January 01, 2012

BlueSANE: integrating functional blueprints with neuroevolution

Jessica H. Lowell
Northeastern University

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER AND INFORMATION SCIENCE
MS THESIS APPROVAL FORM

THESIS TITLE: Blue SANE : Integrating Functional Blueprints with Neuroevolution

AUTHOR:

MS Thesis is approved towards the MS Degree in Computer Science.

Thesis Advisor	*Date* 4/27/12
Thesis Reader	*Date* 4/27/2012
Thesis Reader	*Date*

GRADUATE SCHOOL APPROVAL:

Director, Graduate School	*Date* 4/30/2012

ORIGINAL COPY DEPOSITED IN LIBRARY:

Reference Librarian	*Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature	*Date* 4/30/2012

BlueSANE: Integrating Functional Blueprints with Neuroevolution

Jessica Lowell

Neuroevolution algorithms are an important tool for optimizing neural network design in the fields of control and machine learning. We seek to improve SANE, a classic machine learning algorithm, by optimizing the size of the hidden layer in the neural networks that it generates. We use a technique called functional blueprints that guide the self-organization of systems by specifying their desired behavior, in this case avoidance of over/underfitting. We performed experiments with a simulated double cart-pole balancing benchmark problem which indicate that BlueSANE improves performance by slight to moderate amounts compared to the original SANE algorithm.

I. Introduction

Artificial neural networks are based on the connections between functionally-related biological neurons that make up a nervous system. They are made up of interconnecting artificial neurons, which are software or hardware constructs that mimic the properties of biological neurons. There are several different models of neural networks, including Hodgkin-Huxley [7], Fitzhugh-Nagumo [5], and connectionist backpropagation [18]. In this paper, we are concerned with software neural networks that are computational models for non-linear data modeling, in which the entire neural network serves as a function approximator, with each neuron summing its inputs and passing them through an activation function [14].

Neuroevolution algorithms are useful for generating appropriate neural networks for machine learning and control problems in situations where it is infeasible or otherwise undesirable to design the relevant neural network by hand. They search the solution space for optimal networks directly, and allow for a relatively uncontstrained problem environment. All that they require is an effective quality metric for candidate solutions. There are many existing neuroevolution algorithms [4,6,9,10,12] that are easy to use because they work reasonably well, are well-studied, and have free publicly-available implementations. This thesis provides a means for improving one of these algorithms, using a method called BlueSANE, in which a new guided artificial

evolution technique known as functional blueprints [2] is used to extend the existing SANE algorithm.

1. Motivation

For some simple problems, an effective neural network can be designed by hand. One example of this is the classification of linearly separable data, which can be performed with a simple two-layer perceptron. More complicated and effective neural networks, and other standard machine learning solutions such as decision trees, have been designed for such problems as hand-written digit recognition and time-series analysis, where it is not hard to develop a set of correct input-output pairs for supervised learning. However, in some tasks, such as robot control, multiprocessor resource allocation, or bioreactor process control, it is difficult or impossible to develop this set of correct input-output pairs because of issues such as change in the environment over time or the environment being only partially observable.

These problems are prime candidates for neuroevolution algorithms, which require only the existence of a consistent fitness metric for a candidate classifier or controller. Some of the most widely available and best-studied advanced neuroevolution algorithms include SANE (Symbiotic Adaptive Neuro-Evolution), which evolves individual neurons in the hidden layer of a network [10], EANT (Evolutionary Acquisition of Neural Topologies), which evolves full neural networks with both direct and indirect encoding [7], ESP (Enforced Sub-Populations), an expansion of SANE which evolves subpopulations of neurons with which to compose networks [16], and NEAT (Neuro-Evolution of Augmenting Topologies), which evolves full networks with speciation [17]. Since these algorithms are well-known and widely in use, extensions that improve their speed and/or accuracy, would be in a strong position for adoption by the control and reinforcement learning communities.

This thesis also demonstrates a new application for functional blueprints [1,2], a relatively new technique for guiding systems to self-organize by specifying their desired behavior. The area of functional blueprints currently has little previous work, and this demonstration that they can be used for machine learning may attract more attention to this area.

2. Research Goals

The overall aim of this thesis is to improve the performance of SANE, an existing widely-used neuroevolution algorithm, by incorporating guided evolution in the form of functional blueprints, with the broader intention that this could also lead to improvements in other neuroevolution algorithms. Improvements to SANE and other neuroevolution algorithms could in turn lead to improvements in machine learning, classification, and automated control, in environments where neuroevolution is considered a particularly useful technique. These problems fall under a wide range of real-world domains, such as manufacturing, bioinformatics, game-playing, and aerospace.

As no one has previously used functional blueprints in neuroevolution or any other form of neural network design, another goal of this research is to demonstrate the feasibility of this use of functional blueprints. This opens up a new area of research and engineering.

3. Expected Contributions

The main contributions of this thesis are expected to be:

* A performance improvement to a classic and widely-studied neuroevolution algorithm.

* The pioneering of a new area of study and application for functional blueprints, as neuroevolution itself was a new area of study and application for genetic algorithms.

* A better understanding of functional blueprints and their potential uses.

* The adding of a new technique to the toolbox of evolutionary machine learning.

4. Approach

Integrating functional blueprints into neuroevolution algorithms requires consideration of several problems, including:

1. Determination what aspects of neural network design are suited to guidance by the behavior specification methodology of functional blueprints.
2. Review of neuroevolution algorithms, to decide which one(s) are most influenced by the aspects of design identified in problem (1).
3. Determination of how to quantify the neural network behaviors identified in problem (1) along a range that can be used by functional blueprints.
4. Integration of functional blueprints into the existing algorithm to create a new algorithm.
5. Evaluation of the new algorithm on a benchmark problem, including comparison of their performance with that of the algorithm which it extends.

This thesis addresses each of these problems. First, we discuss different aspects of neural network design, and how neural network design maps to behavior. Then, we review several existing algorithms and explain why SANE was selected. Finally, we discuss the design and implementation of the new algorithm, and its results on a benchmark problem.

II. Related Work

In this section we review the literature on the concepts behind BlueSANE, including neuroevolution, SANE, other SANE improvement methods, and functional blueprints.

1. Neuroevolution

Neuroevolution is an evolutionary learning method that combines genetic algorithms and neural networks. Neural networks are the phenotype of the genetic algorithm, and neural network parameters such as topology or connection weights are the genotype [17]. The algorithm searches for the neural network that is optimal for some particular problem. Essentially, to use a biological

metaphor, the algorithm is evolving the best brain for the problem at hand.

Neuroevolution algorithms are particularly useful for problem classes in which the right neural network would be useful for solving the problem, but it is difficult or impossible to define the correctly-labeled input data needed for supervised learning. They are usually classified as a type of reinforcement learning. They have been successfully used in such applications as music composition [3], playing Go [13], the control of a robot arm [4], and the control of agents in mobile ad-hoc networks [9].

2. SANE

One neuroevolution algorithm, created to evolve neural networks for reinforcement learning quickly in systems with sparse reinforcement, is Symbiotic Adaptive Neuro-Evolution, or SANE [10]. Rather than evolving an entire neural network, SANE starts with fixed input and output layers and evolves the neurons in the hidden layer. It uses symbiotic evolution [10], where each population member (individual neuron) is a partial solution. The algorithm attempts to evolve partial solutions that can be combined with each other to form a strong full solution, and that need each other's "specializations" in order to be effective. Thus the partial solutions have a mutually beneficial symbiotic relationship with other partial solutions, which ensures the diversity of the population and prevents one type of specialization from dominating [10].

When SANE was designed it was faster on benchmark problems than existing state-of-the-art reinforcement learning methods such as GENITOR, Adaptive Heuristic Critic [10], and Q-learning [10]. SANE remains a popular and useful neuroevolution method. It has been shown to perform comparably to more recently-developed neuroevolution methods, such as ESP, NEAT, and EEC, at the game of tic-tac-toe, and outperformed NEAT and half-evaluated EEC at the game of Gobang [16].

3. Improvements to SANE

There have been several attempts to extend SANE and improve its speed or performance.

EuSANE, or Eugenic SANE, attempts to optimize the selection of hidden layer neurons through applying eugenic methods to the generation of new neurons for SANE's neuron population. EuSANE was shown to be several times faster than SANE on the double-pole-balancing benchmark problem [12]. Hierarchical SANE [11] attempts to overcome SANE's tendency to stall in large problems by integrating an exploratory neuron-level search with an exploitive network-level search, and outperformed SANE in the control of a robot arm. ESP [16] is able to converge on solutions more quickly than SANE by evolving subpopulations of neurons and selecting neurons from each subpopulation, rather than simply evolving individual neurons, and allows for recurrent networks.

4. Functional Blueprints

Functional blueprints [2] are a technique in which a design specifies desired behaviors and a means for adapting the system to achieve those behaviors when they are not met. A functional blueprint consists of:

1. A system behavior that does not transition sharply from viability to non-viability.
2. A stress metric that specifies the degree to which a behavior is not being achieved.
3. An incremental program that modifies the structure of the system to relieve stress.
4. A program to construct an initial viable system.

Beal showed that functional blueprints could be used to direct capillary-style growths in desired directions and concentration from a "seed" [2]. Networks of functional blueprints have been used in a preliminary architecture for the adjustment of robot design, and shown to converge on desired behaviors rapidly and reliably [1].

III. Research Questions

Here we give the research hypotheses and explain some of the problems and questions resulting from them.

1. Hypotheses

Very little previous work has been done on functional blueprints. However, functional blueprints are a guided evolutionary computing technique, and extensive work in the field of neuroevolution has shown that a variety of evolutionary computing methods can be used in neural network development [10,15,16,17]. This brings us to our first research hypothesis.

Hypothesis 1: It is possible to use functional blueprints to guide the evolution of neural networks.

The basic SANE algorithm [10] keeps the number of nodes fixed in each layer of the evolving network. The appropriate numbers of nodes in the input and output layers of the neural network are determined by the number of problem inputs and outputs, but design efforts on the part of a human or algorithm are needed to determine the number of nodes in the hidden layer. A hidden layer with too many nodes leads to overfitting, which makes the neural network unable to perform properly on the full range of data that it could be expected to process, and a hidden layer with too few nodes leads to underfitting, which leads to generally poor performance. From these facts, we arrive at our second research hypothesis.

Hypothesis 2: Lack of optimization of the number of nodes in the hidden layer of neural networks is detrimental to the performance of the SANE algorithm.

The basic SANE algorithm does not optimize the number of nodes in the hidden layer of the neural networks that it produces. The BlueSANE extension to the SANE algorithm attempts this optimization. Following from Hypothesis 2, we get our final research hypothesis.

Hypothesis 3: An extension of SANE with functional blueprints integrated to optimize the number of nodes in the hidden layer will outperform the basic SANE algorithm.

Following from this last hypothesis is the question of what we mean by "outperform". This will be discussed below in Section III-2D.

2. Problems and Outstanding Questions

In this section, we discuss the problems implied by the hypotheses of section III-1.

A. Designing Functional Blueprints for a Neural Network

Previous work with functional blueprints has focused on applications to robot design. There are few existing implementations of functional blueprints, and no guides to their implementation, as they are a new technique. In order to use them for neural network design, we devised a spectrum along which the desired behavior can fall, and determined how structural features of the neural network can be manipulated to move the network's behavior along this spectrum. In this case, the desired behavior is the avoidance of underfitting or overfitting, and the structural correlate to this is the number of neurons in the network's hidden layer. Therefore, we decided that the "stress" aspect of the functional blueprint should be linked to degree of underfitting or overfitting, and that the blueprint should relieve this stress by increasing (for the underfitting case) or decreasing (for the overfitting case) the number of neurons in the hidden layer.

B. Integrating Functional Blueprints with SANE

There are numerous existing implementations of SANE in a variety of programming languages. Once we designed a functional blueprint for use with SANE, we had to integrate it. Since the number of neurons in the hidden layer of the network is fixed in basic SANE, we had to modify the code for the original algorithm to allow for varying numbers of hidden-layer neurons.

C. Testing BlueSANE

In order to test BlueSANE's performance and compare it to that of SANE, we needed one or more benchmark problems on which to test it. We decided on the double pole-balancing problem, which is commonly used to test neuroevolution algorithms, because it is such a commonly-used test problem in this domain and it was easy to find an implementation of it.

D. Performance Metrics

There are multiple ways in which we could measure the performance of the SANE and BlueSANE algorithms. One of these is the speed with which the algorithms converge on solution neural networks. Speed is an important factor in the feasibility of using neuroevolution algorithms in real-world control and learning problems. Another is the quality of the generated solution networks.

Speed itself can be broken down into multiple metrics. One of these is the actual time that it takes for the algorithms to generate solutions. Another is the number of generations of candidate neural networks that the algorithm must produce before converging on a solution.

While actual time is important for feasibility, it can be affected by factors outside of the algorithm itself, such as the type of hardware being used, the programming language being used, the programmer's own implementation, or the use of software libraries that allow for disk-based computations or otherwise change how the computer uses its resources. Therefore, our main concern about the actual time required by BlueSANE vs SANE is that BlueSANE not be drastically slower. Because the implementation of SANE that we used took in a fixed number of generations as a parameter, and since the runtimes of SANE and BlueSANE were not significantly different, we chose to use the best fitness (quality) of each algorithm's final-generation solution network as our performance metric.

IV. Algorithm Descriptions

In this section we provide pseudocode for the SANE and BlueSANE algorithms. In Fig. 1, we show pseudocode for a generation of the original SANE algorithm (the algorithm loops through 200 generations).

---

1. Reset fitness ratings for each neuron to 0.
2. Randomly select a subset of the neural population containing a specified number of

neurons.

3. Perform the domain-specific task using the neurons in the selected subset as the hidden layer. Network fitness = performance of network using this subset.
4. Based on candidate network fitness, rate the fitness of each neuron in the subset.
5. Repeat 2-4 until all neurons have been selected a minimum number of times.
6. Sort the neuron population by the cumulative fitness of neurons.
7. Delete the lowest-fitness quarter of neurons.
8. Breed the highest-fitness quarter of neurons with each other and use offspring to replace the deleted quarter.

Figure 1: Pseudocode for one generation of SANE.

In order to convert SANE into BlueSANE, we add the concept of functional blueprints. This is seen in steps 3-4, 6, and 9-10 in Fig. 2 below. In the first generation, the stress stored in the functional blueprint is zero, so steps 3-4 are not triggered. In addition to the candidate network fitness evaluation in SANE, BlueSANE also evaluates the candidate network's fitness on the evaluation problem with varied parameters (the “new situation” in step 6) and stores the difference between the two fitnesses. The purpose of this step is to help determine whether the networks are being overfitted to the version of the problem with the “regular” parameters. In our experiments, the varied parameters were the lengths of the two test poles.

Once the generation's candidate networks have been evaluated on both versions of the problem, BlueSANE checks for underfitting and overfitting. If the best network in the current generation is less fit than the best network in the previous generation, the algorithm detects underfitting, and sets the stress stored in the functional blueprint to the difference between this and the previous generations' best fitness (which will be a negative number). If the difference between the “regular” and “new situation” fitnesses has increased twice in a row, the algorithm determines that networks are being overfitted to the specific parameters of the test problem – in this case, the specific lengths of the test poles in the “regular” network evaluation function – and sets the stress stored in the functional blueprint to the average of the two increases. This takes place in steps 9-10 below. When the next generational iteration starts, the stress level may then trigger steps 3-4, causing a neuron to be added or removed from the hidden layer of the new generation's candidate

networks.

1. Reset fitness ratings for each neuron to 0.
2. Randomly select a subset of the neural population containing a specified number of neurons.
3. If the functional blueprint stress if above a specified threshold, and the number of neurons in the population is greater than 3, delete a neuron from the population
4. If the functional blueprint stress is below a specified threshold, add a neuron to the population.
5. Perform the domain-specific task using the neurons in the selected subset as the hidden layer. Network fitness = performance of network using this subset.
6. Perform the domain-specific task using the neurons in the selected subset as the hidden layer and with varied test conditions. "New situation" network fitness = performance of network using this subset.
7. Based on candidate network fitness, rate the fitness each neuron in the subset.
8. Repeat 2-7 until all neurons have been selected a minimum number of times.
9. If network fitness for best candidate of this generation is less than network fitness for best candidate of previous generation, stress = fitness(this generation) – fitness(previous generation).
10. If the difference between the network fitness and new situation network fitness for the best candidate of the generation has increased twice in a row, stress = average increase in difference over those two occasions.
11. Sort the neuron population by the cumulative fitness of neurons.
12. Delete the lowest-fitness quarter of neurons.
13. Breed the highest-fitness quarter of neurons with each other and use the offspring to replace the deleted quarter.

Figure 2: Pseudocode for one generation of BlueSANE.

V. Implementation Details

This section describes details of the how the algorithm was implemented in source code. It also

describes how to run the code.

1. Algorithm Implementation

The BlueSANE code is written in Java and is extended from the freely-downloadable JavaSANE 1.2 project [19] on the University of Texas at Austin's Neural Network Research Group (NNRG) website. It consists of a package for the algorithm itself, and a package for the double cart-pole balancing implementation. The latter is adapted from the UT-Austin Neural Network Research Group's implementation, available with their ESP neuroevolution algorithm source code [20]. The relationships between the major classes of the code are depicted in Fig. 3.

A variety of configuration parameters, such as the starting number of hidden-layer neurons and the number of candidate networks in a generation, are defined in a configuration class, to which the algorithm code refers throughout. The structure definitions for neurons and networks are each contained in a class, with a separate class to store information about the best network so far on any given run. At the heart of the algorithm, there is a class to build and activate populations of neurons, and another class containing evolutionary methods. Finally, there is a class for the functional blueprint. The algorithm is connected to the double cart-pole balancing test problem through a domain class which defines how the fitness of a candidate neural population is evaluated. When this evaluation function is invoked, it calls the double pole-balancing experiment on the candidate population.

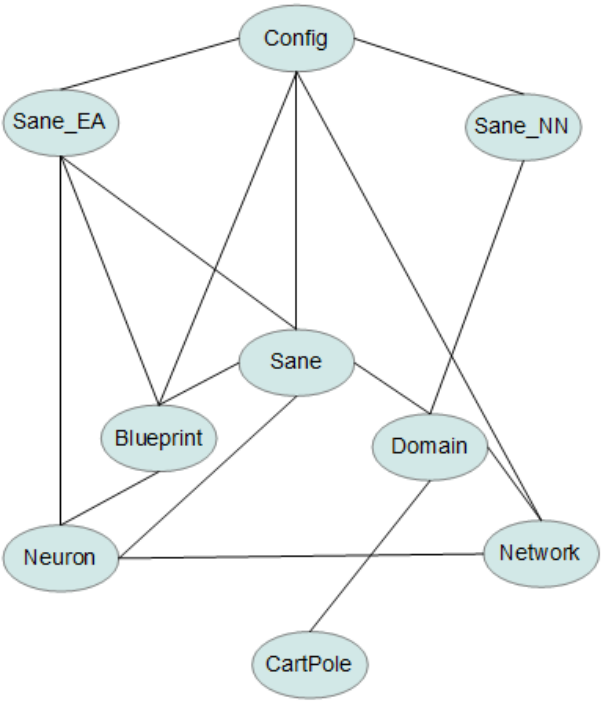

Figure 3: Block diagram of major code classes

2. Running the Code

In the pole-balancing experiment, the pole lengths and starting angles of tilt are constants. If a pole's angle of tilt reaches at least 36 degrees, the pole "falls" and the experiment ends. Each recalculation of the states of the two poles is a "step" in the experiment. The fitness of a candidate network is determined by the number of steps taken by the program before a pole falls.

The program is called from the command line, with parameters for a neural population file, a number of generations, a seed for the random number generator, and an error file. The neural population file - which contains a starting population of neurons for candidate networks - was not used in the experiments, which used random starting populations of neurons. The number of generations, set to 200 for the experiments, is the number of times that the neural network "breeding" will iterate.

VI. Methodology

This section describes the benchmark problem that we use to test BlueSANE, and the methods for conducting experiments.

1. The Double Cart-Pole Balancing Problem

Double cart-pole balancing (Fig. 4), an inverted pendulum problem, is a common benchmark for neuroevolution and other control theory and reinforcement learning algorithms. Notable examples in neuroevolution include Polani and Mikkulainen's use of it to test their EuSANE extension to SANE [12], and Stanley's use of it to test the ESP extension to SANE [16]. In double cart-pole balancing, the neural network or other agent has to control a cart within a confined region. There are two poles of different lengths balanced

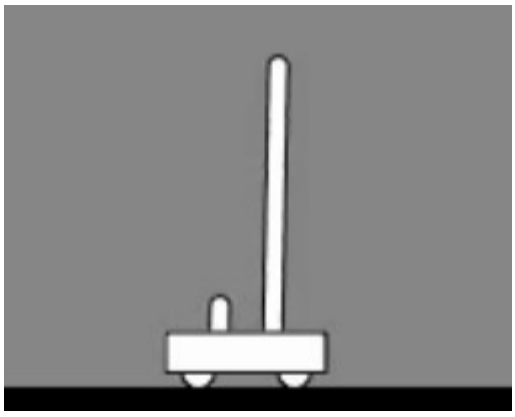

Figure 4: Depiction of double cart-pole balancing

on their ends and hinged to the cart, such that they are free to fall in the plane. The purpose of the controller is to apply horizontal forces to the cart to move it back and forth such that the poles do not fall. Once a pole falls or the cart leaves its confined region the controller receives reinforcement for learning purposes.

System parameters of a pole-balancing system include the mass of the cart, the distance that it is allowed to move from its starting point in either direction, the pole tilt angle at which the system receives a failure signal and inertial properties, the lengths of the poles. Cart-pole problems are strongly non-linear and unstable, which makes them useful for testing nonlinear control systems.

In a neural network controller, the inputs to the network for each time step include the position and velocity of the cart, the tilt angles of each pole, and the angular velocity of each pole. At each time step the controller must resolve which way to push the cart. Learning is represented through changes in the strengths of its connections between neurons.

In Table 1, we show parameters used in the simulation of a double cart-pole balancing environment in our experiments.

Table 1: Double Cart-Pole Environment Simulation Parameters

Gravity	-9.8
Pole Mass 1	0.1
Pole Mass 2	0.01
Cart Mass	1.0
Force on Cart	10.0
Seconds Between State Updates	0.01

2. Experimental Methods

In our first experiment, we ran SANE and BlueSANE ten times each. The number of hidden-layer neurons being evolved in SANE, and the starting number of hidden neurons being evolved in BlueSANE, were both set to 12. The pole lengths were set to 0.07 meters and 0.05 meters in

both SANE and BlueSANE. In BlueSANE, the "new situation" pole lengths, used to detect overfitting as explained above, were set to 0.1 meters and 0.04 meters. The number of generations was set to 200.

In our second experiment, we again ran SANE and BlueSANE ten times each. The staring number of hidden neurons in BlueSANE was the same as in the first experiment, as was the number of generations. However, the "new situation" pole lengths in BlueSANE were set to 0.06 meters each.

VII. Results and Interpretation

In this section, the results of our experiments are presented.

1. Experiment 1

The results of the first experiment, described in the previous section, were promising for BlueSANE. In this experiment, the mean fitness in the final generation of the original SANE algorithm was 33.9. BlueSANE outperformed SANE by 37.8%, achieving a mean final-generation fitness of 46.7 (Fig. 5). This result suggests that the functional blueprint methodology of BlueSANE had a positive effect on the fitness of the candidate solutions.

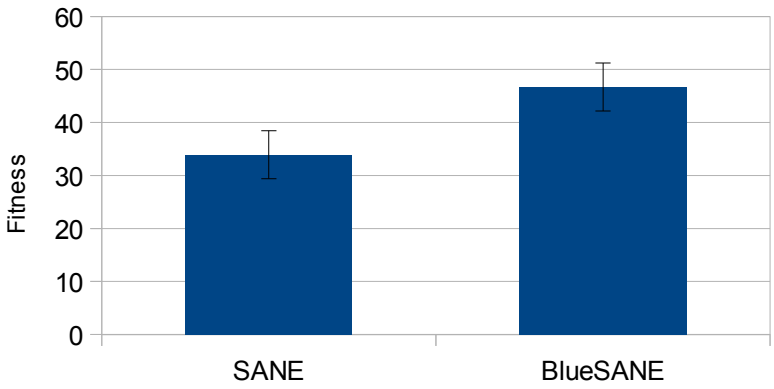

Figure 5: Mean fitness of SANE and BlueSANE for Experiment 1

The maximum fitness achieved by the final generation of the original SANE algorithm was 57 and the maximum achieved by the final generation of BlueSANE was 67, an increase of 17.5% (Fig. 6). This result also suggests that the functional blueprint methodology of BlueSANE had a positive effect on solution fitness; the effect was less

strong when looking at the best trials of each algorithm, however, it was still present. This trend was also noticeable when comparing the next-highest-ranked trial results for each algorithm – the second-best SANE trial result was 53 to the second-best BlueSANE trial's result of 58 (a 9% increase for BlueSANE). However, the third-best results for each algorithm were 38 and 50, respectively (a 31.6% increase for BlueSANE), and the worst trial results for each algorithm were 23 and 40 respectively (a 73.9% increase for BlueSANE). In other words, the "baseline" level of solution fitness produced by BlueSANE was much greater. Both algorithms were sometimes able to find solutions well above this baseline, and BlueSANE was able to find fitter solutions than SANE, but the difference in fitness was more subtle. For another view of these results, see Fig. 7, which depicts the minimum, mean, and maximum fitnesses for SANE and BlueSANE.

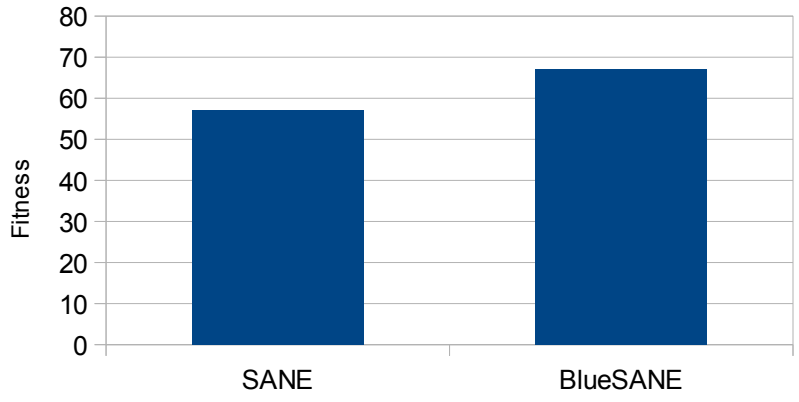

Figure 6: Maximum fitness of SANE and BlueSANE for Experiment 1

2. Experiment 2

The second experiment, also described in the previous section, showed promising results for BlueSANE. In this experiment, we looked at the same SANE trial data as for the previous experiment, in which the mean fitness in the final generation of the original

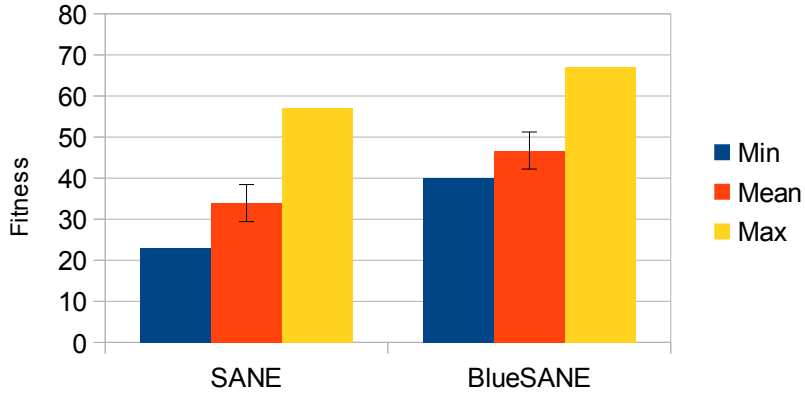

Figure 7: Minimum, mean, and maximum fitnesses of SANE and BlueSANE for Experiment 1

SANE algorithm was 33.9. This experiment's set of ten BlueSANE trials, using a different pair of pole lengths for the overfitting "check", outperformed SANE by 13.6%, achieving a mean final-generation fitness of 38.5 (Fig. 8). This is a smaller difference than in the first experiment, but it suggests that the functional blueprint methodology of BlueSANE had a positive effect on the fitness of the candidate solutions. It also raises the possibility that the pole lengths used in the overfitting check could be a parameter that affects BlueSANE algorithm performance by increasing or decreasing the likelihood of overfitting being detected by the algorithm. Note that this is not the only possible explanation. The difference in effect strength may have been a statistical quirk. Or some other as yet unknown aspect of the pole lengths may have influenced the algorithm's performance. Evaluation of these possibilities will require further study of the algorithm.

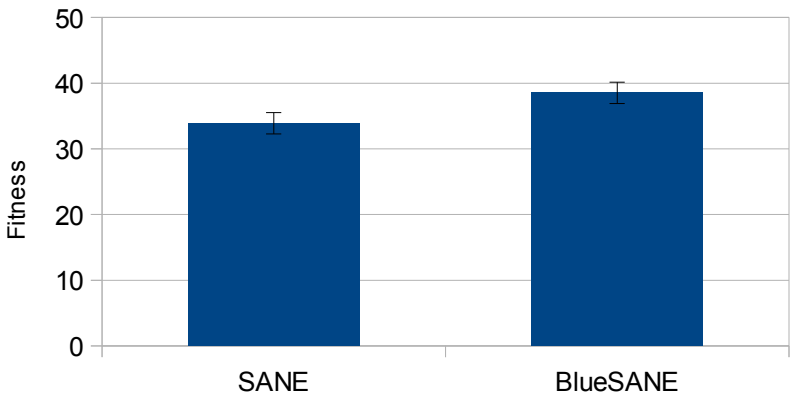

Figure 8: Mean fitness for SANE and BlueSANE in Experiment 2

In this experiment, as in the previous one, the maximum fitness achieved by the final generation of the original SANE algorithm was 57 and the maximum achieved by the final generation of BlueSANE was 67, an increase of 17.5% (Fig. 9). However, in this version, the effect in which BlueSANE's performance was stronger relative to SANE's at the lower fitness levels did not exist. The

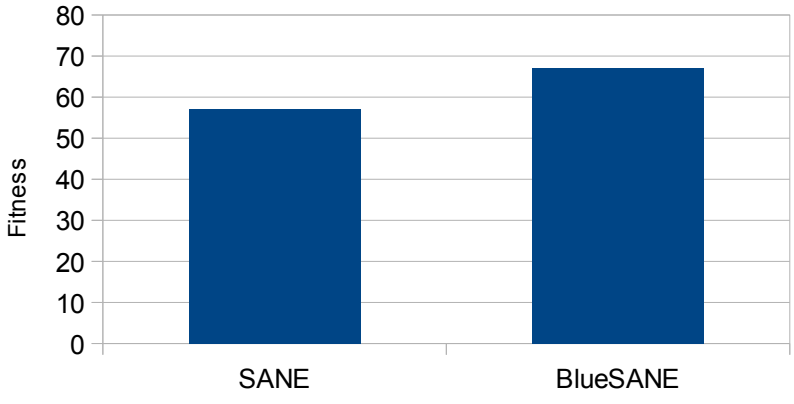

Figure 9: Maximum fitness for SANE and BlueSANE in Experiment 2

difference in fitness produced by the two algorithms in each of their lowest-fitness trials was only 3 (fitnesses of 23 and 26 for SANE and BlueSANE respectively), and differences for middle-performing trials ranged from -3 (the fifth-best-performing SANE trial outperformed the fifth-best-performing BlueSANE trial) to 13. The reason for these different shapes of results is a potential topic of future investigation. Again, we show minimum, mean, and maximum results side by side in Fig. 10.

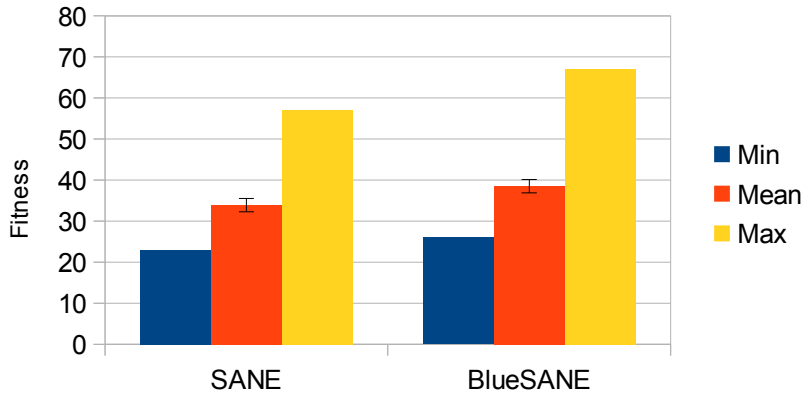

Figure 10: Min, Mean, and Max fitness of SANE and BlueSANE for Experiment 2

In this experiment, unlike in the previous one, we collected information about when BlueSANE added or subtracted hidden neurons, and how many it added or subtracted. We found wide variation, with the number of hidden neurons in the candidate networks of the final generation ranging from 13 to 28 (given a starting number of 12 hidden neurons), depending on the trial. Fig. 11 depicts the mean number of hidden neurons in the candidate networks produced by BlueSANE, by generation, showing a fairly steady rise over the increase in generation number, slightly in early generations, and then leveling off and even declining slightly at the end. The steeper rise in early generations makes intuitive sense, as the candidate networks of early generations have not yet evolved to be optimized for the problem. The leveling off and slight decline in very late generations also makes intuitive sense, as by this time the more-evolved candidate solutions should be better-tailored to the problem and demonstrating strong fitnesses, not only preventing underfitting but potentially causing overfitting.

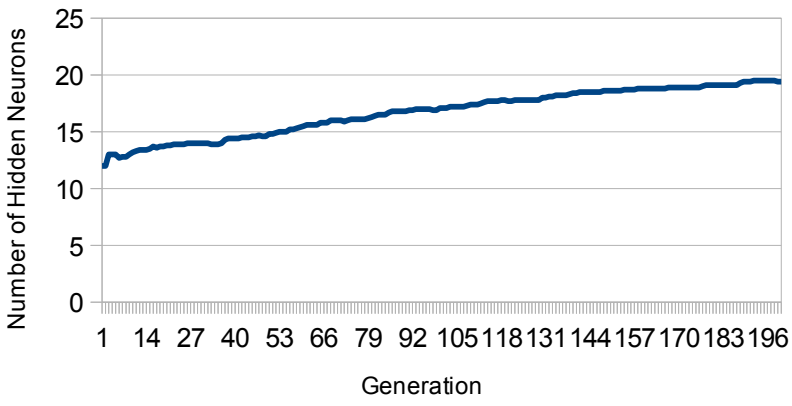

Figure 11: Mean number of hidden neurons across BlueSANE trials by generation, Experiment 2

Fig. 12 depicts the mean number of hidden neurons in ten BlueSANE trials over 200 generations just as the previous figure does, but adds data for the minimum and maximum number of hidden neurons in candidate networks during a BlueSANE trial, so that we may better examine the variation between trials. All three numbers are similar until between generations 40 and 50, at which point they diverge significantly. The maximum number of hidden neurons rises strongly until roughly the 100th generation, when it begins to level off again. The mean number exhibits the behavior described in the previous paragraph. The minimum number drops slightly and then stays constant through the rest of the generations.

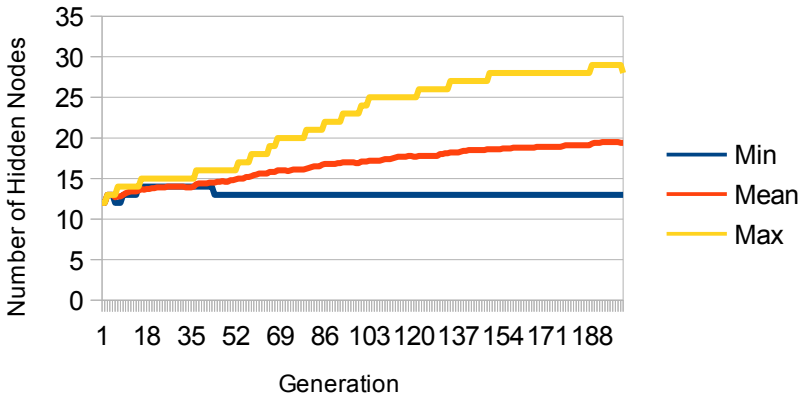

Figure 12: Maximum, mean, and minimum numbers of hidden neurons in BlueSANE trials by generation, Experiment 2

There are different possible interpretations of this data, and a full investigation of them is outside the scope of this document. One interpretation is that some runs of BlueSANE, through discovering a greater diversity of network topologies by random chance, happen to be particularly volatile between generations, which causes underfitting to be detected frequently (since it is detected when top fitness drops between two generations). It is worth noting that a larger number of neurons in the hidden layer may itself increase this volatility, by creating opportunities for a much greater number of possible combinations of

connections, and thus potentially greater volatility. This would create a feedback loop in which a BlueSANE trial that was adding hidden neurons relatively often early on, would then have a tendency to add more with increasing speed until the number of neurons was so high that it stopped making a significant difference in fitness, or even increased the chance of overfitting detection and brought the number back down. This hypothesis would account for the shape of the "maximum" trial in Fig. 12, but we cannot say anything definitive about it one way or the other. A final point is that both the most and least "volatile" trials did well in discovering candidate networks with high fitness – the former had a final score of 67 and the latter a final score of 51 – but while the fitness score of the former increased (and sometimes decreased) in both large and small jumps across the trial, that of the latter showed no movement at all until the fitness nearly doubled midway through the trial, and then stayed constant for the rest of it.

BlueSANE both added and subtracted neurons over the course of the algorithm run in all trials, but in all trials, it added more than it subtracted. This could be caused by actual considerations of underfitting and overfitting in this context – perhaps 12 hidden neurons is genuinely too small a number to maximize a neural network's performance in this domain. On the other hand, as the algorithm required the conditions for overfitting to be met for two consecutive transitions between generations for detection of overfitting, but only required conditions for underfitting to happen once before it was detected, it may be that it is "easier" for it to detect underfitting than overfitting. The reasoning when we programmed this disparity was that a fitness drop between generations (the condition that triggers underfitting detection) is based purely on how the generation of candidate networks is doing on the test problem, but the condition for detecting overfitting requires comparison between performances on the test problem and its "new situation" variant, and a one-time performance gap between the two could be random. We decided to require a trend in the latter case because we were concerned that without it, the algorithm would detect overfitting because of random chance.

VIII. Discussion

In this section, we look back at our research hypotheses and discuss the implications and wider applicability of our results.

1. Reexamining the Research Hypotheses

In Section III-1, we described three research hypotheses for this work. Here, we revisit these hypotheses to look at how they compare to our results.

Hypothesis 1: It is possible to use functional blueprints to guide the evolution of neural networks.

The evidence indicates that this hypothesis is correct. We implemented BlueSANE, in which functional blueprints were used to guide the evolution of neural networks. The candidate neural networks were able to evolve during trials of BlueSANE. The results show that neurons were being added to and removed from the hidden layers of candidate solutions as prescribed by the functional blueprints. The results also show that the networks' evolution was even more effective, on average, than in trials of SANE.

Hypothesis 2: Lack of optimization of the number of nodes in the hidden layer of neural networks is detrimental to the performance of the SANE algorithm.

The evidence is suggestive regarding this hypothesis. BlueSANE, the algorithm which attempted to optimize the number of nodes in the hidden layer performed better, on average, than the unmodified algorithm which was otherwise identical. This indicates that lack of freedom to change the number of hidden nodes was detrimental to SANE's performance.

Hypothesis 3: An extension of SANE with functional blueprints integrated to optimize the number of nodes in the hidden layer will outperform the basic SANE algorithm.

As has already been stated, BlueSANE, an extension of SANE with functional blueprints integrated to optimize the number of hidden layer nodes, performed better on average than SANE in both experiments. The hypothesis is supported.

2. Algorithm Applicability in Neuroevolution

The use of functional blueprints to improve neuroevolution algorithms by adding to or subtracting from the hidden layer could conceivably be applied to any neuroevolution algorithm that evolves a neural network's hidden layer. One example is ESP, an extension of SANE which allows neurons to evolve recurrent connections, which in turn enables them to use memory to make decisions. Another example is EuSANE, a eugenic extension of SANE that constructs individual networks to maximize their fitness based on historical data and correlations between genes and performance, rather than using random mutation and recombination.

The applicability to these SANE-based algorithms is straightforward because they all evolve the hidden neural layer of candidate networks. The "Blue" approach described here is tailored to evolution of the hidden layer because the number of neurons in the hidden layer is associated with over and underfitting, and over and underfitting were characteristics that could be detected and measured to set the stress stored in the functional blueprint. However, it is possible that functional blueprints could be used in other neuroevolutionary algorithms that evolve entire neural networks (such as NEAT and its extensions), or evolve neurons and connections in parallel, if they were provided with a spectrum-based performance metric that could be evaluated in candidate neural networks and used to set stress. BlueSANE's promising initial results demonstrate that it is possible to combine neural network and functional blueprint approaches to learning, self-adaptation, and nonlinear control.

3. Applicability to Other Problems

Here we discuss other areas which might benefit from the functional blueprint approach.

A. Classification and Machine Learning

Areas in which the functional blueprint approach to neuroevolution could be helpful include those of machine learning and pattern classification. Machine learning and pattern classification are areas in which overfitting is a known concern, as classifiers may unintentionally be

overtrained to specific data, and then have difficulty adapting to other data. They are used in such problems as data mining, automatic target recognition, and mathematical finance.

B. Neural Control

BlueSANE is a SANE-based neuroevolution algorithm that evolves the hidden layer of a neural network. The functional blueprint approach to neuroevolution could be used on any problem in which SANE-based neuroevolution algorithms can be effectively used. These include dynamic resource allocation on multiprocessor chips, process control in the metallurgical industry, robot arm control, playing Go, and creating melodies. [3,4,6,13]

4. Unresolved Questions and Future Work

It remains an open question whether the fact that different BlueSANE trials resulted in very different numbers of hidden-layer nodes, is a problem for the algorithm, or something to be expected. On one hand, the purpose of incorporating functional blueprints into SANE is to optimize this number in a way that the basic algorithm does not allow for. If we see that it is producing a wide variety of "optimal" values during different runs, that may indicate that its optimization methodology needs to be improved. On the other hand, the neural network topologies discovered by evolutionary algorithms are always going to be different from trial to trial – the nature of genetic algorithms is that there are many possible "breeding" possibilities due to the number of possible combinations of genes. Some neuroevolution algorithms (for example, NEAT) even take advantage of the variety of potential topologies, by preserving newly-discovered evolutionary lines as "species" so that they have a chance to develop their unique traits.

Another open problem is whether our chosen design of functional blueprints and our metric for the stress stored in functional blueprints are the best ones for improving the performance of neuroevolution algorithms. It is possible that the stress thresholds for adding or removing

neurons should be different for optimal performance, or that the tests for determining overfitting and underfitting should be changed. Though we explained our rationale for focusing on the behaviors of overfitting and underfitting, it may be that there are behaviors other than overfitting and underfitting that could be used to determine the optimal number of neurons in the hidden layer.

Perhaps the most obvious area for future work would be the integration of functional blueprints with other neuroevolution algorithms. It would be straightforward to do this with the family of SANE-derived neuroevolution algorithms, such as ESP, EuSANE, and Hierarchical SANE, because they all use SANE's technique of evolving a hidden-layer neuron population. However, it may be possible to integrate functional blueprints with aspects of other neuroevolution algorithms – not necessarily the optimization of their hidden layer - as well. For example, in an algorithm that evolves only the connections within a neural network, it could be applied to some aspect of optimizing the connection population.

Another question to consider in future research is whether genetic algorithms could be cut out of the process altogether and the network could be created entirely through the guided evolution of functional blueprints. This would require not only defining a desired behavior for the network (which is straightforward), but defining metrics of stress, and aspects of the network that should be modified by different types of stress, that would prompt the network to self-adapt in ways that brought it closer to its desired behavior.

IX. Conclusion

This thesis described BlueSANE, a new algorithm for optimizing neural networks through neuroevolution, that is an extension of the classic neuroevolution algorithm SANE. BlueSANE is a proof of concept for the idea that genetic-algorithm-based neuroevolution can be improved through the incorporation of functional blueprints, a guided evolutionary technique that specifies the desired behavior of a system or component. In BlueSANE, functional blueprints added to SANE the ability to optimize the number of neurons in the hidden layer of a standard three-layer feedforward neural network, by taking advantage of the principle that too many neurons in a

neural network's hidden layer can cause overfitting, and too few can cause underfitting. It did so by removing neurons between generations of candidate networks when it detected overfitting, and adding them when it detected underfitting.

The initial results for BlueSANE were promising when it was tested on simulated double cart-pole balancing, a classic benchmark problem. We ran experiments using two different sets of parameters in the overfitting detection mechanism, and in both experiments, BlueSANE outperformed original SANE, though by different average amounts.

The purpose of BlueSANE is not merely to introduce a new algorithm, but to demonstrate that functional blueprints can be useful in the algorithmic design of neural networks, and that functional blueprints combined with genetic algorithms can produce more effective evolution than genetic algorithms alone. There are many different ways in which functional blueprints could be implemented, or incorporated into SANE itself, and there are many other neuroevolution algorithms that could benefit from the approach. We hope that the work presented here will be a starting point for this field of investigation.

Bibliography

1. Adler, A., Yaman, F., Cleveland, J., Beal, J. *Morphogenetically engineered design variation*. International Conference on Morphological Computation, 2011.

2. Beal, J., *Functional blueprints: An approach to modularity in grown systems*. International Conference on Swarm Intelligence, 2010.

3. Chen, C.-C.J., Miikkulainen, R. *Creating melodies with evolving recurrent neural networks*. Proceedings of the International Joint Conference on Neural Networks (IJCNN '01). Vol. 3. pp 2241-2246.

4. D'Silva, T., Miikkulainen, R. *Learning dynamic obstacle avoidance for a robot arm using*

5. Fitzhugh, R. *Impulses and physiological states in theoretical models of nerve membrane*. Biophysical Journal, Vol. 1, pp 445-466. 1961.

6. Gomez, F., Miikkulainen, R. *Solving non-Markovian control tasks with neuroevolution*.

7. Hodgkin, A., Huxley, A. *A quantitative description of membrane current and its application to condition and excitation in nerve*. Journal of Physiology, Vol. 117, pp 500-544. 1952.

7. Kasshaun, Y., Sommer, G. *Efficient reinforcement learning through evolutionary acquisition of neural topologies*. Proceedings of the 13th European Symposium on Artificial Neural Networks (ESANN '05), pp 259-266. 2005

8. Kaikhah, K., Garlick, R. *Variable hidden layer sizing in Elman recurrent neuroevolution*. Applied Intelligence, Vol. 12:3, pp 193-205. 2000.

9. Knoester, D.B, Goldsby, H.J., McKinley, P.K. *Neuroevolution of mobile ad-hoc networks*. Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO '10). pp 603-610. 2010.

10. Moriarty, D.E., Mikkulainen, R. *Efficient reinforcement learning through symbiotic evolution*. Machine Learning. Vol. 22:1-3. pp 11-32. 1996.

11. Moriarty, D.E., Miikkulainen, R. *Hierarchical evolution of neural networks*. Technical Report, University of Texas at Austin. 1996.

12. Polani, D., Miikkulainen, R. *Eugenic neuroevolution for reinforcement learning*. Proceedings of the 2nd Genetic and Evolutionary Computation Conference (GECCO '00).

13. Richards, N., Moriarty, D.E, Mikkulainen, R. *Evolving neural networks to play Go*. Applied

14. Rummelhart, D.E., McClelland, J. *Parallel Distributed Processing: Explorations in the*

Microstructure of Cognition. Cambridge: MIT Press. 1986.

15. Saravanan, N., Fogel, DB., Nelson, KM. *A comparison of methods for self adaptation in evolutionary algorithms*. Biosystems. Vol. 36:2. pp 157-166. 1995.

16. Shi, M. *An empirical comparison of evolution and coevolution for designing artificial neural network game players*. Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO '08). 2008.

17 Stanley, K.O. *Efficient evolution of neural networks through complexification*. Doctoral dissertation. University of Texas at Austin, Department of Computer Science. 2004.

17. Werbos, P.J. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Doctoral dissertation, Harvard University. 1974.

18. http://nn.cs.utexas.edu/?javasane

19. http://nn.cs.utexas.edu/?ESP-Java