Proceedings of International Joint Conference on Neural Networks, Dallas, Texas, USA, August 4-9, 2013

NeuroEvolutionary Meta-Optimization

Andreas Lang
University of Geneva, CUI
7 route de Drize
1227 Carouge, Switzerland
Email: langandreas@gmx.net

Kenneth O. Stanley
Department of EECS
University of Central Florida
Orlando, FL 32816, USA
Email: kstanley@eecs.ucf.edu

Abstract— **This paper introduces a meta-optimization algorithm called NeuroEvolutionary Meta-Optimization (NEMO) that evolves an algorithm targeted at optimizing only within a specific problem class. More specifically, a form of neural network is evolved that acts as the controller of a kind of optimization algorithm that can potentially exploit problem class-specific structure. NEMO is demonstrated on several benchmark problems that confirm its ability to succeed on problems within the class on which it is trained. The key implication is that it is indeed possible to evolve this kind of meta-optimizer with a neural network-like structure, opening up a promising research direction in automatically evolving such class-specific optimizers.**

I. Introduction

The No Free Lunch Theorem for optimization (NFL) states that an optimization algorithm's "elevated performance over one class of problems is offset by performance over another class" [32]. The assumption is that such an algorithm does not take into account prior knowledge about the problem

Yet of course it is possible to exploit problem-specific knowledge if it is possible to know what kind of problem is being optimized [7, 16, 31]. One potential method for gaining such class-specific knowledge is by training a meta-optimizer on the specific class of interest [2, 18, 20, 28]. This paper introduces such an algorithm, called NeuroEvolutionary Meta-Optimization (NEMO), which evolves optimizers adapted to one problem class by training only on problems of that class. The flexibility of the approach is provided by the NeuroEvolution of Augmenting Topologies (NEAT) method [26, 27] behind it, which evolves a neural network-like structure called a compositional pattern producing network (CPPN) [24] of arbitrary complexity that in NEMO realizes the functionality of an optimization algorithm.

An optimization algorithm can be interpreted as a behavior that controls the movement through the search space. Neural networks, which are commonly applied to control problems, therefore can potentially drive such an optimization process. Hence the evolution of neural networks through NEAT can become a meta-optimization algorithm.

NEAT's ability to evolve increasingly complex structures motivates applying neuroevolution to meta-optimization because the optimal search behavior of an optimization algorithm may require high complexity to represent.

The aim of this paper is to show that such a setup can indeed produce optimizers that are particularly suited to specific problem classes. To validate this capability, NEMO is tested in several standard benchmark optimization problems and compared to a general optimizer called CHEOPS [14] that has performed well on such benchmarks in the past. The main result is that NEMO-evolved optimizers indeed can outperform the general optimizer, confirming that this form of meta-optimization is at least viable. Rather than implying that NEMO is the last word on neuroevolutionary meta-optimization, the hope is that this work inspires further investigation of the potential for such network-driven optimizers to be specialized to specific problem classes.

II. Background and Definitions

This section reviews foundational work about (meta-)optimization as well as NEAT and CPPNs, which are the foundation of the proposed algorithm.

A. Optimization

A *fitness function* is a function mapping a *search space* to a one dimensional *fitness*. For consistency with the idea that low error is best in optimization, fitness in this paper is best when it is lowest. An *optimization problem* (or *problem* for short) is defined as a search for the input to the fitness function producing the lowest fitness possible within the search space. A set of similar problems is called a *problem class*. An *optimization algorithm* (or *optimizer* for short) is an algorithm designed to find an approximate solution for the problem (i.e. a point in search space with a low fitness) within a low number of fitness function queries.

Because in this paper optimizers themselves will be evolved, the fitness of the *optimizer* is called its *meta-fitness*, which is defined as the expected best solution that can be found for a problem within a given problem class.

An optimizer is usually biased towards certain types of problems, i.e. it is more likely that the optimizer solves these intended problems well or in a low number of iterations than other problems. Such bias can be adjusted by the parameters of the optimizer. These parameters might be fixed by the experimenter, like in traditional gradient-based or metaheuristic non-gradient based algorithms [10, 18]; some algorithms [14, 29] allow adapting their parameters dynamically during each run, depending on the revealed knowledge about the current problem.

While there are mechanisms for escaping local optima [12], common optimizers tend to follow the gradient, more or less strictly, that points towards the optimum.

One relevant prior approach to the idea in this paper is Aguilar-Ruiz et al. [1], which proposes an optimization algorithm that reduces the number of fitness function queries by querying an ANN that approximates the fitness function and is adapted iteratively. This idea is pushed further in this paper to enable meta-optimization, discussed next.

B. Meta-Optimization

In this paper, a *meta-optimization problem* is defined as the search for a good optimizer for problems of a given problem class [18]. Therefore, a *meta-optimizer* is an algorithm designed to find a base-optimizer (BO) with a low meta-fitness for a problem class.

As in Vilalta and Drissi [30], *meta-* and *base*-level are distinguished as follows. The meta-optimizer searches in the *meta-search space* of *BOs*; each BO moves through the *base-search space*. Analogously, the fitness of one BO is referred to as *meta-fitness*, whereas *base-fitness* designates the fitness of a point in the base-search space. The meta-fitness of a BO is calculated by the *meta-fitness function*; the *base-fitness function* (FF) determines the base-fitness of a point in the base search space. For a given parametrized meta-optimizer, the meta-search space is always the same, though the base-search space depends on the meta-optimized problem class.

Several machine learning algorithms aim to generate explicit knowledge from a given problem through meta-learning and then to apply this knowledge to strategies that determine the ultimate behavior on this problem [19, 31]. Vilalta et al. [31] describes a general concept of meta-learning through extraction of such explicit knowledge about the problem. Schmidhuber [20] discusses the theoretical *Gödel Machine*, which can act as an optimizer and improve itself in a provably optimal way.

Algorithms for tuning parameters of optimizers exist, e.g. for Genetic Algorithms [5, 33], Evolutionary Algorithms [22], Particle Swarm Optimization [11, 17] and Ant Colony Optimization Algorithms [3]. Neumüller et al. [13] implements a general framework for two interchangeable optimizers, wherein one optimizes the parameters of the other one.

The unique idea in this paper is to apply neuroevolution at the meta-optimization algorithm for generating specific optimizers. The next section introduces the primary neuroevolution algorithm adopted for this purpose.

C. NeuroEvolution of Augmenting Topologies

The *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm [26, 27] evolves both the topology and weights of artificial neural networks (ANNs). An important property of NEAT is that evolution begins with small, simple networks that become more complex over evolution by adding new connections and new nodes. Due to this increasing complexity, low complexity solutions are likely found earlier in evolution than complex solutions [26].

NEAT traditionally evolved ANNs but more recently has been applied to evolving compositional pattern producing networks (CPPNs) as well [24]. CPPNs are similar to ANNs with the exception that the nodes' activation functions within the same network are not necessarily all the same. Instead, they are chosen from a given set of activation functions, including sigmoid, Gaussian, and sine. Patterns produced by a CPPN tend to exhibit regularities like (imperfect) symmetry and repetition (with variation) [21, 24], which is why they are also suited to representing optimizers that might similarly exploit regularities in problem structure.

III. NeuroEvolutionary Meta-Optimization

For the purpose of reporting results, an important assumption in this paper is that FF queries during training are free, i.e. they do not affect the evaluation of FF queries of the resulting BO. This assumption is justified because of course the idea in meta-optimization is that once an optimizer is trained, it can be applied to an indefinite number of future instances of its intended problem class. Therefore, when measuring the performance of the optimizer, the focus is on how well it performs within its problem class, as opposed to how long it took to train during meta-optimization. The proposed approach to meta-optimization aims to realize three key conditions:

- Completeness of meta-search space: The search space of the meta-optimizer is complete, i.e. it can potentially find every possible BO.
- Bias of meta-search space: The meta-search can be biased by the experimenter to influence which kind of optimizers are more likely to be found early during search process.
- Approximation the optimum of the meta-search space: The meta-optimizer approximates or, in a weaker sense, strives towards the optimal BO. That is, the meta-optimizer evolves a BO biased to the given problem class, i.e. the generated BO performs better on this problem class than BOs evolved for other problem classes. This condition coincides with the NFL implying that no general optimizer being good at all problem classes exists.

Furthermore, the meta-optimizer should be able to find BOs that take into account all collected knowledge about the problem for the choice of the next query point. The meta-search space contains BOs that aim for areas of the base-search space that contain high information content, thereby revealing information about where the optimum can be found.

A. The NEMO Algorithm

The *NeuroEvolutionary Meta-Optimization* algorithm (NEMO) is an evolutionary algorithm that evolves networks wherein each network encodes an optimization algorithm. That is, the meta-optimizer is a neuroevolutionary algorithm and the BO is a *network-driven optimizer*.

NEAT begins the meta-optimizer with simple networks and increases their complexity over the course of evolution. Therefore, the meta-search is biased towards optimizers represented by simple networks, i.e. the encoding of the optimizer as a network impacts the bias of the meta-search.

The primary meta-optimization algorithm, which is a simple iterative EA loop run by NEAT, is shown in Algorithm 1.

```
repeat
    // NEAT evolves CPPNs
    cppns ← NEAT(cppns);
    foreach cppn in cppns do
        // Determine meta-fitness of optimizer
        cppn.metaFitness ← MetaFitnessFct(cppn);
    end
until stop criterion is met;
```

Algorithm 1: NEMO algorithm with NEAT evolving CPPNs.

The universal approximation theorem [6] states that every continuous function can be approximated by an ANN. Because ANNs are a subset of CPPNs, this theorem also applies on CPPNs. The optimal optimizer can be described as a function having as inputs and outputs the past query points and best next query point, respectively. Therefore, the meta-search space of BOs is complete.

The meta-fitness function determines the fitness of a CPPN. Therefore, the optimizer corresponding to the CPPN is executed on several base-problems of the given base-problem class and finds for each of these base-problems a solution. The average over the performances on the base-problems defines the meta-fitness returned by the meta-fitness function for the candidate CPPN.

B. Network-Driven Optimizer

A Network-Driven Optimizer (NDO) is a continuous optimization algorithm whose movement through base-search space is determined by a network. The basic algorithm works as follows. To begin, one random point is chosen in the base-search space. From that step onward, the FF of each point is queried and the resulting base-search points and their corresponding fitnesses are saved. As the algorithm progresses, a loop is executed until a stop criterion is met. This loop first inputs all knowledge acquired in past into the network, which is activated to output the *next base-search point* for which the FF will be queried (and whose result is subsequently saved as usual). The skeleton algorithm of the NDO is given in Algorithm 2.

C. Determining the Next Points in the Search

Different methods are conceivable by which the CPPN can determine the next search point depending on the past search points and their fitnesses. This section explores the one investigated in this paper. For simplicity, because this section focuses only the base-layer, the word *base* in front of the words *optimizer*, *search space* and *fitness* is omitted.

Given a query point $p \in P$ and its fitness $FF(p)$, the CPPN is intended to output a *partial prediction value* for any point (called target point t) of the search space. Using this formulation, summing the partial prediction values at t over the outputs of the CPPN for all query points P results in the *global prediction value* $g(t)$ of the target point in question.

```
// Choose position of first query point randomly
queryPointFirst.pos ←
random position in search space;
// Calculate first query point's fitness
queryPointFirst.fitness ← FF(queryPoints);
// Add it as first element to array queryPoints
queryPoints.append(queryPointFirst);
repeat
    // Calculate position of next query point
    queryPointNew.pos ← ApplyCPPN(queryPoints);
    // Query the FF for the new query point
    queryPointNew.fitness ← FF(queryPointNew);
    // Add the new query point to list of query points
    queryPoints.append(queryPointNew);
until stop criterion is met;
```

Algorithm 2: NDO skeleton algorithm (the network is a CPPN). The abstract function applyCPPN is specified by the chosen NDO configuration, e.g. applyCPPN_Prediction.

The target point with the highest global prediction value is the point for which the FF will be queried next.

The inputs to the CPPN include a bias (like in regular ANNs) and the fitness of p; additionally the CPPN takes as inputs either coordinates of p and t or the Euclidean distance between both points as inputs to the CPPN, as well as optionally the current iteration number. The output is the partial prediction value given p and t.

The intent of the partial prediction values is to help determine the position of the next queried search point t_{max}, where $t_{max} \in S$ and S is the search space. In particular, the next queried search point is the one for which the sum of the outputs of the CPPN with all possible past queried points is maximal, i.e. t_{max} must satisfy:

$$g(t) := \sum_{p \in P} CPPN(FF(p), p, t),$$

$$g(t_{max}) \geq g(t),$$

$$\forall t \in S,$$

The next target point is theoretically defined by t_{max}. Because calculating t_{max} analytically is prohibitive (depending on the set of activation functions of the CPPN), it is approximated in the function applyCPPN_prediction(queryPoints) in practice.

Each query point p is in the set of query points P. The function queried(p) indicates if the FF has been queried at position p; in the positive case then lookup(p) already contains the corresponding fitness. The global prediction value for p, referred to as pred(p), is updated in every iteration as the sum of the outputs of the CPPN for all query points that have been queried before, i.e. for which queried(p) = true. The query point with the highest global prediction value is named p_max.

If p_max has been queried before, it will be *expanded*; otherwise the FF will be queried for this point. Expanding a query point p_max means adding randomly n new query points within a hypercube of size $2 \cdot \alpha \cdot r$, centered at p_max, where r is the Manhattan distance of p_max to its closest queried query point. (this expansion of query points has similarities with the Bees Algorithm [23]). The algorithm described in this section is detailed in Algorithm 3.

```
foreach p in P do
    pred(p) ← 0;
    foreach
    p_queried in P where queried(p_queried) = true
    do
        pred(p) += CPPN(lookup(p), p, p_queried);
    end
end
p_max ← p in P with highest pred(p);
if queried(p_max) then
    // Expand query point
    append new query points to P;
else
    // Query FF for query point
    lookup(p_max) ← FF(p_max);
end
```

Algorithm 3: Specification of the function applyCPPN_prediction.

D. Example Execution

An example with a one-dimensional base-fitness function of how an iteration of NEMO proceeds is shown in Figure 1.

An example execution on the two-dimensional Rosenbrock test function is shown in figure 2. Until the 12th iteration, the query points tend to be distributed nearly uniformly over the whole search space, which can be interpreted as *exploration*. Then, from iteration 60 onward, the search begins to concentrate increasingly on one part of the space – an *exploitation*-like behavior.

These two examples are driven by the two CPPNs depicted in figure 3.

IV. Experiments

This section describes the two-dimensional test problems where the main idea is validated. The aim in the experiments is to investigate whether NEMO indeed can create optimizers that perform well on a specific class of problems.

All experiments are implemented with the *SharpNeat 2* software package from Green [8]. The inputs to the CPPN are its bias, the fitness of the queried point, and the Euclidean distance between queried point and target point. The set of activation functions of the CPPN consists of linear, sigmoid, Gaussian and sine function. The range of possible weights is $[-5, 5]$. The size of expansion of query points is determined by $\alpha = 3$.

A CPPN is *sampled* on a problem class by executing the corresponding optimizer on one randomly chosen problem of the problem class and is characterized by the number of FF queries it needs to make. A sample is successful if the

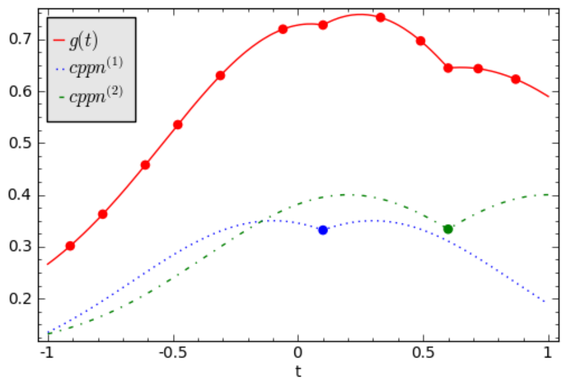

Fig. 1. Example of an iteration of NEMO with the CPPN from figure 3, top. The x-axis between -1 and 1 is the one-dimensional search space; the y-axis measures the partial/global prediction value. Two points have been queried already in this example; they are indicated by a blue and a green point. The points' x-values indicate their position in search space. Given the position of the blue query point, the network predicts analytically for each point of the search space how useful querying this point would be for finding the optimum, indicated by the interrupted blue line (named partial prediction value); analogously for the green point and graph. Summing over both functions, results in the continuous red line of which the maximum indicates where to query or expand next. Practically, this maximum is not found analytically but only approximately by sampling the function at several positions (red points), in this example p_max $= 0.33$ will be queried next.

distance of the best found base fitness to the optimal one is below the threshold of 10^{-6}, which is the success criterion of Nieländer [15]. Execution is stopped when the success criterion is met or a maximum number of iterations of $15,000$ is reached (note that the number of FF queries is lower than the number of iterations).

A *trial* for a particular CPPN optimizer consists of 15 samples and a *test*, as in Nieländer [15], of 10 samples. A test is described by the number of successful runs as well as the average and variance of FF queries of the samples.

In one *run*, NEMO trains an optimizer on one problem class. NEAT evolves 80 generations with a population size of 150 and a species size of 10. In every generation, every individual's meta-fitness is determined by the average number of FF queries of the successful samples of one trial.

The total number of FF queries for training (one run) an optimizer on one problem class is lower than $2.7 \cdot 10^9$.

A. Test Functions

NEMO is trained and tested on four problem classes, each of which is respectively determined by one test FF. A problem class is the set of all shifts F_i of a test function f_i for which its reference point (x^*, y^*) is within the predefined search space S. That way, a range of similar optimization problems can be generated from a particular function. The optimizer can query F_i for each point $(x, y) \in S$ within the search space.

$$F_i(x, y) = f_i(x - x^*, y - y^*).$$

The Rosenbrock function f_1 has a distinct parabolic valley, which slightly descends towards its minimum $f(0, 0) = 0$:

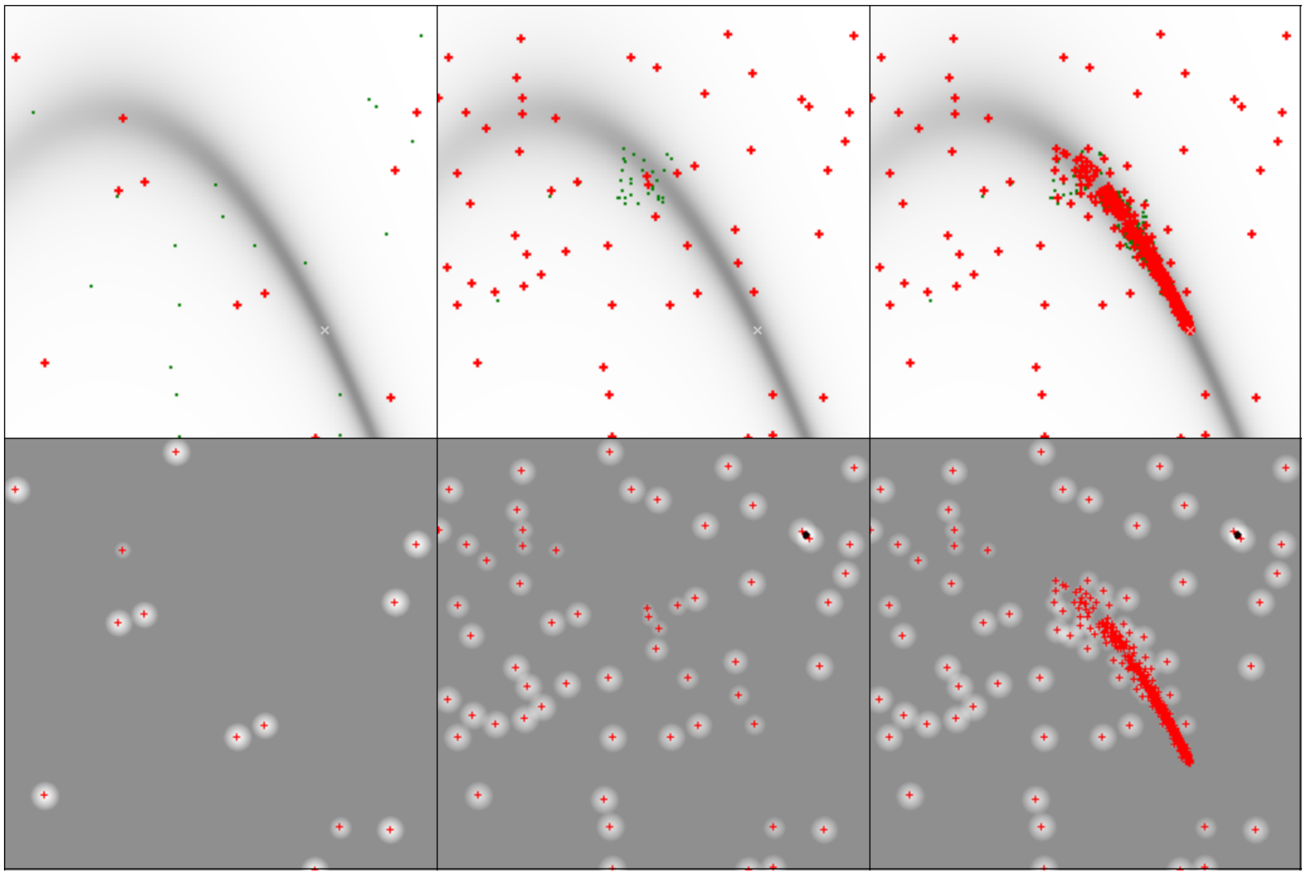
Fig. 2. Example run of NEMO with the CPPN from figure 3, bottom, trained on the Rosenbrock test function. The columns show the 12th iteration (1st column), 60th iteration (2nd column), 702nd iteration (3rd column). Top: In the background, the more gray a pixel is, the higher the base-fitness at this point. Query points are indicated by green dots if they have not been queried yet, and otherwise by a red cross. The optimum is designated by the white "x". Bottom: The global prediction value for the position corresponding to each pixel is shown, where the gray value indicates the global prediction value. The thin red crosses indicate the query points that have been queried already.

$$f_1(x,y) = 100(y-x^2)^2 + (1-x)^2,$$
$$S_1 = [0,2] \times [0,2].$$

The adapted two-dimensional Shubert function f_2 has 18 global optima and further 743 local optima [14].

$$f_2(x,y) = \frac{1}{100} \cdot \Big(\sum_{j=1}^{5} j \cdot \cos\big((j+1) \cdot x + j\big) \Big) \cdot \Big(\sum_{j=1}^{5} j \cdot \cos\big((j+1) \cdot y + j\big) \Big),$$
$$S_2 = [-10,10] \times [-10,10].$$

The scaled and discretized Ratz function from Nieländer [14] has two globally optimal areas around $(-1.457522105, 0)$ and $(1.457522105, 0)$.

$$f_3(x,y) = \left\lfloor 10 \cdot \sin(x^2 + 2 \cdot y^2) \cdot e^{-x^2-y^2} + 1.068 \right\rfloor$$
$$S_3 = [-3,3] \times [-3,3]$$

The fourth test FF f_4 has the shape of a volcano: From the foot of the volcano the fitness decreases towards the center but in a small central square of edge length $2 \cdot r_{vent}$, the fitness is globally optimal.

$$f_4(x,y) = \begin{cases} 0 & \text{if } |x| < r_{vent} \text{ and } |y| < r_{vent} \\ 10 - |x| - |y| & \text{otherwise} \end{cases}$$
$$S_4 = [-1,1] \times [-1,1]$$

where $|z|$ designates the absolute value of z and $r_{vent} = 0.01$.

B. Comparison

On every problem class, NEMO was run five times to find a corresponding optimizer. Each resulting optimizer was tested on all four problem classes (with the hope of course that it performs best on the class for which is was trained). In each trial and each test, one FF is chosen randomly from the problem class, i.e. (x^*, y^*) is chosen from a uniform random distribution in S.

The sets of five optimizers trained on one problem class and tested on a (possibly different) problem class

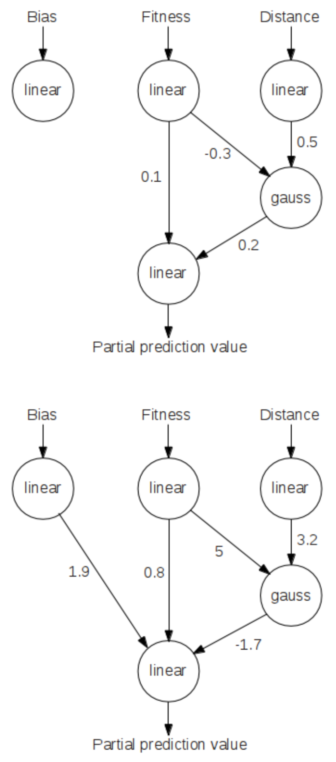

Fig. 3. Example CPPNs. The circles designate nodes and the corresponding activation function for each node is shown in its center. Inputs are bias, the fitness of the queried query point, and the distance between the queried and target query points. Values are propagated with indicated weights in the direction of the arrows. The top network is a manually constructed CPPN used in figure 1; note that the bias is not connected. The bottom CPPN is evolved by NEMO (depicted weights are rounded) and an example run is shown in figure 2.

are compared to the test results of the CHEOPS method [14] (which stands for *Das Chemnitzer hybrid-evolutionäre Optimierungssystem* or in English, *Chemnitz Hybrid Evolutionary Optimization System*) on the latter problem class (for f_1, f_2, f_3) and to theoretical results of random search (for f_4), respectively. CHEOPS is a good choice for comparisons because for all given problem classes its 10 samples were successful. If, for given training and testing problem classes, all of the five runs' CPPN-based optimizer tests are successful in ten out of ten samples, then the number of FF queries are tested for significance $p < 0.05$ according to the Random-Effects Model described in Borenstein et al. [4]. If at least one test was not successful, then NEMO's trained optimizers on this problem class are recorded as not significantly better than CHEOPS. Note that the CHEOPS experiments were not adaptive.

Because optimization algorithms normally follow the gradient of fitness, on the volcano problem a normal optimizer would move away from the central vent. In comparison, a completely random distribution of search points would perform better. Therefore, unlike the other three problem classes, the evolved optimizer on the volcano problem is compared to random search.

V. Results

Successful samples are summarized in table I and more detailed information about the test cases for ten successful samples in all five runs (with comparison data from other algorithms) is shown in table II.

The BO trained on Rosenbrock, Ratz and the volcano-like function reach significantly higher performance than the results of CHEOPS or random search, respectively, on the same test function. On the other hand, training NEMO on Shubert does not result in better optimizers than CHEOPS.

NEMO trained on one problem class is never significantly better, when tested on a *different* problem class, than CHEOPS or random search. Thus optimizers trained with NEMO perform best on the problem classes for which they were trained, as should be the case for a meta-optimizer.

train	Run	test f_1	test f_2	test f_3	test f_4
f_1	1	10	1	5	3
	2	10	3	8	7
	3	10	0	5	5
	4	10	0	3	5
	5	10	1	2	3
f_2	1	8	10	8	5
	2	9	8	6	5
	3	9	10	5	4
	4	0	0	0	7
	5	8	9	10	0
f_3	1	0	0	10	0
	2	0	0	10	3
	3	0	0	10	2
	4	0	0	10	0
	5	0	0	10	2
f_4	1	0	0	0	10
	2	0	0	2	10
	3	0	0	3	10
	4	0	0	1	10
	5	0	0	0	10

TABLE I. Successful samples: NEMO was trained on the problem class indicated in each row and the resulting base-optimizer was tested on the problem class indicated in the column. For each test, the number of successful samples out of 10 samples is indicated. The main result is that NEMO-trained optimizers excel on the class for which they were trained.

VI. Discussion

Finding good optimizers for the problem classes Rosenbrock, Ratz and volcano but not for the Shubert class suggests that the implementation of NEMO in this paper is biased towards mastering the former three. Thus a meta-optimizer itself may only be suited towards producing optimizers for a subset of all possible problem classes. Nevertheless that capability is still useful if such problem classes are important.

Furthermore, that optimizers trained for one problem class perform significantly worse on others supports the idea that NEMO can indeed find optimizers biased to a given problem class. Solving the Ratz function further suggests that the optimizer can also cope with discrete FF, while finding the optimum of volcano shows that an optimizer is able to exploit information content.

P-NEMO trained and tested on f_x				CHEOPS$*$/Random$\dagger$	
FF	Run	$\overline{x}$	σ	$\overline{x}$	σ
f_1	1	3091.5	2795.6	5278.5 $*$	2080.2 $*$
	2	4366.1	3272.5		
	3	2391.9	2123.1		
	4	2709.3	1031.6		
	5	2320.7	2230.0		
f_3	1	966.7	769.7	4081.5 $*$	708.4 $*$
	2	2257.2	1637.4		
	3	899.1	624.6		
	4	1928.4	2214.7		
	5	1275.1	888.7		
f_4	1	140.5	48.3	4421.8 $\dagger$	4321.7 $\dagger$
	2	289.0	420.9		
	3	117.3	36.8		
	4	130.2	46.8		
	5	111.9	36.1		

TABLE II. Training NEMO on f_1, f_3 and f_4 and testing it on the same problem class resulted in ten out of ten successful samples in all five runs. Mean $\overline{x}$ and standard deviation σ of the number of FF queries for these three test cases are indicated both for NEMO and the compared approaches, namely previously reported results from CHEOPS [14] for f_1 and f_3 and theoretical performance of random search for f_4. Because NEMO trained optimizers specialized for each class, it can perform more effectively than generic optimizers like CHEOPS when the optimizer is applied to appropriate classes.

The bias of the meta-search consists of the CPPN and its possible activation functions, which in effect determine the structure of the meta-search space, and the parameters of NEAT, which are responsible for the movement through this space. Because NEMO finds basic search behaviors like exploration and exploitation, it may be biased towards discovering BOs with such properties.

VII. Future Work

The approximation of t_{max} by the optimizer in the proposed implementation does not always match the perfect analytical solution. This discrepancy can potentially impact the meta-search for the right CPPN. To allow the calculation of an analytical solution and thus to avoid such distortion, the activation functions of the CPPN could be changed or a different formulation could be devised in the future.

Information different from fitness, e.g. a gradient measure or specific domain knowledge, can be exploited simply by adding more input nodes to the network. This information can depend on the position in the search space or be constant for a given problem, which would enable NEMO to take advantage of explicit prior knowledge (c.f. [19, 31]). In addition, NEMO can be extended for classes of higher dimensional problems by adding more input nodes for each dimension.

Alternative formulations of the optimizer may be better biased towards real-world problems or other problem classes. For example, the optimizer could be represented by an ANN and be evolved indirectly by an algorithm like HyperNEAT [25]. Position within the search space could be represented by position within the substrate of HyperNEAT and the output could determine the next query point. In this way, the network could find the query point with only one activation and based on an unlimited number of input query points. Another possibility of interpreting the network is as estimation of distribution so that the corresponding NDO is an Estimation of Distribution Algorithm [9].

VIII. Conclusion

The meta-optimization algorithm NEMO and a corresponding base-optimization configuration were proposed. NEMO's meta-search space is complete due to the universal approximation theorem; its meta-search can be biased via the optimizer configuration and the NEAT parameters; NEMO is able to find a solution towards the optimum of the meta-search space, i.e. an optimizer adapted to the trained problem class, as shown empirically.

A trained optimizer in general can theoretically depend on all knowledge found about the base-problem for choosing the next query point, although this idea remains to be demonstrated. However, the results do show that network-based optimizers are able to exploit alternative information content, i.e. by striving towards promising areas despite potentially decreasing fitness.

In conclusion, NEMO is a promising novel direction for meta-optimization that evolves a network. As such, it suggests a new path forward in meta-optimization where more such network configurations can be tested with evolution for their ability to exploit problem-specific information.

IX. Acknowledgments

Special thanks to German National Academic Foundation for financial support, to Aubin Jarry for support in preparation, to Ulf Nieländer for providing unpublished CHEOPS data, and to Sylvain Sardy for assistance in statistics.

References

[1] J. S. Aguilar-Ruiz, D. Mateos, and D. S. Rodríguez. Evolutionary neuroestimation of fitness functions. In F. Moura-Pires and S. Abreu, editors, *Progress in Artificial Intelligence, 11th Protuguese Conference on Artificial Intelligence, EPIA 2003, Beja, Portugal, December 4-7, 2003, Proceedings*, volume 2902 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2003.

[2] J. Baxter. Theoretical models of learning to learn. In *T. Mitchell and S. Thrun (Eds.), Learning*, pages 71–94. Kluwer, 1997.

[3] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. Langdon, editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers, 2002.

[4] M. Borenstein, L. V. Hedges, J. P. T. Higgins, and H. R. Rothstein. *Introduction to Meta-Analysis*. Wiley & Sons, Ltd, 2009.

[5] J. Clune, S. Goings, B. Punch, and E. Goodman. Investigations in meta-gas: panaceas or pipe dreams? In F. Rothlauf, editor, *GECCO Workshops*, pages 235–241. ACM, 2005.

[6] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

[7] C. G. Giraud-Carrier, R. Vilalta, and P. Brazdil. Introduction to the special issue on meta-learning. *Machine Learning*, 54(3):187–193, 2004.

[8] C. Green. Sharpneat 2, September 2010.

[9] M. Hauschild and M. Pelikan. An introduction and survey of estimation of distribution algorithms. *Swarm and Evolutionary Computation*, 1(3):111–128, 2011.

[10] M. Khajehzadeh, M. R. Taha, A. El-Shafie, and M. Eslami. A survey on meta-heuristic global optimization algorithms. *Research Journal of Applied Sciences, Engineering and Technology*, 3, 2011.

[11] M. Meissner, M. Schmuker, and G. Schneider. Optimized particle swarm optimization (opso) and its application to artificial neural network training. *BMC Bioinformatics*, 7:125, 2006.

[12] P. Mills, E. Tsang, Q. Zhang, and J. Ford. A survey of ai-based meta-heuristics for dealing with local optima in local search. Technical report, University of Essex, 2004.

[13] C. Neumüller, S. Wagner, G. Kronberger, and M. Affenzeller. Parameter meta-optimization of metaheuristic optimization algorithms. In *EUROCAST (1)*, volume 6927 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 2011.

[14] U. Nieländer. *CHEOPS: das Chemnitzer hybrid-evolutionäre Optimierungssystem*. PhD thesis, 2009.

[15] U. Nieländer. The Chemnitz Hybrid Evolutionary Optimization System. In J. Filipe and J. Kacprzyk, editors, *IJCCI (ICEC)*, pages 311–320. SciTePress, 2010.

[16] R. M. Oleg Kovářík. Meta-learning and meta-optimization. Technical report, Czech Technical University in Prague, 2012.

[17] M. E. H. Pedersen and A. J. Chipperfield. Simplifying particle swarm optimization. *Appl. Soft Comput.*, 10(2): 618–628, 2010.

[18] M. E. H. Pedersen. *Tuning & Simplifying Heuristical Optimization*. PhD thesis, University of Southampton, January 2010.

[19] T. Schaul and J. Schmidhuber. Metalearning. 5(6):4650, 2010.

[20] J. Schmidhuber. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. Technical Report IDSIA-19-03, arXiv:cs.LO/0309048 v1, IDSIA, Manno-Lugano, Switzerland, September 2003.

[21] J. Secretan, N. Beato, D. B. D.Ambrosio, A. Rodriguez, A. Campbell, J. T. Folsom-Kovarik, and K. O. Stanley. Picbreeder: A case study in collaborative evolutionary exploration of design space. *Evolutionary Computation*, 19(3):345–371, 2011.

[22] S. K. Smit and A. E. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 399–406, 2009.

[23] P. Songmuang and M. Ueno. Bees algorithm for construction of multiple test forms in e-testing. *IEEE Transactions on Learning Technologies*, 4(3):209–221, 2011.

[24] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines, Special Issue on Developmental Systems*, 2007.

[25] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

[26] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[27] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[28] S. Thrun. Lifelong learning: A case study. Technical report, Carnegie Mellon University, 1995.

[29] A. Tuson and P. Ross. Adapting operator settings in genetic algorithms. *Evolutionary Computation*, 6(2): 161–184, 1998.

[30] R. Vilalta and Y. Drissi. Research directions in meta-learning. In *Proceedings of the International Conference on Artificial Intelligence, Las Vegas*, 2001.

[31] R. Vilalta, C. G. Giraud-Carrier, and P. Brazdil. Meta-learning - concepts and techniques. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 717–731. Springer, 2010.

[32] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[33] B. Yuan. A hybrid approach to parameter tuning in genetic algorithms. In *In IEEE International Conference on Evolutionary Computation*, 2005.