Evolutionary Reinforcement Learning of Artificial Neural Networks

Nils T Siebel* and Gerald Sommer
Cognitive Systems Group, Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany

Abstract. In this article we describe EANT2, *Evolutionary Acquisition of Neural Topologies, Version 2*, a method that creates neural networks by evolutionary reinforcement learning. The structure of the networks is developed using mutation operators, starting from a minimal structure. Their parameters are optimised using CMA-ES, *Covariance Matrix Adaptation Evolution Strategy*, a derandomised variant of evolution strategies. EANT2 can create neural networks that are very specialised; they achieve a very good performance while being relatively small. This can be seen in experiments where our method competes with a different one, called NEAT, *NeuroEvolution of Augmenting Topologies*, to create networks that control a robot in a visual servoing scenario.

1 Introduction

Artificial neural networks are computer constructs inspired by the neural structure of the brain. The aim is to approximate the vast learning and signal processing power of the human brain by mimicking its structure and mechanisms. In an artificial neural network (often simply called "neural network"), interconnected neural nodes allow the flow of signals from special input nodes to designated output nodes [23]. This very general concept allows neural networks to be applied to problems in the sciences, engineering and even economics [3, 13, 21, 22, 28]. A further advantage of neural networks is the fact that learning strategies exist that enable them to adapt to a problem.

When a neural network is to be developed for a given problem, two aspects need to be considered:

1. What should be the *structure* (or, *topology*) of the network? More precisely, how many neural nodes does the network need in order to fulfil the demands of the given task, and what connections should be made between these nodes?
2. Given the structure of the neural network, what are the optimal values for its *parameters*? This includes the weights of the connections and possibly other parameters.

1.1 Current Practice

Traditionally the solution to aspect 1, the network's *structure*, is found by trial and error, or somehow determined beforehand using "intuition". Finding the solution to aspect 2, its *parameters*, is therefore the only aspect that is usually considered in the literature. It requires optimisation in a parameter space that can have a very high dimensionality—for difficult tasks it can be up to several hundred. This so-called "curse of dimensionality" is a significant obstacle in machine learning problems[1] [2, 18]. Most of these parameter learning methods can be viewed as a straightforward application of local optimisation algorithms and/or statistical parameter estimation. The popular backpropagation algorithm [23, chap. 7], for instance, is, in effect, a stochastic gradient descent optimisation algorithm [25, chap. 5].

1.2 Problems and Biology-inspired Solutions

The traditional methods described above have the following deficiencies:

1. The common approach to pre-design the network structure is difficult or even infeasible for complicated tasks. It can also result in overly complex networks if the designer cannot find a small structure that solves the task.

*Corresponding author. E-Mail: nils "at" siebel-research.de

[1] When training a network's parameters by examples (e.g. supervised learning) it means that the number of training examples needed increases exponentially with the dimension of the parameter space. When using other methods of determining the parameters (e.g. reinforcement learning, as it is done here) the effects are different but equally detrimental.

This article was published in the International Journal of Hybrid Intelligent Systems 4(3): 171-183, October 2007.

2. Determining the network parameters by local optimisation algorithms like gradient descent-type methods is impracticable for large problems. It is known from mathematical optimisation theory that these algorithms tend to get stuck in local minima [20]. They only work well with very simple (e.g., convex) target functions or if an approximate solution is known beforehand. *(ibid.)*

In short, these methods lack generality and can therefore only be used to design neural networks for a small class of tasks. They are engineering-type approaches; there is nothing wrong with that if one needs to solve only a single, more or less constant problem[2] but it makes them unsatisfactory from a scientific point of view.

In recent years a number of methods have been introduced to overcome these deficiencies by replacing the traditional approaches by more general ones that are inspired by biology. Evolutionary theory tells us that the *structure* of the brain has been developed over a long period of time, starting from simple structures and getting more complex over time. In contrast to that, the *connections* between biological neurons are modified by experience, i.e. learned and refined over a much shorter time span.

In this article we describe a method, called *EANT2, Evolutionary Acquisition of Neural Topologies, Version 2*, that works in very much the same way to create a neural network as a solution to a given task. It is a very general learning algorithm that does not use any pre-defined knowledge of the task or the required solution. Instead, EANT2 uses evolutionary search methods on two levels:

1. In an outer optimisation loop called *structural exploration* new neural *structures* are developed by gradually adding new structure to an initially minimal network that is used as a starting point.

2. In an inner optimisation loop called *structural exploitation* the *parameters* of all currently considered structures are adjusted to maximise the performance of the networks on the given task.

To further develop and test this method, we have created a simulation of a visual servoing scenario: A robot arm with an attached hand is to be controlled by the neural network to move to a position where an object can be picked up. The only input data available to the network is visual data from a camera that overlooks the scene. EANT2 was used with a complete simulation of this visual servoing scenario to learn networks by reinforcement learning. In this article we present results from these experiments with EANT2 and compare them to results obtained by NEAT, a similar method, on the same problem.

The remainder of this article is organised as follows. Section 2 contains an overview over related methods for evolutionary neural network learning. Section 3 describes EANT2, our approach to a solution. In Section 4 we formulate the visual servoing problem that is used for testing the learning methods and review other visual servoing methods. Section 5 contains results from experiments with EANT2 and NEAT; Section 6 concludes the article.

2 Related Work: Methods for Evolutionary Learning of Neural Networks

In this section we review existing methods for evolutionary neural network learning. The paradigm is to learn *both the structure (topology) and the parameters of neural networks* with evolutionary algorithms without being given any information about the nature of the problem. The development of networks is realised through reinforcement learning [27]. This means that candidate solutions which have been generated by the algorithm are evaluated by testing them on the target application. A scalar value of their "fitness" is fed back to the algorithm to help it judge and determine what to do with this candidate. These learning algorithms do not depend on the availability of input-output pairs of the neural network as supervised learning methods do. This makes them applicable to a wider range of problems.

Until recently, only small neural networks have been evolved by evolutionary means [32]. According to Yao, a main reason is the difficulty of evaluating the exact fitness of a newly found structure: In order to fully evaluate a *structure* one needs to find the optimal (or, some near-optimal) *parameters* for it. However, the search for good parameters for a given structure has a high computational complexity unless the problem is very simple. *(ibid.)*

In order to avoid this problem most recent approaches evolve the structure and parameters of the neural networks simultaneously. Examples include EPNet [33], GNARL [1] and NEAT [26]. EPNet uses a modified backpropagation algorithm for parameter optimisation—i.e. a local search method. The muta-

[2] The No Free Lunch Theorem [31] states that solutions that are specifically designed for a particular task always perform better at this task than more general methods. However, they perform worse on most or all other tasks, or if the task changes.

tion operators for searching the space of neural structures are addition and deletion of neural nodes and connections. No crossover is used. A tendency to remove connections/nodes rather than to add new ones is realised in the algorithm. This is done to counteract the "bloat" phenomenon—i.e. ever growing networks with only little fitness improvement, also called "survival of the fattest" [6]. GNARL is similar in that is also uses no crossover during structural mutation. However, it uses an evolutionary algorithm for parameter adjustments. Both parametrical and structural mutation use a "temperature" measure to determine whether large or small random modifications should be applied—a concept known from simulated annealing [17]. In order to calculate the current temperature, some knowledge about the "ideal solution" to the problem, e.g. the maximum fitness, is needed.

The author groups of both EPNet and GNARL are of the opinion that using crossover is not useful during the evolutionary development of neural networks [33, 1]. The research work underlying NEAT, on the other hand, seems to suggest otherwise. The authors have designed and used a crossover operator that allows to produce valid offspring from two given neural networks by first aligning similar or equal subnetworks and then exchanging differing parts. Like GNARL, NEAT uses evolutionary algorithms for both parametrical and structural mutation. However, the probabilities and standard deviations used for random mutation are constant over time. NEAT also incorporates the concept of speciation, i.e. separated subpopulations that aim at cultivating and preserving diversity in the population [6, chap. 9].

3 Developing Neural Networks with EANT2

3.1 Introduction and Historical Notes

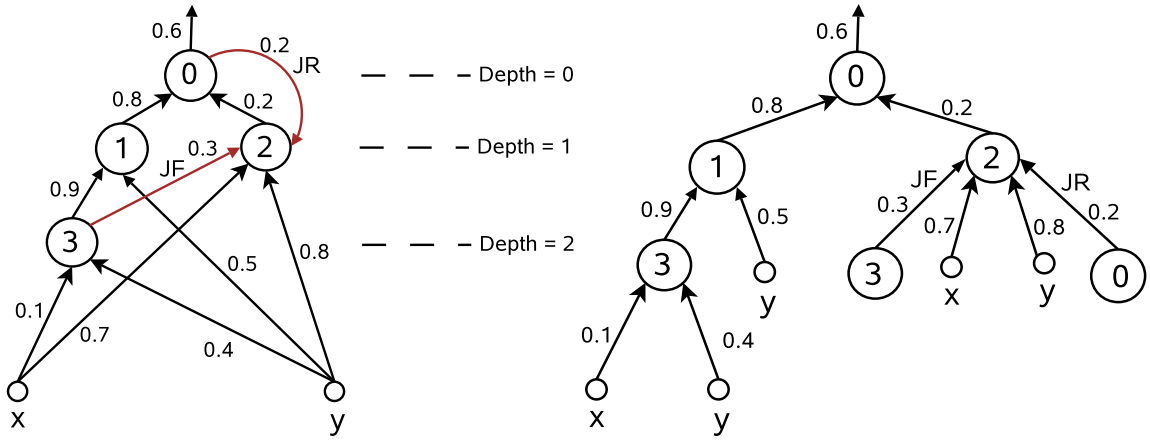
EANT ("Evolutionary Acquisition of Neural Topologies") is an evolutionary reinforcement learning system that realises neural network learning with evolutionary algorithms both for the structural and the parametrical part. It was conceived by Yohannes Kassahun within his PhD project in our research group, which was completed in 2006 [15]. Starting end 2005 EANT has been developed further, its search for structures and parameters replaced by new methods that enable it to find better performing networks and find them faster [24]. In this article we will exclusively focus on this improved version of EANT, called EANT2.

3.2 Representation: The Linear Genome

EANT2 uses a biology-inspired genetic encoding of a neural networks, a *linear genome* of network elements. A gene can be a neuron, an input to the neural network, a bias or a connection between two neurons. There are also "irregular" connections between neural genes which we call "jumper connections". Jumper genes can encode either forward or recurrent connections. Figure 1 shows an example encoding of a neural network using a linear genome. The figures show (a) the neural network to be encoded. It has one forward and one recurrent jumper connection; (b) the neural network interpreted as a tree structure; and (c) the linear genome encoding the neural network. In the linear genome, `N` stands for a neuron, `I` for an input to the neural network, `JF` for a forward jumper connection, and `JR` for a recurrent jumper connection. The numbers beside `N` represent the global identification numbers of the neurons, `x` and `y` are the inputs coded by input genes. As can be seen in the figure, a linear genome can be interpreted as a tree based program if one considers all the inputs to the network and all jumper connections as terminals.

The linear genome encodes the topology of the neural network implicitly in the ordering of the elements of the linear genome. Linear genomes can therefore be evaluated, without decoding them, similar to the way mathematical expressions in postfix notation are evaluated. For example, a neuron gene is followed by its input genes. In order to evaluate it, one can traverse the linear genome from back to front, pushing inputs onto a stack. When encountering a neuron gene one pops as many genes from the stack as there are inputs to the neuron (the number of inputs is stored in the neuron), using their values as input values. The resulting evaluated neuron is again pushed onto the stack, enabling this subnetwork to be used as an input to other neurons. Connection ("jumper") genes make it possible for neuron outputs to be used as input to more than one neuron, see `JF3` in the example above. Together with the bias neurons that are implemented as having a constant value of 1, the linear genome can encode an arbitrary neural network in a very compact format. The length of the linear genome is equal to the number of synaptic network weights.

If one assigns integer values to the genes of a linear genome such that the integer values show the difference between the number of outputs and number of inputs to the genes, one obtains the following rules useful in the evolution of the neural controllers:



(a) Original neural network

(b) Same network in tree format

N 0 W=0.6	N 1 W=0.8	N 3 W=0.9	I x W=0.1	I y W=0.4	I y W=0.5	N 2 W=0.2	JF 3 W=0.3	I x W=0.7	I y W=0.8	JR 0 W=0.2

(c) Corresponding Linear Genome

Figure 1: An example of encoding a neural network using a linear genome

1. The sum of integer values is the same as the number of outputs of the neural controller encoded by the linear genome.
2. A sub-network (sub-linear genome) is a collection of genes starting from a neuron gene and ending at a gene where the sum of integer values assigned to the genes between and including the start neuron gene and the end gene is 1.

Figure 2 illustrates this concept. Please note that only the number of inputs to neural genes is variable, so in order to achieve a compact representation only this number is stored within the linear genome.

Other features of the linear genome, apart from its compactness, include completeness (any network can be encoded) and closedness (the mutation operators described below always produce valid networks.) It is also a very general structure that can be used both for direct and indirect encodings of neural networks, and for modular networks. These properties have been formally proven in [16].

3.3 EANT2's Search for Neural Networks

Figure 3 shows how EANT2 works. The different steps of the algorithm are explained in detail below.

3.3.1 Initialisation

EANT2 usually starts with minimal initial structures. A "minimal" network has no hidden layers or recurrent connections, only 1 neuron per output. Each neuron is connected to approx. 50 % of inputs; the exact percentage and selection of inputs are random. EANT2 gradually develops these simple initial network structures further using the structural and parametrical evolutionary algorithms discussed below. On a larger scale new neural structures are added to a current generation of networks. We call this "structural exploration". On a smaller scale the current individuals (structures) are optimised by changing their parameters: "structural exploitation".

3.3.2 Structural Exploitation

At this stage the structures in the current EANT2 population are exploited by optimising their parameters. Parametrical mutation in the original version, EANT, was implemented using *evolution strategies* [6]. In evolution strategies the so-called *strategy parameters* of the evolutionary algorithm, mainly the standard deviation for random mutation, were themselves adapted by an evolutionary algorithm. This has the advantage that the system needs even less knowledge of the problem than with a different evolutionary algorithm, like evolutionary programming. However, using evolution

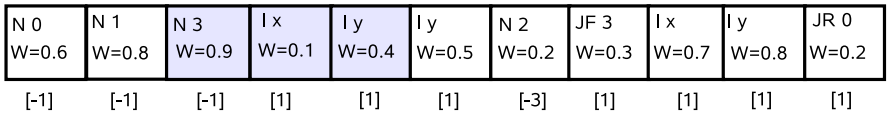


Figure 2: An example of the use of assigning integer values to the genes of the linear genome. The linear genome encodes the neural network shown in Figure 1(a). The numbers in the square brackets below the linear genome show the integer values assigned to the genes of the linear genome. Note that the sum of the integer values is 1 showing that the neural network encoded by the linear genome has only 1 output. The shaded genes form a sub-network. The sum of these values assigned to a sub-network is always 1.

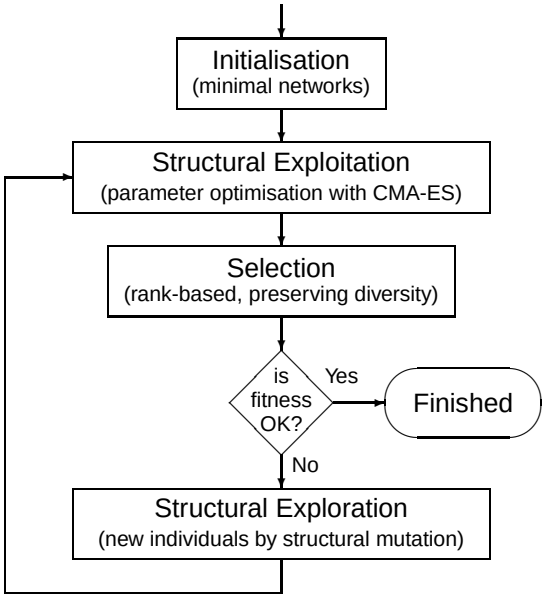


Figure 3: The EANT2 algorithm. Please note that CMA-ES has its own loop which creates a nested loop within EANT2.

strategies for parametrical mutation has the following disadvantages:

1. After a strategy parameter has been adapted it takes many applications of the mutation operator on the corresponding network parameter until the new value of the strategy parameter can be judged. Even then it is unclear when looking at the change in fitness value whether the network performs better/worse because of this adapted strategy parameter or because of other changes that happened during those many generations.

2. The number of strategy parameters adds to the number of total parameters in the system, increasing even further the dimensionality of the space in which ideal parameters are searched.

Disadvantage 1 can be ignored in settings where a very large population size is used. However, it does matter in the context of neural network development where large population sizes are prohibitive unless the problem is very simple.

For these reasons the newer version EANT2 uses CMA-ES, *Covariance Matrix Adaptation Evolution Strategy* [9] in its parameter optimisation. CMA-ES is a variant of evolution strategies that avoids random adaptation of the strategy parameters. Instead, the search area that is spanned by the mutation strategy parameters, expressed here by a covariance matrix, is adapted at each step depending on the parameter and fitness values of current population members. The covariance matrix is comparable to the Hesse matrix in traditional optimisation methods. However, it is estimated by CMA-ES without the use of an analytical derivative or finite differences that would require very many function evaluations. CMA-ES uses sophisticated methods to avoid things like premature convergence and is known for fast convergence to good solutions even with multi-modal and non-separable functions in high-dimensional spaces. *(ibid.)*

When the parameter optimisation with CMA-ES starts it is given for each variable an initial standard deviation used in its sampling of values in the search space. These standard deviations will be used as a starting point only; the search area is adapted by CMA-ES over time. These values are set by EANT2 depending on the current age of the corresponding gene. Parameters for newer structural elements are given a wider search area than older ones. This feature is based on the observation that over time parameters for existing structures tend to become more or less constant as they have been optimised several times. Structural changes at other places may also influence the optimal parameter values for the older structural elements, but usually at a relatively small scale. This is related to the “Cascade-Correlation Learning” paradigm presented by Fahlman and Lebiere [7].

3.3.3 Selection

The selection operator determines which population members are carried on from one generation to the next. Our selection in the outer, structural exploration loop is rank-based and "greedy", preferring individuals that have a larger fitness. If two structures have almost the same fitness the smaller individual is given a higher rank. A consequence of this is that existing structures may grow smaller if structural elements that do not help the performance are removed. In order to maintain diversity in the population, the selection operator also compares individuals by structure, ignoring their parameters. The operator makes sure that not more than 1 copy of an individual and not more than 2 similar individuals are kept in the population. "Similar" in this case means that a structure was derived from an another one by only changing connections, not adding neurons. Again, no network parameters are considered here.

3.3.4 Structural Exploration

In this step new structures are generated and added to the population. This is achieved by applying the following structural mutation operators to the existing structures: Adding a random subnetwork, adding or removing a random connection and adding a random bias. Removal of subnetworks (i.e. neurons together with all their connections) is not done as we found out that this almost never helps in the evolutionary process. The same is valid for a crossover operator, modelled after the one used in NEAT, which is currently not used. New hidden neurons are connected to approx. 50 % of inputs; the exact percentage and selection of inputs are random to enable stochastic search for new structures.

3.3.5 Differences to Other Methods

EANT2 is closely related to the methods described in the related work section above. One main difference is the *clear separation of structural exploration and structural exploitation*. By this we try to make sure a new structural element is tested ("exploited") as much as possible before a decision is made to discard it or keep it, or before other structural modifications are applied. Another main difference is the *use of CMA-ES in the parameter optimisation*. This should yield more optimal parameters more quickly, which is necessary when large networks are to be created. When EANT2's *structural mutation operator* adds a new neuron to a given structure, it also *connects the new neuron* to a random number of other neurons and/or inputs, and the new neuron's output as input to other neurons. Further differences of EANT2 to other recent methods, e.g. NEAT, are a *small number of user-defined algorithm parameters* (the method should be as general as possible), its *compact, linear encoding of the neural network* and the *explicit way of preserving diversity* in the population (unlike speciation in NEAT.)

Figure 4: Robot Arm with Camera and Object

4 The Visual Servoing Task

In order to study the behaviour of EANT2 and other algorithms on large problems we simulate the visual servoing setup shown in Figure 4. A robot is equipped with a camera at the end-effector and has to be steered towards an object of unknown pose. This is achieved in the visual feedback control loop depicted in Figure 5. In our system a neural network shall be used as the controller, determining where to move the robot on the basis of the object's visual appearance. Using the standard terminology by Weiss et al. [30] it is a "Static Image-based Look-and-Move" controller.

4.1 Definitions and Task Description

The object has 4 identifiable markings, see Fig. 4. Its appearance in the image is described by the *image feature vector* $y_n \in \mathbb{R}^8$ that contains the 4 pairs of image coordinates of these markings. The desired pose relative to the object is defined by the object's appearance in that pose by measuring the corresponding *desired image features* $y^\star \in \mathbb{R}^8$ ("teaching by showing".) Object and robot are then moved into a start pose so that the position of the object is unknown to the controller. The system has the task of moving the arm such that

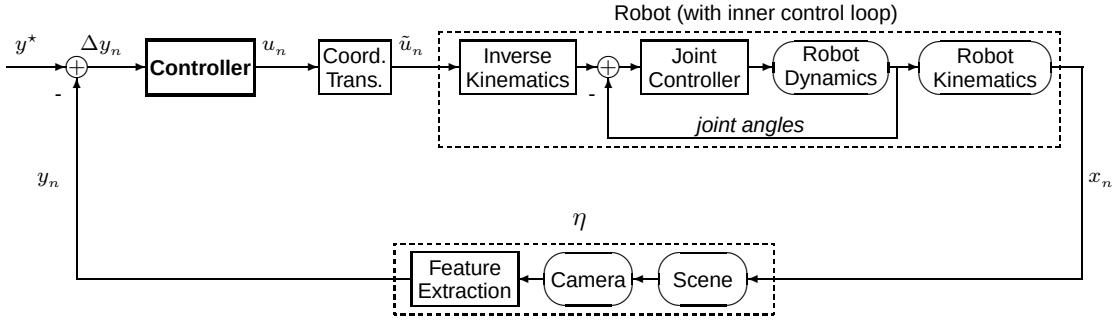


Figure 5: Visual Feedback Control Loop

the current image features resemble the desired image features. This is an iterative process.

The *input* to the controller is the *image error* $\Delta y_n := y^\star - y_n$ and additionally the 2 distances in the image of the diagonally opposing markings, resulting in a 10-dimensional input vector. The *output* of the controller/neural network is a relative movement of the robot in the camera coordinate system: $(\Delta x, \Delta y, \Delta z) \in \mathbb{R}^3$. This output is given as an input to the robot's internal controller which executes the movement. The new state x_{n+1} of the environment (i.e. the robot and scene) is perceived by the system with the camera. This is again used to calculate the next input to the controller, which closes the feedback loop shown in Figure 5.

In our case a neural network is developed as a controller by reinforcement learning as discussed in Section 2. For the assessment of the fitness (performance) of a network N it is tested by evaluating it in the simulated visual servoing setup. For this purpose 1023 different robot start poses and 29 teach poses (desired poses) have been generated. Each start pose is paired with a teach pose to form a task. These tasks contain all ranges and directions of movements. For each task, N is given the visual input data corresponding to the start and teach poses, and its output is executed by a simulated robot. The *fitness function* $F(N)$ measures the negative RMS (root mean square) of the remaining image errors after the robot movements, over all tasks. This means that our fitness function $F(N)$ always takes on negative values with $F(N) = 0$ being the optimal solution. Let y_i denote the new image features after executing one robot movement starting at start pose i. Then $F(N)$ is calculated as follows:

$$F(N) := -\sqrt{\frac{1}{1023}\sum_{i=1}^{1023}\left(\frac{1}{4}\sum_{j=1}^{4} d_j(y_i)^2 + b(y_i)\right)} \tag{1}$$

where

$$d_j(y_i) := \left\| (y^\star)_{2j-1,2j} - (y_i)_{2j-1,2j} \right\|_2 \tag{2}$$

is the distance of the jth marker position from its desired position in the image, and $(y)_{2j-1,2j}$ shall denote the vector comprising of the $2j-1$th and $2j$th component of a vector y. The inner sum of (1) thus sums up the squared deviations of the 4 marker positions in the image. $b(y)$ is a "badness" function that adds to the visual deviation an additional positive measure to punish potentially dangerous situations. If the robot moves such that features are not visible in the image or the object is touched by the robot, $b(y) > 0$, otherwise $b(y) = 0$. All image coordinates are in the camera image on the sensor and have therefore the unit 1 mm. The image sensor in this simulation measures $\frac{8}{3}$ mm $\times$ 2 mm. The average (RMS) image error is -0.85 mm at the start poses, which means that a network N that avoids all robot movements (e.g. a neural network with all weights $= 0$) has $F(N) = -0.85$. $F(N)$ can easily reach values below -0.85 for networks that tend to move the robot away rather than towards the target object.

An analysis of the data set used for training the network was carried out to determine its intrinsic dimensionality. The dimensionality is (approximately) 4, the Eigenvalues being 1.70, 0.71, 0.13, 0.04 and the other 6 Eigenvalues below 1e-15. It is not surprising that the dimensionality is less than 10, and this redundancy makes it more difficult to train the neural networks. However, we see this as a challenge rather than a disadvantage for our research, and the problem encoding is a standard one for visual servoing.

4.2 Related Work: Methods for Visual Servoing

Visual servoing is one of the most important robot vision tasks [12, 30]. Traditionally visual servoing controllers use a simple P-type controller—an approach

known from engineering [4]. In these controllers the output is determined as the minimal vector that solves the locally linearised equations describing the image error as a function of the robot movement. This output is often multiplied by a constant scale factor α, $0 < \alpha < 1$ (dampening.) Sometimes, more elaborate techniques like trust-region methods are also used to control the step size of the controller depending on its current performance [14].

From a mathematical point of view, visual servoing is the iterative minimisation of an error functional that describes differences of objects' visual appearances, by moving in the search space of robot poses. The traditional solution is equivalent to an iterative Gauss-Newton method [8] to minimise the image error, with a linear model ("Image Jacobian") of the objective function [12, 30].

There have also been learning approaches to visual servoing, using neural networks or combined neuro-fuzzy approaches like the one by Suh and Kim [10]. Urban et al. use a Kohonen self-organising map (SOM) to estimate the Image Jacobian for a semi-traditional visual servoing controller [29]. Zeller et al. also train a model that uses a Kohonen SOM, using a simulation, to learn to control the position of a pneumatic robot arm based on 2 exteroceptive and 3 proprioceptive sensor inputs [34].

Many of these methods reduce the complexity of the problem (e.g. they control the robot in as few as 2 degrees of freedom, DOFs) to avoid the problems of learning a complex neural network. Others use a partitioning of the workspace to learn a network of "local experts" that are easier to train [5, 11]. A neural network that controls a robot to move around obstacles is presented in [19]. The network is optimised by a genetic algorithm, however, its structure (topology) is pre-defined and does not evolve.

To our mind it is a shortcoming of most (if not, all) existing learning methods for visual servoing that the solution to the task is modelled by the designer of the software. Whether it be using again an Image Jacobian, or whether it be selecting the size and structure of the neural network "by hand"—that is, by intuition and/or trial and error—*these methods learn only part of the solution by themselves*. Training the neural network then becomes "only" a parameter estimation, even though the curse of dimensionality still makes this very difficult.

5 Experimental Comparison: EANT2 and NEAT

In order to validate learning methods we use the simulated visual servoing scenario as described in the previous section, with 1023 start poses and the definition of the fitness function F from equation (1) in Section 4.1 above. The 10 inputs and 3 outputs to the neural networks are also as above. The computationally expensive evaluation of F which needs 1023 network evaluations and simulated robot movements makes it a priority to develop networks with as few evaluations $F(N)$ as possible.

5.1 The NEAT System

NEAT, *NeuroEvolution of Augmenting Topologies*, by Stanley and Miikkulainen [26] has already been briefly introduced in Section 2. It uses one evolutionary optimisation loop in which structures and parameters of neural networks are mutated, and networks recombined using a crossover operator. The implementation of NEAT used here is the Java-based NEAT4J which is available as a SourceForge project[3]. For reference the original NEAT code by Stanley has also been analysed.

The initial population of NEAT4J consists of randomly generated networks without hidden layers that are either fully or sparsely connected (at an option.) In each generation the population is split into a number of species so that "compatible" individuals belong to the same species. The split is done using a compatibility measurement that incorporates network size, difference of weights and number of different genes. New species are created if necessary. If a species has a good average fitness, its size is increased, otherwise the size is decreased. Species become extinct if their size becomes zero or they excess a certain age. The best individual of each species is kept together with their offspring. New members of a species are spawned by crossover and mutation from their parents who are selected among the best individuals in this species. Mutation is done by a stochastic update of weights and structures. Nodes and connections are added with certain probabilities, but never removed. Existing connections can, however, be enabled or disabled by toggling a flag.

5.1.1 Search for Optimal NEAT4J Parameters

Unfortunately, there is no suggestion how NEAT's 13 evolution and 9 speciation parameters should be

[3] `http://neat4j.sourceforge.net/`

set. We have tried many settings and found out that the values from the examples of the original NEAT mixed with those of NEAT4J form a suitable starting point. The settings were then adapted to tune the system for our visual servoing task.

NEAT tends to enlarge networks if the *probability of toggling connections* on/off is low and slows down the growing of networks if it is high. After some test runs we decided to reduce the probability of toggling (PToggleLink 0.0001) so as to enable NEAT4J to sufficiently optimise the network weights before adding a lot of structure. For the same reason we also decreased the *probabilities for structural mutation* (PAddLink=0.0025, PAddNode=0.00125) after some test runs but left the *probabilities for weight changes* high (PMutation=0.25, PWeightReplaced=0.85.) NEAT reacts very strongly to *bias neurons* and tends to add many of them. However, in a few test runs this made the evolution process get stuck without improving the fitness. We therefore deactivated biases altogether (which makes sense, considering the visual servoing task.) An appropriate *population size* is hard to calculate but concerning the fitness increase over (wall clock) time a smaller population size usually works better than a bigger. Hence, we tested two sizes of populations, 30 and 150. In most cases the smaller population only performed slightly worse. We did not note a significant change in the test outcome when varying parameters for *speciation*.

5.2 The EANT2 System

The EANT2 system which was described in detail in section 3 was used with the following parameters:

- up to 30 individuals in the structural exploration (global population size)
- each individual spawns 2 children through structural mutation
- 2 parallel optimisations of the same individual by CMA-ES
- stop criteria for CMA-ES: maximum standard deviation in covariance matrix less than 0.00005 or iteration (CMA-ES generation) number over 500.

5.3 Results and Discussion

Figure 6 shows the development of the best individual's fitness value. Results from 5 experiments each of EANT2 and NEAT are shown, plotted against the generation number. EANT2's (outer) generation number is increasing much slower than NEAT's because of the inner loop that is contained within the structural exploitation with CMA-ES. The generation spans have therefore been roughly aligned by the number of evaluations of the fitness function, which is the determining factor for the wall clock time used to run the method.

5.3.1 Development of Fitness

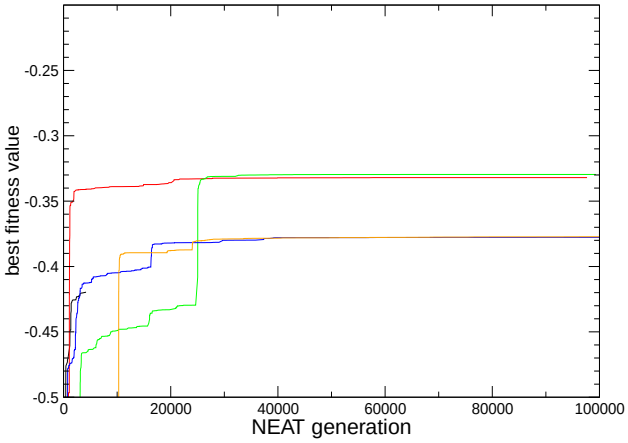
It can be seen that after around 25,000 generations the fitness values in NEAT reach -0.33 (better runs) and -0.38 (worse runs.) They do not improve significantly further until generation 100,000, at which point the experiments were stopped.

In EANT2, a considerable increase in fitness can be seen up to generation 15 (and further, as different experiments show.) After 5 generations the average best individual has a fitness of -0.25, which increases to -0.23 at generation 15.

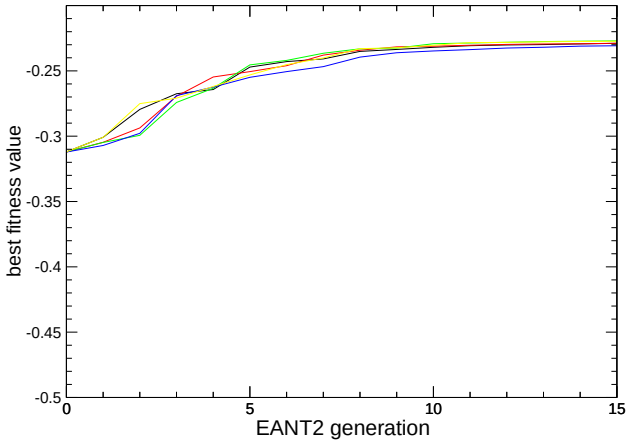
Let us recall that the fitness values are (modulo $b(\cdot)$) the remaining RMS errors in the image after the robot movement. Both methods quickly develop networks that reduce the image error from the initial -0.85 to as low as -0.23 with 1 robot movement. This is a very good result if one compares to the traditional Image Jacobian approach. Calculating the robot movement using the traditional approach (without dampening) yields a fitness of -0.61. In practice visual servoing techniques usually multiply the Image Jacobian step by a scalar dampening factor before executing it. However, this dampening of the optimisation step is independent of the nature of the model that was used to calculate it[4]. Since both the Image Jacobian and our networks calculate the necessary camera movement to minimise the image error in *one* step this is a meaningful comparison and shows that these networks can indeed be used for visual servoing.

This comparison with the standard approach shows that the networks from both NEAT and EANT2 are very competitive when used for visual servoing control. It can of course be expected that a non-linear model will be able to perform better than the linear Image Jacobian model. However, it should be taken into consideration that the Image Jacobian is an analytically derived solution (which is something we aim to avoid.) Also, and more importantly, the Image Jacobian contained the exact distance (z coordinate) of the object from the camera. While this is easy to provide in our simulator in practice it could only be estimated using the image features.

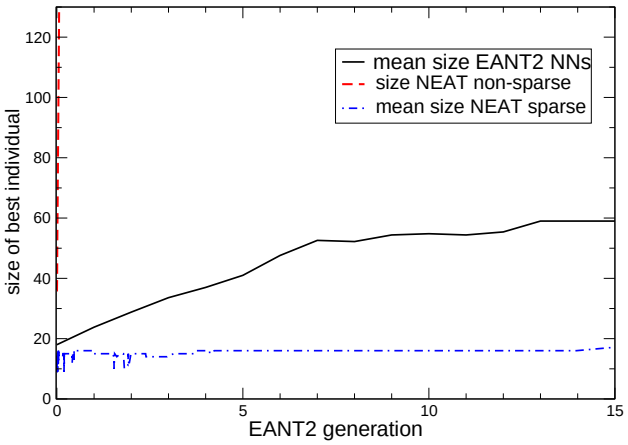
[4] It is nevertheless useful to make it dependent on the correctness of the model, as it is done in *restricted step methods* [8].



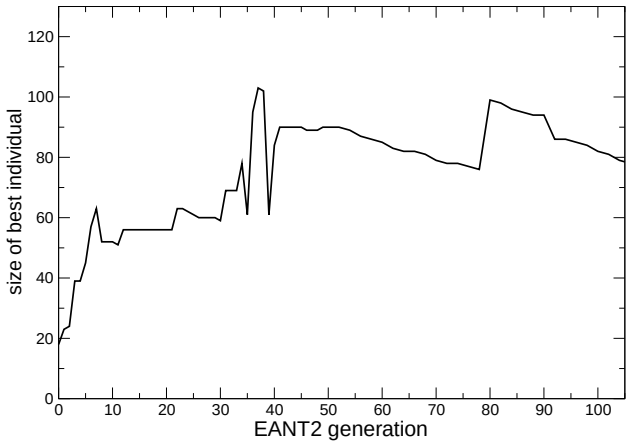
(a) NEAT: best fitness



(b) EANT2: best fitness



(a) Mean sizes, NEAT and EANT2



(b) Long time development, EANT2

Figure 7: Development of network size over time

5.3.2 Development of Network Sizes

Figure 7 shows the development of the neural network sizes over time. The graphs in figure 7(a) have again been aligned by evaluations of the fitness function (i.e. wall clock time.) An analysis of the network sizes shows that NEAT's resulting networks stay "sparse" if that initialisation option was used. The best performing network has 17 genes, with only 2 hidden neurons. Only 1 gene was added between generation 3,000 and 100,000, which explains why the fitness does not increase any further. However, without the "sparse" option NEAT generates networks with sizes approx. 80–140 already after 3,000 generations; their fitness is only around -0.89 to -0.66 and does not increase further with time or network size.

EANT2's networks are larger than the "sparse NEAT" networks, in part due to the different initialisation. The mean size at generation 5 is 41 (fitness -0.25.) Size increases slower as time goes on, with a mean size of 59 at generation 15 (fitness -0.23.) NEAT's mean final network size of 17 is reached by EANT2 at generation 0 (with no hidden neurons.) At this size the average fitnesses of the best individuals are -0.346 (NEAT) and -0.312 (EANT2).

As time goes on EANT2's structures continue to grow much further than NEAT's. Although NEAT does try to add new structure fairly often most of these structural elements are discarded. NEAT has a feature to keep newly created individuals even if they do not perform well in the first few generations of their existence but is seems that this feature does not help here.

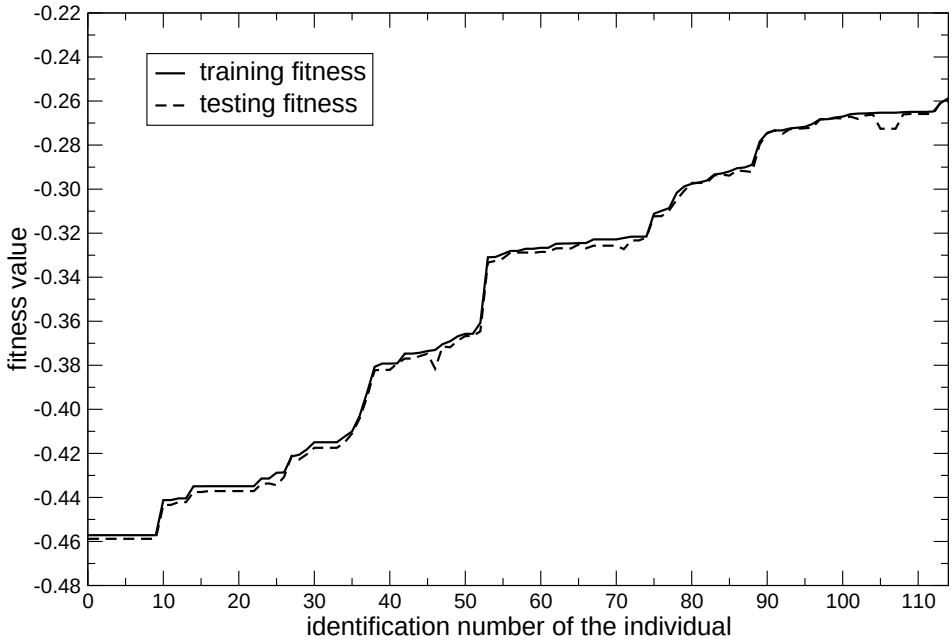


Figure 8: Comparison training vs. testing fitness values, EANT2. Plotted is the fitness of population members, which were sorted by training fitness

In order to see whether network sizes in EANT2 grow further, and to see whether our selection feature that is very slightly influenced by size helps to counteract bloat we ran one EANT2 trial for 106 generations. The resulting network size of the best individual in each generation is plotted in Figure 7(b). It can be seen that network sizes do not grow fast after generation 7 (size: 63.) The graph shows a few sudden changes that are the result of two individuals switching rank, and therefore being drawn for a generation or two. Apart from that one can see something like a sawtooth shape: Individuals have a neuron added that improves the fitness, then over a many generations they do not grow but slowly shrink. This happens when the mutation operator takes away connections that are not needed to maintain the current fitness value. One individual had the size 86 in generation 92 and has shrunk, without decrease (or increase) in fitness, to size 78 in generation 106. As Figure 6(b) already suggested, the overall fitness only improves very slowly after the first dozen or so generations: in this case from -0.2288 in generation 15 to -0.2256 in generation 106.

The two methods, NEAT and EANT2, differ in the way networks are generated, and NEAT performs worse in this scenario. Only when the networks are small and the probability of structural change is low compared to parametrical change can NEAT optimise networks well with its evolutionary algorithm. If some options influence NEAT to produce larger networks they have a significantly worse performance compared to EANT2 networks of the same size. This could mean that neural network parameters in NEAT are not optimised as well, or that structural elements exist that do not help the task well, or both.

Overall, EANT2 always created better networks than NEAT and required less parameter tuning to run successfully.

5.4 Training and Testing

In order to carry out a meaningful analysis of the neural networks trained by the EANT2 system we have generated a test set of 1023 visual servoing tasks. They are comparable with the 1023 tasks the system was trained on. In particular, the fitness value when not moving the robot is the same. However, the testing data require completely different robot movements. All 115 neural networks that were generated as intermediate results during one run of EANT2 were tested, without any change to them, on the testing data. Figure 8 shows a comparison of the resulting fitness values of these individuals, sorted by training fitness. It can be seen that the training and testing fitnesses are very similar indeed. The maximum deviation of testing fitnesses compared to training fitnesses is 2.738 %, the mean deviation 0.5527 % of the fitness value. From this follows that the neural networks developed with our technique did not just memorise the correct responses of the network but are capable of generalising to different, but compatible tasks.

6 Concluding Summary

In this article we have described EANT2, a method to develop both the structure and the parameters of neural networks by evolutionary reinforcement learning. EANT2 differs from other recent methods by implementing a clear separation of structural and parametrical development and the use of CMA-ES during parameter optimisation. It also features a compact linear genetic encoding of the neural network.

In order to validate EANT2, it was used with a complete simulation of a visual servoing scenario to learn neural networks by reinforcement learning. The same task was given to NEAT [26], a similar method. Results from the experiments show that both evolutionary methods can develop networks that make "useful" robot movements, decreasing the image error by moving towards the goal. The performance of both methods is also significantly better than the traditional visual servoing approach.

A comparison of both methods showed that the neural networks created by EANT2 always have a substantially better performance. NEAT also performs good when configured to keep network sizes very small, but then the development of networks comes to a halt, showing almost no improvement over a long runtime. For similar network sizes, EANT2's neural networks perform better.

When looking at the development of network sizes over time it can be seen that EANT2's neural networks do not grow very fast once a certain size is reached. When run over many generations those structural elements that are not needed to maintain the current best fitness value are discarded so that networks often gradually shrink over some time before new "useful" structural elements are added.

Our experimental results show that the our EANT2 method is capable of learning neural networks as solutions to complex and difficult problems. EANT2 can be used as a "black-box" tool to develop networks without being given much information about the nature of the problem. It also does not require a lot of parameter tuning to give useful results. The resulting networks show a very good performance.

Acknowledgements

The contribution of our colleague Yohannes Kassahun, most importantly the development of the original EANT algorithm within his PhD project in our research group, is gratefully acknowledged. He also provided Figures 1 and 2.

The authors also wish to thank Nikolaus Hansen, the developer of CMA-ES, and Kenneth Stanley, the developer of NEAT, for kindly providing source code which helped us to quickly start applying their methods.

References

[1] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65, 1994.

[2] R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, USA, 1961.

[3] A. Beltratti, S. Margarita, and P. Terna. *Neural Networks for Economic and Financial Modelling*. International Thomson Computer Press, London, UK, 1996.

[4] C. C. Bissell. *Control Engineering*. Number 15 in Tutorial Guides in Electronic Engineering. CRC Press, Boca Raton, USA, 2nd edition, 1996.

[5] W. Blase, J. Pauli, and J. Bruske. Vision-based manipulator navigation using mixtures of RBF neural networks. In *International Conference on Neural Network and Brain*, pages 531–534, Bejing, China, April 1998.

[6] Á. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, Berlin, Germany, 2003.

[7] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, Carnegie Mellon University, Pittsburgh, USA, August 1991.

[8] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, Chichester, 2nd edition, 1987.

[9] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[10] K. Hashimoto, editor. *Visual Servoing: Real-Time Control of Robot Manipulators Based on Visual Sensory Feedback*, volume 7 of *Series in Robotics and Automated Systems*. World Scientific Publishing Co., Singapore, 1994.

[11] G. Hermann, P. Wira, and J.-P. Urban. Neural networks organizations to learn complex robotic functions. In *Proceedings of the 11th European Symposium on Artificial Neural Networks (ESANN 2003)*, pages 33–38, Bruges, Belgium, April 2005.

[12] S. Hutchinson, G. Hager, and P. Corke. A tutorial on visual servo control. Tutorial notes, Yale University, New Haven, USA, May 1996.

[13] W. R. Hutchison and K. R. Stephens. The airline marketing tactician (AMT): A commercial application of adaptive networking. In *Proceedings of the 1st IEEE International Conference on Neural Networks, San Diego, USA*, volume 2, pages 753–756, 1987.

[14] M. Jägersand. Visual servoing using trust region methods and estimation of the full coupled visual-motor Jacobian. In *Proceedings of the IASTED Applications of Control and Robotics, Orlando, USA*, pages 105–108, January 1996.

[15] Y. Kassahun. *Towards a Unified Approach to Learning and Adaptation*. PhD thesis, Cognitive Systems Group, Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany, February 2006.

[16] Y. Kassahun, M. Edgington, J. H. Metzen, G. Sommer, and F. Kirchner. Common genetic encoding for both direct and indirect encodings of networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007), London, UK*, pages 1029–1036. ACM Press, 2007.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[18] T. M. Mitchell. *Machine Learning*. McGraw-Hill, London, UK, 1997.

[19] D. E. Moriarty and R. Miikkulainen. Evolving obstacle avoidance behavior in a robot arm. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, Cape Cod, USA, 1996.

[20] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 13:271–369, June 2004.

[21] A.-P. Refenes, editor. *Neural Networks in the Capital Markets*. John Wiley & Sons, New York, Chichester, USA, 1995.

[22] C. Robert, C.-D. Arreto, J. Azerad, and J.-F. Gaudy. Bibliometric overview of the utilization of artificial neural networks in medicine and biology. *Scientometrics*, 59(1):117–130, 2004.

[23] R. Rojas. *Neural Networks - A Systematic Introduction*. Springer Verlag, Berlin, Germany, 1996.

[24] N. T. Siebel and Y. Kassahun. Learning neural networks for visual servoing using evolutionary methods. In *Proceedings of the 6th International Conference on Hybrid Intelligent Systems (HIS'06), Auckland, New Zealand*, page 6 (4 pages), December 2006.

[25] J. C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, Hoboken, USA, 2003.

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

[27] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, USA, March 1998.

[28] R. R. Trippi and E. Turban, editors. *Neural Networks in Finance and Investing*. Probus Publishing Co., Chicago, USA, 1993.

[29] J.-P. Urban, J.-L. Buessler, and J. Gresser. Neural networks for visual servoing in robotics. Technical Report EEA-TROP-TR-97-05, Université de Haute-Alsace, Mulhouse-Colmar, France, November 1997.

[30] L. E. Weiss, A. C. Sanderson, and C. P. Neuman. Dynamic sensor-based control of robots with visual feedback. *IEEE Journal of Robotics and Automation*, 3(5):404–417, October 1987.

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

[32] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, September 1999.

[33] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, May 1997.

[34] M. Zeller, K. R. Wallace, and K. Schulten. Biological visuo-motor control of a pneumatic robot

arm. In C. H. Dagli, M. Akay, C. L. P. Chen, B. R. Fernandez, and J. Ghosh, editors, *Intelligent Engineering Systems Through Artificial Neural Networks. Proceedings of the Artificial Neural Networks in Engineering Conference, New York*, volume 5, pages 645–650. American Society of Mechanical Engineers, 1995.