Training a low-inertia robot to take an optimum path by applying an evolutionary algorithm to the attributes of a Continuous Time Recurrent Neural Network

Introduction

I devised this problem in order to learn about Continuous Time Recurrent Neural Networks and their training using an evolutionary approach. The basic challenge I chose was for a robot to learn how to navigate towards a target while optimising its route to be the most efficient.

Although I had some success, it was limited. In summary, the exercise showed me that the number of variables to optimize in a neural net of even a small size makes for an extremely large "problem space", and an evolutionary approach may take extremely long to find a suitable solution, if it can find one at all.

Problem definition

The problem I set out to solve was as follows.

1. A two-wheeled robot of negligible inertia is used. View from above:

2. The wheels are on either side of the robot and their speed is controlled by two motors, one for each wheel. The wheel speeds are related to the inputs to the motors as follows:

   $\omega_{w_k} \propto \tanh I_k$ where ω_{w_k} is the angular velocity of wheel k, and I_k is the input to the motor for wheel k.

3. Since the velocity of the robot at the centre of each wheel (v_{w_k}) is proportional to the angular velocity of the wheel, we can define v_{w_k} using:

   $v_{w_k} = V_{max} \tanh I_k$ where v_{w_k} is the velocity (m/s) of the robot at the centre of wheel k, and V_{max} is the maximum possible speed of the robot.

4. The robot is initially positioned in a 2-dimensional landscape at (0, 0) facing "East" (parallel to the x-axis). The robot must move to a target location at some point (x, y).

5. The robot is powered by a battery. Energy is depleted from the battery in three ways:

   a) Constant power usage. Energy is depleted at a constant rate, representing constant power usage by, for example, the control circuits.

   b) Distance dependent power usage. Energy is depleted for each unit of distance travelled by the robot, representing the energy needed to move the robot.

   c) Energy sink(s) in the landscape. The energy sinks are entirely artificial devices placed into the landscape to increase the difficulty for the robot to find the optimum route to target.

Without any energy sinks, the lowest energy route is obviously a straight line to the target. Each energy sink is located at a point on the landscape and "drains" energy from the robot according to the relationship

d ≥ 0.01m: $e = {}^{P}/_{d^2}$

d < 0.01m: $e = {}^{P}/_{0.01^2} = 10000P$

where e = energy drain rate in Watts, P is a constant and represents the "strength" of the energy sink, and d is the distance in metres between energy sink and centre of the robot.

6. The robot should learn to find an optimum path for varying "landscapes", each involving a target and 0, 1 or 2 energy sinks. The optimum path is defined as one that uses the least energy (i.e., depletes the battery the least.)

7. The robot has sensors that are able to detect the distance and direction of the target, and also the distance, direction and strength of any energy sinks. The sensors are also able to differentiate between two energy sinks so that the information corresponding to one sink is not confused with the information corresponding to the other. The sensor signals available to the controller network are therefore:

Target	**Energy sink 1**	**Energy sink 2**
Direction	Direction	Direction
Distance	Distance	Distance
	Strength	Strength

Directions are all measured relative to the forward direction of the robot in radians, going from 0 to π anti-clockwise, and 0 to −π clockwise. I chose this scale instead of 0 to 2π so as to avoid a discontinuity at the "straight ahead" position. I thought this might lead to learning difficulties as the direction to target might jump frequently between ~0 and ~2π as the target moved relatively from one side of the "straight ahead" line to the other.

Distance is measured in metres, and energy sink strength is the value of the constant P for each sink.

8. The landscapes are all of similar size in that the original distance between robot and target does not vary to an extreme degree. The longest original distance is never more than 200% of the shortest original distance. However the original direction to target is completely random. Energy sink strengths remain constant during a simulation but may vary from one sink to another.

Robot dynamics

The robot is assumed to have negligible inertia so as to keep the simulation simple. As stated above the robot velocity at the centre of each wheel is given by:

$$v_{w_k} = V_{max} \tanh I_k$$

where v_{w_k} is the velocity (m/s) of the robot at the centre of wheel k, and V_{max} is the maximum possible speed of the robot.

Thus the following equations are used to describe the motion of the robot using Euler integration:

$$v_R = {}^{1}\!/_{2}\,(v_{w_1} + v_{w_2})$$

where v_R is the speed of the robot (m/s) in the direction perpendicular to the wheels' axes and v_{w_k} is the velocity (m/s) of the robot at the centre of wheel k.

$$\omega_R = {}^{1}\!/_{2r}\,(v_{w_2} - v_{w_1})$$

where ω_R is the angular velocity of the robot (rad/s) measured anti-clockwise, and r is the robot radius.

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \\ \theta_{i+1} \end{pmatrix} \approx \begin{pmatrix} x_i + v_{R_i} \cos\theta_i \, dt \\ y_i + v_{R_i} \sin\theta_i \, dt \\ \theta_i + \omega_{R_i} dt \end{pmatrix}$$

where i is the number of time steps since simulation start, θ_i is the angle in radians between the robot forward direction and the x-axis, and dt is a suitably small time interval between each time step.

Network controller definition

The robot controller is a Continuous Time Recurrent Neural Network (CTRNN) of N nodes, where the net dynamics are governed by the equation for each node i:

$$\tau_i \frac{dy_i}{dt} = -y_i + \sum_{j=1}^{N} w_{ji}\sigma(y_j + \theta_j) + I_i$$

where y_i is the value of the node i, w_{ij} is the weight of the connection from node j to node i, θ_j is the bias at node j, τ_i is the time constant of node i, and I_i is the value of the input at node i.

The values of the nodes over time are calculated using Euler integration so that:

$$y_{i_{n+1}} \approx y_{i_n} + \frac{\delta t}{\tau_i}\left(-y_{i_n} + \sum_{j=1}^{N} w_{ji}\sigma(y_j + \theta_j) + I_i\right)$$

where y_{i_n} is the value of node i after n time steps each of length δt seconds.

Although I coded the network to allow for flexible net sizes, in this exercise I only used a network of 11 nodes, as follows:

Node	Input	Use of value
1	--	Taken as the signal to the motor wheel 1
2	--	Taken as the signal to the motor wheel 2
3	Target direction relative to robot direction	--
4	Target distance from robot	--
5	Robot forward velocity	--
6	Sink 1 direction relative to robot direction	--
7	Sink 1 distance from robot	--
8	Sink 1 strength	--
9	Sink 2 direction relative to robot direction	--
10	Sink 2 distance from robot	--
11	Sink 2 strength	--

However I also wrote the code so that nodes could effectively be cut out or "isolated" from the rest of the network by fixing all weights to or from an isolated node to zero. This meant that I could effectively reduce the number of nodes when possible. For example, if training the robot simply to reach a target position in landscapes without any energy sinks, I would isolate nodes 6 to 11.

(In the end, node 5 was isolated in all cases. This node was a hangover from an earlier version of model in which an element of robot energy depletion depended on speed rather than distance travelled.)

Although the code allowed for varying time constants and biases, in all training I fixed the time constants to 1 and biases to 0.

Training approach

Genotype definition

The trainer uses an evolutionary algorithm to attempt to find an optimum set of parameters for the CTRNN. Thus the "genotype" is the set of parameters that defines the CTRNN, being:

- Weights – defining the strength of connections between nodes. For a CTRNN of n nodes there are n^2 weights.
- Biases – defining the bias value at each node.
- Time constants – defining the time constant value at each node.

Thus for a network of 11 nodes the genotype will consist of 143 genes (11 * 11 + 11 + 11). However there was always at least one node "isolated" (node 5) and in many instances more were isolated. When isolated, all the associated weights for that node are set at zero and remain immutable.

In addition, although the code allowed otherwise, I chose to make all biases immutable with a value of 0, and all time constants immutable with a value of 1. Thus the longest genotype would contain 100 genes (10 * 10 + 0 + 0), and it was often significantly shorter depending on the parameters for the particular training exercise.

Discussion
I am unsure whether fixing the biases to zero was an unwise constraint. I wonder whether the fact that the inputs were not rigorously normalised might have created problems for the learner and this might have been offset by biases. However this is speculation and not thought through – more of a note to myself to think through the effect of biases in this scenario.

Fitness function

For each "train" or "learn" a set of landscapes was defined, each with a target and 0, 1 or 2 energy sinks. The goal was to evolve a CTRNN that could, for each landscape, successfully guide the robot to the target using the optimum path (i.e., the path requiring the least energy).

When evaluating the fitness of an individual CTRNN, a simulation was run for each landscape and the combined results were used to evaluate the CTRNN's fitness. Each simulation stopped when one of the following two conditions were met:

- The robot battery level reached 0. This was recorded as a "failure".
- The robot made it to within a certain (short) distance of the target before fully depleting the battery. This was recorded as a "success".

The first component of the fitness function was therefore the % of successes. If the robot achieved success for three landscapes out of 4, say, then the success rate was 75%.

However it seemed likely that there would be many instances where two individuals could achieve the same success rate, but one has obviously out-performed the other.

For example, let us imagine that two individuals have both achieved 3 successes in a 4-landscape problem. However, in the case of one individual the robot used considerably less energy to reach the target in each landscape.

Similarly, let us imagine another scenario where both individuals have failed to achieve any successes. And this time in the evaluations of one individual the robot ran out of battery not far from the target for all four landscapes, while in the evaluations of the other individual, the robot headed off in completely the wrong direction in each landscape and finished a long way from the target.

To allow differentiation in these scenarios, a "fitness score" for each simulation was also used, calculated as:

1. If a success, the amount of energy remaining in the battery at the end of the simulation.
2. If a failure, the distance between robot and target at the end of the simulation multiplied by -1.

This meant that:

- Successes always had a positive fitness score, and those that used less energy had a higher score.
- Failures always had a negative fitness score, and those that ended up further from the target had a lower (i.e., more negative) score.

The simulation fitness scores were then averaged over the landscapes for each individual to give the individual's overall fitness score.

When comparing two individuals performance, the "winner" was chosen as follows:

1. If one individual has more successes than the other, then that individual is the winner.
2. If both have the same number of successes, then the winner is the individual with the higher fitness score.

Evolutionary learning process

I used the "microbial" approach described in (Harvey, 1996) to replicate a generational approach. For each "train", the steps were as follows:

- **Define landscape(s)**. Landscapes were defined, each with a target and 0, 1 or 2 energy sinks.
- **Generate initial population**. Each individual's attributes were generated randomly within constraints to ensure that the individuals have a reasonable chance of success. The parameters for the initial population were as follows:

Population size	20
Mutable weights	Random choice in the range -5 to 5
Weights to / from isolated nodes	0
Biases	0
Time constants	1

- **Select individuals and run tournaments**. I typically ran a minimum of 20 generations and would run up to 100 generations to check whether longer trains would give a better result. Thus the number of tournaments varied between 400 and 1,000. For each tournament:
  - One individual was selected at random.
  - A second individual was selected at random within the "deme" of the first individual. Given the population size of 20 I used a deme of 5.
  - A simulation was run for each landscape for each individual, the results were compared and the "winner" was identified using the fitness function as described above.
  - The loser's genes were replaced with the winner's genes on a random basis: each of the loser's genes was replaced with 50% probability.
  - All of the loser's genes were mutated by ±X, where X is the larger of 0.01 and 1% of the gene's current value. In other words the mutation rate was ±1%, unless the gene's current value was < 1, in which case the mutation rate was ±0.01. (This approach was used to ensure that where a gene value was initially set to zero or a very small value, mutation would not be zero or negligible.)

Balance between randomness and evolution (through mutation and recombination)

The evolutionary approach described above places certain constraints on the extent to which the "problem space" is explored. As is the case with any evolutionary algorithm, the initial randomly-generated population represents a haphazard "guess" at solutions – each a point in the "problem space" defined by the values of the genes for that individual / solution. The evolutionary algorithm then attempts to find "better" solutions by "moving" from those initial start-points to other points in the problem space using two methods:

1. Mutation. This represents a small change, in other words a movement of a small "distance" in the problem space.

2. Recombination. This can lead to a big change, in other words a movement of considerable "distance" in problem space.

However, it must be remembered that recombination does not introduce any new genes into the gene pool. It merely mixes up those that are already there. In other words, mutation is the only true "exploration" of the problem space. We could instead think of recombination as a way of increasing the number of initial "guesses" at solutions – an initial population of twenty individuals represents not just twenty guesses, but a much higher number being the various ways those individuals are recombined during the process. We cannot know how many more individuals there are as a result of recombination, because recombination is a probabilistic process, but we can know that a) there are more, and b) there are not more than a certain finite number. Although it is not the case here, if the recombination were fixed as a "crossover" halfway through the genotype, meaning that under crossover the first half of individual A replaced the first half of individual B (and vice versa), then the maximum number of combinations possible in a population of n is calculated as n^2. (There are two half-genotypes, and there are n possible values for each half-genotype.)

Through initial experimentation, I observed that increasing the number of generations beyond a certain point never seemed to bring any benefit. In other words, if there was an adequate solution to be found in the regions of the problem space defined by the initial population, that solution was found reasonably quickly. Continuing to "crawl" very small distances through the problem space from the start-points (not just the initial individuals but combinations thereof) was unlikely to find a far better solution – rather only very small improvements would be found.

As a result, I decided to introduce more variation into the gene pool. This was done by running many "trains" of, say, 20 generations each, with each population being randomly generated at the beginning of that train. However the best performing individual of the last train would also be included in the initial population of the next train.

Discussion

Recombination in this exercise was set to happen through each gene being copied from tournament winner to tournament loser using a 50% probability. In effect this introduces much more variability that the "fixed" crossover point model described above. Instead of there being only n^2 combinations possible, there is a maximum of n^n possible combinations. With a genotype containing just 10 genes, this is already a massive number. So with hindsight, it now seems superfluous to introduce extra variability by running multiple trials with new populations.

Running the algorithm instead on a single initial population for many more generations would I think be equally likely to produce a good result. However one possible modification would be to change the way the initial population was generated. Instead of choosing the initial value of a particular gene randomly for each individual (the random choice being between the limits defined for that gene), the values could be set to be equidistant from each other so to guarantee an even coverage of that dimension / gene in the problem space. For example, with an initial population of 20 individuals and a particular gene having a range of -5 to 5, the values would start at -5 and increase with an interval of 10/19.

Results

Summary

I took two different approaches to training. The first was to set the problem and see if a solution could be learned. As this did not seem to be working – the problem appeared to be too "hard" for the evolutionary algorithm to find a solution in a reasonable timeframe – I then tried breaking the problem down into the following incremental sub-problems, the idea being that I could include the successful algorithm from the previous sub-problem in the initial population for the next sub-problem. My plan was to build up the difficulty in increments, by training the robot to find an optimum path to a target:

1. Without any energy sinks present.
2. With 1 energy sink present always of the same strength.
3. With 2 energy sinks present always of the same strength.
4. With 2 energy sinks present of different strengths.

In the end the first task was achieved, but tasks 2, 3 and 4 were not. In summary, I cannot be sure why the training failed for the more complex tasks. Possible reasons are:

- The CTRNN required more nodes for a problem of this complexity. The networks I used had only enough nodes in order for each input and output signal to have its own node.
- Inadequate or ineffective pre-processing of input data to suit the learning process. For example, scaling inputs so that they are all in a similar range (e.g., 0 to 1) might help prevent one factor dominating the controller. Given that all weights were constrained to being within a single range (-5 to 5), if one input can have a significantly larger magnitude than others then the learner may struggle to "dial out" its dominance by adjusting weight values.

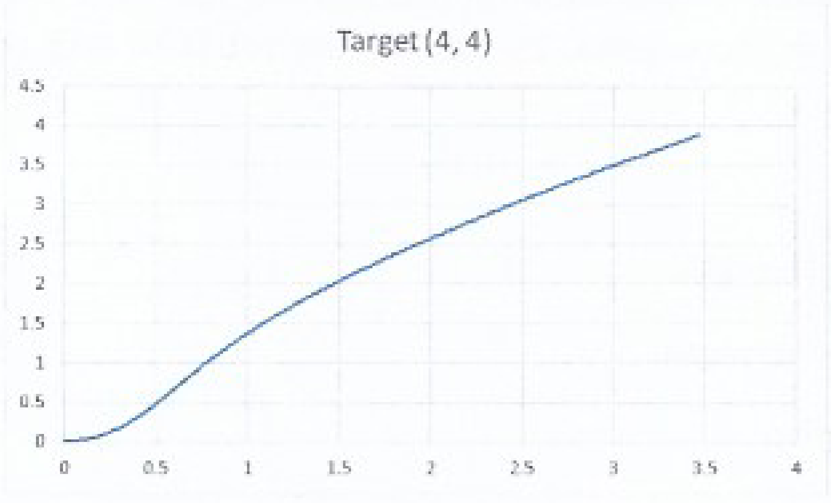
- The problem is too "complex" for an evolutionary algorithm to solve in a reasonable timeframe. Without reducing the dimensionality or giving the robot more "assistance" the solutions may be too scarce within the problem space, thus making the probability of the evolutionary algorithm finding them very low.
- The constraints placed on the CTRNN parameters (values of weights, biases and time constants) were poorly chosen and there are either no or few solutions within the problem space defined by those ranges.

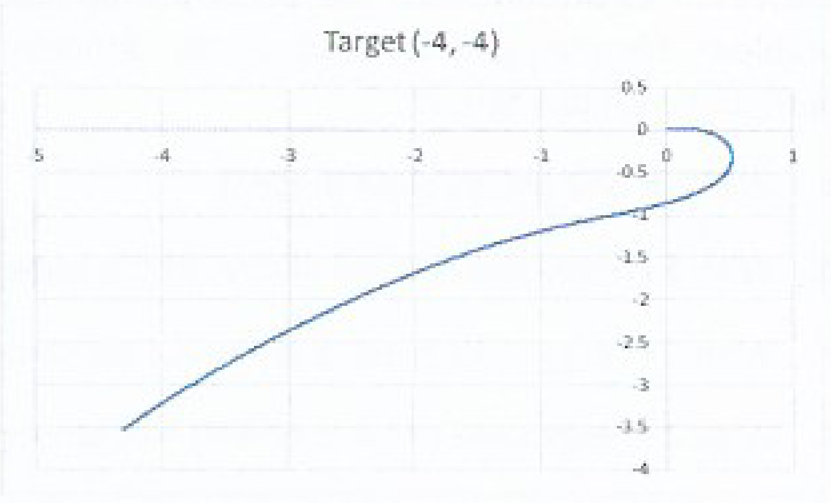
Training results – Target only, no energy sinks

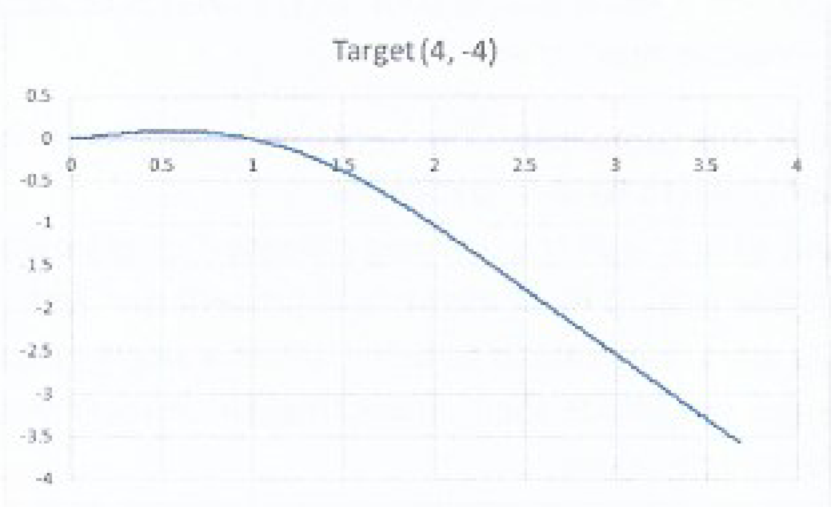
The parameters for the training exercise were as follows:

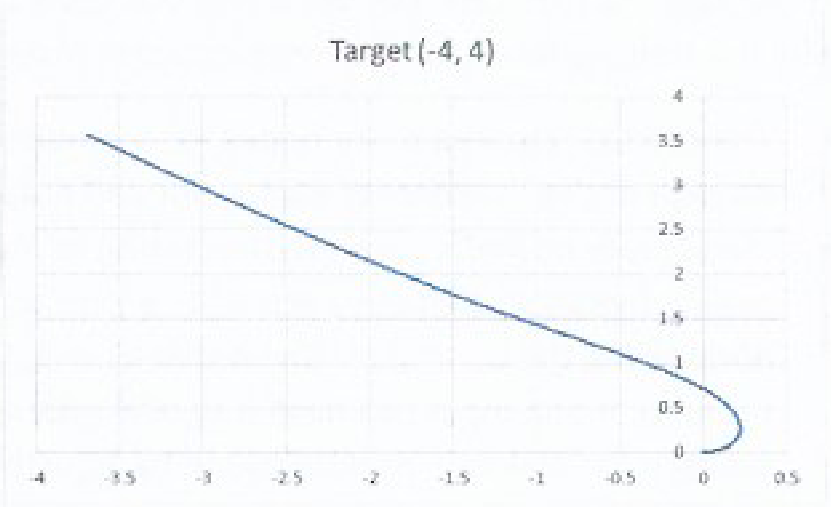
Landscapes	Targets at (4, 4), (-4, -4), (-4, 4), (4, -4), (6, -2), (-6, 2), (10, 0), (-10, 0) The intention was to represent a variety of directions and distances.
Node constraints	Isolate all nodes except those corresponding to motor inputs and target direction. All other nodes do not provide useful information: - Target distance. For problems without energy sinks the target distance is not important. With energy sinks it becomes useful information as relative distance between target and energy sinks is needed to assess optimum path. - Robot forward speed. Irrelevant for all exercises. - Energy sinks speed, direction and strength. Irrelevant since there are no energy sinks. In addition, all weights representing node self-connections were set to zero (i.e., $w_{ij} = 0$ where i = j).

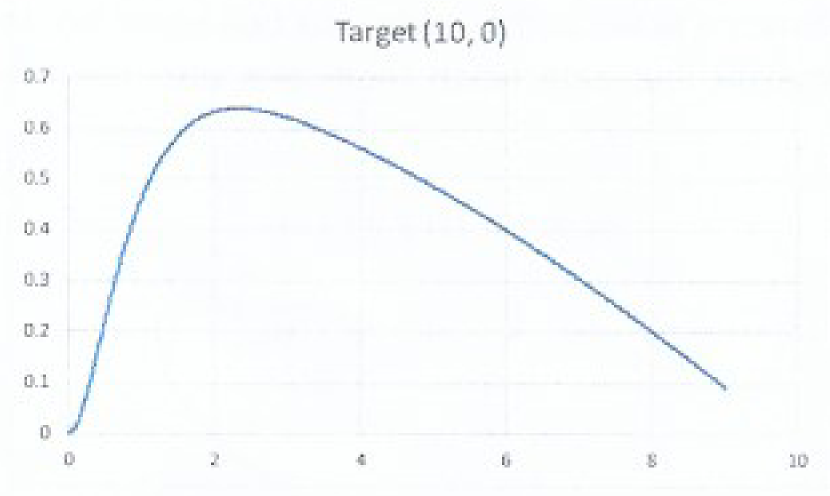
The best results after several trains (in which the best CTRNN from the previous train was included in the initial population for the next train) are illustrated below. In all cases, the robot takes a reasonably direct path to the target. (Note that the y scale varies on the charts so the deviation from a direct path appears exaggerated in some instances.)

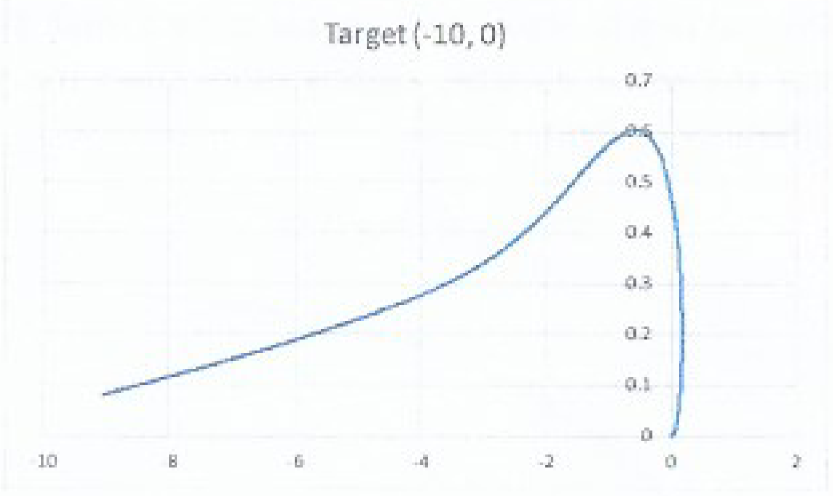


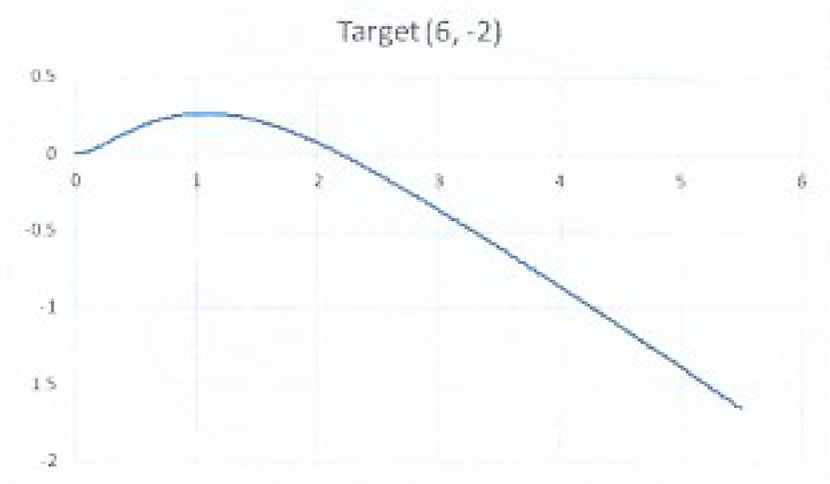


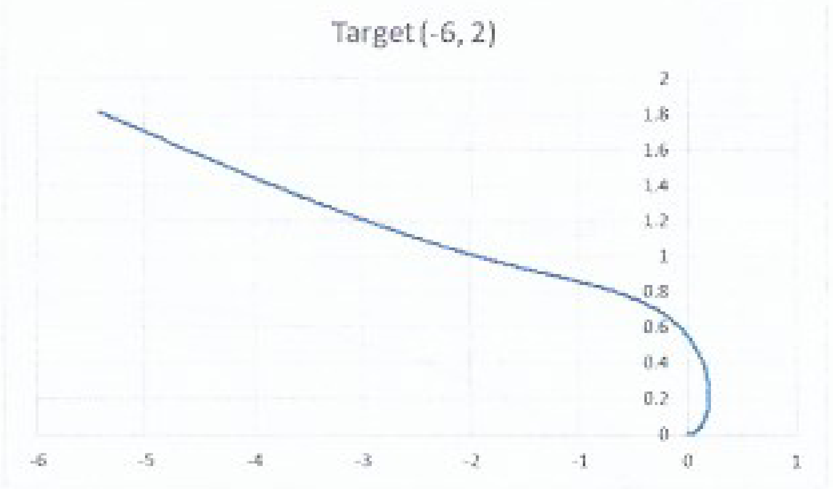










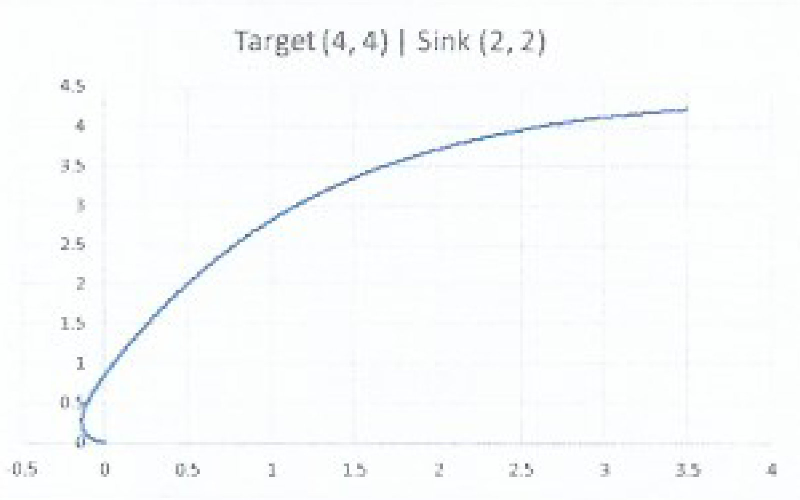


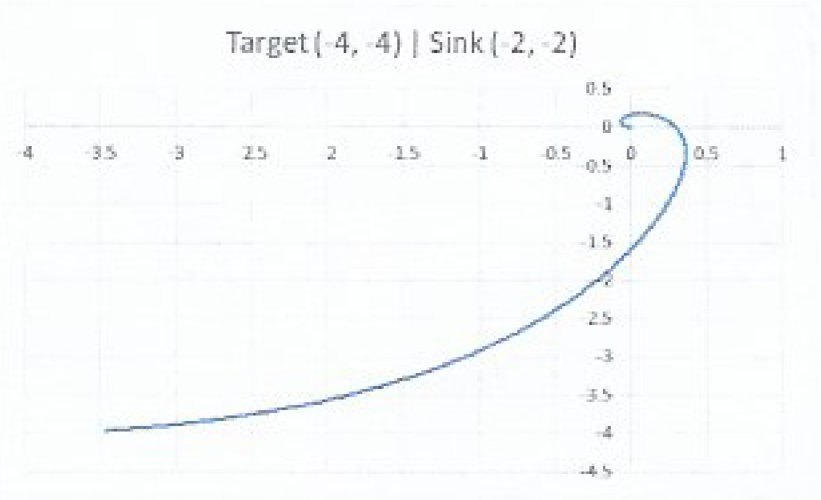
Training results – Target and 1 energy sink always of the same strength

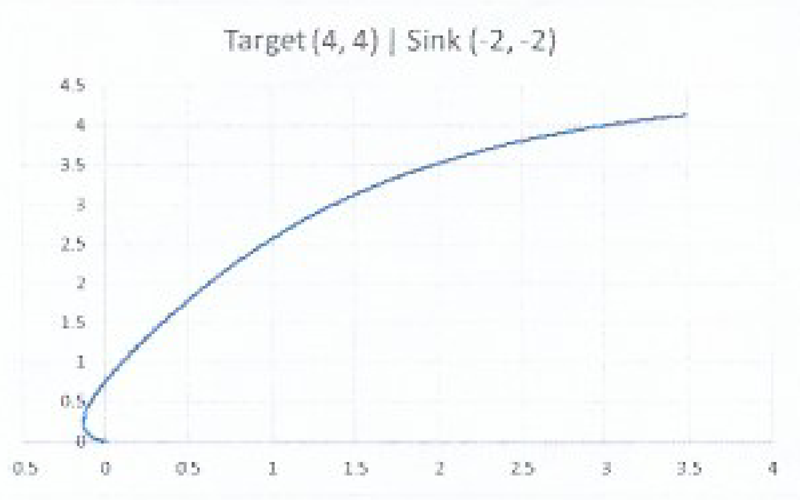
The parameters for the training exercise were as follows:

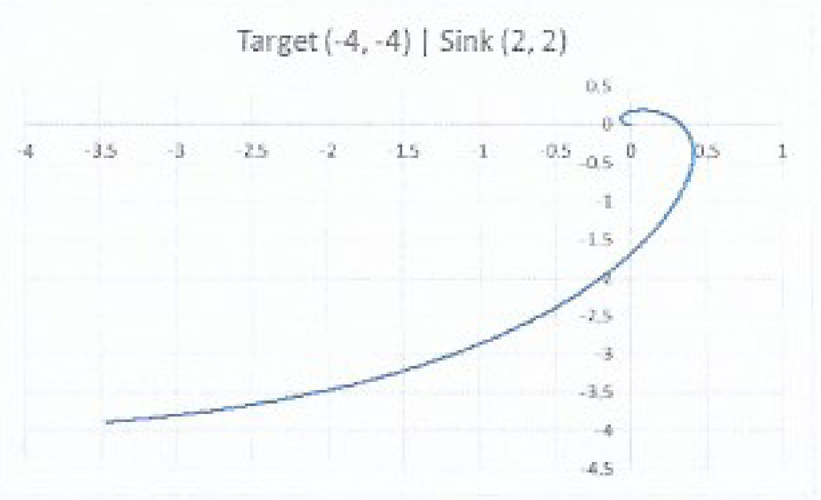
Landscapes	• Target (4, 4), Energy sink (2, 2) • Target (-4, -4), Energy sink (-2, -2) • Target (4, 4), Energy sink (-2, 2) • Target (-4, -4), Energy sink (2, 2) All energy sinks had the same strength of 4. The strength was chosen so that the sinks had a significant effect on battery depletion given the landscape scale, but not so significant that the task was always impossible to achieve. The intention here was to create two scenarios. The first scenario is given by the first two landscapes, where an energy sink is directly between the robot and target, so the optimum path should be to detour around the energy sink. The second scenario is given by the second two landscapes where the robot is directly between the energy sink and the target, so the optimum path should be a straight line to the target.
Node constraints	Isolate the following nodes: - Robot forward speed. Irrelevant for all exercises. - Speed, direction and strength for energy sink 2. Irrelevant since there is no second energy sink. In addition, all weights representing node self-connections were set to zero (i.e., $w_{ij} = 0$ where i = j).

The results after a long period of training are shown below. It seems as though the robot has learnt a curved path towards the target, which is a "good" path to use when the energy sink is between it and the target, but it is far from an optimum path in the second scenario when the robot is between sink and target. There does appear to be a small difference in the paths used in the two scenarios, in that the robot deviates slightly more from the straight line path when there is a sink, but the difference is slight.









Bibliography

Harvey, I. (1996). *The Microbial Genetic Algorithm*. Unpublished.