

MightyMouse: Optimization of Two-task Autonomous Mobile Robot Behavior
Using PSO-Trained Neural Networks

Author: Ryan J. Meuth

Abstract: Much research has been done on training neural networks to control small mobile robots for box pushing behaviors, but these have traditionally focused on using evolutionary algorithms and reinforcement learning schemes. There has also been some research into PSO-based robot search algorithms, but very little exploration has been done on complex, self-sustaining robot behaviors. Here, a method for optimizing Mobile robot behavior in a two-task environment is presented, with an exploration of optimal neural network architectures in the environment. . It was originally intended that the method be executed on actual real world hardware, but due to hardware problems and time constraints a detailed simulation environment was constructed instead. The method was executed on the environment with various network architectures, and it was determined that no single architecture is superior in this environment.

Introduction

In the next 10 years, the industry of consumer robotics appliances is expected to gain an immense amount of popularity. Today's consumer robotics products such as the i-Robot Roomba robotic vacuum, while promising, have several major shortcomings that limit their popularity. These shortcomings lie in the fundamental dependence of these products on human intervention for their operation and maintenance. For example, the Roomba is incapable of charging itself, so when it's battery dies, it is stranded until it's owner places it back on the charging station. Also, while vacuuming, the robot follows a random search pattern, that is a much less than optimal method for covering the floor of the room, often resulting in missed areas and ultimately low task efficiency. One would expect that for \$200, a person should be able to buy something that is not only capable of performing the task it was designed to do, but to also do the task without intervention. One could argue that the Roomba is not a truly autonomous device, and therefore not even truly a robot. It is this topic of autonomy and task efficiency that this project is concerned with.

Thus, we have defined some desirable traits that a mobile robotic appliance should include: self-sufficiency, and task efficiency. It should perform it's designed task to the best of it's ability, and it should be able to perform these tasks without human intervention over the entire course of it's operational cycle. Including these apparently obvious features would facilitate truly autonomous operation, whereby the robot works effectively 'right out of the box.'

However, building these traits into a product is not as easy as it may seem. These are complex behaviors that require a program to handle a diverse set of situations, possibly even to adapt to a changing environment. As such, hard-coded implementations of these behaviors have performed poorly, as it is very difficult to code a set of rules that will be able to handle every situation that the robot encounters. Effective hard-coded robotic control programs tend to take a very long time to implement, as a great deal of testing is necessary. Even so, these effective Hard-coded behaviors often suffer from unreliability and problems introduced by human error.

Computational Intelligence Methods and Implementations

Instead of hard-coding behaviors into mobile robot computational intelligence methods may be much better suited to controlling mobile robot behaviors, especially in dynamic environments, as the robustness and inherent versatility of CI methods allow the robot to make it's own rules about how to deal with the environment most effectively. The problem then becomes how to select and construct and appropriate CI method for the task at hand.

Sprinkhuizen-Kuiper's 2000 paper, Artificial Evolution of Box-Pushing Behavior [8], details an implementation of Evolutionary algorithms for recurrent neural networks applied to evolving box-pushing behaviors in a very simple environment. This work was performed on the Khepera mobile robot which executed a small neural network on-board. This necessity for on-board processing restricted the experimental runs, as the robot must be run through the box-pushing environment so that a candidate network may be evaluated, leading to long evaluation times. Also, the networks were kept at a minimal size, with small hidden layers and very few weights. The results of this experiment showed that a simple, global fitness function gives the most promising results, and a recurrent network architecture with edge-detecting inputs yielded the best controllers.

Previous work in computational intelligence-based robot behavior research has primarily focused on single-task environments and has mostly used Evolutionary Algorithms to produce behavior. This research is useful for evolving task-efficiency in autonomous mobile robots, but falls short in two-task environments with competing goals, as is the case when a robot must balance its own maintenance and the maintenance of the environment through its assigned task.

The Particle Swarm Optimization method may be better suited to evolving mobile robot behavior, as it has been shown in most to converge to global maxima with less iteration than Evolutionary Algorithms. For mobile robots, where evaluation time is expensive, Particle Swarm Optimization may be a better fit, as the number of evaluations to converge on an optimal behavior is fewer, thus the robot may learn faster.

Problem Description

The main research question this project is concerned with is “What is the optimal network architecture for developing mobile robot behaviors using PSO-trained neural networks in a two-task environment?” This question intended to be answered by the implementation of a PSO-trained neural network controller for an existing small mobile robot. However, hardware problems and time limitations prevented full testing on real-world hardware, so a simulation environment was constructed in its place.

Towards this end, PSO techniques and methods for behavioral optimization and evaluation were developed, and an exploration of neural network architectures for a two task environment were conducted.

CI Approach Description

Artificial Neural Networks are a biologically inspired computational intelligence method that is loosely based on biological principles of neuron operation. As we can see in figure 1, artificial neurons operate on rough principles of In-Integrate-Out, where the dot product of an input vector (which can originate from the outside of the network or from the output of other neurons) and a weight vector is passed through a transfer function to produce an activation level that is passed onto downstream neurons or out of the network.

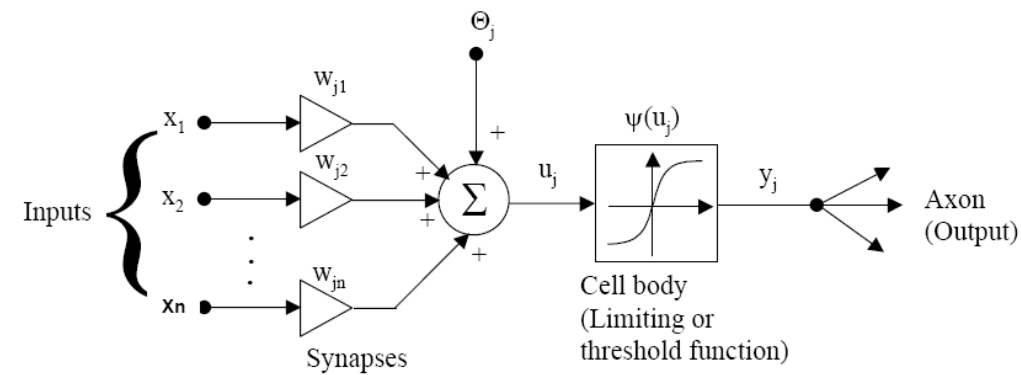


Figure 1. – Artificial Neuron Operational Diagram.

Artificial Neural Networks are not programmed, but trained by modifying their internal weight sets to perform a functional mapping from input to output. In this way, artificial neural networks can be constructed that are able to handle floating-point data types, enabling the network to operate in a continuous environment, such as that of the real world. Artificial Neural Networks can be constructed in a variety of architectures, such as straight feed forward, shown in figure 2, and recurrent neural networks, shown in figure 3. In feed forward networks, the output from each neuron is passed only to down-stream neurons, and out of the network. Here, the weight sets create a direct mapping between the input vector space and the output vector space. In recurrent networks, however, the output of the network is brought back to the input of the network as an additional set of inputs, with an integer temporal delay. This effectively builds in the capability of a sort of memory into the network, so that the next state of the network output can depend on previous states.

This flexible capability makes artificial neural networks a very powerful computational intelligence tool.

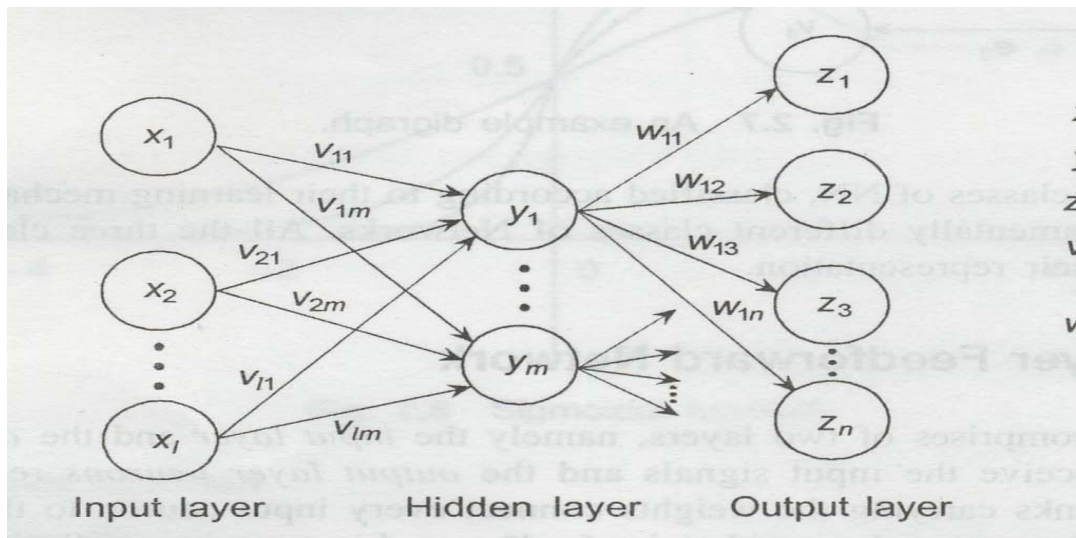


Figure 2. – Feed-Forward Neural Network Architecture.

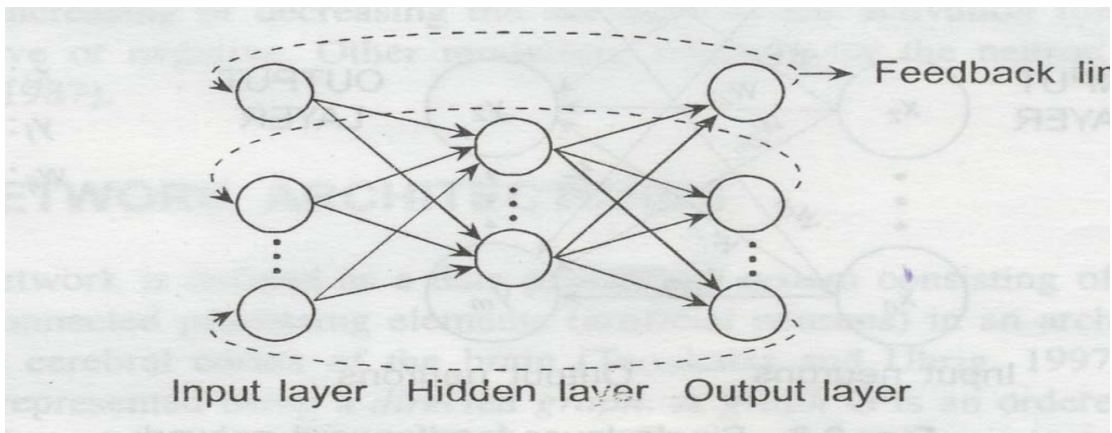


Figure 3 – Recurrent Neural Network Architecture

The typical method for constructing sets of neural network weights is Error back propagation. This method uses the error between the action network output and a desired target output to modify weights according to their contribution to the error. Error Back propagation works well when there is a well-defined set of input and correct output pairs, which is not always the case. In real-world dynamic

environments, it is not know what examples of optimal or close to optimal behavior are, making error back propagation inappropriate.

Another Computational Intelligence technique, Particle Swarm Optimization, is a population-based technique, where candidate solutions called particles are evaluated then moved through the solution-space toward locations of higher fitness. Each particle keeps track of it's position (X), it's velocity (V) and it's local best solution encountered since the beginning of the algorithm. For the population of particles, a global best particle position is maintained. Each particle is evaluated and assigned a fitness based on it's performance, and then it's position is updated through the equations in Figure 5. As we can see, the influence of the particles previous velocity is controlled by the scalar w , and the influence of local and global best positions is controlled by the scalars $c1$ and $c2$, respectively. Also of note is the random influence on position. This promotes exploration of the solution space, and allows the population to climb out of local maxima.

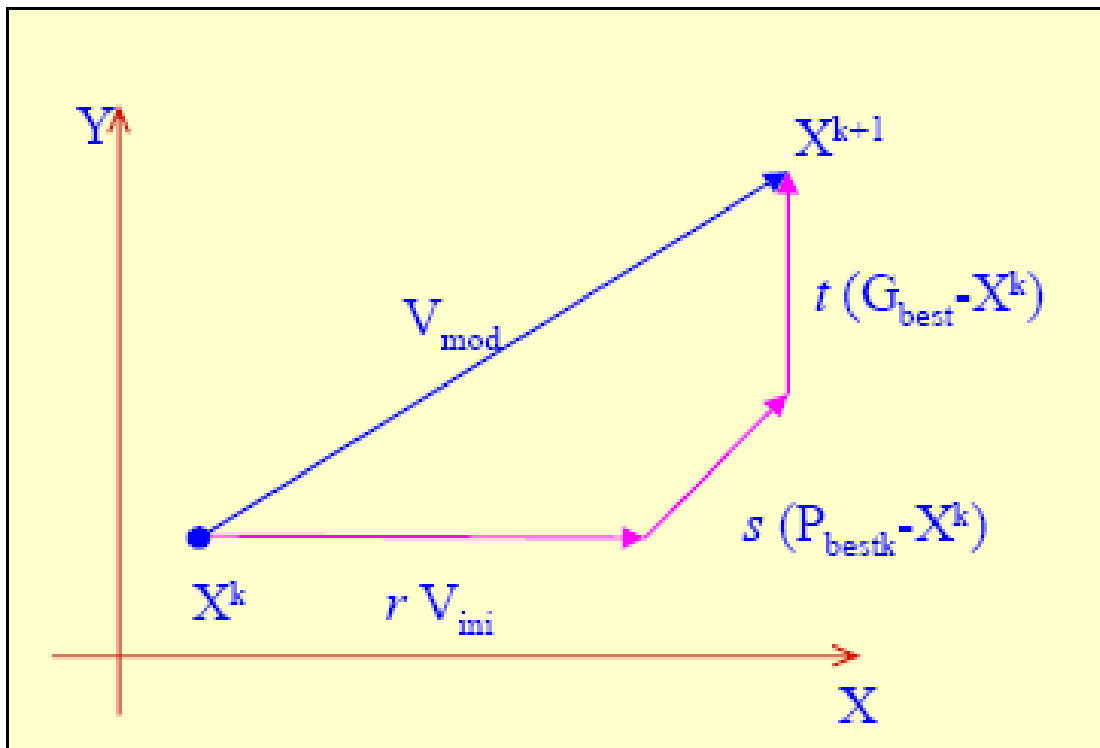


Figure 4 – Particle Motion in Particle Swarm Optimization

$$V_{id} = w \times V_{id} + c_1 \times rand_1 \times (P_{bestid} - X_{id}) + c_2 \times rand_2 \times (G_{bestid} - X_{id})$$

$$X_{id} = X_{id} + V_{id}$$

Figure 5 – Particle Update Equations.

The hybrid of the above two computational intelligence techniques is the PSO-NN architecture, a particle swarm optimization method that uses the weight sets of neural networks as the particle positions. In this way, neural networks with high fitness can be developed using the particle swarm technique. Since the fitness for the particle can be based on the behavior of the network in a real-world environment, this marriage of techniques produces a sort of reinforcement learning method that can optimize behavior in complex environments based off of a simple reward from the environment. Because of these features, the PSO-NN architecture was chosen as the method of exploration for this problem.

Experimental Implementation

The robot that was initially intended for use in this project was the MightyMouse mobile robot, shown in Figure 6. This small mobile robot, based off the chassis of a generic optical mouse, was equipped with two front bump sensors, an IR photodiode for detecting the IR beacon of its charging station, as well as an interface to the mouse's original optical displacement sensor. Also included was internal battery voltage monitoring hardware, allowing the robot to know the state of its own battery. This platform was not able to be used due to hardware problems and time constraints.

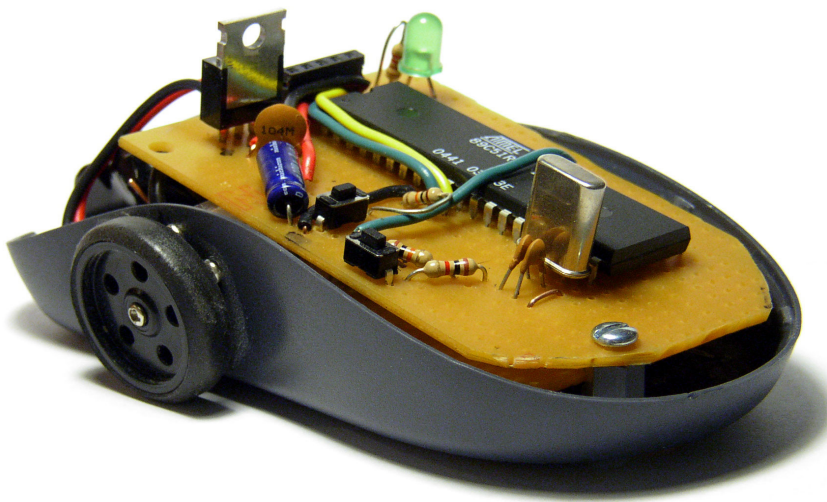


Figure 6 – MightyMouse Mobile Robot. (Actual Size)

In place of the MightyMouse robot, a 2D simulation environment was constructed and the kinematics of the MightyMouse robot was constructed within this environment, called MightySim from here on. The MightySim environment is a detailed continuous 2D simulation environment with collision detection and high-quality kinematics. Though detailed, the environment is fundamentally deterministic, a quality that the real-world does not share, making it unlikely that behaviors found in the simulation environment will translate well to actual hardware.

The experimental world that was created in the MightySim environment is shown in figure 7. The magenta and cyan boxes denote obstacle regions in the environment, and the yellow box denotes a charging region in the environment. The dark-blue path is an example of a path that the simulated robot can take through the world.

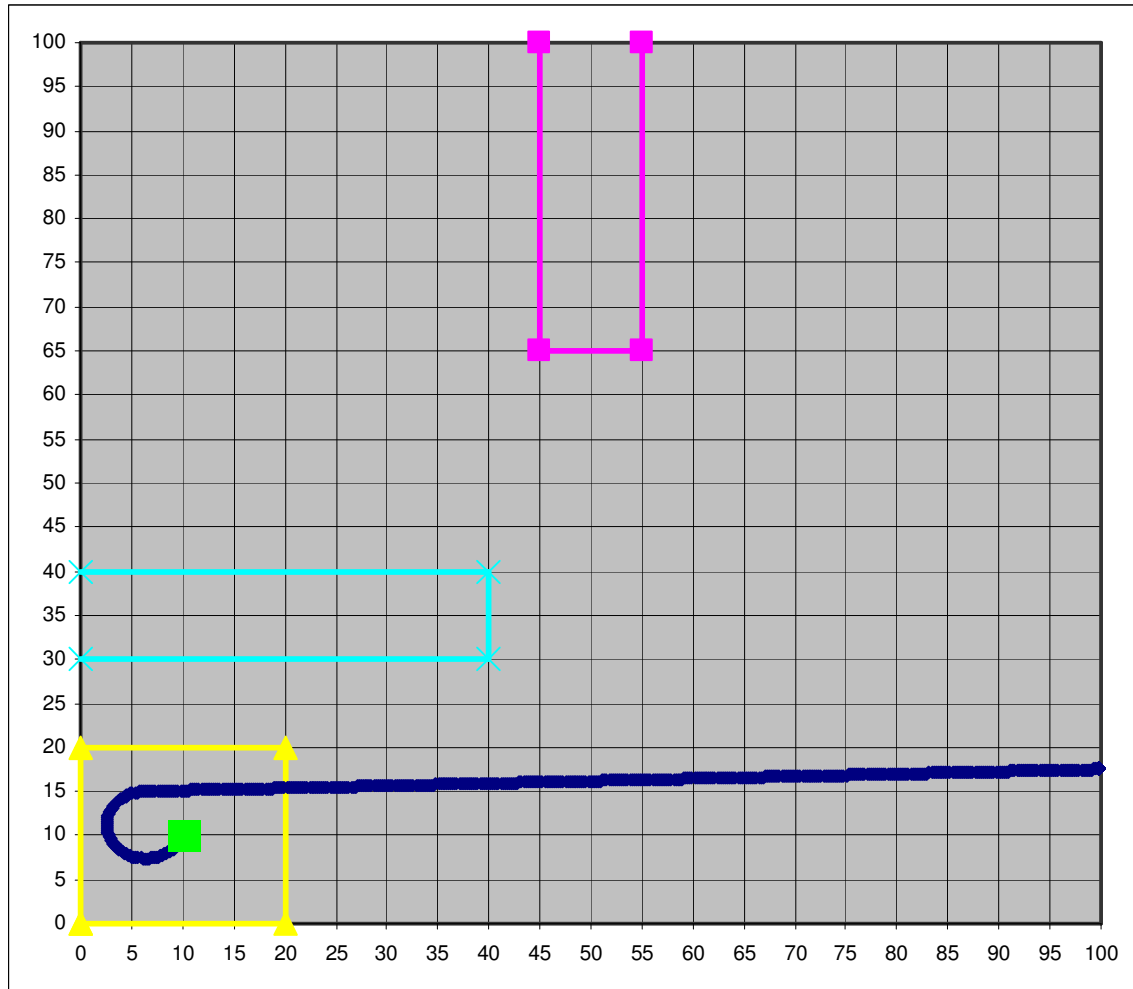


Figure 7 – The MightySim experimental simulation environment layout.

The simulated robot was emulated with the same sensor suite as the actual hardware, including bumper sensors, IR detector and displacement sensing. Bumper sensing was implemented as a pair of test points that sit in front of the simulated robot, creating simulated whiskers. If either of these points collides with a boundary, a sensor flag is set in the robot's internal state variable. The IR detector works in a similar fashion – if the center of the robot is within the charging region, a sensor flag is set in the internal state variable. Displacement simulation was accomplished by calculating the distance between the robot's previous location and the robot's current location.

For this problem, the particle network architectures were 6x14x2. The six sensor status variables were presented to the input of the network, there were 14 hidden neurons, and 2 outputs, corresponding to right and left motor velocities.

The Fitness function for each particle was implemented as the sum between an area coverage score, and a time-to-charge score. In the area coverage phase, the first task is to cover the most area in the environment as possible. This was accomplished by dividing the simulation environment into 400 5x5 square regions. Each particle is given 5000 time steps from the start of the evaluation to cover all 400 regions. The number of regions crossed is used as the fitness. If the particle is able to cover all 400 regions before 5000 time steps elapse, then the remaining time is added to the fitness as a bonus, promoting fast particle coverage behaviors. During the area coverage phase, the sensor input corresponding to battery level is held high, indicating good battery status to the network. The second phase of the evaluation, the return to charging task is implemented by holding the battery level input low, and giving the particle 5000 time steps to return to the charging region. If the particle is able to return to the charging region before 5000 time steps elapse, the remaining time is added to the fitness score. Across both evaluation stages, the maximum possible fitness score becomes 10400.

The PSO-NN was executed with parameters $w=0.8$, $c_1=c_2=2.0$ for 30 runs, each run consisting of 1000 generations with feed forward, recurrent with 1 time-delay line, and recurrent with 2 time delay line network architectures.

Results

Table 1 shows the best fitnesses for each network architecture. Note that the maximum fitnesses differ by only a few points, indicating that architecture was not greatly dependant on success in the environment. Also note that the fitness did not reach the maximum fitness value on any trial, indicating that the task was not accomplished 100%, and that further learning is possible.

Architecture	Best Fitness
Feed-Forward	9671
Recurrent, Td= 1	9677
Recurrent, Td= 2	9676

Table 1. – Best Fitness per architecture.

Figure 8 shows the Average Fitness across 30 runs for each of the evaluated network architectures. We can see that no architecture is obviously more fit than any other. This can be seen with certainty if we examine the results of F and T-tests between the data sets, shown in tables 2 through 5.

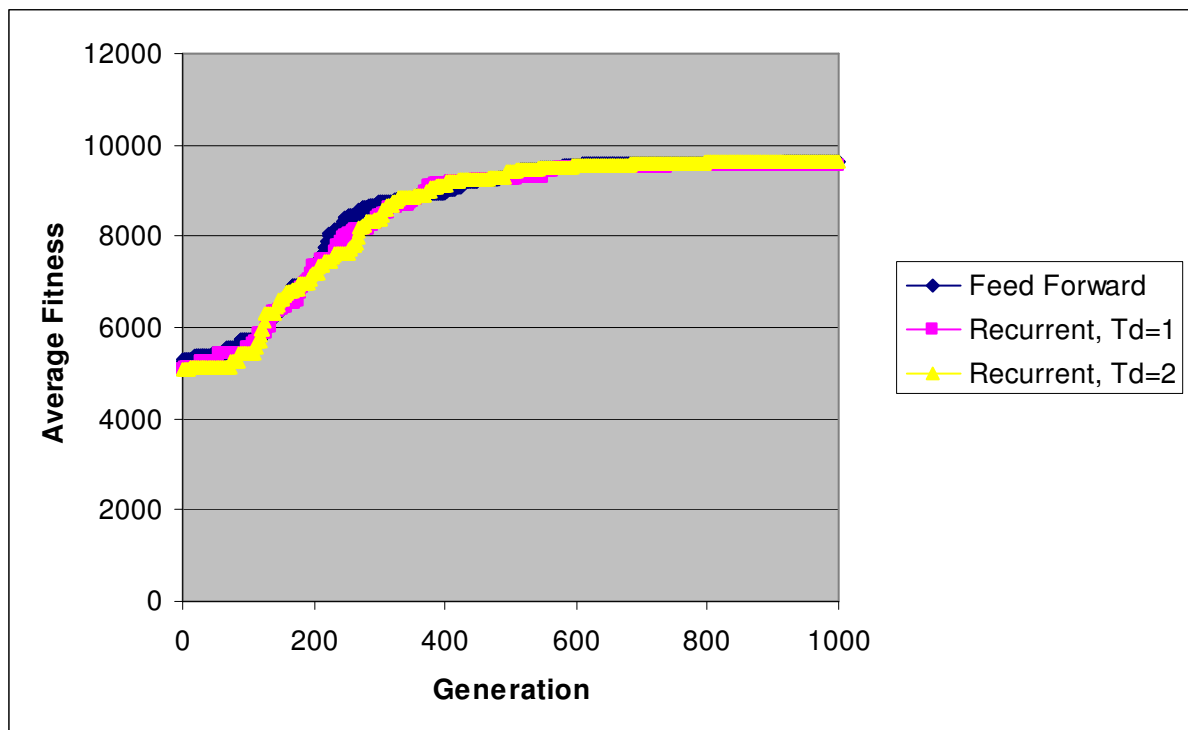


Figure 8 – Average Fitness Across All runs for each network architecture.

F-Test Two-Sample for Variances		
	<i>No Delay</i>	<i>Delay = 1</i>
Mean	8602.905	8507.866
Variance	2030651	2143944
Observations	1001	1001
df	1000	1000
F	0.947157	
P(F<=f) one-tail	0.195407	
F Critical one-tail	0.901154	
F > F Critical - Unequal Variances		
t-Test: Two-Sample Assuming Unequal Variances		
	<i>No Delay</i>	<i>Delay = 1</i>
Mean	8602.905	8507.866
Variance	2030651	2143944
Observations	1001	1001
Hypothesized Mean Difference	0	
df	1999	
t Stat	1.471676	
P(T<=t) one-tail	0.070633	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0.141266	
t Critical two-tail	1.961153	
tStat < tCritical - No Difference		

Table 2 – F and T Test Results for No Delay Average Fitness vs. Delay =1 Average Fitness.

F-Test Two-Sample for Variances		
	<i>No Delay</i>	<i>Delay = 2</i>
Mean	8602.905	8538.431
Variance	2030651	2268166
Observations	1001	1001
df	1000	1000
F	0.895284	
P(F<=f) one-tail	0.040224	
F Critical one-tail	0.901154	
F < F Critical - Equal Variance		
t-Test: Two-Sample Assuming Equal Variances		
	<i>No Delay</i>	<i>Delay = 2</i>
Mean	8602.905	8538.431
Variance	2030651	2268166
Observations	1001	1001
Pooled Variance	2149409	
Hypothesized Mean Difference	0	
df	2000	
t Stat	0.983846	
P(T<=t) one-tail	0.162655	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0.32531	
t Critical two-tail	1.961153	
tStat < tCritical - No Difference		

Table 3 – F and T Test Results for No Delay Average Fitness vs. Delay =2 Average Fitness

F-Test Two-Sample for Variances		
	<i>Delay = 1</i>	<i>Delay = 2</i>
Mean	8507.866	8538.431
Variance	2143944	2268166
Observations	1001	1001
df	1000	1000
F	0.945233	
P(F<=f) one-tail	0.186657	
F Critical one-tail	0.901154	
F > F Critical - Unequal Variance		
t-Test: Two-Sample Assuming Unequal Variances		
	<i>Delay = 1</i>	<i>Delay = 2</i>
Mean	8507.866	8538.431
Variance	2143944	2268166
Observations	1001	1001
Hypothesized Mean Difference	0	
df	1998	
t Stat	-0.46038	
P(T<=t) one-tail	0.322645	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0.64529	
t Critical two-tail	1.961153	
t Stat < tCritical - No Difference		

Table 4 - F and T Test Results for Delay = 1 Average Fitness vs. Delay =2 Average Fitness

Discussion of Results

The above results show that for this experimental setup, there is no architecture that is significantly better at producing two-task mobile robot behavior than any other. This was verified using the two-tailed t-tests with appropriate variances as found by the F-test.

Conclusion

The PSO-NN method and behavioral optimization methods for developing effective two-task mobile robot behaviors was presented, and the characteristics of a simulation environment for evaluating the architecture was also presented. The method was executed on the environment with various network architectures, and it was determined that no single architecture is superior in this environment. In order to further verify this, the system might be run additional times to collect a larger amount of data, or an alternate fitness function might be used to change the dynamics of the system.

Future Work

Future work may include additional runs to collect more data, or the fitness function of the PSO-NN might be modified for different dynamics. It would be interesting to see how the method performs on actual real world hardware. Also, parameter optimization could be performed on the PSO-NN to find the best set of parameters for developing mobile robot behaviors in this environment.

References

- [1] - Kennedy, J.; Eberhart, R.; Particle swarm optimization, Neural Networks, 1995. Proceedings., IEEE International Conference on Volume 4, 27 Nov.-1 Dec. 1995 Page(s):1942 - 1948 vol.4
- [2] - Chunkai Zhang; Huihe Shao; Yu Li; Particle swarm optimisation for evolving artificial neural network, Systems, Man, and Cybernetics, 2000 IEEE International Conference on Volume 4, 8-11 Oct. 2000 Page(s):2487 - 2490 vol.4
- [3] – R.A. Brooks. A robust layered control system for a mobile robot. IEEE journal of robotics and automation, pages 14-23, 1986.
- [4] - Grimaldi, E.A.; Grimaccia, F.; Mussetta, M.; Zich, R.E.; PSO as an effective learning algorithm for neural network applications Computational Electromagnetics and Its Applications, 2004. Proceedings. ICCEA 2004. 2004 3rd International Conference on 1-4 Nov. 2004 Page(s):557 – 560
- [5] - W-P. Lee, J. Hallam & H.H. Lund, Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots, *Proceedings of IEEE 4th International Conference on Evolutionary Computation*, IEEE Press, 1997.
- [6] - I. Sprinkhuizen-Kuyper, E.O. Postma & R. Kortmann, Evolutionary Learning of a Robot Controller: Effect of Neural Network Topology, *Proceedings of the Tenth Belgium-Dutch Conference on Machine Learning (BENELEARN'01)*, A. Feelers (Ed), 55-60, 2001.
- [7] - D. Montana & L. Davis, Training feedforward neural networks using genetic algorithms, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 762-767, Morgan Kaufman, California, 1989.
- [8] - I.G. Sprinkhuizen-Kuyper, Artificial Evolution of Box-Pushing Behavior, *Proceedings of the Twelfth Belgium-Netherlands Artificial Intelligence Conference*, A. van den Bosch & H. Weigand (Eds), 275-282, 2000.
- [9] - K. Shibata, M. Lida, Acquisition of box pushing by direct-vision-based reinforcement learning, *SICE 2003 Annual Conference*, 2322-2327, 2003.
- [10] - S. Doctor, G.K Venayagamoorthy, V.G. Gudise, Optimal PSO for Collective Robotic Search Applications, *Congress on Evolutionary Computation, 2004*, 1390-1395 Vol. 2, 2004

Unfinished Tasks

Though the method presented was not able to run on actual hardware, all other tasks have been accomplished satisfactorily. Future work may include hardware-based evaluations of this method.

Appendix A – Code Listing

MightySim.cpp:

```
// MightySim.cpp : Defines the entry point for the console application.
//

#include "WheelBot.h"
#include "FloatWorld.h"
#include "Utils.h"
#include "PSO.h"

int main(int argc, char* argv[])
{
    FILE* dfile;
    FILE* parmfile;
    int i,j;
    fregion tRegion;
    srand(time(0));
    list<fregion>::iterator h;
    char filename[100];
    char runnum[3];

    if(argc < 2)
        sprintf(runnum, "0");
    else
        sprintf(runnum, "%s", argv[1]);

    for(j=0; j<3; j++)
    {
        sprintf(filename, "run%d.txt", runnum, j);
        dfile = fopen(filename, "w");

        psonet *MightyPSO = new psonet(10, j);

        MightyPSO->cone = 1.2;
        MightyPSO->ctwo = 1.2;
        MightyPSO->w = 0.8;

        MightyPSO->EvalPopulation();
        MightyPSO->GetBest(dfile);

        for(i=0; i<1; i++)
        {
            MightyPSO->MovePopulation();
            MightyPSO->EvalPopulation();
            printf("Gen: %d ", i);
            MightyPSO->GetBest(dfile);
        }
        sprintf(filename, "r%sbesthist.txt", runnum, j);
        MightyPSO->WriteBest(filename);
        delete MightyPSO;
        fclose(dfile);
    }
}
```

```
        return 0;
    }
```

FloatWorld.h:

```
#ifndef FLOATWORLD_H
#define FLOATWORLD_H

#include "Utils.h"
#include <list>

#define PI 3.14159265

using namespace std;

enum regtype{ON, OFF, CHARGE};

class fregion
{
public:
    fregion(){};
    ~fregion(){};

    bool InRegion(fpoint2d pt);

    fpoint2d lowleft;
    fpoint2d topright;

    regtype cell;

    bool hit;
};

class FloatWorld
{
public:
    FloatWorld(double maxx, double maxy);
    ~FloatWorld(){};

    void StartAt(fpoint2d here);
    fregion Move(double heading, double distance);
    bool HitBorder(fpoint2d pt);
    void WriteHistory(char * filename);

    fpoint2d curpos;
    fpoint2d deltapos;

    double max_x;
    double max_y;

    list<fregion> obstacles;
    list<fregion> charger;
    list<fregion> open;

    list<fpoint2d> ephistory;
};
```

```
#endif
```

FloatWorld.cpp:

```
#include "FloatWorld.h"
#include <math.h>
#include <stdio.h>

bool fregion::InRegion(fpoint2d pt)
{
    if(pt.x >= lowleft.x && pt.y >= lowleft.y && pt.x <= topright.x &&
    pt.y <= topright.y)
        return true;
    else
        return false;
}

FloatWorld::FloatWorld(double maxx, double maxy)
{
    max_x = maxx;
    max_y = maxy;

    deltapos.x = 0;
    deltapos.y = 0;
}

void FloatWorld::StartAt(fpoint2d here)
{
    list<fregion>::iterator i;
    list<fpoint2d>::iterator j;

    if(here.x > 0 && here.x < max_x && here.y > 0 && here.y < max_y)
        curpos = here;

    for(i=open.begin(); i!=open.end(); i++)
        i->hit = false;

    ephistory.clear();

    return;
}

fregion FloatWorld::Move(double heading, double distance)
{
    list<fregion>::iterator i;
    fregion tRegion;
    fpoint2d nextpos;
    fpoint2d oldpos = curpos;

    nextpos.x = curpos.x + distance*cos(heading);
    nextpos.y = curpos.y + distance*sin(heading);

    // Check World Bounds
    if(nextpos.x > max_x)
        nextpos.x = max_x;
```

```

if(nextpos.x < 0)
    nextpos.x = 0;

if(nextpos.y > max_y)
    nextpos.y = max_y;

if(nextpos.y < 0)
    nextpos.y = 0;

for(i=open.begin(); i!=open.end(); i++)
{
    if(i->InRegion(nextpos))
    {
        i->hit = true;
        tRegion = *i;
    }
}

for(i=charger.begin(); i!=charger.end(); i++)
{
    if(i->InRegion(nextpos))
    {
        tRegion = *i;
    }
}

for(i=obstacles.begin(); i!=obstacles.end(); i++)
{
    if(i->InRegion(nextpos))
    {
        // Determine Collision Position

        if(curpos.x <= i->lowleft.x) // Hit on Left of Region
            nextpos.x = i->lowleft.x;

        if(curpos.x >= i->topright.x) // Hit on Right of Region
            nextpos.x = i->topright.x;

        if(curpos.y <= i->lowleft.y)
            nextpos.y = i->lowleft.y;

        if(curpos.y >= i->topright.y)
            nextpos.y = i->topright.y;

        tRegion = *i;
    }
}

curpos = nextpos;
deltapos.x = curpos.x - oldpos.x;
deltapos.y = curpos.y - oldpos.y;

ephistory.push_back(curpos);

return tRegion;
}

```

```

bool FloatWorld::HitBorder(fpoint2d pt)
{
    list<fregion>::iterator i;

    // Check World Bounds
    if(pt.x > max_x)
        return true;

    if(pt.x < 0)
        return true;

    if(pt.y > max_y)
        return true;

    if(pt.y < 0)
        return true;

    for(i=obstacles.begin(); i!=obstacles.end(); i++)
    {
        if(i->InRegion(pt))
            return true;
    }

    return false;
}

void FloatWorld::WriteHistory(char *filename)
{
    list<fpoint2d>::iterator i;
    FILE* ofile = fopen(filename, "w");
    double curx, cury;

    for(i=ephistory.begin(); i!=ephistory.end(); i++)
    {
        curx = i->x;
        cury = i->y;
        fprintf(ofile, "%f\t%f\n", curx, cury);
    }
    fclose(ofile);
    return;
}

```

Utils.h:

```

#ifndef UTILS_H
#define UTILS_H

#include <stdlib.h>
#include <time.h>

class ipoint2d
{
public:
    ipoint2d(){x=0; y=0;};
    ipoint2d(int ix, int iy){x=ix; y=iy;};
    int x;

```

```

        int y;
};

class fpoint2d
{
public:
    fpoint2d(){x=0; y=0;};
    fpoint2d(double ix, double iy){x = ix; y = iy;};
    double x;
    double y;
};

int GetRand(int low, int high);
double GetRand(double low, double high);

#endif

```

Utils.cpp:

```

#include "Utils.h"
#include <stdlib.h>

int GetRand(int low, int high)
{
    return (rand()%(high-low))+low;
}

double GetRand(double low, double high)
{
    if(low == 0 && high == 1)
    {
        return (double(rand())/double(RAND_MAX));
    }
    return (double(rand())/double(RAND_MAX))*(high-low) + low;
}

```

WheelBot.h:

```

#ifdef WHEELBOT_H
#define WHEELBOT_H
#include "FloatWorld.h"
#include "Utils.h"

class WheelBot
{
public:
    WheelBot(){};
    ~WheelBot(){};

    FloatWorld* myWorld;

    void Go(double leftw, double rightw);
    void UpdateSensors();

    double heading;
    fpoint2d curpos;
    fregion curregion;

```

```

// Physical Sensor Positions
double lsw_r;
double lsw_theta;

double rsw_r;
double rsw_theta;

bool lsw;
bool rsw;
bool irsense;

double wheelbase;
double wheeldia;
};

#endif

```

WheelBot.cpp:

```

#include "WheelBot.h"
#include <math.h>

// LeftW and RightW Values between 1 and -1;
void WheelBot::Go(double leftw, double rightw)
{
    double nhead = 0;
    double dist = 0;

    //double rarclen = rightw*PI*wheeldia; // Wheel Arc Lengths
    //double larclen = leftw*PI*wheeldia;

    // Max Step size: 1
    double rarclen = rightw;
    double larclen = leftw;
    double diff = rarclen-larclen; // Turn Arc Length, positive increases
    heading, CCW.

    if(rarclen > 0 && larclen > 0)
    {
        if(rarclen > larclen)
            dist = larclen;
        else
            dist = rarclen;
    }

    if(rarclen < 0 && larclen < 0)
    {
        if(rarclen < larclen)
            dist = rarclen;
        else
            dist = larclen;
    }

    nhead = diff / wheelbase;
}

```

```

heading += nhead;
if(heading < 0)
    heading = 2*PI - heading;
if(heading > 2*PI)
    heading = heading - 2*PI;

currregion = myWorld->Move(heading, dist);
curpos = myWorld->curpos;

return;
}

void WheelBot::UpdateSensors()
{
    fpoint2d sensepos;
    double r;
    double theta;

    if(currregion.cell == CHARGE)
        irsense = true;
    else
        irsense = false;

    // Left Switch
    r = lsw_r;
    theta = heading - lsw_theta;
    if(theta < 0)
        theta = 2*PI - theta;
    if(theta > 2*PI)
        theta = theta - 2*PI;

    sensepos.x = curpos.x + r*cos(theta);
    sensepos.y = curpos.y + r*sin(theta);

    lsw = myWorld->HitBorder(sensepos);

    // Right Switch
    r = rsw_r;
    theta = heading + rsw_theta;
    if(theta < 0)
        theta = 2*PI - theta;
    if(theta > 2*PI)
        theta = theta - 2*PI;

    sensepos.x = curpos.x + r*cos(theta);
    sensepos.y = curpos.y + r*sin(theta);

    rsw = myWorld->HitBorder(sensepos);

    return;
}

```

PSONN.h:

```

#ifdef PSONN_H
#define PSONN_H

```

```

#include "particle.h"
#include "bpann.h"
#include "WheelBot.h"
#include <list>

using namespace std;
class psonet
{
public:
    psonet(int pop_sz, int delay);
    ~psonet();

    void EvalPopulation();
    double GetBest(FILE* ofile);
    void MovePopulation();
    void WriteBest(char *filename);
    double w;
    double cone;
    double ctwo;
    WheelBot *Mousey;
private:
    int pop_size;
    int inputsize;
    int hiddensize;
    int outputsize;
    int delaylen;
    bpann *gbest;
    double gbestfit;
    list <particle*> population;
};

#endif

```

PSONN.cpp:

```

#include "PSONN.h"
#include "Utils.h"

psonet::psonet(int pop_sz, int delay)
{
    int i,j;
    fregion tRegion;
    fpoint2d tpoint;
    particle *tPart;
    pop_size = pop_sz;
    inputsize = 6;
    hiddensize = 14;
    outputsize = 2;
    delaylen = delay;
    gbest = new bpann(inputsize, hiddensize, outputsize, delaylen);
    gbestfit = 0;
    for(i=0; i<pop_size; i++)
    {
        tPart = new particle(inputsize, hiddensize, outputsize,
delaylen);
        population.push_back(tPart);
    }
}

```

```

    }

    Mousey = new WheelBot;
    Mousey->myWorld = new FloatWorld(100.0, 100.0);
    Mousey->wheelbase = 3;
    Mousey->wheeldia = 2.5;
    Mousey->lsw_r = 4;
    Mousey->lsw_theta = 0.698132;
    Mousey->rsw_r = 4;
    Mousey->rsw_theta = 0.698132;
    Mousey->heading = 0;
    Mousey->lsw = false;
    Mousey->rsw = false;

    tRegion.cell = ON;
    for(i=0; i<100; i+=5)
    {
        for(j=0; j<100; j+=5)
        {
            tRegion.lowleft.x = i;
            tRegion.lowleft.y = j;
            tRegion.topright.x = i+5;
            tRegion.topright.y = j+5;
            Mousey->myWorld->open.push_back(tRegion);
        }
    }

    tRegion.cell = OFF;
    tRegion.lowleft.x = -1.0;
    tRegion.lowleft.y = 30.0;
    tRegion.topright.x = 40.0;
    tRegion.topright.y = 40.0;
    Mousey->myWorld->obstacles.push_back(tRegion);

    tRegion.cell = OFF;
    tRegion.lowleft.x = 45.0;
    tRegion.lowleft.y = 65.0;
    tRegion.topright.x = 55.0;
    tRegion.topright.y = 101.0;
    Mousey->myWorld->obstacles.push_back(tRegion);

    tRegion.cell = CHARGE;
    tRegion.lowleft.x = 0.0;
    tRegion.lowleft.y = 0.0;
    tRegion.topright.x = 20.0;
    tRegion.topright.y = 20.0;
    Mousey->myWorld->charger.push_back(tRegion);

    Mousey->myWorld->StartAt(fpoint2d(10.0, 10.0));
}

psonet::~~psonet()
{
    list<particle>::iterator i;
    while(!population.empty())
    {

```

```

        delete population.front();
        population.pop_front();
    }
    delete gbest;
    delete Mousey->myWorld;
    delete Mousey;
}

void psonet::EvalPopulation()
{
    list <particle*>::iterator i;
    list<fregion>::iterator h;
    particle* tPart;
    int k,j;
    int Hits;
    int bonus;
    double in[6];
    double out[2];
    int tStart, tEnd;

    int tick=0;
    for(i=population.begin(); i!=population.end(); i++)
    {
        tPart = *i;
        printf("Tick: %d ", tick);
        tick++;
        Mousey->myWorld->StartAt(fpoint2d(10.0, 10.0));
        tStart = time(0);
        bonus = 0;
        for(j=0; j<5000; j++) // Area Coverage Phase
        {
            in[0] = -1;
            in[1] = Mousey->myWorld->deltapos.x;
            in[2] = Mousey->myWorld->deltapos.y;

            if(Mousey->lsw)
                in[3] = 1;
            else
                in[3] = -1;

            if(Mousey->rsw)
                in[4] = 1;
            else
                in[4] = -1;

            tPart->net->ffd(in, out);
            Mousey->Go((2*out[0])-1.0, (2*out[1])-1.0);
            Mousey->UpdateSensors();

            Hits = 0;
            for(h=Mousey->myWorld->open.begin(); h!=Mousey->myWorld->open.end(); h++)
            {
                if(h->hit)
                    Hits++;
            }
        }
    }
}

```

```

    if(Hits == 100)
    {
        bonus = 5000-j;
        break;
    }
}

    Hits += bonus; // Bonus for finishing early.

bonus = 0;
for(j=0; j<5000; j++) // Charger Hunt
{
    in[0] = 1;
    in[1] = Mousey->myWorld->deltapos.x;
    in[2] = Mousey->myWorld->deltapos.y;

    if(Mousey->lsw)
        in[3] = 1;
    else
        in[3] = -1;

    if(Mousey->rsw)
        in[4] = 1;
    else
        in[4] = -1;

    tPart->net->ffd(in, out);
    Mousey->Go((2*out[0])-1.0, (2*out[1])-1.0);
    Mousey->UpdateSensors();

    if(Mousey->curregion.cell == CHARGE)
    {
        bonus = 5000-j;
        break;
    }
}

    tPart->fitness = Hits + bonus;

    if(tPart->fitness > tPart->pbestfit)
    {
        *(tPart->pbest) = *(tPart->net);
        tPart->pbestfit = tPart->fitness;
    }
tEnd = time(0);
printf("Tock: %d\tFit:%f\n", tEnd-tStart, tPart->fitness);
}

    return;
}

void psonet::WriteBest(char *filename)
{
    list<fregion>::iterator h;
    int k,j;
    int Hits;
    double in[6];

```

```

double out[2];

Mousey->myWorld->StartAt(fpoint2d(10.0, 10.0));

for(j=0; j<5000; j++) // Area Coverage Phase
{
    in[0] = -1;
    in[1] = Mousey->myWorld->deltapos.x;
    in[2] = Mousey->myWorld->deltapos.y;

    if(Mousey->lsw)
        in[3] = 1;
    else
        in[3] = -1;

    if(Mousey->rsw)
        in[4] = 1;
    else
        in[4] = -1;

        gbest->ffd(in, out);

    Mousey->Go((2*out[0])-1.0, (2*out[1])-1.0);
    Mousey->UpdateSensors();
    Hits = 0;
    for(h=Mousey->myWorld->open.begin(); h!=Mousey->myWorld-
>open.end(); h++)
    {
        if(h->hit)
            Hits++;
    }

    if(Hits == 100)
        break;
}

for(j=0; j<5000; j++) // Charger Hunt
{
    in[0] = 1;
    in[1] = Mousey->myWorld->deltapos.x;
    in[2] = Mousey->myWorld->deltapos.y;

    if(Mousey->lsw)
        in[3] = 1;
    else
        in[3] = -1;

    if(Mousey->rsw)
        in[4] = 1;
    else
        in[4] = -1;

        gbest->ffd(in, out);

    Mousey->Go((2*out[0])-1.0, (2*out[1])-1.0);
    Mousey->UpdateSensors();
}

```

```

        if(Mouseey->curregion.cell == CHARGE)
            break;
    }

    Mousey->myWorld->WriteHistory(filename);

    return;
}

double psonet::GetBest(FILE* ofile)
{
    list<particle*>::iterator i;
    particle *tPart;
    for(i=population.begin(); i!=population.end(); i++)
    {
        tPart = *i;
        if(tPart->fitness > gbestfit)
        {
            *gbest = *(tPart->net);
            gbestfit = tPart->fitness;
        }
    }

    for(i=population.begin(); i!=population.end(); i++)
    {
        tPart = *i;
        tPart->gbest = gbest;
    }

    printf("Best Fit: %f\n", gbestfit);

    fprintf(ofile, "%f\n", gbestfit);
    return gbestfit;
}

void psonet::MovePopulation()
{
    list<particle*>::iterator i;
    particle *tPart;
    int j,k;

    for(i=population.begin(); i!=population.end(); i++)
    {
        tPart = *i;
        for(j=0; j<tPart->net->inputsizesize; j++)
        {
            for(k=0; k<tPart->net->hiddensize; k++)
            {
                tPart->net->dwih[j][k] = w*(tPart->net->dwih[j][k])
                +ccone*GetRand(0.0,1.0)*((tPart->pbest->wih[j][k])-(tPart->net->wih[j][k]))
                +cttwo*GetRand(0.0,1.0)*((gbest->wih[j][k])-(tPart->net->wih[j][k]));
                tPart->net->wih[j][k] += tPart->net->dwih[j][k];
            }
        }
    }
}

```

```

        for(j=0; j<tPart->net->hiddensize; j++)
        {
            for(k=0; k<tPart->net->outputsize; k++)
            {
                tPart->net->dwho[j][k] = w*(tPart->net->dwho[j][k])+cone*GetRand(0.0,1.0)*((tPart->pbest->who[j][k])-(tPart->net->who[j][k]))+ctwo*GetRand(0.0,1.0)*((gbest->who[j][k])-(tPart->net->who[j][k]));
                tPart->net->who[j][k] += tPart->net->dwho[j][k];
            }
        }
    }

    return;
}

```

particle.h:

```

#ifdef PARTICLE_H
#define PARTICLE_H

#include "bpann.h"

class particle
{
public:
    particle(int in_sz, int hid_sz, int out_sz, int delay);
    ~particle();

    bpann *net;
    bpann *pbest;
    bpann *gbest;
    double fitness;
    double pbestfit;

private:
    int inputsize;
    int hiddensize;
    int outputsize;
    int delaylen;
};

#endif

```

particle.cpp:

```

#include "particle.h"

particle::particle(int in_sz, int hid_sz, int out_sz, int delay)
{
    inputsize = in_sz;
    hiddensize = hid_sz;
    outputsize = out_sz;
    delaylen = delay;
}

```

```

        net = new bpann(inputsize, hiddensize, outputsize, delaylen);
        pbest = new bpann(inputsize, hiddensize, outputsize, delaylen);
        gbest = 0;
        fitness = 0;
        pbestfit = 0;
    }

particle::~~particle()
{
    delete net;
    delete pbest;
}

```

bpann.h:

```

#ifndef BPANN_H
#define BPANN_H

#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
class bpann {
public:
    bpann();
    bpann(int insize, int hidsize, int outsize, int delay);
    ~bpann();
    bpann& operator = (bpann &param);
    void ffd(double* in, double* out);
    double train(double* in, double *out, double* target);
    bool savewts(char *filename);
    bool loadwts(char *filename);
    double mu; // learning rate
    double alpha; // momentum
    int inputsize;
    int hiddensize;
    int outputsize;
    int delaylen;
    double** wih;
    double** who;
    double** dwih;
    double** dwho;
private:
    void init();
    double* input;
    double* output;
    double* hact;
    double* herr;
    double* oerr;
};

double sigm(double x);

#endif

```

bpann.cpp:

```
#include "bpann.h"

#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

double sigm(double x)
{
    return (1/(1+exp(-1*x)));
}

bpann::bpann()
{
}

bpann::bpann(int insize, int hidsize, int outsize, int delay)
{
    //inputsize = insize+1;
    //hidsize = hidsize+1;
    if(delay > 0)
    {
        inputsize = insize + delay*outsize;
        hidsize = hidsize;
        outsize = outsize;
        delaylen = delay;
    }
    else
    {
        inputsize = insize;
        hidsize = hidsize;
        outsize = outsize;
        delaylen = 0;
    }
    mu = 0.9;
    alpha = 0.5;

    init();
}

void bpann::init()
{
    int i, j;

    input = new double[inputsize];
    //input[inputsize-1] = 1;
    wih = new double*[inputsize];
    dwih = new double*[inputsize];
    for(i=0; i<inputsize; i++)
    {
        wih[i] = new double[hidsize];
        dwih[i] = new double[hidsize];
        for(j=0; j<hidsize; j++)
```

```

        {
            wih[i][j] = double(rand())/double(RAND_MAX)-0.5;
            dwih[i][j] = 0;
        }
    }

    who = new double*[hiddensize];
    dwho = new double*[hiddensize];
    hact = new double[hiddensize];
    herr = new double[hiddensize];
    for(i=0; i<hiddensize; i++)
    {
        who[i] = new double[outputsize];
        dwho[i] = new double[outputsize];
        for(j=0; j<outputsize; j++)
        {
            who[i][j] = double(rand())/double(RAND_MAX)-0.5;
            dwho[i][j] = 0;
        }
        hact[i] = 0;
        herr[i] = 0;
    }

    output = new double[outputsize];
    oerr = new double[outputsize];

    for(i=0; i<outputsize; i++)
    {
        output[i] = 0;
        oerr[i] = 0;
    }

    //hact[hiddensize-1] = 1;
    //printf("BPANN Initialized.\n");
}

bpann::~bpann()
{
    int i;
    delete [] input;
    delete [] output;
    delete [] hact;
    delete [] oerr;
    delete [] herr;

    for(i=hiddensize-1; i>=0; i--)
    {
        delete [] who[i];
        delete [] dwho[i];
    }
    delete [] who;
    delete [] dwho;

    for(i=inputsize-1; i>=0; i--)
    {
        delete [] wih[i];
    }
}

```

```

        delete [] dwih[i];
    }
    delete [] wih;
    delete [] dwih;
}

void bpann::ffd(double* in, double *out)
{
    int i, j;

    int in_sz = inputsize - (delaylen*outputsize);

    if(delaylen > 1)
    {
        for(i=delaylen; i>1; i--)
        {
            for(j=0; j<outputsize; j++)
                input[(j*i)+in_sz] = input[(j*(i-1))+in_sz];
        }
    }

    if(delaylen > 0)
    {
        for(i=0; i<outputsize; i++)
            input[i+in_sz] = output[i];
    }

    for(i=0; i<in_sz; i++)
    {
        input[i] = in[i];
    }

    for(i=0; i<hiddensize; i++)
    {
        hact[i] = 0;
        for(j=0; j<inputsize; j++)
        {
            hact[i] = hact[i] + (wih[j][i] * input[j]);
        }
        hact[i] = sigm(hact[i]);
    }
    //hact[hiddensize-1] = 1;

    for(i=0; i<outputsize; i++)
    {
        out[i] = 0;
        for(j=0; j<hiddensize; j++)
        {
            out[i] = out[i] + (who[j][i] * hact[j]);
        }
        output[i] = sigm(out[i]);
        out[i] = output[i];
    }
    return;
}

double bpann::train(double *in, double *out, double *target)

```

```

{
double mse = 0;
double odelt;

int i, j;

ffd(in, out);

for(i=0; i<outputsize; i++)
{
    oerr[i] = target[i] - out[i];
}

for(i=0; i<outputsize; i++)
{
    oerr[i] = (oerr[i] * (1-sigm(out[i])) * sigm(out[i]));

    for(j=0; j<hiddensize; j++)
    {
        odelt = dwho[j][i];
        dwho[j][i] = hact[j] *oerr[i]*mu;
        who[j][i] = who[j][i] + alpha*odelt + dwho[j][i];
    }
}

for(i=0; i<hiddensize; i++)
{
    herr[i] = 0;
    for(j=0; j<outputsize; j++)
    {
        herr[i] = herr[i]+ who[i][j]*oerr[j];
    }
}

for(i=0; i<hiddensize; i++)
{
    herr[i] = (herr[i] * (1-sigm(hact[i])) * sigm(hact[i]) );

    for(j=0; j<inputsize; j++)
    {
        odelt = dwih[j][i];
        dwih[j][i] = in[j] *herr[i]*mu;
        wih[j][i] = wih[j][i] + alpha*odelt + dwih[j][i];
    }
}
ffd(in, out);

//oerr = target - out;
for(i=0; i<outputsize; i++)
{
    oerr[i] = target[i] - out[i];
    mse = mse+(oerr[i]*oerr[i]);
}

mse = mse/2;

return mse;

```

```
}  
  
bpann& bpann::operator = (bpann &param)  
{  
    int i,j;  
  
    for(i=0; i    {  
        for(j=0; j<hiddensize; j++)  
        {  
            wih[i][j] = param.wih[i][j];  
        }  
    }  
    for(i=0; i<hiddensize; i++)  
    {  
        for(j=0; j<outputsize; j++)  
        {  
            who[i][j] = param.who[i][j];  
        }  
    }  
    return *this;  
}
```

Appendix B – Program User’s Manual

The above code, broken up into the listed files, can be compiled and executed under Windows or Linux operating systems. Once compiled, the program is executed by calling the executable, here “mightysim” with the run # appended to it. This functionality is included as a method for keeping track of multiple runs of the program. The run # is used to construct filenames for data logging. For instance, the call:

```
>mightysim 5
```

Will produce 6 output files: run5d0.txt run5d1.txt run5d2.txt r5besthist0.txt r5besthist1.txt r5besthist2.txt

The first three files are records of the best fitness for each generation of the PSO during that run, for each network architecture with 0, 1, or 2 delay lines, as denoted by the “d0” suffix. The “besthist” files contain the path data for the best individual from each run, as a set of 2d points in the world space. This allows the path to be plotted and observed for both debugging and analysis.

The program is intended to be executed as part of a script, similar to this example for the bash shell environment:

```
#!/bin/bash
X=0
while [ $X -le 100 ]
do
    ./mightysim $X > log$X.log
    X=$((X+1))
Done
```

This script runs the program 100 times, and places the program’s output in a log file corresponding to the execution iteration.