure 14.
GP-trees are essentially weighted, directed and acyclic graphs. According to Wiener's Tauberian Theorem, one can approximate any L^1-function, i.e. including all functions $g : \mathbb{R} \to \mathbb{R}$ such that

$$\int_{-\infty}^{\infty} |g(x)|dx < \infty,$$

with linear combinations of translations of another function (e.g. sine, cosine) (WolframMathWorld, 09.11.19). This essentially proves that a GP can produce an approximation of a time-series, given it's absolute integral does not diverge to infinity. Mind the fact that it does not define the means of how to get to said approximation.

Conclusions

A GA can converge substantially faster than a comparable GP if it is well formulated but a GP is likely to produce viable solutions quicker in its population. A GPs performance heavily depends on the parameter choices one makes. A local search method such as Simulated Annealing can reduce the population size and the number of generations necessary to produce good results. Given an appropriate fitness function and operator set a GP can produce a function that can approximate any time-series falling into the category of L^1-functions.

References

WolframMathWorld. Lucky number of euler. *WolframMathWorld*, 04.11.19. URL http://mathworld.wolfram.com/LuckyNumberofEuler.html.

WolframMathWorld. Tauberian theorem. *WolframMathWorld*, 09.11.19. URL http://mathworld.wolfram.com/TauberianTheorem.html.

Appendices

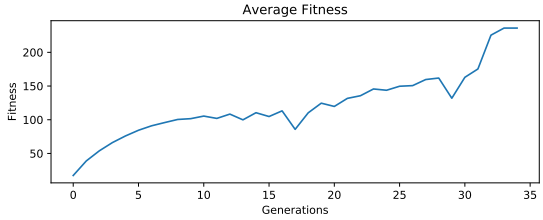

Figure 1: Average fitness of the GA Baseline over its generations

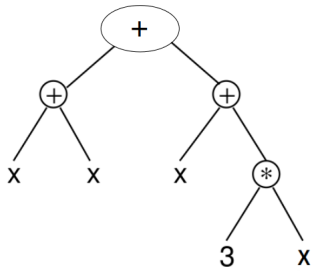

Figure 2: Tree-structure of a polynomial produced by a GP, i.e. $6x$

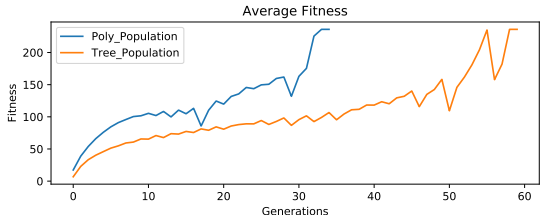

Figure 3: Comparison of the average fitness of the baseline/GA (Poly Population) and the GP (Tree Population) throughout the generations

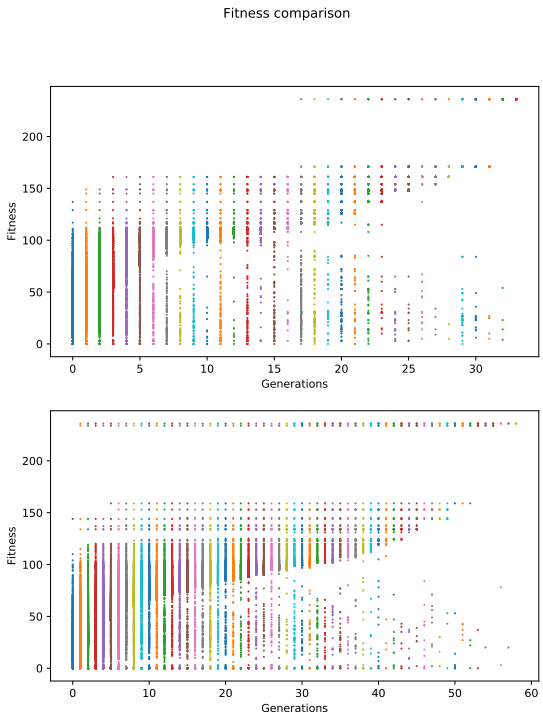

Figure 4: Comparison of the fitness of the baseline/GA (top) and the GP (bottom) polynomials throughout the generations, each polynomial fitness score recorded

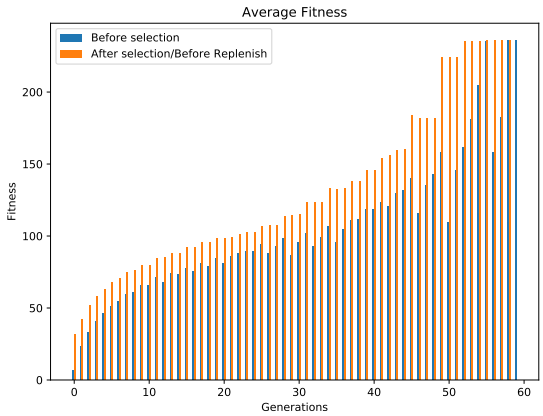

Figure 5: Comparison of the average fitness in selection phase and replenishing phase throughout the generations

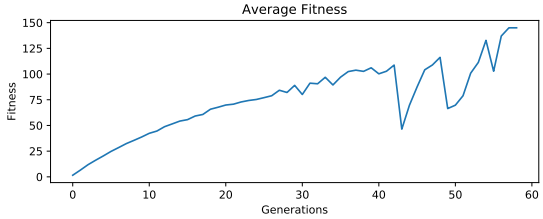

Figure 6: Average fitness throughout the generations with increased search space, i.e. coefficient bounds $[-61, 61]$, depth limit 5 (number of populations 1, population size 5000, birth-rate 0.7, mutation-rate 0.1)

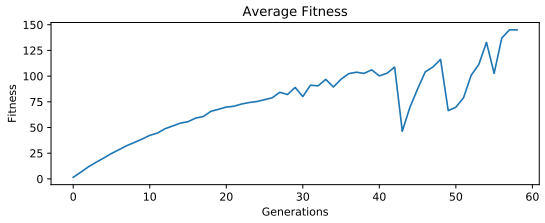

Figure 7: Average fitness throughout the generations with multiple populations, i.e. 3 (coefficient bounds $[-61, 61]$, depth limit 5, number of populations 1, population size 5000, birth-rate 0.7, mutation-rate 0.1)

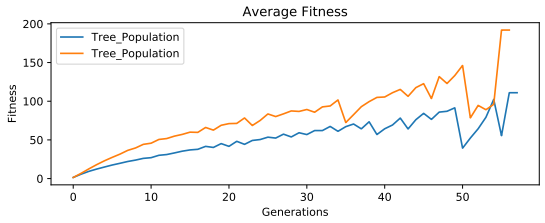

Figure 8: Average fitness of the GP throughout the generations with decreased mutation-rate (0.01, orange) and increased mutation-rate (0.5, blue) (number of populations 1, coefficient bounds $[-61, 61]$, depth limit 5, population size 5000, birth-rate 0.7)

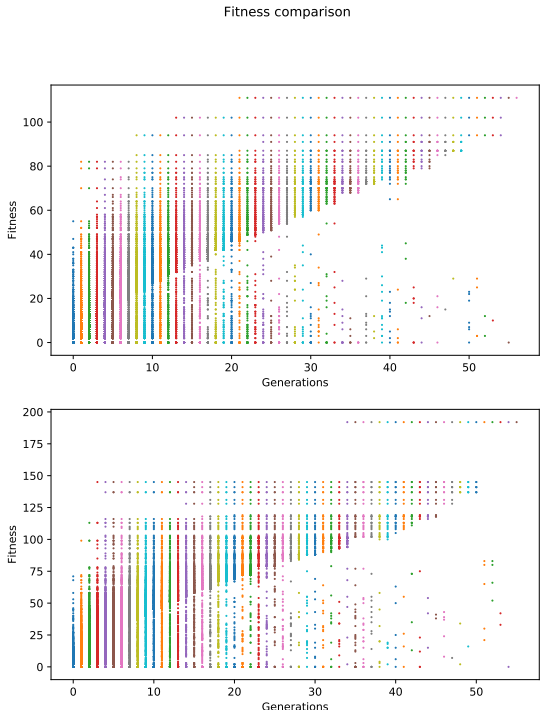

Figure 9: Fitness of the GP throughout the generations with decreased mutation-rate (0.01, top) and increased mutation-rate (0.5, bottom) (number of populations 1, coefficient bounds $[-61, 61]$, depth limit 5, population size 5000, birth-rate 0.7)

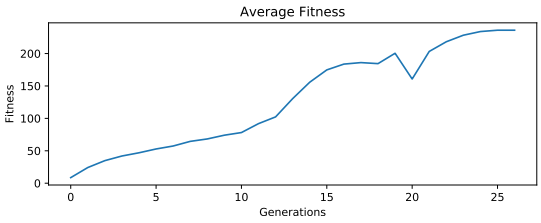

Figure 10: Average fitness of the GP throughout the generations with extended operator function set $max(max(28 + x, max(41 + x, (max(x, 16)(-112) + 56\ mod\ x^2) + (-520 + max(x, 29))mod\ x\ mod\ 2\ mod\ (51x))) + x^2, max(15x, x\ mod\ 53\ mod\ max(11, x)) + (x - 46))$, depth limit 5, population size 5000, birth-rate 0.7)

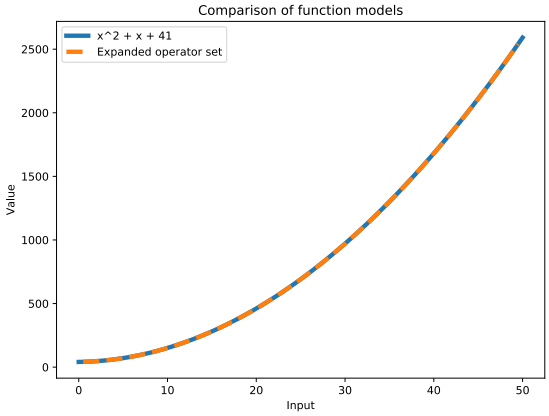

Figure 11: $x^2 + x + 41$ and the GP-tree with extended operator function set $max(max(28 + x, max(41 + x, (max(x, 16)(-112) + 56\ mod\ x^2) + (-520 + max(x, 29))mod\ x\ mod\ 2\ mod\ (51x))) + x^2, max(15x, x\ mod\ 53\ mod\ max(11, x)) + (x - 46))$ over the real number interval $[0.0, 50.0]$ (mutation-rate 0.1, number of populations 1, coefficient bounds $[-61, 61]$, depth limit 5, population size 5000, birth-rate 0.7)

Algorithm 1 Simulated Annealing

Input: p, $iterations$, $temp$, $temp_{change}$
$p_{current} \leftarrow p$
$p_{best} \leftarrow p_{current}$
for $i = 1$ **to** $iterations$ **do**
  $p_i \leftarrow ChangeTerminals(p_{current})$
  $temp \leftarrow temp \cdot temp_{change}$
  if $f(p_i) \leq f(p_{current})$ **then**
    $p_{current} \leftarrow p_i$
    if $f(p_i) \leq f(p_{best})$ **then**
      $p_{best} \leftarrow p_i$
    end if
  else if True $\sim \exp(\frac{f(p_{current}) - f(p_{p_i})}{temp})$ **then**
    $p_{current} \leftarrow p_i$
  end if
end for
return p_{best}

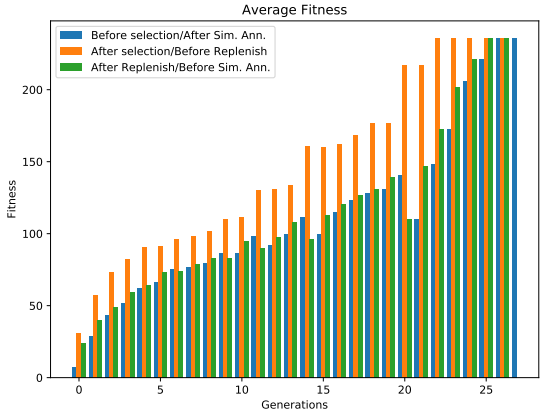

Figure 13: GP with Simulated Annealing average fitness throughout the generations within the different phases (iterations 700, temperature 70000, temperature change 0.98, mutation-rate 0.1, number of populations 1, coefficient bounds $[1, 42]$, depth limit 3, population size 5000, birth-rate 0.7)

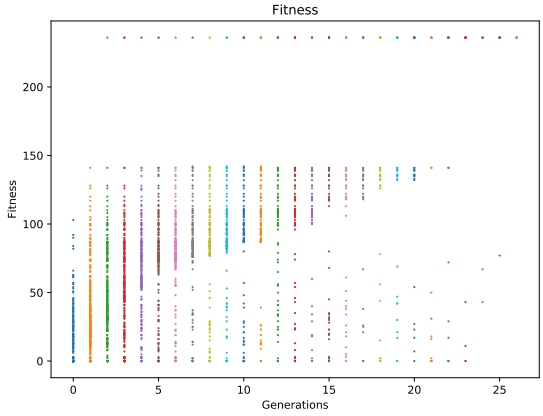

Figure 12: GP with Simulated Annealing fitness throughout the generations (iterations 700, temperature 70000, temperature change 0.98, mutation-rate 0.1, number of populations 1, coefficient bounds $[1, 42]$, depth limit 3, population size 5000, birth-rate 0.7)

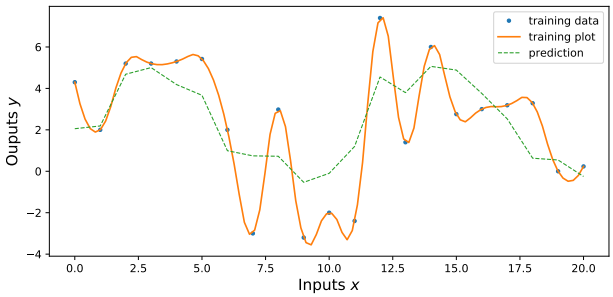

Figure 14: Attempt of a GP to model a time series, (number populations 1, population size 30000, test interval series span, birth rate 0.7, mutation 0.02, coeffs-bound (-50, 50), constraint 3.5, operator functions [add, multiply, sinleft (sin and addition with first input), sinright (sine and addition with second input)]