NTNU – Trondheim
Norwegian University of
Science and Technology

Evolving Echo State Networks for Minimally Cognitive Unsupervised Learning Tasks

Dag Stuan

Master of Science in Computer Science
Submission date: May 2013
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Dag Stuan

Evolving Echo State Networks for Minimally Cognitive Unsupervised Learning Tasks

Master thesis, June 2013

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

NTNU
Innovation and Creativity

Abstract

This thesis investigates the use of Echo State Networks (ESNs) in unsupervised learning environments, by employing Evolutionary Algorithms (EAs) to evolve ESNs to control an agent that performs a novel, minimally-cognitive learning task. The task employed in this thesis is a modified version of the classic video game Frogger. ESNs are investigated since they promise to combine the temporal abilities seen in other Recurrent Neural Networks (RNNs) with a straightforward method to train the network. However, previous work employing ESNs in unsupervised environments is lacking.

The evolved ESNs are compared to feed-forward Artificial Neural Networks (ANNs) as well as ESNs trained with regular supervised learning, in a comparative performance measure to find out which method is best suited to control Frogger.

The results from this thesis show that not only do ESNs work well with EAs, but they surpass traditional feed-forward ANNs on the Frogger task. Additionally, it is shown that for the Frogger task, evolved ESNs also outperform ESNs trained with supervised learning. The results from this work should serve as an encouragement to use ESNs for more tasks in the future, due to their competitive performance and ease of setup.

Sammendrag

This is a Norwegian translation of the abstract.

Denne avhandlingen undersøker bruken av *Echo State* Nettverk (ESNer) i miljøer med uovervåket læring, ved å bruke Evolusjonære Algoritmer (EAer) til å utvikle individer for å kontrollere en agent som utfører en ny, minimalt-kognitiv læringsoppgave. Læringsoppgaven brukt i denne avhandlingen er en modifisert versjon av det klassiske videospillet Frogger. ESNer er undersøkt fordi de lover å kombinere de temporale egenskapene sett i andre Tilbakevendende Nevrale Nettverk (RNNer) med en ukomplisert måte å trene nettverket på. Imidlertid er tidligere arbeid med å bruke ESNer i miljøer med uovervåket læring mangelfullt.

De utviklede ESNene blir sammenlignet med "feed-forward" kunstige nevrale nettverk (ANNer) så vel som ESNer trent med overvåket læring, i en komparativ ytelsesmåling for å finne ut hvilken metode som er best egnet til å kontrollere Frogger.

Resultatene fra denne avhandlingen viser ikke bare at ESNer fungerer med EAer, men at de også overgår tradisjonelle "feed-forward" ANNer på Frogger-oppgaven. I tillegg er det vist at utviklede ESNer også overgår ESNer trent med overvåket læring på Frogger-oppgaven. Resultatene fra dette arbeidet bør sees på som en oppfordring til å bruke ESNer mer i fremtiden, på grunn av deres konkurransedyktige ytelse og enkle oppsett.

Preface

This document was written as the author's master's thesis at the Department of Computer and Information Science at the Norwegian University of Science and Technology, during the spring of 2013.

The goal for the project was to look at the use of Echo State Networks (ESNs) in an unsupervised context, by employing Evolutionary Algorithms (EAs) to evolve ESNs meant to solve a novel minimally cognitive task.

I would like to thank my supervisor Keith L. Downing for guidance during the course of the project, and I would also like to thank Kai Olav Ellefsen for invaluable assistance with SEVANN throughout the project period. This project could not have been completed without their help.

Dag Stuan

Trondheim, May 21, 2013

Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

Contents **v**

List of Figures **ix**

List of Tables **xi**

List of Algorithms **xiii**

List of Abbreviations **xv**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Goals and Research Questions 2
1.3 Research Method 3
1.4 Contributions 3
1.5 Thesis Structure 3

2 Background Theory and Motivation **5**
2.1 Background Theory 5
2.1.1 Neuroevolution 5
2.1.2 Reservoir Computing 11
2.1.3 Echo State Networks 13
2.2 Related Work 20
2.2.1 Minimally Cognitive Learning Tasks 20

2.2.2 Evolutionary Algorithms and Echo State Networks 21
2.3 Structured Literature Review Protocol 22
2.3.1 Search Procedure 22
2.3.2 Selection Criteria 23
2.3.3 Results of the Literary Search 23
2.4 Motivation 23

3 Architecture 25
3.1 SEVANN 25
3.1.1 Scripts 27
3.1.2 Visualization 27
3.1.3 SEVANN Extensions 28
3.1.4 Echo State Network Implementation 30
3.2 Frogger 31
3.2.1 Game Rules 32
3.2.2 Implementation 33
3.2.3 Configuration Script 36
3.2.4 Hard-coded Heuristic 37
3.2.5 Fitness Function 37
3.2.6 Communication with SEVANN 38

4 Experiments and Results 41
4.1 Experimental Plan 41
4.1.1 Best Method for Controlling Frogger 41
4.1.2 Supervised Learning ESNs 42
4.2 Experimental Setup 42
4.2.1 Fitness and Game Setup 42
4.2.2 Best Method for Controlling Frogger 43
4.2.3 ESN Setup 45
4.2.4 Supervised Learning ESNs 47
4.3 Experimental Results 47
4.3.1 Evolved ESNs 47
4.3.2 Feed-Forward ANNs 52
4.3.3 Supervised Learning ESNs 56
4.3.4 Method Comparison 57

5 Evaluation and Conclusion 61
5.1 Evaluation 61
5.2 Discussion 62
5.2.1 ESNs, EAs, and Unsupervised Learning 62
5.2.2 SEVANN 64
5.3 Future Work 64

Bibliography **67**

Appendices **73**

A Fitness graphs **73**
A.1 Evolved ESNs 73
A.2 Evolved Feed-Forward ANNs 79

List of Figures

2.1 ANN encoding in a CPPN 9
2.2 Regular RNN training vs. RC training 12
2.3 A basic echo state network 14
2.4 Basic design of a visually-guided agent 20

3.1 An overview of SEVANN 26
3.2 Available visualization graphs in SEVANN 28
3.3 The original Frogger game 31
3.4 The implemented Frogger game 34
3.5 The input script used to configure the game 36
3.6 Communication between Frogger and SEVANN 39

4.1 Game board used for experiments 43
4.2 ESN setup for experiments 46
4.3 Probing of a well behaving ESN 50
4.4 Fitness plot of the best evolved ESN 50
4.5 Fitness plots of other evolved ESN runs 51
4.6 Probing of a well behaving feed-forward ANN 53
4.7 Fitness plot of the best evolved feed-forward ANN 54
4.8 Fitness plots of other feed-forward ANN runs 55
4.9 Average fitness across evolved runs 59

List of Tables

4.1 Fitness parameters for experiment 43
4.2 EA Parameters for experiment 44
4.3 ANN Parameters for experiment 44
4.4 Run statistics for evolved ESNs 48
4.5 Run statistics for feed-forward ANNs 52
4.6 Run statistics for supervised learning ESNs 56
4.7 Best run method comparison 57

List of Algorithms

1 Running of an ESN with supervised learning. 18
2 Hard coded heuristic for implemented Frogger. 37

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BP	Back Propagation
BPDC	Backpropagation Decorrelation
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
CPPNs	Compositional Pattern-Producing Networks
CTRNN	Continuous Time Recurrent Neural Network
EA	Evolutionary Algorithm
ES	Evolution Strategy
ESN	Echo State Network
ESP	Echo State Property
ESP	Enforced SubPopulations
GUI	Graphical User Interface
HyperNEAT	Hypercube-Based NeuroEvolution of Augmenting Topologies
LSM	Liquid State Machine
NE	Neuroevolution
NEAT	NeuroEvolution of Augmenting Topologies
RC	Reservoir Computing

RESP	Rule-based Enforced SubPopulations
RNN	Recurrent Neural Network
SANE	Symbiotic, Adaptive NeuroEvolution algorithm
SEVANN	Scripted EVolving Artificial Neural Networks

Chapter 1

Introduction

This Chapter provides an introduction to the project, lists goals and research questions, and also covers contributions and thesis structure.

1.1 Background and Motivation

This thesis discusses the usage of echo state networks in a novel minimally cognitive unsupervised learning task.

Minimally cognitive behavior is described by Beer [1996] as the simplest possible agent-environment systems that raise issues of genuine cognitive interest. In a minimally cognitive task, the focus is not on realism, but rather on what kind of behavior results from solving the task.

Evolutionary Algorithms (EAs) are a class of heuristic optimization algorithms inspired by biological evolution. These algorithms attempt to "evolve" solutions by applying genetic operators such as crossover and mutation to a "population" of individuals. There are many sub-areas of EAs, one of which is Neuroevolution (NE). Neuroevolution attempts to use EAs to evolve Artificial Neural Networks (ANNs). An artificial neural network consists of an interconnected group of artificial neurons which are inspired by the inner workings of biological neural networks.

Echo State Networks (ESNs) belong to a subset of ANNs called Recurrent Neural Networks (RNNs). These networks differ from normal feed-forward ANNs in that they allow recurrent connections inside the network [Wikipedia, 2012b]. This

gives the network a distinct advantage compared to feed-forward ANNs in that it can remember its own state, but it also brings with it the disadvantage that training and analyzing the network behavior becomes much more complex. While they are still equally difficult to analyze, echo state networks avoid the problem of training by only training the output links, leaving the rest of the network connections static and randomly generated. This combination of a network that can remember state while at the same time being easy to train makes ESNs a promising and interesting part of RNN research.

The main motivation for this thesis stems from the fact that while echo state networks already show great promise in a variety of supervised learning tasks, their use in unsupervised learning tasks remain largely unexplored. There have been some publications looking into unsupervised or reinforcement learning of ESNs, but not enough to conclusively decide upon their usefulness in these environments. This thesis aims to explore the use of ESNs in unsupervised learning tasks by applying evolutionary algorithms to ESNs in an attempt to solve a novel minimally cognitive unsupervised learning task, a modified version of the classic video game Frogger.

This project is an extension of the autumn project undertaken by the author in the autumn of 2012. For this reason, part of the background knowledge was obtained and small parts of implementation was completed before the start of the master's project.

1.2 Goals and Research Questions

The main goal of the thesis is stated as follows.

> *Investigate the use of echo state networks in unsupervised environments by evolving artificial neural networks to control the classic video game Frogger.*

Along with the main goal, some research questions are stated.

Research question 1 What is the best method for solving the task of controlling the minimally cognitive learning task Frogger: evolved echo state networks, evolved feed-forward artificial neural networks, or echo state networks trained with supervised learning?

Research question 2 Will evolved ESNs yield equal or better performance

than ESNs trained through supervised learning when applied to motor control tasks?

1.3 Research Method

The goals and research questions will be addressed by designing a system in order to do simulations, then designing a set of experiments meant to answer the research questions, and lastly analyze the results of the experiments. The system to be used will be based upon an already existing toolkit for evolving artificial neural networks, called SEVANN. The methodology used will be a mix between a model/abstraction model, and a design/experiment model.

1.4 Contributions

This project aims to contribute with a better understanding of the conceptual relationship between echo state networks, evolutionary algorithms, and unsupervised learning. The project will also provide a comparative performance measure between evolved echo state networks and other methods such as feed-forward artificial neural networks when applied to a minimally cognitive task. In addition to this, the project also aims to make the SEVANN toolkit a more robust framework for evolving artificial neural networks.

1.5 Thesis Structure

This thesis is organized as follows.

Chapter 1 introduces the problem domain, goal, and research questions.

Chapter 2 provides the state of the art of the fields covered in the thesis, specifically Reservoir Computing and Neuroevolution. In addition, this Chapter covers the structured literary search protocol, as well as the motivation for working with the project.

Chapter 3 describes the architecture used for the experiments. This amounts to a description of the implemented system used for the experiments.

Chapter 4 presents the experimental plan, setup, and finally results from the experiments.

Chapter 5 evaluates and concludes the thesis.

Chapter 2

Background Theory and Motivation

This Chapter aims to provide the reader with the background knowledge required to understand the research area in this project, specifically that of Neuroevolution and Reservoir Computing. In addition to covering background theory and related work, the Chapter also covers the structured literature review protocol used to find literature during the project, as well as discussing the motivation behind the project.

2.1 Background Theory

This Section presents background theory obtained during the course of the project. Some of this information was also obtained during the autumn project undertaken before this project.

2.1.1 Neuroevolution

Neuroevolution is a form of machine learning which focus on using Evolutionary Algorithms to train Artificial Neural Networks. There are many approaches to doing this. This Section first presents some recurring themes in neuroevolution, and then describes some of the techniques used. Yao provides a comprehensive,

but somewhat dated, overview of some algorithms used in neuroevolution in [Yao, 1999].

Neuroevolution algorithms can be split into two logical groups, those that evolve connection weights in an ANN with a static (in most cases fully connected) topology, and those that evolve the topology of the ANN along with the weights [James and Tucker, 2004]. When it comes to the group that evolve both topology and weights, this group is further divided between those that evolve the topology and weights at the same time, and those that evolve them separately.

Representation

When adapting ANNs to be used with EAs, one can choose between either a direct or an indirect encoding of the ANN. Most algorithms use a direct representation. However, in recent times, indirect representations have become more popular.

In a direct representation, the genotype in the EA is exactly the same as the phenotype, and the entire neural network is specified in the genotype. In an indirect representation however, the genotype specifies rules that define how an ANN should be constructed.

The simplest direct representation is the binary representation. Here, each weight of a fixed topology ANN is represented by a number of bits, and the EA used does not need any modifications from a standard EA. Other direct representations range from representing weights as real numbers, to more advanced representations like the one used in NEAT [Stanley and Miikkulainen, 2002] which contain additions such as the *innovation number*.

Indirect representations allow the algorithm freedom to not specify the ANN directly in the genotype, but rather let the genotype be a "recipe" for how to create the ANN, and creating the actual ANN in the phenotype. this kind of representation is used in HyperNEAT [Gauci and Stanley, 2010], where the genotype is a Compositional Pattern-Producing Network (CPPN) and the phenotype is an ANN.

NEAT

NEAT is a neuroevolutionary method introduced by Kenneth O. Stanley in 2002 [Stanley and Miikkulainen, 2002]. It was not the first neuroevolutionary method to try evolving both the topology and the weights of a neural network at the same time, it was however, one of the first methods with relative success.

NEAT builds upon some novel ideas within the field of neuroevolution which has proven to be highly effective, and NEAT outperforms other neuroevolutionary approaches on e.g. the double pole balancing task [Stanley and Miikkulainen, 2002].

NEAT is comprehensively described in its introductionary article [Stanley and Miikkulainen, 2002]. The next Sections briefly explain ideas in NEAT which were novel at the time, and are essential to NEATs effectiveness.

Gene tracking through historical markings One of the novel ideas presented in NEAT is the use of historical markings to track the origin of the genes in an ANN. This idea is grounded in the fact that there already exists unexploited information in any evolution that tells us exactly which genes match up with which [Stanley and Miikkulainen, 2002].

This tracking of historical information allows NEAT to effectively bypass the *Competing Conventions Problem* [Montana and Davis, 1989]. The competing conventions problem, also called the permutation problem, states that many permutations of the same vector essentially represent the same functionality, so ANNs with nodes in different orders can have very different genotypes even though their phenotype has the exact same functionality.

Another advantage given by the tracking of historical origins of the genes are that it allows crossover of genomes with different sizes, something which earlier NE algorithms had struggled with. When NEAT crosses over, the genes in both genomes with the same historical markings are lined up, and NEAT has rules controlling what to do with disjoint and excess genes.

Speciation One problem with evolving topology along with weights in an ANN is that in most cases, adding a neuron or connection to a network will not give the resulting network a good fitness initially. Thus, in a regular EA, the new network would be almost instantly disregarded, even though its added neuron or connection could eventually result in a network stronger than the current best.

NEAT solves this problem by speciating the population, such that an individual in the population compete within its own species rather than competing with the whole population. This speciation allows the individual to live longer, and eventually maybe evolve to be the global best network. NEAT allows speciation by *explicit fitness sharing* [Goldberg and Richardson, 1987]. In explicit fitness sharing, each organism in the same species must share the fitness of their niche [Stanley and Miikkulainen, 2002]. An added bonus is that this also ensures

that even if a species is performing well, its population can not grow infinitely and take over the population.

Complexification Another problem with many other neuroevolution algorithms that evolve both topology and weights, is that in many cases, the solutions they provide are overly complex and contain nodes and layers which are unnecessary. This is because many algorithms start with an initial population of random topologies, and have to remove nodes during the running of the EA. NEAT has a different approach.

NEAT starts out with a minimal neural network which only consists of every input being connected to every output, and no hidden layers. From that starting point, NEAT continuously complexifies the network to suit the problem it is presented with. This incremental complexification of the network means that the solutions NEAT provide should be minimally, or close to minimally, complex. Another advantage of the continuous complexification is that NEAT does not have to remove nodes as it runs, since unneeded nodes do not exist.

The next Sections describe some of the many extensions based upon NEAT which has been created in recent years.

HyperNEAT

Hybercube-based NEAT, or HyperNEAT, is an extension to the original NEAT method described in the previous Section. In HyperNEAT, NEAT is altered to evolve indirect representations of ANNs called CPPNs.

As pointed out in [Gauci and Stanley, 2010], NEAT cannot explicitly learn geometric regularities due to its direct representation of a solution. An example is when a checkers piece in one square is threatened by another in an adjacent square. In the checkers domain, NEAT cannot evolve a pattern of connectivity, such as one piece threatening another, that extends across the entire board, and is left having to solve the same problem multiple times. This is due to its direct representation. Because HyperNEAT represents each ANN indirectly in a CPPN, this theoretically means that HyperNEAT is able to discover such patterns of connectivity, and extend them across the board, instead of having to learn the solution several times. Instead of evolving ANNs like NEAT, HyperNEAT evolves CPPNs, which are described in the next Section.

CPPNs Compositional pattern-producing networks, or CPPNs, are networks composed of a variety of activation functions [Risi and Stanley, 2010]. The idea

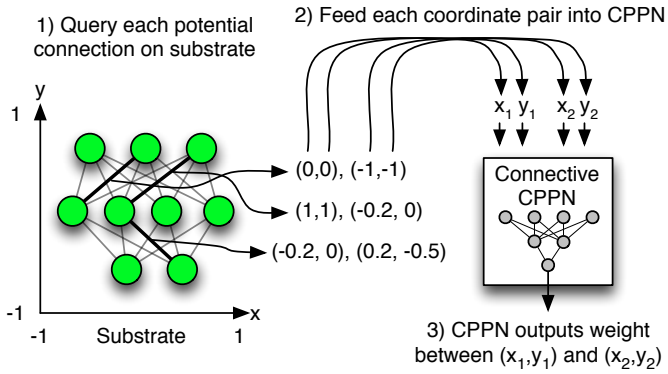


Figure 2.1: **ANN encoding in a CPPN.** A collection of nodes in an ANN, called a *substrate*, is assigned to coordinates in a given range (in this case -1 to 1) in all dimensions. (1) Every possible connection in the substrate is queried to check if it exists, the dark lines in the figure denote example queries. (2) The CPPN is queried with each pair, and determines which activation functions are connected to the pair. (3) The CPPN outputs the connection weight of the connection (0 if it doesn't exist). When all possible connections are queried, the result is the complete ANN.

is that this network acts as a pattern generator which outputs a pattern of connection weights within the geometry of the ANN. When HyperNEAT constructs an ANN, it queries the CPPNs for every possible connection among a pre-chosen set of points in geometric space. If a CPPN has the inputs x_1, y_1, x_2, and y_2, the resulting value when this point is queried is the *weight* between the two points (x_1, y_1) and (x_2, y_2). Each point is a neuron in an ANN. When every possible connection has been queried, the result is a complete ANN. An example of this is provided in Figure 2.1 In effect, the CPPN is painting a pattern on the inside of a four-dimensional hypercube, which is interpreted as the isomorphic connectivity pattern [Risi and Stanley, 2010]. This explains the origin of the name *hypercube-based* NEAT.

Because HyperNEAT can detect patterns in geometric space, the user can e.g. place the sensors of a robot in left to right in the same order they exist on the robot, and HyperNEAT can create CPPNs that correspond to the geometry of the robot. For a complete overview of HyperNEAT, the reader is referred to [Stanley et al., 2009].

Adaptive HyperNEAT

Adaptive HyperNEAT is another extension to the original NEAT-method, or more specifically, an extension to the HyperNEAT-method. In this extension, HyperNEAT is altered to evolve learning rules. Adaptive HyperNEAT was introduced by Sebastian Risi and Kenneth O. Stanley in [Risi and Stanley, 2010]. Adaptive HyperNEAT is grounded in the idea that regular HyperNEAT requires local learning rules to be discovered separately for each connection in the network. The idea is that since HyperNEAT already finds patterns in geometric space, it should also be able to find patterns of *learning rules* for the links in the same space. This should result in an *adaptive* ANN, which is able to change its own weights while the network is running, and not only during learning.

In Adaptive HyperNEAT, the evolved CPPNs are extended with the capability to not only evolve connectivity patterns, but also evolve patterns of learning rules. They use three different models to evolve learning rules, one general *iterated model*, a less general *Hebbian ABC model*, and lastly a *plain Hebbian* model, which all adds outputs to the CPPNs evolved. During runtime, the CPPN is queried, and a new ANN is created, possibly with different weights. They used a T-Maze domain, where the location of the reward sometimes changed, and obtained some fairly good results. The conclusion however, is that because the CPPN must be re-queried for each ANN connection, the current approach is too computationally expensive.

ESP and SANE

Enforced Subpopulations (ESP) is a neuroevolution method that extends the Symbiotic, Adaptive Neuroevolution algorithm (SANE) [Gomez and Miikkulainen, 2003]. Both ESP and SANE differ from other NE algorithms in that they evolve partial solutions or single neurons instead of evolving the entire network at once. SANE selects neurons from a single subpopulation to form ANNs, while ESP uses *explicit subtasks* where a separate subpopulation is allocated for each neuron in the network, and a given neuron can only be recombined with members of its own subpopulation [Gomez and Miikkulainen, 2003]. The evolution in SANE is characterized as *symbiotic evolution* in [Moriarty and Miikkulainen, 1997]. They define symbiotic evolution as a type of coevolution where individuals explicitly cooperate with each other and rely on the presence of other individuals to survive.

Both of these approaches are grounded in the idea that if each neuron can be optimized to do its own task well, the entire network should also work well, given

that every "best neuron" is selected from the population. ESP is comprehensively described in [Gomez and Miikkulainen, 1997] while SANE is covered in [Moriarty and Miikkulainen, 1996].

In [Fan et al., 2003], Fan, Lau, and Miikkulainen extended ESP to utilize prior knowledge in ESP, with a new system they called Rule-based ESP (RESP). They took inspiration from knowledge-based ANNs to transform a set of rules into an ANN, which they trained using the ESP method. In their Prey Capture domain, they were able to obtain better results than ordinary ESP given their prior knowledge of the domain.

CMA-ES

CMA-ES stands for Covariance Matrix Adaptation Evolution Strategy [Hansen and Ostermeier, 2001]. It is a popular Evolution Strategy (ES) for use in optimization problems. For every step in CMA-ES, a set number of individuals $\lambda > 1$ are generated by sampling a multivariate normal distribution $x = m^t + \sigma^t \times \mathcal{N}(0, C^t)$ where m^t is the average fitness of the best individuals in the previous generation, $\mathcal{N}(0, C^t)$ is a normal distribution with mean 0 and covariance matrix C^t, and σ^t is a scaling parameter. The individuals are then tested and sorted according to their fitness, and a new set of individuals are generated. CMA-ES is an almost parameter-free algorithm, and, only the number of offsprings λ is crucial to a successful evolution. CMA-ES has previously been applied to evolutionary ESN tasks with good results.

2.1.2 Reservoir Computing

Reservoir Computing (RC) is a field within Artificial Intelligence (AI) working with recurrent neural networks [Wikipedia, 2012b]. As opposed to a feed-forward neural network, a connection in an RNN may form a directed cycle, meaning that any node in the network can connect to nodes either in front of, or behind it in the network. A node may also be connected to itself. This type of connection topology gives an RNN the possibility of having an "internal state", or in layman's terms, "memory".

RNNs are much better than feed-forward networks at solving *temporal* problems, problems in which the network receives input from a stream of data where each data point is a state at some given time, such as weather or financial data. Temporal problems are possible to solve using normal feed-forward network structures, but it requires the use of extra input parameters, and introducing an artificial time horizon [Schrauwen et al., 2007].

Reservoir Computing has its origin through the observation that when an RNN has certain generic properties, training internal weights are not necessary, and only training of a single readout layer is necessary to get excellent performance in many tasks [Lukoševičius et al., 2012]. This observation is important, due to the fact that training of internal weights in RNNs have always been much more difficult than training feed-forward structures.

When training RNNs with classical gradient descent methods, the gradual change of network parameters during learning drives the network dynamics though bifurcations [Doya, 1993]. These bifurcations can in some cases create instabilities in the gradient information, where a small change in the direction of the gradient descent may lead to a completely different network solution, instead of the normal gradually decreasing error when running gradient descent. This in turn means that methods such as Back-Propagation (BP [Russell and Norvig, 2009, pg. 733-736]) cannot guarantee convergence when used with RNNs.

When an RNN is given the properties mentioned earlier, one may call it a *reservoir*, and this idea has given rise to *reservoir computing*. Figure 2.2 compares regular RNN training methods with RC.

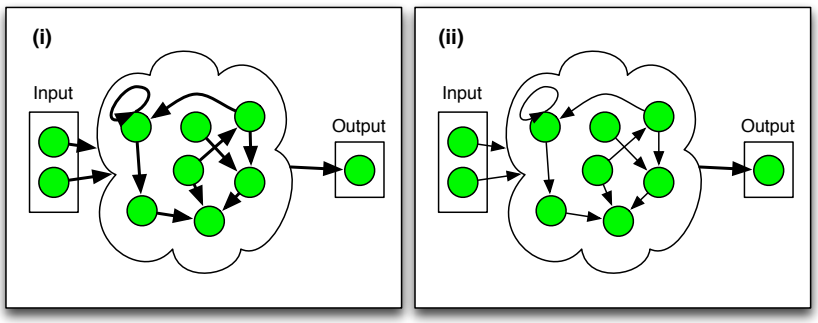


Figure 2.2: **Regular RNN training vs. RC training.** (i) Traditional gradient-descent methods change all the connection weights (bold arrows) to adapt to the training error. (ii) Using reservoir computing, only the output weights need to be trained.

Reservoir computing are mainly comprised of Liquid State Machines (LSNs) and echo state networks, however, many other related methods such as Backpropagation Decorrelation (BPDC) and temporal RNNs are also considered part of reservoir computing. A complete overview can be found in [Lukoševičius and

Jaeger, 2009]. This thesis will only cover LSMs and ESNs, so for an overview the reader is referred to the mentioned article.

Liquid State Machines

Liquid State Machines are one of the two pioneering methods in reservoir computing, the other being ESNs. The theory behind LSMs was developed independently from ESNs, and was introduced by [Maass et al., 2002]. LSMs were developed from a computational neuroscience background, aiming at elucidating the principal computational properties of neural microcircuits [Lukoševičius and Jaeger, 2009]. Due to its computational neuroscience background with the use of spiking neurons, LSMs use more biologically plausible neuron models in its reservoir. In LSM literature, the reservoir is often referred to as the *liquid*, to infer the metaphor of excited states as ripples on the surface of water [Lukoševičius and Jaeger, 2009].

2.1.3 Echo State Networks

Echo state networks are the first pioneering method in reservoir computing. ESNs were introduced by [Jaeger, 2001]. Just like regular RC, ESNs are grounded in the observation that if an RNN has certain generic properties, only the output layer needs to be trained. When utilizing supervised learning, the weights of the output layer can be trained using linear regression, and this will in many cases provide excellent performance. The part of an ESN that is untrained is called a *dynamical reservoir*, and the states of the reservoir are called *echoes* of its input history [Lukoševičius and Jaeger, 2009]. Lukoševičius [2012] provides a comprehensive guide for practical uses of ESNs, but unfortunately the guide is mainly focused on supervised learning.

Figure 2.3 shows a basic setup for an echo state network. Echo state networks have already been applied to real world tasks such as *speech recognition*, *handwriting recognition*, *robot motor control*, *financial forecasting*, and *medical* [Lukoševičius et al., 2012].

Node Activation

Node activation in an echo state network is done through matrix calculations. Given an ESN with K input nodes, N reservoir nodes, and L output nodes such as the one shown in Figure 2.3, its connection weights are contained within the

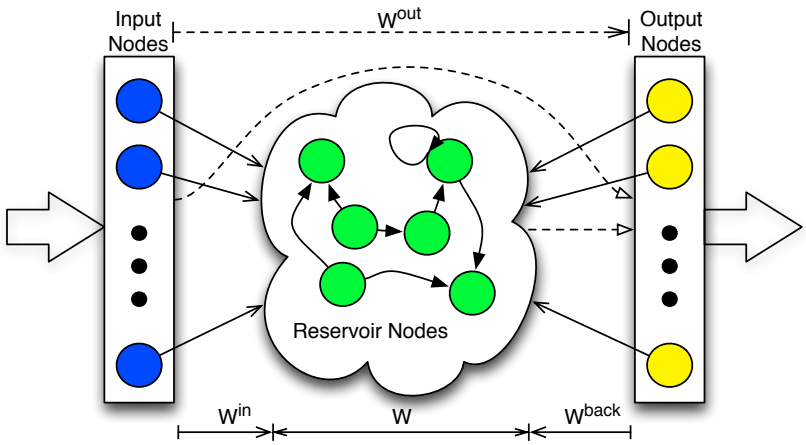


Figure 2.3: **A basic echo state network.** The network consists of K input nodes connected to N reservoir nodes through a weighted connection matrix W^{in}. The reservoir has an internal connection matrix W. W^{back} is an optional back-projection matrix connecting the output to the reservoir. Finally, the weights between the input and reservoir nodes to the L output nodes are collected in the matrix W^{out}. In the figure, the solid lines are static and randomly generated weights, while the dashed lines are trained weights.

matrices W^{in}, W, W^{out}, and optionally W^{back}. W^{in} is a $K \times N$ matrix containing the weights between the input and the reservoir nodes. W is an $N \times N$ matrix containing the weights between the reservoir nodes. W^{out} is an $L \times (K + N)$ matrix containing the weights between both the input- and reservoir-nodes, and the output nodes. And finally W^{back} is an optional $N \times L$ matrix containing the feedback weights between the output and the reservoir. W^{back} is not required for an ESN to function properly, and it has not been applied in this thesis.

At time $n = 1, 2, \ldots, n_{max}$, the inputs are the vector $u(n)$, and the output of the network is $y(n)$. For each time step, the activations of the reservoir nodes are collected in an $N \times 1$ vector $x(n) = (x_1(n), \ldots, x_N(n))$. The standard way of updating the reservoir nodes is as follows.

$$\overline{x}(n+1) = f(Wx(n) + W^{in}u(n+1) + W^{back}y(n)) \tag{2.1}$$

Where f is a sigmoid function such as tanh or logistic sigmoid. However, several

modifications exist to this standard update function, Lukoševičius [2012] recommends introducing a leaking rate α to the reservoir with the intent of controlling the speed of the reservoir dynamics (how dynamic the reservoir update is when receiving new input). The reservoir node update function then becomes

$$x(n+1) = (1-\alpha)x(n) + \alpha\overline{x}(n+1) \tag{2.2}$$

This is the update function used in this thesis. Having calculated $x(n+1)$, one can use W^{out} to calculate the network output as follows

$$y(n+1) = W^{out}[u(n+1); x(n+1)] \tag{2.3}$$

where $[u(n+1); x(n+1)]$ is a serial concatenation of the input vector and the reservoir activation state.

Parameters and Network Creation

Before creating the network, some global parameters must be selected by the user. The most important parameters are the size of the reservoir N, spectral radius of the reservoir $\rho(W)$, and leaking rate α. Other parameters are the input scaling of W^{in}, and the sparsity of the reservoir (the amount of internal connections inside the reservoir).

The spectral radius $\rho(W)$ of the reservoir is the most central of the global parameters in an ESN. Setting it correctly is required for the network to have the "Echo State Property" (ESP). Jaeger [2001] noted that under certain conditions, the network states become asymptotically independent of initial conditions and depend only on input history. Given this type of behavior, an ESN is said to have the "echo state property". Roughly put this means that if the network has been run for a long time, its current state is uniquely defined by its input history, and is independent of its initial network state. This property is governed by the value of the spectral radius $\rho(W)$. A small $\rho(W)$ means that the echoes from the input history will die out fast, while a larger $\rho(W)$ will make the network keep the echoes of its input history longer. This means that the spectral radius should be larger in tasks requiring longer memory of the input [Lukoševičius, 2012]. Even so, it is often hard to know how long input history a given task requires, which means that this parameter often require some trial and error to be set correctly. Lukoševičius [2012] recommends the spectral radius to have a value of less than 1.

As mentioned earlier, the leaking rate α of the reservoir is used with the intent of controlling the speed of reservoir dynamics, or how dynamic the reservoir is in terms of prioritizing new input. A small leaking rate induces slow dynamics, which can increase the short term memory of an ESN, and a large leaking rate has the opposite effect. While the leaking rate is not necessary for the ESP to hold, and its functionality seems relatively similar to that of the spectral radius, Jaeger et al. [2007] shows that using a leaking rate and optimizing it to the dynamics of the task at hand did provide better accuracy for the tasks they tested, and since it adds relatively little complexity to the ESN specification, literature generally recommends applying it. Lukoševičius [2012] recommends the leaking rate be set to match the speed of the dynamics in the input data. This is a rather vague recommendation, so much trial and error are required in setting the leaking rate correctly.

The main difference between the spectral radius and the leaking rate is that the spectral radius controls how long an ESN will keep a memory of previous states in its memory, while the leaking rate determines how much new input will affect the current network state. The spectral radius should be looked at as a way of controlling the amount of previous time steps the network should remember, while the leaking rate should be looked at as a way of stating to the ESN how much its internal state should change when given new input.

Lukoševičius [2012] provides some other best practices for generating an ESN, these are: (i) W should be a sparse matrix, (ii) the mean value of weights should be around zero, and (iii) N should be large enough to introduce new features for prediction performance. As with the other parameters, N will have to be chosen with some trial and error, and should be balanced with runtime performance and task complexity in mind.

The creation of an ESN with the "echo state property" is normally done with the following procedure.

- Generate W as a sparse matrix from a uniform distribution over $[-1, 1]$
- Normalize W to the spectral radius by scaling W with $\rho(W)/|\lambda_{max}|$, where $|\lambda_{max}|$ is the largest absolute value of the eigenvalues of W
- Now the untrained network has the echo state property regardless of how W^{in} and W^{back} are chosen.

After generating W, W^{in} and W^{back} are typically also generated from a uniform distribution over $[-1, 1]$.

Supervised Learning of Echo State Networks

Using ESNs with supervised learning tasks are by far the most common way of applying them. When an ESN is used with supervised learning, the user typically supplies a dataset of inputs U and expected output values Y^{target}. This allows an error to be measured, and the W^{out} matrix is typically created by using linear regression on the output error. The original method for using ESNs with supervised learning follows the following steps:

1. Generate a random ESN following the procedure in the previous Section
2. Run the network using the training input $u(n)$, and collect the corresponding reservoir activation states $x(n)$ into a state-activation matrix X
3. Compute the linear readout weights W^{out} from the reservoir using linear regression, minimizing the error measure between $y(n)$ and $y^{target}(n)$
4. Use the trained network on new input data $u(n)$ computing $y(n)$ using the trained output weights W^{out}

Pseudocode for the first three steps of the process are shown in Algorithm 1. Step four of the process consists of using the network normally, as shown in the end of Algorithm 1, except there is no need for keeping track of the expected outputs in Y^{target}.

Normally, the initial reservoir state of a newly generated network is set to 0. This creates an unnatural starting state, which is unlikely to be visited once the network has "warmed up" to the task. For this reason, the first steps of $u(n)$ are normally discarded and not used for training W^{out}. The network is run normally, except the states are not collected into X. Once the network has "warmed up", the network is run normally, collecting the states.

Step 3 in the procedure above is calculating the readout weights W^{out} using linear regression. This can be done in several ways. The normal way of doing this is called the *Wiener-Hopf* solution. Using the *Wiener-Hopf* solution, W^{out} is calculated as follows. Let $R = X'X$ be the correlation matrix of the collected reservoir states, and $P = X'Y^{target}$ be the cross-correlation matrix between the states and the desired outputs.

$$W^{out} = (R^{-1}P)' \tag{2.4}$$

When R is ill-conditioned, Equation 2.4 is not numerically stable. An ill-conditioned matrix is numerically unstable. This means that when trying to inverse the matrix, one may end up with an unexpected solution due to numerical instabili-

Algorithm 1 Running of an ESN with supervised learning.
The $\leftarrow$ symbol denotes setting of a variable, while the $\rightarrow$ symbol denotes either a function call or a variable access.

```
ESN ← generate_reservoir
num_init_runs ← user defined
run_number ← 0
U, num_training_cases, num_test_cases ← load_dataset
X ← empty matrix
Y^target ← empty matrix

for u(x) IN U[0 : num_training_cases] do
    input ← (u(x) → input)
    expected_output ← (u(x) → output)

    x(n) ← (ESN → calc_new_reservoir_state(input))

    if run_number is larger than num_init_runs then
        append x(n) to X
        append expected_output to Y^target
    end if
    increment run_number
end for

(ESN → W^out) ← calculate_output_weights(X, Y^target)

Y ← empty matrix
Y^target ← empty matrix

for u(x) IN U[num_training_cases : num_training_cases + num_test_cases] do
    input ← (u(x) → input)
    expected_output ← (u(x) → output)

    ESN → calc_new_reservoir_state(input)
    output ← (ESN → calc_output)

    append output to Y
    append expected_output to Y^target
end for

error ← calc_error(Y, Y^target)
```

ties [Cheney and Kincaid, 2008, pg. 321]. If this is the case, another solution is also possible using the pseudoinverse of X.

$$W^{out} = (X^{\dagger} Y^{target})' \tag{2.5}$$

The solution in Equation (2.5) is numerically stable, but it is slower than Equation (2.4). A third solution is also proposed, called Tikhonov regularization or ridge regression [Jaeger et al., 2007]. W^{out} is then calculated as

$$W^{out} = (R + \alpha^2 I)^{-1} P \tag{2.6}$$

where α^2 is a non-negative smoothing number, and I is the identity matrix.

Following the four steps mentioned will in many tasks give excellent performance. However, this way of training a network is limited due to its requirement that an expected output exists, this is not always the case in a real world scenario, which means some other way of training the network will be required.

Unsupervised Learning of Echo State Networks

When modifying the standard ESN specification to utilize unsupervised learning, the problem mainly consist of optimizing the weights in W^{out} representing the output weights of the network. The amount of dimensions in the problem are therefore directly related to the amount of input and reservoir nodes, so in these cases the user must balance the amount of problem dimensions to the complexity of the problem at hand. In addition to the values of W^{out}, some other parameters may also be added to the training, such as spectral radius, leaking rate, and input scaling.

When creating an ESN for use with unsupervised learning, the W^{out} matrix is typically also generated over a uniform distribution, and its weights are optimized throughout the running of the learning algorithm. This kind of problem is well suited for use with an evolutionary algorithm. When running the ESN one no longer needs to collect the reservoir states, and can instead calculate the output directly using Equations (2.2) and (2.3). Just like with supervised learning, the network is usually run for a number of initialization steps in order to “warm up” the network before actually calculating any output.

2.2 Related Work

This Section presents work that directly relates to the problem covered in this thesis.

2.2.1 Minimally Cognitive Learning Tasks

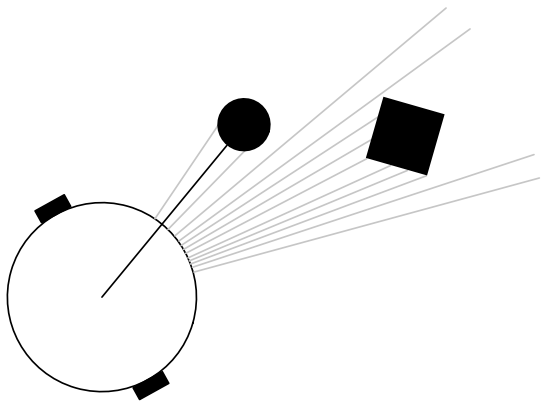


Figure 2.4: **Basic design of a visually-guided agent.** The agent consists of an eye (gray lines), two motors (filled rectangles), and a transparent arm (solid line) with an opaque hand (filled circle). The arm can rotate around the center of the agent and the hand can rotate around its attachment point with the arm. Figure adapted from [Beer, 1996].

Minimally cognitive behavior is described as the simplest possible agent-environment systems that raise issues of genuine cognitive interest [Beer, 1996]. In a minimally cognitive task, the focus is not on realism, but rather on what kind of behavior results from solving the task.

Beer normally applies Continuous Time Recurrent Neural Networks (CTRNNs) to his work with minimally cognitive behavior. A CTRNN is a dynamical systems model of biological neural networks. CTRNNs use a system of differential equations to model the effects on a neuron in the network. Like ESNs, this way of modelling neural input is very efficient since one does not have to model each

neuron in detail, instead resorting to a general model of neural activation updates. The normal way of applying CTRNNs for use with minimally cognitive behavior is to evolve them using a standard EA, and then later model the behavior that results from evolution. CTRNNs were originally proposed by [Hopfield, 1984].

An example of an agent used for minimally cognitive behavior is shown in Figure 2.4. Variations of this agent is used for tasks such as avoiding obstacles, identifying objects and catching them, and avoiding certain objects while catching others. Slocum et al. [2000] explore minimally cognitive behavior further, and evolve agents that can judge the passability of objects relative to their own body size, discriminate between visible parts of themselves and other objects in their environment, predict and remember the future location of objects in order to catch them blind, and switch their attention between multiple distal objects.

Most commonly, agents used for minimally cognitive behavior have a two dimensional environment, and are restricted to a single relatively simple task, with the intent of measuring the behavior exhibited by the agent after evolving a solution to the problem. This description of a minimally cognitive task fits well within the scope of many arcade games developed during the 70s, so games such as Frogger should (with a small amount of changes) also exhibit interesting minimally cognitive behavior.

2.2.2 Evolutionary Algorithms and Echo State Networks

Attempts to use ESNs along with evolutionary algorithms remain a relatively untested field. Some attempts at evolving ESNs have been done, however, the attempts are mainly restricted to supervised learning [Schmidhuber et al., 2007]. This Section presents a few attempts made at coupling EAs with ESNs.

Jiang et al. [2008] study the usage of ESNs with supervised learning for motor control tasks. They use the CMA-ES algorithm to evolve ESNs for use with both the double pole balancing problem and the univariate time series example from Jaeger's initial paper [Jaeger, 2001]. On both examples they obtained competitive results, and they also report better performance if more than just the output weights of the ESN are optimized by the ES.

Devert et al. [2007] use ESNs along with CMA-ES in a study of multi-cellular artificial embryogeny. They compare the performance of ESNs evolved with CMA-ES to the performance of NEAT when trying to evolve target pictures. Their results show that ESNs clearly outperform NEAT on the *disc*-problem. However NEAT

is much more evenly matched on the *half-disc* problem they also tested. In the end however, the combination of ESNs and CMA-ES manages to get a better average performance than NEAT in their experiments.

Chatzidimitriou and Mitkas [2010] use NEAT for evolving ESNs. They adapt NEATs ideas such as historical markings, complexification and speciation to the specifics of ESNs. They try to optimize all parameters of the reservoir along with the output weights using NEAT, starting with a reservoir containing a single neuron. They use this combination both for supervised and reinforcement learning. They test supervised learning using the Mackey-Glass system [Glass and Mackey, 2010], while reinforcement learning is tested using a Mountain Car task [Whiteson and Stone, 2006] as well as the single and double pole balancing tasks. They report that their methodology worked on all tasks, and conclude that reservoir optimization is an area which needs further research.

Despite the fact that most of these articles report good performance in the variety of tasks tested, not enough research has has been done when applying ESNs to EAs for use with unsupervised learning tasks. This thesis attempts to rectify at least some part of this.

2.3 Structured Literature Review Protocol

This Section describes the structured literary search protocol used to find literature during the project.

2.3.1 Search Procedure

One of the requirements of the project was to use a systematic and comprehensive search protocol to, as effectively as possible, end up with the background literature needed for achieving the rest of the goals for the project. In this regard, a search procedure was set up.

Search Keywords: “reservoir computing”, “neural networks”, “artificial neural networks”, “reservoir computing”, “echo state networks”, and “liquid state networks” were used as search terms. They were combined with the terms “evolving”, “neuroevolution”, “robot control”, “unsupervised learning”, “reinforcement learning” to form the keywords used in the search procedure.

The following sources were used for the search:

Evolving Neural Networks [Miikkulainen, 2005] This lecture, of which the slides were given to me by Keith Downing, provided a comprehensive overview of neuroevolution, in addition to providing a range of sources which proved useful.

Google Scholar Because NTNU has a custom Google Scholar page, the search engine was used extensively along with the keywords mentioned above to find relevant articles.

Google In addition to Google Scholar, google was used to have a broader scope of potential matches.

Wikipedia sources Relevant wikipedia articles were used to find article sources.

2.3.2 Selection Criteria

After having compiled studies which matched the search keywords, the studies to read were selected according to these criteria:

- It had to be published in the last 20 years
- It had to have reasonably good results.
- It had to be understandable enough to be possible to implement given my limited resources.

After having selected articles based on these criteria, I further communicated with my advisor to form a final set of articles to be read.

2.3.3 Results of the Literary Search

After having searched and selected articles according to the selection criteria mentioned above and communicated with my advisor on the initial selection, a total of 15 articles were selected and read for further study. The selected articles provided a good background for the field of study.

2.4 Motivation

This project looked very interesting to me because it covers an area which is almost completely uncovered by previous work. During the literary search only a few articles were found which even mention having tried echo state networks

in unsupervised environments, and this meant that there was a lot of potential in this project to bring something new to the Reservoir Computing community. Echo state networks have as previously mentioned been shown to give good results in a number of supervised learning tasks, and given that they easily adapt to unsupervised learning, it is strange that they haven't been tried more. Very few real world tasks are completely supervised, so if echo state networks behave well when applied to reinforcement and unsupervised learning tasks they can potentially be applied to more realistic tasks in the future.

Chapter 3

Architecture

This Chapter aims to describe the architecture used to fulfill the project goal. During the project, a lot of implementation has been done. During the autumn project, a client for the old arcade game Frogger was implemented, and during the master project, the SEVANN toolkit has been expanded both with support for evolving echo state networks, as well as with general enhancements which aim to make it easier to expand in the future.

3.1 SEVANN

Scripted Evolving Artificial Neural Networks, or SEVANN, is a general framework for evolving bio-inspired neural networks [Downing, 2010]. SEVANN uses scripts to allow the user to define both the neural networks to train, and the bio-inspired parameters used to evolve them. The user is free to decide which of the parameters that should be evolved and which are static, so a user can say that one hidden layer should contain e.g. 3 hidden nodes, while another should contain an evolved amount of nodes. SEVANN also contains functionality to let the user visualize the behavior of the evolved ANNs. All of this makes SEVANN a very flexible toolkit to use when attempting to solve a task with neural networks, and the user can theoretically begin using it without extensive knowledge of the science behind artificial neural networks. Figure 3.1 provides an overview of the functionality in SEVANN.

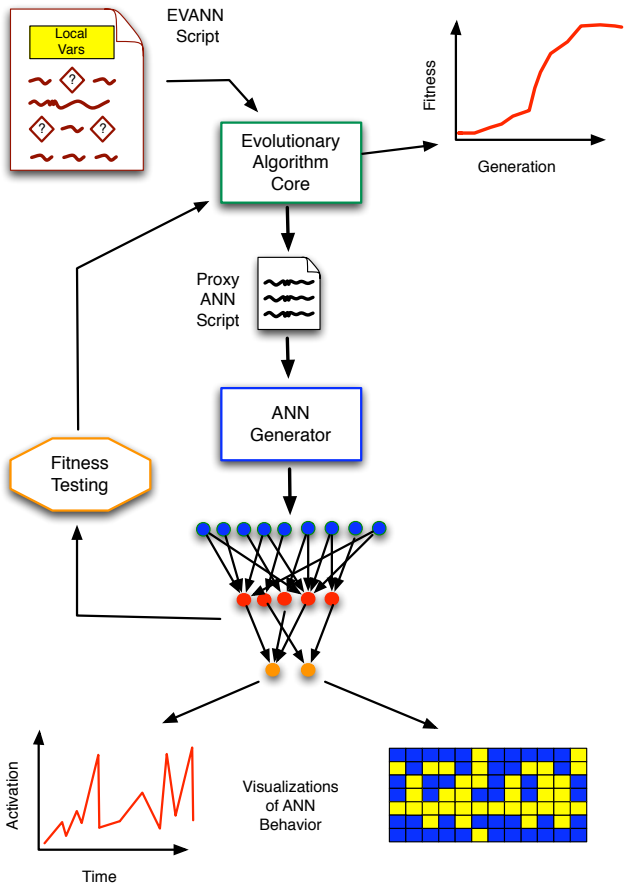


Figure 3.1: **An overview of SEVANN.** (i) The user defines an *EVANN*-script containing the parameters of the EA, topology of the ANN etc. (ii) The script is fed into the *Evolutionary Algorithm Core*, which uses a proxy ANN script to generate an ANN which is then fitness-tested, and, if good enough, brought along for the next generation. The Evolutionary Algorithm Core can display a graph-overview of the fitness of the population for each generation. SEVANN also includes functionality to visualize ANN behavior. Figure used with permission of author [Downing, 2010].

3.1.1 Scripts

To use SEVANN, the user has to define three scripts, an EVA-script, a TOPOLOGY-script, and a NIOMAP.

eva This is the main script. This script first defines the parameters of the evolutionary algorithm to be used, such as population size and number of generations. The script then defines the type of ANN to be evolved, and which file to read the initial topology from. Finally, the location of the *niomap* is defined.

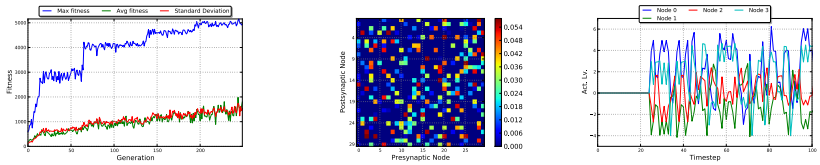
topology This script defines the topology ANN to be trained. In this script, the user defines which parameters should be defined by evolution, and which are fixed. The user is free to let evolution determine almost all the parameters, with the exception of the input and output layers, which require the user to determine the number of nodes. Layer execution order and back-propagation training order are also defined in this script

niomap This script defines the inputs and outputs of the ANN to be evolved. It is possible to use a dataset for inputs and outputs to train the network, as well as an "agent" that provides the data to the inputs using a function defined in this script. The this thesis has used an "agent"-setup.

When the user has defined the scripts, SEVANN does the rest, evolving an ANN suited for the task. A preliminary description of the entire system can be found in [Downing, 2010].

3.1.2 Visualization

In addition to plotting normal fitness graphs over the course of an EA run, SEVANN has a Graphical User Interface (GUI) which allow the user to visualize runs of ANNs after evolution is complete. In addition to doing this directly after an EA run, the user may select a previously saved ANN for use with the visualization tools. The visualization tools allow the user to add *probes* to an ANN which record certain node or link values over time, such as activation value or membrane potential. The tools then graph the selected values over time in either a continuous- or a grid plot. The visualization tools in SEVANN are intended to allow the user to analyze why a given network behaves the way it does. An example of the visualization graphs available to the user can be seen in Figure 3.2.



(a) Fitness graph. (b) Grid plot. (c) Continuous plot.

Figure 3.2: **Available visualization graphs in SEVANN.** These are some example visualization graphs that are used when probing an ANN with SEVANN. Figure a shows a standard fitness graph for the generations of an EA, Figure b shows a grid plot of reservoir weights in an ESN, and Figure c shows a continuous plot of the first 100 output activations over time in an ESN. Notice that during the first steps in Figure c there is no activation of the nodes, as these are ESN initialization steps.

3.1.3 SEVANN Extensions

During the course of the project, several modifications and enhancements to the SEVANN toolkit has been made, both in order to allow easier extension of SEVANN in the future, and also to allow the use of ESNs. The most important enhancements made will be described in the following Sections. In addition to the features described below, preliminary support for supervised learning has also been implemented. However, the supervised learning implementation is mainly directed towards ESNs for now.

Abstraction

In earlier versions of SEVANN, nodes and arcs were represented by objects, and each layer or link had a set of node or arc objects that were used to pass activation values through the network. This works for many ANN types, but not for all. An ESN is one example of an ANN that is better suited using a different representation of links and arcs. For this reason, SEVANN was extended with abstraction in mind.

In the new version of SEVANN, an ANN layer or link can have its nodes or arcs represented in one of two ways, it can either have the normal list of node- or arc-objects, or a list of numbers describing the node activation levels or arc weights. This allows SEVANN to both represent ANN types which does a lot of calculation at the node level, or ANN types which higher level equations for node activation updates, such as matrix calculations. The addition of lists of numbers

for nodes and arcs in SEVANN are done through the notion of abstract layers and links.

The addition of abstract layers and links to SEVANN brings with it several advantages. First, it allows easier extension of SEVANN with ANN types that does not need explicit node or arc objects, and in addition, it makes a lot of the computation run a lot faster, as function calls to objects are a relatively costly operation in Python. The ESN implementation described in the Section 3.1.4 is the first implementation making use of abstract layers and links in SEVANN.

Subclassing

Another important extension done to SEVANN during the project period was to allow for easier subclassing of the ANN implementation. With the new implementation, all that needs to be done to implement a custom ANN-specification is to subclass SEVANNs implementation of the desired class, and add the new class to the python-path. The ability to subclass the main classes of the ANN implementation made it a lot easier to extend SEVANN with support for ESNs, as reimplementing existing functionality was no longer required to extend it.

API

During the project, SEVANN was extended with Application Programming Interface (API) support for saving and loading previously saved ANNs. API support for re-running a loaded ANN was also implemented.

Implementing an API for SEVANN was done for several reasons. The main reason was to allow flexibility when evolving. Previous versions of SEVANN required the use of the GUI during evolutionary runs in order to access the visualization GUI. Since SEVANN is intended to be used in a variety of configurations, this was not optimal. As an example, the runs done for this thesis were done using a Linux server with no access to the GUI, so the visualizations had to be done after evolution was complete and the evolved ANN was saved.

The API was implemented along with extensions to the visualization interface to allow visualization of previously saved ANNs. The ability to rerun a given ANN via an API makes it easy to visualize runs using agents, and also enables the user to easily test an evolved ANN in environments that differ from the one it was originally evolved in. The ANN-API for SEVANN is intended as a first step, and future versions of SEVANN will hopefully extend the API with many more useful features, such as graphing or probing.

3.1.4 Echo State Network Implementation

This Section describes the modifications done to implement support for echo state networks in SEVANN. This Section should also serve as a tutorial for those who would wish to extend SEVANN with new ANN-types in the future.

Due to the implementation of easy subclassing mentioned earlier, implementing ESNs for use with SEVANN was relatively straightforward. All that needed to be done was to subclass SEVANNs implementations of layers and links, modify the parts that was special for ESNs, and then run SEVANN normally. The ESN implementation was done using four classes.

To verify the implementation, the standard ESN-model was first implemented outside SEVANN and verified to be working correctly using established supervised learning tasks. This external implementation was then routinely compared against the SEVANN implementation using the same learning tasks, with the intent of making sure SEVANN behaved similarly.

Classes

The classes used in the ESN implementation are as follows. All the implemented classes subclass SEVANNs implementation.

esann This class is subclassed from *ann.py*, which is the main class of the ANN implementation in SEVANN. The class contains the run-loops for an ANN, and various resets to allow an ANN to be re-run at a later stage. The implementation of supervised ESN learning is also contained within this class.

esn_link This class is subclassed from *annlink.py*, which contains SEVANNs implementation of an ANN link. This class contains logic for initialization of ESN specific links. This class initializes the ESN topology, doing input scaling and spectral radius scaling as needed. The class also contains logic to do linear regression when using supervised learning.

esn_layer This class is subclassed from *annlayer.py*, which contains SEVANNs implementation of a layer in an ANN. This class contains the logic required to fire the reservoir in an ESN.

esn_out_layer Like esn_layer, this class is also subclassed from *annlayer.py*. This class contains the firing mechanism of the output in an ESN. This class requires that it is connected to a single link, which in turn is connected to an esn_layer.

Parameters

ESNs has a set of required parameters that are set before network creation. The most important of which are reservoir size, number of reservoir connections, input scaling, spectral radius, and leaking rate. All of these variables are implemented in SEVANN, and are evolvable based on the user's wishes.

For evolution of ESNs to work correctly, the input-to-reservoir and the reservoir-to-reservoir weights have to remain static throughout evolution. Since SEVANN by default recreates all links for every generation by generating networks from the genotype, the ESN implementation uses a random seed to make sure the links that are static remain static throughout evolution. The user specifies a random seed in the script, and this seed is used to regenerate the same link weights throughout the evolutionary run.

3.2 Frogger

Figure 3.3: **The original Frogger game.** The player controls the frogs, which start out at the bottom of the screen, and the objective is to get all the frogs to the top alive.

The minimally cognitive learning task selected as the learning task for the project was a modified version of the old arcade game Frogger [Wikipedia, 2012a]. In the

original game, the player starts out controlling a frog which is at the bottom of the screen. The objective of the game is to bring the frog to the top of the screen alive. The bottom half of the screen is a road which has a varying amount of traffic, consisting of cars, motor bikes, buses etc. while the top half is a river with logs floating on it. The player has to avoid the cars, use the floating logs to get across the river, and finally end up at the top of the screen safely. In the original game, the player is given five frogs, and the goal is to get as many of them across the screen as possible. Figure 3.3 shows a screenshot from the original Frogger game.

3.2.1 Game Rules

While the general idea of the frog avoiding cars were the same, some modifications to the rules were made with the intent of making the game more suited for EAs.

Coins The original game has survival as the goal, and while this works well for a video game, it has some flaws which make it less suited for use with an agent controlled by an ANN. The fitness of the ANN controlling the frog would be hard to determine, given that the only two states possible would be *alive*, and *dead*, with an optional *time alive*. This makes it hard to calculate a fitness, and determine how well the ANN is behaving. The notion of *coins* was therefore introduced, and the goal of the game is no longer to get to the top alive, but rather to pick up as many coins as possible within a given timeframe. The game board contains one coin at all times. The coin can either be placed on pre-selected positions, or be randomly placed around the game board.

River The original game first requires the player to avoid cars, and then to navigate the river by jumping on top of logs. A twofold task like this was deemed too hard for the ANNs to handle, so the implemented version contains only cars.

Cars In the original Frogger, the cars can have various sizes. Ranging from small motorcycles, to buses. The implementation in this project assumes equal lengths for all cars.

Death In the original game, if a frog dies, the player has one less frog to move to the top. In this implementation, the frog is moved back to the initial position, and the gameplay continues. Note that if the frog attempts to move into a grid square containing both a car and a coin, the frog will pick up the coin and then immediately die.

Movement In the original game, the player can move in all directions, and a switch of direction always includes a jump in the new direction. So if a player is facing forward at (x, y) location $(2, 0)$ and presses the left key, the location will be switched to $(1, 0)$ with the frog facing left. In this implementation, the switching of direction includes no jumps, so if the same was done in this implementation, the frog would still be in location $(2, 0)$ facing left, and a forward key-press would move the frog to $(1, 0)$.

Sight In the original game, the player has vision of the entire game board. For this implementation, it was decided to give the frog a total of nine vision sensors. The sensors are given an activation value of 0 if the square they cover are not covered by a car, and 1 if there is a car there. The frog has a total of 9 sensors. In addition to this, the frog has a boolean sensor which is *true* if the coin is inside the frogs *cone* of vision. This provides a total of 10 input nodes to the ANN, one for each sensor, and one for the cone. A visual explanation of the sensors is provided in Figure 3.4.

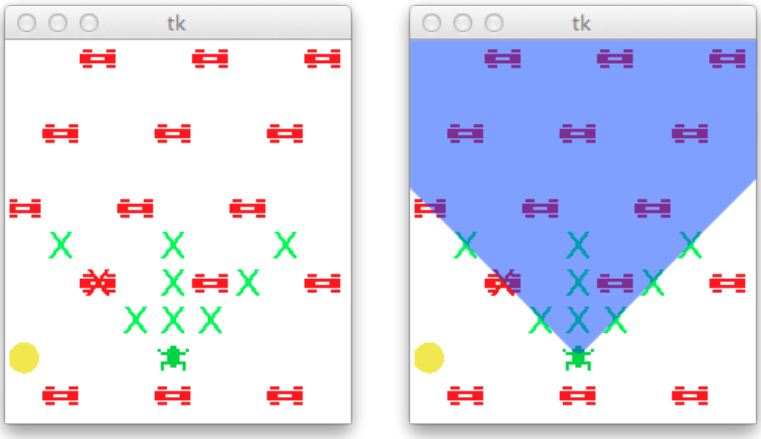
Loop around In this implementation, the frog is allowed to pass through the edges of the game board. When the frog passes through the edge, it ends up on the other side of the game board. The same principle applies to the sight sensors, so the frog can sense the positions of the cars on the other side of the board. However, the *cone* sensor does not wrap around the game board. The reason for this is to encourage the evolved frogs not to walk through the edges, but rather stay inside the boundaries of the game board.

3.2.2 Implementation

A screenshot of the graphical user interface of the implemented game can be found in Figure 3.4. The game is implemented using Python as the programming language. When running the system, the user needs to specify a configuration script, which is described in Section 3.2.3. To make it as easy as possible to integrate the game into SEVANN, the game is very modular. In addition to connecting the system to SEVANN, a hard-coded heuristic was also developed to give a baseline performance, the hard-coded heuristic is described in Section 3.2.4. The system is split into eight classes.

Frogger This class contains the main class of the system, which optionally sets up the GUI, and initializes the game-grid according to the script provided by the user.

Grid This class contains the logic governing the placements of the objects in the



(a) Screenshot of implemented Frogger (b) The vision-cone of the frog.

Figure 3.4: **The implemented Frogger game.** Figure a shows a screenshot of the version of Frogger implemented for this project. The frog can be seen in the second to last row, and its sensors are the crosses. Each vision sensor has a value of 0 if there is no car there, and a value of 1 if the square it covers is currently occupied by a car. The coin can be seen as the filled circle. Figure b shows the same screenshot, but displays the current cone of vision for the frog. If the coin is inside this vision cone, the cone-sensor has a value of 1. Otherwise, it has a value of -1.

game. Every time an object (e.g. the frog) wants to move, it queries the grid, and the grid updates the position of the object. The grid also controls events such as the frog dying or picking up the coin. When using the GUI, the Grid is a thread which executes a step at a given interval to update the game state.

Frog Contains the main logic for the frog. Such as calculating its current vision cone, and also all the movement commands. The frog has a static set of sensors. The vision cone of the frog has a value of -1 or 1 depending on whether or not the coin is inside it.

Sensor The frog has a total of nine sensors. Each sensor has a coordinate relative to the frog, and has an activation value of either 1 or 0 depending on the contents of the square its covering. Whenever something moves on the grid, the sensor values are updated accordingly.

Lane The game-grid is split into lanes, and each lane can either be empty or contain a moving train of cars. The number of lanes and the characteristics of each lane (car speed, number of cars, etc.) are specified in the user defined script.

Car Every lane has a set of cars that drive in it.

Coin The class defining the coin. The every time the coin is picked up by the frog, the coin is asked to pop up at a new location on the grid.

Fitness This file contains the fitness function used to measure the performance of a given frog.

FrogController This class is used to visualize runs coming from saved ANNs generated by SEVANN. This class also contains the hard-coded heuristic used as a baseline for fitness.

3.2.3 Configuration Script

To make the game as flexible and easily configurable as possible, the environment is controlled by a user defined script. The template used for the script is shown in Figure 3.5.

```
Grid:
width,height,square_size
Lanes:
lane_pos_y,num_cars,direction,nap,start_X
...
Frog:
start_X,start_Y
```

(a) Template script

```
Grid:
9,10,24
Lanes:
0,3,0,1,0
2,3,0,1,2
4,3,0,1,4
6,3,0,1,6
9,3,0,1,8
Frog:
4,8
```

(b) Script used in Figure 3.4

Figure 3.5: **The input script used to configure the game.** Figure a shows a template for the script setup. The user first defines the size of the grid, and the *square size* of each square in the grid. The square size is only used when the GUI is displayed. The user then defines a set of *Lanes*. Each lane has a *position*, a *number of cars* running in it, a *direction*, a *nap*, and a *start* X-position. The *nap* is a number defining how many steps the cars should remain stationary before moving. A value of 1 means the cars move every step, while a value of 2 means every other step etc. Finally, the user defines the starting position of the frog. Figure b shows the script used to create the game board shown in Figure 3.4.

3.2.4 Hard-coded Heuristic

The hard-coded heuristic implemented for the system was implemented to give a basic baseline of performance that the evolved ANNs could be compared with. The heuristic used is presented in Algorithm 2.

Algorithm 2 Hard coded heuristic for implemented Frogger.
The $\leftarrow$ symbol denotes setting of a variable, while the $\rightarrow$ symbol denotes either a function call or a variable access.

```
while true do
  if (frog orientation is left OR frog orientation is right) AND front frog sensor
  is activated then
    select a direction at random
    turn frog to selected direction
    frog → forward
  else
    cone ← (frog → calculate_cone)
    coin_coords ← (grid → coin → coords)
    if coin_coords ∈ cone AND NOT front frog sensor is activated then
      frog → forward
    else
      select a direction at random
      turn frog to selected direction
    end if
  end if
end while
```

3.2.5 Fitness Function

The fitness function used to determine the performance of the frog is very important. In order for evolution to succeed, the fitness function has to be designed in such a way that it gives the frogs partial credit for doing some subset of the total task correctly, while still giving very well performing frogs a large enough fitness to make them stand out of the crowd. The best method for controlling Frogger is defined as the method which best balances the task of picking up coins with the task of avoiding the cars, and this balance must be reflected in the fitness function. During the project, the fitness function was redesigned several times with this in mind. The final fitness function used during the experiments is shown in Equation 3.1. The fitness for a frog, denoted F, is given as

$$F = 250 + (w_P \cdot P) - (w_D \cdot D) + (w_C \cdot C) + (w_{M_t} \cdot M_t) - (w_{M_a} \cdot M_a) \quad (3.1)$$

Where P denotes the amount of times the frog has picked up the coin, D denotes the amount of times the frog has died, C denotes the amount of times the frog faced with the coin inside its vision-cone (the amount of times the cone-sensor returned *true* when queried), M_t denotes the amount of times the frog moved towards the coin (moved forward with the coin inside its cone), and M_a denotes the amount of times the frog moved away from the coin (moved forward with the coin outside its cone). w_P, w_D, w_C, w_{M_t}, and w_{M_a} are weights to adjust the importance of each event. The reason the fitness starts at 250 is to avoid a fitness-value lower than 0, since some selection mechanisms such as sigma scaling doesn't support negative fitness values. In addition to the raw fitness function, the fitness is deducted by 400 points if only one output was fired (e.g. the frog only moved forward). Also, a measure of the efficiency of the coin pickups, called the pickup-to-death (PDR) score was included in the fitness calculation, so if the frog picks up more coins than a given threshold T, a PDR-score is added to the fitness. the PDR-score is calculated as $\text{PDR} = w_{PDR} \cdot (P/D)$ where w_{PDR} is the weight given to the PDR-score. The PDR threshold is added to allow evolution to first focus on picking up as many coins as possible, and then later focus on dying as little as possible. If the coin positions during the fitness measurement were fixed, the PDR-score given is divided by 2, since running using fixed position coins is viewed as an easier task than randomly positioned coins. Finally, after potential additions and deductions, if the resulting fitness value is negative, it is nulled.

3.2.6 Communication with SEVANN

As mentioned earlier, the implemented Frogger is set up to be used as an *agent* in SEVANN. The only work needed to be done to accomplish this was to define the scripts mentioned in Section 3.1, as well as defining the *fitness*-function for the ANNs generated by SEVANN, which is used to determine the performance of each ANN in the generated population.

After defining the scripts and the fitness-function, the communication with SEVANN is shown in Figure 3.6.

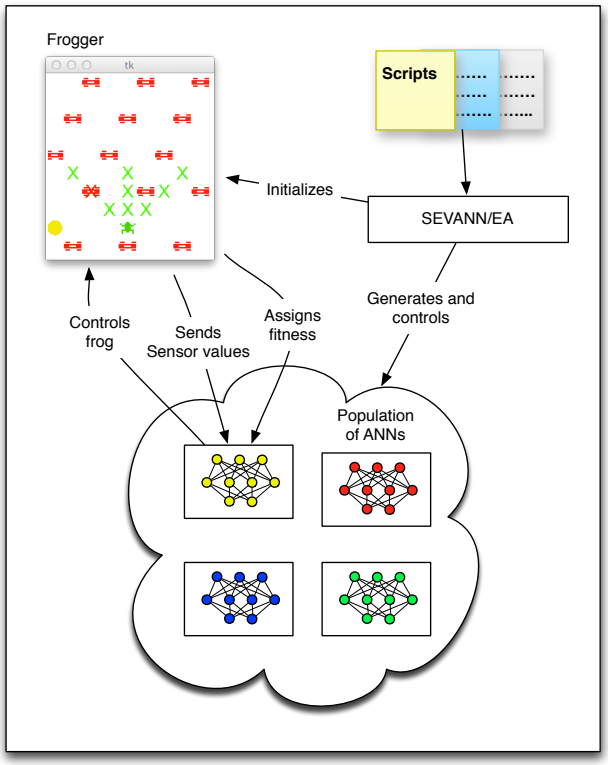


Figure 3.6: **Communication between Frogger and SEVANN.** The user starts SEVANN providing the configuration scripts. SEVANN then initializes Frogger, and a population of ANNs which are created according to the script variables. For every ANN in the population, SEVANN determines its fitness by using it to control the frog a given number of simulation steps. SEVANN first asks for sensor inputs, and uses these as inputs to the network. The output from the network is then used as a move-command in the game. After having controlled each frog the given amount of steps, the game is reset for use with the next ANN. SEVANN follows standard EA procedures such as crossover and mutation, and repeats this process for every ANN in the population until a stopping criteria such as maximum fitness or maximum number of generations is met.

Chapter 4

Experiments and Results

This Chapter presents the experimental setup used to conduct the experiment in the project, as well as describing the results obtained from the experiment.

4.1 Experimental Plan

To address the research questions in Section 1.2, an experiment was set up. The goal of the experiment was first of all to see if evolved ESNs were able to adequately control Frogger at all, and then compare the solution found by the evolved ESNs to the performance of the the hard-coded heuristic, evolved feed-forward ANNs, and finally supervised learning ESNs trained using linear regression. This experiment was intended to answer both which method is best suited for controlling Frogger, and also whether or not evolved ESNs can compete with ESNs trained using linear regression in motor control tasks.

4.1.1 Best Method for Controlling Frogger

To find the best method for controlling Frogger, SEVANN was used to evolve the feed-forward ANNs and the evolved ESNs. SEVANN was also used to evolve the parameters used to create the supervised learning ESNs. The different methods were compared using the fitness measure described in Section 3.2.5. All well-performing runs were visualized using the Frogger GUI to analyze their behavior. In addition, the best evolved ANNs were probed with SEVANNs GUI to attempt at analyzing their activation patterns.

4.1.2 Supervised Learning ESNs

In order to create comparable ESNs using supervised learning, a dataset was created by manually playing the game with random coin positions. Random coin positions were used since a network solving the problem with random coin positions is considered superior to one solving it with fixed coin positions. The dataset contained 2000 steps of manual gameplay where the sensor input and player output were saved for each step. This dataset was used to train the supervised ESNs.

A lot of the performance of ESNs trained using supervised learning depend on the chosen parameters (leaking rate and spectral radius). For this reason, the parameters had to be optimized before the supervised learning ESNs were compared against the other methods. As mentioned earlier, SEVANN was elected to evolve the parameters of the ESNs. Each individual was created using the evolved parameters, and was then trained using linear regression. After training, the individual was tested using the rest of the data from the dataset and given an error-fitness according to its test-error. The individuals with the best error-fitness were then tested using the Frogger-fitness measure described in Section 3.2.5 to compare them with the rest of the methods.

4.2 Experimental Setup

This section describes the experimental setup used to perform the experiment, with the intention that the reader should be able to re-create the experiment.

4.2.1 Fitness and Game Setup

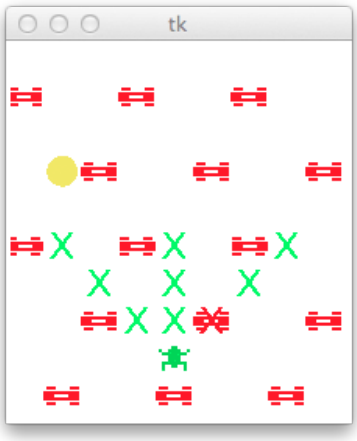
The parameter setup for the fitness function used in the experiment is shown in Table 4.1. In addition to the parameters in the table, the maximum value of the death score was set to 100 to avoid deaths taking over the fitness entirely, and to give partial credit to networks that moved in the correct direction when dying a lot. Every time the network was given new input, one game-step was executed. All the cars were configured to move every other game-step. The script used to configure the game is shown in Figure 4.1a, and a screenshot of the game used is shown in Figure 4.1b.

Fitness Parameters	
Pickup weight, w_P	50
Death weight, w_D	1
Cone weight, w_C	0
Moved towards weight, w_{M_t}	0.1
Moved away weight, w_{M_a}	0.05
Pickup-to-death ratio weight, w_{PDR}	100
Pickup-to-death threshold, T	50

Table 4.1: **Fitness parameters for experiment.** Parameters for the fitness function used in the experiment.

```
Grid:
9,10,24
Lanes:
1,3,0,2,1
3,3,0,2,6
5,3,1,2,5
7,3,0,2,3
9,3,0,2,8
Frog:
4,8
```

(a) Script used in Figure 4.1b



(b) The game board used.

Figure 4.1: **Game board used for experiments.** Figure a shows the script used to create the game board shown in Figure b. In this configuration, Only the middle lane has a different direction than the rest, and all the cars move every other game step. This is the game configuration used to measure the different methods of controlling Frogger.

4.2.2 Best Method for Controlling Frogger

When evolving ANNs for the experiment, a standard evolutionary algorithm which is part of SEVANN was applied. The parameters of the EA is shown in

EA Parameters	
Num generations	250
Population size	50
Crossover points	7
Mutation rate	0.025

Table 4.2: **EA Parameters for experiment.** Parameters of the evolutionary algorithm applied in the experiment.

	ANN Setup		
Parameter	Evo ESNs	SL ESNs	FF ANNs
Input nodes	10	10	10
Bias nodes	1	1	1
Hidden nodes	30	30	1 - 30
Output nodes	4	4	4
Spectral Radius	Evolved	Evolved	N/A
Spectral Radius range	0.0 - 1.0	0.0 - 1.0	N/A
Leaking Rate	Evolved	Evolved	N/A
Leaking Rate range	0.0 - 1.0	0.0 - 1.0	N/A
Input scaling	1.0	1.0	N/A
In $\rightarrow$ Hidden connectivity	Full	Full	Full
In $\rightarrow$ Hidden weights	Random	Random	Evolved
Hidden $\rightarrow$ Hidden connectivity	Random	Random	N/A
Hidden $\rightarrow$ Hidden weights	Random	Random	N/A
Hidden $\rightarrow$ Outputs connectivity	Full	Full	Full
Hidden $\rightarrow$ Outputs weights	Evolved	Trained	Evolved

Table 4.3: **ANN Parameters for experiment.** The parameters used for the different ANNs evolved during the experiment. Unless a parameter was evolved, it was fixed during the run of the EA. The leaking rate and the spectral radius of the ESNs were evolved, but the input scaling was fixed. The output weights for the evolved ESNs were evolved, while they for the supervised learning ESNs were trained using linear regression. For the ESNs, 30 reservoir nodes were chosen since this provided a good compromise between fitness performance and runtime speed. As for the feed-forward ANNs, the EA evolved the amount of nodes needed in one hidden layer. The weights between all evolved links were evolved with values between -1.0 and 1.0.

Table 4.2. The complete parameter setup for the ANNs is shown in Table 4.3. SEVANN uses the concept of an *epoch* to describe one complete run through a

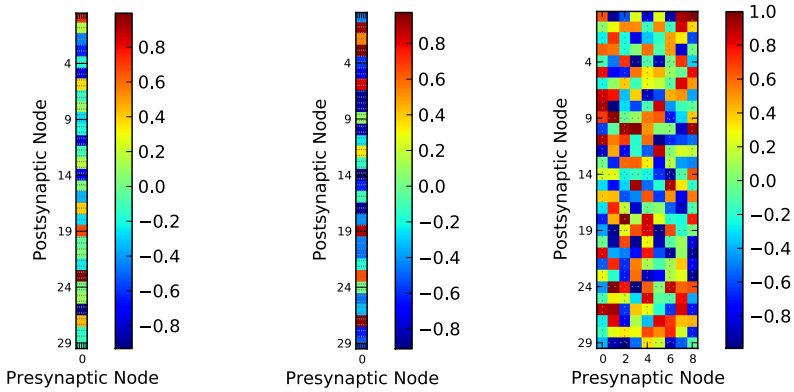
set of training data. Since ESNs require initial runs, the ESNs were run for 1025 simulation steps per epoch, where 25 of the steps were initialization steps. The feed-forward ANNs were run for 1000 simulation steps per epoch. To measure fitness both with fixed coin positions and random coin positions, each individual was run for 11 epochs. In the first epoch the coin locations were fixed according to 4 different locations on the game board. While in the final 10, the coin position was random. After each epoch, the fitness was measured, and both the individual and the game board were reset for the next epoch. In total, each individual was run for 11000 time steps (11275 steps counting the initialization steps for evolved ESNs.). The final fitness for the individual was the average fitness obtained over the 11 epochs of running.

With this setup, each evolutionary run of 250 generations took between 18 to 30 hours to complete. For all tested methods, a total of 30 runs were completed.

In order to find the absolute best method for controlling Frogger, the five best results from each method were compared by running them through 100 epochs with random coin positions. The fitness, amount of pickups and amount of deaths were then averaged over the 100 epochs, and the best result from each method were compared to each other, using the hard-coded heuristic performance as a baseline.

4.2.3 ESN Setup

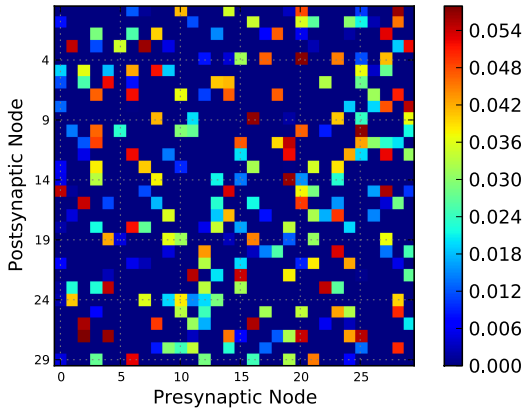
Since the random seed used to generate the ESNs was kept equal across the experiment, the ESNs generated are essentially the same network every time. This gives both the evolved ESNs and the supervised learning ESNs a common ground to start with. The network weights generated are shown in Figure 4.2. Do note that the static weights only are contained within the input-to-reservoir and reservoir-to-reservoir links. The reservoir-to-output weights are the ones that change during evolutionary learning or linear regression training.



(a) Weights from the bias node to the reservoir

(b) Weights from the cone sensor input to the reservoir

(c) Weights from the visual sensor input to the reservoir



(d) Weights inside the reservoir.

Figure 4.2: **ESN setup for experiments.** Due to the use of random seeding, these weights were generated in the same manner across the experiment. Note that the size of the weights inside the reservoir (Figure d) were scaled with the spectral radius, this means that although the weight distribution was equal in all runs, the exact size of the individual weights were not.

4.2.4 Supervised Learning ESNs

When evolving the parameters for the supervised learning ESNs, the fitness function was defined as $1000 - error$, where *error* is the amount of times the network output differed from the expected output. The assumption was that if an ESN trained with supervised learning has a low error on the dataset, it will also perform well in other unseen situations.

During evolution, each individual used 80% of the dataset as training, while the remaining 20% was used to measure the testing error. This means that since the dataset contained 2000 steps, the maximum possible error was 400. Ridge regression was applied when calculating the output weights. Since most of the runs here converged quite quickly toward an optimal fitness, the supervised ESN evolutionary runs were only run for 10 generations each, as running for more generations did not seem to make the fitness any better.

Unlike the setup for the evolved ESNs and the feed-forward ANNs, all of the runs using supervised learning were run for the 100 epochs with random coin positions. In addition, each run was also tested for 1 epoch with fixed coins. The reason all of the runs were tested this way was to test the assumption that a low dataset-error also leads to good general performance.

4.3 Experimental Results

This section presents the results obtained from the experiment. Out of the methods tested to control Frogger, evolved ESNs come out clearly on top. While the other methods tested were able to grasp the task of picking up coins, none were able to match the evolved ESNs when trying to balance the task of picking up coins with the task of avoiding the cars. The next Sections describe method-specific results, before summarizing the results and comparing them with the hard-coded heuristic.

4.3.1 Evolved ESNs

The evolved ESNs seem very well suited for the task of controlling Frogger. Out of the 30 runs conducted, the best obtained fitness was 11534.27, while the lowest obtained fitness was 4731.75. All of the runs conducted were able to find a solution where the frog on average picked up more coins than the amount of times it died. The network with the best found fitness behaved almost perfectly,

with the frog almost never dying while consistently picking up around 65 coins per epoch.

Evolved ESN performance statistics	
Number of runs	30
Best fitness	11534.27
Worst fitness	4731.75
Average fitness	8535.90
St. dev. Fitness	2454.42
Average coin pickups fixed coins	61.80
St. dev. coin pickups fixed coins	10.19
Average deaths fixed coins	0.03
St. dev. deaths fixed coins	0.18
Average coin pickups per epoch random coins	69.38
St. dev. coin pickups per epoch random coins	11.39
Average deaths per epoch random coins	8.36
St. dev. deaths per epoch random coins	12.55

Table 4.4: **Run statistics for evolved ESNs.** These statistics are across all runs with evolved ESNs. The averages are calculated using the best individual from each run.

The solutions found with evolved ESNs were split among runs that ended up stuck at a local maxima of around 80 coin pickups and around 20 deaths per epoch, and runs that picked up slightly fewer coins (around 65) while almost never dying. The runs that got stuck with around 20 deaths per epoch consistently had a better fitness-score with fixed coins than with random coins, while the ones that almost never died consistently had worse fitness with fixed coins than with random coins. These results suggest that a solution that performs well with fixed coins learns a fixed movement pattern since it knows where to expect the next coin. This in turn seem to make it less suited for situations with random coin positions. However, looking at visualizations of these runs, this assumption is hard to confirm.

The results were also split among the networks which used all four available outputs, and those that only employed three of them to solve the task. The effectiveness of either approach are not clear, as well behaving networks both consisted of those using all available outputs, and those only using three. None of the found solutions utilized less than three outputs.

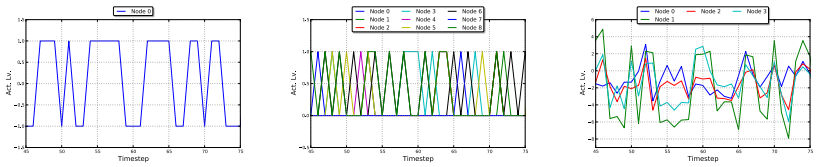
In regards to the parameters evolved, there did not seem to be a correlation between a given set of parameters and the performance of the network. Interest-

ingly, some of the well behaving networks were evolved with a leaking rate of 0.0, thereby skipping the reservoir entirely and only relying on the direct input-to-output connections to make decisions. This was not a consistent trend however, as some other very well performing networks seemed to have a rather high leaking rate as well. The same trend could be seen with regards to the spectral radius, with no specific value giving optimal results. The optimal parameters of the reservoir seem to be intertwined with the chosen output weights.

The fitness plot of the best evolved ESN is shown in Figure 4.4, and fitness plots of four other well performing evolved ESNs are shown in Figure 4.5. A summary of the evolved ESN runs are shown in Table 4.4. The five fittest evolved ESNs were run for 100 epochs with random coin positions. The best performing network after 100 epochs had an average fitness of 11919.85 with a standard deviation of 813.46. This network picked up an average of 66.79 coins with a standard deviation of 4.43, while dying an average of 0.03 times with a standard deviation of 0.17 over the course of the 100 epochs.

Visualization of the best runs showed that the frog had figured out both the task of moving towards the coin, and the task of avoiding the cars well. The frog was in most cases very careful when moving through lanes with traffic, and if the coin was on a trafficked lane, the frog always made sure to move directly out of the lane after having picked up the coin. The frog was also careful not to step directly into oncoming traffic, and in some instances waited for the traffic to pass before moving through a trafficked lane. For a video of the visualization of the best found evolved ESN, the reader is referred to `http://youtu.be/2uP7J8wkfP8`.

Probing the evolved ESNs to analyze their behavior proved to be a rather chaotic task, and no direct patterns separating good behavior from bad were found. The only correlation that was easy to spot was the correlation between the value of the cone-sensor and the value of the forward-sensor in the outputs. A few simulation steps from the probing of a well behaving ESN is shown in Figure 4.3.



(a) Cone sensor. (b) Vision sensors. (c) Network output.

Figure 4.3: **Probing of a well behaving ESN.** These are the inputs and outputs of a well behaving ESN. The y-axis is activation level, while the x-axis is time steps. In figure c, the mapping towards the frog-commands are as follows: Node 0 is mapped to *forward*, node 1 is mapped to *left*, node 2 is mapped to *right*, and node 3 is mapped to *wait*. As can be seen from the graphs, it is hard to determine exactly which inputs cause certain behavior. The only real conclusion to be drawn from the probing is that there seems to be a clear correlation between the value of the cone sensor and the value of node 0 of the output nodes. This is the node which is mapped to the frog moving forward. This observation matches the expected behavior we are looking for. The internal reservoir activation levels are not included in this graph, as they are too chaotic to analyze.

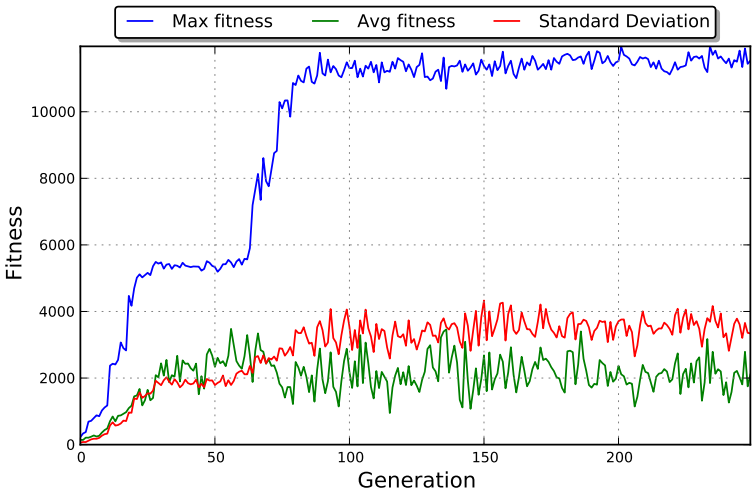
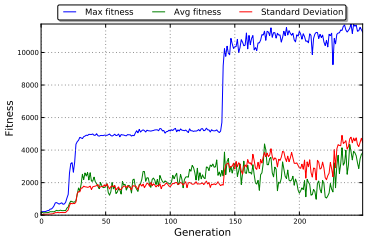
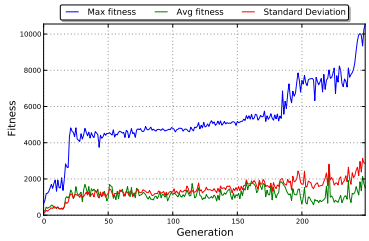


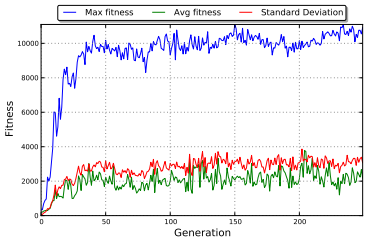
Figure 4.4: **Fitness plot of the best evolved ESN.** This plot shows the best solution found by evolving ESNs. The final fitness after 250 generations was 11534. The leaking rate was 0.92 and the spectral radius was 0.53



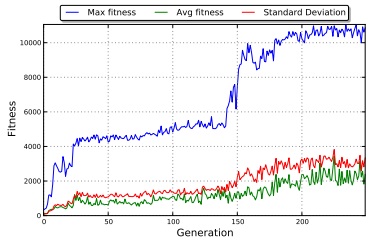
(a) Final fitness 11251, leaking rate 0.0, spectral radius 0.28.



(b) Final fitness 11094, leaking rate 0.97, spectral radius 0.24.



(c) Final fitness 10918, leaking rate 0.98, spectral radius 0.66.



(d) Final fitness 10853, leaking rate 0.785, spectral radius 0.48.

Figure 4.5: **Fitness plots of other evolved ESN runs.** These figures are the four best performing evolved ESNs after the run shown in Figure 4.4

4.3.2 Feed-Forward ANNs

The evolved feed-forward ANNs seem less suited than evolved ESNs for the task of controlling Frogger. Out of the 30 runs conducted, the best obtained fitness was 5282.43, and the lowest obtained fitness was 902.27. While all of the runs found a solution where the frog seemed to grasp the concept of picking up the coin, none of the solutions found by feed-forward ANNs were able to correctly balance the task of picking up coins with the task of avoiding the cars. The fittest evolved feed-forward ANN only slightly beat the worst found evolved ESN solution. The majority of the evolved feed-forward ANNs ended up with between 10 and 20 hidden nodes.

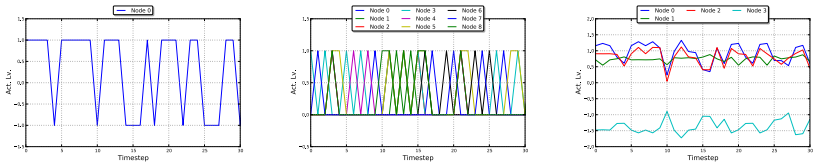
Unlike the evolved ESNs, none of the feed-forward ANNs were able to obtain a better fitness with random coins than with fixed coins. This suggests that the feed-forward ANNs are unable to generalize the solution found to fit the problem of random coin positions. Also unlike the evolved ESNs, the solutions found by evolved feed-forward ANNs never utilized more than three of the available output nodes, and some of them even went as far as to only utilize two. The best performing runs with the feed-forward ANNs resembled the evolved ESN runs that got stuck in the local maxima of 80 pickups with 20 deaths.

Feed-forward ANN performance statistics	
Number of runs	30
Best fitness	5282.43
Worst fitness	902.27
Average fitness	2615.39
St. dev. fitness	1290.70
Average coin pickups fixed coins	77.93
St. dev. coin pickups fixed coins	17.24
Average deaths fixed coins	37.67
St. dev. deaths fixed coins	27.98
Average coin pickups per epoch random coins	41.80
St. dev. coin pickups per epoch random coins	30.18
Average deaths per epoch random coins	52.82
St. dev. deaths per epoch random coins	29.66

Table 4.5: **Run statistics for feed-forward ANNs.** These statistics are across all runs with feed-forward ANNs. The averages are calculated using the best individual from each run.

The fitness plot of the best evolved feed-forward ANN is shown in Figure 4.7, and fitness plots of four other well performing feed-forward ANN runs are shown

in Figure 4.8. A summary of the evolved feed-forward ANN runs are shown in Table 4.5. As with the evolved ESNs, the five fittest evolved feed-forward ANNs were run for 100 epochs with random coin positions. The best performing network after 100 epochs had an average fitness of 4979.27 with a standard deviation of 309.98. This network picked up an average of 85.98 coins with a standard deviation of 5.42, while dying an average of 21.91 times with a standard deviation of 4.08. These statistics bear resemblance to the ESNs that got stuck in a local maxima.



(a) Cone sensor. (b) Vision sensors. (c) Network output.

Figure 4.6: **Probing of a well behaving feed-forward ANN.** These are the inputs and outputs of a well behaving feed-forward ANN. The y-axis is activation level, while the x-axis is time steps. In figure c, the mapping towards the frog-commands are as follows: Node 0 is mapped to *forward*, node 1 is mapped to *left*, node 2 is mapped to *right*, and node 3 is mapped to *wait*. As with the ESN probing, it is hard to determine what kind of behavior is triggered by the vision sensors. The two distinct behaviors seen are the same forward movement seen in the evolved ESNs if the cone sensor is *true*, as well as the ANN completely ignoring the *wait*-sensor. Otherwise, probing the ANN to view its behavior is too chaotic a task.

Visualizing the best feed-forward ANN clearly showed that the network was lacking the memory exhibited by the evolved ESNs. The network was clearly struggling with the fact that it has no concept of state, so it was unable to realize that cars were moving towards the frog. It also seemed less careful than the evolved ESNs, and more focused on getting to the coin quickly even if it meant dying in the process. The behavior exhibited by the best feed-forward ANN was comparable to the behavior of the evolved ESNs that got stuck in local maximas. For a video of the visualization of the best found evolved feed-forward ANN, the reader is referred to `http://youtu.be/fcA5-ifXpuE`.

As with the evolved ESNs, probing the evolved feed-forward ANNs proved too chaotic to give a good picture of which kinds of network structures resulted in good behavior. A few simulation steps from the probing of a well behaving feed-forward ANN is shown in Figure 4.6.

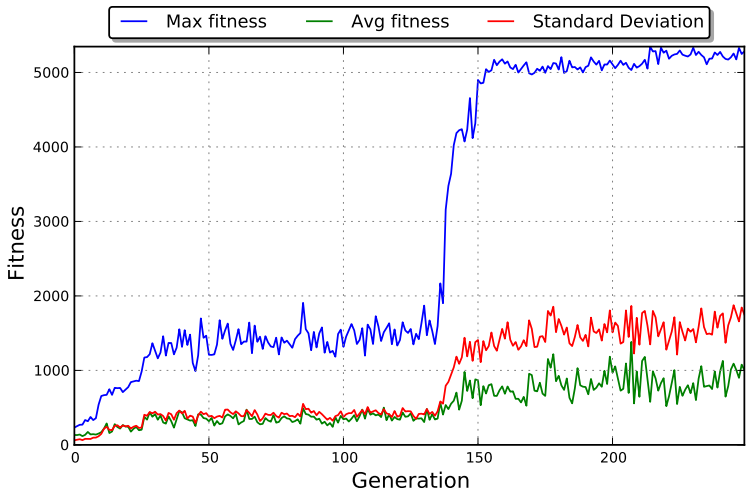
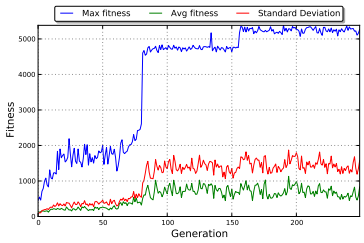
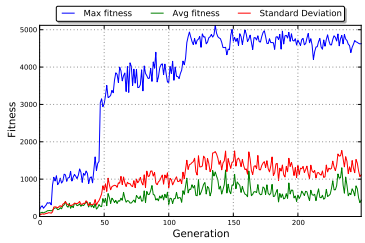


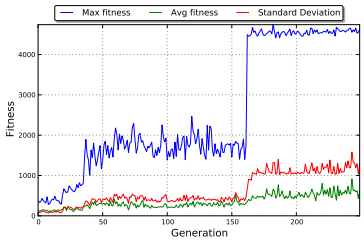
Figure 4.7: **Fitness plot of the best evolved feed-forward ANN.** This plot shows the best solution found by evolving feed-forward ANNs. The final fitness after 250 generations was 5282.43. The amount of nodes in the hidden layer were 5.



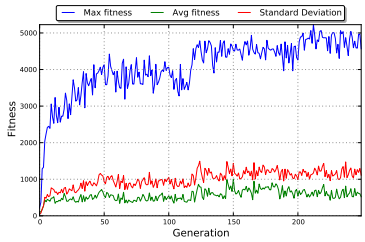
(a) Final fitness 5262.57, hidden nodes 11.



(b) Final fitness 4627.55, hidden nodes 7.



(c) Final fitness 4617.67, hidden nodes 13.



(d) Final fitness 4565.55, hidden nodes 16.

Figure 4.8: **Fitness plots of other feed-forward ANN runs.** These figures are the four best performing evolved feed-forward ANNs after the run shown in Figure 4.7

4.3.3 Supervised Learning ESNs

Supervised learning ESN performance statistics	
Number of runs	30
Best error-fitness	919
Worst error-fitness	917
Average error-fitness	918.36
St. dev. error-fitness	0.71
Best Frogger-fitness fixed coins	6885.15
Worst Frogger-fitness fixed coins	301.25
Average Frogger-fitness fixed coins	4919.54
St. dev. Frogger-fitness fixed coins	972.19
Average coin pickups fixed coins	83.33
St. dev. coin pickups fixed coins	16.99
Average deaths fixed coins	19.97
St. dev. deaths fixed coins	5.48
Best Frogger-fitness random coins	3734.91
Worst Frogger-fitness random coins	1112.06
Average Frogger-fitness random coins	2434.39
St. dev. Frogger-fitness random coins	617.60
Average coin pickups per epoch random coins	40.26
St. dev. coin pickups per epoch random coins	10.72
Average deaths per epoch random coins	25.88
St. dev. deaths per epoch random coins	10.59

Table 4.6: **Run statistics for supervised learning ESNs.** These statistics are across all runs with supervised learning. The error-fitness is the fitness function described in Section 4.2.4, while the Frogger-fitness is the fitness function described in Section 3.2.5.

As mentioned in Section 4.2.4 the supervised learning ESN runs converged quite quickly towards an optimal error-fitness. However, as can be seen from Table 4.6 this did not always lead to optimal performance when measured with the Frogger-fitness. While most of the networks seemed to get the idea of picking up coins, none were competitive with the evolved ESNs in regards to dying as little as possible.

Interestingly, the runs with the worst error-fitness after 10 generations proved to have the best performance during the 100 random coin epochs. This could mean that an optimal error with linear regression leads to overfitting to the dataset used for supervised learning, and thereby leading to less generalization towards

unseen situations.

Like the feed-forward ANNs, the supervised learning ESNs consistently had better Frogger-fitness with fixed coins than with random coins. Again, this suggests that the networks were unable to generalize to unseen situations, instead learning fixed patterns when moving towards the coin. Visualization of the best performing network showed that it seemed to struggle with balancing the task of picking up coins with the task of avoiding cars, in most cases being too aggressive in moving towards the coin.

The individual performing best with random coin positions had an error-fitness of 917 after the 10 generations of running, and a Frogger-fitness of 3734.92 after 100 epochs with random coins. It picked up 62.75 coins per epoch with a standard deviation of 22.21, while dying 23.64 times with a standard deviation of 11.02. Visualizing this frog with random coin positions show that it has indeed grasped the concept of turning and moving towards the coin, but it does not exhibit the same carefulness that the evolved ESNs show when moving towards the coin. In some of the visualization runs it also got stuck trying to move directly towards the coin without caring about the cars in its way. For a video of the visualization of the best found evolved feed-forward ANN, the reader is referred to `http://youtu.be/KLvoBuVoiEo`.

4.3.4 Method Comparison

	Best run method comparison			
	Heuristic	Evo ESNs	SL ESNs	FF ANNs
Fixed coins fitness	3681	7149	5325	7594
Average random coins fitness	2910.87	11919.85	3734.92	4979.27
St. dev. random coins fitness	413.63	813.46	1304.00	309.98
Average coin pickups per epoch	51.62	66.79	62.75	85.98
St. dev. coin pickups per epoch	6.88	4.43	22.21	5.42
Average deaths per epoch	42.03	0.03	23.64	21.91
St. dev. deaths per epoch	7.51	0.17	11.03	4.08

Table 4.7: **Best run method comparison.** This table presents the method comparisons over 100 epochs running with random coin positions. Each epoch was 1000 simulation steps long. This table presents the best performing solution for each method.

The best performing 100 epoch network from each of the methods are compared to determine the best method for controlling Frogger. The results are presented

in Table 4.7. As can be seen from the table, the evolved ESNs come out on top with random coin positions, but are beat by the feed-forward ANNs with fixed coin positions. With fixed coin positions the table is somewhat even, but with random coin positions the evolved ESNs are unmatched by any of the other methods tested. The best feed-forward ANN picked up the most coins per epoch, but did so at the expense of dying a lot more than the best evolved ESN. Worth noting is that all the methods tested beat the heuristic by a fair margin. This suggests that there are intricacies in this task that is difficult for humans to effectively program, and an optimization procedure such as an EA seem better fit for this task. Looking at this comparison, determining a winner is a rather easy task, as none of the other methods were comparatively close to the best evolved ESN when using random coin positions. For a different comparison of the performance, Figure 4.9 shows the average fitness over each generation during the evolved ESN runs and the evolved feed-forward ANN runs.

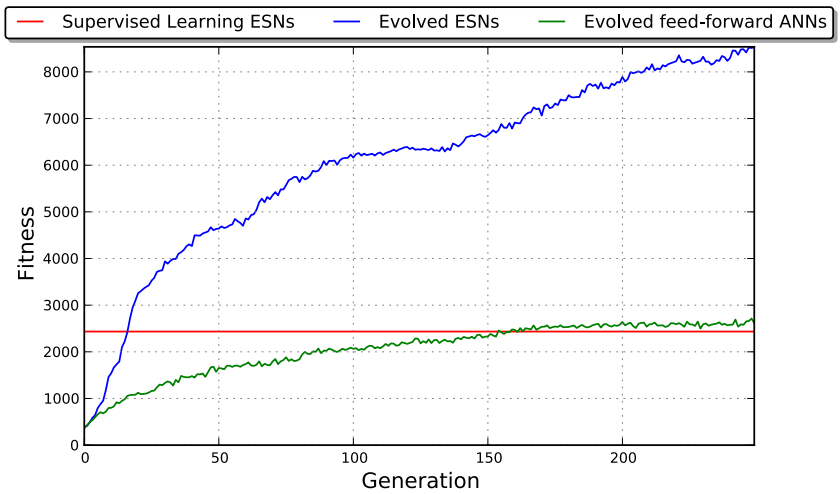


Figure 4.9: **Average fitness across evolved runs.** This plot shows the average fitness between the best individuals for every generation in all the evolutionary runs done during the experiment. As can be seen from the graph, the evolved ESNs quickly gain an advantage in average fitness, and are never challenged by the feed-forward ANNs. The average fitness from the 100 epoch runs of the supervised learning ESNs were added for comparison. On average, the evolved ESNs beat the supervised learning ESNs after about 20 generations, while the feed-forward ANNs beat the supervised learning ESNs after about 160 generations. Given the small difference in the average fitness between feed-forward ANNs and supervised learning ESNs, they seem to be about equally matched at this task.

Chapter 5

Evaluation and Conclusion

This Chapter evaluates the results, provides discussion regarding the project as a whole, and finally concludes the project with suggestions for future work.

5.1 Evaluation

Looking back at the goal and research questions presented in Section 1.2, the main goal for the project was to investigate whether or not a combination of ESNs and EAs would be a good fit for minimally cognitive unsupervised learning tasks, such as the Frogger agent implemented for this project. In order to achieve this goal, the ESN specification was implemented by means of subclassing SEVANNs already existing ANN implementation, and evolved ESNs were then used in an experiment where they were compared against a hard-coded heuristic, evolved feed-forward ANNs, and supervised learning ESNs trained with linear regression. In addition to the main goal of the project, two research questions were posed, and the conducted experiment was intended to answer these questions.

The first research question asked which of the aforementioned methods were best at controlling Frogger. Out of the methods tested, the results clearly show that evolved ESNs seem most fit for this task. Evolved ESNs beat the other methods by a rather wide margin when using random coin positions. The results show that the methods are more evenly matched with fixed coin positions, but since

the preferred solution was one that performed well with random coin positions, evolved ESNs clearly come out on top.

The second research question asked whether or not fully evolved ESNs will yield a better performance than regular ESNs trained using linear regression. As the results from the experiment shows, evolved ESNs seem clearly superior to supervised learning ESNs in this task. While the results suggest that evolved ESNs are more suited for motor control tasks than supervised learning ESNs, it is hard to draw a conclusion as to whether or not this is actually the case. The experiment done in this thesis only gives a performance measure of a single motor control task, and more tasks are required before one can conclusively decide which of the two approaches are best.

In summary, the results from the experiment conducted during the project support the research questions posed, and the implementation done fulfills the overall goal for the project.

5.2 Discussion

This section discusses the merits and limitations of the work conducted during the project.

5.2.1 ESNs, EAs, and Unsupervised Learning

The work discussed in this thesis shines further light on the relationship between ESNs, EAs, and unsupervised learning. Not only has it been shown that is it possible to modify the ESN specification for use with EAs, but as the experiment conducted showed, it can produce better results than conventional methods for certain tasks, such as the minimally cognitive task tested. These results confirm the observations reported by previous work when testing the combination of ESNs and EAs.

In addition to the work done with ESNs, this thesis also introduced a novel learning task for research in minimally cognitive behavior. The modified version of the Frogger-videogame introduced interesting minimally cognitive behavior, and it was easy to observe that the ANNs which had a sense of their current and previous states behaved better in the environment than those without such state-sense. Even though the original task of controlling Frogger is a relatively simple one for humans given our visual access to the entire game board, the evolved ESNs managed to adequately solve the task with only a fraction of the

information available through the cone- and vision sensors. In addition, the hard-coded heuristic, meant to provide an example of how a human could solve the task given the available sensory information, did not provide comparable performance to the evolved ESNs. This suggests that the frogger-game with the sensory equipment available is better suited for computer learning than a hard-coded human approach.

As shown by the experimental results, some sort of state-sense seems to be required to adequately solve the task of both picking up coins and avoiding cars. Visualization of the runs conducted during the experiment, showed that the evolved ESNs were able to pick up on details such as moving directly out of a trafficked lane, or predicting that a car was going to move into a square in the next time step. The feed-forward ANNs however, lacking a sense of state, did not exhibit the same behavior, and were sometimes run over in trafficked lanes when looking for the coin. The supervised learning ESNs did show some sense of state, but they were unable to correctly balance both tasks, instead opting for either too much carefulness or too much aggressiveness when moving towards the coin.

Even though the results are promising for the use of evolved ESNs, the work done in this thesis is limited in several ways. First of all, the experiment conducted in this thesis only tested one minimally cognitive learning task. In order to determine whether or not ESNs in general are a good fit for such tasks, one would need to test them with other pre-existing minimally cognitive tasks, such as object avoidance or object catching.

The dataset used for the supervised learning ESNs was created by hand. This means that there are bound to be errors in it, which again may affect the quality of the learning done by the supervised learning ESNs. This means that the results reported in this thesis may not accurately reflect the performance of ESNs trained with linear regression.

Finally, since this project was done by a single student, the amount of computer resources available were limited. This limitation means that there was an upper limit on the amount of runs that could be done for comparison, both in regards to the complexity of the networks tested, and the longevity of the evolutionary runs. Just as bigger ESN reservoirs may have provided better results, so could a bigger feed-forward ANN hidden layer. With unlimited computer resources, feed-forward ANNs with several hidden layers could also have been tested.

In summary, the work done in this thesis further shows that ESNs coupled with EAs are a viable choice for use with unsupervised learning tasks. In addition, it has been shown that for tasks requiring memory, ESNs trained with EAs can greatly outperform regular feed-forward ANN structures. ESNs trained with EAs have also been shown to outperform ESNs trained with linear regression for the

motor control task tested.

5.2.2 SEVANN

Throughout the project, SEVANN has been used as a toolkit for evolving the ANNs used. This made for a good opportunity to assess SEVANNs usability as a general toolkit for evolving ANNs. SEVANN has the advantage in that it makes it very easy to start with evolving ANNs. The user only needs to specify a few scripts and SEVANN does the rest. With the additions made to SEVANN during the project, it is now also much easier to extend it with new types of ANNs in the future, such as CTRNNs or maybe even deep belief networks. The work conducted in this thesis has also made it easier to employ SEVANN with big evolution tasks, by thoroughly testing its usage with linux servers and long-term evolutionary runs.

While the work conducted in this project has made SEVANN a more robust tool for evolving ANNs, some work still remains before it is ready for "prime time". First of all, SEVANN is still only single threaded, and given how suitable EAs theoretically are for parallelization, it should be a priority to implement multithreading soon. Secondly, SEVANN currently only supports one rather simple kind of EA with a direct representation of the ANN weights. Future extensions of SEVANN would include implementing support for a wide variety of EAs and ESs, such as NEAT or CMA-ES.

Another extension to SEVANN would be the implementation of a better GUI, so that new users can start evolving ANNs without even having to write scripts, but rather create the ANNs directly using a GUI. Preliminary support for this exists in SEVANN today, but a future priority should be to extend and build upon this to make the product easier to use for novices.

5.3 Future Work

This Section address some issues and limitations of this report which remain unexplored or unanswered. It also tries to provide some potential starting points for further research involving EAs and ESNs.

The results from the experiments conducted in this thesis shines further light on the potential uses for ESNs. The fact that they perform so well when coupled with EAs make them a viable option for both unsupervised and reinforcement

learning tasks. However, much of the work done in this thesis can be further explored.

Future explorations may involve trying to evolve even more parameters of the ESNs, such as network size and input scaling, and also splitting the leaking rate into separate parameters for each reservoir neuron. This means adding a lot of complexity to the EA genotype, but it may lead to better performance. In addition, the performance comparison with supervised learning ESNs in this thesis is limited by the quality of the dataset used, and a better dataset may result in completely different behavior. Future work could look into how to better create datasets for use with motor control tasks, or provide a performance measure for the different linear regression methods available to ESNs. Finally, evolving ESNs which generalize well regardless of game board, randomly generating a game board for each epoch, is also left for future work.

Other extensions of this work could also involve trying different evolutionary approaches such as NEAT or CMA-ES to evolve the ESNs, and then comparing the results from these against the ones reported in this thesis, with the goal of finding an optimal approach to evolving ESNs. Comparisons with other types of ANNs are also an obvious extension. Comparing ESNs with e.g. CTRNNs may be a more evenly matched performance measure than feed-forward ANNs. In addition to just evolving ESNs, further exploration may also involve preprocessing input data through a different kind of ANN before feeding it to the ESN, or post-processing output data from ESNs via ANNs or other classifiers.

ESNs in general would also gain from better analysis of their inner workings, and as such, the research community working with RC should strive for better analysis and visualization tools directed at the inner workings of ESNs. This thesis has measured ESN performance, but not its inner behavior.

Bibliography

Beer, R. D. (1996). Toward the evolution of dynamical neural networks for minimally cognitive behavior. *From animals to animats*, 4:421–429.

Chatzidimitriou, K. C. and Mitkas, P. A. (2010). A neat way for evolving echo state networks. In *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 909–914, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Cheney, W. and Kincaid, D. (2008). *Numerical Mathematics and Computing*. International Thomson Publishing, 6th edition.

Devert, A., Bredeche, N., and Schoenauer, M. (2007). Unsupervised Learning of Echo State Networks: A case study in Artificial Embryogeny. In Monmarché, N., El-Ghazali, T., Collet, P., Schoenauer, M., and Lutton, E., editors, *Evolution Artificielle*, volume 4926, pages 278–290, Tours, France.

Downing, K. (2010). A script-based approach to evolving neural networks. In *Proceedings of Norwegian Artificial Intelligence Symposium*.

Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks*, 1:75–80.

Fan, J., Lau, R., and Miikkulainen, R. (2003). Utilizing domain knowledge in neuroevolution.

Gauci, J. and Stanley, K. O. (2010). Autonomous evolution of topographic regularities in artificial neural networks. *Neural Comput.*, 22(7):1860–1898.

Glass, L. and Mackey, M. (2010). Mackey-glass equation. 5(3):6908.

Goldberg, D. E. and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 41–49, Hillsdale, NJ, USA. L. Erlbaum Associates Inc.

Gomez, F. and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, (5):317–342.

Gomez, F. J. and Miikkulainen, R. (2003). Active guidance for a finless rocket using neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 2084–2095, San Francisco. Morgan Kaufmann.

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

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, 81(10):3088–3092.

Jaeger, H. (2001). The ''echo state'' approach to analysing and training recurrent neural networks - with an erratum note.

Jaeger, H., Lukoševičius, M., Popovici, D., and Siewert, U. (2007). Optimization and applications of echo state networks with leaky integrator neurons. *Neural Networks*, 20(3):335 – 352.

James, D. and Tucker, P. (2004). A comparative analysis of simplification and complexification in the evolution of neural network topologies.

Jiang, F., Berry, H., and Schoenauer, M. (2008). Supervised and evolutionary learning of echo state networks. In *Proceedings of the 10th international conference on Parallel Problem Solving from Nature: PPSN X*, pages 215–224, Berlin, Heidelberg. Springer-Verlag.

Lukoševičius, M. (2012). *A practical guide to applying echo state networks*, volume 7700 of *Lecture Notes in Computer Science*, pages 659–686. Springer Berlin Heidelberg, 2 edition.

Lukoševičius, M. and Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149.

Lukoševičius, M., Jaeger, H., and Schrauwen, B. (2012). Reservoir computing trends. *KI - Künstliche Intelligenz*, pages 1–7.

Maass, W., Natschlaeger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560.

Miikkulainen, R. (2005). Evolving neural networks. http://courses.cs.tamu.edu/choe/12summer/315/lectures/main.pdf.

Montana, D. J. and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th international joint conference*

on Artificial intelligence - Volume 1, IJCAI'89, pages 762–767, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Moriarty, D. E. and Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, (AI94-224):11–32.

Moriarty, D. E. and Miikkulainen, R. (1997). Forming neural networks through efficient and adaptive coevolution. *Evol. Comput.*, 5(4):373–399.

Risi, S. and Stanley, K. O. (2010). Indirectly encoding neural plasticity as a pattern of local rules. In *Proceedings of the 11th international conference on Simulation of adaptive behavior: from animals to animats*, SAB'10, pages 533–543, Berlin, Heidelberg. Springer-Verlag.

Russell, S. J. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.

Schmidhuber, J., Wierstra, D., Gagliolo, M., and Gomez, F. (2007). Training recurrent networks by evolino. *NEURAL COMPUTATION*, 19:2007.

Schrauwen, B., Verstraeten, D., and Campenhout, J. V. (2007). An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th European Symposium on Artificial Neural Networks*, pages 471–482.

Slocum, A. C., Downey, D. C., Beer, R. D., and Beer, A. D. (2000). Further experiments in the evolution of minimally cognitive behavior: From perceiving affordances to selective attention. In *In*, pages 430–439. MIT Press.

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

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

Whiteson, S. and Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917.

Wikipedia (2012a). Frogger — wikipedia, the free encyclopedia. [Online; accessed 22-November-2012].

Wikipedia (2012b). Recurrent neural network — wikipedia, the free encyclopedia. [Online; accessed 19-November-2012].

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

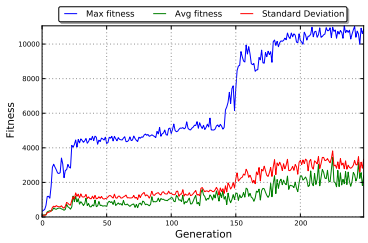
Appendices

Appendix A

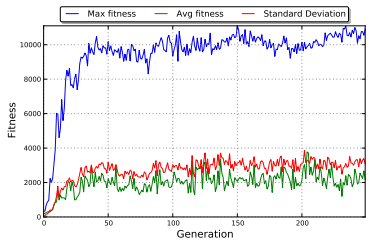
Fitness graphs

This Appendix presents all fitness graphs from the runs done during the experiments. The fitness graphs for the supervised learning ESNs are not included in this appendix, since the runs almost immediately settled on an optimal fitness, leading to an uninteresting fitness graph.

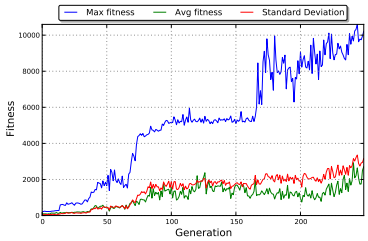
A.1 Evolved ESNs



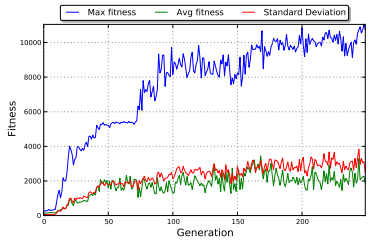
(1) Final fitness 10853, leaking rate 0.85, spectral radius 0.48.



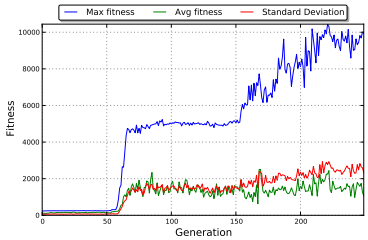
(2) Final fitness 10919, leaking rate 0.98, spectral radius 0.66.



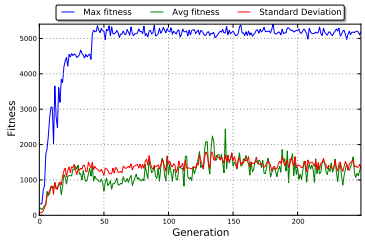
(3) Final fitness 10218, leaking rate 0.90, spectral radius 0.22.



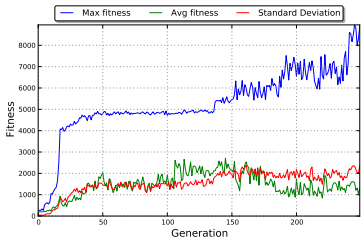
(4) Final fitness 10761, leaking rate 0.01, spectral radius 0.47.



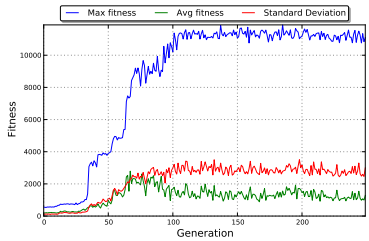
(5) Final fitness 9936, leaking rate 0.81, spectral radius 0.24.



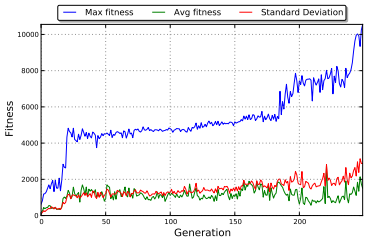
(6) Final fitness 5189, leaking rate 0.98, spectral radius 0.09.



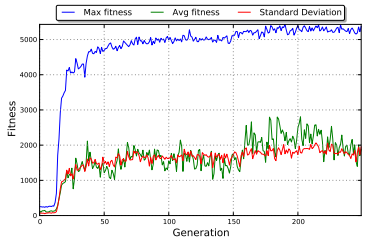
(7) Final fitness 8895, leaking rate 0.93, spectral radius 0.16.



(8) Final fitness 11094, leaking rate 0.97, spectral radius 0.24.

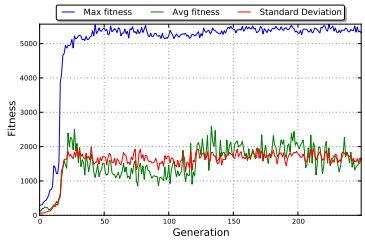


(9) Final fitness 10553, leaking rate 0.99, spectral radius 0.18.

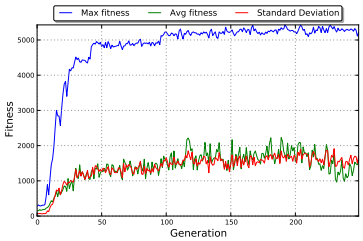


(10) Final fitness 5372, leaking rate 0.06, spectral radius 0.66.

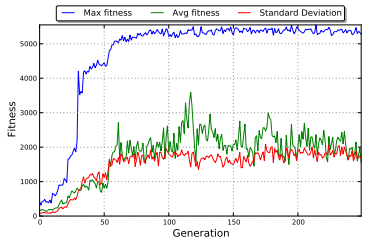
(11) Final fitness 8292, leaking rate 0.03, spectral radius 0.32.



(12) Final fitness 5309, leaking rate 0.73, spectral radius 0.08.

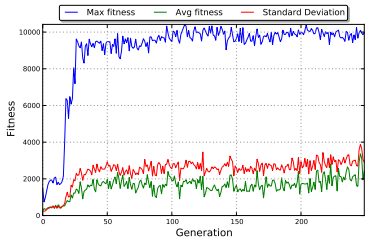


(13) Final fitness 5220, leaking rate 0.47, spectral radius 0.51.

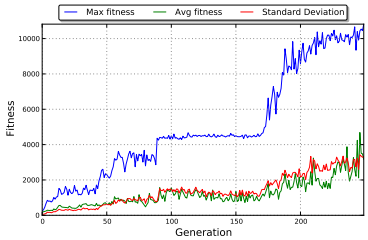


(14) Final fitness 5274, leaking rate 0.88, spectral radius 0.39.

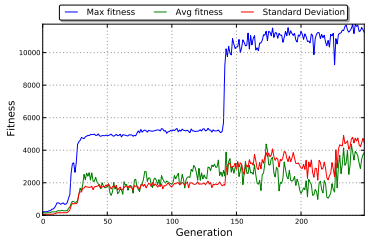
(15) Final fitness 10508, leaking rate 0.01, spectral radius 0.77.



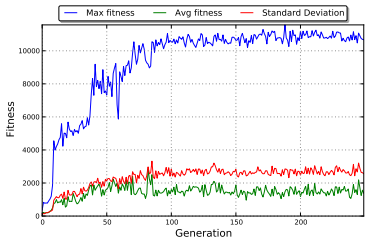
(16) Final fitness 9985, leaking rate 0.00, spectral radius 0.66.



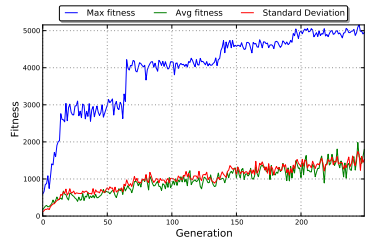
(17) Final fitness 10816, leaking rate 0.95, spectral radius 0.34.



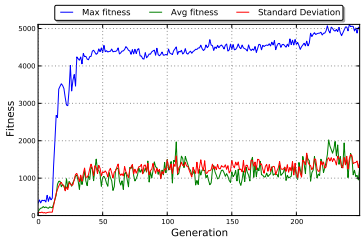
(18) Final fitness 11251, leaking rate 0.00, spectral radius 0.28.



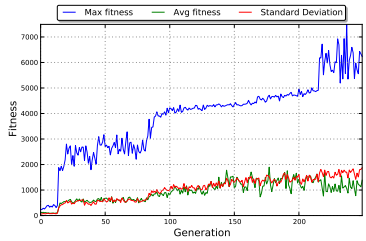
(19) Final fitness 10670, leaking rate 0.74, spectral radius 0.38.



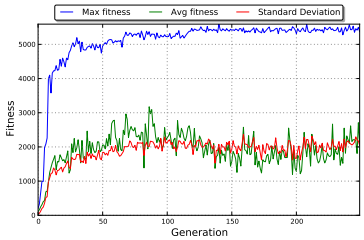
(20) Final fitness 4950, leaking rate 0.20, spectral radius 0.50.



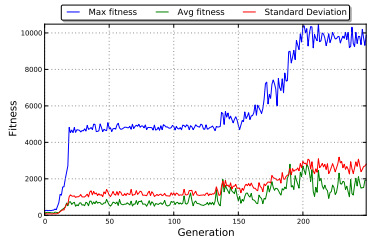
(21) Final fitness 5015, leaking rate 0.72, spectral radius 0.21.



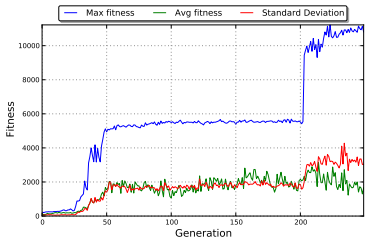
(22) Final fitness 6231, leaking rate 0.73, spectral radius 0.68.



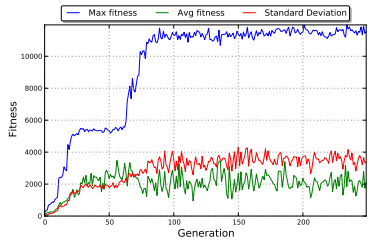
(23) Final fitness 5374, leaking rate 0.00, spectral radius 0.63.



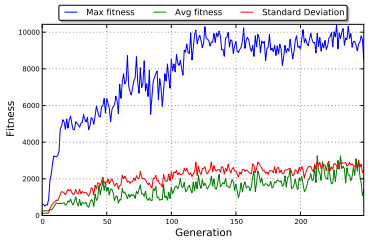
(24) Final fitness 9869, leaking rate 0.84, spectral radius 0.00.



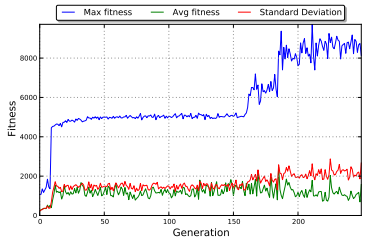
(25) Final fitness 10805, leaking rate 0.98, spectral radius 0.05.



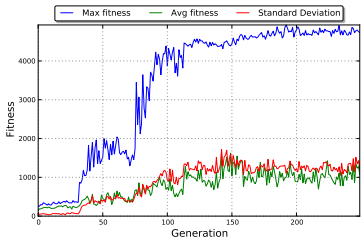
(26) Final fitness 11534, leaking rate 0.92, spectral radius 0.53.



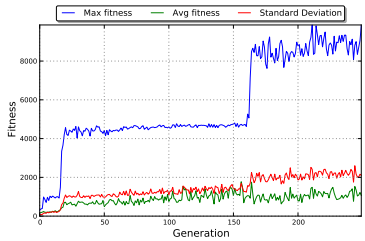
(27) Final fitness 8487, leaking rate 0.96, spectral radius 0.68.



(28) Final fitness 8138, leaking rate 0.88, spectral radius 0.22.

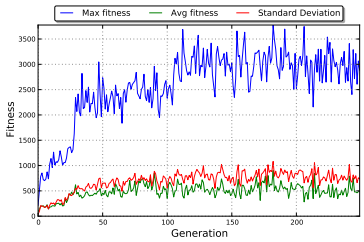


(29) Final fitness 4732, leaking rate 0.81, spectral radius 0.39.

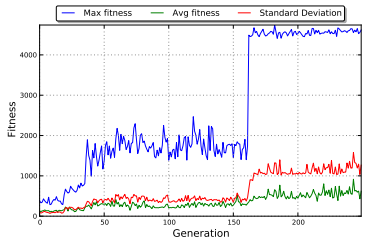


(30) Final fitness 9822, leaking rate 0.91, spectral radius 0.68.

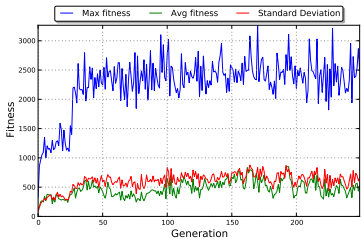
A.2 Evolved Feed-Forward ANNs



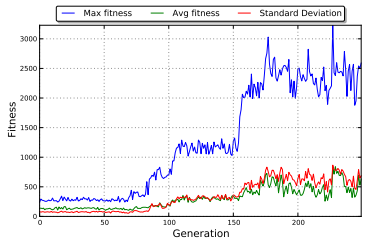
(1) Final fitness 2881, hidden nodes 16



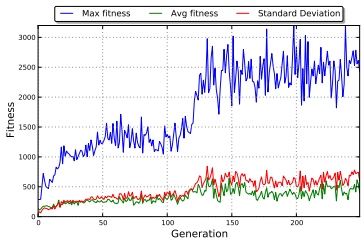
(2) Final fitness 4618, hidden nodes 13



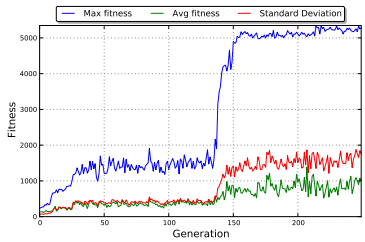
(3) Final fitness 2493, hidden nodes 22



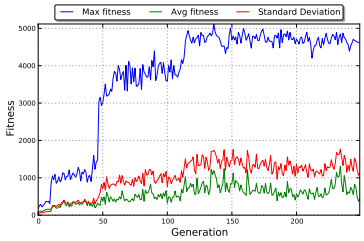
(4) Final fitness 2591, hidden nodes 18



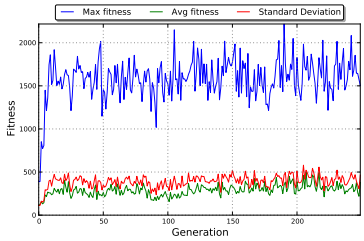
(5) Final fitness 2430, hidden nodes 21



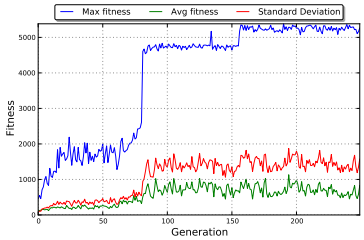
(6) Final fitness 5282, hidden nodes 5



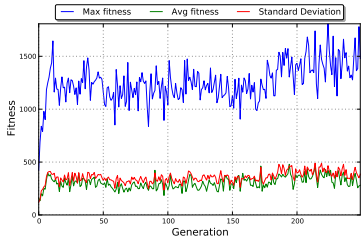
(7) Final fitness 4628, hidden nodes 7



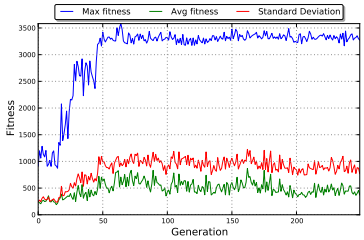
(8) Final fitness 1504, hidden nodes 13



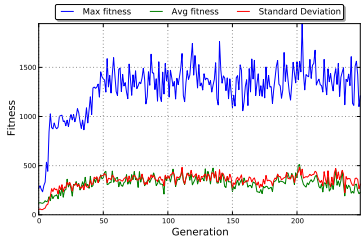
(9) Final fitness 5263, hidden nodes 11



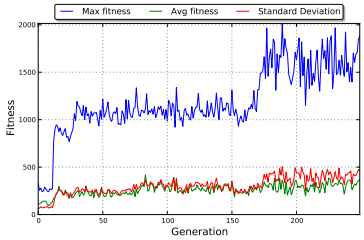
(10) Final fitness 1480, hidden nodes 4



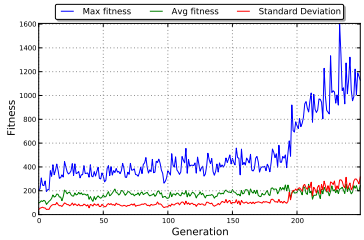
(11) Final fitness 3274, hidden nodes 16



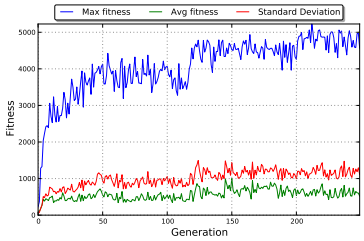
(12) Final fitness 1209, hidden nodes 12



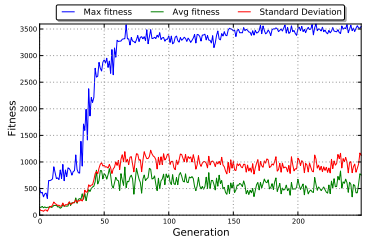
(13) Final fitness 1882, hidden nodes 16



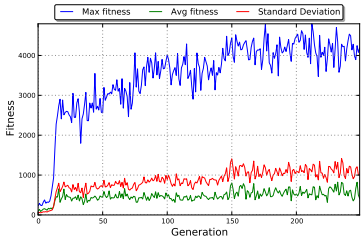
(14) Final fitness 1129, hidden nodes 12



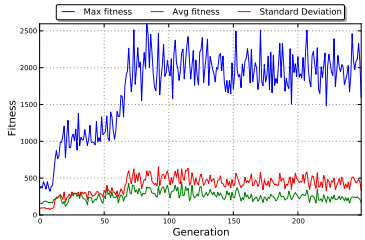
(15) Final fitness 4566, hidden nodes 16



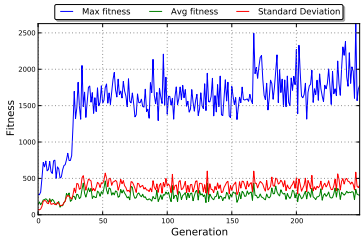
(16) Final fitness 3550, hidden nodes 12



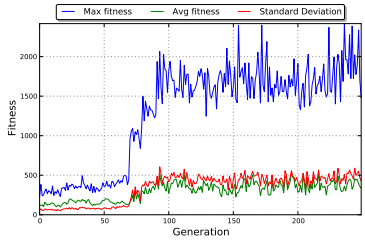
(17) Final fitness 4188, hidden nodes 10



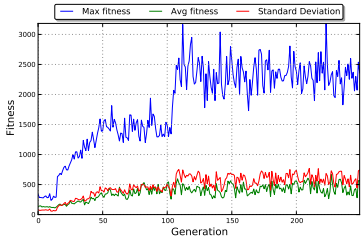
(18) Final fitness 1598, hidden nodes 18



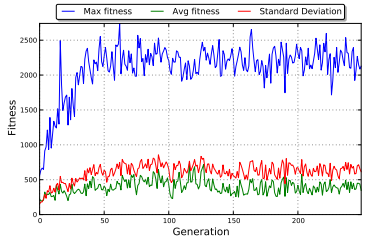
(19) Final fitness 1776, hidden nodes 17



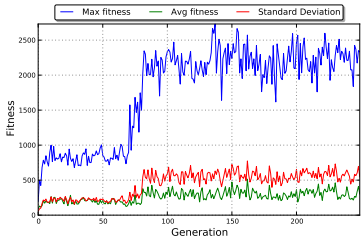
(20) Final fitness 1479, hidden nodes 15



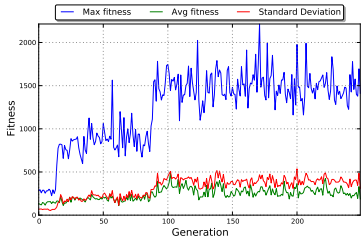
(21) Final fitness 2360, hidden nodes 16



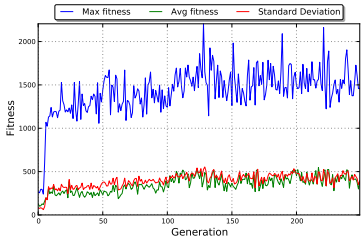
(22) Final fitness 2127, hidden nodes 8



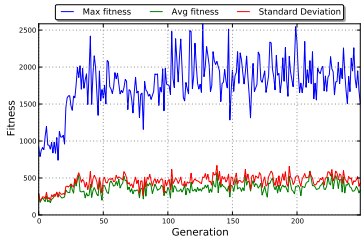
(23) Final fitness 2333, hidden nodes 12



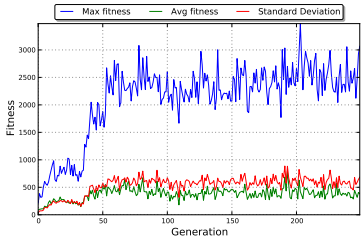
(24) Final fitness 1412, hidden nodes 23



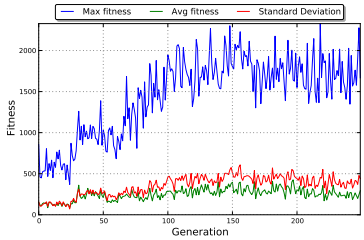
(25) Final fitness 1461, hidden nodes 19



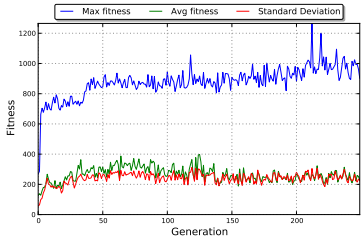
(26) Final fitness 1514, hidden nodes 16



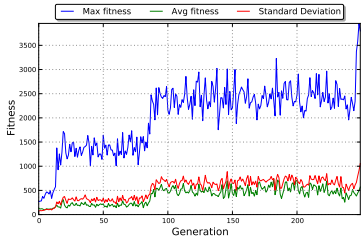
(27) Final fitness 3068, hidden nodes 19



(28) Final fitness 1668, hidden nodes 14



(29) Final fitness 902, hidden nodes 21



(30) Final fitness 3798, hidden nodes 19