Computation Project: Robotic Animal Herding

Pont Lurcock[1]
Worcester College
Oxford University

May 18, 2001

[1]pont@talvi.net

Abstract

An investigation into the use of neural networks and genetic algorithms for herding animals. Parts of a previous Robot Sheepdog Project are reimplemented in a modular, object-oriented style in Java, and new algorithms are considered for controlling the Herder entity in the simulation. A simple recurrent neural network architecture is implemented, and trained using backpropagation and genetic algorithms. Backpropagation is found to be difficult to apply effectively to simple recurrent networks, but some success is achieved with genetic training.

Contents

Chapter 1

Introduction

> Never express yourself more clearly than you are able to think.
>
> — Niels Bohr

This project is an investigation into techniques of robotic animal herding. It builds on work done by the Robot Sheepdog Project, conceived at the Silsoe Research Institute.

For convenience, the term 'duck' is used throughout this report for the (simulated) animals being herded, and 'dog' for the simulated entity doing the herding. For the purposes of this project, it's not very relevant what the simulated entities represent, since the flocking behaviour modelled is fairly generic.

1.1 Previous work on the Robot Sheepdog Project

The original goal of the project was to work towards the development of a robot sheepdog and algorithms for controlling it in order to herd sheep. Through experiments involving a robot and a flock of ducks, algorithms were developed to model the behaviour of the ducks and to control the herding robot. Further experiments were conducted entirely with simulations (written in C++) of the animals, yielding refined algorithms for controlling the ducks and dog. Last year work was also done on graphical representations of the simulation, but this work is not followed up by this project.

1.2 Goals of this project

This project had two main goals:

1. To reimplement the simulation (as opposed to display) parts of the previous year's work, changing the implementation language from Visual C++ to Java, and adding a Swing GUI. This is far from being a straight translation: the existing code was written in a fairly non-modular procedural style. My goal was to produce a well-designed, object-oriented system with clear interfaces between the main simulation and the simulated animals, making it easy to experiment with different models for the dog and ducks.

2. To develop new algorithms for controlling the dog, based on neural networks and genetic algorithms.

Clearly the first goal makes the second a lot easier, and it also provides a convenient framework for any future work on the project.

1.3 An overview of the project's structure

The set-up of the project is loosely modelled on the relationship between a real shepherd, sheepdog and flock of sheep. The simulation takes place in a circular arena populated by a dog and a number of ducks. The animals' control algorithms have full knowledge of the simulation's state, although the ducks only react to events in their immediate vicinity. The dog-control algorithm plays the part not only of the dog, but also of the shepherd. It is therefore assumed to have an overview both of the arena's physical layout and of the overall goal being worked towards.

The arena is initialized with a dog position and a set of duck positions. When the simulation is running, a single step consists of updating the duck positions, followed by the dog position. The simulation finishes when some goal condition is achieved, usually when the ducks have all been herded into a specified region.

Chapter 2

A framework for developing control algorithms

> The height of technical felicity
> is to combine sublime simplicity
> with just sufficient ingenuities
> to show how difficult to do it is
>
> — Piet Hein

2.1 Reimplementation issues in the arena simulation

Previous work on the robot sheepdog project had resulted in a fairly straightforward simulation architecture: a circular arena, containing a number of Ducks and a Dog. On successive iterations the dog and duck positions were updated according to the algorithms which had been developed for their control.

The implementation of this simulation was extremely monolithic: dog control, duck control and general arena-related code were combined into a single "update" function. This is an undesirable state of affairs for several reasons:

1. The agglomeration of concerns makes the code difficult to read and maintain (the update function was over 800 lines long). The use of cut-and-pasted source code rather than functions drastically reduces maintainability.

2. It was also difficult to experiment with different models for the simulated animals, or indeed for the arena itself.

3. The heavy use of global variables and lack of modularity meant that the algorithms were not cleanly separated; as a result, it was possible for the control algorithms to "cheat": making use of information which would not be apparent in a real situation, or directly manipulating the simulation's data structures. It was, for example, possible for a dog-control algorithm to "teleport" the dog instantaneously from one side of the arena to the other, or to directly manipulate the ducks' velocities.

   These examples are a little extreme, but one can imagine more subtle, accidental violations of the simulation's physics. Although this would have no serious consequences within the scope of this project, it would be disastrous for anyone attempting to implement the algorithms in real robots: breaking the laws of physics is a lot easier in a simulation than in real life.[1]

[1] As a character states in the mediocre sci-fi horror film *Event Horizon*, "When you break all the laws of physics, do you seriously think there won't be a price?" In this case, the price turned out to be opening "a portal to a dimension of pure evil" and the gruesome deaths of most of those involved; I do not envisage this happening with the duck simulation, but it's better to be safe than sorry.

2.2 Object-oriented design of the new code

I chose an obvious way to split up the simulation: one object each to represent a dog controller, a duck controller and a simulation state. Careful design of the interfaces would prevent the duck and dog controllers from meddling with anything they shouldn't. All the new code was written in Java.

2.2.1 Supporting classes

I started by implementing some classes necessary for modelling the movement of the animals:

- `Vec` is a `float`-based two-dimensional Cartesian vector class which does everything one would expect a vector to do. It includes methods for most common vector arithmetic operations, as well as a `clone` method. Cloning is important, since only clones of the simulation's state are passed to the dog and duck controllers, keeping the real data inviolable. As much "high-level" functionality (normalization, conversion to polar co-ordinates and so forth) as possible is delegated to `Vec`, in order to minimize cloning and increase speed.[2]
- `Pos` simply encapsulates two `Vecs` representing an object's position and velocity. It also includes a `move()` method which adds the velocity to the position. The intention is that, during simulation, an animal's position will only be manipulated via the velocity vector and the `move()` method, thus ensuring at least slight conformance with the laws of physics.

I considered adding an extra level of realism by only allowing manipulation of *accelerations* rather than velocities; this would disallow behaviour such as instantaneous changes of velocity, and would allow for elegant modelling of collisions: the simulator would simply add an opposing acceleration representing the reaction force from the object collided with. However, I decided against this approach for a number of reasons:

1. The existing algorithms only worked in terms of velocities, and had no concept of acceleration. Since I required at least the existing duck model for my work, I would have had to reimplement and test a new duck-control algorithm using accelerations. This would have been time-consuming and tangential to the project's goals.
2. The collision modelling would not be especially useful, since the dog should not be colliding with walls or ducks anyway.
3. As regards the realism of the simulation, the real robots on which the algorithms had been tested were fairly small, and moved quite slowly. Thus issues of momentum were unlikely to become important.
4. I intended to use neural networks with continuous activation functions to control the dog. Their output would thus be a continuous function of the input and completely instantaneous changes would be impossible.

2.2.2 The Arena, Duck and Dog objects

Now I could implement the classes comprising the core of the simulation. The central class is called `Arena`. It maintains the current state of the simulation and handles the initialization and update of the animal positions. It provides a large number of methods for reading and setting animal positions. However, access to these methods is filtered through three control interfaces, all of which extend a basic `CommonControl` interface.

- The `CommonControl` interface allows reading of the current state of the animals in the simulation, as well as more general parameters such as the arena size.

[2]Backpropagation and genetic algorithms are both computationally intensive, and Java is not known for its blistering speed, so speed considerations are important here.

- The `DogControl` interface allows, in addition, setting of the dog's velocity.
- The `DuckControl` interface allows setting of the duck's velocity.
- The `TrainerControl` interface is intended for use with training algorithms which need to set up particular environments. It gives unrestricted access to dog and duck positions, allowing them to be manipulated at will.

Read-only access to data is given by cloning the appropriate objects. This introduces a time overhead, but it is the only effective way to do it in Java. However, the computationally intensive training process uses an interface which gives direct access to the data structures: the overhead is only incurred during the normal, real-time running of the simulation, which doesn't suffer noticeably from it.

The dog and duck objects in a simulation are both concrete subclasses of the same abstract class, `AnimalModel`. This class contains two methods, `update()` and `postUpdate()`, which are called at every step of the simulation, respectively before and after the current velocities are applied. Note that although the dogs and ducks both implement the same class, they do not have the same access to the `Arena` object: the `AnimalModel`'s constructor is passed a reference to `Arena` cast to either a `DogControl` or a `DuckControl` as appropriate, and so only gets control of its part of the simulation.

2.3 Adding a GUI

It's not much good running a simulation if you can't see what it's doing. The next step was to implement an `ArenaView` class which would provide a display of the simulation's current state. This is a fairly straightforward overhead view implemented in Java2D. Each animal is shown as a circle, with the magnitude and direction of its velocity shown by a projecting line (see Figure 2.1).

2.4 Tying it all together

The whole simulation is controlled by the `DuckSim` class. This provides a simple control panel, maintains references to the arena and animal models, and allows the user to reset and run the simulation. The simulation is run by starting a separate thread which calls the arena object's update function at regular intervals. Figure 2.2 shows an outline UML class diagram of the `ducksim` package.

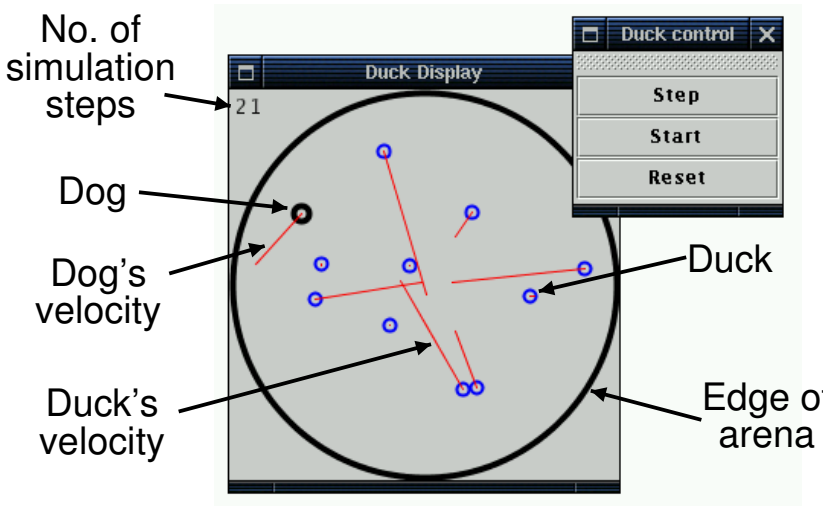


Figure 2.1: The arena display of the duck simulation, and the control panel.

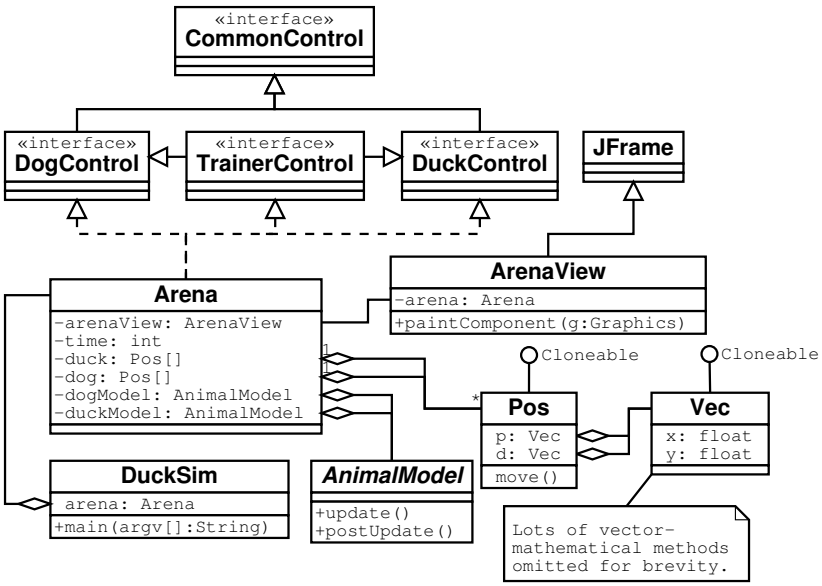


Figure 2.2: Simplified UML class diagram of `ducksim` package. Many attributes and methods are omitted for clarity.

Chapter 3

Neural networks for control

> I have yet to see any problem, however complicated, which, when you looked at it the right way, did not become still more complicated.
>
> — Poul Anderson

3.1 Possible network architectures

With a simulation framework in place, the next step was to investigate new algorithms for control. The aim was to use some form of neural network controller, but this still leaves a bewildering choice due to the huge variety of network architectures. An extensive survey of neural network architectures and training algorithms suitable for control systems is given in [HSZG92].

One the best-known types of neural net is the multilayer feedforward perceptron, most usually trained using backpropagation [RHW86]. A brief description of this method is given in Section A.1. This was one of the first systems I considered; however, both the architecture and the training algorithm are, in their usual form, inadequate for this project. The standard feedforward network cannot be used to model events with temporal structure, since it has no internal memory and the outputs at any point in time are entirely dependent on the inputs[1]. It is clear that, no matter what training algorithm is used, the controller would always produce the same output in the same situation: it would be impossible for it to develop the higher-level strategies needed to control the flock.

Backpropagation as a training algorithm also has its drawbacks for this task: with backpropagation, a network is trained by presenting it with an input and comparing its outputs with the desired set of outputs. However, in this situation the desired output is not as clear as in, say, a pattern-recognition task. Rather, the correctness is measurable only in terms of the *effects* of the outputs on the simulation over a period of time. Measuring the "correctness" of a controller is relatively easy, by awarding scores for achieving desirable tasks. Propagating this information back into the network is harder, although complementary reinforcement backpropagation, discussed in Section 3.5.1, provides one solution.

Another way to extend backpropagation is discussed in [TW88]: the rest of the simulation is also implemented as a neural network, and is grafted onto the controller to be trained, the controller's output nodes forming the inputs of the rest of the system. Backpropagation is then applied as usual, but only the weights of the controller are modified. Due to the complexity of reimplementing the rest of the simulation as a neural net, and the relative paucity of literature on this technique, I decided against attempting this.

[1] Actually, time *can* be modelled by feedforward nets, by representing it as an extra spatial dimension, but there are severe limitations with this approach (discussed in [Elm90]), making it unsuitable for this project.

3.2 Simple recurrent networks (SRNs)

The next stage was to investigate recurrent networks – networks containing connections which do not go strictly forwards from layer to layer. This provides for much more capability in responding to inputs which vary through time, but makes training more difficult and time-consuming. One of the best-known techniques for training recurrent nets, backpropagation through time, becomes very expensive in time and memory when used for longer periods ([RHW86], p. 356).

An architecture which seems to offer many of the benefits of a recurrent network without sacrificing ease of training is the Simple Recurrent Network used by Jordan ([Jor86]) and Elman ([Elm90]). Although its most common application seems to be in linguistic areas, where it can be trained to behave like a finite-state automaton, it has also been used for controllers similar to the one in this project ([Mee96]).

The SRN architecture extends a standard 3-layer feedforward network by adding a layer of *context nodes*; a simple example is shown in Figure 3.1. There are as many of these as there are hidden nodes, and they act as a short-term memory for the hidden layer. Like the input layer, the context layer has forward connections to the hidden layer and is activated before the hidden layer during forward propagation. However, the hidden layer also contains one-for-one backlinks to the context layer (i.e. each hidden node is connected to exactly one context node). These backlinks have a non-adjustable weight of 1 and the context nodes use an identity activation function; in other words, the hidden-layer activations are simply copied into the context layer after they have been calculated. The net effect is that, on every forward-propagation pass, each hidden node is presented with its own previous activation as one of its inputs. In spite of this seemingly short-term memory, SRNs have proved capable of developing fairly long behavioural patterns.

The SRN architecture adds another tunable parameter to those already present in a standard feedforward network: the initial activations of the context nodes. Usually these are reset to some neutral value, such as 0.5, at the start of every sequence of training patterns. The longer the network is run for, the less significant these become, so I didn't investigate the effects of changing their values.

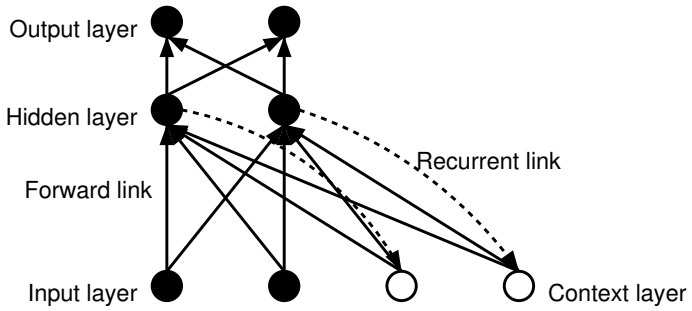


Figure 3.1: A simple recurrent network with two input nodes, two hidden nodes, and two output nodes. The recurrent links to the context layer are shown as dotted arrows.

One of the attractive aspects of SRNs is that they can be trained by simple backpropagation; during the backpropagation phase the context nodes are just treated as another set of input nodes. Meeden ([Mee96]) investigates two training methods, complementary reinforcement backpropagation and a genetic algorithm, and I decided to try the same methods for the dog controller. These methods are described in more detail in Appendix A, and in Sections 3.5.1 and 3.6.1.

3.3 Coupling a neural network to the dog controller

A vital part of the project was the mechanism for actually associating the network's input and output values with the motion of the dog: what information to feed in, and how to interpret the outputs. An obvious first step, both for input and output, was to translate the simulation's

Cartesian co-ordinates to polar ones relative to the dog's current position. Intuitively it seems simpler to develop control algorithms in terms of directions and distances than in terms of x and y co-ordinates.

Input encoding A good rule of thumb with neural networks is never to use more neurons than necessary. Using too large a network needlessly enlarges the space that learning algorithms have to search, thus increasing training times. It can also cause overfitting to noisy data or to specific training samples. I decided to experiment with a few carefully selected and preprocessed values as inputs for the network. I implemented functions to convert positions to a direction and distance from the dog, both scaled to the range $[0, 1]$. The most important data were the position of the nearest point on the wall (vital to avoid colliding with it), the position of the flock centre, and the radius of the flock. I also considered adding the position of the closest duck, but decided to leave it out, at least initially.

Output encoding The network only needs to produce two values, controlling the dog's speed and direction respectively. However there was still choice as to the exact control method: should the speed (or angular velocity) be applied directly, or should it be interpreted as an acceleration and added to the current value? And to what ranges should the outputs be scaled? Again, experimentation was needed. The results are discussed in Section 4.2.

3.4 Implementing neural networks in Java

The first step was to write a Java package to implement SRNs. In a spirit of modularity, I kept this completely separate from the `ducksim` package: the two packages only intersect in the neural dog-controller class attached to `Arena`. I called my neural network package `neurotic`; a UML class diagram of the final version is shown in Figure 3.2.

There is a lot of choice as to the implementation of neural networks in Java. Much of this is to do with the tradeoff between speed on the one hand, and clarity and flexibility on the other. A fast solution would be to store data such as weights and activations in arrays, allowing them to be accessed quickly by training algorithms. However, this implementation would be quite rigidly tied to a particular network structure. Because I wanted the freedom to experiment with different network architectures, and because I didn't anticipate using networks of more than around a dozen neurons at most, I thought a slower, more object-oriented approach would be most suitable.[2]

I made the obvious design choice of creating a `Neuron` class representing a node in the network. At first I attempted storing the link data within the nodes themselves, but this quickly became messy since the data needed to be accessed from both ends of the link. After some experimentation, I introduced a binary association class `Link` representing a link between neurons.

Eventually I made `Neuron` a concrete subclass of an abstract class `Axon`, which represents anything that can produce an activation in a network. This encompasses input nodes, "normal" nodes like Neuron, and the context nodes used in SRNs.

The hard work of creating and running networks of these objects is done by the `Net` class, representing an entire neural network. It initializes a feedforward network or SRN based on parameters passed to the constructor, and provides methods for setting the inputs, running forward-propagation, and getting the ensuing outputs. The most important methods and inner classes are:

- `Net`, the constructor, which creates a new 3-layer network (recurrent or feed-forward) with the number of nodes in each layer supplied as arguments.
- `randomizeWeights(float mean, float range)` initializes every trainable weight in the network to a uniformly distributed random value with the given mean and range.

[2]The (non-neural) duck controller turned out to be the bottleneck when training (see Section 4.2.2) so this was probably the right choice.

- `setIn`, `getIn` and `getOut` are accessor methods for the input and output nodes.
- `Activation` is a wrapper class around an activation function for the net, and `Sigmoid` is a private subclass giving the usual activation function. Since I didn't intend changing this, it is currently "hard-wired", but could easily be altered.
- `LinkEnum` is a public inner class implementing the `Enumeration` interface. Instantiating it gives an enumeration over all the links in the network; it is useful both for `Net`'s internal methods and for other classes which might wish to read or modify weights.
- `getAllWeights` and `setAllWeights` allow access to all the network's weights as an array, and `numWeights()` returns the number of weights in the network.
- Various methods for training the network by backpropagation, described in Section 3.5.2.
- Various methods for debugging.

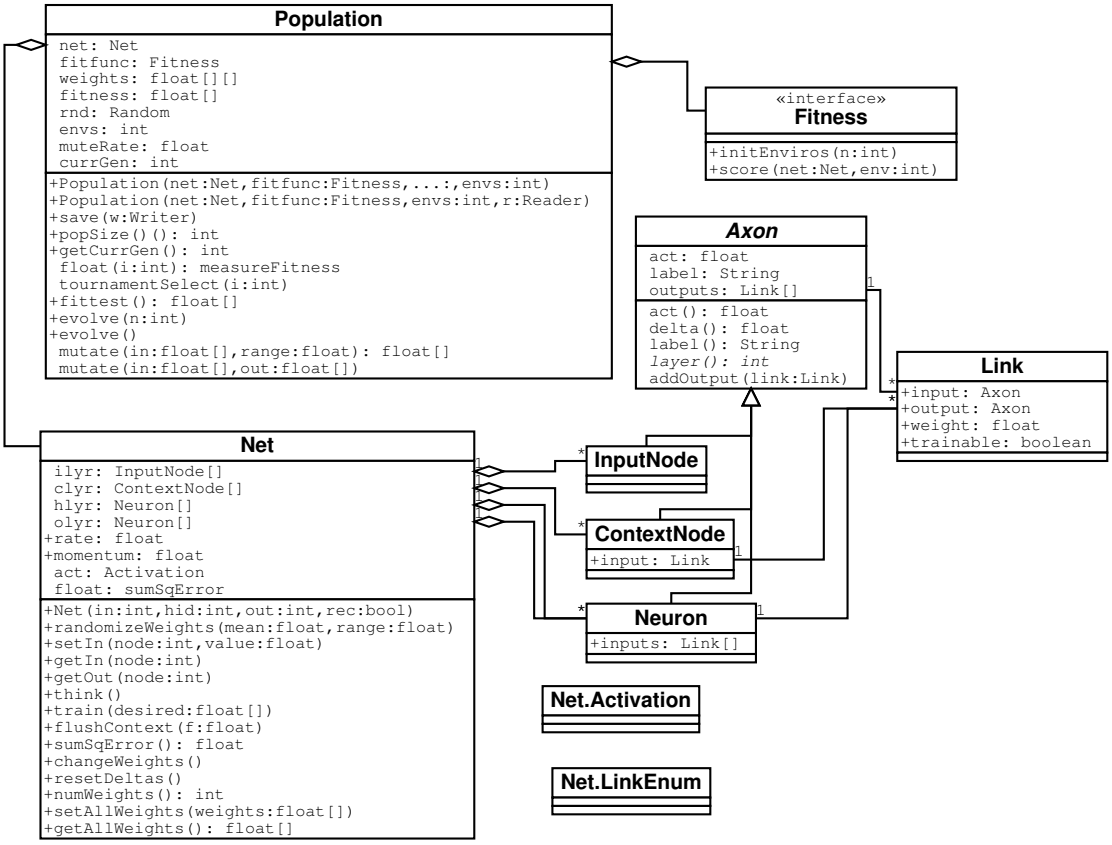


Figure 3.2: UML class diagram of `neurotic` package (some fields and methods are omitted for clarity).

3.5 Training with backpropagation

3.5.1 Complementary reinforcement backpropagation

Complementary reinforcement backpropagation (CRBP), described in [AL90], is a method for getting around the problem discussed in Section 3.1: the fact that normal backpropagation can

only train a network on the basis of its actual outputs, against some other set of desired outputs. CRBP provides a method for training a network on the basis of a fitness function defined by the network's interaction with some wider context.

The principle of CRBP is to extrapolate the network's actual output vector to produce a "virtual" target vector against which to perform the usual backpropagation algorithm. In slightly more detail, the procedure is as follows:

1. Initialize the network randomly, as usual.
2. Run a forward propagation. Take the output vector from the network (the *search vector*). Use it to construct a vector of equal length consisting of 0s and 1s, which [AL90] confusingly refers to as the *output vector*. Each element s_i of the search vector is interpreted as a probability; the corresponding element o_i of the output vector is generated (pseudo-)randomly, taking the value 1 with probability s_i and 0 otherwise.
3. Evaluate the performance of the 'output vector', to produce a result indicating either that it is desirable and should be reinforced, or that it is undesirable and should be discouraged.
4. If the 'output vector' is to be reinforced, it is simply used as the target vector for a backpropagation pass: the error vector $\mathbf{o} - \mathbf{s}$ is propagated through the network.
5. If the 'output vector' is to be discouraged, it is slightly less obvious what to do: clearly the weights should be pushed away from the output vector, but in which direction? In the CRBP algorithm they are pushed in the exact opposite direction: the vector $(\mathbf{1} - \mathbf{o}) - \mathbf{s}$. Generally a lower learning rate is used here than for positive reinforcement, since the network is not necessarily being pushed towards a good set of values: it is merely being pushed away from a bad set.

The procedure is repeated until the error has been sufficiently minimized.

Although CRBP allows reinforcement methods to be used to train neural networks, it is still geared towards immediate reinforcement: the reinforcement value must be calculated after each forward-propagation pass. There is no direct way to define a fitness function over a longer sequence of behaviour. With a recurrent network, the network itself can still develop longer-term strategies from short-term reinforcement, but [Mee96] gives discouraging results about the difficulty of doing this. However, since most of the tasks I was looking at lent themselves to immediate evaluation (for example, "has the size of the flock decreased?"), this was not too great a problem.

3.5.2 Implementation of the backpropagation algorithm

Since I intended to experiment with variations in the backpropagation algorithm, I started with a fairly large, open interface to the backpropagation methods. As it turned out, this interface remained in place: the failure of backpropagation as an effective training method (see Section 4.1) meant that it was not worth improving the interface.

Backpropagation works on a single network, so it made sense to implement it using instance methods of the `Net` class.[3] The main public methods for controlling backpropagation are:

- `resetDeltas()` zeroes the accumulated weight changes built up over successive backpropagation passes.
- `flushContext(float x)` sets the values of all the network's context nodes.
- `train(float[] desired)` performs a forward-propagation pass followed by a backpropagation pass against the given target vector. It is assumed that the input values have already been set with the `setIn()` method. Weight changes are added to the internal accumulator variables, but not applied.

[3]This is in contrast to the genetic algorithm discussed in the next section, which works on populations of networks and thus requires a somewhat different implementation strategy.

- `changeInputWeights()` applies the weight changes accumulated during backpropagation.
- `sumSqError()` returns the sum of squared errors computed on the last backpropagation pass.

The chief reason for separating out the calculation of weight changes from their application is to make it easier to experiment with batch and stochastic weight-update techniques (see page 24).

3.6 Genetic algorithms (GAs)

A brief description of common GA techniques is given in Section A.2. The following sections describe the deviations from "standard practice" in my use of genetic algorithms, and the details of their implementation in Java.

3.6.1 Applying genetic algorithms to neural nets

I restricted the use of genetic algorithms in the project to evolving the actual weights for the network, although there is scope for other applications – for example, determining the network architecture, or adjusting parameters for other learning methods such as backpropagation. However, I thought that using a genetic algorithm for exactly the same task as CRBP would make for an interesting comparison.

Following the approach of [Mee96], I ended up using a somewhat unorthodox variant of genetic training, due to the unusual (for the field of GAs) structure of the solutions I was attempting to evolve:

String encoding The first deviation from the norm is in the encoding of the strings within a population. An individual in the population consists of a string of floating-point numbers representing weights. Of course, it would be possible to convert the weight values to bit-strings, but this would bring no obvious benefits.

Crossover Crossover is not used at all. This may seem a little odd, but for strings of weights it makes sense: the operation of a neural network is highly distributed, with few or no easily identifiable "modules" in most cases. The "parent" networks may be using completely different strategies to achieve the same goal. Even if they are not, the same network architecture can produce the same behaviour using radically different weights.[4] so an arbitrary substring of a "good" weight-string is unlikely to possess any intrinsically good properties of its own.

In [Mee96] it was found that, even if weights were grouped into higher-level blocks to counteract this effect, crossover "did not improve and often hurt performance". For this reason I decided not to use crossover, at least not initially.

Mutation Mutation is used much more heavily than is usual. Partly this is to compensate for the lack of crossover, and partly because mutation need not be such a drastic process when the string elements are continuous values rather than bits – values can just be slightly perturbed rather than flipped to the other of two possible states.

In practice, the mutation method used was a fairly brutal one, but one which is successfully used in [Mee96] and [Fog98]: *every* weight in the string to be mutated is modified by a random amount. [Yao99] also cites several successful applications of similar techniques. The range of modification is configurable by the user, but in practice it was generally between 2 and 10 (that is, ± 1 to ± 5).

[4] As a trivial example, consider a 3-layer feedforward network with activations in the range $[-1, 1]$. Provided the activation function is odd, simply negating all the weights will give a network with identical behaviour

Fitness function The fitness function was implemented in a slightly unusual way. Instead of a wholly objective function which can assign an absolute fitness to any set of weights, I used functions which evaluated a network's fitness with respect to certain environments – i.e., initial states of the simulation. Given an initial state consisting of the positions and velocities of the simulated animals, the fitness function runs the simulation for a certain amount of time and assigns scores based on its behaviour. For example, in most cases the situation "dog runs into wall" would be awarded a negative score, in order to encourage the propagation of networks which avoid this behaviour.

One obvious danger with this method is that a network trained in a specific environment will utilize the specific features of that environment to perform successfully, and may well exhibit very poor performance when placed in other environments. One way to counteract this is to use a set of several training environments and take the total or average score. The modification described below also helps.

Tournament selection The lack of an objective fitness function requires some modification to the overall structure of the evolution algorithm. As discussed above, it would be undesirable to train in a single environment; even training in a fixed set of predefined environments would incur the risk of overspecialization. For this reason, a new set of environments was created at the start of every generation.

The technique used for creating new generations was *tournament selection*. Pairs of individuals are randomly selected from the population, and their fitnesses assessed in the current set of environments. The loser (i.e. the individual with the lower fitness) is replaced by a mutation of the winner. To advance evolution by one generation, this procedure is repeated a number of times equal to the population size. Since the environments are reinitialized at the start of each generation, overspecialized individuals do not survive for long.

3.6.2 Implementing genetic algorithms in Java

Genetic training proved remarkably simple to implement compared to the intricacies of backpropagation. I added a class `Population` to the `neurotic` package. This class is instantiated with a `Net` and a fitness function, as well as a few training parameters such as population and mutation rate. `Population` stores arrays of floats representing weights (either initialized randomly through the supplied network or loaded from a file) and plugs them into the network when fitness needs to be assessed.

The fitness function supplied to the network is just an interface consisting of two methods: `initEnviros(int n)` initializes the requested number of environments (usually called once at the start of each generation), and `score(Net net, int env)` returns the fitness of the given net with respect to the given environment.

The rest of the class just implements the mutation and tournament selection described in the previous section, and provides a few accessor methods for things like the current fittest individual and generation.

Chapter 4

Results of training

> Among all the supervised learning algorithms, backpropagation is probably the most wi(l)dely used.
>
> — Yann LeCun

4.1 Backpropagation

Since backpropagation is a relatively involved algorithm with plenty of scope for bugs in the implementation, I decided to start with some simple tests and work up to the use of CRBP on recurrent networks.

4.1.1 Non-recurrent networks

The first step was to get backpropagation working reliably for non-recurrent networks. The multilayer perceptron equivalent of "hello world" is the exclusive-or function[1]. I successfully trained a non-recurrent network (2+2+1 neurons) to model this function. Training times were highly dependent on the learning rate and momentum parameters; with high values for both (around 1.0 and 0.8 respectively) the network usually required a few hundred passes through the training set to reach a sum of squared errors below 0.001. The code for this example is given in Section B.3.1.

I compared the performance of my backpropagation routines with that of PDP++, a publicly-available neural-net package written in C++. For the XOR example, the behaviour of the two systems was completely identical given the same starting conditions, so I was satisfied that `neurotic` was fairly reliable on non-recurrent networks.

4.1.2 Simple recurrent networks

In a similar vein to the preliminary test for non-recurrent networks, I decided to start by training an SRN on the function used in [Elm90]: sequential exclusive-or. The idea is that a sequence of inputs (1 or 0) is presented to the single input node of an SRN. The input sequence is made up of a number of 3-element subsequences, with the first two elements being random and the third being their exclusive-or. The network has a single output, which is trained against the next element in the sequence. Obviously the network can't predict the first two elements of a subsequence, but Elman succeeded in training it to predict the third element when the second was presented.

I was unfortunately unable to do the same. In spite of extensive experimentation with tunable parameters and repeated careful scrutiny of the code for bugs, the network resolutely failed to converge. Unfortunately [Elm90] is somewhat sketchy on the details of the training process, and I was unable to find any more detailed references on training SRNs with backpropagation.

[1] This is mainly because the XOR function can't be modelled by a single perceptron.

Slight success with SRNs

Eventually I scaled the training problem down to an even simpler one: the network just needs to echo the previous input in the sequence. Training from randomly initialized weights proved just as hopeless as for the XOR problem. However, this problem was so simple that I had little trouble working out a suitable set of weights by hand. This led to some interesting results: I found that if I ran backpropagation with an initial weight-set close enough to my known solution, it would converge. This suggested to me that there was no fundamental problem with the algorithm, but that some maladjustment of network or learning parameters was preventing effective convergence. The program listing is given in Section B.3.2; some sample output is shown below.

```
Performance before training:

Input  ***..*...*....*.*..***..*.*.*.***..***...*....*..****.**.*..
Output ?*??.??..??...??.*.?*??.??.*.*.*?*.?*??..??...??.?*?*?.*???.

After 800 backprop passes

Input  **..*...**..*...*..*.*....**..**.*..**..*.**.***.*..*.*.***.
Output ***..*...**..*...*..*.*....**..**.*..**..*.**.***.*..*.*.***
```

The program starts with a 1-2-1 network initialized with weights close to the ones I calculated. It runs a test by sequentially feeding it 60 random inputs (either 1 or 0, shown as "*" and "." respectively) and showing the output (encoded as "." for under 0.2, "*" for over 0.8, and "?" otherwise).

The network is then trained with backpropagation on another randomly generated binary sequence. After 800 passes, there is a clear improvement in the performance; Figure 4.1 shows how the 11 weights in the network change with training.

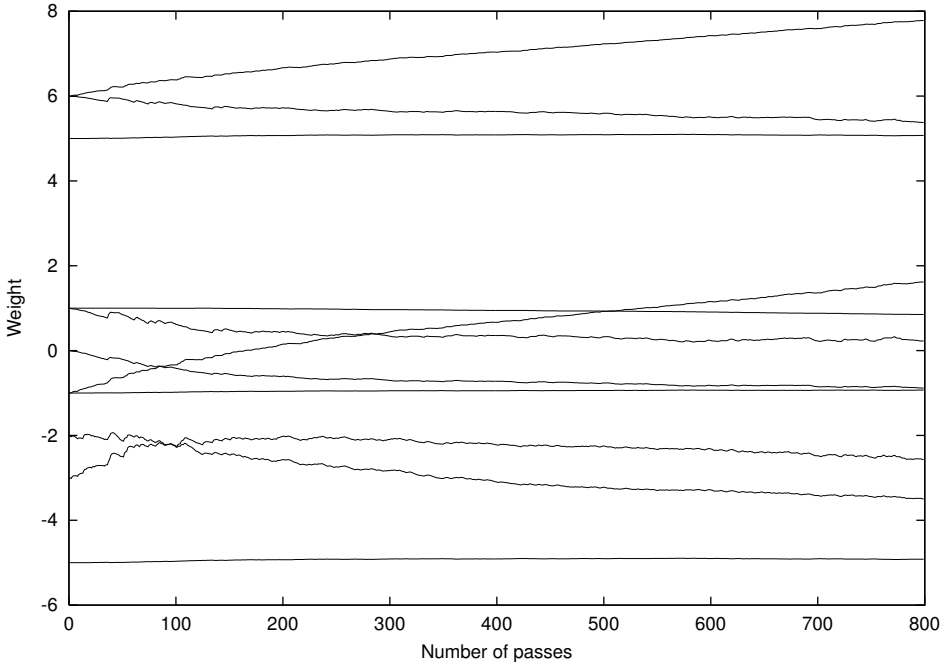


Figure 4.1: Weights in the "input-repeater" neural network of Section 4.1.2, plotted against number of backpropagation training passes.

I spite of this promising result, I had already spent a great deal of time trying to get back-propagation working and could not afford to pursue it any further. I reluctantly abandoned the implementation of CRBP and moved on to genetic algorithms.

4.2 Genetic algorithm

There were two main areas of experimentation here: the actual mechanism for coupling the neural network to the dog controller, and the fitness function which pushes the network's weights towrds the desired behaviour. In addition, there was scope for variation in the parameters of the genetic algorithm – for example, the population size and mutation rate.

I created a new `AnimalControl` class called `DogBreeder`, which forms the link between the duck simulation and the `neurotic` package. The constructor takes a `DogControl` and a `TrainerControl`, for running and training the neural controller respectively. On instantiation, `DogBreeder` creates a network and a population of weights. It also creates a `DbPanel` object, which provides a Swing interface allowing the user to control the training of the networks. Facilities are provided to start and stop evolution (which takes place as a background thread), and to alter parameters such as the population size and the number of hidden nodes. The actual `update()` function which controls the dog's movement is delegated to a servant class implementing `DogBreeder.ControlInterface` (which is essentially just a wrapper for `update()`), since I was making a lot of modifications to this part.

The Swing GUI for the `DogBreeder` class is shown in Figure 4.2.

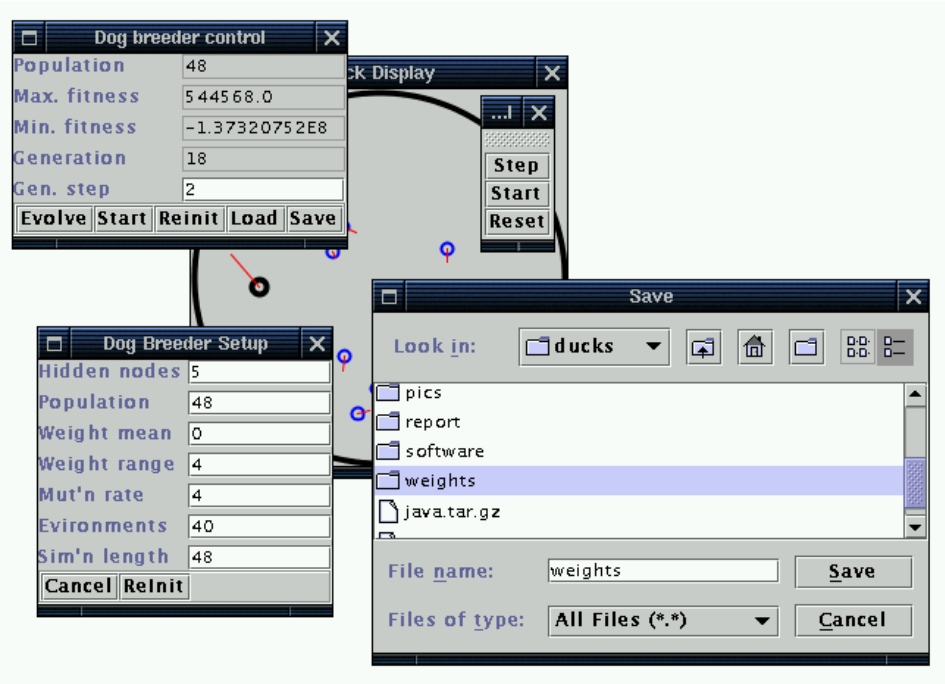


Figure 4.2: The user interface for the `DogBreeder` class.

4.2.1 Avoiding walls

I thought a good preliminary test for the genetic training method would be wall avoidance: the fitness function just imposes a heavy penalty (negative score) for running out of the arena. With only this criterion the most likely result is a dog which doesn't move at all, so the fitness function also adds a positive score proportional to the distance moved over the training period.

I experimented with both direct mappings of the outputs to speed and direction, and with using them as increments to be added on. I also experimented with different scaling factors for the outputs. The goal proved easy to achieve, whatever the exact system used.

The best performance seemed to result from mapping one output directly to the dog's speed, and using the other as an increment to the direction. This was the system I used for most of the later experiments.

`AvoidWallsFitness`, the fitness function I developed for this task, is listed in Section B.1.7. In operation, the controllers it produces tend to move around the arena in circles, although I did also find variants which had a more radial pattern of movement.

4.2.2 Training problems

The two main problems I had with genetic training were closely interrelated. The first was the speed of the simulation. For real-time running of the simulation, my implementation had proved more than fast enough. However, the genetic training was much more computationally intensive; it takes hundreds or thousands of simulation steps to evaluate the fitness of a single individual in a population. This was not a great problem for the wall-avoidance task, since there was no need to involve the ducks in the simulation. However, as soon as I started tackling problems involving ducks, training times shot up – the duck controller consumed the bulk of the processor time simply because there were so many ducks. Depending on the exact parameters, a generation took from thirty seconds to two or more minutes.

The second problem was with specifying fitness functions. I already mentioned the success of the "do nothing" tactic in avoiding walls; similarly degenerate solutions tended to crop up to more complex problems. The solution is to empirically tweak the relative scores assigned by the fitness function to specify the desired behaviour more exactly. The problem with this is that the speed of evolution makes each refinement cycle slow, so it can take a great deal of time to arrive at an effective fitness function.

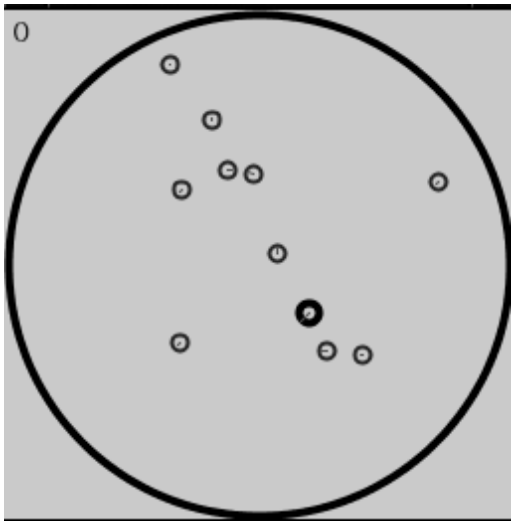
4.2.3 Herding

In spite of these setbacks, I eventually succeeded in producing a neural controller which can herd ducks, albeit imperfectly. The fitness function, `HerdDucksFitness`, is listed in Section B.1.8. The main points of interest are:

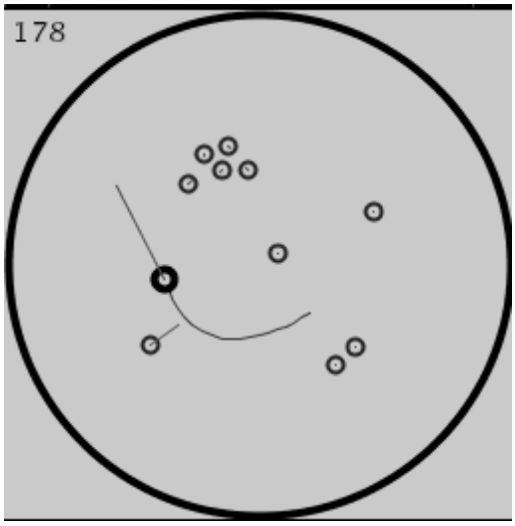
- When initializing environments, the simulation is run for about a hundred steps with no dog motion. This compensates for the ducks' natural tendency to flock even without a herder, by allowing them to settle down.
- As before, the dog is punished for hitting walls, and rewarding for moving.
- The actual herding criterion is calculated by summing the ducks' distances from the goal (here, the arena's centre). At the end of the training period the sum is recalculated. The decrease in total distance is multiplied by a suitable factor and added to the score.
- The inputs to the network were: the angle and distance to the nearest wall point; the angle and distance to the flock centre; and the maximum radius of the flock (i.e. the greatest distance of a duck from the flock centre). Thus the herding strategy was developed without knowledge of any individual duck positions.

In order to show the controller's behaviour over time, I instrumented the `ArenaView` class to draw a trail behind the dog. Figure 4.3 shows the result of 88 generations of training[2] with `HerdDucksFitness`. The results are not perfect, but are definitely promising. It is interesting that the "spiralling" strategy evolved is similar to that developed by Vaughan in [Vau99].

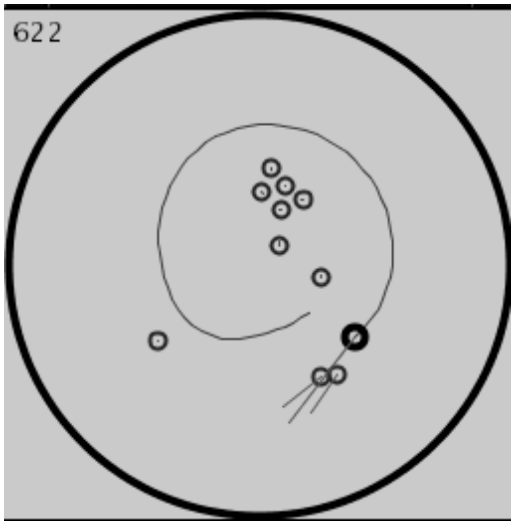
[2]The other parameters were: population size 48, hidden nodes 5, initial weight mean 0, initial weight range 4, mutation rate 4, training environments 40, and simulation steps per environment 80.



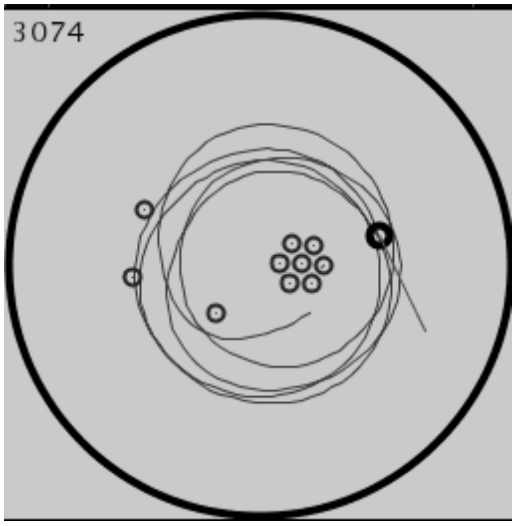
The dog starts in the middle...



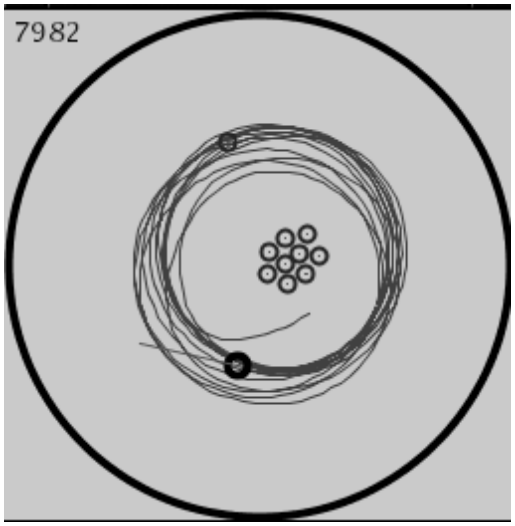
...and rapidly begins to spiral outwards.



The dog widens the spiral in an attempt to enclose all the ducks.



The radius of the spiral changes constantly in response to the flock size, but three ducks are still left out.



Eventually, all but one of the ducks have been successfully herded.

Figure 4.3: The genetically developed herding algorithm in action

Chapter 5

Conclusions

> A conclusion is just the place where you got tired of thinking.
>
> — Martin H. Fischer

5.1 Results of the project

A duck simulation framework The `ducksim` package provides a pleasantly modular, object-oriented framework for implementing and developing duck and herder control algorithms.

Reimplementation of previous work The core parts of the previous duck simulator were tidied up and re-implemented within the new framework.

The `neurotic` package A general-purpose package was written for creating and using neural networks (both feedforward and simple recurrent), with facilities for training them using back-propagation and genetic methods.

A dog controller Genetic training was applied to develop a controller capable of herding ducks into the centre of the arena.

5.2 What I learned from the project

Animal behaviour The background to the project taught me some basics of animal flight and flocking behaviour, how it may be encoded algorithmically, and how it may be utilised for herding.

Neural networks Prior to starting this project I knew very little about neural networks. The research in the initial stages gave me an overview of the field. Writing `neurotic` gave me a good understanding for feed-forward and simple recurrent networks, and of backpropagation. I also learned the principles of genetic algorithms, and about their application to neural networks.

Java, OOP, and OOD This is the largest project I have ever undertaken in Java and my first experience with Swing. The project gave me a chance to apply the object-oriented design techniques from the II.4 Object-Oriented Programming course. Both the duck simulation and the implementation of neural networks involved some careful object-oriented design (and sometimes redesign, after finding limitations in the initial implementation).

Research skills This was my first real experience of independent research. I have had to gather information from a wide variety of sources – books, journals, conference proceedings, technical reports, web pages, FTP sites and the assembled wisdom of the `comp.ai.neural-nets` newsgroup.

5.3 Areas for further investigation

Due to the limited time available for the project there were several areas I was unable to investigate:

Motivational units The previous dog algorithm, in contrast to the neural controller, worked in a large number of distinct stages or steps. Effectively the highest level was a finite state automaton, switching states when certain conditions in the simulation were met. Multi-stage behaviour such as "make the flock size sufficiently small, then move the flock towards the goal" is difficult for a neural network to model directly, but [CP92] suggests a solution in the form of a *motivational unit* – an extra input node which, rather than being influenced directly by the environment, is set by a user or a higher-level control algorithm. The network can learn different modes of behaviour depending on the state of the motivational unit.

Evolution of flocking behaviour This wasn't really a goal of the project, but could form an interesting extension. Flocking behaviour in real animals has evolved because it improves the animals' chances of survival, particularly when faced with predators. There have been successful attempts, such as [WD92], to recreate this evolution in neurally-controlled simulated animals. It would be interesting to evolve a neural controller for ducks in the presence of a simulated predator, and compare its behaviour with that of the duck algorithm constructed in [Vau99].

Faster training When I started using the genetic training algorithm it soon became clear that the slowness of training was the main obstacle to developing good genetic controllers. The main bottleneck seemed to be in the duck updating: I obtained a significant improvement by hand-optimizing the duck controller. Apart from using faster hardware or rewriting the whole thing in a fully compiled language, there are several feasible ways of improving performance: profiling and more optimizing; natively compiling speed-critical parts and interfacing them to the simulator with JNI; and running parallelized versions of backpropagation and genetic training on several machines at once (both these techniques lend themselves well to efficient parallelization).

The user interface The user interface was not intended to be a major part of the project, so I did not spend much time on it. There are many possible improvements, particularly regarding the genetic training algorithm. A user could be employed to supplement the fitness function by selecting and evaluating networks from the population. There is also scope for presenting the user with much more detailed information about the state of the population and of the individuals; an animated display of a working neural net might prove illuminating or, at any rate, pretty.

One neat enhancement would be using the Java reflection API to enable dynamic loading of dog and duck controllers; different controllers could be used without the need for recompilation. This would also speed up the development of algorithms.

Appendix A

Background information

A.1 Feedforward networks and backpropagation

A.1.1 Structure of feedforward networks

An artificial neural network is made up of a set of nodes $U = \{u_1, \ldots, u_n\}$. Any two nodes u_i and u_j may be connected by a directed arc, which has an associated real-valued weight $w_{ij} : i, j \in \{1, \ldots, n\}$. The backpropagation algorithm works with a particular type of *feedforward* network: in this network architecture, nodes are organised into layers, and a link from a node may only connect to a node in a layer strictly above it in the network. Thus the network is acyclic. An example of a three-layer feedforward network is shown in figure A.1.1.

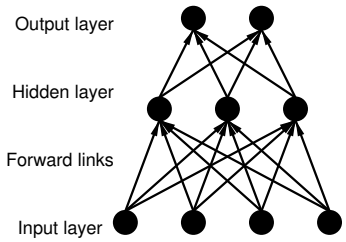


Figure A.1: An example of a feedforward network

Associated with each node is a number, its *activation*. For backpropagation networks this is usually a real in the range $[-1, 1]$ or $[0, 1]$. (My implementation uses $[0, 1]$.) The bottom layer in the network (the one into which no arcs are directed) is termed the *input layer*; the activations of its nodes are set externally and form the input of the network. Predictably, the topmost layer is called the *output layer* and the activations of its nodes will form the network's output.

For a node u_i not in the input layer, the activation is computed as a function of the activations of the nodes u_j for which links from u_j to u_i exist. A weighted sum is taken of the activations, and a predefined *activation function* f is applied to this sum to give u_i's activation. The activation $a(u_i)$ is given by

$$a(u_i) = f(\sum a(w_{ji})u_j)$$

There is one other factor in the activation of a node: its *bias*. This is simply a constant which is added to the weighted input sum before the activation function is applied. The easiest and most common way of implementing bias, and of allowing it to be modified by training algorithms, is to add an input node with a constant non-zero activation (usually 1) and make connections from it to all the hidden and output nodes in the network. The weights of these connections can then be adjusted by training algorithms just like any other weights in the network.

Since there is no restriction on the range of the weights and the activation must be in the range $[0, 1]$, the activation function must map $[-\infty, \infty]$ to $[0, 1]$. Additionally, for backpropagation to work, it must be monotonic and continuous. The most commonly used function (and the one I use) is the logistic function $f(x) = \frac{1}{1+e^{-x}}$.

The process of computing a function using a neural network simply consists of setting the input activations to the input values, then successively computing the activations for nodes in subsequent layers. The activations for the nodes in the final layer form the output.

A.1.2 The backpropagation algorithm

Backpropagation is an algorithm for adjusting the weights of a feedforward network in order to make it model a particular function. It minimizes error by gradient descent: starting from a random point in weight-space, it ascertains the gradient of the error function and adjusts the weights in that direction, working towards a minimum. Unfortunately the minimum is not guaranteed to be global, but it is nevertheless a powerful algorithm.

In outline, backpropagation works like this:

1. A training set is assembled, consisting of sample inputs for network, with a desired output for each input. The weights of the network are initialised to small random values.
2. The network is presented with an input from the training set, and a forward-propagation is run to produce output values.
3. The difference is calculated between the actual output values and the desired outputs from the training set. These error values are propagated backwards through the network, allowing changes in the weights of the connections to be calculated.
4. The weights of the connections are altered.
5. Steps 2-4 are repeated until the network models the desired function with sufficient accuracy.

In slightly more detail, the backprpagation pass works as follows:

1. Take the derivative of the activation function. If the usual logistic function is used, then the identity $f'(x) \equiv f(x)(1 - f(x))$ can be employed to speed up calculation by using the previously computed activations.
2. Associate with each node u_i a real number δ_i.
3. For each node u_i in the output layer, calculate δ_i by $\delta_i = (C_i - a_i)f'(S_i)$, where C_i is the correct target activation for that node, and S_i is the weighted sum used as input to the activation function (so actually $\delta_i = (C_i - a_i)a_i(1 - a_i)$).
4. Now move backwards through the network's hidden layers. For each node compute $\delta_i = (\sum_j w_{ij}\delta_j)f'(S_i)$, where the j range over all the nodes to which forward connections run from i. Since each δ_i value only depends on those of the nodes in front of it in the network, they can all be computed consistently in a single backward pass.
5. Finally, update each weight w_ij with the new value $w_ij + \rho\delta_j a_i$. ρ is the learning rate, usually in the range 0–1 (see below).

There are numerous variations of the algorithm, and several tunable parameters which affect its performance. Some of the main variations, most of which were incorporated into my simulation, are:

- The actual layout of the network. The number of input and output nodes is largely determined by the function which the network is being trained to compute, although in this project there is considerable choice as to how the inputs and outputs should be encoded (see Section 3.3). The number of hidden nodes, however, is an important tunable parameter: too few will result in the network being unable to model the desired function, and too many will make learning slow and unreliable.

  The number of layers is another tunable parameter, but I have kept it at three (four including the context nodes) for this project.

- The distribution of the initial random weights, which can have a significant effect on the solution which the network converges to.

- A *momentum* parameter can be added to the direction of gradient descent, allowing the algorithm to pick up speed if it makes many consecutive changes in the same direction. This can speed up learning significantly, but can also lead to minima being missed.

- The *learning rate* is a fixed factor governing how much the weights are adjusted at each step. Lower values give slower but more reliable convergence.

- Instead of updating the weights at each iteration (this is known as *stochastic* update), the δ_i values can be accumulated over many iterations, or over the whole training set, and only applied at the end (*batch* update).

A.2 Genetic training

Genetic algorithms are conceptually simple, but are widely applicable and can prove highly effective. The method is modelled on the principle of natural selection. It can be employed on virtually any problem where (1) the parameters of a possible solution can be encoded as a relatively compact sequence of numbers or symbols and (2) there exists a *fitness function* or *objective function* which can assign a numerical value reflecting how "good" a candidate solution is.[1]

Genetic algorithms come in numerous variations, but given a suitable problem the general procedure is as follows[2]:

1. Devise a way of encoding the parameters of a solution into a fixed-length string or numbers or symbols. Very often, binary digits are used, but this is by no means mandatory. The choice of encoding method can have a huge impact on the performance of the algorithm.

2. Initialize a set of randomly generated strings, called the *population* for obvious reasons.

3. Repeatedly apply the following three techniques:

   (a) *Reproduction*: Assess the fitness of each individual in the population, and produce a "weighted copy" of the current population. That is, initialize a new population of the same size by, for each string, copying a string selected at random from the old population. The selection is linearly weighted by fitness, so that fitter strings have a higher chance of being selected.

   (b) *Crossover*: select pairs of strings from the population. For each pair, randomly select a point within the length of a string. Break both strings at this point, and exchange their final segments. For example, `001122` and `222333` might be split and recombined halfway to give `001333` and `222122`.

[1] The satisfaction of these two criteria does not, of course, guarantee that a particular genetic algorithm will work!

[2] This summary is based on the descriptions given in [Was93] and [Win92].

(c) *Mutation*: perturb one or more strings. A string is selected at random, an element selected at random within that string, and a random modification applied to that element. Traditionally this method is used very sparingly; mutation rates of the order of one in a thousand are common.

The genetic algorithm used in this project deviates somewhat from this standard outline; the differences are described in Section 3.6.1.

Appendix B

Source code

B.1 `ducksim` package

B.1.1 DuckSim

```
package ducksim;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.geom.*;

class RunSim extends Thread {
  boolean running;
  Arena arena;
  int interval;
  public RunSim(Arena arena, int interval) {
    setDaemon(true);
    this.arena = arena;
    this.interval = interval;
    running = false;
  }
  public void run() {
    running = true;
    while (running) {
      arena.update();
      try { Thread.sleep(interval); }
      catch (InterruptedException e) {}
    }
  }
  public void pleaseStop() { running = false; }
}

public class DuckSim {

  Arena arena;

  class StepAction extends AbstractAction {
    public StepAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      arena.update();
    }
  }

  class StartStopAction extends AbstractAction {
    RunSim runsim;
    public StartStopAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      if (((String) getValue(NAME)).equalsIgnoreCase("Start")) {
        runsim = new RunSim(arena, 10);
        runsim.start();
        putValue(NAME,"Stop");
      } else {
        runsim.pleaseStop();
        putValue(NAME,"Start");
      }
    }
  }

  class ResetAction extends AbstractAction {
    public ResetAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      arena.resetRandom(true);
      arena.repaint();
    }
  }

  public static void main(String[] args)
  { DuckSim ducksim = new DuckSim(); }

  public DuckSim() {
    arena = new Arena(10);
    AnimalModel duckModel = new ChengDucks(arena);
    arena.setDuckModel(duckModel);
    AnimalModel dogModel = new DogBreeder(arena, arena);
    arena.setDogModel(dogModel);

    try {
      UIManager.setLookAndFeel
        (UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }

    // Control frame

    JToolBar toolbar = new JToolBar(JToolBar.VERTICAL);
    JFrame cframe = new JFrame("Duck control");
    cframe.getContentPane().add(toolbar);

    toolbar.setLayout(new GridLayout(3,1));
    toolbar.add(new StepAction("Step"));
    toolbar.add(new StartStopAction("Start"));
    toolbar.add(new ResetAction("Reset"));

    cframe.pack();
    cframe.setVisible(true);

    // Set windows to kill app on closure / Q keypress

    WindowAdapter killOnClose = new WindowAdapter() {
        public void windowClosing(WindowEvent e)
        { System.exit(0); }
      };
    cframe.addWindowListener(killOnClose);

    KeyListener killOnQ = new KeyAdapter() {
```

```
      public void keyTyped(KeyEvent e) {
        if (e.getKeyChar() == 'q' ||
            e.getKeyChar() == 'Q' ) System.exit(0);
      }
    };
  cframe.addKeyListener(killOnQ);
 }
}
```

B.1.2 Arena

```
// Arena: represents the current state of the duck arena.
//       Implements the *Control interfaces, through which the
//       DuckModel and DogModel classes interact with it.

package ducksim;

class Arena implements DuckControl, DogControl, TrainerControl {
  private int numDucks;
  private Pos[] duck;
  private int numDogs;
  private Pos[] dog;
  private java.util.Random rand = new java.util.Random();
  private static final float arenaRadius = 1000;
  private float flockR, flockMR;
  private Vec flockC;
  private boolean flUpToDate; // whether flock{C,R} are updated
  private int time;          // simulation time
  private AnimalModel duckModel;
  private AnimalModel dogModel;
  private ArenaView arenaView;

  Arena(int numDucks) {
    this.numDucks = numDucks;
    duck = new Pos[numDucks];
    flUpToDate = false;
    flockC = new Vec();
    time = 0;
    arenaView = new ArenaView(this);
    resetRandom(true);
  }

  public AnimalModel duckModel() { return duckModel; }
  public AnimalModel dogModel() { return dogModel; }
  public void startTraining() { arenaView.setActive(false); }
  public void stopTraining() { arenaView.setActive(true); }
  public void setDogModel(AnimalModel d) { dogModel = d; }
  public void setDuckModel(AnimalModel d) { duckModel = d; }
  public void repaint() { arenaView.repaint(); }
  public void resetRandom() { resetRandom(false); }
  public int time() { return time; }

  public void resetRandom(boolean setDog) {
    for (int i=0; i<numDucks; i++) {
      duck[i] = new Pos();
      randPoint(duck[i].p, 50f, arenaRadius-50);
      randPoint(duck[i].d, 0f, 0.5f);
    }
    this.numDogs = 1;
    dog = new Pos[numDogs];
    dog[0] = new Pos();
    if (setDog) {
      randPoint(dog[0].p, 50f, arenaRadius-50);
      randPoint(dog[0].d, 0f, 3.5f);
    } else {
      dog[0].p.x = 20;
      dog[0].p.y = 20;
    }
    time = 0;
    arenaView.reset();
  }

  private void randPoint(Vec p, float min, float max) {
    float r = rand.nextFloat()*(max-min) + min;
    float a = rand.nextFloat() * 2 * (float) Math.PI;
    p.x = r * (float) Math.cos(a);
    p.y = r * (float) Math.sin(a);
  }

  public float arenaSize() { return arenaRadius; }
  public float arenaArea() { return arenaRadius*arenaRadius; }
  public float dogRadius() { return arenaRadius/25; }
  public float duckRadius() { return arenaRadius/22; }
  public float goalSize() { return arenaRadius/22; }
  public int ducks() { return numDucks; }
  public int dogs() { return numDogs; }
  public Pos duck(int i) { return (Pos) duck[i].clone(); }
  public Vec duckPos(int i) { return (Vec) duck[i].p.clone(); }
  public Vec duckV(int i) { return (Vec) duck[i].d.clone(); }
  public Pos dog(int i) { return (Pos) dog[i].clone(); }
  public Vec dogPos(int i) { return (Vec) dog[i].p.clone(); }
  public Vec dogV(int i) { return (Vec) dog[i].d.clone(); }
  public void setDuck(int i, Vec v)
  { duck[i].d.x = v.x; duck[i].d.y = v.y; }
  public void setDog(int i, Vec v)
  { dog[i].d.x = v.x; dog[i].d.y = v.y; }
  public void setDongle(int i, float a) { dog[i].nAngle = a; }
  public void setDungle(int i, float a) { duck[i].nAngle = a; }
  public Pos[] cloneAllDogs() { return cloneArray(dog); }
  public Pos[] cloneAllDucks() { return cloneArray(duck); }

  private Pos[] cloneArray(Pos[] in) {
    Pos[] out = new Pos[in.length];
    for (int j=0; j<in.length; j++) out[j] = (Pos) in[j].clone();
    return out;
  }

  // omnipotent trainer methods

  public Pos[] getAllDogs() { return dog; }
  public Pos[] getAllDucks() { return duck; }
  public void setAllDogs(Pos[] newDog) { dog = newDog; }
  public void setAllDucks(Pos[] newDuck) { duck = newDuck; }

  // this is actually a class method, but if I declared it static
  // I couldn't put it in the interface.
  // (inherited from last year's algorithms; I don't use it.)
  public float calcAngle(Vec f) {
    float r = 0.0f; // to prevent "variable not initialized"
    // Matt's updated angle code in floats and radians
    // scope for optimisation here but not going into it yet
    float theta = (float) Math.atan(f.x/f.y); // note sign change!!
    if (f.x > 0 && f.y >0)   r = theta;
    else if (f.x > 0 && f.y < 0) r = (float) Math.PI + theta;
    else if (f.x < 0 && f.y < 0) r = (float) Math.PI + theta;
    else if (f.x < 0 && f.y > 0) r = (float) Math.PI*2 + theta;
    else if (f.x == 0) if (f.y >= 0) r = 0.0f; else r = (float) Math.PI;
    return r;
  }

  public void advance() {
    for (int i=0; i<ducks(); i++) duck[i].move();
    for (int i=0; i<dogs(); i++) dog[i].move();
    flUpToDate = false;
  }

  public void update() {
    time++;
    duckModel.update();
    dogModel.update();
    advance();
    duckModel.postUpdate();
    dogModel.postUpdate();
    arenaView.repaint();
  }

  public float flockRadius() {
    if (!flUpToDate) calcFlockData();
    return flockR;
  }

  public float flockMeanRadius() {
    if (!flUpToDate) calcFlockData();
    return flockMR;
  }

  public Vec flockCentre() {
```

```
    if (!flUpToDate) calcFlockData();
    return (Vec) flockC.clone();
  }

  private void calcFlockData() {
    float nDist;
    flockC.x = flockC.y = 0;
    for (int i=0; i<numDucks; i++) flockC.pleq(duck[i].p);
    flockC.dieq(numDucks);
    flockR = flockMR = 0;
    for (int i=0; i<numDucks; i++) {
      nDist = flockC.distTo(duck[i].p);
      flockR = flockR >= nDist ? flockR : nDist;
      flockMR += nDist;
    }
    flockR += duckRadius();
    flockMR /= numDucks;
    flUpToDate = true;
  }
}

// Abstract animal-model classes, and interfaces through which
// they should call Arena's methods.

interface CommonControl {
  float arenaSize();
  float arenaArea();
  float dogRadius();
  float duckRadius();
  float goalSize();
  int ducks();
  int dogs();
  Pos duck(int duck); // duck position & velocity
  Vec duckPos(int duck); // duck position
  Vec duckV(int duck); // duck velocity
  Pos dog(int dog); // dog position & velocity
  Vec dogPos(int dog); // dog position
  Vec dogV(int dog); // dog velocity
  float calcAngle(Vec v);
  float flockRadius();
  Vec flockCentre();
  float flockMeanRadius();
  Pos[] cloneAllDogs();
  Pos[] cloneAllDucks();
}

// Omnipotent control for training purposes
interface TrainerControl extends DogControl, DuckControl {
  void resetRandom();
  void resetRandom(boolean setDog);

  // Q: why are these "get" methods relegated to the Trainer i/face?
  // A: They return the actual array references (not clones), hence
  //    allow teleportation &c (makes setAllXs kinda pointless, maybe?)
  Pos[] getAllDogs();
  Pos[] getAllDucks();

  void setAllDogs(Pos [] p);
  void setAllDucks(Pos [] p);
  void setDuck(int duck, Vec velocity);
  void setDungle(int duck, float angle);
  void setDog(int dog, Vec velocity);
  void setDongle(int dog, float angle);
  void advance();
  void startTraining();
  void stopTraining();
  AnimalModel duckModel();
  AnimalModel dogModel();
}

interface DuckControl extends CommonControl {
  void setDuck(int duck, Vec velocity);
  void setDungle(int duck, float angle);
}

interface DogControl extends CommonControl {
  void setDog(int dog, Vec velocity);
  void setDongle(int dog, float angle);
}
```

```
// This class is used for both the dog and duck model.
// The difference is that each should only see Arena through
// the appropriate interface.

abstract class AnimalModel {
  abstract void update();
  abstract void postUpdate();
}
```

B.1.3 ArenaView

```
// ArenaView: view class for Arena.

package ducksim;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.font.*;

class ArenaView extends JFrame {

  ArenaPanel p;             // the actual arena display
  private boolean active; // whether we should be updating
  static boolean trace = true;

  public ArenaView(Arena a) {
    super("Duck Display");
    active = true;
    p = new ArenaPanel(a);
    setContentPane(p);
    p.reset();
    pack();
    show();
  }

  public void setActive(boolean a) { active = a; repaint(); }
  public boolean isActive() { return active; }
  public void reset() { p.reset(); }

  class ArenaPanel extends JPanel {

    // whether to produce a trace of the dog's motion
    GeneralPath tracePath;
    Arena arena;
    Font f = new Font("SansSerif",Font.PLAIN,14);
    int nextPathUpdate;

    ArenaPanel(Arena a) {
      super();
      arena = a;
      setSize(256, 256);
      setPreferredSize(new Dimension(256, 256));
      tracePath = trace ? new GeneralPath(0, 1000) : null;
    }

    public void reset() {
      tracePath.reset();
      nextPathUpdate = 0;
    }

    public void paintComponent(Graphics oldGraphics) {
      Graphics2D g = (Graphics2D) oldGraphics;
      int size = getWidth()<getHeight() ? getWidth() : getHeight();
      int centre = size/2;
      int radius = centre;

      g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
      clear(g);

      g.setColor(Color.black);
      g.setFont(f);
      g.setStroke(new BasicStroke(4));
```

```
    g.draw(new Ellipse2D.Float(2,2,size-4,size-4));

    if (active) {
      if (trace) {
        final float x = arena.dogPos(0).x;
        final float y = arena.dogPos(0).y;
        g.setColor(Color.darkGray);
        g.setStroke(new BasicStroke(1));
        if (arena.time()>=nextPathUpdate) {
          if (nextPathUpdate==0) tracePath.moveTo(x,y);
          else tracePath.lineTo(x,y);
          nextPathUpdate += 10;
        }
        AffineTransform a = new AffineTransform();
        a.translate(centre, centre);
        a.scale(radius/arena.arenaSize(), radius/arena.arenaSize());
        g.draw(tracePath.createTransformedShape(a));
      }
      for (int i=0; i<arena.ducks(); i++)
        drawDuck(size, g, arena.duck(i));
      for (int i=0; i<arena.dogs(); i++)
        drawDog(size, g, arena.dog(i));
      g.setColor(Color.black);
      g.drawString(String.valueOf(arena.time()), 4, 16);
    } else {
      drawText(g, "Display inactive");
    }

  }

  private void drawText(Graphics2D g, String text) {
    Rectangle2D box = getBounds();
    FontRenderContext frc = g.getFontRenderContext();
    Rectangle2D bounds = f.getStringBounds(text,frc);
    LineMetrics metrics = f.getLineMetrics(text,frc);
    float width = (float) bounds.getWidth();
    float lineheight = metrics.getHeight();
    float ascent = metrics.getAscent();
    float x0 = (float) (box.getX() + (box.getWidth() - width)/2);
    float y0 = (float) (box.getY() + (box.getHeight() - lineheight)/2
                        + ascent);
    g.drawString(text, x0, y0);
  }

  private void drawDog(int size, Graphics2D g, Pos p) {
    final float r = size / 50;
    final float x = (size/2) + p.p.x * (size/2) / arena.arenaSize();
    final float y = (size/2) + p.p.y * (size/2) / arena.arenaSize();

    g.setColor(Color.black);
    g.setStroke(new BasicStroke(4));
    g.draw(new Ellipse2D.Float(x-r, y-r, 2*r, 2*r));
    g.setColor(Color.red);
    g.setStroke(new BasicStroke(1));
    g.draw(new Line2D.Float(x, y,
                            x+ r*p.d.x * 2f,
                            y + r*p.d.y * 2f ));

  }

  private void drawDuck(int size, Graphics2D g, Pos p) {
    final float r = size / 60;
    final float x = (size/2) + p.p.x * (size/2) / arena.arenaSize();
    final float y = (size/2) + p.p.y * (size/2) / arena.arenaSize();

    g.setColor(Color.blue);
    g.setStroke(new BasicStroke(2));
    g.draw(new Ellipse2D.Float(x-r, y-r, 2*r, 2*r));
    g.setColor(Color.red);
    g.setStroke(new BasicStroke(1));
    g.draw(new Line2D.Float(x, y,
                            x+ r*p.d.x*2,
                            y + r*p.d.y*2));
  }

  // super.paintComponent clears offscreen pixmap,
  // since we're using double buffering by default.
  protected void clear(Graphics g) {
    super.paintComponent(g);
  }
```

```
  }

}
```

B.1.4 ChengDucks

```
/* ChengDucks: the duck model, developed by Richard Cheng
   last year from Richard Vaughan's original algorithm, now translated
   from C++ to Java and tidied up for this project. */

package ducksim;

class ChengDucks extends AnimalModel {

  /* NB: The arena radius is hardwired to 1000. There's not really
     any reason to change this for the purposes of this project, so
     it makes sense to use final statics and gain some extra speed. */

  final static float ARENASIZE = 1000;
  final static float ARENAAREA = ARENASIZE * ARENASIZE;
  final static float DUCKRADIUS = ARENASIZE / 22;
  final static float K1 = ARENAAREA/10;
  final static float K2 = ARENAAREA/100;
  final static float K3 = ARENAAREA/100;
  final static float K4 = ARENAAREA/1; // was ARENAAREA/10, rich
  final static float L = ARENASIZE/6; // was ARENASIZE/4, rich
  final static float WALLDIST = DUCKRADIUS + 15;
  // for duck-wall repulsion
  final static float MOMENTUM = 0.8f;
  // *new* proportion of duck's previous movement vector, rich
  final static float K5 = 1000*1000/10;
  // for duck-duck attraction in new1
  final static float ATTRACTDIST = 1000/4;
  // for duck-duck attraction in new2
  final static float FLIGHTDIST = 1000/4;
  // for duck-herder repulsion
  final static float SUCCDIST = 175;
  final static float DUCKTOPSPEED = 4; // a duck's top speed
  int nFlockRadius;
  DuckControl state;

  ChengDucks(DuckControl state) {
    this.state = state;
  }

  void update()
  {
    // Vec vArenaCentre = new Vec(0,0);
    // We can ignore this since it's zero; speed is more important
    // for now.

    Vec totforce, wallpt, duckpos, otherduckpos, herderpos, duckV, endpt;

    for (int i = 0; i<state.ducks(); i++) {
      float dist, force;
      duckpos = state.duckPos(i);
      totforce = new Vec(0,0);

      // *new* duck attraction instinct 2 (limited range attraction)
      for (int j = 0; j < state.ducks(); j++) {
        if (j == i) continue;

        otherduckpos = state.duckPos(j);
        float attforce, repforce;

        dist = otherduckpos.mi(duckpos).abs();
        attforce = K1 / ((dist + L) * (dist + L));
        repforce = - K2 / (dist * dist);

        attforce -= K1 / ((ATTRACTDIST + L) * (ATTRACTDIST + L));
        repforce -= - K2 / (ATTRACTDIST * ATTRACTDIST);

        if (attforce < 0.0) attforce = 0.0f;
        if (repforce > 0.0) repforce = 0.0f;
        force = attforce + repforce;
        totforce.pleq( otherduckpos.mi(duckpos).unit().ti(force) );
```

```
      }

      wallpt = duckpos.unit().ti(ARENASIZE);
      dist = wallpt.distTo(duckpos);
      force = - K3 / (dist * dist);
      force -= - K3 / (WALLDIST * WALLDIST);
      if (force > 0) force = 0.0f;
      totforce.pleq( duckpos.dirTo(wallpt).ti(force) );

      // *new* herder repulsion instinct
      for (int j = 0; j < state.dogs(); j++) {
        herderpos = state.dogPos(j);
        dist = herderpos.distTo(duckpos);
        force = - K4 / (dist * dist);
        force -= - K4 / (FLIGHTDIST * FLIGHTDIST);
        if (force > 0.0) force = 0.0f;
        totforce.pleq(duckpos.dirTo(herderpos).ti(force));
      }

      duckV = state.duck(i).d.ti(MOMENTUM).
          pl(totforce.ti(1-MOMENTUM));

      // state.setDuck(i,state.duck(i).d.ti(MOMENTUM).
      //           pl(totforce.ti(1-MOMENTUM)));

      // duck speed limiter
      // if (duckV.abs() > 3.5f) duckV.dieq(3.5f);
      if (duckV.abs() > 3.5f) duckV.tieq(3.0f / duckV.abs());

      // richie's duck wall avoidance code.
      endpt = duckpos.pl(duckV);
      if (endpt.abs() > ARENASIZE/2 - 2) {
        wallpt = endpt.unit().ti(ARENASIZE/2-5);
        duckV.mieq( endpt.mi(wallpt).ti(0.01f) );
      }
      state.setDuck(i, duckV);
    }
  }

  void postUpdate() {}
}
```

B.1.5 DogBreeder

```
// DogBreeder

/* Framework for genetically developing neural networks for dog
   control. Includes a user interface, DbPanel, as an inner class.
   Servant classes are used for the fitness function and for the
   actual network / dog-control interface. */

package ducksim;

import neurotic.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;

public class DogBreeder extends AnimalModel {

  DogControl dogCont;
  TrainerControl trainCont;
  Net net;
  Population pop;
  DbPanel controlPanel;
  Fitness fitness;
  int hiddens, popn, mean, range, muteRate, envs, tSteps;
  final static float PI = (float) Math.PI;
  ControlInterface contInt;;

  public DogBreeder(DogControl dogCont,
                    TrainerControl trainCont) {
    this.dogCont = dogCont;
    this.trainCont = trainCont;
    contInt = new DogControlIface(dogCont, trainCont);
    reInit(contInt.inputs(), 48, 0, 4, 4, 20, 64);
    fitness = new HerdDucksFitness(trainCont, tSteps, this, new Vec());
    // fitness = new AvoidWallsFitness(trainCont, tSteps, this, new Vec());
    createNets();
    controlPanel = new DbPanel(this);
    useCurrentBest();
  }

  void reInit(int hiddens, int popn, int mean, int range,
              int muteRate, int envs, int tSteps) {
    this.hiddens = hiddens; this.popn = popn;
    this.mean = mean; this.range = range;
    this.muteRate = muteRate; this.envs = envs;
    this.tSteps = tSteps;
  }

  void createNets() {
    net = new Net(contInt.inputs(), hiddens, 2, true);
    pop = new Population(net, fitness, popn, mean,
                         range, muteRate, envs);
  }

  void evolve() { pop.evolve(); }
  void evolve(int n) { pop.evolve(n); }
  int getCurrGen() { return pop.getCurrGen(); }
  int getPopn() { return pop.popSize(); }
  float getMaxFitness() { return pop.fitness[pop.fittest]; }
  float getMinFitness() { return pop.fitness[pop.leastFit]; }

  public interface ControlInterface {
    void update(Net n, DogControl c);
    int inputs();
  }

  void update(Net n, DogControl c) { contInt.update(n, c); }
  void update() { update(net, dogCont); }

  void postUpdate() { }

  void useCurrentBest() {
    net.setAllWeights(pop.weights[pop.fittest]);
  }

  // Save current parameters and weights to a file
  void save(File f) {
    try {
      Writer w = new FileWriter(f);
      w.write(hiddens +" "+ popn +" "+ mean +" "+ range +" "+
              muteRate +" "+ envs +" "+ tSteps + "\n");
      pop.save(w);
    }
    catch (IOException e) { throw new Error(e.getMessage()); }
  }

  // Load parameters and weights from a file
  // format: hiddens, popn, mean, range, muteRate, envs, tSteps
  void load(File f) {
    try {
      Reader reader = new FileReader(f);
      StreamTokenizer in = new StreamTokenizer(reader);

      hiddens = readInt(in);
      popn = readInt(in);
      mean = readInt(in);
      range = readInt(in);
      muteRate = readInt(in);
      envs = readInt(in);
      tSteps = readInt(in);
      net = new Net(contInt.inputs(), hiddens, 2, true);
      pop = new Population(net, fitness, envs, reader);
      useCurrentBest();
      controlPanel.updateFields();
    }
    catch (IOException e) { throw new Error(e.getMessage()); }
  }

  private int readInt(StreamTokenizer i)
    throws java.io.IOException
  { i.nextToken(); return (int) i.nval; }
```

```
}

class DbPanel extends JFrame {

  JLabel popL, maxL, minL, genL, stepL;
  JTextField popF, maxF, minF, genF, stepF;
  DogBreeder db;
  JToolBar toolbar;
  java.util.Timer timer;

  DbPanel(DogBreeder db) {
    super("Dog breeder control");
    this.db = db;
    popL = new JLabel("Population");
    maxL = new JLabel("Max. fitness");
    minL = new JLabel("Min. fitness");
    genL = new JLabel("Generation");
    stepL = new JLabel("Gen. step");
    popF = new JTextField();
    maxF = new JTextField();
    minF = new JTextField();
    genF = new JTextField();
    stepF = new JTextField("256",5);
    updateFields();

    JPanel p = new JPanel();
    p.setLayout(new GridLayout(5,2,4,4));
    p.add(popL); noEdit(popF); p.add(popF);
    p.add(maxL); noEdit(maxF); p.add(maxF);
    p.add(minL); noEdit(minF); p.add(minF);
    p.add(genL); noEdit(genF); p.add(genF);
    p.add(stepL); p.add(stepF);
    getContentPane().add(p, BorderLayout.NORTH);

    toolbar = new JToolBar();
    toolbar.setBorder(new EtchedBorder());
    toolbar.add(new EvolveStepAction("Evolve"));
    toolbar.add(new StartStopAction("Start"));
    toolbar.add(new ReInitAction("Reinit"));
    toolbar.add(new LoadSaveAction("Load"));
    toolbar.add(new LoadSaveAction("Save"));

    getContentPane().add(toolbar, BorderLayout.SOUTH);

    pack(); setVisible(true);
    timer = new java.util.Timer(true); // kill timer when we exit
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  }

  private void noEdit(JTextField t) {t.setEditable(false);}

  void updateFields() {
    popF.setText(String.valueOf(db.getPopn()));
    maxF.setText(String.valueOf(db.getMaxFitness()));
    minF.setText(String.valueOf(db.getMinFitness()));
    genF.setText(String.valueOf(db.getCurrGen()));
  }

  class EvolveStepAction extends AbstractAction {
    public EvolveStepAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      int step;
      try { step = Integer.parseInt(stepF.getText()); }
      catch (NumberFormatException ex)
        { stepF.setText(String.valueOf(step = 1)); }
      db.evolve(Integer.parseInt(stepF.getText()));
      db.useCurrentBest();
      updateFields();
    }
  }

  class StartStopAction extends AbstractAction {

    Evolve evolution;
    UpdateGen updateGen;
    public StartStopAction(String t) {
      super(t);
      // putValue("NAME","Start");
    }
    public void actionPerformed(ActionEvent e) {
      if (((String) getValue(NAME)).equalsIgnoreCase("Start")) {
        // timer.schedule(evolve = new Evolve(), 0, 2);
        evolution = new Evolve();
        timer.schedule(updateGen = new UpdateGen(), 0, 1000);
        putValue(NAME,"Stop");
        db.trainCont.startTraining();
      } else {
        updateGen.cancel();
        evolution.pleaseStop();
        putValue(NAME,"Start");
        db.trainCont.stopTraining();
      }
    }
  }

  class ReInitAction extends AbstractAction {
    public ReInitAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      SetupFrame setupFrame = new SetupFrame();
      setupFrame.show();
    }
  }

  class LoadSaveAction extends AbstractAction {
    public LoadSaveAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      JFileChooser chooser = new JFileChooser();
      if (((String) getValue(NAME)).equalsIgnoreCase("Load")) {
        if (chooser.showOpenDialog(DbPanel.this) == 0)
          db.load(chooser.getSelectedFile());
      } else if (chooser.showSaveDialog(DbPanel.this) == 0)
        db.save(chooser.getSelectedFile());
    }
  }

  class Evolve extends Thread {
    boolean running;
    public Evolve() {
      setDaemon(true);
      running = true;
      start();
    }
    public void run() {
      while (running) { db.evolve(); updateFields(); }
      db.useCurrentBest();
    }
    public void pleaseStop() { running=false; }
  }

  class UpdateGen extends java.util.TimerTask {
    public void run()
    { genF.setText(String.valueOf(db.getCurrGen())); }
  }

  class SetupFrame extends JFrame {
    JToolBar stoolbar;
    JTextField hidF, popF, meanF, rangeF, muteF, envsF, stepsF;

    SetupFrame() {
      super("Dog Breeder Setup");

      hidF = new JTextField(String.valueOf(db.hiddens));
      popF = new JTextField(String.valueOf(db.popn));
      meanF = new JTextField(String.valueOf(db.mean));
      rangeF = new JTextField(String.valueOf(db.range));
      muteF = new JTextField(String.valueOf(db.muteRate));
      envsF = new JTextField(String.valueOf(db.envs));
      stepsF = new JTextField(String.valueOf(db.tSteps));

      JPanel p = new JPanel();
      p.setLayout(new GridLayout(7,2,4,4));
      p.add(new JLabel("Hidden nodes")); p.add(hidF);
      p.add(new JLabel("Population")); p.add(popF);
      p.add(new JLabel("Weight mean")); p.add(meanF);
      p.add(new JLabel("Weight range")); p.add(rangeF);
      p.add(new JLabel("Mut'n rate")); p.add(muteF);
      p.add(new JLabel("Evironments")); p.add(envsF);
      p.add(new JLabel("Sim'n length")); p.add(stepsF);
      getContentPane().add(p, BorderLayout.NORTH);
```

```
    toolbar = new JToolBar();
    toolbar.setBorder(new EtchedBorder());
    toolbar.add(new CancelAction("Cancel"));
    toolbar.add(new OkAction("ReInit"));
    getContentPane().add(toolbar, BorderLayout.SOUTH);
    pack();
  }

  class CancelAction extends AbstractAction {
    public CancelAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      dispose();
    }
  }

  class OkAction extends AbstractAction {
    public OkAction(String t) { super(t); }
    public void actionPerformed(ActionEvent e) {
      db.reInit(getInt(hidF, 4), getInt(popF, 48),
                getInt(meanF, 0), getInt(rangeF, 4),
                getInt(muteF, 4), getInt(envsF, 20),
                getInt(stepsF, 32));
      db.createNets();
      updateFields();
      dispose();
    }
    private int getInt(JTextField t, int d) {
      return Integer.parseInt(t.getText());
    }
  }

 }

}
```

B.1.6 DogControlIface

```
// DogControlIface

// a class to interface neural network inputs and outputs to
// the DogControl interface

package ducksim;
import neurotic.*;

class DogControlIface implements DogBreeder.ControlInterface {

  DogControl dogCont;
  TrainerControl trainCont;
  final static float PI = (float) Math.PI;
  final static int inputs = 5; // # input nodes

  DogControlIface(DogControl dogCont, TrainerControl trainCont) {
    this.dogCont = dogCont;
    this.trainCont = trainCont;
  }

  public int inputs() { return inputs; }

  public void update(Net net, DogControl control) {
    Pos dogPos = control.dog(0);
    Vec wall = wallPoint(dogPos.p); // nearest wall point

    net.setIn(0, normAngle(wall.angle() - dogPos.d.angle()));
    // Angle to nearest wall point
    net.setIn(1, dogPos.p.distTo(wall) / control.arenaSize());
    // distance to nearest wall point
    net.setIn(2, normAngle( dogPos.p.angleTo(control.flockCentre())
                            - dogPos.d.angle()));
    // angle to flock centre
    net.setIn(3, dogPos.p.distTo(control.flockCentre()) /
              control.arenaSize());
    // distance to flock centre
    net.setIn(4, 2 * control.flockRadius() / control.arenaSize());
    // size of flock
    net.think();
    float theta = dogPos.d.angle();
    float r = dogPos.d.abs();
    r = net.getOut(0) * 16;
    theta += 0.9 * (net.getOut(1) - 0.5f); // * PI
    dogPos.d.x = (float) (r * Math.sin(theta));
    dogPos.d.y = (float) (r * Math.cos(theta));
    control.setDog(0, dogPos.d);
  }

  // map angle in radians onto [0,1]
  // -pi |-> 0, +pi |-> 1, values outside [-pi,pi] wrap.
  private float normAngle(float theta) {
    float out = ( (theta + PI) / (2*PI) ) % 1.0f;
    if (out<0) out += 1.0f;
    return out;
  }

  // find nearest point on arena wall
  Vec wallPoint(Vec v) {
    return v.unit().ti(trainCont.arenaSize()).mi(v);
  }

}
```

B.1.7 AvoidWallsFitness

```
// AvoidWallsFitness

// Fitness function which awards score for moving and for avoiding walls

package ducksim;

import neurotic.*;

class AvoidWallsFitness implements Fitness {
  class Enviro {
    Pos[] dogs; Pos[] ducks;
    // NB: vars not init'ed, since we'll just clone into them
  }
  Enviro[] envs;              // environments to train
  TrainerControl trainCont; // i/face to arena
  int tSteps;                 // #steps per training enviro.
  DogBreeder db;
  Pos[] duck;
  int numDucks;
  AnimalModel duckModel;
  Vec goal;

  public AvoidWallsFitness(TrainerControl t, int s, DogBreeder d,
                           Vec g) {
    trainCont = t;
    tSteps = s;
    db = d;
    goal = g;
    duck = trainCont.getAllDucks();
    numDucks = duck.length;
    duckModel = trainCont.duckModel();
  }

  // Initialise environments
  public void initEnviros(int n) {
    envs = new Enviro[n];

    for (int i=0; i<n; i++) {
      trainCont.resetRandom(true);
      envs[i] = new Enviro();
      envs[i].dogs = trainCont.cloneAllDogs();
      envs[i].ducks = trainCont.cloneAllDucks();
    }
  }

  public float score(Net net, int env) {
    trainCont.setAllDogs(envs[env].dogs);
    trainCont.setAllDucks(envs[env].ducks);

    int i;
```

```
    float score = 50000;
    Pos oldPos = trainCont.dog(0), newPos;
    Pos origPos = (Pos) oldPos.clone();
    float improvement = 0;
    final float r = trainCont.arenaSize() - 30;

    for (i=0; i<tSteps; i++) {
      // duckModel.update();
      /* I've commented this out to speed things up, since
         the dog is (or should be) ignoring the ducks anyway. */
      db.update(net, trainCont);
      trainCont.advance();
      newPos = trainCont.dog(0);

      if (newPos.p.abs() > r - 50) score -= 20000;
      // impose a heavy penalty for hitting a wall
      float a = newPos.d.abs();
      if (a > 4.0) score -= a * 20;
      // penalise dog for moving too fast
      oldPos = newPos;
    }

    // Reward the dog for moving, to avoid falling into the
    // degenerate "do nothing" solution
    score += (origPos.p.distTo(oldPos.p)) * 50;

    return score;
  }

}
```

B.1.8 HerdDucksFitness

```
// HerdDucksFitness

// Fitness function which assigns good scores for decreasing the flock
// size.

package ducksim;

import neurotic.*;

class HerdDucksFitness implements Fitness {
  class Enviro {
    Pos[] dogs; Pos[] ducks;
    // NB: vars not init'ed, since we'll just clone into them
  }
  Enviro[] envs;              // environments to train
  TrainerControl trainCont; // i/face to arena
  int tSteps;                 // #steps per training enviro.
  DogBreeder db;
  Pos[] duck;
  int numDucks;
  AnimalModel duckModel;
  Vec goal;

  public HerdDucksFitness(TrainerControl t, int s, DogBreeder d,
                          Vec g) {
    trainCont = t;
    tSteps = s;
    db = d;
    goal = g;
    duck = trainCont.getAllDucks();
    numDucks = duck.length;
    duckModel = trainCont.duckModel();
  }

  // Initialise environments
  public void initEnviros(int n) {
    envs = new Enviro[n];

    for (int i=0; i<n; i++) {
      trainCont.resetRandom(true);
      envs[i] = new Enviro();
      envs[i].dogs = trainCont.cloneAllDogs();
      trainCont.setDog(0, new Vec(0,0));
      /* Ducks tend to group together naturally, even without
         being herded. Therefore I run the simulation for a bit
         without moving the dog, in order to let them settle a bit
         and get a more neutral starting state. */
      for (int j=0; j<128; j++) {
        duckModel.update();
        trainCont.advance();
        duckModel.postUpdate();
      }
      envs[i].ducks = trainCont.cloneAllDucks();
    }
  }

  public float score(Net net, int env) {
    trainCont.setAllDogs(envs[env].dogs);
    trainCont.setAllDucks(envs[env].ducks);

    int i;
    float score = 50000;
    Pos oldPos = trainCont.dog(0), newPos;
    Pos origPos = (Pos) oldPos.clone();
    float improvement = 0;
    final float r = trainCont.arenaSize() - 30;

    // Measure initial total distance from goal (i.e. centre)
    for (i=0; i<numDucks; i++)
      improvement += duck[i].p.abs();

    for (i=0; i<tSteps; i++) {
      duckModel.update();
      db.update(net, trainCont);
      trainCont.advance();
      duckModel.postUpdate();
      newPos = trainCont.dog(0);

      if (newPos.p.abs() > r - 50) score -= 20000;
      // impose a heavy penalty for hitting a wall
      float a = newPos.d.abs();
      if (a > 6.0) score -= a * 20;
      // penalise dog for moving too fast
      oldPos = newPos;
    }

    // Subtract new total distance from goal,
    // and calculate score accordingly
    for (i=0; i<numDucks; i++)
      improvement -= duck[i].p.abs();
    score += improvement * 10000;

    // Reward the dog for moving, to avoid falling into the
    // degenerate "do nothing" solution
    score += (origPos.p.distTo(oldPos.p)) * 2;

    return score;
  }

}
```

B.1.9 Vec

```
package ducksim;

class Vec implements Cloneable {
  float x,y;
  Vec(float x, float y) {
    this.x = x;
    this.y = y;
  }
  Vec() { this.x = this.y = 0; }

  float abs()
  { return (float) Math.sqrt(x*x + y*y); }
  float distTo(Vec v)
  { return (float) Math.sqrt((x-v.x)*(x-v.x)+(y-v.y)*(y-v.y)); }
  Vec dirTo(Vec v) {
    float dx, dy, a;
    dx = v.x-x; dy = v.y-y;
    a = (float) Math.sqrt(dx*dx+dy*dy);
    if (a==0) throw new Error("Can't normalize a zero vector!");
    return new Vec(dx/a, dy/a);
  }

  // angle in radians: straight down = 0, straight up = +/- pi
  float angle() { return (float) Math.atan2(x,y); }
  float angleTo(Vec v) { return (float) Math.atan2(v.x-x,v.y-y); }
  float angleFrom(Vec v) { return (float) Math.atan2(x-v.x,y-v.y); }

  Vec unit() {
    float a = (float) Math.sqrt(x*x + y*y);
    if (a==0) throw new Error("Can't normalize a zero vector!");
    // if (a==0) return new Vec(0,0);
    return new Vec(x/a, y/a); }
  Vec norm() {
    float a = (float) Math.sqrt(x*x + y*y);
    if (a==0) throw new Error("Can't normalize a zero vector!");
    x/=a; y/=a; return this;
  }
  Vec ti(float a)  { return new Vec(x*a, y*a); }
  Vec tieq(float a) { x*=a; y*=a; return this; }
  Vec di(float a)  { return new Vec(x/a, y/a); }
  Vec dieq(float a) { x/=a; y/=a; return this; }
  Vec pl(float a)  { return new Vec(x+a, y+a); }
  Vec pl(Vec a)    { return new Vec(x+a.x, y+a.y); }
  Vec pleq(Vec a) { x+=a.x; y+=a.y; return this; }
  Vec mi(float a) { return new Vec(x-a, y-a); }
  Vec mieq(float a) { x-=a; y-=a; return this; }
  Vec mi(Vec a)   { return new Vec(x-a.x, y-a.y); }
  Vec mieq(Vec a) { x-=a.x; y-=a.y; return this; }

  boolean equals(float e, Vec v) {
    if (Math.abs(v.x-x)<e && Math.abs(v.y-y)<e) return true;
    return false;
  }

  static Vec add(Vec a, Vec b)
  { return new Vec(a.x+b.x, a.y+b.y); }
  static Vec add(Vec a, float b)
  { return new Vec(a.x+b, a.y+b); }
  static Vec add(float a, Vec b)
  { return new Vec(a+b.x, a+b.y); }
  static Vec sub(Vec a, Vec b)
  { return new Vec(a.x-b.x, a.y-b.y); }

  public Object clone() {
    try {return super.clone();}
    catch (CloneNotSupportedException e) {
      // this should never happen
      throw new InternalError("EEK! " + e.toString());
    }
  }
}
```

B.1.10 Pos

```
// Pos: a class to encapsulate the position and velocity of
//      an entity in the simulation

package ducksim;

class Pos implements Cloneable {
  Vec p, d;
  float nAngle;
  /* the nAngle field is used by last year's algorithms. I'm
     not sure exactly what it represents, and I don't use it. */
  Pos() {
    p = new Vec();
    d = new Vec();
  }
  void move() {
    p.x += d.x;
    p.y += d.y;
  }
  public Object clone() {
    Pos posCopy;
    try {posCopy = (Pos) super.clone();}
    catch (CloneNotSupportedException e) {
      // this should never happen
      throw new InternalError("ERK! " + e.toString());
    }
    posCopy.p = (Vec) p.clone();
    posCopy.d = (Vec) d.clone();
    return posCopy;
  }
}
```

B.2 `neurotic` package

B.2.1 Axon

```
package neurotic;

/* An Axon is either a neuron or an input node: anything
   that can provide an input to a neuron. All we care is
   that the neuron at the other end of the forward connection
   can get at its output. */
abstract class Axon {
  float act;
  String label;
  Link[] outputs;

  // activation value (just returns cached value, *doesn't* recalculate)
  float act() { return act; }
  // arbitrary name, mainly for debugging
  String label() { return label; }

  float delta() {
    throw new Error("Unimplemented delta function called on Axon.");
    // Yes, it is a bit of a fudge, but it keeps a lot of other
    // things neat.
  }

  abstract int layer();      // 0=input, 1=context, 2=hidden, 3=output

  /* Creating two interlinked Axons simultaneously is tricky,
     so we create the input one first, and use this method to tell
```

```
    it who it's outputting to. (This is the responsibility of the
    output Neuron.) */
  void addOutput(Link x) {
    Link[] os = new Link[outputs.length+1];
    System.arraycopy(outputs, 0, os, 0, outputs.length);
    os[outputs.length] = x;
    outputs = os;
    // Vector would be neater, but might slow propagation bits down
  }

}

/* Input node: just stores one value and always returns it as its
   activation. The activation can be changed by the set() method. */
class InputNode extends Axon {
  InputNode(float act, String label) {
    this.act = act;
    this.label = label;
    outputs = new Link[0];
  }
  int layer() { return 0; }
  void set(float act) { this.act = act; }
}

// Class for hidden and output nodes.
class Neuron extends Axon {
  Link[] inputs;
  float delta; // for backprop calculation
  Net net;     // parent network, to determine activation fn & suchlike
  int layer;

  // Initial activation is irrelevant for neuron, so I should
  // probably take it out of the constructor
  public Neuron(Link[] inputs, Net net, int layer, float act,
                String label) {
    this.inputs = inputs;
    this.outputs = new Link[0];
    this.net = net;
    this.layer = layer;
    this.act = act;
    this.label = label;
    // It's our responsibility to set 'output' on the links
    for (int i=0; i<inputs.length; i++) {
      inputs[i].output = this;
      inputs[i].input.addOutput(inputs[i]);
    }
  }

  public float delta() { return delta; }
  public int layer() { return layer; }

  /* Now that I've moved context nodes into their own class,
     this is redundant, at least if I'm sticking to SRNs. */
  public void addInputs(Link[] xs) {
    // first, set the outputs for the links
    for (int i=0; i<xs.length; i++) {
      xs[i].output = this;
      xs[i].input.addOutput(xs[i]);
    }
    Link[] newI = new Link[inputs.length + xs.length];
    System.arraycopy(inputs, 0, newI, 0, inputs.length);
    System.arraycopy(xs, 0, newI, inputs.length, xs.length);
    // weights etc. already initialized by Link constructor
    inputs = newI;
  }

  /* Calculate our activation and cache it in 'act'.
     Does NOT force recalculation of any other activations.
  */
  public void recalc() {
    float sum = 0;
    for (int i=0; i<inputs.length; i++)
      sum += inputs[i].weight * inputs[i].input.act();
    // XXX this is a bit of a mess
    if (false) {
    if (Debug.lots) {
      for (int i=0; i<inputs.length; i++)
        System.err.println(inputs[i].weight+" * "+inputs[i].input.act());
      System.err.println("------------------------");
    }
    if (Float.isNaN(sum)) {
      for (int i=0; i<inputs.length; i++)
        System.err.println(inputs[i].weight+" * "+inputs[i].input.act());
      throw new Error("Looks like we've hit NaN...");
    }
    }
    act = net.act.f(sum);    // activation function
  }

}

class ContextNode extends Axon {
  Link input; // We shouldn't have more than one input
  Net net;    // parent network, to determine activation fn & suchlike

  public ContextNode(Net net, float act, String label) {
    outputs = new Link[0];
    this.net = net;
    act = act;
    this.label = label;
  }

  public int layer() { return 1; }

  /* A context node just copies the output of the neuron it's
     connected to. */
  void recalc() { act = input.input.act(); }

  /* WARNING: If this is called more than once, an output of
     the node at the other end of the old link will still be
     pointing hither, and all hell may well break loose. So
     DON'T DO IT KIDS. */
  void setInput(Link l) {
    l.output = this;
    l.input.addOutput(l);
    input = l;
  }
}
```

B.2.2 Link

```
package neurotic;

import java.util.*;

/* A Link sits between two Axons. Storing stuff like the
   weight in one or the other of them leads to problems when
   the other one wants to get at it; this is neater.
*/
class Link {
  Axon input;       // input neuron
  Axon output;      // output neuron
  float weight;     // weighting of input
  boolean trainable; // backlinks aren't trainable
  float wacc;       // accumulator for batch update
  float dw;         // change in weight (need to save for momentum)

  public Link(Axon i, boolean t, float w, float range)
  { this(i, t, w, range, new Random()); }

  //        input  trainable? mean wt range of weights
  public Link(Axon i, boolean t, float w, float range, Random rnd) {
    input = i;
    output = null;  // this will be set during Neuron construction
    trainable = t;
    wacc = dw = 0f;
    if ( (!t) && (w!=1.0f || range!=0f) && Debug.on)
      System.err.println("Warning: you'd expect a non-trainable "+
                         "link to have a weight of 1.");
    randomizeWeight(w, range, rnd);
  }

  public void randomizeWeight(float mean, float range)
  { randomizeWeight(mean, range, new Random()); }

  public void randomizeWeight(float mean, float range, Random rnd)
  { weight = mean + rnd.nextFloat()*range - range/2; }
```

```
}
```

B.2.3 Net

```
package neurotic;

import java.io.*;
import java.util.*;

/* Simple recurrent neural net package, second attempt. */

class Debug {
    public static boolean on = false;
    public static boolean lots = false;
}

public class Net {
  InputNode[] ilyr;  // input layer
  ContextNode[] clyr; // context layer
  Neuron[] hlyr;      // hidden layer
  Neuron[] olyr;      // output layer
  public float rate; // learning rate
  public float momentum;
  float sumSqError; // sum of squared errors
  Random rnd;         // our random generator (for weights)

    /* Network architecture is normal feedforward, or SRN
       like Elman '90, depending on 'recurrent' flag */
  public Net(int inputs, int hiddens, int outputs, boolean recurrent) {

    rnd = new Random();
    int contexts = recurrent ? hiddens : 0; // # context nodes
    ilyr = new InputNode[inputs+1]; // extra one for threshold bias
    clyr = new ContextNode[contexts];
    hlyr = new Neuron[hiddens];
    olyr = new Neuron[outputs];

    rate = 0.1f;                          // learning rate
    momentum = 0.9f;
    float w_centre = 0f, w_range = 0.5f; // initial weights

    // 1. create the input layer
    for (int i=0; i<inputs; i++)
      ilyr[i] = new InputNode(0, "I"+i);
    ilyr[inputs] = new InputNode(1, "Bi"); // bias

    // 2. create the context layer (will add inputs later)
    for (int i=0; i<contexts; i++)
      clyr[i] = new ContextNode(this, 0.5f, "C"+i);

    // 3. create hidden layer & links from input/context
    for (int i=0; i<hiddens; i++) {
      Link[] inLinks = new Link[inputs+contexts+1];
      for (int j=0; j<inputs; j++)
        inLinks[j] = new Link(ilyr[j], true, w_centre, w_range, rnd);
      for (int j=0; j<contexts; j++)
        inLinks[inputs+j] = new Link(clyr[j], true, w_centre, w_range, rnd);
      inLinks[inputs+contexts] =
        new Link(ilyr[inputs], true, w_centre, w_range, rnd); // bias
      hlyr[i] = new Neuron(inLinks, this, 2, 0.5f, "H"+i);
    }

    // 4. add backlinks from hidden to context
    for (int i=0; i<contexts; i++)
      clyr[i].setInput( new Link(hlyr[i], false, 5f, 0f) );

    // 5. create output layer
    for (int i=0; i<outputs; i++) {
      Link[] inLinks = new Link[hiddens+1];
      for (int j=0; j<hiddens; j++)
        inLinks[j] = new Link(hlyr[j], true, w_centre, w_range, rnd);
      inLinks[hiddens] =
        new Link(ilyr[inputs], true, w_centre, w_range, rnd); // bias
      olyr[i] = new Neuron(inLinks, this, 3, 0.5f, "O"+i);
    }
  }

  // Randomize weights of all links
  public void randomizeWeights(float mean, float range) {
    LinkEnum e = new LinkEnum();
    while (e.hasMoreElements())
      ((Link) e.nextElement()).randomizeWeight(mean, range, rnd);
  }

  // functions for accessing input/output values
  public void setIn(int node, float value) { ilyr[node].set(value); }
  public float getIn(int node) { return ilyr[node].act(); }
  public float getOut(int node) { return olyr[node].act(); }

  /* A class to represent the activation function - really
     just a wrapper around two float->float methods.
     (XXX should this be an interface? ) */
  public abstract class Activation {
    abstract float f(float x); // the activation function
    abstract float dfdx(float x); // its first derivative
    // XXX change the name of dfdx to something that's not a lie
  }

  /* Our standard (logistic) activation function */
  private class Sigmoid extends Activation {
    float f(float x) {
      float fx;
      fx = (float) (1/(1 + java.lang.Math.exp(-x)));
      if (Float.isNaN(fx)) throw new Error("Aaagh! NaN! "+x+"\n");
      return fx;
    }
    /* NB: returns slope at given f(x) (y value). So it's not really
       df/dx. So sue me. */
    float dfdx(float x)
    { return x*(1-x); }
  }

  Activation act = new Sigmoid();

  // Forward-propagate the current input values
  public void think() {
    // order of activation will be: hidden, context, output
    // (so *initial* activations only matter for context nodes)
    for (int i=0; i<hlyr.length; i++) hlyr[i].recalc();
    for (int i=0; i<clyr.length; i++) clyr[i].recalc();
    for (int i=0; i<olyr.length; i++) olyr[i].recalc();
  }

  /* Run a backprop against given (correct) output values.
     Again, it's assumed the inputs have already been set. */
  public void train(float[] desired) {

    think();

    // compute sumSqError

    sumSqError = 0;
    for (int i=0; i<olyr.length; i++) {
      float e = (desired[i] - olyr[i].act());
      sumSqError += e*e;
    }

    // calculate deltas (deltata?)

    for (int i=0; i<olyr.length; i++) {
      olyr[i].delta = (desired[i] - olyr[i].act()) *
        act.dfdx(olyr[i].act());
    }

    for (int i=0; i<hlyr.length; i++) {
      hlyr[i].delta = 0;
      for (int j=0; j<hlyr[i].outputs.length; j++)
        if (hlyr[i].outputs[j].trainable)
          hlyr[i].delta +=
            hlyr[i].outputs[j].weight *
            hlyr[i].outputs[j].output.delta();
      hlyr[i].delta *= act.dfdx(hlyr[i].act());
    }

    calcInputWeights(olyr);
    calcInputWeights(hlyr);
```

```
}

/* Apply (possibly accumulated) weight changes; resetDeltas()
   should be called after this method. */
public void changeWeights() {
  LinkEnum e = new LinkEnum(true);
      // shouldn't make any difference which way we go actually
  while (e.hasMoreElements()) {
    Link l = (Link) e.nextElement();
    l.dw = l.wacc + l.dw * momentum;
    l.weight += rate * l.dw;
  }
}

// Reset accumulated weight changes to zero
public void resetDeltas() {
  for (int i=0; i<olyr.length; i++)
    for (int j=0; j<olyr[i].inputs.length; j++)
      olyr[i].inputs[j].wacc = 0;
  for (int i=0; i<hlyr.length; i++)
    for (int j=0; j<hlyr[i].inputs.length; j++)
      hlyr[i].inputs[j].wacc = 0;
}

/* calculate changes in input weights & add them to the
   wacc (weight-accumulator) fields. */
private void calcInputWeights(Neuron[] ns) {
  for (int i=0; i<ns.length; i++)
    for (int j=0; j<ns[i].inputs.length; j++) {
      Link ln = ns[i].inputs[j];
      ln.wacc += ln.output.delta() * ln.input.act();
    }
}

// overwrite the cached activations of the context nodes with
// the given value. Handy for starting new training patterns
public void flushContext(float x)
{ for (int i=0; i<clyr.length; i++) clyr[i].act = x; }

// return sum of squared errors computed during backprop
public float sumSqError() { return sumSqError; }

// read weights in from a file (does not check for bad input)
public void loadWeights(File f) {
  try {
    StreamTokenizer input = new StreamTokenizer
      (new BufferedReader (new InputStreamReader
        (new FileInputStream(f))));

    LinkEnum enum = new LinkEnum();
    while (enum.hasMoreElements()) {
      while (input.nextToken() != input.TT_NUMBER);
      ((Link) enum.nextElement()).weight = (float) input.nval;
    }
  }
  catch (IOException e) { throw new Error(e.getMessage()); }
}

/* Return number of trainable weights in the network. Handy
   if creating an array for getAllWeights(). */
public int numWeights() {
  return (ilyr.length+clyr.length)*hlyr.length // ->hidden
    +   (1 + hlyr.length) * olyr.length ; // ->output
}

/* Output all weights as a single array of floats
   order: hidden, output (input wts of each) */
public float[] getAllWeights() {
  float[] w = new float[numWeights()];
  int i=0;
  LinkEnum e = new LinkEnum();
  while (e.hasMoreElements())
    w[i++] = ((Link) e.nextElement()).weight;
  return w;
}

/* Set all weights from the supplied array of floats.
   Order same as getAllWeights(). */
public void setAllWeights(float w[]) {
  if (w.length != numWeights())
    throw new Error("Wrong number of weights.");
  int i=0;
  LinkEnum e = new LinkEnum();
  while (e.hasMoreElements())
    ((Link) e.nextElement()).weight = w[i++];
}

/* Enumerate over all (trainable) links in network. It can
   be relied upon always to enumerate in the same order. */
public class LinkEnum implements Enumeration {
  private Neuron[] layer;
  private int i,j;
  private Link next;
  private boolean nextIsValid;
  private boolean backwards; // i.e. output then hidden

  public LinkEnum() { this(false); }

  public LinkEnum(boolean backwards) {
    this.backwards = backwards;
    layer = backwards ? olyr : hlyr;
    i = j = 0;
    nextIsValid = true;
    cacheNext();
  }

  private void cacheNext() {
    if (!nextIsValid) throw new Error
      ("LinkEnum fell off the end. This really shouldn't happen.");
    if (j>=layer[i].inputs.length) {j=0; i++;}
    if (i>=layer.length) {
      i = 0;
      if (backwards) layer = (layer==olyr) ? hlyr : null;
      else           layer = (layer==hlyr) ? olyr : null;
    }
    if (layer==null)
      nextIsValid = false;
    else {
      next = layer[i].inputs[j];
      nextIsValid = true;
    }
    j++;
  }

  public boolean hasMoreElements() { return nextIsValid; }
  public Object nextElement() {
    if (!nextIsValid) throw new Error
      ("nextElement() called on empty link enumerator.");
    Link r = next;
    cacheNext();
    return r;
  }
}

// ============== Debugging routines below here

// Dump a line describing weights to given o/p stream
// (same order as dumpWeights, naturellement)
public void describeWeights(java.io.PrintStream out) {
  for (int i=0; i<hlyr.length; i++)
    for (int j=0; j<hlyr[i].inputs.length; j++)
      out.print(hlyr[i].inputs[j].input.label()+hlyr[i].label()+" ");
  for (int i=0; i<olyr.length; i++)
    for (int j=0; j<olyr[i].inputs.length; j++)
      out.print(olyr[i].inputs[j].input.label()+olyr[i].label()+" ");
  out.println();
}

// Dump the weightings to the given output stream
public void dumpWeights(java.io.PrintStream out) {
  dumpWeights(out, hlyr);
  dumpWeights(out, olyr);
  out.println();
}

public void dumpWeights(java.io.PrintStream out,
                        Neuron[] nodes) {
  for (int i=0; i<nodes.length; i++)
    for (int j=0; j<nodes[i].inputs.length; j++)
```

```
      out.print(nodes[i].inputs[j].weight+" ");
  }


  // Dump the activations to the given output stream
  public void dumpActs(java.io.PrintStream out) {
    dumpActs(out, olyr, "Outputs: ");
    dumpActs(out, hlyr, "Hiddens: ");
    dumpActs(out, clyr, "Context: ");
    dumpActs(out, ilyr, "Inputs : ");
  }

  public void dumpActs(java.io.PrintStream out, Axon[] ns,
                       String label) {
    out.print(label);
    for (int i=0; i<ns.length; i++) out.print(ns[i].act()+" ");
    out.println();
  }


}
```

B.2.4 Fitness

```
package neurotic;

import neurotic.*;

public interface Fitness {
  void initEnviros(int n);  // create test environments for the net
  float score(Net net, int env);
        // evaluate fitness in specified environment
}
```

B.2.5 Population

```
package neurotic;

import java.util.*;
import java.io.*;
import java.text.*;

public class Population {

  Net net;
  Fitness fitfunc;     // the fitness function
  public float[][] weights; // weights of networks
  public float[] fitness;     // fitnesses of individuals
  Random rnd;
  int envs;            // number of environments for determining fitness
  float muteRate;   // mutation rate
  int currGen;         // current generation
  public int fittest, leastFit; // most and least fit individuals

  public Population(Net net, Fitness fitfunc, int size,
                    float mean, float range, float muteRate,
                    int envs) {
    this.net = net;
    this.fitfunc = fitfunc;
    this.muteRate = muteRate;
    rnd = new Random();
    this.envs = envs;
    weights = new float[size][];
    this.fitness = new float[size];
    randomizePopulation(size, mean, range);
  }

  // load weights from input stream instead of initialising randomly
  // file format: size muteRate currGen net.numWeights weights
  public Population(Net net, Fitness fitfunc, int envs,
                    Reader r) {
    this.net = net;
    this.fitfunc = fitfunc;
    rnd = new Random();
    this.envs = envs;
    try {
      StreamTokenizer in = new StreamTokenizer(r);
      in.nextToken(); int size = (int) in.nval;
      this.fitness = new float[size];
      in.nextToken(); muteRate = (float) in.nval;
      in.nextToken(); currGen = (int) in.nval;
      in.nextToken(); int numWts = (int) in.nval;
      if (numWts != net.numWeights())
        throw new Error("Population: error reading input file: "+
                        "wrong number of weights for this network.");
      weights = new float[size][];
      for (int i=0; i<size; i++) {
        weights[i] = new float[numWts];
        for (int j=0; j<numWts; j++) {
          in.nextToken();
          weights[i][j] = (float) in.nval;
        }
      }
    }
    catch (IOException e) { throw new Error(e.getMessage()); }

  }

  // save weights to output stream
  public void save(Writer w) {
    DecimalFormat d = new DecimalFormat("0.##########");
    try {
      w.write(weights.length + " " + muteRate + " " +
              currGen + " " + net.numWeights() + "\n");
      for (int i=0; i<weights.length; i++) {
        for (int j=0; j<net.numWeights(); j++) {
          w.write(d.format(weights[i][j]) + " ");
        }
        w.write("\n");
      }
    }
    catch (IOException e) { throw new Error(e.getMessage()); }
  }

  public int popSize() { return weights.length; }
  public int getCurrGen() { return currGen; }

  public void randomizePopulation(int size, float mean, float range) {
    for (int i=0; i<size; i++) {
      net.randomizeWeights(mean, range);
      weights[i] = net.getAllWeights();
    }
    currGen = 0;
  }

  float measureFitness(int i) {
    float total = 0;
    // System.err.println(i);
    net.setAllWeights(weights[i]);
    net.flushContext(0.5f);
    for (int j=0; j<envs; j++) {
      total += fitfunc.score(net, j);
    }
    return total;
  }

  // randomly select two candidates, let them compete
  // in the predefined environments, and replace the
  // loser by a mutation of the winner
  void tournamentSelect() {
    int a = rnd.nextInt(popSize());
    int b = rnd.nextInt(popSize()-1); if (b>=a) b++;
    int winner = fitness[a] > fitness[b] ? a : b;
    int loser = (a==winner) ? b : a;
    mutate(weights[winner],weights[loser]);
    fitness[loser] = measureFitness(loser);
  }

  // find individuals with least and greatest fitness
  public void recalcStats() {
    fittest = leastFit = -1;
    float max = Float.NEGATIVE_INFINITY;
```

```
    float min = Float.POSITIVE_INFINITY;
    for (int i=0; i<popSize(); i++) {
      System.err.println("> " + i + " : " + fitness[i]);
      if (fitness[i]>max) {fittest=i; max=fitness[i];}
      if (fitness[i]<min) {leastFit=i; min=fitness[i];}
    }
    if (fittest==-1) throw new Error
                     ("All "+popSize()+" fitnesses "+
                      "are Float.NEGATIVE_INFINITY.");
    if (leastFit==-1) throw new Error
                     ("All "+popSize()+" fitnesses "+
                      "are Float.POSITIVE_INFINITY.");
  }

  public float[] fittest() {
    return weights[fittest];
  }

  // progress evolution by n generations
  public void evolve(int n) {
    for (int i=0; i<n; i++) {
      fitfunc.initEnviros(envs);
      for (int j=0; j<popSize(); j++) fitness[j] = measureFitness(j);
      for (int j=0; j<popSize(); j++) tournamentSelect();
    }
    currGen += n;
    recalcStats();
  }

  public void evolve() { evolve(1); }

  // returns a mutation of the given float array
  float[] mutate(float[] in, float range) {
    float[] out = (float[]) in.clone();
    for (int i=0; i<out.length; i++)
      out[i] += (rnd.nextFloat() - 0.5f) * range;
    return out;
  }

  // overwrites 'out' with a mutation of 'in'
  void mutate(float[] in, float[] out) {
    for (int i=0; i<out.length; i++)
      out[i] = in[i] + (rnd.nextFloat() - 0.5f) * muteRate;
  }

}
```

B.3 `testsuite` package

B.3.1 XorBP

```
// XorBP: learn the XOR function by backpropagation

package testsuite;

import neurotic.*;
import java.io.*;

public class XorBP {

  static java.util.Random rnd = new java.util.Random();

  public static void main(String[] argv) {
    Net net = new Net(2,2,1,false);
    final float[][] trainingSet =
      {{0,0, 0},
       {0,1, 1},
       {1,0, 1},
       {1,1, 0}};

    net.rate = 1.0f;
    net.momentum = 0.8f;
    float[] target = new float[1];

    int k = 0;
    do {
      for (int i=0; i<trainingSet.length; i++) {
        net.resetDeltas();
        net.setIn(0,trainingSet[i][0]);
        net.setIn(1,trainingSet[i][1]);
        target[0] = trainingSet[i][2];
        net.train(target);
        net.changeWeights();
      }
      k++;
    } while (net.sumSqError() > 0.001f);


    System.out.println(k + " passes through training set.");
    for (int i=0; i<trainingSet.length; i++) {
      net.setIn(0,trainingSet[i][0]);
      net.setIn(1,trainingSet[i][1]);
      net.think();
      System.out.println(trainingSet[i][0] + " XOR " +
                         trainingSet[i][1] + " = " +
                         net.getOut(0));
    }

  }
}
```

B.3.2 BPparrot

```
// Parrot: try to learn to repeat previous input by backpropagation

package testsuite;

import neurotic.*;
import java.io.*;

public class BPparrot {

  static java.util.Random rnd = new java.util.Random();

  public static void main(String[] argv) {
    Net net = new Net(1,2,1,true);

    /* Initial weights hand-calculated. Good performance is
       given by the set        {8,0,0,-4,0,8,0,-4,0,8,-4}
       so I start it fairly close to this and let backprop
       do the rest. */

    net.setAllWeights(new float[] {5,-1,1,-5,1,6,-1,-3,0,6,-2});
    boolean curr, prev = false;
    float[] target = new float[1];

    net.rate = 0.5f;
    net.momentum = 0.5f;

    System.out.println("Performance before training:\n");
    test(net);

    int k;
    for (k=0; k<800; k++) {
      net.resetDeltas();
      curr = rnd.nextBoolean();
      net.setIn(0, curr ? 1 : 0);
      target[0] = prev ? 1 : 0;
```

```
      net.train(target);
      prev = curr;
      net.dumpWeights(System.err);
      net.changeWeights();
    }

    System.out.println("\nAfter " + k + " backprop passes\n");
    test(net);

  }

  static void test(Net net) {
    StringBuffer ins, outs;
    boolean curr, prev = false;
    ins = new StringBuffer("Input ");
    outs = new StringBuffer("Output ");
    for (int i=0; i<60; i++) {
      curr = rnd.nextBoolean();
      net.setIn(0, curr ? 1 : 0);
      net.think();
      ins.append(curr ? "*" : ".");
      if (net.getOut(0) < 0.2) outs.append(".");
      else if (net.getOut(0) < 0.8) outs.append("?");
      else outs.append("*");
      prev = curr;
    }
    System.out.println(ins);
    System.out.println(outs);
    // net.dumpWeights(System.err);
  }

}
```

Bibliography

[AL90] D. H. Ackley and M. S. Littman. Generalization and scaling in reinforcement learning. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 550–557, San Mateo, 1990. (Denver 1989), Morgan Kaufmann.

[Che00] Richard Cheng. The roboduck project, 2000. Report on Computation project.

[CP92] Federico Cecconi and Domenico Parisi. Neural networks with motivational units. In J.-A. Meyer, H.L. Roitblat, and S.W. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 346–355, Cambridge, MA, 1992. MIT Press.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, apr 1990.

[Fog98] David B. Fogel. Evolving neural networks. In David B. Fogel, editor, *Evolutionary Computation: the fossil record*, pages 481–484. IEEE Press, Piscataway, NJ, 1998.

[Gal93] S. I. Gallant. *Neural Network Learning and Expert Systems*. The MIT Press, Cambridge, MA, 1993.

[HSZG92] K. J. Hunt, D. Sbarbaro, R. Zbikowski, and P. J. Gawthrop. Neural networks for control systems — a survey. *Automatica*, 28(6):1083–1112, 1992.

[Jor86] M. I. Jordan. Serial order: A parallel distributed processing approach. Technical Report ICS Report 8604, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA, May 1986.

[Lau00] Yun-Tung Lau. *The Art of Objects: Object-Oriented Design and Architecture*. Addison-Wesley, 2000.

[Mee96] Lisa A. Meeden. An incremental approach to developing intelligent neural network controllers for robots. *IEEE Transactions on Systems, Man, and Cybernetics. Part B: Cybernetics*, 26(3):474–485, 1996.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP research group., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. MIT Press, Cambridge, MA, 1986.

[SSCM88] D. Servan-Schreiber, A. Cleeremans, and J. McClelland. Encoding sequential structure in simple recurrent networks. Technical Report CMU-CS-88-183, Carnegie Mellon University, CS Dept., Pittsburgh PA, 1988.

[TW88] V. V. Tolat and B. Widrow. An adaptive 'broom balancer' with visual inputs. In *IEEE International Conference on Neural Networks*, volume II, pages 641–647, San Diego, CA, July 24–27 1988.

[Vau99] Richard Vaughan. *Experiments in Animal-Interactive Robotics.* DPhil thesis, Oriel College, Oxford, 1999.

[Was93] P. D. Wasserman. *Advanced Methods in Neural Computing.* Van Nostrand Reinhold, New York, 1993.

[WD92] G. M. Werner and M. G. Dyer. Evolution of herding behaviour in artificial animals. In J.-A. Meyer, H.L. Roitblat, and S.W. Wilson, editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 393–399, Cambridge, MA, 1992. MIT Press.

[Win92] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, third edition, 1992.

[Yao99] Yao. Evolving artificial neural networks. *PIEEE: Proceedings of the IEEE*, 87, 1999.