KTH Computer Science
and Communication

Evolution of Artificial Brains in Simulating Animal Behaviour

Comparing radial basis, linear and random functions for decision-making

BJÖRN TEGELUND
JOHAN WIKSTRÖM

Bachelor's Thesis at CSC
Supervisor: Petter Ögren
Examiner: Mårten Björkman

TRITA xxx yyyy-nn

Abstract

This is a skeleton for KTH theses. More documentation regarding the KTH thesis class file can be found in the package documentation.

Referat

Evolution av artificiella hjärnor vid simulering av djurs beteende

Denna fil qer ett avhandlingsskelett. Mer information om LATEX-mallen finns i dokumentationen till paketet.

Contents

Chapter 1

Introduction

1.1 Background

Genetic algorithms are algorithms that emulate evolution to achieve a optimal solution to a problem[1]. These algorithms have many uses and we wanted to investigate whether the phenomena which occur in nature due to evolution also occurs when using genetic algorithms. In nature evolution has spawned a wide variety of survival strategies such as adaptation, camouflage colours and mimicry. Genetic algorithms however, are very simplified mathematical models of these mechanisms which means that these phenomena might not occur at all.

1.2 Scope and Objectives

Our main objective is to compare brains for simulating animal behaviour using radial-basis functions and linear functions. As a baseline we will also use brains which make random decisions. To train our animals we use genetic algorithms. The genetic algorithms provide the genes which will then be used in their brains to make decisions. We will look at statistics such as learning speed, execution speed and how well they can adapt to their surroundings.

The animals will be given different tasks to do in each experiment, ranging from simply gathering food to avoiding predators to mimicking things in the world to survive. In each experiment the three brain architectures are compared and contrasted. By doing these simulations we wish to find the pros and cons for both brains using radial-basis functions and linear functions, when used for artificial intelligence in this way. We also wish to find how complex of a task a brain using linear functions is able to solve, and in the cases where it is not able to solve the problem completely we wish to know how much better is it than the random brain.

[1]Proof

1.3 Achievements

Chapter 2

Technical Overview

2.1 Genetic Algorithms

2.1.1 Overview

What are genetic algorithms?
How are they typically used?
How do they relate to genetics and evolution?
Overview of algorithm

Genetic algorithms are a way of approximating solutions to NP-hard problems, a group of problems that are extremely computationally expensive. They are a form of machine learning algorithms typically used to solve problems with a large or complex search space and where other machine learning algorithms fail [3]. Genetic algorithms mimic natural selection in nature by defining a set of genomes, or individuals, where each genome represents a possible solution to the given problem. This genome is commonly stored as a string of 0:s and 1:s or as a list of floating point values. This set of genomes undergoes several iterations, or generations, of small improvements by fitness evaluation, selection , mutation and crossover. These operators all have equivalents in real evolution and eventually, the genomes will converge to a solution which represents a local minima in the search space. The advantage of genetic algorithms is that the crossover operator enables a wider search over the problem domain than many other approaches, using less calculations [4]. In the following subsections, the most crucial parts of the genetic algorithm will be explained and in figure 2.1 we depict an overview of the genetic algorithm in pseudo code.

2.1.2 Fitness

Fitness is the only means of measuring the success of the solution. In nature, fitness is simply determined by the number of offspring produced by the individual and the percentage of that offspring which survives long enough to generate new offspring. In genetic algorithms however, the fitness function can be any value that describes

```
$S \leftarrow$ a random distribution of genes
**while** the genes have not converged towards a solution **do**
    $F \leftarrow fitness(S)$            ▷ Calculate fitness values for all $s \in S$
    $X \leftarrow select(S, F)$          ▷ Get a multiset of S using fitness values
    $C \leftarrow crossover(X)$          ▷ Apply crossover operator on multiset
    $M \leftarrow mutate(C)$             ▷ Apply mutation operator on crossed genes
    $S \leftarrow M$                     ▷ Restart with the new generation
**end while**
```

Figure 2.1. An overview of the genetic algorithm used in the experiments

each individual's success at solving the problem at hand [3] and genetic algorithms can get significant performance advantages by simply selecting the correct fitness function. The fitness function should be strictly positive for all inputs and preserve some form of relative internal ordering of the individuals. In most problems there are multiple choices of fitness functions and the best choice of fitness function is unique for every problem. For this reason, we experimented with different fitness functions before settling for simply the life length of our individuals.

2.1.3 Selection

Selection is the process of selecting which organsims that should be allowed to reproduce and in what proportions. In nature, selection is closely tied to fitness, the fittest individual is also "selected" most often, but in genetic algorithms there are several different ways to model this selection process. A good fitness function should strike a balance between favorizing the fittest individuals and allowing less fit individuals to survive in a reduced number. One trivial selection function could be to simply select all individuals that pass a certain fitness threshold [3] [TROR DET STOD DÄR IAF]. This is a bad idea, however, since there are few ways to determine the correct threshold value. In the beginning of the evolution, a high threshold will exclude most of the genes due to low general fitness and this will lead to a fast reduction of genetic variation. In the later stages of the evolution all individuals will pass the threshold, rendering the selection function useless.

A better approach is the so called roulette wheel selection where each individual is mapped to an area of a "roulette wheel". A larger fitness value means a larger area on the roulette wheel and the selection function simply generates random values corresponding to the areas of this roulette wheel. In this way, the individuals are chosen based on a biased stochastic variable and there is room for individuals with high fitness to dominate as well as for individuals with low fitness to be included by chance.

Undrar om vi ska skriva något om NSGA II...

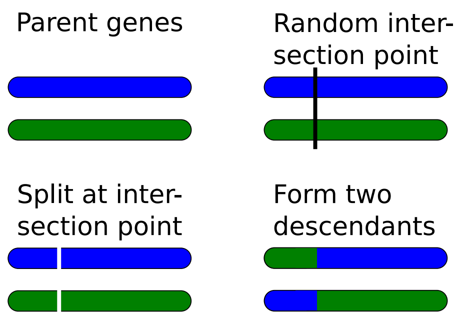

Figure 2.2. A schematic drawing showing single point crossover of two genomes.

2.1.4 Mutation

Mutation is a random, usually small, change of an individuals genome. In nature In genetic algorithms it is usually implemented as a small probability for each gene to mutate. When the genome consists of a series of 0:s and 1:s, the mutation operator is simply a flip of that bit and when the genes consist of floating point values, the mutation can be an addition or multiplication of a random value. If the mutation rate is too high, it becomes very hard to reach convergence since the good solutions are mutated into averaged solutions.

2.1.5 Crossover

Crossover is the operation of combining two genomes into two new genomes. One common crossover operator is known as single point crossover and it consists of splitting two genomes at the same position and merging the splitted parts into two new genomes. The split position is chosen randomly and the two new genomes share no genes with each other. This process is displayed in Figure 2.2 There is also multiple point crossover where there are multiple splitting positions as well as uniform crossover where each gene can be exchanged with a certain probability.

2.2 Radial Basis Functions

2.2.1 Overview

A radial basis function (RBF) is a bell-shaped function whose value depends on the distance from some origin, denoted μ in the formula. Radial basis functions are commonly used in neural networks as a way to encode input information. They are favourable to use as they have locality, something which linear functions do not. In particular they are used for function approximation, as any function can be approximated as the sum of a number of weighted radial basis functions. A

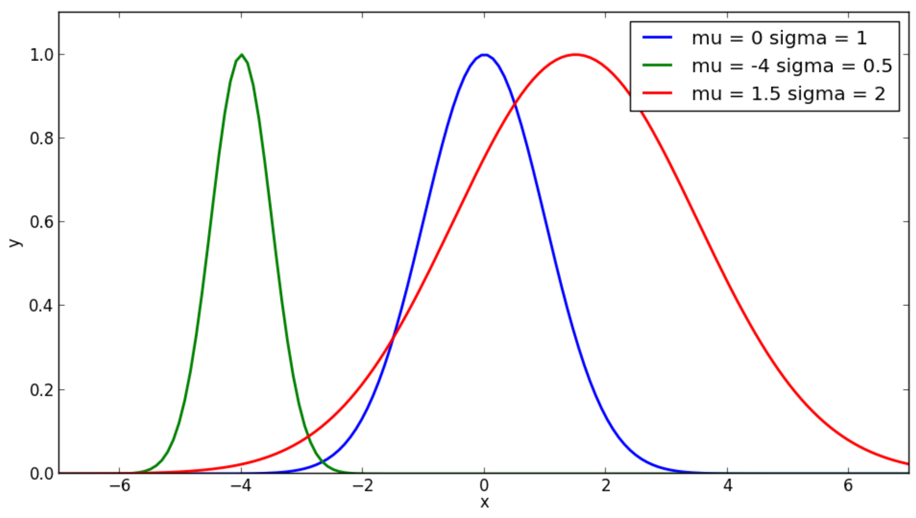

Figure 2.3. Three one-dimensional RBFs with varying μ and σ values.

$$f(x,y,z) = \exp(-(\frac{(x-\mu_x)^2}{2\sigma_x^2})) + \exp(-(\frac{(y-\mu_y)^2}{2\sigma_y^2})) + \exp(-(\frac{(z-\mu_z)^2}{2\sigma_z^2})) \quad (2.1)$$

Figure 2.4. A sum of three radial basis functions, corresponding to three input values.

property of radial basis functions which can both be interpreted as an advantage and disadvantage is that their value never exceeds a given constant, compared to a linear function which can grow to infinitely high or low numbers.

Radial basis functions are commonly implemented using a formula such as **??**, which is a three-dimensional function centred around (μ_x, μ_y, μ_z). The width of the bell-curve in each dimension is determined by σ_x, σ_y and σ_z respectively.

Chapter 3

Implementation

3.1 Model

3.1.1 Simulation in Python

We chose python as implementation language mainly because it is easy to build prototypes quickliy in that language. Python also has an abundance of third party libraries which helps reduce implementation cost significantly. We chose to use a third party library called DEAP (Distributed Evolutionary Algorithms in Python) for the evolutionary algorithms. DEAP proved to be flexible enough for the task since it allowed us to define our own selection, mutation and crossover algorithms as well as using default versions. Apart from DEAP we have also used pygame and matplotlib for graphical rendering as well as pypy, numpy and python's built in support for multiprocessing to decrease running times. For a comprehensive list of tools used, please see Appendix A.

Square world with walls, circular inhabitants with antennae
Good and bad food sources
Predators?
Possible modifications to enforce different behaviour?
What libraries will we use? Why?
Optimisations?

3.1.2 Methods of Enforcing Behaviour

Adding input
Additional terms in brain calculations
Adding predators
More kinds of food
Being able to see more things in the world
Placing objects into the world

$$\Delta r = \begin{cases} 0 & \text{if } l_4 = 0 \text{ and } r_4 = 0, \\ S(\mathbf{g_{1-3}} \circ \mathbf{x_l}) & \text{if } l_4 \neq 0 \text{ and } r_4 = 0 \\ S(\mathbf{g_{4-6}} \circ \mathbf{x_r}) & \text{if } r_4 \neq 0 \text{ and } l_4 = 0 \\ S(\mathbf{g_{1-3}} \circ \mathbf{x_l} + \mathbf{g_{4-6}} \circ \mathbf{x_r}) & \text{if } r_4 \neq 0 \text{ and } l_4 \neq 0 \end{cases} \tag{3.1}$$

Figure 3.1. The linear brain's decision formula for change of rotation Δr. S corresponds to a sigmoid function described in chapter 3.1.4.

3.1.3 Decision making

All brains in our simulation have a similar structure, they are functions with eight inputs and two outputs. When input is received by a creature, it is in form of eight numbers, four for each antenna. Three of the inputs for each antenna are the red-, green- and blue components of the currently detected object's colour. These inputs are normalized to the interval $[0, 1]$. The fourth input is zero when no object is detected and one when an object is.

The outputs consists of the values Δr and Δs which denotes change of rotation and change of speed. Both output values are normalized to $[-1, 1]$ to account for the possibilities of negative rotation and negative acceleration. These values are then translated into reasonable values in the simulation. The maximum acceleration is determined by the size of the creatures, the size of the world, the maximum speed of the creatures and so on. The maxumum allowed change in rotation is 180° or π since the interval $[-\pi, \pi]$ covers the entire circle. A larger allowed change in rotation would have made learning harder since there would be multiple correct responses to a situation, e.g $\Delta r = v, \Delta r = v + 2\pi, \Delta r = v + 4\pi \ldots$.

3.1.4 Linear Decision Making

In the linear brain we have the simplest possible artificial intelligence. We have twelve genes in total in the interval $[-1, 1]$ These genes correspond to three colors per antenna times two antennae times two outputs from the brain. Both Δr and Δs are calculated using the same method. In equation 3.1, the first three components of the left and right antennae vectors, the ones containing color data, are called $\mathbf{x_l}$ and $\mathbf{x_r}$ respectively. The fourth components are called l_4 and r_4. There are twelve genes involved in total in the linear brain, six of them applies to this equation and they are divided into two vectors $\mathbf{g_{1-3}}$ and $\mathbf{g_{4-6}}$ using genes 1-3 and 4-6. The change in speed Δs is calculated in the exact same way using the same inputs but genes 7-12 instead.

In the equation we are using a sigmoid function S to limit the outputs to $[-1, 1]$. We used $\frac{1}{1+e^{-x}} * 2 - 1$ as sigmoid but any function $f : x \to y, x \in [-6, 6], y \in [-1, 1]$ would have worked since the only concern was to limit the output range to $[-1, 1]$. We chose the sigmoid however since we wanted linear behaviour with a high steepness near $x = 0$ since that gives the best learning rate and a flatter curve at the

$$\Delta r = \begin{cases} 0 & \text{if } l_4 = 0 \text{ and } r_4 = 0, \\ S(f(\mathbf{x_l})) & \text{if } l_4 \neq 0 \text{ and } r_4 = 0 \\ S(f(\mathbf{x_r})) & \text{if } l_4 = 0 \text{ and } r_4 \neq 0 \\ S(f(\mathbf{x_l}) + f(\mathbf{x_r})) & \text{if } l_4 \neq 0 \text{ and } r_4 \neq 0 \end{cases} \tag{3.2}$$

Figure 3.2. Calculating the Δr using RBF functions. Twelve genes are implicitly used, 6 genes for σ:s and μ:s in f (see Figure 2.4) using input $\mathbf{x_l}$ and 6 for $\mathbf{x_r}$

extremes. An x value near the extremes of $[-6, 6]$ corresponds to radical behaviour such as turning 180° or accelerating very rapidly, an x value near 0 corresponds to making minor adjustments of speed and angle when encountering an object. A high x value also corresponds to a very rare event occuring, namely that both antennae detect objects with very high color values. Since we wanted to train our creatures to behave rationally on the common events, we chose the sigmoid function which slows down learning of rare, extreme events and behavoiurs, far from 0, and accelerates learning of common, rational events and behaviours, close to zero. In this way, the creatures still have the ability to react strongly, e g turning 180° when seeing an predator, but learning focuses on the most interesting behavoiurs namely which way to turn or which way to accelerate. We could just as easily have used $f(x) = x/6$ as a normalizing function, but that would probably have slowed down learning considerably.

3.1.5 RBF-Based Decision Making

In RBF-based decision making, the three inputs to each antenna are used in the function displayed in Figure **??**. For each antenna, Δr and Δs are computed by summing RBF function values and normalising them using the same function S as in chapter 3.1.4. And as in chapter 3.1.4 Δr and Δs are calculated separately using the same function and inputs but using different genes.

Each radial basis function has a σ and μ which are decided by the creature's genes. σ and μ are in the ranges of $[0, 1]$ and $[-1, 1]$ respectively and this means that the RBF brain needs twice as many genes. The formula for RBF decision making is shown in Figure 3.2

The difference between using radial basis functions and linear functions is that you have a larger possibility to approximate any decision-making strategy. For example an RBF-based brain could make the distinction between different shades of green and thus react differently to them, while a linear function could only decide if more green is better or worse.

3.1.6 Random Decision Making

The random decision making brain is very basic. The process is illustrated in Figure 3.3.

$$\Delta r = \begin{cases} 0 & \text{if } l_4 = 0 \text{ and } r_4 = 0, \end{cases} \tag{3.3}$$

Figure 3.3. Calculating Δr using random brain and the same notation as in Figure 3.1 and Figure 3.2

3.1.7 Genetic Algorithm

What kind of genetic algorithm do we plan to use?
Fitness, crossover, mutation, selection, which ones?
Flow chart of our algorithm
NSGA-II
Roulette selection

3.1.8 Experiments

(In every experiment linear and RBF are compared, eventually mixed)
Interested in:
Time to maximal fitness
Highest possible fitness

- Adaptation 1. Finding and eating food
- Adaptation 2. Finding and eating food, avoiding "bad" food
- Co-evolution and extinction 1. Predator vs prey, prey eats food as in first experiment. (bushes are bad for predators)
- Speciation and co-operation 1. Allow the creatures to evolve which "colour" food they can eat, attempt to create two separate species from one, one species eating green bushes and one red.
- Mimicry 1. Making the prey mimic bushes or predators to avoid being eaten.
- Mimicry 2. Predators attempt to mimic bushes to make prey run into them.
- Group forming 1. Rerun all previous experiments, but with the ability to detect the nearest creature of the same species.

Chapter 4

Results

4.1 Simulation results

4.1.1 Observed Behaviours

4.2 Discussion

4.2.1 Constraints and problems

Performance problems
Other unexpected problems?

4.3 Conclusions and Future Work

Bibliography

[1] Montana, D. J., and Davis, L. (1989, August). Training feedforward neural networks using genetic algorithms. In *Proceedings of the eleventh international joint conference on artificial Intelligence* (Vol. 1, pp. 762-767).

[2] Langton, C. G., and Shimohara, T. (Eds.). (1997). *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems* (Vol. 5). Mit Press.

[3] Marsland, S. (2009). *Machine Learning, an Algorithmic Perspective.* CSC-Press

[4] Holland, J. H. (1992). Genetic algorithms. *Scientific american, 267(1)*, 66-72.

Appendix A

Third party libraries and tools used

- DEAP - Distributed Evolutionary Algorithms in Python `http://deap.gel.ulaval.ca/doc/default/index.html`
- PyPy - A fast Just-in-Time compiler for Python. `http://pypy.org/`
- Pygame - A computer game and visualization package for python. `http://www.pygame.org/docs/`
- NumPy - A Python package for numerical calculations. `http://www.numpy.org/`
- matplotlib - A python package for rendering graphs. `http://matplotlib.org/`