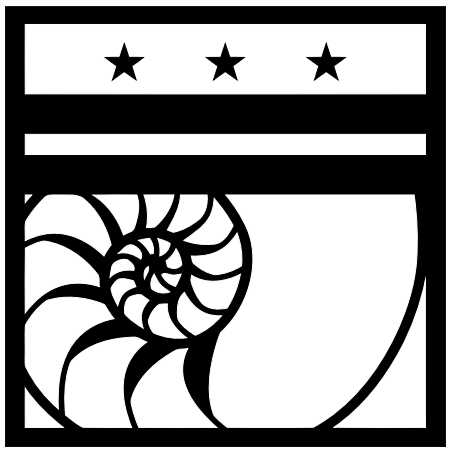
Stochastic Search and Neural Networks

Natural language and ARtificial intelligence Group



Problem Spaces

- Feature is a single point of data
- Feature vector is a collection of points of data (all relating to different things)
- Problem space is the space that spans all possible combinations of features

Problem Space Example

- We have height, weight, eye color and gender
- Each item in the list above is a feature
- A sample feature vector would be:
  [6'3", 140lbs, blue, male]
- Problem space is the combination of all feature vectors (all heights, eye colors, etc.)

Fitness Landscape

- Also known as "Error Spaces"
- Adds error vector/point to the problem space
- Fitness landscape is the set of all feature vectors **and** error vectors

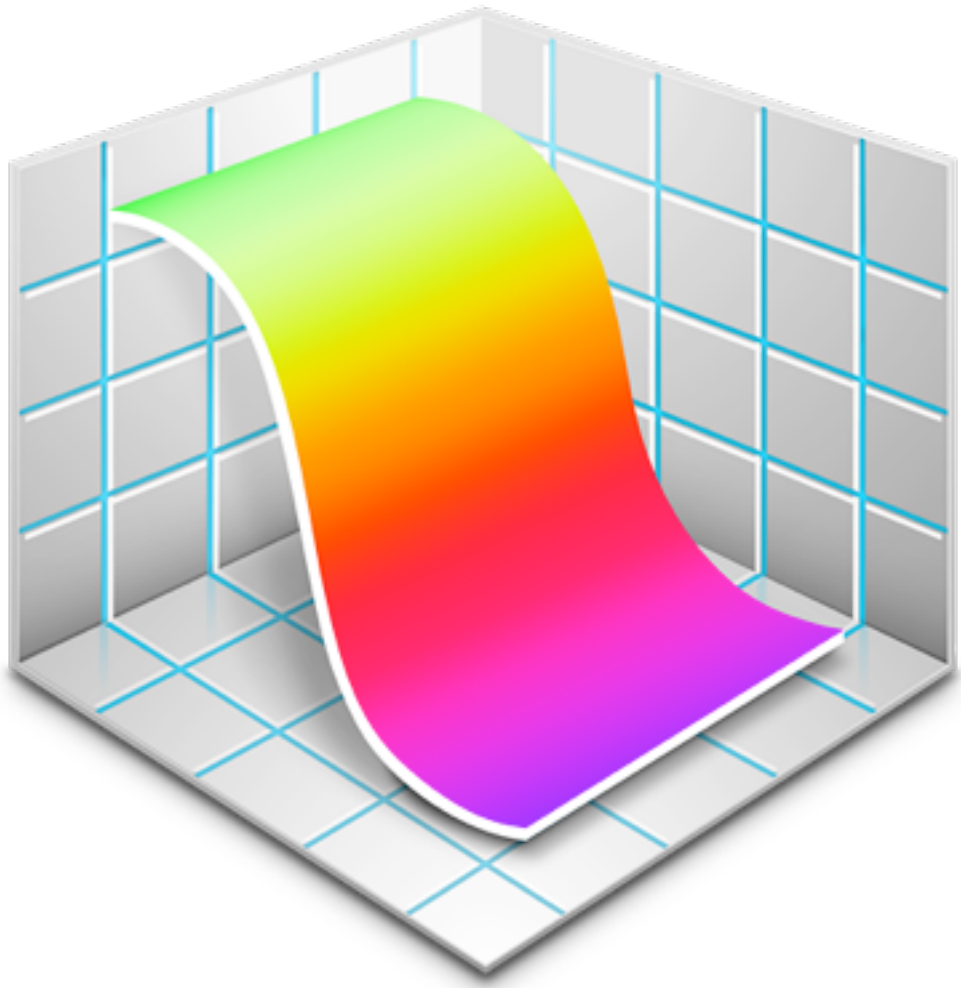
Fitness Landscape Example

- Last time out PSO was looking for the vector [5, 8] in the set of all real numbers
- Time to show the problem space and the fitness landscape

Optimization Problems

- AI can be defined as an optimization where the user want's to minimize error
- Two main types stochastic and gradient search
  - Gradient follows the shape of the fitness landscape
  - Stochastic wanders throughout the problem space

Grapher Time



Problem Space Search

Fitness Landscape Search

PSO Overview

- Initialize the particles
  - Set starting location, velocity and fitness
- While stop criteria hasn't been met
  - Check each particles fitness saving the best ever and the best at this time
  - Flock particles toward the best at this time and the best ever

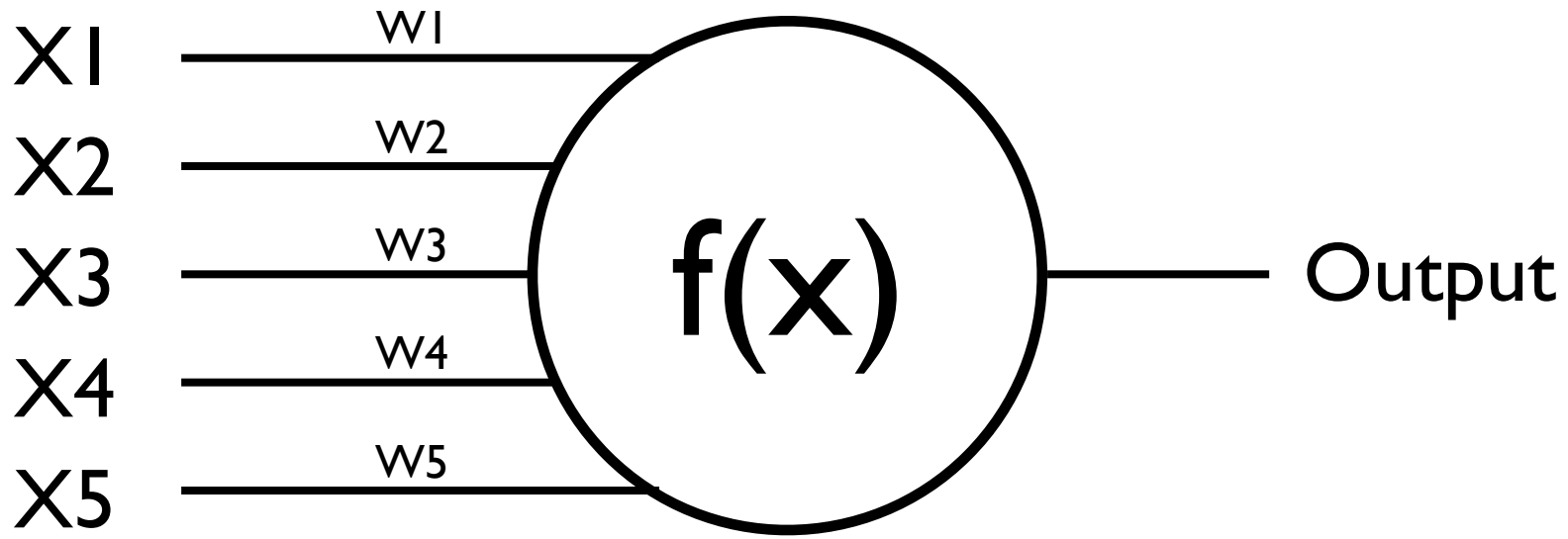
PSO Runtime

```
function run_pso (param, fitness_func)

    local particles = init_particles(num, dim, random)

    local pbest = particles[1]
    local gbest = particles[1]

    while iterations > 0 and pbest.f > success do
        pbest, gbest = calc_fitness(particles, fitness_func, pbest)
        update_particles(particles, pbest, gbest, pphi, gphi)
        print_particles(particles)
        iterations = iterations - 1
    end

    return pbest
end
```

PSO Updates

```
function update_position (particle)
    local position = {}
    for i=1, getn(particle.p) do
        tinsert(position, particle.p[i] + particle.v[i])
    end
    return position
end

function update_velocity (particle, pbest, gbest, pphi, gphi)
    local velocity = {}
    for i=1, getn(particle.v) do
        local prand = pphi*random()
        local grand = gphi*random()
        local pdiff = pbest.p[i] - particle.p[i]
        local gdiff = gbest.p[i] - particle.p[i]
        local vel = particle.v[i] + (prand * pdiff) + (grand * gdiff)
        tinsert(velocity, vel)
    end
    return velocity
end
```

Perceptrons



Neural Networks

- Networks of perceptrons or nodes
- Known for pattern matching abilities
- Applied to a wide variety of problems
- Eventually taken over by Kernel Methods
- Originally developed in the 40's

Neural Networks

- Category of AI techniques
- Describes a data structure more than an algorithm
- Can be trained in a number of ways:
  - Particle Swarm Optimizers
  - Backpropagation
  - Genetic Algorithms

Neural Networks

- Consist of a couple of directed graphs
  - A graph describing the topology (with or without cycles)
  - A graph describing the weights
- And a couple of arrays
  - Node outputs
  - Inputs from problem

Neural Networks

```
{
	topology = {
		{0,0,0,1,1,0,0,0,0},
		{0,0,0,1,1,0,0,0,0},
		{0,0,0,1,1,0,0,0,0},
		{0,0,0,0,0,1,1,1,1},
		{0,0,0,0,0,1,1,1,1},
		{0,0,0,0,0,0,0,0,0},
		{0,0,0,0,0,0,0,0,0},
		{0,0,0,0,0,0,0,0,0},
		{0,0,0,0,0,0,0,0,0},
	},
	weights = {
		{-71.74,-115.30,160.06,-545.12,-609.04,226.24,-123.90,-44.55,59.67},
		{-28.71,56.79,4.64,-241.84,36.33,2.90,22.42,963.65,-243.77},
		{34.48,-23.07,18.73,5.06,-49.50,20.98,-672.58,-32.82,-210.05},
		{204.72,126.70,68.57,95.91,232.92,290.05,-47.89,-33.47,-243.83},
		{97.01,27.22,-10.15,-40.74,77.39,-1.12,50.06,-36.96,-36.25},
		{99.34,19.34,-49.66,207.24,-27.07,-68.55,-88.64,67.78,-60.34},
		{-49.95,88.91,174.00,17.13,179.01,-24.76,-62.09,51.42,-642.90},
		{161.85,-38.85,13.80,-111.31,88.18,106.61,77.94,-125.00,-18.63},
		{-38.83,41.05,126.66,81.32,-13.21,330.66,157.38,-58.26,232.19},
	},
	inputs = 3,
	outputs = 4,
	node_outputs = {0,0,0,0,0,0,0,0,0}
}
```

NN Initialization

```
function init_neuralnet (topology, inputs, outputs)
    local neuralnet = {}
    neuralnet.topology = topology
    neuralnet.weights = init_weights(getn(topology))
    neuralnet.inputs = inputs
    neuralnet.outputs = outputs
    neuralnet.node_outputs = init_node_outputs(getn(topology))
    return neuralnet
end
```

```
function init_node_outputs(nodes)
    local outputs = {}
    for i=1, nodes do
        tinsert(outputs, 0)
    end
    return outputs
end
```

```
function init_weights(nodes)
    local weights = {}
    for i=1, nodes do
        local row = {}
        for j=1, nodes do
            tinsert(row, 1)
        end
        tinsert(weights, row)
    end
    return weights
end
```

NN Run (Chunk 1)

```
function run_net(net, inputs)
    -- set the proper inputs in the node outputs.
    -- i.e. artificially fire the input nodes based
    -- on input given.
    for i=1, getn(inputs) do
        if inputs[i] == 1 then
            net.node_outputs[i] = 1
        else
            net.node_outputs[i] = 0
        end
    end
```

NN Run (Chunk 2)

```
-- Fire all the nodes (in a naive feed forward fashion)
-- NOTE: Fix this later for recurrent neural networks.
-- NOTE: Naive means slow as balls.
for i=net.inputs+1, getn(net.topology) do
    local action_potential = 0
    for j=1, getn(net.topology) do
        if i ~= j then -- Beware there be hack-dragons in this code
            action_potential = action_potential +
                (net.topology[j][i] *
                net.weights[j][i] *
                net.node_outputs[j])
        end
    end
    if action_potential >= .95 then
        net.node_outputs[i] = 1
    else
        net.node_outputs[i] = 0
    end
end
```

NN Run (Chunk 3)

```
    -- Gather the outputs from the output nodes.
    local output = {}
    -- For loops are clumsy in Lua, since there is no test statement
    -- and defaults to testing for equality I need to add the 1 to get
    -- the proper amount of outputs.
    for i=getn(net.node_outputs), getn(net.node_outputs)-net.outputs+1, -1 do
        tinsert(output, net.node_outputs[i])
    end

    return output
end
```

PSO Neural Nets

- PSO's optimize some fitness function over a vector
- Lets make the fitness function construct a weight graph out of the particle's position vector and stuff it in a neural net
- Then we can run test cases on the net and see how bad it does

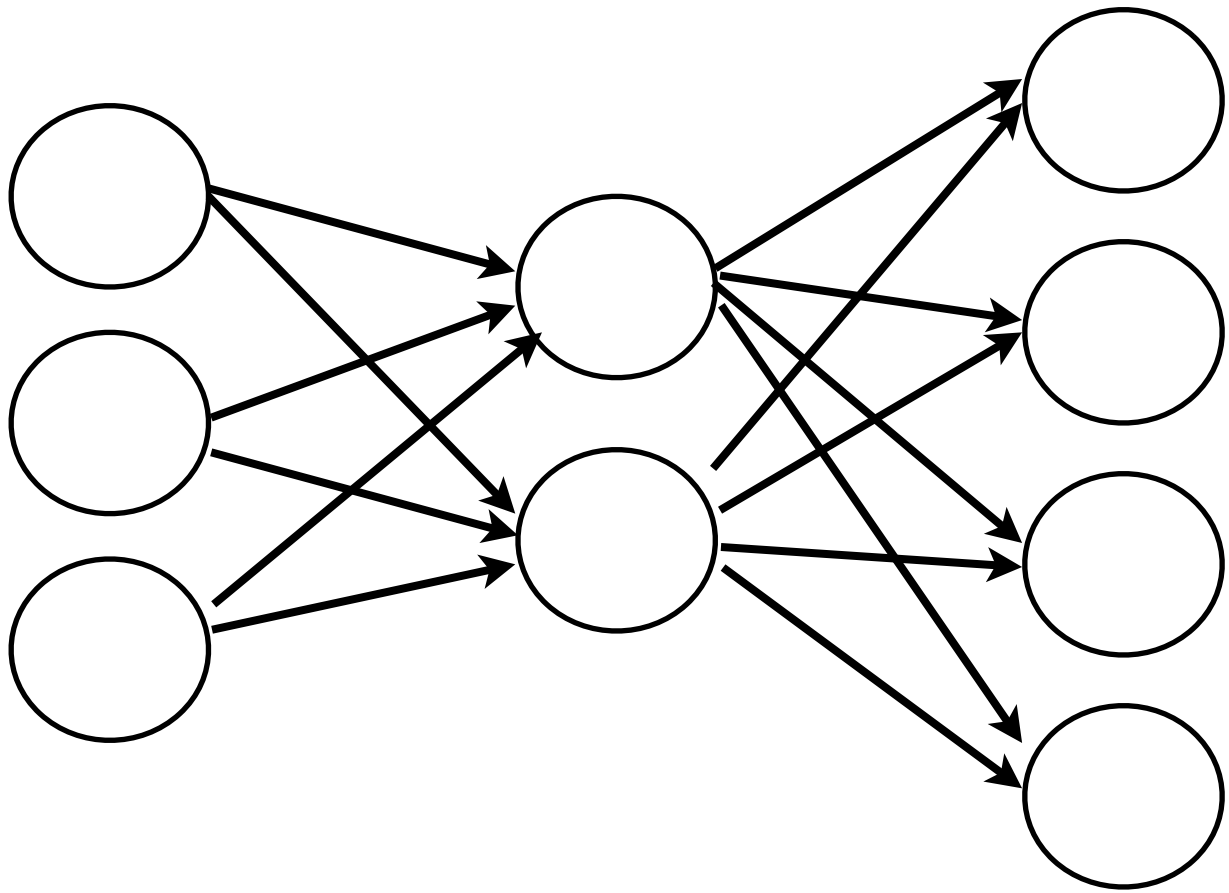
NN Training Task

- Given a binary number output 0-4 output a decimal number (or representation thereof)

4 = 1 0 0
3 = 0 1 1
2 = 0 1 0
1 = 0 0 1
0 = 0 0 0

1 0 0 0 = 4
0 1 0 0 = 3
0 0 1 0 = 2
0 0 0 1 = 1
0 0 0 0 = 0

Neural Net Topology



Fitness Function

- Create a Neural Net using weights defined by the current particle's position
- For each test case check the hamming distance from expected to actual
- Return the average hamming distance

Fitness Function (Chunk 1)

```
function fitness_func (particle)
    local topology = {
        {0, 0, 0, 1, 1, 0, 0, 0, 0},
        {0, 0, 0, 1, 1, 0, 0, 0, 0},
        {0, 0, 0, 1, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1, 1, 1, 1},
        {0, 0, 0, 0, 0, 1, 1, 1, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0}
    }
```

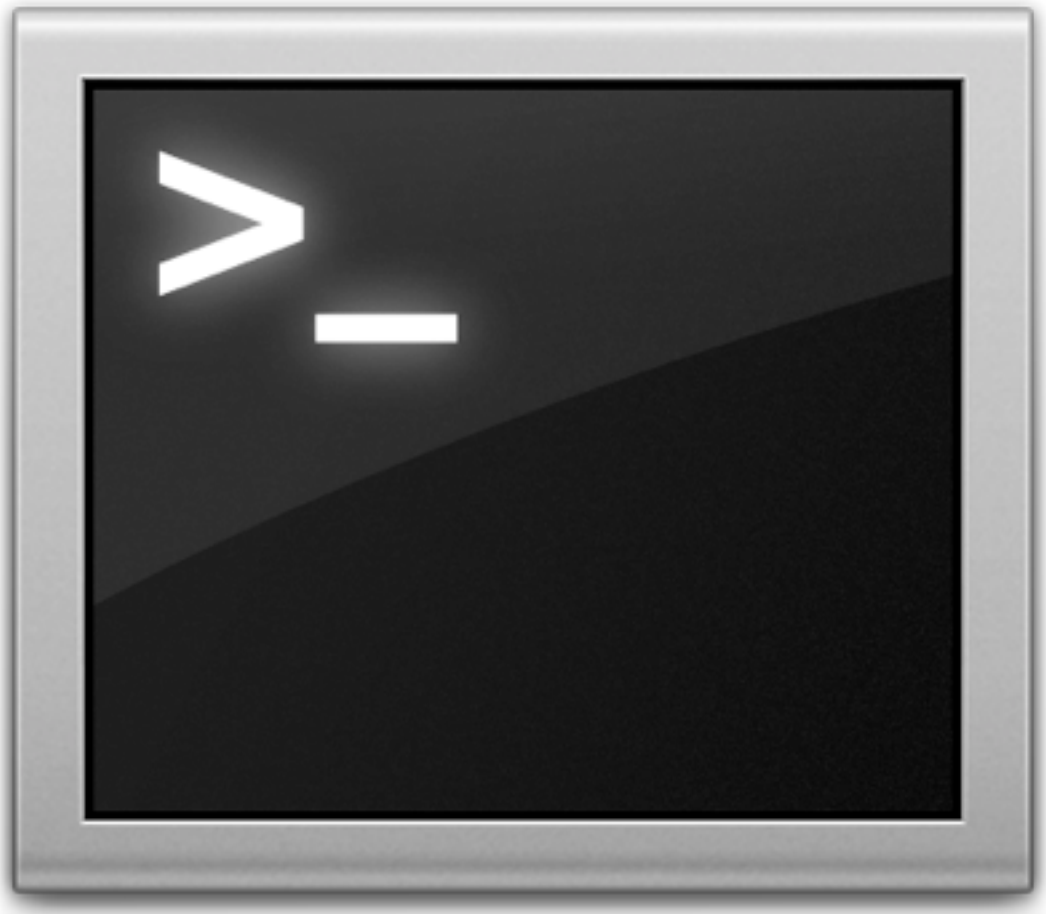
Fitness Function (Chunk 2)

```
local test_inputs = {
    {0, 0, 0},
    {0, 0, 1},
    {0, 1, 0},
    {0, 1, 1},
    {1, 0, 0},
}

local test_outputs = {
    {0, 0, 0, 0},
    {0, 0, 0, 1},
    {0, 0, 1, 0},
    {0, 1, 0, 0},
    {1, 0, 0, 0}
}
```

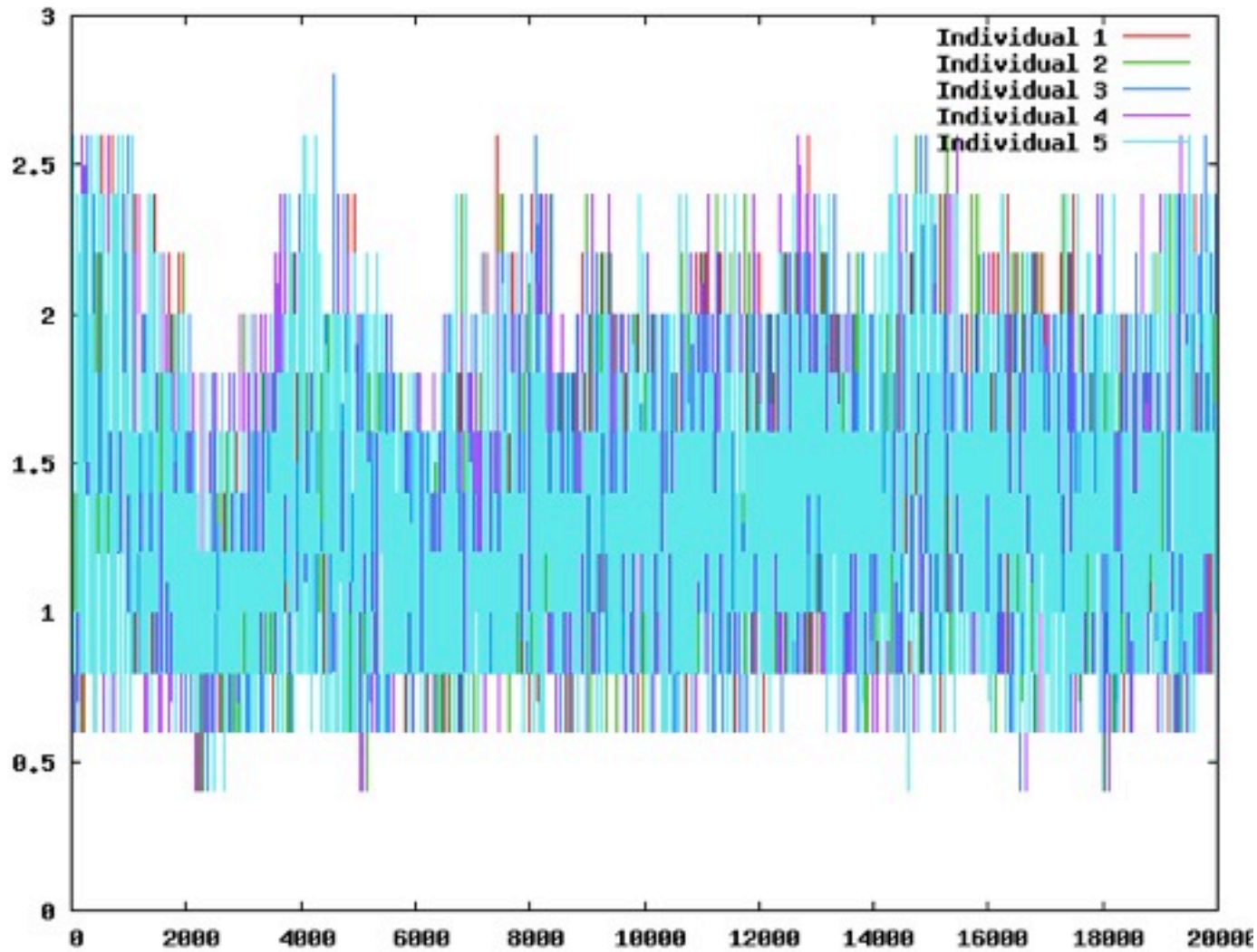
Fitness Function (Chunk 3)

```
    local nn = make_net_from_pso(topology, particle)

    local err = 0
    for i=1, getn(test_inputs) do
        local output = run_net(nn, test_inputs[i])
        for j=1, getn(output) do
            if output[j] ~= test_outputs[i][j] then
                err = err + 1
            end
        end
        reset_outputs(nn)
    end
    return err/getn(test_inputs)
end
```

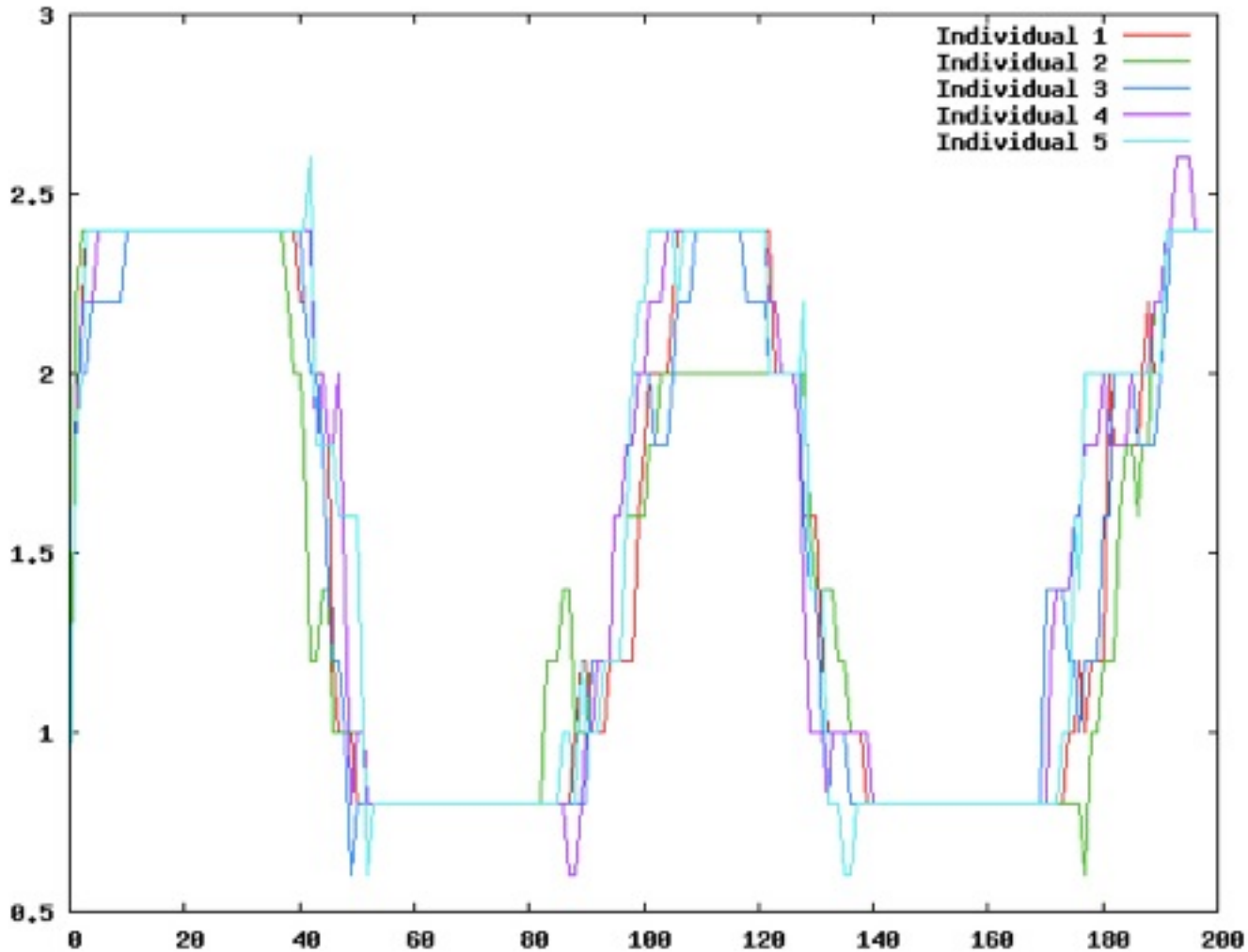
Lets see the code



Fitness over Iterations



Fitness over Less Time



Off The Cuff

- Supervised versus unsupervised learning
- Generalizability of results and some tricks
- Simulation versus the physical world
- Data versus smarts